Stacks, queues and linked lists

Ref As Pseudocode

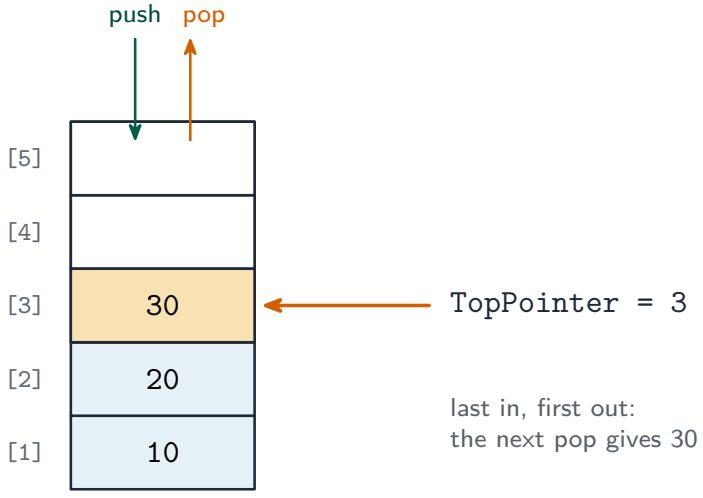
A stack in an array

An **abstract data type** (ADT) is a collection of data together with the operations allowed on it. The user of an ADT only needs to know **what** the operations do, not how they work inside. At AS you build three ADTs from **arrays**: a stack, a queue and a linked list.

A **stack** is **last in, first out** (LIFO): the last item added is the first item taken off, like a pile of plates.

- **Push** push : add an item to the top.
- **Pop** pop : remove the item at the top.

The array holds the items, and a variable `TopPointer` holds the index of the top item. When the stack is empty, `TopPointer` is 0.



A stack in an array: TopPointer marks the last item pushed

```
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()
```

The output is 30, 20, then 40: always the **most recent** item that is still there.

- `Push` checks for a **full** stack first; `Pop` checks for an **empty** stack first. Mark schemes look for these checks.
- `Pop` does not need to clear the old value from the array. Moving `TopPointer` down is enough: that space will be written over by the next `Push`.

Common mistakes

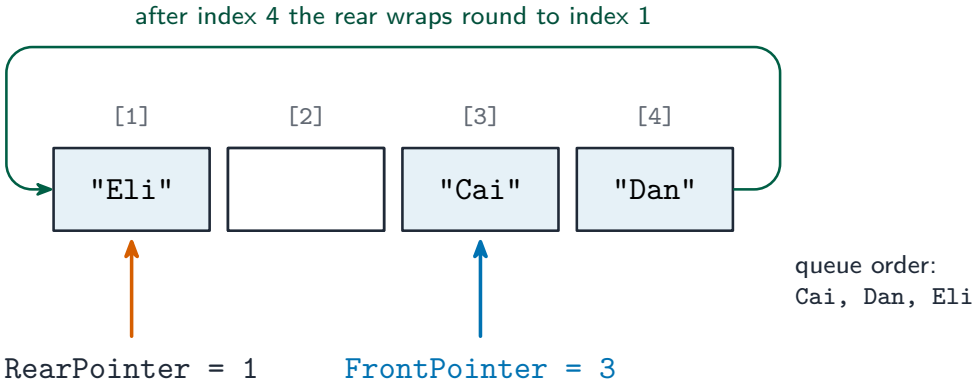
- Changing the pointer in the wrong order. In `Push`, increase `TopPointer` **then** store; in `Pop`, read the item **then** decrease `TopPointer`.
- Forgetting the full or empty check.

A circular queue in an array

A **queue** is **first in, first out** (FIFO): items leave in the order they arrived, like people waiting in a line.

- **Enqueue** enqueue : add an item at the **rear**.
- **Dequeue** dequeue : remove the item at the **front**.

If the queue just moved forward in the array, it would soon run off the end even with empty space at the start. A **circular queue** solves this: when a pointer passes the last index, it **wraps round** to index 1 again.



A circular queue: when a pointer passes the last index it goes back to 1

The queue uses three variables:

- `FrontPointer`: the index of the item at the front;
- `RearPointer`: the index of the item at the rear;
- `NumberInQueue`: how many items there are, which makes "full" and "empty" easy to test.

```
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
```


exercises.lol exercises.com.cn

```
      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` goes into index 1 again, because `RearPointer` wrapped round. The queue still gives the names back in the order they arrived.

The wrap-round can also be written with `MOD`: `RearPointer ← (RearPointer MOD MaxSize) + 1`. Both forms are fine.

Common mistakes

- Forgetting the wrap-round, so the pointer goes past the last index.
- Starting `RearPointer` at 1. With the version above, it starts at 0, so the first `Enqueue` moves it to 1.

A linked list in arrays

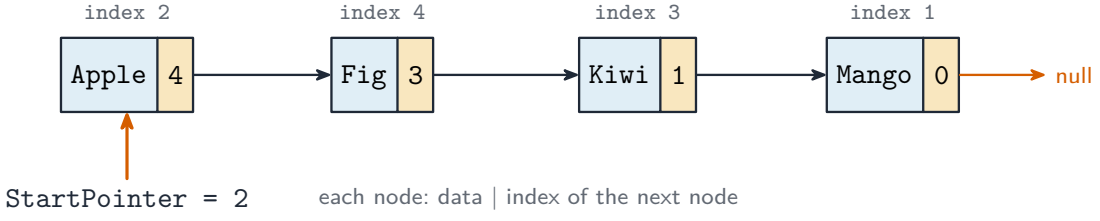
A **linked list** is a list where each item, called a **node**, holds its data **and** a **pointer** to the next node. The items do not have to be stored in order: the pointers give the order.

With arrays, a linked list needs:

- an array for the data, and an array `NextPointer` of the same size;
- `StartPointer`: the index of the **first** node;
- a **null pointer** null pointer, here 0, that marks the end of the list.

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

With `StartPointer = 2`, the list reads Apple → Fig → Kiwi → Mango, which is alphabetical order, although the array is not.



Following the pointers from StartPointer gives the list order

Following the pointers

To visit every node, start at `StartPointer` and follow each `NextPointer` until you reach the null pointer:

```
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
```

Adding and removing a node

To add or remove a node, you change **pointers only**; no data moves.

- **Remove** "Kiwi": make the node before it ("Fig", index 4) point to the node after it ("Mango", index 1). Set `NextPointer[4] ← 1`.
- **Add** a new node: store its data in a free element, point it to the node that should follow it, then point the node before it to the new node.

Unused elements are usually kept in a second list, the **free list**, with its own `FreeListPointer`. Full insert and delete algorithms with a free list are A-Level (Paper 4) content; at AS you should be able to follow the pointers and say what changes.

Practice

1. In the table above, which index holds the **last** node of the list? How do you know?

2. Using the queue above, what is the value of `RearPointer` after `"Fay"` is added?

<details> <summary>Model answer</summary>

1. Index 1 (`"Mango"`). Its `NextPointer` is 0, the null pointer, so no node comes after it.

2. `RearPointer` is 2. The six enqueues move it 1, 2, 3, then (after two dequeues) 4, then it wraps round to 1 for `"Eli"` and moves to 2 for `"Fay"`.

</details>

Common mistakes

- Reading the linked list in **index order** (1, 2, 3, 4) instead of following the pointers.
- Mixing up the two ADTs: a stack takes from the **same end** it adds to; a queue takes from the **other** end.
- Writing `Current ← Current + 1` to move along a linked list. The next node is `NextPointer[Current]`, which is usually somewhere else.