ITI 1121. Introduction to Computing II
Winter 2018

Assignment 4
(Last modified on March 31, 2018)

Deadline: April 11, 2018, 11:30 pm

[PDF]

Learning objectives

- Modifying a linked structure
- Adapting the implementation of an iterator
- Writing code using an iterator
- Writing recursive methods to process the elements of a list
- Implementing methods for a binary search tree

1 ITIStringBuffer (25 marks)

In Java, **String** instances are immutables objects, so their states cannot be changed. As a consequence, the instruction line 4 below:

```
1 String marco = "Marco";
2 String polo = " Polo";
3
4 marco = marco + polo;
```

does not actually **update** the **String** instance referenced by *marco*, since it cannot be modified. Instead,

1. a new **String** instance of size **marco.length() + polo.length()** is created,
2. the current characters in the instance *marco* are copied into that new instance,
3. the characters in of the instance *polo* are then copied into that new instance,
4. and the reference *marco* is updated to point at that new **String** instance

This process is usually acceptable, except when it needs to be repeated a large number of times. A typical example would be the **toString()** method of our lists, queues, etc., which may have to process thousands of **String** instances. A test shown below shows that concatenating 10,000 strings took anywhere between 3 and 12 seconds depending on the size of the strings. This is not sustainable.

Of course, Java offers some solid alternatives, such as the class StringBuffer. The goal of this exercise is to create **ITIStringBuffer**, our own implementation of **StringBuffer**. Our class will not be as optimized as the original **StringBuffer**, but will be **many** orders of magnitudes faster than a direct **String** concatenation.

In addition to its constructor, **ITIStringBuffer** only has two methods:

- **void append(String s)**, adds the **String** instance designated by *s* to the set of strings references already added to the **ITIStringBuffer** instance,
- **String toString()** returns a reference to a **String** object which contains the text of all the strings added so far via the method **append**, in the order in which they have been added.

The following code shows how we will use **ITIStringBuffer**

```
ITIStringBuffer marco = new ITIStringBuffer();
marco.append("Marco");
marco.append(" Polo");
System.out.println(marco.toString());
marco.append(" walks into a bar...");
System.out.println(marco);
```

This will produce

```
Marco Polo
Marco Polo walks into a bar...
```

We have the following constraints for our **ITIStringBuffer** implementation:

1. Of course, we cannot use **StringBuffer**, or any other class that would do the work for us.
2. Our implementation will use **one, and only one** instance of a **SinglyLinkedList**.
3. The method **append** will **always** be **fast**, that is, its execution time will always be reliable, and in **no circumstance** will it be a function of the number of elements in the list.
4. We can call the method **toString** at any time, and then keep using the **append** method as needed
5. If the method **toString** is called several times in a row, without calling **append** in between, then it should provide the result without noticeable delay.

.

Here is an example of using **ITIStringBuffer**, and comparing it with concatenating strings:

```
public class Q1 {

    private static int sizeOfTest = 10000;
    private static String testString = "This is some string that I am testing with ...";

    private static String resultBuffer = "";
    private static String resultConcat = "";
    private static ITIStringBuffer buffer = new ITIStringBuffer();

    private static void testStringBuffer() {

        long start = System.currentTimeMillis();
                for(int i = 0; i < sizeOfTest; i++){
                        buffer.append(testString + i );
                }
        long inter = System.currentTimeMillis();
        resultBuffer = buffer.toString();
        long end = System.currentTimeMillis();

                System.out.print("It took " + (end-start) + " ms with ITI1121StringBuffer ("
                        + (inter-start) + " to append and "
                        + (end - inter) + " to generate).");

        inter = System.currentTimeMillis();
        resultBuffer = buffer.toString();
        end = System.currentTimeMillis();
                System.out.println(" Regenerating the string took " + (end-inter) + " ms.");
    }

    private static void testStringAppend(){

        long start = System.currentTimeMillis();
                for(int i = 0; i < sizeOfTest; i++){
                        resultConcat += testString + i;
                }
        long end = System.currentTimeMillis();

                System.out.println("It took " + (end-start) + " ms directly with String.");
    }

    private static void oneRound() {
```

```
42
43          testStringBuffer();
44          testStringAppend();
45          if(!resultBuffer.equals(resultConcat)) {
46              System.out.println("Error, the strings are not the same");
47              return;
48          } else{
49              System.out.println("OK, the strings are the same.");
50          }
51
52      }
53
54
55      public static void main(String[] args) {
56
57          System.out.println("First round");
58          oneRound();
59
60          System.out.println("Second round");
61          oneRound();
62
63          System.out.println("Third round");
64          oneRound();
65      }
66  }
```

Here is a sample output for the code above:

```
1  $ java Test
2  First round
3  It took 14 ms with ITI1121StringBuffer (8 to append and 6 to generate). Regenerating the string took 0 ms.
4  It took 2786 ms directly with String.
5  OK, the strings are the same.
6  Second round
7  It took 6 ms with ITI1121StringBuffer (2 to append and 4 to generate). Regenerating the string took 0 ms.
8  It took 6761 ms directly with String.
9  OK, the strings are the same.
10 Third round
11 It took 6 ms with ITI1121StringBuffer (2 to append and 4 to generate). Regenerating the string took 0 ms.
12 It took 12073 ms directly with String.
13 OK, the strings are the same.
14 $
```

The **String** constructor String(char[] value) and the **String** method char[] toCharArray() will help you with your implementation.

Files

Use the following files as a starting point.

- ITIStringBuffer.java
- SinglyLinkedList.java and List.java
- Q1.java

2 Iterator (20 marks)

An **Iterator** allows a programmer to traverse a data structure efficiently without exposing the implementation. For this question, you must follow the directives below and modify the implementation accordingly.

2.1 int nextIndex()

Modify the interface **Iterator** to have a method **int nextIndex()**. Make all the necessary changes to the class **LinkedList** to implement the method.

- The method **nextIndex** returns the index of the element that would be returned by a subsequent call to **next()**.

The following Java code shows the expected behaviour.

```
1  LinkedList<String> alphabet;
2  alphabet = new LinkedList<String>();
3
4  alphabet.add("alpha");
5  alphabet.add("bravo");
6  alphabet.add("charlie");
7  alphabet.add("delta");
8  alphabet.add("echo");
9
10 Iterator<String> i;
11 i = alphabet.iterator();
12
13 while (i.hasNext()) {
14     System.out.println(i.nextIndex());
15     System.out.println(i.next());
16 }
17 System.out.println(i.nextIndex());
```

Executing the above Java code, produces the following output.

```
$ java Test
0
alpha
1
bravo
2
charlie
3
delta
4
echo
5
```

2.2 Iterator<E> iterator(int nexIndex)

Modify the interface **List** to have a method Iterator <E> iterator (**int** nexIndex). Make all the necessary changes to the class **LinkedList** to implement the method.

- Returns an iterator over the elements in this list, starting at the specified position in the list. The specified index indicates the first element that would be returned by an initial call to **next()**. The method throws **IndexOutOfBoundsException**, if the **nextIndex** is out of range.

2.3 Iterator<E> iterator(Iterator<E> other)

Modify the interface **List** to have a method Iterator <E> iterator (Iterator <E> other). Make all the necessary changes to the class **LinkedList** to implement the method.

- Returns an iterator over the elements in this list, starting at the same position as **other**. Calling the method **next** of this iterator or that of **other** should return the same element.

Files

Use the following files as a starting point.

- Iterator.java
- List.java
- LinkedList.java
- Q2.java

3 Recursive List Processing (15 marks)

Implement the method **int count(E fromElement, E toElement)** for the class **SinglyLinkedList<E>** presented in class. The method returns the number of elements in the list between the **first** instance of *fromElement* and the **first** instance of *toElement* in the list, including ***fromElement* and *toElement***.

We have the following constraints and specifications for our methods:

- The implementation of **count** must be *recursive*. No mark will be given for an iterative implementation.
- If **fromElement** cannot be found in the list, the method returns 0.
- If **toElement** cannot be found in the list after **fromElement**, the method returns the number of elements in the list between the first instance of **fromElement** and the end of the list, **fromElement** included.
- The method should be efficient and not visit nodes it doesn't need to visit.

Here is an illustration of the use of that method:

```
1  public class Q3 {
2
3      private static void testCount(String first, String second, SinglyLinkedList<String> l) {
4        System.out.println("Distance between " + first
5          + " and " + second +" is " + l.count(first,second));
6      }
7      public static void main(String[] args) {
8
9          SinglyLinkedList<String> l;
10         l = new SinglyLinkedList<String>();
11
12         l.add("A");
13         l.add("B");
14         l.add("C");
15         l.add("D");
16         l.add("A");
17         l.add("B");
18         l.add("C");
19         l.add("D");
20         l.add("E");
21
22         System.out.println(l);
23         testCount("A","A",l);
24         testCount("A","B",l);
25         testCount("A","D",l);
26         testCount("A","E",l);
27         testCount("A","F",l);
28         testCount("B","B",l);
29         testCount("B","A",l);
30         testCount("B","D",l);
31         testCount("B","E",l);
32         testCount("B","F",l);
33         testCount("E","A",l);
34         testCount("F","A",l);
35         testCount(null,"A",l);
36         testCount("A",null,l);
37
38     }
39 }
```

Running this code produces the following output:

```
1  [A B C D A B C D E]
2  Distance between A and A is 1
3  Distance between A and B is 2
4  Distance between A and D is 4
5  Distance between A and E is 9
6  Distance between A and F is 9
7  Distance between B and B is 1
8  Distance between B and A is 4
9  Distance between B and D is 3
10 Distance between B and E is 8
```

```
Distance between B and F is 8
Distance between E and A is 1
Distance between F and A is 0
Distance between null and A is 0
Distance between A and null is 9
```

Files

Use the following files as a starting point.

- SinglyLinkedList.java and List.java
- Q3.java

4 Binary Search Tree (15 marks)

Implement the method **int count(E low, E high)** for the class **BinarySearchTree** 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 be penalized.
- 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**.

File

Use the following file as a starting point.

- BinarySearchTree.java
- Q4.java

Files

You must hand in a **zip** file (no other file format will be accepted). The name of the top directory has to have the following form: **a4_3000000_3000001**, where 3000000 and 3000001 are the student numbers of the team members submitting the assignment (simply repeat the same number if your team has one member). The folder must contain the following files.

- A text file **README.txt** which contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- The source code of **all** your files (all the necessary files to compile and execute your program).
- The corresponding JavaDoc doc directory.
- StudentInfo.java, properly completed and properly called from your main method.

The archive a4_3000000_3000001.zip contains the files that you can use as a starting point.

WARNINGS

- Failing to strictly follow the submission instructions will cause automated test tools to fail on your submission. Consequently, your submission will **not** get marked.
- A tool will be used to detect similarities between submissions. We will run that tool on all submissions, across all the sections of the course (including French sections). Submissions that are flagged by the tool will receive a mark of 0.
- It is your responsibility to ensure that your submission is indeed received by the back-end software, Brightspace. If your submission is not there by the deadline, it will obviously **not** get marked.
- Late submissions will not be accepted.

Last modified: March 31, 2018