Traversal functions:

Pr-order traversal function: (RT, L, R)

```
void preorder(struct node *ptr)
{
      if(ptr!=NULL)
      {
            printf("%d\t",ptr->data);    //print root
            preorder(ptr->left);  //process left subtree recursively
            preorder(ptr->right); //process right subtree recursively
      }
}
```

In-order traversal function: (L, RT, R)

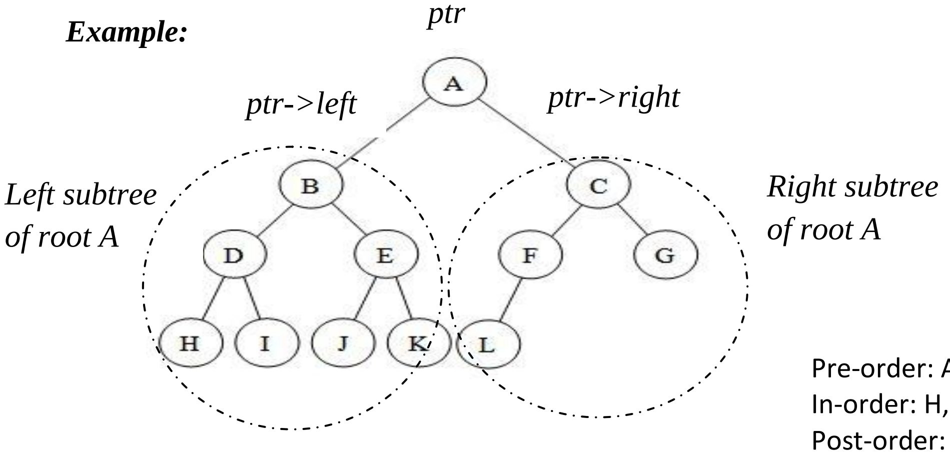
```
void inorder(struct node *ptr)
{
      if(ptr!=NULL)
      {
            inorder(ptr->left);   //process left subtree recursively
            printf("%d\t",ptr->data);    //print root
            inorder(ptr->right);  //process right subtree recursively
      }
}
```

Post-order traversal function: (L, R, RT)

```
void postorder(struct node *ptr)
{
      if(ptr!=NULL)
      {

            postorder(ptr->left); //process left subtree recursively
            postorder(ptr->right);//process right subtree recursively
            printf("%d\t",ptr->data);    //print root
      }
}
```

Example:



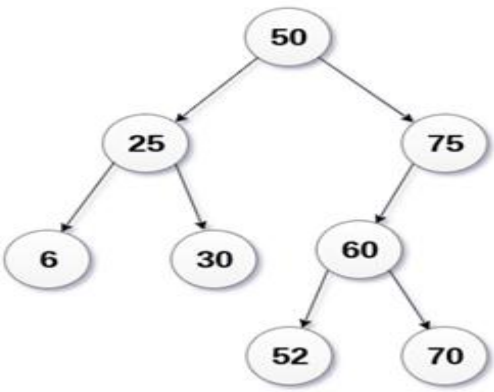
Pre-order: A, B, D, H, I, E, J, K, C, F, L, G
In-order: H, D, I, B, J, E, K, A, L, F, C, G
Post-order: H, I, D, J, K, E, B, L, F, G, C, A

Binary Search Tree (BST)

A Binary Search Tree (BST) is a binary tree in which all the nodes in the tree follow the below properties:

- All nodes of left sub-tree are less than the root node.
- All nodes of right sub-tree are greater than the root node.
- Both sub-trees of each node are also BSTs. i.e. they have the above two properties

Example:



Insertion in BST:

- A new node is always inserted at leaf node positions following the BST properties.
- In order to insert an element, the search starts from root of the tree.
- The new element is compared with the root node:
  - If it is less than the root node, then left sub-tree is recursively searched,
  - If it is greater than the root node, then right sub-tree is recursively searched.
- After reaching the appropriate leaf node position, the new node is inserted.

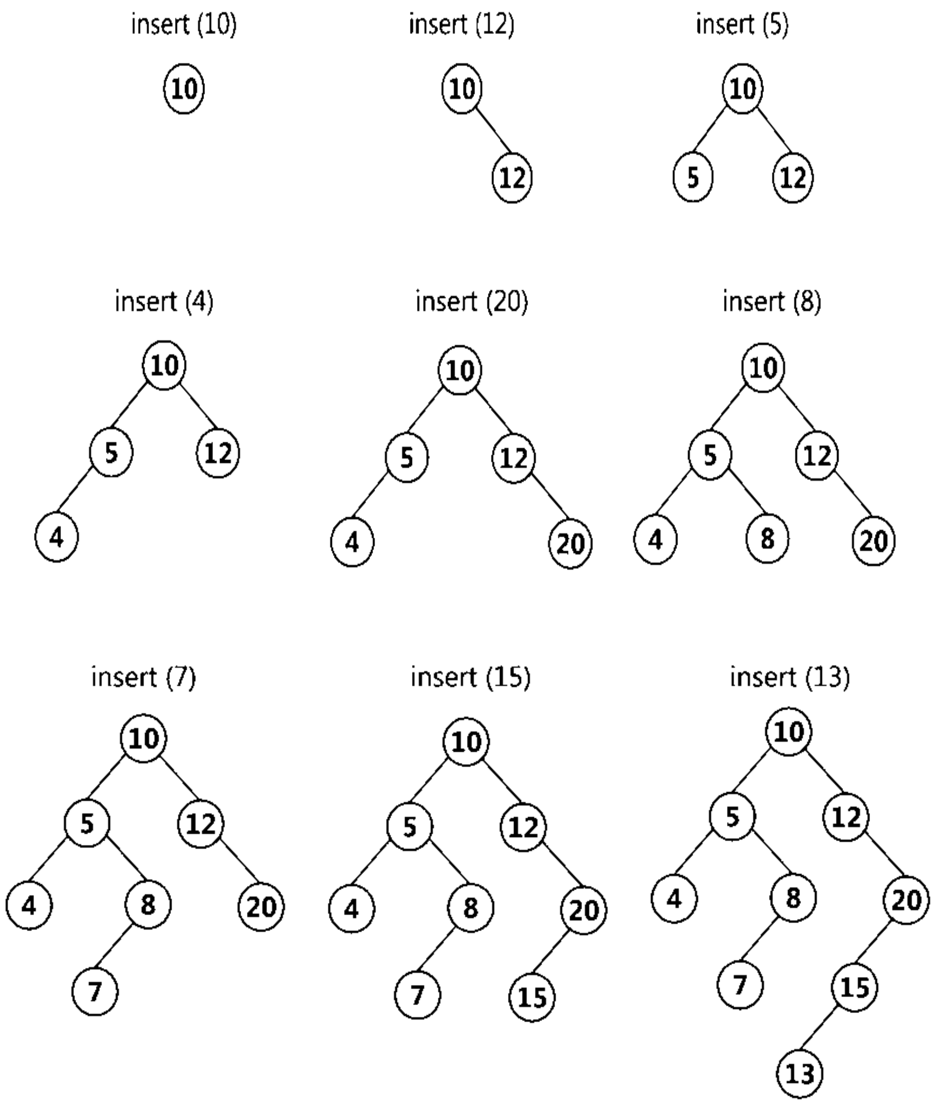
Example-1:

```
      100                              100
     /   \        Insert 40           /    \
   20     500     --------->        20      500
  /  \                             /  \
10    30                         10    30
                                         \
                                          40
```

Example-2: Construct a BST by inserting the following sequence of numbers:

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

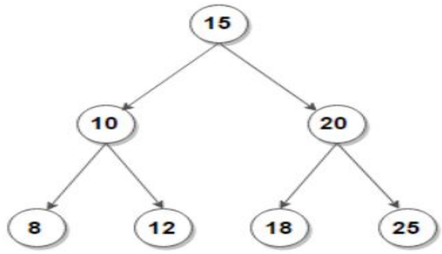
Answer:



Example-3: Construct a BST by inserting the following sequence of numbers:

15, 20, 18, 10, 12, 25, 8

Answer: Derive all steps by your own

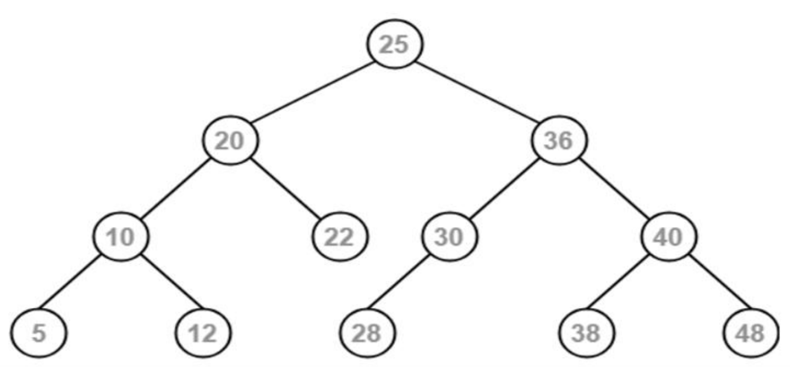


Final step: output of example-3

Example-4: Construct a BST step wise by inserting the following sequence of numbers:

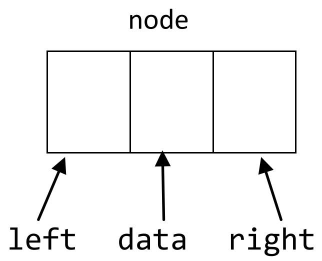
25, 36, 30, 40, 20, 28, 22, 10, 48, 12, 38, 5

Answer: Derive all steps by your own



Final step: output of example-4

Representation of a node in BST:




```
struct node
{
    int data;
    struct node *left, *right;

}*root=NULL;
```

Insert function:

```
struct node* insert(struct node *temp, int ele)
{
      if(temp==NULL)
            {
            temp =(struct node*)malloc(sizeof(struct node));
            temp ->data=ele;
            temp ->left=NULL;
            temp ->right=NULL;
            }
      else
            {
            if(ele < temp->data)
                  temp->left=insert(temp->left,ele);
            else
                  {
                  if(ele > temp->data)
                        temp->right=insert(temp->right,ele);
                  }
            }
      return temp;

}
```

<u>Program:</u>

// Menu driven program for Insert and pre-order traversal in BST:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
      int data;
      struct node *left,*right;
}*root=NULL;
struct node *insert(struct node *,int);
void preorder(struct node*);

main()
{
      int ch,x;
      while(1)
      {
            printf("\nMenu: \n1: insert\n2: pre-order traversal\n3:
exit\n");
            printf("\n Enter your choice");
            scanf("%d",&ch);
            switch(ch)
            {
                  case(1):
                        printf("enter the data to insert:");
                        scanf("%d",&x);
                        root=insert(root,x);
                        break;
                  case(2):
                        preorder(root);
                        break;
                  case(3):
                        exit(0);
                  default:
                        printf("Invalid option");
            }
      }
}

struct node *insert(struct node *temp,int ele)
      {
      .........
      .........
      //write insert function definition written above
      }


void preorder(struct node *ptr)
      {
      .........
      .........
      //write preorder traversal function definition written above
      }
```

Searching in BST:

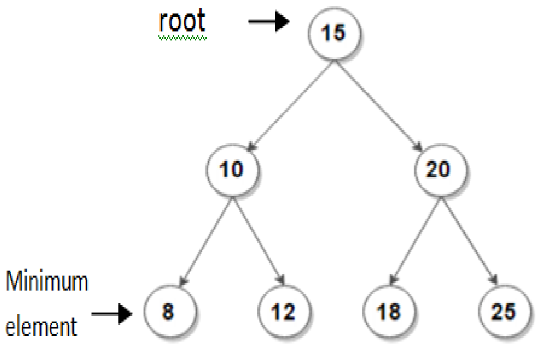
- In order to search a given value in BST, first it is compare with the root node-
  - if it matches the root value, then root is returned.
  - if it is smaller than the root value, the left sub-tree is searched recursively.
  - if it is greater than the root value, the right sub-tree is searched recursively.

```
//searching function
struct node* search(struct node* temp,int val)
{
      struct node *p;
      p=temp;
      while( p!=NULL && p->data!=val)
      {
            if(p->data > val)
                  p=p->left;
            else
            {
                  if(p->data < val)
                        p=p->right;
            }
      }
      if(p==NULL)
            printf("element not found");
      else
            return p;

}
```

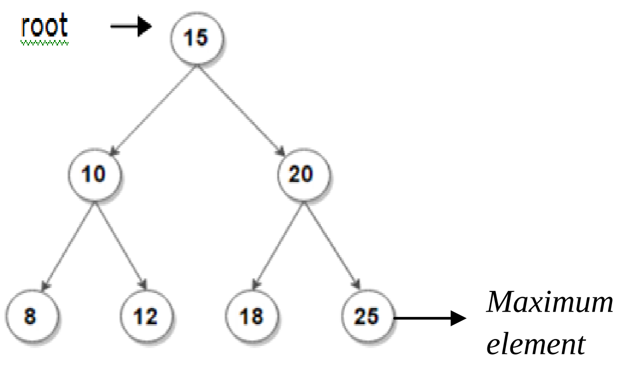
Find minimum element in BST

```
struct node* findmin(struct node* temp)
{
      if(temp==NULL)
            return NULL;
      else
      {
            if(temp->left==NULL)
                  return temp;
            else
                  return(findmin(temp->left));
      }
}
```



<u>Find maximum element in BST:</u>

```
struct node* findmax(struct node* temp)
      {
            ……… .
            ………
            //write the function by your own
      }
```



<u>Deletion in BST:</u>

- Deleting a node from BST includes following three cases:

  1) Node to be deleted is a leaf node
  2) Node to be deleted has only one child
  3) Node to be deleted has two children

Case-1: ***Node to be deleted is a leaf node:***

- Simply remove the node from the tree.

Example:

```
        50                                   50
     /      \         delete(20)          /     \
    30       70       --------->         30      70
   /  \     /  \                           \    /   \
 20    40  60    80                         40  60    80
```

Case-2: ***Node to be deleted has only one child:***

- Copy the child to the node and delete the child node

Example:

```
    50                            50
 /      \       delete(30)      /    \
30       70     --------->    40      70
  \     /  \                         /  \
   40  60   80                      60   80
```

Case-3: Node to be deleted has two children:

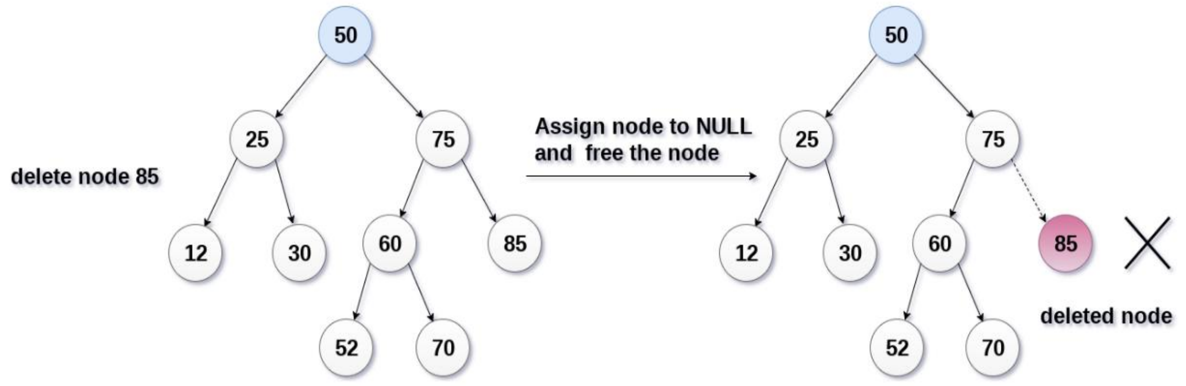
- Find in-order successor of the node.
- Copy contents of the in-order successor to the node
- Delete the in-order successor.

Example-1:

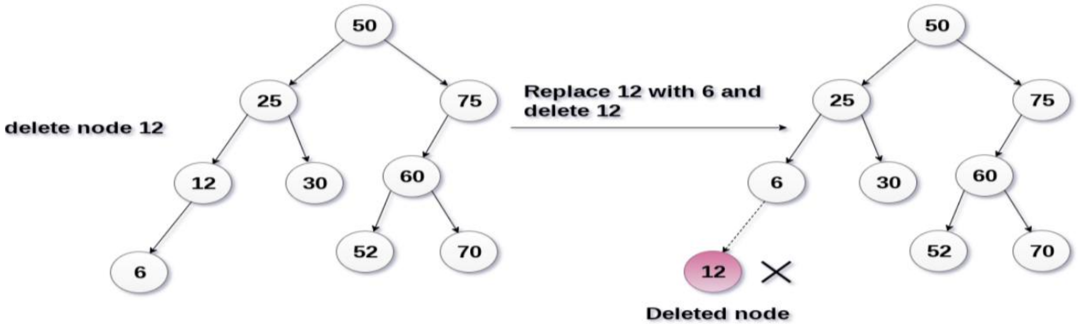
```
    50                             60
 /      \        delete(50)      /    \
40       70      --------->    40      70
        /  \                              \
       60   80                             80
```

<u>Other examples on Deletion in BST:</u>

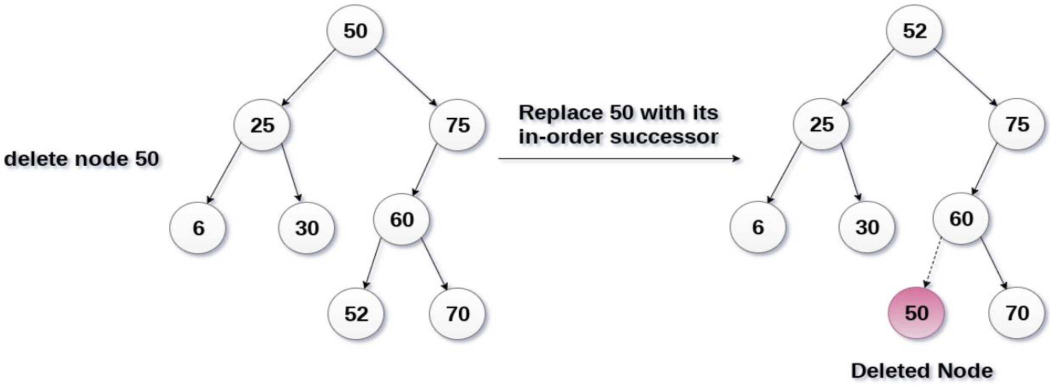
Example-1



Example-2



Example-3:



//Deletion function for BST

```
struct node* delete(struct node* temp, int val)
{

    if (temp == NULL)
      return temp;

    /* If the value to be deleted is smaller than the root's value,
     then it lies in left subtree */
    if (val < temp->data)
        temp->left = delete(temp->left, val);

    /* If the value to be deleted is greater than the root's value,
     then it lies in right subtree */
    else if (val > temp->data)
        temp->right = delete(temp->right, val);

    /* if the value is same as root's value, then this is the node
     to be deleted */
    else
    {
        // node with only one child or no child
        if (temp->left == NULL)
        {
            struct node *p = temp->right;
            free(temp);
```

```
            return p;
        }
        else if (temp->right == NULL)
        {
            struct node *p = temp->left;
            free(temp);
            return p;
        }

        /* node with two children: Get the inorder successor- smallest
         in the right subtree/*
        struct node* p = findmin(temp->right);

        // Copy the inorder successor's content to this node
        temp->data = p->data;

        // Delete the inorder successor
        temp->right = delete(temp->right, p->data);
    }
    return temp;
}
```