Aim:-

Construct a threaded binary search tree by inserting values in the given order and traverse it in inorder traversal using threads.

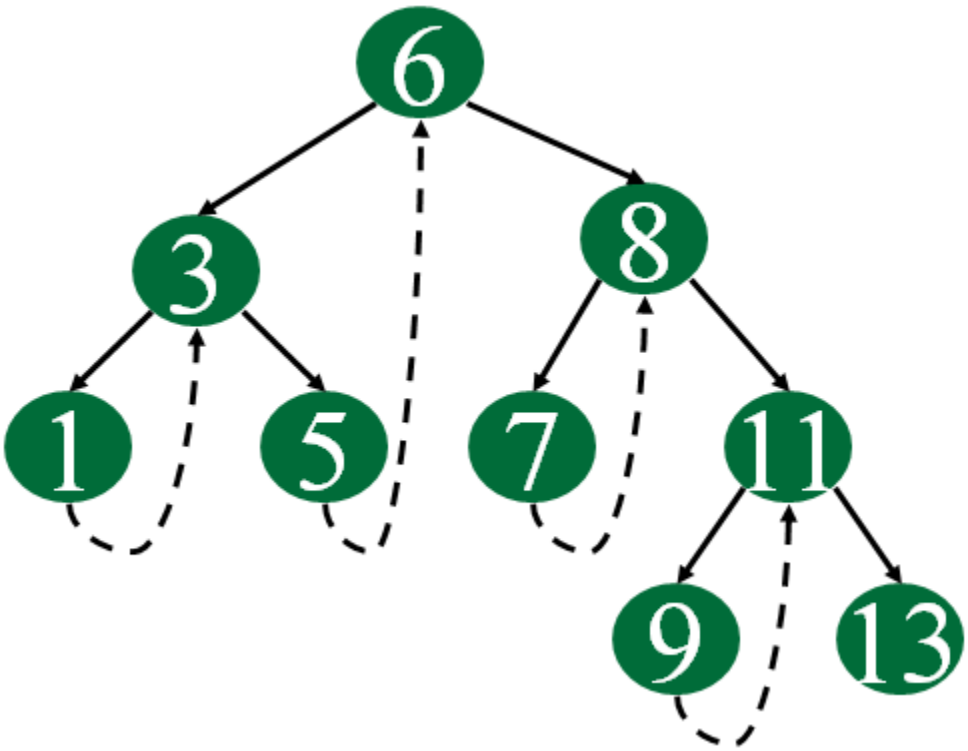
Objective:-

To Construct a threaded binary search tree by inserting values in the given order and traverse it in inorder traversal using threads.

Theory:-

Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack.The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).
There are two types of threaded binary trees.
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)
Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.
The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads



Traversal algorithm

Let's start with the main reason we're using a threaded binary tree. It is now very easy to find the in-order predecessor and the in-order successor of any node in the tree. If the node has no `left`/`right` child, we can simply return the node's `leftThread`/`rightThread`. Otherwise, it is trivial to move down the tree and find the correct node.

```
  func predecessor() -> ThreadedBinaryTree<T>? {
    if let left = left {
      return left.maximum()
    } else {
      return leftThread
    }
  }

  func successor() -> ThreadedBinaryTree<T>? {
    if let right = right {
      return right.minimum()
    } else {
      return rightThread
    }
  }
```

Note: `maximum()` and `minimum()` are methods of `ThreadedBinaryTree` which return the largest/smallest node in a given sub-tree. See the implementation for more detail.

Because these are `ThreadedBinaryTree` methods, we can call `node.predecessor()` or `node.successor()` to obtain the predecessor or successor of any `node`, provided that `node` is a `ThreadedBinaryTree` object. Because predecessors and/or successors are tracked, an in-order traversal of a threaded binary tree is much more efficient than the recursive algorithm outlined above. We use these predecessor/successor attributes to great effect in this new algorithm for both forward and backward traversals:

```
    public func traverseInOrderForward(_ visit: (T) -> Void) {
        var n: ThreadedBinaryTree
        n = minimum()
        while true {
            visit(n.value)
            if let successor = n.successor() {
                n = successor
            } else {
                break
            }
        }
    }

    public func traverseInOrderBackward(_ visit: (T) -> Void) {
        var n: ThreadedBinaryTree
        n = maximum()
```

```
        while true {
            visit(n.value)
            if let predecessor = n.predecessor() {
                n = predecessor
            } else {
                break
            }
        }
    }
```

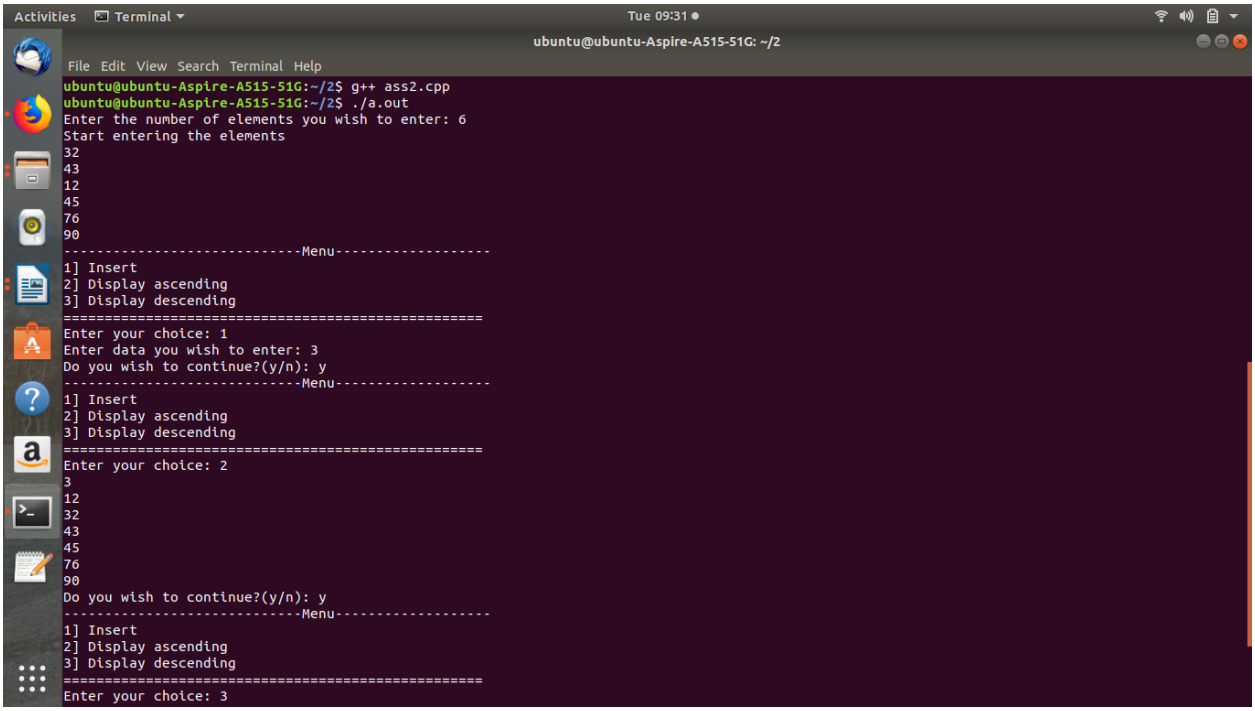
Program Code:-

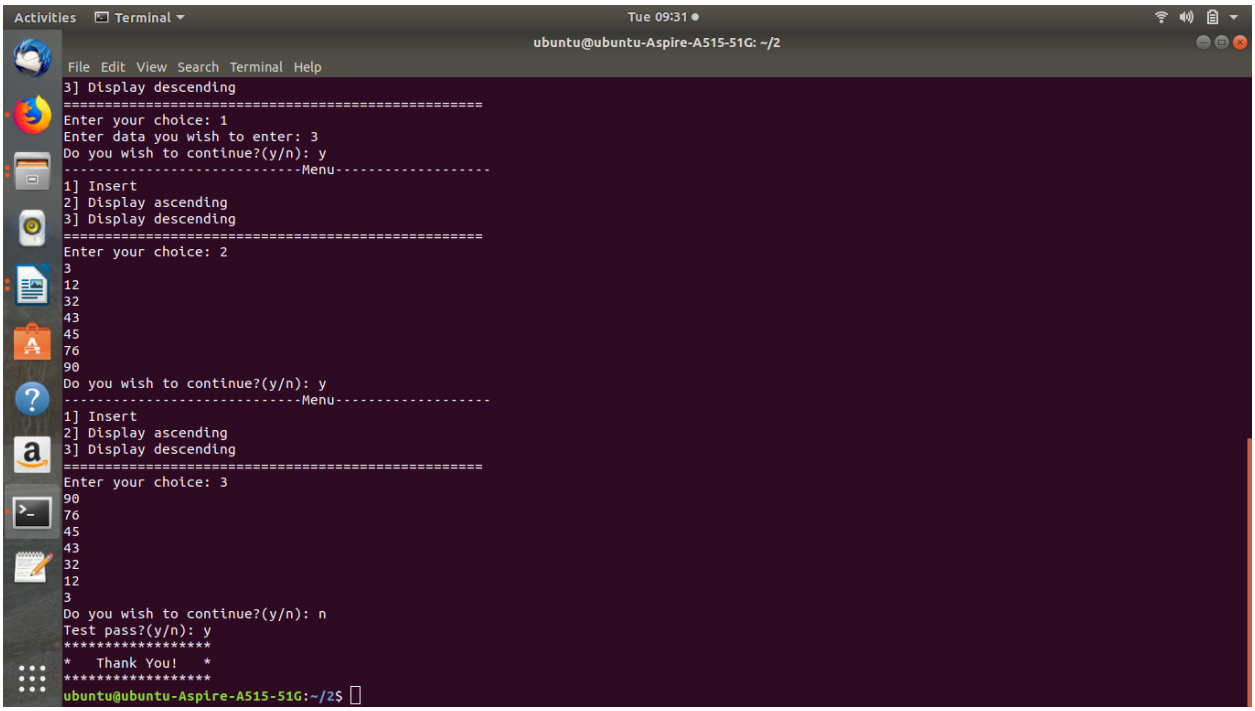
```
        #include <iostream>
using namespace std;
class tnode
{
    int data;
    int l,r;
    tnode *lt,*rt;
public:
    tnode *create(int item)
    {
        tnode *nn=new tnode;
        nn->data=item;
        nn->lt=nn->rt=NULL;
        nn->l=nn->r=1;
        return nn;
    }
    tnode *insert_r(tnode *r,int item)
    {
        tnode *rp=r;
        tnode *p=NULL;
        while(rp!=NULL)
        {
            p=rp;
            if(item<rp->data)
            {
                if(rp->l==0)
                    rp=rp->lt;
                else
                    break;
            }
            else if(item>rp->data)
            {
                if(rp->r==0)
                    rp=rp->rt;
                else
                    break;
            }
        }
        tnode *nn;
        nn=nn->create(item);
        if(p==NULL)
        {
            r=nn;
        }
        else if(item<p->data)
        {
            nn->lt=p->lt;
            nn->rt=p;
            p->lt=nn;
            p->l=0;
        }
        else if(item>p->data)
        {
            nn->rt=p->rt;
            nn->lt=p;
            p->rt=nn;
            p->r=0;
        }
        return r;
    }
    void inorder(tnode *r)
```

```
    {
        tnode *temp=r;
        while(temp->l==0)
        {
            temp=temp->lt;
        }
        while(temp!=NULL)
        {
            cout<<temp->data<<endl;
            if(temp->r==1)
                temp=temp->rt;
            else
            {
                temp=temp->rt;
                while(temp->l==0)
                {
                    temp=temp->lt;
                }
            }
        }
    }
    void reverse_inorder(tnode *r)
    {
        tnode *temp=r;
        while(temp->r==0)
        {
            temp=temp->rt;
        }
        while(temp!=NULL)
        {
            cout<<temp->data<<endl;
            if(temp->l==1)
                temp=temp->lt;
            else
            {
                temp=temp->lt;
                while(temp->r==0)
                {
                    temp=temp->rt;
                }
            }
        }
    }
};
tnode *root=NULL;
int main()
{
    char r;
    do
    {
        int flag=0;
        root=NULL;
        char op;
        do
        {
            if(flag==0)
            {
                int n;
                cout<<"Enter the number of elements you wish to enter: ";
                cin>>n;
                int a[n];
                cout<<"Start entering the elements\n";
                for(int i =0;i<n;i++)
                {
                    cin>>a[i];
                    root=root->insert_r(root,a[i]);
                }
                flag=1;
            }
            int c;
            cout<<"----------------------------Menu------------------\
n";
            cout<<"1] Insert\n2] Display ascending\n3] Display
descending";
            cout<<"\n=================================================\
n";
            cout<<"Enter your choice: ";
```

```
            cin>>c;
            switch(c)
            {
                case 1: {
                            int item;
                            cout<<"Enter data you wish to enter: ";
                            cin>>item;
                            root=root->insert_r(root, item);
                        }
                        break;
                case 2: {
                            root->inorder(root);
                        }
                        break;
                case 3: {
                            root->reverse_inorder(root);
                        }
                        break;
                default:cout<<"Invalid\n";
            }
            cout<<"Do you wish to continue?(y/n): ";
            cin>>op;
            }while(op=='y' || op=='Y');
            cout<<"Test pass?(y/n): ";
            cin>>r;
        }while(r=='n' || r=='N');
        cout<<"******************\n";
        cout<<"*   Thank You!   *\n";
        cout<<"******************\n";
        return 0;
}
```

Output Screenshot:-





Conclusion:-

We executed the Threaded Binary Tree Using The Tree Data structure.