OCR Computer Science A Level

1.4.2 Data Structures

Intermediate 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 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)` `>> 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. 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.

Linked Lists, Graphs, Stacks, Queues, and Trees

Linked Lists

A dynamic data structure used to hold an ordered sequence. Items do not have to be in contiguous data locations. Each item is called a node, and contains a data field alongside 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 actual data value. The pointer field contains the address of the next item in the list. Linked lists also store the index of the first item along with the index of the next available space as pointers. 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 the end of the linked list. Traversing the linked list above would produce:
`'Example', 'Linked', 'List'`

Manipulating a linked list

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'.

The following procedure is used to remove the word 'Linked' 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'.

The node is not truly removed from the list, only ignored. Although this is easier, it wastes memory. Storing pointers also means more memory is required compared to an array.

Graphs

A graph is a set of vertices/nodes connected by edges/arcs. There are three types:

- 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.

Graphs can be represented using either an adjacency matrix or an adjacency list.

Advantages of using Adjacency Matrix	Advantages of using Adjacency List
Convenient to work with	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 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

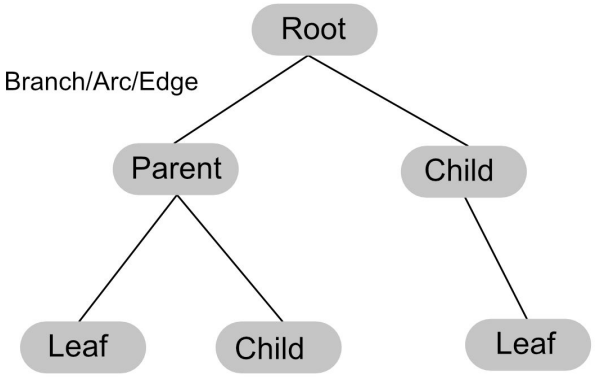
rearPointer : 0
maxSize : 5

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

Trees

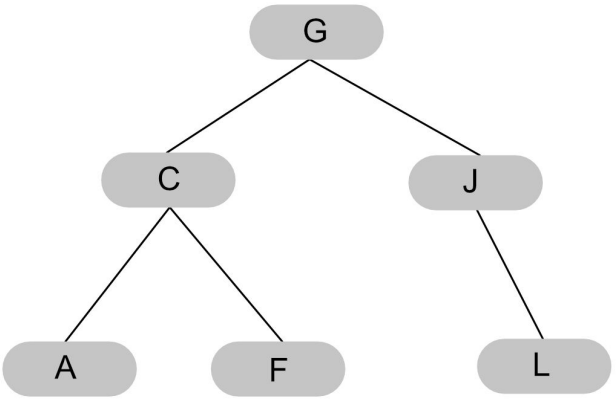
A tree is a connected form of a graph.

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 tree in which each node has a maximum of two children. These represent information in a way that is easy to search. The most common way to represent a binary tree is by storing each node with a left pointer and a right pointer.

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

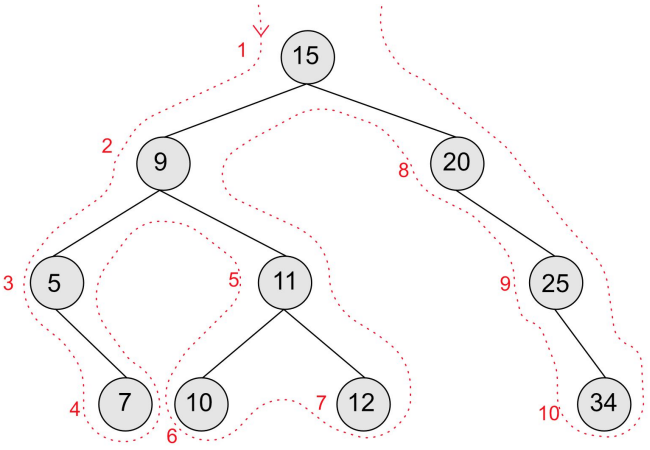


Traversing a binary tree

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

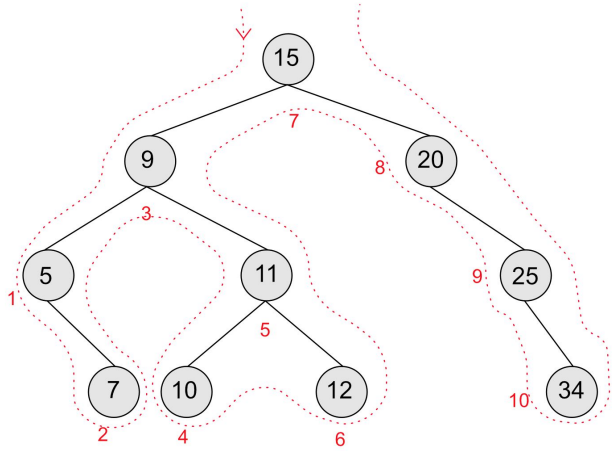
Pre-order traversal follows the order: root node, left subtree, then right subtree.
Using the outline method, nodes are traversed in the order in which you pass them on the left, beginning at the left-hand side of the root node.
Pre-order traversal is used in programming languages in which the operation 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

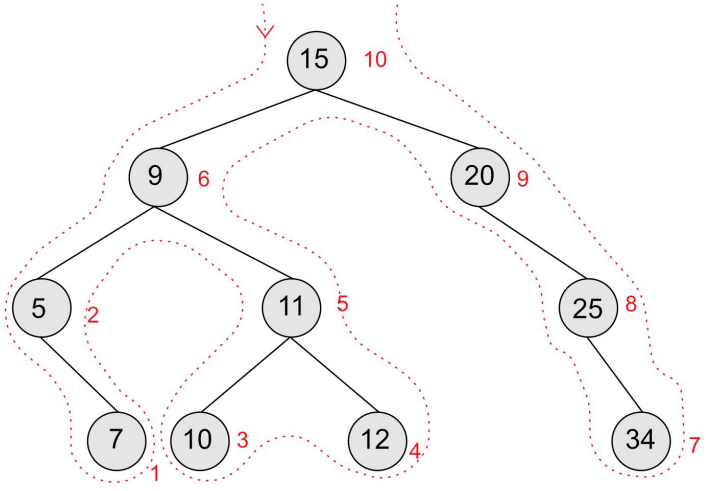
In-order traversal follows the order: left subtree, root node, right subtree.
Using the outline method, nodes are traversed in the order in which you pass under them, beginning at the first node from the left which does not have two child nodes.
This is useful for traversing the nodes in sequential order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



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) to produce a unique output (the hash). It exists to map the key to a unique index in the hash table.

Sometimes, two keys might produce the same hashed value. This is called a collision, in which case the item is typically placed in the next available location. A good hashing algorithm should have a low rate of collisions.

Hash tables are commonly 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.

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**