Part I. MINI-ESSAYS. Answer the questions as completely as you can without sacrificing conciseness. The space provided should be a good indication of the length of answer required.

1. In review class, we considered an implementation of queues using back-to-back stacks. Can you think of an implementation of stacks that uses queues as “backbone”? Describe in general terms how you would do this, or, if you think it is not possible, explain why it is not. (10 pts.)

Because queues follow a FIFO behavior, it is very difficult to simulate the LIFO behavior of a stack. Whereas in the queue simulation using stacks, we were able to unstuck a stack in order to reverse the order of elements (so that now, the oldest elements are on top of the stack), there is not simple way to effect this with queues in order to have the newest element to be the “current” element. The best attempt we can think of is to requeue the $n-1$ queue elements in front of the last element so that at the end, the last element is now at the head of the queue:

```
1 2 3 4 5  
```

requeue front element 4 times

```
5 1 2 3 4  
```

This means that querying the stack/queue takes $O(n)$ time when it was only $O(1)$ for most other implementations. Then, of course, there is the issue of subsequent “push” operations. All in all, a very unnatural implementation and difficult because of the inherent mismatch between the LIFO behavior of stacks and the FIFO principle queues follow.
2. Discuss substantially the similarities and differences between stacks and queues as dispensers. Try to include priority queues in your answer specifically as a possible "unifying link" between the two dispensers. (10 pts.)

The main difference is that stacks display LIFO behavior while queues follow a FIFO principle. Otherwise, they are both linear structures and are thus implementable using list-like "backbones." Some of the implementation schemes we have seen involve the use of arrays, or vectors, or linked node structures. Both admit $O(1)$ methods for accessing the "current" element. Stacks are in a better position efficiency-wise because the access to the current element is done at the same "end" for both introducing and removing elements. Queues have access at both ends – one end for introducing new elements (the rear of the queue), the other end for removal of the current element (the front of the queue). As such, array implementations of queues may suffer an $O(n)$ performance if not done properly. But $O(1)$ is achievable with the use of circular arrays, exploiting modulo arithmetic in the management of the array index.

An intriguing concept is the use of priority queues to simulate either of the two linear structures. If we used chronology or time stamp as the "order" function that determines what the "current" element is for the priority queue, then we achieve stack performance if we equate "high" priority with "later" time stamp, and achieve queue performance if we consider "high" priority as synonymous with "earlier" time stamp. The switch from one behavior to the other can be achieved with the same order function except perhaps for the sign of the functional value (i.e., the greater the magnitude of a value, the "smaller" its negative is). So if we took time stamp as the order function — say for a particular object obj, we took a reading of the system clock to be its "priority" obj.priority() — then by simply taking $(-1)*obj.priority()$, we switch the behavior of the priority queue from being a queue to being a stack.

In this respect, priority queues are indeed a unifying link between the two dispensers.
Part II. MULTIPLE CHOICE. Select from among the given options the best answer for the given question. Record the answers in the supplied scantron sheet. (1 pt. each)

1. A(n) _________ is an object used to access each element in a container.
   a. pointer
   b. iterator
   c. interface
   d. abstract class
   e. index

2. The client controls this kind of iterator by invoking its queries and commands:
   a. external iterator
   b. internal iterator
   c. list iterator
   d. container iterator
   e. index iterator

3. This kind of iterator assumes the client’s responsibility for its own control and operation:
   a. external iterator
   b. internal iterator
   c. list iterator
   d. container iterator
   e. index iterator

4. A bridge is a design pattern that allows for loose ______ between the abstraction hierarchy and the implementation hierarchy.
   a. coupling
   b. application
   c. behavior
   d. control
   e. modifiability

5. The use of Java _______ makes it possible to collapse the two hierarchies (abstraction and implementation).
   a. abstract classes
   b. interfaces
   c. implementations
   d. specifications
   e. generics

6. This java.util interface models a rather generalized container:
   a. Comparator
   b. Iterator
   c. Observer
   d. Collection
   e. None of these answers

7. This java.util interface specifies the following methods: hasNext(), next(), and remove().
   a. Comparator
   b. Iterator
   c. Observer
   d. Container
   e. None of these answers

8. This concrete java.util class implements a growable array of objects.
   a. Comparator
   b. Vector
   c. Arrays
   d. HashSet
   e. None of these answers
9. A _____ is also known as a *key-value* table.
   a. concordance  
   b. thesaurus  
   c. dictionary  
   d. All of these answers  
   e. None of these answers

10. This is a container that allows access and removal of elements in a predetermined way.
    a. dispenser  
    b. accessor  
    c. reference  
    d. precondition  
    e. All of these answers are valid.

11. In a dispenser, this is the item that is readily accessed or removed.
    a. oldest element  
    b. current element  
    c. newest element  
    d. largest element  
    e. smallest element

12. This dispenser adheres to a LIFO principle in its operation.
    a. queue  
    b. stack  
    c. priority queue  
    d. dictionary  
    e. list

13. This dispenser adheres to a FIFO principle in its operation.
    a. queue  
    b. stack  
    c. priority queue  
    d. dictionary  
    e. list

14. In a priority queue, the “priority” between elements is determined by an object of this functional type:
    a. Object  
    b. Map  
    c. Iterator  
    d. Priority  
    e. Order

15. A _______ is defined to be a container with a “fundamental structural relationship associating the items in the container.”
    a. data type  
    b. data structure  
    c. file structure  
    d. set  
    e. list

16. This is informally defined as a finite collection of nodes one of which is distinguished and is called the root.
    a. tree  
    b. linked list  
    c. queue  
    d. stack  
    e. tuber

17. The number of subtrees rooted at a node is defined to be that node’s _______.
    a. growth rate  
    b. order  
    c. degree  
    d. depth  
    e. height
18. A(n) _____ in a tree is a node with degree equal to 0.
   a. root
   b. parent
   c. ancestor
   d. leaf
   e. edge

19. The height of the root determines a tree's _____.
   a. height
   b. width
   c. size
   d. degree
   e. None of these is a valid answer.

20. A complete binary tree of height $h$ has this many nodes. [Hint: Try it for a tree with just one node, i.e., $h=0$.]
   a. $2^{h+1} - 1$
   b. $2^h$
   c. $2^{h+1} + 1$
   d. $2^{h-1} + 1$
   e. None of these answers is valid

21. This kind of traversal (or iteration) of a binary tree visits the root first, then traverses the left subtree, and then traverses the right subtree.
   a. complete
   b. postorder
   c. breadth-first
   d. inorder
   e. preorder

22. This kind of traversal (or iteration) of a binary tree, that is also a binary search tree, visits the elements in the tree in sorted order.
   a. preorder
   b. inorder
   c. postorder
   d. All of these are valid answers
   e. None of these is a valid answer.

23. One advantage to the array-based implementation of binary trees is that:
   a. they utilize all of the allocated space
   b. they use index-based referencing
   c. they achieve sublinear performance
   d. they accommodate “ghost” nodes
   e. All of these are valid answers.

24. ________ binary tree implementations can be constructed in a similar manner to linked list implementations, i.e., with the use of “private” nodes.
   a. List-based
   b. Linked
   c. Array-based
   d. Vector-based
   e. All flavors of

25. General trees can be implemented using binary trees as “backbone” of the implementation. Assess the truth of this statement.
   a. It is a true statement
   b. It is a false statement.
26. The inner class that encapsulates node functionality in linked binary tree implementations is ______.
   a. protected class BTNode
   b. private class BTNode
   c. public class BTNode
   d. class BTNode
   e. interface BTNode

27. The interface that specifies the functionality of a binary tree implementation is:
   a. BinaryTreeImplementation
   b. ListImplementation
   c. SetImplementation
   d. QueueImplementation
   e. StackImplementation

28. A binary tree can be defined recursively as either an empty tree or a structure consisting of these components, except for:
   a. the root element
   b. the left (binary) subtree
   c. the right (binary) subtree
   d. a leaf element
   e. None of the above (i.e., all are necessary components)

29. This sometimes unavoidable condition creates problems of semantics in certain tree operations like replacing the left subtree with a new one.
   a. malformed expressions
   b. structural sharing
   c. syntactical errors
   d. bad parameters
   e. NullReferenceException

30. The next chapter in the supplementary materials (Ch. 27) is about binary search trees and heaps. Assess the truth of this claim.
   a. It is a true statement.
   b. It is a false statement.

---

Study the static class diagrams below – they will be helpful in answering some of the questions on the following pages:
Part III. CODING. Supply the answers that are requested based on the description of the programming situation provided.

1. To provide a means of representing a set of integers, supply the formal definition of class `IntegerSet` which extends class `Set` (just the first lines of code is needed here, not the complete code for the class). Base your answer on the static diagram on the previous page. (5 pts.)

```java
public class IntegerSet extends Set {
    public IntegerSet() {
        super();
    }
}
```

2. To provide a means of representing a set of odd integers, we have declared the following variable:

   ```java
   IntegerSet oddNumbers = new IntegerSet();
   ```

   We want to include in this set (`oddNumbers`), the first 50 odd positive integers. Class `Integer` provides a constructor that accepts an `int` value and wraps it inside an `Integer` object:

   ```java
   public Integer( int number )
   ```

   Create a “wrapped” object for the given number

   Supply the segment of code that deposits the first 50 odd positive integers into the variable `oddNumbers`: (5 pts.)

   ```java
   IntegerSet oddNumber = new IntegerSet();

   for ( int i=1; i<=50; i=i+1 )
       oddNumber.add( new Integer( 2*i-1 ) );
   ```
3. We have decided that the performance of our vector-based implementation (the default implementation) of the set of odd integers is just too slow for our needs. We wish to convert our set (which now has VectorSetImplementation as the type of its implementation component) to one which has a HashSetImplementation as the type of its implementation component.

The following implementation detail from the abstract class Set will be helpful:

```java
public abstract class Set implements Cloneable {

// Codes for various implementations
public static final int VECTOR_BASED = 1;
public static final int HASH_BASED = 2;

/**
 * Create a Set with the specified implementation.
 * require:
 * implementation == Set.VECTOR_BASED || implementation == Set.HASH_BASED
 */
protected Set( int implementation ) {
    assert ( implementation == VECTOR_BASED || implementation == HASH_BASED );
    if ( implementation == VECTOR_BASED ) {
        imp = new VectorSetImplementation();
    } else if ( implementation == DYNAMIC ) {
        imp = new HashSetImplementation();
    }
}

/**
 * Create a Set with the default implementation.
 * Default implementation is VECTOR_BASED.
 * ensure:
 * this.isEmpty()
 */
protected Set() {
    this( VECTOR_BASED );
}

/**
 * Create a Set with the specified implementation.
 * ensure:
 * this.isEmpty()
 */
protected Set( SetImplementation implementation ) {
    assert ( implementation != null );
    imp = implementation;
}

private SetImplementation imp;
} //end of class Set
```

(continued next page)
Supply the segment of code that will convert oddNumbers into a hash-based set: (10 pts.)

```java
IntegerSet hashBasedSet =
    new IntegerSet( HASH_BASED );

Iterator iter = oddNumbers.iterator();

while ( iter.hasNext() )
    hashBasedSet.add( iter.next() );

oddNumbers = hashBasedSet;
```

Part IV. MORE CODING. Supply the answers that are requested.

1. Consider the specification of the class Stack. We give part of this below:

```java
/**
   * A dispenser with LIFO behavior
   */
public abstract class Stack
    implements Cloneable {

    /**
     * Create an empty Stack with
     * default implementation.
     */
    protected Stack() {
    }

   protected Stack( StackImplementation imp ) {
       //instance variables
       private StackImplementation imp;
   } //end Stack

    //instance variables
    private StackImplementation imp;
}
```

The stack implementation will be list-based and will have the following different flavors: bounded (using a BoundedList), dynamic (using a DynamicList), and linked (using a LinkedList).
Below we illustrate the design of this model:

In implementing the class BoundedStackImplementation, we choose to have a BoundedList as a component to represent our stack:

```java
public class BoundedStackImplementation
    implements StackImplementation {
    private BoundedList stackList;  //stack represented as list
} //end BoundedStackImplementation
```

Your task is to supply the implementation of the method `push()` in this class. Recall that the "bottom" of the stack is identified with the list location with index 0 with the stack's top element being the "active" list location with the highest index (see illustration below):

In the space provided on the next page, complete the method `push()` by supplying the correct sequence of statements that would effect the introduction of a new element into the stack. Remember that in this implementation, the list representing the stack is a component of the class.
(a) Supply the implementation of the method `push()` in the space below: (10 pts.)

```java
/**
* Add the specified element to this Stack
*/
public void push( Object obj ) {

    assert ( obj != null );
    assert ( !this.isFull() );

    this.stackList.append( obj );
}

//end of push
```

(b) Assess the method complexity of your implementation as a function of the size of the stack at time of the “push”. No explanation of your answer is required: (5 pts.)

> The method complexity is obviously $\mathcal{O}(1)$ since `append()` for lists has that complexity and there are no other method invocations involved in the implementation of `push()`.

In implementing the `LinkedStackImplementation` class, we choose instead to extend class `LinkedList`:

```java
public class LinkedStackImplementation extends LinkedList
    implements StackImplementation {

    //end LinkedStackImplementation
```

Thus, the stack is a linked list. Your task is to supply the implementation of the method `pop()` in this class. How to use a list to represent a stack does not change for this class (see illustration on previous page). Supply your implementation in the space on the next page:
(c) Supply the implementation of the method `pop()` in the space below: (10 pts.)

```java
/**
 * Remove the element of this Stack that was most 
 * recently added
 */
public void pop() {
    assert (!this.isEmpty());
    super.remove( super.size() - 1 );
}
```

(d) Assess the method complexity of the `pop()` command above. Briefly explain your answer. (5 pts.)

The method complexity is not that obvious, but it will be $O(n)$ since `remove()` for linked-lists, especially for the last element in the linked node structure, involves a sequential search traversing the links connecting the nodes. This is done in a loop process in the implementation of `remove()`.

If we had followed the textbook strategy of representing stacks with lists such that the top of the stack is actually located at the list position with index 0, then the `remove()` would only require $O(1)$ work to effect.
BONUS: Provide the implementation of the query `isEmpty()` which determines whether the stack is empty or not for either of the two stack implementations given earlier: (5 pts.)

```java
public boolean isEmpty() {
    return (this.size() == 0);
}
```

Write your answer here.