COS 226 **Algorithms and Data Structures** **Spring 2017**

Written Exam 1

J. Lumbroso

There are 10 questions on this exam, worth a total of 70 points. You have 80 minutes. There is one question per lecture, numbered corresponding to the lectures, The questions are *not in order of difficulty.* If a question seems difficult to you, skip it and come back to it.

Policies. The exam is closed book and closed Internet, except that you are allowed to use a one page cheatsheet (8.5-by-11 paper, one sides, in your own handwriting). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. You have 80 minutes to complete the test. **This exam is preprocessed by computer. If you use pencil (and eraser), write darkly. Write all answers inside the designated rectangles. Do not write on corner marks.**

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

This exam. Do not remove this exam from this room. In the space below, *print your name and NetID*; write and sign the Honor Code pledge.

"*I pledge my honor that I have not violated the Honor Code during this examination.*"

NONE

Name:

NetID:

Room:

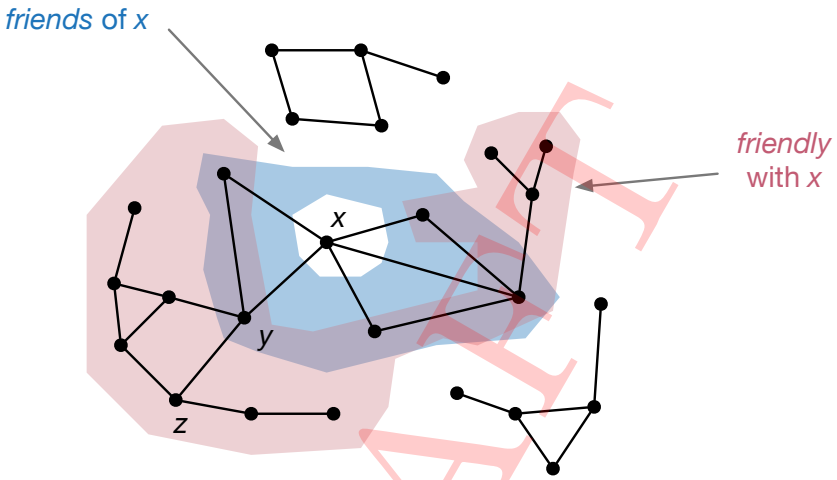
Precept: P01 ○ P02 ○ P02A ○ P03 ○ P03A ○ P04 ○ P04A ○ P04B ○ P05 ○ P05A ○

Q1. Union-Find (10 points).

(i) [6 pts] Consider the `WeightedQuickUnionUF.java` (WQUF) data structure is used for a social network in the following way:

- the sites of the WQUF data structure are *people* (you can assume they are stored with a name);
- the operation union(x, y) is called on people x and y when these two people are *friends of each other*.

Finally, if x and y are friends of each other, and y and z are friends of each other, then we consider that x and z are *friendly*. See the figure below for an illustrative example.

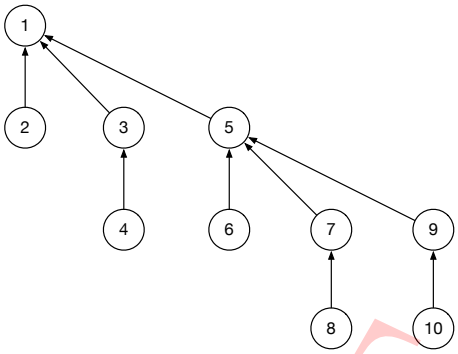


If N is the total number of person (or site of the WQUF data structure), which of the following questions can be answered in at most linear time in the worst case using *only* find queries to the WQUF data structure defined above?

Fill in a bubble on each line to indicate whether it is ***True*** *or* ***False*** *that each task can be addressed by writing a client method, which uses only makes calls to the public API of WQUF and which runs in linear time in the worst case.*

	True	False
A. Determine whether two distinct given people, x and y, are friends.	○	⊗
B. Determine whether two distinct given people, x and y, are *friendly* (see definition above).	⊗	○
C. Given a person, x, iterate over all people that are *friendly* with x and distinct from x (for instance to print the name of each person).	○	⊗

(ii) [4 pts] Consider the following tree representing the inner state of a `WeightedQuickUnionUF.java` (WQUF) data structure which has been modified to allow for *path compression* (as implemented in the code is provided in the course slides).

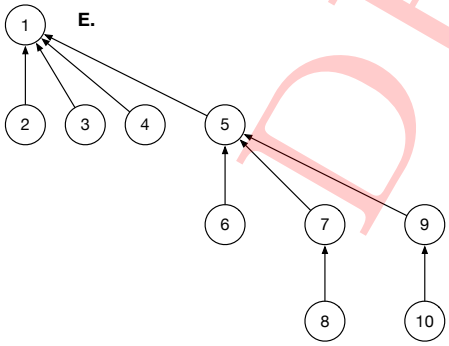


Which of the following diagrams can be obtained from the above WQUF data structure after making *exactly one* find operation? Each diagram is to be considered independently from the others.

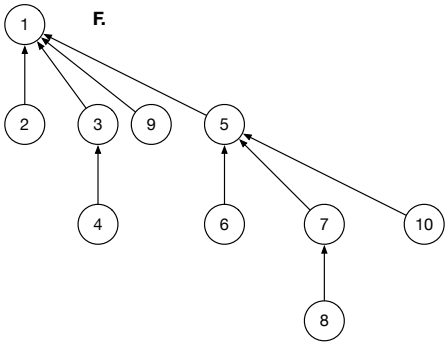
Fill in a bubble on each line to indicate whether it is ***True*** *or* ***False*** *that a tree can be the result of calling* find *exactly once on the above WQUF with path compression.*

	True	False
Diagram **E.**	⊗	○
Diagram **F.**	○	⊗
Diagram **G.**	⊗	○
Diagram **H.**	⊗	○

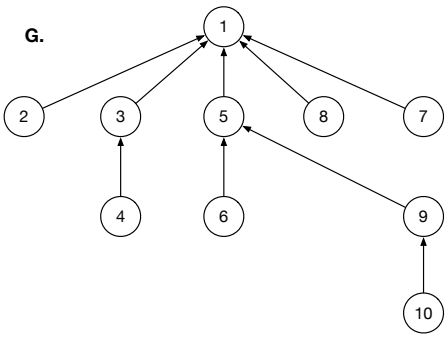
E.



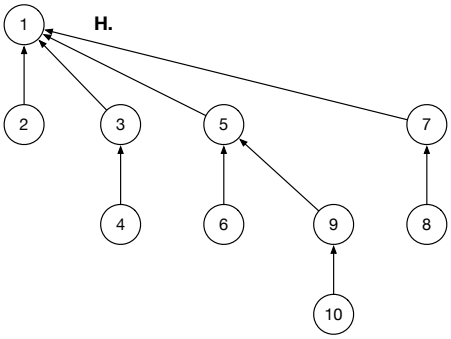
F.



G.



H.



Q2. Analysis of Algorithms (9 points).

We are interested in the problem of finding a pair of duplicate elements in an array containing N unsorted integers. We consider three different methods:

- METHOD I. We use two nested loops, to consider every pair of elements of the array. For each pair, we check to see if the elements are identical (and stop iterating over the pairs as soon as we have found a pair of duplicates).
- METHOD II. We sort the elements using quicksort (1 pivot, first element is pivot, no 3-way partitioning). Once the array is sorted, the duplicates contiguous, next to each other, so we simply iterate to find two adjacent elements that are identical (and stop iterating as soon as we have found a pair of duplicates).
- METHOD III. We use a hash table. We iterate over the array. For each element, we check if it is contained in the hash table: if it is not, we add it to the hash table; if it is, we have detected a pair of duplicates (and we may stop).

For each method, provide the order of growth of the *best case* run time (under the uniform hashing assumption).

	constant	logarithmic	linear	linearithmic	quadratic
A. Order of growth of best case of METHOD I	⊗	○	○	○	○
B. Order of growth of best case of METHOD II	○	○	○	⊗	○
C. Order of growth of best case of METHOD III	⊗	○	○	○	○

Same question, assuming now that *the array does not contain any duplicate.*

	constant	logarithmic	linear	linearithmic	quadratic
D. Order of growth of best case of METHOD I	○	○	○	○	⊗
E. Order of growth of best case of METHOD II	○	○	○	⊗	○
F. Order of growth of best case of METHOD III	○	○	⊗	○	○

Q3. Stacks and Queues (6 points).

Consider the following program. (Recall that `%` is the remainder operator; for example, `5 % 2` is equal to `1`.)

```
import edu.princeton.cs.algs4.Queue;
import edu.princeton.cs.algs4.Stack;
import edu.princeton.cs.algs4.StdOut;

public class Ordering {
    static final int MAX = 10;

    public static void main(String[] args) {
        Queue<Integer> queue = new Queue<Integer>();
        Stack<Integer> stack = new Stack<Integer>();

        for (int i = 0; i < MAX; i++) {
            if (i % 2 == 0)
                queue.enqueue(i);
            else
                stack.push(i);
        }


        for (int i = 0; i < MAX; i++) {
            int j;
            if (i % 2 == 0)
                j = queue.dequeue();
            else
                j = stack.pop();
            StdOut.print(j + " ");
        }
        StdOut.println();
    }
}
```

Write the sequence of numbers that the main method of the above class prints out?

0 9 2 7 4 5 6 3 8 1

Q4. Elementary Sorts (6 points).

(i) [2 pts] How many inversions does the permutation `4 1 3 2 5 6` contain?

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

(ii) [4 pts] Recall that the operations that we often count to measure performance are the array comparisons (which involve comparing two cells of an array) and array exchanges (which involve swapping the contents of two cells in an array).

Consider the following property:

Every exchange made by the sorting algorithm decreases the number of inversions of the sequence by exactly 1.

Indicate whether or not this property applies to each of the elementary sorts.

	True	**False**
A. Selection sort	○	⊗
B. Insertion sort	⊗	○
C. Shell sort	○	⊗

Q5. Mergesort (6 points).

(i) [3 pts] Below are several statements made about the standard (top-down) mergesort.

Fill in a bubble on each line to indicate whether each statement is ***True*** *or* ***False****.*

	True	False
A. Any two items are compared with one another no more than once during mergesort.	⊗	○
B. A single key can be involved in as many as $\sim N$ compares when mergesorting an array containing N distinct keys.	⊗	○
C. When mergesorting an array of N keys, the number of calls to merge() is $\sim N$. (Recall that merge() is called only on subarrays of length 2 or more.)	⊗	○

(ii) [3 pts] We now assume it is possible to merge two *sorted* sublists (of any size) in constant time[1].

Express the average running time, as a function of N, of the standard textbook 2-way mergesort, modified only to use this new (imaginary) constant-time merging algorithm.

N [order of growth]

[1]This assumption is physically impossible with our current knowledge in CS. We only make this assumption in the context of this exam question!

Q6. Quicksort (9 points).

Answer these questions about fully sorting an array using quicksort with 3-way partitioning.

(i) [3 pts] Suppose that the input is a randomly-ordered array with $9N$ elements, having $3N$ occurrences of the letter x and $6N$ occurrences of y, with the alphabetical order, x < y.

Fill in the one bubble on each row that best describes the number of compares used in each case. If you obtain a fractional coefficient, for the number of comparisons, round to the closest integer.

	$\sim 8N$	$\sim 9N$	$\sim 10N$	$\sim 11N$	$\sim 12N$	$\sim 13N$	$\sim 14N$	$\sim 15N$	$\sim 16N$
A. Best case	○	○	○	○	⊗	○	○	○	○
B. Average case	○	○	○	○	○	⊗	○	○	○
C. Worst case	○	○	○	○	○	○	○	⊗	○

(ii) [6 pts] Suppose that the input is a randomly-ordered array with $5N$ elements, having N occurrences of the letter x, N occurrences of y, and $3N$ occurrences of z, with the alphabetical order, x < y < z.

Fill in the one bubble on each row that best describes the number of compares used in each case. If you obtain a fractional coefficient, for the number of comparisons, round to the closest integer.

	$\sim 8N$	$\sim 9N$	$\sim 10N$	$\sim 11N$	$\sim 12N$	$\sim 13N$	$\sim 14N$	$\sim 15N$	$\sim 16N$
D. Best case	⊗	○	○	○	○	○	○	○	○
E. Average case	○	⊗	○	○	○	○	○	○	○
F. Worst case	○	○	○	○	⊗	○	○	○	○

Q7. Priority Queues / Heaps (7 points).

Consider the following implementation of the `MaxPQ` abstract data type, which is identical to the textbook's implementation with the exception of the method `delMax`, which has been modified.

```
1  public class MaxPQ<Key> implements Iterable<Key> {
2      private Key[] pq;                    // store items at indices 1 to n
3      private int n;                       // number of items on priority queue
4      // ...
5
6      /* "sink", "swim" and "resize" are the same implementations as the textbook */
7
8      private void swim(int k) { ... }
9
10     private void sink(int k) { ... }
11
12     private void resize(int capacity) {
13         Key[] temp = (Key[]) new Object[capacity];
14         for (int i = 1; i <= n; i++) {
15             temp[i] = pq[i];
16         }
17         pq = temp;
18     }
19
20     public Key delMax() {
21       if (isEmpty())
22         throw new NoSuchElementException("Priority queue underflow");
23       Key max = pq[1];
24       exch(1, n--);
25       swim(n);
26       pq[n+1] = null;
27       if ((n > 0) && (n == (pq.length - 1) / 4))
28         resize(pq.length / 2);
29       return max;
30     }
31 }
```

The method `delMax` contains a bug, which can be fixed by modifying a single line of code. *Please indicate below which line of code to modify, by filling the corresponding bubble.*

Line of code to modify

20	21	22	23	24	25	26	27	28	29	30
○	○	○	○	○	⊗	○	○	○	○	○

Please indicate below what you would replace the original line of code with.

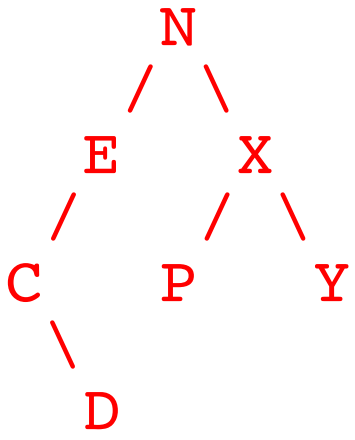
sink(1);

Q8. Binary Trees and BSTs (5 points).

If the in-order traversal of that binary tree prints nodes labeled `C D E N P X Y`, and the post-order traversal of a binary tree prints nodes labeled `D C E P Y X N`, then what sequences of labels does the pre-order traversal print?

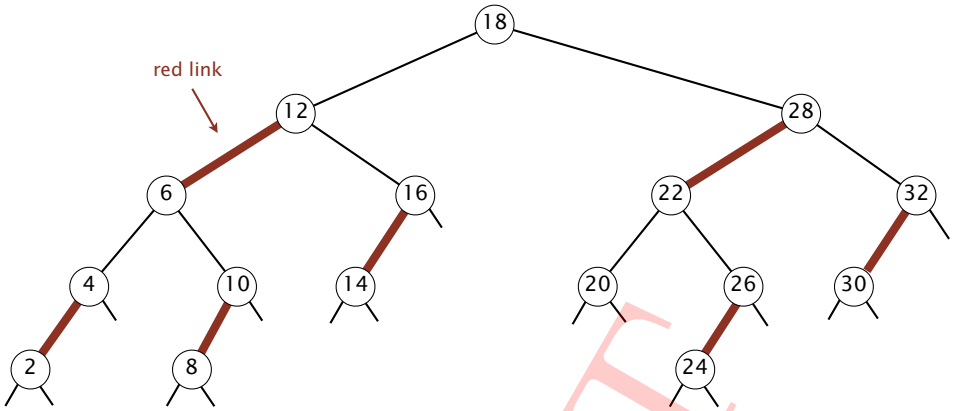
N E C D X P Y

(You may use the space below as scratch work. Make sure you only enter the sequence of labels in the above box.)



Q9. Left-Leaning Red-Black BSTs (4 points).

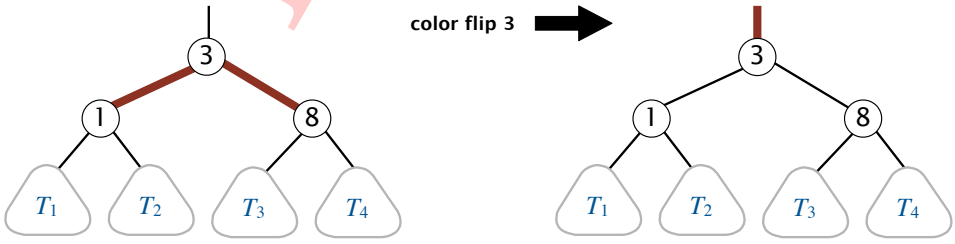
Consider the following left-leaning red-black BST.



Suppose that you insert the key 7 into this left-leaning red-black BST above.

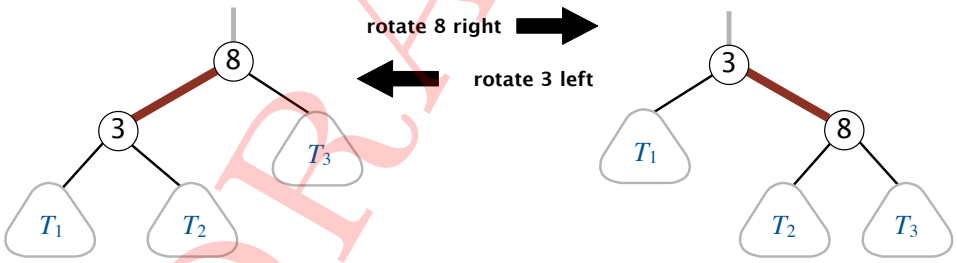
(a) Which of the following color flips result from the insertion of key 7 into the original LLRB? *Fill in all the boxes which apply.*

- ☐ Color flip 6
- ☐ Color flip 7
- ☒ Color flip 8
- ☐ Color flip 10
- ☐ Color flip 12
- ☐ Color flip 14
- ☐ Color flip 16
- ☐ Color flip 18



(b) Which of the following rotations result from the insertion of key 7 into the original LLRB? *Fill in all the boxes which apply.*

- [x] Rotate 6 left
- [ ] Rotate 6 right
- [ ] Rotate 7 left
- [ ] Rotate 7 right
- [ ] Rotate 8 left
- [ ] Rotate 8 right
- [ ] Rotate 10 left
- [x] Rotate 10 right
- [ ] Rotate 12 left
- [x] Rotate 12 right
- [ ] Rotate 14 left
- [ ] Rotate 14 right
- [ ] Rotate 16 left
- [ ] Rotate 16 right
- [ ] Rotate 18 left
- [ ] Rotate 18 right



Q10. Hashing (5 points).

Recall that hashing involves storing keys in a table, at the address computed by a *hash function*. When the hash function provides the same address for two different keys, we say that there is a *collision*.

Separate chaining and linear probing are two different strategies to address collisions. For each property below, indicate whether it is more characteristic of either separate chaining or linear probing.

	Separate chaining	Linear probing
A. Less wasted space.	○	⊗
B. Performance degrades gracefully.	⊗	○
C. The degradation caused by bad hash function tends to be amplified by a phenomenon called *clustering*.	○	⊗
D. Better cache performance.	○	⊗
E. Easier to implement delete.	⊗	○