COS 226 Spring 2019 Midterm reference solutions

1. Memory [6 points]

Use the 64-bit memory cost model from lecture and the textbook to answer the questions below.

A. How much memory does a `Queue` consume as a function of the number of `Node`s n? Exclude the memory for the items themselves. Use tilde notation to simplify your answer.

```
public class Queue<Item>
{
      private Node first, last;
      private class Node
      {
            Item item;
            Node next;
      }
      ...
}
```

Answer: ~ $40\,n$ bytes

B. How much memory does a `BookListing` use? Assume that ISBNs are always 13 characters long.

```
public class BookListing
{
      private char[] isbn;
      private double price;
      ...
}
```

Answer: 88 bytes

C. How much memory does a `Queue` of `BookListing`s use as a function of the number of book listings n? Include all referenced memory. Use tilde notation to simplify your answer.

Answer: ~ $128\,n$ bytes

2. Analysis of algorithms [6 points]

Calculate the number of invocations of `op()` in each of the following code snippets as a function of n. Use tilde notation to simplify your answer.

```
public static void alpha (int n) {
      for (int i = 0; i < n/2 + 10; i++)
            op();
}
```

Answer: ~ $\frac{1}{2}n$

```
public static void bravo (int n) {
      for (int i = 0; i < n*n; i++)
            for (int j = 0; j < i; j++)
                  op();
}
```

Answer: ~ $\frac{1}{2}n^4$

```
public static void charlie (int n) {
      if (n <= 1) return;
      charlie(n/3);
      for (int i = 0; i < n; i++)
            op();
      charlie(n/3);
      charlie(n/3);
}
```

Answer: ~ $n \log_3 n$

Extra credit [1 point].
For `delta`, assume that n is a power of 2.
(Hint: this one is tricky; you may want to attempt it at the end.)

```
public static void delta (int n) {
      for (int i = 1; i < n; i = i * 2)
            delta(i);
      op();
}
```

Answer: ~ n

3. Union find [6 points]

Consider the following sequence of union commands on a set of 10 elements:

1—7 4—0 7—2 6—2 9—5 2—7 9—0

Show the resulting array for each of the following union-find implementations.

Recall that the role of `p` and `q` in `union(p, q)` can be swapped without affecting the correctness of the algorithm. For each implementation below, feel free to use either order, as long as you're consistent across `union` invocations (the answers resulting from either order will receive full credit).

A. Quick Find

	0	1	2	3	4	5	6	7	8	9
id[]	0	2	2	3	0	0	2	2	8	0

B. Quick Union

	0	1	2	3	4	5	6	7	8	9
parent[]	0	7	2	3	0	0	2	2	8	5

C. Weighted Quick Union

	0	1	2	3	4	5	6	7	8	9
parent[]	4	1	1	3	9	9	1	1	8	9

4. Sorting [10 points]

For each of the following sort algorithms, what is the order of growth (as a function of n) of the number of compares needed to sort an array of n distinct items that's already sorted in reverse order?

A. Insertion sort

$\theta($ n^2 $)$

B. Selection sort

$\theta($ n^2 $)$

C. Mergesort, as implemented in Java for sorting objects

$\theta($ $n \log n$ $)$

D. Quicksort (with shuffling, average case)

$\theta($ $n \log n$ $)$

E. Heapsort

$\theta($ $n \log n$ $)$

The code below repeatedly invokes the same sort algorithm n times on an array, where n is the length of the array:

```
public static void repeated_sort(Comparable[] a){
      for (int i = 0; i < a.length; i++)
            Sorter.sort(a);
}
```

Assuming that the initial array `a` is sorted in reverse order, what is the order of growth (as a function of n) of the number of compares made by `repeated_sort` when `Sorter.sort` is replaced by each of the following algorithms?

F. Insertion sort

$\theta($ n^2 $)$

G. Selection sort

$\theta($ n^3 $)$

H. Mergesort, as implemented in Java for sorting objects

$\theta($ n^2 $)$

I. Quicksort (with shuffling, average case)

$\theta($ $n^2 \log n$ $)$

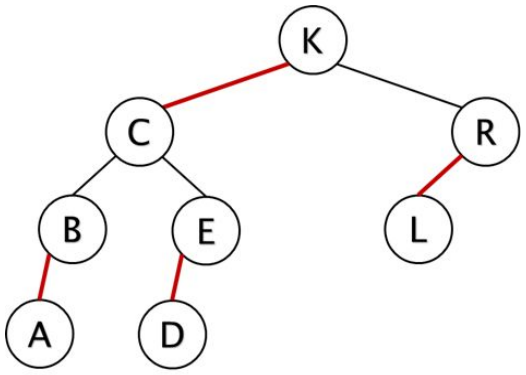
J. Heapsort

$\theta($ $n^2 \log n$ $)$

The last one is tricky! Java's mergesort includes an optimization for already-sorted arrays, as discussed in lecture.

5. Search trees [9 points]

A. Label each node in the following binary tree with keys from the set {R, E, D, B, L, A, C, K} so that it is a valid Binary Search Tree with respect to alphabetical ordering of keys.



B. Now design an algorithm to do what you did in part A. Specifically, given a binary tree with n nodes, where all the keys are `null`, and an array of n distinct keys, replace each `null` key in the binary tree with one of the keys in the array so that it forms a valid BST.

Give a concise English description of your algorithm. It will be graded for correctness, efficiency, and clarity. For full credit, the number of compares made by your algorithm must be proportional to n log n in the worst case.

1. Sort the keys using mergesort or heapsort
2. Traverse the tree inorder and replace keys with successive items from the sorted array.

Recall from lecture and textbook that inorder traversal yields keys in sorted order. Here we flip that observation to replace keys in sorted order.

C. In what order might the eight keys have been inserted so that it would have resulted in the BST above?

K C R B E L A D

D. Is your answer to part C unique, or is there a different order that would have resulted in the same BST?

Unique [ ] Not unique [X]

E. Now label each edge in the binary tree (in part A above) with *r* or *b*, denoting RED or BLACK, so that the tree is a valid Left-Leaning Red-Black Tree.

6. Debugging [6 points]

Consider the following implementation of a linear probing hash table. Code is shown only for the `put` method, which has a bug. Assume that all other methods are correctly implemented.

```
01    public class LinearProbingHashST<Key, Value>
02    {
03          private int n;   // number of key-value pairs in the symbol table
04          private int m;   // size of table
05          private Key[] keys;
06          private Value[] vals;
07
08          public LinearProbingHashST(int capacity) { ... }
09
10          private int hash(Key key) { ... }
11
12          // resizes the keys and vals arrays to the given size,
13          // rehashing all the keys
14          private void resize(int capacity) {
15                ...
16          }
17
18          public void put(Key key, Value val) {
19                if (n / m >= 0.5)
20                      resize (2 * m);
21                int i;
22                for (i = hash(key); keys[i] != null; i = (i+1) % m)
23                      if (keys[i].equals(key))
24                            break;
25                keys[i] = key;
26                vals[i] = val;
27                n++;
28          }
29
30          public Value get(Key key) { ... }
31
32          public void delete(Key key) { ... }
33    }
```

The intent behind the code is to resize the symbol table when it is half full, but it fails to do so. Describe how to fix this bug: write the line number of the code that you would change, and how you would change the code.

Line number	Fixed code
19	`if (2 * n >= m)`

What are the effects of this bug? Check all that apply.

	Compilation error.
	Incorrect values returned by `get`.
	Infinite loop in `put` when table gets full.
	Hash table consumes too much memory compared to correct implementation.
X	`get` makes too many `equals` calls compared to correct implementation.
X	`put` makes too many `equals` calls compared to correct implementation.

The code has an unintended second bug — n *shouldn't be incremented when replacing a value. The fix is:*

Line number	*Fixed code*
24	*`{ vals[i] = val; return; }`*

Of course, finding and fixing either bug received full credit, and the second part of the question was graded in reference to the bug that you reported.

7. Data structure and algorithm design [15 points]

A Point is an object consisting of an x- and a y-coordinate. Your goal is to maintain a collection of Points that supports the following operations:

- Add a Point to the collection
- Return the Point with the lowest x-coordinate
- Return the Point with the lowest y-coordinate
- Delete the Point with the lowest x-coordinate
- Delete the Point with the lowest y-coordinate

If there are multiple Points with the same x- or y-coordinate, you may choose among them arbitrarily.

Your answers below will be graded for correctness, efficiency, and clarity. For full credit:

- Any sequence of n invocations of the supported operations (in any order), starting from an empty collection, must complete in time proportional to n log n in the worst case.
- Returning the Point with the lowest x- or y-coordinate must take constant time.

You may make any standard technical assumptions that we have seen in this course.

There are two main possible approaches: one that uses two symbol tables and one that uses two priority queues. In the symbol-table approach, the main challenge is that duplicate keys are not allowed, whereas duplicate coordinates are allowed in this problem. In the priority-queue approach, the main challenge is that when a point is deleted from one PQ, the PQ API doesn't allow deleting it from the other PQ. Both problems can be solved; the solution below uses the priority-queue approach.

A. Describe the data structures you would use. Specifically, for any new data structures you need, write the class declaration. For any data structures from lectures/textbook that you would use, succinctly describe how you would use them and what modifications are needed (if any).

We'll use two min-heaps, one of which maintains a heap ordering of points by *x*-coordinate and the other by *y*-coordinate. We need 3 new classes; the latter two are the keys for the heaps.

```
public class PointKey {
   private Point point;
   private boolean deleted; // has this Point been deleted from collection?
}

public class PointKeyX {
   private PointKey pk;
   // compareTo() compares the x-coordinate
}

public class PointKeyY {
   // similar to PointKeyX
}
```

B. Give a concise English description of your algorithm for adding a Point to the collection. Feel free to use some pseudocode if you think it will improve clarity.

- Create a new `PointKey` referencing the given `Point`, with `deleted` set to `false`.
- Create a new `PointKeyX` and `PointKeyY` that reference the new `PointKey`.
- Insert them into the respective heaps.

C. Give a concise English description of your algorithm for returning the Point with the lowest *x*- or *y*-coordinate. Feel free to use some pseudocode if you think it will improve clarity.

Call `min()` on the respective heap and return the `Point` that it references.

D. Give a concise English description of your algorithm for deleting the Point with the lowest *x*- or *y*-coordinate. Feel free to use some pseudocode if you think it will improve clarity.

Pseudocode for deleting the Point with the lowest *x*-coordinate:

```
pkx = x_min_heap.delete_min()
pkx.pk.deleted = True

while(x_min_heap.min().pk.deleted == True)
    x_min_heap.delete_min()

while(y_min_heap.min().pk.deleted == True)
    y_min_heap.delete_min()
```

When we delete a Point, the corresponding `PointKey` *is marked as deleted, but it may remain in one of the heaps. The code above ensures that the minimum key in either heap never corresponds to an already-deleted Point.*

E. What is the worst-case order-of-growth running time of your design for a sequence of n invocations of the supported operations (in any order), starting from an empty collection?

$\theta($ $n \log n$ $)$

F. Explain your answer to part E.

Returning the minimum is constant time, so we can ignore those operations.

Each point that is inserted is inserted once into each heap, and each point that is deleted is deleted at most once from each heap. So the number of compares of our design is at most twice the number of compares for a sequence of n inserts/deletes into a single binary heap. That number is $\theta(n \log n)$, because each binary heap operation is logarithmic.