UNIVERSITY OF TORONTO
Faculty of Arts and Science

DECEMBER 2016 EXAMINATIONS

CSC263H1F

Duration - 3 hours

Instructor(s): David Liu

No Aids Allowed

Name:

Student Number:

Please read the following guidelines carefully!

- Please write your name and student number on the front of the exam.
- This examination has 7 questions. There are a total of **18 pages, DOUBLE-SIDED**.
- Answer questions clearly and completely. Give complete justifications for all answers unless explicitly asked not to. You may use any claim/result from class, unless you are being asked to prove that claim/result, or explicitly told not to.
- You must earn a grade of **at least 40% on this exam to pass this course.**

Take a deep breath.

This is your chance to show us

How much you've learned.

We **WANT** to give you the credit

That you've earned.

A number does not define you.

It's been a real pleasure teaching you this term.

Good luck!

Question	Grade	Out of
Q1		8
Q2		10
Q3		7
Q4		11
Q5		9
Q6		6
Q7		13
Total		64

1. **[8 marks] Algorithm analysis, problem lower bounds.** Consider the following operation:

   - DISJOINT(A, B): A and B are non-empty lists of integers. Return True if A and B have *no* elements in common, and False otherwise.

   (a) **[1 mark]** State one input (pair of lists) to DISJOINT that would make this operation return True, and one input that would make it return False. Clearly label which one is which. No justification required.

   (b) **[3 marks]** Suppose we use the following algorithm to solve this problem:

```
1 def Disjoint(A, B):
2     for i from 0 to A.length - 1:
3         for j from 0 to B.length - 1:
4             if A[i] == B[j]:
5                 return False
6     return True
```

   Let n be the length of A, and m be the length of B. Prove that the worst-case running time of this implementation of DISJOINT is $\Omega(mn)$.

(c) [**4 marks**] Let n be the length of A, and m be the length of B. Use the adversary argument to show that *any* correct implementation of DISJOINT must perform at least $n + m$ array accesses in total.

2. **[10 marks] Priority Queues and Heaps.**

Your goal is to implement the following abstract data type. This ADT is very similar to a priority queue, except each item has an additional associated *secondary priority*.

DoublePriorityQueue ADT

- INSERT($PQ, item, p1, p2$): insert the given item with priority $p1$ and *secondary priority* $p2$.
- FINDMAX(PQ): return the item with the highest priority.
- EXTRACTMAX(PQ): remove and return the item with the highest priority.
- FINDMAXSECONDARY(PQ): return the item with the highest *secondary priority*.

Your implementation must have the following worst-case runtimes:

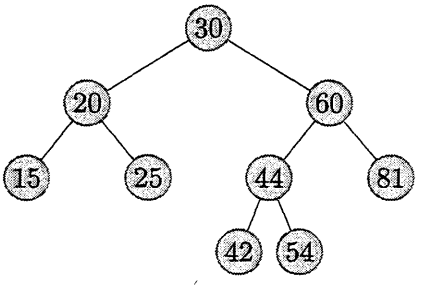
- INSERT, EXTRACTMAX: $\mathcal{O}(\log n)$, where n is the number of items.
- FINDMAX, FINDMAXSECONDARY: $\mathcal{O}(1)$, i.e., independent of the number of items.

(a) **[5 marks] Describe in English** a data structure composed of *two heaps* that could be used to implement this ADT. Specify what data is stored in the heaps, but do not discuss the operations themselves. You may assume an infinite amount of allocated memory for the heap arrays (so don't worry about array resizing).

(b) **[5 marks]** Show how to implement ExtractMax (in pseudocode) using your data structure, and briefly justify why this implementation runs in $\mathcal{O}(\log n)$ time in the worst case.

3. **[7 marks] Binary Search Trees and AVL Trees.**

   (a) **[3 marks]** Consider the following AVL tree:



   Suppose we insert a value chosen uniformly at random between 1 and 100 inclusive, using the AVL Tree INSERT algorithm from lecture. Calculate the **expected number of rotations** that will occur for this insertion. You do not need to simplify nor add fractions in your final answer.

   You may make the following two assumptions in your work:

   - If the chosen value is already in the AVL Tree, no rotations occur.
   - At most one node will have an imbalance fixed. That is, the possible number of rotations performed is always between 0 and 2.

(b) **[1 mark]** Let n be a positive integer. Suppose we have an empty binary search tree, and insert the numbers $\{1, \ldots, n\}$ into it, in some order, using the naïve INSERT algorithm for BSTs (no rotations). Let B_n be the number of permutations that result in a BST of height exactly n.
Find the value of B_1. No justification is required.

(c) **[3 marks]** Prove that for all $n \geq 2$, $B_n = 2B_{n-1}$. **Do not** assume that $B_n = 2^{n-1}$.

4. **[11 marks] Hash tables.** Recall the DISJOINT operation from Question 1:

   - DISJOINT(A, B): A and B are non-empty lists of integers. Return True if A and B have *no* elements in common, and False otherwise.

   In this question, you'll think about how to use a hash table to implement this function.

   (a) **[5 marks]** Show how to implement DISJOINT using a hash table with *closed addressing (chaining)*. Clearly state the size of the hash table you are using; this should depend on the size of the input lists to make your algorithm efficient on average.

   Write both **pseudocode** as well as **brief English justification** about the correctness of your algorithm. You solution will be evaluated based on both correctness and efficiency, but do not analyse the running time of your algorithm here.

   Hint: don't forget that A can contain duplicates, and B can contain duplicates. Your algorithm should be able to handle such inputs.

(b) **[3 marks]** Find an asymptotic upper bound on the *worst-case* running time of your algorithm. Your upper bound should be tight, but you do not need to prove this.

(c) **[3 marks]** Assume the simple uniform hashing assumption applies to all values hashed in your algorithm (that is, each value has an equal probability of hashing to any spot in the hash table). Find an asymptotic upper bound on the *average-case* running time of your algorithm.
Your upper bound should be tight, but you do not need to prove this.

5. [**9 marks**] **Graph Searches**. Consider the following puzzle game. We have a collection W of n distinct strings, where each string is a word consisting of the 26 lowercase letters. We are given a *start word* and *target word* from W, and want to get from the start word to the target word using a sequence of "valid moves." A *valid move* in this game is one of two different operations:

   - Change one letter of the current word. For example, going from "bore" to "core".
   - Insert one letter into any position of the current word. For example, going from "core" to "score".

   For example, here is a sequence of valid moves to get from "bore" to "slope" (assume all the words used are in W:

   - bore, core, score, scope, slope

   Our goal is to implement a function that finds the **shortest sequence of valid moves** to get from the start word to the target word. You may assume that the start and target words are different.

   (a) [**3 marks**] Show how to represent this problem using some type of graph we have studied in this course. Be sure to state what the vertices and edges in the graph correspond to.

(b) **[6 marks]** Show how to solve this problem using a variation of breadth-first search with your graph representation from part (a). Your algorithm should take as input the graph representation of the puzzle game, and the start and target words. It should return the *entire sequence* of words from the start word to the target word, or `null` if it is impossible to reach the target word from the start word using valid moves.

Give both **pseudocode** and a **brief English description** of what your algorithm does. You do not need to analyse the efficiency of your algorithm.

6. **[6 marks] Minimum Spanning Trees.** Here is a heap-based version of Prim's algorithm, which stores the set of edge extensions in a heap.

```
1  def PrimMST(G):
2      v = pick starting vertex from G.vertices
3      TV = {v}  # MST vertices
4      TE = {}   # MST edges
5      extensions = new heap  # treat weights as priority, with minimum weight at the top
6      for each edge on v:
7          Insert(extensions, edge)
8
9      while TV != G.vertices:
10         e = ExtractMin(extensions)
11         u = endpoint of e not in TV
12
13         TV = TV + {u}
14         TE = TE + {e}
15         for each edge on u:
16             if other endpoint of edge is not in TV:
17                 Insert(extensions, edge)
18
19     return (TV, TE)
```

(a) **[3 marks]** Find an exact upper bound on the size of the `extensions` heap during the execution of this algorithm.

(b) **[3 marks]** Assume that the input graph $G = (V, E)$ is connected, so that $|E| \geq |V| - 1$. Prove that this implementation of PRIMMST has a worst-case running time of $O(|E| \log |E|)$.

7. **[13 marks] Amortized analysis, Disjoint sets.**

   (a) **[5 marks]** Let k be a natural number greater than 1. Consider a dynamic array that supports only INSERT, with the following modifications:

   - The initial number of allocated spots is 0.
   - Whenever we INSERT into a full array (the array's `size` is equal to its `allocated`), the amount of space allocated *increases by k*.

   In other words, the number of allocated spots starts at 0, then becomes k, then $2k$, $3k$, etc.

   Let $M > 0$ be a multiple of k. Find the exact aggregate (i.e., total) cost of performing M INSERT operations using this modified dynamic array implementation. As in lecture, count only the number of array accesses, and remember that copying an element costs *two* accesses.

(b) [**5 marks**] Consider the disjoint set implementation that uses the union-by-rank heuristic.

```
1  def Union(DS, x, y):
2      root1 = Find(DS, x)
3      root2 = Find(DS, y)
4
5      if root1 == root2:
6          return
7
8      if root1.rank > root2.rank:
9          root2.parent = root1
10     else if root1.rank < root2.rank:
11         root1.parent = root2
12     else:
13         root1.parent = root2
14         root2.rank = root2.rank + 1
```

Let x be a set representative of one of the disjoint sets, i.e., the root of one of the trees, and let r be the `rank` of x. Prove by induction on r that there are *at least* 2^r elements in the same set as x. Remember that all nodes start in their own individual disjoint set, and initial `rank` 0.

(c) [**3 marks**] Suppose we have a disjoint set that uses path compression, but *not* union-by-rank. Furthermore, suppose the n items stored are the numbers $\{1, \ldots, n\}$, which are all in the same disjoint set, arranged in a tree of height n (the maximum possible height).

Suppose we pick x and y independently and uniformly at random from $\{1, \ldots, n\}$, and then perform the operations FIND(x) and FIND(y), in that order.

Find the **exact probability** that the FIND(y) operation visits exactly two nodes.

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question.*

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question.*