COS 226 Algorithms and Data Structures Spring 2018

Midterm

This exam has 8 questions (including question 0) worth a total of 80 points. You have 80 minutes. This exam is preprocessed by a computer, so please **write darkly** and **write your answers inside the designated spaces. Whenever there is a multiple choice question, fill in the appropriate bubble(s) with your pencil.**

Policies. The exam is closed book, except that you are allowed to use a one page cheatsheet (8.5-by-11 paper, one side, in your own handwriting). No electronic devices are permitted.

Discussing this exam. Discussing the contents of this exam before solutions have been posted is a violation of the Honor Code.

This exam. Do not remove this exam from this room. Write your name, NetID, and the room in which you are taking the exam in the space below. Mark your precept number. Also, write and sign the Honor Code pledge. You may fill in this information now.

Name:

NetID:

Exam room:

Precept:

P01	P02	P02A	P03	P04	P04A	P04B	P05	P05A	P05B
◯	◯	◯	◯	◯	◯	◯	◯	◯	◯

"I pledge my honor that I will not violate the Honor Code during this examination."

Signature

0. **Initialization. (2 points)**

In the space provided on the front of the exam, write your name, NetID, and the room in which you are taking the exam; mark your precept number; and write and sign the Honor Code pledge.

1. **Short questions: Union-Find (6 points)**

Note: there are `union` implementations from the book added for your reference at the bottom of the page.

a) How many connected components result after performing the following sequence of union operations on a set of 10 items?

1 – 2 3 – 4 5 – 6 7 – 8 7 – 9 2 – 8 0 – 5 1 – 9

○ 1 ○ 2 ● 3 ○ 4

b) Suppose the following set of union operations were performed in quick-find.

1 – 2 3 – 4 5 – 6 7 – 8 7 – 9 2 – 8 0 – 5 1 – 9

How many array accesses required to determine `find`(7)

● 1 ○ 2 ○ 3 ○ 4

c) What is the maximum number of `id[]` array entries that can change (from one value to a different value) during one call to `union` when using the quick-find data structure on N elements?

○ 1 ○ $\log N$ ● $N - 1$ ○ N

For your reference, below is the `union` implementation for quick-union and quick-find from the book:

```
public void union(int p, int q) {
// from quick-union
  int i = find(p);
  int j = find(q);
  if (i == j) return;
  id[i] = j;
  count--;
}
```

```
public void union(int p, int q) {
// from quick-find
  int pID = id[p];
  int qID = id[q];
  if (pID == qID) return;
  for (int i = 0; i < id.length; i++)
     if (id[i] == pID) id[i] = qID;
  count--;
}
```

2. **Short questions: data structure analysis (10 points)**

a) Suppose that, starting from an empty data structure, we perform N push operations in our resizing array implementation of a stack. How many times is the `resize()` method called?

◯ constant ● logarithmic ◯ linear ◯ quadratic

b) Suppose that you implement a queue using a null-terminated singly-linked list, maintaining a reference to the item least recently added (the front of the list) but not maintaining a reference to the item most recently added (the end of the list). What are the worst case running times for enqueue and dequeue?

◯ constant time for both enqueue and dequeue
◯ constant time for enqueue and linear time for dequeue
● linear time for enqueue and constant time for dequeue
◯ linear time for both enqueue and dequeue

c) Consider the code fragment:

```
for (int k = 1; k < N; k = k*2)
    sum++;
```

How many addition operations does the above code fragment perform as a function of N?

◯ $\sim N$ ◯ $\sim N/2$ ● $\sim \log_2 N$ ◯ $\sim 0.5 \log_2 N$

d) Which of the following order-of-growth classifications represent the (best case, worst case) number of array accesses used to binary search an array of size N?

◯ (constant, constant) ● (constant, logarithmic) ◯ (constant, linear)
◯ (logarithmic, logarithmic) ◯ (logarithmic, linear) ◯ (linear, linear)

e) The code below shows the instance variables for `WeightedQuickUnionUF`.

```
public class WeightedQuickUnionUF {
    private int[] id;    // id[i] = parent of i
    private int[] sz;    // sz[i] = number of objects in subtree rooted at i
    private int count;   // number of components
    ...
```

How much memory (in bytes) does a `WeightedQuickUnionUF` object use as a function of N?

◯ $\sim 2N$ ◯ $\sim 3N$ ◯ $\sim 4N$ ● $\sim 8N$

◯ $\sim 12N$ ◯ $\sim 16N$ ◯ $\sim 28N$ ◯ $\sim 32N$

3. **Short questions: Sorting. (10 points)**

a) Consider the data type Temp defined below.

```
public class Temp implements Comparable<Temp> {
    private final double deg;

    public Temp(double deg) {
       this.deg = deg;
    }

    public int compareTo(Temp that) {
       double EPS = 0.1;
       if (this.deg < that.deg - EPS)
          return -1;
       if (this.deg > that.deg + EPS)
          return +1;
       return  0;
     }
}
```

Which of the following required properties of the Comparable interface does the `compareTo()` method violate?

◯ Antisymmetry (for all v and w if both $v <= w$ and $w <= v$ then $v = w$)
● Transitivity (for all v, w, and x, if both $v <= w$ and $w <= x$, then $v <= x$)
◯ Totality (either $v <= w$ or $w <= v$ or both)
◯ Reflexivity (for all v, $v = v$)
◯ None of the above

b) How many compares does selection sort make when the input array is already sorted?

◯ constant ◯ logarithmic ◯ linear ● quadratic ◯ exponential

c) How many compares does insertion sort make on an input array that is already sorted?

◯ constant ◯ logarithmic ● linear ◯ quadratic ◯ exponential

d) In QuickSort, suppose we partition an array of size 1251 and the pivot ends up in position 867. What is the entire set of positions in which the *median* might be found? *Assume no elements are equal to each other.*

◯ Positions 0 through 383. ◯ Positions 0 through 625.
● Positions 0 through 866. ◯ Any position.

e) If we have an array of size N with only 3 different values for its elements, what is the approximate probability that the first QuickSort partition results in a completely sorted array? Assume there are an equal number of each element in the array.

● 0% ● 33% ◯ 67% ◯ 100%

Remark: it depends on whether 2-way or 3-way sort is used: it's near-0 for 2-way, but ~ 1/3 for 3-way — we'd need to select a middle value as a pivot. We accepted both answers.

4. **Data structure analysis. (8 points)**

Consider the following task `QueueExchangeRandom(Queue<Item> q)`: given a queue `q` with at least two elements, pick two uniformly random elements from `q` and exchange their location. In each of the following scenarios, indicate the best running time that can be achieved by an implementation of `QueueExchangeRandom` on a queue with n elements.

Number of operations proportional to:	**1**	$\log \mathbf{n}$	**n**	$\mathbf{n} \log \mathbf{n}$
Only with access to the standard public Queue API	○	○	●	○
With access to the private variables of `q`*, where the Queue is implemented as a resizable array*	●	○	○	○
With access to the private variables of `q`*, where the Queue is implemented as a linked list*	○	○	●	○
With access to the private variables of `q`*, where the Queue is implemented as a red-black BST, with values being the values in the queue, and keys representing the insertion order into the queue*[1]	○	●	○	○

[1]You may assume that the BST is the full implementation here, which supports all standard ST key-valued operations as well as insertions and deletions in logarithmic worst-case time.

5. **Debugging. (12 points)**

Consider the following code snippet for finding the minimum in a stack of integers without affecting its contents:

```
1  public int findMin(Stack<Integer> s1)
3  {
5    Stack<Integer> tmp = new Stack<Integer>();
7    if (s1==null)
9      throw new java.lang.IllegalArgumentException("Null stack");
11   if (s1.isEmpty())
13     throw new java.lang.IllegalArgumentException("Empty stack");
15   int min = s1.pop();
17   int t;
19   while (!s1.isEmpty())
21   {
23     t = s1.pop();
25     if(t < min)
27        min = t;
29     tmp.push(t);
31   }
33   while (!tmp.isEmpty())
35     s1.push(tmp.pop());
37   return min;
39 }
```

There is a bug in this code. The next two questions refer to that bug.

1) What is/are the undesirable **side effect(s) of the bug** which may occur during execution? Check **all** that may apply:

Null pointer exception	○
Side effects on the contents of the stack after execution	●
Returned answer incorrect	○
Linear-time execution	○

2) The bug can be fixed by inserting one line of code into the snippet. Identify line number to insert

○ 14 ● 16 ○ 22 ○ 24 ○ 26

○ 28 ○ 30 ○ 32 ○ 34 ○ 36

Write the code to insert in the box below.

```
tmp.push(min);
```

6. **MinPQ. (20 points)**

In the unit on priority queues we have covered minimum and maximum oriented priority queues `MinPQ` and `MaxPQ`. The purpose of this problem is to implement a *median oriented priority queue*. The median of a list containing $2n+1$ elements is the middle $n+1$-st element. The median of a list with $2n$ elements is the n-th element. So the median of $\{1, 5, 7\}$ is 5, and the median of $\{1, 2, 5, 7\}$ is 2. Here we will focus on key components of the API:

`MedPQ()`	create an empty priority queue
`void insert(Key v)`	insert a key
`Key median()`	returns the median
`Key delMedian()`	returns and removes the median

The goal of the implementation is to perform the insertions and deletions in time proportional to $\log_2$(current size), and to perform the `median()` operation in constant time. There are multiple ways of doing this, but here we will focus on an implementation with two private priority queues (which we assume are implemented using a heap).

```
public class MedPQ<Key> implements Iterable<Key> {
    private MaxPQ<Key> left;
    private MinPQ<Key> right;

    public MedPQ() {
         left = new MaxPQ<Key>();
         right = new MinPQ<Key>();
    }
```

In the box below, describe what is stored in `left` and `right` in your implementation:

At a high level, `left` will hold the bottom half of the list, and `right` will hold the top half.

More specifically, if there are $2n$ elements in the list, the bottom n will be in `left` and the top n will be in `right`. If there are $2n+1$ elements in the list, the bottom $n+1$ will be in `left` and the top n will be in `right`. This way, median is always just the largest element of `left`.

During insertion and deletion all we have to do is to maintain these two rules:
(1) elements in `left` are $\leq$ elements in `right`, and
(2) the size of `right` $\leq$ the size of `left` $\leq$ the size of `right`$+1$.

Implement the `median()` method (which should run in constant time).

```
public median() {
     if (isEmpty()) throw new NoSuchElementException("Priority queue underflow");

return left.max();

}
```

Implement the `insert` method (which should run in logarithmic time, which means a constant number of calls to methods for `left` and `right`).

```
public void insert(Key k) {
     if (k == null) throw new IllegalArgumentException("Null key");

if(left.isEmpty()) {
    left.insert(k);
    return;}

if(less(k,left.max()))
    left.insert(k);
else
    right.insert(k);

if (left.size() > right.size()+1)
    right.insert(left.delMax());

if (left.size() < right.size())
    left.insert(right.delMin());

}
```

Implement the `delMedian` method (which should run in logarithmic time, which means a constant number of calls to methods for `left` and `right`).

```
public Key delMedian() {
     if (isEmpty()) throw new NoSuchElementException("Priority queue underflow");

 Key k = left.delMax();

 if (left.size() < right.size())
     left.insert(right.delMin());

 return k;
}
```

7. **Which is the loneliest number? (12 points)**

Given a collection of integers, the distance between an element i and the collection is the minimum difference between i and the closest element in the collection. For example, consider the collection $\{1, 5, 10, 15, 17\}$. The distance between 10 and the collection is 5, and the distance between 5 and the collection is 4. The loneliest element in a collection of integers is defined as the element with maximum distance to the collection. In the above-mentioned collection, the loneliest element is 10. the next loneliest elements are 1 and 5.

Note: parts (a) and (b) below are not directly connected.

(a) Given a typical Binary Search Tree implementation representing a list of n integers, how long would the best algorithm take to find the loneliest element in the average case? Time proportional to:

○ 1 ○ $\log n$ ● n ○ $n \log n$ ○ n^2

(b) Describe how to implement an object which maintains a list of integers with insertion and deletion, and which allows for querying for the loneliest element. For full marks, the memory used should be proportional to n — the current size of the list, `insert`, `delete` and `loneliest` should take time proportional to $\log n$.

You need not implement any methods, just give a clear and concise description of which data structure(s) you would use as instance variables, and how your solution would work in the boxes below. You may use the space on the last page as scrap paper.

In the box below, provide a concise and precise description of the instance variables you would use and their purpose:

Solution:

The logarithmic insertion/deletion requirement is a hint that LLRBs are involved. In fact, we need two LLRBs, one to keep track of the integers, and one to keep track of the distances.

In the *numbers LLRB* implementing a ST, the keys are the numbers and the values are distances to the closest element in the list.

In the *distances LLRB*, we would like the keys to be just distances, but unfortunately different numbers can have the same distance. Therefore, we need to have a wrapper comparable class of the form (distance, number) which is compared first based on distance, and in case of a tie based on the number. Since we are promised that all the numbers are distinct, there will be no identical pairs in the distance tree (ignoring this issue only led to 1-2 point deduction).

In the box below, describe how you would implement `loneliest()`. Use pseudocode or concise, but precise, English prose, or a combination of both.

Run `max` on the *distances LLRB* to obtain a (distance, number) pair. Return the number.

In the box below, describe how you would implement `insert(int v)`. Use pseudocode or concise, but precise, English prose, or a combination of both.

Check if v is in the *numbers LLRB*. If it is, return, since no further action is necessary.

Let $l = floor(v-1)$ and $r = ceiling(v+1)$ be the successor and predecessor of v in the list of numbers (one or both could be null).

Calculate the distance d of v to the closest element. Insert (v, d) into the *numbers LLRB*.

Recalculate the distances $d_{l,new}$ and $d_{r,new}$ of l and r (assuming they are not null).

Retrieve the distances $d_{l,old}$ and $d_{r,old}$, which are the values corresponding to keys l and r in the *numbers LLRB*.

Update the values corresponding to l and r in the *numbers LLRB* to $d_{l,new}$ and $d_{r,new}$.

Update the *distances LLRB*:
Remove $(d_{l,old}, l)$ and $(d_{r,old}, r)$;
Insert $(d_{l,new}, l)$, $(d_{r,new}, r)$, and (d, v).