10 Data Types and Structures

10.1 Data Types and Records

Candidates should be able to:	Notes and guidance
Select and use appropriate data types for a problem solution	including integer, real, char, string, Boolean, date (pseudocode will use the following data types: INTEGER, REAL, CHAR, STRING, BOOLEAN, DATE, ARRAY, FILE)
Show understanding of the purpose of a record structure to hold a set of data of different data types under one identifier	Write pseudocode to define a record structure Write pseudocode to read data from a record structure and save data to a record structure

10.2 Arrays

Candidates should be able to:	Notes and guidance
Use the technical terms associated with arrays	Including index, upper and lower bound
Select a suitable data structure (1D or 2D array) to use for a given task	
Write pseudocode for 1D and 2D arrays	
Write pseudocode to process array data	Sort using a bubble sort Search using a linear search

10.3 Files

Candidates should be able to:	Notes and guidance
Show understanding of why files are needed	
Write pseudocode to handle text files that consist of one or more lines	

10.4 Introduction to Abstract Data Types (ADT)

Candidates should be able to:	Notes and guidance
Show understanding that an ADT is a collection of data and a set of operations on those data	
Show understanding that a stack, queue and linked list are examples of ADTs	Describe the key features of a stack, queue and linked list and justify their use for a given situation
Use a stack, queue and linked list to store data	Candidates will not be required to write pseudocode for these structures, but they should be able to add, edit and delete data from these structures
Describe how a queue, stack and linked list can be implemented using arrays	

Data types

Primitive data types

What are data types?

- A data type is a classification of data into groups according to the **kind of data they represent**
- Computers use different data types to represent different types of data in a program
- The basic data types include:
  - **Integer**: used to represent **whole numbers**, either positive or negative
    - Examples: 10, -5, 0
  - **Real**: used to represent **numbers with a fractional part**, either positive or negative
    - Examples: 3.14, -2.5, 0.0
  - **Char**: used to represent a **single character** such as a letter, digit or symbol
    - Examples: 'a', 'B', '5', '$'
  - **String**: used to represent a **sequence of alphanumerical characters**
    - Examples: "Hello World", "1234", "@#$%"
  - **Boolean**: used to represent **true or false** values
    - Examples: True, False
  - **Date**: used to store a **calendar date**
    - Example: DD/MM/YY
- It is important to choose the **correct data type for a given situation** to ensure accuracy and efficiency in the program
- In pseudocode and some other programming languages, **data types must be declared** before they can be used
  - The data types are declared with each **data item** to be used
  - Each data item is given a **unique** name called an **identifier**
  - DECLARE <identifier> : <data type>

Data Type	Example Value	Pseudocode	Python	Java	VB.NET
Integer	10	DECLARE Age : INTEGER Age ← 10	age = 10	int age = 10;	Dim age As Integer = 10
Real	3.14	DECLARE Temp : REAL Temp ← 3.14	temp = 3.14	double temp = 3.14;	Dim temp As Double = 3.14
Char	'A'	DECLARE Grade : CHAR Grade ← 'A'	grade = 'A'	char grade = 'A';	Dim grade As Char = "A"c
String	"Hello"	DECLARE Name : STRING Name ← "Hello"	name = "Hello"	String name = "Hello";	Dim name As String = "Hello"
Boolean	TRUE	DECLARE LoggedIn : BOOLEAN LoggedIn ← TRUE	logged_in = True	boolean loggedIn = true;	Dim loggedIn As Boolean = True
Date	24/04/2025	DECLARE DOB : DATE DOB ← "24/04/2025"	dob = "24/04/2025" (string or datetime)	LocalDate dob = LocalDate.of(2025, 4, 24);	Dim dob As Date = #24/04/2025#

Record structures

Purpose of records

What is a record?

- A record is a **composite data structure** used to store a **collection of related data items**
- Each item **can be** of a **different data type**
- It allows a programmer to group related values under **one identifier**
- Records help create a **more structured and readable** approach to storing and managing data
- A record contains a **fixed number of fields**, each with its own name and type
- In pseudocode, a record can by declared as:

```
TYPE <Typename>
   DECLARE <identifier> : <data type>
   DECLARE <identifier> : <data type>
   DECLARE <identifier> : <data type>
   ..
   ..
ENDTYPE
```

- Suppose we want to store information about a car
- We can define a **record type** like this:

```
TYPE Car
   DECLARE Make : STRING
   DECLARE Model : STRING
   DECLARE Colour : STRING
   DECLARE Price : REAL
   DECLARE DateOfRegistration : DATE
ENDTYPE
```

- We can then **create a variable** of this type:

```
DECLARE MyCar : Car
```

- And **assign values** to its fields:

```
MyCar.Make ← "Toyota"
MyCar.Model ← "Yaris"
MyCar.Colour ← "Red"
MyCar.Price ← 14995.99
MyCar.DateOfRegistration ← "12/03/2022"
```

- To **read from the record** we can:

```
OUTPUT "Make: ", MyCar.Make
OUTPUT "Model: ", MyCar.Model
OUTPUT "Colour: ", MyCar.Colour
OUTPUT "Price: £", MyCar.Price
OUTPUT "Date of Registration: ", MyCar.DateOfRegistration
```

Examiner Tips and Tricks

- **Records** in programming are a type of **data structure** used to group related data in your code
- These are **not the same** as records in a database, which refer to a **complete set of fields** on a single entity in a table (**row**)

Key features of records

Feature	Explanation
Can store **different data types**	Unlike arrays, records are not limited to a single type
Fields are **named**	Each item in the record has a meaningful identifier
Offers **structured storage**	Easier to manage complex data about real-world entities

Array basics

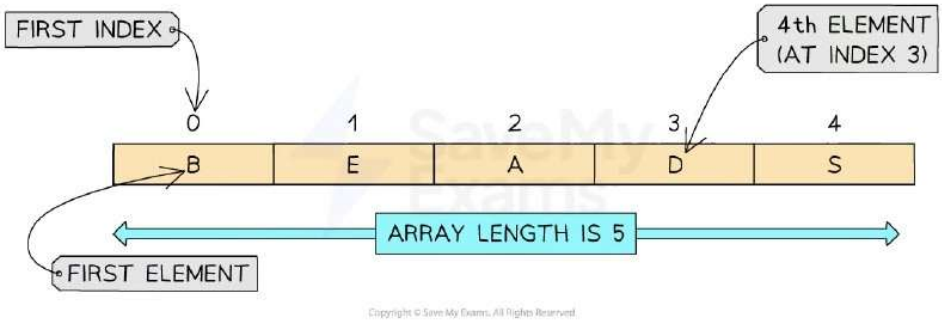
Arrays

What is an array?

- An array is an **ordered, static set of elements**
- Can only store **1 data type**
- The **position of each element** in an array is identified using the array's **index**
- The array's **first element** is the **lower bound (LB)**
- The array's **last element** is the **upper bound (UB)**
- The **lower bound of an array** is typically **0 or 1** depending on the language being used
- An array can be **one-dimensional** or **multi-dimensional**

One-dimensional (1D) arrays

- A 1D array is a **linear array**



- To declare a 1D array in pseudocode you **must include** the lower bound, upper bound and data types:

```
DECLARE <identifier> : ARRAY[LB:UB] OF <data type>
```

- In this example a 1D array of **five elements** each containing **single character** can be declared as:

```
DECLARE Letters : ARRAY[0:4] OF CHAR
```

- An example **complete program** could be:

```
// Declare the array
DECLARE Letters : ARRAY[0:4] OF CHAR

// Assign values to each index
Letters[0] ← 'B'
Letters[1] ← 'E'
```

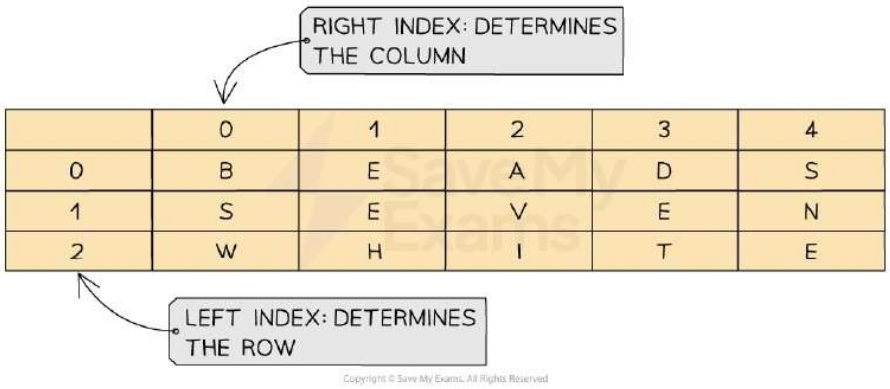
```
Letters[2] ← 'A'
Letters[3] ← 'D'
Letters[4] ← 'S'

// Output the full array
FOR Index ← 0 TO 4
  OUTPUT Letters[Index]
NEXT Index
```

- The array is declared with **indices** from 0 to 4
- Each element stores a single character using the CHAR **data type**
- The **loop** outputs each letter in order

Two-dimensional (2D) arrays

- A 2D array can be **visualised as a table**
- When navigating through a 2D array you first have to go **down the rows** and then **across the columns** to find a position within the array



	0	1	2	3	4
0	B	E	A	D	S
1	S	E	V	E	N
2	W	H	I	T	E

- In 2D arrays the following must be declared:
  - Lower bound for rows (**LBR**) & upper bound for rows (**UBR**)
  - Lower bound for columns (**LBC**) & upper bound for columns (**UBC**)

```
DECLARE <identifier> : ARRAY[LBR:UBR, LBC:UBC] OF <data type>
```

- In this example a **2D array** can be declared as:

```
DECLARE Letters : ARRAY[0:2, 0:4] OF CHAR
```

- An example **complete program** could be:

```
DECLARE Letters : ARRAY[0:2, 0:4] OF CHAR

// Row 0
Grid[0,0] ← 'B'
Grid[0,1] ← 'E'
Grid[0,2] ← 'A'
Grid[0,3] ← 'D'
```

```
Grid[0,4] ← 'S'

// Row 1
Grid[1,0] ← 'S'
Grid[1,1] ← 'E'
Grid[1,2] ← 'V'
Grid[1,3] ← 'E'
Grid[1,4] ← 'N'

// Row 2
Grid[2,0] ← 'W'
Grid[2,1] ← 'H'
Grid[2,2] ← 'I'
Grid[2,3] ← 'T'
Grid[2,4] ← 'E'
```

- ARRAY[0:2, 0:4] creates **3 rows** and **5 columns**
- First index is the **row**, second is the **column**: Letters[row, column]
- All elements are of type CHAR

Worked Example

A program reads data from a file and searches for specific data.

The main program needs to read 25 integer data items from the text file Data.txt into a local 1D array, DataArray

Write program code to declare the local array DataArray **[1]**

Answer

- 1D array with name DataArray (with 25 elements of type Integer) [1 mark]

Java

```
public static Integer[] DataArray = new Integer[25];
```

VB.NET

```
Dim DataArray(24) As Integer
```

Python

```
DataArray = [] #25 elements Integer
```

Array pseudocode

Searching arrays

What is a linear search?

- A linear search **starts with the first value** in a dataset and **checks every value one at a time** until all values have been checked
- A linear search can be performed **even if the values are not in order**

How do you perform a linear search?

Step	Instruction
1	Check the first value
2	**IF** it is the value you are looking for ▪ STOP!
3	**ELSE** move to the next value and check
4	**REPEAT UNTIL** you have checked all values and not found the value you are looking for

- A linear search can be used to **find elements in an array**
- Each element in the array is **checked in order**, from the **lower bound** to the **upper bound**, until the item is found or the upper bound is reached
- An example pseudocode algorithm to perform a linear search on a **1D array** could be:

```
DECLARE List : ARRAY[0:4] OF INTEGER
DECLARE Target : INTEGER
DECLARE Found : BOOLEAN
DECLARE Index : INTEGER

// Initialise array values
List[0] ← 12
List[1] ← 25
List[2] ← 37
List[3] ← 42
List[4] ← 56

// Input the value to search for
OUTPUT "Enter the value to search for:"
INPUT Target

Found ← FALSE
Index ← 0
```

```
WHILE Index <= 4 AND Found = FALSE DO
  IF List[Index] = Target THEN
    Found ← TRUE
  ELSE
    Index ← Index + 1
  ENDIF
ENDWHILE

IF Found = TRUE THEN
  OUTPUT "Item found at index ", Index
ELSE
  OUTPUT "Item not found"
ENDIF
```

- Works on a fixed-size array `List[0:4]`
- Uses a `WHILE` loop for flexibility (early exit if found)
- Clearly separates **input**, **processing**, and **output**

Identifier table

Identifier	Data type	Description
List	ARRAY[0:4] OF INTEGER	Stores the list of numbers to search through
Target	INTEGER	The number the user wants to search for
Found	BOOLEAN	Tracks whether the target value has been found
Index	INTEGER	The current position being checked in the array

Sorting arrays

What is a bubble sort?

- A bubble sort is a simple sorting algorithm that starts at the beginning of a dataset and **checks values** in **'pairs'** and **swaps them** if they are **not in the correct order**
- One full run of comparisons from beginning to end is called a '**pass**', a bubble sort may require **multiple** '**passes**' to sort the dataset
- The algorithm is finished when there are **no more swaps to make**

How do you perform a bubble sort?

Step	Instruction
1	Compare the first two values in the dataset

2	**IF** they are in the wrong order... ▪ Swap them
3	Compare the next two values
4	**REPEAT** step 2 & 3 until you reach the end of the dataset (**pass 1**)
5	**IF** you have made any swaps... ▪ **REPEAT** from the start (pass 2,3,4...)
6	**ELSE** you have not made any swaps... ▪ STOP! the list is in the correct order

Example

- Perform a bubble sort on the following dataset

5	2	4	1	6	3

Step	Instruction
1	Compare the first two values in the dataset 5 | 2 | 4 | 1 | 6 | 3
2	**IF** they are in the wrong order... ▪ Swap them 2 | 5 | 4 | 1 | 6 | 3
3	Compare the next two values 2 | 5 | 4 | 1 | 6 | 3
4	**REPEAT** step 2 & 3 until you reach the end of the dataset ▪ 5 & 4 SWAP! 2 | 4 | 5 | 1 | 6 | 3

	▪ 5 & 1 SWAP! 2 | 4 | 1 | 5 | 6 | 3 ▪ 5 & 6 NO SWAP! 2 | 4 | 1 | 5 | 6 | 3 ▪ 6 & 3 SWAP! 2 | 4 | 1 | 5 | 3 | 6 ▪ End of pass 1
5	**IF** you have made any swaps... ▪ **REPEAT** from the start ▪ End of pass 2 (swaps made) 2 | 1 | 4 | 3 | 5 | 6 ▪ End of pass 3 (swaps made) 1 | 2 | 3 | 4 | 5 | 6 ▪ End of pass 4 (no swaps) 1 | 2 | 3 | 4 | 5 | 6
6	**ELSE** you have not made any swaps... ▪ STOP! the list is in the correct order

- A bubble sort can be used to **put elements in an array in order**
- A **pseudocode algorithm** for a bubble sort on a 1D array could be:

```
DECLARE Numbers : ARRAY[0:5] OF INTEGER
DECLARE Temp : INTEGER
DECLARE Pass : INTEGER
DECLARE Index : INTEGER
DECLARE Swapped : BOOLEAN

// Initialise the array
Numbers[0] ← 5
Numbers[1] ← 2
Numbers[2] ← 4
```

```
Numbers[3] ← 1
Numbers[4] ← 6
Numbers[5] ← 3

// Bubble sort algorithm
FOR Pass ← 1 TO 5
  Swapped ← FALSE
  FOR Index ← 0 TO 4
    IF Numbers[Index] > Numbers[Index + 1] THEN
      Temp ← Numbers[Index]
      Numbers[Index] ← Numbers[Index + 1]
      Numbers[Index + 1] ← Temp
      Swapped ← TRUE
    ENDIF
  NEXT Index
  IF Swapped = FALSE THEN
    // List is already sorted
    EXIT
  ENDIF
NEXT Pass

// Output the sorted array
FOR Index ← 0 TO 5
  OUTPUT Numbers[Index]
NEXT Index
```

- Uses a fixed array of size 6
- Includes an **optimisation** to exit early if no swaps were made on a pass

Identifier table

Identifier	Data type	Description
Numbers	ARRAY[0:5] OF INTEGER	Stores the list of numbers to be sorted
Temp	INTEGER	Temporary variable used for swapping two elements in the array
Pass	INTEGER	Tracks the number of passes through the array
Index	INTEGER	Tracks the current position being compared in the array
Swapped	BOOLEAN	Indicates whether a swap occurred during a pass (used to optimise sorting)

File handling

Purpose of files

What is file handling?

- File handling is **the use of programming techniques** to work with information stored in **text files**
- Examples of file handing techniques are:
  - **opening** text files
  - **reading** text files
  - **writing** text files
  - **closing** text files

Concept	CIE A Level Pseudocode
Open file for reading	OPENFILE fruit.txt FOR READ
Open file for writing	OPENFILE Shopping.txt FOR WRITE
Close file	CLOSEFILE fruit.txt
Read line	READFILE fruit.txt, line
Write line	WRITE fruit.txt, "Oranges"
Check end of file	IF NOT EOF(fruit.txt) THEN
Create new file	OPENFILE Shopping.txt FOR WRITE (same as writing – it overwrites or creates)
Append to file	OPENFILE Shopping.txt FOR APPEND

- An example program written in pseudocode to:
  - **Opens** a text file for reading
  - **Reads** each line (a fruit name)
  - **Counts** how many fruits are in the file
  - **Outputs** the total

- **Closes** the file

```
DECLARE FruitName : STRING
DECLARE FruitCount : INTEGER

FruitCount ← 0

OPENFILE FruitFile FOR READ

WHILE NOT EOF(FruitFile) DO
   READFILE FruitFile, FruitName
   FruitCount ← FruitCount + 1
ENDWHILE

CLOSEFILE FruitFile

OUTPUT "Total number of fruits: ", FruitCount
```

Identifier table

Identifier	Data type	Description
FruitName	STRING	Stores the name of each fruit read from the file
FruitCount	INTEGER	Keeps track of how many fruits have been read
FruitFile	FILE	The file containing the fruit names (e.g. "fruit.txt")

Worked Example

Write program code to read the contents of Data.txt into DataArray

Answer

- Opening file Data.txt to read [1 mark]
- Looping through all the 25/EOF ... [1 mark]
- ... reading each line and storing/appending into array [1 mark]
- Exception handling with appropriate output [1 mark]
- Closing the file (in an appropriate place) [1 mark]

Example program code:

Java

```
Integer Counter = 0;
try{
   Scanner Scanner1 = new Scanner(new File("Data.txt"));
   while(Scanner1.hasNextLine()){
      DataArray[Counter] = Integer.parseInt(Scanner1.next());
      Counter++;
   }
   Scanner1.close();
}catch(FileNotFoundException ex){
```

```
    System.out.println("No data file found");
}
```

VB.NET

```
try
  Dim DataReader As New System.IO.StreamReader("Data.txt")
  Dim X As Integer = 0
  Do Until DataReader.EndOfStream
    DataArray(X) = DataReader.ReadLine()
    X = X + 1
  Loop
  DataReader.Close()
Catch ex As Exception
  Console.WriteLine("Invalid file")
End Try
```

Python

```
try:
  DataFile = open("Data.txt",'r')
  for Line in DataFile:
    DataArray.append(int(Line))
  DataFile.close()
except IOError:
  print("Could not find file")
```

ADT overview

What is an abstract data type (ADT)?

- An abstract data type (ADT) is a **collection of data** and **a set of operations** on that data
- Three common ADT's are:
  - **Stacks**
  - **Queues**
  - **Linked lists**

Stack operations

What is a stack?

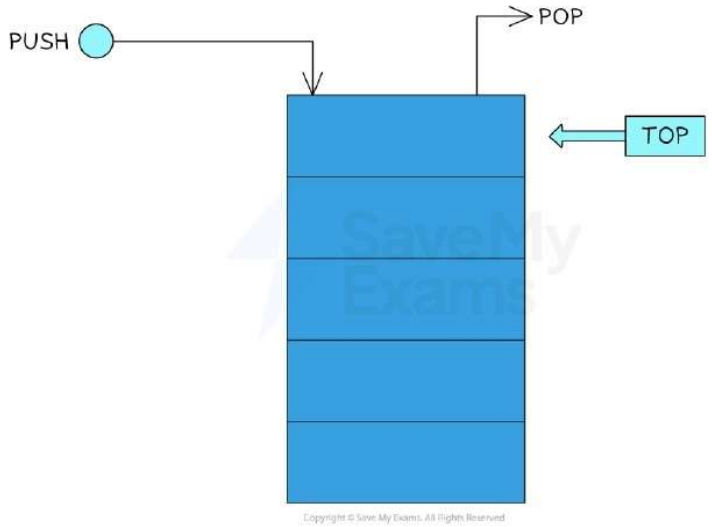
- A stack is an abstract data type that stores data using the **Last In, First Out (LIFO)** principle
- A stack is like **a pile of plates**, the last item you put on is the first one you take off
  - **PUSH**: Add an item to the top of the stack
  - **POP**: Remove the item from the top
- The **first item pushed** onto the stack will be the **last one popped** off
- A stack uses two pointers:
  - **A base pointer** - points to the first item in the stack
  - **A top pointer** - points to the last item in the stack

Main operations

Operation	Description
isEmpty()	This checks is the stack is empty by checking the value of the top pointer
push(value)	This adds a new value to the end of the list, it will need to check the stack is not full before pushing to the stack.
peek()	This returns the top value of the stack. First check the stack is not empty by looking at the value of the top pointer.
pop()	This removes and returns the top value of the stack. This first checks if the stack is not empty by looking at the value of the top pointer.

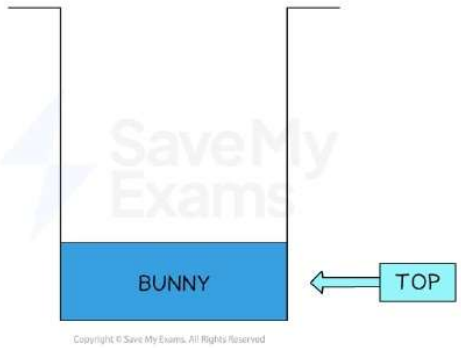
size()	Returns the size of the stack
isFull()	Checks if the stack is full and returns the Boolean value, this compares the stack size to the top pointer.

- Stacks use a **pointer** which points to the top of the stack, where the next piece of data will be added (pushed) or the current piece of data can be removed (popped)

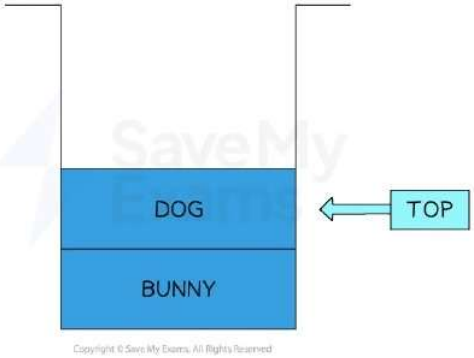


Pushing data to a stack

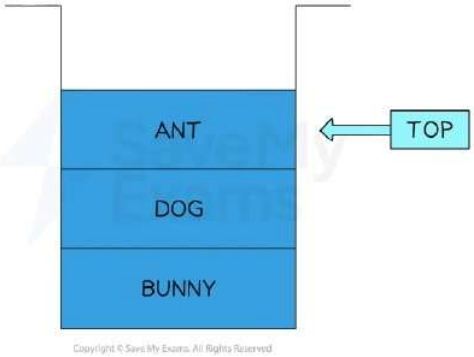
- When pushing (**adding**) data to a stack, the data is pushed to the position of the pointer
- Once pushed, the **pointer will increment by 1**, signifying the top of the stack
- Since the stack is a **static data structure**, an attempt to push an item on to a full stack is called a **stack overflow**
- This would push 'Bunny' onto the empty stack



- This would push 'Dog' to the top of the stack

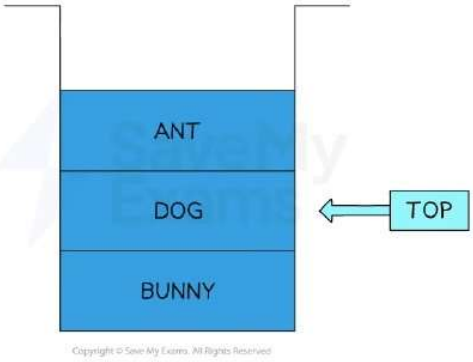


- This would push 'Ant' to the top of the stack



Popping data from a stack

- When popping (**removing**) data from a stack, the data is popped from the position of the pointer
- Once popped, the **pointer will decrement by 1**, to point at the new top of the stack
- Since the stack is a static data structure, an attempt to pop an item from an empty stack is called a **stack underflow**
- This would pop 'Ant' from the top of the stack
- The new top of the stack would become 'Dog'



- Note that the data that is 'popped' isn't necessarily erased
- The pointer 'top' moves to show that 'Ant' is no longer in the stack and depending on the implementation, 'Ant' can be deleted or replaced with a null value, or left to be overwritten

Queue operations

What is a queue?

- A queue is an abstract data type that **stores data in the order it arrives**, using the **First In, First Out (FIFO)** principle
- A queue works like **a line of people waiting at a shop**, the first one in is the first one served
  - **ENQUEUE**: Add an item to the back of the queue
  - **DEQUEUE**: Remove the item from the front
- The **first item added** to the queue is the **first one removed**

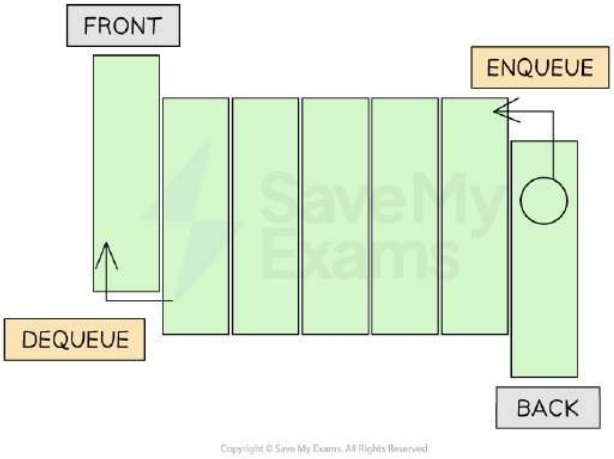
Main operations

Operation	Description
enQueue(value)	Adding an element to the back of the queue
deQueue()	Returning an element from the front of the queue
peek()	Return the value of the item from the front of the queue without removing it from the queue
isEmpty()	Check whether the queue is empty

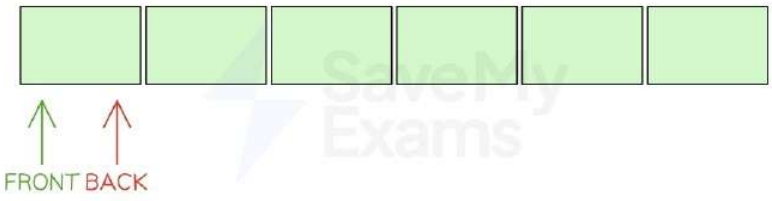
isFull()	Check whether the queue is full (if the size has a constraint)

Linear queues

- A linear queue is a data structure that **consists of an array**
- Items are added to the **next available space** in the queue starting from the front
- Items are then **removed from the front** of the queue



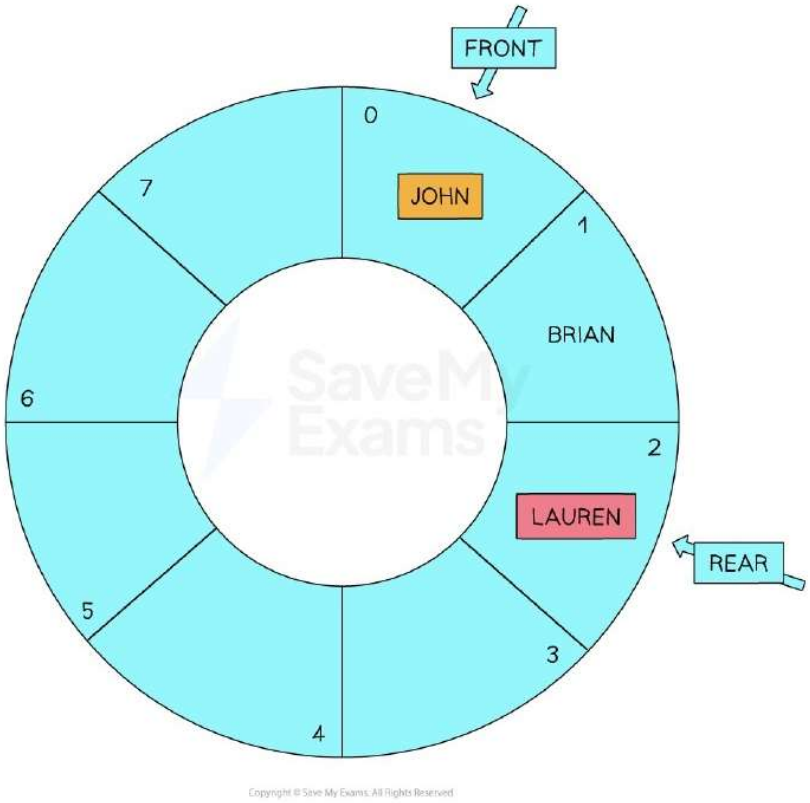
- Before **adding an item to the queue** you need to **check that the queue is not full**
- If the end of the array has not been reached, the rear **index pointer is incremented** and the new item is added to the queue
- In the example below, you can see the queue as the data Adam, Brian and Linda are enqueued.



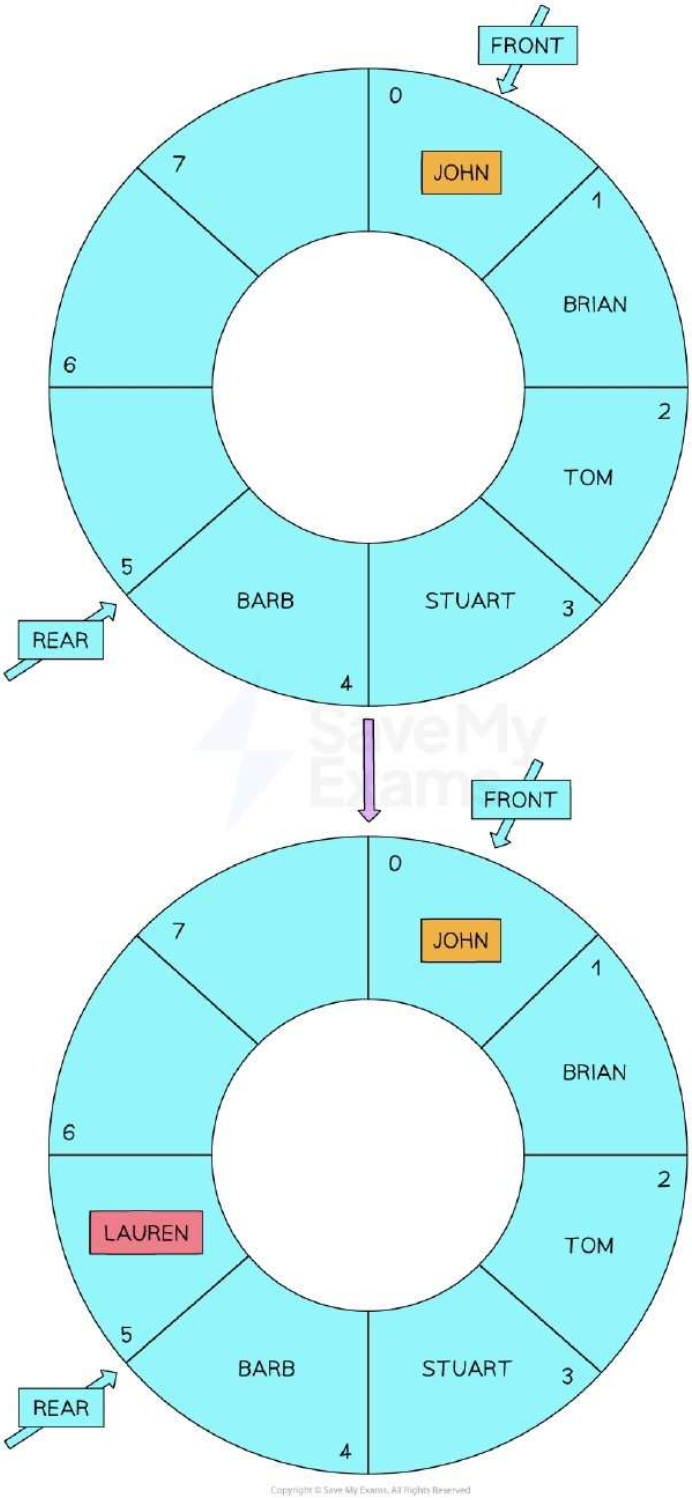
- Before **removing** an item from the queue, you need to make sure that the queue is not empty
- If the queue is not empty the item at the front of the queue is returned and **the front is incremented by 1**
- In the example below, you can see the queue as the data Adam is dequeued

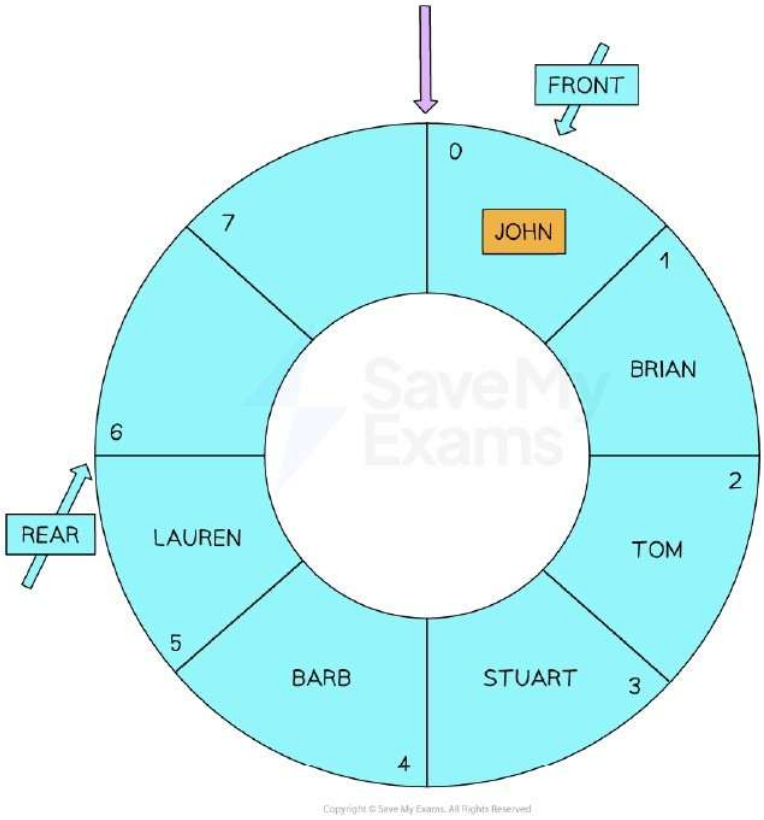
Circular queues

- A circular queue is **a static array** that has a **fixed capacity**
- It means that as you add items to the queue you will **eventually reach the end** of the array
- When items are dequeued, **space is freed up at the start of the array**
- It would **take time** to move items up to the start of the array to free up space at the end, so a Circular queue (or circular buffer) is implemented
- It **reuses empty slots** at the front of the array that are caused when items are dequeued
- As items are enqueued and the rear index pointer reaches the last position of the array, it wraps around to point to the start of the array as long as it isn't full
- When items are dequeued the front index pointer will wrap around until it passes the rear index points which would show the queue is empty

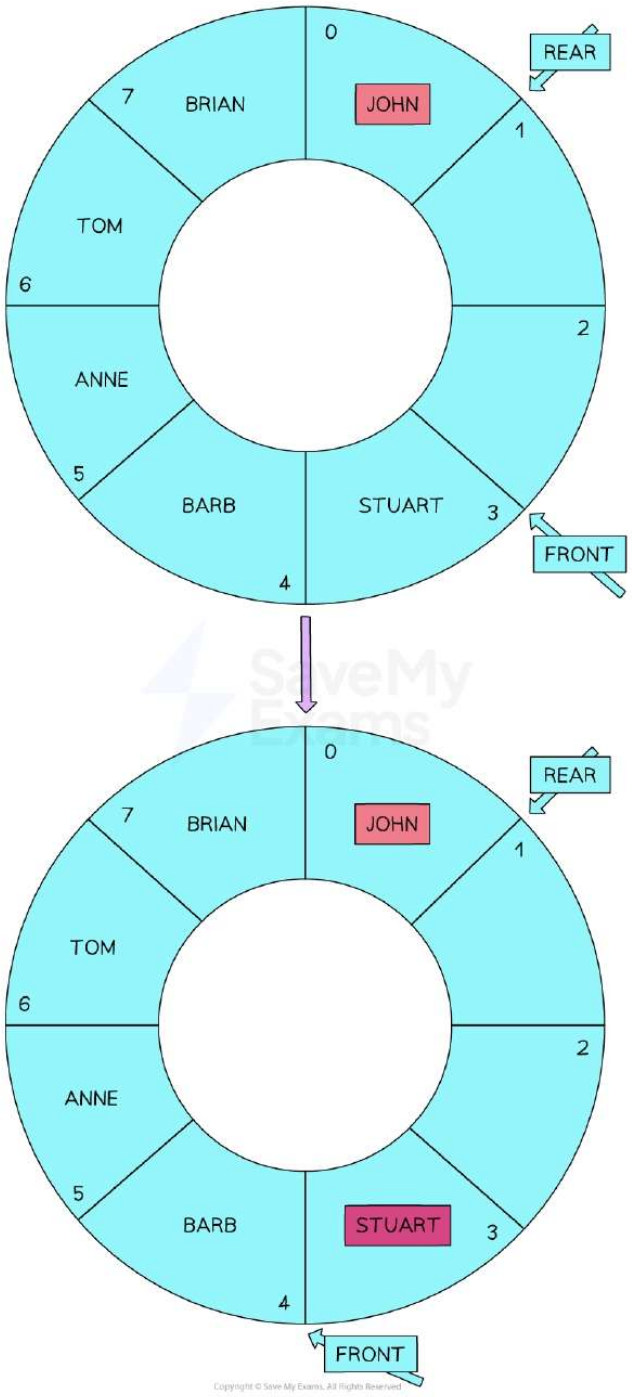


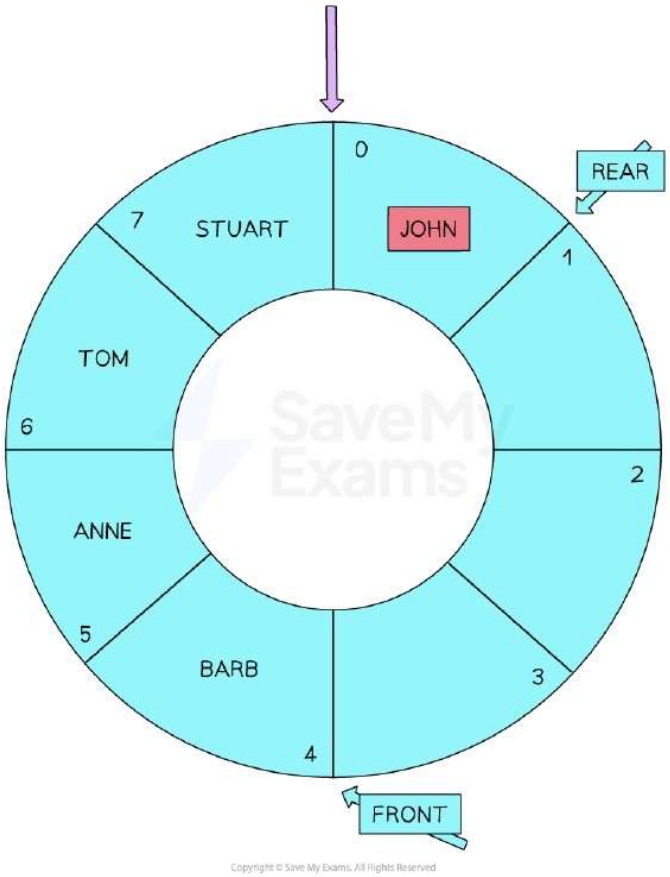
- Before **adding** an item to the circular queue you need to check that the **array is not full**
- It will be full if the next position to be used **is already occupied** by the item at the front of the queue
- If the queue is not full then the **rear index pointer must be adjusted** to reference the next free position so that the new item can be added
- In the example below, you can see the circular queue as the data Lauren is enqueued.





- Before **removing** an item from the queue, you need to make sure that the queue is not empty
- If the queue is not empty the **item at the front of the queue is returned**
- If this is the only item that was in the queue the **rear and front pointers are reset**, otherwise the pointer moves to reference the next item in the queue
- In the example below, you can see the queue as the data Adam is dequeued


0
REAR
JOHN
1
2
3
FRONT
4
BARB
5
ANNE
6
TOM
7
BRIAN
0
REAR
JOHN
1
2
STUART
3
FRONT
4
BARB
5
ANNE
6
TOM
7
BRIAN



Linked list operations

What is a linked list?

- A linked list is an abstract data type where each item, or **node**, contains a **data field** and a **pointer to the next node** in the sequence
- A linked list is **a chain of items**, where each item stores **two things**:
  - The **actual data**
  - **A pointer** to the next item in the list
- New items are usually added to the **start** of the list, and each one **links** to the next
- Like a **scavenger hunt** where each clue tells you where to go next
- Each item is called a **node** and contains a data field alongside another address which is called a pointer
- For example:

Index	Data	Pointer
0	'Apple'	2
1	'Pineapple'	0
2	'Melon'	-
3		

Start = 1 NextFree = 3

- The data field contains the **value of the actual data** which is part of the list
- The pointer field contains the **address of the next item** in the list

Examiner Tips and Tricks

- Linked lists can only be traversed by following each item's next pointer until the end of the list is located

Traverse a linked list

- Check if the linked list is empty
- Start at the node the pointer is pointing to (Start = 1)
- Output the item at the current node ('Pineapple')
- Follow the pointer to the next node repeating through each node until the end of the linked list.
- When the pointer field is empty/null it signals that the end of the linked list has been reached
- Traversing the example above would produce: 'Pineapple', 'Apple', 'Melon'

Adding a node

- An advantage of using linked lists is that values can easily be added or removed by editing the points
- The following example will add the word 'Orange' after the word 'Pineapple'

1. Check there is free memory to insert data into the node

2. Add the new value to the end of the linked list and update the 'NextFree' pointer

3	'Orange'	

Start = 1 NextFree = 4

3. The pointer field of the word 'Pineapple' is updated to point to 'Orange', at position 3

1	'Pineapple'	3

4. The pointer field of the word 'Orange' is updated to point to 'Apple' at position 0

3	'Orange'	0

4. When traversed this linked list will now output: 'Pineapple', 'Orange', 'Apple', 'Melon'

Removing a node

- Removing a node also involves updating nodes, this time to bypass the deleted node
- The following example will remove the word 'Apple' from the original linked list

1. Update the pointer field of 'Pineapple' to point to 'Melon' at index 2

Index	Data	Pointer
0	'Apple'	2
1	'Pineapple'	2
2	'Melon'	-

2. When traversed this linked list will now output: 'Pineapple', 'Melon'

- The node is not truly removed from the list; it is only ignored
- Although this is easier it does waste memory
- Storing pointers themselves also means that more memory is required compared to an array
- Items in a linked list are also stored in a sequence, they can only be traversed within that order so an item cannot be directly accessed as is possible in an array