CURTIN UNIVERSITY (CRICOS number: 00301J)

Department of Computing, Faculty of Engineering and Science

Data Structures and Algorithms (DSA1002)

PRACTICAL 4 – LINKED LISTS

AIMS

- To create a general purpose Linked List class.
- To extend the linked list class with an iterator.
- To convert stack/queue to use the linked list.
- To save the linked list using serialization.

BEFORE THE PRACTICAL:

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

ACTIVITY 1: UML DIAGRAMS

Before you start coding, draw the UML diagrams for DSAListNode, DSALinkedList and their test harnesses. Get feedback on the diagrams before you start coding.

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()

ACTIVITY 2: LINKED LISTS IMPLEMENTATION

Now let's create a linked list class:

- Use the pseudocode from the lectures and the book to assist you in developing DSAListNode and DSALinkedList.
- **<u>BE AWARE THAT THE LECTURE NOTE PSEUDOCODE IS FOR A SINGLE-ENDED LINKED LIST. YOU MUST UPGRADE IT TO BE A DOUBLY-LINKED, DOUBLE-ENDED LINKED LIST!</u>**
  - Java: 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

- Start with a Single-ended, Singly-linked list and test it before going forward.
- Redevelop 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.

- Notes:
  - 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).

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

ACTIVITY 3: ITERATOR FOR THE LINKED LIST

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. For example, we would need this for when an 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) 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. Make sure the iterator methods are in your UML.

ACTIVITY 4: USE DSALINKEDLIST FOR DSASTACK AND DSAQUEUE

Copy your existing DSAStack and DSAQueue into your Practical 3 directory, then convert them to use an DSALinkedList instead of an array. This is pretty easy as it is largely just a matter of hollowing-out the existing methods and calling the appropriate method in DSALinkedList:

- For DSAQueue, have enqueue() perform an insertLast() in DSALinkedList. Conversely, to dequeue() use a combination of peekFirst() and removeFirst() to access the first element and remove it. In other words, organise the DSALinkedList ‘backwards’ so that you can take from index 0 rather than having to first determine the size of the list.
- For DSAStack, have push() perform an insertFirst() and pop() do a peekFirst() + removeFirst() to get the LIFO behaviour. Similar simplifications occur for isEmpty() and other methods.
- Some things can even be deleted: isFull(), count, MAX_CAPACITY, alternate constructor, etc
- Since DSALinkedList has an Iterator, we might as well expose it in DSAStack and DSAQueue to get a free Iterator for these classes. For example:

```
public class DSAStack implements Iterable      // To support for-each loop
{
  private DSALinkedList list;              // List for the stack
  ...                                          // Other DSAStack field and methods
  public Iterator iterator() {
    return list.iterator();                // Expose list's iterator
  }
}
```

ACTIVITY 5: ADD FILE I/O TO YOUR TEST HARNESS

Set up a menu system provide options including:

1) read a serialized file
2) display the list
3) write a serialized file

Add the code required for these tasks with files of integers. Using these routines you should be able to test your list functions and explore building list, saving to file and re-reading in those files to create new lists.

SUBMISSION DELIVERABLE:

Your classes (all that are required for this program) are due before the beginning of your next tutorial. Also include any other relevant classes that you may have created.
SUBMIT ELECTRONICALLY VIA MOODLE, under the *Assessments* section.

MARKING GUIDE

Your submission will be marked as follows:

- [2] UML Diagrams
- [2] Your DSALinkedList and DSAListNode are implemented properly. Your test harness should exercise all the functions in the Linked List.
- [2] Your DSALinkedListIterator is implemented properly. The test harness should test all iterator functionality.
- [2] You have reworked your stacks and queues to use Linked Lists. You should be able to use the previous test harnesses, with minor modifications.
- [2] You can save and load your list using serialization.