CS 61B (Clancy, Yelick) Solutions and grading standards for exam 2 Spring 2001

Exam information

345 students took the exam. Scores ranged from 3 to 25, with a median of 19 and an average of 18.1. There were 176 scores between 19 and 25, 125 between 12.5 and 18.5, 42 between 6 and 12, and 2 less than 6. (Were you to receive 75% of the points on all your exams, plus good grades on homework and lab, you would receive an A–; similarly, a test grade of 50% may be projected to a B–.)

There were two versions of the exam, A, and B. (The version indicator appears at the bottom of the first page.) Versions were essentially identical except for small changes in some of the problems.

If you think we made a mistake in grading your exam, describe the mistake in writing and hand the description with the exam to your lab t.a. or to Mike Clancy. We will regrade the entire exam.

Solutions and grading standards for versions A and B

Problem 0 (1 point)

You lost 1 point on this problem if you did any of the following:

- you earned some credit on a problem and did not put your name on the page,
- you did not indicate your lab section or t.a. (or gave conflicting information), or
- you failed to put the names of your neighbors on the exam.

The reason for this apparent harshness is that exams can get misplaced or come unstapled, and we would like to make sure that every page is identifiable. We also need to know where you will expect to get your exam returned. Finally, we occasionally need to know where students were sitting in the class room while the exam was being administered.

Problem 1 (4 points)

This problem, based on lab assignments 5 and 6 and project 1, was identical on the two versions. It asked you to fill in the code for an enumeration of integers in an Interval object. Enumerations you saw in lab had the following components:

- an instance variable that refers to the next item to return;
- a constructor that initializes the instance variable;
- a hasMoreElements method that checks the value of the instance variable to make sure it refers to a legal element of the collection being enumerated;
- a nextElement method that saves the current item, updates the instance variable, and then returns the item.

A corresponding instance variable in this problem would be something that indicates the next value to return from the Interval object. You don't need a reference, however; you can store the int value from the interval directly. Here's the code:

```
public IntervalEnumeration ( ) {
   nextInt = myLow;
}

public boolean hasMoreElements ( ) {
   return nextInt <= myHigh;
}

public Object nextElement ( ) {
   if (!hasMoreElements ( )) {
      throw new NoSuchElementException ("interval ran out");
   }
   nextInt++;
   return new Integer (nextInt - 1);
}

private int nextInt;
```

Note, incidentally, that IntervalEnumeration is declared inside Interval, so its methods are allowed to access myLow and myHigh.

Another approach was to use a Vector or LinkedList object to keep track of the state of the enumeration. Initially, all the integers in the interval are stored in the vector. If the vector is nonempty, its first element is the next item to return; this item is then removed in the nextElement method. Here is code:

```
public IntervalEnumeration ( ) {
   v = new Vector ( );
   for (int k=myLow; k<=myHigh; k++) {
      v.addElement (new Integer (k));
   }
}

public boolean hasMoreElements ( ) {
   return v.size ( ) > 0;
}

public Object nextElement ( ) {
   if (!hasMoreElements ( )) {
      throw new NoSuchElementException ("interval ran out");
   }
   Object x = v.elementAt (0);
   v.removeElementAt (0);
   return x;
}

Vector v;
```

This approach can be simplified by relying on Vector's own enumeration method:

```
public IntervalEnumeration ( ) {
   v = new Vector ( );
   for (int k=myLow; k<=myHigh; k++) {
      v.addElement (new Integer (k));
   }
   Enumeration enum = v.elements ( );
}
```

```
public boolean hasMoreElements ( ) {
   return enum.hasMoreElements ( );
}
public Object nextElement ( ) {
   return enum.nextElement ( );
}
Vector v;
Enumeration enum;
```

The Vector enumeration will take care of throwing NoSuchElementException.

This problem was worth 4 points, and was graded as follows. (Some graders marked abbreviating codes E1, …, E11 to indicate errors; the codes are listed below with the errors they abbreviate.)

- 1 point was deducted for a minor logic error. Such errors included off-by-one processing (E1), failing to throw an exception where necessary (E2), returning an int rather than an Integer object (E4), returning prematurely (E5), use of 0 for a sentinel value (E6), assigning an int to an Integer (E7), casting an int to an Integer (E8), and failing to initialize an array or vector (E9).
- 2 points were deducted for changing the state of the Interval object during the enumeration (E3).
- Solutions with more serious errors received 0, ½, or 1 point depending on how much understanding they displayed of the various components of the enumeration. (Some of these were labeled E11.) A prominent example, worth 1 out of 4, was the apparent confusion of implements with extends; some students seemed to assume that they could just use the methods of the Enumeration "superclass".

Problem 2 (5 points)

This problem, based on lab assignment 6, was also identical on the two versions. It asked for code to exchange the first two elements of a linked list. Here is a solution:

```
private void exchangeFirstTwoNodes ( ) {
   if (myFirst.myNext == null) {
      throw new NoSuchElementException ("list not long enough");
   }
   ListNode second = myFirst.myNext;
   myFirst.next = second.myNext;
   second.myNext = myFirst;
   myFirst = second;
}
```

Some students created new ListNodes to do the exchange, for example as follows:

```
ListNode new2 = new ListNode (myFirst.myItem, myFirst.myNext.myNext);
ListNode new1 = new ListNode (myFirst.myNext.myItem, new2);
myFirst = new1;
```

You were warned not to assign into a myItem variable, and technically the ListNode constructor does such an assignment. However, we decided to allow this solution because it didn't involve a *direct* assignment to a myItem variable.

Grading (out of 5 points) was as follows. You lost all 5 points for any assignment to a myItem variable, as promised in the problem statement. In other solutions, 2 points were awarded for throwing an exception correctly for a 1-element list, and 3 points were earned for correctly modifying the three reference variables (myFirst, myFirst.myNext, and myFirst.myNext.myNext). Returning rather than throwing an exception lost you both exception points; you lost 1 point for a minor error in throwing the exception, for example, missing new, wrong condition, or missing parentheses in the call to the exception constructor. You lost 1 point for each missing reference update, 1 point for each minor error (e.g. wrong instance variable names, private local variables, etc.

Errors in linking were common: losing nodes, creating cycles, etc.. You needed to use at least one temporary variable in a correct solution.

A common error was to assume that the list had a size variable or method; this lost 1 point. Other students assumed that the list contained a sentinel node; nothing in the problem specifies this, however. (This error may have resulted in a solution that was substantially different from what we expected, and thus was misevaluated. If the assumption of a sentinel was the *only* error you made on this problem, you should receive 4 out of 5 points.) Another common error was to omit necessary references to myFirst, for example by using myNext consistently to mean myFirst.myNext. Many students also seemed not to realize that List and ListNode are two separate classes.

Problem 3 (4 points)

This problem involved analyzing code similar to that in lab assignment 7. Note that BetterInterval.equals does not override Object.equals because of the different parameter type, so BetterInterval.equals will only be called when the compiler can determine that two BetterInterval objects are involved in the comparison. Version A's answers were the following:

expression	*result + explanation*
`b = intvl1.equals (intvl2);`	true; since both intvl1 and intvl2 are declared as BetterIntervals, BetterInterval.equals is used.
`b = v1.equals (v2);`	false; Vector comparisons use Object.equals to compare vector elements unless Object.equals is overridden.
`b = intvl3.equals (intvl4);`	false; the apparent type of intvl4 is Object, so the only match is Object.equals.
`b = ((BetterInterval) intvl3).equals ((BetterInterval) intvl4);`	true; the apparent type of the recast intvl3 and intvl4 is BetterInterval, so BetterInterval.equals is used.

In version B, the first two sets of statements were switched and the last two sets of statements were switched from corresponding code in version A. Thus the version B answers are false, true, true, false.

1 point was awarded per correct answer. No partial credit was awarded.

Problem 4 (7 points)

Version A was identical to version B except that the order of the columns were reversed for all the choices. Here are solutions. Recall that Θ(1) means constant time.

Sorted doubly linked list implementation:

operation	*running time + explanation*
finding k or k's proper position in the list (for insertion)	Θ(n); linear search is necessary
then inserting k into the list if it's not already there	Θ(1); doubly linked lists allow in-place insertion or deletion
then updating the median if necessary	Θ(1); there are four cases, depending on whether there are an odd or even number of values in the list and on whether k was inserted before or after the median
locating the median (for deletion)	Θ(1); one of the instance variables is keeping track of the median
then removing it from the linked list,	Θ(1); doubly linked lists allow in-place insertion or deletion
then updating the median if necessary	Θ(1); the median moves to what was either its left or right neighbor depending on the number of values in the list

Vector implementation:

operation	*running time + explanation*
finding k or k's proper position in the vector (for insertion)	Θ(log n); binary search is used
then inserting k into the vector if it's not already there	Θ(n); insertion at the start of the vector requires shifting all the subsequent elements one position
then updating the median if necessary	Θ(1); the median is always the element at position (myElements.size( )–1) / 2, and Vector.insertElementAt has already incremented the size in the preceding step
locating the median (for deletion)	Θ(1); the median is at position (myElements.size( )–1) / 2
then removing it from the vector,	Θ(n); subsequent elements must be shifted back
then updating the median if necessary	Θ(1); Vector.removeElementAt decrements the size, from which the position of the median may immediately be computed

The answer to part c, identifying the total running time of the add method in the Vector implementation, is Θ(n). This is Θ(log n) for finding k's position + Θ(n) for inserting k. Θ(log n) is dominated by Θ(n), and thus disappears from the Θ estimate.

In parts a and b, ½ point was awarded per correct answer; part c was worth 1 point. There was no partial credit. You were allowed in the Vector implementation to say "none" or "not necessary" instead of circling Θ(1).

Some other notes: The three steps of the each high-level operation are not completely independent. (That was the purpose of our saying "then removing ...", "then inserting ...", and so on.) It is likely, in fact, that the code for these steps would all be in a single method and could thus share local variables. Also, though you were not allowed to assume any extra *instance* variables, nothing ruled out the use of local temporary variables (in order to implement the operation "as fast as possible" as directed in the problem statement). An example would be in deleting and updating the median:

```
// after checking that median.myNext and median.myPrev were nonnull
ListNode prev = median.myPrev;
ListNode next = median.myNext;
prev.myNext = next;
next.myPrev = prev;
mySize--;
myMedian = mySize%2? prev: next;
```

Problem 5 (4 points)

This problem was identical on both versions. A repOk method would need to check several properties beyond those given:

- mySize is the number of nodes in the list;
- the myItem variable in each node contains a nonnull object;
- the myItem variable in each node contains an Integer object;
- the nodes are arranged in order of their myItem values;
- myMedian contains a reference to the middle node in the list.

You received 1 point for each nontrivial property that couldn't be inferred from others you gave or from those we gave you. Almost all 4-point answers listed some property about mySize, something about the contents of myItem, something relating adjacent nodes, and something about myMedian. We evaluated breadth more than depth; thus you received a point for saying that myMedian referred to some node in the list, or for saying that mySize > 0 if you did not also say that mySize is the number of nodes in the list. Two closely related properties—for example, (a) each myItem value is less than or equal to that of its successor node, and (b) there are no duplicate values in the list—received only 1 point. (However, you got credit both for saying that all myItem values were nonnull and that they contained Integer objects.)

We think that almost any property involving only myPrev and myNext variables can be derived from the given information.

A few small errors lost ½ point: a comparison that would crash with a one-element list, and a comparison of pointers (e.g. myMedian < myMedian.myNext).