//Assignment no 5

#include <iostream>

using namespace std;

class node

{

public:

int data;

node\* left;

node\* right;

int lbit, rbit;

};

class TBT

{

node\* root;

public:

node\* head;

TBT()

{

root = NULL;

}

node \* getroot()

{

return(root);

}

void createtree();

void inorder(node\* root);

};

void TBT :: createtree()

{

char ans;

node\* new1, \*curr;

head = new node;

head->data = -999;

head->right = head;

do

{

new1 = new node;

new1->lbit = 0;

new1->rbit = 0;

cout << "\nEnter your data: ";

cin >> new1->data;

if(root == NULL)

{

root = new1;

head->left = root;

head->lbit = 1;

root->left = head;

root->right = head;

}

else

{

curr = root;

int flag = 0;

if(new1->data < curr->data)

{

while(flag == 0)

{

if(curr->lbit == 0)

{

new1->left = curr->left;

curr->left = new1;

curr->lbit = 1;

new1->right = curr;

flag = 1;

}

else

{

curr = curr->left;

}

}

}

else if(new1->data > curr->data)

{

while(flag == 0)

{

if(curr->rbit == 0)

{

new1->right = curr->right;

curr->right = new1;

curr->rbit = 1;

new1->left = curr;

flag = 1;

}

else

{

curr = curr->right;

}

}

}

else

{

cout << "\nData already exists!";

}

}

cout << "\nDo you want to add more node?(y/n)";

cin >> ans;

}while(ans == 'Y' || ans == 'y');

}

void TBT :: inorder(node \*root)

{

node \*temp;

temp = root;

int flag =0;

if(root == NULL)

{

cout<<"\nTree is empty";

}

else

{

while(temp != head)

{

if(temp->lbit==1 && flag ==0)

{

temp = temp ->left;

}

else

{

cout<<" "<<temp ->data;

if(temp ->rbit == 1)

{

temp = temp ->right;

flag=0;

}

else

{

temp = temp->right;

flag =1;

}

}

}

}

}

int main()

{

TBT t;

int ch;

do

{

cout << "\n-------MENU-------";

cout << "\n1. Insert";

cout << "\n2. Display";

cout << "\n3. Exit";

cout << "\nEnter your choice: ";

cin >> ch;

switch(ch)

{

case 1:

t.createtree();

break;

case 2:

t.inorder(t.getroot());

break;

case 3:

exit(0);

break;

default:

cout << "\nInvalid Choice Entered!";

}

}while(ch != 3);

return 0;

}

output:

gescoe@gescoe-OptiPlex-3010:~/Desktop/SE-A-55$ g++ thread.cpp

gescoe@gescoe-OptiPlex-3010:~/Desktop/SE-A-55$ ./a.out

-------MENU-------

1. Insert
2. Display
3. Exit

Enter your choice: 1

Enter your data: 5

Do you want to add more node?(y/n)y

Enter your data: 1

Do you want to add more node?(y/n)y

Enter your data: 10

Do you want to add more node?(y/n)n

-------MENU-------

1. Insert
2. Display
3. Exit

Enter your choice: 2

1. 5 10

-------MENU-------

1. Insert
2. Display
3. Exit

Enter your choice: 3

[gescoe@gescoe-OptiPlex-3010](mailto:gescoe@gescoe-OptiPlex-3010):~/Desktop/SE-A-55$

### ****Threaded Binary Tree (TBT): Theory, Algorithm, and Implementation****

#### ****Introduction****

A **Threaded Binary Tree (TBT)** is a variant of the **binary tree** in which null pointers (usually used for the left and right child pointers of a node when there is no child) are replaced with **threads**. These threads connect the node to its **in-order successor** (in the case of the right child) or **in-order predecessor** (in the case of the left child). This structure significantly improves the efficiency of **in-order traversal** of the tree, as it eliminates the need for recursion or stack-based traversal mechanisms, which are typically required in standard binary trees.

In essence, a **Threaded Binary Tree** stores additional information (called **threads**) in place of NULL child pointers, making it easy to traverse the tree in **in-order sequence** without recursion.

#### ****Advantages of Threaded Binary Trees****

1. **Efficient In-order Traversal**:
   * In regular binary trees, in-order traversal usually requires either recursion or a stack, which takes up both time and space. In threaded binary trees, threads allow for a traversal mechanism that does not require recursion or a stack, significantly reducing the time and space complexity of traversal operations.
2. **Memory Efficiency**:
   * By utilizing the NULL pointers as threads (which would otherwise not serve any purpose), threaded binary trees utilize the memory more efficiently.
3. **Direct Navigation**:
   * Using threads allows for direct access to the in-order predecessor and successor nodes. This makes the operations faster compared to regular trees where you have to traverse the tree repeatedly for each node.
4. **Simplified Algorithms**:
   * The algorithm for tree traversal becomes simplified because we do not need to track the parent or use auxiliary data structures like stacks. The threads make in-order traversal **iterative** and efficient.
5. **Better Traversal Time**:
   * With threaded binary trees, the tree traversal becomes **O(n)**, where n is the number of nodes in the tree. This is more efficient than the traditional recursive tree traversal, especially in large trees.

#### ****Types of Threaded Binary Trees****

1. **Single Threaded Binary Tree**:
   * In a **single threaded binary tree**, only one of the child pointers (either left or right) is used as a thread to point to the next node in the in-order traversal. The left thread points to the in-order predecessor and the right thread points to the in-order successor. Typically, the **right thread** is used in single-threaded binary trees.
2. **Double Threaded Binary Tree**:
   * A **double threaded binary tree** is a more advanced version where both **left** and **right** pointers are used as threads. In this case:
     + If a node has no left child, the left pointer is a thread pointing to the node’s in-order predecessor.
     + If a node has no right child, the right pointer is a thread pointing to the node’s in-order successor.
   * This kind of threaded tree allows traversal in both directions, which is useful for certain applications that require bi-directional traversals.

#### ****Threaded Binary Tree Node Structure****

Each node in a threaded binary tree contains the following attributes:

1. **Data**: Stores the value or key of the node.
2. **Left Pointer (**left**)**: Points to the left child of the node, or if there is no left child, it points to the **in-order predecessor**.
3. **Right Pointer (**right**)**: Points to the right child of the node, or if there is no right child, it points to the **in-order successor**.
4. **Left Bit (**lbit**)**: A flag used to indicate whether the left pointer is a normal child pointer (0) or a thread pointer (1).
5. **Right Bit (**rbit**)**: A flag used to indicate whether the right pointer is a normal child pointer (0) or a thread pointer (1).

#### ****Basic Operations on Threaded Binary Trees****

1. **Insertion**:
   * Inserting a node into a threaded binary tree follows the same basic rules as inserting a node into a regular binary search tree (BST). The new node is inserted based on comparisons, and if the left or right child is NULL, it is replaced by a thread pointing to the predecessor or successor respectively. The lbit and rbit flags are set accordingly.
2. **In-order Traversal**:
   * The primary advantage of threaded binary trees is their ability to perform an in-order traversal without recursion or a stack. By following the threads, you can easily visit each node in the correct order of traversal.
3. **Deletion**:
   * Deletion in a threaded binary tree is similar to deletion in a regular binary tree. After a node is deleted, the corresponding threads must be re-adjusted to ensure that the tree structure and the threading mechanism remain intact.
4. **Searching**:
   * Searching in a threaded binary tree is similar to searching in a regular binary search tree. You compare the key of the current node with the key you’re searching for and navigate the tree left or right based on the result of the comparison. If no child is found in either direction, you can follow the threads to find the in-order predecessor or successor.

#### ****Algorithm for Inserting a Node in a Threaded Binary Tree****

Here is a step-by-step description of the insertion algorithm in a threaded binary tree:

1. **Create a New Node**:
   * A new node is created with the given data. The left and right pointers are initially set to NULL. The lbit and rbit flags are also set to 0.
2. **Find the Correct Position**:
   * Starting from the root, traverse the tree based on comparisons. If the data of the new node is smaller than the current node’s data, move to the left child; if the data is larger, move to the right child.
3. **Insertion**:
   * When you find an appropriate spot (where a left or right child is NULL), insert the new node there.
   * If the left child is NULL, create a thread and make the left pointer of the parent node point to the new node as its in-order predecessor.
   * If the right child is NULL, create a thread and make the right pointer of the parent node point to the new node as its in-order successor.
4. **Update the** lbit **and** rbit **Flags**:
   * Set the lbit and rbit flags for the parent node based on whether the pointers are threads or regular child pointers.

#### ****Algorithm for In-order Traversal of a Threaded Binary Tree****

The algorithm for performing an **in-order traversal** in a threaded binary tree can be described as follows:

1. **Start at the Leftmost Node**:
   * Begin traversal from the root. If the left child is not NULL, move to the leftmost node of the tree. This node will be the smallest in the in-order traversal.
2. **Follow Threads for Traversal**:
   * For each node, print its data.
   * If the node has a left thread (lbit == 1), move to the in-order predecessor (left thread).
   * If the node has a right thread (rbit == 1), move to the in-order successor (right thread).
3. **End Condition**:
   * The traversal stops when the current node is the head node (sentinel node), marking the end of the traversal.

#### ****Code Explanation:****

Below is an explanation of the **C++ implementation** provided:

1. node **Class**:
   * This class represents a node in the threaded binary tree. It has data, left and right pointers, and lbit and rbit flags.
2. TBT **Class**:
   * This class manages the threaded binary tree. It contains methods for creating the tree (createtree) and performing in-order traversal (inorder).
3. createtree **Method**:
   * This method allows the user to insert new nodes into the tree interactively. The tree is created based on the comparison of values, and the lbit and rbit flags are updated accordingly.
4. inorder **Method**:
   * This method performs an **in-order traversal** using the threads. It visits each node based on the threads and prints the node's data.
5. **Main Program**:
   * The program runs a loop where the user can insert nodes or display the tree by performing an in-order traversal. The menu provides options for inserting nodes, displaying the tree, or exiting the program.

#### ****Example Walkthrough****

Let's consider inserting the values 5, 1, and 10 into the threaded binary tree:

1. **Insert 5**:
   * As the tree is empty, the node with data 5 is inserted as the root node. Both left and right pointers are set to point to the head, and lbit and rbit are set to 0.
2. **Insert 1**:
   * The data 1 is smaller than 5, so it is inserted as the left child of 5. The left pointer of 5 is updated to point to 1 as its in-order predecessor, and the lbit is set to 1.
3. **Insert 10**:
   * The data 10 is greater than 5, so it is inserted as the right child of 5. The right pointer of 5 is updated to point to 10 as its in-order successor, and the rbit is set to 1.

**In-order traversal** of this tree would output: 1 5 10, which corresponds to the in-order traversal sequence of the tree.

#### ****Conclusion****

Threaded Binary Trees (TBTs) are an excellent data structure for improving the efficiency of in-order traversal, particularly in scenarios where recursion or a stack-based approach would be costly. They offer a memory-efficient and time-efficient solution by using threads to replace null pointers. The **TBT algorithm** provides a simple way to implement insertion and traversal operations that make them suitable for applications where tree traversal is a key operation.