CSC 211 Assignment 5

Binary Search Trees

Due Monday, May 1st by 11PM

Background

Earlier in the semester, we looked at **binary search**, which requires some sort of specialized data structure: either a completely-sorted list of entries to be searched (such as a phonebook, or the result of your quicksort from Assignment 4) or a **binary search tree**.

For this assignment, you have been given the interface for a templated binary search tree class, and you must write the implementation. A binary search tree consists of a **root** node, which can have two children, a *left* and a *right* child. By convention, a binary search tree will uphold the **invariant** property that, for any node, all descendents of that node that are **less** than it will appear in its **left** branch, while all descendents that are **greater** than it will appear in its **right** branch. Thus, *binary search* can use this property to conduct search quickly (specifically, in $O(\log n)$ time).

Algorithmically, this should be easier than Assignment 4; the new challenge will be dealing with **pointers** and writing code that follows chains of pointers.

You will also be writing a **templated class**. In theory, it should support any type.

In practice, I will test your templated `BST` class by creating binary search trees containing the following data types:

- `int`
- `double`
- `std::string`
- `DNA`

This, coupled with some current C++ limitations, is why the following appears at the bottom of `bst.cpp`:

```
template class BST<int>;
template class BST<double>;
```

```
template class BST<std::string>;
template class BST<DNA>;
```

In order to support other types, you would need to add `template class` statements for those types.

Getting Started

- Get into your Docker development environment as usual (**remember to `cd` into the `data` directory**).
- Download the assignment framework with `git clone https://github.com/csc211/a5`
- Look in the resulting directory `a5`.
- You'll see some familiar files: a working `dna.cpp` and its header file, as well as a `bst.cpp` (which isn't yet complete), a `bst.h`, and a `main.cpp`.
- Note: `./compile` will not work yet.
  - You need to fill in the function definitions in `bst.cpp`
  - I have provided you with correct implementations of the default constructor and the **destructor**, which you don't need to worry about.
- Once you can compile, you'll produce an executable binary called `bsttest`, so you can run it as `./bsttest` (with whatever arguments you implement.)

Testing

- Unlike Assignment 4, I've written a **very limited** `main.cpp`. You should read it and be sure that you understand what it does.
- Once you get your code to compile, you can try running `bsttest` based on the `main.cpp` I provided.
  - A correct solution **will** print out the number "1".
  - However, an **incorrect** solution might also print out the number "1".
  - This is because the `main.cpp` I provided you is not an exhaustive test.
  - It's just meant to give you an idea how to use the `BST` class.
- Write a more comprehensive `main.cpp` to test that your `BST` class works properly with multiple data types.

Requirements

Other than the default constructor and the destructor, every method promised by the interface `bst.h` is missing. You must fill them all in.

Some methods take a **comparator** function. This is just like in Assignment 4: a function that returns a `bool` based on two items, in this case of using templated type variables.

This is your first time writing a templated class, so pay attention to the definitions of the two provided methods already present in `bst.cpp`.

Above all, **read the method comments in the interface file**. They explain how each method should work.

Helpful hints

You will often finding yourself checking whether or not a pointer *points* to anything useful. The canonical way to do this is to check whether the pointer is equal to `nullptr`.

Remember that the `find` method is supposed to `throw std::runtime_error` if it can't find what it's looking for.

The method comments in the interface file give some useful hints. In particular, give some thought to the comments in `find()` and `insert()`.

For your own sake, write your own tests and run them before you try submitting to gradescope!

If you are iterating on Gradescope before you have passed some sensible tests of your own, you are **wasting time**. Gradescope takes **minutes** to give you an answer, and it's not always informative, and you can't debug it. Your own test cases take **milliseconds** to give you an answer, and you can debug it!

Compile early and often!

The compiler is your friend, not your enemy. Compile after you write a little bit of code; it's far easier to find a problem if you know it's in the last few lines of code you wrote.

Grading Rubric

For this assignment, correctly passing all tests on Gradescope is worth 70% of your grade.

Another 10% is based on **how many data types** your `main.cpp` tested, out of the four required. In other words, you can write a perfect implementation, document and design it perfectly, and still receive only 90% because you didn't adequately test any of them (note: my `main.cpp` as provided is not adequate testing of even the `int` data type).

As usual, 20% of your grade will be based on design and representation. Indentation issues due to gradescope will not be penalized, but we still expect proper comments and **consistent** style.

Submitting

You will submit **two files**: `bst.cpp` **and** `main.cpp` via Gradescope, where the functional correctness of your `bst.cpp` will be graded automatically. You can submit as many times as you like; only the last submission will count towards your grade.

Lateness

Submissions will be accepted late with a 10% penalty per day, up to two days late.

Contents of bst.h

```
#include <vector>

#ifndef _bst_h
#define _bst_h

template <typename T> class BST {
public:

    /*
    Default constructor.
    */
    BST();

    /*
    Destructor.
    Traverses any subtrees.
    */
    ~BST();

    /*
    Single-instance constructor.
    Given some content of type T,
    creates a new BST node with its content field populated.
    Sets left and right children to nullptr.
```

```
    */
    BST(T content);

    /*
    Vector constructor.
    Given a vector<T> and a comparator function,
    inserts every element of the vector into a BST.
    The comparator is used to determine a "less-than" relationship.
    The BST object returned by this constructor is the
    root of the binary search tree.
    */
    BST(std::vector<T> contents, bool comparator(T&, T&));

    /*
    getter for the content field
    */
    T getContent();

    /*
    Performs a binary search on the BST, given a comparator function.
    This uses a notion of equality based on the "less-than" comparator:
    (a == b) if and only if !(a<b) && !(b<a)
    */
    T find(T query, bool comparator(T&, T&));

    /*
    Inserts element x, of type T, into the BST.
    This preserves the BST invariant:
    a left child is always less than its parent, while
    a right child is always not less than its parent.
    This relies on the "less-than" comparator.
    */
    void insert(T x, bool comparator(T&, T&));



private:
    T content;
    BST<T> *left, *right;
};


#endif
```

Contents of bst.cpp

```
#include "bst.h"
#include <string>
#include "dna.h"
#include <stdexcept>
#include <iostream>

template <typename T>
BST<T>::BST() {
    /*
    slightly ugly: because we don't have a parameter of type T,
    we have to call new and dereference the pointer.
    Other constructors should NOT do this.
    */
    this->content = *(new T);
    this->left = nullptr;
    this->right = nullptr;
}

template <typename T>
BST<T>::~BST() {
    // Recursively call delete on left and right subtrees.
    if (this->left != nullptr) delete(this->left);
    if (this->right != nullptr) delete(this->right);
}

/*
FILL IN THE MISSING METHODS!
*/

/*
Instantiate this template class for four types.
This is to allow the separation between interface and implementation.
*/
template class BST<int>;
template class BST<double>;
template class BST<std::string>;
template class BST<DNA>;
```