

### Exam information

The average score on the exam was 17.9; the median was 17. (Were you to receive a grade of 23 on both your midterm exams, 45 on the final exam, plus good grades on homework and lab, you would receive an A–; similarly, a test grade of 16 may be projected to a B–.)

### Problem 1 (3 points)

This problem involved identifying an incorrect parameter declaration in a method that would cause two apparently identical `LineNumber` objects to not be identified as identical by the `HashMap` `put` method.

First, we can identify some noncandidates for the incorrectly declared method. We can't touch anything in the `HashMap` class, and `Expression` methods aren't involved in hash table insertion. That narrows down the problem to the `LineNumber` class.

We know that the `put` method, given a `LineNumber` and some associated value as argument, does the following:

1. It calls `LineNumber.hashCode` to determine the hash value of the `LineNumber`.
2. Using `LineNumber.equals`, it compares the `LineNumber` to everything in the table with the same hash value to see if a key/value pair with the given `LineNumber` as key is already in the table.
3. If it finds a match, it replaces the associated value by the second argument to `put`. If not, it inserts the `LineNumber/value` pair into the table.

The `get` method does steps 1 and 2, and returns the associated value if it finds the `LineNumber` among the keys.

An important step in the design of hashable objects is the overriding of `Object.hashCode` and `Object.equals`. `Object.hashCode` merely returns the machine address of the object; `Object.equals` returns `true` only when an object is compared to itself. The two `LineNumber` objects are seen as different by the `Object` methods: `Object.hashCode` returns two different hash values and `Object.equals` returns `false` when comparing them.

Thus a possible cause of the problem is the failure to override `hashCode` or `equals`. Supplying a parameter for `hashCode` or a parameter of an incorrect type to `equals`, e.g. saying

```
boolean equals (LineNumber n) ...
```

instead of

```
boolean equals (Object obj) ...
```

would produce the observed behavior; either error would result in an inability to retrieve the first `LineNumber` object from the table.

Mentioning the fact that the two `LineNumbers` were different objects earned you at least 1 point for this problem. Mentioning the relevance of the `equals` or `hashCode` method earned you at least 1 other point.

**Problem 2 (6 points)**

You were to design a data structure named `songsByGenre` for songs that would yield fast retrieval by genre. A `HashMap` table that associates each genre with *the collection of songs of that genre* would be the most appropriate data structure. One might use an `ArrayList` or a `LinkedList` to represent the collection. Adding a song to this data structure would involve the following `put` method:

```
void put (Song s) {
    LinkedList list = songsByGenre.get (s.genre ( ));
    if (list == null) {
        list = new LinkedList ( );
        songsByGenre.put (s.genre ( ), list);
    }
    list.add (s);
}
```

Parts a and b were each worth 3 points. Part a involved the design of the data structure. If your design involved linear search through all the genres rather than the direct access of a hash table, you lost 1 point in this part. Part b involved the implementation of the design as reflected by the `put` method. The answer

```
void put (Song s) {
    songsByGenre.put (s.genre ( ), s);
}
```

was *very* common, perhaps due to a misconception that consecutive calls to `HashMap.put` with the same keys would somehow collect the corresponding values instead of *replacing* the mapping in the table. This answer earned 0 points in part b.

**Problem 3 (7 points)**

This problem requested a BinaryTree method that built an expression tree that represents its parenthesis-free String argument, taking into account the conventional precedence relationship between addition and multiplication.

Here's a solution.

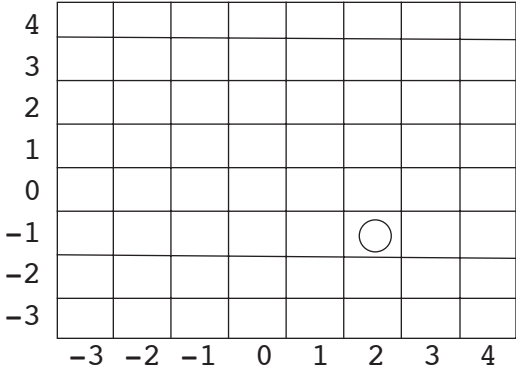
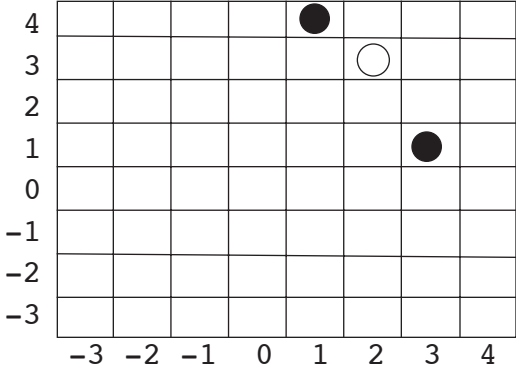
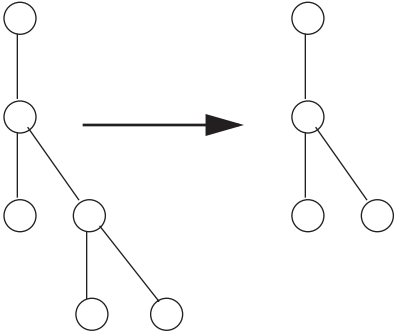
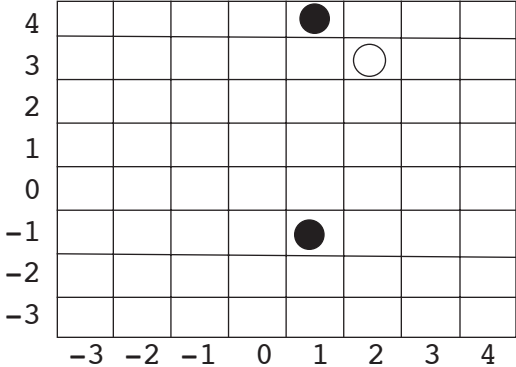
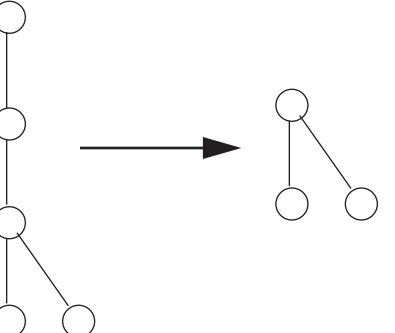
```
public BinaryTree noParensArithExprTree (String expr) {
    BinaryTree t = new BinaryTree ( );
    t.myRoot = helper (expr);
}

private TreeNode helper (String expr) {
    if (expr.length ( ) == 1) {
        return new TreeNode (expr);
    }
    String opnd1, opnd2;
    int plusPos = expr.indexOf ('+');
    int timesPos = expr.indexOf ('*');
    if (plusPos != -1) {
        opnd1 = expr.substring (0, plusPos);
        opnd2 = expr.substring (plusPos+1);
        return new TreeNode ("+", helper (opnd1), helper (opnd2));
    } else {
        opnd1 = expr.substring (0, timesPos);
        opnd2 = expr.substring (timesPos+1);
        return new TreeNode ("*", helper (opnd1), helper (opnd2));
    }
}
```

Most solutions to this problem were close to correct. 1-point deductions resulted from off-by-one errors in a call to substring or incorrectly checking for a variable name. For answers further from correct (e.g. a couple of you submitted pseudocode instead of Java), you needed a recursive call to get at least 2 points.

**Problem 4 (7 points)**

This problem involved devising informative test cases for deletion in a quad tree. The test suite we were looking for was four out of the following five cases:

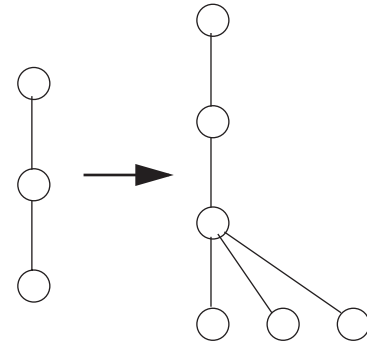
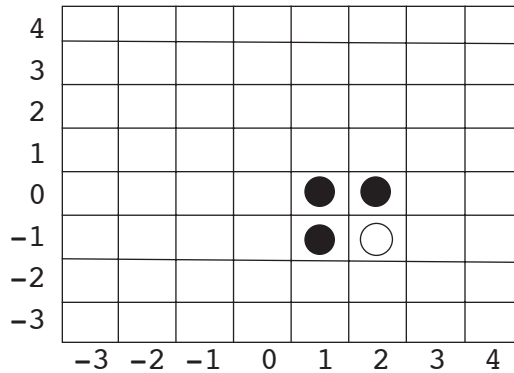
<i>description</i>	<i>situation</i>	<i>tree transformation</i>
deletion leaving an empty tree		
replacement of two nodes by one in the next larger region		
the previous case, with more than one level of the tree restructured		

*description*

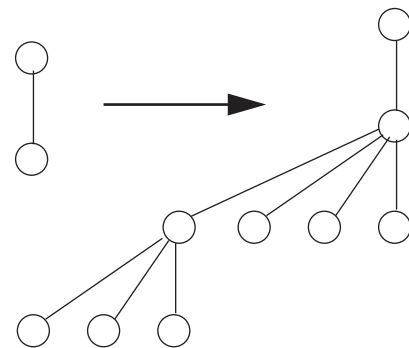
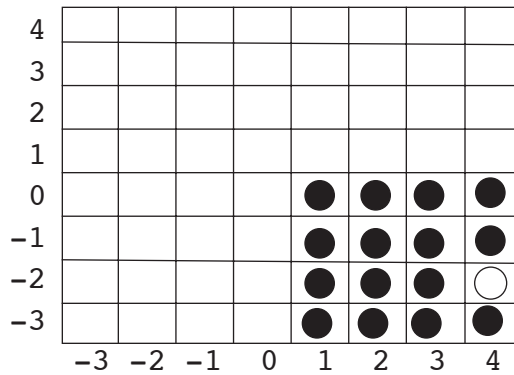
*situation*

*tree transformation*

replacement of  
a full node by a  
mixed node  
with three  
children



the previous  
case, with  
more than one  
level of the tree  
restructured



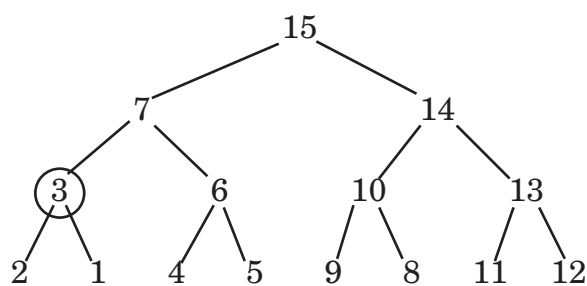
You lost 1 point for a case that merely involved removal of a nonempty branch of the tree from a node with other children, on the grounds that it provides less evidence of correctness of the code than a removal that requires more radical restructuring of the tree. You received a 2-point deduction for a test case that was the same as one of your other cases except for the tree level of the node being deleted.

**Problem 5 (7 points)**

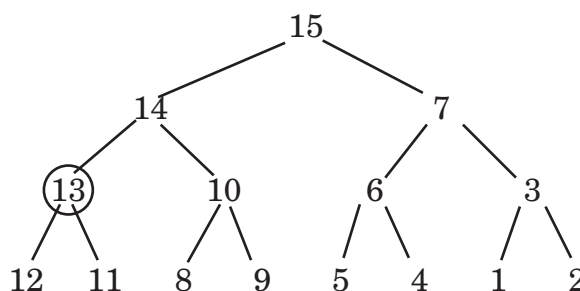
For this problem, you were to write a method that deletes the element at array position  $k$  in a binary max heap of  $n$  elements, then to provide a tight big-Oh estimate of the running time of your method. Where the element to delete is the top of the heap, we merely use the standard heap removal algorithm. Deleting the last element in the heap involves only a removal of the element from the `myValues` array. For other elements, the most efficient removal method essentially involves two steps:

1. Restore the heap *shape* by replacing the  $k$ th element by the last element.
2. Restore the heap *ordering* by bubbling the new  $k$ th element up or down.

Given below are two examples that show that *both* bubbling directions must be considered (the element being deleted is circled in each example).



requires bubbling up of the 12  
after it replaces the 3



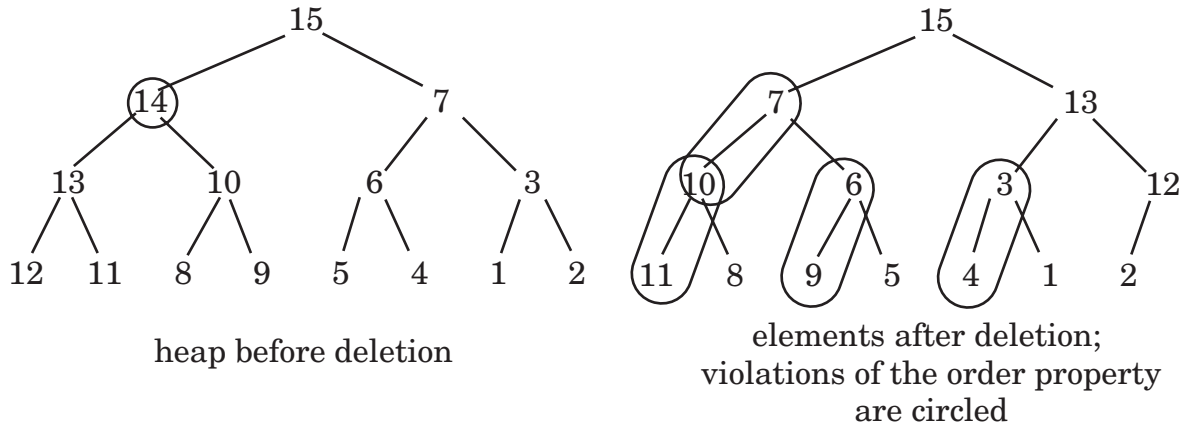
requires bubbling down of the 2  
after it replaces the 13

The worst case time required by this algorithm depends on the bubbling direction of the new  $k$ th element. If it moves up the tree, the worst case time is  $O(\log k)$ , the depth of the heap representing elements  $0, \dots, k$ . If it moves down the tree, it will move at most  $O(\log n - \log k)$  levels.

A slower algorithm removes the  $k$ th element from the `myValues` vector (an  $O(n-k)$  operation) and then applies an algorithm used in Heapsort to create a heap from  $n$  elements in  $O(n)$  operations. The heap creation method is named `heapify` in *Data Structures into Java*. Total time required is  $O(n)$ .

An even slower method merely does heap insertion of elements  $k+1$  through  $n-1$ . In general, this is essentially an  $O(n \log n)$  operation, though it's faster when  $k$  is close to  $n$ .

There were two noteworthy incorrect solutions. One was to promote the larger of element  $k$ 's two children, then to promote the larger child of the promoted element, and so on to the last level of the heap. This method, however, usually produces a "hole" in the bottom level of the heap; the deletions in the diagram above demonstrate this flaw. A second was to remove the  $k$ th element from the `myValues` vector and then bubble only the new  $k$ th element up or down. This sliding-down process, however, can dramatically affect the heap structure, as shown on the next page.



Part a of this problem, the design of the delete method, was worth 5 points, and part b, the analysis of your method, was worth 2. In part b, a correct estimate of the time your algorithm would take was worth 2, an insufficiently specific estimate was worth 1, and an incorrect estimate received 0. Deductions in part a were as follows:

- Overlooking the need to consider both bubbling directions in the most efficient algorithm: -1.
- Using the  $O(n)$  heap recreation algorithm: -2.
- Using that algorithm but calling it “heap sort”: -3.
- Heap-inserting everything after position  $k$ : -3.
- Repeatedly promoting the larger child, starting at the deleted element: -3.
- Some other algorithm that either didn’t produce a heap or did so in  $O(n^2)$  time or worse: -4.

Insufficiently detailed explanations resulted in avoidable deductions for some of you. Examples of overly vague solutions included using the term “insert” without specifying whether you meant heap insertion or vector insertion, using “child” without saying which child, using “successor” without saying what that was or how you would find it, or calling one of the *Data Structures into Java* methods without giving its argument.