



## 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)

UNIVERSITY OF CAMBRIDGE

Computer Laboratory > Teaching > Course material 2009-10 > Algorithms I

search 8-Z contact

Computer Laboratory  
Course material 2009-10

---

**Additional Topics**  
Advanced Category Theory in Computer Science  
Advanced Computer Design  
Advanced Graphs  
Advanced Systems Topics  
Advanced Topics in Computer Systems  
Advanced Topics in Concurrency  
Advanced Topics in Programming Languages

**Algorithms I**  
Information for supervisors

**Algorithms II**  
An Algebraic Approach to Internet Routing  
Artificial Intelligence I  
Artificial Intelligence II  
Automated Reasoning  
Basic Rewriting Theory  
Bioinformatics  
Building an Internet Router \*  
Business Studies  
Categorical Logic  
Category Theory for Computer Science  
Chip Multiprocessors  
Compositional Architectures  
Compiler Construction  
Complexity Theory  
Computation Theory

**Computer Design**  
Computer Graphics and Image Processing  
Computer Vision  
Concepts in Programming Languages  
Concurrent and Distributed Systems  
Databases  
Denotational Semantics  
Digital Communication I  
Digital Communication II  
Digital Electronics  
Digital Signal Processing  
Discrete Mathematics I

---

**Algorithms I**  
**2009-10**

**Principal lecturer:** Dr Robert Harlie  
**Taken by:** Part IA CST, Part IANST, Part I PPS  
**Syllabus**  
**Past exam questions:** Algorithms I, Algorithms  
**Information for supervisors** (contact lecturer for access permission)

**Course Overview**

This course is intended to expose IA students to the basics of algorithms. It does so by:

- analysing a range of sorting algorithms;
- introducing established algorithm strategies (divide and conquer, etc);
- introducing and analysing a series of data structures from stacks to red-black trees

**Course Notes**

The course handout was written by Dr Frank Stajano and is an excellent document around which to base your studies. It is itself based on the course textbook (*Introduction to Algorithms* by Cormen, et al. You can [download the PDF here](#)).

Although the notes are excellent, they are not my preferred format for lecture presentation. Consequently I use a set of slides that I annotate during lectures. These you can download here (including annotations):

- [Lecture 1](#) (Intro, asymptotic analysis, assertions, insertion sort)
- [Lecture 2](#) (Insertion, selection, bubble, binary insertion and mergesort algorithms and analysis)
- [Lecture 3](#) (Quicksort)
- [Lecture 4](#) (Quickselect, Heapsort, Distribution Sort)
- [Lecture 5](#) (Algorithm Design)

**Examples Sheet**

The lecture notes contain a series of exercises meant primarily to focus and challenge the reader as they go. They are not intended for supervisions per se, although a subset are certainly appropriate.

To help guide supervisions, I have produced an examples sheet that picks out some of the exercises from the notes and supplements them with some others I have dreamed up. This sheet will be updated as the course proceeds. You can [download the current examples sheet here](#) (Last updated: 04.05.10).

In addition, you should have no difficulty in finding other exercises in the textbook or others like it.

# 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

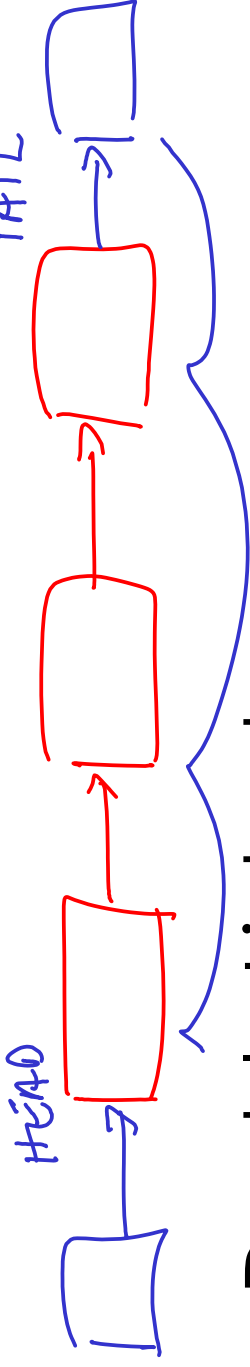
- A sequence of items
  - `add(item i, position p)`: insert item *i* into the list in position *p*.
  - `delete(position p)`: delete the item at *p*
  - `is_empty()`: returns true iff the list is empty
  - `get(position p)`: get the item at position *p*

# List Types

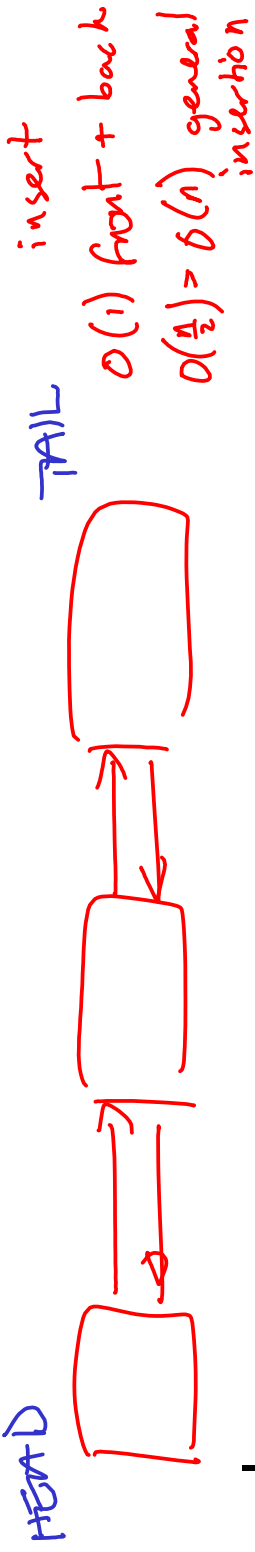
[unsorted]

insert  $O(1)$  at front  
 $O(p)$  insertion  $O(n)$

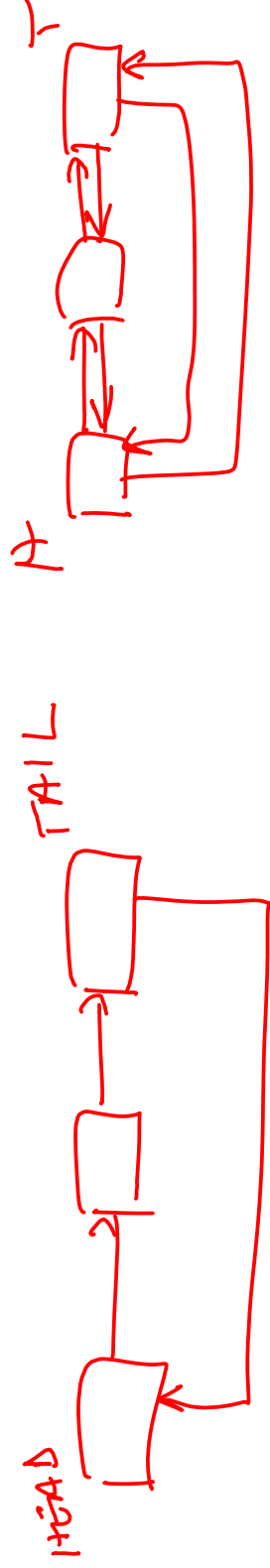
## Single



## Doubly-Linked

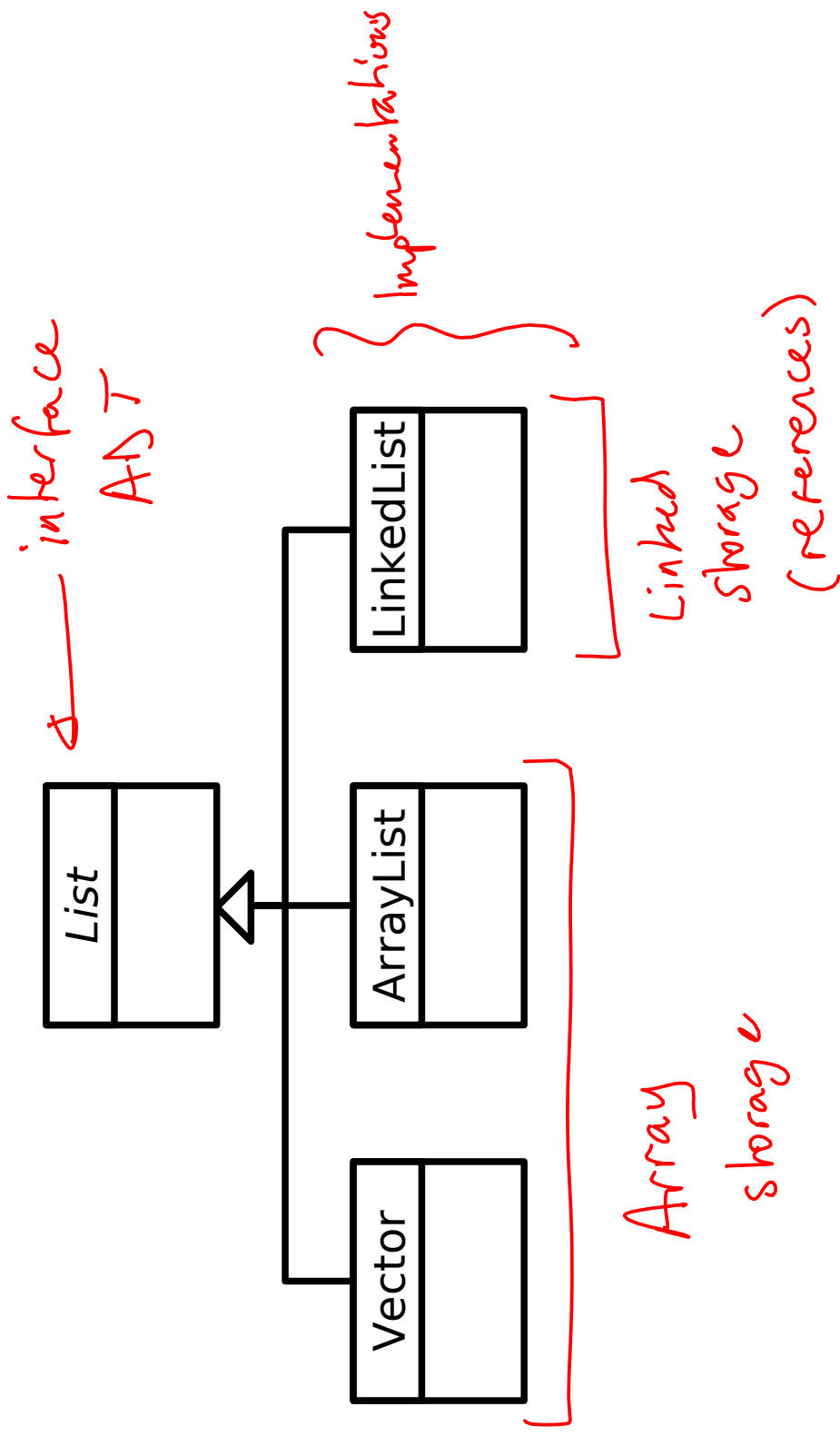


## Circular



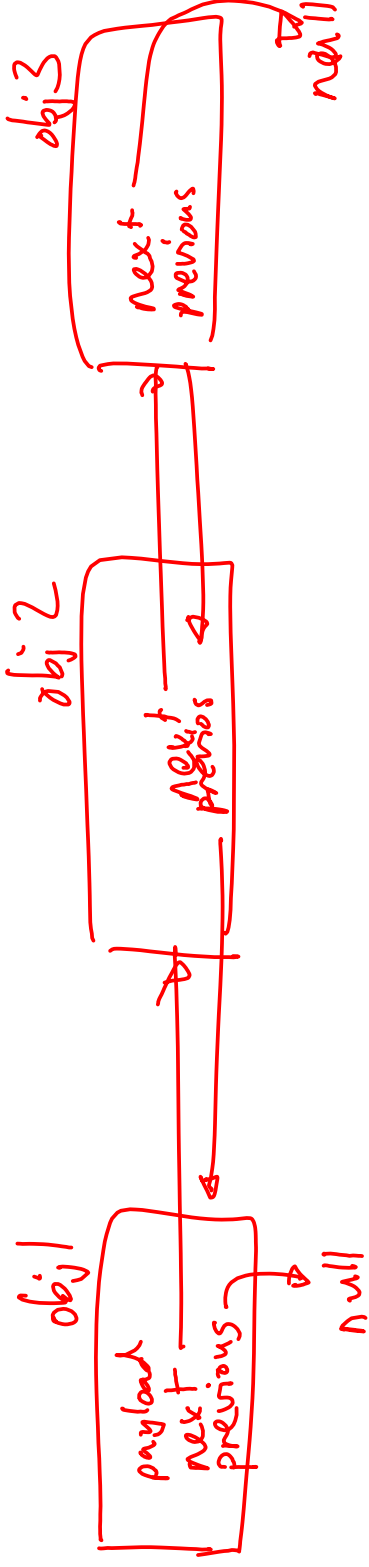
# Java List Interface

Part of Collections





# 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  $O(n)$   
 $O(1)$
- delete: Traverse the list to find the position, then delete  $O(n)$   
 $O(1)$
- 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

1<sup>st</sup> 2<sup>nd</sup>

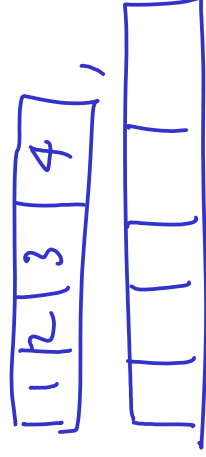


Memory overhead is zero

Random  
access

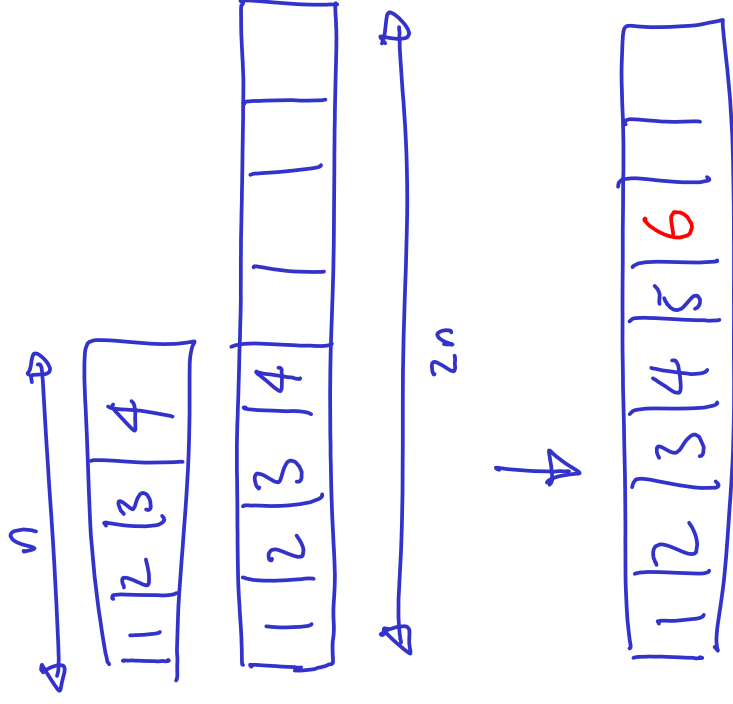
# ArrayList: add to end

Increase array size.



Making + filling a new  
array

# ArrayList: add to end



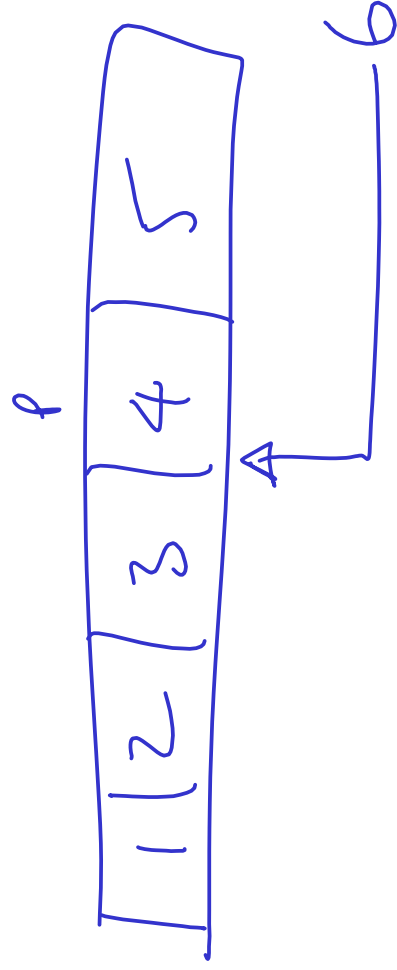
Most additions will just be direct  $O(1)$

Some additions will require expansion  $O(n)$

Add  $n$  elements in  $O(n)$  time

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

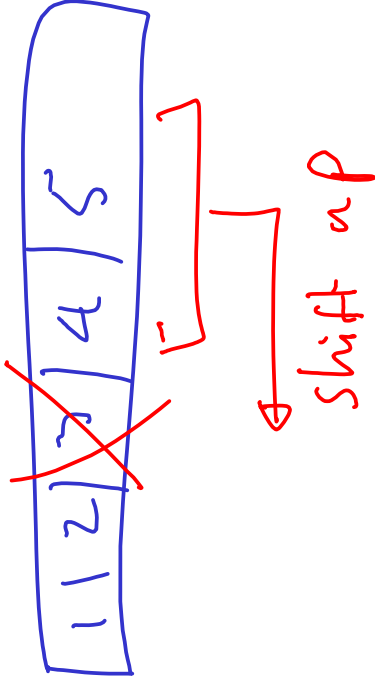
# ArrayList: add to middle



- i) There is space  $\Rightarrow$  move  $O(n-p)$  elements
- ii) Create more space, then (i)

# ArrayList

- delete



$O(n-p)$  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.

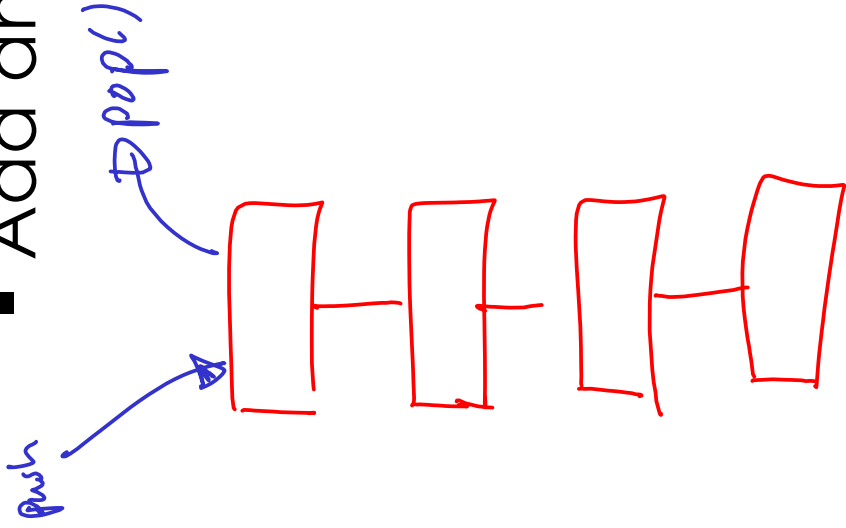
<u>200,000</u>	
AL	LL
438ms	1458ms
80ms	

Java `array list` ⇒ 10 items initialize.



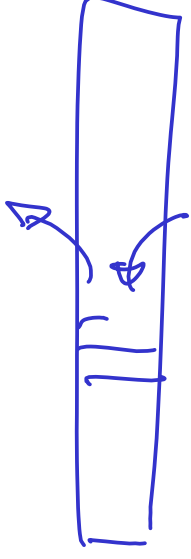
# ADT 2: Stack

- 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

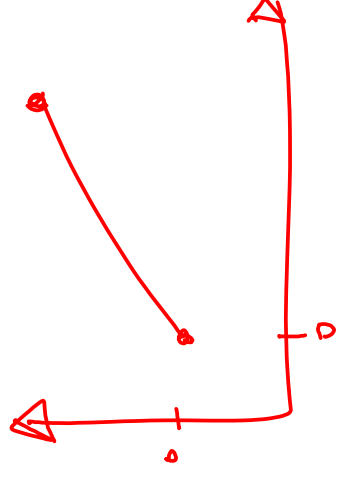
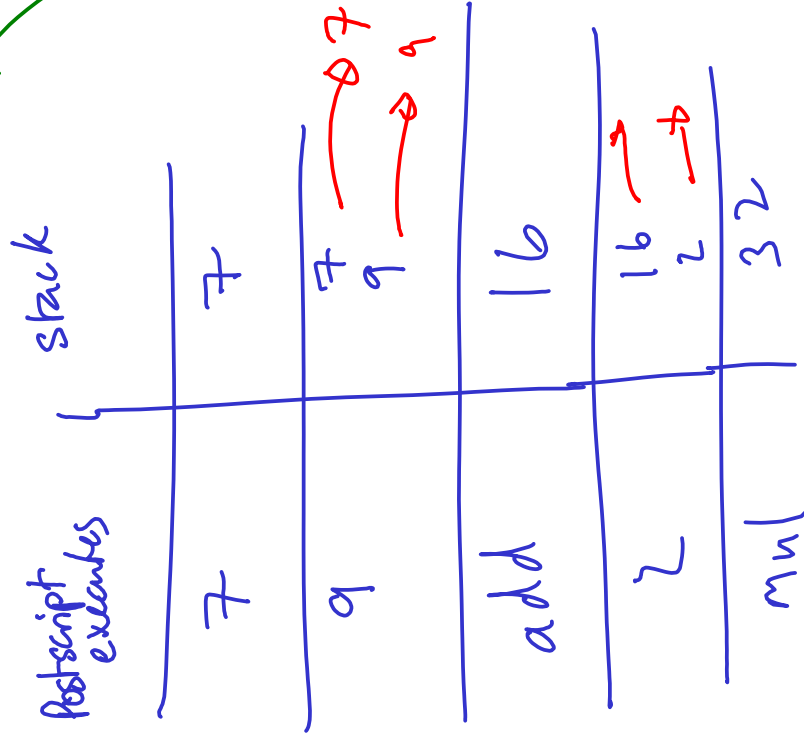


# Stack Languages!

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

7 9 add 2 mul

0 0 moveb 20 20 lineto



Reverse polish notation

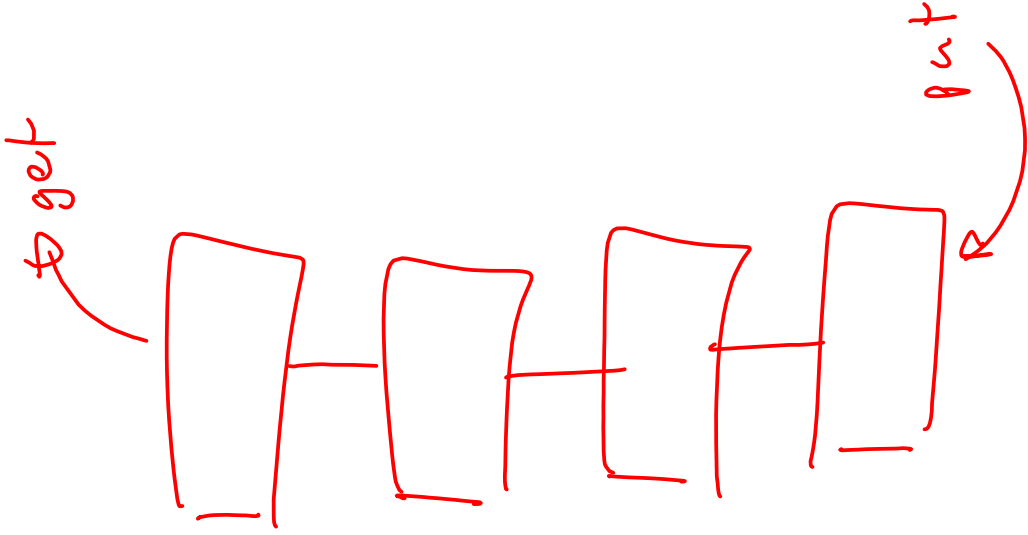
$$(7+9) \times 2$$

# 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*

*put front*



- `putFront(item i)`  
`putRear(item i)`

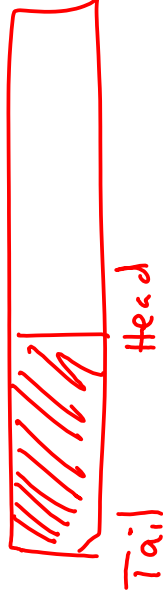
- `getFront()`  
`getRear()`

- In many ways Stack and Queue are just crippled versions of Deque!

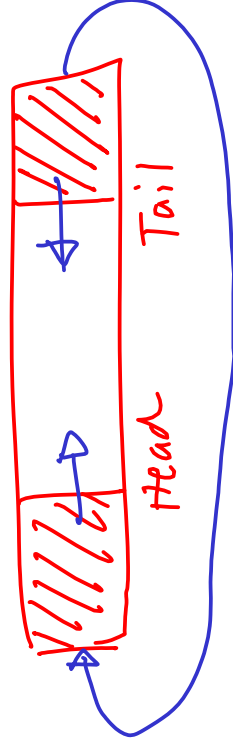
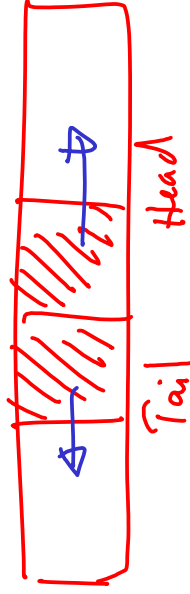
# Deque Implementation

- Doubly linked lists work fine  $O(1)$  for everything
- Arrays need more thought...

put rear() 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



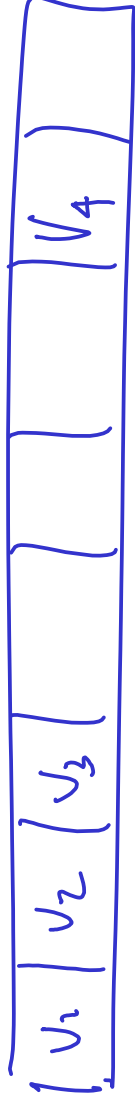
`"map"`



# Table Naïve Array Implementation

Keys: integers  $0 \rightarrow N$

0 ( 2 3 4 ... N



set(i) }  $O(1)$   
get(i)

$0 \dots N$

Space  $O(\text{range})$

# Table list implementation



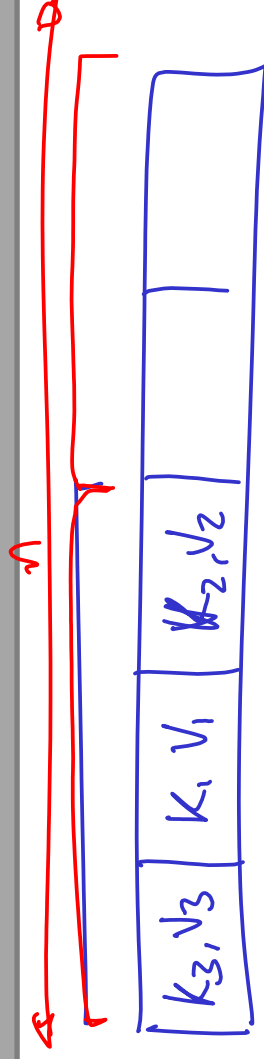
get(): scan list  $O(n)$

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

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

$O(n)$  space.

# Smarter Table implementation: Array



$$k_3 < k_1 < k_2$$

$$\text{get}() \rightarrow O(\lg n)$$

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

$$\begin{aligned} f(n) &= f\left(\frac{n}{2}\right) + kn \\ &= O(\lg n) \end{aligned}$$