

10.4.1 The circular queue, a linked list in arrays, and choosing the right ADT

Name: _____ Class: _____ Date: _____

Total: 29 marks

KEY WORDS queue 队列 • stack 栈 • linked list 链表 • circular queue 循环队列
• free list 空闲列表

Objective

Build the skills to answer the **abstract data type** 抽象数据类型 questions that sheet 10.4 does not reach. That sheet defines an ADT, gives the order rules of a **stack** 栈 and a **queue** 队列 and implements a stack in an array; this one covers the two implementations it leaves out and that the exam asks about most: the **circular queue** 循环队列 and its wrapping pointers, and a **linked list** 链表 held in arrays with a **free list** 空闲列表. It also covers **justifying** which ADT suits a given situation.

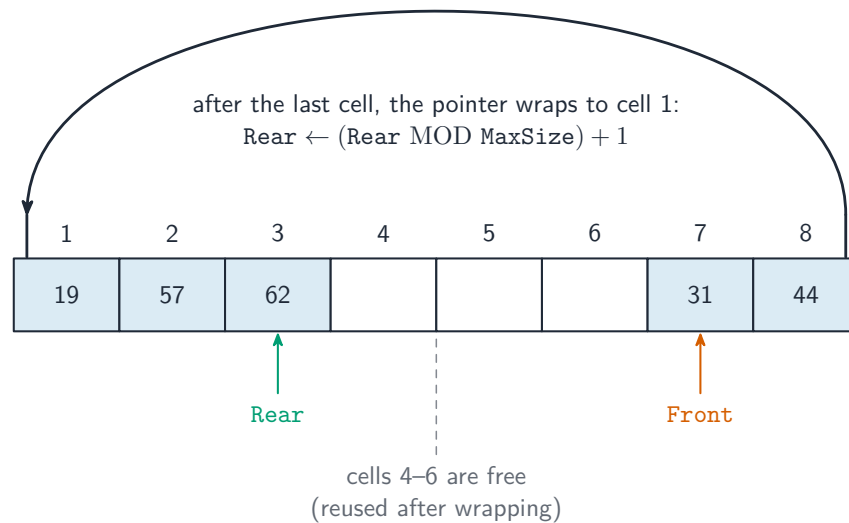
You must be able to:

- describe the key features of a stack, a queue and a linked list, and **justify** their use for a given situation
- describe how a queue, a stack and a linked list can be implemented using **arrays**
- use the pointers of a circular queue, including the wrap, and state the checks made before adding and removing
- add and delete a node in a linked list held in arrays, using the free list

1 Worked examples

Study these first. Each one shows a **method** that a question later on this sheet needs.

- The circular queue, and why it exists



A queue adds at the **rear** and removes from the **front**, so both pointers only ever move **forward**. In a plain linear queue the front pointer marches up the array and the cells behind it can never be used again: after a few hundred operations the queue reports itself full with most of the array empty.

A **circular queue** fixes that by wrapping a pointer back to the start when it passes the end:

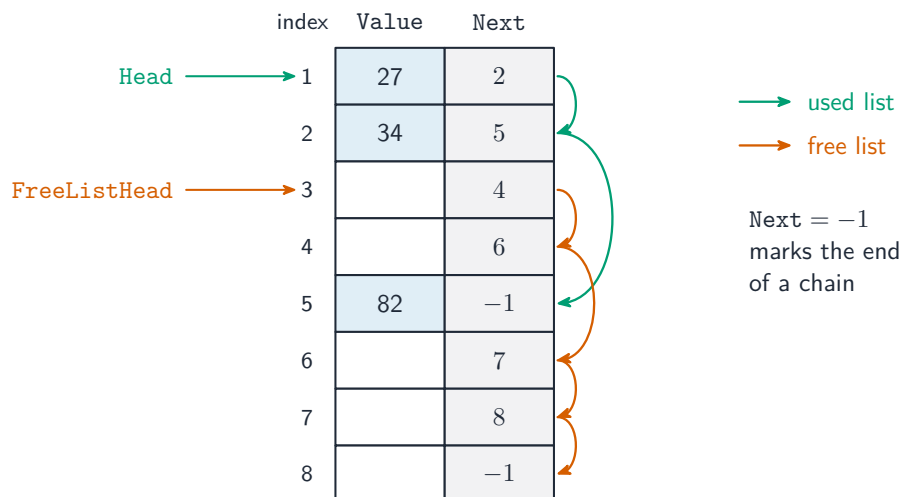
$$\text{Rear} \leftarrow (\text{Rear} \text{ MOD } \text{MaxSize}) + 1$$

With $\text{MaxSize} = 6$: if **Rear** is 5 then $(5 \bmod 6) + 1 = 6$, so the next item goes in cell 6; if **Rear** is 6 then $(6 \bmod 6) + 1 = 1$, and the pointer **wraps back** to cell 1. The same formula moves **Front**.

The two checks, and they are marks:

- before **adding** (enqueue), check the queue is **not full** – there is an unused element
- before **removing** (dequeue), check the queue is **not empty** – there is at least one item
- Keep a **NumberInQueue** counter, and both checks are one comparison: full when it equals **MaxSize**, empty when it is 0. Without a counter, **Front = Rear** cannot tell full from empty, because both happen when the pointers meet. Say which convention you are using; the mark is for being explicit.

■ A linked list held in arrays



Two parallel arrays: `Data[i]` holds a value and `Pointer[i]` holds the **index** of the next node, with 0 meaning "no next node". **Start** holds the index of the first node.

The unused slots are not left loose – they are chained into a **free list** of their own, headed by `FreeListHead`, exactly as the data is chained. That gives the flexibility of a linked structure with the fixed allocation of an array.

Worked example. *Six slots. `Start = 2`, and the list is $2 \rightarrow 5 \rightarrow 1$ holding **Ann**, **Ben**, **Cal**, with `Pointer[1] = 0`. `FreeListHead = 3`, and the free list is $3 \rightarrow 4 \rightarrow 6$. Insert **Bea** between **Ann** and **Ben**.*

1. **Take a slot from the free list:** index 3. Move the head on – `FreeListHead` $\leftarrow$ `Pointer[3]`, which is 4.
2. **Store the value:** `Data[3]` $\leftarrow$ "Bea".
3. **Point the new node at what the previous one pointed at:** `Pointer[3]` $\leftarrow$ `Pointer[2]`, which is 5.
4. **Point the previous node at the new one:** `Pointer[2]` $\leftarrow$ 3.

The list is now $2 \rightarrow 3 \rightarrow 5 \rightarrow 1$ and the free list is $4 \rightarrow 6$.

- **Step 3 must come before step 4.** Setting `Pointer[2]` first destroys the address of **Ben**, and the rest of the list is lost.

Deleting Ben, at index 5: find the node **before** it (index 3), then

- `Pointer[3]  $\leftarrow$  Pointer[5]`, which is 1, so the list bypasses it,
- return the slot: `Pointer[5]  $\leftarrow$  FreeListHead`, then `FreeListHead  $\leftarrow$  5`.

The list is $2 \rightarrow 3 \rightarrow 1$ and the free list is $5 \rightarrow 4 \rightarrow 6$. Nothing is erased – a node is "deleted" by nothing pointing at it any more.

■ Choosing the right ADT

Situation	ADT	Because
undo in an editor; return addresses of nested calls; reversing something	stack	the last thing stored is the first needed back (LIFO)
a print spooler; keystrokes waiting; jobs served in turn	queue	the first thing stored is the first served (FIFO)
a list that is often inserted into or deleted from in the middle, and whose length is not known	linked list	inserting or deleting only changes pointers , with no shifting of the other items

- A justification must name the **property**, not the structure: “a queue, because the first job submitted must print first” earns the mark; “a queue, because it is a queue of jobs” does not.
- Against a 1-D array, a linked list inserts and deletes without shifting anything and can grow until memory runs out; the costs are a pointer stored with every item and having to follow n pointers to reach the n th item, because there is no direct index.

2 Practice

Now use the methods above.

2.1 State **one** key feature of a **stack** and **one** key feature of a **queue**. [2]

2.2 A circular queue is held in an array with **MaxSize** = 6. Give the statement that moves **Rear** on by one, and state the value it gives when **Rear** is 6. [2]

2.3 State what is meant by the **free list** in a linked list held in arrays. [2]

2.4 State the check that must be made **before** adding an item to a queue, and the check that must be made **before** removing one. [2]

2.5 Name the most suitable ADT for each situation. [2]

- (a) storing the return addresses of nested procedure calls
- (b) holding documents waiting to be printed

3 Exam-style questions

3.1 A circular queue is held in $Q[1:6]$, with `Front` = 4, `Rear` = 6 and `NumberInQueue` = 3.

- (a) An item is added. Give the new value of `Rear`, and state which cell the item is stored in. [2]

- (b) Two items are then removed. Give the new value of `Front` and of `NumberInQueue`. [2]

- (c) State how the program knows the queue is **full**, given that it keeps `NumberInQueue`. [1]

3.2 Explain why a **linear** queue held in an array wastes space, and how a **circular** queue avoids it. [3]

3.3 A linked list is held in `Data[1:6]` and `Pointer[1:6]`. `Start = 2`, the list is $2 \rightarrow 5 \rightarrow 1$, `Pointer[1] = 0`, and the free list is $3 \rightarrow 4 \rightarrow 6$ with `FreeListHead = 3`.

Describe the steps that insert a new value between the nodes at index 2 and index 5, giving the new value of every pointer that changes. [4]

3.4 Using the list as it is **after** the insertion in 3.3, describe the steps that **delete** the node at index 5, including what happens to its slot. [4]

3.5 A hospital records patients waiting for a doctor. Patients are seen in the order they arrive, but a patient may leave before being seen, and the number waiting is not known in advance.

Name a suitable ADT and **justify** your choice.

[3]

Solutions

Each answer shows the steps, then the **result**.

2.1

- a **stack** is **LIFO**: items are added and removed at the same end, the top, so the last item added is the first removed
- a **queue** is **FIFO**: items are added at the rear and removed from the front, so the first item added is the first removed

2.2

- $\text{Rear} \leftarrow (\text{Rear} \bmod 6) + 1$
- when **Rear** is 6: $(6 \bmod 6) + 1 = 1$, so the pointer wraps back to cell 1

2.3

- the free list is a chain of the array's **unused** slots, headed by its own pointer
- a slot is taken from it when a node is inserted and returned to it when a node is deleted, so the program always knows where there is room

2.4

- before adding: check the queue is **not full** – that there is an unused element
- before removing: check the queue is **not empty** – that there is at least one item

2.5

(a) a **stack**

(b) a **queue**

3.1

(a) $\text{Rear} \leftarrow (6 \bmod 6) + 1 = 1$, so **Rear** becomes **1**

- the item is stored in **cell 1**: the pointer has wrapped round to the start

(b) **Front** moves on twice: $(4 \bmod 6) + 1 = 5$, then $(5 \bmod 6) + 1 = 6$, so **Front** is **6**

- **NumberInQueue** was 4 after the addition, so after two removals it is **2**

(c) the queue is full when **NumberInQueue** equals **MaxSize**, that is 6

3.2

- in a linear queue both pointers only ever move **forward**, so once **Front** has passed a cell that cell can never be reused
- the queue therefore reports itself full when **Rear** reaches the end of the array, even though the cells before **Front** are empty
- a circular queue wraps a pointer back to cell 1 when it passes the last cell, using $(\text{Pointer} \bmod \text{MaxSize}) + 1$, so every cell is reused and the queue is full only when it really holds **MaxSize** items

3.3

- take the slot at `FreeListHead`, index **3**, and move the head on: `FreeListHead <- Pointer[3]`, which is **4**
- store the value in `Data[3]`
- `Pointer[3] <- Pointer[2]`, which is **5**, so the new node points at what the previous node pointed at
- `Pointer[2] <- 3`, so the previous node now points at the new one
- the list is $2 \rightarrow 3 \rightarrow 5 \rightarrow 1$ and the free list is $4 \rightarrow 6$
- the order matters: setting `Pointer[2]` before `Pointer[3]` destroys the address of the rest of the list

3.4

- find the node **before** index 5, which is index 3, by following the pointers from `Start`
- `Pointer[3] <- Pointer[5]`, which is **1**, so the list bypasses the deleted node
- return the slot to the free list: `Pointer[5] <- FreeListHead`, which is **4**
- `FreeListHead <- 5`, so the free list is now $5 \rightarrow 4 \rightarrow 6$ and the list is $2 \rightarrow 3 \rightarrow 1$
- the data in `Data[5]` is not erased; the node is deleted because nothing points at it any more

3.5

- a **linked list**
- patients are seen in arrival order, and a linked list keeps them in that order by following the pointers from the head
- a patient may leave before being seen, and deleting a node from the middle of a linked list changes only pointers, with no shifting of the other items, which an array would need; and the list can grow as far as memory allows, which matters because the number waiting is not known in advance
- a plain queue is accepted **only** if the answer deals with removal from the middle; a queue removes from the front only, so a patient leaving early cannot be taken out of it