Cornell University
Computer Science 211

Final Examination 19 May 2006

Solutions

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.

   False

   (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).

   False

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

   True

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

   False

   (e) Trees and singly-linked lists are both DAGs.

   True

   (f) Binary search requires linear time in the worst case.

   False

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

   False

   (h) Swing Listeners are an example of the Observer pattern.

   True

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

   True

   (j) Breadth-first search uses a stack.

   False

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.

   __X__ 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
__X__ 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
__X__ 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
__X__ 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.

Answer:
$2^{h+1}-1$. *Recall that the height of a tree is the length of the longest path from the root to any leaf.*

(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 $n \geq \max(b,2)$)

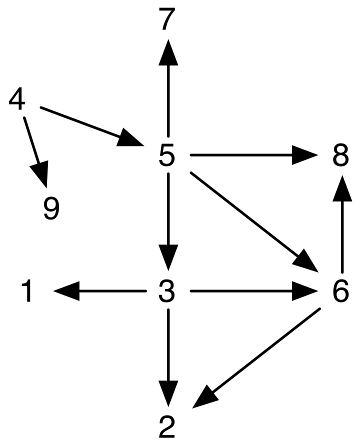
Answer:
We need to show that there exists some k and n_0 such that $a\lg(n+b) \leq k\lg n$ for all $n \geq n_0$. Consider $n_0 = \max(b,2)$ and $k = 2a$. Then $\lg(n+b) \leq \lg 2n = 1+\lg n$. So $a\lg(n+b) \leq a\lg n + a$, and since $1 \leq \lg n$, this is bounded by $2a\lg n = k\lg n$, as desired.

(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 $h/2$?)

Answer:
Binary search takes time that is $O(h)$. So we need to show that h is $O(\lg n)$. Consider the set of nodes that are at a depth of $h/2$ or less. These nodes form a perfectly balanced tree, and therefore there must be $2^{h/2+1}-1$ of them. Therefore, we know that $2^{h/2+1}-1 \leq n$, so $h \leq 2\lg(n+1)-2$. But by part (b), $2\lg(n+1)-2$ is $O(\lg n)$, so h is too.

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.

Answer:
The nodes 1 and 2 are the farthest, at distance 3. So one of these two must be visited last regardless of how the BFS is done.

(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)?

Answer:

4, 5, 3, 1, 2, 6, 8, 7, 9

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

Answer:
If we use the depth-first search just given to do the topological sort, we end up with the following:
4, 9, 5, 7, 3, 6, 8, 2, 1
Many people gave a topological ordering that didn't correspond to the DFS of part (b). Partial credit was assigned in these cases.

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?

Answer:
Creators: `Date`, `tomorrow`, `yesterday`.
Observers: `month`, `day`, `tomorrow`, `yesterday`, `toString`
Mutators: none
This is 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).

Answer:

```
/** Creates a new Date representing the next day
    of the year. Requires that the current day is
    not December 31. For February 28, the next day
    is February 29 if the year is a leap year,
    March 1 otherwise.
    @param leap_year whether the year is a leap year.
 */
public Date tomorrow(boolean leap_year) { ... }
```

Some people simply assumed that it was a leap year. But this violated the original intent of the data abstraction, so it wasn't a good solution.

(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?

Answer:
The invariant is that `day_count` *must be between 1 and 366.*

(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.

Answer:

```
private Date(int c) { day_count = c; }
public Date tomorrow(boolean leap_year) {
  Date ret = new Date(day_count + 1);
  if (!leap_year && day_count == 59)
     ret.day_count = 61;
  return ret;
}
```

If you made December 31 wrap around in `tomorrow()`*, you needed to check for that too.*

(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);
}
```

Answer:

```
YearDate(int month, int day, int year) {
  super(month, day);
  this.year = year;
}
String toString() {
  return super.toString() + ", " + year;
}
YearDate tomorrow() {
   Date d = tomorrow(is_leap_year(year));
   if (d.month() == 1 && d.day() == 1) {
     return new YearDate(d.month(), d.day(), year+1);
   } else {
     return new YearDate(d.month(), d.day(), year);
   }
}
```

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.

Answer:

```
ListNode<Integer> lateElements(ListNode<Integer> elems) {
  return lateN(0, elems);
}
/** Return a list of the elements whose value is less than
 *  their index plus n */
ListNode<Integer> lateN(int n, ListNode<Integer> elems) {
  ListNode<Integer> rest;
  if (next == null) rest = null;
  else rest = lateN(n+1, elems.next);
  if (elems.element < n) {
    ListNode<Integer> ret = new ListNode<Integer>();
    ret.element = element;
    ret.next = rest;
    return ret;
  }
  return rest;
}
```

(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.)

Answer:

We want to prove that `lateElements` *works. Assuming that* `lateN` *works as described by its specification,* `lateElements` *must work because zero plus the index is equal to the index. but we need to show that* `lateN` *works as advertised.*

The statement to prove is that `lateN(m,elems)` *returns a newly created list containing all the elements whose value is less than* `index+m`*. We will prove this by induction on the length of the list, which we will call n. The induction hypothesis is that on a list of length n,* `lateN(m,elems)` *returns a newly created list containing all the elements whose value is less than* `index+m`*.*

The base case for the induction is $n = 0$, an empty (`null`*) list. In this case it correctly returns an empty list.*

In the inductive step, we assume that `lateN` *works correctly for a list of length n, where n is some integer greater than or equal to 0. Our goal is to prove that it works for a list of length $n + 1$. Since length of the incoming list is $n + 1 \geq 1$, we know we don't return* `null` *immediately. The call to* `lateN` *is on* `elems.next`*, which is a list of length n. So by the induction hypothesis, we know that* `rest` *contains the elements in the list that starts at* `elems.next` *and whose value is less than the index in that list (call it i), plus $\mathtt{m} + 1$. But the index in that list is one less than the index according to the current list (call it j). Therefore, if the value of an element is less than $i + \mathtt{m} + 1$, it is less than $j + \mathtt{m}$. So the recursive call returns a newly created list containing all the late elements in the rest of the list.*

In the remainder of the method, if the current element is less than n, its value is less than its index (0) plus n, so it is prepended to the returned list. Otherwise, the current element should not be included in the result list, and the list `rest` *can be returned directly.*