ASSIGNMENT 2

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

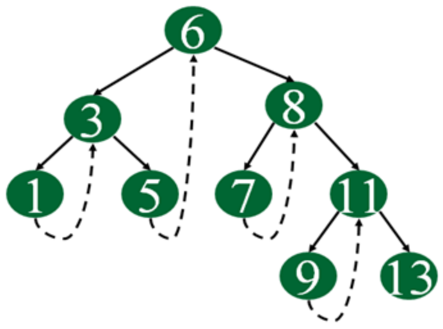
2. **Objective** – To create an threaded binary tree by using concepts of basic binary tree and additional concepts related to it and insert an element and and traversing the tree.

3. **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).

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

The threads are also useful for fast accessing ancestors of a node.



Insertion in Binary threaded tree is similar to insertion in binary tree but we will have to adjust the threads after insertion of each element.

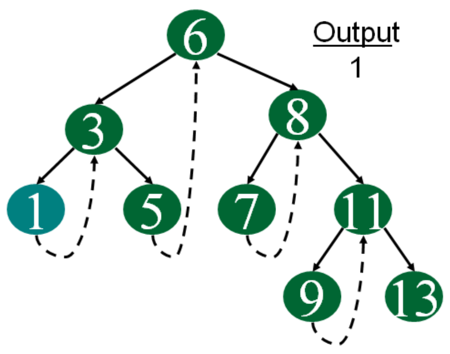
A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node**.**

- We have the pointers reference the next node in an inorder traversal; called threads
- We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer

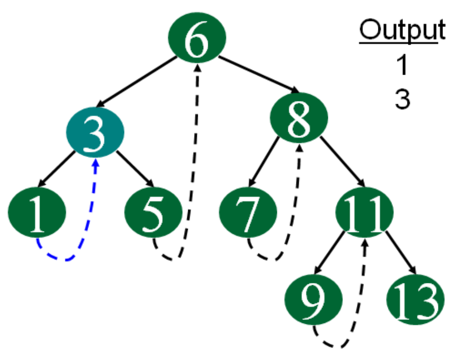
Why do we need Threaded Binary Tree?

- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals.
- Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal

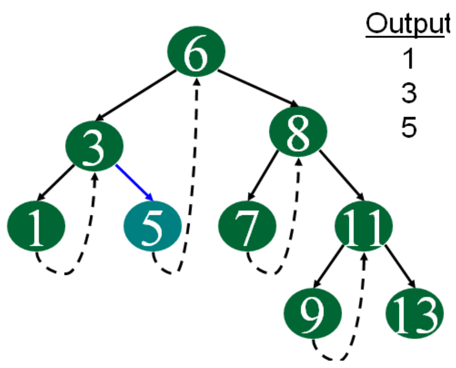
Following diagram demonstrates inorder order traversal using threads.



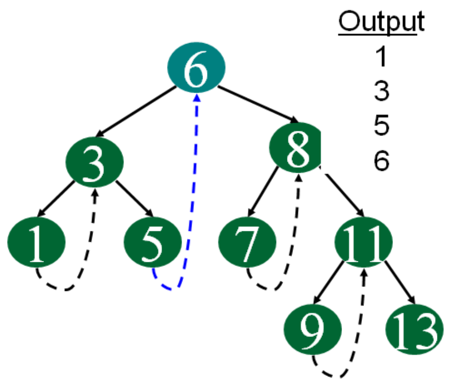
Start at leftmost node, print it



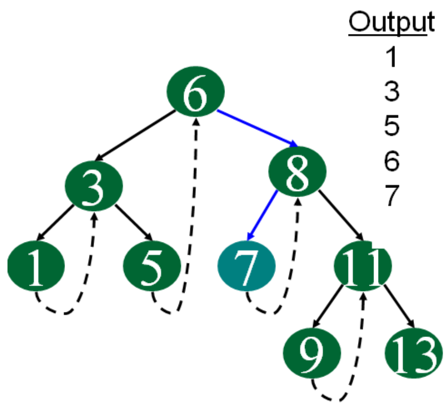
Follow thread to right, print node



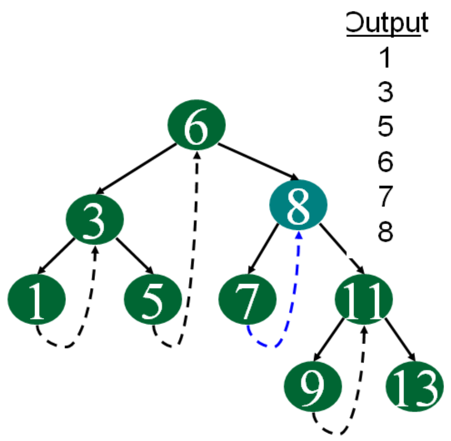
Follow link to right, go to leftmost node and print



Follow thread to right, print node



Follow link to right, go to leftmost node and print



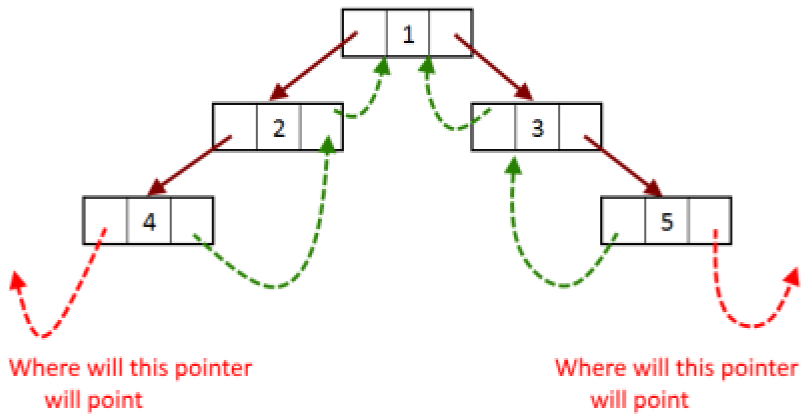
Follow thread to right, print node

continue same way for remaining node.....

4. **Algorithm** –

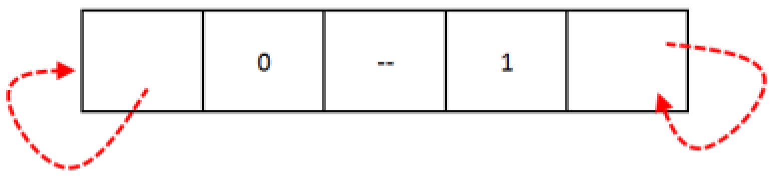
1.

```
class Node {
   int data;
   int leftBit;
   int rightBit;
   Node left;
   Node right;

   public Node(int data) {
      this.data = data;
   }
}
```


1
2
3
4
5
Where will this pointer will point
Where will this pointer will point

2. Need of a Dummy Node:
As we saw that references left most reference and right most reference pointers has nowhere to point to so we need a dummy node and this node will always present even when tree is empty.

In this dummy node we will put ***rightBit = 1*** and its right child will point to it self and ***leftBit = 0***, so we will construct the threaded tree as the left child of dummy node.
Let's see how the dummy node will look like:



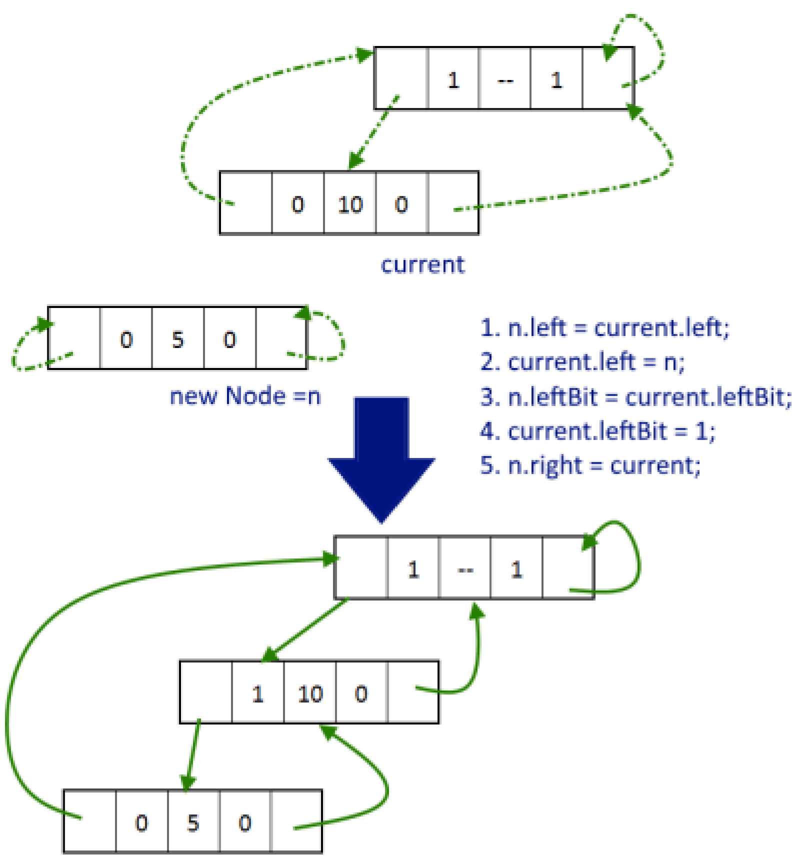
Dummy Node

3.Insert():

The insert operation will be quite similar to Insert operation in Binary search tree with few modifications.

1. To insert a node our first task is to find the place to insert the node.
2. First check if tree is empty, means tree has just dummy node then then insert the new node into left subtree of the dummy node.
3. If tree is not empty then find the place to insert the node, just like in normal BST.
4. If new node is smaller than or equal to current node then check if ***leftBit =0***, if yes then we have found the place to insert the node, it will be in the left of the subtree and if ***leftBit=1*** then go left.
5. If new node is greater than current node then check if ***rightBit =0***, if yes then we have found the place to insert the node, it will be in the right of the subtree and if ***rightBit=1*** then go right.
6. Repeat step 4 and 5 till the place to be inserted is not found.
7. Once decided where the node will be inserted, next task would be to insert the node. first we will see how the node will be inserted as left child.

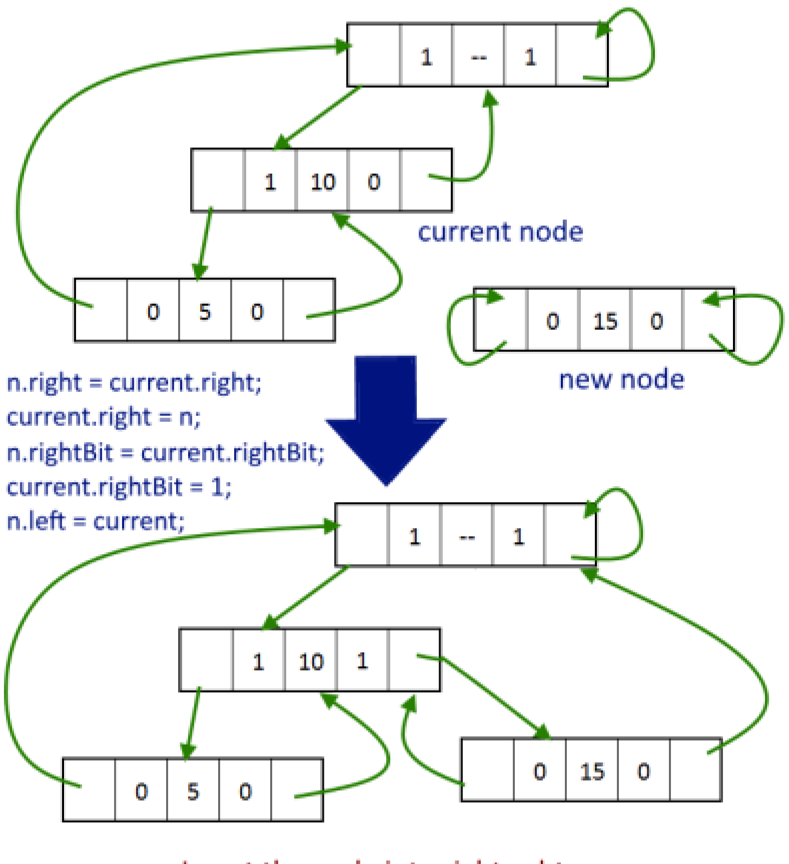
```
n.left = current.left;

current.left = n;

n.leftBit = current.leftBit;

current.leftBit = 1;

n.right = current;
```



Insert the node in the left sub tree

8. To insert the node as right child.

```
n.right = current.right;

current.right = n;

n.rightBit = current.rightBit;

current.rightBit = 1;

n.left = current;
```



Insert the node into right subtree

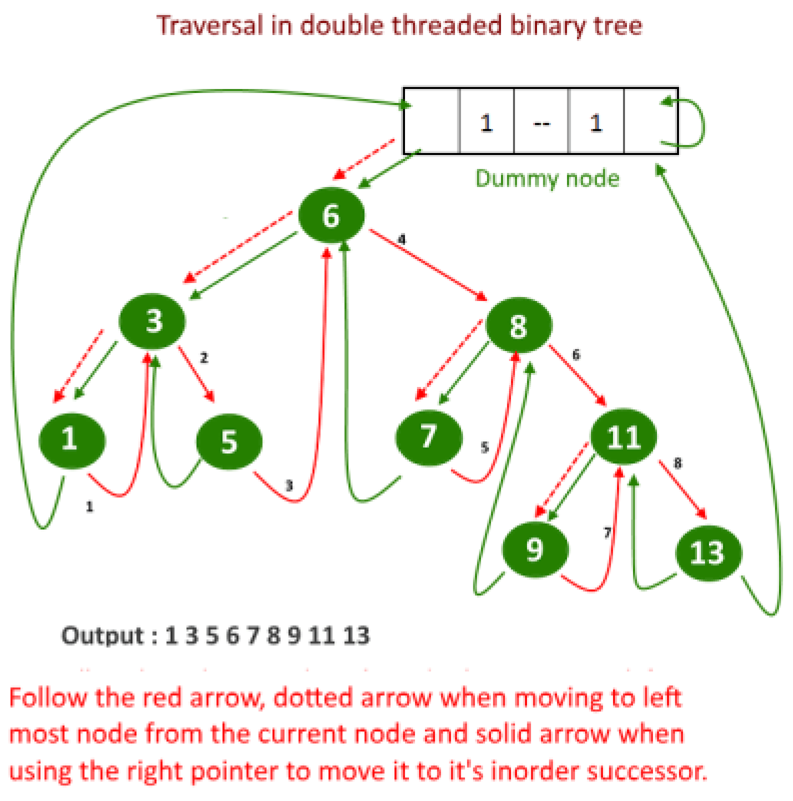
4.Traverse():

Now we will see how to traverse in the double threaded binary tree, we do not need a recursion to do that which means it won't require stack, it will be done n one single traversal in O(n).

Starting from left most node in the tree, keep traversing the inorder successor and print it.(click here to read more about inorder successor in a tree).

See the image below for more understanding.

```
void inOrder(struct Node *root)
{
    struct Node *cur = leftmost(root);
    while (cur != NULL)
    {
        printf("%d ", cur->data);

        // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
            cur = cur->right;
            else // Else go to the leftmost child in right
subtree
                cur = leftmost(cur->right);
    }
}
```


Traversal in double threaded binary tree
1
--
1
Dummy node
6
3
8
1
5
7
11
9
13
1
2
3
4
5
6
7
8
Output : 1 3 5 6 7 8 9 11 13
Follow the red arrow, dotted arrow when moving to left most node from the current node and solid arrow when using the right pointer to move it to it's inorder successor.

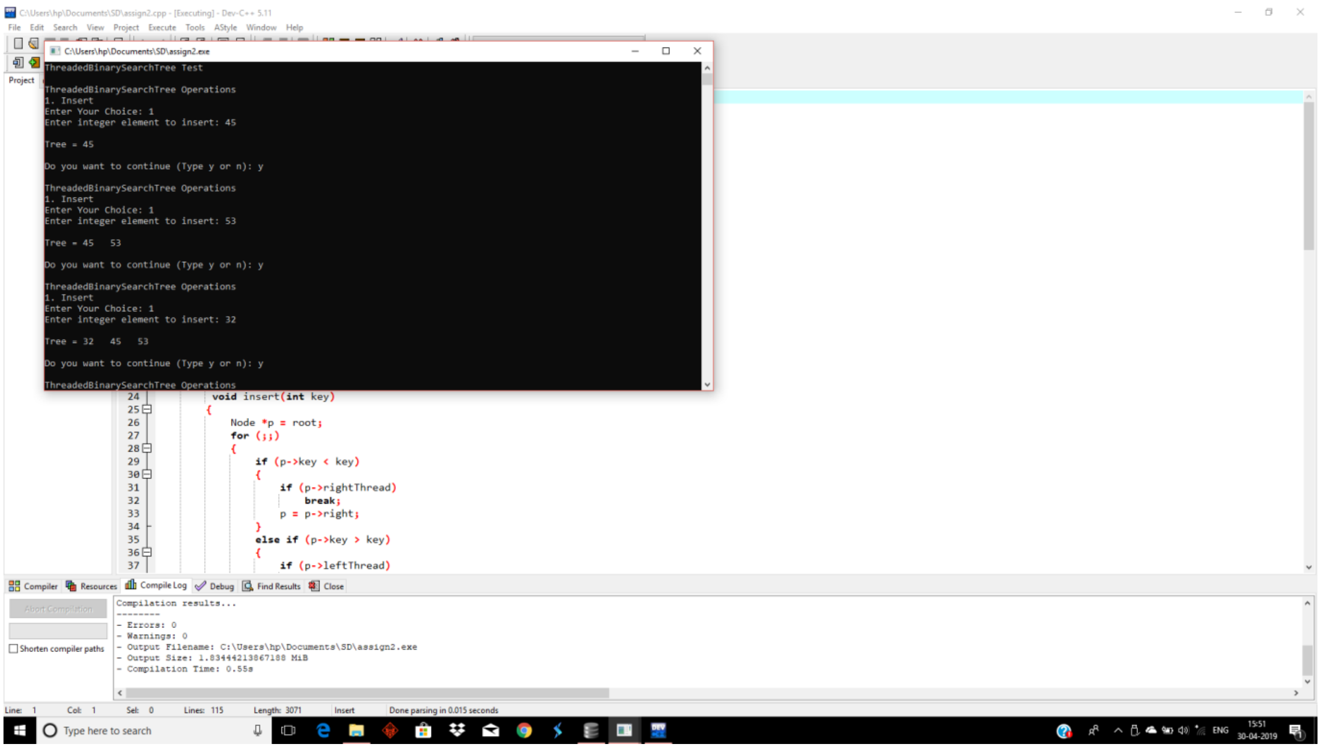
5. **Program code –**

```
#include <iostream>

#include <cstdlib>

#define MAX_VALUE 10000

using namespace std;

class Node

{

        public:

      int key;

      Node *left, *right;
```

```
        bool leftThread, rightThread;
  };
  class ThreadedBinarySearchTree
  {
          private:
        Node *root;
     public:
        ThreadedBinarySearchTree()
        {
           root = new Node();
           root->right = root->left = root;
           root->leftThread = true;
           root->key = MAX_VALUE;
        }
         void insert(int key)
        {
           Node *p = root;
           for (;;)
           {
              if (p->key < key)
              {
                 if (p->rightThread)
                    break;
                 p = p->right;
              }
              else if (p->key > key)
        {
            if (p->leftThread)
                break;
            p = p->left;
        }
        else
        {
             return;
        }
    }
    Node *tmp = new Node();
    tmp->key = key;
    tmp->rightThread = tmp->leftThread = true;
    if (p->key < key)
    {
        tmp->right = p->right;
        tmp->left = p;
        p->right = tmp;
        p->rightThread = false;
    }
    else
    {
        tmp->right = p;
        tmp->left = p->left;
        p->left = tmp;
        p->leftThread = false;
```

```
            }
        }
        void printTree()
        {
            Node *tmp = root, *p;
            for (;;)
            {
                p = tmp;
                tmp = tmp->right;
                if (!p->rightThread)
                {
                    while (!tmp->leftThread)
                    {
                        tmp = tmp->left;
                    }
                }
                if (tmp == root)
                    break;
                cout<<tmp->key<<"   ";
            }
            cout<<endl;
        }
};
int main()
{
    ThreadedBinarySearchTree tbst;
    cout<<"ThreadedBinarySearchTree Test\n";
    char ch;
    int choice, val;
    do
    {
      cout<<"\nThreadedBinarySearchTree Operations\n";
      cout<<"1. Insert "<<endl;
      cout<<"Enter Your Choice: ";
      cin>>choice;
      switch (choice)
      {
      case 1 :
         cout<<"Enter integer element to insert: ";
         cin>>val;
         tbst.insert(val);
         break;
      default :
         cout<<"Wrong Entry \n ";
         break;
      }
      cout<<"\nTree = ";
      tbst.printTree();
      cout<<"\nDo you want to continue (Type y or n): ";
      cin>>ch;
    }
    while (ch == 'Y'|| ch == 'y');
    return 0;
}
```

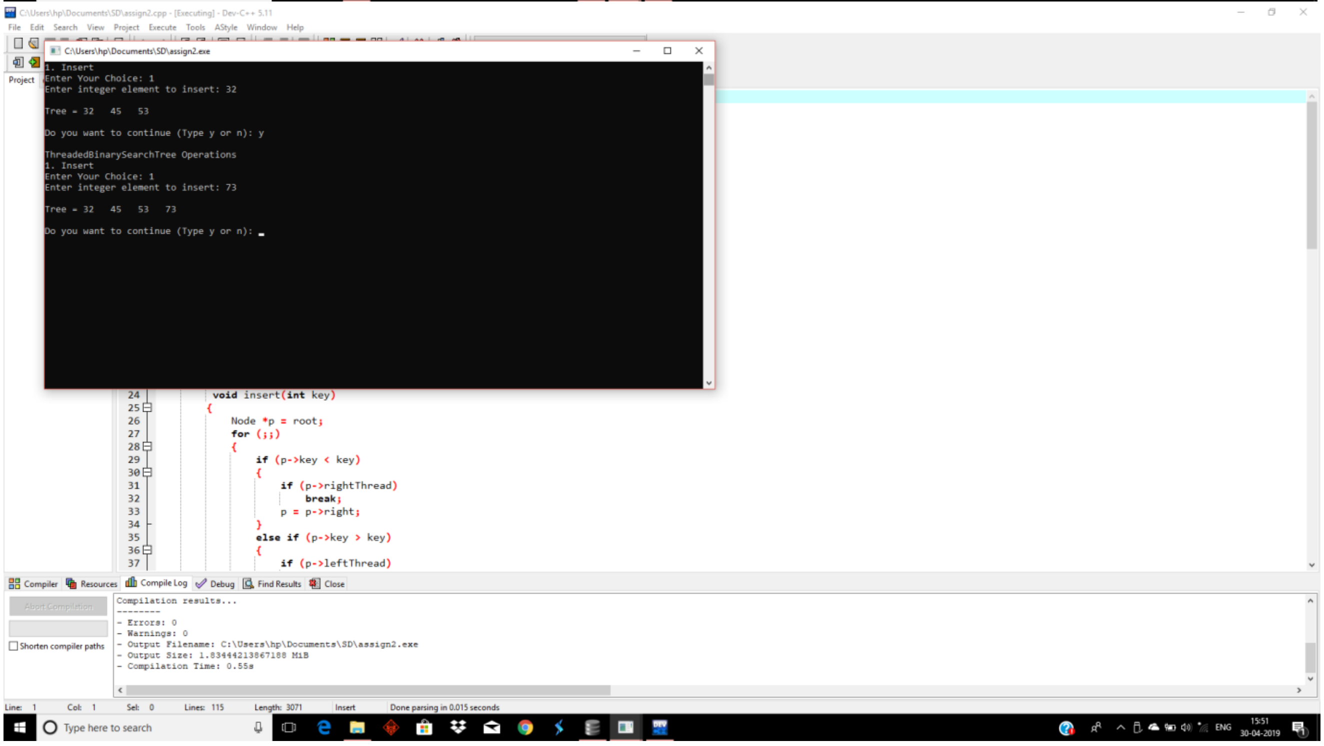
6. **Output screen shots/Copy Past From Terminal –**



```
ThreadedBinarySearchTree Test

ThreadedBinarySearchTree Operations
1. Insert
Enter Your Choice: 1
Enter integer element to insert: 45

Tree = 45

Do you want to continue (Type y or n): y

ThreadedBinarySearchTree Operations
1. Insert
Enter Your Choice: 1
Enter integer element to insert: 53

Tree = 45   53

Do you want to continue (Type y or n): y

ThreadedBinarySearchTree Operations
1. Insert
Enter Your Choice: 1
Enter integer element to insert: 32

Tree = 32   45   53

Do you want to continue (Type y or n): y

ThreadedBinarySearchTree Operations
```

```cpp
    void insert(int key)
    {
        Node *p = root;
        for (;;)
        {
            if (p->key < key)
            {
                if (p->rightThread)
                    break;
                p = p->right;
            }
            else if (p->key > key)
            {
                if (p->leftThread)
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\hp\Documents\SD\assign2.exe
- Output Size: 1.83444213867188 MiB
- Compilation Time: 0.55s
```



```
1. Insert
Enter Your Choice: 1
Enter integer element to insert: 32

Tree = 32   45   53

Do you want to continue (Type y or n): y

ThreadedBinarySearchTree Operations
1. Insert
Enter Your Choice: 1
Enter integer element to insert: 73

Tree = 32   45   53   73

Do you want to continue (Type y or n): _
```

```cpp
    void insert(int key)
    {
        Node *p = root;
        for (;;)
        {
            if (p->key < key)
            {
                if (p->rightThread)
                    break;
                p = p->right;
            }
            else if (p->key > key)
            {
                if (p->leftThread)
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\hp\Documents\SD\assign2.exe
- Output Size: 1.83444213867188 MiB
- Compilation Time: 0.55s
```

7. **Conclusion** -

Non-recursive pre-order, in-order and post-order traversal can be implemented without a stack.

Memory required to store a node increases. Each node has to store the information whether the links is normal links or threaded links.

Insertion and deletion operation becomes more difficult.