CS 15 Homework 3: Binary Search Trees

Introduction

In this homework you will implement a binary search tree (BST) that supports a multiset[1] (also known as a *bag*), which is a set that can contain duplicate values. The tree in this assignment will be almost the same as what you saw in class. The only difference is the way in which we handle duplicates. We'll describe this in more detail below, but the essense is that your BST will store integer *keys* (no values, you'll only store the keys) and track how many times any given key has been inserted into the collection. A key that has not been inserted appears 0 times, of course. For example, suppose we do this:

```
#include "BinarySearchTree"

int main() {
    BinarySearchTree multiset;

    multiset.insert(108);
    multiset.insert(-8);
    multiset.insert(108);
    multiset.insert(108);
    multiset.insert(0);
    multiset.remove(108);

    return 0;
}
```

After this code runs, `multiset` will contain 0 instances of 99, 1 instance each of -8 and 0, and 2 instances of 108 (because 108 was inserted 3 times

[1] see: `https://en.wikipedia.org/wiki/Multiset`

and removed once). If we removed 108 two more times, then the `multiset` would have 0 instances of 108.

Getting Started

Starter code is available by running the command

```
cp -r /comp/15m1/files/hw03/ .
```

We also provided a `Makefile`, so run `make` to compile the code. Execute it with the command `./hw03`. The `binary` folder contains the driver (discussed below) which has been compiled and linked with the reference executable. The folder contains different binaries which have been compiled on different operating systems; use the version that works on your system.

The Driver

You have been given an example driver for this program in `hw3.cpp`. In the driver, you are also given an extra function that can be used to print out your trees. However, the print function needs your `tree_height` function to work. Looking at the trees that print (after doing operations such as `insert`, `remove`, etc) can be very helpful, so we encourage you to implement `tree_height` as early as possible. **The driver does not represent complete testing. It's just a way for you to compare your implementation with the reference.** You will definitely want to write one or more testing programs for your code. A testing program will have its own main function and can link against your compiled `BinarySearchTree` implementation `.o` file.

The Assignment

Starter Code

For this assignment, we will provide:

- The `BinarySearchTree.h` header file
- A structured outline of the `BinarySearchTree.cpp` implementation file.

You are responsible for implementing most of the functions. We have thoroughly commented `BinarySearchTree.h`, so please refer to that file for specific information.

Recursive Requirement

The Binary Search Tree functions must all be implemented recursively to earn full credit. You may call a recursive helper function, but recursion should be involved in all of them.

Don't change public function declarations

You are not allowed to change the header or functionality of any of the public function declarations. However, you may add private helper functions you find helpful.

BinarySearchTree Specification

You are responsible for writing the following functions:

Public

Do not change the function signature of any public function

- Default Constructor
    - Initialize members of class to default values
- Assignment Operator (=) Overload
    - Check for self-assignment
        - only do something if not self assignment (so, if you do thisTree = thisTree the program should do nothing)
    - Delete any memory still used by the original tree
    - Perform a deep copy of the tree that's being copied
    - Once you are finished you have to `return *this` (which returns a reference to the tree object on the left of the assignment)
- `int find_min() const`

- Note: There are two `find_min` functions (one with and one without parameters). The public one is already implemented, you have to implement the private one listed below.

- `int find_max() const`
  - Returns the largest value stored in the tree. Returns the smallest possible integer if tree is empty.
  - You have to implement both the public and private versions of this function.

- `bool contains(int value) const`
  - As with `find_min` and `find_max` (see below) there are two `contains` functions (implement both)
  - The public function has one parameter (an `int`) and returns `true` if the tree contains the given value, `false` otherwise
  - The `public` function calls the recursive (`private`) function to do the search

The use of `const` here at the end of the function declarations means that the function will not modify the `BinarySearchTree` instance that the function is called on (the object pointed to by `this` when the function runs).

Private

Note: Many of these private functions have a public overloaded (another function with the same name) version that the client will call. Some of the public overloads have been implemented for you. **Each of the following functions must be implemented with a recursive algorithm.** You can make the function itself call a recursive helper function of your own design, but the solution must be done using recursion. We did not write the word "recursion" in all the specifications, but every function here must use a recursive process as its solution.

- `void post_order_delete(Node *node)`
  - **Recursively** deletes all `Node`s in the tree in a post-order fashion.

- `Node *pre_order_copy(Node *node) const`
  - **Recursively** copies all `Node`s in the tree in a pre-order fashion.

- This is a **deep copy**: if the provided node has children, we copy it and then (recursively) copy the children `Node`s and then their children, and so on.

- `bool contains(Node *node, int value) const`
  - Return `true` if the tree contains the given value, `false` otherwise.
  - Use the BST invariants to make your code as efficient as possible.

- `void insert(Node *node, Node *parent, int value)`
  - Insert must preserve the BST invariants. That is, the tree should still be a BST after insertion.
    - If you insert a number that is not in the tree, you must create a new `Node` with count equal to 1.
    - If we are inserting a value that is already present in the tree, we increase the count in that `Node`.
  - You may change the parameters of this function (the `private` insert) if it will make your code cleaner, or easier to understand. If you do change it, then you will need to change the call to it from inside of the `public` function, too. Do **not**, however, change the `public` function declaration.

- `bool remove(Node *node, Node *parent, int value)`
  - Needs to recursively search for the `Node` to delete.
  - If the element is not in the tree you don't need to do anything to the tree.
  - If the element is in the tree you should decrease its count.
  - If node is the element to remove, and its count has dropped down to zero you have to remove it from the tree.
    - When removing a `Node` that has two children you theoretically have two options to replace the value in that `Node`:
      - use the value of the `Node` with the largest value in the left subtree
      - use the value of the `Node` with the smallest value in the right subtree

      Even though both are generally okay, our reference implementation uses the **smallest value in the right subtree**. Please make sure to do the same or discrepancies between the reference solution and yours will appear.

- Returns `true` if an element was removed from the tree, `false` otherwise.
- Before attempting to implement this function (the private remove), we **strongly** recommend that you answer all of the `README` Questions.
- You may change the parameters of this function (the `private` remove) if it'll make your code cleaner, or easier to understand. If you do change it, then you will need to change the call to it from inside of the `public` function, too. Do **not**, however, change the `public` function declaration.

- `int tree_height(Node *node) const`
  - The height of a `Node` is the number of hops you need to do to reach a leaf (the length of the longest path to a leaf from that `Node`). The height of a tree is the height of the `root`.
  - An empty tree (no `root`) is considered to have a height of -1.
  - A tree with just a single `Node` has a height of 0.

- `int node_count(Node *node) const`
  - This function returns the number of `Node`s, which represents the number of distinct values in the tree.
  - An empty tree has 0 `Node`s, a tree with a single vertex has 1 `Node`, and so on
  - The `count` field in the `Node` struct is ignored for this function. So a single number that has been inserted 10 times still counts as a single `Node`.

- `int count_total(Node *node) const`
  - Rather than simply counting the number of `Node`s, you have to sum all values that have been inserted into the tree.
  - Be sure to use the `count` field! If we insert 3, 4, and 3 again in the tree, `count_total` should return 10.

- `Node *find_min(Node *node) const`
  - Returns the address of the `Node` with the smallest data value.
  - You'll recurse through tree to find `Node` with smallest value and return address of that `Node`.

- **Remember the BST invariants!**

- `Node *find_max(Node *node) const`
  - Returns the address of the `Node` with the largest data value.
  - You'll recurse through tree to find `Node` with largest value and return address of that `Node`.
  - **Remember the BST invariants!**

README Outline and Questions

> Note the specific questions below.
> You MUST include answers in your `README`

With your code files you will also submit a `README` file. The file is named `README`. There is no `.text` or any other suffix. The contents is in plain text with lines less than 80 columns wide. You can format your `README` however you like, but it should be well-organized and readable. Include the following sections:

A The title of the homework and the author's name (you)

B The purpose of the program

C Acknowledgements for any help you received

D For 2 points on the final assignment grade: How much time did you spend on this homework in hours? (Your data will be anonymized, aggregated with other students' answers, and shared with the department — anonymously.)

E The files that you provided and a short description of what each file is and its purpose

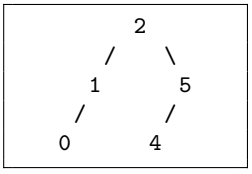
F How to compile and run your program

G Details and an explanation of how you tested the various parts of assignment and the program as a whole. You may reference the testing files that you submitted to aid in your explanation.

Each of the sections should be clearly delineated and begin with a section heading describing the content of the section. You should not only have the section letter used in the outline above. In addition, answer the following questions in your `README`:

1. Review the remove function explanation in the homework specification. Will your implementation of the remove function use (the privately defined) `find_min()` or `find_max()`? Why?
2. Is it possible for (the privately defined) `find_min()` or `find_max()` to return a value that does not point to a valid node? Why or why not?
3. a If you answered yes to 2, then what value is returned? In what case will that value be returned?

   b If you answered no to Q2, then then consider the tree below and specify the node returned when `find_min()` is invoked on the right child of the node with value of 5:



4. Write pseudocode for your private `find_min()` function.
5. Write pseudocode for your private `find_max()` function.
6. Write pseudocode for your private `post_order_delete()` function.

Implementation Advice

- Do **NOT** implement everything at once!
- Do **NOT** write code immediately.
- Before writing code for any function, draw before and after pictures and write, in English, an algorithm for the function. Only after you have tried this on several examples should you think about coding.
- Remember the *write a little, compile a little, test a little* mantra. This will save hours of headaches down the road.

- Keep all the functions you make to test your program in `main`. You need to submit the tests together with your code. Try to group them in a nice way (this will be further discussed in lecture). We want to see, and will evaluate, your test code, so don't delete it!
- If you need help, TAs will ask about your testing plan and ask to see what tests you have written. They will likely ask you to comment out the most recent (failing) tests and ask you to demonstrate your previous tests.
- Be sure your files have header comments, and that those header comments include your name, the assignment, the date, the file's purpose, and acknowledgements for any help you received.

Submitting

Be sure your files have header comments, and that those header comments include your name, the assignment, the date, and acknowledgements for any help you received. Please submit this assignment in the method discussed in class. Don't forget to submit your testing programs, too! Since we didn't assign any file name(s) for these, we cannot give you the provide command.

Assessment

Your work will be assessed in the following manner:

- **Code**
  Did you implement all functions correctly? Graders will read your code and look for key elements of correctness, including whether you met the specification above.
- **Automatic testing**
  We will compile your `BinarySearchTree` class to produce a `BinarySearchTree.o`, which we will then link our own `main` program (not yours). Our `main` program will do a battery of tests like `the_hw3`'s, and the output of this program, running with your class implementation, will be compared against expected output (in which the tests all pass). You can (and should) compare your own output to the reference implementation's results on the driver file to be sure you don't fail tests due to simple mistakes (like incorrectly formatted output).

- **Coding style and readability**
  This includes header comments, function contracts, proper indentation, abiding by the 80 column limit, keeping function under 30 lines, putting spaces around binary operators and after comments, etc. It also includes choices of variable names, appropriate internal documentation, and clarity. Each public function will invoke its corresponding private function, but that is an implementation detail and should not be mentioned in the purpose field of function contracts.

- **Compilation warnings**
  Programs you submit should compile and link with no errors or warnings. Most warnings are actual errors in your program: **pay attention to them!**

- **Memory errors and leaks**
  Memory errors are when your program uses memory incorrectly (e. g., by an out of bounds array reference). These are errors in your program, even if those errors don't show up in the tests. We will deduct for these. Your program should also not leak memory, i. e., your program should recycle (delete) all the memory it allocates on the heap (with new) by the time the program terminates. We recommend you learn to use `valgrind`, a program that can detect any memory errors and leaks — we use `valgrind` to grade this part of the assignment. If `valgrind` reports any memory errors and/or leaks, points will be deducted.

- **Testing**
  We will give you feedback on your testing strategies as described in your `README` and as exemplified in your testing files.