Worksheet 4: Linked Lists and Iterators

Updated: 20th March, 2019

Aims

- Implement and test a Linked List ADT (single-ended, singly-linked)
- Extend and test your Linked List ADT to be double-ended and doubly-linked
- Extend and test your Linked List ADT to provide an iterator
- Extend and test your previous Stack implementation to offer a Linked List version
- Extend and test your previous Queue implementation to offer a Linked List version

Read through the practical before starting to ensure you understand where the activities will be leading. Almost all of required code is given as pseudocode in the lecture.

Ensure you have completed the activities from previous practicals because this week will build on top of the classes developed.

1. Linked Lists - Single/Single

As a starting point to developing code, draw a UML diagram to represent DSALinkedList and DSAListNode classes, along with you teat harness code. Use the pseudocode from the lectures to assist you in developing DSAListNode and DSALinkedList.

Include at least the following public methods for DSALinkedList:

- boolean isEmpty()
- void insertFirst(Object inValue)
- void insertLast(Object inValue) – this will be much simpler with a tail pointer
- Object peekFirst()
- Object peekLast()
- Object removeFirst()
- Object removeLast()

Ensure that the peeks and removeFirst() return the value of the ListNode!

As you build the class, you should create a matching test harness to Unit Test the class. This test harness should exercise all class fields and methods, as well as any validation code you have included.

In this unit, invalid input/results should raise an exception. Your test harness should expect the exception and record it as a passed test if an exception occurs. There is a sample Unit Test Harness on Blackboard that you can use as a guide.

Python Students:

UML diagrams should be fairly language independent... Python students should indicate "private" methods, even if they wont teachnically be provate. You can put an underscare in front of provate methods as a convention.

Java Students:

You may decide to make DSAListNode a separate .java file or place it as a private class within DSALinkedList. Note that the latter will mean that you cannot return DSAListNode to any client/user of DSALinkedList. This is actually good design since it promotes information hiding (how the linked list works under the covers should not be something that clients should know about). See the Lecture Notes for how to make a private inner class

2. Linked Lists - Double/Double

Extend the list to be doubly-linked, double-ended – that is, maintain both a head and a tail pointer as member fields. This makes the linked list far more efficient for queues.

- When implementing insertFirst(), use linked list diagrams like those in the lectures to help you decide how to maintain tail as well as head. Consider the possible cases: inserting into (i) empty list, (ii) one-item list, (iii) multi-item list. Some might end up working the same, but you still need to think it through.
- insertLast() is your next task: again, drawing diagrams can help.
- removeFirst() can be tricky: again, consider all the above three cases, but note that each case must be handled explicitly (in particular, removing the node in a one item list is a special case since it is both the first and last node).

3. Iterable Lists

Although we have a linked list, it's not really complete without some way of iterating over the elements. To this end we will implement an iterator so that a client of DSALinkedList can iterate through all items in the list. Why iterate with a stack and a queue when you can only take from the top or front? Because there are plenty of times when you want to see what is in the stack/queue, but don't actually want to take from the stack/queue. In particular, we will need this for when the application's user requests to view the orders that are yet to be processed.

Java:

- Create a new class called DSALinkedListIterator that implements the Iterator interface. This time you definitely want to make it a private class inside DSALinkedList (see Lecture 7b) since it must not be exposed externally apart from its Iterator interface.

- Use the code in the lecture notes to guide you on designing and implementing your iterator class. Remember that as an inner class, DSALinkedListIterator has access to the DSALinkedList's private fields – in particular, we want to start at the list's head.
- For Iterator.remove(), just throw an UnsupportedOperationException since it is an optional method anyway.
- Remember to make DSALinkedList implement the Iterable interface and add a public Iterator iterator() method to return a new instance of DSALinkedListIterator. This will also need you to add an import java.util.*; line at the top.

When done, write a suitable test harness to test your iterator-enabled linked list thoroughly. Similarly, we can write iterators for the array-based queues.

4. Queue with a Linked List

We will now extend the Queue implementation to use a Linked List instead of an array. As with the Circular Queue last week, the DSAQueue will be the abstract class above the new queue implementation. Now we will add a new implementation, using a linked list. Draw a UML diagram to represent the relationship between the classes.

You will need to initialise the queue as a linked list, then match the linked list methods to the enqueue/dequeue behaviour.

As a starting point, copy your DSAQueue code and test harness from the previous Practical. Once complete, your queue test harness can be easily modified to test the linked list queue.

5. Stack with a Linked List

We will now extend the Stack implementation to use a Linked List as an alternative to an array. The DSAStack will be an abstract class above the existing stack implementation. Now we will add a new implementation, using a linked list. Draw a UML diagram to represent the relationship between the classes.

You will need to initialise the stack as a linked list, then match the linked list methods to the push/pop behaviour.

As a starting point, copy your DSAStack code and test harness from the previous Practical. Once complete, your stack test harness can be easily modified to test the linked list stack.

Practical Submission

Your UML, code (classes, tests, drivers) and input files are all due before the start of your next practical. All files should be combined into a single zip file and submitted via Blackboard.

Your assessment will be based on:

- UML diagrams
- Linked List implementation
- Test harness for linked list
- Linked list Iterator implementation and testing
- Extending Stack and Queue, with test harness

SUBMIT ELECTRONICALLY VIA BLACKBOARD, under the Assessment section.

End of Worksheet