CS5800: Algorithms Spring 2018
Assignment 5

Out: 1 March 2018 **Due: 15 March 2018, 8:59pm**

Instructions:

- The assignment is due at the time and date specified. Late assignments will not be accepted.
- You must work on all the problems and write the solutions by yourself! Finding solutions to homework problems on the web, or by asking students in and outside the course is strictly prohibited. This would be defeat the purpose of learning by doing the assignment.
- You must submit typed solutions. You may use plain text or a word processor like Microsoft Word or LaTeX for your submissions. You may hand-sketch and scan any diagrams that you support your answer.
- If you are not comfortable with Word or Latex, please solve these problems by hand before devoting time to typing them. Do not waste precious time investigating typesetting up front!

You are provided some starter code with this assignment, that you are expected to use and extend in some of these questions. The code is complete with some JUnit tests. If you are not familiar with setting up a Java environment that works with JUnit, please see the video on blackboard on how to work with IntelliJ. You are not required to use this IDE.

1. (30 points)

A priority queue has the following operations:

1. add(item,priority)
2. getItemWithHighestPriority()

The priority is assumed to be a non-negative integer. The item can be anything in general. The items may be orderable (comparable).

A priority queue is normally implemented using a binary heap. A binary heap can be efficiently implemented using an array as a complete binary tree. Both the above operations on a binary heap can be implemented in $O(\log n)$ **worst case** time. To summarize, both operations move several array elements around to maintain the heap structure (operations often called "percolate"). See Chapter 6 of the book for more details.

As an implementation detail, each node of a binary heap (i.e. each element in the underlying array) stores directly the item and its priority (instead of a pointer/reference to the actual data). Storing everything in the heap itself avoids indirection, which improves cache performance of the heap contents.

A useful, but non-standard operation is changePriority(item,new_priority). This operation can be used to increase or decrease an existing item's priority. Once the item is found in the binary heap, this operation can be implemented in $O(\log n)$ time. The problem is finding this item efficiently: the binary heap is arranged by priority, not item. Thus searching a binary heap by item amounts to a linear-time search of the underlying array, which causes the change priority operation to run in linear time.

Provide a design that implements a priority queue using an ordinary binary heap, supports all operations including changePriority(item,new_priority) in $O(\log n)$ **worst case** time. Your answer must consist of any algorithms and data structures you use, and must prove that the overall running time is within the specified bounds. Your answer must specify everything so that an average graduate student in CS 5800 should be able to implement your idea by reading only your answer, and assuming knowledge of how a binary heap is normally implemented.

2. (30 points)

You are given a binary search tree T without any balancing mechanisms that stores n items. Due to repeated additions and deletions, the tree has become lopsided/unbalanced, which is having an adverse effect on its efficiency.

- Provide an efficient way to create a binary search tree T' that has the same data as T, that also does not have any balancing mechanisms, and yet implements search in $O(\log n)$ time after it has been fully created. You may assume that once T' is created, it is not changed (alternatively, if it is changed, this process can be repeated to get back the efficient run times). Analyze your algorithm in space and time.
- You are provided two implementations for such a "normal" binary search tree. Add code to both implementations wherever applicable to implement this feature. You may add any additional tests as well, but we will not grade them.

Please submit both answer and code for this question.

3. (30 points)

- Provide an algorithm that checks whether two binary search trees are the same. Two binary search trees are the same if they contain the same data, and they have the same structure (i.e. if you draw them on paper they will look identical). Analyze your algorithm in time and space.
- Provide an algorithm that will create an identical copy (i.e. same as above) of a binary search tree. Analyze your algorithm in time and space.
- Implement both these algorithms in the given code (both implementations).

Please submit both answer and code for this question. You may implement all questions in one code base, instead of separating them for different questions.

4. (10 points)

Briefly discuss the differences between the two approaches of implementing a binary search tree provided and extended by you, in terms of how algorithms are implemented, ease of use and understanding, efficiency, etc.