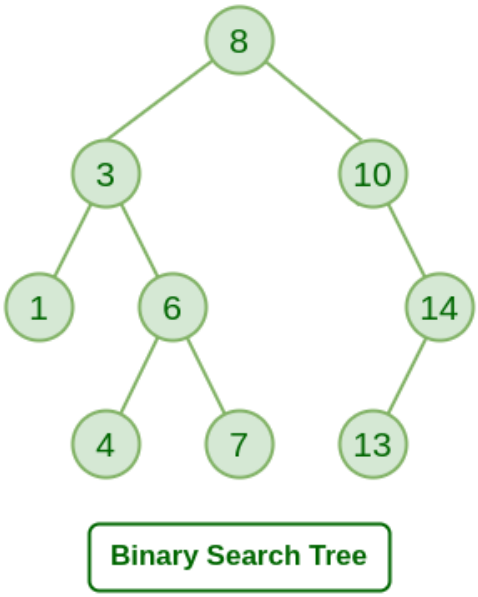
Binary Search Tree

What is Binary Search 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.



Binary Search Tree

Introduction to Binary Search Tree

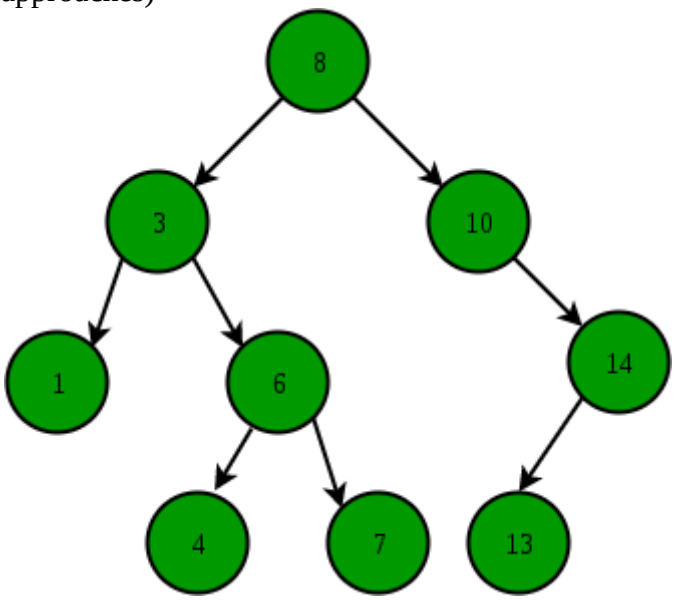
A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree.

The root of a BST is the node that has the largest value in the left subtree and the smallest value in the right subtree. Each left subtree is a BST with nodes that have smaller values than the root and each right subtree is a BST with nodes that have larger values than the root.

Binary Search Tree is a node-based binary tree data structure that 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.

- This means everything to the left of the root is less than the value of the root and everything to the right of the root is greater than the value of the root. Due to this performing, a binary search is very easy.
- The left and right subtree each must also be a binary search tree. There must be no duplicate nodes(BST may have duplicate values with different handling approaches)



Handling approach for Duplicate values in the Binary Search tree:

- You can not allow the duplicated values at all.
- We must follow a consistent process throughout i.e either store duplicate value at the left or store the duplicate value at the right of the root, but be consistent with your approach.
- We can keep the counter with the node and if we found the duplicate value, then we can increment the counter

Basic Operations

Following are the basic operations of a tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Delete** − Deletes an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

Below are the various operations that can be performed on a BST:

- Insert a node into a BST
- Inorder traversal
- Preorder traversal
- Postorder traversal

- Level order traversal
- Print nodes at given Level
- Print all leaf nodes
- Print all non leaf node
- Right view of BST
- Left view of BST
- Height of BST
- Delete a Node of BST
- Smallest Node of the BST
- Total number of nodes in a BST
- Delete a BST

Applications of BST:

- **Graph algorithms:** BSTs can be used to implement graph algorithms, such as in minimum spanning tree algorithms.
- **Priority Queues:** BSTs can be used to implement priority queues, where the element with the highest priority is at the root of the tree, and elements with lower priority are stored in the subtrees.
- **Self-balancing binary search tree:** BSTs can be used as a self-balancing data structures such as AVL tree and Red-black tree.
- **Data storage and retrieval:** BSTs can be used to store and retrieve data quickly, such as in databases, where searching for a specific record can be done in logarithmic time.

Advantages:

- **Fast search:** Searching for a specific value in a BST has an average time complexity of O(log n), where n is the number of nodes in the tree. This is much faster than searching for an element in an array or linked list, which have a time complexity of O(n) in the worst case.
- **In-order traversal:** BSTs can be traversed in-order, which visits the left subtree, the root, and the right subtree. This can be used to sort a dataset.
- **Space efficient:** BSTs are space efficient as they do not store any redundant information, unlike arrays and linked lists.
- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Disadvantages:

- **Skewed trees:** If a tree becomes skewed, the time complexity of search, insertion, and deletion operations will be O(n) instead of O(log n), which can make the tree inefficient.
- **Additional time required:** Self-balancing trees require additional time to maintain balance during insertion and deletion operations.
- **Efficiency:** BSTs are not efficient for datasets with many duplicates as they will waste space.

Example of creating a binary search tree

Suppose the data elements are - **45, 15, 79, 90, 10, 55, 12, 20, 50**

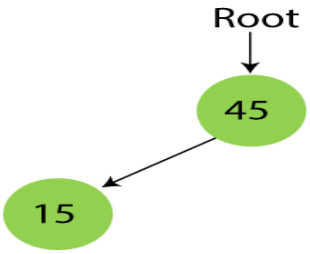
- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

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.

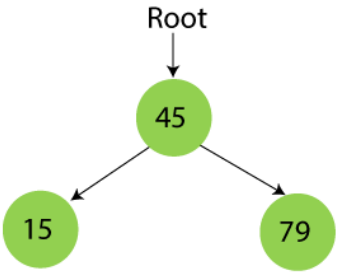
Step 2 - Insert 15.

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



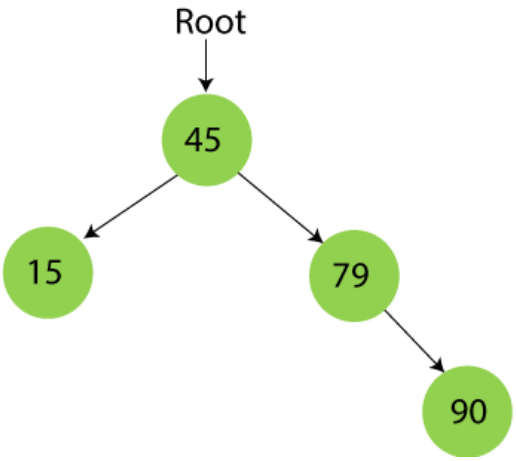
Step 3 - Insert 79.

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



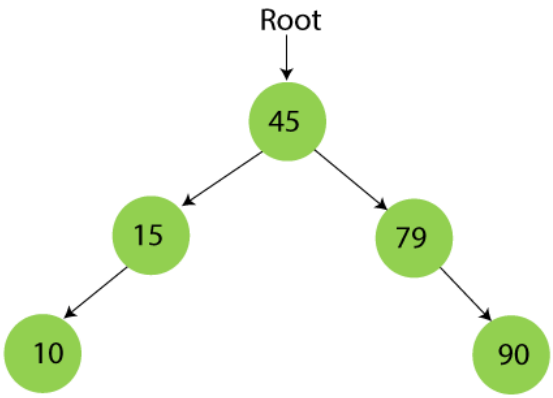
Step 4 - Insert 90.

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



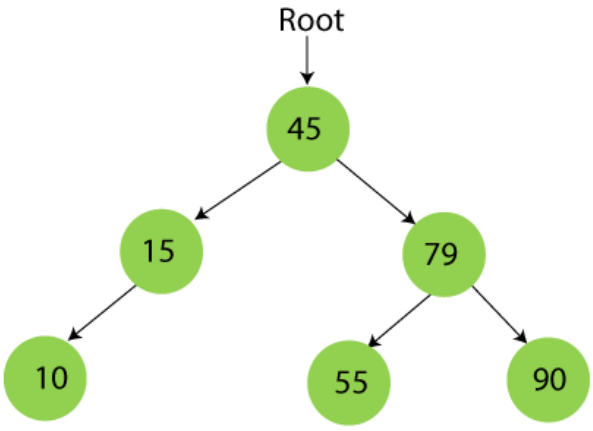
Step 5 - Insert 10.

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



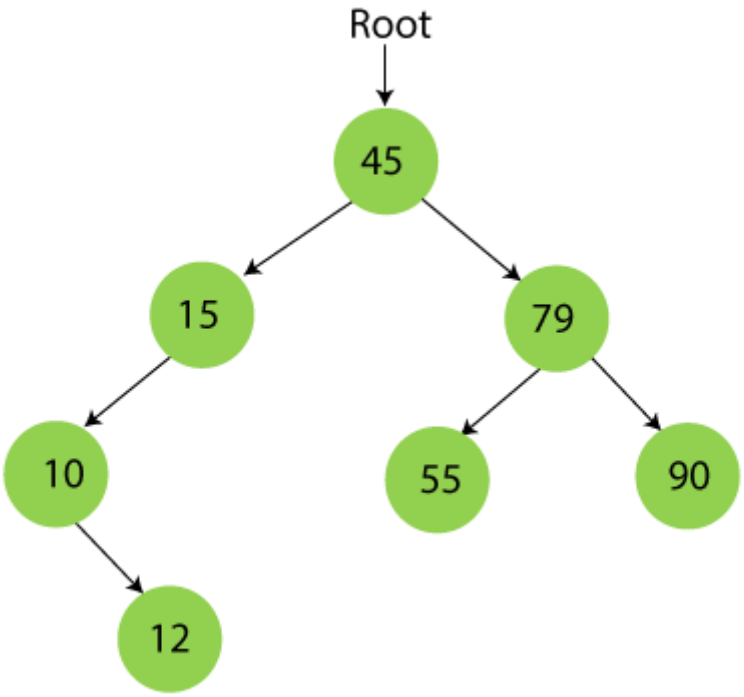
Step 6 - Insert 55.

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



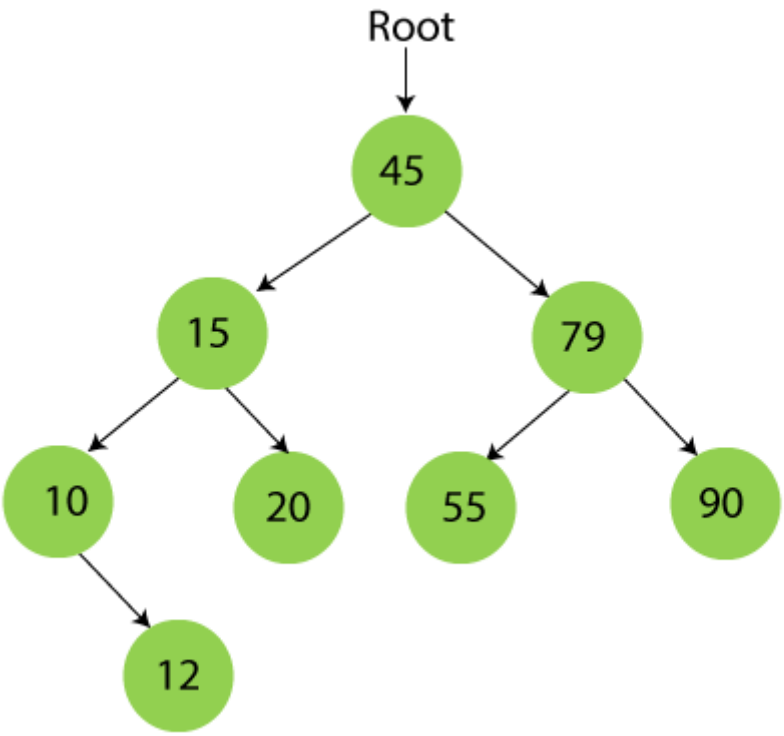
Step 7 - Insert 12.

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



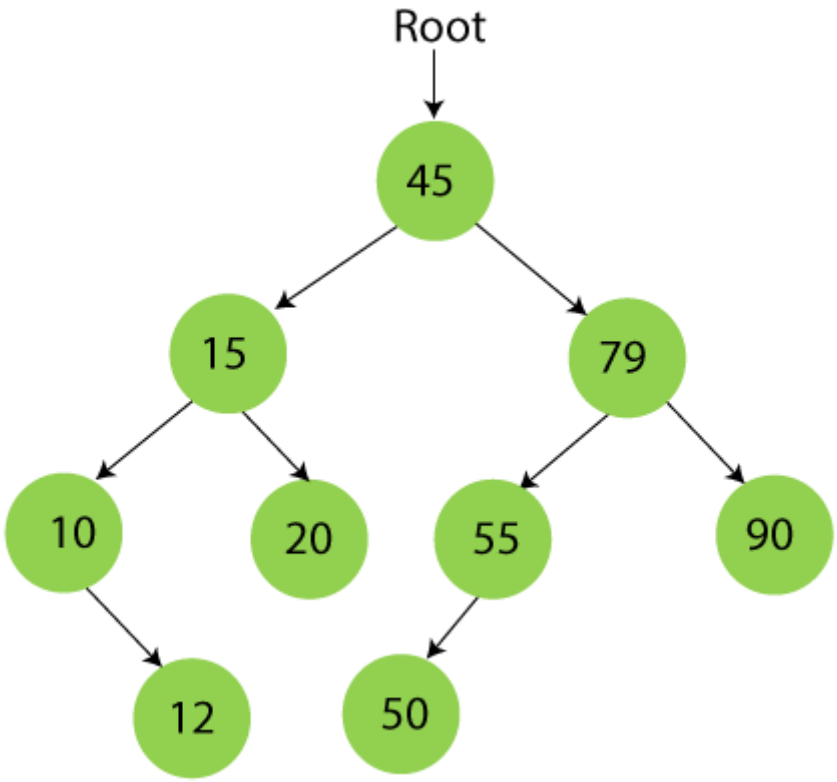
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree 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 subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

Let's understand how a search is performed on a binary search tree.

Searching in Binary search tree

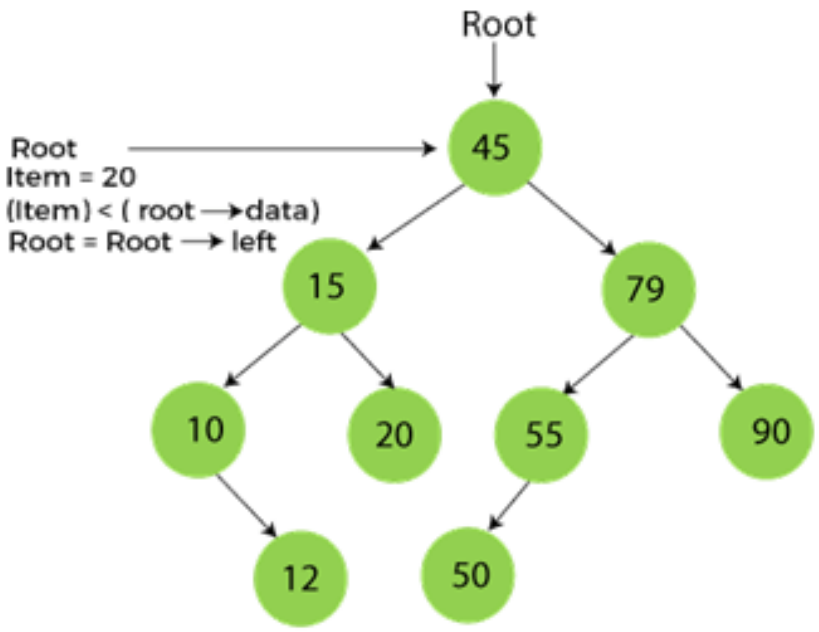
Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.

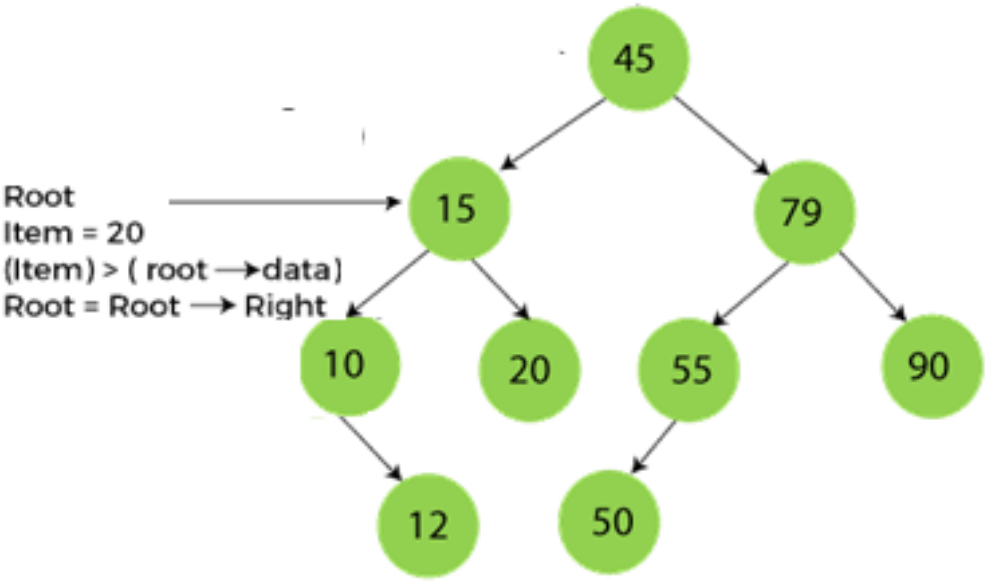
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

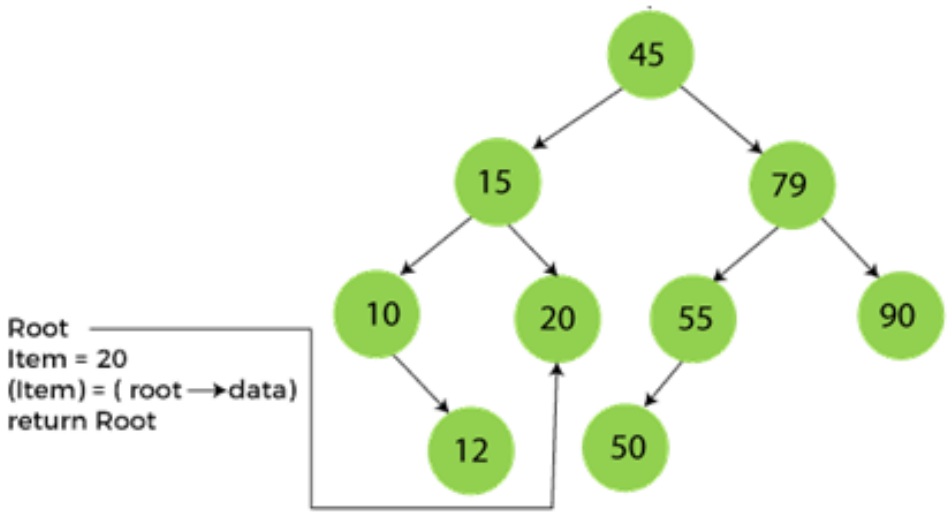
Step1:



Step2:



Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

Algorithm to search an element in Binary search tree

1. Search (root, item)
2. Step 1 - if (item = root → data) or (root = NULL)
3. return root
4. else if (item < **root** → data)
5. return Search(root → left, item)
6. else
7. return Search(root → right, item)
8. END if
9. Step 2 - END

Implementation of Binary Search Tree

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>

structbtnode
{
int value;
structbtnode *l;
```

```
structbtnode *r;
}*root = null, *temp = null, *t2, *t1;

void delete1();
void insert();
void delete();
void inorder(structbtnode *t);
void create();
void search(structbtnode *t);
void preorder(structbtnode *t);
void postorder(structbtnode *t);
void search1(structbtnode *t,int data);
int smallest(structbtnode *t);
int largest(structbtnode *t);

int flag = 1;

void main()
{
intch;

printf("\noperations ---");
printf("\n1 – insert an element into tree\n");
printf("2 – delete an element from the tree\n");
printf("3 – inorder traversal\n");
printf("4 – preorder traversal\n");
printf("5 – postorder traversal\n");
printf("6 – exit\n");
while(1)
   {
printf("\nenter your choice : ");
scanf("%d", &ch);
       switch (ch)
       {
       case 1:
insert();
          break;
       case 2:
delete();
          break;
       case 3:
inorder(root);
          break;
       case 4:
```

```
preorder(root);
            break;
        case 5:
postorder(root);
            break;
        case 6:
exit(0);
default :
printf("wrong choice, please enter correct choice  ");
            break;
        }
    }
}

/* to insert a node in the tree */
void insert()
{
create();
    if (root == null)
        root = temp;
    else
        search(root);
}

/* to create a node */
void create()
{
int data;
printf("enter data of node to be inserted : ");
scanf("%d", &data);
    temp = (structbtnode*)malloc(1*sizeof(structbtnode));
    temp->value = data;
    temp->l = temp->r = null;
}
/* function to search the appropriate position to insert the new node */
void search(structbtnode *t)
{
    if ((temp->value > t->value) && (t->r != null))      /* value more than root
node value insert at right */
        search(t->r);
    else if ((temp->value > t->value) && (t->r == null))
        t->r = temp;
    else if ((temp->value < t->value) && (t->l != null))    /* value less than root
node value insert at left */
```

```
        search(t->l);
    else if ((temp->value < t->value) && (t->l == null))
        t->l = temp;
}
/* recursive function to perform inorder traversal of tree */
void inorder(structbtnode *t)
{
    if (root == null)
    {
printf("no elements in a tree to display");
        return;
    }
    if (t->l != null)
inorder(t->l);
printf("%d -> ", t->value);
    if (t->r != null)
inorder(t->r);
}

/* to check for the deleted node */
void delete()
{
int data;

    if (root == null)
    {
printf("no elements in a tree to delete");
        return;
    }
printf("enter the data to be deleted : ");
scanf("%d", &data);
    t1 = root;
    t2 = root;
    search1(root, data);
}

/* to find the preorder traversal */
void preorder(structbtnode *t)
{
    if (root == null)
    {
printf("no elements in a tree to display");
        return;
    }
```

```
printf("%d -> ", t->value);
   if (t->l != null)
preorder(t->l);
   if (t->r != null)
preorder(t->r);
}
/* to find the postorder traversal */
void postorder(structbtnode *t)
{
   if (root == null)
   {
printf("no elements in a tree to display ");
      return;
   }
   if (t->l != null)
postorder(t->l);
   if (t->r != null)
postorder(t->r);
printf("%d -> ", t->value);
}
/* search for the appropriate position to insert the new node */
void search1(structbtnode *t, int data)
{
   if ((data>t->value))
   {
      t1 = t;
      search1(t->r, data);
   }
   else if ((data < t->value))
   {
      t1 = t;
      search1(t->l, data);
   }
   else if ((data==t->value))
   {
      delete1(t);
   }
}

/* to delete a node */
void delete1(structbtnode *t)
{
int k;
```

```
/* to delete leaf node */
if ((t->l == null) && (t->r == null))
{
   if (t1->l == t)
   {
      t1->l = null;
   }
   else
   {
      t1->r = null;
   }
   t = null;
   free(t);
   return;
}

/* to delete node having one left hand child */
else if ((t->r == null))
{
   if (t1 == t)
   {
      root = t->l;
      t1 = root;
   }
   else if (t1->l == t)
   {
      t1->l = t->l;

   }
   else
   {
      t1->r = t->l;
   }
   t = null;
   free(t);
   return;
}

/* to delete node having right hand child */
else if (t->l == null)
{
   if (t1 == t)
   {
      root = t->r;
```

```
            t1 = root;
        }
        else if (t1->r == t)
            t1->r = t->r;
        else
            t1->l = t->r;
        t == null;
        free(t);
        return;
    }

    /* to delete node having two child */
    else if ((t->l != null) && (t->r != null))
    {
        t2 = root;
        if (t->r != null)
        {
            k = smallest(t->r);
            flag = 1;
        }
        else
        {
            k =largest(t->l);
            flag = 2;
        }
        search1(root, k);
        t->value = k;
    }

}

/* to find the smallest element in the right sub tree */
int smallest(structbtnode *t)
{
    t2 = t;
    if (t->l != null)
    {
        t2 = t;
        return(smallest(t->l));
    }
    else
        return (t->value);
}
```

```
/* to find the largest element in the left sub tree */
int largest(structbtnode *t)
{
   if (t->r != null)
   {
      t2 = t;
      return(largest(t->r));
   }
   else
      return(t->value);
}
```

OUTPUT:

OPERATIONS ---

1 – Insert an element into tree
2 – Delete an element from the tree
3 – Inorder Traversal
4 – Preorder Traversal
5 – Postorder Traversal
6 – Exit

Enter your choice : 1
Enter data of node to be inserted : 40

Enter your choice : 1
Enter data of node to be inserted : 20
Enter your choice : 1
Enter data of node to be inserted : 10
Enter your choice : 1
Enter data of node to be inserted : 30
Enter your choice : 1
Enter data of node to be inserted : 60
Enter your choice : 1
Enter data of node to be inserted : 80
Enter your choice : 1
Enter data of node to be inserted : 90