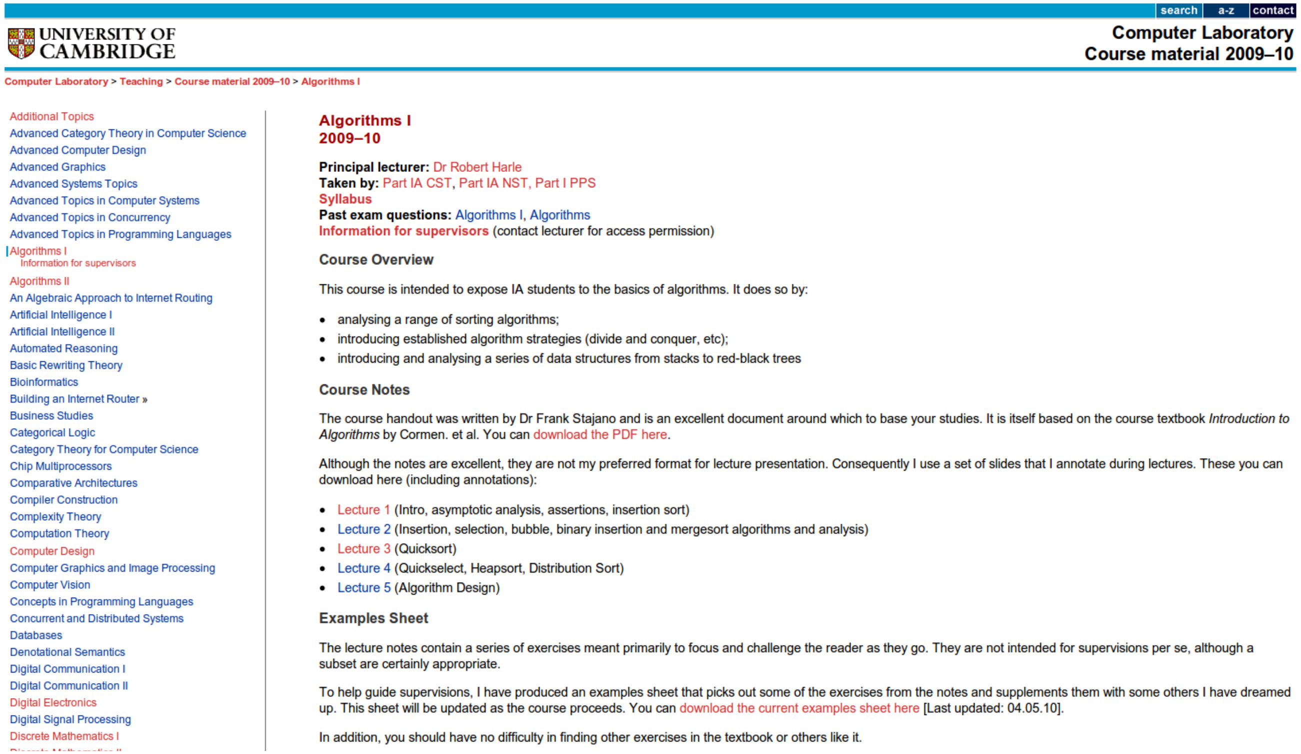
Part III: Data Structures

Aside: Examples Sheet

- If you have been using the examples sheet on the web page, it has now been updated (and will be continually)



Data Structures

- In OOP we saw that there was an advantage in creating classes
  - Allowed us to group primitive types into one entity
  - Allowed us to group together data and the operations allowed on it

- Even without the OOP goodies (inheritance, etc) it is useful to do this for other languages
  - Rename to "data structures"

Why are they in this Course?

- Often we find that data structures provide natural support to parts of an algorithm
  - Think heapsort
- We will be looking at a range of data structures (some that you've already used) from a theoretical standpoint
- We start with a look at how to represent some fairly fundamental data structures

Abstract Data Types

- An ADT is a model of a data structure or type

- It defines the functionality expected of the data structure (but not the implementation)
  - Like a specification of the data structure
  - Maps to the interface notion in Java

- Frees us from implementation details and makes it easy to swap in new algorithms for the operations as they are discovered

- You can make your own of course, but there exists a set of ADTs that you should just know...

ADT 1: List

- A sequence of items
  - add(item i, position p): insert item i into the list in position p.
  - delete(position p): delete the item at p
  - is_empty(): returns true iff the list is empty
  - get(position p): get the item at position p

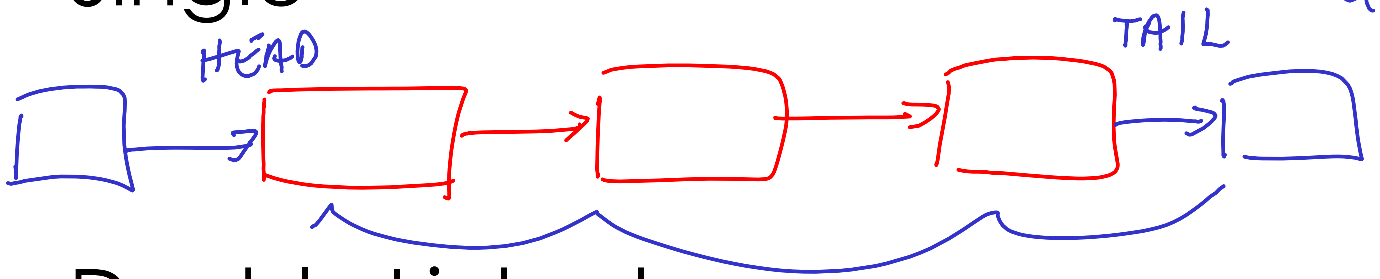
List Types

[unsorted]

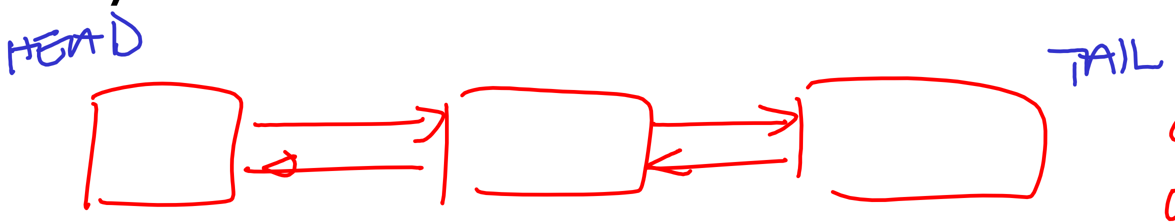
insert $O(1)$ at front

$O(p)$ insertion $O(n)$

- Single



- Doubly-Linked

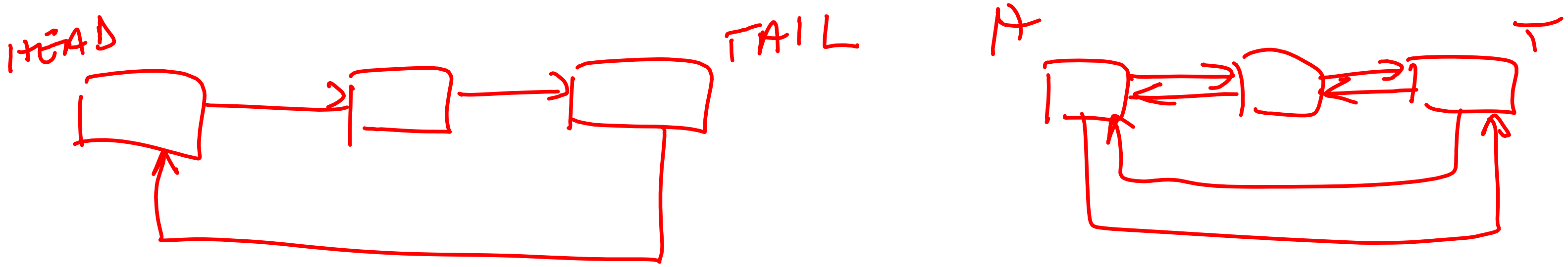


insert

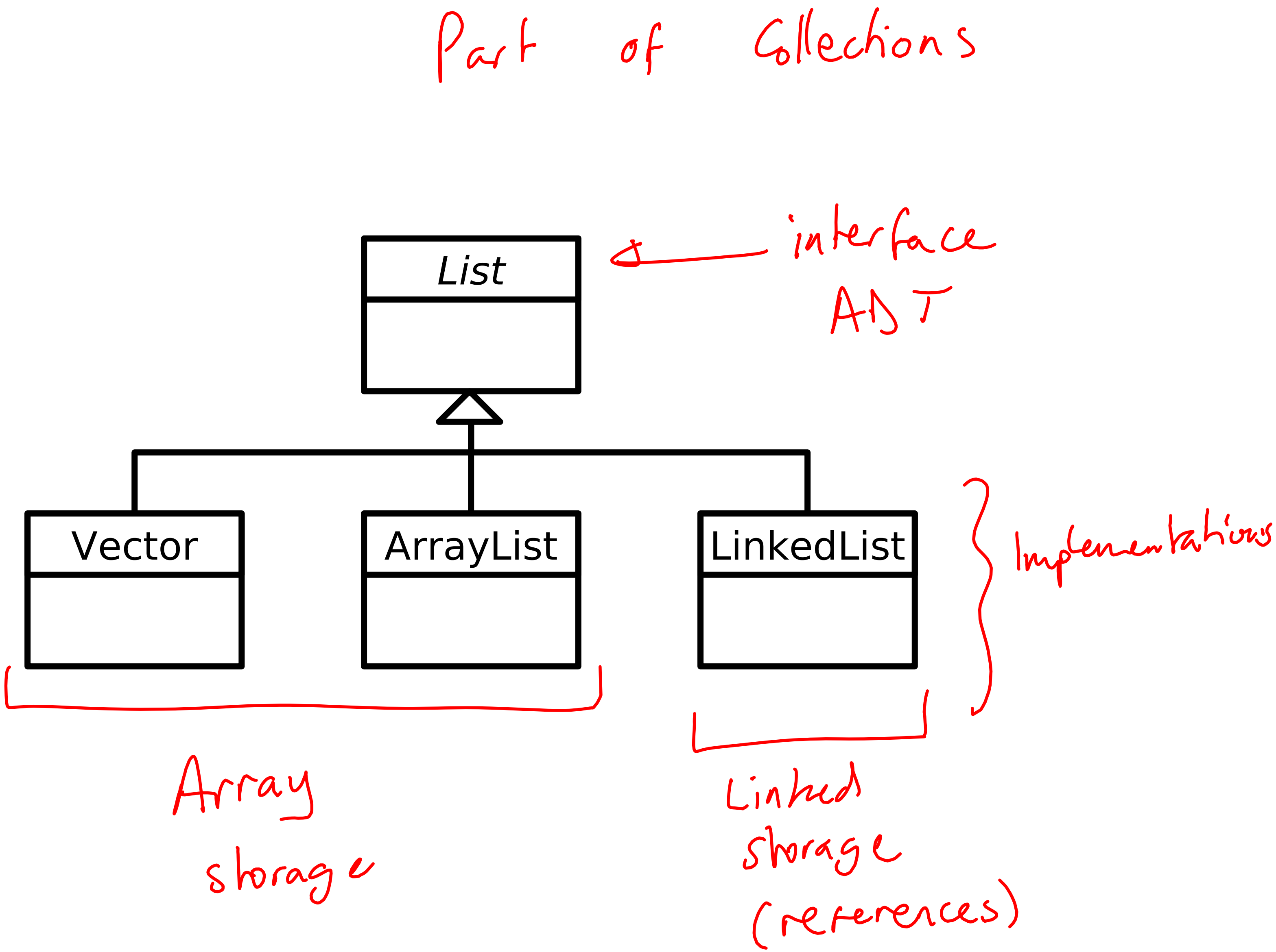
$O(1)$ front + back

$O(\frac{n}{2}) > O(n)$ general insertion

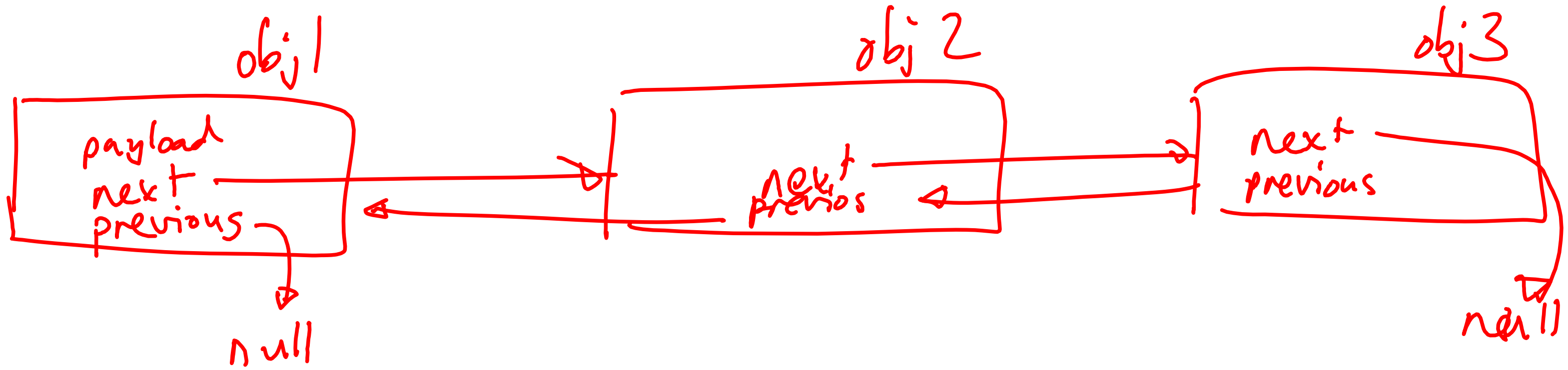
- Circular



Java List Interface



Java Linked List




```
public class MyLinkedList {
    int payload;
    LinkedList next;
    LinkedList previous;
}
```

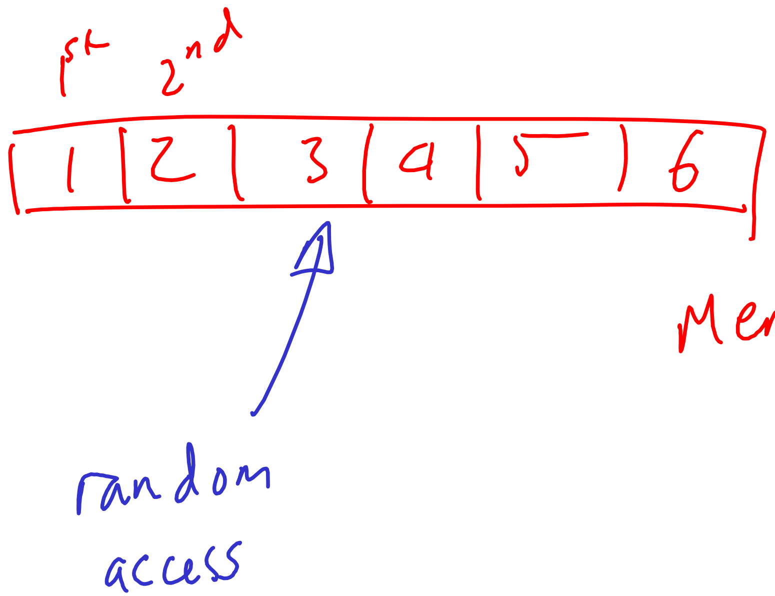
references

LinkedList Costs

- add: Traverse the list to find the position, create object, then insert $O(n)$
  $O(n)$ $O(1)$ $O(1)$
- delete: Traverse the list to find the position, then delete $O(n)$
  $O(n)$ $O(1)$
- isEmpty: Check the head element $O(1)$
- get: Traverse the list to find the element $O(n)$
  $O(n)$

ArrayList and Vector

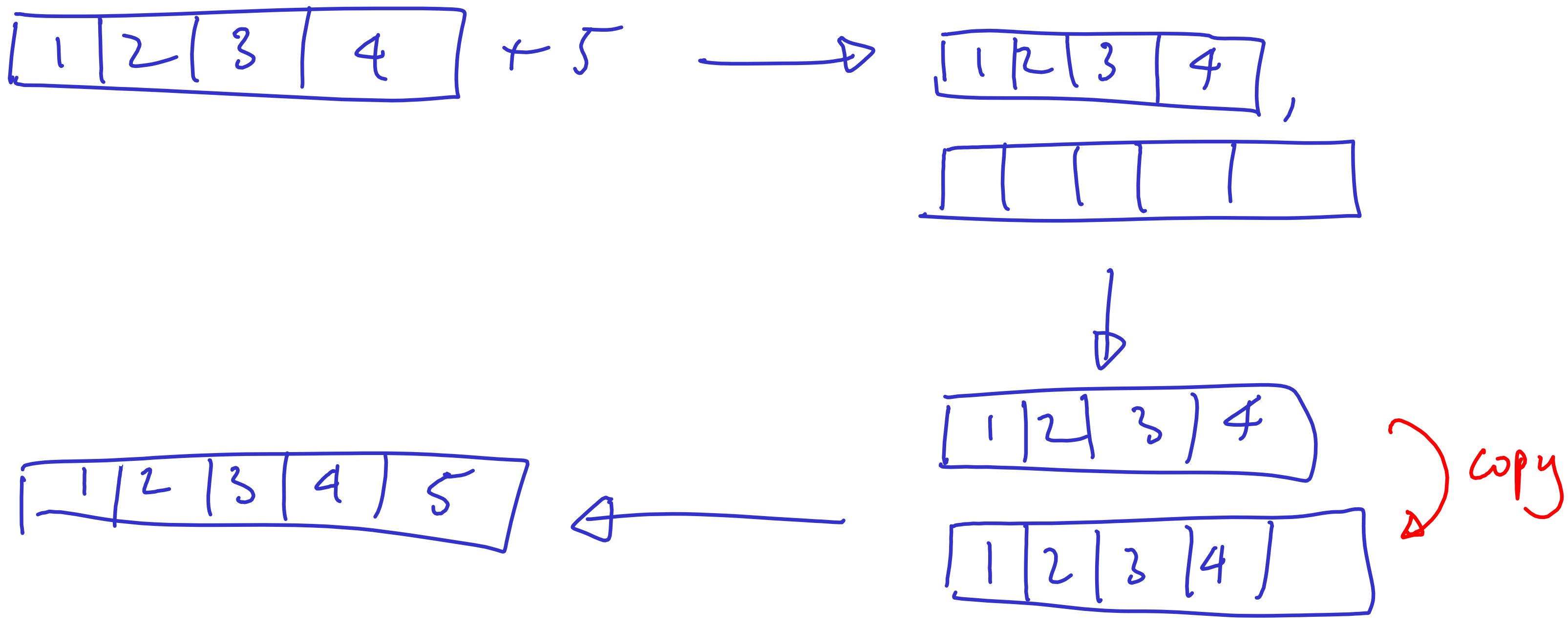
- Think of these as the same thing, but Vector is 'synchronised' and hence slower in normal usage
  - i.e. threadsafe



Memory overhead is zero

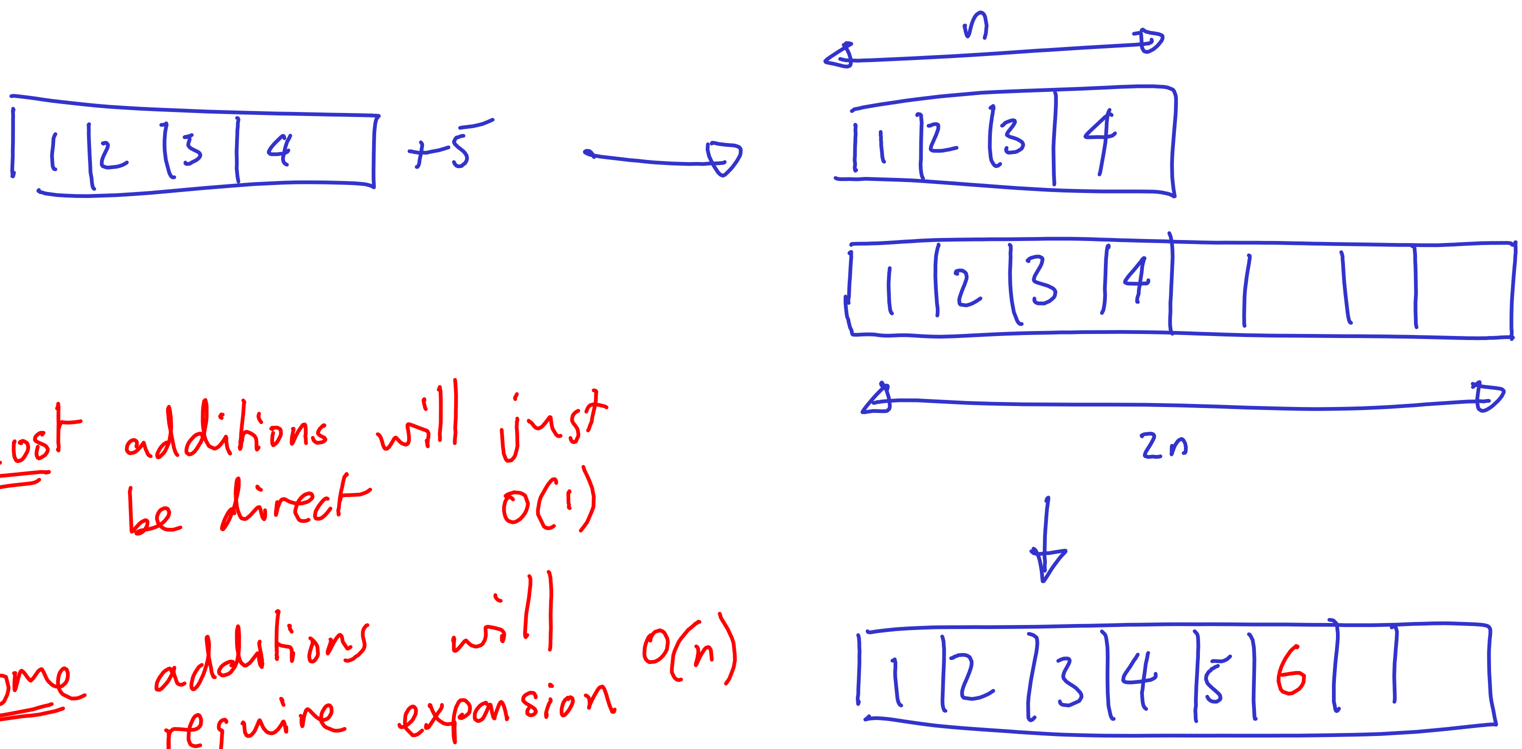
ArrayList: add to end

Increase array size.



Making + filling a new array

ArrayList: add to end



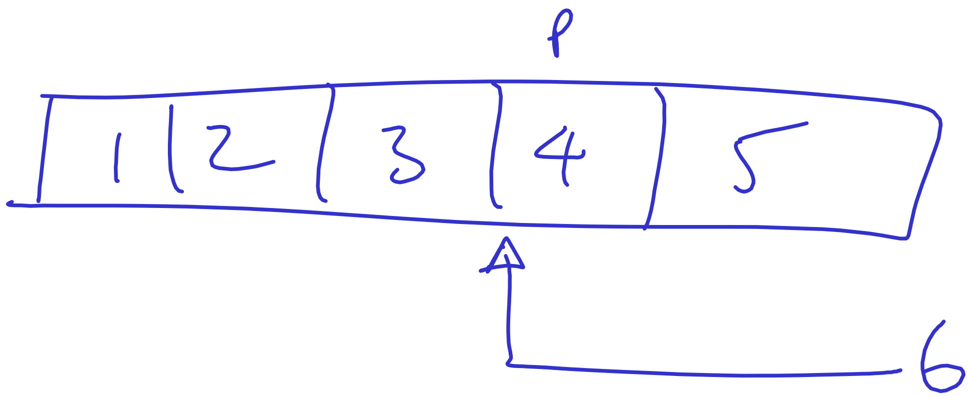
Most additions will just be direct $O(1)$

Some additions will require expansion $O(n)$

Add n elements in $O(n)$ time

$\Rightarrow$ Amortized cost $O(1)$

ArrayList: add to middle

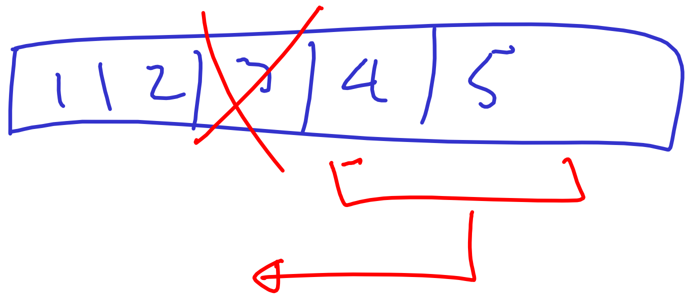


i) There is space $\Rightarrow$ move $O(n-p)$ elements

ii) Create more space, then (i)

ArrayList

- delete



shift up

O(n-p) shifts

- get

O(1) Random access

Which to use?

- Right now your default is probably to use LinkedList in java
- For a general usage, ArrayList is usually more efficient

LinkedList – good for insertion/deletion in a random pos.
– memory overhead + slow search

Arraylist – great for everything except insert in random pos.

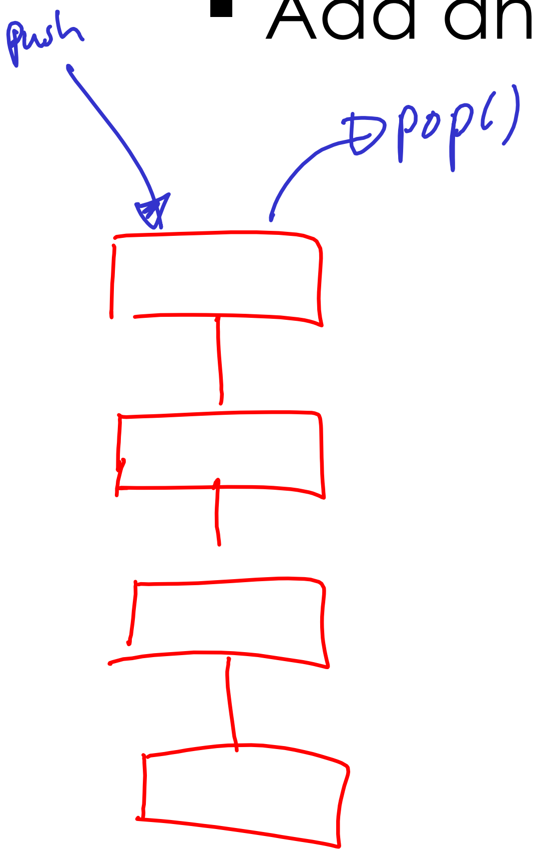
200,000

AL	LL
438ms	1458ms
80ms	

Java arraylist ⇒ 10 items initialise.

ADT 2: Stack

- Analogy: A stack of plates (**LIFO**) Last in First out
  - Add and take from the top

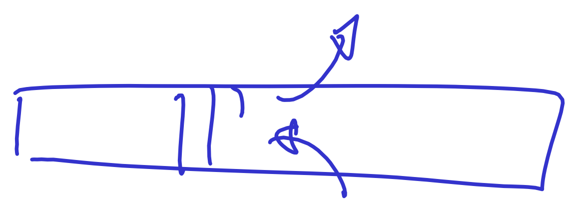


  - push(item i): add i to the top
  - pop(): remove i from the top after returning it
  - top(): get the item on top
  - isEmpty(): return true iff stack is empty

Stack Implementations

- Linked List
  - Everything happens at the head so O(p) is now O(1)
  - Still have a memory overhead associated with each node

- Array
  - Do everything at the tail of the array: good performance characteristics
  - Java's Stack uses Vector as its base



Stack Languages!

- Postscript (language for text and gfx layout on a page: printer speak)

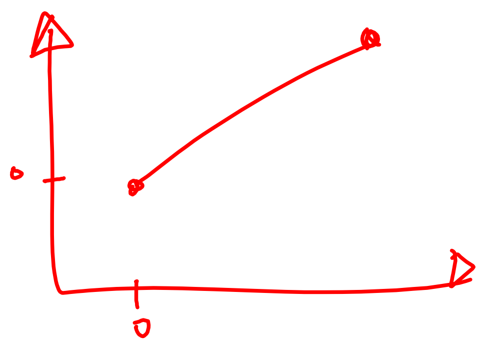
7 9 add 2 mul

Postscript executes	stack	
7	7	
9	7 9	7 9
add	16	
2	16 2	
mul	32	

Reverse polish notation

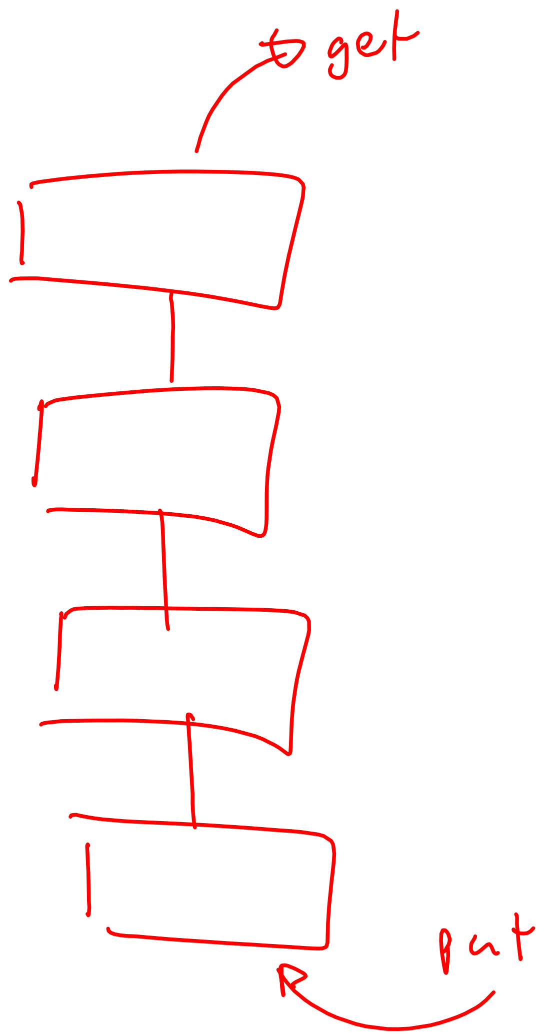
$(7+9) \times 2$

0 0 moveto 20 20 lineto



ADT 3: Queue

- Analogy: A queue (duh!) (**FIFO**) First In First Out

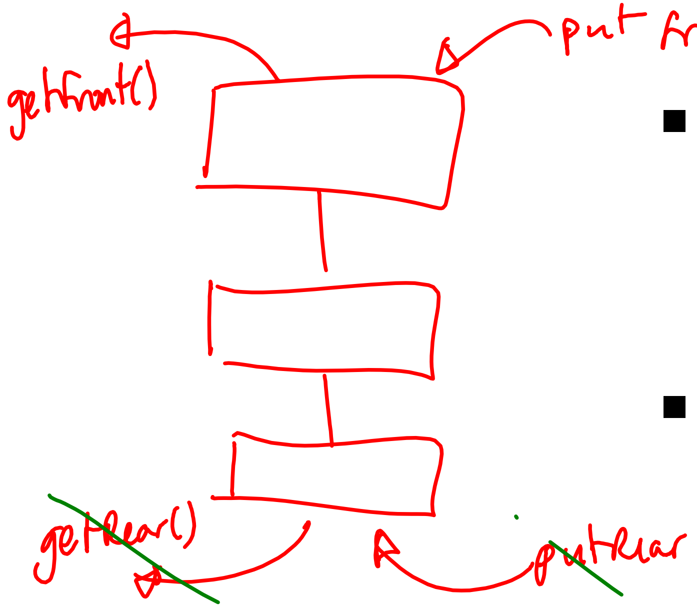


- put(item i): add i to the bottom
- get(): Take top element out of the queue and return it
- first(): get the item on top
- isEmpty(): return true iff queue is empty

Aside: Deque

- Sometimes useful to have a double-ended queue (Deque) D. E. Queue

DEQ



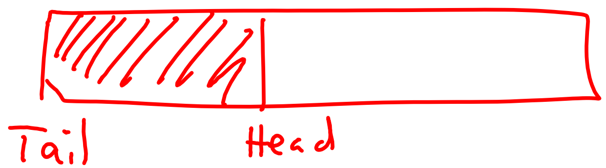
  - putFront(item i)
    putRear(item I)
  - getFront()
    getRear()
- In many ways Stack and Queue are just crippled versions of Deque!

Deque Implementation

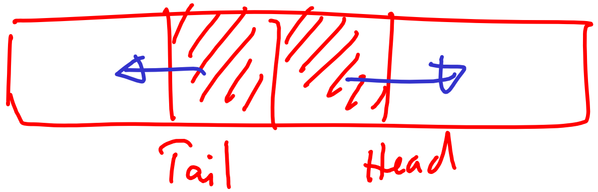
- Doubly linked lists work fine

$O(1)$ for everything

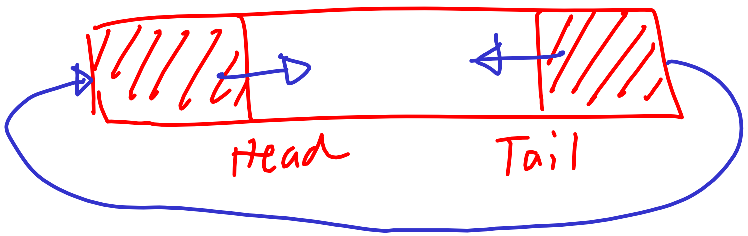
- Arrays need more thought…



putRear() always give us an expansion



Only good if you add equally to tail and head



Circular Buffer.

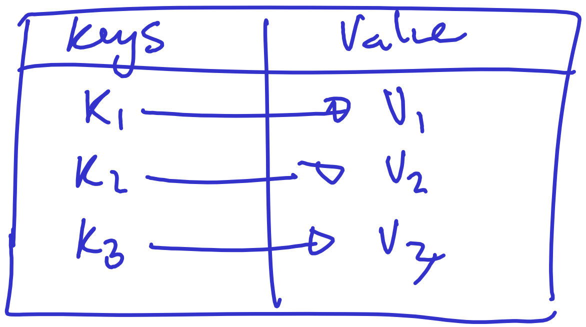
Java's Queue

- As with List, the Queue interface is implemented by different classes
  - LinkedList is the most basic

- You are supposed to select the implementation that best suits your needs

ADT 4: Table

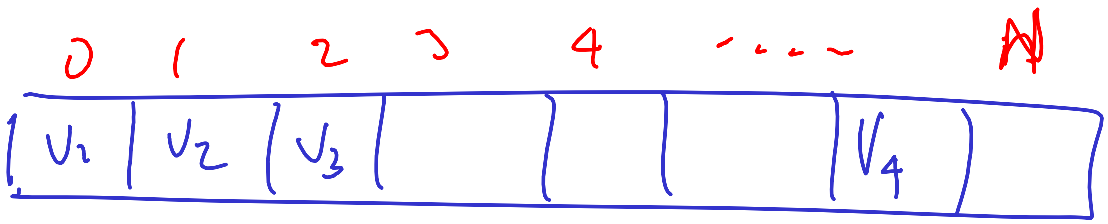
- A *dictionary* or some keys mapped to values



  - set(key k, value v): add the mapping pair (k->v)
  - get(key k): return the value for key k
  - delete(key k): remove any pair or pairs with the key k
  - isEmpty(): return true iff table is empty

Table Naïve Array Implementation

Keys : integers $0 \to N$

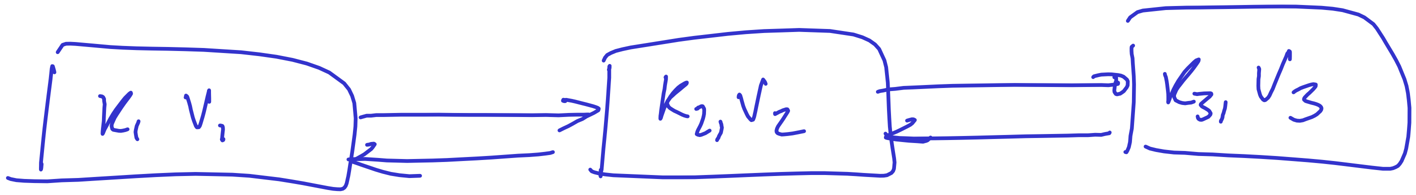


$\left.\begin{array}{l} \text{set()} \\ \text{get()} \end{array}\right\} O(1)$

0...N

Space $O(\text{range})$

Table list implementation



get() : scan list $O(n)$

set() : $O(1)$ iff duplicates

scan list, update or add at end $O(n)$

$\underline{O(n)}$ space.

Smarter Table implementation: Array

n

k_3, v_3	k_1, v_1	k_2, v_2		

$k_3 < k_1 < k_2$

get() $\rightarrow$ $O(\lg n)$

set() $\rightarrow$ $O(\lg n)$ find place
shift may give expansion

$f(n) = f(\frac{n}{2}) + kn$
$= O(\lg n)$