

**COSC 2006 FINAL EXAM  
DATA STRUCTURES I**

**Saturday, December 13, 2008, 9:00 am**

*Time Allowed: 3 hours*

*Instructor: Barry G. Adams*

**Name (PLEASE PRINT)** \_\_\_\_\_

**Student #** \_\_\_\_\_

1. *Answer ALL questions. Write your answers on this questionnaire.*
  2. *Use back of exam pages if necessary.*
  3. *Do not write comments in your programs.*
  4. *No aids permitted.*
  5. *Number of Questions: 8*
  6. *Total Marks: 70*
-

**Question 1 (3 + 3 + 3 + 3 = 12 marks)**

Suppose that a dynamic array implementation of a generic `ArrayBag<E>` or `ArraySet<E>` has instance data fields

```
private E[] data; // array of references to the elements
private int manyItems; // number of elements in bag
```

and a `reallocate` method with prototype `public void reallocate()` that doubles the size of the array.

(a) For `ArrayBag<E>` write a `clone` method with prototype

```
public ArrayBag<E> clone()
```

that returns a clone (copy) of this bag.

**Answer:**

(b) For `ArraySet<E>` write the `contains` method with prototype

```
public boolean contains(E element)
```

that returns true if the given `element` is in this set and false otherwise.

**Answer:**

(c) For `ArrayBag<E>` write the `remove` method with prototype

```
public boolean remove(E target)
```

which removes one copy of the given `target` element from this bag and returns true if the `target` element was found. Assume that `null` data elements are **not** allowed in the bag.

**Answer:**

(d) For `ArraySet<E>` write the `add` method with prototype

```
public void add(E element)
```

that adds the given `element` to this set only if it is not already in the set.

**Answer:**

**Question 2 (3 + 3 + 3 = 9 marks)**

Given the `Node<E>` class on page 15 with data elements of type `E`, write the following methods for this class

- (a) the `addNodeAfter` method.

**Answer:**

- (b) a static method to return the length of a given list with prototype

```
public static <E> int listLength(Node<E> list)
```

where `list` is a reference to the head of the list.

**Answer:**

- (c) a static method to make a copy of a given list with prototype

```
public static <E> Node<E> listCopy(Node<E> source)
```

where `source` is a reference to the head of the list to be copied and the return value is a reference to the copy. [**Hint:** Make a one-element copy of the head of `source`, then traverse the remaining nodes, if any, and copy them using `addNodeAfter`.]

**Answer:**

**Question 3 (3 + 2 + 3 = 8 marks)**

The `LinkedBag<E>` class uses the `Node<E>` class on page 15 and has the data fields

```
private Node<E> head; // reference to head node of list representing this bag
private int manyNodes; // number of elements in this bag
```

(a) Write the `toString` method with prototype

```
public String toString()
```

that returns a string of the form `[e1,e2,...,en]`

**Answer:**

(b) Write the `add` method with prototype

```
public void add(E element)
```

that adds the given `element` to this bag.

**Answer:**

(c) Write the `countOccurrences` method with prototype

```
public int countOccurrences(E target)
```

that returns the number of times the given `target` element appears in this bag. Assume that null data elements are **not** allowed in the bag.

**Answer:**

**Question 4 (5 marks)**

Assume that the `LinkedBag<E>` class from Question 3 has the following method which returns an iterator for the list.

```
public LinkedList<E> iterator()
{
    return new LinkedList<E>(head);
}
```

Write the `LinkedList<E>` class that implements the `Iterator<E>` interface

```
public Interface Iterator<E>
{
    public boolean hasNext();
    public E next();
    public void remove(); // not implemented
}
```

**Answer:**

**Question 5 (2 + 2 + 2 + 2 + 2 + 2 = 12 marks)**

Write the following recursive methods using the List<E> game on page 16.

(a)

```
/**
 * append two lists
 * @param <E> the type of the data elements in the list
 * @param list1 first list
 * @param list2 second list
 * @return new list obtained by appending list2 to end of list1
 */
public static <E> List<E> append(List<E> list1, List<E> list2)
```

**Answer:**

(b)

```
/**
 * Find the number of occurrences of a given data element in a list
 * @param <E> the type of the data elements in the list
 * @param list the list
 * @param data the data element to find multiplicity of
 * @return the number of times data element occurs in the list
 */
public static <E> int multiplicity(List<E> list, E data)
```

**Answer:**

(c) 

```
/**
 * Remove first occurrence of a given data element from a list
 * @param <E> the type of the data elements in the list
 * @param data the data element to remove from list
 * @param list the list to remove data element from
 * @return a new list with data element removed
 */
public static <E> List<E> remove(E data, List<E> list)
```

**Answer:**

(d) 

```
/**
 * Test if two lists are identical (same data elements, same order)
 * @param <E> the type of the data elements in the list
 * @param list1 the first list
 * @param list2 the second list
 * @return true if lists are equal else false
 */
public static <E> boolean equal(List<E> list1, List<E> list2)
```

**Answer:**

(e)

```
/**
 * Remove all occurrences of a given data element from a list
 * @param <E> the type of the data elements in the list
 * @param data the data element to remove from list
 * @param list the list to remove data element from
 * @return a new list with all occurrences of data element removed
 */
public static <E> List<E> removeAll(E data, List<E> list)
```

**Answer:**

(f)

```
/**
 * Replace first occurrence of a given element from a list
 * @param <E> the type of the data elements in the list
 * @param oldData the data element to replace
 * @param newData the replacement data element
 * @param list the list to replace data element in
 * @return a new list with oldData replaced by newData
 */
public static <E> List<E> replace(E oldData, E newData, List<E> list)
```

**Answer:**

**Question 6 (10 marks)**

Given the `Stack<E>` interface on page 17 and the `DynamicArrayStack<E>` class summary on page 20 write the constructor and the `push`, `pop`, `peek` and `reallocate` methods for `DynamicArrayStack<E>` class.

**Answer: If you need more space use next page:**

**Answer: Question 6 continued:**

**Question 7 (10 marks)**

Given the `Queue<E>` interface on page 18 and the `LinkedListQueue<E>` class summary on page 20 write the constructor, and the `enqueue`, `dequeue`, and `front` methods for the `LinkedListQueue<E>` class.

**Answer: If you need more space use next page:**

**Answer: Question 7 continued:**

**Question 8 (2 + 2 = 4 marks)**

Given the `Deque<E>` interface on page 19 and the `LinkedDeque<E>` class summary on page 21 that uses the internal doubly linked node class `DLNode<T>` and sentinel nodes at the front and rear,

- (a) write the `insertFront` method,

**Answer:**

- (b) write the `insertRear` method.

**Answer:**

**The Node<E> class**

```
public class Node<E>
{
    private E data;
    private Node<E> link;

    public Node(E data, Node<E> link) {...}
    public E getData() {...}
    public Node<E> getLink() {...}
    public void setData(E newData) {...}
    public void setLink(Node<E> newLink) {...}
    public void addNodeAfter(E element) {...}
    public void removeNodeAfter() {...}

    // static utility methods go here
}
```

**The List<E> class for the list game**

```
/**
 * The five rules for the list game.
 * NOTE: The empty list is denoted by null
 */
public class List<E>
{
    // data fields go here

    /**
     * Return the head of a list.
     * @param <E> type of the list data
     * @param list the list to operate on
     * @return the head of the list
     */
    public static <E> E head(List<E> list) {...}

    /**
     * Return the tail of a list
     * @param <E> the type of the list data
     * @param list the list to operate on
     * @return the tail of the list
     * @throws IllegalArgumentException
     *         if the list is empty
     */
    public static <E> List<E> tail(List<E> list) {...}

    /**
     * Test for an empty list
     * @param <E> the type of the list data
     * @param list the list to operate on
     * @return true if list is empty else false
     */
    public static <E> boolean isEmpty(List<E> list) {...}

    /**
     * Construct a new list from a head and a tail
     * @param <E> the type of the list data
     * @param head the head of the new list
     * @param tail the tail of the new list
     * @return the new list
     */
    public static <E> List<E> cons(E head, List<E> tail) {...}

    /**
     * Return a string representation of this list.
     * @return list of form List[d1,d2,...]
     */
    public String toString() {...}
}
```

**The Stack<E> Interface**

```
public interface Stack<E>
{
    /**
     * Push an element onto the stack.
     * @param e the element to push.
     * @postcondition e is the new element at top.
     */
    public void push(E e);

    /**
     * Pop an object from stack and return it.
     * @precondition stack is not empty.
     * @postcondition top element of stack is returned and removed.
     * @throws EmptyStackException if stack is empty
     */
    public E pop() throws EmptyStackException;

    /**
     * Return top element without popping it.
     * @precondition stack is not empty.
     * @postcondition stack unchanged, top element returned.
     * @throws EmptyStackException if stack is empty
     */
    public E peek() throws EmptyStackException;

    /**
     * Remove all elements from the stack.
     * @postcondition the stack is empty
     */
    public void clear();

    /**
     * Return true if stack is empty else false.
     * @postcondition value returned is true if stack is
     * empty else false.
     */
    public boolean isEmpty();

    /**
     * Return the number of elements on stack.
     * @postcondition number of elements is returned.
     */
    public int size();

    /**
     * Return a string representation of a stack.
     * The format is [a,b,c,...] where a is the top of stack.
     * @return the string representation
     */
    public String toString();
}
```

**The Queue<E> Interface**

```
public interface Queue<E>
{
    /**
     * Insert a new element at the rear (end) of the queue.
     * @param e the element to insert
     * @postcondition the element e has be inserted at the
     * rear of the queue and the size has been incremented
     */
    public void enqueue(E e);

    /**
     * Remove and return the element at the front of the queue.
     * @precondition the queue is not empty
     * @postcondition the element at the front of the queue was
     * removed and returned and the size has been decremented
     * @throws EmptyQueueException if the queue is empty
     */
    public E dequeue() throws EmptyQueueException;

    /**
     * Return the element at the front of the queue.
     * @precondition the queue is not empty
     * @postcondition the element at front of the queue was
     * returned and the queue is not changed
     * @throws EmptyQueueException if the queue is empty
     */
    public E front() throws EmptyQueueException;

    /**
     * Remove all elements from the queue.
     * @postcondition the queue is empty
     */
    public void clear();

    /**
     * Test if the queue is empty
     * @postcondition true was returned if the queue was empty
     * and false was returned otherwise.
     */
    public boolean isEmpty();

    /**
     * Return the number of elements in the queue.
     * @postcondition the number of elements in the queue
     * was returned.
     */
    public int size();

    /**
     * Return a string representation of a queue.
     * The format is [a,b,c,...] where a is the front
     * of the queue.
     * @return the string representation
     */
    public String toString();
}
```

## The Deque<E> Interface

```
public interface Deque<E>
{ /** Insert a new element at the front of the deque.
  * @param e the element to insert
  * @postcondition the element e is inserted at
  * the front of the deque and size is incremented
  */
  public void insertFront(E e);

  /** Insert a new element at the rear of the deque.
  * @param e the element to insert
  * @postcondition the element e is inserted at
  * the rear of the deque and size is incremented.
  */
  public void insertRear(E e);

  /** Remove and return the element at the front of the deque.
  * @precondition the deque is not empty
  * @postcondition the element at the front of the deque is
  * removed and returned and the size is decremented.
  * @throws EmptyDequeException if deque is empty
  */
  public E removeFront() throws EmptyDequeException;

  /** Remove and return the element at the rear of the deque.
  * @precondition the deque is not empty
  * @postcondition the element at the rear of the deque is
  * removed and returned and the size is decremented.
  * @throws EmptyDequeException if deque is empty
  */
  public E removeRear() throws EmptyDequeException;

  /** Return the element at the front of the deque.
  * @precondition the deque is not empty
  * @postcondition the front element is returned and the deque is unchanged.
  * @throws EmptyDequeException if deque is empty
  */
  public E front() throws EmptyDequeException;

  /** Return the element at the rear of the deque.
  * @precondition the deque is not empty
  * @postcondition the rear element is returned and the deque is unchanged.
  * @throws EmptyDequeException if deque is empty
  */
  public E rear() throws EmptyDequeException;
  /** Remove all elements from the deque.
  * @postcondition the deque is empty
  */
  public void clear();
  /** Test if the deque is empty.
  * @postcondition true is returned if the deque is empty
  * and false is returned otherwise.
  */
  public boolean isEmpty();
  /** Return the number of elements in the deque.
  * @postcondition the number of elements in the deque is returned.
  */
  public int size();
  /** Return a string representation of a deque.
  * The format is [a,b,c,...] where a is the front of the deque.
  * @return the string representation
  */
  public String toString();
}
```

**The DynamicArrayStack<E> class summary**

```
public class DynamicArrayStack<E> implements Stack<E>
{
    private E[] data; // array for stack elements
    private int top; // index of top element or -1

    public DynamicArrayStack(int initialCapacity) {...}

    public void push(E e) {...}
    public E pop() throws EmptyStackException {...}
    public E peek() throws EmptyStackException {...}

    public void clear() {...}
    public boolean isEmpty() {...}
    public int size() {...}

    // format is [a,b,c,...] where a is top of stack
    public String toString() {...}

    // make a new array twice as big as current one
    // and copy data array to it
    private void reallocate() {...}
}
```

**The LinkedQueue<E> class summary**

```
public class LinkedQueue<E> implements Queue<E>
{
    private Node<E> head; // head of the queue
    private Node<E> tail; // tail of the queue
    private int size; // number of elements in queue

    public LinkedQueue() {...}

    public void enqueue(E e) {...}
    public E dequeue() throws EmptyQueueException {...}
    public E front() throws EmptyQueueException {...}

    public void clear() {...}
    public boolean isEmpty() {...}
    public int size() {...}
    public String toString() {...}

    // Inner class for the nodes

    private class Node<T>
    {
        private T data;
        private Node<T> next;

        public Node(T data, Node<T> next)
        {
            this.data = data;
            this.next = next;
        }
    }
}
```

**The LinkedDeque<E> class summary**

```
public class LinkedDeque<E> implements Deque<E>
{
    // A doubly linked structure is used to hold the data.
    // Sentinel nodes are used at each end of the structure to
    // simplify the operations. An empty list has a header and
    // trailer that reference each other.

    private DLNode<E> header; // sentinel node for head
    private DLNode<E> trailer; // sentinel node for tail
    private int size; // number of elements in deque

    public LinkedDeque()
    {
        clear();
    }

    public void insertFront(E e) {...}
    public void insertRear(E e) {...}

    public E removeFront() throws EmptyDequeException {...}
    public E removeRear() throws EmptyDequeException {...}

    public E front() throws EmptyDequeException {...}
    public E rear() throws EmptyDequeException {...}

    public void clear() {...}
    public boolean isEmpty() {...}
    public int size() {...}
    public String toString() {...}

    // Inner class for doubly linked nodes

    public class DLNode<T>
    {
        private T data;
        private DLNode<T> prev; // previous
        private DLNode<T> next;

        public DLNode()
        {
            this(null, null, null);
        }

        public DLNode(T data, DLNode<T> prev, DLNode<T> next)
        {
            this.data = data;
            this.prev = prev;
            this.next = next;
        }
    }
}
```