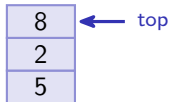


Data Types & Structures

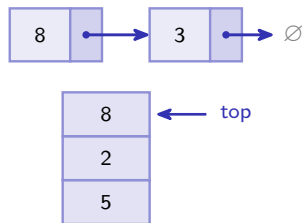
AS Computer Science ■ Topic 10

A-Level Computer Science



What we will cover

- 1 Data types and records 10.1
- 2 Arrays 10.2
- 3 Bubble sort, bounds 10.2.1
- 4 Array patterns 10.2.2
- 5 Files 10.3
- 6 Stacks, queues, lists 10.4
- 7 ADTs in arrays 10.4.1
- 8 Recap



1

Data types and records

Sheet 10.1 – what you need to know

You must be able to:

- choose a data type, and give the type of an expression
- declare a record with `TYPE ... ENDTYPE`, and an array of records
- use dot notation to read and set a field

What the exam asks

“Write pseudocode to declare the record structure for type RentalRecord.” [4]

“Complete the table by giving the data type of the evaluated expression.” [4]

“Complete each pseudocode expression so that it evaluates to the value shown.” [5]

Practise on exercise sheet 10.1, then try the real exam questions on past-paper sheet 10.1.

Data types and records

Pick the smallest precise **data type** 数据类型:

- 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** 记录 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

BOOLEAN

TRUE or FALSE – for **flags**

DATE

so dates sort and compare correctly

**a record
structure 记录结构**

a 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.

Declaring a record, and an array of records

```

TYPE RentalRecord
  DECLARE CarID : STRING
  DECLARE CostPerHour : REAL
  DECLARE Hours : INTEGER
  DECLARE Returned : BOOLEAN
ENDTYPE

DECLARE Hire : ARRAY[1:50] OF RentalRecord
Hire[1].CostPerHour ← 4.50

```

Type of a result: RIGHT(S, 2) STRING; NOT X,
IS_NUM(S) BOOLEAN; N * 2.5 REAL.

- **dot notation:**
Hire[1].CostPerHour
- one array of records keeps each car's data **together**: no parallel arrays to keep in step, one index, fields of different types
- ARRAY[0:99, 0:99] has $100 \times 100 = 10\,000$ elements
- a **local** variable exists only inside its module; a **global** one everywhere

2

Arrays

Sheet 10.2 – what you need to know

You must be able to:

- declare, fill and total 1-D and 2-D arrays
- write a linear search that stops when it finds the item
- trace an array procedure

What the exam asks

“Complete the pseudocode for Store(). All variables used must be declared.” [6]

“Comment on why the programmer chose -1 as the initial value of FoundAt.” [1]

“Write pseudocode for module CheckMark().” [6]

Practise on exercise sheet 10.2, then try the real exam questions on past-paper sheet 10.2.

Arrays

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

1-D

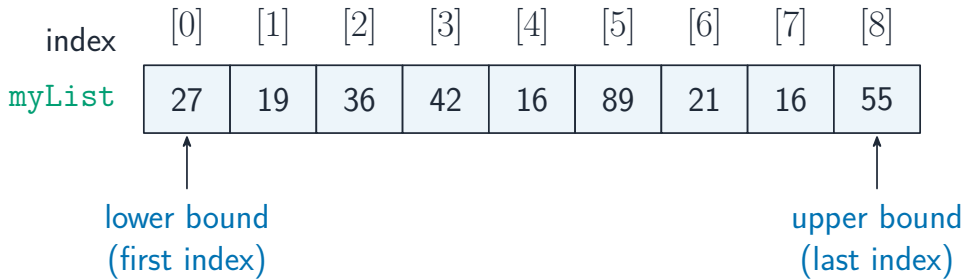


2-D – [row, column]



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 索引	the position of an element within the array
bounds 边界	the lowest and highest valid indices – going outside them is an error
dimension 维度	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 遍历	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)

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.

Linear search that stops early

```
FoundAt ← -1
Index ← 1
WHILE FoundAt = -1 AND Index ≤ 100
    IF Names[Index] = Target THEN
        FoundAt ← Index
    ENDIF
    Index ← Index + 1
ENDWHILE
```

“Comment on why the programmer chose `-1` as the initial value of FoundAt.” [1]

`-1` can **never** be a valid index of `Names[1:100]`, so it means “not found”.

A FOR loop always runs to 100; a WHILE with two conditions **stops** as soon as the name is found.

Row totals of a 2-D array: an outer loop over rows, Total set to 0 **inside** it, an inner loop over columns.

3

Bubble sort, bounds

Sheet 10.2.1 – what you need to know

You must be able to:

- write an efficient bubble sort, with a flag
- count the elements from an array's bounds
- dry-run a procedure in a trace table

What the exam asks

“Complete the trace table by dry running the procedure ... CALL Process(3)” [6]

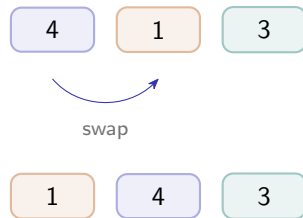
“Write pseudocode for the procedure Sort().” [8]

“Draw a program flowchart to represent the algorithm.” [5]

Practise on exercise sheet 10.2.1, then try the real exam questions on past-paper sheet 10.2.1.

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)

An efficient bubble sort

```
Last ← 9
REPEAT
  NoSwaps ← TRUE
  FOR J ← 1 TO Last
    IF Data[J] > Data[J + 1] THEN
      Temp ← Data[J]
      Data[J] ← Data[J + 1]
      Data[J + 1] ← Temp
      NoSwaps ← FALSE
    ENDIF
  NEXT J
  Last ← Last - 1
UNTIL NoSwaps = TRUE OR Last = 0
```

- Temp holds one value during the **swap**
- NoSwaps: a pass with no swap means the data is **sorted**, so stop
- Last shrinks: after each pass the largest value is already in place
- sorting a 2-D array by column 2: compare Data[J, 2], swap **both** columns

Elements from bounds: [5:14] has 14 – 5 + 1 = 10; [1:150, 1:2] has 300.

4

Array patterns

Sheet 10.2.2 – what you need to know

You must be able to:

- stop storing when an array is full
- use a Boolean array as a lookup table
- read to a sentinel, and size an array for a task

What the exam asks

“Complete the pseudocode for Store(). All variables used must be declared.” [6]

“Comment on why the programmer chose `-1` as the initial value of FoundAt.” [1]

“Write pseudocode for module CheckMark().” [6]

Practise on exercise sheet 10.2.2, then try the real exam questions on past-paper sheet 10.2.

Three array patterns from the exam

Capacity and count

Keep Count of the slots used. Store only while $\text{Count} < \text{Capacity}$; a full array stops the input. A strictly increasing rule: store a value only if it is **greater** than $\text{Data}[\text{Count}]$.

A Boolean lookup

Possible[1:100] of BOOLEAN, all FALSE; nested loops (FOR ... STEP) set Possible[N] $\leftarrow$ TRUE for each value that can occur. Testing a value is then one look-up.

A sentinel, and sizing

Read until -999, which is never real data, and do not store it.

Hourly readings for a week from 6 sensors:

$6 \times 24 \times 7 = 1008$ elements, $\text{ARRAY}[1:6, 1:168]$.

Pass an array to a module BYREF when the module must change it.

5

Files

Sheet 10.3 – what you need to know

You must be able to:

- open, read, write, append and close a text file
- read to the end of a file with EOF
- design a line format with separators, and convert types

What the exam asks

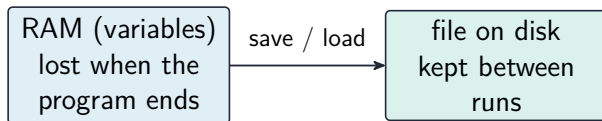
“Complete the following numbered steps: Step 1: open the file ... in ...” **[6]**

“Write pseudocode for module AddNewCustomers().” **[8]**

“Explain the additional file design changes needed to extract each item after the line is read.” **[3]**

Practise on exercise sheet 10.3, then try the real exam questions on past-paper sheet 10.3.

Files



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

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

Reading and appending a text file

```
OPENFILE "Stock.txt" FOR READ
Count ← 0
WHILE NOT EOF("Stock.txt")
    READFILE "Stock.txt", Line
    Count ← Count + 1
ENDWHILE
CLOSEFILE "Stock.txt"

OPENFILE "Log.txt" FOR APPEND
Line ← NUM_TO_STR(ID) & "," & Name
WRITEFILE "Log.txt", Line
CLOSEFILE "Log.txt"
```

- WRITE **replaces** the file; APPEND adds to its end
- a file holds **strings**:
NUM_TO_STR before writing,
STR_TO_NUM after reading
- fields on one line need a **separator** that cannot occur in the data; for encrypted text, store each field's **length** first
- **serial**: in the order added;
sequential: in key order

6

Stacks, queues, lists

Sheet 10.4 – what you need to know

You must be able to:

- describe a stack (LIFO), a queue (FIFO) and a linked list
- write Push and Pop, with full and empty checks
- set the pointers of a linked list

What the exam asks

“Complete the pseudocode for Pop().” [5]

“Describe how the queue could be used to reverse the items currently stored on the stack.” [7]

“Complete the pointer field to produce a linked list that is in alphabetical order.” [3]

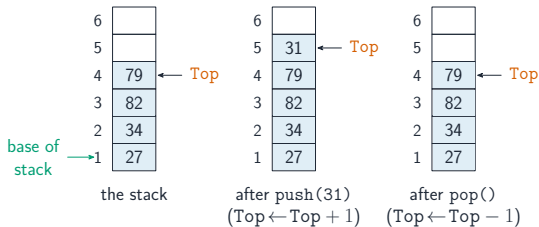
Practise on exercise sheet 10.4, then try the real exam questions on past-paper sheet 10.4.

Files, stacks and queues

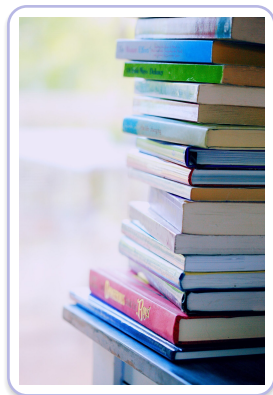
a file 文件	data on secondary storage 辅助存储器, kept between program runs
why needed	variables live in RAM and disappear when the program stops
a text file 文本文件	read and written line by line, and readable by a person
a stack	add and remove at the top only – LIFO . A pile of books: the last one on is the first 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

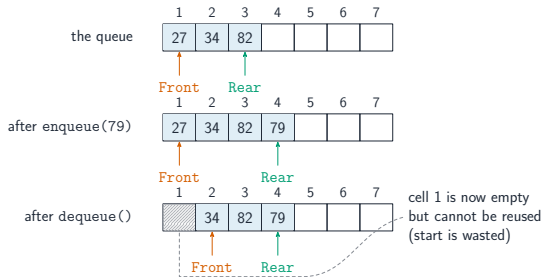


A **stack** 栈 is **LIFO** 后进先出: push and pop happen at the **top** – undo history, procedure calls.



A stack is last-in, first-out

Abstract data types: the queue



A queue is first-in, first-out

A **queue** 队列 is **FIFO** 先进先出: enqueue at the back, dequeue at the front – print **spooling** 排队打印, scheduling.

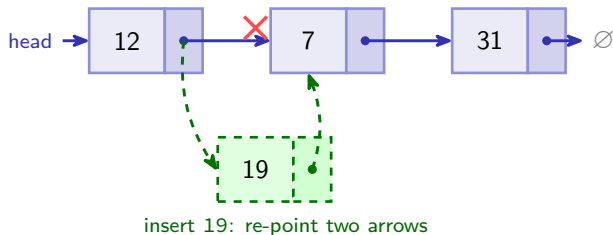
Push and Pop

```
PROCEDURE Push(Item : STRING)
  IF TopOfStack = 10 THEN
    OUTPUT "Stack full"
  ELSE
    TopOfStack ← TopOfStack + 1
    Stack[TopOfStack] ← Item
  ENDIF
ENDPROCEDURE
```

```
FUNCTION Pop() RETURNS STRING
  DECLARE Item : STRING
  Item ← ""
  IF TopOfStack > 0 THEN
    Item ← Stack[TopOfStack]
    TopOfStack ← TopOfStack - 1
  ENDIF
  RETURN Item
ENDFUNCTION
```

Here TopOfStack points at the **last item** (0 when empty). If it points at the **next free** slot instead, Push stores first and then adds 1, and Pop subtracts 1 first.

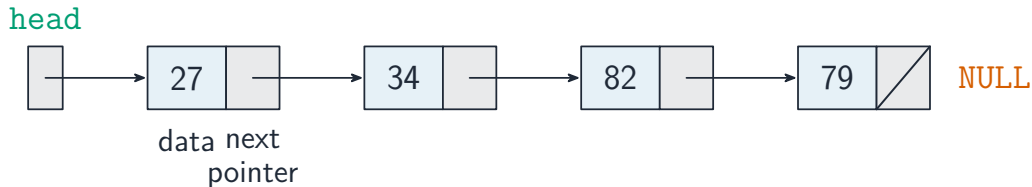
Linked list



A **linked list** 链表 chains **nodes** 节点, each a value + a **pointer** 指针 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.

Reversing a stack, and ordering a linked list

“Describe how the queue could be used to reverse the items on the stack.” [7]

- 1 Pop the top item from the stack.
- 2 Add it to the rear of the queue.
- 3 Repeat until the stack is empty.
- 4 Remove the item at the front of the queue.
- 5 Push it onto the stack.
- 6 Repeat until the queue is empty: the order is reversed.

index	data	pointer
1	Lemur	3
2	Ant	4
3	Zebra	0
4	Fox	1

Start pointer = 2: Ant → Fox → Lemur → Zebra.

The null pointer must not be a valid index: 0 for arrays from 1, -1 for arrays from 0.

7

ADTs in arrays

Sheet 10.4.1 – what you need to know

You must be able to:

- add to and remove from a circular queue, wrapping the pointers
- insert and delete in a linked list using a free list
- choose an ADT for a scenario

What the exam asks

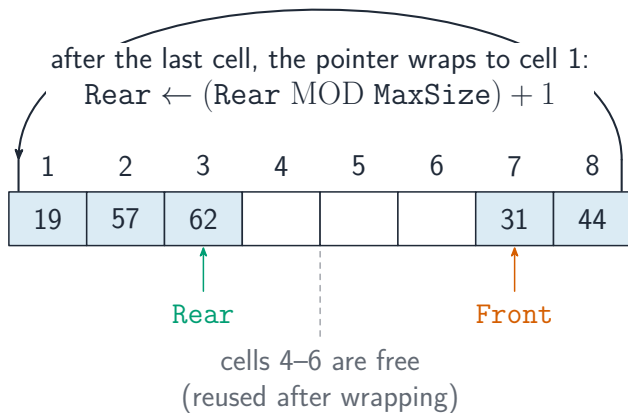
“A function `Push()` will add a value onto the stack by manipulating the array and variables...” [5]

“A module `AddTo()` will add a value to the queue...” [6]

“A new node is inserted between two nodes...” [4]

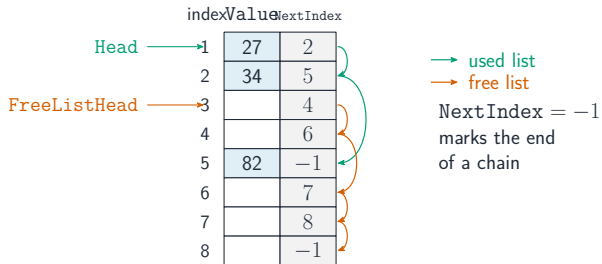
Practise on exercise sheet 10.4.1, then try the real exam questions on past-paper sheet 10.4.1.

ADTs with arrays



- stack: a Top index; full = **overflow** 溢出, empty = **underflow** 下溢
- queue: a **circular array** 循环数组 – 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** 空闲列表 chains the unused slots – linked-structure flexibility with static allocation.

Worked example – a circular queue, indexed from 1

```
PROCEDURE AddTo(Item : STRING)
  IF NumItems = 6 THEN
    OUTPUT "Queue full"
  ELSE
    IF Rear = 6 THEN
      Rear ← 1
    ELSE
      Rear ← Rear + 1
    ENDIF
    Queue[Rear] ← Item
    NumItems ← NumItems + 1
  ENDIF
ENDPROCEDURE
```

Queue[1:6]: Front = 5,
Rear = 5, one item. Add two,
then remove two.

Adding moves Rear: $5 \rightarrow 6 \rightarrow 1$
(it wraps).
Removing moves Front the
same way: $5 \rightarrow 6 \rightarrow 1$.
Now Front = Rear = 1: exactly
one item is left.

Indexed from 0, the same wrap is
(Rear + 1) MOD 6.

Inserting and deleting with a free list; choosing an ADT

Insert a new node

- 1 take the node at the **free pointer**;
move the free pointer to the next free node
- 2 store the data in it
- 3 find the nodes it goes **between**
- 4 new node's pointer $\leftarrow$ the next node;
previous node's pointer $\leftarrow$ the new node

Delete a node

Point the previous node **past** it; add its slot to the **front** of the free list.

scenario

ADT

the return addresses of nested calls	stack
"undo" in an editor	stack
documents waiting for a printer	queue
names kept in alphabetical order with many insertions	linked list

An array linked list has a **fixed size**, and it can only be searched from the start.

8

Recap

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

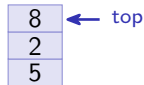
4

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