CSCI 2270 Practice Midterm 2

Directions

Read these directions carefully.
You will be solving two of the three programming problems detailed below. *Problem 1* is **mandatory**, but you only have to do **either** *Problem 2* **or** *Problem 3*. For each problem you must submit three documents:

1. A C++ program that solves the given problem.
2. A text file that contains the output of your program when it is run.
3. A short document that contains any issues or concerns you have about the given problem, as well as any information that we need to understand, compile, or run your solution.

The text file you submit will help us understand your thought process. Mention anything you have done to write, test, and debug your code. Incomplete code can still receive points if you show that you have identified the errors and tried to debug them.

Your submission should be valid C++ 11 code. The solutions should use similar types and functions to those in the given starter code, but you are welcome to make modifications as you see fit. We will be running this code on other computers, so make sure to avoid **any** undefined behavior such as uninitialized variables.

Problem 1 (Mandatory)

Task:

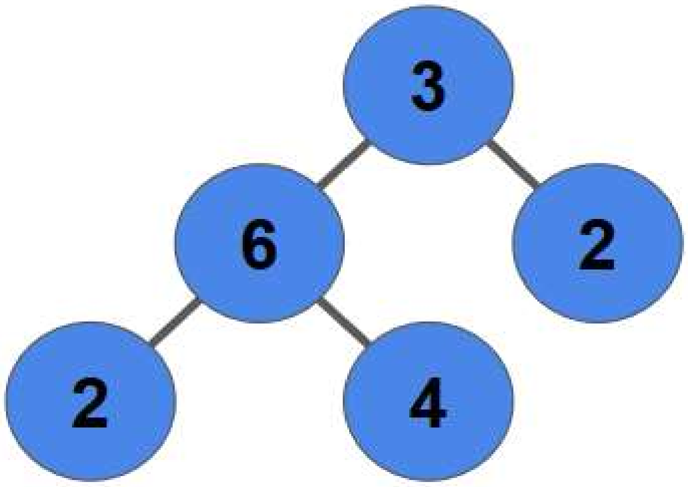
Write a program that creates a binary tree of integers, and a function that returns the product of all the nodes in the tree.

Requirements:

1. Implement a `calculateProduct` function:

   ```
   int BinaryTree::calculateProduct(); // example declaration
   ```

   This function should multiply together every node in the tree and return the result. If the tree is empty, it should return `0`.
   Examples:
   Given the following tree:

   - Calling `calculateProduct()` should return `288`.

2. Write a main function that creates a tree of integers, then calls the function defined in part (1) to determine the product of all nodes in the tree. Call it on multiple trees to show that it works in all cases, and print out the results.

3. **Test your function** to make sure that it works in every case, no matter how many nodes are in the tree.

Problem 2

Task:

Write a program that creates a hash table that uses chaining, as well as a function to change the number of hash slots in that table.

Requirements:

1. Implement a `resizeHashTable` function:

   ```
   void HashTable::resizeHashTable(newSize); // example declaration
   ```

   This function should resize the given hash table by changing the number of usable hash slots. The new table size will always be greater than zero. It should allow sizes that are both larger and smaller than the current size.

2. Write a main function that creates a hash table. You may use the hash algorithm from Assignment 7, or create your own, as long as it is relatively random. You should the function defined in part (1) to resize the table. Create tables of multiple sizes to show that it works in all cases, and print out the results.

3. **Test your function** to make sure that it works in every case, no matter how many elements are in the table or what the new size is.

Problem 3

Task:

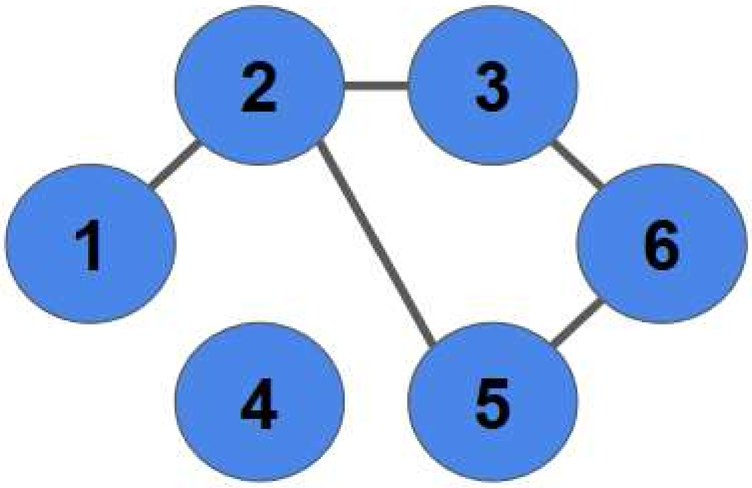
Create a graph of integers, and write a function that prints out a route between two points that passes through the smalles number of nodes.

Requirements:

1. Implement a `printShortestPath` function:

   ```
   void Graph::printShortestPath(int start, int end); // example declaration
   ```

   This function should print out the values of all the nodes along a path from *start* to *end* that passes through the fewest nodes. If multiple such paths exist, any of them are valid. If no such path exists, print a message to the user instead.
   Examples:
   Given the following graph:

   - Calling `printShortestPath(1, 6)` should print out `1 -> 2 -> 3 -> 6` or `1 -> 2 -> 5 -> 6`. Both are valid answers.
   - Calling `printShortestPath(2, 4)` should print a message to the user telling them that no such path exists.

2. Write a main function that creates a graph and calls the function defined in part (1) to find the shortest path between two nodes. Test this on different graph configurations to verify that it works on multiple graphs.

3. **Test your function** to make sure that it works in every case, no matter what the configuration of the graph is.