Primer 1.

```
#include <iostream>
#include <cstdlib>
using namespace std;

class BinarySearchTree {
    private:
        struct tree_node {
           tree_node* left;
           tree_node* right;
           int data;
        };
        tree_node* root;

    public:
        BinarySearchTree() {
           root = NULL;
        }

        bool isEmpty() const { return root==NULL; }
        void print_inorder();
        void inorder(tree_node*);
        void print_preorder();
        void preorder(tree_node*);
        void print_postorder();
        void postorder(tree_node*);
        void insert(int);
        void remove(int);
};

// Smaller elements go left
// larger elements go right
void BinarySearchTree::insert(int d) {
    tree_node* t = new tree_node;
    tree_node* parent;
    t->data = d;
    t->left = NULL;
    t->right = NULL;
    parent = NULL;

    // is this a new tree?
    if(isEmpty()) root = t;
    else {
        //Note: ALL insertions are as leaf nodes
        tree_node* curr;
        curr = root;
        // Find the Node's parent
        while(curr) {
            parent = curr;
            if(t->data > curr->data) curr = curr->right;
            else curr = curr->left;
        }

        if(t->data < parent->data)
           parent->left = t;
        else
           parent->right = t;
    }
```

```
}

void BinarySearchTree::remove(int d) {
    //Locate the element
    bool found = false;
    if(isEmpty()) {
        cout<<" This Tree is empty! "<<endl;
        return;
    }

    tree_node* curr;
    tree_node* parent;
    curr = root;

    while(curr != NULL) {
         if(curr->data == d) {
            found = true;
            break;
         } else {
             parent = curr;
             if(d>curr->data) curr = curr->right;
             else curr = curr->left;
         }
    }
    if(!found) {
        cout<<" Data not found! "<<endl;
        return;
    }

         // 3 cases :
    // 1. We're removing a leaf node
    // 2. We're removing a node with a single child
    // 3. we're removing a node with 2 children

    // Node with single child
    if((curr->left == NULL && curr->right != NULL)|| (curr->left != NULL
            && curr->right == NULL)) {
       if(curr->left == NULL && curr->right != NULL) {
           if(parent->left == curr) {
             parent->left = curr->right;
             delete curr;
           } else {
             parent->right = curr->right;
             delete curr;
           }
       }
       else { // left child present, no right child
           if(parent->left == curr) {
             parent->left = curr->left;
             delete curr;
           }
           else {
             parent->right = curr->left;
             delete curr;
           }
        }
        return;
    }
```

```
    //We're looking at a leaf node
    if(curr->left == NULL && curr->right == NULL) {
        if(parent->left == curr)
            parent->left = NULL;
        else parent->right = NULL;
            delete curr;
            return;
    }

    //Node with 2 children
    // replace node with smallest value in right subtree
    if (curr->left != NULL && curr->right != NULL) {
        tree_node* chkr;
        chkr = curr->right;
        if((chkr->left == NULL) && (chkr->right == NULL)) {
            curr = chkr;
            delete chkr;
            curr->right = NULL;
        } else { // right child has children
            //if the node's right child has a left child
            // Move all the way down left to locate smallest element
            if((curr->right)->left != NULL) {
                tree_node* lcurr;
                tree_node* lcurrp;
                lcurrp = curr->right;
                lcurr = (curr->right)->left;
                while(lcurr->left != NULL) {
                   lcurrp = lcurr;
                       lcurr = lcurr->left;
                }

               curr->data = lcurr->data;
                delete lcurr;
                lcurrp->left = NULL;
            } else {
                tree_node* tmp;
                tmp = curr->right;
                curr->data = tmp->data;
                curr->right = tmp->right;
                delete tmp;
            }
        }
        return;
    }
}

void BinarySearchTree::print_inorder() {
  inorder(root);
}

void BinarySearchTree::inorder(tree_node* p) {
    if(p != NULL) {
        if(p->left)
            inorder(p->left);

        cout<<" "<<p->data<<" ";
        if(p->right)
            inorder(p->right);
    }
```

```
    else return;
}

void BinarySearchTree::print_preorder() {
    preorder(root);
}

void BinarySearchTree::preorder(tree_node* p) {
    if(p != NULL) {
        cout<<" "<<p->data<<" ";
        if(p->left)
           preorder(p->left);
        if(p->right)
           preorder(p->right);
    }
    else return;
}

void BinarySearchTree::print_postorder() {
    postorder(root);
}

void BinarySearchTree::postorder(tree_node* p) {
    if(p != NULL) {
        if(p->left)
           postorder(p->left);
        if(p->right)
           postorder(p->right);

       cout<<" "<<p->data<<" ";
    }
    else return;
}

int main() {
    BinarySearchTree b;
    int ch,tmp,tmp1;
    while(1) {
       cout<<endl<<endl;
       cout<<" Binary Search Tree Operations "<<endl;
       cout<<" ------------------------------ "<<endl;
       cout<<" 1. Insertion/Creation "<<endl;
       cout<<" 2. In-Order Traversal "<<endl;
       cout<<" 3. Pre-Order Traversal "<<endl;
       cout<<" 4. Post-Order Traversal "<<endl;
       cout<<" 5. Removal "<<endl;
       cout<<" 6. Exit "<<endl;
       cout<<" Enter your choice : ";
       cin>>ch;
       switch(ch) {
           case 1 : cout<<" Enter Number to be inserted : ";
                    cin>>tmp;
                    b.insert(tmp);
                    break;
           case 2 : cout<<endl;
                    cout<<" In-Order Traversal "<<endl;
                    cout<<" -------------------"<<endl;
                    b.print_inorder();
                    break;
```

```
            case 3 : cout<<endl;
                     cout<<" Pre-Order Traversal "<<endl;
                     cout<<" -------------------"<<endl;
                     b.print_preorder();
                     break;
            case 4 : cout<<endl;
                     cout<<" Post-Order Traversal "<<endl;
                     cout<<" --------------------"<<endl;
                     b.print_postorder();
                     break;
            case 5 : cout<<" Enter data to be deleted : ";
                     cin>>tmp1;
                     b.remove(tmp1);
                     break;
            case 6 :
                     return 0;
        }
    }
}
```

Primer 2.

```
#include <iostream>
using namespace std;

struct tree {
    int data;
    tree *left;
    tree *right;
}*sptr, *q;

void rightcheck();
void leftcheck();
void search();

int insdata;
tree *node;

main() {
    node=new tree;
    cout<<"   PLEASE  PUT  THE root->>";
    cin>>node->data;
    sptr=node;
    q=sptr;
    node->left=NULL;
    node->right=NULL;

    cout<<"   GIVE THE child->>";
    cin>>insdata;
    search();

    while(insdata!=0) {
        if(insdata>sptr->data)
            rightcheck();
        else
            leftcheck();

        cout<<"   GIVE  THE  child->>";
        cin>>insdata;
```

```
        search();
        sptr=node;
    }
    getchar();
}

void rightcheck() {
    if(sptr->right==NULL) {
        cout<<"    "<<insdata<<" IS THE  RIGHT  child of "<<q->data<<endl;
        sptr->right=new tree;
        sptr=sptr->right;
        sptr->data=insdata;
        sptr->left=NULL;
        sptr->right=NULL;
        q=node;
    }
    else {
        if(insdata>sptr->data) {
            sptr=sptr->right;
            q=sptr;
            if(insdata>sptr->data)
                rightcheck();
            else
                leftcheck();
        }
        else {
            sptr=sptr->left;
            q=sptr;
            leftcheck();
        }
    }
}

void leftcheck() {
    if(sptr->left==NULL) {
        cout<<"   "<<insdata<<" IS THE LEFT child of "<<q->data<<endl;
        sptr->left=new tree;
        sptr=sptr->left;
        sptr->data=insdata;
        sptr->right=NULL;
        sptr->left=NULL;
        q=node;
    }else {
        if(insdata<sptr->data) {
            sptr=sptr->left;
            q=sptr;
            if(insdata>sptr->data)
                rightcheck();
            else
                leftcheck();
        }
        else {
            sptr=sptr->right;
            q=sptr;
            rightcheck();}
    }
}

void search() {
```

```
    sptr=node;
    while(sptr!=NULL) {
        if(insdata==sptr->data) {
            cout<<"This is not insertable.";
            cout<<"\nInsert child ";
            cin>>insdata;
            search();
            break;
        } else {
            if(insdata>sptr->data)
                sptr=sptr->right;
            else
                sptr=sptr->left;
        }
    }
    sptr=node;
}
```

Primer 3.

```
#include <iostream>
using namespace std;

struct node {
    int data;
    node *left;
    node *right;
};

node *tree=NULL;
node *insert(node *tree,int ele);

void preorder(node *tree);
void inorder(node *tree);
void postorder(node *tree);
int count=1;

main() {
    int ch,ele;
    do {
        cout<<"\n\t1----INSERT A NODE IN A BINARY TREE.";
        cout<<"\n\t2----PRE-ORDER TRAVERSAL.";
        cout<<"\n\t3----IN-ORDER TRAVERSAL.";
        cout<<"\n\t4----POST-ORDER TRAVERSAL.";
        cout<<"\n\t5----EXIT.";
        cout<<"\n\tENTER CHOICE::";

        cin>>ch;

        switch(ch) {
            case 1:
                cout<<"\n\tENTER THE ELEMENT::";
                cin>>ele;
                tree=insert(tree,ele);
                break;

            case 2:
                cout<<"\n\t****PRE-ORDER TRAVERSAL OF A TREE****";
                preorder(tree);
```

```
                break;

            case 3:
                cout<<"\n\t****IN-ORDER TRAVERSAL OF A TREE****";
                inorder(tree);
                break;

            case 4:
                cout<<"\n\t****POST-ORDER TRAVERSAL OF A TREE****";
                postorder(tree);
                break;

            case 5:
                exit(0);
        }
    }while(ch!=5);
}

node *insert(node *tree,int ele) {
    if(tree==NULL) {
        tree=new node;
        tree->left=tree->right=NULL;
        tree->data=ele;
        count++;
    }
    else if(count%2==0)
        tree->left=insert(tree->left,ele);
    else
        tree->right=insert(tree->right,ele);

    return(tree);
}

void preorder(node *tree) {
    if(tree!=NULL) {
        cout<<tree->data<<" ";
        preorder(tree->left);
        preorder(tree->right);
        getchar();
    }
}

void inorder(node *tree) {
    if(tree!=NULL) {
        inorder(tree->left);
        cout<<tree->data<<" ";
        inorder(tree->right);
        getchar();
    }
}

void postorder(node *tree) {
    if(tree!=NULL) {
        postorder(tree->left);
        postorder(tree->right);
        cout<<tree->data<<" ";
        getchar();
    }
}
```