University of California at Berkeley
Department of Electrical Engineering and Computer Sciences
Computer Science Division

**Spring 2005**                                                                    **Jonathan Shewchuk**

# CS 61B: Final Exam

This is an open book, open notes exam. Electronic devices are forbidden on your person, including cell phones and laptops. Please put them on the desk at the front of the room (and turn your cell phone off) or risk losing a point. **Do not open your exam until you are told to do so!**

**Name:**

**Login:**

**Lab TA:**

**Lab time:**

*Do not write in these boxes.*

| Problem # | Possible | Score |
|---|---|---|
| 1. Miscellany | 8 | |
| 2. Data structures | 6 | |
| 3. Sorting | 12 | |
| 4. Augmented trees | 5 | |
| 5. Asymptotic analysis | 6 | |
| 6. Amortized analysis | 6 | |
| 7. Index sort | 7 | |
| Total | 50 | |

**Problem 1.** (8 points) **A Miscellany.**

a. (2 points) Consider the following method to make all the nodes in a doubly-linked list (circularly linked, with sentinel) available for garbage collection. Recall that `head` references the sentinel. Assume that listnodes are referenced *only* by other listnodes in the same list.

```
public void garbageList() {
  if (size > 0) {
    head.next.prev = null;  //  ?  Erase references to sentinel.
    head.prev.next = null;  //  ?
  }
  head.next = null;         //  ?  Erase references to other nodes.
  head.prev = null;         //  ?
  head = null;
}
```

Are the lines marked with question marks necessary to ensure that all the listnodes are available to be garbage collected? Explain.

b. (1 point) Recall that a `SimpleBoard` object stores an $8 \times 8$ array `grid` in which each cell has value 0, 1, or 2. Suppose you want to store lots of `SimpleBoard`s in a hash table. Can you think of a reason why the following hash code will not distribute SimpleBoard objects evenly among the buckets? (Assume the compression function is good.)

```
public class SimpleBoard {
  private int[][] grid;

  public int hashCode() {
    int code = 0;
    for (int i = 0; i < 8; i++) {
      for (int j = 0; j < 8; j++) {
        code = code + (i + j) * grid[i][j];
      }
    }
    return code;
  }
}
```

c. (2 points) Every word falls into one of eight categories: nouns, verbs, pronouns, adjectives, adverbs, prepositions, conjunctions, and interjections. Suppose you want to be able to look up the category of any English word in $\mathcal{O}(1)$ time. You could use a hash table to map each word to a category. The hash table's chains could use `ListNodes`, wherein each `ListNode` contains a reference to a word, and an extra field (perhaps a `short`) that indicates the category of the word.

However, we want to use as little memory as possible. We can save some memory by eliminating the extra field from each `ListNode`. Explain how we can do this, yet still determine a word's category in $\mathcal{O}(1)$ time. Hint: you can adjust the data structure in some other way, but don't increase the memory use by more than a small constant. (That's an added constant, not a constant factor.)

d. (1 point) What does the following Java code print?

```java
try {
  int[][] x = new int[5][5];
  Object y = x[5];
} catch (ClassCastException e1) {
  System.out.println("First");
  throw new NullPointerException();
} catch (ArrayIndexOutOfBoundsException e2) {
  System.out.println("Second");
  throw new NullPointerException();
} catch (NullPointerException e3) {
  System.out.println("Third");
} finally {
  System.out.println("Fourth");
}
```

e. (1 point) Circle the sorting algorithms whose **asymptotic** running time is affected by the input keys (their values and their order). Do not circle algorithms whose worst-case and best-case asymptotic running times are the same.

mergesort     bucket sort     insertion sort     selection sort     quicksort (with first item as pivot)

f. (1 point) Suppose you insert fifteen keys, the integers 1 through 15, into a splay tree in an arbitrary order, and you happen to get a perfectly balanced tree. The last key inserted was _____.
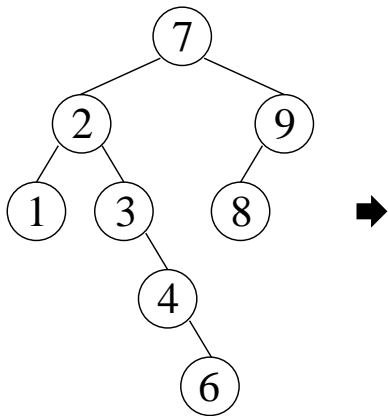
**Problem 2.** (6 points) **Operations on Data Structures.**

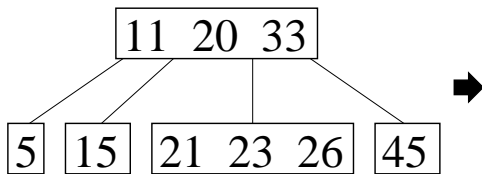a. (2 points) What does the following binary heap look like after `insert(1)`?
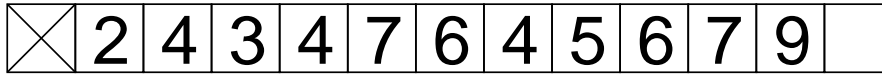
| ✕ | 2 | 4 | 3 | 4 | 7 | 6 | 4 | 5 | 6 | 7 | 9 | |

| ✕ | | | | | | | | | | | | | | ⬅ Fill in your answer here

b. (2 points) Draw the following splay tree after execution of the operation `remove(6)`.

```
            7
          /   \
         2     9
        / \   /
       1   3 8
            \
             4
              \
               6
```
➡

c. (2 points) Draw the following 2-3-4 tree after execution of the operation `insert(30)`.

```
        | 11  20  33 |
       /     |    |     \
    | 5 | | 15 | | 21 23 26 | | 45 |
```
➡

**Problem 3.**   (12 points) **Sorting.**

a. (3 points) Show how array-based quicksort sorts the array 5  9  7  4  0  2  8  8. Always choose the last key in an array (or subarray) to be the pivot. Draw the array once for each swap.

b. (1 point) Consider a list-based quicksort that computes the average (mean) value $i$ of all the keys in the list, then chooses the pivot to be $i + 1$. As usual, we partition the list into three separate lists: keys less than $i + 1$, keys equal to $i + 1$ (of which there might be none), and keys greater than $i + 1$. We sort the first and last lists recursively, then concatenate the three lists together. What's wrong with this algorithm?

c. (2 points) Consider the following sorting algorithm. Loop through the list of input items, inserting each item in turn into an ordinary binary search tree. Then, perform an inorder traversal of the tree, and output the items in inorder.

The worst-case running time of this algorithm is in $\Theta($_____$)$.

The best-case running time of this algorithm is in $\Theta($_____$)$.

d. (2 points) Describe how to modify the tree in part (c) so that the algorithm does **both** of the following.

- it runs in $\mathcal{O}(n \log n)$ worst-case time, and
- it runs in $\mathcal{O}(n)$ time if the input array is already sorted.

Explain why it runs in $\mathcal{O}(n)$ time in the latter case.

e. (2 points) Suppose you are using radix sort to sort a list of Java `ints`, and it is important to sort them as quickly as possible. Explain why you would **never** choose to use exactly 512 buckets for this sort.

f. (2 points) Professor Pretentious claims to have a comparison-based sorting algorithm that can take a valid binary heap, and sort its items in $\mathcal{O}(n)$ time. The good Professor claims that he gets around the $\Omega(n \log n)$-time lower bound for comparison-based sorting by taking advantage of the fact that the items in a binary heap are already "partly sorted." Give a convincing argument that the Professor's claims are bogus.

g. (2 **bonus points**) Suppose you have a **directed** graph $G = (V, E)$ represented by an adjacency list. You want to "sort" the vertices in $V$ in an ordering such that for every edge $(v, w) \in E$, vertex $v$ comes before vertex $w$. Note that there may be many such orderings, or (if the graph has a directed loop) none at all.

Suggest (in English) an algorithm that produces such an order in $\mathcal{O}(|V| + |E|)$ time if one exists. Hint: you may add extra fields to the Vertex objects if you like.

**Problem 4.** (5 points) **Augmented Binary Search Trees.**

The *rank* of a key $k$ in a set $S$ is the number of keys in $S$ that are less than or equal to $k$ (including $k$ itself). We want to store $S$ in an ordinary binary search tree so that we can determine the rank of any key in it in $\mathcal{O}(h)$ time, where $h$ is the height of the tree. Note that because $S$ is a set, there are **no duplicate keys** in the tree. Note also that you should **not** perform rotations or otherwise restructure the tree.

a. (1 point) What information should you store in each node of the tree?

b. (2 points) Let $k$ be a key in $S$. Describe an algorithm that determines the rank of $k$ in $\mathcal{O}(d)$ time, where $d$ is the depth of $k$ in the tree.

c. (2 points) Describe an algorithm that inserts a new key $k$ in $\mathcal{O}(h)$ time, where $h$ is the height of the tree. If $k$ is already in the tree, your algorithm should change nothing (as duplicate keys are not allowed). Otherwise, your algorithm must make all necessary adjustments to the information stored in the tree to ensure that your algorithm in part (b) still works.

**Problem 5.**   (6 points) **Asymptotic Analysis.**

a. (2 points) Rewrite the expression $\mathcal{O}(8x \log_7 \frac{y^4}{8} + \max\{3y^4 \log^9 y, 5\sqrt{x}\} + 2^{2y} + 10x \sin x)$ in the simplest possible form. (No proof is required. Assume $x \geq 2$ and $y \geq 2$.)

b. (4 points) Prove formally and rigorously, omitting no details, that $x^2 + 10xy + y^2 \in \mathcal{O}(x^2 + y^2)$. (Both $x$ and $y$ are positive variables that can grow arbitrarily large.)

**Problem 6.** (6 points) **Amortized Analysis.**

Consider a queue data structure that supports four operations.

```
public boolean isEmpty();
public void enqueue(Object item);
public Object dequeue() throws EmptyQueueException;
public void multiDequeue(int i) throws EmptyQueueException;
```

The first three are standard queue operations. The multiDequeue operation takes an integer i and de-queues i items (without returning any of them).

Show that the operations enqueue, dequeue, and multiDequeue run in $\mathcal{O}(1)$ *amortized* time, even though any single multiDequeue operation runs in $\Theta(i)$ time.

   a. (2 points) Suppose that it costs $1 of actual time to enqueue an item, and $1 of actual time to dequeue a *single* item and possibly return it or throw an exception. Assign an *amortized cost* to each of the three operations enqueue, dequeue, and multiDequeue, such that your bank balance will never become negative. For full marks, give the **smallest** possible amortized costs.

```
enqueue:        $
dequeue:        $
multiDequeue:   $
```

   b. (2 points) Explain why your amortized values are correct (i.e. why the bank balance will never be negative). Hint: what is the relationship between the queue and the number of dollars in the bank?

   c. (2 points) Suppose that *instead* of multiDequeue, we have a different operation:

```
public void multiEnqueue(Object item, int i);
```

which enqueues i copies of item to the queue, at a cost of i dollars of actual time. Can we prove that the amortized cost of every queue operation is in $\mathcal{O}(1)$? Why or why not?

**Problem 7.** (7 points) **Index Sort.**

Write a method `indexSort` that uses the following algorithm to sort an array `x` of `Thing` objects. Loop through the `Things` in `x`. For each `Thing`, calculate what index it will be stored at in the output array `y`. Obviously, this index depends on the keys of the other `Things` in the array `x`. Copy the `Thing`'s reference from `x` to `y`.

   Note that `x` may contain duplicate keys, and it is your responsibility to make sure your sort is **stable**.

   Your sort should run in $\mathcal{O}(n^2)$ time, where $n$ is the length of `x`. For full marks, each `Thing`'s reference should be copied **only once ever**. Do **not** change the array `x`.

```
public class Thing {
  int key;                // Use the "key" field as the sort key.
  Object value;

  public static Thing[] indexSort(Thing[] x) {      // Returns sorted copy of x.
    Thing[] y;            // Don't forget to construct a new array.
```

```
    return y;
  }
}
```

Check here if your answer
is continued on the back. |____|