202 – Datastructures Using Java

(MCA – S.V. University, Tirupati)

II – SEMESTER

STUDY MATERIAL

Prepared by: Mrs.P.Lalitha, MCA,
Dept. of Computer Science,
RCR B - SCHOOL

RCR
INSTITUTE OF MANAGEMENT& TECHNOLOGY
#1, RCRAveune, Karakambadi Road, Tirupati – 517520.

UNIT-1

Linear Data Structures: Abstract data Types-Asymptotic Notations: Big-Oh, Omega and Theta-Best, Worst and Average case Analysis: Definition and example-Arrays and its representations-Stacks and Queues-Linked lists-Linked list based implementation of Stacks and Queues – Evaluation of Expressions – Linked list based polynomial addition.

UNIT-II

Non-Linear Data Structures; Trees-Binary Trees - Binary tree representation and traversals-Threaded binary trees – Binary tree representation of trees-Application of trees: Set representation and Union-Find operations – Graph and its representations-Graph Traversals DFS and BFS-Connected components, Applications of Graphs-Minimum cost spanning tree using Kruskal's algorithm, single Source Shortest Path Problem.

UNIT-III

Search Structures And Priority Queues: AVLTrees-Red-BlackTrees-SplayTrees-BinaryHeap-Leftist Heap-Implementation of priority Queue ADT with Heap

UNIT-IV

Sorting: Insertion sort-Merge sort-Quick sort-Heap sort-Radix Sort- Comparison of sorting algorithms In terms of Complexity-Sorting with disks-k-way merging-Sorting with tapes-Polyphase merge.

UNIT-V

SearchingAndIndexing:LinearSearch-BinarySearch.-Hashtables-Overflowhandling-CylinderSurfaceIndexing-HashIndex-B-TreeIndexing,B+Trees.

Textbook:

1. SartajSahni, DataStrutures, Algorithms and Applications in Java, Second Edition, University Press..
2. GregoryL.Heilman, Datastructures, Algorithms and Object Oriented Programming, Tata Mcgraw-Hill, New Delhi; 2002.

References:

I.Jean-PaulTremblayandPaulG..Sorenson, An Introduction to Data structures with Applications, Second Edition, Tata Mc Graw-Hill , New Delhi, 1991.
2. AlfredV.Aho, John E.HopcroftandJeffryD.Ullman, DataStructuresandAlgoritlu11S, PearsonEducation, NewDelhi, 2006

Basic Concepts: Introduction to Data Structures:

A data structure is a way of storing data in a computer so that it can be used efficiently and it will allow the most efficient algorithm to be used. The choice of the data structure begins from the choice of an abstract data type (ADT). A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structure introduction refers to a scheme for organizing data, or in other words it is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations.

A data structure should be seen as a logical concept that must address two fundamental concerns.

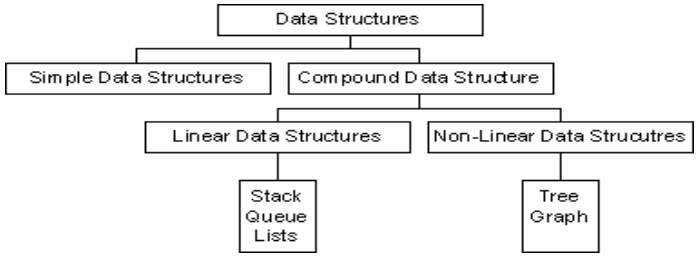
1. First, how the data will be stored, and
2. Second, what operations will be performed on it.

As data structure is a scheme for data organization so the functional definition of a data structure should be independent of its implementation. The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation. The way in which the data is organized affects the performance of a program for different tasks. Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data. Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees,and graphs.

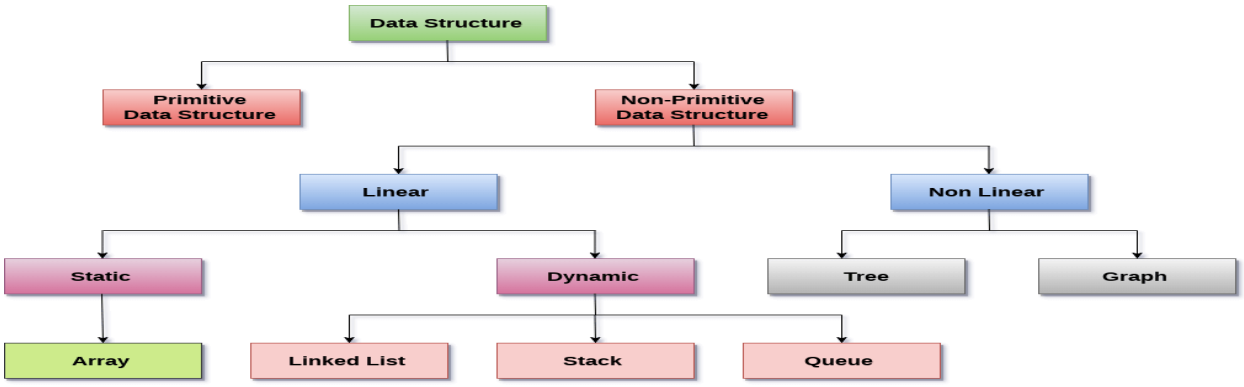
Classification of Data Structures:

Data structures can be classified as

- Simple data structure
- Compound data structure
- Linear data structure
- Non linear data structure



[Fig 1.1 Classification of Data Structures]



Simple Data Structure:

Simple data structure can be constructed with the help of primitive data structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, arrays, pointers, structures, unions, etc. are examples of

primitive data structures.

Compound Data structure:

Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as

- Linear data structure
- Non-linear data structure

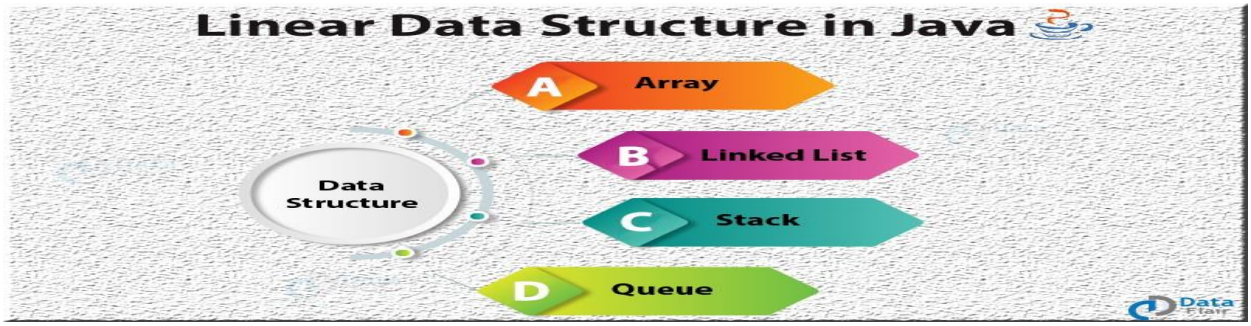
UNIT-1

➢ Linear Data structures

Linear data structures in Java is a way to organize the data in the language in a particular way so to use them in the most effective way. The main reason to classify them is that we need less complexity and less space.

There are 4 types of Java linear data structures

The main linear data structures in Java are:



1. Java Array

An array used to store data of the type homogenous at a contiguous place, size for the array is to define beforehand.

- Accessing Time: O(1) [This is possible because it stores elements at contiguous locations]
- Search Time: O(n) for Sequential Search: O(log n) for Binary Search [If Array is sorted]
- Insertion Time: O(n) [The worst case occurs when insertion happens at the beginning of an array and requires shifting all of the elements]
- Deletion Time: O(n) [This is the worst case occurs when deletion occurs at the starting of an array and requires shifting all of the elements]

Java Array Example

For storing the marks of a student we can create an array as this saves us from using a different variable for every subject, we can simply add data by traversing the array.

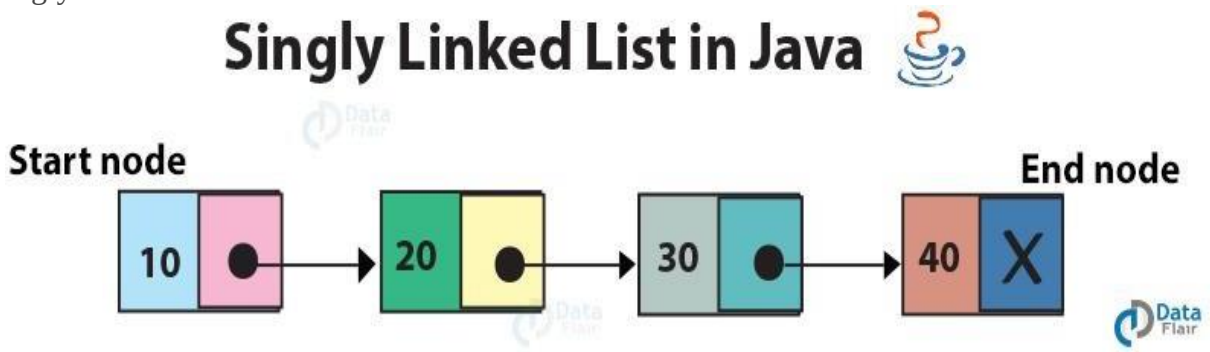
2. Java Linked List

A linked list is another important linear data structures in Java, is similar to an array with the only difference that here every element (we call it a 'node') is treated as a separate object, every node has two parts, one the data and the reference to the next node.

Types of Linked List in Java-

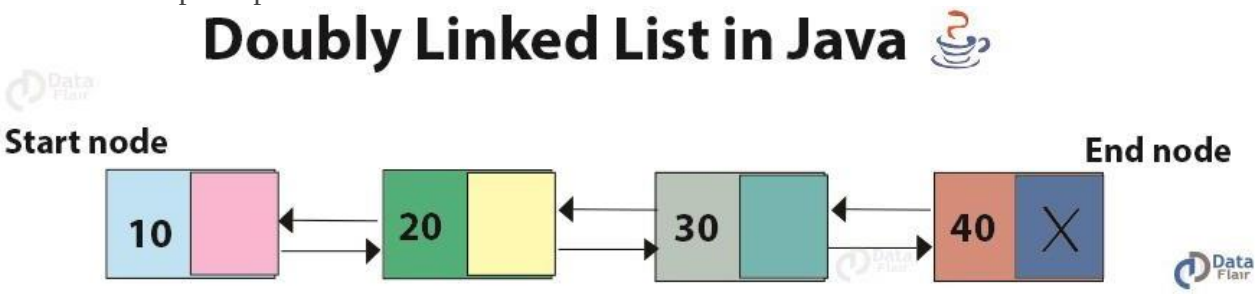
Singly linked list

Singly linked list stores data and the reference to the next node or a null value.



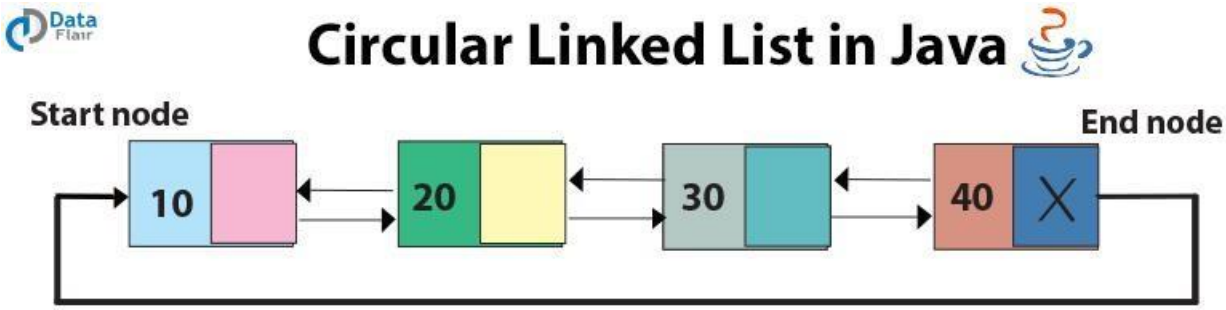
Doubly linked list

It is the same as a double linked list with the difference that it has two references, one for the last node and one for the previous node. This helps us traverse in both the directions and also we don't need explicit permission for deletion of nodes.



Circular Linked List

In this Java Linked List, all the nodes align to form a circle and also there is no NULL at the end, it helps us to define any node as first and also helps to implement a circular queue.



- Accessing time of an element: O(n)
- Search time of an element: O(n)

- Insertion of an Element: O(1) [If we are at the position where we have to insert an element]
- Deletion of an Element: O(1) [If we know the address of node previous the node to be deleted]

Example –

Consider the last example (Circular Linked List) where we need to add the marks of the students and a new subject is added to the list, we cannot add this to array as the size is already fixed and if we make an array which is larger than there will be empty spaces left, so to overcome this drawback we use linked lists.

- The only drawback is that we cannot randomly access the nodes which were quite easy in arrays.

3. Java Stack

It is one of the best linear data structures in Java, runs on the principle of **Last First Out In (LIFO)**. It allows the user to have two operations, viz., Push and Pop. Push allows us to add elements while Pop allows removing the last element. Both operations can take place at the same end.

- Insertion: O(1)
- Deletion: O(1)
- Access Time: O(n) [Worst Case]
- It allows Insertion and Deletion on only one end.

Example – Stacks in Java are utilized for keeping up work calls (the last called work must complete execution first), we can simply evacuate recursion with the assistance of stacks. Stack data Structures in Java is additionally utilized as a part of situations .

4. Java Queue

A Queue is a last linear data structures in Java, offers the option **First In, First Out (FIFO)**, which helps us to save a collection of data, it is an abstract data type. It provides two major options enqueue, the way toward adding a component to the collection. The component is included from the backside and dequeue, the way toward expelling the principal component that was included. The component is expelled from the front side. It can actualize by utilizing both exhibit and connected rundown.

- Insertion: O(1)
- Deletion: O(1)
- Access Time: O(n) [Worst Case]

Example – Queue in Java as the name says is the information structure worked by the lines of transport stop or prepare where the individual who is remaining in the front of the queue(standing for a very long time) is the first to get the ticket. So any circumstance where assets are shared among various clients and served on first start things out server premise. Cases incorporate CPU planning, Disk Scheduling. Another utilization of queue is when information is exchanged non-concurrently (information did not really get at the same rate as sent) between two procedures. Illustrations incorporate IO Buffers, channels, document IO, and so on.

Circular Queue

The upside of this Linear data structures in Java is that it diminishes wastage of room in the event of cluster execution, as the inclusion of the (n+1)'th component is done at the 0'th record on the off chance that it is vacant.

Operations applied on linear data structure:

The following list of operations applied on linear data structures

1. Add an element
2. Delete an element

3. Traverse
4. Sort the list of elements
5. Search for a data element

For example Stack Queue, Tables, List, and Linked Lists.

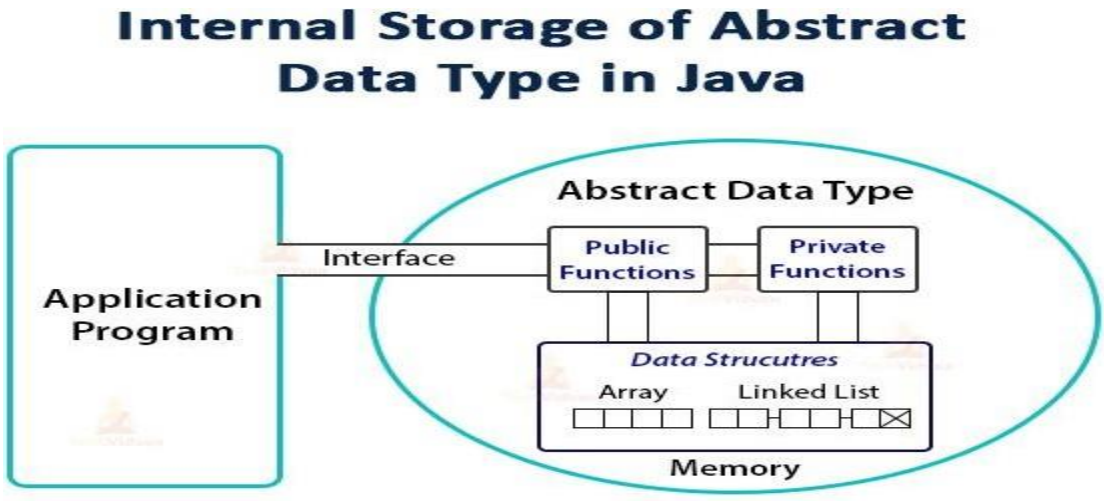
➢ Abstract Data Type

We know that a data type signifies the type and space taken by the data used in programs. An Abstract Data Type is a special data type that is defined by a set of values and a set of operations on that type.

We call these data types as **"abstract"** because these are independent of any implementation. We can use these data types and perform different operations with them, but we do not know how these operations are working internally.

The implementation details of these data types are totally invisible to the users. It does not specify how the data stores in the memory area and what algorithms are useful for implementing the operations on the data.

An Abstract Data Type in Java is useful in the implementation of data structures. Java library provides various Abstract Data Types such as List, Stack, Queue, Set, Map as inbuilt interfaces that we implement using various data structures.



Types and Operations of Java Abstract Data Type

Types

We can classify the Abstract data types either as **built-in** or **user-defined** or as **mutable** or **immutable**.

If an Abstract Data Type is mutable then we can change the objects of its type and if it is immutable then we can't change its object.

For example, the Date class is mutable because we can call its setMonth() method and observe the change with the getMonth() operation. But String is immutable because its operations do not change the existing objects but create new String objects

Operations

There are following types of operations of an abstract type:

- **Creators:** Creators create new objects of the type. It may take an object as an argument.

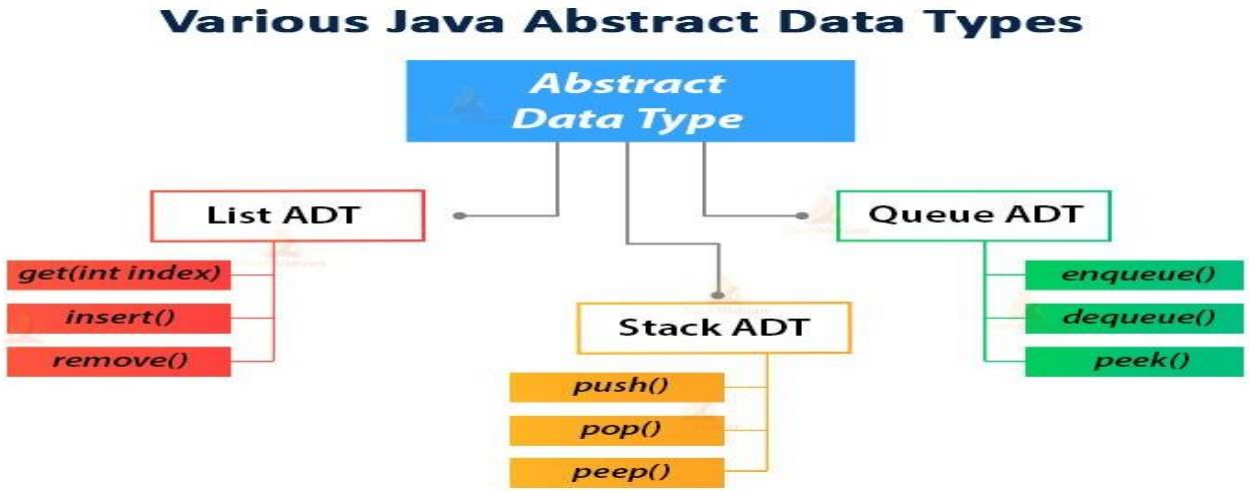
- **Producers:** Producers create new objects from old objects of the type. For example, the concat() method of the String is a producer that takes two strings and produces a new String representing their concatenation.
- **Observers:** Observers take the objects of the abstract type and return objects of a different type. For example, the size() method of the **List** returns an **int**.
- **Mutators:** Mutators change objects. For example, the add() method of List changes a list by adding an element to the end.

Java Abstract Data Type Examples

Below are some examples of abstract data types, along with some of their operations and the types.

1. **int** is a primitive integer type of Java. int is **immutable**, so it has no mutators. Its operations are:
   - **creators:** The numeric literals 0, 1, 2, 3,…
   - **producers:** Arithmetic operators +, -, ×, ÷
   - **observers:** Comparison operators ==, !=, <, >
   - **mutators:** None (it's immutable)
2. The **list** is an interface of Java List. The list is **mutable**. Its operations are:
   - **creators:** ArrayList and LinkedList constructors, Collections.singletonList
   - **producers:** Collections.unmodifiableList
   - **observers:** size, get
   - **mutators:** add, remove, addAll, Collections.sort
3. A **string** is Java's string type. The string is **immutable**. Its operations are:
   - **creators:** String constructors
   - **producers:** concat, substring, toUpperCase
   - **observers:** length, charAt
   - **mutators:** none (it's immutable)

List of Java Abstract Data Type



Now, Let's start exploring different Java Abstract Data Types in Java:

1. List ADT

The List Abstract Data Type is a type of list that contains similar elements in sequential order. The list ADT is a collection of elements that have a linear relationship with each other. A linear relationship means that each element of the list has a unique successor.

The List ADT is an interface, that is, other classes give the actual implementation of the data type. For example, Array Data Structure internally implements the **ArrayList** class while the List Data Structure internally implements the **LinkedList** class.

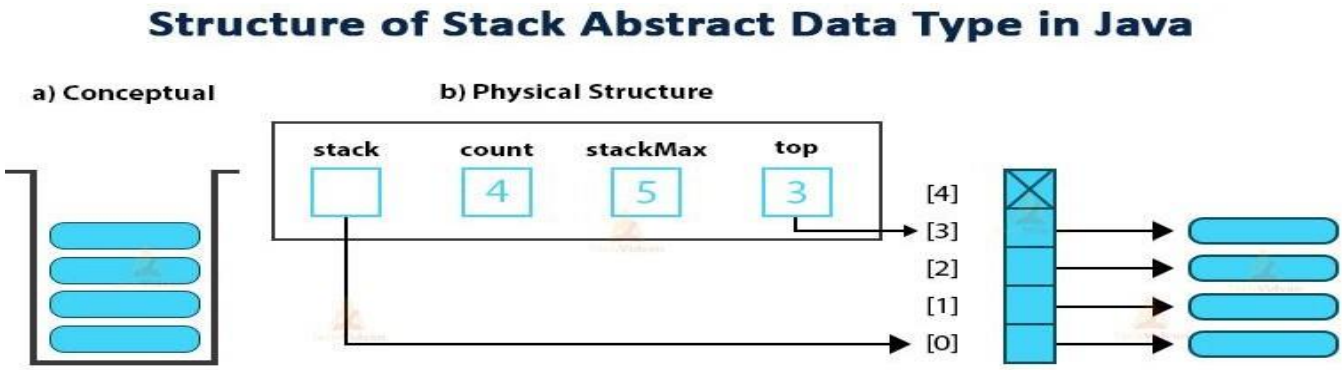
- **get(int index):** Returns an element at the specified index from the list.
- **insert():** Inserts an element at any position.
- **remove():** Removes the first occurrence of any element from a list.
- **removeAt():** Removes the element at a predefined area from a non-empty list.
- **Replace():** Replaces an element by another element.
- **size():** Returns the number of elements of the list.
- **isEmpty():** Returns true if the list is empty, else returns false.
- **isFull():** Returns true if the list is full, else returns false.

2. Stack ADT

A stack is a LIFO ("Last In, First Out") data structure that contains similar elements arranged in an ordered sequence. All the operations in stack take place at the **top** of the stack.

- **Stack ADT** is a collection of homogeneous data items (elements), in which all insertions and deletions occur at one end, called the top of the stack.
- In Stack ADT Implementation, there is a pointer to the data, instead of storing the data at each node.
- The program allocates the memory for the data and passes the address to the stack ADT.
- The start node and the data nodes encapsulate together in the ADT. Only the pointer to the stack is visible to the calling function.
- The stack head structure also contains a pointer to the top of the stack and also the count of the number of entries currently in the stack.

The below diagram shows the whole structure of the Stack ADT:



We can perform the following operations on the stack –

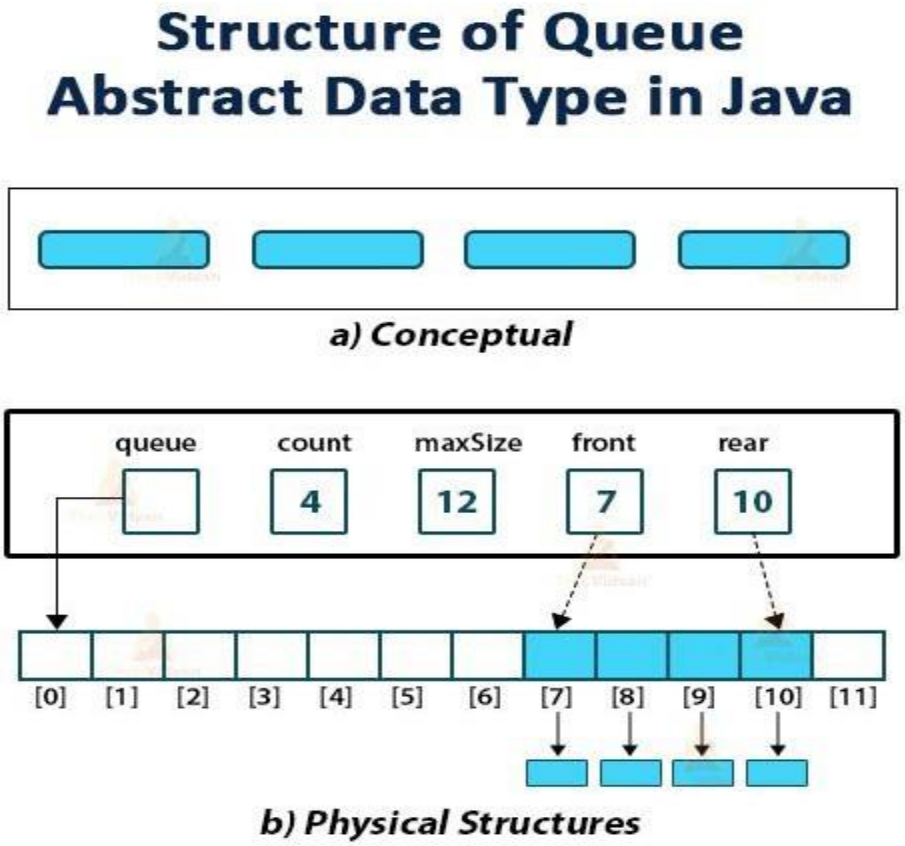
- **push():** It inserts an element at the top of the stack if the stack is not full.
- **pop():** It removes or pops an element from the top of the stack if the stack is not empty.
- **peep():** Returns the top element of the stack without removing it.
- **size():** Returns the size of the stack.
- **isEmpty():** If the stack is empty it returns true, else it returns false.
- **isFull():** If the stack is full it returns true, else it returns false.

3 Queue ADT.

A Queue is a FIFO ("First In, First Out") data structure that contains similar types of elements arranged sequentially. We can perform the operations on a queue at both ends; insertion takes place at rear end deletion takes place at the front end.

Queue ADT is a collection in which the arrangement of the elements of the same type is in a sequential manner.

- The design of the Queue abstract data type (ADT) is the same as the basic design of the Stack ADT.
- Each node of the queue contains a void pointer to the data and a link pointer to the next element of the queue. The program allocates the memory for storing the data.



Operations performed on the queue are as follows:

- **enqueue():** It inserts or adds an element at the end of the queue.
- **dequeue():** Removes an element from the front side of the queue.
- **peek():** Returns the starting element of the queue without removing it.
- **size():** This function returns the number of elements in the queue.
- **isEmpty():** If the queue is empty, it returns true, otherwise it returns false.
- **isFull():** If the queue is full, it returns true, otherwise it returns false.

Designing an Abstract Data Type in Java

To design an abstract data type we have to choose good operations and determine how they should behave. Here are a few rules for designing an ADT.

- It's better to combine simple and few operations in powerful ways, rather than a lot of complex operations.
- Each operation in an Abstract Data Type should have a clear purpose and should have a logical behavior rather than a range of special cases. All the special cases would make operation difficult to understand and use.
- The set of operations should be adequate so that there are enough kinds of computations that users likely want to do.

- The type may be either generic, for example, a graph, a list or a set, or it may be domain-specific for example, an employee database, a street map, a phone book, etc. But there should not be a combination of generic and domain-specific features.

Which Java Abstract Data Type to choose?

Now after having brief knowledge of Java Abstract Data Types, we will discuss the scenarios to choose between either of List, Stack or Queue ADT.

List ADT is a collection of elements and stores them sequentially and which we can access using their indices. We can opt for this ADT in cases that involve indexed or sequential access or removal of elements.
For example, we can use various implementations of List ADT to store data of a list of employees in sorted order for sequential access or removal.

A Stack is a Last In First out data structure, and thus we can use implementations of **Stack ADT** in scenarios where we need to firstly access the most recently inserted elements.
For example, the function call stack of every programming language has the common requirement of this kind of LIFO data structure where there is a need to execute the most recent function in the stack.

The queue is a First In First Out data structure and we can choose the **Queue ADT** in scenarios where we need to access the elements in their order of insertion.
For example, one such scenario is request handling by web servers. Web servers make it possible to ensure the fairness of request handling according to their order of arrival by maintaining an internal queue for the requests.

➢ Asymptotic Notations

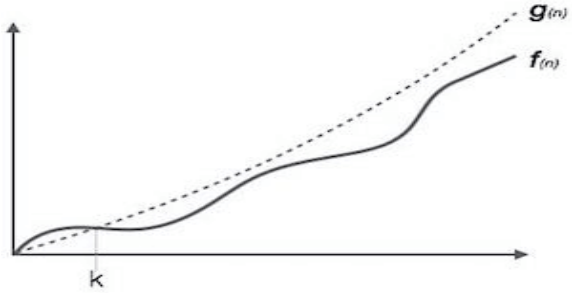
The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

- Big oh Notation (O)
- Omega Notation (Ω)
- Theta Notation (θ)

Big oh Notation (O)

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:



For example:
If **f(n)** and **g(n)** are the two functions defined for positive integers,
then **f(n)** = **O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)≤c.g(n) for all n≥no

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

Let's understand through examples

Example 1: f(n)=2n+3 , g(n)=n

Now, we have to find **Is f(n)=O(g(n))?**

To check f(n)=O(g(n)), it must satisfy the given condition:

f(n)<=c.g(n)

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2*1+3<=5*1

5<=5

For n=1, the above condition is true.

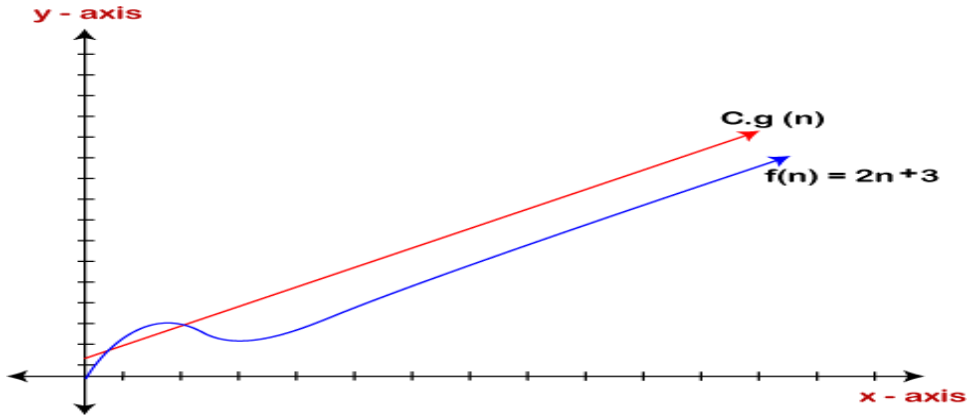
If n=2

2*2+3<=5*2

7<=10

For n=2, the above condition is true.

We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n). It can be represented graphically as:



The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

Omega Notation (Ω)

- It basically describes the best-case scenario which is opposite to the big o notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.

If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- Ω notation i.e. the Greek letter "omega". It is used to bound the growth of running time for large input size.

If **f(n)** and **g(n)** are the two functions defined for positive integers,

then **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)>=c.g(n) for all n≥no and c>0

Let's consider a simple example.

If f(n) = 2n+3, g(n) = n,

Is f(n)= **Ω** (g(n))?

It must satisfy the condition:

f(n)>=c.g(n)

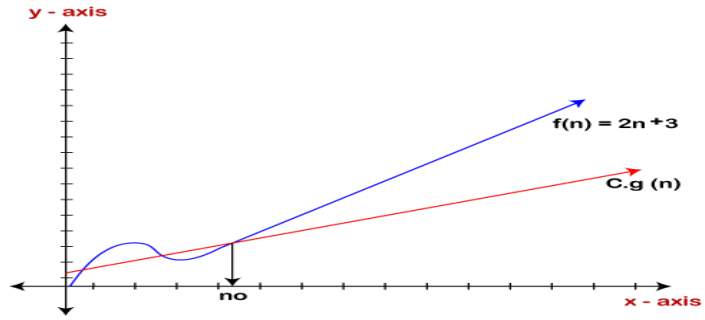
To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.

2n+3>=c*n

Suppose c=1

2n+3>=n (This equation will be true for any value of n starting from 1).

Therefore, it is proved that g(n) is big omega of 2n+3 function.



As we can see in the above figure that g(n) function is the lower bound of the f(n) function when the value of c is equal to 1. Therefore, this notation gives the fastest running time. But, we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process.

Theta Notation (θ)

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

Let's understand the big theta notation mathematically:

Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:

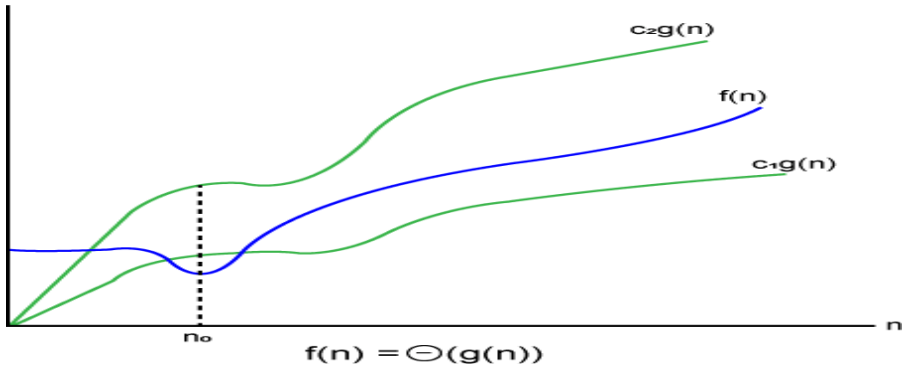
f(n)= θg(n)

The above condition is satisfied only if when

c1.g(n)<=f(n)<=c2.g(n)

where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between. The condition **f(n)= θg(n)** will be true if and only if c1.g(n) is less than or equal to

f(n) and c2.g(n) is greater than or equal to f(n). The graphical representation of theta notation is given below:



Let consider the same example where f(n)=2n+3 g(n)=n
As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5. The c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).
c1.g(n)<=f(n)<=c2.g(n)
Replace g(n) by n and f(n) by 2n+3
c1.n <=2n+3<=c2.n
if c1=1, c2=2, n=1
1*1 <=2*1+3 <=2*1
1 <= **5** <= **2** // for n=1, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)
If n=2
1*2<=2*2+3<=2*2
2<=7<=4 // for n=2,
Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n). So, this is the average-case scenario which provides the realistic time complexity.

➢ Best, Average and Worst case Analysis

Worst case: It defines the input for which the algorithm takes a huge time.
Average case: It takes average time for the program execution.
Best case: It defines the input for which the algorithm takes the lowest time
Introduction

We all know that the running time of an algorithm increases (or remains constant in case of constant running time) as the input size (nn) increases. Sometimes even if the size of the input is same, the running time varies among different instances of the input. In that case, we perform best, average and worst-case analysis. The best case gives the minimum time, the worst case running time gives the maximum time and average case running time gives the time required on average to execute the algorithm. I will explain all these concepts with the help of two examples - (i) Linear Search and (ii) Insertion sort.

Best Case Analysis

Consider the example of Linear Search where we search for an item in an array. If the item is in the array, we return the corresponding index, otherwise, we return -1. The code for linear search is given below.

```
1       int search(int a, int n, int item) {
2          int i;
3          for (i = 0; i < n; i++) {
4                return if (a[i] == item) {
```

```
5                return a[i]
6             }
7          }
8       }
```

Variable a is an array, n is the size of the array and item is the item we are looking for in the array. When the item we are looking for is in the very first position of the array, it will return the index immediately. The for loop runs only once. So the complexity, in this case, will be O(1)O(1). This is the called the best case.

Consider another example of insertion sort. Insertion sort sorts the items in the input array in an ascending (or descending) order. It maintains the sorted and un-sorted parts in an array. It takes the items from the un-sorted part and inserts into the sorted part in its appropriate position. The figure below shows one snapshot of the insertion operation.

In the figure, items [1, 4, 7, 11, 53] are already sorted and now we want to place 33 in its appropriate place. The item to be inserted are compared with the items from right to left one-by-one until we found an item that is smaller than the item we are trying to insert. We compare 33 with 53 since 53 is bigger we move one position to the left and compare 33 with 11. Since 11 is smaller than 33, we place 33 just after 11 and move 53 one step to the right. Here we did 2 comparisons. It the item was 55 instead of 33, we would have performed only one comparison. That means, if the array is already sorted then only one comparison is necessary to place each item to its appropriate place and one scan of the array would sort it. The code for insertion operation is given below.

```
1       void sort(int a, int n) {
2          int i, j;
3          for (i = 0; i < n; i++) {
4            j = i-1;
5            key = a[i];
6            while (j >= 0 && a[j] > key)
7            {
8               a[j+1] = a[j];
9               j = j-1;
10           }
11           a[j+1] = key;
12         }
13      }
```

When items are already sorted, then the while loop executes only once for each item. There are total n items, so the running time would be O(n)O(n). So the best case running time of insertion sort is O(n)O(n).

The best case gives us a lower bound on the running time for any input. If the best case of the algorithm is O(n)O(n) then we know that for any input the program needs at least O(n)O(n) time to run. In reality, we rarely need the best case for our algorithm. We never design an algorithm based on the best case scenario.

Worst Case Analysis

In real life, most of the time we do the worst case analysis of an algorithm. Worst case running time is the longest running time for any input of size nn.

In the linear search, the worst case happens when the item we are searching is in the last position of the array or the item is not in the array. In both the cases, we need to go through all nn items in the array. The worst case runtime is, therefore, O(n)O(n). Worst case performance is more important than the best case performance in case of linear search because of the following reasons.

1. The item we are searching is rarely in the first position. If the array has 1000 items from 1 to 1000. If we randomly search the item from 1 to 1000, there is 0.001 percent chance that the item will be in the first position.
2. Most of the time the item is not in the array (or database in general). In which case it takes the worst case running time to run.

Similarly, in insertion sort, the worst case scenario occurs when the items are reverse sorted. The number of comparisons in the worst case will be in the order of n2n2 and hence the running time is O(n2)O(n2).

Knowing the worst-case performance of an algorithm provides a guarantee that the algorithm will never take any time longer.

Average Case Analysis

Sometimes we do the average case analysis on algorithms. Most of the time the average case is roughly as bad as the worst case. In the case of insertion sort, when we try to insert a new item to its appropriate position, we compare the new item with half of the sorted item on average. The complexity is still in the order of n2n2 which is the worst-case running time.

It is usually harder to analyze the average behavior of an algorithm than to analyze its behavior in the worst case. This is because it may not be apparent what constitutes an "average" input for a particular problem. A useful analysis of the average behavior of an algorithm, therefore, requires a prior knowledge of the distribution of the input instances which is an unrealistic requirement. Therefore often we assume that all inputs of a given size are equally likely and do the probabilistic analysis for the average case.

➢ Arrays and its implementation

Definition

- Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in programming language which can store the primitive type of data such as int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different subject. instead of that, we can define an array which can store the marks in each subject at a the contiguous memory locations.

The array **marks[10]** defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. **marks[0]** denotes the marks in first subject, **marks[1]** denotes the marks in 2nd subject and so on.

Properties of the Array

1. Each element is of same data type and carries a same size i.e. int = 4 bytes.
2. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
3. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of data element.

Complexity of Array operations

Time and space complexity of various array operations are described in the following table.

Time Complexity

Algorithm	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Space Complexity

In array, space complexity for worst case is **O(n)**.

Advantages of Array

- Array provides the single name for the group of variables of the same type therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process, we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

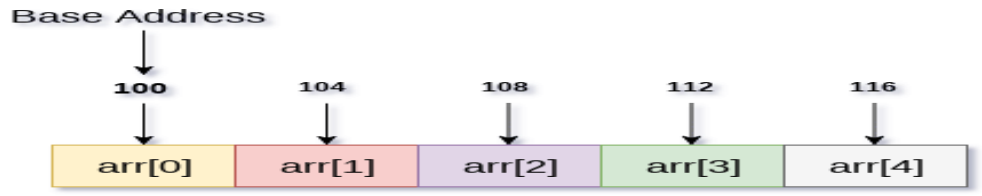
Memory Allocation of the array

As we have mentioned, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of first element in the main memory. Each element of the array is represented by a proper indexing.

The indexing of the array can be defined in three ways.

1. 0 (zero - based indexing) : The first element of the array will be arr[0].
2. 1 (one - based indexing) : The first element of the array will be arr[1].
3. n (n - based indexing) : The first element of the array can reside at any random index number.

In the following image, we have shown the memory allocation of an array arr of size 5. The array follows 0-based indexing approach. The base address of the array is 100th byte. This will be the address of arr[0]. Here, the size of int is 4 bytes therefore each element will take 4 bytes in the memory.



int arr[5]

The following program, ArrayDemo, creates an array of integers, puts some values in the array, and prints each value to standard output.

```
class ArrayDemo {
   public static void main(String[] args) {
      // declares an array of integers
      int[] anArray;
      // allocates memory for 10 integers
      anArray = new int[10];
          // initialize first element
```

```
        anArray[0] = 100;
        // initialize second element
        anArray[1] = 200;
        // and so forth
        anArray[2] = 300;
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;
        System.out.println("Element at index 0: "
                           + anArray[0]);
        System.out.println("Element at index 1: "
                           + anArray[1]);
        System.out.println("Element at index 2: "
                           + anArray[2]);
        System.out.println("Element at index 3: "
                           + anArray[3]);
        System.out.println("Element at index 4: "
                           + anArray[4]);
        System.out.println("Element at index 5: "
                           + anArray[5]);
        System.out.println("Element at index 6: "
                           + anArray[6]);
        System.out.println("Element at index 7: "
                           + anArray[7]);
        System.out.println("Element at index 8: "
                           + anArray[8]);
        System.out.println("Element at index 9: "
                           + anArray[9]);
    }
}
```

The output from this program is:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

In a real-world programming situation, you would probably use one of the supported looping constructs to iterate through each element of the array, rather than write each line individually as in the preceding example.

Declaring a Variable to Refer to an Array

The preceding program declares an array (named anArray) with the following line of code:

```
// declares an array of integers
int[] anArray;
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as type[], where type is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the naming section. As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type.

Similarly, you can declare arrays of other types:

```
byte[]  anArrayOfBytes;
short[] anArrayOfShorts;
long[]  anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

You can also place the brackets after the array's name:

```
// this form is discouraged
float anArrayOfFloats[];
```

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

Creating, Initializing, and Accessing an Array

One way to create an array is with the new operator. The next statement in the ArrayDemo program allocates an array with enough memory for 10 integer elements and assigns the array to the anArray variable.

```
// create an array of integers
anArray = new int[10];
```

If this statement is missing, then the compiler prints an error like the following, and compilation fails:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // and so forth
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

Here the length of the array is determined by the number of values provided between braces and separated by commas.
You can also declare an array of arrays (also known as a multidimensional array) by using two or more sets of brackets, such as String[][] names. Each element, therefore, must be accessed by a corresponding number of index values.
In the Java programming language, a multidimensional array is an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following MultiDimArrayDemo program:

```
class MultiDimArrayDemo {
   public static void main(String[] args) {
      String[][] names = {
         {"Mr. ", "Mrs. ", "Ms. "},
         {"Smith", "Jones"}
      };
      // Mr. Smith
      System.out.println(names[0][0] + names[1][0]);
      // Ms. Jones
      System.out.println(names[0][2] + names[1][1]);
   }
}
```

The output from this program is:

```
Mr. Smith
Ms. Jones
```

Finally, you can use the built-in length property to determine the size of any array. The following code prints the array's size to standard output:

```
System.out.println(anArray.length);
```

Copying Arrays

The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

The two Object arguments specify the array to copy from and the array to copy to. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, ArrayCopyDemo, declares an array of String elements. It uses the System.arraycopy method to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {
   public static void main(String[] args) {
      String[] copyFrom = {
         "Affogato", "Americano", "Cappuccino", "Corretto", "Cortado",
         "Doppio", "Espresso", "Frappucino", "Freddo", "Lungo", "Macchiato",
         "Marocchino", "Ristretto" };
       String[] copyTo = new String[7];
      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      for (String coffee : copyTo) {
         System.out.print(coffee + " ");
```

```
    }
  }
}
```

The output from this program is:

Cappuccino Corretto Cortado Doppio Espresso Frappucino Freddo

Array Manipulations

Arrays are a powerful and useful concept used in programming. Java SE provides methods to perform some of the most common manipulations related to arrays. For instance, the ArrayCopyDemo example uses the arraycopy method of the System class instead of manually iterating through the elements of the source array and placing each one into the destination array. This is performed behind the scenes, enabling the developer to use just one line of code to call the method.

```
class ArrayCopyOfDemo {
  public static void main(String[] args) {
    String[] copyFrom = {
      "Affogato", "Americano", "Cappuccino", "Corretto", "Cortado",
      "Doppio", "Espresso", "Frappucino", "Freddo", "Lungo", "Macchiato",
      "Marocchino", "Ristretto" };
       String[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);
    for (String coffee : copyTo) {
      System.out.print(coffee + " ");
    }
  }
}
```

➢ STACK

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack. This feature makes it LIFO data structure. LIFO stands for Last-in-first-out.

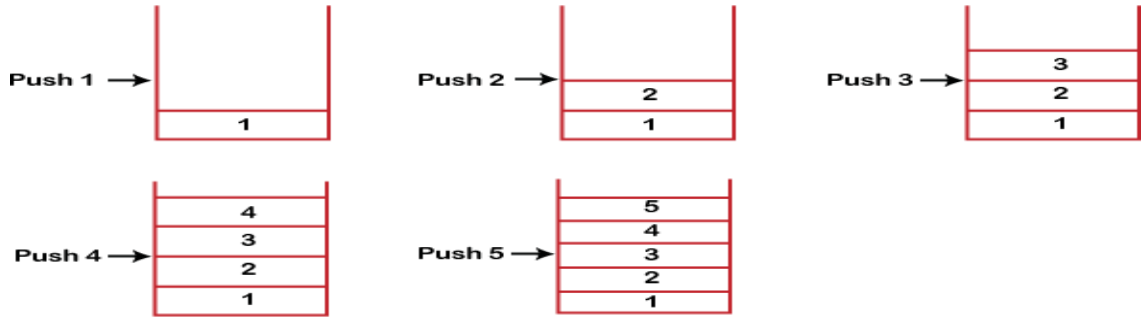
Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation

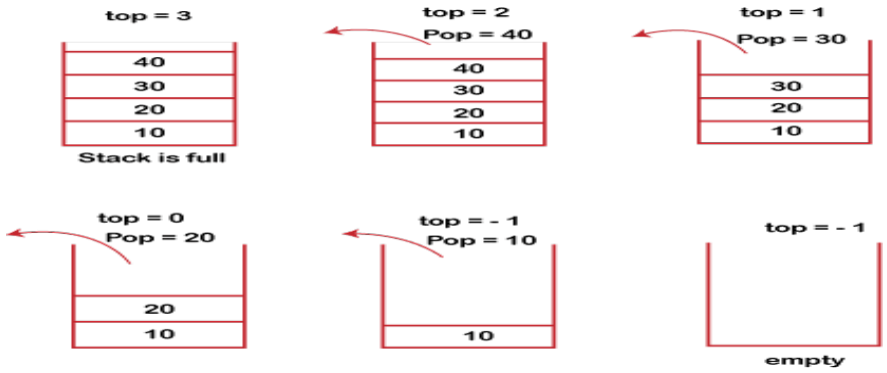
The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the **overflow** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the **max** size of the stack.

POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the **underflow** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the **top**
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Applications of Stack

The following are the applications of the stack:

- **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:

```
1. Public static void main()
2. {
3.    System.out.println("Hello");
4.    Sytem.out.println("javaTpoint");
5. }
```

As we know, each program has an opening and closing braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" string, so we can achieve this with the help of a stack. First, we push all the characters of the string in a stack until we reach the null character.

  After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **undo/redo:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the

text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.

- **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
- **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
- Infix to prefix
- Infix to postfix
- Prefix to infix
- Prefix to postfix
  Postfix to infix
- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

➢ QUEUE

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

Example

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull
   if rear equals to MAXSIZE
      return true
   else
      return false
   endif
end procedure
```

Implementation of isfull() function

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty
```

```
  if front is less than MIN  OR front is greater than rear
    return true
  else
    return false
  endif
  end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

```
bool isempty() {
  if(front < 0 || front > rear)
    return true;
  else
    return false;
}
```

➢ LINKEDLIST

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.

- **Linked List** − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

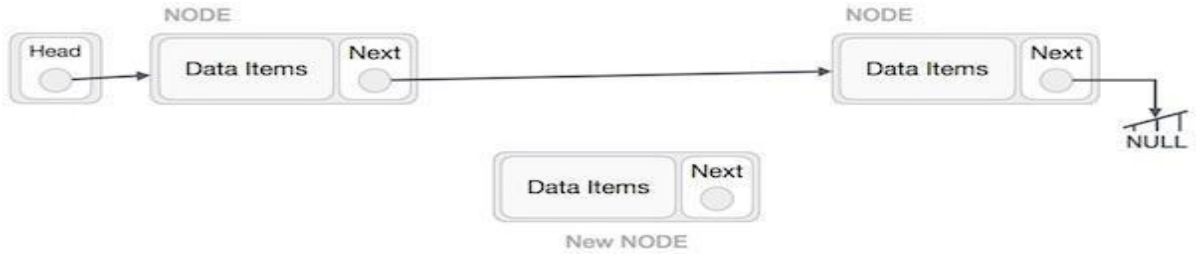
Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.

- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Insertion Operation

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

NewNode.next −> RightNode;

It should look like this –

Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;

This will put the new node in the middle of the two. The new list should look like this −

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

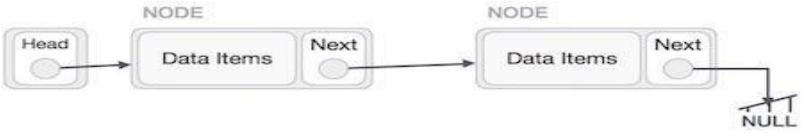
Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

The left (previous) node of the target node now should point to the next node of the target node LeftNode.next −> TargetNode.next;

This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

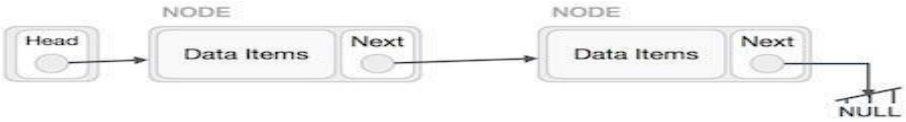
TargetNode.next −> NULL;

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



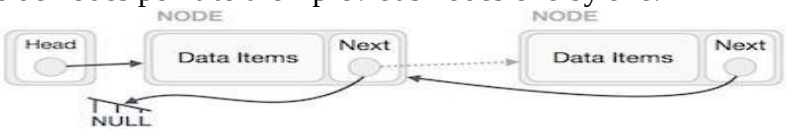
Reverse Operation

This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.

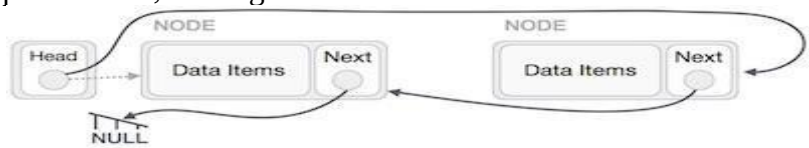


First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node –

We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



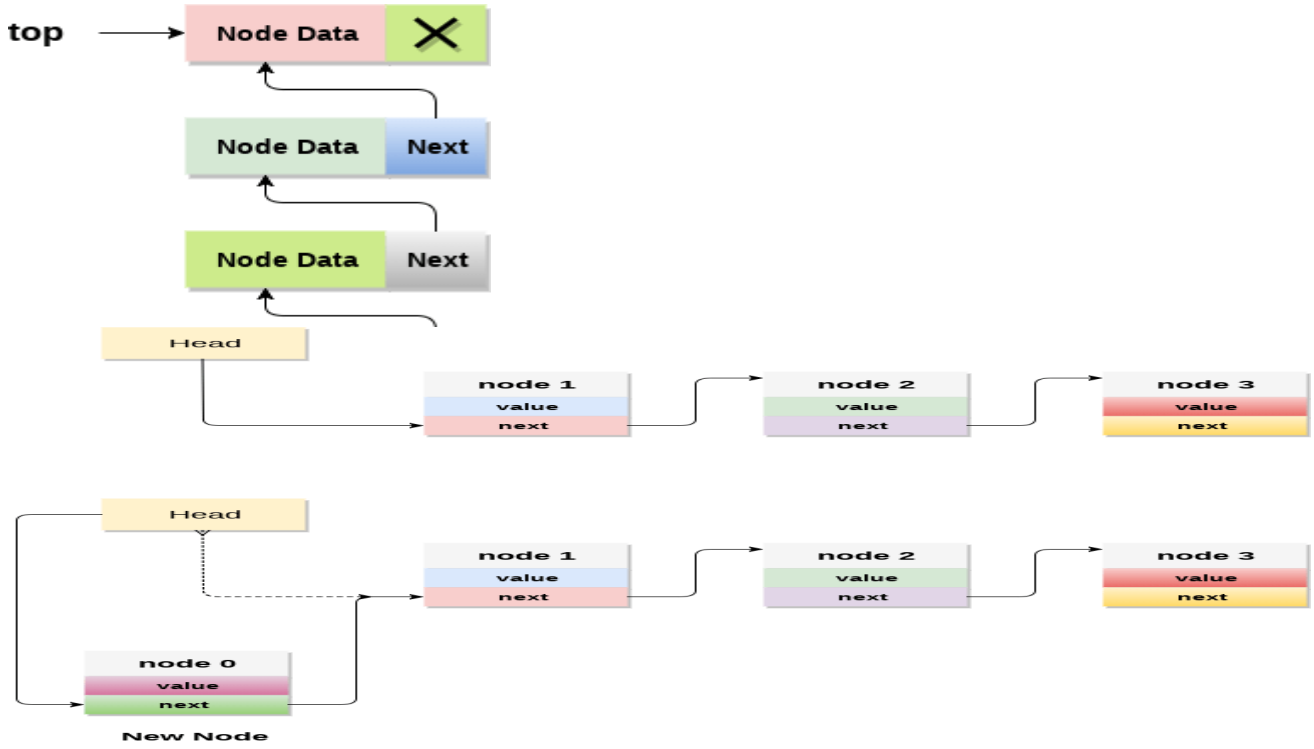
Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.

➢ Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory
complexity in both the scenario is same for all the operations i.e. push, pop and peek.
In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each n
immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is n



The
alwa
addr
way

performed in linked list implementation of stack.
Adding a node to the stack (Push operation)
Adding a node to the stack is referred to as push operation. Pushing an element to a stack in linked list imp
that of an array implementation. In order to push an element onto the stack, the following steps are involvedt
Create a node first and allocate memory to it.
If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to
assign null to the address part of the node.
If there are some nodes in the list already, then we have to add the new element in the beginning of the list (
the stack). For this purpose, assign the address of the starting element to the address field of the new node
starting node of the list.
Time Complexity : o(1)
implementation :

```
void push ()
{
   int val;
   struct node *ptr =(struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("not able to push the element");
   }
   else
   {
```

```
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
           ptr->val = val;
           ptr -> next = NULL;
           head=ptr;
        }
        else
        {
           ptr->val = val;
           ptr->next = head;
           head=ptr;

        }
        printf("Item pushed");

    }
}
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as pop operation. Deleting a node from the linked lis different from that in the array implementation. In order to pop an element from the stack, we need to follow

Check for the underflow condition: The underflow condition occurs when we try to pop from an already em empty if the head pointer of the list points to null.

Adjust the head pointer accordingly: In stack, the elements are popped only from one end, therefore, the val must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

C implementation

```
void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");

    }
}
```

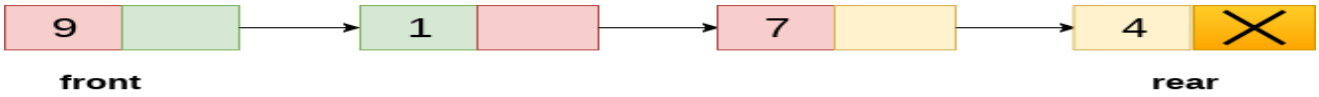
Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form o

need to follow the following steps.
Copy the head pointer into a temporary pointer.
Move the temporary pointer through all the nodes of the list and print the value field attached to every node.
Time Complexity : o(n)

➢ Linked List implementation of Queue

The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).
In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.
In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.riggers in SQL (Hindi)
Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.
The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue
There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.
Insert operation
The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.
Firstly, allocate the memory for the new node ptr by using the following statement.
Ptr = (struct node *) malloc (sizeof(struct node));
There can be the two scenario of inserting this new node ptr into the linked queue.
In the first scenario, we insert element into an empty queue. In this case, the condition front = NULL becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item;
      if(front == NULL)
      {
         front = ptr;
         rear = ptr;
         front -> next = NULL;
         rear -> next = NULL;
      }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node ptr. We also need to make the next pointer of rear point to NULL.

```
rear -> next = ptr;
         rear = ptr;
         rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

algorithm

step 1: allocate the space for the new node ptr

step 2: set ptr -> data = val

step 3: if front = null

set front = rear = ptr

set front -> next = rear -> next = null

else

set rear -> next = ptr

set rear = ptr

set rear -> next = null

[end of if]

Step 4: END

```
void insert(struct node *ptr, int item; )
{
      ptr = (struct node *) malloc (sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW\n");
      return;
   }
   else
   {
      ptr -> data = item;
      if(front == NULL)
      {
         front = ptr;
         rear = ptr;
         front -> next = NULL;
         rear -> next = NULL;
      }
      else
      {
         rear -> next = ptr;
         rear = ptr;
         rear->next = NULL;
      }
   }
}
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

ptr = front;

```
        front = front -> next;
        free(ptr);
```

The algorithm and function is given as follows.

Algorithm

Step 1: IF FRONT = NULL
Write " Underflow " Go to Step 5 [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END

```
void delete (struct node *ptr)
{
   if(front == NULL)
   {
      printf("\nUNDERFLOW\n");
      return;
   }
   else
   {
      ptr = front;
      front = front -> next;
      free(ptr);
   }
}
```

➢ Evaluation of Expressions

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations −

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr.No.	Operator	Precedence
1	Exponentiation ^	Highest
2	Multiplication (∗) & Division (/)	Second Highest
3	Addition (+) & Subtraction (−)	Lowest

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands are consumed

Step 6 – pop the stack and perform operation

➢ Linked list based polynomial addition

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots. + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

one is the coefficient

other is the exponent

Example:

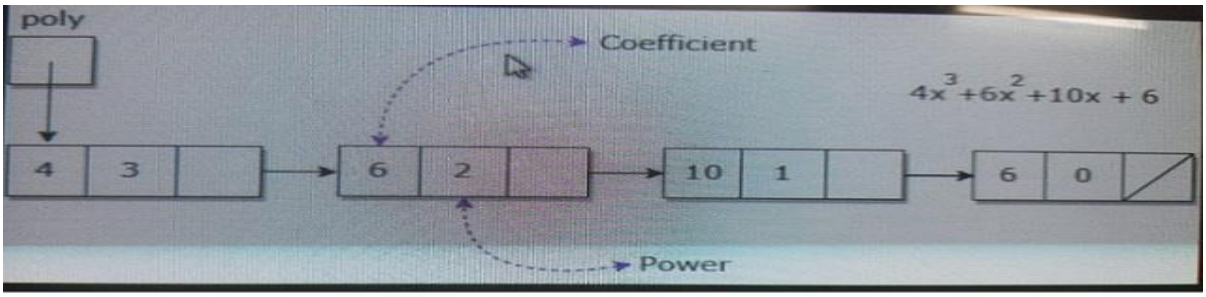
$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

The sign of each coefficient and exponent is stored within the coefficient and the exponent itself

Additional terms having equal exponent is possible one

The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



Representation of Polynomial

Polynomial can be represented in the various ways. These are:

By the use of arrays

By the use of Linked List

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Example:

Input:

1st number = $5x^2 + 4x^1 + 2x^0$

2nd number = $-5x^1 - 5x^0$

Output:

$5x^2-1x^1-3x^0$

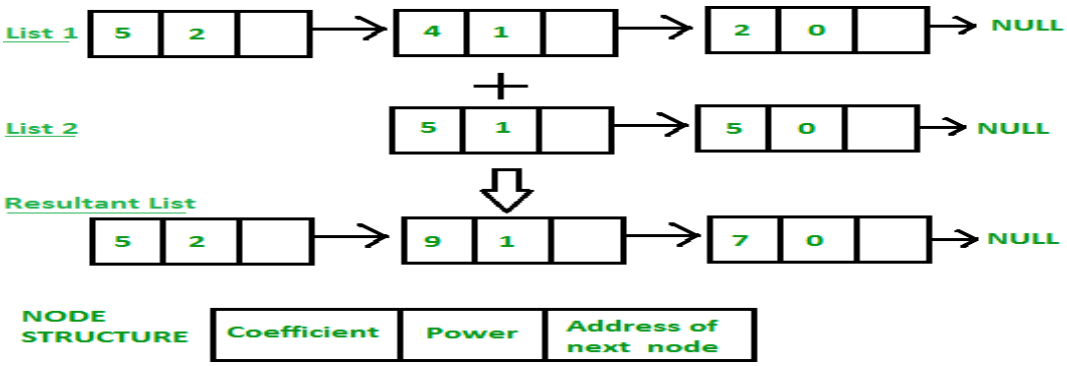
Input:

1st number = $5x^3 + 4x^2 + 2x^0$

2nd number = 5x^1 - 5x^0

Output:

$5x^3 + 4x^2 + 5x^1 - 3x^0$



Algorithm :

1. If both the numbers are null then return
2. else if compare the power, if same then add the coefficients and recursively call addPolynomials on the next elements of both the numbers.
3. else if the power of first number is greater then print the current element of first number and recursively call addPolynomial on the next element of the first number and current element of the second number.
4. else print the current element of the second number and recursively call addPolynomial on the current element of first number and next element of second number.

```
 import java.io.*;
import java.util.Scanner;
class Polynomial {
        public static Node addPolynomial(Node p1, Node p2)
        {
        Node a = p1, b = p2, newHead = new Node(0, 0),c = newHead;
                while (a != null || b != null)
           {
                if (a == null)
                  {
                                c.next = b;
                                break;
                        }
                        else if (b == null) {
                                c.next = a;
                                break;
                        }
                        else if (a.pow == b.pow) {
                                c.next = new Node(a.coeff + b.coeff, a.pow);
                                a = a.next;
                                b = b.next;
                        }
                        else if (a.pow > b.pow) {
                                c.next = new Node(a.coeff, a.pow);
                                a = a.next;
                        }
                        else if (a.pow < b.pow) {
                                c.next = new Node(b.coeff, b.pow);
```

```
                        b = b.next;
                    }
                    c = c.next;
                }
                return newHead.next;
        }
    }
    // Utilities for Linked List Nodes
    class Node {
        int coeff;
        int pow;
        Node next;
        Node(int a, int b)
        {
                coeff = a;
                pow = b;
                next = null;
        }
    }
    //Linked List main class
    class LinkedList {
        public static void main(String args[])
        {
                Node start1 = null, cur1 = null, start2 = null,cur2 = null;
                int[] list1_coeff = { 5, 4, 2 };
                int[] list1_pow = { 2, 1, 0 };
                int  n = list1_coeff.length;
                int i = 0;
                while (n-- > 0) {
                        int a = list1_coeff[i];
                        int b = list1_pow[i];
                        Node ptr = new Node(a, b);
                        if (start1 == null) {
                                start1 = ptr;
                                cur1 = ptr;
                        }
                        else {
                                cur1.next = ptr;
                                cur1 = ptr;
                        }
                        i++;
                }
                int[] list2_coeff = { -5, -5 };
                int[] list2_pow = { 1, 0 };
                n = list2_coeff.length;
                i = 0;
                while (n-- > 0) {
                        int a = list2_coeff[i];
```

```
                int b = list2_pow[i];
                Node ptr = new Node(a, b);
                if (start2 == null) {
                        start2 = ptr;
                        cur2 = ptr;
                }

                else {
                        cur2.next = ptr;
                        cur2 = ptr;
                }
                i++;
        }
        Polynomial obj = new Polynomial();
        Node sum = obj.addPolynomial(start1, start2);
        Node trav = sum;
        while (trav != null) {
                System.out.print(trav.coeff + "x^" + trav.pow);
                if (trav.next != null)
                        System.out.print(" + ");
                trav = trav.next;
        }
        System.out.println();
}}
```

UNIT-2

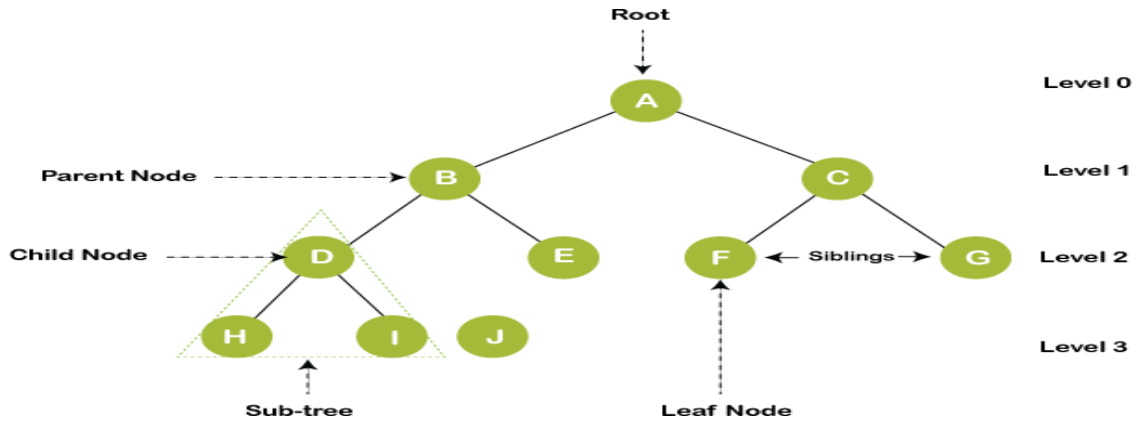
➢ Non-Linear Data Structures

A non-linear data structure is also another type of data structure in which the data elements are not arranged in a contiguous manner. As the arrangement is non sequential, so the data elements cannot be traversed or accessed in a single run. In the case of linear data structure, element is connected to two elements (previous and the next element), whereas, in the non-linear data structure, an element can be connected to more than two elements..

Trees and **Graphs** are the types of non-linear data structure.

- **Tree**

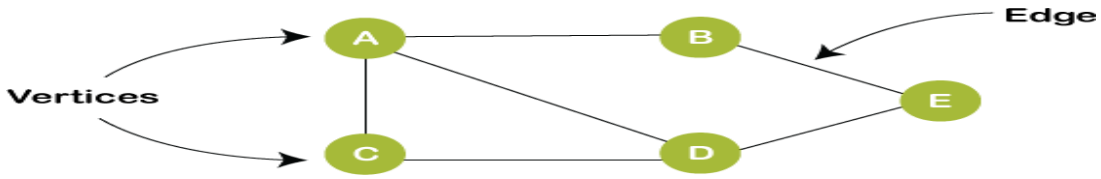
It is a non-linear data structure that consists of various linked nodes. It has a hierarchical tree structure that forms a parent-child relationship. The diagrammatic representation of a **tree** data structure is shown below:



For example, the posts of employees are arranged in a tree data structure like managers, officers, clerk. In the above figure, **A** represents a manager, **B** and **C** represent the officers, and other nodes represent the clerks.

- **Graph**

  A graph is a non-linear data structure that has a finite number of vertices and edges, and these edge to connect the vertices. The vertices are used to store the data elements, while the edges repr relationship between the vertices. A graph is used in various real-world problems like telephone circuit networks, social networks like LinkedIn, Facebook. In the case of facebook, a single us considered as a node, and the connection of a user with others is known as edges.

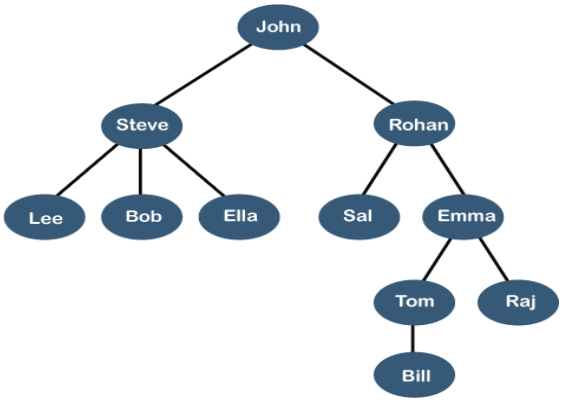


➢ Tree Data Structure

We read the linear data structures like an array, linked list, stack and queue in which all the elements are arranged in a sequential manner. The different data structures are used for different kinds of data.

Some factors are considered for choosing the data structure:

- What type of data needs to be stored?: It might be a possibility that a certain data structure can be the best fit for some kind of data.
- Cost of operations: If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the binary search. The binary search works very fast for the simple list as it divides the search space into half.



- Memory usage: Sometimes, we want a data structure that utilizes less memory.

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:

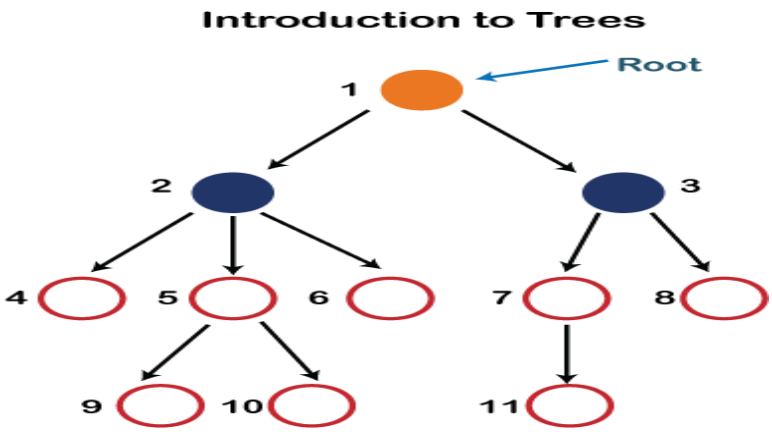
The above tree shows the organization hierarchy of some company. In the above structure, john is the CEO of the company, and John has two direct reports named as Steve and Rohan. Steve has three direct reports named Lee, Bob, Ella where Steve is a manager. Bob has two direct reports named Sal and Emma. Emma has two direct reports named Tom and Raj. Tom has one direct report named Bill. This particular logical structure is known as a Tree. Its structure is similar to the real tree, so it is named a Tree. In this structure, the root is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data in a hierarchical way.

Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.

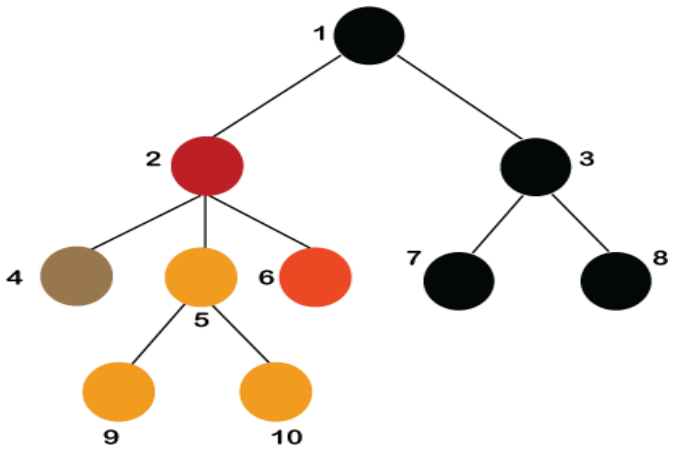
Some basic terms used in Tree data structure.

Let's consider the tree structure, which is shown below:



In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a link between the two nodes.

- Root: The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is the root node of the tree. If a node is directly linked to some other node, it would be called a parent-child relationship.
- Child node: If the node is a descendant of any node, then the node is known as a child node.
- Parent: If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- Sibling: The nodes that have the same parent are known as siblings.
- Leaf Node:- The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.



- Internal nodes: A node has atleast one child node known as an internal
- Ancestor node:- An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- Descendant: The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- Recursive data structure: The tree is also known as a recursive data structure. A tree can be defined as recursively because the distinguished node in a tree data structure

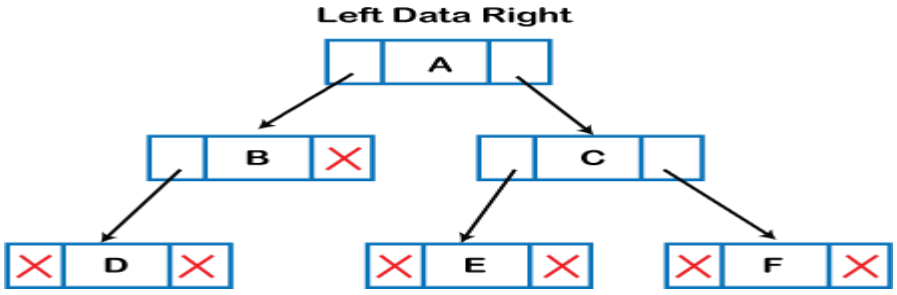
is known as a root node. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.
Number of edges: If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.

- Depth of node x: The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- Height of node x: The height of node x can be defined as the longest path from the node x to the leaf node.

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

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

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Applications of trees

The following are the applications of trees:

- Storing naturally hierarchical data: Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- Organize data: It is used to organize data for efficient insertion, deletion and

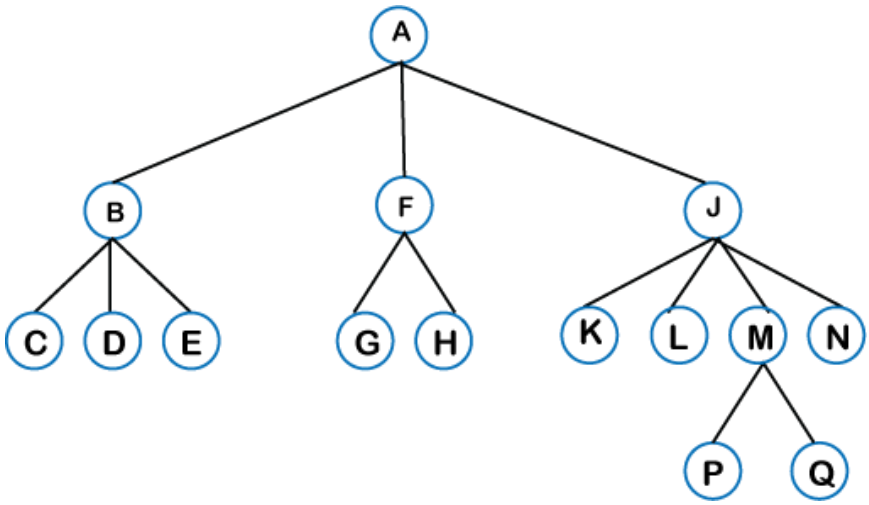
searching. For example, a binary tree has a logN time for searching an element.

- Trie: It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- Heap: It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- B-Tree and B+Tree: B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- Routing table: The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

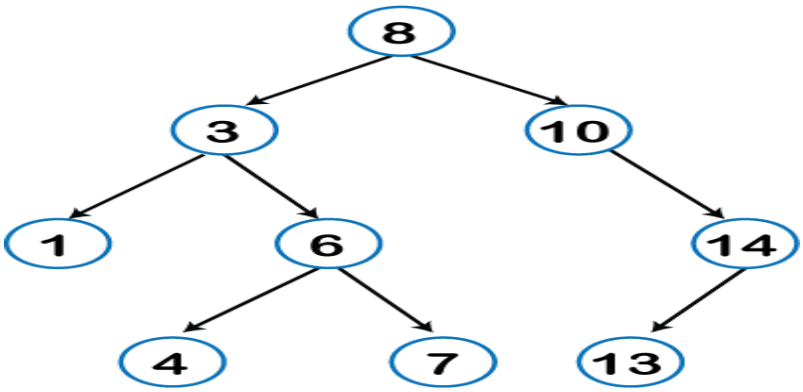
The following are the types a tree data structure: of

- **General tree:** The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as subtrees.



There can be **n** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered. Every non-empty tree has a downward edge, and these edges are connected to the nodes known as **child nodes**. The root node is labeled with level 0. The nodes that have the same parent are known as **siblings**.

- **Binary tree:** Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



Binary Search tree: Binary search tree is a non-linear data structure in which one node is connected to **n** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer). Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

A node can be created with the help of a user-defined data type known as **struct,** as shown below:

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

The above is the node structure with three fields: data field, the second field is the left pointer of the node type, and the third field is the right pointer of the node type.

- **AVL tree**

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the **binary tree** as well as of the **binary search tree**. It is a self-balancing binary search tree that was invented by **Adelson Velsky Lindas**. Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the **balancing factor**.

We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the difference between the height of the left subtree and the height of the right subtree. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.

- **Red-Black Tree**

The red-Black tree is the binary search tree. The prerequisite of the Red-Black tree is that we should know about the binary search tree. In a binary search tree, the value of the left-subtree should be less than the value of that node, and the value of the right-subtree should be greater than the value of that node. As we know that the time complexity of binary search in the average case is $\log_2 n$, the best case is O(1), and the worst case is O(n).
When any operation is performed on the tree, we want our tree to be balanced so that all the operations like searching, insertion, deletion, etc., take less time, and all these operations will have the time complexity of **$\log_2 n$.**
The red-black tree is a self-balancing binary search tree. AVL tree is also a height balancing binary search tree then **why do we require a Red-Black tree**. In the AVL tree, we do not know how many rotations would be required to balance the tree, but in the Red-black tree, a maximum of 2 rotations are required to balance the tree. It contains one extra bit that represents either the red or black color of a node to ensure the balancing of the tree.

- **Splay tree**

The splay tree data structure is also binary search tree in which recently accessed element is placed at the root position of tree by performing some rotation operations. Here, **splaying** means the recently accessed node. It is a **self-balancing** binary search tree having no explicit balance condition like **AVL** tree.

It might be a possibility that height of the splay tree is not balanced, i.e., height of both left and right subtrees may differ, but the operations in splay tree takes order of **logN** time where **n** is the number of nodes.

Splay tree is a balanced tree but it cannot be considered as a height balanced tree because after each operation, rotation is performed which leads to a balanced tree.

- **Treap**

Treap data structure came from the Tree and Heap data structure. So, it comprises the properties of both Tree and Heap data structures. In Binary search tree, each node on the left subtree must be equal or less than the value of the root node and each node on the right subtree must be equal or greater than the value of the root node. In heap data structure, both right and left subtrees contain larger keys than the root; therefore, we can say that the root node contains the lowest value.
In treap data structure, each node has both **key** and **priority** where key is derived from the Binary search tree and priority is derived from the heap data structure.
The **Treap** data structure follows two properties which are given below:

- Right child of a node>=current node and left child of a node <=current node (binary tree)
- Children of any subtree must be greater than the node (heap)
- **B-tree**

B-tree is a balanced **m-way** tree where **m** defines the order of the tree. Till now, we read that the node contains only one key but b-tree can have more than one key, and more than 2 children. It always maintains the sorted data. In binary tree, it is possible that leaf nodes can be at different levels, but in b-tree, all the leaf nodes must be at the same level.

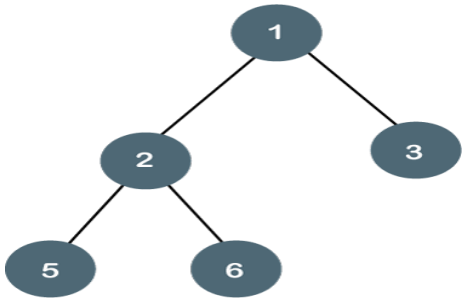
If order is m then node has the following properties:

- Each node in a b-tree can have maximum **m** children
- For minimum children, a leaf node has 0 children, root node has minimum 2 children and internal node has minimum ceiling of m/2 children. For example, the value of m is 5 which means that a node can have 5 children and internal nodes can contain maximum 3 children.
- Each node has maximum (m-1) keys.

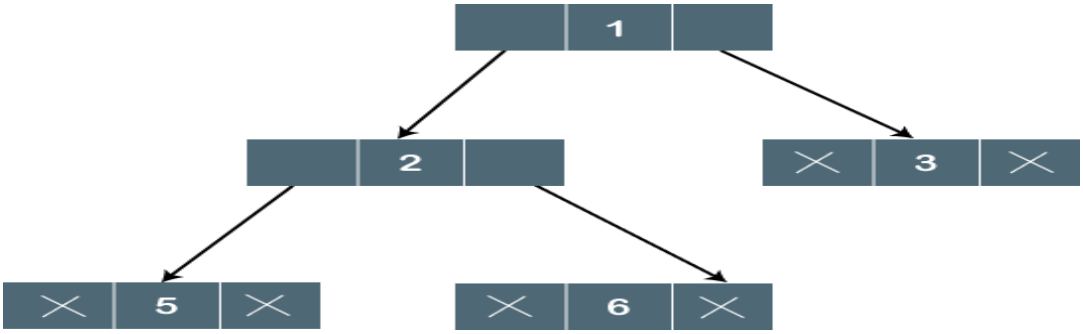
The root node must contain minimum 1 key and all other nodes must contain atleast **ceiling of m/2 minus 1** keys

➢ Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.
Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain NULL pointer on both left and right parts.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots 2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to h+1.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would

be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$\log_2(n+1) = \log2(2^{h+1})$

$\log_2(n+1) = h+1$

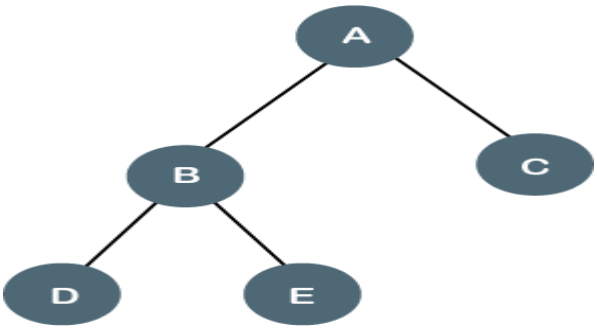
$h = \log_2(n+1) - 1$

The maximum height can be computed as:

As we know that,

$n = h+1$

$h = n-1$



Types of Binary Tree

There are four types of Binary tree:

- Full/ proper/ strict Binary tree
- Complete Binary tree
- Perfect Binary tree
- Degenerate Binary tree
- Balanced Binary tree

1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.

In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., $2^{h+1} -1$.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is $\log_2(n+1) - 1$.
- The maximum height of the full binary tree can be computed as:

$n= 2*h - 1$

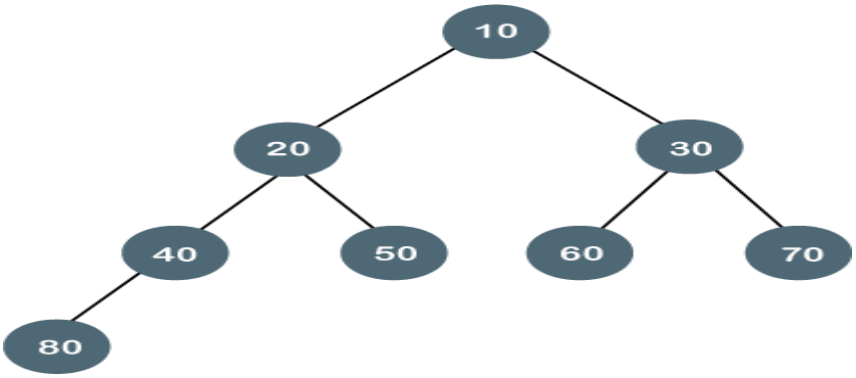
$n+1 = 2*h$

$h = n+1/2$

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.



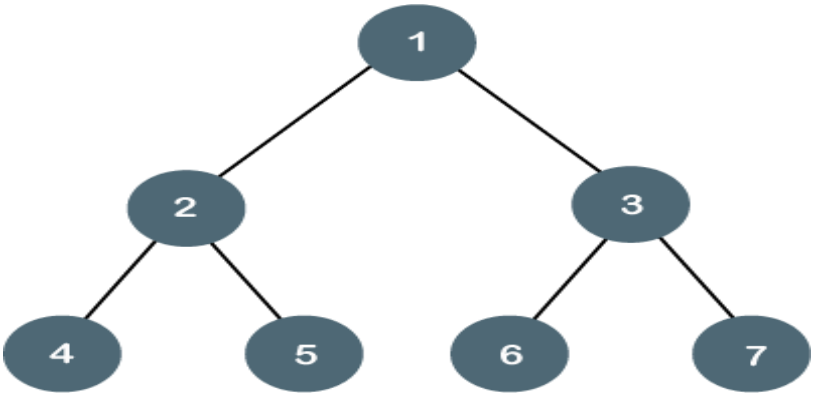
The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.
- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is $\log_2(n+1) - 1$.
- The maximum height of a complete binary tree is

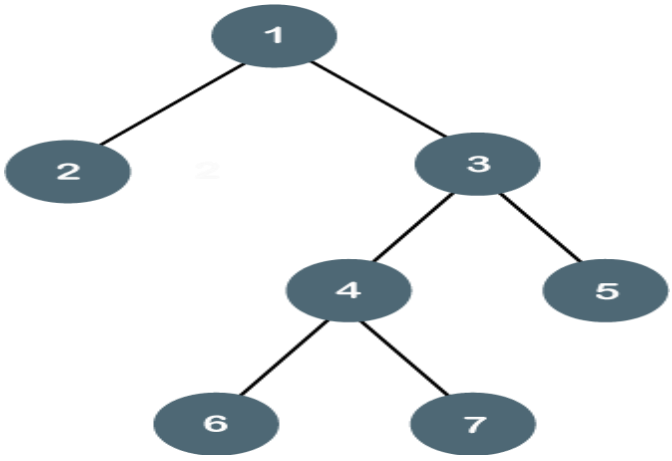
Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



Let's look at a simple example of a perfect binary tree.

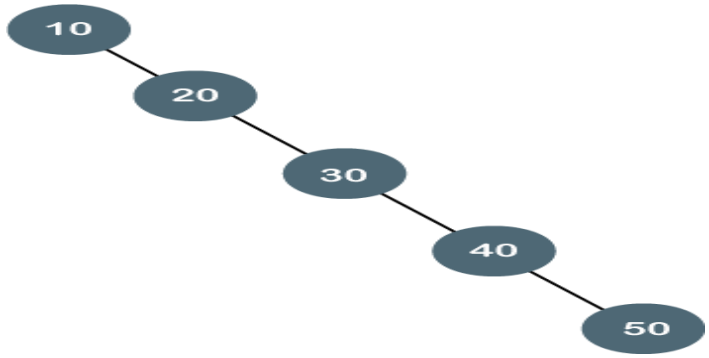
The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.



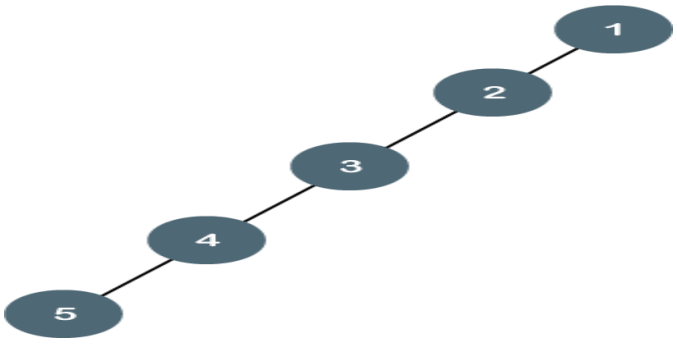
Note: All the perfect binary trees are the complete binary trees as well as the full binary tr but vice versa is not true, i.e., all complete binary trees and full binary trees are the perf binary trees.

Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children. Let's understand the Degenerate binary tree through examples.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

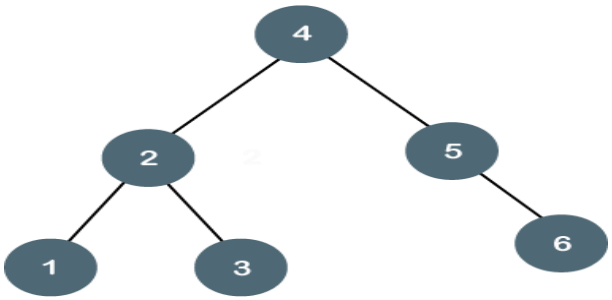


The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

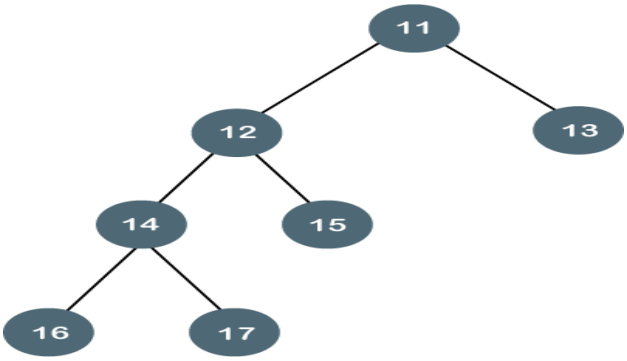
Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1.

For example, AVL and Red-Black trees are balanced binary tree.
Let's understand the balanced binary tree through examples.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.



The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

Binary Tree Implementation

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

```
struct node
{
  int data,
  struct node *left, *right;
}
```

In the above structure, data is the value, left pointer contains the address of the left node, and right pointer contains the address of the right node.

When all the nodes are created, then it forms a binary tree structure. The process of visiting the nodes is known as tree traversal. There are three types traversals used to visit a node:

- Inorder traversal
- Preorder traversal
- Postorder traversal

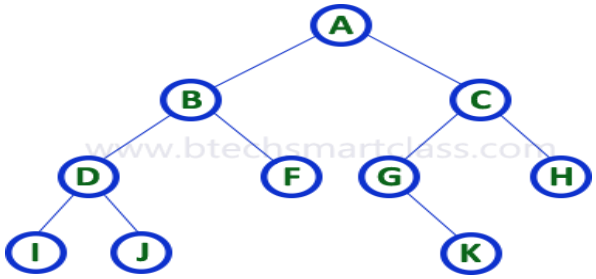
➢ Binary tree Representation and Traversals

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

Consider the above example of a binary tree and it is represented as follows...

To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **2n + 1**.

2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...

The above example of the binary tree represented using Linked list representation is shown as follows...

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-

tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −**D → B → E → A → F → C → G**

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed −

Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step3 − Visit root node.

➢ Threaded binary trees

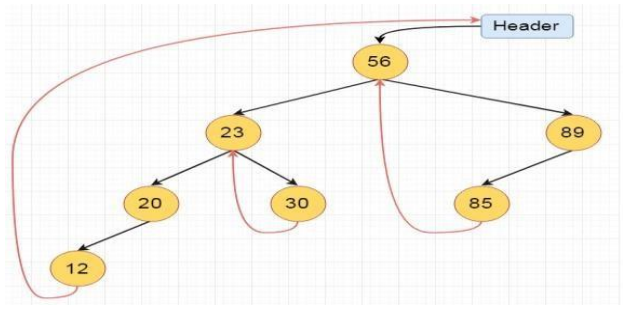
Here we will see the threaded binary tree data structure. We know that the binary tree nodes may have at most two children. But if they have only one children, or no children, the link part in the linked list representation remains null. Using threaded binary tree representation, we can reuse that empty links by making some threads.

If one node has some vacant left or right child area, that will be used as thread. There are two types of threaded binary tree. The single threaded tree or fully threaded binary tree. In single threaded mode, there are another two variations. Left threaded and right threaded.

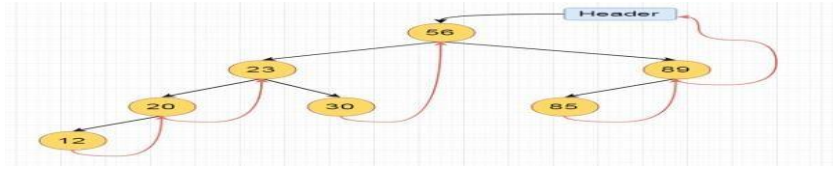
In the left threaded mode if some node has no left child, then the left pointer will point to its inorder predecessor, similarly in the right threaded mode if some node has no right child, then the right pointer will point to its inorder successor. In both cases, if no successor or predecessor is present, then it will point to header node.

For fully threaded binary tree, each node has five fields. Three fields like normal binary tree node, another two fields to store Boolean value to denote whether link of that side is actual link or thread.

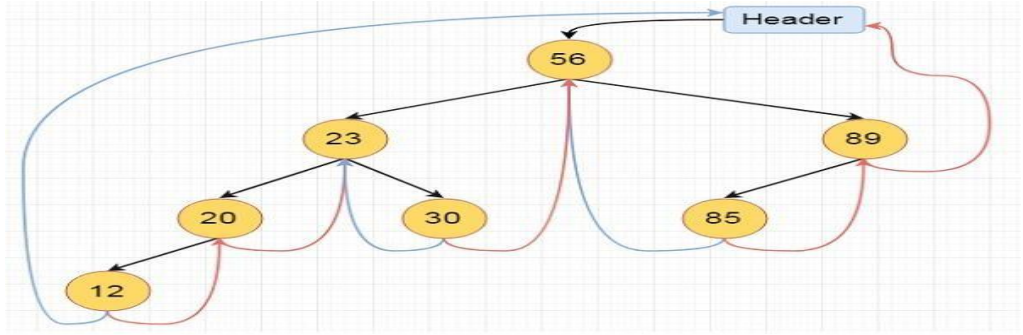
Left Thread Flag	Left Link	Data	Right Link	Right Thread Flag



These are the examples of left and right threaded tree



This is the fully threaded binary tree



Representing a Set as Tree:

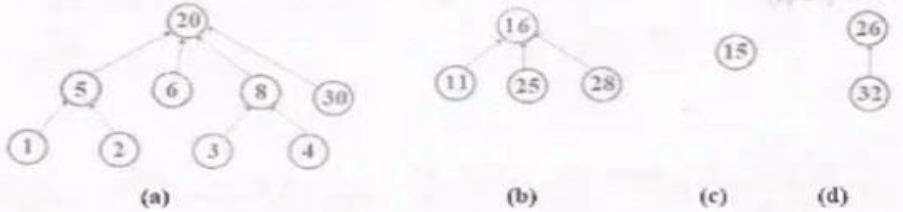
Any set S may be represented as tree with |S| nodes, one node per element.

- Any element of the set is selected as a root, any subset of the remaining elements could be the children of the root, any subset of the remaining elements could be the grand children of the root and so on.
- To represent each set by a tree, each element points to its parent.
- The representative of a set is the root.

Note that the trees are not necessarily binary tree

Example

Below are the some set representation of trees



(a) (b) (c) (d)

The elements 1, 2, 20, 30 and so on are in the set with root 20.
The elements 11, 16, 25 and 28 are in the set 16.
The element 15 is in the set 15.
The elements 26 and 32 are in the set 26.

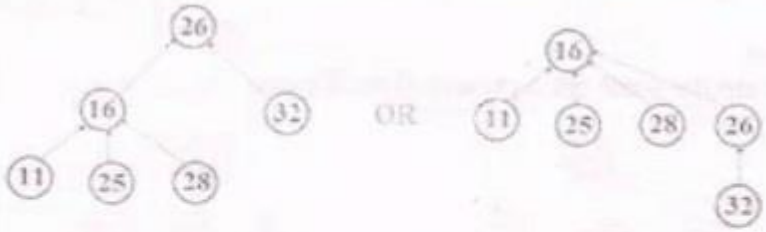
- *union():* Join two subsets into a single subset.

To unite the two trees of elements, we make one tree a sub tree of the other

```
function union(x, y)
        xRoot := findSet(x)
        yRoot := findSet(y)
        parent[yRoot] := xRoot
```

Example

union(b,d)

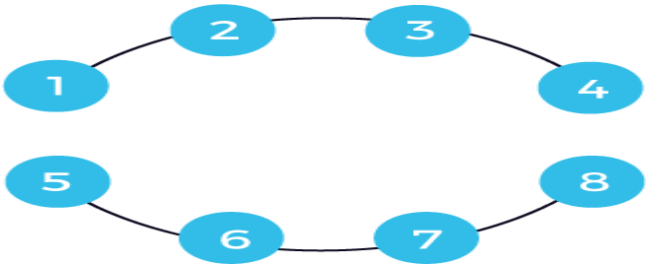


➢ Union-Find operations

Disjoint set data structure

The disjoint set data structure is also known as union-find data structure and merge-find set. It is a data structure that contains a collection of disjoint or non-overlapping sets. The disjoint set means that when the set is partitioned into the disjoint subsets. The various operations can be performed on the disjoint subsets. In this case, we can add new sets, we can merge the sets, and we can also find the representative member of a set. It also allows to find out whether the two elements are in the same set or not efficiently.

The disjoint set can be defined as the subsets where there is no common element between the two sets. Let's understand the disjoint sets through an example.



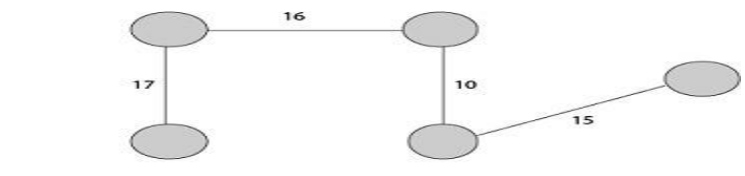
s1 = {1, 2, 3, 4}
s2 = {5, 6, 7, 8}

We have two subsets named s1 and s2. The s1 subset contains the elements 1, 2, 3, 4, while s2 contains the elements 5, 6, 7, 8. Since there is no common element between these two sets, we will not get anything if we consider the intersection between these two sets. This is also known as a disjoint set where no elements are common. Now the question arises how we can perform the operations on them. We can perform only two operations, i.e., find and union.

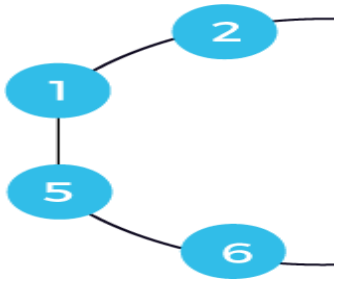
In the case of find operation, we have to check that the element is present in which set. There are two sets named s1 and s2 shown below:

Suppose we want to perform the union operation on these two sets. First, we have to check whether the elements on which we are performing the union operation belong to different or same sets. If they belong to the different sets, then we can perform the union operation; otherwise, not. For example, we want to perform the union operation between 4 and 8. Since 4 and 8 belong to different sets, so we apply the union operation. Once the union operation is performed, the edge will be added between the 4 and 8 shown as below:

When the union operation is applied, the set would be represented as:



Minimum Cost Spanning Tree
Total Cost = 17+16+10+15=58



s1Us2 = {1, 2, 3, 4, 5, 6, 7,

8}

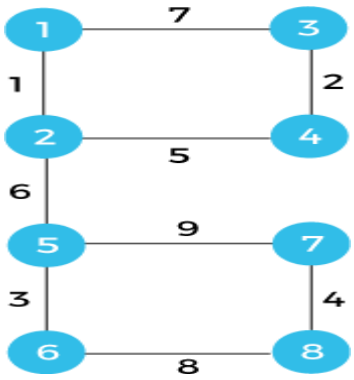
Suppose we add one more edge between 1 and 5. Now the final set can be represented as:

s3 = {1, 2, 3, 4, 5, 6, 7, 8}

If we consider any element from the above set, then all the elements belong to the same set; it means that the cycle exists in a graph.

How can we detect a cycle in a graph?

We will understand this concept through an example. Consider the below example to detect a cycle with the help of using disjoint sets.



U = {1, 2, 3, 4, 5, 6, 7, 8}

Each vertex is labelled with some weight. There is a universal set with 8 vertices. We will consider each edge one by one and form the sets.

First, we consider vertices 1 and 2. Both belong to the universal set; we perform the union operation between elements 1 and 2. We will add the elements 1 and 2 in a set s1 and remove these two elements from the universal set shown below:

s1 = {1, 2}

The vertices that we consider now are 3 and 4. Both the vertices belong to the universal set; we perform the union operation between elements 3 and 4. We will form the set s3 having elements 3 and 4 and remove the elements from the universal set shown as below:

s2 = {3, 4}

The vertices that we consider now are 5 and 6. Both the vertices belong to the universal set, so we perform the union operation between elements 5 and 6. We will form the set s3 having elements 5 and 6 and will remove these elements from the universal set shown as below:

s3 = {5, 6}

The vertices that we consider now are 7 and 8. Both the vertices belong to the universal set, so we perform the union operation between elements 7 and 8. We will form the set s4 having elements 7 and 8 and will remove these elements from the universal set shown as below:

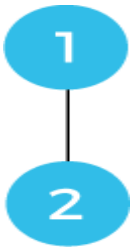
s4 = {7, 8}

The next edge that we take is (2, 4). The vertex 2 is in set 1, and vertex 4 is in set 2, so both the vertices are in different sets. When we apply the union operation, then it will form the new set shown as below:

s5 = {1, 2, 3, 4}

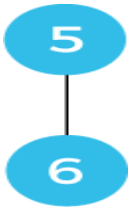
The next edge that we consider is (2, 5). The vertex 2 is in set 5, and the vertex 5 is in set s3, so both the vertices are in different sets. When we apply the union operation, then it will form the new set shown as below:

s6 = {1, 2, 3, 4, 5, 6}
Now, two sets are left which are given below:
s4 = {7, 8}
s6 = {1, 2, 3, 4, 5, 6}
The next edge is (1, 3). Since both the vertices, i.e.,1 and 3 belong to the same set, so it forms a cycle. We will not consider this vertex.
The next edge is (6, 8). Since both vertices 6 and 8 belong to the different vertices s4 and s6, we will perform the union operation. The union operation will form the new set shown as below:
s7 = {1, 2, 3, 4, 5, 6, 7, 8}
The last edge is left, which is (5, 7). Since both the vertices belong to the same set named s7, a cycle is formed.
How can we represent the sets graphically?
We have a universal set which is given below:
U = {1, 2, 3, 4, 5, 6, 7, 8}
We will consider each edge one by one to represent graphically.
First, we consider the vertices 1 and 2, i.e., (1, 2) and represent them through graphically shown as below:



In the above figure, vertex 1 is the parent of vertex 2.
Now we consider the vertices 3 and 4, i.e., (3, 4) and represent them graphically shown as below:

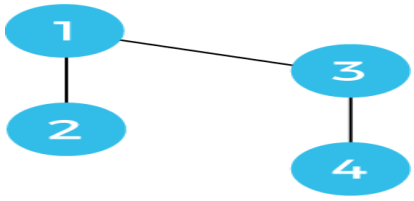
In the above figure, vertex 3 is the parent of vertex 4.
Consider the vertices 5 and 6, i.e., (5, 6) and represent them graphically shown as below:
In the above figure, vertex 5 is the parent of vertex 6.



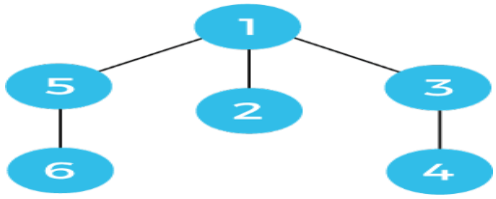
Now, we consider the vertices 7 and 8, i.e., (7, 8) and represent them through graphically shown as below:

In the above figure, vertex 7 is the parent of vertex 8.

Now we consider the edge (2, 4). Since 2 and 4 belong to different sets, so we need to perform the union operation. In the above case, we observe that 1 is the parent of vertex 2 whereas vertex 3 is the parent of vertex 4. When we perform the union operation on the two sets, i.e., s1 and s2, then 1 vertex would be the parent of vertex 3 shown as below:



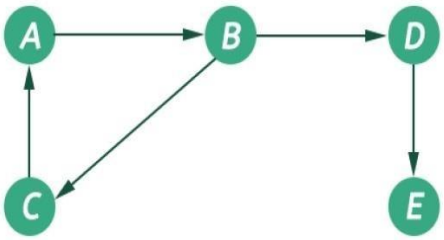
The next edge is (2, 5) having weight 6. Since 2 and 5 are in two different sets so we will perform the union operation. We make vertex 5 as a child of the vertex 1 shown as below: We have chosen vertex 5 as a child of vertex 1 because the vertex of the graph having parent 1 is more than the graph having parent 5.



The next edge is (1, 3) having weight 7. Both vertices 1 and 3 are in the same set, so there is no need to perform any union operation. Since both the vertices belong to the same set; therefore, there is a cycle. We have detected a cycle, so we will consider the edges further.

➢ Graph and its representations

The graph is a non-linear data structures. This represents data using nodes, and their relations using edges. A graph G has two sections. The vertices, and edges. Vertices are represented using set V, and Edges are represented as set E. So the graph notation is G(V,E). Let us see one example to get the idea.



In this graph, there are five vertices and five edges. The edges are directed. As an example, if we choose the edge connecting vertices B and D, the source vertex is B and destination is D. So we can move B to D but not move from D to B.

Graph data structure is represented using following representations...

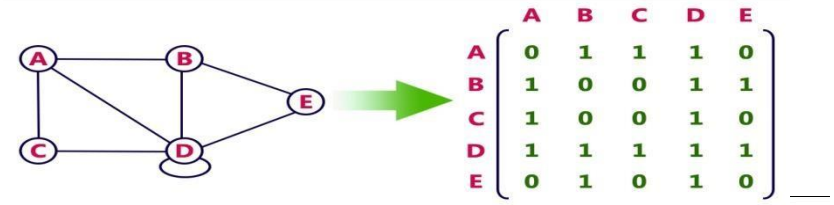
1. **Adjacency Matrix**
2. **Incidence Matrix**

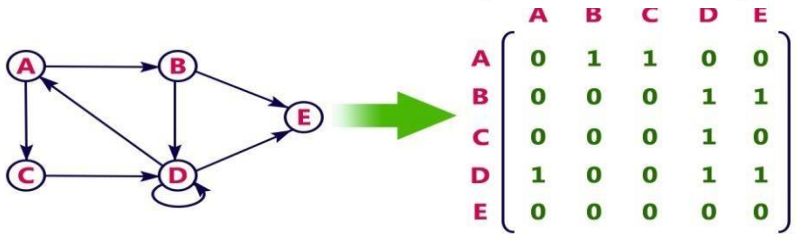
3. Adjacency List

The graphs are non-linear, and it has no regular structure. To represent a graph in

Adjacency Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex. For example, consider the following undirected graph representation...

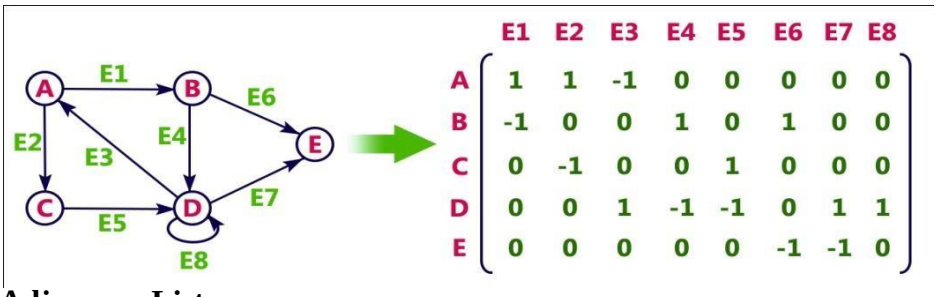




Directed graph representation...

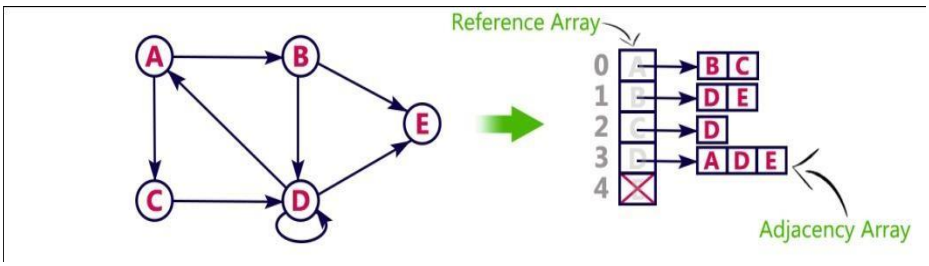
Incidence Matrix

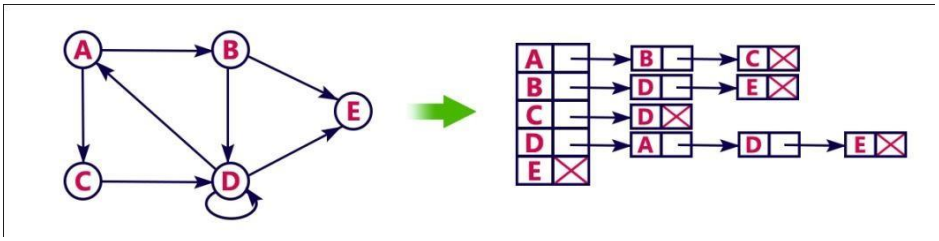
In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.
For example, consider the following directed graph representation..



Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices. For example, consider the following directed graph representation implemented using l





This representation can also be implemented using an array as follows

➢ Graph Traversal

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

➢ Depth First Search (DFS) Algorithm

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

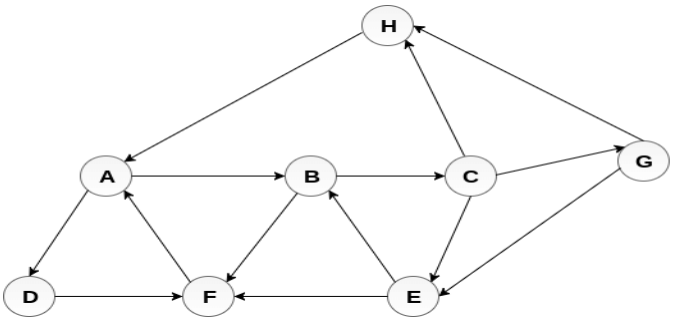
Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Example :

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Solution :

Push H onto the stack

STACK : H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

Print H

STACK : A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

Print A

Stack : B, D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

Print D

Stack : B, F

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

Print F

Stack : B

Pop the top of the stack i.e. B and push all the neighbours

Print B

Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

Print C

Stack : E, G

Pop the top of the stack i.e. G and push all its neighbours.

Print G

Stack : E

Pop the top of the stack i.e. E and push all its neighbours.

Print E

Stack :

Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be :

H → A → D → F → B → C → G → E

➢ Breadth First Search (BFS) Algorithm

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice

Algorithm

Step 1: SET STATUS = 1 (ready state)
for each node in G

Step 2: Enqueue the starting node A
and set its STATUS = 2
(waiting state)

Step 3: Repeat Steps 4 and 5 until
QUEUE is empty

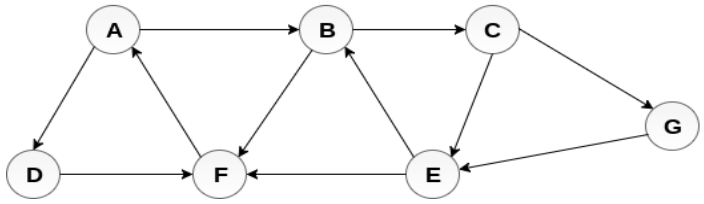
Step 4: Dequeue a node N. Process it
and set its STATUS = 3
(processed state).

Step 5: Enqueue all the neighbours of
N that are in the ready state
(whose STATUS = 1) and set
their STATUS = 2
(waiting state)
[END OF LOOP]

Step 6: EXIT

Example

Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

Solution:

Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely **QUEUE1** and **QUEUE2**. **QUEUE1** holds all the nodes that are to be processed while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**.

Let's start examining the graph from Node A.

1. Add A to QUEUE1 and NULL to QUEUE2.

QUEUE1 = {A}

QUEUE2 = {NULL}

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

QUEUE1 = {B, D}

QUEUE2 = {A}

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

QUEUE1 = {D, C, F}

QUEUE2 = {A, B}

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

QUEUE1 = {C, F}

QUEUE2 = { A, B, D}

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

QUEUE1 = {F, E, G}

QUEUE2 = {A, B, D, C}

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

QUEUE1 = {E, G}

QUEUE2 = {A, B, D, C, F}

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.

QUEUE1 = {G}

QUEUE2 = {A, B, D, C, F, E}

Now, backtrack from E to A, using the nodes available in QUEUE2.

The minimum path will be **A → B → C → E**.

➢ Applications of graphs

Since they are powerful abstractions, graphs can be very important in modeling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

1. Social network graphs: to tweet or not to tweet. Graphs that represent who knows whom,who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, oreven who might be a good match for who, or is that whom

1. Transportation networks. In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations.

They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

2. Utility graphs. The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliabil- ity of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.

3. Document link graphs. The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

4. Protein-protein interactions graphs. Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.

5. Network packet traffic graphs. Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

6. Scene graphs. In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

7. Finite element meshes. In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements forms a graph that iscalled a finite element mesh.

8. Robot planning. Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths forautonomous vehicles.

9. Neural networks. Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^{11} neurons and close to 10^{15} synapses.
10. Graphs in quantum field theory. Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).

11. Semantic networks. Vertices represent words or concepts and edges represent the rela- tionships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.

12. Graphs in epidemiology. Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

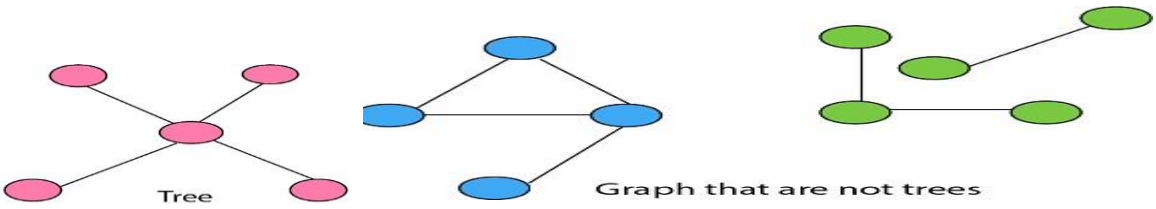
13. Graphs in compilers. Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
14. Constraint graphs. Graphs are often used to represent constraints among items. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells thatoverlap.
15. Dependence graphs. Graphs can be used to represent dependences or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abidingby the dependences.

➢ Introduction of Minimum Spanning Tree

Tree:

A tree is a graph with the following properties:

1. The graph is connected (can go from anywhere to anywhere)
2. There are no cyclic (Acyclic)



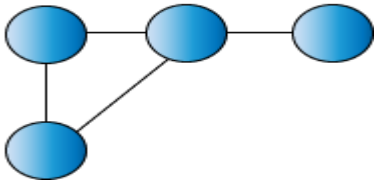
Spanning Tree:

Given a connected undirected graph, a spanning tree of that graph is a subgraph that is a tree and joined all vertices. A single graph can have many spanning trees.

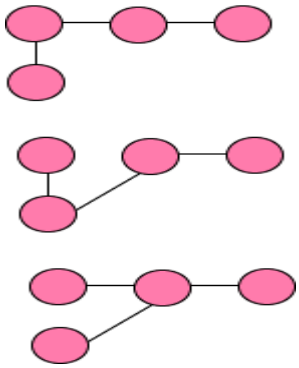
For Example:

For the above-connected graph. There can be multiple spanning Trees like

Connected Undirected Graph



Spanning Trees



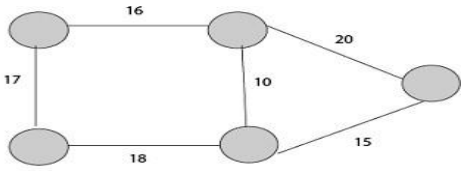
Properties of Spanning Tree:

1. There may be several minimum spanning trees of the same weight having the minimum number of edges.
2. If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
3. If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.
4. A connected graph G can have more than one spanning trees.
5. A disconnected graph can't have to span the tree, or it can't span all the vertices.
6. Spanning Tree doesn't contain cycles.
7. Spanning Tree has **(n-1) edges** where n is the number of vertices.

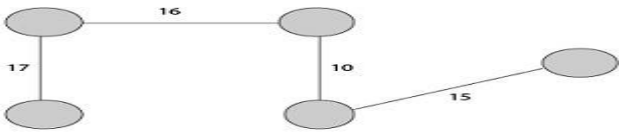
Addition of even one single edge results in the spanning tree losing its property of **Acyclicity** and elimination of one single edge results in its losing the property of connectivity.

Minimum Spanning Tree:

Minimum Spanning Tree is a Spanning Tree which has minimum total cost. If we have a linked undirected graph with a weight (or cost) combine with each edge. Then the cost of spanning tree would be the sum of the cost of its edges.



Connected , Undirected Graph



Minimum Cost Spanning Tree
Total Cost = 17+16+10+15=58

➢ Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

If the graph is not linked, then it finds a Minimum Spanning Tree.
Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

MST- KRUSKAL (G, w)

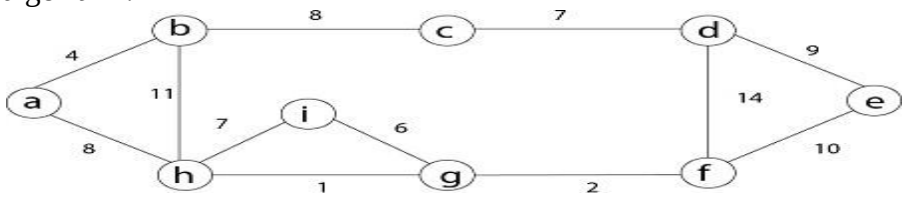
1. A ← ∅
2. for each vertex v ∈ V [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) ∈ E, taken in non decreasing order by weight
6. do if FIND-SET (μ) ≠ if FIND-SET (v)
7. then A ← A U {(u, v)}
8. UNION (u, v)
9. return A

Analysis: Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and log V^2= 2 x log V is O (log V).
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, V ≤ 2 E, so log V is O (log E).

Thus the total time is
O (E log E) = O (E log V).

For Example: Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



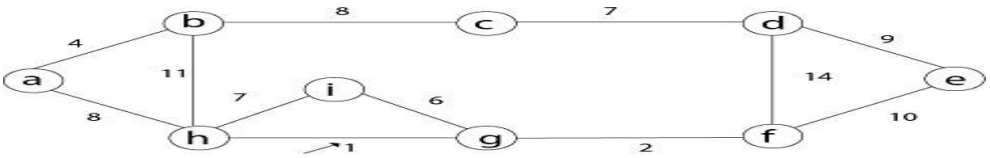
Solution: First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.
There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	i	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

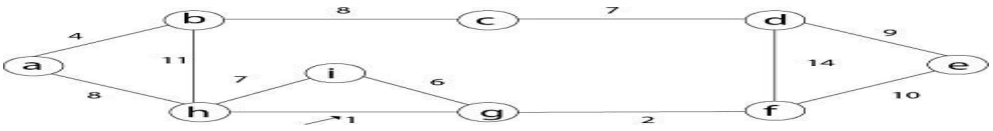
Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong

to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

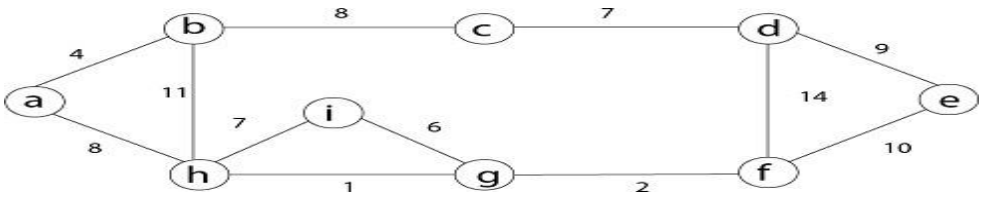
Step1: So, first take (h, g) edge



Step 2: then (g, f) edge.

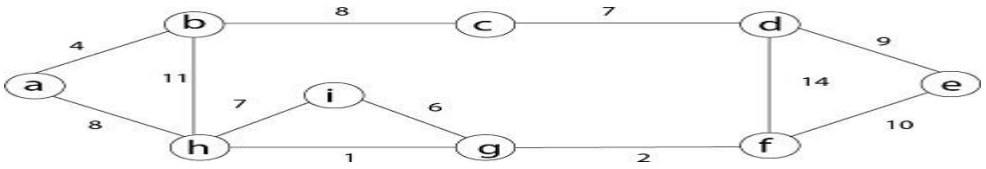


Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes



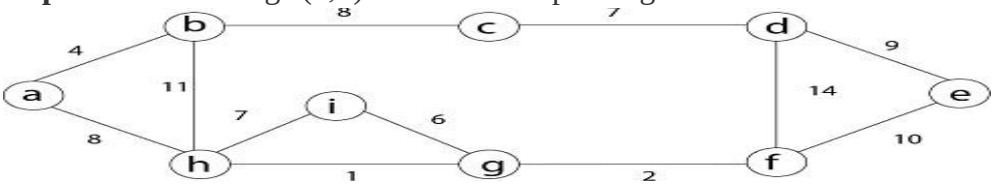
Step 4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



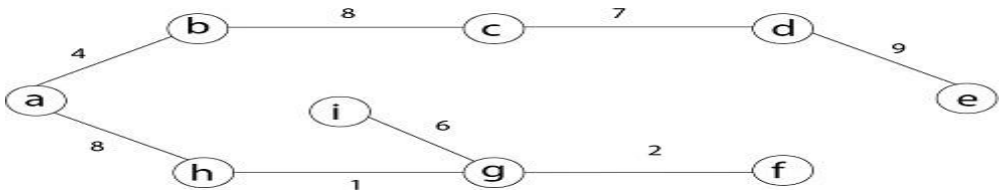
Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

Step 6: After that edge (d, f) and the final spanning tree is shown as in dark lines.



Step 7: This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

e → f, b → h, d → f [cycle will be formed]



Minimum Cost MST

➢ Dijkstra's Algorithm for Single Source Shortest Path Problem

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.

Dijkstra's Algorithm maintains a set S of vertices whose final shortest - path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d [v] = δ (s, v). The algorithm repeatedly selects the vertex u ∈ V - S with the minimum shortest - path estimate, insert u into S and relaxes all edges leaving u.

Because it always chooses the "lightest" or "closest" vertex in V - S to insert into set S, it is called as the greedy strategy.

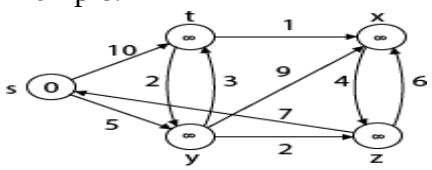
Dijkstra's Algorithm (G, w, s)

1. INITIALIZE - SINGLE - SOURCE (G, s)
2. S←∅
3. Q←V [G]
4. while Q ≠ ∅
5. do u ← EXTRACT - MIN (Q)
6. S ← S U {u}
7. for each vertex v ∈ Adj [u]
8. do RELAX (u, v, w)

Analysis: The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big - O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract - Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is O (|V2 |+|E|=O(V2).

10.1M

Example:



Solution:

Step1: Q =[s, t, x, y, z]

We scanned vertices one by one and find out its adjacent. Calculate the distance of each adjacent to the source vertices.

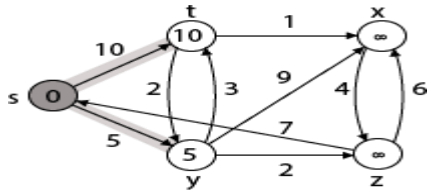
We make a stack, which contains those vertices which are selected after computation of shortest distance.

Firstly we take's' in stack M (which is a source)

M = [S] Q = [t, x, y, z]

Step 2: Now find the adjacent of s that are t and y.

Adj [s] → t, y [Here s is u and t and y are v]



Case - (i) s → t

d [v] > d [u] + w [u, v]

d [t] > d [s] + w [s, t]

∞ > 0 + 10 [false condition]

Then d [t] ← 10

π [t] ← 5

Adj [s] ← t, y

Case - (ii) s→ y

d [v] > d [u] + w [u, v]

d [y] > d [s] + w [s, y]

∞ > 0 + 5 [false condition]

∞ > 5

Then d [y] ← 5

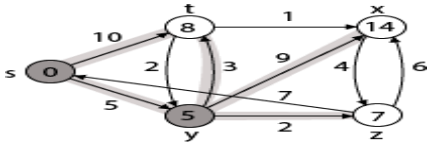
π [y] ← 5

By comparing case (i) and case (ii)

Adj [s] → t = 10, y = 5

y is shortest

y is assigned in 5 = [s, y]



Step 3: Now find the adjacent of y that is t, x, z.

Adj [y] → t, x, z [Here y is u and t, x, z are v]

Case - (i) y →t

d [v] > d [u] + w [u, v]

d [t] > d [y] + w [y, t]

10 > 5 + 3

10 > 8

Then d [t] ← 8

π [t] ← y

Case - (ii) y → x

d [v] > d [u] + w [u, v]

d [x] > d [y] + w [y, x]

∞ > 5 + 9

∞ > 14

Then d [x] ← 14

π [x] ← 14

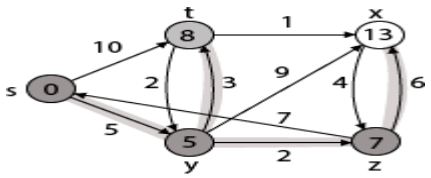
Case - (iii) y → z

d [v] > d [u] + w [u, v]

d [z] > d [y] + w [y, z]
∞ > 5 + 2
∞ > 7

Then d [z] ← 7
π [z] ← y

By comparing case (i), case (ii) and case (iii)
Adj [y] → x = 14, t = 8, z =7
z is shortest
z is assigned in 7 = [s, z]

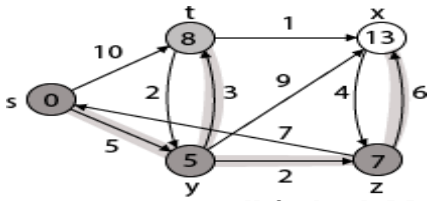


Step - 4 Now we will find adj [z] that are s, x
Adj [z] → [x, s] [Here z is u and s and x are v]
Case - (i) z → x
d [v] > d [u] + w [u, v]
d [x] > d [z] + w [z, x]
14 > 7 + 6
14 > 13

Then d [x] ← 13
π [x] ← z

Case - (ii) z → s
d [v] > d [u] + w [u, v]
d [s] > d [z] + w [z, s]
0 > 7 + 7
0 > 14

∴ This condition does not satisfy so it will be discarded.
Now we have x = 13.



Step 5: Now we will find Adj [t]
Adj [t] → [x, y] [Here t is u and x and y are v]
Case - (i) t → x
d [v] > d [u] + w [u, v]
d [x] > d [t] + w [t, x]
13 > 8 + 1
13 > 9

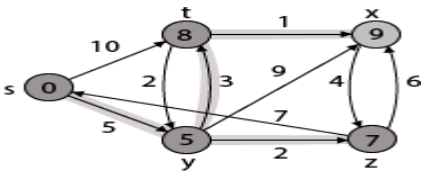
Then d [x] ← 9
π [x] ← t

Case - (ii) t → y
d [v] > d [u] + w [u, v]

d [y] > d [t] + w [t, y]

5 > 10

∴ This condition does not satisfy so it will be discarded.



Thus we get all shortest path vertex as

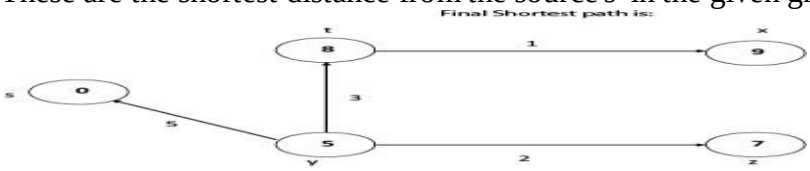
Weight from s to y is 5

Weight from s to z is 7

Weight from s to t is 8

Weight from s to x is 9

These are the shortest distance from the source's' in the given graph.



Disadvantage of Dijkstra's Algorithm:

It does a blind search, so wastes a lot of time while processing.

It can't handle negative edges.

It leads to the acyclic graph and most often cannot obtain the right shortest path.

We need to keep track of vertices that have been visited.

UNIT-3

➢ Search strucutures

Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. Thus the efficient storage of data to facilitate fast searching is an important issue. In this section, we shall investigate the performance of some searching algorithms and the data structures which they use.

Sequential Searches

Let's examine how long it will take to find an item matching a key in the collections we have discussed so far. We're interested in:

a. the average time
b. the worst-case time and
c. the best possible time.

However, we will generally be most concerned with the worst-case time as calculations based on worst-case times can lead to guaranteed performance predictions. Conveniently, the worst-case times are generally easier to calculate than average times.

If there are **n** items in our collection - whether it is stored as an array or as a linked list - then it is obvious that in the worst case, when there is no item in the collection with the desired key, then **n** comparisons of the key with keys of the items in the collection will have to be made.

To simplify analysis and comparison of algorithms, we look for a dominant operation and count the number of times that dominant operation has to be performed. In the case of searching, the dominant operation is the comparison, since the search requires n comparisons in the worst case, we say this is a **O(n)** (pronounce this "big-Oh-n" or "Oh-n") algorithm. The best case - in which the first comparison returns a match - requires a single comparison and is **O(1)**. The average time depends on the probability that the key will be found in the collection - this is something that we would not expect to know in the majority of cases. Thus in this case, as in most others, estimation of the average time is of little utility. If the performance of the system is vital, i.e. it's part of a life-critical system, then we must use the worst case in our design calculations as it represents the best guaranteed performance.

Binary Search

However, if we place our items in an array and sort them in either ascending or descending order on the key first, then we can obtain much better performance with an algorithm called **binary search**.

In binary search, we first compare the key with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array; if it's greater then the item sought must lie in the upper half of the array. So we repeat the procedure on the lower (or upper) half of the array.

Our FindInCollection function can now be implemented:

```
static void *bin_search( collection c, int low, int high, void *key ) {
        int mid;
        /* Termination check */
        if (low > high) return NULL;
        mid = (high+low)/2;
        switch (memcmp(ItemKey(c->items[mid]),key,c->size)) {
                /* Match, return item found */
                case 0: return c->items[mid];
                /* key is less than mid, search lower half */
                case -1: return bin_search( c, low, mid-1, key);
                /* key is greater than mid, search upper half */
```

```
                case 1: return bin_search( c, mid+1, high, key );
                default : return NULL;
                }
        }

void *FindInCollection( collection c, void *key ) {
/* Find an item in a collection
  Pre-condition:
        c is a collection created by ConsCollection
        c is sorted in ascending order of the key
        key != NULL
  Post-condition: returns an item identified by key if
  one exists, otherwise returns NULL
*/
        int low, high;
        low = 0; high = c->item_cnt-1;
        return bin_search( c, low, high, key );
}
```

Points to note:

a. bin_search is recursive: it determines whether the search key lies in the lower or upper half of the array, then calls itself on the appropriate half.
b. There is a termination condition (two of them in fact!)
    i. If low > high then the partition to be searched has no elements in it and
    ii. If there is a match with the element in the middle of the current partition, then we can return immediately.
c. AddToCollection will need to be modified to ensure that each item added is placed in its correct place in the array. The procedure is simple:
    i. Search the array until the correct spot to insert the new item is found,
    ii. Move all the following items up one position and
    iii. Insert the new item into the empty position thus created.
d. bin_search is declared static. It is a local function and is not used outside this class: if it were not declared static, it would be exported and be available to all parts of the program. The static declaration also allows other classes to use the same name internally.

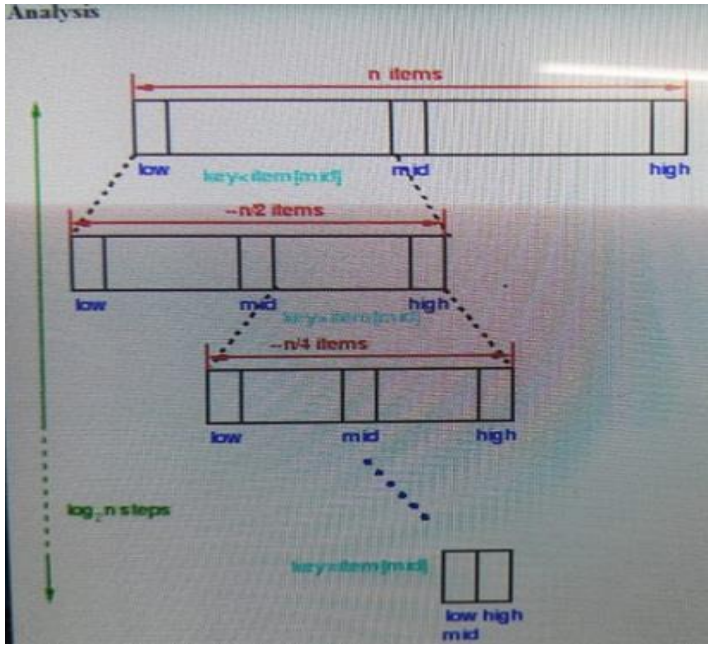
   static reduces the visibility of a function an should be used wherever possible to control access to functions!

Analysis

Each step of the algorithm divides the block of items being searched in half. We can divide a set of **n** items in half at most $\log_2 n$ times.

Thus the running time of a binary search is proportional to **log n** and we say this is a **O(log n)** algorithm.

Binary search requires a more complex program than our original search and thus for small n it may run slower than the simple linear search. However, for large n,



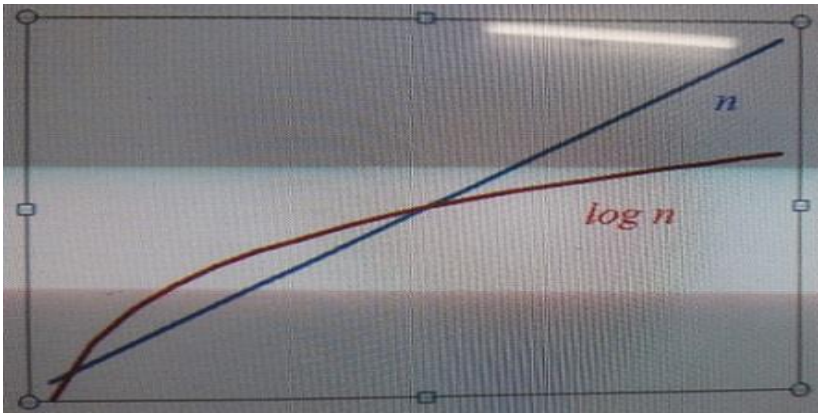
Plot
of n and log n vs n .

Thus at large n, log n is much smaller than n, consequently
an O(log n) algorithm is much faster than an O(n) one.

We will examine this behaviour more formally in a later section. First, let's see what we can do about the insertion (AddToCollection) operation.

In the worst case, insertion may require **n** operations to insert into a sorted list.

1. We can find the place in the list where the new item belongs using binary search in **O(log n)** operations.
2. However, we have to shuffle all the following items up one place to make way for the new one. In the worst case, the new item is the first in the list, requiring **n** move operations for the shuffle!



A similar analysis will show that deletion is also an **O(n)** operation.

If our collection is static, ie it doesn't change very often - if at all - then we may not be concerned with the time required to change its contents: we may be prepared for the initial build of the collection and the occasional insertion and deletion to take some time. In return, we will be able to use a simple data structure (an array) which has little memory overhead.

However, if our collection is large and dynamic, ie items are being added and deleted continually, then we can obtain considerably better performance using a data structure called a tree.

➢ What is a priority queue?

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- poll(): This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- add(2): This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- poll(): It will remove '2' element from the priority queue as it has the highest priority queue.
- add(5): It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

There are two types of priority queue:

- Ascending order priority queue: In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.

- Descending order priority queue: In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.

Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which INFO list contains the data elements, PRN list contains the priority numbers of each data element available in the INFO list, and LINK basically contains the address of the next node.

	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step.

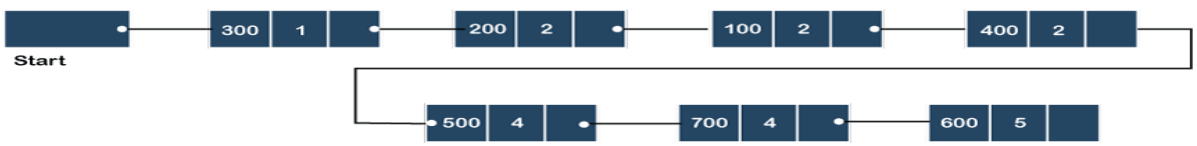
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing

the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

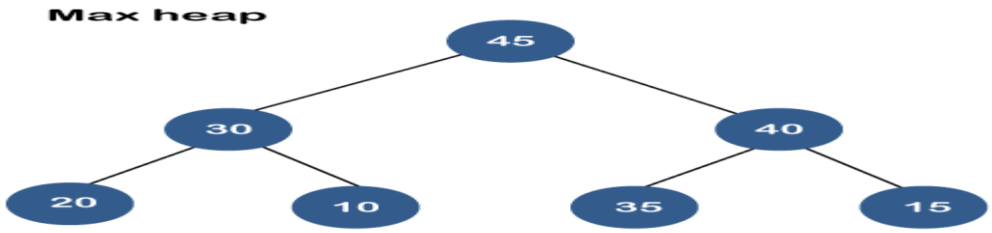
Analysis of complexities using different implementations

Implementation	add	Remove	peek
Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

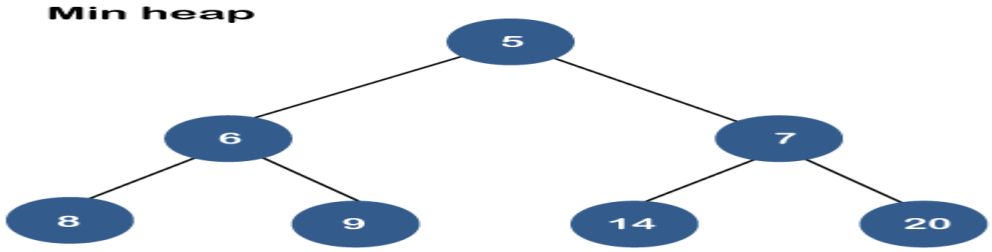
What is Heap?

A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

- Max heap: The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.



- Min heap: The min heap is a heap in which the value of the parent node is less than the value of the child nodes.



Both the heaps are the binary heap, as each has exactly two child nodes.

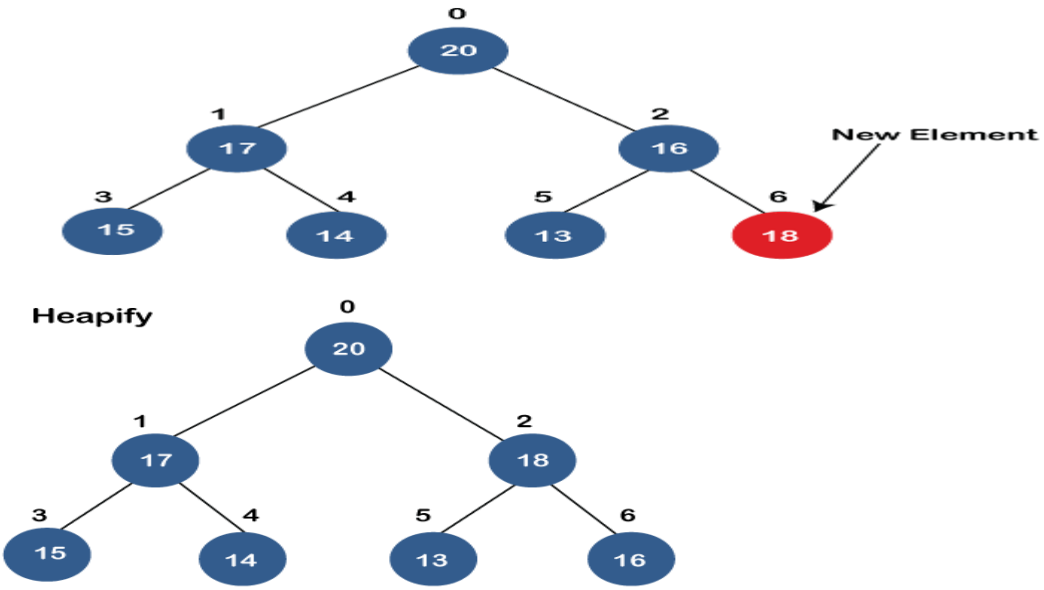
Priority Queue Operations

The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

- Inserting the element in a priority queue (max heap)

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.



- Removing the minimum element from the priority queue

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

➢ AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

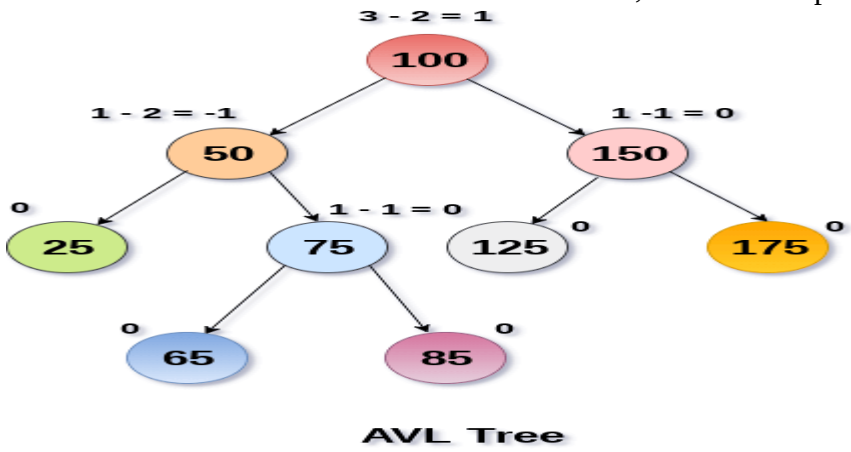
Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

- **Why AVL Tree?**

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is O(h). However, it can be extended to O(n) if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be O(log n) where n is the number of nodes.

AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A

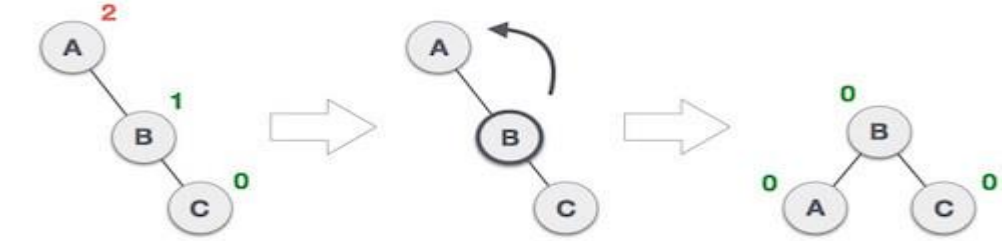
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

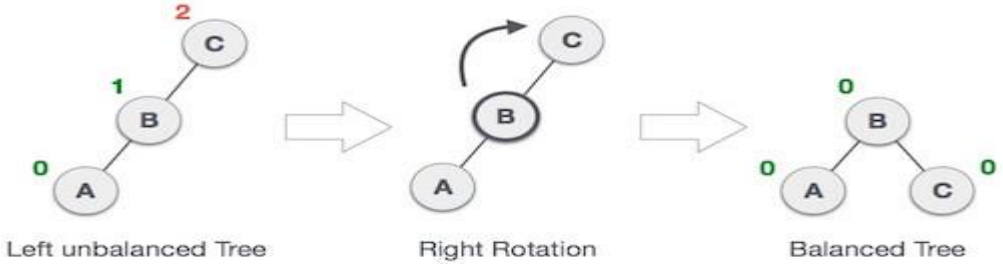
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

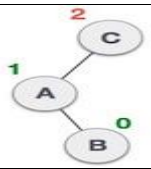


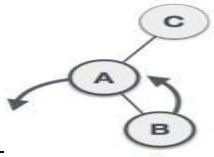
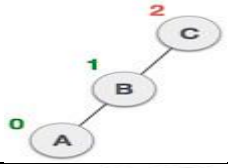
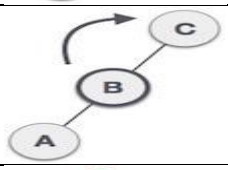
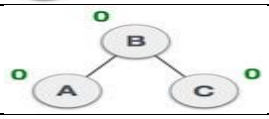
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

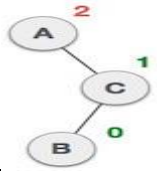
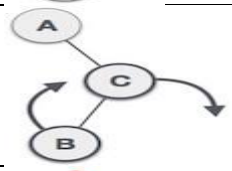
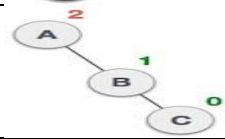
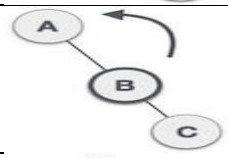
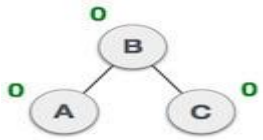
Let us understand each and every step very clearly:

State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C

	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

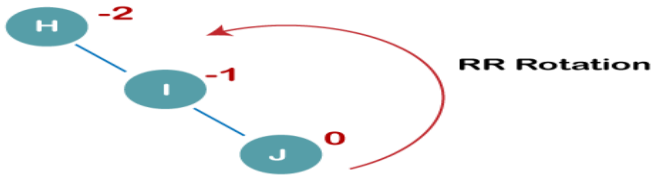
As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.
	After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

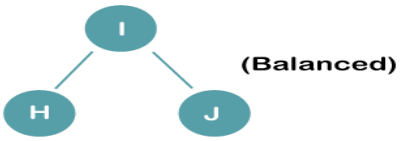
Q: Construct an AVL tree having the following elements

H, I, J, B, A, E, C, F, D, G, K, L

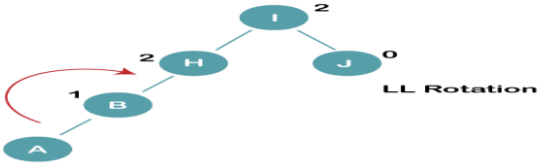
1. Insert H, I, J



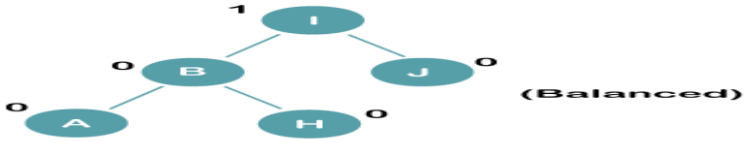
On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.
The resultant balance tree is:



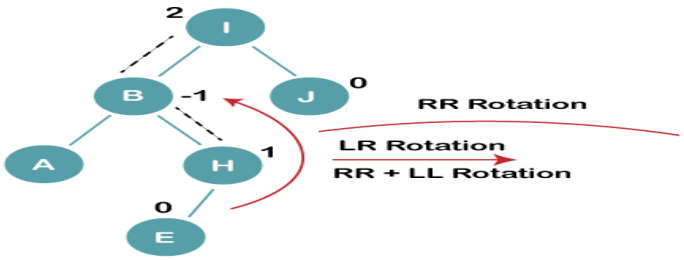
2. Insert B, A



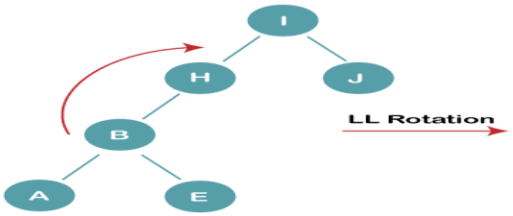
On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.
The resultant balance tree is:



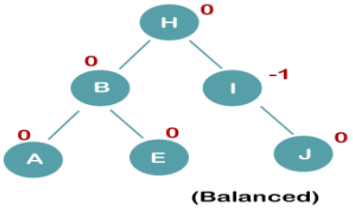
3. Insert E



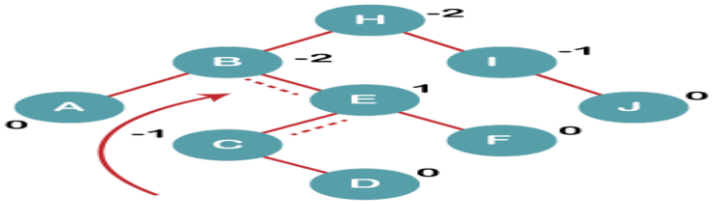
On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation
3 a) We first perform RR rotation on node B
The resultant tree after RR rotation is:



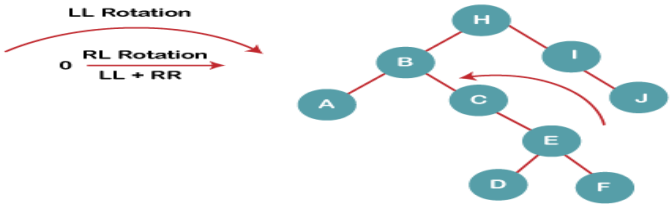
3b) We first perform LL rotation on the node I
The resultant balanced tree after LL rotation is:



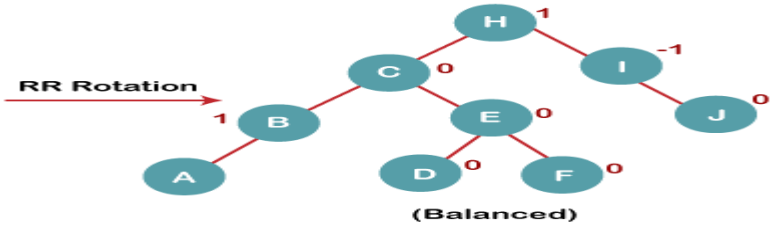
4. Insert C, F, D



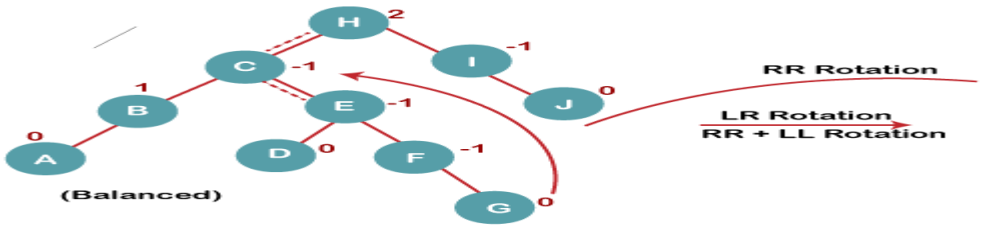
On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.
4a) We first perform LL rotation on node E
The resultant tree after LL rotation is:



4b) We then perform RR rotation on node B
The resultant balanced tree after RR rotation is:



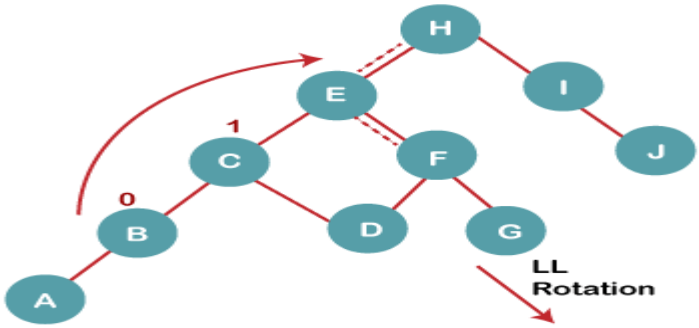
5. Insert G



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

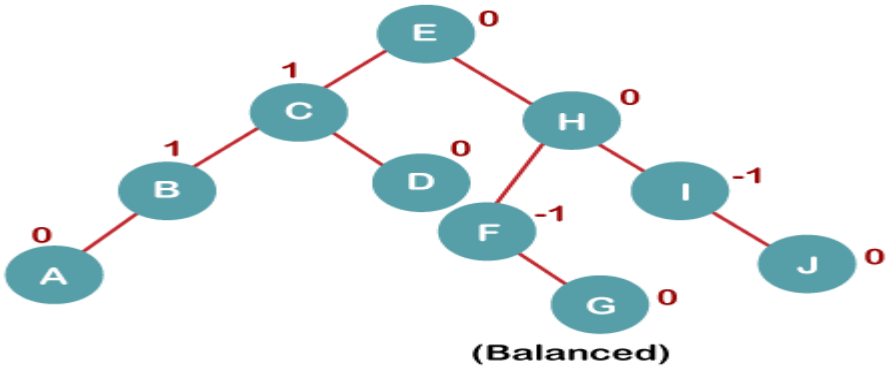
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:

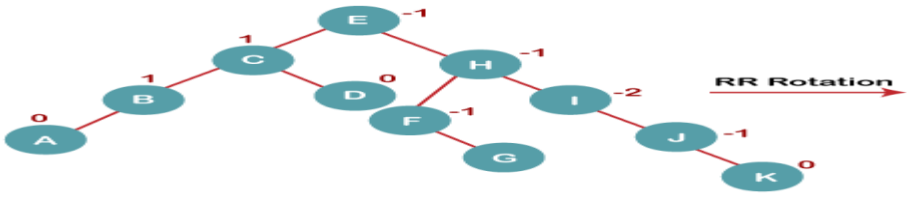


5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:

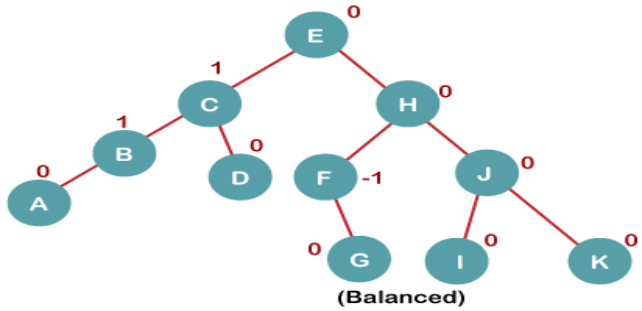


6. Insert K



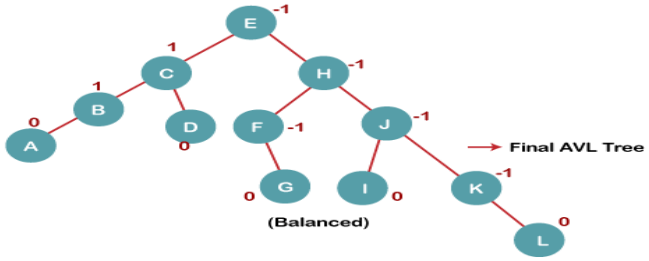
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



➢ Red –Black Trees

Introduction:

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black). These colours are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. This tree was invented in 1972 by Rudolf Bayer.

It must be noted that as each node requires only 1 bit of space to store the colour information, these types of trees show identical memory footprint to the classic (uncoloured) binary search tree.

Rules That Every Red-Black Tree Follows:

1. Every node has a colour either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes 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 the 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.

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)

2.	Insert	O(log n)
3.	Delete	O(log n)

"n" is the total number of elements in the red-black tree.

Comparison with AVL Tree:

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

How does a Red-Black Tree ensure balance?

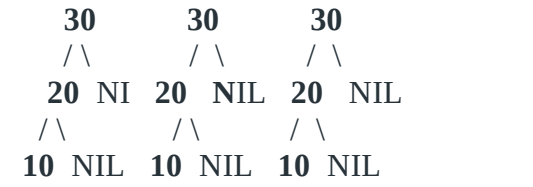
A simple example to understand balancing is, a chain of 3 nodes is not possible in the Red-Black tree. We can try any combination of colours and see all of them violate Red-Black tree property.

A chain of 3 nodes is not possible in Red-Black Trees.

Following are **NOT** Red-Black Trees

A chain of 3 nodes is not possible in Red-Black Trees.

Following are **NOT** Red-Black Trees



Violates Property 4. Violates Property 4 Violates Property 3

Following are different possible Red-Black Trees with above 3 keys

Interesting points about Red-Black Tree:

1. Black height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. Leaf nodes are also counted as black nodes. So, a red-black tree of height h has black height >= h/2.
2. Height of a red-black tree with n nodes is $h <= 2 \log_2(n + 1)$.
3. All leaves (NIL) are black.
4. The black depth of a node is defined as the number of black nodes from the root to that node i.e the number of black ancestors.
5. Every red-black tree is a special case of a binary tree.

Black Height of a Red-Black Tree :

Black height is the number of black nodes on a path from the root to a leaf. Leaf nodes are also counted black nodes. From the above properties 3 and 4, we can derive, **a Red-Black Tree of height h has black-height >= h/2**.

Number of nodes from a node to its farthest descendant leaf is no more than twice as the number of nodes to the nearest descendant leaf.

Every Red Black Tree with n nodes has height<= $2Log_2(n+1)$
This can be proved using the following facts:

1. For a general Binary Tree, let **k** be the minimum number of nodes on all root to NULL paths, then n >= $2^k - 1$ (Ex. If k is 3, then n is at least 7). This expression can also be written as k <= $Log_2(n+1)$.
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 of black nodes in a Red-Black tree is at least ⌊ n/2 ⌋ where n is the total number of nodes.

From the above points, we can conclude the fact that Red Black Tree with **n** nodes has height <= $2Log_2(n+1)$

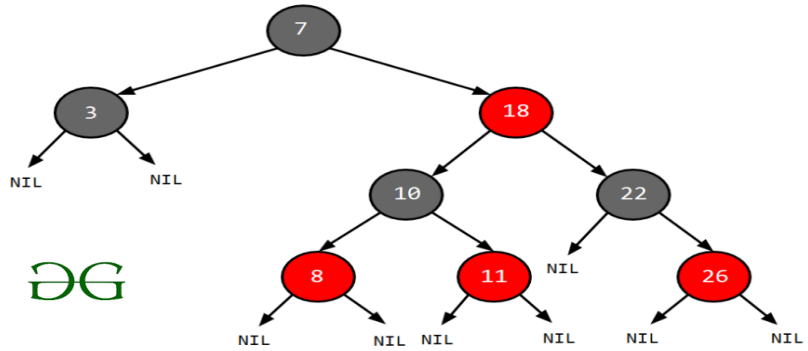
Search Operation in Red-black Tree:

As every red-black tree is a special case of a binary tree so the searching algorithm of a red-black tree is similar to that of a binary tree.

Algorithm:

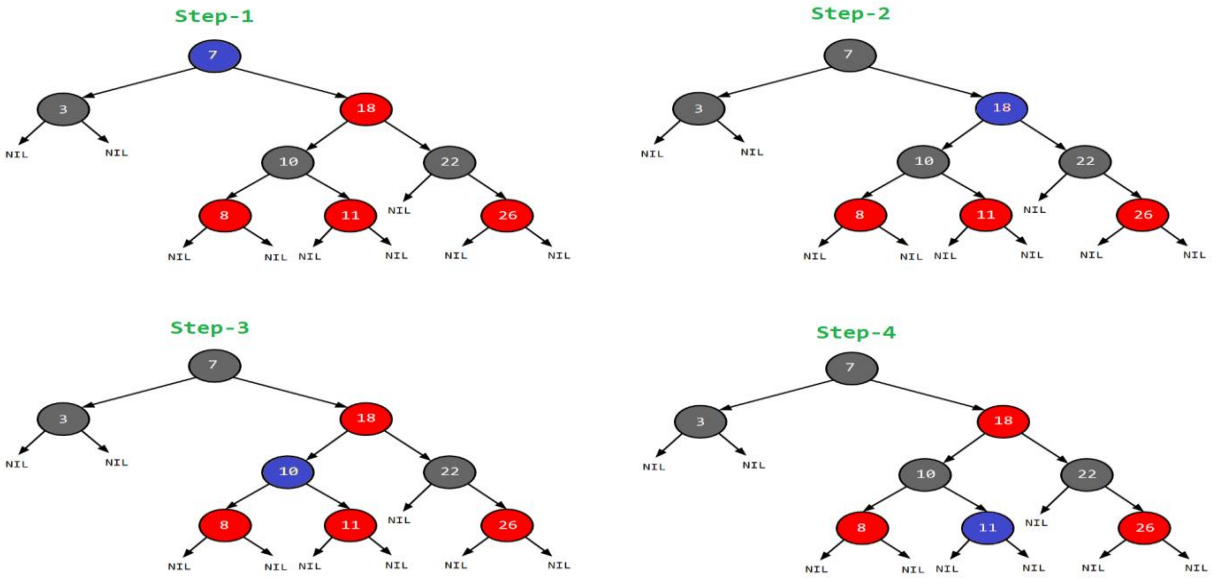
```
search Element (tree, val)
Step 1:
If tree -> data = val OR tree = NULL
   Return tree
Else
If val < data
      Return searchElement (tree -> left, val)
   Else
      Return searchElement (tree -> right, val)
   [ End of if ]
[ End of if ]
Step 2: END
```

Example: Searching 11 in the following red-black tree.



Solution:

1. Start from the root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. If the element to search is found anywhere, return true, else return false.



Just follow the blue bubble.

In this post, we introduced Red-Black trees and discussed how balance is ensured. The hard part is to maintain balance when keys are added and removed. We have also seen how to search an element from the red-black tree. We will soon be discussing insertion and deletion operations in coming posts on the Red-Black tree.

Insertion and Deletion

Red-BlackTree Insertion

Red-Black Tree Deletion

Applications:

1. Most of the self-balancing BST library functions like map and set in C++ (OR TreeSet and TreeMap in Java) use Red-Black Tree.
2. It is used to implement CPU Scheduling Linux. Completely Fair Scheduler uses it.
3. Besides they are used in the K-mean clustering algorithm for reducing time complexity.
4. Moreover, MySQL also uses the Red-Black tree for indexes on tables.

➢ Splay Trees

Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree. A splay tree is defined as follows...

Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.

Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations.

In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertion

operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

In splay tree, to splay any element we use the following rotation operations...

Rotations in Splay Tree

- **1. Zig Rotation**
- **2. Zag Rotation**
- **3. Zig - Zig Rotation**
- **4. Zag - Zag Rotation**
- **5. Zig - Zag Rotation**
- **6. Zag - Zig Rotation**

Example

Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...

Zag Rotation

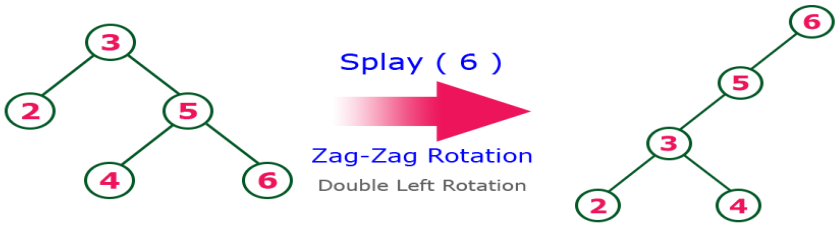
The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...

Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...

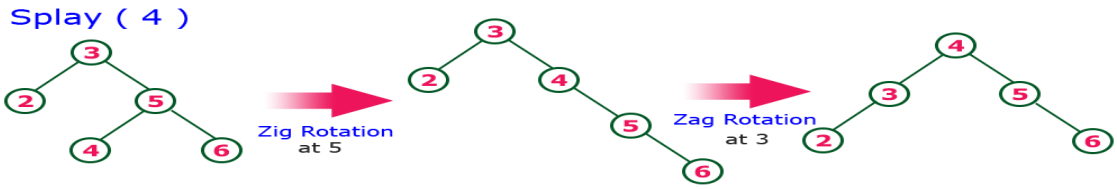
Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



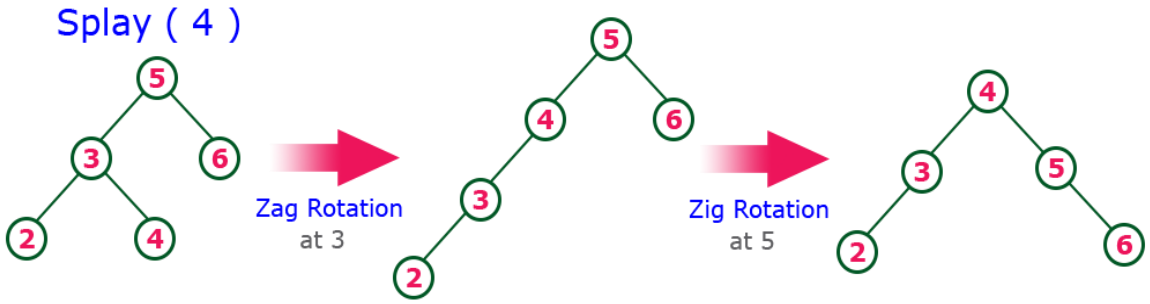
Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



Every Splay tree must be a binary search tree but it is need not to be balanced tree.

Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **newNode** as Root node and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
- **Step 4 -** After insertion, **Splay** the **newNode**

Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search logic.tree.

- Binary Heap

A **binary heap** is a data structure, which looks similar to a complete binary tree. Heap data structure obeys ordering properties discussed below. Generally, a Heap is represented by an array. In this chapter, we are representing a heap by **H**.

As the elements of a heap is stored in an array, considering the starting index as **1**, the position of the parent node of $\mathbf{i^{th}}$ element can be found at $\lfloor i/2 \rfloor$. Left child and right child of $\mathbf{i^{th}}$ node is at position **2i** and **2i + 1**.

A binary heap can be classified further as either a **max-heap** or a **min-heap** based on the ordering property.

Max-Heap

In this heap, the key value of a node is greater than or equal to the key value of the highest child.

Hence, **H[Parent(i)] ≥ H[i]**

Min-Heap

In mean-heap, the key value of a node is lesser than or equal to the key value of the lowest child.

Hence, **H[Parent(i)] ≤ H[i]**

In this context, basic operations are shown below with respect to Max-Heap. Insertion and deletion of elements in and from heaps need rearrangement of elements. Hence, **Heapify** function needs to be called.

Array Representation

A complete binary tree can be represented by an array, storing its elements using level order traversal.

Let us consider a heap (as shown below) which will be represented by an array **H**.

Considering the starting index as **0**, using level order traversal, the elements are being kept in an array as follows.

Considering the starting index as **0**, using level order traversal, the elements are being kept in an array as follows.

Index	0	1	2	3	4	5	6	7	8	...
elements	70	30	50	12	20	35	25	4	8	...

In this context, operations on heap are being represented with respect to Max-Heap.

To find the index of the parent of an element at index **i**, the following algorithm **Parent (numbers[], i)** is used.

Algorithm: Parent (numbers[], i)

```
if i == 1
  return NULL
else
  [i / 2]
```

The index of the left child of an element at index **i** can be found using the following algorithm, **Left-Child (numbers[], i)**.

Algorithm: Left-Child (numbers[], i)

```
If 2 * i ≤ heapsize
  return [2 * i]
```

else
  return NULL

The index of the right child of an element at index **i** can be found using the following algorithm, **Right-Child(numbers[], i)**.

Algorithm: Right-Child (numbers[], i)
if 2 * i < heapsize
  return [2 * i + 1]
else
  return NULL

To insert an element in a heap, the new element is initially appended to the end of the heap as the last element of the array.

After inserting this element, heap property may be violated, hence the heap property is repaired by comparing the added element with its parent and moving the added element up a level, swapping positions with the parent. This process is called **percolation up**.

The comparison is repeated until the parent is larger than or equal to the percolating element.

Algorithm: Max-Heap-Insert (numbers[], key)
heapsize = heapsize + 1
numbers[heapsize] = -∞
i = heapsize
numbers[i] = key
while i > 1 and numbers[Parent(numbers[], i)] < numbers[i]
  exchange(numbers[i], numbers[Parent(numbers[], i)])
  i = Parent (numbers[], i)

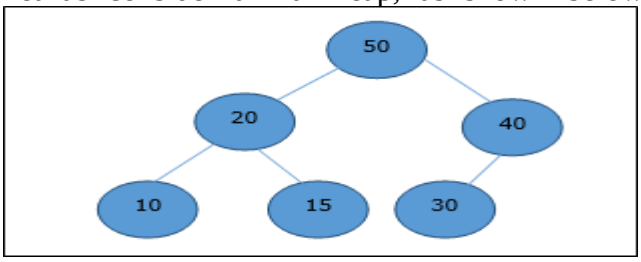
Analysis

Initially, an element is being added at the end of the array. If it violates the heap property, the element is exchanged performed.with its parent. The height of the tree is **log n**. Maximum **log n** number of operations needs to be

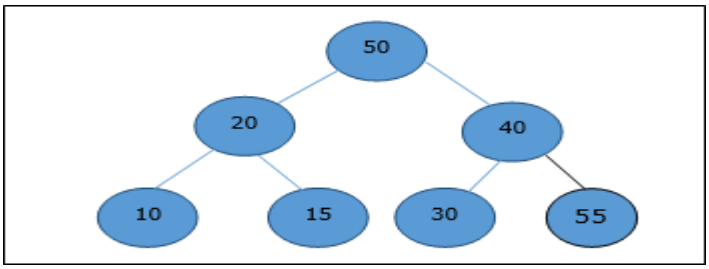
Hence, the complexity of this function is **O(log n)**.

Example

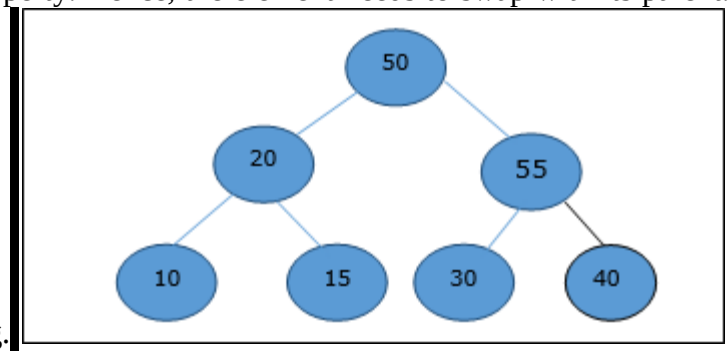
Let us consider a max-heap, as shown below, where a new element 5 needs to be added.



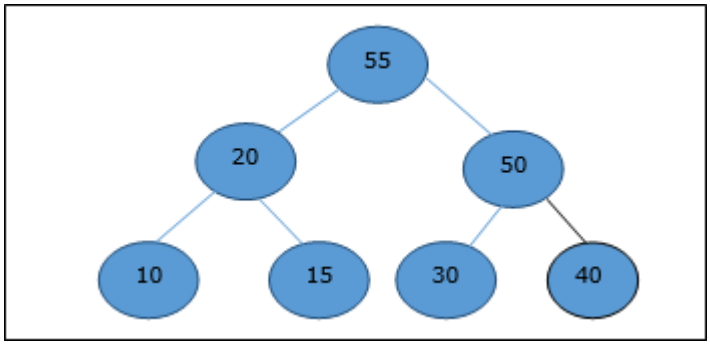
Initially, 55 will be added at the end of this array.



After insertion, it violates the heap property. Hence, the element needs to swap with its parent. After swap, the heap looks like the following.



Again, the element violates the property of heap. Hence, it is swapped with its parent.



Now, we have to stop.

Heapify method rearranges the elements of an array where the left and right sub-tree of **ith** element obeys the heap property.

Algorithm: Max-Heapify(numbers[], i)

```
leftchild := numbers[2i]
rightchild := numbers [2i + 1]
if leftchild ≤ numbers[].size and numbers[leftchild] > numbers[i]
   largest := leftchild
else
   largest := i
if rightchild ≤ numbers[].size and numbers[rightchild] > numbers[largest]
   largest := rightchild
if largest ≠ i
   swap numbers[i] with numbers[largest]
```

Max-Heapify(numbers, largest)

When the provided array does not obey the heap property, Heap is built based on the following algorithm **Build-Max-Heap (numbers[])**.

Algorithm: Build-Max-Heap(numbers[])

numbers[].size := numbers[].length
fori = ⌊ numbers[].length/2 ⌋ to 1 by -1
  Max-Heapify (numbers[], i)

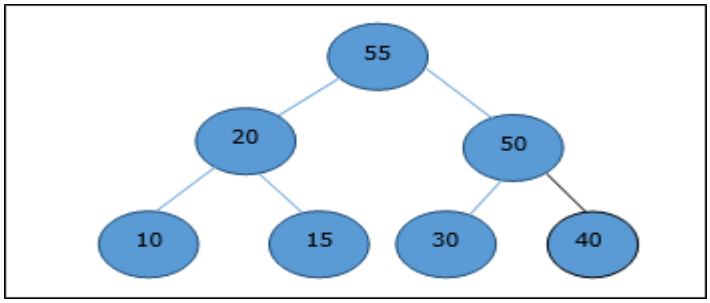
Extract method is used to extract the root element of a Heap. Following is the algorithm.

Algorithm: Heap-Extract-Max (numbers[])

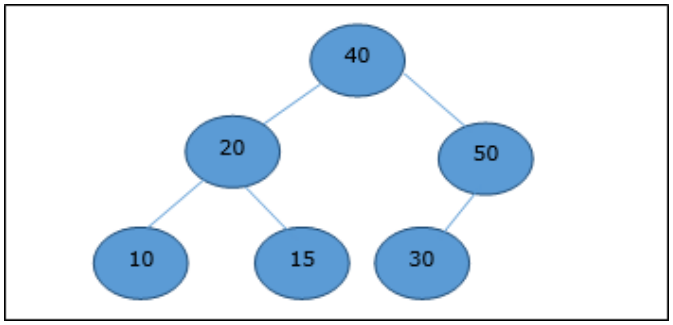
max = numbers[1]
numbers[1] = numbers[heapsize]
heapsize = heapsize – 1
Max-Heapify (numbers[], 1)
return max

Example

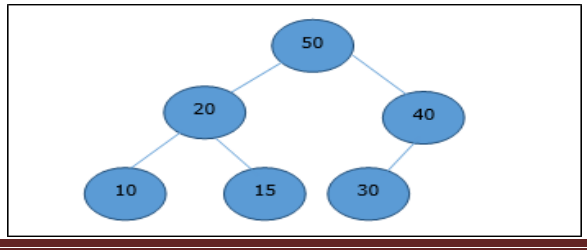
Let us consider the same example discussed previously. Now we want to extract an element. This method will return the root element of the heap.



After deletion of the root element, the last element will be moved to the root position.



Now, Heapify function will be called. After Heapify, the following heap is generated.



➢ Leftist Heap

A leftist tree or leftist heap is a priority queue implemented with a variant of a binary heap. Every node has an **s-value (or rank or distance)** which is the distance to the nearest leaf. In contrast to a binary heap (Which is always a complete binary tree), a leftist tree may be very unbalanced.

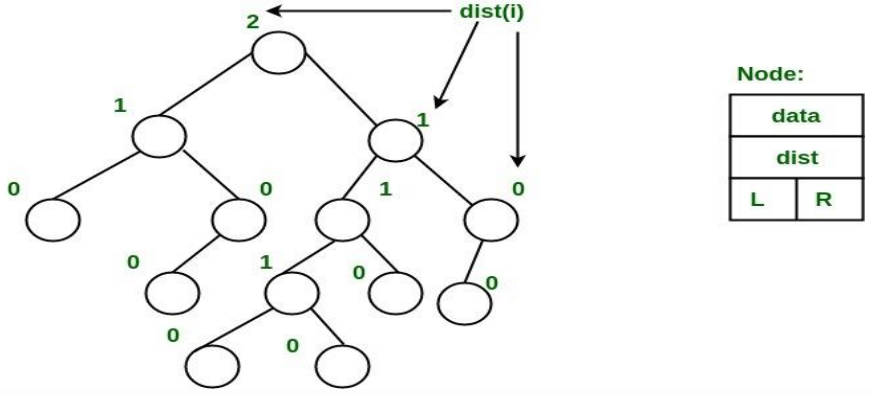
Below are time complexities of **Leftist Tree / Heap**.

Function	Complexity	Comparison
1) Get Min:	O(1)	[same as both Binary and Binomial]
2) Delete Min:	O(Log n)	[same as both Binary and Binomial]
3) Insert:	O(Log n)	[O(Log n) in Binary and O(1) in Binomial and O(Log n) for worst case]
4) Merge:	O(Log n)	[O(Log n) in Binomial]

A leftist tree is a binary tree with properties:

1. **Normal Min Heap Property :** key(i) >= key(parent(i))
2. **Heavier on left side :** dist(right(i)) <= dist(left(i)). Here, dist(i) is the number of edges on the shortest path from node i to a leaf node in extended binary tree representation (In this representation, a null child is considered as external or leaf node). The shortest path to a descendant external node is through the right child. Every subtree is also a leftist tree and dist(i) = 1 + dist(right(i)).

Example: The below leftist tree is presented with its distance calculated for each node with the procedure mentioned above. The rightmost node has a rank of 0 as the right subtree of this node is null and its parent has a distance of 1 by dist(i) = 1 + dist(right(i)). The same is followed for each node and their s-value(or rank) is calculated.



From above second property, we can draw two conclusions :

1. The path from root to rightmost leaf is the shortest path from root to a leaf.
2. If the path to rightmost leaf has x nodes, then leftist heap has atleast $2^x - 1$ nodes. This means the length of path to rightmost leaf is O(log n) for a leftist heap with n nodes.

Operations :

1. The main operation is merge().
2. deleteMin() (or extractMin() can be done by removing root and calling merge() for left and right subtrees.
3. insert() can be done be create a leftist tree with single key (key to be inserted) and calling merge() for given tree and tree with single node.

Idea behind Merging :

Since right subtree is smaller, the idea is to merge right subtree of a tree with other tree. Below are abstract steps.

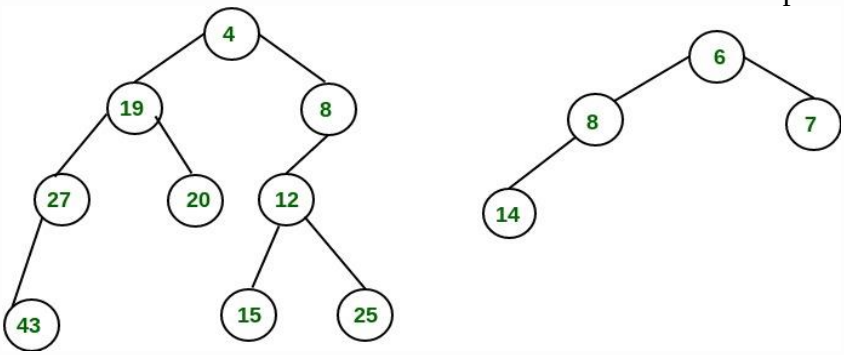
1. Put the root with smaller value as the new root.
2. Hang its left subtree on the left.
3. Recursively merge its right subtree and the other tree.
4. Before returning from recursion:
   – Update dist() of merged root.
   – Swap left and right subtrees just below root, if needed, to keep leftist property of merged result

Detailed Steps for Merge:

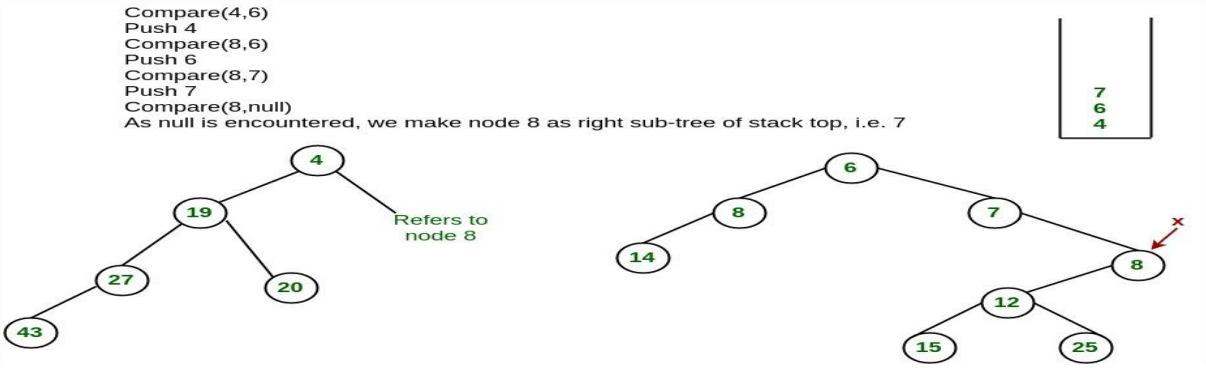
1. Compare the roots of two heaps.
2. Push the smaller key into an empty stack, and move to the right child of smaller key.
3. Recursively compare two keys and go on pushing the smaller key onto the stack and move to its right child.
4. Repeat until a null node is reached.
5. Take the last node processed and make it the right child of the node at top of the stack, and convert it to leftist heap if the properties of leftist heap are violated.
6. Recursively go on popping the elements from the stack and making them the right child of new stack top.

Example:

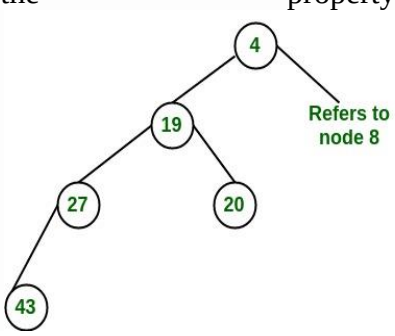
Consider two leftist heaps given below:

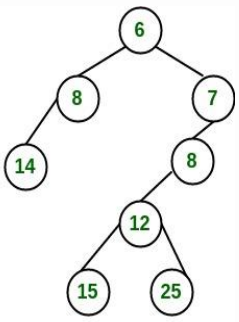


Merge them into a single leftist heap

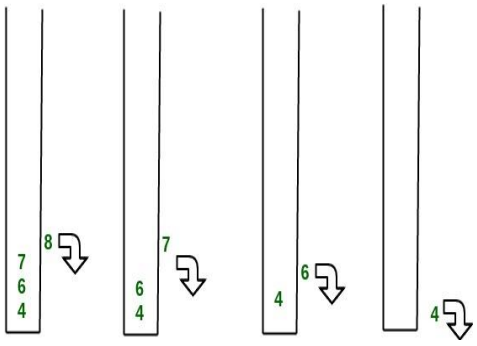


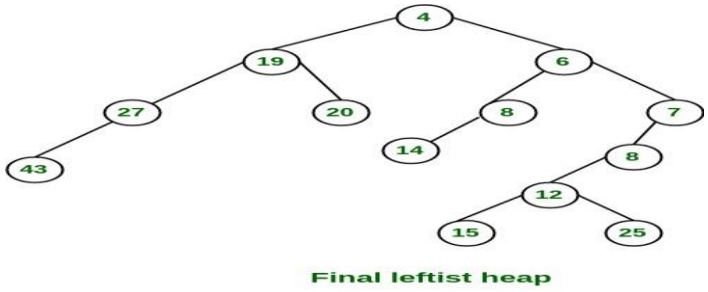
The subtree at node 7 violates the property of leftist heap so we swap it with the left child and retain the property of leftist heap.





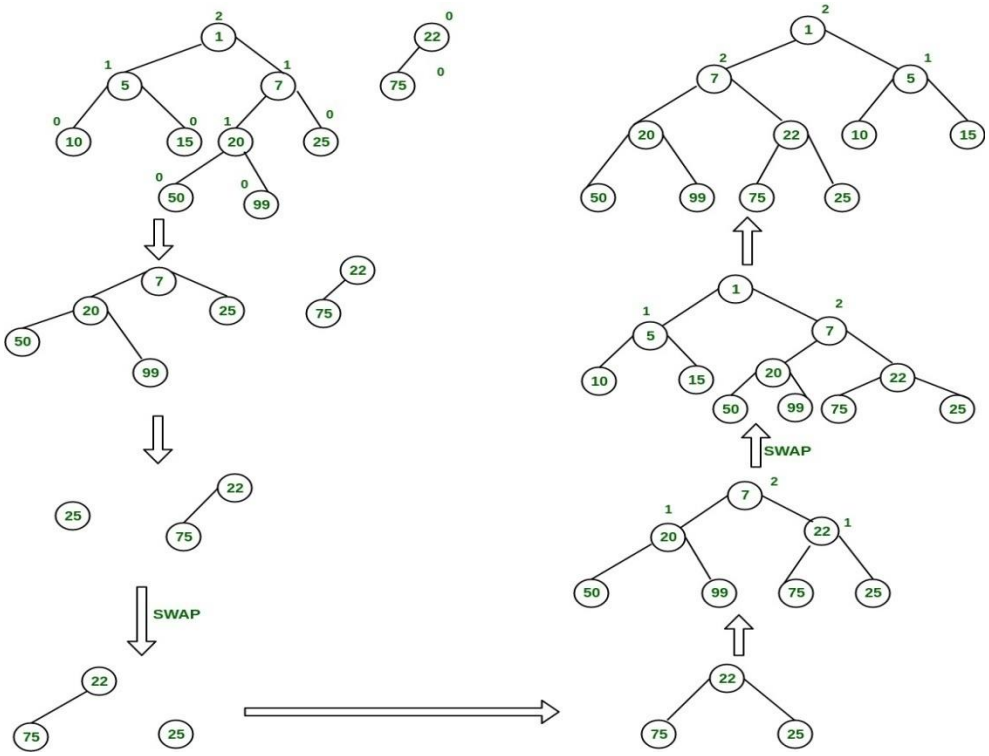
Convert to leftist heap. Repeat the process





The worst case time complexity of this algorithm is O(log n) in the worst case, where n is the number of nodes in the leftist heap.

Another example of merging two leftist heap:



➢ Implementation of Priority Queue ADT with Heap

Priority Queue

A priority queue is an ADT (Abstract Data Type) for maintaining a set S of elements, with each element having a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority and vice versa. If two elements have the same priority, they are served according to their order in the queue. It basically supports the following operations:

1. push(x): inserts an element x in set S – usually an O(log(n)) operation.
2. top() or peek(): returns the element of S with highest (or lowest) priority (but does not modify the queue) – O(1) operation.
3. pop(): returns and removes the element of S with highest (or lowest) priority – usually an O(log(n)) operation.

Heap Data Structure

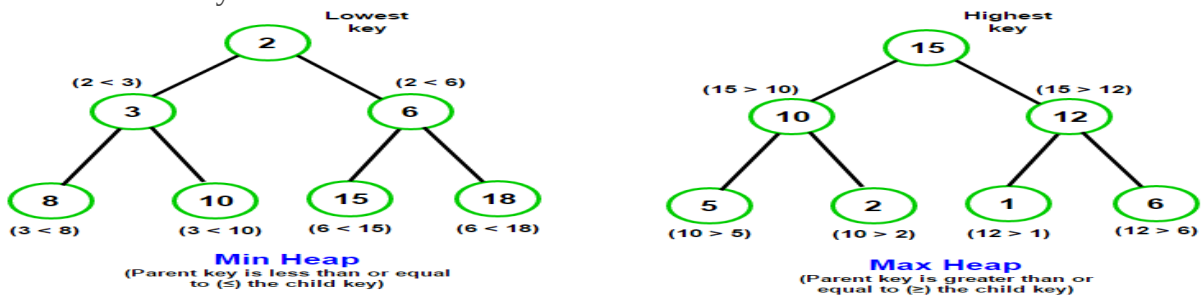
Heap data structure can be used to implement a priority queue. A heap data structure should not be confused with the heap, a pool of memory used for dynamic memory allocation. A common implementation of a heap is the binary heap, which is defined as a binary tree with two additional properties:

- Structural property: A binary heap is a complete binary tree, i.e., all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- Heap Property: The key stored in each node is either "greater than or equal to" or "less than or equal to" the keys in the node's children.

Min Heap and Max Heap

A heap can be classified further as either a "max-heap" or a "min-heap".

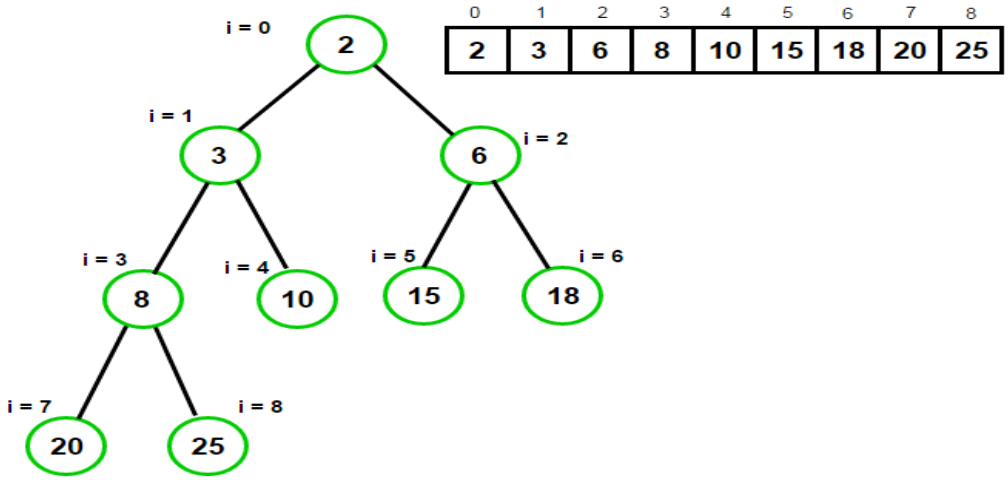
- In a max-heap, the keys of parent nodes are always greater than or equal to those of the children, and the highest key is in the root node.
- In a min-heap, the keys of parent nodes are less than or equal to those of the children, and the lowest key is in the root node.



The highest (or lowest) priority element is always stored at the root in the binary heap. Binary heaps have the smallest possible height of log(n), where n is the total number of nodes in a heap.

Operations on Heap

A binary heap is a complete binary tree, but we usually never use a binary tree for implementing heaps. We store keys in an array and use their relative positions within that array to represent child-parent relationships. The following diagram shows an array representing a min-heap:



The complete binary tree maps the binary tree structure into the array indices, where each array index represents a node. The index of the left or the right child of the parent node is simple expressions. For a zero-based array, the root node is stored at index 0. If i is the index of the current node, then,

PARENT (i) = floor((i – 1)/2)
LEFT-CHILD(i) = 2×i + 1
RIGHT_CHILD(i) = 2×i + 2
Min Heap Property: A[PARENT[i]] <= A[i]
Max Heap Property: A[PARENT[i]] >= A[i]

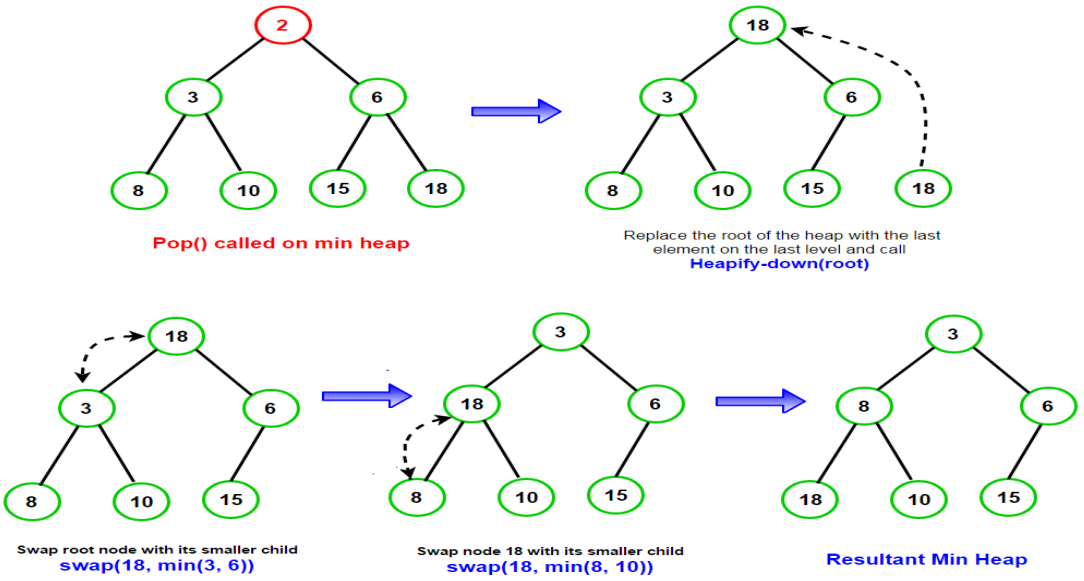
Heapify operation

Heapify operation forms the crux of all major heap operations. It can be implemented as heapify-up and heapify-down.

heapify-down (i) can be invoked if element A[i] violates the heap property with its two direct children. It converts the binary tree rooted at index i into a heap by moving A[i] down the tree.

It does so by comparing A[i] with its left & right child and swapping A[i] with the smaller child for min-heaps & the larger child for max-heaps, and then calling heapify-down on the corresponding child, i.e., heapify-down(LEFT_CHILD(i)) or heapify-down(RIGHT_CHILD(i)). The process is repeated till the heap property is fixed. The complexity of the heapify-down operation is O(log(n)).

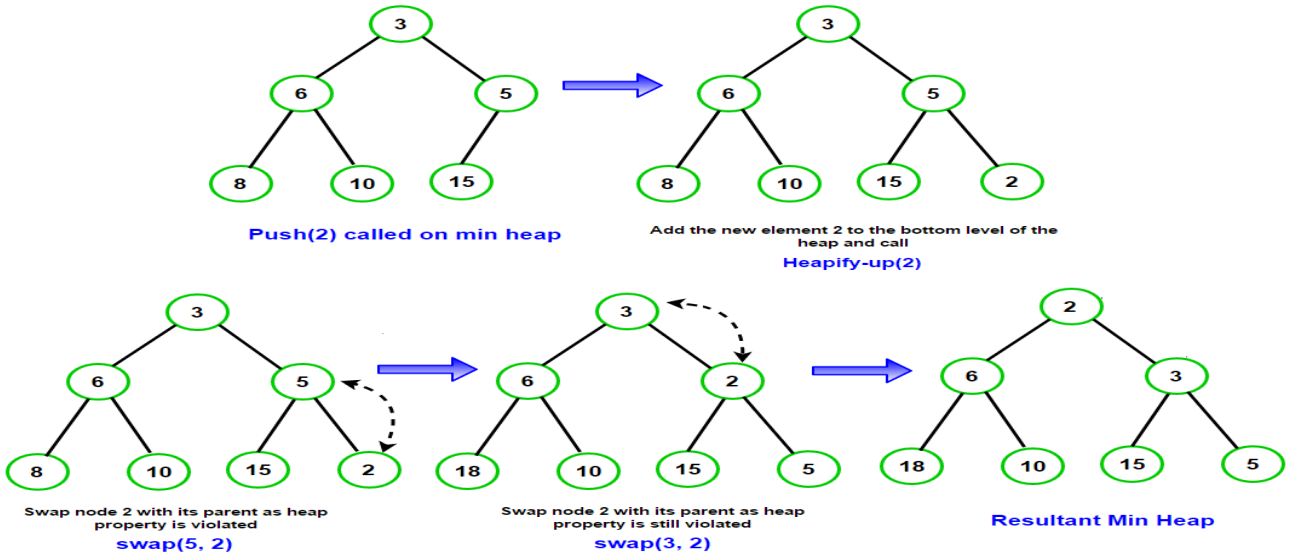
heapify-down is used in pop() operation of the binary heap. The idea is to replace the heap's root with the last element on the last level and call heapify-down on the root. The following diagram illustrates the process:



heapify-up(i) can be invoked if the parent of an element A[i] violates the heap property. It converts the binary tree into a heap by moving A[i] up the tree.

It does so by comparing A[i] with its parent and swapping the two if the heap property is violated. We then call heapify-up on the parent, i.e., heapify-up(PARENT(i)). The process is repeated till the heap property is fixed. The complexity of the heapify-up operation is O(log(n)).

heapify-up is used in push() operation of the binary heap. The idea is to add the new element to the heap's bottom level and call heapify-up on the last node. The following diagram illustrates the process:



Applications of Heap

Heaps are crucial in several efficient graph algorithms, such as Dijkstra's shortest path algorithm and Prim's algorithm for minimum spanning tree. They are also used in the Heapsort sorting algorithm and Huffman coding (A Data Compression technique).

UNIT-4

➢ Sorting algorithms

Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order. For example, consider an array A = {A1, A2, A3, A4, ?? An }, the array is called to be in ascending order if element of A are arranged like A1 > A2 > A3 > A4 > A5 > ? > An .

Consider an array;

int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9)

The Array sorted in ascending order will be given as:

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

There are many techniques by using which, sorting can be performed. In this section of the tutorial, we will discuss each method in detail.

➢ Insertion Sort

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

Technique

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k-1], A[k-2], and so on until we find an element A[j] such that, A[j]<=A[k].

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

Complexity

Complexity	**Best Case**	**Average Case**	**Worst Case**
Time	Ω(n)	θ(n^2)	o(n^2)
Space			o(1)

Algorithm

Step 1: Repeat Steps 2 to 5 for K = 1 to N-1
Step 2: SET TEMP = ARR[K]
Step 3: SET J = K - 1
Step 4: Repeat while TEMP <=ARR[J]
SET ARR[J + 1] = ARR[J]
SET J = J - 1
[END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
[END OF LOOP]
Step 6: EXIT

Java Program

```
public class InsertionSort {
1.  public static void main(String[] args) {
2.      int[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
3.      for(int k=1; k<10; k++)
4.      {
5.          int temp = a[k];
6.          int j= k-1;
7.          while(j>=0 && temp <= a[j])
8.          {
9.              a[j+1] = a[j];
10.             j = j-1;
11.         }
12.         a[j+1] = temp;
13.     }
14.     System.out.println("printing sorted elements ...");
15.     for(int i=0;i<10;i++)
16.     {
17.         System.out.println(a[i]);
18.     }
19. }
20. }
```

Output:

```
Printing sorted elements . . .
7
9
10
12
23
23
34
44
78
101
```

➢ Merge sort

Merge sort is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.

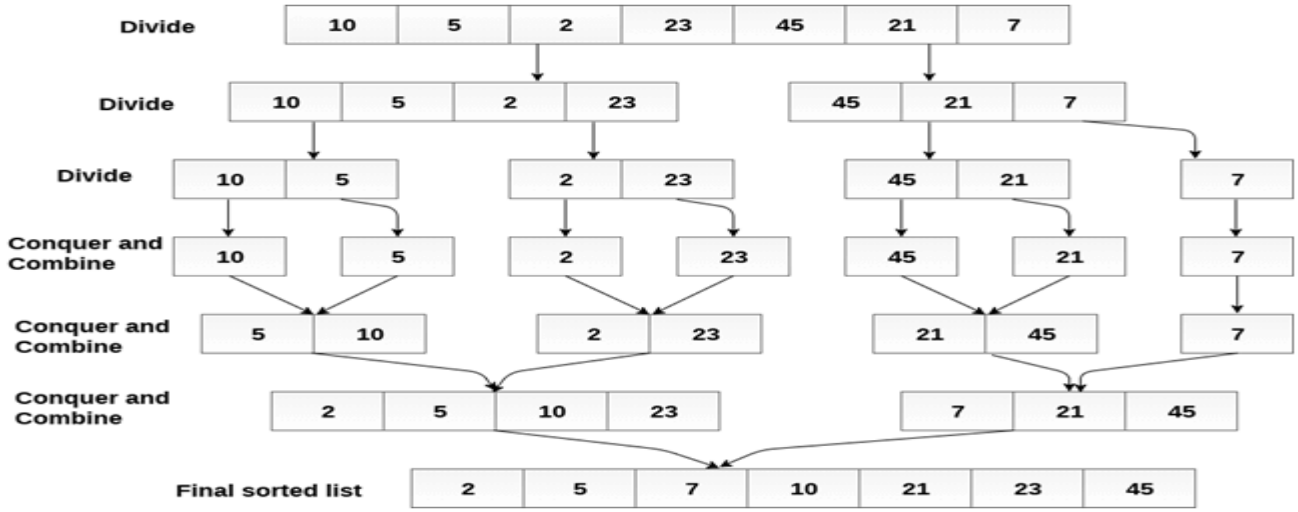
1. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
2. Conquer means sort the two sub-arrays recursively using the merge sort.
3. Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

The main idea behind merge sort is that, the short list takes less time to be sorted.

Complexity

Complexity	Best case	Average Case	Worst Case
Time Complexity	O(n log n)	O(n log n)	O(n log n)
Space Complexity			O(n)

Example :**Consider the following array of 7 elements. Sort the array by using merge sort.A = {10, 5, 2, 23, 45, 21, 7}**



Algorithm
step 1: [initialize] set i = beg, j = mid + 1, index = 0
step 2: repeat while (i <= mid) and (j<=end)if arr[i] < arr[j]set temp[index] = arr[i]
set i = i + 1 else
set temp[index] = arr[j]
set j = j + 1
[end of if]
set index = index + 1
[end of loop]
step 3: [copy the remaining
elements of right sub-array, if
any]
if i > mid
repeat while j <= end
set temp[index] = arr[j]
set index = index + 1, set j = j + 1
[end of loop]
[copy the remaining elements of
left sub-array, if any]
else
repeat while i <= mid
set temp[index] = arr[i]
set index = index + 1, set i = i + 1
[end of loop]
[end of if]
step 4: [copy the contents of temp back to arr] set k = 0

step 5: repeat while k < indexset arr[k] = temp[k] set k = k + 1
[end of loop]
step 6: exit
merge_sort(arr, beg, end) difference between jdk, jre, and jvm
step 1: if beg < end set mid = (beg + end)/2
call merge_sort (arr, beg, mid)
call merge_sort (arr, mid + 1, end)
merge (arr, beg, mid, end)
[end of if]
step 2: end
Java program

```
public class MyMergeSort
{
void merge(int arr[], int beg, int mid, int end)
{
 int l = mid - beg + 1;
int r = end - mid;
intLeftArray[] = new int [l];
intRightArray[] = new int [r];
for (int i=0; i<l; ++i)
LeftArray[i] = arr[beg + i];
for (int j=0; j<r; ++j)
RightArray[j] = arr[mid + 1+ j];
 int i = 0, j = 0;
int k = beg;
while (i<l&&j<r)
{
if (LeftArray[i] <= RightArray[j])
{
arr[k] = LeftArray[i];
i++;
}
else
{
arr[k] = RightArray[j];
j++;
}
k++;
}
while (i<l)
{
arr[k] = LeftArray[i];
i++;
k++;
}
while (j<r)
{
```

```
arr[k] = RightArray[j];
j++;
k++;
}
}
void sort(int arr[], int beg, int end)
{
if (beg<end)
{
int mid = (beg+end)/2;
sort(arr, beg, mid);
sort(arr , mid+1, end);
merge(arr, beg, mid, end);
}
}
public static void main(String args[])
{
intarr[] = {90,23,101,45,65,23,67,89,34,23};
MyMergeSort ob = new MyMergeSort();
ob.sort(arr, 0, arr.length-1);
System.out.println("\nSorted array");
for(int i =0; i<arr.length;i++)
{  System.out.println(arr[i]+"");  }}}
```

Output:

```
Sorted array
23
23
23
34
45
65
67
89
90
101
```

- Quick Sort

Quick sort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach. The algorithm processes the array in the following way.

1. Set the first index of the array to left and loc variable. Set the last index of the array to right variable. i.e. left = 0, loc = 0, en d = n - 1, where n is the length of the array.
2. Start from the right of the array and scan the complete array from right to beginning comparing each element of the array with the element pointed by loc.
   Ensure that, a[loc] is less than a[right].

1. If this is the case, then continue with the comparison until right becomes equal to the loc.
2. If a[loc] > a[right], then swap the two values. And go to step 3.
3. Set, loc = right

1. start from element pointed by left and compare each element in its way with the element pointed by the variable loc. Ensure that a[loc] > a[left]
    1. if this is the case, then continue with the comparison until loc becomes equal to left.
    2. [loc] < a[right], then swap the two values and go to step 2.
    3. Set, loc = left.

Complexity

Complexity	**Best Case**	**Average Case**	**Worst Case**
Time Complexity	O(n) for 3 way partition or O(n log n) simple partition	O(n log n)	$O(n^2)$
Space Complexity			O(log n)

Algorithm
partition (arr, beg, end, loc)
step 1: [initialize] set left = beg, right = end, loc = beg, flag =
step 2: repeat steps 3 to 6 while flag =
step 3: repeat while arr[loc] <=arr[right]
and loc != right
set right = right - 1
[end of loop]
step 4: if loc = right
set flag = 1
else if arr[loc] > arr[right]
swap arr[loc] with arr[right]
set loc = right
[end of if]
step 5: if flag = 0
repeat while arr[loc] >= arr[left] and loc != left
set left = left + 1
[end of loop]
step 6:if loc = left
set flag = 1
else if arr[loc] < arr[left]
swap arr[loc] with arr[left]
set loc = left
[end of if]
[end of if]
step 7: [end of loop]
step 8: end
quick_sort (arr, beg, end)
step 1: if (beg < end)
call partition (arr, beg, end, loc)
call quicksort(arr, beg, loc - 1)

```
call quicksort(arr, loc + 1, end)
[end of if]
step 2: end
public class QuickSort {
public static void main(String[] args) {
      int i;
      int[] arr={90,23,101,45,65,23,67,89,34,23};
      quickSort(arr, 0, 9);
      System.out.println("\n The sorted array is: \n");
      for(i=0;i<10;i++)
      System.out.println(arr[i]);
   }
   public static int partition(int a[], int beg, int end)
   {
      int left, right, temp, loc, flag;
      loc = left = beg;
      right = end;
      flag = 0;
      while(flag != 1)
      {
         while((a[loc] <= a[right]) && (loc!=right))
         right--;
         if(loc==right)
         flag =1;
         elseif(a[loc]>a[right])
         {
            temp = a[loc];
            a[loc] = a[right];
            a[right] = temp;
            loc = right;
         }
         if(flag!=1)
         {
            while((a[loc] >= a[left]) && (loc!=left))
            left++;
            if(loc==left)
            flag =1;
            elseif(a[loc] <a[left])
            {
               temp = a[loc];
               a[loc] = a[left];
               a[left] = temp;
               loc = left;
            }
         }
      }
```

```
        returnloc;
    }
    static void quickSort(int a[], int beg, int end)
    {
        int loc;
        if(beg<end)
        {
            loc = partition(a, beg, end);
            quickSort(a, beg, loc-1);
            quickSort(a, loc+1, end);
        }
    }
}
```

Output:

```
The sorted array is:
23
23
23
34
45
65
67
89
90
101
```

- Heap Sort

Heap sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

- Build a heap H, using the elements of ARR.
- Repeatedly delete the root element of the heap formed in phase 1.

Complexity

Complexity	Best Case	Average Case	Worst case
Time Complexity	Ω(n log (n))	θ(n log (n))	O(n log (n))
Space Complexity			O(1)

Algorithm

HEAP_SORT (ARR, N)
Step 1: [Build Heap H]
Repeat for i=0 to N-1
CALL INSERT_HEAP(ARR, N, ARR[i])
[END OF LOOP]
Step 2: Repeatedly Delete the root element
Repeat while N > 0
CALL Delete_Heap(ARR,N,VAL)

SET N = N+1
[END OF LOOP]
Step 3: END

```
// Java program for implementation of Heap Sort
public class HeapSort
{
        public void sort(int arr[])
        {
         int n = arr.length;

         // Build heap (rearrange array)
         for (int i = n / 2 - 1; i >= 0; i--)
                heapify(arr, n, i);

         // One by one extract an element from heap
         for (int i=n-1; i>=0; i--)
         {
                // Move current root to end
                int temp = arr[0];
                arr[0] = arr[i];
                arr[i] = temp;

                // call max heapify on the reduced heap
                heapify(arr, i, 0);
         }
        }

        // To heapify a subtree rooted with node i which is
        // an index in arr[]. n is size of heap
        void heapify(int arr[], int n, int i)
        {
         int largest = i; // Initialize largest as root
         int l = 2*i + 1; // left = 2*i + 1
         int r = 2*i + 2; // right = 2*i + 2

         // If left child is larger than root
         if (l < n && arr[l] > arr[largest])
                largest = l;

         // If right child is larger than largest so far
         if (r < n && arr[r] > arr[largest])
                largest = r;

         // If largest is not root
         if (largest != i)
         {
```

```
                int swap = arr[i];
                arr[i] = arr[largest];
                arr[largest] = swap;

                // Recursively heapify the affected sub-tree
                heapify(arr, n, largest);
            }
        }
        /* A utility function to print array of size n */
        static void printArray(int arr[])
        {
            int n = arr.length;
            for (int i=0; i<n; ++i)
                System.out.print(arr[i]+" ");
            System.out.println();
        }
        // Driver program
        public static void main(String args[])
        {
            int arr[] = {12, 11, 13, 5, 6, 7};
            int n = arr.length;

            HeapSort ob = new HeapSort();
            ob.sort(arr);
            System.out.println("Sorted array is");
            printArray(arr);
        }
    }
```

- Radix Sort

Radix sort processes the elements the same way in which the names of the students are sorted according to their alphabetical order. There are 26 radix in that case due to the fact that, there are 26 alphabets in English. In the first pass, the names are grouped according to the ascending order of the first letter of names.

In the second pass, the names are grouped according to the ascending order of the second letter. The same process continues until we find the sorted list of names. The bucket are used to store the names produced in each pass. The number of passes depends upon the length of the name with the maximum letter.

In the case of integers, radix sort sorts the numbers according to their digits. The comparisons are made among the digits of the number from LSB to MSB. The number of passes depend upon the length of the number with the most number of digits.

Complexity

Complexity	**Best Case**	**Average Case**	**Worst Case**
Time Complexity	Ω(n+k)	θ(nk)	O(nk)
Space Complexity			O(n+k)

Example

Consider the array of length 6 given below. Sort the array by using Radix sort.

A = {10, 2, 901, 803, 1024}

Pass 1: (Sort the list according to the digits at 0's place)

10, 901, 2, 803, 1024.

Pass 2: (Sort the list according to the digits at 10's place)

02, 10, 901, 803, 1024

Pass 3: (Sort the list according to the digits at 100's place)

02, 10, 1024, 803, 901.

Pass 4: (Sort the list according to the digits at 1000's place)

02, 10, 803, 901, 1024

Therefore, the list generated in the step 4 is the sorted list, arranged from radix sort.

algorithm

step 1:find the largest number in arr as large

step 2: [initialize] set nop = number of digits

in large

step 3: set pass =0

step 4: repeat step 5 while pass <= nop-1

step 5: set i = 0 and initialize buckets

step 6:repeat steps 7 to 9 while i<n-1< li=""></n-1<>

step 7: set digit = digit at passth place in a[i]

step 8: add a[i] to the bucket numbered digit

step 9: increment bucket count for bucket

numbered digit

[end of loop]

step 10: collect the numbers in the bucket

[end of loop]

step 11: end

Javaprogram

```
public class Radix_Sort {
public static void main(String[] args) {
  int i;
  Scanner sc = new Scanner(System.in);
  int[] a = {90,23,101,45,65,23,67,89,34,23};
  radix_sort(a);
  System.out.println("\n The sorted array is: \n");
  for(i=0;i<10;i++)
     System.out.println(a[i]);
}
  static int largest(inta[])
{
  int larger=a[0], i;
  for(i=1;i<10;i++)
  {
     if(a[i]>larger)
     larger = a[i];
```

```
        }
        returnlarger;
    }
    static void radix_sort(inta[])
    {
        int bucket[][]=newint[10][10];
        int bucket_count[]=newint[10];
        int i, j, k, remainder, NOP=0, divisor=1, larger, pass;
        larger = largest(a);
        while(larger>0)
        {
            NOP++;
            larger/=10;
        }
        for(pass=0;pass<NOP;pass++) // Initialize the buckets
        {
            for(i=0;i<10;i++)
            bucket_count[i]=0;
            for(i=0;i<10;i++)
            {
                // sort the numbers according to the digit at passth place
                remainder = (a[i]/divisor)%10;
                bucket[remainder][bucket_count[remainder]] = a[i];
                bucket_count[remainder] += 1;
            }
            // collect the numbers after PASS pass
            i=0;
            for(k=0;k<10;k++)
            {
                for(j=0;j<bucket_count[k];j++)
                {
                    a[i] = bucket[k][j];
                    i++;
                }
            }
            divisor *= 10;
        }
    }
}
```

Output:

```
The sorted array is:
23
23
23
34
45
```

```
65
67
89
90
101
```

➢ Analysis of different sorting techniques

Here we will discuss important properties of different sorting techniques including their complexity, stability and memory constraints.

Efficiency of an algorithm depends on two parameters:

1. Time Complexity
2. Space Complexity

Time Complexity: Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time taken also depends on some external factors like the compiler used, processor's speed, etc.

Space Complexity: Space Complexity is the total memory space required by the program for its execution.

Both are calculated as the function of input size(n).

One important thing here is that in spite of these parameters the efficiency of an algorithm also depends upon the **nature** and **size of** the **input.**

Following is a quick revision sheet that you may refer at last minute

Algorithm	Time Complexity		
	Best	**Average**	**Worst**
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	θ(nk)	O(nk)

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.

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.

Merge sort –

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

Heap sort –

Best, average and worst case time complexity: nlogn 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, $T(n) = O(nlogn)$

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.

➢ Sorting with disks

We will first illustrate merge sort using disks and then SORTING analyse it as an external sorting method.

Example

The file F containing 600 records is to be sorted. The main memory is capable of sorting of 1000 records at a time. The input file F is stored on one disk and we have in addition another scratch disk. The block lengthof the input file is 500 records.

We see that the file could be treated as 6 sets of 1000 recordes each. Each set is sorted and stored on thescratch disk as a 'run'. These 5 runs will then be merged as follows:

Allocate 3 blocks of memory each capable of holding 500 records. Two of these buffers B1 and B2 will be treated as input buffers and the third B3 as the output buffer. We have now the following.

1. 6 runs Rl, R2, R3, R4, R5, R6 on the scratch disk.
   3 buffers Bl, B2 and B3.
   Read 500 records from R1 into B1.
   Read 500 records from R2 into B2.
   Merge B1 and B2 and write into B3.
   When B3 is full - write it out to the disk as run R11.
   Similarly merge R3 and R4 to get run R12.
   Merge R5 and R6 to get run R13.

Thus, from 6 runs of size 1000 each, we have now 3 runs of size 2000 each. The steps are repeated for steps R11 and R12 to get a run of size 4000.

This run is merged with R13 to get a single sorted run of size 6000. Pictorially, this can be represented as:

Input file F with 6000 records

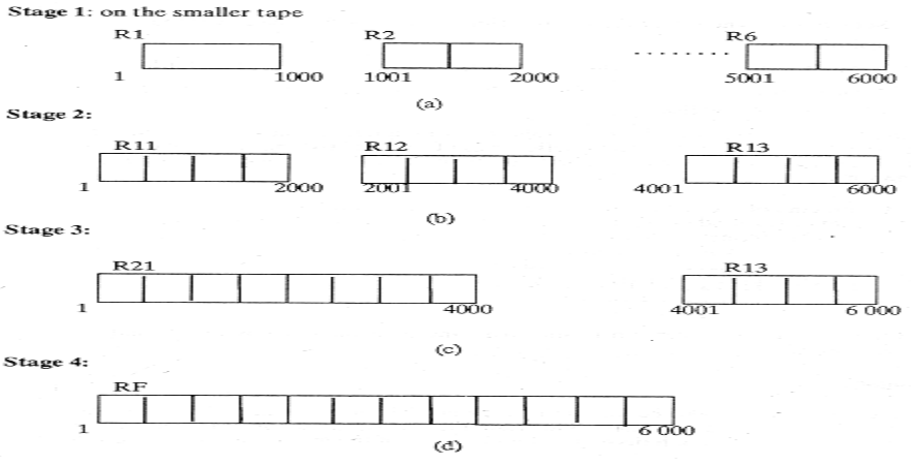


Figure 2: Merge Sort

The divisions in each run indicate the number of blocks.

Analysis

T1 - Seek time
T2 - Latency time
T3 - Transmission time for 1 block of 5000 records
T - T1 + T2 + T3
T4 - Time to internally sort 1000 records
nTM - Time to merge n records from input buffers to the output buffer.

In stage 1 we read 6000/500 = 12 blocks internally sort 6000/1000= 6 sets of 1000 records
write 6000/500 = 12 blocks
Therefore, time taken in stage 1 = 24T + 6T4 In stage 2 we read 12 blocks
write 12 blocks
Merge 5 x 2000 = 6000 records
Time taken in stage 2 = 24T + 600OTM In stage 3 we merge only 2 runs
Therefore we read 8 blocks
write 8 blocks
merge 2 x 2000 = 4000 records
Time taken in stage 3 = 16T + 400OTM In stage 4 we read 12 blocks
write 12 blocks
merge 4000 + 2000 = 4000 records
Therefore time taken in stage 4 = 24T + 600OTM

Total time taken is	=	24T + 6T4 + 24T 6000TM + 16T + 4000TM + 24T + 6000Tm
	=	88T + 6T4 + 1000Tm

It is seen that the largest influencing factor is TM, which depends on the number of passes made over thedata or the number of times runs must he combined.

We have assumed a uniform seek and latency time for all blocks for simplicity of analysis. This may not betrue in real life situations.

Time could also be reduced by exploiting the parallel features available, i.e. input/output and CPUprocessing carried out at the same time.

We will now focus on method to optimise the effects of the above factors, i.e. to say we will be carefully concerned with buffering and block size, assuming the internal sorting algorithms and the seek and latencytime factors are the best possible.

➢ K-Way Merging

In the above example, we used 2 way merging, i.e. combinations of two runs at a time. The number of passes over the data can be reduced by combining more runs at a time, hence the K-way merge where K2. In the game example, suppose we had used a 3 way merge then

at stage 2 we would have had 2 runs of size 3000 each

at stage 3 we would have had a single sorted run of size 6000 This would have affected our analysis as follows:

Stage 1 = 24T + 6T4
Stage 2 = 24T + 600OTM
Stage 3 = 24T + 600OTM
Total = 74T + 6T4 + 1200Otm

There is obviously an enoromous dip in the contribution of TM to the total time.

This advantage is accompanied by certain side effects. The time merge K runs , Would obviously be more than the time to merge 2 runs. Here the value of TM itself could out of K in each step of the merge is needed. One such method is the use of a selection tree.

A selection tree is a binary tree in which each node represents the smaller of its children. It therefore follows that the root will be the smallest node. The way it makes is simple, Initially, the selection tree is built from the lst item of each of the K runs. The one which gets to the root is selected as the smallest. Then the next item in the run from which the root was selected enters the tree and the tree gets restructured to get a new root and so on till the K runs are merged.

Example

Consider a selection tree for the following 4 runs:

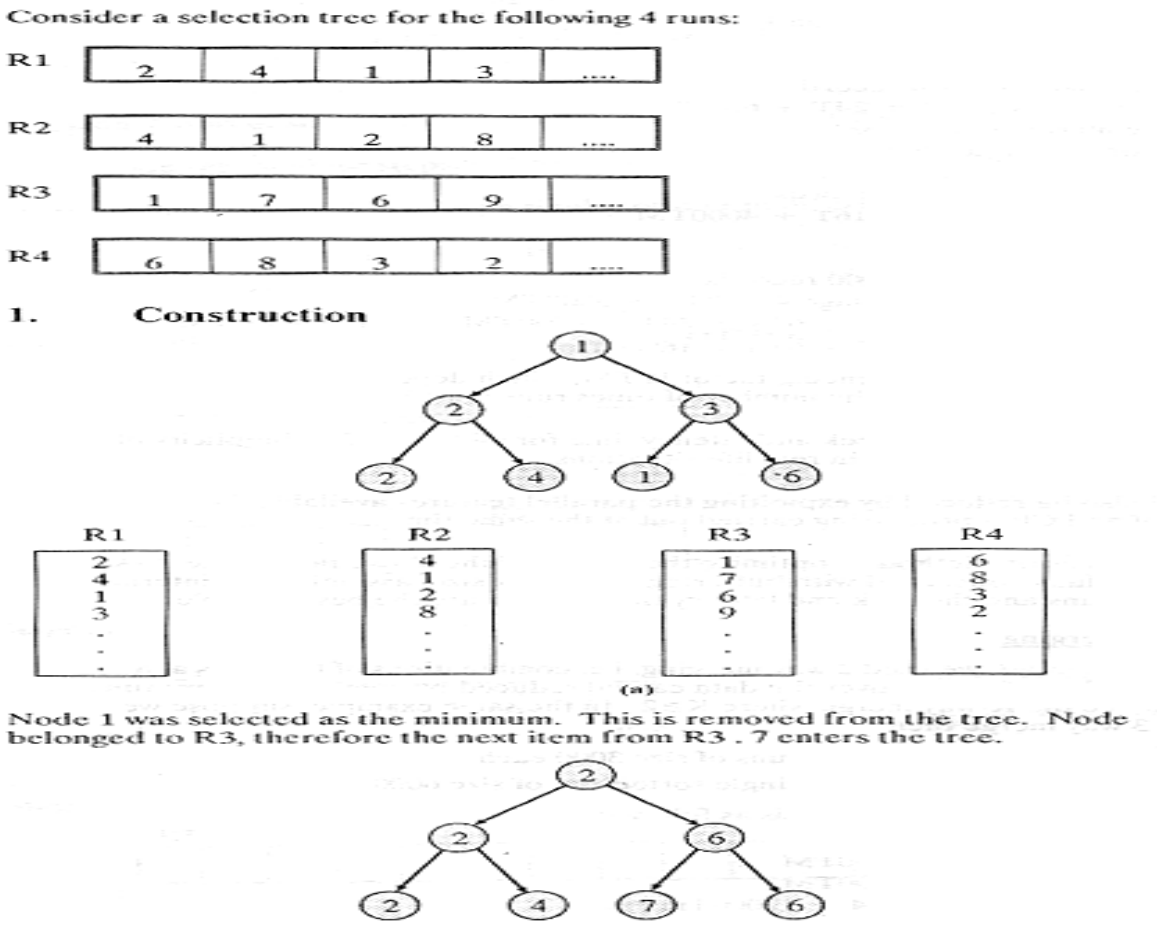
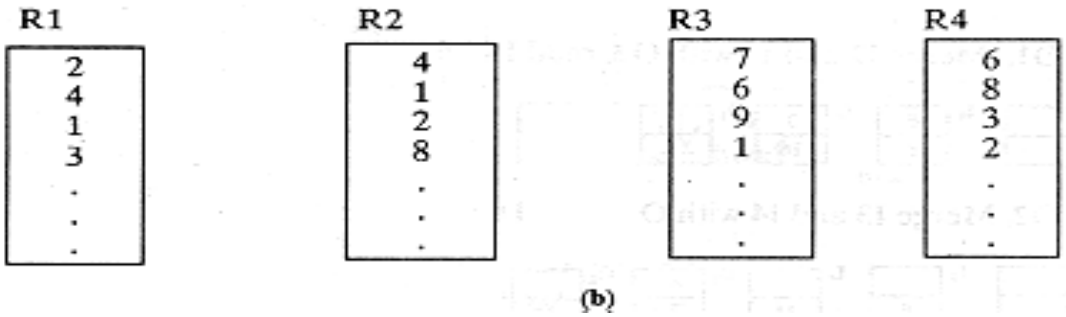


Figure 3: K-way Merging

The new root is 2. This came from R1 in the next step, the next item from Rl will enter the tree and so on. In going into a higher order merge, we save on input/output time. But with the increase in the order of the merge the number of buffers required also increases at the minimum to merge K runs we need K + 1 buffers. Now internal memory is a fixed entity, therefore if the number of buffers increases, there size must decrease. This implies a smaller block size on the disk and hence larger number of input/outputs. So beyond an optional value of K, we find that the advantage of reduced number of passes is offset by the disadvantage of increased input/output.



➢ Sorting With Tapes

Sorting with tapes is essentially similar to the merge sort used for sorting with disks. The differences arise due to the sequential access restriction of tapes. This makes the selection time prior to data transmission an important factor, unlike seek time and latency time. Thus in sorting with tapes we will be more concerned with arrangement of blocks and runs on the tape so as to reduce the selection or access time.

As in 6.3.4 we will use an example to determine the factors involved.

Example: A file of 6000 records is to be sorted. It is stored on a tape and the block length is 500. The main memory can sort upto a 1000 records at a time. We have in addition 4 search tapes T1 -T4 The steps in merging can be summarized as follows (figure 5):

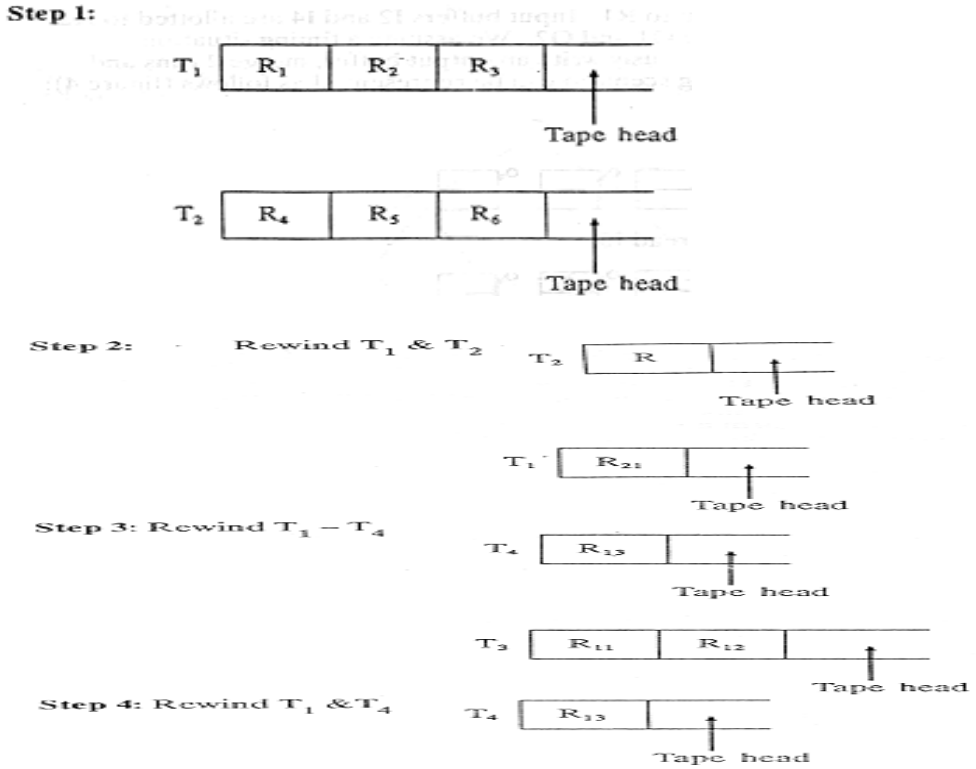


Figure 5: Sorting with Tapes

Analysis

t_{is} = time taken to internally sort 750 records.

t_{rw} = time to read or write. One block of 250 records. Onto tape starting from present position of tape read/write head.

t_{rew} = time to rewind tape over a length corresponding to one block.

nt_m = time to merge n records from input buffers to output buffers using a 2-way merge.

A = delay caused by having to wait for T_4 to be mounted in case we are ready to use Tn before it is mounted.

Total time = $6t_{is} + 132t_{rw} + 12000t_m + 51t_{rew} + A$

The above computing time analysis assumes that no operations are carried out in parallel. The analysis could be carried further or in the case of disks to show the dependence of sort time on the number of passes made on the data.

Balanced Merge Sorts:

We see that the computing time in the above example depends as, in the case of disk sorting, essentially on the number of passes being made over the data. Use of a higher order merge results in a decrease in the number of passes being made without significantly changing the internal merge time. Hence we would like to use as high an order merge as possible. In disk sorting, the order of merge was limited essentially by the amount of main memory available for input/output buffers. A k-way merge required the use of 2k + 2 buffers. Another more severe restriction on the merge order in the case of tapes is the number of tapes available. In order to avoid excessive seek time, it is necessary that runs being merged be on different tapes. Thus a k-way merge requires at least k-tapes for use as input tapes during the merge.

In addition, another tape is required for the output generated during the merge. Hence at least k + 1 tapes must be available for a k-way tape merge. Using k + 1 tapes for a k-way merge requires an additional pass over the output tape to redistribute the runs onto k-tapes for the next level of merge. This redistribution pass can be avoided by using 2k tapes. During the k-way merge, k of these tapes are used as input tapes and the remaining k as output tapes. At the next merge level, the role of input-output tapes is interchanged. These two approaches are now examined. Algorithms x1 and x2 perform a k-way merge with the k + 1 taoes , strategy while algorithm x3 performs a k-way merge using the 2k tapes strategy

```
Procedure x1;
 [sort a file of records from a given input tape
 using a k-way merge given tapes t1, ... t k+1
 are available for the sort.] label 200
 begin
 Create runs from the input tape distributing them
 evenly over tapes t1,..tk Rewind t1...tk and also at the
 input tape;
 If there is only one run goto 200; [the sorted
 file is on t1].replace input tape by tk+1;
while true do
 [repeatedly merge onto tk+1 and redistribute. back onto t1,tk].
begin
 merge runs from t1t k+1 onto tk+1;
 rewind t1,.....tk+1;
 If number of + runs on tk+1 = 1
 then goto 200;[output on tk+1]
 evenly distribute from tk+1 onto t1, ,tk;
 rewind t1. tk+1;
 end;
      200 end; [of x1]
```

Analysis: To simplify the analysis we assume that the number of runs generated(m) is a power of k. One pass over the entire file includes both reading and writing. The number of passes in the while loop is log k^m merge passes and log k^{m-1} redistribution. passes . . the total number of passes is 21og k^m. If the time to rewind the entire input tape is trew, then the non-overlapping rewind time is approximately 21ogk$^{m.trew}$

A cleverer way to tackle the problem is to rotate the output tape, i.e. tapes are used as output tapes in the cyclic order, K+ 1, 1, 2,..., k. This makes the redistribution from the output tape make less than a full pass over the file. Algorithm x2 describes the process.

```
Procedure x2
[Same definitions as for x1] label 200
begin
Create runs from the input file distributing them evenly over tapes t1,tk ;
Rewind t1,.., tk and also the input tape; If there is only one run then goto 200; [sorted file on t1]
   replace input tape by tk+1;
i = k + 1 ; i is index of the output tape] while true do
begin
merge runs from the k tapes tj; I < =j < = k + 1 and i onto ti; rewind t1     ,tk+1;
If number of runs on ti = 1 then goto 200 [output on t1]
evenly distribute (k-1)1k of the runs on ti onto
tape tj, k = j < = k + 1 and j i and j i mod (k + 1) + 1; rewind tapes tj, 1 < = j < = k + 1 and j i;
i: = i mod (k+ 1) + 1; end;
200 end; [end of x2]
```

Analysis: The only difference between algorithms x1 and x2 is in the redistributing time. m is the number of runs generated in line 5. Redistributing is done log k^{m-1} times, but each such pass reads and writes only (k-1)/k of the file. Hence the effective number of passes made over the data is (2-1/k) log k^m + 1/k. for two- way merging on 3 tapes algorithm x^2 will make 3/2 log k^m + 1/2 passes while x1 will make 21ogk^m passes. If trew is the rewind time then the non-overlapping rewind time for x^2 is utmost (1 + 1/k)(log k^m) trew +
(1-1/k) trew. Instead of distributing runs as shown we could write the first m/k runs on one tape, begin rewind, write the next m/k runs on the second tape, begin rewind etc. In this case we can begin filling input buffers for the next merge level while some of the tapes are still rewinding.

In case a k-way merge is using 2k tapes, no redistribution is needed and so the number of passes made is only logk^{m+1}. This implies that if 2k + 1 tapes are available than a 2k way merge will make (2-1/2k)log2k^m
+ 1/(2k) passes while a k-way merge utilizing 2k tapes will make only logk^{m+1} passes. The table compares the number of passes being made in the two methods for some value of k.

k	2k-way	k-way
1.	3/2 log2m + 1/2	-
2.	7/8 log2m + 1/4	log2M + 1
3.	1.124 log3m + 1/6	log3m + 1
4.	1.25 log4m + 1/8	log4m + 1

As is evident from the table, for k 2 a k-way merge using 2k tapes is better than a 2k-way merge using 2k +1 tapes.

Algorithm x3 makes a k-way merge sort using 2k tapes.

Procedure x3;

[sort a file of records from a given input tape using a k-way merge on 2k tapes]

begin

Create runs from the input file distributing them evenly over tapes t1 .. tk; rewind t1 ... tk; rewind the input tape;

replace the input tape by tape t2k; i: = 0; while total number of runs tik+1 ... t1k+k 1 do begin

j: = 1-i;

perform a k-way merge from tik+1,... tik+k evenly distributing output runs onto tjk+1,. tjk+k;

rewind t1 .. t2k;

i: = j ; [switch input and output tapes]

end;

[sorted file is on tik+1] end; [end of x3]

Analysis: To simplify the analysis, assume that m is a power of k. In addition to the initial run creation pass, the algorithm makes $\log_l m$ merge passes. Let trew be the time to rewind the entire input file. If m is a power of k then the time to rewind tapes $t_1,...t_{2k}$ in the while loop takes $t_{rew/k}$. for each but the last loop iteration. The last rewind takes time t_{rew} The total rewind time is therefore bound by $(2 + (\log_k m - 1)/k)t_{rew}$.

It should be noted that all the above algorithms use the buffering strategy developed in the k-way merge. The proper choice of buffer lengths and the merge order (restricted by the number of tapes available) would result in an almost complete overlap of internal processing with input/output time. At the end of each level of merge, processing will have to wait till the tapes rewind. This wait can be minimized using the run distribution strategy developed in algorithm x2.

➢ Polyphase merging:

The problem with balanced multiway merging is that it requires either an excessive number of tape units or excessive copying polyphase merging is a method to eliminate virtually all the copying by changing the way in which the small sorted blocks are merged together. The basic idea is to distribute the sorted blocks produced by replacement selection somewhat unevenly among the available tape units (learning one empty) and then to apply a 'merge = until-empty' strategy at which point one of theinput tapes and the output tape switch roles.

For example suppose that we have just 3 tapes, and we start with the initial configuration of sorted blocks on the tapes as shown at the top of the figure. Tape 3is initially empty the output tape for the first merges.

Tape 1 B P S T U-J O - B H O -E F N S - H J O.
2 FHY - BNQ - FM.
3 .

Tape 1 EFNS.HJO.
2 .
3 B F H P S T U Y-B J N O Q. B F H M O.

Tape 1 .
2 B E F F H N P S S T U Y- B H J J N O O Q.
3 B F H M O.

Now after three two way merges from tapes 1 and 2 to tape 3, the second tape becomes empty. Then after two two-way merges from tapes 1 and 3 onto tape 2, the first tape becomes empty. The sort is completed in
two more steps. First a 2-way merge from tapes 2 and 3 onto tape 1 leaves one file on tape 1, one file on tape 2.Then a two-way merge from tapes 1 and 2 leaves the entire sorted file on tape 3.

This merge - until-empty strategy can be extended to work for an arbitrary number of tapes. The figure indicates how 6 tapes might be used to sort with 497 initial runs. If we start as shown with tape 2 the output tape, tape 1 having 61 initial runs etc, then after running a five-way "merge-until-empty", we have tape 1 empty, tape 2 with 61 runs etc. as shown in the second column of the figure. At this point we can rewind tape 1 and make it the output tape and rewind tape 2 and make it an input tape. Continuing in this way we arrive at the entire file sorted on tape 1 as shown by the last column. The merge is broken up into many phases which don't involve all the data, but no direct copying is involved.

The main difficulty in implementing polyphase merge is to determine how to distribute the initial runs. The table a can be built by working backwards: take the largest number in each column, make it zero and add it to each of the other numbers to get the previous column. This technique works for any number of tapes (at least 3): the numbers which arise are "generalized Fibonacci numbers". Of course the number of initial runs may not be known in advance, and it probably won't be exactly a generalized Fibonacci number. Thus a number of "dummy" runs must be added to make the number of initial runs exactly what 'Is needed for the table.

Tape 1	61	0	31	15	7	3	1	0	1
Tape 2	0	61	30	14	6	2	0	1	0
Tape 3	120	59	28	12	4	0	2	1	0
Tape 4	116	59	24	8	0	4	2	1	0
Tape 5	108	47	16	0	8	4	2	1	0
Tape 6	92	31	0	16	8	4	2	1	0

Run distribution for sin-tape polyphase merge.

Remarks: A factor we have not considered is the time taken to rewind the tape. Before the merge for the next phase can begin, the tape must be rewound and the computer is essentially idle. It is possible to modify the above method so that virtually all rewind time is overlapped with internal processing and read/write on other tapes. However, the savings over the multiway balanced merge are quite limited. Even polyphase merging is better than balanced merging only for small P, and that not substantially. For P8, balanced merging is likely to run faster than polyphase and for smaller P, the effect of polyphase merging is to save two tapes.

UNIT-5

- Searching And Indexing:

Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted. The algorithm of linear search is given as follows

algorithm
linear_search(a, n, val)
step 1: [initialize] set pos = -1
step 2: [initialize] set i = 1
step 3: repeat step 4 while i<=n
step 4: if a[i] = val
set pos = i
print posgo to step 6
[end of if]
set i = i + 1
[end of loop]
step 5: if pos = -1
print " value is not presentin the array "
[end of if]
step 6: exit

Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

```
import java.util.Scanner;
public class Leniear_Search {
public static void main(String[] args) {
int[] arr = {10, 23, 15, 8, 4, 3, 25, 30, 34, 2, 19};
int item,flag=0;
Scanner sc = new Scanner(System.in);
System.out.println("Enter Item ?");
```

```
item = sc.nextInt();
for(int i = 0; i<10; i++)
{
   if(arr[i]==item)
   {
      flag = i+1;
      break;
   }
   else
      flag = 0;
}
if(flag != 0)
{
   System.out.println("Item found at location" + flag);
}
else
   System.out.println("Item not found"); }}
```

Output:

```
Enter Item ?
23
Item found at location 2
Enter Item ?
22
Item not found
```

- Binary Search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search algorithm is given below.

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)Step 1: [initialize] set beg = lower_bound
END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <=END
Step 3: SET MID = (BEG + END)/2
Step 4: IF A[MID] = VAL
SET POS = MID
PRINT POS
Go to Step 6
ELSE IF A[MID] > VAL
SET END = MID - 1
ELSE
SET BEG = MID + 1
[END OF IF]
[END OF LOOP]

Step 5: IF POS = -1
PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
[END OF IF]
Step 6: EXIT

Complexity

SN	Performance	Complexity
1	Worst case	O(log n)
2	Best case	O(1)
3	Average Case	O(log n)
4	Worst case space complexity	O(1)

Example

Let us consider an array arr = {1, 5, 7, 8, 13, 19, 20, 23, 29}. Find the location of the item 23 in the array.

In 1st step :

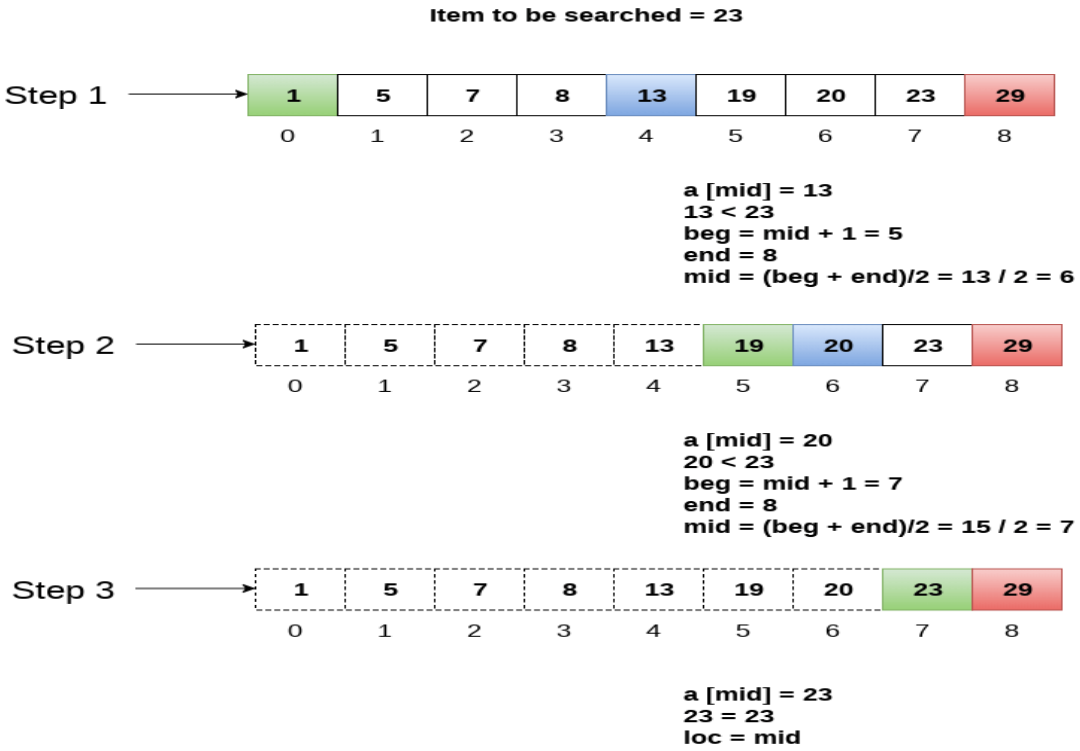
1. BEG = 0
2. END = 8ron
3. MID = 4
4. a[mid] = a[4] = 13 < 23, therefore

in Second step:

1. Beg = mid +1 = 5
2. End = 8
3. mid = 13/2 = 6
4. a[mid] = a[6] = 20 < 23, therefore;

in third step:

1. beg = mid + 1 = 7
2. End = 8
3. mid = 15/2 = 7
4. a[mid] = a[7]
5. a[7] = 23 = item;
6. therefore, set location = mid;
7. The location of the item will be 7.




```
import java.util.*;
public class BinarySearch {
public static void main(String[] args) {
   int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
   int item, location = -1;
   System.out.println("Enter the item which you want to search");
   Scanner sc = new Scanner(System.in);
   item = sc.nextInt();
   location = binarySearch(arr,0,9,item);
   if(location != -1)
   System.out.println("the location of the item is "+location);
   else
      System.out.println("Item not found");
   }
public static int binarySearch(int[] a, int beg, int end, int item)
{
   int mid;
   if(end >= beg)
   {
      mid = (beg + end)/2;
      if(a[mid] == item)
      {
         return mid+1;
```

```
    }
    else if(a[mid] < item)
    {
       return binarySearch(a,mid+1,end,item);
    }
    else
    {
       return binarySearch(a,beg,mid-1,item);
    } }
  return -1;   }}
```

Output:

```
Enter the item which you want to search
45
the location of the item is 5
```

➢ Hash Table

Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.

A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:

Hash(key)= index;

When we pass the key in the hash function, then it gives the index.

Hash(john) = 3;

The above example adds the john at the index 3.

Drawback of Hash function

A Hash function assigns each value with a unique key. Sometimes hash table uses an imperfect hash function that causes a collision because the hash function generates the same key of two different values.

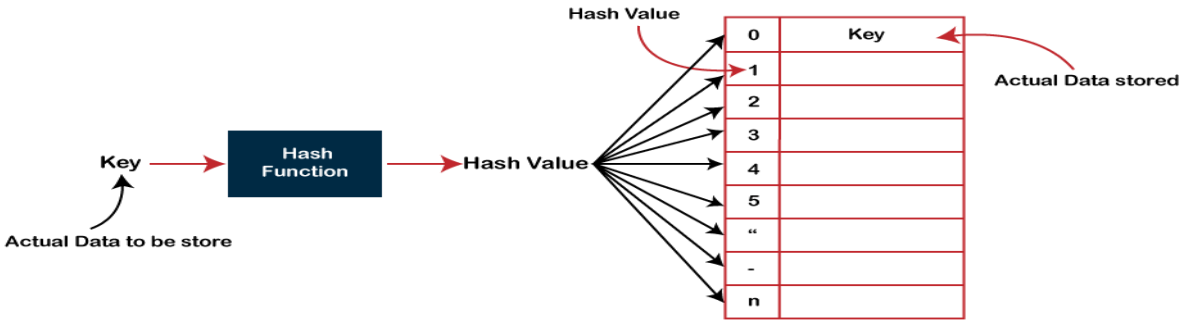
Hashing

Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1). Till now, we read the two techniques for searching, i.e., linear search and binary search. The worst time complexity in linear search is O(n), and O(logn) in binary search. In both the searching techniques, the searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique came that provides a constant time.

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:

Index = hash(key)



There are three ways of calculating the hash function:

- **Division method**
- **Folding method**
- **Mid square method**

In the division method, the hash function can be defined as:

$h(k_i) = k_i \% m;$

where **m** is the size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:

h(6) = 6%10 = 6

The index is 6 at which the value is stored.

Collision

When the two different values have the same value, then the problem occurs between the two values, known as a collision. In the above example, the value is stored at index 6. If the key value is 26, then the index would be:

h(26) = 26%10 = 6

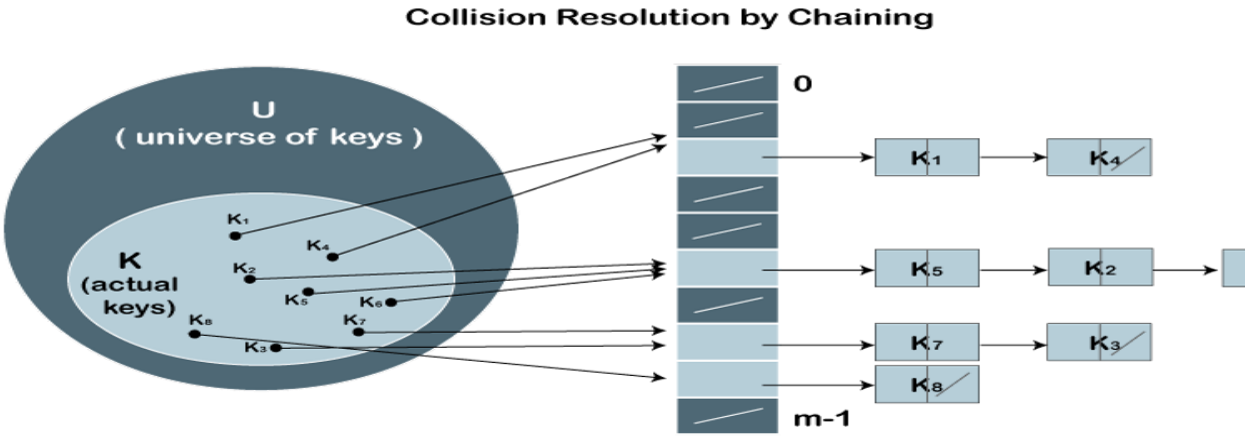
Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem. To resolve these collisions, we have some techniques known as collision techniques.

The following are the collision techniques:

- Open Hashing: It is also known as closed addressing.
- Closed Hashing: It is also known as open addressing.

Open Hashing

In Open Hashing, one of the methods used to resolve the collision is known as a chaining method.



Let's first understand the chaining to resolve the collision.
Suppose we have a list of key values
A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3
In this case, we cannot directly use $h(k) = k_i/m$ as h(k) = 2k+3

- The index of key value 3 is:

index = h(3) = (2(3)+3)%10 = 9
The value 3 would be stored at the index 9.

- The index of key value 2 is:

index = h(2) = (2(2)+3)%10 = 7
The value 2 would be stored at the index 7.

- The index of key value 9 is:

index = h(9) = (2(9)+3)%10 = 1
The value 9 would be stored at the index 1.

- The index of key value 6 is:

index = h(6) = (2(6)+3)%10 = 5
The value 6 would be stored at the index 5.

- The index of key value 11 is:

index = h(11) = (2(11)+3)%10 = 5
The value 11 would be stored at the index 5. Now, we have two values (6, 11) stored at the same index, i.e., 5. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 11 to this list. After the creation of the new list, the newly created list will be linked to the list having value 6.

- The index of key value 13 is:

index = h(13) = (2(13)+3)%10 = 9
The value 13 would be stored at index 9. Now, we have two values (3, 13) stored at the same index, i.e., 9. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 13 to this list. After the creation of the new list, the newly created list will be linked to the list having value 3.

- The index of key value 7 is:

index = h(7) = (2(7)+3)%10 = 7
The value 7 would be stored at index 7. Now, we have two values (2, 7) stored at the same index, i.e., 7. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 7 to this list. After the creation of the new list, the newly created list will be linked to the list having value 2.

- The index of key value 12 is:

index = h(12) = (2(12)+3)%10 = 7
According to the above calculation, the value 12 must be stored at index 7, but the value 2 exists at index 7. So, we will create a new list and add 12 to the list. The newly created list will be linked to the list having a value 7.

The calculated index value associated with each key value is shown in the below table:

key	**Location(u)**
3	**((2*3)+3)%10 = 9**
2	**((2*2)+3)%10 = 7**
9	**((2*9)+3)%10 = 1**
6	**((2*6)+3)%10 = 5**
11	**((2*11)+3)%10 = 5**

13	**((2*13)+3)%10 = 9**
7	**((2*7)+3)%10 = 7**
12	**((2*12)+3)%10 = 7**

Closed Hashing

In Closed hashing, three techniques are used to resolve the collision:

1. Linear probing
2. Quadratic probing
3. Double Hashing technique

Linear Probing

Linear probing is one of the forms of open addressing. As we know that each cell in the hash table contains a key-value pair, so when the collision occurs by mapping a new key to the cell already occupied by another key, then linear probing technique searches for the closest free locations and adds a new key to that empty cell. In this case, searching is performed sequentially, starting from the position where the collision occurs till the empty cell is not found.

Let's understand the linear probing through an example.

Consider the above example for the linear probing:

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5 respectively. The calculated index value of 11 is 5 which is already occupied by another key value, i.e., 6. When linear probing is applied, the nearest empty cell to the index 5 is 6; therefore, the value 11 will be added at the index 6.

The next key value is 13. The index value associated with this key value is 9 when hash function is applied. The cell is already filled at index 9. When linear probing is applied, the nearest empty cell to the index 9 is 0; therefore, the value 13 will be added at the index 0.

The next key value is 7. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 8; therefore, the value 7 will be added at the index 8.

The next key value is 12. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 2; therefore, the value 12 will be added at the index 2.

Quadratic Probing

In case of linear probing, searching is performed linearly. In contrast, quadratic probing is an open addressing technique that uses quadratic polynomial for searching until a empty slot is found.

It can also be defined as that it allows the insertion ki at first free location from **$(u+i^2)$%m where i=0 to m-1.**

Let's understand the quadratic probing through an example.

Consider the same example which we discussed in the linear probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5, respectively. We do not need to apply the quadratic probing technique on these key values as there is no occurrence of the collision.

The index value of 11 is 5, but this location is already occupied by the 6. So, we apply the quadratic probing technique.

When i = 0

Index= $(5+0^2)$%10 = 5

When i=1

Since location 6 is empty, so the value 11 will be added at the index 6.

The next element is 13. When the hash function is applied on 13, then the index value comes out to be 9, which we already discussed in the chaining method. At index 9, the cell is occupied by another value, i.e., 3. So, we will apply the quadratic probing technique to calculate the free location.

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

Index = $(5+1^2)\%10 = 6$

When i=0

Index = $(9+0^2)\%10 = 9$

When i=1

Index = $(9+1^2)\%10 = 0$

Since location 0 is empty, so the value 13 will be added at the index 0.

The next element is 7. When the hash function is applied on 7, then the index value comes out to be 7, which we already discussed in the chaining method. At index 7, the cell is occupied by another value, i.e., 7. So, we will apply the quadratic probing technique to calculate the free location.

When i=0

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

When i=1

Index = $(7+1^2)\%10 = 8$

Since location 8 is empty, so the value 7 will be added at the index 8.

The next element is 12. When the hash function is applied on 12, then the index value comes out to be 7. When we observe the hash table then we will get to know that the cell at index 7 is already occupied by the value 2. So, we apply the Quadratic probing technique on 12 to determine the free location.

When i=0

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

When i=1

Index = $(7+1^2)\%10 = 8$

When i=2

Index = $(7+2^2)\%10 = 1$

When i=3

Index = $(7+3^2)\%10 = 6$

When i=4

Index = $(7+4^2)\%10 = 3$

Since the location 3 is empty, so the value 12 would be stored at the index 3.

The final hash table would be:

Therefore, the order of the elements is 13, 9, _, 12, _, 6, 11, 2, 7, 3.

Double Hashing

Double hashing is an open addressing technique which is used to avoid the collisions. When the collision occurs then this technique uses the secondary hash of the key. It uses one hash value as an index to move forward until the empty location is found.

In double hashing, two hash functions are used. Suppose $h_1(k)$ is one of the hash functions used to calculate the locations whereas $h_2(k)$ is another hash function. It can be defined as "insert k_i at first free place from ***(u+v*i)%m*** where i=(0 to m-1)". In this case, u is the location computed

using the hash function and v is equal to (h_2(k)%m).
Consider the same example that we use in quadratic probing.
A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and
h_1(k) = 2k+3
h_2(k) = 3k+1

key	Location (u)	v	probe
3	**((2*3)+3)%10 = 9**	-	1
2	**((2*2)+3)%10 = 7**	-	1
9	**((2*9)+3)%10 = 1**	-	1
6	**((2*6)+3)%10 = 5**	-	1
11	**((2*11)+3)%10 = 5**	(3(11)+1)%10 =4	3
13	**((2*13)+3)%10 = 9**	(3(13)+1)%10 = 0	
7	**((2*7)+3)%10 = 7**	(3(7)+1)%10 = 2	
12	**((2*12)+3)%10 = 7**	(3(12)+1)%10 = 7	2

As we know that no collision would occur while inserting the keys (3, 2, 9, 6), so we will not apply double hashing on these key values.
On inserting the key 11 in a hash table, collision will occur because the calculated index value of 11 is 5 which is already occupied by some another value. Therefore, we will apply the double hashing technique on key 11. When the key value is 11, the value of v is 4.
Now, substituting the values of u and v in (u+v*i)%m
When i=0
Index = (5+4*0)%10 =5
When i=1
Index = (5+4*1)%10 = 9
When i=2
Index = (5+4*2)%10 = 3
Since the location 3 is empty in a hash table; therefore, the key 11 is added at the index 3.
The next element is 13. The calculated index value of 13 is 9 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 0.
Now, substituting the values of u and v in (u+v*i)%m
When i=0
Index = (9+0*0)%10 = 9
We will get 9 value in all the iterations from 0 to m-1 as the value of v is zero. Therefore, we cannot insert 13 into a hash table.
The next element is 7. The calculated index value of 7 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 2.
Now, substituting the values of u and v in (u+v*i)%m
When i=0
Index = (7 + 2*0)%10 = 7
When i=1
Index = (7+2*1)%10 = 9
When i=2
Index = (7+2*2)%10 = 1
When i=3

Index = (7+2*3)%10 = 3
When i=4
Index = (7+2*4)%10 = 5
When i=5
Index = (7+2*5)%10 = 7
When i=6
Index = (7+2*6)%10 = 9
When i=7
Index = (7+2*7)%10 = 1
When i=8
Index = (7+2*8)%10 = 3
When i=9
Index = (7+2*9)%10 = 5
Since we checked all the cases of i (from 0 to 9), but we do not find suitable place to insert 7. Therefore, key 7 cannot be inserted in a hash table.
The next element is 12. The calculated index value of 12 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 7.
Now, substituting the values of u and v in (u+v*i)%m
When i=0
Index = (7+7*0)%10 = 7
When i=1
Index = (7+7*1)%10 = 4
Since the location 4 is empty; therefore, the key 12 is inserted at the index 4.
The final hash table would be:

The order of the elements is _, 9, _, 11, 12, 6, _, 2, _, 3.

➢ Cylinder Surface Indexing

Cylinder surface indexing is a very simple technique used only for the primary key index of a sequentially ordered file.
The index file will contain two fields—cylinder index and several surface indices.

There are multiple cylinders, and each cylinder has multiple surfaces. If the file needs mcylinders for storage then the cylinder index will contain m entries
When a record with a particular key value has to be searched, then the following steps are

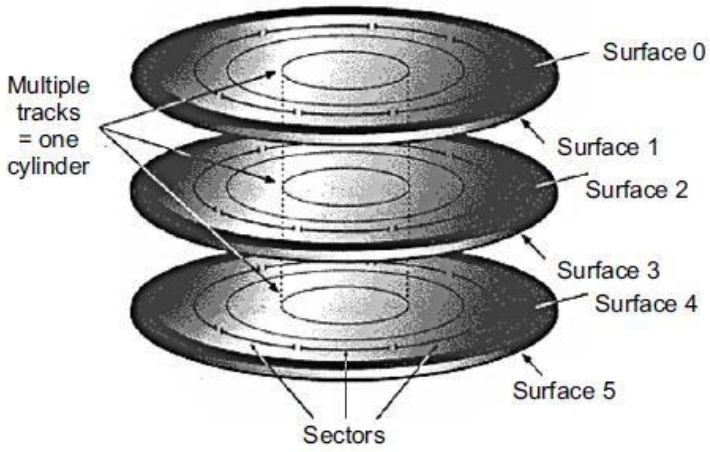


Figure Physical and logical organization of disk

performed:

- First the cylinder index of the file is read into memory.
- Second, the cylinder index is searched to determine which cylinder holds the desired record. For this, either the binary search technique can be used or the cylinder index can be made to store an array of pointers to the starting of individual key values. In either case the search will take O (log m) time.
- After the cylinder index is searched, appropriate cylinder is determined.
- Depending on the cylinder, the surface index corresponding to the cylinder is thenretrieved from the disk.
- Since the number of surfaces on a disk is very small, linear search can be used todetermine surface index of the record.
- Once the cylinder and the surface are determined, the corresponding track is read andsearched for the record with the desired key.

Hence, the total number of disk accesses is three—first, for accessing the cylinder index, second for accessing the surface index, and third for getting the track address.

➢ Hashed Indices

Hashing is used map to the same address, then collision occurs and schemes to resolvethese collisions are applied to generate a new address to compute the address of a record by using a hash function on the search keyvalue.
The hashed values

Choosing a **good hash function** is critical to the success of this technique. By a good hashfunction, it mean two things.

1. First, a good hash function, irrespective of the number of search keys, gives anaverage-case lookup that is a small constant.
2. Second, the function distributes records uniformly and randomly among the buckets,where a bucket is defined as a unit of one or more records

The **worst hash function** is one that maps all the keys to the same bucket.

The drawback of using hashed indices includes:

- Though the number of buckets is fixed, the number of files may grow with time.
- If the number of buckets is too large, storage space is wasted.
- If the number of buckets is too small, there may be too many collisions.

The following operations are performed in a hashed file organization.

Insertion

To insert a record that has k_i as its search value, use the hash function $h(k_i)$ to compute the address of the bucket for that record.

If the bucket is free, store the record else use chaining to store the record.

Search

To search a record having the key value k_i, use $h(k_i)$ to compute the address of the bucketwhere the record is stored.

The bucket may contain one or several records, so check for every record in the bucket to retrieve the desired record with the given key value.

Deletion

To delete a record with key value k_i, use $h(k_i)$ to compute the address of the bucket where the record is stored. The bucket may contain one or several records so check for every record in the bucket, and then delete the record.

➢ B-Tree (Balanced Tree) Indices

It is impractical to maintain the entire database in the memory, hence B-trees are used to index the data in order to provide fast access.

B-trees are used for its data retrieval speed, ease of maintenance, and simplicity.

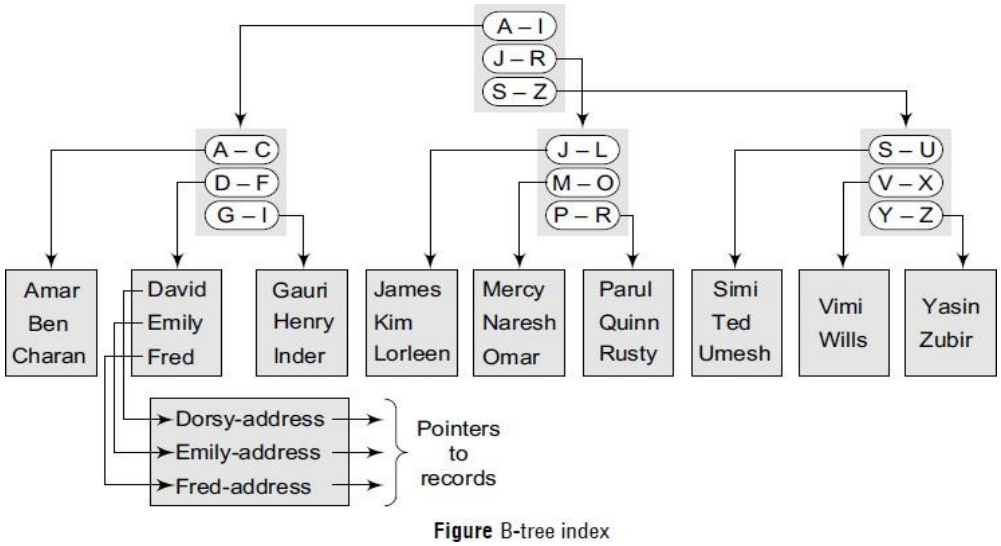


Figure B-tree index

- It forms a tree structure with the root at the top. The index consists of a B-tree (balanced tree) structure based on the values of the indexed column.
- In this example, the indexed column is name and the B-tree is created using all

the existing names that are the values of the indexed column.

- The upper blocks of the tree contain index data pointing to the next lower block, thus forming a hierarchical structure. The lowest level blocks, also known as leaf blocks, contain pointers to the data rows stored in the table.

The B-tree structure has the following advantages:

- Since the leaf nodes of a B-tree are at the same depth, retrieval of any record fromanywhere in the index takes approximately the same time.
- B-trees improve the performance of a wide range of queries that either searches avalue having an exact match or for a value within specified range.
- B-trees provide fast and efficient algorithms to insert, update, and delete records thatmaintain the key order.
- B-trees perform well for small as well as large tables. Their performance does notdegrade as the size of a table grows.

B-trees optimize costly disk access.

➢ B+ Tree

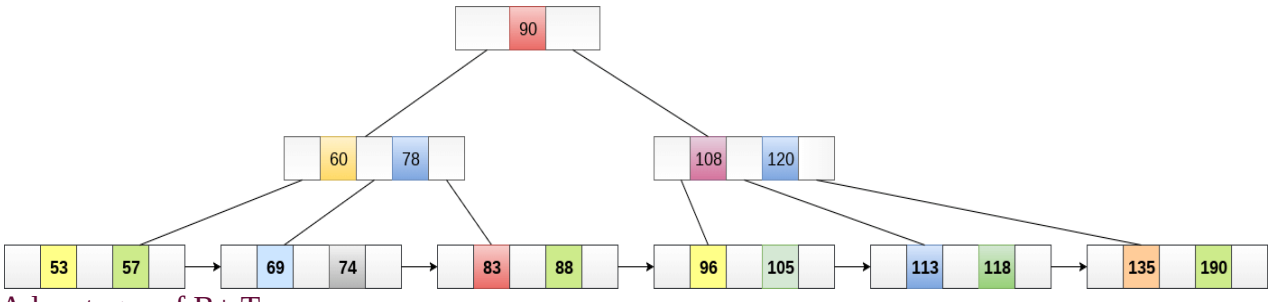
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.

5. Faster search queries as the data is stored only on the leaf nodes.

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Insertion in B+ Tree

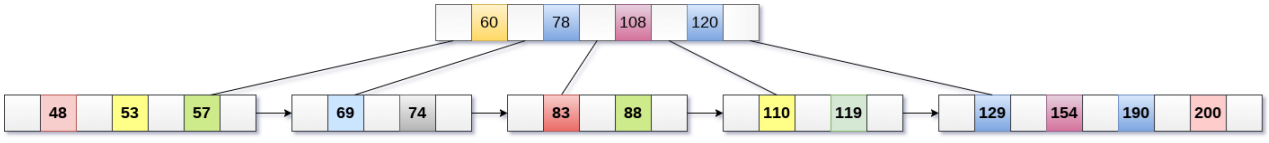
Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

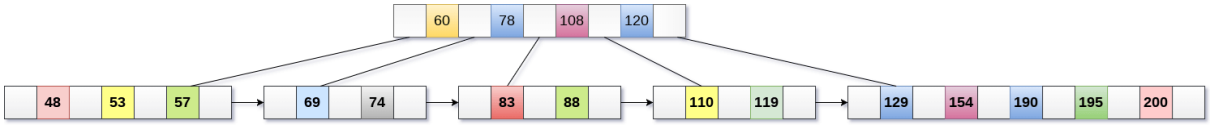
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example :

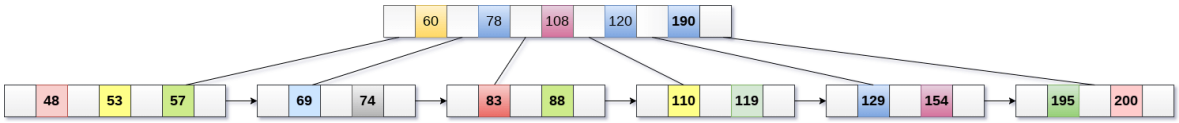
Insert the value 195 into the B+ tree of order 5 shown in the following figure.



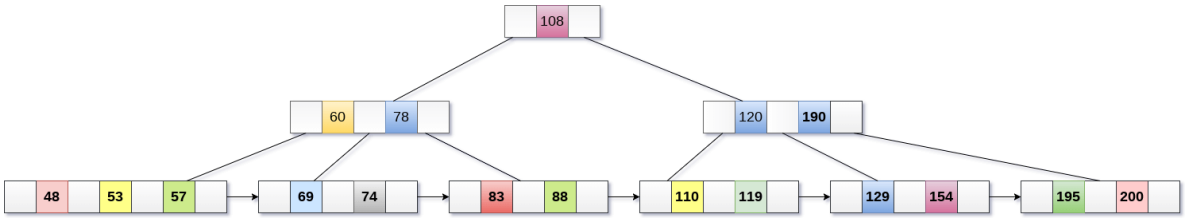
195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



Deletion in B+ Tree

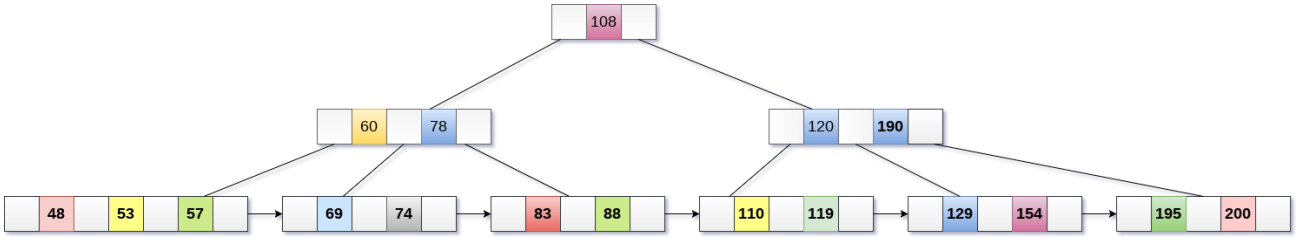
Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

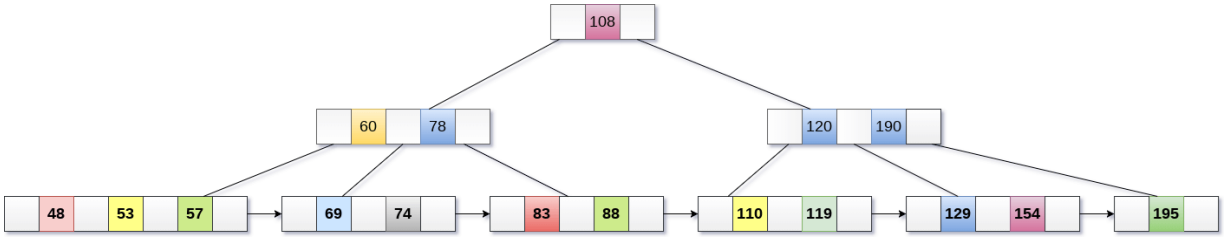
Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Example

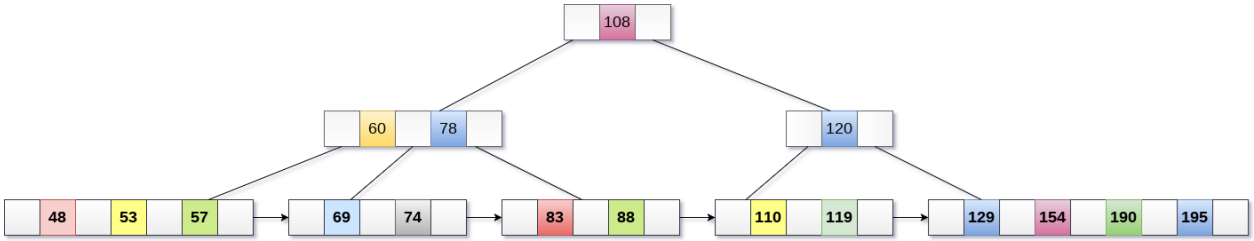
Delete the key 200 from the B+ Tree shown in the following figure.



200 is present in the right sub-tree of 190, after 195. delete it.

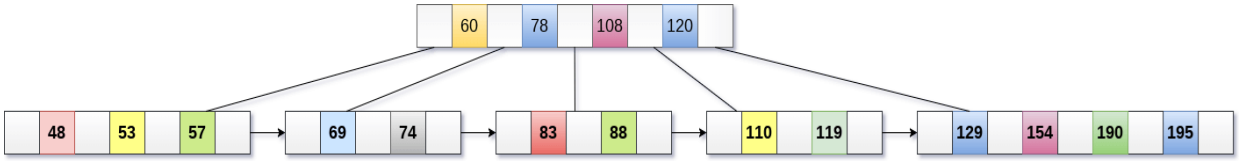


Merge the two nodes by using 195, 190, 154 and 129.



Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.

Now, the height of B+ tree will be decreased by 1.



Data Structures Using JAVA

Unit-1

1. Abstract Data Types.

2. Asymptotic Notations.

3. Case Analysis (Best, Worst and Average case).

4. Arrays, stacks, Queues and Linked Lists.

5. Evaluation of Expressions and Polynomial addition in Linked List.

Unit-II

1. Trees, Binary tree representations and Traversals.

2. Threaded binary trees, Application of trees.

3. Set, Union-find operations.

4. Graph, Representaion, Traversals (DFS and BFS), Connected components, Applications of Graph

5. Minimum Cost spanning tree using Kruskal's algorithm, Dijkstra's algorithm for single source shortest path problem.

Unit-III

1. Search Structures and priority Queues

2. AVL, Red-Black Trees, Splay Trees, Binary Heap

3. Priority Queue Implementation ADT with Heap.

Unit-IV

1. Merge Sort, Quick Sort

2. Comparision of Sorting Algorithms in terms of Complexity.

3. Sorting with disks-k-way merging.

4. Sorting with tapes-Polyphase merge.

Unit-V

1. Linear Search, Binary Search.

2. Hash Tables-Overflow Handling.

3.Cylinder Suface,Hash Indexes

4. B-Tree Indexing and B + trees