CS61B Spring 2001 Midterm #3
Professor Mike Clancy

Read and fill in this page now.
Do NOT turn the page until you are told to do so.

Your name:

Your login name:

Your lab section day and time:

Your lab t.a.:

Name of the person sitting to your left:

Name of the person sitting to your right:

Problem 0	Total:	/20
Problem 1		
Problem 2	Problem 4	
Problem 3		

This is an open-book test. You have approximately fifty minutes to complete it. You may consult any books, notes, or other paper-based inanimate objects available to you. To avoid confusion, read the problems carefully. If you find it hard to understand a problem, ask us to explain it. If you have a question during the test, please come to the front or the side of the room to ask it.

Some students are taking this exam late. Please do not talk to them, mail them information, or post anything about the exam to news groups until after Wednesday.

This exam comprises 10% of the points on which your final grade will be based. Partial credit may be given for wrong answers. Your exam should contain five problems (numbered 0 through 4) on twelve pages. Please write your answers in the spaces provided in the test; in particular, we will not grade anything on the back of an exam page unless we are clearly told on the front of the page to look there.

Relax--this exam is not worth having heart failure about.

Problem 0 (1 point, 1 minute)

Put your login name on each page. Also make sure you have provided the information requested on the first page.

Problem 1 (5 points, 15 minutes)

Consider the code given below, which applies the heapsort algorithm to sort the values in the argument array in place.

```
/*
* REQUIRES: values is an initialized array;
* MODIFIES: values.
* EFFECTS: Sorts the elements of values from smallest to largest.
*/

public static void heapSort (int [ ] values) {
if (values.length <= 1) {
return;
}
changeToMaxHeap (values);
for (int k=values.length-1; k>0; k--) {
int temp = values[k];
values[k] = values[0];
values[0] = temp;
reheapifyDown (values, 0, k);
}
}

/*
* REQUIRES: values is an initialized array;
* 0 ≤ heapSize ≤ values.length; 0 ≤ index < heapSize;
* in the "almost-heap" rooted at values[index],
* only values[index] may violate the heap property.
* MODIFIES: values.
* EFFECTS: Moves values[index] down in its subheap if necessary
* so that all nodes in the heap rooted at values[index] satisfy
* the heap property.
*/
private static void reheapifyDown (int [ ] values, int index,
int heapSize) ...

/*
* REQUIRES: values is an initialized array;
* MODIFIES: values.
* EFFECTS: Rearranges the elements of values so that values represents a max heap.
*/
private static void changeToMaxHeap (int [ ] values) ...
```

Part a

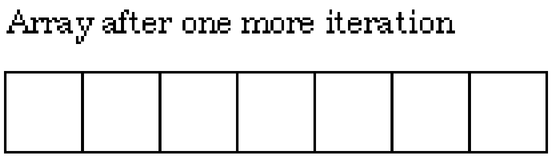
After some number of calls to *reheapifyDown* , the *values* array contains the elements

```
6 5 4 3 1 2 7
```

How many iterations of the loop have been completed? Briefly justify your answer, explaining why you know it's not more iterations and why you know it's not fewer.

Part b

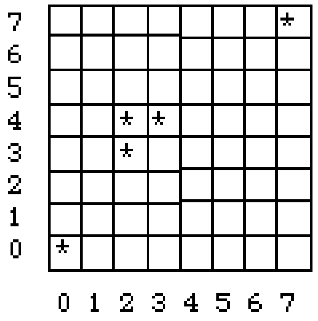
Fill in the boxes below to show the array that results after executing the next iteration of the loop. Also show how you derived your answer.

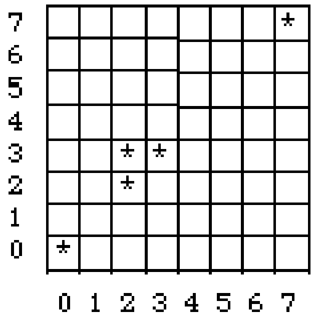


Explanation:

Problem 2 (3 points, 10 minutes)

Consider the following arrangements of creatures (represented by *'s) in an ocean.





ocean A ocean B

Suppose that each of the above oceans were represented as a *QuadTree* whose root represents the quadrant going from *(0,0)* to *(7,7)* . (The description of *QuadTree* s from project 2 appears at the end of this exam.) Call these trees *treeA* (representing ocean A) and *treeB* (representing ocean B).

Part a

The depth is the maximum number of nodes on a path from the root to a leaf. The depth of an empty tree is 0, and the depth of a tree whose only node is a leaf is 1.

Circle the correct answer below.

treeA* is deeper than *treeB

1. *treeA* and *treeB* are equally deep

2. *treeB* is deeper than *treeA*

Part b

For each of the above arrangements, give an *(x,y)* position somewhere in the quadrant going from *(0,0)* to *(7,7)* at which adding a creature would increase the depth of the tree. The answer may be "none" if no single call to *addCreature* would increase the depth of the tree.

Position at which adding a creature would increase the depth of *treeA* :

Position at which adding a creature would increase the depth of *treeB* :

Problem 3 (3 points, 8 minutes)

Consider the following algorithm to sort an array of objects named *values* . We'll call it *exam3sort* .

```
BST tree = new BST ( );
for (int k=0; k<values.length; k++) {
tree.insert (values[k]);
}
Enumeration enum = tree.inorder ( );
for (int k=0; k<values.length; k++) {
values[k] = enum.nextElement ( );
}
```

Assume that the *insert* method uses the standard insertion algorithm for binary search trees and makes no effort to keep the tree balanced, and that the *inorder* method (from homework assignment 5) returns an enumeration of the tree elements in inorder.

Indicate which of the following algorithms performs most like *exam3sort* , when performance is measured as in lab assignment 11 on elements initially in random order, in increasing order, and in decreasing order.

Quicksort (choosing the first item as the pivot, unlike in lab 11)

1. insertion sort (as in lab 11)
2. selection sort (as in lab 11)
3. merge sort (as in lab 11)

Briefly explain the similarities between your choice and *exam3sort* .

This page is intentionally left blank.

Problem 4 (8 points, 16 minutes)

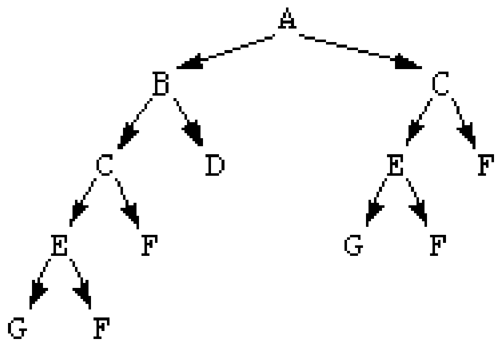
Consider an *Exam3Structure* class representing a binary tree whose framework appears below.

```
public class Exam3Structure {

// Exam3Structure methods would go here.

private TreeNode myRoot;

private class TreeNode {

/*
* INVARIANT: myItem values are always non-null .
*/

public TreeNode myLeft;
public TreeNode myRight;
public Object myItem;

/*
* REQUIRES: item != null.
* EFFECTS: Initializes a TreeNode object with the given values .
*/

public TreeNode (Object item, TreeNode left, TreeNode right) {
myItem = item;
myLeft = left;
myRight = right;
}

/*
* REQUIRES: The data structure rooted at this isn't circular.
* EFFECTS: Returns true when the data structure rooted at this
* is structurally equivalent to the data structure rooted at x.
*/

public boolean equals (Object x) {
return equalsHelper (this, (TreeNode) x);
}

private static boolean equalsHelper (TreeNode t1, TreeNode t2) {
if (t1 == null) {
return t2 == null;
} else if (t2 == null || !t1.myItem.equals(t2.myItem)) {
return false;
} else {
return equalsHelper (t1.myLeft, t2.myLeft)
&& equalsHelper (t1.myRight, t2.myRight);
}
}

/*
* EFFECTS: Return a hash value for the structure rooted at this.
*/
public int hashCode ( ) ...
}
}
```

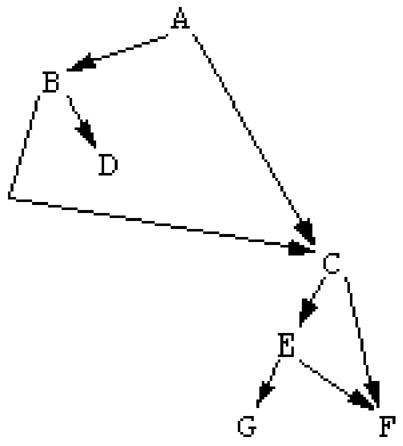
Part a

Complete an *Exam3Structure* method named *removeCopies* that optimizes the number of *TreeNode* s required to store the tree values in the following way. For any two *TreeNode* s *t1* and *t2* in the tree (representing subtrees of the *Exam3Structure* object) for which *t1.equals(t2)* , one of them should be replaced by a reference to the other. For example, *removeCopies* might transform the tree on the left in the diagram below into the structure on the right.

before call to *removeCopies*



after call to *removeCopies*



Fill in the code in the framework on the next page. Your *removeCopies* method must take time, on the average, proportional to the number of nodes in the tree, assuming that hash table operations take constant time on the average. Assume that a *hashCode* method has been defined for *TreeNode* objects, and that no changes will be made to the *Exam3Structure* object after the call to *removeCopies* .

You need no more than four lines of code (including the *return* statements) to answer this question.

Part a, continued

Fill in your answer to part a below. The *Hashtable* class used below is *java.util.Hashtable* . Its relevant method signatures appear at the end of this exam.

```
import java.util.*;

public class Exam3Structure {

// Other Exam3Structure methods and the TreeNode declaration go here.

private TreeNode myRoot;

public void removeCopies ( ) {
Hashtable table = new Hashtable (mySize);
if (myRoot != null) {
myRoot = helper (myRoot, table);
}
}
```

```
private TreeNode helper (TreeNode node, Hashtable table) {
if (node.myLeft != null) {
node.myLeft = helper2 (node.myLeft, table);
}
if (node.myRight != null) {
node.myRight = helper2 (node.myRight, table);
}
}
```

```
private TreeNode helper2 (TreeNode node, Hashtable table) {
if (table.containsKey (node)) {
return ____________________________________ ;
} else {
return ____________________________________ ;
}
}
}
```

Part b

Choose the best among the following implementations of *TreeNode.hashCode ( )* . Also justify your choice, both by giving one or more advantages for the implementation you choose and by listing at least one disadvantage for each of the others.

Implementation A

```
public int hashCode ( ) {
// Returns the default hash value, .
// computed from the object's reference.
return super.hashCode ( );
}
```

Implementation B

```
public int hashCode ( ) {
int returnValue = myItem.hashCode();
if (myLeft != null) {
returnValue += myLeft.hashCode();
}
if (myRight != null) {
returnValue += myRight.hashCode();
}
return returnValue;
}
```

Implementation C

```
public int hashCode ( ) {
int returnValue = myItem.hashCode();
if (myLeft != null) {
returnValue +=
myLeft.myItem.hashCode();
}
if (myRight != null) {
returnValue +=
myRight.myItem.hashCode();
}
return returnValue;
}
```

Implementation D

```
public int hashCode ( ) {
int returnValue = super.hashCode ();
if (myLeft != null) {
returnValue += myLeft.hashCode();
}
if (myRight != null) {
returnValue += myRight.hashCode();
}
return returnValue;
}
```

Implementation	*Advantage or disadvantage*
A	
B	
C	
D	

Best implementation: _____

Description of *QuadTree* from project 2

[A] *QuadTree* representation ... recursively divides the domain into quadrants.

Here is a picture of the ocean divided into quadrants with some positions shown:

```
------------------------------------
|0,15 | 15,15|
| | |
| | |
| | |
| | |
| | |
| | |
|0,8 |8,8 |
-----------------------------------
| 7,7| | 15,7|
| | | |
| | | |
| | 11,4| |
| |----------------|
| | 11,3| |
| | | |
| | | |
|0,0 |8,0 | 15,0|
----------------------------------
```

The lower right quadrant has been subdivided. The quad tree representation for this data structure would have three levels:

```
root (0,0 to 15,15)

/ / \ \
(0,0 to 7,7) (0,8 to 0,15) (8,0 to 15,7) (8,8 to 15,15)
/ / \ \
/ / \ \
(8,0 to 11,3) (...) (...) (...)
```

The ...'s represent the numbers of the other three squares in the subdivided lower right quadrant. The order of the child nodes is not important to us, although you should pick some order so that you can easily determine which child to look in for a given position.

There are two kinds of nodes in a quad tree:

- a leaf node holds exactly one *Creature* , where the Creature's *(x,y)* position must be in the range of that quadrant.
- an internal node holds no creatures, but has between 1 and 4 children.

The quadrants of child nodes should always be contained in the quadrant of the parent node, and the child nodes should cover 1/4 of the area of the parent. For this reason, you should use powers of two for the dimension of your quadrants, so that you can easily subdivide them.

List of *java.util.Hashtable* methods

```
public Hashtable (int initialCapacity, float loadFactor);

public Hashtable (int initialCapacity);

public Hashtable ( );

public Object clone ( );
public boolean contains (Object value);
public boolean containsKey (Object key);
public Enumeration elements ( );
public Object get (Object key);
public boolean isEmpty ( );
public Enumeration keys ( );
public Object put (Object key, Object value);
public Object remove (Object key);
public int size ( );
public String toString ( );
```

List of *java.util.Stack* methods

```
public Stack ( );
public boolean empty ( );
public Object peek ( ); // returns top of the stack without popping it
public Object pop ( );
public Object push (Object item); // returns the pushed item
```