Data Types and Structures

A-Level Computer Science

Choosing data types

Every variable needs a **data type** 数据类型 — the kind of value it holds and the operations allowed:

- `INTEGER` — a whole number (`42`, `-7`). For counts, indexes, IDs.
- `REAL` — a number with a fractional part (`3.14`). For money, measurements.
- `STRING` — characters in quotes (`"Hello"`). For text.
- `CHAR` — a single character (`'A'`).
- `BOOLEAN` — `TRUE` or `FALSE`. For flags.
- `DATE` — a calendar date.

Pick the smallest precise type that fits: `INTEGER` for whole counts, `BOOLEAN` for flags (not the strings `"yes"`/`"no"`).

The "give the appropriate data type" tables are decided by how the value is used: the average mark of a class is `REAL` (it has a fractional part); an email address is `STRING`; the number of students is `INTEGER`; whether a student has paid is `BOOLEAN`; a date of birth is `DATE`; an array index is always `INTEGER`; a single grade letter is `CHAR`; a phone number is a `STRING`, because it starts with `0` and is never used in arithmetic. A `BOOLEAN` is used for a flag with only two states: whether a search has found its target, whether a member has paid, whether a seat is booked. For the identifier table, the variable name must be meaningful too: `NumberOfPeople`, not `n`.

Records

A **record** 记录 (a **record structure** 记录结构) holds **several fields of different types** under one name — useful when several values describe one thing.

```
TYPE TStockItem
    DECLARE ItemID : INTEGER
    DECLARE Category : STRING
    DECLARE ItemCost : REAL
    DECLARE InStock : BOOLEAN
ENDTYPE
```


This defines the **type** `TStockItem`; declare variables of it:

```
DECLARE Item1 : TStockItem
DECLARE Items : ARRAY[1:100] OF TStockItem
```

Use dot notation to reach each **field** 字段:

```
Item1.Category ← "Fruit"
OUTPUT Item1.Category, " costs ", Item1.ItemCost
```

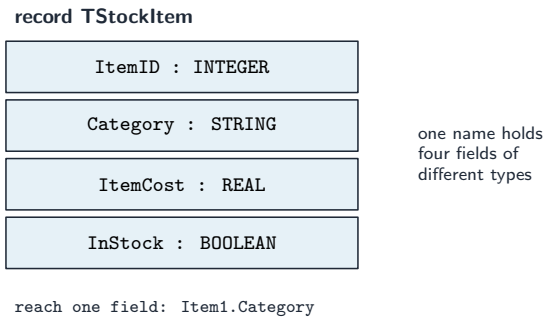
Use a record when values **always belong together** (a customer, a stock item); use separate variables for unrelated values.

Worked example. A club stores, for each student, a student ID (a string), a name, a date of birth and up to three club numbers (integers). Write pseudocode to declare the record type, an array to hold 3000 students, and a statement that stores a name in the first element.

```
TYPE Student
    DECLARE StudentID : STRING
    DECLARE Name : STRING
    DECLARE DateOfBirth : DATE
    DECLARE Club : ARRAY[1:3] OF INTEGER
ENDTYPE

DECLARE Membership : ARRAY[1:3000] OF Student
Membership[1].Name ← "Li Wei"
```

The marks: `TYPE` with the identifier and `ENDTYPE`; each field declared with a suitable type; the array declared with its bounds and `OF Student`; the field reached with the index and a dot. A "state the error in the record declaration" question usually points at a missing `ENDTYPE`, a field with no type, or a field declared as a `STRING` that must hold arithmetic. Two conventions score marks on their own: an **unused element** is marked with a value that cannot be real data (an empty string, `-1`, an ID of `0`), and it is good practice to use the same marker everywhere so that every module can recognise an unused slot; an unused club field is `0`. The benefits of an array of records, for a "state three benefits": all the data for one entity is held under one identifier; the fields can have different data types; one array replaces several parallel arrays that would have to be kept in step; the whole set can be processed by one loop or passed as one parameter; and adding a field changes the type definition only. For one customer the suitable structure is a **record** (fields of different types under one name); for all customers it is an **array of records**.



A record holds several fields of different types under one name

Arrays

An **array** 数组 is an ordered collection of items of the **same type**, under one name, reached by an **index** 索引.

- **element** 元素 — one item in the array.
- **bounds** 边界 — the lowest and highest valid indices.
- **dimension** 维度 — 1-D (a list), 2-D (a table), etc.
- **lower bound** 下界 and **upper bound** 上界 — the first and last valid index; the number of elements is upper bound minus lower bound plus one, and for a 2-D array the product of the two counts.

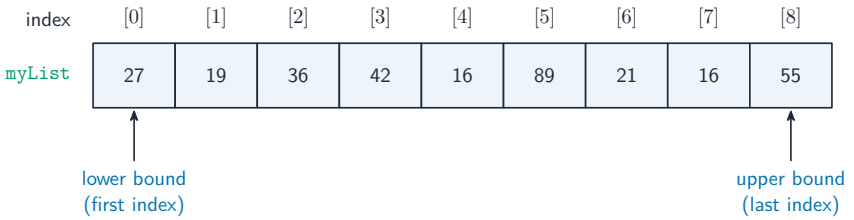
So in `ThisArray[n] ← 42` the array has one dimension, the index is the variable `n` (an `INTEGER`), and the element at that index receives `42`. Before an array can be declared you need its **data type** as well as its bounds. To declare 120 values that may include a decimal place: `DECLARE Data : ARRAY[1:120] OF REAL`; a 150-row, two-column table of strings: `DECLARE Data : ARRAY[1:150, 1:2] OF STRING`, which has 300 elements. The benefits of an array over separate variables, for a two-mark explain: one identifier instead of thirty; the elements can be processed by a loop with the index as the counter; the size is easy to change; and the whole set can be passed to a module as one parameter. An array can also replace a chain of selection statements: `DaysInMonth[Month]` looks up the answer directly instead of twelve `IF` clauses, which is shorter, faster to write and easier to maintain.

1-D arrays

```
DECLARE Names : ARRAY[1:5] OF STRING
Names[3] ← "Cara"
OUTPUT Names[3]
```

Process every element with a `FOR` loop:

```
FOR i ← 1 TO 5
    OUTPUT Names[i]
NEXT i
```

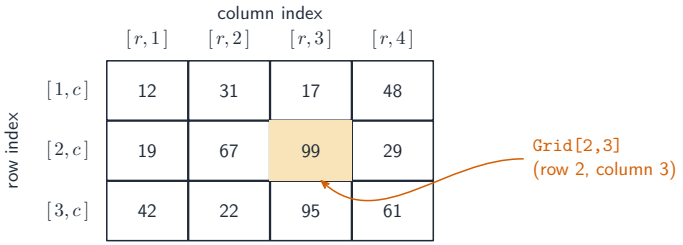


A 1-D array (a list) with indices and bounds

2-D arrays (2D array)

```
DECLARE Grid : ARRAY[1:3, 1:4] OF INTEGER
Grid[2, 3] ← 99
```

The first index is the row, the second the column. Use nested loops to visit every cell. Use **1-D** for a single sequence, **2-D** for two natural dimensions (a grid, rows × columns).



A 2-D array (a table) with row and column indices

Common operations

A **linear search** 线性查找 checks each element until found:

```
FOR i ← 1 TO n
    IF A[i] = Target THEN
        OUTPUT "Found at ", i
    ENDIF
NEXT i
```

To find a sum, count, maximum or minimum, set a running variable then sweep through:

```
Max ← A[1]
FOR i ← 2 TO n
    IF A[i] > Max THEN Max ← A[i]
NEXT i
```

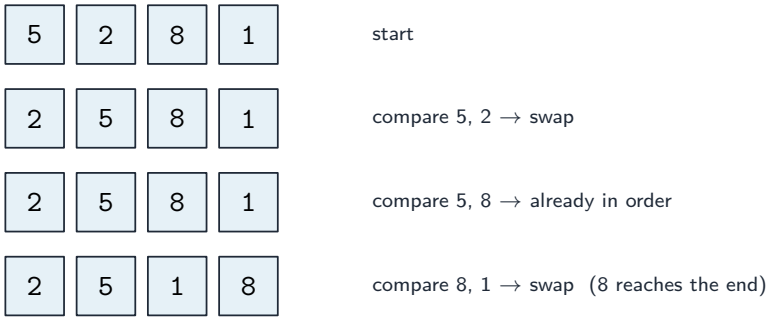
A **bubble sort** 冒泡排序 puts an array in order: pass through it comparing each adjacent pair and **swapping** any that are out of order; repeat the passes until one pass makes no swaps.

Paper 2 asks for these algorithms both as pseudocode and as **steps in words**, and sometimes in their "efficient" form:

- **Largest value**: set `Largest` to the first element; for each remaining element, if it is bigger than `Largest`, store it in `Largest`; after the loop output `Largest`. For the *position* of the largest, keep a second variable that stores the index each time `Largest` changes.
- **Linear search returning a position**: set `FoundAt ← -1` before the loop (a value that can never be a valid index, so it means "not found"); loop through the array; when the element matches, store the index and leave the loop; after the loop test `FoundAt`.
- **Count or output the non-blank elements**: compare each element with the marker for an unused element (`""` or `-1`) and count or output only those that differ.
- **Remove an item**: find its index by a linear search; move every later element one place towards the start, so the gap closes; mark the last element as unused (or decrease the count).
- **Insert into a sorted array**: find the first index whose element is larger; move that element and every later one one place towards the end; store the new value in the gap.
- **Efficient bubble sort**: a `Swapped` flag so that the passes stop as soon as a pass makes no swap, and an upper limit that falls by one each pass because the largest value has already reached the end.


```
REPEAT
    Swapped ← FALSE
    FOR Index ← 1 TO Limit - 1
        IF Data[Index] > Data[Index + 1] THEN
            Temp ← Data[Index]
            Data[Index] ← Data[Index + 1]
            Data[Index + 1] ← Temp
            Swapped ← TRUE
        ENDIF
    NEXT Index
    Limit ← Limit - 1
UNTIL Swapped = FALSE
```

The marks are for the outer loop that repeats until no swaps, the flag set inside the `IF`, the three-line swap with a temporary variable, and the shrinking limit. A sort in "steps" (stepwise refinement) is: repeat until sorted; on each pass compare adjacent pairs; swap a pair that is out of order; after each pass the largest unsorted value is at the end. Two 1-D arrays of records or of parallel data are processed with one loop and one index; a 2-D array needs a nested loop, the outer over rows and the inner over columns, and a search in one row fixes the row index and loops over the column.

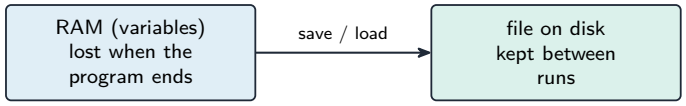


One pass of a bubble sort: adjacent pairs are compared and swapped, bubbling the largest value to the end

Files

A **file** 文件 is data stored on **secondary storage** 辅助存储器, kept between program runs. Variables in RAM disappear when the program ends, so to save data permanently

(high scores, records, settings) the program writes to a file. Files also let programs share data and restart from a saved state.



Variables in RAM vanish when the program ends; a file on disk persists between runs

A **text file** 文本文件 holds one or more lines of readable characters; programs read and write **text files** line by line. Open a file before use and **close** it after:

```
OPENFILE "data.txt" FOR READ      // or FOR WRITE, FOR APPEND
WHILE NOT EOF("data.txt") DO
    READFILE "data.txt", LineString
    OUTPUT LineString
ENDWHILE
CLOSEFILE "data.txt"
```

`EOF` tests the **end of file** 文件结束 before reading. To write:

```
OPENFILE "log.txt" FOR WRITE
FOR i ← 1 TO 100
    WRITEFILE "log.txt", "Event " & i
NEXT i
CLOSEFILE "log.txt"
```

Always **close** every file — otherwise buffered writes may be lost and other programs may be locked out.

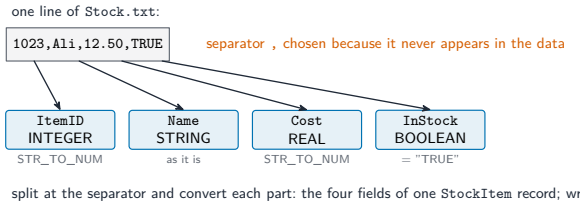
Why files (two marks): the data is kept after the program ends, so it is available the next time the program runs; it can be shared with other programs; and it can hold more than fits in memory. The characteristic of a text file that lets a program work through it is that it is a sequence of lines, read one after another from the start. The three **modes**: `READ` to read from the start; `WRITE` to create a new file, which **deletes any existing contents**, so it cannot be used to add to a file; `APPEND` to add lines at the end of an existing file. Test `EOF` before every read, and open the file only once, even when several modules use it.

Worked example. Write pseudocode for a procedure `LastLines(FileName : STRING)` that outputs the last three lines of a text file, in order.

```
PROCEDURE LastLines(BYVAL FileName : STRING)
    DECLARE LineX, LineY, LineZ : STRING
    LineX ← ""
    LineY ← ""
    LineZ ← ""
    OPENFILE FileName FOR READ
    WHILE NOT EOF(FileName) DO
        LineX ← LineY
        LineY ← LineZ
        READFILE FileName, LineZ
    ENDWHILE
    CLOSEFILE FileName
    OUTPUT LineX
    OUTPUT LineY
    OUTPUT LineZ
ENDPROCEDURE
```

Each new line pushes the previous three along, so when the file ends the three variables hold its last three lines; a file with fewer lines outputs empty strings. To output the *first* five lines, count the lines read and stop the loop at five or at `EOF`, whichever comes first; a file that is empty is detected by `EOF` being `TRUE` immediately after opening.

Fields in a line. A text file holds strings, so a record is written as one line with its fields joined by a **separator** 分隔符 character, and each number or Boolean converted with `NUM_TO_STR` (and read back with `STR_TO_NUM`, or by comparing with `"TRUE"`). Choose a separator that can never appear in the data: a comma or `|` for names and numbers, never a space when a name may contain one. If a field may contain any character, the separator can be confused with data; the fix is to put each field on its own line, or to write the field's length before it. One item per line is simple to read back but uses more lines and makes a record harder to see as a unit. Reading a file whose lines are in a known order (ascending by an ID) allows the search to stop as soon as a larger ID is read, instead of reading to the end. A save file that is created each time the game is saved needs a **meaningful filename**, for instance the player's name and the date and time, so that any earlier save can be restored.



One line of a text file is one record: fields joined by a separator, converted to their types when read back

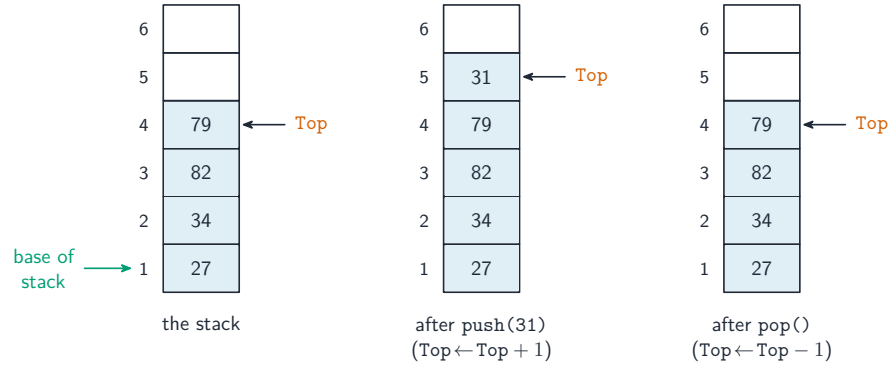
Abstract Data Types (ADTs)

An **Abstract Data Type** 抽象数据类型 (ADT) is a **collection of data plus operations on it**, defined by **what** it does, not how it is stored. The user works only through the operations; the implementation is hidden, so it can change without affecting code that uses the ADT. Know three: stack, queue, linked list.

The one-mark definition: an ADT is a collection of data together with a set of operations on that data. A stack, a queue, a linked list, a binary tree and an array are all ADTs. To **justify** a choice: a queue when items must be handled in the order they arrived (print jobs, key presses, customers in a shop), because it is first in, first out; a stack when the most recent item must be handled first (undo, going back through web pages, reversing an order, the return addresses of nested calls), because it is last in, first out; a linked list when items are inserted and deleted in the middle of an ordered sequence often, because only pointers change and nothing has to be shifted. To **compare a stack and a queue**: both are linear structures of items with an order, both are implemented with an array and pointers, and both need a check for full before adding and for empty before removing; a stack has one pointer and adds and removes at the same end, a queue has two pointers and adds at one end and removes at the other.

Stack

A **stack** 栈 works in **LIFO** 后进先出 order (Last In, First Out). Operations: **push** 入栈 (add to the top), **pop** 出栈 (remove from the top), peek (look at the top), and tests for empty/full. Uses: undo history, function-call return addresses, expression parsing, backtracking.



Push and pop change the top pointer; the base pointer stays put

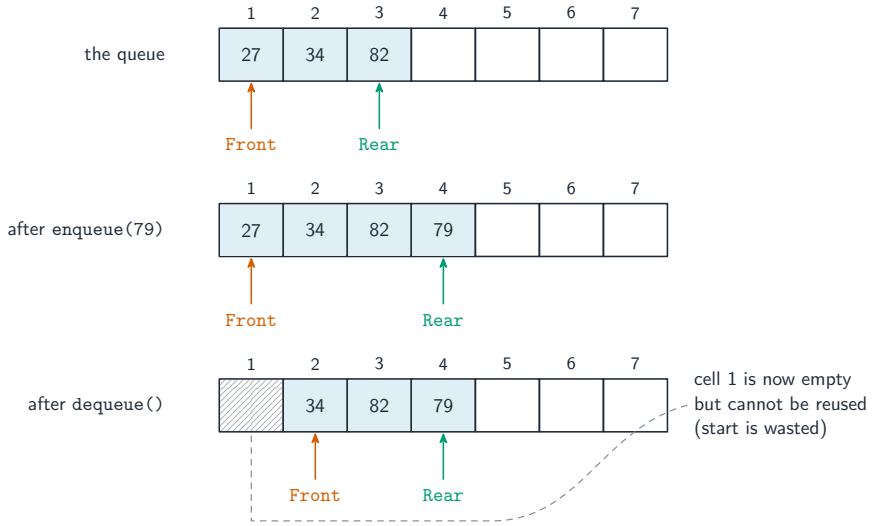
Worked example. A stack of characters holds, from the bottom, `'P'`, `'N'`, `'Z'`, `'X'`, `'Y'`, `'W'`, with the top-of-stack pointer at `'W'` (memory location 202 of 200–207). The operations `POP`, `POP`, `PUSH 'A'`, `PUSH 'B'`, `POP` are performed. What is on the stack, and where does the pointer point?

The two pops remove `'W'` then `'Y'`; the pushes add `'A'` then `'B'` in their places; the last pop removes `'B'`. The stack now holds `'P'`, `'N'`, `'Z'`, `'X'`, `'A'` and the pointer is at `'A'`, location 203. The value that has been on the stack longest is the bottom item, `'P'`; at most five further pops are possible before the stack is empty, and a pop on an empty stack is an error, which is why `Pop()` tests for empty first. A `Push()` function that returns `TRUE` on success first tests whether the pointer is at the top of the array (full) and returns `FALSE` if so. The array elements need no initialising before use, because the pointer alone says which elements are in use.

*A pile of books is a stack you can see. You can only add or take a book from the **top**, so the last one you put on is the first one you take off — that is exactly LIFO*

Queue

A **queue** 队列 works in **FIFO** 先进先出 order (First In, First Out). Operations: **enqueue** 入队 (add to the rear), **dequeue** 出队 (remove from the front), and tests for empty/full. Uses: print spooling, scheduling, breadth-first search, buffering.



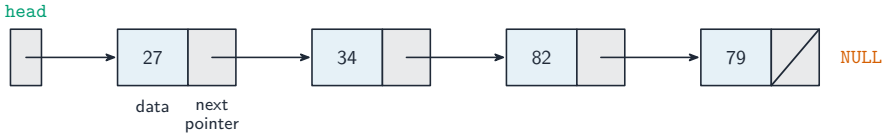
Enqueue adds at the rear; dequeue removes from the front

To **describe adding** an item: check that the queue is not full; store the item at the position given by the end-of-queue pointer; increment the end pointer (and the count). To **describe removing**: check that the queue is not empty; read the item at the front pointer; increment the front pointer (and decrement the count). State the convention you use: if the end pointer marks the next free space, front and end pointers being equal means the queue is **empty**; if it marks the last item, equal pointers mean one item. In a linear queue the front pointer only ever moves forward, so cells behind it are wasted; that is what the circular queue below fixes. The two features of a queue to state: items are added at the rear and removed from the front, so the first item added is the first removed.

*A line of people is a queue you can see. You join at the **back** and are served from the **front**, so whoever waited longest is served first — that is exactly FIFO*

Linked list

A **linked list** 链表 stores data as a sequence of **nodes** 节点. Each node holds a value and a **pointer** 指针 to the next node; a head pointer marks the start, and the last node's pointer is a sentinel (e.g. `NULL`). Operations: insert, delete, search, and **traverse** 遍历 (visit each node in order). Its advantage over an array is cheap insertion/deletion (just adjust pointers); its disadvantage is slow random access (you must follow pointers from the head).



A linked list: each node points to the next

Adding a node in order (four marks): traverse the list from the head, following the pointers, until the node before the position is found (the last node whose value is smaller); take a free node and store the new value in it; set the new node's pointer to the address the previous node pointed to; set the previous node's pointer to the new node. If the new value belongs at the front, the head pointer is changed instead. **Deleting** a node: find the node before it, and set that node's pointer to the address the deleted node pointed to, so the list bypasses it; the freed node returns to the free list. Compared with a 1-D array, inserting or deleting in a linked list needs no shifting of the other items, and the list can grow until memory runs out; the cost is the extra pointer stored with every item, and that reaching the nth item means following n pointers, since there is no direct index.

Implementing ADTs using arrays

Stack using an array

Hold items in `Stack[1:MaxSize]` with an integer `Top` (0 when empty).

- `Push(x)`: if `Top = MaxSize` the stack is full (**overflow** 溢出); else `Top ← Top + 1`; `Stack[Top] ← x`.
- `Pop()`: if `Top = 0` the stack is empty (**underflow** 下溢); else return `Stack[Top]` and `Top ← Top - 1`.

Queue using a circular array

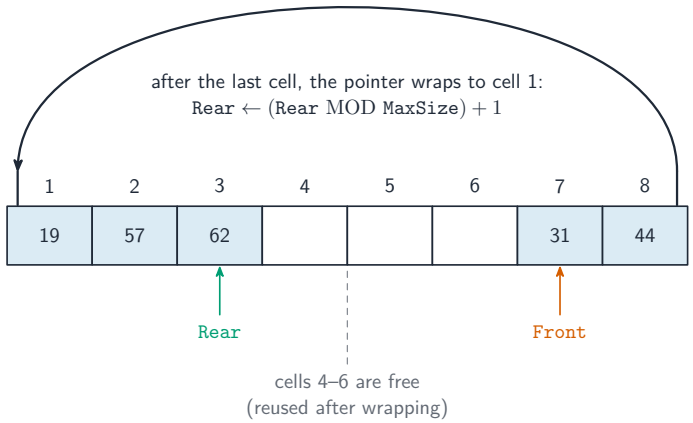
A simple queue lets `Front` and `Rear` march off the end, wasting the start. The fix is a **circular array** 循环数组 — when a pointer reaches `MaxSize` it wraps back to 1:

- `Enqueue(x)`: check full; else `Rear ← (Rear MOD MaxSize) + 1`; `Queue[Rear] ← x`.
- `Dequeue()`: check empty; else return `Queue[Front]` and `Front ← (Front MOD MaxSize) + 1`.

Track a separate count to tell empty from full.

The algorithm for the end pointer, in words: if the count equals the size, report that the queue is full and stop; otherwise add one to the end pointer; if it is now past the last index, set it to the first index; store the item there and add one to the count. The declarations that a five-mark "describe the declaration and initialisation" answer lists: the array with its size and element type; a front pointer and an end pointer, both initialised to the first index (or the front to the first index and the end to the next free space); and a count of items, initialised to 0.

For example, with `MaxSize = 6`: if `Rear = 5`, then `(5 MOD 6) + 1 = 6`, so the next item goes in cell 6; if `Rear = 6`, then `(6 MOD 6) + 1 = 1`, so the pointer **wraps back** to cell 1.



A circular queue wraps the pointers back to the start of the array

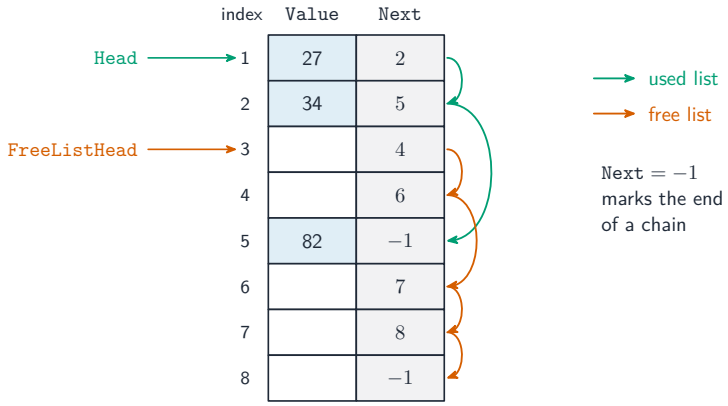
Linked list using an array

Use an array of records, each with a `Next` index:

```
TYPE TNode
    DECLARE Value : INTEGER
    DECLARE Next : INTEGER     // index of the next node, or -1 for end
ENDTYPE

DECLARE Nodes : ARRAY[1:MaxSize] OF TNode
DECLARE Head : INTEGER         // index of first node, -1 if empty
DECLARE FreeListHead : INTEGER // first available free node
```

A **free list** 空闲列表 chains the unused slots, just as the data list chains its used ones. To insert: take a slot from `FreeListHead`, set the new node's value and `Next`, and update the previous node's `Next` (or `Head`). To delete: unlink the node and return its slot to the free list. This gives the flexibility of a linked structure with the static allocation of an array.



A linked list stored in an array: a data array and a pointer array

Worked example. A linked list is held in a `Data` array and a `Pointer` array, with `Start` pointing to index 1. The list is $1 \to 3 \to 4$ (index 1 holds `D40`, index 3 holds `D32`, index 4 holds `D11`, whose pointer is $\emptyset$); the free list starts at index 2 and continues $2 \to 5$. Insert `D6` between `D32` and `D11`.

Take the first free node, index 2, and set `FreeStart` to its pointer, 5; store `D6` in `Data[2]`; set `Pointer[2]` to the value `Pointer[3]` held, which is 4; set `Pointer[3]` to 2. The list reads $1 \to 3 \to 2 \to 4$ and the free list is $5 \to \emptyset$. The answer to "how can the linked list be implemented" is exactly these parts: an array (or array of records)

for the data, a parallel array for the pointers holding indices, a start pointer, a free-list pointer and a null value such as -1 for the end.

Worked example. A circular queue is held in an array of size 5 (indices 0 to 4) with `Front = 3`, `Rear = 3` and one item stored. Two items are added, then two are removed. Where are the pointers, and why use a *circular* queue at all? Every move uses `(pointer + 1) MOD size`, so the pointers **wrap**. Adding twice moves `Rear`: $3 \to 4$, then $4 \to 0$ (because $(4 + 1) \bmod 5 = 0$), so `Rear = 0` and three items are stored. Removing twice moves `Front` the same way: $3 \to 4$, then $4 \to 0$, leaving `Front = 0` and one item. The wrap is the whole point: in a **linear** array queue the pointers march to the end and the freed space at the front is wasted even when the queue is empty. Remember a queue removes at the **Front** and adds at the **Rear** - a stack uses one pointer for both.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly.

Term	Definition
record	a data structure that holds a set of data items (fields) of different data types under one identifier
array	a data structure that holds a fixed number of elements of the same data type under one identifier, each accessed by an index
index	the number that identifies one element of an array
upper bound, lower bound	the largest and smallest valid index of an array
text file	a file that stores data as lines of characters, which a program reads and writes one line at a time
abstract data type	a collection of data together with a set of operations on that data
stack	a list in which items are added to and removed from the same end, the top, so the last item added is the first removed (LIFO)
queue	a list in which items are added at the rear and removed from the front, so the first item added is the first removed (FIFO)
linked list	a list in which each node holds a data item and a pointer to the next node, with a start pointer to the first node
pointer	a variable that holds the address (or index) of a node or of a position in a structure
linear search	checking each element in turn from the first until the target is found or the end is reached

Term	Definition
bubble sort	repeated passes through the array comparing adjacent pairs and swapping those out of order, until a pass makes no swaps

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).

Common mistakes

- A record declaration without `ENDTYPE`, or fields without types. Every field is a `DECLARE` line with a type.
- Reading past the end of a file, or writing with `WRITE` when the file must keep its contents. Test `EOF` before each read; use `APPEND` to add.
- Writing a number to a text file without converting it. A file holds strings: `NUM_TO_STR` out, `STR_TO_NUM` back.
- Forgetting the checks. `Push` and enqueue test for full first; `Pop` and dequeue test for empty first, and the answer says so.
- Losing the rest of the list when inserting a node. Set the new node's pointer to the old next node before changing the previous node's pointer.
- A linear search that never says "not found". Initialise the position to -1 and test it after the loop.