Chapter X – Data Structures

By the end of this chapter you should:

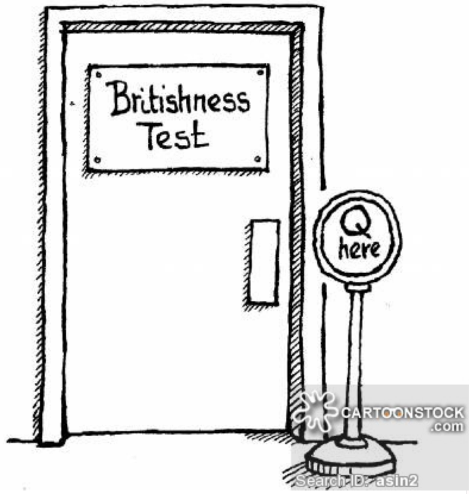
- Understand how to use an array in different ways to store data
- Be able to implement a queue and a stack

What is a data structure?

- A collection of data values
- …with some relationship between them
- …and certain functions that can be carried out on them
- It’s a storage format that allows you to access and change data in memory efficiently
- One or more pointers are used to show where to access/ manipulate/ store data
  - ….so that you don’t have to keep sorting it and moving it around

- Many data structures are represented by an abstract idea that we understand e.g. a 2D array as a table, a queue and a stack of data, a binary tree:
  - these are simple representations that show how something works and behaves that in reality aren’t quite what the computer system actually does!

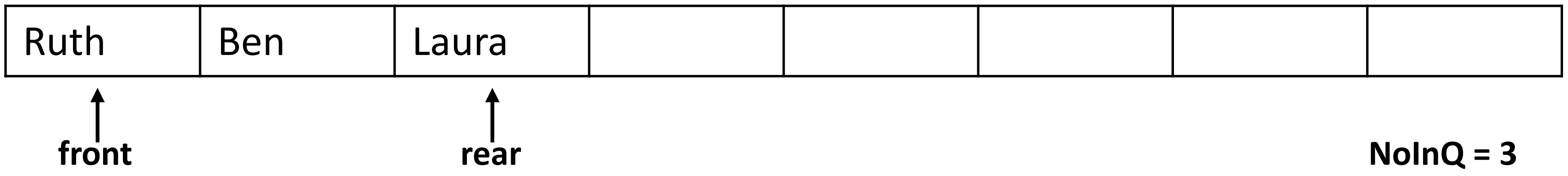
Queues

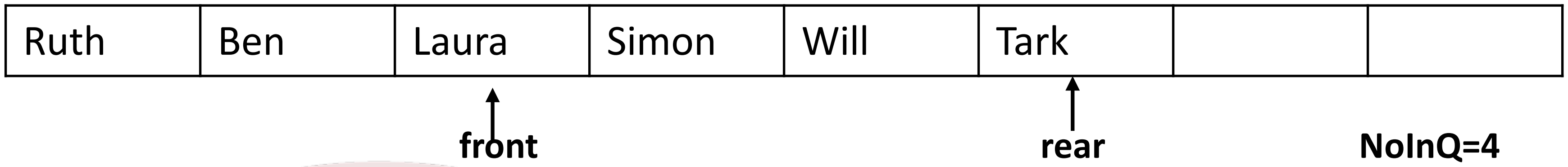


We all know the 'rules' of a queue:

- Operate as **First In First Out** (FIFO)
- Two possible operations:
  - Add an item to the end of the queue (enqueue)
  - Take an item from the front (dequeue)
- Data can only be retrieved in the order it was entered
  - i.e. you can only remove the item at the front of the queue
- The size of the queue depends upon the number of items of data – it's **dynamic** (not static like an array)
- The only difference between this and a 'real' queue is that when we remove an item of data, we don't move data towards the front (unlike people who would shuffle closer to the front when someone left the queue!) – we simply adjust pointers

Queue Representation





1) Ruth, Ben and Laura form a queue. Ruth is at the front, Laura at the rear.
2) Ruth is processed, then Ben is processed. The front pointer moves each time so that it points at the next item to be processed. Laura is now at the front of the queue.
3) Simon, Will and Tark have joined the queue and the back pointer has incremented along each time someone has been added.
4) Notice the number in queue variable being used–why might we need this?

Uses of Queues

- **Print Queues** - jobs needing to be printed in a networked environment tend to be done in a first come first printed order
- **Keyboard Buffer** - allows a whole line to be typed and stored until the processor can process it. It wouldn’t work well if the letters didn’t come out in the order you typed them!
- **Storing processes** and jobs to be run by the computer – operating systems use a wide range of scheduling methods to decide what order to carry out tasks in but most of these involve some sort of simple job queues within them

Using an array as a queue

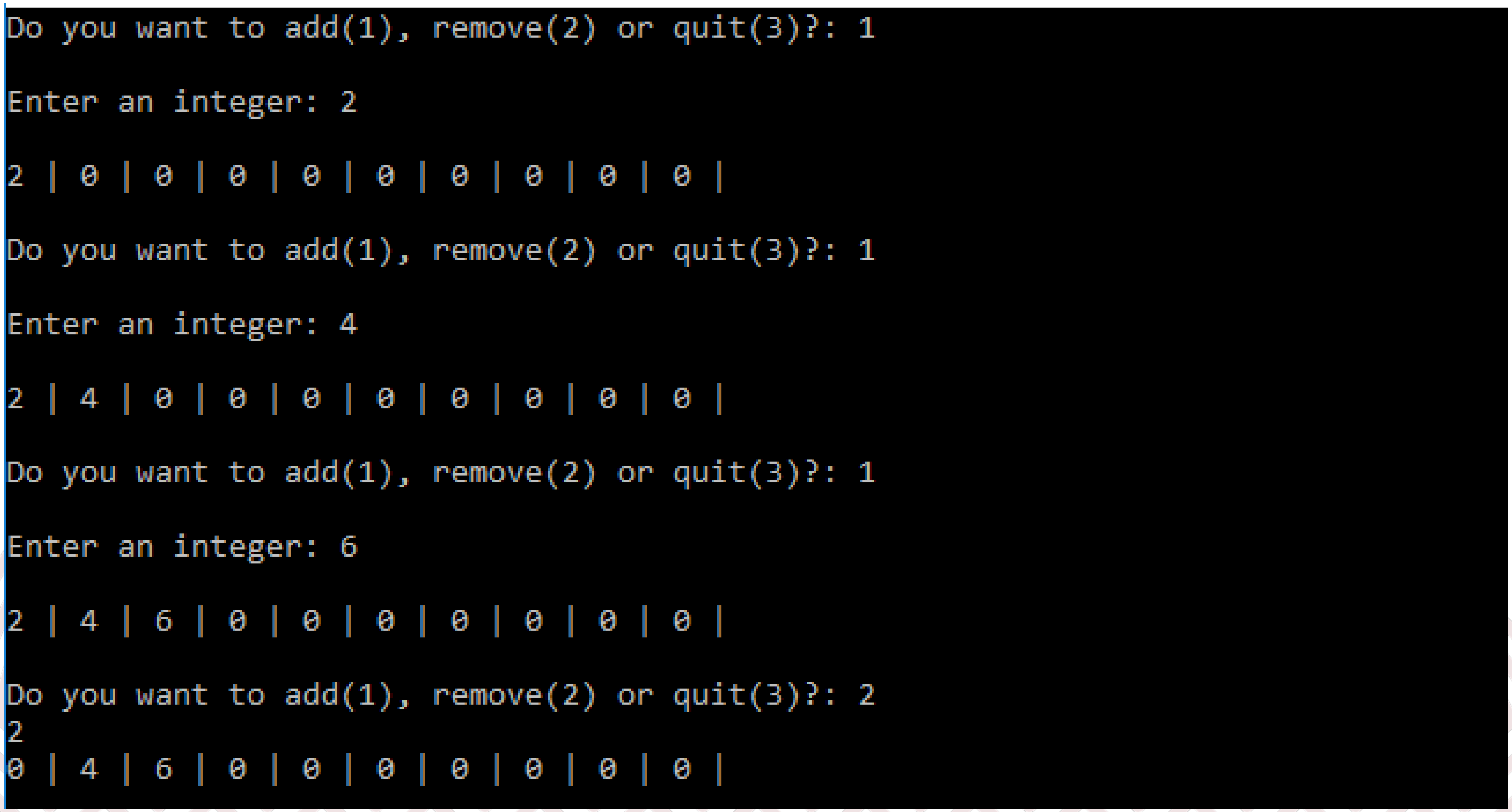
- Using the Q array and pointers shown (front, rear, NoInQ)
- If adding (enqueue):
  - Check for **overflow** (if the Q is full)
  - Get & then add data
  - Adjust relevant pointers
- If removing (dequeue):
  - Check for **underflow** (if the Q is empty)
  - Remove data
  - Adjust relevant pointers

- Copy & save your own version of the **QueueSkeleton** program from P:\A level Computing\C Programming Examples\ **QueueSkeleton.cpp**

- Using the algorithm outline given write the code for the **enqueue** and **dequeue** subroutines as outlined above

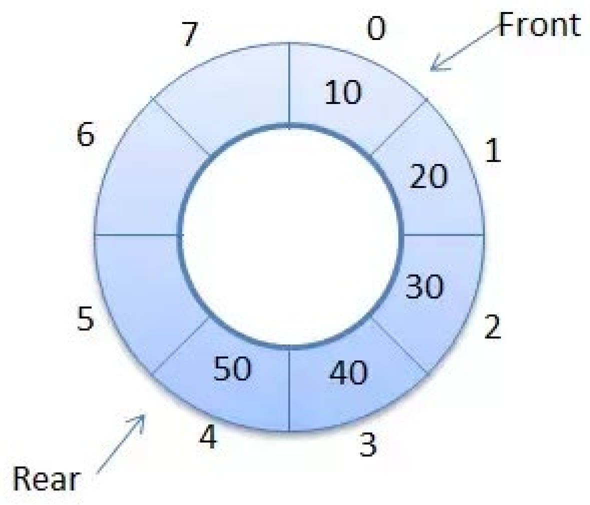
- Don't change anything else!!

Example Output



```
Do you want to add(1), remove(2) or quit(3)?: 1

Enter an integer: 2

2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Do you want to add(1), remove(2) or quit(3)?: 1

Enter an integer: 4

2 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Do you want to add(1), remove(2) or quit(3)?: 1

Enter an integer: 6

2 | 4 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Do you want to add(1), remove(2) or quit(3)?: 2
2
0 | 4 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
```

Circular Queues



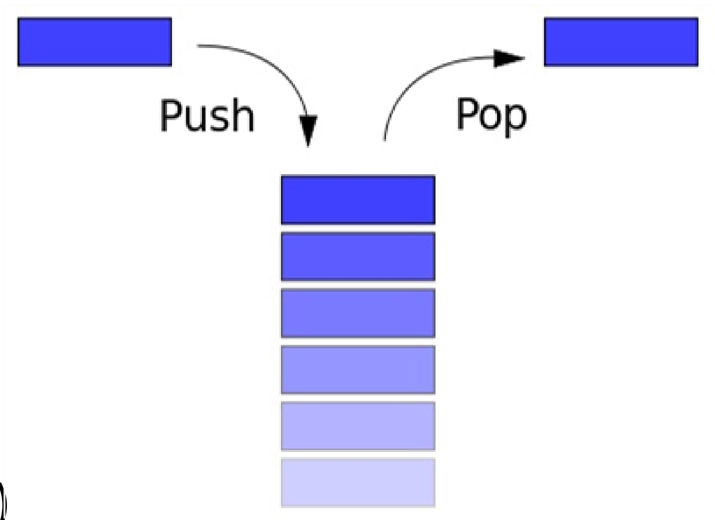
- This basic queue isn't a lot of use – once we have filled it and emptied it, even if there's nothing in the Q, my rear pointer is at the end and I can't add any more data

- SO we need to be able to make use of any empty space…..
- …instead of visualising our Q as a line of data, instead we need to think of it like the diagram shown

- When the **Rear** pointer hits the end of the Q, **so long as the queue isn't full** (how would we know this?) – it will start again at Q[0]
- ….The **front** pointer will also behave like this, moving back to 0 when it reaches the end

- Edit your Queue program so that it works this way instead and can reuse the empty space at the front of a Q once data has been removed – make sure you test it well

Stacks



- Operate as **Last In First Out (LiFO)**

- Two possible operations:
  - Add an item to the top of the stack (**push**)
  - Remove an item from the top of the stack (**pop**)

- Data is retrieved in the opposite order it was entered
  - i.e. you can only remove the item at the top of the stack

- A stack is also **dynamic** and depends upon the number of items of data in it

- All we need to work with our stack is a pointer to the **top**
- ….and a variable to store the **maxSize** of the stack

Uses of Stacks

- **Undo lists**- in most programs you can ‘undo’ a series of the last actions you took – you need this to happen in the reverse order you did them in
- **Web browsing ‘back’**- again, you usually want to go backwards to the last page you were on, and it remembers these in reverse order
- **Subroutines –** when we leave one subroutine and jump to another, all the local variables AND the address of the line of code you were running have to be stored so they can be reloaded once the function/procedure you jumped to have been run. We might jump around to a number of different subs and the data from each will be reloaded in opposite order
- **Recursion** – more about this soon. It’s similar to above except more painful!
- **Evaluating mathematical expressions** – stacks are used for reading something called ‘reverse polish’ – we’ll learn more about this when we look at how compilers work

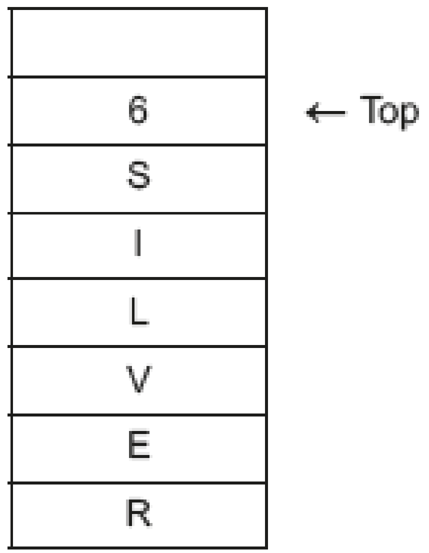
Using an array as a stack

Edit the Queue program you wrote

- Using the Stack array and pointers shown
- If adding (push):
  - Check for **overflow** (if the stack is full)
  - Get & then add data
  - Adjust top pointer
- If removing (pop):
  - Check for **underflow** (if the stack is empty)
  - Remove data
  - Adjust top pointer

Stack Exercises

A stack, in shared memory, is being used to pass a single variable length ASCII string between two sub-systems. The string is placed in the stack one character at a time in reverse order with the last byte holding the number of characters pushed i.e the text "SILVER" would be held in the stack as:



Write a program to demonstrate how this would work, using a stack approach.

The code should take a string as input, and push it onto the stack as described above, then output the stack so the result can be seen on screen.