You are to answer at least nine (9) questions, including at least one from each of Sections A, B, C, and D.

**Section A: ADT Applications**

**A.1.** Show the string of characters that would be printed if the `printPreOrder()` method (see below) were applied to the binary tree below (whose nodes contain characters).

```java
public void printPreOrder(RecBinTree t) {
    if (t.isEmpty())
        {} //
    else {
        System.out.print( t.getRoot() );
        printPreOrder(t.leftSubtree());
        printPreOrder(t.rightSubtree());
    }
}
```

```
        'k'
    /`
   / `
  /  `
 /   `
/     `

'j'
/
/  
/    
/      
/        

'e'
```
A.2. Let $T$ be the following binary search tree (BST).

```
          21
           / \  \\
          /   \\    \\
         /     \\  \\
        /       \\  \\
       /        \\  \\
      /         \\  \\
     /          \\  \\
    /           \\  \\
   /            \\  \\
  /             \\  \\
 /              \\  \\
/                \\  \\
```

Recall that, to insert an item $x$ into a BST, we first search for it. If $x$ is not already in the tree, the search will have ended at an empty subtree. We replace that empty subtree with a node containing $x$. To delete an item $x$ from a BST, we first search for it. If $x$ occurs in the tree, the search will have ended at a node $v$ containing $x$. We then apply the method shown below to “delete” node $v$.

```java
public void deleteNode(Node v) {
    if (v is a leaf) {
        detach v from its parent;
    } else if (v has exactly one child) {
        let w be the child of v;
        detach v from w;
        if (v is the root) {
            make w the root;
        } else {
            let u be the parent of v;
            attach w to u (so that w replaces v as a child of u);
        }
    } else { // v has two children
        let y be the "leftmost" node in right subtree of v
        copy into v the value contained in node y;
        deleteNode(y); // recursively delete node y
    }
}
```

Apply to $T$ each of the following operations and for each one show the tree that results. (*Note:* Each operation is to be performed **upon** $T$. Do not, for example, apply (c) to the result of
applying (a) and then (b) to $T$.) You need not draw the entire tree each time, but show the local area within $T$ where changes occurred, as well as enough surrounding context to leave no doubt as to what the updated tree looks like.

(a) insert 15  
(b) insert 33  
(c) delete 7  
(d) delete 34  
(e) delete 6
A.3. Apply the HeapSort algorithm, as given below, to the sequence of integers 8, 12, 5, 4, 0, 9. Show what the heap would look like after each iteration in each phase of the algorithm’s execution. (You need not show what the heap looks like during the middle of a siftUp or siftDown.)

```java
/* Phase 1: Build Heap */
int h = empty heap;
while (there are more items in the collection) {
    int x = next value in the collection;
    h.insert(x);
}
/* Phase 2: Dismantle Heap */
while ( !h.isEmpty() ) {
    int x = h.getMin(); // Let x be value in root of h.
    PRINT x;           // Or store x in "next" location of some array.
    h.deleteMin();     // Copy value of "last" leaf into root, delete
                     // that leaf, and sift down from root.
}
```

Phase 1:

Phase 2:
Section B: Programming with ADT's

B.1. Assume that Stack and Queue are classes among whose methods include the following:

Stack:
- boolean isEmpty(): returns true iff stack is empty
- Object topOf(): returns item at top of stack
- void pop(): removes item from top of stack
- void push(Object item): inserts item at top of stack

Queue:
- boolean isEmpty(): returns true iff queue is empty
- Object frontOf(): returns item at front of queue
- void dequeue(): removes item from front of queue
- void enqueue(Object item): inserts item at rear of queue

Supply the missing body of the method reverseStack(), which, given a stack, reverses its elements (i.e., turns the stack upside down). Any variable used in your method must be of type Object, int, boolean, Stack, or Queue. In particular, arrays are not to be used.

```java
/* pre: none */
post: final contents of stk are "upside down" relative to starting contents */
public static void reverseStack( Stack stk ) {
```
B.2. Longest Hill in a List

Here are informal specifications for the `PositionalList` interface that we covered in class.

Observers:

------------
isEmpty()  : answers the question "Is list empty?"
lengthOf() : yields # of nodes in list
view()     : yields item at current position
atFront()  : answers "Is current position the front of list?"
atRear()   : answers "Is current position the rear of list?"

Notes: In an empty list, the only possible current position is both the front and the rear. An exception is thrown if view() is invoked when atRear() holds.

Navigators (Positional Mutators):

---------------------------
toFront()   : sets current position to front of list
toRear()    : sets current position to rear of list
toNext()    : moves current position one place towards rear
toPrev()    : moves current position one place towards front

Notes: When applied to an empty list, neither toFront() nor toRear() has any effect. (The current position is already there.) When applied to a list whose current position is the front (respectively, the rear), toPrev() (respectively, toNext()) throws an exception.

Node Mutators:

------------
replace(x)  : replaces current node’s contents with x
insertFront(x) : inserts new node containing x at front of list
insertRear(x)  : inserts new node containing x at rear of list
insert(x)    : inserts new node containing x so that it is the predecessor of the current position,
remove()     : removes the current node, making its successor the current position

Notes: Invocation of either replace() or remove() throws an exception if atRear() holds.

Here we are concerned with lists containing items from the “wrapper” class `Integer` (although, for convenience, you may assume that the items are of the primitive type `int`). For such a list, define an ascending hill to be any (contiguous) sublist in which each element’s successor is at least as large as itself. Similarly, a descending hill is a (contiguous) sublist in which each element’s successor is no larger than itself. A hill is any ascending hill or descending hill. The length of a hill is defined to be one less than the number of items involved in it.

As an example, consider the list \( \langle 6,2,-3,5,8,8,2,0,9 \rangle \) Among its hills are the descending hills \( \langle 6,2,-3 \rangle \) of length two and \( \langle 8,8,2,0 \rangle \) of length three and the ascending hill \( \langle -3,5,8,8 \rangle \) of length three.
Supply the missing code in the method `maxHillLength()` below, which returns the length of the longest hill in the list given to it. You may assume the existence of a method `max`, which returns the larger of the two integers passed to it.

```java
// pre: 1.lengthOf() > 0
// post: returns length of longest hill in list
public int maxHillLength(PositionalList list) {
    list.toFront();
    int prevItem = ___________________________;  
    list.toNext();  

    int maxAsc = 0, maxDesc = 0, curAsc = 0; curDesc = 0;  
    /* loop invariant:  
    prevItem = value in node immediately preceding current node &  
    maxAsc = length of longest ascending hill seen so far &  
    maxDesc = length of longest descending hill seen so far &  
    curAsc = length of longest ascending hill ending at previous node &  
    curDesc = length of longest descending hill ending at previous node  
    */  
    while (!list.atRear()) {  
        int curItem = ___________________________;  
        if ( ___________________________ )  
            { curAsc = ___________; curDesc = ___________; }  
        else if ( ___________________________ )  
            { curAsc = ___________; curDesc = ___________; }  
        else  
            { curAsc = ___________; curDesc = ___________; }  

        maxAsc = _______________; maxDesc = _______________;  

        prevItem = curItem;  
        list.toNext();  
    }  
    /* list.atRear() holds now; hence the loop invariant tells us that  
    * maxAsc and maxDesc are the lengths of the longest ascending and  
    * descending hills, respectively, in the entire list.  
    */  
    return max( maxAsc, maxDesc );
}
```
B.3. Assume the existence of a class `RecBinTree` that models the binary tree concept, following the recursive paradigm. Assume that among its methods are the following:

```java
boolean isEmpty() : returns true iff the tree has no nodes
BinTree leftSubtree() : returns subtree rooted at left child of root node
BinTree rightSubtree() : returns subtree rooted at right child of root node
```

Supply the code missing from the method `numNodes()`, which, given a binary tree, returns the number of nodes in the tree.

**Hint:** Given the methods at your disposal, it would be ridiculous to devise a non-recursive solution. To aid you in this, consider that an empty tree has no nodes and that a nonempty tree includes all the nodes in its left subtree, plus all the nodes in its right subtree, plus its root.

```java
/**
 * pre: none
 * post: returns # of nodes in t
 */
public static int numNodes( RecBinTree t ) {
    if ( ________________ )
        { return 0; }
    else {
        }
}
```
Section C: Concrete Representations of ADT’s

C.1. Consider the following Java class:

```java
public class Node {
    /* instance variables */
    private int datum;
    private Node next;

    /* constructors */
    public Node( ... ) { ... }
    ...
    ...

    /* instance methods */
    
    public void goofy() {
        Node p = this,
        Node q, r;

        while ( p != null )
        { q = p.next; r = q.next; q.next = p; p = r; }
        }
    ... ...
    }
```

By examining the class, we see that a Node object includes an int field (called datum), plus a reference to a Node (called next). Thus, it is possible to build a linked chain of Node objects. Suppose that a client program has built the following such chain:

```
        +-----+ +-----+ +-----+ +-----+ +-----+ +-----+
      | 3   |--->| 8   |--->| 1   |--->| 6   |--->| 0   |--->| 5   |---|--
      +-----+ +-----+ +-----+ +-----+ +-----+ +-----+
```

The number in each box represents the datum field of the corresponding Node object and the arrow leaving a box represents that object’s next field. As suggested by the picture, the client has a Node reference, called george, that points to the Node containing 3 in its datum field.

Show what the chain of Nodes would “look like” immediately after the invocation george.goofy() has completed execution. (To receive partial credit in case your answer is incorrect, show what the chain looks like after each iteration of the loop.)
C.2. Consider the array-based representation of (positional) lists covered in class. (An earlier question described the intended effect of each operation defined on such lists.) This representation could be expressed within a class `PosListViaArray` as follows:

```java
public class PosListViaArray implements PositionalList{
    /* instance variables */
    private int[] pred;       // array holding pointers to node predecessors
    private int[] succ;       // array holding pointers to node successors
    private Object[] contents; // array holding items on list
    private int front;        // points to array element holding first node
    private int current;      // points to array element corresponding to
                              // current position
    private int avail;        // points to first array element on avail chain
    ...
    ...
}
```

Pictured below is one concrete representation of a particular list. We use a dash ("-"), to indicate the contents of a field whose value is, at the moment, irrelevant. Recall that one of the “representation invariants” is that the only array elements containing meaningful values are those in locations zero through `numItems`.

```
prev  contents  next
+-----------------------+
 0 | 3 | - | -1 |
 1 | 2 | COW | 5 |
 2 | 4 | RAT | 1 |
 3 | 5 | GORN | 0 |
 4 | -1 | ELK | 2 |
 5 | 1 | ANT | 3 |
 6 | - | - | - |
 7 | - | - | - |
 8 | - | - | - |
 9 | - | - | - |
```

```
front  current
+---+  +---+
 4 | 1 |
 1 |
```

```
numItems
+---+
 5 |
```
(a) Illustrate the list represented above by filling in the nodes below with the correct animal names. Don’t forget to indicate the current position by drawing an arrow to it.

```
+---------+---------+---------+---------+---------+
|         |         |         |         |         |
+---------+---------+---------+---------+---------+  x
+---------+---------+---------+---------+---------+
```

(b) Suppose that a node containing PIG were inserted via the `insert()` operation. Show the updated *concrete representation* of the list.

(c) Suppose that a node were removed via the `remove()` operation. Show the updated *concrete representation* of the list. (Perform this operation upon the original list, not upon that obtained by performing (b).)
C.3. Recall the *circular chain of references* approach for representing a queue. Here is a (picture of) a concrete representation for a particular queue under this approach:

```
+-----------<-+-----------<-+-----------
  |          |          |
  v          v          v
+-----------+-----------+-----------
cow | *-+------| dog | *-+------| cat | * |
+-----------+-----------+-----------
  |          |          |
  |          |          |
  |          |          |
  rear
```

(a) List the items in the queue, from front to rear:

(b) Suppose that `dequeue()` is applied to this queue. Draw (a picture of) the updated concrete representation.

(c) Suppose that pig is placed into the queue via invocation of the `enqueue()` method. Draw (a picture of) the updated concrete representation.
Section D: Algorithm Analysis

D.1. Assume that the method reverseArray, when applied to a segment of an array of Objects, reverses the elements of that segment in linear time (i.e., in an amount of time proportional to the length of the segment).

Consider the recursive method gordon below. Let $T(n)$ be the function describing its running time; that is, for $n \geq 0$, let $T(n)$ equal the execution time of gordon when applied to an array segment of length $n$. $T$ can be characterized via a recurrence relation. In the space provided, complete the description of this recurrence relation. Do not “solve” it, just complete its description.

Note that, in both recursive calls, the array segment passed as a parameter has length one less than the length of the incoming segment.

For extra credit, describe what effect gordon has on the array segment to which it is applied.

```java
/* pre: 0 <= low <= high <= a.length
    post: has some unknown effect upon a[low..high-1]
*/
public static void gordon ( Object[] a, int low, int high ) {
    if ( low == high )
        ;
    else {
        reverseArray(a, low, high); // reverses a[low..high-1]
        gordon(a, low+1, high);
        reverseArray(a, low, high); // reverses a[low..high-1]
        gordon(a, low, high-1);
    }
}
```

$$T(n) = \begin{cases} 
\text{__________________________} & \text{if } n = 0 \\
\text{__________________________} & \text{if } n > 0 
\end{cases}$$
D.2. Solve the following recurrence relation. That is, describe $T(n)$ via an equation of the form $T(n) = E$, where there are no occurrences of $T$ in $E$. Recall that the sequence $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots$ (i.e., the sum of the reciprocals of the powers of two) approaches 2. Also recall that the value $k$ satisfying $2^k = n$ is the log to the base 2 of $n$ and is written $\log_2 n$ (You may assume, for convenience, that $n$ is a power of two.)

\[
T(n) = \begin{cases} 
  a & \text{if } n \leq 1 \\
  b + cn + T(n/2) & \text{if } n > 1 
\end{cases}
\]
D.3. In each of the following code segments, assume that the value of \( n \) is a measure of input size. For each code segment, describe (as a function of \( n \)) how many times the most deeply nested loop will iterate. (Your answer need not be exact, but it should be a good estimate.) Then express the code segment’s asymptotic running time using big-Oh notation. A code segment labeled S is assumed to run in constant time.

**Note:** In (b), you may assume that \( n \) is a power of two. Recall that \( 1+2+3+\cdots+k = \frac{k^2+k}{2} \).

<table>
<thead>
<tr>
<th>CODE SEGMENT</th>
<th>ESTIMATE OF # OF LOOP ITERATIONS</th>
<th>ASYMPTOTIC RUNNING TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) for (int i=0; i != n; i = i+1) { int j = 0; while ( j &lt; 2*i ) { S j = j+1; } }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b) int i = n; while ( i &gt; 0 ) { for (int j = 0; j != n; j = j+1) { S } i = i/2; }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c) int i = 5*n; while (i &gt; 0) { S; i = i-2; }</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Section E: OO Issues

E.1. This question concerns the following collection of Java classes, forming a hierarchy of Worker classes.

```java
public abstract class Worker {

    protected String name;

    public Worker(String name) { this.name = name; }
    public String getName() { return this.name; }
    public String toString() { return this.name; }
    public abstract void work();
}

public class Volunteer extends Worker {

    public int hoursVolunteered;
    public Volunteer(String name) { super(name); }
    public String toString() { return super.toString() + " volunteer"; }
    public void work() { ... }
}

public abstract class Employee extends Worker {

    public String toString() { return super.toString() + " employee"; }
}

public class SalariedEmployee extends Employee {

    protected float annualSalary;

    public SalariedEmployee(String name, float salary)
        { super(name); annualSalary = salary; }

    public float payThisWeek() { return annualSalary / 52.0; }
    public void work() { ... }
}

public class HourlyEmployee extends Employee {

    protected float hourlyWage;
    protected float hoursWorkedThisWeek;

    public HourlyEmployee(String name, float wage)
        { super(name); hourlyWage = wage; }

    public void setHours(float hours) { hoursWorkedThisWeek = hours; }
    public float payThisWeek() { return hourlyWage * hoursWorkedThisWeek; }
    public void work() { ... }
}
```
(a) Consider the following method appearing in a class that is a client of the classes in the Worker hierarchy:

```java
public float sumOfPay(Employee[] e) {
    float result;
    for (int i=0; i != e.length; i = i+1)
        result = result + e[i].payThisWeek();
    return result;
}
```

Explain why the statement comprising the loop body is syntactically incorrect. To correct the error, it suffices to add something to the Employee class. Show what that is.

For each of the following code segments, either indicate which statement/expression is syntactically illegal (and explain why) or show what output it produces.

(b) 
Worker w = new Worker("George");
System.out.println( w.toString() );

(c) 
Worker w = new Volunteer("Mary");
System.out.println( w.toString() );

(d) 
SalariedEmployee w = new SalariedEmployee("Beth", 52000.0);
System.out.println( w.toString() + w.payThisWeek() );

(e) 
Worker w = new SalariedEmployee("Beth", 52000.0);
System.out.println( w.toString() + w.payThisWeek() );
Section F: General Programming

F.1. Supply the body of the Java method `pairSumCntr()` (see below), which, given an integer `m` and an array `b[]` of integers whose elements are in increasing order, calculates the number of pairs of elements in `b[]` that have sum `m`. To state it precisely, the method is to compute the number of ordered pairs in the set

\[ S_{m,b} = \{(i,j) \mid 0 \leq i < j < b.length \land b[i] + b[j] = m\} \]

For example, suppose that `m = 14` and that `b[]` were as follows:

```
  0 1 2 3 4 5 6 7 8 9 10 11
  -------------------------------------------
  b | -5| -1| 0 | 2 | 5 | 8 | 9 | 14| 17| 19| 24| 31|
  -------------------------------------------
```

The correct result is 3, because \( S_{m,b} = \{(0,9),(2,7),(4,6)\} \). That is, the three pairs of elements in `b[]` that sum to `14` are (1) `b[0]` and `b[9]`, (2) `b[2]` and `b[7]`, and (3) `b[4]` and `b[6].`

**Remark:** A top-notch solution will run in linear time. A moderately good solution will run in \( O(n \lg n) \) time. A naive solution will run in \( O(n^2) \) time.

**Remark:** You may assume the existence of a boolean binary search method `binSearch()` whose behavior is such that the call `binSearch(a, low, high, p)` yields `true` if and only if the value `p` occurs in the array segment `a[low..high-1]`. This will help in devising the “moderately good” solution mentioned above.

**Remark:** If you wish to devise a recursive solution, you may assume that `pairSumCntr()` has two extra int parameters, `low` and `high`, and that its purpose is to calculate the number of pairs of elements in `b[low..high-1]` that sum to `m`.

```java
/* pre: for all j satisfying 0 <= j < b.length-1, b[j] < b[j+1]
   post: value returned is number of ways to choose two elements
       of b[] such that their sum equals m.
*/
public static int pairSumCntr( int m, int[] b ) {
```