Project 1: Sequence of Rectangles

Total: 100 points

Posted 09/15/2016

Due 10/08/2016

Objective

The purpose of this project is to develop a generic collection class that uses a linked list. You will also review how to construct and use Java classes as well as obtain experience with software design and testing.

ABET Program Learning Outcomes:

- The ability to recognize the need for data structures and choose the appropriate data structure (b,c,i,j)
- The ability to design and implement algorithms for various searching problems (a,b,c,k,i,j)

Introduction - The Problem

In this project your program demonstrates two applications of the linked list based sequence ADT. The first stores Rectangle objects, the second stores String objects. Your program must test all the sequence operations at least in one of the two data structure created. The rectangle objects are randomly created and added to the list, the string objects (names) you choose directly or read them from a file you create for input.

Design, Implementation

The implementation of the data structure ADT will be based on two user defined classes: **`Node`** and **`LinkedSequence`**, both generic. A **`Rectangle`** class must also be added to project, since in one of the applications the stored data are Rectangle objects. The other application utilizes the String class from the Java library. For testing purposes two more classes will be needed: **`TestNode`** to test the Node class methods, and **`LinkedApplications`** to use the data structure.

1. The **`Rectangle`** class (not generic) has two fields named **width** and **length,** both of type **int.** You may use package access to the fields. The class has
   - a constructor taking two parameters to initialize the fields
   - a toString( ) method which returns a String message that specifies the data field values
   - an equals( ) method which returns true if each of the corresponding field values are equal in two given rectangles

2. **Node**

This class is generic, yet as for a template, you are advised to study the IntNodeClass specifications (pp 208-211) in the book, as well as the implementations in the previous sections and on pages 212-214. As for a generic Node class there are some hints in the book on pages 283-285.

Our **Node** class is similar but not identical to the template. We may discard or modify some methods and we may include new methods which are practical in our applications.

Partial class documentation:

- Fields: as usual, but with the generic type parameter
- Constructor: takes two parameters to initialize the fields

Instance methods

- **`addNodeAfter()`** as in the book, but not void, returns the link it creates.
- **`removeNodeAfter()`** as in the book, but returns the link it creates
- **`toString()`** creates and returns a string message which reveals the stored values as well as the links. (This method is recursive, see HW 4.)

Static methods

- **`listCopy()`** as in the book, but, since our `addNodeAfter( )` returns the link, you should assign the return value directly to copyTail . That is, combine the two copyTail statements of the while loop in one statement.
- **`listPosition()`** as in the book
- **`listLength()`**as in the book
- **`getTail() This is a new method!`** Starting at a given node parameter the method iterates through the list until arrives at the tail, then it returns the tail. The iteration is the same as in the listCopy method. This method helps to maintain the tail reference, and combined with the listCopy method, the listCopyWithTail becomes superfluous
- **`listCopyWithTail()`** omitted
- **`listPart()`** omitted
- **`listSearch()`**omitted

3. **LinkedSequence**

This class is also implemented as a **generic** class. The specification of a non-generic DoubleLinkedSeq class is fairly well detailed in your reading, pp 232 –

238, it helps to design the generic class. Also, the LinkedBag implementation in the book can be helpful, but the analogy is rather weak.

Fields

- The field manyNodes is optional.

It can be convenient to have direct access to the size of the sequence, but this variable needs careful update in each method that manipulates the size, and this can be a source of errors. For instance, the removeNodeAfter() method changes the size most of the time, but no change when tail is the reference. The listLength() method of the Node class makes the field dispensable, but the length algorithm is O(n).

There are five private variables of type Node:

- head
- tail
- cursor
- precursor
- dummy

All these except dummy require accessor and mutator methods.
You may instantiate dummy with two null parameters at the declaration.

Constructor

- Takes one node parameter to initialize the head. Calls the static getTail method of the Node class with head for parameter to initialize tail. Assigns dummy's link the head. Note that dummy does not store any info data but null, and it is not part of the data structure. Its link is always the head.

Instance Methods

- **`addAfter`**( ) Takes a parameter for the new **data** value to be added to the structure and returns the currently added node. Cursor reference is always updated to the added node. A call to the addNodeAfter method of the Node class shall add the new node to the list
  `(i)` after dummy if head is null
  `(ii)` after tail if cursor is null but head is not
  `(iii)` after cursor if cursor is not null

  Build the selection logic carefully and update the relevant fields as necessary
- **`addBefore`**( ) Takes a parameter for the new **data** value to be stored in the structure and returns the currently added node. Cursor reference is

always updated to the added node. A call to the addNodeAfter method of the Node class shall add the new node to the list
`(i)` after dummy if precursor is null
`(ii)` after precursor if precursor is not null

- **`addAll( )`** Takes another linked sequence for parameter (‘**other**’) and joins the parameter list to this list after the tail. If the parameter is null or empty, the method returns. Otherwise, if the calling list is empty (head is null) this head assigned other head and this tail assigned other tail; else tail link assigned other head and tail assigned other tail.
- **`advance()`** advances the cursor forward one step, if the cursor is not null and not the tail; precursor is updated accordingly. Null cursor is advanced to head, tail cursor advanced to null.
- **`clone( )`** Returns a copy of the calling sequences. See the specifications in the book.
- **`concatenate( )`** static method; takes two linked sequence parameters and creates a third sequence by adding the second after the first. You have to use the listCopy( ) and getTail( ) static methods from Node. **Do not use** listCopyWithTail( ).
- **`removeCurrent()`** removes and returns the current node from the lists. Cursor and precursor are updated as necessary. If the head is removed, the new head is the cursor.
- **`displayList( )`** convenience method. Prints all the nodes to the console; use the toString( ) call with respect to head.
- **`isCurrent(), getCurrent(), start( )`** are all by the book, testing is not required

4. `TestNode`

Create three-node long list(s) of String type and test all the Node methods. This part of the Project is an extension of HW 4, you may use your code from that assignment.

5. `LinkedApplications`

Requirements

- Create short three to five long String type linked list to test all the methods.
- The displayList( ) method must be tested on Rectangle type list
- Add two **static** fields of type **long** to the application class named **squares** and **occurrences**
- Add a static void method named **counting** to the applications class. The method takes a Rectangle array named boxes and a Rectangle object named target for parameters.
- The method counts the number of squares among the array element, and stores the value in the **squares** field, and it also counts the number of array elements that are equal to the target and stores the result in the **occurrences** field.

- Step 1: Create a LinkedSequence of 100 000 Rectangle objects each having integer dimensions randomly selected between 1 and 30.
- Step 2: Verify that the listPosition() method returns the tail reference for position number 100 000.
- Step 3: instantiate a Rectangle array to the list length (do not use literals) and load the Rectangles from the list to the array.
- Step 4: Create a target Rectangle with side 15, 15 and call the counting method passing your array and target as parameters.
- Step 5: Measure the running time of each step above as well as the combined time and record the results. Hint: use the method call **System.nanoTime()** to record the current real time in nanoseconds (the return type of the method is **long**); note that **10^9** nanos make one second.
- Step 6: Repeat Steps 1- 5 for 1 000 000 Rectangle objects
- Step 7: Repeat Steps 1 – 5. 10 000 000 Rectangle objects.
- Step 8: Repeat Step 7 by adding the non-random target rectangle of step 4 to the empty list 10 000 000 times. Check out if the random selection in step 7 is a significant overhead for the running time of the algorithm or not.
- Analyze your running time observations, deduce Big-Oh estimates and advise about the expected time for the case of 100 000 000 rectangles. Attach your report as a comment to the source code following the application class.

Documentation: Each method in Node and LinkedSequence must be documented with the full specification; for documentation on the fields compose and include the ADT invariant to the LinkedSequence class only. The book (p. 233) hints in a reasonable way to this task.

Evaluation (100 points)

1. Documentation and Style………………………………………….. 20 pts
2. Correct implementation
   Node class methods (3 pts each)………………………….21 pts
   LinkedSequence class methods (8 x 3 + 3 x1) ………..…27 pts
3. Correct testing
   Node………………………………………………………….10 pts
   LinkedSequence……………………………………………12 pts
4. Running time analysis and report ………………………………...10 pts

Total……………………………………………………………………...100 pts

Submit:

Zip the project folder containing all sour codes and the report. Upload at BlackBoard