5 Tree

5.1 TREE

Definition

"Tree is a non-linear data structure. A tree is defined recursively as a collection of nodes."

Terminology:

Root:

It is the topmost node of a tree.

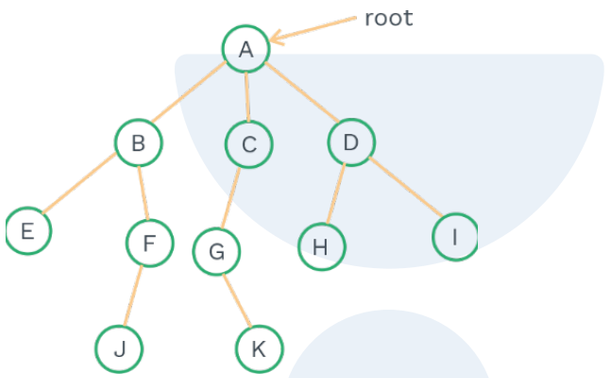


Fig. 5.1 Binary tree

Edge:

An edge refers to the link from parents to children (all links in Fig. 5.1).

Leaf:

A node, which has no children, is known as leaf node.

Example: E, H, J, K and L are all leaf nodes.

Siblings:

Two (or more) children are called to be siblings, if they have same parent node.

Example:

F and E are siblings.

H and I are siblings.

C, B and D are siblings.

Depth:

Depth of a node is the number of links/edges from root to a particular node.

Example: Depth of G = 2.

Level:

- Level = 1+ number of edges between a node and root
  =1+ depth of a node
  where, depth starts from 0
  Example: In Fig. 5.1, B, C and D are on the same level.

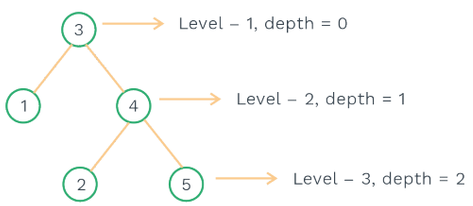


Fig. 5.2

Ancestor and descendant:

- A node 'x' is an ancestor of a node 'y', if there exists a path from the root to 'y', and 'x' appears on the path.
- The node 'y' is called a descendant of 'x'. For example A, C and G are the ancestors for K in Fig. 5.1.

Height:

- Height of a node is the number of edges in the path from that node to its most distant leaf node.
- Height of root node in the below figure = 3.
- In the below example, the height of B is 2 (B–F–J). Height of tree = 3 as shown in below figure.

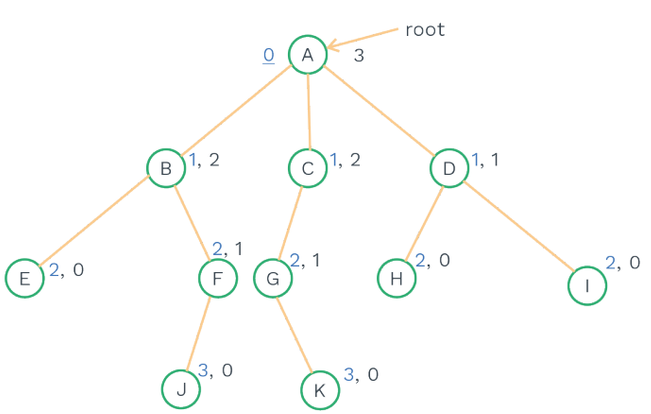


Fig. 5.3

Note:

i) Number in blue represent the depth of the node
ii) Number in black represent the height of the node.

Height of tree:

- Height of tree is the height of root node.
- Height of root node is the number of edges to its the most distant leaf node.

Skew tree:

It is a binary tree, where every node is having one child except for the leaf node.

Fig. 5.4 Skew Tree

Left skew tree:

"It is binary tree, where each node will have only one left child except the leaf node."

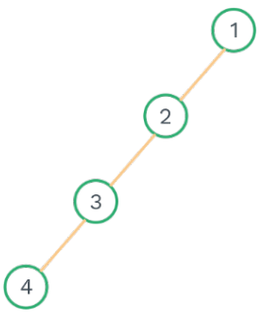


Fig. 5.5 Left Skew Tree

Right skew tree:

"It is binary tree, where each node will have only one right child except the leaf node."

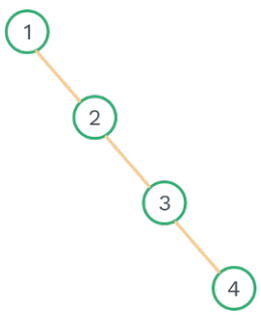


Fig. 5.6 Right Skew Tree

Binary trees:

- Each node in a binary tree can have either 0, 1 or 2 children.
- A binary tree without any node is also a valid binary tree and called as an empty or null binary tree.

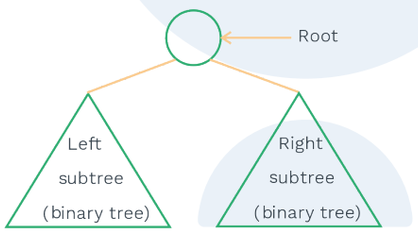


Fig. 5.7 Conceptual Structure of a Binary Tree

Example:

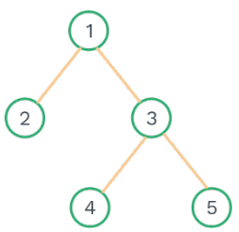


Fig. 5.8

Strict binary tree:

A strict binary tree is a kind of tree, where each node will have either no children or exactly 2 children.

Example:

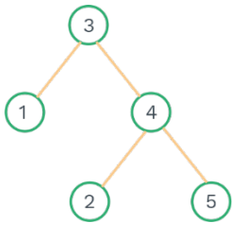


Fig. 5.9 Strict Binary Tree

Full binary tree:

A full binary tree is a binary tree, in which all the internal nodes have exactly two children and all the leaf nodes are present at the last level.

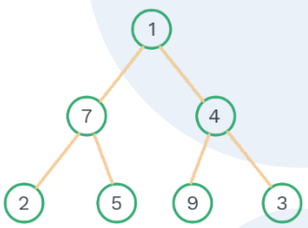
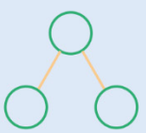
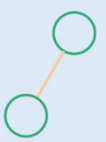
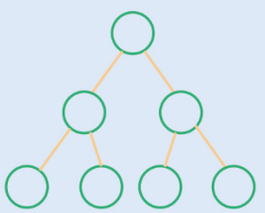
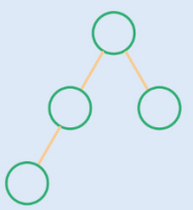


Fig. 5.10 Full Binary Tree

Complete binary tree:

- A complete binary tree is a kind of tree, where except the last level all the other levels are completely filled.

Properties of complete binary trees:

Height of tree	Maximum number of nodes	Minimum number of nodes
h = 0	 1	 1
h = 1	 3	 2
h = 2	 7	 4

In a complete binary tree with n nodes, the number of internal nodes = $\lfloor n / 2 \rfloor$

The maximum number of nodes in a complete binary tree = $2^{(h+1)}-1$.

Properties of full binary tree:

- Number of nodes(n) in a full binary tree of height h = 2^{h+1}-1.

  at depth = 0, Number of nodes = $2^0 = 1$
  at depth = 1, Number of nodes = $2^1 = 2$
  at depth = 2, Number of nodes = $2^2 = 4$
  at depth h, Number of nodes = 2^h

  $[2^0 + 2^1 + 2^2 + \ldots\ldots\ldots 2^h] = 2^{(h+1)}-1$

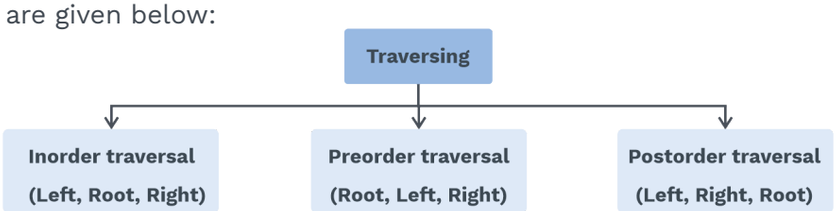
- Number of leaf-nodes in a full binary tree = 2^h.
- Number of non-leaf nodes in a full binary tree =(2^h)-1.

Note:

The definition of a Strict binary tree, Full binary tree and complete binary tree varies in different text books but in GATE, the definition will be specified properly in the question.

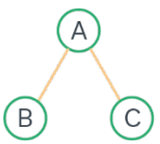
Traversing

- To see the information present in every node, we visit every node, which is called traversing.
- The most popular traversing methods which are applied in binary tree, are given below:



- Generally, they are applicable on the binary tree, but they can be extended to ternary tree or m–way tree.
- Inorder traversal:
  i) Visit left subtree **ii)** Visit root **iii)** Visit right subtree
- Preorder traversal:
  i) Visit root **ii)** Visit left subtree **iii)** Visit right subtree
- Postorder traversal:
  i) Visit left subtree **ii)** Visit right subtree **iii)** Visit root

Example:



Inorder traversal : BAC
Preorder traversal : ABC
Postorder traversal : BCA

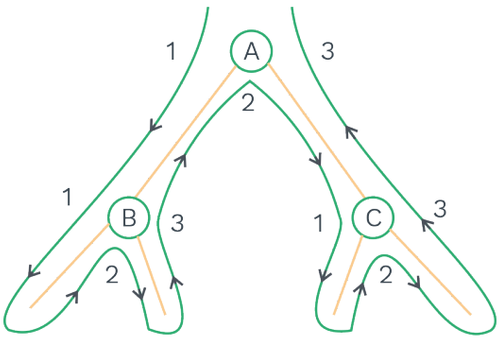
Finding out inorder, preorder and postorder

- For a given tree, if any node (even that node is leaf node) does not have any children, then give them dummy children.
- After that, start traversing on the tree from the top to down, left to right.

Preorder

- To get a preorder of any tree, whenever we visit any node for the 1st time, we print it.

Example:



So, the order, in which we visit the node for 1st time is ABC. So, ABC is preorder of tree.

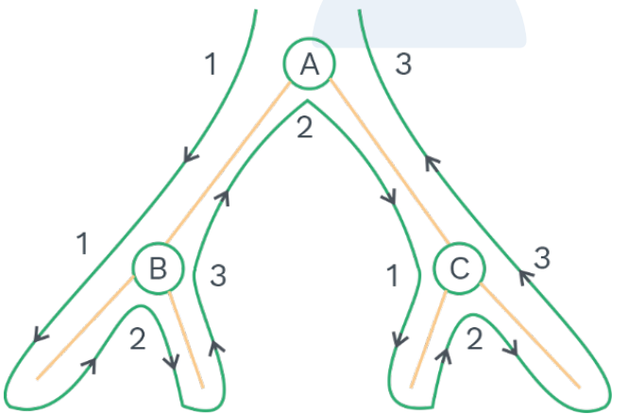
If a node has no child, then its left and right pointers will point to NULL

Note:

Every node will be visited three times before we finish walking entire tree and go back to the root node.

Inorder

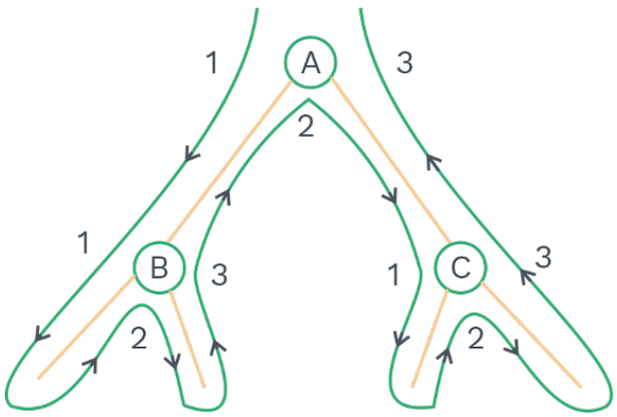
- To get Inorder of any tree whenever we visit any node for the 2nd time, then print it.



- BAC is an order of nodes, in which nodes are visited for the 2nd time. So, BAC is the inorder traversal of tree.

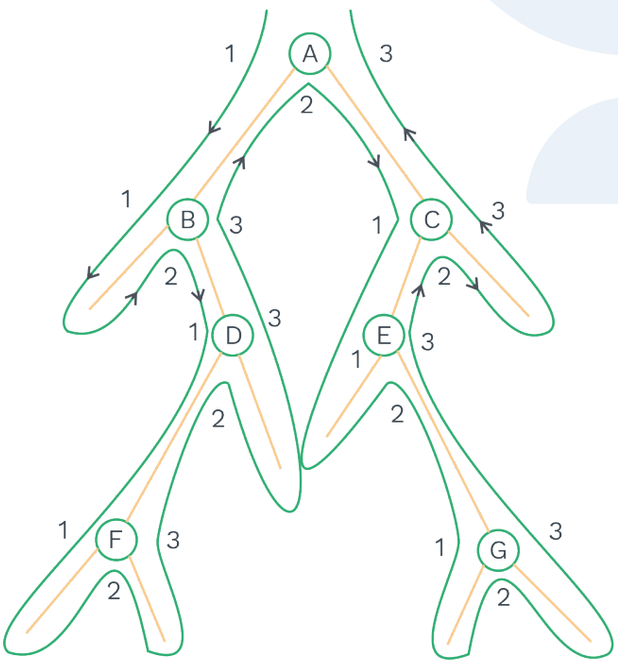
Postorder

- To get postorder of any tree whenever we visit any node for the 3rd time, then print it while walking the tree.



- BCA is an order of nodes, in which nodes are visited for 3rd time. So, BCA → postorder–traversal of tree.

Example:



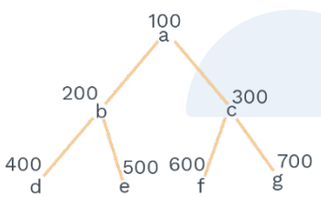
We added dummy nodes for each node, which are not having children.
Preorder traversal: ABDFCEG
Inorder traversal: BFDAEGC
Postorder traversal: FDBGECA

Implementation of inorder traversal

```
Struct node
    {
            char data;
            struct node *left , *right;
    };
    void Inorder (struct node *t)
    {
            if (t)
            {
               Inorder (t → left);
               printf ("%c", t → data);
               Inorder (t → right);
            }
    }
```

Example:

Let the binary tree and address are as shown below:



So, the output is dbeafcg.

Time complexity:

- The best way to analyse the code is to see how many times the codes are visiting the node and time taken by each node at each time.

So, total time complexity = 3 × n × constant time
= O(n)

Space complexity:

- Space complexity depends on the number of levels in the tree, and depending on the level of tree, the stack grows.
- In the worst case, if a tree has 'n' nodes, then n level can be possible, because in worst case, a tree can be a skew–tree.

- So, for every node, there will be a data in the stack.
  Therefore,

Space Complexity : O(n)

Preorder traversal

- Code:

```
                Struct node
                {
                        char data;
                        struct node *left , *right;
                };
void preorder (struct node *t)
    {
        if (t)
            {
                printf ("%c", t → data);
                preorder (t → left);
                preorder (t → right);
            }
        }
```

Similar to Inorder Traversal, in Preorder traversal also, we have to visit each node once. Thus,

Time Complexity = O(n)

In worst case, Tree can be a skewed tree. So, in the worst case, the height of the tree can be n. Therefore,

Space Complexity = O(n)

Postorder traversal

- The code of postorder traversal is shown below:

```
                Struct node
                {
                        char data;
                        struct node *left , *right;
                };
```

```
void postorder (struct node *t)
    {
            if (t)
               {
                   postorder (t → left);
                   postorder (t → right);
                   printf ("%c", t → data);
               }
    }
```

Similarly, in Postorder traversal also, we have to visit each node once. Thus,

Time Complexity = O(n)

In worst case, Tree can be skewed one. So in the worst case, height of the tree can be n. Therefore,

Space Complexity = O(n)

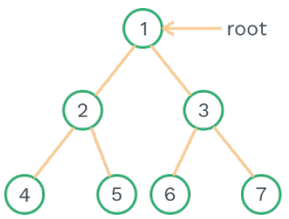
Note:

Space and Time complexity of Inorder, Preorder and Postorder traversals are O(n) only.

Level order traversal:

- The root of the tree as input is given to the algorithm.
- The while loop in the algorithm dequeues the root, prints it and enqueues it's left and right nodes.
- The procedure is repeated until the queue is not empty.

Example:



The order, in which the nodes need to be visited = 1,2,3,4,5,6,7

Note:

Level order traversal of a tree is same as breadth-first traversal for the tree.

Double order

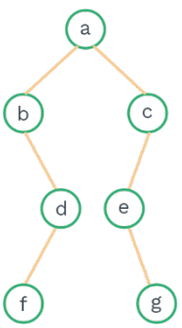
- **1)** In double order traversal, we print each node two times based on code given.

Recursive code:

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

void DO (struct node *t)
{
  if(t)
  {
      printf ("%c", t → data);
      DO (t → left);
      printf ("%c", t → data);
      DO (t → right);
    }
}
```

Example:



The double order traversal of the above tree is abbdffdaceeggc.

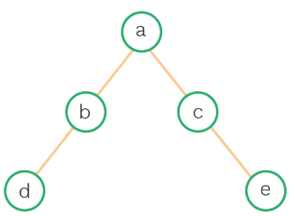
Triple order traversal:

- Triple order traversal means printing every node three times based on below code.

Code:

```
struct node
{
        char data;
        struct node *left;
        struct node *right;
};
void TO(struct node *t)
{
    if(t)
    {
        printf (" %c", t → data);
        TO(t → left);
        printf ("%c", t → data);
        TO (t → right);
        printf (" %c", t → data);
    }
}
```

Example:



- Triple order traversal of the above tree is abdddbbacceeeca.

Unlabeled tree:
In an unlabeled tree, the nodes do not have a specific name.

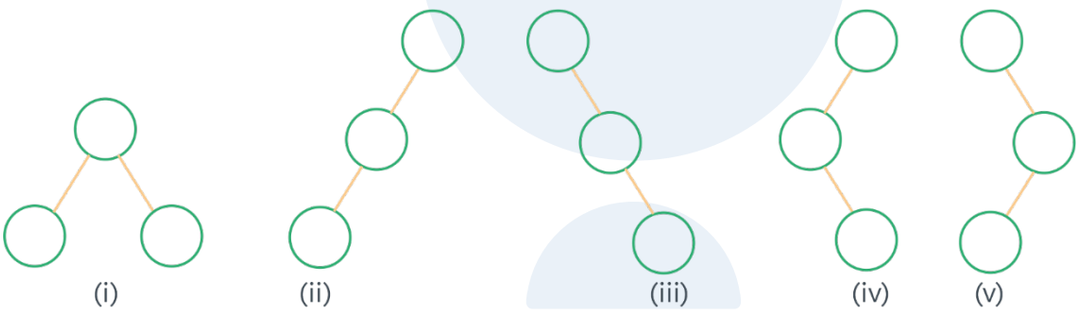
Labeled tree:
In a labeled tree, the nodes have a specific name.

Number of binary tree:

- With one node, the number of unlabeled binary trees possible is one, and the node itself is a root.

- With two nodes, the number of unlabeled binary trees possible is 2 and are given below:

- With three nodes, the number of unlabeled binary trees possible is 5 and are shown below:

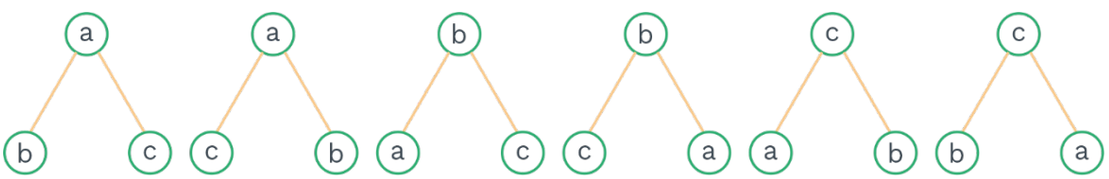


Note:

If we name the node, then the number of trees possible will be even more.

- Number of binary trees possible with 'n' unlabelled nodes is $^{2n}C_n / (n+1)$.

  Also known as catlan number.
- Number of binary trees possible with 'n' labelled nodes is $\left[{}^{2n}C_n / (n+1)\right] \times n!$.
- In case of a binary tree having three nodes, each unlabelled tree can be represented in 6 ways in a labelled tree as shown below:
  (Let a, b, c are the names of nodes)



- Every unlabelled tree with three nodes can be labelled in 6 different ways.
- So, the total number of labelled binary trees possible = 5 * 6 = 30.
- Given n nodes, the number of structured (i.e., unlabelled) binary trees possible are ${}^{2n}C_n / (n+1)$, and for any given structure, we can find one tree, which is having the particular preorder or inorder or postorder.

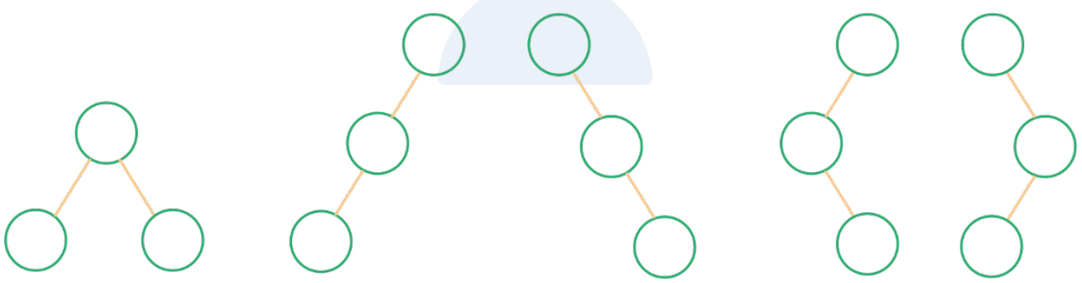
SOLVED EXAMPLES

Q1 **Give all binary trees with three nodes P, Q and R, which have preorder as PQR.**

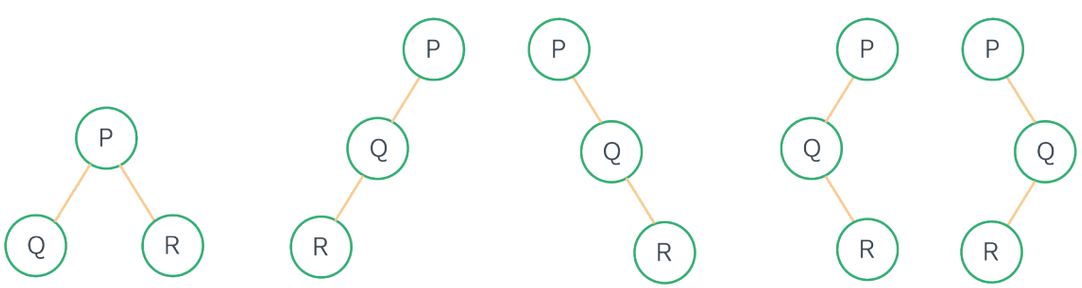
Sol: With three nodes, the number of structures $= \frac{{}^{2n}C_n}{(n+1)}$

$$= \frac{{}^{6}C_3}{4} = \frac{\frac{6!}{3!3!}}{4} = \frac{\frac{6.5.4}{3.2}}{4} = 5.$$

So, five structures are possible, as shown below:



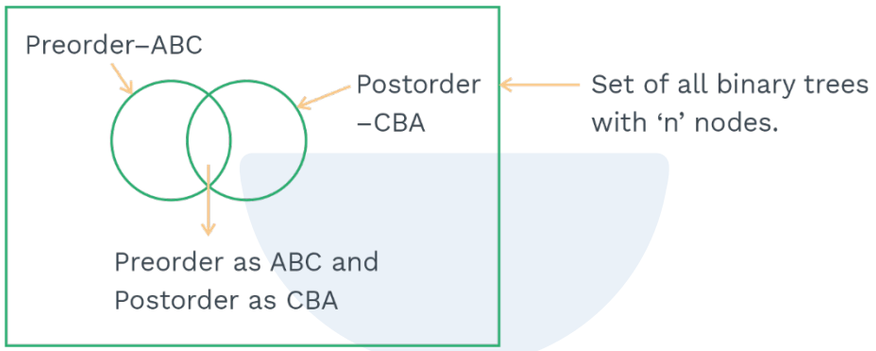
And each structure will represent one particular preorder, i.e., PQR, as shown below:



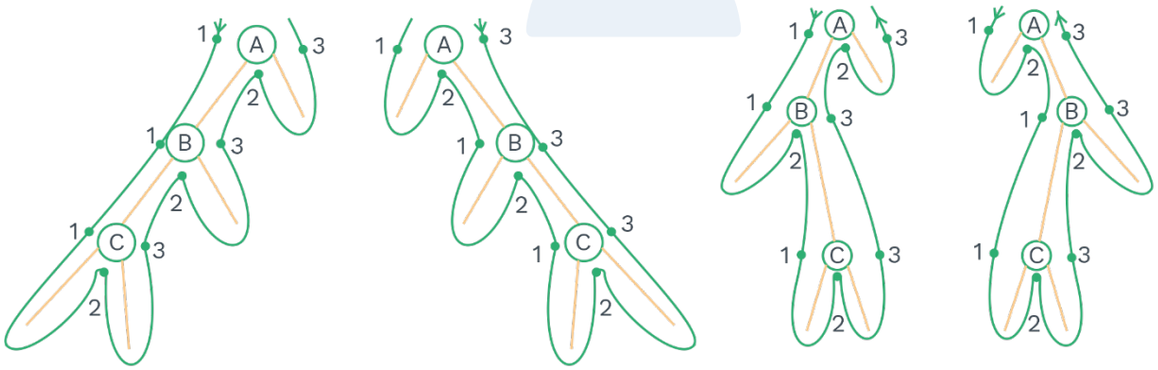
Q2 **What are the number of binary trees possible, if preorder is ABC and postorder is CBA.**

Sol: The number of binary trees possible with three nodes and postorder CBA is also five, because we have five structures possible.

In each structure, there will be exactly one binary tree with CBA as a postorder.



The binary tree with three nodes A, B and C having preorder as ABC and postorder as CBA are given below:



Postorder – CBA
Inorder – CBA
Preorder – ABC

Postorder – CBA
Inorder – ABC
Preorder – ABC

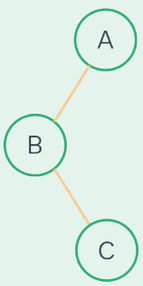
Postorder – CBA
Inorder – BCA
Preorder – ABC

Postorder – CBA
Inorder – ACB
Preorder – ABC

So, the number of a binary tree, which is having preorder ABC and postorder CBA are 4.

Note:

If we apply one more filter as inorder BCA, then we will get only one tree as given below.

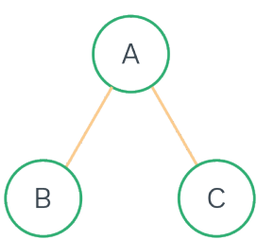


There will always be only one binary tree satisfying all the conditions given specified preorder, postorder and inorder.

Q3 **Let us say, there are three nodes, and three nodes are labelled as A, B and C. The preorder is given as ABC and inorder is given as BAC. Construct the binary tree for the specific order.**

Sol: In preorder, the root will be on the left most side of the given preorder sequence. So, A is the root of the binary tree.

In inorder, the root will be at the middle, and the left subtree will be at the left of the root. Similarly, the right subtree will be at the right of the root.
Hence, the binary tree will look like as shown below:



Preorder : ABC (Root,Left,Right)
Inorder: BAC (Left,Root,Right)

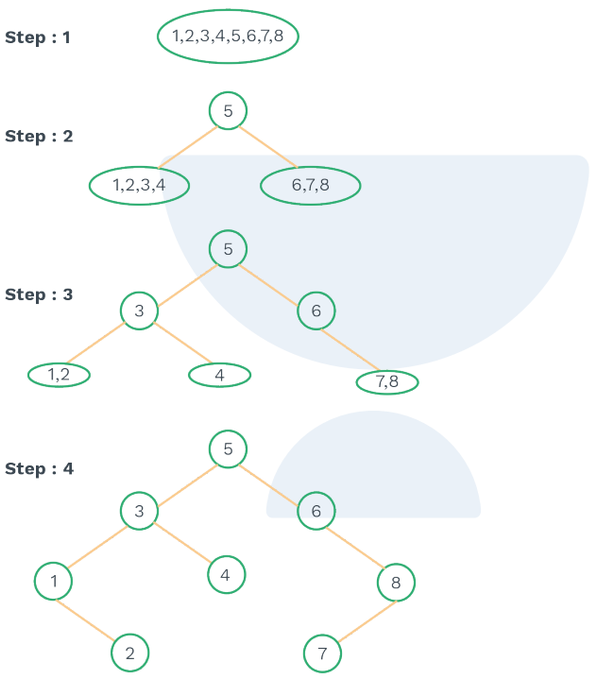
Note:

i) If postorder is given then we get root from right to left.
ii) So, if preorder and inorder are given or postorder and inorder is given, we will be able to construct a unique binary tree.

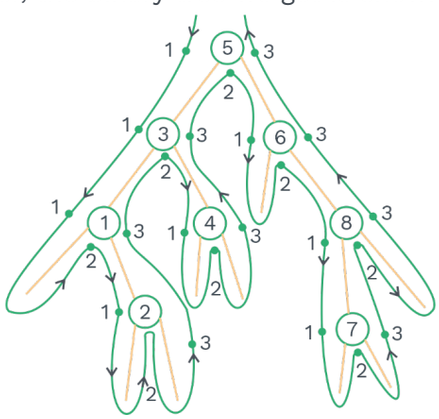
Q4 **Given an inorder as 1, 2, 3, 4, 5, 6, 7, 8 and preorder as 5, 3, 1, 2, 4, 6, 8, 7. Find postorder = ?**

Sol: Preorder: (Root,Left,Right)

Inorder: (Left,Root,Right)
Thus, 5 will be the root.



Using Preorder and Inorder, the binary tree we get is shown below:



The postorder is 2, 1, 4, 3, 7, 8, 6, 5.

Note:

If only one order is given, we will not be able to create a unique binary–tree.

Recursive program to count number of nodes

Suppose 'NN' represents number of nodes.

The recursive equation to count number of nodes is shown below:

NN(T) = 1 + NN(LST) + NN(RST)

Here, LST and RST represent left–subtree and right-subtree, respectively.

Base condition:

NN(T) = 0; when T = 0 (if T is NULL)

Recursive program to count the number of nodes:

```
struct node
{
  int i;
  struct node * left;
  struct node * right;
};
int NN(struct Node *t)
{
  if(t)
    return (1+NN(t → left) + NN(t → right));
  else
    return 0;
}
```

Recursive program to count the number of leaves

Let 'NL' denotes the number of leaf nodes in the tree.

The recursive equation is given below:

NL(T) = 1; if T is leaf

= NL(LST) + NL(RST), otherwise

Program:

```
int NL(struct node *t) {
 if (t == NULL)
    return 0;
 else if (t → left == NULL && t → right == NULL)
    return 1; /* this condition checks whether 't' is leaf or not */
 else
    return (NL(t → left) + NL(t → right));
 }
```

Recursive program to count number of non-leaf

Let 'NNL' denotes the number of non-leaf.
The recursive equation is given below:
NNL(T) = 0, if T is leaf or T is NULL
= 1 + NNL(LST) + NNL(RST), otherwise

Recursive program

```
int NNL(struct node *t) /* Here, root of the tree is passed */
{
  if(t == NULL)
    return 0;
  if (t → left == NULL && t → right == NULL)
    return 0;
  else
    return (1 + NNL(t → left) + NNL(t → right));
}
```

Time complexity:

- Here, every time we visit the node, we are doing constant work.
- Here, three times we are visiting the each node.

So, Time complexity = 3 * C * n = O(n)

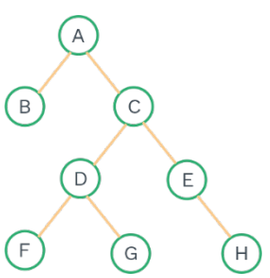
Space complexity

Space complexity depends upon the tree's height, and in the worst-case scenario, tree will be skewed.
Space complexity= O(n)

Full node

- A node having all children is called a full node.
- In a binary tree, a node having exactly two children is called a full node.
- Every full node is non-leaf node, but all non-leaf nodes are not a full node.

Ex:



Leaf nodes are: B, F, G, H
Non–leaf nodes are: A, C, D, E
Full nodes are: A, C, D

Note:

Full nodes will always be a subset of non-leaf nodes.

The recursive equation of the full node is given below:
Let FN denotes full node.

FN(T) = 0; T = NULL
= 0; T is a leaf
= FN(T → LST) + FN(T → RST); if T has only one child.
= FN(T → LST) + FN(T → RST) +1; if T is a full node.

where LST means Left-Subtree and RST means Right-Subtree

Program:

```
int FN(struct node *t)
{
  if (!t)
    return 0;
  if (!t → left && ! t → right)
    return 0;
  return (FN (t → left) + FN (t → right) + (t → left && t → right) ? 1: 0);
}
```

Time Complexity : O(n)
Space Complexity : O(n)

Recursive program to find height of a tree

Let us say, the height of the tree pointed by the pointer T is H(T).

The recursive equation is:

$$H(T) = \begin{cases} 0; \text{if T is empty} \\ 0; \text{if T is leaf} \\ 1 + \max(H(LST), H(RST)); \text{otherwise} \end{cases}$$

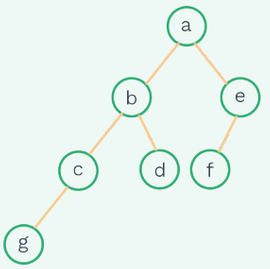
Program:

```
int H(struct node *t)
{
 if (!t && (!t → left && !t → right))
     return 0;
 else
      return(1+ ((H(t → left) > H(t → right)) ?
H(t → left): H (t → right);
}
```

Time Complexity : O(n)
Space Complexity : O(n)

Previous Years' Question

Which of the following sequences denotes the post order traversal sequence of the below tree?



a) fegcdba **b)** gcbdafe
c) gcdbfea **d)** fedgcba
Sol. c) **(GATE: 1996)**

Previous Years' Question

An array X of n distinct integers is interpreted as a complete binary tree. The index of the first element of the array is 0. The index of the parent of element X[i], $i \neq 0$, is?

a) $\left\lfloor \frac{i}{2} \right\rfloor$ **b)** $\left\lceil \frac{i-1}{2} \right\rceil$ **c)** $\left\lceil \frac{i}{2} \right\rceil$ **d)** $\left\lceil \frac{i}{2} \right\rceil - 1$

Sol: d) **(GATE: 2006)**

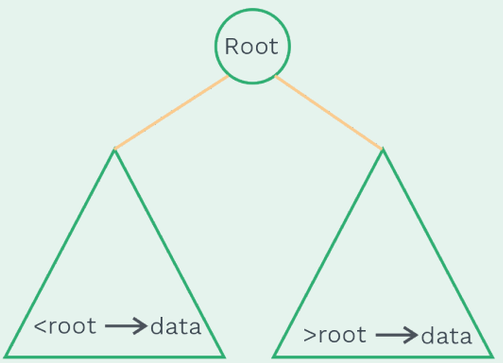
Binary search trees (BSTs)

- BSTs are useful for searching.
- Binary Search Tree Property:- For all nodes, the value of a node should be greater than all nodes in left subtree and smaller than all nodes in right subtree.

Note:

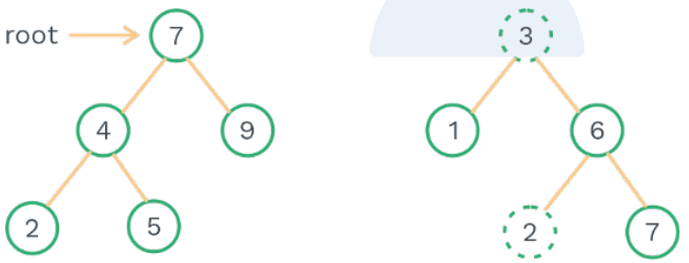
i) The above-mentioned property needs to be satisfied by every node present in the tree.

ii) Both the left-subtree and right-subtree have to be BSTs.



Example:

In the below diagram, the tree on left is a BST, but the right one is not a BST. As node 2 is smaller than 3, but is present in the right subtree of node 3. Thus, it is violating the property of BST.



Declaration of BST

- BST is a binary tree whose in-order traversal gives a sorted order of elements.

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

Operation on binary search trees

Some of the operations supported by BSTs are

- Find Minimum element
- Find Maximum element
- Insertion in BST
- Deletion in BST
- Searching an element

Note:

- Since root value is always in between left subtree data's and right subtree data's, performing an Inorder traversal on BST produces a sorted list in an increasing order.
- Searching of a key element:
  if key_element < root → data, then search only in the left subtree.
  if key_element > root → data, then search only in the right subtree.
  if key_element == root → data then key_element is found
  In worst case, searching time in Binary Search Tree will be O(n).

Finding an element in binary search trees

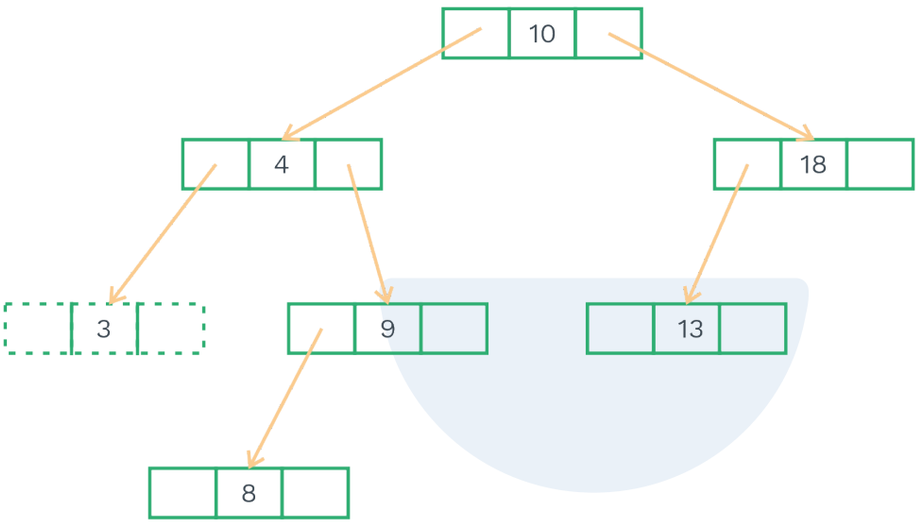
- First, start traversing from root node, BST-property is used to find any element in BST.
- If the value of the element, which we want to search is less than the value of root-node, then search only in the left sub-tree.
- If the value of the element, which we want to search is greater than the value of root-node, then search only in the right sub-tree.

```
struct BST_Node *Find(struct BST_Node * root, int data)
{
  if(root == NULL)
     return NULL;
  if(data < root → data)
     return (Find(root → left, data));
  else if(data > root → data)
     return (Find (root → right, data));
  else if(root->data ==data)
     return root;
}
```

Time Complexity : O(n), in the worst case (when BST is a skewtree).
Space Complexity : O(n), for a recursive stack, in the worst case.

Finding a minimum element in binary search trees

- In BSTs, the minimum element is the leftmost node, which is either a leaf node or a node with a right child but not with a left child.
- In the below example, the minimum element is 3.



Time Complexity : O(n), worst case (when BST is a left skew tree).
Space Complexity : O(n) for a recursive function stack.

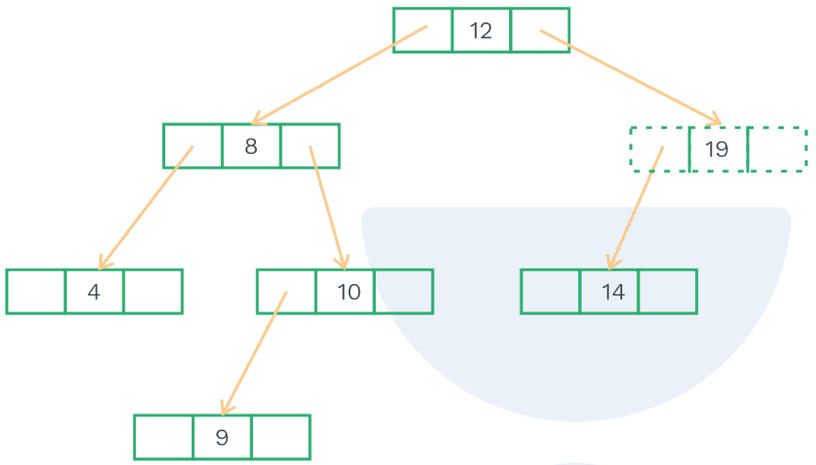
- Non-recursive way to explain the above algorithm:

```
struct Node *Minimum(struct Node *Root)
{
    if(Root==NULL)
       return NULL;
    while(Root → l_child !=NULL)
       Root=Root → l_child;
    return Root; }
```

Time Complexity : O(n)
Space Complexity : O(1)

Finding a maximum element in binary search trees

- The maximum element in a BST is the rightmost node, i.e., the right most leaf node in right subtree.
  (OR)
  A node in the right subtree having a left child but not having a right child.
- The maximum element in the below shown BST = 19.




```
struct Node *Maximum(struct Node *Root)
{
  if(Root==NULL)
     return NULL;
  else if(Root → r_child==NULL)
     return root;
  else
     return Maximum(root → r_child);
}
```

Time Complexity : O(n), worst case (when BST is a right skew tree).
Space Complexity : O(n) for a recursive stack.

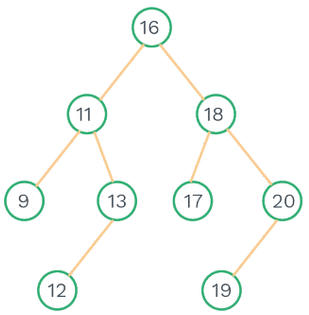
Inorder predecessor:

In BST, Inorder predecessor is the greatest element in the left subtree of a node.

Inorder successor:

In BST, Inorder successor is the smallest element in the right subtree of a node.

eg:

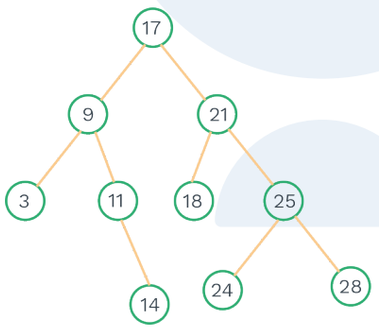


Inorder predecessor (16) = 13

Inorder successor (16) = 17

Insertion in BST

- We have to first search for the location to insert an element (node) into a Binary-Search-Tree.
- Find operation is used to find location of data to insert.
- **eg:** Insert an element 12 into the BST.



Search for element y = 12
12 < 17, go to the left subtree of 17
12 > 9, go to the right subtree of 9
12 > 11, go to the right subtree of 11
12 < 14, go to the left side of 14 and insert 12.

Time Complexity : O(n) ; when Tree is skewed Tree
Space Complexity : O(n) for recursive stack.
Space Complexity = O(1) ; for the iterative code.

Deletion in BST

- If a node to be deleted is a leaf node, then simply delete it. Deleting a non-leaf node is a complex task as other nodes may need to be shuffled after deletion. The process of deletion can be seen below:
- First search the location of the node to be deleted.

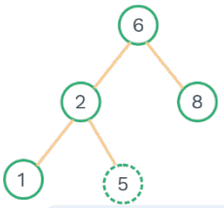
- Following are the cases to delete a node in BST:

For leaf node:

- To delete the node having no child, simply deallocate the memory and add the NULL pointer to its parent.

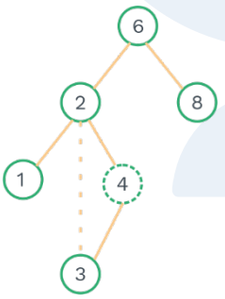
Example:

- If we have to delete a node 5, just delete it.



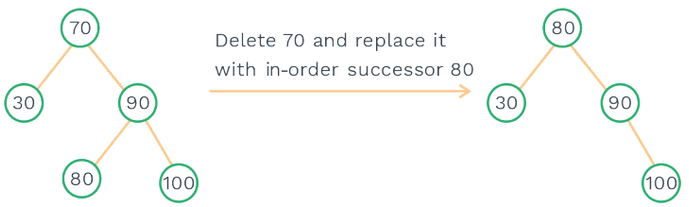
Node having one child:

- When we have to delete the node having only one child.
- Delete that node and point the node's parent-pointer to its children.



Node having two children:

When we have to delete a node having two children, delete that node and replace that with either Inorder successor (OR) Inorder predecessor.



Time Complexity : O(n), in worst case,
when Tree is skewed one.

Time Complexity: (logn), in best case.

Previous Years' Question

A program takes as input a balanced binary search tree with n leaf nodes and computes the value of a function g(x) for each node x. If the cost of computing g(x) is:

$$\min\begin{pmatrix}\text{number of leaf – nodes}\\ \text{in left – subtree of x}\end{pmatrix}\begin{pmatrix}\text{number of leaf – nodes}\\ \text{, in right – subtree of x}\end{pmatrix}$$

Then the worst–case time complexity of the program is?

a) $\theta(n)$ **b)** $\theta(n\log n)$ **c)** $\theta(n^2)$ **d)** $\theta(n^2\log n)$

Sol: b) **(GATE: 2004)**

Previous Years' Question

Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences CANNOT be the sequence of nodes examined?

a) {10, 75, 64, 43, 60, 57, 55}
b) {90, 12, 68, 34, 62, 45, 55}
c) {9, 85, 47, 68, 43, 57, 55}
d) {79, 14, 72, 56, 16, 53, 55}

Sol: c) **(GATE: 2006)**

Previous Years' Question

A data structure is required for storing a set of integers such that each of the following operations can be done in O(logn) time, where n is the number of elements in the set.
i) Deletion of the smallest element
II) Insertion of an element if it is not already present in the set

Which of the following data structures can be used for this purpose?
a) A heap can be used but not a balanced binary search tree.
b) A balanced binary search tree can be used but not a heap.
c) Both balanced binary search tree and heap can be used.
d) Neither balanced search tree nor heap can be used.
Sol: b) **(GATE: 2003)**

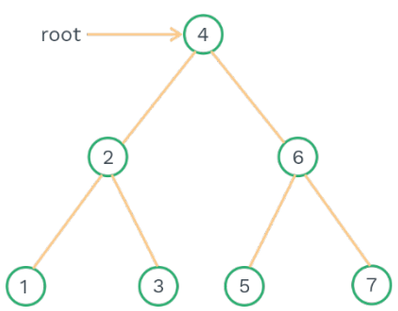
Balanced binary search trees

- Worst case time required for searching in a BST in O(n).
- It occurs when BST is either left-skewed or right-skewed.
- The worst-case time complexity can be reduced to O(logn) using the concept of balanced binary search tree.
- Suppose H(K) represents the height of a balanced-tree.
  Here, K=balance factor
  where Balance factor=Height of left subtree-Height of right subtree.

Full balanced binary search trees

- A binary search tree is known to be full balanced BSTs, if balance factor of each node = 0.

For example:



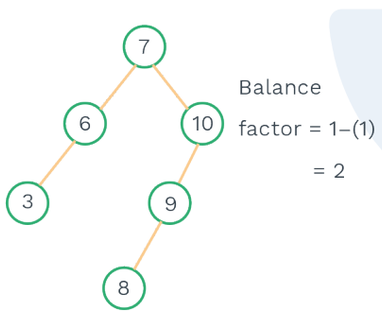
AVL (Adelson–Velskil and Landis) trees

Definition

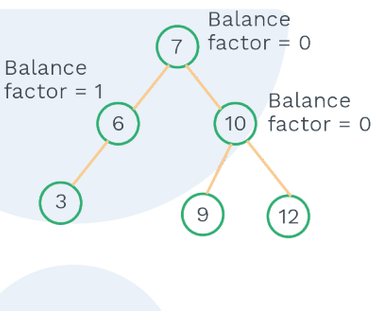
It is a height balanced binary search tree.
For each node K, the height of the left and right subtrees of K differ by at most 1. It means the balance factor of an AVL tree should be (-1,0,1).

What is balance factor?

It is the difference between heights of left sub tree and right sub tree. It means Balance factor(K) = Height of LST(K)-Height of RST(K).



NOT an AVL Tree as balance factor of each node is not following AVL – property.



An AVL Tree each node is following AVL–property. Balance factor of each node is either –1, 0 or 1.

Note:

Height of a Null Tree is –1.

Minimum/maximum number of nodes in AVL tree

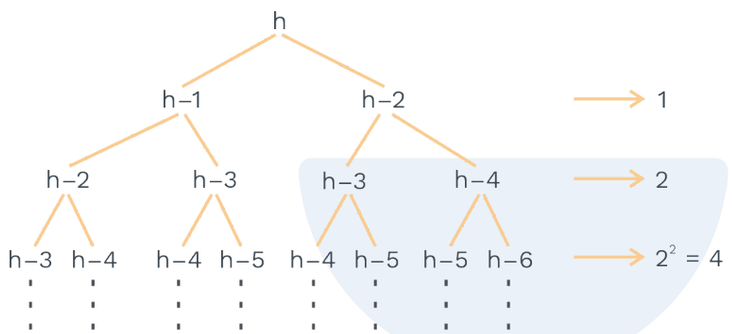
- Suppose N(h) denotes the number of nodes present in the AVL tree, where h denotes the height of an AVL tree.
- For minimum number of nodes having height=h, left subtree of an AVL tree should be of height (h-1) and right subtree should be of height (h-2). (OR)
- The left subtree of an AVL tree should be of height (h-2) and right subtree should be of height (h-1).

- Therefore, the recursive equation for minimum number of nodes of height h:

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

- Where N(h-1)= minimum number of nodes of height (h-1)
- N(h-2)= minimum number of nodes of height (h-2)

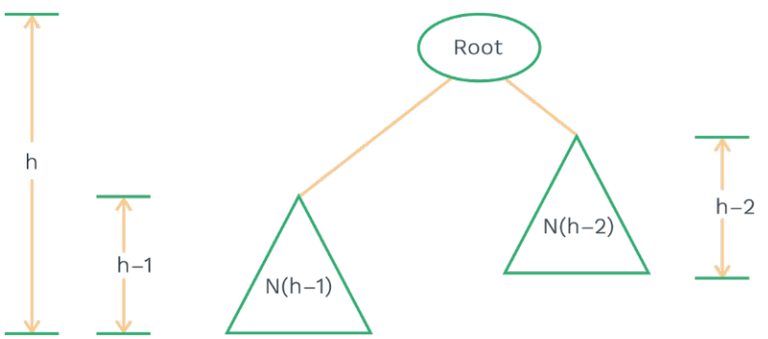
From the above equation:



$$n(h) = 2^0 + 2^1 + 2^2 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots + 2^{h-1}$$

$$n(h) = 2^h - 1$$

$$h = \theta(\log_2 n)$$



So, the maximum height of an AVL tree = O(logn)
For maximum number of nodes, the recursive equation, we get:
N(h)=N(h-1)+N(h-1) +1
=2N(h-1) +1
It is a case of full binary tree.
After solving the above recurrence relation, we will get:
height h = O(log n)

Note:

All above cases will lead to height of an AVL = O(logn), where n denotes the number of nodes.

AVL tree declaration

```
struct AT
{
  int d;
  struct AT *Lt ,*Rt;
  int h;
};
```

In the above AVL Tree Declaration, AT represents AVL Tree, d represents data, *Lt represents left pointer, *Rt represents right pointer and h represents height.

Height of an AVL tree

```
int h(struct AT *root)
{
  if(root==Null)
    return -1;
  else
    return root → h;
}
```

Time Complexity : O(1).

Rotation:

- When we insert an element to an AVL tree or delete an element from an AVL tree, the structure of tree will likely to get changed.
- As insertion/deletion of an element can result in an increase/decrease of the height of subtree by 1.
- To correct it, we need to check the balance factor of each node after every insertion/deletion of an element in an AVL-Tree and then rotation of tree is required.
- There are mainly two types of rotations to restore the properties of AVL-Tree:
  Single Rotation and Double Rotation.

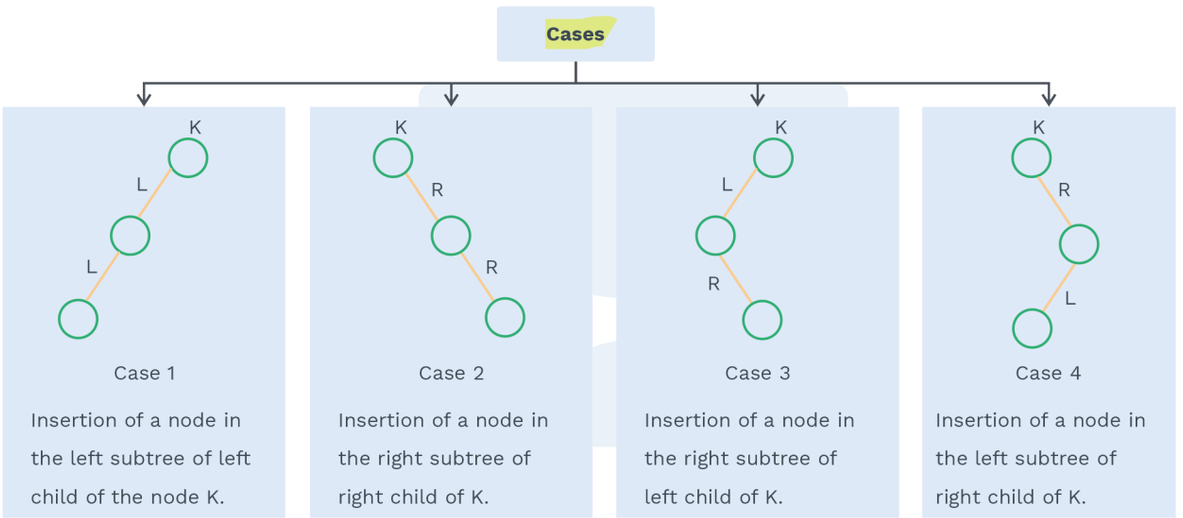
Observation

- When we insert a new node in an AVL tree, balance factor of those nodes get affected, which are present in the path from root to the insertion-point.

- So, after insertion of a new node, we need to check the balance factor of 1st node present in the path from insertion-point to root and so on.
- We need to restore the AVL-Tree property according to the type of violation.

Types of violation

- What type of violation can occur?
- Suppose K is the node, where the imbalance occurs due to different type of insertion.
- There are four cases possible:

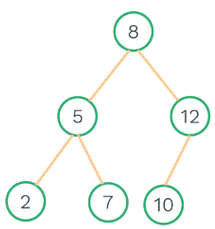


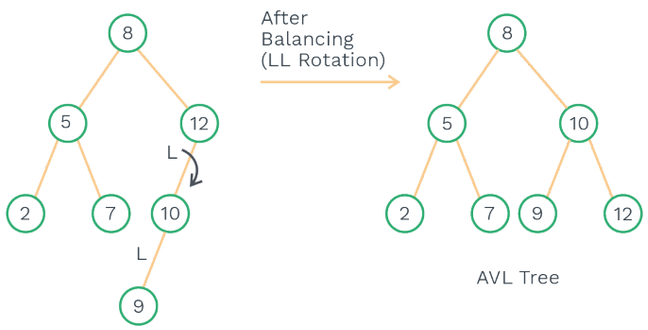
Balancing of a tree using rotation

Single rotations

Left left rotation (LL rotation):

- Suppose a new node 9 is inserted into the given AVL tree, an imbalance occurs as balance factor of some nodes get changed. We need to perform LL-Rotation in this case.
- Given AVL tree

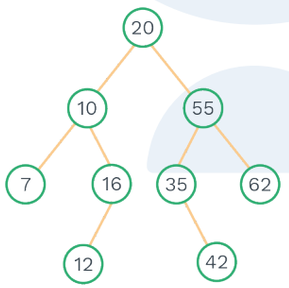




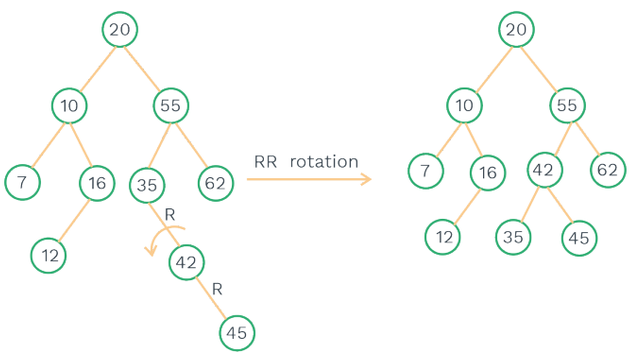
Time complexity: O(1)
Space complexity: O(1)

Right right rotation (RR rotation)

- When a new node 45 is inserted into the given AVL tree, an imbalance occurs, as balance factor of some nodes get changed.
  Given AVL Tree



Example:



Time complexity: O(1)
Space complexity: O(1)

Double rotation

Sometimes, Double rotation is required to restore the AVL-tree properties as single rotation is not enough to solve it.

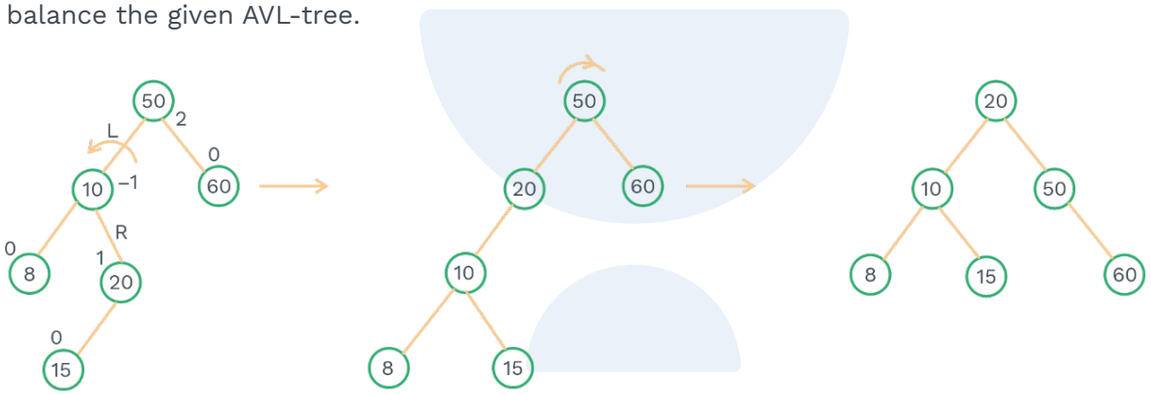
Left right rotation (LR rotation)

- LR Rotation is required when insertion occurs at the right subtree of the left child of a node.

Example

When a new node 15 is inserted, Imbalancing of AVL-tree occurs.

Here, we have to perform LR Rotation(double rotation) to balance the given AVL-tree.

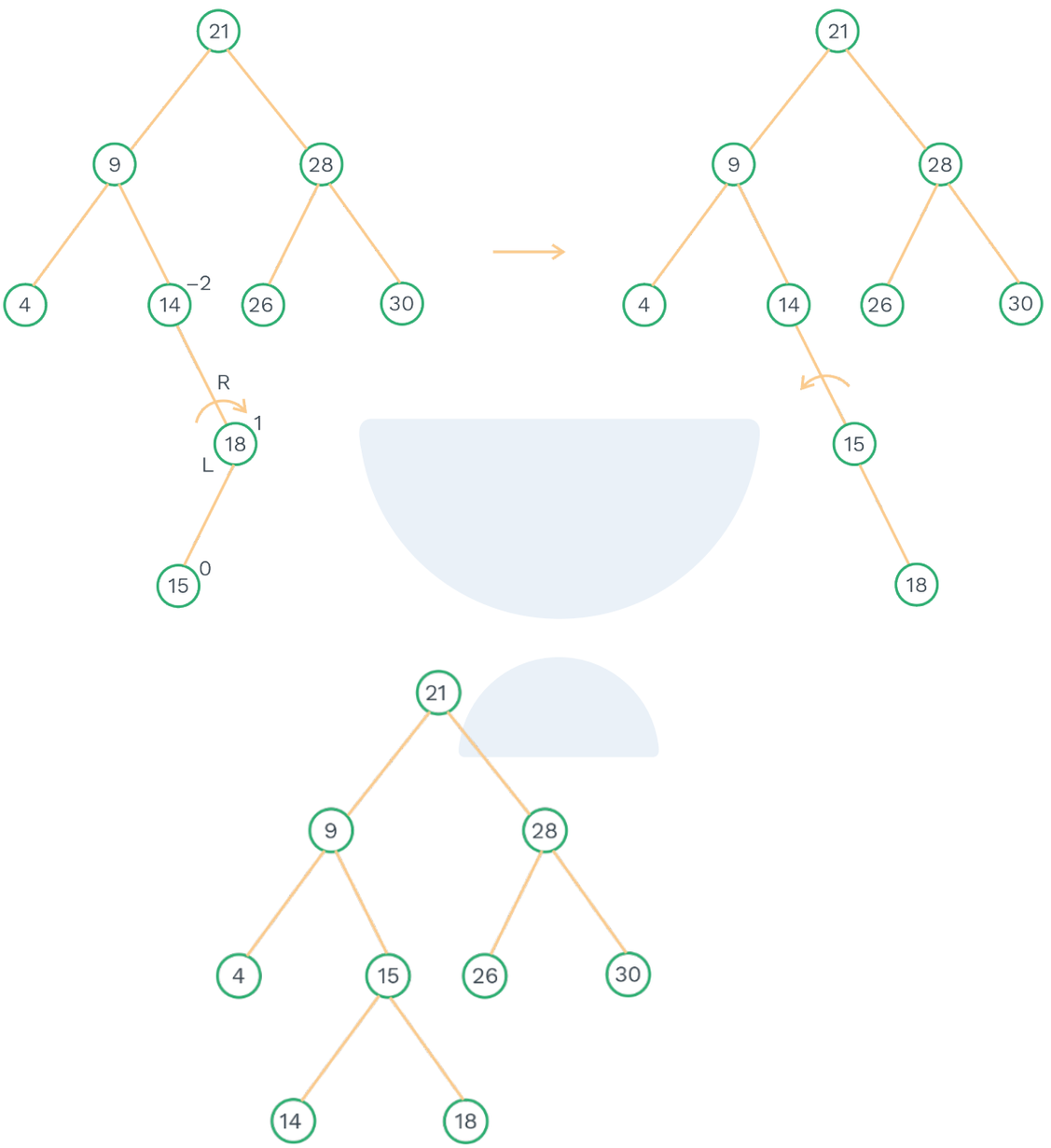


Right left rotation (RL rotation)

- RL Rotation is required when insertion occurs at the left subtree of the right child of a node.

Example:

- When a new node 15 is inserted, Imbalancing of AVL-tree occurs.
- Here, we have to perform RL Rotation(double rotation) to balance the given AVL-tree



Time and space complexity of insertion in an AVL-tree:

Time Complexity = O(log n)

Note:

i) In red-black tree and 2-3 tree an element can be searched in O (log n) time

ii) If "I" is the number of internal nodes of a complete n-ary tree, Then, the number of leaves present in it can be represented by I(n–1) + 1.

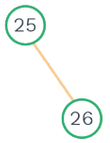
iii) So, if we have 'n' internal nodes of degree 2 in a binary tree, then we have (n+1) leaf nodes.

SOLVED EXAMPLES

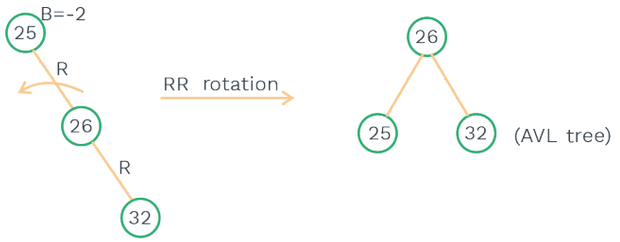
Q1 **Construct an AVL tree for the given sequence**
25, 26, 32, 8, 5, 13, 29, 17, 15

Sol: Step 1: Insert 25

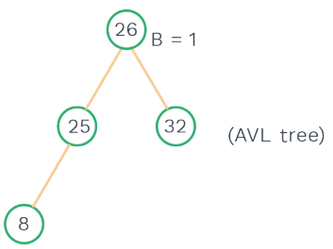
Step 2: Insert 26



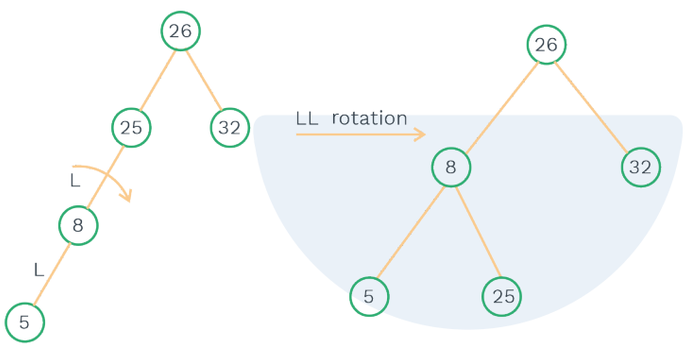
Step 3: Insert 32



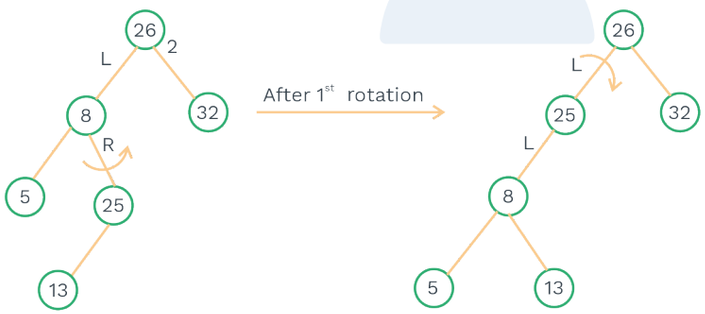
Step 4: Insert 8

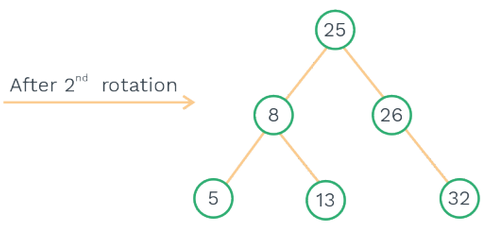


Step 5: Insert 5

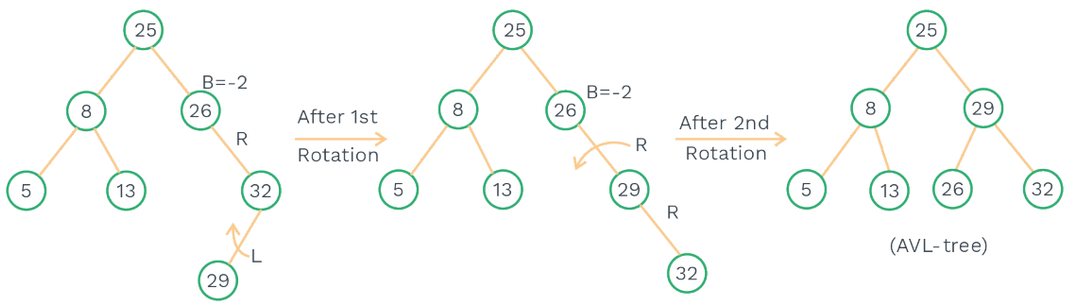


Step 6: Insert 13

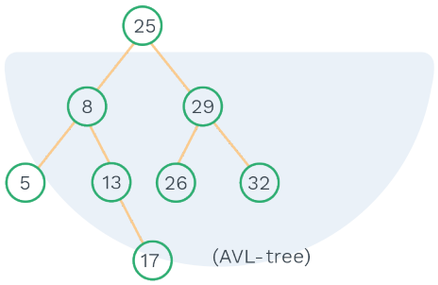




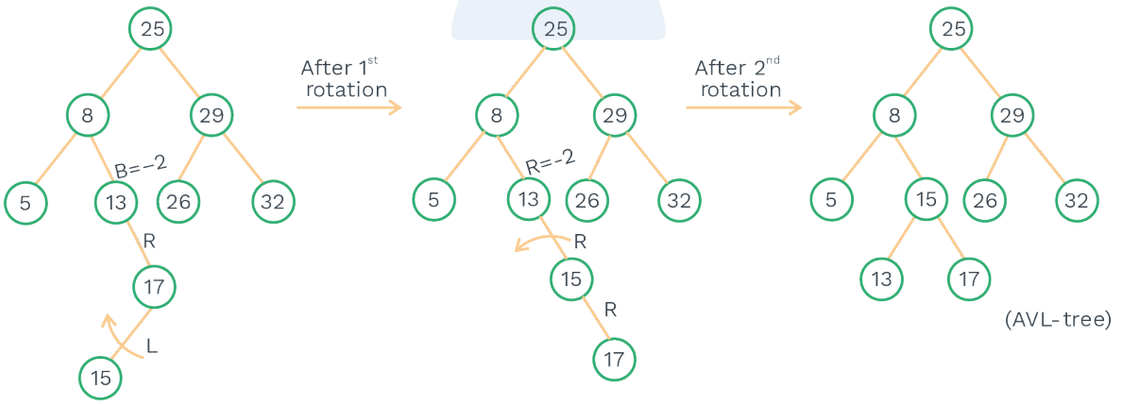
Step 7: Insert 29



Step 8 : Insert 17



Step 9 : Insert 15



Previous Years' Question

Which of the following is TRUE?

a) The cost of searching an AVL tree is θ(logn), but that of a binary search tree is O(n).

b) The cost of searching an AVL tree is θ(logn), but that of a complete binary tree is θ(nlogn).

c) The cost of searching a binary search tree is O(logn), but that of an AVL tree is θ(n).

d) The cost of searching an AVL tree is θ(nlogn), but that of a binary search tree is O(n).

Sol: a) **(GATE: 2008)**

Heap

- Heap is a nearly complete binary tree.
- It means, all the leaf nodes must be either at level h or (h-1).
- The main property of a heap is that the value of each parent node should be either: less than equal to its children(min-heap)
  OR
  greater than equal to the value of its children(max-heap).
- Heap takes less time for inserting an element, for finding minimum in case of min-heap, for finding maximum element in case of max-heap, for deleting an element etc.
- The time complexity details of different data-structures for different operations are as shown below:

Data structure	Insertion	Search	Find Min.	Delete Min.
Unsorted array	O(1)	O(n)	O(n)	O(n)
Sorted array (increasing order)	O(n)	O(log n)	O(1)	O(n)
Unsorted linked list	O(1)	O(n)	O(n)	O(n)
Min-heap	O(log n)	O(n)	O(1)	O(log n)

Unsorted array:

- **Insertion:** In an Unsorted array, insertion can happen at the end of the array, so it will take O(1) as time complexity.
- **Seaching:** In an unsorted array, to search for an element, we need to visit all the elements in the array, i.e., linear search. Hence, its time complexity is O(n).
- **Finding minimum element:** Similarly, to find the minimum element in an unsorted array, we might need to visit all the elements in the array in worst case, i.e., linear search. Hence, its time complexity is O(n).
- **Deletion of minimum element:** To delete a minimum element in an unsorted array, we first have to find the minimum element, which we want to delete and then after deleting that element, remaining (n-1) elements need to be moved in worst case given n number of elements in an unsorted array.

  Total complexity = O(n) +O(n) =O(n).

Sorted array (Let 'n' be number of elements in increasing order)

- **Insertion:** Insertion takes O(n) time. Since it is a sorted array, we need to place the element in its correct position. After that, in the worst case, we need to move the remaining (n–1) elements, so, the overall time complexity is O(n).
- **Seaching:** Since it is a sorted array, we can apply binary search, and it will take time complexity as O(logn).
- **Finding minimum element:** Since the sorted array is in an increasing order the 1st element is the minimum element. Hence, O(1) is the time complexity to find a minimum element.
- **Deletion of minimum element:** Now, if we want to delete the minimum element, we need to move remaining (n–1) element ahead. Hence the time complexity is O(n).

Unsorted linked list (having 'n' element)

- Insertion: Insertion can happen in O(1), because it is an unsorted linked list.
- Searching: To search an element, we need to visit every element of a list. So, In worst case its time complexity is O(n).
- Finding minimum element: To find the minimum element, we need to visit every element of the list in worst case, as linked list is unsorted. Hence, its time complexity is O(n).
- Deletion of minimum element: To delete minimum, finding minimum takes O(n), and deleting it will take O(1) time.

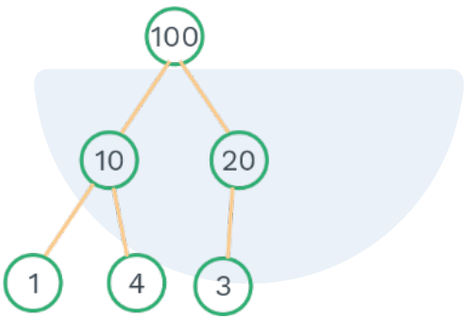
Min-heap, (having 'n' element)

- The time taken by min heap to insert and delete an element in min heap =O(log n).

- In min heap, first element (root) is the smallest element; thus, the time taken by min-heap to find a minimum element = O(1)
- In the worst case, searching an element in a heap requires to visit all the nodes once.
  So, time complexity= O(n)

Max-heap:

Definition: Max-heap is a complete binary tree, where the root value is maximum. It means, the root element must be greater than all the elements present in the left subtree and right subtree.

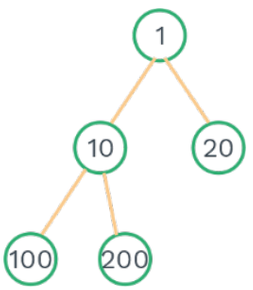


Min-heap:

Definition: Min–heap is a complete binary tree, where the root value must be smallest.

- It means the root element should be smaller than all the elements present in the left subtree and right subtree.

Example:



Array representation of a complete binary tree

- Heap can be represented using the arrays.
- As we know, Heap is nearly a complete binary tree. This method to represent the heap does not waste any space(location).

- Suppose all the elements are stored in array starting from index 0.
- In this case, the representation of the previous given heap can be given as:

1	10	20	100	200
0	1	2	3	4

- The index of a parent for a child at index 'i' is $\lceil i / 2 \rceil - 1$.
- Left child of the ith element is at 2*i + 1.
- Right child of the ith element is at 2(i+1).
- Suppose the index of the array starts at 1, then
- The index of a parent for a child at index 'i' is $\left\lfloor \frac{i}{2} \right\rfloor$.
- Left child of the ith element is 2*i.
- Right child of the ith element is 2*i+1.

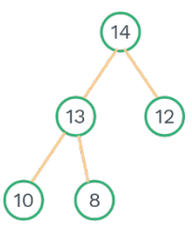
Note:

Here, multiplication with 2 indicates the left shift, and division with 2 implies the right shift. It is easy to implement like this.

SOLVED EXAMPLES

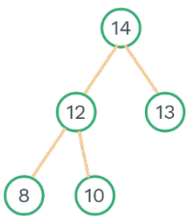
Q1 **Given the array of element check whether it is binary max–heap or not?**
a) 14, 13, 12, 10, 8
b) 14, 12, 13, 8, 10
c) 14, 13, 8, 12, 10
d) 14, 13, 12, 8, 10
e) 89, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70

Sol: **a)**



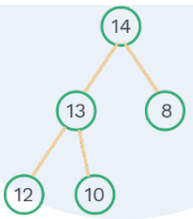
It is a binary max-heap, because all the nodes are satisfying the heap property. Here, A.heapsize is 5.

b)



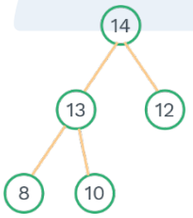
This is also a binary max-heap, because all the nodes are satisfying the heap property. Here, A.heapsize is 5.

c)



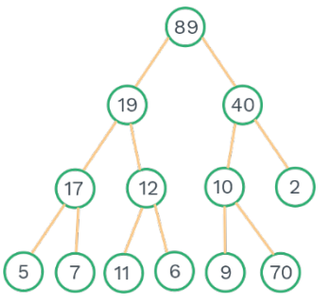
This is also a binary max-heap, because all the nodes are satisfying the heap property. Here, A.heapsize is 5.

d)



This is also a binary max-heap, because all the nodes are satisfying the heap property. Here, A.heapsize is 5.

e)



This is not a max-heap, because 70 is larger than the parent node, which is 10.

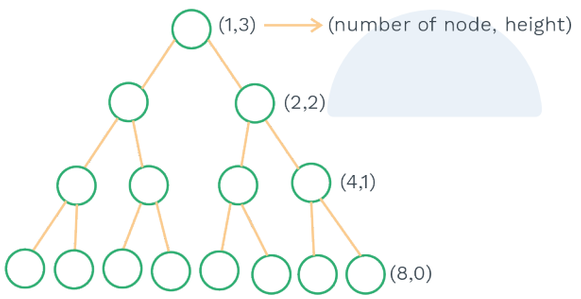
Note:

- Every leaf is a heap.
- For a given set of numbers, there may be more than one heap.
- If an array is in decreasing order, then it is always a max–heap.
- If the array is in ascending order, then it is always a min–heap.
- The starting index of leaf nodes of a complete binary tree having 'n' nodes is, $\left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \text{ to } n\right)$ and the number of non–leaf is $\left(1 \text{ to } \left\lfloor \frac{n}{2} \right\rfloor\right)$.

Note:

Maximum number of nodes at a given level of height 'h' of complete binary tree (having n number of nodes) is $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Example:



Maximum number of nodes at height = 0

$$= \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

$$= \left\lceil \frac{15}{2^{0+1}} \right\rceil == \left\lceil \frac{15}{2} \right\rceil = 8$$

So, at most eight nodes will be at height '0' in this particular tree.

Heapifying an element

- Sometimes, when we insert an element into a heap, it might violate the property of heap.

- Heapifying is the process of maintaining the heap property (i.e., every node satisfies the property of the heap according to the max(min) heap).
- For example, to heapify an element in a max-heap:
- First, find its child with maximum value and swap it with the current node.
- Continue this process as long as all the nodes satisfy the heap-property.
- MAX-HEAPIFY(A, i), where A is the heap and 'i' is the index of element, where insertion happens.
- This function can be applied, only if the left subtree and the right subtree tree are to be max-heap.

```
Max-Heapify (A, i)
{
    l = 2i; // 'l' indicates left child index
    r = 2i + 1; // 'r' indicates right child index
    if (l ≤ A.heapsize and A[l] > A[i])
            largest = l;
    else
            largest = i;
    if (r <= A.heapsize and A[r] > A[largest])
            largest = r;
    if (largest ≠ i)
            exchange A[i] with A[largest]
            Max-Heapify (A, largest)
}
```

- At each step, the largest of the elements A[i], A[LEFT(i)] and A[RIGHT(i)] are determined, and its index is stored in largest.
- If A[i] is largest, then the subtree rooted at node i is already a max-heap, and the procedure terminates.
- Otherwise, one of the two children has the largest element, and A[i] is swapped with A[largest], which causes node i and its children to satisfy the max–heap property.
- The node indexed by largest, however, now has the original value A[i], and thus the subtree rooted at largest might violate the max-heap property.
- Consequently, we call MAX-HEAPIFY recursively on that subtree.
- The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the q(1) time to fix up the relationships among the elements A[i], A[LEFT(i)], and A[RIGHT(i)], plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs).

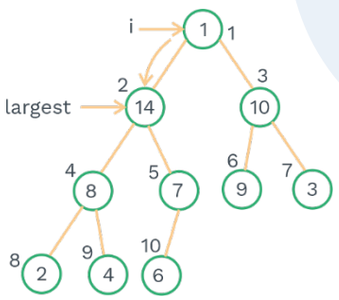
- The children's subtrees have a size at most 2n/3. The worst case occurs when the bottom level of the tree is exactly half full, and therefore, we can describe the running time of MAX-HEAPIFY by the recurrence

$$T(n) \leq T(2n/3) + \theta(1)$$

- The solution to this recurrence, by case 2 of the master theorem, is T(n) = O(logn).
- Alternatively, we can characterise the running time of MAX-HEAPIFY on a node of height "h" as O(h).

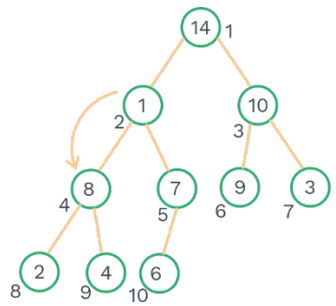
Example:
Let us assume that node 1 is at the root as shown below:

Step: 1



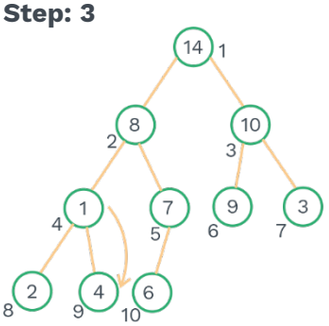
⇒ Swap (14, 1). Since, 14 is the largest value.

Step: 2



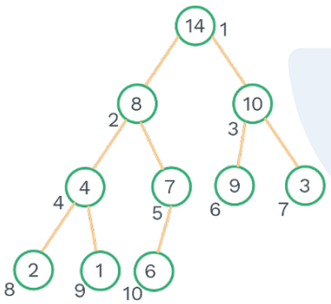
⇒ Swap (8, 1). Since, 8 is the largest element.

Step: 3



⇒ Swap (4, 1). Since, 4 is the largest number.

Step: 4



Previous Years' Question

What is the maximum height of any AVL tree with 7 nodes? Assume that the height of a tree with a single node is 0.

a) 2 **b)** 3
c) 4 **d)** 5
Sol: b) **(GATE: 2017)**

Space complexity:

- Since, it is a recursive function and for worst case (log n), a recursive call can be made.
- So, the function is called the number of level times.
- Number of levels will be logn in worst case.
- So, space complexity = O(log n)

Note:

i) In worst case, the number of recursive call = number of levels.
ii) Average space complexity = θ(log n)
iii) Best case space complexity = Ω(1)

Build max-heap

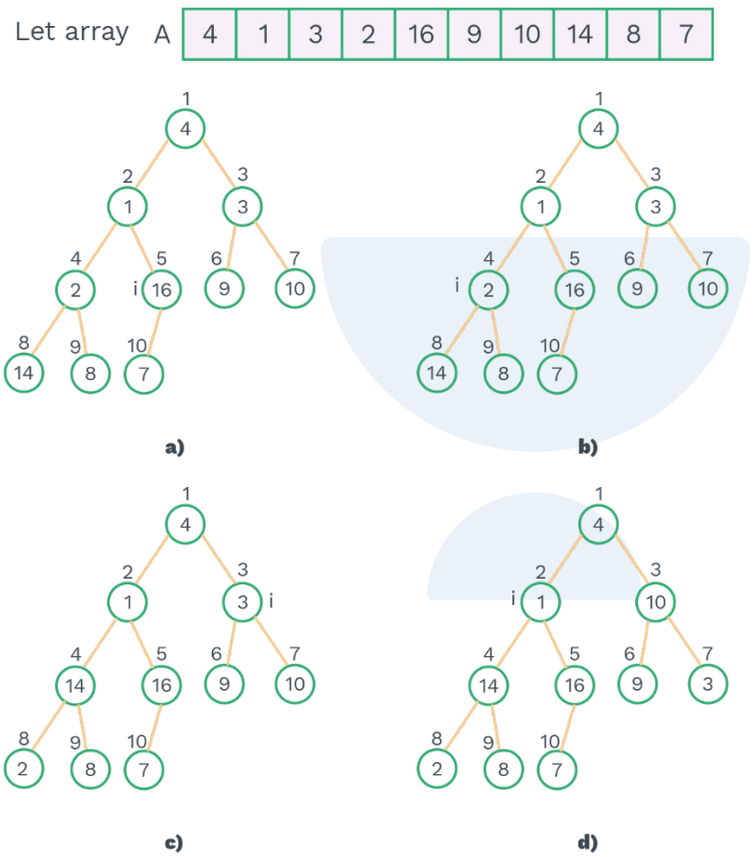
- To convert a given array A[1....n] into a max heap, Max-Heapify procedure is used in bottom to up fashion.

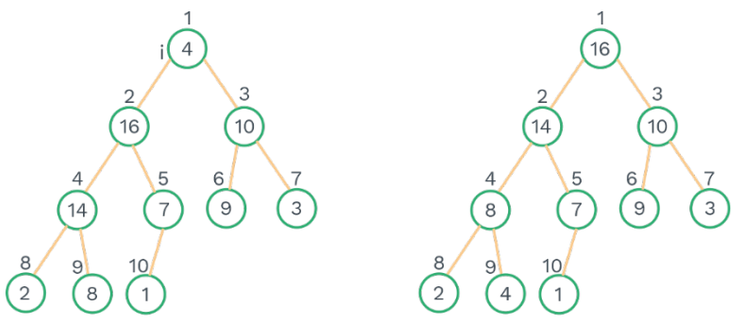
BUILD-MAX-HEAP(A)

1) A.heap–size = A.length
2) for i = ⌊A.length / 2⌋ downto 1
3) MAX-HEAPIFY (A, i)

Note:

As all the leaf nodes are by default heap.
So, we have to start the iteration from last non-leaf node to node index 1(initial).





e) f)

- As we can see in the above figures that index i points to node 5 first and then call Max-Heapify.
- In the next iteration, index i will point to 4 and so on.
- Final result we get is Figure (f) which is final max-heap.

Time complexity:

- Time taken by the Max-Heapify procedure when called on a node having height 'h' = O(h)
- Therefore, the total time taken by Build-Max-Heap is as:

$$\sum_{h=0}^{\log n}\left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \quad \left(\begin{array}{l}\text{Here, 'n' is the total number of nodes in}\\ \text{a tree, where h is height of node.}\end{array}\right)$$

$$\sum_{h=0}^{\log n}\left\lceil \frac{n}{2^{h}.2} \right\rceil (ch)$$

$$= \frac{cn}{2}\sum_{h=0}^{\log n}\left(\frac{h}{2^{h}}\right) \ldots\ldots\ldots (1)$$

$$= O\left(\frac{cn}{2}\sum_{h=0}^{\infty}\frac{h}{2^{h}}\right) \ldots\ldots\ldots (2) \quad \left(\begin{array}{l}\because \text{eq.}(2) \text{ is greater than eq. } (1),\\ \text{that's why we are putting Big Oh}\end{array}\right)$$

$$= O(n) \quad \left[\because \sum_{h=0}^{\infty}\frac{h}{2^{h}} = 2\right] (\text{this is a harmonic propagation}).$$

So, time complexity is O(n).

Space complexity:

- Build max heap space complexity will be same as space complexity taken by max-heapify (A,i).
- It will take maximum, when it is called from root.
- That's why the space complexity is O(log n).

Deletion of Maximum element from max-heap

```
Heap_Delete_Max(A)
{
  if(A.heap–size < 1)
     printf ("Heap Underflow");
  max = A[1];
  A[1] = A[A.heap-size];
  A.heap–size = A.heap–size – 1;
  MAX–HEAPIFY (A, 1);
  return max;
}
```

- Here, space complexity and time complexity will be for MAX-HEAPIFY (A,1) only.
- So, time and space complexity are O(log n).

Increase Key in max-heap

Let 'A' be the array, and 'i' be the index, whose value we want to increase to key.

```
Heap_Increase_Key (A, i, key)
{
  if (key < A[i])
     printf ("error");
  A[i] = key;
  while (i > 1 && A[i/2] < A[i])
  {
      exchange A[i/2] and A[i]
      i=i/2;
   }
}
```

Time complexity:

- In worst case, the leaf node will get increased, and then Heap_Increase_Key (A, i, Key) will get called from leaf.
- That's why it has to go till the root node. So, it will take O(log n).

Space complexity:

- Heap_Increase_Key(A, i, key) does not require any extra space. Hence, its space complexity is O(1).

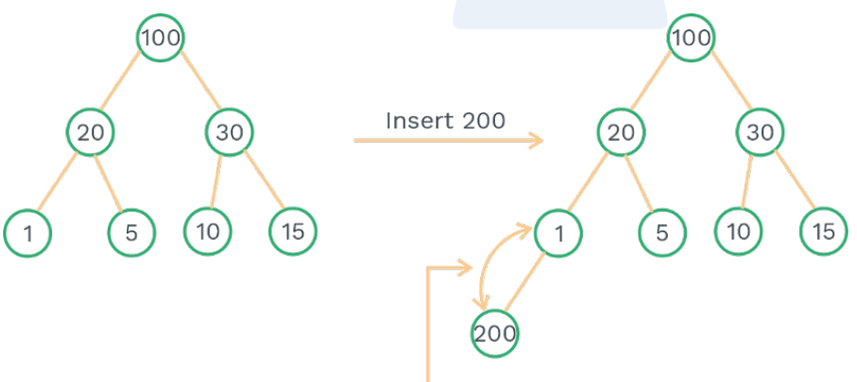
Note:

- For Heap_decrease_Key( ) on max–heap, we have to just call the max–heapify at that index, where decrease key happen, and after that we will get a heap.
- Space Complexity of Heap_decrease_Key( ) is O(1).
- Time Complexity of Heap_decrease_Key( ) is O(log n).

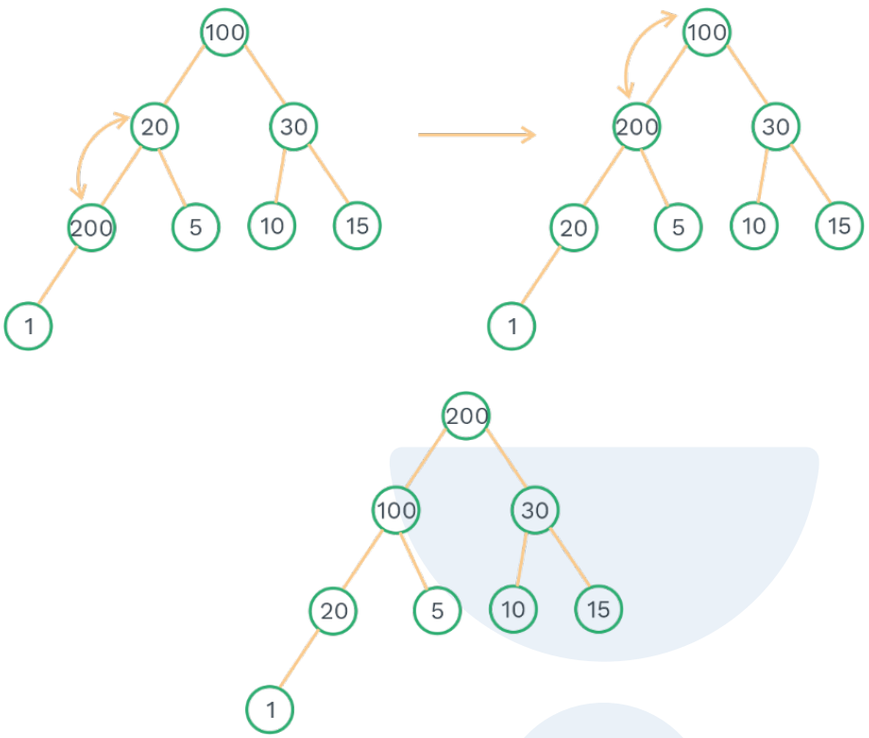
Insert an element in max-heap:

- Whenever we want to insert an element in max–heap, then insert the element at last index (position).
- After insertion, compare it with parent, and if parent is lesser than new element, then swap it.
- We will continue this procedure, until either we reach at root, or if parent is greater than the child.

Example:



Since, newly inserted element is greater. So, swap (200, 1).



- So, in worst case, the newly inserted element will have to traverse from the leaf node to the root.
- Hence, time complexity is O(log n).
- We can even use the max_heap_increase_key (A, i, Key) to implement the insertion into max-heap.

Procedure:

i) At new node we will insert an element $-\infty$. ($-\infty$ is some element, which could be very very smaller than all the elements).

ii) Since-infinity is there, then increasing that node from -infinity to newly inserted node and to do that we called increased key and we got the max-heap.

Finding minimum element in max-heap (having 'n' elements)

- Finding a minimum element will take O(n), because minimum will be present at leaf.
- The starting index of leaf node of heap having n nodes is

$$\left(\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \text{to } n\right).$$

- So, minimum element will be in from $\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right)$ to n.
- So, at most we need to search $\frac{n}{2}$ element to find minimum. Hence, the time complexity is $O\left(\frac{n}{2}\right) = O(n)$.

Search an element in Heap

- To search an element in Heap, we may need to visit every element.
- Let us assume number of elements in heap are n. Then, time complexity to search an element is O(n).

Delete an element in Heap

- To delete an element, first we need to search an element.
- To search an element, we require O(n) as time complexity.
- After deletion of an element, the tree may violate the Heap condition. So, again we need to call heapify function.
- Hence, its time complexity is O(n).

Time complexity of max-heap with different operations

Operations in max-heap	Time complexity
Find Maximum	O(1)
Delete Maximum	O(log n)
Insert an Element	O(log n)
Key Increase	O(log n)
Decrease Key	O(log n)
Search a random element	O(n)
Find minimum	O(n)
Delete an element	O(n)

- Similarly, we can have all these operations on min-heap.

Heapsort

- In this algorithm, we are swapping the root element with the last element in the heap, and then reducing the heapsize by 1. After that, we are applying the Max-heapify in the root.
- In this way, we can sort the Heap.

Pseudocode:

```
Heap_Sort (A) /* A is the array having 'n' elements.
{
  Build-Max-Heap(A)
  for (i = A.length down to 2)
  {
     exchange A[1] with A[i]
     A.heapsize = A.heapsize – 1
     Max-Heapify (A, 1)
   }
}
```

Time complexity:

In Heapsort,

Step 1: Building max heap takes O(n) time.

Step 2:

Swapping of A[1] with A[n]

A.heapsize = A.heapsize – 1

Max-Heapify(A,1)

All these lines will take time as O(log n) in one iteration due to Max-Heapify.

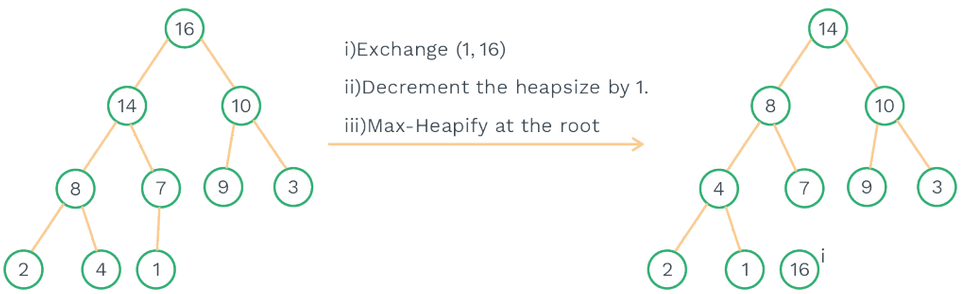
For n nodes(elements), it will take overall O(n log n) time.

Overall time complexity

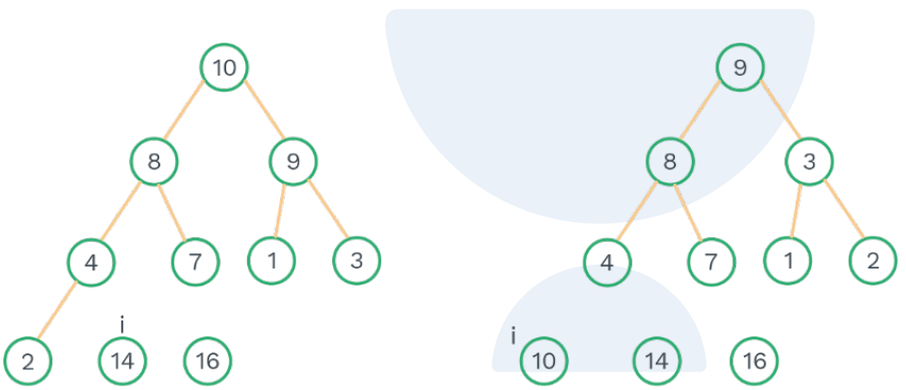
=O(n) + O(n log n)

=O(n log n)

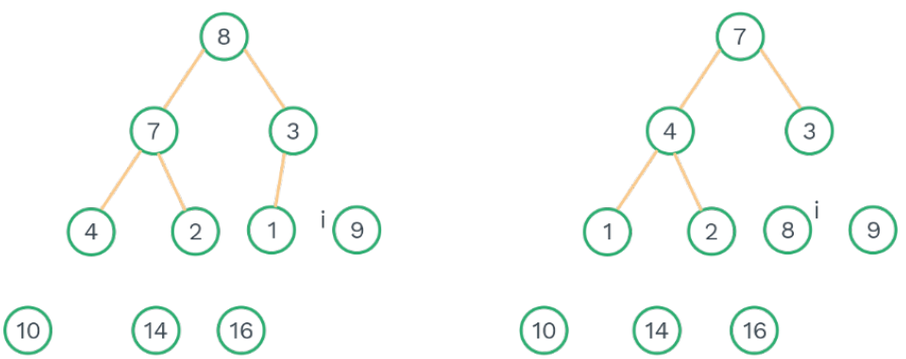
Note: In the below example of heapsort, we have already built the heap.


16
14
10
8
7
9
3
2
4
1
i)Exchange (1, 16)
ii)Decrement the heapsize by 1.
iii)Max-Heapify at the root
14
8
10
4
7
9
3
2
1
16
i

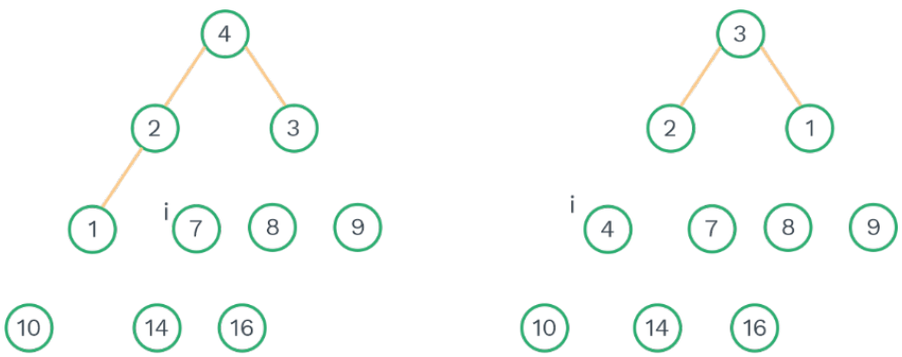
a) **b)**


10
8
9
4
7
1
3
i
2
14
16
9
8
3
4
7
1
2
i
10
14
16

c) **d)**

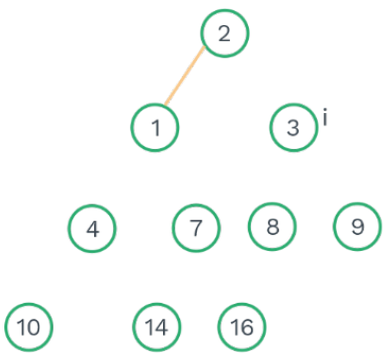

8
7
3
4
2
1
i
9
10
14
16
7
4
3
1
2
8
i
9
10
14
16

e) **f)**



g)

h)



i)

j)

(K)

- In the above figures, Figure(a) shows the max-heap after the call of Build-Max-Heap.
- Figures (b) to (j) show swapping of A[1] with A[i] and decrement of heapsize each time and the position of i on the max-heap after each call of Max-Heapify.

Previous Years' Question

Consider the process of inserting an element into a Max Heap, where the Max Heap is represented by an array. Suppose we perform a binary search on the path from the new leaf to the root to find the position for the newly inserted element, the number of comparisons performed is:

a) $\theta(\log_2 n)$ **b)** $\theta(\log_2 \log_2 n)$

c) $\theta(n)$ **d)** $\theta(n \log_2 n)$

Sol: b) **(GATE: 2007)**

Previous Years' Question

We have a binary heap on n elements and wish to insert n more elements (not necessarily one after another) into this heap. The total time required for this is

a) $\theta(\log n)$ **b)** $\theta(n)$

c) $\theta(n \log n)$ **d)** $\theta(n^2)$

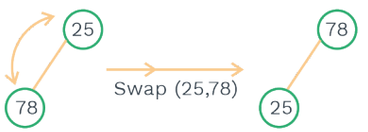
Sol: b)

SOLVED EXAMPLES

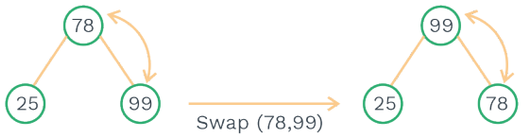
Q1 **Draw binary max-heap for the following given sequence:**
25, 78, 99, 98, 82, 100, 102

Sol: Step 1: Insert 25

Step 2: Insert 78

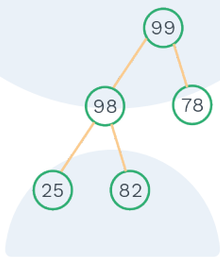


Step 3: Insert 99

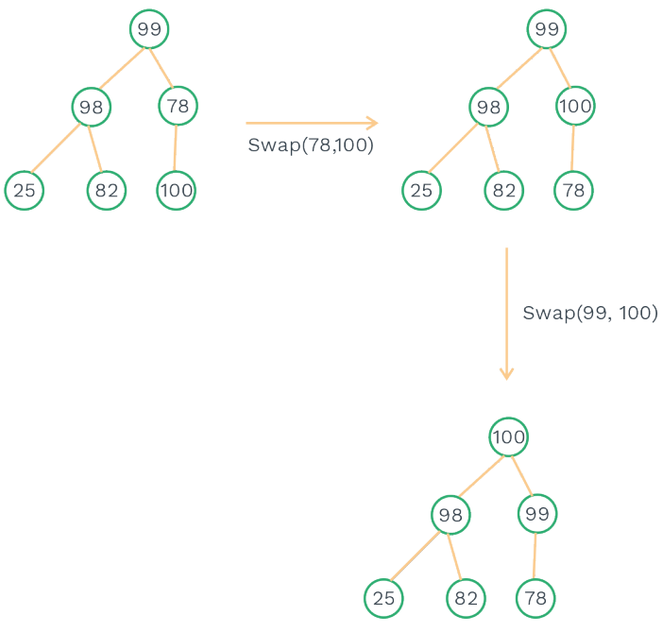


Step 4: Insert 98

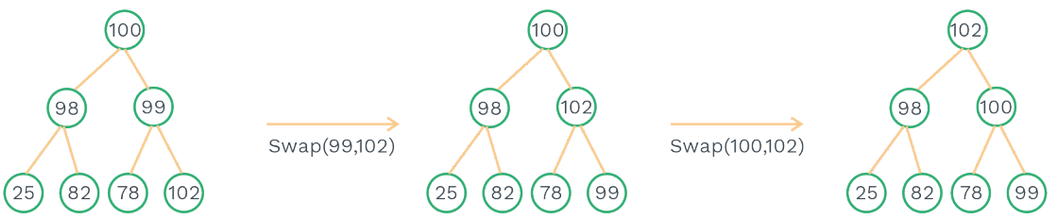
Step 5: Insert 82



Step 6: insert 100



Step 7: Insert 102

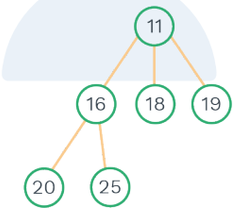


Q2 **Which of the following sequence represents ternary max-heap:**
a) 11,16,18,19,20,25
b) 15,17,25,30,35,5
c) 20,4,2,5,7,1
d) 40,10,25,30,9,2

Sol: **d)**

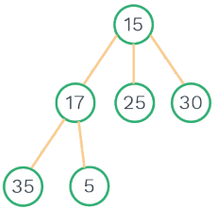
Explanation: Since, it is a ternary max-heap. So, it will have 3 children.

Option a):



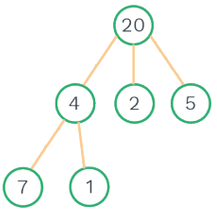
It is not a max-heap. It is a min-heap.

Option b):



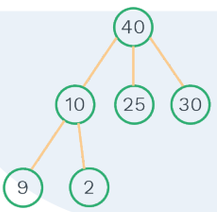
Here, node 25 is greater than node 15. Therefore, it is not a max-heap.

Option c):



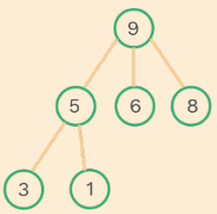
Here, node 7 is greater than node 4.
∴ It is not a max-heap.

Option d):

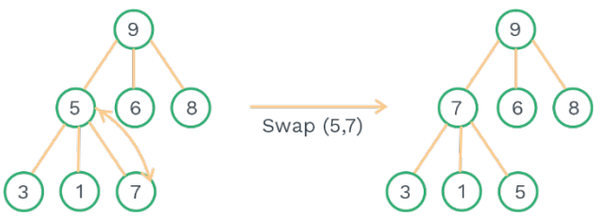


It is a max-heap.

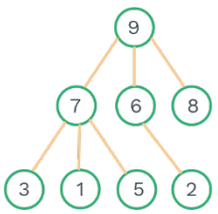
Q3 **Insert 7, 2, 10, 4 in that order in the ternary max heap as shown below:**



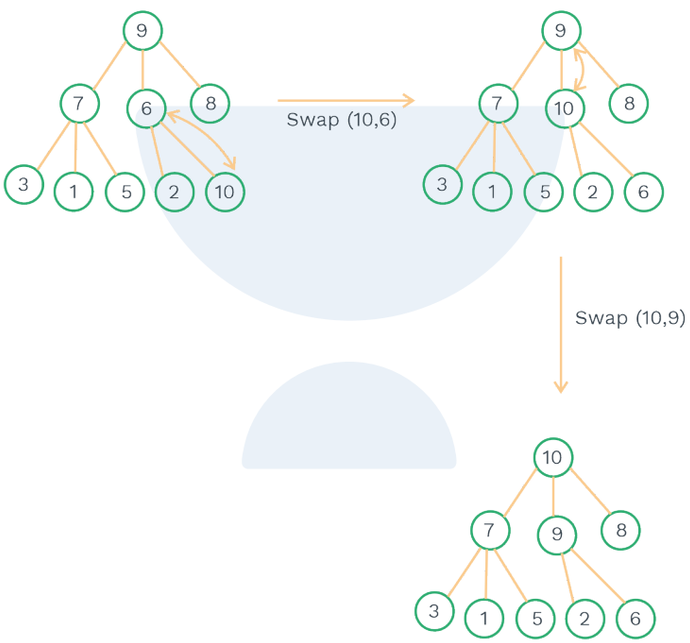
Sol. Step 1: Insert 7



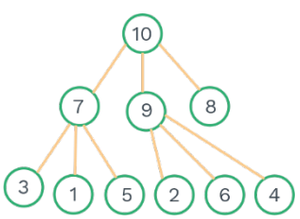
Step 2: Insert 2



Step 3: Insert 10



Step 4: Insert 4



Q4 **Number of binary tree with 4 nodes A, B, C and D which is having preorder as ABCD.**

Sol: Number of structure with 'n' nodes = $\frac{^{2n}C_n}{(n+1)}$

So, with 4 nodes, number of structure = $\frac{^{8}C_4}{5}$

$$= \frac{\frac{8!}{4!4!}}{5}$$

$$= \frac{8 \times 7 \times 6 \times 5 \times 4!}{4! \times 4! \times 5}$$

$$= \frac{8 \times 7 \times 6 \times 5}{4 \times 3 \times 2 \times 1 \times 5} = 14$$

Since, each structure will represent one particular preorder, i.e, ABCD.
So, 14 is the number of binary tree with 4 nodes A, B, C and D which is having preorder as ABCD.

Q5 **Given an inorder as 4, 12, 16, 27, 29, 34, 44, 50, 52, 65, 77, 88, 92, 93 and pre-order as 50, 27, 16, 4, 12, 34, 29, 44, 88, 65, 52, 77, 93, 92 of binary tree. What is postorder?**
a) 12, 4, 16, 29, 44, 34, 27, 52, 77, 65, 92, 93, 88, 50
b) 29, 44, 16, 4, 12, 34, 27, 52, 77, 65, 92, 93, 50, 88
c) 12, 4, 16, 29, 34, 44, 27, 52, 77, 65, 92, 93, 88, 50
d) 12, 4, 16, 29, 44, 34, 27, 52, 77, 65, 92, 93, 50, 88

Sol: **a)**
Explanation:
Preorder: (Root,Left,Right)
Visit Root, Visit Left subtree, Visit Right subtree

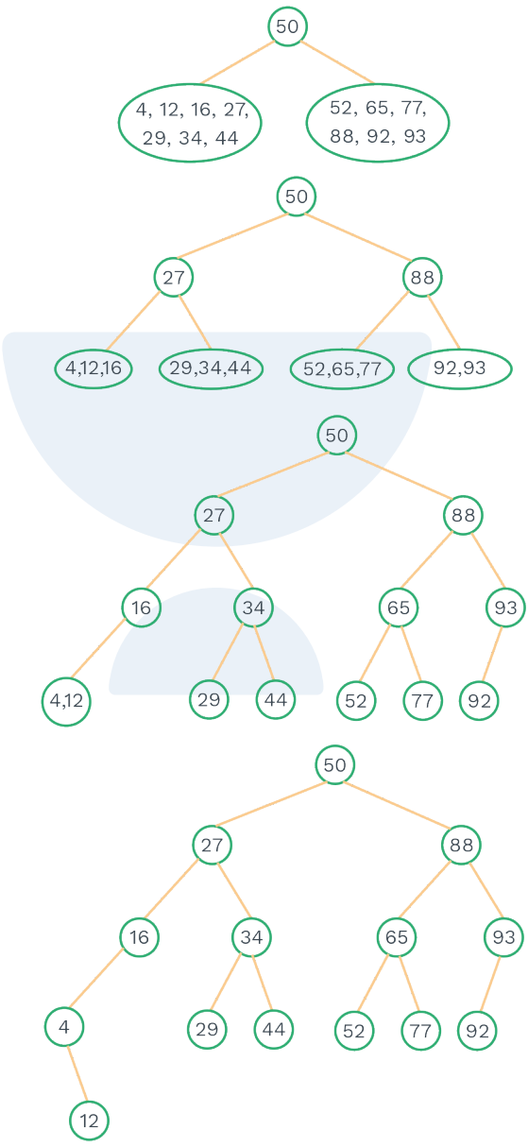
Inorder: (Left,Root,Right)
Visit Left subtree, Visit Root, Visit Right subtree
So, according to preorder, 50 is the root node.

Inorder: (Left, Root, Right)

4, 12, 16, 27, 29, 34, 44, 50, 52, 65, 77, 88, 92, 93

Preorder: (Root, Left, Right)

50, 27, 16, 4, 12, 34, 29, 44, 88, 65, 52, 77, 93, 92



So, the postorder is 12, 4, 16, 29, 44, 34, 27, 52, 77, 65, 92, 93, 88, 50.
We just traversed the tree from top to down, left to right.
Whenever we visit a node for the last time, we print it.

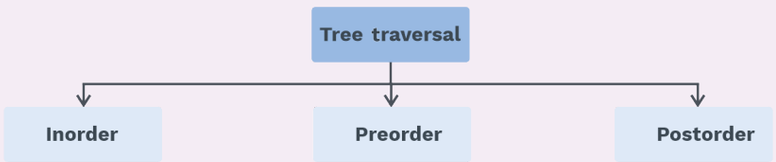
Q6 **Suppose the nodes in BST are given as:**
50,30,80,20,48,60,90,10,15
What will be the Inorder traversal of BST:
a) 10,15,20,30,48,50,60,80,90
b) 15,20,30,10,50,48,80,90,60
c) 10,15,20,30,50,48,60,80,90
d) None of above

Sol: **a)**

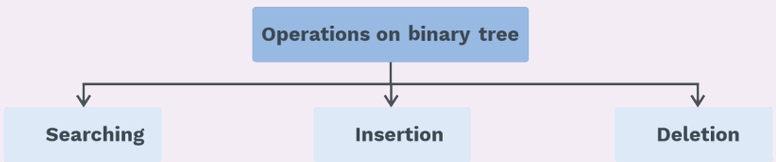
Explanation:
Inorder traversal of a binary search tree is in ascending order (Sorted order) of key value of nodes.
So, a) is the correct option.

Chapter Summary

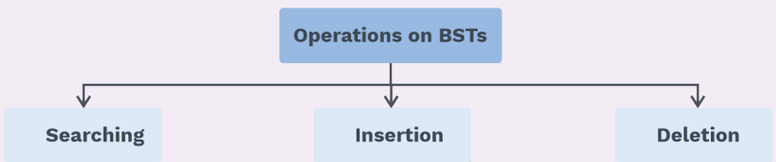
- **Tree:** It is an example of non-linear data structures.
- **Binary trees:** A tree in every node is having atmost 2 children (i.e. either 0 child or 1 child or 2 children).
- **Tree traversal:**



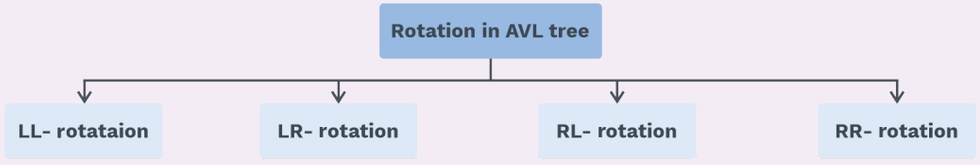
- **Operations on binary tree:**



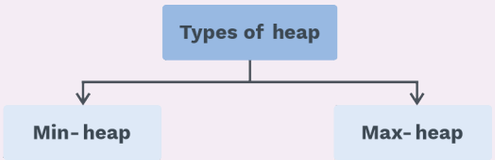
- **Binary seach tree:** It is used for searching.
- In BSTs, the element present in the left subtree should be less than the root element, and the element present in the right subtree should be greater than the root element.
- Operations on BSTs:



- **AVL-tree:** It is a height balanced-tree.
- Balance factor of each node in AVL tree must be either -1 or 0 or 1.
- Rotation in AVL tree:



- **Heap:** It is a complete binary-tree, which takes O(log n) time to insert an element.



- **HeapSort:**
- First swap the root element with the last element in the heap.
- After that decrease the heapsize by 1
- Then perform max-heapify on the root.