2.	Convert given binary tree into threaded binary search tree. Analyze time and space complexity of the algorithm.

```
#include<iostream>
#include<iomanip>
using namespace std;
class node {
        int data;
        node *left;
        node *right;
   bool isRightThread;
public:
        node() {
                left = NULL;
                right = NULL;
                data = 0;
        }
        friend class bsTree;
        friend class stack;
};
class stack {
        node *arr[50];
        int top;
public:
        stack() {
                top = -1;
        }
        int isFull() {
```

```
        if (top == 50)
                return 1;
        else
                return 0;
}
int isEmpty() {
        if (top == -1)
                return 1;
        else
                return 0;
}
void push(node *add) {
        if (isFull())
                cout << "stack is full" << endl;
        else {
                top++;
                arr[top] = add;
        }
}
node* pop() {
        if (isEmpty()) {
                cout << "Nothing to pop" << endl;
                return NULL;
        } else {
                node *dat = arr[top];
                top--;
                return dat;
                }
        }
};
class bsTree {
public:
        node *root;
        bsTree() {
                root = NULL;
        }
        node* creatNode();
        void insertNode();
        void disMin();
        void disTree();
        void convertToTBST(node* , node* );
   void inOrderTBST(node*);
   node* leftMost(node*);

};
node* bsTree::creatNode() {
        node* temp;
        temp = new node;
        cout << "Enter data to be inserted:";
        cin >> temp->data;
        temp->left = NULL;
        temp->right = NULL;
        return temp;
}
```

```
void bsTree::insertNode()
{
    bsTree bs;
    node *temp;
    temp = bs.creatNode();
    if (root == NULL)
        root = temp;
    else {
        node *ptr;
        ptr = root;
        while (1) {
                    if (ptr->data > temp->data)
                    {
                        if (ptr->left == NULL)
                        {
                            ptr->left = temp;
                            break;
                        } else
                    ptr = ptr->left;
              }
                else
              if (ptr->data < temp->data)
              {
                     if (ptr->right == NULL)
                    {
                     ptr->right = temp;
                     break;
                  } else
                    ptr = ptr->right;
            }
        } /* end while */
     } /* end else */
}



void bsTree::convertToTBST(node* temp, node* prev) {
    if (temp)
    {
        convertToTBST(temp->right, prev);
        if (temp->right== NULL && prev != NULL)
  {
            temp->right = prev;
            temp->isRightThread = true;
        }
        convertToTBST(temp->left, temp);
    }
}
```

```
void bsTree::inOrderTBST(node* temp)
 {
        node* cur = leftMost(temp);

        while (cur)
    {
                cout << cur->data << "\t";


                if (cur->isRightThread)
                        cur = cur->right;
                else
                        cur = leftMost(cur->right);
          } /* end while */
}

node* bsTree::leftMost(node* temp)
 {

                while (temp != NULL && temp->left!= NULL )
                        temp = temp->left;

  return temp;
}
```

```
void bsTree::disTree()

{

    if (root == NULL)

        cout << "Tree is empty." << endl;

    else

    {

        node *ptr = root;

        stack s;

        while (1){

                    while (ptr != NULL)

                    {

                        s.push(ptr);

                        cout << ptr->data << left << setw(12) << "\t" << ptr
<< "\t"

                        << left << setw(12) << ptr->left << "\t" << ptr->right

                        << endl;

                        ptr = ptr->left;

                 } /* end while */

              if (!s.isEmpty())

               {

                 ptr = s.pop();

                 ptr = ptr->right;

               }

              else

                  break;

           } /* end while */

    } /* end else */

}
```

```
/*void bsTree::disTree(node* root) {
 if (root == NULL) {
 return;
 } else {
 bsTree bs;
 bs.disTree(root->left);
 cout << root->data << " ";
 bs.disTree(root->right);
 }
 }*/

int main() {
    bsTree bs;
    int slct;
    char ch;
    do {
        cout << "#menu:" << "\n\t1.Create Binary Search Tree"
                << "\n\t 2.Display ( Preorder )" << "\n\t3. Convert to TBST"
                << "\n\t4. Display TBST" << "\n\t5. Exit " << endl;
        cout << "Select--> ";
        cin >> slct;
        switch (slct) {
        case 1:
            ch = 'y';
            while (ch == 'y' || ch == 'Y')
             {  bs.insertNode();
```

```
                    cout << "#continue insertion (y/n) ?";
                    cin >> ch;
               }
            cout << "Binary Search tree created successfully." << endl;
            break;
        case 2:
            cout << "Preorder traversal of tree is as follow:" << endl;
            bs.disTree();
            break;
        case 3:
            bs.convertToTBST(bs.root,NULL);
            cout << " BT to TBT conversion ....  successful." << endl;
             break;
        case 4:
            cout << " TBT Traversal is as follow:" << endl;
               bs.inOrderTBST(bs.root);
               break;
        case 5:
            return 0;
        default:
            cout << "Invalid Choice." << endl;
        }
        cout << "\n#menu/exit (y/n) ?";
        cin >> ch;
    } while (ch == 'y' || ch == 'Y');
    return 0;
}
```