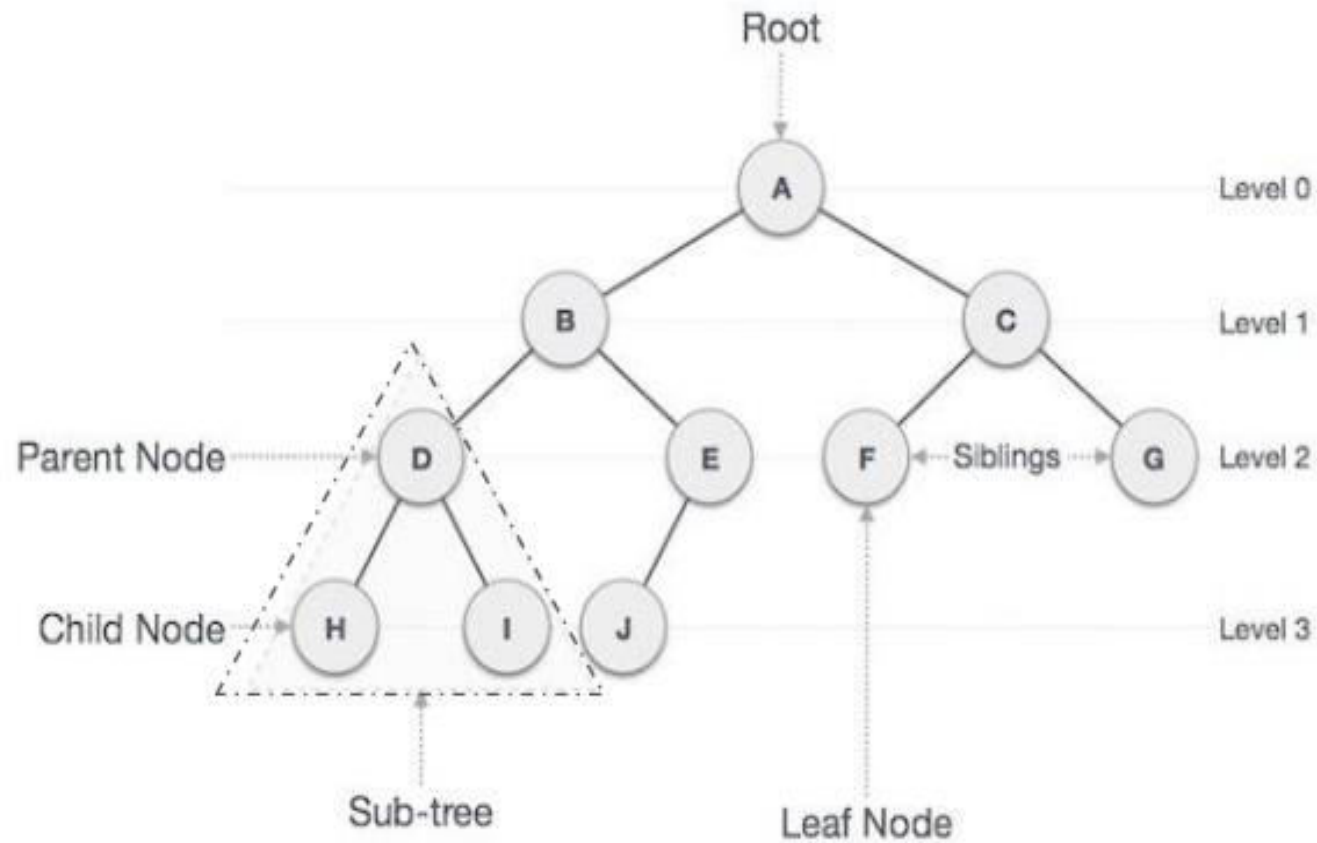


Trees

Basic Terminology

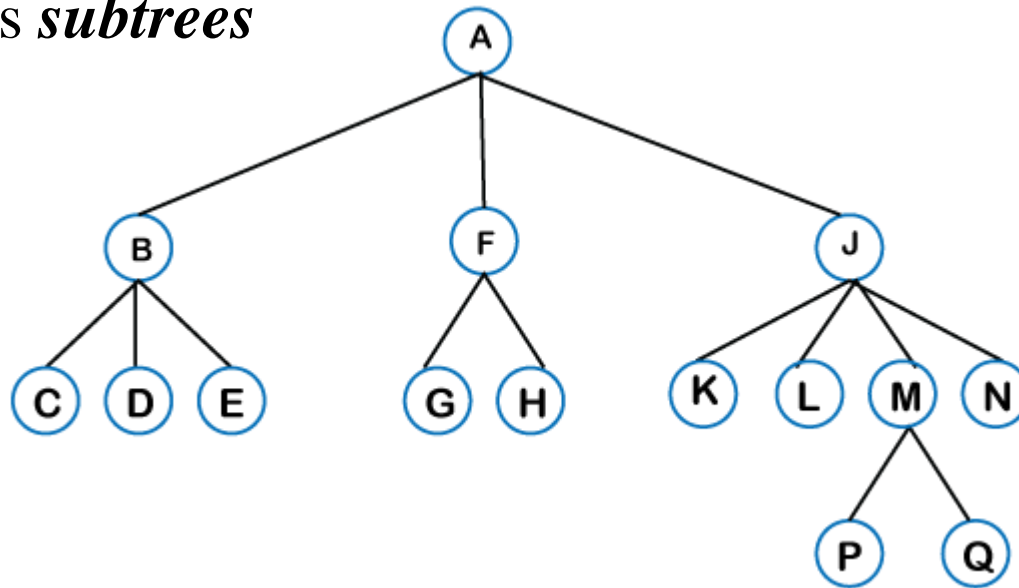


- Introduction to Tree
- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements 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. Each node contains some data and the link or reference of other nodes that can be called children.

- Root is a special node in a tree. The entire tree originates from it. It does not have a parent. E.g. A
- Parent node is an immediate predecessor of a node. E.g. B is parent of D and E
- All immediate successors of a node are its children. D and E are children of B.
- Node which does not have any child is called as leaf. E.g. H, I, J, F and G are leaf nodes
- Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf. E.g. Line between A and B is edge
- Nodes with the same parent are called siblings. E.g. D and E are siblings
- Path is a number of successive edges from source node to destination node. A-B-E-J is path from node A to E
- The Depth The number of edges from the tree's node to the root is.
- The Height of a tree is the number of edges from the node to the deepest leaf. The tree height is also considered the root height E.g. Height of A is edges between A and H, as that is the longest path, which is 3.
- Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on. E. g. Level of H, I & J is 3. Level of D, E, F & G is 2
- Degree of a node represents the number of children of a node. E.g. Degree of D is 2 and of E is 1
- Descendants of a node represent subtree. E.g. Nodes D, H, I represent one subtree.
- 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.
- Descendant: The immediate successor of the given node is known as a descendant of a node.

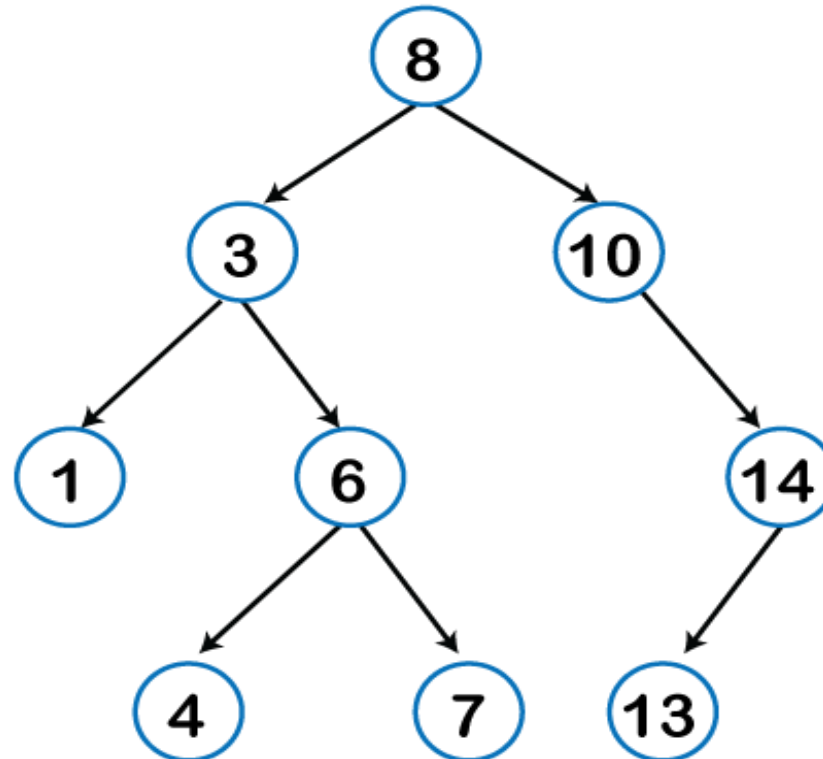
- **General tree:**

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*

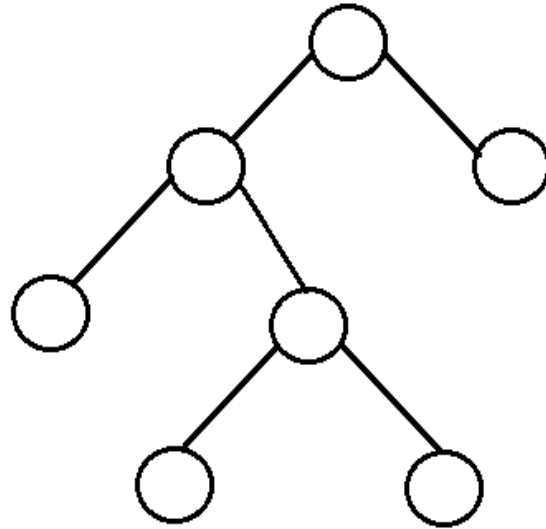


- Binary Tree:

In a Binary tree, every node can have at most 2 children, left and right. In diagram below, node 3 forms left subtree and node 10 forms right subtree.

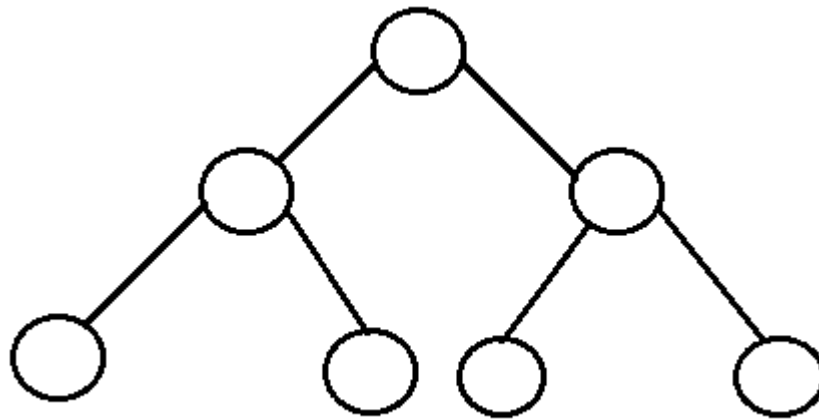


- Full Binary Tree: It is a tree in which every node in the tree has either 0 or 2 children



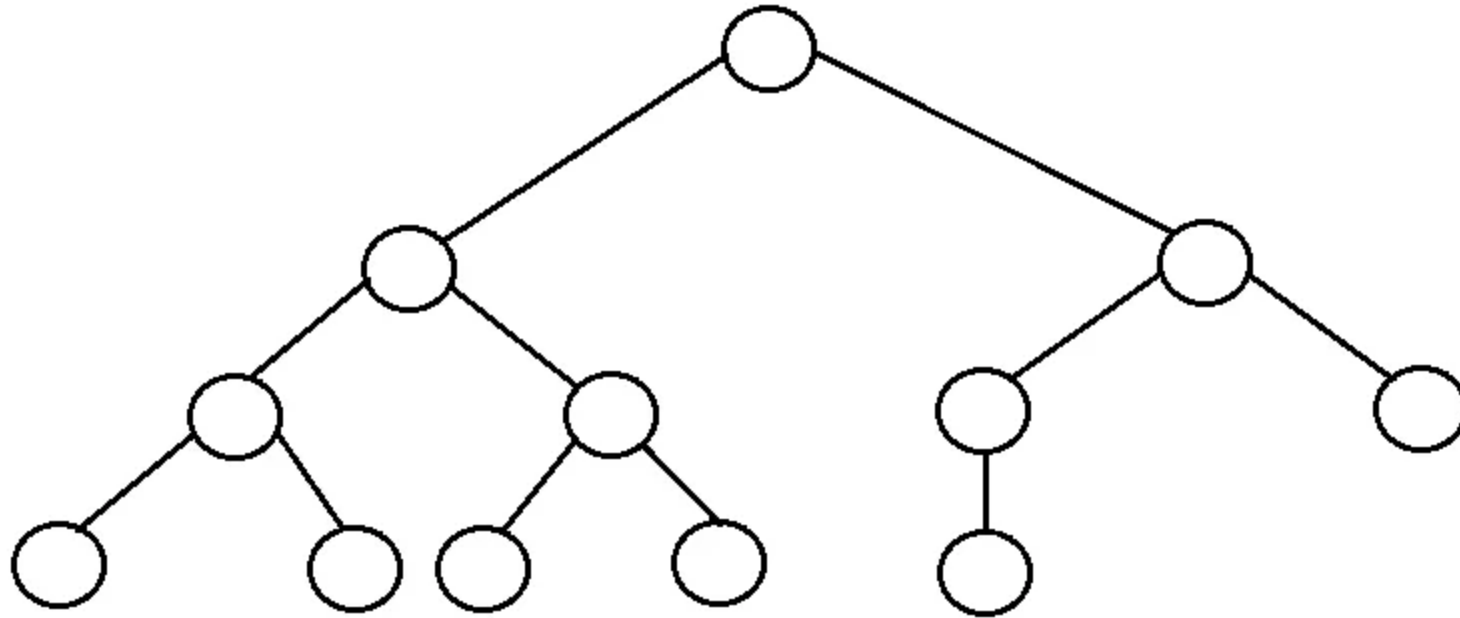
- The number of nodes, n , in a full binary tree is at least $n = 2^h - 1$, and at most $n = 2^{h+1} - 1$, where h is the height of the tree.
- The number of leaf nodes l , in a full binary tree is number, L of internal nodes + 1, i.e, $l = L + 1$.

Perfect binary tree: It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



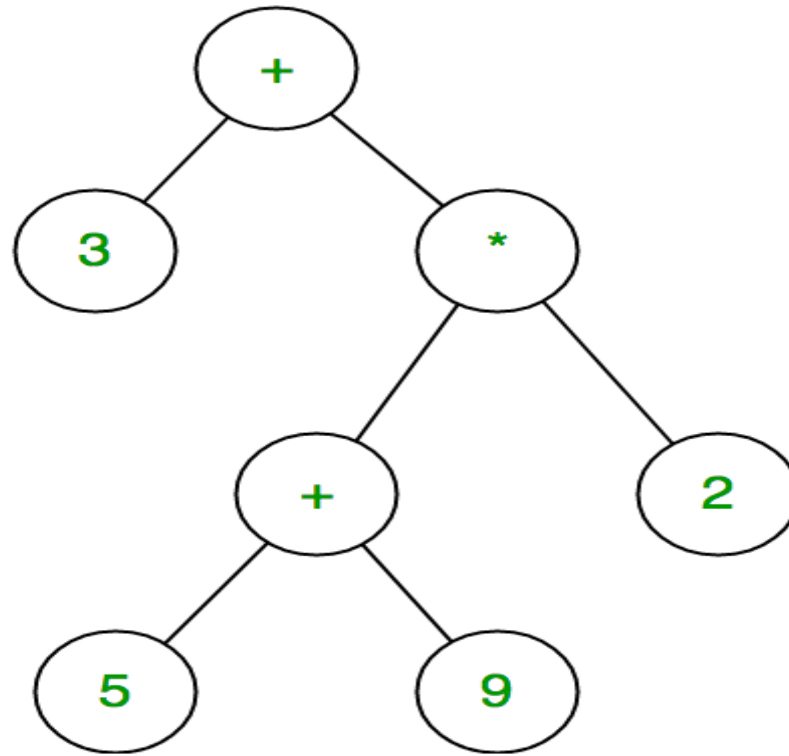
- A perfect binary tree with l leaves has $n = 2 * l - 1$ nodes.
- In perfect full binary tree, $l = 2^h$ and $n = 2^{(h+1)} - 1$ where, n is number of nodes, h is height of tree and l is number of leaf nodes

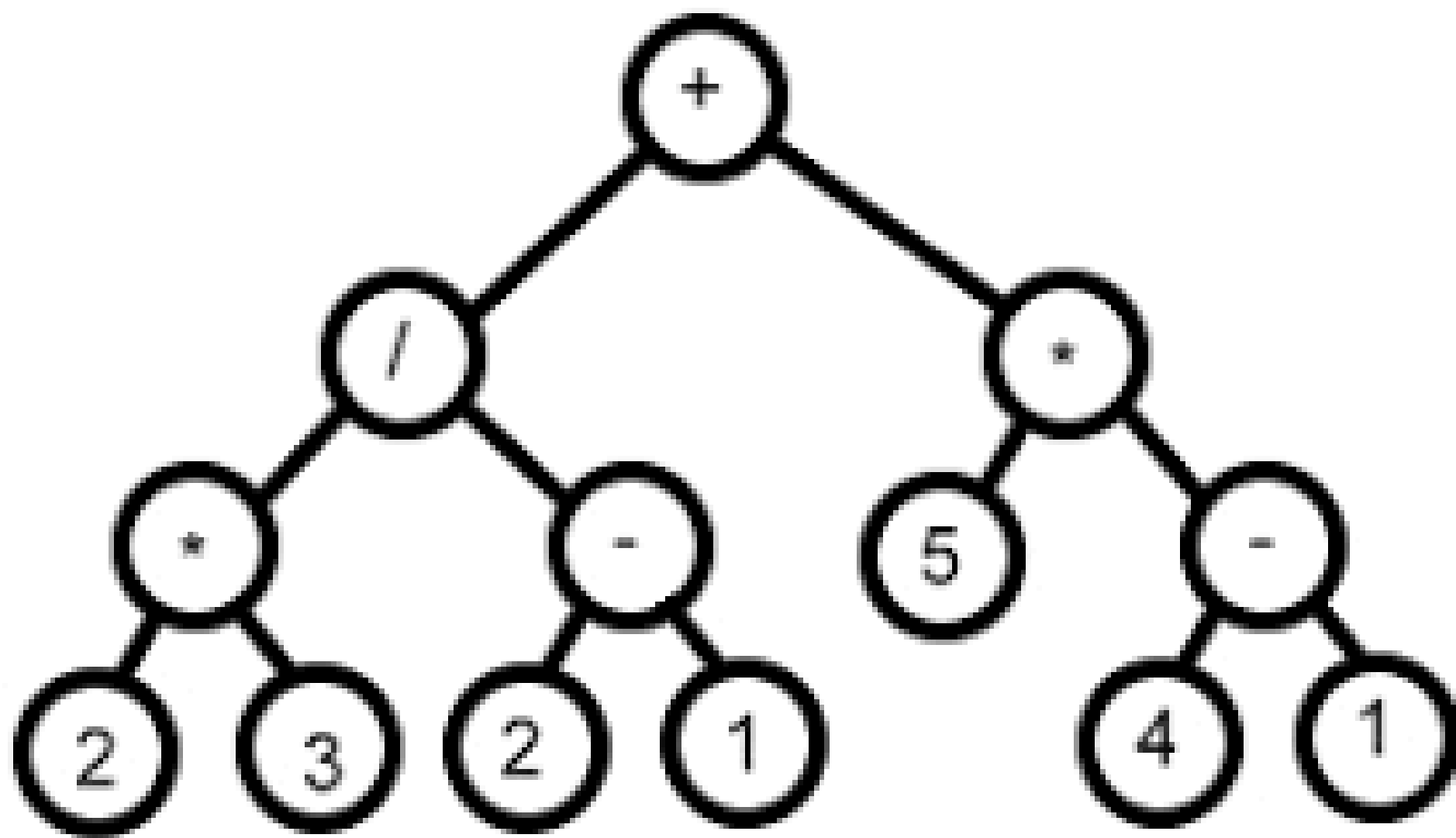
- **Complete Binary Tree:** A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.



In complete binary tree, the left and right children of node K is $2*K$ and $2*K+1$ respectively. The parent of node K is $\lfloor K/2 \rfloor$.

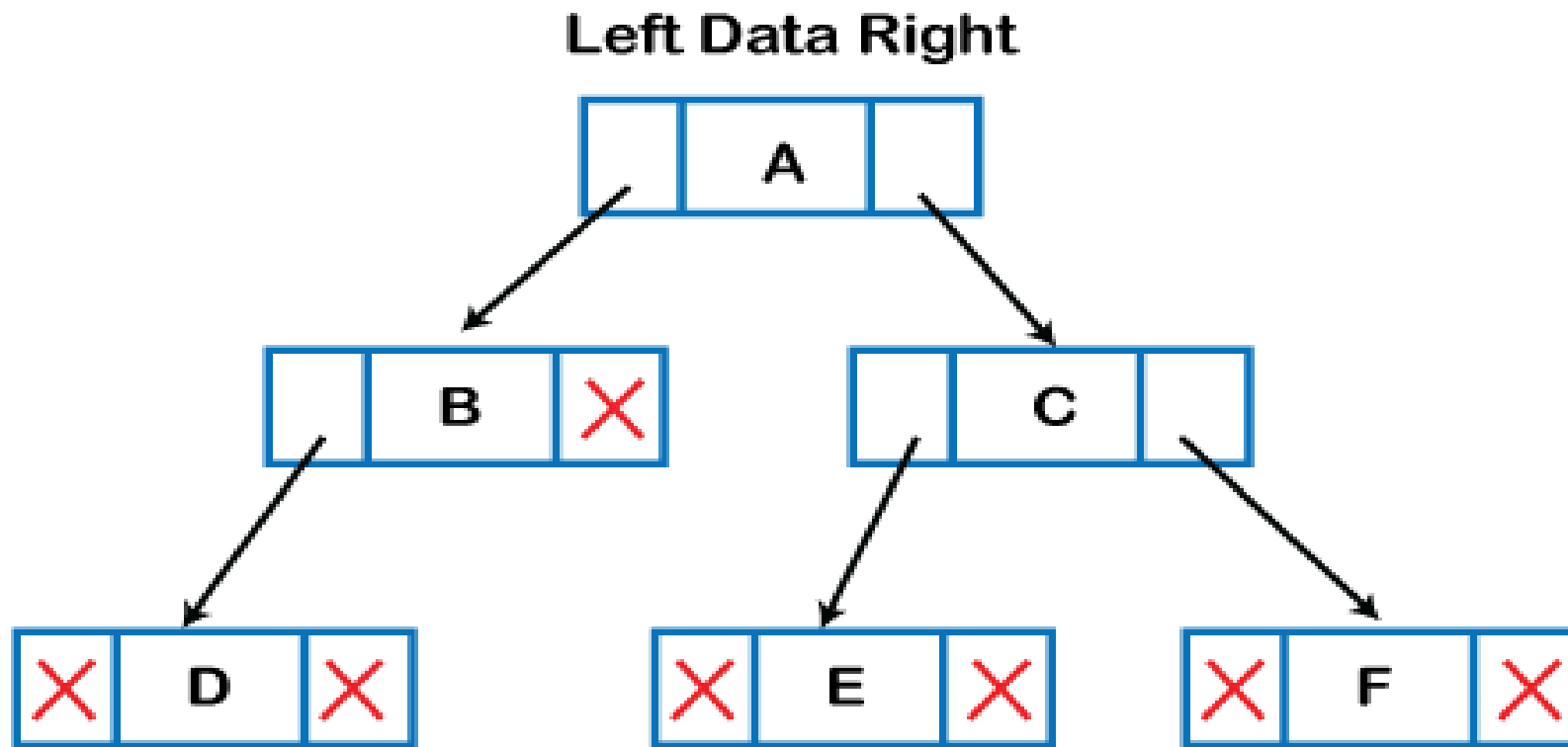
- Expression Tree: The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand.
- Expression tree for $3 + ((5+9)*2)$ would be:





Expression tree for $2 * 3 / (2 - 1) + 5 * (4 - 1)$

Representation of Binary Tree in memory



- The above figure shows the representation of the tree data structure in the memory. In linked and dynamic representation, the linked list data structure is used. Each node constitutes of a data part and two link parts. The two link parts store address of left and right child nodes. Data part is used to store the information about the binary tree element. This is a better representation as nodes can be added or deleted at any location. Memory utilization is better in this binary tree representation.

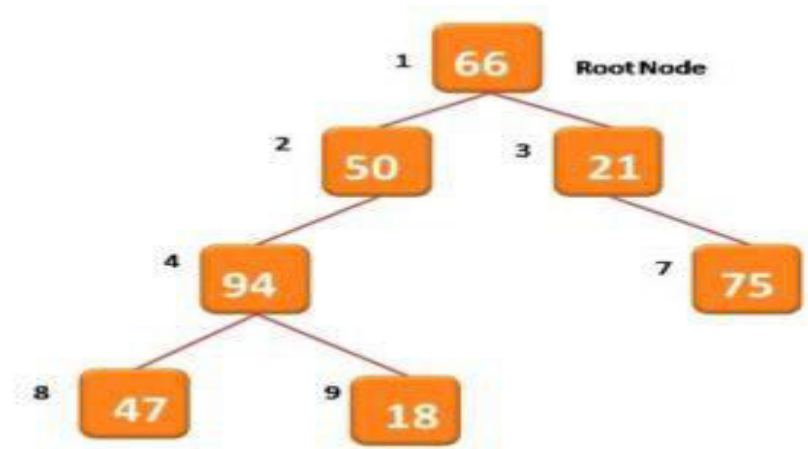
```
struct node
{
    int data;
    struct node *left;
    struct node *right;
}
```

Sequential Representation of binary Trees:

Suppose T is a binary tree that is complete or nearly complete. Then there is an efficient way of maintaining T in memory called sequential representation of T . This representation uses only a single linear array $Tree$ as follows:

- The root R of T is stored in $Tree[1]$.
- If node N occupies $Tree[k]$, then its left child is stored in $Tree[2*k]$ and its right child is stored in $Tree[2*k + 1]$.

Again Null is used to indicate an empty subtree. If $Tree[1] = \text{Null}$ then the tree is empty.



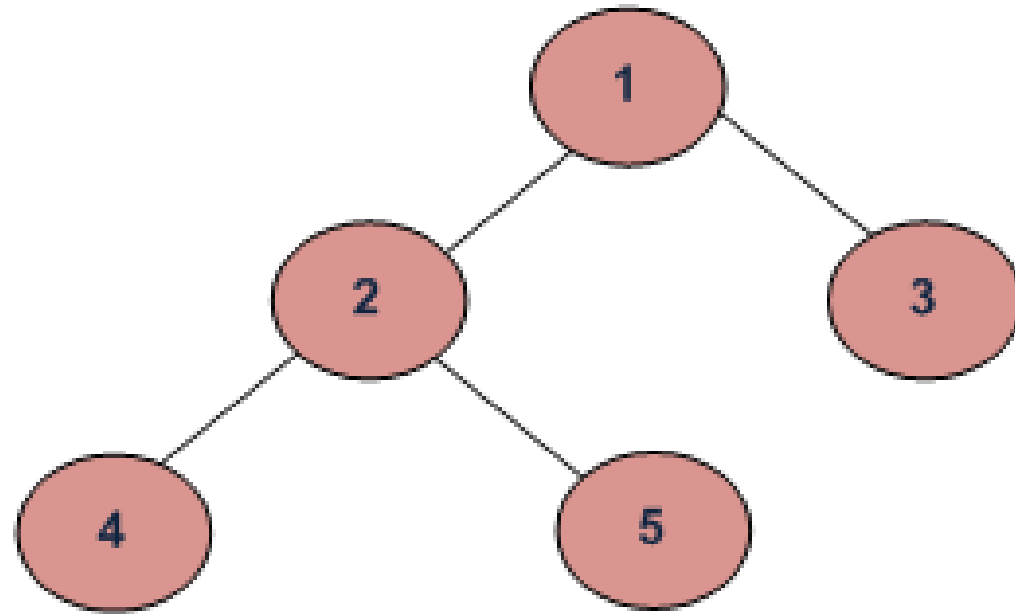
1	2	3	4	5	6	7	8	9
66	50	21	94			75	47	18

Traversing Binary Tree

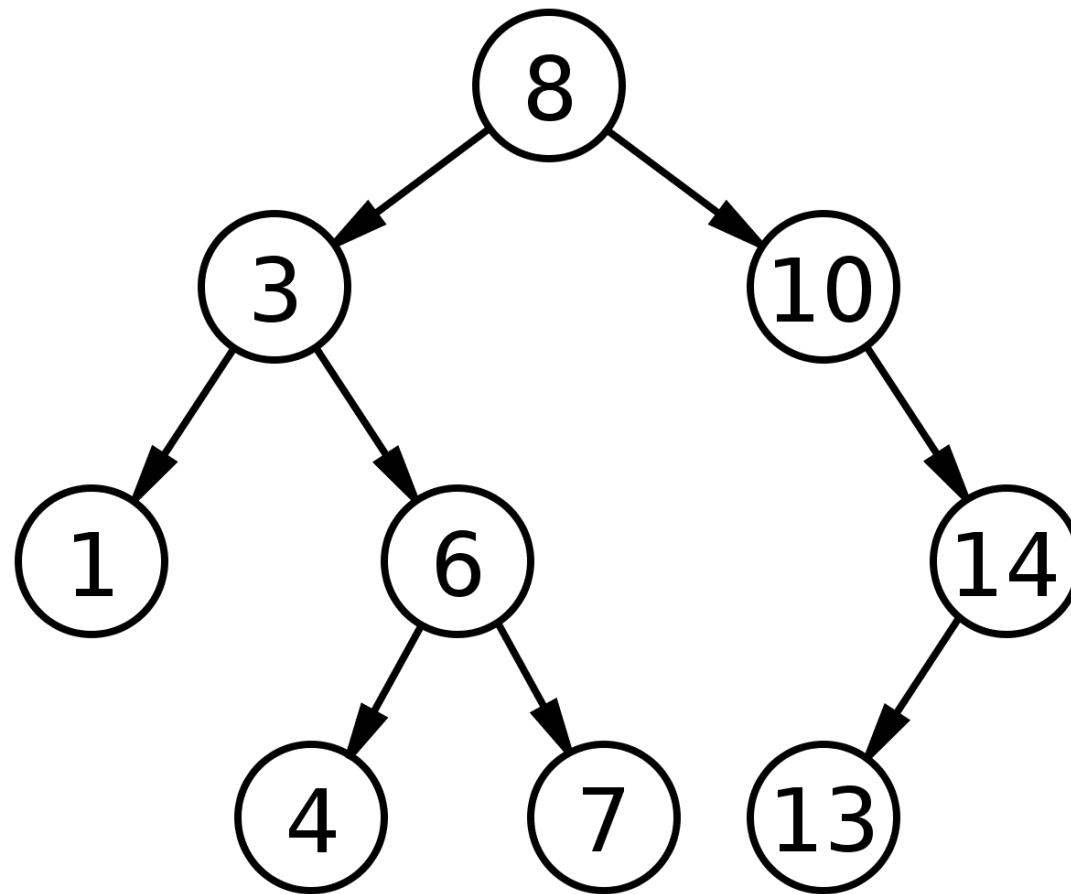
- There are three standard ways of traversing a binary tree T with root R. These three algorithms are called preorder, inorder, postorder, are as follows:
- Preorder
 1. Process the root R.
 2. Traverse the left subtree of R in preorder.
 3. Traverse the right subtree of R in preorder.
- Inorder
 1. Traverse the left subtree of R in inorder.
 2. Process the root R.
 3. Traverse the right subtree of R in inorder.
- Postorder
 1. Traverse the left subtree of R in postorder.
 2. Traverse the right subtree of R in postorder.
 3. Process the root R.

The three algorithms are sometimes called , respectively, the root-left-right(**RLR**)traversal, the left-root-right(**LRR**) traversal and the left-right-root(**LRR**) traversal.

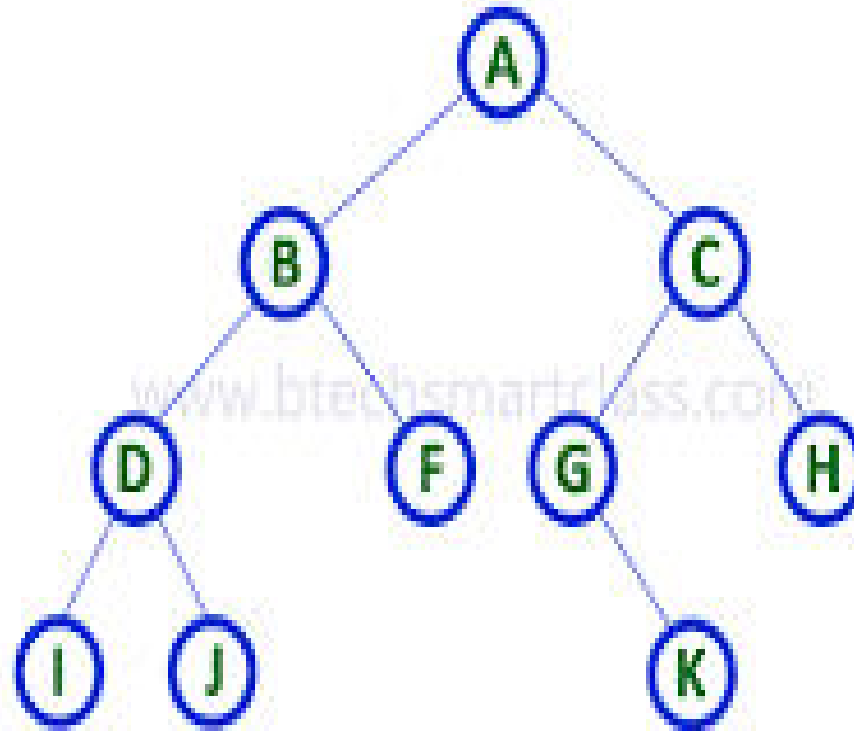
Preorder: 1 2 4 5 3
Inorder: 4 2 5 1 3
Postorder: 4 5 2 3 1



Preorder: 8 3 1 6 4 7 10 14 13
Inorder: 1 3 4 6 7 8 10 13 14
postorder: 1 4 7 6 3 13 14 10 8



Preorder: A B D I J F C G K H
Inorder: I D J B F A G K C H
Postorder: I J D F B K G H C A



InOrder(root) visits nodes in the following order:

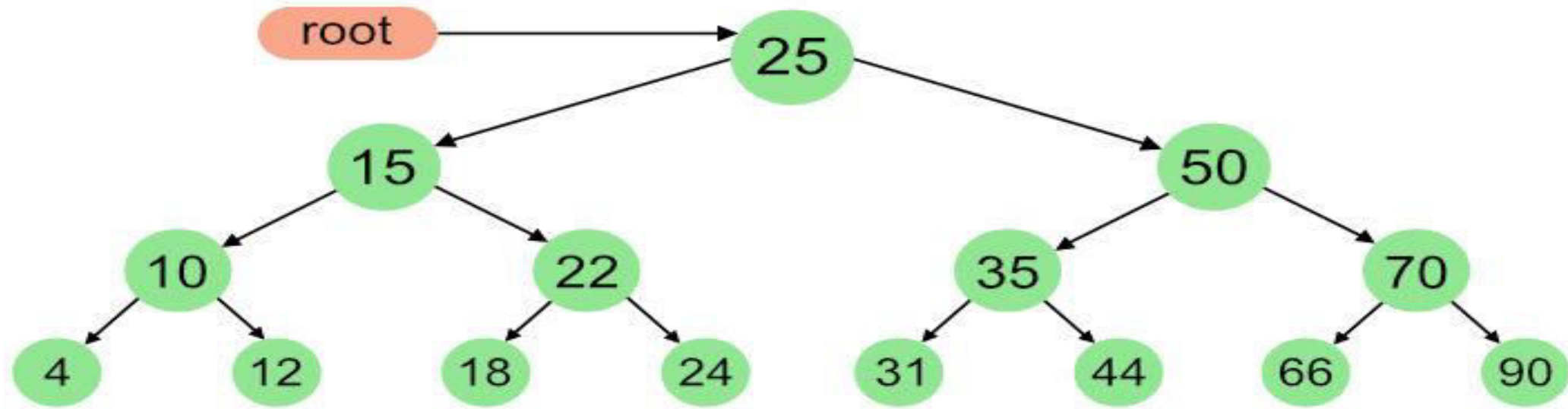
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

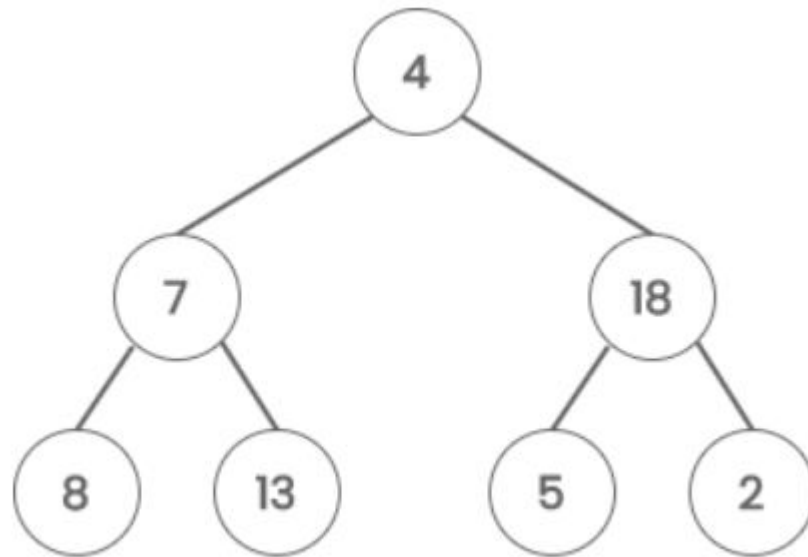
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Pre-order Traversal Without Recursion

The following operations are performed to traverse a binary tree in pre-order using a stack:

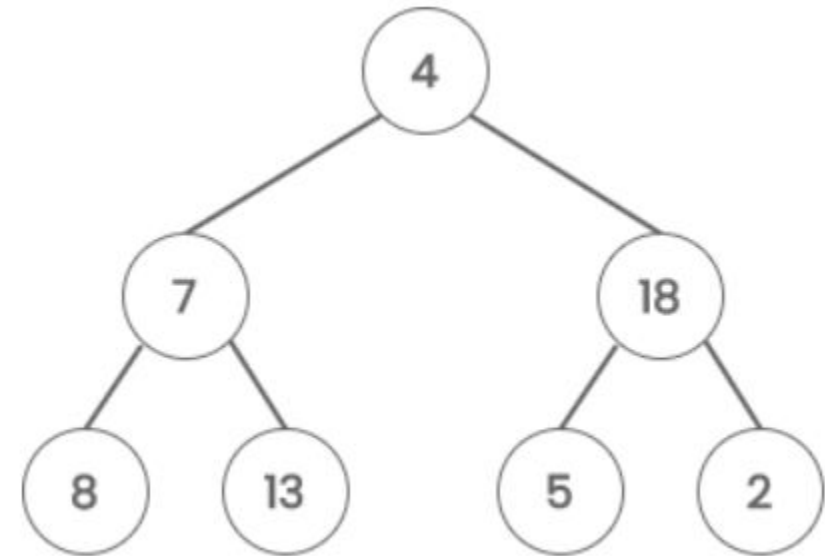
1. Start with root node and push onto stack.
2. Repeat while the stack is not empty
 - a. POP & store the top element (PTR) from the stack and process the node.
 - b. PUSH the right child of PTR onto to stack.
 - c. PUSH the left child of PTR onto to stack.



In-order Traversal Without Recursion

The following operations are performed to traverse a binary tree in in-order using a stack:

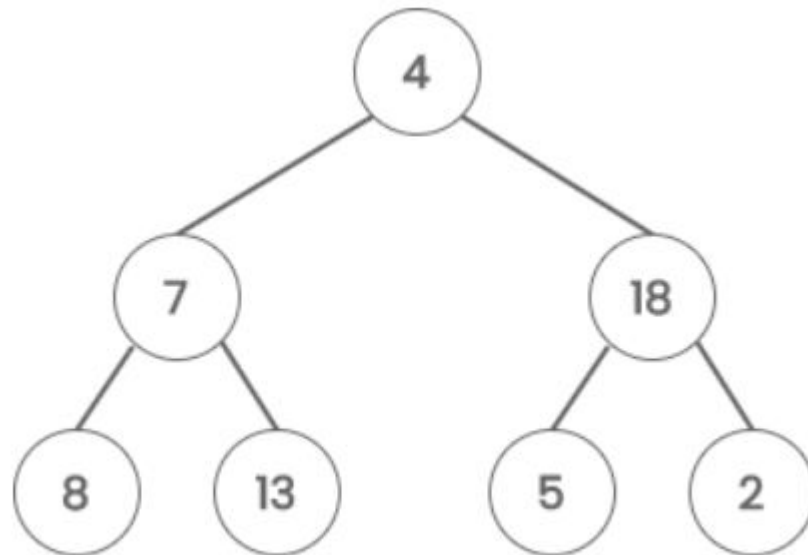
1. Create an empty stack
2. Start from the root, call it PTR.
3. Push PTR onto stack and set $\text{PTR} = \text{PTR} \rightarrow \text{left}$ until PTR is not NULL.
4. If PTR is NULL and stack is not empty, then
 - i. Pop element from stack and print it
 - ii. set $\text{PTR} = \text{popped_ele} \rightarrow \text{right}$.
 - iii. Go to step 3
5. If PTR is NULL && stack is empty then,
we are done.



Post-order Traversal Without Recursion

The following operations are performed to traverse a binary tree in post-order using two stacks S1 and S2:

1. Push root to first stack S1.
2. Loop while S1 is not empty
 - i. pop top node of S1 and push to S2.
 - ii. Push node->left & node->right to S1
5. Print contents of S2.



Two trees are equal or identical when they have same data and arrangement of data is also same.

To identify if two trees are identical, we need to traverse both trees simultaneously, and while traversing we need to compare data and children of the trees.

sameTree(tree1, tree2)

1. If both trees are empty then return 1.
2. Else If both trees are non - empty
 - (a) Check data of the root nodes (tree1->data == tree2->data)
 - (b) Check left subtrees recursively i.e.,
call sameTree(tree1->left_subtree, tree2->left_subtree)
 - (c) Check right subtrees recursively i.e.,
call sameTree(tree1->right_subtree, tree2->right_subtree)
 - (d) If a,b and c are true then return 1.
- 3 Else return 0 (one is empty and other is not)

Given a binary tree, we have to create a clone of given binary tree. We will traverse the binary tree using pre order traversal and create a clone of every node. The problem of creating a duplicate tree can be broken down to sub problems of creating duplicate sub trees.

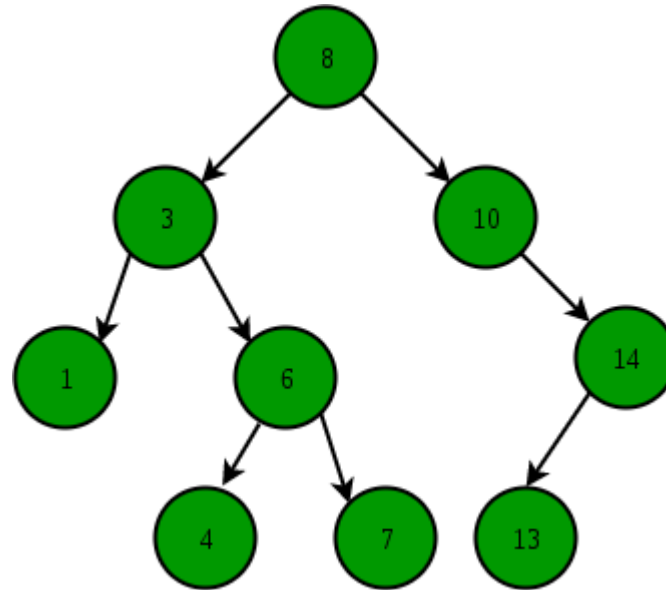
Algorithm to create a duplicate binary tree

Let "root" be the root node of binary tree.

1. If root is equal to NULL, then return NULL.
2. Create a new node and copy data of root node into new node.
3. Recursively, create clone of left sub tree and make it left sub tree of new node.
4. Recursively, create clone of right sub tree and make it right sub tree of new node.
5. Return new node.

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

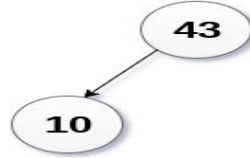


Create BST -43, 10, 79, 90, 12, 54, 11, 9,
50

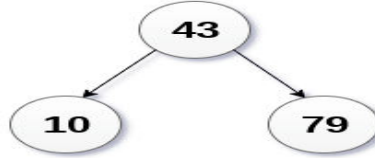
Step 1



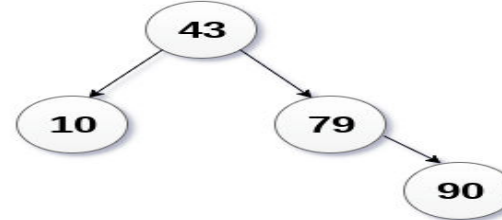
Step 2



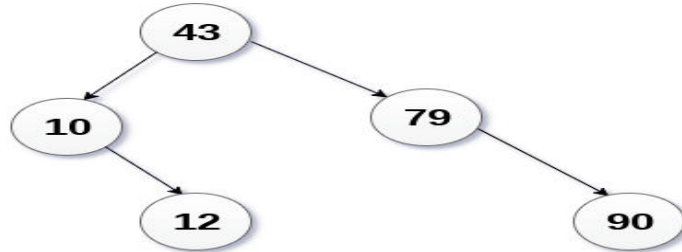
Step 3



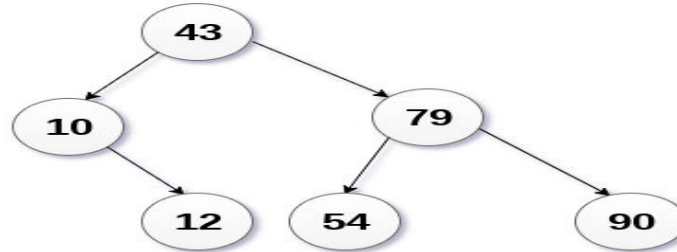
Step 4



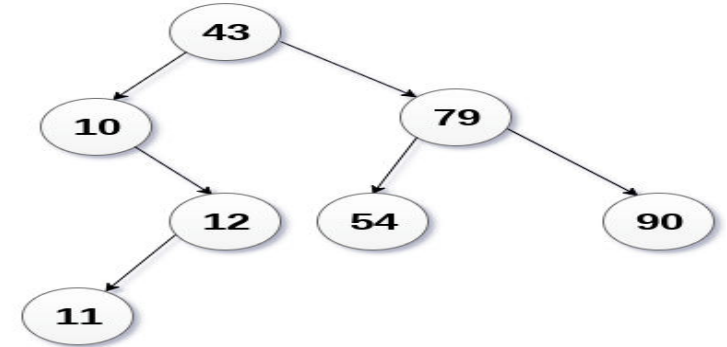
Step 5



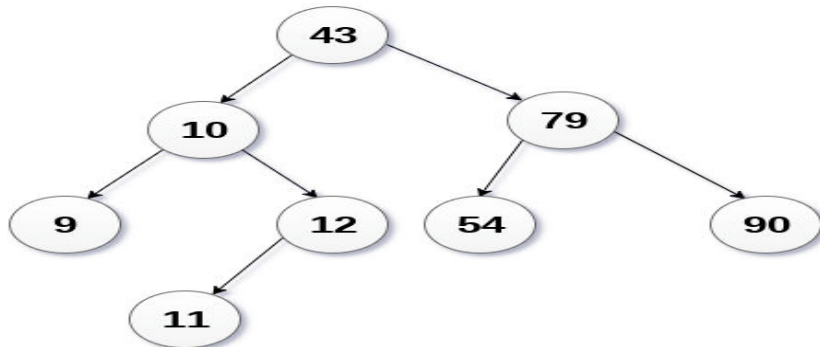
Step 6



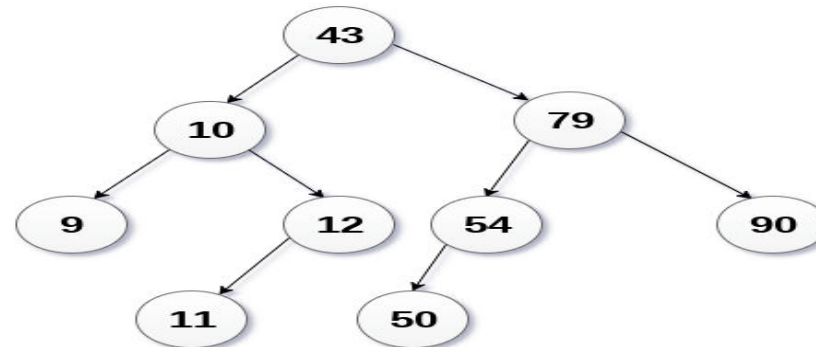
Step 7



Step 8



Step 9



Binary search Tree Creation

Search Operation in BST

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Search Operation in BST

Search (ROOT, ITEM)

Step 1: IF ROOT -> DATA = ITEM OR ROOT = NULL

Return ROOT

ELSE

IF ROOT -> DATA < ITEM

Return search(ROOT -> LEFT, ITEM)

ELSE

Return search(ROOT -> RIGHT, ITEM)

[END OF IF]

[END OF IF]

Step 2: END

Insert Operation in BST

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must not violate the property of binary search tree at each value.

Step 1 - Allocate the memory for tree.

Step 2 - Set the data part to the value and set the left and right pointer of tree, point to NULL.

Step 3 - If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.

Step 4 - Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.

Step 5 - If this is false, then perform this operation recursively with the right sub-tree of the root.

Insert Operation in BST

Insert (TREE, ITEM)

Step 1: IF TREE = NULL

 Allocate memory for TREE

 SET TREE -> DATA = ITEM

 SET TREE -> LEFT = TREE -> RIGHT = NULL

ELSE

 IF ITEM < TREE -> DATA

 Insert(TREE -> LEFT, ITEM)

ELSE

 Insert(TREE -> RIGHT, ITEM)

[END OF IF]

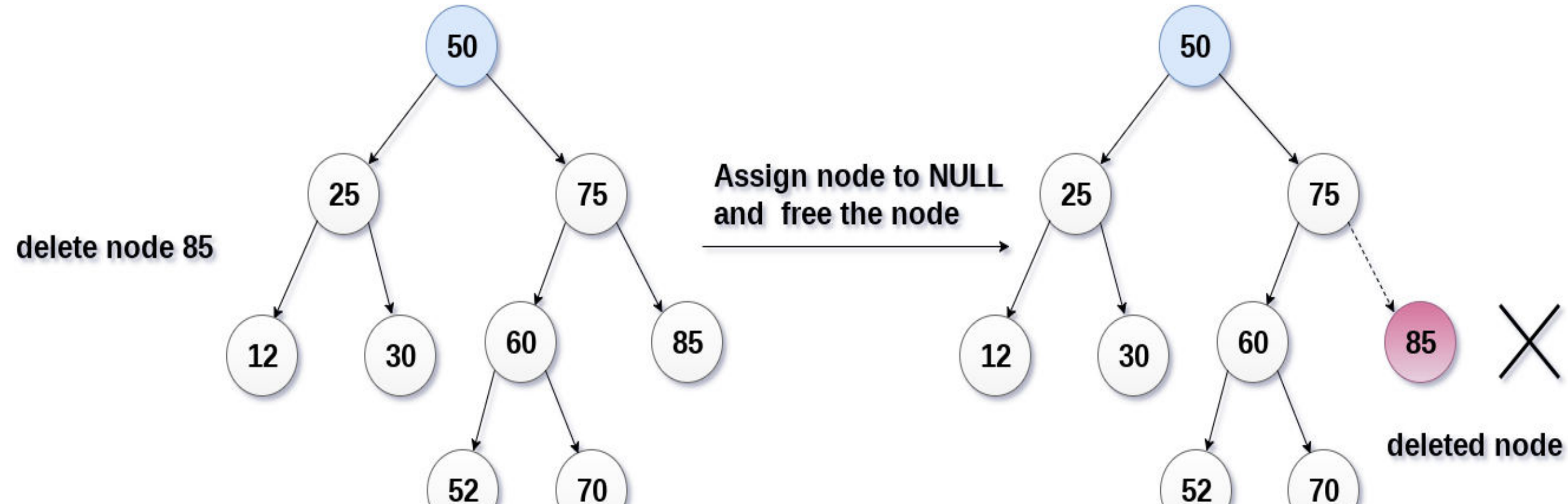
[END OF IF]

Step 2: END

- **Delete operation in BST**

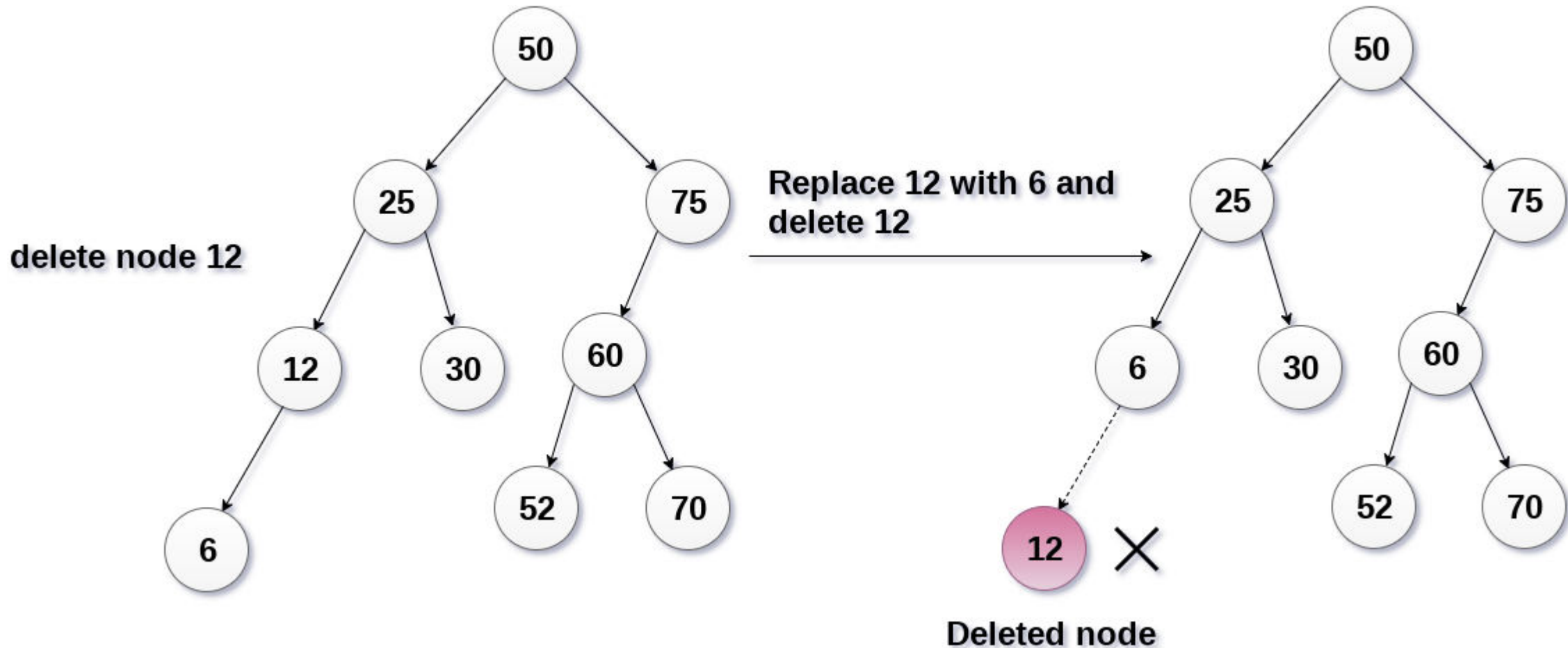
Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

- The node to be deleted is a leaf node: It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.



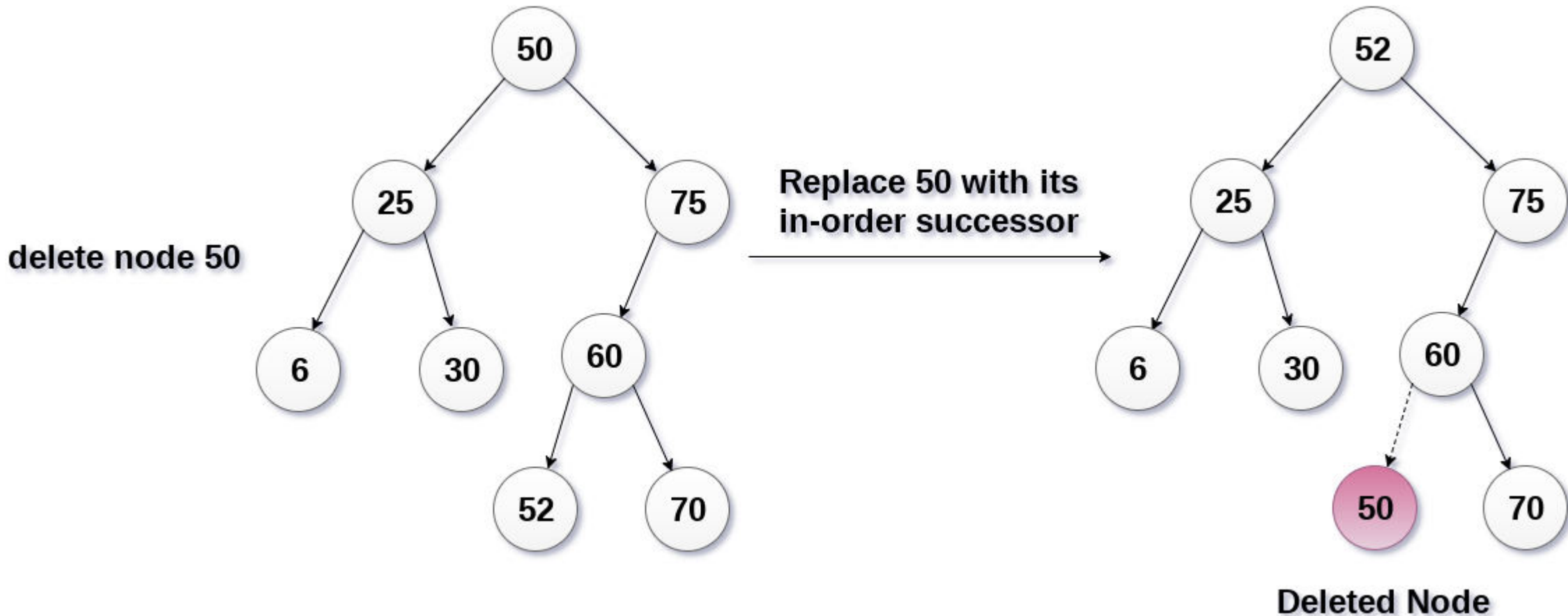
- **Delete operation in BST**

The node to be deleted has only one child: In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.



- **Delete operation in BST**

The node to be deleted has two children: It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.



- **Delete operation in BST**

You need to delete the node with value item and then return the root of the modified tree. First, we need to find the node to be deleted and then replace it by the appropriate node if needed.

Check if the root is NULL, if it is, just return the root itself. It's an empty tree!

If $\text{root.val} < \text{item}$, recurse the right subtree.

If $\text{root.val} > \text{item}$, recurse the left subtree.

If both above conditions above false, this means $\text{root.val} == \text{item}$.

Now we first need to check how many child did root have.

CASE 1: No Child \rightarrow Just delete root or deallocate space occupied by it

CASE 2: One Child \rightarrow Replace root by its child

CASE 3: Two Children

Find the inorder successor of the root (Its the smallest element of its right subtree). Let's call it new_root.

Replace root by its inorder successor

Now recurse to the right subtree and delete new_root.

9. Return the root.

```

TreeNode delete_element(TreeNode root, int item)
{
    if ( root == NULL )
        return root
    if ( root.val < item)
        root.right = delete_element(root.right, item)
    else if ( root.val > item )
        root.left = delete_element(root.left, item)
    else if ( root.val == item )
    { // No child
        if ( root.left == NULL and root.right == NULL )
            root=NULL; delete root
        // Only one child
        else if ( root.left == NULL or root.right == NULL )
        {
            if(root.left != NULL)
            {
                TreeNode temp = root
                root = root.left
                delete temp
            }

            else
            {
                TreeNode temp= root
                root = root.right
                delete temp
            }
        }
    }
}

```

```

// Both left and right child exist
    else
    {
        TreeNode new_root = find_inorder_successor(root.right)
        root.val = new_root.val
        root.right = delete_element(root.right, new_root.val)
    }
}
return root
}

TreeNode find_inorder_successor(TreeNode root)
{
    TreeNode curr = root
    while ( curr.left != NULL )
        curr = curr.left

    return curr
}

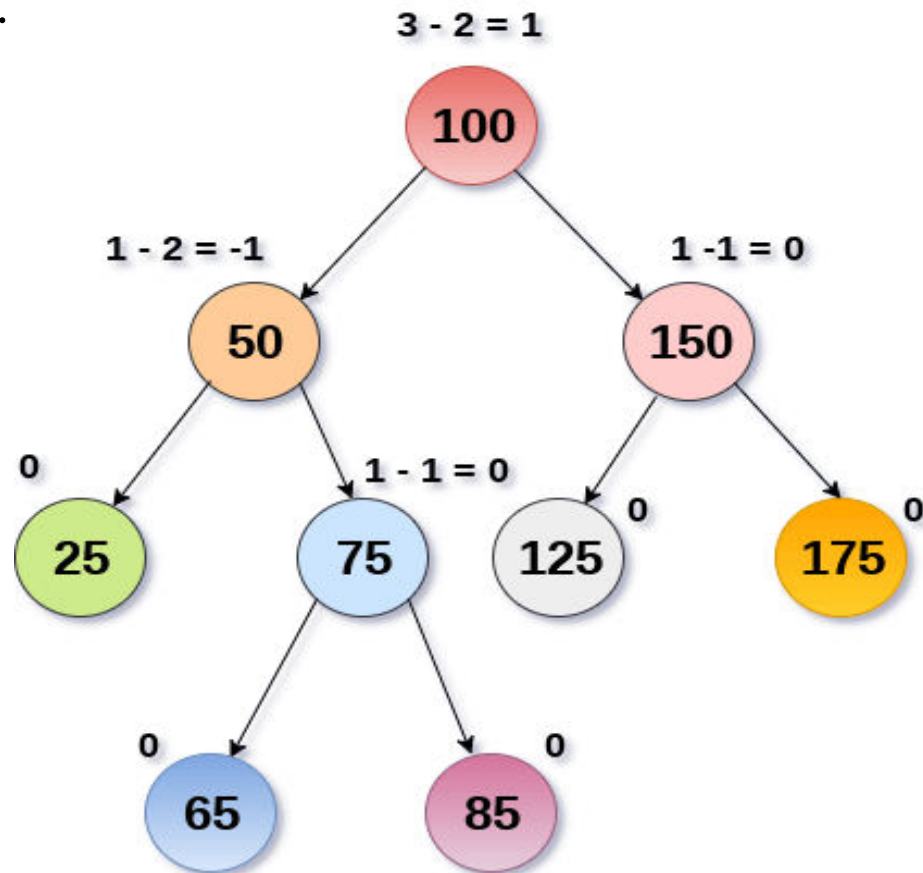
```

AVL Tree

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honor 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) = $\text{height}(\text{left}(k)) - \text{height}(\text{right}(k))$
- 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.

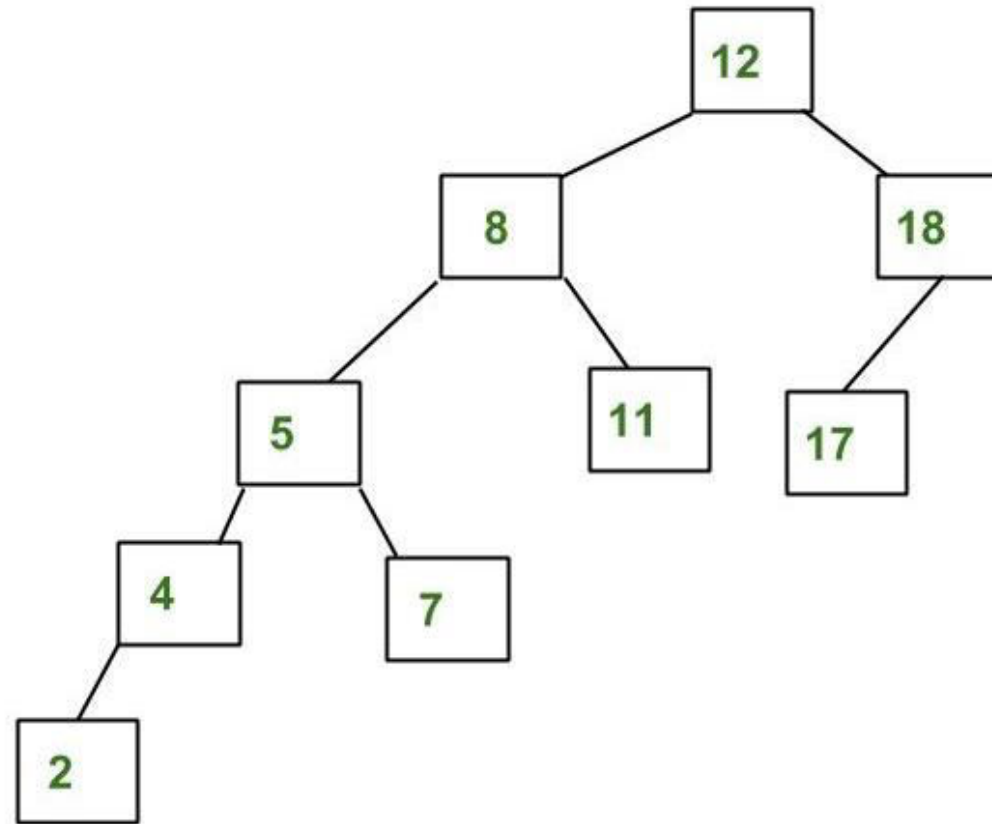
AVL Tree

- **Definition:** An empty binary tree is an AVL tree. A non empty binary tree T is an AVL tree iff given $T(L)$ and $T(R)$ to be the left and right subtrees of T and $h(T(L))$ and $h(T(R))$ to be the heights of subtrees $T(L)$ and $T(R)$ respectively, $T(L)$ and $T(R)$ are AVL trees and $|h(T(L)) - h(T(R))| \leq 1$.
- $h(T(L)) - h(T(R))$ is known as the balance factor (BF) and for an AVL tree the balance factor of a node can be either 0, 1, or -1.



AVL Tree

Not an 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.

Operations on AVL tree

- 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.
- 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.
- 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.

AVL Rotations

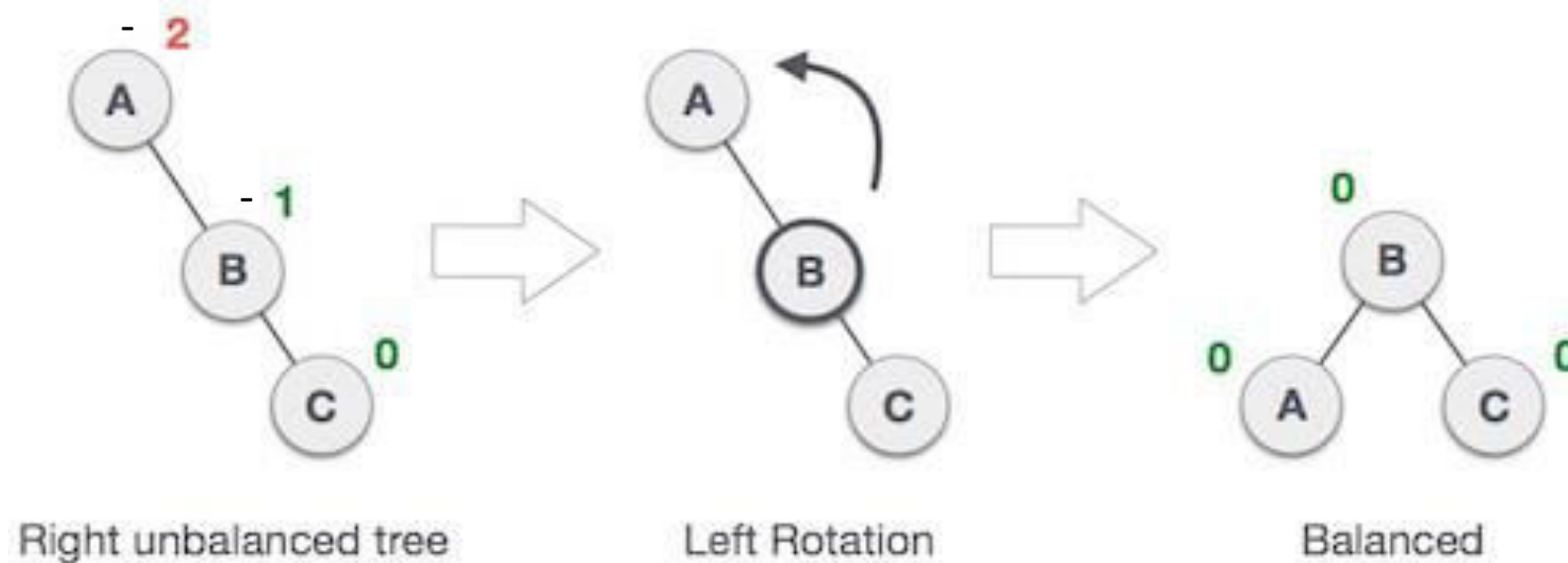
- We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

1. LL rotation: Inserted node is in the left subtree of left subtree of A
2. RR rotation : Inserted node is in the right subtree of right subtree of A
3. LR rotation : Inserted node is in the right subtree of left subtree of A
4. RL 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.

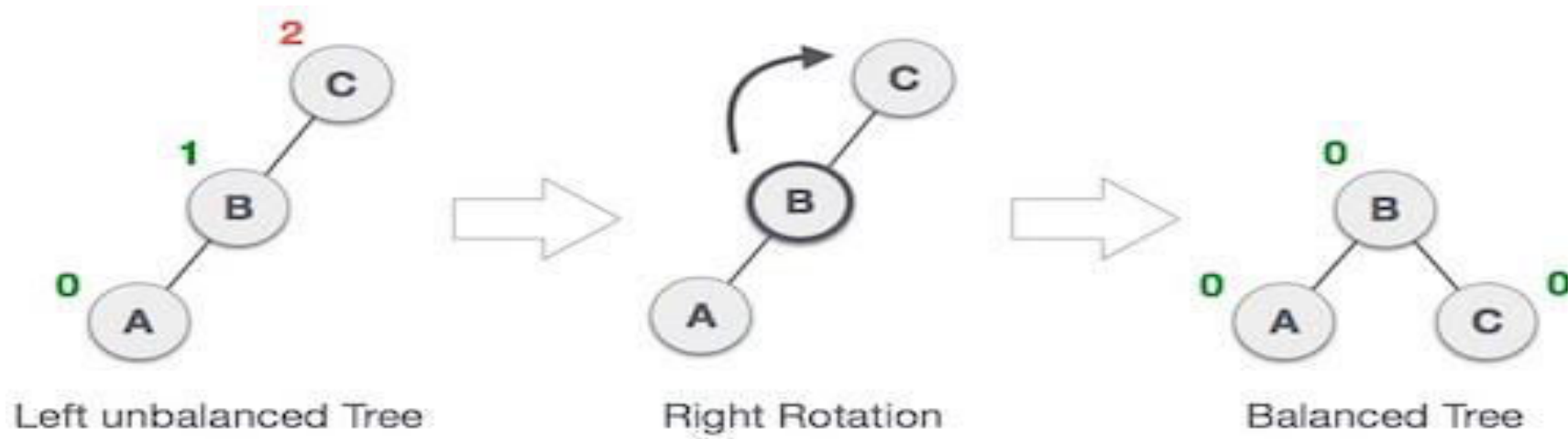
The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2,

- 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



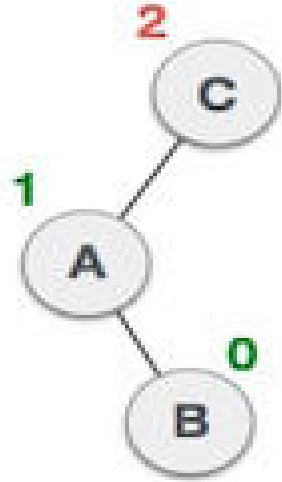
- In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

- 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.

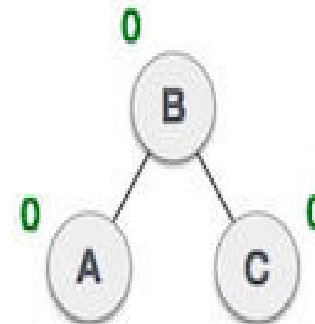
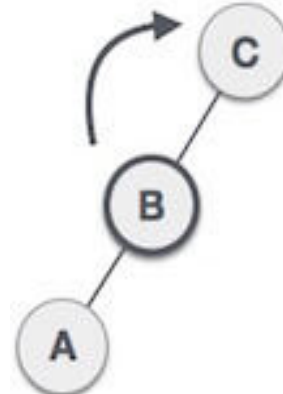
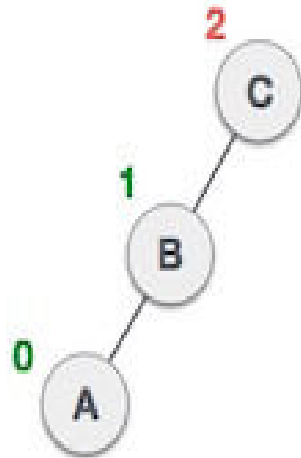
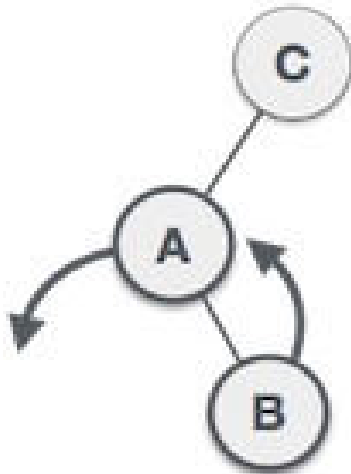
- LR Rotation
- Double rotations are bit tougher than single rotation. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other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



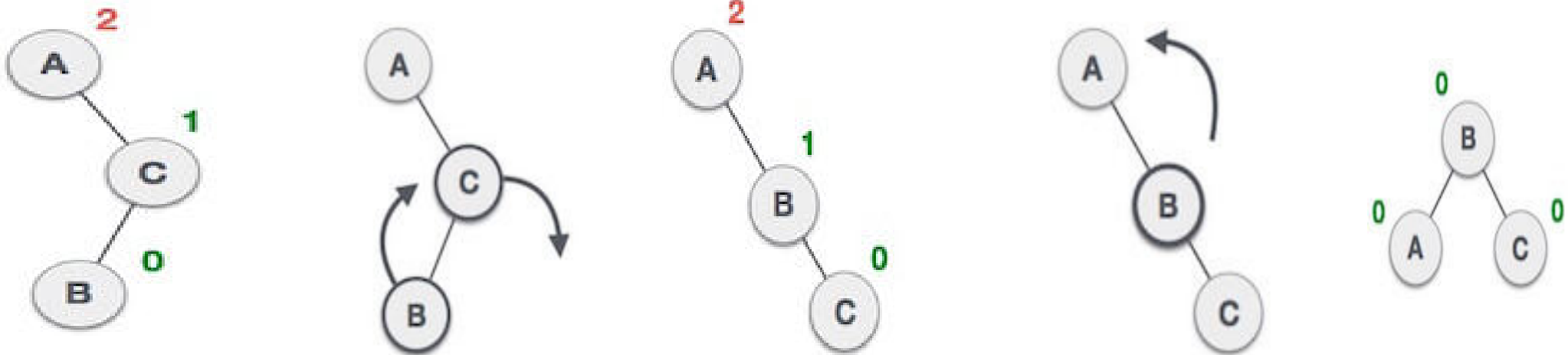
A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C.

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.

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



- R L rotation
- 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.



- Delete Operation in AVL tree:

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations. The two types of rotations are L rotation and R rotation.

If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

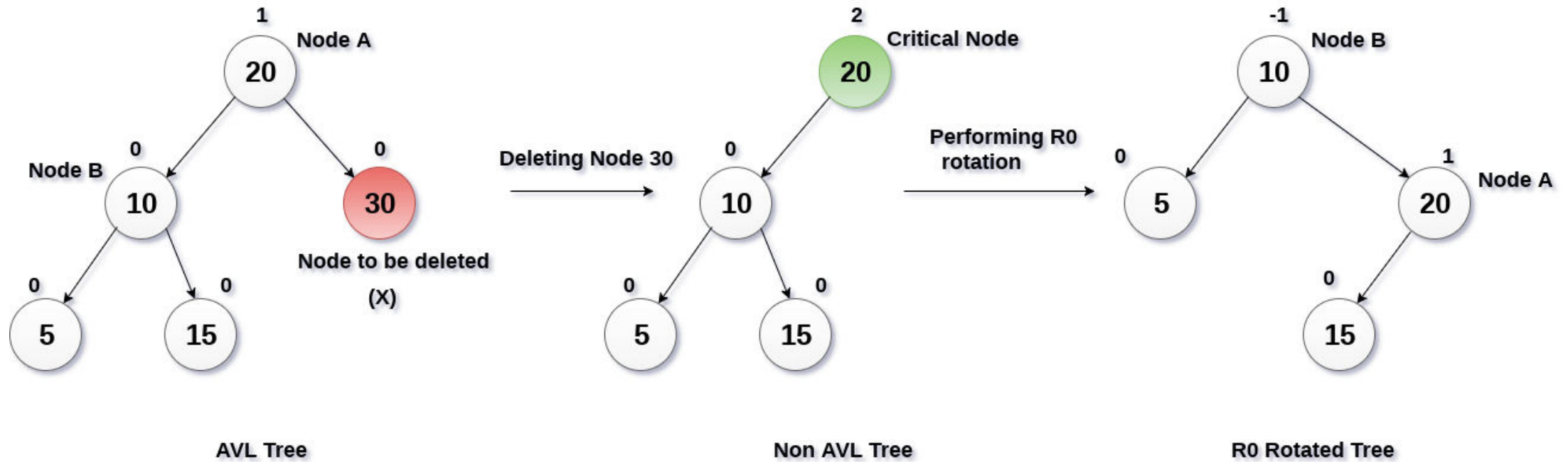
Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three different situations:

R0 rotation

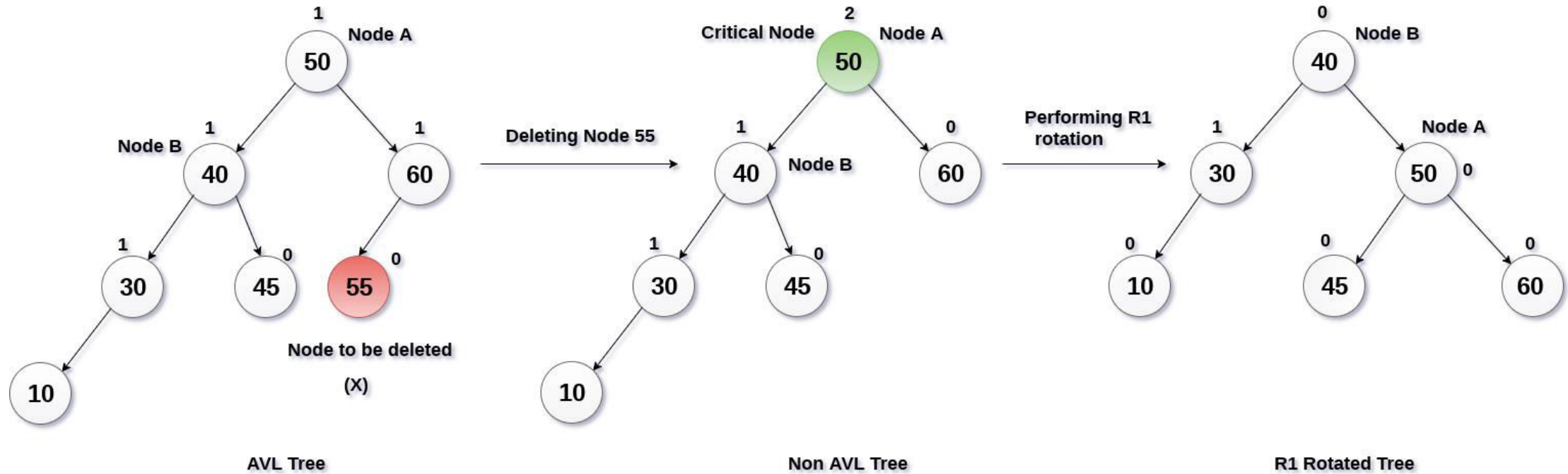
R1 Rotation

R-1 Rotation

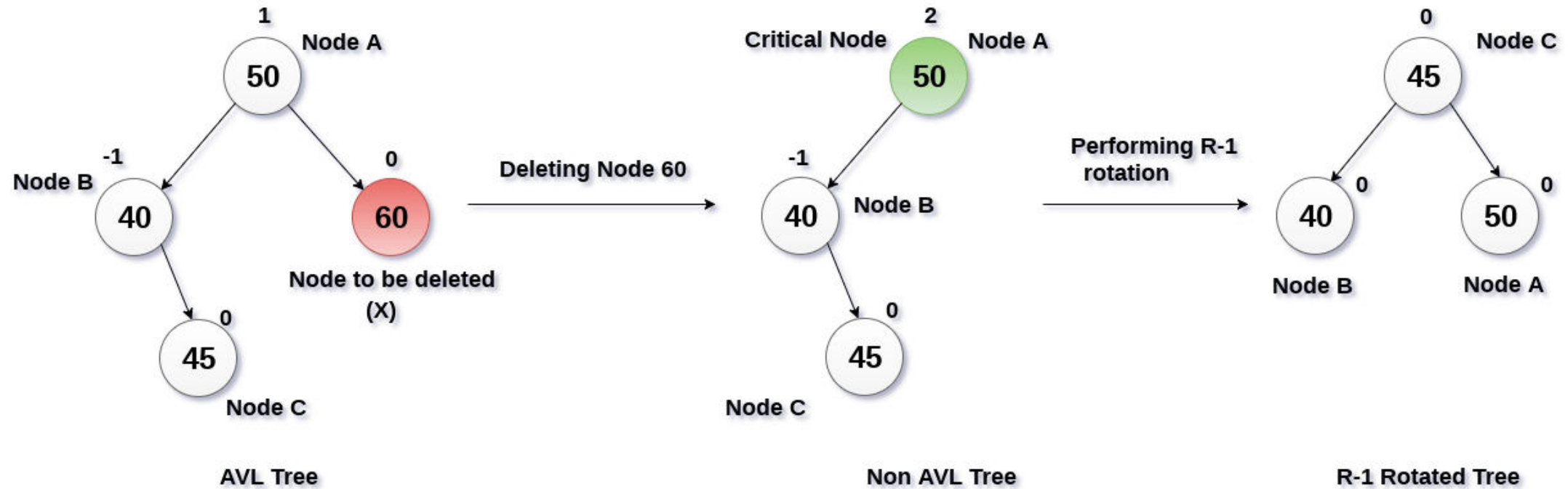
R0 Rotation



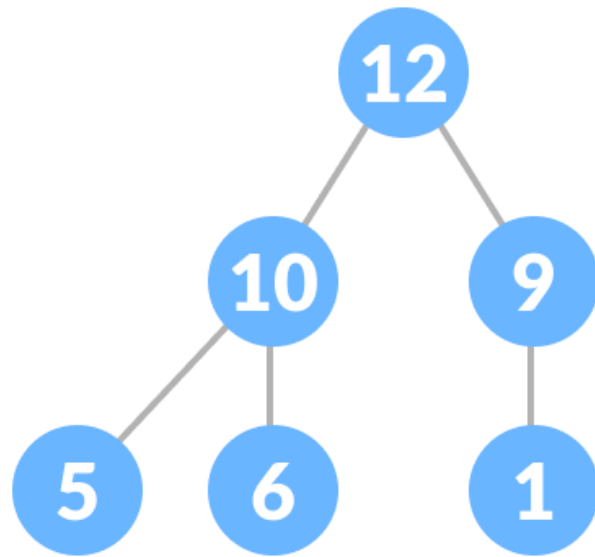
R1 Rotation



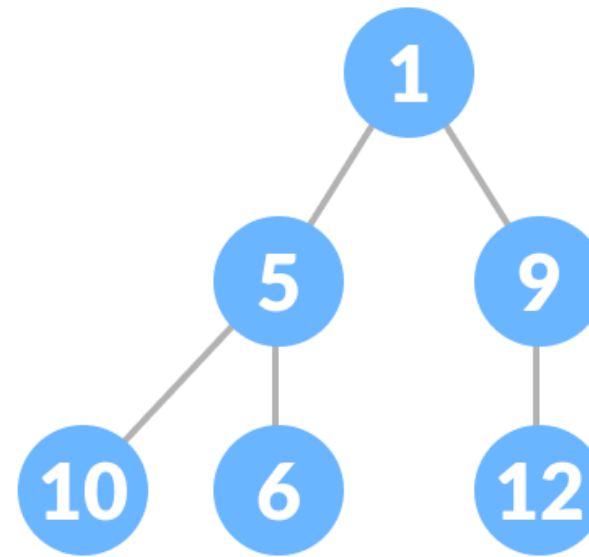
R-1 Rotation



- **Heaps and it's operations**
- Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if
- it is a complete binary tree
- All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children are smaller than the root and so on. Such a heap is called a max-heap. If instead, all parent nodes are smaller than their children, it is called a min-heap



Max Heap



Min Heap

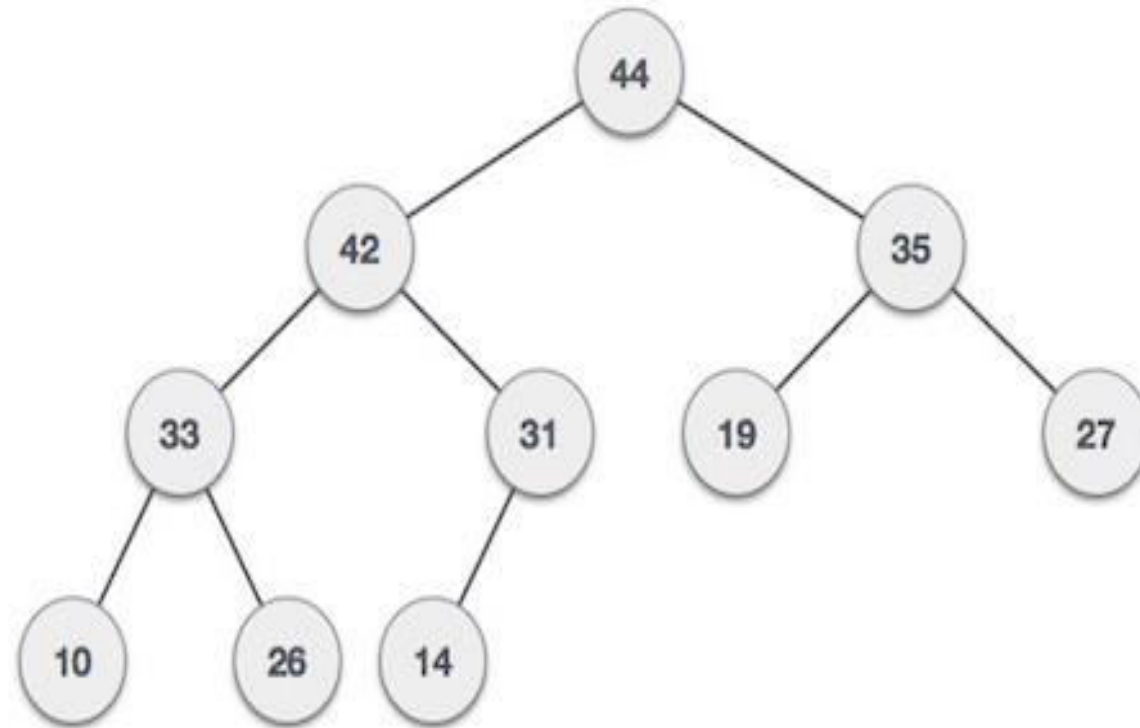
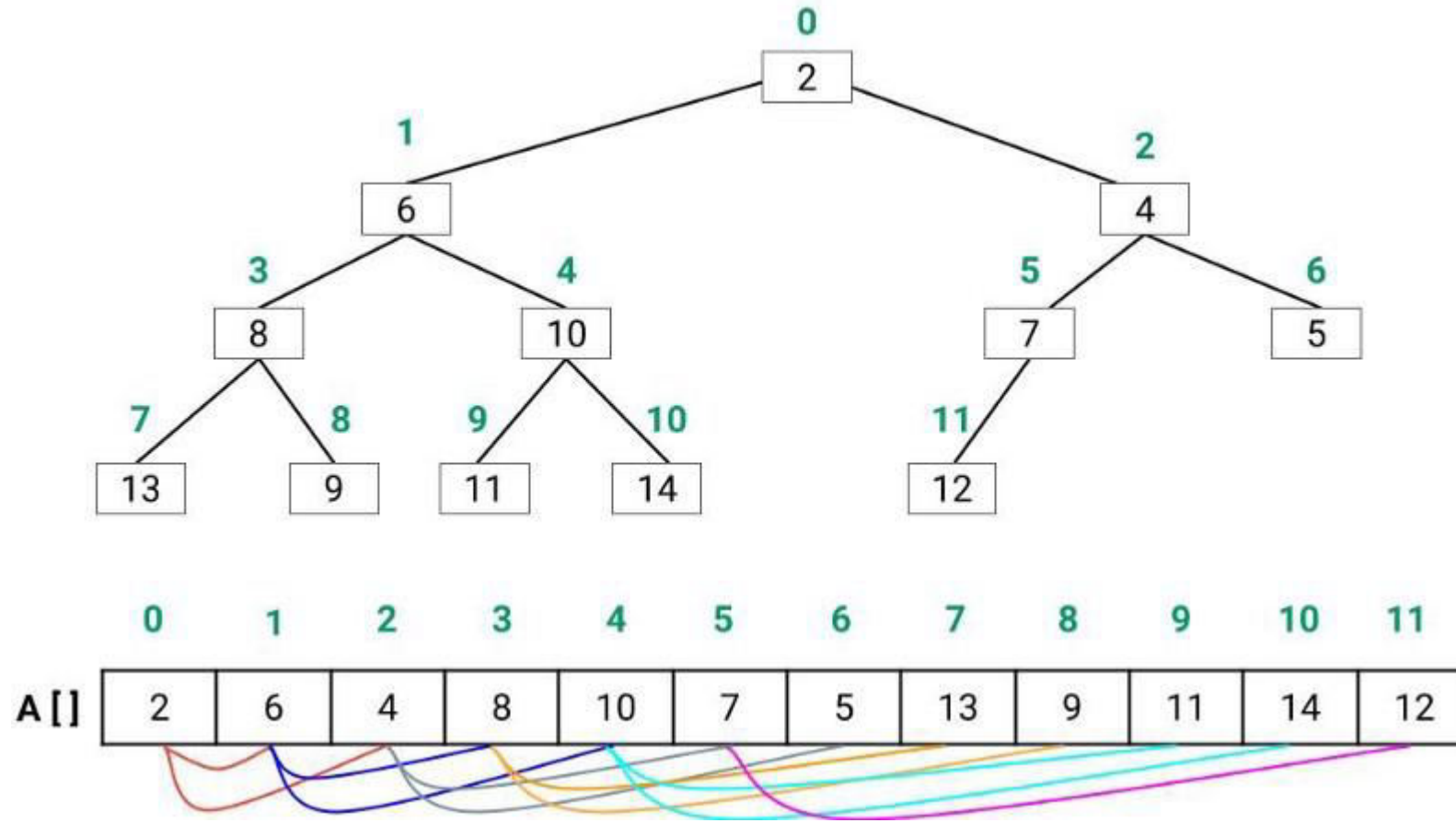


Fig: Max Heap

Array implementation of Heap

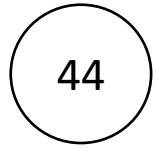
- A binary heap can be represented using an array where the indices of the array capture the parent-child relationship. Suppose $A[]$ be a heap array of size n :
- The root of the binary heap is stored at $A[0]$.
- Given element $A[i]$, the children of this element are stored in $A[2*i + 1]$ and $A[2*i + 2]$, if they exist.
- The left child of i denoted as $\text{left}(i) = A[2*i + 1]$, if $2i + 1 < n$
- The right child of i denoted as $\text{right}(i) = A[2*i + 2]$, if $2i + 2 < n$
- The parent of $A[i]$ is stored in $A[(i-1)/2]$.

Array implementation of Heap

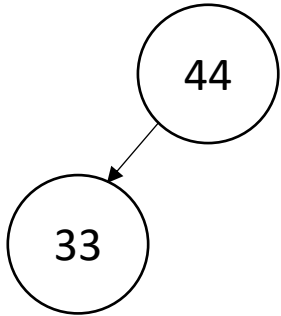


• **Create Heap Tree : 44, 33, 77, 11, 55, 88, 66**

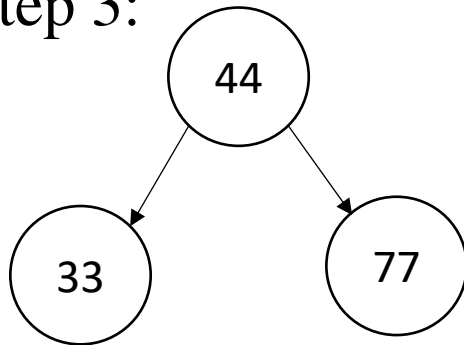
• Step 1:



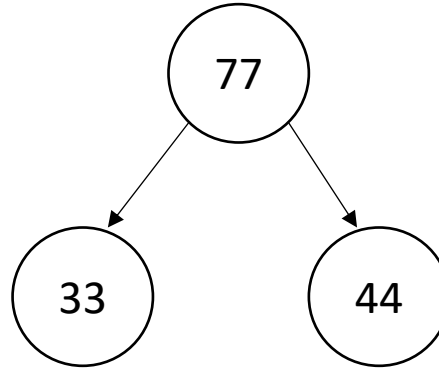
• Step 2:



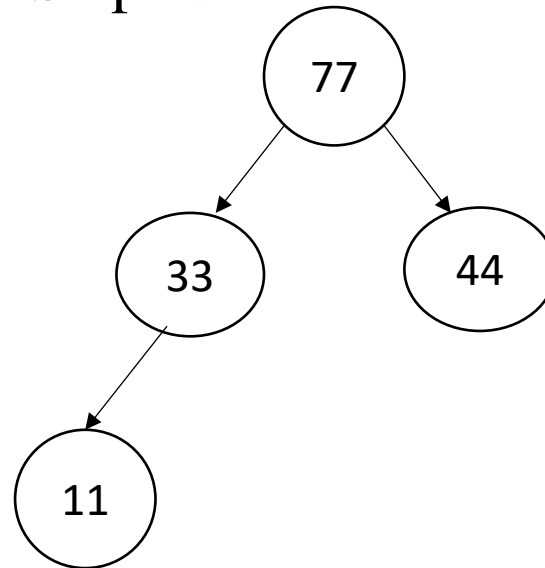
Step 3:



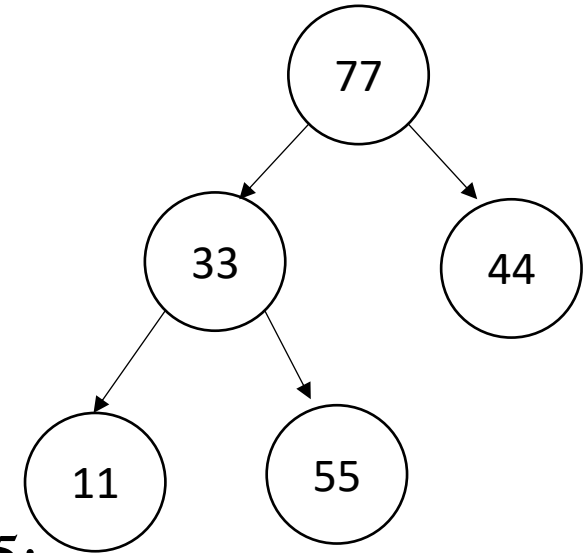
Step 3:



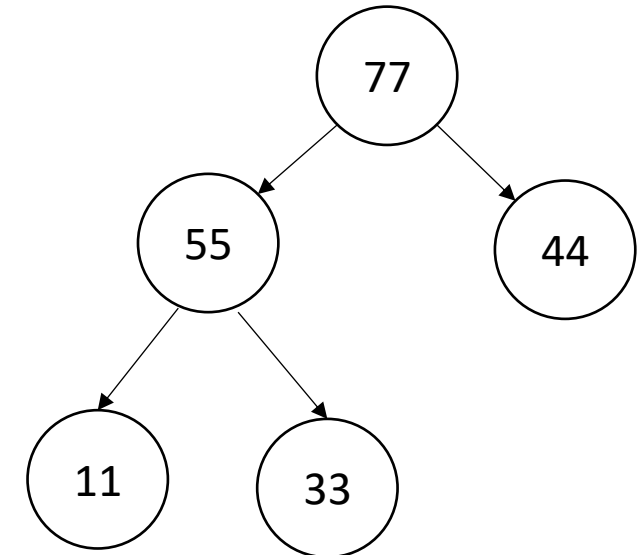
Step 4:



Step 5:

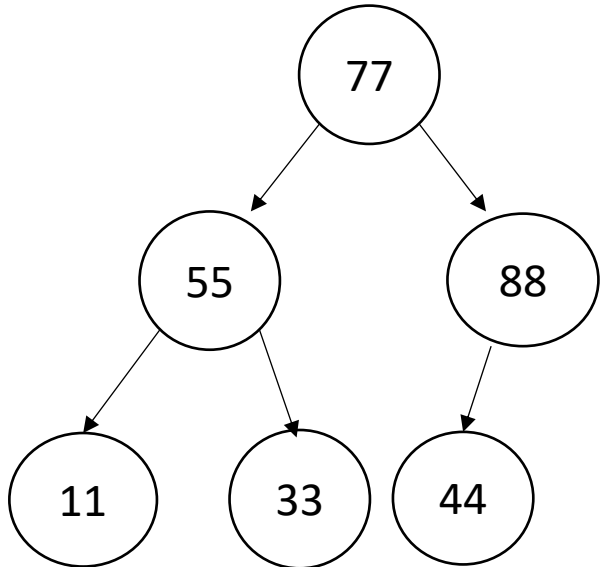
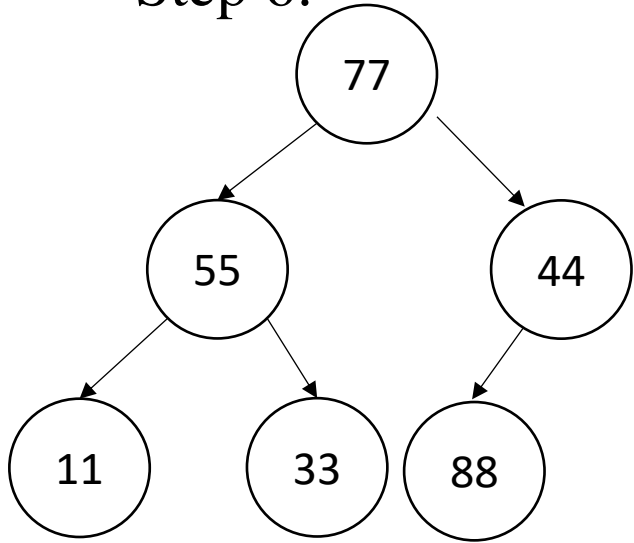


Step 5:

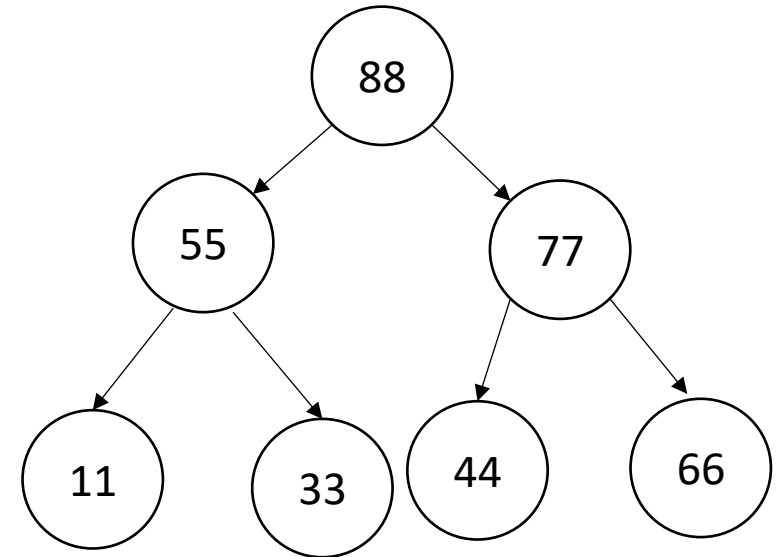
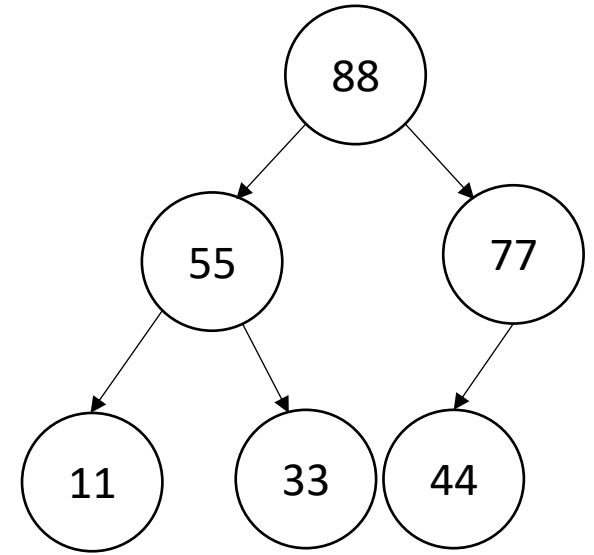


- **44, 33, 77, 11, 55, 88, 66**

- Step 6:



Step 7:



We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

- Step 1 – Create a new node at the end of heap.
- Step 2 – Assign new value to the node.
- Step 3 – Compare the value of this child node with its parent.
- Step 4 – If value of parent is less than child, then swap them.
- Step 5 – Repeat step 3 & 4 until Heap property holds.

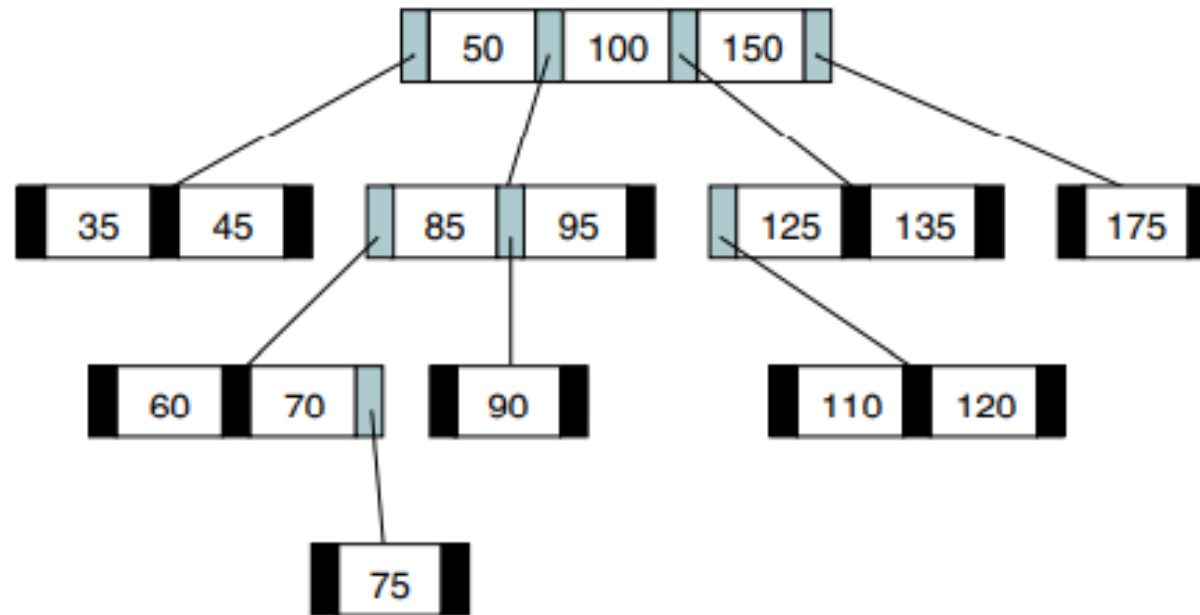
We derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

- Step 1 – Remove root node.
- Step 2 – Move the last element of last level to root.
- Step 3 – Compare the value of this child node with its parent.
- Step 4 – If value of parent is less than child, then swap them.
- Step 5 – Repeat step 3 & 4 until Heap property holds.

Introduction to Multiway Tree

A multiway tree is defined as a tree that can have more than two children. If a multiway tree can have maximum m children, then this tree is called as multiway tree of order m (or an m -way tree).

As with the other trees that have been studied, the nodes in an m -way tree will be made up of $m-1$ key fields and pointers to children.



Four-way Tree

An m-way tree is a search tree in which each node can have from 0 to m subtrees, where m is defined as the B-tree order. Given a nonempty multiway tree, we can identify the following properties:

1. Each node has 0 to m subtrees.
2. A node with $k < m$ subtrees contains k subtrees and $k - 1$ data entries.
3. The key values in the first subtree are all less than the key value in the first entry; the key values in the other subtrees are all greater than or equal to the key value in their parent entry.
4. The keys of the data entries are ordered $key_1 \leq key_2 \leq \dots \leq key_k$.
5. All subtrees are themselves multiway trees.

By definition an m-way search tree is a m-way tree in which following condition should be satisfied :

- Each node is associated with m children and m-1 key fields
- The keys in each node are arranged in ascending order.
- The keys in the first j children are less than the j-th key.
- The keys in the last m-j children are higher than the j-th key.

Graphs

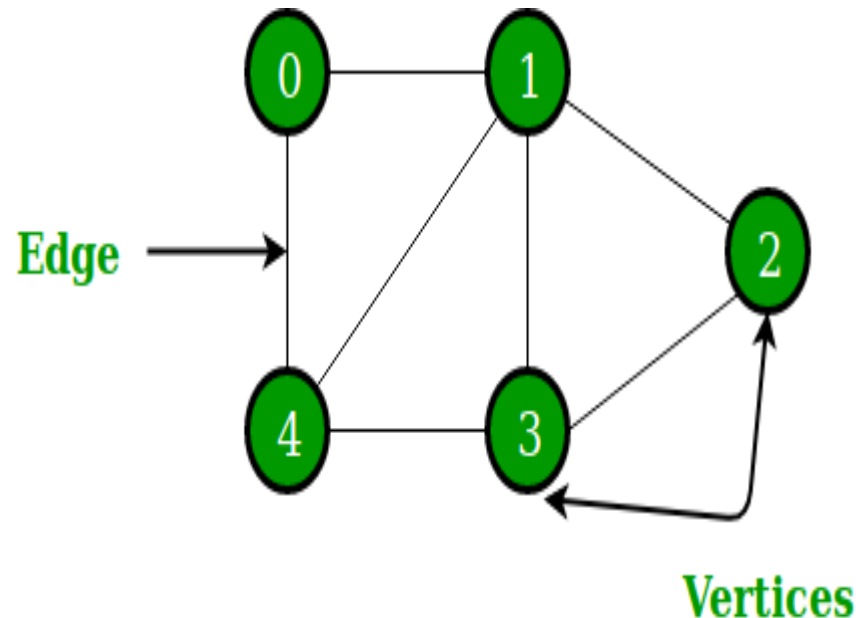
Basic Terminology

A graph is a nonlinear data structure. It is a collection of nodes that have data and are connected to other nodes.

A graph G consists of two things:

1. A set V of elements called nodes (or points or vertices)
2. A set E of edges such that each edge e in E is identified with a unique (unordered) pair $[u, v]$ of nodes in V , denoted by $e = [u, v]$

In the above Graph, the set of vertices $V = \{0,1,2,3,4\}$ and the set of edges $E = \{01, 12, 23, 34, 04, 14, 13\}$.



Suppose $e = (u, v)$ then the nodes u and v are called the endpoints of e . u and v are said to be adjacent nodes or neighbors.

- The degree of node u , written as $\deg(u)$, is the number of edges containing u . if $\deg(u) = 0$ that is, if u does not belong to any edge, then u is called an isolated node.
- A path p of length n from a node u to a node v is defined as a sequence of $n + 1$ nodes.

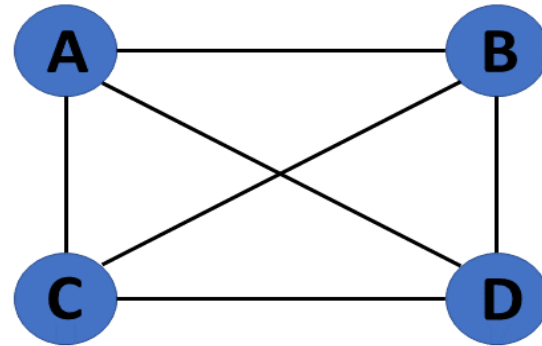
$$P = (v_0, v_1, v_2, \dots, v_n)$$

Such that $u = v_0$, v_{i-1} is adjacent to v_i for $i = 1, 2, \dots, n$, and $v_n = v$.

- The path P is said to be closed if $v_0 = v_n$.
- The path P is said to be simple if all the nodes are distinct, with the exception that v_0 may equal to v_n .
A cycle is a closed simple path with length 3 or more.
- Connected Graph: A connected graph is the one in which some path exists between every two vertices (u, v) in V . There are no isolated nodes in connected graph.
- A connected graph without any cycles is called tree graph or free tree or simply tree.

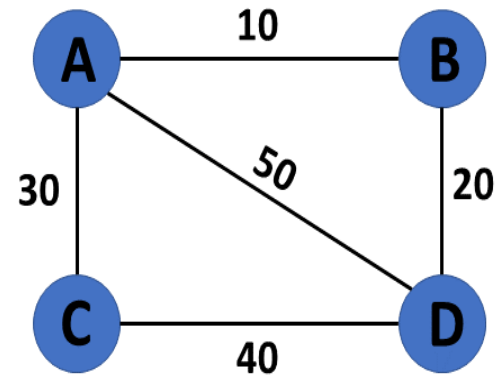
- Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contains $\frac{n(n-1)}{2}$ edges where n is the number of nodes in the graph.

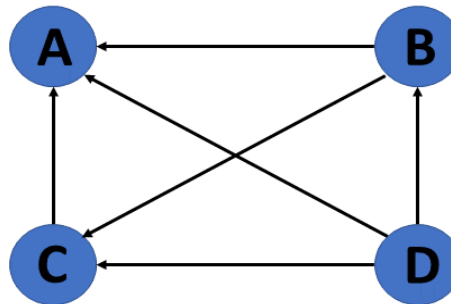


- Weighted Graph

A graph $G = (V, E)$ is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



- Multiple edges: distinct edges e and e' are called multiple edges if they connect the same endpoints, that is, if $e = [u, v]$ and $e' = [v, u]$.
- Loops: an edge e is called a loop if it has identical endpoints, that is, if $e = [u, u]$.
- Multi Graph: If there are numerous edges between a pair of vertices in a graph $G = (V, E)$, the graph is referred to as a multigraph.
- Directed Graph: A directed graph also referred to as a digraph, is a set of nodes connected by edges, each with a direction, in other words, each edge e is identified with an ordered pair (u, v) of nodes in G rather than an unordered pair $[u, v]$.



Suppose G is directed graph with a directed edge $e = (u, v)$. Then e is also called an arc, moreover following terminology is used.

- e begins at u and ends at v .
- u is the origin or initial point of e , and v is the destination or terminal point of e .
- u is a predecessor of v , and v is a successor or neighbor of u .
- u is adjacent to v , v is adjacent to u .
- The outdegree of a node u in G , written $\text{outdeg}(u)$, is the number of edges beginning at u , similarly, the indegree of u , written $\text{indeg}(u)$, is the number of edges ending at u .
- A node u is called source if it has positive outdegree but zero indegree. Similarly, u is called a sink if it has a zero outdegree but a positive indegree.
- A directed graph is said to be connected, or strongly connected, if for each pair u, v of nodes in G there is a path from u to v and there is also a path from v to u . Graph G is said to be unilaterally connected if for any pair u, v of nodes in G there is a path from u to v or a path from v to u .

There are two standard ways of maintaining a graph G in memory of a computer. One way called the sequential representation of G , is by means of its Adjacency Matrix A . the other way called the linked representation of G , is by means of linked list of neighbors.

1. Adjacency Matrix

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[][]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .

Pros: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done $O(1)$.
Cons: Consumes more space $O(V^2)$. Even if the graph is sparse(contains less number of edges), it consumes the same space.

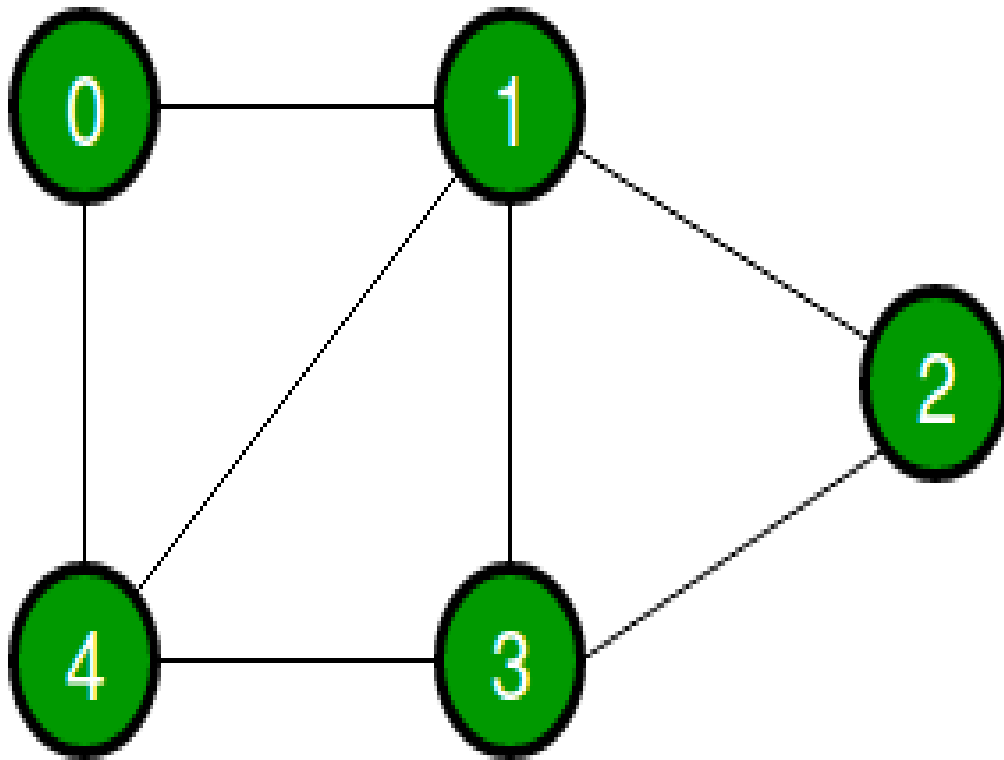
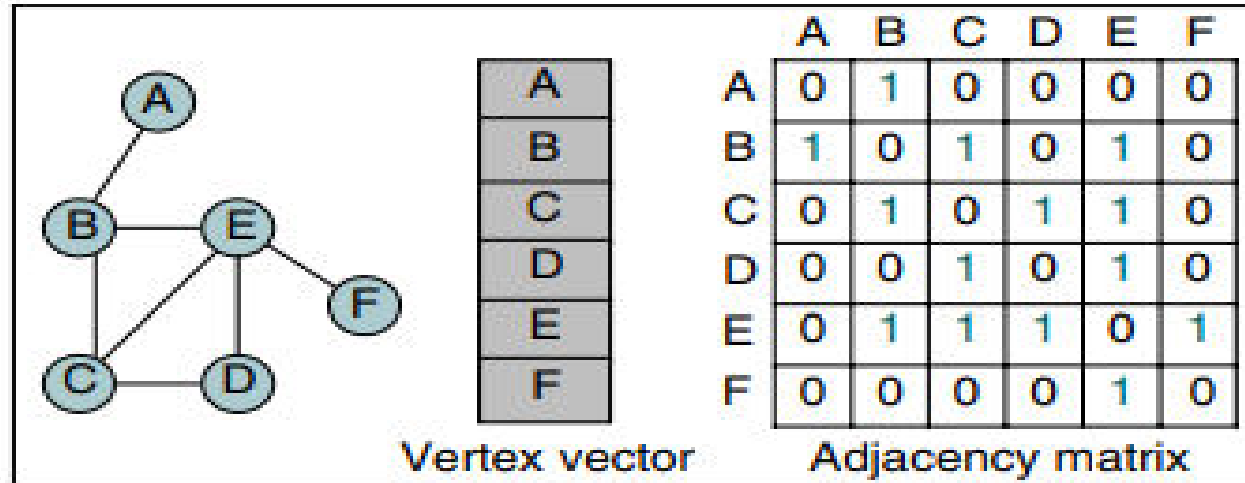


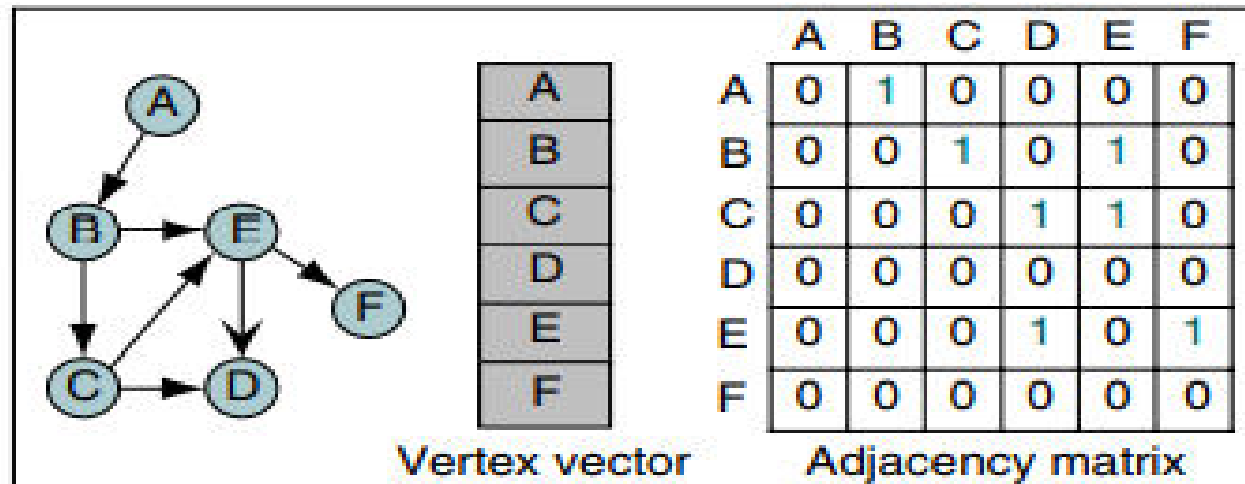
Fig: Graph

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Fig: Adjacency matrix



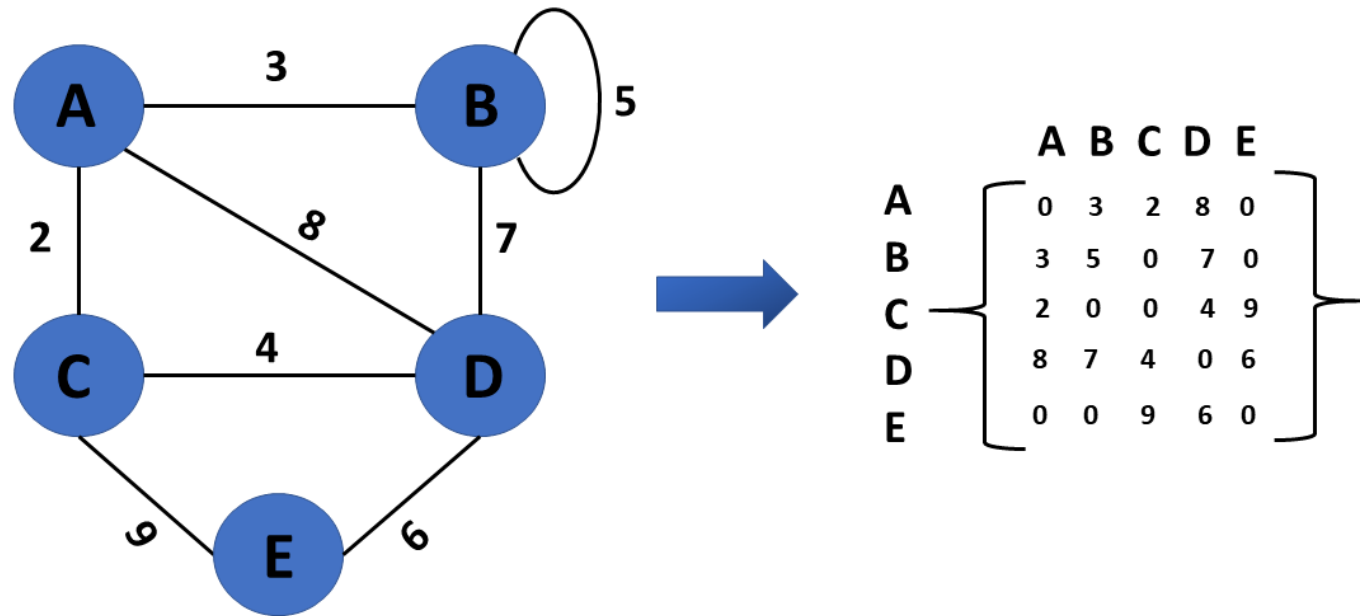
(a) Adjacency matrix for nondirected graph



(b) Adjacency matrix for directed graph

- Weighted Undirected Graph Representation

Weight or cost is indicated at the graph's edge, a weighted graph representing these values in the matrix.



Adjacency List:

- An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list.
- The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.
- In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.
- In the case of weighted directed graph, each node contains an extra field that is called the weight of the node.
- An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space. In adjacency list it is easy to add new nodes.

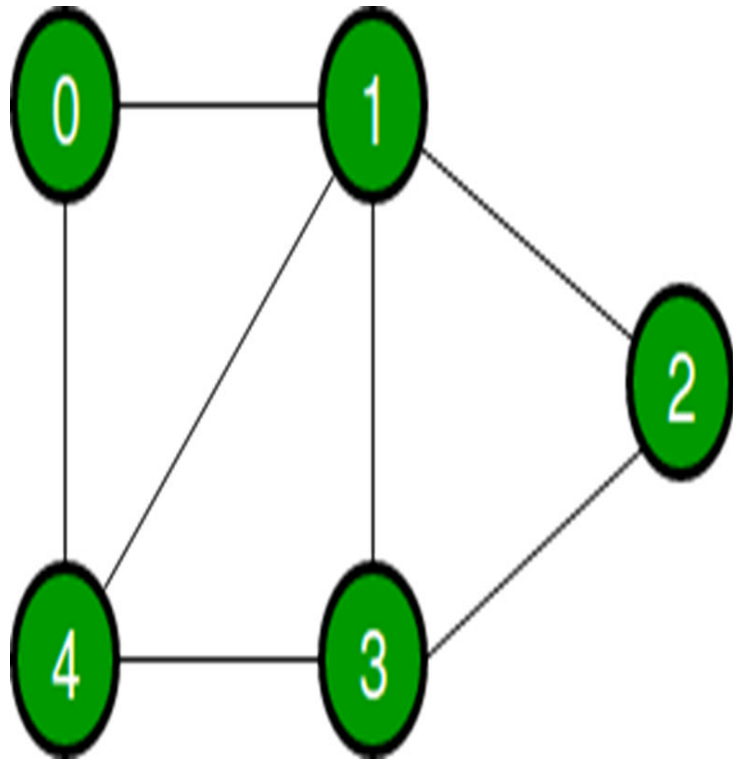


Fig: Graph

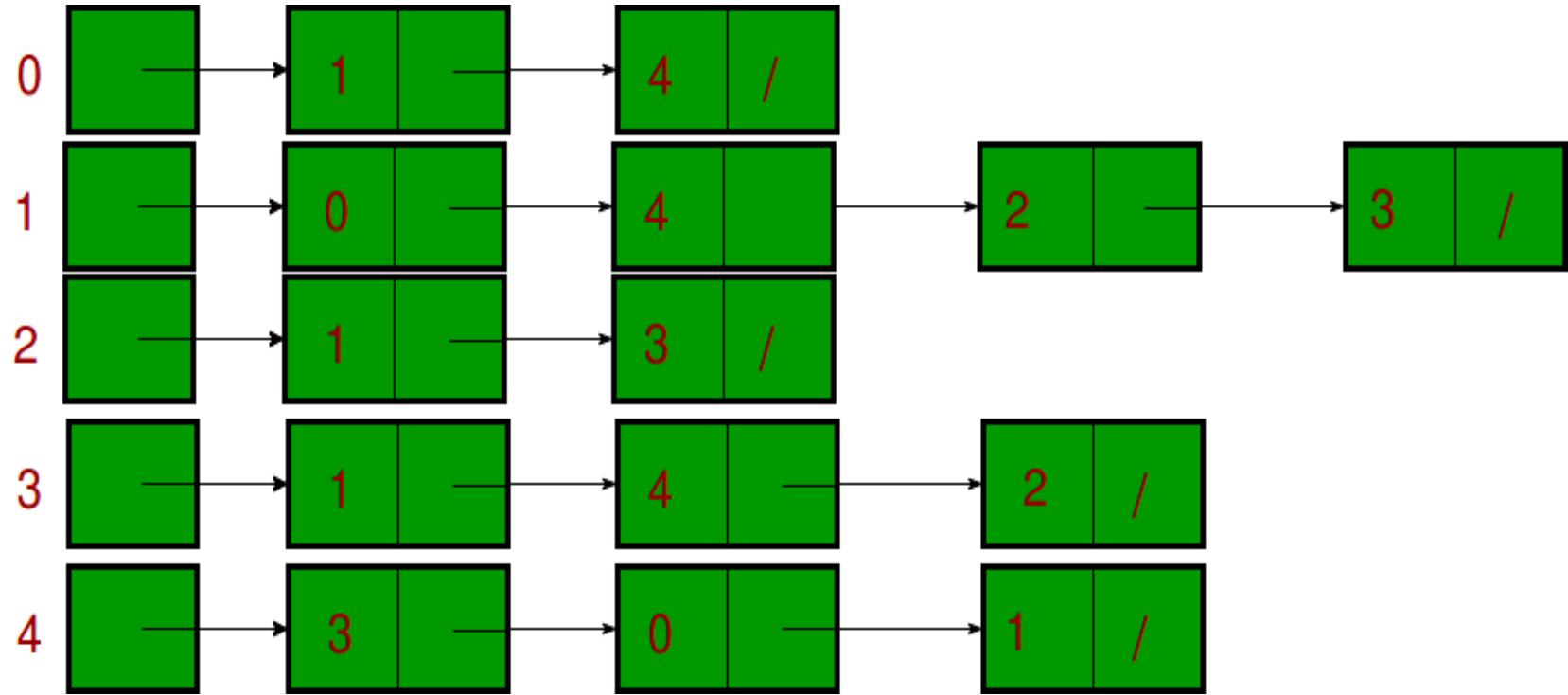
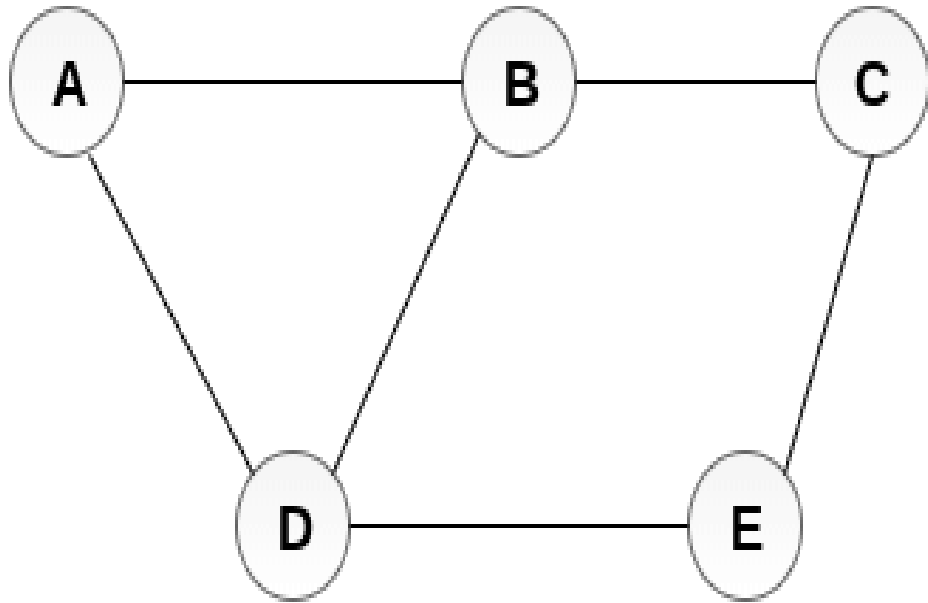
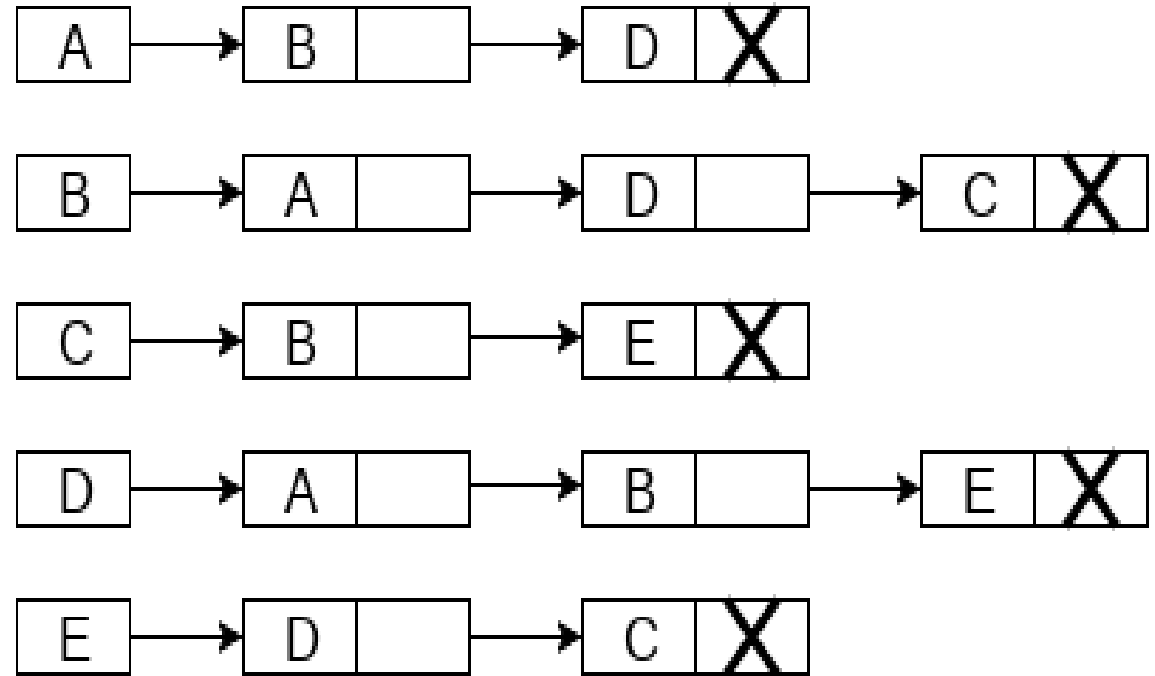


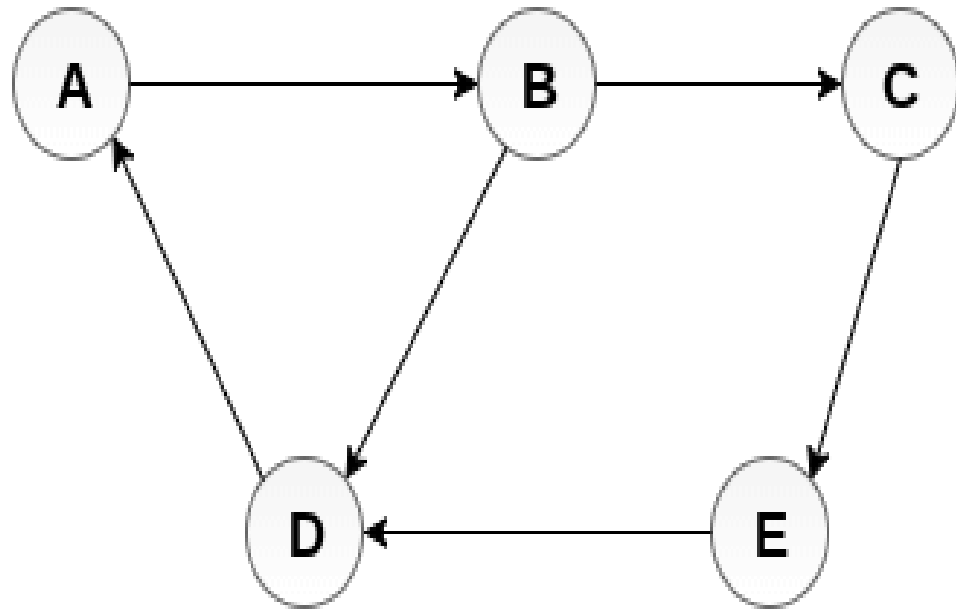
Fig: Adjacency List



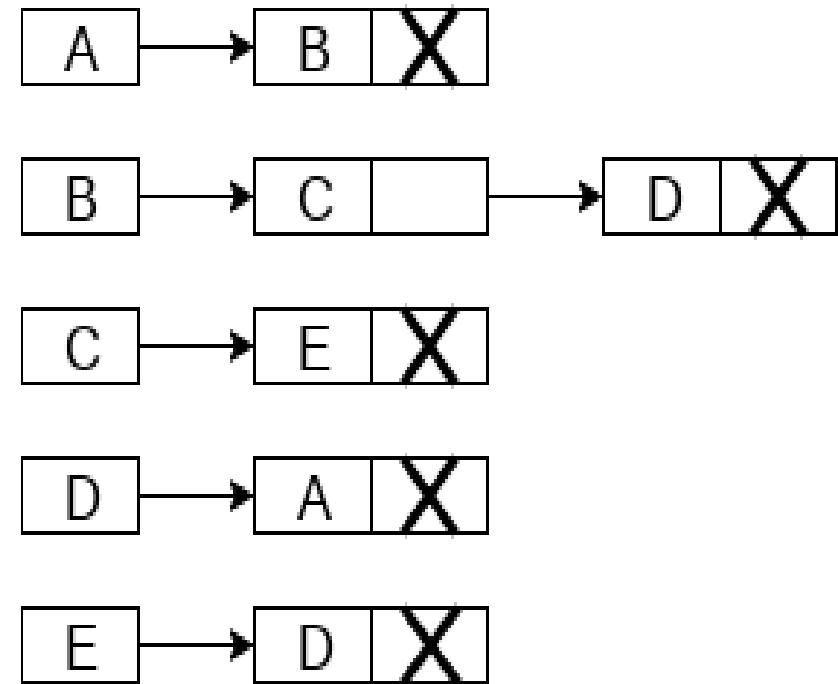
Undirected Graph



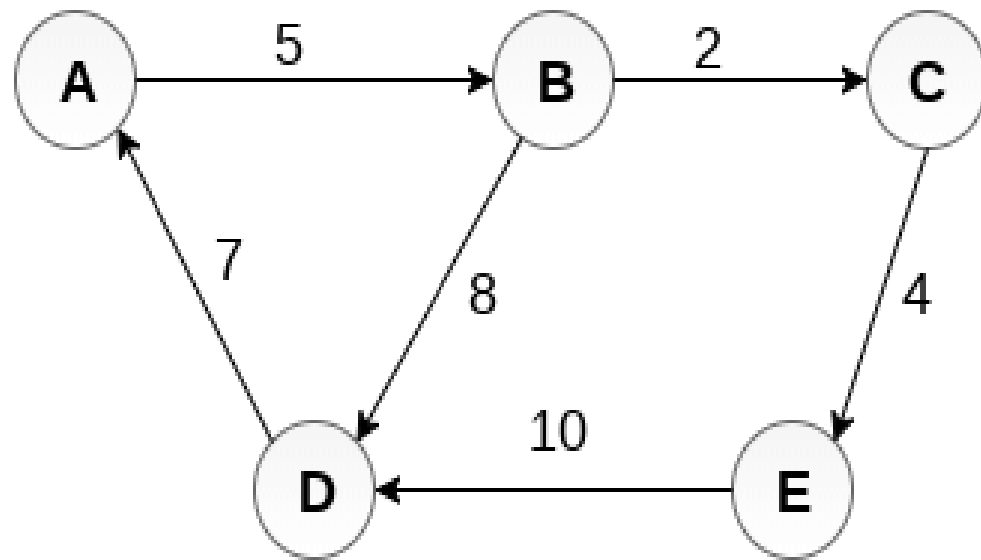
Adjacency List



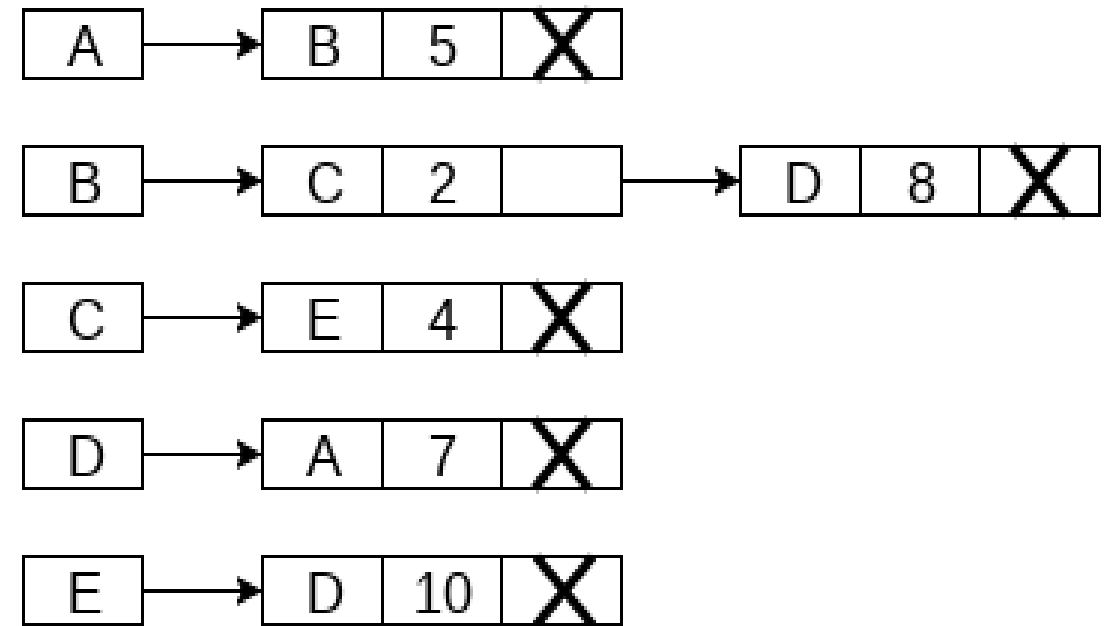
Directed Graph



Adjacency List



Weighted Directed Graph

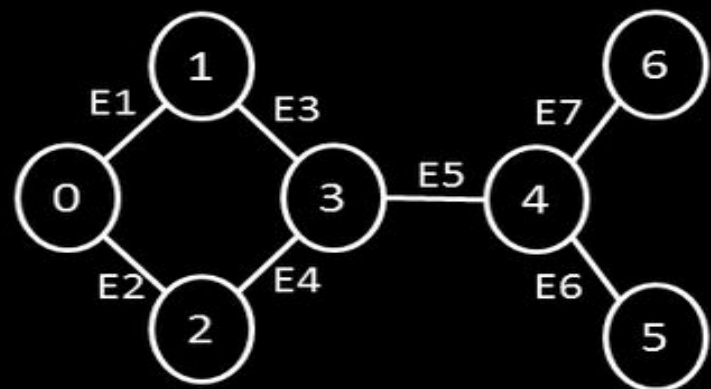


Adjacency List

Adjacency Multi-list representation

- Modified version of adjacency lists
- Edge based rather than vertex based representation of graph

Example...



Undirected Graph

Edge 1		0	1	Edge 2	Edge 3
Edge 2		0	2	NULL	Edge 4
Edge 3		1	3	NULL	Edge 4
Edge 4		2	3	NULL	Edge 5
Edge 5		3	4	NULL	Edge 6
Edge 6		4	5	Edge 7	NULL
Edge 7		4	6	NULL	NULL

Adjacency Multi-list

GRAPH TRAVERSAL

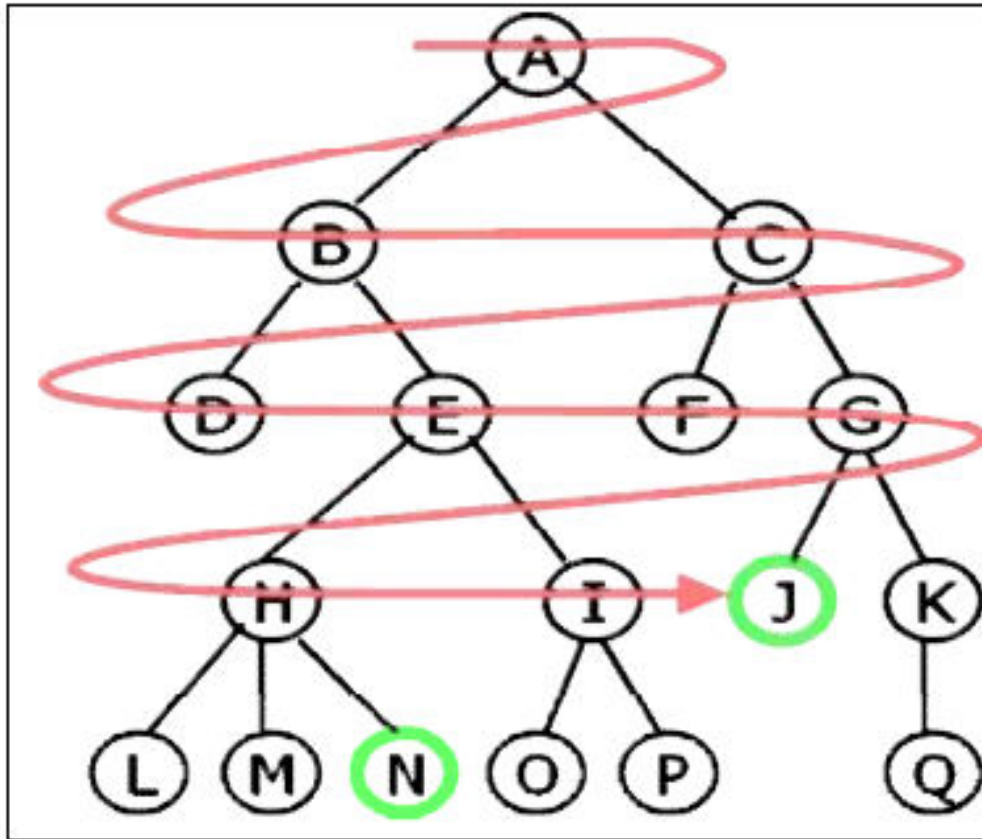
- Graph traversal is the problem of visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way. The order in which the vertices are visited may be important, and may depend upon the particular algorithm.

The two common traversals:

- breadth-first search
- depth-first search

- **Breadth-First Search**

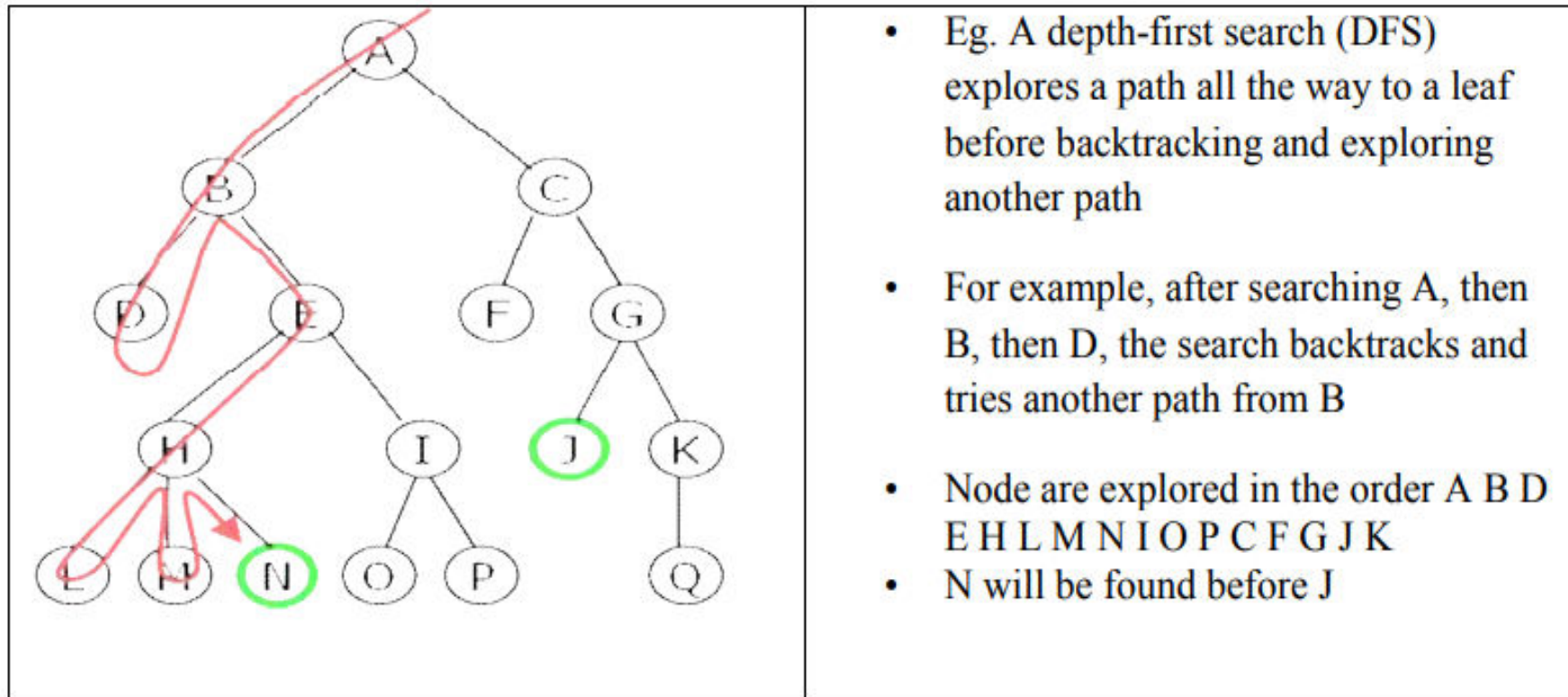
- In a breadth-first search, we begin by visiting the start vertex v . Next all unvisited vertices adjacent to v are visited. Unvisited vertices adjacent to these newly visited vertices are then visited and so on.



- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N

- **Depth-First Search**

- We begin by visiting the start vertex v . Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w . The search terminates when no unvisited vertex can be reached from any of the visited vertices.



- BFS:
- 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

- BFS:

- Algorithm

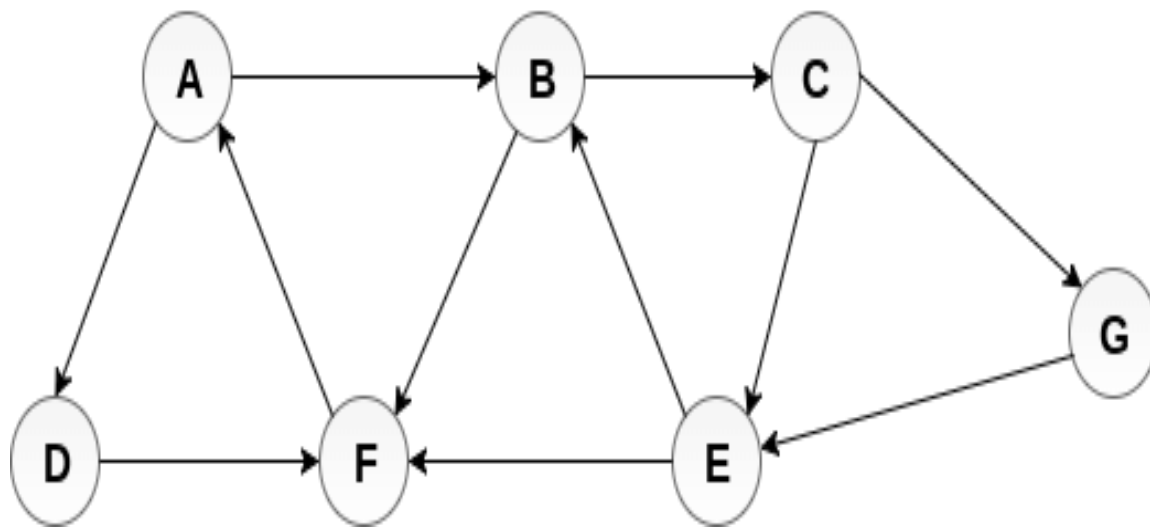
1. Put root in a Queue

2. Repeat until Queue is empty:

- Dequeue a node

- Process it

- Add it's children to queue(if not in queue or not yet visited)



Adjacency Lists

A : B, D

B : C, F

C : E, G

G : E

E : B, F

F : A

D : F

- 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.

QUEUE1 = {G} QUEUE2 = {A, B, D, C, F, E}

- 8. Remove G from queue, all of G's neighbours has already been processed we will not add it again. All the nodes are visited

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

- DFS:
- 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

- DFS:

- Algorithm

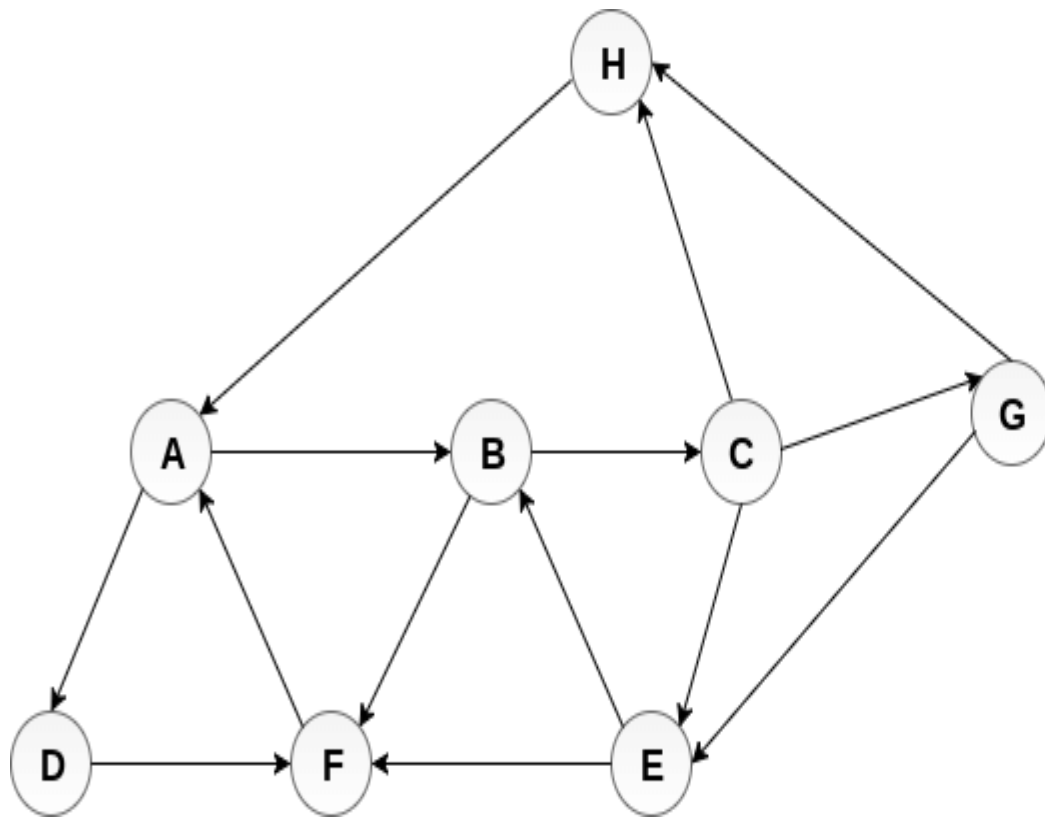
- Push the starting node A on the stack

- Repeat Steps until STACK is empty

- Pop the top node N.

- Process it

- Push on the stack all the neighbours of N(unvisited/ not in stack)



Adjacency Lists

A : B, D

B : C, F

C : E, G, H

G : E, H

E : B, F

F : A

D : F

H : A

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

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 :

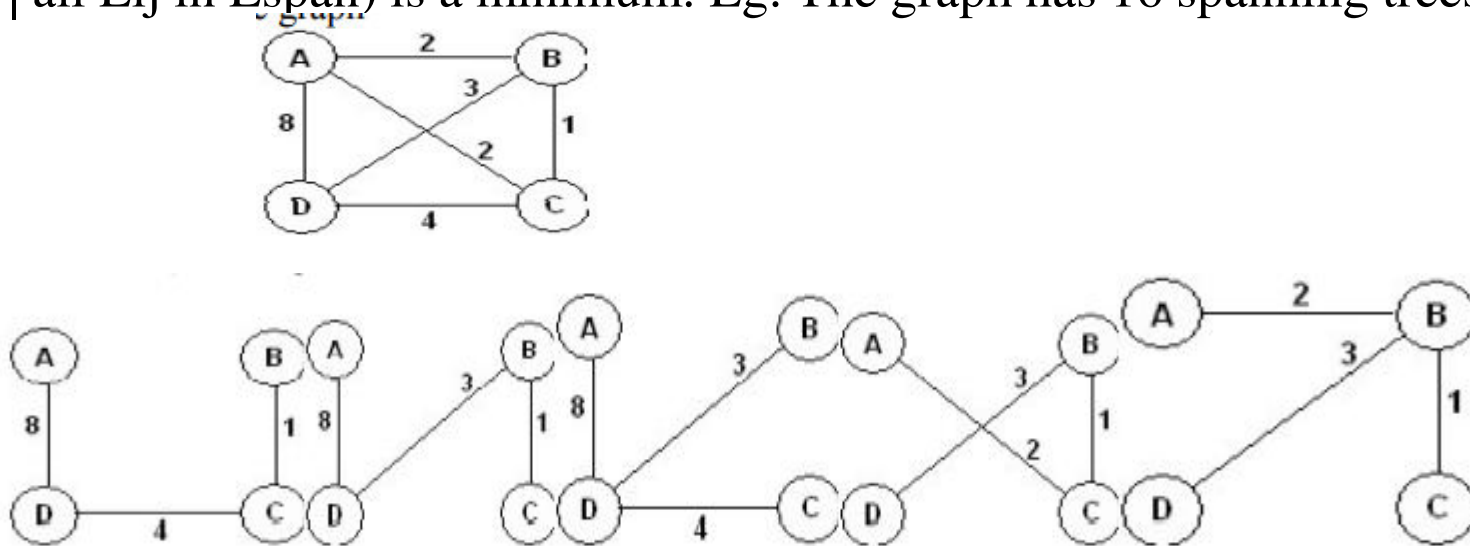
H -> A -> D -> F -> B -> C -> G -> E

SPANNING TREE :

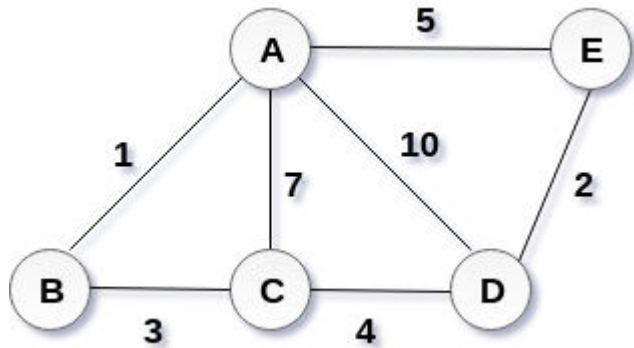
- A spanning tree of a graph, G , is a set of $|V|-1$ edges that connect all vertices of the graph. Thus a minimum spanning tree for G is a graph, $T = (V', E')$ with the following properties:
- $V' = V$
- T is connected
- T is acyclic.

Minimum Spanning Tree

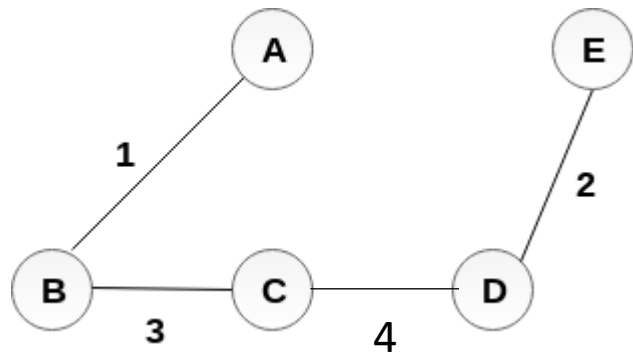
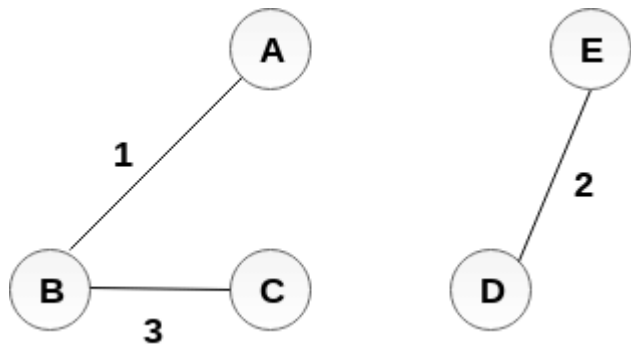
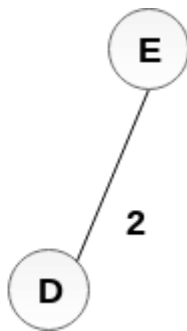
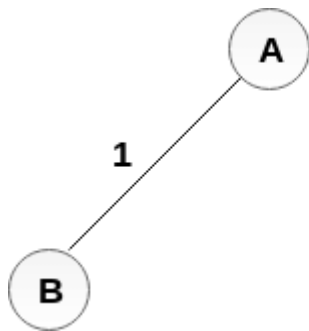
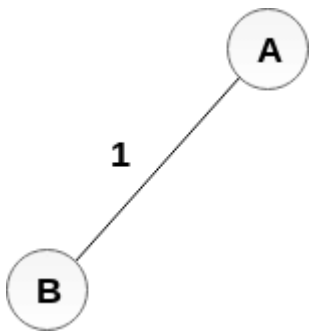
- In general, it is possible to construct multiple spanning trees for a graph, G . If a cost, C_{ij} , is associated with each edge, $E_{ij} = (V_i, V_j)$, then the minimum spanning tree is the set of edges, E_{span} , forming a spanning tree, such that:
- $C = \sum(C_{ij} \mid \text{all } E_{ij} \text{ in } E_{span})$ is a minimum. Eg. The graph has 16 spanning trees.



- Kruskal's algorithm: An approach to determine minimum cost spanning tree of a graph has been given by Kruskal. Here a minimum cost spanning tree T , is built edge by edge. Edges are considered for inclusion in T , in non-decreasing order of their costs. An edge is included in T if it does not form a cycle with the edges already in T .
- Since G is connected and has $n > 0$ vertices, exactly $(n - 1)$ edge will be selected for inclusion in T .
- Algorithm
 1. Sort the edges by weight in ascending order.
 2. Select the lowest cost edge from the list. Remove this edge from list and add this edge to the tree. If the addition results in a cycle discard this edge.
 3. Stop when $(n-1)$ edges have been added to the tree.



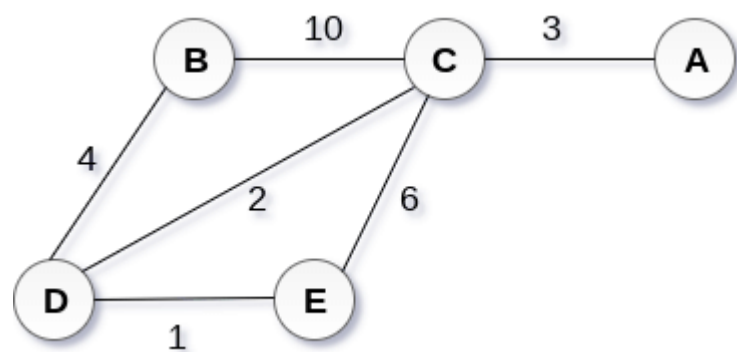
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10



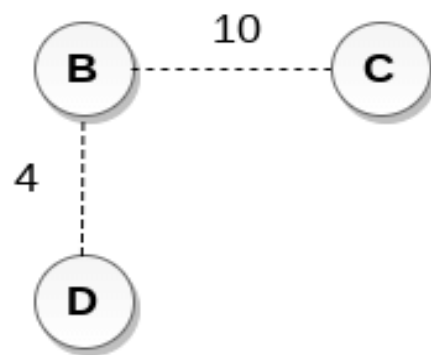
- Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.
- Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

- Prim's algorithm

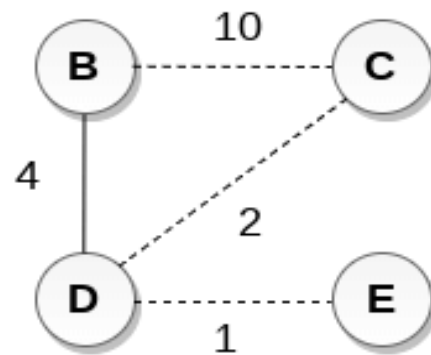
1. Choose an arbitrary vertex V_i and add it to the tree.
2. Select the lowest cost edge that connects vertex V_i to another vertex V_j without forming any cycle.
3. Stop when $n-1$ edges have been added to the tree.



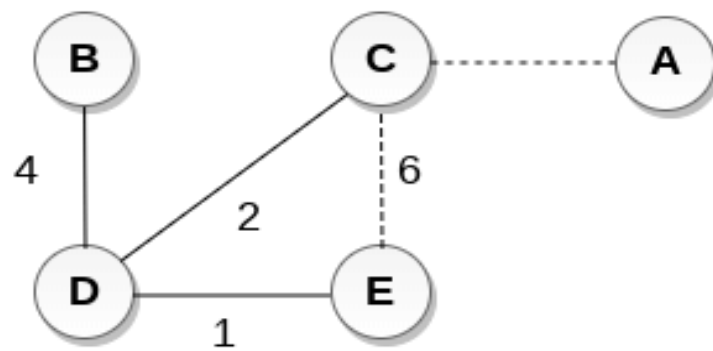
Step 1



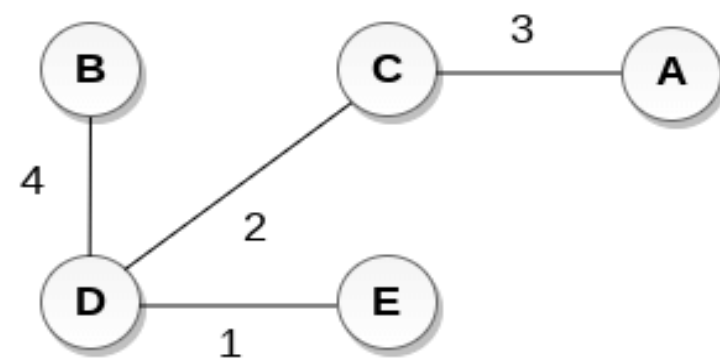
Step 2



Step 3



Step 4



Step 5

Searching & Sorting Technique

Searching

- Searching refers to the operation of finding the location of a given element in a collection of elements. If the element does appear in that list, then the location of that element is recorded and searching is said to be successful. Otherwise the search is said to be unsuccessful.
- Linear search is also called as sequential search algorithm. It is the simplest searching algorithm. In Linear search, 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 is found, then the location of the item is returned; otherwise, the algorithm returns NULL.
- Linear search is implemented using following steps...

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with the first element in the list.

Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function

Step 4 - If both are not matched, then compare search element with the next element in the list.

Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.

Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

- It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted.
The worst-case time complexity of linear search is $O(n)$.

The following steps are followed to search for an element $k = 1$ in the list below.



Start from the first element, compare k with each element x

$k = 1$



↑
 $k \neq 2$



↑
 $k \neq 4$



↑
 $k \neq 0$

If $x == k$, return the index.



↑
 $k = 1$

- Algorithm:

1. Start

2. linear_search (Array , value)

- For each element in the array

- If (searched element == value)

- Return's the searched element location

- end if

- end for

3. end

- Linear search on a list of n elements. In the worst case, the search must visit every element once. This happens when the value being searched for is either the last element in the list, or is not in the list. However, on average, assuming the value searched for is in the list and each list element is equally likely to be the value searched for, the search visits only $n/2$ elements. In best case the array is already sorted i.e $O(1)$.

- Binary search
- A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order.
- In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index is returned.
- Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right.
- If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned..

- Binary search is implemented using following steps...

Step 1 - Read the search element from the user.

Step 2 - Find the middle element in the sorted list.

Step 3 - Compare the search element with the middle element in the sorted list.

Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.

Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.

Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.


Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.

Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Binary Search

Search 46




0	1	2	3	4	5	6	7	8	9
4	10	16	24	32	46	76	112	144	182

$46 > 32$
take upper half




L=0	1	2	3	M=4	5	6	7	8	H=9
4	10	16	24	32	46	76	112	144	182

$46 < 112$
take lower half



0	1	2	3	4	L=5	6	M=7	8	H=9
4	10	16	24	32	46	76	112	144	182

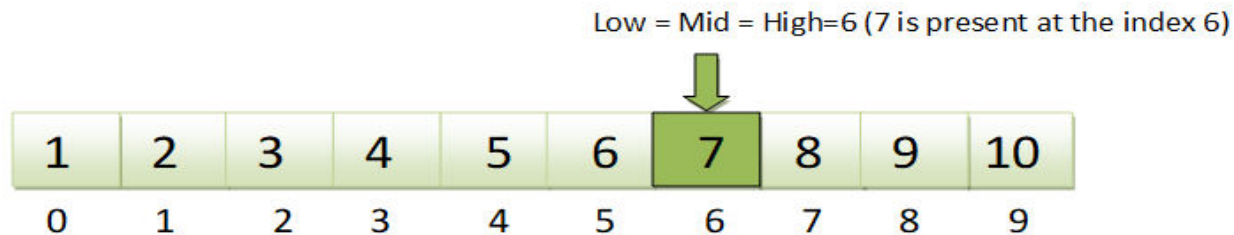
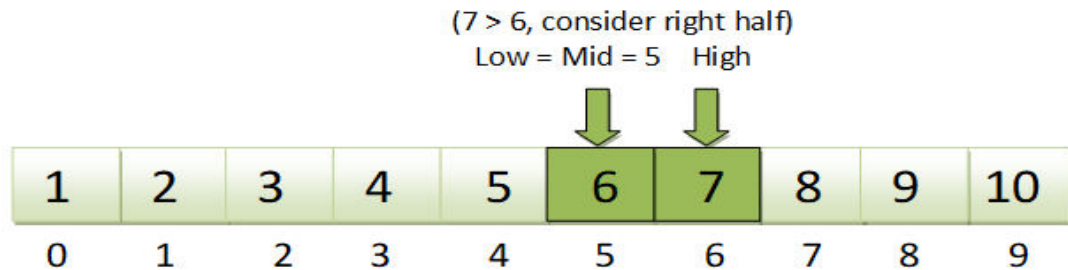
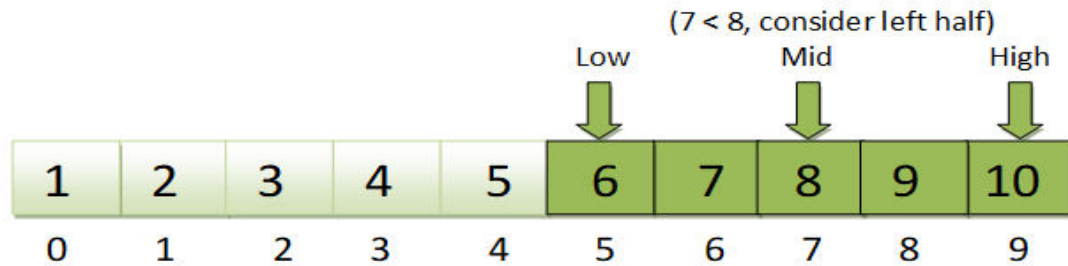
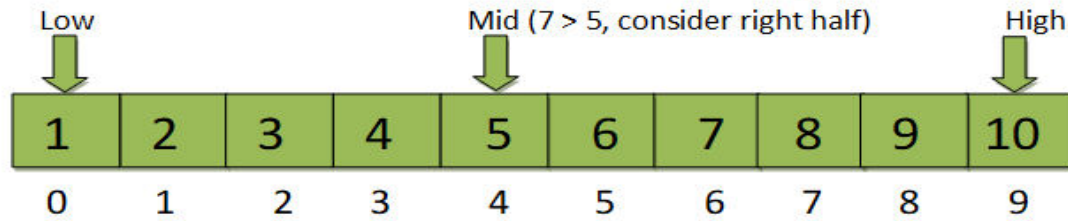
Found 46
at Index.5



0	1	2	3	4	L=M=5	H=6	7	8	9
4	10	16	24	32	46	76	112	144	182

Binary Search

Search the number 7 in the array



A 11 20 25 32 45 60 80 low=0 high=6 mid=low+high/2 SE 11

i 0 1 2 3 4 5 6

1. Mid =3 A[3]=SE 32!=11

2. 32>11 high=mid-1 high =3-1=2

Low=0 high=2 mid=1

A[1]=20 20!=11

3. 20>11

High=mid-1=1-1=0

Low =0 high=0 mid =0

A[0]!=10

10<11

- Binary Search Algorithm

1. Set $BEG = 0$ and $END = N - 1$
2. Set $MID = (BEG + END) / 2$
3. Repeat step 4 to 8 While $(BEG \leq END)$ and $(A[MID] \neq ITEM)$
4. If $(ITEM < A[MID])$ Then
5. Set $END = MID - 1$
6. Else
7. Set $BEG = MID + 1$ [End of If]
8. Set $MID = (BEG + END) / 2$
9. If $(A[MID] == ITEM)$ Then
10. Print: ITEM exists at location MID
11. Else
12. Print: ITEM doesn't exist
- [End of If]
13. Exit

- **Fibonacci Search:** The fibonacci search is an efficient searching algorithm which works on sorted array of length n . It is a comparison-based algorithm and it returns the index of an element which we want to search in array and if that element is not there then it returns -1.

- **Examples:**

1. **Input:** $\text{arr}[] = \{20, 30, 40, 50, 60\}$, $x = 50$

Output: 3 (Element x is present at index 3.)

2. **Input:** $\text{arr}[] = \{25, 35, 45, 55, 65\}$, $x = 15$

Output: -1 (Element x is not present.)

- In this, the fibonacci numbers are used to search an element in given sorted array.

Similarities with Binary Search:

- Fibonacci search works on sorted array only as the binary search.
- The time taken by fibonacci search in worst case is $O(\log n)$ which is similar to the binary search.
- The divide and conquer technique is used by fibonacci search.

Differences with Binary Search:

- Binary search divides array into equal parts but the fibonacci search divides array into unequal parts.
- In fibonacci search, there is no use “/” operator instead of this, it uses + and – operator.
- Fibonacci search can be used if the size of array is larger.

Background:

- $F(n) = F(n-1) + F(n-2)$, $F(0) = 0$, $F(1) = 1$ is way to define fibonacci numbers recursively.
- First few Fibonacci Numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Steps

1. Find $F(K)$ (Kth Fibonacci number) which is greater than or equal to n .
2. if $F(K) = 0$ then
 stop and print element not present
3. $offset = -1$
4. $index = \min(offset + F(K-2), n-1)$
5. if $Search_Ele == A[index]$
 return index & stop the search
 if $Search_Ele > A[index]$
 $K = K - 1$ $offset = index$ and repeat steps 4,5
 if $Search_Ele < A[index]$
 $K = K - 2$ repeat steps 4, 5

A - 2 5 7 13 21 28 31 n = 7 SE = 21

K - 0 1 2 3 4 5 6

F(K)-0 1 1 2 3 5 8

1. $F[K] \geq n$ $F[K]=8$ $K=6$ $\text{offset}=-1$ $F(K)=8$, $F(k-1)+F(K-2)=5+3$

$\text{Index} = \min(\text{offset}+F(K-2), n-1) = \min(-1+3, 7-1) = \min(2,6)=2$

$A[2]=21$ $7!=21$

2. $A[2]<21$ $F(k)=F[k-1]=5$ $k=5$ $\text{offset}=\text{index}=2$

$\text{Index}(\min(2+2,6)=\min(4,6)=4$

$A[4]=21$

Algorithm 1: Fibonacci Search Algorithm

```
input  : An array arr
          number of elements in arr n
          element to be searched key
output: index of key in arr or  $-1$ 

begin
    // Initialize  $F_{k-2}$ ,  $F_{k-1}$  and  $F_k$ 
     $F_{k-2} \leftarrow 0$ 
     $F_{k-1} \leftarrow 1$ 
     $F_k \leftarrow F_{k-1} + F_{k-2}$ 
    // Find the greatest  $F_k$  that is less than n
    while  $F_k \leq n$  do
         $F_{k-2} \leftarrow F_{k-1}$ 
         $F_{k-1} \leftarrow F_k$ 
         $F_k \leftarrow F_{k-1} + F_{k-2}$ 
    end
    // Initialize offset to  $-1$ 
    offset  $\leftarrow -1$ 
    while  $F_{k-2} \geq 0$  do
        index  $\leftarrow \min(\text{offset} + F_{k-2}, n - 1)$ 
        // arr[index] < key so we search to left of  $F_{k-2}$ 
        if arr[index] < key then
             $F_k \leftarrow F_{k-1}$ 
             $F_{k-1} \leftarrow F_{k-2}$ 
             $F_{k-2} \leftarrow F_k - F_{k-1}$ 
            offset  $\leftarrow \text{index}$ 
        // arr[index] > key so we search to right of  $F_{k-2}$ 
        else if arr[index] > key then
             $F_k \leftarrow F_{k-2}$ 
             $F_{k-1} \leftarrow F_{k-1} - F_{k-2}$ 
             $F_{k-2} \leftarrow F_k - F_{k-1}$ 
        // arr[index] == key so we return index
        else
            return index
        end
    end

    // Compare the last element of arr with key
    if  $F_{k-1}$  & arr[offset + 1] == key then
        return offset + 1
    end

    // key not found in arr
    return  $-1$ 
end
```

- Sorting :
- Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching.
- There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc. Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data: Roll No., Name, Age Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

- CATEGORIES OF SORTING

- An internal sort is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory.
- External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub files are combined into a single larger file.
- We can say a sorting algorithm is stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

- Insertion Sort:
- It is a very simple and efficient sorting algorithm for sorting a small number of elements in which the sorted array is built one element at a time. The main idea behind insertion sort is that it inserts each element into its proper location in the sorted array.
- Let us take there are n elements the array `arr`. Then process of inserting each element in proper place is as
- Pass 1- `arr[0]` is already sorted because of only one element.
- Pass 2- `arr[1]` is inserted before or after `arr[0]`. So `arr[0]` and `arr[1]` are sorted.
- Pass 3- `arr[2]` is inserted before `arr[0]` or in between `arr[0]` and `arr[1]` or after `arr[1]`. So `arr[0]`, `arr[1]` and `arr[2]` are sorted.
- Pass 4- `arr[3]` is inserted into its proper place in array `arr[0]`, `arr[1]`, `arr[2]` So, `arr[0]` `arr[1]` `arr[2]` and `arr[3]` are sorted.
-
-
-
- Pass N - `arr[n-1]` is inserted into its proper place in array. `arr[0]`, `arr[1]`, `arr[2]`,.....
`arr[n-2]`. So, `arr[0]` `arr[1]`,..... `arr[n-1]` are sorted.

Insertion sort has following advantages-

- Simple implementation
- Efficient for small data sets
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount $O(1)$ of additional memory space

Time Complexity:

- Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is $O(n)$.
- Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is $O(n^2)$.
- Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is $O(n^2)$.

5 2 1 3 6 4

- Compare first and second elements i.e., 5 and 2, arrange in increasing order.

2 5 1 3 6 4

- Next, sort the first three elements in increasing order.

1 2 5 3 6 4

- Next, sort the first four elements in increasing order.

1 2 3 5 6 4

- Next, sort the first five elements in increasing order.

1 2 3 5 6 4

- Next, sort the first six elements in increasing order.

1 2 3 4 5 6

- Algorithm:

- Insertion(int a[],int n)

1. Start

2. set $i=1$

3. repeat the steps 4,5,8 and 9 while($i < n$)

4. set $temp=a[i]$ and $j=i-1$

5. repeat steps 6 and 7 while $j \geq 0 \ \&\& \ a[j] > temp$

6. set $a[j+1] = a[j]$

7. $j=j-1$

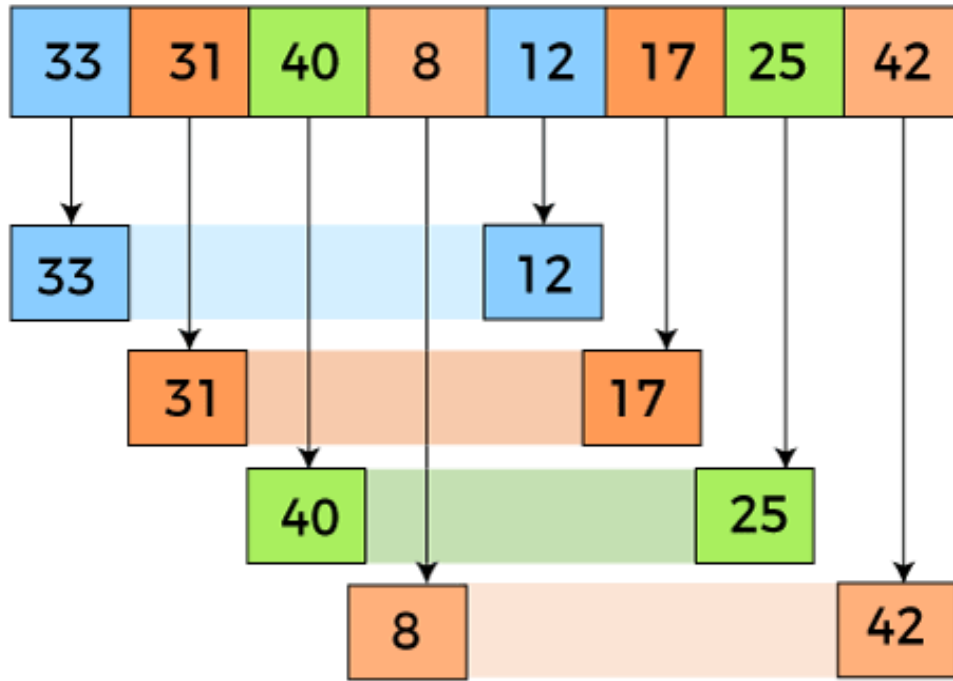
8. set $a[j+1]=temp$

9. set $i=i+1$

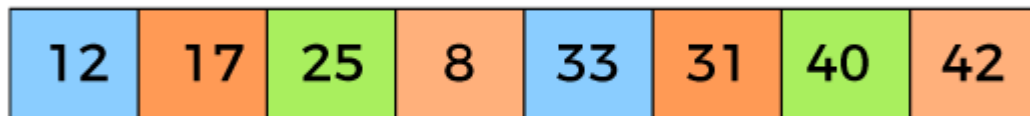
10. stop

- Shell Sort:
- It is a sorting algorithm that is an extended version of insertion sort. Shell sort has improved the average time complexity of insertion sort. As similar to insertion sort, it is a comparison-based and in-place sorting algorithm. Shell sort is efficient for medium-sized data sets.
- In insertion sort, at a time, elements can be moved ahead by one position only. To move an element to a far-away position, many movements are required that increase the algorithm's execution time. But shell sort overcomes this drawback of insertion sort. It allows the movement and swapping of far-away elements as well.
- This algorithm first sorts the elements that are far away from each other, then it subsequently reduces the gap between them. This gap is called as **interval**.
- Some of the intervals that can be used in the shell sort algorithm are:
- Shell's original sequence: $N/2, N/4, \dots, 1$.
- Papernov & Stasevich increment: 1,3,5,9,17,33,65.....
- Hibbard's increments: 1,3,7,15,31,63,127,255,511.....
- Pratt: 1,2,3,4,6,9,8,12,18,27,16,24,36,54,81,.....

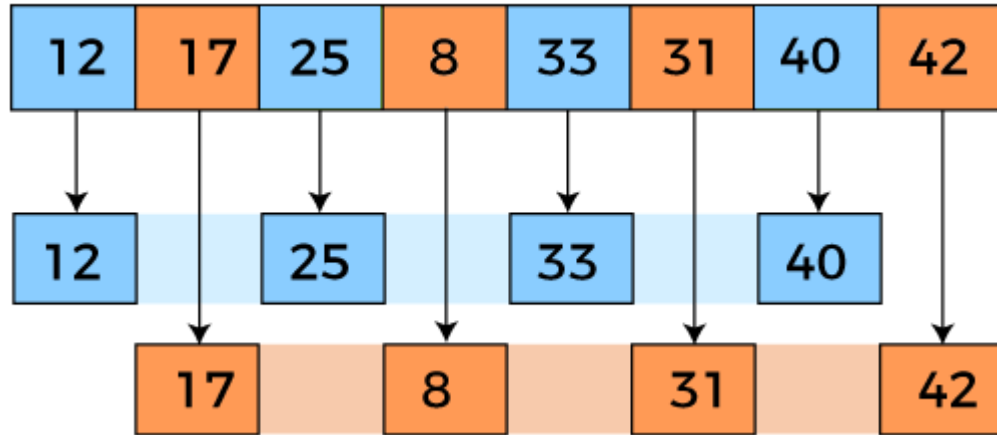
At the interval of 4, the sublists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.



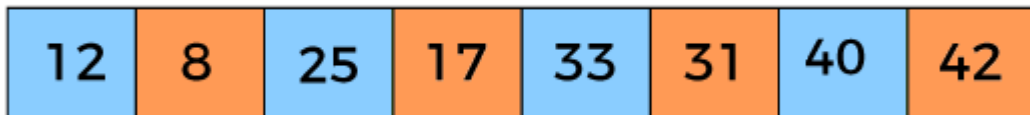
Now, we have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -



- In the second loop, elements are lying at the interval of 2 ($n/4 = 2$), where $n = 8$.
- Now, we are taking the interval of **2** to sort the rest of the array. With an interval of 2, two sublists will be generated - {12, 25, 33, 40}, and {17, 8, 31, 42}.



Now, we again have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -



- In the third loop, elements are lying at the interval of 1 ($n/8 = 1$), where $n = 8$. At last, we use the interval of value 1 to sort the rest of the array elements. In this step, shell sort uses insertion sort to sort the array elements.

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

- Algorithm:

1. ShellSort(a, n) // 'a' is the given array, 'n' is the size of array

2. for (gap = n/2; gap > 0; gap=gap / 2)

3. for (j = gap; j < n-1; j ++)

4. for (i = j-gap; i >= 0 ; i =i-gap)

5. if(a[i+gap] > a[i])

 break;

 else

 swap(a[i+gap],a[i])

 [End of if]

 [End of for]

 [End of for]

6. End ShellSort

- The performance of the shell sort depends on the type of sequence used for a given input array.
- **Best Case Complexity** - It occurs when there is no sorting required, i.e., the array is already sorted. The best-case time complexity of Shell sort is **$O(n \cdot \log n)$** .
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Shell sort is **$O(n \cdot \log n)$** .
- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Shell sort is **$O(n^2)$** .

- Heap sort is a comparison-based sorting technique based on Binary Heap data structure.
- Binary Heap
A Binary Heap is a Complete Binary Tree where items are stored in a special order such that the value in a parent node is greater or smaller than the values in its two children nodes. The former is called max heap and the latter is called min-heap. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- The heap can be represented by a binary tree or array. Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-efficient. If the parent node is stored at index I , the left child can be calculated by $2 * I + 1$ and the right child by $2 * I + 2$.
- The process of reshaping a binary tree into a Heap data structure is known as 'heapify'. A binary tree is a tree data structure that has two child nodes at max. If a node's children nodes are 'heapified', then only 'heapify' process can be applied over that parent node. A heap should always be a complete binary tree.
- Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called 'heapify' on all the non-leaf elements of the heap. i.e. 'heapify' uses recursion.

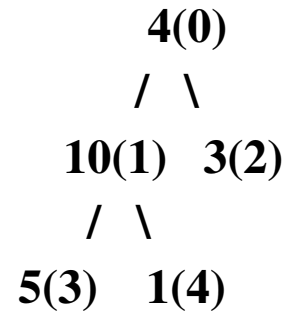
Example:

```

      30(0)
     /  \
  70(1) 50(2)

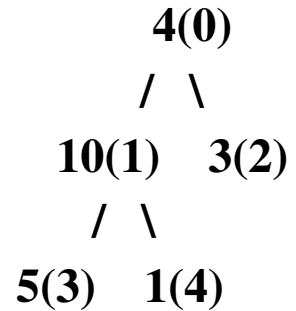
```

Input data: 4, 10, 3, 5, 1

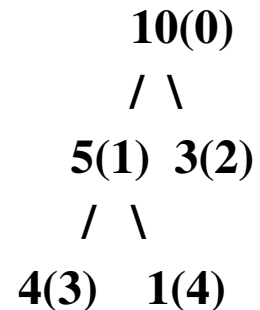


The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:



Applying heapify procedure to index 0:



The heapify procedure calls itself recursively to build heap in top down manner.

Algorithm for “heapify”:

heapify(array)

Root = array[0]

Largest = largest(array[0] , array [2 * 0 + 1], array[2 * 0 + 2])

if(Root != Largest)

Swap(Root, Largest)

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3. Repeat step 2 while the size of the heap is greater than 1(heap contains one element).

- **MaxHeapify(A, i)**
 - largest = i
 - l = left(i)
 - r = right(i)
 - if** l ≤ heap-size[A] and A[l] > A[largest]
 - then largest = l
 - else if** r ≤ heap-size[A] and A[r] > A[largest]
 - then largest = r
 - if** largest ≠ i
 - then swap A[i] with A[largest]
 - MaxHeapify(A, largest)
- end func
- **BuildMaxHeap(A)**
 - heap-size[A] = length[A]
 - for** i = |length[A]/2| downto 1
 - do** MaxHeapify(A, i)
- end func
- **HeapSort(A)**
 - BuildMaxHeap(A)
 - for** i = length[A] downto 2
 - do** swap A[1] with A[i]
 - heap-size[A] = heap-size[A] − 1
 - MaxHeapify(A, 1)
- end func

Quick Sort

- The Quick sort algorithm was developed by a British computer scientist Tony Hoare in 1959. The name "Quick Sort" comes from the fact that, quick sort is capable of sorting a list of data elements significantly faster than other sorting algorithms.
- It is one of the most efficient sorting algorithms and is based on divide and conquer strategy. It picks an element as pivot and partitions the given array. The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use divide and conquer strategy. In quick sort, the partition of the list is performed based on the element called pivot.
- Here pivot element is one of the elements in the list. The list is divided into two partitions such that "all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot". There are many different versions of Quick Sort that pick pivot in different ways.
 1. Always pick first element as pivot.
 2. Always pick last element as pivot.
 3. Pick a random element as pivot.
 4. Pick median as pivot.
- The key process in Quick Sort is partition().

The steps in the Quick sort algorithm are:

1. Select Pivot Element.
2. Split the given array with respect to pivot element in a way such that every element to the left of the pivot is smaller than or equal to the pivot and every element to the right of the pivot is greater than or equal to the pivot.
3. Repeat step 1 and 2 with the left sub list.
4. Repeat step 1 and 2 with the right sub list.

In Quick sort algorithm, partitioning of the list is performed using following steps...

1. Consider the first element of the list as pivot (i.e., Element at first position in the list).
2. Define two variables i and j . Set i and j to first and last elements of the list respectively.
3. Increment i until $\text{list}[i] \leq \text{pivot}$ then stop.
4. Decrement j until $\text{list}[j] > \text{pivot}$ then stop.
5. If $i < j$ then exchange $\text{list}[i]$ and $\text{list}[j]$.
6. Repeat steps 3,4 & 5 until $i > j$.
7. Exchange the pivot element with $\text{list}[j]$ element.

low i high, j

42	37	11	98	36	72	65	10	88	78
----	----	----	----	----	----	----	----	----	----

42>37 so, i++



42 > 98 stop,
i++, &
compare 42
with a[j] = 78

with $a[j] = 78$
 42 < 78 so, j--

				i					j	
42	37	11	98	36	72	65	10	88	78	42 < 88 so, j--

i

42	37	11	98	36	72	65	10	88	78
----	----	----	----	----	----	----	----	----	----

42 < 10 stop, j--



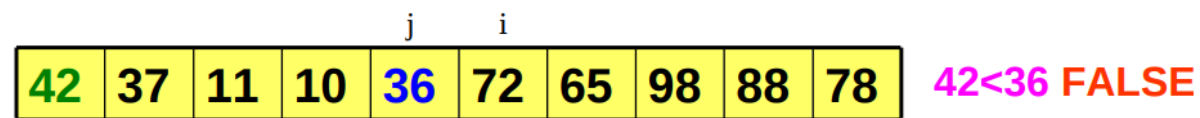
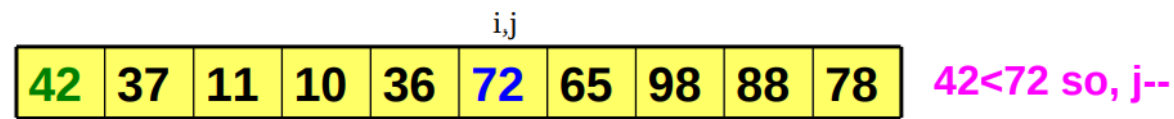
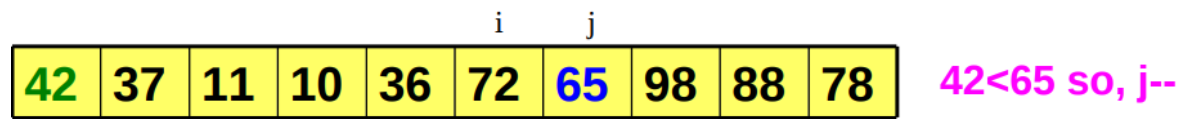
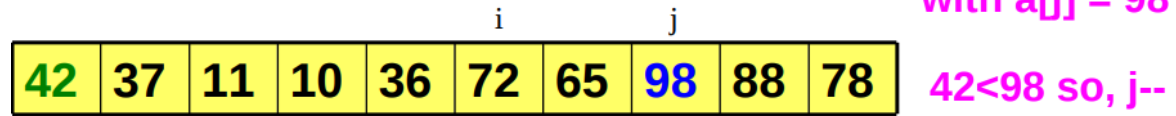
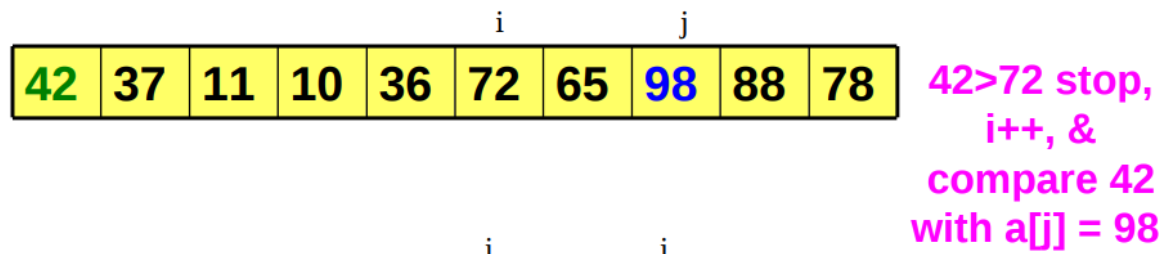
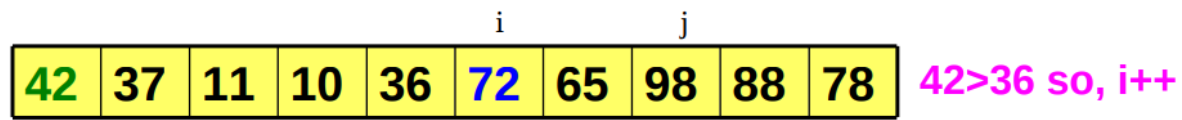
Since $i < j$, exchange $a[i]$ with $a[j]$ and repeat the process

			i				j			
42	37	11	10	36	72	65	98	88	78	42>10 so, i++

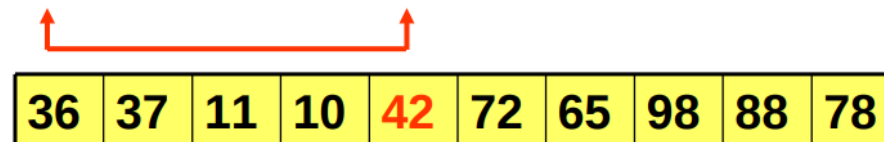
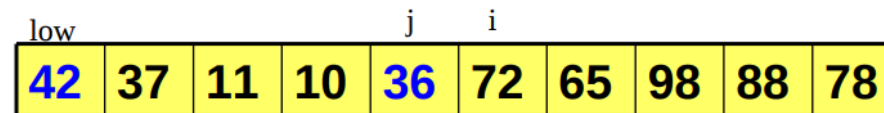
i

42	37	11	10	36	72	65	98	88	78
----	----	----	----	----	----	----	----	----	----

42>36 so, i++



Since i exceeds j , exchange $a[\text{low}]$ with $a[j]$.



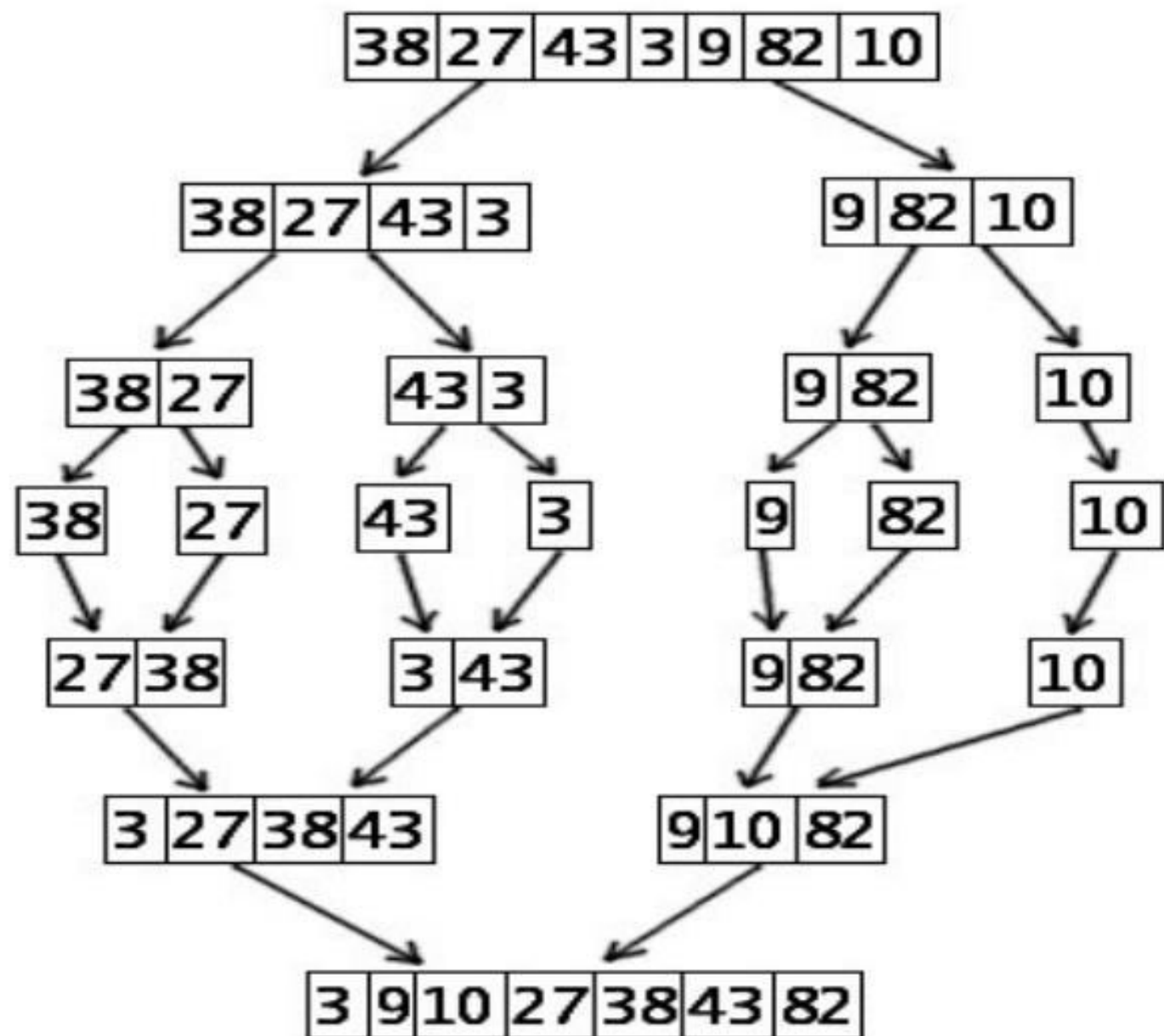
Merge Sort

- Merge sort is a comparison based sorting algorithm. It is based on divide and conquer rule. In this algorithm we divide the array into equal halves first and then combines them in a sorted manner. This sorting algorithm suffers from space complexity but is it a stable algorithm because it preserves the input order of equal elements in the sorted output.
- It is most respected and trusted sorting algorithm because of its time complexity in worst-case being **$O(n \log n)$** .

Algorithm

- Step 1: if it is only one element in the list it is already sorted, return.
- Step 2: divide the list recursively into two halves until it can no more be divided.
- Step 3: merge the smaller lists into new list in sorted order.

Merge sort keeps on dividing the list into equal halves until it can no more be divided. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.



MERGE_SORT(arr, beg, end)

1. if $beg < end$
2. set $mid = (beg + end)/2$
3. MERGE_SORT(arr, beg, mid)
4. MERGE_SORT(arr, mid + 1, end)
5. MERGE (arr, beg, mid, end)
6. end of if

END MERGE_SORT

The important part of the merge sort is the MERGE function. This function performs the merging of two sorted sub-arrays that are $A[beg...mid]$ and $A[mid+1...end]$, to build one sorted array $A[beg...end]$. So, the inputs of the MERGE function are $A[]$, beg, mid, and end.

- void merge(int A[], int start, int mid, int end) {
//stores the starting position of both parts in temporary variables.
int p = start ,q = mid+1;
int Arr[end-start+1] , k=0;
for(int i = start ;i <= end ;i++) {
if(p > mid) //checks if first part comes to an end or not .
Arr[k++] = A[q++] ;
else if (q > end) //checks if second part comes to an end or not
Arr[k++] = A[p++] ;
else if(A[p] < A[q]) //checks which part has smaller element.
Arr[k++] = A[p++] ;
else
Arr[k++] = A[q++];
}
for (int p=0 ; p< k ;p ++) {
/* Now the real array has elements in sorted manner including both parts.*/
A[start++] = Arr[p] ;
}}

Radix sort

- Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order. In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from least significant digit to the most significant digit.
- Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers. For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.
- The Radix sort algorithm is performed using the following steps...
 - Step 1 - Define 10 queues each representing a bucket for each digit from 0 to 9.
 - Step 2 - Consider the least significant digit of each number in the list which is to be sorted.
 - Step 3 - Insert each number into their respective queue based on the least significant digit.
 - Step 4 - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
 - Step 5 - Repeat from step 3 based on the next least significant digit.
 - Step 6 - Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.



Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

82, 901, 1000, 12, 1500, 770, 550 & 230



Group all the numbers from queue-0 to queue-9 in the order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

100, 150, 901, 82, 12, 23, 55 & 77



Group all the numbers from queue-0 to queue-9 in the order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundred placed digit) of every number.

100, 901, 12, 23, 150, 55, 77 & 82



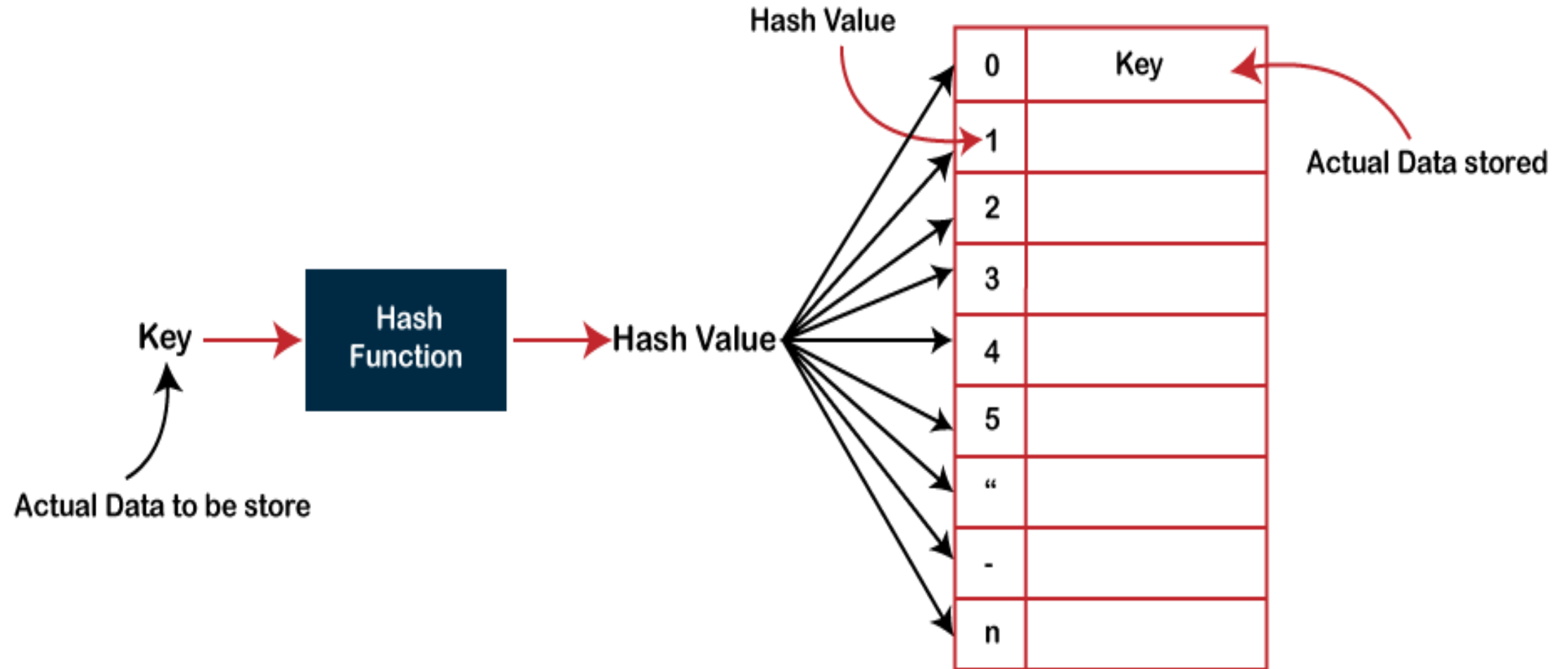
Group all the numbers from queue-0 to queue-9 in the order they have inserted & consider the list for next step as input list.

12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the increasing order.

- Hashing:
- Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is $O(1)$. Till now, we studied the techniques for searching, i.e., linear search and binary search. The worst time complexity in linear search is $O(n)$, and $O(\log n)$ in binary search. The drawbacks of both searching methods are as the number of elements increases, time taken to perform the search also increases. This becomes problematic when total number of elements become too large.
- 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.
- Hashing is well known technique to search any particular element among several elements. It minimizes the number of comparisons while performing the search.
- 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.
- In hashing, large keys are converted into small keys by using hash functions. The values are stored in a data structure called hash table.
- Hash key value is a special value that serves as an index for a data item. It indicates where the data item should be stored in the hash table. Hash key value is generated using a hash function.
- E.g. In colleges, each student is assigned a unique roll number that can be used to retrieve information about them.

Hashing



- Types of Hash Functions-

There are various types of hash functions available such as-

- Mid Square Hash Function
- Division Hash Function
- Folding Hash Function etc.

The properties of a good hash function are-

- It should be easy to compute.
- It should minimize the number of collisions.
- It should distribute the keys uniformly over the table.

Collision

- No matter what the hash function, there is possibility that two different keys could resolve to the same hash address. This situation is known as collision.

Handling the collision:

- Separate Chaining (Open hashing)
- Open addressing (closed hashing) (Linear probing, Quadratic probing, random probing, Double hashing)

$$36 \% 8 = 4$$

$$18 \% 8 = 2$$

$$72 \% 8 = 0$$

$$43 \% 8 = 3$$

$$6 \% 8 = 6$$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
72		18	43	36		6	

Fig: Division method

Chaining: To handle the collision, This technique creates a linked list to the slot for which collision occurs.

- The new key is then inserted in the linked list.
- These linked lists to the slots appear like chains.
- That is why, this technique is called as **separate chaining**.

Hash key = key % table size

$$36 \% 8 = 4$$

$$18 \% 8 = 2$$

$$72 \% 8 = 0$$

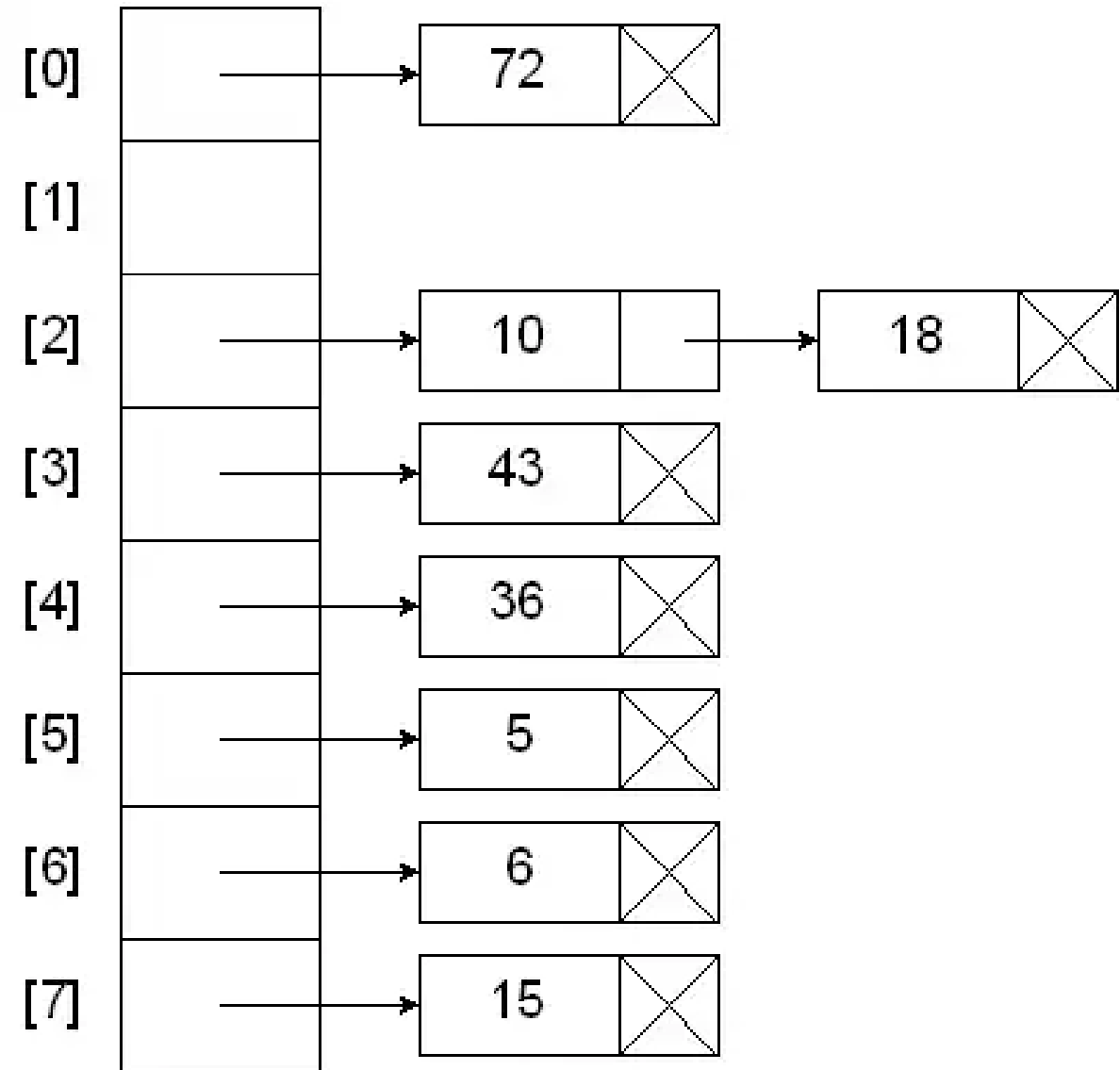
$$43 \% 8 = 3$$

$$6 \% 8 = 6$$

$$10 \% 8 = 2$$

$$5 \% 8 = 5$$

$$15 \% 8 = 7$$



- **Open Addressing:**

- Linear Probing-In linear probing,

When collision occurs, we linearly probe for the next bucket.

We keep probing until an empty bucket is found.

- Quadratic Probing- In quadratic probing,

When collision occurs, we probe for i^2 'th bucket in i^{th} iteration.

We keep probing until an empty bucket is found.

- Double Hashing- In double hashing,

We use another hash function $\text{hash2}(x)$ and look for $i * \text{hash2}(x)$ bucket in i^{th} iteration.

It requires more computation time as two hash functions need to be computed.

Linear Probing

Hash key = key % table size

$$36 \% 8 = 4$$

$$18 \% 8 = 2$$

$$72 \% 8 = 0$$

$$43 \% 8 = 3$$

$$6 \% 8 = 6$$

$$10 \% 8 = 2$$

$$5 \% 8 = 5$$

$$15 \% 8 = 7$$

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5