COS 226	Algorithms and Data Structures	Fall 2015

Midterm Exam

You have 80 minutes for this exam. The exam is closed book, except that you are allowed to use one page of notes (8.5-by-11, one side, in your own handwriting). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. You may use the back of each page for scratch space, or to continue long answers.

Name:

NetID:

Room:

Precept:

P01	9:00	Andy Guna
P02	10:00	Andy Guna
P02A	10:00	Elena Sizikova
P03	11:00	Maia Ginsburg
P03A	11:00	Nora Coler
P04	12:30	Maia Ginsburg
P04A	12:30	Miles Carlsten
P05	1:30	Tom Wu

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

Problem	Score
0	
1	
2	
3	
4	
Sub 1	

Problem	Score
5	
6	
7	
8	
Sub 2	

Total:

0. Constructor. (1 point)

In the space provided on the front of the exam, write your name and Princeton netID; write the name of the room in which you are taking the exam; mark your precept number; and write and sign the honor code.

1. The Usual COS226 Sorting Question. (16 points)

The column on the left is an array of strings to be sorted or shuffled. The column on the right is in sorted order. The other columns are the contents of the array at some intermediate step during one of the algorithms below. Write the number of each algorithm under the corresponding column. Use each number exactly once.

mars	care	dart	barn	barn	barn	yard	bark	lard	bark
part	gary	lard	fare	care	care	warm	barn	care	barn
care	mars	care	rare	fare	dart	vary	card	gary	card
gary	part	gary	jars	gary	fare	part	care	barn	care
barn	barn	barn	harp	harp	gary	tart	dart	card	dart
park	fare	card	mars	jars	harp	park	earn	farm	earn
rare	park	farm	gary	mars	jars	rare	fare	fare	fare
fare	rare	fare	warm	park	mars	gary	farm	harm	farm
warm	harp	harm	care	part	park	mars	gary	earn	gary
tarp	jars	earn	tarp	rare	part	tarp	harm	jars	harm
jars	tarp	jars	part	tarp	rare	oars	harp	harp	harp
harp	warm	harp	park	warm	tarp	nary	jars	bark	jars
vary	bark	bark	vary	bark	vary	care	vary	dart	lard
dart	dart	mars	dart	dart	warm	dart	part	mars	mars
bark	vary	vary	bark	earn	bark	bark	mars	yard	nary
yard	yard	yard	yard	harm	yard	fare	yard	vary	oars
earn	earn	tarp	earn	vary	earn	earn	park	tarp	park
harm	farm	warm	harm	yard	harm	harm	tarp	warm	part
farm	harm	rare	farm	card	farm	farm	rare	tart	rare
tart	tart	tart	tart	farm	tart	barn	tart	rare	tarp
card	card	park	card	lard	card	card	warm	park	tart
lard	lard	part	lard	nary	lard	lard	lard	oars	vary
oars	nary	oars	oars	oars	oars	jars	oars	nary	warm
nary	oars	nary	nary	tart	nary	harp	nary	part	yard
----	----	----	----	----	----	----	----	----	----
0									9

(0) Original input

(1) Knuth shuffle

(2) Selection sort

(3) Insertion sort

(4) Mergesort *(top-down)*

(5) Mergesort *(bottom-up)*

(6) Heapsort

(7) Quicksort *(no shuffle)*

(8) 3-way Quicksort *(no shuffle)*

(9) Sorted

2. Playing Cards. (16 points)

We would like to sort playing cards from a deck. Associated with each card is a denomination (1 to 13) and a suit (CLUBS $<$ DIAMONDS $<$ HEARTS $<$ SPADES).

A card c_1 is considered less than a card c_2 if either of the following is true:

- the suit of c_1 is less than the suit of c_2, or
- c_1 and c_2 are of the same suit, but the denomination of c_1 is less than the denomination of c_2.

(a) Let us first consider sorting cards of the same suit, based purely on their denominations. Specifically, consider using ***2-way quicksort*** to sort the 3, 4, 5, 6, 7, 8 and 9 of hearts. After a random shuffle, we have the following sequence of denominations: 5, 6, 8, 3, 9, 4, 7. Show the result of the first call to `partition()` by giving contents of the array ***after each exchange***. Please write ***only the two elements that were exchanged***.

5	6	8	3	9	4	7

Now write the entire contents of the partitioned array, and ***draw a box*** around each of the left and right subarrays on which recursive calls will be executed.

(b) The `Card` class is implemented in Java as follows. Complete the `compareTo()` function, implementing the ordering described on the previous page, and assuming that the argument is not null.

```
public class Card implements Comparable<Card> {
    // Comparators by suit and by denomination
    public static final Comparator<Card> SUIT_ORDER = new SuitOrder();
    public static final Comparator<Card> DENOM_ORDER = new DenomOrder();

    // Suit of the card (CLUBS = 1, DIAMONDS = 2, HEARTS = 3, SPADES = 4)
    private final int suit;

    // Denomination of the card
    private final int denom;

    public Card(int suit, int denom) {
        if (suit < 1 || suit > 4)
            throw new IllegalArgumentException("Invalid suit");
        if (denom < 1 || denom > 13)
            throw new IllegalArgumentException("Invalid denomination");
        this.suit = suit;
        this.denom = denom;
    }

    // COMPLETE THE FOLLOWING FUNCTION
    public int compareTo(Card that) {









    }

    // Compare cards according to the suit only
    private static class SuitOrder implements Comparator<Card> {
        // Implementation not shown
    }

    // Compare cards according to the denomination only
    private static class DenomOrder implements Comparator<Card> {
        // Implementation not shown
    }
}
```

(c) Suppose that the variable `cards` is an array of cards. We could sort it, using your `compareTo` function, with a call to `MergeX.sort(cards)`. Which of the following code fragments would produce an equivalent final result? Circle ***all*** equivalent code fragments.

Option 1:

```
MergeX.sort(cards, Card.SUIT_ORDER);
MergeX.sort(cards, Card.DENOM_ORDER);
```

Option 2:

```
MergeX.sort(cards, Card.DENOM_ORDER);
MergeX.sort(cards, Card.SUIT_ORDER);
```

Option 3:

```
MergeX.sort(cards);
MergeX.sort(cards, Card.SUIT_ORDER);
```

Option 4:

```
MergeX.sort(cards, Card.DENOM_ORDER);
MergeX.sort(cards);
```

Option 5:

```
Quick.sort(cards, Card.SUIT_ORDER);
Quick.sort(cards, Card.DENOM_ORDER);
```

Option 6:

```
Quick.sort(cards, Card.DENOM_ORDER);
Quick.sort(cards, Card.SUIT_ORDER);
```

Option 7:

```
MergeX.sort(cards);
Quick.sort(cards, Card.SUIT_ORDER);
```

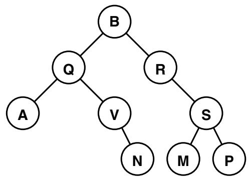
Option 8:

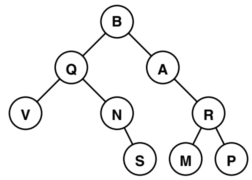
```
Quick.sort(cards, Card.DENOM_ORDER);
MergeX.sort(cards);
```

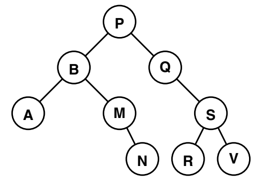
3. Traversing Trees. (10 points)

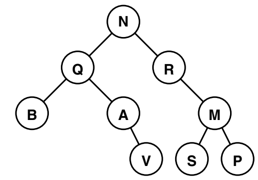
(a) Circle the correct ***binary tree*** (not necessarily a BST) that would produce both of the following traversals:

In-order: AQVNBRMSP
Pre-order: BQAVNRSMP



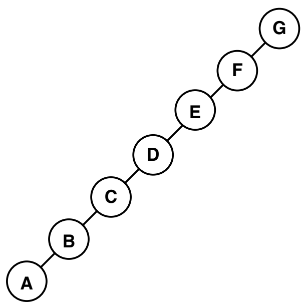


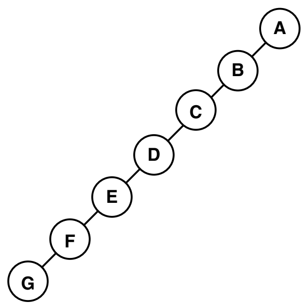


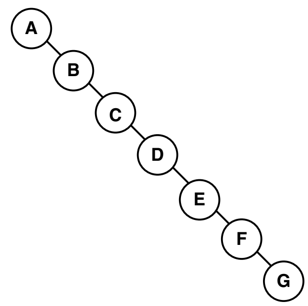


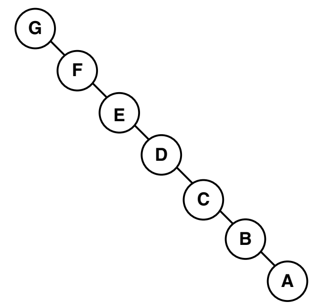
(b) Circle the correct ***Binary Search Tree*** that would produce the following traversal:

Post-order: ABCDEFG









(c) If you know that a tree is a BST, which of the following ***is*** or ***is not*** always sufficient to reconstruct it? For each one, write ***yes*** if it is enough to reconstruct the tree, or ***no*** if it is not.

Pre-order traversal:

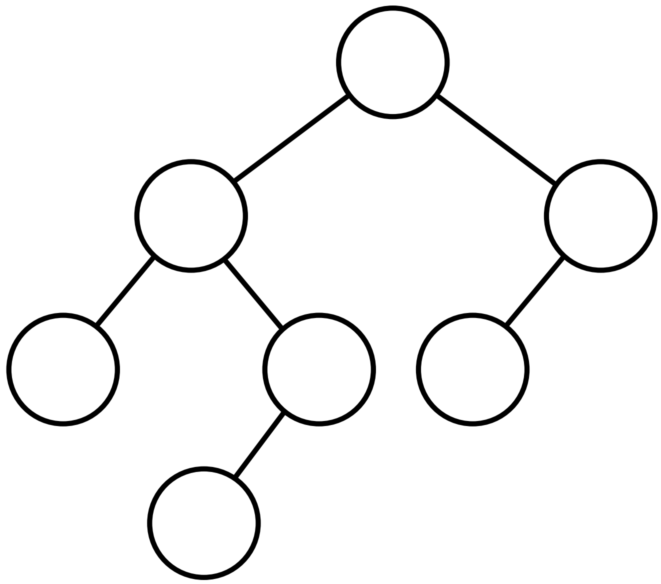
In-order traversal:

Post-order traversal:

Level-order traversal:

4. BSTs, LLRB and otherwise. (12 points)

(a) Label each node in the following binary tree with numbers from the set $\{2, 26, 10, 27, 20, 15, 42\}$ so that it is a legal Binary Search Tree. (Hint: use the back of the page as scratch space, and only write down the answer once you have it.)



(b) Now label each *edge* in the figure with `r` or `b`, denoting RED and BLACK, so that the tree is a legal Left-Leaning Red-Black Tree.

(c) Considering your labeling in (b), is it possible to assign *different* red/black labels and still satisfy the LLRB-tree conditions? (Answer ***yes*** if a different labeling is possible, or ***no*** if your labeling in (b) is unique.)

(d) If the answer to (c) is ***yes***, draw and label the second tree. If the answer is ***no***, how do you know that the red-black labeled tree must be unique?

5. Heaps. (10 points)

Starting from the following max-heap (using the array representation presented in lecture), give the resulting array after each operation:

X	10	7	4	5	6	2	3	0	1	

(a) After `insert(9)`

X										

(b) After `delMax()`, starting from the original heap (i.e., assuming that (a) has ***not*** been performed)

X										

(c) For implementing a max-priority queue, which of the following are advantages of a resizing-array implementation of a heap ***over*** a sorted linked list? Circle ***all*** that apply.

expected time for `insert` is lower

`insert` has lower worst-case order of growth

expected time for `delMax` is lower

`delMax` has lower worst-case order of growth

expected storage cost is lower

`max` has lower worst-case order of growth

6. FortyTwoPQ. (15 points)

You have been hired by Deep Thought Enterprises to implement a priority-queue-like data structure supporting the following operations:

- `insert()` an item in $O(\log N)$ time.
- `fortytwo()` — return the 42^{nd} smallest item in constant time.
- `delFortyTwo()` — delete the 42^{nd} smallest item in $O(\log N)$ time.

Explain how you would implement the required functionality, using one or more data structures that we have seen in class. Write pseudocode for each of the three operations listed above. You may assume that $N > 42$, and omit all checks for smaller N.

For full credit, your implementation should support finding the k^{th} smallest item with an order-of-growth running time ***independent*** of k. That is, it should be possible to change 42 to some other constant (at compile time) without changing the order-of-growth running time.

If you need more space, use the back of the sheet.

7. Divide and Conquer. (12 points)

Consider the following three algorithms:

- **Algorithm 1** solves problems of size N by recursively dividing them into 2 sub-problems of size $N/2$ and combining the results in time c (where c is some constant).
- **Algorithm 2** solves problems of size N by solving one sub-problem of size $N/2$ and peforming some processing taking some constant time c.
- **Algorithm 3** solves problems of size N by solving two sub-problems of size $N/2$ and performing a *linear* amount (i.e., cN where c is some constant) of extra work.

(a) For each algorithm, write down a *recurrence relation* showing how $T(N)$, the running time on an instance of size N, depends on the running time of a smaller instance.

Algorithm 1: $T(N) =$

Algorithm 2: $T(N) =$

Algorithm 3: $T(N) =$

(b) For each recurrence relation, pick the solution for $T(N)$ from the following list. Just write the letter corresponding to the correct running time.

Algorithm 1:

Algorithm 2:

Algorithm 3:

A: $T(N) \sim c$
B: $T(N) \sim c \log N$
C: $T(N) \sim cN$
D: $T(N) \sim cN \log N$
E: $T(N) \sim cN^2$

(c) For each of the following algorithms, pick which of the above classes of algorithms (1, 2, or 3) applies to that algorithm:

Mergesort:

Binary search in a sorted array:

Quicksort (if partitioning always divides the array in half):

8. You didn't think we forgot about the assignments, did you? (8 points)

(a) Suppose we wanted to simulate percolation in a *cube* with N sites on a side, with each site connected to its neighbors up, down, left, right, forward, and back. If we used `WeightedQuickUnionUF`, what would be the order of growth of the expected running time, as a function of N?

a. N^2
b. $N^2 \log N$
c. N^3
d. $N^3 \log N$
e. N^4
f. $N^4 \log N$
g. None of the above.

(b) If you run your `BinarySearchDeluxe` on a sorted array with N items but only 3 distinct keys, what is the order of growth of the expected running time for a call to `firstIndexOf()`?

a. constant
b. $\log N$
c. $\log_N 3$
d. N
e. $N \log N$
f. None of the above.

(c) True or False: The amount of memory necessary to solve `8puzzle` is equal to some constant times the size of the game board.

(d) True or False: `8puzzle` will still work without implementing the critical optimization, but it may take much more memory and running time to find the answer.

(e) True or False: it is always legal to call `equals` on two objects that do not have the same type.

(f) True or False: a `KdTreeST` always has a lower order-of-growth running time than the brute-force `PointST` for the `contains()` operation, for all possible query points.

(g) True or False: a `KdTreeST` always has a lower order-of-growth running time than the brute-force `PointST` for the `range()` operation, for all possible query rectangles.

(h) True or False: a `KdTreeST` always has a lower order-of-growth running time than the brute-force `PointST` for the `nearest()` operation, for all possible query points.