OCR Computer Science AS Level

1.4.2 Data Structures

Intermediate 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 linear. Arrays are always taken as zero-indexed, unless stated otherwise. Elements are selected using the syntax: `oneDimensionalArray[x]`, where x is the position of the element.

A two-dimensional array can be visualised as a table or spreadsheet. When finding a given position in a 2D array, you first go down the rows and then across the columns. Selecting elements requires the following syntax to be used: `twoDimensionalArray[z,y,x]`

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.

Records

A record is a row in a file and is made up of fields. Records are used in databases. Each field in a record can be identified using the syntax: `recordName.fieldName`. First, however, a record must be created by creating a variable. The syntax below shows the variable 'fighter' being created from a record structure called 'fighterDataType':

```
fighter : fighterDataType
```

Its attributes can then 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. Items in lists can be stored non-contiguously and can be of more than one data type, which is not possible in an array.

Manipulating lists

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)`	Searches for a value in the

	`>> False`	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. Tuples are immutable, which means elements cannot be added or removed once a tuple has been created. Tuples are initialised using regular brackets and elements are accessed in the same way as elements in an array.

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 used to reverse an action, such as to go back a page in web browsers and in 'undo' buttons. Stacks are implemented using a pointer which points to the top of the stack, where the next piece of data will be inserted.

Manipulating a stack

Stack Operations	Example	Description
isEmpty()	`Stack.isEmpty()` `>> True`	Checks if the stack is empty.

push(value)	`Stack.append("Nadia")`
	`>>`
	`Stack.append("Elijah")`
	`>>`
		Adds a new value to the end of the list.

peek()	`Stack.peek()` `>> "Elijah"`	Returns the top value from the stack.

pop()	`Stack.pop()` `>> "Elijah"`	Removes and returns the top value of the stack.
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.

Queue

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 used in printers to store print jobs, keyboards and simulators.

In a linear queue, items are added into the next available space, starting from the front. Items are removed from the front of the queue. Queues make use of two pointers: pointing to the front and back of the queue.

Manipulating a queue

The highlighted boxes in the example below show 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			

Positions from which data has been removed cannot be used again, making a linear queue an ineffective implementation of a queue.

Circular queues are coded so that once the queue's rear pointer is equal to the maximum size of the queue, the queue can loop back to the front and store values here, provided that there is empty space. Therefore, circular queues use space more effectively, although they are harder to implement.

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

`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.

deQueue()	`Queue.deQueue()` `>>`	Removes the item from the front of the queue. .
isEmpty()	`Queue.isEmpty()` `>> False`	Checks if the queue if empty
isFull()	`Queue.isFull()` `>> False`	Checks if the queue is full