COS 226 Algorithms and Data Structures Spring 2015

Midterm

This test has 9 questions worth a total of 55 points. You have 80 minutes. The exam is closed book, except that you are allowed to use a one page cheatsheet. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. **Write out and sign the Honor Code pledge just before turning in the test.**

Again, because this exam is preprocessed by computer: make sure your writing is dark, do not write any answers outside of the designated frames, and do not write on the corner marks.

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

Name:

netID:

Room:

Precept: P01 P01A P02 P03 P04 P05 P05A P06 P06A P06B P07

Problem	Score
0	
1	
2	
3	
4	
Sub 1	

Problem	Score
5	
6	
7	
8	
Sub 2	

Total	

P01	Th 11	Andy Guna
P01A	Th 11	Shivam Agarwal
P02	Th 12:30	Andy Guna
P03	Th 1:30	Swati Roy
P04	F 10	Robert MacDavid
P05	F 11	Robert MacDavid
P05A	F 11	Shivam Agarwal
P06	F 2:30	Jérémie Lumbroso
P06A	F 2:30	Ryan Beckett
P06B	F 2:30	Josh Wetzel
P07	F 3	Jérémie Lumbroso

0. **Initialization. (1 point)**

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

1. **Memory and Data Structures. (3 points)**

Using the 64-bit memory cost model from lecture, how many bytes does each `SearchNode` object use?

```
public class SearchNode implements Comparable<SearchNode> {
    private int moves;           // number of moves to get to this node
    private int priority;        // cache of the priority
    private Board board;         // the current board
    private SearchNode prev;     // the previous node

    // [constructor and instance methods omitted]
}
```

When an object is referenced, do not include the memory for the object, but do include the memory for the reference to the object. Also, note that `SearchNode` is a top-level class (and not a nested class).

 bytes

2. **Eight Sorting Algorithms and a Shuffling Algorithm. (8 points)**

The column on the left is the original input of strings to be sorted or shuffled; the column on the right are the strings in sorted order; the other columns are the contents at some intermediate step during one of the algorithms listed below. Match up each algorithm by writing its number under the corresponding column. Use each number exactly once.

,										
0	cyan	cafe	bark	bole	gold	cyan	bark	bole	aqua	aqua
1	cafe	aqua	cafe	bone	flax	bone	bole	bone	bark	bark
2	ruby	bole	aqua	cafe	dust	lava	bone	cafe	bole	bole
3	bole	bone	bole	cyan	fawn	bole	buff	cyan	bone	bone
4	flax	buff	cyan	flax	cyan	fawn	cafe	fawn	buff	buff
5	bone	bark	bone	rose	bole	flax	cyan	flax	cafe	cafe
6	rose	cyan	buff	ruby	cafe	cafe	fawn	gold	cyan	cyan
7	sage	cyan	cyan	sage	buff	jade	flax	jade	cyan	cyan
8	fawn	leaf	fawn	bark	cyan	gray	gold	leaf	dust	dust
9	leaf	gold	leaf	buff	bone	rose	gray	rose	fawn	fawn
10	gold	jade	gold	fawn	aqua	buff	jade	ruby	flax	flax
11	jade	lava	jade	gold	bark	gold	lava	sage	gold	gold
12	lava	fawn	lava	gray	gray	sage	leaf	bark	gray	gray
13	bark	gray	sage	jade	jade	leaf	rose	buff	jade	jade
14	gray	sage	gray	lava	kobi	ruby	ruby	cyan	kobi	kobi
15	buff	kobi	rose	leaf	lava	bark	sage	gray	lava	lava
16	kobi	rose	kobi	aqua	leaf	kobi	kobi	kobi	ruby	leaf
17	cyan	dust	flax	cyan	mint	cyan	cyan	lava	rose	mint
18	dust	silk	dust	dust	palm	dust	dust	dust	leaf	palm
19	silk	palm	silk	kobi	rose	silk	silk	silk	silk	rose
20	palm	sand	palm	mint	ruby	palm	palm	palm	palm	ruby
21	sand	flax	sand	palm	sage	sand	sand	sand	sand	sage
22	aqua	mint	ruby	sand	sand	aqua	aqua	aqua	sage	sand
23	mint	ruby	mint	silk	silk	mint	mint	mint	mint	silk
	----	----	----	----	----	----	----	----	----	----

0 9

(0) Original input

(1) Selection sort

(2) Insertion sort

(3) Mergesort (*top-down*)

(4) Mergesort (*bottom-up*)

(5) Quicksort (*standard, no shuffle*)

(6) Quicksort (*Dijkstra 3-way, no shuffle*)

(7) Heapsort

(8) Knuth shuffle

(9) Sorted

3. **Analysis of Algorithms and Sorting. (8 points)**

Consider the following function that sorts an array of N comparable keys.

```
public static void sort(Comparable[] a) {
    int N = a.length;
    for (int i = 0; i < N; i++)          // outer loop
        for (int j = N-1; j > i; j--)
            if (less(a[j], a[j-1]))
                exch(a, j, j-1);
}
```

(a) Which of these invariants does the above code satisfy at the end of each outer `i` loop? Check all that apply:

- ☐ Entries `a[0]` through `a[i]` are in sorted order.
- ☐ Entries `a[0]` through `a[i]` contain the smallest keys in the entire array.
- ☐ Entries `a[N-i-1]` through `a[N-1]` are in sorted order.
- ☐ Entries `a[N-i-1]` through `a[N-1]` contain the largest keys in the entire array.

(b) Which of the following are properties of the sorting algorithm? Check all that apply.

- ☐ The sorting algorithm is stable.
- ☐ The sorting algorithm is in-place.

(c) How many *compares* does `sort()` make to sort an array of N keys in the *best case*?

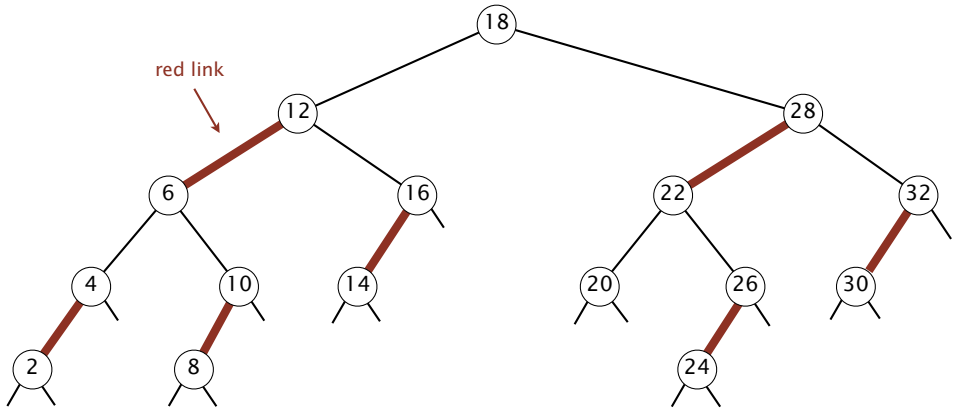
~ 0	$\sim N$	$\sim \log_2 N$	$\sim 1/4 \cdot N^2$	$\sim 1/2 \cdot N^2$	$\sim 3/4 \cdot N^2$
☐	☐	☐	☐	☐	☐

(d) How does the number of *exchanges* made by `sort()` relate to the number of exchanges made by insertion sort? Check all that apply.

- ☐ Insertion sort makes strictly more exchanges for some arrays.
- ☐ Insertion sort makes strictly fewer exchanges for some arrays.
- ☐ Insertion sort makes strictly more exchanges for all arrays.
- ☐ Insertion sort makes strictly fewer exchanges for all arrays.
- ☐ Insertion sort makes exactly the same number of exchanges for all arrays.

4. **Red-Black BSTs. (6 points)**

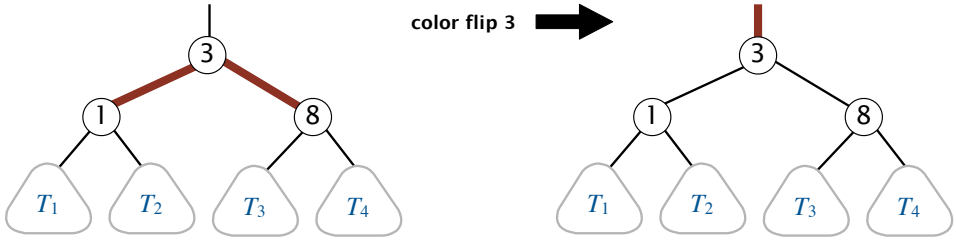
Consider the following left-leaning red-black BST.



Suppose that you insert the key 23 into this left-leaning red-black BST above.

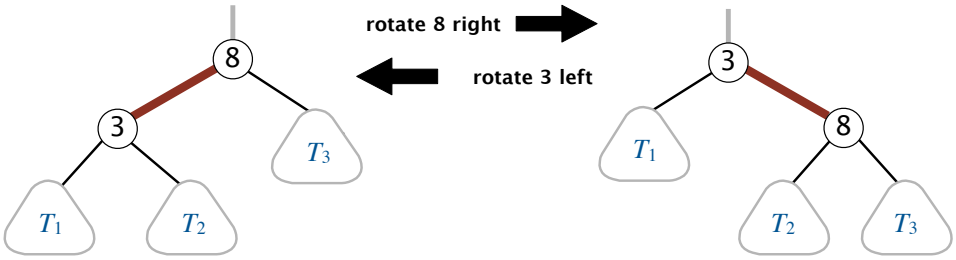
(a) Which of the following color flips result? Check all that apply.

- ☐ Color flip 18
- ☐ Color flip 22
- ☐ Color flip 23
- ☐ Color flip 24
- ☐ Color flip 26
- ☐ Color flip 28



(b) Which of the following rotations result? Check all that apply.

- ☐ Rotate 18 left
- ☐ Rotate 18 right
- ☐ Rotate 22 left
- ☐ Rotate 22 right
- ☐ Rotate 23 left
- ☐ Rotate 23 right
- ☐ Rotate 24 left
- ☐ Rotate 24 right
- ☐ Rotate 26 left
- ☐ Rotate 26 right
- ☐ Rotate 28 left
- ☐ Rotate 28 right



5. **Hash Tables. (6 points)**

(a) What is a *collision* in a hash table implementation of a symbol table? Check the best definition.

- ○ Two key-value pairs that have equal keys but different values.
- ○ Two key-value pairs that have different keys and hash to different indices.
- ○ Two key-value pairs that have different keys but hash to the same index.
- ○ Two key-value pairs that have equal keys but hash to different indices.

(b) A linear-probing hash table of length 10 uses the hash function $h(x) = x \bmod 10$. After inserting six integer keys into an initially empty hash table, the array of keys is:

0	1	2	3	4	5	6	7	8	9
		42	23	34	52	46	33		

Which of the following choice(s) are insertion sequences resulting in the above hash table? Assume that the length of the hash table does not change during the insertions. Check all that apply.

- ☐ 46, 42, 34, 52, 23, 33
- ☐ 34, 42, 23, 52, 33, 46
- ☐ 46, 34, 42, 23, 52, 33
- ☐ 42, 46, 33, 23, 34, 52
- ☐ 42, 23, 34, 52, 46, 33

(c) Suppose that your hash function does *not* satisfy the *uniform hashing assumption*. Which of the following can result? Check all that apply.

- ☐ Poor performance for *insert*.
- ☐ Poor performance for *search hit*.
- ☐ Poor performance for *search miss*.
- ☐ Uneven distribution of lengths of chains in separate-chaining hash table.
- ☐ Large clusters in linear-probing hash table.
- ☐ Linear-probing hash table can become 100% full.

6. **Properties of Data Structures. (6 points)**

Check whether each of the following statements is *true* or *false*.

(a) **Binary heaps.**

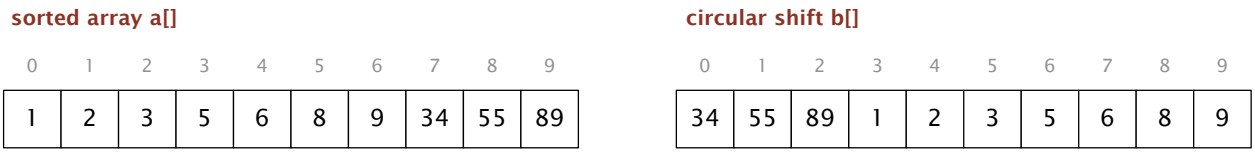
	True	False
Let `a[]` be a max-oriented binary heap that contains the N distinct integers $1, 2, \ldots, N$ in `a[1]` through `a[N]`. Then, key N must be in `a[1]`; key $N-1$ must be in either `a[2]` or `a[3]`; and key $N-2$ must be in either `a[2]` or `a[3]`.	○	○
The order of growth of the total number of compares to insert N distinct keys in *descending order* into an initially empty max-oriented binary heap is N.	○	○
A *3-heap* is an array representation (using 1-based indexing) of a complete 3-way tree, where the key in each node is greater than (or equal to) its children's keys. In the worst case, the number of compares to insert a key in a 3-heap containing N keys is $\sim 1 \log_3 N$.	○	○

(b) **Binary search trees.**

	True	False
It is possible to insert any batch of N keys into an initially empty binary search tree using at most $2N$ compares.	○	○
The height of any left-leaning red-black BST with N keys is always between $\sim 1 \log_2 N$ and $\sim 2 \log_2 N$.	○	○
The subtree rooted at any node of a 2-3 tree is itself a 2-3 tree.	○	○

7. **Algorithm Design. (8 points)**

Let $a = a_0, a_1, \ldots, a_{N-1}$ be an array of length N. An array b is a *circular shift* of a if it consists of the subarray $a_k, a_{k+1}, \ldots, a_{N-1}$ followed by the subarray $a_0, a_1, \ldots, a_{k-1}$ for some integer k. In the example below, b is a circular shift of a (with $k = 7$ and $N = 10$).



Suppose that you are given an array b that is a circular shift of some *sorted* array (but you have access to neither k nor the sorted array). Assume that the array b consists of N comparable keys, no two of which are equal. Design an efficient algorithm to determine whether a given key appears in the array b.

The order of growth of the running time of your algorithm should be $\log N$ (or better) in the worst case, where N is the length of the array.

Briefly describe your algorithm, using either crisp and concise prose or code. Your answer will be graded on correctness, efficiency, and clarity.

8. **Data Structure Design. (9 points)**

Design an efficient data type to store a *threaded set of strings*, which maintains a set of strings (no duplicates) and the order in which the strings were inserted, according to the following API:

`public class ThreadedSet`	
`ThreadedSet()`	*create an empty threaded set*
`void add(String s)`	*add the string to the set (if it is not already in the set)*
`boolean contains(String s)`	*is the string s in the set?*
`String previousKey(String s)`	*the string added to the set immediately before s (null if s is the first string added; exception if s is not in set)*

Here is an example:

```
ThreadedSet set = new ThreadedSet();
set.add("aardvark");              // [ "aardvark" ]
set.add("bear");                  // [ "aardvark", "bear" ]
set.add("cat");                   // [ "aardvark", "bear", "cat" ]
set.add("bear");                  // [ "aardvark", "bear", "cat" ]
                                  // (adding a duplicate key has no effect)

set.contains("bear");             // true
set.contains("tiger");            // false
set.previousKey("cat");           // "bear"
set.previousKey("bear");          // "aardvark"
set.previousKey("aardvark");      // null
```

Your answer will be graded on correctness, efficiency, and clarity.

(a) Declare the instance variables for your `ThreadedSet` data type. You may declare nested classes or use data types that we have considered in this course.

```
public class ThreadedSet {
```

```
}
```

(b) Briefly describe how to implement each of the operations, using either crisp and concise prose or code.

- **void** `add(String s)`:

- **boolean** `contains(String s)`:

- `String previousKey(String s)`:

(c) Under reasonable technical assumptions, what is the order of growth of each of the methods as a function of the number of keys N in the data structure? Assume that the length of all strings is bounded by a constant.

	1	$\log N$	$\sqrt{N}$	N	$N \log N$	N^2
`add()`	○	○	○	○	○	○
`contains()`	○	○	○	○	○	○
`previousKey()`	○	○	○	○	○	○