Faculty of Arts & Sciences
Department of Computer Science
CMPS 270 – Software Construction
Fall 2022 – Assignment 2

Guidelines and Grading Policy

A program that does not compile will receive a **zero**. For every exercise, make sure you upload your solutions to your GitHub repository and maintain proper version control practices (commit your work regularly and meaningfully). You are also required to include the specifications of each method, in addition to the testing strategy and test-cases used to validate your program's correctness. The GitHub, specifications, and testing skills will account for 50% of the grade. The remaining 50% is for the program's correctness.

Exercise 1. Templated Binary Trees in C++ (call your program my_btree.cpp)

In this assignment you will implement a binary tree that can store characters (char) and integers (int). A node in the tree can be defined using the following:

```
template <typename T>
struct treeNode {
  T *data;
  struct treeNode * left;
  struct treeNode * right;
};
```

You will implement the following functions (that need to support both int and char nodes):

1. The - addNode - function which will add a node at the highest possible level in the tree. In other words, the add node should add the newly created node as a child of a node in the tree that does not already have two children and whose height (distance from the root of the tree) is the smallest. You can break ties arbitrarily but you should always the left child before the right child if both are null.
2. The - deleteNode - function will delete a node only if it has no children. The deleteNode function will accept either a char or an int, since you have to search for the node before deleting it. You can use is_same to figure our whether a node is holding a char or an int (http://www.cplusplus.com/reference/type_traits/is_same/). When a node cannot be delete the function should print "CANNOT DELETE NODE".
3. The - treeSize - function will return the number of nodes in the tree.
4. The - subtreeSize - function will accept an int or a char and will return the size of the tree rooted at that node (if the node is found) and 0 otherwise.
5. The - postOrderPrint - function will do a post order traversal of the tree starting at the root (and print the content of each node) (https://en.wikipedia.org/wiki/Tree_traversal)
6. The - preOrderPrint - function will do a pre order traversal of the tree starting at the root (and print the content of each node).

7. The - inOrderPrint - function will do a post order traversal of the tree starting at the root (and print the content of each node).

8. Test all your functions.

Exercise 2. Templated Stack (call your program my_stack.cpp)

A stack data structure stores a set of items, and allows accessing them via the following operations:

1. Push – add a new item to the stack
2. Pop – remove the most recently added item that is still in the stack (i.e. not yet been popped)
3. Top – Retrieve. For more explanation, see http://en.wikipedia.org/wiki/Stack(datastructure).

Using templates, implement a Stack class that can be used to store items of any type. You do not need to implement any constructors or destructors; the default constructor should be sufficient, and you will not need to use new. Your class should support the following 3 public functions (assuming T is the parameterized type which the Stack is storing):

1. bool Stack::empty() – returns whether the stack is empty
2. void Stack::push(const T &item) – adds item to the stack
3. T &Stack::top() – returns a reference to the most-recently-added item
4. void Stack::pop() – removes the most-recently-added item from the stack

Use an STL vector, deque, or list to implement your Stack. You may not use the STL stack class. Make the member functions const where appropriate, and add const/nonconst versions of each function for which it is appropriate. In the case where the Stack is empty, pop should do nothing, and top should behave exactly as normal (this will of course cause an error). When working with templated classes, you cannot separate the function implementations into a separate .cpp file; put all your code in the class definition. (Hint: You can represent pushing by adding to the end of your vector, and popping by removing the last element of the vector. This ensures that the popped item is always the most recently inserted one that has not yet been popped.)

Exercise 3. Operator Overloading (call your program op_overload.cpp)

Make a friend function that implements a + operator for Stacks. The behavior of the + operator should be such that when you write a + b, you get a new stack containing a's items followed by b's items (assuming a and b are both Stacks), in their original order. Thus, in the following example, the contents of c would be the same as if 1, 2, 3, and 4 had been pushed onto it in that order. (c.top() would therefore return the value 4.) Put your code for this exercise in the same file as for the previous exercise.

```
Stack <int > a, b;
a.push(1);
a.push(2);
b.push(3);
b.push(4);
Stack <int > c = a + b;
```

Exercise 4. Graphs. (call your program my_graph.cpp)

A graph is a mathematical data structure consisting of nodes and edges connecting them. To help you visualize it, you can think of a graph as a map of "cities" (nodes) and "roads" (edges) connecting them. In an directed graph, the direction of the edge matters – that is, an edge from A to B is not also an edge from B to A. You can read more at Wikipedia: http://en.wikipedia.org/wiki/Graph(mathematics).

One way to represent a graph is by assigning each node a unique ID number. Then, for each node ID n, you can store a list of node ID's to which n has an outgoing edge. This list is called an adjacency list. Write a Graph class that uses STL containers (vectors, maps, etc.) to represent a directed graph. Each node should be represented by a unique integer (an int). Provide the following member functions:

1. Graph::Graph(const vector &starts, const vector &ends)
   Constructs a Graph with the given set of edges, where starts and ends represent the ordered list of edges' start and endpoints. For instance, consider the following example:

   ```
   start: 0 0 0 4 4 3
   end:   1 2 3 3 1 1
   ```

   These vectors are used to initialize a Graph object representing a graph where there is a directed edge from 0 to 1, 2, and 3. Two directed edges from 4 to 3 and 4 to 1. And a directed edge from 3 to 1.
2. int Graph::numOutgoing(const int nodeID) const
   Returns the number of outgoing edges from nodeID – that is, edges with nodeID as the start point.
3. const vector< *int* > &Graph::adjacent(const int nodeID) const
   Returns a reference to the list of nodes to which nodeID has outgoing edges

Hint: Use the following data type to associate adjacency lists with node ID's: $map < int, vector < int >>$. This will also allow for easy lookup of the adjacency list for a given node ID. Note that the [] operator for maps inserts the item if it isn't already there.) The constructor should throw an invalid argument exception of the two vectors are not the same length. (This standard exception class is defined in header file stdexcept.) The other member functions should do likewise if the nodeID provided does not actually exist in the graph. (Hint: Use the map::find, documented at http://www.cplusplus.com/reference/stl/map/find/)

Exercise 5. Cycles in graphs (call your program graph_cycles.cpp)

Extend the class from the previous exercise by adding a function hasCycle that will check if the graph has a non-trivial cycle. That is, the graph has a cycle that contains at least 2 vertices. Your function should return the cycle.

In addition to your GitHub submission, add your programs to one folder: username.a02.zip (or .rar) and submit it to moodle.