Sample Final Exam

This is an *open book* exam. You have 150[*][†] minutes to complete the exam. You may use the back of the preceding page for additional space if necessary, but be sure to mark you answers clearly.

In some cases, there may be a variety of implementation choices. The most credit will be given to the most elegant, appropriate, and efficient solutions.

Problem	Points	Description	Score
1	10	Short Answer	
2	10	Queues	
3	10	StackSort	
4	10	Heaps	
5	10	Binary Trees	
6	10	Hashing	
7	10	Iterators	
8	10	Time Complexity	
9	10	Graphs	
10	10	Data Structure Design	
Total	100		

I have neither given nor received aid on this examination.

Signature: ______________________________

Name: ____________________________

[*]In fact, 150 minutes is too little time for this practice exam! We are providing a larger- and harder-than-usual set of questions to help you prepare for the exam. The actual final will be closer to 7 questions.

[†]Problem 10 is harder than any question you should expect to see on the final exam. But think about it anyway: you will be surprised at how well a little stretching prepares you!

1. (*10 points*) .... Short Answer

Show your work and justify answers where appropriate.

a. A tree with n distinct elements is both a min-heap and a binary search tree. What must it look like?

b. Which tree traversal would you use to print an expression tree in human-readable form?

c. Which tree traversal would you use to evaluate an expression tree?

d. We applied sorting methods primarily to arrays and `Vector`s. Of the following sort algorithms, which are most appropriate to sort a `SinglyLinkedList`: insertion sort, selection sort, quicksort, merge sort?

e. When we rewrite a recursive algorithm to be iterative, we generally must introduce which kind of data structure to aid in simulating the recursion?

2. (*10 points*) .... Queues

Recall that the `Queue` interface may be implemented using an array to store the queue elements. Suppose that two `int` values are used to keep track of the ends of the queue. We treat the array as circular: adding or deleting an element may cause the head or tail to "wrap around" to the beginning of the array.

You are to provide a Java implementation of `class CircularQueueArray` by filling in the bodies of the methods below. Note that there is no instance variable which stored the number of elements currently in the queue; you must compute this from the values of `head` and `tail`. You may **not** add any additional instance variables.

```
public class CircularQueueArray {
  // instance variables
  protected int head, tail;
  protected Object[] data;

  // constructor: build an empty queue of capacity n
  public CircularQueueArray(int n) {









  }

  // pre: queue is not fill
  // post: adds value to the queue
  public void enqueue(Object value) {









  }
```

```
  // pre: queue is not empty
  // post: removes value from the head of the queue
  public Object dequeue() {









  }

  // post: return the number of elements in the queue
  public int size() {






  }

  // post: returns true iff queue is empty
  public boolean isEmpty() {






  }

  // post: returns true iff queue is full
  public boolean isFull() {






  }
}
```

Name: _______________

3. (*10 points*) .... Stacks

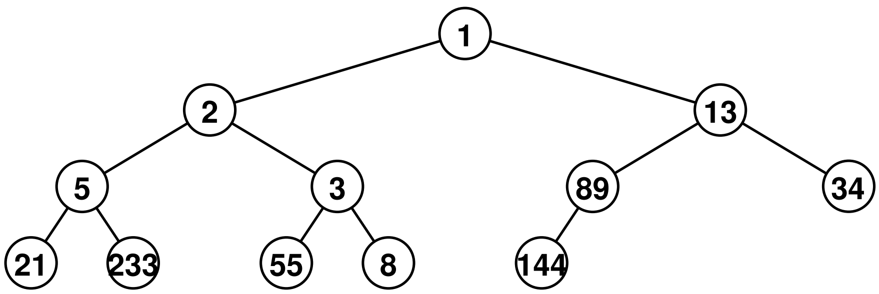
Suppose you are given an iterator that will let you access a sequence of `Comparable` elements. You would like to sort them, but the only data structure available to you is an implementation of the `Stack` interface in the structure5 package (say, `StackList`), and memory constraints only allow you to make a small (constant) number of stacks. Because the elements are available only through an `Iterator`, so you must process each item as it is returned by the `next()` method of the `Iterator`. The sort method should return a `Stack` containing the sorted elements, with the smallest at the top of the stack. Please fill in the body of the method.

```
public static Stack StackSort(Iterator iter) {
  // pre: iter is an Iterator over a structure containing Comparables
  // post: a Stack is returned with the elements sorted, smallest on top




}
```

Name: _______________

4. (*10 points*) .... Heaps

Recall the definition of a min-heap, a binary tree in which each node's value is no bigger than that of each of its descendants. For the rest of this question, we presume the Vector implementation of heaps (`class VectorHeap`). Consider the following tree, which is a min-heap.



a. Show the order in which the elements would be stored in the `Vector` underlying our `VectorHeap`.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

b. Show the steps involved in adding the value 4 to the heap. **Use drawings of the tree, not the vector.**

Name: _______________

c. Using the original tree (not the one with the 4 added), show the steps involved in removing the minimum value of the heap.

d. Why is the `VectorHeap` implementation of a priority queue better than one that uses a linked list implementation of regular queues, modified to keep all items in order by priority? Hint: Your answer should compare the complexities of the add and remove operations.

Name: _______________

5. (*10 points*) .... Binary Trees

Suppose we have a `BinaryTree` that contains only `Comparable` values.

a. It is often useful to find the minimum and maximum values in the tree. Implement the method `maximum` as a member of `class BinaryTree`. As a guide, relevant sections of `BinaryTree.java` from the `structure5` package are included after this question. Your method should return the `Comparable` that is the maximum value in the tree. It should return `null` if called on an empty tree.

```
public Comparable maximum() {
  // pre: the values in this tree are all Comparable
  // post: the maximum value in the tree is returned










}
```

b. What is the worst-case complexity of `maximum` on a tree containing n values?

c. What is the complexity of `maximum` on a full tree containing n values?

Name: _______________

d. Consider the following method, which I propose as a member of `class BinaryTree`:

```
public boolean isBST() {
  // post: returns true iff the tree rooted here is a binary search tree
  if (this == EMPTY) return true;
  return left().isBST() && right().isBST();
}
```

As written, this method will not always return the correct value. Explain why, then provide a correct method. You may use `minimum()` and `maximum()` from part (a), as well as any other methods of `BinaryTree`.

```
public boolean isBST() {




}
```

e. In `class BinaryTree`, why is the `setLeft()` method `public`, but the `setParent()` method is `protected`?

Name: _______________

```
public class BinaryTree {
    protected Object val; // value associated with node
    protected BinaryTree parent; // parent of node
    protected BinaryTree left; // left child of node
    protected BinaryTree right; // right child of node
    // The unique empty node
    public static final BinaryTree EMPTY = new BinaryTree();

    // A one-time constructor, for constructing empty trees.
    private BinaryTree() {
        val = null; parent = null; left = right = this;
    }

    // Constructs a tree node with no children.  Value of the node
    // is provided by the user
    public BinaryTree(Object value) {
        val = value; parent = null; left = right = EMPTY;
    }

    // Constructs a tree node with no children.  Value of the node
    // and subtrees are provided by the user
    public BinaryTree(Object value, BinaryTree left, BinaryTree right) {
        this(value);
        setLeft(left);
        setRight(right);
    }

    // Get left subtree of current node
    public BinaryTree left() {
        return left;
    }

    // Get right subtree of current node
    public BinaryTree right() {
        return right;
    }

    // Get reference to parent of this node
    public BinaryTree parent() {
        return parent;
    }

    // Update the left subtree of this node.  Parent of the left subtree
    // is updated consistently.  Existing subtree is detached
    public void setLeft(BinaryTree newLeft) {
        if (isEmpty()) return;
        if (left.parent() == this) left.setParent(null);
        left = newLeft;
        left.setParent(this);
    }

    // Update the right subtree of this node.  Parent of the right subtree
    // is updated consistently.  Existing subtree is detached
    public void setRight(BinaryTree newRight) {
        if (isEmpty()) return;
        if (right.parent() == this) right.setParent(null);
        right = newRight;
        right.setParent(this);
    }
```

```
// Update the parent of this node
protected void setParent(BinaryTree newParent) {
    parent = newParent;
}

// Returns the number of descendants of node
public int size() {
    if (this == EMPTY) return 0;
    return left().size() + right.size() + 1;
}

// Returns reference to root of tree containing n
public BinaryTree root() {
    if (parent() == null) return this;
    else return parent().root();
}

// Returns height of node in tree.  Height is maximum path
// length to descendant
public int height() {
    if (this == EMPTY) return -1;
    return 1 + Math.max(left.height(),right.height());
}

// Compute the depth of a node.  The depth is the path length
// from node to root
public int depth() {
    if (parent() == null) return 0;
    return 1 + parent.depth();
}

// Returns true if tree is full.  A tree is full if adding a node
// to tree would necessarily increase its height
public boolean isFull() {
    if (this == EMPTY) return true;
    if (left().height() != right().height()) return false;
    return left().isFull() && right().isFull();
}

// Returns true if tree is empty.
public boolean isEmpty() {
    return this == EMPTY;
}

// Return whether tree is complete.  A complete tree has minimal height
// and any holes in tree would appear in last level to right.
public boolean isComplete() {
    int leftHeight, rightHeight;
    boolean leftIsFull, rightIsFull, leftIsComplete, rightIsComplete;
    if (this == EMPTY) return true;
    leftHeight = left().height();
    rightHeight = right().height();
    leftIsFull = left().isFull();
    rightIsFull = right().isFull();
    leftIsComplete = left().isComplete();
    rightIsComplete = right().isComplete();

    // case 1: left is full, right is complete, heights same
    if (leftIsFull && rightIsComplete &&
```

```
            (leftHeight == rightHeight)) return true;
        // case 2: left is complete, right is full, heights differ
        if (leftIsComplete && rightIsFull &&
            (leftHeight == (rightHeight + 1))) return true;
        return false;
    }

    // Return true iff the tree is height balanced.  A tree is height
    // balanced iff at every node the difference in heights of subtrees is
    // no greater than one
    public boolean isBalanced() {
        if (this == EMPTY) return true;
        return (Math.abs(left().height()-right().height()) <= 1) &&
                left().isBalanced() && right().isBalanced();
    }

    // Returns value associated with this node
    public Object value() {
        return val;
    }
}
```

Name: ______________

6. *(10 points)* .... Hashing

a. What is meant by the "load factor" of a hash table?

b. We take care to make sure our hash functions return the same hash code for any two equivalent (by the `equals()` method) objects. Why?

c. We also said that a good size for a hash table would be a prime or "almost prime" number. Why?

d. A hash table with *ordered linear probing* maintains an order among keys considered during the rehashing process. When the keys are encountered, say, in increasing order, the performance of a failed lookup approaches that of a successful search. Describe how a key might be inserted into the ordered sequence of values that compete for the same initial table entry.

e. Is the hash table constructed using ordered linear probing as described in part (d) really just an ordered vector? Why or why not?

f. One means of potentially reducing the complexity of computing the hash code for `String`s is to compute it once – when the `String` is constructed. Future calls to `hashCode()` would return this precomputed value. Since Java `String`s are immutable, that is, they cannot change once constructed, this could work. Do you think this is a good idea? Why or why not?

7. (*10 points*) .... Iterators

Given a pair of iterators that return data in sorted order, a `Mergerator` merges them into a single iterator that returns all data in sorted order. Here is a simple example of how it might be used:

```
String a[] = { "brown", "cow", "moo" };
String b[] = { "another", "sentence", "sorted" };

AbstractIterator<String>> iter1 = ArrayIterator<String>(a);
AbstractIterator<String>> iter2 = ArrayIterator<String>(b);

Iterator<String> merger = new Mergerator(iter1,iter2);
while (merger.hasNext()) {
  System.out.println(merger.next());
}
```

This code prints out:

```
      another brown cow moo sentence sorted
```

This problem asks about implementing the `Mergerator` class to provide the `AbstractIterator` methods: `reset()`, `next()`, `hasNext()`. (You do not need to implement `get()`; also, your `next()` method should not rely on `get()`.)

Complete the code for the `Mergerator` class. Declare any instance variables you will use in your class and implement the constructor and methods (`reset`, `hasNext`, and `next`). Your class should be parameterized by the type `E` of elements that will be returned by the underlying iterators.

```
public class Mergerator <E extends Comparable<E>> extends AbstractIterator<E> {



    public Mergerator(                                       ) {












    }
```

```
    public void reset() {








    }

    public boolean hasNext() {








    }

    public E next() {
















    }
}
```

Name: _______________

8. *(10 points)* .... Time Complexity

Suppose you are given n lists, each of which is of size n and each of which is sorted in increasing order. We wish to merge these lists into a single sorted list L, with all n^2 elements. For each algorithm below, determine its time complexity (Big O) and justify your result.

a. At each step, examine the smallest element from each list; take the smallest of those elements, remove it from its list and add it to the end of L. Repeat until all input lists are empty.

b. Merge the lists in pairs, obtaining $\frac{n}{2}$ lists of size $2n$. Repeat, obtaining $\frac{n}{4}$ lists of size $4n$, and so on, until one list remains.

Name: _______________

9. (*10 points*) .... Graphs

a. In studying Prim's Algorithm, we saw that some edges removed from the priority queue are not useful in that they lead to a vertex we have already visited. This is true even if we only insert edges leading to unvisited vertices into the priority queue. How is this possible?

b. Recall the `Trie` structure you implemented for the Lexicon lab. It was a general tree, where a node in the tree could have an arbitrary number of children. Trees are nothing more than graphs with some restrictions on the edges allowed. You could store the same information in a `Graph` by making a `Vertex` for each tree node and adding `Edge`s representing the links to the children. Which `Graph` implementation would you use for this, and why? How does its time and space complexity compare to your `Trie` implementation?

Name: _______________

c. Consider the following definition of a graph.

Def: A graph G consists of a set V, whose members are called the vertices of G, together with a set E, of edges, which are pairs of *distinct* vertices from V (no edges from a vertex back to itself).

Prove by induction that an undirected graph G with n vertices has at most $n(n-1)/2$ edges.

Name: _______________

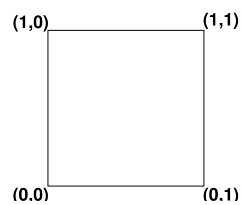
10. (*10 points*) .... Data Structure Design

Be careful to answer all parts of this ridiculously long question!

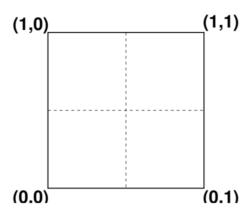
The structures we have studied this semester all store Java `Object`s. Many structures place no restrictions on the types of the objects that can be stored. `OrderedStructure`s limit the types of objects that may be store to those which implement the `Comparable` interface, to allow storage based on an ordering of the objects using the `compareTo()` method. If our objects have Cartesian coordinates, perhaps they could be stored in a structure that uses this information. These objects might be mesh elements in a scientific computation using the finite element method, particles in a particle-in-cell simulation, or coordinates of cities or other points of interest on a map. Much of what we look at in this problem works for coordinates of any dimension, but for simplicity, we will consider objects with two-dimensional coordinates. Such `Object`s can implement the following interface.

```
public interface CoordObject {
    // pre: none
    // post: returns object's x coordinate
    public double getX();

    // pre: none
    // post: returns object's y coordinate
    public double getY();
}
```

Now we wish to store n `CoordObject`s in a structure. Suppose we know that all of these objects have coordinates some range. Again for simplicity, we will say they are all in the square bounded by $(0,0)$ and $(1,1)$.



Suppose we break the square into 4 smaller squares:



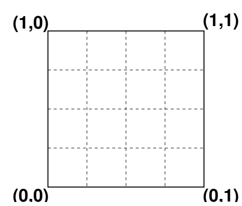
Name: _______________

Objects now can be placed into one of four groups (bins) based on coordinates. We can define methods `add()`, `contains()`, *etc.* that insert into or search in the right bin.

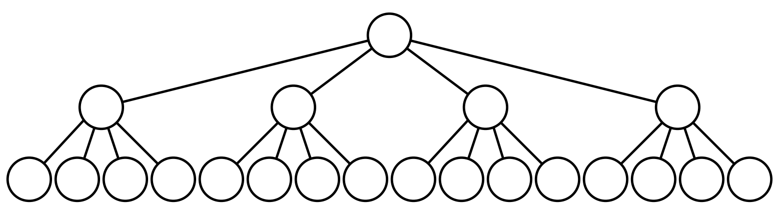
a. What structure would you use to store the objects that belong in each bin?

b. Suppose the n objects are inserted and are approximately evenly distributed among the four bins. How many comparisons will be needed for a call to `contains()` on an object that is not found? Be sure to show how you get your answer.

Perhaps we should break into more squares:



We could continue adding more and more bins until a reasonably small number of objects ends up in each bin. We can view the structure that we are creating as a *quadtree* (or in three dimensions, an *octree*). Consider this tree:



Name: _______________

The root represents the square we started with (the universe). The next level represents the four-way subdivision, and the leaves represent the 16-way subdivision. The nodes are called `quadrants`. Only the leaf nodes contain objects stored in our quadtree. The others implicitly "contain" the objects contained by their descendants. During a search, the coordinates of an object are used to guide a search through the tree to the appropriate leaf quadrant.

c. Describe how a call to `contains()` would work on a quadtree of height h.

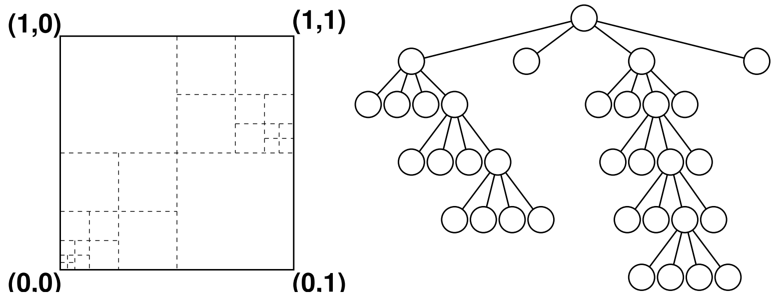
d. Again assuming a regular distribution of the objects among the bins, what is the complexity of an unsuccessful `contains()` method call? Explain briefly.

e. Now suppose that the distribution is not regular at all – for the tree of height h, the n objects end up with half in one bin and half in another. How does this affect the complexity of an unsuccessful `contains()` call?

Perhaps we should add more levels to the tree. If we do this "quadrant refinement" only for the part of the tree that becomes overcrowded, we can say our tree is *adaptive*. We start with only a root node. Objects are added to the root until a capacity threshold is exceeded. The root is then divided into four child quadrants, and the objects it contains are distributed among the four children. If subsequent additions of objects cause one of the children to exceed its capacity, it does the same. Parts of the tree corresponding to parts of the universe that contain most of the objects are refined more, while less crowded parts remain coarse.

Name: _______________

If object coordinates are most likely to be near the origin and near $(0.9, 0.6)$, we might end up with a tree that looks like this:



f. In both figures above, mark with an X the quadrant which would contain an object with coordinates $(0.3, 0.3)$.

g. The following is an implementation of an adaptive quadtree structure. Like the `BinaryTree` from the structure package, it is defined recursively, so subtrees are themselves valid quadtrees. Some methods required by `AbstractStructure` have been left out for simplicity. Please fill in the bodies of the methods marked with `// ** WRITE THIS METHOD`. You need not explicitly enforce preconditions. Be sure to make use of the `protected` methods I have provided for you as appropriate.

```
public class Quadrant extends AbstractStructure {
    // the bounding coordinates of the space occupied by this quadrant
    protected double minX, minY, maxX, maxY;

    // child quadrants, all non-null for interior quadrants
    // all null for leaf quadrants
    protected Quadrant children[];

    // constants to refer to the child quadrants in meaningful ways
    // these are the indices into the children array
    public static final int NorthEast=0;
    public static final int NorthWest=1;
    public static final int SouthWest=2;
    public static final int SouthEast=3;

    // Threshold at which the quadrant should refine into four new
    // children and distribute its contents among them.
    // We could make this a parameter, but it is constant for simplicity
    protected static final int capacity=64;
```

Name: _______________

```
// The contents
protected Structure contents;

// create a new quadrant.  Only leaf quadrants are created by the
// constructor.  Leaf quadrants may become interior quadrants
// if they are later refined during an add operation
public Quadrant(double minX, double minY, double maxX, double maxY) {
    this.minX=minX; this.minY=minY; this.maxX=maxX; this.maxY=maxY;
    children[0]=null; children[1]=null; children[2]=null; children[3]=null;
    // we can use any Structure here, for simplicity, a SinglyLinkedList
    contents=new SinglyLinkedList();
}

// pre: none
// post: returns whether this is a leaf quadrant
protected boolean isLeaf() {
    // contents is non-null only at leaf quadrants
    return contents!=null;
}

// pre: this must be an interior quadrant
//      (x,y) of object must lie within the bounding box of this quadrant
// post: return child Quadrant containing the coordinates of the object
protected Quadrant getChildQuadrant(Object obj) {
    CoordObject coord=(CoordObject)obj;
    double x=coord.getX(); double y=coord.getY();
    double midX=(minX+maxX)/2; double midY=(minY+maxY)/2;
    if (x < midX) {
        if (y < midY) return children[SouthWest];
        else return children[NorthWest];
    } else {
        if (y < midY) return children[SouthEast];
        else return children[NorthEast];
    }
}

// refine this quadrant
// pre: the quadrant is a leaf and has exceeded its capacity
// post: four child quadrants are created, the contents of this
//       quadrant are dispersed among them.  This quadrant
//       is transformed into an interior quadrant
protected void refine() {
    // first create the children
    children=new Quadrant[4];
    double midX=(minX+maxX)/2;
    double midY=(minY+maxY)/2;
    children[SouthWest]=new Quadrant(minX, minY, midX, midY);
    children[NorthWest]=new Quadrant(minX, midY, midX, maxY);
    children[SouthEast]=new Quadrant(midX, minY, maxX, midY);
    children[NorthEast]=new Quadrant(midX, midY, maxX, maxY);

    // loop over the contents, insert as appropriate
    Iterator i=contents.iterator();
    while (i.hasNext()) {
        Object obj=i.next();
        Quadrant child=getChildQuadrant(obj);
        child.add(obj);
```

```
    }

    // destroy contents by letting the garbage collector have at it
    contents=null;
}

// pre: object being added is a CoordObject, whose coordinates
//      lie within the bounding box of this quadrant
// post: the object is inserted into this Quadrant or one of its
//       children.  If this is an interior quadrant, the request
//       is passed to a child quadrant.  If this is a leaf, it is
//       inserted here.  If this add causes the quadrant's capacity
//       to be exceeded, the quadrant will be refined.
public void add(Object obj) {
    // ** WRITE THIS METHOD











}
```

Name: _______________

```
// pre: none
// post: returns number of objects contained in this quadrant for a leaf,
//       sum of sizes of children for non-leaf
public int size() {
    // ** WRITE THIS METHOD










}

// pre: none
// post: clears this quadrant (for a leaf) or all of its children
//       (for interior)
public void clear() {
    // ** WRITE THIS METHOD










}
```

Name: _______________

```
    // pre: none
    // post: removes the specified object from this quadrant (for a leaf)
    //       or the appropriate child (for interior)
    //       object removed is returned, or null is returned if the object
    //       was not found
    public Object remove(Object obj) {
        // ** WRITE THIS METHOD








    }

    // pre: none
    // post: return true iff the object is contained in this quadrant's
    //       contents (for a leaf) or in the appropriate child quadrant
    //       (for interior)
    public boolean contains(Object obj) {
        // since this method is almost the same as remove, you need
        // not provide it.
    }
}
```

h. What is the complexity of your `add()` method? Explain.