July 26, 2010

Using a Linked List to create a Stack and Queue

We've already demonstrated how to make a Linked List, and a stack using Nodes, so let's take this a step further and make a Stack using a Linked List.

```
class Stack<T> {
	LinkedList<T> elements;


	public Stack() { elements = new LinkedList<T>(); }
	public boolean isEmpty() { return elements.size() == 0; }
	public boolean isFull() { return false; }
	public int size() { return elements.size(); }
	public void push(T data) { elements.insert(0, data); }
	public T pop() { return elements.remove(0); }
	public T peak() { return elements.get(0); }
}
```

Likewise, we can create a similar structure based around the Queue. Probably the biggest drawback to this approach is that if the user encounters an error, all of the error messages will appear to come from a LinkedList, which will probably confuse them.

Doubly Linked Lists

Let's review the basic interpretation of a Node:

- A data field.
- A node reference pointing to the next data node (or this is null if there is no next data field)

A double linked list requires an additional field to the idea of a Node: one node reference pointing to the element that should go before this node, or null if there is no previous node.
The major downside to a doubly linked list is that insertions and deletions (which are already complicated in the singly linked list implementation) are now even more complicated.

The upside to a doubly linked list is speed. In addition to having a Node link always point to the front of the list (or “head”), there is also a Node link pointing to the back of the list (or “tail”). If you have a Nelements (where N is a very large number) in a singly linked list, it takes N operations to get the value of the last node. In a doubly linked list, the operation time is O(1) to grab the last element.
For most algorithms requiring a set of data that grows, the operations of insertion and removal happen on the end extremities. By having two variables “head” and “tail”, we can make insertion and removal on the list ends have a runtime of O(1). For insertion and removal operations that happen somewhere in the middle of this list, this still requires an O(N) algorithm.

Circular Doubly Linked Lists

In a Circular Doubly Linked List, the first node's previous field (typically set to null) is set to the last node. Likewise, the last node's next field (typically null) is set to the first node. The result is a list with no definitive head or tail.

Node List

A Node List is yet another variation of a LinkedList designed for speed. Rather than working with positions and indexes into a list, we work directly with the nodes themselves. The advantage of a Node list is the end user gets a dramatic speedup when using a NodeList. The downside is that the end user has to work with nodes of data, and most users don't know (and probably don't care) what a node really is. The Node List is probably the fastest and the most memory efficient of all the node-based list structures. Unfortunately, it puts a dramatic learning curve burden on the user, which means they are less likely to use this data structure. People will not use what they cannot predict happens when they do use it.

```
class Node<T>:
    Node<T> prev
    Node<T> next
    T       data

    Node(Node<T> prev, Node<T> next, T data):
        this.prev = prev
        this.next = next
        this.data = data

    T element():
        if prev == null && next == null:
            print "Element does not exist in list."
            exit
        return data;

    Node<T> getNext():
        return next

    Node<T> getPrev():
        return prev

    void setNext(Node<T> next):
        this.next = next

    void setPrev(Node<T> prev):
        this.prev = prev

    void setElement(T data):
        this.data = data

class NodeList<T>:
    int size
    Node<T> head
    Node<T> tail
```

```
  NodeList():
          size = 0
          head = new Node<T>(null,null,null)
          prev = new Node<T>(null,null,null)
          tail.setPrev(head)
          head.setNext(null)

    // Reporting Methods (Page 232 and 233 in the textbook)
    Node<T> first()
    Node<T> last()

    int size()
    boolean isEmpty()
    prev(p)
    next(p)

    // Update Methods
    set(p,e)
    addFirst(e)
    addLast(e)
    addBefore(p)
    addAfter(p,e)
    remove(p)
```

Array Lists

An ArrayList is similar with respect to the LinkedList. They both do all the same things. You can insert, remove, get and set with both. They both provide a size method and a “isEmpty” method. Whereas a LinkedList is implemented with an internal Node class, ArrayList is implemented with a standard array. To the outside user, there is no difference between the two. On the inside, there is a major difference in how the two work.

The ArrayList has two internal memory values: a array_size number, which keeps track of the array length, and a size number, which keeps track of how many elements the user thinks is in the array. One major difference between the ArrayList and the LinkedList is that when data is removed from a LinkedList, the memory which held the data is freed and released back to the computer for other programs to use. In an ArrayList, when data is removed, the memory is still being retained by the ArrayList object. The only way to free the memory from an ArrayList is to destroy the whole object.

ArrayList are very fast, but use lots of memory, often needlessly. LinkedList are very slow, but are very memory efficient.

An ArrayList always starts with a size of 0 (from the user's perspective) and an array_size of 2 (from the object's perspective). Any time data is inserted into the ArrayList beyond the number recorded in array_size, this procedure is always performed:

1. Double the value in array_size.
2. Declare a new array of length array_size.
3. Move all of the values in the elements array to this new array.

4. Point the elements pointer to this new array.
5. Java's Garbage Collection will destroy the old elements array automatically.

Imagine that we have a list that is defined this way:

```
ArrayList l = new ArrayList<Character>();
```

User Perception	Actual	
Empty list		_,_
l.insert('A',0)	A	A,_
l.insert('B',1)	A,B	A,B
l.insert('C',2)	A,B,C	A,B,C,_
l.insert('D',3)	A,B,C,D	A,B,C,D
l.insert('E',4)	A,B,C,D,E	A,B,C,D,E,_,_,_

The ArrayList requires three fields in order to work: a “array_size” integer to control the length of the internal array, a size element to keep track of the array size from the user's perspective, and an elements array.

```
class ArrayList<T>:
    int size
    int array_size
    T elements

    ArrayList():
        size = 0
        array_size = 2
        elements = (T[]) new Object[array_size]

    resizeArray():
        array_size = array_size * 2
        T newArray = (T[]) new Object[array_size]
        for (int i = 0; i < size; i++)
            newArray[i] = elements[i]
        elements = newArray

    boolean isEmpty():
        return size == 0

    int size():
        return size

    void set(T data, int pos):
        if pos < 0 || pos >= size:
            print "Error: Array Out Of Bounds: "+pos
            exit
        elements[pos] = data

    int get(int pos):
        if pos < 0 || pos >= size:
            print "Error: Array Out Of Bounds: "+pos
            exit
        return element[pos]

    int remove(int pos):
        if pos < 0 || pos >= size:
            print "Error: Array Out Of Bounds: "+pos
            exit

        T temp = elements[pos]
        for (int i = pos; i < size-1; i++)
            elements[i] = elements[i+1]
        size = size - 1
        return temp

    void insert(T data, int pos):
        if pos < 0 || pos > size:
            print "Error: Array Out Of Bounds: "+pos
            exit
        if size == array_size:
            resizeArray()
        for (int i = size-1; i >= pos; i--)
            elements[i+1] = elements[i]
        elements[pos] = data
        size = size + 1
```