Cornell University
Computer Science 211

Second Preliminary Examination 18 April 2006

There are 4 problems on this exam. It is 8 pages long, so make sure you have the whole exam. You will have $1\frac{1}{2}$ hours in which to work on the problems. You will likely find some problems easier than others; read all problems before beginning to work, and use your time wisely. The prelim is worth 100 points total. The point breakdown for the parts of each problem is printed with the problem. Some of the problems have several parts, so make sure you do all of them!

This is an open-book examination; you may use the textbooks, copies of the course notes, or your own notes. Please keep your materials to yourself. If you bring loose-leaf notes, they should be stapled securely together. You may not write on your notes or books during the exam.

Do all written work on the exam itself. If you are running low on space, write on the back of the exam sheets and be sure to write (OVER) on the front side. It is to your advantage to show your work—we will award partial credit for incorrect solutions that are headed in the right direction. If you feel rushed, try to write a brief statement that captures key ideas relevant to the solution of the problem.

If you finish in the last ten minutes of the exam, please remain in your seat until the end of the exam as a courtesy to your fellow students. Also, remember to turn off all cell phones and pagers that may interrupt the exam.

Problem	Points	Score
1	24	
2	25	
3	26	
4	25	
Total	100	

Name and NetID _____

1. True/false [24 pts] (parts a–h)

   Each question is worth 3 points.

a. ____ The worst-case running time of quicksort is worse than that of mergesort.

b. ____ The job of a specification is to explain how code is implemented.

c. ____ The lower bound for general sorting algorithms (algorithms that work on any type) is $\Omega(n \log n)$

d. ____ Binary search on an array takes $\Omega(n \log n)$ time.

e. ____ Open addressing works relatively well with a high load factor.

f. ____ Insertion sort is faster than quicksort on a sorted array.

g. ____ Stacks are FIFO.

h. ____ Set abstractions can be implemented simply using map abstractions.

2. Sorting [25 pts] (parts a–c)

(a) [10 pts] Suppose we use mergesort to sort an array containing the following sequence of elements: 1, 5, 6, 3, 2, 4, 9, 0. For this example input, illustrate how merge sort works by drawing the array states that result after each merge.

(b) [7 pts] Suppose we use quicksort but as a pivot we use the mean (average) of the elements. For simplicity, let us assume that the mean of the elements in the array (or any subarray) always partitions the elements into two equal-sized sets. Explain briefly why quicksort will still take $O(n \lg n)$ time in that case.

(c) [8 pts] Explain briefly why the worst-case time of quicksort with a mean pivot is still $O(n^2)$. (Hint: consider an array containing elements $i!$).

3. Hash tables [26 pts] (parts a–c)

(a) [10 pts] Suppose we have a hash table with an table size of 10, into which we are inserting the integers from 1 to 10. Our hash function $h(x)$ is the square of the element, modulo the table size (i.e., the remainder when the square is divided by the table size. For example, $h(9) = 1$.) Draw the hash table that results, assuming it uses chaining.

(b) [6 pts] When might this hash function $h(x) = x^2$ mod m be a poor choice, assuming m is the length of the hash table? Discuss briefly with an example.

(c) [10 pts] You are given an array of numbers of length n and a sum s. We can create an $O(n)$ algorithm that determines whether two numbers in the array add to the value s. For example, for the array 1, 3, 2, 5 and the sum $s = 8$, the result is `true`, but for $s = 2$ and $s = 10$, it would be `false`. You should clearly describe an algorithm that accomplishes this in $O(n)$ time, and justify why it is $O(n)$. You may write code or pseudocode, but this is not required. Solutions that are not $O(n)$ will be accepted, with some penalty.

4. Implementing a collection [25 pts] (parts a–d)

Consider the following implementation of a set abstraction as a circularly linked list. This implementation uses an extra list node object to represent the end of the list, rather than `null`. This is known as a **sentinel** object; this sentinel also serves as the header object for the whole list, avoiding having a separate class. The `elem` field in the sentinel is unused.

```
class List<T> implements Collection<T> {
  private T elem;
  private List<T> next;

  /** Create an empty list. */
  public List() {
     next = this;
  }
  public boolean contains(T x) {
    List<T> curr = next;
    while (curr != this) {
      if (curr.elem == x) return true;
      curr = curr.next;
    }
    return false;
  }
  public boolean add(T x) {
    if (contains(x)) return false;
    List<T> nw = new List<T>();
    nw.elem = x;
    nw.next = next;
    next = nw;
    return true;
  }
  public boolean remove(T x) {
    List<T> curr = this;
    while (curr.next != this) {
      if (curr.next.elem == x) {
        curr.next = curr.next.next;
        return true;
      }
      curr = curr.next;
    }
    return false;
  }

  public Iterator<T> iterator() {
    return new ListIterator();
  }
  private class ListIterator implements Iterator<T> {
     List<T> curr = next;
     boolean hasNext() { ...  }
     T next() { ...  }
  }
}
```

(a) [4 pts] Draw what the data structure created by `List<Integer>()` looks like.

(b) [4 pts] Now draw what it looks like after adding the element 4.

(c) [4 pts] Identify an error common to the implementations of both the `contains` and `remove` methods and explain how to fix each of them by changing one line of code in each.

(d) [13 pts] Implement the iterator class `ListIterator` with code in which the two . . .'s are filled in. All iterator operations should be $O(1)$. (Hint: the containing `List` object can be accessed in `ListIterator` with the expression `List.this`.) If you can't figure out how to do it by just filling in the . . .'s, you can implement it some other way with only a 3-point deduction (if it works).

Bonus problem

Consider the following implementation of a set. We use a pair of arrays in which the longer array is kept sorted and the shorter one is not. The length of the shorter array is the square root of the length of the longer array, although it may have some empty entries if it is not full yet. To add an element, we insert it in order in the short array, moving other elements as necessary. If the short array fills up, we allocate larger long and short arrays, and merge the two old arrays into the new long array. The new short array is empty. To find an element, we use binary search on both the long and short arrays.

Show that this data structure gives $O(\lg n)$ lookup time, but the amortized time to add an element is $O(\sqrt{n})$.