Exercise walk-through

Introduction to Abstract Data Types (ADT) ▪ 10.4

eXercise

You must be able to

- explain that an **ADT** is data **plus** the operations on it
- use a **stack** (LIFO), a **queue** (FIFO) and a **linked list**
- describe how each is implemented using an **array**

Method — What is an ADT

An **ADT** is a **collection of data** together with a **set of operations** on it. You use it through its operations only — **how** it is stored is hidden, so the implementation can change without breaking your code.

Method — Stack — LIFO (Last In, First Out)

- **push** — add to the **top**; **pop** — remove from the **top**.
- Stored in `Stack[1:MaxSize]` with an integer `Top` (0 = empty).
- **Push**: full when `Top = MaxSize` (**overflow**); else `Top ← Top + 1`, `Stack[Top] ← x`.
- **Pop**: empty when `Top = 0` (**underflow**); else return `Stack[Top]`, `Top ← Top - 1`.

Method — Queue — FIFO (First In, First Out)

- **enqueue** — add to the **rear**; **dequeue** — remove from the **front**.
- Stored in `Queue[1:MaxSize]` with a `Front` pointer, a `Rear` pointer and a `Count`.
- **Enqueue**: full when `Count = MaxSize`; else `Rear ← (Rear MOD MaxSize) + 1`, `Queue[Rear] ← x`, `Count ← Count + 1`.
- **Dequeue**: empty when `Count = 0`; else return `Queue[Front]`, `Front ← (Front MOD MaxSize) + 1`, `Count ← Count - 1`.
- The `MOD` makes it a **circular array**, so the pointers wrap round instead of running off the end.

Method — Linked list

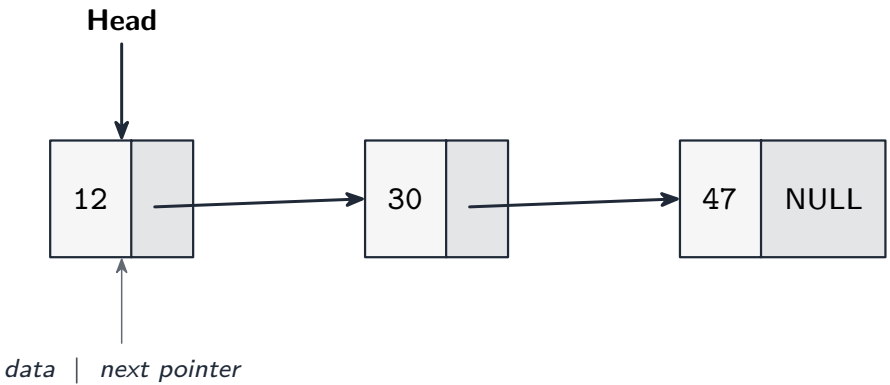
Each **node** holds a value and a **pointer** to the next node. `Head` marks the first node; the last node's pointer is `NULL`.
Traverse — follow the pointers from `Head` to `NULL`:

```
Current ← Head
WHILE Current <> NULL DO
    OUTPUT Current.Value
    Current ← Current.Next
ENDWHILE
```

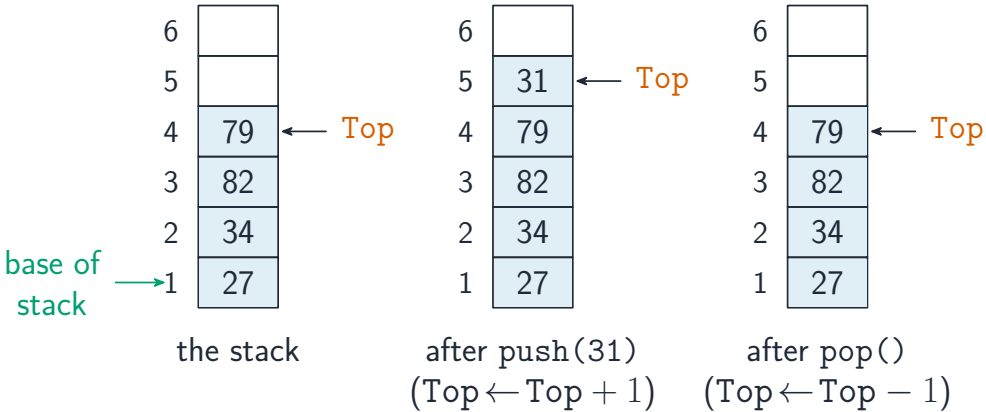
Method — Linked list

Advantage over an array: cheap insert/delete (adjust pointers). Disadvantage: slow random access (you must follow pointers from `Head`).

Method — Linked list



Method — Linked list



A stack is another abstract data type (push/pop)

Practice 2.1 ▪ 1 mark

State what an **Abstract Data Type** is.

Solution 2.1

Method: Linked list

Worked steps

A **collection of data together with the operations** that can be performed on it (implementation hidden) B1.

Practice 2.2 ▪ 2 marks

State the order rule of a **stack** and of a **queue** (use the terms LIFO / FIFO).

Solution 2.2

Worked steps

Stack = **LIFO** (Last In, First Out); Queue = **FIFO** (First In, First Out) B1 B1.

Practice 2.3 ▪ 2 marks

Name the stack operation that **adds** an item and the one that **removes** an item.

Solution 2.3

Worked steps

Add = **push**; remove = **pop** B1 B1.

Practice 2.4 ▪ 2 marks

In a linked list, state what the **Head** pointer identifies and what the **last node's** pointer holds.

Solution 2.4

Method: Linked list

Worked steps

Head identifies the **first node** in the list B1; the last node's pointer holds **NULL** (a null/sentinel pointer = end of list) B1.

3.1 (i)

```
IF Top = 10 THEN
    OUTPUT "Stack full (overflow)"
ELSE
    Top ← Top + 1
    Stack[Top] ← NewItem
ENDIF
```

Solution 2.4

M1 M1 M1 A1 *test 'Top = 10', overflow message, 'Top ← Top + 1', 'Stack[Top] ← NewItem'*

Exam-style 3.1 ▪ 4 marks

A stack is implemented as `Stack[1:10]` with an integer `Top`.
(i) Write pseudocode for `Push(NewItem)` that checks for **overflow**.
(ii) State what **underflow** means.

Solution 3.1

Method: Stack — LIFO (Last In, First Out)

Worked steps

(ii) Trying to **pop from an empty stack** (`Top = 0`) — there is nothing to remove B1.

Exam-style 3.2 ▪ 3 marks

The items `A`, `B`, `C`, `D` are pushed onto an empty stack in that order, then **two** items are popped. State the two items removed (**in order**) and which item is now on top.

Solution 3.2

Worked steps

Popped: **D**, then **C** (LIFO order) B1 B1; now on top: **B** B1.

Exam-style 3.3 ▪ 3 marks

A linked list stores nodes that each have a `Value` and a `Next` pointer.
(i) Describe how to **traverse** the list and output every value.
(ii) State **one** advantage of a linked list over an array.

Solution 3.3

Method: Linked list

Worked steps

(ii) Any one: items can be **inserted/deleted** without shifting others (just change pointers); the list can grow/shrink at run time B1.