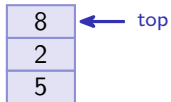


Data Types & Structures

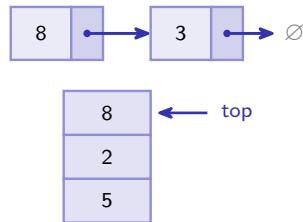
AS Computer Science ■ Topic 10

A-Level Computer Science



What we will cover

- 1 Data types & records
- 2 Arrays
- 3 Files
- 4 Stacks, queues, linked lists
- 5 Implementing with arrays
- 6 Recap





Data types and records

Pick the smallest precise **data type** :

- **INTEGER** – a whole number:
counts, indexes, IDs
- **REAL** – has a fractional part:
money, measurements
- **STRING** – characters in quotes,
for text
- **CHAR** – one character;
BOOLEAN; **DATE**

A **record** groups **several types**
under one name (dot notation
`Item1.Category`).

record TStockItem

ItemID : INTEGER

Category : STRING

ItemCost : REAL

InStock : BOOLEAN

one name holds
four fields of
different types

reach one field: `Item1.Category`

Choosing a data type, and when to use a record

INTEGER / REAL

a whole number; one with a fractional part

CHAR / STRING

a single character; a sequence of them

BOOLEANTRUE or FALSE – for **flags****DATE**

so dates sort and compare correctly

a record structurea group of fields of *different* types, under one name**when to use one**when the values **always belong together** – a customer, a stock item. Use separate variables for unrelated values

An array holds many of the **same** type; a record holds one of **several** types. Most real data needs an array *of* records.



Arrays

An **array** holds many values of the **same** type under one name, reached by an **index** .
A **record** is the opposite: different types, always belonging together.

1-D

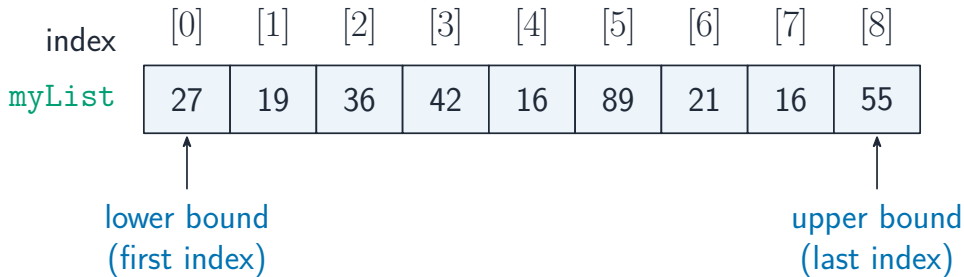


2-D – [row, column]



1-D arrays

One index. Marks [3] is the third element – but check the **lower bound**: Cambridge pseudocode usually starts at 1, most languages at 0.



Array vocabulary, and the standard sweep

index

the position of an element within the array

bounds

the **lowest and highest valid indices** – going outside them is an error

dimension

how many indices are needed: 1-D for a single sequence, 2-D for a table

a 2-D array

the **first** index is the row, the **second** the column; **nested loops** visit every cell

to traverse

visit every element in turn

the sweep pattern

to find a sum, count, maximum or minimum, set a **running variable** before the loop and update it inside

The running variable must be initialised **outside** the loop. Inside, it resets every pass – the single commonest bug in this topic.

2-D arrays: a table

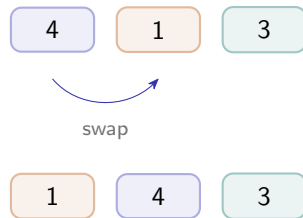
Two indices, **row first**: `Grid[2,5]` is row 2, column 5.

		column index			
		$[r, 1]$	$[r, 2]$	$[r, 3]$	$[r, 4]$
row index	$[1, c]$	12	31	17	48
	$[2, c]$	19	67	99	29
	$[3, c]$	42	22	95	61

`Grid[2,3]`
(row 2, column 3)

Searching and sorting

Linear search walks from the start until it finds the target – works on any list, $O(n)$. **Bubble sort** swaps adjacent pairs that are out of order, so the largest “bubbles” to the end each pass – $O(n^2)$.



One pass of a bubble sort

One pass of a bubble sort: adjacent pairs are compared and swapped, bubbling the largest value to the end

5	2	8	1
---	---	---	---

start

2	5	8	1
---	---	---	---

compare 5, 2 $\rightarrow$ swap

2	5	8	1
---	---	---	---

compare 5, 8 $\rightarrow$ already in order

2	5	1	8
---	---	---	---

compare 8, 1 $\rightarrow$ swap (8 reaches the end)

Worked example: find the largest

```
DECLARE Marks : ARRAY[1:4] OF INTEGER
Max ← Marks[1]
FOR i ← 2 TO 4
  IF Marks[i] > Max THEN
    Max ← Marks[i]
  ENDIF
NEXT i
OUTPUT Max
```

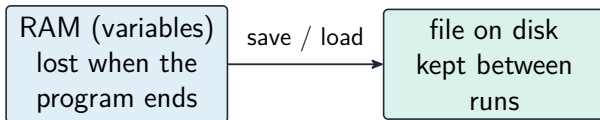
[1]	[2]	[3]	[4]
7	3	9	2

i	Marks[i]	Max
—	—	7
2	3	7
3	9	9
4	2	9

output: **9**

Start Max at the **first element**, not at 0 – then compare from the second.

Files



A **file** keeps data on disk **between runs** (RAM is lost at exit).

- OPENFILE for READ/WRITE/APPEND
- EOF tests the **end of file**
- always CLOSEFILE – or buffered writes are lost

Files, stacks and queues

a file

data on **secondary storage** , kept **between program runs**

why needed

variables live in RAM and disappear when the program stops

a text file

read and written line by line, and readable by a person

a stack

add and remove at the **top** only – **LIFO**. A pile of books: the last one on is the first one off

a queue

join at the **back**, served from the **front** – **FIFO**. Whoever waited longest is served first

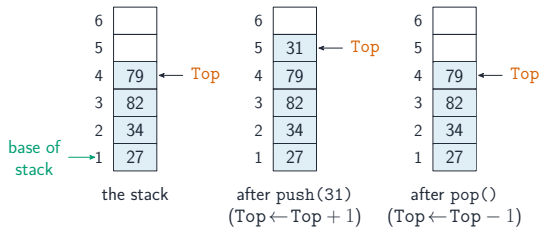
the operations

stack: push and pop. Queue: enqueue and dequeue. Both need an **empty** and a **full** check

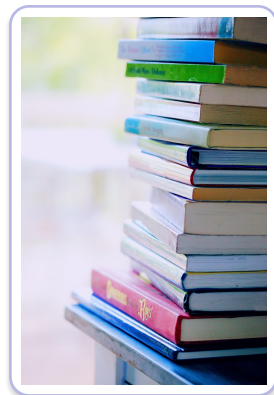
Choose by which end you need: undo history and call returns are stacks; print jobs and buffers are queues.



Abstract data types: the stack

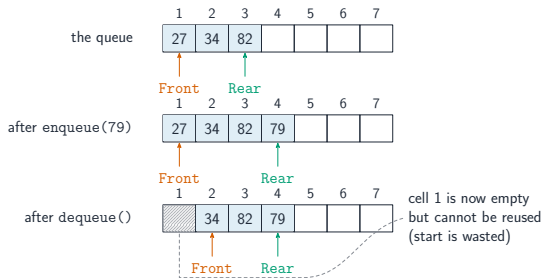


A **stack** is **LIFO** : push and pop happen at the **top** – undo history, procedure calls.



A stack is last-in, first-out

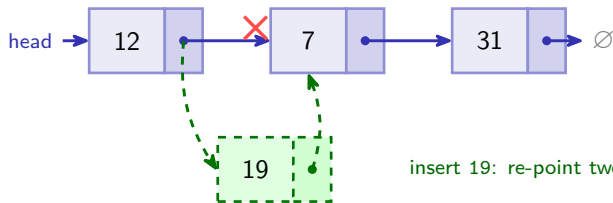
Abstract data types: the queue



A queue is first-in, first-out

A **queue** is **FIFO** : enqueue at the back,
 dequeue at the front – print **spooling** ,
 scheduling.

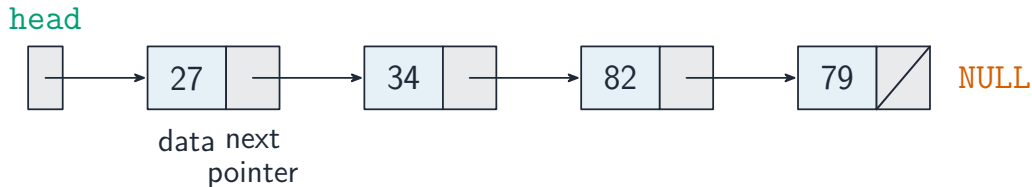
Linked list



A **linked list** chains **nodes**, each a value + a **pointer** to the next.

- + cheap insert/delete –
nothing shifts
- – slow random access

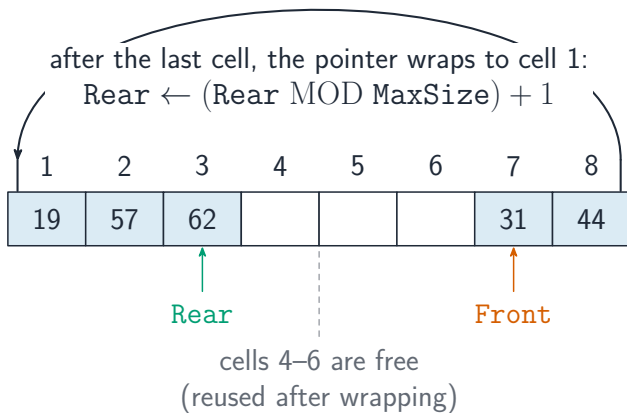
A linked list, drawn out



Each node carries its data **and** a pointer to the next; the last points at null.

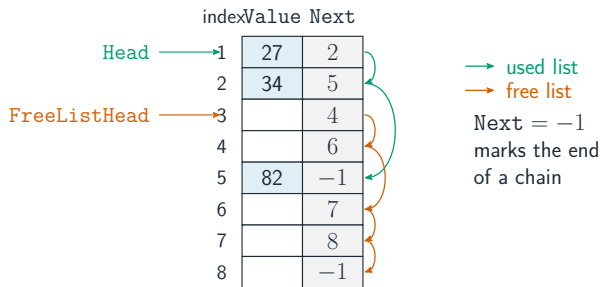


ADTs with arrays



- stack: a Top index; full = **overflow**, empty = **underflow**
- queue: a **circular array** – pointers wrap with $(p \bmod \text{MaxSize}) + 1$

Linked list in an array



Store nodes in an array of records, each with a Next index. A **free list** chains the unused slots – linked-structure flexibility with static allocation.

Worked example – a circular queue

A circular queue in an array of size 5 has $\text{Front} = 3$, $\text{Rear} = 3$ and one item. Two items are added, then two removed. Where are the pointers?

- Every move is $(\text{pointer} + 1) \text{ MOD size}$, so the pointers **wrap**.
- Adding twice moves Rear : $3 \rightarrow 4 \rightarrow 0$ (since $(4 + 1) \bmod 5 = 0$), leaving $\text{Rear} = 0$ and three items stored.
- Removing twice moves Front the same way, leaving $\text{Front} = 0$ and one item.

The wrap is the point: in a linear array queue the pointers march to the end and the freed space at the front can never be reused.

Exam tips

- Choose the right **data structure** and justify it (a record for mixed fields, a 2-D array for a grid).
- Know how to implement a **stack, queue and linked list** with an array and pointers (top; front/rear; next).
- Distinguish an **ADT** (its behaviour) from its implementation (array plus pointers).



Topic 10 – key takeaways

1

Records & arrays

records mix types; arrays hold one type

2

Files

persist data; open, read/write, close

3

Stack & queue

LIFO vs FIFO

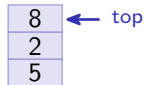
4

Linked list

nodes with pointers; cheap insert/delete

Check yourself

- 1 Which holds several different types together?
- 2 Which ADT is FIFO?
- 3 Why use a circular array for a queue?
- 4 One advantage of a linked list over an array?



Answers 1. a record 2. a queue 3. reuse the wrapped-round cells 4. cheap insertion/deletion