Computer Science 2XC3: Graded Lab IV

Please read all instructions carefully. If you have any questions please reach out to your TA's to guide you through the task. All the questions are provided in this pdf and the code is in the ipynb file. You have the freedom to incorporate your code and reflections all in 1 ipynb, or code in jupyter notebook and reflections (and charts) in separate Word document. You will submit a ipynb file, a pdf-converted version of your .pynb file (and your reflections file in case you decide to keep them in a separate file). Whatever you decide, keep your work organized. Answers that are difficult to locate in a maze of text and code will **not** be graded.

Part 1: Implementing and Analyzing Red-Black Trees

In your 2C03 lectures, you have studied Red-Black Trees (RBT) and seen (at least in theory) how they are implemented. If you do not remember them or never studied them, now is the time. Below are a few useful links that include the pseudo-code implementation of RBT. I will try to cover some of the concepts in class, but since covering theory is NOT the goal of this class, I might not go in depth.

- Introduction to Algorithms, CLRS, 3rd Edition
- https://www.cs.cornell.edu/courses/cs312/2005sp/lectures/lec15.html
- https://www.youtube.com/watch?v=ZxCvM-9BaXE

Given in the notebook is a partial implementation of Red Black Trees with three methods rotate_left (), rotate_right(), fix().Rotate_left performs the operation of rotating the node left and Rotate_right performs the operation of rotating the node right. The nodes also point back to their parents, and the fix method updates the tree once the rotation has been made.

Part 1.1

In this exercise, complete these implementations by enforcing the properties of Binary Search Trees that make it an RBT. Two of these properties are:

- Red nodes cannot have red children.
- All simple paths from the root to a leaf must contain the same number of black nodes.

Every time you insert an element in the tree, the above properties get violated and need to be fixed (remember how we adjust the heap in priority queues?)

Part 1.2

The essential difference between RBTs and BSTs is that RBTs are self-balancing and hence offer a computational advantage. However, that may not always be the case. In this experiment, you will design an experiment to test out this theory. Generate a list of length 10000 elements and create both BSTs and RBTs using that list. You will realize that the kind of empirical tests you have done in the past may not work here. So how can you test which is better? The height of the two trees can give you a good intuition. Test the average difference between the height of the two trees created on the same list. In your reflection section, explain why this difference is important and how it can be leveraged to select between RBTs and BSTs. You run multiple experiments on different lists and plot a graph on tree heights to support your reflection.

Part 1.3

In this experiment, generate a perfectly sorted list and record the difference between the height of the two trees for multiple rounds. Then create different versions of the list varying the level of "unsortedness" (like you have done in previous labs with near-sorted lists and unsorted lists). You can use controlled quicksort to create these versions of near-sorted list. In your reflection, describe the experiment design, and the runs, and show in the graph how the "degree of sortedness" impacts the height of the two trees. Refer to the swap count experiment you ran to determine the degree of sortedness. Write a detailed reflection on what you observe and why it might happen.

Part 2: Making Binary Search Dynamic

Binary search of a sorted array takes logarithmic search time. However, if you notice closely, the insertion of new element takes linear time. By keeping several sorted arrays, you can improve the time for insertion. For instance, you can SEARCH and INSERT separately on a set of n values. It can go something like this for a set of n elements:

Let $k = [\lg (n + 1)$, and let the binary representation of n be $\{n_{k-1}, n_{k-2}, \dots, n_0\}$. Maintain k sorted arrays $A_0, A_1, \dots, A_{k-1}$ where for $i = 0,1, \dots., k - 1$, the length of the array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1 \; or \; n_i = 0$, respectively. The total number of elements held in all k array is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, elements in different arrays bear no relationship to each other. This can allow binary search to be implemented using dynamic programming.

Part 2.1

Implement the SEARCH, INSERT and DELETE operations for binary search using the above description. For this, the code has not been provided. Feel free to go back to previous labs and use the Binary Search implementations discussed there, to build on.

Part 2.2

In this section, you will compare your implementation against traditional implementations. Take ALL the Binary Search implementations we have discussed in the first few weeks and design an experiment to compare the performance against your dynamic implementation.

In your reflection section, describe in detail the experiment design, what are the various list sizes and the number of trials you chose. Also, describe your observations. Did you notice any performance improvement? When does dynamic binary search outperform others? When is it an overkill? Describe in detail your experiments design, and the observations.