Cornell University
Computer Science 211

Final Examination 19 May 2006

---

There are 6 problems on this exam. It is 10 pages long, so make sure you have the whole exam. You will have $2\frac{1}{2}$ 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.

A few parts are marked with a (*). These are trickier problems that you may want to save for later if you are feeling pressed for time, because they are harder than their point value indicates.

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 | 20 | |
| 2 | 16 | |
| 3 | 18 | |
| 4 | 14 | |
| 5 | 20 | |
| 6 | 12 | |
| Total | 100 | |

Name and NetID _____

1. True/false [20 pts] (parts a–j)

   Each question is worth 2 points.

a. ____ The essential property of a good user interface is that it is easy for programmers to implement.

b. ____ DFS takes $O(|E| + |V|)$ time if the graph is represented as an adjacency matrix. $|E|$ is the number of edges and $|V|$ is the number of vertices(nodes).

c. ____ The worst-case performance of looking up an element in a hash table is $O(n)$, where $n$ is the number of elements.

d. ____ Binary search trees are faster than hash tables in practice.

e. ____ Trees and singly-linked lists are both DAGs.

f. ____ Binary search requires linear time in the worst case.

g. ____ Insertion sort has worst-case running time that is $O(n \lg n)$.

h. ____ Swing Listeners are an example of the Observer pattern.

i. ____ In the worst case, both breadth-first and depth-first search can require state that is $O(|V|)$ in size.

j. ____ Breadth-first search uses a stack.

2. Choosing data structures [16 pts] (parts a–d)

Check the best data structure for solving each of the following problems.

(a) [4 pts] You have a large collection of objects representing vehicles with New York State license plates. You need to be able to look up a vehicle given its license plate.

_____ Hash table
_____ Linked list
_____ Tree
_____ Binary search tree
_____ Array
_____ Resizable array

(b) [4 pts] You want to keep track of appointments and other events organized by their time and date. It is important to be able to efficiently find all the events between a starting time/day and an ending time/day.

_____ Hash table
_____ Linked list
_____ Doubly-linked list
_____ Binary heap
_____ Binary search tree
_____ Array

(c) [4 pts] You are writing a program that watches packets going by on the network and records the IP address of the host machine sending each packet, for later processing. It's important that your program not pause for any significant amount of time because it might miss some packets.

_____ Hash table
_____ Linked list
_____ Graph
_____ Binary search tree
_____ Array
_____ Resizable array

(d) [4 pts] You are implementing an interactive program development environment (like Eclipse) for the Java language. You need a data structure that keeps track of all the classes and packages and lets you find, for example, the set of classes in a given package or its subpackages, or classes nested inside a given class.

_____ Linked list
_____ Tree
_____ Binary search tree
_____ Binary heap
_____ Array
_____ Resizable array

3. Asymptotic complexity and binary search trees [18 pts] (parts a–c)

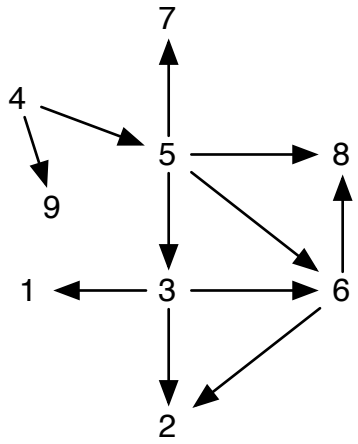
(a) [4 pts] Consider a perfectly balanced binary tree of height $h$. What is the largest possible number of nodes it can contain? Be precise.

(b) [8 pts] Consider the function $f(n) = a \lg(n+b)$, where $a$ and $b$ are positive constants. Show that this function is $O(\lg n)$. (**Hint:** consider $n \geq \max(b, 2)$)

(c) [6 pts] (*) Let us say that a binary search tree is *roughly balanced* if the depth of the deepest node is no more than twice the depth of the least deep node that is missing one or both of its children (i.e., it has null instead of a child). Show that in such a tree (a *red-black tree* is one example), binary search takes time $O(\lg n)$, where $n$ is the number of nodes in the tree. You may take the result of part (b) as given. (**Hint:** how many nodes have a depth less or equal to $h/2$?)

4. Graphs [14 pts] (parts a–c)

Consider the following graph:




(a) [5 pts] Suppose we perform a breadth-first search of this graph starting at node 4. Which nodes could be the last node visited by the search? Explain briefly.

(b) [5 pts] Suppose we perform a recursive depth-first search of this graph starting from 4. From any given node, we visit its successors in ascending order of their value. In what order are the nodes visited (marked gray)?

(c) [4 pts] Now, give the topological sort of the nodes corresponding to the DFS of part 4(b).

5. Specification [20 pts] (parts a–e)

Suppose we are given a data abstraction representing a day of the year, with the following interface:

```
/* A Date is a day of a unspecified year. For example, May 17 or
 * February 29.
 */
class Date {
  /** Make a new Date object with the specified month and day. */
      Requires: "month" must be between 1 and 12.
                "day" must be between 1 and the number of
                days in that month (29 in case of February).
  public Date(int month, int day) { ... }
  /** The month (1-12). */
  public int month() { ... }
  /** The day (1-31). */
  public int day() { ... }
  /** A human readable representation, e.g., "May 17" */
  public String toString() { ... }
  /** The next day of the year. */
  public Date tomorrow() { ... }
  /** The previous day of the year. */
  public Date yesterday() { ... }
}
```

(a) [3 pts] Which methods are creators? Which are observers? Which are mutators? Is this a mutable or an immutable data abstraction?

(b) [4 pts] The methods `tomorrow` and `yesterday` fail to specify some behavior, and furthermore, they don't consider the possibility of leap years. Change the signature and specification of the `tomorrow` method to correct these problems. (You have some flexibility here; all reasonable solutions will be accepted).

(c) [2 pts] Now, suppose we want to implement this class with just a single integer field representing the number of days since January 1, counted according to a leap year.

```
// The number of days elapsed since the beginning of a leap year,
// inclusive. Thus, January 1 is represented by 1 and December 31, by 366.
   private int day_count;
```

What, if anything, is the representation (data structure) invariant for this class?

(d) [5 pts] Implement the method `tomorrow` using the representation of part 5(c) and your specification. You may use existing methods and you may define additional private methods and private constructors.

(e) [6 pts] (*) Suppose we wanted to define a subclass of `Date` that also kept track of the year, called `YearDate`. We could add a field for that:

```
class YearDate extends Date {
    private int year;
    ...
}
```

Show how to implement the following without changing the superclass:

- A constructor `YearDate(int month, int day, int year)`
- The method `String toString()`.
- A method `YearDate tomorrow()` that returns the next day, increasing the year if necessary. You may assume a function `is_leap_year(int year)` is already implemented for you:

```
/** Is this a leap year in the Gregorian calendar. */
boolean is_leap_year(int year) {
  return (year%4 == 0) && (year%100 != 0) || (year%400 == 0);
}
```

6. Recursion and Induction [12 pts] (parts a–b)

We are using this list data structure:

```
class ListNode<T> {
  T element;
  ListNode<T> next;
}
```

Given a list of integers, we will say that an integer is "late" if its value is less than its index in the list, considering the first list element to have index zero. For example, in a list containing the elements 1, 3, 7, 2, 10, 5, 4, the late elements would be 2 and 4. The element 5 is not late, because it is at index 5.

(a) [6 pts] Use recursion to write a static method `lateElements` that operates on a `ListNode<Integer>` and returns a newly created list containing all its late elements. You may define helper methods.

(b) [6 pts] (*) Now write an proof that your implementation works, using induction. Be sure to clearly state both what you are proving and your induction hypothesis. (**Hint:** Your induction hypothesis will likely be more general than the statement that you are proving.)