Rajdhani
DATE / /

Roll No. 23355

Title: Assignment 6 : Threaded Binary Tree

Aim : To implement a threaded binary tree

Problem statement : Implement Inorder Threaded binary tree. Traverse the implemented tree in pre-order and inorder.

Theory :

- Limitations of normal binary tree:

1) Too many null pointers :
The binary tree node have at most two children. But if they have only one child or no children, the link part in the link representation remains null.

n : number of nodes
number of non-null links : $n-1$
total link : $2n$
null links : $2n-(n-1) = \underline{\underline{n+1}}$

2) Tempory data structure (stack) is required to implement non recursive traversal algorithm.

- Threaded Binary Tree

The concept of Threaded Binary Tree (TBT) is introduced to overcome the limilations of binary tree.
The idea of TBT is to make inorder traversal faster and do it without stack & without recursion.
A binary tree is made threaded by making all right child pointers that would normally be NULL point to inorder successor of node (if it exist).

There are two types of TBT :

1) Single Threaded : Where a NULL right pointer is made to point to inorder successor (if successor exist).

2) Double threaded : Where both the left and right pointers are made to point to inorder predecessor and inorder successor respectively.

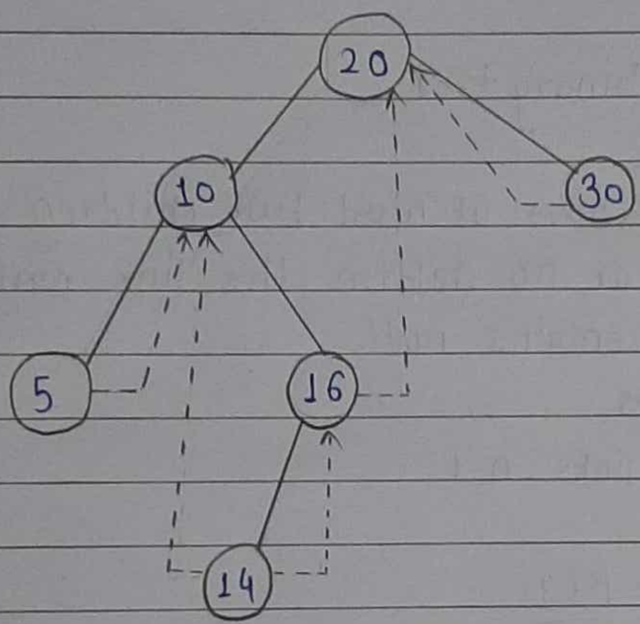


fig. TBT (Double threaded)

Structure of Threaded Node :

```
struct Node
{
    int data ;
    Node *left , *right ;
    bool lthread , rthread ;
}
```

lthread	left	data	right	rthread

fig. Representation of Node in TBT

Advantages of TBT over normal binary tree:

1) No wastage of memory for null pointers.
2) Non recursive traversal without stack
3) Node can keep record of its roots.
4) Backward traverse is possible

- Algorithm :

1) TBT creation using inorder threading :

```
Procedure Insert (data)
        ptr ← root
        while ptr ≠ NULL
            // check for duplicate value
              If data = ptr→data
                  print "duplicate value"
                  return
            // check for right child or left child
             If data < ptr→data
                 If ptr→lthread = false
                      ptr = ptr→left
                 Else
                      break
            Else
```

```
~~If data~~
    If ptr->rthread == false
        ptr = ptr->right
    Else
        break

End while
Node * newN = getNode (data)
//Insertion in empty tree i.e. creation
If ptr = NULL
    root = ~~ptr~~ newN
    newN->left = NULL
    newN->right = NULL
// Insertion as left child
~~Else If ptr->~~
Else If newN->data < ptr->data
    newN->left = ptr->left
    newN->right = ptr
    ptr->lthread = false
    ptr->left = newN

// insertion as right child
Else
    newN->left = ptr
    newN->right = ptr->right
    ptr->rthread = false
    ptr->right = newN

End If
return true
```

Example:

1) case 1 : Insertion in empty tree

Insert 20 =>

root = NULL

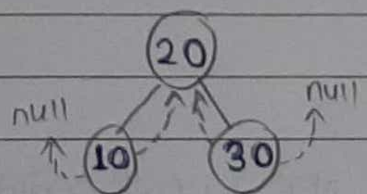
∴ root = newN.

newN → left = NULL

newN → right = NULL

T	/	20	/	T

2) case 2 : Insertion as left child



insert 5 =>

ptr is pointing at node at value 10.

newN contains value 5.

newN → left = ptr → left

newN → right = ptr

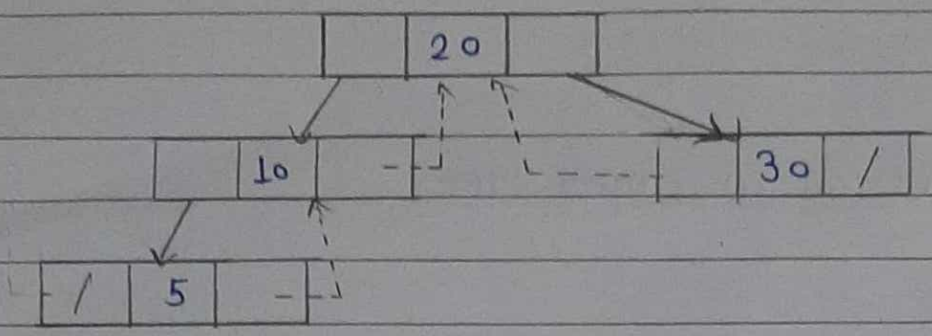
ptr → lthread = false

ptr → left = newN

∴ newN → left is pointing to null

newN → right is pointing to node with value 10

lthread of ptr is removed & ptr → left is pointing to newN.

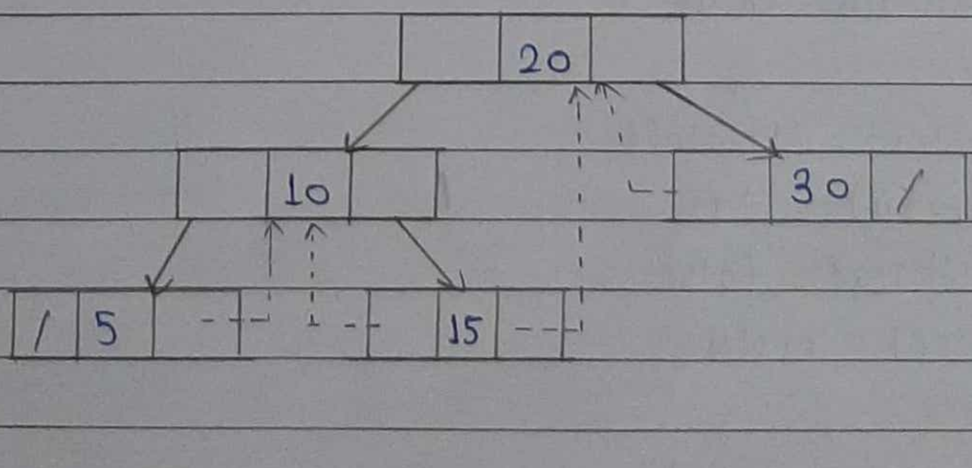


3) case 3: Insertion as right child

insert 15 ⇒
ptr is pointing to node with value 10.
newN contains node with value 15

newN→left = ptr
newN→right = ptr→right
ptr→rthread = false
ptr→right = newN

newN→right is pointing to ptr & newN→right ~~is~~ pointing to node with value 20. rthredd of ptr is removed. ptr→right is pointing to newN



2) Inorder traversal :

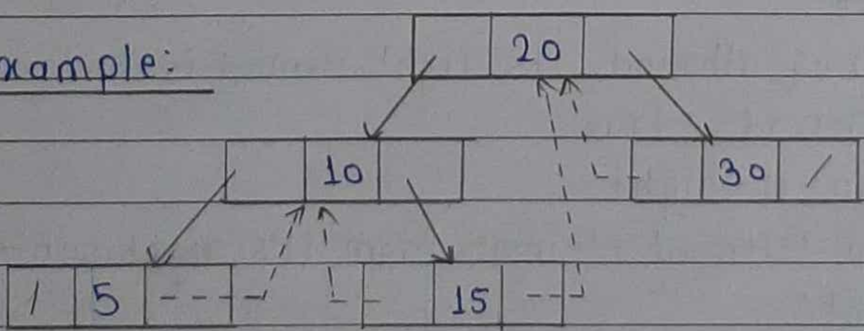
```
Procedure inorder
    If root = NULL
        print "Empty Tree"
        return
    Else
        curr = root
        // find leftmost element
        while curr -> lthread = false
            curr = curr -> left
        End while
        // Traverse till last node
        while curr ≠ NULL
            print curr -> data
            // find inorder successor
            curr = inorderSuccessor (curr)
        End while
End

Procedure inordersuccessor ( Node *n)
    ~~If n = NULL~~
    // if node has rthread, its right element is successor
    If n -> rthread = = true
        return n -> right
    // Else find leftmost element from it's right subtree
    n = n -> right
    while n -> lthread = false
        n = n -> left
    return n
```

Rajdhani
DATE / /

3) Preorder Traversal :

Procedure Preorder
```
    If root = NULL
        print "Empty tree"
        return
    Else
      // print all nodes
      while curr ≠ NULL
          print curr→data
          // If it has left child, move to left
          If curr→lthread = false
              curr = curr→left
          //Else move to right subtree
          while curr→rthread = true && curr→right ≠NULL
              curr = curr→right
          End while
          If curr != NULL
              curr = curr→right
      End while
  End
```

Example:



Inorder traversal : 5, 10, 15, 20, 30
Preorder traversal : 20, 10, 5, 15, 30

- Validations:

1) Duplicate numbers are not allowed
2) only integer data for tree creation & insertion.

Test cases:

1) Random input
2) Sorted input
3) Input for skewed tree concept

Conclusion:

The idea of TBT is to make inorder traversal faster & do it without stack & without recursion.

Space complexity of TBT is O(1)
For inorder traversal it take O(n) time without recursion & stack.