CSCI 2270 Accommodation 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 third document 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.
- Your solutions should use similar types and functions to the example code provided, but you are welcome to make modifications as you see fit.
- Show how you have tested your code. You will be graded higher for code with complete tests.
- 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 (not a search tree), and a function that can find the maximum value in that tree.

Note: You should search through the tree to find the maximum, rather than storing the maximum as nodes are added to the tree or using some other workaround.

Requirements:

1. Implement a `findMaximum` function:

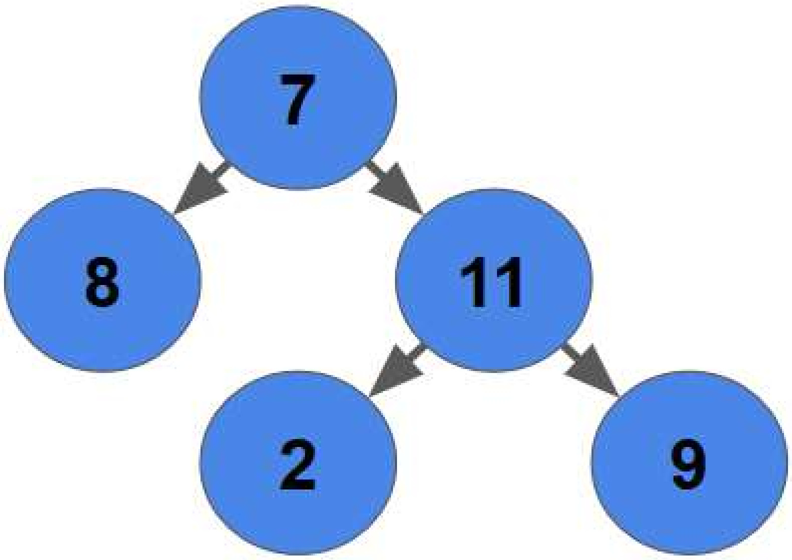
    ```
    int findMaximum(Node *n); // example declaration
    ```

    This function should find the maximum node in the tree. If the tree is empty, it should return `0`.

    Hint: This function will likely be recursive.

    Examples:

    - Calling `findMaximum(root)` on the following tree should return `11`:



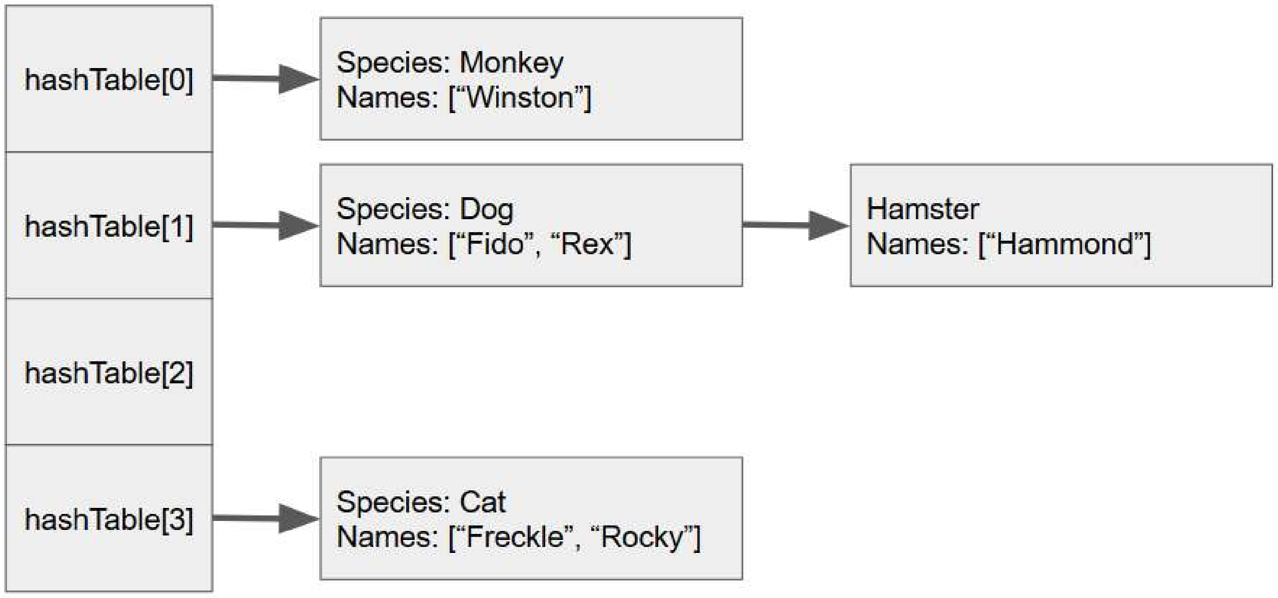
2. Write a main function that creates a tree of integers, then calls the function defined in part (1) to determine the maximum value in that tree. Call it on multiple trees to show that it works in all cases, and print out the results.

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

Problem 2

Task:

Write a program that creates a hash table that stores animals. Each animal has a name and a species, such as `Rocky` and `Cat`. The animals should be stored in the hash table by *species*, and each element in the hash table should store a list of all the names of animals with that species. If multiple animals of the same species are added to the table, all of the names should be added to a single list. An example of this is setup shown below, that uses a hash table size of 4 and collision resolution via chaining:



Requirements:

1. Implement an `insert` and `print` function:

   ```
   // example declarations
   void HashTable::insert(string name, string species);
   void HashTable::print(string species);
   ```

   The `insert` function should take a name and a species, and add that name to the list of animals of that species. If the species doesn't already exist in the table, it should add that species to the table.

   The `print` function should print out the names of *all* animals with the given species.

   Examples:

   If the hash table is in the same state as the example given in the task description above:

   - Calling `ht.print("Cat")` should print `Freckle, Rocky` .
   - Calling `ht.print("Hamster")` should print `Hammond` .
   - Calling `ht.print("Cow")` should print nothing.

2. Write a main function that creates a hash table. Call `insert` and `print` to add elements to that table and display them.

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. Now imagine that each node in the graph could be assigned a color: red or blue. Write a function that determines if it's possible to color every node in the graph either red or blue such that no two adjacent nodes have the same color.

Requirements:

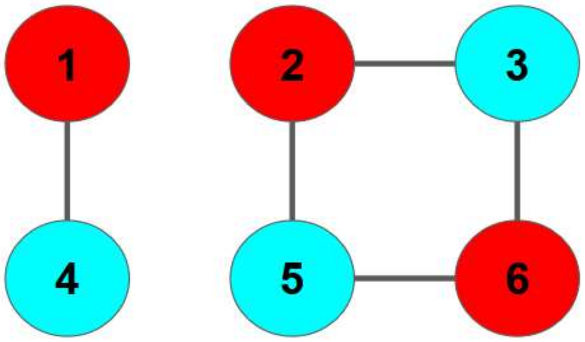
1. Implement a `isColorable` function:

   ```
   bool Graph::isColorable(); // example declaration
   ```

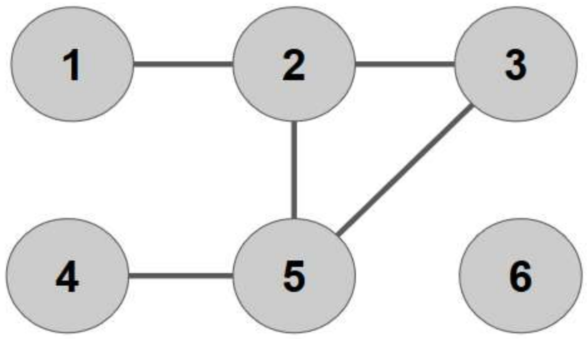
   This function should return `true` if each node in the graph can be colored either red or blue without any two adjacent nodes sharing a color.

   Examples:

   - Calling `graph.isColorable()` on the following graph should return `true` (colors added to show valid solution):

   - Calling `graph.isColorable()` on the following graph should return `false` , as no valid coloring is possible:

2. Write a main function that creates a graph and calls the function defined in part (1) to find out whether that graph is colorable or not. 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.