# ITI 1121. Introduction to Computer Science II

Laboratory 10

Winter 2016

## Objectives

- Further understanding of iterators
- Further understanding of (doubly) linked lists

## Part I
# Iterator

For the first part of the laboratory there is a class to store an unlimited number of bits: zeros and ones. This class, called **BitList**, is a linked list to store the bits. Unlike the previous list implementations that we looked at, the values inside the nodes will be **int**s.

- Iterator.java
- BitList.java
- BitListTest.java

## 1 BitList

Because most operations on bits work from right to left, your implementation must store the bits in the "right-to-left" order — i.e. with the rightmost bit in the first node of the list.

For example, the bits "11010" must be stored in a list such that the bit 0 is the first, followed by 1, followed by 0, followed by, 1, followed by 1:

```
-> 0 -> 1 -> 0 -> 1 -> 1
```

Complete the implementation of the class **BitList**.

1. **public BitList( String s );** creates a list of bits representing the input string **s**.

   The given string, **s**, must be a string of 0s and 1s, otherwise the constructor must throw an exception of type **IllegalArgumentException**.

   This constructor initializes the new **BitList** instance to represent the value in the string. Each character in the string represents one bit of the list, with the rightmost character in the string being the low order bit.

   For example, given the string "1010111" the constructor should initialize the new **BitList** to include this list of bits:

   ```
   -> 1 -> 1 -> 1 -> 0 -> 1 -> 0 -> 1
   ```

If given the empty string, the constructor should create an empty list — **null** is a not a valid argument.

The constructor **must not** remove the leading zeroes. For example, given "0001" the constructor must create a list of size 4:

```
-> 1 -> 0 -> 0 -> 0
```

# 2 Iterative

Create a class called Iterative. Implement the methods below. Your solutions must be iterative (i.e. uses iterators).

## 2.1 static BitList complement( BitList in )

Write a static iterative method that returns a new BitList that is the complement of its input. The complement of 0 is 1, and vice-versa. The complement of a list of bits is a new list such that each bit is the complement of the bit at the same position in the original list. Here are 4 examples.

```
1011
0100

0
1

01
10

0000111
1111000
```

## 2.2 static BitList or( BitList a, BitList b )

Write a static iterative method that returns the **or** of two **BitLists**. This is a list of the same length as **a** and **b** such that each bit is the **or** of the bits at the equivalent position in the lists **a** and **b**. It throws an exception, **IllegalArgumentException**, if either list is empty or the lists are of different lengths.

```
a = 10001
b = 00011
a or b = 10011
```

# Part II
# Iterator

## 3 CircularQueue

This question is about circular queue and iterator. The class **CircularQueue** uses the implementation technique called "circular array" to implement a fixed-size queue. This class also provides an implementation of an iterator.

```
public interface Iterator<E> {
    // Returns the next element in the iteration.
    public abstract E next();

    // Returns true if the iteration has more elements.
    public abstract boolean hasNext();
}
```

- This implementation uses a **fixed-size circular array**;
- An object of the class **CircularQueue** has a method **iterator** that returns an object of the class **CircularQueueIterator**. This class implements the interface **Iterator**;
- A call to the method **hasNext** of an iterator object returns **true** if there are more elements (cyclically) in the queue, and **false** otherwise;
- A call to the method **next** of an iterator object returns the next element in the queue (cyclically). Specifically, the first call to **next** returns the front element, the second call returns the element immediately after the first element, etc. The value of the "current" index of the iterator must wrap around the fixed-size array in the class **CircularQueue**. Eventually, a call to the method **next** will return the rear element of the queue. At this point, a call to the method **hasNext** returns **false** and a call to the method **next** will cause **NoSuchElementException** to be thrown.

Complete the implementation of the class **CircularQueue**.

- Queue.java
- CircularQueue.java

The template below provides hints for your implementation.

```
public class CircularQueue<E> implements Queue<E> {

    private static final int DEFAULT_CAPACITY = 100;
    private int front, rear, size;
    private E[] elems;

    public CircularQueue( int capacity ) {
            elems = (E[]) new Object[ capacity ];
            front = 0;
            rear = -1;
            size = 0;
    }

    public boolean isEmpty() {
            return ( size == 0 );
    }

    public void enqueue( E value ) {
            rear = ( rear+1 ) % elems.length;
            elems[ rear ] = value;
            size = Math.min( size + 1, elems.length ) ;
```

```
    }

    public E dequeue() {
        E savedValue = elems[ front ];
        elems[ front ] = null;
        size--;
        front = ( front+1 ) % elems.length;
        return savedValue;
    }

    private ___________ CircularQueueIterator implements Iterator<E> {

        private ___________ current = ___________;

        public E next() {

            if ( ____________________________________ ) {
                throw new NoSuchElementException();
            }









            return ____________________________;
        }

        public boolean hasNext() {
            boolean result;

            result = ____________________________________________________;

            return result;
        }
    } // End of CircularQueueIterator


    public ___________________ iterator() {

        return ____________________________________;
    }

} // End of CircularQueue
```

# Part III
# LinkedList

## 4 remove(int from, int to)

Implement the method **remove( int from, int to )** for the class **LinkedList**. This instance method removes all the elements in the specified range from this list and returns a new list that contains all the removed elements, in their original order. The implementation of **LinkedList** has the following characteristics:

- An instance always starts off with a dummy node, which serves as a marker for the start of the list. The dummy node is never used to store data. The empty list consists of the dummy node only;
- In the implementation for this question, the nodes of the list are doubly linked;

- In this implementation, the list is circular, i.e. the reference **next** of the last node of the list is pointing at the dummy node, the reference **previous** of the dummy node is pointing at the last element of the list. In the empty list, the dummy node is the first and last node of the list, its references **previous** and **next** are pointing at the node itself;
- Since the last node is easily accessed, because it is always the previous node of the dummy node, the header of the list does not have (need) a tail pointer.

**Example**: if **xs** is a reference designating a list containing the following elements **[a,b,c,d,e,f]**, after the method call **ys = xs.remove(2,3)**, the list designated by **xs** contains **[a,b,e,f]**, and **ys** designates a list containing **[c,d]**.

**You cannot use the methods of the class LinkedList. In particular, you cannot use the methods add() or remove().**

- LinkedList.java

**Hint:** draw detailed memory diagrams.

# Part IV
# Quiz

## 5 Question

For this question, we use the list iterator of Java. This iterator has a method **add**. Hence, the iterator can add elements to a list.

- http://download.oracle.com/javase/6/docs/api/java/util/ListIterator.html

```
import java.util.ListIterator;
import java.util.LinkedList;

public class Test {

    public static void main( String[] args ) {

        LinkedList<String> a, b;

        a = new LinkedList<String>();
        b = new LinkedList<String>();

        a.add( "alpha" );
        a.add( "bravo" );
        a.add( "tango" );
        a.add( "charlie" );

        ListIterator<String> i, j;

        i = a.listIterator();
        j = b.listIterator();

        while ( i.hasNext() ) {
            j.add( i.next() );
        }
```

```
        System.out.println( a );
        System.out.println( b );

    }
}
```

What is the result of the execution of the above Java program?

**A.** charlie, tango, bravo, alpha
charlie, tango, bravo, alpha

**B.** alpha, bravo, tango, charlie
alpha, bravo, tango, charlie

**C.** charlie, tango, bravo, alpha
alpha, bravo, tango, charlie

**D.** alpha, bravo, tango, charlie
charlie, tango, bravo, alpha

**E.** Runs into an infinite loop

- Write your answer to the above question directly in the **Submission** text field of the submission Web page:
- https://uottawa.blackboard.com/.

**Last Modified: March 24, 2016**