Linked List Data Structures Homework
CSS 162 : Programming Methodology
Instructor Rob Nash

Building Lists, Stacks and Queues with Linked Lists

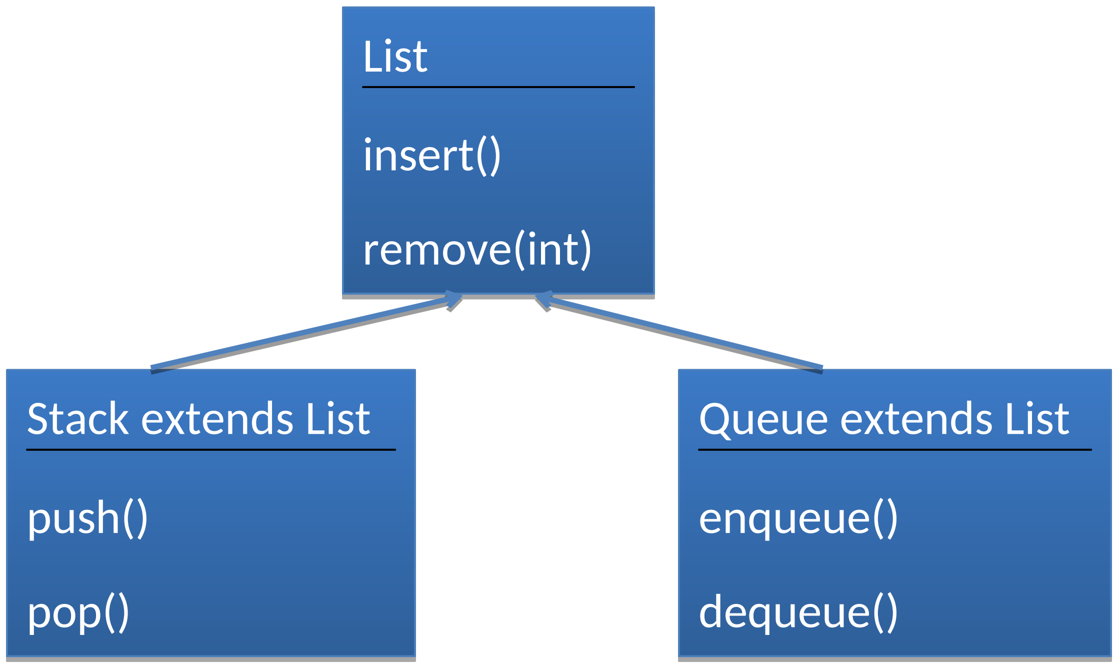
Summary

In this assignment, we'll build three classes that will implement a List, a Stack, and a Queue using Nodes and references rather than arrays. The Stack and Queue will inherit from the List (a linked list) class, and so these subclasses will be rather short. Then, build a simple LinkedListException class that will be thrown in place of RuntimeExceptions whenever exceptional or error situations occur.

Note: Throughout this document, we will usually refer to the List implementation using Nodes and references (i.e., as a linked list) as "List". However, you should not confuse this with the Java built-in interface List, which has the same name. At this point, it won't be an issue; it's just something to keep in mind in the future.

Introduction

In this section, we'll describe the List data structure in detail. Once we've built a List superclass, building two new classes that extend list will be trivial in comparison. The inheritance hierarchy we are going to implement is depicted below; let's start by implementing the top level first.



The List Class

Data

- Node head = null; //the start of the linked list
    - Use no other instance variables here: No int size or Node tail, etc.
    - You probably will implement this as an inner class

Methods

- public void insert(Object next, int index)   (15-30 LOC)
    - Make a new node that holds the Object "next", and insert this at the position specified by index
        - Divide this method into sections/cases using an if
            - If adding to an empty list (changes head) { }
            - else if adding to a single element list (changes head) {}
            - else if adding to a populated list (n>=2)
        - In each case above, note that the following subcases exist
            - Adding to the head of the list (changes head)
            - Adding to the body or the tail of the list (not head)
    - The cases above exist because of the increased complexity involved when implementing a linked list, but essentially reduce to determining "do I change head? Or a node indirectly attached to head?" Note that not all subcases apply to each outer case; for example, you don't worry about adding to the body or tail of a list if your adding to an empty list (the first outer case). Read chapter 15 for more details on this.
- public Object remove(int index) (15-30 LOC)
    - Find and delete the node at the position specified by index
        - Note that this method, like add() above, has the following cases
            - Deleting on an empty list
            - Deleting on a single element list (head only)
            - Deleting on a populated list (n>=2)
        - And, just like above, there are the following subcases
            - Deleting the head of the list (changes head)
            - Deleting from the body or tail of the list (no changes to head)
    - The cases above exist due to the complexity of correctly managing a list of nodes and a head pointer. Note your software will not provide for all (3x2==6) cases, since some subcases don't apply to their enclosing (outer) case. For example, if you're deleting from a single element list (head only), then you don't have to worry about deleting from the body or tail (as this subcase doesn't exist).

- public int size();
    - An standard accessor
- public String toString()
    - Enumerate your list and return this enumeration as a String.
- public boolean isEmpty();
    - Returns true if empty, false otherwise
- public int indexOf(Object target)
    - A sequential search that returns the index if found or -1 if not found

The Stack Subclass

In this section, we'll build a structure that functions like a Stack externally, but is built on top of nodes and pointers internally. Since we'll be extending the List superclass, we only have to provide a few functions (push, pop) that redirect to superclass functions(add, remove) and so this class will be quite short. When you've finished this class, test it using the corresponding driver software.

Data

- None when inheriting from List.

Methods

- public void push(Object next);
    - inserts to the beginning of the list
    - Consider using your superclass methods here rather than implementing push
- public Object pop();
    - removes from the beginning of the list
    - Consider using your superclass methods here rather than building pop by hand
- public static void main(String[] args)
    - A driver to test your stack – see below for more information.

The Queue Subclass

Now, we'll turn our attention to building a Queue. Remember when we used to cut and paste code and logic, then make subtle changes to it to build our new class? Instead, we'll use inheritance here, automate the copy-and-paste, and just focus on implementing the few methods that differ in our subclass (enqueue(), dequeue()). When you're done with this class, test it using the supplied driver software.

Data

- None when inheriting from List.

Methods

- public void enqueue(Object next);

    - inserts to the beginning of the list
- public Object dequeue();
    - removes from the end of the list
- public static void main(String[] args)
    - A driver to test your stack – see below for more information.

Stack & Queue Methods to Override

What happens when you inherit from a class, but you only want 70% of the parent class functionality? Sometimes you can inherit and override to address this, which we will do here, but do note that this is a warning flag and should make you investigate other design options instead. Its known that our use of inheritance here is somewhat hackneyed - the "is a" relationship is being stretched almost to a breaking point with our use of Stacks and Queues which are Lists. Even conceptually, we can all agree that a Stack is not a List and then some (complete inheritance), but rather a List with limitations (partial inheritance). We'll address this with an industry grade solution (Collections in Java) later in this class, but for today, we need to add some "hacks" to our code to block the client from using our Stacks & Queues incorrectly. Whenever you're hacking rather than adding, you're violating a major Object-Oriented design principle, but we'll accept these limitations for today and notice how quickly one can build a Stack or Queue given a List to start from. This reuse can be accomplished using Composition here; another strong alternative to overfitting inheritance. To "fix" our subclasses, we must:

- Override insert() from the superclass to simply call push() or enqueue() without using the index.
- Override remove() from the superclass to call pop() or dequeue() without using the index.
- Do we need to override isEmpty() or indexOf()?

The LinkedListException Class

This class should be under 15 lines of executable code and is an exercise in inheriting from classes provided to you by the Java API. Your class should have only two methods (both constructors) and no data items; see the slides on exceptions for an example of such a class. Throw this exception when an error or exceptional situation occurs in your code, such as a pop() on an empty stack.

Requirements

- You must build at least 4 classes {List, Stack, Queue, LinkedListException}. **Use these names exactly, please.**
- For each data structure class, you must build a driver to test your data structure (see below)
    - Consider common cases and unusual, corner cases.
    - For instance, what if your Stack, Queue, List is empty and I call remove() or pop()?

Building Drivers to Test Your Data Structures

Write a main driver function in each of your three List, Stack, and Queue classes. Here is an ***incomplete*** main that might be included as a driver to begin testing the List class:

```
public static void main(String[] args) {
        List empty = new List();
        List one = new List();
        List multiple = new List();

        one.append(5);
        multiple.append(10);
        multiple.append(20);
        multiple.append(30);


        System.out.println("Empty:"+empty);
        System.out.println("One:"+one);
        System.out.println("Multiple:"+ multiple);

        one.delete(0);
        multiple.delete(1);
        System.out.println("One (upon delete):"+one);
        System.out.println("Multiple (upon delete):"+ multiple);

        //one.insert(600, 1);
        multiple.insert(400, 2);
        System.out.println("One (on insert):"+one);
        System.out.println("Multiple(on insert):"+ multiple);
    }
```

This example, however, is just a starting point. You'll want to add additional tests. Below is a list of some more tests you can write for each of the data structures. Note that this list is not exhaustive and **I expect you to write additional tests** beyond these:

- Make a main in the list class
  - Test inserts for each case outlined in the add and remove functions above, to be sure each section of logic works in these more complex functions
  - Add a test that attempts to remove elements from an empty list
  - Add a test that attempts to remove elements past the size of the list
- Make a main in the stack class
  - Add a test that demonstrates pushes and pops actually reverse the order of the stored elements
  - Add a test that attempts to pop elements from an empty list
- Make a main in the queue class
  - Add a test that demonstrates the FIFO ordering principle of queues
  - Add a test that attempts to dequeue elements from an empty queue

Modification History

Minor edits by Johnny Lin, November 2015.