Linked Lists

Updated: 7th December, 2021

Aims

- To create a general purpose linked list class.
- To extend the linked list class with an Iterator.
- To convert your previously implemented stack and queue to use the linked list.
- To save the Linked list using serialisation.

Before the Practical

- Read this practical sheet fully before starting.
- Ensure that you have completed the activities from the previous practical, as you will build upon the classes developed last week.

Activities

1. UML Diagrams

Before you start coding, draw the UML diagrams for `DSAListNode`, `DSALinkedList` and their test harnesses.
Include at least the following public methods for `DSALinkedList`:

- boolean `isEmpty()`
- boolean `isFull()`
- 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()`

Get feedback on the diagrams before you start coding.

2. Linked Lists Implementation

Let's create a Linked List class.

- Use the pseudocode from the lecture slides and the textbook to assist you in developing `DSAListNode` and `DSALinkedList`.

- *Be aware that the lecture slide pseudocode is for a **Single-Ended Linked List**. You must upgrade it to be a **Doubly-Linked, Double-Ended Linked List!***
- Start with a Single-ended Singly Linked List and write a test harness to fully test it before going forward, ensuring exception handling is implemented.
- 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.

Note:

- 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).

  Refer to the Lecture Slides for how to make a private inner class
- 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 following possible cases:
  (a) Empty list
  (b) One-item list
  (c) 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 `peek` and `remove` methods return the *value* of the ListNode!

3. Iterator for the Linked List

Although we now 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. When done, write a suitable test harness to test your iterator-enabled linked list thoroughly. Ensure that the iterator methods are in your UML diagrams.

Why iterate with a stack and a queue when you can only take from the top or the front? Because there are plenty of times when you want to see what is in the stack/queue, but don't particularly wish 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.

Note:
Java Students:

- Create a new class called `DSALinkedListIterator` that implements the Iterator interface. This time, you will definitely want to make it a private class *inside* the `DSALinkedList` since it must not be exposed externally apart from its Iterator interface.
- Use the code in the lecture slides 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.

Python Students:

- Use to code in the lecture slides as a guide to define the special methods `__iter__()` and `__next__()` to build an iterator protocol.
  - `__iter__()` sets up your cursor to hold the current position in the collection.
  - `__next__()` defines how you traverse to the next element in the collection. When you reach the end and there is no data left to be returned, raise the `StopIteration` exception
- Once you have these methods, you can iterate through a linked list using a **for** loop.
- Alternatively, you may create a generator using the `yield` statement which maintains the current values of the method/function until the next call. `__next__()` and `StopIteration` are automatically implied.
  The code for this can also be found in the lecture slides.

4. Use DSALinkedList for DSAStack and DSAQueue

Copy your existing `DSAStack` and `DSAQueue` into your Practical 4 directory. You will convert them to use a *linked list* instead of an *array* data structure. This is simple as it is largely just a matter of hollowing-out the existing methods and calling the appropriate methods from your `DSALinkedList` class. Update your UML diagrams to include `DSAStack` and `DSAQueue`.

- 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. You may reverse these operations and still achieve FIFO behaviour.

- For `DSAStack`, have `push()` perform an `insertFirst()`.
  Conversely to `pop()`, use a combination of `peekFirst()` and `removeFirst()` to access the first element and remove it. Again, you may reverse these operations and still achieve LIFO behaviour.
- Similar simplifications occur for `isEmpty()` and other methods. Meanwhile, some things can even be removed from your classes such as `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
        }
}
```

```
class DSAQueue():
        def __init__(self):
                self.DSAQueue = DSALinkedList(). # List for the Queue

        ...  # Other DSAQueue methods

        def __iter__(self):
                return iter(self.DSAQueue). # Exposes list's iterator
```

Remember to cite your previously submitted work, you may leave a comment in your code.

5. Interactive Menu and File I/O for DSALinkedList

Now that we have a linked list, we want to setup an interactive menu system to explore building a list, saving it to a file and re-reading in the file to create new lists. Include at least the following options:

(a) InsertFirst/InsertLast on the list

(b) RemoveFirst/RemoveLast on the list

(c) Display the list

(d) Write a serialised file

(e) Read a serialised file

Use the code in the lecture slides as a guide on how to create a serialised file.

Submission Deliverable

- Your code and UML diagrams are due 2 weeks from your current tutorial session.
  - You will demonstrate your work to your tutors during that session
  - If you have completed the practical earlier, you can demonstrate your work during the next session
- You must **submit** your code and any test data that you have been using **electronically via Blackboard** under the *Assessments* section before your demonstration.
  - Java students, please do not submit the *.class files

Marking Guide

Your submission will be marked as follows:

- [2] Your UML diagrams for your linked list, stack, queue and test harnesses
- [2] Your DSALinkedList and DSAListNode are implemented properly - your test harness should test all functions in the linked list.
- [2] Your DSALinkedListIterator is implemented properly - your test harness should test all iterator functionality.
- [2] Your DSAStack and DSAQueue have been re-developed to use linked lists - your previous test harnesses can be used with minor modifications. *Remember to cite your work.*
- [2] Your interactive menu can save and load your list using serialisation

End of Worksheet