The Java Vector class
[available in the* java.util *package]

Note on the Java Utilities Package.

The ***java.util*** is an important package contains about 20 utility classes. It includes classes that include time and date utilities:

GregorianCalendar	[Earth only. Not for Mars]
Date	[Earth only. Not for Mars]
TimeZone	[Earth only. Not for Mars]

It also has general ***container*** classes, such as

Vector	[topic covered; see text, pp 208-209]
Hashtable	[topic covered later]
Dictionary	[topic covered later]
Stack	[topic covered]
ArrayList	[similar to Vector; see text, pp 208-209]

It also contains:

Random

a class used for generating random numbers.

Java's **Vector** has the following ***[generic]*** methods:

int	size()
boolean	isEmpty ()
object	elementAt (int r)
void	setElementAt (object o, int r) *//replaces object at rank r with o*
void	insertElementAt (object o, int r) *//inserts object o at rank r*
void	removeElementAt (int r)

//and some other useful methods

These are similar, but not completely the same as those exemplified in the text.

However, Java's ***insertElementAt()*** is essentially the same as ***insertAtRank()*** in the textbook.

Suppose now, that we want to use some Vector objects in a main() program:

```
import java.util.Vector;

class TestVector {

public static void main() (String [] args) {
    Vector      v1, v2, v3;
    int         capacity = 17;
    int         capacityIncrement = 8;

//Three versions of constructor available

    v1 = new Vector (capacity, capacityIncrement);
                    //vector has initial capacity 17
                    //on overflow, capacity increases by 8
          //BAD CHOICE

    v2 = new Vector (capacity);
                  //vector has initial capacity 17
                 //on overflow, capacity doubles, the default
          //GOOD CHOICE

    v3 = new Vector ();
                  //vector has default initial capacity, 10
                 //on overflow, capacity doubles, the default
          //GOOD CHOICE
    - - -;
}}
```

On overflow, whether it is better to:
Have vector capacity increase by an increment k, or
Have vector capacity double ?

On the surface it might look as if it is more efficient to have an incremental increase k on overflow.

But this is false. Do NOT use the increment capacity option, except where only one, or maybe two, extensions are anticipated.

First, we have seen that allowing for doubling, when inserting at the top of the vector, is **O**(n).

We will now show that use of a fixed increment k is **O**(n^2) [and **Omega**(n^2)] for inserting at the top of the vector.

PROOF that a Vector increment extension method is O(n^2)

Suppose the initial capacity is C, and the increment is k. And to exemplify, we will suppose C = 10 and k = 3.

Suppose also that n is the number of insertions into the vector that just manages to trigger m extensions of the vector. As an example, take n = 20.

STEP 1

n = 20 items = 10 + 3 + 3 + 3 +1 items inserted

clearly causes:

1 (at 10) + 1 + 1 + 1 = 4 = m extensions

From this we can see that

$$
\begin{aligned}
n = \; 20 &= 10 + 3*3 + 1 = 10 + (m-1)*3 + 1 \\
&= C + (m-1)k + 1
\end{aligned}
$$

So m-1 = (n- 1 - C)/k [= nearest integer below (n - C)/k for any n]

STEP 2:

Now calculate the number of overflow copying operations as we add items to the vector.

Value of n	*copied elements*	*copied elements*
For n = 10	0	0
For n = 11:	10	C + 0
For n = 14:	13	C + k
For n = 17:	16	C + 2k
For n = 20:	19	C + 3k or [C + (m-1)k]

Total copying operations:

10 + 13 + 16 + 19   mC + k(1 + 2 + … + (m-1)]

= 4*10 + 3(1 + 2 + (4-1))

= 58

= mC + k[(m - 1)m/2]

= m(C - 0.5k) + 0.5km^2

but m -1 = (n -1 -C)/k = 3

So Total copying operation, or f(n) is:

f(n)= [(n -1 - C + k)/k][C - 0.5k] + 0.5k[(n -1 -C + k)/k]^2

= [12/3][8.5] + 0.5* 3[4* 4]

so f(n) = 4*8.5 + 24 = 58 for C = 10, n = 20, k =3

f(n) is clearly of the form: f(n) = an^2 + bn + c

so that f(n) is **O**(n^2) [and **Omega**(n^2), see page 121] for insertion at the top of the vector.

List ADT implementation using doubly-linked List

Suppose an ADT just like a vector, except that we do not use rank to identify the location of an element in the vector. Instead imagine each data element in the sequence S as having a ***fixed physical position***, in practice a memory address. For example, suppose the sequence S below, with nodes as shown **before update.** Notice the **header** and **trailer** nodes.

prev	*next*	*element*	***node position or address***	
null	**f**	**null**	h	***//header node***
h	q	Helium	f	
f	a	Lithium	q	
q	g	Berylium	a	
a	m	Boron	g	
g	v	Karbon	m	
t	b	Nitrogen	v	
v	t	Fluorine	b	
b	**null**	**null**	t	**//trailer node**

Now consider some methods that manipulate this ADT.

1. Position retrieval methods: *[see also text, p. 211]*

first () returns f
//what's the first position in the sequence?

last () returns b
//what's the last position in the sequence?

prev(m) returns g
//what's the position in the sequence before position m?

next (v) returns v
//what's the position in the sequence after position m?

List Update Methods *[see also text, p. 211]*

replace (m, "Carbon")	Karbon returned, S updated
insertFirst("Hydrogen")	Hydrogen is new first in S returns d
insertLast ("Neon")	Hydrogen new last in S returns k
insertAfter(v, "Oxygen")	Oxygen inserted after Nitrogen returns p
insertBefore(q, "WMDgen")	WMG inserted before Lithium returns z
remove (z)	returns WMDgen

Simple object retrieval methods *//Not in text book*

elementAt (m) returns Karbon
//What's the object at position m?

nextElement (m) returns Nitrogen
//What's the object after position m?

prevElement(m) returns Boron
//What's the object previous to position m?

firstElement () returns Helium
//What's the first element of the sequence?

lastElement() returns Fluorine
//What's the last element of the sequence?

Note: If a program manipulating a List object keeps track of the positions for data items inserted, there will be no need for these methods, except for ***elementAt (p).***

New S, after update:

node			*node position or address*	
prev	*next*	*element*		
null	**d**	**null**	h	***//header node***
h	***q***	***Hydrogen***	***d***	*//new*
d	q	Helium	f	
f	a	Lithium	q	
q	g	Berylium	a	
a	m	Boron	g	
g	v	Carbon	m	*//replaced*
p	p	Nitrogen	v	
v	***b***	***Oxygen***	***p***	*//new*
p	k	Fluorine	b	
b	***t***	***Neon***	***k***	*//new*
k	**null**	**null**	t	***//trailer node***

Notice that the [chemical] elements are maintained in rising Atomic Numbr or Atomic Weight (lightest first)

Exceptions

Many of the methods above will throw an exception if the position used as a parameter in the method is invalid.

Typical reasons for a position p being invalid are:

p is null

p has been deleted from list

p is in a different list

We attempt to access the position previous to p when p is the first position in the list

We attempt to access the position after p when p is the last position in the list.

So just about every method that manipulates the list has to check out if p is invalid in any of the above ways. In the textbook, in the implementation of NodeList list class, a workhorse method

checkPosition (position p) { }

or checkPosition (DNode p) { }

is used for this purpose.