UNIT-II

Extended Binary Tree and Threaded Binary Trees with focus on In-order Threading, Representation of Algebraic Expressions and its implementation. Representation and Creation of Binary Search Trees (BST), implementation of insertion, deletion and searching in BST.

Heaps: Introduction, Types of Heaps – Min binary heap, Max binary heap.

****

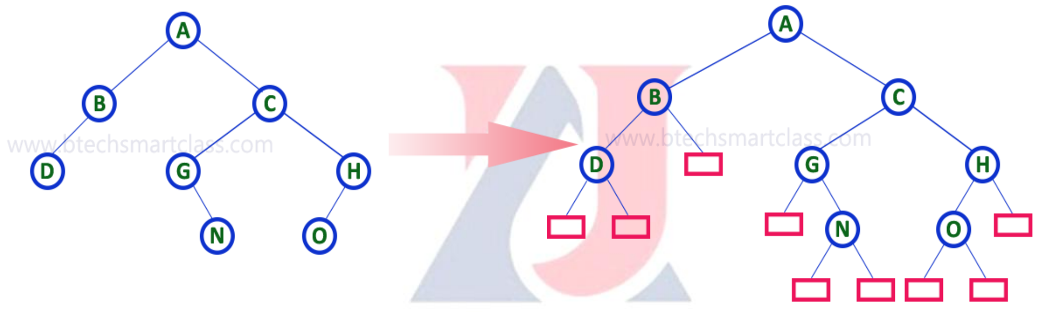
Extended Binary Tree:

*Extended binary tree is a type of binary tree in which all the null sub tree of the original tree are replaced with special nodes called **external nodes** whereas other nodes are called **internal nodes***

(or)

***An Extended Binary Tree, also known as a 2-tree or a Strict Binary Tree, is a specialized type of binary tree where every node has either exactly zero or exactly two children*.**

EX-1



In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

Key characteristics:

- **Internal Nodes:**

These are the nodes that originated from the "regular" binary tree and have exactly two children.

- **External Nodes (or Null Nodes):**

These are special nodes added to represent the null subtrees of the original binary tree. They have no children and act as leaf nodes in the extended structure.

How it's formed:

An extended binary tree is created by replacing every null subtree (where a node would have a missing child) in a standard binary tree with an external node. This process ensures that all original nodes (now internal nodes) have two children, and all newly added external nodes have zero children.

Properties:

- All internal nodes have a degree of two (two children).
- All external nodes have a degree of zero (no children) and are leaf nodes.
- The resulting structure often resembles a complete binary tree.

Applications:

Extended binary trees are useful in various applications, including:

➢ **Representation of algebraic expressions:**

They can effectively represent expressions where operations have two operands.

➢ **Calculating weighted path length:**

In scenarios where external nodes store weights, the total path length can be easily calculated.

➢ **Converting to a complete binary tree:**

By adding external nodes, any binary tree can be represented in a form resembling a complete binary tree, which is beneficial for certain data structures like heaps.

Threaded Binary Trees:

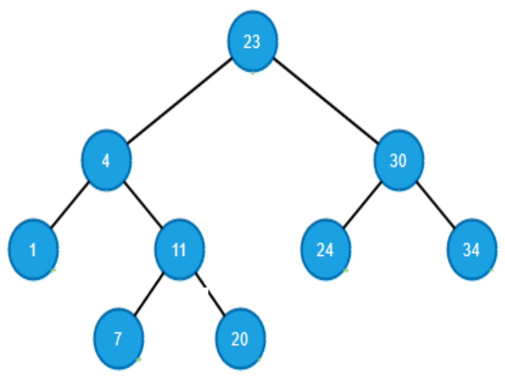
➢ In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space.

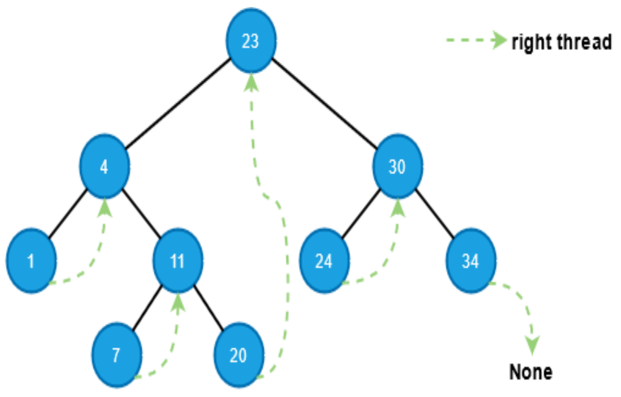
➢ If a binary tree consists of n nodes then n+1 link fields contain NULL values. So in order to effectively manage the space, in which the NULL links are replaced with special links known as **threads**. Such binary trees with threads are known as threaded binary trees.

"Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor."

a) **Single Threaded Binary Tree**: A single threaded binary tree is a binary tree in which a NULL right pointer is made to point to the **In-order successor** (Right thread).

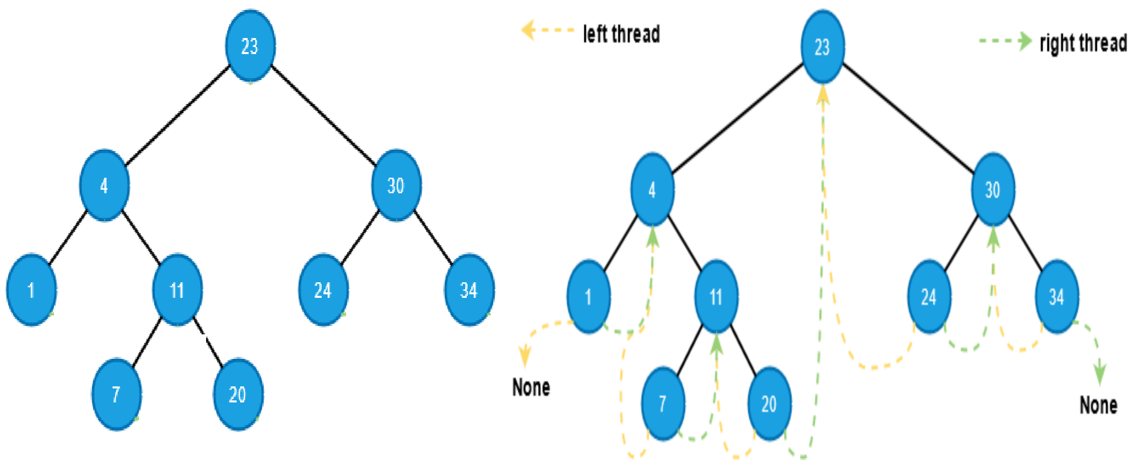
Example: In-order of the give binary tree: **1 4 7 11 20 23 24 30 34**





b) **Double Threaded Binary Tree:** A double threaded binary tree is a binary tree in which both the left and right NULL pointers are made to point to the **In-order predecessor** and **In-order successor** respectively.

Example: In-order of the give binary tree: **1 4 7 11 20 23 24 30 34**



In-order threading : In-order threading in a binary tree replaces null pointers with "threads" that point to a node's **inorder successor** or **predecessor,** enabling efficient in-order traversal without recursion or a stack. This technique is particularly useful when memory is limited or when frequent in-order traversals are required.

Benefits of In-order Threading:

- **Reduced Memory Usage:**

Eliminates the need for a stack during in-order traversal, which can be beneficial when memory is a constraint.

- **Improved Traversal Speed:**

Enables faster in-order traversal, especially when compared to recursive approaches.

- **Efficient for BSTs:**

Particularly useful in Binary Search Trees (BSTs) for finding the inorder successor or predecessor.

Advantages of Threaded Binary Tree

Let's discuss some advantages of a Threaded Binary tree.

- **No need for stacks or recursion:** Unlike binary trees, threaded binary trees do not require a stack or recursion for their traversal.
- **Optimal memory usage:** Another advantage of threaded binary tree data structure is that it decreases memory wastage. In normal binary trees, whenever a node's left/right pointer

is NULL, memory is wasted. But with threaded binary trees, we are overcoming this problem by storing its inorder predecessor/successor.

- **Time complexity:** In-order traversal in a threaded binary tree is fast because we get the next node in O(1) time than a normal binary tree that takes O(Height). But insertion and deletion operations take more time for the threaded binary tree.
- **Backward traversal:** In a double-threaded binary tree, we can even do a backward traversal.

Disadvantages of Threaded Binary tree:

Let's discuss some disadvantages that might create a problem for a programmer using this.

- **Complicated insertion and deletion:** By storing the inorder predecessor/ successor for the node with a null left/right pointer, we make the insertion and deletion of a node more time-consuming and a highly complex process.
- **Extra memory usage:** We use additional memory in the form of rightThread and leftThread variables to distinguish between a thread from an ordinary link. (However, there are more efficient methods to differentiate between a thread and an ordinary link)

Applications of Threaded Binary Tree:

- There are some applications of threaded binary trees:
- **Efficient In-Order Traversal**: Threaded binary trees enhance in-order traversal efficiency by eliminating the need for recursion or stacks
- **Space Efficiency:** Threaded trees save space by eliminating null pointers, reducing memory overhead
- **Simplified Tree Traversal:** Threaded structures simplify and streamline tree traversal algorithms
- **Fast Searching and Retrieval:** Threaded binary trees enable faster navigation, improving the performance of search operations
- **Threaded Tree-based Iterators**: Threaded trees are useful for implementing efficient **iterators for various tree traversal orders**
- **Binary Search Tree Operations**: Threaded trees enhance efficiency in operations like finding minimum/maximum elements or predecessors/successor

Representation of Algebraic expressions

An expression tree is a binary tree used to represent expressions where the leaves represent operands (constants or variables) and the internal nodes represent operators. Here are the rules for constructing and representing an expression tree:

➢ **Nodes:**

- **Leaf Nodes**: Represent operands (constants or variables).
- **Internal Nodes:** Represent operators (+, -, *, /, etc.).

➢ **Binary Operators:**

- Each binary operator (e.g., +, -, *, /) must have exactly two children.
- The left child represents the left operand.
- The right child represents the right operand.

➢ **Parentheses:**

- Parentheses are implicit in the structure of the tree.
- The tree structure determines the order of operations.

➢ **Order of Operations:**

- The tree inherently respects the order of operations (precedence of operators).
- Operations higher up in the tree (closer to the root) are performed later.

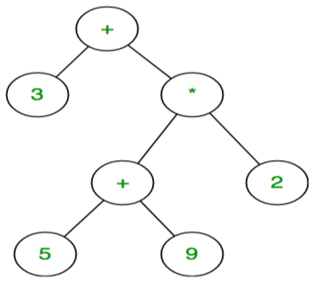
Example:1: ((3 + 5) * (7 - 2))

1. Create the tree:
   - The root is *.
   - The left child of * is + with children 3 and 5.
   - The right child of * is - with children 7 and 2.

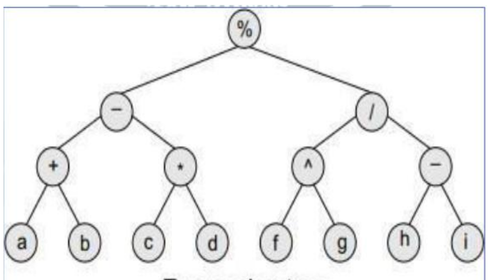
```
     *
    / \
   +   -
  / \ / \
 3  5 7  2
```

2. Traversals:
   - In-order: 3 + 5 * 7 - 2
   - Pre-order: * + 3 5 - 7 2
   - Post-order: 3 5 + 7 2 - *

Example:2: 3 + ((5+9)*2)

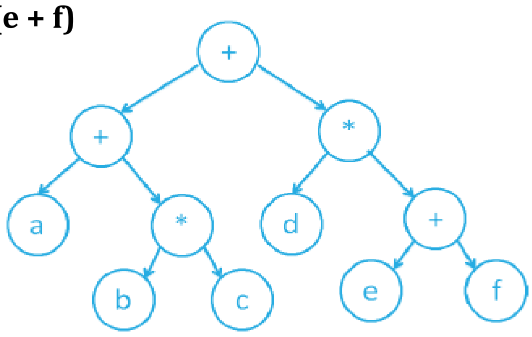


Example:3: **((a + b) – (c * d)) % ((e ^f) / (g – h))**



Expression tree

Example:4: **a + (b * c) + d * (e + f)**



Implementation of Algebraic Expressions

Algorithm: Construct Expression Tree from Infix Expression

In infix notation, operators are placed between operands. This is the most common way of writing arithmetic expressions. The construction process of Infix to Expression Tree is as follows:

1. Start with the lowest precedence operator (if multiple operators are present) and make this operator the root of the tree.
2. The left subtree is created from the part of the expression to the left of the operator, and the right subtree is created from the part of the expression to the right of the operator.
3. Recursively apply this process to the left and right subexpressions

Example: A + B * C

- The **+** has a lower precedence than *****, so **+** becomes the **root.**
- Operand **A** becomes the left child of **+.**
- The expression **B*C** becomes the right subtree. Since ***** has the highest precedence, it is the root of this subtree, and B and C are its children.

```
      +
     / \
    A   *
       / \
      B   C
```

Algorithm: Construct Expression Tree from Prefix Expression

1. **Start from the Rightmost character** of the prefix expression.
2. **Scan the expression in reverse (right to left)**.
3. **For each character**:
    - If it is an **operand**:
        - Create a **node.**
        - Push it to a **stack.**
    - If it is an **operator**:
        - Create a new node with the operator.
        - **Pop two nodes** from the stack.
        - Assign them as **left** and **right** children respectively.
        - Push this new node back to the stack.
4. **Final node on stack** is the **root** of the expression tree.

Example: + A * B C

- Push ‘A’, ‘B’, and ‘C’ onto the stack.
- Pop ‘B’ and ‘C’, create a node with *, and push it back onto the stack.
- Pop ‘A’ and the * node, create a node with + and push it back onto the stack.
- The final stack contains the root of the tree.

```
      +
     / \
    A   *
       / \
      B   C
```

Algorithm: Construct Expression Tree from Postfix Expression

1. **Initialize an empty stack**
2. **Scan the postfix expression from left to right**
3. For each symbol in the expression:

   If the symbol is an operand:

   - Create a **new node** with the symbol
   - Push it onto the stack

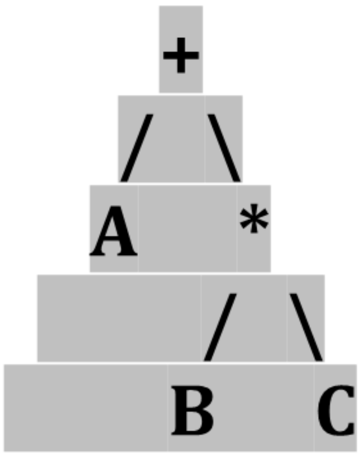
   If the symbol is an operator:

   - Pop two nodes from the stack
   - Create a **new node** with the operator
   - Set the second-popped node as the **left child**
   - Set the first-popped node as the **right child**
   - Push the new node back onto the stack

4. At the end, the only node left in the stack is the **root of the expression tree**

Example: A B C * +

- Push 'A', 'B', and 'C' onto the stack.
- Pop 'B' and 'C', create a node with *, and push it back onto the stack.
- Pop 'A' and the * node, create a node with + and push it back onto the stack.
- The final stack contains the root of the tree.



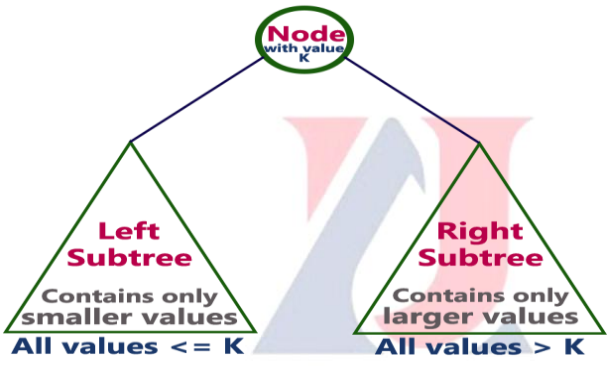
Binary Search Tree (BST)

In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right**.**

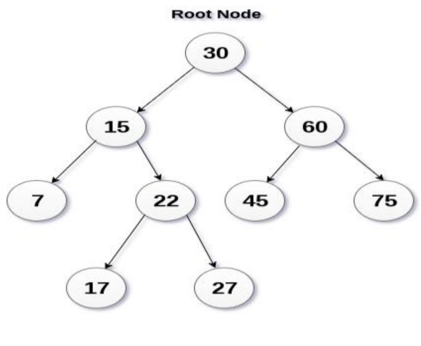
Binary Search Tree (BST):

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.

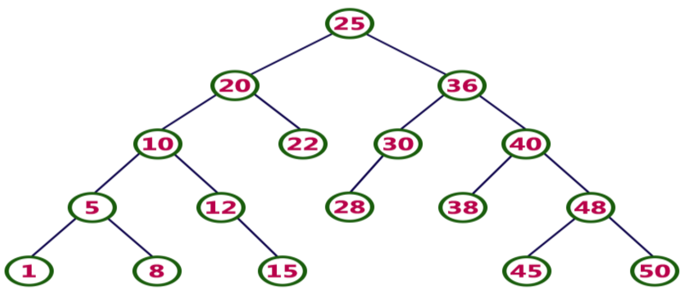
Binary Search Tree is a binary tree in which every node contains only smaller values in its left sub tree and only larger values in its right sub tree.



Example:



Binary Search Tree



Every binary search tree is a binary tree but every binary tree needs not to be binary search tree.

Operations on a Binary Search Tree:

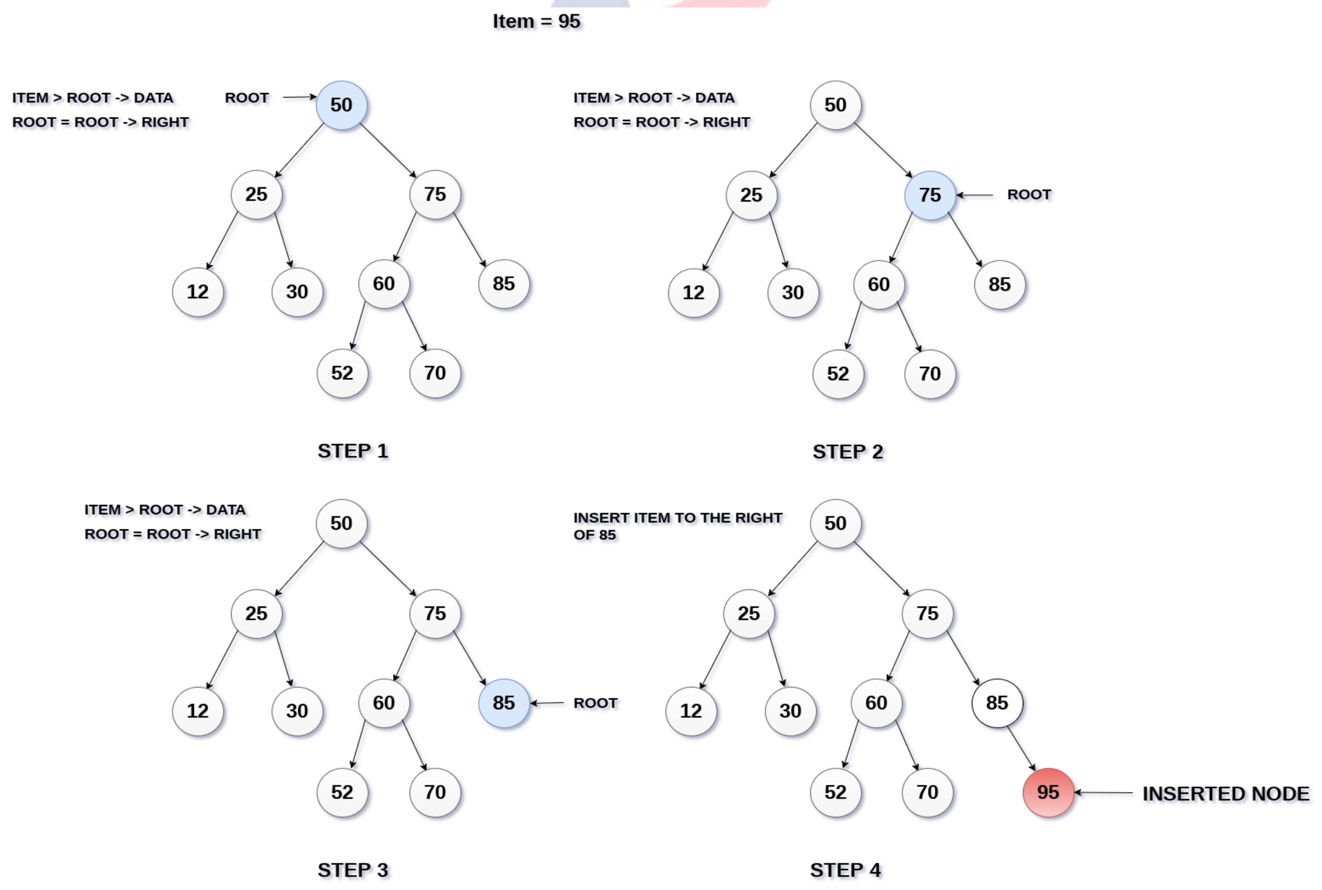
The following operations are performed on a binary search tree...

1. **Insertion**
2. **Search**
3. **Deletion**

1. Insertion:

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.**

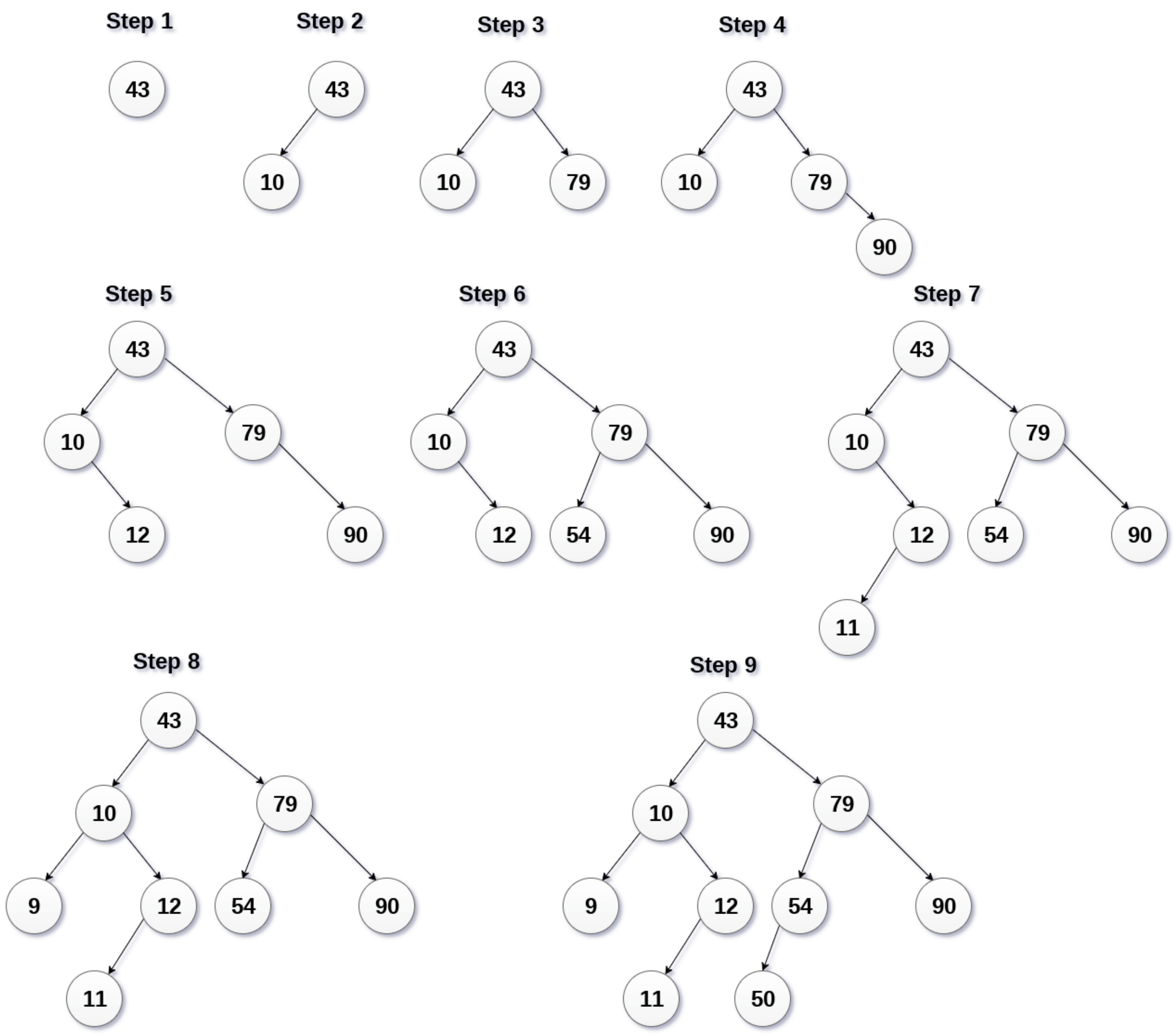


Example-1: Create the binary search tree using the following data elements.

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.

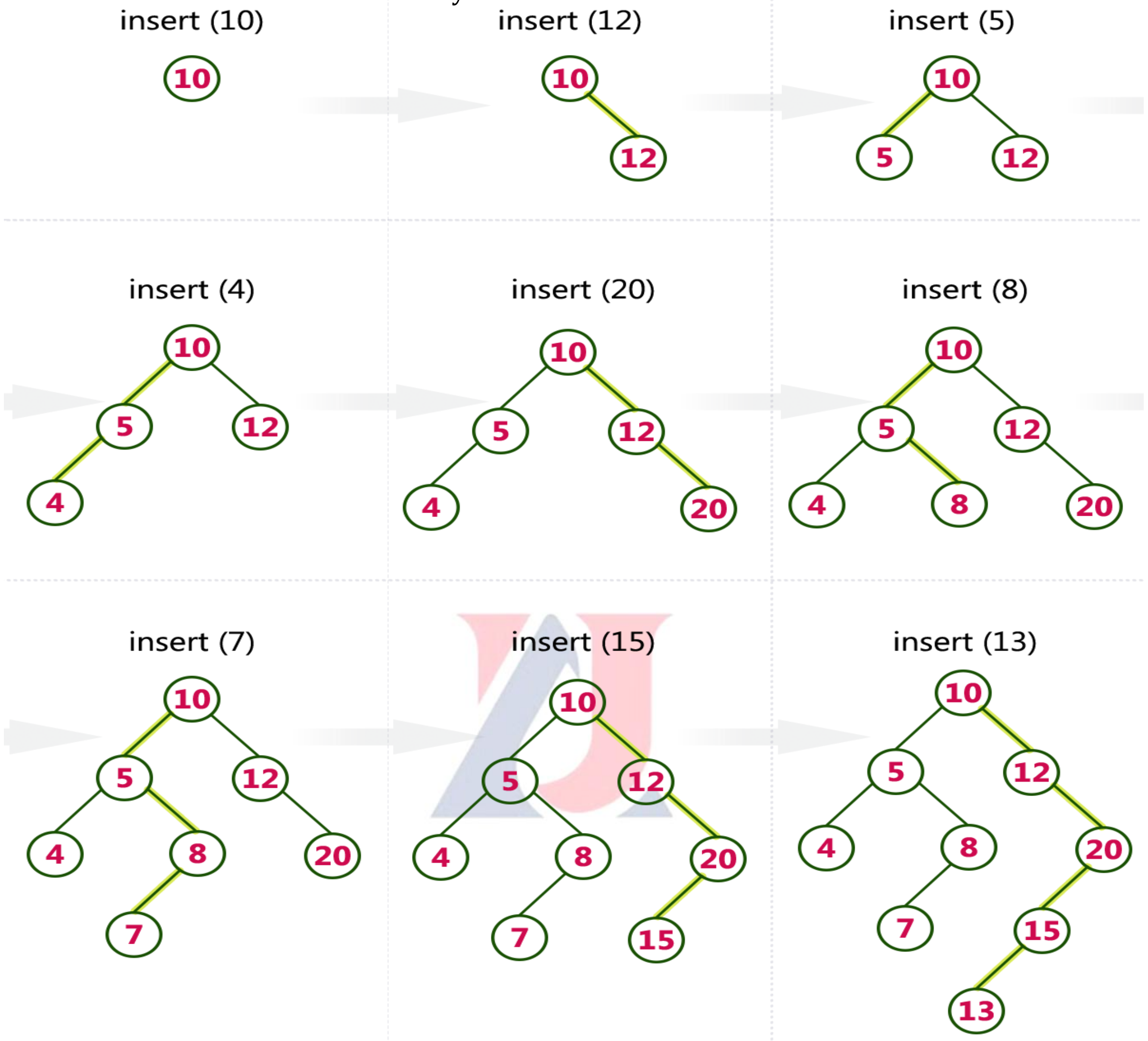
The process of creating BST by using the given elements is shown in the image below.



Example-2: Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...

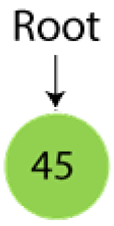


Example-3: Construct a Binary Search Tree by inserting the following sequence of numbers...

45, 15, 79, 90, 10, 55, 12, 20, 50

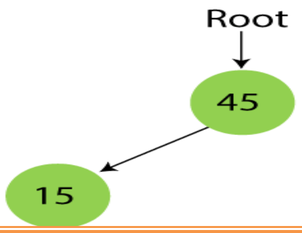
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.



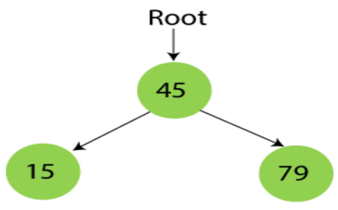
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left sub tree.



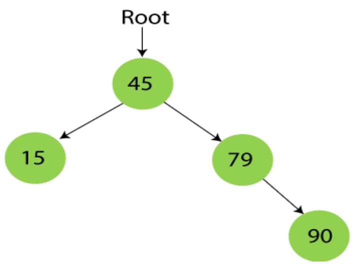
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right sub tree.



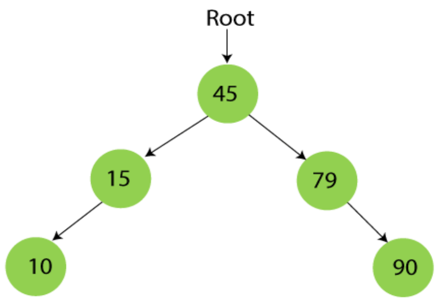
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right sub tree of 79.



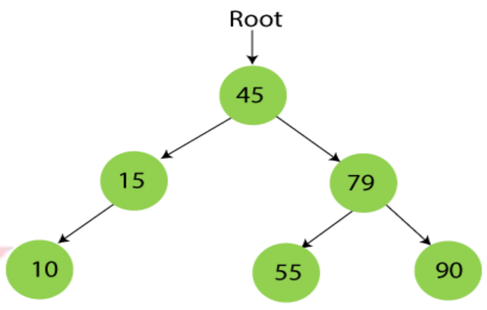
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left sub tree of 15.



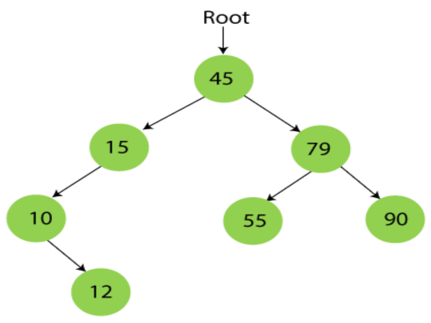
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left sub tree of 79.



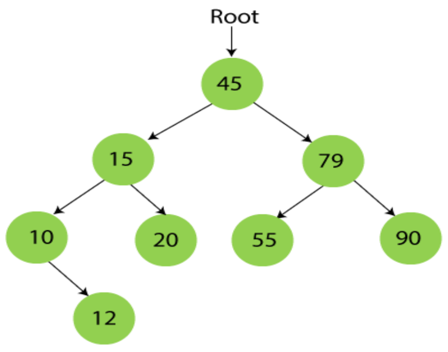
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right sub tree of 10.



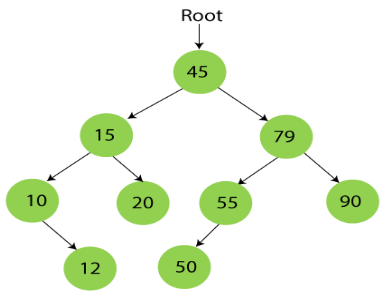
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right sub tree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left sub tree of 55.



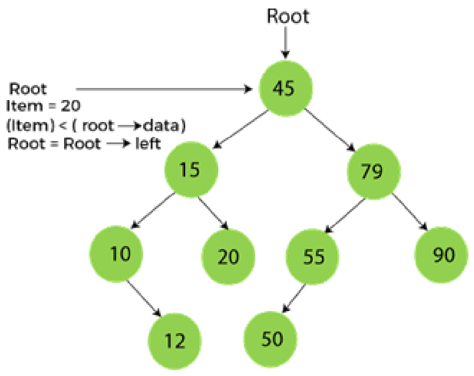
2. Searching:

Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST are stored in a particular order.

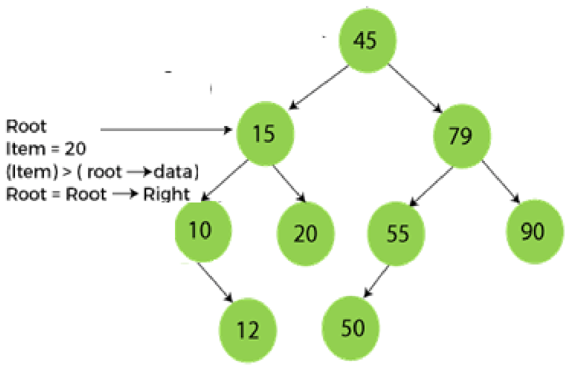
1. **Step 1 -** Read the search element from the user.
2. **Step 2 -** Compare the search element with the value of root node in the tree.
3. **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
4. **Step 4 -** If both are not matched, then check whether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

Example-1: Suppose we have to find node 20 from the below tree.

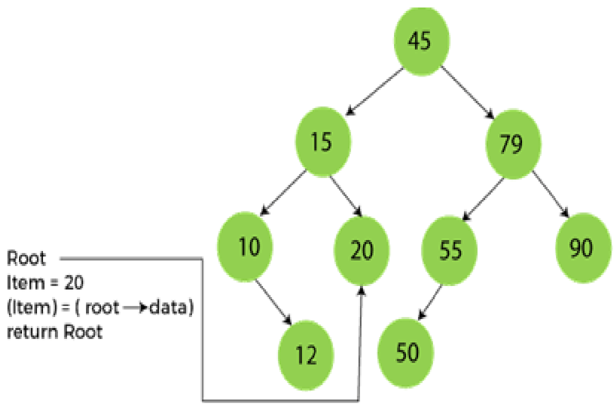
Step1:



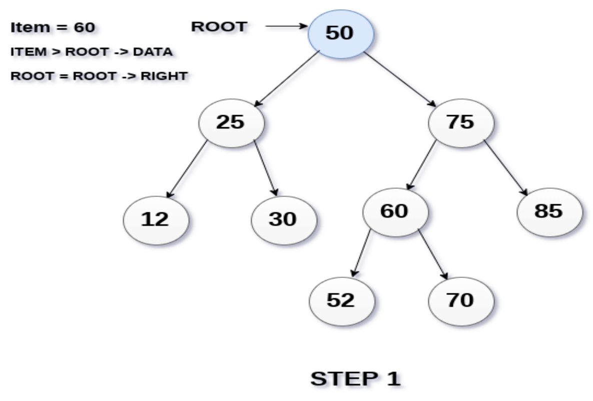
Step2:

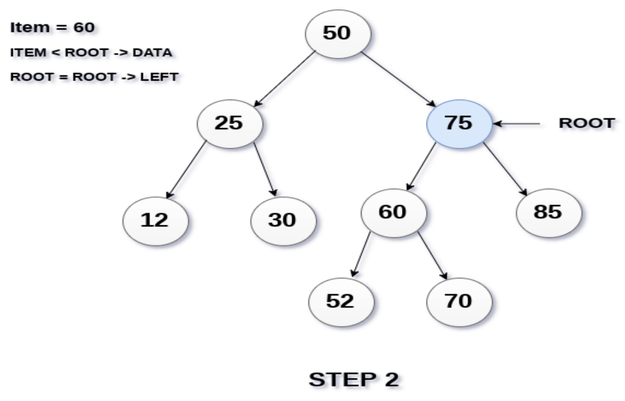


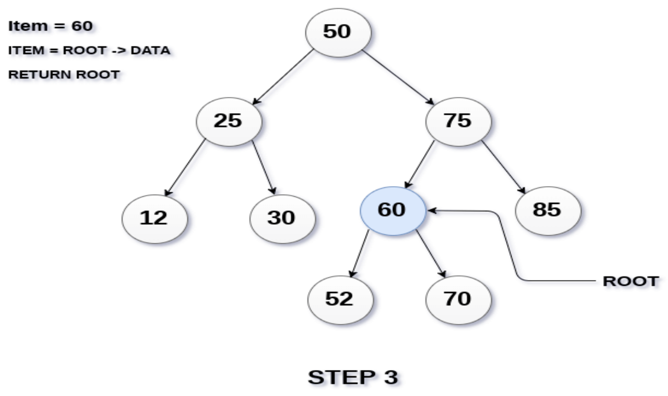
Step3:



Example-2: Suppose we have to find node 60 from the below tree







3. 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.

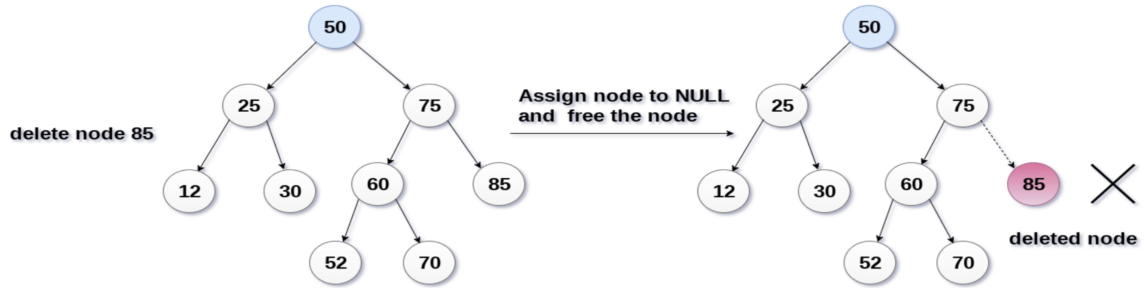
- **Case 1:** Deleting a Leaf node (A node with no children)
- **Case 2:** Deleting a node with one child
- **Case 3:** Deleting a node with two children

Case 1: **Deleting a leaf node:**

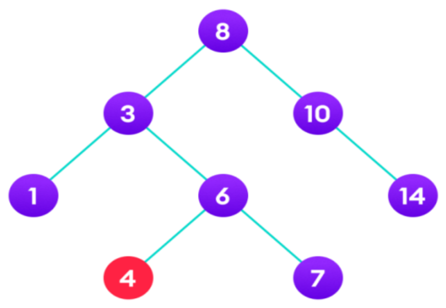
We use the following steps to delete a leaf node from BST...

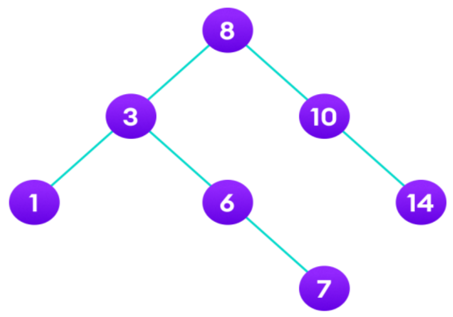
- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** in this case, replace the leaf node with the NULL and simple free the allocated space. Delete the node using **free** function (If it is a leaf) and terminate the function.

In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



In the following image, we are deleting the node 4, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



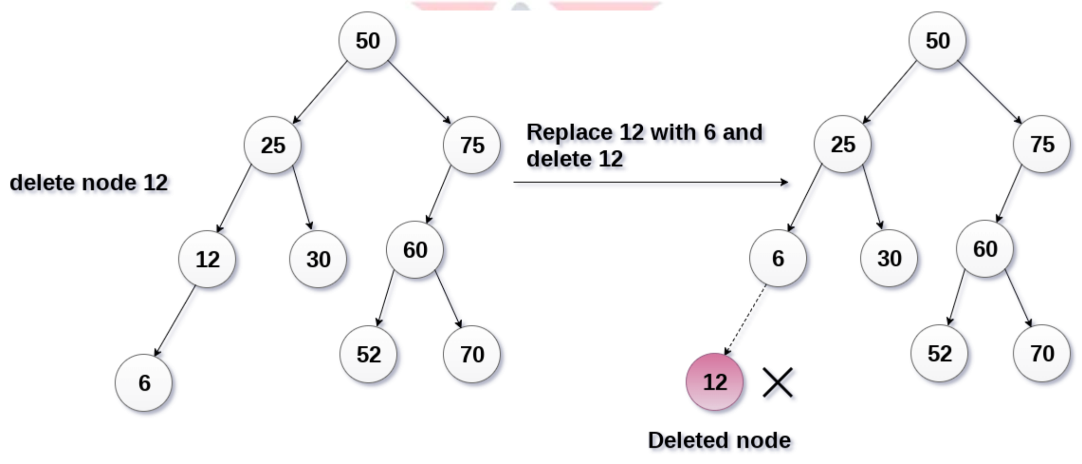


Case 2: **Deleting a node with one child:**

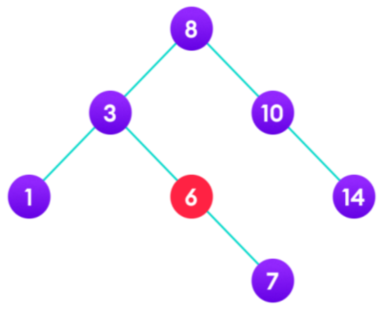
We use the following steps to delete a node with one child from BST...

- **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.

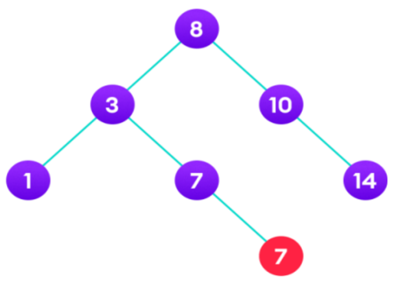
In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted



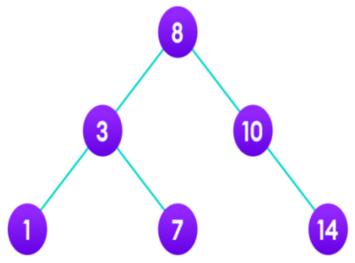
In the following image, the node 6 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 7 (which is now leaf node) will simply be deleted



6 is to be deleted



Copy the value of its child to the node and delete the child



Final tree

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 sub tree** (**In order Predecessor)** (OR) the **smallest** node in its **right sub tree (In order Successor).**
- **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. After the procedure, replace the node with NULL and free the allocated space.

(Or)

Step 1: Get the inorder successor of that node.

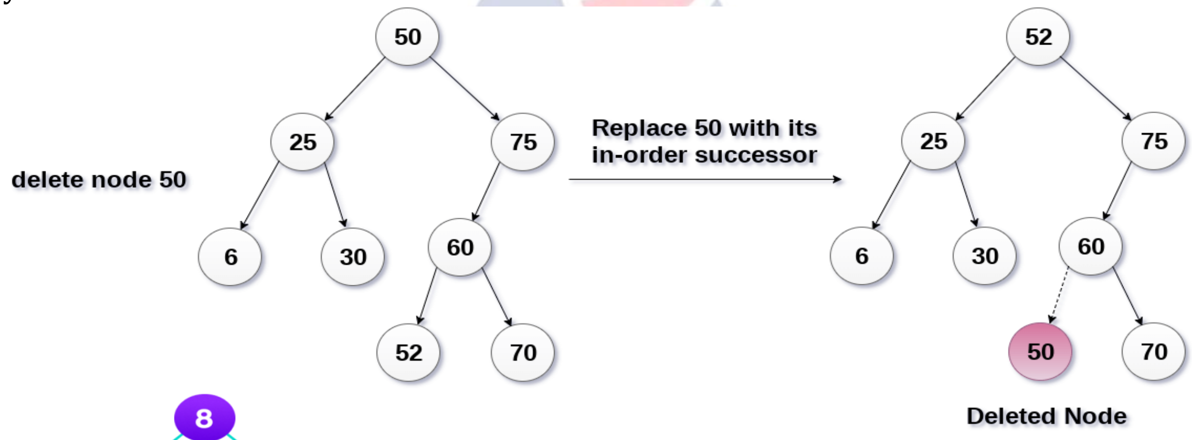
Step 2: Replace the node with the inorder successor.

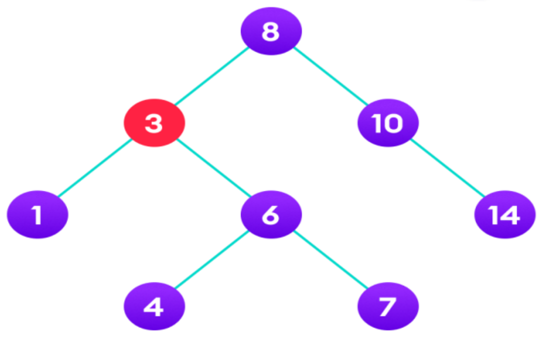
Step 3: Remove the inorder successor from its original position.

In the following image, the node 50 is to be deleted which is the root node of the tree.

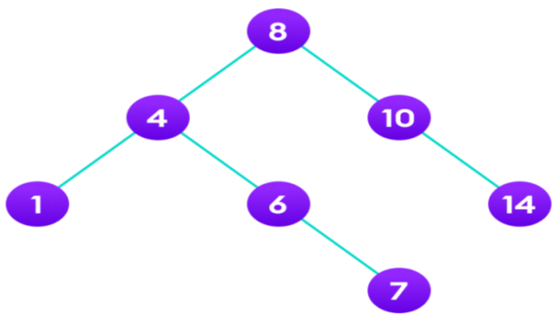
The **in-order traversal** of the tree given below. 6, 25, 30, 50, 52, 60, 70, 75.

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





3 is to be deleted. Copy the value of the in order successor (4) to the node



Delete the in order successor node (4)

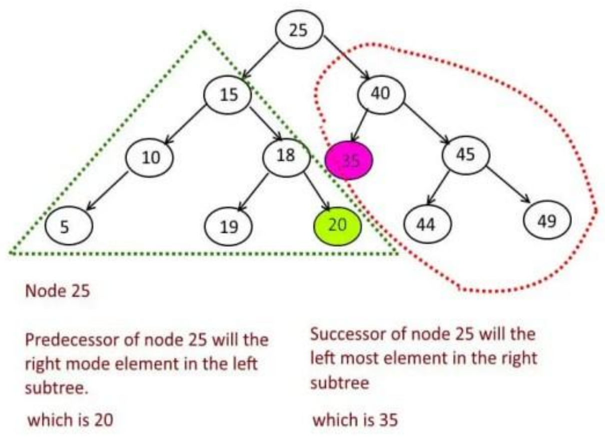
In order Predecessor and In order Successor :

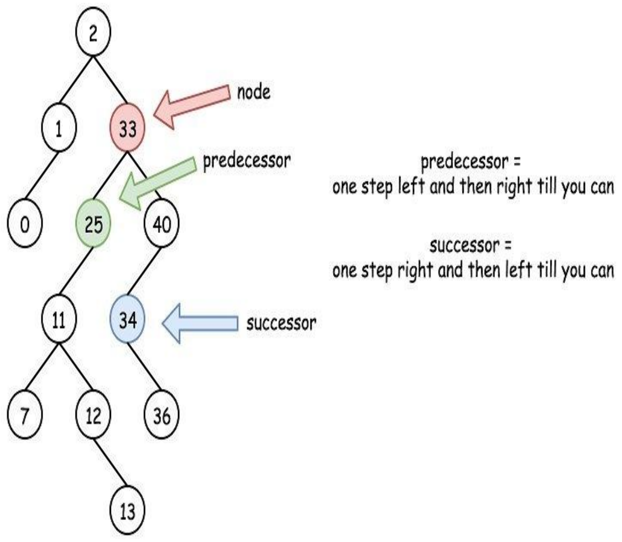
When you do the in order traversal of a binary tree, the neighbours of given node are called Predecessor (the node lies behind of given node) and Successor (the node lies ahead of given node).

Approach:

- First compare the node with root.
- Now predecessor will be the right most node in left sub tree
- Now successor will be the left most node in right sub tree

Example:


25
15
40
10
18
35
45
5
19
20
44
49
Node 25
Predecessor of node 25 will the right mode element in the left subtree.
which is 20
Successor of node 25 will the left most element in the right subtree
which is 35


2
1
33
node
predecessor
0
25
40
11
34
successor
7
12
36
13
predecessor = one step left and then right till you can
successor = one step right and then left till you can

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.

BST – Insertion, Deletion and Search Implementation

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int key;
struct node *left;
struct node *right;
};
//return a new node with the given value
struct node *getNode(int val)
{
struct node *newNode;
newNode=malloc(sizeof(struct node));

newNode->key=val;
newNode->left=NULL;
newNode->right=NULL;

return newNode;
}
```

Node Creation and allocating memory

```
//insert nodes in the Binary Search Tree
struct node *insertNode(struct node *root,int val)
{
if(root==NULL)
return getNode(val);

if(root->key<val)
root->right=insertNode(root->right,val);

if(root->key>val)
root->left=insertNode(root->left,val);

return root;
}
```

Inserting data/elements in BST Tree as Per Rules

```
//inoder traversal of the binary search tree

int inorder(struct node *root)
{
if(root==NULL)
return 0;

//travers the left subtree
inorder(root->left);

//visit the root
printf("%d-",root->key);

//traverse the right subtree
inorder(root->right);
return 0;
}
```

Display the elements in In-order Traversal

//preoder traversal of the binary search tree

```
int preorder(struct node *root)
{
if(root==NULL)
return 0;

//visit the root
printf("%d-",root->key);

//travers the left subtree
preorder(root->left);

//traverse the right subtree
preorder(root->right);
return 0;
}
```

Display the elements in Pre-order Traversal

//inoder traversal of the binary search tree

```
int postorder(struct node *root)
{
if(root==NULL)
return 0;

//travers the left subtree
postorder(root->left);

//traverse the right subtree
postorder(root->right);


//visit the root
printf("%d-",root->key);
return 0;
}
```

Display the elements in Post-order Traversal

```
int main()
{
struct node *root=NULL;
int data,d,choice;
char c;
while(1)
{
printf("\nselect one of the operations:");
printf("\n1.To insert a new node in the Binary Search Tree");
printf("\n2.To display the nodes in inorder");
printf("\n3.To display the nodes in prorder");
printf("\n4.To display the nodes in postorder");
 printf("\n");
printf("\nenter your choice");
scanf("%d",&choice);
switch(choice)
{
case 1:

printf("\nEnter the value to be inserted\n");
scanf("%d",&data);
root=insertNode(root,data);
break;

case 2:
printf("\n Inorder Traversal of the Binary Tree:\n");
inorder(root);
break;

case 3:
printf("\n Prorder Traversal of the Binary Tree:\n");
preorder(root);
break;

case 4:
printf("\n Inorder Traversal of the Binary Tree:\n");
postorder(root);
break;

default:
printf("Wrong Entry\n");
case 5: exit(0);
}
}
}
```

Heaps

What is a heap?

- A heap is a **complete binary tree,** and the binary tree is a tree in which the node can have the utmost two children.
- 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.

❖ In a heap data structure, nodes are arranged based on their values. A heap data structure sometimes also called as **Binary Heap.**

There are two types of heap data structures and they are as follows...

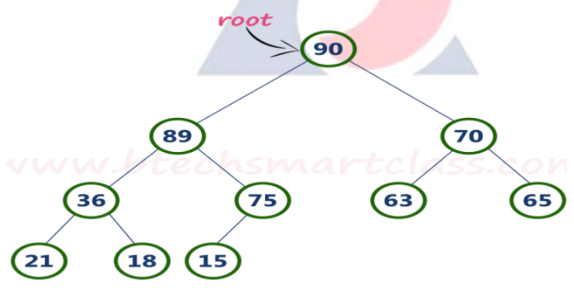
1. **Max Heap**
2. **Min Heap**

1. Max Heap:

Max heap is defined as follows...

Max heap is a specialized complete binary tree in which every parent node contains greater or equal value than its child nodes.

Example:



Heap elements representation in the form of array (**Level Order**)

90	89	70	36	75	63	65	21	18	15

Above tree is satisfying both Ordering property and Structural property according to the Max Heap data structure.

Operations on Max Heap

The following operations are performed on a Max heap data structure...

1. **Finding Maximum**
2. **Insertion**
3. **Deletion**

Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In a max heap, **the root node has the maximum value than all other nodes.** So, directly we can display root node value as the maximum value in max heap.

Insertion in Max Heap

Insertion Operation in max heap is performed as follows...

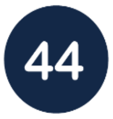
- **Step 1 -** Insert the **newNode** as **last leaf** from left to right.
- **Step 2 -** Compare **newNode value** with its **Parent node.**
- **Step 3 -** If **newNode value is greater** than its parent, then **swap** both of them.
- **Step 4 -** Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

Example-1: Construct a max heap of elements-
44, 33, 77, 11, 55, 88, 66

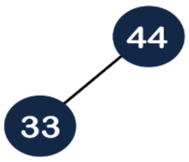
Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
- Secondly, the value of the parent node should be greater than the either of its child.

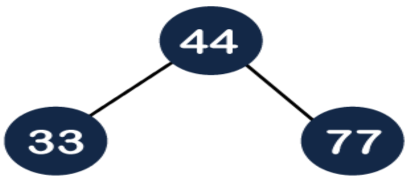
Step 1: First we add the 44 element in the tree as shown below:



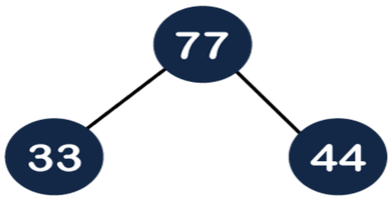
Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:



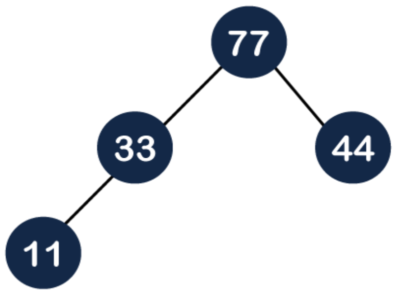
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



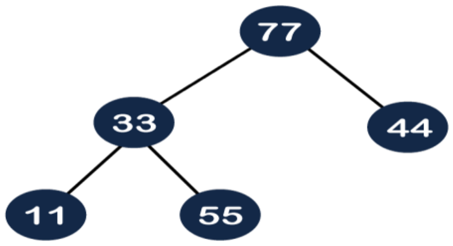
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



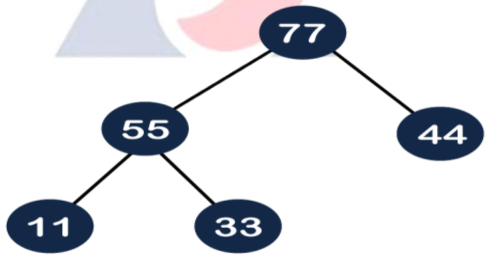
Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:



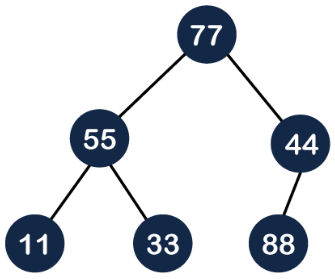
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:



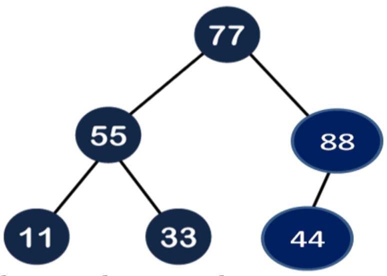
As we can observe in the above figure that it does not satisfy the property of the max heap because 33<55, so we will swap these two values as shown below:



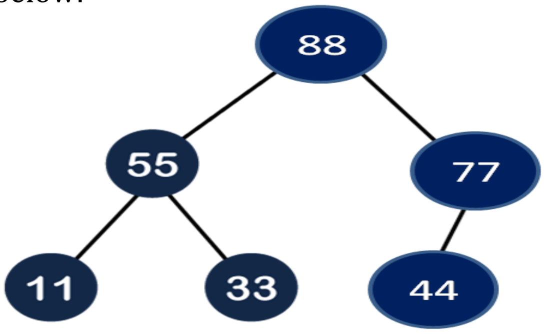
Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:



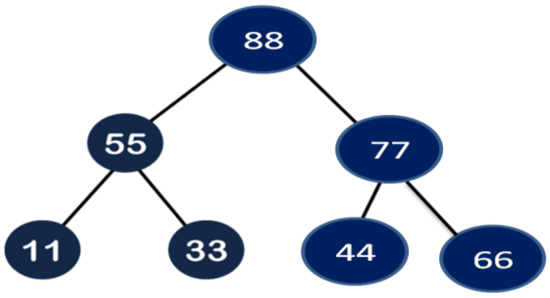
- ➢ As we can observe in the above figure that it does not satisfy the property of the max heap because 44<88, so we will swap these two values as shown below:



➢ Again, it is violating the max heap property because 88>77 so we will swap these two values as shown below:



Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:



In the above figure, we can observe that the tree satisfies the property of max heap; therefore, it is a heap tree.

Example-2: Construct a max heap FROM **array of elements**(**Method-2**)

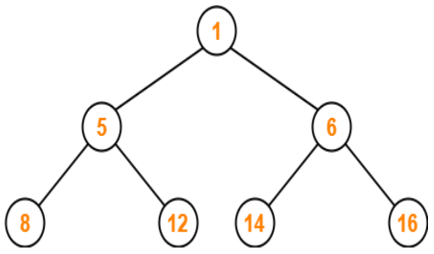
1, 5, 6, 8, 12, 14, 16

Represent the elements in the array

1	**5**	**6**	**8**	**12**	**14**	**16**

Step-01:

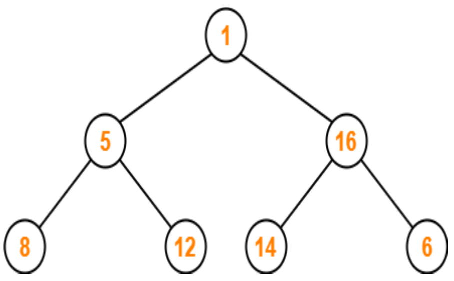
We convert the given array of elements into an almost **complete binary tree**-



Step-02:

- We ensure that the tree is a max heap.
- Node 6 contains greater element in its right child node.
- So, we swap node 6 and node 16.

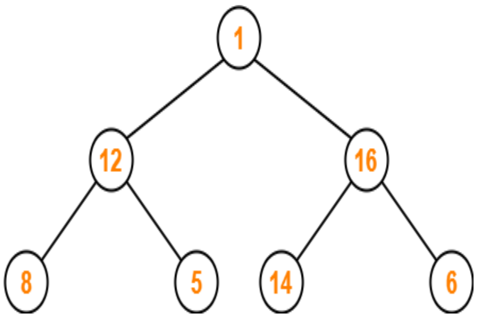
The resulting tree is-



Step-03:

- Node 5 contains greater element in its right child node.
- So, we swap node 5 and node 12.

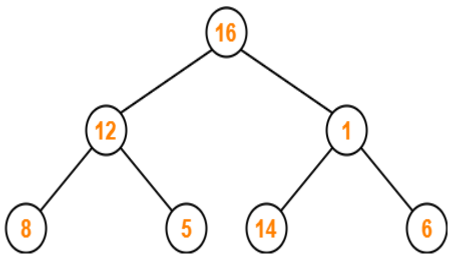
The resulting tree is-



Step-04:

- Node 1 contains greater element in its right child node.
- So, we swap node 1 and node 16.

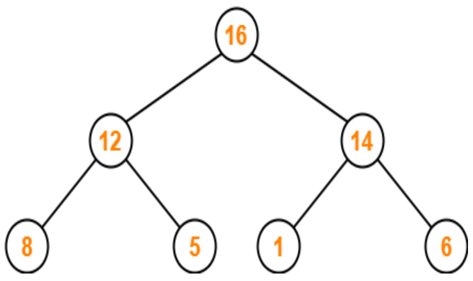
The resulting tree is-



Step-05:

- Node 1 contains greater element in its left child node.
- So, we swap node 1 and node 14.

The resulting tree is-



This is the required max heap for the given array of elements.

Example-2: Construct a max heap FROM array of elements-(**Method-2**)

50, 30, 20, 15, 10, 8, 16

Insert a new node with value 60.

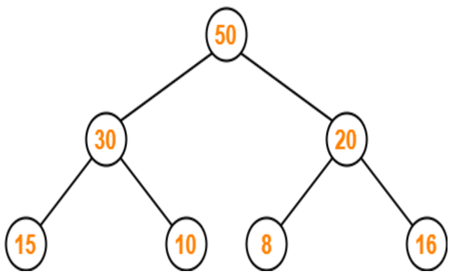
Represent the elements in the array

50	**30**	**20**	**15**	**10**	**8**	**16**

Step-01:

We convert the given array of elements into a heap tree-

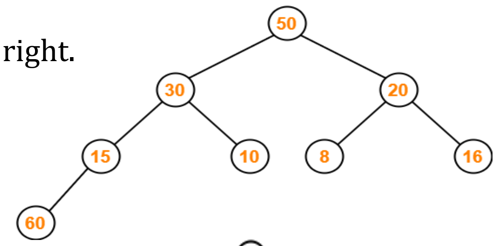
We convert the given array of elements into a heap tree-



Step-02:

We insert the new element 60 as a next leaf node from left to right.

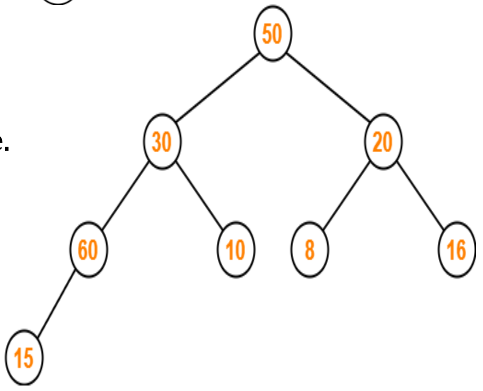
The resulting tree is-



Step-03:

- We ensure that the tree is a max heap.
- Node 15 contains greater element in its left child node.
- So, we swap node 15 and node 60.

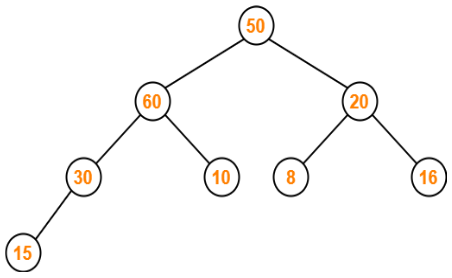
The resulting tree is-



Step-04:

- Node 30 contains greater element in its left child node.
- So, we swap node 30 and node 60.

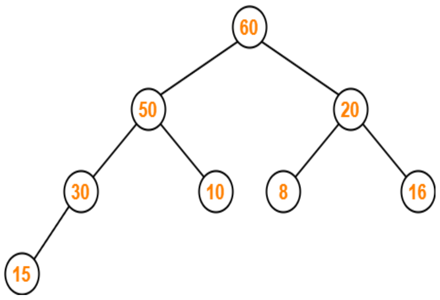
The resulting tree is-



Step-05:

- Node 50 contains greater element in its left child node.
- So, we swap node 50 and node 60.

The resulting tree is-



This is the required max heap after inserting the node with value 60.

Deletion Operation in Max Heap:

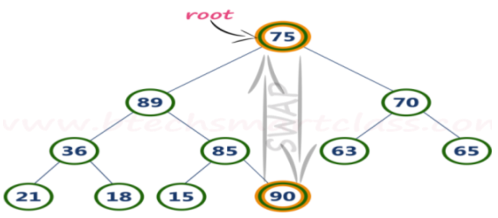
- ➢ In a max heap, deleting the last node is very simple as it does not disturb max heap properties.
- ➢ Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete the root node from a max heap...

- **Step 1 - Swap** the **root** node with **last** node in max heap
- **Step 2 - Delete** last node.
- **Step 3 -** Now, compare **root value** with its **left child value**.
- **Step 4 -** If **root value is smaller** than its left child, then compare **left child** with its **right sibling**. Else goto **Step 6**
- **Step 5 -** If **left child value is larger** than its **right sibling**, then **swap root** with **left child** otherwise **swap root** with its **right child**.

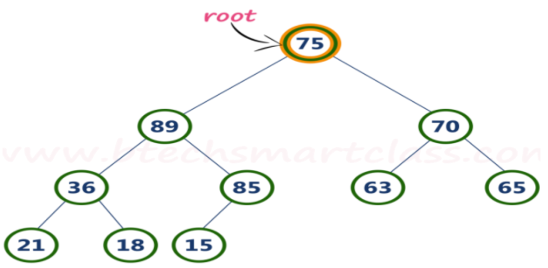
- **Step 6 -** If **root value is larger** than its left child, then compare **root value** with its **right child** value.
- **Step 7 -** If **root value is smaller** than its **right child**, then **swap root** with **right child** otherwise **stop the process**.
- **Step 8 -** Repeat the same until root node fixes at its exact position.

Example: Consider the above max heap. **Delete root node (90) from the max heap.**

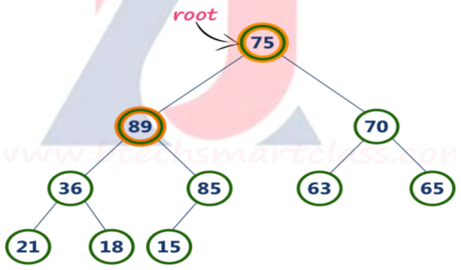
Step 1 - Swap the **root node (90)** with **last node 75** in max heap. After swapping max heap is as follows...



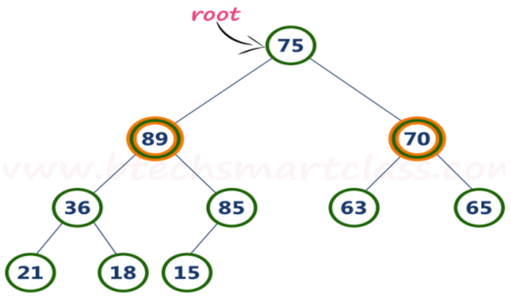
Step 2 - Delete last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...



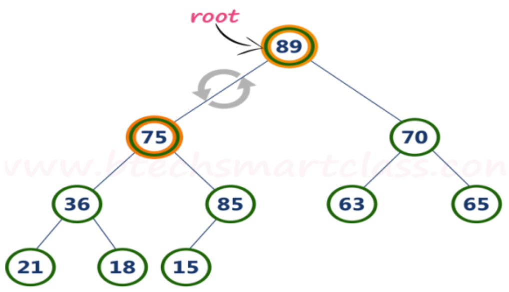
Step 3 - Compare **root node (75)** with its **left child (89)**



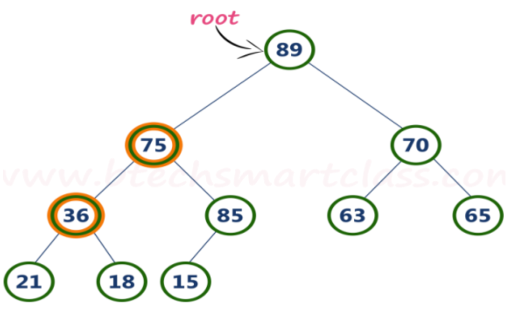
Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).



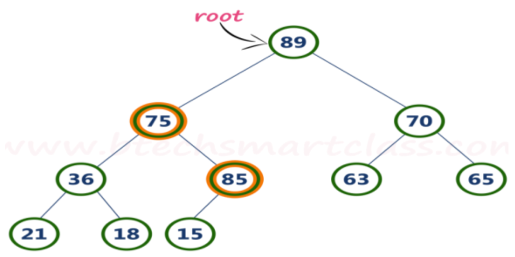
Step 4 - Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.



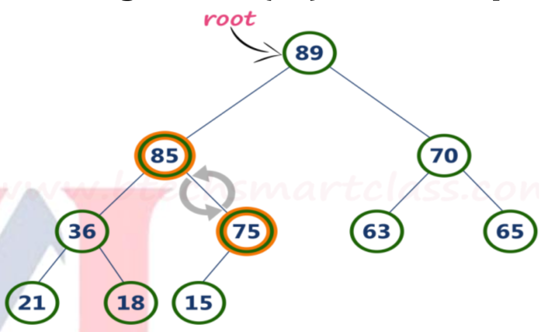
Step 5 - Now, again compare **75** with its **left child (36)**.



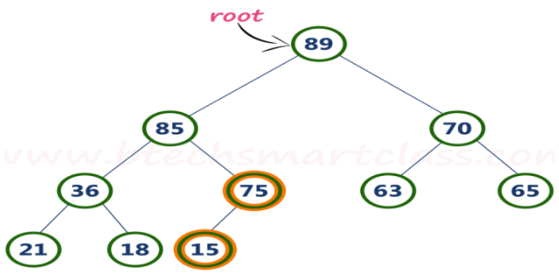
Here, node with value **75** is larger than its left child. So, we compare node **75** with its right child **85**.



Step 6 - Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...

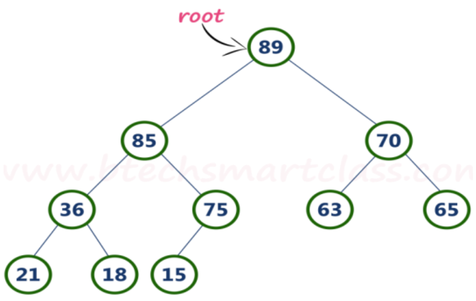


Step 7 - Now, compare node with value **75** with its left child (**15**).



Here, node with value **75** is larger than its left child (**15**) and it does not have right child. So we stop the process.

Finally, max heap after deleting root node (90) is as follows...



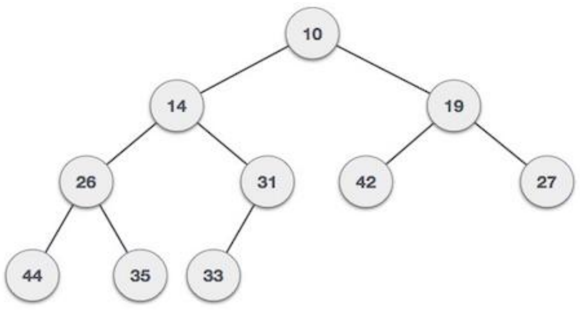
2. Min -Heap:

In a Min-Heap the key present at the root node must be less than or equal among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree. In a Min-Heap the minimum key element present at the root.

Min heap is defined as follows...

Min heap is a specialized complete binary tree in which every parent node contains lesser or equal value than its child nodes.

Example:



Insertion in Max Heap

Insertion Operation in max heap is performed as follows...

- **Step 1 -** Insert the **newNode** as **last leaf** from left to right.
- **Step 2 -** Compare **newNode value** with its **Parent node**.
- **Step 3 -** If **newNode value is smaller** than its parent, then **swap** both of them.
- **Step 4 -** Repeat step 2 and step 3 until newNode value is greater than its parent node (or) newNode reaches to root.

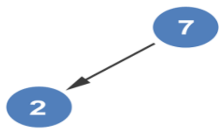
Example-1: Constructing Min-Heap from given elements
7, 2,9,4,5,3

Represent the elements in the array

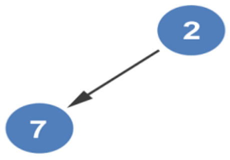
7	**2**	**9**	**4**	**5**	**3**

Step-1: We start by adding the first node, *7*.

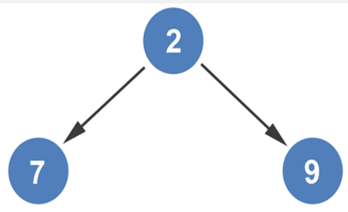
Step-2: Heaps are built from left to right. The next element that's added is *2*.



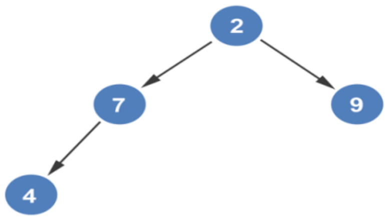
Since we're constructing a min-heap, all the children nodes must be smaller than the parent node. Number *2* is smaller than *7*, so the two nodes are swapped.



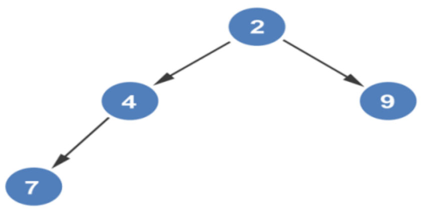
Step-3: Next, element *9* is added. Since *9* is larger than *2*, no swapping is necessary.



Step-4: Next, element 4 is added.

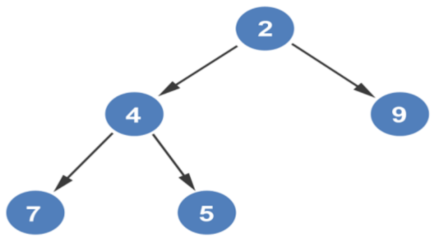


Since 4 is smaller than *7*, the two nodes are swapped.

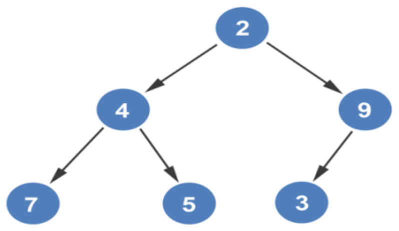


Element 4 is then checked with element *2* to make sure it's not smaller. Since it's not, the elements remain in their current positions.

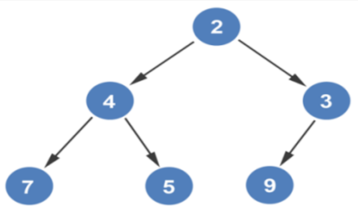
Step-5: Next, element *5* is added. Since element *4* is already smaller than element *5*, the nodes remain in their current positions.



Step-6: Finally, element *3* is added.



Since element *3* is smaller than element 9, the two nodes are swapped.



Element 3 is also compared to element 2. Since element 3 is larger than element 2, the two nodes remain in their current positions. There are no additional array elements.

This completes the construction of the min-heap tree.

2	**4**	**3**	**7**	**5**	**9**