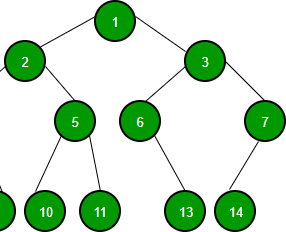
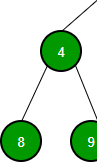
**UNIT III: (BINARY SEARCH TREES, AVL TREES, SPLAY TREES & RED BLACK TREES)**

**TREE:** A tree is recursively defined as a set of one or more nodes and branches(edges) where one node is designated as 'root' of the tree and all the remaining nodes can be partitioned into non-empty sets, each of which is a sub-tree with respect to root of the tree. It is a non-linear data structure compared to arrays, linked lists.



**Fig.** Example of a Tree

a

a

b

c

b

c

d

e

f

d

e

A Tree

Not a Tree

Figure 5.1.1 A Tree and a not a tree

**BASIC TERMINOLOGY:**

**Node:** A data unit to store information.

**Branch or edge:** The Link between two nodes.

**Root node:** It is the top most node in the tree.

**Leaf Node:** The node which does not have any children(external node).

**Degree of the Node:** It is the number of edges connected to that node.

**In-Degree:** Number of edges arriving to that node.

**Out-Degree:** Number of edges leaving from that node.

**Level:** It is the length of the path from the root to that node.

* Root node is said to be at level 0.

**Depth or Height of the Tree:** It is the maximum level of any node in the tree.

**Internal node:** A non-leaf node.

**Parent:** The unique predecessor of a node is termed as its parent.

**Child:** The successor of a node is termed as its child node.

**Siblings:** Nodes of the same parent node.

**Path:** Sequence of consecutive edges is called a path. (1-2-4-8)

**Degree of the Tree:** Maximum degree of a node in that tree.



1

Level 0

2

3

Level 1

4

5

6

7

Level 2

8

9

Level 3

Figure : Levels tree

**IMPORTANT POINTS TO BE NOTED:**

### Is Root node considered as internal node?

Any vertex (node) for which there exist one or more children are called as internal vertices, the root of a tree is an internal vertex unless it is the only vertex in the tree.

### Degree of a node = In-Degree + Out-Degree

* + Any node in a tree has a maximum of in-degree 'one'.
  + for any node in a binary tree, the maximum out-degree 'two'.
  + root node has in-degree 'zero' .
  + leaf node has out-degree 'zero'.

**Applications of trees:**

1. Used to represent hierarchies.
2. Used to represent simple as well as complex data.
3. Used for implementing other data structures like hash tables, sets, and maps.
4. Used for compiler construction.
5. Used in data base design.
6. Used in file system directories.
7. Used in symbol tables.

**BINARY TREE:**

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.



A

left child

B

C right child

left subtree

D

E

F

G

right subtree

H

I

A tree with no nodes is called as a **null** tree. A binary tree is shown in figure 5.2.1.

Figure 5.2.1. Binary Tree

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

* For every node in binary tree, In-degree is always 'one'.

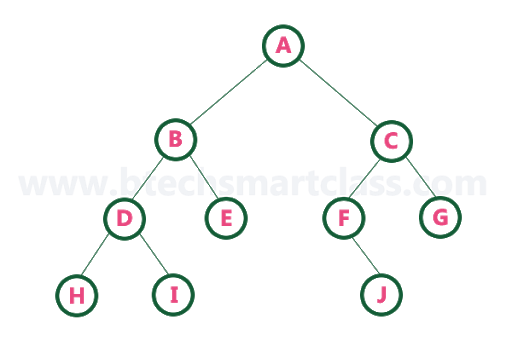
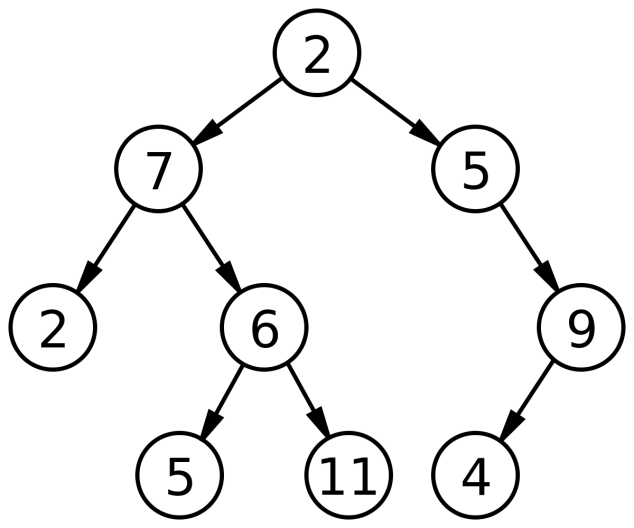
Out-degree is always less than or equal to 'two'. Maximum degree of any node is 'three'.

* Every binary tree is a tree, but every tree is not a binary tree.

### PROPERTIES OF A BINARY TREE:

1. The maximum number of nodes at level ‘L’ of a binary tree is 2L.
2. Maximum number of nodes in a binary tree of height ‘h’ is 2h+1 – 1.
3. In Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.

d



|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** |  |  |  | **J** |

**IMPLEMENTATION OF A BINARY TREE USING DOUBLY LINKED LIST:**

# Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation 2. Linked List Representation**

Consider the following binary tree...



# 1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

A node has an index *i*, its children are found at indices 2*i*+1 and 2*i*+2, while its parent (if any) is found at index *floor((i-1)/2)* (assuming the root of the tree stored in the array at an index zero).  
Consider the above example of a binary tree and it is represented as follows...

Root node is at a[0] and its children are at a[1] and a[2] i.e A is at a[0] and

B& C are at a[1] & a[2].

Children of B are at a[3] and a[4]

Children of c are at a[5] and a[6]



To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **2n + 1**.



0 1 2 3 4 5 6

## 2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.  
In this linked list representation, a node has the following structure...



The above example of the binary tree represented using Linked list representation is shown as follows...



|  |  |  |
| --- | --- | --- |
|  | A |  |

|  |  |  |
| --- | --- | --- |
|  | B |  |

|  |  |  |
| --- | --- | --- |
|  | C |  |

|  |  |  |
| --- | --- | --- |
|  | D |  |

|  |  |  |
| --- | --- | --- |
| **X** | E | **X** |

|  |  |  |
| --- | --- | --- |
| **X** | F | **X** |

|  |  |  |
| --- | --- | --- |
| **X** | G | **X** |

|  |  |  |
| --- | --- | --- |
| **X** | H | **X** |

|  |  |  |
| --- | --- | --- |
| **X** | I | **X** |

Figure: Linked representation for the binary tree

A

root

B

C

D E F G

H I

Typically, creation of a node is done as, A Node contains:

1. Data
2. Pointer to left child
3. Pointer to right child

struct node { int data;

struct node \*leftChild; struct node \*rightChild;

};

Then respective links are given to the nodes, creating branches between the nodes by using two self-referencing structure pointers.

### TYPES OF BINARY TREES:

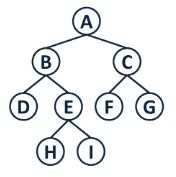
There are different types of binary trees,

1. Full Binary Tree
2. Complete Binary Tree

### FULL OR STRICTLY BINARY TREE

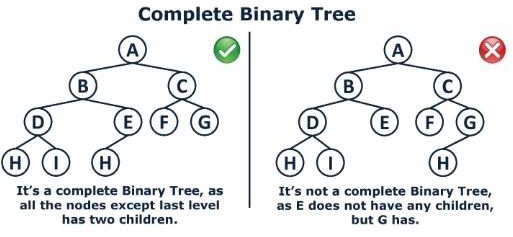
A Binary Tree is said to be a Full or Strictly Binary Tree, If all the nodes other than leaf nodes has 0 or 2 children, then that is Full Binary Tree.

* + All the nodes in a Full or Strictly Binary Tree are of out-degree zero or two, never degree one.



### COMPLETE BINARY TREE

A Binary Tree is said to be Complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all the nodes as left as possible.



**3. Perfect Binary Tree** A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

**Full Binary Tree** A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

18

/ \

15 30

/ \ / \

40 50 100 40

18

/ \

15 20

/ \

40 50

/ \

30 50

18

/ \

40 30

/ \

100 40

***In a Full Binary Tree, number of leaf nodes is the number of internal nodes plus 1***  
       L = I + 1  
Where L = Number of leaf nodes, I = Number of internal nodes  
See [Handshaking Lemma and Tree](https://www.geeksforgeeks.org/handshaking-lemma-and-interesting-tree-properties/) for proof.

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

The following are examples of Complete Binary Trees

18

/ \

15 30

/ \ / \

40 50 100 40

18

/ \

15 30

/ \ / \

40 50 100 40

/ \ /

8 7 9

Practical example of Complete Binary Tree is [Binary Heap](http://quiz.geeksforgeeks.org/binary-heap/).

**Perfect Binary Tree** A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.  
The following are the examples of Perfect Binary Trees.

18

/ \

15 30

/ \ / \

40 50 100 40

18

/ \

15 30

A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has 2h – 1 node.

An example of a Perfect binary tree is ancestors in the family. Keep a person at root, parents as children, parents of parents as their children.

### Binary Tree Traversal Techniques:

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are four common ways to traverse a binary tree:

* + 1. *Preorder*
    2. *Inorder*
    3. *Postorder*
    4. *Level order*

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time at which a root node is visited.

### Recursive Traversal Algorithms:

**Inorder Traversal:**

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

* + - 1. Visit the left subtree, using inorder.
      2. Visit the root.
      3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows: void inorder(node \*root)

{

if(root != NULL)

{

inorder(root->lchild);

print root -> data; inorder(root->rchild);

}

}

### Preorder Traversal:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows: void preorder(node \*root)

{

if( root != NULL )

{

print root -> data; preorder (root -> lchild); preorder (root -> rchild);

}

}

### Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for postorder traversal is as follows: void postorder(node \*root)

{

if( root != NULL )

{

postorder (root -> lchild); postorder (root -> rchild); print (root -> data);

}

}

Traverse the following binary tree in pre, post, inorder and level order.



A

B

C

D

E

F

G

H I

* P reo rde r t ra v e rs a l y ie lds: A, B, D, C , E, G , F , H, I
* Po sto rder travers a l yields: D, B, G , E, H, I, F , C , A
* Ino rder travers a l yields: D, B, A, E, G , C , H, F , I

Bin a ry T re e Pre, P o st , In ord er T rav ers in g

### Example 2:

Traverse the following binary tree in pre, post, inorder and level order.

P

F

S

B H R Y

G

T

Z

W

* P reo rde r t ra v e rs a l y ie lds:

P , F , B, H, G , S, R, Y, T, W , Z

* Po sto rder travers a l yields:

B, G , H, F , R, W , T, Z, Y, S, P

* Ino rder travers a l yields:

B, F , G , H, P , R, S, T, W , Y, Z

Bin a ry T re e Pre, P o st , In ord er T rav ers in g

### Example 3:

Traverse the following binary tree in pre, post, inorder and level order.

2

7

5

2

6

9

5 11 4

* P reo rde r t ra v e rs a l y ie lds: 2 , 7, 2, 6, 5, 11 , 5, 9, 4
* Po sto rder t rav ars a l y ields: 2 , 5, 11 , 6, 7, 4, 9, 5, 2
* Ino rder t rav ars a l y ields: 2 , 7, 5, 6, 11 , 2, 5, 4, 9

Bin a ry T re e Pre, P o st , In ord er T rav ers in g

### Example 4:

Traverse the following binary tree in pre, post, inorder and level order.

A

B

C

D

E

G

H

K

L

M

* P reo rde r t ra v e rs a l y ie lds: A, B, D, G , K, H, L, M , C , E
* Po sto rder t rav ars a l y ields: K, G , L, M , H, D, B, E, C , A
* Ino rder t rav ars a l y ields:

K, G , D, L, H, M , B, A, E, C

Bin a ry T re e Pre, P o st , In ord er T rav ers in g

### Building Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

* Inorder and preorder
* Inorder and postorder

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.

Same technique can be applied repeatedly to form sub-trees.

It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

### Example 1:

Construct a binary tree from a given preorder and inorder sequence: Preorder: A B D G C E H I F

Inorder: D G B A H E I C F

### Solution:

*From Preorder sequence* ***A*** *B D G C E H I F*, *the root is: A*

From Inorder sequence *D G B* ***A*** *H E I C F*, we get the left and right sub trees:

*Left sub tree is: D G B Right sub tree is: H E I C F*

The Binary tree upto this point looks like:



A

D G B

H E I C F

To find the root, left and right sub trees for D G B:

*From the preorder sequence* ***B*** *D G, the root of tree is: B*

From the inorder sequence D G **B,** we can find that D and G are to the left of B. The Binary tree upto this point looks like:



A

B

H E I C F

D G

To find the root, left and right sub trees for D G:

*From the preorder sequence* **D** *G, the root of the tree is: D*

From the inorder sequence ***D*** *G*, we can find that there is no left node to D and G is at the right of D.

The Binary tree upto this point looks like:



A

B

H E I C F

D

G

To find the root, left and right sub trees for H E I C F:

*From the preorder sequence* ***C*** *E H I F, the root of the left sub tree is: C*

From the inorder sequence *H E I* ***C*** *F*, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:



A

B

C

D

H E I

F

G

To find the root, left and right sub trees for H E I:

*From the preorder sequence* ***E*** *H I, the root of the tree is: E*

From the inorder sequence *H* ***E*** *I*, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:



A

B

C

D

E

F

G

H

I

### Example 2:

Construct a binary tree from a given postorder and inorder sequence: Inorder: D G B A H E I C F

Postorder: G D B H I E F C A

### Solution:

*From Postorder sequence* G D B H I E F C **A**, *the root is: A*

From Inorder sequence *D G B* **A** *H E I C F*, we get the left and right sub trees:

*Left sub tree is: D G B Right sub tree is: H E I C F*

The Binary tree upto this point looks like:



A

H E I C F

D G B

To find the root, left and right sub trees for D G B:

*From the postorder sequence G D B, the root of tree is: B*

From the inorder sequence *D G* **B,** we can find that D G are to the left of B and there is no right subtree for B.

The Binary tree upto this point looks like:



A

B

H E I C F

D G

To find the root, left and right sub trees for D G:

*From the postorder sequence G* ***D****, the root of the tree is: D*

From the inorder sequence **D** *G*, we can find that is no left subtree for D and G is to the right of D.

The Binary tree upto this point looks like:



A

B

H E I C F

D

G

To find the root, left and right sub trees for H E I C F:

*From the postorder sequence H I E F* ***C****, the root of the left sub tree is: C*

From the inorder sequence H E I ***C*** F, we can find that H E I are to the left of C and F is the right subtree for C.

The Binary tree upto this point looks like:



A

B

C

D

H E I

F

G

To find the root, left and right sub trees for H E I:

*From the postorder sequence H I* ***E****, the root of the tree is: E*

From the inorder sequence *H* **E** *I*, we can find that H is left subtree for E and I is to the right of E.

The Binary tree upto this point looks like:



A

B

C

D

E

F

G

H

I

### Example 3:

Construct a binary tree from a given preorder and inorder sequence: Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9

Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

### Solution:

*From Preorder sequence* **n6** n2 n1 n4 n3 n5 n9 n7 n8, *the root is: n6*

From Inorder sequence *n1 n2 n3 n4 n5* **n6** *n7 n8 n9*, we get the left and right sub trees:

*Left sub tree is:* n1 n2 n3 n4 n5

*Right sub tree is:* n7 n8 n9

The Binary tree upto this point looks like:



n6

n7 n8 n9

n1 n2 n3 n4 n5

To find the root, left and right sub trees for n1 n2 n3 n4 n5:

*From the preorder sequence* ***n2*** *n1 n4 n3 n5, the root of tree is: n2*

From the inorder sequence *n1* **n2** *n3 n4 n5***,** we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2. The Binary tree upto this point looks like:



n6

n2

n7 n8 n9

n1

n3 n4 n5

To find the root, left and right sub trees for n3 n4 n5:

*From the preorder sequence* ***n4*** *n3 n5, the root of the tree is: n4*

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is at the right of n4.

The Binary tree upto this point looks like:



n6

n2

n7 n8 n9

n1

n4

n3

n5

To find the root, left and right sub trees for n7 n8 n9:

*From the preorder sequence* ***n9*** *n7 n8, the root of the left sub tree is: n9*

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are at the left of n9 and no right subtree of n9.

The Binary tree upto this point looks like:



n6

n2

n9

n1

n4 n7 n8

n3

n5

To find the root, left and right sub trees for n7 n8:

*From the preorder sequence* ***n7*** *n8, the root of the tree is: n7*

From the inorder sequence **n7** *n8*, we can find that is no left subtree for n7 and n8 is at the right of n7.

The Binary tree upto this point looks like:



n6

n2

n9

n1

n4 n7

n3

n5 n8

### Example 4:

Construct a binary tree from a given postorder and inorder sequence: Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9

Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

### Solution:

*From Postorder sequence n1 n3 n5 n4 n2 n8 n7 n9* **n6**, *the root is: n6*

From Inorder sequence *n1 n2 n3 n4 n5* **n6** *n7 n8 n9*, we get the left and right sub trees:

*Left sub tree is: n1 n2 n3 n4 n5 Right sub tree is: n7 n8 n9*

The Binary tree upto this point looks like:



n6

n7 n8 n9

n1 n2 n3 n4 n5

To find the root, left and right sub trees for *n1 n2 n3 n4 n5*:

*From the postorder sequence n1 n3 n5 n4* **n2***, the root of tree is: n2*

From the inorder sequence *n1* **n2** *n3 n4 n5***,** we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.

The Binary tree upto this point looks like:



n6

n2

n7 n8 n9

n1

n3 n4 n5

To find the root, left and right sub trees for n3 n4 n5:

*From the postorder sequence n3 n5* **n4***, the root of the tree is: n4*

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is to the right of n4. The Binary tree upto this point looks like:



n6

n2

n7 n8 n9

n1

n4

n3

n5

To find the root, left and right sub trees for n7 n8 and n9:

*From the postorder sequence n8 n7* **n9***, the root of the left sub tree is: n9*

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are to the left of n9 and no right subtree for n9.

The Binary tree upto this point looks like:



n6

n2

n9

n1

n4 n7 n8

n3

n5

To find the root, left and right sub trees for n7 and n8:

*From the postorder sequence n8* **n7***, the root of the tree is: n7*

From the inorder sequence **n7** *n8*, we can find that there is no left subtree for n7 and n8 is to the right of n7. The Binary tree upto this point looks like:



n6

n2

n9

n1

n4

n7

n3

n5

n8

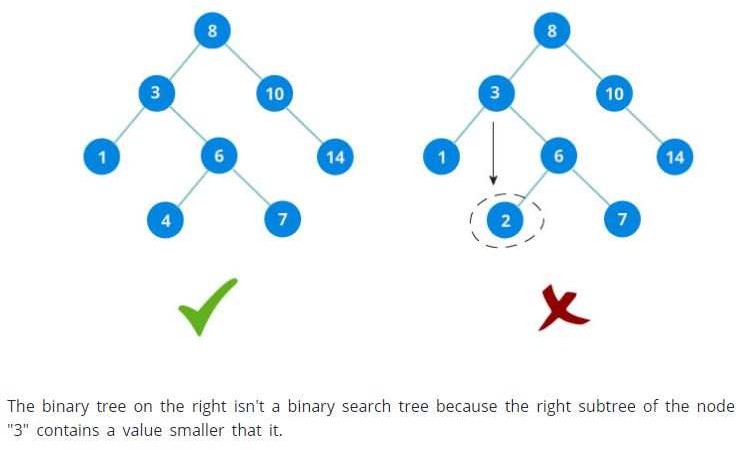
## BINARY SEARCH TREES (BST):

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. **IT HAS ALL THE PROPERTIES OF A BINARY TREE.**

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

* + The left subtree of a node contains only nodes with keys lesser than the node’s key.
  + The right subtree of a node contains only nodes with keys greater than the node’s key.
  + The left and right subtree each must also be a binary search tree.
  + This rule will be recursively applied to all the left and right sub-trees of the root.

Rule of Binary Search Tree: Left Child ≤ Root ≤ Right Child



**NOTE:** Every Binary Search Tree is a Binary Tree, But not every Binary Tree is a Binary Search Tree.

#### Tree > Binary Tree > Binary Search Tree

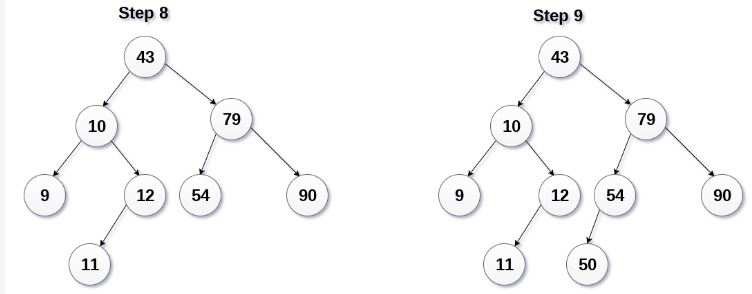
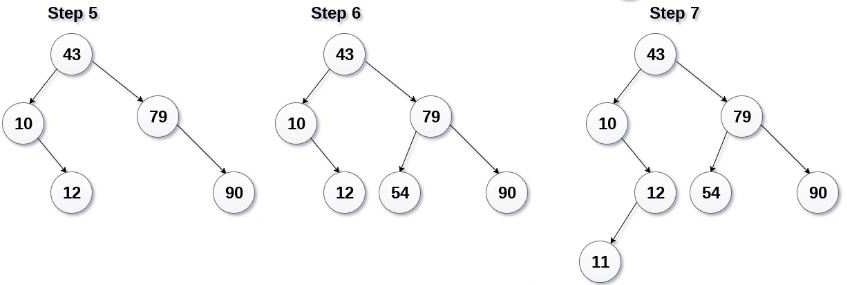
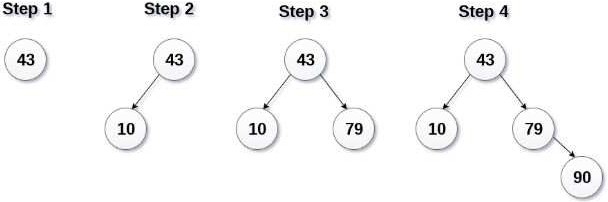
**Advantages of using Binary Search Tree:**

1. Searching become very efficient in a binary search tree since, we skip half of the elements in each step of search process.
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(log2n) time. In worst case, the time it takes to search an element is O(n).
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.
4. It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

**CREATION OF BINARY SEARCH TREE:**

**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.
4. Continue the process recursively for all the entries.



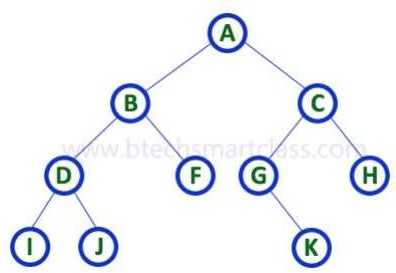
**NOTE:** If you observe carefully, we can see that the left-most element is the smallest, and right-most element is the largest in a binary search tree.

**BINARY SEARCH TREE TRAVERSALS:**

**Traversal:** Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

### There are mainly three types of Traversal techniques for BST:

1. Pre-Order Traversal (root-left-right)
2. In-Order Traversal (left-root-right)
3. Post-Order Traversal (left-right-root) Consider the following binary tree...



### In - Order Traversal ( Left Child - Root - Right Child )

In In-Order traversal, the root node is visited between the left child and right child.

In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

#### I - D - J - B - F - A - G - K - C – H

1. **Pre - Order Traversal ( Root - Left Child - Right Child )**

In Pre-Order traversal, the root node is visited before the left child and right child nodes.

In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the leftmost child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H.

After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this, we have completed node C's root and left parts. Next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

#### A - B - D - I - J - F - C - G - K - H

1. **Post - Order Traversal ( Left Child - Right Child - Root )**

In Post-Order traversal, the root node is visited after left child and right child.

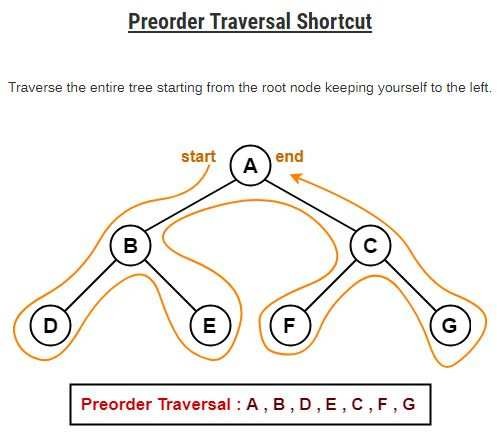
In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

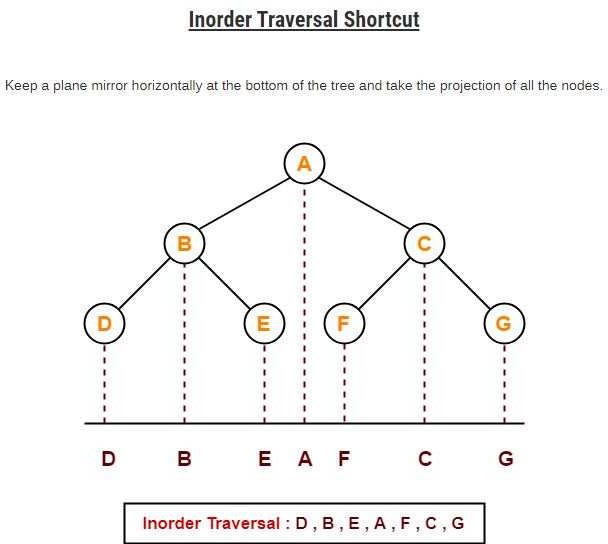
#### I - J - D - F - B - K - G - H - C - A

**IMPORTANT POINTS TO BE NOTED:**



* **Application of Pre-Order Traversal:**

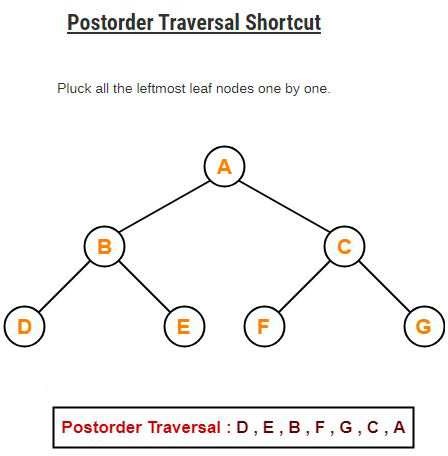
1. used to create a copy of the tree.
2. used to get prefix expression of an expression tree.



### Application of In-Order Traversal:

* 1. used to get infix expression of an expression tree.

**NOTE:** In-Order traversal of BST always produces sorted output.



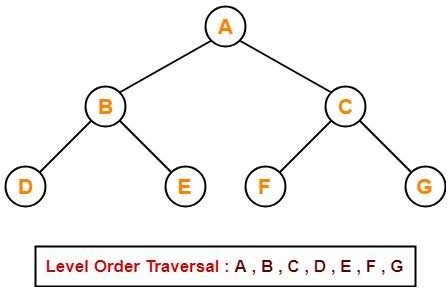
### Application of Post-Order Traversal:

1. used to get postfix expression of an expression tree.
2. used to delete the tree. This is because it deletes the children first and then it deletes the parent.

### BREADTH FIRST SEARCH TRAVERSAL:

Breadth First Traversal of a tree prints all the nodes of a tree level by level. Breadth First Traversal is also called as Level Order Traversal.

#### EXAMPLE:



**APPLICATION OF BFS TRAVERSAL:**

Level order traversal is used to print the data in the same order as stored in the array representation of a complete binary tree.

**ALGORITHM FOR BINARY TREE TRAVERSAL:**

**Pre-Order**

* Visit and print the node.
* Traverse the left sub-tree, (recursively call In-Order(root->left))
* Traverse the right sub-tree, (recursively call In-Order(root->right))

1. void preorder(struct node \*root) 2. {

3. if(root!=NULL) 4. {

1. printf("\n%d",root->data);
2. preorder(root->left);
3. preorder(root->right); 8. }

9. }

### In-Order

* Traverse the left sub-tree, (recursively call In-Order(root->left))
* Visit and print the node.
* Traverse the right sub-tree, (recursively call In-Order(root->right))

1. void inorder(struct node \*root) 2. {

3. if(root!=NULL) 4. {

1. inorder(root->left);
2. printf("\n%d",root->data);
3. inorder(root->right); 8. }

9. }

### Post-Order

* Traverse the left sub-tree, (recursively call In-Order(root->left))
* Traverse the right sub-tree, (recursively call In-Order(root->right))
* Visit and print the node.

1. void postorder(struct node \*root) 2. {

3. if(root!=NULL) 4. {

1. postorder(root->left);
2. postorder(root->right);
3. printf("\n%d",root->data); 8. }

9. }

### OPERATIONS ON A BINARY SEARCH TREE:

The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

### Search Operation in BST

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

* + Step 1 - Read the search element from the user.
  + Step 2 - Compare the search element with the value of root node in the tree.
  + Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
  + Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
  + Step 5 - If search element is smaller, then continue the search process in left subtree.
  + Step 6- If search element is larger, then continue the search process in right subtree.
  + Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node
  + Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
  + Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

### Insertion Operation in BST

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

* + Step 1 - Create a newNode with given value and set its left and right to NULL.
  + Step 2 - Check whether tree is Empty.
  + Step 3 - If the tree is Empty, then set root to newNode.
  + Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).
  + Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.
  + Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).
  + Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

### Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree includes following three cases...

* + Case 1: Deleting a Leaf node (A node with no children)
  + Case 2: Deleting a node with one child
  + Case 3: Deleting a node with two children

### Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

* + Step 1 - Find the node to be deleted using search operation
  + Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

### Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

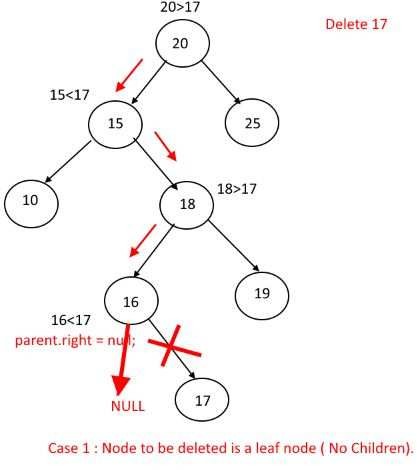
* + Step 1 - Find the node to be deleted using search operation
  + Step 2 - If it has only one child then create a link between its parent node and child node.
  + Step 3 - Delete the node using free function and terminate the function.

### Case 3: Deleting a node with two children

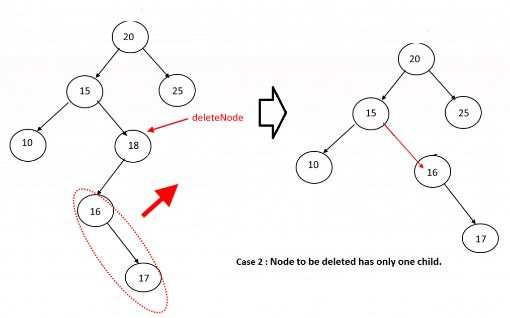
We use the following steps to delete a node with two children from BST...

* + Step 1 - Find the node to be deleted using search operation
  + Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
  + Step 3 - Swap both deleting node and node which is found in the above step.
  + Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2
  + Step 5 - If it comes to case 1, then delete using case 1 logic.
  + Step 6- If it comes to case 2, then delete using case 2 logic.
  + Step 7 - Repeat the same process until the node is deleted from the tree.

### ILLUSTRATION OF DELETE OPERATION IN BST:



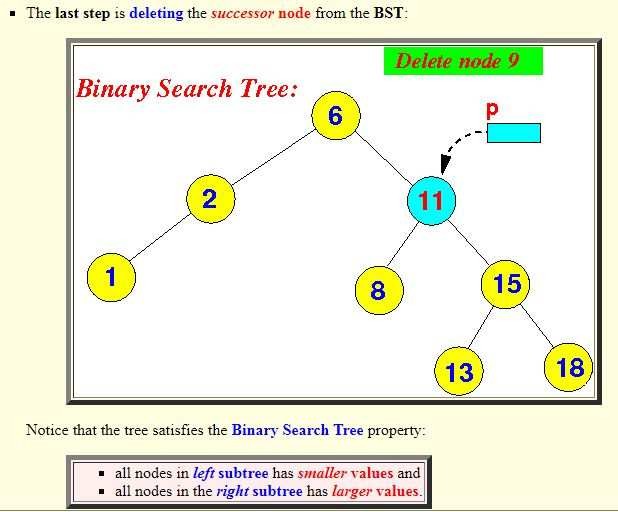
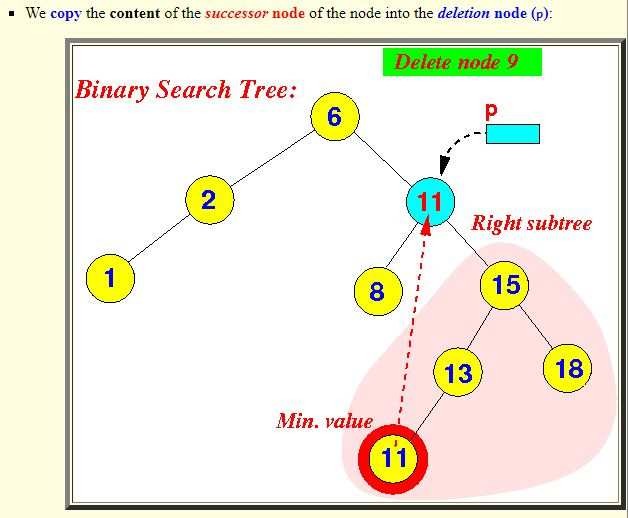
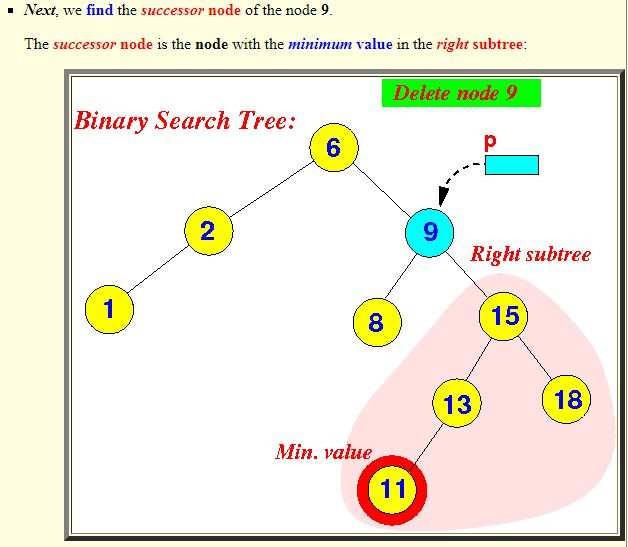
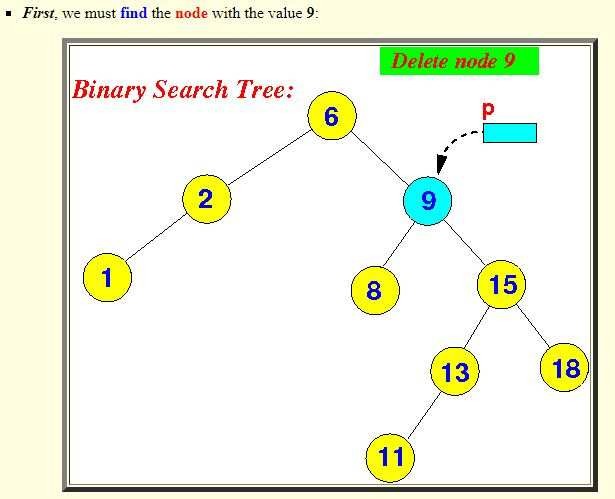
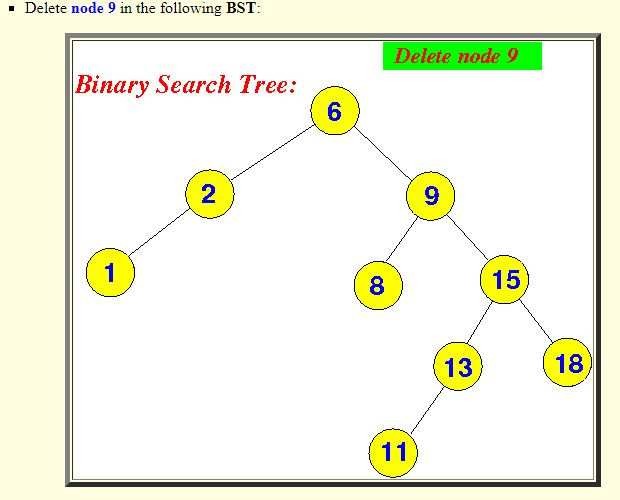
**1.**



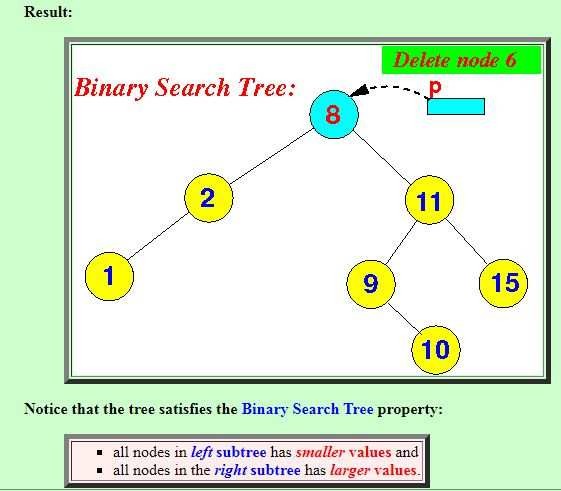
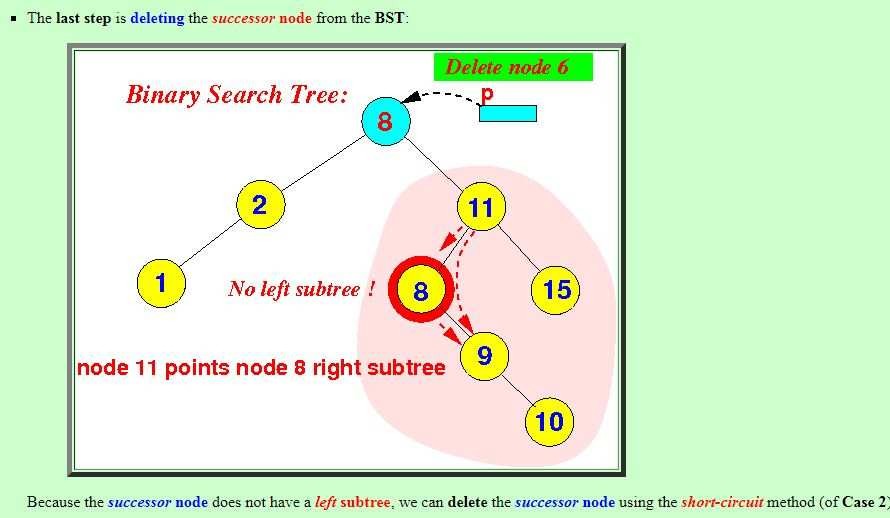
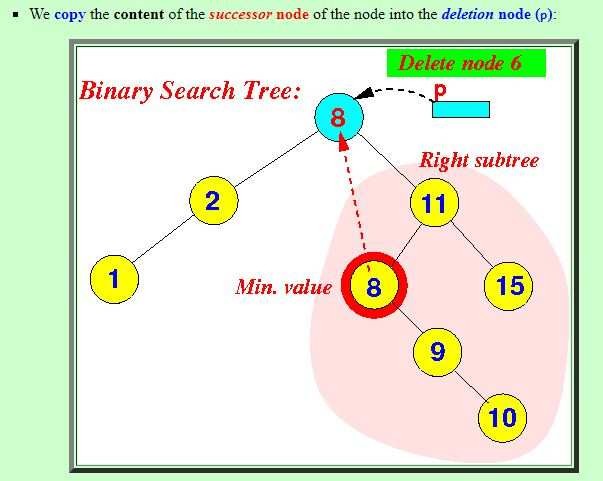
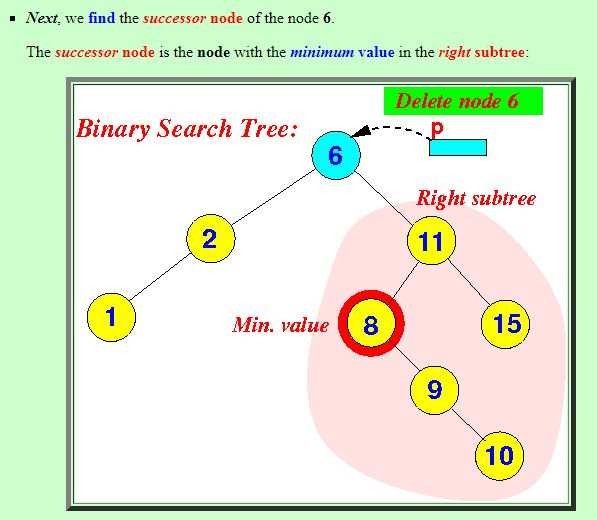
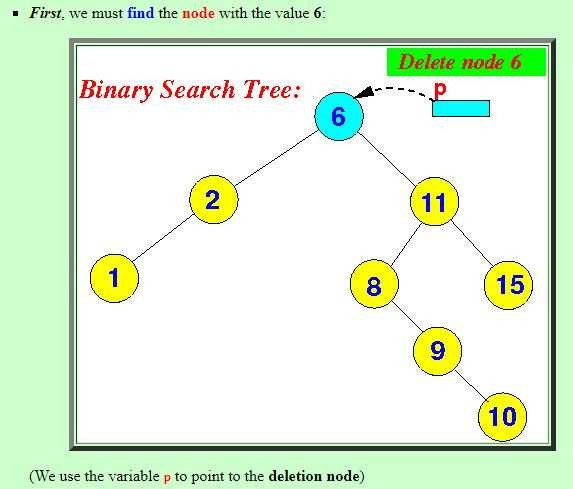
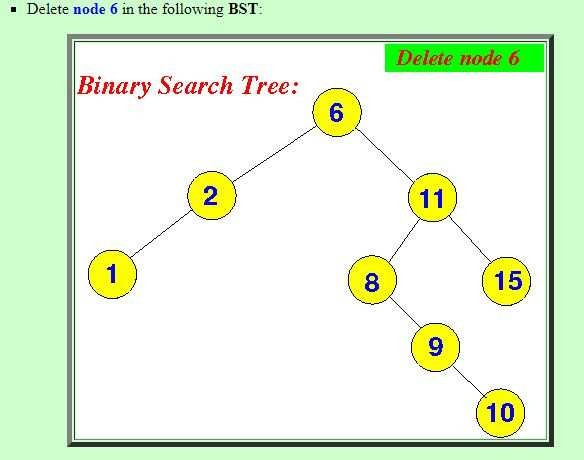
**2.**

**DELETION OF NODE WITH TWO CHILDREN(CASE 3):**

**EXAMPLE 1:**



**EXAMPLE 2:**



**AVL TREE:**

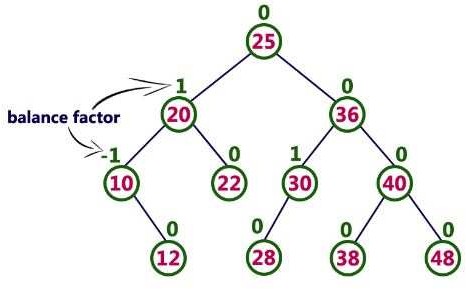
AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as balance factor. The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.

#### RULE OF AVL TREE:

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

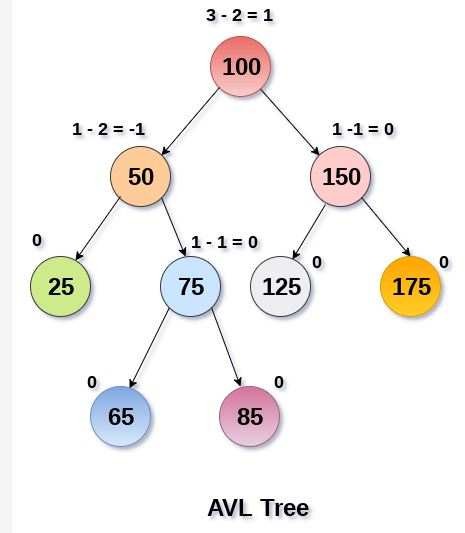
Where, Balance Factor = (Height of left subtree - Height of right subtree)

## EXAMPLE OF AVL TREE:



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

**NOTE:** Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

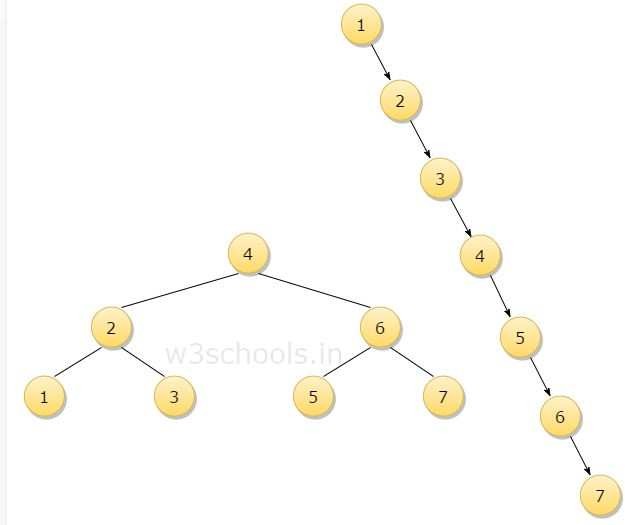


## WHY AVL TREES?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree

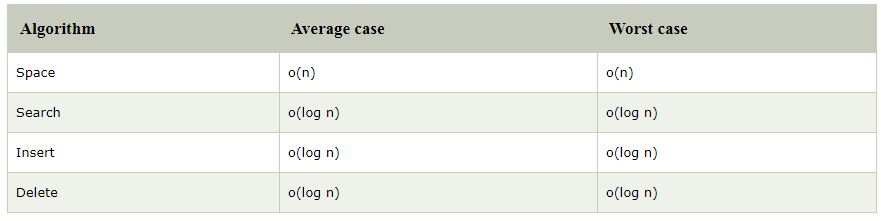
Since AVL trees are height balance trees, operations like insertion and deletion have low time complexity. Let us consider an example:

If you have the following tree having keys 1, 2, 3, 4, 5, 6, 7 and then the binary tree will be like :



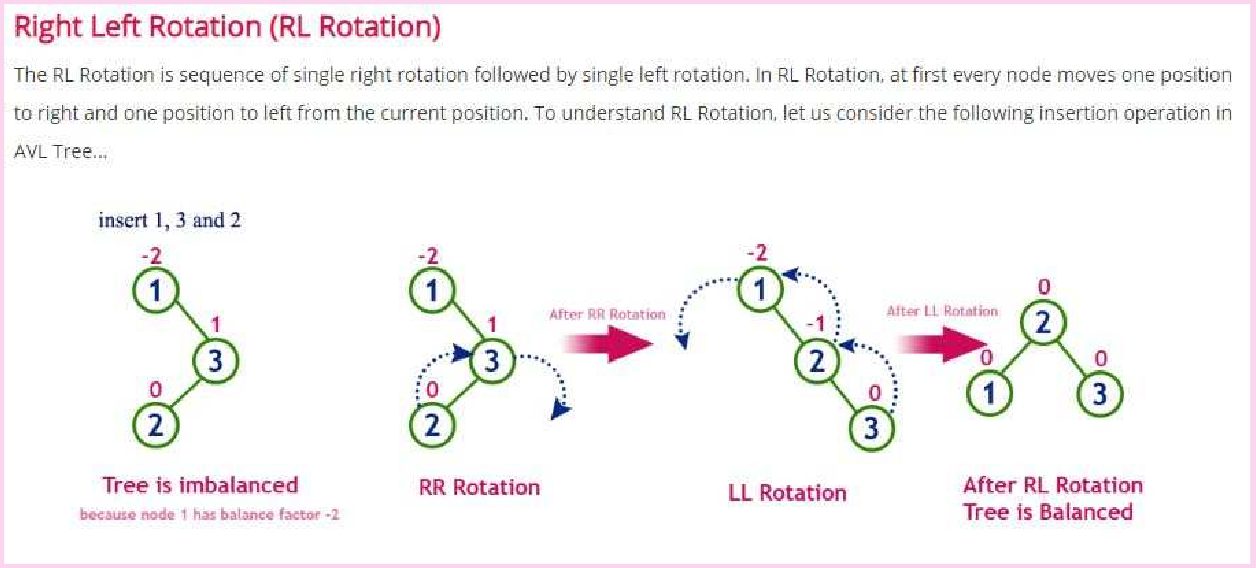
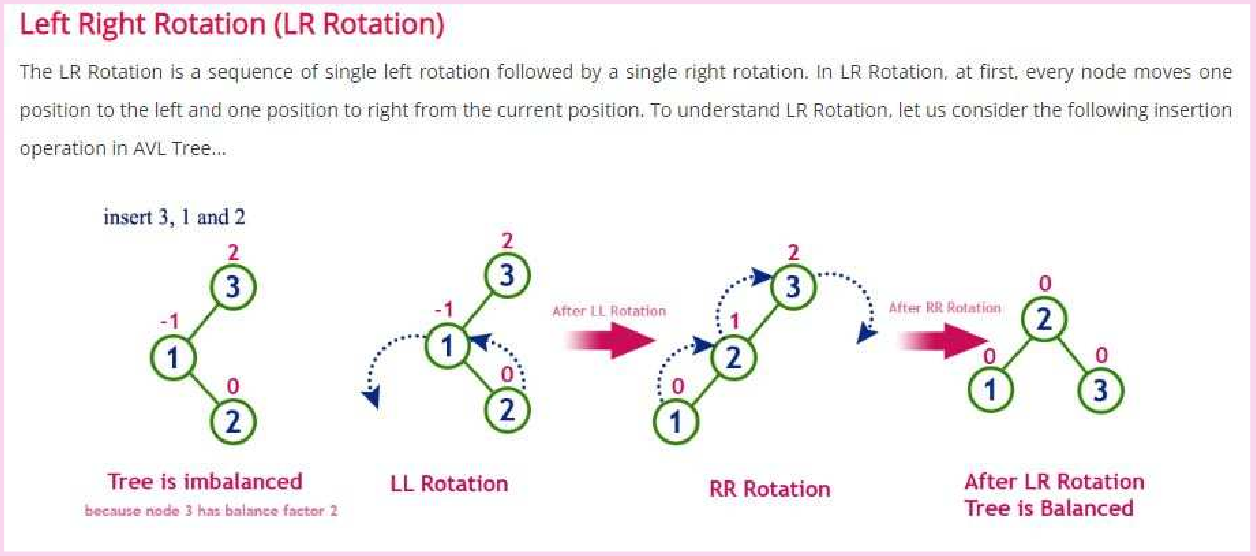
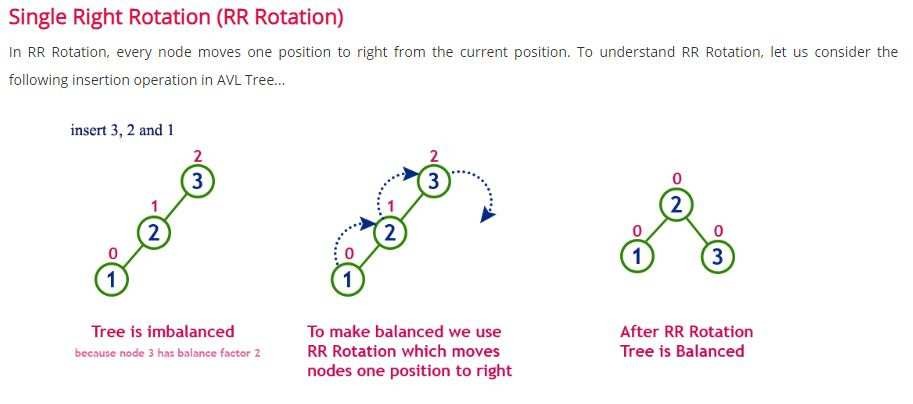
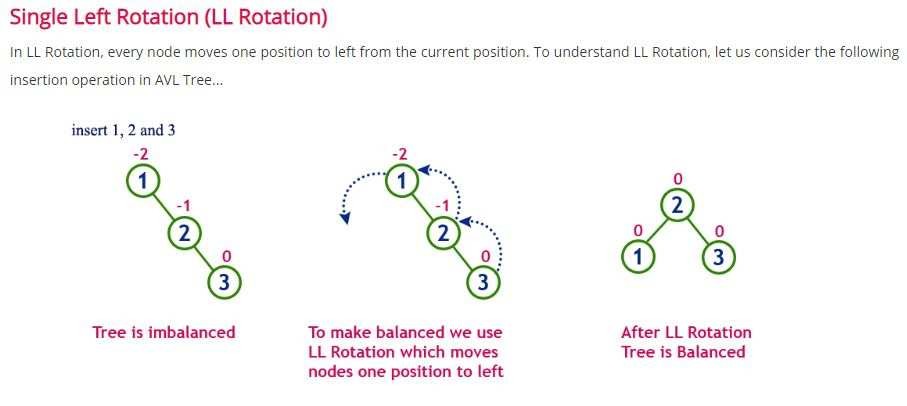
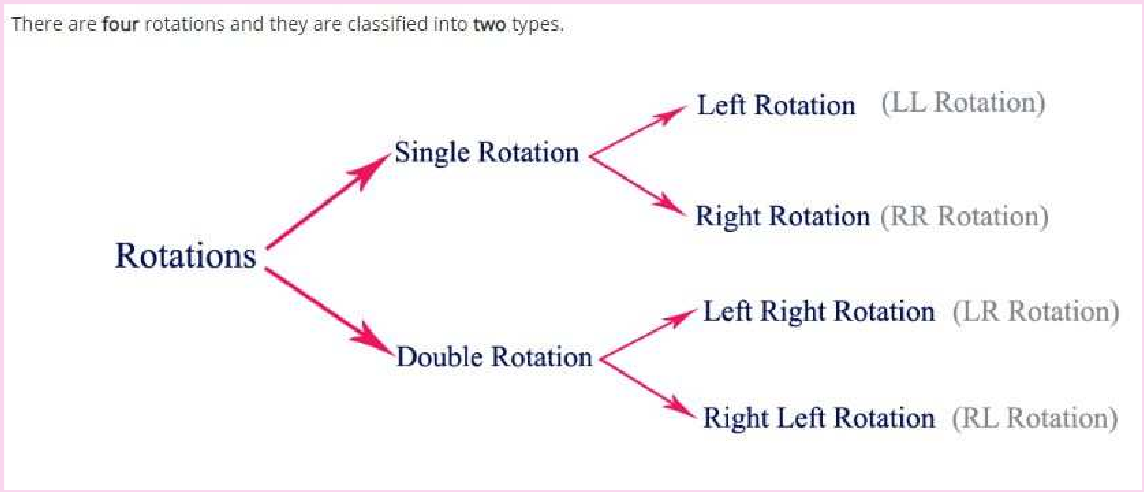
To insert a node with a key Q in the binary tree, the algorithm requires seven comparisons, but if you insert the same key in AVL tree, from the above 1st figure, you can see that the algorithm will require three comparisons.

### COMPLEXITY



**ROTATIONS IN AN AVL TREES:**

Rotation is the process of moving nodes either to left or to right to make the tree balanced.



## OPERATIONS ON AN AVL TREE:

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

#### The following operations are performed on AVL tree...

1. Search
2. Insertion
3. Deletion

**Search Operation in AVL Tree**

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

* + Step 1 - Read the search element from the user.
  + Step 2 - Compare the search element with the value of root node in the tree.
  + Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
  + Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
  + Step 5 - If search element is smaller, then continue the search process in left subtree.
  + Step 6 - If search element is larger, then continue the search process in right subtree.
  + Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
  + Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
  + Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

### Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

* + Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
  + Step 2 - After insertion, check the Balance Factor of every node.
  + Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
  + Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

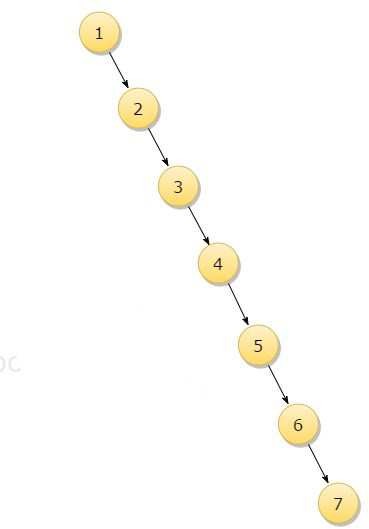
### Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance

Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

**EXAMPLE PROBLEMS:**

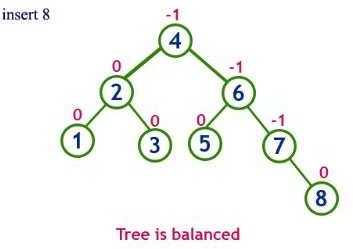
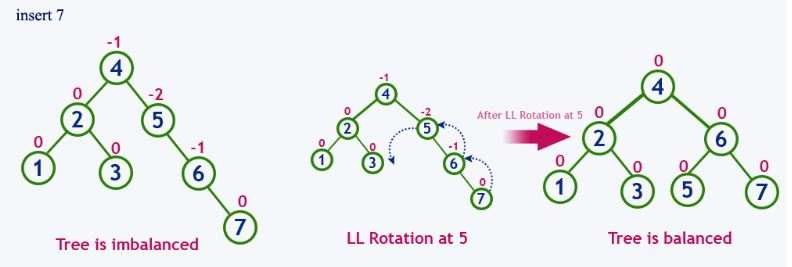
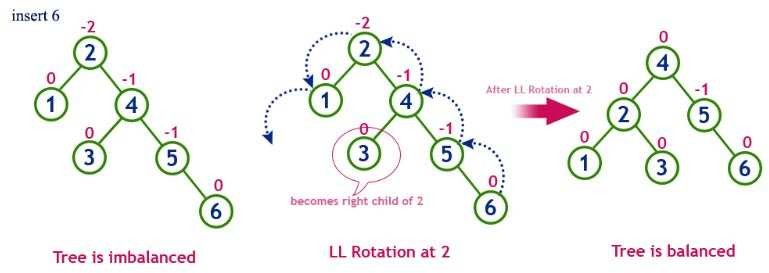
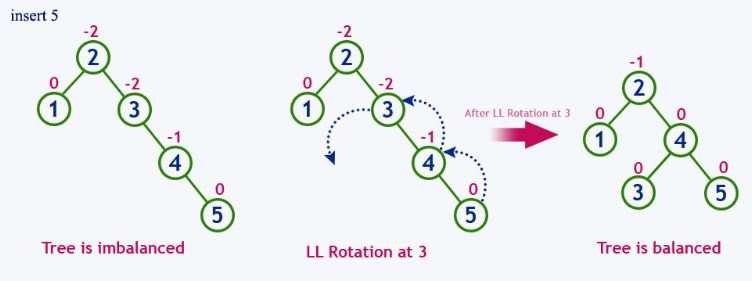
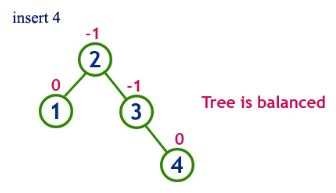
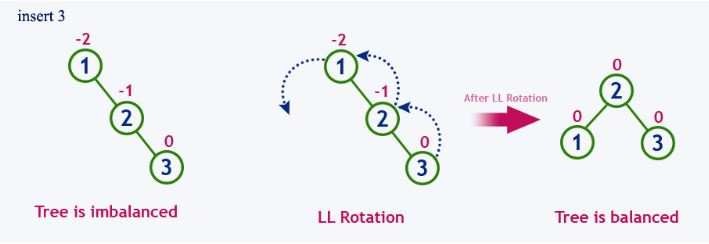
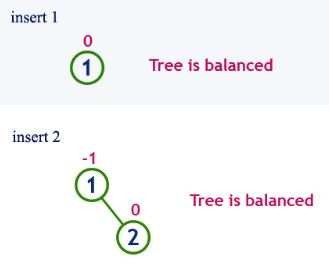
### Q1.CONSTRUCT BST - 1,2,3,4,5,6,7.



**RESULT:**

**In this case, It becomes a Skew Tree. If we try to search 7, It's gonna take O(n) time. To improve complexity, we'll re-construct this tree(similar) using AVL Tree Properties.**

**Q2. Construct an AVL Tree by inserting numbers from 1 to 8.**

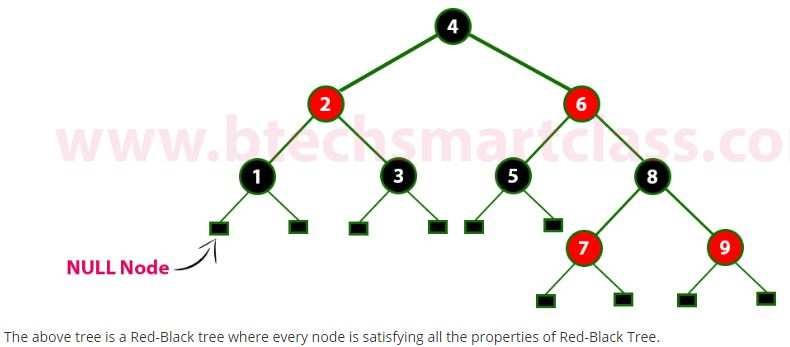


**RED-BLACK TREES**

Red-Black Tree is a self-balancing Binary Search Tree (BST) in which every node is colored either RED or BLACK.

In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties:

* + Property #1: Red - Black Tree must be a Binary Search Tree.
  + Property #2: The ROOT node must be colored BLACK.
  + Property #3: The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
  + Property #4: In all the paths of the tree, there should be same number of BLACK colored nodes.
  + Property #5: Every new node must be inserted with RED color.
  + Property #6: Every leaf (i.e, NULL node) must be colored BLACK.



**NOTE**: Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.

### How does a Red-Black Tree ensure balance?

A simple example to understand balancing is, a chain of 3 nodes is not possible in the Red-Black tree. We can try any combination of colours and see all of them violate Red-Black tree property.



A chain of 3 nodes is nodes is not possible in Red-Black Trees.

Following are **NOT** Red-Black Trees

Following are different possible Red-Black Trees with above

3 keys

### Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of a Red-Black tree is always O(Logn) where n is the number of nodes in the tree.

### Comparison with AVL Tree

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red

Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

### Advantages of Red Black Tree

1. Red black tree are useful when we need insertion and deletion relatively frequent.
2. Red-black trees are self-balancing so these operations are guaranteed to be O(logn).
3. They have relatively low constants in a wide variety of scenarios. **NOTE**: Every Red Black Tree with n nodes has height <= 2Log2(n+1) **Applications of Red Black Tree**
4. Most of the self-balancing BST library functions like map and set in C++ (OR TreeSet and TreeMap in Java) use Red Black Tree
5. It is used to implement CPU Scheduling Linux. Completely Fair Scheduler uses it.

## Insertion into RED BLACK Tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

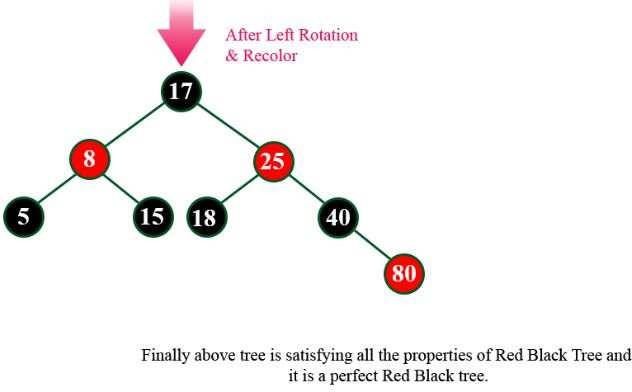
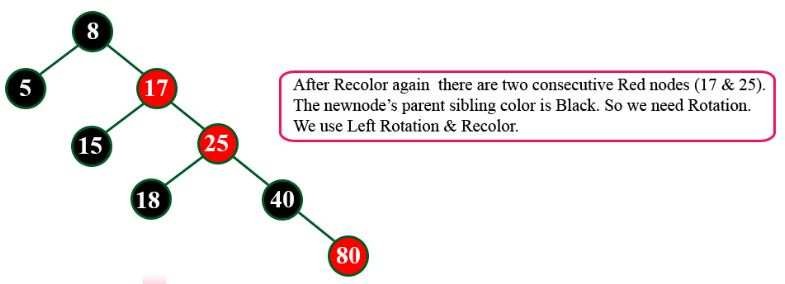
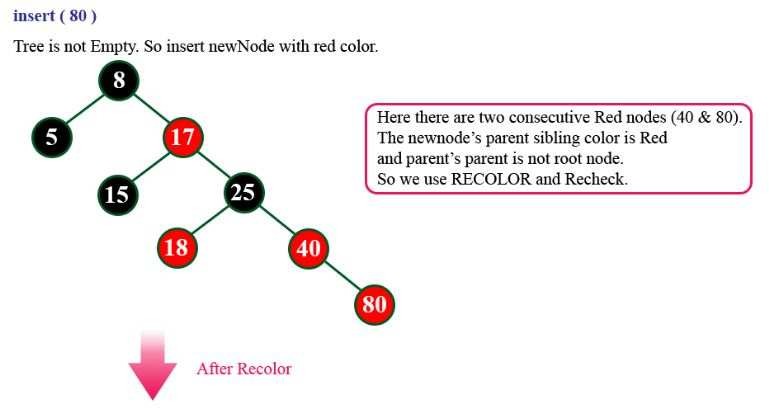
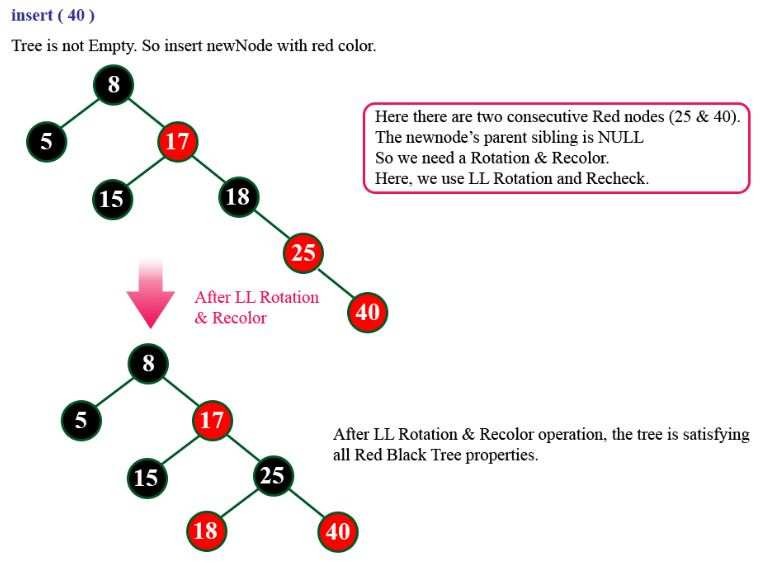
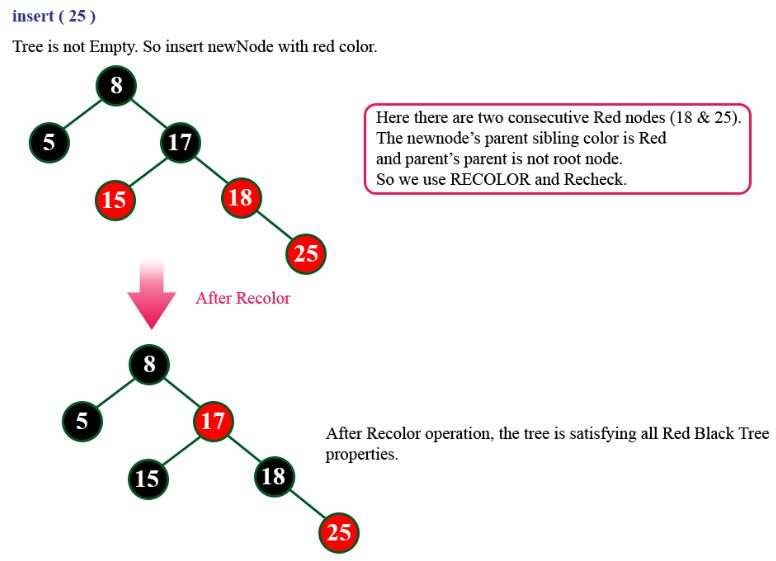
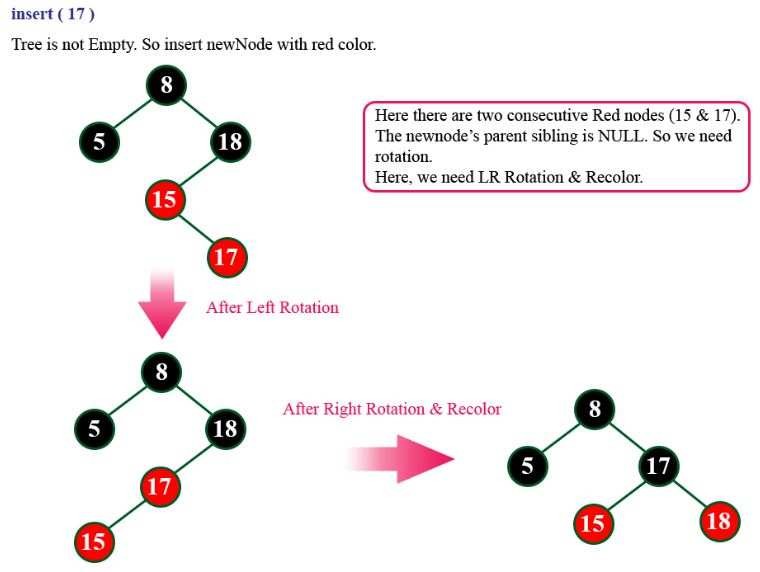
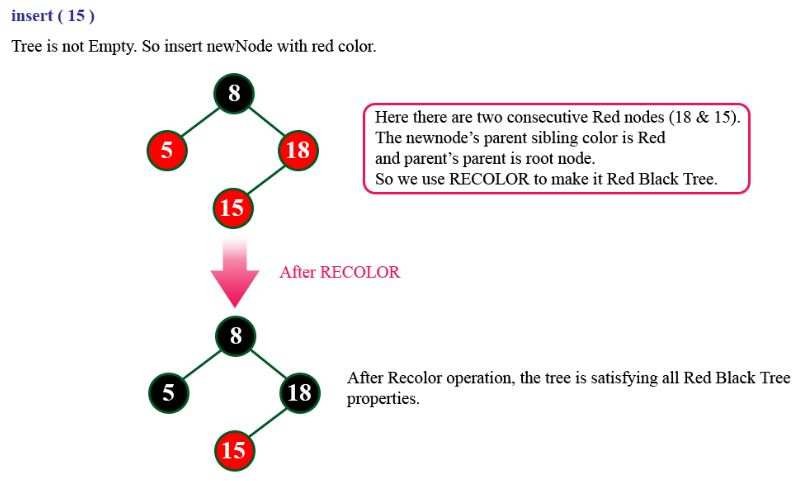
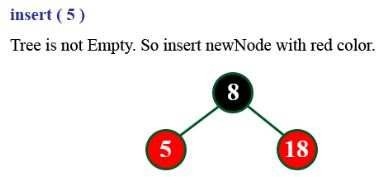
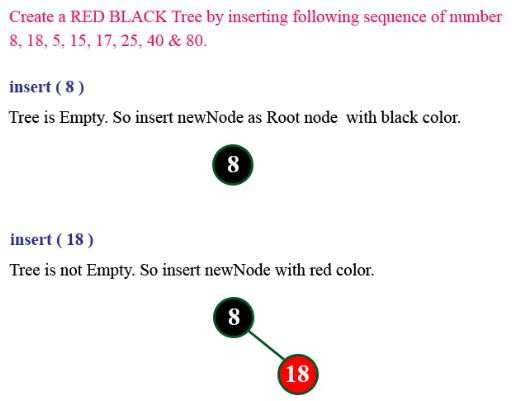
#### Recolor

* 1. **Rotation**
  2. **Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using the following steps...

* Step 1 - Check whether tree is Empty.
* Step 2 - If tree is Empty then insert the newNode as Root node with color Black and exit from the operation.
* Step 3 - If tree is not Empty then insert the newNode as leaf node with color Red.
* Step 4 - If the parent of newNode is Black then exit from the operation.
* Step 5 - If the parent of newNode is Red then check the color of parentnode's sibling of newNode.
* Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.
* Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

**EXAMPLE:**

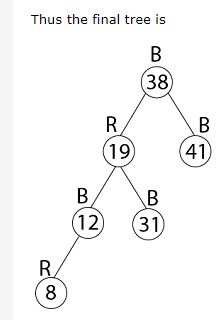
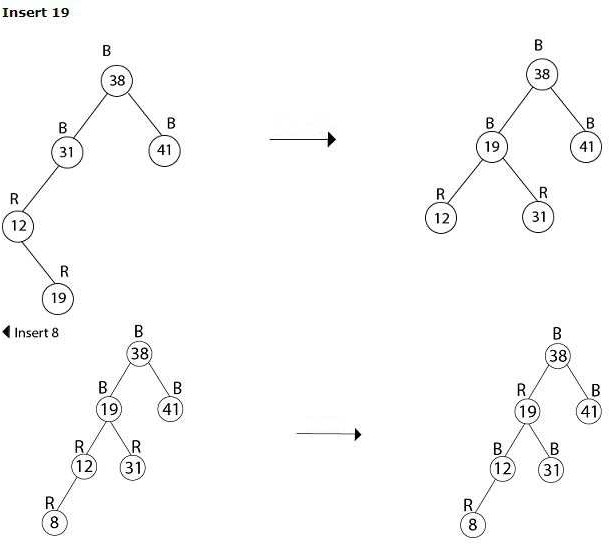
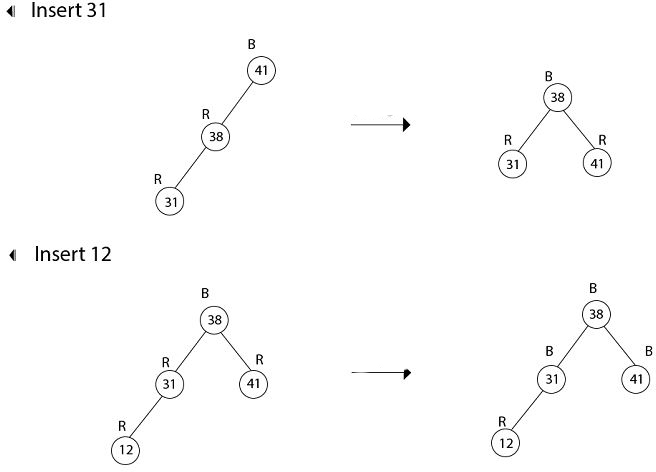
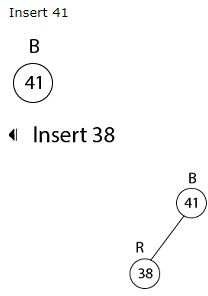


**Deletion Operation in Red Black Tree**

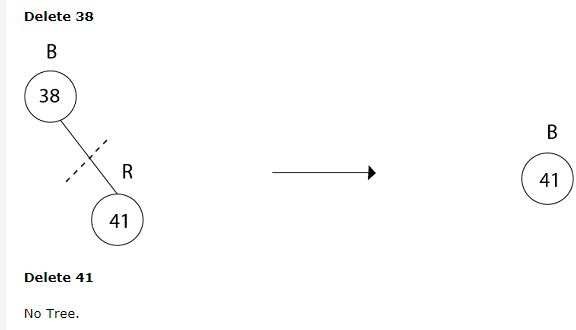
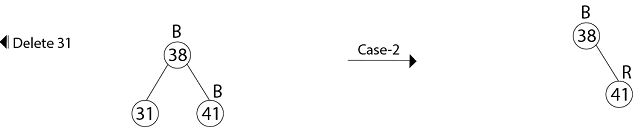
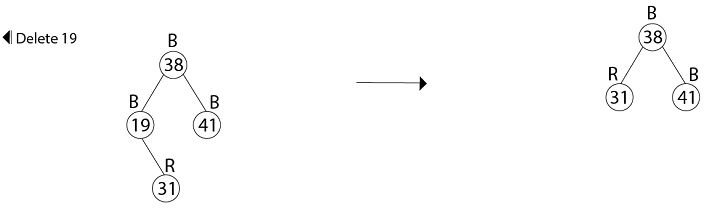
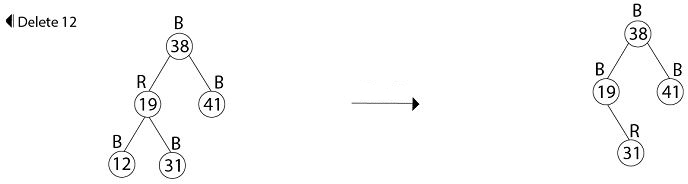
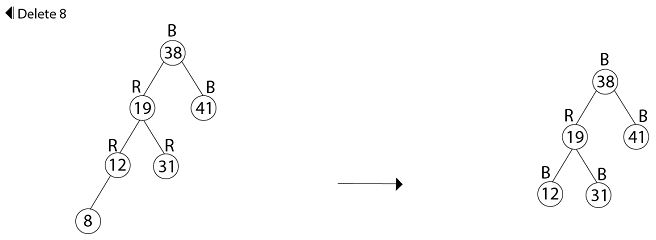
The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Red- Black Tree properties. If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

### EXAMPLE:

**Q. Show the red-black trees that result after successively inserting the keys 41,38,31,12,19,8 into an initially empty red-black tree.**



**Q. In a previous example, we found that the red-black tree that results from successively inserting the keys 41,38,31,12,19,8 into an initially empty tree. Now show the red-black trees that result from the successful deletion of the keys in the order 8, 12, 19,31,38,41.**



**For more examples of deletion:** https://[www.geeksforgeeks.org/red-black-tree-set-3-](http://www.geeksforgeeks.org/red-black-tree-set-3-) delete-2/

# SPLAY TREES:

**Definition**: Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is O(n). The worst case occurs when the tree is skewed. We can get the worst case time complexity as O(Logn) with AVL and Red-Black Trees.

### Can we do better than AVL or Red-Black trees in practical situations?

Like AVL and Red-Black Trees, Splay tree is also self-balancing BST. The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

All splay tree operations run in O(log n) time on average, where n is the number of entries in the tree. Any single operation can take Theta(n) time in the worst case.

**Splaying**

*Splaying* an element, is the process of bringing it to the root position by performing suitable rotation operations.

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process

and then splaying that searched element so that it is placed at the root of the tree.

In splay tree, to splay any element we use the following rotation operations...

## Rotations in Splay Tree

* 1. Zig Rotation
* 2. Zag Rotation
* 3. Zig - Zig Rotation
* 4. Zag - Zag Rotation
* 5. Zig - Zag Rotation
* 6. Zag - Zig Rotation

## Zig Rotation

The Zig Rotation in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...



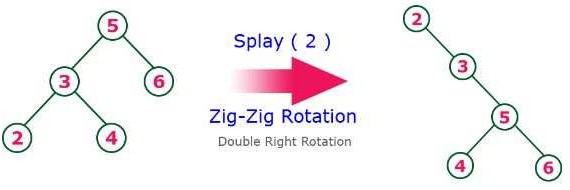
## Zag Rotation

The Zag Rotation in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...



## Zig-Zig Rotation

The Zig-Zig Rotation in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



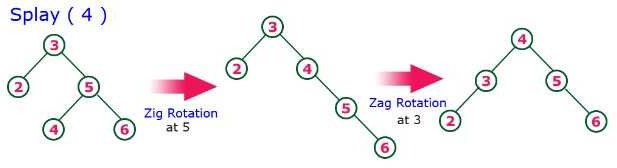
## Zag-Zag Rotation

The Zag-Zag Rotation in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



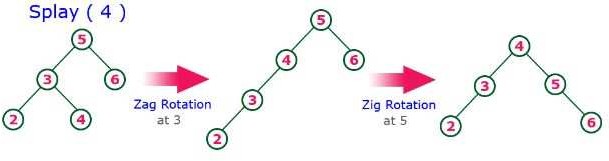
## Zig-Zag Rotation

The Zig-Zag Rotation in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



## Zag-Zig Rotation

The Zag-Zig Rotation in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



**NOTE:** Every Splay tree must be a binary search tree but it is need not to be balanced tree.

## Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

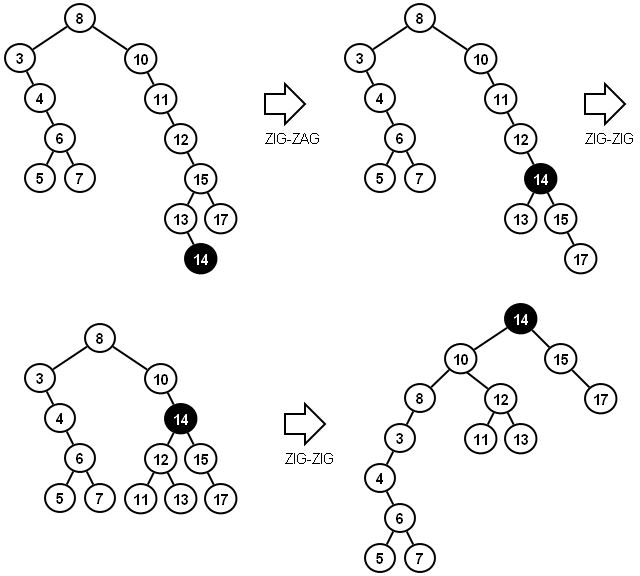
* Step 1 - Check whether tree is Empty.
* Step 2 - If tree is Empty then insert the newNode as Root node and exit from the operation.
* Step 3 - If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
* Step 4 - After insertion, Splay the newNode

## Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to splay that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

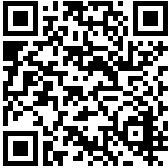
## EXAMPLE:

**Here's an example We're splaying the 14:**



**VISUALIZATION LINKS:**

**BINARY SEARCH TREE:** https://[www.cs.usfca.edu/~galles/visualization/BST.html](http://www.cs.usfca.edu/~galles/visualization/BST.html)



**AVL TREE:** https://[www.cs.usfca.edu/~galles/visualization/AVLtree.html](http://www.cs.usfca.edu/~galles/visualization/AVLtree.html)



**RED BLACK TREE:** https://[www.cs.usfca.edu/~galles/visualization/RedBlack.html](http://www.cs.usfca.edu/~galles/visualization/RedBlack.html)



**SPLAY TREE:** https://[www.cs.usfca.edu/~galles/visualization/SplayTree.html](http://www.cs.usfca.edu/~galles/visualization/SplayTree.html)

