

Project 1

Balanced binary

Due: 11 September 2024

See the document “General Project Notes” for notes about how all projects will work in this course

Do not start work on this project until you’ve read the “AI” and “Progress reports” section of the handout

You saw basic binary search trees in 162, and may remember that their weakness is that in the worst case they behave like linked lists, with many operations requiring $O(n)$ time. The solution to this is to require that every operation maintain *balance* in the tree, and if carefully done you can guarantee add, search, and remove operations in $O(\lg n)$ time even in the worst case. One data structure with such guarantees is the red-black tree, which we’ll be covering in class over the next few days.

In this project, you’ll complete the implementation of a balanced binary search tree.

Objectives

In the course of this project, the successful student will:

- read complex code written by someone else (me),
- read and write unit tests,
- perform a case analysis, i.e. exhaustively enumerate the logically-possible cases in a system, and
- manipulate link-based data structures using pointers/references.

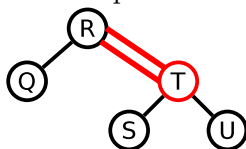
Prep work

Your starting point is a partial implementation that I have written. Start by creating a working directory for this project and copying my files into it; I’ve put them in `/home/shared/262/rbtrees/`. There are implementations

in C++, Java, and Python, and you may choose any of them to work with. (Please only copy the subdirectory for the language you plan to work with, or at least only include one of them when you hand in!)

The prep work mostly involves reading the existing code and becoming familiar with it. To give you some starting points on working with the code, I want you to perform the following tasks:

0. Read the readme! There's important instructions there on how to compile and run the code.
1. Fix the three compiler errors in the `RBTree` definition.
2. Fill in the `removeLeafRight` test case in the unit tests for `RBTree` to actually test that the code correctly removes leaves that are the right child of their parent node; compile it and run the tests to make sure it works. (Note that there are a few other cases that fail—`removeRotateThreeRight`, `removeRotateCaseII`, and `removeMystery`. Don't worry about those yet.)
3. Compile the speed test, run it, and put its output into a file named `speed.txt`.
4. At the bottom of the readme file, add lines that show exactly what the `TreeDump` algorithm would output if it were given the following tree:¹



You can either generate it by hand (after looking at the examples and seeing how `TreeDump` works) or write a small program to generate it.

This work is due Wednesday **at 8pm**. You'll have a chance to ask questions about it in class on Wednesday to clear up any last-minute issues, but if you haven't started it you won't have time to finish. When you're ready to hand it in, use the `handin` script as described at the end of this document.

¹To make the red nodes more distinct and easier to see, I generally draw both the nodes and their inbound edges in red, and I draw the inbound edges as a double line. The double line is visible even when the red isn't (and is suggestive of what the ASCII diagrams will look like...).

You shouldn't have to edit either the `TreeDump` algorithms or their test cases, but you're encouraged to look at them—*especially* the test cases, which will help you understand what `TreeDump` is doing.

Design work

Now that you've gotten the general layout, it's time to dive in to the algorithmic work. This will focus in particular on three parts of the program:

1. Verifying BST and red-black conditions. In `isValidRBTree` and in `validateAndCountBlack`, there is code to verify correctness of a red-black tree. Describe the algorithm. What is it checking for? How does it work?
2. Adding values. The main tree-modifying work here is concentrated in `splitFour` and `maybeRotate`. Draw a “before” and “after” diagram for a scenario where `splitFour` would be called, and what that tree should look like after the call to `splitFour`. Then, noting that there are four numbered cases inside `maybeRotate`, draw “before” and “after” diagrams for each distinct case. Use concrete, actual values in your diagrams (i.e. don't just draw the nodes). Note that in one of the cases my code is buggy and in another it is absent; your “after” drawings will reflect what *should* happen in those cases, based on your understanding of 234 and red-black trees, not what the code currently does.
3. Removing values. The important tree manipulations when removing values from the tree are managed in `guaranteeRedLeaf`, which works to ensure that by the time we get down to the leaf node that will be removed, that leaf node is a red one (while preserving the red-black conditions along the way). This code distinguishes twelve different cases, although some of those cases require no work to be done and all the actual work is handed off to helper functions. (Seven of the cases are not numbered; the last five are roman-numeraled I–V.) Again, draw diagrams to illustrate each case. Where no change is made, you can just say that, but for the cases where a tree manipulation occurs, draw an “after” diagram to illustrate it. You don't need to draw the entire red-black tree, just the portion relevant to the algorithm, but as before, make sure you use concrete values and indicate what value

the algorithm is seeking to (eventually) delete. Cases IV and V at the end are meant to be the mirror image of cases II and III respectively.

Write your design work on paper (or do it on your laptop, but paper's probably easier for this) and bring it to class; this work is due **at the start of class** on Wednesday the 30th. If you're really stuck on something, do your best, make a note of it, and move on; we'll be discussing this extensively in class. In fact, discussing this is the main topic of the day.

Final version

A full-credit final version will be a complete, non-buggy, working implementation of a red-black tree **TOGETHER WITH** convincing proof that it is correct. The form that proof takes will be a complete, rounded-out set of test cases along with empirical speed tests comparing the red-black implementation with the naïve BST implementation.

Note that I am not able to spend a ton of time with your program (and in fact may not read it at all, and definitely won't do your debugging for you), so your documentation will need to tell me anything I need to know to run and test your program. I'll also be running your tests against buggy code I've written, and your code against a complete set of test cases that I've written—so you shouldn't go changing the public interface of anything. Having complete and correct documentation is an easy 10 points, but if your documentation omits important info or tells me the wrong thing, you'll get less than full credit there.

After prep work (15 points), design work (10 points), documentation (10 points), and progress reports (15 points), there remain 100 points in the rubric, which will be awarded according to the table on the next page. Under each score, I show (for your convenience) the total cumulative points if you get that item plus *all* the previous points, and the letter grade this corresponds to. It is arranged roughly in the order I suggest you attempt them, with the earlier ones being easier or enlightening with respect to the later ones, but you can in general get points for the later ones (if they work) without getting the earlier ones.

BUT: if your code doesn't compile, or compiles but immediately crashes when it's run, you will get zero of these points. Don't let this happen to you!

Make good use of the case analysis you did in your design work to help you work through the rest of the implementation work—both in writing test cases and in understanding just what the implementation needs to be doing.

(continued next page)

Score	Description
10 (60/D-)	Get all the prep work done, whether or not you got it working by the original deadline. Include <i>both</i> the speed test for the naïve BST <i>and</i> the one for the red-black implementation.
15 (75/D+)	Add test cases for the other three cases in <code>maybeRotate</code> (currently only <code>addRotateCase2</code> is tested). 5 each.
5 (80/D+)	Fix the bug in the implementation of addition case 3.
10 (90/C)	Do the implementation of addition case 4.
5 (95/C)	Fix the bug in <code>rotateThreeNodeRight</code> .
5 (100/C+)	Write the <code>removeRotateCaseIII</code> test case.
5 (105/B-)	Write the <code>removeRotateCaseIV</code> and <code>removeRotateCaseV</code> test cases.
10 (115/B)	Add conditions to <code>guaranteeRedLeaf</code> to correctly identify cases IV and V (which are the mirror images of cases II and III).
10 (125/B+)	Implement deletion case II.
10 (135/A)	Implement deletion cases IV and V.
5 (140/A)	In the <code>add</code> method of <code>RBTree</code> , there is an <code>if</code> statement involving the parent. It is algorithmically important, although its omission doesn't currently break any test cases. Figure out why it's there, and write an appropriately-named test case that would break if it were omitted or commented out.
5 (145/A)	In the <code>remove</code> method of <code>RBTree</code> , there is an <code>if</code> statement to specially handle the case where the root and both its children are all effectively 2-nodes (i.e. none of them have red children). Using only public methods (<code>add</code> and <code>remove</code>), construct a test case that confirms that this code works correctly.
5 (150/A+)	The <code>removeMystery</code> test case builds a scenario that, in the current implementation, fails. Debug the problem and fix it.

Your final work is due, via the handin script (see below), by Wednesday the 11th at 8pm.

Handing in

There is a script called `handin` installed on all the lab machines that you'll use to hand in your code (once for the prep work, and then again for each progress report and for the final handin). This course is `cmsc262`, and the assignment is `proj1` so you'll type something like

```
handin cmsc262 proj1 bbst/
```

if the directory with your work in it happened to be named `bbst`, or just

```
handin cmsc262 proj1 .
```

(note the dot) if you are *in* the directory you mean to hand in. Read the output of the `handin` script—it will tell you just which files it handed in, or give you a hopefully-helpful error message to help you fix the problem.

You can run the `handin` script as many times as you like; for scoring purposes I only look at the latest-handed-in version. Even if you've handed in the “final unit” (see below), if you then realise you now understand a bug you created (and the deadline hasn't passed yet), fix it and resubmit for a higher score!

Progress reports

First, go read the note in the general project notes about “Self analysis and progress reports”.

In this project, I'm designating the following “units”:

- The prep work
- All the points for the basic add cases (test cases, bugs, implementation)
- All the points for the basic deletion cases (test cases, bugs, implementation)
- The special cases (the last 15 points on the rubric)

That means if you get the prep work done, do a bunch of work on the addition cases, and at least take a stab at some aspect of the deletion cases, you're handing in at least three times.

AI

You have the option of not using generative AI on this project.

If you're at all interested in playing with it, though, I encourage it, within the constraints I lay out in this section. (There will be similar sections in each project this semester but they might not have the same constraints!)

- You **MUST NOT** just paste in the project handout to get a final program to hand in.
- In fact, you **SHOULD NOT** paste in text from this handout at all (though you can paraphrase it when you're writing your AI prompt).
- But **MAY** paste short snippets of my provided code into an AI prompt, if you think that'll be helpful.
- Any code that is entirely, largely, or even just partially AI-generated **MUST** be marked off with comments that indicate what parts were AI generated, and which AI (e.g. ChatGPT, Gemini, etc) provided them.
- If you use generative AI at all, whether or not you made use of the results, you **MUST** create a subdirectory called "**ai**" and make a txt file in it for each of your AI interactions with a transcript that includes your prompt and everything in the AI's response (again, regardless of whether you ended up using it). The filename should include the AI name and either a datestamp or sequence number (e.g. `chatgpt-2.txt` or `claude-20240829.txt`).
- Remember that you should be working one piece at a time, so your AI requests **SHOULD** be sort of focused, and talking about how the AI helped you **SHOULD** be part of your self-analysis in the progress report.

This document was written and prepared without the use of generative AI.