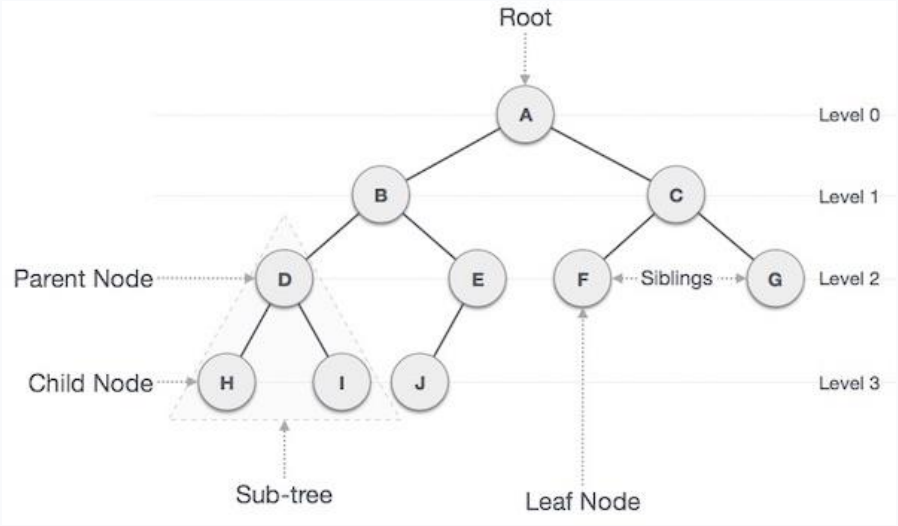
UNIT-5(TREES)

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



Why Tree Data Structure?

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

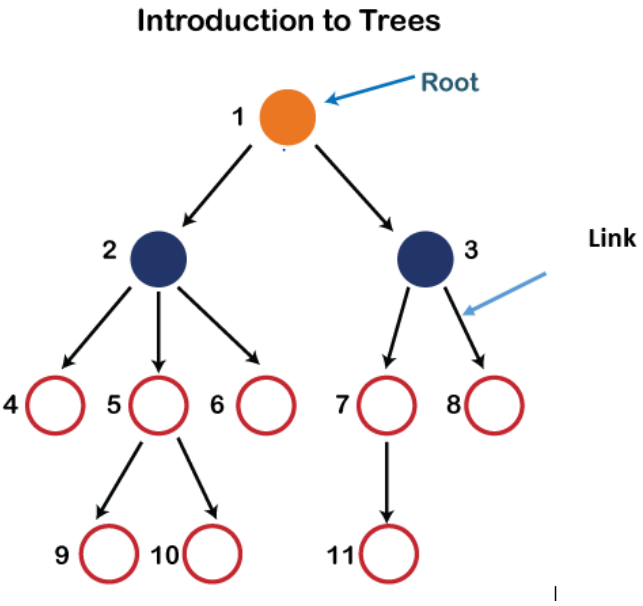
Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Properties of Tree: Every tree has a specific root node. A root node can cross each tree node. It is called root, as the tree was the only root. Every child has only one parent, but the parent can have many children.

Let's understand some key points of the Tree data structure.

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

Some basic terms used in Tree data structure.

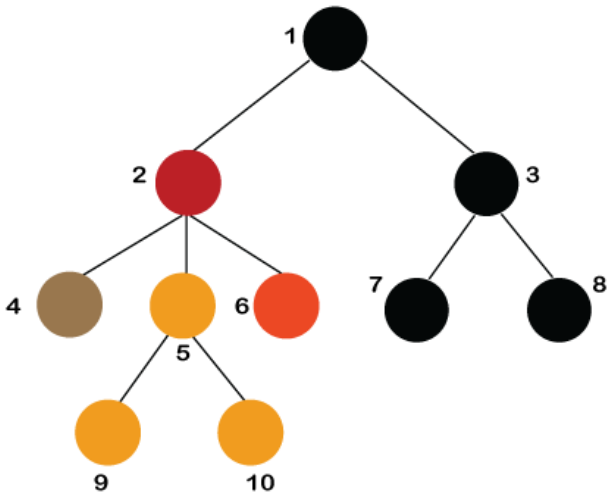


- **Node**:A node is an entity that contains a key or value and pointers to its child nodes.
- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node/External node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.

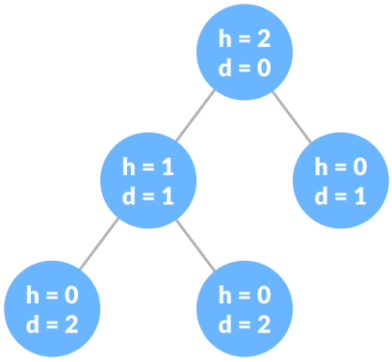
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.
- **Edge:** It is the link between any two nodes.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.



- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have at least one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

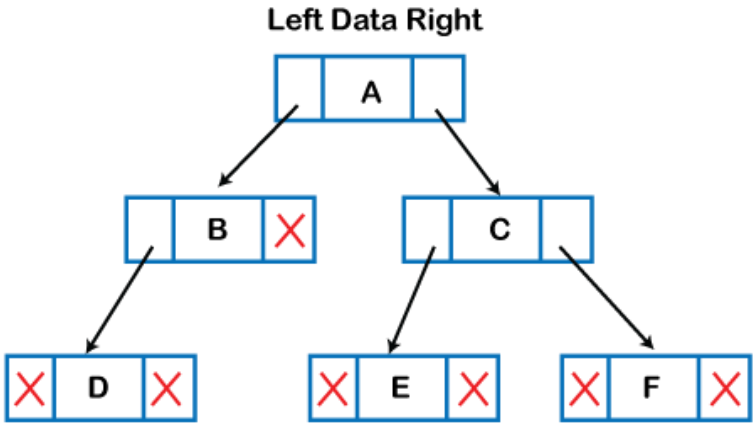


Degree of a Node

The degree of a node is the total number of branches of that node.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

```
1. struct node
2. {
3.   int data;
4. struct node *left;
5. struct node *right;
6. }
```

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Applications of trees

The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

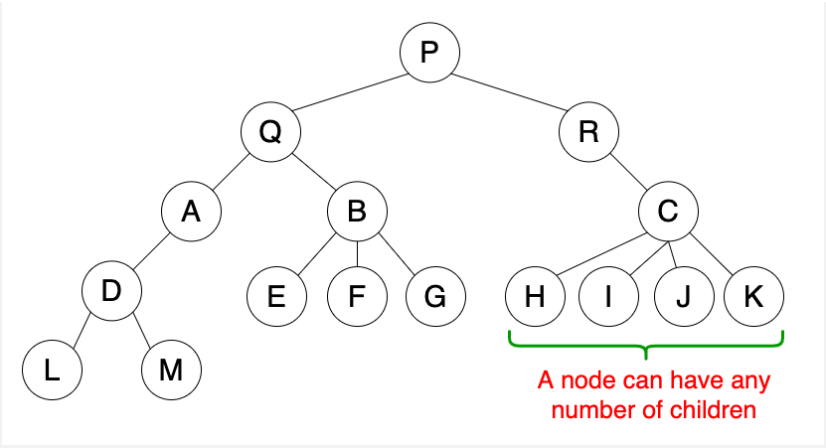
TYPES OF TREE: →As there are many types of tree in the forest in the same manner there are many types of tree in the data structure.

1. General Tree

If no constraint is placed on the tree's hierarchy, a tree is called a general tree. Every node may have infinite numbers of children in General Tree. The tree is the super-set of all other trees.

Properties

1. Follow properties of a tree.
2. A node can have any number of children.



Usage:→Used to store hierarchical data such as folder structures.

1. **Binary Tree:→**A **binary tree** is a tree data structure where the following properties can be found.

Properties

1. Follow properties of a tree.
2. A node can have at most two child nodes (children).
3. These two child nodes are known as the ***left child*** and ***right child***.

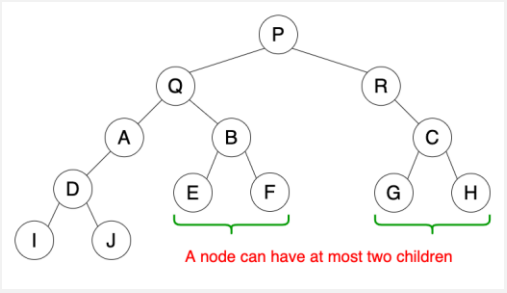


Fig 3. Binary tree

Usage

1. Used by compilers to build syntax trees.
2. Used to implement expression parsers and expression solvers.
3. Used to store router-tables in routers.

3. Binary Search Tree

A **binary search tree** is a more constricted extension of a binary tree.

Properties

1. Follow properties of a binary tree.
2. Has a unique property known as the **binary-search-tree property**. This property states that the value (or key) of the left child of a given node should be less than or equal to the parent value and the value of the right child should be greater than or equal to the parent value.

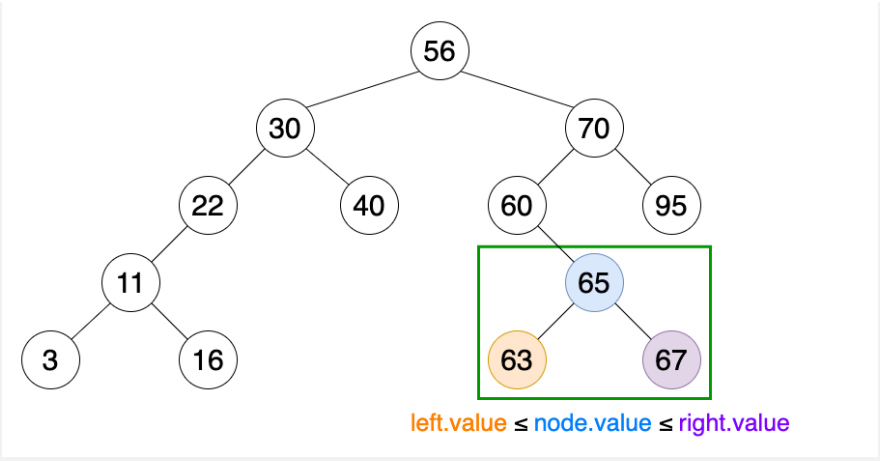


Fig 4. Binary search tree

Usage

1. Used to implement simple sorting algorithms.
2. Can be used as priority queues.
3. Used in many search applications where data are constantly entering and leaving.

4. AVL tree

An **AVL tree** is a self-balancing binary search tree. This is the first tree introduced which automatically balances its height.

Properties

1. Follow properties of binary search trees.
2. Self-balancing.
3. Each node stores a value called a **balance factor** which is the difference in height between its left subtree and right subtree.
4. All the nodes must have a balance factor of -1, 0 or 1.

After performing insertions or deletions, if there is at least one node that does not have a balance factor of -1, 0 or 1 then rotations should be performed to balance the tree (self-balancing).

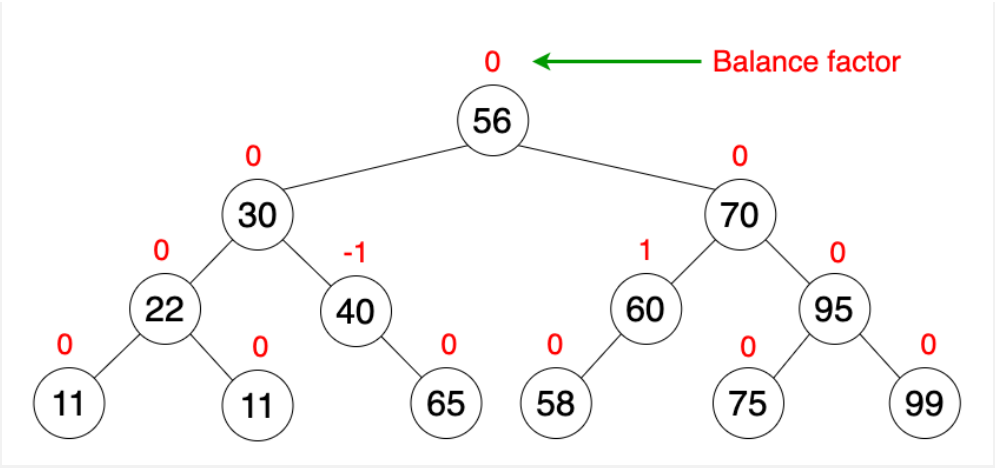


Fig 5. AVL tree

Usage

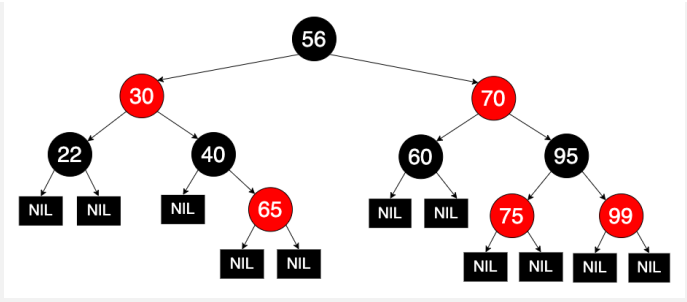
1. Used in situations where frequent insertions are involved.
2. Used in Memory management subsystem of the Linux kernel to search memory regions of processes during preemption.

5. Red-black tree

A red-black tree is a self-balancing binary search tree, where each node has a colour; red or black. The colours of the nodes are used to make sure that the tree remains approximately balanced during insertions and deletions.

Properties

1. Follow properties of binary search trees.
2. Self-balancing.
3. Each node is either red or black.
4. The root is black (sometimes omitted).
5. All leaves (denoted as NIL) are black.
6. If a node is red, then both its children are black.
7. Every path from a given node to any of its leaf nodes must go through the same number of black nodes.



Usage

1. As a base for data structures used in computational geometry.
2. Used in the *Completely Fair Scheduler* used in current Linux kernels.
3. Used in the *epoll* system call implementation of Linux kernel.

6. Splay tree

A **splay tree** is a self-balancing binary search tree.

Properties

1. Follow properties of binary search trees.
2. Self-balancing.
3. Recently accessed elements are quick to access again.

After performing a search, insertion or deletion, splay trees perform an action called **splaying** where the tree is rearranged (using rotations) so that the particular element is placed at the root of the tree.

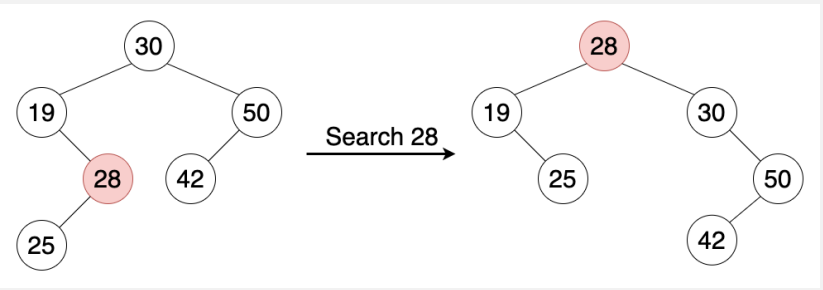


Fig 7. Splay tree search

Usage

1. Used to implement caches

2. Used in garbage collectors.
3. Used in data compression

7. Treap

A **treap** (the name derived from **tree + heap**) is a binary search tree.

Properties

1. Each node has two values; a **key** and a **priority**.
2. The keys follow the binary-search-tree property.
3. The priorities (which are random values) follow the heap property.

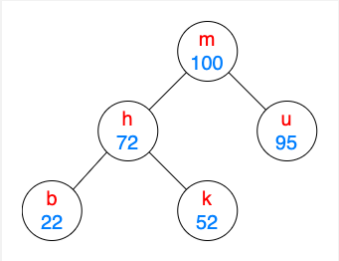


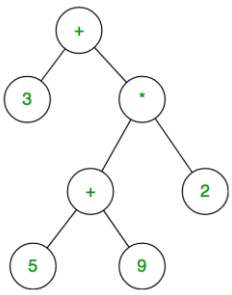
Fig 8. Treap (red coloured alphabetic keys follow BST property and blue coloured numeric values follow max heap order(Where the value of the root node is greater than or equal to either of its children)

Usage

1. Used to maintain authorization certificates in public-key cryptosystems.
2. Can be used to perform fast set operations.

Expression Tree

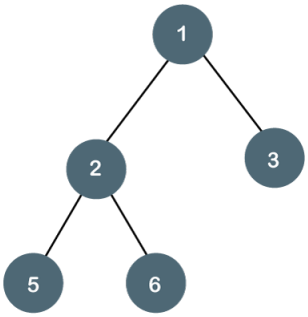
The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9)*2) would be:



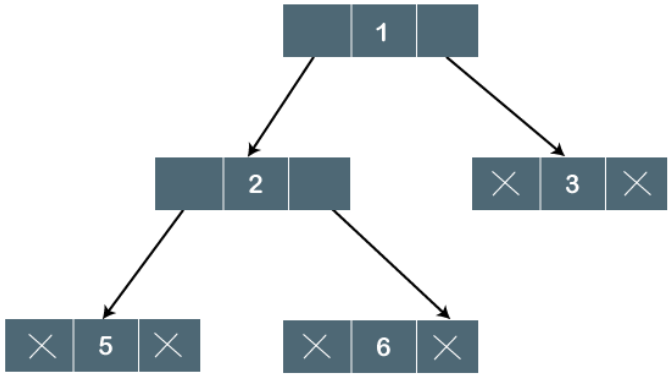
Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of 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+….2^h) = 2^{h+1} -1.

- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$\log_2(n+1) = \log2(2^{h+1})$

$\log_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

The maximum height can be computed as:

As we know that,

$n = h+1$

$h = n-1$

Types of Binary Tree

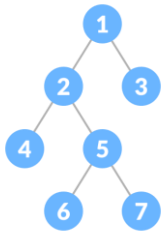
There are four types of Binary tree:

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

- **Degenerate Binary tree**
- **Balanced Binary tree**

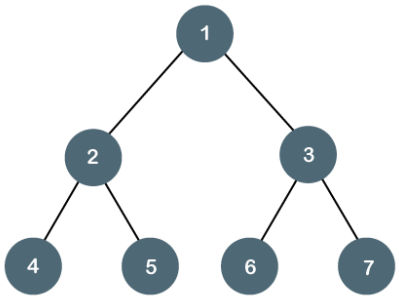
1. Full/ proper/ strict Binary tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



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.

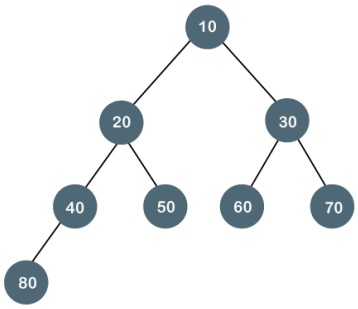


> ***Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.***

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.

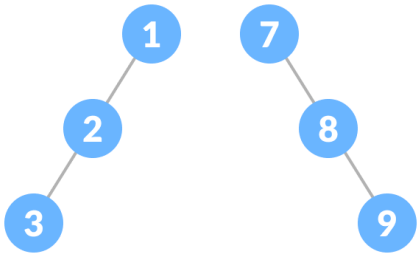


Degenerate Binary Tree

A degenerate or pathological tree is the tree having a single child either left or right.

Skewed Binary Tree

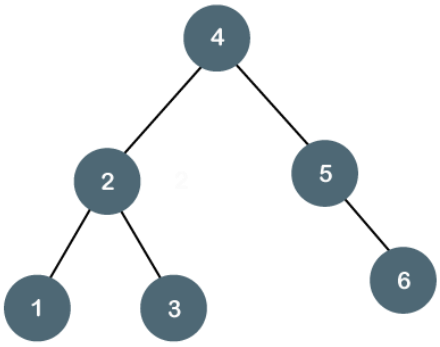
A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: **left-skewed binary tree** and **right-skewed binary tree**.



Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

Let's understand the balanced binary tree through examples.



Binary Tree Creation/Implementation

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

```
1. struct node
2. {
3.    int data,
4.    struct node *left, *right;
5. }  bnode
```

6. In the above structure, **data** is the value, **left pointer** contains the address of the left node, and **right pointer** contains the address of the right node.

Binary Tree program in C

```
#include<stdio.h>
1.       struct node
2.      {
3.           int data;
4.           struct node *left, *right;
5.       }
6.       void main()
7.     {
8.        struct node *root;
9.        root = create();
10.    }
11. struct node *create()
12. {
```

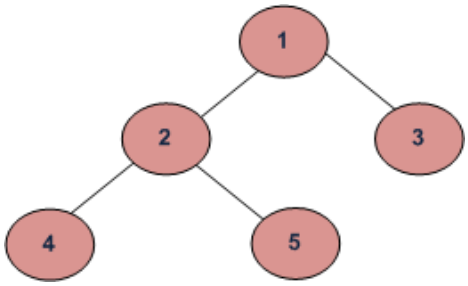
```
13.  struct node *temp;
14.  int data;
15.  temp = (struct node *)malloc(sizeof(struct node));
16.  printf("Press 0 to exit");
17.  printf("\nPress 1 for new node");
18.  printf("Enter your choice : ");
19.  scanf("%d", &choice);
20.  if(choice==0)
21.{
22.return 0;
23.}
24.else
25.{
26.  printf("Enter the data:");
27.  scanf("%d", &data);
28.  temp->data = data;
29.  printf("Enter the left child of %d", data);
30.  temp->left = create();
31.printf("Enter the right child of %d", data);
32.temp->right = create();
33.return temp;
34.}
35.}
```

The above code is calling the create() function recursively and creating new node on each recursive call. When all the nodes are created, then it forms a binary tree structure.

Tree Traversal:→The process of visiting the nodes is known as tree traversal. Tree Traversal Algorithms can be classified broadly in the following two categories by the order in which the nodes are visited:

1. **Depth-First Search (DFS) Algorithm:** It starts with the root node and first visits all nodes of one branch as deep as possible of the chosen Node and before backtracking, it visits all other branches in a similar fashion. There are three sub-types under this, which we will cover in this article.
2. **Breadth-First Search (BFS) Algorithm:** It also starts from the root node and visits all nodes of current depth before moving to the next depth in the tree.

Depth first search:→
1)Inorder traversal 2)Preorder traversal 3)Postorder traversal



(a) Inorder (Left, Root, Right) : 4 2 5 1 3

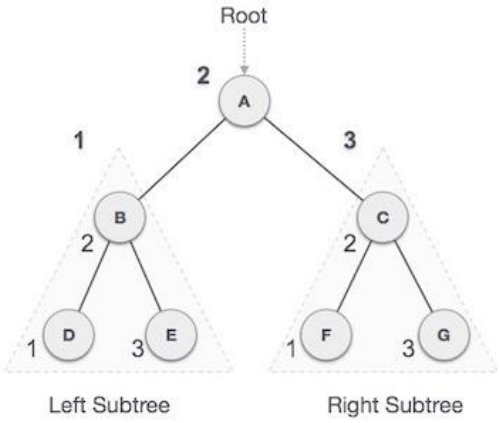
(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

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 −

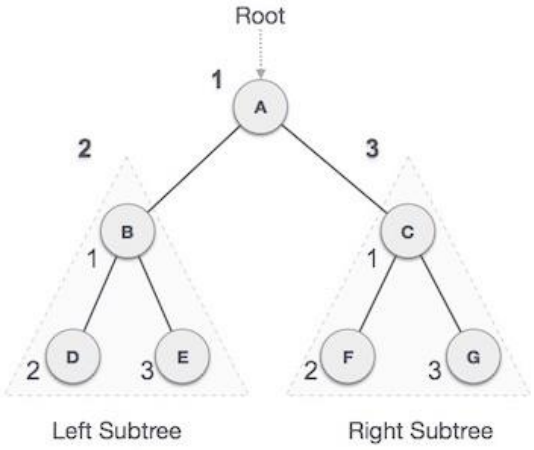
$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.(traverse the left subtree inorder)
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 −

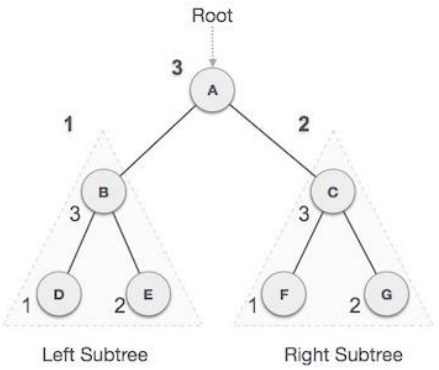
$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

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 −

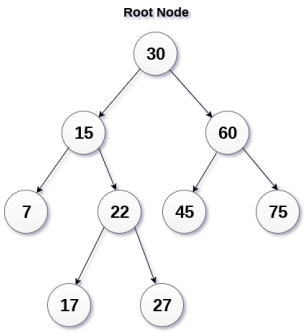
$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

Binary Search Tree

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



Binary Search Tree

Advantages of using binary search tree

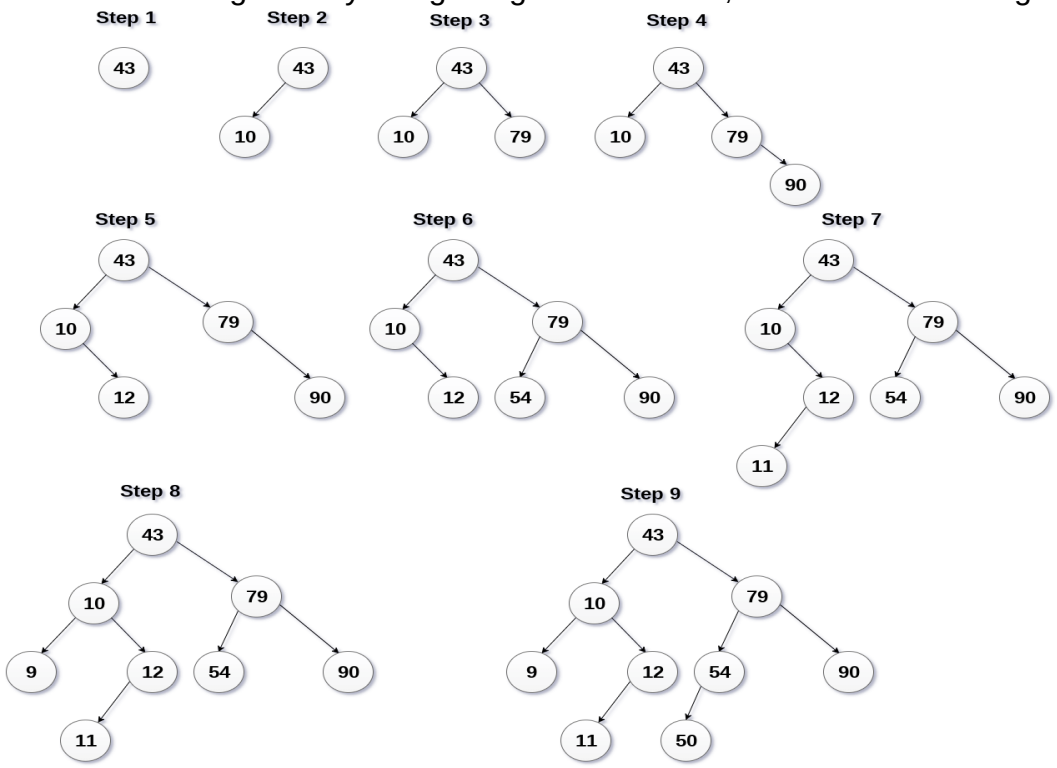
1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o($\log_2 n$) time. In worst case, the time it takes to search an element is 0(n).
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

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

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

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

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



Binary search Tree Creation

Operations on Binary Search Tree

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

SN	Operation	Description

1	Searching	Finding the location of some specific element in a binary search tree.
2	Insertion	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	Deletion	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

Searching:➔Searching means finding or locating some specific element or node within a data structure.

1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.

Algorithm:

Search (root, ITEM)

```
STEP-1) If root == NULL
        return NULL;
STEP2) If ITEM == root->data
            return root->data;
STEP3) If ITEM < root->data
            return search(root->left,ITEM)
STEP4) If ITEM> root->data
              return search(root->right,ITEM)
step5) END
```

Insertion

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must note violate the property of binary search tree at each value.

1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right sub-tree of the root.

Insert (TREE, ITEM)

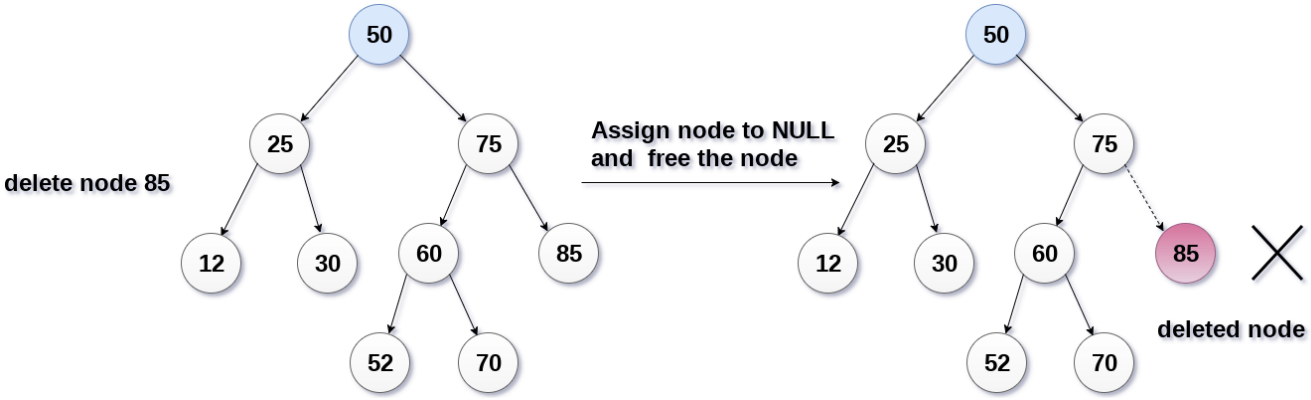
- **Step 1:** IF TREE = NULL
  Allocate memory for TREE
  SET TREE -> DATA = ITEM
  SET TREE -> LEFT = TREE -> RIGHT = NULL
  ELSE
  IF ITEM < TREE -> DATA
  Insert(TREE -> LEFT, ITEM)
  ELSE
  Insert(TREE -> RIGHT, ITEM)
  [END OF IF]
  [END OF IF]
- **Step 2:** END

Deletion

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

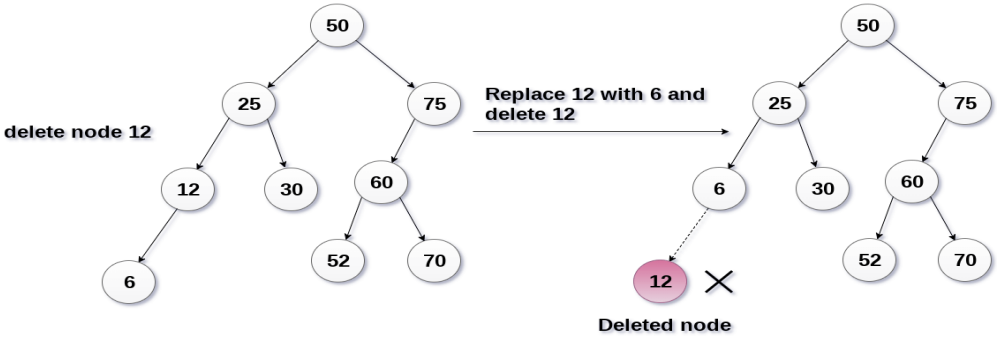
The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.



The node to be deleted has only one child.

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

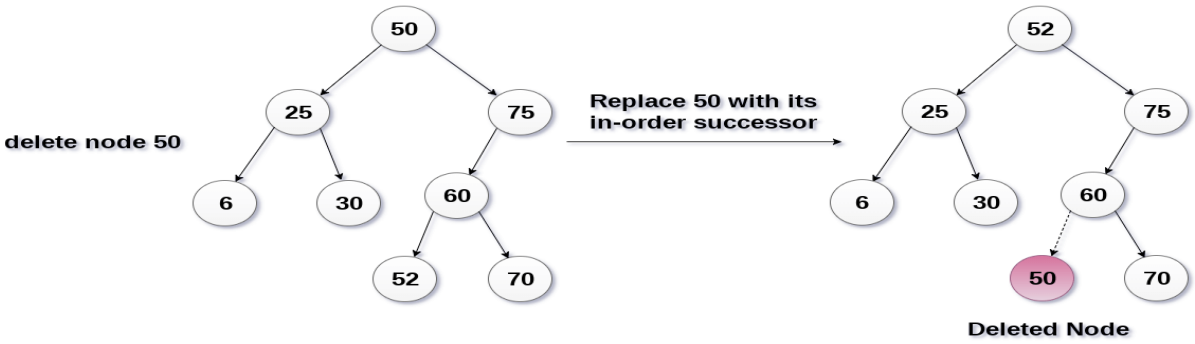


The node to be deleted has two children.

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

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree is:→6, 25, 30, 50, 52, 60, 70, 75.

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



Algorithm

Delete (TREE, ITEM)

- **Step 1:** IF TREE = NULL
  Write "item not found in the tree"
- ELSE IF (ITEM < TREE -> DATA)
  Delete(TREE->LEFT, ITEM)
  ELSE IF(ITEM > TREE -> DATA)
  Delete(TREE -> RIGHT, ITEM)
  ELSE IF (TREE -> LEFT AND TREE -> RIGHT)
  SET TEMP = findLargestNode(TREE -> LEFT)
  SET TREE -> DATA = TEMP -> DATA
  Delete(TREE -> LEFT, TEMP -> DATA)
  ELSE
  SET TEMP = TREE
  IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
  SET TREE = NULL
  ELSE IF TREE -> LEFT != NULL
  SET TREE = TREE -> LEFT
  ELSE
  SET TREE = TREE -> RIGHT
  [END OF IF]
  FREE TEMP
  [END OF IF]
- **Step 2:** END

Or we can use the following algorithm also

1. Input the number of nodes
2. Input the nodes of the tree
3. Consider the first element as the root and insert all the elements

4. Input the data of the node to be deleted
5. If the node is a leaf node ,delete the node directly
6. Else if the node has one child, copy the child to the node to be deleted and delete the child node
7. Else if the node has two children, find the in order successor or predecessor.
8. Copy the contents of the in order successor to the node to be deleted and delete the in order successor.

Heap Data Structure

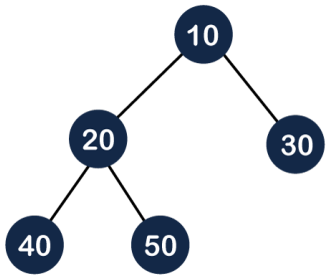
What is Heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

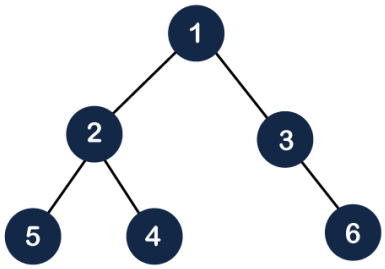
What is a complete binary tree?

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.

Let's understand through an example.



In the above figure, we can observe that all the internal nodes are completely filled except the leaf node; therefore, we can say that the above tree is a complete binary tree.



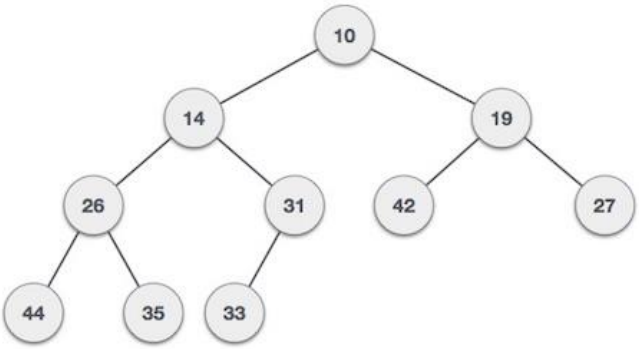
The above figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

Note: The heap tree is a special balanced binary tree data structure where the root node is compared with its children and arrange accordingly.

How can we arrange the nodes in the Tree?

There are two types of the heap:

- Min Heap
- Max heap
- **Min-Heap** − Where the value of the root node is less than or equal to either of its children.

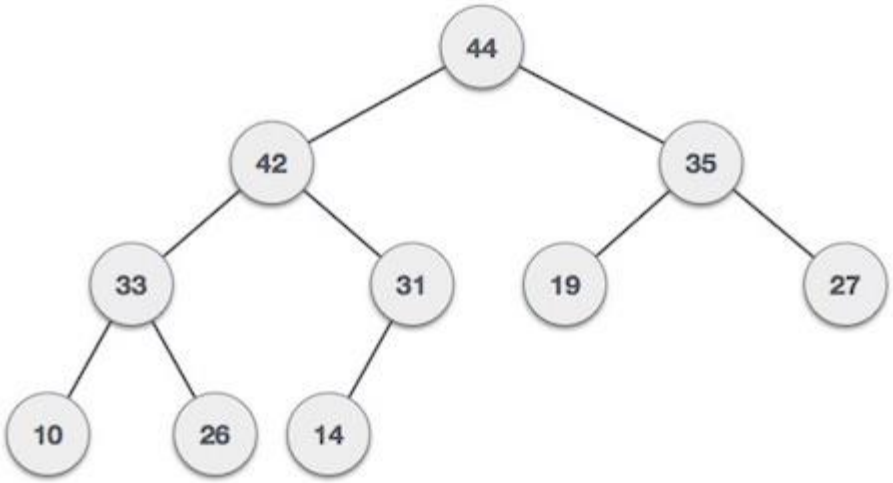


Algorithm:→

Step 1 − Create a new node at the end of heap.

- **Step 2** − Assign new value to the node.
- **Step 3** − Compare the value of this child node with its parent.
- **Step 4** − If value of parent is grater than child, then swap them.
- **Step 5** − Repeat step 3 & 4 until Heap property holds.

- **Max-Heap** − Where the value of the root node is greater than or equal to either of its children.

- 

Algorithm:→

Step 1 − Create a new node at the end of heap.

- **Step 2** − Assign new value to the node.
- **Step 3** − Compare the value of this child node with its parent.
- **Step 4** − If value of parent is lesser than child, then swap them.
- **Step 5** − Repeat step 3 & 4 until Heap property holds.
-