COS 226 **Algorithms and Data Structures** **Spring 2019**

Midterm

This exam has **7 questions** worth a total of **58 points**. You have **80 minutes**. The exam is preprocessed by a computer (yay algorithms!), so please write inside the designated spaces.

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

Discussing the contents of this exam before solutions have been posted is a violation of the Honor Code. Do not remove this exam from this room.

Name:

NetID:

Exam Room:

Precept:

P01	P01A	P02	P03	P04	P04A	P05	P05A	P05B	P06
○	○	○	○	○	○	○	○	○	○

Write this in the box below:

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

_____

Signature

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: ~ ~~32n~~ 40n 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: ~ ~~120n~~ 128n 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: ~ $\frac{n}{2}$

```
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[]	0	7	7	3	0	0	7	7	8	5
	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

$\theta($ $n \lg n$ $)$

D. Quicksort (with shuffling, average case)

$\theta($ $n \lg n$ $)$

E. Heapsort

$\theta($ $n \lg n$ $)$

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

sort n times an array with n elements

```
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

$\theta($ ~~$n^2 \lg n$~~ $)$

✓ n^2

I. Quicksort (with shuffling, average case)

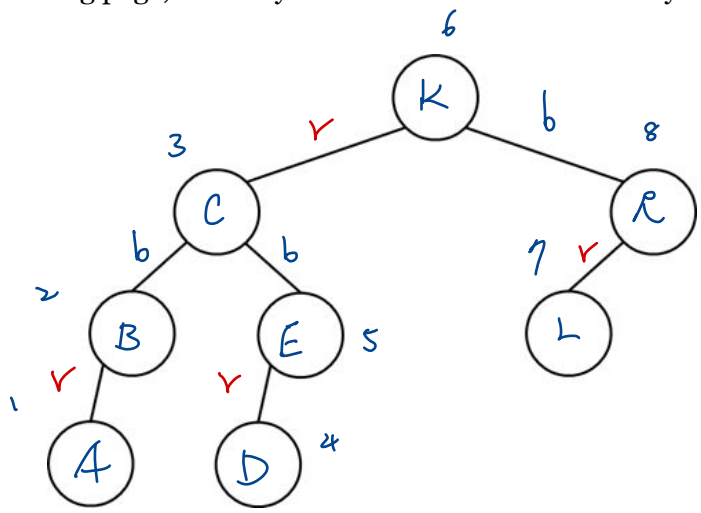
$\theta($ $n^2 \lg n$ $)$

J. Heapsort

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

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. (Hint: use the scratch space on the facing page, and only write down the answer once you have it.)



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. use Quicksort to sort the given array to ascending order
2. def insert(k, node):
       if (node == null): return k   # base case
       k = insert(k, node.right)
       if (k >= 0): node.key = a[k--]
       k = insert(k, node.left)
       return k
3. if __name__ == '__main__':
       insert(len(array)-1, root)   # len(array) = n, root: root node of BST
```

✓ alt: In-order:
node's left child → visit action on node → node's right child

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

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) → Breadth-first search

R L K E D C B A → in-order

e.g. K C B A E D R L → pre-order

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

but if with other algo, e.g., traverse the left child first, the order can be different (reverse).

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.

Scratch space

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.

24 vals[i] = val; return;

Line number	Fixed code
~~27~~	delet line 27 and add if (keys[i] == null) n++; before line25

✓19 ✓ if (2 * n > m)

(ie, insert the new key into the place where keys[i] is null.)

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.
- [X] ✓ Hash table consumes too much memory compared to correct implementation.
- [X] `get` makes too many `equals` calls compared to correct implementation. ($\frac{n}{m} \geq 0.5$)
- [X] `put` makes too many `equals` calls compared to correct implementation.

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

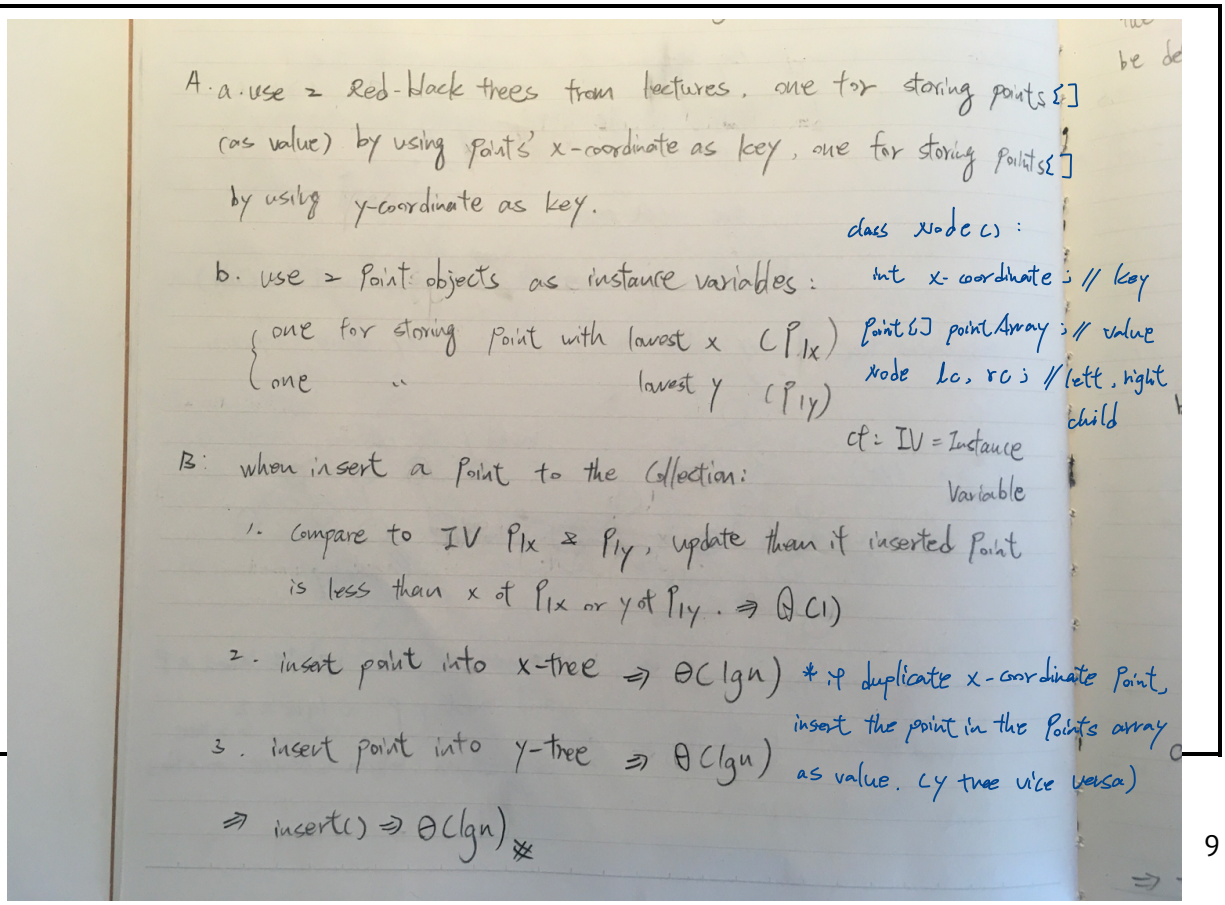
If there are multiple Points with the same *x*- or *y*-coordinate, you may choose among them arbitrarily.

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.

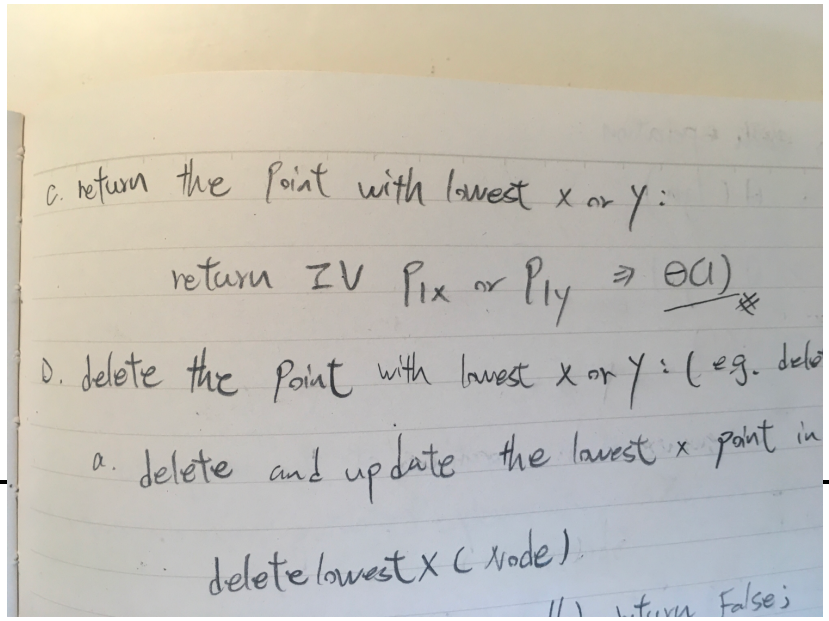
A. Describe the data structures you would use. Specifically, for any new data structures you need, write the class declaration. For any data structures from lectures/textbook that you would use, succinctly describe how you would use them and what modifications are needed (if any).



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.

Answer in previous page (A.)

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.



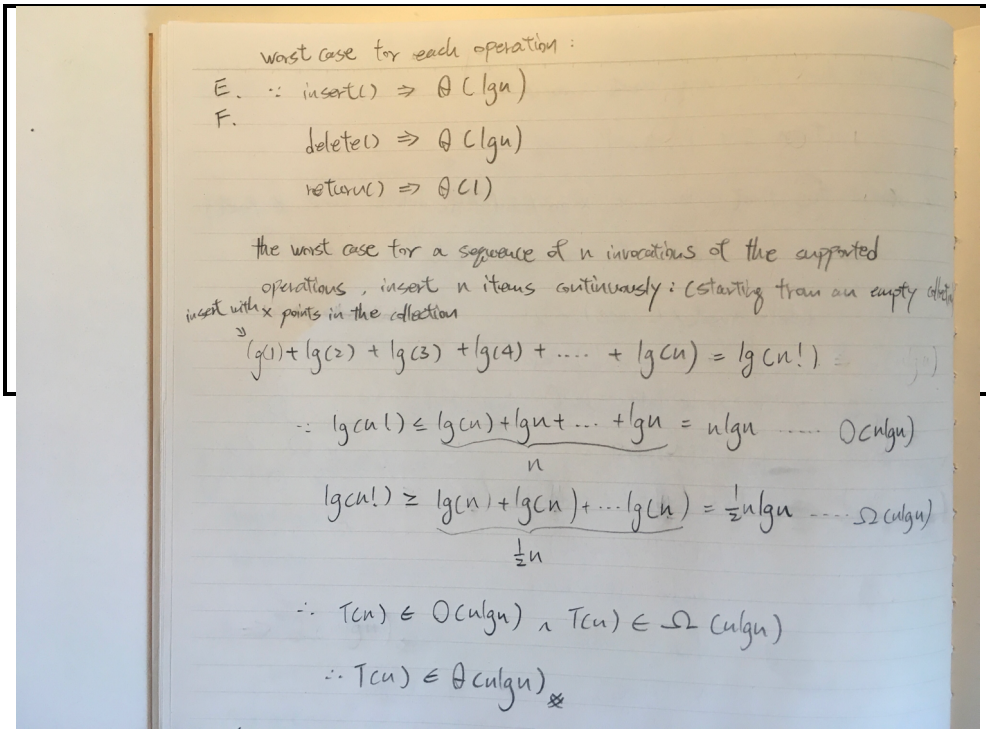
D. Give a concise English description of your algorithm for deleting the Point with the lowest *x*- or *y*-coordinate. Feel free to use some pseudocode if you think it will improve clarity.

answer in the last page.

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?

$\theta($ nlgn $)$

F. Explain your answer to part E.



O(1) ✱

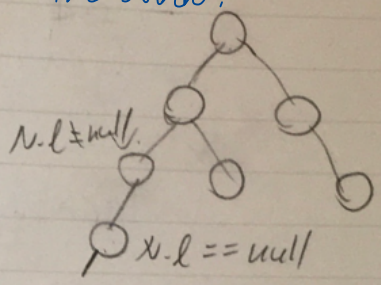
D. delete the Point with lowest x or y : (e.g. delete lowest x Point)

a. delete and update the lowest x point in x-tree :

```
deleteLowestX (xNode)
    if (xNode == null) return False;
    if (xNode.left == null) return True;
    leftChildIsLowest =
        deleteLowestX (xNode.left)
    if (leftChildIsLowest):
        xNode.left = null
        P1x = xNode
    return False;
```

* delete is similar to the normal balanced BST, but delete the point in the value point array first before delete the xNode.

the node should be deleted →



⇒ O(lgn) ✱

b. find and delete the Point with lowest x (can be saved in a temp variable) in the y-tree and delete it by using standard delete() in red-black tree. ⇒ O(lgn) ✱

• in case there are multiple points with the same y-coordinate, also compare the x-coordinate (as value in a y-tree) when find the target y-coordinate

c. if deleted Point's y-coordinate equals lowest y-coordinate, update P_{1y}. ⇒ O(1) ✱

⇒ for delete the Point with lowest y-coordinate, vice-versa.

7. (official solution):

a. Data structure

> 2 min-heap. one for x-coordinate priority another for y-coordinate priority:

```
public class Pointkey {
    private Point point;
    private boolean deleted;
    // has public methods return x, y coordinates or compare Pointkey based on x or y coordinate.
public class PointkeyX implements Comparable<PointkeyX>
    private Pointkey pk;
    // implement CompareTo() that compares x-c of pk (of other PointkeyX) }
public class PointkeyY implements Comparable<PointkeyY>
    private Pointkey pk;
    // ..  ..  y-c of pk (of othe PointkeyY) }
```

⇒ last 2 class are the keys for the 2 heaps, first one is the Point object referenced by keys in both heaps.

b. adding a point: $(\sim lgn \Rightarrow \theta(lgn))$

- create an new Pointkey with the inserted point, set the deleted to false.
- create new PointkeyX, PointkeyY reference to the Pointkey, and insert them in X-heap and y-heap, respectively.

c. return the lowest x or y-coordinate

- call min() on x or y heap and return the Point that Pointkey references.

$(\theta(1))$

d. deleting a point:

ie. delete the point with least x-coordinate.

```
pkx = xheap.delMin();
pkx.pk.deleted = true;
while (xheap.min().pk.deleted == true) {
    xheap.delMin();
}
while (yheap.min().pk.deleted == true) {
    yheap.delMin();
}
```

e.g. $\Theta(n \log n)$

∵ return method min() is constant: $\Theta(1)$ ⇒ ignored

n times insert/delete a point:

each point is inserted once into each heap, and each point deleted is deleted at most once from each tree (could be deleted in future operation). Therefore the # of compares is twice of the # required by n inserts/deletes into a single heap tree $\sim 2n \lg n$

∴ $\Theta(n \lg n)$ *

or $\Theta(n \log n)$