

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").

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.

record TStockItem

ItemID : INTEGER
Category : STRING
ItemCost : REAL
InStock : BOOLEAN

one name holds
four fields of
different types

reach one field: Item1.Category

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.

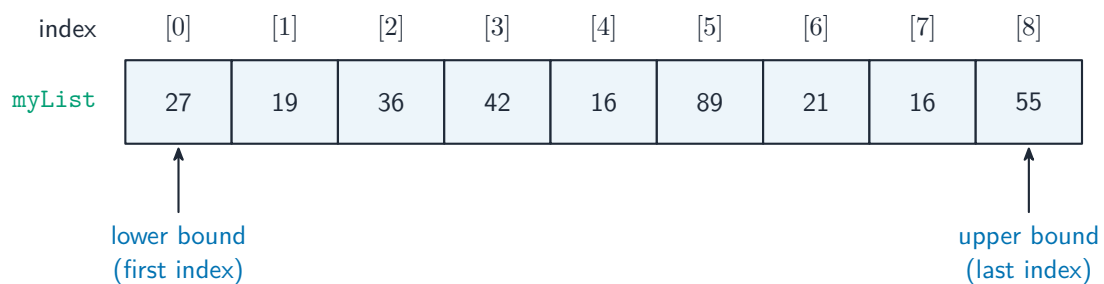
- **element** 元素—one item in the array.
- **bounds** 边界—the lowest and highest valid indices.
- **dimension** 维度—1-D (a list), 2-D (a table), etc.

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 $\times$ columns).

	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)

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

Common operations

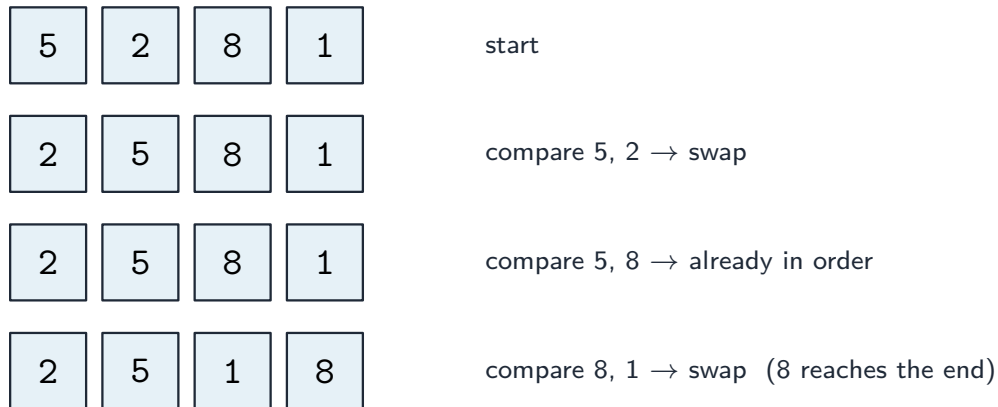
A **linear search** 线性查找 checks each element until found:

```
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
```

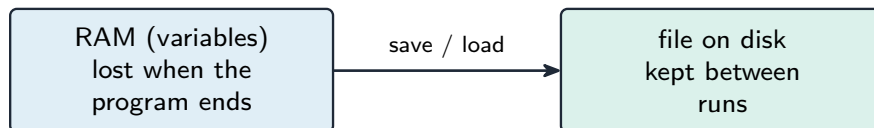
A **bubble sort** 冒泡排序 puts an array in order: pass through it comparing each adjacent pair and **swapping** any that are out of order; repeat the passes until one pass makes no swaps.



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"
  
```

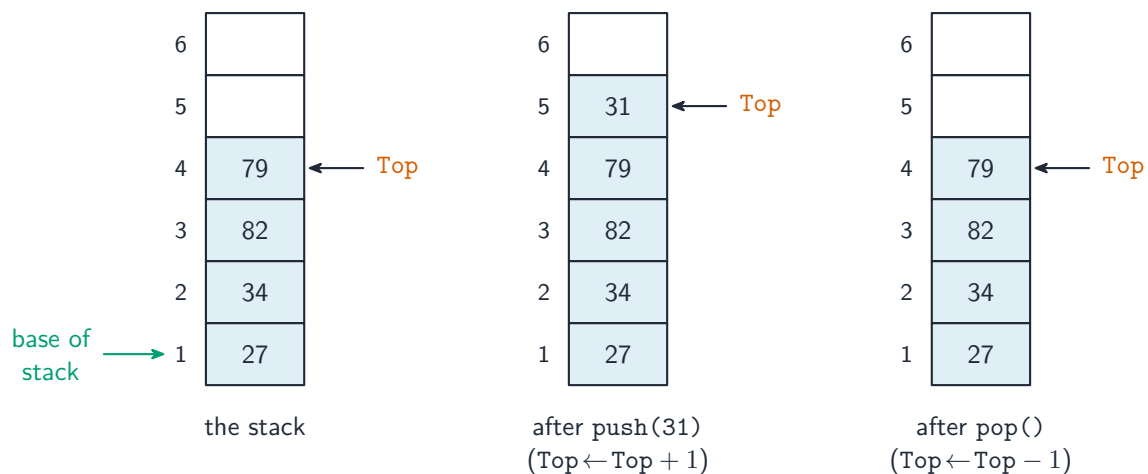
Always **close** every file —otherwise buffered writes may be lost and other programs may be locked out.

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.

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

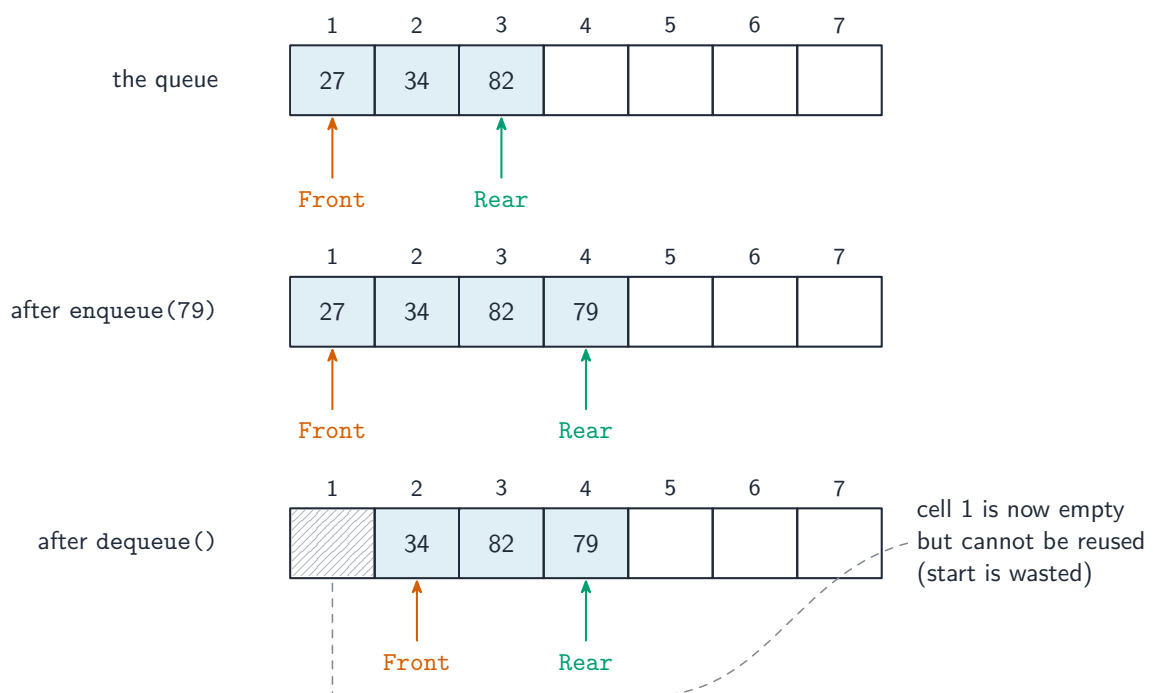


*A pile of books is a stack you can see. You can only add or take a book from the **top**, so the last one you put on is the first one you take off—that is exactly **LIFO***

Image: Alexander Grey, CC0 (commons.wikimedia.org)

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

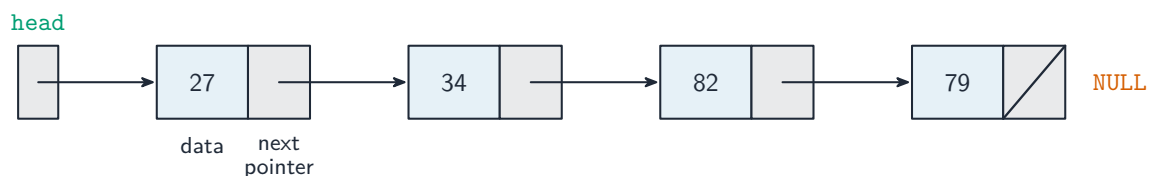


*A line of people is a queue you can see. You join at the **back** and are served from the **front**, so whoever waited longest is served first—that is exactly **FIFO***

Image: Diliff, CC BY-SA 3.0 (commons.wikimedia.org)

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

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`.

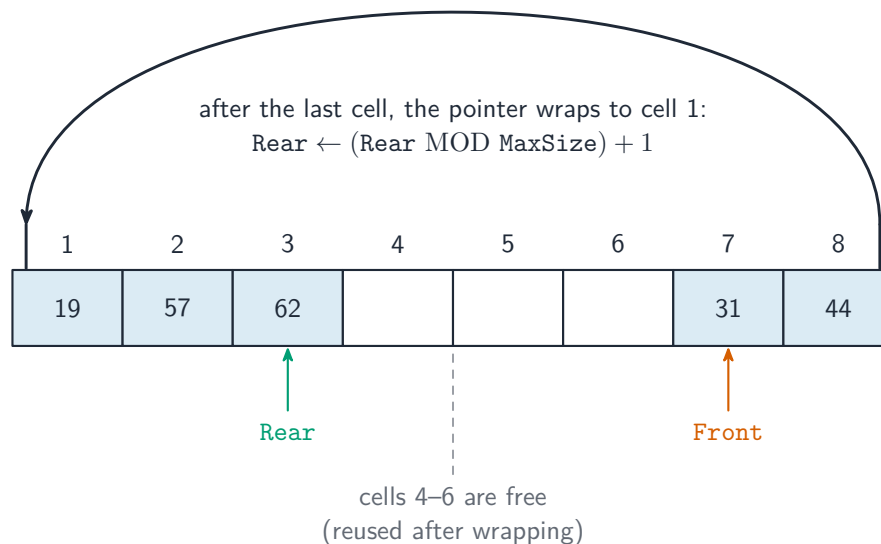
Queue using a circular array

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.

For example, with $\text{MaxSize} = 6$: if $\text{Rear} = 5$, then $(5 \bmod 6) + 1 = 6$, so the next item goes in cell 6; if $\text{Rear} = 6$, then $(6 \bmod 6) + 1 = 1$, 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:

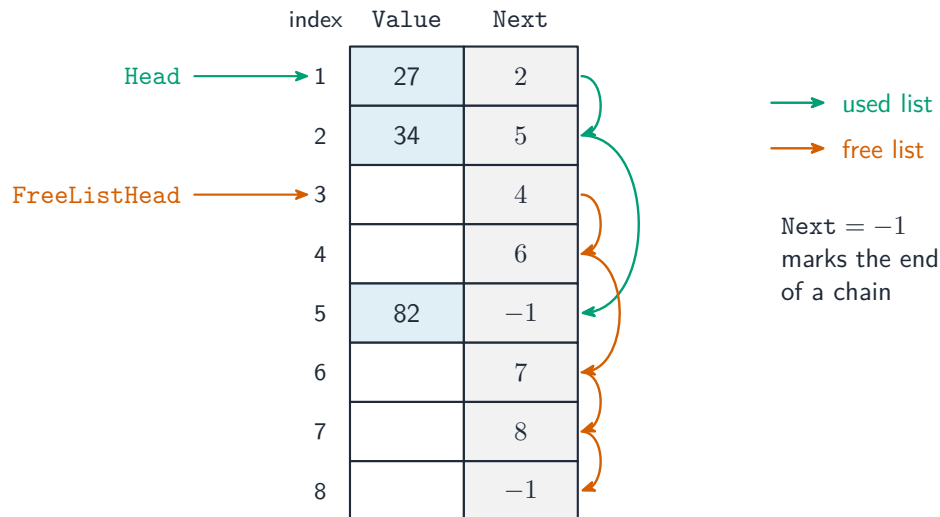
```

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 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 → 4, then 4 → 0 (because $(4 + 1) \bmod 5 = 0$), so **Rear** = 0 and three items are stored. Removing twice moves **Front** the same way: 3 → 4, then 4 → 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.

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).