

ADVANCE DATA STRUCTURES AND ALGORITHMS
(R20D5802)

DIGITAL NOTES
M.TECH I YEAR - I SEM (R20)
(2021-2022)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MALLAREDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution - UGC, Govt. of India)

(Recognized under 2(f) and 12 (B) of UGC ACT 1956)

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

Objectives:

The fundamental design, analysis, and implementation of basic data structures. Basic concepts in the specification and analysis of programs.

1. Principles for good program design, especially the uses of data abstraction. Significance of algorithms in the computer field
2. Various aspects of algorithm development Qualities of a good solution

TEXT BOOKS:**UNIT I**

Algorithms, Performance analysis- time complexity and space complexity, Asymptotic Notation- Big Oh, Omega and Theta notations, Complexity Analysis Examples. Data structures-Linear and non linear data structures, ADT concept, Linear List ADT, Array representation, Linked representation, Vector representation, singly linked lists -insertion, deletion, search operations, doubly linked lists-insertion, deletion operations, circular lists. Representation of single, two dimensional arrays, Sparse matrices and their representation.

UNIT II

Stack and Queue ADTs, array and linked list representations, infix to postfix conversion using stack, implementation of recursion, Circular queue-insertion and deletion, Dequeue ADT, array and linked list representations, Priority queue ADT, implementation using Heaps, Insertion into a Max Heap, Deletion from a Max Heap, java.util package-ArrayList, Linked List, Vector classes, Stacks and Queues in java.util, Iterators in java.util.

UNIT III

Searching–Linear and binary search methods, Hashing-Hash functions, Collision Resolution methods-Open Addressing, Chaining, Hashing in java.util-HashMap, HashSet, Hashtable. Sorting –Bubble sort, Insertion sort, Quick sort, Merge sort, Heap sort, Radix sort, comparison of sorting methods.

UNIT IV

Trees- Ordinary and Binary trees terminology, Properties of Binary trees, Binary tree ADT, representations, recursive and non recursive traversals, Java code for traversals, Threaded binary trees. Graphs- Graphs terminology, Graph ADT, representations, graph traversals/search methods-dfs and bfs, Java code for graph traversals, Applications of Graphs-Minimum cost spanning tree using Kruskal's algorithm, Dijkstra's algorithm for Single Source Shortest Path Problem.

UNIT V

Search trees- Binary search tree-Binary search tree ADT, insertion, deletion and searching operations, Balanced search trees, AVL trees-Definition and examples only, Red Black trees – Definition and examples only, B-Trees-definition, insertion and searching operations, Trees in java.util- TreeSet, Tree Map Classes, Tries(examples only),Comparison of Search trees. Text compression-Huffman coding and decoding, Pattern matching-KMP algorithm.

TEXT BOOKS:

1. Data structures, Algorithms and Applications in Java, S.Sahni, Universities Press.
2. Data structures and Algorithms in Java, Adam Drozdek, 3rd edition, Cengage Learning.
3. Data structures and Algorithm Analysis in Java, M.A.Weiss, 2nd edition,
4. Addison-Wesley (Pearson Education).

REFERENCE BOOKS:

1. Java for Programmers, Deitel and Deitel, Pearson education.
2. Data structures and Algorithms in Java, R.Lafore, Pearson education.
3. Java: The Complete Reference, 8th editon, Herbert Schildt, TMH.
4. Data structures and Algorithms in Java, M.T.Goodrich, R.Tomassia, 3rd edition, WileyIndia Edition.
5. Data structures and the Java Collection Frame work,W.J.Collins, Mc Graw Hill.
6. Classic Data structures in Java, T.Budd, Addison-Wesley (Pearson Education).
7. Data structures with Java, Ford and Topp, Pearson Education.
8. Data structures using Java, D.S.Malik and P.S.Nair, Cengage learning.
9. Data structures with Java, J.R.Hubbard and A.Huray, PHI Pvt. Ltd.
10. Data structures and Software Development in an Object-Oriented Domain, J.P.Tremblay and G.A.Cheston, Java edition, Pearson Education.

INDEX

| UNIT NO | TOPIC | PAGE NO |
|---------|---|---------|
| I | Algorithms | 6-8 |
| | Performance analysis- time complexity and space complexity | 6-8 |
| | Asymptotic Notation- Big Oh, Omega and Theta notations | 8-10 |
| | Complexity Analysis Examples | 10-15 |
| | Data structures-Linear and Non Linear Data Structures | 15-17 |
| | ADT concept | 15-17 |
| | Linear List ADT, Array representation, Linked representation, Vector representation | 17-22 |
| | Singly linked lists - insertion, deletion, search operations | 22-30 |
| | Doubly linked lists-insertion, deletion operations | 40-49 |
| | Circular lists - insertion, deletion, search operations | 30-40 |
| | Representation of single, two dimensional arrays | 50-51 |
| | Sparse matrices and their representation | 49-52 |
| II | Stack ADT Array and Linked list representations | 53-61 |
| | Queue ADT Array and Linked list representations | 67-77 |
| | Infix to Postfix conversion using stack, Implementation of Recursion | 62-67 |
| | Circular Queue- insertion and deletion | 84-89 |
| | Deque ADT Array and Linked list representations | 78-83 |
| | Priority Queue ADT | 89-93 |
| | Implementation using Heaps, Insertion into a Max Heap, Deletion from a Max Heap | 94-97 |
| | Java.util package-ArrayList, Linked List, Vector classes | 98-104 |
| | Stacks and Queues in java.util | 105-106 |
| | Iterators in java.util | 106-107 |
| III | Searching | 108-126 |
| | Linear and binary search methods | 108-113 |
| | Hashing-Hash functions | 114-116 |
| | Collision Resolution methods-Open Addressing, Chaining, Hashing in java.util-HashMap, HashSet, Hashtable. | 117-126 |
| | Sorting | 126-152 |
| | Bubble sort, Insertion sort, Quick sort, Merge sort, Heap sort, Radix sort, comparison of sorting methods | 126-152 |

| UNIT NO | TOPIC | PAGE NO |
|---------|--|---------|
| IV | Trees | 153-201 |
| | Ordinary and Binary trees terminology | 153-155 |
| | Properties of Binary trees | 155-156 |
| | Binary tree ADT, representations, Recursive and non recursive traversals, Java code for traversals | 156-165 |
| | Threaded binary trees | 166-167 |
| | Graphs- Graphs terminology, Graph ADT representations, graph traversals/search methods-dfs and bfs, Java code for graph traversals | 167-184 |
| | Applications of Graphs-Minimum cost spanning tree using Kruskal's algorithm | 185-195 |
| | Dijkstra's algorithm for Single Source Shortest Path Problem. | 196-201 |
| V | Search trees | 202-277 |
| | Binary search tree-Binary search tree ADT, insertion, deletion and searching operations | 202-213 |
| | Balanced search trees | 271-277 |
| | AVL trees-Definition and examples only | 213-225 |
| | Red Black trees – Definition and examples only | 237-243 |
| | B-Trees-definition, insertion and searching operations | 225-236 |
| | Trees in java.util- TreeSet, Tree Map Classes, Tries(examples only) | 261-271 |
| | Comparison of Search trees | 258-259 |
| | Text compression-Huffman coding and decoding | 244-250 |
| | Pattern matching-KMP algorithm | 251-258 |

UNIT -1

Basic concepts of Algorithm

Preliminaries of Algorithm:

An algorithm may be defined as a finite sequence of instructions each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

The algorithm word originated from the Arabic word “Algorism” which is linked to the name of the Arabic mathematician Al Khwarizmi. He is considered to be the first algorithm designer for adding numbers.

Structure and Properties of Algorithm:

An algorithm has the following structure

1. Input Step
2. Assignment Step
3. Decision Step
4. Repetitive Step
5. Output Step

1. **Finiteness:** An algorithm must terminate after a finite number of steps.
2. **Definiteness:** The steps of the algorithm must be precisely defined or unambiguously specified.
3. **Generality:** An algorithm must be generic enough to solve all problems of a particular class.
4. **Effectiveness:** the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation.
5. **Input-Output:** The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties.

1. **To save time (Time Complexity):** A program that runs faster is a better program.
2. **To save space (Space Complexity):** A program that saves space over a competing program is considerable desirable.

Efficiency of Algorithms

The performances of algorithms can be measured on the scales of **time** and **space**. The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity: The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.

Space Complexity: The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.

The time complexity of an algorithm can be computed either by an **empirical** or **theoretical** approach. The **empirical** or **posteriori testing** approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

Analyzing Algorithms:

Suppose M is an algorithm, and suppose n is the size of the input data. Clearly the complexity $f(n)$ of M increases as n increases. It is usually the rate of increase of $f(n)$ with some standard functions. The most common computing times are

$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$

Example:

Program Segment A

```
-----
x =x + 2;
-----
-
```

Program Segment B

```
-----
for k =1 to n
do
    x =x + 2;
end;
-----
```

Program Segment C

```
-----
for j =1 to n do
    for x = 1 to n
do
        x =x + 2;
    en
d
end-----
```

Total Frequency Count of Program Segment A

| Program Statements | Frequency Count |
|--------------------------------|-----------------|
| ----- $x = x + 2;$ ----- | 1 |
| Total Frequency Count | 1 |

Total Frequency Count of Program Segment B

| Program Statements | Frequency Count |
|---------------------------|-----------------|
| ----- for k =1 to n do | (n+1) |
| $x = x + 2;$ end; | n |
| ----- | n |
| Total Frequency Count | 3n+1 |

Total Frequency Count of Program Segment C

| Program Statements | Frequency Count |
|---|---|
| ----- for j =1 to n do for x = 1 to n do $x = x + 2;$ End end; | (n+1) $n(n+1)$ n_2 2 n N |
| Total Frequency Count | $3n^2 + 3n + 1$ |

The total frequency counts of the program segments A, B and C given by 1, $(3n+1)$ and $(3n^2 + 3n + 1)$ respectively are expressed as $O(1)$, $O(n)$ and $O(n^2)$. These are referred to as the time complexities of the program segments since they are indicative of the running times of the program segments. In a similar manner space complexities of a program can also be expressed in terms of mathematical notations, which is nothing but the amount of memory they require for their execution.

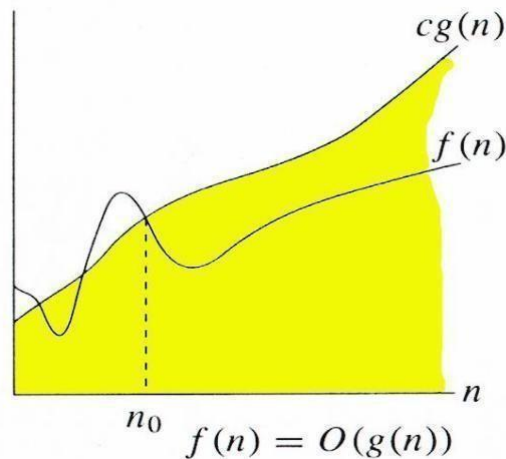
Asymptotic Notations:

It is often used to describe how the size of the input data affects an algorithm's usage of

computational resources. Running time of an algorithm is described as a function of input size n for large n .

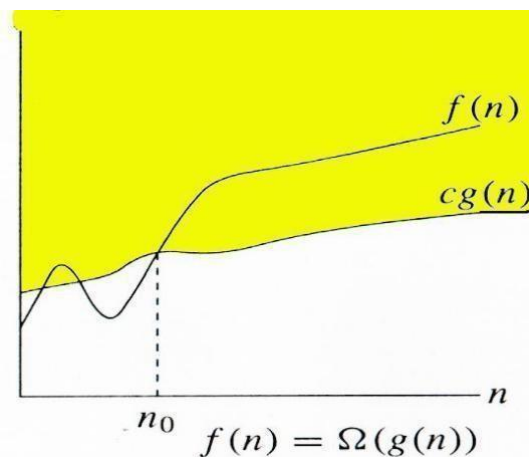
Big oh(O): Definition: $f(n) = O(g(n))$ (read as f of n is big oh of g of n) if there exist a positive integer n_0 and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$. Here $g(n)$ is the upper bound of the function $f(n)$.

| $f(n)$ | $g(n)$ | |
|-----------------------|--------|-----------------|
| $16n^3 + 45n^2 + 12n$ | n^3 | $f(n) = O(n^3)$ |
| $34n - 40$ | n | $f(n) = O(n)$ |
| 50 | 1 | $f(n) = O(1)$ |



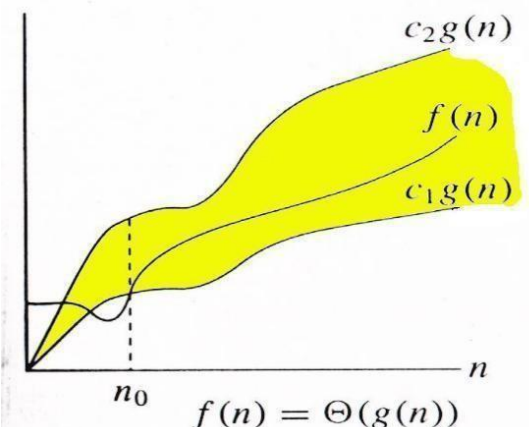
Omega(Ω): Definition: $f(n) = \Omega(g(n))$ (read as f of n is omega of g of n), if there exists a positive integer n_0 and a positive number c such that $|f(n)| \geq c|g(n)|$ for all $n \geq n_0$. Here $g(n)$ is the lower bound of the function $f(n)$.

| $f(n)$ | $g(n)$ | |
|--------------------|--------|----------------------|
| $16n^3 + 8n^2 + 2$ | N^3 | $f(n) = \Omega(n^3)$ |
| $24n + \dots$ | N | $f(n) = \Omega(n)$ |



Theta(Θ): Definition: $f(n) = \Theta(g(n))$ (read as f of n is theta of g of n), if there exists a positive integer n_0 and two positive constants c_1 and c_2 such that $c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)|$ for all $n \geq n_0$. The function $g(n)$ is both an upper bound and a lower bound for the function $f(n)$ for all values of $n, n \geq n_0$.

| $f(n)$ | $g(n)$ | |
|----------------------|--------|----------------------|
| $16n^3 + 30n^2 - 90$ | n^2 | $f(n) = \Theta(n^2)$ |



Little oh(o): Definition: $f(n) = O(g(n))$ (read as f of n is little oh of g of n), if $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$.

| $f(n)$ | $g(n)$ | |
|-----------|--------|--|
| $18n + 9$ | n^2 | $f(n) = o(n^2)$ since $f(n) = O(n^2)$ and $f(n) \neq \Omega(n^2)$ however $f(n) \neq O(n)$. |

Relations Between O, Ω , Θ :

Theorem : For any two functions $g(n)$ and $f(n)$,

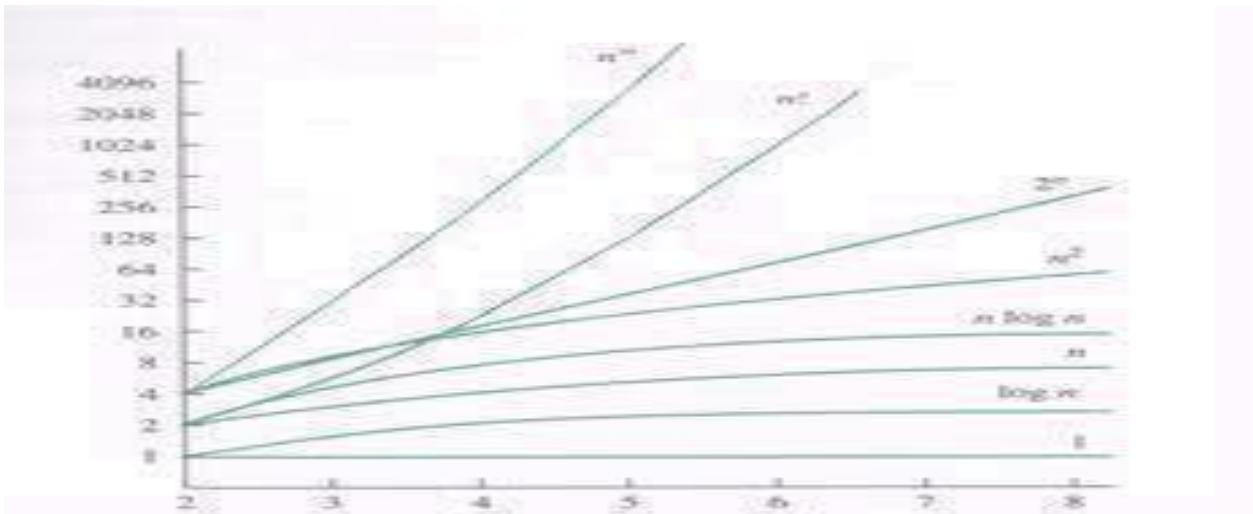
$$f(n) = \Theta(g(n)) \text{ iff}$$

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

Time Complexity:

| Complexity | Notation | Description |
|-------------|-------------|---|
| Constant | $O(1)$ | Constant number of operations, not depending on the input data size. |
| Logarithmic | $O(\log n)$ | Number of operations proportional of $\log(n)$ where n is the size of the input data. |
| Linear | $O(n)$ | Number of operations proportional to the input data size. |
| Quadratic | $O(n^2)$ | Number of operations proportional to the square of the size of the input data. |
| Cubic | $O(n^3)$ | Number of operations proportional to the cube of the size of the input data. |
| Exponential | $O(2^n)$ | Exponential number of operations, fast growing. |
| | $O(k^n)$ | |
| | $O(n!)$ | |

Time Complexities of various Algorithms:



Numerical Comparison of Different Algorithms:

| S.No. | $\log_2 n$ | n | $n \log_2 n$ | n^2 | n^3 | 2^n |
|-------|------------|-----|--------------|-------|-------|-------|
| 1. | 0 | 1 | 1 | 1 | 1 | 2 |
| 2. | 1 | 2 | 2 | 4 | 8 | 4 |
| 3. | 2 | 4 | 8 | 16 | 64 | 16 |
| 4. | 3 | 8 | 24 | 64 | 512 | 256 |
| 5. | 4 | 16 | 64 | 256 | 4096 | 65536 |

Reasons for analyzing algorithms:

- To predict the resources that the algorithm requires
 - Computational Time(CPU consumption).
 - Memory Space(RAM consumption).
 - Communication bandwidth consumption.
- To predict the running time of an algorithm
 - Total number of primitive operations executed.

Recursion Definition:

- Recursion is a technique that solves a problem by solving a smaller problem of the same type.
- A recursive function is a function invoking itself, either directly or indirectly.
- Recursion can be used as an alternative to iteration.
- Recursion is an important and powerful tool in problem solving and programming.
- Recursion is a programming technique that naturally implements the divide and conquer problem solving methodology.

Four criteria of a Recursive Solution:

- A recursive function calls itself.
- Each recursive call solves an identical, but smaller problem.
- A test for the **base case** enables the recursive calls to stop.

- There must be a case of the problem (known as **base case** or **stopping case**) that is handled differently from the other cases.
- In the **base case**, the recursive calls stop and the problem is solved directly.

4. Eventually, one of the smaller problems must be the base case.

Linear Search:

1. Read search element.
2. Call function linear search function by passing N value, array and search element.
3. If $a[i] == k$, return i value, else return -1, returned value is stored in pos.
4. If $pos == -1$ print element not found, else print $pos+1$ value.

Source Code:

(Recursive)

```
#include<stdio.h>
#include<conio.h>
void linear_search(int n,int a[20],int i,int k)
{
    if(i>=n
    )
    {
        printf("%d is not found",k);
        return;

    }
    if(a[i]==k)
    {
        printf("%d is found at
        %d",k,i+1); return;
    }
    else
        linear_search(n,a,i+1,k);
}
void main()
{
    int i,a[20],n,k;
    clrscr();
    printf("Enter no of
    elements:"); scanf("%d",&n);
    printf("Enter elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("Enter search
    element:"); scanf("%d",&k);
    linear_search(n,a,0,k);
    getch();
}
```

Input & Output:

```
Enter    no    of
elements:3    Enter
elements:1 2 3 Enter
search element:6 6
is not found
```

Enter no of elements:5
Enter elements:1 2 3 4
5 Enter search
element:3
3 is found at position 3

Time Complexity of Linear Search:

If input array is not sorted, then the solution is to use a sequential search.

Unsuccessful search: $O(N)$

Successful Search: Worst case: $O(N)$

Average case: $O(N/2)$

Binary Search:

1. Read search data.
2. Call binary_search function with values N, array, and data.
3. If low is less than high, making mid value as mean of low and high.
4. If $a[mid] == data$, make $flag = 1$ and break, else if data is less than $a[mid]$ make $high = mid - 1$, else $low = mid + 1$.
5. If $flag == 1$, print data found at $mid + 1$, else notfound.

Source Code:

(Recursive)

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
void binary_search(int a[20],int data,int low,int high)
```

```
{  
    int mid;  
    if(low<=high  
    )  
    {  
        mid=(low+high)/  
        2;  
        if(a[mid]==data)  
            printf("Data found at %d",mid+1);  
        else  
            if(a[mid]>data)  
                binary_search(a,data,low,mid-1);  
            else  
                binary_search(a,data,mid+1,high);  
    }  
}
```

```
void main()  
{
```

```
    int i,a[20],n,data;  
    clrscr();  
    printf("Enter no of  
elements:"); scanf("%d",&n);  
    printf("Enter elements:");  
    for(i=0;i<n;i++)  
        scanf("%d",&a[i]);  
    printf("Enter search  
element:");  
    scanf("%d",&data);  
    binary_search(a,data,0,n-1);  
    getch();  
}
```

}

Input & Output:

Enter no of elements:3
 Enter elements:1 2 3
 Enter search
 element:25 Not found

Enter no of
 elements:3 Enter
 elements:1 2 3 Enter
 search element:3
 Data found at 3

Time Complexity of Binary Search:

Time Complexity for binary search is $O(\log_2 N)$

Fibonacci Search:**Source Code:**

(Recursive)

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
void fib_search(int a[],int n,int search,int pos,int begin,int end)
```

```
{
    int fib[20]={0,1,1,2,3,5,8,13,21,34,55,89,144};
    if(end<=0)
    {
        printf("\nNot found");
        return;//data not found
    }
    else
    {
        pos=begin+fib[--end];
        if(a[pos]==search && pos<n)
        {
            printf("\n Found at
            %d",pos); return;//data
            found
        }
        if((pos>=n)||((search<a[pos]))
            fib_search(a,n,search,pos,begin,e
            nd);
        else
        {
            begin=pos+
            1; end--;
            fib_search(a,n,search,pos,begin,end);
        }
    }
}

void main()
{
    int
    n,i,a[20],search,pos=0,begin=0,k=0,end;
    int
    fib[20]={0,1,1,2,3,5,8,13,21,34,55,89,144}
    ;
    clrscr();
```



```
printf("Enter the  
n:");  
scanf("%d",&n);  
printf("Enter elements to array:");  
for(i=0;i<n;i++)  
    scanf("%d",&a[i]);
```

```

printf("Enter the search
element:"); scanf("%d",&search);
while(fib[k]<n)
{
    k++;
}
end=k;
printf("Max.no of passes :
%d",end);
fib_search(a,n,search,pos,begin,e
nd); getch();
}

```

Input & Output:

Enter the n:5

Enter elements to array:1 2 3 6

59 Enter the search element:56

Max no of passes required is : 5

Search element not found.....

Time Complexity of Fibonacci Search:

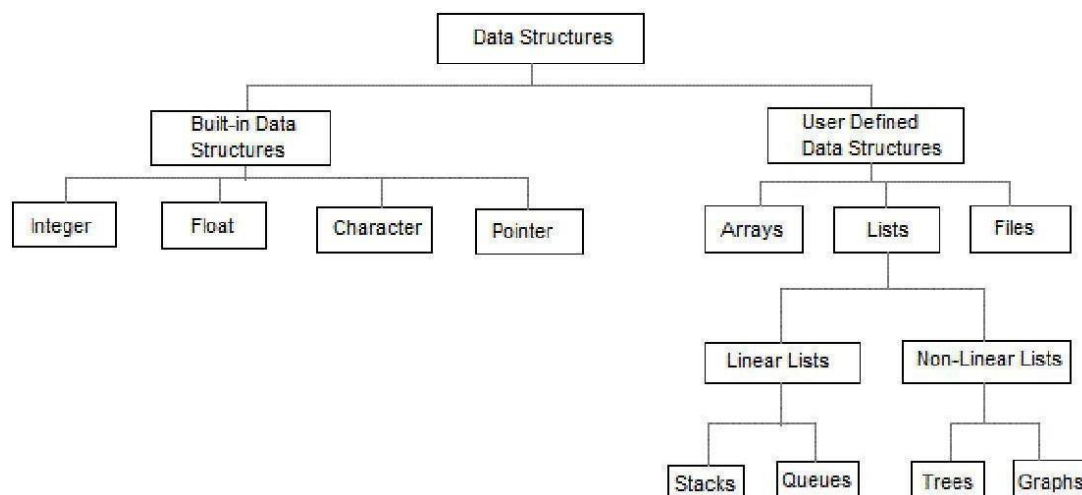
Time complexity for Fibonacci search is $O(\log_2 N)$

Data structure

A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways

Abstract Data Type

In computer science, an abstract data type (ADT) is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. When a class is used as a type, it is an abstract type that refers to a hidden representation. In this model an ADT is typically implemented as a class, and each instance of the ADT is usually a n object of that class. In ADT all the implementation details are hidden



1. Linear data structures are the data structures in which data is arranged in a list or in a sequence.
2. Non linear data structures are the data structures in which data may be arranged in a hierarchical manner

LINEAR DATA STRUCTURE

Stacks and Queues are both special-purpose lists, that restrict how the application can access data. This is done so that the structures can optimize themselves for speed. Both data structures are very simple, can be implemented with both linked-lists and vectors, and are used in many different programming applications.

STACK

A Stack is a data type that only allows users to access the newest member of the list. It is analogous to a stack of paper, where you add and remove paper from the top, but never look at the papers below it.

A typical Stack implementation supports 3 operations: Push(), Pop(), and Top().

1. Push() will add an item to the *end* of the list. This takes constant time.
2. Pop() will remove the item at the *end* of the list. This takes constant time.
3. Top() will return the value of the item at the top.

All operations on a stack happen in constant time, because no matter what, the stack is always working with the top- most value, and the stack always knows exactly where that is. This is the main reason why Stacks are so amazingly fast.

QUEUE

A Queue is a data structure where you can only access the oldest item in the list. It is analogous to a line in the grocery store, where many people may be in the line, but the person in the front gets serviced first.

A typical Queue implementation has 3 operations, which are similar to the functions in Stacks. They are: enqueue(), dequeue(), and Front().

1. Enqueue() will add an item to the end of the list. This takes constant time.
2. Dequeue() will remove an item from the beginning of the list. This takes constant time.
3. Front() will return the value of front-most item.

Queues, like Stacks, are very fast because all of the operations are simple, and constant-time.

I will provide a sample implementation in C. However, this code will produce a Queue that cannot resize when it runs out of room.

NON LINEAR DATA STRUCTURE

TREE:

In computer science, a **tree** is a widely used abstract data type (ADT)—or data structure implementing this ADT—that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

DEFNITION: A tree is a data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

GRAPH:

In computer science, a **graph** is an abstract data type that is meant to implement the undirected graph and directed graph concepts from mathematics, specifically the field of graphtheory.

A graph data structure consists of a finite (and possibly mutable) set of *vertices* or *nodes* or *points*, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges*, *arcs*, or *lines* for an undirected graph and as *arrows*, *directed edges*, *directed arcs*, or *directed lines* for a directed graph. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some *edge value*, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

LIST ADT

List is basically the collection of elements arranged in a sequential manner. In memory we can store the list in two ways: one way is we can store the elements in sequential memory locations. That means we can store the list in arrays. The other way is we can use pointers or links to associate elements sequentially. This is known as linked list.

Array representation

You will know an array is simply an area of memory allocated for a set number of elements of a known size. You can access these elements by their index (position in the array) and also set or retrieve the value stored.

An array is always of a fixed size; it does not grow as more elements are required. The programmer must ensure that only valid values in the array are accessed, and must remember the location in the array of each value. Arrays are basic types in most programming languages

Linked representation

A linked list is made up of a linear series of nodes (For non-linear arrangements of nodes, see Trees and Graphs. These nodes, unlike the elements in an array, do not have to be located next to each other in memory in order to be accessed. The reason is that each node contains a link to another node. The most basic node would have a data field and just one link field. This node would be a part of what is known as a **singly linked list**, in which all nodes contain only a *next* link. This is different than a **doubly linked list**, in which all nodes have two links, a *next* and a *previous*.

The linked list requires linear $O(N)$ time to find or access a node, because there is no simple formula as listed above for the array to give the memory location of the node. One must traverse all links from the beginning until the requested node is reached. If nodes are to be inserted at the beginning or end of a linked list, the time is $O(1)$, since references or pointers, depending on the language, can be maintained to the *head* and *tail* nodes. If a node should be inserted in the middle or at some arbitrary position, the running time is not actually $O(1)$, as the operation to get to the position in the list is $O(N)$.

Vector representation

Vectors are much like arrays. Operations on a vector offer the same big O as their counterparts on an array. Like arrays, vector data is allocated in contiguous memory.

Unlike static arrays, which are always of a fixed size, vectors can be grown. This can be done either explicitly or by adding more data. In order to do this efficiently, the typical vector implementation grows by

doubling its allocated

space (rather than incrementing it) and often has more space allocated to it at any one time than it needs. This is because reallocating memory is usually an expensive operation. Vectors are simply arrays which have wrapped grow/shrink functions.

Vector vs ArrayList in Java

ArrayList and Vectors both implement the List interface and both use **(dynamically resizable) arrays** for its internal data structure, much like using an ordinary array.

Syntax:

```
ArrayList<T> al = new  
ArrayList<T>(); Vector<T> v =
```

Major Differences between ArrayList and Vector:

1. **Synchronization** : Vector is **synchronized**, which means only one thread at a time can access the code, while arrayList is **not synchronized**, which means multiple threads can work on arrayList at the same time. For example, if one thread is performing an add operation, then there can be another thread performing a remove operation in a multithreading environment.

If multiple threads access arrayList concurrently, then we must synchronize the block of the code which modifies the list structurally, or alternatively allow simple element modifications. Structural modification means addition or deletion of element(s) from the list. Setting the value of an existing element is not a structural modification.

2. **Performance: ArrayList is faster**, since it is non-synchronized, while vector operations give slower performance since they are synchronized (thread-safe). If one thread works on a vector, it has acquired a lock on it, which forces any other thread wanting to work on it to have to wait until the lock is released.
3. **Data Growth**: ArrayList and Vector **both grow and shrink dynamically** to maintain optimal use of storage – but the way they resize is different. ArrayList increments 50% of the current array size if the number of elements exceeds its capacity, while vector increments 100% – essentially doubling the current arraysize.
4. **Traversal**: Vector can use both **Enumeration and Iterator** for traversing over elements of vector while ArrayList can only use **Iterator** for traversing.
5. *Note: ArrayList is preferable when there is no specific requirement to use vector.*

```
// Java Program to illustrate use of ArrayList  
// and Vector in Java  
import java.io.*;  
import java.util.*;  
  
class GFG  
{  
    public static void main (String[] args)  
    {  
        // creating an ArrayList  
        ArrayList<String> al = new ArrayList<String>();  
        // adding object to arraylist  
        al.add("Practice.GeeksforGeeks.org");  
    }  
}
```

```
al.add("quiz.GeeksforGeeks.org");
al.add("code.GeeksforGeeks.org");
al.add("contribute.GeeksforGeeks.org");
// traversing elements using
Iterator'
System.out.println("ArrayList
elements are:"); Iterator it =
al.iterator();
while (it.hasNext())
    System.out.println(it.next());
;

// creating Vector
Vector<String> v = new Vector<String>();
v.addElement("Practice");
v.addElement("quiz");
v.addElement("code");

// traversing elements using Enumeration
System.out.println("\nVector elements
are:"); Enumeration e = v.elements();
while (e.hasMoreElements())
    System.out.println(e.nextElement());
}
}
```

Output:

```
ArrayList elements are:
Practice.GeeksforGeeks.org
quiz.GeeksforGeeks.org
code.GeeksforGeeks.org
contribute.GeeksforGeeks.org
Vector elements are:
Practice
quiz
code
```

How to choose between ArrayList and Vector?

1. ArrayList is unsynchronized and not thread-safe, whereas Vectors are. Only one thread can call methods on a Vector at a time, which is a slight overhead, but helpful when safety is a concern. Therefore, in a single- threaded case, arrayList is the obvious choice, but where multithreading is concerned, vectors are often

preferable.

2. If we don't know how much data we are going to have, but know the rate at which it grows, Vector has an advantage, since we can set the increment value in vectors.
3. ArrayList is newer and faster. If we don't have any explicit requirements for using either of them, we use ArrayList over vector.

LINKED LIST

Introduction to Linked List:

A **linked list** is a linear collection of data elements, called **nodes**, where the linear order is given by means of **pointers**. Each **node** is divided into two parts:

1. The first part contains the **information** of the element and
2. The second part contains the address of the next node (**link /next pointer field**) in the list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

1. In array implementation of the linked lists a fixed set of nodes represented by an array is established at the beginning of the execution
2. A pointer to a node is represented by the relative position of the node within the array.
3. In array implementation, it is not possible to determine the number of nodes required for the linked list. Therefore;
 - a. Less number of nodes can be allocated which means that the program will have overflow problem.
 - b. More number of nodes can be allocated which means that some amount of the memory storage will be wasted.
4. The solution to this problem is to allow nodes that are **dynamic**, rather than static.
5. When a node is required storage is reserved /allocated for it and when a node is no longer needed, the memory storage is released /freed.

Advantages of linked lists

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.

2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

Comparison between array and linked list:

| ARRAY | LINKED LIST |
|--|---|
| Size of an array is fixed | Size of a list is not fixed |
| Memory is allocated from stack | Memory is allocated from heap |
| It is necessary to specify the number of elements during declaration (i.e., during compile time). | It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time). |
| It occupies less memory than a linked list for the same number of elements. | It occupies more memory. |
| Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room. | Inserting a new element at any position can be carried out easily. |
| Deleting an element from an array is not possible. | Deleting an element is possible. |

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

$$P(x) = a_0 X^n + a_1 X^{n-1} + \dots + a_{n-1} X + a_n$$

2. Represent very large numbers and operations of the large number such as addition, multiplication and

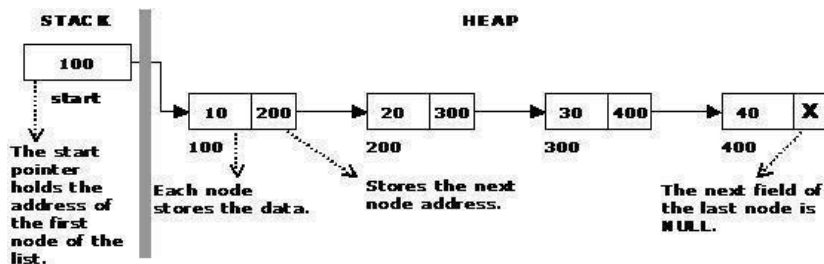
division.

3. Linked lists are to implement stack, queue, trees and graphs.
4. Implement the symbol table in compiler construction.

Single Linked List:

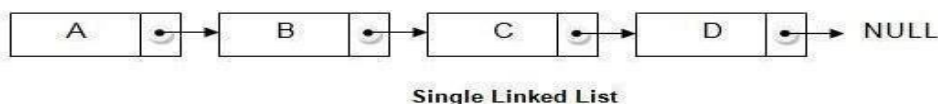
The simplest kind of linked list is a singly-linked list, which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

A singly linked list's node is divided into two parts. The first part holds or points to information about the node, and second part holds the address of next node. A singly linked list travels one way.



The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.



Implementation of Single Linked List:

1. Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
2. Initialize the start pointer to be NULL.

```
struct slinklist
{
    int data;
    struct slinklist* next;
};

typedef struct slinklist node;

node *start = NULL;
```



The basic operations in a single linked list are:

1. Creation.
2. Insertion.
3. Deletion.
4. Traversing.

Advantages of singly linked list:

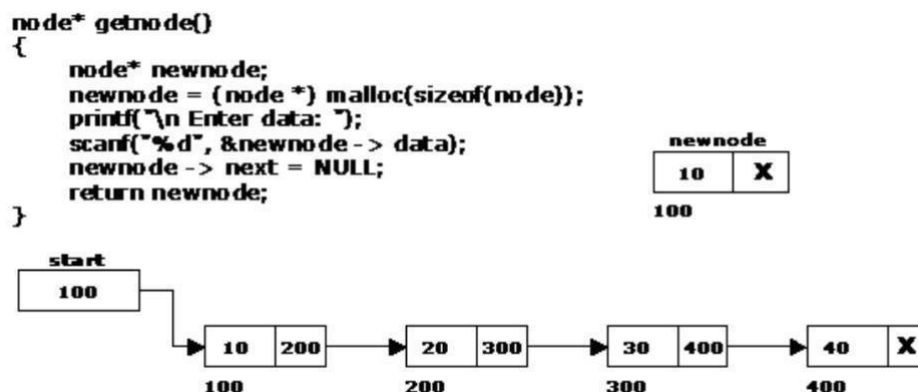
1. Dynamic data structure.
2. We can perform deletion and insertion anywhere in the list.
3. We can merge two lists easily.

Disadvantages of singly linked list:

1. Backward traversing is not possible in singly linked list.
2. Insertion is easy but deletion takes some additional time, because of backward traversing.

Creating a node for Single Linked List

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the `malloc()` function. The function `getnode()`, is used for creating a node, after allocating memory for the structure of type `node`, the information for the item (i.e., data) has to be read from the user, set next field to `NULL` and finally returns the address of the node.

**Insertion of a Node:**

The new node can then be inserted at three different places namely:

1. Inserting a node at the beginning.
2. Inserting a node at the end.
3. Inserting a node at intermediate position.

Inserting a node at the beginning:

Insertion of a new node is quite simple. It is just a 2-step algorithm which is performed to insert a node at the start of a singly linked list.

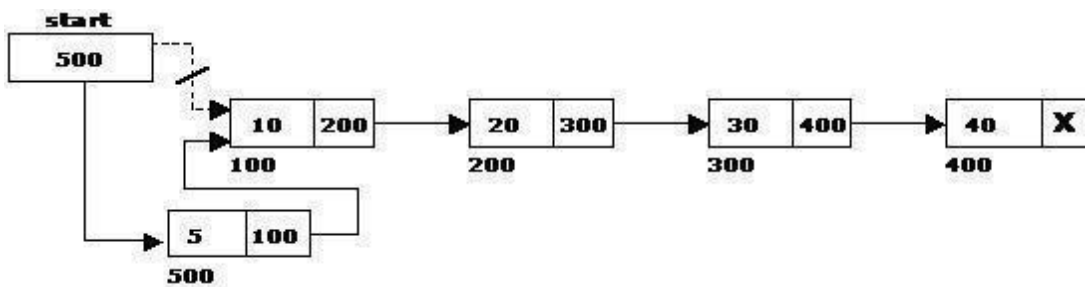
1. New node should be connected to the first node, which means the head. This can be achieved by

putting the address of the head in the next field of the new node.

2. New node should be considered as a head. It can be achieved by declaring head equals to a newnode.

1. Get the new node using `getnode()`. `newnode = getnode();`
2. If the list is empty then `start = newnode`.
3. If the list is not empty, follow the steps given below:

`newnode -> next =`
`start; start = newnode;`



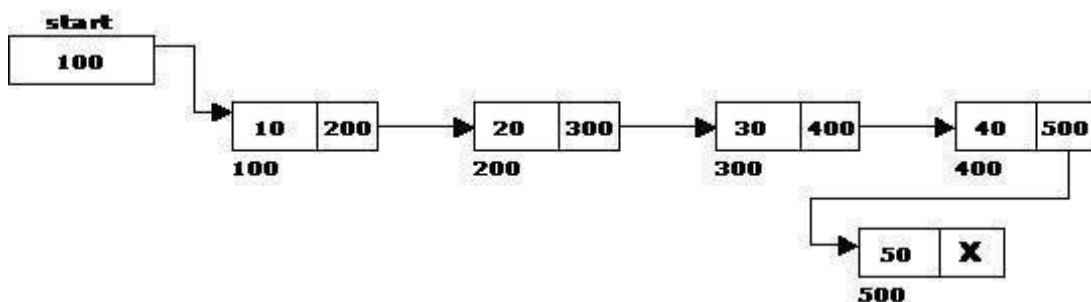
Inserting a node at the end:

The insertion of a node at the end of a linked list is the same as we have done in node creation function. If you noticed then, we inserted the newly created node at the end of the linked list. So this process is the same.

1. The following steps are followed to insert a new node at the end of the list:
2. Get the new node using `getnode()`
`newnode = getnode();`
3. If the list is empty then `start = newnode`.
4. If the list is not empty follow the steps given below:

`temp = start;`
`while(temp -> next != NULL)`

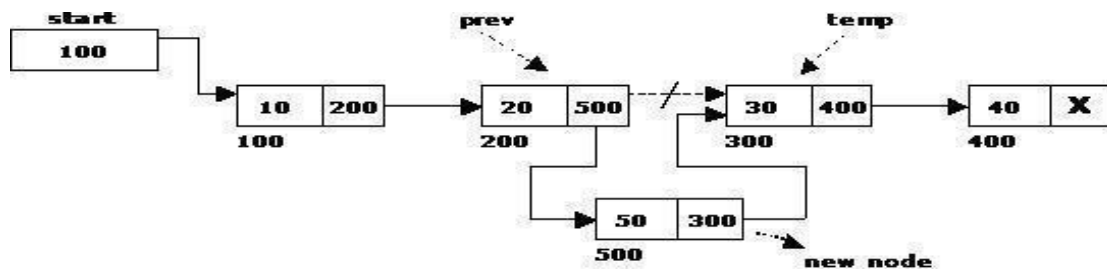
1. `temp = temp -> next;`
2. `temp -> next = newnode;`



Inserting a node at intermediate position:

1. The following steps are followed, to insert a new node in an intermediate position in the list
2. Get the new node using `getnode()`.
`newnode = getnode();`

3. Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
4. Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
5. After reaching the specified position, follow the steps given below:



prev -> next =
newnode; newnode ->
next = temp;

Deletion of a node:

A node can be deleted from the list from three different places namely.

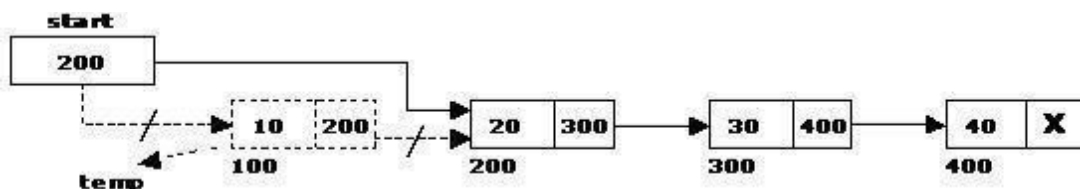
1. Deleting a node at the beginning.
2. Deleting a node at the end.
3. Deleting a node at intermediate position.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

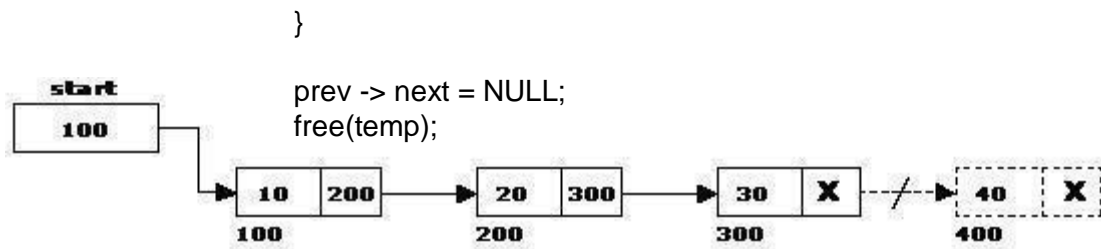
- i. temp = start;
- ii. start = start -> next;
- iii. free(temp);



Deleting a node at the end:

1. The following steps are followed to delete a node at the end of the list:
2. If list is empty then display 'Empty List' message.
3. If the list is not empty, follow the steps given below:


```
temp = prev = start;
while(temp -> next !=
NULL)
{
  1. prev = temp;
  2. temp = temp -> next;
```



Deleting a node at Intermediate position:

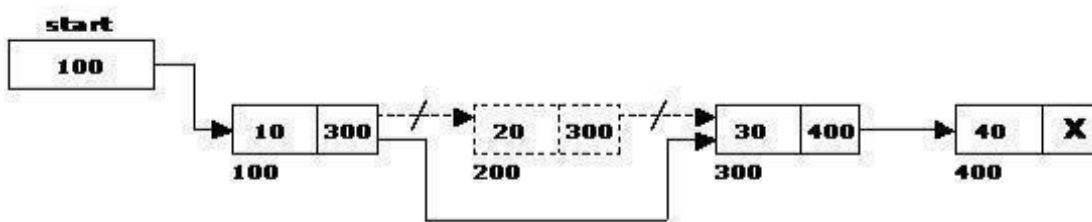
The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

1. If list is empty then display 'Empty List' message
2. If the list is not empty, follow the steps

given below. if(pos > 1 && pos < nodectr)

```

{
    temp = prev = start; ctr =
    1; while(ctr < pos)
    {
        prev = temp;
        temp = temp -> next;
        ctr++;
    }
    prev -> next = temp -> next;
    free(temp); printf("\n node deleted..");
}
  
```



Traversal and displaying a list (Left to Right):

Traversing a list involves the following steps:

1. Assign the address of start pointer to a temp pointer.
2. Display the information from the data field of each node.

Counting the Number of Nodes:

```

int countnode(node *st)
{
    if(st == NULL)
        return 0;
    else
        return(1 + countnode(st -> next));
}
  
```

Source Code (LINKED LIST USING JAVA PROGRAM)

```
Using LinkedList
//LinkedListDemo.java class
LinkedList implements List
{
class Node
{
Object data; // data item
Node next; // refers to next node in the list

Node( Object d ) // constructor
{
data = d;
} // 'next' is automatically set to null
}
Node head; // head refers to first
node Node p; // p refers to current
node int count; // current number of
nodes
public void insertFirst(Object item) // insert at the beginning of list
{
p = new Node(item); // create new node
p.next = head; // new node refers to old
head head = p; // new head refers to new
node count++;
}
public void insertAfter(Object item,Object key)
{
p = find(key); // get "location of key
item" if( p == null )
System.out.println(key + " key is not
found"); else
{
Node q = new Node(item); // create new node
q.next = p.next; // new node next refers to
p.next p.next = q; // p.next refers to new node
count++;
}
}
public Node find(Object key)
{
p = head;
while( p != null ) // start at beginning of list until end of list
{
if( p.data == key ) return p; // if found, return key
address p = p.next; // move to next node }

return null; // if key search is unsuccessful, return null
}
public Object deleteFirst() // delete first node
{
if( isEmpty() )
{
```

```
System.out.println("List is empty: no
deletion"); return null;
}
Node tmp = head; // tmp saves reference to
head head = tmp.next;
count--;
return tmp.data;
}
public Object deleteAfter(Object key) // delete node after key item
{
    p = find(key); // p = "location of key
node" if( p == null )
{
    System.out.println(key + " key is not
found"); return null;
}
if( p.next == null ) // if(there is no node after key node)
{
    System.out.println("No
deletion"); return null;
}
else
{
    Node tmp = p.next; // save node after key node
    p.next = tmp.next; // point to next of node
deleted count--;
return tmp.data; // return deleted node
}
}
public void displayList()
{
    p = head; // assign mem. address of 'head' to 'p'
    System.out.print("\nLinked List: ");
    while( p != null ) // start at beginning of list until
end of list {
        System.out.print(p.data + " -> "); // print
data p = p.next; // move to next node
    }
    System.out.println(p); // prints 'null'
}
public boolean isEmpty() // true if list is empty
{
    return (head == null);
}
public int size()
{
    return count;
}
} // end of LinkeList
class class
LinkedListDemo
{
    public static void main(String[] args)
```



```
{
    LinkedList list = new LinkedList(); //
    create list object list.createList(4); // create
    4 nodes list.displayList();
    list.insertFirst(55); // insert 55 as first
    node list.displayList();
    list.insertAfter(66, 33); // insert 66 after 33
    list.displayList();
    Object item = list.deleteFirst(); // delete
    first node if( item != null )
    {
        System.out.println("deleteFirst():
        " + item); list.displayList();
    }
    item = list.deleteAfter(22); // delete a node after
    node(22) if( item != null )
    {
        System.out.println("deleteAfter(22): " +
        item); list.displayList();
    }
    System.out.println("size(): " + list.size());
}
}
```

OUTPUT:

```

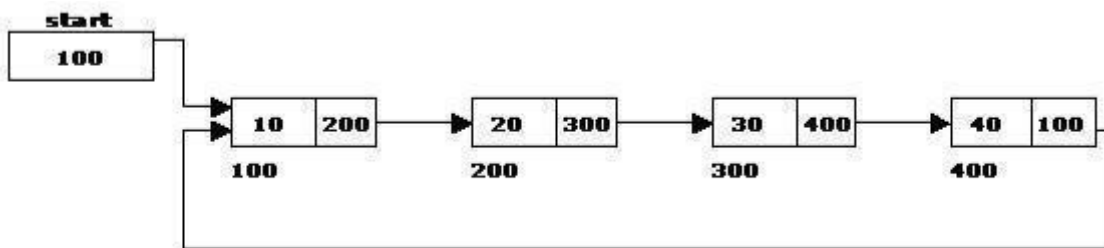
C:\WINDOWS\system32\cmd.exe

D:\SHIRISHA>java LinkedListDemo
Linked List: 11 -> 22 -> 33 -> 44 -> null
Linked List: 55 -> 11 -> 22 -> 33 -> 44 -> null
Linked List: 55 -> 11 -> 22 -> 33 -> 66 -> 44 -> null
deleteFirst(): 55
Linked List: 11 -> 22 -> 33 -> 66 -> 44 -> null
deleteAfter(22): 33
Linked List: 11 -> 22 -> 66 -> 44 -> null
size(): 4
D:\SHIRISHA>

```

Circular Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list.



The basic operations in a circular single linked list are:

1. Creation.
2. Insertion.
3. Deletion.
4. Traversing.

Creating a circular single Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

1. Insertion in an empty List

Initially when the list is empty, *last* pointer will be NULL.

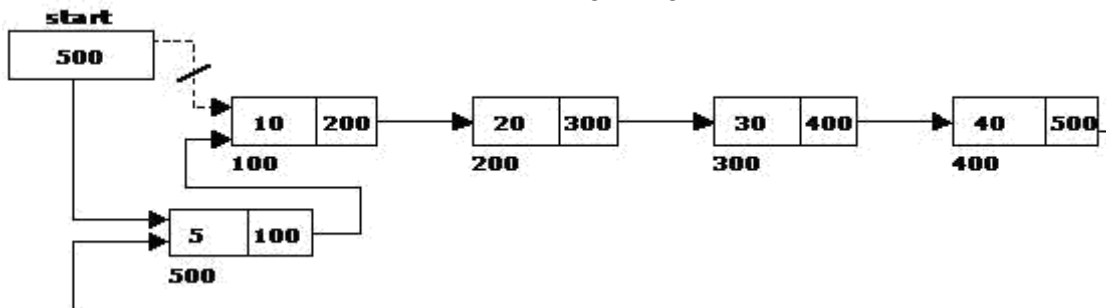
After insertion, T is the last node so pointer *last* points to node T. And Node T is first and last node, so T is pointing to itself.

Insertion a Node at the beginning of the list

To Insert a node at the beginning of the list, follow these step:

1. Create a node, say 5.
2. . Make 5 -> next = last -> next.
3. . last -> next = 5.

The following steps are to be followed to insert a new node at the beginning of the circular list: Function to insert node in the beginning of the List,



Inserting a node at the end:

To Insert a node at the end of the list, follow these step:

1. Create a node, say 50.
2. Make 50 -> next = last -> next;
3. last -> next = 50.
4. last = 50.

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using
`getnode(). newnode
 = getnode();`
2. If the list is empty, assign
 new node as start.
`start = newnode;
 newnode -> next =
 start;`
3. If the list is not empty follow the
 steps given below: `temp =
 start;`

`while(temp -> next !=`

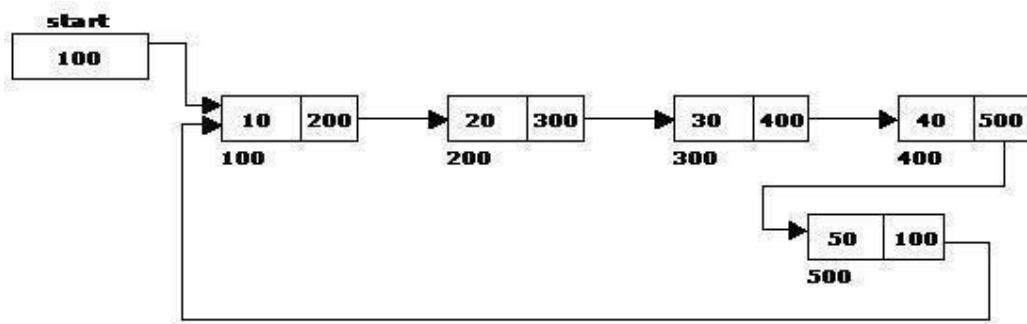
`start) temp =`

`temp -> next;`

`temp -> next =`

`newnode; newnode`

`-> next = start;`



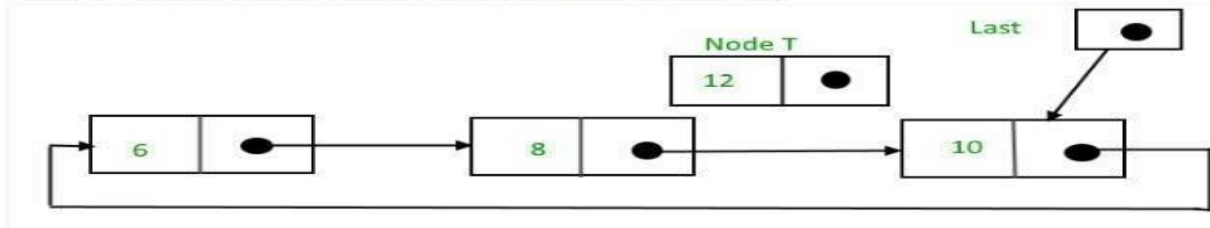
Insertion in between the nodes circular linked list

To Insert a node at the end of the list, follow these step:

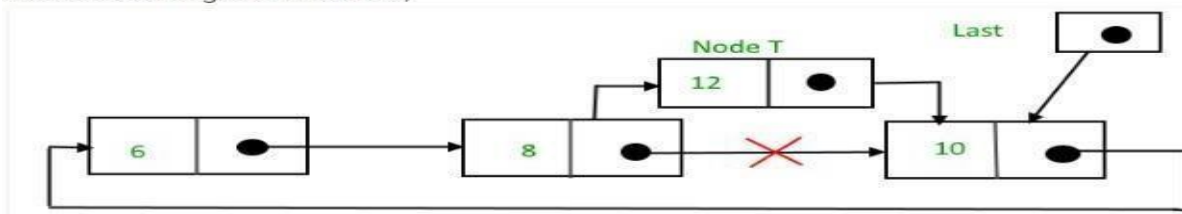
1. Create a node, say T.
2. Search the node after which T need to be insert, say that node be P.
3. Make T -> next = P -> next;
4. P -> next = T.

Suppose 12 need to be insert after node having value 10,
10, After searching and insertion,

Suppose 12 need to be insert after node having value 10,



After searching and insertion,



Deleting a node at the beginning:

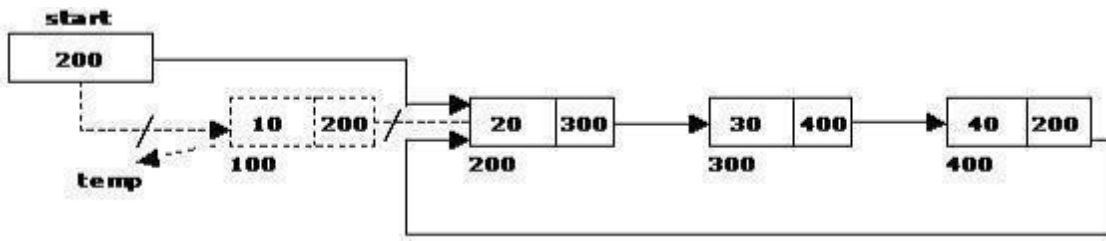
The following steps are followed, to delete a node at the beginning of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below: last = temp = start;

```

while(last -> next !=
      start) last = last
      -
      > next; start = start
      -> next;
      last -> next = start;

```



Deleting a node at the end:

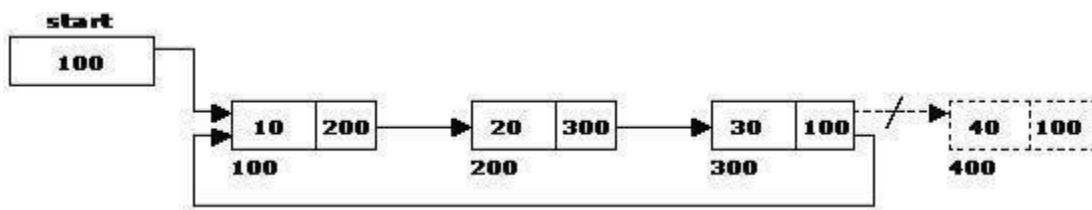
The following steps are followed to delete a node at the end of the list:

1. If the list is empty, display a message 'Empty List'.
2. If the list is not empty, follow the steps given below:


```

temp = start;
prev = start;
while(temp -> next
      != start)
{
  prev = temp;
  temp = temp -> next;
}
prev -> next = start;

```
3. After deleting the node, if the list is empty then start = NULL.



Traversing a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow

```
the steps given below:  
temp = start;  
  
do  
  
{temp = temp -> next;  
  
} while(temp != start);
```

Source Code:(CIRCULAR LINKED LIST USING JAVA PROG insertion traversing techniques)

```
class GFG  
{  
static class Node  
{  
    int  
    data;  
    Node  
    next;  
};  
static Node addToEmpty(Node last, int data)  
{  
    // This function is only for  
    empty list if (last != null)  
    return last;  
  
    // Creating a node  
    dynamically. Node temp = new  
    Node();  
  
    // Assigning  
    the data.  
    temp.data =  
    data; last =  
    temp;  
  
    // Creating the link.  
    last.next = last;  
  
    return last;  
}  
static Node addBegin(Node last, int data)  
{  
    if (last == null)
```

```
        return addToEmpty(last,

data); Node temp = new

Node();

temp.data = data;
temp.next =
last.next;
last.next = temp;
return last;
}

static Node addEnd(Node last, int data)
{
    if (last == null)
        return addToEmpty(last,

data); Node temp = new

Node();

temp.data = data;
temp.next =
last.next;
last.next = temp;
last =
temp;
return
last;
}

static Node addAfter(Node last, int data, int item)
{
    if (last ==
        null)
        return
        null;

    Node
temp, p;
p =
last.next;
```

```
do
{
    if (p.data == item)
    {
        temp = new
        Node();
        temp.data =
        data;
        temp.next =
        p.next; p.next
        = temp;

        if (p ==
            last)
            last =
            temp;
        return last;
    }
    p = p.next;
} while(p != last.next);
System.out.println(item + " not present
in the list."); return last;
}

static void traverse(Node last)
{
    Node p;

    // If list is empty,
    return. if (last ==
    null)
    {
        System.out.println("Li
        st is empty.");
        return;
    }

    // Pointing to first Node of the
    list. p = last.next;

    // Traversing the
    list. do
    {
        System.out.print(p.data +
```



```

        " "); p = p.next;

    }
    while(p != last.next);

}

// Driven code
public static void main(String[] args)
{
    Node last = null;

    last =
    addToEmpty(last, 6);
    last = addBegin(last,
    4); last =
    addBegin(last, 2);
    last = addEnd(last,
    8); last =
    addEnd(last, 12);
    last = addAfter(last, 10, 8);

    traverse(last);
}
}

```

OUTPUT:2 4 6 8 10 12

(CIRCULAR LINKED LIST USING JAVA PROG Deletion techniques)

```

// Java program to delete a given key from
// linked

```

```
list.
```

```
class GFG
```

```

{
    /* ure for a node */
    static class Node
    {
        int
        data;
        Node
        next;
    };
}

```

```

/* Function to insert a node at the
beginning of a Circular linked list */

```

```
static Node push(Node head_ref, int data)
{
    // Create a new node and make head as next
    // of it.
    Node ptr1 = new
    Node(); ptr1.data
    = data; ptr1.next
    = head_ref;

    /* If linked list is not null
    then set the next of last node */
    if (head_ref != null) {
        // Find the node before head and update
        // next of it.
        Node temp = head_ref;
        while (temp.next !=
            head_ref) temp =
            temp.next;
        temp.next = ptr1;
    }
    else
        ptr1.next = ptr1; /*For the first node */

    head_ref =
    ptr1;
    return
    head_ref;
}

/* Function to print nodes in a
given circular linked list */
static void printList(Node head)
{
    Node temp =
    head; if (head
    != null) {
        do {
            System.out.printf("%d ",
                temp.data); temp = temp.next;
        } while (temp != head);
    }

    System.out.printf("\n");
}
```

```
/* Function to delete a given node from the list
*/ static Node deleteNode(Node head, int key) {

    if (head ==
        null)
        return
        null;

    // Find the required
    node Node curr = head,
    prev = new
    Node(); while (curr.data !=
        key) { if (curr.next ==
        head) {
            System.out.printf("\nGiven node is not found"
                               + " in the list!!!");

            break;
        }

        prev =
        curr; curr =
        curr.next;
    }

    // Check if node is only
    node if (curr.next ==
    head) {
        head = null;
        return head;
    }

    // If more than one node, check if
    // it is first
    node if (curr
    == head) {
        prev = head;
        while (prev.next !=
            head) prev =
            prev.next;
        head =
        curr.next;
        prev.next =
        head;
    }
}
```

```
// check if node is last
node else if (curr.next
== head) {
    prev.next = head;
}
else {
    prev.next = curr.next;
}
return head;
}
/* Driver program to test above functions
*/ public static void main(String args[]) {

    /* Initialize lists as
    empty */ Node head = null;
    /* Created linked list will be 2.5.7.8.10 */
    head =
    push(head, 2);
    head =
    push(head, 5);
    head =
    push(head, 7);
    head =
    push(head, 8);
    head = push(head, 10);
    System.out.printf("List Before
    Deletion: "); printList(head);
    head = deleteNode(head, 7);

    System.out.printf("List After
    Deletion: "); printList(head);
}
}
```

List Before Deletion: 10 8 7 5 2

List After Deletion: 10 8 5 2

Double Linked List:

Doubly-linked list is a more sophisticated form of linked list data structure. Each node of the list contain two references (or links) – one to the previous node and other to the next node. The previous link of the first node and the next link of the last node points to NULL. In comparison to singly-linked list, doubly- linked list requires handling of more pointers but less information is

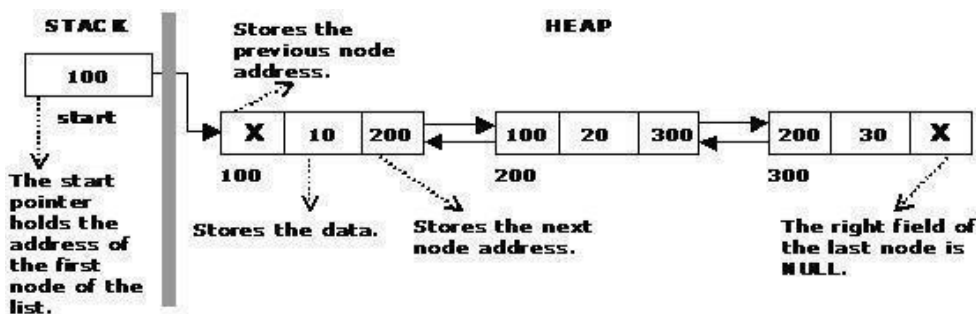
required as one can use the previous links to observe the preceding element. It has a dynamic size, which can be determined only at run time.

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi- directional traversing. Each node contains three fields:

1. Left link.
2. Data.
3. Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data. The basic operations in a double linked list are:

1. Creation.
2. Insertion.
3. Deletion.
4. Traversing.



The beginning of the double linked list is stored in a "start" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

```
struct dlinklist
{
    struct dlinklist *left;
    int data;
    struct dlinklist *right;
};

typedef struct dlinklist node;
node *start = NULL;
```

nodes: left data right

Empty list: start
 NULL

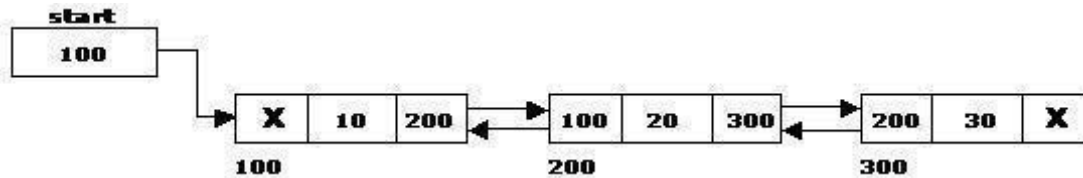
Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function.

Creating a Double Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

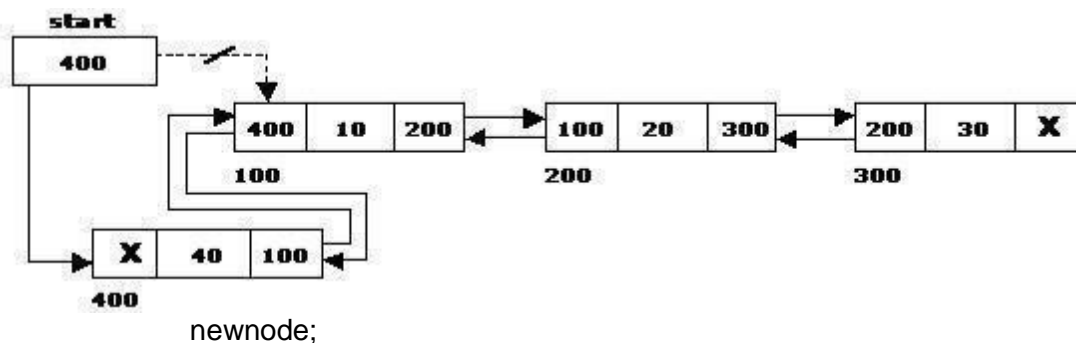
1. Get the new node using `getnode()`.
`newnode = getnode();`
2. If the list is empty then `start = newnode`.
3. If the list is not empty, follow the steps given below:
 - i. The left field of the new node is made to point the previous node.
 - ii. The previous node's right field must be assigned with address of the new node.
4. Repeat the above steps 'n' times.



Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

1. Get the new node using `getnode()`.
`newnode = getnode();`
2. If the list is empty then `start = newnode`.
3. If the list is not empty, follow the steps given below: `newnode -> right = start;`
`start -> left = newnode;`
`start = newnode;`



Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using `getnode()`.
`newnode = getnode();`
2. If the list is empty then `start = newnode`.

3. If the list is not empty follow the steps

givenbelow: temp = start;

while(temp -> right !=

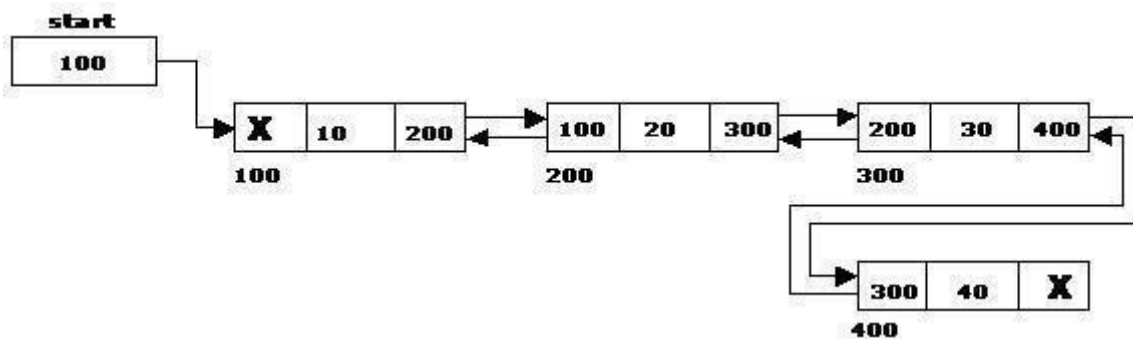
NULL) temp = temp

-> right;

temp -> right =

newnode; newnode ->

left = temp;



Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

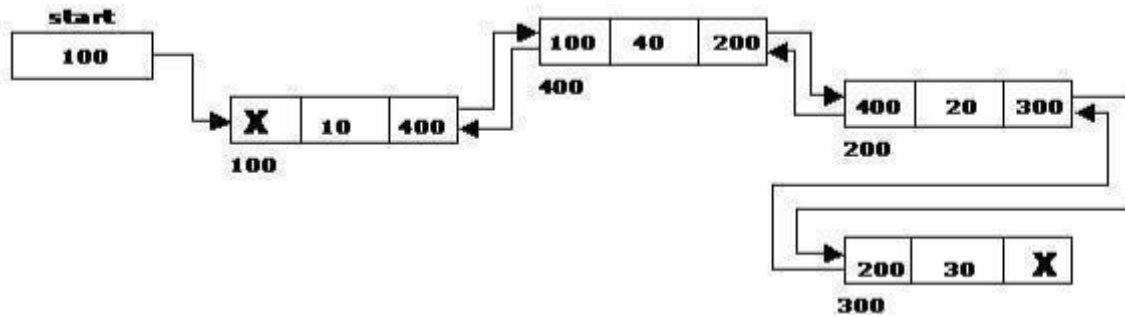
1. Get the new node using getnode(). newnode=getnode();
2. Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
3. Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
4. After reaching the specified position, follow the steps given below: newnode -> left = temp;

newnode -> right = temp ->

right; temp -> right -> left =

newnode; temp -> right =

newnode;



Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps

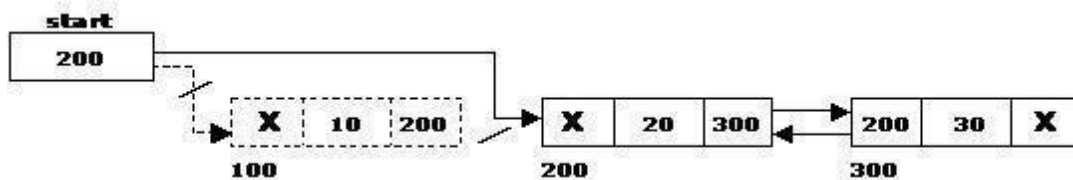
givenbelow: temp = start;

start = start ->

right; start -> left

= NULL;

free(temp);



Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

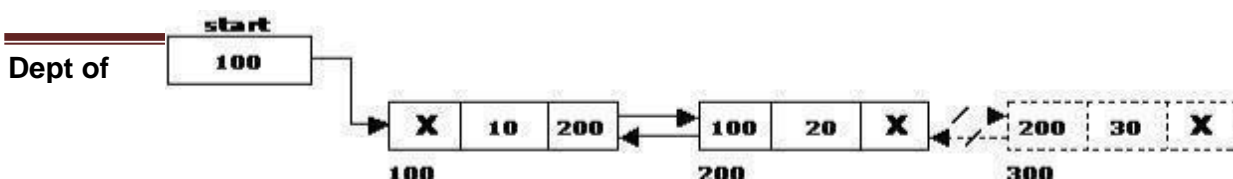
1. If list is empty then display 'Empty List' message
2. If the list is not empty, follow the steps given below: temp = start;

while(temp -> right != NULL)

```
{
    temp = temp -> right;
}
```

temp -> left -> right =

NULL; free(temp);



Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

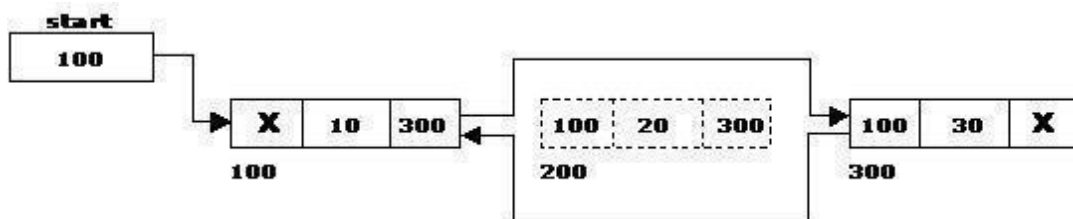
1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:
 - i. Get the position of the node to delete.
 - ii. Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
 - iii. Then perform the following steps:


```
if(pos > 1 && pos < nodectr)
{temp = start;
  i = 1; while(i < pos)
  {temp = temp -> right; i++;}

  temp -> right -> left = temp -> left;
  temp -> left -> right = temp -> right;

  free(temp);

  printf("\n node deleted..");}
```

**Traversal and displaying a list (Left to Right):**

The following steps are followed, to traverse a list from left to right:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = start;
while(temp != NULL)
{
  print temp -> data; temp = temp -> right;
}
```

Traversal and displaying a list (Right to Left):

The following steps are followed, to traverse a list from right to left:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below: temp = start;

while(temp -> right != NULL) temp = temp -> right;

while(temp != NULL)
{print temp -> data; temp = temp -> left;
}

Source Code: (DOUBLE LINKED LIST USING JAVA PROG)

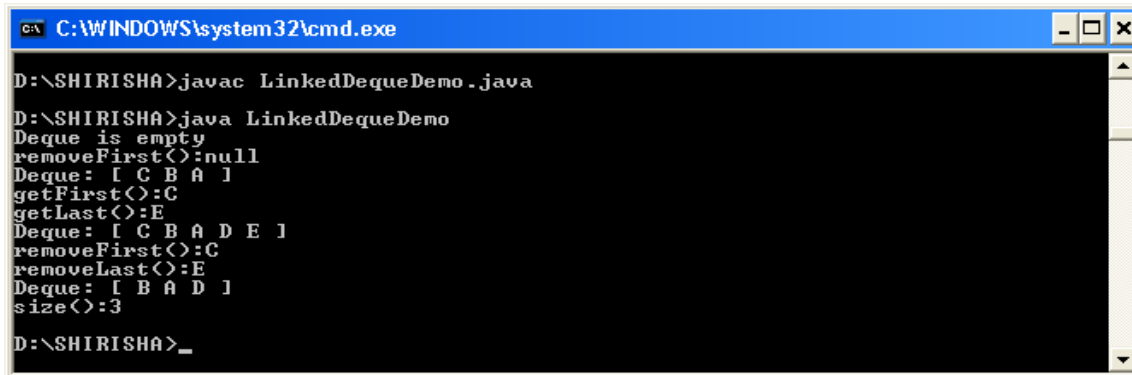
```
class LinkedDeque
{
    public class DequeNode
    {
        DequeNode prev;
        Object data;
        DequeNode next;
        DequeNode( Object item ) // constructor
        {
            data = item;
        } // prev & next automatically refer to null
    }
    private DequeNode first, last;
    private int count;
    public void addFirst(Object item)
    {
        if( isEmpty() )
            first = last = new DequeNode(item);
        else
        {
            DequeNode tmp = new DequeNode(item);
            tmp.next = first;
            first.prev = tmp;
            first = tmp;
        }
        count++;
    }
}
```

```
public void addLast(Object item)
```

```
{
    if( isEmpty() )
        first = last = new DequeNode(item);
    else
    {
        DequeNode tmp = new DequeNode(item);
        tmp.prev = last;
        last.next = tmp;
        last = tmp;
    }
    count++;
}
public Object removeFirst()
{
    if( isEmpty() )
    {
        System.out.println("Deque is empty");
        return null;
    }
    Else
    {
        Object item = first.data;
        first = first.next;
        first.prev = null;
        count--;
        return item;
    }
}
public Object removeLast()
{
    if( isEmpty() )
    {
        System.out.println("Deque is empty");
        return null;
    }
    else
    {
        Object item = last.data;
        last = last.prev;
        last.next = null;
        count--;
        return item;
    }
}
public Object getFirst()
{
    if( !isEmpty() )
```

```
        return( first.data );
    else
        return null;
    }
    public Object getLast()
    {
        if( !isEmpty() )
            return( last.data );
        else return null;
    }
    public boolean isEmpty()
    {
        return (count == 0);
    }
    public int size()
    {
        return(count);
    }
    public void display()
    {
        DequeNode p = first;
        System.out.print("Deque: [ ");
        while( p != null )
        {
            System.out.print( p.data + " " );
            p = p.next;
        }
        System.out.println("]");
    }
}
class LinkedDequeDemo
{
    public static void main( String args[])
    {
        LinkedDeque dq = new LinkedDeque();
        System.out.println("removeFirst(): " + dq.removeFirst());
        dq.addFirst('A');
        dq.addFirst('B');
        dq.addFirst('C');
        dq.display();
        dq.addLast('D');
        dq.addLast('E');
        System.out.println("getFirst(): " + dq.getFirst());
        System.out.println("getLast(): " + dq.getLast());
        dq.display();
        System.out.println("removeFirst(): " + dq.removeFirst());
        System.out.println("removeLast(): " + dq.removeLast());
        dq.display();
    }
}
```

```
        System.out.println("size():" + dq.size());  
    }  
}
```

OUTPUT:

```
C:\WINDOWS\system32\cmd.exe  
D:\SHIRISHA>javac LinkedDequeDemo.java  
D:\SHIRISHA>java LinkedDequeDemo  
Deque is empty  
removeFirst():null  
Deque: [ C B A ]  
getFirst():C  
getLast():E  
Deque: [ C B A D E ]  
removeFirst():C  
removeLast():E  
Deque: [ B A D ]  
size():3  
D:\SHIRISHA>_
```

SPARSE MATRICES AND THEIR REPRESENTATION

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total $m \times n$ values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements.

Example:

```
00304  
00570  
00000  
02600
```

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value)**.

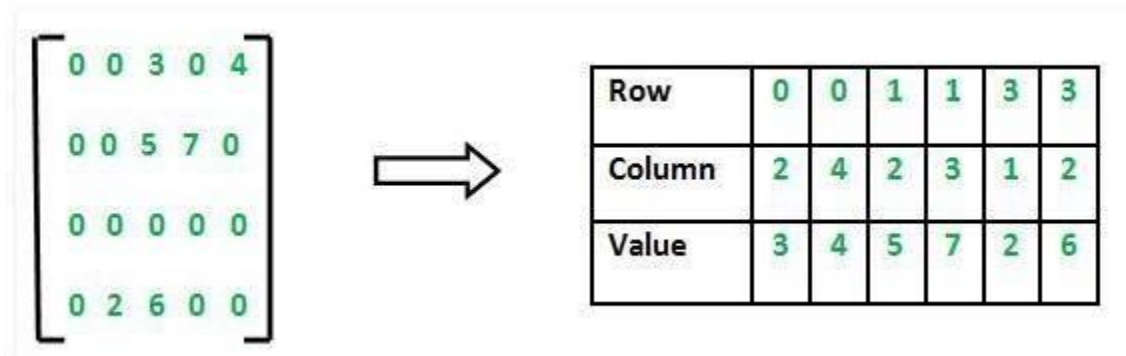
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



EXAMPLE:

```

/ Java program for Sparse Matrix
Representation // using Array
class GFG
{
    public static void main(String[] args)
    {
        int sparseMatrix[][]
            = {
                {0, 0, 3, 0, 4},
                {0, 0, 5, 7, 0},
                {0, 0, 0, 0, 0},
                {0, 2, 6, 0, 0}
            };

        int size = 0;
        for (int i = 0; i < 4; i++)
        {
            for (int j = 0; j < 5; j++)
            {
                if (sparseMatrix[i][j] != 0)
                {
                    size++;
                }
            }
        }
    }
}

```

```
        }
    }
}

// number of columns in compactMatrix (size) must be
// equal to number of non - zero elements in
// sparseMatrix
int compactMatrix[][] = new int[3][size];

// Making of new matrix
int k = 0;
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 5; j++)
    {
        if (sparseMatrix[i][j] != 0)
        {
            compactMatrix[0][k] = i;
            compactMatrix[1][k] = j;
            compactMatrix[2][k] = sparseMatrix[i][j];
            k++;
        }
    }
}

for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < size; j++)
    {
        System.out.printf("%d ", compactMatrix[i][j]);
    }
    System.out.printf("\n");
}
}
```

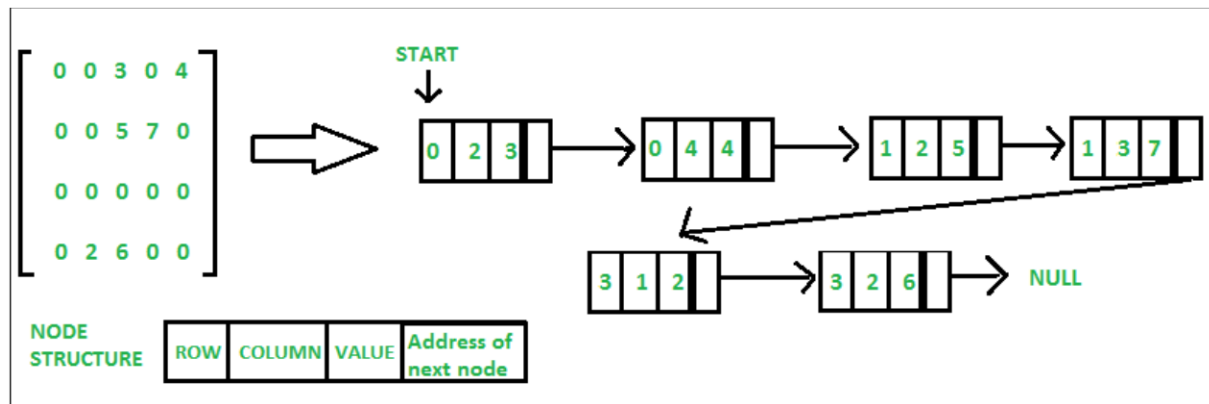
OUTPUT:

```
0 0 1 1 33
2 4 2 3 12
3 4 5 7 26
```


Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

1. **Row:** Index of row, where non-zero element is located
2. **Column:** Index of column, where non-zero element is located
3. **Value:** Value of the non zero element located at index – (row,column)
4. **Next node:** Address of the next node



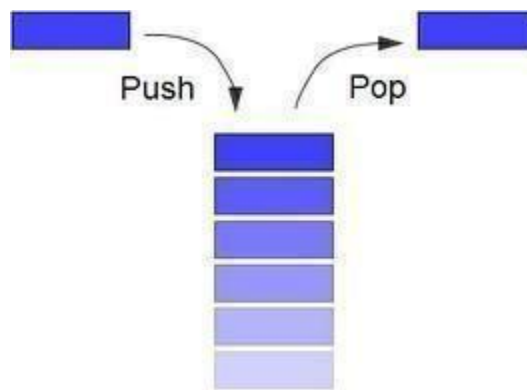
UNIT – 2

STACKS AND QUEUES

Basic Stack Operations:

A stack is a container of objects that are inserted and removed according to the last-in first-out (**LIFO**) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

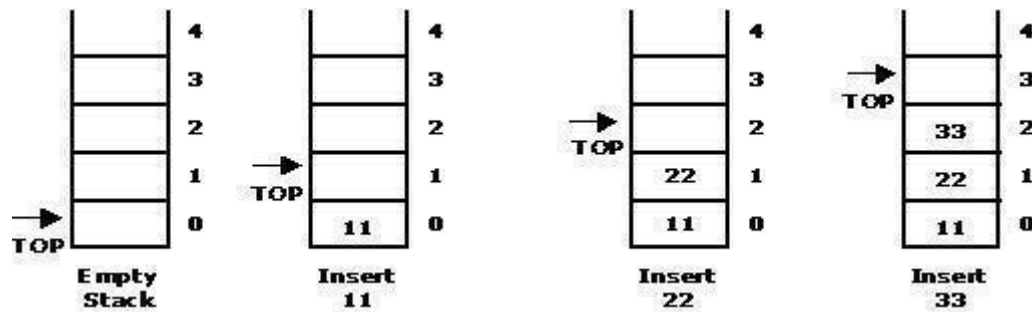
A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.



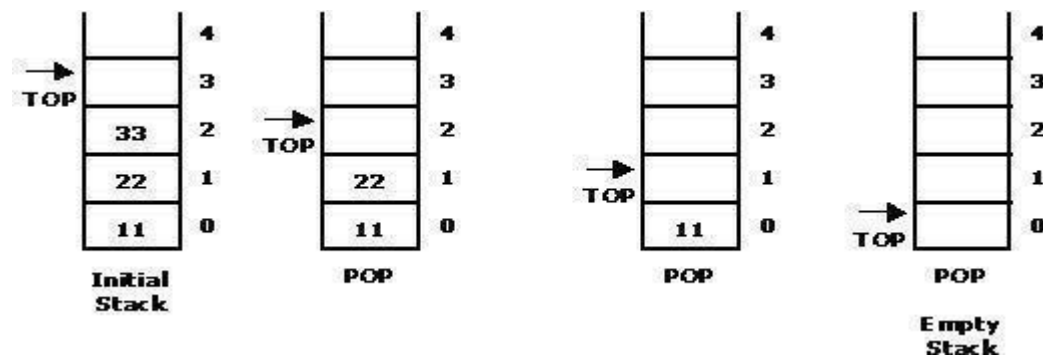
Representation of a Stack using Arrays:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition.

When an element is added to a stack, the operation is performed by push().



When an element is taken off from the stack, the operation is performed by pop().



Source code for stack operations, using array:

Procedure:

STACK: Stack is a linear data structure which works under the principle of **last in first out**. **Basic operations:** push, pop, display.

1. **PUSH:** if (top==MAX), display **Stack overflow** else reading the data and making stack [top] =data and incrementing the top value by doing top++.
2. **Pop:** if (top==0), display **Stack underflow** else printing the element at the top of the stack and decrementing the top value by doing the top--.

DISPLAY: IF (TOP==0), display **Stack is empty** else printing the elements in the stack from stack [0] to stack [top].

SOURCE CODE:

stack ADT using array

```
import java.io.*;
class stackclass
{
    int top,ele,stack[],size;
```

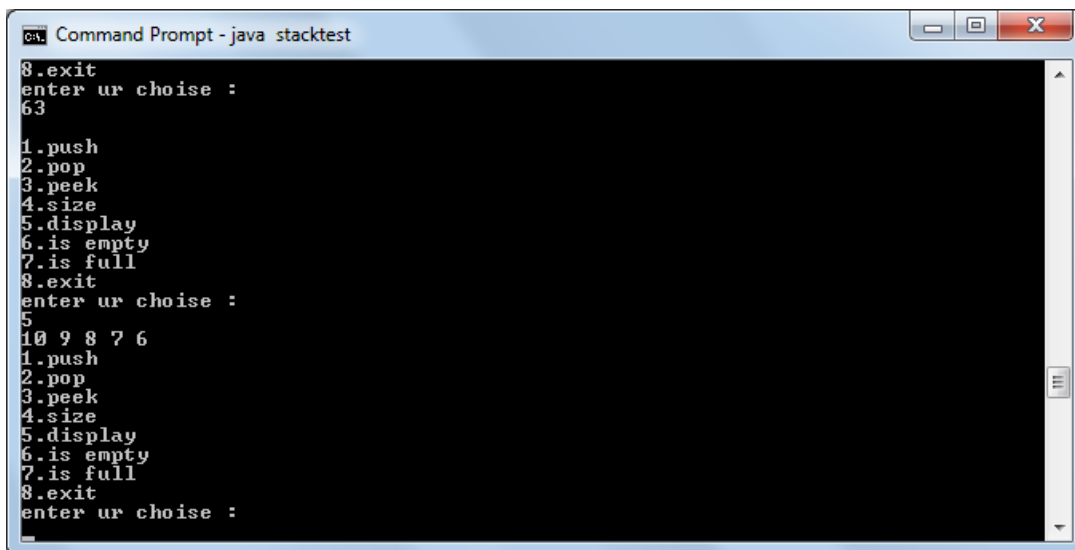
```
stackclass(int n)
{
    stack=new int[n];
    size=n;
    top= -1;
}
void push(int x)
{
    ele=x;
    stack[++top]=ele;
}
int pop()
{
    if(!isempty())
    {
        System.out.println("Deleted element is");
        return stack[top--];
    }
    else
    {
        System.out.println("stack is empty");
        return -1;
    }
}
boolean isempty()
{
    if(top==-1)
        return true;
    else
        return false;
}
boolean isfull()
{
    if(size>(top+1))
        return false;
    else
        return true;
}
int peek()
{
    if(!isempty())
        return stack[top];
    else
    {
        System.out.println("stack is empty");
        return -1;
    }
}
```

```
}
void size()
{
    System.out.println("size of the stack is :"+(top+1));
}
void display()
{
    if(!isempty())
    {
        for(int i=top;i>=0;i--)
            System.out.print(stack[i]+" ");
    }
    else
        System.out.println("stack is empty");
}
}

class stacktest
{
    public static void main(String args[])throws Exception
    {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("enter the size of stack");
        int size=Integer.parseInt(br.readLine());
        stackclass s=new stackclass(size);
        int ch,ele;
        do
        {
            System.out.println();
            System.out.println("1.push");
            System.out.println("2.pop");
            System.out.println("3.peek");
            System.out.println("4.size");
            System.out.println("5.display");
            System.out.println("6.is empty");
            System.out.println("7.is full");
            System.out.println("8.exit");
            System.out.println("enter ur choise :");
            ch=Integer.parseInt(br.readLine());
            switch(ch)
            {
                case 1:if(!s.isfull())
                    {
                        System.out.println("enter the element to insert: ");
                        ele=Integer.parseInt(br.readLine());
                        s.push(ele);
                    }
            }
        }
    }
}
```

```
        else
        {
            System.out.print("stack is overflow");
        }
        break;
    case 2: int del=s.pop();
        if(del!=-1)
            System.out.println(del+" is deleted");
        break;
    case 3: int p=s.peek();
        if(p!=-1)
            System.out.println("peek element is: +p);
        break;
    case 4: s.size();
        break;
    case 5: s.display();
        break;
    case 6: boolean b=s.isEmpty();
        System.out.println(b);
        break;
    case 7: boolean b1=s.isFull();
        System.out.println(b1);
        break;
    case 8 : System.exit(1);
    }
}while(ch!=0);
}
```

OUTPUT:



```
CA: Command Prompt - java stacktest
8.exit
enter ur choise :
63
1.push
2.pop
3.peek
4.size
5.display
6.is empty
7.is full
8.exit
enter ur choise :
5
10 9 8 7 6
1.push
2.pop
3.peek
4.size
5.display
6.is empty
7.is full
8.exit
enter ur choise :
```

```

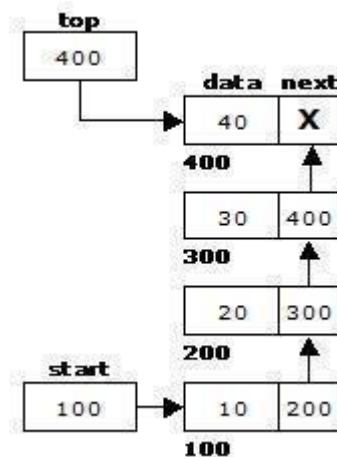
Command Prompt - java stacktest
2
Deleted element is
9 is deleted

1.push
2.pop
3.peek
4.size
5.display
6.is empty
7.is full
8.exit
enter ur choise :
5
8 7 6
1.push
2.pop
3.peek
4.size
5.display
6.is empty
7.is full
8.exit
enter ur choise :

```

Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using top pointer.



Source code for stack operations, using linked list:

STACK ADT USING SINGLE LINKED LIST

```

import java.io.*;
class Stack1
{
    Stack1 top,next,prev;
    int data;
    Stack1()
    {

```

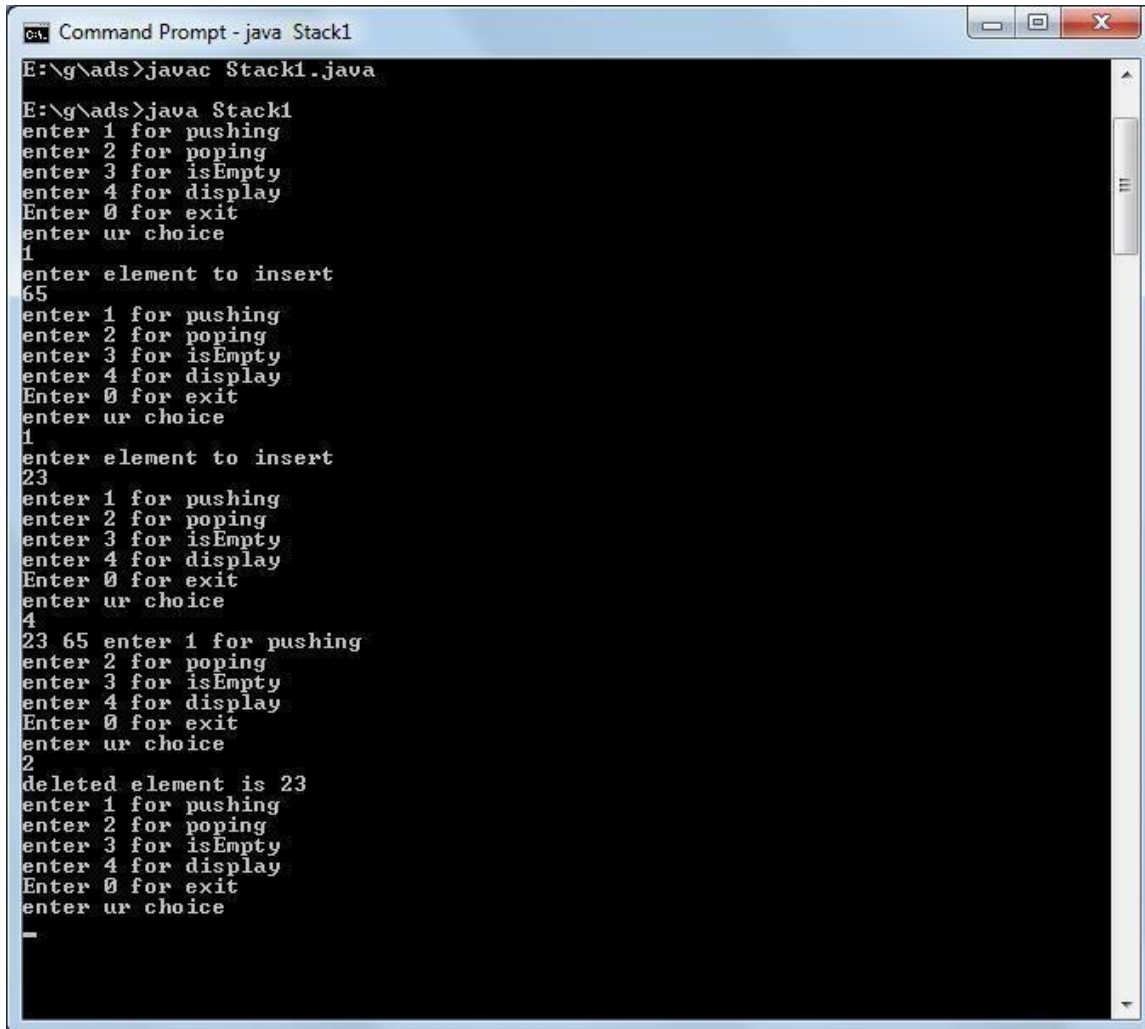
```
        data=0;
        next=prev=null;
    }
    Stack1(int d)
    {
        data=d;
        next=prev=null;
    }
    void push(int n)
    {
        Stack1 nn;
        nn=new Stack1(n);
        if(top==null)
            top=nn;
        else
        {
            nn.next=top;
            top.prev=nn;
            top=nn;
        }
    }
    int pop()
    {
        int k=top.data;
        if(top.next==null)
        {
            top=null;
            return k;
        }
        else
        {
            top=top.next;
            top.prev=null;
            return k;
        }
    }
    boolean isEmpty()
    {
        if(top==null)
            return true;
        else
            return false;
    }

    void display()
    {
        Stack1 ptr;
```



```
        for(ptr=top;ptr!=null;ptr=ptr.next)
            System.out.print(ptr.data+" ");
    }
    public static void main(String args[ ])throws Exception
    {
        int x;
        int ch;
        BufferedReader b=new BufferedReader(new InputStreamReader(System.in));
        Stack1 a=new Stack1();
        do{
            System.out.println("enter 1 for pushing");
            System.out.println("enter 2 for popping");
            System.out.println("enter 3 for isEmpty");
            System.out.println("enter 4 for display");
            System.out.println("Enter 0 for exit");
            System.out.println("enter ur choice ");
            ch=Integer.parseInt(b.readLine());
            switch(ch)
            {
                case 1:System.out.println("enter element to insert");
                    int e=Integer.parseInt(b.readLine());
                    a.push(e);
                    break;
                case 2:if(!a.isEmpty())
                    {
                        int p=a.pop();
                        System.out.println("deleted element is "+p);
                    }
                    else
                    {
                        System.out.println("stack is empty");
                    }
                    break;
                case 3:System.out.println(a.isEmpty());
                    break;
                case 4:if(!a.isEmpty())
                    {
                        a.display();
                    }
                    else
                    {
                        System.out.println("list is empty");
                    }
            }
        }while(ch!=0);
    }
}
```

OUTPUT:



```
Command Prompt - java Stack1
E:\g\ads>javac Stack1.java
E:\g\ads>java Stack1
enter 1 for pushing
enter 2 for popping
enter 3 for isEmpty
enter 4 for display
Enter 0 for exit
enter ur choice
1
enter element to insert
65
enter 1 for pushing
enter 2 for popping
enter 3 for isEmpty
enter 4 for display
Enter 0 for exit
enter ur choice
1
enter element to insert
23
enter 1 for pushing
enter 2 for popping
enter 3 for isEmpty
enter 4 for display
Enter 0 for exit
enter ur choice
4
23 65 enter 1 for pushing
enter 2 for popping
enter 3 for isEmpty
enter 4 for display
Enter 0 for exit
enter ur choice
2
deleted element is 23
enter 1 for pushing
enter 2 for popping
enter 3 for isEmpty
enter 4 for display
Enter 0 for exit
enter ur choice
```

Stack Applications:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.
6. Depth first search uses a stack data structure to find an element from a graph.

In-fix- to Postfix Transformation:**Procedure:**

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.
 - b) If the scanned symbol is an operand, then place directly in the postfix expression (output).
 - c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
 - d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Convert the following infix expression $A + B * C - D / E * H$ into its equivalent postfix expression.

| Symbol | Postfix string | Stack | Remarks |
|--------|----------------|-------|---------|
| A | A | | |
| + | A | + | |
| B | A B | + | |
| * | A B | + * | |
| C | A B C | - | |

| | | | |
|---------------|-------------|--|--|
| - | ABC*+ | - | |
| D | ABC*+D | - | |
| / | ABC*+D | - / | |
| E | ABC*+DE | - / | |
| * | ABC*+DE/ | - * | |
| H | ABC*+DE/H | - * | |
| End of string | ABC*+DE/H*- | The input is now empty. Pop the output symbols from the stack until it is empty. | |

Source Code:

```

import java.io.*;
class
InfixToPostfix
{
    java.util.Stack<Character> stk =new java.util.Stack<Character>();

    public String toPostfix(String infix)
    {
        infix = "(" + infix + ")"; // enclose infix expr
        within parentheses String postfix = "";
        /* scan the infix char-by-char until end of string is reached
        */ for( int i=0; i<infix.length(); i++) {

            char ch, item;
            ch = infix.charAt(i);
            if( isOperand(ch) ) // if(ch is an operand), then
                postfix = postfix + ch; // append ch to postfix
            string

```

```
if( ch == '(' ) // if(ch is a left-bracket),
then stk.push(ch); // push onto the
stack
if( isOperator(ch) ) // if(ch is an operator), then
{
item = stk.pop(); // pop an item from the stack
/* if(item is an operator), then check the precedence of ch and item*/
if( isOperator(item) )
{
if( precedence(item) >= precedence(ch) )
{
stk.push(item
);
stk.push(ch);
}
else
{
postfix = postfix +
item; stk.push(ch);
}
}
else
{
stk.push(item
);
stk.push(ch);
}
} // end of if(isOperator(ch))

if( ch == ')' )
{
item = stk.pop();
while( item != '(' )
{
postfix = postfix +
item; item =
stk.pop();
}
} // end of for-
loop return
postfix;
} // end of toPostfix() method

public boolean isOperand(char c)
{
return(c >= 'A' && c <= 'Z');
}

public boolean isOperator(char c)
{
```

```
return( c=='+' || c=='-' || c=='*' || c=='/' );  
}
```

```

public int precedence(char c)
{
    int rank = 1; // rank = 1 for '*' or
    '/' if( c == '+' || c == '-' ) rank = 2;
    return rank;
}

//InfixToPostfixDemo.java
//InfixToPostfixDemo
{
    public static void main(String args[]) throws IOException
    {
        InfixToPostfix obj = new InfixToPostfix();
        BufferedReader br=new BufferedReader(new
        InputStreamReader(System.in)); System.out.println("Enter
        Expression:"); String infix = br.readLine();

        //String infix = "A*(B+C/D)-E";
        System.out.println("infix: " + infix );
        System.out.println("postfix:"+obj.toPostfix(infix
        ));
    }
}

```

OUTPU

T:

```

C:\ Command Prompt
symbol : class BufferedReader
location: class InfixToPostfixDemo
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
^
InfixToPostfixDemo.java:76: cannot find symbol
symbol : class BufferedReader
location: class InfixToPostfixDemo
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
^
InfixToPostfixDemo.java:76: cannot find symbol
symbol : class InputStreamReader
location: class InfixToPostfixDemo
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
^
3 errors
E:\g\ads>javac InfixToPostfixDemo.java
E:\g\ads>java InfixToPostfixDemo
Enter Expression:
A+B-(C*D)/E
infix: A+B-(C*D)/E
postfix: AB*E/-+
E:\g\ads>

```

Evaluating Arithmetic Expressions:**Procedure:**

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

| Symb ol | Operand 1 | Operand 2 | Valu e | Stack | Remark s |
|------------|--------------|--------------|-----------|----------|---|
| 6 | | | | 6 | |
| 5 | | | | 6, 5 | |
| 2 | | | | 6,5,2 | |
| 3 | | | | 6,5,2,3 | The first four symbols are placed on the stack. |
| + | 2 | 3 | 5 | 6,5,5 | Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed |
| 8 | 2 | 3 | 5 | 6,5,5,8 | Next 8 is pushed |
| * | 5 | 8 | 40 | 6, 5, 40 | Now a '**' is seen, so 8 and 5 are popped as $8 * 5 = 40$ is Pushed |

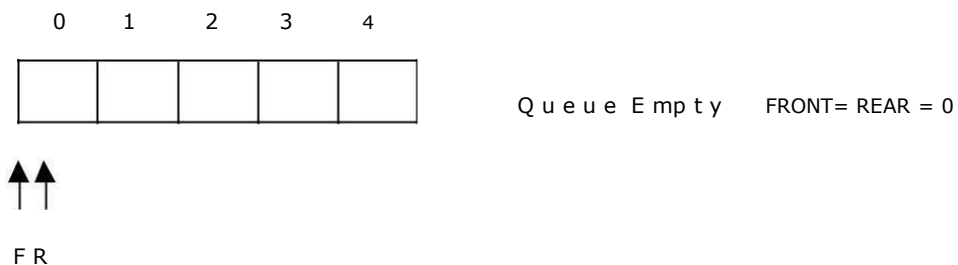
| | | | | | |
|---|----|----|-----|------------|---|
| + | 5 | 40 | 45 | 6, 45 | Next, a '+' is seen, so 40 and 5 are popped and $40 + 5 = 45$ is pushed |
| 3 | 5 | 40 | 45 | 6, 45, 3 | Now, 3 is pushed |
| + | 45 | 3 | 48 | 6, 48 | Next, '+' pops 3 and 45 and pushes $45 + 3 = 48$ is pushed |
| * | 6 | 48 | 288 | 288 | Finally, a '*' is seen and 48 and 6 are popped, the result $6 * 48 = 288$ is pushed |

Basic Queue Operations:

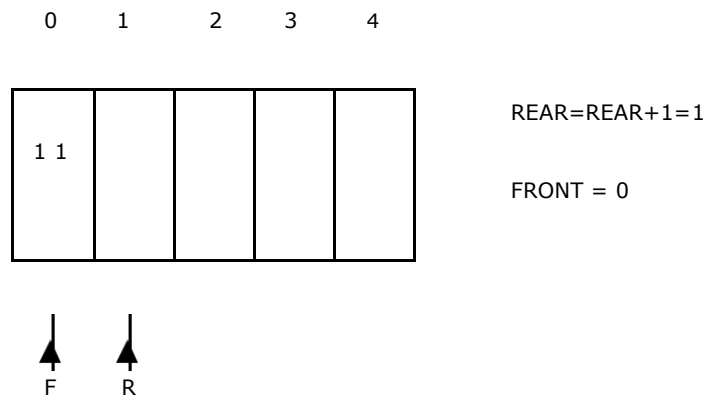
A queue is a data structure that is best described as "first in, first out". A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. A real world example of a queue is people waiting in line at the bank. As each person enters the bank, he or she is "enqueued" at the back of the line. When a teller becomes available, they are "dequeued" at the front of the line.

Representation of a Queue using Array:

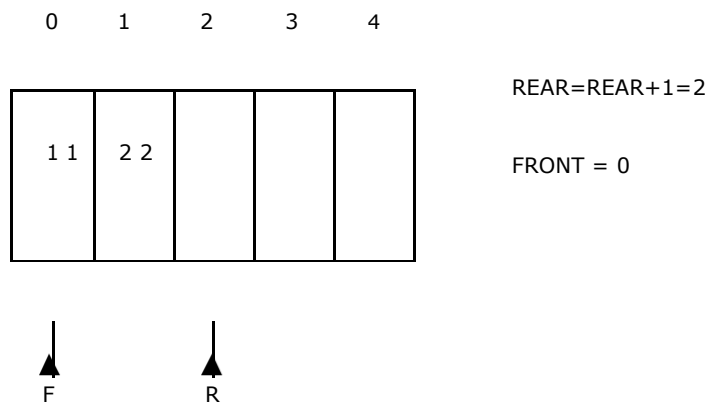
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



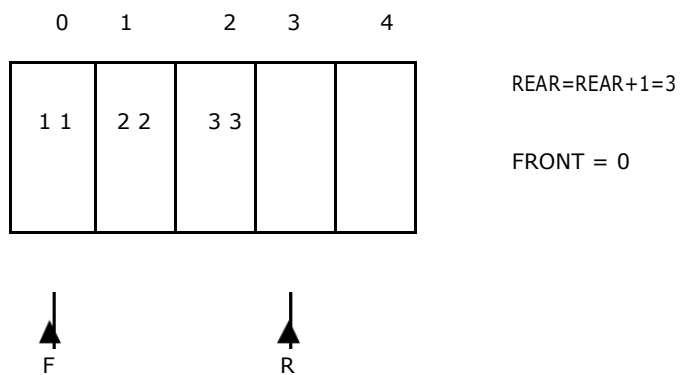
Now, insert 11 to the queue. Then queue status will be:



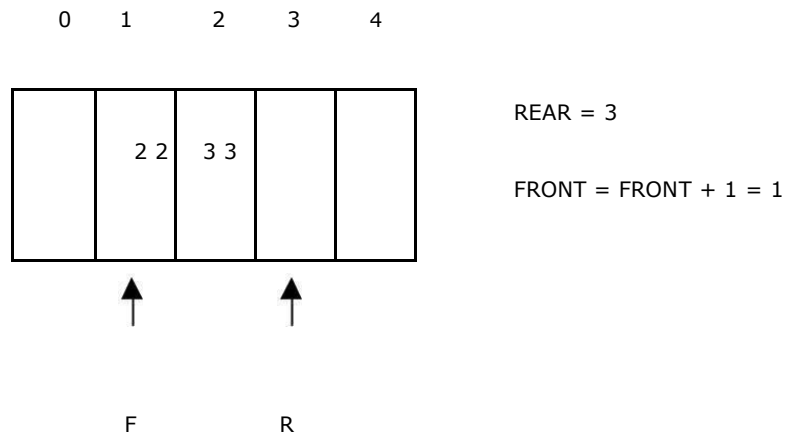
Next, insert 22 to the queue. Then the queue status is:



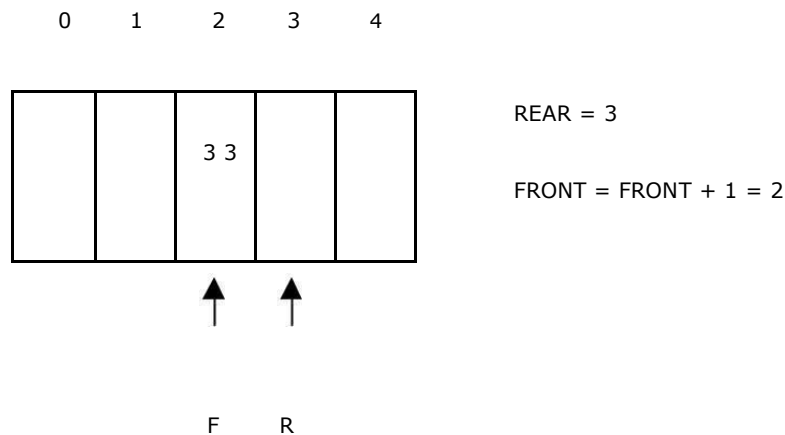
Again insert another element 33 to the queue. The status of the queue is:



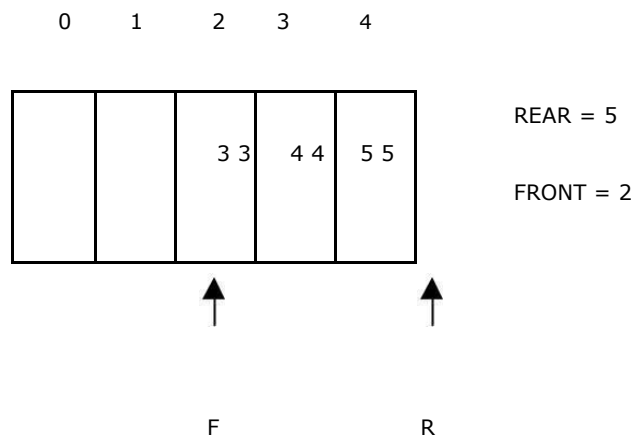
Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



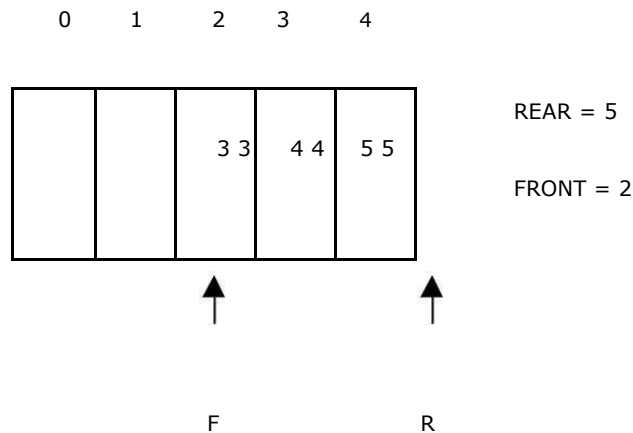
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



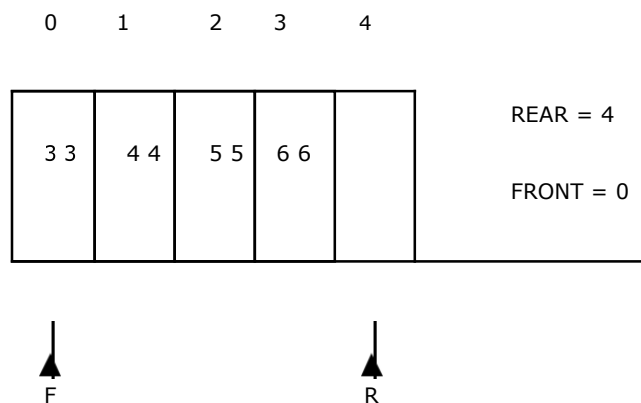
Now, insert new elements 44 and 55 into the queue. The queue status is:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue**.

Procedure for Queue operations using array:

In order to create a queue we require a one dimensional array $Q(1:n)$ and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and *rear* always points to the last element in the queue. Thus, $front = rear$ if and only if there are no elements in the queue. The initial condition then is $front = rear = 0$.

The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insertQ(): inserts an element at the end of queue Q.
2. deleteQ(): deletes the first element of Q.
3. displayQ(): displays the elements in the queue.

Source Code:

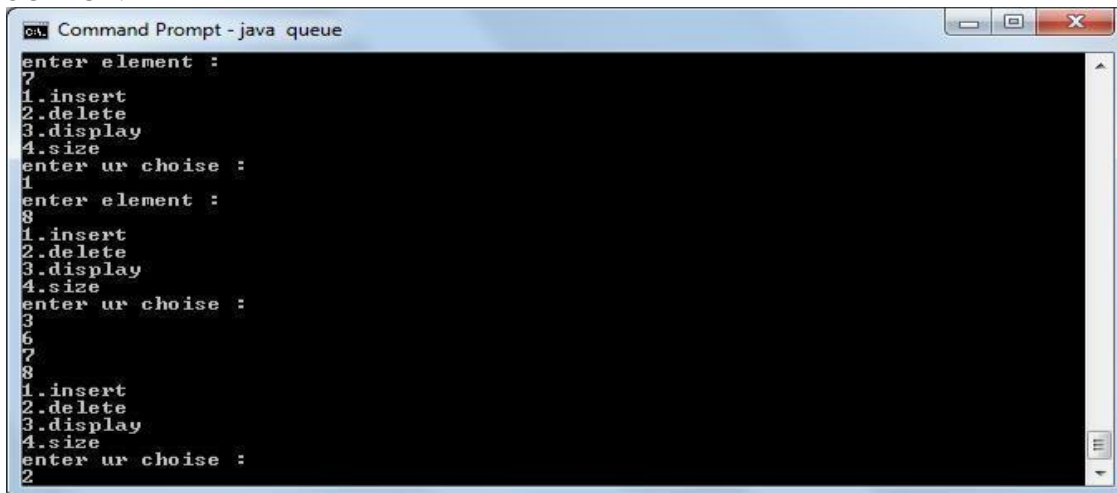
Queue ADT using array

```
import java.util.*;
class queue
{
    int front,rear;
    int que[];
    int max,count=0;
    queue(int n)
    {
        max=n;
        que=new int[max];
        front=rear=-1;
    }
    boolean isfull()
    {
        if(rear==(max-1))
            return true;
        else
            return false;
    }
    boolean isempty()
    {
        if(front==-1)
            return true;
        else
            return false;
    }
}
```

```
void insert(int n)
{
    if(isfull())
        System.out.println("list is full");
    else
    {
        rear++;
        que[rear]=n;
        if(front== -1)
            front=0;
        count++;
    }
}
int delete()
{
    int x;
    if(isempty())
        return -1;
    else
    {
        x=que[front];
        que[front]=0;
        if(front==rear)
            front=rear=-1;
        else
            front++;
        count--;
    }
    return x;
}
void display()
{
    if(isempty())
        System.out.println("queue is empty");
    else
        for(int i=front;i<=rear;i++)
            System.out.println(que[i]);
}
int size()
{
    return count;
}
public static void main(String args[])
{
    int ch;
    Scanner s=new Scanner(System.in);
    System.out.println("enter limit");
    int n=s.nextInt();
    queue q=new queue(n);
    do
    {
```

```
System.out.println("1.insert");
System.out.println("2.delete");
System.out.println("3.display");
System.out.println("4.size");
System.out.println("enter ur choise :");
ch=s.nextInt();
switch(ch)
{
    case 1:System.out.println("enter element :");
        int n1=s.nextInt();
        q.insert(n1);
        break;
    case 2:int c1=q.delete();
        if(c1>0)
            System.out.println("deleted element is :"+c1);
        else
            System.out.println("can't delete");
        break;
    case 3:q.display();
        break;
    case 4:System.out.println("queue size is "+q.size());
        break;
}
}
while(ch!=0);
}
```

OUTPUT:



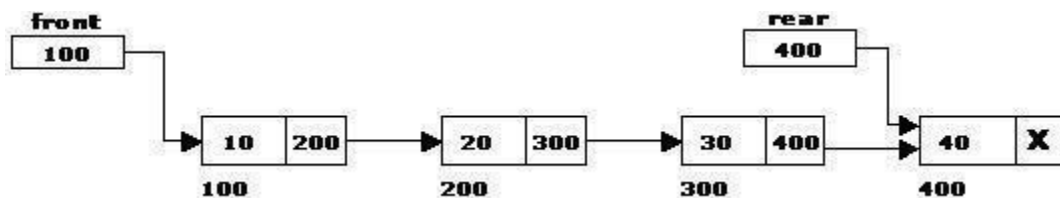
```
Command Prompt - java queue
enter element :
7
1.insert
2.delete
3.display
4.size
enter ur choise :
1
enter element :
8
1.insert
2.delete
3.display
4.size
enter ur choise :
3
6
7
8
1.insert
2.delete
3.display
4.size
enter ur choise :
2
```

```

Command Prompt - java queue
3.display
4.size
enter ur choise :
3
8
1.insert
2.delete
3.display
4.size
enter ur choise :
2
deleted element is :8
1.insert
2.delete
3.display
4.size
enter ur choise :
3
queue is empty
1.insert
2.delete
3.display
4.size
enter ur choise :

```

Linked List Implementation of Queue: We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers front and rear for our linked queue implementation.



Source Code:

```

Queue ADT USING LINKED LIST
import java.io.*;
class Qlnk
{
    Qlnk front,rear,next;
    int data;
    Qlnk()
    {
        data=0;
        next=null;
    }
    Qlnk(int d)
    {

```

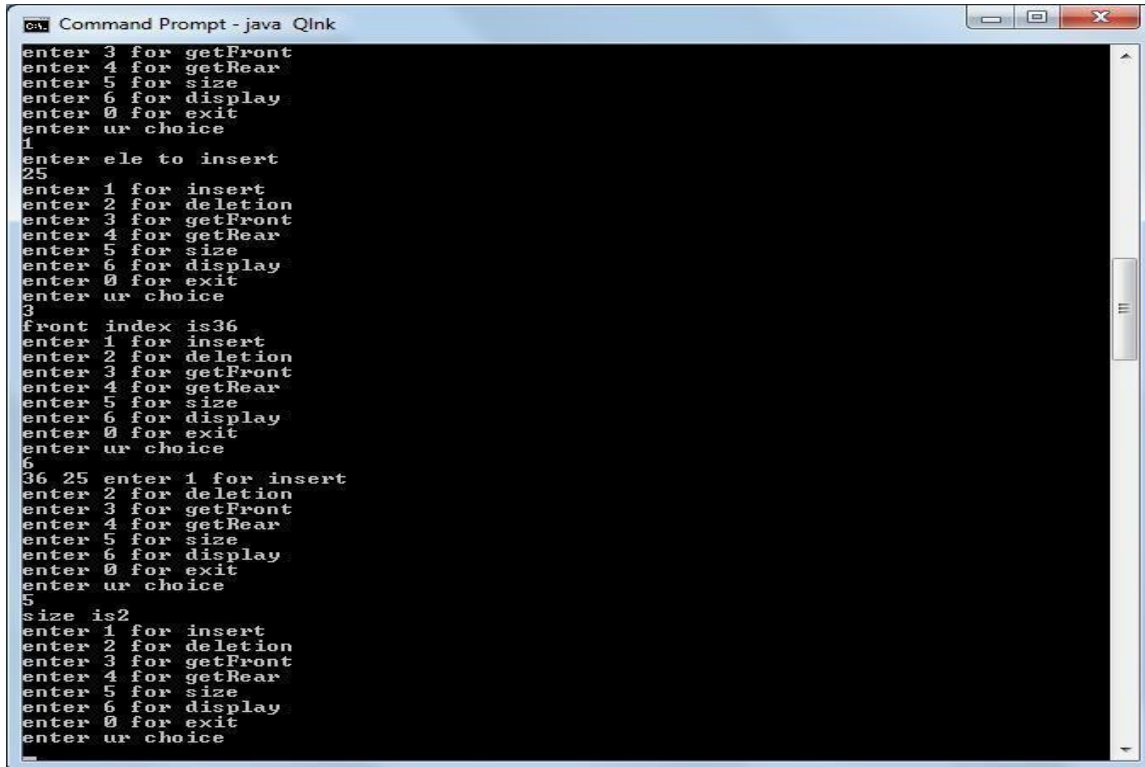


```
data=d;
next=null;
}
Qlnk getFront()
{
return front;
}
Qlnk getRear()
{
return rear;
}
void insertelm(int item)
{
Qlnk nn;
nn=new Qlnk(item);
if(isEmpty())
{
front=rear=nn;
}
else
{
rear.next=nn;
rear=nn;
}
}
int delelm()
{
if(isEmpty())
{
System.out.println("deletion failed");
return -1;
}
else
{
int k=front.data;
if(front!=rear)
front=front.next;
else
rear=front=null;
return k;
}
}
boolean isEmpty()
{
if(rear==null)
return true;
else
```

```
return false;
}
int size()
{
    Qlnk ptr;
    int cnt=0;
    for(ptr=front;ptr!=null;ptr=ptr.next)
        cnt++;
    return cnt;
}
void display()
{
    Qlnk ptr;
    if(!isEmpty())
    {
        for(ptr=front;ptr!=null;ptr=ptr.next)
            System.out.print(ptr.data+" ");
    }
    else
        System.out.println("q is empty");
}
public static void main(String arr[])throws Exception
{
    BufferedReader br=new BufferedReader(new
    InputStreamReader(System.in)); Qlnk m=new Qlnk();
    int ch;
    do
    {
        System.out.println("enter 1 for insert");
        System.out.println("enter 2 for deletion");
        System.out.println("enter 3 for getFront");
        System.out.println("enter 4 for getRear");
        System.out.println("enter 5 for size");
        System.out.println("enter 6 for display");
        System.out.println("enter 0 for exit");
        System.out.println("enter ur choice");
        ch=Integer.parseInt(br.readLine());
        switch(ch)
        {
            case 1: System.out.println("enter ele to insert");
                int item=Integer.parseInt(br.readLine());
                m.insertelm(item);break;
            case 2: int k=m.delelm();
                System.out.println("deleted ele is "+k);break;
            case 3: System.out.println("front index is "+(m.getFront()).data);break;
            case 4: System.out.println("rear index is "+(m.getRear()).data);break;
            case 5: System.out.println("size is "+m.size());break;
```

```
case 6:m.display();break;
}
}while(ch!=0);
}
}
```

OUTPUT:



```
Command Prompt - java QLink
enter 3 for getFront
enter 4 for getRear
enter 5 for size
enter 6 for display
enter 0 for exit
enter ur choice
1
enter ele to insert
25
enter 1 for insert
enter 2 for deletion
enter 3 for getFront
enter 4 for getRear
enter 5 for size
enter 6 for display
enter 0 for exit
enter ur choice
3
front index is 36
enter 1 for insert
enter 2 for deletion
enter 3 for getFront
enter 4 for getRear
enter 5 for size
enter 6 for display
enter 0 for exit
enter ur choice
6
36 25 enter 1 for insert
enter 2 for deletion
enter 3 for getFront
enter 4 for getRear
enter 5 for size
enter 6 for display
enter 0 for exit
enter ur choice
5
size is 2
enter 1 for insert
enter 2 for deletion
enter 3 for getFront
enter 4 for getRear
enter 5 for size
enter 6 for display
enter 0 for exit
enter ur choice
```

Applications of Queues:

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

Disadvantages of Linear Queue:

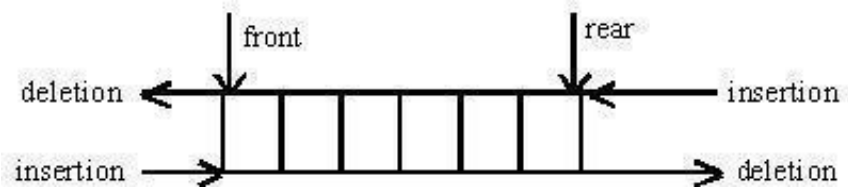
There are two problems associated with linear queue. They are:

1. Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
2. Signaling queue full: even if the queue is having vacant position.

DEQUE(Double Ended Queue)

A **double-ended queue (dequeue)**, often abbreviated to **deque**, pronounced *deck*) generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail). It is also often called a **head-tail linked list**. Like an ordinary queue, a double-ended queue is a data structure it supports the following operations: enq_front, enq_back, deq_front, deq_back, and empty. Dequeue can behave like a queue by using only enq_front and deq_front, and behaves like a stack by using only enq_front and deq_rear.

The DeQueue is represented as follows.



DEQUE can be represented in two ways they are

- 1) Input restricted DEQUE (IRD)
- 2) output restricted DEQUE (ORD)

The output restricted DEQUE allows deletions from only one end and input restricted DEQUE allow insertions at only one end. The DEQUE can be constructed in two ways they are

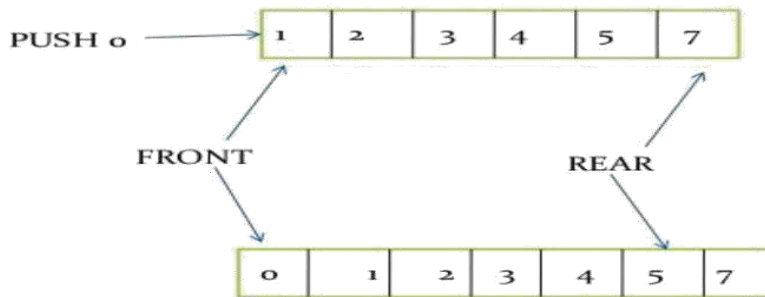
- 1) Using array
- 2) Using linked list

Operations in DEQUE

1. Insert element at back
2. Insert element at front
3. Remove element at front
4. Remove element at back

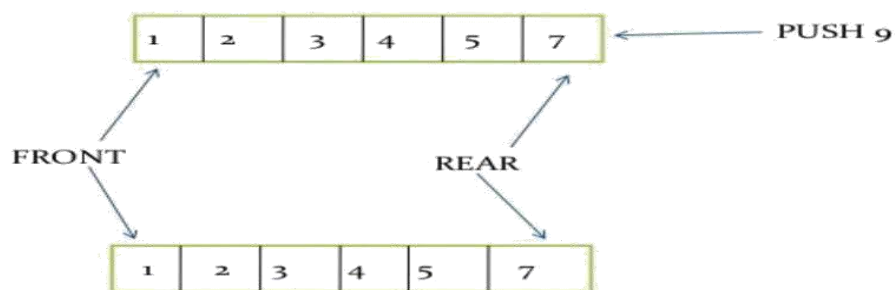
Insert_front

- `insert_front()` is a operation used to push an element into the front of the *Deque*.



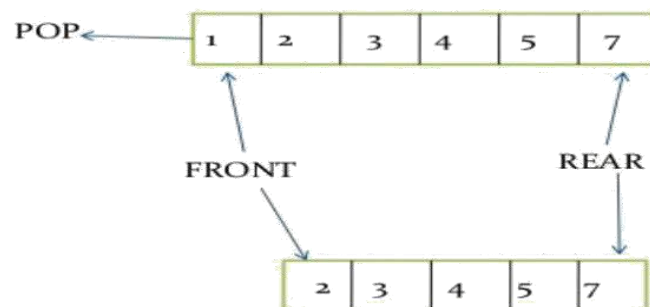
Insert_back

- `insert_back()` is a operation used to push an element at the back of a *Deque*.



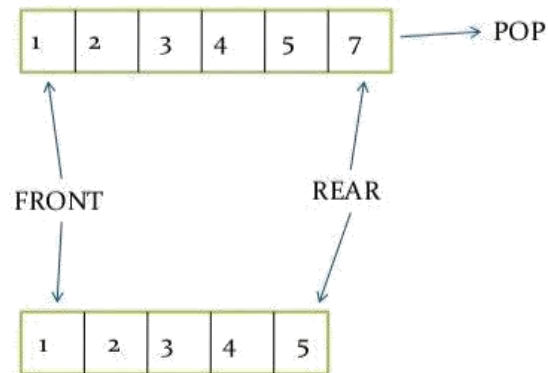
Remove_front

- `remove_front()` is a operation used to pop an element on front of the *Deque*.



Remove_back

- `remove_front()` is a operation used to pop an element on front of the *Deque*.



Applications of DEQUE:

1. The A-Steal algorithm implements task scheduling for several processors (multiprocessor scheduling).
2. The processor gets the first element from the deque.
3. When one of the processor completes execution of its own threads it can steal athread from another processor.
4. It gets the last element from the deque of another processor and executes it.

SOURCE CODE: DEQUE USING LINKED LIST GO TO 8C PROGRAM IN

LAB RECORD DEQUE USING ARRAYS

```
class ArrayDeque
{
    private int maxSize;
    private Object[] que;
    private int first;
    private int last;
    private int count; // current number of items in
    deque public ArrayDeque(int s) // constructor
    {
        maxSize = s;
        que = new Object[maxSize];
        first = last = -1;
    }
}
```

```
        count = 0;
    }
    public void insertLast(Object item)
    {
        if(count == maxSize)
        {
            System.out.println("Deque is full");
            return;
        }
        last = (last+1) % maxSize;
        que[last] = item;
        if(first == -1 && last == 0)
            first = 0;
        count++;
    }
    public Object deleteLast()
    {
        if(count == 0)
        {
            System.out.println("Deque is empty");
            return(' ');
        }
        Object item = que[last];
        que[last] = ' ';
        if(last > 0)
            last = (last-1) % maxSize;
        count--;
        if(count == 0)
            first = last = -1;
        return(item);
    }
    public void insertFirst(Object item)
    {
        if(count == maxSize)
        {
            System.out.println("Deque is full"); return;
        }
        if(first > 0)
            first = (first-1) % maxSize;
        else if(first == 0)
            first = maxSize-1;
        que[first] = item;
        count++;
    }
    public Object deleteFirst()
    {
        if(count == 0)
```

```

        {
            System.out.println("Deque is empty"); return(' ');
        }
        Object item = que[first];
        que[first] = ' ';
        if(first == maxSize-1)
            first = 0;
        else
            first = (first+1) % maxSize;
        count--;
        if(count == 0)
            first = last = -1;
        return(item);
    }
    void display()
    {
        System.out.println("-----");
        System.out.print("first:"+first + " , last:"+ last);
        System.out.println(", count: " + count);
        System.out.println(" 0 1 2 3 4 5");
        System.out.print("Deque: ");
        for( int i=0; i<maxSize; i++ )
            System.out.print(que[i]+ " ");
        System.out.println("\n ----- ");
    }
    public boolean isEmpty() // true if queue is empty
    {
        return (count == 0);
    }
    public boolean isFull() // true if queue is full
    {
        return (count == maxSize);
    }
}
class ArrayDequeDemo
{
    public static void main(String[] args)
    {
        ArrayDeque q1 = new ArrayDeque(6); // queue holds a max of 6
        items q1.insertLast('A'); /* (a) */
        q1.insertLast('B');
        q1.insertLast('C');
        q1.insertLast('D');
        System.out.println("deleteFirst():"+q1.deleteFirst());
        q1.display();
        q1.insertLast('E'); /* (b) */
        q1.display();
    }
}

```



```

        /* (c) */
        System.out.println("deleteLast():"+q1.deleteLast());
        System.out.println("deleteLast():"+q1.deleteLast());
        q1.display();
        q1.insertFirst('P');
        q1.insertFirst('Q'); /* (d) */
        q1.insertFirst('R');
        q1.display();
        q1.deleteFirst();
        q1.display(); /* (e) */
        q1.insertFirst('X');
        q1.display(); /* (f) */
        q1.insertLast('Y');
        q1.display(); /* (g) */
        q1.insertLast('Z');
        q1.display(); /* (h) */
    }
}

```

OUTPUT:

```

C:\g\ads>javac ArrayDequeDemo.java
C:\g\ads>java ArrayDequeDemo
deleteFirst():A
-----
first:1, last:3, count: 3
0 1 2 3 4 5
Deque: B C D null null
-----
first:1, last:4, count: 4
0 1 2 3 4 5
Deque: B C D E null
-----
deleteLast():E
deleteLast():D
-----
first:1, last:2, count: 2
0 1 2 3 4 5
Deque: B C null
-----
first:4, last:2, count: 5
0 1 2 3 4 5
Deque: P B C R Q
-----
first:5, last:2, count: 4
0 1 2 3 4 5
Deque: P B C Q
-----
first:4, last:2, count: 5
0 1 2 3 4 5
Deque: P B C X Q
-----
first:4, last:3, count: 6
0 1 2 3 4 5
Deque: P B C Y X Q
Deque is full
first:4, last:3, count: 6
0 1 2 3 4 5
Deque: P B C Y X Q
-----
C:\g\ads>

```

Circular Queue:

Circular queue is a linear data structure. It follows FIFO principle. In circular queue the last node is connected back to the first node to make a circle.

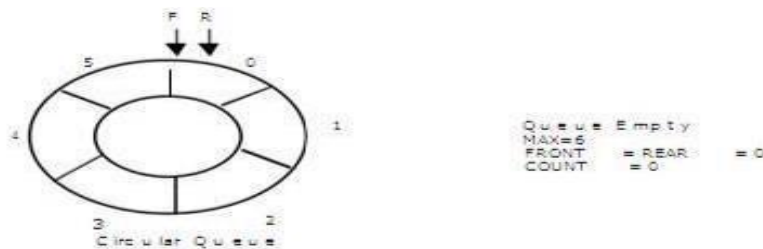
1. Circular linked list follow the First In First Out principle
2. Elements are added at the rear end and the elements are deleted at front end of the queue
3. Both the front and the rear pointers points to the beginning of the array.
4. It is also called as "Ring buffer".
5. Items can inserted and deleted from a queue in $O(1)$

time. Circular Queue can be created in three ways they are

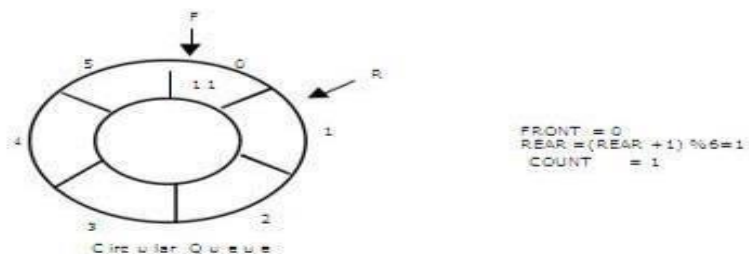
1. Using single linked list
2. Using double linked list
3. Using arrays

Representation of Circular Queue:

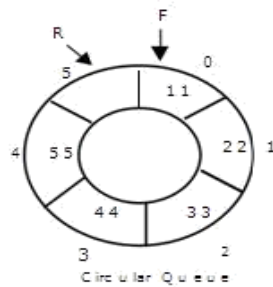
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Now, insert 11 to the circular queue. Then circular queue status will be:



Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:

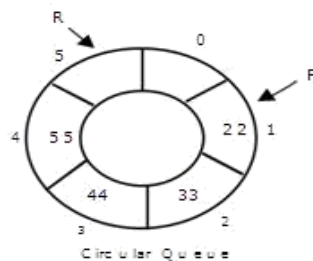


```

FRONT = 0, REAR = 5
REAR = REAR % 6 = 5
COUNT = 5

```

Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:

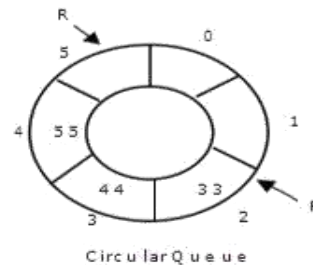


```

FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4

```

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:

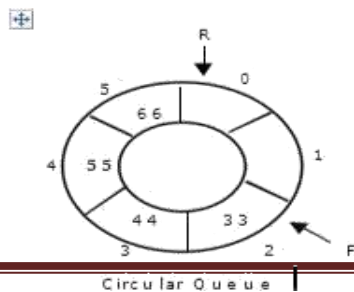


```

FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3

```

Again, insert another element 66 to the circular queue. The status of the circular queue is:

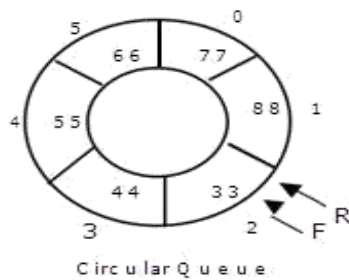


```

FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4

```

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



FRONT = 2, REAR = 2
 REAR = REAR % 6 = 2
 COUNT = 6

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

Source Code:

```
import java.util.*;
class CirQue
{
    int front,rear,next=0;
    int que[];
    int max,count=0;
    CirQue(int n)
    {
        max=n;
        que=new int[max];
        front=rear=-1;
    }
    boolean isfull()
    {
        if(front==(rear+1)%max)
            return true;
        else
            return false;
    }
    boolean isempty()
    {
        if(front==-1&&rear==-1)
            return true;
        else
            return false;
    }
}
```

```
int delete()
{
    if(isempty())
    {
        return -1;
    }
    else
    {
        count --;
        int x=que[front];
        if(front==rear)
            front=rear=-1;
        else
        {
            next=(front+1)%max;
            front=next;
        }
        return x;
    } }
void insert(int item)
{
    if(isempty())
    {
        que[++rear]=item;
        front=rear;
        count ++;
    }
    else if(!isfull())
    {
        next=(rear+1)%max;
        if(next!=front)
        {
            que[next]=item;
            rear=next;
        }
        count ++;
    }
    else
        System.out.println("q is full");
}
```

```
void display()
{
    if(isempty())
        System.out.println("queue is empty");
    else
        next=(front)%max;
    while(next<=rear)
    {
        System.out.println(que[next]); next++;
    }
}
int size()
{
    return count;
}
public static void main(String args[])
{
    int ch;
    Scanner s=new Scanner(System.in);
    System.out.println("enter limit");
    int n=s.nextInt();
    CirQue q=new CirQue(n);
    do
    { System.out.println("1.insert");
      System.out.println("2.delete");
      System.out.println("3.display");
      System.out.println("4.size");
      System.out.println("enter ur choice
      :"); ch=s.nextInt();
      switch(ch)
      { case 1: System.out.println("enter element :"); int
        n1=s.nextInt();
        q.insert(n1);
        break;
        case 2: int c1=q.delete();
        if(c1>0)
        System.out.println("deleted element is :"+c1);
        else
        System.out.println("can't delete");
        break;
        case 3: q.display();
        break;
        case 4: System.out.println("queue size is
        "+q.size()); break;
      }
    }
    while(ch!=0);
}
```

```

    }
}

```

OUTPUT:

```

C:\> java CirQue
Enter limit
5
1.insert
2.delete
3.display
4.size
enter ur choice :
1
enter element :
3
1.insert
2.delete
3.display
4.size
enter ur choice :
1
enter element :
4
1.insert
2.delete
3.display
4.size
enter ur choice :
1
enter element :
5
1.insert
2.delete
3.display
4.size
enter ur choice :
2
deleted element is :3
1.insert
2.delete
3.display
4.size
enter ur choice :
2
deleted element is :4
1.insert
2.delete
3.display
4.size
enter ur choice :
4
queue size is 1

```

Priority queue ADT:

Priority**Queue****DEFINITION:**

A priority queue is a collection of zero or more elements. Each element has a priority or value.

1. Unlike the queues, which are FIFO structures, the order of deleting from a priority queue is determined by the element priority.
2. Elements are removed/deleted either in increasing or decreasing order of priority rather than in the order in which they arrived in the queue.

There are two types of priority queues:

Min priority queue: Collection of elements in which the items can be inserted arbitrarily, but only smallest element can be removed.

Max priority queue: Collection of elements in which insertion of items can be in any order but only largest element can be removed.

In priority queue, the elements are arranged in any order and out of which only the smallest or

largest element allowed to delete each time.

The implementation of priority queue can be done using arrays or linked list. The data structure **heap** is used to implement the priority queue effectively.

APPLICATIONS:

1. The typical example of priority queue is scheduling the jobs in operating system. Typically OS allocates priority to jobs. The jobs are placed in the queue and position of the job in priority queue determines their priority. In OS there are 3 jobs- real time jobs, foreground jobs and background jobs. The OS always schedules the real time jobs first. If there is no real time jobs pending then it schedules foreground jobs. Lastly if no real time and foreground jobs are pending then OS schedules the background jobs.
2. In network communication, the manage limited bandwidth for transmission the priority queue is used.
3. In simulation modeling to manage the discrete events the priority queue is used.

OPERATIONS ON PRIORITY QUEUE

1. Find an element
2. Insert a new element
3. Remove or delete an element

The abstract data type specification for a max priority queue is given below. The specification for a min priority queue is the same as ordinary queue except while deletion, find and remove the element with minimum priority

ABSTRACT DATA TYPE(ADT):

Abstract data type maxPriorityQueue

{

Instances

Finite collection of elements, each has a priority

Operations empty():return true iff the queue is

empty size() :return number of elements in the
queue

top() :return element with maximum priority

del() :remove the element with largest priority from the

queue insert(x): insert the element x into the queue }

SOURCE CODE:

LinkedPriorityQueueDemo.java

```
class Node
{
    String data; // data item
    int prn; // priority number (minimum has highest priority)
    Node next; // "next" refers to the next node
    Node( String str, int p ) // constructor
    {
        data = str;
        prn = p;
    } // "next" is automatically set to null
}
class LinkedPriorityQueue
{
    Node head; // "head" refers to first node
    public void insert(String item, int pkey) // insert item after pkey
    {
        Node newNode = new Node(item, pkey); // create new node
        int k;
        if( head == null ) k = 1;
        else if( newNode.prn < head.prn ) k = 2;
        else k = 3;
        switch( k )
        {
            case 1: head = newNode; // Q is empty, add head node
                    head.next = null;
                    break;
            case 2: Node oldHead = head; // add one item before head
                    head = newNode;
                    newNode.next = oldHead;
                    break;
            case 3: Node p = head; // add item before a node
                    Node prev = p;
                    Node nodeBefore = null;
                    while( p != null )
                    {
                        if( newNode.prn < p.prn )
                        {
                            nodeBefore = p;
                            break;
                        }
                        else
                        {
                            prev = p; // save previous node of current node
                            p = p.next; // move to next node
                        }
                    }
                    if( nodeBefore != null )
                        nodeBefore.next = newNode;
                    else
                        head = newNode;
                    break;
        }
    }
}
```

```
        }
        } // end of while
        newNode.next = nodeBefore;
        prev.next = newNode;
    } // end of switch
} // end of insert() method
public Node delete()
{
    if( isEmpty() )
    {
        System.out.println("Queue is empty");
        return null;
    }
    else
    {
        Node tmp = head;
        head = head.next;
        return tmp;
    }
}
public void displayList()
{
    Node p = head; // assign address of head to p
    System.out.print("\nQueue: ");
    while( p != null ) // start at beginning of list until end of list
    {
        System.out.print(p.data+"(" +p.prn+ ")" + " ");
        p = p.next; // move to next node
    }
    System.out.println();
}
public boolean isEmpty() // true if list is empty
{
    return (head == null);
}
public Node peek() // get first item
{
    return head;
}
}
class LinkedPriorityQueueDemo
{
    public static void main(String[] args)
    {
        LinkedPriorityQueue pq = new LinkedPriorityQueue(); // create new queue list
        Node item;
        pq.insert("Babu", 3);
        pq.insert("Nitin", 2);
    }
}
```

```

pq.insert("Laxmi", 2);
pq.insert("Kim", 1);
pq.insert("Jimmy", 3);
pq.displayList();
item = pq.delete();
if( item != null )
System.out.println("delete():" + item.data + "(" + item.prn + ")");
pq.displayList();
pq.insert("Scot", 2);
pq.insert("Anu", 1);
pq.insert("Lehar", 4);
pq.displayList();
} }

```

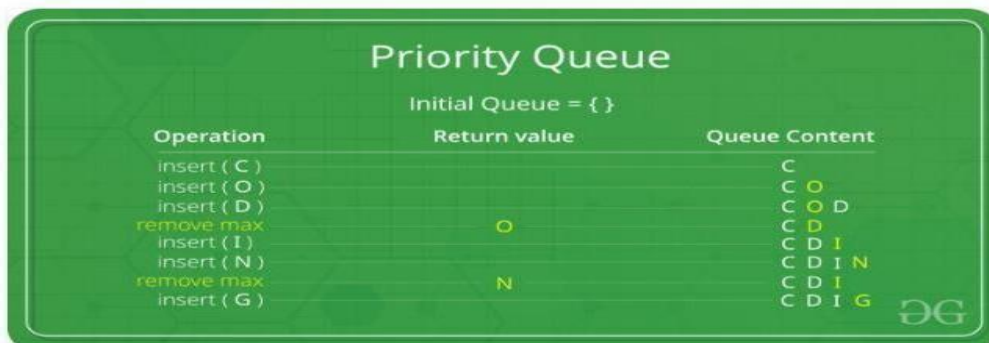
OUTPUT:

```

C:\> javac LinkedDequeDemo.java
E:\g\ads> java LinkedDequeDemo
Deque is empty
removeFirst():null
Deque: [ C B A ]
getFirst():C
getLast():E
Deque: [ C B A D E ]
removeFirst():C
removeLast():E
Deque: [ B A D ]
size():3

E:\g\ads> javac LinkedPriorityQueueDemo.java
E:\g\ads> java LinkedPriorityQueueDemo
Queue: Kim<1> Nitin<2> Laxmi<2> Babu<3> Jimmy<3>
delete():Kim<1>
Queue: Nitin<2> Laxmi<2> Babu<3> Jimmy<3>
Queue: Anu<1> Nitin<2> Laxmi<2> Scot<2> Babu<3> Jimmy<3> Lehar<4>
E:\g\ads>

```

DIAGRAMMATICAL REPRESENTATION OF PRIORITY QUEUE

A typical priority queue supports following operations.

insert(item, priority): Inserts an item with given priority.

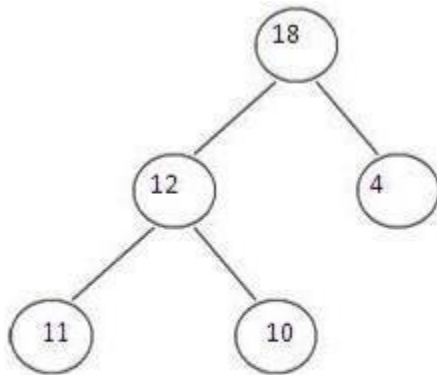
getHighestPriority(): Returns the highest priority item.

deleteHighestPriority(): Removes the highest priority item.

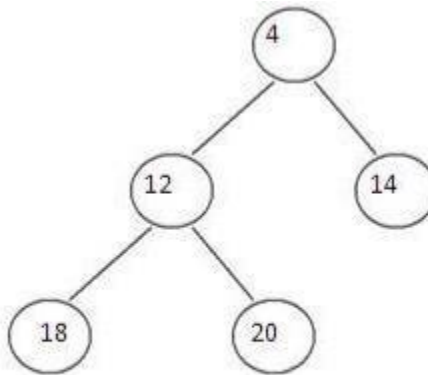
HEAPS

Heap is a tree data structure denoted by either a max heap or a min heap.

A max heap is a tree in which value of each node is greater than or equal to value of its children nodes. A min heap is a tree in which value of each node is less than or equal to value of its children nodes.



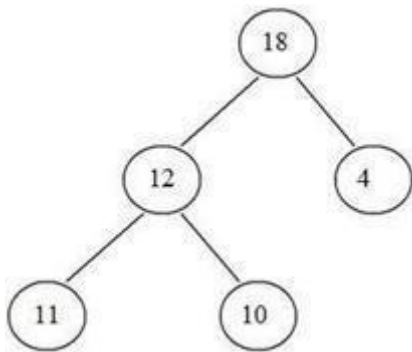
Max heap



Min heap

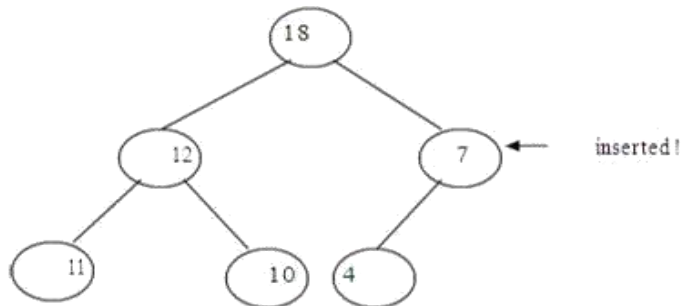
Insertion of element in the Heap:

Consider a max heap as given below:

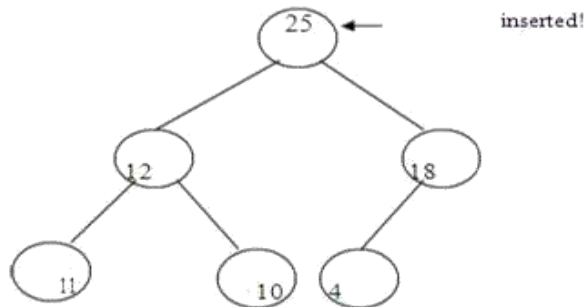


Now if we want to insert 7. We cannot insert 7 as left child of 4. This is because the max heap has a property that value of any node is always greater than the parent nodes. Hence 7 will bubble up 4 will be left child of 7.

Note: When a new node is to be inserted in complete binary tree we start from bottom and from left child on the current level. The heap is always a complete binary tree.



If we want to insert node 25, then as 25 is greatest element it should be the root. Hence 25 will bubble up and 18 will move down.



The insertion strategy just outlined makes a single bubbling pass from a leaf toward the root. At each level we do (1) work, so we should be able to implement the strategy to have complexity $O(\text{height}) = O(\log n)$.

void Heap::insert(int item)

```

{
    int temp;    //temp node starts at leaf and moves
                up. temp=++size;
    while(temp!=1 && heap[temp/2]<item) //moving element down {

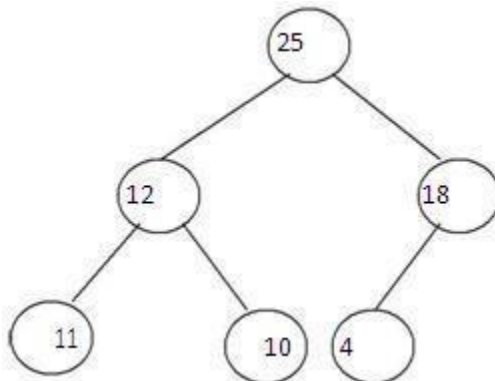
        H[temp] = H[temp/2]; temp=temp/2;
        //finding the parent
    }
    H[temp]=item;
}

```

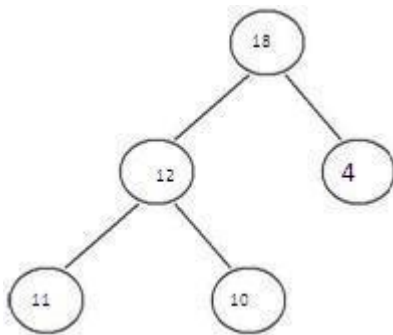
Deletion of element from the heap:

For deletion operation always the maximum element is deleted from heap. In Max heap the maximum element is always present at root. And if root element is deleted then we need to reheapify the tree.

Consider a Max heap

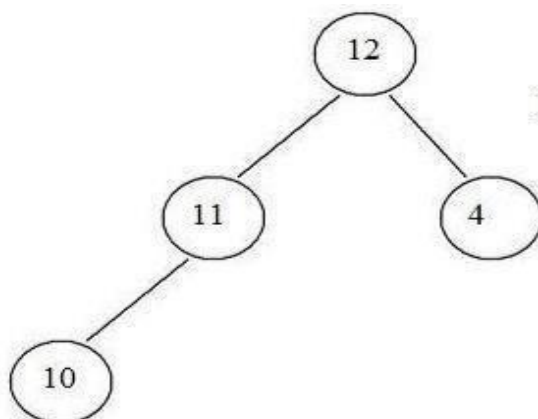


Delete root element:25, Now we cannot put either 12 or 18 as root node and that should be greater than all its children elements.



Now we cannot put 4 at the root as it will not satisfy the heap property. Hence we will bubble up 18 and place 18 at root, and 4 at position of 18.

If 18 gets deleted then 12 becomes root and 11 becomes parent node of 10.



Make tree a complete binary tree.

Thus deletion operation can be performed. The time complexity of deletion operation is $O(\log n)$.

1. Remove the maximum element which is present at the root. Then a hole is created at the root.
2. Now reheapify the tree. Start moving from root to children nodes. If any maximum element is found then place it at root. Ensure that the tree is satisfying the heap property or not.
3. Repeat the step 1 and 2 if any more elements are to be deleted.

```
void heap::delet(int item)
{

int item,
temp;
if(size==0)

cout<<"Heap is empty\n"; else
{

//remove the last elemnt and
reheapify item=H[size--];

//item is placed at root temp=1;
child=2;
while(child<=size)
{
if(child<size && H[child]<H[child+1])
child++; if(item>=H[child])
break;
H[temp]=H[child]
; temp=child;
child=child*2;
}

//place the largest item at root
H[temp]=item;
}
```

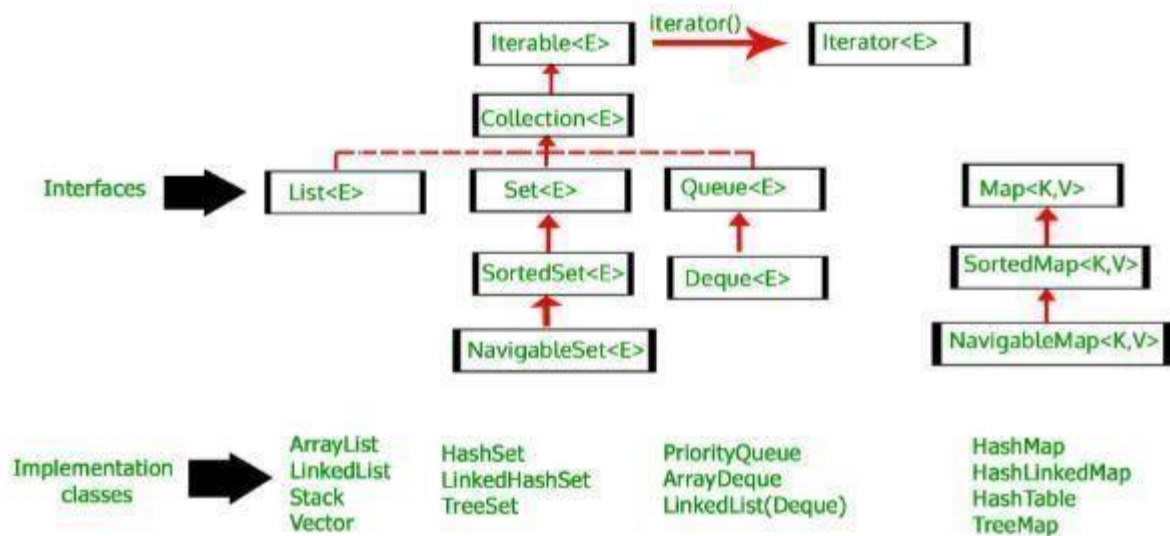
Applications Of Heap:

1. Heap is used in sorting algorithms. One such algorithm using heap is known as heap sort.
2. In priority queue implementation the heap is used.

ArrayList in Java

ArrayList is a part of collection framework and is present in java.util package. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

1. ArrayList inherits AbstractList class and implements List interface.
2. ArrayList is initialized by a size, however the size can increase if collection grows or shrunk if objects are removed from the collection.
3. Java ArrayList allows us to randomly access the list.
4. ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases (see this for details).
5. ArrayList in Java can be seen as similar to vector in C++.



Constructors in Java ArrayList:

1. ArrayList(): This constructor is used to build an empty array list
2. ArrayList(Collection c): This constructor is used to build an array list initialized with the elements from collection c
3. ArrayList(int capacity): This constructor is used to build an array list with initial capacity being specified

Methods in Java ArrayList:

1. forEach(Consumer action): Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
2. retainAll(Collection c): Retains only the elements in this list that are contained in the specified collection.

3. `removeIf(Predicate filter)`: Removes all of the elements of this collection that satisfy the given predicate.
4. `contains(Object o)`: Returns true if this list contains the specified element.
5. `remove(int index)`: Removes the element at the specified position in this list.
6. `remove(Object o)`: Removes the first occurrence of the specified element from this list, if it is present.
7. `get(int index)`: Returns the element at the specified position in this list.
8. `subList(int fromIndex, int toIndex)`: Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
9. `splitter()`: Creates a late-binding and fail-fast `Splitter` over the elements in this list.
10. `set(int index, E element)`: Replaces the element at the specified position in this list with the specified element.
11. `size()`: Returns the number of elements in this list.
12. `removeAll(Collection c)`: Removes from this list all of its elements that are contained in the specified collection.
13. `ensureCapacity(int minCapacity)`: Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
14. `listIterator()`: Returns a list iterator over the elements in this list (in proper sequence).
15. `listIterator(int index)`: Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
16. `isEmpty()`: Returns true if this list contains no elements.
17. `removeRange(int fromIndex, int toIndex)`: Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive.
18. `void clear()`: This method is used to remove all the elements from any list.
19. `void add(int index, Object element)`: This method is used to insert a specific element at a specific position `index` in a list.
20. `void trimToSize()`: This method is used to trim the capacity of the instance of the `ArrayList` to the list's current size.
21. `int indexOf(Object O)`: The index the first occurrence of a specific element is either returned, or -1 in case the element is not in the list.
22. `int lastIndexOf(Object O)`: The index the last occurrence of a specific element is either returned, or -1 in case the element is not in the list.
23. `Object clone()`: This method is used to return a shallow copy of an `ArrayList`.
24. `Object[] toArray()`: This method is used to return an array containing all of the elements in the list in correct order.
25. `Object[] toArray(Object[] O)`: It is also used to return an array containing all of the elements in this list in the correct order same as the previous method.

26. **boolean addAll(Collection C):** This method is used to append all the elements from a specific collection to the end of the mentioned list, in such a order that the values are returned by the specified collection's iterator.
27. **boolean add(Object o):** This method is used to append a specified element to the end of a list.
28. **boolean addAll(int index, Collection C):** Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list.

LinkedList in Java

Linked List are linear data structures where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. It also has few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access.

To store the elements in a linked list we use a doubly linked list which provides a linear data structure and also used to inherit an abstract class and implement list and deque interfaces.

In Java, LinkedList class implements the list interface. The LinkedList class also consists of various constructors and methods like other java collections.

Constructors for Java LinkedList:

1. **LinkedList():** Used to create an empty linked list.
2. **LinkedList(Collection C):** Used to create a ordered list which contains all the elements of a specified collection, as returned by the collection's iterator.

Methods for Java LinkedList:

1. **add(int index, E element):** This method Inserts the specified element at the specified position in this list.
2. **add(E e):** This method Appends the specified element to the end of this list.
3. **addAll(int index, Collection c):** This method Inserts all of the elements in the specified collection into this list, starting at the specified position.
4. **addAll(Collection c):** This method Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
5. **addFirst(E e):** This method Inserts the specified element at the beginning of this list.
6. **addLast(E e):** This method Appends the specified element to the end of this list.
7. **clear():** This method removes all of the elements from this list.
8. **clone():** This method returns a shallow copy of this LinkedList.
9. **contains(Object o):** This method returns true if this list contains the specified element.

10. **descendingIterator():** This method returns an iterator over the elements in this deque in reverse sequential order.
11. **element():** This method retrieves, but does not remove, the head (first element) of this list.
12. **get(int index):** This method returns the element at the specified position in this list.
13. **getFirst():** This method returns the first element in this list.
14. **getLast():** This method returns the last element in this list.
15. **indexOf(Object o):** This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
16. **lastIndexOf(Object o):** This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
17. **listIterator(int index):** This method returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
18. **offer(E e):** This method Adds the specified element as the tail (last element) of this list.
19. **offerFirst(E e):** This method Inserts the specified element at the front of this list.
20. **offerLast(E e):** This method Inserts the specified element at the end of this list.
21. **peek():** This method retrieves, but does not remove, the head (first element) of this list.
22. **peekFirst():** This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty.
23. **peekLast():** This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.
24. **poll():** This method retrieves and removes the head (first element) of this list.
25. **pollFirst():** This method retrieves and removes the first element of this list, or returns null if this list is empty.
26. **pollLast():** This method retrieves and removes the last element of this list, or returns null if this list is empty.
27. **pop():** This method Pops an element from the stack represented by this list.
28. **push(E e):** This method Pushes an element onto the stack represented by this list.
29. **remove():** This method retrieves and removes the head (first element) of this list.
30. **remove(int index):** This method removes the element at the specified position in this list.
31. **remove(Object o):** This method removes the first occurrence of the specified element from this list, if it is present.
32. **removeFirst():** This method removes and returns the first element from this list.
33. **removeFirstOccurrence(Object o):** This method removes the first occurrence of the specified element in this list (when traversing the list from head to tail).
34. **removeLast():** This method removes and returns the last element from this list.
35. **removeLastOccurrence(Object o):** This method removes the last occurrence of the specified element in this list (when traversing the list from head to tail).

- 36. set(int index, E element):** This method replaces the element at the specified position in this list with the specified element.

- 37. **size()**: This method returns the number of elements in this list.
- 38. **splititerator()**: This method Creates a late-binding and fail-fast Spliterator over the elements in this list.
- 39. **toArray()**: This method returns an array containing all of the elements in this list in proper sequence (from first to last element).
- 40. **toArray(T[] a)**: This method returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Java.util.Vector Class in Java

The Vector class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections.

- 1. Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
- 2. They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.
- 3. It extends **AbstractList** and implements **List** interfaces.

Constructor

- 1. **Vector()**: Creates a default vector of initial capacity is 10.
- 2. **Vector(int size)**: Creates a vector whose initial capacity is specified by size.
- 3. **Vector(int size, int incr)**: Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time that a vector is resized upward.
- 4. **Vector(Collection c)**: Creates a vector that contains the elements of collection c.

Important points regarding Increment of vector capacity

If increment is specified, Vector will expand according to it in each allocation cycle but if increment is not specified then vector's capacity get doubled in each allocation cycle. Vector defines three protected data member:

- 1. **int capacityIncrement**: Contains the increment value.
- 2. **int elementCount**: Number of elements currently in vector stored in it.
- 3. **Object elementData[]**: Array that holds the vector is stored in it.

METHODS IN VECTOR :

1. **boolean add(Object obj):** This method appends the specified element to the end of this vector.
2. **Syntax:** public boolean add(Object obj)
3. **Returns:** true if the specified element is added
4. successfully into the Vector, otherwise it returns false.
5. **Exception:** NA.

6. **void add(int index, Object obj):** This method inserts the specified element at the specified position in this Vector.

7. **Syntax:** public void add(int index, Object obj)
8. **Returns:** NA.
9. **Exception:** IndexOutOfBoundsException, method throws this exception
10. if the index (obj position) we are trying to access is out of range
11. (index size()).

12. **boolean addAll(Collection c)** This method appends all of the elements

in the specified Collection to the end of this Vector.

13. **Syntax:** public boolean addAll(Collection c)
14. **Returns:** Returns true if operation succeeded otherwise false.
15. **Exception:** NullPointerException thrown if collection is null.

16. **boolean addAll(int index, Collection c)** This method inserts all of the elements in the specified Collection into this Vector at the specified position.

17. **Syntax:** public boolean addAll(int index, Collection c)
18. **Returns:** true if this list changed as a result of the call.
19. **Exception:** IndexOutOfBoundsException -- If the index is out of range,
20. NullPointerException -- If the specified collection is null.

void clear() This method removes all of the elements from this vector.

Syntax: public void clear()

Returns: NA.

Exception: NA.

Object clone() This method returns a clone of this vector.

Syntax: public Object clone()

Returns: a clone of this ArrayList instance.

Exception: NA.

boolean contains(Object o): This method returns true if this vector contains the specified element.

Syntax: public boolean contains(object o)

Returns: true if the operation is succeeded otherwise false.

Exception: NA.

boolean isEmpty(): This method tests if this vector has no components.

Syntax: public boolean isEmpty()

Returns: true if vector is empty otherwise false.

Exception: NA.

int indexOf(Object o): This method returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.

Syntax: public int indexOf(Object o)

Returns: the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. **Exception:** NA.

Object get(int index): This method returns the element at the specified position in this Vector.

Syntax: public Object get(int index)

Returns: returns the element at specified positions .

Exception: IndexOutOfBoundsException -- if the index is out of range.

Java.Util Package-Stacks

The java.util.Stack class represents a last-in-first-out (LIFO) stack of objects. When a stack is first created, it contains no items.

In this class, the last element inserted is accessed first. Class declaration

Following is the declaration for java.util.Stack class:

```
public class  
Stack<E> extends  
Vector<E> Class  
constructors
```

| | |
|------|---|
| S.N. | Constructor & Description |
| 1 | Stack() This constructor creates an empty stack. |

Class methods

| | |
|------|---|
| S.N. | Method & Description |
| 1 | boolean empty() This method tests if this stack is empty. |
| 2 | E peek() This method looks at the object at the top of this stack without removing it from the stack. |
| 3 | E pop() This method removes the object at the top of this stack and returns that object as the value of this function. |
| 4 | E push(E item) This method pushes an item onto the top of this stack. |
| 5 | int search(Object o) This method returns the 1-based position where an object is on this stack. |

Java.util.Interfaces

The java.util.Interfaces contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

S.N. Interface & Description

1 Deque<E>

This is a linear collection that supports element insertion and removal at both ends.

2 Enumeration<E>

This is an object that implements the Enumeration interface generates a series of elements, one at a time.

3 EventListener

This is a tagging interface that all event listener interfaces must extend.

4 Formattable

This is the Formattable interface must be implemented by any class that needs to perform custom formatting using the 's' conversion specifier of Formatter.

5 Iterator<E>

This is an iterator over a collection.

6 Queue<E>

This is a collection designed for holding elements prior to processing.

Iterators IN JAVA.UTIL

Iterator' is an interface which belongs to collection framework. It allows us to traverse the collection, access the data element and remove the data elements of the collection. **java.util** package has **public interface Iterator** and contains three methods:

1. **boolean hasNext():** It returns true if Iterator has more element to iterate.
2. **Object next():** It returns the next element in the collection until the hasNext() method return true. This method throws 'NoSuchElementException' if there is no nextelement.
3. **void remove():** It removes the current element in the collection. This methodthrows 'IllegalStateException' if this function is called before next() is

invoked.

List Iterator in Java is an Iterator which allows users to traverse Collection in both direction. It contains the following methods:

1. **void add(Object object):** It inserts object immediately before the element that is returned by the next() function.
2. **boolean hasNext():** It returns true if the list has a next element.
3. **boolean hasPrevious():** It returns true if the list has a previous element.
4. **Object next():** It returns the next element of the list. It throws 'NoSuchElementException' if there is no next element in the list.
5. **Object previous():** It returns the previous element of the list. It throws 'NoSuchElementException' if there is no previous element.
6. **void remove():** It removes the current element from the list. It throws 'IllegalStateException' if this function is called before next() or previous() is invoked.

UNIT-3

SEARCHING LINEAR AND BINARY SEARCH METHODS:

Searching: Searching is the technique of finding desired data items that has been stored within some data structure. Data structures can include linked lists, arrays, search trees, hash tables, or various other storage methods. The appropriate search algorithm often depends on the data structure being searched. Search algorithms can be classified based on their mechanism of searching. They are

1. Linear searching
2. Binary searching

Linear or Sequential searching: Linear Search is the most natural searching method and It is very simple but very poor in performance at times .In this method, the searching begins with searching every element of the list till the required record is found. The elements in the list may be in any order. i.e. sorted or unsorted.

We begin search by comparing the first element of the list with the target element. If it matches, the search ends and position of the element is returned. Otherwise, we will move to next element and compare. In this way, the target element is compared with all the elements until a match occurs. If the match do not occur and there are no more elements to be compared, we conclude that target element is absent in the list by returning position as -1.

For example consider the following list of elements. 55 95 75 85 11 25 65 45

Suppose we want to search for element 11(i.e. Target element = 11). We first compare the target element with first element in list i.e. 55. Since both are not matching we move on the next elements in the list and compare. Finally we will find the match after 5 comparisons at position 4 starting from position 0.

Linear search can be implemented in two ways. i) **Non recursive** ii) **recursive**

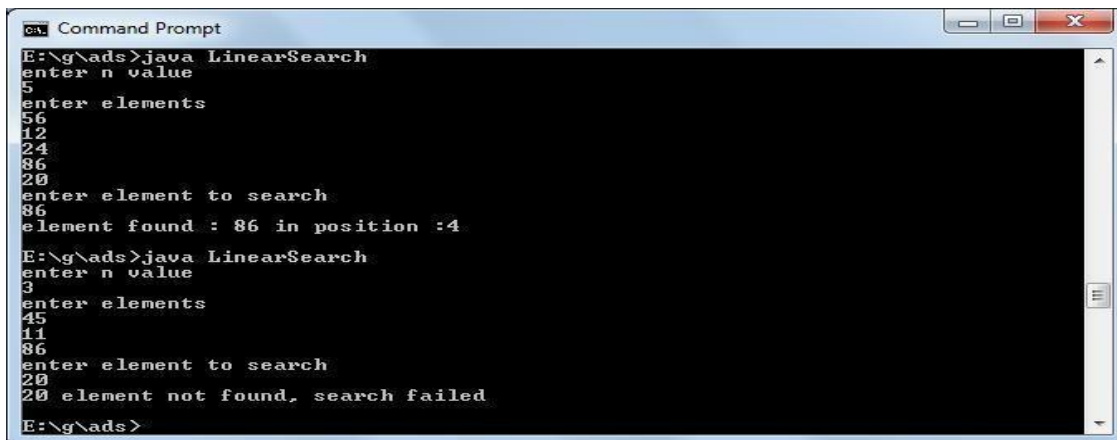
Algorithm for Linear search

```
Linear_Search (A[ ], N, val ,  
pos ) Step 1 : Set pos = -1  
and k = 0 Step 2 : Repeat  
while k < N  
    Begin  
    Step 3 : if A[ k ] =  
    val  
        Set pos =  
        k print pos  
        Goto step  
        5  
    End while
```

Step 4 : print "Value is not
present" Step 5 : Exit

Linear Search using non-recursive function

```
import java.io.*;
class LinearSearch
{
    public static void main(String args[]) throws IOException
    {
        int count=0;
        BufferedReader br=new BufferedReader(new
        InputStreamReader(System.in));
        System.out.println("enter n value");
        int n=Integer.parseInt(br.readLine());
        int arr[]=new int[n];
        System.out.println("enter elements");
        for(int i=0;i<n;i++)
        {
            arr[i]=Integer.parseInt(br.readLine());
        }
        System.out.println("enter element to search");
        int key=Integer.parseInt(br.readLine());
        for(int i=0;i<n;i++)
        {
            if(arr[i]==key)
                System.out.println("element found : " + key + " in position : " +
                (i+1)); else
                    count++;
        }
        if(count==n)
            System.out.println(key + " element not found, search failed");
    }
}
```

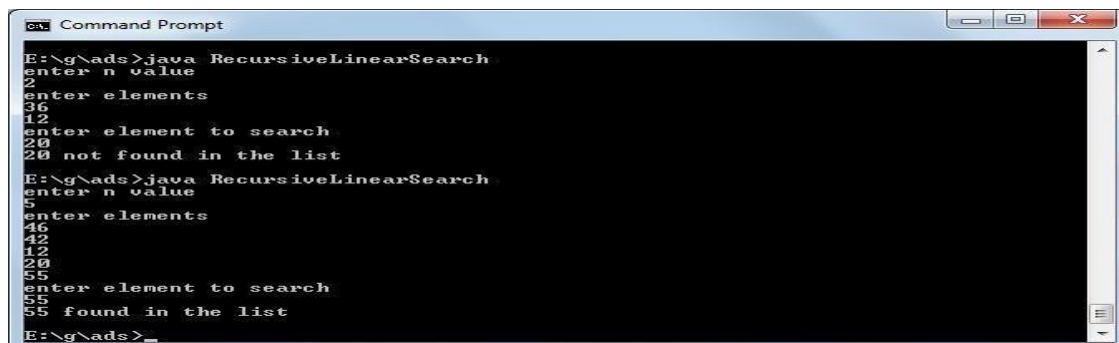
OUTPUT:

```
cmd - Command Prompt
E:\g\ads>java LinearSearch
enter n value
5
enter elements
56
12
24
86
20
enter element to search
86
element found : 86 in position :4
E:\g\ads>java LinearSearch
enter n value
3
enter elements
45
11
86
enter element to search
20
20 element not found, search failed
E:\g\ads>
```

Linear Search using recursive function

```
import java.io.*;
class RecursiveLinearSearch
{
    public static int arr[], key;
    public static void main(String args[]) throws IOException
    {
        BufferedReader br=new BufferedReader(new
        InputStreamReader(System.in));
        System.out.println("enter n value");
        int n=Integer.parseInt(br.readLine());
        arr=new int[n];
        System.out.println("enter elements");
        for(int i=0;i<n;i++)
        {
            arr[i]=Integer.parseInt(br.readLine());
        }
        System.out.println("enter element to search");
        key=Integer.parseInt(br.readLine());

        if( linearSearch(arr.length-1) )
            System.out.println(key + " found in the list" );
        else
            System.out.println(key + " not found in the list");
    }
    static boolean linearSearch(int n)
    {
        if( n < 0 ) return false;
        if(key == arr[n])
            return true;
        else
            return linearSearch(n-1);
    }
}
```

OUTPUT:

```

C:\> Command Prompt
E:\g\ads>java RecursiveLinearSearch
enter n value
2
enter elements
36
12
enter element to search
20
20 not found in the list
E:\g\ads>java RecursiveLinearSearch
enter n value
5
enter elements
46
42
12
20
55
enter element to search
55
55 found in the list
E:\g\ads>
```


BINARY SEARCHING

Binary search is a fast search algorithm with run-time complexity of $O(\log n)$. This search algorithm works on the principle of divide and conquer. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Before applying binary searching, the list of items should be sorted in ascending or descending order. Best case time complexity is **$O(1)$** and Worst case time complexity is



$O(\log n)$

Algorithm:

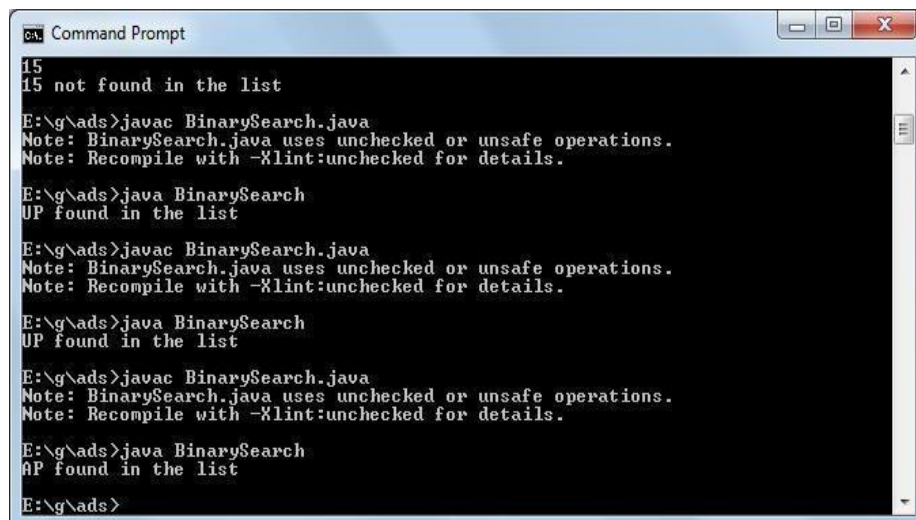
```

Binary_Search (A [ ], U_bound, VAL)
Step 1 : set BEG = 0 , END = U_bound ,
        POS = -1
Step 2 : Repeat while (BEG <=
        END )
Step 3 :   set MID = ( BEG + END ) / 2
Step 4 :   if A [ MID ] == VAL then
            POS = MID
            print VAL " is available at ",
            POS GoTo Step 6
        End if
        if A [ MID ] > VAL
            then set END =
                MID - 1
        Else
            set BEG = MID + 1
        End if
    End
  
```

```
        while
Step 5 : if POS = -1 then
            print VAL " is not present "
        End if
Step 6 : EXIT
```

(b) Binary Search using non-recursive function**SOURCE CODE:**

```
class BinarySearch
{
static Object[] a = { "AP", "KA", "MH", "MP", "OR", "TN", "UP",
    "WB"}; static Object key = "UP";
    public static void main(String args[])
    {
        if( binarySearch() )
            System.out.println(key + " found in the list");
        else
            System.out.println(key + " not found in the list");
    }
    static boolean binarySearch()
    {
        int c, mid, low = 0, high = a.length-1;
        while( low <= high)
        {
            mid = (low + high)/2;
            c = ((Comparable)key).compareTo(a[mid]);
            if( c < 0) high = mid-1;
            else if( c > 0) low = mid+1;
            else return true;
        }
        return false;
    }
}
```

OUTPUT:

```
15
15 not found in the list
E:\g\ads>javac BinarySearch.java
Note: BinarySearch.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

E:\g\ads>java BinarySearch
UP found in the list

E:\g\ads>javac BinarySearch.java
Note: BinarySearch.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

E:\g\ads>java BinarySearch
UP found in the list

E:\g\ads>javac BinarySearch.java
Note: BinarySearch.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

E:\g\ads>java BinarySearch
AP found in the list
E:\g\ads>
```

Binary Search using recursive function

```

import java.io.*;
class RecursiveBinarySearch
{
    public static int arr[], key;
    public static void main(String args[]) throws IOException
    {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("enter n value");
        int n=Integer.parseInt(br.readLine());
        arr=new int[n];
        System.out.println("enter elements");
        for(int i=0;i<n;i++)
        {
            arr[i]=Integer.parseInt(br.readLine());
        }
        System.out.println("enter element to search");
        key=Integer.parseInt(br.readLine());

        if( binarySearch(0, arr.length-1) )
            System.out.println(key + " found in the list");
        else
            System.out.println(key + " not found in the list");
    }
    static boolean binarySearch(int low, int high)
    {
        if( low > high ) return false;
        int mid = (low + high)/2;
        int c = ((Comparable)key).compareTo(arr[mid]);
        if( c < 0 ) return binarySearch(low, mid-1);
        else if( c > 0 ) return binarySearch(mid+1, high);
        else return true;
    }
}

```

OUTPUT:

```

C:\> Command Prompt
enter elements
5n
Exception in thread "main" java.lang.NumberFormatException: For input string: "5n"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:458)
    at java.lang.Integer.parseInt(Integer.java:499)
    at RecursiveBinarySearch.main(RecursiveBinarySearch.java:15)
E:\g\ads> javac RecursiveBinarySearch.java
Note: RecursiveBinarySearch.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
E:\g\ads> java RecursiveBinarySearch
5P not found in the list
E:\g\ads> javac RecursiveBinarySearch.java
Note: RecursiveBinarySearch.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
E:\g\ads> java RecursiveBinarySearch
5P found in the list
E:\g\ads>

```

HASHING AND HASH FUNCTIONS:

Hash table is a data structure used for storing and retrieving data very quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associated with some key.

Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is then dependent upon the size of the hash table

The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

Hash function is a function which is used to put the data in the hash table. Hence one can use the same hash function to retrieve the data from the hash table. Thus hash function is used to implement the hash table.

The integer returned by the hash function is called hash key.

For example: Consider that we want place some employee records in the hash table The record of employee is placed with the help of key: employee ID. The employee ID is a 7 digit number for placing the record in the hash table. To place the record 7 digit number is converted into 3 digits by taking only last three digits of the key.

If the key is 496700 it can be stored at 0th position. The second key 8421002, the record of those key is placed at 2nd position in the array. Hence the hash function will be- $H(\text{key}) = \text{key} \% 1000$ Where $\text{key} \% 1000$ is a hash function and key obtained by hash function is called hash key.

❓ **Bucket and Home bucket:** The hash function $H(\text{key})$ is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket.

The function $H(\text{key})$ is home bucket for the dictionary with pair whose value is key.

TYPES OF HASH FUNCTION

There are various types of hash functions that are used to place the record in the hash table-

1 **Division Method:** The hash function depends upon the remainder of division. Typically the divisor is table length.

For eg; If the record 54, 72, 89, 37 is placed in the hash table and if the table size is 10 then

$h(\text{key}) = \text{record} \% \text{table size}$

$54 \% 10 = 4$

$72 \% 10 = 2$

$89 \% 10 = 9$

$37 \% 10 = 7$

| | |
|---|----|
| 0 | |
| 1 | |
| 2 | 72 |
| 3 | |
| 4 | 54 |
| 5 | |
| 6 | |
| 7 | 37 |
| 8 | |
| 9 | 89 |

2. Mid Square:

In the mid square method, the key is squared and the middle or mid part of the result is used as the index. If the key is a string, it has to be preprocessed to produce a number. Consider that if we want to place a record 3111 then

$$3111^2 = 9678321$$

for the hash table of size 1000

$$H(3111) = 783 \text{ (the middle 3}$$

digits)}

3. Multiplicative hash function:

The given record is multiplied by some constant value. The formula for computing the hash key

is- $H(\text{key}) = \text{floor}(p * (\text{fractional part of key} * A))$ where p is integer constant and A is constant real number. **Donald Knuth suggested to use constant $A = 0.61803398987$**

If key 107 and $p=50$ then

$$H(\text{key}) = \text{floor}(50 * (107 * 0.61803398987))$$

$$= \text{floor}(3306.4818458045)$$

$$= 3306$$

At 3306 location in the hash table the record 107 will be placed.

4. Digit Folding:

The key is divided into separate parts and using some simple operation these parts are combined to produce the hash key.

For eg; consider a record 12365412 then it is divided into separate parts as 123 654 12 and these are added together

$$H(\text{key}) = 123 + 654 + 12$$

$$= 789$$

The record will be placed at location 789

5. Digit Analysis:

The digit analysis is used in a situation when all the identifiers are known in advance. We first transform the identifiers into numbers using some radix, r . Then examine the digits of each identifier. Some digits having most skewed distributions are deleted. This deleting of digits is continued until the number of remaining digits is small enough to give an address in the range of the hash table. Then these digits are used to calculate the hash address.

COLLISION

the hash function is a function that returns the key value using which the record can be placed in the hash table. Thus this function helps us in placing the record in the hash table at appropriate position and due to this we can retrieve the record directly from that location. This function need to be designed very carefully and it should not return the same hash key address for two different records. This is an undesirable situation in hashing.

Definition: The situation in which the hash function returns the same hash key (home bucket) for more than one record is called collision and two same hash keys returned for different records is called synonym.

Similarly when there is no room for a new pair in the hash table then such a situation is called overflow. Sometimes when we handle collision it may lead to overflow conditions. Collision and overflow show the poor hash functions.

For example,

Consider a hash function.

$H(\text{key}) = \text{recordkey} \% 10$ having the hash table size of 10

The record keys to be placed are

131, 44, 43, 78, 19, 36, 57 and 77

$131 \% 10 = 1$

$44 \% 10 = 4$

$43 \% 10 = 3$

$78 \% 10 = 8$

$19 \% 10 = 9$

$36 \% 10 = 6$

$57 \% 10 = 7$

$77 \% 10 = 7$

| | |
|---|-----|
| 0 | |
| 1 | 131 |
| 2 | |
| 3 | 43 |
| 4 | 44 |
| 5 | |
| 6 | 36 |
| 7 | 57 |
| 8 | 78 |
| 9 | 19 |

Now if we try to place 77 in the hash table then we get the hash key to be 7 and at index 7 already the record key 57 is placed. This situation is called **collision**. From the index 7 if we look for next vacant position at subsequent indices 8,9 then we find that there is no room to place 77 in the hash table. This situation is called **overflow**.

COLLISION RESOLUTION TECHNIQUES

If collision occurs then it should be handled by applying some techniques. Such a technique is called collision handling technique.

1. Chaining
2. Open addressing (linear probing)
3. Quadratic probing
4. Double hashing
5. Double hashing
6. Rehashing

CHAINING

In collision handling method chaining is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then a linked list(chain) is maintained at the home bucket.

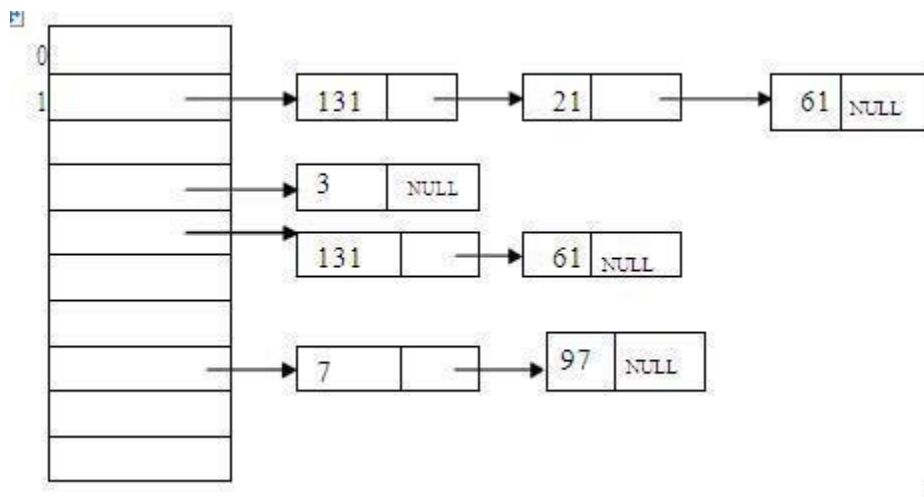
For eg;

Consider the keys to be placed in their home buckets are 131, 3, 4, 21, 61, 7, 97, 8, 9

then we will apply a hash function as $H(\text{key}) =$

$\text{key} \% D$ Where D is the size of table. The hash

table will be- Here $D = 10$



A chain is maintained for colliding elements. for instance 131 has a home bucket (key) 1. similarly key 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1.

OPEN ADDRESSING – LINEAR PROBING

This is the easiest method of handling collision. When collision occurs i.e. when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing (open addressing), the hash table is represented as a one-dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we inset elements into the table. Then using some suitable hash function the element can be inserted into the hash table.

For example:

Consider that following keys are to be inserted in the hash

table 131, 4, 8, 7, 21, 5, 31, 61, 9, 29

Initially, we will put the following keys in the hash table.

We will use Division hash function. That means the keys are placed using the

formula $H(\text{key}) = \text{key} \% \text{tablesize}$

$H(\text{key}) = \text{key} \% 10$

For instance the element 131 can be

placed at $H(\text{key}) = 131 \% 10$

$= 1$

Index 1 will be the home bucket for 131. Continuing in this fashion we will place 4,

8, 7. Now the next key to be inserted is 21. According to the hash function

$H(\text{key}) = 21 \% 10$

$H(\text{key}) = 1$

But the index 1 location is already occupied by 131 i.e. collision occurs. To resolve this collision we will linearly move down and at the next empty location we will prob the element. Therefore 21 will be placed at the index 2. If the next element is 5 then we get the home bucket for 5 as index 5 and this bucket is empty so we will put the element 5 at index 5.

| Index | Key | Key | Key |
|-------|------|------|------|
| 0 | NULL | NULL | NULL |
| 1 | 131 | 131 | 131 |
| 2 | NULL | 21 | 21 |
| 3 | NULL | NULL | 31 |
| 4 | 4 | 4 | 4 |
| 5 | NULL | 5 | 5 |
| 6 | NULL | NULL | 61 |
| 7 | 7 | 7 | 7 |
| 8 | 8 | 8 | 8 |
| 9 | NULL | NULL | NULL |

after placing keys 31, 61

The next record key is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0th index.

Problem with linear probing:

One major problem with linear probing is primary clustering. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved.

$$19\%10 = 9$$

$$18\%10 = 8$$

$$39\%10 = 9$$

$$29\%10 = 9$$

$$8\%10 = 8$$

cluster is formed

rest of the table is empty

this cluster problem can be solved by quadratic probing.

| Key |
|-----|
| 39 |
| 29 |
| 8 |
| |
| |
| |
| |
| 18 |
| 19 |

QUADRATIC PROBING:

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.



$$H(\text{key}) = (\text{Hash}(\text{key}) + i^2) \% m$$

where m can be table size or any prime number.

for eg: If we have to insert following elements in the hash table with table size 10:

37, 90, 55, 22, 17, 49, 87

$$37 \% 10 = 7$$

$$90 \% 10 = 0$$

$$55 \% 10 = 5$$

$$22 \% 10 = 2$$

$$11 \% 10 = 1$$

Now if we want to place 17 a collision will occur as $17 \% 10 = 7$ and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

$$H_i(\text{key}) = (\text{Hash}(\text{key}) + i^2) \% m$$

Consider $i = 0$ then

$$(17 + 0^2) \% 10 = 7$$

□

$$(17 + 1^2) \% 10 = 8, \text{ when } i = 1$$

The bucket 8 is empty hence we will place the element at index 8. Then comes 49 which will be placed at index 9.

$$49 \% 10 = 9$$

Now to place 87 we will use quadratic probing.

$$(87 + 0) \% 10 = 7$$

$$(87 + 1) \% 10 = 8 \dots \text{but already occupied}$$

$$(87 + 2^2) \% 10 = 1 \dots \text{already occupied}$$

$$(87 + 3^2) \% 10 = 6$$

It is observed that if we want place all the necessary elements in the hash table the size of divisor (m) should be twice as large as total number of elements.

DOUBLE HASHING

| | |
|---|----|
| 0 | 90 |
| 1 | 11 |
| 2 | 22 |
| 3 | |
| 4 | |
| 5 | 55 |
| 6 | |
| 7 | 37 |
| 8 | |
| 9 | |

| | |
|---|----|
| 0 | 90 |
| 1 | 11 |
| 2 | 22 |
| 3 | |
| 4 | |
| 5 | 55 |
| 6 | |
| 7 | 37 |
| 8 | 49 |
| 9 | |

| | |
|---|----|
| 0 | 90 |
| 1 | 11 |
| 2 | 22 |
| 3 | |
| 4 | |
| 5 | 55 |
| 6 | 87 |
| 7 | 37 |
| 8 | 49 |
| 9 | |

DOUBLE HASHING

Double hashing is technique in which a second hash function is applied to the key when a collision occurs. By applying the second hash function we will get the number of positions from the point of collision to insert.

There are two important rules to be followed for the second function:

1. it must never evaluate to zero.
2. must make sure that all cells can be probed. The formula to be used for double hashing is

$$H_1(\text{key}) = \text{key} \bmod \text{table size}$$

$$H_2(\text{key}) = M - (\text{key} \bmod M)$$

where M is a prime number smaller than the size of the table.

Consider the following elements to be placed in the hash table of size 10: 37, 90, 45, 22, 17, 49, 55

Initially insert the elements using the formula for $H_1(\text{key})$.

Insert 37, 90, 45, 22

$$H_1(37) = 37 \% 10 = 7$$

$$H_1(90) = 90 \% 10 = 0$$

$$H_1(45) = 45 \% 10 = 5$$

$$H_1(22) = 22 \% 10 = 2$$

$$H_1(49) = 49 \% 10 = 9$$

| Key |
|-----|
| 90 |
| |
| 22 |
| |
| |
| 45 |
| |
| 37 |
| |
| 49 |

Now if 17 to be inserted then

$$H_1(17) = 17 \% 10 = 7$$

$$H_2(\text{key}) = M - (\text{key} \% M)$$

Here M is prime number smaller than the size of the table. Prime number smaller than table size 10 is 7

Hence $M = 7$

$$\begin{aligned} H_2(17) &= 7 - (17 \% 7) \\ &= 7 - 3 = 4 \end{aligned}$$

That means we have to insert the element 17 at 4 places from 37. In short we have jumps. Therefore the 17 will be placed at index 1.

| Key |
|-----|
| 90 |
| 17 |
| 22 |
| |
| |
| 45 |
| |
| 37 |
| |
| 49 |

Now to insert number 55

$$H_1(55) = 55 \% 10 = 5 \rightarrow \text{Collision}$$

$$H_2(55) = 7 - (55 \% 7) \\ = 7 - 6 = 1$$

That means we have to take one jump from index 5 to place 55.
Finally the hash table will be -

| Key |
|-----|
| 90 |
| 17 |
| 22 |
| |
| |
| 45 |
| 55 |
| 37 |
| |
| 49 |

Comparison of Quadratic Probing & Double Hashing

The double hashing requires another hash function whose probing efficiency is same as some another hash function required when handling random collision.

The double hashing is more complex to implement than quadratic probing. The quadratic probing is fast technique than double hashing.

REHASHING

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable if the total size of table is a prime number. There are situations in which the rehashing is required.

1. When table is completely full
2. With quadratic probing when the table is filled half.
3. When insertions fail due to overflow.
4. In such situations, we have to transfer entries from old table to the new table by re-computing their positions using hash functions.
5. Consider we have to insert the elements 37, 90, 55, 22, 17, 49, and 87. the table size is 10 and will use hash function.,

6. $H(\text{key}) = \text{key} \bmod \text{table size}$

7. $37 \% 10 = 7$
8. $90 \% 10 = 0$
9. $55 \% 10 = 5$
10. $22 \% 10 = 2$
11. $17 \% 10 = 7$ Collision solved by linear probing
12. $49 \% 10 = 9$

Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be

$H(\text{key}) = \text{key} \bmod 23$

$$37 \% 23 = 14$$

$$90 \% 23 = 21$$

$$55 \% 23 = 9$$

$$22 \% 23 = 22$$

$$17 \% 23 = 17$$

$$49 \% 23 = 3$$

$$87 \% 23 = 18$$

| | |
|----|----|
| 0 | 90 |
| 1 | 11 |
| 2 | 22 |
| 3 | |
| 4 | |
| 5 | 55 |
| 6 | 87 |
| 7 | 37 |
| 8 | 49 |
| 9 | |
| 10 | |
| 11 | |
| 12 | |
| 13 | |
| 14 | |
| 15 | |
| 16 | |
| 17 | |
| 18 | |
| 19 | |
| 20 | |
| 21 | |
| 22 | |
| 23 | |

Now the hash table is sufficiently large to accommodate new insertions.

Advantages:

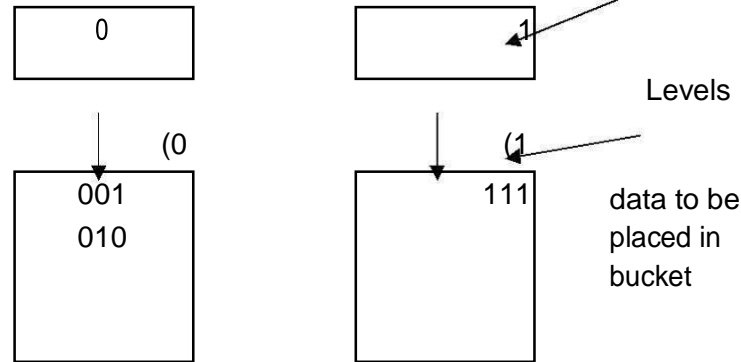
1. This technique provides the programmer a flexibility to enlarge the table size if required.
2. Only the space gets doubled with simple hash function which avoids occurrence of collisions.

EXTENSIBLE HASHING

Extensible hashing is a technique which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bits. Extensible hashing grow and shrink similar to B-trees.

In extensible hashing referring the size of directory the elements are to be placed in buckets. The levels are indicated in parenthesis.

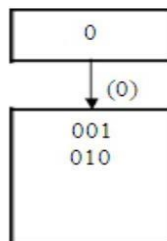
For eg:



The bucket can hold the data of its global depth. If data in bucket is more than global depth then, split the bucket and double the directory.

Consider we have to insert 1, 4, 5, 7, 8, 10. Assume each page can hold 2 data entries (2 is the depth).

Step 1: Insert 1, 4

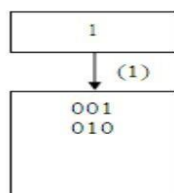
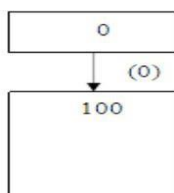


1 = 001

4 = 100

We will examine last bit of data and insert the data in bucket.

Insert 5. The bucket is full. Hence double the directory.



1 = 001

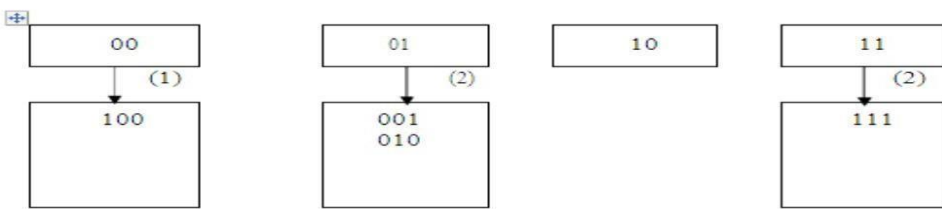
4 = 100

5 = 101

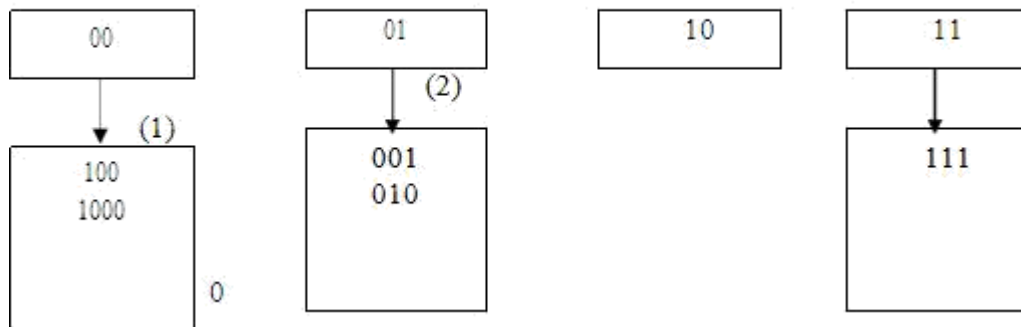
Based on last bit the data is inserted.

Step 2: Insert 7
7 = 111

But as depth is full we can not insert 7 here. Then double the directory and split the bucket. After insertion of 7. Now consider last two bits.



Step 3: Insert 8 i.e. 1000

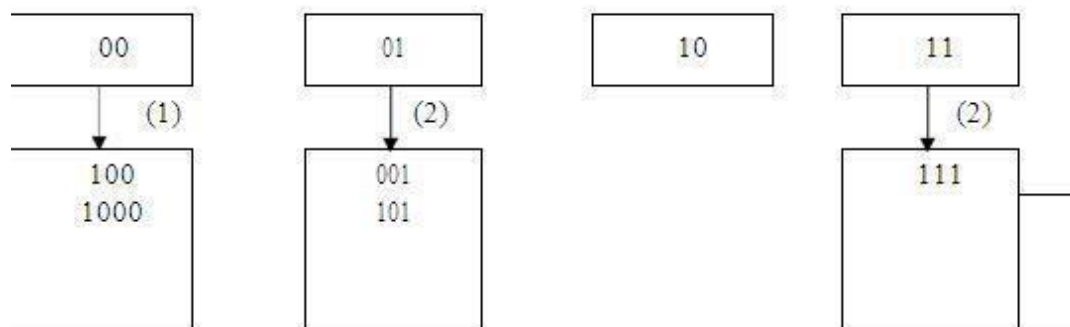


Thus the data is inserted using extensible hashing.

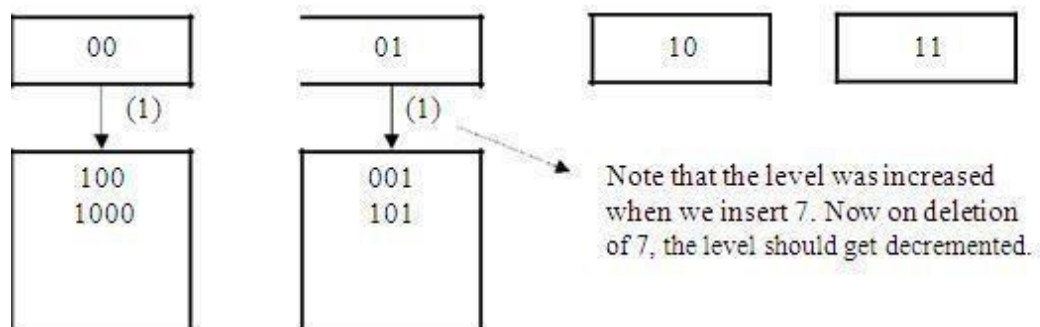
Deletion Operation:

If we want to delete 10 then, simply make the bucket of 10 empty.

If we want to delete 10 then, simply make the bucket of 10 empty.



Delete 7.



Applications of hashing:

1. In compilers to keep track of declared variables.
2. For online spelling checking the hashing functions are used.
3. Hashing helps in Game playing programs to store the moves made.
4. For browser program while caching the web pages, hashing is used.
5. Construct a *message authentication code* (MAC)
6. Digital signature.
7. Time stamping
8. Key updating: key is hashed at specific intervals resulting in new key

SORTING TECHNIQUES:

Sorting in general refers to various methods of arranging or ordering things based on criteria (numerical, chronological, alphabetical, hierarchical etc.). There are many approaches to sorting data and each has its own merits and demerits.

Bubble Sort:

Bubble Sort is probably one of the oldest, easiest, straight-forward, inefficient sorting algorithms. Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists. Bubble sort is not a stable sort which means that if two same elements are there in the list, they may not get their same order with respect to each other.

Bubble Sort Algorithm:

Step 1: Repeat Steps 2 and 3 for i=1 to 10

Step 2: Set j=1

Step 3: Repeat while j<=n

(A) if a[i] < a[j]

Then interchange a[i] and a[j] [End of if]

(B) Set j = j+1

[End of Inner Loop]

[End of Step 1 Outer Loop]

Step 4: Exit

Step-by-step example:

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass:

(**5**1428) (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(1**5**428) (1 **4** **5** 2 8), Swap since $5 > 4$

(14**5**28) (1 4 **2** **5** 8), Swap since $5 > 2$

(142**5**8)

(142**5**8),

Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

Second Pass:

(14258) (14258)

(14**2**58) (1 **2** **4** 5 8), Swap since $4 > 2$

(124**5**8) (12458)

(124**5**8) (12458)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(12458) (12458)

(12458) (12458)

(12458) (12458)

(12458) (12458)

Time**Complexity:**

Worst Case Performance **$O(N^2)$**

Best Case Performance **$O(N^2)$**

Average Case Performance **$O(N^2)$**

SOURCE CODE:**Bubble Sort**

```
import
java.io.*; class
BubbleSort
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader br=new BufferedReader(new
                                InputStreamReader(System.i
n)); System.out.println("enter n value");
        int
n=Integer.parseInt(br.readLine());
        int arr[]=new int[n];
        System.out.println("enter
elements"); for(int i=0;i<n;i++)
        {
            arr[i]=Integer.parseInt(br.readLine());
        }
        System.out.print("\n Unsorted array:
"); display( arr );
        bubbleSort( arr );
        System.out.print("\n Sorted array:
"); display( arr );
    }
    static void bubbleSort(int[] a)
    {
        int i, pass, exch, n =
a.length; int tmp;
        for( pass = 0; pass < n; pass++ )
        {
            exch = 0;
            for( i = 0; i < n-pass-1; i++ )
                if( ((Comparable)a[i]).compareTo(a[i+1]) > 0)
                {
                    tmp = a[i];
                    a[i] = a[i+1];
                    a[i+1] =
tmp;
                    exch++;
                }
            if( exch == 0 ) return;
        }
    }
    static void display( int a[] )
    {
```

```

        for( int i = 0; i < a.length;
            i++ ) System.out.print( a[i]
                + " " );
    }
}

```

OUTPUT:

```

C:\> Command Prompt
E:\g\ads>
E:\g\ads> javac BubbleSort.java
Note: BubbleSort.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
E:\g\ads> java BubbleSort
enter n value
10
enter elements
89
52
12
48
75
99
85
35
65
49
Unsorted array: 89 52 12 48 75 99 85 35 65 49
Sorted array: 12 35 48 49 52 65 75 85 89 99
E:\g\ads>
E:\g\ads>

```

Insertion Sort:

An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an **incremental** algorithm. It builds the sorted sequence one number at a time. This is a suitable sorting technique in playing card games. Insertion sort provides several advantages:

1. Simple implementation
2. Efficient for (quite) small data sets
3. Adaptive (i.e., efficient) for data sets that are already substantially sorted: the time complexity is $O(n + d)$, where d is the number of inversions
4. More efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is $O(n)$
5. Stable; i.e., does not change the relative order of elements with equal keys
6. In-place; i.e., only requires a constant amount $O(1)$ of additional memory space
7. Online; i.e., can sort a list as it receives it

Step-by-step example

| | | |
|--------|--|--|
| Step 1 | | Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12. |
| Step 2 | | Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3. |
| Step 3 | | Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12. |
| Step 4 | | Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12. |
| | | Sorted Array in Ascending Order |

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Suppose, you want to sort elements in ascending as in above figure. Then,

1. Step 1: The second element of an array is compared with the elements that appear before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.

2. Step 2: The third element of an array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it is kept in the position as it is. After second step, first three elements of an array will be sorted.

3. Step 3: Similarly, the fourth element of an array is compared with the elements that appear before it (first, second and third element) and the same procedure is applied and that element is inserted in the proper position. After third step, first four elements of an array will be sorted.

If there are n elements to be sorted. Then, this procedure is repeated $n-1$ times to get sorted list of array.

Time Complexity:

| | |
|---------------------------------|----------|
| Worst Case Performance | $O(N^2)$ |
|) Best Case Performance(nearly) | $O(N)$ |
| Average Case Performance | $O(N^2)$ |
|) | |

Source Code:

```
//Insertion Sort import
java.io.*; class
InsertionSort
{
public static void main(String[] args) throws IOException
{
BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
System.out.println("enter n value"); int
n=Integer.parseInt(br.readLine()); int arr[]=new
int[n]; System.out.println("enter elements"); for(int
i=0;i<n;i++)
{
arr[i]=Integer.parseInt(br.readLine());
}
System.out.print("\n Unsorted array: "); display( arr );
insertionSort( arr ); System.out.print("\n Sorted
array: "); display( arr );
}
static void insertionSort(int a[])
{
int i, j, n = a.length; int item;
for( j = 1; j < n; j++ )
```

```
{  
item = a[j]; i = j-1;  
while( i >= 0 && ((Comparable)item).compareTo(a[i]) < 0){
```

```

a[i+1] = a[i]; i = i-1;
}
a[i+1] = item;
}}
static void display( int a[] )
{
for( int i = 0; i < a.length; i++ ) System.out.print( a[i]
+ " " );
}}

```

OUTPUT:

```

C:\> Command Prompt

Unsorted array: 89 52 12 48 75 99 85 35 65 49
Sorted array: 12 35 48 49 52 65 75 85 89 99
E:\g\ads>
E:\g\ads> javac InsertionSort.java
Note: InsertionSort.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
E:\g\ads> java InsertionSort
enter n value
5
enter elements
99
45
85
23
12

Unsorted array: 99 45 85 23 12
Sorted array: 12 23 45 85 99
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>

```

Quick sort: It is a divide and conquer algorithm. Developed by Tony Hoare in 1959. Quick sort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick sort can then recursively sort the sub-arrays.

ALGORITHM

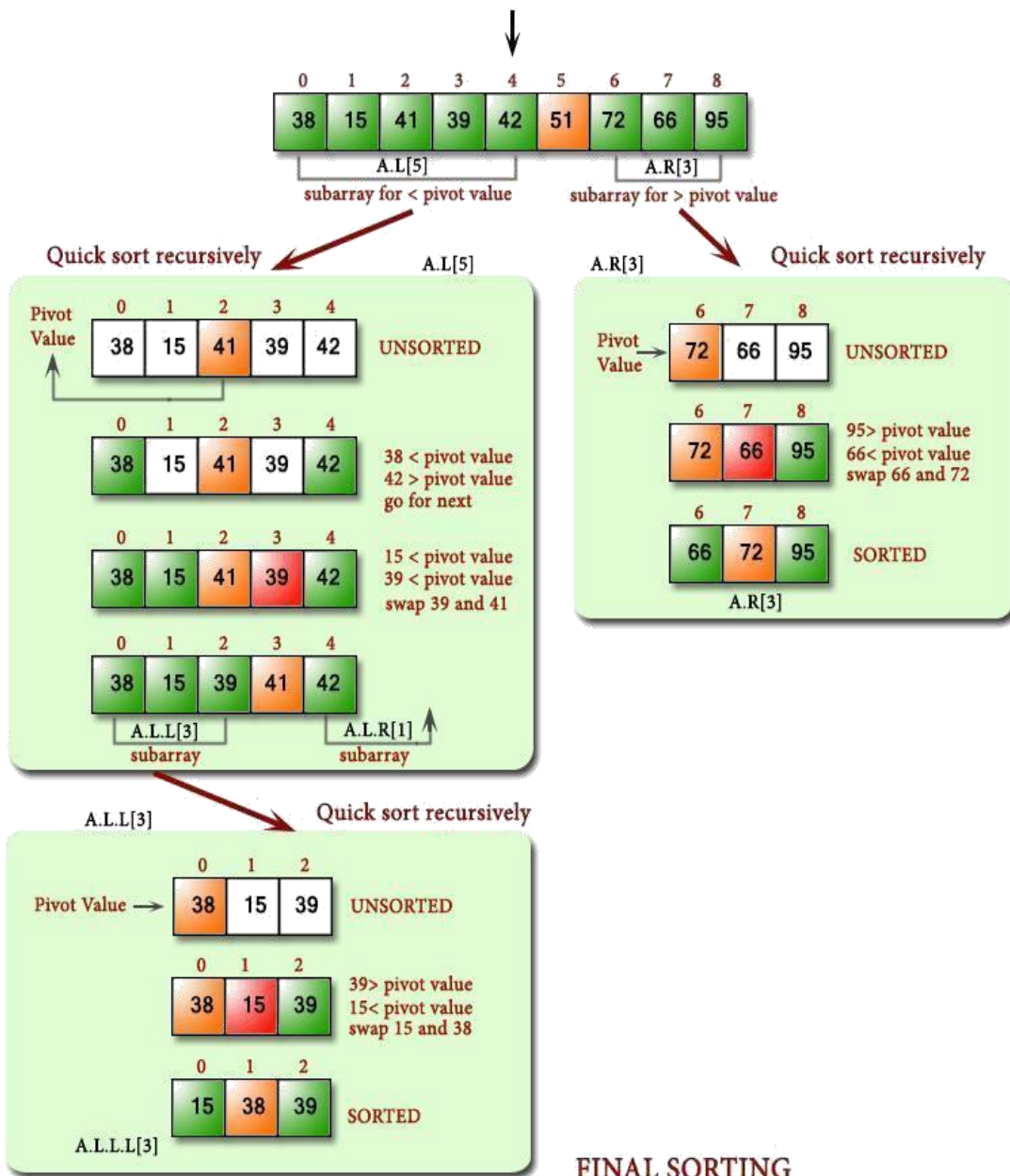
M:

Step 1: Pick an element, called a pivot, from the array.

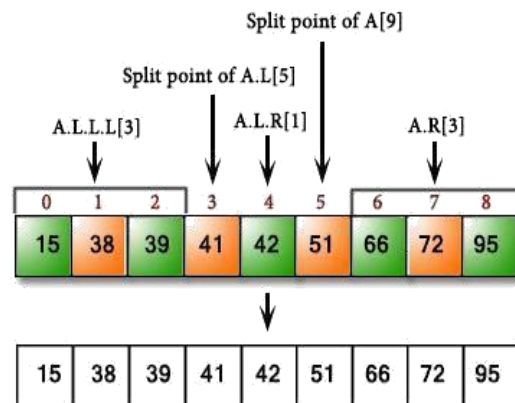
Step 2: Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

Step 3: Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.





FINAL SORTING



Advantages:

1. One of the fastest algorithms on average.
2. Does not need additional memory (the sorting takes place in the array - this is called **in-place** processing).

WORST CASE $O(N^2)$ BEST CASE $O(N \log_2 N)$ AVERAGE CASE $O(N \log_2 N)$

SOURCE CODE:

```
//Quick Sort
```

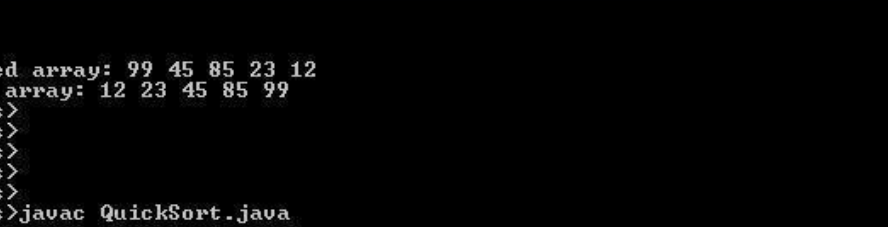
```
import java.io.*;
class QuickSort
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader br=new BufferedReader(new
        InputStreamReader(System.in));
        System.out.println("enter n value"); int
        n=Integer.parseInt(br.readLine()); int arr[]=new
        int[n]; System.out.println("enter elements");
        for(int i=0;i<n;i++)
        {
            arr[i]=Integer.parseInt(br.readLine());
        }

        System.out.print("\n Unsorted array: "); display( arr
        );

        quickSort( arr, 0, arr.length-1 );
        System.out.print("\n Sorted array: "); display(
        arr );
    }
    static void quickSort(int a[], int left, int right)
    {
        int newleft = left, newright = right; int amid,
        tmp;
        amid = a[(left + right)/2]; do
        {
            while( (a[newleft] < amid) && (newleft < right)) newleft++;
            while( (amid < a[newright]) && (newright > left)) newright--;
            if(newleft <= newright)
            {
                tmp = a[newleft]; a[newleft] = a[newright];
                a[newright] = tmp; newleft++; newright--;
            }
        } while(newleft <= newright); if(left < newright)
        quickSort(a, left, newright);
        if(newleft < right)
        quickSort(a, newleft, right);
    }
}
```

```
static void display( int a[] )
{
for( int i = 0; i < a.length; i++ ) System.out.print(
a[i] + " " );
}
}
```

OUTPUT:



```
23
12

Unsorted array: 99 45 85 23 12
Sorted array: 12 23 45 85 99
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>javac QuickSort.java
E:\g\ads>java QuickSort
enter n value
5
enter elements
63
48
52
99
12

Unsorted array: 63 48 52 99 12
Sorted array: 12 48 52 63 99
E:\g\ads>
```

MERGE SORT:

Merge Sort:

Merge sort is based on Divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merged to get a sorted list. The two unsorted lists are sorted by continually calling the merge-sort algorithm; we eventually get a list of size 1 which is already sorted. The two lists of size 1 are then merged.

Merge Sort Procedure: This is a divide and conquer algorithm. This works as follows :

1. Divide the input which we have to sort into two parts in the middle. Call it the left part and right part. Example: Say the input is -10 32 45 -78 91 1 0 -16 then the left part will be -10 32 45 -78 and the right part will be 91 1 0 6.
2. Sort each of them separately. Note that here sort does not mean to sort it using some other method. We use the same function recursively.
3. Then merge the two sorted parts.

Input the total number of elements that are there in an array (number_of_elements). Input the array (array[number_of_elements]). Then call the function MergeSort() to sort the input array. MergeSort() function sorts the array in the range [left,right] i.e. from index left to index right inclusive. Merge()

function merges the two sorted parts. Sorted parts will be from [left, mid] and [mid+1, right]. After merging output the sorted array.

MergeSort() function:

It takes the array, left-most and right-most index of the array to be sorted as arguments. Middle index (mid) of the array is calculated as $(\text{left} + \text{right})/2$. Check if $(\text{left} < \text{right})$ cause we have to sort only when $\text{left} < \text{right}$ because when $\text{left} = \text{right}$ it is anyhow sorted. Sort the left part by calling MergeSort() function again over the left part MergeSort(array, left, mid) and the right part by recursive call of MergeSort function as MergeSort(array, mid + 1, right). Lastly merge the two arrays using the Merge function.

Merge() function:

It takes the array, left-most , middle and right-most index of the array to be merged as arguments. Finally copy back the sorted array to the original array.

WORST CASE $O(N \log_2 N)$ BEST CASE $O(N \log_2 N)$ AVERAGE CASE $O(N \log_2 N)$ **Source Code: Merge Sort**

```
import java.io.*;
class MergeSort
{
    int[] a;
    int[] tmp;
    MergeSort(int[] arr)
    {
        a = arr;
        tmp = new int[a.length];
    }
    public static void main(String[] args) throws IOException
    {
        BufferedReader br=new BufferedReader(new
            InputStreamReader(System.in));
        System.out.println("enter n value");
        int n=Integer.parseInt(br.readLine());
        int arr[]=new int[n];
        System.out.println("enter elements");
        for(int i=0;i<n;i++)
        {
            arr[i]=Integer.parseInt(br.readLine());
        }

        System.out.print("\n Unsorted array: ");
        display( arr );
        MergeSort ms=new MergeSort(arr);
        ms.msort();
        System.out.print("\n Sorted array: ");
        display( arr );
    }
}
```

```

void msort()
{
    sort(0, a.length-1);
}
void sort(int left, int right)
{
    if(left < right)
    {
        int mid = (left+right)/2;
        sort(left, mid);
        sort(mid+1, right);
        merge(left, mid, right);
    }
}
void merge(int left, int mid, int right)
{
    int i = left;
    int j = left;
    int k = mid+1;
    while( j <= mid && k <= right )
    {
        if(a[j] < a[k])
            tmp[i++] = a[j++];
        else
            tmp[i++] = a[k++];
    }
    while( j <= mid )
        tmp[i++] = a[j++];
    for(i=left; i < k; i++)
        a[i] = tmp[i];
}
static void display( int a[] )
{
    for( int i = 0; i < a.length; i++ )
        System.out.print( a[i] + " " );
}
}

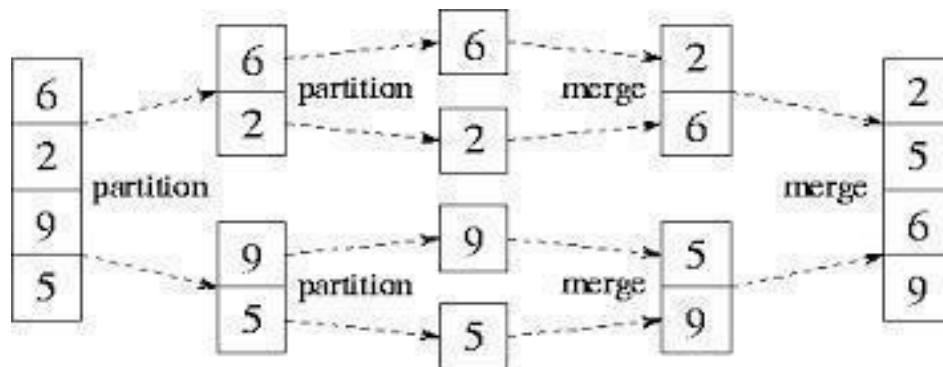
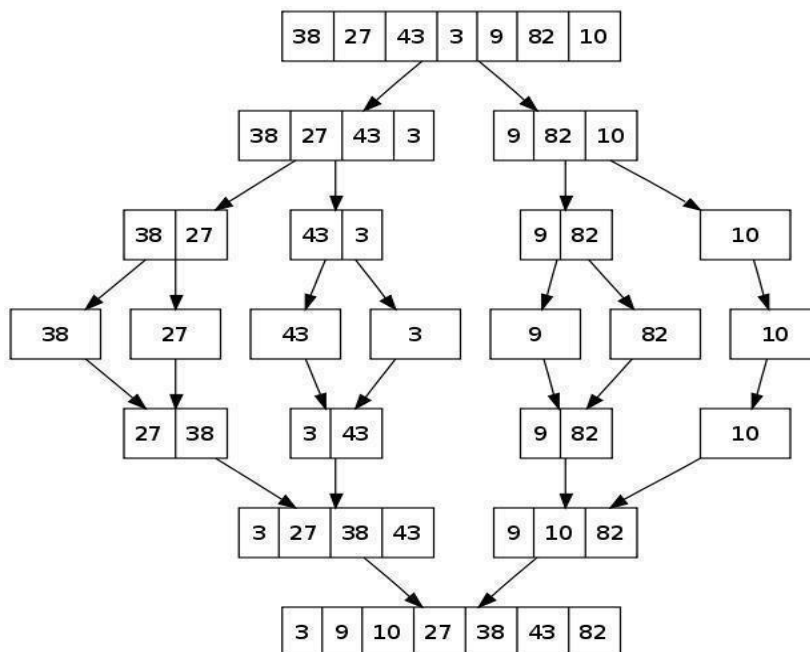
```

OUTPUT:

```

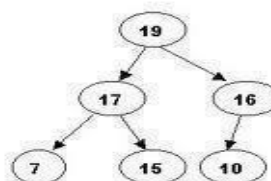
C:\> Command Prompt
63
48
52
99
12
Unsorted array: 63 48 52 99 12
Sorted array: 12 48 52 63 99
E:\g\ads>javac MergeSort.java
E:\g\ads>java MergeSort
enter n value
5
enter elements
63
58
15
24
34
Unsorted array: 63 58 15 24 34
Sorted array: 15 24 34 58 63
E:\g\ads>
E:\g\ads>
E:\g\ads>

```

STEP BY STEP PROCEDURE:**EX: TAKEN SET 6,2,9,5 TO SORTED LIST****2,5,6,9 EXAMPLE 1****EXAMPLE 2: TAKING SET 38,27,43,3,9,82,10 TO SORTED LIST AS 3,9,10,27,38,43,82****HEAP SORT:**

The heap sort algorithm can be divided into two parts. In the first step, a heap is built out of the data. In the second step, a sorted array is created by repeatedly removing the largest element from the heap, and inserting it into the array. The heap is reconstructed after each removal. Once all objects have been removed from the heap, we have a sorted array. The direction of the sorted elements can be varied by choosing a min-heap or max-heap in step one. Heap sort can be performed in place. The array can be split into two parts, the sorted array and the heap.

The **(Binary) heap** data structure is an **array** object that can be viewed as a nearly complete binary tree.



Heap Sort Algorithm:

Step 1. Build Heap – $O(n)$ -Build binary tree taking N items as input, ensuring the heap structure property is held, in other words, build a complete binary tree. Heapify the binary tree making sure the binary tree satisfies the Heap Order property.

Step 2. Perform n deleteMax operations – $O(\log(n))$ - Delete the maximum element in the heap – which

SOURCE CODE:

```
import java.io.*;

class HeapSort
{
    int[] a;
    int maxSize; int
    currentSize;
    public HeapSort(int m)
    {
        maxSize = m; currentSize =
        0;
        a = new int[maxSize];
    }
    public static void main(String[] args) throws IOException
    {

        BufferedReader br=new BufferedReader(new InputStreamReader(System.in)); System.out.println("enter n
        value");
        int n=Integer.parseInt(br.readLine()); int arr[]=new
        int[n]; System.out.println("enter elements"); for(int
        i=0;i<n;i++)
        {
            arr[i]=Integer.parseInt(br.readLine());
        }

        System.out.print("\n Unsorted array: "); display( arr );

        HeapSort hs=new HeapSort(n); hs.heapsort(arr);

        System.out.print("\n Sorted array: "); display( arr );
    }

    public boolean insert(int key)
    {
        if(currentSize == maxSize) return false;
        a[currentSize] = key;
        moveUp(currentSize++); return true;
    }
    public void moveUp(int index) {
        int parent = (index-1)/2; int bottom =
        a[index];
        while(index > 0 && a[parent] < bottom)
        {
```

```
a[index] = a[parent]; index = parent;
parent = (parent-1)/2;
}
a[index] = bottom;
}
public int remove()
{
if( isEmpty() )
{
System.out.println("Heap is empty"); return -1;
}
int root = a[0];
a[0] = a[--currentSize];
moveDown(0);
return root;
}
public void moveDown(int index)
{
int largerChild; int top =
a[index];
while(index < currentSize/2)
{
int leftChild = 2*index+1; int rightChild =
2*index+2;
if(rightChild<currentSize && a[leftChild]<a[rightChild] ) largerChild = rightChild;
else
largerChild = leftChild;
if(top >= a[largerChild]) break; a[index] =
a[largerChild]; index = largerChild;
}
a[index] = top;
}
public boolean isEmpty()
{
return currentSize==0;
}
void heapsort(int []arr)
{
HeapSort h = new HeapSort(arr.length); for(int i = 0; i
< arr.length; i++)
h.insert(arr[i]);
for( int i = arr.length-1; i >= 0; i-- ) arr[i] =
h.remove();
}
static void display( int a[] )
{
for( int i = 0; i < a.length; i++ ) System.out.print( a[i]
+ " ");
}
}
```

OUTPUT:

```

Command Prompt
15
24
34

Unsorted array: 63 58 15 24 34
Sorted array: 15 24 34 58 63
E:\g\ads>
E:\g\ads>
E:\g\ads>javac HeapSort.java
E:\g\ads>java HeapSort
enter n value
5
enter elements
65
48
96
75
23

Unsorted array: 65 48 96 75 23
Sorted array: 23 48 65 75 96
E:\g\ads>
E:\g\ads>
E:\g\ads>

```

Given an array of 6 elements: 15, 19, 10, 7, 17, 16, sort it in ascending order using heap sort. Steps:

1. Consider the values of the elements as priorities and build the heaptree.
2. Start deleteMin operations, storing each deleted element at the end of the heap array. After performing step 2, the order of the elements will be opposite to the order in the heap tree. Hence, if we want the elements to be sorted in ascending order, we need to build the heap tree in descending order - the greatest element will have the highest priority.

Note that we use only one array , treating its parts differently:

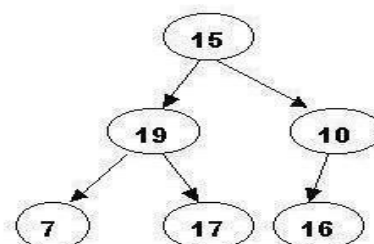
- a. when building the heap tree, part of the array will be considered as the heap, and the rest part -the original array.
- b. when sorting, part of the array will be the heap, and the rest part - the sortedarray.

This will be indicated by colors: white for the original array, blue for the heap and red for the sorted array

Here is the array: 15, 19, 10, 7, 17, 6

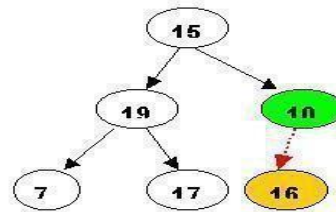
A. Building the heap tree

The array represented as a tree, complete but not ordered:



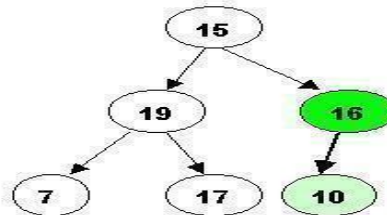
Start with the rightmost node at height 1 - the node at position 3 = $\text{Size}/2$. It has one greater child and has to be percolated down:

| | | | | | |
|----|----|----|---|----|----|
| 15 | 19 | 10 | 7 | 17 | 16 |
|----|----|----|---|----|----|



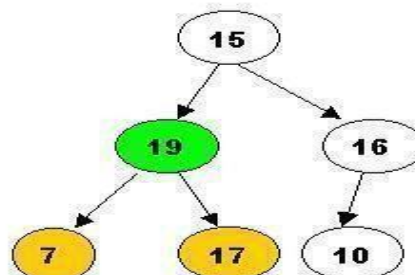
After processing array[3] the situation is:

| | | | | | |
|----|----|----|---|----|----|
| 15 | 19 | 16 | 7 | 17 | 10 |
|----|----|----|---|----|----|



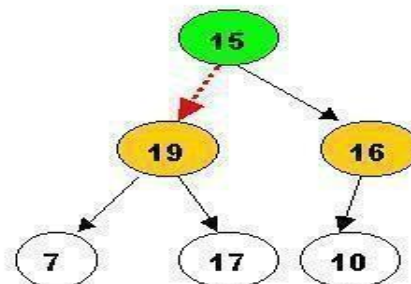
Next comes array[2]. Its children are smaller, so no percolation is needed.

| | | | | | |
|----|----|----|---|----|----|
| 15 | 19 | 16 | 7 | 17 | 10 |
|----|----|----|---|----|----|



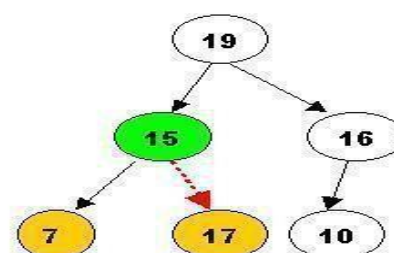
The last node to be processed is array[1]. Its left child is the greater of the children. The item at array[1] has to be percolated down to the left, swapped with array[2].

| | | | | | |
|----|----|----|---|----|----|
| 15 | 19 | 16 | 7 | 17 | 10 |
|----|----|----|---|----|----|



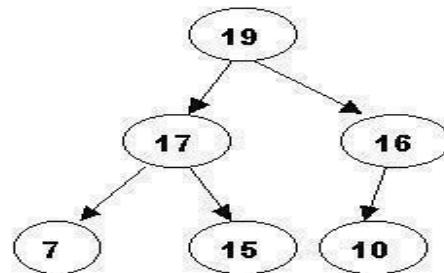
As a result the situation is:

| | | | | | |
|----|----|----|---|----|----|
| 19 | 15 | 16 | 7 | 17 | 10 |
|----|----|----|---|----|----|



The children of array[2] are greater, and item 15 has to be moved down further, swapped with array[5].

| | | | | | |
|----|----|----|---|----|----|
| 19 | 17 | 16 | 7 | 15 | 10 |
|----|----|----|---|----|----|



Now the tree is ordered, and the binary heap is built.

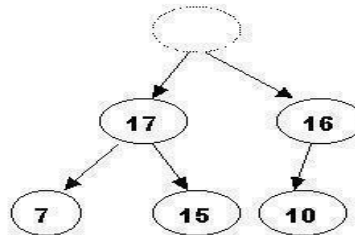
B. Sorting - performing deleteMax operations:

1. Delete the top element 19.

Store 19 in a temporary place. A hole is created at the top

| | | | | | |
|--|----|----|---|----|----|
| | 17 | 16 | 7 | 15 | 10 |
|--|----|----|---|----|----|

19



Swap 19 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array

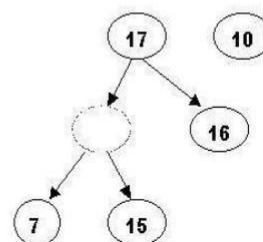
| | | | | | |
|--|----|----|---|----|----|
| | 17 | 16 | 7 | 15 | 19 |
|--|----|----|---|----|----|

10

Percolate down the

| | | | | | |
|----|--|----|---|----|----|
| 17 | | 16 | 7 | 15 | 19 |
|----|--|----|---|----|----|

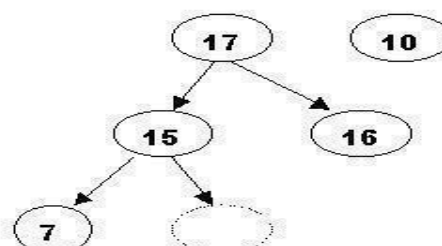
10



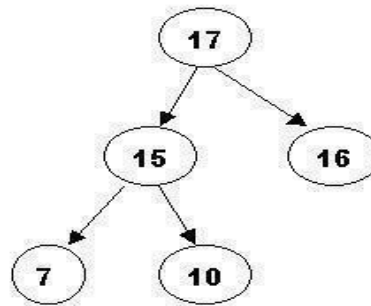
Percolate once more (10 is less than 15, so it cannot be inserted in the previous hole)

| | | | | | |
|----|----|----|---|--|----|
| 17 | 15 | 16 | 7 | | 19 |
|----|----|----|---|--|----|

10



Now 10 can be inserted in the hole

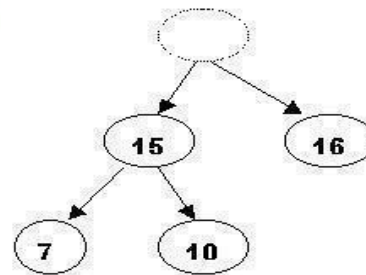


2. DeleteMax the top element 17

Store 17 in a temporary place. A hole is created at the top



17



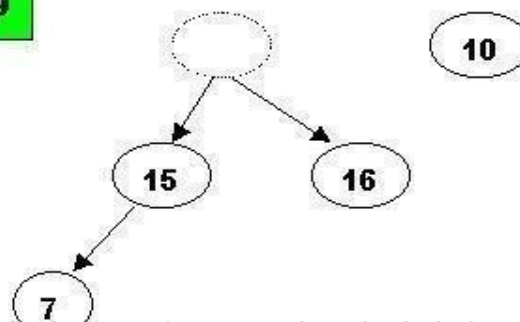
Swap 17 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the

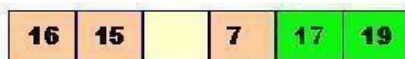
heap. Instead it becomes a cell from the sorted array



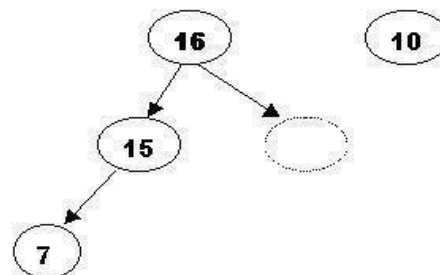
10



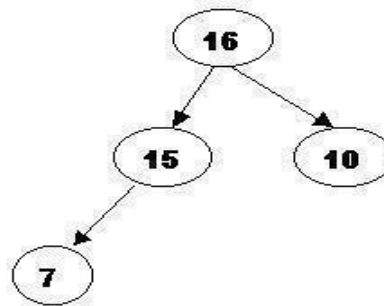
The element 10 is less than the children of the hole, and we percolate the hole down:



10

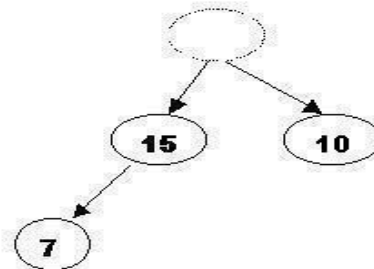


Insert 10 in the hole



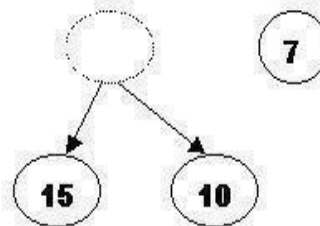
3. DeleteMax 16

Store 16 in a temporary place. A hole is created at the top

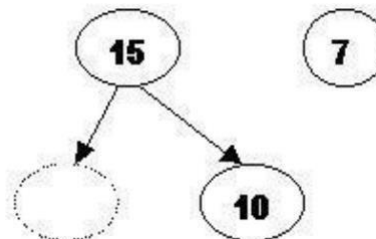


Swap 16 with the last element of the heap.

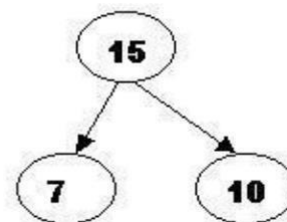
As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



Percolate the hole down (7 cannot be inserted there - it is less than the children of the hole)

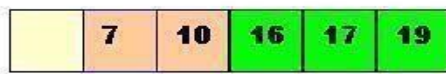


Insert 7 in the

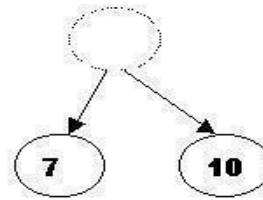


4. DeleteMax the top element 15

Store 15 in a temporary location. A hole is created.



15

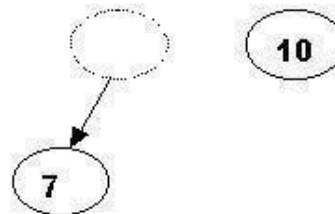


Swap 15 with the last element of the heap.

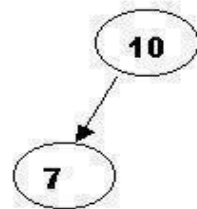
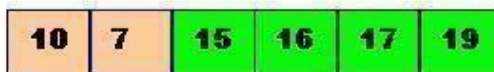
As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a position from the sorted array



10



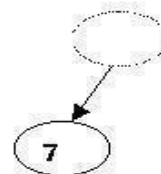
Store 10 in the hole (10 is greater than the children of the hole)

**5. DeleteMax the top element 10.**

Remove 10 from the heap and store it into a temporary location.



10



Swap 10 with the last element of the heap.

As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



7



Store 7 in the hole (as the only remaining element in the heap)



7 is the last element from the heap, so now the array is sorted



Time

Complexity:

| | |
|-------------------------------|-----------------|
| Worst Case Performance | $O(N \log_2 N)$ |
| Best Case Performance(nearly) | $O(N \log_2 N)$ |
| Average Case Performance | $O(N \log_2 N)$ |

Radix/Bucket Sort:

Bucket sort, or **bin sort**, is a sorting algorithm that works by partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a distribution sort, and is a cousin of radix sort in the most to least significant digit flavor.

Bucket sort works as follows:

1. Set up an array of initially empty "buckets".
2. **Scatter**: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. **Gather**: Visit the buckets in order and put all elements back into the original array.

Radix/Bucket Procedure:

1. Reading the list of elements. Calling the Radix sort function.
2. Checking the biggest number (big) in the list and number of digits in the biggest number (nd).
3. Inserting the numbers in to the buckets based on the one's digits and collecting the numbers and again inserting in to buckets based on the ten's digits and soon...
4. Inserting and collecting is continued 'nd' times. The elements get sorted.
5. Displaying the elements in the list after sorting.

Step-by-step example: Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: 170, 90, 802, 2, 24,

45, 75, 66 Sorting by next digit (10s place) gives:

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives: 2, 24, 45, 66, 75, 90, 170, 802

It is important to realize that each of the above steps requires just a single pass over the data, since each item can be placed in its correct bucket without having to be compared with other items.

Some LSD radix sort implementations allocate space for buckets by first counting the number of keys that belong in each bucket before moving keys into those buckets. The number of times that each digit occurs is stored in an array. Consider the previous list of keys viewed in a different way:

170, 045, 075, 090, 002, 024, 802, 066

The first counting pass starts on the least significant digit of each key, producing an array of bucket sizes:

2 (bucket size for digits of 0: 170, 090)
 2 (bucket size for digits of 2: 002, 802)
 1 (bucket size for digits of 4: 024)
 2 (bucket size for digits of 5: 045, 075)
 1 (bucket size for digits of 6: 066)

A second counting pass on the next more significant digit of each key will produce an array of bucket sizes:

2 (bucket size for digits of 0: 002, 802)
 1 (bucket size for digits of 2: 024)
 1 (bucket size for digits of 4: 045)
 1 (bucket size for digits of 6: 066)
 2 (bucket size for digits of 7: 170, 075)
 1 (bucket size for digits of 9: 090)

A third and final counting pass on the most significant digit of each key will produce an array of bucket sizes:

6 (bucket size for digits of 0: 002, 024, 045, 066, 075, 090)
 1 (bucket size for digits of 1: 170)
 1 (bucket size for digits of 8: 802)

Time Complexity:

Worst Case Performance $O(N \log_2 N)$
 Best Case Performance(nearly) $O(N \log_2 N)$
 Average Case Performance $O(N \log_2 N)$

SOURCE CODE:

```
import java.io.*;
class RadixSort
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("enter n value");
        int n=Integer.parseInt(br.readLine());
        BufferedReader br1=new BufferedReader(new
        InputStreamReader(System.in)); System.out.println("enter maximum value");
        int n1=Integer.parseInt(br1.readLine());
        int arr[]=new int[n];
        System.out.println("enter elements");
        for(int i=0;i<n;i++)
        {
            arr[i]=Integer.parseInt(br.readLine());
        }

        System.out.print("\n Unsorted array: ");
        display( arr );

        radixSort(arr,n,n1);

        System.out.print("\n Sorted array: ");
        display( arr );
    }
    static void radixSort(int[] arr, int radix, int maxDigits)
    {
        int d, j, k, m, divisor;
        java.util.LinkedList[] queue = new
        java.util.LinkedList[radix]; for( d = 0; d < radix; d++ )
            queue[d] = new java.util.LinkedList();
        divisor = 1;
        for(d = 1; d <= maxDigits; d++)
        {
            for(j = 0; j < arr.length; j++)
            {
                m = (arr[j]/divisor) % radix;
                queue[m].addLast(new Integer(arr[j]));
            }
        }
    }
}
```



```

        divisor = divisor*radix;
        for(j = k = 0; j < radix; j++)
        {
            while( !queue[j].isEmpty())
                arr[k++] = (Integer)queue[j].removeFirst();
        }
    }
    static void display( int a[] )
    {
        for( int i = 0; i < a.length; i++ )
            System.out.print( a[i] + " " );
    }
}

```

OUTPUT:

```

C:\> Command Prompt
E:\g\ads> javac RadixSort.java
Note: RadixSort.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
E:\g\ads>
E:\g\ads> java RadixSort
enter n value
10
enter maximum value
5
enter elements
46
23
85
96
99
33
24
18
20
34

Unsorted array: 46 23 85 96 99 33 24 18 20 34
Sorted array: 18 20 23 24 33 34 46 85 96 99
E:\g\ads>

```

Time complexities:

| Algorithm | Worst case | Average case | Best case |
|----------------|---------------|---------------|---------------|
| Bubble sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| selection sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Insertion sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Quick sort | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ |
| Merge sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| Heap sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| Linear search | $O(n)$ | $O(n)$ | $O(1)$ |
| Binary search | $O(\log n)$ | $O(\log n)$ | $O(1)$ |

Analysis of different sorting techniques

In this article, we will discuss important properties of different sorting techniques including their complexity, stability and memory constraints. Before understanding this article, you should understand basics of different sorting techniques

Time complexity Analysis –

We have discussed the best, average and worst case complexity of different sorting techniques with possible scenarios.

Comparison based sorting –

In comparison based sorting, elements of an array are compared with each other to find the sorted array.

- **Bubble sort and Insertion sort –**

Average and worst case time complexity: n^2

Best case time complexity: n when array is already sorted. Worst case: when the array is reverse sorted.

- **Selection sort –**

Best, average and worst case time complexity: n^2 which is independent of distribution of data.

- **Merge sort –**

Best, average and worst case time complexity: $n \log n$ which is independent of distribution of data.

- **Heap sort –**

Best, average and worst case time complexity: $n \log n$ which is independent of distribution of data.

- **Quick sort –**

It is a divide and conquer approach with recurrence relation:

- $$T(n) = T(k) + T(n-k-1) + cn$$

Worst case: when the array is sorted or reverse sorted, the partition algorithm divides the array in two subarrays with 0 and $n-1$ elements. Therefore,

$$T(n) = T(0) + T(n-1) + cn$$

Solving this we get, $T(n) = O(n^2)$

Best case and Average case: On an average, the partition algorithm divides the array in two subarrays with equal size. Therefore,

$$T(n) = 2T(n/2) + cn$$

Solving this we get, $T(n) = O(n \log n)$

Non-comparison based sorting –

In non-comparison based sorting, elements of array are not compared with each other to find the sorted array.

- **Radix sort –**

Best, average and worst case time complexity: nk where k is the maximum number of digits in elements of array.

- **Count sort –**

Best, average and worst case time complexity: $n+k$ where k is the size of count array.

- **Bucket sort –**

Best and average time complexity: $n+k$ where k is the number of buckets. Worst case time complexity: n^2 if all elements belong to same bucket.

In-place/Outplace technique –

A sorting technique is in-place if it does not use any extra memory to sort the array. Among the comparison based techniques discussed, only merge sort is out-place technique as it requires an extra array to merge the sorted subarrays.

Among the non-comparison based techniques discussed, all are out-place techniques. Counting sort uses a counting array and bucket sort uses a hash table for sorting the array.

Online/Offline technique –

A sorting technique is considered Online if it can accept new data while the procedure is ongoing i.e. complete data is not required to start the sorting operation.

Among the comparison based techniques discussed, only Insertion Sort qualifies for this because of the underlying algorithm it uses i.e. it processes the array (not just elements) from left to right and if new elements are added to the right, it doesn't impact the ongoing operation.

Stable/Unstable technique –

A sorting technique is stable if it does not change the order of elements with the same value. Out of comparison based techniques, bubble sort, insertion sort and merge sort are stable techniques. Selection sort is unstable as it may change the order of elements with the same value. For example, consider the array 4, 4, 1, 3.

In the first iteration, the minimum element found is 1 and it is swapped with 4 at 0th position. Therefore, the order of 4 with respect to 4 at the 1st position will change. Similarly, quick sort and heap sort are also unstable.

Out of non-comparison based techniques, Counting sort and Bucket sort are stable sorting techniques whereas radix sort stability depends on the underlying algorithm used for sorting.

Time and Space Complexity Comparison Table:

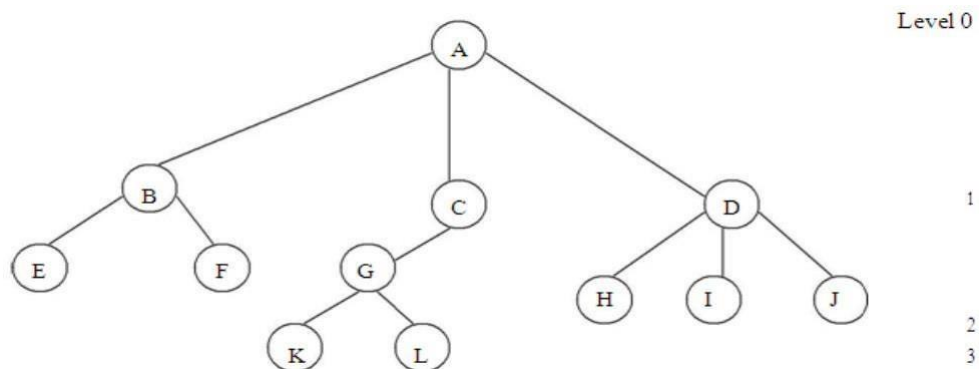
| SORTING ALGORITHM | TIME COMPLEXITY | | | SPACE COMPLEXITY |
|-------------------|--------------------|--------------------|---------------|------------------|
| | BEST CASE | AVERAGE CASE | WORST CASE | WORST CASE |
| Bubble Sort | $\Omega(N)$ | $\Theta(N^2)$ | $O(N^2)$ | $O(1)$ |
| Selection Sort | $\Omega(N^2)$ | $\Theta(N^2)$ | $O(N^2)$ | $O(1)$ |
| Insertion Sort | $\Omega(N)$ | $\Theta(N^2)$ | $O(N^2)$ | $O(1)$ |
| Merge Sort | $\Omega(N \log N)$ | $\Theta(N \log N)$ | $O(N \log N)$ | $O(N)$ |
| Heap Sort | $\Omega(N \log N)$ | $\Theta(N \log N)$ | $O(N \log N)$ | $O(1)$ |
| Quick Sort | $\Omega(N \log N)$ | $\Theta(N \log N)$ | $O(N^2)$ | $O(N \log N)$ |
| Radix Sort | $\Omega(N k)$ | $\Theta(N k)$ | $O(N k)$ | $O(N + k)$ |
| Count Sort | $\Omega(N + k)$ | $\Theta(N + k)$ | $O(N + k)$ | $O(k)$ |
| Bucket Sort | $\Omega(N + k)$ | $\Theta(N + k)$ | $O(N^2)$ | $O(N)$ |

UNIT 4

Trees- Ordinary and Binary trees terminology

TREES

A Tree is a data structure in which each element is attached to one or more elements directly beneath it.



Terminology

- The connections between elements are called **branches**.
- A tree has a single root, called **root** node, which is shown at the top of the tree. i.e. root is always at the highest level 0.
- Each node has exactly one node above it, called **parent**. Eg: A is the parent of B,C and D.
- The nodes just below a node are called its **children**. ie. child nodes are one level lower than the parent node.
- A node which does not have any child called **leaf or terminal node**. Eg: E, F, K, L, H, I and M are Leaf node. Nodes with at least one child are called **non terminal or internal nodes**.
- The child nodes of same parent are said to be **siblings**.
- A **path** in a tree is a list of distinct nodes in which successive nodes are connected by branches in the tree.
- The **length** of a particular path is the number of branches in that path. The **degree** of a node of a tree is the number of children of that node.
- The maximum number of children a node can have is often referred to as the **order** of a tree. The **height or depth** of a tree is the length of the longest path from root to any leaf.

1. **Root:** This is the unique node in the tree to which further sub trees are attached. Eg:A
2. **Degree of the node:** The total number of sub-trees attached to the node is called the degree of the node. Eg: For node A degree is 3. For node K degree is 0
3. **Leaves:** These are the terminal nodes of the tree. The nodes with degree 0 are always the leaf nodes. Eg: E, F, K, L, H, I, J
4. **Internal nodes:** The nodes other than the root node and the leaves are called the internal

nodes. Eg: B, C, D, G

5. Parent nodes: The node which is having further sub-trees(branches) is called the parent node of those sub-trees. Eg: B is the parent node of E and F.
6. Predecessor: While displaying the tree, if some particular node occurs previous to some other node then that node is called the predecessor of the other node. Eg: E is the predecessor of the node B.
7. Successor: The node which occurs next to some other node is a successor node. Eg: B is the successor of E and F.
8. Level of the tree: The root node is always considered at level 0, then its adjacent children are supposed to be at level 1 and so on. Eg: A is at level 0, B,C,D are at level 1, E,F,G,H,I,J are at level 2, K,L are at level 3.
9. Height of the tree: The maximum level is the height of the tree. Here height of the tree is 3. The height of the tree is also called depth of the tree.
10. Degree of tree: The maximum degree of the node is called the degree of the tree.

BINARY TREES

Binary tree is a tree in which each node has at most two children, a left child and a right child. Thus the order of binary tree is 2.

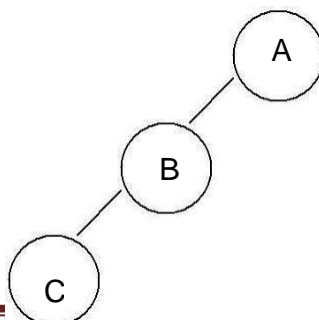
A binary tree is either empty or consists of a node called the root, left and right sub trees are themselves binary trees.

A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint trees called left sub-tree and right sub-tree.

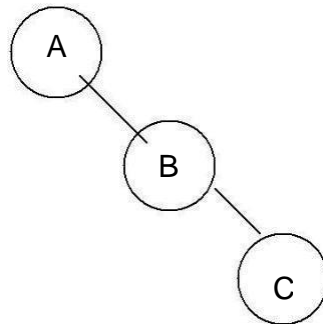
In binary tree each node will have one data field and two pointer fields for representing the sub- branches. The degree of each node in the binary tree will be at the most two.

Types Of Binary Trees: There are 3 types of binary trees:

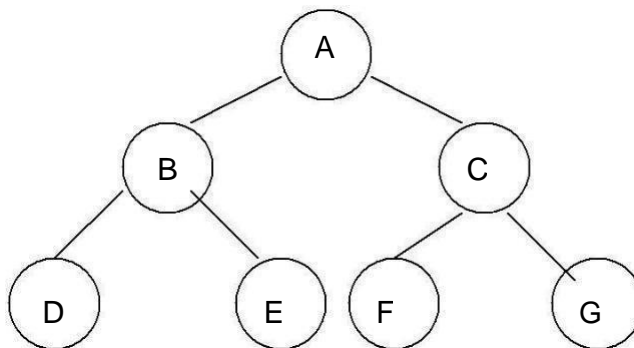
1. **Left skewed binary tree:** If the right sub-tree is missing in every node of a tree we call it as left skewed tree.



2. Right skewed binary tree: If the left sub-tree is missing in every node of a tree we call it isright CC sub-tree.



3. Complete binary tree: The tree in which degree of each node is at the most two is called a complete binary tree. In a complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. So we can say that a complete binary tree depth d will contain exactly 2^l nodes at each level l , where l is from 0 to d .

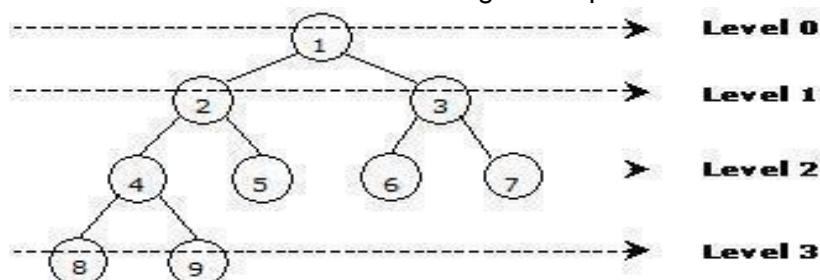


Note:

1. A binary tree of depth n will have maximum $2^n - 1$ nodes.
2. A complete binary tree of level l will have maximum 2^l nodes at each level, where l starts from 0.
3. Any binary tree with n nodes will have at the most $n+1$ null branches.
4. The total number of edges in a complete binary tree with n terminal nodes are $2(n-1)$.

Assigning level numbers and Numbering of nodes for a binary tree:

The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of an complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent.



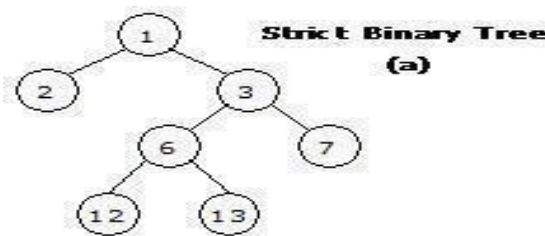
Properties of Binary Trees:

Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then
 - a. Maximum number of leaves = 2^h
 - b. Maximum number of nodes = $2^{h+1} - 1$
2. If a binary tree contains m nodes at level l , it contains at most $2m$ nodes at level $l + 1$.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l nodes at level l .
4. The total number of edges in a full binary tree with n nodes is $n - 1$.

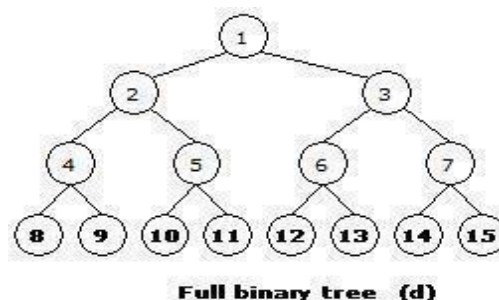
Strictly Binary tree:

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a strictly binary tree. Thus the tree of figure 7.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains $2n - 1$ nodes.



Full Binary tree:

A full binary tree of height h has all its leaves at level h . Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children. A full binary tree with height h has $2^{h+1} - 1$ nodes. A full binary tree of height h is a *strictly binary tree* all of whose leaves are at level h .



For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

Binary Tree Representation

A binary tree can be represented mainly in 2 ways:

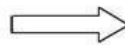
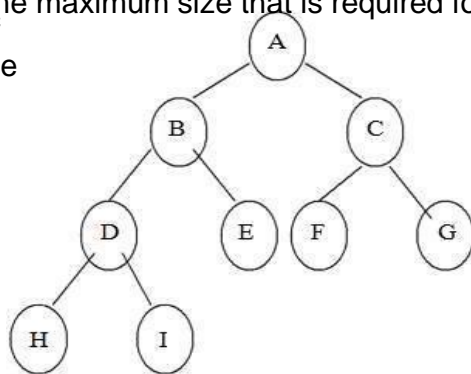
- a) Sequential Representation
- b) Linked Representation

a) Sequential Representation

The simplest way to represent binary trees in memory is the sequential representation that uses one-dimensional array.

- 1) The root of binary tree is stored in the 1st location of array
- 2) If a node is in the j^{th} location of array, then its left child is in the location $2j+1$ and its right child in the location $2j+2$

The maximum size that is required for an array to store a tree is $2^{d+1}-1$, where d is the depth of the tree



| POSITION | ARRAY |
|----------|-------|
| 0 | A |
| 1 | B |
| 2 | C |
| 3 | D |
| 4 | E |
| 5 | F |
| 6 | G |
| 7 | H |
| 8 | I |
| ... | ... |
| ... | ... |
| ... | ... |
| ... | ... |

Advantages of sequential representation:

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left children of any particular node is fast because of the random access.

Disadvantages of sequential representation:

1. The major disadvantage with this type of representation is wastage of memory. For example in the skewed tree half of the array is unutilized.
2. In this type of representation the maximum depth of the tree has to be fixed. Because we have to decide the array size. If we choose the array size quite larger than the depth of the tree, then it will be wastage of the memory. And if we choose array size lesser than the depth of the tree then we will be unable to represent some part of the tree.
3. The insertions and deletion of any node in the tree will be costlier as other nodes have to be adjusted at appropriate positions so that the meaning of binary tree can

be preserved.

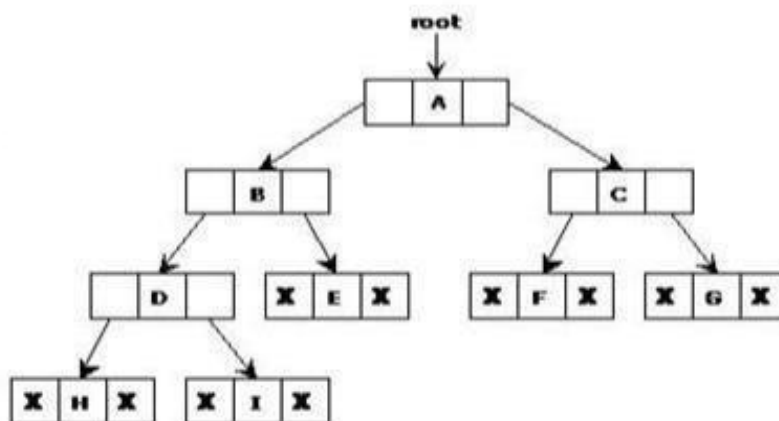
As these drawbacks are there with this sequential type of representation, we will search for more flexible representation. So instead of array we will make use of linked list to represent the tree.

b) Linked Representation

Linked representation of trees in memory is implemented using pointers. Since each node in a binary tree can have maximum two children, a node in a linked representation has two pointers for both left and right child, and one information field. If a node does not have any child, the corresponding pointer field is made NULL pointer.

In linked list each node will look like this:

| | | |
|------------|------|-------------|
| Left Child | Data | Right Child |
|------------|------|-------------|



Advantages of linked representation:

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree. Using dynamic memory concept one can create as much memory(nodes) as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.
2. Insertions and deletions which are the most common operations can be done without moving the nodes.

Disadvantages of linked representation:

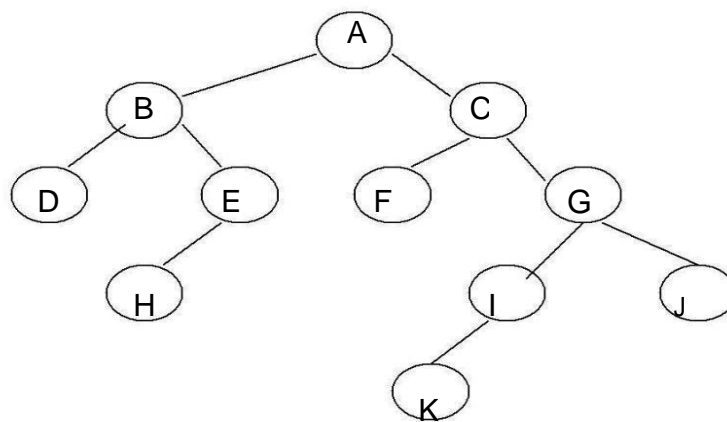
1. This representation does not provide direct access to a node and special algorithms are required.
2. This representation needs additional space in each node for storing the left and right sub-trees.

TRAVERSING A BINARY TREE

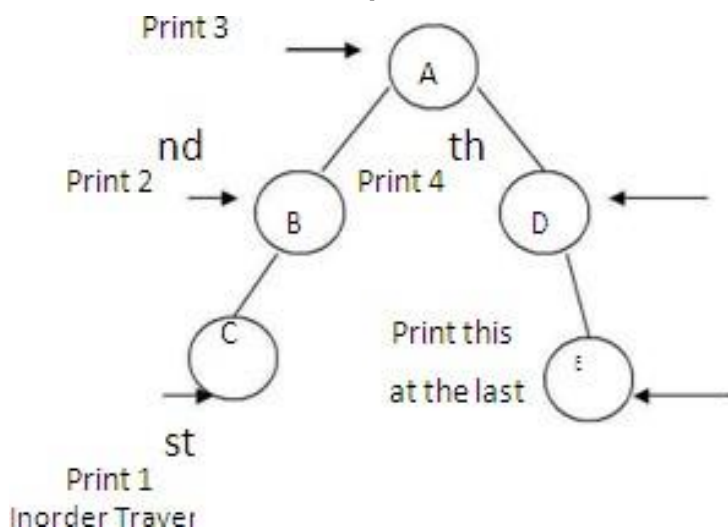
Traversing a tree means that processing it so that each node is visited exactly once. A binary tree can be traversed a number of ways. The most common tree traversals are

In-order Pre-order and Post-order

| | | |
|------------|--|--------------|
| Pre-order | 1.Visit the root Root Left Right 2.Traverse the left sub tree in pre- order 3.Traverse the right sub tree in pre-order. | |
| In-order | 1.Traverse the left sub tree in in-order Right 2.Visit the root 3.Traverse the right sub tree in in-order. | Left Root |
| Post-order | 1.Traverse the left sub tree in post-order Root 2.Traverse the right sub tree in post-order. 3.Visit the root | Left Right |



The pre-order traversal is: A B D E H F C I J K
 The post-order traversal is: D H E B F K I J G C A



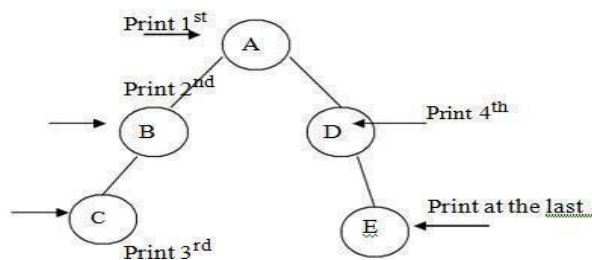
C-B-A-D-E is the inorder traversal i.e. first we go towards the leftmost node. i.e. C so print that node C. Then go back to the node B and print B. Then root node A then move towards the right sub-tree print D and finally E. Thus we are following the tracing sequence of Left|Root|Right. This type of traversal is called inorder traversal. The basic principle is to traverse left sub-tree then root

and then the right sub-tree.

Pseudo Code:

```
template <class T>
void inorder(bintree<T> *temp)
{
    if(temp!=NULL)
    {
        inorder(temp->left);
        cout<<"temp-
        >data";
        inorder(temp-
        >right);
    }
}
```

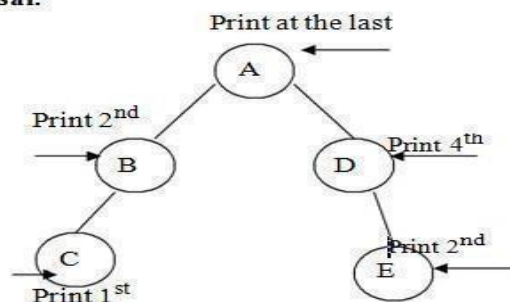
Preorder Traversal:



A-B-C-D-E is the preorder traversal of the above fig. We are following Root|Left|Right path i.e. data at the root node will be printed first then we move on the left sub-tree and go on printing the data till we reach to the left most node. Print the data at that node and then move to the right sub-tree. Follow the same principle at each sub-tree and go on printing the data accordingly.

```
template <class T>
void preorder(bintree<T> *temp)
{
    if(temp!=NULL)
    {
        cout<<"temp->data";    preorder(temp->left);
        preorder(temp->right); } }
```

Postorder Traversal:



From figure the postorder traversal is C-D-B-E-A. In the postorder traversal we are following the Left|Right|Root principle i.e. move to the leftmost node, if right sub-tree is there or not if not then print the leftmost node, if right sub-tree is there move towards the right most node. The key idea here is that at each sub-tree we are following the Left|Right|Root principle and print the data accordingly.

Pseudo Code:

```

template <class T>
void postorder(bintree<T> *temp)
{
if(temp!=NULL)
{
postorder(temp->left);
postorder(temp-
>right); cout<<"temp-
>data";
}
}

```

Source code:

Recursive and non-recursive functions to traverse the given binary tree in Preorder b) Inorder
c) Postorder.

PROGRAM:

```

class Node
{
    Object
    data;
    Node left;
    Node right;
    Node( Object d ) // constructor
    {
        data = d;
    }
}
class BinaryTree
{
    Object
    tree[]; int
    maxSize;
    java.util.Stack<Node> stk = new
    java.util.Stack<Node>(); BinaryTree( Object a[], int
    n ) // constructor
    {
        maxSize = n;
        tree = new
        Object[maxSize]; for( int
        i=0; i<maxSize; i++ )
            tree[i] = a[i];
    }
    public Node buildTree( int index )
    {
        Node p = null;

```

```
if( tree[index] != null )  
{
```

```
        p = new Node(tree[index]);
        p.left =
        buildTree(2*index+1); p.right
        = buildTree(2*index+2);
    }
    return p;
}
/* Recursive methods - Binary tree
traversals */ public void inorder(Node
p)
{
    if( p != null )
    {
        inorder(p.left);
        System.out.print(p.data + "
"); inorder(p.right);
    }
}
public void preorder(Node p)
{
    if( p != null )
    {
        System.out.print(p.data + "
"); preorder(p.left);
        preorder(p.right);
    }
}
public void postorder(Node p)
{
    if( p != null )
    {
        postorder(p.left);
        postorder(p.right);
        System.out.print(p.data + "
");
    }
}

/* Non-recursive methods - Binary tree
traversals */ public void preorderIterative(Node
p)
{
    if(p == null )
    {
        System.out.println("Tree is
empty"); return;
    }
    stk.push(p);
    while( !stk.isEmpty() )
```

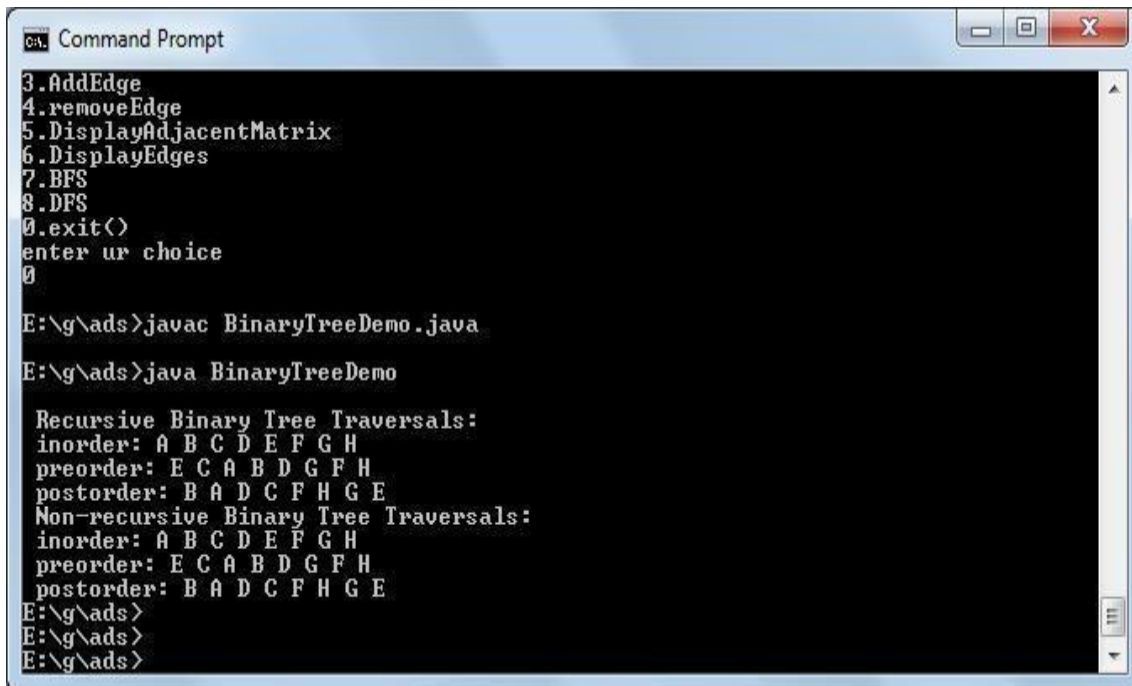
{


```
        p =
        stk.pop(); if(
        p != null )
        {
            System.out.print(p.data + " ");
            stk.push(p.right);
            stk.push(p.left);
        }
    }
}
public void inorderIterative(Node p)
{
    if(p == null )
    {
        System.out.println("Tree is
        empty"); return;
    }
    while( !stk.isEmpty() || p != null )
    {
        if( p != null )
        {
            stk.push(p); // push left-most path onto
            stack p = p.left;
        }
        else
        {
            p = stk.pop(); // assign popped node to p
            System.out.print(p.data + " "); // print node
            data p = p.right; // move p to right subtree
        }
    }
}
}
public void postorderIterative(Node p)
{
    if(p == null )
    {
        System.out.println("Tree is
        empty"); return;
    }
    Node tmp = p;
    while( p != null
    )
    {
        while( p.left != null )
        {
            stk.push(p
            ); p =
            p.left;
```

}

```
        while( p != null && (p.right == null || p.right == tmp ))
        {
            System.out.print(p.data + " "); // print node data
            tmp = p;
            if( stk.isEmpty() )
                return;
            p = stk.pop();
        }
        stk.push(p
    ); p =
        p.right;
    }
}
} // end of BinaryTree class

class BinaryTreeDemo
{
    public static void main(String args[])
    {
        Object arr[] = {'E', 'C', 'G', 'A', 'D', 'F', 'H', null, 'B',
            null, null, null, null, null, null, null, null };
        BinaryTree t = new BinaryTree( arr, arr.length );
        Node root = t.buildTree(0); // buildTree() returns reference to root
        System.out.print("\n Recursive Binary Tree Traversals:");
        System.out.print("\n inorder: ");
        t.inorder(root);
        System.out.print("\n preorder: ");
        t.preorder(root);
        System.out.print("\n postorder:
    "); t.postorder(root);
        System.out.print("\n Non-recursive Binary Tree
    Traversals:"); System.out.print("\n inorder: ");
        t.inorderIterative(root);
        System.out.print("\n preorder: ");
        t.preorderIterative(root);
        System.out.print("\n postorder: ");
        t.postorderIterative(root);
    }
}
```

OUTPUT:

```
Command Prompt
3.AddEdge
4.removeEdge
5.DisplayAdjacentMatrix
6.DisplayEdges
7.BFS
8.DFS
0.exit()
enter ur choice
0

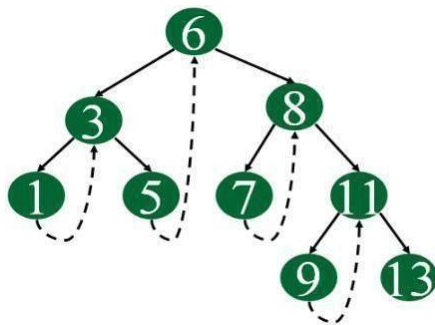
E:\g\ads>javac BinaryTreeDemo.java
E:\g\ads>java BinaryTreeDemo

Recursive Binary Tree Traversals:
inorder: A B C D E F G H
preorder: E C A B D G F H
postorder: B A D C F H G E
Non-recursive Binary Tree Traversals:
inorder: A B C D E F G H
preorder: E C A B D G F H
postorder: B A D C F H G E
E:\g\ads>
E:\g\ads>
E:\g\ads>
```

Threaded Binary Tree

Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists). There are two types of threaded binary trees.

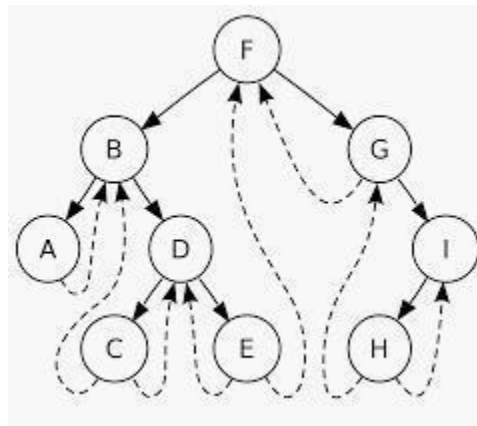
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)



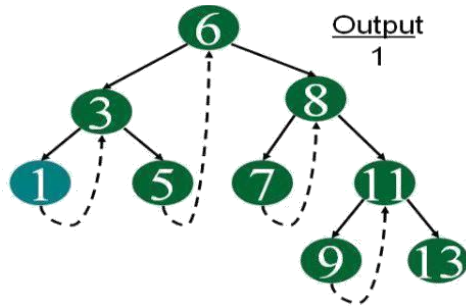
Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

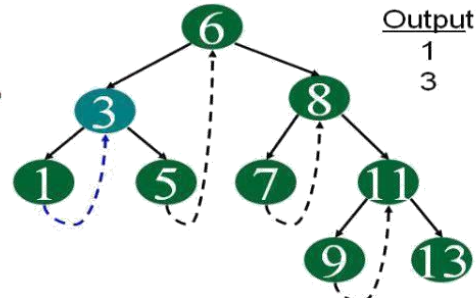
Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



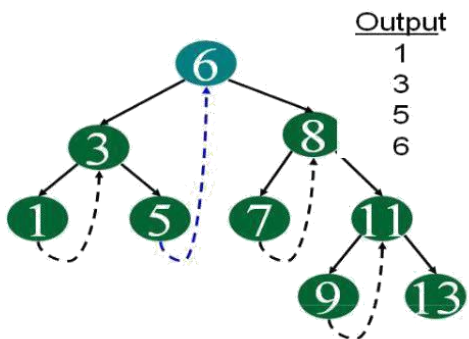
Following diagram demonstrates inorder order traversal using threads.



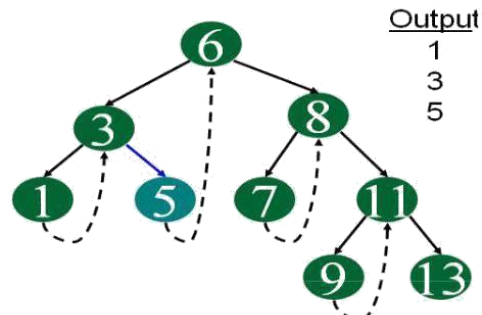
Start at leftmost node, print it



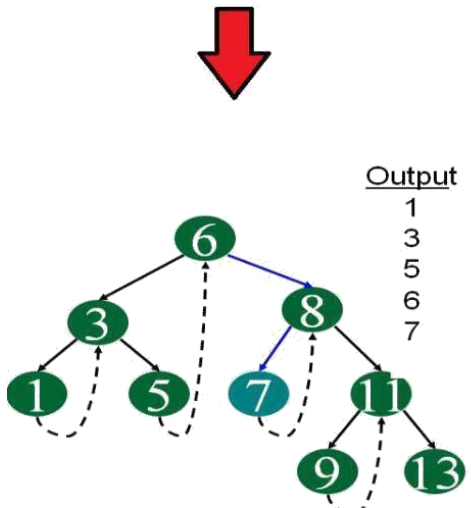
Follow thread to right, print node



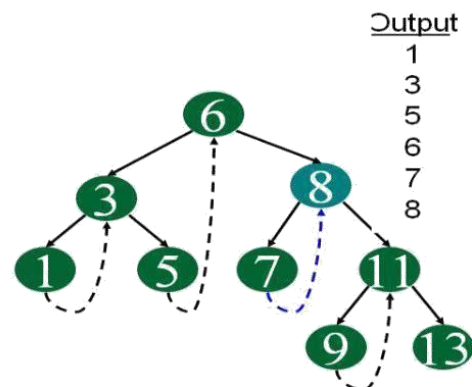
Follow thread to right, print node



Follow link to right, go to leftmost node and print



Follow link to right, go to leftmost node and print



Follow thread to right, print node

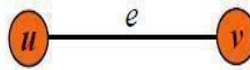
continue same way for remaining node.....

Terminology of Graph

Graphs:-

A graph G is a discrete structure consisting of nodes (called vertices) and lines joining the nodes (called edges). Two vertices are adjacent to each other if they are joined by an edge. The edge joining the two vertices is said to be an edge incident with them. We use $V(G)$ and $E(G)$ to denote the set of vertices and edges of G respectively.

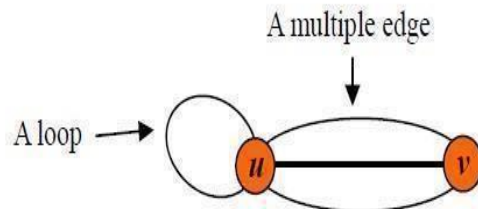
Example



u and v are adjacent vertices; e is an edge incident with u and v . e can also be denoted by uv or vu .

Loops and Multiple Edges

An edge joining only one vertex is called a *loop*. If there are more than one edge joining u and v of G , then all edges joining u and v form a *multiple edge* of G .



Simple Graph

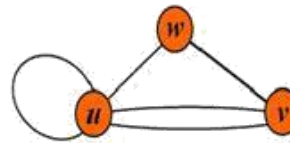
A *simple graph* is a graph containing no loops and multiple edges.

Degrees of Vertices

The degree of a vertex is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex v is denoted by $\deg(v)$ or $d(v)$.

Example

$$d(u)=5, \quad d(v)=3 \quad \text{and} \quad d(w)=2$$



Complete Graphs

The *complete graph* on n vertices, denoted by K_n , is the simple graph in which any pair of vertices are adjacent.

Examples

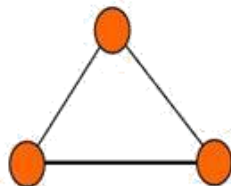
I) K_1



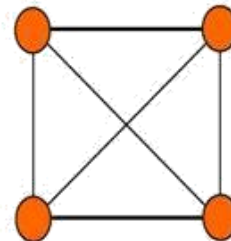
II) K_2



III) K_3



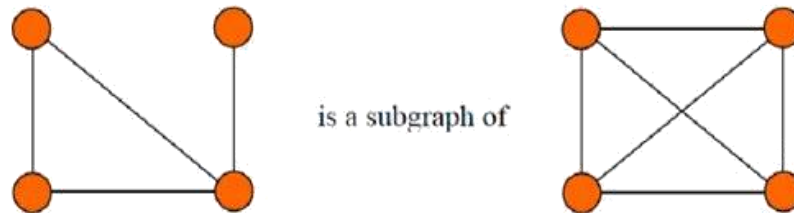
IV) K_4



Subgraphs

A *subgraph* of a graph G is a graph H where $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

Example

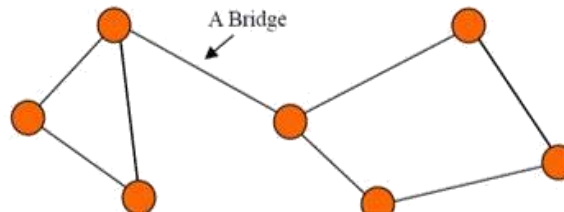


Connected Graphs

A graph is *connected* if there is a path between every pair of distinct vertices of the graph. An edge uv in a connected graph G is called a *bridge* if $G - uv$, the graph obtained by deleting uv from G , is not connected.

Example

A connected graph



Graph Representations

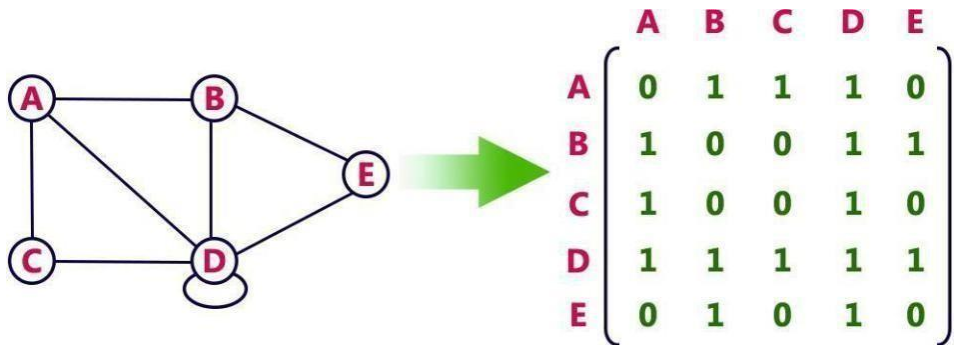
Graph data structure is represented using following representations...

1. Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

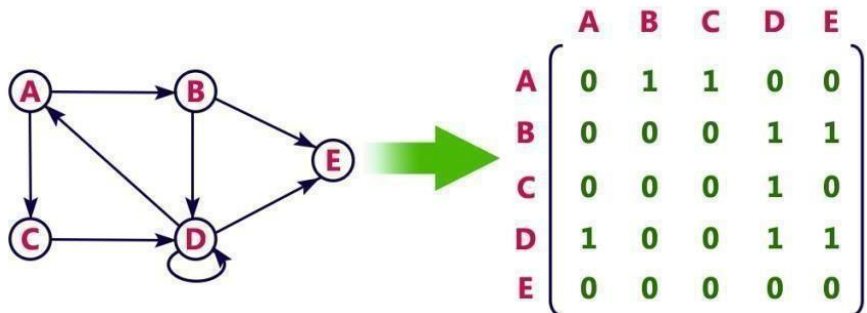
Adjacency Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



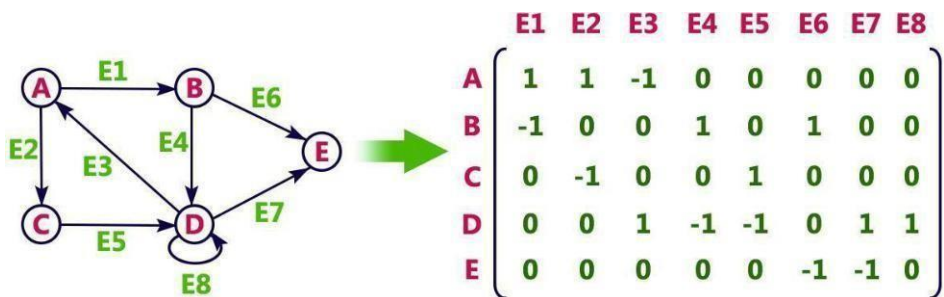
Directed graph representation...



Incidence Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

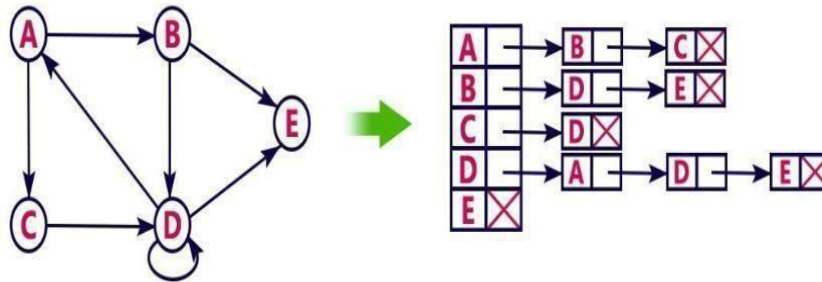
For example, consider the following directed graph representation...



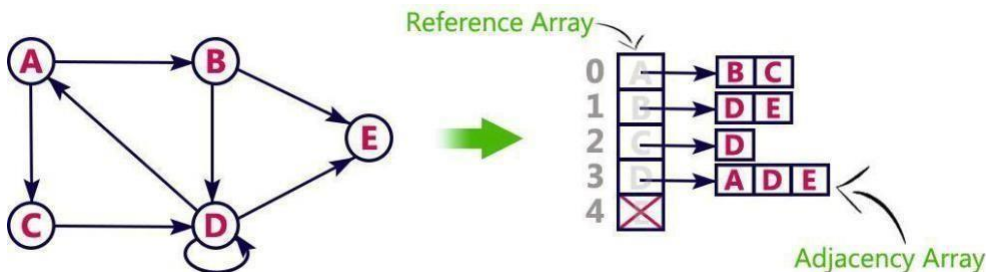
Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



Graph traversals

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking. Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

DFS-iterative(G,s); //Where G is graph and s is source vertex

Let S be stack

S.push(s) //Inserting s in stack

mark s as visited.

while (S is not

empty):

```
//Pop a vertex from stack to visit
```

```
next v = S.top( )
```

S.pop()

```
//Push all the neighbours of v in stack that
are not visited for all neighbours w of v in
Graph G:
```

if w is not visited :

S.push(w)

mark w as visited

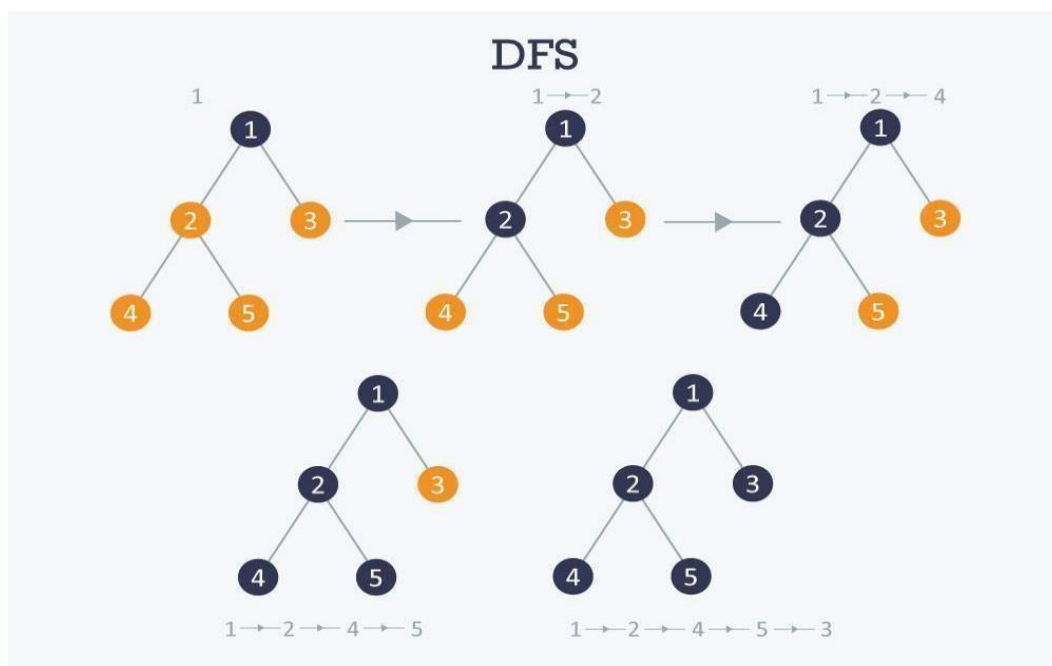
DFS-recursive(G, s):

mark s as visited

for all neighbours w of s in

Graph G: if w is not visited:

DFS-recursive(G, w)



Applications of Depth First Search

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph. Following are the problems that use DFS as a building block.

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See this for details)

3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z .

- i) Call $\text{DFS}(G, u)$ with u as the start vertex.
 - ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
 - iii) As soon as destination vertex z is encountered, return the path as the contents of the stack
- See this for details.

4) Topological Sorting See this for details.

5) To test if a graph is bipartite

We can augment either BFS or DFS when we first discover a new vertex, color it opposite to its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

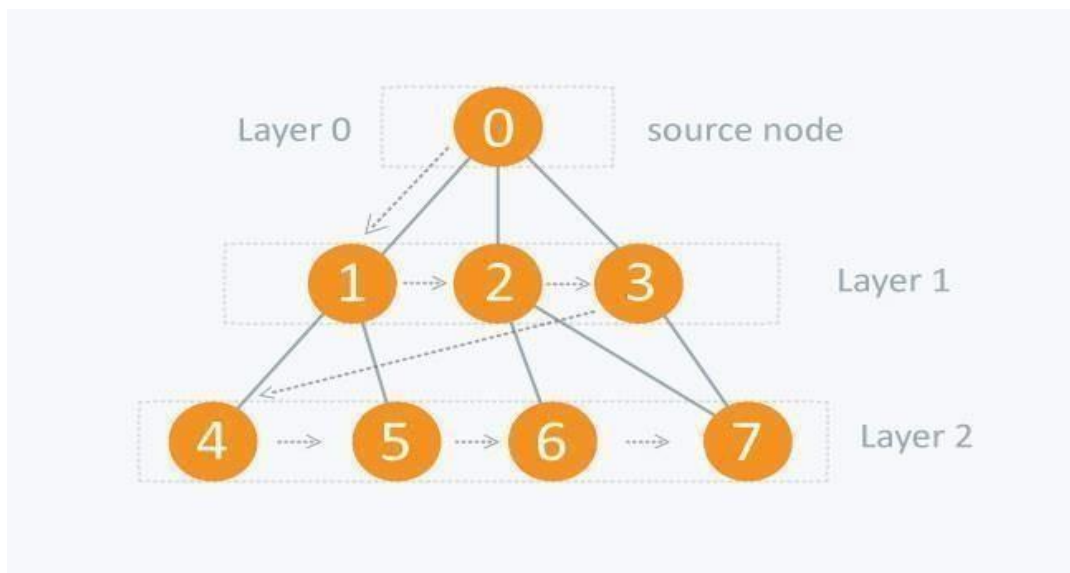
6) Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based algorithm for finding Strongly Connected Components)

7) Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

Breadth First Search (BFS);

There are many ways to traverse graphs. BFS is the most commonly used approach. BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes. As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer



Applications of Breadth First Traversal

We have earlier discussed Breadth First Traversal Algorithm for Graphs. We have also discussed Applications of Depth First Traversal. In this article, applications of Breadth First Search are discussed.

1) Shortest Path and Minimum Spanning Tree for unweighted graph In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) Crawlers in Search Engines: Crawlers build index using Bread First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.

4) Social Networking Websites: In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.

5) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.

6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) In Garbage Collection: Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference.

8) Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

9) Ford–Fulkerson algorithm In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.

10) To test if a graph is Bipartite We can either use Breadth First or Depth First Traversal.

11) Path Finding We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Many algorithms like Prim's Minimum Spanning Tree and Dijkstra's Single Source Shortest Path use structure similar to Breadth First Search.

There can be many more applications as Breadth First Search is one of the core algorithm for Graphs.

Source code for BFS & DFS

Java programs for the implementation of bfs and dfs for a given graph.

```
//bfs
import java.io.*;
class quelist
{
    public int front;
    public int rear;
    public int maxsize;
    public int[] que;
```

```
public queList(int size)
{
    maxsize = size;
    que = new int[size];
    front = rear = -1;
}

public void display()
{
    for(int i = front; i <= rear; i++)
        System.out.print(que[i] + " ");
}

public void enqueue(int x)
{
    if(front == -1)
        front = 0;
    que[++rear] = x;
}

public int dequeue()
{
    int temp = que[front];
    front = front + 1;
    return temp;
}

public boolean isEmpty()
{
    return ((front > rear) || (front == -1));
}
}

class vertex
{
    public char label;
    public boolean wasvisited;

    public vertex(char lab)
    {
        label = lab;
        wasvisited = false;
    }
}
```



```
class graph
{
    public final int MAX = 20;
    public int nverts;
    public int adj[][];
    public vertex vlist[];
    quelist qu;

    public graph()
    {
        nverts = 0;
        vlist = new vertex[MAX];
        adj = new int[MAX][MAX];
        qu = new quelist(MAX);
        for(int i=0;i<MAX;i++)
            for(int j=0;j<MAX;j++)
                adj[i][j] = 0;
    }
    public void addver(char lab)
    {
        vlist[nverts++] = new vertex(lab);
    }

    public void addedge(int start,int end)
    {
        adj[start][end] = 1;
        adj[end][start] = 1;
    }

    public int getadjunvis(int i)
    {
        for(int j=0;j<nverts;j++)
            if((adj[i][j]==1)&&(vlist[j].wasvisited==false))
                return j;
        return (MAX+1);
    }

    public void display(int i)
    {
        System.out.print(vlist[i].label);
    }

    public int getind(char l)
    {
        for(int i=0;i<nverts;i++)
            if(vlist[i].label==l)
```

```
        return i;
    return (MAX+1);
}

public void brfs()
{
    vlist[0].wasvisited = true;
    display(0);
    qu.enqueue(0);
    int v2;
    while(!(qu.isEmpty()))
    {
        int v1 = qu.dequeue();
        while((v2=getadjunvis(v1))!=(MAX+1))
        {
            vlist[v2].wasvisited = true;
            display(v2);
            qu.enqueue(v2);
        }
    }
    System.out.print("\n");
}
}
class bfs
{
    public static void main(String args[])throws IOException
    {
        graph gr = new graph();
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        System.out.println("Enter the number of vertices");
        int n = Integer.parseInt(br.readLine());
        System.out.println("Enter the labels for the
vertices"); for(int i=0;i<n;i++)
        {
            String temp = br.readLine();
            char ch = temp.charAt(0);
            gr.addver(ch);
        }
        System.out.println("Enter the number of edges");
        int edg = Integer.parseInt(br.readLine());
        System.out.println("Enter the vertices which you need to connect");
        for(int j=0;j<edg;j++)
        {
            System.out.println("Enter the first vertex");
            String t = br.readLine();
```

```

char c = t.charAt(0);
int start = gr.getind(c);

System.out.println("Enter the second vertex");
t = br.readLine();
c = t.charAt(0);
int end = gr.getind(c);

gr.addedge(start,end);
}
System.out.print("The vertices in the graph traversed
breadthwise:"); gr.brfs();
}
}

```

OUTPUT:

```

C:\> Command Prompt

E:\g\ads>javac bfs.java
E:\g\ads>java bfs
Enter the number of vertices
5
Enter the labels for the vertices
1
2
3
4
5
Enter the number of edges
6
Enter the vertices which you need to connect
Enter the first vertex
1
Enter the second vertex
2
Enter the first vertex
2
Enter the second vertex
3
Enter the first vertex
3
Enter the second vertex
4
Enter the first vertex
4
Enter the second vertex
5
Enter the first vertex
1
Enter the second vertex
5
Enter the first vertex
1
Enter the second vertex
4
The vertices in the graph traversed breadthwise:12453
E:\g\ads>_

```

//dfs

```

import java.io.*;
import java.util.*;

```

```

class Stack
{
    int stk[]=new int[10];
    int top;

```

```
Stack()
{
    top=-1;
}
void push (int item)
{
    if (top==9)
        System.out.println("Stack overflow");
    else
        stk[++top]=item;
}/*end push*/

boolean isempty()
{
    if (top<0)
        return true;
    else
        return false;
}/*end isempty*/

int pop()
{
    if (isempty())
    {
        System.out.println("Stack underflow");
        return 0;
    }
    else
        return (stk[top--]);
}/*end pop*/

void stacktop()
{
    if(isempty())
        System.out.println("Stack underflow ");
    else
        System.out.println("Stack top is "+(stk[top]));
}/*end stacktop*/

void display()
{
    System.out.println("Stack-->");
    for(int i=0;i<=top;i++)
        System.out.println(stk[i]);
}/*end display*/
}
```

```

class Graph
{
    int MAXSIZE=51;
    int adj[][]=new int[MAXSIZE][MAXSIZE];
    int visited[]=new int [MAXSIZE];
    Stack s=new Stack();
    /*Function for Depth-First-Search    */

    void createGraph()
    {
        int n,i,j,parent,adj_parent,initial_node;
        int ans=0,ans1=0;

        System.out.print("\nEnter total number elements in a Undirected Graph
        :"); n=getNumber();
        for ( i=1;i<=n;i++)
            for( j=1;j<=n;j++)
                adj[i][j]=0;

        /*All graph nodes are unvisited, hence assigned zero to visited field of each node
        */ for (int c=1;c<=50;c++)
            visited[c]=0;
        System.out.println("\nEnter graph structure for BFS ");

        do
        {
            System.out.print("\nEnter parent node :");
            parent=getNumber();
            do
            {
                System.out.print("\nEnter adjacent node for node "+parent+ " :
                "); adj_parent=getNumber();
                adj[parent][adj_parent]=1;
                adj[adj_parent][parent]=1;
                System.out.print("\nContinue to add adjacent node for "+parent+"(1/0)?");
                ans1= getNumber();
            } while (ans1==1);
            System.out.print("\nContinue to add graph
            node?"); ans= getNumber();
        }while (ans ==1);

        System.out.print("\nAdjacency matrix for your graph is
        :\n"); for (i=1;i<=n;i++)
        {

```

```
        for (j=1;j<=n;j++)
            System.out.print(" "+adj[i][j]);
        System.out.print("\n");
    }

    System.out.println("\nYour Undirected Graph is
:"); for (i=1;i<=n;i++)
    {
        System.out.print("\nVertex "+i+"is connected to :
"); for (j=1;j<=n;j++)
        {
            if (adj[i][j]==1)
                System.out.print(" "+j);
        }
    }
    System.out.println("\nEnter the initial node for BFS
traversal:"); initial_node=getNumber();
    DFS (initial_node, n);
}

void DFS (int initial_node,int n)
{
    int u,i;
    s.top = -1;
    s.push(initial_node);
    System.out.println("\nDFS traversal for given graph is : ");
    while(!s.isEmpty())
    {
        u=s.pop();
        if(visited[u]==0)
        {
            System.out.print("\n"+u);
            visited[u]=1;
        }
        for (i=1;i<=n;i++)
        {
            if((adj[u][i]==1) && (visited[i]==0))
            {
                s.push(u);
                visited[i]=1;
                System.out.print(" "+i);
                u = i;
            }
        }
    }
}
}/* end of DFS function */
```

```

int getNumber()
{
    String str;
    int ne=0;
    InputStreamReader input=new InputStreamReader(System.in);
    BufferedReader in=new BufferedReader(input);
    try
    {
        str=in.readLine();
        ne=Integer.parseInt(str);
    }
    catch(Exception e)
    {
        System.out.println("I/O Error");
    }
    return ne;  }}

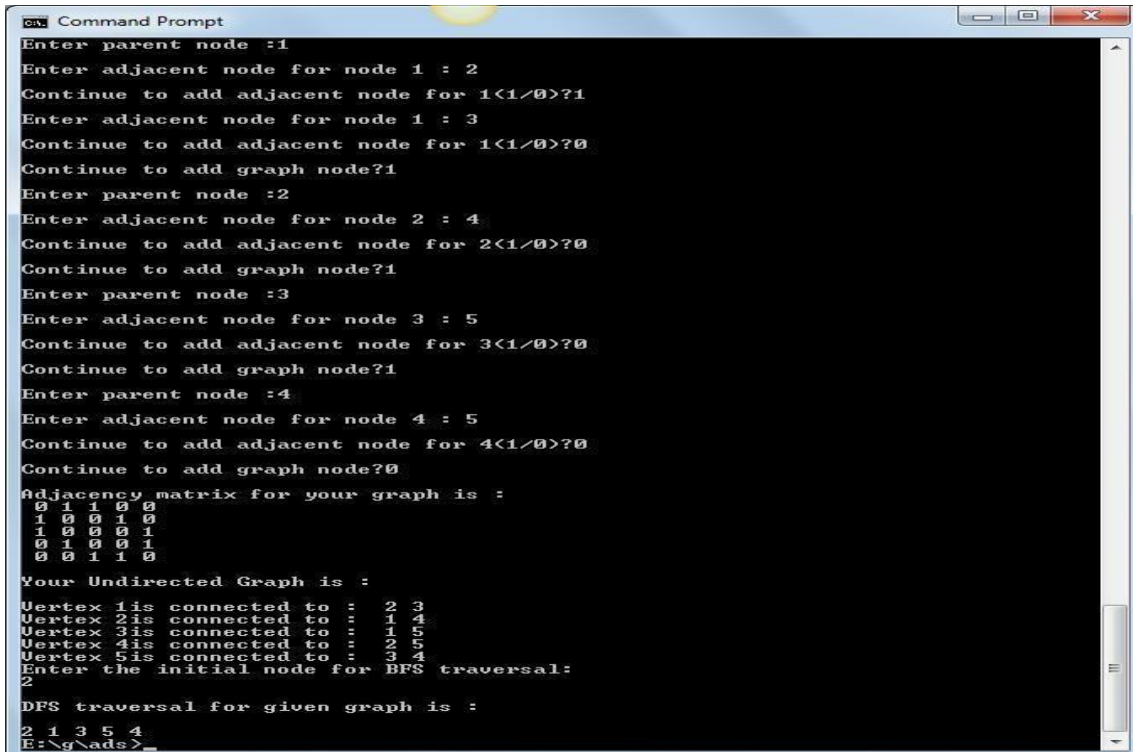
```

```

class Graph_DFS
{
    public static void main(String args[])
    {
        Graph g=new Graph();
        g.createGraph(); }      /* end of program */}

```

OUTPUT:



```

C:\> Command Prompt
Enter parent node :1
Enter adjacent node for node 1 : 2
Continue to add adjacent node for 1<1/0>?1
Enter adjacent node for node 1 : 3
Continue to add adjacent node for 1<1/0>?0
Continue to add graph node?1
Enter parent node :2
Enter adjacent node for node 2 : 4
Continue to add adjacent node for 2<1/0>?0
Continue to add graph node?1
Enter parent node :3
Enter adjacent node for node 3 : 5
Continue to add adjacent node for 3<1/0>?0
Continue to add graph node?1
Enter parent node :4
Enter adjacent node for node 4 : 5
Continue to add adjacent node for 4<1/0>?0
Continue to add graph node?0
Adjacency matrix for your graph is :
0 1 1 0 0
1 0 0 1 0
1 0 0 0 1
0 1 0 0 1
0 0 1 1 0
Your Undirected Graph is :
Vertex 1is connected to : 2 3
Vertex 2is connected to : 1 4
Vertex 3is connected to : 1 5
Vertex 4is connected to : 2 5
Vertex 5is connected to : 3 4
Enter the initial node for BFS traversal:
2
DFS traversal for given graph is :
2 1 3 5 4
E:\g\ads>

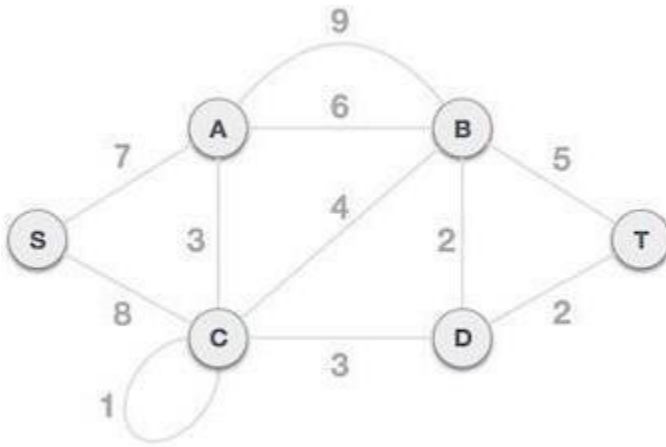
```

Applications of Graphs

Minimum cost spanning tree using Kruskal's algorithm

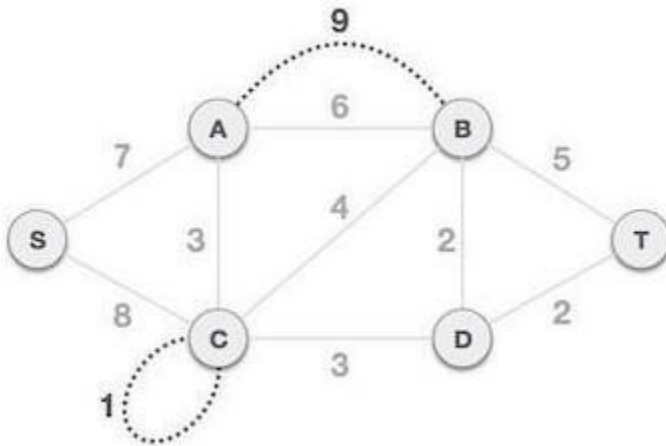
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example –

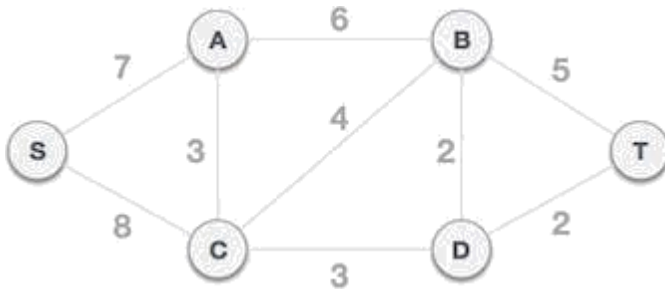


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



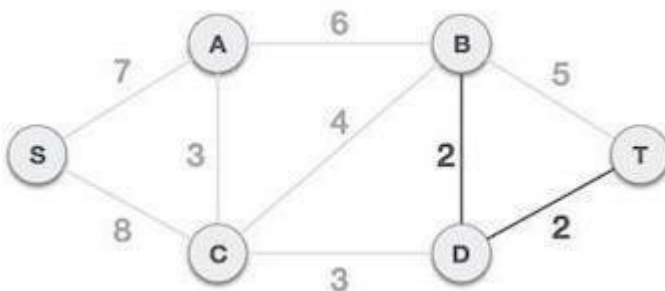
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

| | | | | | | | | |
|------|------|------|------|------|------|------|------|------|
| B, D | D, T | A, C | C, D | C, B | B, T | A, B | S, A | S, C |
| 2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 |

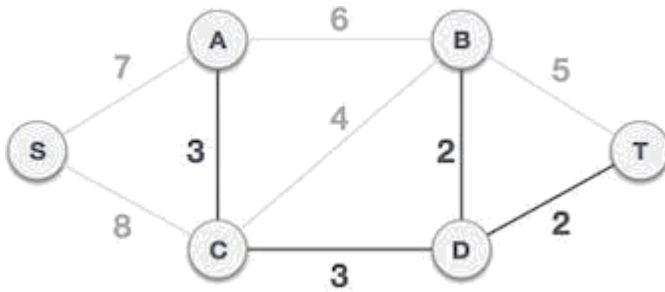
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

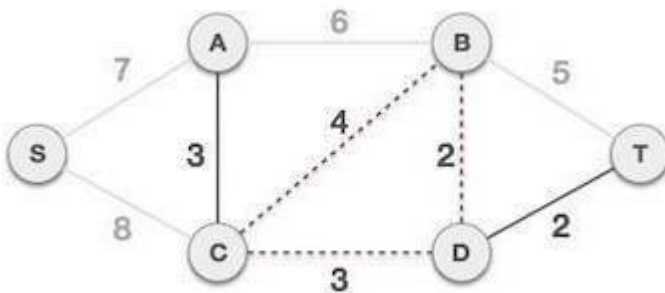


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

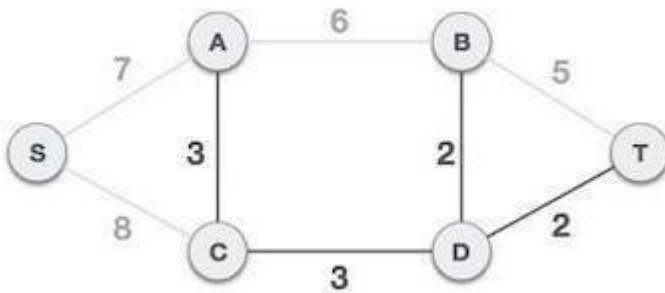
Next cost is 3, and associated edges are A,C and C,D. We add them again –



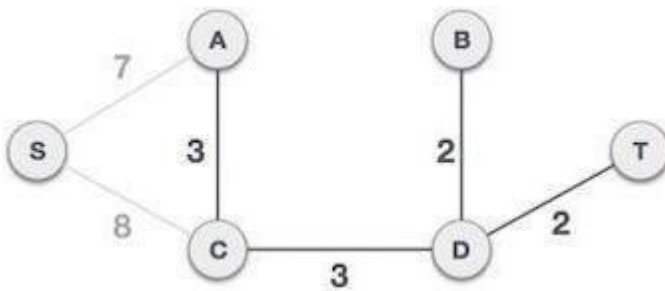
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.



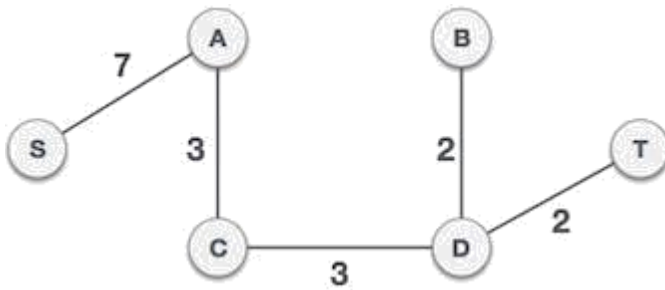
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

ANOTHER EXAMPLE

EX2:

What is Minimum Spanning Tree?

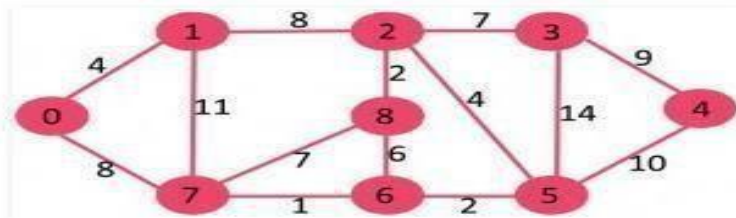
Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?

A minimum spanning tree has $(V - 1)$ edges where V is the number of vertices in the given graph. Below are the steps for finding MST using Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are $(V-1)$ edges in the spanning tree.

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having $(9 - 1) = 8$ edges.

After sorting:

| Weight | Src | Dest |
|--------|-----|------|
| 1 | 7 | 6 |
| 2 | 8 | 2 |
| 2 | 6 | 5 |
| 4 | 0 | 1 |
| 4 | 2 | 5 |
| 6 | 8 | 6 |
| 7 | 2 | 3 |
| 7 | 7 | 8 |
| 8 | 0 | 7 |
| 8 | 1 | 2 |
| 9 | 3 | 4 |
| 10 | 5 | 4 |
| 11 | 1 | 7 |
| 14 | 3 | 5 |

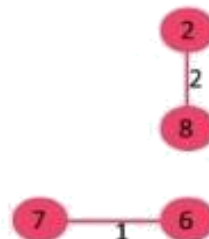
Now pick all edges one by one from sorted list of edges

Time Complexity: $O(E \log E)$ or $O(E \log V)$. Sorting of edges takes $O(E \log E)$ time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost $O(\log V)$ time. So overall complexity is $O(E \log E + E \log V)$ time. The value of E can be atmost $O(V^2)$, so $O(\log V)$ are $O(\log E)$ same. Therefore, overall time complexity is $O(E \log E)$ or $O(E \log V)$

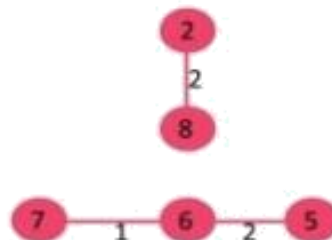
1. *Pick edge 7-6:* No cycle is formed, include it.



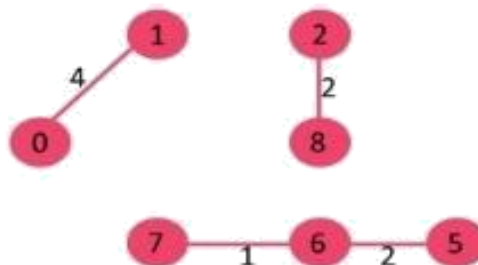
2. *Pick edge 8-2:* No cycle is formed, include it.



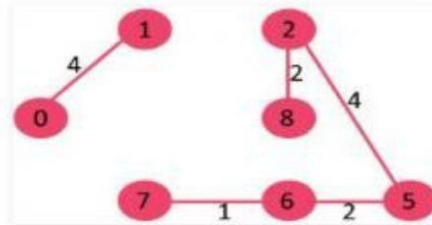
3. *Pick edge 6-5:* No cycle is formed, include it.



4. *Pick edge 0-1:* No cycle is formed, include it.

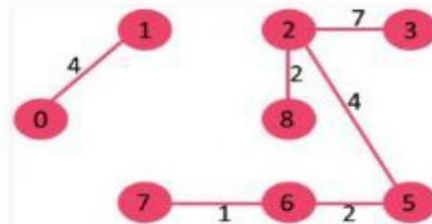


5. Pick edge 2-5: No cycle is formed, include it.



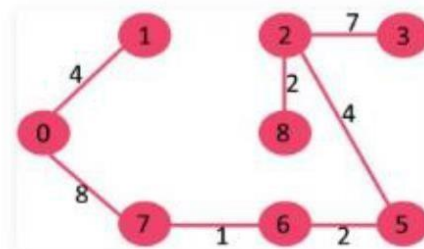
6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.



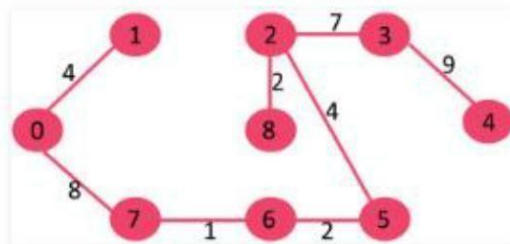
8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals $(V - 1)$, the algorithm stops here.

Java program that implements Kruskal's algorithm to generate minimum cost spanning tree.

SOURCE CODE:

```
import java.io.*;
import
java.util.*; class
Graph
{
    int i,n; //no of nodes
    int noe; //no edges in the graph
    int graph_edge[][]=new
    int[100][4]; int tree[][]=new int
    [10][10];
    int sets[][]=new int[100][10];
    int top[]=new int[100];
    int cost=0;
    int getNumber()
    {
        String
        str; int
        ne=0;
        InputStreamReader input=new
        InputStreamReader(System.in); BufferedReader in=new
        BufferedReader(input); try
        {
            str=in.readLine();
            ne=Integer.parseInt(st
            r);
        }
        catch(Exception e)
        {
            System.out.println("I/O Error");
        }
        return ne;
    }/*end getNumber*/
    void read_graph()
    {
        System.out.print("Enter the no. of nodes in the undirected weighted
graph
::");
        n=getNumber();

        noe=0;

        System.out.println("Enter the weights for the following edges
::\n"); for(int i=1;i<=n;i++)
        {
            for(int j=i+1;j<=n;j++)
            {
                System.out.print(" < "+i+" , "+j+" >
```

```
::"); int w;  
w=getNumber();
```



```

        if(w!=0
        )
        {
            noe++;
            graph_edge[noe][1]=i
            ;
            graph_edge[noe][2]=j
            ;
            graph_edge[noe][3]=
            w;
        }
    }
}

void sort_edges()
{
    /**** Sort the edges using bubble sort in increasing order*****/

    for(int i=1;i<=noe-1;i++)
    {
        for(int j=1;j<=noe-i;j++)
        {
            if(graph_edge[j][3]>graph_edge[j+1][3])
            {
                int t=graph_edge[j][1];
                graph_edge[j][1]=graph_edge[j+1][1];
                graph_edge[j+1][1]=t;

                t=graph_edge[j][2];
                graph_edge[j][2]=graph_edge[j+1][2];
                graph_edge[j+1][2]=t;

                t=graph_edge[j][3];
                graph_edge[j][3]=graph_edge[j+1][3];
                graph_edge[j+1][3]=t;
            }
        }
    }
}

void algorithm()
{
    // ->make a set for each node for(int
    i=1;i<=n;i++)
    {
        sets[i][1]=i
        ; top[i]=1;
    }

    System.out.println("\n\nThe algorithm starts ::\n\n");

    for(i=1;i<=noe;i++)
    {

```

```

int
p1=find_node(graph_edge[i][1]);
int
p2=find_node(graph_edge[i][2]);

if(p1!=p2)
{
    System.out.print("The edge included in the tree is
    ::"); System.out.print("< "+graph_edge[i][1]+" , ");
    System.out.println(graph_edge[i][2]+" > ");
    cost=cost+graph_edge[i][3];

    tree[graph_edge[i][1]][graph_edge[i][2]]=graph_edge[i]
    [3];
    tree[graph_edge[i][2]][graph_edge[i][1]]=graph_edge[i]
    [3];

    // Mix the two sets

    for(int
    j=1;j<=top[p2];j++)
    {
        top[p1]++;
        sets[p1][top[p1]]=sets[p2][j];
    }
    top[p2]=0;
}
else
e
{
    System.out.println("Inclusion of the edge ");
    System.out.print(" < "+graph_edge[i][1]+" , ");
    System.out.println(graph_edge[i][2]+"> forms a
    cycle
    so it is
    removed\n\n");
}

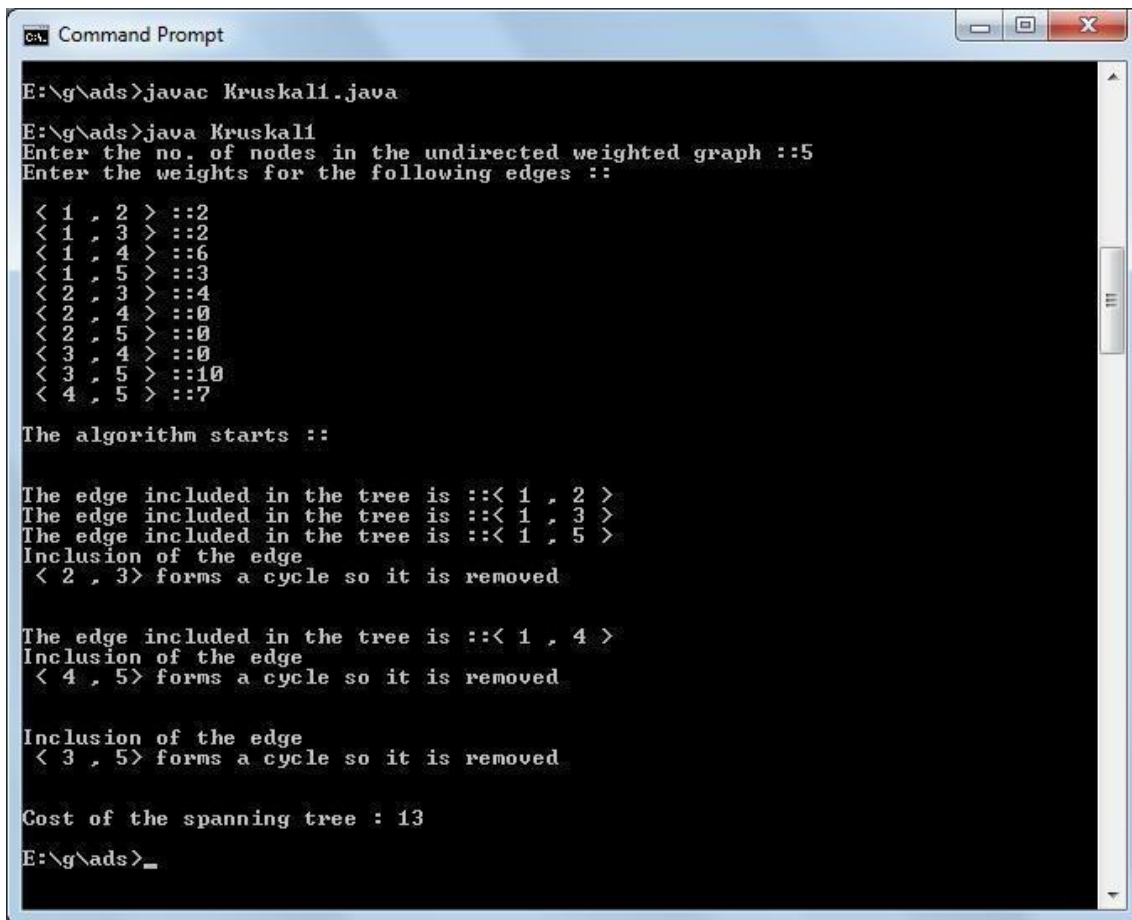
System.out.println("Cost of the spanning tree : "+cost);
}

int find_node(int n)
{
    for(int i=1;i<=noe;i++)
    {
        for(int j=1;j<=top[i];j++)
        {
            if(n==sets[i][j])
                return i;
        }
    }
}

```

```
        return -1;  
    }  
}
```

```
class Kruskal1
{
    public static void main(String args[])
    {
        Graph obj=new
        Graph();
        obj.read_graph();
        obj.sort_edges();
        obj.algorithm();
    }
}
```

OUTPUT:

```
Command Prompt

E:\g\ads>javac Kruskal1.java

E:\g\ads>java Kruskal1
Enter the no. of nodes in the undirected weighted graph ::5
Enter the weights for the following edges ::

< 1 , 2 > ::2
< 1 , 3 > ::2
< 1 , 4 > ::6
< 1 , 5 > ::3
< 2 , 3 > ::4
< 2 , 4 > ::0
< 2 , 5 > ::0
< 3 , 4 > ::0
< 3 , 5 > ::10
< 4 , 5 > ::7

The algorithm starts ::

The edge included in the tree is ::< 1 , 2 >
The edge included in the tree is ::< 1 , 3 >
The edge included in the tree is ::< 1 , 5 >
Inclusion of the edge
< 2 , 3 > forms a cycle so it is removed

The edge included in the tree is ::< 1 , 4 >
Inclusion of the edge
< 4 , 5 > forms a cycle so it is removed

Inclusion of the edge
< 3 , 5 > forms a cycle so it is removed

Cost of the spanning tree : 13

E:\g\ads>_
```

Dijkstra's algorithm for Single Source Shortest Path Problem

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While *sptSet* doesn't include all vertices

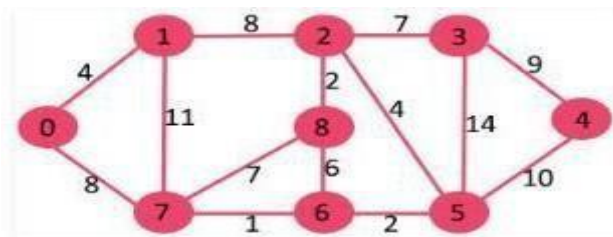
....a) Pick a vertex *u* which is not there in *sptSet* and has minimum distance value.

....b) Include *u* to *sptSet*.

....c) Update distance value of all adjacent vertices of *u*. To update the distance values, iterate through

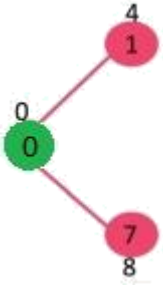
all adjacent vertices. For every adjacent vertex *v*, if sum of distance value of *u* (from source) and weight of edge *u-v*, is less than the distance value of *v*, then update the distance value of *v*.

Let us understand with the following example:

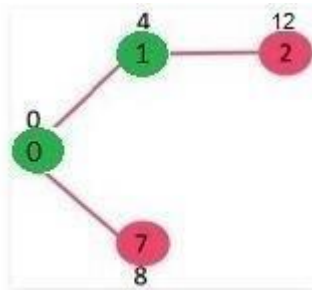


The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and

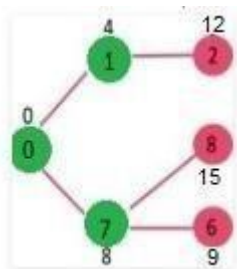
8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



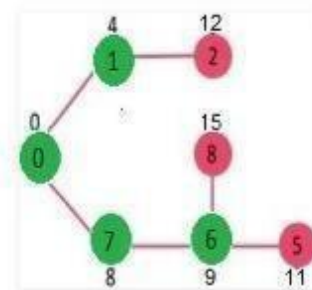
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



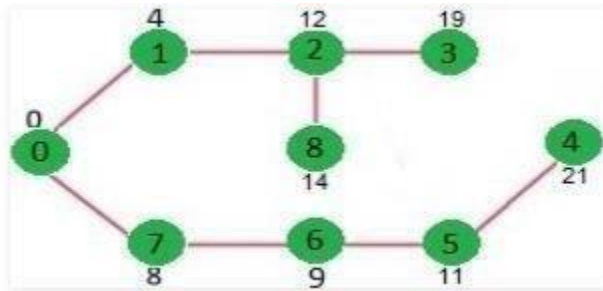
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



We use a boolean array *sptSet[]* to represent the set of vertices included in SPT. If a value *sptSet[v]* is true, then vertex *v* is included in SPT, otherwise not. Array *dist[]* is used to store shortest distance values of all vertices.

Notes:

- 1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim's implementation) and use it to show the shortest path from source to different vertices.
- 2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.
- 3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from the source to a single target, we can break the for the loop when the picked minimum distance vertex is equal to target (Step 3.a of the algorithm).
- 4) Time Complexity of the implementation is $O(V^2)$. If the input graph is represented using adjacency list, it can be reduced to $O(E \log V)$ with the help of binary heap.
- 5) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, Bellman–Ford algorithm can be used, we will soon be discussing it as a separate post.

java program to implement Dijkstra's algorithm for single shortest path

problem SOURCE CODE:

```
import java.util.PriorityQueue;
import java.util.List;
import java.util.ArrayList;
import
java.util.Collections;
class Vertex implements Comparable<Vertex>
```

{


```
public final String name;
public Edge[] adjacencies;
public double minDistance
=
Double.POSITIVE_INFINITY; public Vertex previous;
public Vertex(String argName) { name = argName;
} public String toString() { return name; } public int
compareTo(Vertex other)
{
    return Double.compare(minDistance, other.minDistance);
}
}

class Edge
{
    public final Vertex target;
    public final double weight;
    public Edge(Vertex argTarget, double argWeight)
    { target = argTarget; weight = argWeight; }
}

class Dijkstra1
{
    public static void computePaths(Vertex source)
    {
        source.minDistance = 0.;
        PriorityQueue<Vertex> vertexQueue =
        new
        PriorityQueue<Vertex>(); vertexQueue.add(source);

        while
        (!vertexQueue.isEmpty()) {
            Vertex u =
            vertexQueue.poll();

            // Visit each edge exiting
            u for (Edge e :
            u.adjacencies)
            {
                Vertex v = e.target;
                double weight =
                e.weight;
                double distanceThroughU = u.minDistance +
                weight; if (distanceThroughU <
                v.minDistance) {
                    vertexQueue.remove(v);
                    v.minDistance = distanceThroughU
                }
            }
        }
    }
}
```

```
        ; v.previous = u;  
        vertexQueue.add(  
            v);  
    }  
}
```

```

    }
}

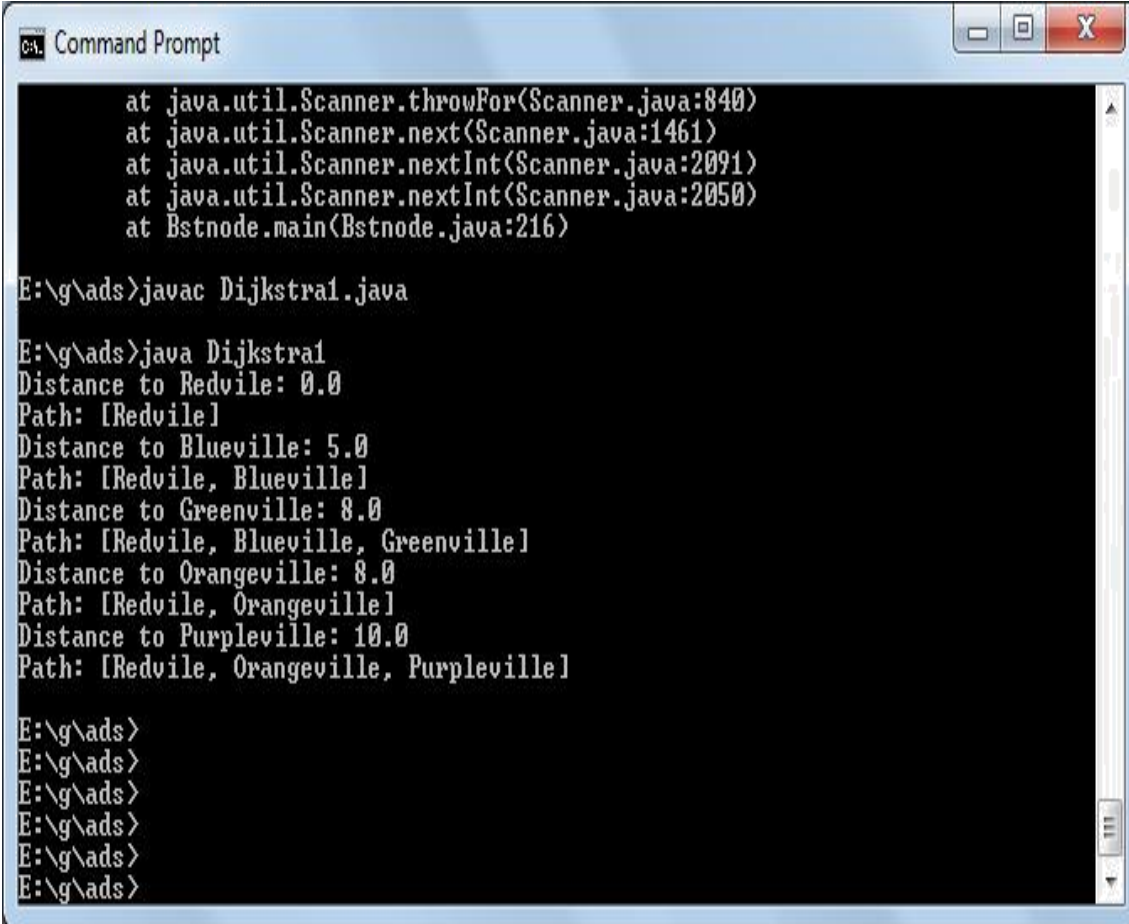
public static List<Vertex> getShortestPathTo(Vertex target)
{
    List<Vertex> path = new ArrayList<Vertex>();
    for (Vertex vertex = target; vertex != null; vertex = vertex.previous)
        path.add(vertex);
    Collections.reverse(path);
    return path;
}

public static void main(String[] args)
{
    Vertex v0 = new Vertex("Redville");
    Vertex v1 = new
    Vertex("Blueville"); Vertex v2 =
    new Vertex("Greenville"); Vertex v3
    = new Vertex("Orangeville"); Vertex
    v4 = new Vertex("Purpleville");

    v0.adjacencies = new Edge[]{ new Edge(v1, 5),
                                   new Edge(v2, 10),
                                   new Edge(v3, 8) };
    v1.adjacencies = new Edge[]{ new Edge(v0, 5),
                                   new Edge(v2, 3),
                                   new Edge(v4, 7) };
    v2.adjacencies = new Edge[]{ new Edge(v0,
    10), new Edge(v1, 3) };
    v3.adjacencies = new Edge[]{ new Edge(v0, 8),
                                   new Edge(v4, 2) };
    v4.adjacencies = new Edge[]{ new
    Edge(v1, 7),
                                   new Edge(v3, 2) };
    Vertex[] vertices = { v0, v1, v2, v3, v4 };
    computePaths(v0);
    for (Vertex v : vertices)
    {
        System.out.println("Distance to " + v + ": " +
        v.minDistance); List<Vertex> path =
        getShortestPathTo(v); System.out.println("Path: " +
        path);
    }
}
}

```

OUTPUT:



```
at java.util.Scanner.throwFor(Scanner.java:840)
at java.util.Scanner.next(Scanner.java:1461)
at java.util.Scanner.nextInt(Scanner.java:2091)
at java.util.Scanner.nextInt(Scanner.java:2050)
at Bstnode.main(Bstnode.java:216)

E:\g\ads>javac Dijkstra1.java

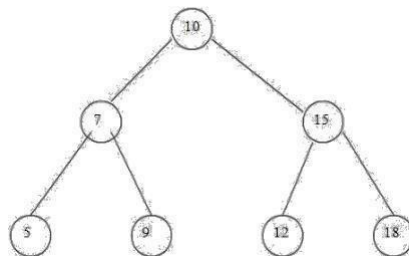
E:\g\ads>java Dijkstra1
Distance to Redvile: 0.0
Path: [Redvile]
Distance to Blueville: 5.0
Path: [Redvile, Blueville]
Distance to Greenville: 8.0
Path: [Redvile, Blueville, Greenville]
Distance to Orangeville: 8.0
Path: [Redvile, Orangeville]
Distance to Purpleville: 10.0
Path: [Redvile, Orangeville, Purpleville]

E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
```

UNIT V

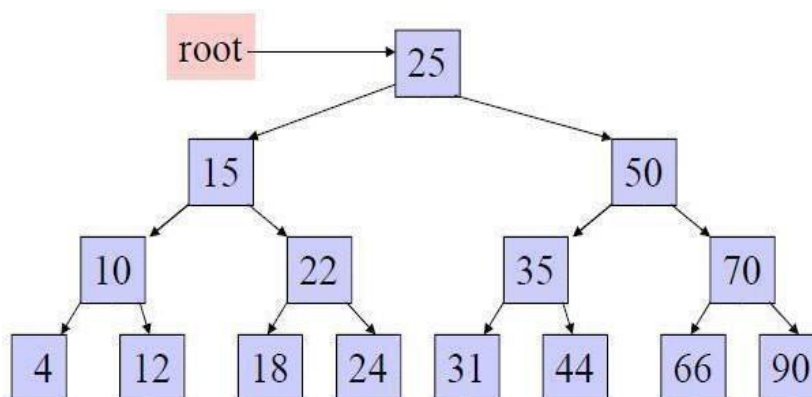
BINARY SEARCH TREE

In the simple binary tree the nodes are arranged in any fashion. Depending on user's desire the new nodes can be attached as a left or right child of any desired node. In such a case finding for any node is a long cut procedure, because in that case we have to search the entire tree. And thus the searching time complexity will get increased unnecessarily. So to make the searching algorithm faster in a binary tree we will go for building the binary search tree. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged. That means values at **left sub-tree** <



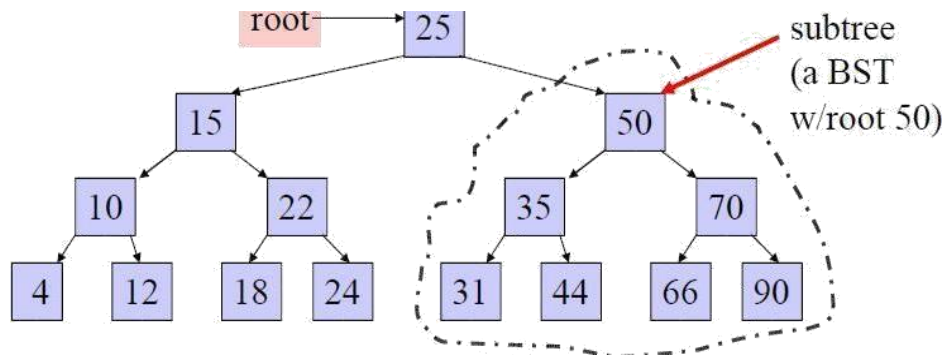
root node value < right sub-tree values.

1. Hierarchical data structure with a single reference to root node
2. Each node has at most two child nodes (a left and a right child)
3. Nodes are organized by the Binary Search property:
 1. Every node is ordered by some key data field(s)
 2. For every node in the tree, its key is greater than its left child's key and less than its right child's key



Some BST Terminology

1. The Root node is the top node in the hierarchy
2. A Child node has exactly one Parent node, a Parent node has at most two child nodes, Sibling nodes share the same Parent node (ex. node 22 is a child of node 15)
3. A Leaf node has no child nodes, an Interior node has at least one child node (ex. 18 is a leaf node)
4. Every node in the BST is a Subtree of the BST rooted at that node



Operations On Binary Search Tree:

The basic operations which can be performed on binary search tree are.

1. Insertion of a node in binary search tree.
2. Deletion of a node from binary search tree.
3. Searching for a particular node in binary search tree.

Insertion of a node in binary search tree.

While inserting any node in binary search tree, look for its appropriate position in the binary search tree. We start comparing this new node with each node of the tree. If the value of the node which is to be inserted is greater than the value of the current node we move on to the right sub-branch otherwise we move on to the left sub-branch. As soon as the appropriate position is found we attach this new node as left or right child appropriately.

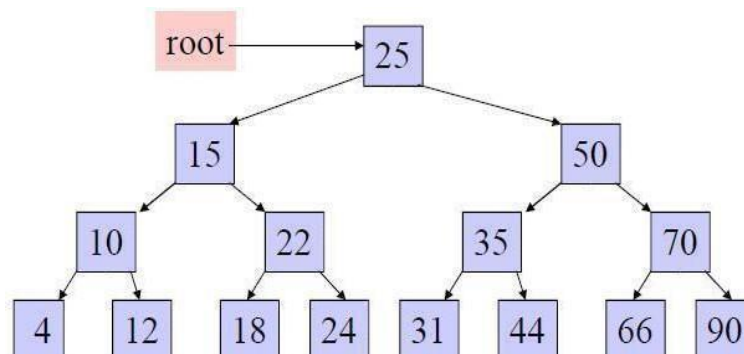
Steps to follow insertion into Binary Search Tree

Always insert new node as leaf
 node Start at root node as
 current node

If new node's key < current's key
 If current node has a left child, search
 left Else add new node as current's
 left child

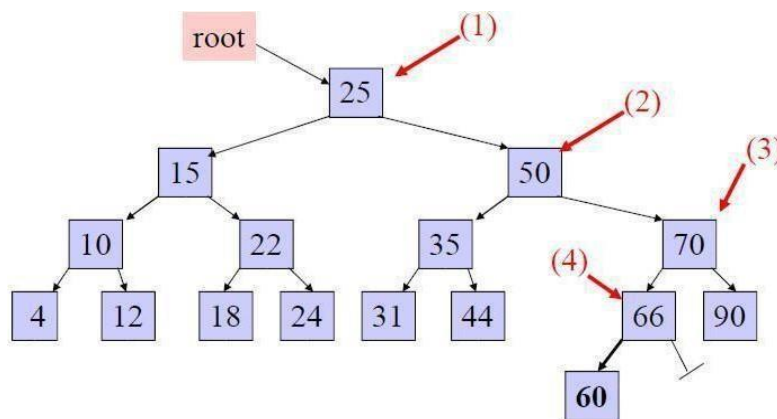
If new node's key > current's key
 If current node has a right child, search
 right Else add new node as current's
 right child

Before Insertion BST



Example: insert 60 in the tree:

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child



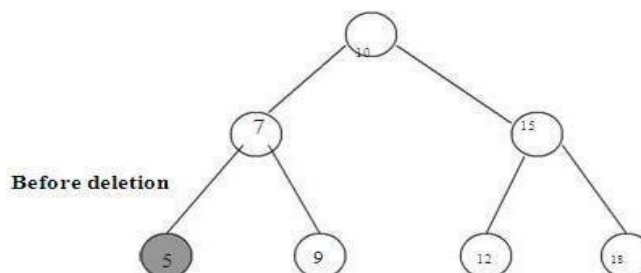
Deletion of a node from binary search tree.

For deletion of any node from binary search tree there are three which are possible.

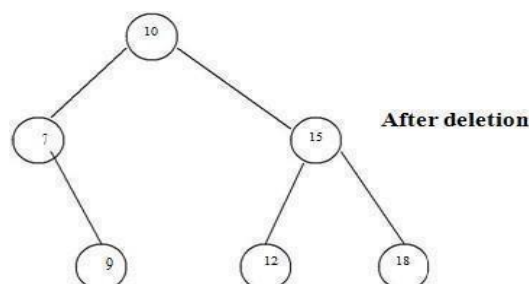
- i. Deletion of leaf node.
- ii. Deletion of a node having one child.
- iii. Deletion of a node having two children.

Deletion of leaf node.

This is the simplest deletion, in which we set the left or right pointer of parent node as NULL.

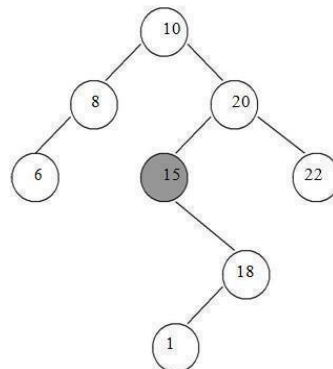


From the above fig, we want to delete the node having value 5 then we will set left pointer of its parent node as NULL. That is left pointer of node having value 7 is set to NULL.

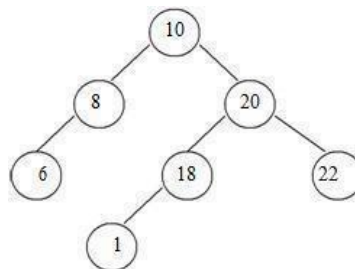


Deletion of a node having one child.

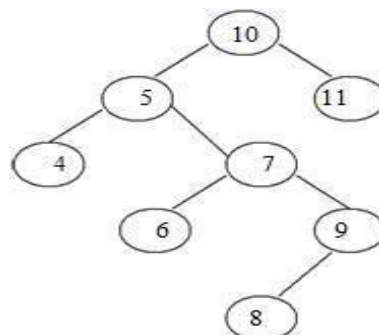
To explain this kind of deletion, consider a tree as given below.



If we want to delete the node 15, then we will simply copy node 18 at place of 16 and then set the node free

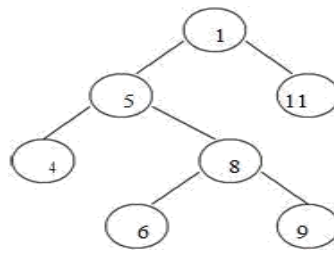
**Deletion of a node having two children.**

Consider a tree as given below.



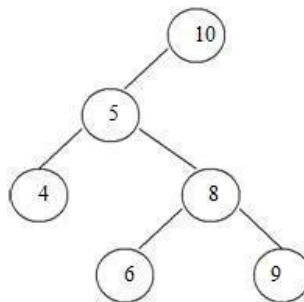
Let us consider that we want to delete node having value 7. We will then find out the inorder successor of node 7. We will then find out the inorder successor of node 7. The inorder successor will be simply copied at location of node 7.

That means copy 8 at the position where value of node is 7. Set left pointer of 9 as NULL. This completes the deletion procedure.



Searching for a node in binary search tree.

In searching, the node which we want to search is called a key node. The key node will be compared with each node starting from root node if value of key node is greater than current node then we search for it on right sub branch otherwise on left sub branch. If we reach to leaf node and still we do not get the value of key node then we declare "node is not present in the tree".



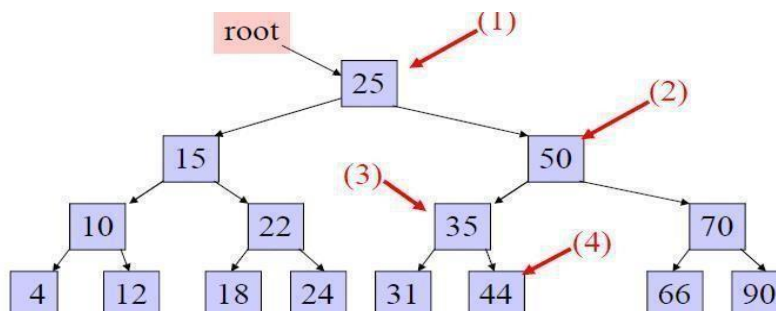
In the below tree, if we want to search for value 9. Then we will compare 9 with root node 10. As 9 is less than 10 we will search on left sub branch. Now compare 9 with 5, but 9 is greater than 5. So we will move on right sub tree. Now compare 9 with 8 but 9 is greater than 8 we will move on right sub branch. As the node we will get holds the value 9. Thus the desired node can be searched.

Another **example for search a node**

in BST Example: search for 45 in the tree

(key fields are show in node rather than in separate obj ref to by data field):

1. start at the root, 45 is greater than 25, search in right subtree
2. 45 is less than 50, search in 50's left subtree
3. 45 is greater than 35, search in 35's right subtree
4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST



SOURCE CODE FOR BST

Java program to perform the following operations:

- a) Construct a binary search tree of elements.**
- b) Search for a key element in the above binary search tree.**
- c) Delete an element from the above binary search tree.**

```
import
java.util.*; class
Bstnode
{
    Bstnode
    rc,lc;
    Bstnode
    root; int
    data;
    Bstnode()
    {
        data=0;
        rc=lc=null
        ;
    }
    Bstnode(int item)
    {
        data=item
        ;
        lc=rc=null
        ;
    }
    Bstnode[] search(int key)
    {
        Bstnode par ,ptr;
        Bstnode b[]=new
        Bstnode[2]; ptr=root;
        par=null;
        while(ptr!=null
        )
        {
            if(ptr.data==key)
            {
                b[0]=par;
                b[1]=ptr;
                return
                b;
            }
            else if(ptr.data<key)
            {
                ptr=ptr.rc;
            }
            else
            {
                ptr=ptr.lc;
            }
        }
    }
}
```

```
    par=ptr;  
    ptr=ptr.rc  
    ;  
}
```

```
    par=ptr;  
    ptr=ptr.lc  
    ;
```

```
        }
        b[0]=par;b[1]=ptr
    ; return b;
}
void insert(int item)
{
    Bstnode arr[]=new
    Bstnode[2]; Bstnode nn=new
    Bstnode(item);
    arr=search(item);
    if(root!=null)
    {
        Bstnode
        par=arr[0];
        Bstnode
        ptr=arr[1];
        if(ptr!=null)
            System.out.println("key already existed");
        else
        {
            if(par.data<item)
                par.rc=nn;
            else
                par.lc=nn;
        }
    }
    else
        root=nn;
}
void inorder(Bstnode ptr)
{
    if(ptr!=null)
    {
        inorder(ptr.lc);
        System.out.println(ptr.data
        ); inorder(ptr.rc);
    }
}
void preorder(Bstnode ptr)
{
    if(ptr!=null)
    {
        System.out.println(ptr.data
        ); inorder(ptr.lc);
        inorder(ptr.rc);
    }
}
void postorder(Bstnode ptr)
{
    if(ptr!=null)
```

```
        {
            inorder(ptr.lc);
            inorder(ptr.rc);
            System.out.println(ptr.data
        );
        }
    }
}
int deleteleaf(Bstnode par,Bstnode ptr)
{
    if(par!=null)
    {
        if(par.lc==ptr)
            par.lc=null;
        else
            par.rc=null;
    }
    else
    {
        root=null;
        ; return ptr.data;
    }
}
int delete1childnode(Bstnode par,Bstnode ptr)
{
    if(par!=null)
    {
        if(par.lc==ptr)
        {
            if(ptr.lc==null)
                par.lc=ptr.rc;
            else
                par.lc=ptr.lc;
        }
        else if(par.rc==ptr)
        {
            if(ptr.lc==null)
                par.rc=ptr.rc;
            else
                par.rc=ptr.lc;
        }
    }
    else
    {
        if(ptr.rc!=null)
            root=ptr.rc;
        else
            root=ptr.lc;
    }
    return ptr.data;
}
int delete2childnode(Bstnode par,Bstnode ptr)
```

```

{
    Bstnode
    ptr1=ptr.rc;
    Bstnode par1=null;
    while(ptr1.lc!=null)
    {
        par1=ptr1;
        ptr1=ptr1.lc;
    }
    if(par1!=null
    )
    {
        if(ptr1.r
        lc!=null)
        par1.lc=ptr1.r
        c;

        els
        e        par1.lc=nul
        l; ptr1.lc=ptr.lc;
        ptr1.rc=ptr.rc;

    }
    else // if par1=null
        ptr1.lc =
        ptr.lc; if(par!=null)
    {
        if(par.lc==ptr)
            par.lc=ptr1;
        els
        e        par.rc=ptr1;
    }
    els
    e
        root=ptr1;
    return ptr.data;
}
int deletenode(int item)
{
    Bstnode
    ptr=root,par=null;
    boolean flag=false;
    int k;
    while(ptr!=null&&flag==fals
    e)
    {
        if(item<ptr.data)
        {
            par=ptr;
            ptr=ptr.lc

```

```
        ;  
    }  
    else if(item>ptr.data)  
    {  
        par=ptr  
        ; ptr=ptr.rc;  
    }
```

```

        else
        {
            ptr.data=item
            ; flag=true;
        }
    }
    if(flag==false)
    {
        System.out.println("item not found hence can not
        delete"); return -1;    }
        if(ptr.lc==null&&ptr.rc==nul
        l)
            k=deleteleaf(par,ptr);
        else if(ptr.lc!=null&&ptr.rc!=null)
            k=delete2childnode(par,pt
            r);
        else
            k=delete1childnode(par,ptr);
        return k;
    }
    public static void main(String saichandra[])
    {
        Bstnode b=new Bstnode();
        Scanner s=new Scanner
        (System.in); int ch;
        do
        {
            System.out.println("1.insert");
            System.out.println("2.delete");
            System.out.println("3.search");
            System.out.println("4.inorder");
            System.out.println("5.preorder");
            System.out.println("6.postorder");
            System.out.print("enter ur
            choice:"); ch=s.nextInt();
            switch(ch)
            {
                case 1: System.out.print("enter
                element:"); int n=s.nextInt();
                b.insert(n)
                ; break;
                case 2: if(b.root!=null)
                {
                    System.out.print("enter element:");
                    int n1=s.nextInt();
                    int
                    res=b.deletenode(n1);
                    if(res!=-1)
                    System.out.println("deleted element is:"+res);

```


}

```

        else
            System.out.println("no elements in
            tree"); break;
    case 3:if(b.root!=null)
        {
            System.out.println("enter search
            element"); int key=s.nextInt();
            Bstnode search1[]=new
            Bstnode[2];
            search1=b.search(key);
            if(search1[1]!=null)
                System.out.println("key is found");
            else
                System.out.println("key not found");
                if(search1[0]!=null)
                    {
                        if(search1[1]!=null)
                            System.out.println("parent of the searched element is:"+search1[0].data);
                    }
                else
                    System.out.println("key is root no parent exist");
                }
            else
                System.out.println("no elements in
                tree"); break;
    case 4:if(b.root!=null)
        b.inorder(b.root)
        ; else
            System.out.println("no elements in
            tree"); break;
    case 5:if(b.root!=null)
        b.preorder(b.root)
        ; else
            System.out.println("no elements in
            tree"); break;
    case 6:if(b.root!=null)
        b.postorder(b.root)
        ; else
            System.out.println("no elements in
            tree"); break;
    }
    }while(ch!=0);
}
}

```

OUTPUT:

```

Command Prompt - java Bstnode
6.postorder
enter ur choice:1
enter element:9
1.insert
2.delete
3.search
4.inorder
5.preorder
6.postorder
enter ur choice:1
enter element:6
1.insert
2.delete
3.search
4.inorder
5.preorder
6.postorder
enter ur choice:4
3
6
9
1.insert
2.delete
3.search
4.inorder
5.preorder
6.postorder
enter ur choice:3
enter search element
9
key is found
parent of the searched element is:3
1.insert
2.delete
3.search
4.inorder
5.preorder
6.postorder
enter ur choice:2
enter element:9
deleted element is:9
1.insert
2.delete
3.search
4.inorder
5.preorder
6.postorder
enter ur choice:2
enter element:5
item not found hence can not delete
1.insert
2.delete
3.search
4.inorder
5.preorder
6.postorder
enter ur choice:

```

Balanced search trees**1 AVL trees-Definition and examples only****2 Red Black trees – Definition and examples only****3 B-Trees-definition, insertion and searching operations****AVL trees-Definition and examples only****AVL TREES**

Adelson Velski and Landis in 1962 introduced binary tree structure that is balanced with respect to height of sub trees. The tree can be made balanced and because of this retrieval of any node can be done in $O(\log n)$ times, where n is total number of nodes. From the name of these scientists the tree is called AVL tree.

Definition:

An empty tree is height balanced if T is a non empty binary tree with TL and TR as its left and right sub trees. The T is height balanced if and only if

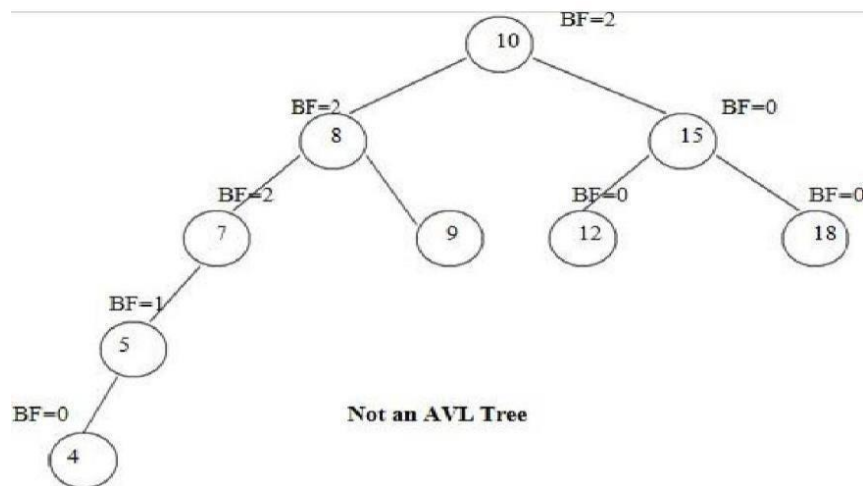
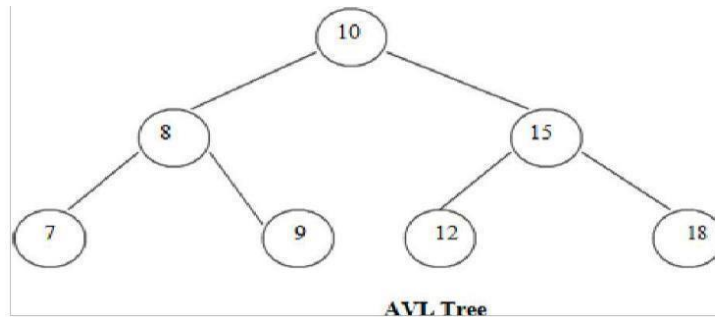
- i. TL and TR are height balanced.

- ii. $hL - hR \leq 1$ where hL and hR are heights of TL and TR.
The idea of balancing a tree is obtained by calculating the balance factor of a tree.

Definition of Balance Factor:

The balance factor $BF(T)$ of a node in binary tree is defined to be $hL - hR$ where hL and hR are heights of left and right sub trees of T .

For any node in AVL tree the balance factor i.e. $BF(T)$ is **-1, 0 or +1**.



Height of AVL Tree:

Theorem: The height of AVL tree with n elements (nodes) is $O(\log n)$.

Proof: Let an AVL tree with n nodes in it. N_h be the minimum number of nodes in an AVL tree of height h .

In worst case, one sub tree may have height $h-1$ and other sub tree may have height $h-2$. And both these sub trees are AVL trees. Since for every node in AVL tree the height of left and right sub trees differ by at most 1.

Hence

$$N_h = N_{h-1} + N_{h-2} + 1$$

Where N_h denotes the minimum number of nodes in an AVL tree of height h . $N_0=0$ $N_1=1$

We can also write it

$$N > N_h = N_{h-1} + N_{h-2} + 1$$

$$> 2N_{h-2} + 1$$

$$> 2(2N_{h-3} + 1) + 1$$

$$> 4N_{h-3} + 3$$

.

.

$$> 2^i N_{h-2i} + 2^i - 1$$

If value of h is even, let $i = h/2$

Then equation

becomes

$$N > 2^{h/2} N_2$$

$$= N > 2^{(h-1)/2} \times 4 \quad (N_2 = 4)$$

$$= O(\log N)$$

If value of h is odd, let $i = (h-1)/2$ then equation

$$\text{becomes } N > 2^{(h-1)/2} N_1 \quad N > 2^{(h-1)/2} \times 1 \quad H = O(\log$$

$N)$

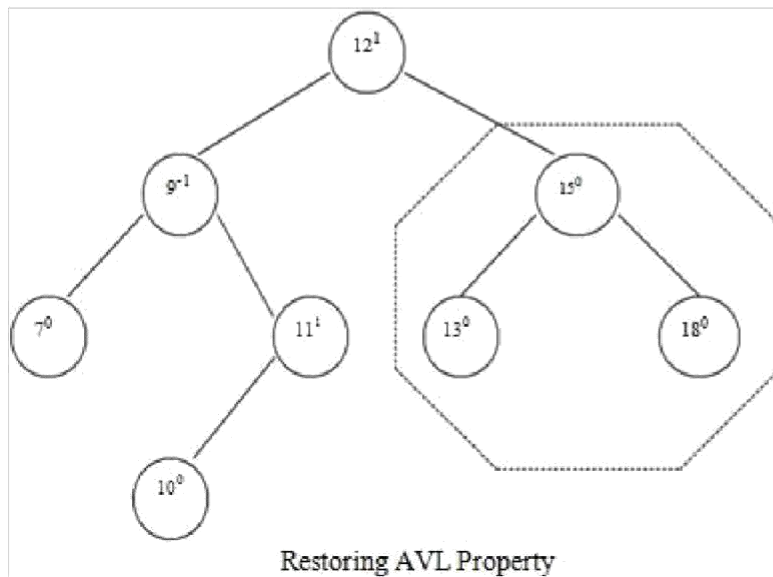
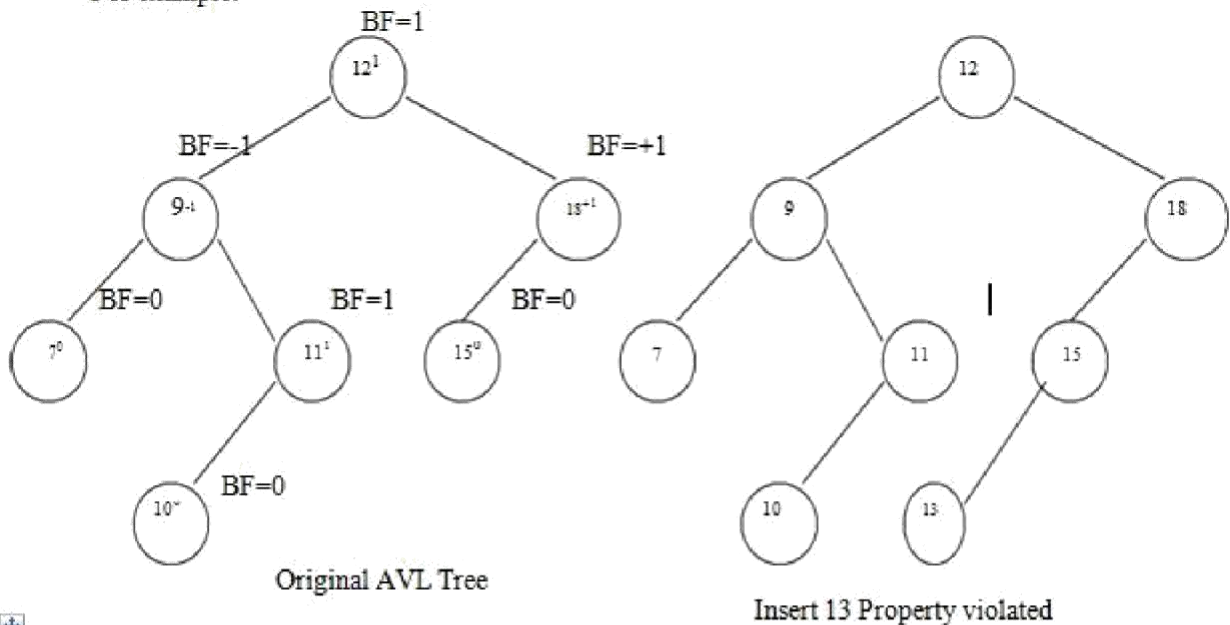
This proves that height of AVL tree is always $O(\log N)$. Hence search, insertion and deletion can be carried out in logarithmic time.

Representation of AVL Tree

1. The AVL tree follows the property of binary search tree. In fact AVL trees are basically binary search trees with balance factors as -1, 0, or +1.
2. After insertion of any node in an AVL tree if the balance factor of any node becomes other than -1, 0, or +1 then it is said that AVL property is violated. Then we have to restore the destroyed balance condition. The balance factor is denoted at right top

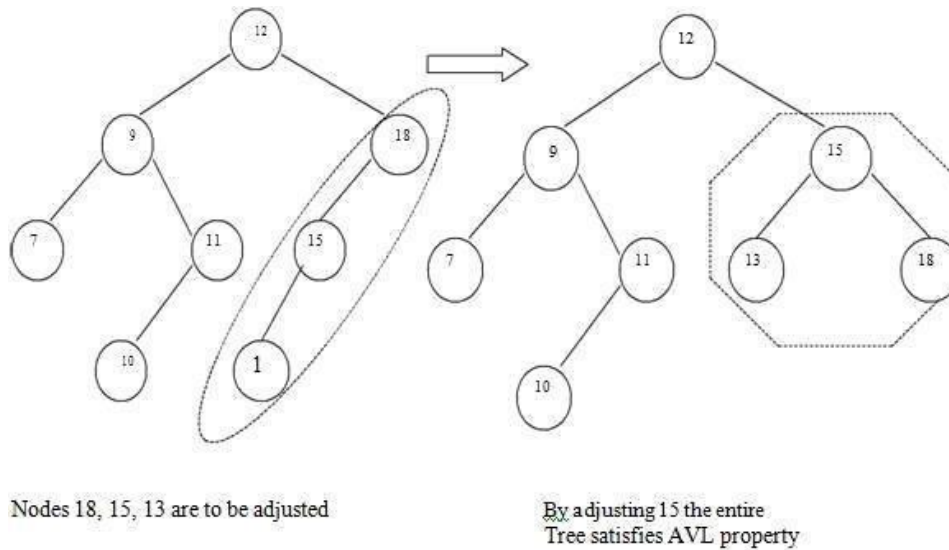
corner inside the node.

For example:



- └ After insertion of a new node if balance condition gets destroyed, then the nodes on that path(new node insertion point to root) needs to be readjusted. That means only the affected sub tree is to be rebalanced.
- └ The rebalancing should be such that entire tree should satisfy AVL property.

In above given example-



Insertion of a node.

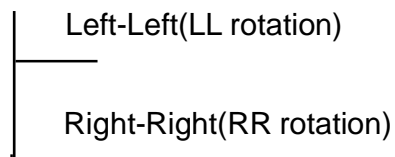
There are four different cases when rebalancing is required after insertion of new node.

1. An insertion of new node into left sub tree of left child. (LL).
2. An insertion of new node into right sub tree of left child. (LR).
3. An insertion of new node into left sub tree of right child.(RL).
4. An insertion of new node into right sub tree of rightchild.(RR).

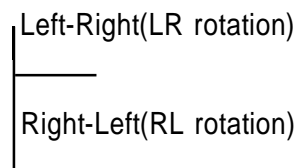
Some modifications done on AVL tree in order to rebalance it is called rotations of AVL tree

There are two types of rotations:

Single rotation



Double rotation



Insertion Algorithm:

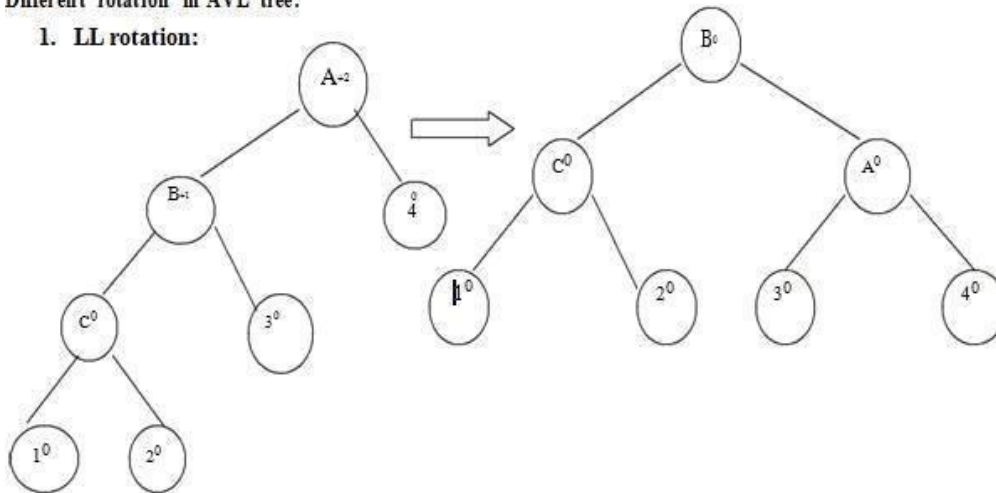
1. Insert a new node as new leaf just as an ordinary binary search tree.

2. Now trace the path from insertion point (new node inserted as leaf) towards root. For each node 'n' encountered, check if heights of left (n) and right (n) differ by at most 1.
 - a. If yes, move towards parent (n).
 - b. Otherwise restructure by doing either a single rotation or a double rotation.

Thus once we perform a rotation at node 'n' we do not require to perform any rotation at any ancestor on 'n'.

Different rotation in AVL tree:

1. LL rotation:

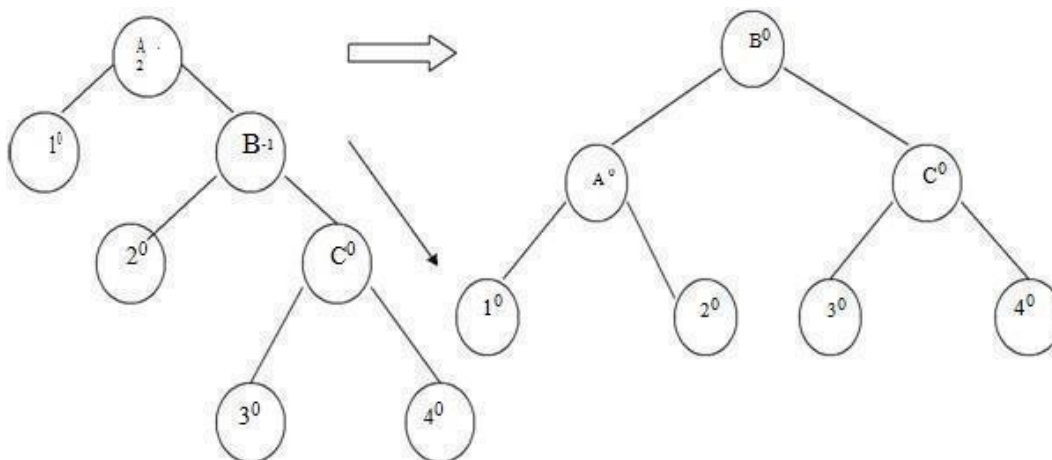


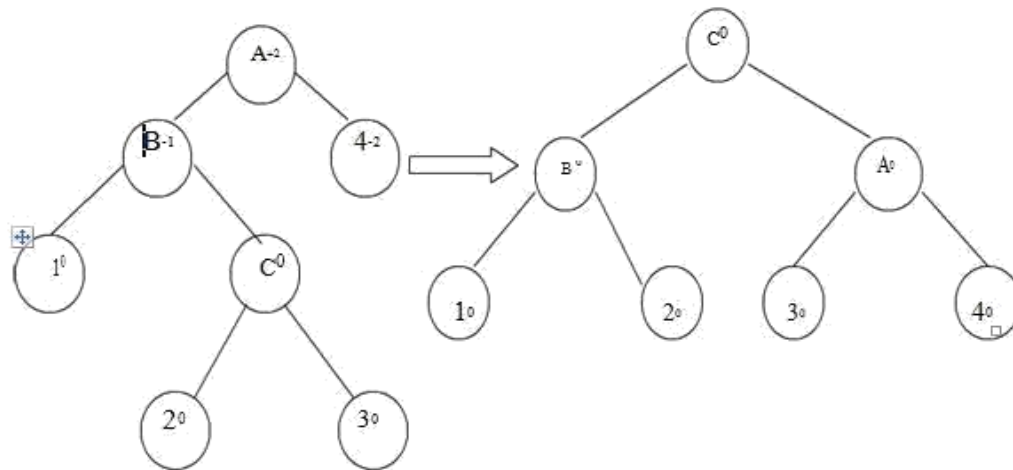
When node '1' gets inserted as a left child of node 'C' then AVL property gets destroyed i.e. node A has balance factor +2.

The LL rotation has to be applied to rebalance the nodes.

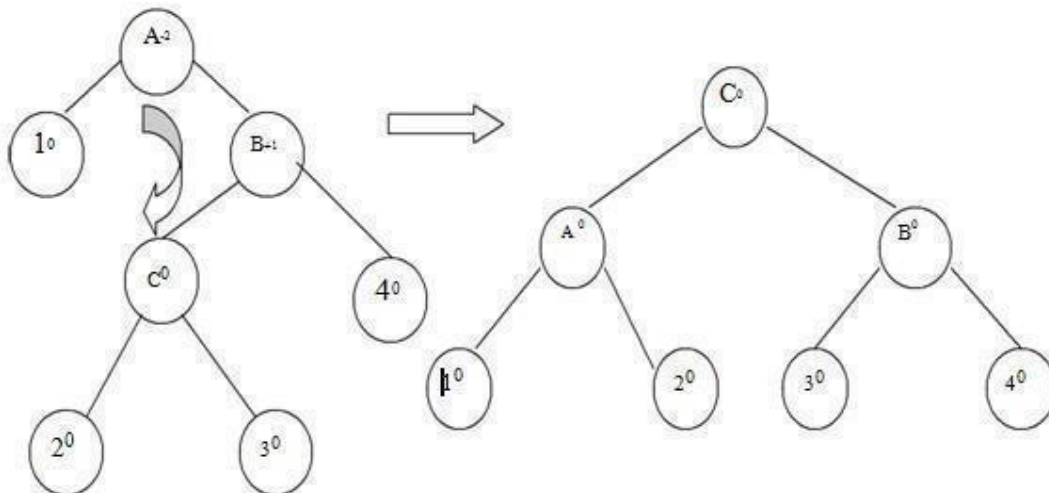
2. RR rotation:

When node '4' gets attached as right child of node 'C' then node 'A' gets unbalanced. The rotation which needs to be applied is RR rotation as shown in fig.



3. LR rotation:

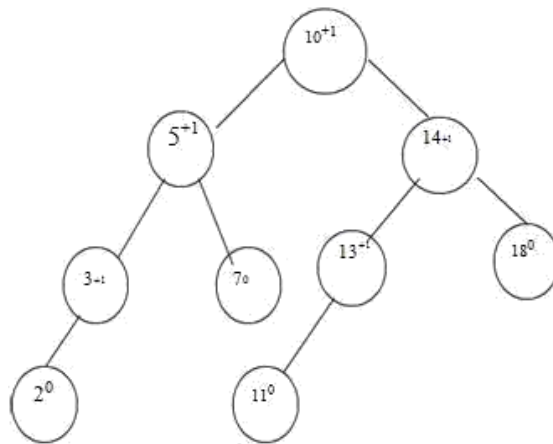
When node '3' is attached as a right child of node 'C' then unbalancing occurs because of LR. Hence LR rotation needs to be applied.

4. RL rotation

When node '2' is attached as a left child of node 'C' then node 'A' gets unbalanced as its balance factor becomes -2. Then RL rotation needs to be applied to rebalance the AVL tree.

Example:

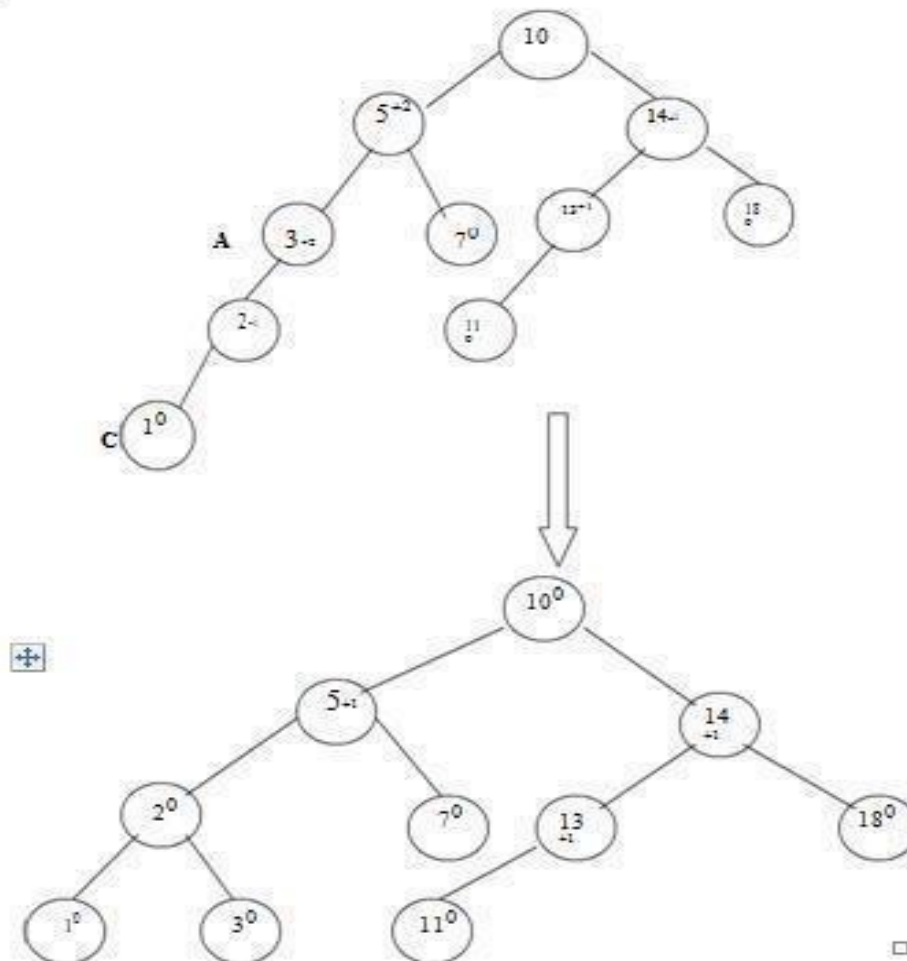
Insert 1, 25, 28, 12 in the following AVL tree.



Insert 1

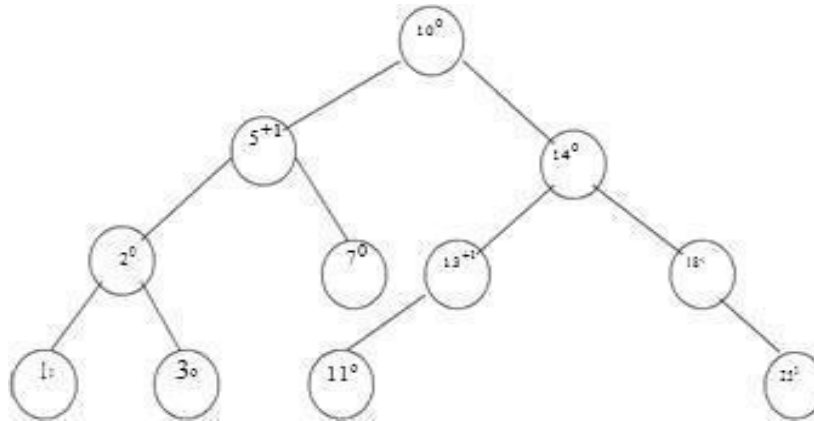
To insert node '1' we have to attach it as a left child of '2'. This will unbalance the tree as follows.

We will apply LL rotation to preserve AVL property of it.

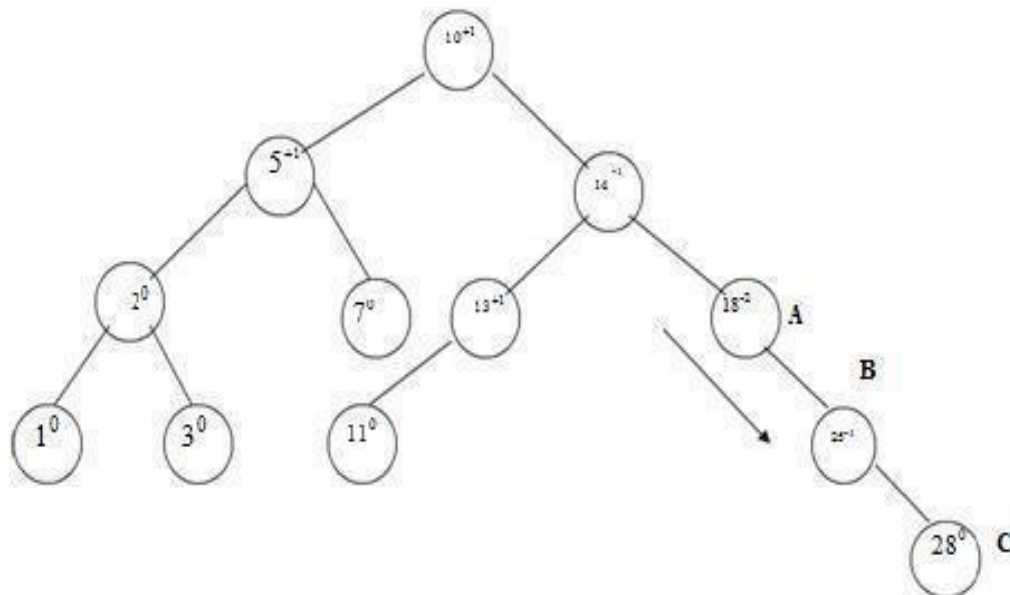


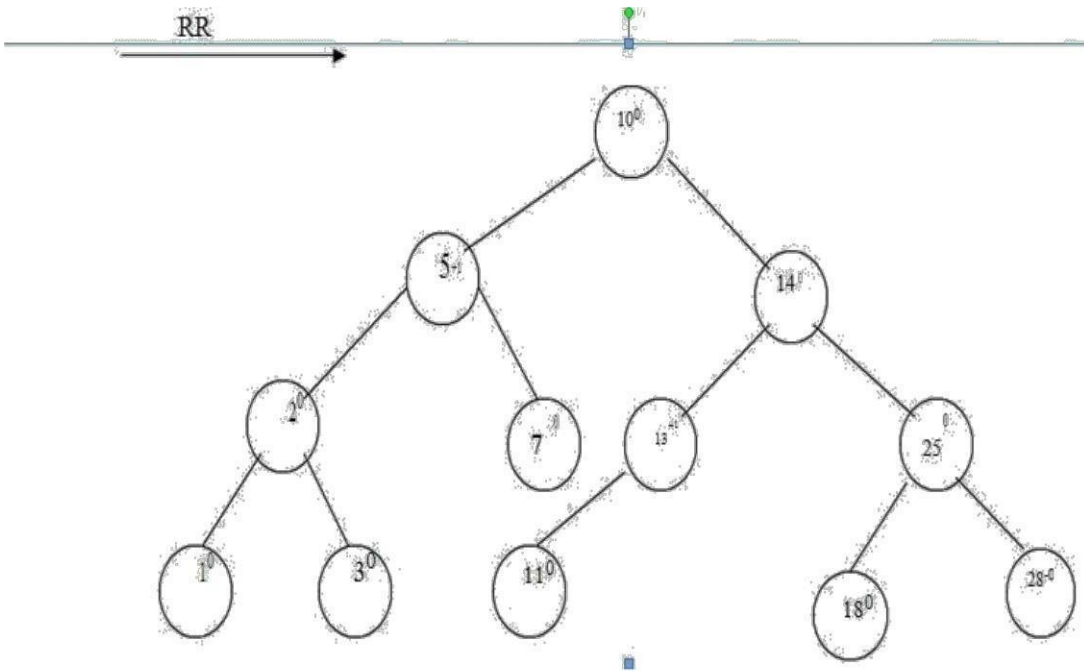
Insert 25

We will attach 25 as a right child of 18. No balancing is required as entire tree preserves the AVL property

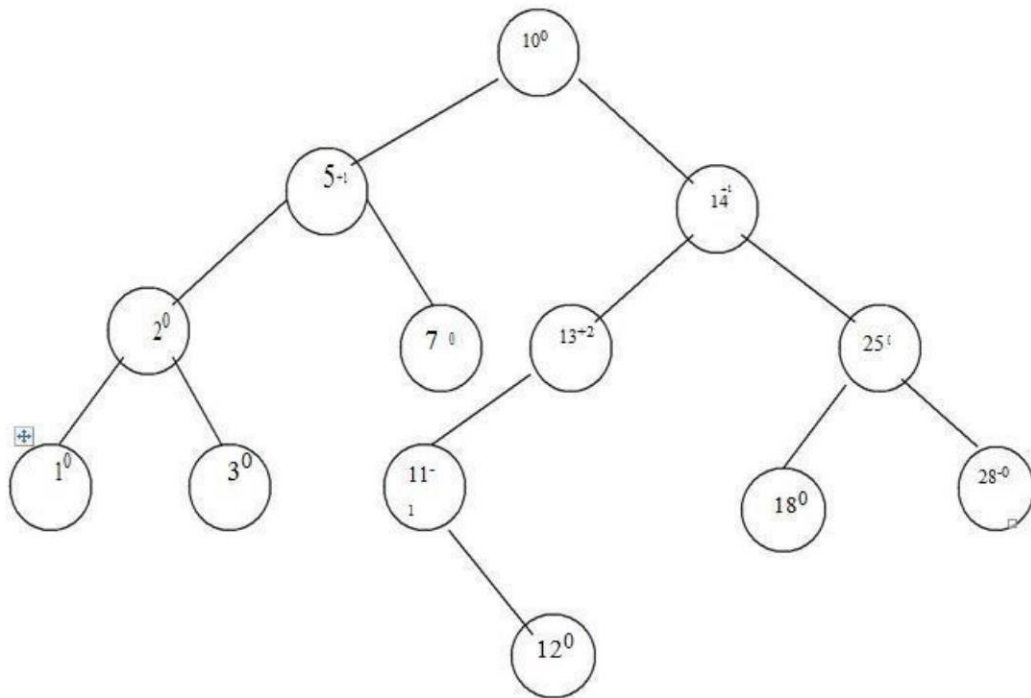
**Insert 28**

The node '28' is attached as a right child of 25. RR rotation is required to rebalance.

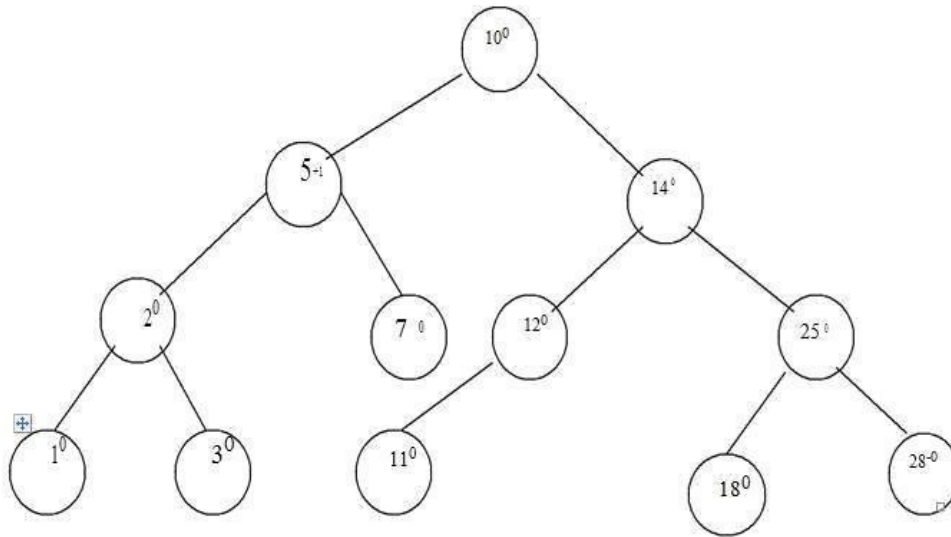




I



To rebalance the tree we have to apply LR rotation.



Thus by applying various rotations depending upon direction of insertion of new node the AVL tree can be restructured.

Deletion:

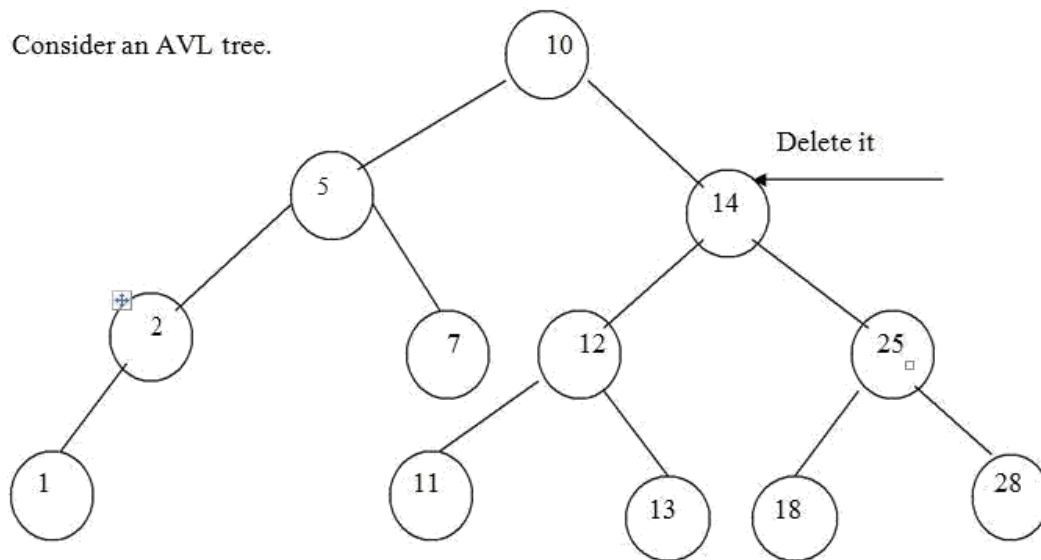
Even after deletion of any particular node from AVL tree, the tree has to be restructured in order to preserve AVL property. And thereby various rotations need to be applied.

Algorithm for deletion:

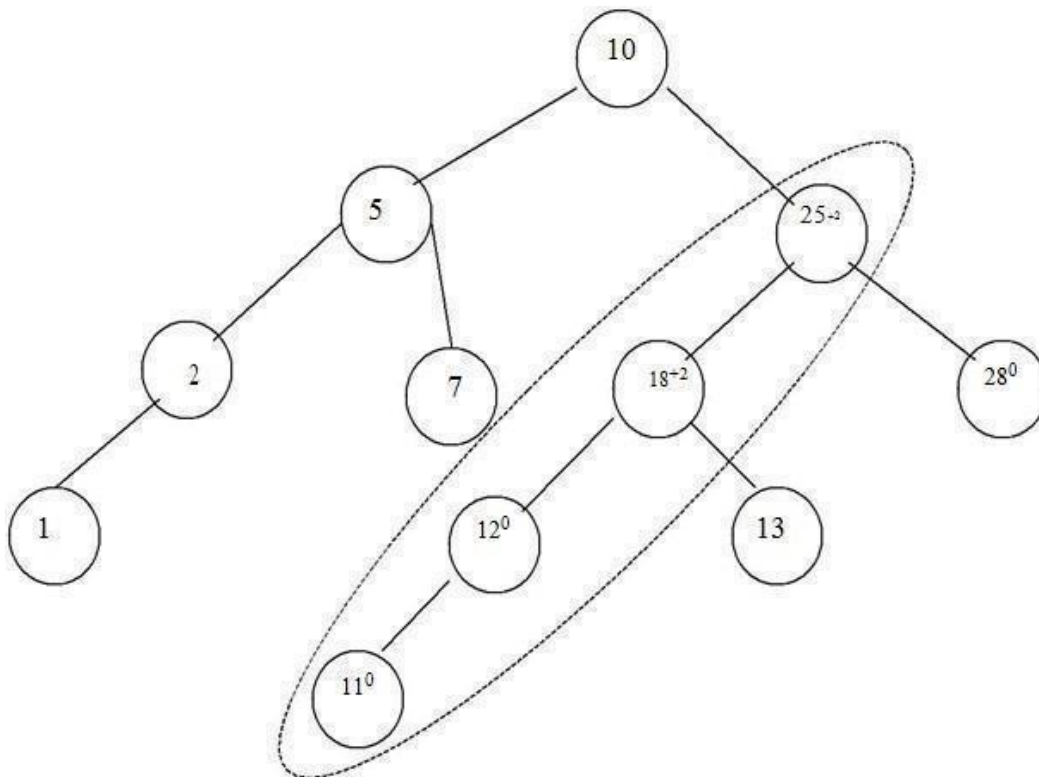
The deletion algorithm is more complex than insertion algorithm.

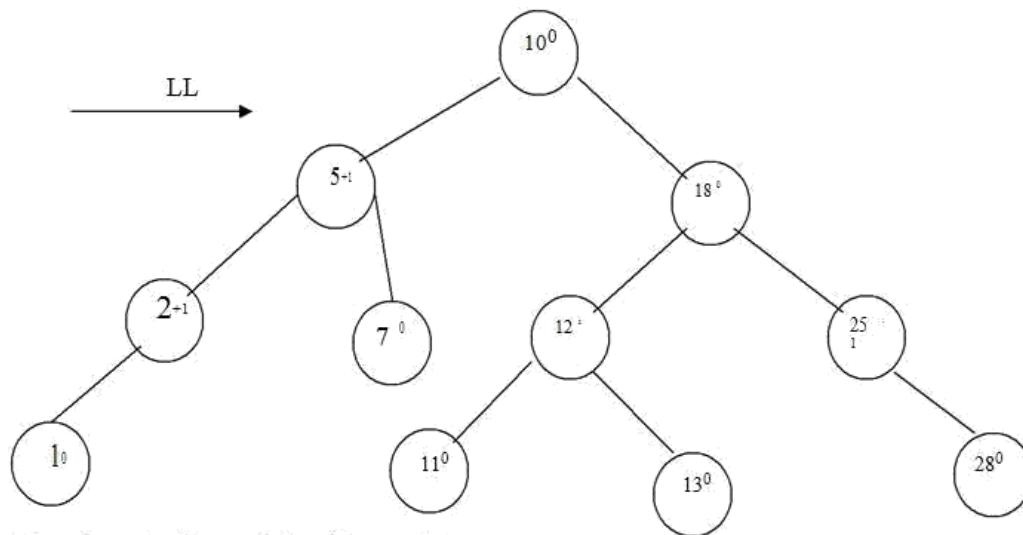
1. Search the node which is to be deleted.
2. a) If the node to be deleted is a leaf node then simply make it NULL to remove.
b) The node to be deleted is not a leaf node i.e. node may have one or two children, then the node must be swapped with its inorder successor. Once the node is swapped, we can remove this node.
3. Now we have to traverse back up the path towards root, checking the balance factor of every node along the path. If we encounter unbalancing in some sub tree then balance that sub tree using appropriate single or double rotations.
4. The deletion algorithm takes **$O(\log n)$** time to delete any node.

Consider an AVL tree.



The tree becomes





Thus the node 14 gets deleted from AVL tree.

Searching:

The searching of a node in an AVL tree is very simple. As AVL tree is basically binary search tree, the algorithm used for searching a node from binary search tree is the same one is used to search a node from AVL tree.

The searching of a node from AVL tree takes **$O(\log n)$** time.

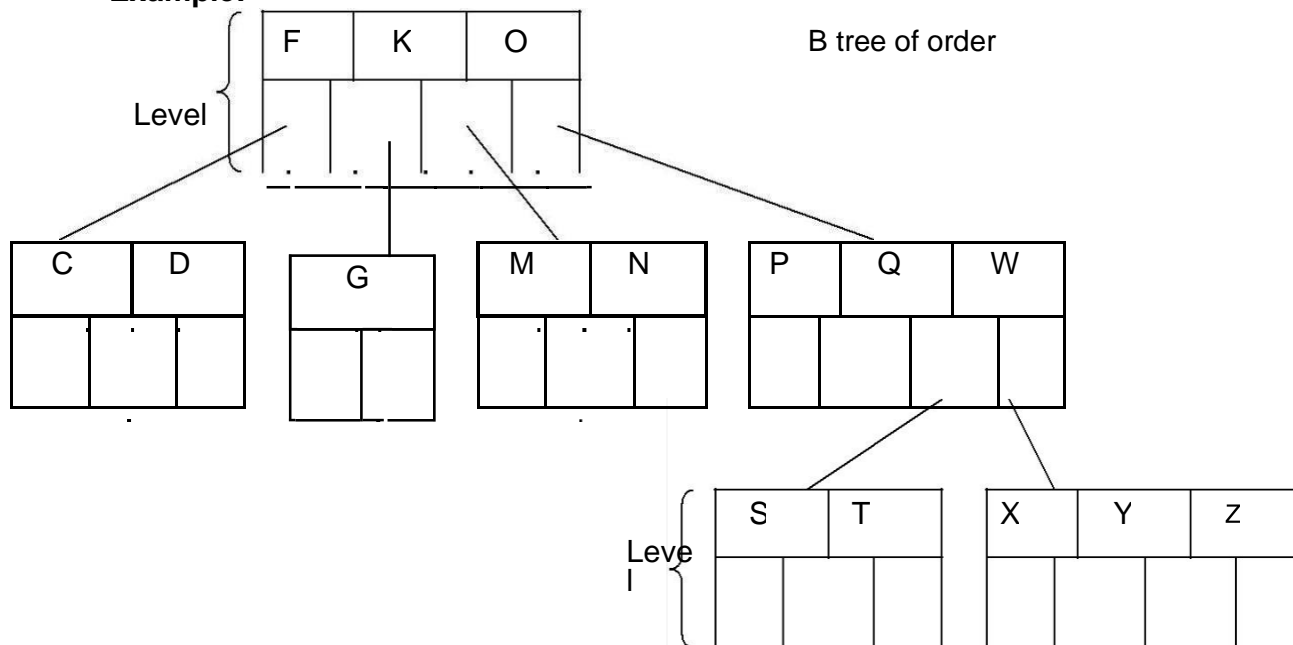
BTREES

- Multi-way trees are tree data structures with more than two branches at a node. The data structures of m-way search trees, B trees and Tries belong to this category of tree structures.
- AVL search trees are height balanced versions of binary search trees, provide efficient retrievals and storage operations. The complexity of insert, delete and search operations on AVL search trees is $O(\log n)$.
- Applications such as File indexing where the entries in an index may be very large, maintaining the index as m-way search trees provides a better option than AVL search trees which are but only balanced binary search trees.
- While binary search trees are two-way search trees, m-way search trees are extended binary search trees and hence provide efficient retrievals.
- B trees are height balanced versions of m-way search trees and they do not recommend representation of keys with varying sizes. Tries are tree based data structures that support keys with varying sizes.

Definition:

A B tree of order m is an m -way search tree and hence may be empty. If non empty, then the following properties are satisfied on its extended tree representation

1. The root node must have at least two child nodes and at most m child nodes.
2. All internal nodes other than the root node must have at least $\lfloor m/2 \rfloor$ non empty child nodes and at most m non empty child nodes.
3. The number of keys in each internal node is one less than its number of child nodes and these keys partition the keys of the tree into sub trees.
4. All external nodes are at the same level.

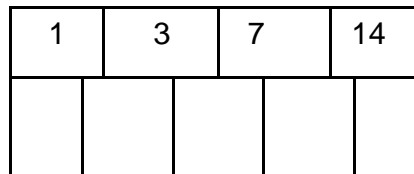
Example:

INSERTION

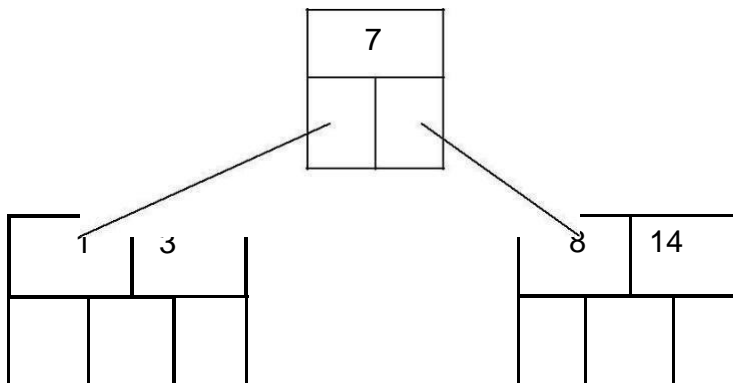
For example construct a B-tree of order 5 using following numbers. 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19

The order 5 means at the most 4 keys are allowed. The internal node should have at least 3 non empty children and each leaf node must contain at least 2 keys.

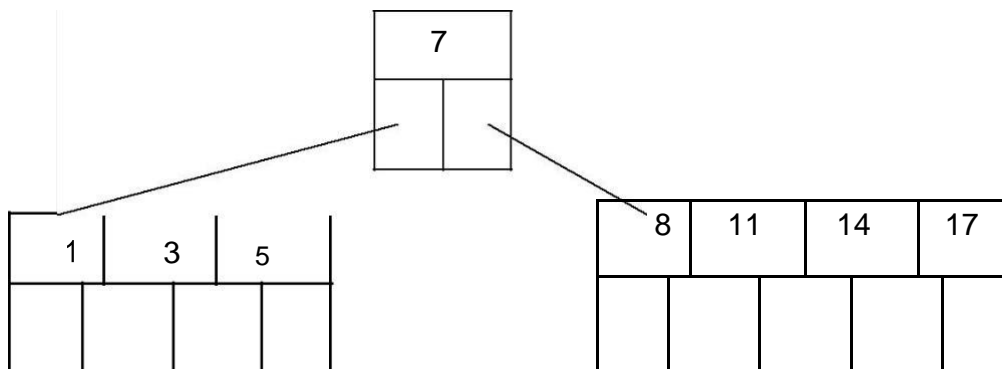
Step 1: Insert 3, 14, 7, 1



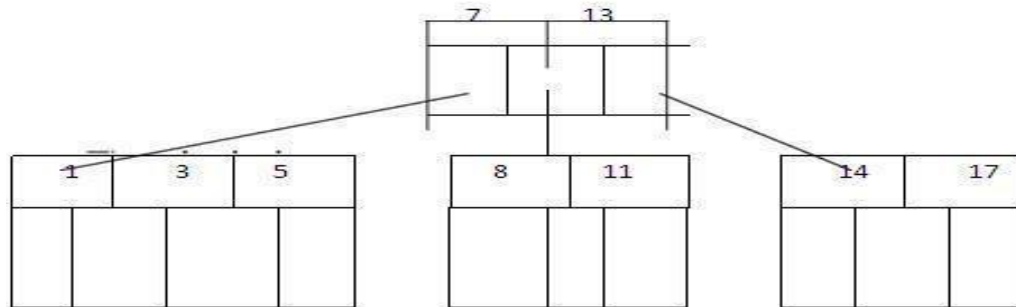
Step 2: Insert 8, Since the node is full split the node at medium 1, 3, 7, 8, 14



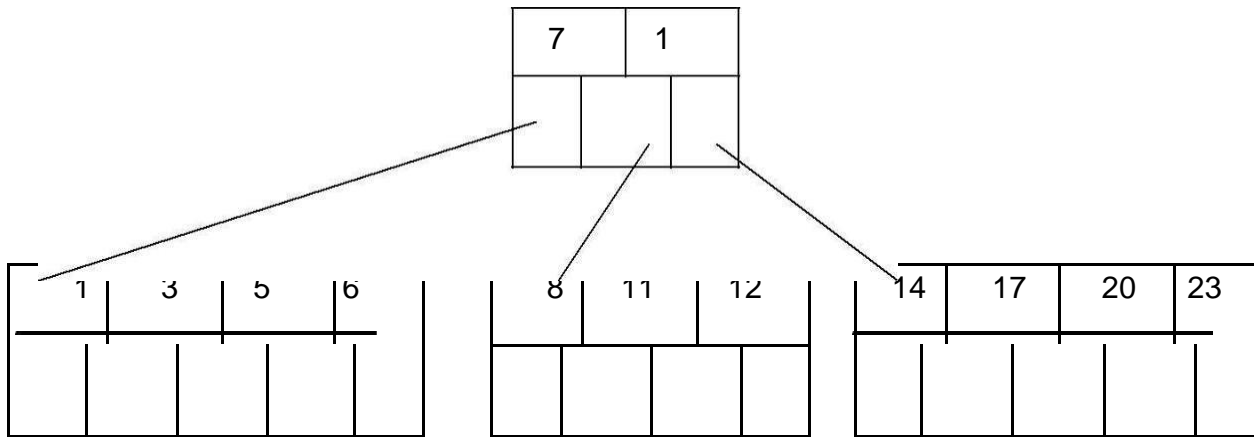
Step 3: Insert 5, 11, 17 which can be easily inserted in a B-tree.



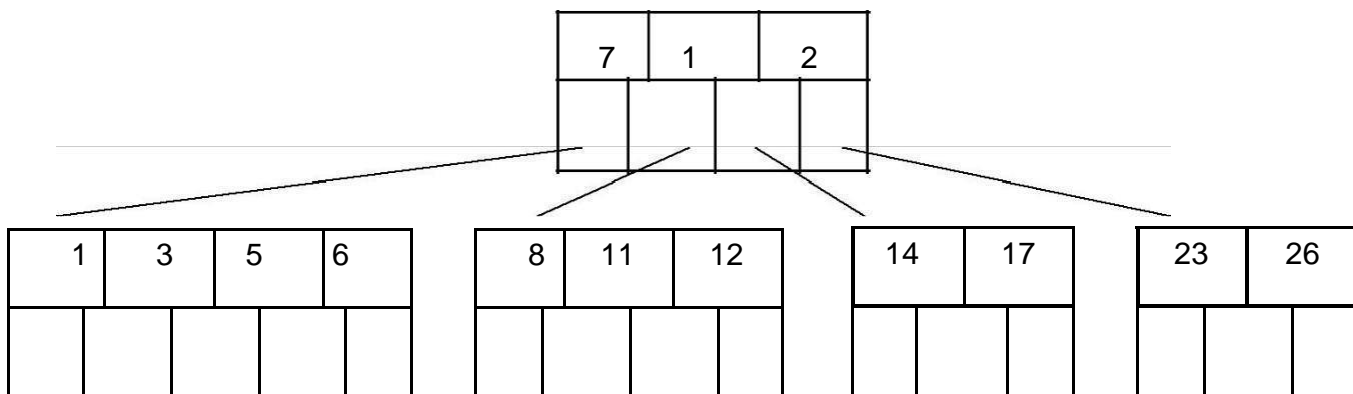
Step 4: Now insert 13. But if we insert 13 then the leaf node will have 5 keys which is not allowed. Hence 8, 11, 13, 14, 17 is split and medium node 13 is moved up.



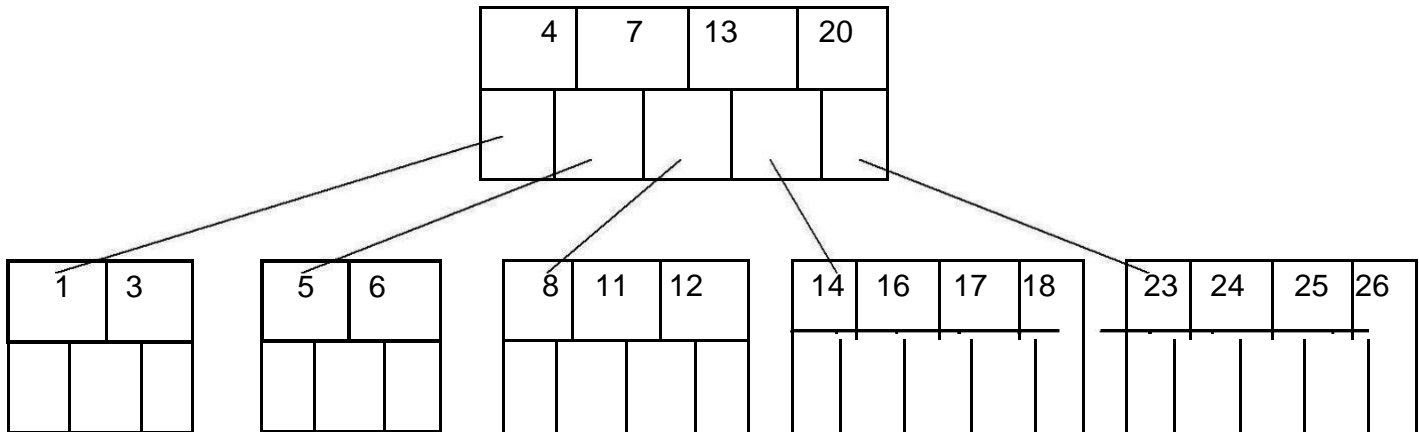
Step 5: Now insert 6, 23, 12, 20 without any split.



Step 6: The 26 is inserted to the right most leaf node. Hence 14, 17, 20, 23, 26 the node is split and 20 will be moved up.

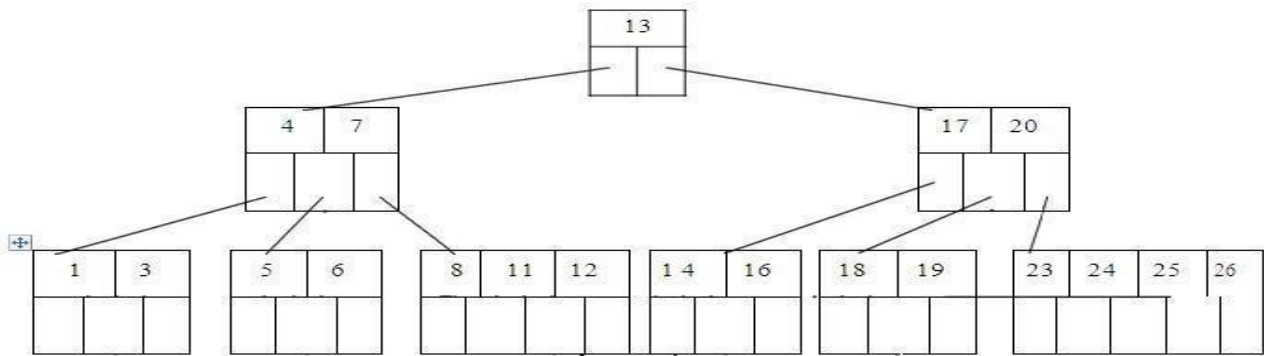


Step 7: Insertion of node 4 causes left most node to split. The 1, 3, 4, 5, 6 causes key 4 to move up. Then insert 16, 18, 24, 25.



Step 8: Finally insert 19. Then 4, 7, 13, 19, 20 needs to be split. The median 13 will be moved up to from a root node.

The tree then will be -

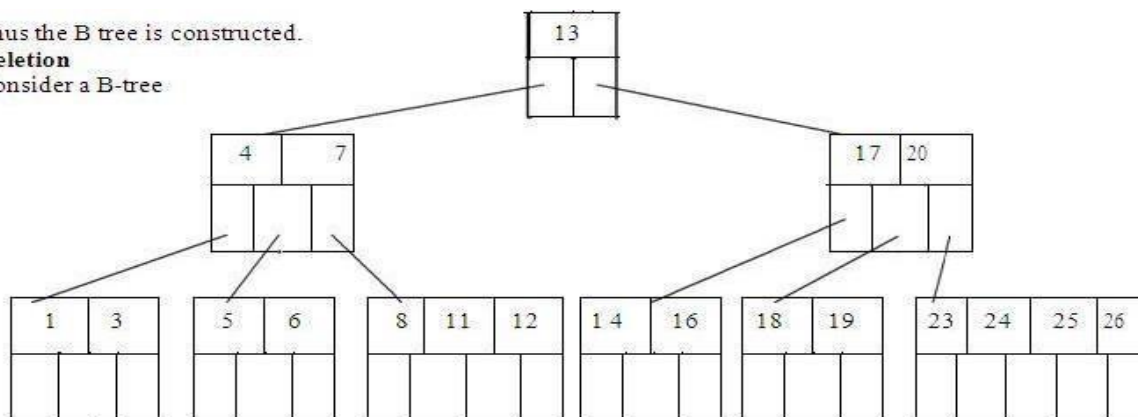


Deletion:

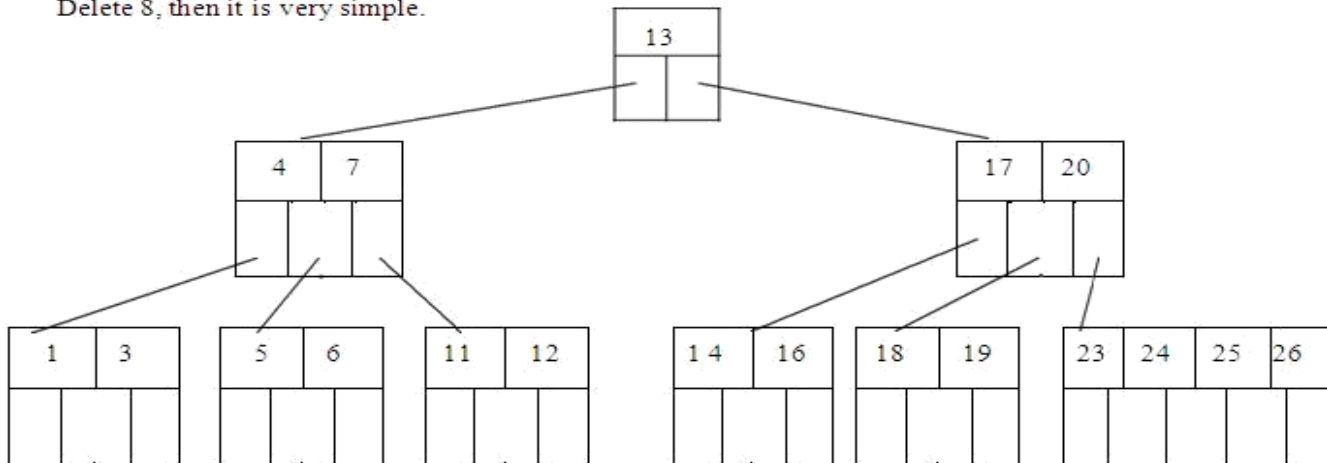
Thus the B tree is constructed.

Deletion

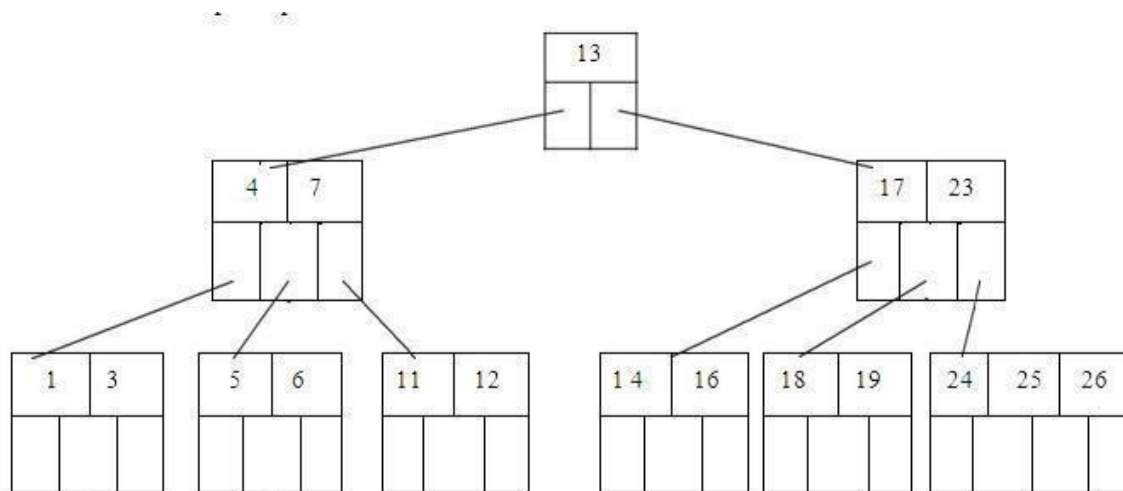
Consider a B-tree



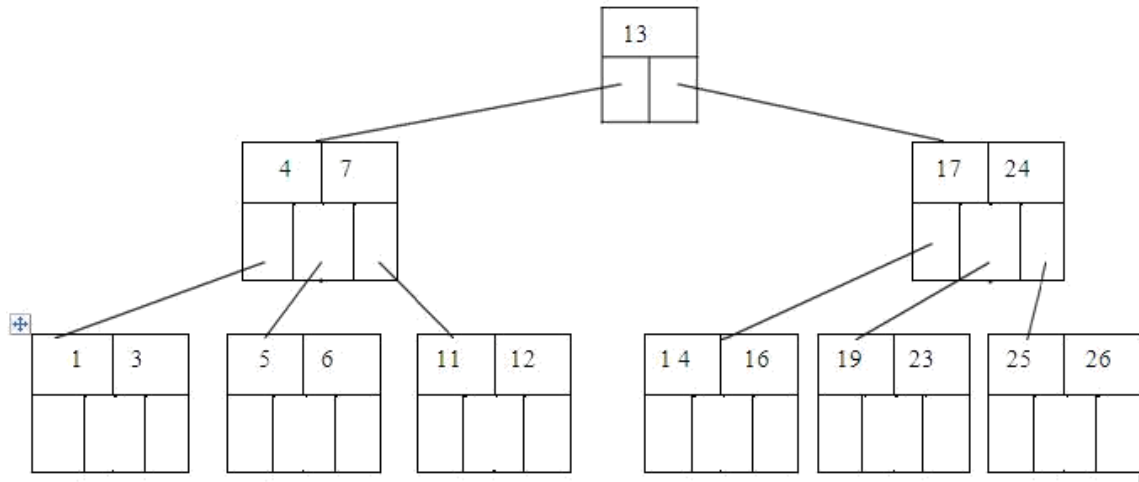
Delete 8, then it is very simple.



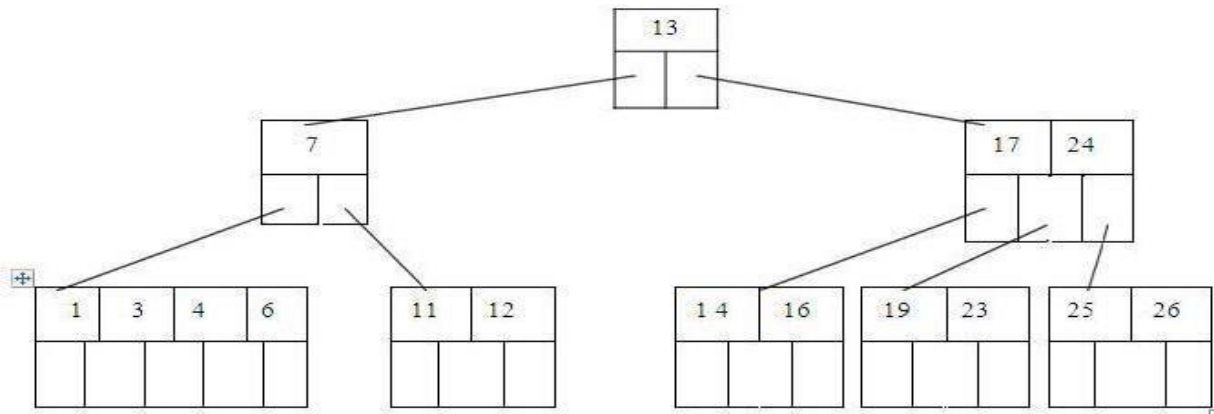
Now we will delete 20, the 20 is not in a leaf node so we will find its successor which is 23, Hence 23 will be moved up to replace 20.



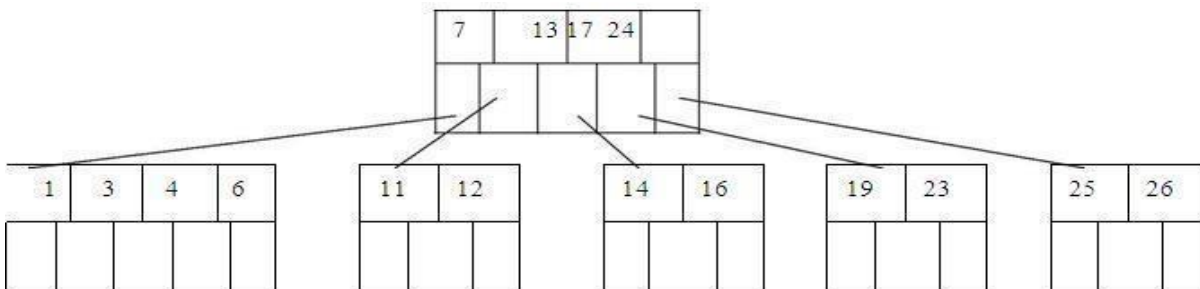
Next we will delete 18. Deletion of 18 from the corresponding node causes the node with only one key, which is not desired (as per rule 4) in B-tree of order 5. The sibling node to immediate right has an extra key. In such a case we can borrow a key from parent and move spare key of sibling up.



Now delete 5. But deletion of 5 is not easy. The first thing is 5 is from leaf node. Secondly this leaf node has no extra keys nor siblings to immediate left or right. In such a situation we can combine this node with one of the siblings. That means remove 5 and combine 6 with the node 1, 3. To make the tree balanced we have to move parent's key down. Hence we will move 4 down as 4 is between 1, 3, and 6. The tree will be-

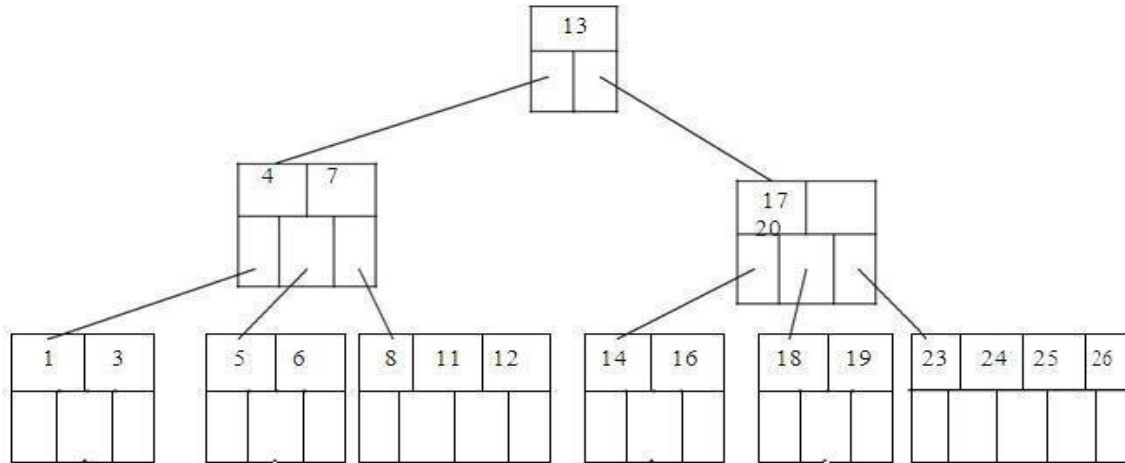


But again internal node of 7 contains only one key which not allowed in B-tree. We then will try to borrow a key from sibling. But sibling 17, 24 has no spare key. Hence we can do is that, combine 7 with 13 and 17, 24. Hence the B-tree will be



Searching

The search operation on B-tree is similar to a search to a search on binary search tree. Instead of choosing between a left and right child as in binary tree, B-tree makes an m-way choice. Consider a B- tree as given below.



If we want to search 11 then

- i. $11 < 13$; Hence search left node
- ii. $11 > 7$; Hence right most node
- iii. $11 > 8$; move in second block
- iv. node 11 is found

The running time of search operation depends upon the height of the tree. It is $O(\log n)$.

Height of B-tree

The maximum height of B-tree gives an upper bound on number of disk access. The maximum number of keys in a B-tree of order $2m$ and depth h is

$$1 + 2m + 2m(m+1) + 2m(m+1)^2 + \dots + 2m(m+1)^{h-1}$$

h

$$= 1 + \sum_{i=1}^{h-1} 2m(m+1)^i$$

$i=1$

The maximum height of B-tree with n keys

$$\frac{\log_{m+1} n}{n)^{2m}} = O(\log$$

Write a Java program to perform the following operations:

a) Insertion into a B-tree b) Searching in a B-tree

```
class BTree
{
    final int MAX = 4;
    final int MIN = 2;
    class BTreeNode // B-Tree node
    {
        int count;
        int key[] = new int[MAX+1];
        BTreeNode child[] = new BTreeNode[MAX+1];
    }
    BTreeNode root = new BTreeNode();
    class Ref // This class creates an object reference
    {
        int m;
    } // and is used to retain/save index values
    // of current node between method calls.
    /*
    * New key is inserted into an appropriate node.
    * No node has key equal to new key (duplicate keys are not allowed.
    */
    void insertTree( int val )
    {
        Ref i = new Ref();
        BTreeNode c = new BTreeNode();
        BTreeNode node = new BTreeNode();
        boolean pushup;
        pushup = pushDown( val, root, i, c );
        if ( pushup )
        {
            node.count = 1;
            node.key[1] = i.m;
            node.child[0] = root;
            node.child[1] = c;
            root = node;
        }
    }
    /*
    * New key is inserted into subtree to which current node points.
    * If pushup becomes true, then height of the tree grows.
    */
    boolean pushDown( int val, BTreeNode node, Ref p, BTreeNode c )
    {
        Ref k = new Ref();
        if ( node == null )
        {
            p.m = val;
            c = null;
            return true;
        }
    }
}
```



```

        else
        {
            if ( searchNode( val, node, k ) )
                System.out.println("Key already exists.");
            if ( pushDown( val, node.child[k.m], p, c ) )
            {
                if ( node.count < MAX )
                {
                    pushIn( p.m, c, node, k.m );
                    return false;
                }
                else
                {
                    split( p.m, c, node, k.m, p, c );
                    return true;
                }
            }
            return false;
        }
    }
}
/*
 * Search through a B-Tree for a target key in the node: val
 * Outputs target node and its position (pos) in the node
 */
BTNode searchTree( int val, BTNode root, Ref pos )
{
    if ( root == null )
        return null ;
    else
    {
        if ( searchNode( val, root, pos ) )
            return root;
        else
            return searchTree( val, root.child[pos.m], pos );
    }
}
/*
 * This method determines if the target key is present in
 * the current node, or not. Seraches keys in the current node;
 * returns position of the target, or child on which to continue search.
 */
boolean searchNode( int val, BTNode node, Ref pos )
{
    if ( val < node.key[1] )
    {
        pos.m = 0 ;
        return false ;
    }
    else
    {
        pos.m = node.count ;

```

```

        while ( ( val < node.key[pos.m] ) && pos.m > 1 )
            (pos.m)--;
    if ( val == node.key[pos.m] )
        return true;
    else
        return false;
    }
}
/*
* Inserts the key into a node, if there is room
* for the insertion
*/
void pushIn( int val, BTreeNode c, BTreeNode node, int k )
{
    int i ;
    for ( i = node.count; i > k ; i-- )
    {
        node.key[i + 1] = node.key[i];
        node.child[i + 1] = node.child[i];
    }
    node.key[k + 1] = val ;
    node.child[k + 1] = c ;
    node.count++ ;
}
/*
* Splits a full node into current node and new right child
* with median. */
void split( int val, BTreeNode c, BTreeNode node,int k, Ref y, BTreeNode newnode )
{
    int i, mid; // mid is median
    if ( k <= MIN )
        mid = MIN;
    else
        mid = MIN + 1;
    newnode = new BTreeNode();
    for ( i = mid+1; i <= MAX; i++ )
    {
        newnode.key[i-mid] = node.key[i];
        newnode.child[i-mid] = node.child[i];
    }
    newnode.count = MAX - mid;
    node.count = mid;
    if ( k <= MIN )
        pushIn ( val, c, node, k );
    else
        pushIn ( val, c, newnode, k-mid );
    y.m = node.key[node.count];
    newnode.child[0] = node.child[node.count] ;
    node.count-- ;
} // calls display( )

```

```

void displayTree()
{
    display( root );
}
// displays the B-Tree
void display( BTreeNode root )
{
    int i;
    if ( root != null )
    {
        for ( i = 0; i < root.count; i++ )
        {
            display( root.child[i] );
            System.out.print( root.key[i+1] + " " );
        }
        display( root.child[i] );
    }
}
} // end of BTree class
//////////////////////////////// BTreeDemo.java //////////////////////////////////
class BTreeDemo
{
    public static void main( String[] args )
    {
        BTree bt = new BTree();

        int[] arr = { 11, 23, 21, 12, 31, 18, 25, 35, 29, 20, 45,
                     27, 42, 55, 15, 33, 36, 47, 50, 39 };
        for ( int i = 0; i < arr.length; i++ )
            bt.insertTree( arr[i] );
        System.out.println("B-Tree of order 5:");
        bt.displayTree();
    }
}

```

```

C:\ Command Prompt

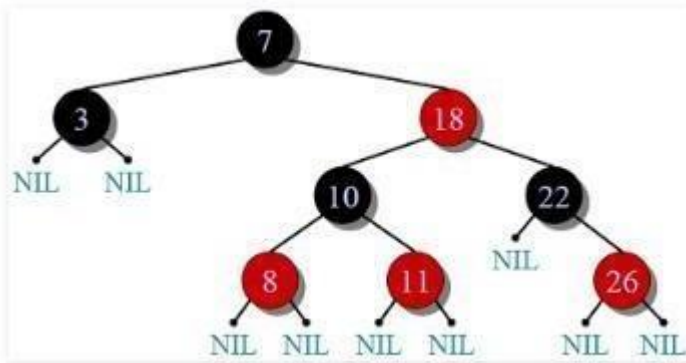
E:\g\ads>java BTreeDemo
B-Tree of order 5:
11 12 15 18 20 21 23 25 27 29 31 33 35 36 39 42 45 47 50 55
E:\g\ads>java BTreeDemo.java
Exception in thread "main" java.lang.NoClassDefFoundError: BTreeDemo/java
Caused by: java.lang.ClassNotFoundException: BTreeDemo.java
    at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
Could not find the main class: BTreeDemo.java.  Program will exit.
E:\g\ads>javac BTreeDemo.java
E:\g\ads>java BTreeDemo
B-Tree of order 5:
11 12 15 18 20 21 23 25 27 29 31 33 35 36 39 42 45 47 50 55
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>

```

Red-Black Tree | Set 1 (Introduction)

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.

- 1) Every node has a color either red or black.
- 2) Root of tree is always black.
- 3) There are no two adjacent red nodes (A red node cannot have a red parent or redchild).



- 4) Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of a Red-Black tree is always $O(\log n)$ where n is the number of nodes in the tree.

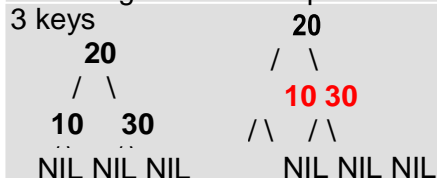
Red-Black Tree ensure balance?

A simple example to understand balancing is, a chain of 3 nodes is not possible in the Red-Black tree. We can try any combination of colours and see all of them violate Red-Black tree property. A chain of 3 nodes

is nodes is not possible in Red-Black Trees.



Following are different possible Red-Black Trees with above



From the above examples, we get some idea how Red-Black trees ensure balance. Following is an important fact about balancing in Red-Black Trees.

Black Height of a Red-Black Tree :

Black height is number of black nodes on a path from root to a leaf. Leaf nodes are also counted black nodes. From above properties 3 and 4, we can derive, a Red-Black Tree of height h has black-height $\geq h/2$.

Every Red Black Tree with n nodes has height \leq

$2\log_2(n+1)$ This can be proved using following facts:

1. For a general Binary Tree, let k be the minimum number of nodes on all root to NULL paths, then $n \geq 2^k - 1$ (Ex. If k is 3, then n is atleast 7). This expression can also be written as $k \leq \log_2(n+1)$
2. From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $\log_2(n+1)$ black nodes.
3. From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least $\lfloor n/2 \rfloor$ where n is the total number of nodes.

From above 2 points, we can conclude the fact that Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$

Red-Black Tree (Insert)

In AVL tree insertion, we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.

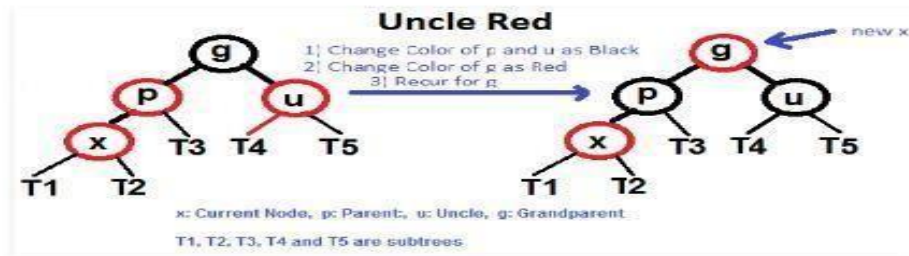
- 1) Recoloring
- 2) Rotation

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithm has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Color of a NULL node is considered as

BLACK. Let x be the newly inserted node.

- 1) Perform standard BST insertion and make the color of newly inserted nodes as RED.
- 2) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).
- 3) Do following if color of x 's parent is not BLACK and x is not root.
 - a) If x 's uncle is RED
 - 4) (Grand parent must have been black from property 4)
 - (i) Change color of parent and uncle as BLACK.
 - (ii) color of grand parent as RED.
 - (iii) Change $x = x$'s grandparent, repeat steps 2 and 3 for new x .



....b) If x's uncle is **BLACK**, then there can be four configurations for x, x's parent (p) and x's grandparent

(g) (This is similar to **AVL Tree**)

.....i) Left Left Case (p is left child of g and x is left child of p)

.....ii) Left Right Case (p is left child of g and x is right child of p)

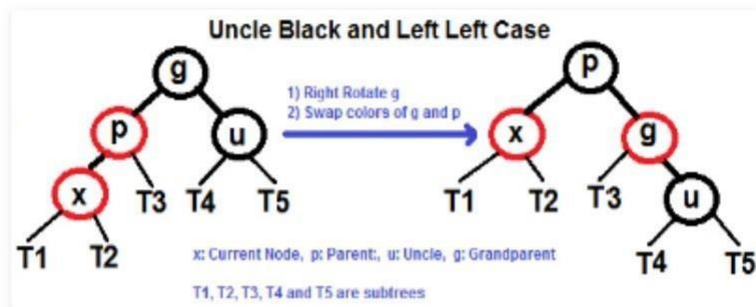
.....iii) Right Right Case (Mirror of case i)

.....iv) Right Left Case (Mirror of case ii)

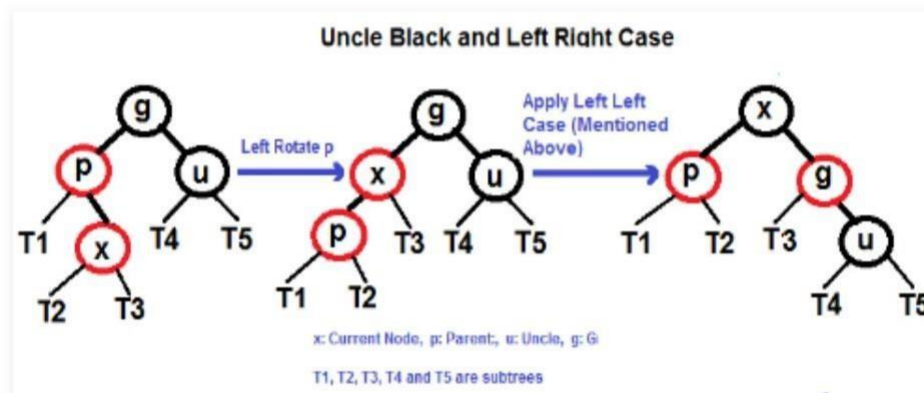
Following are operations to be performed in four subcases when uncle is **BLACK**.

All four cases when Uncle is **BLACK**

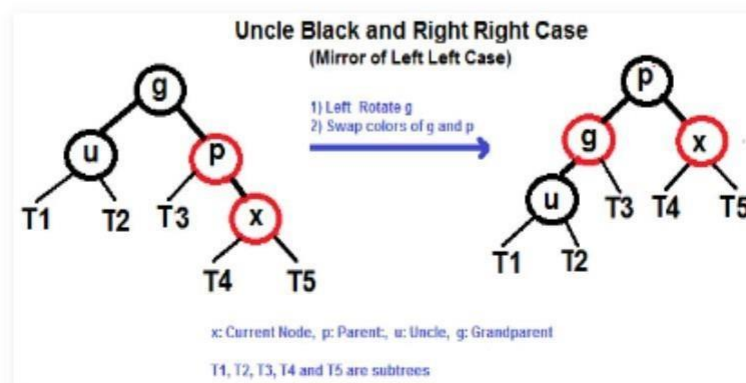
Left Left Case (See g, p and x)



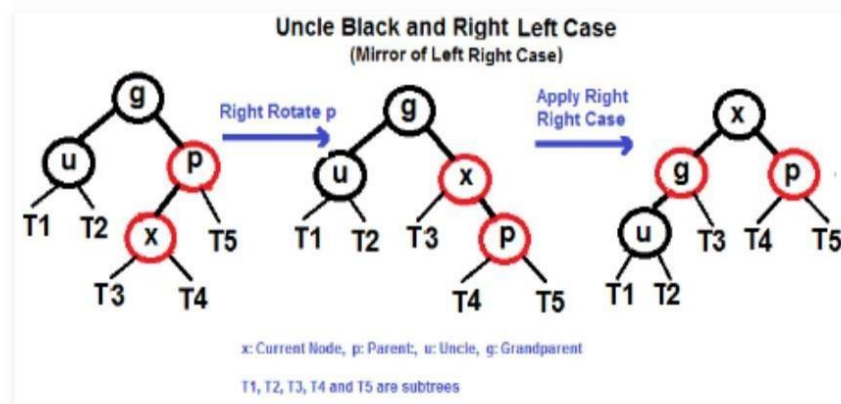
Left Right Case (See g, p and x)



Right Right Case (See g, p and x)

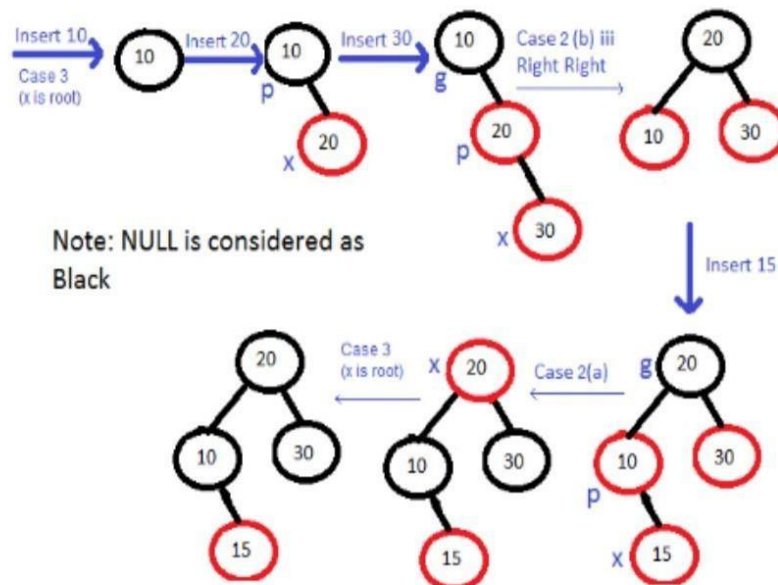


Right Left Case (See g, p and x)



Examples of Insertion

Insert 10, 20, 30 and 15 in an empty tree



Red-Black Tree (Delete)

Insertion Vs Deletion:

Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.

In insert operation, we check color of uncle to decide the appropriate case. In delete operation, **we check color of sibling** to decide the appropriate case.

The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

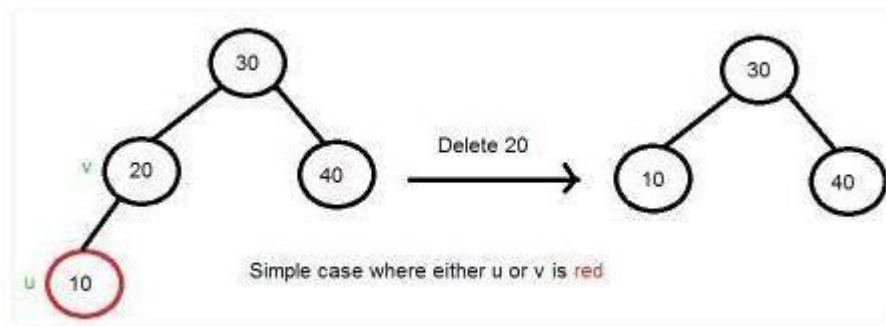
Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as **double black**. The main task now becomes to convert this double black to single black.

Deletion Steps

Following are detailed steps for deletion.

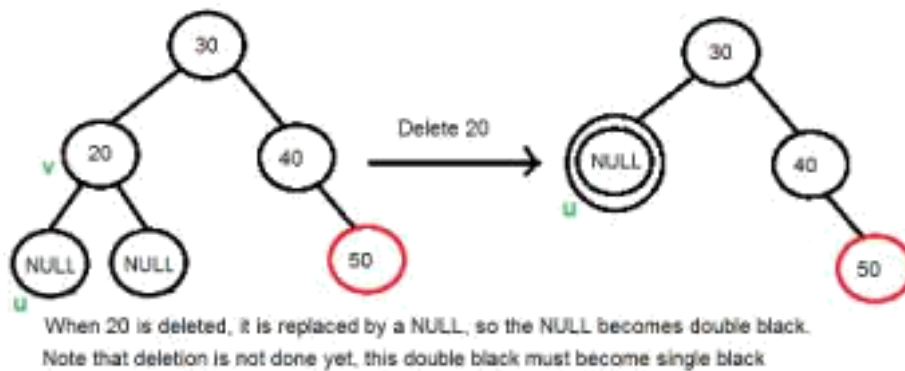
1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

2) **Simple Case: If either u or v is red**, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



3) If Both u and v are Black.

Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.

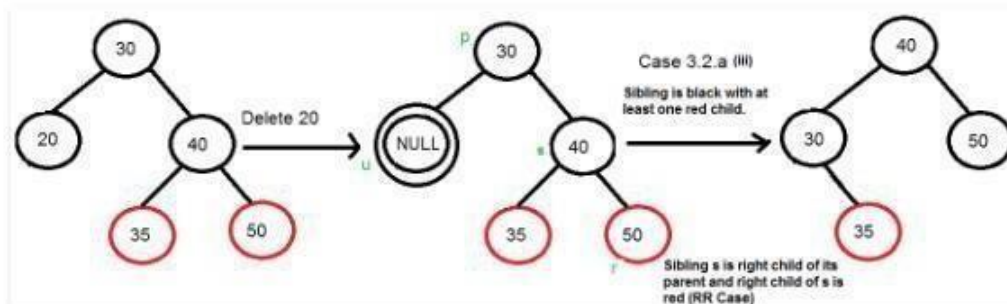


Do following while the current node u is double black and it is not root. Let sibling of node be s .
(a): If sibling s is black and at least one of sibling's children is red, perform rotation(s). Let the red child of s be r . This case can be divided in four subcases depending upon positions of s and r .

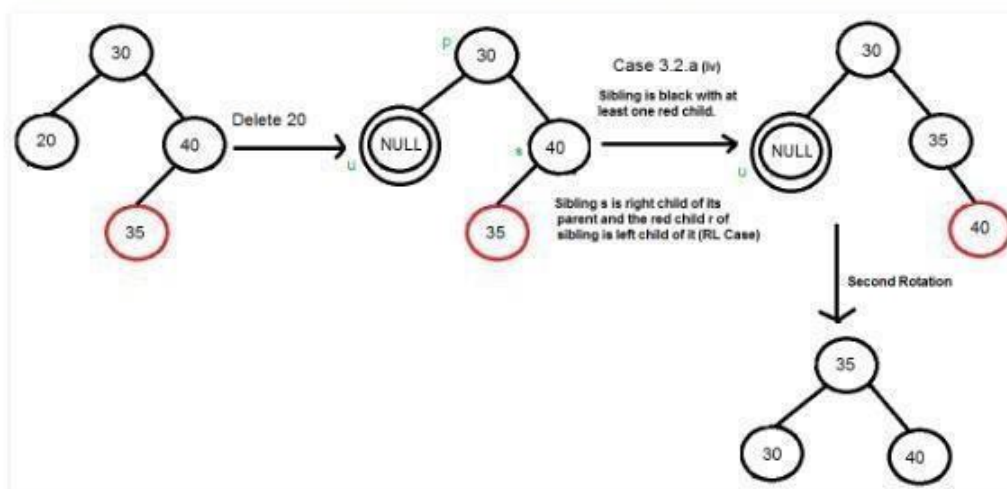
.....(i) Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

.....(ii) Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram

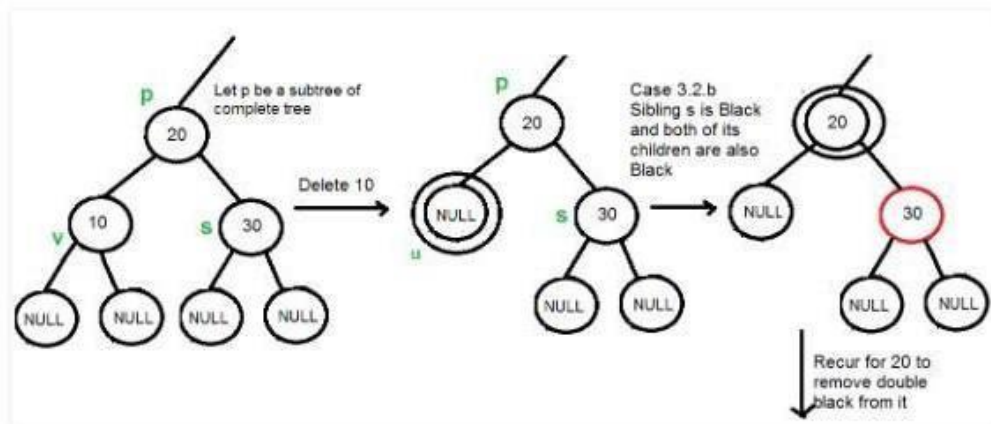
(iii) Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



.....(iv) Right Left Case (s is right child of its parent and r is left child of s)



.....(b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.



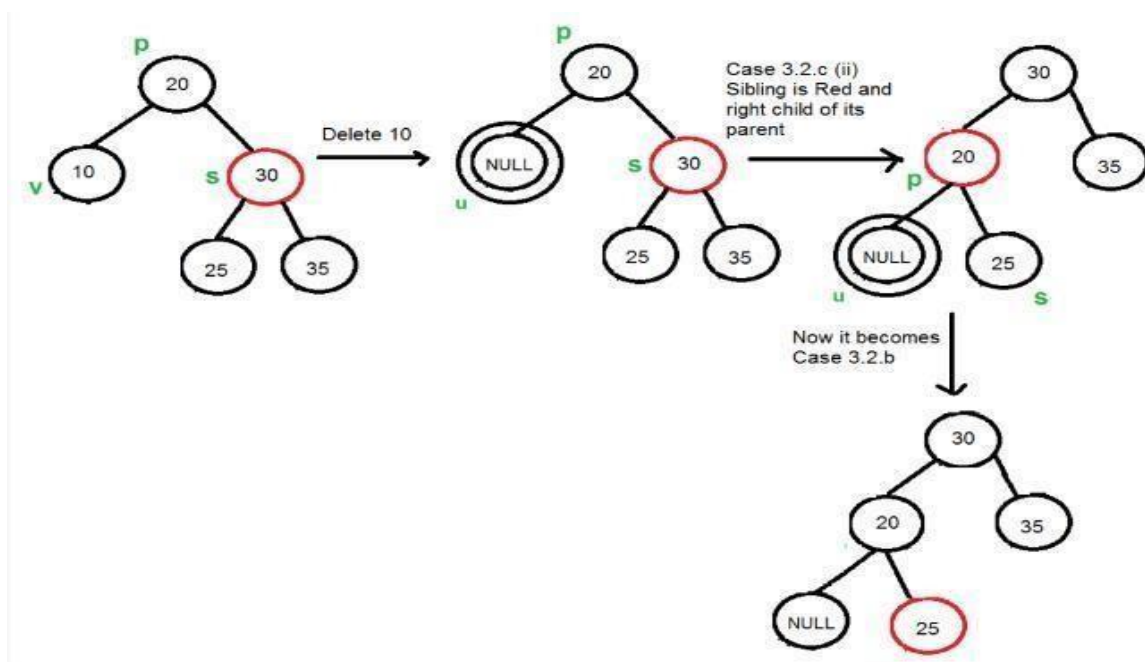
In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

.....(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The

new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

.....(i) Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

.....(iii) Right Case (s is right child of its parent). We left rotate the parent p.



If u is root, make it single black and return (Black height of complete tree reduces by 1).

Text compression-Huffman coding and decoding

Huffman Coding | Greedy Algo-3

Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

See [this](#) for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding

- 1) Build a Huffman Tree from input characters.
- 2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

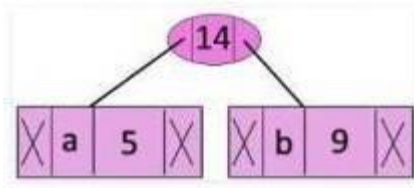
1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

| character | Frequency |
|-----------|-----------|
| a | 5 |
| b | 9 |
| c | 12 |
| d | 13 |
| e | 16 |
| f | 45 |

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

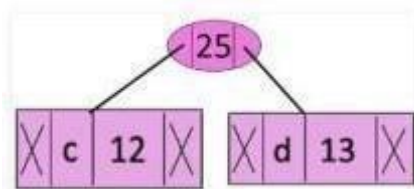
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency $5+9=14$.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

| character | Frequency |
|----------------------|-----------|
| c | 12 |
| d | 13 |
| Internal Node | 14 |
| e | 16 |
| f | 45 |

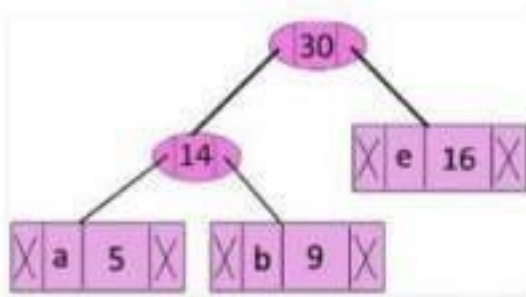
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency $12+13=25$



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

| character | Frequency |
|----------------------|-----------|
| Internal Node | 14 |
| e | 16 |
| Internal Node | 25 |
| f | 45 |

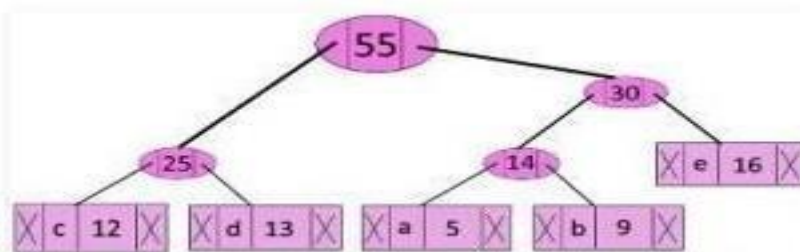
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency $14 + 16 = 30$



Now min heap contains 3 nodes.

| character | Frequency |
|---------------|-----------|
| Internal Node | 25 |
| Internal Node | 30 |
| f | 45 |

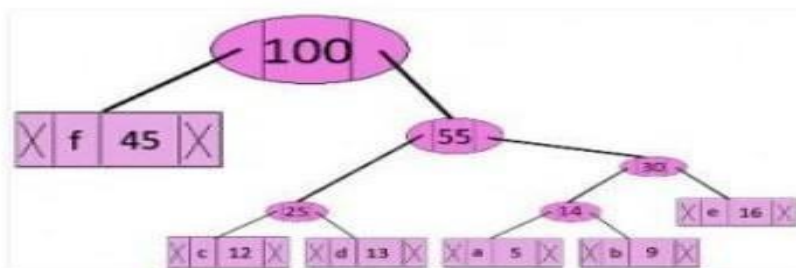
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency $25 + 30 = 55$



Now min heap contains 2 nodes.

| character | Frequency |
|---------------|-----------|
| f | 45 |
| Internal Node | 55 |

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency $45 + 55 = 100$



Now min heap contains only one node.

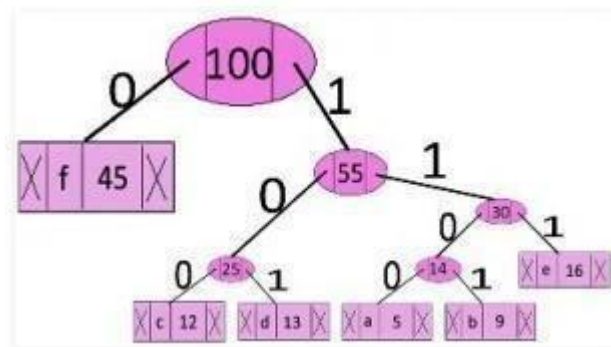
| character | Frequency |
|-----------|-----------|
|-----------|-----------|

| | |
|---------------|-----|
| Internal Node | 100 |
|---------------|-----|

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

| character | code-word |
|-----------|-----------|
|-----------|-----------|

| | |
|---|---|
| F | 0 |
|---|---|

| | |
|---|-----|
| C | 100 |
|---|-----|

| | |
|---|-----|
| D | 101 |
|---|-----|

| | |
|---|------|
| A | 1100 |
|---|------|

| | |
|---|------|
| B | 1101 |
|---|------|

| | |
|---|-----|
| E | 111 |
|---|-----|

```
import
java.util.PriorityQueue;
import java.util.Scanner;
import java.util.Comparator;
// node class is the basic structure
// of each node present in the Huffman -
tree. class HuffmanNode {
    int data; char c;
    HuffmanNode left;
    HuffmanNode
    right;
}
```



```
// comparator class helps to compare the node
// on the basis of one of its attribute.
// Here we will be compared
// on the basis of data values of the nodes.
class MyComparator implements Comparator<HuffmanNode>
{ public int compare(HuffmanNode x, HuffmanNode y)
{
    return x.data - y.data;
}
}

public class Huffman {

    // recursive function to print the
    // huffman-code through the tree traversal.
    // Here s is the huffman - code generated.
    public static void printCode(HuffmanNode root, String s)
    {
        // base case; if the left and right are null
        // then its a leaf node and we print
        // the code s generated by traversing the
        // tree. if (root.left
        == null
        && root.right
        == null
        && Character.isLetter(root.c)) {

            // c is the character in the node
            System.out.println(root.c + ":" +
            s);

            return;
        }

        // if we go to left then add "0" to the code.
        // if we go to the right add "1" to the code.
        // recursive calls for left and
        // right sub-tree of the
        // generated tree.
        printCode(root.left, s + "0");
        printCode(root.right, s + "1");
    }
}
```

```
// main function
public static void main(String[] args)
{

    Scanner s = new Scanner(System.in);

    // number of
    characters. int n = 6;
    char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f'
    }; int[] charfreq = { 5, 9, 12, 13, 16, 45
    };

    // creating a priority queue q.
    // makes a min-priority queue(min-
    heap). PriorityQueue<HuffmanNode>
    q
        = new PriorityQueue<HuffmanNode>(n, new

    MyComparator()); for (int i = 0; i < n; i++) {

        // creating a Huffman node object
        // and add it to the priority queue.
        HuffmanNode hn = new
        HuffmanNode();

        hn.c = charArray[i];
        hn.data =
        charfreq[i];

        hn.left = null;
        hn.right =
        null;

        // add functions adds
        // the huffman node to the
        queue. q.add(hn);
    }

    // create a root node
    HuffmanNode root =
    null;

    // Here we will extract the two minimum value
    // from the heap each time until
    // its size reduces to 1, extract until
```

```
// all the nodes are  
extracted. while (q.size() >  
1) {
```

```
// first min extract.
HuffmanNode x =
q.peek(); q.poll();

// second min extract.
HuffmanNode y =
q.peek(); q.poll();

// new node f which is equal
HuffmanNode f = new
HuffmanNode();

// to the sum of the frequency of the two nodes
// assigning values to the f
node. f.data = x.data + y.data;
f.c = '-';

// first extracted node as left
child. f.left = x;

// second extracted node as the right
child. f.right = y;

// marking the f node as the root
node. root = f;

// add this node to the
priority- queue. q.add(f);
}
// print the codes by traversing
the tree printCode(root, "");
}}

// This code is contributed by Kunwar Desh Deepak Singh
```

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Time complexity: $O(n \log n)$ where n is the number of unique characters. If there are n nodes, `extractMin()` is called $2^{*}(n - 1)$ times. `extractMin()` takes $O(\log n)$ time as it calls `minHeapify()`. So, overall complexity is $O(n \log n)$.

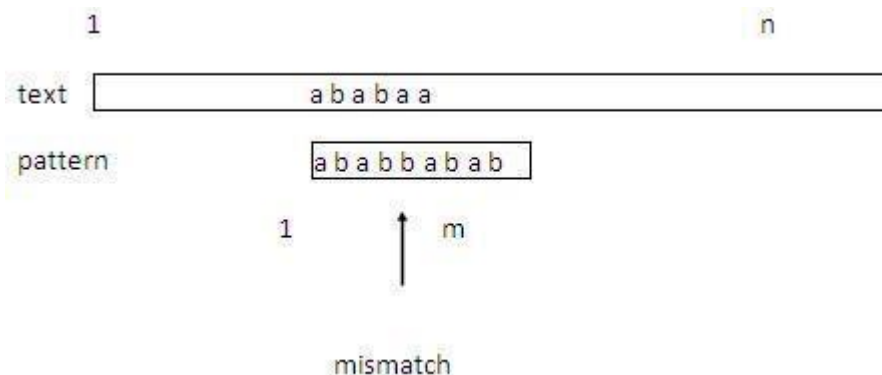
Pattern matching-KMP algorithm.

Pattern Matching Algorithms

Introduction

Pattern matching is to find a pattern, which is relatively small, in a text, which is supposed to be very large. Patterns and texts can be one-dimensional, or two-dimensional. In the case of one-dimensional, examples can be text editor and DNA analysis. In the text editor, we have 26 characters and some special symbols, whereas in the DNA case, we have four characters of A, C, G, and T. In the text editor, a pattern is often a word, whose length is around 10, and the length of the text is a few hundred up to one million. In the DNA analysis, a gene is a few hundred long and the human genome is about 3 billion long.

In the case of two-dimensional, the typical application is a pattern matching in computer vision. A pattern is about (100, 100) and text typically (1024,1024). Either one-dimensional or two-dimensional, the text is very large, and therefore a fast algorithm to find the occurrence(s) of pattern in it is needed. We start from a naive algorithm for one-dimensional.



At first the pattern is set to the left end of the text, and matching process starts. After a mismatch is found, pattern is shifted one place right and a new matching process starts, and so on. The pattern and text are in arrays `pat[1..m]` and `text[1..n]` respectively.

Algorithm 1. Naive pattern matching

```

algorithm 1. j:=1;

2. while j <= n-m+1 do begin
3.   i:=1;
4.   while (i<=m) and (pat[i]=text[j]) do begin
5.     i:=i+1;
6.     j:=j+1
7.   end;
8.   if i<=m then j:=j-i+2 /* shift the pattern one place right */
9.   else write("found at ", j-i+1)
10. end.

```

The worst case happens when pat and text are all a's but b at the end, such as pat = aaaaab and text = aaaaaaaaaaaaaaaaaaaaaaaab. The time is obviously $O(mn)$. On average the situation is not as bad, but in the next section we introduce a much better algorithm. We call the operation $\text{pat}[i]=\text{text}[j]$ a comparison between characters, and measure the complexity of a given algorithm by the number of character comparisons. The rest of computing time is proportional to this measure.

Knuth-Morris-Pratt algorithm (KMP algorithm)

When we shift the pattern to the right after a mismatch is found at i on the pattern and j on the text, we did not utilise the matching history for the portion $\text{pat}[1..i]$ and $\text{text}[j-i+1..j]$. If we can get information on how much to shift in advance, we can shift the pattern more, as shown in the following example.

Example 1.

```

      1 2 3 4 5 6 7 8 9 10 11 12 13 14
text  a b a b a a b b a b a b
ba pattern a b a b b
      a b a b b
        a b a b b
          a b a b b

```

After mismatch at 5, there is no point in shifting the pattern one place. On the other hand we know “ab” repeats itself in the pattern. We also need the condition $\text{pat}[3] \neq \text{pat}[5]$ to ensure that we do not waste a match at position 5. Thus after shifting two places, we resume matching at position 5. Now we have a mismatch at position 6. This time shifting two places does not work, since “ab” repeats itself and we know that shifting two places will invite a mismatch.

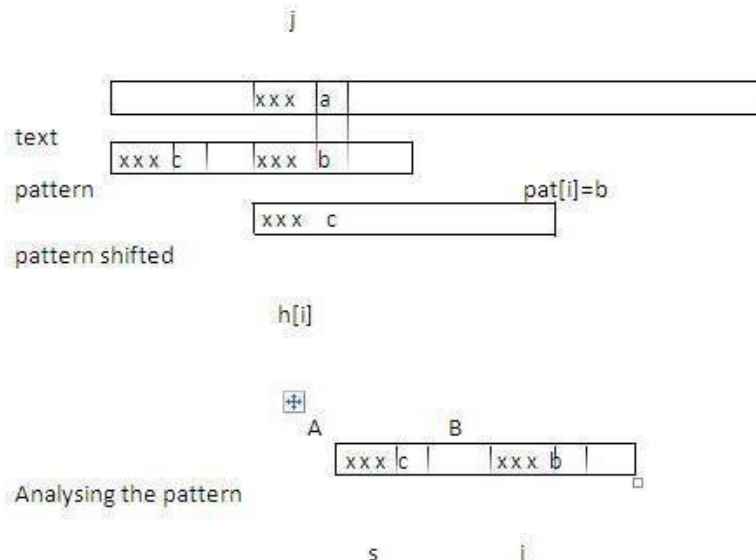
The condition $\text{pat}[1] < \text{pat}[4]$ is satisfied. Thus we shift pat three places and resume matching at position 6, and find a mismatch at position 8. For a similar reason to the previous case, we shift pat three places, and finally we find the pattern at position 9 of the text. We spent 15 comparisons between characters. If we followed Algorithm 1, we would spend 23 comparisons. Confirm this.

The information of how much to shift is given by array $h[1..m]$, which is defined by $h[1] = 0$

$$h[i] = \max \{ s \mid (s=0) \text{ or } (pat[1 \dots s-1] = pat[i-s+1 \dots i-1]) \text{ and}$$

pat[s]<>pat[i]) } The situation is illustrated in the following figure.

Main matching process



The meaning of $h[i]$ is to maximise the portion A and B in the above figure, and require $b < c$. The value of $h[i]$ is such maximum s . Then in the main matching process, we can resume matching after we shift the pattern after a mismatch at i on the pattern to position $h[i]$ on the pattern, and we can keep going with the pointer j on the text. In other words, we need not to backtrack on the text. The maximisation of such s , (or minimisation of shift), is necessary in order not to overlook an occurrence of the pattern in the text. The main matching algorithm follows.

Algorithm 2. Matching

algorithm 1. $i:=1$; $j:=1$;

2. **while** ($i \leq m$) **and** ($j \leq n$) **do begin**

3. **while** ($i > 0$) **and** ($\text{pat}[i] \neq \text{text}[j]$) **do**

$i:=h[i]$; 4. $i:=i+1$; $j:=j+1$

5. **end**

6. **if** $i \leq m$ **then write** ("not found")

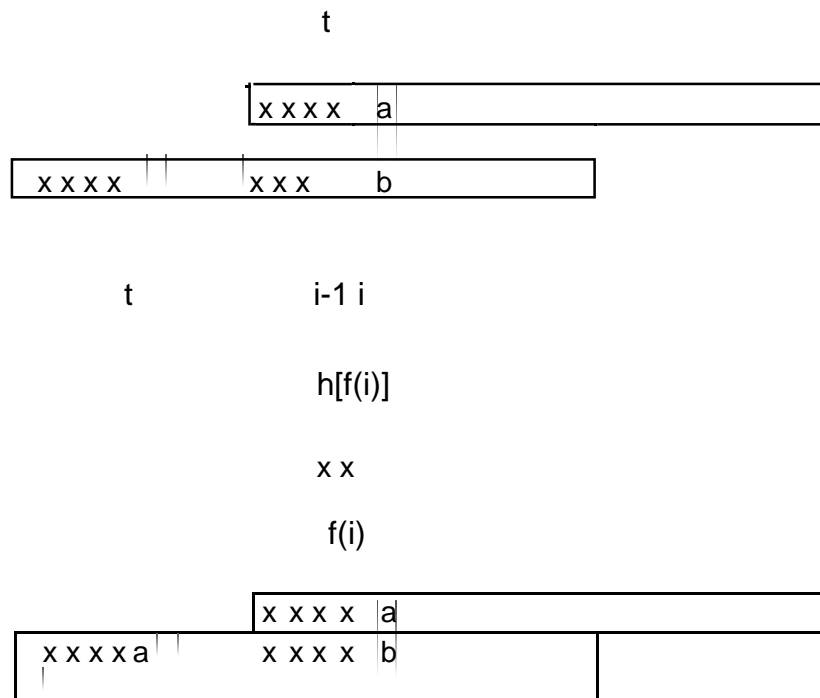
7. **else write** ("found at", $j-i+1$)

Let the function $f(i)$ be defined

by $f(1) = 0$

$f(i) = \max\{s \mid (1 \leq s < i) \text{ and } (\text{pat}[1 \dots s-1] = \text{pat}[i-s+1 \dots i-1])\}$

The definitions of $h[i]$ and $f(i)$ are similar. In the latter we do not care about $\text{pat}[s] \neq \text{pat}[i]$. The computation of h is like pattern matching of pat on itself.



Algorithm 3. Computation of h

```

1. t:=0; h[1]:=0;
2. for i:=2 to m do begin
3.   /* t = f(i-1) */
4.   while (t>0) and (pat[i-1]<>pat[t]) do
t:=h[t]; 5. t:=t+1;
6.   /* t=f(i) */
7.   if pat[i]<>pat[t] then h[i]:=t else h[i]:=h[t]
8. end

```

The computation of h proceeds from left to right. Suppose $t=f(i-1)$. If $pat[t]=pat[i-1]$, we can extend the matched portion by one at line 5. If $pat[t] \neq pat[i-1]$, by repeating $t:=h[t]$, we effectively slide the pattern over itself until we find $pat[t]=pat[i-1]$, and we can extend the matched portion. If $pat[i] \neq pat[t]$ at line 7, the position t satisfies the condition for h, and so $h[i]$ can be set to t. If $pat[i]=pat[t]$, by going one step by $h[t]$, the position will satisfy the condition for h, and thus $h[i]$ can be set to $h[t]$.

Example. pat = a b a b b

i = 2, at line 7 $t=1$, $pat[1] \neq pat[2]$, $f(2) = 1$,

```

      h[2]:=1 i
a b a b b
      a b a b
        b t

```

i = 3, $t=1$ at line 4. $pat[1] \neq pat[2]$, $t:=h[1]=0$, at line 7 $t=1$, $f(3)=1$,

```

      pat[1]=pat[3], h[3]:=0 a b a b b
        a b a b b

```

i = 4, $t=1$ at line 4. $pat[3]=pat[1]$, $t:=t+1$, $t=2$. At line 7, $pat[4]=pat[2]$, $h[4]:=h[2]=1$

i = 5, $t=2$, $f(4)=2$. At line 4, $pat[4]=pat[2]$, $t:=t+1=3$. $f(5)=3$. At line 7, $pat[5] \neq pat[3]$, $h[5]:=t=3$

Finally we have

```

i | 1   2 3   4 5
-----
pat | a  b a     b b
f |   0 1 1   2 3
h |       01013

```

The time of Algorithm 2 is clearly $O(n)$, since each character in the text is examined at most twice, which gives the upper bound on the number of comparisons. Also, whenever a mismatch is found, the pattern is shifted to the right. The shifting can not take place more than $n-m_1$ times. The analysis of Algorithm 3 is a little more tricky. Trace the changes on the value of t in the algorithm. We have a doubly nested loop, one by the outer for and the other by while. The value of t can be increased by one at line 5, and $m-1$ times in total, which we regard as income. If we get into the while loop, the value of t is decreased, which we regard as expenditure. Since the total income is $m-1$, and t can not go to negative, the total number of executions of $t:=h[t]$ can not exceed $m-1$. Thus the total time is $O(m)$.

Summarising these analyses, the total time for the KMP algorithm, which includes the pre- processing of the pattern, is $O(m+n)$, which is linear.

Source code:.

//KMPDemo.java

```

import
java.io.*; class
KMPDemo
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader br=new BufferedReader(new
                                InputStreamReader(System.in));
        System.out.println(" Enter any String:");
        String T = br.readLine();
        BufferedReader br1=new BufferedReader(new
                                InputStreamReader(System.in));
        System.out.println(" Enter String for pattern
        matching:"); String P = br1.readLine();
    }
}

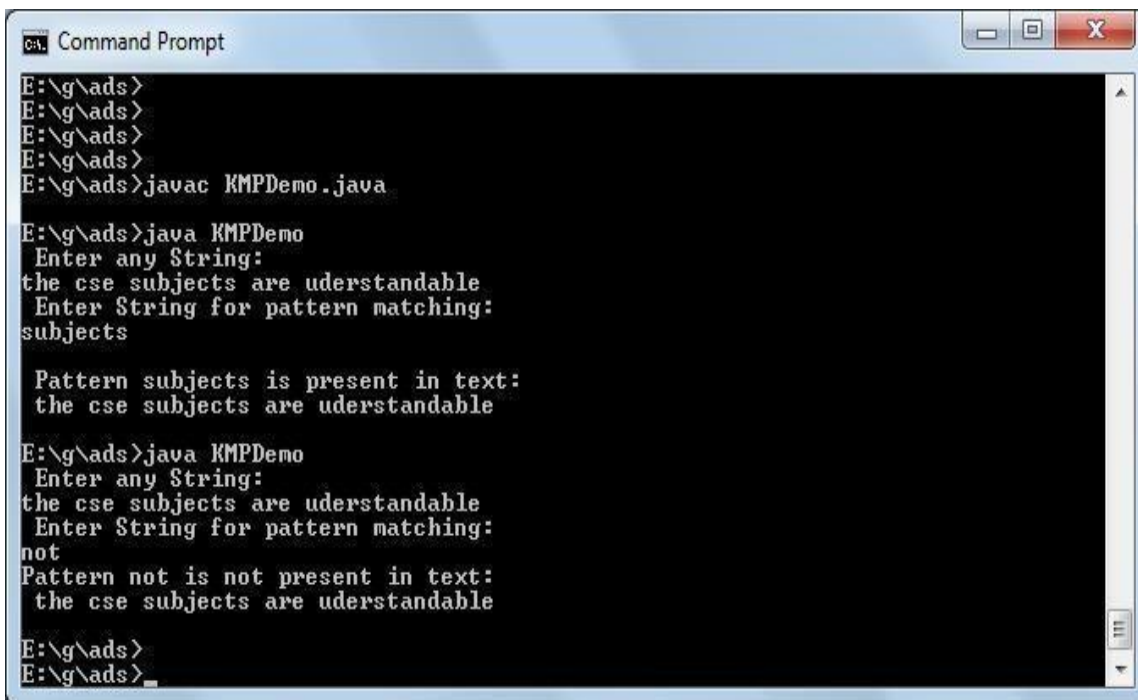
```

```
        boolean isMatch = kmp(T,
        P); if(isMatch)
        System.out.println("\n Pattern " + P + " is present in text:\n "
        + T); else
        System.out.println("Pattern " + P + " is not present in text:\n " + T);
    }
    static boolean kmp(String T, String P)
    {
        int n =
        T.length(); int m
        = P.length();
        int[] fail =
        computeFailFunction(P); int i =
        0; // text index
        int j = 0; // pattern index
        while( i < n )
        {
            if( P.charAt(j) == T.charAt(i) )
            {
                if( j == m-1 )
                    return true;

                i++;
                ;
                j++;
                ;
            }
            else if( j > 0 )
                j = fail[j]-
                1]; else i++;
        }
        return
        false;
    }
    static int[] computeFailFunction( String P )
    {
        int m = P.length();
        int[] fail = new
        int[m]; fail[0] = 0;
        int i = 1;
        int j = 0;
        while( i < m )
        {
```

```
        if( P.charAt(j) == P.charAt(i) ) // j+1 characters match
        {
            fail[i] =
                j+1; i++;
            j++;
        }
        else if( j > 0 ) // j follows a matching
            prefix j = fail[j-1];
        else // no match
        {
            fail[i] =
                0; i++;
        }
    }
}
}}    return fail;
```

OUTPUT:



```
Command Prompt
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>
E:\g\ads>javac KMPDemo.java
E:\g\ads>java KMPDemo
Enter any String:
the cse subjects are understandable
Enter String for pattern matching:
subjects

Pattern subjects is present in text:
the cse subjects are understandable

E:\g\ads>java KMPDemo
Enter any String:
the cse subjects are understandable
Enter String for pattern matching:
not
Pattern not is not present in text:
the cse subjects are understandable

E:\g\ads>
E:\g\ads>
```

Comparison of Search Trees in Data Structure

Here we will see some search trees and their differences. There are many different search trees. They are different in nature. The basic search tree is Binary Search Tree (BST). Some other search trees are AVL tree, B tree, Red-Black tree, splay tree etc.

These trees can be compared based on their operations. We will see the time complexity of these trees

| Search Tree | Average Case | | |
|--------------------|---------------|---------------|---------------|
| | Insert | Delete | Search |
| Binary Search Tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| AVL tree | $O(\log_2 n)$ | $O(\log_2 n)$ | $O(\log_2 n)$ |
| B Tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| Red-Black Tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| Splay Tree | $O(\log_2 n)$ | $O(\log_2 n)$ | $O(\log_2 n)$ |

| Search Tree | Worst Case | | |
|--------------------|---------------|---------------|---------------|
| | Insert | Delete | Search |
| Binary Search Tree | $O(n)$ | $O(n)$ | $O(n)$ |
| AVL tree | $O(\log_2 n)$ | $O(\log_2 n)$ | $O(\log_2 n)$ |
| B Tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| Red-Black Tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| Splay Tree | $O(\log_2 n)$ | $O(\log_2 n)$ | $O(\log_2 n)$ |

Trees in java.util- TreeSet, Tree Map Classes, Tries(examples only)

TreeSet in Java

TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface. It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. The TreeSet implements a NavigableSet interface by inheriting AbstractSet class.

Few important features of TreeSet are as follows:

1. TreeSet implements the SortedSet interface so duplicate values are not allowed.
2. Objects in a TreeSet are stored in a sorted and ascending order.
3. TreeSet does not preserve the insertion order of elements but elements are sorted by keys.
4. TreeSet does not allow to insert Heterogeneous objects. It will throw classCastException at Runtime if trying to add heterogeneous objects.
5. TreeSet serves as an excellent choice for storing large amounts of sorted information which are supposed to be accessed quickly because of its faster access and retrieval time.
6. TreeSet is basically implementation of a self-balancing binary search tree like Red-Black Tree. Therefore operations like add, remove and search take $O(\log n)$ time. And operations like printing n elements in sorted order takes $O(n)$ time.

Constructors of TreeSet class:

1. **TreeSet t = new TreeSet();**
This will create empty TreeSet object in which elements will get stored in default natural sorting order.
2. **TreeSet t = new TreeSet(Comparator comp);**
This constructor is used when external specification of sorting order of elements is needed.
3. **TreeSet t = new TreeSet(Collection col);**
This constructor is used when any conversion is needed from any Collection object to TreeSet object.
4. **TreeSet t = new TreeSet(SortedSet s);**
This constructor is used to convert SortedSet object to TreeSet Object.

Synchronized TreeSet:

The implementation in a TreeSet is not synchronized in a sense that if multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be “wrapped” using the Collections.synchronizedSortedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set

Below program illustrates the basic operation of a TreeSet:

// Java program to demonstrate insertions in
TreeSet import java.util.*;

```
class TreeSetDemo {  
    public static void main(String[] args)  
    {  
        TreeSet<String> ts1 = new TreeSet<String>();  
  
        // Elements are added using add()  
        method ts1.add("A");  
        ts1.add("B");  
        ts1.add("C");  
  
        // Duplicates will not get  
        insert ts1.add("C");  
  
        // Elements get stored in default natural  
        // Sorting  
        Order(Ascending)  
        System.out.println(ts1);  
    }  
}
```

Output:

[A, B, C]

Two things must be kept in mind while creating and adding elements into a TreeSet:

- Firstly, insertion of null into a TreeSet throws *NullPointerException* because while insertion of null, it gets compared to the existing elements and null cannot be compared to any value.
- Secondly, if we are depending on default natural sorting order, compulsory the object should be **homogeneous** and **comparable** otherwise we will get **RuntimeException: ClassCastException**

NOTE:

1. An object is said to be comparable if and only if the corresponding class implements **Comparable interface**.
2. **String** class and all **Wrapper** classes already implements Comparable interface but **StringBuffer** class doesn't implements Comparable interface. Hence we got *ClassCastException* in the above example.
3. For an empty tree-set, when trying to insert null as first value, one will get NPE from JDK 7. From 1.7 onwards null is not at all accepted by TreeSet. However upto JDK 6, null will be accepted as first value, but any if insertion of any more values in the TreeSet, will also

throw

NullPointerException.

Hence it was considered as bug and thus removed in JDK 7.

Methods of TreeSet class:

TreeSet implements SortedSet so it has availability of all methods in Collection, Set and SortedSet interfaces. Following are the methods in TreeSet interface.

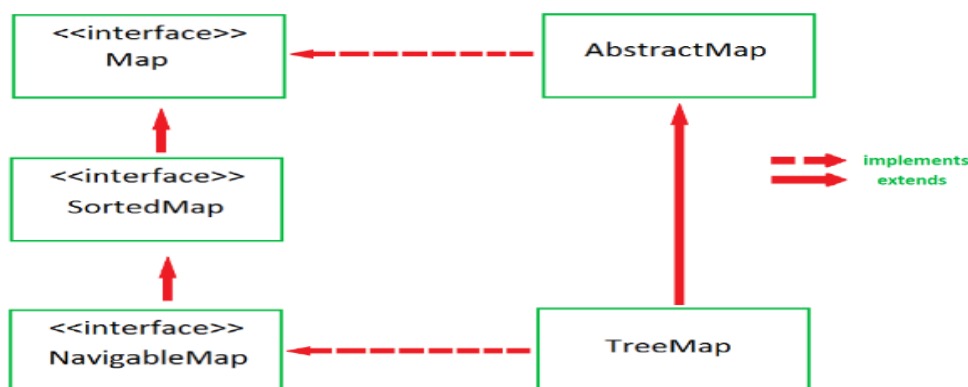
1. void add(Object o): This method will add specified element according to some sorting order in TreeSet. Duplicate entries will not get added.
2. boolean addAll(Collection c): This method will add all elements of specified Collection to the set. Elements in Collection should be homogeneous otherwise ClassCastException will be thrown. Duplicate Entries of Collection will not be added to TreeSet.
3. void clear(): This method will remove all the elements.
4. boolean contains(Object o): This method will return true if given element is present in TreeSet else it will return false.
5. Object first(): This method will return first element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
6. Object last(): This method will return last element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
7. SortedSet headSet(Object toElement): This method will return elements of TreeSet which are less than the specified element.
8. SortedSet tailSet(Object fromElement): This method will return elements of TreeSet which are greater than or equal to the specified element.
9. SortedSet subSet(Object fromElement, Object toElement): This method will return elements ranging from fromElement to toElement. fromElement is inclusive and toElement is exclusive.
10. boolean isEmpty(): This method is used to return true if this set contains no elements or is empty and false for the opposite case.
11. Object clone(): The method is used to return a shallow copy of the set, which is just a simple copied set.
12. int size(): This method is used to return the size of the set or the number of elements present in the set.
13. boolean remove(Object o): This method is used to return a specific element from the set.
14. Iterator iterator(): Returns an iterator for iterating over the elements of the set.
15. Comparator comparator(): This method will return Comparator used to sort elements in TreeSet or it will return null if default natural sorting order is used.
16. **ceiling(E e)**: This method returns the least element in this set greater than or equal to the given element, or null if there is no such element.

17. **descendingIterator():** This method returns an iterator over the elements in this set in descending order.
18. **descendingSet():** This method returns a reverse order view of the elements contained in this set.
19. **floor(E e):** This method returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
20. **higher(E e):** This method returns the least element in this set strictly greater than the given element, or null if there is no such element.
21. **lower(E e):** This method returns the greatest element in this set strictly less than the given element, or null if there is no such element.

TreeMap in Java

The TreeMap in Java is used to implement Map interface and NavigableMap along with the Abstract Class. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and storing the key-value pairs. The storing order maintained by the treemap must be consistent with equals just like any other sorted map, irrespective of the explicit comparators. The treemap implementation is not synchronized in the sense that if a map is accessed by multiple threads, concurrently and at least one of the threads modifies the map structurally, it must be synchronized externally. Some important features of the treemap are:

1. This class is a member of Java Collections Framework.
2. The class implements Map interfaces
including NavigableMap, SortedMap and extends
AbstractMap
3. TreeMap in Java does not allow null keys (like Map) and thus a NullPointerException is thrown. However, multiple null values can be associated with different keys.
4. All Map.Entry pairs returned by methods in this class and its views represent snapshots of mappings at the time they were produced. They do not support the Entry.setValue method.



Performance factors:

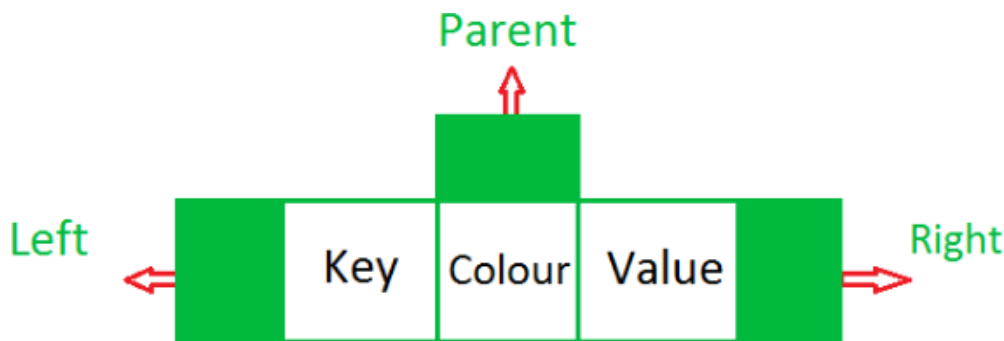
TreeMap is not synchronized and thus is not thread-safe. For multithreaded environments, accidental unsynchronized access to the map is prevented by:

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
```

Internal structure: The methods in TreeMap while getting keyset and values, return Iterator that are fail-fast in nature, thus any concurrent modification will throw ConcurrentModificationException.

TreeMap is based upon tree data structure. Each node in the tree has,

- 3 Variables (*K* key=Key, *V* value=Value, *boolean* color=Color)
- 3 References (*Entry* left = Left, *Entry* right = Right, *Entry* parent = Parent)



Constructors in TreeMap:

- 1 **TreeMap()** : Constructs an empty tree map that will be sorted by using the natural order of its keys.
- 2 **TreeMap(Comparator comp)** : Constructs an empty tree-based map that will be sorted by using the Comparator comp.
- 3 **TreeMap(Map m)** : Initializes a tree map with the entries from m, which will be sorted by using the natural order of the keys.
- 4 **TreeMap(SortedMap sm)** : Initializes a tree map with the entries from sm, which will be sorted in the same order as sm

Time Complexity: The algorithmic implementation is adapted from those of Red-Black Tree in Introduction to Algorithms (Eastern Economy Edition)

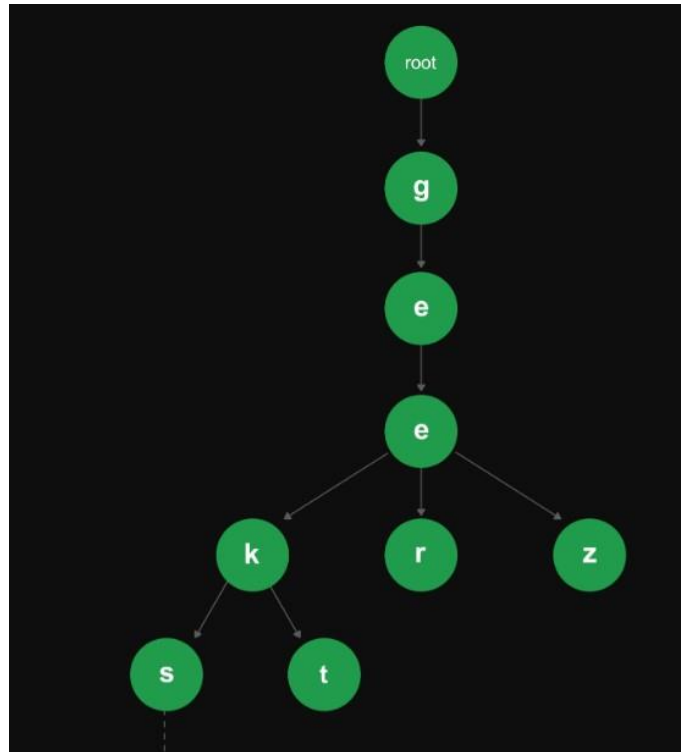
This provides guaranteed **log(n)** time cost for the **containsKey**, **get**, **put** and **remove** operations.

Methods of TreeMap:

- 1 **boolean containsKey(Object key):** Returns true if this map contains a mapping for the specified key.
- 2 **boolean containsValue(Object value):** Returns true if this map maps one or more keys to the specified value.
- 3 **Object firstKey():** Returns the first (lowest) key currently in this sorted map.
- 4 **Object get(Object key):** Returns the value to which this map maps the specified key.
- 5 **Object lastKey():** Returns the last (highest) key currently in this sorted map.
- 6 **Object remove(Object key):** Removes the mapping for this key from this TreeMap if present.
- 7 **void putAll(Map map):** Copies all of the mappings from the specified map to this map.
- 8 **Set entrySet():** Returns a set view of the mappings contained in this map.
- 9 **int size():** Returns the number of key-value mappings in this map.
- 10 **Collection values():** Returns a collection view of the values contained in this map.
- 11 **Object clone():** The method returns a shallow copy of this TreeMap.
- 12 **void clear():** The method removes all mappings from this TreeMap and clears the map.
- 13 **SortedMap headMap(Object key_value):** The method returns a view of the portion of the map strictly less than the parameter key_value.
- 14 **Set keySet():** The method returns a Set view of the keys contained in the treemap.
- 15 **Object put(Object key, Object value):** The method is used to insert a mapping into a map
- 16 **SortedMap subMap((K startKey, K endKey):** The method returns the portion of this map whose keys range from startKey, inclusive, to endKey, exclusive.
- 17 **Object firstKey():** The method returns the first key currently in this tree map.

Trie | (Insert and Search)

Trie is an efficient information re~~trieval~~**trieval** data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to $M * \log N$, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in $O(M)$ time. However the penalty is on Trie storage requirements



Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node. A Trie node field *isEndOfWord* is used to distinguish the node as end of word node. A simple structure to represent nodes of the English alphabet can be as following,

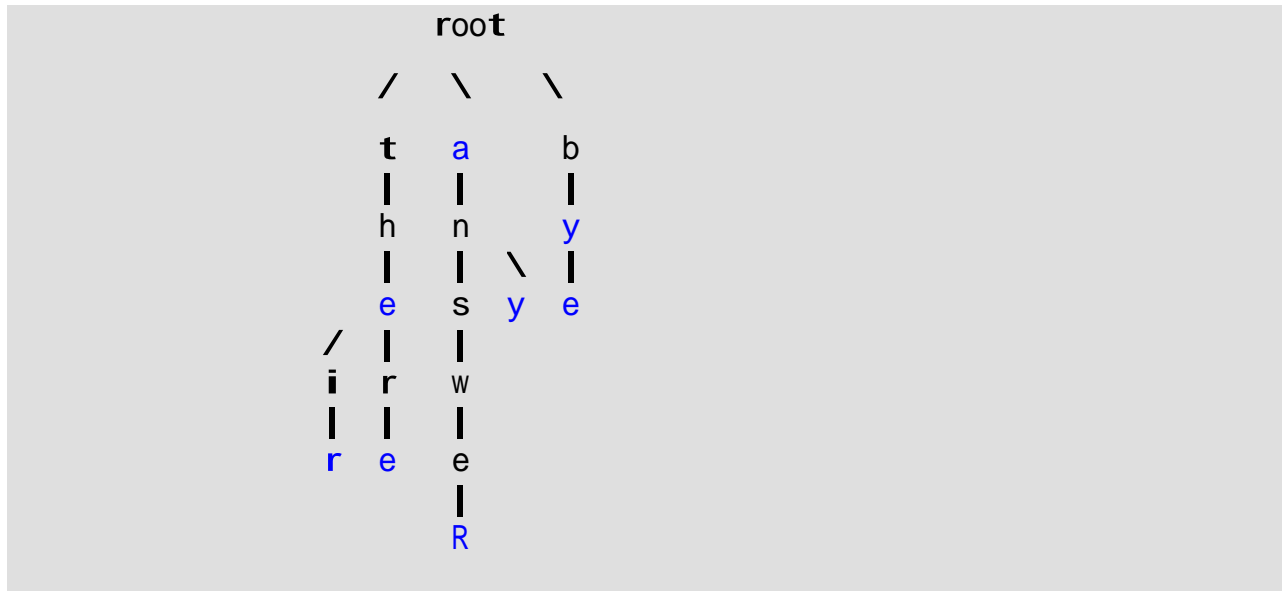
```

// Trie node
struct
TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];
    // isEndOfWord is true if the node
    // represents end of a
    word bool isEndOfWord;
};
  
```

Inserting a key into Trie is a simple approach. Every character of the input key is inserted as an individual Trie node. Note that the *children* is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

Searching for a key is similar to insert operation, however, we only compare the characters and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the *isEndofWord* field of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.

The following picture explains construction of trie using keys given in the example below,



In the picture, every character is of type *trie_node_t*. For example, the *root* is of type *trie_node_t*, and its children *a*, *b* and *t* are filled, all other nodes of *root* will be NULL. Similarly, “a” at the next level is having only one child (“n”), all other children are NULL. The leaf nodes are in blue.

```
// Java implementation of search and insert operations
// on Trie
public class Trie {

    // Alphabet size (# of symbols)
    static final int ALPHABET_SIZE = 26;

    // trie node
    static class TrieNode
    {
        TrieNode[] children = new TrieNode[ALPHABET_SIZE];

        // isEndOfWord is true if the node represents
        // end of a word
        boolean isEndOfWord;

        TrieNode(){
            isEndOfWord = false;
            for (int i = 0; i < ALPHABET_SIZE; i++)
                children[i] = null;
        }
    }
}
```

```
    }
};

static TrieNode root;

// If not present, inserts key into trie
// If the key is prefix of trie node,
// just marks leaf node
static void insert(String key)
{
    int level;
    int length = key.length();
    int index;

    TrieNode pCrawl = root;

    for (level = 0; level < length; level++)
    {
        index = key.charAt(level) - 'a';
        if (pCrawl.children[index] == null)
            pCrawl.children[index] = new TrieNode();

        pCrawl = pCrawl.children[index];
    }

    // mark last node as leaf
    pCrawl.isEndOfWord = true;
}

// Returns true if key presents in trie, else false
static boolean search(String key)
{
    int level;
    int length = key.length();
    int index;
    TrieNode pCrawl = root;

    for (level = 0; level < length; level++)
    {
        index = key.charAt(level) - 'a';

        if (pCrawl.children[index] == null)
            return false;

        pCrawl = pCrawl.children[index];
    }

    return (pCrawl != null && pCrawl.isEndOfWord);
}

// Driver
public static void main(String args[])
{
    // Input keys (use only 'a' through 'z' and lower case)
```



```
String keys[] = {"the", "a", "there", "answer", "any",  
                "by", "bye", "their"};  
  
String output[] = {"Not present in trie", "Present in trie"};  
  
root = new TrieNode();  
  
// Construct trie  
int i;  
for (i = 0; i < keys.length ; i++)  
    insert(keys[i]);  
  
// Search for different keys  
if(search("the") == true)  
    System.out.println("the --- " + output[1]);  
else System.out.println("the --- " + output[0]);  
  
if(search("these") == true)  
    System.out.println("these --- " + output[1]);  
else System.out.println("these --- " + output[0]);  
  
if(search("their") == true)  
    System.out.println("their --- " + output[1]);  
else System.out.println("their --- " + output[0]);  
  
if(search("thaw") == true)  
    System.out.println("thaw --- " + output[1]);  
else System.out.println("thaw --- " + output[0]);  
  
}  
// This code is contributed by Sumit Ghosh
```

output

```
the --- Present in trie  
these --- Not present in trie  
their --- Present in trie  
thaw --- Not present in trie
```

Trie | (Delete)

In the [previous post](#) on [trie](#) we have described how to insert and search a node in trie. Here is an algorithm how to delete a node from trie.

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

Balanced search trees

A data-structure is a method for storing data so that operations you care about can be performed quickly. Data structures are typically used as part of some larger algorithm or system, and good data structures are often crucial when especially fast performance is needed. We will be focusing in particular on what are called dictionary data structures, that support insert and lookup operations (and usually delete as well). Specifically, Definition 8.1 A dictionary data structure is a data structure supporting the following operations:

1. **insert(key, object):** insert the (key, object) pair. For instance, this could be a word and its definition, a name and phone number, etc. The key is what will be used to access the object.
2. **lookup(key):** return the associated object.
3. **delete(key):** remove the key and its object from the data structure. We may or may not care about this operation.

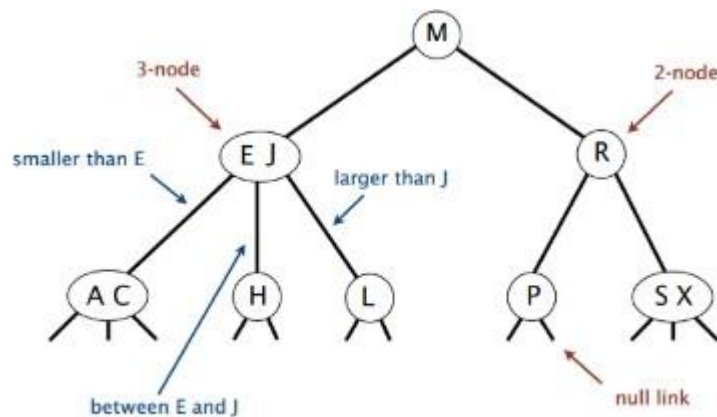
A **balanced binary search tree** is a tree that automatically keeps its height small (guaranteed to be logarithmic) for a sequence of insertions and deletions. This structure provide efficient implementations for abstract data structures such as associative arrays.

The primary step to get the flexibility that we need to guarantee balance in binary search trees is to allow the nodes in our trees to hold more than one key. This can be done using **2–3 search trees** (not binary, but balanced).

2–3 Search Trees

The 2–3 tree is a way to generalize BSTs to provide the flexibility that we need to guarantee fast performance. It allows 1 or 2 keys per node. It allows for the possibility of a 3-node and 2-node.

- **2-node:** one key, two children; left is less, and right is greater than the key.
- **3-node:** two keys, three children; left is less, middle is between, and right is greater than the two keys.



A 2-3 search tree — algs4.cs.princeton.edu

Properties of 2-3 Trees

- **Perfect Balance:** Every path from the root to the null link has the same length.
- **Symmetric Order:** Every node is larger than all the nodes on the left subtree, smaller than the keys on the right subtree, and in case of 3-node, all nodes in the middle are between the two keys of the 3-node. So, we can traverse the nodes in ascending order; In-order traversal.

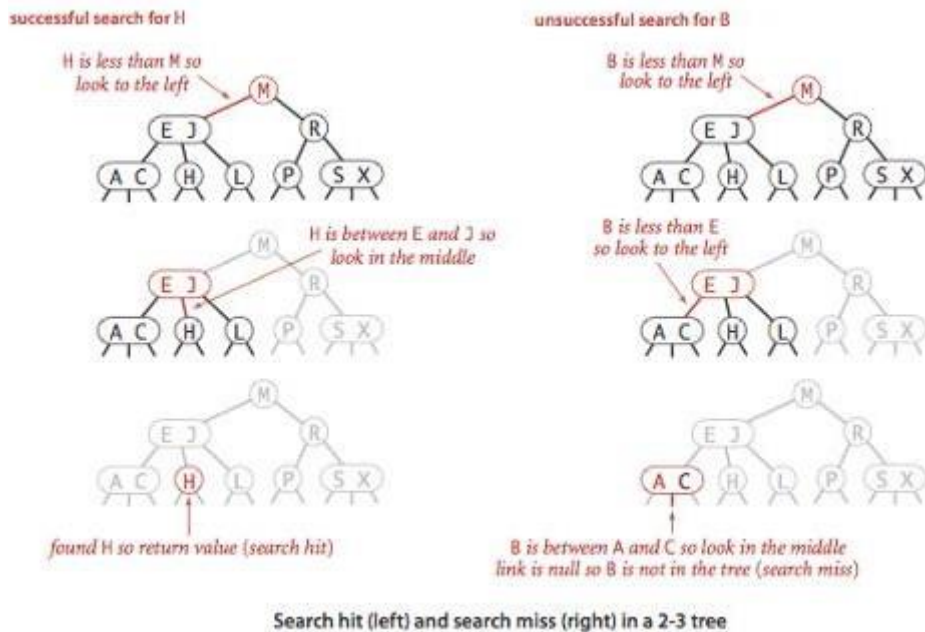
Operations Overview

We aren't going to discuss the implementation code, because it's complicated, rather, we will be giving an overview of two of the main operations of a 2-3 search tree. These operations are *search* and *insert*.

search

Searching for an item in a 2-3 tree is similar to searching for an item in a binary search tree since it maintains a symmetric order.

You compare between the given key against the key(s) in the node. If smaller than, go left. If between the two keys(of a 3-node), go to the middle link. If greater than, go right

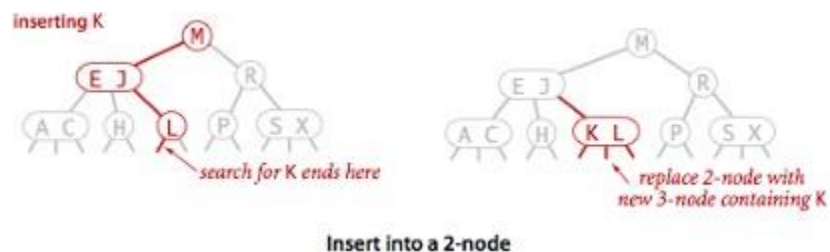


Insert (into a 2-node)

All insertion operations starts with searching for the node (at the bottom) where you can insert the new node into it.

If the node at which the search terminates is a 2-node, we just replace it with a 3-node containing its key and the new key to be inserted.

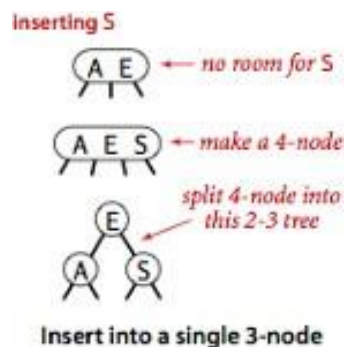
*In 2–3 search trees, we insert into a node, and **not** attaching a new node to a null link (like in BSTs), ... Why? To remain perfectly balanced.*



Insert (into a 3-node)

Suppose that we want to insert into a single 3-node. Such a node has no room for a new key. So, to be able to perform this insertion, we temporarily convert the 3-node into a 4-node (a node with three keys, and four children).

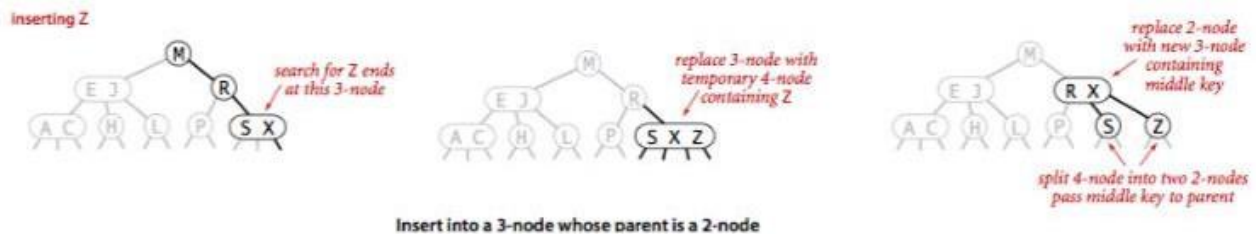
Then, we split the 4-node into three 2-nodes, one with the middle key (at the root), one with the smallest of the three keys (pointed to by the left link of the root), and one with the largest of the three keys (pointed to by the right link of the root).



Insert (into a 3-node whose parent is a 2-node)

Suppose that the search ends at a 3-node at the bottom whose parent is a 2-node.

In this case, we follow the same steps as just described, by making a temporary 4-node, then splitting the 4-node, but then, instead of creating a new node to hold the middle key, we move the middle key to the parent node (2-node).

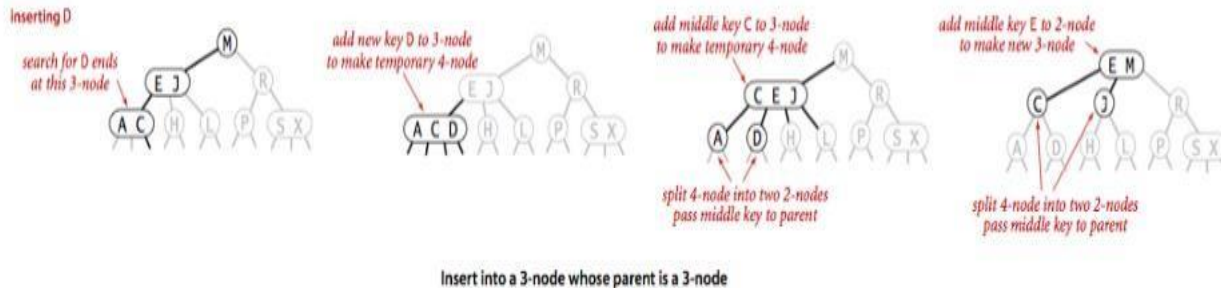


Insert (into a 3-node whose parent is a 3-node)

Now suppose that the search ends at a 3-node at the bottom whose parent is a 3-node.

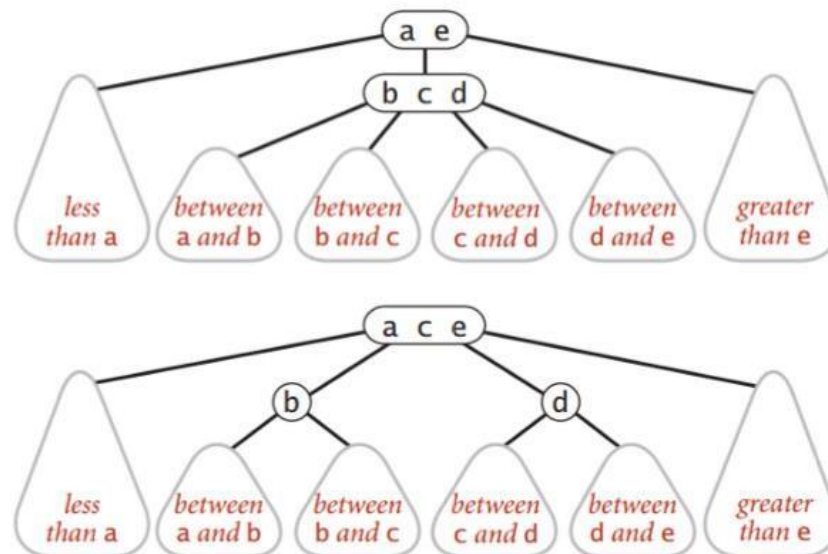
Again, we make a temporary 4-node, then split the 4-node, moving the middle key to the parent node (3-node). Since the parent node is a 3-node, we convert it into a temporary new 4- node. Then, we perform exactly the same transformation on that node.

We continue doing this transformation as we go up the tree; splitting 4-nodes and moving the middle keys to their parents until reaching a 2-node, which we replace it with a 3-node that does not to be further split, or until reaching a 3-node at the root.



Transformation In 2–3 Tress

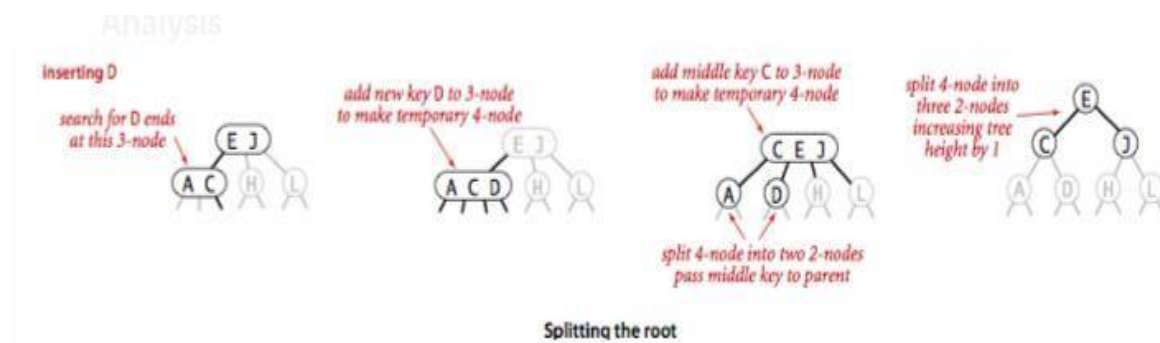
Only constant number of operations needed to do transformation of splitting a 4-node, and also converting 3-node to 4-node and so on.



These transformations preserve the properties of a 2–3 tree that the tree is in a symmetric order and perfectly balanced.

This is because, when we insert or move keys around, we keep the keys in order; we maintain a symmetric order.

And, we increase the height of the tree when we end up with a temporary 4-node at the root. In this case we split the temporary 4-node into three 2-nodes. So, we can still split the root node (4-node) while maintaining perfect balance in the tree.



Analysis


The cost of these operations is proportional to the height of the tree.

Since it maintains a perfect black balance tree. It guarantees performance of $O(\log N)$ in all operations.

- The **worst** case when all the nodes are 2-nodes; tree height is $\log N$.
- The **best** case when all the nodes are 3-nodes; tree height is $\log N$ (to the base of 3).

Here is a summary, for symbol table implementations after introducing the 2–3 search trees.

| implementation | guarantee | | | average case | | | ordered ops? | key interface |
|---------------------------------------|-----------|-----------|-----------|-----------------|-----------------|-----------------|--------------|---------------|
| | search | insert | delete | search hit | insert | delete | | |
| sequential search (unordered list) | N | N | N | $\frac{1}{2} N$ | N | $\frac{1}{2} N$ | | equals() |
| binary search (ordered array) | $\lg N$ | N | N | $\lg N$ | $\frac{1}{2} N$ | $\frac{1}{2} N$ | ✓ | compareTo() |
| BST | N | N | N | $1.39 \lg N$ | $1.39 \lg N$ | \sqrt{N} | ✓ | compareTo() |
| 2-3 tree | $c \lg N$ | $c \lg N$ | $c \lg N$ | $c \lg N$ | $c \lg N$ | $c \lg N$ | ✓ | compareTo() |



 constant c depend upon implementation

Constant c depend upon Implementation