# Data Structures

Data Structure is a way to store and organize data so that it can be used efficiently.

Our Data Structure includes all topics of Data Structure such as Array, Pointer, Structure, Linked List, Stack, Queue, Graph, Searching, Sorting, Programs, etc.

## Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vitle role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

### Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned.

**Data:** Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

**Group Items:** Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

**Record:** Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

**File:** A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

**Attribute and Entity:** An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

**Field:** Field is a single elementary unit of information representing the attribute of an entity.

### Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arrise the following problems:

**Processor speed:** To handle very large amout of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

**Data Search:** Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

**Multiple requests:** If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

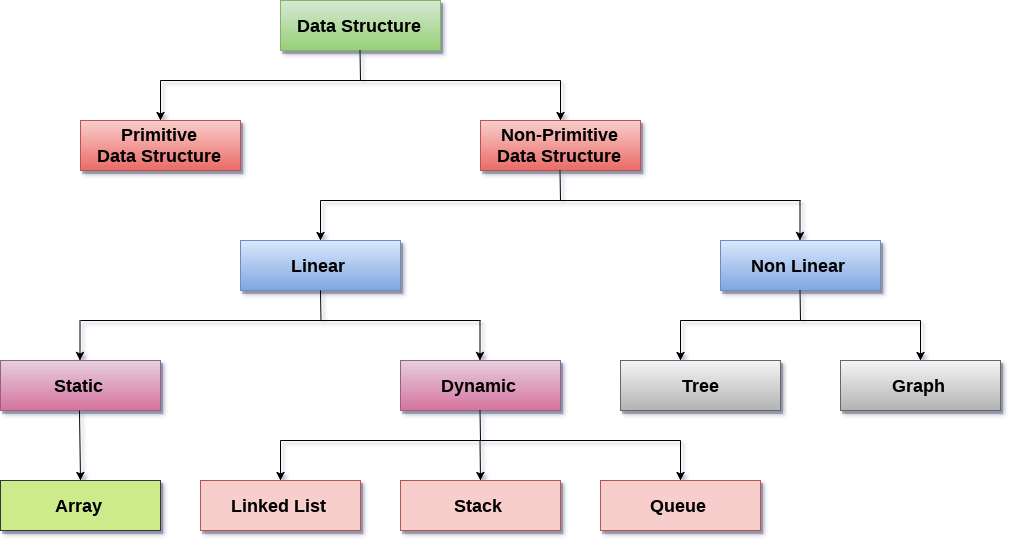
in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

### Advantages of Data Structures

**Efficiency:** Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

**Reusability:** Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

**Abstraction:** Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.



**Linear Data Structures:** A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

**Arrays:** An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],......... age[98], age[99].

**Linked List:** Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

**Stack:** Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

**Queue:** Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

**Non Linear Data Structures:** This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

**Trees:** Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the herierchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classfied into many categories which will be discussed later in this tutorial.

**Graphs:** Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

### Operations on data structure

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

**Example:** If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

# Algorithm

An algorithm is a procedure having well defined steps for solving a particular problem. Algorithm is finite set of logic or instructions, written in order for accomplish the certain predefined task. It is not the complete program or code, it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudo code.

The major categories of algorithms are given below:

* **Sort:** Algorithm developed for sorting the items in certain order.
* **Search:** Algorithm developed for searching the items inside a data structure.
* **Delete:** Algorithm developed for deleting the existing element from the data structure.
* **Insert:** Algorithm developed for inserting an item inside a data structure.
* **Update:** Algorithm developed for updating the existing element inside a data structure.

The performance of algorithm is measured on the basis of following properties:

* **Time complexity:** It is a way of representing the amount of time needed by a program to run to the completion.
* **Space complexity:** It is the amount of memory space required by an algorithm, during a course of its execution. Space complexity is required in situations when limited memory is available and for the multi user system.

Each algorithm must have:

* **Specification:** Description of the computational procedure.
* **Pre-conditions:** The condition(s) on input.
* **Body of the Algorithm:** A sequence of clear and unambiguous instructions.
* **Post-conditions:** The condition(s) on output.

**Example:** Design an algorithm to multiply the two numbers x and y and display the result in z.

* Step 1 START
* Step 2 declare three integers x, y & z
* Step 3 define values of x & y
* Step 4 multiply values of x & y
* Step 5 store the output of step 4 in z
* Step 6 print z
* Step 7 STOP

. Alternatively the algorithm can be written as ?

* Step 1 START MULTIPLY
* Step 2 get values of x & y
* Step 3 z← x \* y
* Step 4 display z
* Step 5 STOP

### Characteristics of an Algorithm

An algorithm must follow the mentioned below characteristics:

* **Input:** An algorithm must have 0 or well defined inputs.
* **Output:** An algorithm must have 1 or well defined outputs, and should match with the desired output.
* **Feasibility:** An algorithm must be terminated after the finite number of steps.
* **Independent:** An algorithm must have step-by-step directions which is independent of any programming code.
* **Unambiguous:** An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and lead to only one meaning.

# Structure

A structure is a composite data type that defines a grouped list of variables that are to be placed under one name in a block of memory. It allows different variables to be accessed by using a single pointer to the structure.

**Syntax**

1. struct structure\_name
2. {
3. data\_type member1;
4. data\_type member2;
5. .
6. .
7. data\_type memeber;
8. };

### Advantages

* It can hold variables of different data types.
* We can create objects containing different types of attributes.
* It allows us to re-use the data layout across programs.
* It is used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

# Linked List

* Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
* A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
* The last node of the list contains pointer to the null.

DS Linked List

## Uses of Linked List

* The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
* list size is limited to the memory size and doesn't need to be declared in advance.
* Empty node can not be present in the linked list.
* We can store values of primitive types or objects in the singly linked list.

## Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

## Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

DS Singly Linked List

In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

## Complexity

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Data Structure** | **Time Complexity** |  |  |  |  |  |  |  | **Space Compleity** |
|  | **Average** |  |  |  | **Worst** |  |  |  | **Worst** |
|  | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |  |
| Singly Linked List | θ(n) | θ(n) | θ(1) | θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |

## Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

### Node Creation

1. struct node
2. {
3. **int** data;
4. struct node \*next;
5. };
6. struct node \*head, \*ptr;
7. ptr = (struct node \*)malloc(sizeof(struct node \*));

### Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Insertion at beginning](https://www.javatpoint.com/insertion-in-singly-linked-list-at-beginning) | It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list. |
| 2 | [Insertion at end of the list](https://www.javatpoint.com/insertion-in-singly-linked-list-at-end) | It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario. |
| 3 | [Insertion after specified node](https://www.javatpoint.com/insertion-in-singly-linked-list-after-specified-node) | It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. . |

### Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Deletion at beginning](https://www.javatpoint.com/deletion-in-singly-linked-list-at-beginning) | It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers. |
| 2 | [Deletion at the end of the list](https://www.javatpoint.com/deletion-in-singly-linked-list-at-end) | It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios. |
| 3 | [Deletion after specified node](https://www.javatpoint.com/deletion-in-singly-linked-list-after-specified-node) | It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list. |
| 4 | [Traversing](https://www.javatpoint.com/traversing-in-singly-linked-list) | In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list. |
| 5 | [Searching](https://www.javatpoint.com/searching-in-singly-linked-list) | In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. . |

# Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



In C, structure of a node in doubly linked list can be given as :

1. struct node
2. {
3. struct node \*prev;
4. **int** data;
5. struct node \*next;
6. }

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

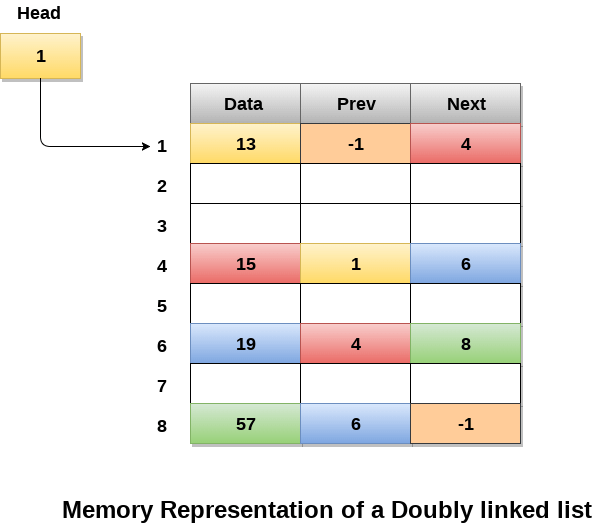
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

## Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



## Operations on doubly linked list

**Node Creation**

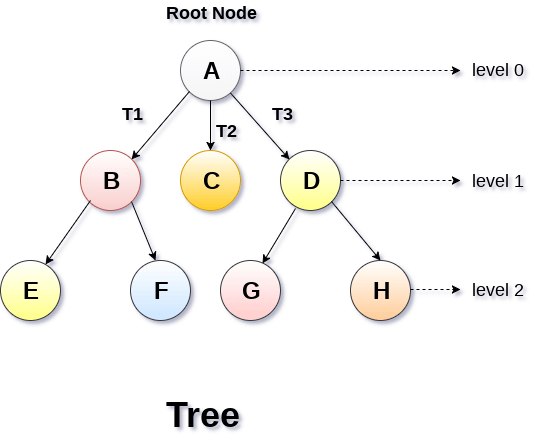
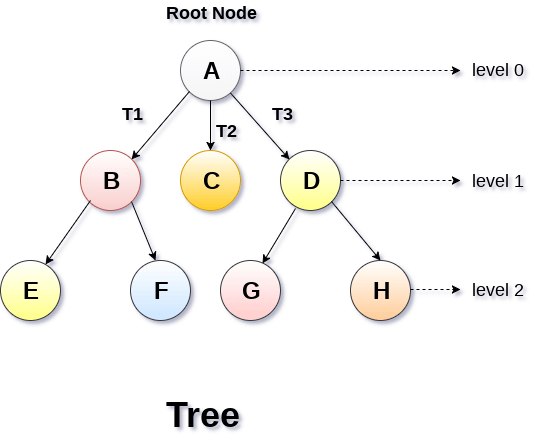
1. struct node
2. {
3. struct node \*prev;
4. **int** data;
5. struct node \*next;
6. };
7. struct node \*head;

All the remaining operations regarding doubly linked list are described in the following table.

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Insertion at beginning](https://www.javatpoint.com/insertion-in-doubly-linked-list-at-beginning) | Adding the node into the linked list at beginning. |
| 2 | [Insertion at end](https://www.javatpoint.com/insertion-in-doubly-linked-list-at-the-end) | Adding the node into the linked list to the end. |
| 3 | [Insertion after specified node](https://www.javatpoint.com/insertion-in-doubly-linked-list-after-specified-node) | Adding the node into the linked list after the specified node. |
| 4 | [Deletion at beginning](https://www.javatpoint.com/deletion-in-doubly-linked-list-at-beginning) | Removing the node from beginning of the list |
| 5 | [Deletion at the end](https://www.javatpoint.com/deletion-in-doubly-linked-list-at-the-end) | Removing the node from end of the list. |
| 6 | [Deletion of the node having given data](https://www.javatpoint.com/deletion-in-doubly-linked-list-after-the-specified-node) | Removing the node which is present just after the node containing the given data. |
| 7 | [Searching](https://www.javatpoint.com/searching-in-doubly-linked-list) | Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null. |
| 8 | [Traversing](https://www.javatpoint.com/traversing-in-doubly-linked-list) | Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc. |

# Tree

* A Tree is a recursive data structure containing the set of one or more data nodes where one node is designated as the root of the tree while the remaining nodes are called as the children of the root.
* The nodes other than the root node are partitioned into the non empty sets where each one of them is to be called sub-tree.
* Nodes of a tree either maintain a parent-child relationship between them or they are sister nodes.
* In a general tree, A node can have any number of children nodes but it can have only a single parent.
* The following image shows a tree, where the node A is the root node of the tree while the other nodes can be seen as the children of A.



## Basic terminology

* **Root Node** :- The root node is the topmost node in the tree hierarchy. In other words, the root node is the one which doesn't have any parent.
* **Sub Tree** :- If the root node is not null, the tree T1, T2 and T3 is called sub-trees of the root node.
* **Leaf Node** :- The node of tree, which doesn't have any child node, is called leaf node. 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.
* **Path** :- The sequence of consecutive edges is called path. In the tree shown in the above image, path to the node E is A→ B → E.
* **Ancestor node** :- An ancestor of a node is any predecessor node on a path from root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, the node F have the ancestors, B and A.
* **Degree** :- Degree of a node is equal to number of children, a node have. In the tree shown in the above image, the degree of node B is 2. Degree of a leaf node is always 0 while in a complete binary tree, degree of each node is equal to 2.
* **Level Number** :- Each node of the tree is assigned a level number in such a way that each node is present at one level higher than its parent. Root node of the tree is always present at level 0.

## Static representation of tree

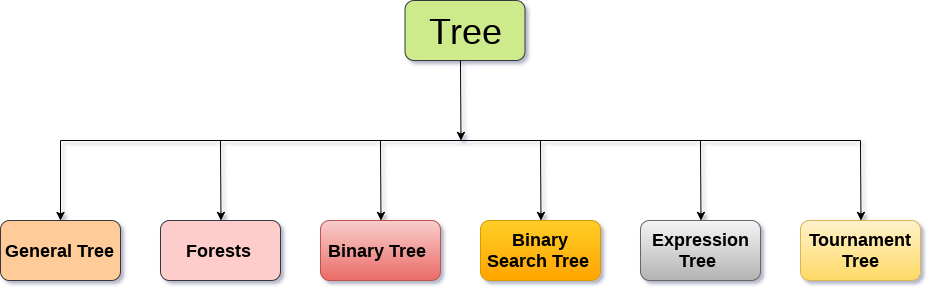
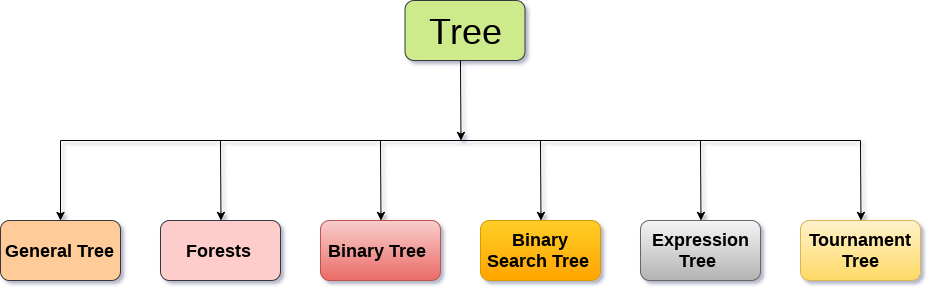
1. #define MAXNODE 500
2. struct treenode {
3. **int** root;
4. **int** father;
5. **int** son;
6. **int** next;
7. }

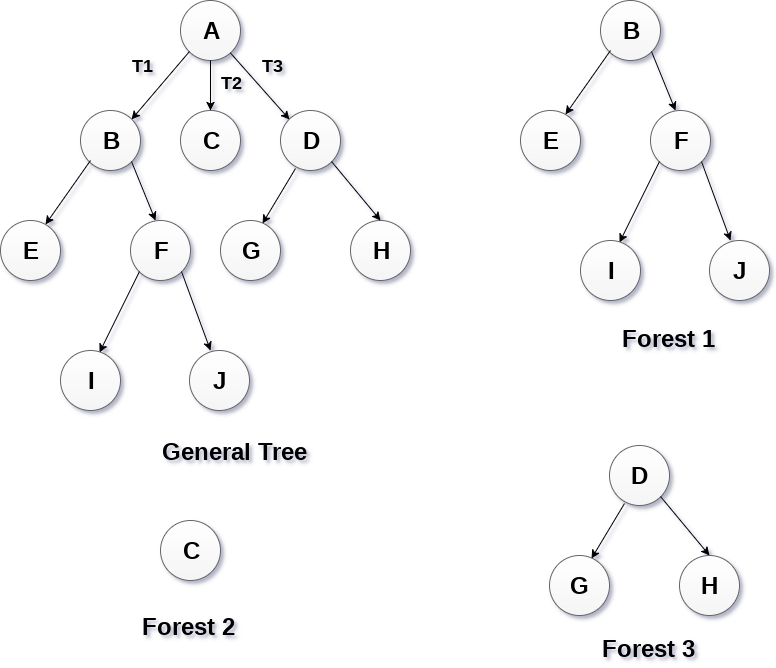
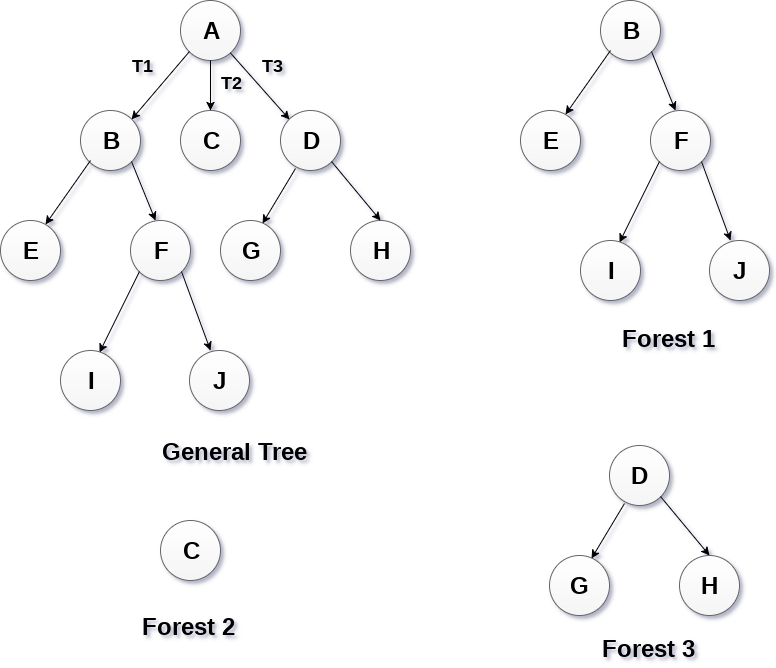
## Dynamic representation of tree

1. struct treenode
2. {
3. **int** root;
4. struct treenode \*father;
5. struct treenode \*son
6. struct treenode \*next;
7. }

## Types of Tree

The tree data structure can be classified into six different categories.





## Binary Tree

Binary tree is a data structure in which each node can have at most 2 children. The node present at the top most level is called the root node. A node with the 0 children is called leaf node. Binary Trees are used in the applications like expression evaluation and many more. We will discuss binary tree in detail, later in this tutorial.

## Binary Search Tree

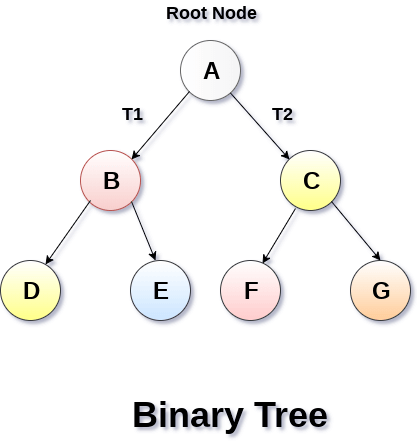
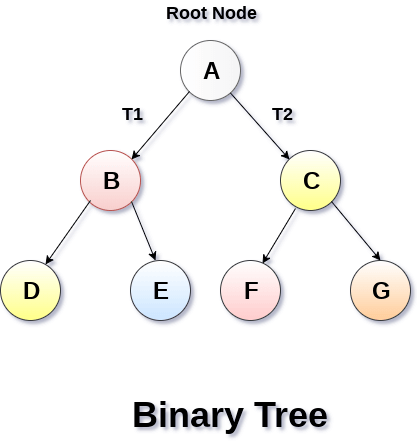
Binary search tree is an ordered binary tree. All the elements in the left sub-tree are less than the root while elements present in the right sub-tree are greater than or equal to the root node element. Binary search trees are used in most of the applications of computer science domain like searching, sorting, etc.

# Binary Tree

Binary Tree is a special type of generic tree in which, each node can have at most two children. Binary tree is generally partitioned into three disjoint subsets.

1. Root of the node
2. left sub-tree which is also a binary tree.
3. Right binary sub-tree

A binary Tree is shown in the following image.

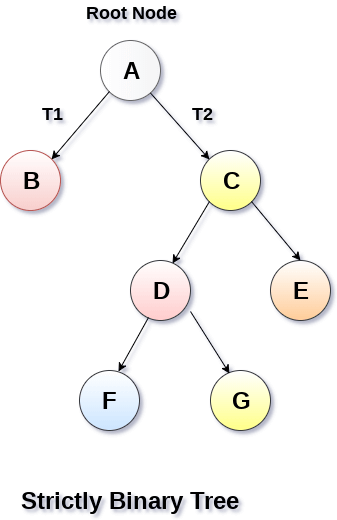
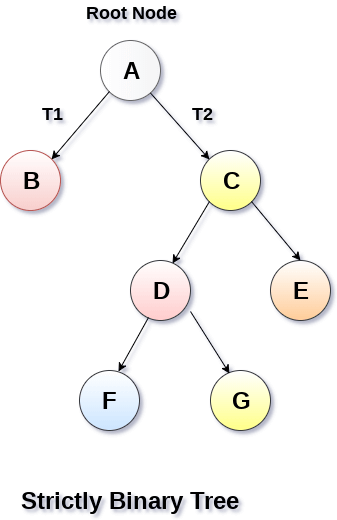


## Types of Binary Tree

### 1. Strictly Binary Tree

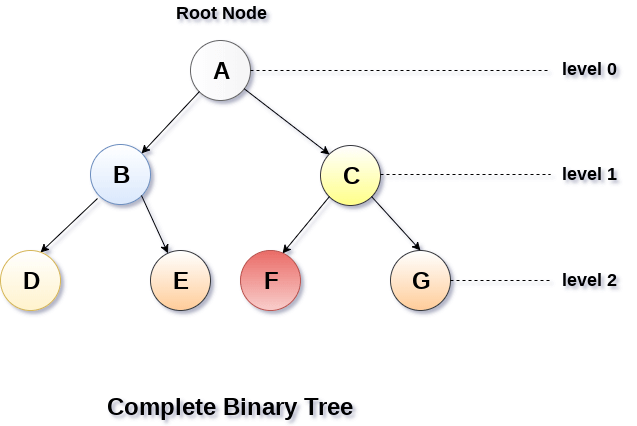
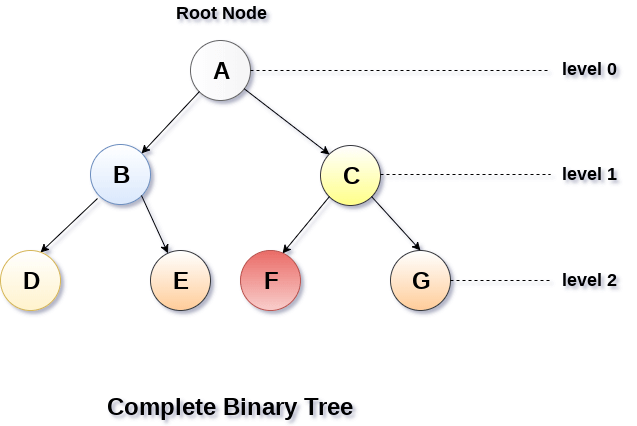
In Strictly Binary Tree, every non-leaf node contain non-empty left and right sub-trees. In other words, the degree of every non-leaf node will always be 2. A strictly binary tree with n leaves, will have (2n - 1) nodes.

A strictly binary tree is shown in the following figure.



### 2. Complete Binary Tree

A Binary Tree is said to be a complete binary tree if all of the leaves are located at the same level d. A complete binary tree is a binary tree that contains exactly 2^l nodes at each level between level 0 and d. The total number of nodes in a complete binary tree with depth d is 2d+1-1 where leaf nodes are 2d while non-leaf nodes are 2d-1.



## Binary Tree Traversal

|  |  |  |
| --- | --- | --- |
| **SN** | **Traversal** | **Description** |
| 1 | [Pre-order Traversal](https://www.javatpoint.com/binary-tree-preorder-traversal) | Traverse the root first then traverse into the left sub-tree and right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively. |
| 2 | [In-order Traversal](https://www.javatpoint.com/binary-tree-inorder-traversal) | Traverse the left sub-tree first, and then traverse the root and the right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively. |
| 3 | [Post-order Traversal](https://www.javatpoint.com/binary-tree-postorder-traversal) | Traverse the left sub-tree and then traverse the right sub-tree and root respectively. This procedure will be applied to each sub-tree of the tree recursively. |

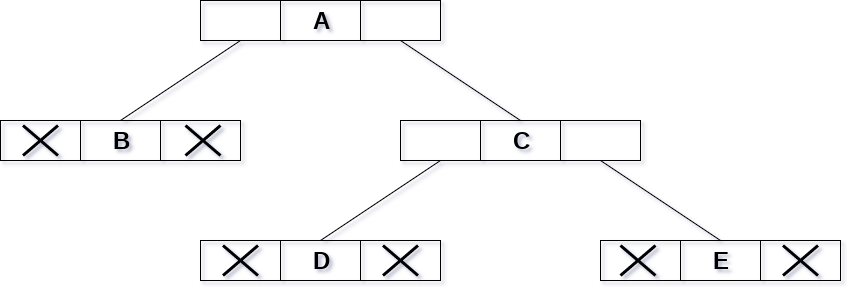
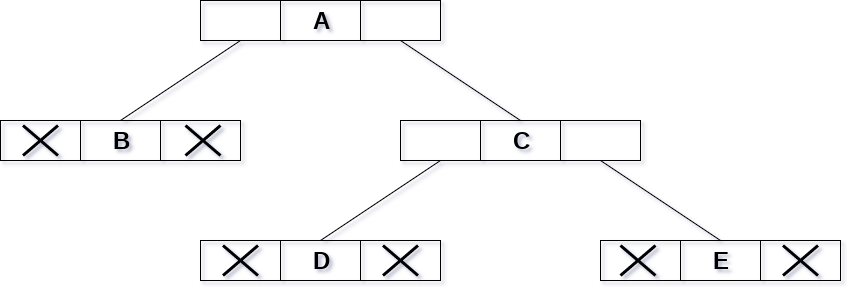
## Binary Tree representation

There are two types of representation of a binary tree:

### 1. Linked Representation

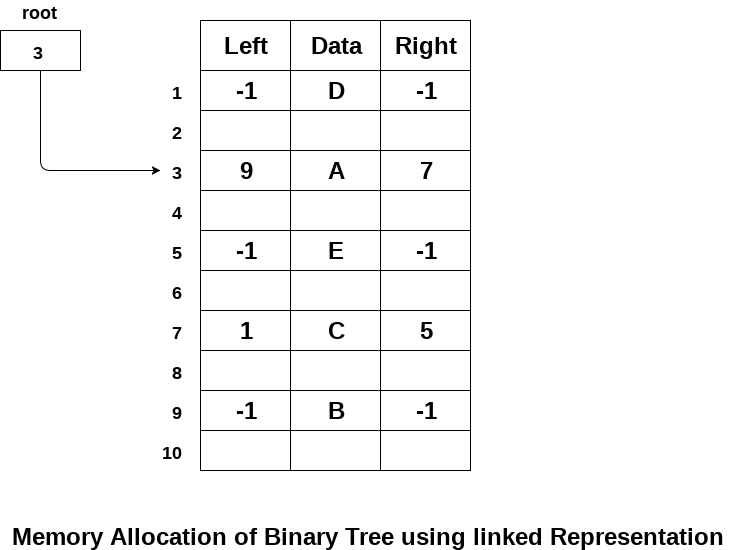
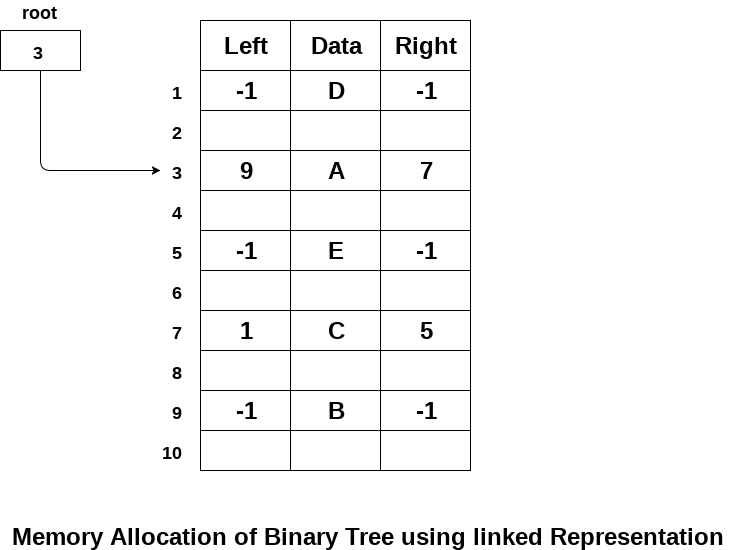
In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non-contiguous memory locations and linked together by inheriting parent child relationship like a tree. every node contains three parts : pointer to the left node, data element and pointer to the right node. Each binary tree has a root pointer which points to the root node of the binary tree. In an empty binary tree, the root pointer will point to null.

Consider the binary tree given in the figure below.



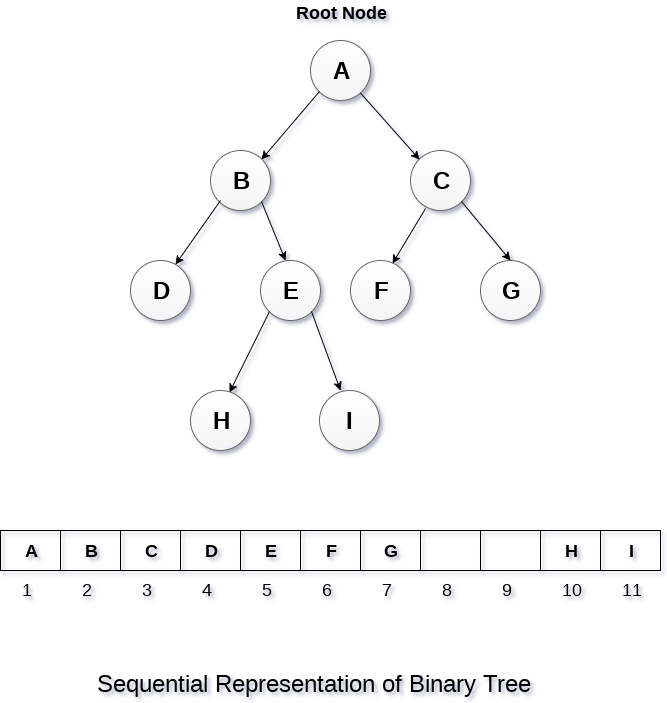
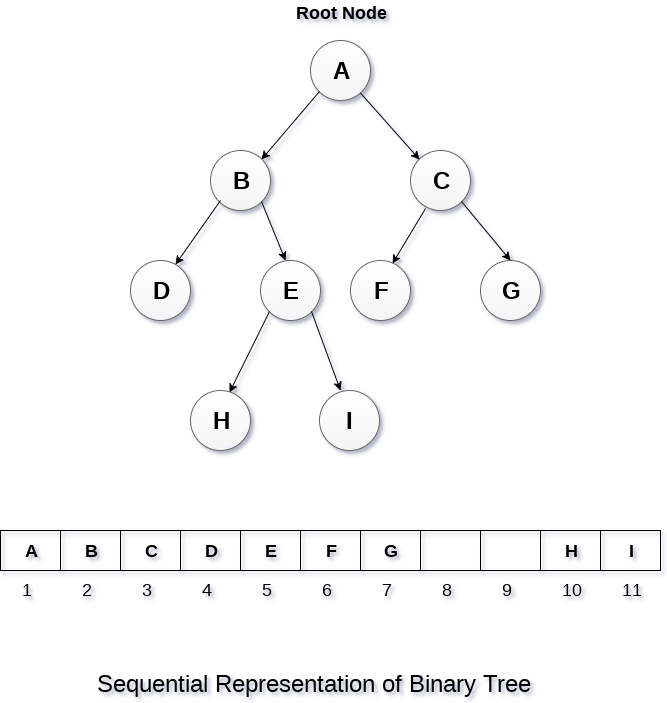
In the above figure, a tree is seen as the collection of nodes where each node contains three parts : left pointer, data element and right pointer. Left pointer stores the address of the left child while the right pointer stores the address of the right child. The leaf node contains **null** in its left and right pointers.

The following image shows about how the memory will be allocated for the binary tree by using linked representation. There is a special pointer maintained in the memory which points to the root node of the tree. Every node in the tree contains the address of its left and right child. Leaf node contains null in its left and right pointers.



### 2. Sequential Representation

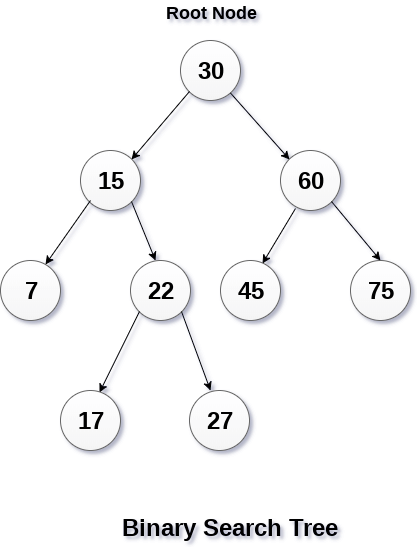
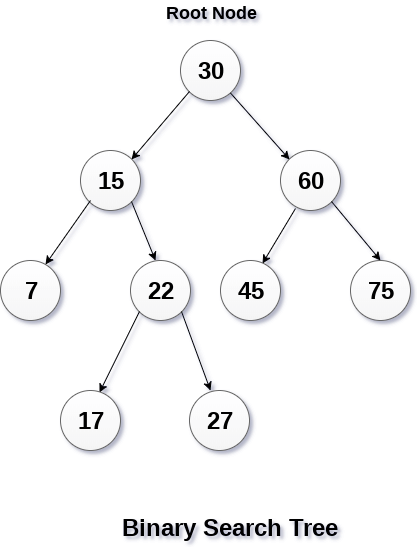
This is the simplest memory allocation technique to store the tree elements but it is an inefficient technique since it requires a lot of space to store the tree elements. A binary tree is shown in the following figure along with its memory allocation.



In this representation, an array is used to store the tree elements. Size of the array will be equal to the number of nodes present in the tree. The root node of the tree will be present at the 1st index of the array. If a node is stored at ith index then its left and right children will be stored at 2i and 2i+1 location. If the 1st index of the array i.e. tree[1] is 0, it means that the tree is empty.

# Binary Search Tree

1. Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
2. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
3. Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
4. This rule will be recursively applied to all the left and right sub-trees of the root.



A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.

## Advantages of using binary search tree

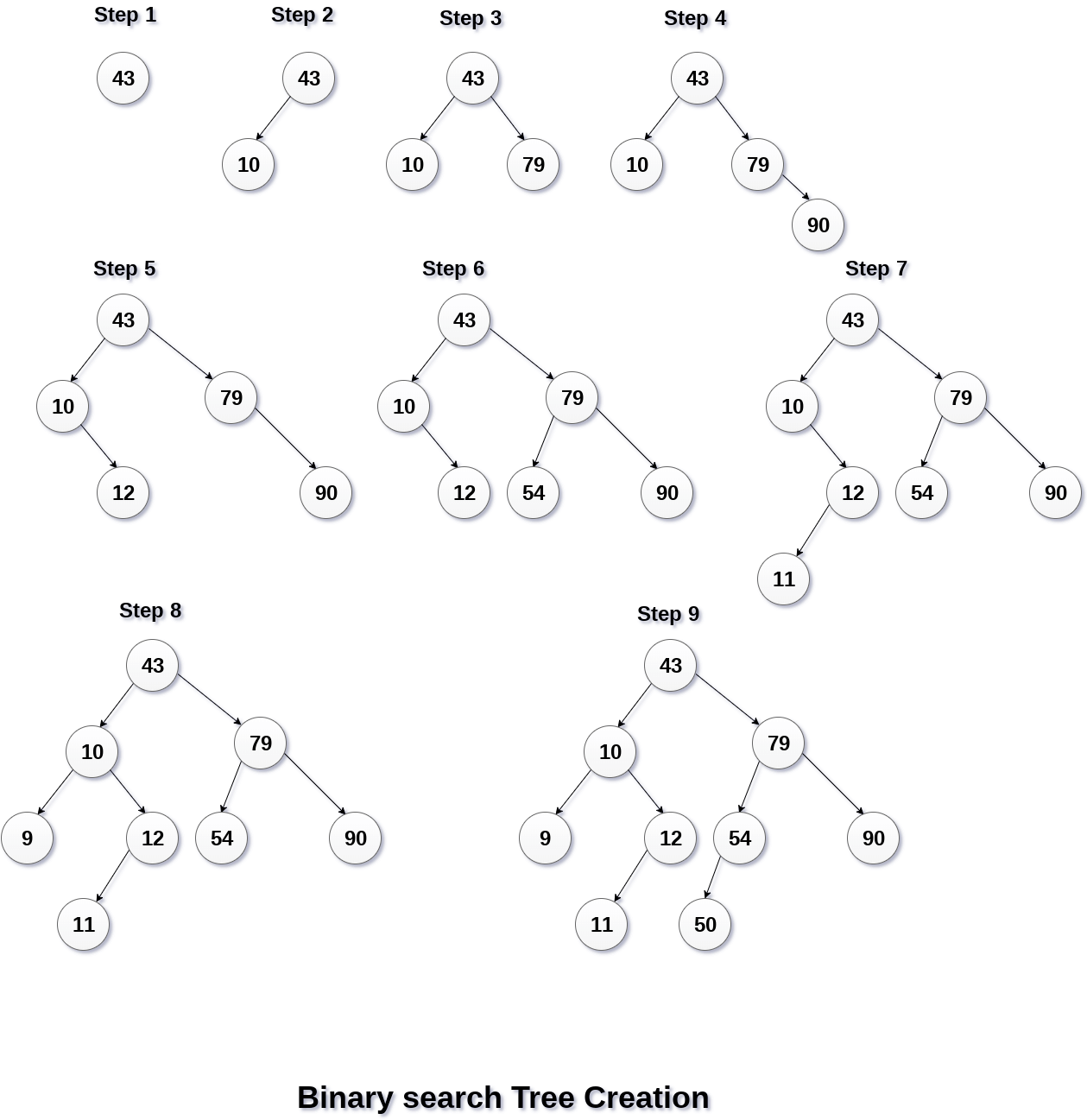
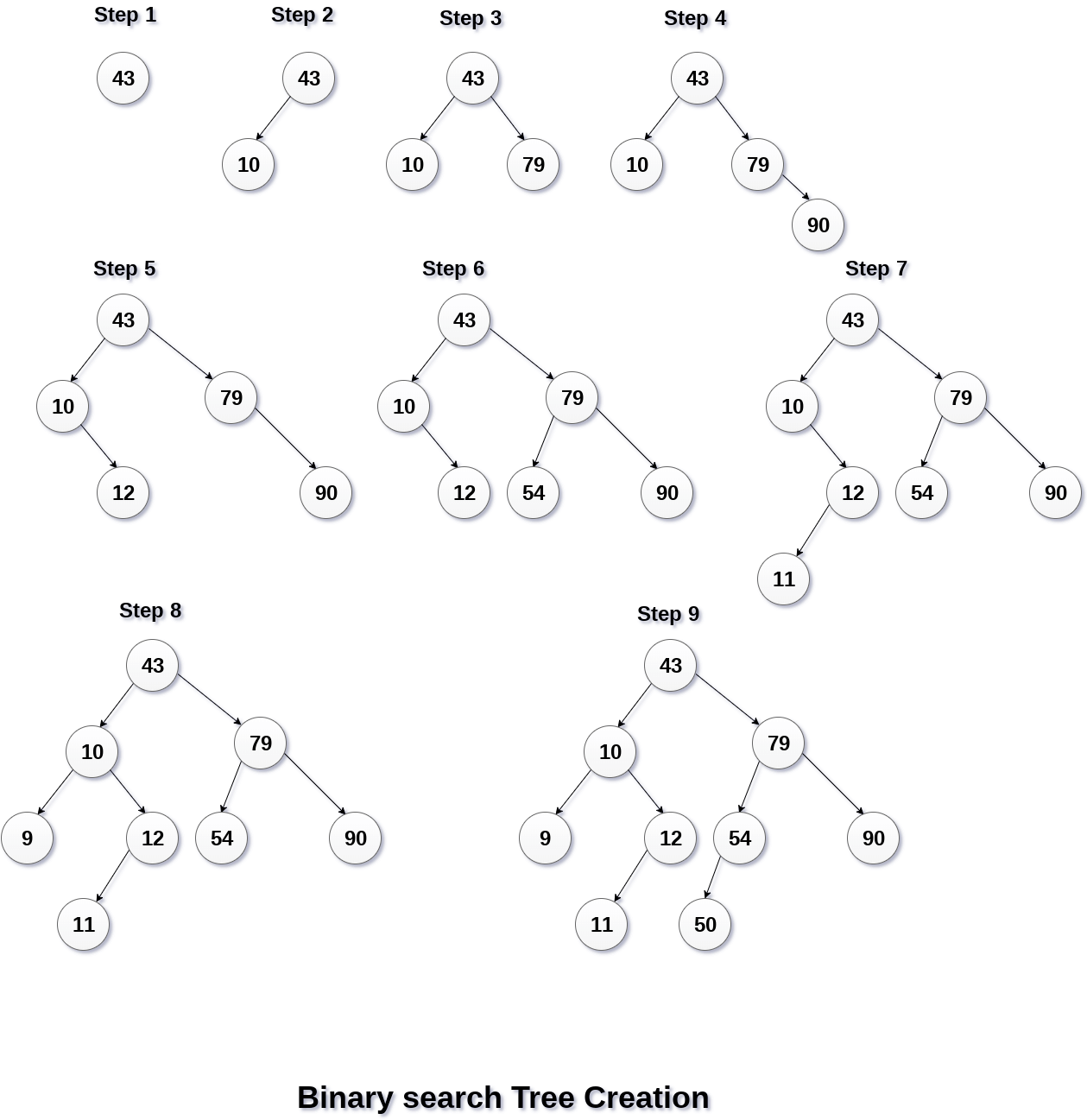
1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o(log2n) time. In worst case, the time it takes to search an element is 0(n).
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

### Q. Create the binary search tree using the following data elements.

**43, 10, 79, 90, 12, 54, 11, 9, 50**

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.



## Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree.

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Searching in BST](https://www.javatpoint.com/searching-in-binary-search-tree) | Finding the location of some specific element in a binary search tree. |
| 2 | [Insertion in BST](https://www.javatpoint.com/insertion-in-binary-search-tree) | Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate. |
| 3 | [Deletion in BST](https://www.javatpoint.com/deletion-in-binary-search-tree) | Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have. |

# 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.

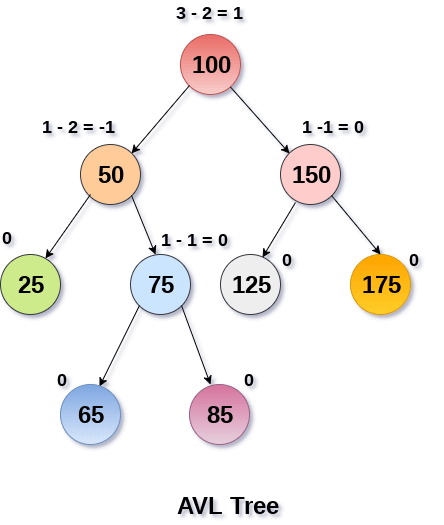
## Balance Factor (k) = height (left(k)) - height (right(k))

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.



## 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](https://www.javatpoint.com/insertion-in-avl-tree) | 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](https://www.javatpoint.com/deletion-in-avl-tree) | 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.

# B Tree

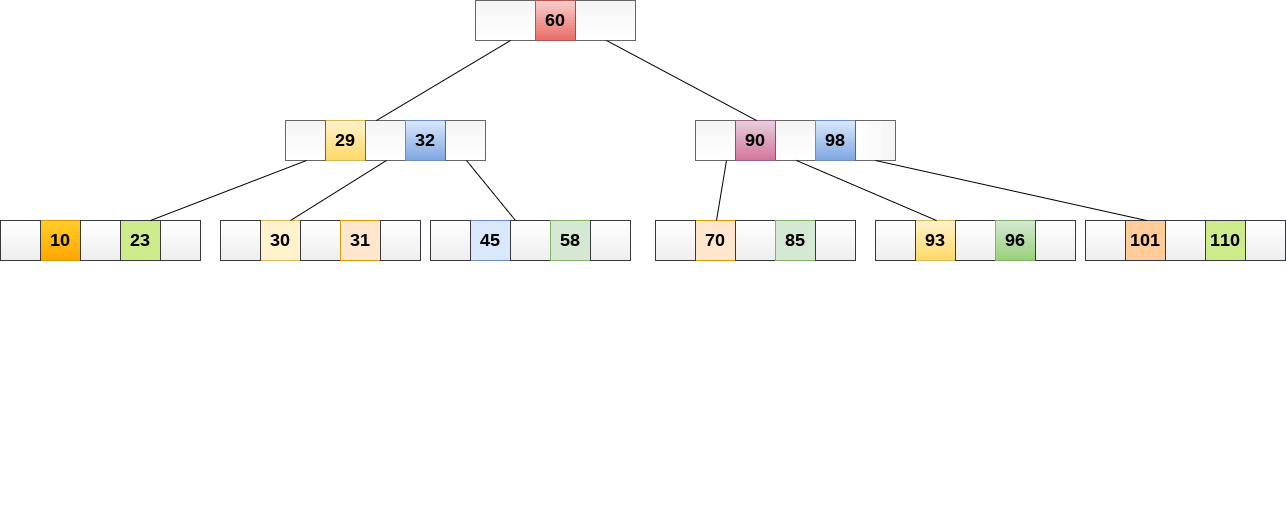
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

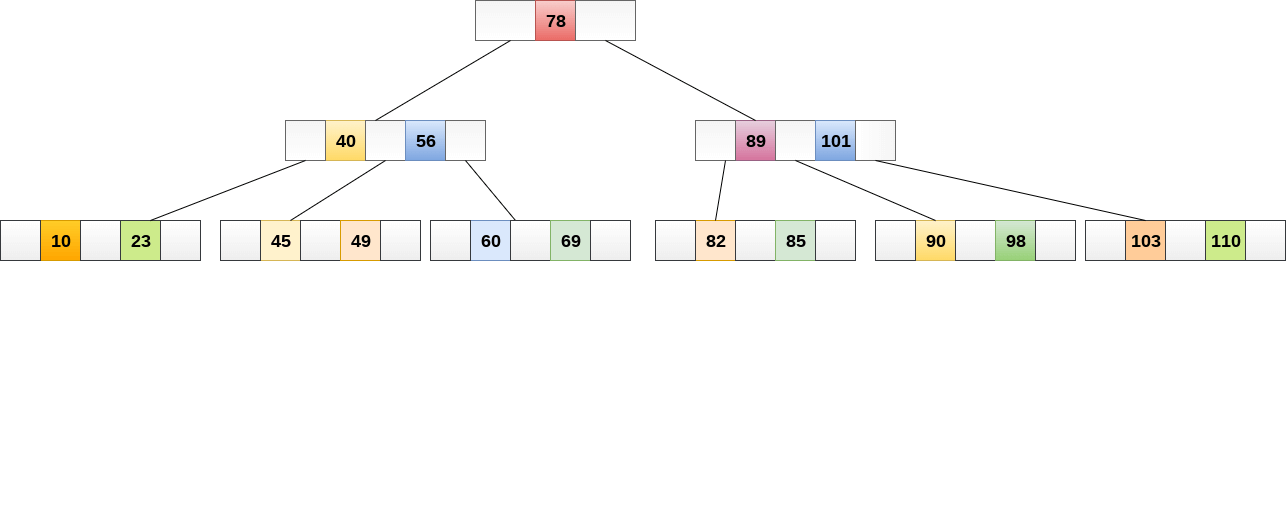
## Operations

### Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.



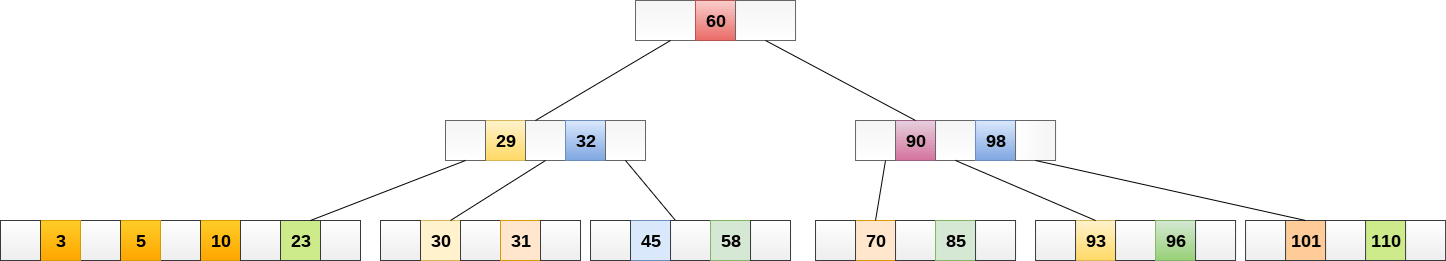
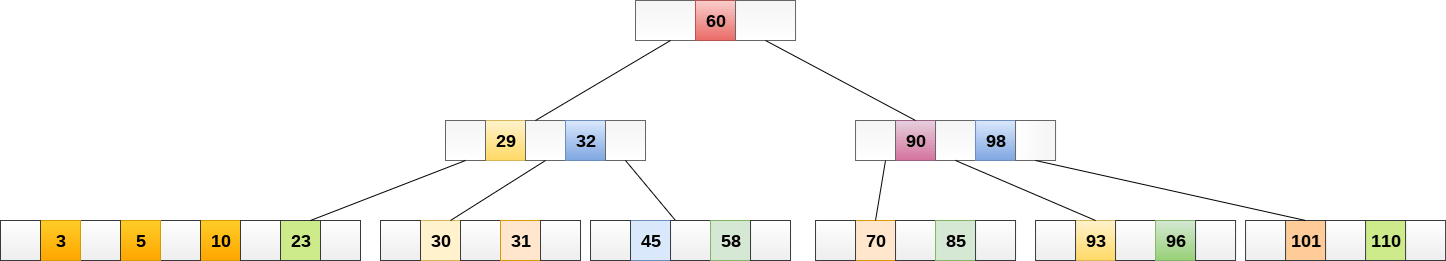
### Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

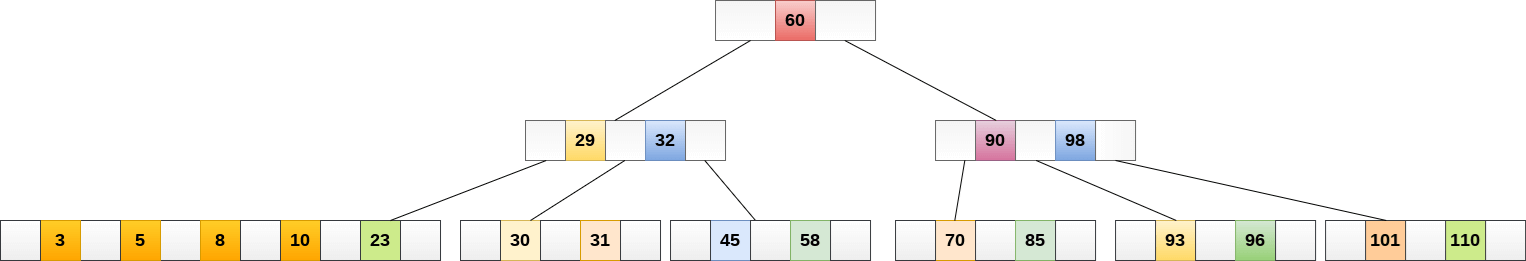
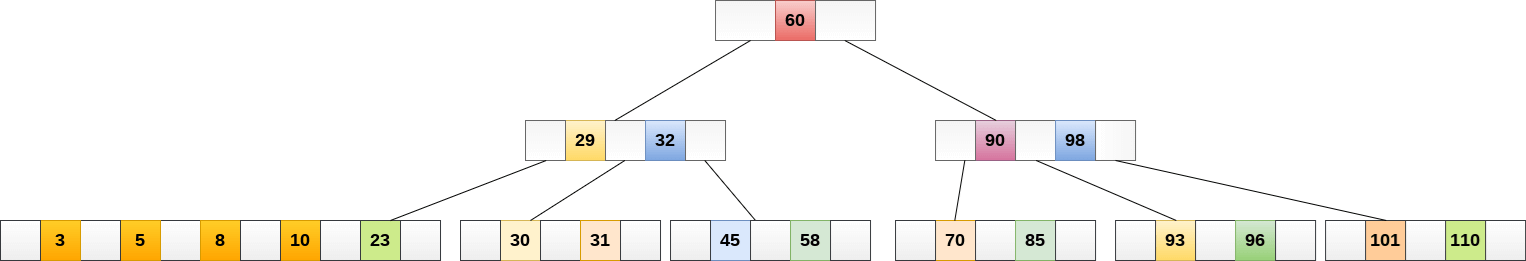
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
   * Insert the new element in the increasing order of elements.
   * Split the node into the two nodes at the median.
   * Push the median element upto its parent node.
   * If the parent node also contain m-1 number of keys, then split it too by following the same steps.

**Example:**

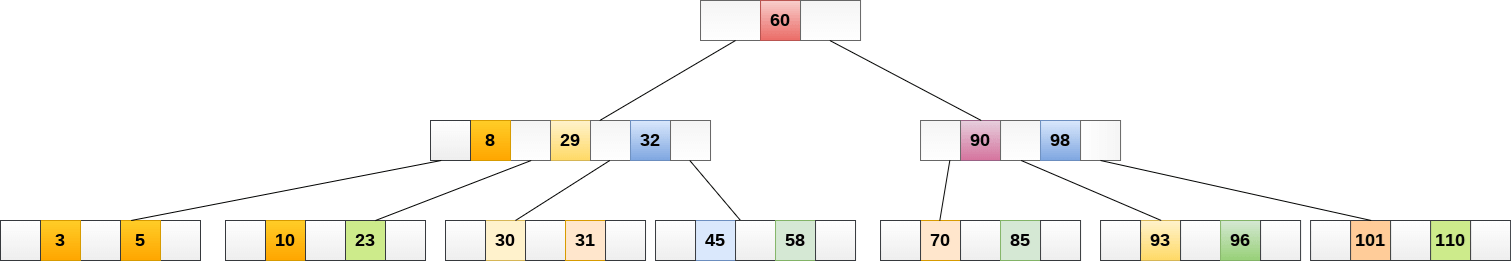
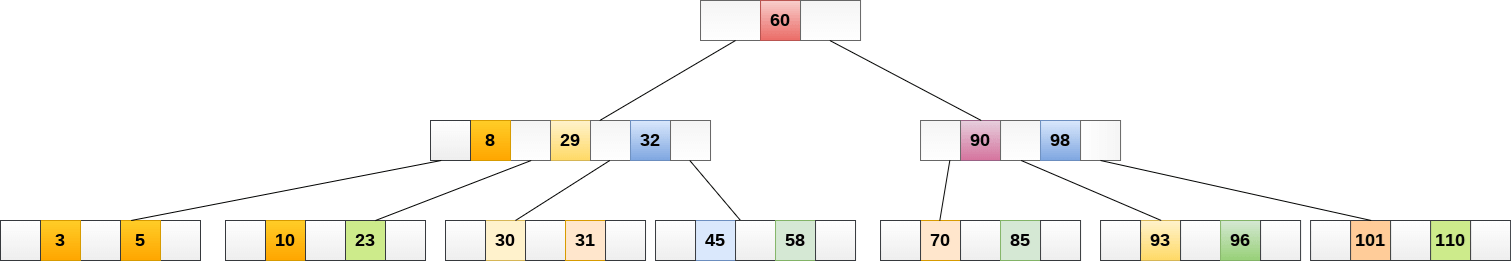
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than (5 -1 = 4 ) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



### Deletion

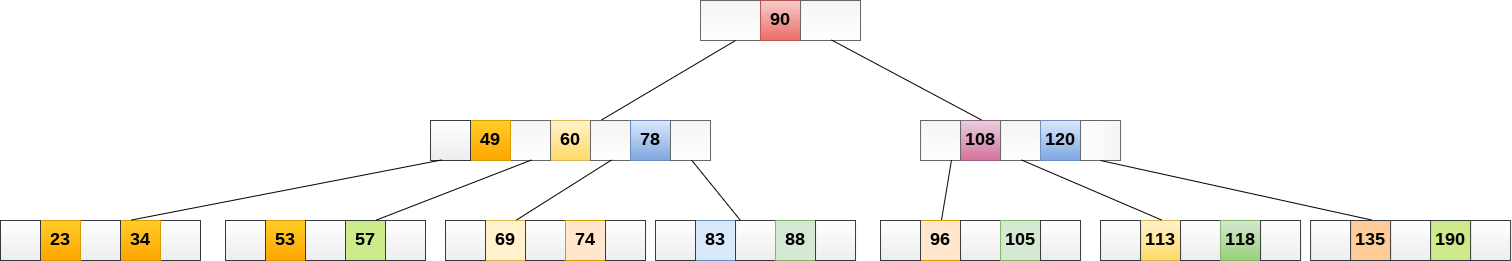
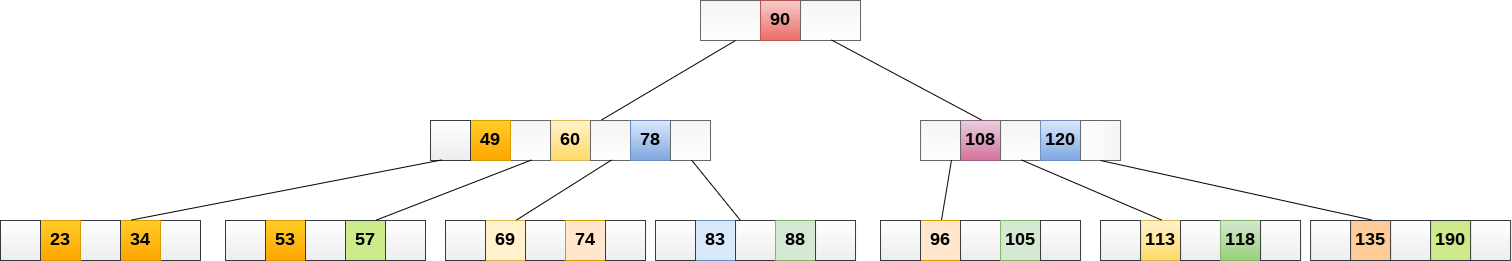
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
   * If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
   * If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

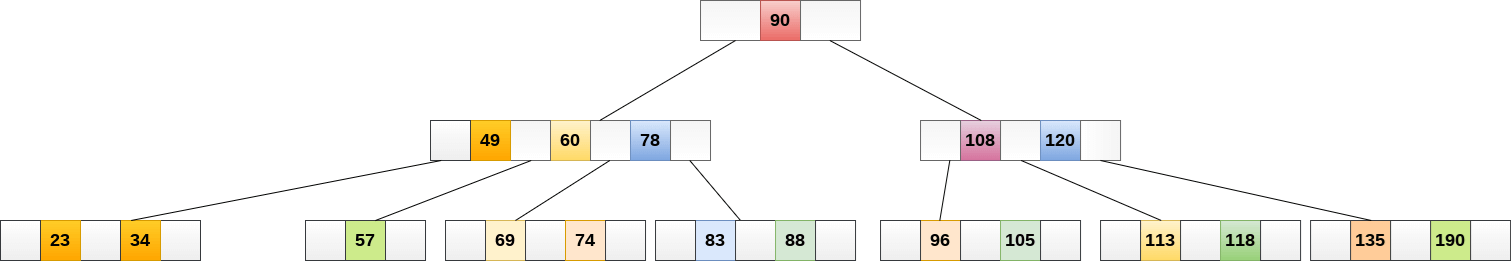
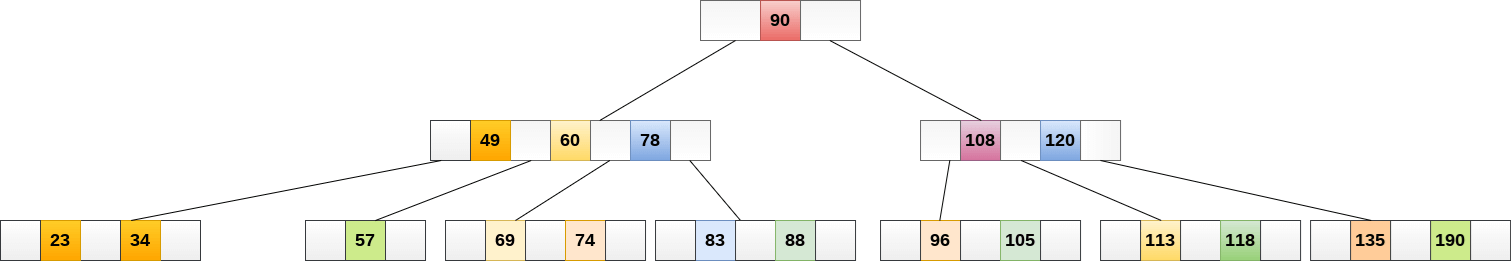
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

**Example 1**

Delete the node 53 from the B Tree of order 5 shown in the following figure.

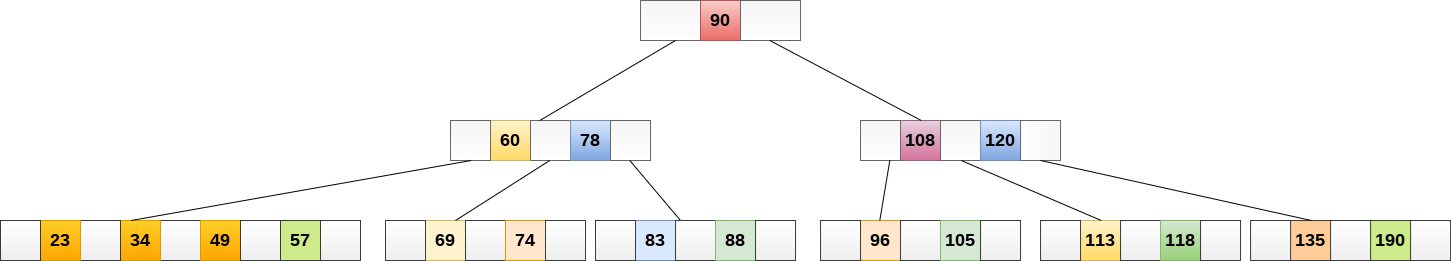
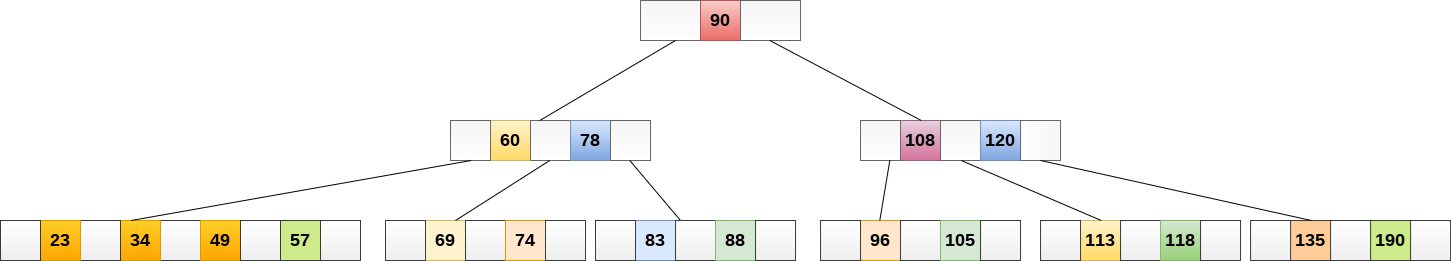


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



## Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

# Graph Traversal Algorithm

In this part of the tutorial we will discuss the techniques by using which, we can traverse all the vertices of the graph.

Traversing the graph means examining all the nodes and vertices of the graph. There are two standard methods by using which, we can traverse the graphs. Lets discuss each one of them in detail.

* Breadth First Search
* Depth First Search

## 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.

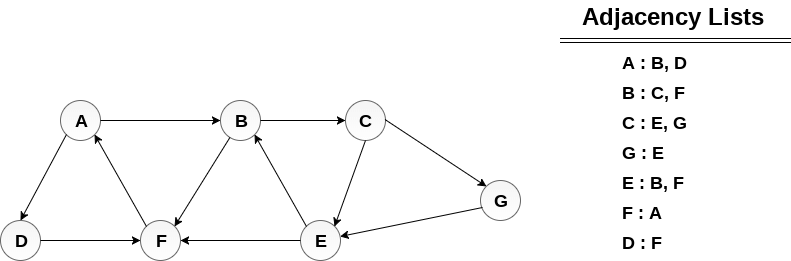
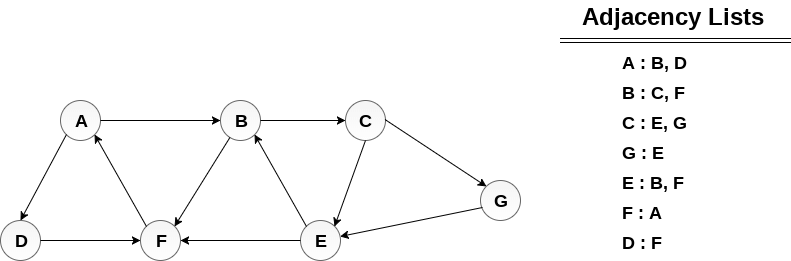
The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and 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.



## 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**.

**Lets start examining the graph from Node A.**

1. Add A to QUEUE1 and NULL to QUEUE2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

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

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

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

1. QUEUE1 = {D, C, F}
2. 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.

1. QUEUE1 = {C, F}
2. QUEUE2 = { A, B, D}

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

1. QUEUE1 = {F, E, G}
2. 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.

1. QUEUE1 = {E, G}
2. 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.

1. QUEUE1 = {G}
2. 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**.

# 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.

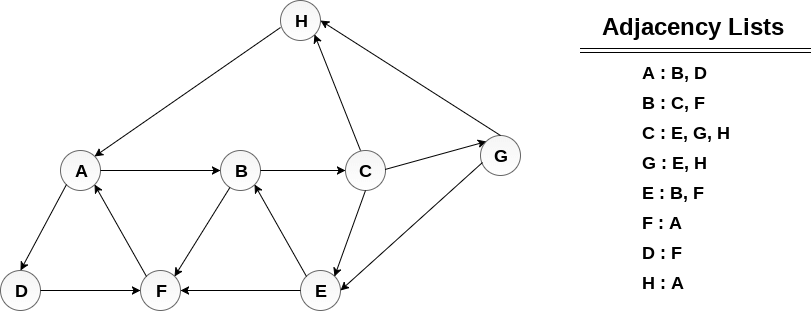
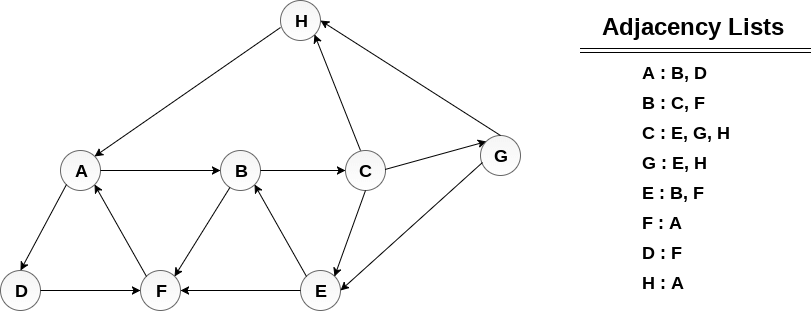
The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

## 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.



### Solution :

Push H onto the stack

1. 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.

1. Print H
2. 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.

1. Print A
2. 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.

1. Print D
2. 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.

1. Print F
2. Stack : B

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

1. Print B
2. Stack : C

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

1. Print C
2. Stack : E, G

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

1. Print G
2. Stack : E

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

1. Print E
2. 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 :

1. H → A → D → F → B → C → G → E

# 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 POS  
  Go 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) |

### Java Program

1. **import** java.util.Scanner;
3. **public** **class** Leniear\_Search {
4. **public** **static** **void** main(String[] args) {
5. **int**[] arr = {10, 23, 15, 8, 4, 3, 25, 30, 34, 2, 19};
6. **int** item,flag=0;
7. Scanner sc = **new** Scanner(System.in);
8. System.out.println("Enter Item ?");
9. item = sc.nextInt();
10. **for**(**int** i = 0; i<10; i++)
11. {
12. **if**(arr[i]==item)
13. {
14. flag = i+1;
15. **break**;
16. }
17. **else**
18. flag = 0;
19. }
20. **if**(flag != 0)
21. {
22. System.out.println("Item found at location" + flag);
23. }
24. **else**
25. System.out.println("Item not found");
27. }
28. }

**Output:**

Enter Item ?23Item found at location 2Enter Item ?22Item 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 :**

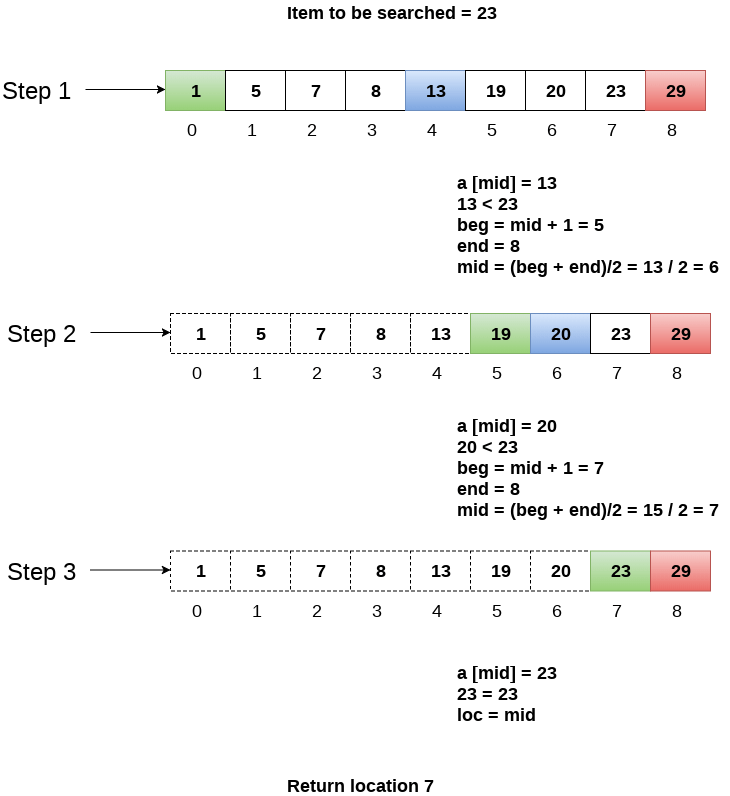
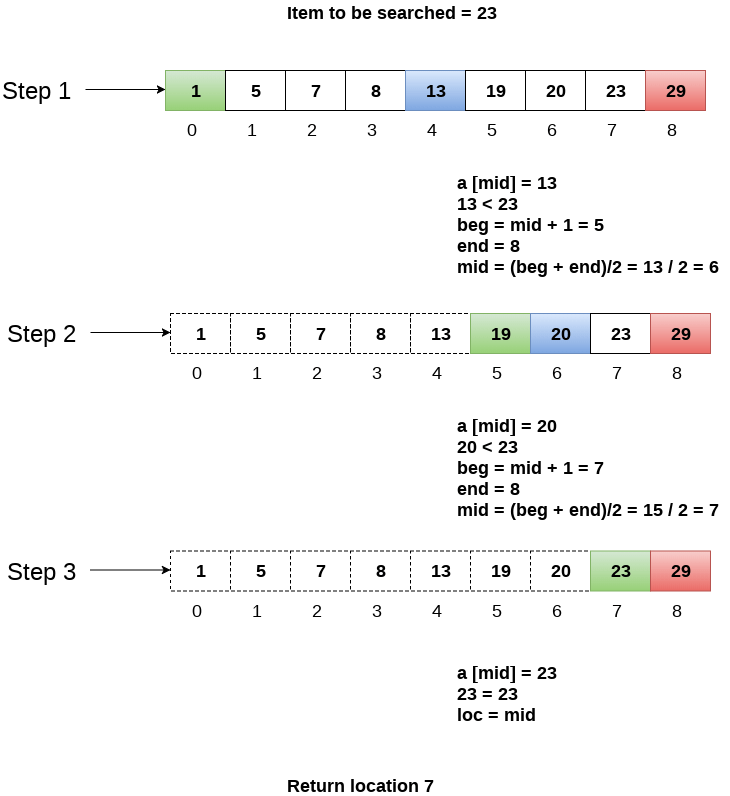
1. BEG = 0
2. END = 8ron
3. MID = 4
4. a[mid] = a[4] = 13 < 23, therefore

**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.



## Binary Search Program using Recursion

### Java

1. **import** java.util.\*;
2. **public** **class** BinarySearch {
3. **public** **static** **void** main(String[] args) {
4. **int**[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
5. **int** item, location = -1;
6. System.out.println("Enter the item which you want to search");
7. Scanner sc = **new** Scanner(System.in);
8. item = sc.nextInt();
9. location = binarySearch(arr,0,9,item);
10. **if**(location != -1)
11. System.out.println("the location of the item is "+location);
12. **else**
13. System.out.println("Item not found");
14. }
15. **public** **static** **int** binarySearch(**int**[] a, **int** beg, **int** end, **int** item)
16. {
17. **int** mid;
18. **if**(end >= beg)
19. {
20. mid = (beg + end)/2;
21. **if**(a[mid] == item)
22. {
23. **return** mid+1;
24. }
25. **else** **if**(a[mid] < item)
26. {
27. **return** binarySearch(a,mid+1,end,item);
28. }
29. **else**
30. {
31. **return** binarySearch(a,beg,mid-1,item);
32. }
34. }
35. **return** -1;
36. }
37. }

**Output:**

Enter the item which you want to search 45 the location of the item is 5

**Sortings**

# Bubble Sort

In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Consider an array A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.

1. In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.
3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of this pass. The smallest element of the list is placed at the first index of the list.

## Algorithm :

* **Step 1**: Repeat Step 2 For i = 0 to N-1
* **Step 2**: Repeat For J = i + 1 to N - I
* **Step 3**: IF A[J] > A[i]  
  SWAP A[J] and A[i]  
  [END OF INNER LOOP]  
  [END OF OUTER LOOP
* **Step 4**: EXIT

## Complexity

|  |  |
| --- | --- |
| **Scenario** | **Complexity** |
| Space | O(1) |
| Worst case running time | O(n2) |
| Average case running time | O(n) |
| Best case running time | O(n2) |

## Java Program

1. **public** **class** BubbleSort {
2. **public** **static** **void** main(String[] args) {
3. **int**[] a = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
4. **for**(**int** i=0;i<10;i++)
5. {
6. **for** (**int** j=0;j<10;j++)
7. {
8. **if**(a[i]<a[j])
9. {
10. **int** temp = a[i];
11. a[i]=a[j];
12. a[j] = temp;
13. }
14. }
15. }
16. System.out.println("Printing Sorted List ...");
17. **for**(**int** i=0;i<10;i++)
18. {
19. System.out.println(a[i]);
20. }
21. }
22. }

**Output:**

Printing Sorted List . . . 7910122334344478 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.
3. 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
4. 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(n2) |
| 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

## Java Program

1. **public** **class** QuickSort {
2. **public** **static** **void** main(String[] args) {
3. **int** i;
4. **int**[] arr={90,23,101,45,65,23,67,89,34,23};
5. quickSort(arr, 0, 9);
6. System.out.println("\n The sorted array is: \n");
7. **for**(i=0;i<10;i++)
8. System.out.println(arr[i]);
9. }
10. **public** **static** **int** partition(**int** a[], **int** beg, **int** end)
11. {
13. **int** left, right, temp, loc, flag;
14. loc = left = beg;
15. right = end;
16. flag = 0;
17. **while**(flag != 1)
18. {
19. **while**((a[loc] <= a[right]) && (loc!=right))
20. right--;
21. **if**(loc==right)
22. flag =1;
23. elseif(a[loc]>a[right])
24. {
25. temp = a[loc];
26. a[loc] = a[right];
27. a[right] = temp;
28. loc = right;
29. }
30. **if**(flag!=1)
31. {
32. **while**((a[loc] >= a[left]) && (loc!=left))
33. left++;
34. **if**(loc==left)
35. flag =1;
36. elseif(a[loc] <a[left])
37. {
38. temp = a[loc];
39. a[loc] = a[left];
40. a[left] = temp;
41. loc = left;
42. }
43. }
44. }
45. returnloc;
46. }
47. **static** **void** quickSort(**int** a[], **int** beg, **int** end)
48. {
50. **int** loc;
51. **if**(beg<end)
52. {
53. loc = partition(a, beg, end);
54. quickSort(a, beg, loc-1);
55. quickSort(a, loc+1, end);
56. }
57. }
58. }

**Output:**

The sorted array is: 232323344565678990101

# 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

1. #include<stdio.h>
2. **int** temp;
4. **void** heapify(**int** arr[], **int** size, **int** i)
5. {
6. **int** largest = i;
7. **int** left = 2\*i + 1;
8. **int** right = 2\*i + 2;
10. **if** (left < size && arr[left] >arr[largest])
11. largest = left;
13. **if** (right < size && arr[right] > arr[largest])
14. largest = right;
16. **if** (largest != i)
17. {
18. temp = arr[i];
19. arr[i]= arr[largest];
20. arr[largest] = temp;
21. heapify(arr, size, largest);
22. }
23. }
25. **void** heapSort(**int** arr[], **int** size)
26. {
27. **int** i;
28. **for** (i = size / 2 - 1; i >= 0; i--)
29. heapify(arr, size, i);
30. **for** (i=size-1; i>=0; i--)
31. {
32. temp = arr[0];
33. arr[0]= arr[i];
34. arr[i] = temp;
35. heapify(arr, i, 0);
36. }
37. }
39. **void** main()
40. {
41. **int** arr[] = {1, 10, 2, 3, 4, 1, 2, 100, 23, 2};
42. **int** i;
43. **int** size = sizeof(arr)/sizeof(arr[0]);
45. heapSort(arr, size);
47. printf("printing sorted elements\n");
48. **for** (i=0; i<size; ++i)
49. printf("%d\n",arr[i]);
50. }

**Output:**

printing sorted elements 11222341023100

# Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

* In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
* In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
* In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].

Therefore, by following the above explained process, the elements A[0], A[1], A[2],...., A[n-1] are sorted.

## Example

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

**A = {10, 2, 3, 90, 43, 56}.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Pass** | **Pos** | **A[0]** | **A[1]** | **A[2]** | **A[3]** | **A[4]** | **A[5]** |
| 1 | 1 | 2 | 10 | 3 | 90 | 43 | 56 |
| 2 | 2 | 2 | 3 | 10 | 90 | 43 | 56 |
| 3 | 2 | 2 | 3 | 10 | 90 | 43 | 56 |
| 4 | 4 | 2 | 3 | 10 | 43 | 90 | 56 |
| 5 | 5 | **2** | **3** | **10** | **43** | **56** | **90** |

Sorted A = {2, 3, 10, 43, 56, 90}

## Complexity

|  |  |  |  |
| --- | --- | --- | --- |
| **Complexity** | **Best Case** | **Average Case** | **Worst Case** |
| Time | Ω(n) | θ(n2) | o(n2) |
| Space |  |  | o(1) |

## Algorithm

**SELECTION SORT(ARR, N)**

* **Step 1**: Repeat Steps 2 and 3 for K = 1 to N-1
* **Step 2**: CALL SMALLEST(ARR, K, N, POS)
* **Step 3**: SWAP A[K] with ARR[POS]  
  [END OF LOOP]
* **Step 4**: EXIT

**SMALLEST (ARR, K, N, POS)**

* **Step 1**: [INITIALIZE] SET SMALL = ARR[K]
* **Step 2**: [INITIALIZE] SET POS = K
* **Step 3**: Repeat for J = K+1 to N -1  
  IF SMALL > ARR[J]  
  SET SMALL = ARR[J]  
  SET POS = J  
  [END OF IF]  
  [END OF LOOP]
* **Step 4**: RETURN POS

# 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.

1. 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 < INDEX  
  SET ARR[K] = TEMP[K]  
  SET K = K + 1  
  [END OF LOOP]
* **Step 6**: Exit

**MERGE\_SORT(ARR, BEG, END)**

* **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

1. **public** **class** MyMergeSort
2. {
3. **void** merge(**int** arr[], **int** beg, **int** mid, **int** end)
4. {
6. **int** l = mid - beg + 1;
7. **int** r = end - mid;
9. intLeftArray[] = **new** **int** [l];
10. intRightArray[] = **new** **int** [r];
12. **for** (**int** i=0; i<l; ++i)
13. LeftArray[i] = arr[beg + i];
15. **for** (**int** j=0; j<r; ++j)
16. RightArray[j] = arr[mid + 1+ j];

19. **int** i = 0, j = 0;
20. **int** k = beg;
21. **while** (i<l&&j<r)
22. {
23. **if** (LeftArray[i] <= RightArray[j])
24. {
25. arr[k] = LeftArray[i];
26. i++;
27. }
28. **else**
29. {
30. arr[k] = RightArray[j];
31. j++;
32. }
33. k++;
34. }
35. **while** (i<l)
36. {
37. arr[k] = LeftArray[i];
38. i++;
39. k++;
40. }
42. **while** (j<r)
43. {
44. arr[k] = RightArray[j];
45. j++;
46. k++;
47. }
48. }
50. **void** sort(**int** arr[], **int** beg, **int** end)
51. {
52. **if** (beg<end)
53. {
54. **int** mid = (beg+end)/2;
55. sort(arr, beg, mid);
56. sort(arr , mid+1, end);
57. merge(arr, beg, mid, end);
58. }
59. }
60. **public** **static** **void** main(String args[])
61. {
62. intarr[] = {90,23,101,45,65,23,67,89,34,23};
63. MyMergeSort ob = **new** MyMergeSort();
64. ob.sort(arr, 0, arr.length-1);
66. System.out.println("\nSorted array");
67. **for**(**int** i =0; i<arr.length;i++)
68. {
69. System.out.println(arr[i]+"");
70. }
71. }
72. }

**Output:**

Sorted array 232323344565678990101

# 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
* **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

## Java Program

1. **public** **class** Radix\_Sort {
2. **public** **static** **void** main(String[] args) {
3. **int** i;
4. Scanner sc = **new** Scanner(System.in);
5. **int**[] a = {90,23,101,45,65,23,67,89,34,23};
6. radix\_sort(a);
7. System.out.println("\n The sorted array is: \n");
8. **for**(i=0;i<10;i++)
9. System.out.println(a[i]);
10. }
12. **static** **int** largest(inta[])
13. {
14. **int** larger=a[0], i;
15. **for**(i=1;i<10;i++)
16. {
17. **if**(a[i]>larger)
18. larger = a[i];
19. }
20. returnlarger;
21. }
22. **static** **void** radix\_sort(inta[])
23. {
24. **int** bucket[][]=newint[10][10];
25. **int** bucket\_count[]=newint[10];
26. **int** i, j, k, remainder, NOP=0, divisor=1, larger, pass;
27. larger = largest(a);
28. **while**(larger>0)
29. {
30. NOP++;
31. larger/=10;
32. }
33. **for**(pass=0;pass<NOP;pass++) // Initialize the buckets
34. {
35. **for**(i=0;i<10;i++)
36. bucket\_count[i]=0;
37. **for**(i=0;i<10;i++)
38. {
39. // sort the numbers according to the digit at passth place
40. remainder = (a[i]/divisor)%10;
41. bucket[remainder][bucket\_count[remainder]] = a[i];
42. bucket\_count[remainder] += 1;
43. }
44. // collect the numbers after PASS pass
45. i=0;
46. **for**(k=0;k<10;k++)
47. {
48. **for**(j=0;j<bucket\_count[k];j++)
49. {
50. a[i] = bucket[k][j];
51. i++;
52. }
53. }
54. divisor \*= 10;
55. }
56. }
57. }

**Output:**

The sorted array is:232323344565678990101