UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS212 - Data structures and algorithms

Practical 4 Specifications - Red-Black Tree

Release date: 01-04-2024 at 06:00
Due date: 05-04-2024 at 23:59
Total marks: 220

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually; no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of the built-in Java data structures. Doing so will result in a zero mark. You may only use native 1-dimensional arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experiences a runtime error, you will be awarded a zero mark. Runtime errors are considered unsafe programming.
- Read the entire specification before you start coding.
- **Ensure your code compiles with Java 8**
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent), and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers to avoid any misunderstanding.** Also note that the OOP principle of code reuse does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

On completion of this practical, you will have gained experience with the following:

- Red-Black Trees, including:
  - Inserting data into Red-Black Trees and subsequent balancing rotations
  - Removing data from Red-Black Trees and subsequent balancing rotations
  - Validating Red-Black Trees in terms of the Red-Black tree properties

4 Introduction

Red Black Trees are a type of self-balancing binary data structure in the realm of computer science. The tree's structure ensures that it remains balanced with each insertion and deletion operation, thus maintaining optimal search, insertion, and deletion time complexities in the worst-case scenario. In a Red-Black Tree, each node contains an extra bit for denoting the colour of the node, either red or black, along with the standard links to the node's children and the data the node holds.

Balance is ensured by maintaining a set of properties:

1. Each node is either red or black.
2. The root of the tree is always black.
3. If a node is red, then its children are black.
4. Every path from a given node to any of its descendant NULL nodes contains the same number of black nodes.

Maintaining these properties guarantees O(log n) time complexity for the core tree operations. This makes RBTs invaluable in applications that require high operation efficiency regardless of the number of elements stored, such as database engines and file systems. Even Linux uses Red-Black trees for CPU scheduling.

This practical assignment aims to guide you through the implementation of a Red-Black Tree in Java, with a focus on implementing insertion and deletion and their associated rotations. As always, you are encouraged to introduce helper methods.

Additionally, a visualization tool for Red Black Trees has been developed and made available at `https://cos212.online/RedBlackTreeClient/page.html` (yes, we now have an SSL certificate) to support your understanding and debugging efforts. The visualiser provides a graphical representation of a working tree that is closely aligned with the memo and should offer insights into complex scenarios that may arise during the development of your Red Black Tree. We especially encourage you to use the visualiser to build test cases to validate your own code's output.

5 Tasks

You are tasked with implementing a Red-Black Tree. The `RedBlackNode` class and a skeleton for the `RedBlackTree` class have been provided. Your implementation must adhere to the given class diagram (Figure 1). **Please note that the testing of `isValidRedBlackTree` and `topDownDelete` depend on `bottomUpInsert`. If `bottomUpInsert` is incorrect then the testing of other functions is likely to fail too.**

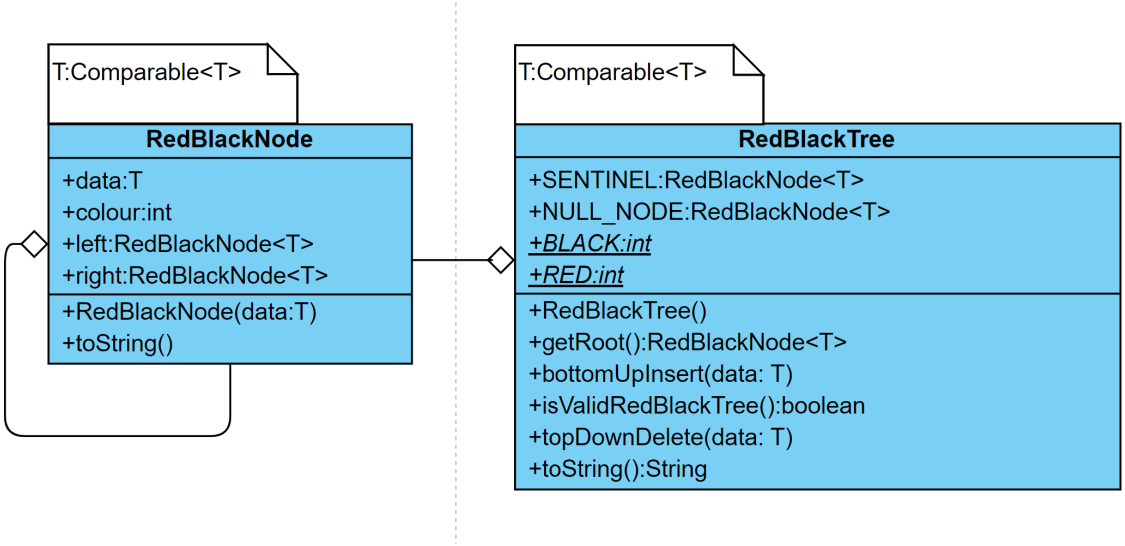

Figure 1: RedBlackNode and RedBlack tree UML class diagram

5.1 RedBlackTree<T extends Comparable<T> >

- Members:
  - SENTINEL: RedBlackNode<T>
    - The sentinel node of the Red-Black Tree.
    - The root will always be on the right of the sentinel node.
  - NULL_NODE: RedBlackNode<T>
    - The null node of the Red-Black Tree.
    - Since the null node needs a colour (black), we create a special node to function as the null node instead of the standard `null`.
    - If we want to know if a node is "null" we compare against this node.
  - RED: int
    - The value used to represent a red node.
  - BLACK: int
    - The value used to represent a black node.
- Methods:
  - RedBlackTree()
    - Constructor to initialize the Red-Balck Tree with a sentinel and a null node.
    - The initial children of the sentinel and null node should be set here.
  - getRoot():RedBlackNode<T>
    - Returns the root of the tree.
    - If the tree is empty returns the NULL_NODE
  - isValidRedBlackTree():boolean
    - Returns true if the tree adheres to all the Red-Black Tree properties, namely:
      1. Each node is either red or black.
      2. The root of the tree is always black.
      3. If a node is red, then its children are black.

4. Every path from a given node to any of its descendant NULL nodes contains the same number of black nodes.

    * Returns false if any of the Red-Black Tree properties are violated.

  - bottomUpInsert(data: T): void
    * Inserts a node with the given data into the tree.
    * This insertion should be done bottom-up, so any rotations and colour changes needed to restore the Red-Black Tree properties should be done after the new node is inserted. Note that fixing violations of the Red-Black Tree properties lower down in the tree (further from the root) may induce violations higher up the tree (closer to the root). The induced violations must also be dealt with.
    * Please refer to section 12.2.1 in the prescribed textbook for a detailed breakdown of the cases to consider.
    * If the tree already contains the given data then nothing should happen.
  - topDownDelete(data: T): void
    * Deletes the node with the given data from the tree.
    * If the tree does not contain a node with the given data then nothing should happen. Note that no rotations should be performed when checking whether or not the node is in the tree. Only once the node is confirmed to be in the tree can the top-down deletion begin, starting from the sentinel node.
    * Deletion of a node with two children should be done using the smallest node in the right subtree. If the node to be deleted only has a right child deletion should still be done using the smallest node in the right subtree. If the node to be deleted only has a left child deletion should be done using the largest node in the left subtree.
    * Deletion of a node with two children should be done using the smallest node in the right subtree. If the node to be deleted only has a right child deletion should still be done using the smallest node in the right subtree. If the node to be deleted only has a left child deletion should be done using the largest node in the left subtree.
    * For case 2A (the current node, X, has two black children), if both children of T are red, apply case 2A3 (where P, T and T's red child form a zig-zig). For example, if we delete 930 from Figure 2 we should choose 518 as the red child of interest since 852, 693 and 518 are in a zig-zig pattern. We should therefore rotate 693 over 852 then make 518 black, 693 red, 852 black and 930 red. A similar zig-zig (now right-right instead of left-left) should be chosen for the symmetric case.
    * Please refer to section 12.2.3 in the prescribed textbook for a detailed breakdown of the cases to consider.

- toString():String
  - Returns a string representation of the tree
  - This has been implemented for you. Please do not change it as we may test against it on FitchFork.

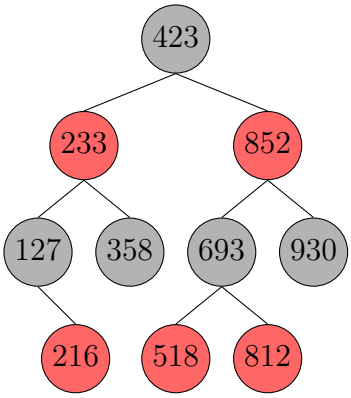

Figure 2: Red-Black Tree with ambiguous case 2A

6 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, approximately 10% of the assignment marks will be assigned to your testing skills. To do this, you will need to submit a testing main (inside the Main.java file) that will be used to test an Instructor-provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the jacoco tool. The following set of commands will be used to run jacoco:

```
javac *.java
rm -Rf cov
mkdir ./cov
java -javaagent:jacocoagent.jar=excludes=org.jacoco.*,destfile=./cov/output.exec
    -cp ./ Main
mv *.class ./cov
java -jar ./jacococli.jar report ./cov/output.exec --classfiles ./cov --html
    ./cov/report
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

and we will scale this ratio according to the size of the class.
The mark you will receive for the testing coverage task is determined using table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the instructor-provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

7 Upload checklist

The following files should be in the root of your archive

- Main.java
- RedBlackTree.java
- RedBlackNode.java
- Any other .java files that your solution requires
- Any .txt files you use for testing

8 Allowed libraries

- None, sorry :(

9 Submission

You need to submit your source files on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. Your code must be able to be compiled with the Java 8 standard.

For this practical, you will have 5 upload opportunities and your best mark will be your final mark. Upload your archive to the appropriate slot on the FitchFork website well before the deadline. **No late submissions will be accepted!**