ITI 1121. Introduction to Computer Science II

Laboratory 10

Summer 2016

Objectives:

- Part A: Solving two questions.
- Part B: Further understanding of (doubly) linked lists and interfaces.

Part A:

In this part the TA will solve the following questions. The students need not hand anything for this part.

Question 1

The class **DynamicArrayStack** below increases or decreases its physical size according to the needs of the application.

• **DynamicArrayStack** uses an array to store the elements of this stack;

• The interface **Stack** and its implementation, **DynamicArrayStack**, have a formal parameter type (in other words, the implementation uses the concept of generics types, introduced in Java 1.5);

• The initial capacity of this array is given by the first parameter of the constructor;

• The physical size of the array is increased by a fixed amount (increment) when the method **void push(E elem)** is called and the array is full;

• The physical size of the array is decreased by a fixed amount (**increment**) during a call to the method **E pop()** if the number of free cells becomes **increment** or more;
• The **increment** is given by the second parameter of the constructor;

• The instance variable **top** designates the **top** element (i.e. the cell where the last element was inserted, or **-1** if the stack is empty).

A. Correct at least 5 mistakes (compile-time or runtime errors) in the partial implementation (see next two pages) by marking the error with a circle and writing down the correction.(10 marks)

B. Complete the partial implementation of the class DynamicArrayStack given the above information. (10 marks)

```
    // Instance variables
    private static E[] elems;              // Stores the elements of this stack
    private static int top = -1;           // Designates the top element
    private final int capacity;            // Memorizes the initial capacity
    private final int increment;           // Used to increase/decrease the size

    public DynamicArrayStack( int capacity, int increment ) {

        E[] elems =         new Object[ capacity ];
        this.capacity = capacity;
        this.increment = increment;
    }

    // Returns true if this stack is empty;
    public boolean isEmpty() {
        return top == 0;
    }


    public void push( E element ) {
        if ( ______________________________ ) {
        increaseSize();
        }

        elems[ top ] = element;
        top++;
    }

    private void increaseSize() {

        E[] newElems;
        int newSize;

        newSize= elems.length + increment;

        newElems = ______________________________;

        for ( int i=0; i<elems.length; i++ ) {
            newElems[ i ] = elems[ i ];
        }
        _________________________________;
    }
```

// Continue to next page ...

DynamicArrayStack (continued)

```
	public E peek() {
		return ______________________________;
	}



	// Complete the implementation of pop()
	public E pop() {

		E saved;
		saved = elems[ top ];

		elems[ top ] = _______________;
		top--;
		return saved;
	}



	private void decreaseSize() {
		E[] newElems;
		int newSize;
		newSize = elems.length - increment;

		if ( newSize < capacity ) {
			newSize = capacity;
		}
		________________________________________;

		for ( int i=0; i<=top; i++ ) {
			newElems[ i ] = elems[ i ];
		}

		elems = newElems;
	}
} // End of DynamicArrayStack
```

C. Complete the implementation of the method main.
It declares a stack of Integer objects, creates a new stack of Integer objects, pushes 20 elements onto the stack, removes and prints those elements(10 marks).

```
public class Test {

    public static void main( String[] args ) {


        // Declare a reference to a stack of Integer objects

        _____________________ s;


        // Create an instance of DynamicStack to store Integer objects

        s = ________________________________________;


        for ( int i=0; i<20; i++ ) {

            s.push( new Integer( i ) );
        }

        while ( ! s.isEmpty() ) {

            // Declares an Integer

            ________________ elem;


            // Removes an element from the stack

            elem = ________________;

            System.out.println( elem );
        }
    }
} // End of Test
```

Question 2

What will be printed on the screen if the method main of the class Lis, presented in the following pages, is executed? Read carefully all the methods before answering!

```
public class Lis {
        private Object[] obj;
        private int n = 0;

        public Lis (int capInit) {
                obj= new Object[capInit];
        }
        public int dim() {
                return n;
        }
        public void Add (Object ob) {
                if (n >= obj.length) {
                        Object[ ] tmp= obj;
                        obj= new Object[tmp.length * 3 / 2];
                        for (int i=0; i<n; i++) {
                                obj[i]= tmp[i];
                        }
                }
                obj[n++]= ob;
        }
        public void Remove() {
                if (n == 0)
                        return;
                obj[--n] = null;
                if (n < obj.length/2) {
                        Object[] tmp = obj;
                        obj = new Object[1+tmp.length/2];
                        for (int i=0; i<n; i++) {
                                obj[i]= tmp[i];
                        }
                }
        }//continue to next page
public void Display () {
        for (int i=0; i<obj.length; i++) {
                if (obj[i] == null)
                        System.out.println ("[" + i + "] = null");
                else
                        System.out.println ("[" + i + "] = " + obj[i]);
        }
}

        public static void main (String[] arg) {
                Lis test = new Lis (4);
                test.Add ("paul");
                test.Add ("eve");
                test.Add ("sam");
                System.out.println ("a)");
                test.Display ();
                test.Add ("tim");
                test.Add ("amanda");
                System.out.println ("b)");
                test.Display ();
                test.Remove ();
                test.Remove ();
                test.Remove ();
                System.out.println ("c)");
                test.Display ();
        }
}
```

Part B:

1 Objectives

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

2 Introduction

In this laboratory, you will create a (doubly) linked list implementation of the interface OrderedStructure, which declares the following methods.

- **int size();**
- **boolean add(Comparable obj);**
- **Object get(int pos);**
- **void remove(int pos).**

Implementations of this interface should be such that the elements are kept in increasing order. The order of the elements is defined by the implementation of the method compareTo(Object obj) declared by the interface Comparable. The classes Integer and String both implement the interface Comparable, you can use these for the tests. Objects of any other classes that implement the interface Comparable can be stored in an OrderedList.

IMPORTANT: Since the main objective of the laboratory is to study doubly linked lists, generics will not be used. Since generics are not used, 1) some warnings will be issued when compiling the source code. But also, 2) the type of the return value of the method get(int pos) will be Object. Hence, the users of the class will be forced to use a type cast when accessing the content of an OrderedList. There is an optional exercise at the end of the laboratory consisting of rewriting the implementation using generics. A solution will be posted next week.

File

- OrderedStructure.java
- Documentation

3 OrderedList

1. Create an implementation for the interface OrderedStructure. This implementation, called OrderedList, will use doubly-linked nodes and should have a head and a tail pointer.

Furthermore, the implementation should throw exceptions where this applies. See OrderedStructure.

1. Here is a step-by-step approach for implementing the class OrderedList. This will be a top-down approach, focusing on the general organization of the class first (adding the variables, the nested class, as well as empty methods). Once, the overall implementation is complete, we will implement the methods one by one.

   First, let's create a template for the class. At this point, we are ignoring details such as the implementation of the body of the methods, we are focusing on the necessary variables and the need for a static class called Node. This template needs to contain a static nested class called Node, the necessary instance variables, and empty definitions for all the methods of the interface OrderedStructure.

   The compiler will not allow us to have empty declarations for the methods that return a result. To circumvent this problem, we could add a "dummy" return statement, such as returning the value -99 for the method size. Knowing that later we will give it a proper implementation.

   However, we may later forget to change the implementation, and this will cause all sorts of problems. Because of this, I prefer creating an initial method that consists a throw statement.

   throw new UnsupportedOperationException("not implemented yet!");

   I can now compile the class and start working on the implementation of the methods one by one. Any attempt at using a method that has not been implemented yet will be signaled with the proper exception to remind us that we still have to implement that method. You can ask your TA for further information if needed.

   Create a test class called OrderedListTest, at this point it will contain only a main method that declares an OrderedList and creates an OrderedList object. (I leave it up to you to JUnit or not for implementing the test cases.)

   Make sure that your implementation is correct, i.e. has all the elements presented above and compiles. When you are done compiling all the files, proceed with the next step, implementing the method size().

   Files

   - OrderedStructure.java
   - OrderedList.java
   - OrderedListTest.java
2. Implement the method size(), i.e. replace the throw statement by an iterative implementation that traverses the list and counts the number of elements. Add a

test to the test class, simply to check that the method size works for the empty list. We will check the other cases when the method add has been completed.

Files

- OrderedStructure.java
- OrderedList.java
- OrderedListTest.java

3. Implement the method boolean add(Comparable obj); adds an element in increasing order according to the class' natural comparison method (i.e. uses the method compareTo). Returns true if the element can be successfully added to this OrderedList, and false otherwise.
4. Add test cases for the method add. It will be difficult to thoroughly test the implementation, but at least the size of the list should change as new elements are added.
5. Implement Object get(int pos). It returns the element at the specified position in this OrderedList; the first element has the index 0. This operation must not change the state of the OrderedList;
6. Add test cases for the methods add and get to the test class. You are now be in a better position for testing all three methods, add, get and size. In particular, you should be able to add elements, use a while loop to get all the elements of the list, one by one, using the method get. Make sure that all three methods are fully debugged before continuing.

Files

- OrderedStructure.java
- OrderedList.java
- OrderedListTest.java

7. Implement void remove(int pos); Removes the element at the specified position in this OrderedList; the first element has the index 0.
8. Add test cases for the method remove to the test class. Make sure that the method remove (as well as all the other methods) is fully debugged before continuing.

Files

- OrderedStructure.java
- OrderedList.java
- OrderedListTest.java

We now have a complete implementation of the interface OrderedStructure.

2. Write an instance method, void merge(OrderedList other), that adds all the elements of the other list to this list, such that this list preserves it property of being an ordered list. For example, let a and b be two OrderedLists, such that a contains “D”, “E” and “G”, and b contains “A”, “C”, “D” and “F”, the call a.merge(b) transforms a such that it now

contains the following elements “A”, “C”, “D”, “D”, “E”, “F” and “G”; b should not be changed by the method call. The class String implements the interface Comparable and could be used for testing.

The objectives are to learn how to traverse and transform a doubly linked list. Therefore, you are not allowed to use any auxiliary or existing methods, in particular add, for your implementation!

Your implementation should traverse both lists and insert the elements (values) of the other list, in order, into this list. Remember that it is better to practice now than at the final examination!

Files

- OrderedStructure.java
- OrderedList.java
- OrderedListTest.java

(Advanced and optional topic) Use your implementation of the class OrderedList as a starting point for creating a parametrized implementation of the following interface.

```
public interface OrderedStructure< T extends Comparable<T> > {

   public abstract int size();

   public abstract boolean add( T elem ) throws IllegalArgumentException;

   public abstract T get( int pos ) throws IndexOutOfBoundsException;

   public abstract void remove( int pos ) throws IndexOutOfBoundsException;

}
```

The added benefit of using generics will be that all the elements of a list will be of the same type. Therefore, the method compareTo(other) will always be passed an object of the same type as the instance.

- OrderedStructure.java
- OrderedList.java
- OrderedListTest.java

4 Quiz (1 mark)

For the ArrayList implementation of the interface List.

1. Insertions at intermediate positions are always fast.
2. Adding an element at the first position is always fast.

3. Removing an element is always fast.
4. Reading the value of an intermediate position is always fast.

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

Last Modified: July 9, 2016