CS 206 Final Review

Spring 2019

NAME: _____

The instructions you'll also see on the final are shown below. The final itself will be *shorter* than this packet and you will be given *more* room to write your answers. For a list of *topics* that will be covered on the midterm, see the class syllabus for weeks 9 - 15 as well as the contents of lecture notes and assignments. While example topics are given on this practice sheet, this is not an exhaustive list of all possible problems or even types of problems that may be on the exam.

1. You have **180 minutes** to complete this exam. All answers should be written on this exam (you may use the back of the pages).
2. This is a closed book exam. **No notes or other aids are allowed** other than a calculator.
3. The exam contains **XX pages** including this instruction sheet. Make sure you have all the pages. Each question's point value is next to its number. The total exam is worth **XX points**.
4. In order to be eligible for as much partial credit as possible, show all of your work for each problem, and **clearly indicate your answers**. Credit cannot be given for illegible answers.
5. Java code that you write should be as close to correct (runnable) as possible. Small syntax errors will not cost you points, but code that is unclear *will*. **If you are not sure how to write correct code for a problem, you can include a clear comment about how you think the code should be written that describes the algorithm (steps) it should follow to be correct for partial credit.**
6. Be sure to choose good variable names and comment if you think it will help. **You may lose points for correct but uncommented or unclear code - if I don't understand it.**
7. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.
8. Be sure to read the entire exam before answering any questions, so you have time to think about the potentially tricky problems.
9. This examination is **NOT** to be shared with students who take this class in subsequent years, nor circulated in any manner. Thank you for abiding by the Honor Code in respecting this restriction.

1 Running Algorithms and Data Structures

One good way to study for computer science exams is to run any algorithms and draw any associated data structures on a variety of inputs. Here are some example questions to get you started — obviously, you can make your own additional practice problems by simply changing the input examples.

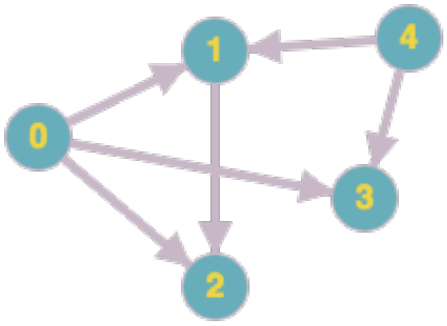

Figure 1: Graph 1

a) Draw the adjacency list and adjacency matrix for Graph 1. For each node also indicate the number of neighbors it has, and if the graph is directed also indicate the in-degree and out-degree of all of the nodes.

b) Insert the following items into a binary search tree *in the given order*. Do this twice: the first time you should insert them into a binary search tree that is potentially unbalanced, the second time you should insert them into an AVL tree, being sure to maintain the balance property and clearly labeling the checks and rotations performed at each step. When inserting into the AVL tree, you should make it clear what rotations have been done by drawing a new copy of the tree after each insertion, both before and after the rotation. Items to insert:

   3, 1, 4, 5, 8, 2, 6, 9, 7

c) Perform pre-order, in-order, and post-order traversals on both of the above trees and clearly write down the results. Are the traversal results the same for both trees? Why or why not?

d) Delete the following items from the binary search trees you created above *in the given order*. Draw the tree after each item is deleted. In the case of the AVL tree, draw the tree both after the deletion and after the rebalancing is done. Items to delete:

   3, 8, 2, 6, 7

e) Insert the following items into a *min*-heap *in the given order*. Draw the heap after each insertion.

   9, 3, 2, 5, 7, 1, 13

f) Delete the following items from the min-heap you created above in the given order. Draw the heap after each item is deleted. Items to delete:

   1, 2, 9

g) Using the hash function $h(x) = x\%7$ and a maximum hash table size of $N = 7$, create three hash tables using i) linear probing, ii) quadratic probing, and then iii) double hashing with the function $d(x) = 3 - x\%3$ and insert the following hash keys into those three tables:

35, -10, 21, 7, 0, 2

Calculate and write down the *load factor* of the hash tables after each insertion.

h) Showing your work by writing down each step of the algorithm and each new sorted list or sublist, sort the following list of numbers using i) bubble sort, ii) heap sort, iii) merge sort, and iv) quick sort:

3, 1, 6, 9, 7, 3, 8, 4

i) Make a union-find tree structure. After each operation, draw the new structure. Perform the operations below:

create(A)
create(B)
create(C)
create(D)
union(A, B)
union(C, D)
find(D)
union(A, C)
find(C)

2 Complexity

In addition to being able to run algorithms and simulate the resulting data structures, it's important to know the computational complexity of these operations and express them using big-Oh notation. You should be able to do this both for algorithms and data structures that you've seen before, and for those you haven't.

a) Determine the complexity of each operation or algorithm you performed when running the examples in the previous section. Since complexity depends on a specific implementation, you may need to make some assumptions about how, e.g., the trees are represented as a data structure in order to determine this. On the exam, such assumptions would be made explicitly for you.

b) Suppose that you had access to a priority queue data structure that could perform insertions and poll operations in $O(1)$ time. What would the complexity of heap-sort be if it used this new data structure instead of a heap?

3 Code

If you haven't already finished all of the assignments and labs to your satisfaction, one good way to study is to go back to the labs to finish or polish them. In addition, there are some example problems below - these should give you an idea of what types of problems to expect, but are not an exhaustive list of possible coding questions. The below problems also expect you to write more code than would be expected on an exam (where some of the code would likely be provided for you, and you might be writing a single function). Be sure to try writing this code on paper! (It's of course fine to test your answers later by running them on the computer.)

a) While we focused on an array-based implementation of a heap in assignment 7, another way to implement a heap is actually to use a `LinkedBinaryTree`. As an exercise, let's work through some of the standard operations.

1) `insert`: Items should be inserted into the heap by inserting into a non-perfect subtree. You may find it useful to create a method within your `Node` class to determine if the subtree rooted at that node is perfect. Recall that in a perfect subtree the size of the subtree $size = 2^{height+1} - 1$, thus, you may also find it useful to keep track of the size and height of each subtree. Once you determine where to insert the new node and insert it, your insert method should then call `upHeap` (see below) to fix the heap property.

2) `remove`: In order to remove the given element from the heap, you'll need to first find the node containing that element. You can do this by making a helper function that recursively looks at all nodes of the tree and returns the node where the element is equal to the given one. Once you have the `Node` object to remove, you'll find the "last" node in the heap (i.e., the one in the position that should be removed to maintain the heap shape), swap that with the node to remove, and then remove the node. Finally, use the `downHeap` helper method you should write (see below) to fix the heap property of the swapped node.

3) `upHeap`: Create a `void upHeap(Node node)` method that recursively swaps the given node up the tree and is called on a node after it has been inserted in the bottom level of the tree or swapped to a part of the tree where it violates the heap property.

4) `downHeap`: Create a `void downHeap(Node node)` method that recur- sively swaps the given node with the larger of the two children down the tree until the heap property is satisfied.

5) `swap`: You may find it useful to have a helper method: `private void swap(Node nodeA, Node nodeB)` that does the work of swapping two nodes as well as all of their associated links that can be called from both heapifyUp and heapifyDown. This method should:

   i. Store all of the relevant original information for each of the nodes to swap: the left child, right child, parent, height, and size.

   ii. Set each of the above pieces of information to its swapped value for both nodes. Note that in order to do this, you'll need to have some way to identify whether a node was a left or right child of its parent. You could do this by modifying your `Node` class to keep track of this information (in which case you'll also need to be careful to set that information correctly within `swap`) or by checking equality.

   iii. Be sure to also set the relevant left child, right child, and parent pointers for the parent of the given nodes and the children of the given nodes.

b) Assume that you are given code for a `Node` class that has the usual getters and setters for a `Node` in a `LinkedBinaryTree`. Write a method to perform a breadth-first tree traversal; it should take a `Node` as input and return a String representing a breadth-first traversal of the subtree rooted at the given node.

c) Assume that you are writing an UndirectedGraph class that uses an underlying adjacency matrix representation for the graph. Given the index i for a Node, write a method to determine the degree of Node i.