Cornell University
Computer Science 211
First Preliminary Examination 9 March 2006

Solutions

1. Short answers [24 pts] (parts a–f)

   Answer true (T) or false (F). Four points off for each missing or wrong answer. You can justify your answer but you do not need to.

   (a) A `String` is a mutable value in Java.

   false: you can't change a `String` *value*

   (b) In a doubly linked list, a node has references to the previous and next nodes in the list.

   true

   (c) A tail-recursive method can call itself twice and add the results together.

   false: then something is done after the last recursive call: adding their results together.

   (d) A recursive method is always better than an iterative one.

   false

   (e) In Java, a class can implement multiple interfaces.

   true

   (f) In a context-free grammar (the kind of grammar we've seen in class), the left-hand-side of a production is always a single nonterminal symbol.

   true: see the notes for the Feb. 2 lecture.

2. Implementation and inheritance [25 pts] (parts a–c)

   Suppose we have an interface `Shape` that is part of a geometry package. It has the following operations:

```
/** A two-dimensional shape on the plane, with Cartesian coordinates. */
interface Shape {
  // These four methods give coordinates that bound the shape in a
  // rectangle.
  float leftX();
  float rightX();
  float bottomY();
  float topY();
  /** The area of the shape. */
  float area();
}
```

   Now, we want to implement some different shapes as part of the package. Here is how we might implement rectangles and circles: e.g.:

```
class Rectangle implements Shape {
  private float x0, x1, y0, y1;
  public Rectangle(float x0, float x1, float y0, float y1) {
    this.x0 = x0; this.x1 = x1; this.y0 = y0; this.y1 = y1;
  }
  public float leftX() { return x0; }
  public float rightX() { return x1; }
  public float bottomY() { return y0; }
  public float topY() { return y1; }
  public float area() { return (x1 - x0) * (y1 - y0); }
}
class Circle implements Shape {
  private float center_x, center_y, radius;
  public Circle(float cx, float cy, float r) {
    center_x = cx; center_y = cy; r = radius;
  }
  public float leftX() { return center_x - radius; }
  public float rightX() { return center_x + radius; }
  public float bottomY() { return center_y - radius; }
  public float topY() { ... }
  public float area() { return Math.PI*radius*radius; }
}
```

(a) [3 pts] Implement `Circle.topY`. Is it an creator, observer, or mutator?

Answer:

```
public float topY() { return center_y + radius; }
```

Observer.

(b) [10 pts] Suppose we want to implement a class `Square` that inherits from `Rectangle`, but has a constructor `Square(float center_x, float center_y, float size)`. Complete the ...'s in the following implementation:

Answer:

```
class Square extends Rectangle {
    Square(float cx, float cy, float size) {
       super(cx - size/2, cx + size/2, cy - size/2, cy + size/2);
    }
}
```

(c) [12 pts] Now, suppose we also want to implement an annulus, the donut-shaped region lying between two concentric circles. Complete the implementation below by filling in the "...". What methods, if any, of `Circle` need to be overridden?

Answer:
We need to override `area`.

```
class Annulus extends Circle {
    float inner_radius;
    Annulus(float cx, float cy, float outer_radius, float inner_radius)
    {
        super(cx, cy, outer_radius);
        this.inner_radius = inner_radius;
    }
    float area() {
        return super.area() - PI*inner_radius*inner_radius;
    }
}
```

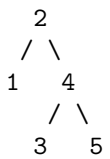
3. Induction [26 pts] (parts a–c)

One way to implement a binary search tree containing integers in Java is as follows:

```
class TreeNode {
  private TreeNode left, right;
  private int element;
  ...
}
```

(a) [5 pts] Draw a picture of a binary search tree of height 2 containing the elements 1, 2, 3, 4, and 5, with the number 2 at the root.

Answer:



(b) [5 pts]

In a tree that satisfies the binary search tree invariant, the following conditions hold:

i. Every element in the `left` subtree is less than the root element.
ii. Every element in the `right` subtree is greater than the root element.
iii. The left and right subtrees also satisfy the BST invariant themselves.

We can also define a "reversed BST invariant":

i. Every element in the `left` subtree (if any) is *greater* than the root element.
ii. Every element in the `right` subtree (if any) is *less* than the root element.
iii. The left and right subtrees (if any) also satisfy the reversed BST invariant themselves.

If a tree satisfies the reversed BST invariant, what can we say about the order in which elements are visited in an in-order traversal of the tree?

Answer:

They will be visited in reversed (descending) order.

(c) [16 pts] Consider the following recursive method operating on a `TreeNode`:

```
static void swap(TreeNode t) {
    if (t.left != null) swap(t.left);
    if (t.right != null) swap(t.right);
    TreeNode tmp = t.left; t.left = t.right; t.right = tmp;
}
```

Use induction on the height of the binary search tree `t` to prove that after a call to the method on `t`, the tree will satisfy the "reversed BST invariant" and will contain all the same elements as before the call. Be sure to clearly state the property you are proving, what kind of induction you are using, and your induction hypothesis.

Answer:

Proof: *by strong induction on the height h of the tree* `t`*. The property $P(h)$ to be proved is that* `swap` *converts a BST of height h into one that satisfies the reversed BST invariant, and that it contains all the same elements it did originally. We will use strong induction, so the induction hypothesis is that $P(h')$ holds for all $0 \leq h' \leq h$.*

Base case: *$h = 0$. In this case,* `t.left` *and* `t.right` *are null, and the whole method has no effect. The resulting tree clearly satisfies the reversed BST and contains the same elements.*

Inductive case: *Assume that $P(h')$ holds for all $0 \leq h' \leq h$. We need to show that $P(h+1)$ holds. Therefore, we assume that we have a BST of height $h+1$. Its left and right subtrees are both BSTs of height less than or equal to h. Therefore, by the induction hypothesis, we know that the recursive calls to* `swap` *will make both of these trees satisfy the reversed invariant, while keeping all the same elements. Now, every element in the (old) left subtree is still less than the current element, because they're all the same elements as before, and similarly with the right subtree. Therefore, when left and right are swapped, the right subtree will now contain only* smaller *elements and the left subtree only* greater *ones, and the reversed BST invariant clearly holds.*

4. ADTs [25 pts] (parts a–d)

The standard Java interface `SortedSet` describes a set whose elements have an ordering. Abstractly, the set keeps its elements in sorted order. Here is a similar interface that happens to describe only sets of integers:

```
/** A set of unique elements kept in ascending sorted order. */
interface IntSortedSet {
   /** Add x to the set. */
   boolean add(int x); % XXX better to be void so it's clearly a mutator.
   /** Tests whether x is in the set.
    *  @return whether x is in the set. */
   boolean contains(int x);
   /** Remove x. */
   void remove(Object x);
   /** Return the first element in the set.
    *  @return the first (lowest) element. */
   int first();
}
```

(a) [6 pts] The specification of `remove` has at least one serious problem. Clearly identify a problem and write a better specification. You may change the signature if you justify it. (**Note:** We don't consider a failure to produce nice javadoc a "serious" problem.)

Answer:

The problem with the spec is that it doesn't say what happens when the element is not in the set.

```
/** Remove an element. If o is an element of the set,
 * removes it. Otherwise, the set is unchanged.
 */
void remove(Object x);
```

(b) [6 pts] The specification of one of the observers has at least one serious problem. Write a better specification. You may change the signature if you justify it.

Answer:
The problem is what the method does with an empty set. One possibility is to add a "Checks" clause:

```
/** Return the first element in the set. Checks: the set is nonempty.
 *  @return the first (lowest) element. */
int first();
```

You could also add a "Requires" clause instead, or change the signature to throw an exception in the case where the set is empty.

Now, suppose that we want to implement this interface with a linked list kept in sorted order:

```
class SortedList implements IntSortedSet {
    // Rep Invariant: The list is kept in ascending sorted order
    // and contains no duplicate elements.
    int element;
    SortedList next;
    ...
}
```

(c) [6 pts] Implement the `first` method of `SortedList`.

Answer:

```
int first() {
   return element;
}
```

(d) [7 pts] Now, suppose we wanted to write a class `UnsortedList`, just like `SortedList` except that it has no representation invariant. But it should still implement the `IntSortedSet` interface. Could we implement the `first` method for this class? Write code for the `first` method or explain why it can't be done.

Answer:
It can be done. We just can't rely on the first element being at the beginning of the list:

```
int first() {
    int min = element;
    IntUnsortedList curr = this;
    while (curr != null) {
       if (curr.element < min) min = curr.element;
       curr = curr.next;
    }
    return min;
}
```

A lot of people got confused because they thought that they were supposed to return both the first element of the list, and the lowest element. The thing to remember is that we have to look at things from the standpoint of the client of the abstraction. The client shouldn't have to know or care that the set happens to be implemented as a list; the client is supposed to be able to think about this object as a sorted set. Therefore, the "first" element is the lowest. Whether the list is sorted or unsorted is a detail of the implementation that may affect efficiency but does not affect the answer that is supposed to be returned.