ASSINGMENT NO.	**3**
TITLE	Threaded Binary Search Tree
PROBLEM STATEMENT /DEFINITION	Create an inordered threaded binary search tree. Perform inorder, preorder traversals without recursion and deletion of a node. Analyze time and space complexity of the algorithm.
OBJECTIVE	To understand practical implementation and usage of Threaded binary search tree for solving the problems.
OUTCOME	After successful completion of this assignment, students will be able to implement and use threaded binary search tree for efficient solution to problems
S/W PACKAGES AND **HARDWARE APPARATUS USED**	• (64-bit)64-BIT Fedora 17 or latest 64-bit update of equivalent Open source OS • Programming Tools (64-bit) Latest Open source update of Eclipse Programming frame work
REFERENCES	1. E. Horowitz S. Sahani, D. Mehata, "Fundamentals of data structures in C++", Galgotia Book Source, New Delhi, 1995, ISBN: 1678298 2. Sartaj Sahani, ―Data Structures, Algorithms andApplications in C++‖, Second Edition, University Press, ISBN:81-7371522 X.
INSTRUCTIONS FOR **WRITING JOURNAL**	1. Date 2. Assignment no. and title 3. Problem definition 4. Learning Objective 5. Learning Outcome 6. Software / Hardware requirement 7. Concepts related Theory 8. Algorithms/ Pseudo Code 9. Class ADT 10. Test cases 11. Conclusion/Analysis

Prerequisites:

- Basic knowledge of linked list, its operations and its implementation, searching techniques
- Object oriented programming features

Learning Objectives:

- To understand practical implementation and usage of threaded binary search tree for solving the problems

Learning Outcomes:

- After successful completion of this assignment, students will be able to implement and use threaded binary search tree for efficient solution to problems

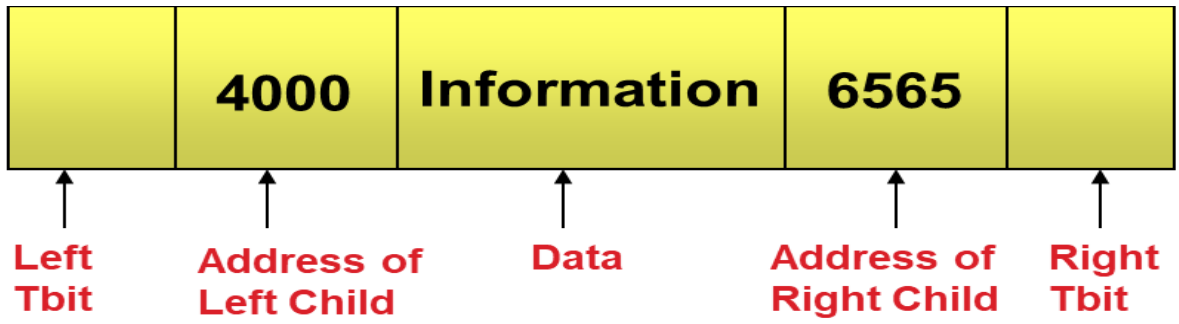
Concepts related Theory:

In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. If a binary tree consists of **n** nodes then **n+1** link fields contain NULL values. So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as **threaded binary trees**. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.

Threaded binary search tree is BST in which all right pointers of node which point to NULL are changed and made to point to inorder successor current node (These are called as single threaded trees). In completely threaded tree (or double threaded trees), left pointer of node which points to NULL is made to point to inorder predecessor of current node if inorder predecessor exists.

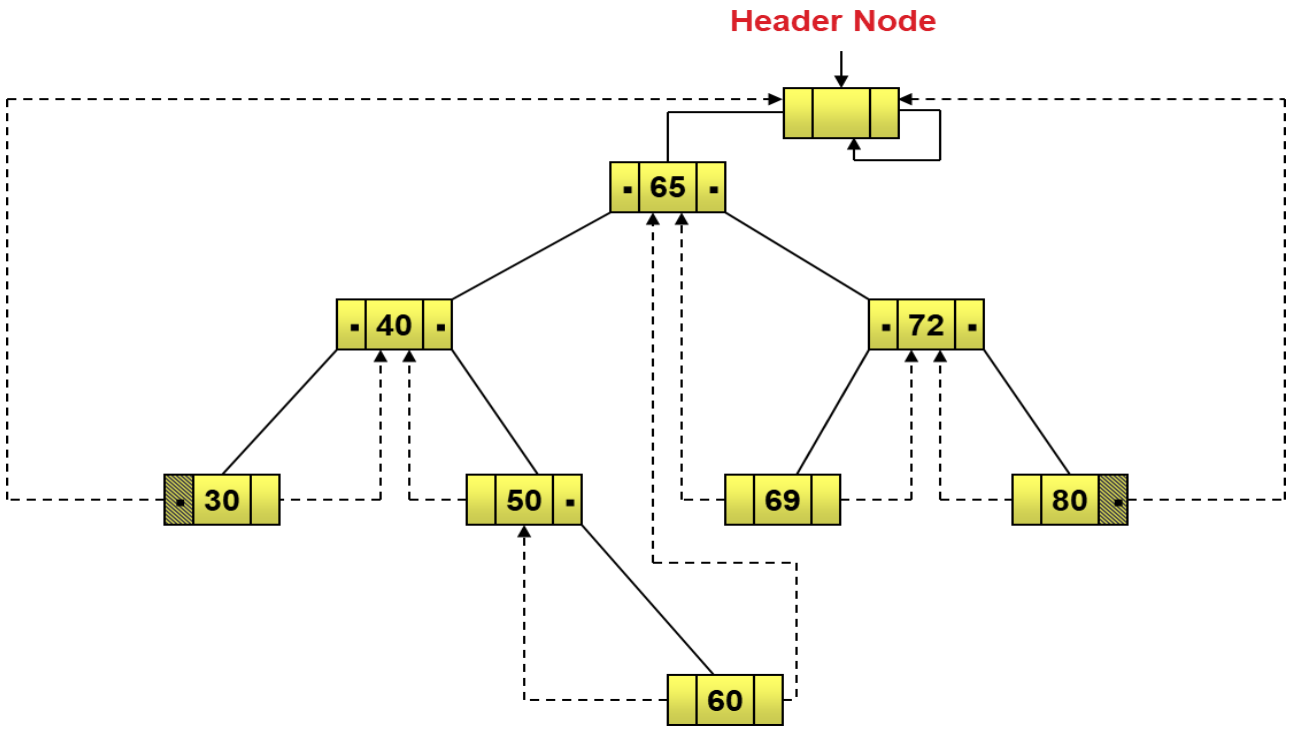
Now, there is small thing needs to be taken care of. A right or left pointer can have now two meanings: One that it points to next real node, second it is pointing inorder successor or predecessor of node that means it is creating a thread. To store this information, we added a bool in each node, which indicates whether pointer is real or thread.

TBT Node representation:



Representation:

Following is a pictorial representation of TBT along with header node–



Basic Operations:

Following are the basic operations of a tree −

- Insert − Inserts an element in a tree.
- Search − Search an element in a tree.

- Traversal − Travels an element in a tree.
- Deletion − Delete an element in a tree.

Insert Operation:

Like BST we search for the key value in the tree. If key is already present, then we return otherwise the new key is inserted at the point where search terminates. In BST, search terminates either when we find the key or when we reach a NULL left or right pointer. Here all left and right NULL pointers are replaced by threads except left pointer of first node and right pointer of last node. So here search will be unsuccessful when we reach a NULL pointer or a thread.

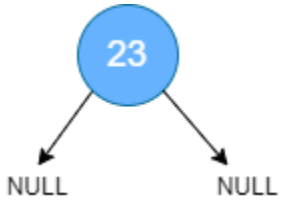
Now, let's look at the ways to insert a node:

Case 1: When a new node is inserted in an empty tree
Both left and right pointers of tmp will be set to NULL and new node becomes the root.

root = tmp;

tmp -> left = NULL;

tmp -> right = NULL;



Case 2: When new node inserted as the left child

After inserting the node at its proper place we have to make its left and right threads points to inorder predecessor and successor respectively. The node which was inorder successor. So the left and right threads of the new node will be-

tmp -> left = par ->left;

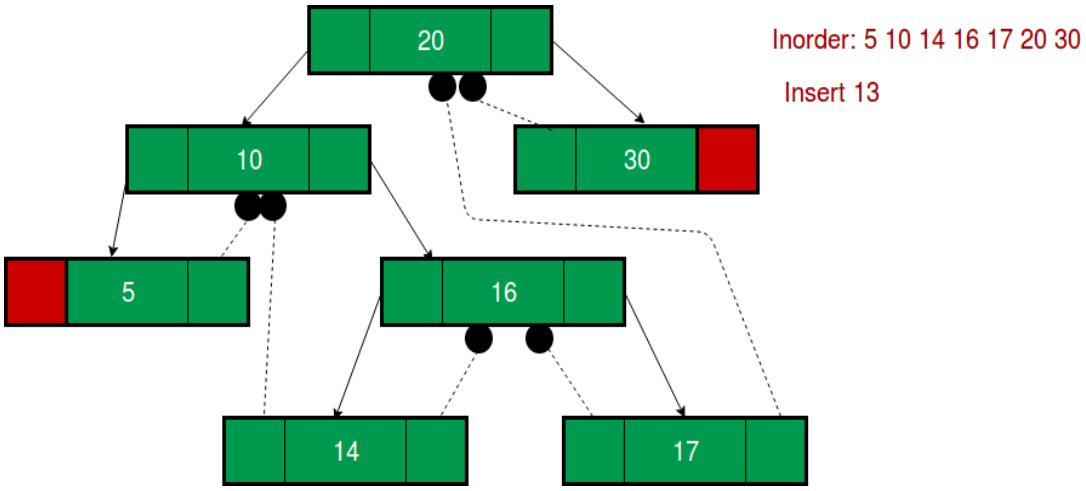
tmp -> right = par;

Before insertion, the left pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

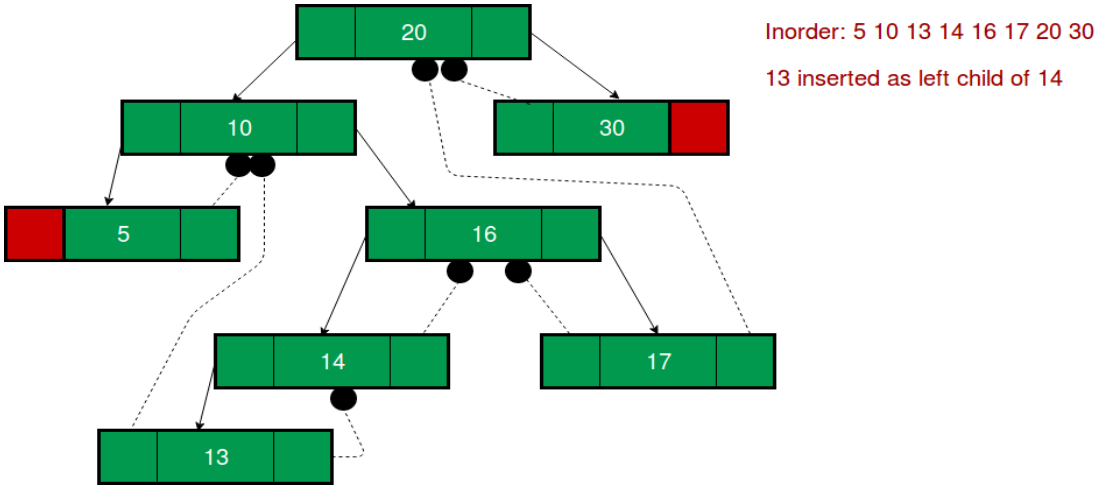
par -> lthread = false;

par -> left = temp;

Following example show a node being inserted as left child of its parent.



After insertion of 13,



Predecessor of 14 becomes the predecessor of 13, so left thread of 13 points to 10.
Successor of 13 is 14, so right thread of 13 points to left child which is 13.
Left pointer of 14 is not a thread now, it points to left child which is 13.

Case 3: When new node is inserted as the right child

The parent of tmp is its inorder predecessor. The node which was inorder successor of the parent is now the inorder successor of this node tmp. So the left and right threads of the new node will be-

tmp -> left = par;

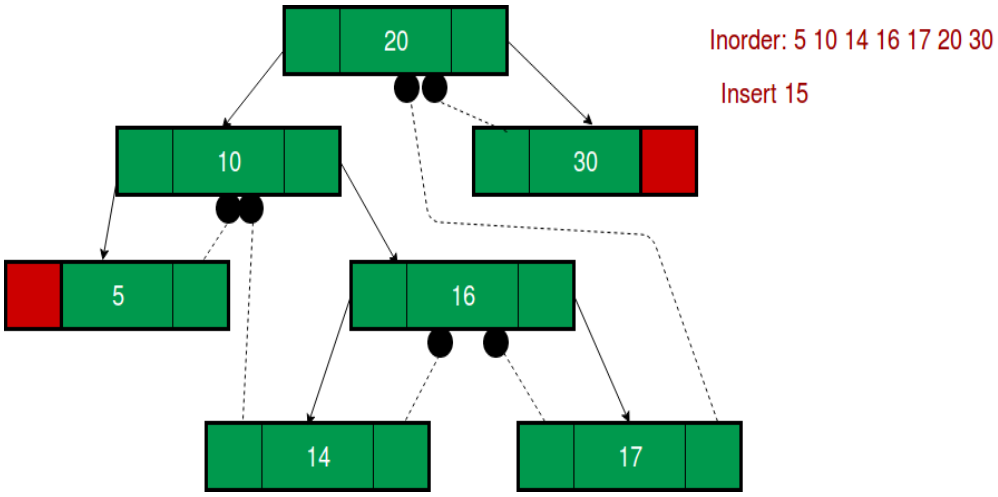
tmp -> right = par -> right;

Before insertion, the right pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

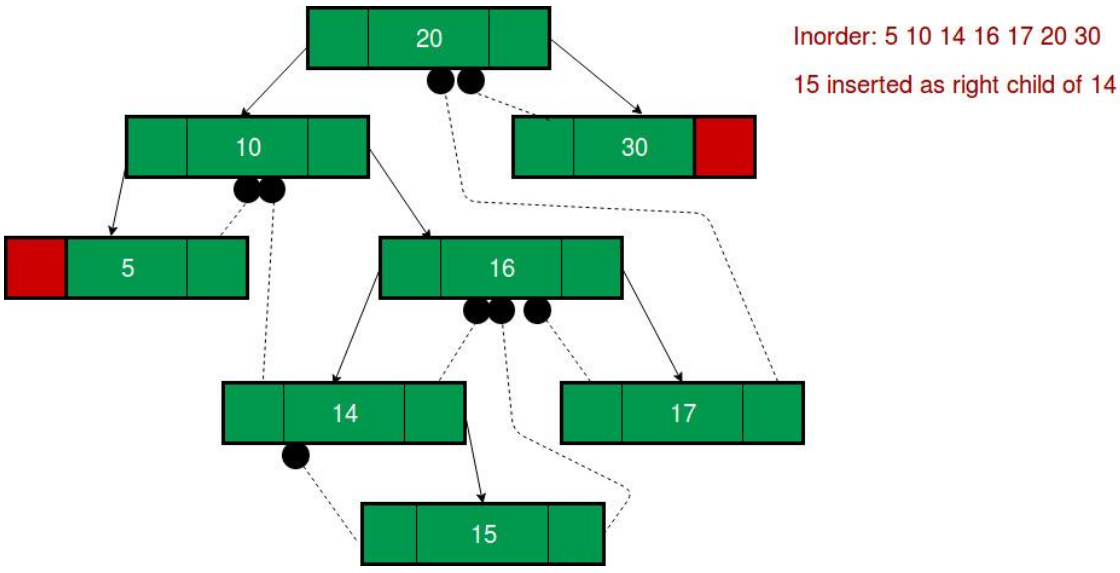
par -> rthread = false;

par -> right = tmp;

Following example shows a node being inserted as right child of its parent.



After 15 inserted,



Successor of 14 becomes the successor of 15, so right thread of 15 points to 16
Predecessor of 15 is 14, so left thread of 15 points to 14.
Right pointer of 14 is not a thread now, it points to right child which is 15.

Delete Operation:

In deletion, first the key to be deleted is searched, and then there are different cases for deleting the Node in which key is found.

Case A: Leaf Node need to be deleted

In BST, for deleting a leaf Node the left or right pointer of parent was set to NULL. Here instead of setting the pointer to NULL it is made a thread.
If the leaf Node is to be deleted is left child of its parent then after deletion, left pointer of parent should become a thread pointing to its predecessor of the parent Node after deletion.

par -> lthread = true;

par -> left = ptr -> left;

If the leaf Node to be deleted is right child of its parent then after deletion, right pointer of parent should become a thread pointing to its successor. The Node which was inorder successor of the leaf Node before deletion will become the inorder successor of the parent Node after deletion.

par -> rthread = true;

par -> right = ptr -> right;

Case B: Node to be deleted has only one child

After deleting the Node as in a BST, the inorder successor and inorder predecessor of the Node are found out.

s = inSucc(ptr);

p = inPred(ptr);

If Node to be deleted has left subtree, then after deletion right thread of its predecessor should point to its successor.

p->right = s;

Before deletion 15 is predecessor and 2 is successor of 16. After deletion of 16, the Node 20 becomes the successor of 15, so right thread of 15 will point to 20.
If Node to be deleted has right subtree, then after deletion left thread of its successor should point to its predecessor.

s->left = p;

Before deletion of 25 is predecessor and 34 is successor of 30. After deletion of 30, the Node 25 becomes the predecessor of 34, so left thread of 34 will point to 25.

Case C: Node to be deleted has two children

We find inorder successor of Node ptr (Node to be deleted) and then copy the information of this successor into Node ptr. After this inorder successor Node is deleted using either Case A or Case B.

ADT:

```
 class Node
  {
Private:
        int info;
        Node *left,*right;
        bool lthread;  // False if left pointer points to predecessor in Inorder Traversal
       bool rthread; // False if right pointer points to successor in Inorder Traversal
public:
        Node();
 };


Class TBTree
{
Node *root;
public:
//Threaded Binary search Tree operations
insert();
Search();
traversal();
delete();
};
```

Algorithms and Pseudo code:

Insert Operation:

```
// Insert a Node in Binary Threaded Tree
Algorithm insert(ikey)
{
   // Searching for a Node with given value
   Node *ptr = root;
   Node *par = NULL; // Parent of key to be inserted
   while (ptr != NULL)
   {
      // If key already exists, return
      if (ikey == (ptr->info))
      {
         print("Duplicate Key ");
         return;
      }

      par = ptr; // Update parent pointer

      // Moving on left subtree.
      if (ikey < ptr->info)
      {
         if (ptr -> lthread == false)
            ptr = ptr -> left;
         else
            break;
      }

      // Moving on right subtree.
      else
      {
         if (ptr->rthread == false)
            ptr = ptr -> right;
         else
            break;
      }
   }

   // Create a new node
   Node *tmp = new Node;
   tmp -> info = ikey;
   tmp -> lthread = true;
   tmp -> rthread = true;

   if (par == NULL)
```

```
    {
        root = tmp;
        tmp -> left = NULL;
        tmp -> right = NULL;
    }
    else if (ikey < (par -> info))
    {
        tmp -> left = par -> left;
        tmp -> right = par;
        par -> lthread = false;
        par -> left = tmp;
    }
    else
    {
        tmp -> left = par;
        tmp -> right = par -> right;
        par -> rthread = false;
        par -> right = tmp;
    }

}

// Returns inorder successor using rthread
Algorithm inorderSuccessor(ptr)
{
    // If rthread is set, we can quickly find
    if (ptr -> rthread == true)
        return ptr->right;

    // Else return leftmost child of right subtree
    ptr = ptr -> right;
    while (ptr -> lthread == false)
        ptr = ptr -> left;
    return ptr;
}

// Printing the threaded tree
Algorithm  inorder()
{
    if (root == NULL)
        print("Tree is empty");

    // Reach leftmost node
    ptr = root;
    while (ptr -> lthread == false)
        ptr = ptr -> left;
```

```
    // One by one print successors
    while (ptr != NULL)
    {
        Print(ptr -> info);
        ptr = inorderSuccessor(ptr);
    }
}


// Delete a Node from Binary Threaded Tree

Algorithm  delThreadedBST(dkey)
{
    // Initialize parent as NULL and ptrent
    // Node as root.
    par = NULL, ptr = root;
    // Set true if key is found
   found = 0;

    // Search key in BST : find Node and its
    // parent.
    while (ptr != NULL) {
        if (dkey == ptr->info) {
            found = 1;
            break;
        }
        par = ptr;
        if (dkey < ptr->info) {
            if (ptr->lthread == false)
                ptr = ptr->left;
            else
                break;
        }
        else {
            if (ptr->rthread == false)
                ptr = ptr->right;
            else
                break;
        }
    }

    if (found == 0)
        print("dkey not present in tree\n");

    // Two Children
    else if (ptr->lthread == false && ptr->rthread == false)
```

```
            root = caseC(par, ptr);

        // Only Left Child
        else if (ptr->lthread == false)
            root = caseB(par, ptr);

        // Only Right Child
        else if (ptr->rthread == false)
            root = caseB(par, ptr);

        // No child
        else
            root = caseA(par, ptr);

        return root;
    }
```

//Inorder Traversal of a Threaded Binary Tree

```
Algorithm inorder( )
{
    if(root == NULL )
    {
        print("Tree is empty");
        return;
    }
    ptr=root;
    while(ptr->lthread==false)
        ptr=ptr->left;
    while( ptr!=NULL )
    {
        print(ptr->info);
        ptr=in_succ(ptr);
    }
}
```

//Preorder Traversal of a Threaded Binary Tree

```
Algorithm preorder()
{
    if(root==NULL)
    {
        print("Tree is empty");
        return;
    }
    ptr=root;
    while(ptr!=NULL)
    {
        Print(ptr->info);
        if(ptr->lthread==false)
            ptr=ptr->left;
        else if(ptr->rthread==false)
            ptr=ptr->right;
        else
        {
            while(ptr!=NULL && ptr->rthread==true)
                ptr=ptr->right;
            if(ptr!=NULL)
                ptr=ptr->right;
        }
    }
}
```

Time complexity – O(h), where h is the height of the threaded binary search tree. The height of the tree can be 'n' in the worst case and all the keys are inserted in ascending or descending order(Skewed Tree) where all the nodes except the leaf have only one child and we may have to travel from to root to the deepest leaf node

Space complexity – O(1)

Conclusion:

- Students have understood and implemented the TBST along with all its primitive operations, to make inorder traversal faster and do it without recursion.
- Understood and written the ADT for TBST
- Able to use the TBST for efficient solution in searching problems

Sample Questions:

1. Define threaded binary tree. Explain its common uses.
2. What is the advantage of a threaded binary tree over a binary tree?
3. What are the disadvantages of a Threaded Binary Tree?
4. Explain implementation of traversal of a binary tree.
5. Why do we need Threaded binary search trees?
6. How to create a threaded BST?
7. How to convert a Given Binary Tree to Threaded Binary Tree?
8. What are the drawbacks of bi-threaded trees? Are single threaded trees enough to do traversals on the tree? Justify.
9. Did you require stacks to do get the output along with threads? Justify.
10. Justify that only single threads are required to traverse the tree efficiently.