CURTIN UNIVERSITY (CRICOS number: 00301J)
Faculty of Engineering and Science
Department of Computing
Data Structures and Algorithms

Practical 8

Aims

- To create a general purpose Linked List class.
- To extend the linked list class with an iterator and convert stack/queue to use the list
- To convert the data structures built so far to use generics
- To get the provided ShedIron and ShedNickel working with your existing classes

Before the Practical:

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

Activity 1: Implement DSALinkedList

Now let's create a linked list class to replace the arrays in DSAStack and DSAQueue, making them much more flexible.

- Use the pseudocode from the lectures and the book to assist you in developing DSAListNode and DSALinkedList.
- **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!**
  - 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
- Develop 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.
- 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()

- 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 2: Implement an Iterator for DSALinkedList

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.

- 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.

Activity 3: Use DSALinkedList for DSAStack and DSAQueue (Due final prac)

Make a backup copy of your existing DSAStack and DSAQueue – call them DSAStackArray and DSAQueueArray. Then convert DSAStack and DSAQueue to use an DSALinkedList

instead of an Object[] array. This is pretty easy as it is largely just a matter of hollowing-out the existing methods and merely 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 c'tor, 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 4: Convert ADTs to Generics

So far we have used Object as a means of making general-purpose stacks, queues and lists that are capable of storing any type of data. However it is a pain to have to always cast from Object to the type you want, not to mention prone to bugs. So now we'll take DSALinkedList, DSAStack and DSAQueue and use Generics to make them general-purpose *whilst also being type-specific*. This is not very hard to do, so is worth trying out.

- Follow the lecture notes on what to do. For example, with DSALinkedList:
  - Open DSALinkedList.java
  - Append <E> everywhere you see "DSALinkedList" to convert it into a generic
  - Similarly, append <E> to every "DSAListNode" and "DSALinkedListIterator"
  - Be careful to put <E> *everywhere*, including changing all Object variables to E, otherwise 'unchecked or unsafe operations' warnings or something about the use of a 'raw type' will pop up during compilation.
  - The only places where you *don't* append <E> are to the constructors where they are declared (this will cause a compilation error). You will have to add an <E> when creating a new object of a class eg queue = new DSAQueue<E>().

    - <E> also has to be appended to the end of Iterator<E> and Iterable<E>.
- Do the same for DSAQueue and DSAStack.
    - Note that the DSALinkedList classes in DSAQueue should also have a <E> appended on the end – what is happening is that the 'E' in DSAQueue<E> is being 'transferred' to be the parameter for the declaration of private DSALinkedList<E> queue.
- In the definition of iterNext(), if you made DSAListNode a private inner class of DSALinkedList, don't forget to put the full DSALinkedList<E>.DSAListNode<E> rather than just DSAListNode<E> - Java has a bug with generics on this.

As usual, when done, write a suitable test harness to test everything thoroughly. Use the data from RandomNames7000.csv

Activity 5: Get IShed, ShedIron and ShedNickel Working

Rather than have you implement the Shed classes, download the pre-written IShed.java, ShedIron.java and ShedNickel.java and make sure they compile against your versions of DSAStack and DSAQueue. This is a useful exercise in itself: you need to take someone else's code and integrate it with your own code – something that you will do quite a bit of during your career as a programmer!

Have a look at the code. Notice that IShed is *nothing but* a list of method names – that's because it is an interface (like Iterator and Iterable) and so only defines *what* but not *how*. The *how* is done in ShedIron and ShedNickel, which implement the IShed interface.

Get the code all compiling and you will have finished with all the 'plumbing' infrastructure and be ready for the final push at the end of semester: writing the user interface and stitching the entire application together. If you finish early, start on the user interface right away because it's not a small amount of coding.

Submission Deliverable:

Your completed `DSALinkedList.java` class (activity 1) is due at the beginning of your next tutorial.

SUBMIT ELECTRONICALLY VIA BLACKBOARD, under the *Assessments* section.

NOTE THAT YOU WILL BE SUBMITTING THE COMPLETE DSA MINING APPLICATION AT THE END OF THE SEMESTER.

So use the next week to get as much done as you can - **Don’t leave it until the last minute because it’s not a small amount of coding and testing!**