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 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.

Arrays

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

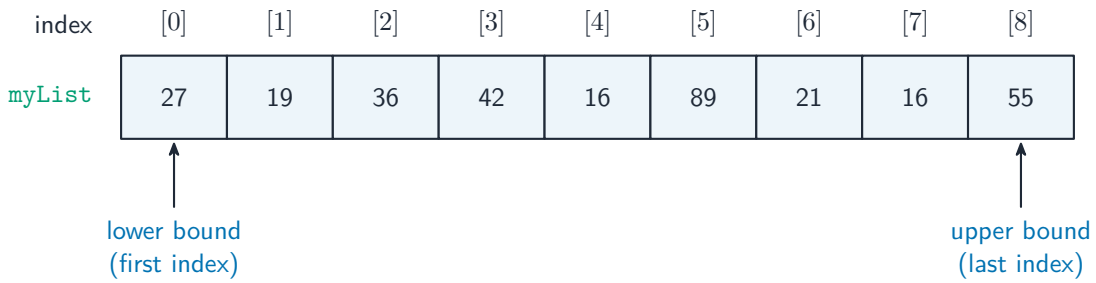
- **element** 元素—one item in the array.
- **bounds** 边界—the lowest and highest valid indices.
- **dimension** 维度—1-D (a list), 2-D (a table), etc.

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
```

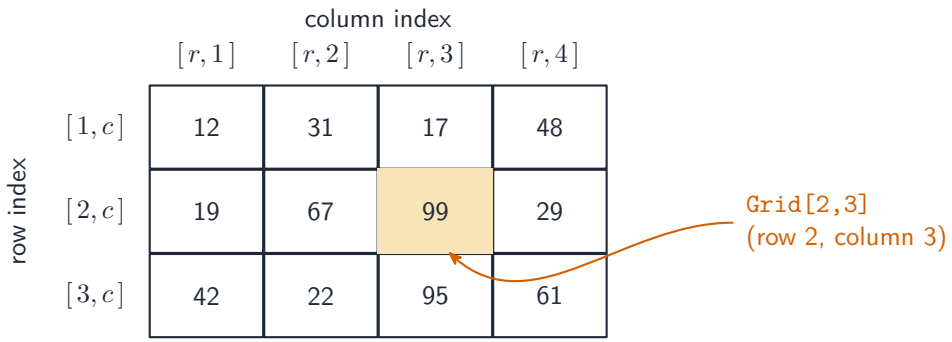


A 1-D array (a list) with indices and bounds

2-D arrays

```
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
```

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.

A **text file** 文本文件 is lines of readable characters. 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.

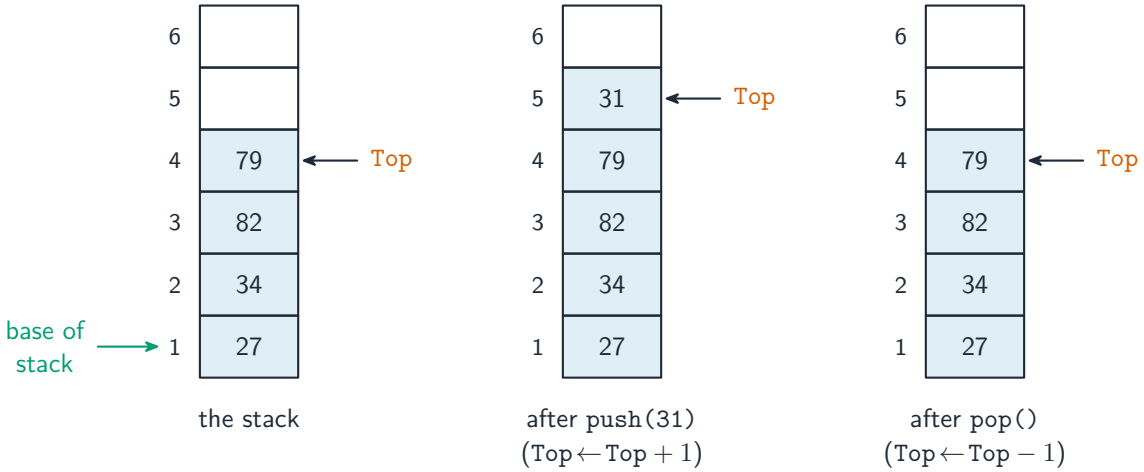
Abstract Data Types (ADTs)

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

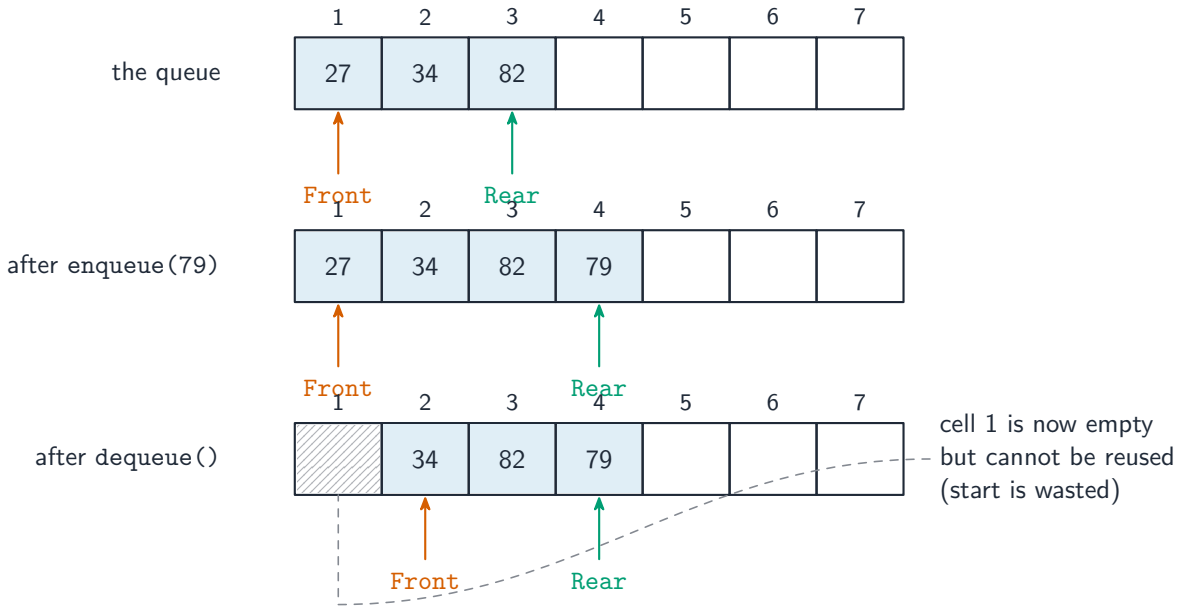
A **stack** 栈 works in **LIFO** 后进先出 order (Last In, First Out). Operations: **push** 入栈 (add to the top), **pop** 出栈 (remove from the top), peek (look at the top), and tests for empty/full. Uses: undo history, function-call return addresses, expression parsing, backtracking.



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

Queue

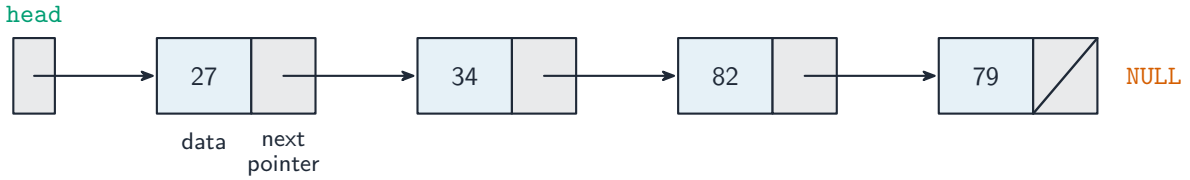
A **queue** 队列 works in **FIFO** 先进先出 order (First In, First Out). Operations: **enqueue** 入队 (add to the rear), **dequeue** 出队 (remove from the front), and tests for empty/full. Uses: print spooling, scheduling, breadth-first search, buffering.



Enqueue adds at the rear; dequeue removes from the front

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).

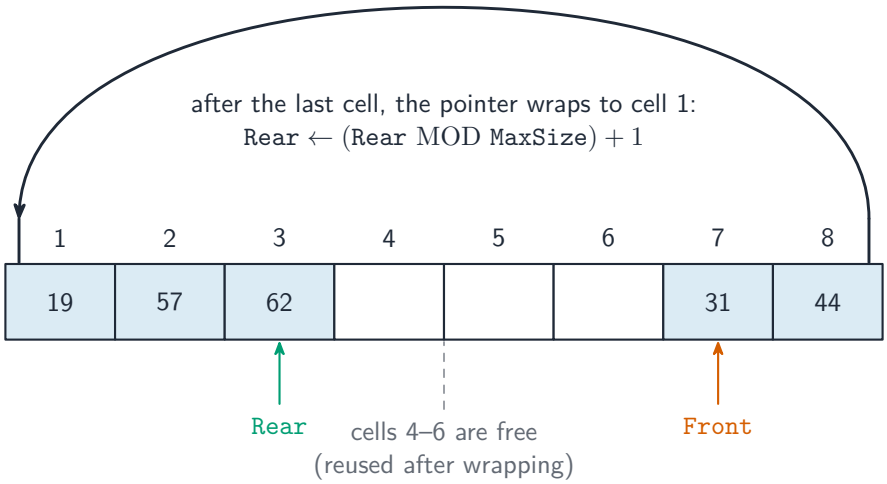
- `Push(x)`: if `Top = MaxSize` the stack is full (**overflow** 溢出); else `Top ← Top + 1`; `Stack[Top] ← x`.
- `Pop()`: if `Top = 0` the stack is empty (**underflow** 下溢); else return `Stack[Top]` and `Top ← Top - 1`.

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.



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

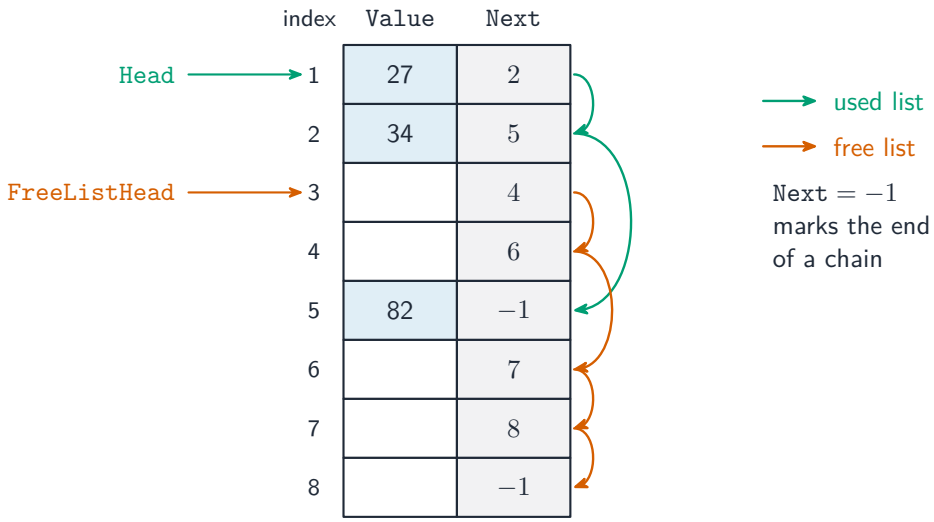
Linked list using an array

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