



SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

AN AUTONOMOUS INSTITUTION

Accredited by NBA–AICTE and Accredited by NAAC–UGC with 'A' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai



DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

COURSE NAME : 19CS307- DATA STRUCTURES

II YEAR / III SEMESTER

Unit III

NON LINEAR DATA STRUCTURES

TREES

A tree is a hierarchical data structure, which has one root node and multiple child nodes (branches). Unlike, Arrays, Linked Lists, Stacks or queues, a Tree is non linear data structure.

Nodes are the basic building blocks of a tree structure that store some data/value.

Why Trees?

One reason could be that we want to store information that follows, natural hierarchy, like how we store folders in a computer system.

Components

- 1 Root
- 2 Parent
- 3 Child nodes
- 4 Siblings
- 5 Leaves
- 6 Branch
- 7 Sub-Tree

- 8 Ancestor
- 9 Descendants
- 10 Null Nodes

Terminologies

- 1 Degree
- 2 Edges
- 3 Path
- 4 Depth
- 5 Level
- 6 Height

1. Root

The topmost node of a tree is known as the root.

There exists only one root node per tree.

Taking the image above as reference, node 'a' is the root of the tree as shown here.

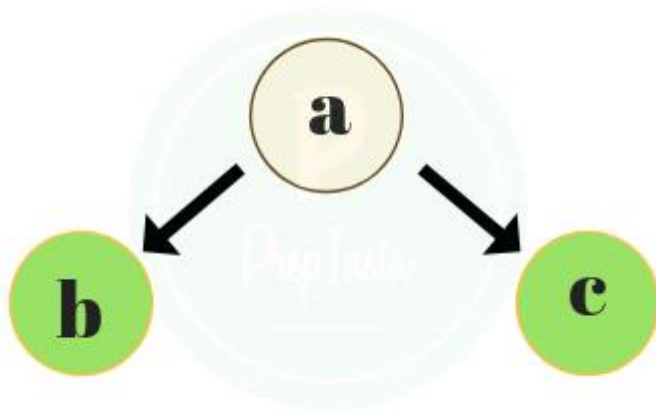


2. Parent

Any node which has an edge directed downwards to the child node is known as parent node.

Each parent can have one or more child node.

In the given image we can see, the node 'a' is the parent of the nodes 'b' and 'c'.

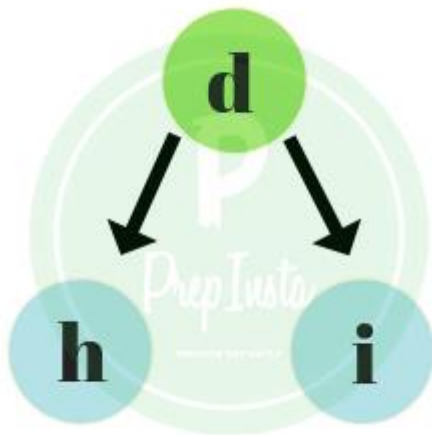


3. Child Node

Any node which has an edge directed upwards to the parent node is known as child node.

Each child node has a single parent node.

In the given image we can see, the nodes 'h' and 'i' are the child nodes of the node 'd'.



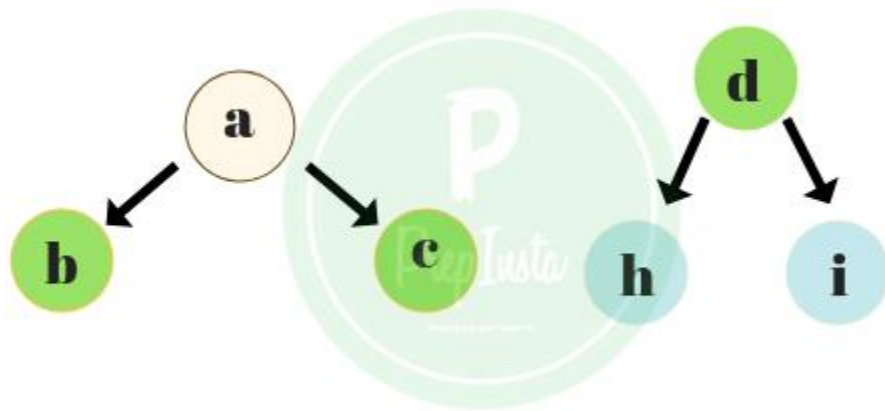
4. Sibling

A set of nodes that are extended from the same parent are known as the siblings.

In the given image, we can see that

the nodes 'b' and 'c' are sibling nodes.

the nodes 'h' and 'i' are sibling nodes.

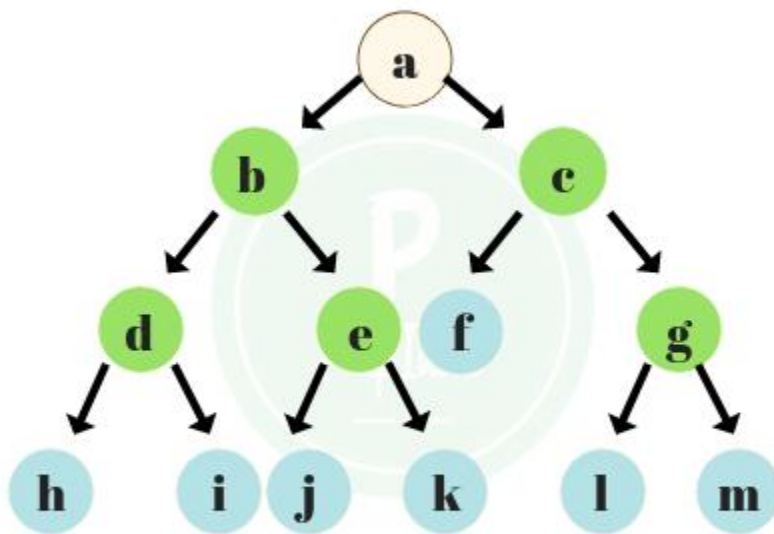


5. Leaf(external nodes)

Any node that does not have any child node is known as the leaf node.

In the given image, the nodes 'f', 'h', 'i', 'j', 'k', 'l' and 'm' are leaf nodes since these nodes are terminal and have no further child nodes.

Total Number of leaf nodes in a Binary Tree = Total Number of nodes with 2 children + 1

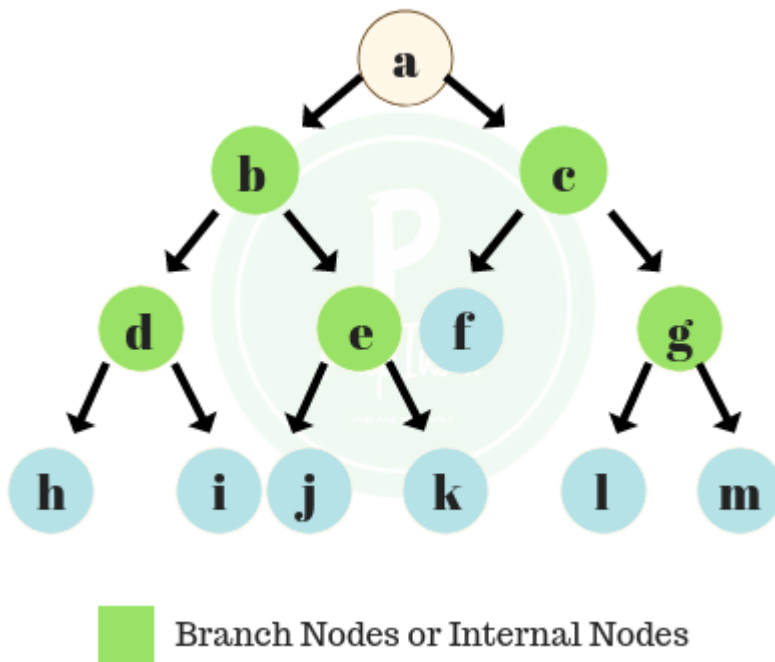


 Leaf Nodes or External Nodes

6. Branch (internal nodes)

Any node which has at least one child node is known as branch node.

In the given image, the nodes 'b', 'c', 'd', 'e', 'g' are branch nodes since each of these nodes extend further to their respective children.

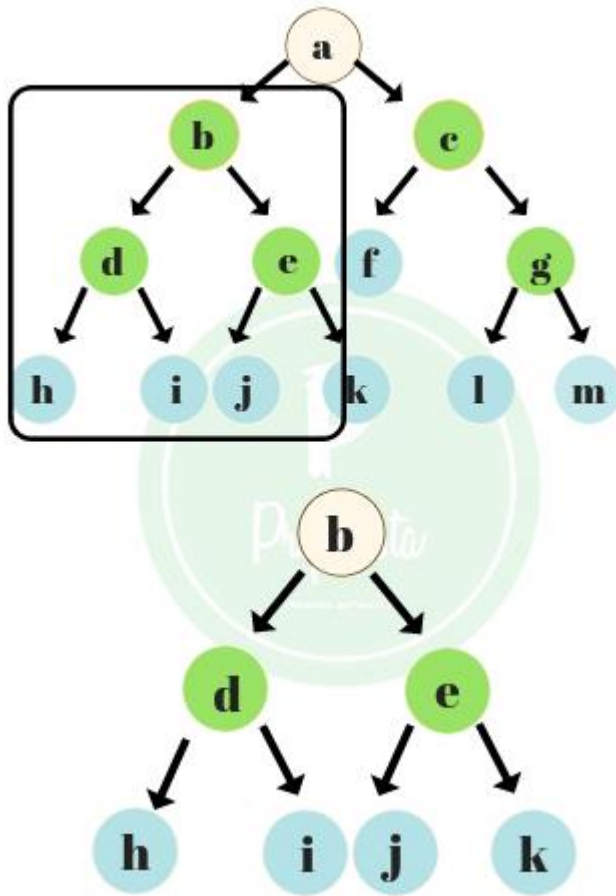


7. Sub-Tree

A sub tree of a tree is defined as a tree that consists of a node along with all its descendants.

In the image, we can see that a sub tree can be extended from node 'b' which will be termed as the left sub tree.

Similarly, a sub tree can be extended from the node 'c' which will be termed as the right sub tree.

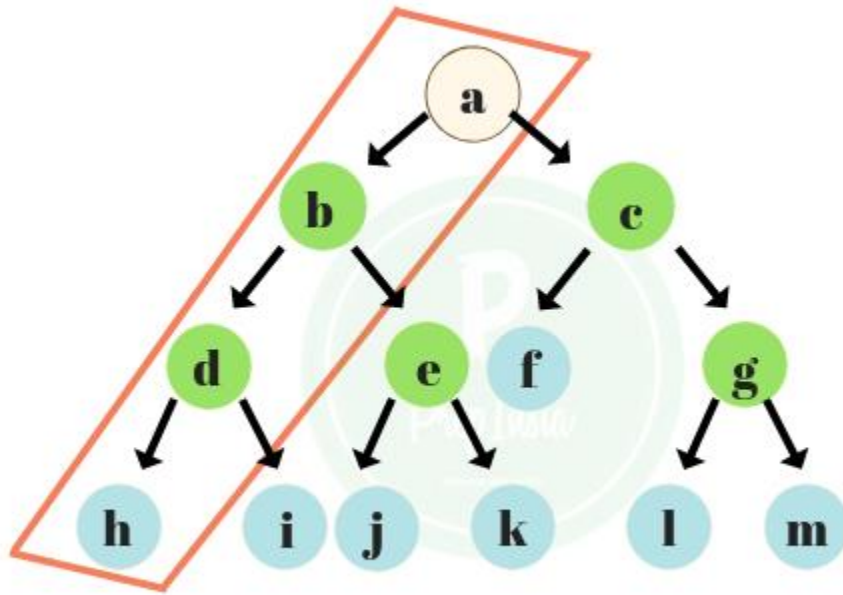


8. Ancestor

Any predecessor of a node along with all the ancestors of the predecessor of that node is known as the ancestor.

The root node has no ancestors.

In the given image, the ancestors of the node 'h' will be 'd', 'b' and 'a'.

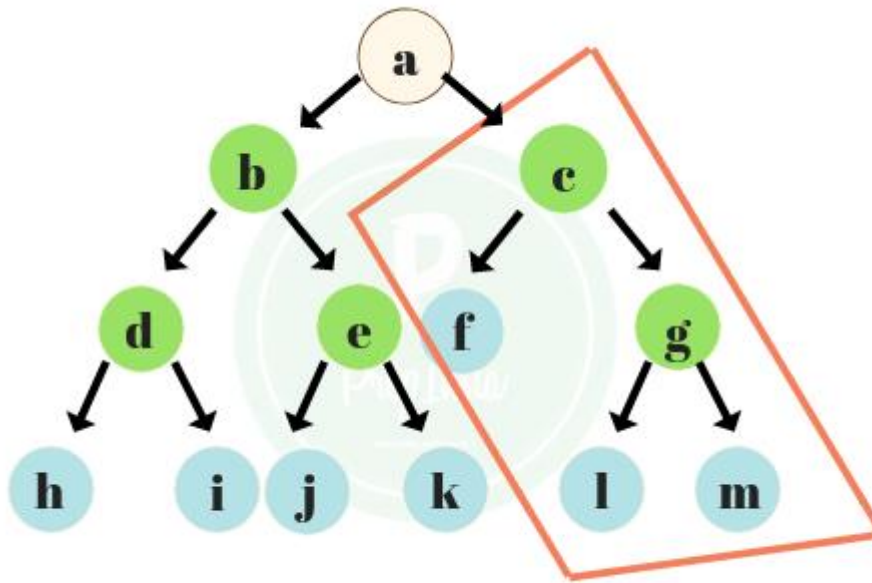


9. Descendant

All the children of a node along with all the descendants of the children of a node is known as descendant.

A leaf node has no descendants.

In the given image, if we consider the node 'c', the descendants of node 'c' will be nodes 'f', 'g', 'l' and 'm'.

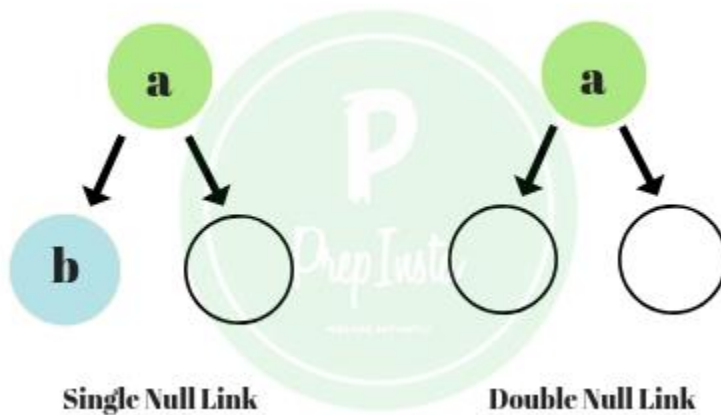


10. Null Nodes

If in a binary tree, a node has only one child it is said to have a single null link.

Similarly if a node has no child node it is said to have two null links.

We can see in the image given the two cases of null links.



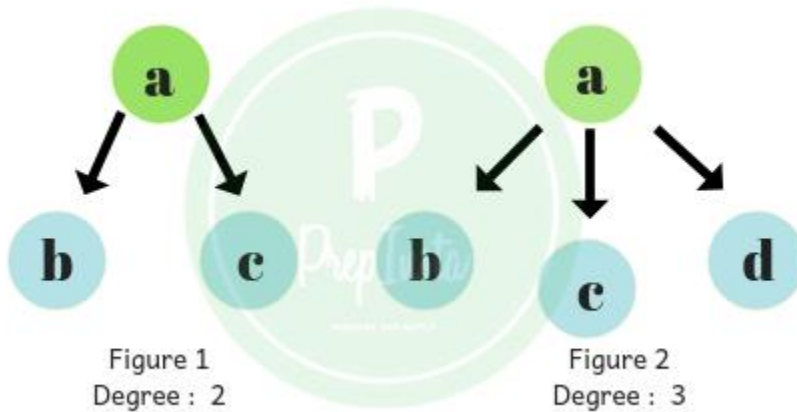
Terminologies

1. Degree.

The degree of a node can be defined as its number of sub trees.

A node with zero degree is a leaf node.

A node with maximum degree is the root node in the tree.



2. Edge

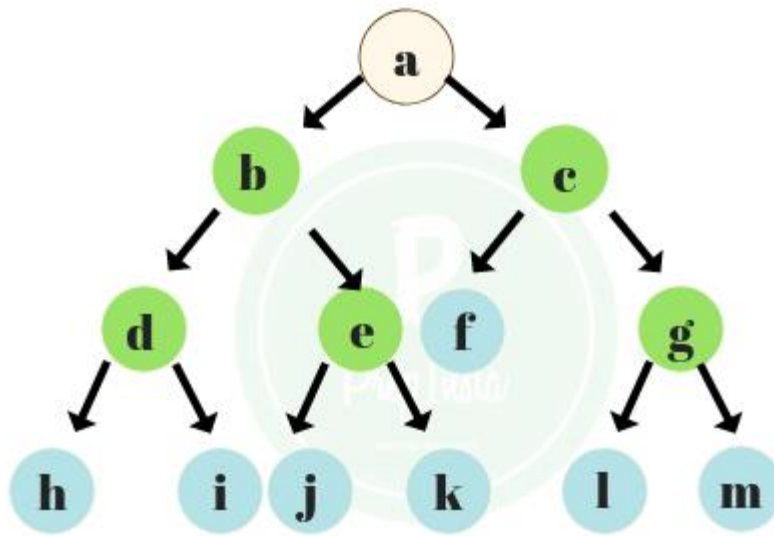
An edge can be defined as a connection or a link from one node to another node.

In the given image, we have 3 edges from node 'a' to node 'h'.

If we see the image, we can see clearly that we have a total of 13 nodes and 12 edges.

Thus, we can say that,

No. of edges = No. of nodes - 1



Tree has 13 nodes & 12 edges



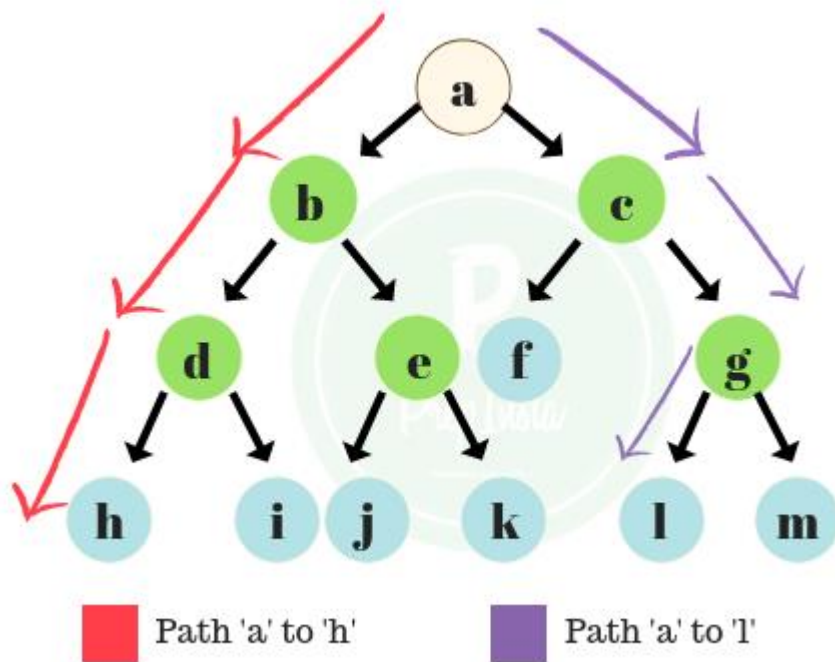
3. Path

A course of nodes and edges for operations such as traversal, etc is known as a path.

Let us see with the help of an example taking the image as reference,

The path from node 'a' to the node 'h' is represented in red which consist of 4 nodes and 3 edges.

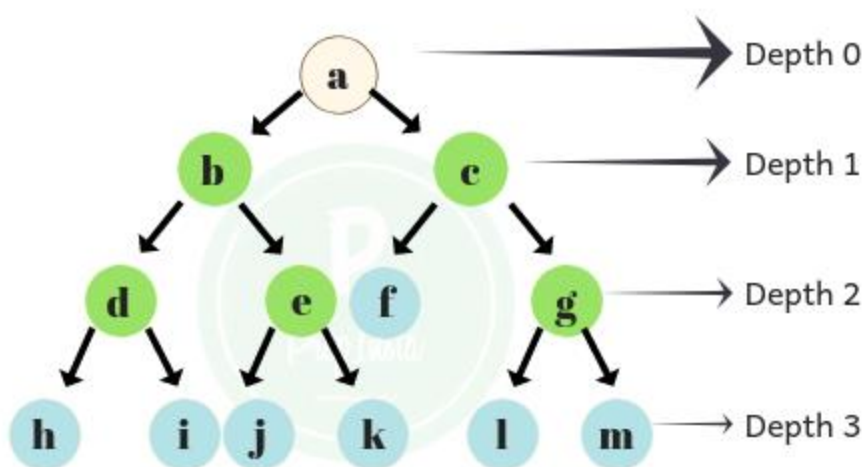
Similarly, the path from node 'a' to 'l' is represented in purple which consist of 4 nodes and 3 edges.



4. Depth

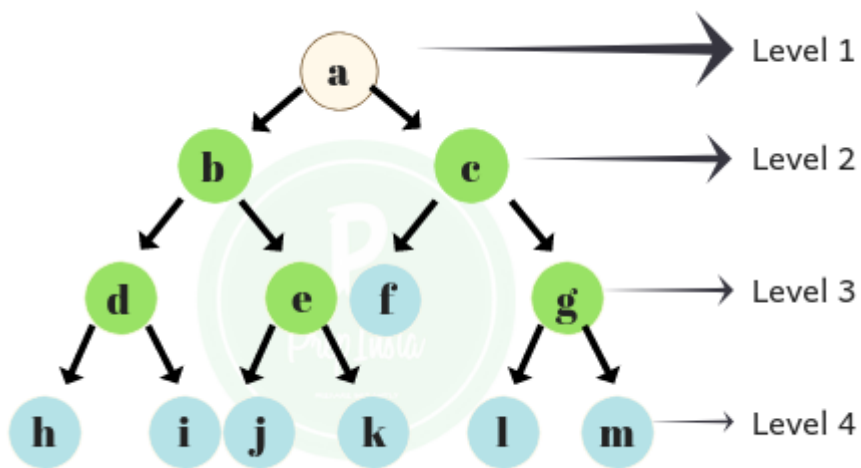
The number of edges that lie in between the path from a node to the root in a tree is defined as the depth of the tree.

In the image given here, we can see the depth of each node. For instance the depth of the root node is zero.



5. Level

- Level of a node is symbolic of the generation of a given node. It is one greater than the level of its parent.
- For instance, in the given image we can see, the level of the node 'b' and 'c' is one more than the level of their parent node 'a'.



Note – Formula Max number of nodes at any given level for the tree would be

1. 2^L (If level Starts from 0)
2. 2^{L-1} (If level Starts from 1)

Example (Considering level starts from 1) –

- Level 1 (Max nodes) $= 2^{1-1} = 1$
- Level 2 (Max nodes) $= 2^{2-1} = 2$
- Level 3 (Max nodes) $= 2^{3-1} = 4$
- Level 4 (Max nodes) $= 2^{4-1} = 8$

Thus max possible nodes in the tree for Level 4 would be

- $2^{1-1} + 2^{2-1} + 2^{3-1} + 2^{4-1} = 1 + 2 + 4 + 8 = 15$

This above can be generalised to a formula as :

Maximum number of nodes (Considering Level Starts from 1) =

- $1 + 2 + 4 + 8 + \dots + 2^{L-1} = (2^L - 1)$

The above formula will change to $(2^{L+1} - 1)$ (If level starts from 0)

Formula Max number of nodes **at any given level** for the tree would be:

1. 2^L (If level Starts from 0)
2. 2^{L-1} (If level Starts from 1)

Formula Max number of nodes **in the whole tree** would be:

1. $2^{L+1} - 1$ (If level Starts from 0)
2. $2^L - 1$ (If level Starts from 1)

6. Height

- Height of a node can be defined as the longest path downwards between the root node and a leaf.
- For example in the given image, we can see that the height of the node 'I' is 3; since the distance between it and the root node is 3.
- Height of a tree starts from 0

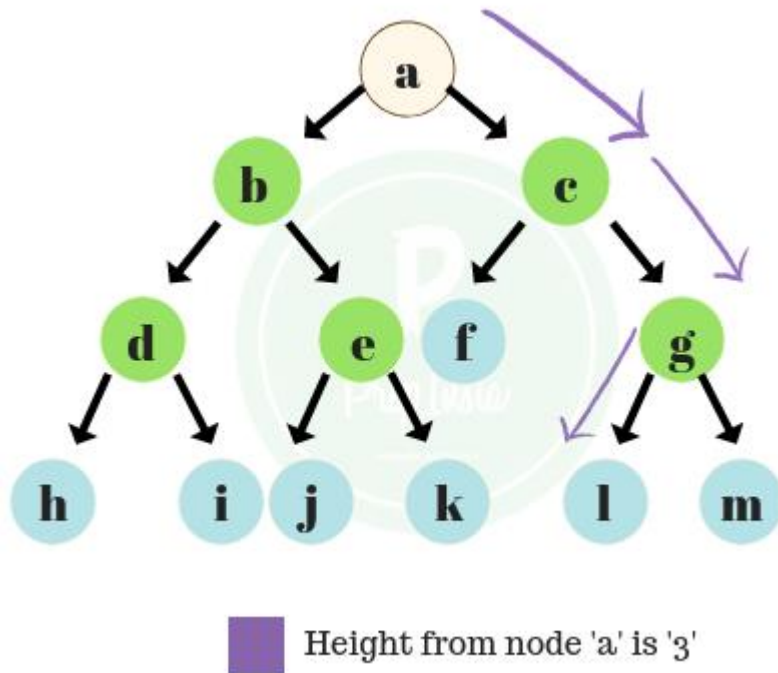
Formula –

Height of tree is h then Max nodes a [Full Binary Tree](#) will have

- $(2^{h+1} - 1)$ nodes (if h starts from 0)
- $(2^h - 1)$ nodes (if h starts from 1)

Formula – Minimum number of nodes in a Binary Tree of height h would be

- $(h + 1)$ (if h starts from 0)
- h (if h starts from 1)



Types of Trees

- Binary tree.
- Binary search tree
- AVL tree.
- B tree

Operations performed on a Tree

- Enumerating
- Traversing and Searching
- Insertion
- Deletion

Applications of a Tree

- It provides a simple and systematic method to store and represent the data in a hierarchical form.
- It stores the data/values in a way that provides ease of search and traversal.
- It executes the insertion/deletion operation within a moderate range of time.

Tree Traversal

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

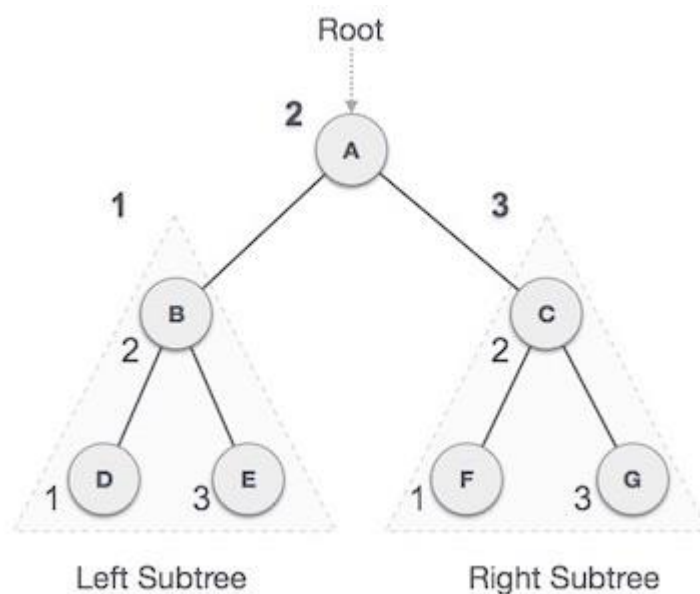
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

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

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

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



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

$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Algorithm

Until all nodes are traversed –

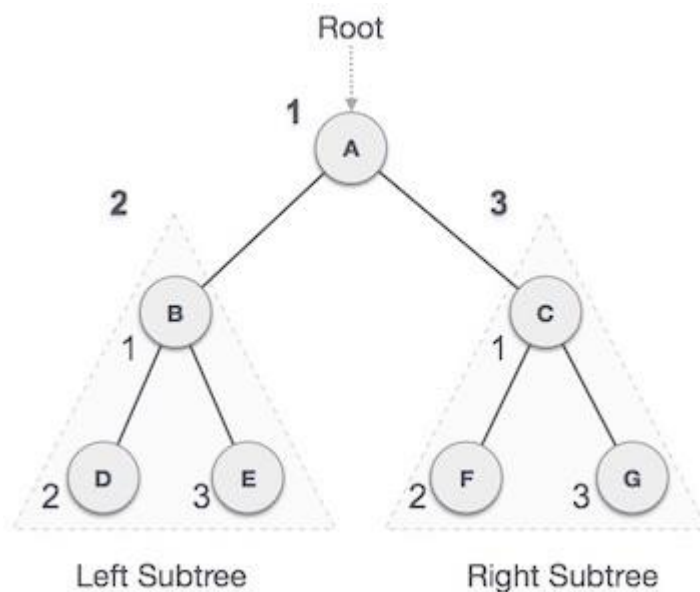
Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

Pre-order Traversal

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



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

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

Algorithm

Until all nodes are traversed –

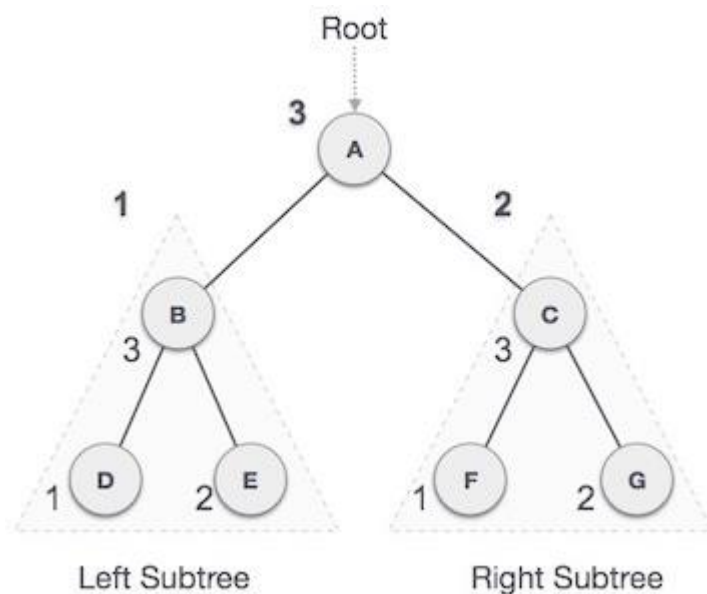
Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

Post-order Traversal

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



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

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Algorithm

Until all nodes are traversed –

Step 1 – Recursively traverse left subtree.

Step 2 – Recursively traverse right subtree.

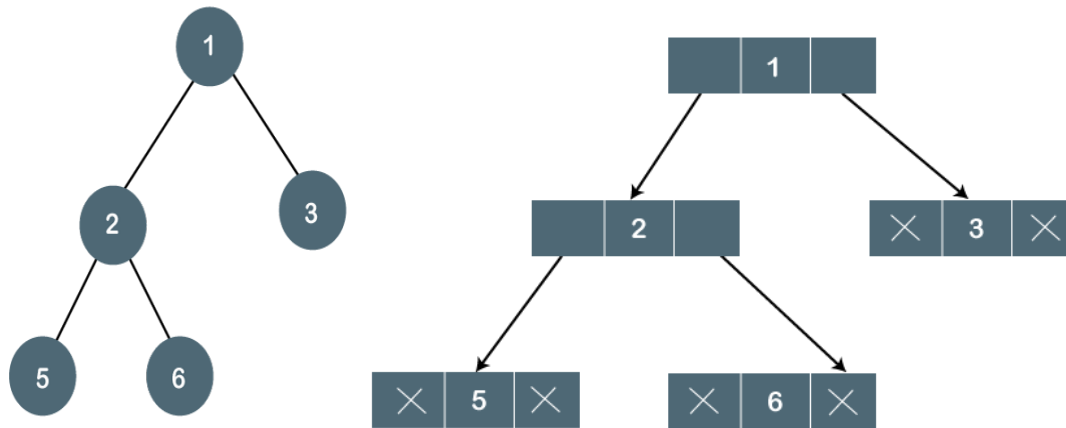
Step 3 – Visit root node.

Binary Tree ADT

Definition

Binary Tree is defined as a Tree data structure with at most 2 children. Here, binary name itself suggests that 'two'; therefore, each node can have 0, 1 or 2 children.

Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



Properties of a Binary tree

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

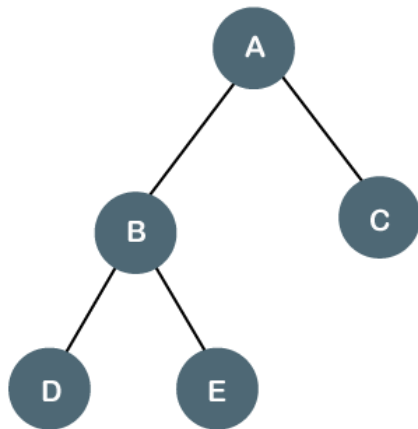
Types of Binary tree:

There are four types of Binary tree:

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

1. Full/ proper/ strict Binary tree

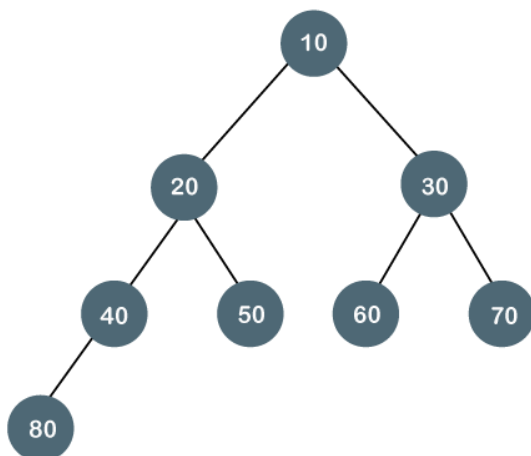
The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.



Complete Binary Tree

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

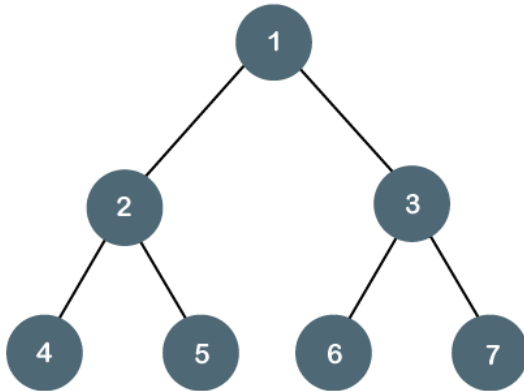
Let's create a complete binary tree.



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

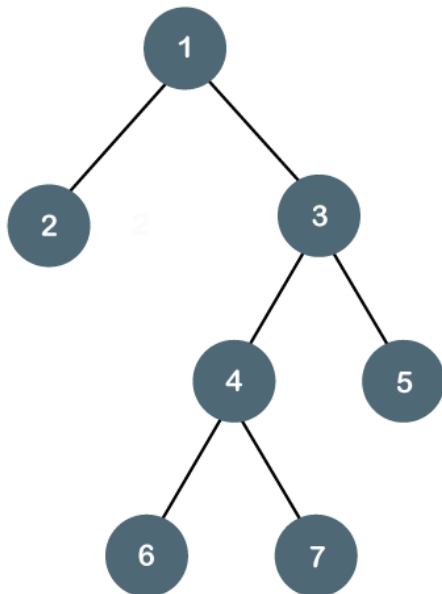
Perfect Binary Tree

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



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

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

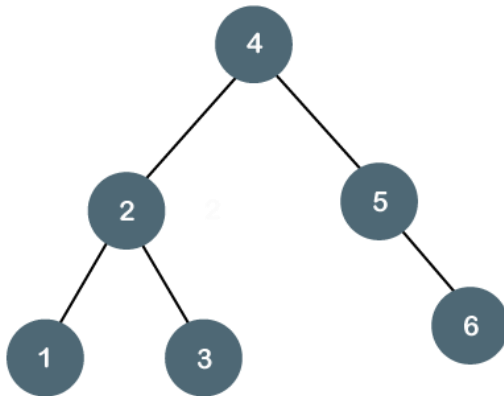


Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa are not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.

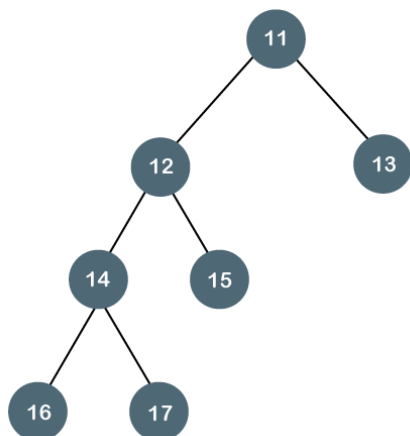
Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by at most 1. For example, *AVL*

Let's understand the balanced binary tree through examples.



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



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

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.

Properties:

Expression Tree is a special kind of binary tree with the following properties:

- Each leaf is an operand. Examples: a, b, c, 6, 100
- The root and internal nodes are operators. Examples: +, -, *, /, ^
- Sub trees are sub expressions with the root being an operator.

Traversal Techniques

There are 3 standard traversal techniques to represent the 3 different expression formats.

In order Traversal

We can produce an infix expression by recursively printing out

- the left expression,
- the root, and
- The right expression.

$(a+(b*c))+(d*(e + f))$

Postorder Traversal

The postfix expression can be evaluated by recursively printing out

- the left expression,
- the right expression and
- then the root

$a\ b\ c\ *\ +\ d\ e\ f\ *\ +\$

Preorder Traversal

We can also evaluate prefix expression by recursively printing out:

- the root,
- the left expression and
- the right expression.

$++a * b c * d + e f$

Construction of Expression Tree

Let us consider a postfix expression is given as an input for constructing an expression tree.

Following are the step to construct an expression tree:

1. Read one symbol at a time from the postfix expression.
2. Check if the symbol is an operand or operator.
3. If the symbol is an operand, create a one node tree and push a pointer onto a stack
4. If the symbol is an operator, pop two pointers from the stack namely T1 & T2 and form a new tree with root as the operator, T1 & T2 as a left and right child
5. A pointer to this new tree is pushed onto the stack

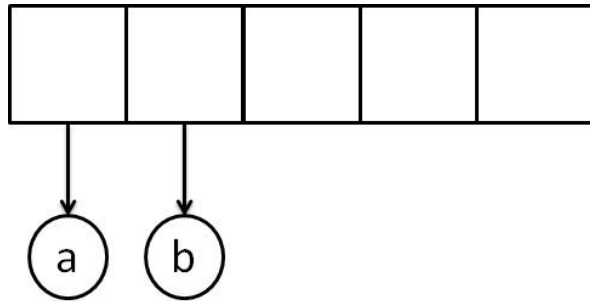
Thus, An expression is created or constructed by reading the symbols or numbers from the left. If operand, create a node. If operator, create a tree with operator as root and two pointers to left and right subtree

Example - Postfix Expression Construction

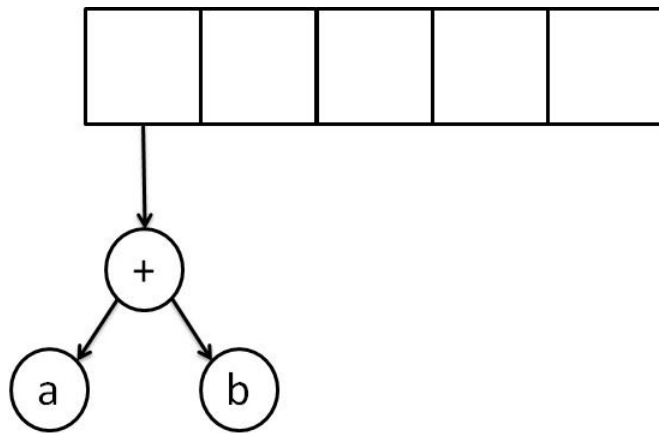
The input is:

$a b + c *$

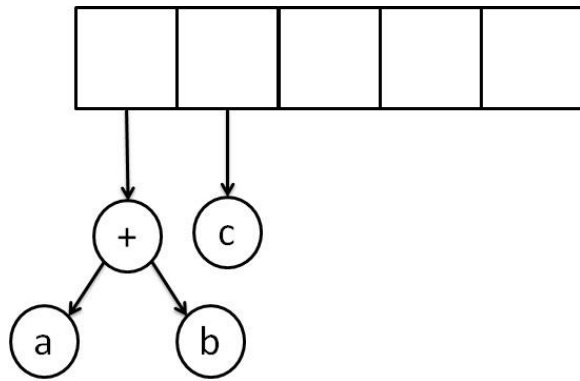
The first two symbols are operands, we create one-node tree and push a pointer to them onto the stack.



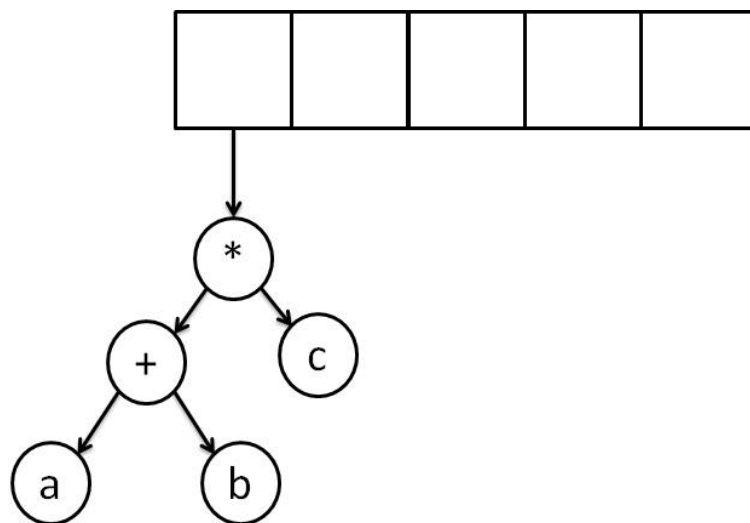
Next, read a '+' symbol, so two pointers to tree are popped, a new tree is formed and push a pointer to it onto the stack.



Next, 'c' is read, we create one node tree and push a pointer to it onto the stack.



Finally, the last symbol is read ' * ', we pop two tree pointers and form a new tree with a, ' * ' as root, and a pointer to the final tree remains on the stack.



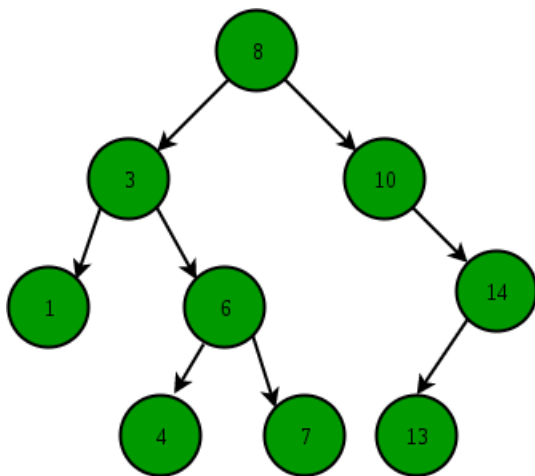
Binary Search Tree

Binary Search Tree is a variant of the binary tree following a particular order with which the nodes should be organized.

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.

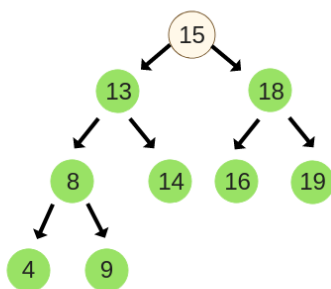
left_subtree(data) < root node < right_subtree(data).



For example: [15 , 18 , 13 , 8 , 19 , 4 , 16 , 9 , 14]



Representation of a BST



Operations in a BST

1. Traversal
2. Search
3. Insertion
4. Deletion

Traversal

1. The term traversal means to visit each node in a tree exactly once. In linear lists the order of traversal is first to last. However, in trees there exists no such natural linear order.
2. In binary tree or binary search trees, there exist two methods for traversal, namely:
 - Depth first search.
 - Breadth first search.

Depth first search

DFS can be defined as an algorithm, based on backtracking, dedicated to traversing a tree structure by first visiting the root first and then the left sub-tree and the right sub-tree.

It is of three types:

- In-order traversal.
- Pre-order traversal.
- Post-order traversal.

• **Pre-Order Traversal**

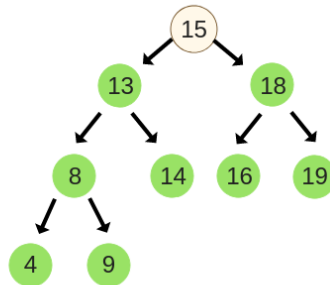
For traversal of a non-empty binary tree or binary search tree in Pre-order, following sequence of operations is performed.

- Visit the root node.
- Traverse the left sub tree in pre-order.
- Traverse the right sub tree in pre-order.

Considering the tree example above the Pre- order traversal of the tree would be,



Representation of a BST



pre_order = 15 , 13 , 8 , 4 , 9 , 14 , 18 , 16 , 19.

- **In-Order Traversal**

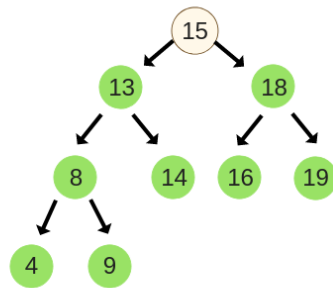
For traversal of a non-empty binary tree or binary search tree in **In-order**, following sequence of operations is performed.

- Traverse the left sub tree in in-order.
- Visit the root node.
- Traverse the right sub tree in in-order.

Considering the tree example above the In- order traversal of the tree would be,



Representation of a BST



in_order = 4 , 8 , 9 , 13 , 14 , 15 , 16 , 18 , 19.

- **Post-Order Traversal**

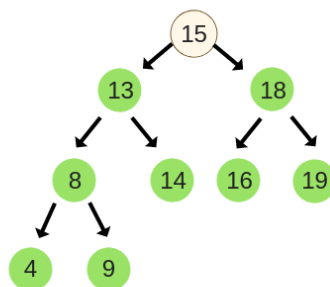
For traversal of a non-empty binary tree or binary search tree in Post-order, following sequence of operations is performed.

- Traverse the left sub tree in post-order.
- Traverse the right sub tree in post-order.
- Visit the root node.

Considering the tree example above the Pre- order traversal of the tree would be,



Representation of a BST



post_order = 4 , 9 , 8 , 14 , 13 , 16 , 19 , 18 , 15

Breadth first search

- BFS can be defined as an algorithm dedicated to traversing a tree structure, level by level or depth by depth.
- This category of tree traversal initializes from the root node and explores all the adjacent nodes on a current level and then moves on to the next level, and so on.

Taking the same example as above, the BFS traversal of the tree is shown in the image here.

bfs_traversal = 15 , 13 , 18 , 8 , 14 , 16 , 19 , 4 , 9

Search Operation

The algorithm depends on the property of BST that if each left sub tree has values below root and each right sub tree has values above the root.

Algorithm:

```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```

Insert Operation

Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

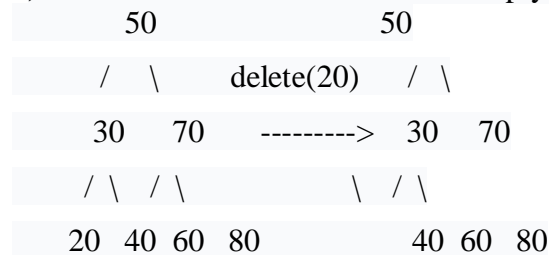
Algorithm:

```
If node == NULL
    return createNode(data)
if (data < node->data)
    node->left = insert(node->left, data);
else if (data > node->data)
```

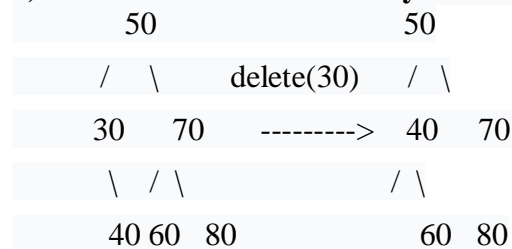
```
node->right = insert(node->right, data);  
return node;
```

Deletion

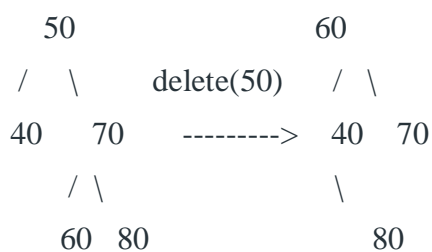
1) Node to be deleted is the leaf: Simply remove from the tree.



2) Node to be deleted has only one child: Copy the child to the node and delete the child



3) Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor.



The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in the right child of the node.

<https://www.programiz.com/dsa/binary-search-tree>

Binary Search Tree Applications

NON LINEAR STRUCTURES / 19CS307 - DATA STRUCTURES /MS.M.KANCHANA/CST/SNSCE

1. In multilevel indexing in the database
2. For dynamic sorting
3. For managing virtual memory areas in Unix kernel

AVL Tree

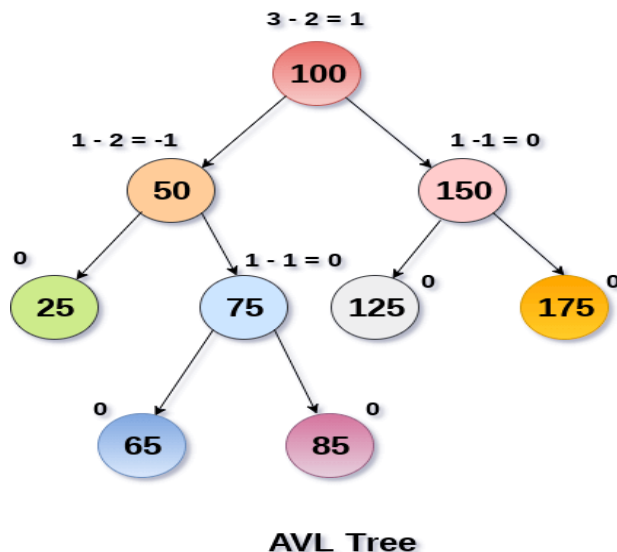
AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

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

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

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

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.



AVL Rotations

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

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

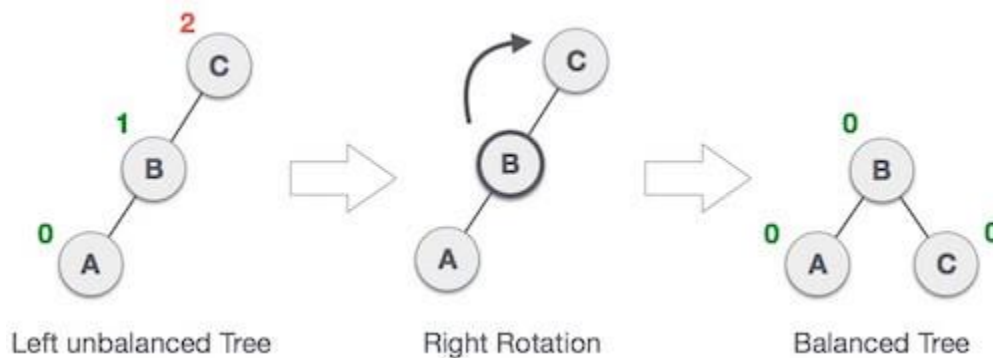
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

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

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

1. LL Rotation

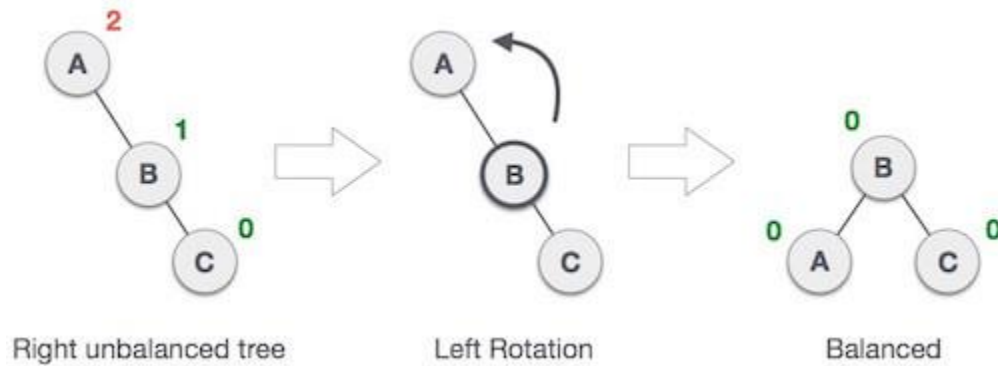
When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



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.

2. RR Rotation

When BST becomes unbalanced, due to a node is inserted into the right sub tree 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



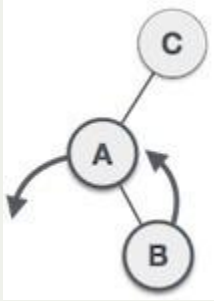
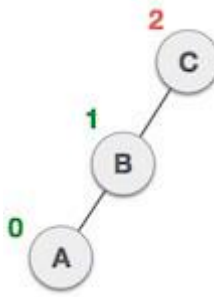
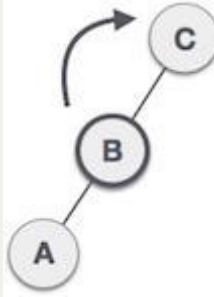
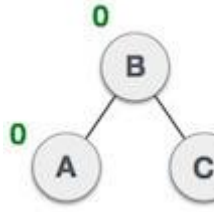
In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

3. LR Rotation

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

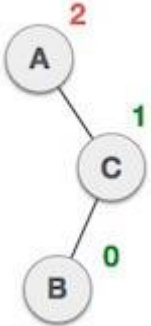
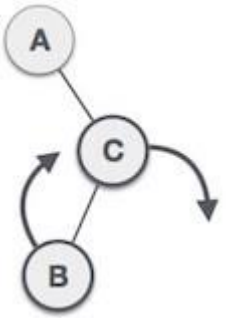
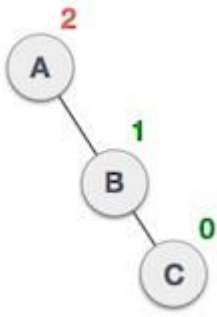
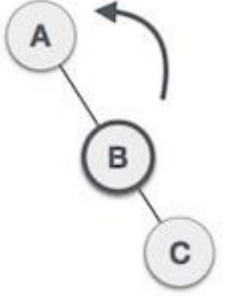
Let us understand each and every step very clearly:

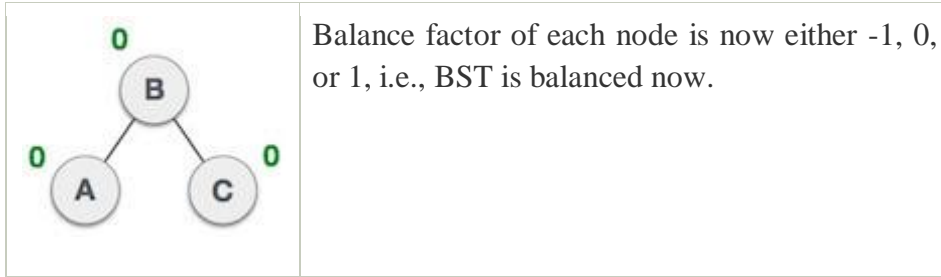
State	Action
	<p>A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C</p>

	<p>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.</p>
	<p>After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C</p>
	<p>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</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.</p>

4. RL Rotation

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

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



AVL Example:

<https://prepinsta.com/data-structures/avl-tree-insertion/>

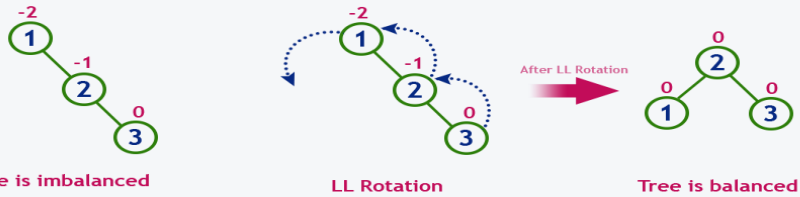
insert 1



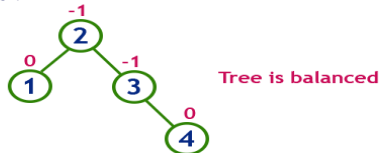
insert 2



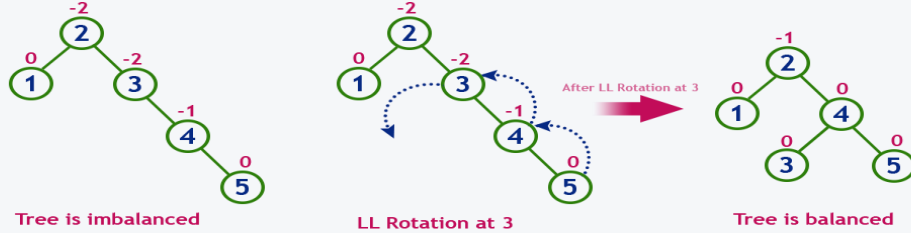
insert 3



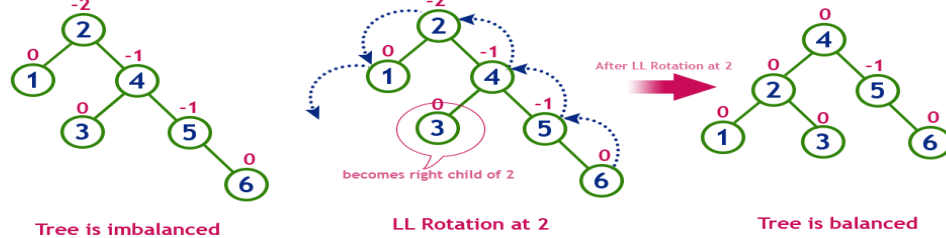
insert 4



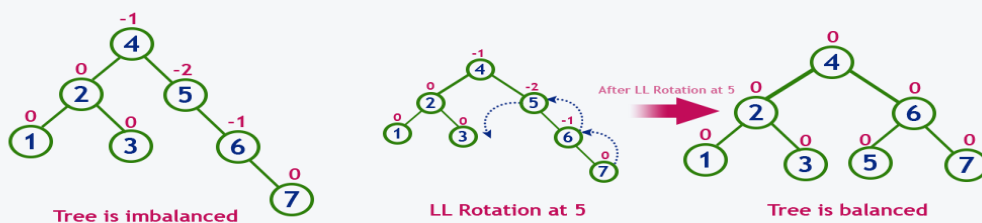
insert 5



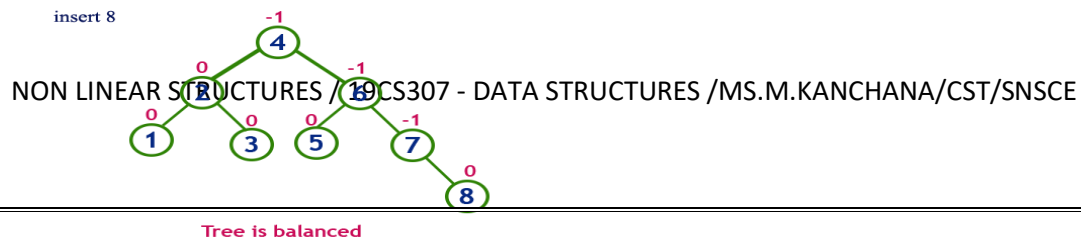
insert 6



insert 7



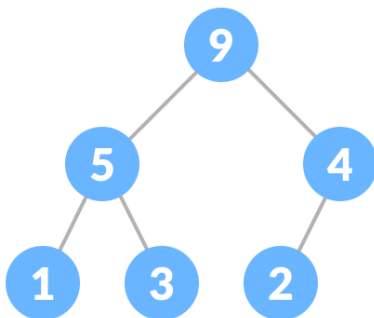
insert 8



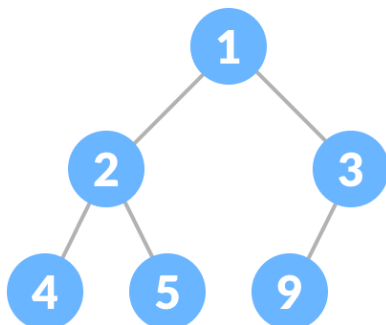
Heap

Heap data structure is [a complete binary tree](#) that satisfies **the heap property**, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called **max heap property**.



- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called **min heap property**.



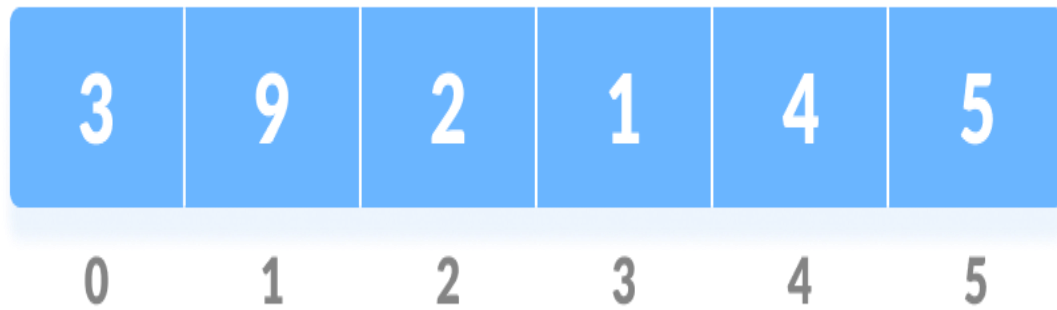
Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

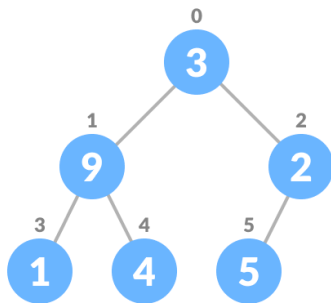
Heapify

Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

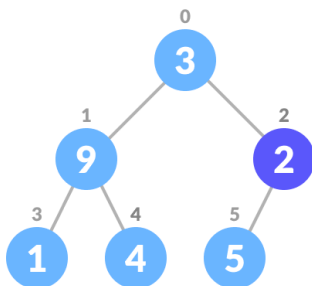
1. Let the input array be



2. Create a complete binary tree from the array Complete binary tree

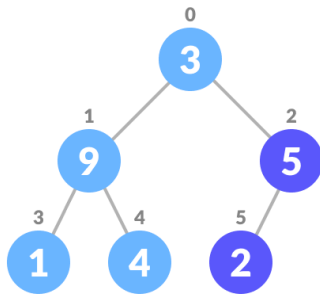


3. Start from the first index of non-leaf node whose index is given by $n/2 - 1$.



4. Start from the first on leaf node
5. Set current element i as largest.

6. The index of left child is given by $2i + 1$ and the right child is given by $2i + 2$.
If leftChild is greater than currentElement (i.e. element at i th index), set leftChildIndex as largest.
If rightChild is greater than element in largest, set rightChildIndex as largest.



7. Swap largest with currentElementSwap if necessary
8. Repeat steps 3-7 until the subtrees are also heapified.

Algorithm

Heapify(array, size, i)

set i as largest

leftChild = $2i + 1$

rightChild = $2i + 2$

if leftChild > array[largest]

set leftChildIndex as largest

if rightChild > array[largest]

set rightChildIndex as largest

swap array[i] and array[largest]

To create a Max-Heap:

MaxHeap(array, size)

loop from the first index of non-leaf node down to zero

call heapify

For Min-Heap, both leftChild and rightChild must be larger than the parent for all nodes.

Insert Element into Heap

Algorithm for insertion in Max Heap

If there is no node,

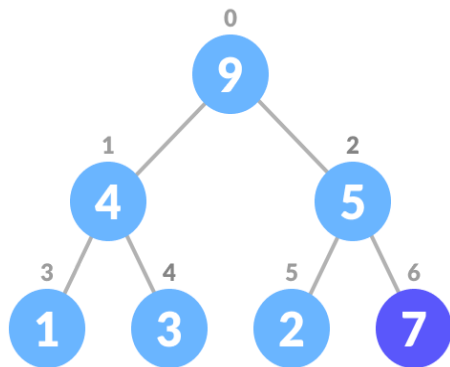
create a newNode.

else (a node is already present)

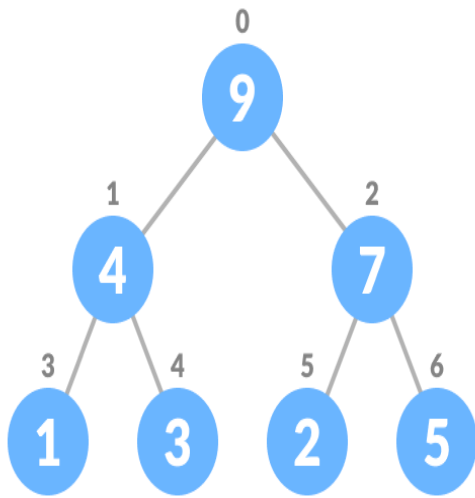
insert the newNode at the end (last node from left to right.)

heapify the array

1. Insert the new element at the end of the tree. Insert at the end



2. Heapify the tree.



For Min Heap, the above algorithm is modified so that parentNode is always smaller than newNode.

Delete Element from Heap

Algorithm for deletion in Max Heap

If nodeToBeDeleted is the leafNode

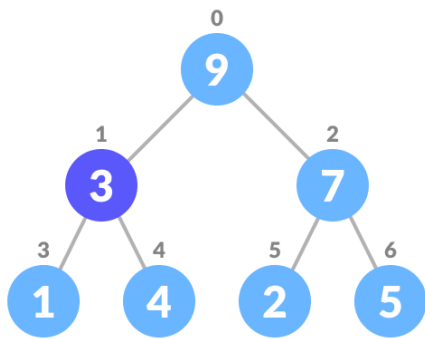
remove the node

Else swap nodeToBeDeleted with the lastLeafNode

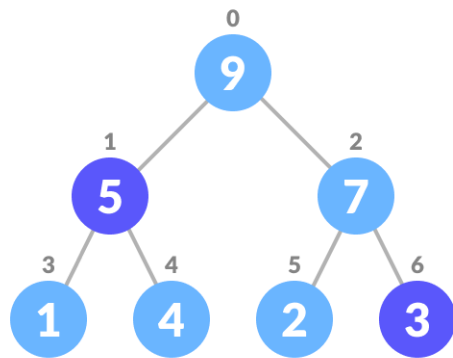
remove nodeToBeDeleted

heapify the array

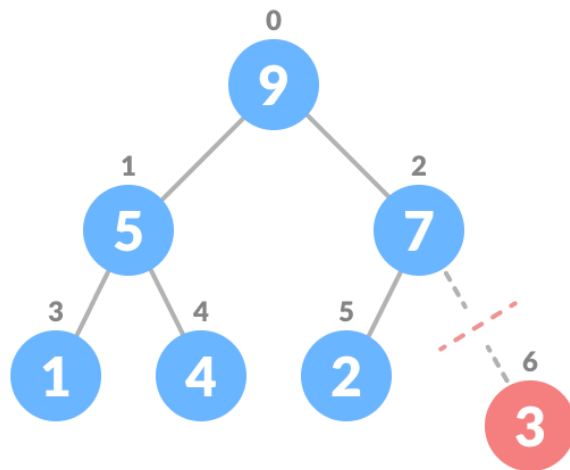
1. Select the element to be deleted.



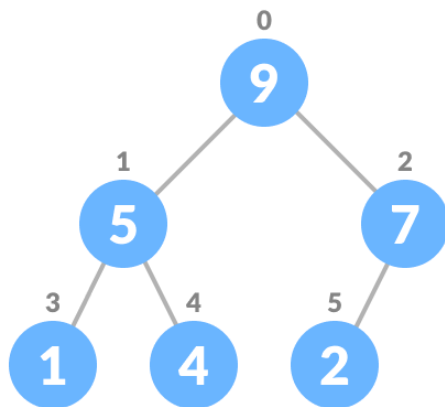
2. Swap it with the last element.



3. Remove the last element.



4. Heapify the tree.



For Min Heap, above algorithm is modified so that both childNodes are greater smaller than currentNode.

Peek (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

return rootNode

Extract-Max/Min

Extract-Max returns the node with maximum value after removing it from a Max Heap whereas Extract-Min returns the node with minimum after removing it from Min Heap.

Applications of the Heap Data Structure

1. Heap is used in the construction of [priority queues](#). We can insert, delete, identify the highest priority element, or insert and extract with priority, among other things, in $O(\log N)$ time using a priority queue.

Although data structures such as BST, AVL trees, and the Red-Black tree can accomplish the same functionalities, they are more complex than heaps.

Priority queues themselves have more advanced uses, such as bandwidth control in an n/w router, prims and Dijkstra's algorithm, Huffman coding, and the BFS algorithm.

A real-Life example where a priority queue can be used:

This type of queue could be used when customers who take a short time to serve are given priority instead of those customers who arrive early. For example, customers with a small bill to pay may be given precedence at a licensing center. The average waiting time for all clients in the queue is reduced due to this.

2. **Order statistics:** The Heap data structure can be used to find the kth smallest / largest element in an array quickly and effectively.
3. Heap Sort is used in systems concerned with security and embedded systems, such as the Linux Kernel.