# CS 15: Homework 03: 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 essence 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 no instances of 99, one instance each of -8 and 0, and two instances of 108 (because 108 was inserted 3 times and removed once). If we removed 108 two more times, then the `multiset` would have no instances of 108 anymore.

## Getting Started

Starter code is available by running the following command on the homework server:

`/comp/15m1/files/hw03/setup`

We also provide a `Makefile`, and `make` compiles the code. Execute it with command `./hw3`

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

## The Driver

You have been given a driver for this program in `hw3.cpp`. **Do not modify this driver!** In addition to the driver, you are 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, each with their own `main` function as needed.

## The Assignment

For this assignment, we will provide the `BinarySearchTree` header file (`BinarySearchTree`.h) and a structured outline of the `BinarySearchTree` functions (`BinarySearchTree.cpp`). You are responsible for implementing most of the functions. We have thoroughly commented `BinarySearchTree.h`, so please refer to that file for specific information.

**Two important notes:**

- To receive full credit, the BST functionality must all be implemented using recursion. In order to implement one of these pieces of functionality, you may write non-recursive `public` wrapper functions that themselves simply call recursive `private` helper functions, but recursion should eventually be used to implement tree functionality.
- You should not change the header of any of the `public` function declarations already given to you. In addition, do not change the intended functionality of `private` functions we ask you to write—for example, if such a function has a `const` indicator, you should not change that. However, you may change the *inputs* used by `private` functions if that is convenient, and you may of course add any other `private` helper functions you find useful.

## BinarySearchTree Specification

You are responsible for writing the following functions (others are given to you):

### public functions

**Reminder:** *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 a 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 actually already implemented, and you just need 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 the tree is empty.
  - You have to implement both the `public` and `private` versions of this function.
- `bool contains(int value) const`
  - As with `find_max`, 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, and `false` otherwise.
  - The `public` function calls the recursive (`private`) function to do the search.

**Note**: 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). Indicating this explicitly allows the compiler to make sure we do not make such modifications by mistake.

### private functions

**Note**: Many of these private functions have an overloaded (another function with the same name) `public` version that client code can call; some of those `public` functions may be recursive, and some may not. Each of the following `private` functions *must* be implemented with a recursive algorithm, however. You can make the function itself call a different, recursive helper function of your own design, but the solution must be done using recursion. We did not write the word "recursion" or "recursively" in all the specifications, but we are telling you this here: every function here must use a recursive process as its solution.

- `void post_order_delete(Node *node)`
  - Recursively deletes all `Node` objects in the tree in a post-order fashion.
- `Node *pre_order_copy(Node *node) const`
  - Recursively copies all `Node` objects 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, then their children, and so on.
- `bool contains(Node *node, int value) const`
  - Return `true` if the tree contains the given `value`, `false` otherwise.
  - This should be as efficient as possible, not searching parts of the tree that it does not need to search.
- `void insert(Node *node, Node *parent, int value)`
  - This must preserve the BST *invariants*. That is, the tree should still be a BST after insertion is complete.
    - If we insert a number that is not already in the tree, a new `Node` with count equal to 1 must be created.
    - If we insert a value that is already present in the tree, the count in the `Node` storing that value must increase by 1.
  - You *may* change the parameters of this function 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)`
  - This must search for the `Node` storing `value` recursively.
  - If `value` is not in the tree, nothing should happen to the tree.
  - If `value` is in the tree, the count for the `Node` containing it should decrease by 1.
  - If the count of some `Node` reaches 0, that `Node` should be removed from the tree.
    - When removing a `Node` that has two children you have two options of how to replace the value in that `Node`, each of which would properly maintain the BST structure:
      1. Use the *largest* value in the *left* subtree.
      2. Use the *smallest* value in the *right* subtree.

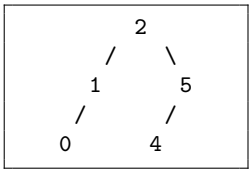
      Even though both are generally okay, our reference implementation uses the *smallest value in the right subtree*. Please make sure to do the same or discrepancies will appear during testing.
  - Returns `true` if an element was removed from the tree, `false` otherwise.
  - Before attempting to implement this function (the private remove), we ***strongly*** recommend that you do the preparation outlined in the section on **Things to Think About and Do**, on page 6.
  - You *may* change the parameters of this function 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.
- `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 overall height of a tree is the height of the `root`.
  - An empty tree is considered to have a height of $-1$.
  - A tree with just a `root` has a height of 0.
- `int node_count(Node *node) const`
  - This function returns the number of `Node` objects, which represents the number of distinct values in the tree. An empty tree has 0 `Node` objects, a tree with a single `root` has 1, etc.
  - The `count` field in the `Node struct` is ignored for this function. So a number that has been inserted 10 times still counts as a single `Node`, for example.

- `int count_total(Node *node) const`
  - Rather than simply counting the number of `Node`s, this function sums all values that have been inserted into the tree. For example, if we insert 3, 4, and then 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.
  - Should be implemented recursively, and efficiently, avoiding any searching down branches that do not need to be searched.
- `Node *find_max(Node *node) const`
  - Returns the address of the `Node` with the largest data value.
  - Should be implemented recursively, and efficiently, avoiding any searching down branches that do not need to be searched.

## Things to Think About and Do when Designing a Solution

Each of the following steps are recommended ***before*** writing your code:

1. Review the explanation of the `remove` in the homework specification. Will your implementation of this function need to use (the privately defined) `find_min()` or `find_max()`? Why or why not?
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. Based upon your answer to the prior question:
   a. If you answered yes, then what value would be returned, exactly? In what case will that value be returned?
   b. If you answered no, then consider the tree below. Which node is 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.

## README Outline and Questions

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 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 above.)

## Implementation Advice

- Do not try to 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 code you write with its own `main` function, allowing you to build and run tests without editing the `hw3.cpp` vile. You can also submit the tests together with your code. Try to group them in a nice way. We want to see, and will evaluate, your test code, so don't delete it!
- 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.

## Reference Implementation

We have provided a reference implementation called `the_hw3`. We encourage you to execute it and compare the result of your program with ours.

## Submitting

Submit your code on Gradescope, along with your README. You can include any testing code you want to show us (written into its own separate files).