Lab 6: BST and Stacks

The project **BST** contains **five (5) files**:

- Files **data_int_1.txt** and **data_int_2.txt**
    - Small list of integers to test the implementation.

- **Main.cpp**
    - **Function processTree:** Open the text files for reading and calls function insert of the **BST** class to insert data in the BST.
    - **Function testTree:** Tests the implementation of functions that traverse the tree and return the number of nodes in the tree.

- **BST.h** and **BST.cpp**
    - **Class Node:** We are including the **BST** class as a **friend class** to be able to access the **private** member variables directly (this will simplify the implementation when using pointers). The class creates objects that store an integer and two pointers, one to the left and one to the right.

    - **Class BST:** Creates an object that stores a pointer to the root of the **BST** and a count to keep track of how many nodes are in the tree. We will implement the following functions:
        - **Default Constructor**
            - It initializes the member variable of the class.
        - **Destructor**
            - Calls the function **destroy**.
        - Function **destroyTree**
            - Calls the function **destroy**. This function can be called anywhere outside the class.
        - Function **destroy**
            - This is a **recursive** and **private** function.
                - Why is it private? Because we have a parameter, the root, that it is a private member of the class; therefore, other classes should not have access to the root and should not call the root as a parameter.
            - **Parameter:** a pointer to the root of a tree (this might be a subtree) passed by reference.
                - Why are we passing by reference? Because we are deleting the pointer at each turn and we need to keep this information when backtracking.
            - This function allows us to use recursion to destroy all nodes in the tree. Because we need to re-send the root of the next subtree every time we recur, we need to pass a parameter (a pointer to the root of the subtree).
        - Function **recursiveInorder**
            - The function checks if the tree is empty and prints an error message. If the tree is not empty, it calls the function recursiveInorder(Node*);
        - Function **recursiveInorder(Node*)**
            - This is a **recursive** and **private** function (for the same reasons specified in the **destroy** function).
                - ➔ Pseudocode is on slide 38 of Lecture 7 (Trees).

- **NOTE:** the cout statement determines the order of the function
  - As a **first** statement → produces a **preorder** traversal
  - As a **last** statement → produces a **postorder** traversal
  - (This will be useful when testing)
- Function **totalNodes()**
  - This function returns the number of nodes in the BST.

Your job is to complete the implementation of the **BST class** with a mix of **recursive** and **non-recursive** functions as specified below.

- Function **insert()**
  - Inserts an item in the BST
  - **Parameters:** The item to insert
  - This function is **non-recursive**
  - Consider the case when the item to insert is already in the tree and output the **error** message, "The item to insert is already in the list. Duplicates are not allowed."

- Function **nonRecursiveInorder()**
  - This is a **non-recursive** function (as the identifier implies)
  - The recursive version of the traversal automatically does backtracking for you. Because this is a **non-recursive** function, instead of adding a third pointer to each node to go back up, it is easier to use a **stack**. In the **inorder** traversal of a binary tree, for each node, the left subtree is visited first, then the node, and then the right subtree. It follows that in an inorder traversal, the first node visited is the leftmost node of the binary tree. To get to the leftmost node of the binary tree, we start by traversing the binary tree at the root node and then follow the link of each node until the left link of a node becomes null. Because links go in only one direction, to get back to a node, we must save a pointer to the node before moving to the child node. Moreover, the nodes must be backtracked in the order they were traversed. It follows that while backtracking, the nodes must be visited in a last-in, first-out manner. This can be done by using a stack. Therefore, you will need to create a stack of pointers.
  - The general algorithm goes as follows:

```
Algorithm nonRecursiveInorder()
Set a pointer current to the root
While (current is not NULL or the stack is non-empty)
    If (current is not NULL)
        Push current into the stack      // you are visiting the node
        Move current to the left node
    Else
        Pop pointer from the stack and store it in current
        Output the data of current node
        Move current to the right node
```

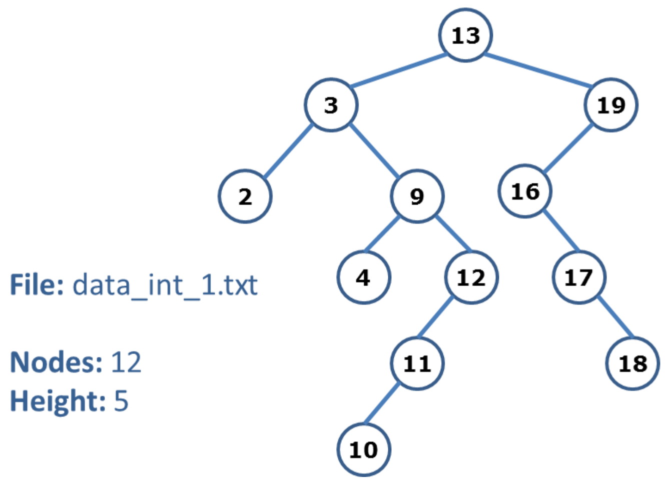
- Function **nonRecursivePreorder()**
    - This is a **non-recursive** function (as the identifier implies)
    - This is similar to the **inorder** traversal. You will need to modify to slightly modify it to print the **preorder** sequence.

- Function **nonRecursivePostorder()**
    - This is a **non-recursive** function (as the identifier implies)
    - This is slightly different from the other two traversals. In a postorder traversal of a binary tree, for each node, first the left subtree is visited, then the right subtree is visited, and then the node is visited. As in the case of an inorder traversal, in a postorder traversal, the first node visited is the leftmost node of the binary tree. Because for each node the left and right subtrees are visited before visiting the node, we must indicated to the node whether the left and right subtrees have been visited. After visiting the left subtree of a node and before visiting the node, we must visit its right subtree. Therefore, after returning from a left subtree. We must tell the node that the right subtree needs to be visited, and after visiting the right subtree we must tell the node that it can now be visited. To do this, other than saving a pointer to the node (to get back to the right subtree and to the node itself), we also save an integer value of 2 before moving to the left subtree and an integer value of 2 before moving to the right subtree (you will create an additional stack of type int). Whenever the stack is popped, the integer value associated with that pointer is popped as well. This integer value tells whether the left and right subtrees of a node have been visited.
    - The general algorithm goes as follows:

```
Algorithm nonRecursivePostorder()
Set a pointer current to the root
Set the value to 0
Push current and 1 into the stacks
Move current to the left child
While (neither stack is empty)
    If (current is not NULL and the value is 0)
        Push current and 1 into the stacks
        Move current to the left node
    Else
        Pop pointer and value from the stack and store them
        If (value is 1)
            Push current and 2 into the stacks
            Move current to the right child
            Set value to 0
        Else
            Visit current    // you are printing it
```

IMPORTANT: Do not simply translate the pseudocode to code, but try to understand what you are doing, because this is going to be part of the final exam.

Below you can find the visual representation of the BST after inserting the data from the **data_int_1.txt** file.



You **should** work with another student on this lab, and you are **expected** to consult other students for ideas and to check your code.

Turn in your **BST.cpp** file. If you have worked with another student, turn in **ONLY ONE COPY**.

- If you are NOT done by the end of class, name your file: **A200_TEMP_L6_Yourlastname_Yourfirstname_Otherstudentlastname_Otherstudentfirstname** The complete project is due **next week on Thursday, by 6:00 pm**.

- If you are done by the end of class, name your file: **A200_L6_ Yourlastname_Yourfirstname_Otherstudentlastname_Otherstudentfirstname**