栈、队列与链表

Ref As Pseudocode

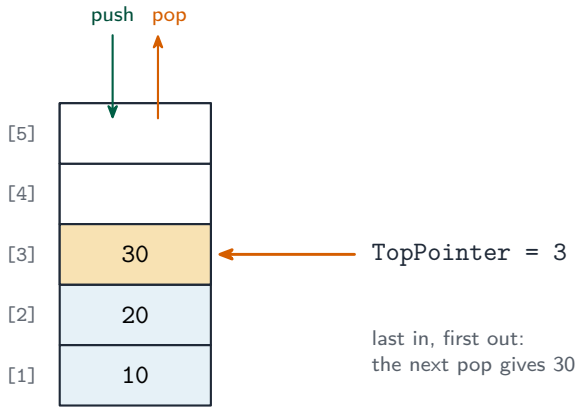
用数组实现栈

抽象数据类型（abstract data type，ADT）是一组数据，加上允许对这些数据进行的操作。使用 ADT 的人只需要知道各个操作**做什么**，不需要知道它们内部是怎么实现的。在 AS 阶段，你要用**数组**搭建三种 ADT：栈、队列和链表。

栈（stack）是**后进先出**（LIFO）的：最后放进去的项目最先被取出，就像一摞盘子。

- **压入**（push）：把一个项目放到顶部。
- **弹出**（pop）：取走顶部的项目。

数组存放各个项目，变量 `TopPointer` 存放顶部项目的索引。栈为空时，`TopPointer` 为 0。



用数组实现的栈：*TopPointer* 指向最后压入的项目

exercises.lol exercises.com.cn

```
CONSTANT MaxSize = 5
DECLARE Stack : ARRAY[1:5] OF INTEGER
DECLARE TopPointer : INTEGER

PROCEDURE Push(Item : INTEGER)
   IF TopPointer = MaxSize THEN
      OUTPUT "Stack full: cannot push ", Item
   ELSE
      TopPointer ← TopPointer + 1
      Stack[TopPointer] ← Item
   ENDIF
ENDPROCEDURE

FUNCTION Pop() RETURNS INTEGER
   DECLARE Item : INTEGER
   IF TopPointer = 0 THEN
      OUTPUT "Stack empty"
      RETURN -1
   ELSE
      Item ← Stack[TopPointer]
      TopPointer ← TopPointer - 1
      RETURN Item
   ENDIF
ENDFUNCTION

TopPointer ← 0
CALL Push(10)
CALL Push(20)
CALL Push(30)
OUTPUT "Popped ", Pop()
OUTPUT "Popped ", Pop()
CALL Push(40)
OUTPUT "Popped ", Pop()
```

输出依次是 30、20、40：取出的永远是还留在栈里的**最新**项目。

- `Push` 先检查栈是否**已满**；`Pop` 先检查栈是否**为空**。评分标准会看这些检查。
- `Pop` 不需要清除数组里的旧值。把 `TopPointer` 往下移就够了：那个位置会被下一次 `Push` 覆盖。

常见错误

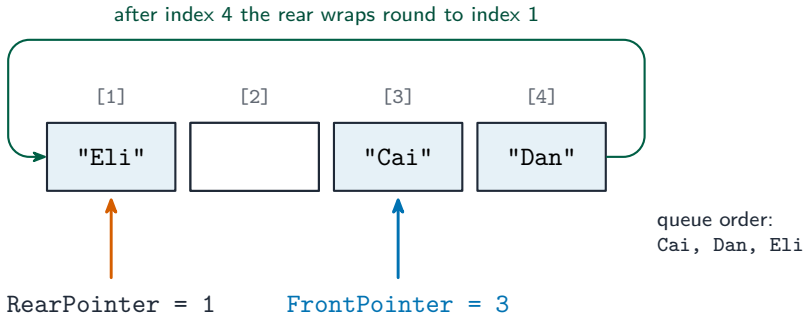
- 修改指针的顺序错了。在 `Push` 中，先增加 `TopPointer` **再**存值；在 `Pop` 中，先取出项目**再**减小 `TopPointer`。
- 忘记做“已满”或“为空”的检查。

用数组实现循环队列

队列（queue）是**先进先出**（FIFO）的：项目按到达的顺序离开，就像排队的人。

- **入队**（enqueue）：在**队尾**添加一个项目。
- **出队**（dequeue）：从**队头**取走一个项目。

如果队列只是在数组中一直往前移动，即使数组开头已经空出位置，它也很快会跑出数组末尾。**循环队列**（circular queue）解决了这个问题：指针越过最后一个索引时，会**绕回**索引 1。



循环队列：指针越过最后一个索引后回到 *1*

这个队列使用三个变量：

- `FrontPointer`：队头项目的索引；
- `RearPointer`：队尾项目的索引；
- `NumberInQueue`：项目的个数，有了它，判断”已满”和”为空”都很容易。

```
CONSTANT MaxSize = 4
DECLARE Queue : ARRAY[1:4] OF STRING
DECLARE FrontPointer : INTEGER
DECLARE RearPointer : INTEGER
DECLARE NumberInQueue : INTEGER

PROCEDURE Enqueue(Item : STRING)
   IF NumberInQueue = MaxSize THEN
      OUTPUT "Queue full"
   ELSE
      RearPointer ← RearPointer + 1
      IF RearPointer > MaxSize THEN
         RearPointer ← 1
      ENDIF
      Queue[RearPointer] ← Item
      NumberInQueue ← NumberInQueue + 1
   ENDIF
ENDPROCEDURE

FUNCTION Dequeue() RETURNS STRING
   DECLARE Item : STRING
   IF NumberInQueue = 0 THEN
      RETURN "EMPTY"
   ELSE
      Item ← Queue[FrontPointer]
      FrontPointer ← FrontPointer + 1
      IF FrontPointer > MaxSize THEN
         FrontPointer ← 1
      ENDIF
      NumberInQueue ← NumberInQueue - 1
      RETURN Item
   ENDIF
ENDFUNCTION

FrontPointer ← 1
RearPointer ← 0
NumberInQueue ← 0
CALL Enqueue("Ann")
CALL Enqueue("Ben")
CALL Enqueue("Cai")
OUTPUT Dequeue()
OUTPUT Dequeue()
CALL Enqueue("Dan")
CALL Enqueue("Eli")
CALL Enqueue("Fay")
OUTPUT Dequeue(), " ", Dequeue(), " ", Dequeue(), " ", Dequeue()
```

`Eli` 又被放进了索引 1，因为 `RearPointer` 绕回来了。队列仍然按到达的顺序把名字送出来。

绕回也可以用 `MOD` 来写：`RearPointer ← (RearPointer MOD MaxSize) + 1`。两种写法都可以。

常见错误

- 忘记绕回，指针跑出了最后一个索引。
- 让 `RearPointer` 从 1 开始。在上面的写法中它从 0 开始，第一次 `Enqueue` 时才移到 1。

exercises.lol exercises.com.cn

用数组实现链表

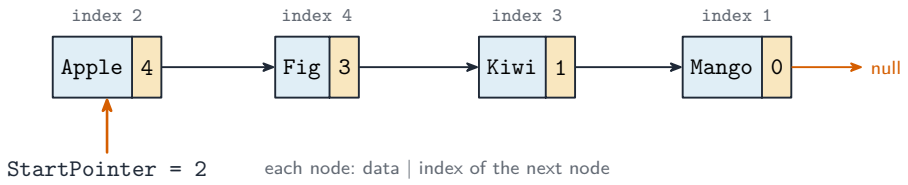
链表（linked list）是这样一种列表：每个项目叫作一个**节点**（node），它既存放数据，**也**存放一个指向下一个节点的**指针**（pointer）。各个项目不必按顺序存放：顺序由指针决定。

用数组实现链表需要：

- 一个存放数据的数组，以及一个同样大小的数组 `NextPointer`；
- `StartPointer`：**第一个**节点的索引；
- 一个**空指针**（null pointer），这里用 0，表示链表的结尾。

索引	Data	NextPointer
1	"Mango"	0
2	"Apple"	4
3	"Kiwi"	1
4	"Fig"	3

当 `StartPointer = 2` 时，这个链表读作 Apple → Fig → Kiwi → Mango，是按字母顺序排列的，尽管数组本身并不是。



从 *StartPointer* 出发沿着指针走，就得到链表的顺序

exercises.lol exercises.com.cn

沿着指针走

要访问每个节点，从 `StartPointer` 开始，沿着每个 `NextPointer` 一直走，直到遇到空指针：

```
CONSTANT NullPointer = 0
DECLARE Data : ARRAY[1:4] OF STRING
DECLARE NextPointer : ARRAY[1:4] OF INTEGER
DECLARE StartPointer : INTEGER
DECLARE Current : INTEGER
Data[1] ← "Mango"
NextPointer[1] ← 0
Data[2] ← "Apple"
NextPointer[2] ← 4
Data[3] ← "Kiwi"
NextPointer[3] ← 1
Data[4] ← "Fig"
NextPointer[4] ← 3
StartPointer ← 2

Current ← StartPointer
WHILE Current <> NullPointer
   OUTPUT Data[Current]
   Current ← NextPointer[Current]
ENDWHILE
```

添加与删除节点

添加或删除节点时，**只改指针**，数据不用移动。

- **删除** “Kiwi”：让它前面的节点（“Fig”，索引 4）指向它后面的节点（“Mango”，索引 1），即 `NextPointer[4] ← 1`。
- **添加**新节点：把它的数据存进一个空闲元素，让它指向应该排在它后面的节点，再让它前面的节点指向这个新节点。

没有使用的元素通常放在第二个链表里，叫作**空闲链表**，它有自己的 `FreeListPointer`。带空闲链表的完整插入和删除算法属于 A-Level（Paper 4）的内容；在 AS 阶段，你应该能沿着指针走，并说出哪些地方发生了变化。

练习

1. 在上面的表格中，链表的**最后**一个节点在哪个索引？你是怎么知道的？

2. 使用上面的队列，`"Fay"` 入队之后，`RearPointer` 的值是多少？

<details> <summary> 参考答案 </summary>

1. 索引 1（`"Mango"`）。它的 `NextPointer` 是 0，也就是空指针，所以它后面没有节点了。
2. `RearPointer` 是 2。前三次入队把它移到 1、2、3；（两次出队之后）下一次移到 4；然后为 `"Eli"` 绕回到 1，再为 `"Fay"` 移到 2。

</details>

常见错误

- 按**索引顺序**（1、2、3、4）读链表，而不是沿着指针走。
- 把两种 ADT 弄混：栈从添加的**同一端**取出；队列从**另一端**取出。
- 用 `Current ← Current + 1` 在链表中前进。下一个节点是 `NextPointer[Current]`，它通常在别的位置。

exercises.lol exercises.com.cn