OCR A Level Computer Science

Your notes

4.2 Data Structures

Contents

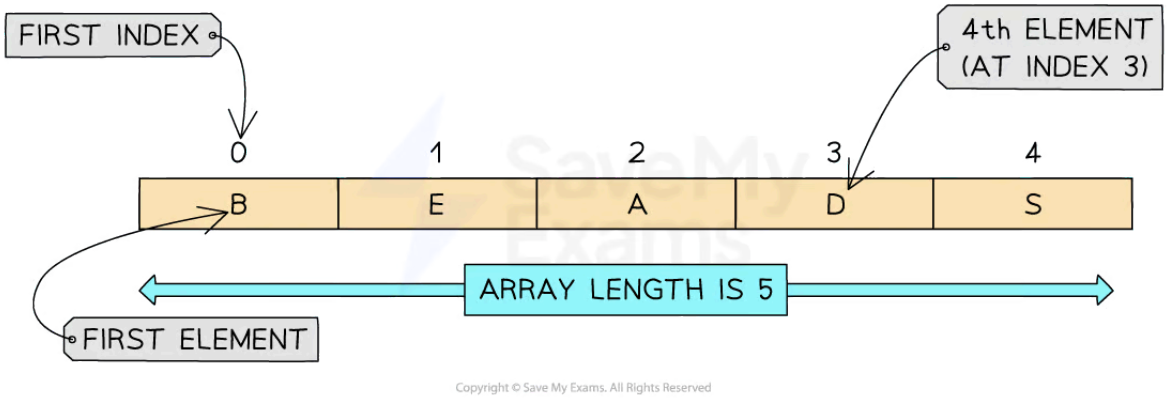
Arrays

Your notes

1D Arrays

What is an Array?

- An array is an ordered, static set of elements
- Can only store 1 data type
- A 1D array is a linear array



Structure of a 1D array

Example in Pseudocode

In this example we will be creating a one-dimensional array called 'array' which contains 5 integers.

- To create the array we can use the following syntax:

```
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
```

- We can access the individual elements of the array by using the following syntax:

```
array[index]
```

- We can also modify the individual elements by assigning new values to specific indices using the following syntax:

```
array[index] = newValue
```

- We can also use the len function to determine the length of the array by using the following syntax:
  len(array)
- In the example we have iterated through the array to output each element within the array. We have used a For Loop for this.

```
// Creating a one-dimensional array
array array[5]
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5


// Accessing elements of the array
print(array[0])
print(array[2])


// Modifying elements of the array
array[1] = 10
print(array)


// Iterating over the array
for element in array
 print(element)


// Length of array
length = len(array)
print(length)
```

Example in Python

Creating a one-dimensional array called 'array' which contains 5 integers.

- Create the array with the following syntax:
  array = [1, 2, 3, 4, 5]
- Access the individual elements of the array by using the following syntax:
  array[index]
- Modify the individual elements by assigning new values to specific indices using the following syntax:
  array[index] = newValue
- Use the len function to determine the length of the array by using the following syntax:
  len(array)


Your notes

- In the example the array has been iterated through to output each element within the array. A for loop has been used for this


Your notes

```
# Creating a one-dimensional array
array = [1, 2, 3, 4, 5]

# Accessing elements of the array
print(array[0])   # Output: 1
print(array[2])   # Output: 3

# Modifying elements of the array
array[1] = 10
print(array)      # Output: [1, 10, 3, 4, 5]

# Iterating over the array
for element in array:
    print(element)

# Output:
# 1
# 10
# 3
# 4
# 5

# Length of the array
length = len(array)
print(length)     # Output: 5
```

Example in Java

Creating a one-dimensional array called 'array' which contains 5 integers.

- To create the array, use the following syntax:
  `int[] array = {1, 2, 3, 4, 5};`
- Access the individual elements of the array by using the following syntax:
  `array[index]`
- Modify the individual elements by assigning new values to specific indices using the following syntax:
  `array[index] = newValue;`
- Use the following syntax to print the array as a string:
  `arrays.toString(array)`
- Use the length function to determine the length of the array by using the following syntax:
  `array.length;`

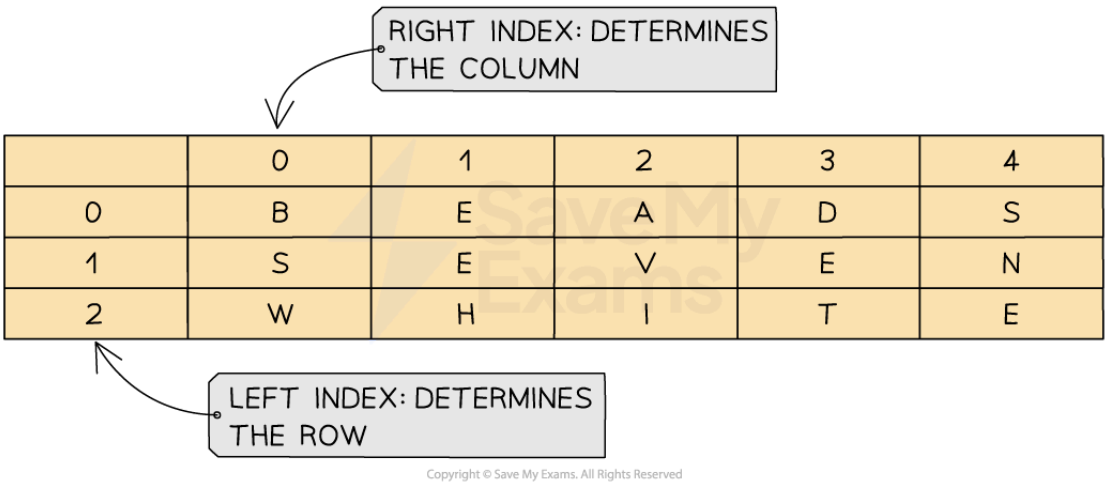
- In the example, the array has been iterated through to output each element within the array. A for loop has been used for this


Your notes

```
public class OneDimensionalArrayExample {
    public static void main(String[] args) {
        // Creating a one-dimensional array
        int[] array = {1, 2, 3, 4, 5};

        // Accessing elements of the array
        System.out.println(array[0]);   // Output: 1
        System.out.println(array[2]);   // Output: 3

        // Modifying elements of the array
        array[1] = 10;
        System.out.println(Arrays.toString(array));   // Output: [1, 10, 3, 4, 5]

        // Iterating over the array
        for (int i = 0; i < array.length; i++) {
            System.out.println(array[i]);
        }

        // Output:
        // 1
        // 10
        // 3
        // 4
        // 5

        // Length of the array
        int length = array.length;
        System.out.println(length);   // Output: 5
    }
}
```

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



Structure of a 2D array

Your notes

Example in Pseudocode

```
// Define the dimensions of the 2D array
ROWS = 3
COLS = 4

// Create a 2D array with the specified dimensions
array_2d = new Array[ROWS][COLS]

// Initialize the 2D array with values (optional)
for row = 0 to ROWS-1:
   for col = 0 to COLS-1:
      array_2d[row][col] = initial_value

// Accessing elements in the 2D array
value = array_2d[row_index][col_index]
```

Example in Python

```
# Method 1: Initialising an empty 2D array
rows = 3
cols = 4
array_2d = [[0 for _ in range(cols)] for _ in range(rows)]
# The above code creates a 2D array with 3 rows and 4 columns, filled with zeros.

# Method 2: Initialising a 2D array with values
array_2d = [[1, 2, 3],
            [4, 5, 6],
```

```
        [7, 8, 9]]
# The above code creates a 2D array with 3 rows and 3 columns, with the specified values.

# Accessing elements in the 2D array
print(array_2d[0][0])  # Output: 1
print(array_2d[1][2])  # Output: 6
```

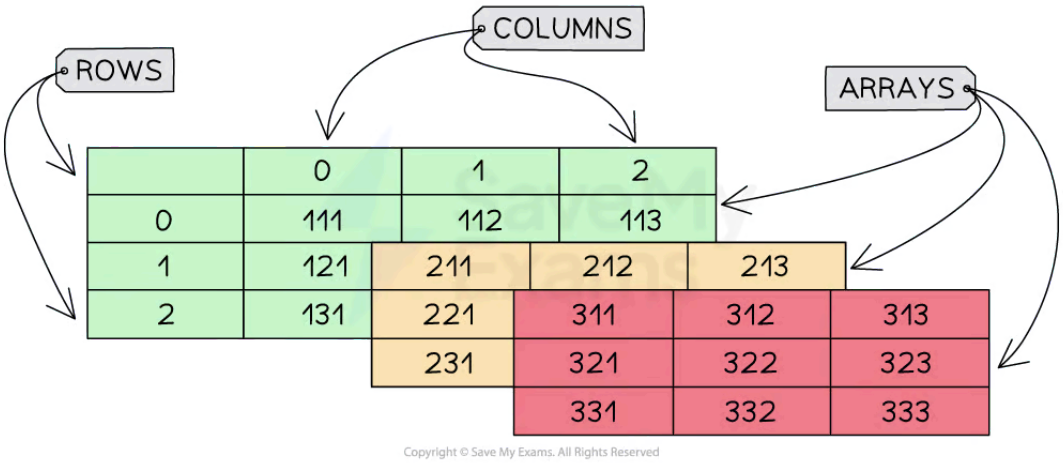
Example in Java

```
// Method 1: Initialising an empty 2D array
int rows = 3;
int cols = 4;
int[][] array2D = new int[rows][cols];
// The above code creates a 2D array with 3 rows and 4 columns, filled with zeros.

// Method 2: Initialising a 2D array with values
int[][] array2D = { {1, 2, 3},
                    {4, 5, 6},
                    {7, 8, 9} };
// The above code creates a 2D array with 3 rows and 3 columns, with the specified values.

// Accessing elements in the 2D array
System.out.println(array2D[0][0]); // Output: 1
System.out.println(array2D[1][2]); // Output: 6
```

3D Arrays

- A 3D array can be visualised as a multi-page spreadsheet and can also be thought of as multiple 2D arrays
- Selecting an element within a 3D array requires the following syntax to be used:
  `3DArrayName[z, y, x]`
- This is where z is the array index, y is the row index and x is the column index

Your notes



Structure of a 3D array

Example in Pseudocode

```
// Define the dimensions of the 3D array
ROWS = 3
COLS = 4
DEPTH = 2

// Create a 3D array with the specified dimensions
array_3d = new Array[ROWS][COLS][DEPTH]

// Initialize the 3D array with values (optional)
for row = 0 to ROWS-1:
   for col = 0 to COLS-1:
      for depth = 0 to DEPTH-1:
         array_3d[row][col][depth] = initial_value

// Accessing elements in the 3D array
value = array_3d[row_index][col_index][depth_index]
```

Example in Python

```
# Method 1: Initialising an empty 3D array
rows = 3
cols = 4
depth = 2
array_3d = [[[0 for _ in range(depth)] for _ in range(cols)] for _ in range(rows)]
# The above code creates a 3D array with 3 rows, 4 columns, and 2 depths, filled with zeros.
```

Your notes

```
# Method 2: Initialising a 3D array with values
array_3d = [[[1, 2], [3, 4], [5, 6], [7, 8]],
        [[9, 10], [11, 12], [13, 14], [15, 16]],
        [[17, 18], [19, 20], [21, 22], [23, 24]]]
# The above code creates a 3D array with the specified values.

# Accessing elements in the 3D array
print(array_3d[0][0][0])  # Output: 1
print(array_3d[1][2][1])  # Output: 14
```

Example in Java

```
// Method 1: Initialising an empty 3D array
int rows = 3;
int cols = 4;
int depth = 2;
int[][][] array3D = new int[rows][cols][depth];
// The above code creates a 3D array with 3 rows, 4 columns, and 2 depths, filled with zeros.

// Method 2: Initialising a 3D array with values
int[][][] array3D = { { {1, 2}, {3, 4}, {5, 6}, {7, 8} },
                { {9, 10}, {11, 12}, {13, 14}, {15, 16} },
                { {17, 18}, {19, 20}, {21, 22}, {23, 24} } };
// The above code creates a 3D array with the specified values.

// Accessing elements in the 3D array
System.out.println(array3D[0][0][0]);  // Output: 1
System.out.println(array3D[1][2][1]);  // Output: 14
```

Your notes

Records, Lists & Tuples


Your notes

Records

What is a Record?

- A record is a row in a file and is made up of fields. They are used in databases as shown in the example below

ID	firstName	surname
001	Tony	Smith
002	Peter	Pavard
003	Robert	Brown

- The above example contains 3 records, where each record has 3 fields. A record can be declared as follows

Example in Pseudocode

```
personDataType = record
 integer ID
 string firstName
 string surname
endRecord
```

Example in Python

```
class Person:
    def __init__(self, ID, firstName, surname):
        self.ID = ID
        self.firstName = firstName
        self.surname = surname

# Creating an instance of the Person record
person_record = Person(1, "Tony", "Smith")

# Accessing the fields of the Person record
print("ID:", person_record.ID)
```

```
print("First Name:", person_record.firstName)
print("Surname:", person_record.surname)
```

- In the code above, a class is defined called “Person” with an ‘__init__’ method that initialises the fields of the record
- Create an instance of the “Person” record called ‘person_record’ with an ID of 1, first name “Tony”, and surname “Smith”
- Finally access and print the values of the fields using dot notation (‘object.field’)


Your notes

Example in Java

```
public class Person {
    private int ID;
    private String firstName;
    private String surname;

    public Person(int ID, String firstName, String surname) {
        this.ID = ID;
        this.firstName = firstName;
        this.surname = surname;
    }

    public int getID() {
        return ID;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getSurname() {
        return surname;
    }

    public void setID(int ID) {
        this.ID = ID;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }
}
```

- Define a class called 'Person' with private fields for ID, firstName, and surname
- Then a constructor that allows you to set these fields when creating an instance of the record
- Additionally, provide getter and setter methods for each field to access and modify their values
- To use the record in Java, create an instance of 'Person' class and set or retrieve the field values using the provided methods
- For example:


Your notes

```
public class Main {
  public static void main(String[] args) {
    Person personRecord = new Person(1, "Tony", "Smith");

    // Accessing the fields of the Person record
    int ID = personRecord.getID();
    String firstName = personRecord.getFirstName();
    String surname = personRecord.getSurname();

    System.out.println("ID: " + ID);
    System.out.println("First Name: " + firstName);
    System.out.println("Surname: " + surname);
  }
}
```

- This creates an instance of the 'Person' record called 'personRecord' with an ID of 1, first name "Tony", and surname "Smith"
- Then uses the getter methods to retrieve the field values and print them to the console

Lists

What is a List?

- A list is a data structure that consists of a number of items where the items can occur more than once
- Lists are similar to 1D arrays and elements can be accessed in the same way. The difference is that list values are stored non-contiguously
- This means they do not have to be stored next to each other in memory, which is different to how data is stored for arrays
- Lists can also contain elements of more than one data type, unlike arrays

Example in Pseudocode

```
list pets = ["dog", "cat", "rabbit", "fish"]
```

Example in Python

```
pets = ["dog", "cat", "rabbit", "fish"]
```

Example in Java

```
import java.util.ArrayList;
import java.util.List;

public class Main {
   public static void main(String[] args) {
      List<String> pets = new ArrayList<>();

      // Adding elements to the pets list
      pets.add("dog");
      pets.add("cat");
      pets.add("rabbit");
      pets.add("fish");

      // Printing the elements in the pets list
      for (String pet : pets) {
         System.out.println(pet);
      }
   }
}
```

Worked Example

A programmer has written the following code designed to take in ten names then print them in a numbered list.

```
name1 = input("Enter a name: ")
name2 = input("Enter a name: ")
name3 = input("Enter a name: ")
name4 = input("Enter a name: ")
name5 = input("Enter a name: ")
name6 = input("Enter a name: ")
name7 = input("Enter a name: ")
name8 = input("Enter a name: ")
name9 = input("Enter a name: ")
name10 = input("Enter a name: ")
```


Your notes

```
print("1. " + name1)
print("2. " + name2)
print("3. " + name3)
print("4. " + name4)
print("5. " + name5)
print("6. " + name6)
print("7. " + name7)
print("8. " + name8)
print("9. " + name9)
print("10. " + name10)
```

It has been suggested that this code could be made more efficient and easier to maintain using an array or a list.

Write a more efficient version of the programmer's code using an array or a list.

5 marks

How to answer this question:

- Start by having a variable names which is initialised as an empty list []
  `names = []`
- Then have a for loop run from i = 0 to i = 9 (inclusive), indicating that it will iterate 10 times
  `for i = 0 to 9`
- Inside the loop, the input() function is used to prompt the user to enter a name. The entered name is then appended to the names list using the append() method
  `names.append(input("Enter a name: "))`
- After the first for loop finishes executing, the list names will contain 10 names entered by the user
- Then a second for loop also runs from i = 0 to i = 9 (inclusive)
  `for i = 0 to 9`
- Inside this loop, the print() function is used to display the index of each name (incremented by 1) along with the name itself. The expression i + 1 represents the current index, and (i + 1) + ". " is concatenated with names[i] to form the output string
  `print((i+1)+". "+names[i])`
- The loop repeats this process for each name in the names list, printing the index number and corresponding name on separate lines

Answer:

Example answer that gets full marks:

```
names = []
for i = 0 to 9
  names.append(input("Enter a name: "))
next i
for i = 0 to 9
```

Your notes

```
    print((i+1)+". "+names[i])
next i
```

Declaration of list/array [1]
For loop with runs ten times [1]
Inputting name to correct location each interaction [1]
For/while loop which outputs each name [1]
Names are formatted with numbers 1–10 and a dot preceding each one [1]


Your notes

Manipulating Lists

There are a range of operations that can be performed involving lists in the table below:

List Operation	Description	Pseudocode
isEmpty()	Checks if the list is empty	Pets.isEmpty()
append(value)	Adds a new value to the end of the list	Pets.append(dog)
remove(value)	Removes the value the first time it appears in the list	Pets.remove(cat)
length()	Returns the length of the list	Pets.length()
index(value)	Returns the position of the item	Pets.index(rabbit)
insert(position, value)	Inserts a value at a given position	Pets.insert(3,snake)
pop()	Returns and removes the last value	Pets.pop()

Tuples

What is an Tuple?

- A tuple is an ordered set of values of any type
- It is immutable, this means that it can't be changed
- Elements, therefore, can't be added or removed once it has been created
- Tuples are initialised using regular brackets instead of square brackets

- Elements in a tuple are accessed in a similar way to elements in an array, with the exception that values can't be changed


Your notes

Example in Pseudocode

```
tuple1 = (1, 2, 3)
```

Example in Python

```
person = ("John", 25, "USA")

print(person)
```

When should Arrays, Records, Lists & Tuples be used?

Data Structure	Appropriate Usage	Possible use cases
Array	A fixed-size collection of elements and need random access to elements inside by index. They are efficient for accessing elements by index but less efficient for dynamic resizing of insertion/removal of elements in the middle.	List of student grades A table of numbers for a mathematical function A grid of pixels for an image
Records	A group of related data which is a single entity. They are useful for organising and manipulating data with different attributes of fields.	A shopping list A to-do list A list of contacts in an address book
Lists	A dynamic collection of elements that may change in size. They provide flexibility for adding, removing and modifying elements.	A customer record in a database A student record in a school system A product record in an online store

Your notes

Linked Lists

What is a Linked List?

- A linked list is a dynamic data structure that is used to hold an ordered sequence
- The items which form the sequence do not have to be in contiguous data locations
- 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
- Linked lists also store the index of the first item as a pointer
- They also store a pointer identifying the index of the next available space

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

Linked Lists: Traversing, Adding & Removing Data

Traverse a linked list


Your notes

- 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'	-

Your notes

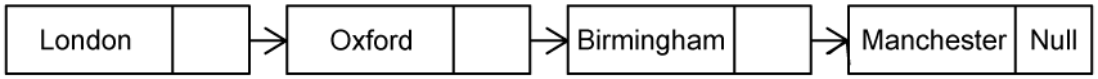
2. When traversed this linked list will now output: 'Pineapple', 'Melon'

- 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

Worked Example

A coach company offers tours of the UK

A linked list stores the names of cities on a coach tour in the order they are visited.



linked-lists-question

The tour is amended. The new itinerary is London, Oxford, Manchester then York. Explain how Birmingham is removed from the linked list and how York is added. You may use the diagram to illustrate your answer.

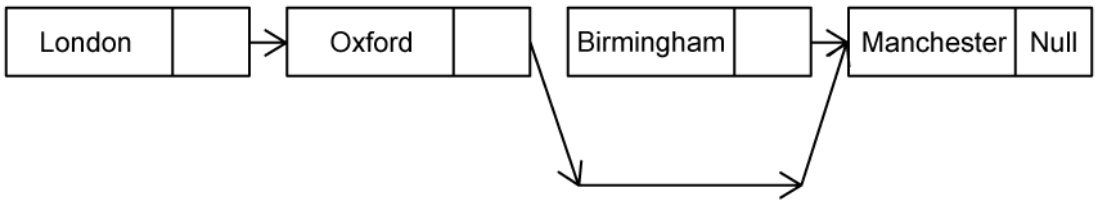
4 marks

Answer:

Example answer that gets full marks:

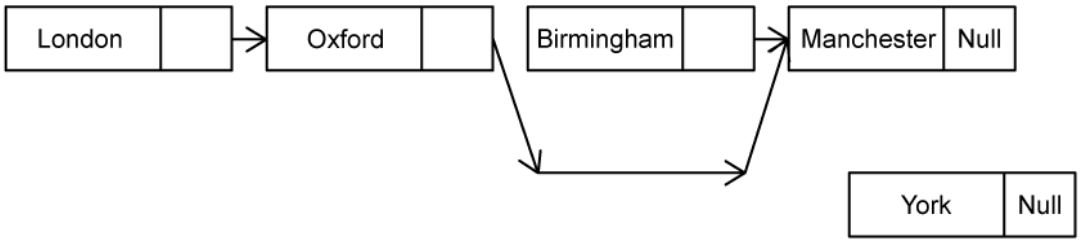
Your notes

The first thing you need to do is change the Oxford pointer to go around Birmingham to point to Manchester. **[1]**



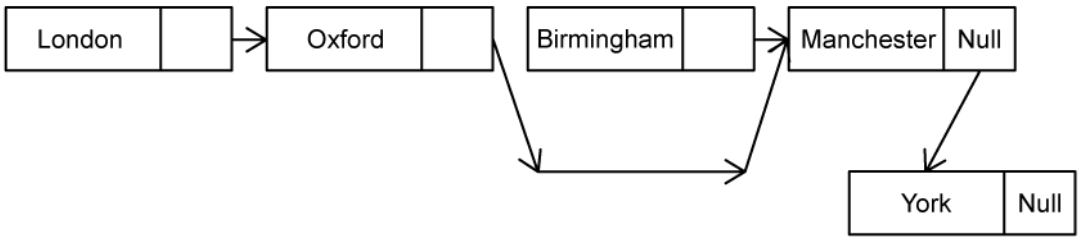
linked lists WE 1

Then you create a node called York and this is placed at the next free space in the list. **[1]**



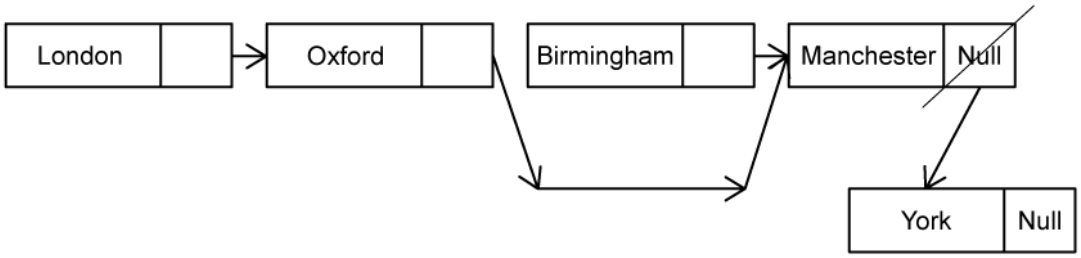
linked lists WE 2

Manchester is kept in the same position and it's pointer changed to point to York. **[1]**



linked lists WE 3

The York node then points to Null. **[1]**



linked lists WE 4

Your notes

Your notes

Stacks

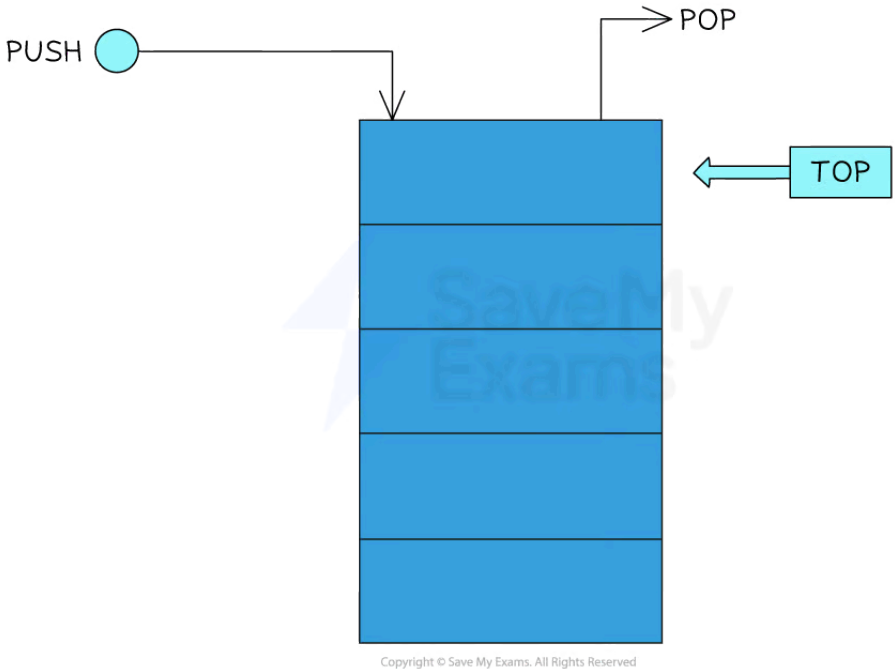
What is a Stack?

- A **stack** is a last in first out (**LIFO**) data structure
- Items can only be added to or removed from the top of the stack
- Stacks are key structures in the computing world as they are used to reserve an action, such as go back a page or undo
- A real-life example of a stack would be a pile (or stack) of plates at a buffet; you can only take a plate from the top, and if you need to reach the bottom one you have to remove all the others first
- It is often implemented using an array
- Where the maximum size required is known in advance, static stacks are preferred as they are easier to implement and make more efficient use of memory
- A stack has 6 main operations, which can be seen below

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)

Your notes



Adding & Removing Data From a Stack

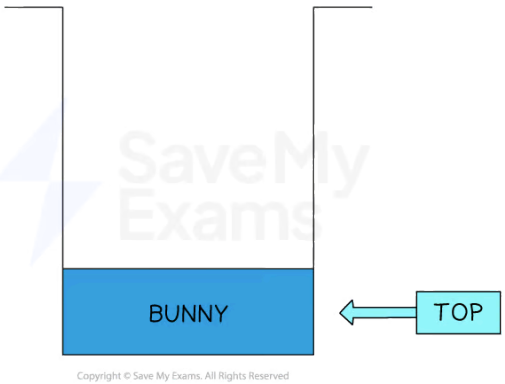
Pushing Data to a Stack

- 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

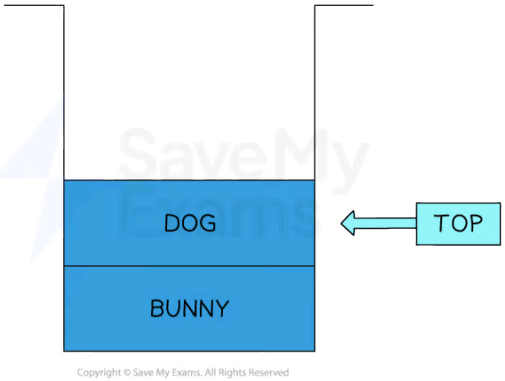
Visual Example

- This would push Bunny onto the empty stack

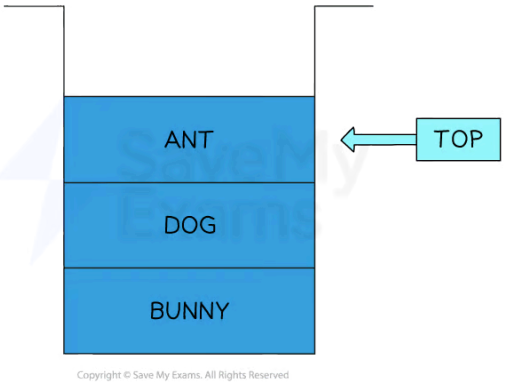
Your notes



- This would push Dog to the top of the stack



- This would push Ant to the top of the stack



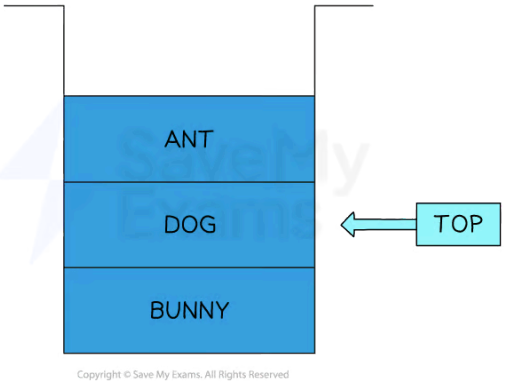
Popping Data From a Stack

Your notes

- 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

Visual Example

- 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

Worked Example

Stacks and queues are both data structures.

A stack is shown below before a set of operations are carried out on it.

Draw what the stack shown below would look like after the following operations:

pop(), push("A"), push("B"), pop(), push("E"), push("F")

Pointer	Data
→	Z

	Y
	X

2 marks

Answer:

- Start by using the operation pop() to remove Z from the top of the stack

Pointer	Data
→	Y
	X

- Use the operation push("A") to add A to the top of the stack

Pointer	Data
→	**A**
	Y
	X

- Then use the operation push("B") to add B to the top of the stack

Pointer	Data
→	**B**
	A
	Y
	X

- Then use pop() to remove the top item in the stack

Pointer	Data
→	**A**
	Y
	X

- Then use push("E") to add E to the top of the stack

Pointer	Data

Your notes

→	**E**
	A
	Y
	X

- Finally use push("F") to add F to the top of the stack

Pointer	Data
→	**F**
	E
	A
	Y
	X

- F at the top of the stack with E directly below it [1]
- A,Y,X. directly below E (with no other entries) [1]

F
E
A
Y
X


Your notes


Your notes

Queues

What is a Queue?

- A queue is a First in First out (**FIFO)** data structure
- Items are added to the end of the queue and removed from the front
  - Imagine a queue in a shop where customers are served from the front and new customers join the back
- A queue has a back (tail) pointer and a front (head) pointer.
- Much the same as with a stack, an attempt to **enqueue** an item when the queue is full is called a **queue overflow**. Similarly, trying to **dequeue** an item from an empty queue is called a **queue underflow**.
- A queue can be static or dynamic
- There are three types of queues
  1. Linear Queue
  2. Circular Queue
  3. Priority Queue

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)

Queue Keywords

Keyword	Definition
Queue	An abstract data structure that holds an ordered, linear sequence of items. It is a First in First out structure.
FIFO	First In First Out
Static Data Structure	Has a fixed size and can't change at run time.
Dynamic Data Structure	Able to adapt and accommodate changes to the data inside it so it doesn't waste as much space in memory.
Pointer	An object that stores a memory address

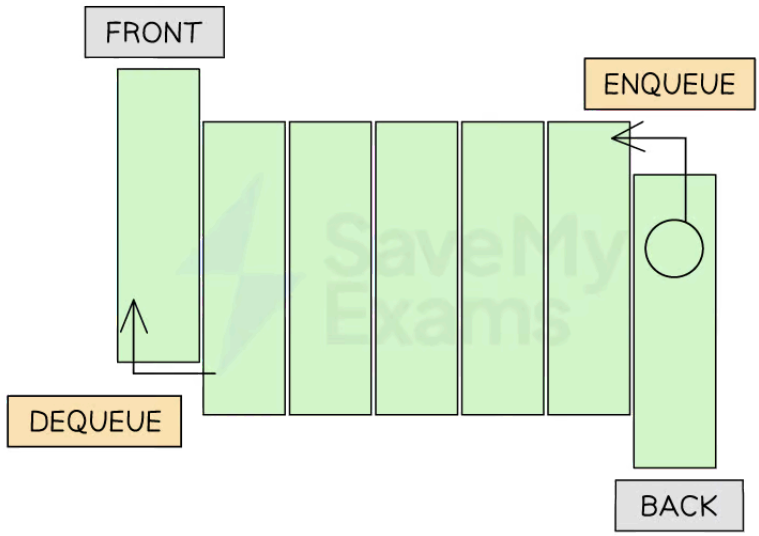

Your notes

Linear Queues

What Are Linear Queues?

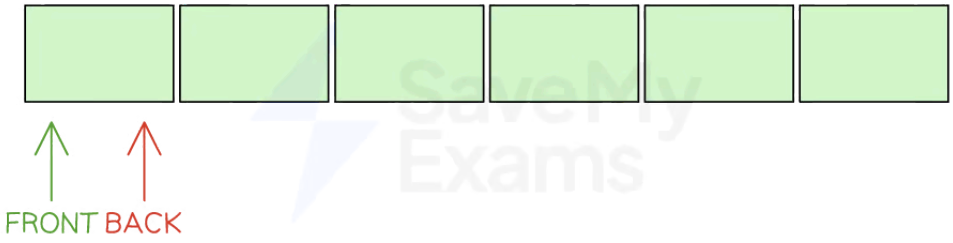
- A linear queue is a data structure that consists of an array.
- Items are added to the next available space in the queue starting from the front
- Items are then removed from the front of the queue

Your notes

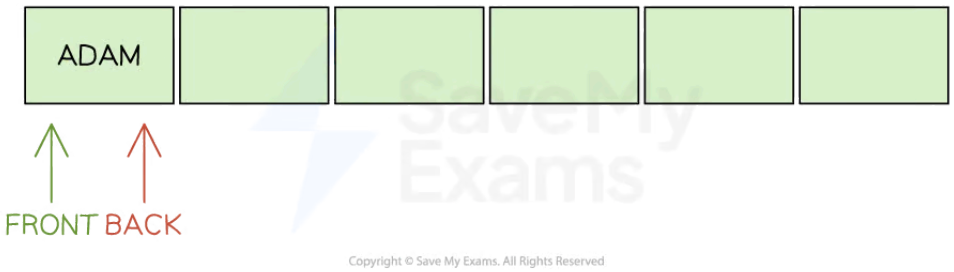


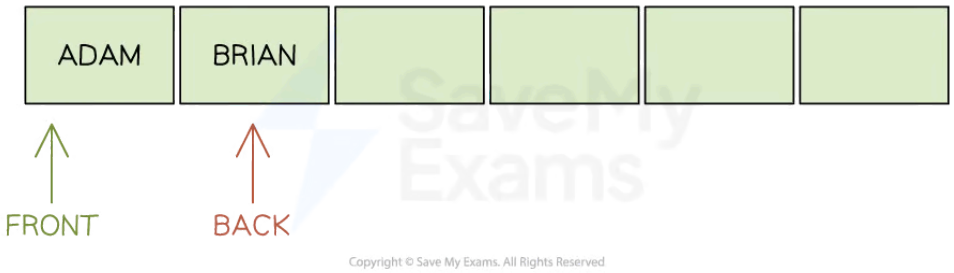
Enqueue Data

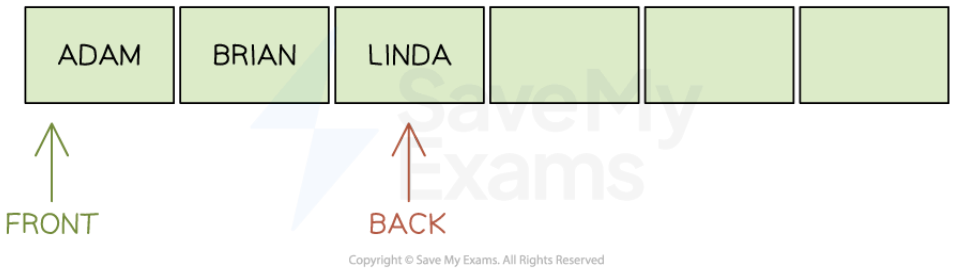
- Before adding an item to the queue you need to check that the queue is not full
- If the end of the array has not been reached, the rear index pointer is incremented and the new item is added to the queue
- In the example below, you can see the queue as the data Adam, Brian and Linda are enqueued.



Your notes

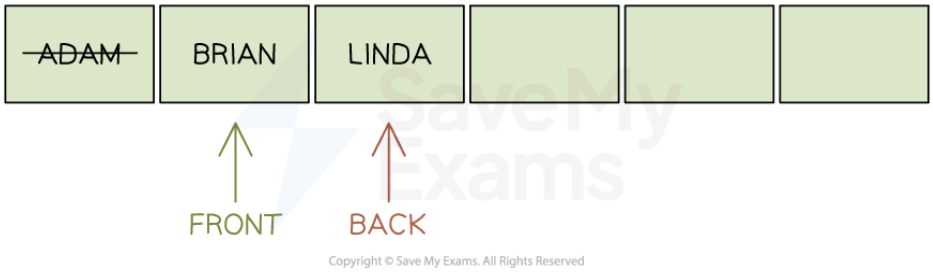






Dequeue Data

- 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



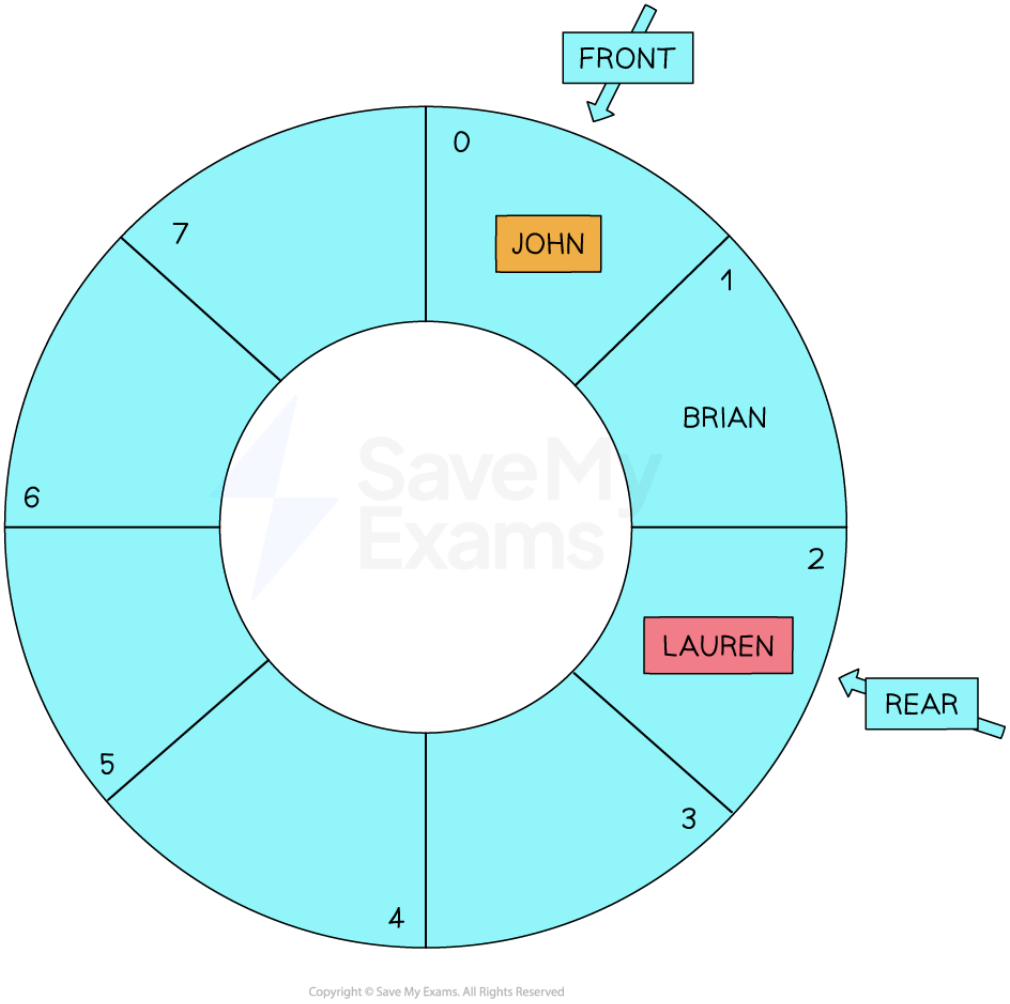
Your notes

Circular Queues

What Are Circular Queues?

- A static array that has a fixed capacity which 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

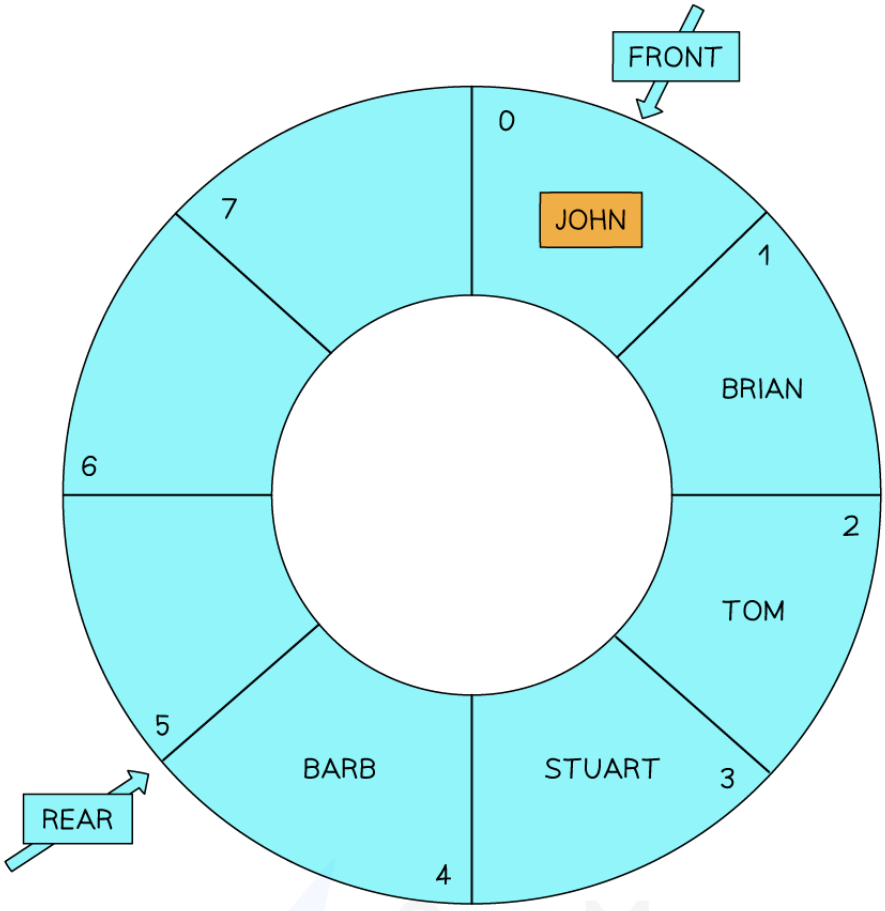
Your notes

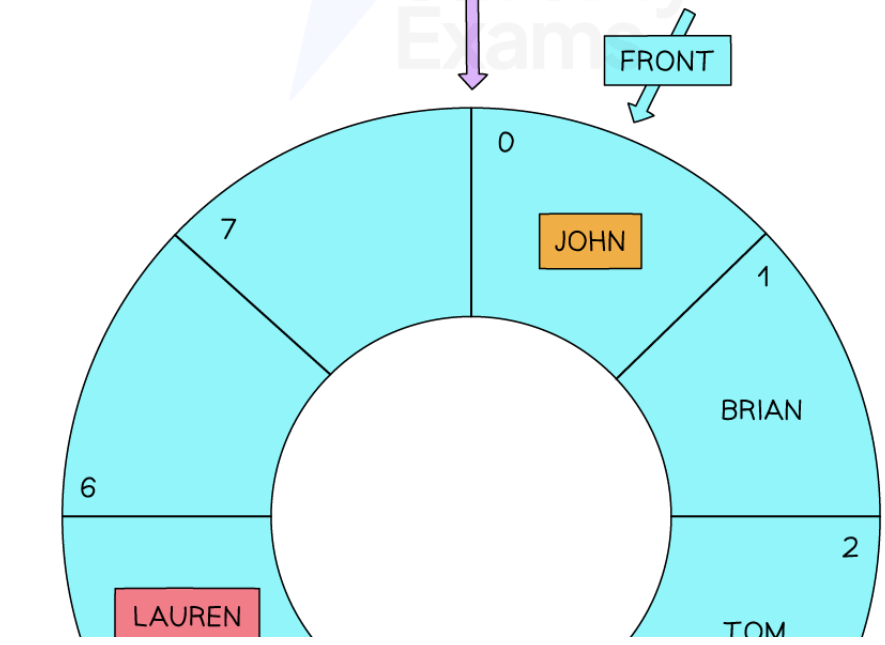


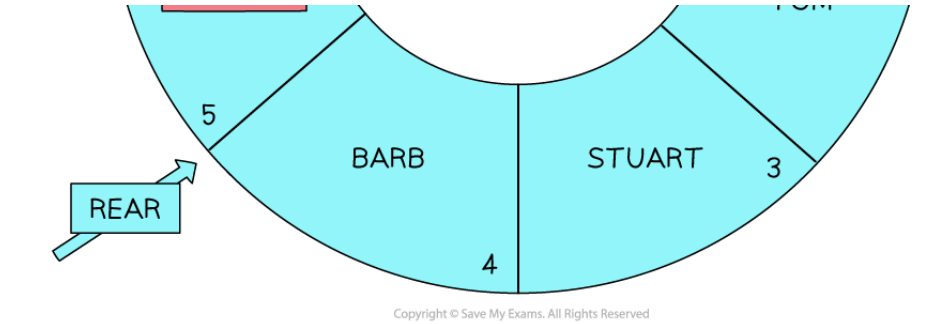
Enqueue Data

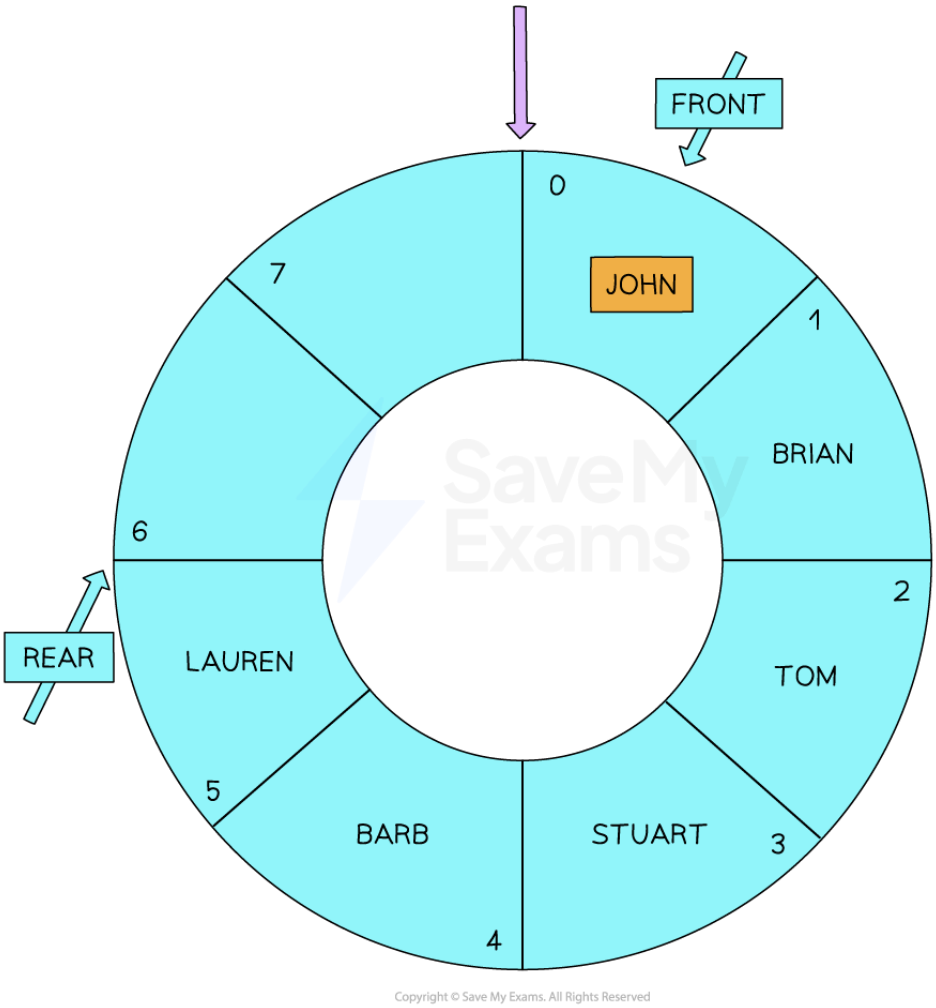
- Before adding an item to the circular queue you need to check that the array is not full
- It will be full if the next position to be used is already occupied by the item at the front of the queue
- If the queue is not full then the rear index pointer must be adjusted to reference the next free position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

Your notes









Your notes

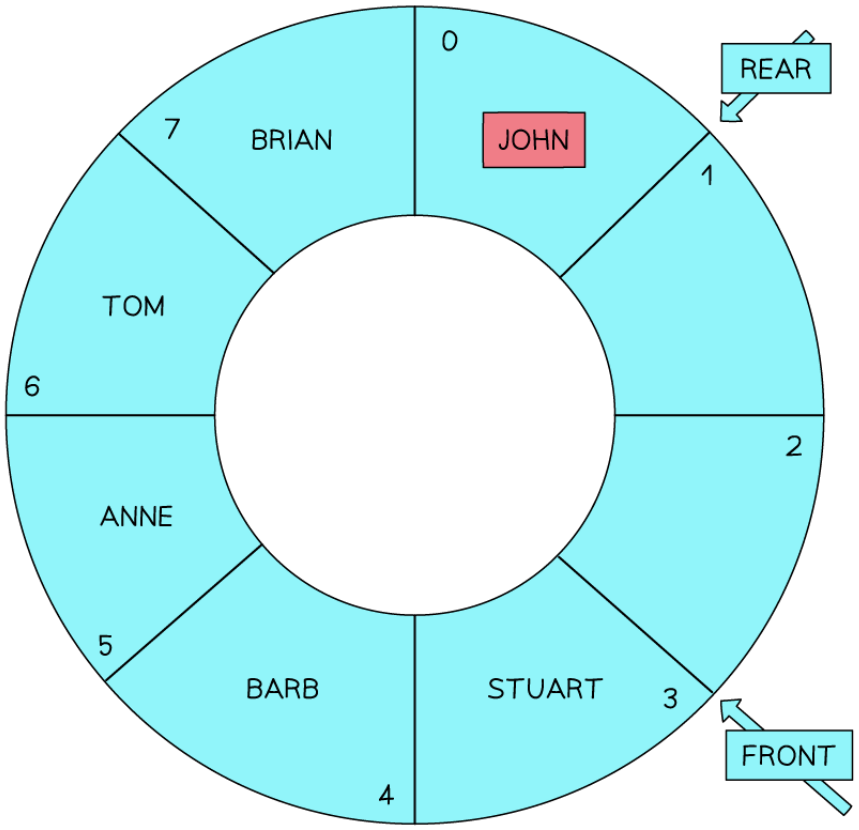
Dequeue Data

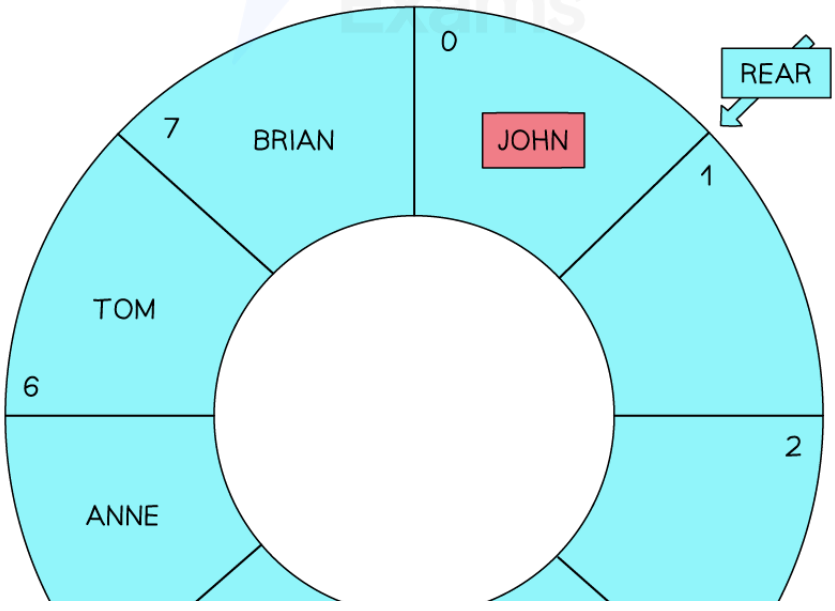
- Before removing an item from the queue, you need to make sure that the queue is not empty

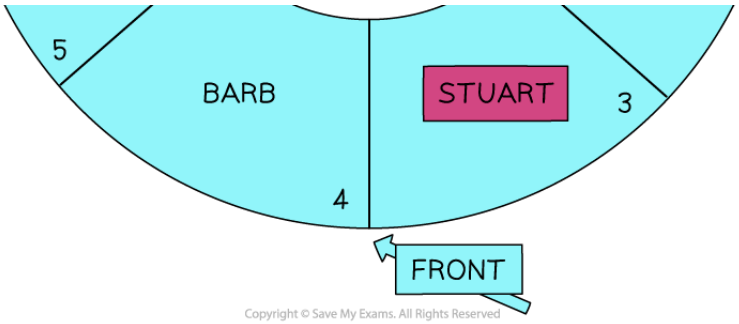
- If the queue is not empty the item at the front of the queue is returned
- If this is the only item that was in the queue the rear and front pointers are reset, otherwise the pointer moves to reference the next item in the queue
- In the example below, you can see the queue as the data Adam is dequeued

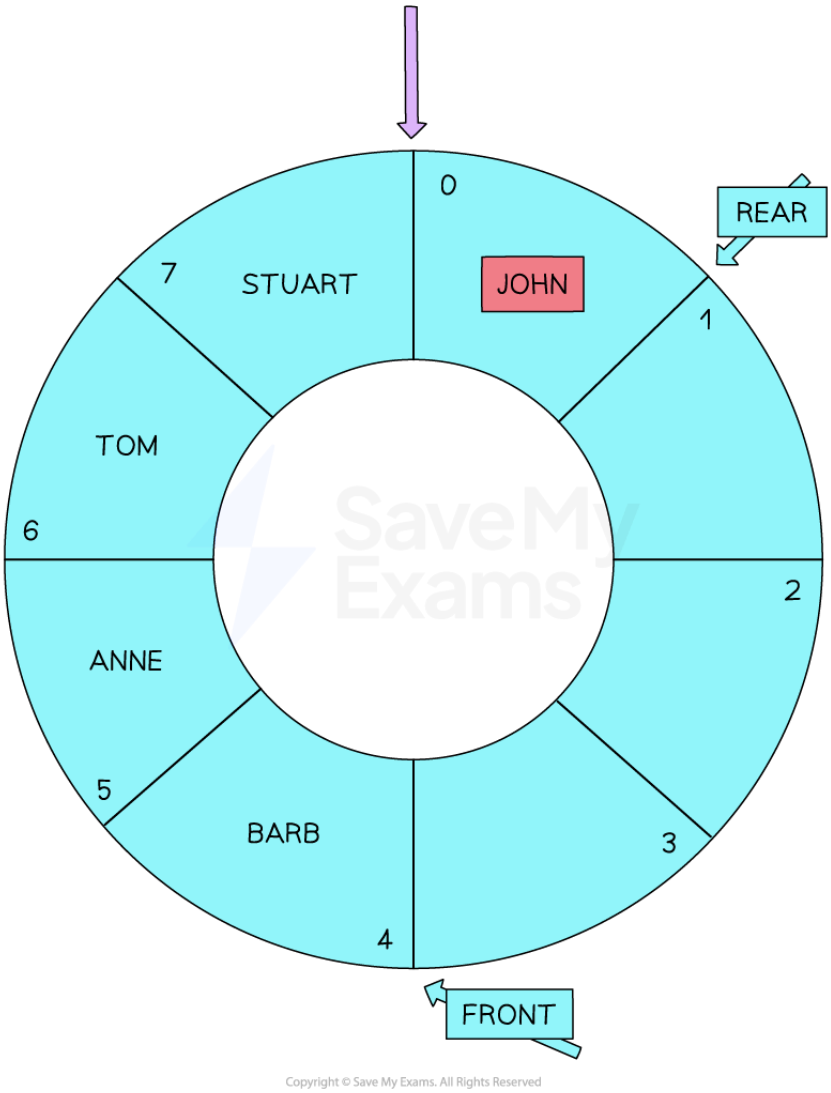

Your notes

Your notes









Your notes

Your notes

Examiner Tips and Tricks

When asked to display a queue visually, make sure that you also identify the position of the front and rear pointers.

Worked Example

The current contents of a queue called 'colours', have been implemented in an array are shown below:

0	1	2	3	4	5	6	7
red	**yellow**	**green**	**blue**	**grey**			
↑ *front*				↑ *back*			

Front = 0
End = 4

The queue has the subprograms enqueue and dequeue. The subprogram 'enqueue' is used to add items to the queue and the subprogram 'dequeue' removes items from the queue.

Use the following diagram to show the queue shown above after the following program statements have run:

```
enqueue("orange")
dequeue()
enqueue("maroon")
dequeue()
dequeue()
```

4 marks

Answer:

1. The first thing is to enqueue orange.

0	1	2	3	4	5	6	7
red	**yellow**	**green**	**blue**	**grey**	**orange**		
↑ *front*					↑ *back*		

2. Then apply a dequeue() this will remove the first item in the queue.

0	1	2	3	4	5	6	7
	yellow	**green**	**blue**	**grey**	**orange**		
	↑ *front*				↑ *back*		

3. Then enqueue maroon.

0	1	2	3	4	5	6	7
	yellow	**green**	**blue**	**grey**	**orange**	**maroon**	
	↑ *front*					↑ *back*	

4. Then apply another dequeue to remove the next item from the front.

0	1	2	3	4	5	6	7
		green	**blue**	**grey**	**orange**	**maroon**	
		↑ *front*				↑ *back*	

5. Then apply the final dequeue command to remove the next item from the front.

0	1	2	3	4	5	6	7
			blue	**grey**	**orange**	**maroon**	
			↑ *front*			↑ *back*	

- Elements are in the queue (correct four colours)
- … in the correct positions
- 'Front' points to the first element in the queue
- 'End 'points to the last element in the queue

Your notes

Your notes

Graphs

What is a Graph?

- A graph is a set of vertices/nodes that are connected by edges/pointers. Graphs can be placed into the following categories:
- **Directed Graph:** The edges can only be traversed in one direction
- **Undirected Graph:** The edges can be traversed in both directions
- **Weighted Graph:** A value is attached to each edge

Computers are able to process graphs by using either an **adjacency matrix** or an **adjacency list.**

The examples below will be illustrated using an adjacency matrix, more information on programming using both matrices and lists is available in section 8.

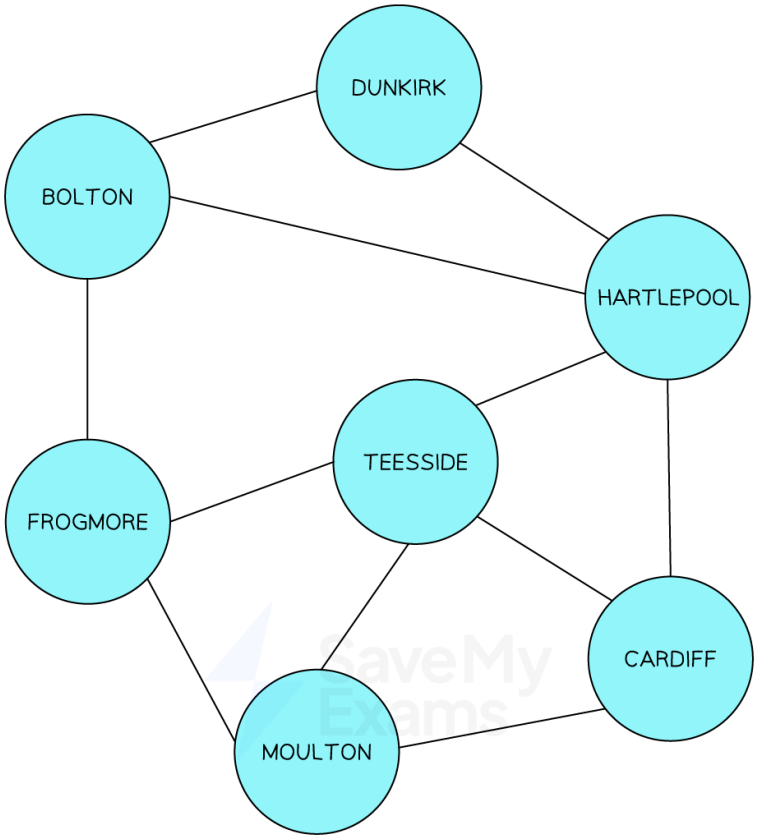
Undirected, Unweighted Graph

For unweighted graphs:

- 1 is set when an edge exists between 2 nodes
- 0 is set when there is no edge between 2 nodes

Below is the adjacency matrix for an undirected, unweighted graph:

Your notes



	B	C	D	F	H	M	T
B	0	0	1	1	1	0	0
C	0	0	0	0	1	1	1
D	1	0	0	0	1	0	0
F	1	0	0	0	0	1	1
H	1	1	1	0	0	0	1
M	0	1	0	1	0	0	1
T	0	1	0	1	1	1	0

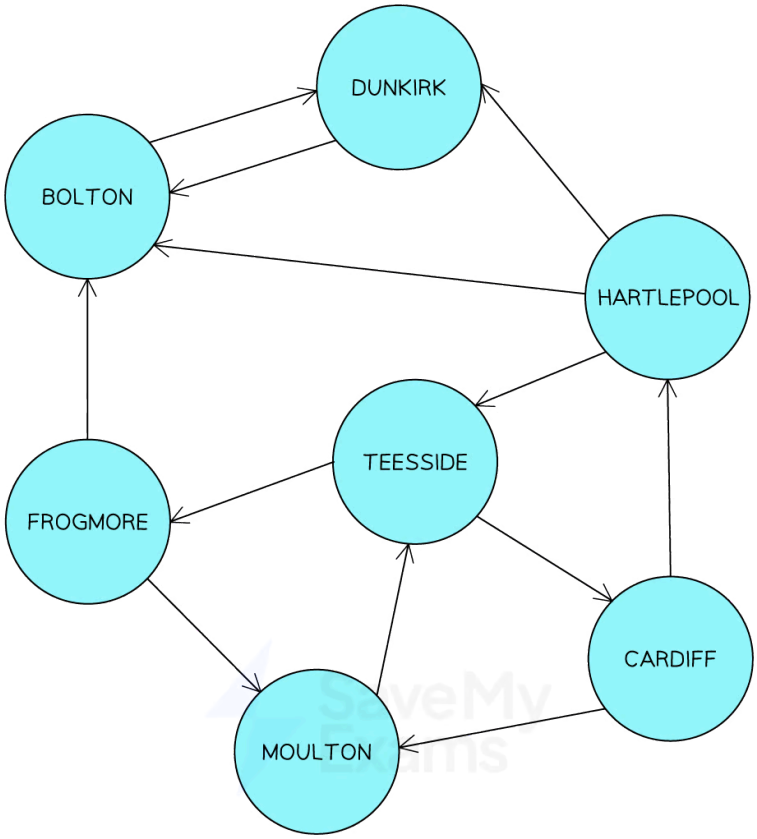
For this graph, the names are abbreviated to the first letter.

- The data in an adjacency matrix for an unweighted graph is symmetric
- To determine the presence or absence of an edge, you inspect the row (or column) that represents a node. *Example: If you wanted to see if there is an edge between Frogmore and Hartlepool you can look at the cross-section of row 3 (for Frogmore) and column 4 (for Hartlepool)*
- You need a method (e.g **Dictionary**) to record which row/column is used for a specific node's edge data
- The data values for the nodes (in this example names of towns) are not stored in the matrix


Your notes

Directed, Unweighted Graph

Your notes



	B	C	D	F	H	M	T
B	0	0	1	0	0	0	0
C	0	0	0	0	1	1	0
D	1	0	0	0	0	0	0
F	1	0	0	0	0	1	0
H	1	0	1	0	0	0	1
M	0	0	0	0	0	0	1
T	0	1	0	1	0	0	0

Above is the adjacency matrix for a directed, unweighted graph.
For this graph, the names are abbreviated to the first letter.

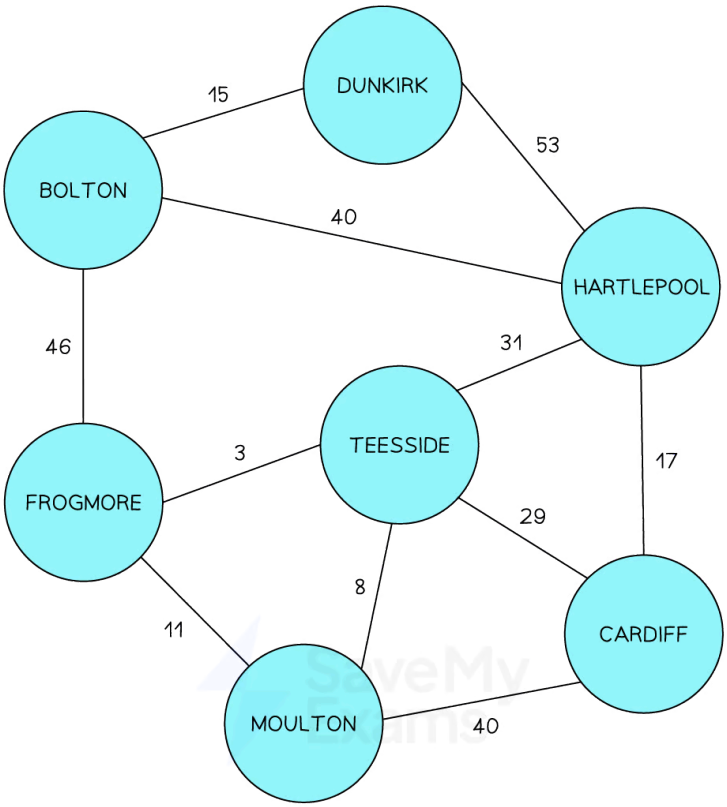
- Due to the lack of symmetry in this matrix, you cannot access the data by both row and column, you must know the direction in which the values are stored
- *E.g.: In row 0 for Bolton there is only one neighbour (Dunkirk) because there is a single directed edge from Bolton towards Dunkirk. This is because there are 3 edges directed towards Bolton (from Dunkirk, Hartlepool and Frogmore)*
- If you are implementing a graph as an adjacency matrix you will need to choose the direction and make sure that the data is stored and accessed correctly


Your notes

Undirected, Weighted Graph

- For weighted graphs the values of the weights are stored in the adjacency matrix
- If an edge does not exist between two nodes, then a very large number (normally the infinity symbol ∞) is set

Your notes



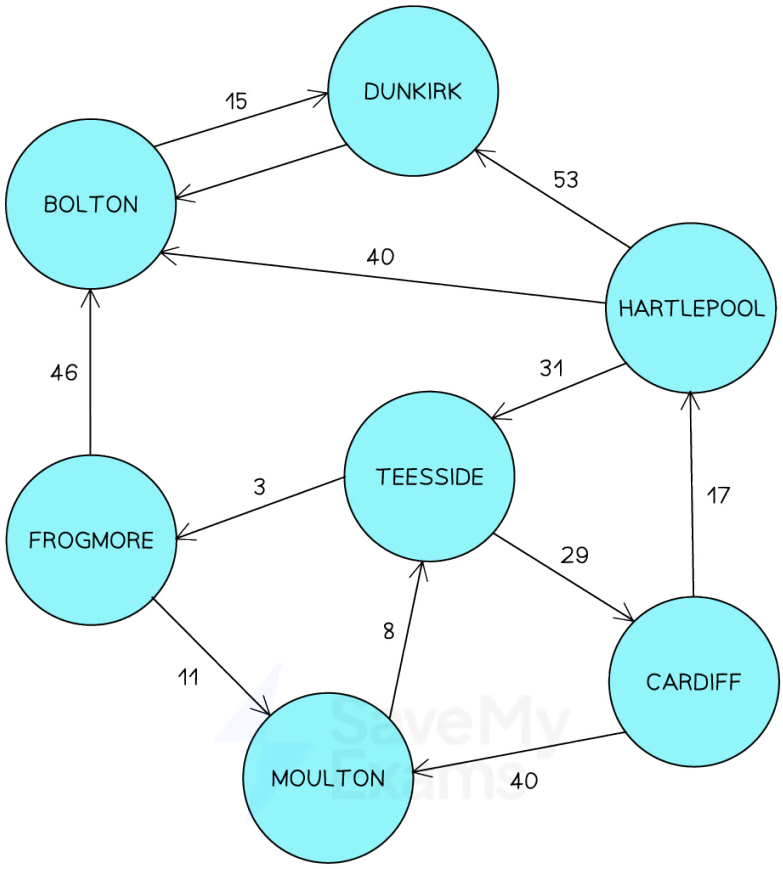
	B	C	D	F	H	M	T
B	∞	∞	15	46	40	∞	∞
C	∞	∞	∞	∞	17	40	29
D	15	∞	∞	∞	53	∞	∞
F	46	∞	∞	∞	∞	11	3
H	40	17	53	∞	∞	∞	31
M	∞	40	∞	11	∞	8	8
T	∞	29	∞	3	31	∞	∞

Above is the adjacency matrix for an undirected, unweighted graph.
For this graph, the names are abbreviated to the first letter.

Directed, Weighted Graph


Your notes

Your notes



	B	C	D	F	H	M	T
B	∞	∞	15	∞	∞	∞	∞
C	∞	∞	∞	∞	17	40	∞
D	15	∞	∞	∞	∞	∞	∞
F	46	∞	∞	∞	∞	11	∞
H	40	∞	53	∞	∞	∞	31
M	∞	∞	∞	∞	∞	∞	8
T	∞	29	∞	3	∞	∞	∞

Above is the adjacency matrix for a directed weighted graph:
For this graph, the names are abbreviated to the first letter.

Your notes

The Applications of Graphs

Graphs have many uses in the world of Computer Science, for example:

- Social Networks
- Transport Networks
- Operating Systems

Keyword	Definition
Directed Graph	A directed graph is a set of objects that are connected together, where the edges are directed from one vertex to another.
Undirected Graph	An undirected graph is a set of objects that are connected together, where the edges do not have a direction.
Weighted Graph	**A weighted graph is a set of objects that are connected together, where a weight is assigned to each edge.**
Adjacency Matrix	Also known as the connection matrix is a matrix containing rows and columns which is used to present a simple labelled graph.
Adjacency List	**This is a collection of unordered lists used to represent a finite graph.**
Vertices/Nodes	A vertex (or node) of a graph is one of the objects that are connected.
Edges/Arcs	An edge (or arc) is one of the connections between the nodes (or vertices) of the network.
Dictionary	A collection of names, definitions, and attributes about data elements that are being used or captured in a database, information system, or part of a research project.

Examiner Tips and Tricks

It is not important what the graph looks like, but which vertices are connected.


Your notes


Your notes

Graphs: Traversing, Adding & Removing Data

How do you Traverse a Graph?

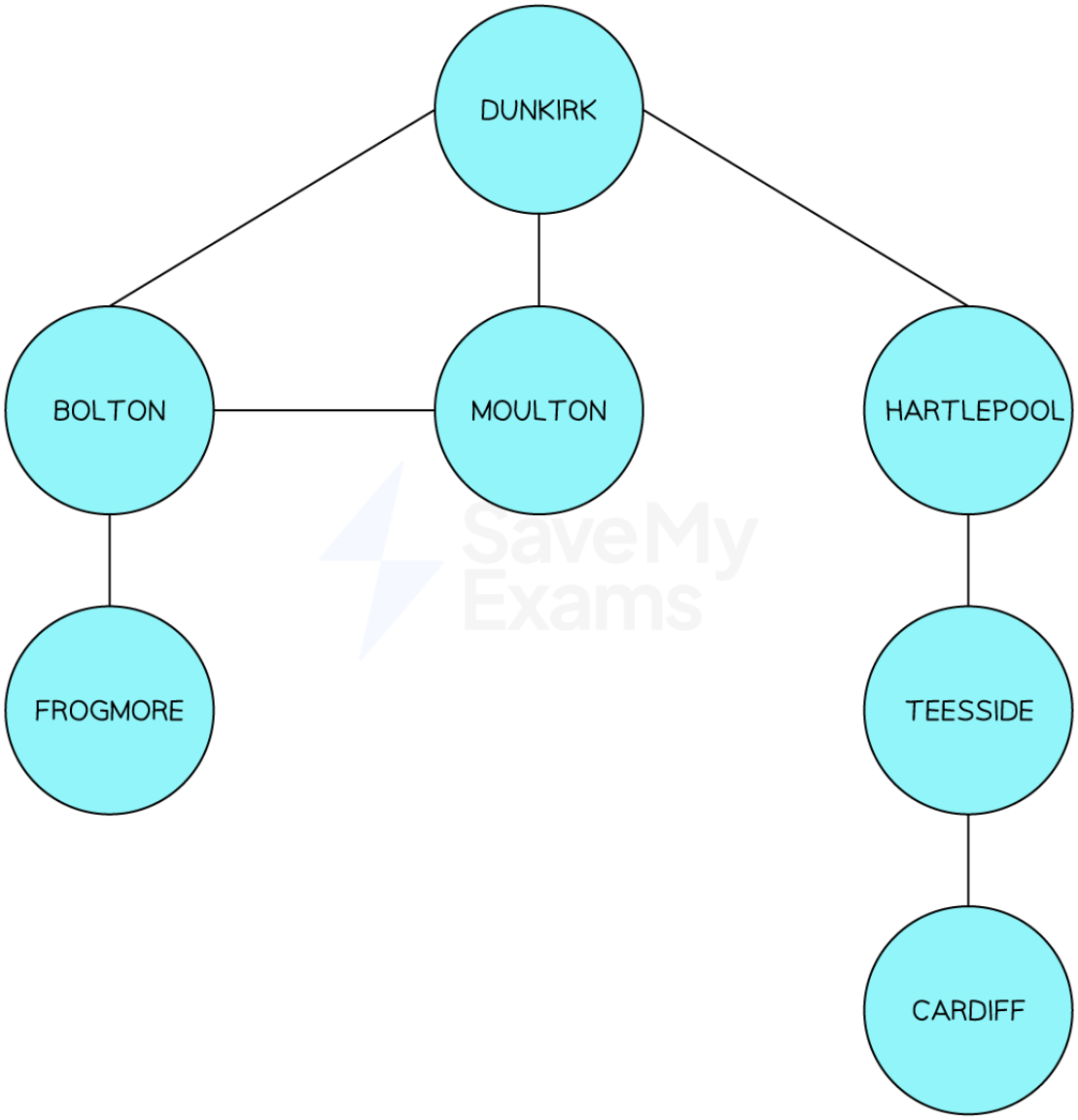
There are two approaches to traversing a graph:

- A breadth-first search
- A depth breadth-first search

Breadth-First Search

- A breadth-first search is a graph traversal algorithm which systematically visits all neighbours of a given vertex before moving on to their neighbours. It then repeats this layer by layer.
- This method makes use of a queue data structure to enqueue each node as long as it is not already in a list of visited nodes.

Your notes



1. Take the example above, the root node, Dunkirk, would be set as the current node and then added to the visited list

Dunkirk

1. Moving from left to right, we must check if the connected node isn't already in the visited list, if it is not, it is **enqueued** to the queue.

Your notes

The first version of the queue would have appeared as

Front Pointer	Back Pointer	Position	Data
		5	
		4	
		3	
→	→	2	Hartlepool
→	→	1	Moulton
→	→	0	Bolton
→	→	-1	Empty

1. That linked vertex is then added to the visited list. Finally, it is removed from the queue as this process repeats. This means that all nodes, from left to right are enqueued to the queue in turn, before being added to the visited list, and then dequeued from the queue
2. Output all of the visited vertices

Our final visited list would appear as

Dunkirk, Bolton, Moulton, Hartlepool, Frogmore, Teesside, Cardiff

The final version of the queue would have appeared as

Front Pointer	Back Pointer	Position	Data
→	→	5	Cardiff
→	→	4	Teesside
→	→	3	Frogmore

→	→	2	Hartlepool
→	→	1	Moulton
→	→	0	Bolton
→	→	-1	Empty


Your notes

Depth-First Search

- A depth-first search is a graph traversal algorithm which uses an edge-based system.
- This method makes use of a stack data structure to push each visited node onto the stack and then pop them from the stack when there are no nodes left to visit.

Examiner Tips and Tricks

There is no right method to use when performing a depth-first search, it simply depends on how the algorithm is implemented. Most mark schemes commonly traverse the left-most path first, therefore that will be used in this example.

Using the example from above, Dunkirk, would be set as the current node and then added to the visited list

Dunkirk

1. Then check if the connected node isn't already in the visited list, if it is not, it pushed to the stack.

The first version of the stack would appear as

Pointer	Position	Data
	5	
	4	
	3	

→	2	Bolton
→	1	Moulton
→	0	Hartlepool
→	-1	Empty

Your notes

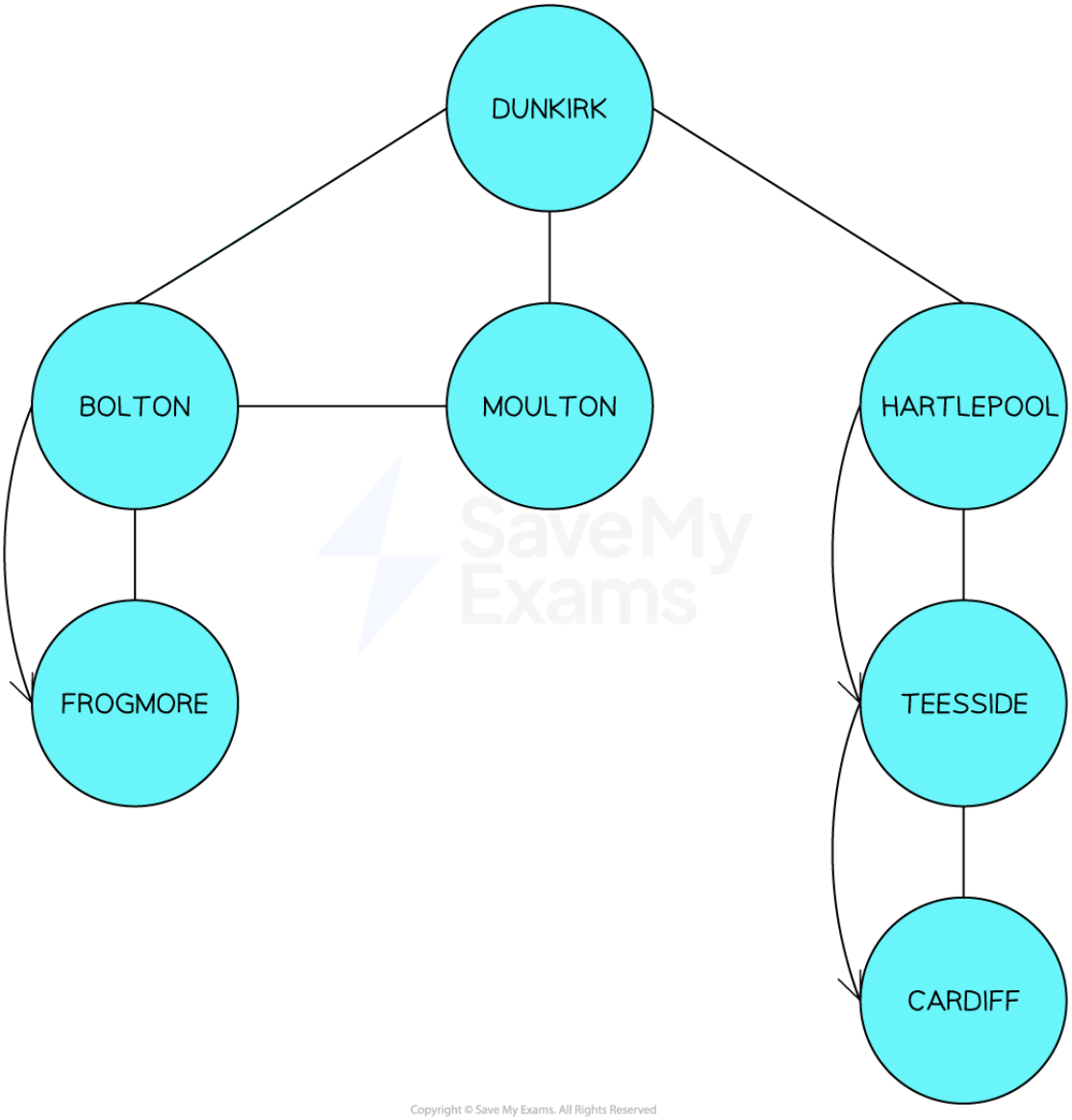
1. Next, any connected node that is not on the visited list is then pushed to the stack.
2. Then pop the stack to remove the item and set it as the current node.
3. Output all of the visited nodes

Our final visited list would appear as

Dunkirk, Bolton, Frogmore, Moulton, Hartlepool, Teesside, Cardiff

You could visualise how the graph has been traversed below, with the left-most column first, then the middle, then the right-most column.

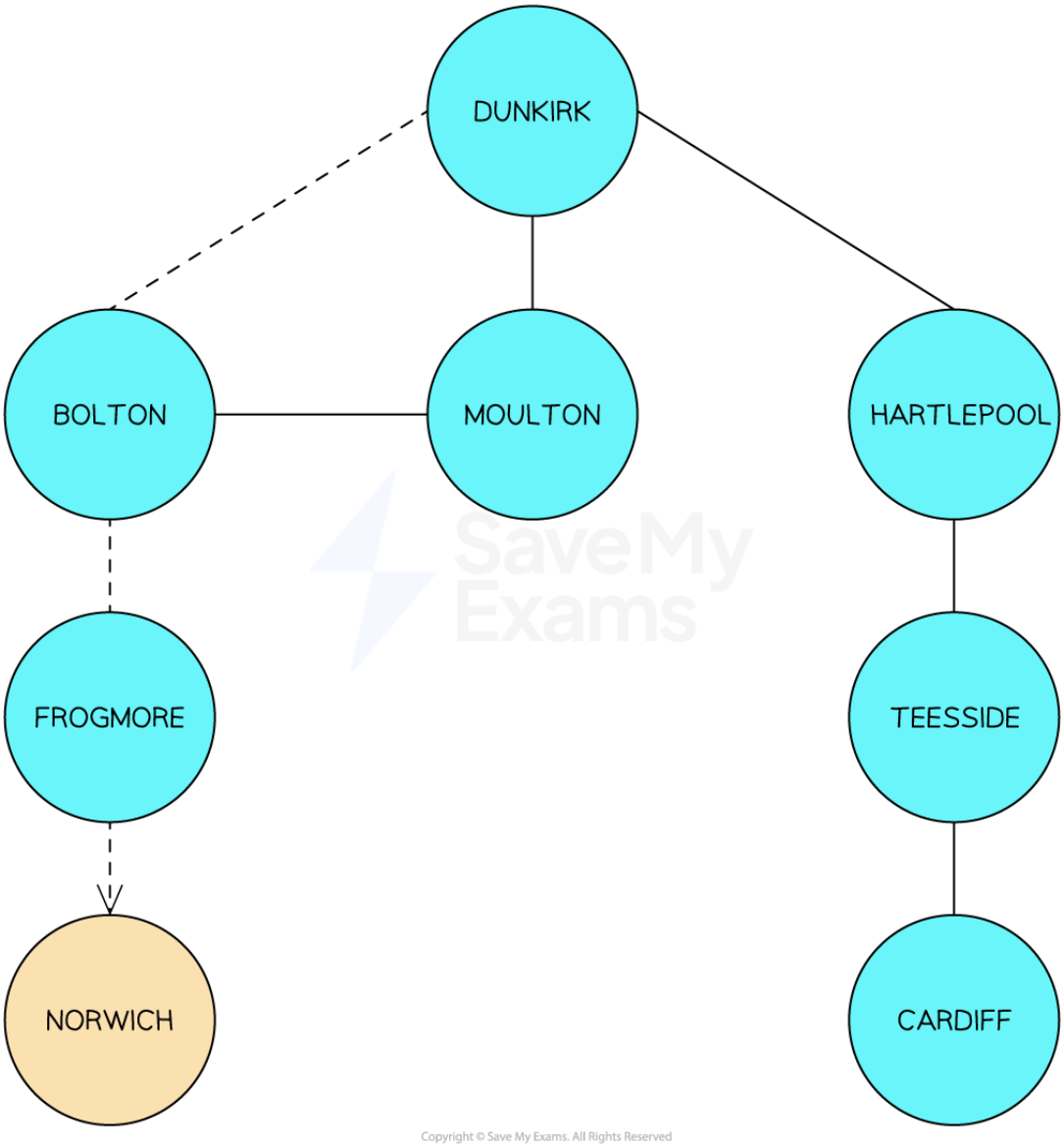
Your notes



Adding & Removing Data in Graphs

- There is no single algorithm for adding or deleting a node in a graph. This is because a graph can have a number of nodes connected to any number of other nodes.
- As a result, it is more important that there are a clear set of instructions to easily traverse the graph to find the specific node.

Your notes



- A Binary Tree is a special type of graph and it does have an algorithm for adding and deleting items. This is covered in the content titles' Binary Search Trees'.

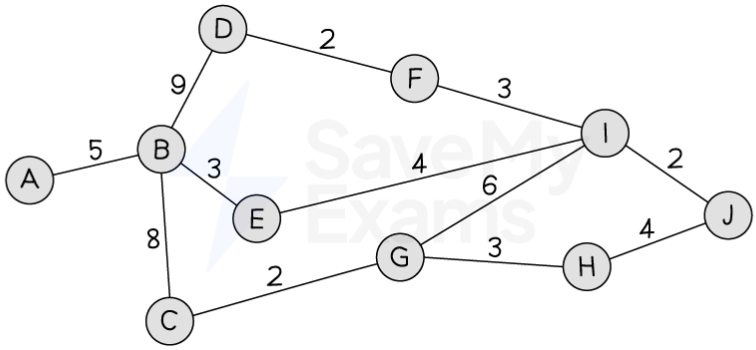
Examiner Tips and Tricks

It is important to note that it is not important what the graph looks like, but which vertices are connected.

Your notes

Worked Example

A puzzle has multiple ways of reaching the end solution. The graph below shows a graph that represents all possible routes to the solution. The starting point of the game is represented by A, the solution is represented by J. The other points in the graph are possible intermediary stages.



The graph is a visualisation of the problem.

i. Identify one difference between a graph and a tree [1]

- A graph has cycles **[1]**
- A graph can be directed/undirected [1]
- A tree has a hierarchy (e.g. Parent/Child) **[1]**

ii. Explain how the graph is an abstraction of the problem [2]

- The puzzle is not shown in the diagram **[1]**
- The graph shows different sequences of sub-problems in the puzzle that can be solved to get the final solution **[1]**
- The puzzle does not have all states visible at once **[1]**

iii. Identify two advantages of using a visualisation such as the one shown above [2]

- Visualisation benefits humans rather than computers **[1]**
- Visualisations present the information in a simpler form to understand **[1]**
- Visualisation can best explain complex situations **[1]**

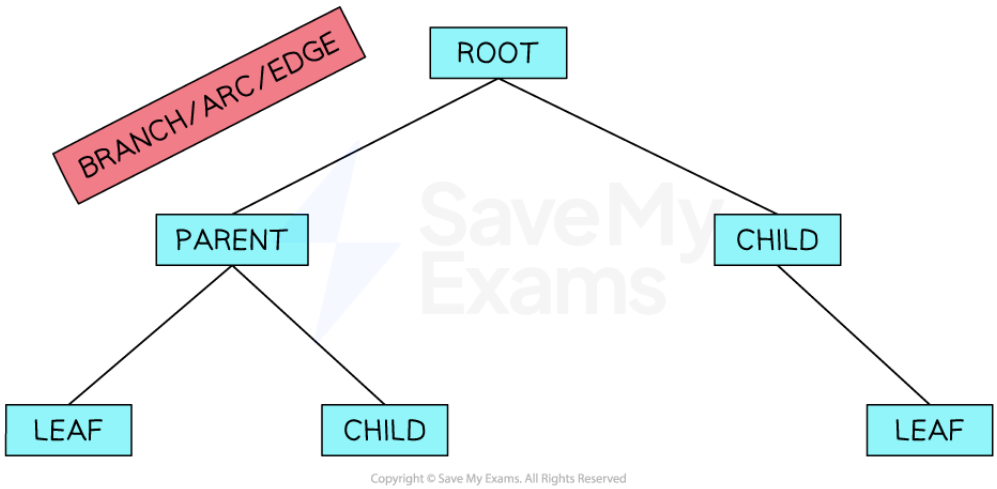
Trees

Your notes

Trees Data Structures

What is a Tree?

- A tree is a connected, undirected form of a graph with **nodes** and **pointers**
- Trees have a root node which is the top node; we visualise a tree with the roots at the top and the leaves at the bottom
- Nodes are connected to other nodes using pointers/edges/branches, with the lower-level nodes being the children of the higher-level nodes
- The endpoint of a tree is called a leaf
- The height of a tree is equal to the number of edges that connect the root node to the leaf node that is furthest away from it



- Nodes are connected by parent-child relationships
- If a path is marked from the root towards a node, a parent node is the first one and the child node is the next
- A node can have multiple children
- A leaf node is a node with no children

What are Trees used for?

Trees can be used for a range of applications:

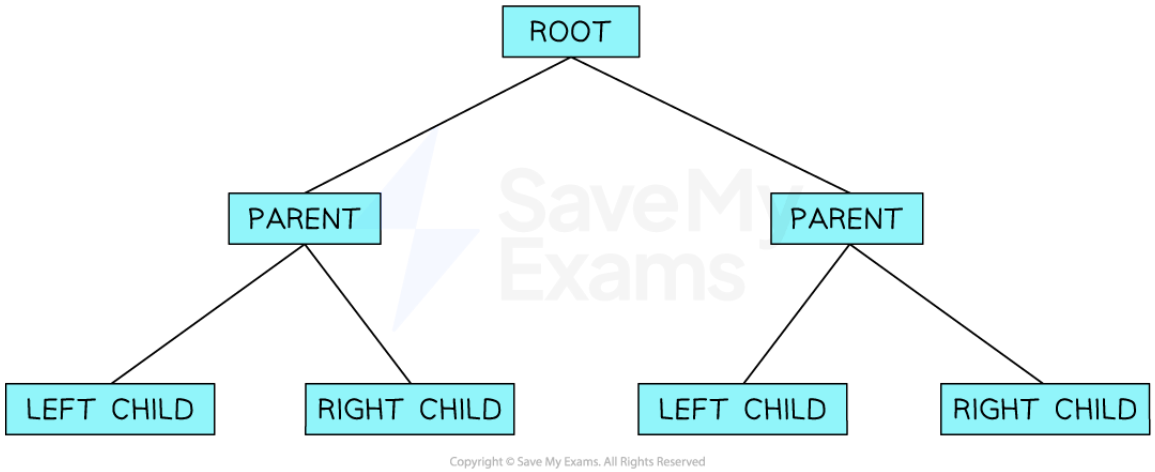
- Managing folder structures
- Binary Trees are used in routes to store routing tables
- Binary Search Trees can be built to speed up searching
- Expression trees can be used to represent algebraic and Boolean expressions that simplify the processing of the expression

Your notes

Traversing Tree Data Structures

What is a Binary Tree?

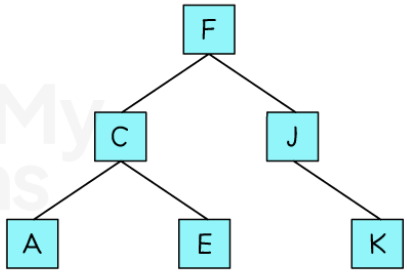
- A binary tree is a rooted tree where every node has a maximum of 2 nodes
- A binary tree is essentially a **graph** and therefore can be implemented in the same way
- For your exam, you must understand how to traverse a tree data structure, add new data to a tree and remove data from a tree.



- The most common way to represent a binary tree is by storing each node with a left and right pointer. This information is usually implemented using 2D arrays

Your notes

Index	Left Pointer	Data Value	Right Pointer
0	1	F	3
1	2	C	4
2	–	A	–
3	–	J	5
4	–	E	–
5	–	K	–



Traversing of a Binary Tree

- There are 2 methods of traversing a binary; depth-first and breadth-first.
- Both are important to understand and you should be able to output the order of the nodes using both methods

Depth-First Traversal of a Binary Tree

- There are 3 methods to traverse a binary tree using a depth-first traversal: Pre-Order, In-Order and Post-Order. For the OCR specification, you are only required to understand post-order traversal

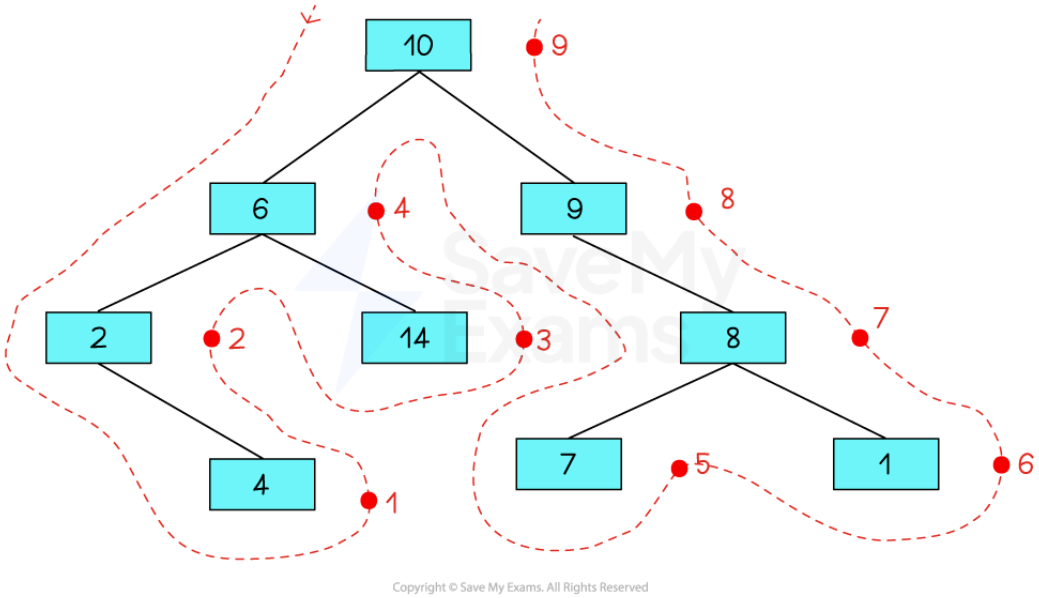
Post-Order Traversal

1. Left Subtree
2. Right Subtree
3. Root Node

- Using the outline method, imagine there is a dot on the right-hand side of each node
- Nodes are traversed in the order in which you pass them on the right
  - Start at the bottom left of the binary tree
  - Work your way up the left half of the tree
  - Visit the bottom of the right half of the tree

Your notes

- Making your way back up toward the root node at the top



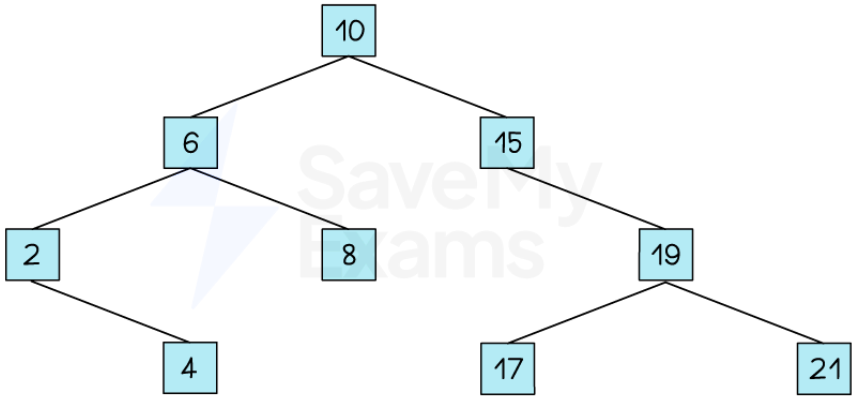
- The order of traversal is: **4, 2, 14, 6, 7, 1, 8, 9, 10**

Breadth-First Traversal of a Tree

How Do I Traverse a Tree Using a Breadth First Traversal?

- The breadth-first traversal of a tree simply means to:
  - Begin with the root node
  - Move to the left-most node under the root
  - Output each node, moving from left to right, just as though you were reading a book
  - Continue through each level of the tree

Your notes



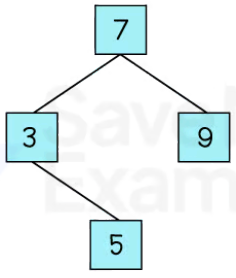
- Using the image above, a breadth-first traversal would output:
  - 10, 6, 15, 2, 8, 19, 4, 17, 21

Adding Data to a Tree

How Do You Add Data to a Binary Tree?

- As mentioned above, a tree is a fundamental data structure in Computer Science and students must be able to understand how data can be added and removed in trees
- To add a value to a binary tree you need to complete the following:

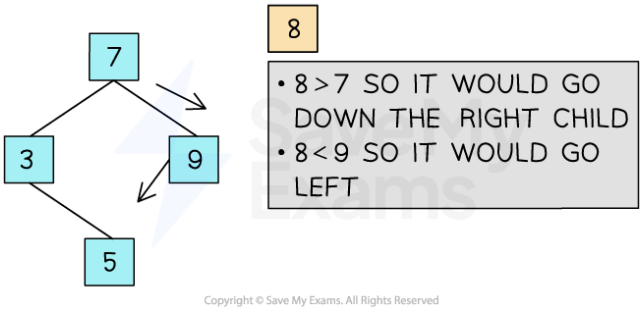
1. Start with an empty tree or existing tree



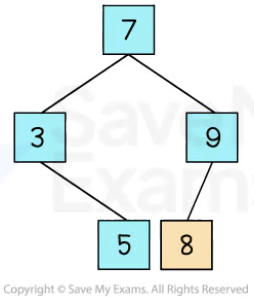
2. Identify the position where the new value should be inserted according to the rules of a binary tree
   - If the tree is empty, the new value will become the root node

- If the value is less than the current node's value, move to the left child
- If the value is greater than the current node's value, move to the right child
- Repeat this process until you reach a vacant spot where the new value can be inserted

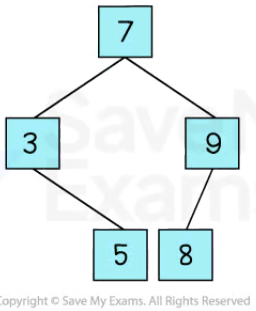
Your notes



3. Insert the new value into the identified vacant spot, creating a new node at that position



4. After insertion, verify that the binary tree maintains its structure and properties

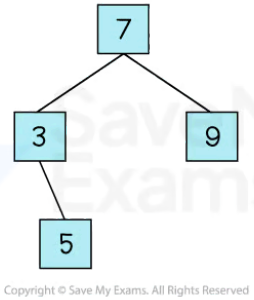


Removing Data From a Tree

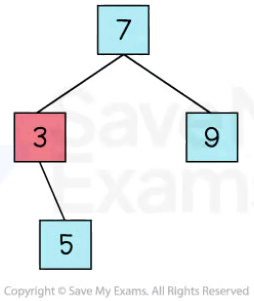
How Do You Remove Data from a Binary Tree?

- To remove a value from a binary tree you need to complete the following:

1. Start with an existing tree



2. Search for the node containing the value you want to remove



3. If the node is found:
   a. If the node has no children (leaf node), simply remove it from the tree
   b. If the node has one child, replace the node with its child
   c. If the node has two children, find the replacement node by:

   - Option 1: Find the minimum value in its right subtree (or the maximum value in its left subtree)
   - Option 2: Choose either the leftmost node in the right subtree or the rightmost node in the left subtree. Remove the replacement node from its original location and place it in the position of the node to be deleted.

Your notes

Your notes

4. After removal, adjust the binary tree structure if necessary to maintain its properties and integrity

Tree Keywords

Keyword	Definition
Node	An item in a tree
Edge	Connects two nodes together and is also known as a branch or pointer
Root	A single node which does not have any incoming nodes
Child	A node with incoming edges
Parent	A node with outgoing edges
Subtree	A subsection of a tree consisting of a parent and all the children of a parent
Leaf	A node with no children
Traversing	The process of visiting each node in a tree data structure, exactly once

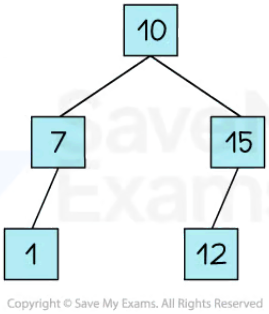
Your notes

Binary Search Trees

What is a Binary Search Tree?

- A binary search tree is a rooted tree where the nodes of the tree are ordered
- It is ordered to optimise searching
- If the order is ascending, the nodes on the left of the **subtree** have values that are lower than the root node, the nodes to the right have a higher value than the root node

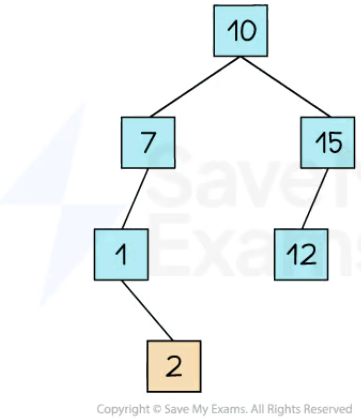
How do you Insert a value in a Binary Search Tree?



- To insert the value '2' into the binary tree above we need to compare the item to the nodes in the tree

1. Compare 2 to 10; 2 is smaller than 10, so you go down to the left child
2. Compare 2 to 7; 2 is smaller than 7 so you go down to the left child
3. Compare 2 to 1; 2 is higher than 1 so you go down to the right and add as a child node of 1

Your notes



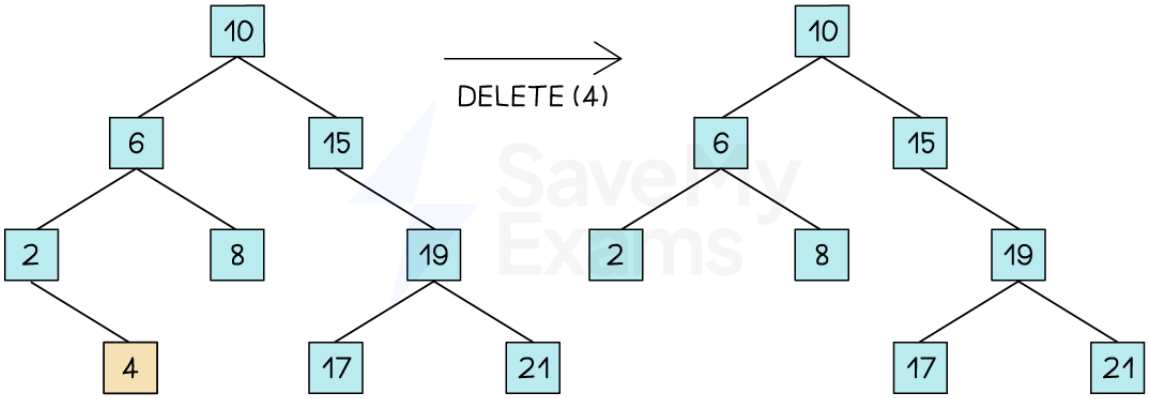
Removing Data from a Binary Search Tree

How do you delete data from a Binary Search Tree?

To delete a node from the Binary Search Tree then there are 3 possibilities:

1. The Node is a Leaf Node

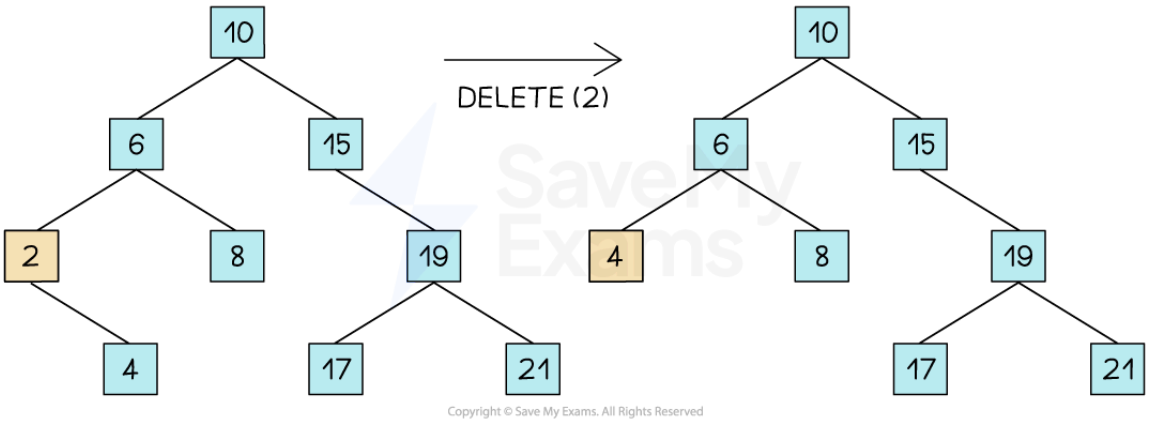
- If the node to be deleted is a leaf node, then we can directly delete this node as it has no child nodes. Node 4 does not have any child nodes so it can be deleted straight away



2. **The Node has only 1 child**

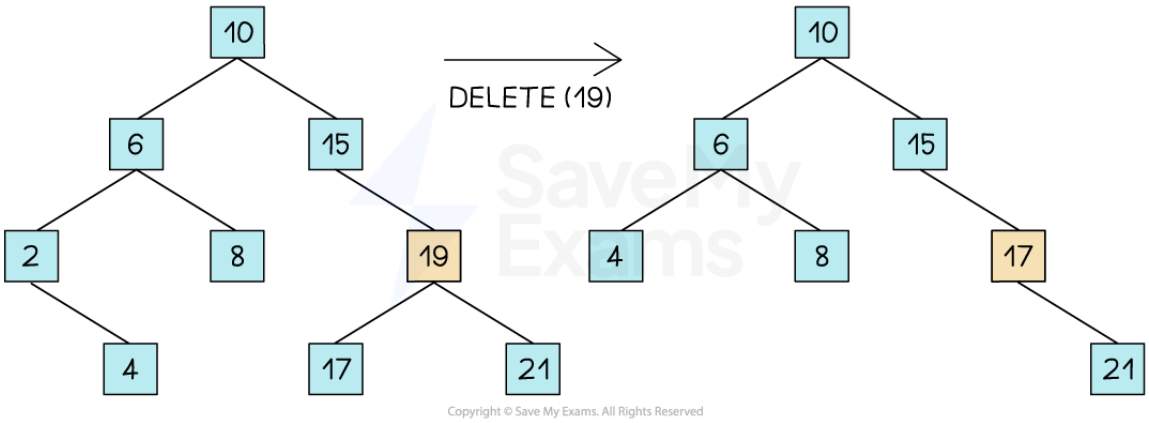
- If the node to be deleted has 1 child, then we copy the value of the child in the node and then delete the node. In the diagram, '2' has been replaced by '4', which was its child node originally

Your notes



3. **The Node has 2 children**

- If the node to be deleted has 2 children, then we replace the node with the **in-order successor** of the node or simply the minimum node in the right subtree if the right subtree of the node is not empty. We then replace the node with this minimum node and delete the node



Examiner Tips and Tricks

Remember that Binary Search Trees can have a maximum of 2 children for each parent node, but it is not a requirement that they have 2.

Hash Tables

What is a Hash Table?

- A hash table is an associative array which is coupled with a hash function. The function takes in data and releases an output. The role of the hash function is to map the key to an index in the hash table.
- Each piece of data is mapped to a unique value using the hash function
- It is sometimes possible for two inputs to result in the same hashed value. This is known as a collision
- A good hashing algorithm should have a low probability of collisions occurring but if it does occur, the item is typically placed in the next available location. This is called collision resolution
- Hash tables are normally used for indexing as they provide fast access to data due to keys having a unique 1–1 relationship with the address at which they are stored.
- Hash tables enable very efficient searching. In the best case, data can be retrieved from a hash table in constant time

A Character Hash Table

- Not all key values are simple integers so if it's a character then the hash function will need to do more processing.
- An alphanumeric key that is made up of any four letters or numbers. An example would be C6IA
- To hash the code, the characters can be converted to their ASCII value codes:

Character	ASCII code
C	67
6	54
I	73
A	65

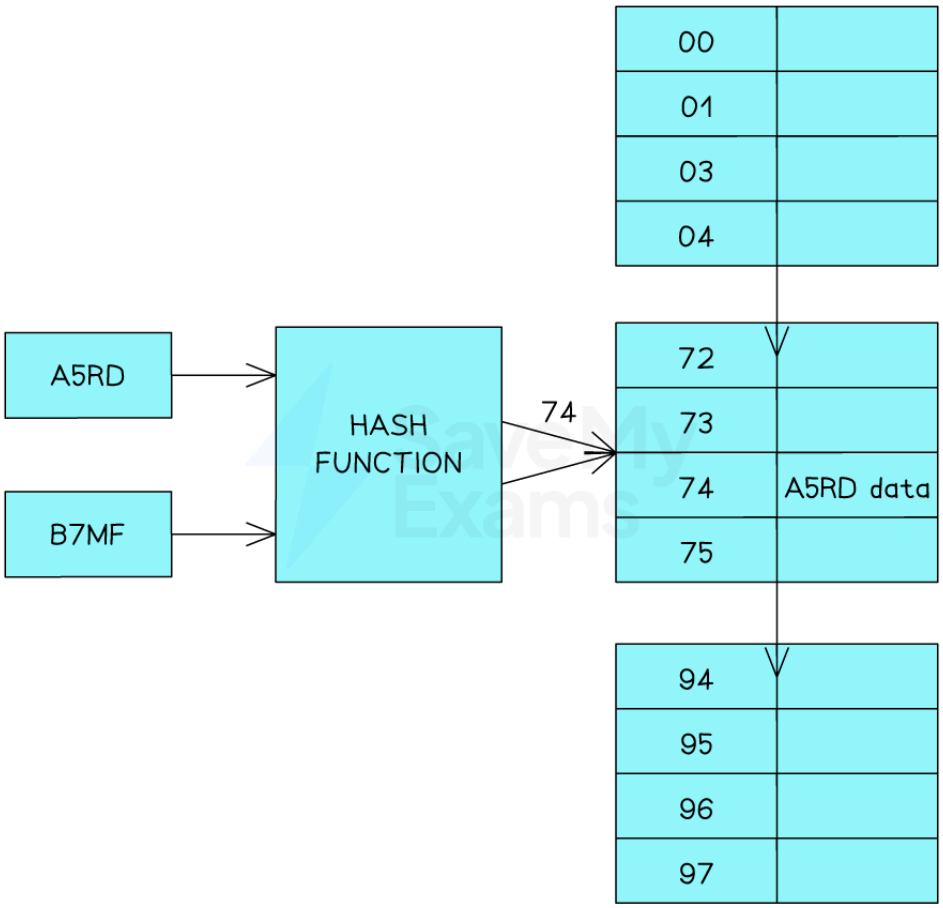
- Then add the values to get a total: 67 + 54 + 73 + 65 = 259

- Then we apply the **MOD** function: 259 MOD 97 = 65
- This then means the data associated with C6IA will be stored in position 65 in the hash table

Your notes

Collisions in a Hash Table

- A collision occurs when the hash function produces the same hash value for two or more keys
- B7MF produces the same hash value (74) as A5RD

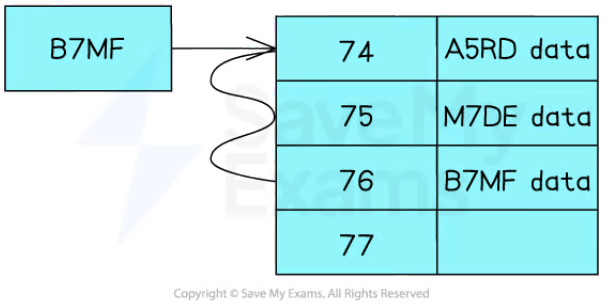


- Collisions are a problem when using hash tables because you can't store two sets of data at the same location.
- There are two ways to handle collisions Linear probing or chaining

Linear Probing

Your notes

- When there is a collision the data is placed in the next free position in the hash table
- Either it can probe sequentially or use an interval (e.g check every 3rd slot) until an empty bucket is found



- In the diagram the data with the key A5RD has already been inserted into the location 74 of the table. When you want to insert B7MF the same hash value is produced
- So because the space is already occupied by A5RD it checks the next location which is 75, as this is occupied it checks the next location which is 76 which is empty so the data will be placed in this location
- If there are lots of collisions the has table can end up very messy

Retrieving Data:

- When there has been a collision the data will not be in the correct location. The process of retrieving data from the table where collisions have been dealt with by linear probing is as follows:
  - Hash the key to generate the index value
  - Examine the indexed position to see if the key (which is stored together with the data) matches the key of the data you are looking for
  - If there is no match then each location that follows is checked (at the appropriate interval) until the matching record is found
  - If you reach an empty bucket without finding a matching key then the data is not in the hash table
- This significantly reduces the efficiency of using a hash table for searching.
- Hash tables should always be designed to have extra capacity

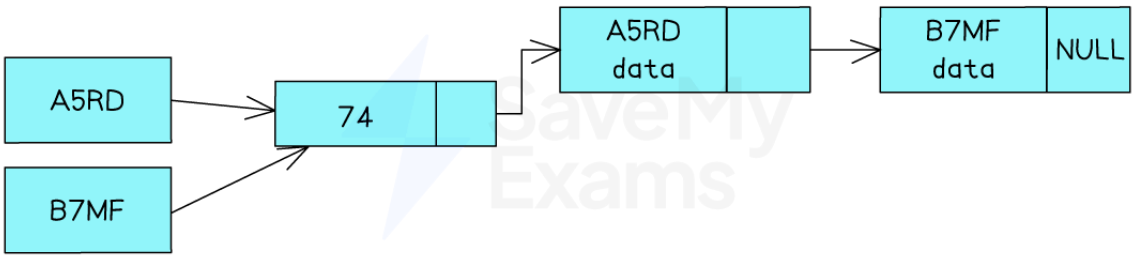
Chaining

- Another way of dealing with collisions is to use linked lists. Instead of storing the actual data in the hash table, you store a pointer to a location where the data is stored. Each data item is stored as a node with

3 attributes:

- The key value
- The data
- A pointer to the next node

- The first value to be stored will have a Null pointer as it is the only item in the list
- When there is a collision the pointer for the first node location will be updated to point to the new node
- This will create a chain of nodes whose keys resolve to the same hash value



Retrieving data

- When collisions have been dealt with using chaining the data will always be accessed through a single location in the hash table. The process for retrieving it is as follows:
  - Hash the key to generate the index value
  - Examine the key of the node to see if it matches the key that you are looking for
  - If it is not the node you are looking for follow the linked list until you find the node
  - If you reach the end of the list without finding a matching key (or the head pointer was null), the data is not in the hash table
- In the worse case (where every key hashes to the same value), you could still end up carrying out a linear search
- It is important that the hash function is designed and tested to minimise the number of possible collisions

Rehashing

- Items will be added and deleted from the hash table. If the table starts to fill up, or a large number of items are in the wrong place, the performance of the hash table will degrade. Rehashing is used to address this problem

Your notes

- Rehashing created a new table (larger if needed). The key for each item in the existing table will be rehashed and the item will be inserted into the new table
- If the new table is larger the hashing algorithm will need to be modified as it will need to generate a larger range of hash values
- Rehashing is also an opportunity to review the overall effectiveness of the hash function and to review how the collisions are handled


Your notes

Hash Tables: Adding, Removing & Traversing Data

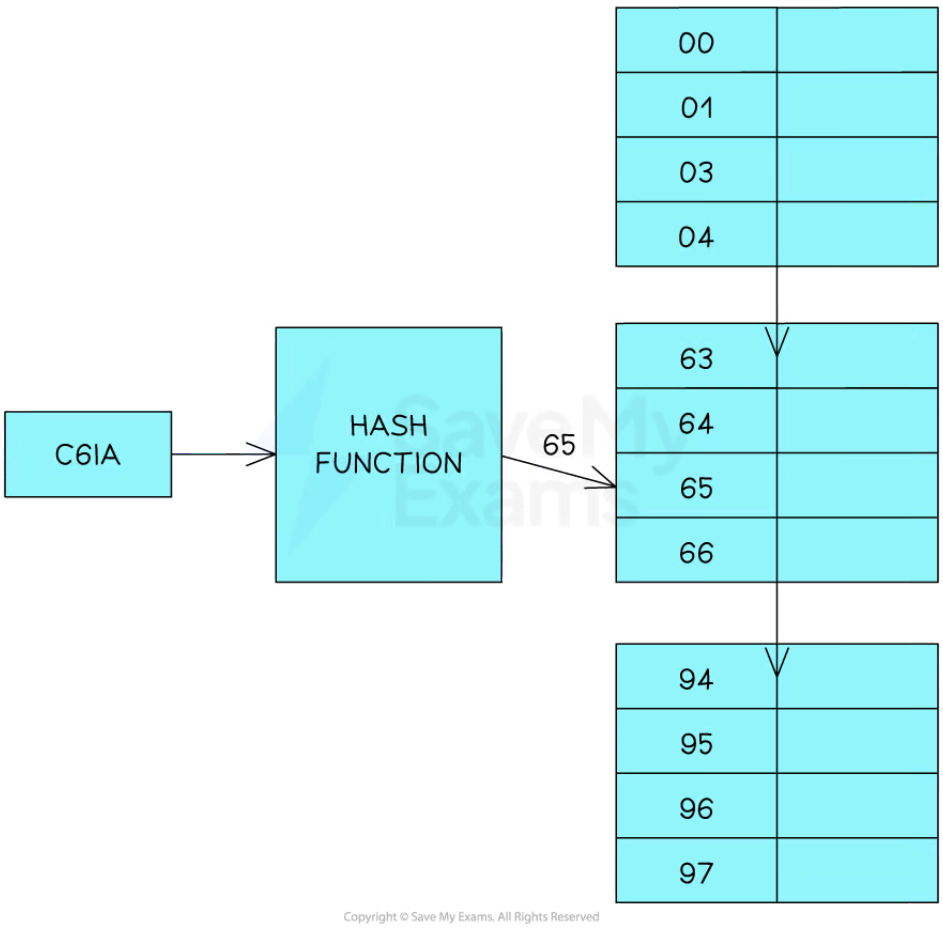
How do you add data to a Hash Table?

- To insert data into a hash table you use the hash function to generate the index of the position in the array that will be used to store the data
- The key must be stored as part of or alongside the data
- C6IA will be stored at position 65

Hash Key	Hashing Function	Hash Value
C6IA	Applying hashing function	Hash value = 65

- To be sure this is possible, position 65 will be checked to see if it is empty. If it is, the new data item is inserted into this position
- If position 65 already contains data the overflow table will be used
  - This would take place by checking the first position in the overflow table to see if it is empty.
  - If it is not empty, it will iterate through the table until an empty position is found.

Your notes



Algorithm for Adding to a Hash Table

Below is the pseudocode for the algorithm.

```
FUNCTION hash_function(key, table_size):
    // Compute the hash value for the given key
    hash_value = key MOD table_size
    RETURN hash_value

FUNCTION add_to_hash_table(hash_table, key, value, table_size):
    // Compute the hash value for the key using the hash function
    hash_value = hash_function(key, table_size)

    // Create a new entry with the key and value
    entry = (key, value)

    // Check if the hash value already exists in the hash table
```

```
    if hash_table[hash_value] is empty:
        // If the slot is empty, add the entry directly
        hash_table[hash_value] = [entry]
    else:
        // If the slot is not empty, handle collisions using a separate chaining approach
        // Append the entry to the existing list in the slot
        hash_table[hash_value].append(entry)

ENDFUNCTION
```


Your notes

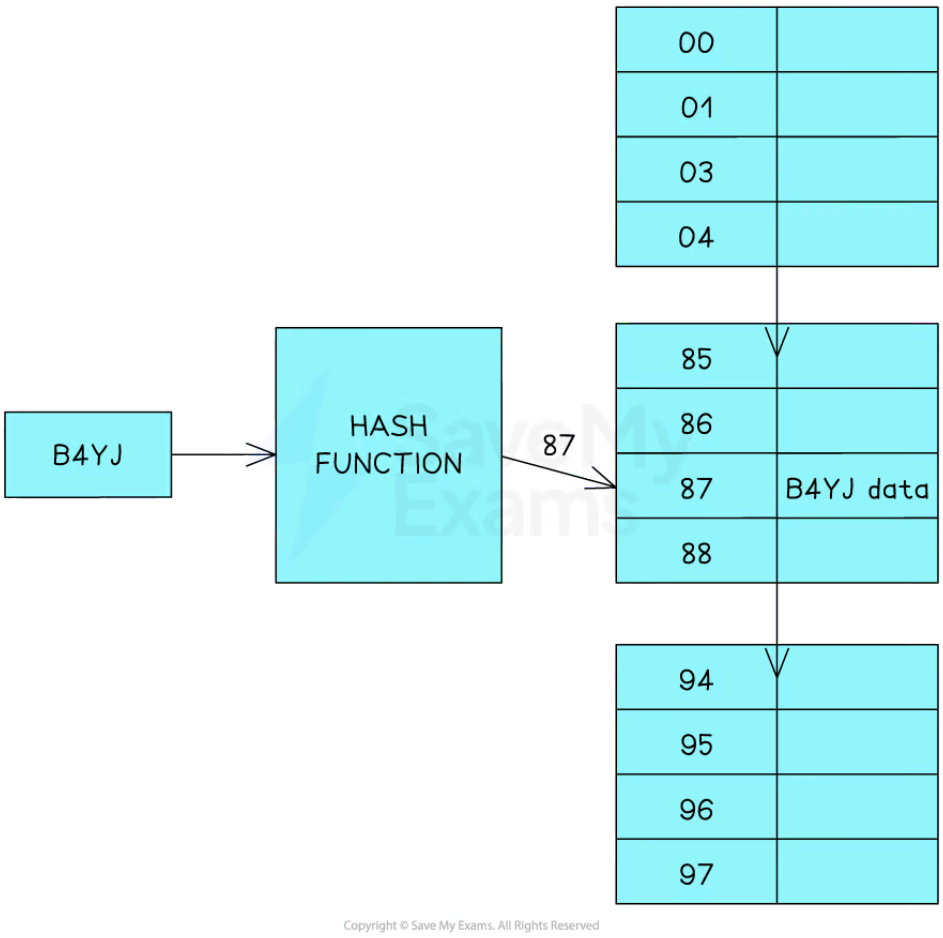
Removing Data from a Hash Table

How do you remove data from a hash table?

- To retrieve data from a hash table, the hash function is applied to the key to generate the index for the position within the array
- The key to B4YJ gives the hash value of 87
- The data can be retrieved in a single operation. You don't need to use a binary or linear search, you can go to the correct position because an array allows direct access by index
- If the data item is found it is removed (the memory address is marked as available)
- If the data item is not found, then the overflow table is to be searched using the same linear approach mentioned above

Hash Key	Hashing Function	Hash Value
B4YJ	Applying hashing function	Hash value = 87

Your notes




```
FUNCTION removeFromHashTable(hashTable, key):
    // Calculate the hash value for the given key
    hashValue = hashFunction(key)

    // Get the list of entries at the computed hash value
    entryList = hashTable[hashValue]

    // Find the entry with the matching key and remove it
    FOR entry IN entryList:
        IF entry.key == key:
            entryList.remove(entry)
            RETURN

    // Key not found
    PRINT "Key not found in the hash table"

ENDFUNCTION
```

Traversing a Hash Table

How do you traverse a Hash Table?

- Traversing, Adding and Removing data from a hash table are all done in a similar fashion.
- A hash value is generated in the same manner as before.
  - If the item being searched for is found, it is output
  - If the item being searched for is not found then the overflow table must be checked in a linear manner.
- The main advantage of a hash table is that long searches are not necessary - additional searches are only required if there are any collisions


Your notes