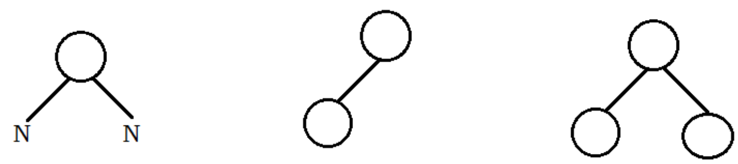
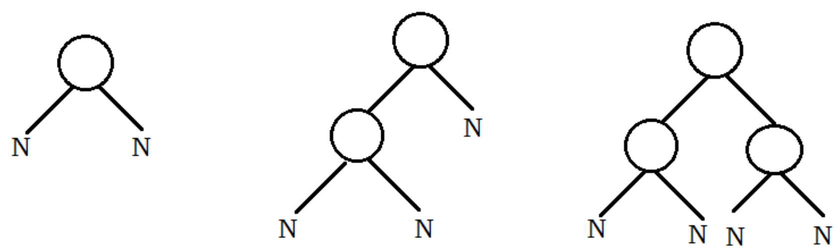
Binary Search Tree

- BST is a non-linear data structure.
- One data element is connected to multiple elements in this structure.
- We represent the data in nodes.
- Every node has 3 fields
    1. Data filed
    2. Left child
    3. Right child
- In BST, every node has at most 2 children.



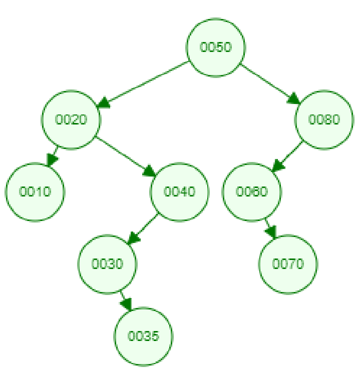
The tree with N nodes having N+1 null nodes.



- We store information into BST by comparing with Parent node.
- Least value is connected to left side of Parent node.
- Highest value is connected to right side of Parent node.
- BST not allow duplicates.
- Keys(elements) must be unique to store the data.

Inserting elements into BST:

50, 20, 80, 10, 40, 30, 35, 60, 70



Node structure: We represent the node using user data type called structure.

```
struct Node
{
        int data;
        struct Node *left;
        struct Node *right;
};
struct Node *root=NULL;
```

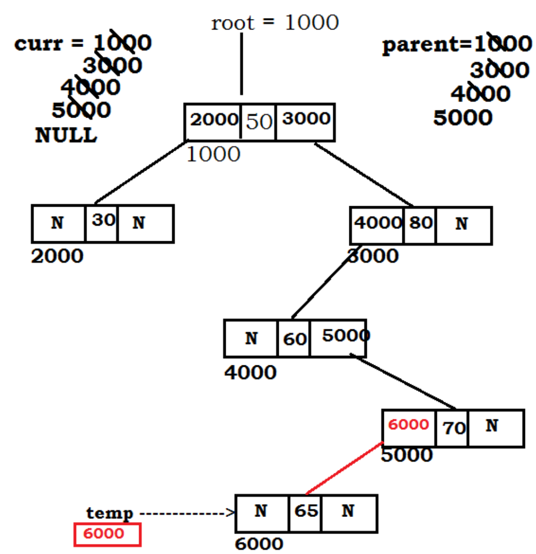
Operations:

1. Insert
2. Delete
3. Traverse

Insertion:

- Construct the Node.
- Place the Node data.
- Find the Parent node in the tree.
- Connect to Left or Right depends on the value of new node.

```
void insert(int ele)
{
    struct Node *temp;
    struct Node *p, *curr;

    temp = malloc( );
    temp->left=NULL;
    temp->right=NULL;
    if(root==NULL)
    {
        root = temp;
    }
    else
    {
        curr = root ;
        while(curr)
        {
            parent = curr;
            if(ele > curr->data){
                curr=curr->right;
            }
            else{
                curr=curr->left;
            }
        }
        if(ele < parent->data)
            parent->left = temp;
        else
            parent->right = temp;
    }
}
```

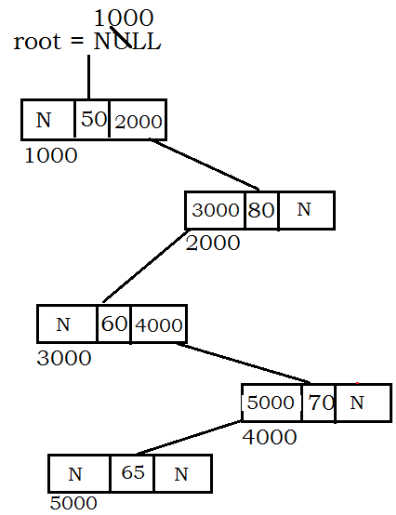


Node deletion:

- We need to find whether the element is present or not.
- If the element is present, we need to delete the node and re-arrange other nodes.
- If not present, display "Element not Found".

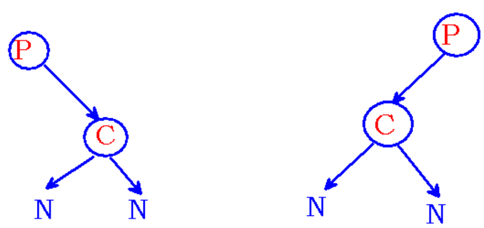
```
delete(int ele)
{
    struct Node *curr,
        *parent ;
    int found=0 ;

    curr = root ;
    while(curr)
    {
        if(curr->data == ele)
        {
            found = 1 ;
            break ;
        }
        else
        {
            parent = curr ;
            if(ele>curr->data){
                curr=curr->right ;
            }
            else{
                curr = curr->left;
            }
        }
    }
    if(!found)
    {
        printf("No such element
                to delete \n");
        return ;
    }
```

3 cases to remove the element which is present in the BST



Case 1: Deleting the element has No child

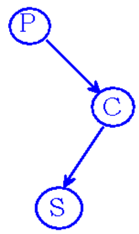
The Node has no child



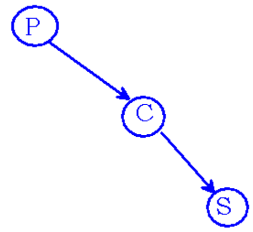

```
if(curr->left==NULL && curr->right==NULL)
{
        if(curr==parent->right)
        {
                parent->right=NULL;
        }
        else
        {
                parent->left=NULL;
        }
        free(curr);
}
```

Case 2: Deleting element that has single child

Node is connected right to Parent

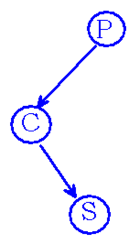


Node left is not null

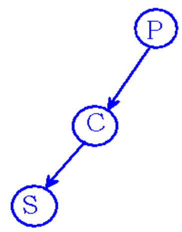


Node right is not null

Node is connected to left of Parent



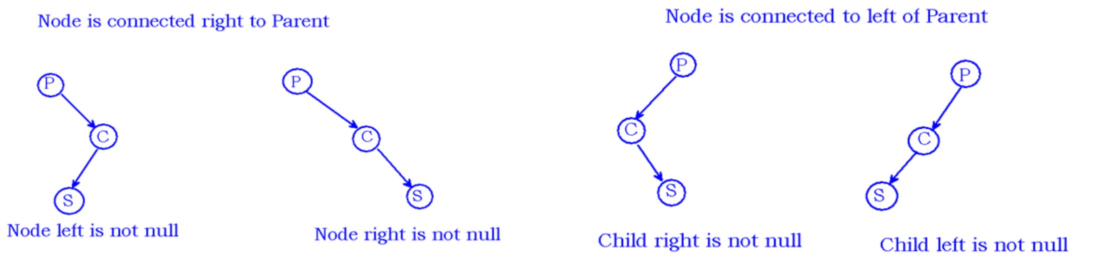
Child right is not null



Child left is not null

```
if((curr->left==NULL && curr->right!=NULL) ||
                (curr->left!=NULL && curr->right==NULL))
{
        if(curr->left==NULL && curr->right!=NULL)  // diag - 2,3
        {
                if(curr==parent->right)  // diag - 2
                {
                        parent->right = curr->right ;
                }
                else  // diag - 3
                {
                        parent->left = curr->right ;
                }
                curr->right=NULL;
                free(curr);
        }
        else  // diag - 1,4
        {
                if(curr==parent->right)  // diag - 1
                {
                        parent->right = curr->left;
                }
                else  // diag - 4
```

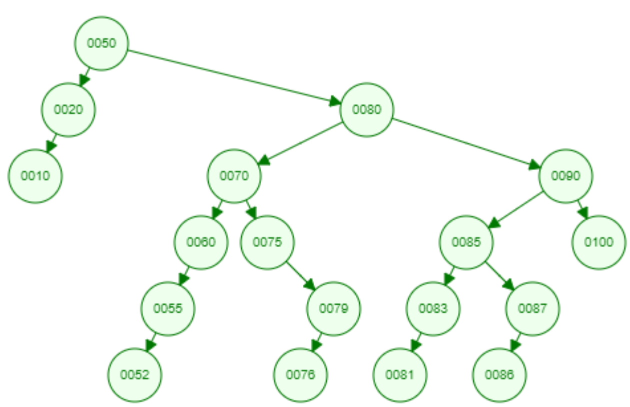
```
            {
                    parent->left = curr->left;
            }
            curr->left = NULL;
            free(curr);
        }
}
```

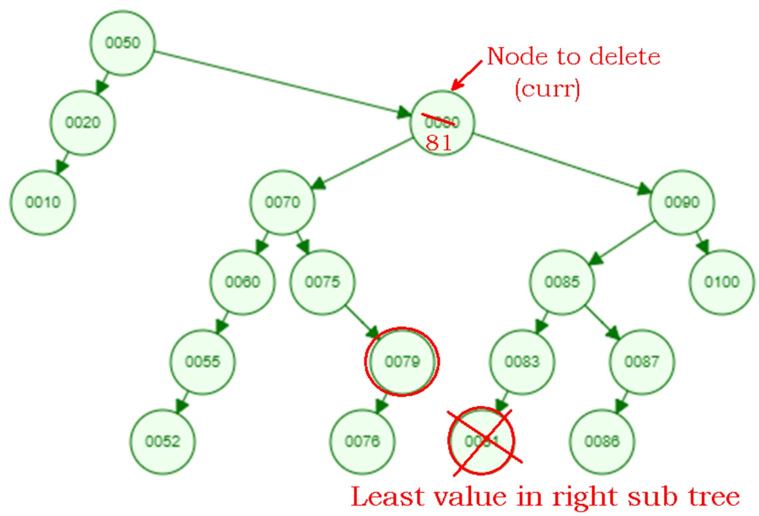



```
if((c->left == NULL && c->right != NULL)) || (c->left != NULL && c->right ==NULL))
{                 2 , 3                                                    1 , 4

    if(c->left == NULL && c->right != NULL) --> 2, 3
        if(c==p->right) --> 2
            p->right = curr->right ;  curr->right = NULL ; free(curr);
        else --> 3
            p->left = curr->right ;   curr->right = NULL ; free(curr);
    else --> 1, 4
        if(c==p->right) --> 1
            p->right = curr->left ;  curr->left = NULL ; free(curr);
        else ---> 4
            p->left = curr->left ;   curr->left = NULL ; free(curr);

}
```

Case 3: Deleting the element that has 2 children.

- Replace the current node data with
  - Least element in the right sub tree node data or
  - Highest element in the left sub tree node data.
- Removes the data swapped node.





The Node has 2 children condition:

```
        if(curr->left!=NULL && curr->right != NULL)
        {
                Logic...
        }
```

If curr-> right has no left child and right child:

```
        t1 = curr->right ;
        if(t1->left == NULL && t1->right == NULL)
        {
                curr->data = t1->data;
                curr->right = NULL;
                free(t1);
        }
```

If curr-> right has no left child but right child is present:

```
        if(t1->right!=NULL && t1->left==NULL)
        {
        curr->data=t1->data;
        curr->right=t1->right;
        t1->right=NULL;
        free(t1);
        }
```

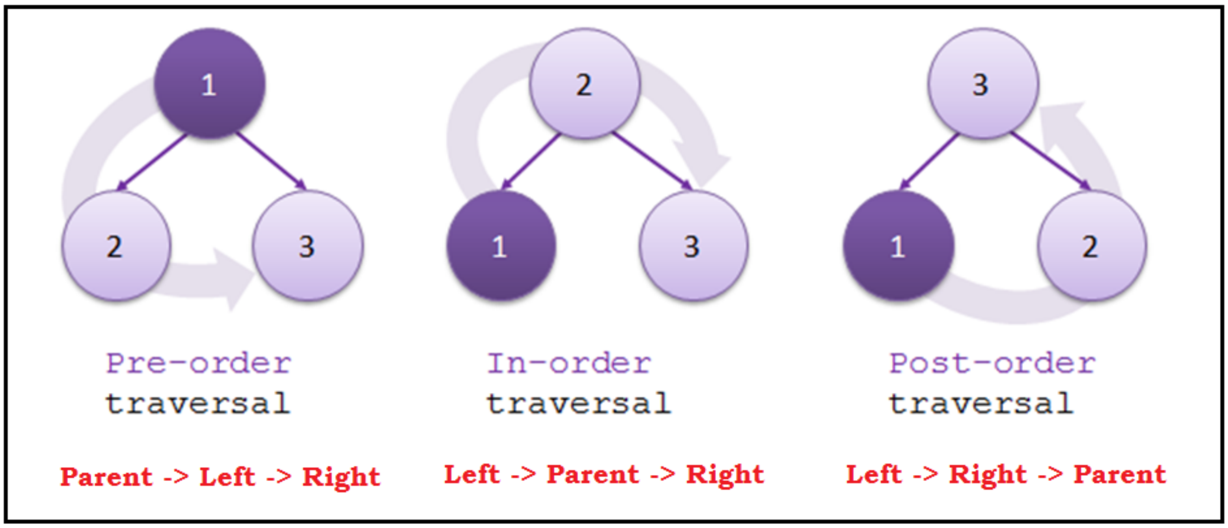
If curr->right has left child:

```
t1 = curr->right;
if(t1->left != NULL)
{
        t2 = t1->left;
}
while(t2->left)
{
```

```
        t1 = t1->left;
        t2 = t2->left;
}
curr->data = t2->data;
t1->left = t2->right;
t2->right = NULL;
free(t2);
```

Traversal:

- **We can traverse the tree in 3 ways**
  - **In order traversal**
  - **Pre order traversal**
  - **Post order traversal**



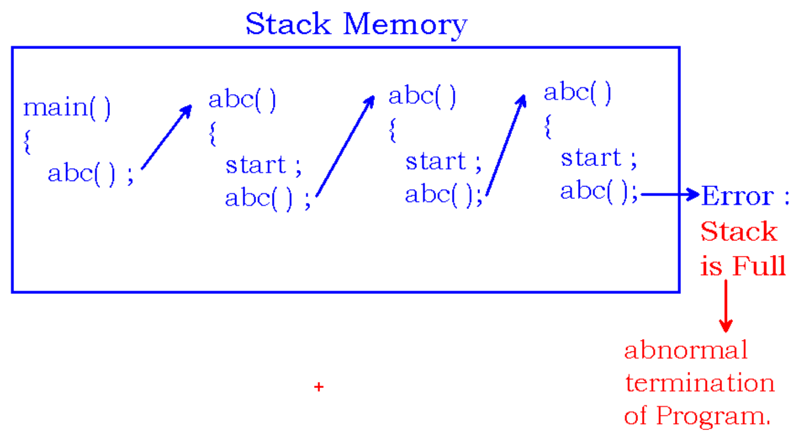
Recursion:

- Function calling itself.
- Calling the function from the definition of same function.

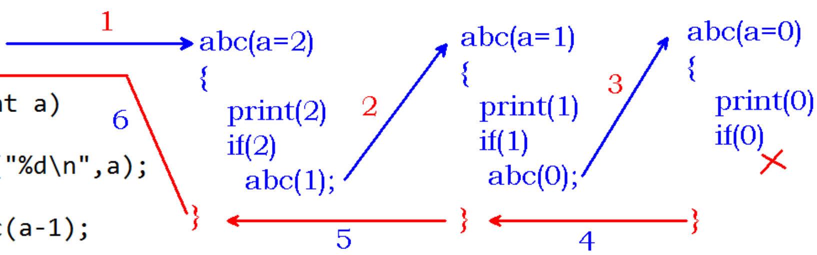
Notes:

- When we invoke a function, memory will be allocated inside the Stack.
- While the application is executing, if the memory is full, the program terminates abnormally with Runtime error: Stack is Full.

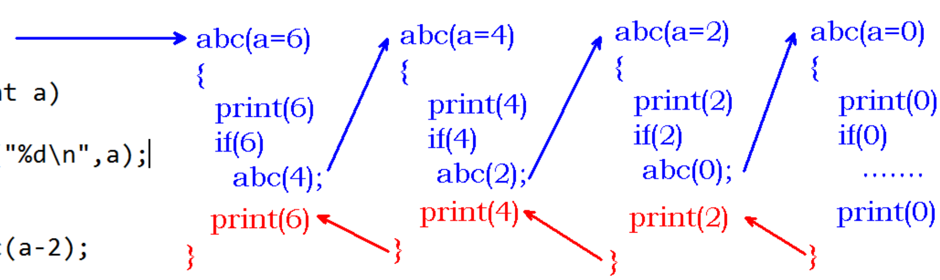
```c
#include<stdio.h>
void abc();
int main()
{
    abc();
}
void abc()
{
    printf("Start \n");
    abc();
    printf("End \n");
}
```



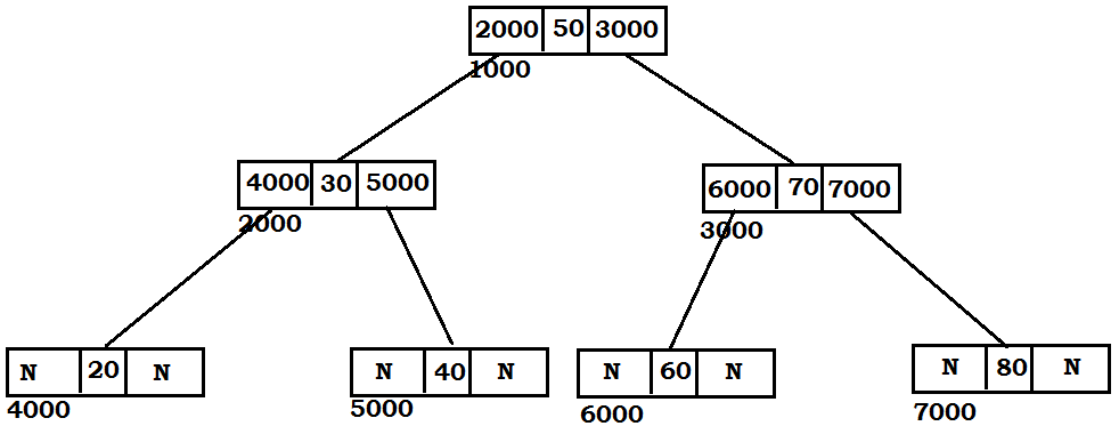

```c
#include<stdio.h>
void abc(int);
int main()
{
    abc(2);
}
void abc(int a)
{
    printf("%d\n",a);
    if(a)
        abc(a-1);
}
```



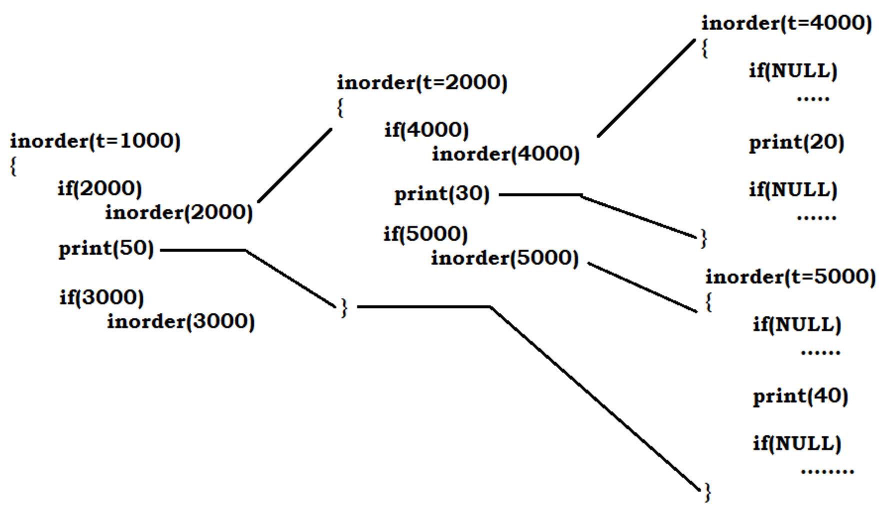

```c
#include<stdio.h>
void abc(int);
int main()
{
    abc(6);
}
void abc(int a)
{
    printf("%d\n",a);
    if(a)
    {
        abc(a-2);
    }
    printf("%d\n",a);
}
```

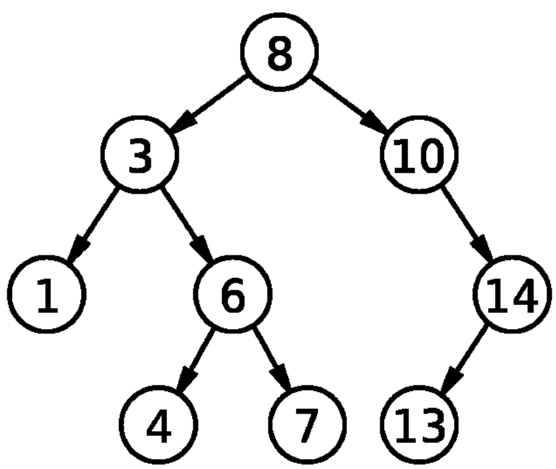


BST Traversal:




```
PrintInOrder()
{
      if(root==NULL)
            BST is empty;
      else
            inorder(root);
}

inorder(struct Node* t)
{
      if(t->left)
            inorder(t->left);

      print(t->data);

      if(t->right)
            inroder(t->righ);
}
```





InOrder:
PreOrder:
PostOrder:

Elements : 6,9,2,8,4,0,7,1,6,3
Construct BST and Traverse Post order.