PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Department of Computer Science

Sample Term Test – Solutions

CSC 263H1
Duration — 100 minutes

Examination Aids: One 8.5"x11" sheet of paper, handwritten on one side.

The questions in this sample test and their solutions have been collected from tests and exams of past offerings of the course. Therefore, the style and approaches used in the sample solutions might slightly be different than what we did in class.

IMPORTANT NOTE: One of the key factors in a test is **time** pressure. You may be able to answer a question when you have an extended amount of time, but not within the time limit the examiner expects. The followings are the time expectations that designers of the questions in this sample test had in mind:

Q1: 10 min

Q2: 10 min

Q3: 25 min

Q4: 20 min

Q5: 25 min

When answering the questions **time yourself**! If you cannot complete a question within the corresponding time limit it means that you are not prepared yet and need to practice more.

Good Luck!

Question 1. [0 MARK]

Part (a) [0 MARK]

What is the *exact* number of key swaps performed by the non-linear-time BUILD-MAX-HEAP procedure (i.e., the one with $\Theta(n \log n)$ worst-case run time) when it is executed on array $A = \langle 25, 6, 19, 9, 8, 15, 22, 7, 2, 1, 3, 4 \rangle$?

Number of key swaps: _____

Answer: 3

Part (b) [0 MARK]

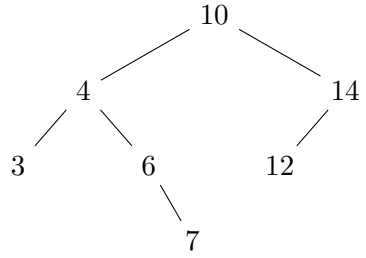
Perform an INSERT(12) operation on the Max Heap array $A = \langle 25, 9, 19, 6, 8, 15, 4 \rangle$, and *show the resulting array*. Do *not* show any intermediary result.

Answer: $A = \langle 25, 12, 19, 9, 8, 15, 4, 6 \rangle$

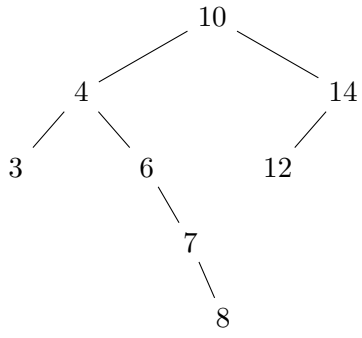
Question 2. [0 MARK]

Part (a) [0 MARK]

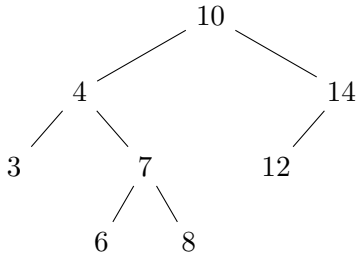
Insert a node with key 8 into the AVL tree pictured below. Show the intermediate and resulting trees.



SAMPLE SOLUTION:

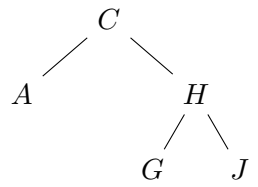


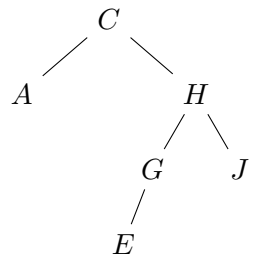
Left rotation over 7



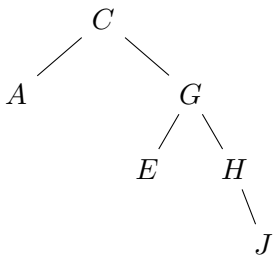
Part (b) [0 MARK]

Insert a node with key E into the AVL tree pictured below. Show the resulting tree by either drawing directly on the original tree or by drawing a new tree in the extra space.

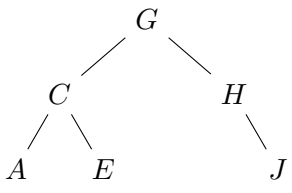




Right rotation over G



Left rotation over G



Question 3. [0 MARK]

Consider the following Python code which simulates rolling a pair of dice and counting the number of rolls until we reach n pairs. We are interested in the number of time the `random.randint` method is called, which corresponds to the number or rolls. Do not express the complexity in $\mathcal{O}$ notation but give exact expressions.

```
from random import randint

# Attempt to roll n pairs. Use a maximum of 10n rolls.
def countRolls(n):
    count = 0
    tries = 0
    while count < n and tries < 10*n:
        die1 = randint(1,6)      # roll the first die
        die2 = randint(1,6)      # roll the second die
        if die1 == die2:
            count += 1
        tries += 2       # count these 2 rolls even if we don't get a pair
    return count, tries
```

Part (a) [0 MARK]

Perform a worst-case analysis of `countRolls`.

SAMPLE SOLUTION:

- Worst-case occurs when the n'th pair occurs last or when n pairs do not occur in the 5*n tries.
- For each of 5*n tries we do 2 rolls for a total of 10*n rolls.
- We can't do worse than this because the algorithm stops after 10*n rolls regardless of how many pairs have been found. It can only stop earlier than this, not run longer.

Part (b) [0 MARK]

Perform an average-case analysis of `countRolls`. You do not need to simplify your expressions.

SAMPLE SOLUTION: There are at most 5*n attempts to roll a pair. We will count the attempts from 1 to $5n$. The probability of rolling a pair on each attempt is $\frac{1}{6}$. Each of these attempts is independent. The probability of rolling the n^{th} pair on the i^{th} roll is given by

$$p_{n,i} = \begin{cases} 0 & 1 \leq i < n \\ \frac{1}{6}^n & i = n \\ \frac{1}{6}^n \frac{5}{6}^{i-n} \binom{i-1}{n-1} & n < i \leq 5 * n \end{cases}$$

Then you still need the probability that we roll all 5*n attempts and don't get n pairs. We can calculate this as follows:

$$p_{\text{not found}} = 1 - \sum_{i=1}^{5*n} p[n^{th}\text{pair found on roll}i]$$

Using both those pieces we calculate the average number of rolls as:

$$10n * p_{\text{not found}} + \sum_{i=1}^{5n}(2i * p_{n,j})$$

Question 4. [0 MARK]

SMALLESTA(A, k):
    Create a max-heap from the first k elements in A, used to store the k smallest elements seen so far.
    $H \leftarrow A[0 \ldots k-1]$
    MAXHEAPIFY(H) called BUILD-MAX-HEAP in textbook
    Now examine each element in A to see if it is one of the smallest seen so far.
    for $i = k, k+1, \ldots, A.length - 1$:
        if $A[i] <$ MAX(H):
            EXTRACTMAX(H)
            INSERT$(H, A[i])$
    return H

The algorithm uses H to keep track of the k smallest elements seen so far. Initially, this is just the first k elements in A. Then, as each element of A is examined, it is compared with the largest element in H: if it is smaller, then it is one of the k smallest seen so far, so H is modified by removing its largest element and replacing it with $A[i]$ (we could make this slightly more efficient by copying $A[i]$ directly into the "root" of H and then percolating down the value). At the end, H will contain the k smallest elements in A.

Worst-case runtime:

- $\Theta(k)$ for creating max-heap H.
- $\Theta(\log k)$ for each iteration of the main loop (for EXTRACTMAX and INSERT).
  Note that for an array of size n where the k largest elements appear in indices 0 to $k-1$ (and there are no duplicates), the if-condition is satisfied in every iteration.
- Total is $\Theta(k + (n-k)\log k)$.

Question 5. [0 MARK]

You are designing an ADT that contains information about students. Each student has a name, an age and a score. In addition to being able to insert and delete students, you must provide the following new operation in worst case complexity $\mathcal{O}(\log n)$ where n is the number of students. You may assume that the ages are unique.

- *FindBestWithinAge*(*agelimit*): returns the student with the highest score from all students whose age does not exceed *agelimit*.

You will accomplish this by augmenting an AVL tree.

1. What will you use as the key for the tree?

   Solution: age — if we needed to look up students by name we would need an additional data-struture to map from name to age – another AVL tree or a hashtable would do.

2. What additional information will you store at each node?

   Solution We would need name and score but also the maximum score of the nodes in the subtree rooted at this node. Call it *max_in_subtree*.

3. Draw a valid AVL tree for the following records showing any additional information.

Name	Age	Score
Don	8	172
Eshan	10	180
Ken	14	190
Kevin	9	165
Matt	11	185
Nick	6	167
Sam	7	169
Scott	12	187

   Solution Lots of options that are valid. The tree must obey the AVL balance property.

4. Give the algorithm for *FindBestWithinAge*(*agelimit*) explaining briefly why it is $\mathcal{O}(\log n)$.

   Solution

```
def find_best(age_limit, current_root):

    if current_root == NIL:  #this age_limit is not in the tree
        return -infinity
    elif current_root.age == age_limit:   # we've found the node with this age
        return max(current_root.score, find_best(current_root.left)
    elif age_limit < current_root.age:  # going left
        return find_best(age_limit, current_root.left)
    elif age_limit > current_root.age:  # going right
        # all the values in root and left_subtree are less than age_limit
        return max(current_root.score, current_root.left.max_in_subtree,
```

```
                    find_best(age_limit, current_root.right))

    FindBestWithinAge(agelimit):
        return find_best(age_limit, root)
```

```
Note: Create a NIL node for every left and right child where none exists.
NIL.max_in_subtree = -infinity
```

5. What else do you need to be concerned with when you augment this data-structure? Address this concern here.

   Solution

   You need to keep the new information *max_in_subtree* correct on insertions and deletions (including any rotations) on the AVL tree.

6. The assumption that we have only one student in each category is ridiculous. What would have to change in the data-structure or algorithms to accommodate non-unique ages? How would those changes affect the runtime of the operations?

   Solution Now when we find a node with the age_limit we can't know if there are other nodes with this same value in either the left or right subtrees (or both!). So we need to explore the right subtree. In the solution above the second case gets added to the fourth case. The runtime is still $\mathcal{O}(\log n)$ because we still move down one level of depth in the tree with each recursive call and do a constant amount of work per call.