CSE30: Lab #13 – Binary Trees

Overview

In this lab, we will explore a new useful data structure: **Binary Trees**. A binary tree is a very efficient data structure to store and retrieve **sorted data**. They are frequently used to implement maps and dictionaries (multimaps). In this lab, you will implement a simple form of binary trees.

Getting started

Create a new directory in your main development directory (probably on Desktop/CSE30) called Lab_13. Try to use the terminal on your own, without getting help from the TA to setup the new directories (try to learn/remember the terminal commands).

The g++ syntax to compile classes is slightly different than for a single program comprised of a main (and potential functions):

g++ *class1.h class1.cpp class2.h class2.cpp mainSource.cpp* -o *executable*

where:

- g++ is the compiler (installed on your Linux system) of C++ source files,
- ***mainSource.cpp*** is the source file for your main program (main function),
- ***class1.h*** is the class declaration for your 1st class,
- ***class1.cpp*** is your 1st class definition,
- ***class2.h*** is the class declaration for your 2nd class,
- ***class2.cpp*** is your 2nd class definition (and so on...),
- -o tells the compiler that you want to give the executable its own name, and
- ***executable*** is the name you want to give your program.

As an example, if we have a main source file called ***Exercise1.cpp***, the class declaration called ***LinkedList.h***, the class definition called ***LinkedList.cpp***, and want to create an executable called ***aTestProgram***, you would type:

g++ *LinkedList.h LinkedList.cpp Exercise1.cpp* -o *aTestProgram*

Assuming that your program compiled successfully (i.e. no errors were found), you can run your program as you normally would by typing "***./aTestProgram***" in the terminal/console.

Good coding practices (worth 2 points!)

Writing code that is understandable by humans is as important as being correct for compilers. Writing good code will help you complete the code, debug it and … get good grades. It is very important to learn as soon as possible, because bad habits are hard to get rid of and good habits become effortless. Someone (guess who) reads your code will be in a better mood if it is easy to understand … leading to better grades! This lab will include 2 points (10% for code quality):

- Explanations with comments
- Meaningful names
- Indenting of blocks { } and nesting …
- Proper use of spaces, parentheses, etc. to
- Visible, clear logic
- One / simple statements per line
- Anything that keeps your style consistent

(Exercise)

In this part of the lab, you will be implementing the basic functions for a Binary Tree, which are provided in the class declaration **BTree.h** (on UCMCROPS). In other words, you have to **create and implement the class implementation in a file called *BTree.cpp***. The main file you have to use for this lab is also provided on UCMCROPS (***Exercise.cpp***). ***DO NOT modify the class declaration (BTree.h) or main file (Exercise.cpp)***. Looking at the class declaration, you will find that a **Node** is defined as a structure comprised of a key value (***key_value***, of type **int**) and two pointers to the child nodes (***left*** and ***right***, of type **Node pointer**). You will also notice (under ***private***) that you will be keeping track of your Binary Tree using a Node pointer, ***root***. This ***root*** pointer should always point to the **root element** of your Binary Tree. If the Binary Tree is empty, the *root* pointer should point to **NULL**.

In this part of the lab, you will need to implement the following functions for the **BTree** class:

- **Default Constructor**: initializes the root, the binary tree is empty.
- **Destructor**: deletes the entire Binary Tree by calling ***destroy_tree()***.
- **BTree_root()**: returns a pointer pointing to the root of the Binary Tree.
- **destroy_tree(node *leaf)**: a recursive function that destroys a subtree with the input argument (*leaf*) as root. This function will check if *leaf* exists, then recursively destroy its left and right child nodes.
- **insert(int key, node *leaf)**: a recursive function that compares the input argument *key* with the *key_value* of the other input argument *leaf*. If *key* is less than *key_value*, the same function is called with the ***left child node of leaf*** as the new input argument;

otherwise, the ***right child node of leaf*** will be used as the new input argument. When the leaf node is empty (NULL), a new node is created and its *key_value* is set to *key* (remember to set its left and right child nodes to NULL).

- **search(int key, node *leaf)**: a recursive function that compares the input argument *key* with the *key_value* of the other input argument *leaf*. It returns the pointer to *leaf* if *key* = *key_value*. If *key* < *key_value*, the same function is called with the ***left child node of leaf*** as the new input argument; otherwise, the ***right child node of leaf*** will be used as the new input argument. It returns NULL if *leaf* is NULL (it reaches the end of the tree but the key is not found).
- **insert(int key)**: a public function that inserts a key into the tree. It creates a new node as root with *key_value* equals to *key* if the tree is empty, otherwise it calls ***insert(key, root)*** to insert the key at the correct location.
- **search(int key)**: a public function that starts the search of the input argument *key* from the root node. It returns the pointer to the node that has the same *key_value* as *key* (you do not need to perform any comparison in this function, only need to call ***search(int key, node *leaf)*** with the correct input arguments to start the search).
- **destroy_tree()**: a public function that calls ***destroy_tree(node *leaf)*** with the correct input argument to destroy the whole tree.

Note:

- a pointer does not point to anything unless you 1) use the reference operator (&) in front of a variable, 2) you dynamically allocate memory for it (using the **new** operator), or 3) you assign it (set its value to) to another pointer
- every time you want to create a node, you will have to use the **new** operator
- every time you allocate memory dynamically for a pointer using the **new** keyword, you have to make sure that you de-allocate the memory once you do not need the data anymore. This can be done using **delete** (in the ***destructor***, ***clear*** and ***remove*** functions).

Sample output from Exercise.cpp:

Root has key: 10
Left child of root has key: 6
Right child of root has key: 14

Searching for key 14...
Key 14 found!
Its left child has key: 11
Its left child has key: 18

Searching for key 13...
Key 13 not found!

What to hand in

When you are done with this lab assignment, you are ready to submit your work. Make sure you have included the following ***before*** you press Submit:

- A compressed file that includes **BTree.h**, **BTree.cpp**, **Exercise.cpp**, and a list of Collaborators.