**Array List ADT**

Index vs Rank: Index is size-1

Properties:

Main methods:  
  
For get,set, and remove, must just check index is within size-1

get(i); get value at index i

set (i,e): set value at index i

add(i,e):: add value at index i.   
-If the array is not full, this works  
-the values at that index and after are moved one to the right   
-This is why the adding is O(1) if you add to the end, but O(n) if at beginning, and O(n-i-1) if at somewhere between (I is index)  
-but what if the array is full? We replace the array with a bigger one.

remove(i): remove value at index i   
-Move all values on the right, one to the left. That’s why it is normally O(n) if at beginning, O(1) if at end, and O(n-i-1) at between

Support:

size(): get size  
isEmpty(): checks if empty

Performance: O(1) for everything except add or remove: O(n) (we look at worst case).  
set(i,e) returns the old value.  
Space: O(n)

In a circular list, add and remove also have O(1) at beginning.

**Growth Strats NBNB**

**Incremental Strategy**: Increase size by constant c.  
**Doubling Strategy**: Double the size

We must compare these by looking at the total time T(n) needed to perform a series of n add(o) operations.  
-We assume we start with array of size 1, that isEmpty  
-The average time of an add operation will be T(n)/n.. Where T(n) is total time.  
  
Note that fixed arrays: You either have too much space, or not enough space.  
  
  
Doubling: You use more space, but is used less frequently  
-When only looking at copying the array to the bigger array, and adding the value. This does not look at creating the new array.  
n + 1 + 2+ … + 2^k = n + 2^(k+1) – 1 = 3n-1  
k = log2(n)  
The average time is O(1) as avg: n/n = O(1)

Incremental: You frequently use this, but use less space.  
we replace the array k=n/c times. Where c is the constant increase.   
Same as doubling, only look at copying and adding value.  
n+c+2c+3c+…+kc … as we copy c times… then 2c times… then 3c times… and add n values  
= n + c(1+2+…+k)  
= n + ck(k+1)/2  
= n + n(n/c+1)/2 = n + (n^2/c + 1)/2 … this line is my own  
Therefore, the avg time is n^2/n = O(n).  
  
Therefore, doubling is better here.

**Adapter Patter**

Use the instance of the class you want to adapt, in the main class.

**Position ADT**

Has a single method:  
element(): gives the element at a given position

Positions are relative.

A position p, associated with element e, does not change even if index of e changes.  
If we remove e, thereby destroying by, it changes.

We can access the index e even if its index changes, using the position p.

In a linked list example with nodes, P would be like a header/tail, but instead called a “pointer/position”, pointing to that one node. (This is not advised to do though).

**Positional List ADT (IN semester test)**

The node in a linked list, is the position of an element.  
Therefore adding and removals is O(1)  
-as when you add, you can get the before and after at constant time.  
-as when you remove, you can get the before and after at constant time.  
-no need to go through whole list as if counting an index

Main Methods:  
P first(): returns position of first element  
P last(): returns position of last element  
P prev(p): returns position before p position  
P next(p): returns position after p position  
-these are not nodes, but positions

Update Methods:  
set(p,e)  
addFirst(e)  
addLast(e)  
addBefore(p,e)  
addAfter(p,e)  
remove(p)  
  
-You can do addFirst/AddLast using addBefore and addAfter

**Iterator Properties**

Abstracts the scanning through a collection of elements.  
Can access and make changes to current element in traversal, and for to next element in traversal.  
Traversal is independent from specific implementation of collection.

Main Methods:  
hasNext() – checks if there is a next element in the list  
next() – traverses to the next element

-Extends the concept of position by adding a traversal capability.  
-Implementation with an array or a singly linked list.

**Iterable classes**

Augment: Stack, Queue, Array List, List, Sequence.  
-Iterator <E> iterator(): returns an iterator over the elements.

**Types of Iterators**

Two Types:

Snapshot: creates a copy of your list, and the iterator works on that copy. Eg Have 3 iterators on 3 different snapshots.

Dynamic: Iterator works on the original list. Eg Have 3 Iterators on same list. **Any** **changes** that are made, which are different to the original list the iterator was created on, an **exception** will be thrown. (This is not the case with the snapshot)

**List Iterator ADT**

This allows you to traverse backwards and forwards, as a uni Iterator cant do this on a doubly linked list.  
  
Methods:

add(e)  
hasNext()  
hasPrevious()  
previous()  
next() – returns element after cursor, and sets cursor to that next element  
set(e)  
remove() – removes the last element returned by next or previous

Variant: **Position Iterators**

Create a positions() methods which returns an Iterable object for the positions.  
ie, there is a next and previous methods for your current positions.

**Sequence ADT**

Provides explicit access to the elements in the list either by **indices** or **positions**.

Sequence ADT is the union of Array List and Positional List ADTs.  
  
generic methods:  
size(), isEmpty()  
  
ArrayList-Based methods:  
get(i), set(i,o) add(i,o), remove(i)  
  
List-Based methods:  
first(), last() …. Ect  
  
Bridge Methods:  
atIndex(i), indexOf(p)  
  
Useful for?  
small databases, generic replacement for stack, queue, array list, or list.

Linked List:  
position-based methods O(1)  
Index Based methods O(n)

**Principal of Locality: Move to Front Heuristic**

List that orders the favorite things in non descending number of times of access.