Course: CSC 115 (J)

Lab 06: Implementing Abstract Data Types (Stack, Queue) using Linked List

Preparation

Make sure you understand the basic structure of a linked list, as discussed in class and implemented in Lab 5.

Objectives

You will be using a linked list data structure to implement the stack and queue abstract data types (ADTs) discussed in class.

During lab 6, you will:

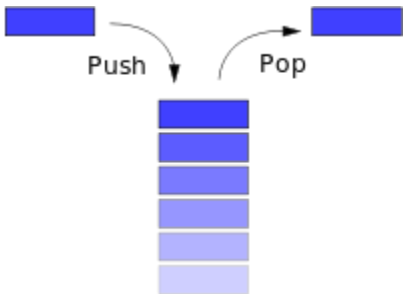
- Write a LLStack class that implements the Stack interface using a linked list.
- Write a LLQueue class that implements the Queue interface using a linked list.
- Come to understand the different behaviors the stack and queue ADTs provide

Background

Stack

A stack is an area of memory that stores and retrieves objects in a specific order. For instance, it can be used to hold all local variables and parameters used by any function, and remember the order in which functions are called so that function returns occur correctly.

The Stack stores objects in last-in-first-out (LIFO) order. The insertion operation in stack is typically known as PUSH, and the operation of removal of an object from the stack is known as POP. Insertion and deletion take place at one end (top) of the list. Best way to implement a stack is to use a linked list.



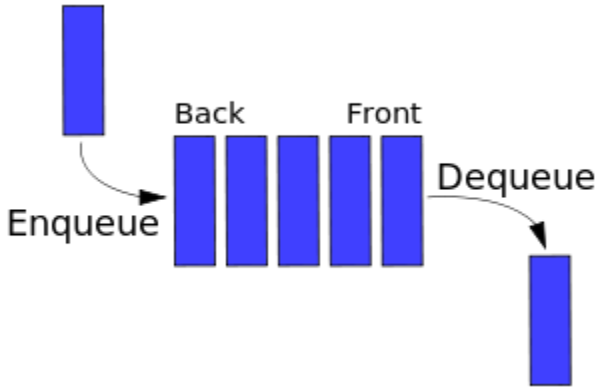
[http://en.wikipedia.org/wiki/Stack]

Typical representation of a Stack

Queue

Queue is an abstract data type in which the objects in the collection are kept in order; the addition of entities occurs at the rear terminal position and removal of entities from the front terminal position. The structure is known as First-In-First-Out (FIFO) data structure. The insertion operation in queue is typically known as Enqueue, and the operation of removal of an object from the queue is known as Dequeue.

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.



[http://en.wikipedia.org/wiki/Stack]

Typical representation of a Queue

Part I – Implementing a Stack

1. Download *Node.java* and *Stack.java* from connex.
2. Download and complete *LLStack.java* so that it implements the interface provided in *Stack.java* using a linked list and includes the following features:
    - A constructor that initializes an empty linked list.
    - A *toString()* method that returns the data as a single *String*.
    - Fill in each method specified in the *Stack* interface.
    - Make sure to compile and test your *LLStack* as you complete each method. You can use do this by un-commenting the code provided in the main method, or by writing your own test operations.

Part II – Implementing a Queue

1. Download *Queue.java* from connex.
2. Download and complete *LLQueue.java* so that it implements the interface provided in *Queue.java* using a linked list and includes the following features:
    - A constructor that initializes an empty linked list. Note that implementing a queue efficiently with a linked list requires us to keep track of both the head and tail of the list.

- A *toString()* method that returns the data as a single String.
- Fill in each method specified in the *Queue* interface.
- Make sure to compile and test your *LLQueue* as you complete each method. You can use do this by un-commenting the code provided in the main method, or by writing your own test operations.
- Use the *LLStack* you wrote in Part-I as inspiration. Two of the methods in *LLQueue* will be exactly the same, one will require a minor change, and the fourth will be quite different.

Part III - Error Checking

Up to this point, we have assumed a benevolent user who will never try to peek/remove from an empty data structure. Change your code in both *LLStack.java* and *LLQueue.java* so that it does not crash if the user tries to peek/pop/dequeue from an empty list, but instead prints out a message on the screen informing the user of their error.

Have fun