Linked Lists

Alexander Ferworn

Unrelated Facts Worth Remembering

- Ensure training is training is hard, realistic and of an intensity and duration expected in operations (CFP 309(3) Chap. 17, Pg.17.7, Para. 4E)
- Go Soothingly in the grease mud, as there lurks the skid demon. (English translation of a Japanese traffic sign)
- Profanity is the one language that all programmers understand (anon)

Table of Contents

1 Introduction

We have discussed arrays and how they are implemented. This document describes another means of accumulating a list—a linked list. It is interesting that Java actually provides native support for linked lists in the vector class, but we will concern ourselves with implementing them ourselves.

2 Linked List

You are all familiar with the use of arrays to keep a set of elements in memory. There are many good things about arrays--they allow us to randomly access cells, that is vital in some applications, addressing cells is also easy since each cell has a nice numeric address and each comes after the other. There are applications where we need dynamic space allocation and that is why we introduce you to linked lists.

Here are few reasons for why arrays might not meet our storage needs perfectly:

1. In general, ArrayLists aside, arrays are static, they don't grow when you want them to magically.
2. If you need to insert in the middle of an array, you must first make room for the new element. For instance, if the new value needs to go into position 10, elements in

position 10 and on have to move to make room for it. Removing values cause the same sort of problem.

3. There is also no concept of physically removing a cell, if you allocate 100 cells, you have 100 cells forever. We may be working on an application where we can't predict the number of cells that we need.
4. There is a concern about the arrays space utilization. To be sure that we have enough room, we sometimes need to allocate a large number of cells. This could lead to programs that cause excessive disk cashing. In such cases, the operating system can't keep the whole program in memory at one time. Every time a portion of the program is addressed that is not in memory, the system needs to read it from the disk and unload a different portion of the program that is in memory. Of course, none of this activity has anything to do with what the program is trying to do. I guess, it is like the program spinning its wheels once in a while.
5. To make sure that operations like insertion or deletion are performed efficiently, we end up developing algorithms that are complex and harder to follow. The circular queue is a good example, it uses the array space efficiently, but it is harder to follow than the simple version.

Linked lists can be implemented with arrays, but typically in languages like Java that support dynamic allocation, they are not. You can create cells as you need them, and can remove them when you wish. But, you don't have a nice numeric address like you do with arrays. The address of cells are NOT in sequence nicely like they are in arrays, in fact, we don't refer to them as cells, we call them nodes.

You have to get each element of the sequence to remember the address of the next. In the object oriented paradigm, we say that each node keeps a reference to the next node. sometimes, we design our nodes to even keep a reference to a previous node, we refer to linked lists with such nodes as doublely-linked lists. Other times we design the linked list, so that the last node points to the first one. Such linked lists are referred to as circular linked lists. Linked lists are a tremendous asset. We add nodes as we need to, we can put a node between two other nodes by changing their references.

In java, we can delete a node by simply making their previous node reference their next node, if we have a doublely linked list, their next node will now reference their previous node also. Java's Garbage Collector will collect any space that is left unreferenced. So, by simply not referencing a node we have effectively deleted it.

Picture a node as an object that has two parts. One part containing data, as many pieces as needed, and one or more references to other nodes. We will first look at the definition of a singly linked list. Notice that it is as if a node can be inside another node. Such recursive definitions are common for linked lists.

Also notice the constructor, it is designed to allow for setting both the data part of the node and the reference to the next node. The last thing you should notice lack of a

qualifier like private or public. This simply means that Node is accessible in the package that it sits in. So, for example if we design a Node class as part of the stringQueue package we get to use it to create nodes in our dynamicStringQueue class which is in the same package. More about this later.

```
class Node
{
      dataType element;
      Node next;
      Node (Node n, dataType e)
      {
            next=n;
            element=e;
      }
}
```

Here is a graphical image of what the linked lists would look like, if we were simply holding some numeric values:

10 ---> 20 ---> 30 ---> 40 --->||

Here is the definition of a doublely linked list, note that we have an additional reference to a node (prev).

```
class Node
{
      dataType element;
      Node next;
      Node prev;
      Node (Node n,Node p, dataType e)
      {
            next=n;
            prev=p;
            element=e;
      }
}
```

Here is a graphical image of what a doublely linked list would look like, if we were simply holding some numeric values:

||<--- 10 <---> 20 <---> 30 <---> 40 --->||

When implementing any variation of linked lists, you must have a reference to at least one node in the list.

```
Node head=null;
```

In this statement we are declaring a variable head which is set to null. null is a value in java that means empty and we use it when setting or checking reference variables. Consider head as a variable that holds a reference to an object of class Node, of course, it is null at this point.

```
head=new Node(null,10);
```

In this statement we are making head reference a newly created node that will contain 10 with the next reference as null. of course, we are assuming that our dataType is int and that we are using the first Node class shown above. It should be obvious that we can't have two classes with the same name, so, if we need two node classes in a package, we need to name them differently.

Consider the following list of elements with head referencing the node that contains 10:

10 ---> 20 ---> 30 ---> 40 --->||
head

Let's examine a few statements and see what happens to the list:

```
head.element=12;
```

12 ---> 20 ---> 30 ---> 40 --->||
head

```
head.next.element=25;
```

12 ---> 25 ---> 30 ---> 40 --->||
head

Create a new node referenced by temp. This node will contain 17 with its next field set to what head's next is. Sorry about the crude graphics.

```
Node temp=new(head.next,17);
```

```
  12 ---> 25 ---> 30 ---> 40 --->||
 head      ^
           |
          17
         temp
```

The following statement, effectively, adds temp's node to the list. `head.next=temp;`

```
12 ---> 17 ---> 25 ---> 30 ---> 40 --->||
```

head temp

```
temp.next=temp.next.next;
//remove the node that contains 25!
```

```
12 ---> 17 ---> 30 ---> 40 --->||
head   temp
```

3 Abstract Data Types

An abstract data type (ADT) is a model where a data type is defined by its behavior (semantics) with reference to the values and operations of and on the data of this type. An ADT is agnostic as to how it is implemented as long as its semantics are respected. An array is an example of an ADT with the operations create (an array), get (a value from the array) and set (a value in the array) defined on data that has a single name and type.

4 The Stack

A stack is an ADT with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added.

Stacks are referred to as "last in, first out" (LIFO) ADTs. The term LIFO stems from the fact that, using these operations, each element "popped off" a stack in series of pushes and pops is the last (most recent) element that was "pushed onto" the stack.

4.1 Array Implementation

```
public class Stack
{
	private int TOS=-1;
	private String[] stack_array;
	private int max_size;

	public Stack(int size)
	{
		stack_array = new String[size];
		max_size = size;
	}

	public String topval()
	{
		if(TOS < 0) return "NULL";
		return stack_array[TOS];
	}

	public String pop()
	{
		if(TOS < 0) return "NULL";
		System.out.println("pop: " + stack_array[TOS]);
		return stack_array[TOS--];
	}
```

```
	public boolean push(String item)
	{
		if(TOS == max_size) return false;
		TOS++;
		stack_array[TOS] = item;
		System.out.println("push: " + item);
		return true;
	}
}
```

4.2 Linked List Implementation

4.2.1 Making a "node"

```
public class SList
{
   int val;
   SList next;
}
```

4.2.2 Implementing the defined operations

```
public class LStack
{
    private SList TOS;

    public void Stack(int size)
    {
        TOS = null;
    }

    public SList topval()
    {
        return TOS;
    }

    public int pop()
    {
        int val;
        val = TOS.val;
        TOS = TOS.next;
        return val;
    }

    public void push(int item)
    {
        SList node = new SList();
        node.val = item;
        node.next = TOS;
        TOS = node;
        System.out.println("push: " + TOS.val);
    }
}
```

5 The Queue

A queue is and ADT in which the entities (nodes) in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position, known as "enqueuer", and removal of entities from the front terminal

position, known as "dequeuer". This makes the queue a First-In-First-Out (FIFO) data structure.

In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a peek or front operation is also entered, returning the value of the front element without dequeuing it.

5.1 Array Implementation of a Queue

It is really hard to implement a queue with an array (not impossible), why?

5.2 Linked List Implementation of a Queue

```
public class Queue_List{
	int val;
	String name;
	Queue_List next;
	Queue_List prev;
}

public class Queue{
	Queue_List first;
	Queue_List last;
	String queue_name;

	public Queue(String name){
		// Make a new queue
		queue_name=name;
		first= null;
		last= null;
	}

	public void join(int val, String name){
		Queue_List temp = new Queue_List();
		temp.val = val;
		temp.name = name;

		if(first == null){
			first = temp;
			last = temp;
			temp.next = null;
			temp.prev = null;
		}
		else{
			temp.prev = last;
			temp.next = null;
			last.next = temp;
			last = temp;
		}
	}
```

```
	public Queue_List leave(){
		Queue_List temp = first;
		if(temp == null) return null; //empty queue
		first = first.next;
		return temp;
	}

	public void dump_queue(){
	    // note: dumping a queue is not normal, why?
		Queue_List node = first;
		System.out.println("The queue " +
			queue_name + " looks like this...");
		System.out.print("first->");
		while(node!= null)
		{
			System.out.print("<-(" + node.val +
				"," + node.name + ")" + "->");
			node=node.next;
		}
		System.out.print("<-last\n");
	}
}

public class Queue_Driver{
	public static void main(String[] args){
		Queue a_queue = new Queue("a_queue");
		for(int i=0;i<5;i++){
			a_queue.join(i,"name " + i);
			a_queue.dump_queue();
		}
		for(int i=0;i<6;i++){
			Queue_List temp = a_queue.leave();
			if (temp!=null)
				System.out.println
				("The following has been dequeued: " +
					temp.val + "," + temp.name);
			else
				System.out.println("Nothing was dequeued.");
			a_queue.dump_queue();
		}
	}
}
```

6 Dynamic String Queue

As you may be able to tell, you gain incredible flexibility when you implement a queue with a linked list. Let us consider dynamicStringQueue class. First of all, it uses a doublely linked list which means the second Node class shown above is utilized. Note that two reference variables are used to keep track of both front of the list and the back of the list, first and last. This will facilitate easy insertion and deletions for our queue. Both variables start as null so the constructor doesn't have any code. Just checking last or first against null should determine if if queue is empty. You will find that full always returns

false as you can't get full in this implementation. Lets take a careful look at enqueue and dequeue.

```
public void enqueue(String x)
{
      if (first==null)
      {
            first=new Node(null,null,x);
            last=first;
      }
      else
      {
            last.next=new Node(null,last,x);
            last=last.next;
      }
}
```

We first check for first being null, that helps us figure out if we need both first and last set, or we only need to add a new node to the end of the list. The statement first=new Node(null,null,x); sets first to reference a newly created node. This new node will contain the new value in it with both its prev and next node references set to null. Note that last gets to reference the same node as first does.

When there are elements already in the list, we create a new node that we place after last. This new node will have a pre node reference that will be set to last. last=last.next makes last reference the new node. Consider the following list:

```
||<--- jones <---> fair <---> mary <---> leo --->||
       first                            last
```

Here is what happens when we enqueue mark:

```
||<--- jones <---> fair <---> mary <---> leo <---> mark --->||
       first                                        last
```

Let use consider dequeue:

```
public String dequeue()
{
      Node old_first=first;
      if (first == last)
      {
            first= null;
            last= null;
      }
      else
      {
            first=first.next;
            first.prev=null;
```

```
            }
            return old_first.element;
        }
```

Node old_first=first; allows us to remember the reference to the node in front of the queue, since the manipulations that follow will effectively take that node out of the list. If the node in front of the queue is the last node in the queue, we set both first and last to null. Otherwise, we just make the node right after first, the first node by saying first=first.next;. We then use the statement first.prev=null; to set this newly appointed first node's prev variable to null. Now only old_first references the old first node. Once we return the element in the node referenced by old_first this node can be garbage collected.