OCR Computer Science A Level

1.4.2 Data Structures

Advanced Notes

Specification

1.4.2 a)

- Arrays
- Records
- Lists
- Tuples

1.4.2 b)

- Linked List
- Graphs
- Stack
- Queue
- Tree
- Binary Search Tree
- Hash Table

1.4.2 c)

- Traversing data structures
- Adding data and removing data from data structures

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.

Manipulating lists

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
```

Linked Lists, Graphs, Stacks, Queues, and Trees

Linked Lists

A linked list is a dynamic data structure 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 called a link or pointer field.

Index	Data	Pointer
0	'Linked'	2
1	'Example'	0
2	'List'	-
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 in the list as a pointer - which in this case is 'Example' at position 1 - as well as a pointer identifying the index of the next available space, which is 3 in our example. When traversing a linked list, the algorithm begins at the index given by the 'Start' pointer and outputs the values at each node until it finds that the pointer field is empty or null. This signals that the end of the linked list has been reached.

Traversing the linked list above would produce:

```
'Example', 'Linked', 'List'
```

Manipulating a linked list

One advantage of using linked lists is that values can easily be added or removed by editing pointers. The following procedure is used to add the word 'OCR' after the word 'Example':

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

3	'OCR'	

Start = 1 NextFree=4

2. The pointer field of the word ‘Example’ is updated to point to ‘OCR’, at position 3.

1	‘Example’’	3

3. The pointer field of the word ‘OCR’ is updated to point to ‘Linked’, at position 0.

3	‘OCR’	0

4. When traversed, this linked list will now output ‘Example’, ‘OCR’, ‘Linked’, ‘List’.

Removing a node also involves updating nodes, this time to bypass the deleted node. The following procedure is used to remove the word ‘List’ from the original linked list:

1. Update the pointer field of ‘Example’ to point to ‘List’ at index 2.

0	‘Linked’	2
1	‘Example’	2
2	‘List’	-

2. When traversed, this linked list will now print ‘Example’, ‘List’.

As you can see, the node is not truly removed from the list, it is only ignored. Although this is easier, this wastes memory. Storing pointers also means more memory is required compared to an array. As items in linked lists are stored in a sequence, they can only be traversed in this order; an item cannot be directly accessed, as is possible in an array.

Random Access

The ability to access a specific element directly, given its index. This is possible in an array.

Graphs

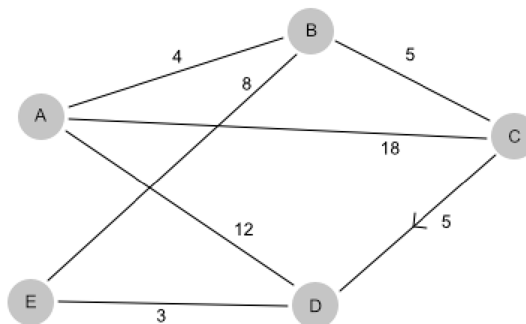
A graph is a set of vertices/nodes connected by edges/arcs. Graphs can be placed into the following categories:

- Directed Graph: The edges can only be traversed in one direction.
- Undirected Graph: The edges can be traversed in both directions.
- Weighted Graph: A cost is attached to each edge.

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

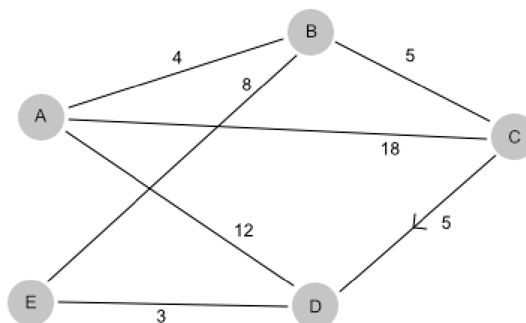
Adjacency Matrix

	A	B	C	D	E
A	-	4	18	12	-
B	4	-	5	-	8
C	18	5	-	5	-
D	12	-	-	-	3
E	-	8	-	3	-



Adjacency List

A	→	{B:4, C:18, D:12}
B	→	{A:4, C:5, E:8}
C	→	{A:18, B:5, D:5}
D	→	{A:12, E:3}
E	→	{B:8, D:3}



Adjacency Matrix Advantages	Adjacency List Advantages
Convenient to work with due to quicker access times	More space efficient for large, sparse networks
Easy to add nodes	

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.

Manipulating a stack

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:

```
nameOfStack.function(Parameters)
```

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.

Manipulating a queue

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

Operations on a queue are performed using the syntax below:
`nameOfQueue.function(Parameters)`

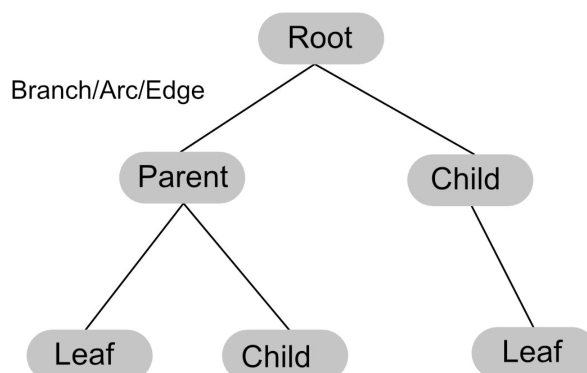
Queue Operations	Example	Description
enQueue(value)	Queue.enQueue("Nadia") >> Queue.enQueue("Elijah") >>	Adds a new item to the end of the queue. Increments the back pointer.
deQueue()	Queue.deQueue() >>	Removes the item from the front of the queue. Increments the front pointer.
isEmpty()	Queue.isEmpty() >> 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.

Trees

A tree is a connected form of a graph. Trees have a root node which is the top node in any tree. Nodes are connected to other nodes using branches, with the lower-level nodes being the children of the higher-level nodes.

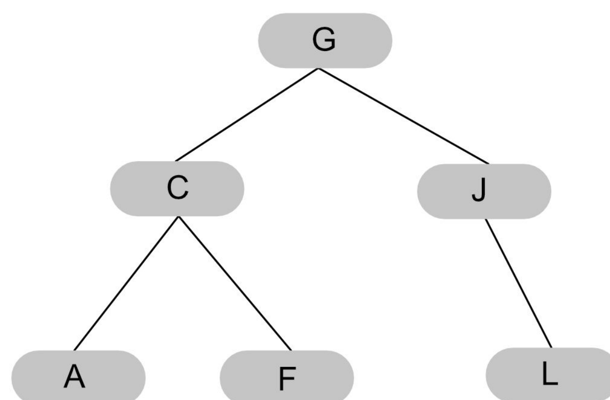
Below are some terms you should be familiar with:

Node	An item in the tree
Edge	Connects two nodes together and is also known as a branch, or arc
Root	A single node which does not have any incoming nodes
Child	A node with incoming edges
Parent	A node with outgoing edges
Subtree	Subsection of a tree consisting of a parent and all the children of a parent
Leaf	A node with no children



A binary tree is a type of tree in which each node has a maximum of two children. These are used to represent information for binary searches, as information in these trees is easy to search through. The most common way to represent a binary tree is storing each node with a left pointer and a right pointer. This information is usually implemented using two-dimensional arrays.

Index	Left Pointer	Data Value	Right Pointer
0	1	G	3
1	2	C	4
2	-	A	-
3	-	J	5
4	-	F	-
5	-	L	-

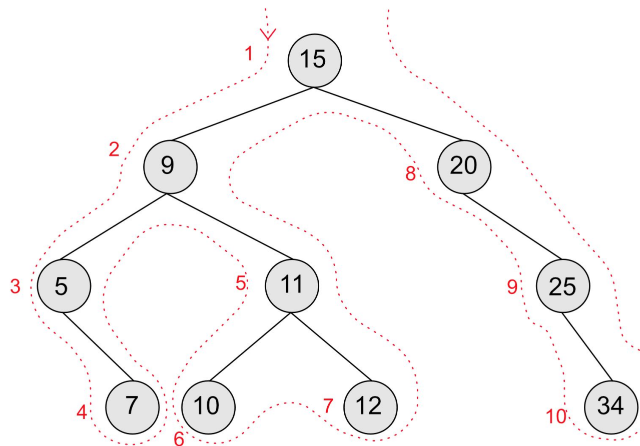


Traversing a binary tree

There are three methods of traversing a binary tree: Pre-order, In-order and Post-order.
A simple way of remembering these is using the outline method, which is described below.

Pre-order Traversal

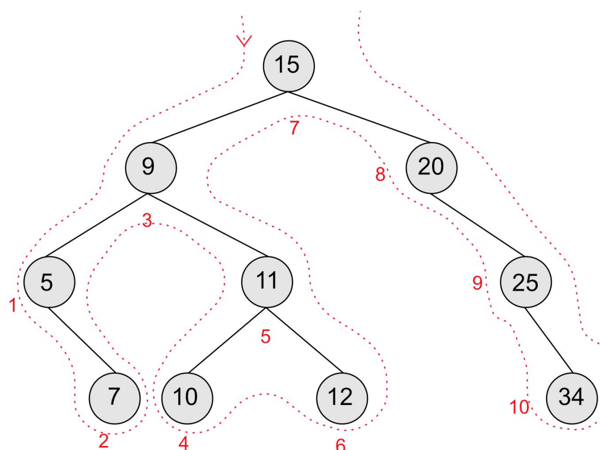
Pre-order traversal follows the order: root node, left subtree, then right subtree.
Using the outline method, nodes are traversed in the order in which you pass them on the left, beginning at the left-hand side of the root node.
Pre-order traversal is used in programming languages in which the operation is written before the values. This means a + b would be written as + a b, as shown in the diagram.



The order of traversal is: 15, 9, 5, 7, 11, 10, 12, 20, 25, 34

In-order Traversal

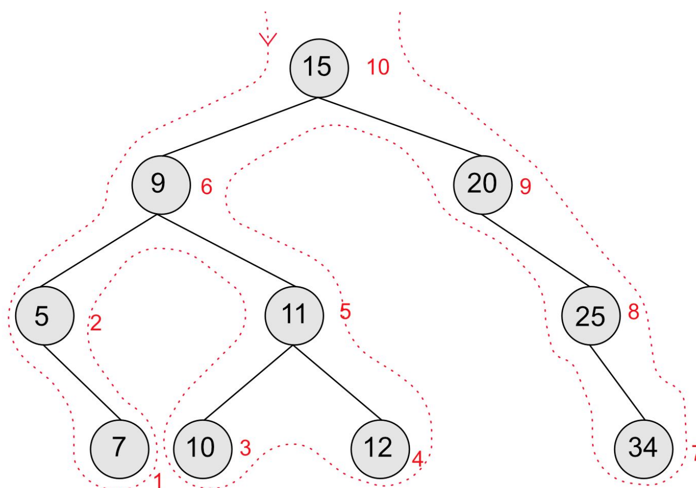
In-order traversal follows the order: left subtree, root node, right subtree.
Using the outline method, nodes are traversed in the order in which you pass under them, beginning at the first node from the left which does not have two child nodes.
This is useful for traversing the nodes in sequential order.



Order: 5, 7, 9, 10, 11, 12, 15, 20, 25, 34

Post-order Traversal

Post order traversal follows the order: left subtree, right subtree, root node.
Using the outline method, nodes are traversed in the order in which you pass them on the right.



Order: 7, 5, 10, 12, 11, 9, 34, 25, 20, 15

Hash Tables

A hash table is an array which is coupled with a hash function. The hash function takes in data (a key) and releases an output (the hash). The role of the hash function is to map the key to an index in the hash table.

> **Synoptic Link**
>
> A **hash** can also be used to **securely store data** such as pins and passwords.
>
> Uses of hashing are covered in **1.3.1** under **Different Uses of Hashing**

Each piece of data is mapped to a unique value using the hash function. However, 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 in the event that 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, one-to-one relationship with the address at which they are stored..