Ngoc Nguyen NITK17K - 1700062

Final Project

Question
Using the provided ADT for trees in the class, write a binary tree class to store the arbitrary number of items, and implement two traversal algorithms BFS and DFS. Write a simple "main" function with a menu for the user to test your trees functionalities.

Github link
https://github.com/NgocNguyenGit/Algorithms-and-Data-Structures

Internal logic of the program
Firstly, the author would like to analyse the project requirement by briefly defining some relative acronyms:

- ADT stands for abstract data type. An Abstract List (or List ADT) is linearly ordered data where the programmer explicitly defines the ordering.
- BFS stands for breadth-first traversals
- DFS stands for depth-first traversals

This project applies knowledges of:

- Node-based storage
- Stack
- Queue

Binary tree implementation
The Final project implements a simple binary tree using C++ whose nodes have at most two child nodes – the left and right child.

```
//Class defined as a node
class Node {
private:
public:
    int key_value;
    Node *left;
    Node *right;
};
```

Figure 1.1. Class defined as a node

By using the provided ADT for trees in the class, the author created a simple Binary_tree class including constructor, destructor, and some relevant operations to support the purpose of the project – test the two traversal algorithms breadth-first traversal and depth-first traversal of the arbitrary number of items.

```
//Class binary tree implementation
class Binary_tree {
private:
public:
    Binary_tree();
    ~Binary_tree();

    Node *root;
```

Figure 1.2. A part of the Binary_tree class

Depth-first traversal implementation

In this project, the DFS algorithm is implemented in two methods, one using stack and one without. Three recursive implementations of a DFS are given here to traverse a tree:

- PreOrder traversal
- InOrder traversal
- PostOrder traversal

The logic of class DFS_preOrder is that the process will first visit the root, then traverse the left subtree, and traverse the right subtree.

```
void Binary_tree::DFS_preOrder(Node *parent){
    if (parent != NULL){
        cout<<parent->key_value<<" ";
        DFS_preOrder(parent->left);
        DFS_preOrder(parent->right);
    }
}
```

Figure 2.1. PreOrder traversal

The logic of class DFS_inOrder is that the process will first traverse the left subtree, then visit the root, and traverse the right subtree.

```
void Binary_tree::DFS_inOrder(Node *parent){
    if (parent != NULL){
        DFS_inOrder(parent->left);
        cout<<parent->key_value<<" ";
        DFS_inOrder(parent->right);
    }
}
```

Figure 2.2. InOrder traversal

The logic of class DFS_postOrder is that the process will first traverse the left subtree, then traverse the right subtree, and visit the root.

```
void Binary_tree::DFS_postOrder(Node *parent){
    if (parent != NULL){
        DFS_postOrder(parent->left);
        DFS_postOrder(parent->right);
        cout<<parent->key_value<<" ";
    }
}
```

Figure 2.3. PostOrder traversal

An alternative implementation would be using a stack by creating an empty stack and pushing the root node (parent) onto the stack. Then, while the stack is not empty, the program will pop to the node on top of the stack and push all of its children onto the stack in reverse order.

```
void Binary_tree::DFS_stack(Node *parent){      //Depdth-first using a stack
    if (parent == NULL){
        return;
    }

    //Create an empty stack for DFS
    stack<Node *> s;
    //Push root and initialise height
    s.push(parent);

    while (s.empty() == false){
        Node *node = s.top();
        s.pop();
        cout<<node->key_value<<" ";

        //Push left tree
        if (node->left != NULL){
            s.push(node->left);
        }

        //Push right tree
        if (node->right != NULL){
            s.push(node->right);
        }
    }

}
```

Figure 2.4. DFS using a stack

Breadth-first traversal implementation

To implement BFS traversal, the program applied the algorithm which is creating an empty queue and push the root directory (parent) onto the queue. Then, while the queue is not empty, pop to the front directory (parent) off of the queue and push all of its sub-directories onto the queue.

```
void Binary_tree::BFS(Node *parent){            //Breadth-first using a queue
    if (parent == NULL){
        return;
    }
    //Create an empty queue for BFS
    queue<Node *> q;
    //Enqueue root and initialise height
    q.push(parent);

    while (q.empty() == false){
        //Print front of queue and remove it from queue
        Node *node = q.front();
        cout<<node->key_value<<" ";
        q.pop();

        //Enqueue left tree
        if (node->left != NULL){
            q.push(node->left);
        }

        //Enqueue right tree
        if (node->right != NULL){
            q.push(node->right);
        }
    }
}
```

Figure 3.1. BFS using queue

In main()

The code for the two above methods being called in the main would be shown in the Figure 4.1.

```
#include <iostream>
#include "binaryTree.h"
using namespace std;
int main(int argc, const char * argv[]) {
    Binary_tree *BST = new Binary_tree();
    BST->insert(1);
    BST->insert(2);
    BST->insert(7);
    BST->insert(3);
    BST->insert(10);
    BST->insert(5);
    BST->insert(8);

    cout<<"Pre Order DFS: ";
    BST->DFS_preOrder(BST->root);
    cout<<endl;

    cout<<"In Order DFS: ";
    BST->DFS_inOrder(BST->root);
    cout<<endl;

    cout<<"Post Order DFS: ";
    BST->DFS_postOrder(BST->root);
    cout<<endl;

    cout<<"DFS using stack: ";
    BST->DFS_stack(BST->root);
    cout<<endl;

    cout<<"BFS using queue: ";
    BST->BFS(BST->root);
    cout<<endl;
    return 0;
}
```

Figure 4.1. Calling methods in main()

Example explanation
In order to test if the program works properly or not, the author used a simple binary tree as shown below:

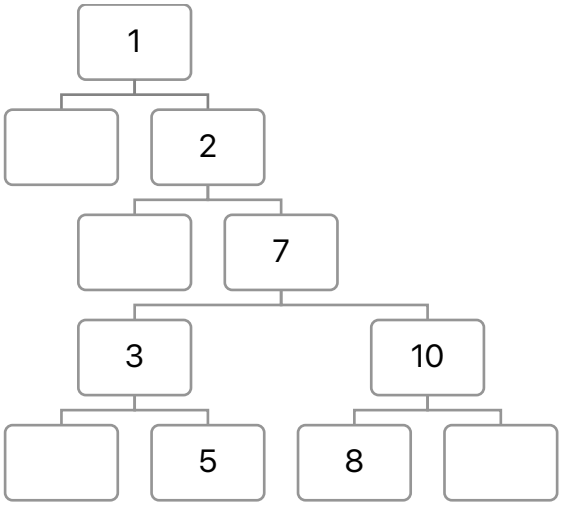

Figure 5.1. An example of binary tree

If the algorithm was implemented correctly, the program should show this result:

```
Pre Order DFS: 1 2 7 3 5 10 8
In Order DFS: 1 2 3 5 7 8 10
Post Order DFS: 5 3 8 10 7 2 1
DFS using stack: 1 2 7 10 8 3 5
BFS using queue: 1 2 7 3 10 5 8
Program ended with exit code: 0
```

Figure 5.2. Correct result of the below example

The node storing the 1, represented here merely as 1, is the root node, linking to the left and right child nodes, with the left node storing a lower value than the parent node, and the node on the right storing a greater value than the parent node.