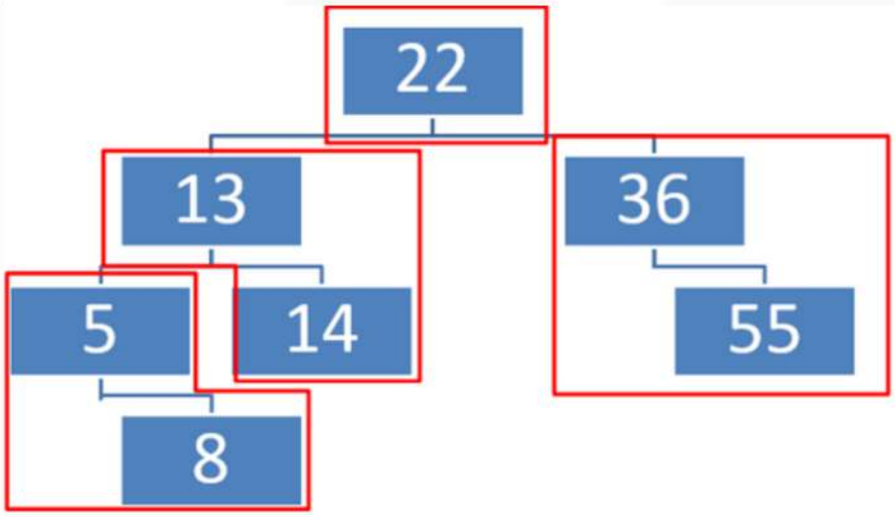
GCST

2.3.1 ALGORITHMS FOR THE MAIN DATA STRUCTURES

TOPIC WISE EXAM QUESTIONS

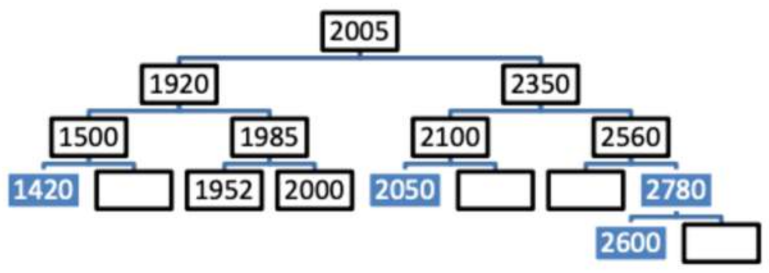
ANSWERS

A-LEVEL OCR

1	(a)	(i)	1 mark each to max 2 • It is a hierarchical structure / not directed • Data is stored in nodes • Nodes are linked by branches/edges • It has a root node • Each node has zero or more nodes 'beneath' it // nodes can link to child nodes • It has leaf nodes / nodes without any lower nodes are leaf nodes • It has no cycles/loops (distinguishing it from a graph)	2	
1	(a)	(ii)	1 mark each • Root node 22 at the start • 13 and 14 in correct order • 5 and 8 in correct order • 36 and 55 in correct order 	4	Do not allow nodes to be drawn downwards.
1	(a)	(iii)	1 mark each • **Search/traverse tree** until the required node is found • Set the parent node pointer to the leaf node to null • Add the deleted node to the free storage list // leave for garbage clear up	2	
1	(a)	(iv)	1 mark each to max 4 • Check if the root node **is equal** to search value and if so…. • …return/output/report found • If value **is less** than root node take left subtree • If value **is greater** than root node take right subtree • Repeat process with the subtree… • …until search value is found • …until no more branches can be travelled.	4	
1	(a)	(v)	1 mark each • Visiting A first… • …Then visiting F, C… • …Then visiting L, T, P… • …Visiting H last Solution: A, F,C, L,T,P, H	4	
1	(a)	(vi)	1 mark each to max 2 • When a leaf node is reached… • …the traversal backtracks to the leaf's parent node • …backtracks to last node with unvisited children	2	Candidates may use an example from the tree in 1a(v) to illustrate their response. If an answer gives implementational detail of how a stack is used, map to the bullet points given.

3	(a)		1 mark each • `headPointer`: To indicate the first element in the list • `freeListPointer`: To indicate the next index to store data in (the freeList)	2	
3	(b)		It doesn't point to another **node** Indicates the end of the linked list	1	
3	(c)		• first output red… • …remainder of list correct e.g. red blue grey green purple orange	2	
3	(d)		1 mark each to max 4 Check space available in the free list • Check to make sure `freeListPointer` is not Null Add new data item to first free space in free list • Insert new data item at index `freeListPointer` (index 4) Append e.g. • Traverse to / locate the end of the list (index 3 'orange') • Set the pointer of the last item in the linked list to `freeListPointer` (pointer at index 3 'orange' changes from Null to 4)… • ... update `freeListPointer` to the location that new data item pointer is pointing to at present. (`freeListPointer` changes from 4 to 5) • ... update pointer from new data item to Null (index 4 pointer changes from 5 to Null) Prepend e.g. • Update `freeListPointer` to point to the location that the pointer from the first item in the free list is pointing to (`freeListPointer` changes from 4 to 5) • ... Update pointer from new data item to `headPointer` (index 4 pointer changes from 5 to 1) • ... Update `headPointer` to the index of new data item (`headPointer` changes from 1 to 4)	4	Note descriptions could be for either appending an item to the end of the current list or prepending it to the start. There are different ways to achieve this. Allow answers that illustrate solutions by example from the table in Fig 3 at the start of the question. Reponses must refer to the relevant pointers or give clear exemplifications.
3	(e)		1 mark for each statement	5	Ignore case of identifiers in pseudocode Only penalise excessive spaces within identifier names if obvious.

```
function findNode(toFind, headPointer, linkedList)
  currentNode = headPointer
  while(currentNode != NULL)
    if linkedList[currentNode].data == toFind then
      return currentNode
    else
      currentNode = linkedList[currentNode].pointer
    endif
  endwhile
  return -1
endfunction
```

2a	2005	1	
2b	1 mark for each to max 2 • 1500 • 1952 • 2000 • 2100 • 2560	2	
2c	1 mark for each in the correct place • 1420 • 2050 • 2780 • 2600  2005 1920 2350 1500 1985 2100 2560 1420 1952 2000 2050 2780 2600	4	
2d	**Mark Band 3 – High level** **(7-9 marks)** The candidate demonstrates a thorough knowledge and understanding of search traversals; the material is generally accurate and detailed. The candidate is able to apply their knowledge and understanding directly and consistently to the context provided. Evidence/examples will be explicitly relevant to the explanation. *There is a well-developed line of reasoning which is clear and logically structured. The information presented is relevant and substantiated.* **Mark Band 2 – Mid level** **(4-6 marks)** The candidate demonstrates reasonable knowledge and understanding of search traversals; the material is generally accurate but at times underdeveloped. The candidate is able to apply their knowledge and understanding directly to the context provided although one or two opportunities are missed. Evidence/examples are for the most part implicitly relevant to the explanation. The candidate provides a reasonable discussion, the majority of which is focused. Evaluative comments are, for the most part appropriate, although one or two opportunities for development are missed. *There is a line of reasoning presented with some structure. The information presented is in the most part relevant and supported by some evidence.* **Mark Band 1 – Low Level** **(1-3 marks)** The candidate demonstrates a basic knowledge of search traversals with limited understanding shown; the material is basic and contains some inaccuracies. The candidates makes a limited attempt to apply acquired knowledge and understanding to the context provided. The candidate provides a limited discussion which is narrow in focus. Judgements if made are weak and unsubstantiated. *The information is basic and comunicated in an unstructured way. The information is supported by limited evidence and the relationship to the evidence may not be clear.* **0 marks** No attempt to answer the question or response is not worthy of credit.	**9** **AO1.1** **(2)** **AO1.2** **(2)** **AO2.1** **(2)** **AO3.3** **(3)**	**AO1: Knowledge and Understanding** **Indicative content** • Breadth first takes first value then visits all nodes connected to it. It then takes all nodes connected to those nodes. • Depth first goes to the left node, this becomes a new tree. It continues going to the left until a leaf. It then returns this, then goes right and repeats from the start. Follow left, follow right, take root. **AO2: Application** • Breadth will return 2005 1920 2350 1500 1985 2100 2560 (1420) 1952 2000 (2050) (2780) (2600) • Post-order / Depth will return (1420) 1500 1952 2000 1985 1920 (2050) 2100 (2600) (2780) 2560 2350 2005 **AO3: Evaluation** **Evaluations may vary and include one or more of the following points:** • Breadth is more efficient when the data searched for is closer to the root. • Depth is more efficient when data to be search for is further down. • Depth memory requirement is linear • Depth can be written recursively to aid understanding. • Breadth in general is better time complexity • In large trees depth may never return a value Candidates are not expected to know the complexities for the search traversals, however credit should be awarded if candidates choose to include these. Limit to band 2 if there is no evaluation of BFS/DFS
5ai	1 mark e.g. In directed **arcs/edges** may only go in 1 direction // in undirected **arcs/edges** can go in both directions	1	
5aii	1 mark • More than one path is allowed in a graph • Graphs do not have a <u>root</u> node • Graphs can be weighted • Graphs can have loops/cycles	1	Allow answers where candidates have given the reverse. e.g. a tree does not have loops.

8ai	1 mark for correct data 1 mark for correc top of stack pointer pointerValue 6 Index Data 8 7 6 5 7 4 6 3 3 2 6 1 5 0 10	2	
8aii	1 mark per bullet • Point to the next free space in the array • Points to the top of the stack	1	

8bi	1 mark per correctly completed statement
	
	e.g.
	`pubic function pop()`
	  `if pointerValue == 0 then`
	    `return -1`
	  `else`
	    `pointerValue = pointerValue -1`
	    `returnValue = stackArray[pointerValue]`
	    `return returnValue`
	  `endif`
	`endfunction`
		5	

8bii	1 mark per bullet to max 6
	• function header
	• ..taking parameter (ignore byval/byref)
	• checking if stack is full (pointerValue at 100)…
	• …and returning false
	• (otherwise) adding value to top of stack
	• …incrementing top of stack pointer
	• return true
	
	e.g.
	`function push(value)`
	  `if pointerValue < 100 then`
	    `stackArray[pointerValue] = value`
	    `pointerValue = pointerValue + 1`
	    `return true`
	  `else`
	     `return false`
	  `endif`
	`endfunction`
		6	Ignore additional parameters in function definition
			
			Do not accept the return of string values
			
			FT following a reasonable attempt to check if the stack is full

8biv	1 mark for each completed statement
	
	`returnValue = true`
	`while returnValue == true`
	  `returnValue = mathsStack.push(input("Enter Number"))`
	  `if returnValue == false then`
	    `print("Stack full")`
	  `endif`
		4	Accept equivalent for print e.g. output

8bv	1 mark per bullet to max 8
	• initialise a total to 0 outside of loop
	• looping
	• removing an item from the stack using the method pop
	• check if stack is empty
	• (if not) add value returned to total
	• ...outputting total
	• counting how many values are returned
	• stopping loop when either 20 items removed or no items left
	
	`total = 0`
	`quantity = 0`
	`returnValue = 0`
	
	`while quantity<20 and retunValue!=-1`
	`  returnValue = mathsStack.pop()`
	`  if(returnValue != -1) then`
	`    quantity = quantity + 1`
	`    total = total + returnValue`
	`    print(total)`
	`  endif`
	`endwhile`
		8	

8ci	1 mark per bullet to max • Queue has head pointer and tail pointer • When an item is enqueued the tail pointer increments • When an item is dequeued the head pointer increments	**3**	Max 1 mark for Enqueue/Dequeue operations if description of effect on tail/head pointers not given
8cii	**Mark Band 3 – High level** **(7-9 marks)** The candidate demonstrates a thorough knowledge and understanding of object-oriented and procedural programming; the material is generally accurate and detailed. The candidate is able to apply their knowledge and understanding directly and consistently to the context provided. Evidence/examples will be explicitly relevant to the explanation. *There is a well-developed line of reasoning which is clear and logically structured. The information presented is relevant and substantiated.* **Mark Band 2 – Mid level** **(4-6 marks)** The candidate demonstrates reasonable knowledge and understanding of object-oriented and procedural programming; the material is generally accurate but at times underdeveloped. The candidate is able to apply their knowledge and understanding directly to the context provided although one or two opportunities are missed. Evidence/examples are for the most part implicitly relevant to the explanation. The candidate provides a reasonable discussion, the majority of which is focused. Evaluative comments are, for the most part appropriate, although one or two opportunities for development are missed. *There is a line of reasoning presented with some structure. The information presented is in the most part relevant and supported by some evidence.* **Mark Band 1 – Low Level** **(1-3 marks)** The candidate demonstrates a basic knowledge of object-oriented and procedural programming with limited understanding shown; the material is basic and contains some inaccuracies. The candidates makes a limited attempt to apply acquired knowledge and understanding to the context provided. The candidate provides a limited discussion which is narrow in focus. Judgements if made are weak and unsubstantiated. *The information is basic and comunicated in an unstructured way. The information is supported by limited evidence and the relationship to the evidence may not be clear.* **0 marks** No attempt to answer the question or response is not worthy of credit.	**9** **AO1.1** **(2)** **AO1.2** **(2)** **AO2.1** **(2)** **AO3.3** **(3)**	**AO1: Knowledge and Understanding Indicative content** • OOP defines an object as an independent entity • OOP defines the attributes of the object and the methods that can be applied to it • attributes could be private to restrict accidental changes • Procedural the statements are executed in the order they are written **AO2: Application** • OOP allows for an object to be created from the queue • Many instances of this queue can then be declared in the main program. • Procedural will need each queue to be declared individually • Procedural will need to make use of subroutines where the queue will need to be sent and returned each time. **AO3: Evaluation** • OOP you can create multiple instances of the queue as required by the program without having to re-write all of the declarations etc. In procedural you would have to write separate code for each new stack • OOP reduces amount of code needed therefore fewer errors are likely as code is written once and then used multiple times • OOP can reduce mistakes because the subroutines are self-contained in procedural it would need to make sure the correct values are passed and returned, or global variables may be required which uses excess memory.

GCST

2021

1d	1 mark per bullet to max 6 • Visit root node **M** • Visit **E** and **S** • Visit **C** and **J** (from E) • …then **P** and **V** (from S) • Visit **G** and **K** (from J) • Visit **L** (from **K**)	6 AO1.2 (1) AO2.1 (3) AO2.2 (2)	

7a	1 mark for the purpose and 1 mark for matching appropriate name (4 marks total), e.g: • Pointer to the first element in the queue • firstElement // any other meaningful name • Pointer to the last element in the queue // Pointer to the first free element in the queue • lastElement / any other meaningful name	4 AO1.2 (4)	Must cover purpose and name for 2 marks for each pointer.
7b	1 mark per bullet up to a maximum of 5 marks, e.g: • Check if the queue is full • … if the firstElement pointer (+1) = lastElement // length variable == queue's capacity • … if it is return False • Adds element at lastElement (+1) position // Adds element at startposition+length • … increments lastElement pointer • If lastElement is greater than last Index // pointer becomes pointer MOD array.size • …reset to 0	5 AO1.2 (5)	Look out for variations in representing the queue

2020

1b	One node (node A) has more than 2 connections Nodes aren't ordered (e.g. F is C's left child)	**1** AO2.1 (1)	
1c	1 mark for identification • Null pointers	**1** AO2.1 (1)	
1d	1 mark per bullet • Take **A** as starting node • Visit **B, C** and **E** • Visit **D, F, G** and **H** • Visit **I** and **J**	**4** AO1.2 (2) AO2.2 (2)	Allow the reverse ordering from right to left e.g. A; E, C, B; H, G, F, D; J, I
1ei	1 mark per bullet to max 3 • Search the tree to find the location of Node E // by example of search • Replace the content of node E with blank/null/equivalent • Make node A point to the node H • Add node E to the empty node list	**3** AO1.2 (3)	
1eii	1 mark per bullet to max 3 • Search the tree to find the location of node G // by example of search • Create a new node with value K • Add a pointer from node G to the new node • Make node K point to null/equivalent	**3** AO1.2 (3)	
1f	1 mark per similarity to max 2 • Both consists of nodes • Both are connected by edges/links • Both are non-linear data structures • Both are dynamic data structures 1 mark per difference to max 2 • Tree is 1-directional whereas a graph is 2-directional • Tree has a root node whereas a graph does not have a (clear) root node • Tree will not have cycles whereas graphs can contain cycles • Tree will not be weighted whereas edges in a graph can be weighted	**4** AO1.1 (4)	

5a	1 mark per pointer • queueHead: Point to the first element in the queue // next element to remove • queueTail: Point to the last element in the queue	**2** AO1.2 (2)	
5b	1 mark per bullet up to max 5 • first 3 jobs removed • 128 and 129 added in positions 4 and 5 respectively • no additional jobs • queueHead being 3 (FT errors) • queueTail being 5 (FT errors) `queueHead` 3 `queueTail` 5 6 | 5 | `job-129` 4 | `job-128` 3 | `job-127` 2 | 1 | 0 |	**5** AO2.1 (2) AO2.2 (3)	The underlying implementation of the queue has not been specified, so allow alternative valid answers. e.g. queueHead = 0 queueTail = 2 Location 2: 129 Location 1: 128 Location 0: 127

5ci	1 mark per bullet to max 5
	• Function declaration
	• Checking if queue is empty
	• ...returning `null`
	• (Otherwise) incrementing `queueHead`
	• ...returning `buffer[queueHead-1]`
	e.g.
	`function dequeue()`
	  `if queueHead > queueTail then`
	    `return null`
	  `else`
	    `queueHead = queueHead + 1`
	    `return buffer[queueHead-1]`
	  `endif`
	`endfunction`
		5
		AO2.2
		(2)
		AO3.3
		(3)
			Note: Accept alternative valid underlying implementation answers e.g. Shifting all elements in queue forward.

5cii	1 mark per bullet to max 6
	• Function declaration taking parameter
	• Checking if queue is full
	• ...returning -1
	• (Otherwise) incrementing queueTail
	• Adding newJob to buffer(queueTail)
	• Returning 1
	
	e.g.
	`function enqueue(newJob)`
	  `if queueTail == 99 then`
	    `return -1`
	  `else`
	    `queueTail = queueTail + 1`
	    `buffer[queueTail] = newJob`
	    `return 1`
	  `endif`
	`endfunction`
		6
		AO2.2
		(3)
		AO3.3
		(3)	

5ciii	1 mark per bullet to max 8
	• Inputting user choice
	• If enqueue chosen input job name
	• …call enqueue with input value as parameter
	• …check if return value is -1 and output full
	• …otherwise output message that item is added
	• If dequeue chosen
	• …call dequeue **and** save returned value
	• …output returned value (jobname) if not null
	• …or output queue is empty
	e.g.
	`main()`
	`  choice = input("Add or remove?")`
	`  if choice == "ADD" then`
	`    jobname = input("Enter job name")`
	`    returnValue = enqueue(jobname)`
	`    if returnValue == -1 then`
	`      print("Queue full")`
	`    else`
	`      print("Job added")`
	`    endif`
	`  else`
	`    returnValue = dequeue()`
	`    if returnValue == null then`
	`     print("Queue empty")`
	`    else`
	`     output returnValue`
	`    endif`
	`  endif`
	`endmain`
		8
		AO2.2
		(2)
		AO3.3
		(6)
			Allow equivalent checks / logic

5d	1 mark per bullet to 3 • Check if either head or tail are incremented to above 99 • … set to be 0 instead • When checking if array is full check if (queueTail == queueHead – 1) OR (queueTail==99 AND queueHead==0)	**3** AO2.1 (1) AO2.2 (2)	Credit equivalent modulo arithmetic solution
5e	1 mark per bullet to max 3, e.g. • Use a different structure e.g. a linked list • …items can be added at different points in the linked list depending on priority • …by changing the pointers to items needing priority • Have different queues for different priorities • …add the job to the queue relevant to its priority • …print all the jobs in the highest priority queue first	**3** AO2.1 (2) AO2.1 (1)	Allow other suitable descriptions that show how the program could be amended.
6ai	1 mark per bullet • Points to where the next/first free node is • To add data into the linked listed.	**2** AO1.2 (1) AO2.2 (1)	
6aii	Points to the first element in the linked list	**1** AO1.2 (1)	

6aiii

1 mark per bullet
- No change made to nodes/pointers unaffected by this removal
- Index 0 points to 2 instead of 3
- Node 9 points to 3 instead of -1 // Node freeListPointer points to 3 instead of 4
- Node 3 points to 4 // -1 (must match MP3

Solution:

headPointer `0`

freeListPointer `4`

index	data	pointer
0	2.6	2
1	3.5	-1
2	1.8	1
3	~~6.9~~	-1
4		5
5		6
6		7
7		8
8		9
9		3

Alternative Solution:

headPointer `0`

freeListPointer `3`

index	data	pointer
0	2.6	2
1	3.5	-1
2	1.8	1
3	~~6.9~~	4
4		5
5		6
6		7
7		8
8		9
9		-1

Marks: **4** AO1.2 (1) AO2.2 (1)

Guidance: 6.9 3 may or may not be written by candidates, both are acceptable.

Candidates may add the node freed up (node 3) to the start or the end of the free storage. Award marks for both approaches.

6bii	1 mark per bullet to max 2 • A get method allows the attribute to be accessed / returned • A set method allows the attribute to be changed (with parameters)	**2** AO2.2 (2)
6c	1 mark per bullet to max 6 • Initialise message string • Start with the headPointer • Check if the headPointer is null • …return that the list is empty • Check the pointer of the node at headPointer • If it is not null/-1/the last element • loop through all the linkedList elements • …concatenate the pointer to the message • …replacing the pointer with the current node's pointer each time • …if the data is found concatenate the pointer and "found" to the message **and** return it • …if the loop ends and the data item is not found, concatenate "not found" to the message **and** return it	**6** AO1.2 (2) AO2.1 (2) AO2.2 (2)

6di	1 mark for identifying error and correction (identification may be implicit) • Line 02 tempPointer should become headPointer, not -1 `tempPointer = headPointer` • Line 05 message should say it's empty not full `print("List is empty")` • Line 07 pointer should be tempPointer `while linkedList[tempPointer].getPointer() != -1` • Line 08 Incorrect call to node pointer `dataToPrint = dataToPrint + " " + linkedList[tempPointer].getData()` • Line 09 assignment is wrong way `tempPointer = linkedList[tempPointer].getPointer()` • Line 11 missing final parenthesis `print(dataToPrint+ " " + linkedList[tempPointer].getData())`	**3** AO2.1 (2) AO2.2 (2)	Do not award marks for stating the line number without a valid correction.

GCST

6e	**Mark Band 3 – High level (9-12 marks)** The candidate demonstrates a thorough knowledge and understanding of the object orientied techniques; the material is generally accurate and detailed. The candidate is able to apply their knowledge and understanding directly and consistently to the context provided. Evidence/examples will be explicitly relevant to the explanation.The candidate is able to weigh up the use of all of the object orientied techniques which results in a supported and realistic judgment as to whether it is possible to use them in this context. *There is a well-developed line of reasoning which is clear and logically structured. The information presented is relevant and substantiated.* **Mark Band 2 – Mid level (5-8 marks)** The candidate demonstrates reasonable knowledge and understanding of the object orientied techniques; the material is generally accurate but at times underdeveloped. The candidate is able to apply their knowledge and understanding directly to the context provided although one or two opportunities are missed. Evidence/examples are for the most part implicitly relevant to the explanation. The candidate makes a reasonable attempt to come to a conclusion showing some recognition of influencing factors that would determine whether it is possible to use each object orientied technique in this context. *There is a line of reasoning presented with some structure. The information presented is in the most part relevant and supported by some evidence* **Mark Band 1 – Low Level (1-4 marks)** The candidate demonstrates a basic knowledge of the object orientied techniques with limited understanding shown; the material is basic and contains some inaccuracies. The candidates makes a limited attempt to apply acquired knowledge and understanding to the context provided. The candidate provides nothing more than an unsupported assertion. *The information is basic and comunicated in an unstructured way. The information is supported by limited evidence and the relationship to the evidence may not be clear.*	12 AO1.1 (3) AO1.2 (3) AO2.1 (3) AO3.3 (3)	**AO1: Knowledge and Understanding** **Indicative content** • Classes, this a template. It will define what attributes and methods an object should have. • Objects, when you create an instance of a class. Each object that is instantiated from the same class will share the same attributes and methods. • Inheritance, when a sub class takes on the attributes and methods from a superclass/parent class. It can also have its own extra attributes/methods. • Overriding, when a method name is the same in a parent and sub class, then the method in the parent/super class will be overridden • Encapsulation, this protects attributes of an object by making them private so that they can't be accessed or altered accidentally by other objects. **AO2: Application** • A class can be used to declare the attributes and methods for the linked list. These will initialise the nodes and join them. • Objects can then be used be used to instantiate the class each time a new linked list is needed. Each can be given a different identifier by the other programs. • Further subclasses may be used by other programs. These can therefore take on the attributes and methods from the base class. These can also be changed or overridden depending on the purpose of the other programs. • Encapsulation can be used by using set and get methods to ensure that the nodes in the linked list are changed in a way that is intended. **AO3: Evaluation**
	0 marks No attempt to answer the question or response is not worthy of credit.		• Use of OPP techniques will allow for code reusability. His linked list can be saved as library and then reused many times leading to less code. • OOP also allows programs to be easier to modify and maintain.

AS - Level

3	(a)	(i)	1 mark per bullet up to a maximum of 4 marks: • A queue is First In First Out (FIFO) • Therefore bookings will be executed in the order they have received • A stack is Last In First Out (LIFO) • Therefore the bookings would be executed from the most recent booking	4 A02.2 (4)	
3	(a)	(ii)	`custNumber`	1 A03.3 (1)	
3	(a)	(iii)	1 mark per bullet up to a maximum of 2 marks: • Correct logic for incrementing the tail pointer by 1 (e.g. `tail = tail +1`) • Correct logic for adding `custNumber` to the tail pointer (e.g. `queue[tail]=custNumber`)	2 A03.2 (2)	

3	(a)	(iv)	• If the tail is greater than 10 / `maxElements`	1
				A02.2
				(1)
					Accept:
					`not((tail + 1) > maxElements)`
					Or
					`(tail + 1) <= maxElements`
					Or equivalent

1a	1 mark per bullet • Queue outputs data in a First In First Out fashion • It will retrieve the temperature values in the order they were recorded or • Stack outputs the data in a Last In First Out fashion • It will retrieve the temperature values in the reverse of the order they were recorded	2 AO1.2 (1) AO2.2 (1)	Mark Point 1 is the definition Mark Point 2 is for context of the temperature values
1bi	It returns a value	1 AO2.1 (1)	

1bii	1 mark per completed word
	`processedData[0] = 0`
	`firstDay =` **dequeue()**
	`for count = 1 to 6`
	    `processedData[`**count**`] = dequeue() -` **firstDay**
	`next count`
		3
		
		AO2.2 (1)
		AO3.2 (2)
			Exact answers only

2ai	Binary Tree / Binary Search Tree	1 AO2.1 (1)	
2aii	1 mark per bullet • 1st layer: Lily • 2nd layer: Daisy, Sunflower • 3rd layer: Begonia, Hosta, Peony 4th layer: Rose	3 AO1.2 (1) AO2.1 (1) AO2.2 (1)	
2aiii	1 mark per bullet to max 4. Max 2 marks for no application to the tree. • Depth first starts at the root (Lily) • ...and goes all the way down one branch to the bottom (Begonia) • It stores which nodes it has visited / pushes nodes visited onto a stack • When it cannot go any further • ...It then backtracks/returns to the previous node • And continues to backtrack until a node is reached with unvisited children. • ...and checks down that branch • In the tree shown, after visiting Begonia, the algorithm would backtrack to Daisy... • ...and would then visit Hosta *(Accept any other example)*	4 AO1.1 (1) AO1.2 (1) AO2.1 (1) AO2.2 (1)	

2bi

1 mark per bullet

- Correct NextPointer values
- Suitable end/null pointer

Data item	Data	NextPointer
0	Begonia	**1**
1	Daisy	**2**
2	Hosta	**3**
3	Lily	**4**
4	Peony	**5**
5	Rose	**6**
6	Sunflower	null

2

AO2.1 (2)

Exact values only. Allow -1 for null pointer or equivalent such as Φ.
Do not allow a blank or 0.

2bii

1 mark per bullet

- Lavender added in position 7
- ...Hosta points to 7
- ...Lavender points to 3

Data item	Data	NextPointer
0	Begonia	1
1	Daisy	2
2	Hosta	**7**
3	Lily	4
4	Peony	5
5	Rose	6
6	Sunflower	null
7	Lavender	**3**

3

AO1.2 (1)
AO2.2 (2)

Do not credit answers that do not place lavend
in position 7 and then update pointer positions

2biii

1 mark per bullet

- Traverse the list to the item immediately prior to the item to be removed (1)
- ... which is DataItem 1 - Daisy
- Find the value of the NextPointer of the item to be removed
- ... which is the NextPointer of DataItem 2 - Hosta, value 7
- Set the nextPointer of the item prior to the item to be removed to the NextPointer value of the DataItem to be removed
- ... update the NextPointer of DataItem 1 - Daisy from 2 to 7 (Lavender)

4

AO1.1 (1)
AO1.2 (1)
AO2.2 (2)

Find the *item before* item to be deleted (Daisy)
Find nextPtr of item to be deleted (Hosta)
Update nextPtr of the *item before (Daisy)* to the nextPtr of item to be deleted (Hosta)
i.e. Daisy 2 is updated to Daisy 7

Allow FT from 2b(ii/iii) if candidate has used table in fig 2.1 (e.g. Daisy would now point to Lily at position 3)

2biv

1 mark per bullet

- Start at the `firstElement` in the list
- Correctly looping until null pointer found / end of list
- Outputting the data element
- Accessing the pointer to the next element
- Appropriate comment(s)

e.g.

```
currentElement = firstElement
while(currentElement != null) //Continue until last
node
  print(plantList[currentElement,0])
  currentElement = plantList[currentElement,1]
endwhile
```

5

AO2.1 (1)
AO2.2 (2)
AO3.2 (2)

Note: Solution must utilise pointers in a linked list; it cannot use a FOR loop as the number of elements is not known and the data is not in order by index number

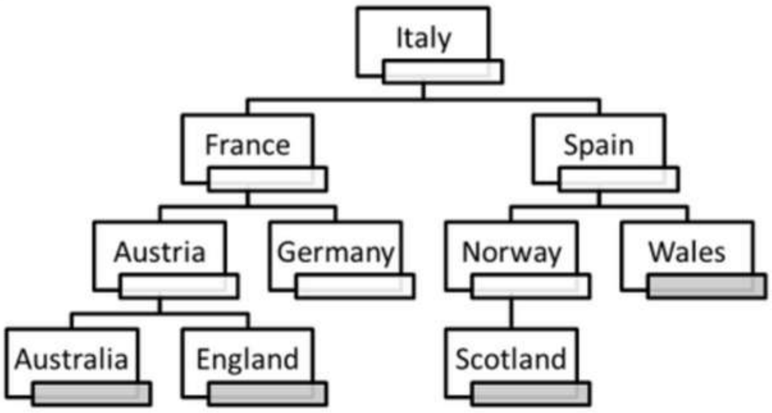
Note: identifiers given in the question as `plantList` and `firstElement` should be used accurately in the solution

Note: allow credit for answers that interpret the data structure as an array of records/structures with data/pointer fields

3	a		1 mark per bullet to max 2 • both are pointers • front gives the (array) position/index of the first element in the queue • end gives the (array) position/index of the last element in the queue	**2** **AO1.1 (1)** **AO1.2 (1)**	
3	b	i	1 mark per bullet • elements in the queue (correct four colours) • …in the correct positions • front points to the first element in queue • end points to the last elements in queue | | | | blue | grey | orange | maroon | | front = 3 end = 6	**4** **AO1.1 (1)** **AO1.2 (1)** **AO2.1 (1)** **AO2.2 (1)**	Allow FT for values for front/end
3	b	ii	Function has to return a value, procedure does not	**1** **AO1.1 (1)**	
3	b	iii	1 mark per bullet to max 4 • Check if the queue is full… • … e.g. if the start and end pointers are equal // are at index 0 // description of checking circular queue • …If the queue is full, return full / error message • Otherwise, increment the end pointer… • … and add the new item at the position indicated by the end pointer • …return added/equivalent	**4** **AO1.1 (2)** **AO2.1 (2)**	

GCST

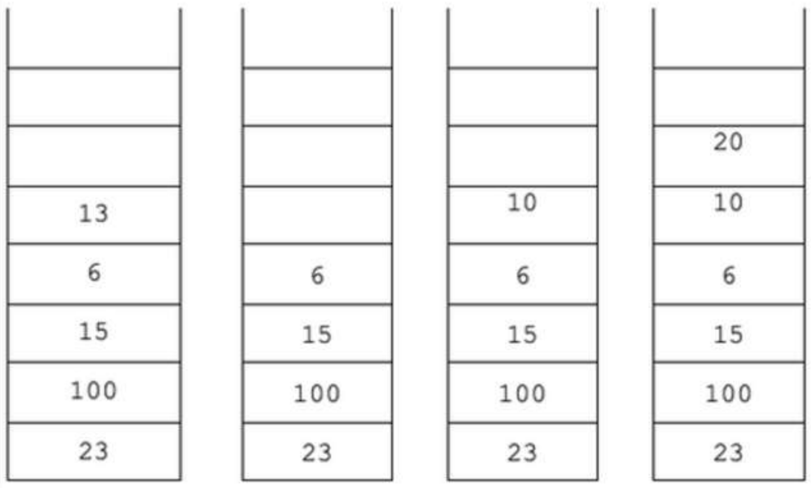
2018

1	(a)		 Italy France Spain Austria Germany Norway Wales Australia England Scotland 1 mark for each of: - Scotland in correct place - Wales in correct place - Australia and England both in correct place	3 AO2.2 (3)	
1	(b)		1 mark per bullet to max • Italy • France, Spain • Austria, Germany, Norway	3 AO1.1 (1) AO2.1 (1) AO2.2 (1)	

1	(c)	(i)	1 mark per bullet to max 5 (see code below)	5 AO2.2 (2) AO3.2 (3)	The line **elseif thisNode < searchValue then** should have read **elseif thisNode > searchValue then** If candidates attempt to correct the code and their answers are consistent with, and work with their amendment, such answers should be credited.

```
function searchForData(currentNode:byVal, searchValue:byVal)
      thisNode = getData(currentNode)
      if thisNode == searchValue then
         return true
      elseif thisNode < searchValue then
         if currentNode.left () != null then
              return (searchForData(currentNode.left (),
searchValue))
         else
              return false
         endif
     else
         if currentNode.right() != null then
         return (searchForData(currentNode.right (),
searchValue))
         else
              return false
         endif
     endif
endfunction
```

1	(c)	(ii)	• It's a binary tree • It's ordered / sorted	2 AO2.2 (2)	

5	(a)		1 mark for each correct stack  13 6 15 100 23 6 15 100 23 10 6 15 100 23 20 10 6 15 100 23	**4** **AO1.2 (2)** **AO2.2 (2)**	
5	(b)	(i)	1 mark per bullet, max 2 for insert, max 2 for remove push • Check if the stack is full (pointer = array.length/array.length+1) • If it is not – insert the item • If it is – return/error that the stack is full pop • Check if the stack is empty (pointer = 0/1) • If it is – return/error that the stack is empty • If it is not – return the item	4 AO1.2 (2) AO2.2 (2)	
5	(b)	(ii)	1 mark per line, 1 for change • line 02 • Include an OR with variations (e.g. `userAnswer = "PUSH" OR userAnswer = "Push"` etc.)/Convert input to uppercase/lowercase and just compare to equivalent	2 AO2.2 (2)	
5	(c)		1 mark per bullet to max 3 • Array size defined • A stack pointer is used to point to the top of the stack • When an item is pushed the stack pointer is incremented • When an item is popped the stack pointer is decremented	3 AO1.2 (1) AO2.1 (1) AO2.2 (1)	
6	(a)	(i)	1 mark per bullet to max 3 • Record is a data structure… • …A class is a template for making data structures (objects) • Class also has methods (which describes functionality) • Both store data of different types • Which can be accessed by their names • But classes can make them accessible via methods • Both can have multiple 'instances' • Class can include visibility of properties / private	3 AO1.2 (3)	

6	(a)	(ii)	1 mark per space
			
			`recordStructure` **`items`**
			    `itemName :` **`String`**
			    `cost` `: Currency`
			    `dateArrival` `: Date`
			    `transferred :` **`Boolean`**
			`endRecordStructure`
				5
				AO2.2 (2)
				AO3.2 (3)	

6	(a)	(iii)	1 mark per bullet to max 3
			• Declaring box1 as an item
			• Using Box1. (or equivalent) for each variable
			• Setting each variable (matching 6aii) correctly
			
			e.g.
			`Box1 : Items`
			`Box1.itemName = "Box"`
			`Box1.cost = 22.58`
			`Box1.dateArrival = "1/5/2018"`
			`Box1.transfered = True`
				3
				AO2.2 (2)
				AO3.2 (1)
					Ensure variable names for cost and dateArrival are consistent with variable names given in a(ii)

6	(b)	(i)	1 mark per bullet to max 2 • A data structure • FIFO (first in first out)	2 AO1.1 (2)	

6	(b)	(ii)	1 mark per bullet to max 2 • Properties (are encapsulated) and can only be accessed through their methods • Enforce validation through the method // inappropriate data can be caught before entered • Cannot be changed/accessed accidentally	2 AO1.2 (2)	

6	(b)	(iii)	1 mark per bullet to max
			• Constructor method/`new`
			• Setting `head` and `tail` to 0 within constructor method
			
			e.g.
			`public procedure new()`
			`  head = 0`
			`  tail = 0`
			`  numItems = 0`
			`endprocedure`
				2
				AO2.2 (1)
				AO3.2 (1)	

6	(b)	(iv)	1 mark per bullet to max 6
			• Function declaration, taking item as a parameter
			• Checking if the queue is full…
			• …outputting/reporting error and returning `false`
			• Adding the item to the `tail` position
			• Correctly updating the `tail` pointer (either before or after addition)
			• Incrementing `numItems` and returning `true` if successful
			
			e.g.
			`public function enqueue(newItem : items) : boolean`
			`  if numItems = 10 then`
			`    print("Error: The queue is full")`
			`    return false`
			`  else`
			`    theItems[tail] = newItem`
			`    if tail = 9 then`
			`      tail = 0`
			`    else`
			`      tail += 1`
			`    endif`
			`    numItems += 1`
			`    return true`
			`  endif`
			`endprocedure`
				6
				AO2.2 (3)
				AO3.1 (1)
				AO3.2 (2)	

6	(b)	(v)	e.g. `myItems = (new) itemQueue()`	1 AO2.1 (1)	Allow follow through if they have parameter in 6(b)(iii)

6	(b)	(vi)	1 mark per bullet to max 5
			• Procedure declaration for `insertItems`
			• Asking for input of data items for a new item .....
			• ...using record structure correctly
			• Use of `myItems.enqueue`
			• Looping while the queue is not full
			
			e.g.
			`procedure insertItems()`
			
			`  newItem : Items`
			`  itemCount = myItems.getnumItems()`
			
			`  while itemCount < 10`
			`    newItem.itemName = input("Enter the item name")`
			`    newItem.cost = input("Enter the item cost")`
			`    newItem.dateArrival = input("Enter the date of arrival")`
			`    newItem.transferred = input("Has it been transferred?")`
			`    myItems.enqueue(newItem)`
			`    itemCount = itemCount + 1`
			`  endwhile`
			
			`  myItems.setnumItems(itemCount)`
			
			`endprocedure`
				5
				AO2.2 (2)
				AO3.1 (1)
				AO3.2 (2)	

1 a — 1 mark per bullet to max 2 — **2 AO1.2 (1) AO2.2 (1)**

- top = 6 (allow FT if numbers entered incorrectly)
- Numbers entered in order

top	6

index	stackItem
9	
8	
7	
6	
5	1000
4	59
3	2
2	13
1	22
0	20

1 b i — A procedure does not return a value / a function has to return a value — **1 AO1.2 (1)**

1 b ii — 1 mark per bullet, max 2 for by value, max 2 for by reference — **4 AO1.2 (4)**

by value:

- A local copy of the data is used
- Data is discarded when the subprogram exits
- Does not override/change the original data

by reference:

- Memory location of data is sent
- Changes are made to the original data
- Changes remain after the subprogram exits

1 b iii — 1 mark for each completed space to max 5 — **5 AO2.2 (2) AO3.2 (3)** — Accept `numStack.length()` instead of `20`

```
function addItem(number)
    if top = 20 then
        return false
    else
        numStack[top] = number
        top = top + 1
        return true
    endif
endfunction
```

1 b iv — 1 mark for each bullet for max 6 — **6 AO1.2 (3) AO2.2 (3)**

- Initialising total to 0 and add to true
- top = 4 and total = 8
- total = 20 and add = false
- top = 3, total = 14, add = true
- top 2, 1. Total 16, -4, add false, true
- Output = -4

	numStack								
top	0	1	2	3	4	5	total	add	Output
5	20	2	6	12	8				
							0	true	
4							8		
							20	false	
3							14	true	
2							16	false	
1							-4	true	
0									-4

2	a	i	1 mark per bullet to max 2 • Can store multiple values under one identifier • The data can be of different data types	**2** **AO1.1** **(2)**	
2	a	ii	1 mark per bullet to max 1 • Allows easy access/indexing/manipulation of each data item in turn • 1D Array can hold multiple items of same data type • Maximum number of array elements is known	**1** **AO2.2** **(1)**	
2	a	iii	1 mark per bullet to max 1 • Queue • List • Tuple	**1** **AO1.1** **(1)**	Graph Tree Linked list

2	c	i	1 mark per bullet to max 7 • Inputting the number of players • …with validation between 1 and 10 • Inputting and storing the name of the correct number of players • Outputting the round number at the start of all 8 rounds • Outputting each player's number, in each round • Inputting the players' score for each round • Add the score to their `totalScore` e.g, (see code below)	**7** **AO2.2** **(1)** **AO3.2** **(6)**	

```
Do
   numberOfPlayers = input("Enter the number of players between 1 and
10")
until numberOfPlayers >= 1 AND numberOfPlayers <= 10

for player = 0 to numberOfPlayers - 1
  scores[player].playerName = input("Enter player name")
next player

for roundNum = 0 to 7
  print("Round " + str(roundNum + 1))
  for player = 0 to numberOfPlayers - 1
     score = input("Enter player " + str(player + 1) + " score")
     scores[player].totalScore = scores[player].totalScore + score
  next player
next roundNum
```

2	a		1 mark per bullet, to max 2, e.g. • Orders can be processed in the order they are in the queue • Orders can be inserted at any place in the list e.g. high priority item inserted earlier in the list • Orders can be deleted from any position in the list once they are complete • List is dynamic… • … to allow orders to be added / deleted	2 AO1.2 (1) AO2.1 (1)	
2	b	i	1 mark per bullet • `nodeNo` and `next` columns are both correct • `orderNo` column is correct	2 AO1.2 (2)	

nodeNo	orderNo	next
∅	154	1
1	157	2
2	155	3
3	156	4
4	158	∅

2	b	ii	1 mark per correct column	3 AO1.2 (3)	

nodeNo	orderNo	next
∅	154	4
1	157	2
2	155	3
3	156	∅
4	159	1

2	c	i	• The index/subscript of the array acts as the nodeNo	1 AO1.2 (1)	

2	c	iv	1 mark per bullet to max • Order 190 is added to the end • Pointers are updated • 186 will point to 4 • 190 will point to 2 OR	4 AO1.2 (2) AO2.1 (2)	If a diagram is given then the r updating the pointers is implici

Index	0	1	2	3	4
Data	184	186	185	187	190
Pointer	1	4	3		2

3	c		1 mark per bullet to max 5 • Use of appropriate loop • Correct end condition (length of `message`) • Correct use of `.push` with `messageStack` • Accessing `substring` (or equivalent) correctly • Appropriate comment(s)	5 AO2.1 (2) AO3.2 (3)	

```
procedure pushToStack(message)
   for x = 0 to message.length() //loop through each
                                 //letter
       messageStack.push(message.substring(x,1)) //take
              //each character and push onto stack
   next x  //move to next letter
endprocedure
```

3	d		1 mark per bullet to max 5 • Pop element from stack • Convert to ASCII value • Subtract 10 from ASCII value • Convert back to character • Append/concatenate with variable	5 AO1.2 (2) AO2.2 (3)	Accept pseudocode equivalent.
4	a		Tree // Graph (undirected)	1 AO1.2 (1)	Do not accept binary tree
4	b	i	1 mark per bullet to max 4 • Depth-first goes to left child node when it can... • If there is no left child it goes to the right child • when there are no child nodes the algorithm 'visits' it' and backtracks to the parent node. • Breadth-first visits all nodes connected directly to start node... • Then visits all nodes directly connected to each of those nodes (and then all nodes directly connected to those nodes and so on…) • Depth-first uses a stack • Breadth-first uses a queue	4 AO1.2 (4)	
4	b	ii	1 mark per node in correct order D→K→L→H→B→G→(X)	6 AO2.1 (6)	
4	b	iii	Max 3 e.g. • When a node does not have any node to visit e.g. D • The algorithm goes back to the previous visited node e.g. B • To check for further nodes to visit e.g. H • This repeats until a new node can be visited, or all nodes have been visited	3 AO1.2 (2) AO2.1 (1)	

4	a		1 mark per bullet to max 3 e.g. • A queue is First In First Out (FIFO) [1] • The questions are retrieved in the order they are stored [1] • Questions can be added to the end [1] • Dynamic structure... [1] • ...expands to take more questions [1]	**3** **AO1.2 (2)** **AO2.1 (1)**	
4	b	i	"6+1" in the correct box. [1] Speech marks present [1] "2*3" | "1+4" | "3-1" | "10/2" | "3+6" | "6+1" | |	**2** **AO2.1 (2)**	
4	b	ii	1 mark for head, 1 for tail `head` = 0 [1] `tail` = 5 [1]	**2** **AO2.1 (2)**	

4	c		1 mark for pseudocode/code that meets each bullet
			• Checking if queue is empty [1]
			• ...outputting message/reporting error [1]
			• Outputting element in `questions` at index `head` [1]
			• Increment `head` [1]
			
			e.g.
			`procedure remove()`
			`    if head == tail + 1 then`
			`        print("No questions")`
			`    else`
			`        print(questions[head])`
			`        head = head + 1`
			`    endif`
			`endprocedure`
				4
				AO3.2 (4)	

4	d		1 mark for pseudocode/code that meets each bullet
			• Input a question [1]
			• Check if `tail` is full and outputs message/reports error [1]
			• Increment `tail` [1]
			• Adds question to `tail` of `questions` [1]
			
			e.g.
			`procedure add()`
			`    maxElements = 10`
			`    item = input("Enter a question")`
			`    if tail == maxElements - 1 then`
			`        print("Queue is full")`
			`    else`
			`        tail = tail + 1`
			`        questions[tail]=item`
			
			`    endif`
			`endprocedure`
				4
				AO3.2 (4)	

4			• String length calculated (1)
			• Correct number of characters from passed string taken ... (1)
			• ... in reverse order (1)
			• Characters placed in stack in correct order (1)
			• String length placed in stack at correct point (1)
			• Meaningful variable names used (1)
			(AO2.1)
			
			Example program
			
			`procedure passToStack(passString)`
			`   stringLen = passString.Length()`
			`   if stringLen == 0 then`
			`      stack[0]=0`
			`   else`
			`      stackPtr = 0`
			`      stringPtr = stringLen - 1`
			`      for i = 1 TO stringLen`
			`         stack[stackPtr] =`
			`passString[stringPtr]`
			`         stackPtr = stackPtr + 1`
			`         stringPtr = stringPtr -1`
			`      next i`
			`      stack[stackPtr] = stringLen`
			`   endif`
			`endprocedure`
				6	Allow `StackPtr` to be used instead of `i` in loop, as we would not expect them to know that some compilers do not always increment "loop counter" when they exit loops (i.e. loop counter on exit is undefined)
					
					Accept candidates using built-in stack methods e.g.
					`stack.push(word.substring(i,1))`
					
					Do not penalise for syntax errors if the logic can clearly be followed.
					
					Max 6 mark
					
					Examiner's Comments
					
					Many of the same comments regarding pseudocode as in 4b once again applied in 4d. An encouraging number of able candidates produced quite elegant solutions.