**Experiment No 2**

**AIM:** Study of Threaded Binary Tree as a non-linear Data Structure

**PROBLEM STATEMENT:** C++ program to implement a threaded binary tree and perform five operations on it:

insertion, in-order traversal, deletion, search, and display

**REQUIREMENT:**Turbo C/ GCC Compiler

**OPERATING SYSTEM:** Windows/Linux/Unix.

**THEORY:**

**Threaded Binary Tree**

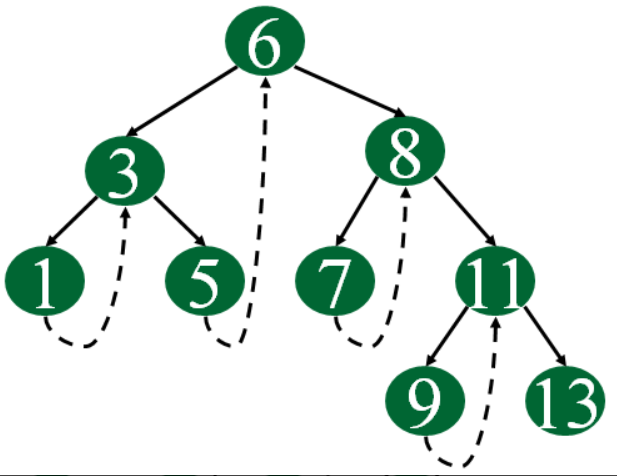
Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

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. Threads that link nodes directly to their in-order predecessor or successor, thereby providing a way to traverse the tree without using recursion or a stack.

Threaded binary trees can be useful when space is a concern, as they can eliminate the need for a stack during traversal. However, they can be more complex to implement than standard binary trees.

**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.

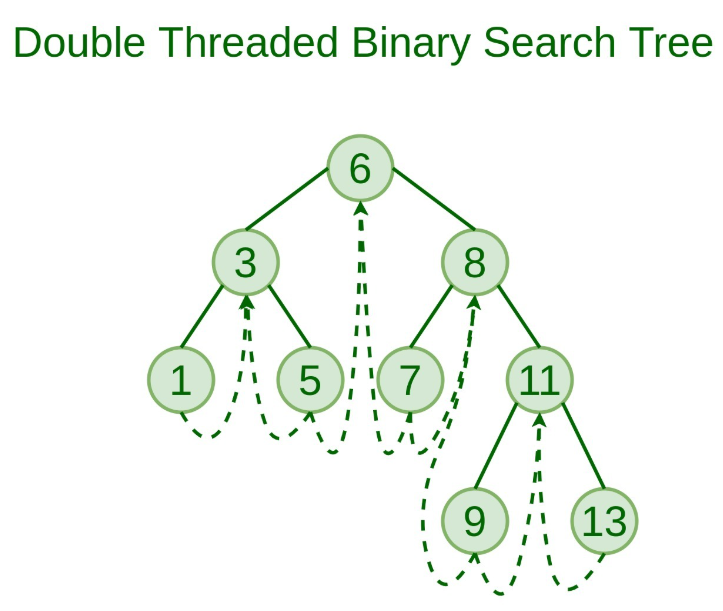
The threads are also useful for fast accessing ancestors of a node.  
Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



**Double Threaded Binary Search Tree:** is a binary search tree in which the nodes are not every left NULL pointer points to its inorder predecessor and the right NULL pointer points to the inorder successor.  
The threads are also useful for fast accessing the ancestors of a node.

 Double Threaded Binary Search Tree is one of the most used types of Advanced data structures used in many real-time applications like places where there are recent insertion and traversal of all elements of the search tree.

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



**CONCLUSION:** Program for threaded binary treee is implemented successfully.

// double threaded binary search tree

#include <iostream>

using namespace std;

struct thtree

{

bool isleft ;

struct thtree \*left ;

int data ;

struct thtree \*right ;

bool isright ;

} ;

void insert ( struct thtree \*\*, int ) ;

void delete1 ( struct thtree \*\*, int ) ;

void search ( struct thtree \*\*, int, struct thtree \*\*,

struct thtree \*\*, int \* ) ;

void inorder ( struct thtree \* ) ;

void deltree ( struct thtree \*\* ) ;

void find ( struct thtree \*\*, int ) ;

int main( )

{

struct thtree \*th\_head ;

int ch, val;

th\_head = NULL ; /\* empty tree \*/

while(1)

{

cout<<"\n1:insert\n2:delete\n3:display\n4: Search\n5:Exit\n";

cout<<"Enter the choice: ";

cin>>ch;

switch(ch)

{

case 1:

cout<<"\nEnter the data";

cin>>val;

insert ( &th\_head, val) ;

break;

case 2:

cout<<"\nEnter the data";

cin>>val;

delete1 ( &th\_head, val) ;

break;

case 3:

cout<<"\nThreaded Binary tree:\n" ;

inorder ( th\_head ) ; break;

case 4:

cout<<"\nEnter the data";

cin>>val;

find ( &th\_head, val) ; break;

case 5:

exit(0);

}

}

return 0;

}

/\* inserts a node in a threaded binary tree \*/

void insert ( struct thtree \*\*s, int num )

{

struct thtree \*p, \*z, \*head = \*s ;

/\* allocating a new node \*/

z = new thtree();

z -> isleft = true ; /\* indicates a thread \*/

z -> data = num ; /\* assign new data \*/

z -> isright = true ; /\* indicates a thread \*/

/\* if tree is empty \*/

if ( \*s == NULL )

{

head = new thtree();

/\* the entire tree is treated as a left sub-tree of the head node \*/

head -> isleft = false ;

head -> left = z ; /\* z becomes leftchild of the head node \*/

head -> data = -9999 ; /\* no data \*/

head -> right = head ; /\* right link will always be pointing

to itself \*/

head -> isright = false ;

\*s = head ;

z -> left = head ; /\* left thread to head \*/

z -> right = head ; /\* right thread to head \*/

}

else /\* if tree is non-empty \*/

{

p = head -> left ;

/\* traverse till the thread is found attached to the head \*/

while ( p != head )

{

if ( p -> data > num )

{

if ( p -> isleft != true ) /\* checking for a thread \*/

p = p -> left ;

else

{

z -> left = p -> left ;

p -> left = z ;

p -> isleft = false ; /\* indicates a link \*/

z -> isright = true ;

z -> right = p ;

return ;

}

}

else

{

if ( p -> data < num )

{

if ( p -> isright != true )

p = p -> right ;

else

{

z -> right = p -> right ;

p -> right = z ;

p -> isright = false ; /\* indicates a link \*/

z -> isleft = true ;

z -> left = p ;

return ;

}

}

}

}

}

}

/\* deletes a node from the binary search tree \*/

void delete1 ( struct thtree \*\*root, int num )

{

int found ;

struct thtree \*parent, \*x, \*xsucc ;

/\* if tree is empty \*/

if ( \*root == NULL )

{

cout<<"\nTree is empty" ;

return ;

}

parent = x = NULL ;

/\* call to search function to find the node to be deleted \*/

search ( root, num, &parent, &x, &found ) ;

/\* if the node to deleted is not found \*/

if ( found == false )

{

cout<<"\nData not found";

return ;

}

/\* if the node to be deleted has two children \*/

if ( x -> isleft == false && x -> isright == false )

{

parent = x ;

xsucc = x -> right ;

while ( xsucc -> isleft == false )

{

parent = xsucc ;

xsucc = xsucc -> left ;

}

x -> data = xsucc -> data ;

x = xsucc ;

}

/\* if the node to be deleted has no child \*/

if ( x -> isleft == true && x -> isright == true )

{

/\* if node to be deleted is a root node \*/

if ( parent == NULL )

{

( \*root ) -> left = \*root ;

( \*root ) -> isleft = true ;

free ( x ) ;

return ;

}

if ( parent -> right == x )

{

parent -> isright = true ;

parent -> right = x -> right ;

}

else

{

parent -> isleft = true ;

parent -> left = x -> left ;

}

free ( x ) ;

return ;

}

/\* if the node to be deleted has only rightchild \*/

if ( x -> isleft == true && x -> isright == false )

{

/\* node to be deleted is a root node \*/

if ( parent == NULL )

{

( \*root ) -> left = x -> right ;

free ( x ) ;

return ;

}

if ( parent -> left == x )

{

parent -> left = x -> right ;

x -> right -> left = x -> left ; //predecessor

}

else

{

parent -> right = x -> right ;

x -> right -> left = parent ;

}

free ( x ) ;

return ;

}

/\* if the node to be deleted has only left child \*/

if ( x -> isleft == false && x -> isright == true )

{

/\* the node to be deleted is a root node \*/

if ( parent == NULL )

{

parent = x ;

xsucc = x -> left ;

while ( xsucc -> isright == false )

xsucc = xsucc -> right ;

xsucc -> right = \*root ;

( \*root ) -> left = x -> left ;

free ( x ) ;

return ;

}

if ( parent -> left == x )

{

parent -> left = x -> left ;

x -> left -> right = parent ;

}

else

{

parent -> right = x -> left ;

x -> left -> right = x -> right ;

}

free ( x ) ;

return ;

}

}

void find ( struct thtree \*\*root, int num )

{

int found ;

struct thtree \*parent, \*x, \*xsucc ;

/\* if tree is empty \*/

if ( \*root == NULL )

{

cout<<"\nTree is empty" ;

return ;

}

parent = x = NULL ;

/\* call to search function to find the node to be deleted \*/

search ( root, num, &parent, &x, &found ) ;

/\* if the node to deleted is not found \*/

if ( found == false )

{

cout<<"\nData not found";

return ;

}

else

cout<<"\nData found";

}

/\* returns the address of the node to be deleted, address of its parent and

whether the node is found or not \*/

void search ( struct thtree \*\*root, int num, struct thtree \*\*par,

struct thtree \*\*x, int \*found )

{

struct thtree \*q ;

q = ( \*root ) -> left ;

\*found = false ;

\*par = NULL ;

while ( q != \*root )

{

/\* if the node to be deleted is found \*/

if ( q -> data == num )

{

\*found = true ;

\*x = q ;

return ;

}

\*par = q ;

if ( q -> data > num )

{

if ( q -> isleft == true )

{

\*found = false ;

x = NULL ;

return ;

}

q = q -> left ;

}

else

{

if ( q -> isright == true )

{

\*found = false ;

\*x = NULL ;

return ;

}

q = q -> right ;

}

}

}

/\* traverses the threaded binary tree in inorder \*/

void inorder ( struct thtree \*root )

{

struct thtree \*p ;

p = root -> left ;

while ( p != root )

{

while ( p -> isleft == false )

p = p -> left ;

cout<< p -> data ;

while ( p -> isright == true )

{

p = p -> right ;

if ( p == root )

break ;

cout<< p -> data ;

}

p = p -> right ;

}

}

void deltree ( struct thtree \*\*root )

{

while ( ( \*root ) -> left != \*root )

delete1 ( root, ( \*root ) -> left -> data ) ;

}