Sample Final Exam

This is a *closed 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.

Be sure to give yourself enough time to answer each question— the points should help you manage your time.

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	AVL Trees	
8	10	Time Complexity	
9	10	Graphs	
Total	90		

I have neither given nor received aid on this examination.

Signature: ______________________________

Name: ____________________________

---

[*] In fact, 150 minutes may be too little time for these problems! This is a problem suite to help you prepare for the exam. The actual final will most likely have 7-8 questions.

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 would you use to sort a `SinglyLinkedList`: insertion sort, selection sort, quicksort, merge sort? Briefly explain your answer.

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

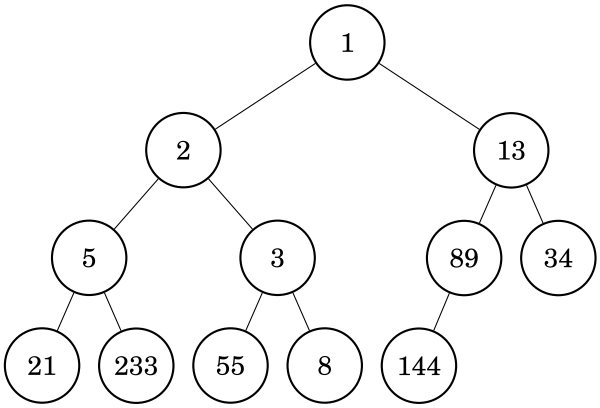
Suppose you are given an iterator that will let you access a sequence of `Comparable` elements. You would like to sort them, but the only data structure available to you is an implementation of the `Stack` interface in the structure5 package (say, `StackList`), and memory constraints only allow you to 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 <E extends Comparable<E>>
                            Stack<E> StackSort(Iterator<E> iter) {
  // pre: iter is Iterator over structure containing elements of type E
  // 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.**

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 an instance of `BinaryTree<E>` where type `E` implements `Comparable`.

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`. Relevant sections of `BinaryTree.java` from the structure5 package are included on pages 10–12 to guide you. Your method should return the value that is the maximum value in the tree. It should return `null` if called on an empty tree. We put in an assert to get you started.

```
public static <E extends Comparable<E> > E maximum() {
  // 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 static <T extends Comparable<T> > boolean isBST(BinaryTree<T> tree) {
  // post: returns true iff the tree rooted here is a binary search tree
  if (isEmpty() ) 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`?

```
public class BinaryTree<E>{
    protected E val; // value associated with node
    protected BinaryTree<E> parent; // parent of node
    protected BinaryTree<E> left; // left child of node
    protected BinaryTree<E> right; // right child of node

    // 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(E value) {
        val = value;
        parent = null;
        left = new BinaryTree<E>();
        right = new BinaryTree<E>();
    }

    // Constructs a tree node with tree children.  Value of the node
    // and subtrees are provided by the user
    public BinaryTree(E value, BinaryTree<E> left, BinaryTree<E> right) {
        this(value);
        if (left == null) { left = new BinaryTree<E>(); }
        setLeft(left);
        if (right == null) { right = new BinaryTree<E>(); }
        setRight(right);
    }

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

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

    // Get reference to parent of this node
    public BinaryTree<E> 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<E> 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<E> newRight) {
```

```
        if (isEmpty()) return;
        if (right != null && right.parent() == this) right.setParent(null);
        right = newRight;
        right.setParent(this);
    }

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

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

    // Returns reference to root of tree containing n
    public BinaryTree<E> 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 (isEmpty()) 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 (isEmpty()) return true;
        if (left().height() != right().height()) return false;
        return left().isFull() && right().isFull();
    }

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

    // 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 (isEmpty()) 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 (isEmpty()) return true;
        return (Math.abs(left().height()-right().height()) <= 1) &&
                left().isBalanced() && right().isBalanced();
    }

    // Returns value associated with this node
    public E 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?

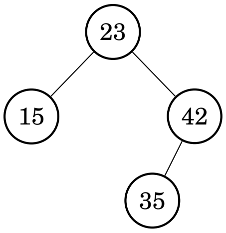
f. One means of potentially reducing the complexity of computing the hash code for `String`s is to compute it once – when the `String` is constructed. Future calls to `hashCode()` would return this precomputed value. Since Java `String`s are immutable, that is, they cannot change once constructed, this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

7. (*10 points*) .... AVL Trees

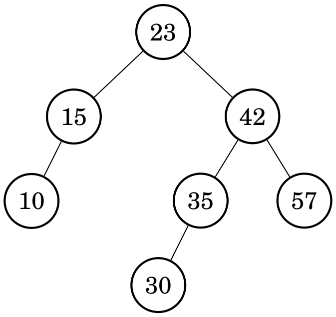
a. Recall that AVL Trees do not necessarily maintain a perfect balance, but require that each tree node's children satisfy the *AVL Condition*. State the AVL condition.

b. We could make our balance criteria more strict and require that the tree must have minimum height for its size, but this is rarely done. Give two reasons why this is the case.

c. Label each node in the following tree with its balance factor. Is this tree an AVL tree?



d. Show the result of performing a left rotation on nodes 23 and 42 in the following tree. This rotation should make 42 the root of the resulting tree.



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 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. How is this possible, given that we insert only edges leading to unvisited vertices into the priority queue?

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?

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, and there cannot be two different edges between the same pair of vertices).

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