CS 271 (Fall 2024) Project 2 – Trees, trees, and more trees!

Instructor: Prof. Michael Chavrimootoo
Due: Thursday, November 7, 2024, 11:59PM

1 Project Overview

In this project, you will implement a binary search tree (BST) class and a red-black tree (RBT) class. Your implementations will be useful to you in Project 3. By the end of this project, you will have a better understanding of the implementation of BSTs and RBTs.

1.1 Learning Goals

The goal of this project is to get you to be more familiar with BSTs and RBTs, and understand the details of the implementations of these trees.

Tip 1 *The coding parts of this project are significantly more challenging than they have been in the prior projects, so I highly recommend getting the code for BSTs working first as early as possible—you should be able to code it all up this week itself—before doing the RBT implementations. More details on that in Section 3.*

1.2 Use of Other Work

While you may consult any code you have personally written for this class, and any code in the course textbook, you are not allowed to use any other sources, which includes and is not limited to, your peers outside your group, other textbooks, the internet, and AI tools (such as Gemini and ChatGPT).

1.3 Group Work

You will be completing this project as part of your current groups. To ensure that everyone is contributing meaningfully to this project, you should each keep track of the contributions of each member's contributions. You will need to describe your contributions (see Section 2).

I expect everyone to contribute meaningfully to **both** the coding and non-coding parts of the project.

Tip 2 *If a member of your group is not working as fast as you would like, the right thing to do would be to help them complete their task (e.g., by explaining things that are not making sense to them) without doing the task for them. That may take longer, but doing their part of the project for them would be a disservice to the both of you.*

1.4 Submission

There are two submissions to be made with this project. Both are due by Thursday, November 7, 2024, 11:59PM.

1.4.1 The First Submission

You should submit a ZIP file containing all your code, and unzipping your ZIP file should yield a directory called `GroupX`, where `X` is your group number, and your project files should be in that directory (not in a subdirectory).

You will submit exactly four files (within your zipped directory):

1. `BSTNode.hpp` – a header file for the `BSTNode` class.
2. `BSTNode.cpp` – implementations for the `BSTNode` class.
3. `BST.hpp` – a header file for the `BST` class.
4. `BST.cpp` – implementations for the `BST` class.
5. `RBTreeNode.hpp` – a header file for the `RBTreeNode` class.
6. `RBTreeNode.cpp` – implementations for the `RBTreeNode` class.
7. `RBTree.hpp` – a header file for the `RBTree` class.
8. `RBTree.cpp` – implementations for the `RBTree` class.
9. `mytests.cpp` – a driver for your test cases.
10. `customexceptions.hpp` – declarations and implementations of your custom exceptions.
11. `Makefile` – the makefile.

Each group will make *one* submission via Canvas, i.e., only one person in each group will make a submission. This item is called "Project 2 - Code Submission" in Canvas.

1.4.2 The Second Submission

Each student must submit a file called `lastname_firstname_README.pdf` (obviously replacing `lastname` and `firstname` with your own last and first names, respectively). Each student must make their own submission. And each student must write their report independently, i.e., you cannot collaborate on your reports (see more in Section 2). This item is called "Project 2 - Individual Reports" in Canvas. More details on this document are given in Section 2.

1.5 Submission Structure

Include a driver file called `mytests.cpp` that runs your various tests. Each test should be contained within a single function, and each such function should be thoroughly commented, describing the purpose of the test and what it demonstrates.

After running your tests, your driver should output how many tests passed and how many tests failed as the last line of its output.

You must include a Makefile that compiles your executables in the following way:

1. `make mytests` should compile and run your tests that demonstrate the correctness of your code.

Your tests should be comprehensive and extensive, while being of reasonable size. Nonetheless, I will also have my own tests that your code will be tested against.

If your code fails to run or to compile, you will automatically receive a zero for this component of the project.

1.6 Documentation

You must document your code properly. This includes:

1. A full comment header at the top of each source file containing the file name, the developer names, the creation date, and also a brief description of the file's contents.

2. Each method should have a full comment block including a description of what the method does, a "Parameter" section that lists explicit pre-conditions on the inputs to the function, and a "Return" section that explicitly describes the return value and/or any side effects.

3. Make careful use of inline comments to explain various parts of the code that may not be obvious from the code syntax.

2 The README

Each student must submit a README file, which is an essential component of this project, and it is yet another way in which you communicate your technical contributions as a group. A README is a common file included in programs and "code repositories" that explain to a user or to another developer what the directory contains, the purpose of each file in the directory, how the program works, how to run the program, etc. An individual who has never seen your project's description should be able to understand what the project is about and how your solution works solely by reading the README and your comments.

Your README should be organized in the following (brief) sections:

- Project Overview - Give a brief description of your project and the list of contributors.

- Project Structure - Give a description of your submission directory, i.e., the list of files/directories in your project and a brief description of each.

- Major Design Decisions - Discuss any major design decision made by your group (if any); e.g., if you chose to implement additional functions, or use functional abstractions not described in the project, then you should explain what you did and why you did it.
- Group Challenges - Discuss any major problem your group encountered while implementing the project.
- Individual Reflection - Describe your individual contributions to the project and briefly describe any challenges you (not your group) faced in this project.
- Known Issues - If there are any known issues with your code, mention those here. For each issue, try to best explain what may be causing the issue and what would be a potential solution; by thinking through the cause of the issue, you may find a way to fix it :)
- Additional Information - Any additional information you find relevant; keep it brief.

While you cannot collaborate to *write* your READMEs, you are certainly free to discuss elements of the README with your group, the TA, and me. However, you cannot produce any written material or recordings during those discussions.

Your submission should be a single PDF file.

Be careful: When writing your README, you may not use text from the project description or other sources verbatim. You can only use your own words. If there is evidence that you received unauthorized help in your reports, you will receive a zero for that part of the assignment and a formal academic integrity violation report will be filed.

3 Design Requirements

As mentioned earlier, the coding parts for this project are significantly more demanding, and so it has been broken down into two components. You may use your own, personally written notes from class or the textbook to guide you through the implementations, and you cannot use any other resources. You can find pseudocode for all of the following methods in the textbook.

In both parts, make sure you are appropriately handling allocation and deallocation. Naturally, you should implement constructors (default and copy), assignment operators, and destructors, and implement all the additional methods listed below. You may implement additional methods as you see fit if they are needed to develop your code.

For each class `X` that you implement, you must provide your declaration in `X.hpp` and the implementations in `X.cpp`. You can store all your custom exceptions in `customexceptions.hpp`.

3.1 Part I – The `BSTNode` and `BST` Classes

You will implement two classes in this part: `BSTNode` and `BST`.

Details on the additional methods for `BSTNode.cpp`:

1. `BSTNode<T>* treeMin()`
   Returns a pointer to a node with minimal value in the tree rooted at `this`.

2. `BSTNode<T>* treeMax()`
   Returns a pointer to a node with maximal value in the tree rooted at `this`.

3. `void printPreOrderTraversal() const`
   Prints to `stdout` the preorder traversal of the tree rooted at `this`.

4. `void printInOrderTraversal() const`
   Prints to `stdout` the inorder traversal of the tree rooted at `this`.

5. `void printPostOrderTraversal() const`
   Prints to `stdout` the postorder traversal of the tree rooted at `this`.

Details on the additional methods for `BST.cpp`:

1. `void transplant(BSTNode<T> *oldNode, BSTNode<T> *newNode)`
   Implement the `TRANSPLANT` method from the textbook.

2. `bool isEmpty() const`
   Determine if the tree is empty in constant time.

3. `long size() const`
   Determine the number of nodes in the tree in constant time.

4. `BSTNode<T>* insert(T value)`
   Inserts a node with value `value` into the tree and returns a pointer to the inserted node.

5. `void remove(T value)`
   Removes a node with value `value` from the tree. If no such node exists, your code must throw a `value_not_in_tree_exception` exception (which you must define). If the tree is empty, your code must throw a `empty_tree_exception` exception (which you must define). Use the method described in the textbook to preserve the integrity of all pointers using your code.

6. `BSTNode<T>* search(T value) const`
   Returns a pointer to a node with value `value`.

7. `BSTNode<T>* treeMin() const`
   Returns a pointer to a node with minimal value in the tree. If the tree is empty, your code must throw a `empty_tree_exception` exception (which you must define).

8. `BSTNode<T>* treeMax() const`
   Returns a pointer to a node with maximal value in the tree. If the tree is empty, your code must throw a `empty_tree_exception` exception (which you must define).

9. `void printPreOrderTraversal() const`
   Prints to `stdout` the preorder traversal of the tree.

10. `void printInOrderTraversal() const`
    Prints to `stdout` the inorder traversal of the tree.

11. `void printPostOrderTraversal() const`
    Prints to `stdout` the postorder traversal of the tree.

Tip 3 *I highly recommend you implement the iterative versions of the search, remove, and insert methods as those are vastly more efficient, and understanding them is pretty straightforward once you understand how BSTs work.*

3.2 Part II – The `RBTreeNode` and `RBTree` Classes

Tip 4 *Make sure your implementation from Part I actually works before doing Part II. I can give you partial credit if one of the two parts works, but not if both fail to work.*

You will implement two classes in this part: `RBTreeNode` and `RBTree`. You are welcome to derive those classes from the ones in Part I, but that is neither required nor expected.

Details on the additional methods for `RBTreeNode.cpp`:

1. `RBTreeNode<T>* treeMin()`
   Returns a pointer to a node with minimal value in the tree rooted at `this`.

2. `RBTreeNode<T>* treeMax()`
   Returns a pointer to a node with maximal value in the tree rooted at `this`.

3. `void printPreOrderTraversal() const`
   Prints to `stdout` the preorder traversal of the tree rooted at `this`.

4. `void printInOrderTraversal() const`
   Prints to `stdout` the inorder traversal of the tree rooted at `this`.

5. `void printPostOrderTraversal() const`
   Prints to `stdout` the postorder traversal of the tree rooted at `this`.

Details on the additional methods for `RBTree.cpp`:

1. `void transplant(RBTreeNode<T> *oldNode, RBTreeNode<T> *newNode)`
   Implement the `RB-TRANSPLANT` method from the textbook.

2. `bool isEmpty() const`
   Determine if the tree is empty in constant time.

3. `long size() const`
   Determine the number of nodes in the tree in constant time.

4. `RBTreeNode<T>* insert(T value)`
   Inserts a node with value `value` into the tree and returns a pointer to the inserted node.

5. `void remove(T value)`
   Removes a node with value `value` from the tree. If no such node exists, your code must throw a `value_not_in_tree_exception` exception. Use the method described in the textbook to preserve the integrity of all pointers using your code.

6. `RBTreeNode<T>* search(T value) const`
   Returns a pointer to a node with value `value`.

7. `RBTreeNode<T>* treeMin() const`
   Returns a pointer to a node with minimal value in the tree.

8. `RBTreeNode<T>* treeMax() const`
   Returns a pointer to a node with maximal value in the tree.

9. `void printPreOrderTraversal() const`
   Prints to `stdout` the preorder traversal of the tree.

10. `void printInOrderTraversal() const`
    Prints to `stdout` the inorder traversal of the tree.

11. `void printPostOrderTraversal() const`
    Prints to `stdout` the postorder traversal of the tree.