OCR Computer Science AS Level

1.4.2 Data Structures

Advanced Notes

Specification

1.4.2 a)

- Arrays
- Records
- Lists
- Tuples

1.4.2 b)

- Stack
- Queue

Arrays, Records, Lists, and Tuples

Arrays

An array is an ordered, finite set of elements of a single type. A 1D (one-dimensional) array is a linear array. Unless stated in the question, arrays are always taken to be zero-indexed. This means that the first element in the array is considered to be at position zero. Below is an example of a one-dimensional array:

```
oneDimensionalArray = [1, 23, 12, 14, 16, 29, 12]      //creates a
                                                         1D array
print(oneDimensionalArray[3])
     >> 14
```

A two-dimensional array can be visualised as a table or spreadsheet. When searching through a 2D array, you first go down the rows and then across the columns to find a given position. This is the reverse to the method used to find a set of coordinates. Below is an example involving a two-dimensional array.

```
twoDimensionalArray = [[123, 28, 90, 38, 88, 23, 47],[1, 23, 12,
14, 16, 29, 12]]

print(twoDimensionalArray)
     >> [[23,  28,  90,  38,  88,  23,  47],
         [ 1,  23,  12,  14,  16,  29,  12]]

print(twoDimensionalArray[1,3])       // Goes down and then across
     >> 14
```

A three-dimensional array can be visualised as a multi-page spreadsheet and can be thought of as multiple 2D arrays. Selecting an element in a 3D array requires the following syntax to be used: `threeDimensionalArray[z,y,x]`, where z is the array number, y is the row number and x is the column number.

```
threeDimensionalArray = [[[12,8],[9,6,19]],[[241,89,4,1],[19,2]]]
print(threeDimensionalArray[0,1,2])
     >> 19
```

Records

A record is more commonly referred to as a row in a file and is made up of fields. Records are used in databases, as shown in the table below:

ID	FirstName	Surname
001	Antony	Joshua
002	Tyson	Fury
003	Deonte	Wilder

Above is a file containing three records, where each record has three fields. A record is declared in the following manner:

```
fighterDataType = record
     integer   ID
     string    FirstName
     string    Surname
end record
```

Each field in the record can be identified by `recordName.fieldName`. First, however, the record must be created. When creating a record, a variable must first be declared:
`fighter : fighterDataType`
Then its attributes can be accessed, using the following syntax:
`fighter.FirstName`

Lists

A list is a data structure consisting of a number of ordered 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, as data in arrays is stored. Lists can also contain elements of more than one data type, unlike arrays.

There are a range of operations that can be performed involving lists, described in the table below. The following structure is used when manipulating lists:

```
List = [23, 36, 62, 49 , 23, 29, 12]
List.function(Parameters)
```

List Operations	Example	Description

isEmpty()	`List.isEmpty()`
	`>> False`
		Checks if the list is empty

append(value)	`List.append(15)`
	`>>`
		Adds a new value to the end of the list

remove(value)	`List.remove(23)`
	`>>`
		Removes the value the first time it appears in the list

search(value)	`List.search(38)`
	`>> False`
		Searches for a value in the list.

length()	`List.length()`
	`>> 7`
		Returns the length of the list

index(value)	`List.index(23)`
	`>> 0`
		Returns the position of the item

insert(position, value)	`List.insert(4,25)`
	`>>`
		Inserts a value at a given position

pop()	`List.pop()`
	`>>12`
		Returns and removes the last value in the list

pop(position)	`list.pop(3)`	Returns and removes the value in the list at the given position

Tuples

An ordered set of values of any type is called a tuple. A tuple is immutable, which means it cannot be changed: elements cannot be added or removed once it has been created. Tuples are initialised using regular brackets instead of square brackets.

```
tupleExample = (“Value1”, 2, “Value3”)
```

Elements in a tuple are accessed in a similar way to elements in an array, with the exception that values in a tuple cannot be changed or removed. Attempting to do so will result in a syntax error.

```
print(tupleExample[0])
     >> Value1 :
tupleExample[0] = “ChangedValue”
     >> Syntax Error
```

Stacks and Queues

Stacks

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 data structures in computer science; they are used to reverse an action, such as to go back a page in web browsers. The 'undo' buttons that applications widely make use of also utilise stacks. A stack can be implemented as either a static structure or a dynamic structure. 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. Stacks are implemented using a pointer which points to the top of the stack, where the next piece of data will be inserted.

There are numerous operations that can be performed on a stack and that you need to be aware of. The following syntax must be used when calling a function on a stack:_

Stack Operations	Example	Description

isEmpty()	`Stack.isEmpty()`
	`>> True`
		Checks if the stack is empty. Works by checking the value of the top pointer.

push(value)	`Stack.append("Nadia")`
	`>>`
	`Stack.append("Elijah")`
	`>>`
		Adds a new value to the end of the list. Needs to check that the stack is not full before pushing to the stack.

peek()	`Stack.peek()`
	`>> "Elijah"`
		Returns the top value from the stack. First checks the stack is not empty by looking at value of top pointer.

pop()	`Stack.pop()`
	`>> "Elijah"`
		Removes and returns the top value of the stack. First checks the stack is not empty by looking at value of top pointer.

size()	`Stack.size()`
	`>> 2`
		Returns the size of the stack

isFull()	`Stack.isFull()`
	`>> False`
		Checks if the stack if full and returns a Boolean value. Works by comparing stack size to the top pointer.

Queues

A queue is a first in first out (FIFO) data structure; items are added to the end of the queue and are removed from the front of the queue. Queues are commonly used in printers, keyboards and simulators. There are a few different ways in which a queue can be implemented, but they all follow the same basic principles.

A linear queue is a data structure consisting of an array. Items are added into the next available space in the queue, starting from the front. Items are removed from the front of the queue. Queues make use of two pointers: one pointing to the front of the queue and one pointing to the back of the queue, where the next item can be added.

The highlighted box shows the front of the queue.

```
enQueue(Task3)  // enQueue(item) is how items are added to a queue
```

Position	0	1	2	3	4	5
Data	Task1	Task2	Task3			

```
deQueue() // deQueue(item) is how items are removed from a queue
```

Position	0	1	2	3	4	5
Data		Task2	Task3			

```
enQueue(Task4)
```

Position	0	1	2	3	4	5
Data		Task2	Task3	Task4		

```
deQueue()
```

Position	0	1	2	3	4	5
Data			Task3	Task4		

As the queue removes items, there are addresses in the array which cannot be used again, making a linear queue an ineffective implementation of a queue.
Circular queues try to solve this. A circular queue operates in a similar way to a linear queue in that it is a FIFO structure. However, it is coded in a way that once the queue's rear pointer is equal to the maximum size of the queue, it can loop back to the front of the array and store values here, provided that it is empty. Therefore, circular queues can use space in an array more effectively, although they are harder to implement.

Below is an example illustrating how the rear pointer in a circular queue works:

`enQueue(Task5)`

Position	0	1	2	3	4	5
Data			Task3	Task4	Task5	

`rearPointer : 4`
`maxSize : 5`

`enQueue(Task6)`

Position	0	1	2	3	4	5
Data			Task3	Task4	Task5	Task6

`rearPointer : 5`
`maxSize : 5`

`enQueue(Task7)`

Position	0	1	2	3	4	5
Data	Task7		Task3	Task4	Task5	Task6

`rearPointer : 0`
`maxSize : 5`

Queue Operations	Example	Description

enQueue(value)	`Queue.enQueue("Nadia")`
	 `>>`
	 `Queue.enQueue("Elijah")`
	 `>>`
		Adds a new item to the end of the queue. Increments the back pointer.

deQueue()	`Queue.deQueue()`
	`>>`
		Removes the item from the front of the queue.
		Increments the front pointer.

isEmpty()	`Queue.isEmpty()`
	`>> False`
		Checks if the queue if empty by comparing the front and back pointer.

isFull()	`Queue.isFull()`
	`>> False`
		Checks if the queue is full by comparing the back pointer and queue size.