

Vision 2023

A Course for GATE & PSUs

Computer Science Engineering

Programming and
Data structures

CHAPTER 5

Trees

CHAPTER

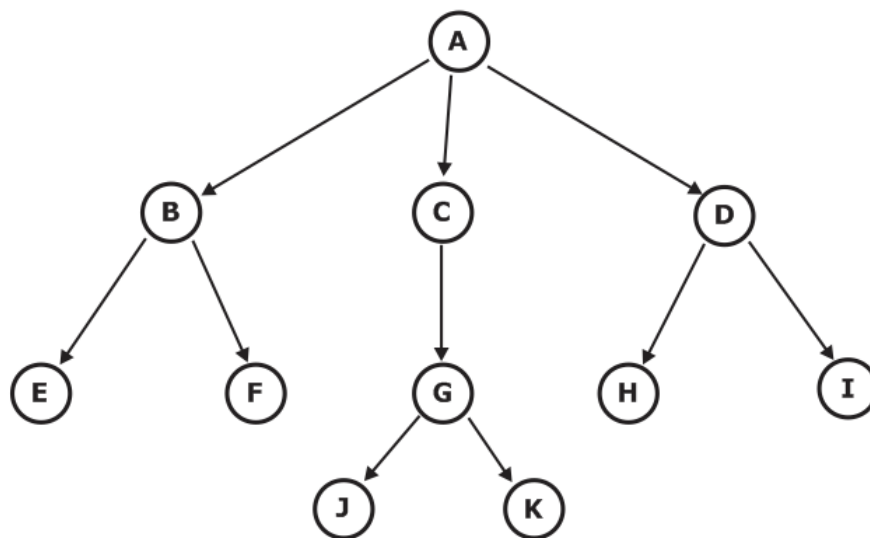
5

PROGRAMMING & DATA STRUCTURES

TREES

1. TREES

- A tree is a non-linear data structure designated at a special node called the root and elements are arranged in levels without containing cycles.
- All the elements are arranged in layers.
- A unique path traverses from root to any node of the tree.
- Every child has only one parent, but the parent can have many children.
- If a tree has N vertices(nodes) then the number of edges is always one less than the number of nodes(vertices) i.e $N-1$. If it has more than $N-1$ edges it is called a graph not a tree.



- A is the root of the tree
- A is Parent of B, C and D
- B is child of A
- B, C and D are siblings
- A is grand-parent of E, F, G, H and I

1.1. TYPES OF TREES

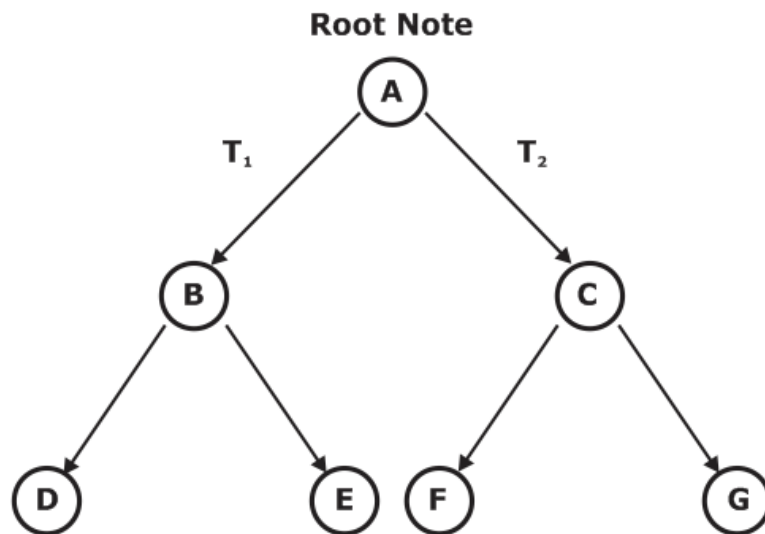
1.1.1. Binary Tree:

It is a special type of tree where each node of the tree contains either 0 or 1 or 2 children.
(or)

Binary Tree is either empty, or it consists of a root with two binary

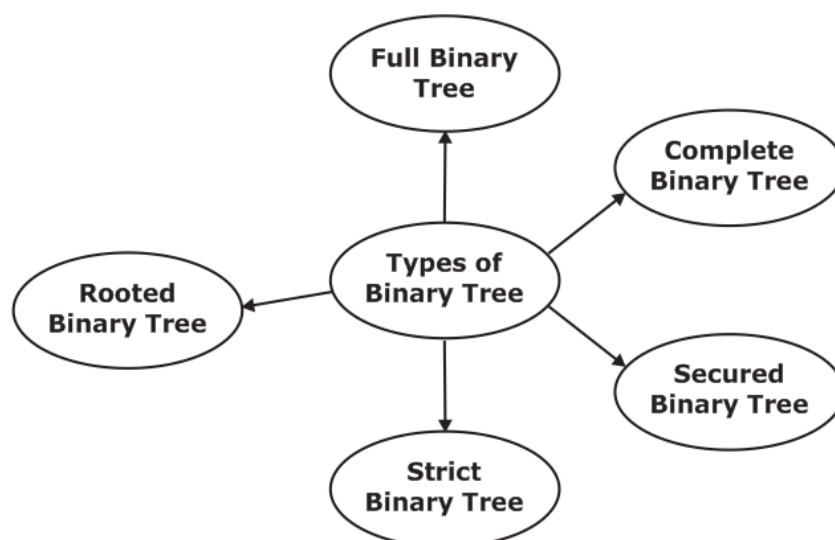
Properties of binary tree :

- Binary tree partitioned into three parts.
- First subset contains root of tree
- Second subset is called left subtree
- Another subset is called right subtree
- Each subtree is a binary tree.
- Degree of any node is 0/1/2.
- The maximum number of nodes in a tree with height 'h' is $2^{h+1} - 1$.
- The maximum number of nodes at level 'i' is $2^i - 1$.
- For any non-empty binary tree, the number of terminal nodes with n^2 , nodes of degree 2 is $N_0 = n^2 + 1$
- The maximum number of nodes in a tree with depth d is $2^d - 1$.



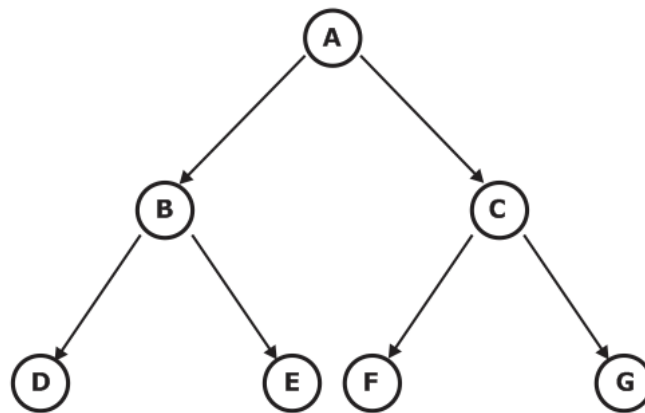
- Time Complexities:
 Insertion: $O(n)$ {Every Case}
 Deletion: $O(n)$ {Every Case}

1.1.2. Types of Binary Trees:



a. Full Binary Tree

- If each node of binary tree has either two children or no child at all, is said to be a **Full Binary Tree**.
- Full binary tree is also called as **Strictly Binary Tree**.

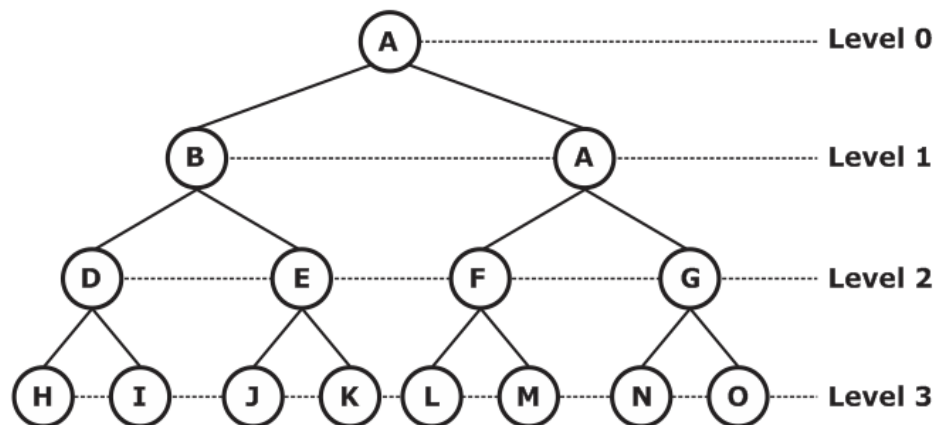


Full Binary Tree

- Every node in the tree has either 0 or 2 children.
- Full binary tree is used to represent mathematical expressions.

b. Complete Binary Tree

- If all levels of tree are completely filled except the last level and the last level has all keys as left as possible, is said to be a **Complete Binary Tree**.
- Complete binary tree is also called as **Perfect Binary Tree**.



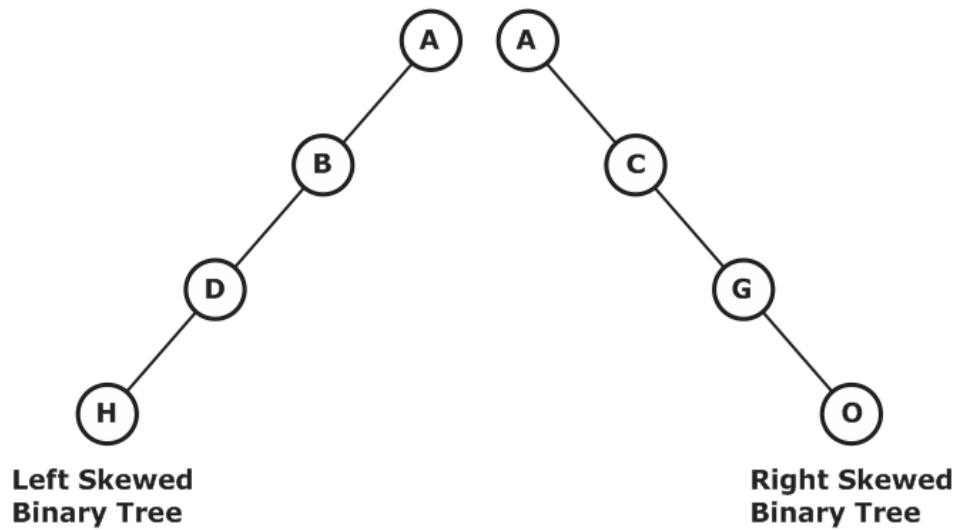
Complete Binary Tree

- In a complete binary tree, every internal node has exactly two children and all leaf nodes are at the same level.
- For example, at Level 2, there must be $2^2 = 4$ nodes and at Level 3 there must be $2^3 = 8$ nodes.

c. Skewed Binary Tree

- If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.

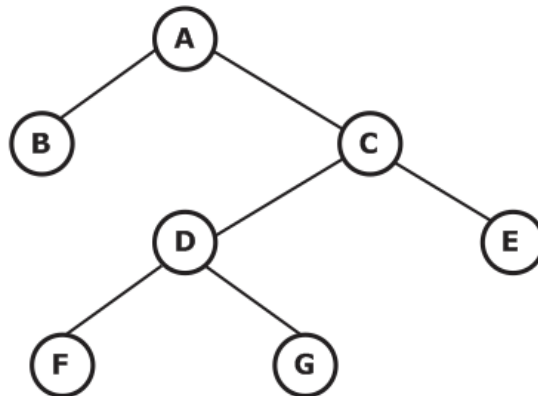
- In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.



- In a left skewed tree, most of the nodes have the left child without corresponding right child.
- In a right skewed tree, most of the nodes have the right child without corresponding left child.

d. Strict Binary Tree:

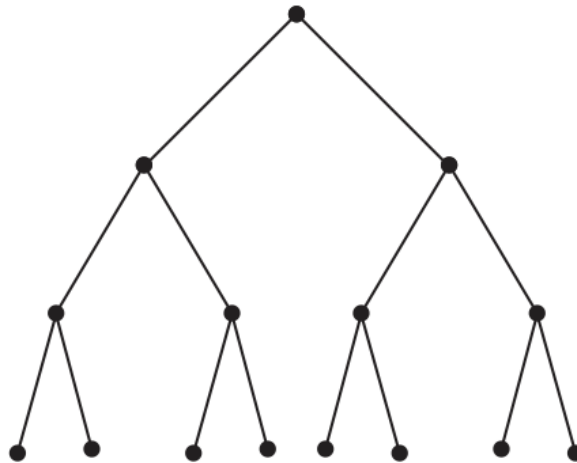
- If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is called a strict binary tree.
- A strict binary tree with n leaves always contains $2n - 1$ nodes.



- If every non-leaf node in a **binary tree** has nonempty left and right subtrees, the **tree** is termed a **strict binary tree**. Or, to put it another way, all of the nodes in a **strict binary tree** are of degree zero or two, never degree one.

e. Rooted Binary Tree:

A rooted binary tree is a binary tree in which only the root is allowed to have degree 2. The remaining nodes have degrees equal to either 1 or 3.

**f. Binary Search Tree:**

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- a. All the elements of the left subtree of a node have a key less than or equal to its parent node's key.
- b. All the elements of the right subtree of a node have a key greater than its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left subtree and the right subtree and can be defined as –

$$\text{left_subtree (keys)} \leq \text{node (key)} \leq \text{right_subtree (keys)}$$

• Time Complexities:

Insertion: $O(n)$ {worst Case}

$O(1)$ { Best Case}

$O(\log n)$ { Average Case}

Deletion: $O(n)$ {worst Case}

$O(1)$ { Best Case}

$O(\log n)$ { Average Case}

1.2. TREE TRAVERSAL

Binary tree traversing methods :

The binary tree contains 3 parts :

V = root

L = Left subtree

R = Right Subtree

1.2.1. Pre-order : (V, L, R)

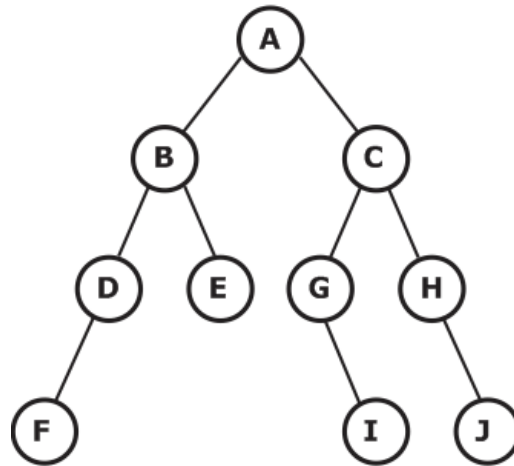
- Visit root of the tree first
- Traverse the left-subtree in pre-order
- Traverse the right-subtree in preorder

1.2.2. In-order : (L, V, R)

- Traverse the left-subtree in in-order
- Visit Root of the tree
- Traverse right-subtree in in-order

1.2.3. Post-order : (L, R, V)

- Traverse the left subtree in post-order
- Traverse the Right-subtree in post-order
- Visit root of the tree

Example 1 :

Pre-order : A B D F E C G I H J

In-order : F D B E A G I C H J

Post-order : F D E B I G J H C A

Pre-order, In-order and post-order uniquely identify the tree.

1.3. OPERATIONS ON BINARY SEARCH TREE**1.3.1. BST Insertion:**

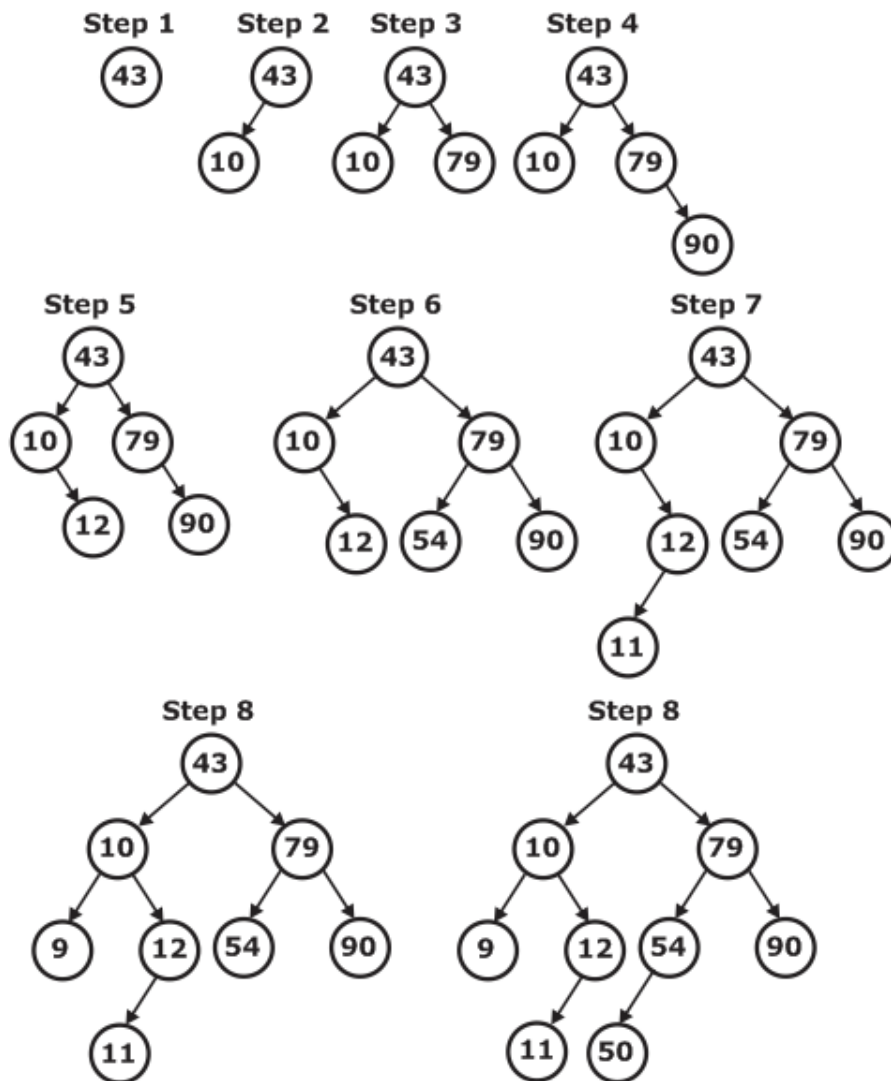
Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that it must not violate the property of binary search tree at each value.

1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of the tree, pointing to NULL.
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.
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.
5. If this is false, then perform this operation recursively with the right subtree of the root.

Example 2: Insert 43, 10, 79, 90, 12, 54, 11, 9, 50 in a BST.

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

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



Binary search Tree Creation

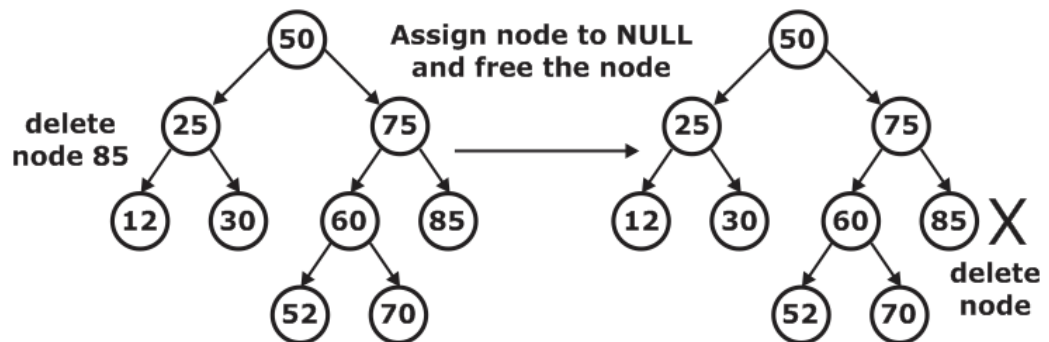
1.3.2. BST Deletion:

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 the binary search tree doesn't violate. There are three situations of deleting a node from a binary search tree.

CASE 1: 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 simply free the allocated space.

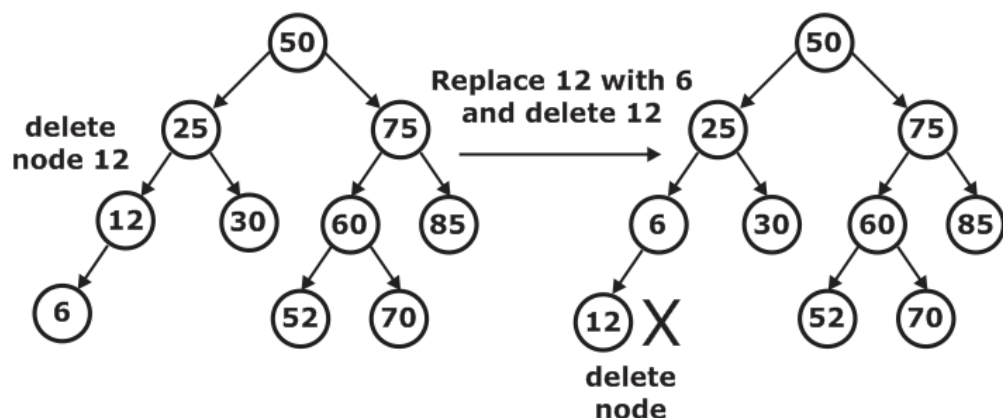
Here node 85 is deleted, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



CASE 2: 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.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



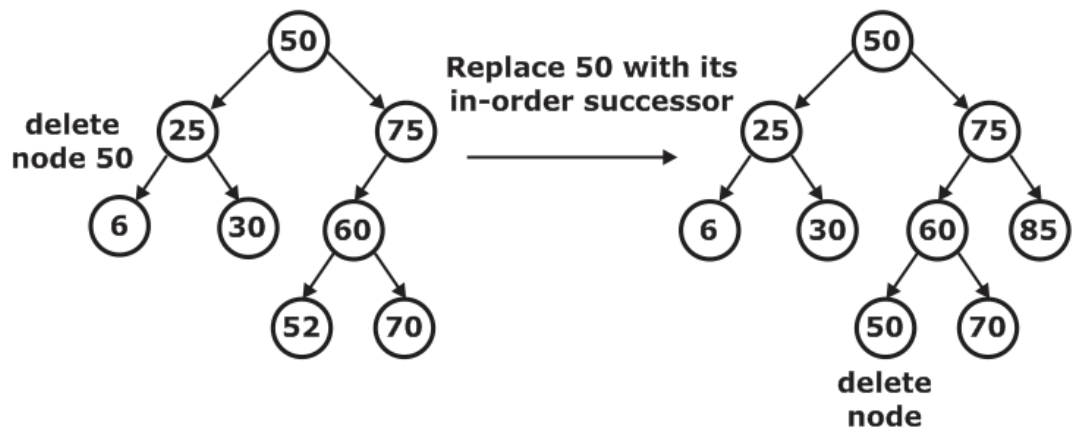
CASE 3: The node to be deleted has two children.

It is a bit complex compared to the 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.

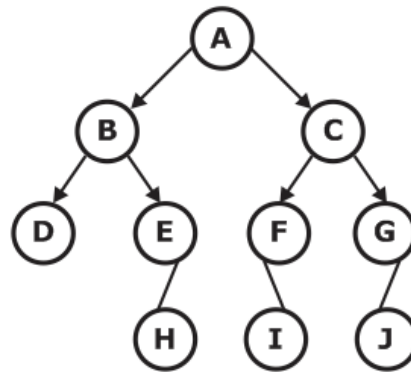
In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



Example 3: Which of the following list of nodes corresponds to post order traversal of the binary tree shown below?



- (A) A B C D E F G H I J
- (B) J I H G F E D C B A
- (C) D H E B I F J G C A
- (D) D E H F I G J B C A

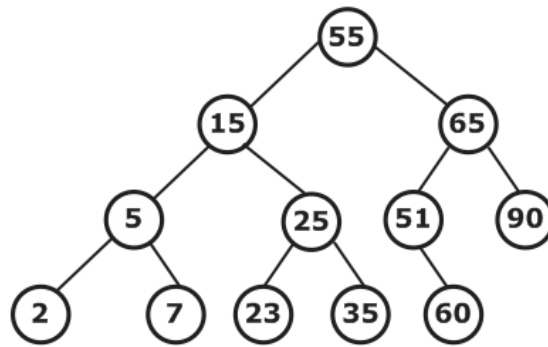
Sol: The correct option is C as in post order traversal we follow left child, right child, root. Hence DHEBIFJGCA.

Example 4: A binary search tree is generated by inserting in order the following integers: 55, 15, 65, 5, 25, 59, 90, 2, 7, 35, 60, 23

The number of nodes in the left subtree and right subtree of the root respectively are

- (A) 8, 3
- (B) 7, 4
- (C) 3, 8
- (D) 4, 7

Ans. 55, 15, 65, 5, 25, 59, 90, 2, 7, 35, 60, 23



Here 55 is the root node so, all the elements smaller than 55 will go on the left of all the elements greater than 55 will go on the RHS.

Hence, option B is correct

Example 5: How many distinct binary search trees can be constructed out of 4 distinct keys?

- a.14
- b.24
- c.35
- d.5

Solution-

Number of distinct binary search trees possible with 4 distinct keys

$$\begin{aligned}
 &= {}^{2n}C_n / n+1 \\
 &= {}^{2 \times 4}C_4 / 4+1 \\
 &= {}^8C_4 / 5 \\
 &= 14
 \end{aligned}$$

Thus, Option (A) is correct.

Example 6: Consider we want to draw all the binary trees possible with 3 unlabelled nodes.

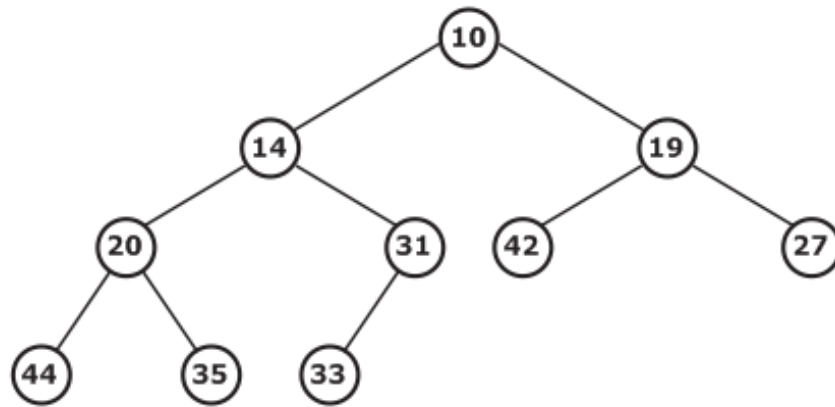
Sol: Number of binary trees possible with 3 unlabelled nodes

$$\begin{aligned}
 &= {}^{2 \times 3}C_3 / (3 + 1) \\
 &= {}^6C_3 / 4 \\
 &= 5
 \end{aligned}$$

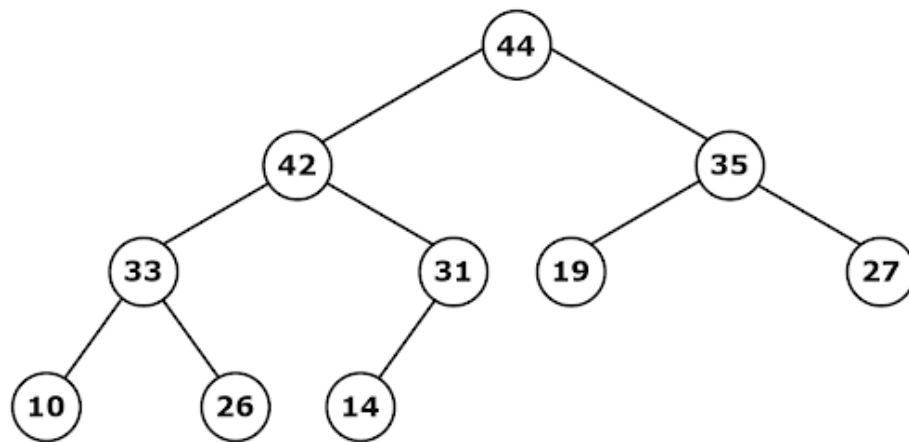
1.4 HEAPS

- Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly.
- Heaps also have an additional property that all leaves should be at h or h-1 levels, where h is the height of the tree.

Min-Heap – Where the value of the root node is less than or equal to either of its children. The same property must be true for all subtrees.



Max-Heap – Where the value of the root node is greater than or equal to either of its children. The same property must be true for all subtrees.



1.4.1. Implementation of a Heap in an array

To implement a min-heap using an array, put the min-heap in the image above into an array:

10	14	19	26	31	42	27	10	26	14
----	----	----	----	----	----	----	----	----	----

By looking at the i -th index in an array:

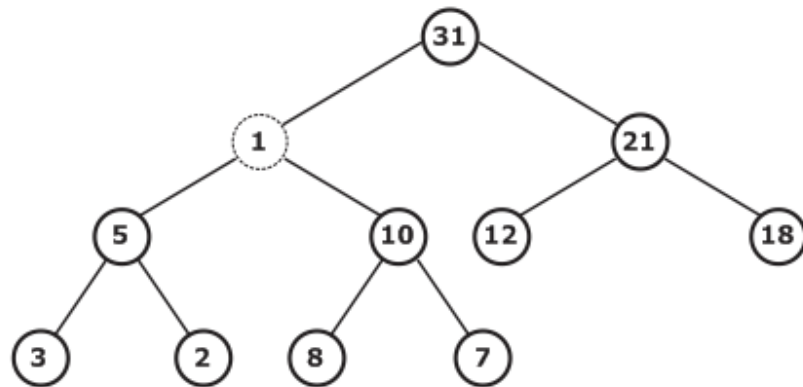
- Parent is at the floor $(i-1)/2$ index.
- Left child is at $2 * i + 1$ index.
- Right child is at $2 * i + 2$ index.

In complete binary trees, each level is filled up before another level is added and the levels are filled from left to right.

1.4.2. Heapifying an element:

After inserting an element into heap, it may not form a heap as it may not satisfy the heap property. In that case, adjust the location of the heap to make it heap again. This process is called heapifying.

In max-heap, to heapify an element, find the maximum of its children and swap it with the current element and continue this process until the heap property is satisfied at every node.

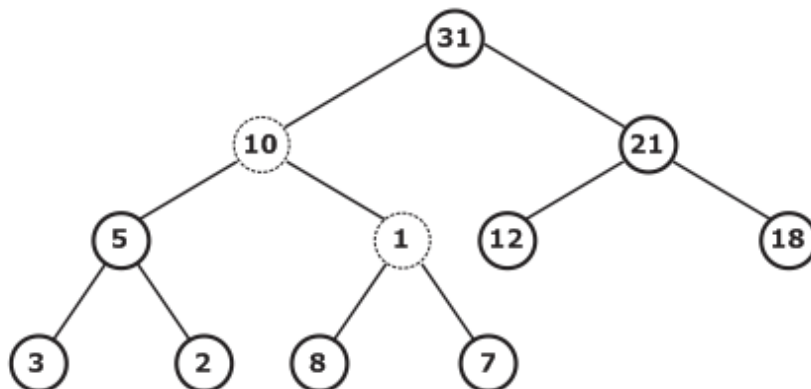


One important property of heap is that, if an element is not satisfying the heap property then all the elements from that element to root will also have the same problem.

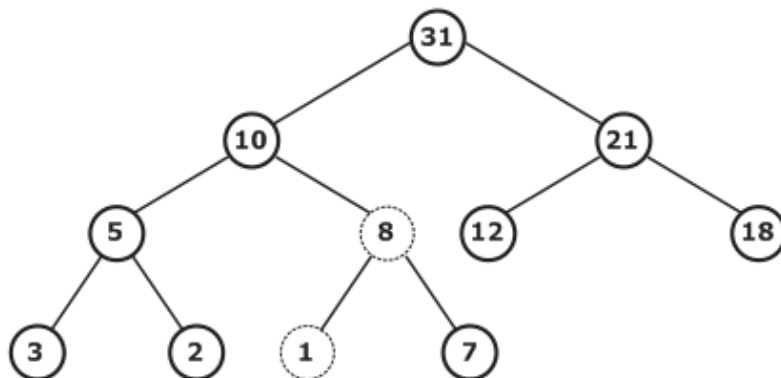
In below example, element 1 is not satisfying the heap property and its parent 31 is also having the issue. Similarly, if we heapify an element then all the elements from that element to root will also satisfy the heap property automatically.

In the above heap, the element 1 is not satisfying the heap property so, heapifying this element.

To heapify 1, find the maximum of its children and swap with that.



Continue this process until the element satisfies the heap properties. Now, swap 1 with 8.



Now the tree is satisfying the heap property.

1.4.3. Inserting an Element:

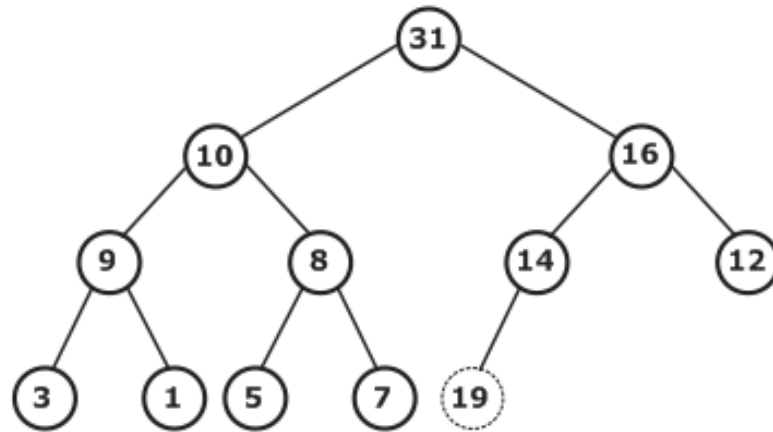
Insertion of an element is very much similar to the heapify and deletion process.

Increase the heap size

Keep the new element at the end of the heap (tree)

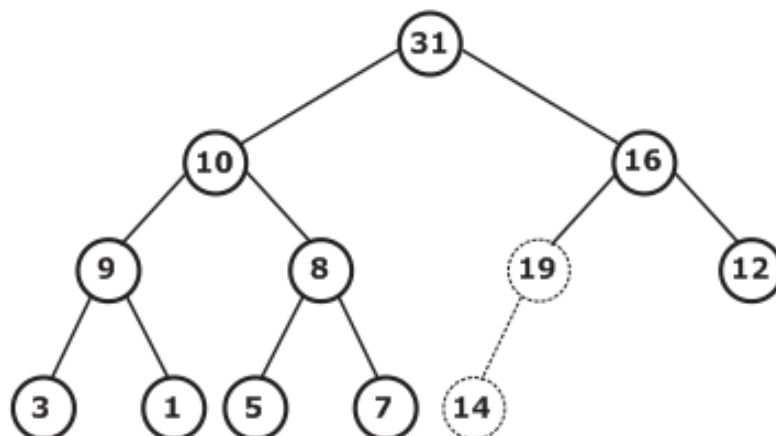
Heapify the element from bottom to top (root)

Here inserting the element 19 at the end of the heap, the heap property is not satisfied.

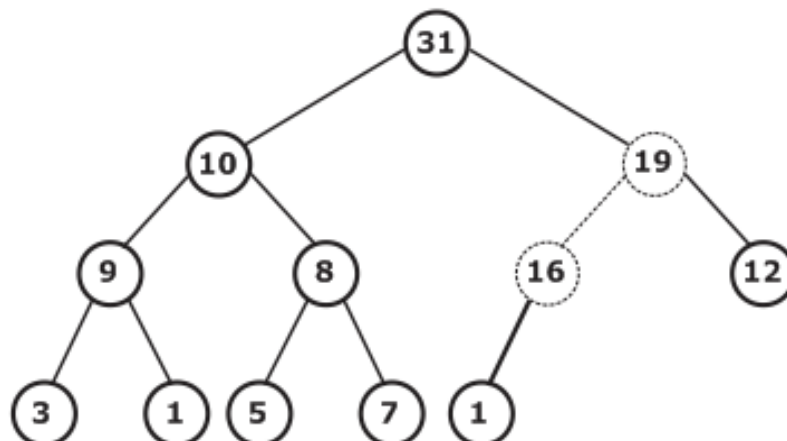


In-order to heapify this element (19), compare it with its parent and adjust them.

Swapping 19 and 14 gives :



Again swap 19 and 16 :



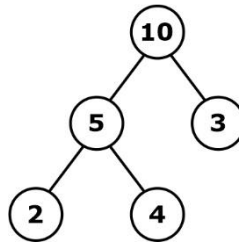
Now the tree is satisfying the heap property.

1.4.4. Deleting an Element:

Deleting an element at any intermediary position in the heap can be costly, so replace the element to be deleted by the last element and delete the last element of the Heap.

- Replace the root or element to be deleted by the last element.
- Delete the last element from the Heap.
- Since, the last element is placed at the position of the root node. So, it may not follow the heap property. Therefore, **heapify** the last node placed at the position of root.

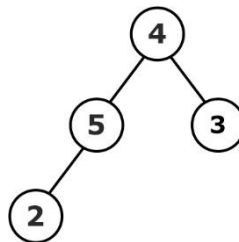
Example 7 : Suppose the Heap is a Max-Heap as:



The element to be deleted is root, i.e. 10.

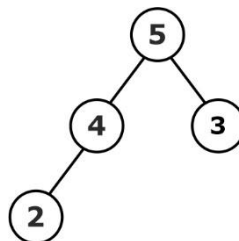
Sol: Here, the last element is 4.

Step 1: Replace the last element with root, and delete it.



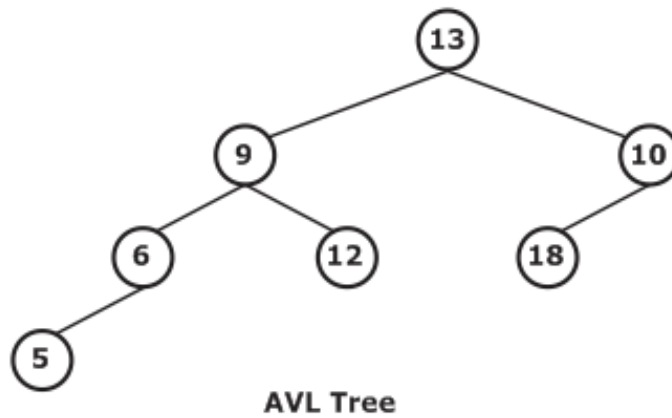
Step 2: Heapify root.

Final Heap:



1.5. AVL TREE

- AVL tree is a height balanced tree.
- It is a self-balancing binary search tree.
- AVL tree is another balanced binary search tree.
- AVL trees have a faster retrieval.
- In AVL tree, heights of the left and right subtree cannot be more than one for all nodes.
- To find out the balanced factor:



- The above tree is an AVL tree because the difference between heights of left and right subtrees for every node is less than or equal to 1.
- The above tree is not AVL because the difference between heights of left and right subtrees for 9 and 19 is greater than 1.
- It checks the height of the left and right subtree and assures that the difference is not more than 1. The difference is called the balance factor.
- Time Complexities:
Insertion: $O(\log n)$
Deletion: $O(\log n)$

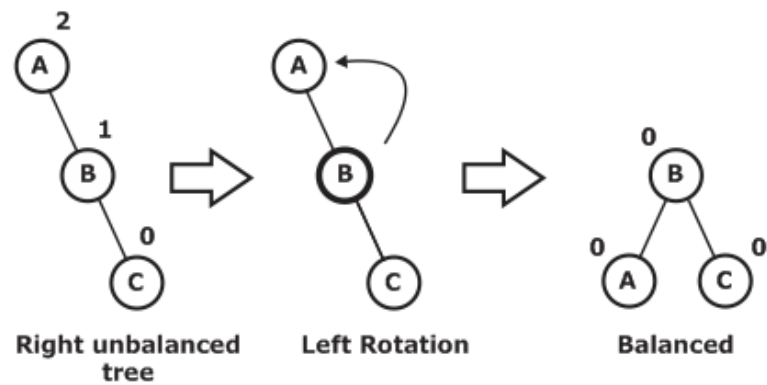
1.5.1. Operations on AVL:

a. Insertion:

- Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. The new node is added into the AVL tree as the leaf node. 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. Rotation is required only if the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required.
- Depending upon the type of insertion, the Rotations are categorized into four categories

i. Left Rotation

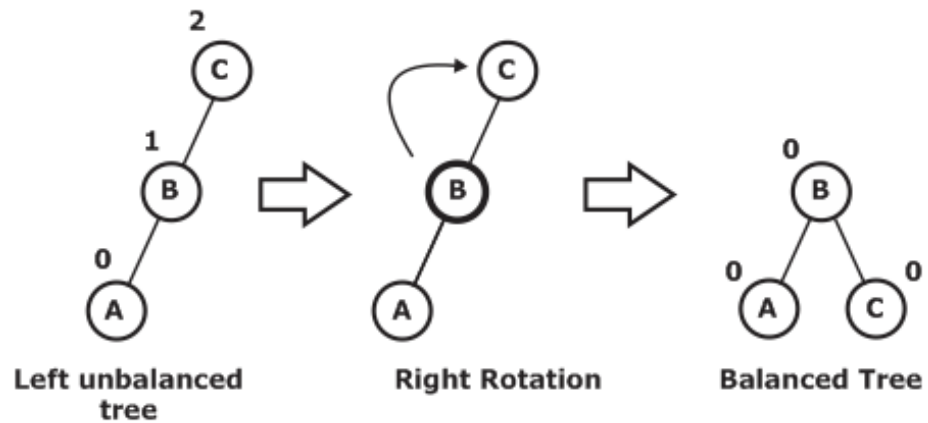
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



Here, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. Hence, left rotation is performed by making **A** the left-subtree of B.

ii.Right Rotation


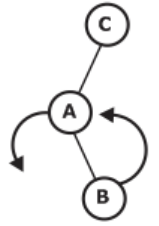
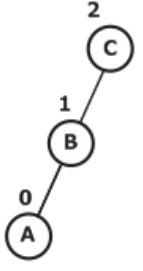
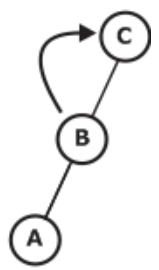
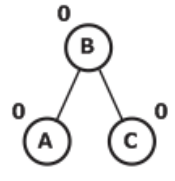
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



Here, the unbalanced node becomes the right child of its left child by performing a right rotation.

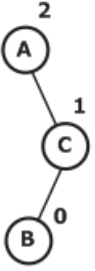
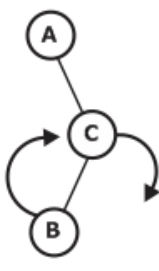
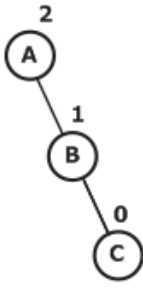
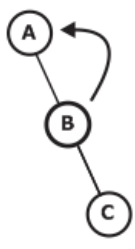
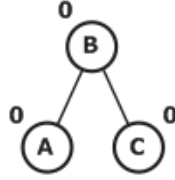
iii. Left-Right Rotation

A left-right rotation is a combination of left rotation followed by right rotation.

State	Action
	<p>A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.</p>
	<p>Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>Now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

iv. Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

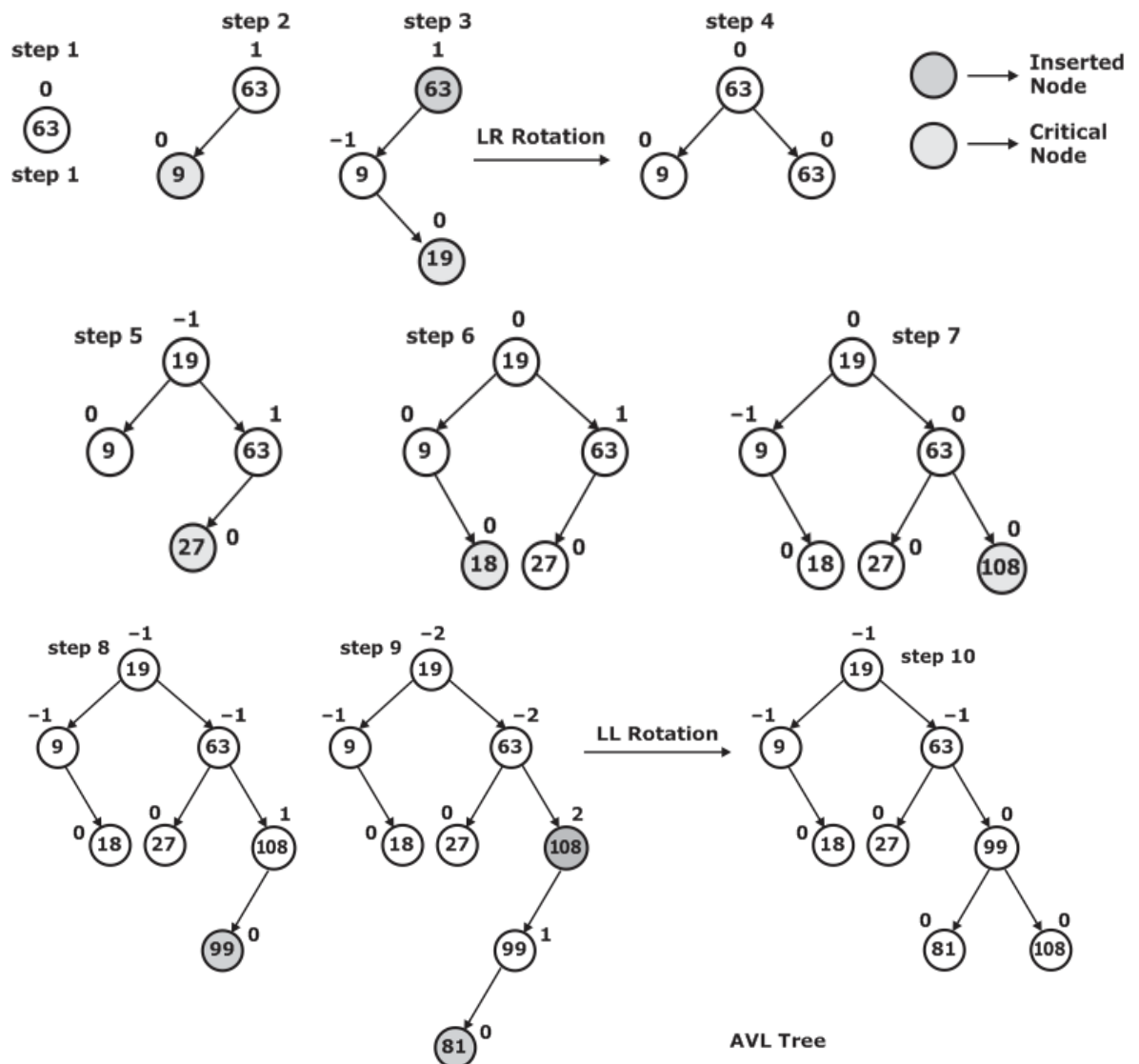
State	Action
	A node has been inserted into the left subtree of the right subtree. This makes A , an unbalanced node with balance factor 2.
	First, perform the right rotation along C node, making C the right subtree of its own left subtree B . Now, B becomes the right subtree of A .
	Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B .
	The tree is now balanced.

Example 8 : Construct an AVL tree by inserting the following elements in the given order.

63, 9, 19, 27, 18, 108, 99, 81

- At each step, the balance factor for every node is calculated.
- If it is found to be other than -1, 0, 1 then we need a rotation to rebalance the tree.
- The type of rotation will be estimated by the location of the inserted element with respect to the critical node.

All the elements are inserted in order to maintain the order of the binary search tree.



b. Deletion:

Delete(T, k) means delete a node with the key k from the AVL tree T .

- I) Find the node x where k is stored
- II) Delete the contents of node x

There are three possible cases:

- 1) If x has no children (i.e., is a leaf), delete x .
- 2) If x has one child, let x' be the child of x .

Notice: x' cannot have a child, since subtrees of T can differ in height by at most one

- replace the contents of x with the contents of x'
- delete x' (a leaf)

3) If x has two children,

- find x 's successor z (which has no left child)
- replace x 's contents with z 's contents, and
- delete z .

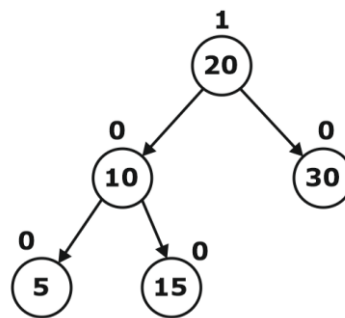
[since z has at most one child, so we use case (1) or (2) to delete z]

III) Third: Go from the deleted leaf towards the

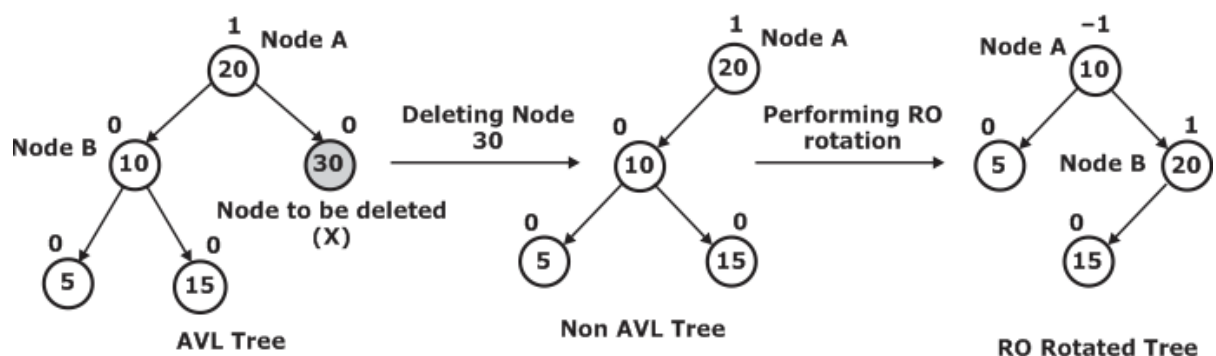
root and at each ancestor of that leaf:

- update the balance factor
- rebalance with rotations if necessary.

Example 9 : Delete the node 30 from the AVL tree shown in the following image.

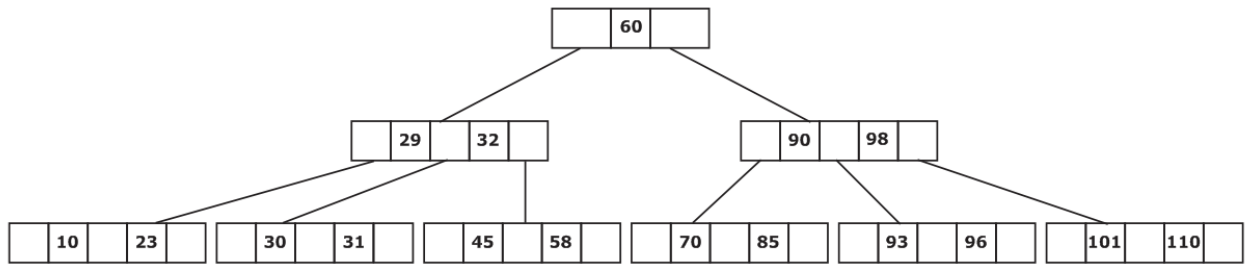


Sol: In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



1.6. B- TREE

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



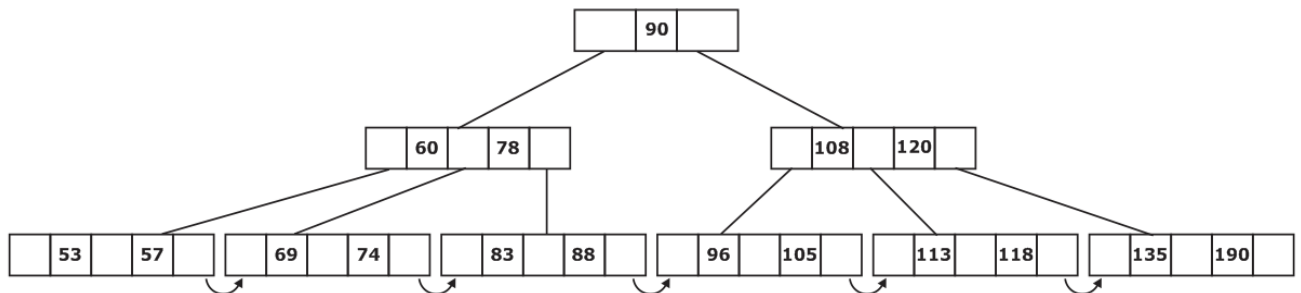
- Time Complexities:

Insertion: $O(\log n)$

Deletion: $O(\log n)$

1.7. B⁺ TREE

- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.
- The leaf nodes of a B+ tree are linked together in the form of singly linked lists to make the search queries more efficient.



- Time Complexities:

Insertion: $O(\log n)$

Deletion: $O(\log n)$

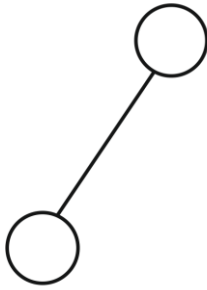
Example 10 : Construct minimal AVL trees of height 0, 1, 2, 3, and 4. What is the number of nodes in a minimal AVL tree of height 5?

Sol. Let $n(h)$ be the number of nodes in a minimal AVL tree with height h .

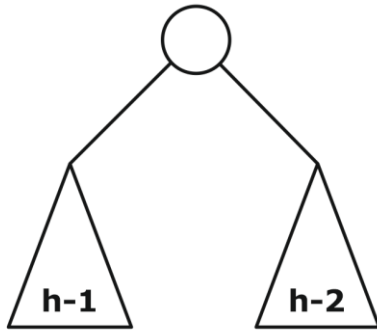
$$N(0) = 1$$



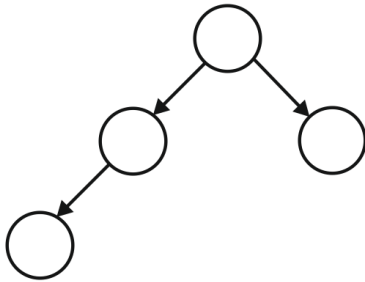
$$N(1) = 2$$



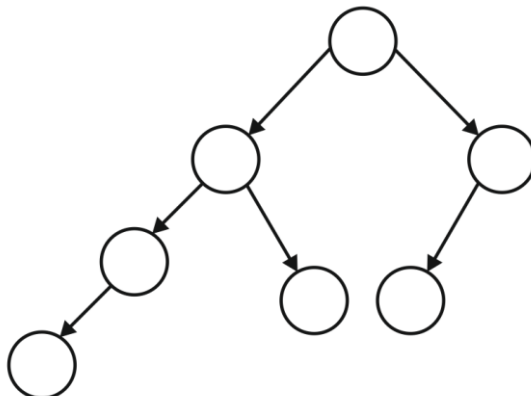
$$N(h) = 1 + N(h - 1) + N(h - 2)$$



$$\begin{aligned} N(2) &= 1 + N(1) + N(0) \\ &= 1 + 2 + 1 = 4 \end{aligned}$$



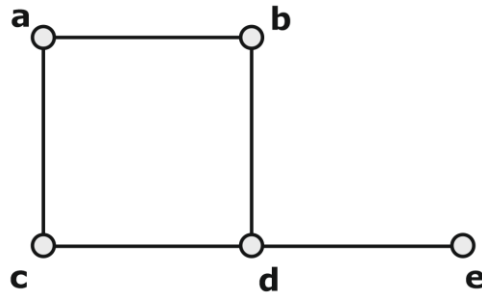
$$\begin{aligned} N(3) &= 1 + N(2) + n(1) \\ &= 1 + 4 + 2 = 7 \end{aligned}$$



$$\begin{aligned} N(4) &= 1 + N(3) + N(2) \\ &= 1 + 7 + 4 = 12 \end{aligned}$$

2. Graphs

- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.
Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

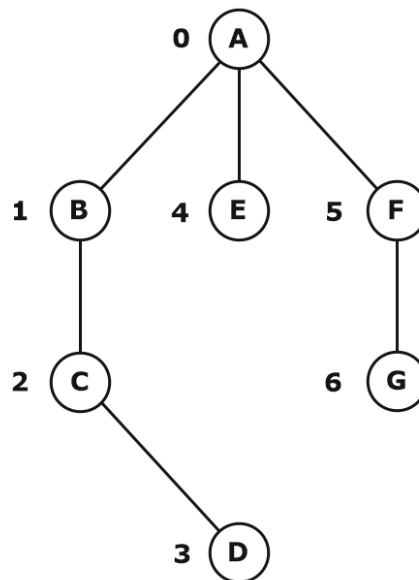
$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- Vertex** – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on, represent edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph –

- **Add Vertex** – Adds a vertex to the graph.
- **Add Edge** – Adds an edge between the two vertices of the graph.
- **Display Vertex** – Displays a vertex of the graph.

2.1 Types of Graphs

2.1.1 Null Graph

A graph having no edges is called a Null Graph.

Example

● a

●

b

●

c

In the above graph, there are three vertices named 'a', 'b', and 'c', but there are no edges among them. Hence it is a Null Graph.

2.1.2. Trivial Graph

A graph with only one vertex is called a Trivial Graph.

Example

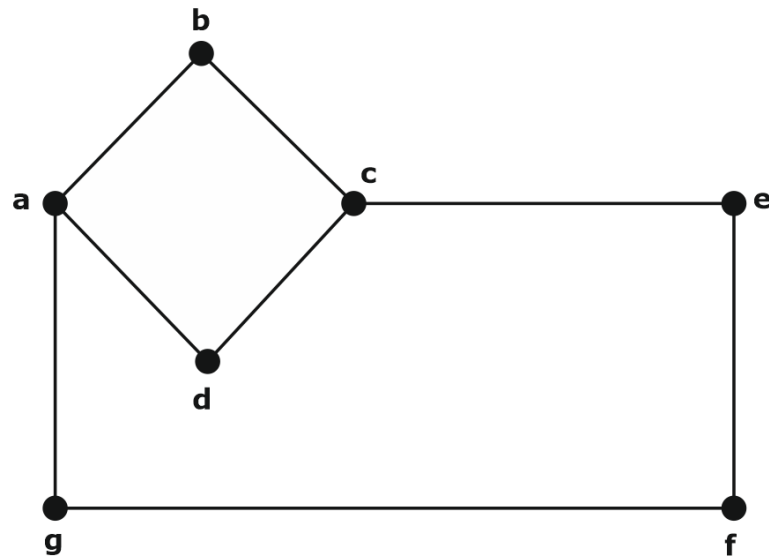
● a

In the above shown graph, there is only one vertex 'a' with no other edges. Hence it is a Trivial graph.

2.1.3 Non-Directed Graph

A non-directed graph contains edges but the edges are not directed ones.

Example

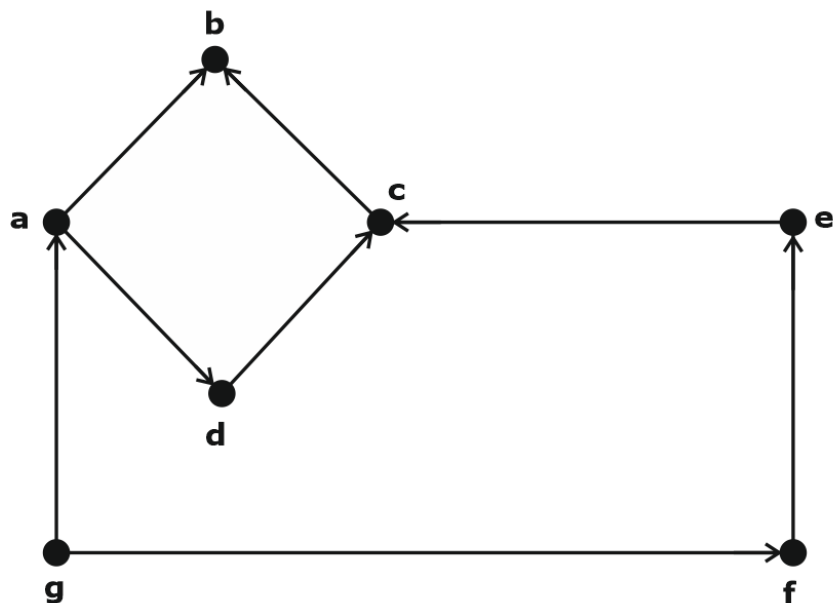


In this graph, 'a', 'b', 'c', 'd', 'e', 'f', 'g' are the vertices, and 'ab', 'bc', 'cd', 'da', 'ag', 'gf', 'ef' are the edges of the graph. Since it is a non-directed graph, the edges 'ab' and 'ba' are the same. Similarly other edges are also considered in the same way.

2.1.4 Directed Graph

In a directed graph, each edge has a direction.

Example



In the above graph, we have seven vertices 'a', 'b', 'c', 'd', 'e', 'f', and 'g', and eight edges 'ab', 'cb', 'dc', 'ad', 'ec', 'fe', 'gf', and 'ga'. As it is a directed graph, each edge bears an arrow mark that shows its direction. Note that in a directed graph, 'ab' is different from 'ba'.

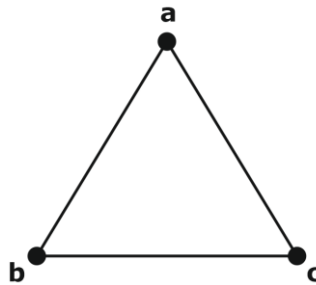
2.1.5 Simple Graph

A graph with no loops and no parallel edges is called a simple graph.

- The maximum number of edges possible in a single graph with 'n' vertices is nC_2 where ${}^nC_2 = n(n-1)/2$.
- The number of simple graphs possible with 'n' vertices = $2^{{}^nC_2} = 2^{n(n-1)/2}$.

Example

In the following graph, there are 3 vertices with 3 edges which is maximum excluding the parallel edges and loops. This can be proved by using the above formulae.



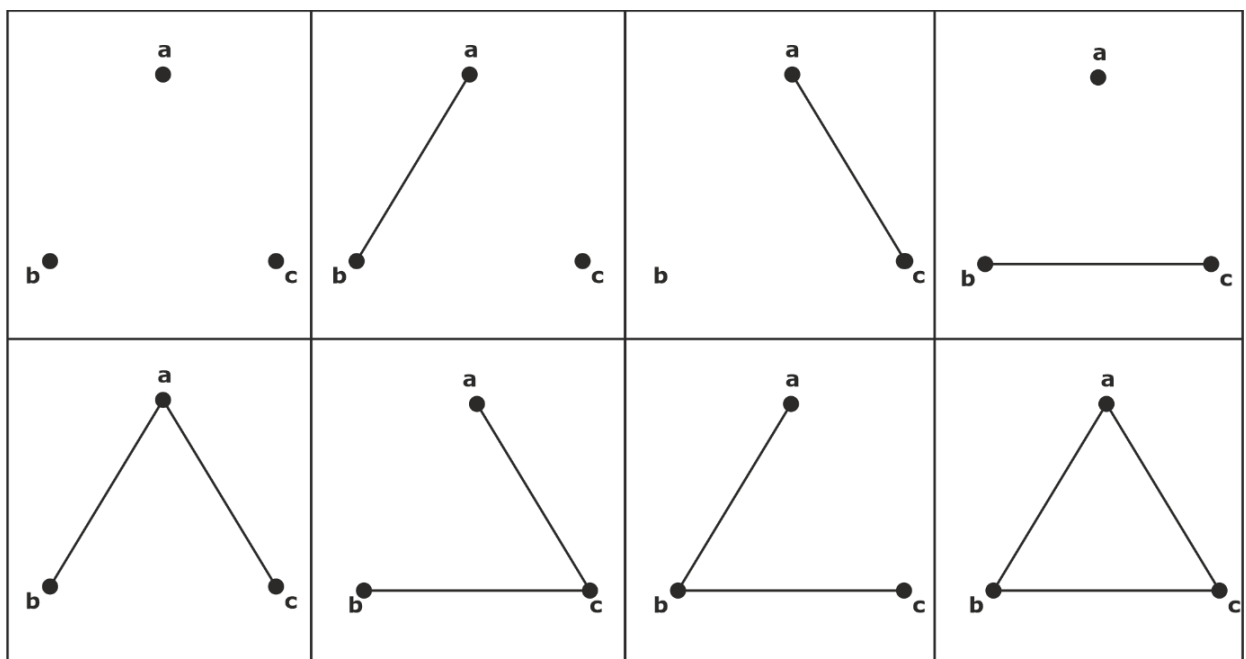
The maximum number of edges with $n=3$ vertices –

$$\begin{aligned} {}^nC_2 &= n(n-1)/2 \\ &= 3(3-1)/2 \\ &= 6/2 \\ &= 3 \text{ edges} \end{aligned}$$

The maximum number of simple graphs with $n=3$ vertices –

$$\begin{aligned} 2^{{}^nC_2} &= 2^{n(n-1)/2} \\ &= 2^{3(3-1)/2} \\ &= 2^3 \\ &= 8 \end{aligned}$$

These 8 graphs are as shown below –

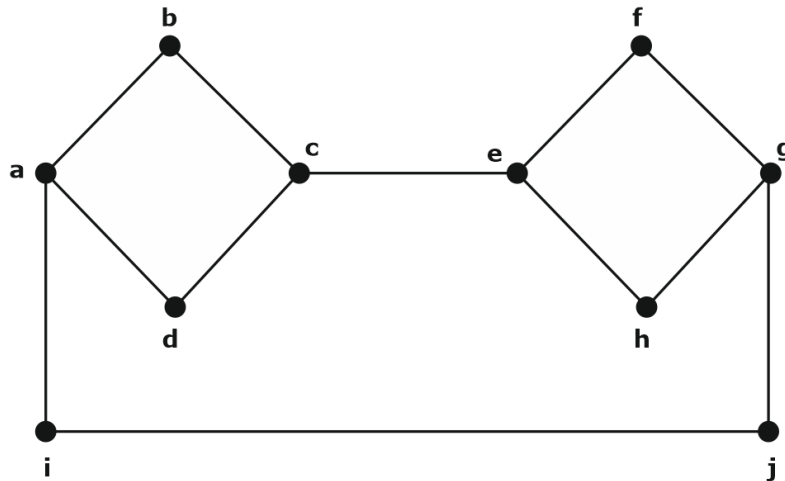


2.1.6 Connected Graph

A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

Example

In the following graph, each vertex has its own edge connected to another edge. Hence it is a connected graph.

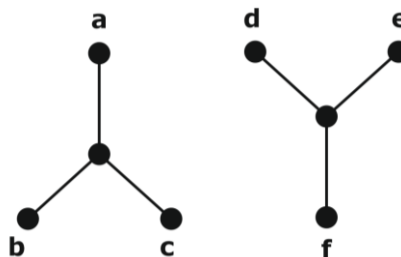


2.1.7 Disconnected Graph

A graph G is disconnected, if it does not contain at least two connected vertices.

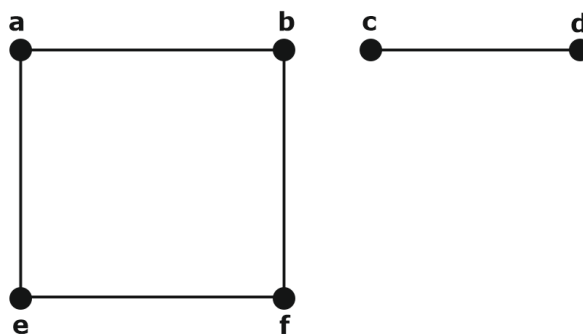
Example 1

The following graph is an example of a Disconnected Graph, where there are two components, one with 'a', 'b', 'c', 'd' vertices and another with 'e', 'f', 'g', 'h' vertices.



The two components are independent and not connected to each other. Hence it is called disconnected graph.

Example 2



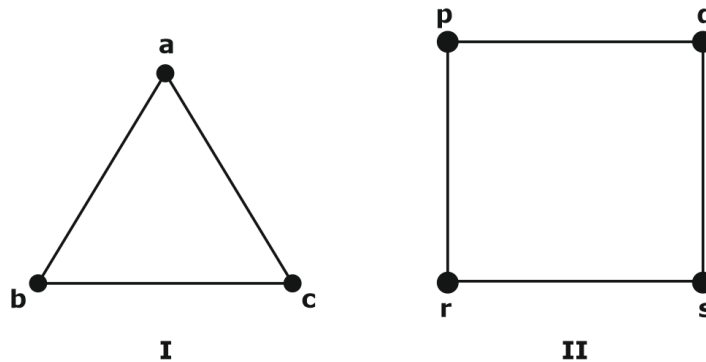
In this example, there are two independent components, a-b-f-e and c-d, which are not connected to each other. Hence this is a disconnected graph.

2.1.8 Regular Graph

A graph G is said to be regular, **if all its vertices have the same degree**. In a graph, if the degree of each vertex is ' k ', then the graph is called a ' k -regular graph'.

Example

In the following graphs, all the vertices have the same degree. So these graphs are called regular graphs.



In both the graphs, all the vertices have degree 2. They are called 2-Regular Graphs.

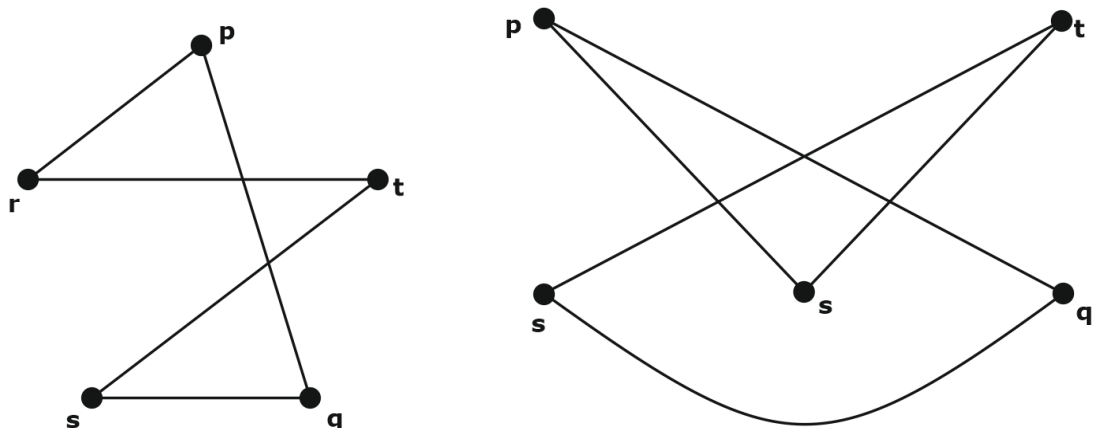
Complete Graph

A simple graph with ' n ' mutual vertices is called a complete graph and it is **denoted by ' K_n '**. In the graph, **a vertex should have edges with all other vertices**, then it is called a complete graph.

In other words, if a vertex is connected to all other vertices in a graph, then it is called a complete graph.

Example

In the following graphs, each vertex in the graph is connected with all the remaining vertices in the graph except by itself.



2.1.9. Cycle Graph

A simple graph with ' n ' vertices ($n \geq 3$) and ' n ' edges is called a cycle graph if all its edges form a cycle of length ' n '.

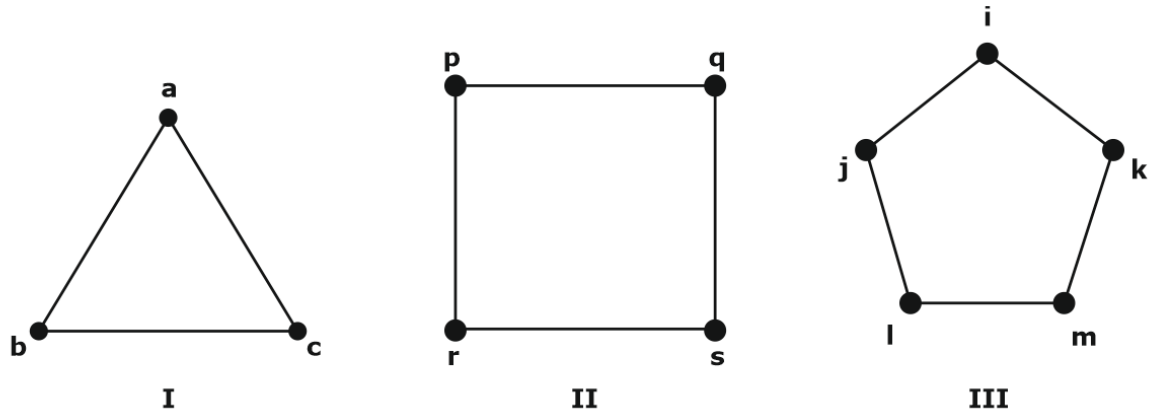
If the degree of each vertex in the graph is two, then it is called a Cycle Graph.

Notation – C_n

Example

Take a look at the following graphs –

- Graph I has 3 vertices with 3 edges which form a cycle 'ab-bc-ca'.
- Graph II has 4 vertices with 4 edges which form a cycle 'pq-qs-sr-rp'.
- Graph III has 5 vertices with 5 edges which form a cycle 'ik-km-ml-lj-ji'.



Hence all the given graphs are cycle graphs.

3. Graph Traversal

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

3.1. DFS (Depth First Search)

3.2. BFS (Breadth First Search)

3.1. DFS (Depth First Search)

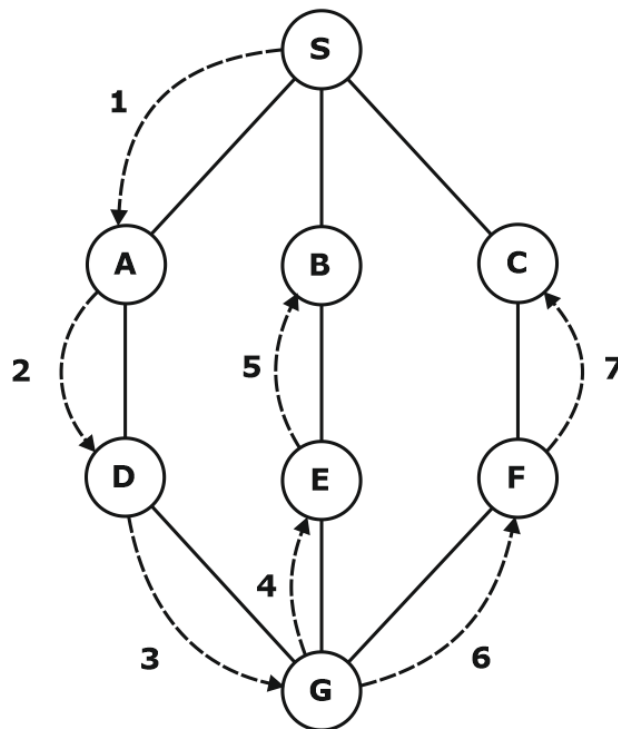
DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

- **Step 1** - Define a Stack of size total number of vertices in the graph.
- **Step 2** - Select any vertex as starting point for traversal. Visit that vertex and push it onto the Stack.
- **Step 3** - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it onto the stack.
- **Step 4** - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

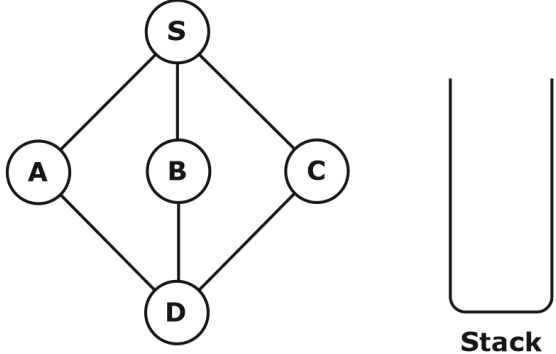
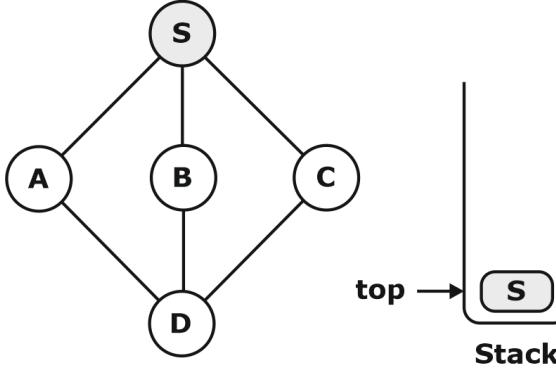
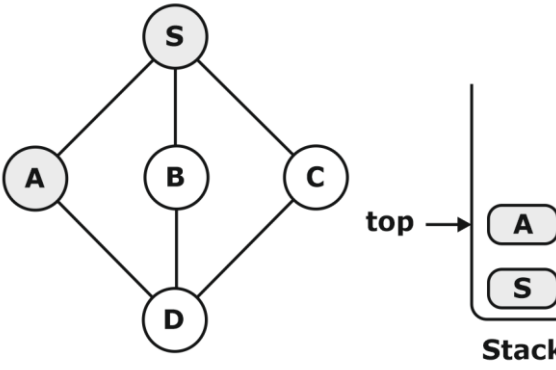
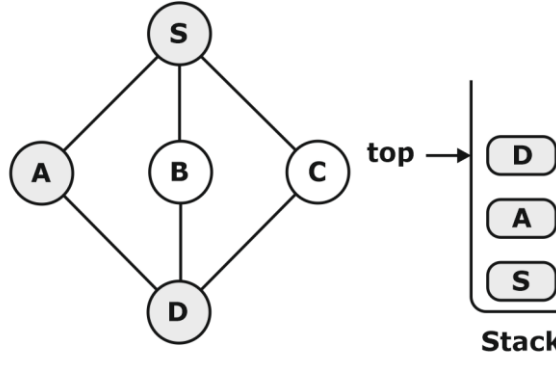
- **Step 5** - When there is no new vertex to visit then use backtracking and pop one vertex from the stack.
- **Step 6** - Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7** - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.

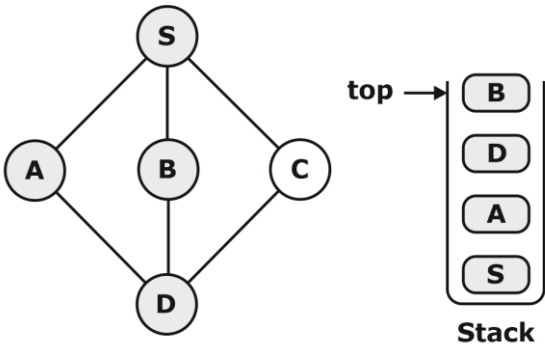
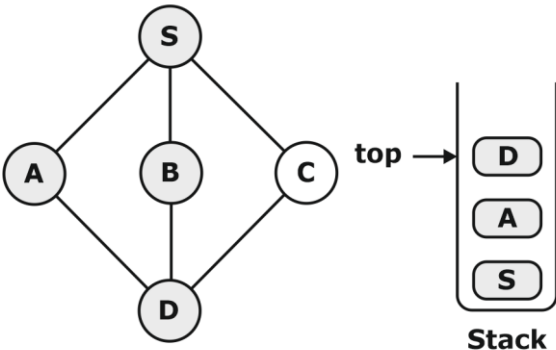
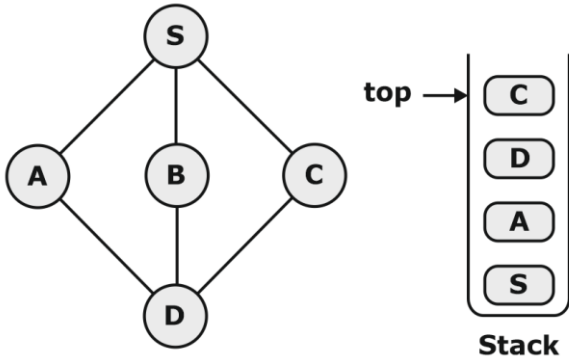
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, the DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

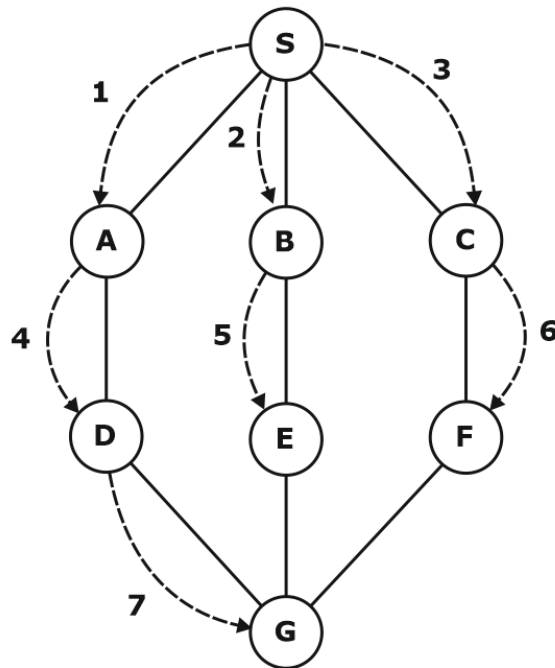
Step	Traversal	Description
1		Initialize the stack.
2		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.
4		Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.

Step	Traversal	Description
5		We choose B , mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6		We check the stack top to return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.
7		Only unvisited adjacent node is from D is C now. So we visit C , mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

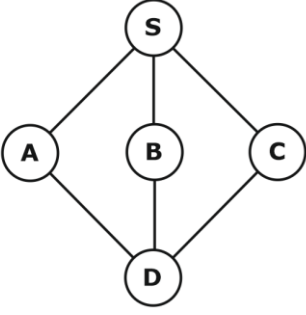
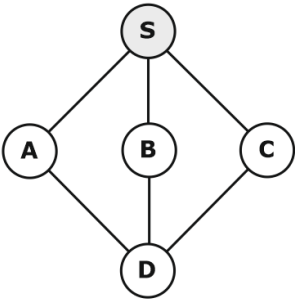
3.2. Breadth first Search

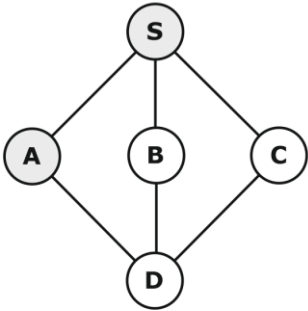
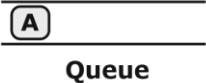
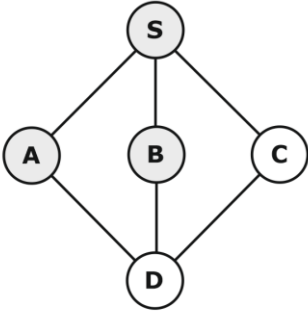
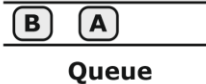
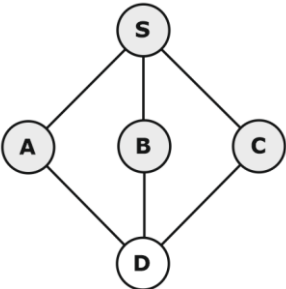
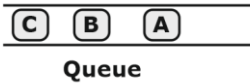
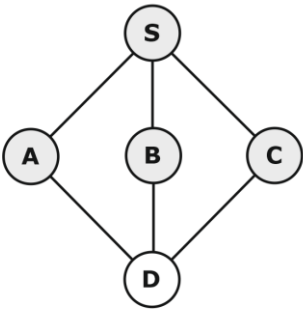
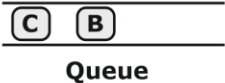
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

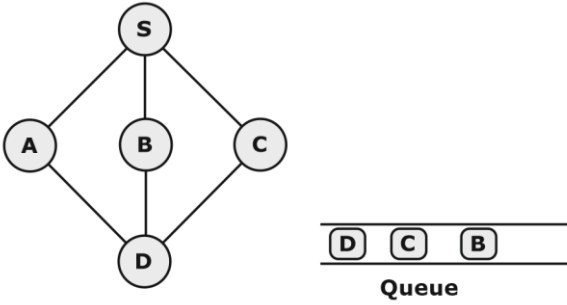


As in the example given above, the BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1	 <div style="display: flex; align-items: center; margin-left: 20px;"> <div style="border-bottom: 1px solid black; width: 100px; margin-bottom: 2px;"></div> <div style="border-bottom: 1px solid black; width: 100px; margin-bottom: 2px;"></div> <div style="margin-left: 10px;">Queue</div> </div>	Initialize the queue.
2	 <div style="display: flex; align-items: center; margin-left: 20px;"> <div style="border-bottom: 1px solid black; width: 100px; margin-bottom: 2px;"></div> <div style="border-bottom: 1px solid black; width: 100px; margin-bottom: 2px;"></div> <div style="margin-left: 10px;">Queue</div> </div>	We start from visiting S (starting node), and mark it as visited.

Step	Traversal	Description
3	 	We then see an unvisited adjacent node from S . In this example, we have three nodes but alphabetically we choose A , mark it as visited and enqueue it.
4	 	Next, the unvisited adjacent node from S is B . We mark it as visited and enqueue it.
5	 	Next, the unvisited adjacent node from S is C . We mark it as visited and enqueue it.
6	 	Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A .

Step	Traversal	Description
7		From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

3.3. Difference between DFT and BFT

BFT	DFT
BFS stands for Breadth First Search.	DFS stands for Depth First Search.
BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
BFS can be used to find a single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from source.
BFS considers all neighbours first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to a win-win situation, we stop.
The Time complexity of BFS is $O(V + E)$ when Adjacency List is used and $O(V^2)$ when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also $O(V + E)$ when Adjacency List is used and $O(V^2)$ when Adjacency Matrix is used, where V stands for vertices and E stands for edges.
