ITI 1121. Introduction to Computing II
Winter 2015

Assignment 4 [PDF]
(Last modified on April 16, 2015)

Deadline: Tuesday April 14, 2015, 23:59

Solution

- a4_2015.zip

Learning objectives

- Implementing map using linked elements
- Further understanding of circular queues
- Applying queues to problem solving
- Applying iterators to problem solving
- Introduction to recursive list processing
- A basic introduction to tree structures

Hint: This assignment comprises a series of questions similar to those you might get on the final examination. Since the questions are independent one from another, solve them as soon as the related concepts have been presented in class. For instance, you should be able to solve questions 1, 2, and 3 during the week of March 23, 2015.

1 LinkedDictionary (25 marks)

This question is about the interface **Map** of **Assignment #3**. Here, we ask you to provide a linked-based implementation. **LinkedDictionary** keeps track of **String-Token** (identifier-value) associations. Such data structure is sometimes called a symbol table.

- The class **LinkedDictionary** implements the interface **Map<String,Token>**.
- The elements of the dictionary are stored in linked elements, objects of a private static nested class, named **Elem**. An **Elem** object stores a **key** (type String), a **value** (type Token), and **next** (type Elem).

Files:

- **01/LinkedDictionary.java**
- 01/Map.java
- 01/Token.java
- 01/LinkedDictionaryTest.java

The above JUnit test cases can be used to validate **LinkedDictionary**. Do not assume that the tests are exhaustive. In particular, do not assume that a method that passes these tests gets a perfect score.

2 Queue (15 marks)

For this question, a **circular array** is used to implement a queue. You must implement a special method **dequeue** that takes one parameter (an integer) specifying how many elements need to be removed from the queue. The method **dequeue** returns a reference to a list containing all the elements that have been removed, in reverse of order of removal. Use the predefined class **LinkedList** as an implementation of the interface **List** for returning the values.

- For your implementation of the method **dequeue**, you cannot use the methods of the class **CircularQueue**. Your code needs to manipulate directly the instance variables, **elems**, **front**, **rear**, and **size**.

The execution of the Java program below displays "[2,1,0]".

```
CircularQueue<Integer> q;
q = new CircularQueue<Integer>(100);

for (int i=0; i<8; i++) {
    q.enqueue(i);
}

List<Integer> l;
l = q.dequeue(3);

System.out.print("[");
for (int i=0; i<l.size(); i++) {
    if (i>0) {
        System.out.print(",");
    }
    System.out.print(l.get(i));
}
System.out.println("]");
```

Files:

- 02/Queue.java
- 02/CircularQueue.java
- 02/EmptyQueueException.java
- 02/Test.java

3 Applications of queues (20 marks)

The context for this question is a finance software package called **JStock**. Corporations are raising funds by selling **shares** (equal portions of their capital). Collectively, the shares of a given corporation is called its **stock**. The shares prices vary, here on a daily basis. A shareholder makes a **capital gain** when selling shares if the selling price is higher than the price at which the shares were bought; or, suffers a **capital loss** if the selling price is lower than the price at which they were bought.

When selling shares, the calculation of the **capital gain** is easy if all the shares were purchased at the same price (e.g. a single transaction). However, the computation is more complex when selling shares acquired through several transactions. In that case, standard accounting principles dictate that the oldest shares must be sold first.

For example, a shareholder purchases 100 shares at \$20 each in a first transaction, then purchases 20 shares at \$24 each in a second transaction, then purchases 200 shares at \$36 each in a third transaction, and then sells 150 shares at \$30 each. In that case, the capital gain is $100 \times (30 - 20) + 20 \times (30 - 24) + 30 \times (30 - 36) = 940$ dollars.

JStock is a software package to help shareholders manage their portfolio. For simplicity, **JStock** holds shares of single stock (the shares of single corporation). However, the shares are generally acquired through several transactions.

All the shares that were purchased in a given transaction are kept in a **Transaction** object. The number of shares and the price for each one is specified when creating a new **Transaction** object. The class declares the methods **getShares** and **getSharePrice** that returns for a given transaction the number of shares and the price of each share, respectively. The class also implements a method to decrease the number of shares of a given transaction, the method is called **sell**, its parameter is the number of shares sold.

Implement the class **JStock**. **JStock** uses a queue to store the transactions of a given stock.

1. Implement the method **void buy(int num, float sharePrice)**. It adds a new transaction at the rear of the queue. The values of the parameters are used to create a new transaction;
2. Implement the method **float sell(int num, float sharePrice)**. It updates the queue of transactions so as to reduce the total number of shares by **num**, and returns the resulting capital gain or capital loss (a negative gain);
3. Implement the method **float getValue()** that returns the total value of the portfolio. This is the sum of the value of all the transactions. The value of a transaction is simply the product of the number of shares by the share price.

Files:

- **03/JStock.java**
- 03/Transaction.java
- 03/Queue.java
- 03/LinkedQueue.java
- 03/EmptyQueueException.java

4 Iterators (15 marks)

Write a method named **frequency** that takes as input a list of objects containing a **character** value and a **boolean** flag initially set to **false**, and prints the frequency of each character of the list. Every time an element is **counted**, its flag is set to **true** so that it is not printed or counted a second time.

```
List<Tuple> l;
l = new LinkedList<Tuple>();

l.add(new Tuple('a')); l.add(new Tuple('b')); l.add(new Tuple('a'));
l.add(new Tuple('c')); l.add(new Tuple('b')); l.add(new Tuple('a'));
l.add(new Tuple('c')); l.add(new Tuple('a')); l.add(new Tuple('d'));
l.add(new Tuple('d')); l.add(new Tuple('b'));

Frequency.frequency(l);
```

Executing the above program produces the following output “a : 4, b : 3, c : 2, d : 2”. Below you will find a schematic representation of the list for the execution of the above program. Each set of parentheses contains a character and boolean value. Here **t** and **f** are used to represent **true** and **false**, respectively. This is the list before the execution of the program.

```
(a,f)->(b,f)->(a,f)->(c,f)->(b,f)->(a,f)->(c,f)->(a,f)->(d,f)->(d,f)->(b,f)
```

The method **frequency** will first display **a : 4**. The list will have been transformed as follows.

```
(a,t)->(b,f)->(a,t)->(c,f)->(b,f)->(a,t)->(c,f)->(a,t)->(d,f)->(d,f)->(b,f)
```

Next, the method displays **b : 3**. The list will have been transformed as follows.

```
(a,t)->(b,t)->(a,t)->(c,f)->(b,t)->(a,t)->(c,f)->(a,t)->(d,f)->(d,f)->(b,t)
```

Next, the method displays **c : 2**. The list will have been transformed as follows.

```
(a,t)->(b,t)->(a,t)->(c,t)->(b,t)->(a,t)->(c,t)->(a,t)->(d,f)->(d,f)->(b,t)
```

Finally, the method displays **d : 2**. The list will have been transformed as follows.

```
(a,t)->(b,t)->(a,t)->(c,t)->(b,t)->(a,t)->(c,t)->(a,t)->(d,t)->(d,t)->(b,t)
```

Your implementation must comply with the following directives:

- You need to use iterators to traverse the list. In fact, the only method of the **List** that you can use is the method **iterator**, which returns an iterator on the list.
- The frequency table should not be saved, just printed. In particular, you cannot use arrays, lists, stacks or queues to store counts. The counts are simply printed.

Objects of the class **Tuple** are used to store a **character** and a **boolean**. The value of the **boolean** is initially **false**. The method **toggle** is used to invert the value of **visited**.

```java
public class Tuple {

    private char c;
    private boolean visited;

    public Tuple(char c) {
        this.c = c;
        visited = false;
    }

    public void toggle() {
        visited = ! visited;
    }

    public boolean visited() {
        return visited;
    }

    public char getChar() {
        return c;
    }

    public String toString() {
        if (visited) {
            return "(" + c + ",t)";
        } else {
            return "(" + c + ",f)";
        }
    }
}
```

This question is about the abstract data type **List** and iterators. The interface **List** declares a method named **iterator**, which has a return value of type **Iterator**.
Files:

- **04/Frequency.java**
- 04/Tuple.java
- 04/Test.java

5 Recursion (15 marks)

In the class **SinglyLinkedList**, write a **recursive** (instance) method that returns a list of all the positions where a given object is found. The method must be implemented following the technique presented in class for implementing recursive methods inside the class. Briefly, there will be a public method that initiates the first call to the recursive private method.

Let **l** designate a list containing the elements {A, B, A, A, C, D, A}, then a call **l.indexOfAll(A)** would return the following list of positions {0, 2, 3, 6}.
Files:

- 05/SinglyLinkedList.java
- 05/Test.java

6 Binary Search Tree (10 marks)

Implement the method **int count(E low, E high)** for the binary search tree presented in class. The method returns the number of elements in the tree that are greater than or equal to **low** and smaller than or equal to **high**.

- The elements stored in a binary search tree implement the interface **Comparable<E>**. Recall that the method **int compareTo(E other)** returns a negative integer, zero, or a positive integer as the instance is less than, equal to, or greater than the specified object.

- A method that is visiting too many nodes will get a maximum of 9 marks.
- Given a binary search tree, **t**, containing the values **1, 2, 3, 4, 5, 6, 7, 8**, the call **t.count(3,6)** returns the value **4**.

Files:

- 06/BinarySearchTree.java
- 06/Test.java

7 Rules and regulation

Follow all the directives available on the assignment directives web page, and submit your assignment through the on-line submission system uottawa.blackboard.com. You must preferably do the assignment in teams of two, but you can also do the assignment individually. Pay attention to the directives and answer all the following questions.

Submit a zip file with six (6) sub-directories: 01 to 06. You must hand in all the above files. Here is a zip file with the given files.

- a4.zip

A Frequently Asked Questions (FAQ)

1. **"Should I worry about memory leaks?"**

   Yes, you have to make sure that your code does not fail to promote the release of unwanted objects.

2. **"Should I worry about the efficiency of my logic?"**

   Yes, you need to get used to creating efficient logic as much as possible.

3. **"Why are there so many questions on this assignment?"**

   This assignment is a very good exercise for data structures. You will need to master the creation and manipulation of data structures in your computing career. It is also a very good exercise for the final exam!

4. **"For the method indexOfAll, shouldn't the return type be SinglyLinkedList<Integer>?**

   Abosolutely! I've made the change.

5. **"For the class SinglyLinkedList in Question 5, shouldn't the method addFirst throw an exception if the parameter is null?"**

   Good catch! I'm changing it now....

Last Modified: April 16, 2015