Computer Science 2XC3: Lab 6 and 7

Dr. Vincent Maccio

Please read the following carefully. There are no completion/participation grades in labs. However, the TAs are there to guide and aid you in your tasks.

# Purpose

This lab focuses on implementing advanced data-structures. That is, it focuses on implementing self-balancing binary trees. There will be fewer experiments/analysis in this lab as the primary challenge with some of these data-structures can be getting them to work correctly! This lab will include but may not be limited to:

* A complete implementation of Red Black Trees (RBTs)
* An analysis of how much better if at all RBTs are over simple BSTs
  + Average-case and worst-case considerations

During the second week you will cover the following (but not necessarily limited to):

* Empirically analyse a formally defined tree structure to try and show its height it O(logn) where n is the number of nodes.

# Part 1

Throughout this lab you will be creating and ultimately submitting a lab report. Your lab report should look professional and complete. It should include the following:

* Title page
* Table of Content
* Table of Figures
* An executive summary highlighting some of the main takeaways of your experiments/analysis (this can be presented in bullet form if you wish)
* An appendix explaining to the TA how to navigate your code (for example, which .py file to find which implementation/experiment in)

For each experiment, include a clear section in your lab report which pertains to that experiment.

## Red Black Trees

In your 2C03 lectures you should have seen the insert functionality of Red-Black Trees (RBTs). I will also briefly go over the lab6.py code in your lectures this week. I do not have time in lecture to review all of the insert functionality. However, for a more advanced data structure like RBTs even something like insert can get confusing -- it is worth reviewing. Do a bit of independent research to (re)familiarize yourself with how these data structures behave. There are many good resources regarding the functionality and pseudo-code implementation of Red-black insert. For example:

* Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
* Introduction to Algorithms, 3rd Edition

both go over RBTs in detail. There are also many good online articles/tutorials regarding RBTS, such as:

* https://www.geeksforgeeks.org/c-program-red-black-tree-insertion/

Or if you are like me and cannot get enough YouTube the following two videos are also good. Careful, there is an error in the second one (see the comments).

* <https://www.youtube.com/watch?v=v6eDztNiJwo&t=924s&ab_channel=RobEdwards>
* <https://www.youtube.com/watch?v=5IBxA-bZZH8&t=242s&ab_channel=MichaelSambol>

In lab6.py you will find two partially completed classes: RBNode and RBTree. Together, once completed, they implement RBTs. It is your responsibility to complete three methods across these two classes: rotate\_left(), rotate\_right(), and fix().

* Rotate left and right are traditional rotate methods for binary trees. Pay close attention to what is pointing to what. Note that the nodes also point back to their parents. Also pay attention to how root works in the RBTree class. Once a rotation is made this may need to be updated!
* Read the insert method already partially implemented. Right now, it is quite similar to a traditional BST insert method. This is because none of the RBT properties are being enforced. As you know RBTs have 4-6 crucial properties (depending on what source your read). Here, the two most crucial to pay attention to are:
  + Red nodes cannot have red children.
  + All simple paths from the root to a leaf must contain the same number of black nodes.

As it stands now, these properties may be violated by an insert. You need to fix this. That is, you need to implement the fix() method to properly implement a RBT insert. Be careful, the devil is in the details here. I have given you a print method to help you test your tree's functionality. Submit your completed implementation in rbt.py.

## Experiment 1

We know that RBTs are self balancing, and BSTs are not. So should we always use RBTs over BSTs? Run an experiment where you create RBTs and BSTs based of randomly generated lists of numbers of length 10,000. What is the average difference in the height between the two? Comment on whether you think there is a case where you would prefer a BST over an RBT. You do not need to empirically validate this, but what is your instinct on the performance of a BST insert to a RBT insert for trees of similar heights? In your report your experiment should include:

* An explicit outline of how your experiment ran. For example, how many random lists were generated, etc.
* There is no need to have a graph for this section, but your results should be clearly stated.
* A brief discussion and conclusion regarding the results. A few sentences are fine here.

## Experiment 2

Run an experiment similar to Experiment 1, however, instead of generating arbitrary lists, start with a perfectly sorted list, record the difference in height and then repeat these experiments while allowing the generated lists to become more “unsorted” by allowing some random number of swaps (similar to what we did in the Sorting Lab – Experiment 5). In your report your experiment should include:

* An explicit outline of the experiments you ran. List length, number of swaps, number of runs, etc.
* A graph of number of swaps vs average different in height. For the number of swaps you do no necessarily need to have data-points for 0, 1, 2, 3,… That is, you can “step” through some values or choose certain values you may find interesting. For example: 0,1,2,3,4,5,10,25,50,100,.. etc.
* A brief discussion and conclusion regarding the results. A few sentences are fine here.

# Part 2

Often times (as you may have noted from Part 1), a lot of effort is put into determining properties of binary trees to ensure that their heights are O(logn). For example, RBTs are self-balancing, and therefore, have height O(logn), while BSTs have height in the worst case O(n). However, I think any reasonable person would agree that BSTs are “simpler” in nature.

Things can get even more complicated when we relax the constraint that our tree must be binary. That is, what if nodes can have more than two children? Consider an XC3-Tree, where XC3-Trees have the following properties:

1. If the root node of an XC3-Tree has *i* children, we say that XC3-Tree has degree *i*.
2. Each child of the root node of an XC3-Tree is also an XC3-Tree.
3. The *ith* child of the root node of an XC3-Tree has degree (*i*-2), if *i* > 2, and has degree 0 otherwise.

For example, the figure below would represent a degree 4 XC3-Tree:

A picture containing diagram

Description automatically generated

Even from the image above your intuition may not be clear on whether XC3-Trees are in fact “balanced” or “lop-sided” as the degree gets large. Our end goal for this portion of the lab is to show empirically that the height of a XC3-Tree is O(logn), where n is the number of nodes in the tree. Note, this is not a proof of the bound.

## Implementation

For the empirical analysis in the rest of the lab, it is worth implementing a rudimentary version of the XC3-Trees. In your implementation there is actually no need for the nodes to hold any values. However, from your data structure you should be able to determine the: degree, height, and number of nodes. You may implement this however you wish.

## Experiment 3

Create XC3-Trees with degrees 0 to 25. In you report, based off your trees write an equation *h(i)*, where *h(i)* returns the height of a degree *i* XC3-Tree. Explain why this is the case. These results likely aren’t mind-blowing but it is a reasonable place to start.

## Experiment 4

Create XC3-Trees with degrees 0 to 25. In your report, based off your trees write an equation for *nodes(i)*, where *nodes(i)* returns the number of nodes in a degree *i* XC3-Tree. At first glance this may seem tricky, but take a look at the values and see if you can find a pattern. Have you seen this pattern or a similar pattern to this before. You do not need to formally prove your claim, but give a plain English explanation to why you believe your claim holds. This explanation should relate back to the structure of XC3-Trees.

In your report, create an argument why the height of an XC3-Tree with n nodes is O(log(n)). Use the hints below to craft your argument:

* Due to how bases are changed in logarithms, to show f(n) = O(logn) it is enough to show f(n) = O(log­an) for any constant *a*.
* Take a look at how the golden ratio relates to your other observations in this experiment.

# Grading and Submission

Your group will submit the following documents to Avenue:

* report.docx (or .pdf, or whatever – as long as a reasonable person can open it)
* code.zip (all your source code pertaining to the lab – including experiment code)

In addition to the grade allocations below, your report may lose up to 20% of the final grade for not looking professional, having formatting/style issues, graphs presented in a messy manner, etc. Moreover, you may lose grades for not including elements explicitly mentioned in the Part 1 section of this document. Find a rough grade breakdown below:

Part 1

|  |  |
| --- | --- |
| RBT Insert Correctness | 30% |
| Experiment 1 | 10% |
| Experiment 2 | 10% |

Part 2

|  |  |
| --- | --- |
| Experiment 3 | 10% |
| Experiment 4 – Number of nodes pattern | 20% |
| Experiment 4 – Argument on height bound | 20% |