CS 61B Mock Exam
July 30, 2008

Disclaimer: This is mock exam is designed to give you hints for your review. It by no means resembles the actual midterm and you should not use these questions as the only basis in your exam preparation.

You have 45 minutes to complete this mock exam.

CS 61B Mock Exam
July 30, 2008

Problem 1 (Asymptotic Analysis)

Suppose . Assume $f(x)$ and $g(x)$ are positive for all values of x. For each of the following statements, say whether it must be true or must be false or could be either:

$g(x) \in O(f(x))$
Either
$g(x) \in \Omega(f(x))$
True
$g(x) \in \Theta(f(x))$
Either
$f(x) \in \Omega(x \cdot g(x))$
False
$g(x) \in \Omega(x \cdot f(x))$
Either

Problem 2 (Hashing)

Suppose we have defined a hash set for storing elements of class MyClass with b buckets and chains implemented as linked lists. First, n different elements are stored in the table. Now we wish to store one more, called newElem, which is not equal to any of the others. We'd like to do it with as few comparisons (calls to MyClass.equals) between newElem and other elements as possible. Ordinarily we'd ensure this by choosing a very good MyClass.hashCode method. But there's a catch: a dastardly adversary is trying to make the number of comparisons as large as possible.

In terms of n and b, what is the fewest possible number of comparisons it will take to add newElem to the table if:

a) The adversary gets to choose MyClass.hashCode(), but then we get to choose newElem (knowing his choice). What does his MyClass.hashCode look like? You don't need to give any actual code.

n

```
public int hashCode() {
return 0;
}
```

b) The adversary gets to choose newElem, but then we get to choose MyClass.hashCode (knowing his choice). What does our MyClass.hashCode look like?

```
0 comparisons (or 1, if you count the call to equals in hashCode)

public int hashCode() {
if(this.equals(newElem))
return 0;
else
return 1;
}
```

c) We get to choose MyClass.hashCode, and then the adversary gets to choose newElem (knowing our choice).

```
ceiling(n/b)
```

(Optional bonus question—do this part last)

Now suppose we implement the hash table in a different way: rather than storing the chains as linked lists, we store them as binary search trees.

d) What method must we implement in MyClass to ensure this is possible?

```
compareTo
```

Assume through some wizardry that we ensure the search trees are maximally balanced. What are the answers to questions a) through c) above with this new data structure, with comparisons now meaning a call to the method given as the answer to part d)?

a)

```
log(n)
```

b)

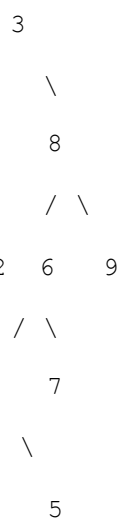
```
0
```

c)

```
log(ceiling(n/b))
```

Problem 3. Binary Tree

1. Draw the binary search tree created by inserting these values in this order: 3 8 6 0 9 2 7 4 1 5



2. Write the inorder traversal of the tree you just drew.

0 1 2 3 4 5 6 7 8 9

3. Given the following class definition:

```
class BinaryTree{
    BinaryTreeNode root;
    int size;

    class BinaryTreeNode{
        int item;
        BinaryTreeNode parent;
        BinaryTreeNode left;
        BinarTreeNode right;
    }
}
```

Write a method `largestKeyNotLargerThan(int k)` that takes an integer key *k* and return the largest key in the binary tree smaller than or equal to k.

```
BinaryTreeNode helper(BinaryTreeNode node, int k) {
// This method returns the element in the subtree rooted at node
// whose item is the largest not larger than k, or null if every
// item in the subtree is greater than k.
if(node == null) return null; // base case
if(node.item == k) return this; // best we could possibly see
if(node.item > k) return helper(node.left, k); // too big; have to
// try smaller values
// If we've gotten this far, we know node.item < k
BinaryTreeNode temp = helper(node.right, k);
if(temp == null) return this; // no bigger options exist than this
return temp;
}

int largetKeyNotLargerThan(int k) {
BinaryTreeNode temp = helper(root, k);
if(temp == null) return ERROR_CODE; // or throw an exception
else return temp.item;
}
```

Problem 4. Sorting

1. If you know the input array to your sorting algorithm is almost sorted, which sorting algorithm would you use? Why?

Insertion sort. On an array with only a constant number of elements out of place, it operates in linear time, whereas all the n log (n) sort algorithms take n log(n) time even when given a sorted list.

2. Picking a good pivot is essential for quicksort to run efficiently; a bad pivot can cause the algorithm to degenerate into worst case $O(n^2)$ running time. Assuming that a particular quicksort implementation always picks the element in the middle of the array, come up with an adversarial input (i.e. worse-case input) that would cause the corresponding quicksort algorithm to run in $O(n^2)$ time.

Assumptions: we are dealing with the linked list version of quicksort, rather than the in-place array version. The "middle" of a list of length 2n is the element at position n; the "middle" of a list of length 2n+1 is element at position n+1.

We design our input so that the pivot is always the minimum element in each sublist. This gives rise to two unbalanced new sublists, one of size 0 and the other of size n-1. The idea is to put the minimum value in the middle of the list, and then to have the next smallest elements expand out from there on either side so that as each minimum is taken out, the next smallest element becomes the new middle. For a list of size 9, the adversarial input looks like this:

8 6 4 2 1 3 5 7 9