Past-paper walk-through

Introduction to Abstract Data Types (ADT) ▪ 10.4

eXercise

Questions in this set

- 9618/23 N25 Q3 ▪ 5 marks
- 9618/31 N25 Q9 ▪ 9 marks
- 9618/43 N25 Q2 ▪ 24 marks
- 9618/21 J25 Q5 ▪ 7 marks
- 9618/23 J25 Q2 ▪ 11 marks
- 9618/31 J25 Q4 ▪ 7 marks
- 9618/22 N24 Q3 ▪ 7 marks
- 9618/23 N24 Q3 ▪ 8 marks
- 9618/21 J24 Q3 ▪ 11 marks

Reading the mark scheme

- **B1** a mark that stands on its own – the point is either made or not.
- **M1** a *method* mark: the right approach, even if the number is wrong.
- **A1** an *answer* mark, and it usually needs its M mark first.
- **C1** a working mark on the way to the answer.
- // separates answers the examiner accepts equally.
- **ecf** = error carried forward: a later part is marked on your own earlier answer.
- **owtte** = "or words to that effect" – the wording need not match.

Question 3

9618/23 N25 ▪ Q3 ▪ 5 marks

A program uses a stack to hold up to 30 integer values. The stack is implemented using a global integer variable and a global 1D array.
The array is declared in pseudocode: DECLARE ThisStack : ARRAY[1:30] OF INTEGER
Stack design notes:

- The global variable `SP` acts as a stack pointer. `SP` contains the array index of the last value pushed onto the stack.
- If the stack is empty, then `SP` is assigned the value zero.
- The first item added to the stack will be stored in `ThisStack`
- `SP` is incremented each time an item is added to the stack.

Question 3

A function `Pop()` is written to remove an item from `ThisStack`. The function returns an item of type `PopData` which is defined in pseudocode:
TYPE PopData DECLARE Data : INTEGER DECLARE Exists : BOOLEAN
ENDTYPE
The value removed from the stack is assigned to `Data` and `Exists` is set to `TRUE`. If it is not possible to remove a value from the stack, then `Exists` is set to `FALSE`
Complete the pseudocode for `Pop()`
FUNCTION Pop() RETURNS PopData
PopData.Exists <- // Stack is empty
ELSE
PopData.Data <-
PopData.Exists <-
ENDIF

Question 3

```
RETURN ThisPop
ENDFUNCTION
```

[5]

Solution 3

Worked steps

Read the design notes before writing anything — they fix every decision.

- `SP` holds the index of the **last value pushed**, so the top of the stack is `ThisStack[SP]`, not `ThisStack[SP - 1]`.
- An empty stack has `SP = 0`, so "is it empty?" is the test `SP < 1`.
- The function returns a **record** of type `PopData`, which is how it reports both a value and whether there was one.

```
FUNCTION Pop() RETURNS PopData
    DECLARE ThisPop : PopData

    IF SP < 1 THEN
        ThisPop.Exists <- FALSE
    ELSE
        ThisPop.Data   <- ThisStack[SP]
        ThisPop.Exists <- TRUE
        SP <- SP - 1
    ENDIF
    RETURN ThisPop
ENDFUNCTION
```

Solution 3

Five marks, and they are awarded for five separate decisions:

1. **Both** assignments to `.Exists` — FALSE on the empty branch and TRUE on the other. These share one mark, so a solution that sets only one of them earns nothing here.
2. Declaring a local variable of type `PopData` to build the answer in.
3. The **empty test**, `SP < 1` or `SP = 0`.
4. Reading the value: `ThisPop.Data <- ThisStack[SP]`.
5. **Decrementing** `SP`.

Solution 3

Two things decide whether this works. Read the value **before** moving the pointer — decrement first and you return the item below the top. And do **not** erase `ThisStack[SP]`: the value stays in the array and is simply outside the stack now, which is the whole reason the array never has to be initialised.

Solution 3

Returning a record is what lets one function answer two questions at once. A function that returned only the value would have no way to say "the stack was empty" that could not also be a legitimate stored value. **Mark scheme**

- FUNCTION Pop() RETURNS PopData
- DECLARE ThisPop : PopData MP 2
- IF SP < 1 // SP = 0 THEN MP 3
- PopData.Exists <- FALSE // Stack is empty MP 1
- ELSE
- PopData.Data <- ThisStack[SP] MP 4
- PopData.Exists <- TRUE MP 1
- SP <- SP – 1 MP 5
- ENDIF
- RETURN ThisPop‘
- ENDFUNCTION
- Mark as follows:

Question 9

9618/31 N25 ▪ Q9 ▪ 9 marks

A stack has been implemented using pseudocode to store a maximum of 100 string items using the global variables in the following table:

(a)(i) Complete the **pseudocode** for the function to remove a data item from the stack.

DECLARE DataItem : STRING DataItem <- ""

ELSE DataItem <- "You cannot remove data; the stack is empty" ENDIF

ENDFUNCTION **[5]**

(a)(ii) Write the **pseudocode** to output the data item removed from the stack with an appropriate message. **[1]**

(b) A stack is used to implement recursion.

State the **three** essential features of recursion.

1. 2. 3. **[3]**

Question 9

Identifier	Data type	Description	Initialisation value
`Base`	`INTEGER`	pointer for the bottom of the stack	`0`
`Top`	`INTEGER`	pointer for the top of the stack	`-1`
`StackArray`	`STRING`	1D array to implement the stack	`[0:99]`
`Max`	`INTEGER`	maximum number of items in the stack	`100`

Solution 9

(a)(i) Worked steps

A Pop has to answer two questions in one function: is there anything to remove, and if so what. Deal with the empty case first.

```
FUNCTION Pop() RETURNS STRING
    DECLARE DataItem : STRING
    DataItem <- ""
    IF Top > -1 THEN
        DataItem <- StackArray[Top]
        Top <- Top - 1
    ELSE
        DataItem <- "You cannot remove data; the stack is empty"
    ENDIF
    RETURN DataItem
ENDFUNCTION
```

Solution 9

The five gaps, and what each has to be:

- `RETURNS STRING` on the header — the function hands a value back.
- The **emptiness test** on `Top`, before any indexing. Getting this wrong reads outside the array.
- Read the item at `StackArray[Top]` — the top element, not `Top + 1`.
- **Decrement** `Top` after reading: the item is gone.
- A message in the ELSE branch, so the caller is told rather than getting an empty string.

Solution 9

Read then decrement is the order that matters. Decrementing first would return the item below the top. **Mark scheme**

- One mark for each correctly completed line (Max 5)
- FUNCTION Pop() RETURNS STRING
- DECLARE DataItem : STRING
- DataItem <- ""
- IF Top > –1 // Top >= Base THEN
- DataItem <- StackArray[Top]
- Top <- Top – 1
- ELSE
- DataItem <- "You cannot remove data; the stack is empty"
- ENDIF
- RETURN DataItem // StackArray[Top + 1]

Solution 9

(a)(ii) Worked steps

```
OUTPUT "The data removed from the stack is ", Pop()
```

Solution 9

Call the function inside the OUTPUT: it returns a value, so it can be used wherever a value is expected.

Mark scheme

- OUTPUT "The data removed from the stack is ", Pop()

Solution 9

(b) Mark scheme

- One mark per mark point (Max 3)
- MP1
- A recursive algorithm must call itself / have a general case
- MP2
- It must have a base case / have a stopping condition
- MP3
- It must change its state and move towards the base case

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data.
The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string ""
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

Question 2

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

(a) Write program code to declare and initialise `Queue`, `QueueHead`, `QueueTail` and `NumberItems`

Save your program as **Question2_N25**.

Copy and paste the program code into part **2(a)** in the evidence document. **[2]**

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data. The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string `""`
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

Question 2

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

(b) The function `Enqueue()` takes a string as a parameter. The function checks if the queue is full, and returns Boolean `FALSE` if the queue is full.

If the queue is not full, the parameter is inserted into the next position in `Queue`. The function updates the appropriate pointer(s), updates `NumberItems` and then returns Boolean `TRUE`

Write program code for `Enqueue()`

Save your program.

Copy and paste the program code into part **2(b)** in the evidence document. **[5]**

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data. The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string `""`
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

Question 2

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

(c) The function `Dequeue()` returns the string `"False"` if the queue is empty.
If the queue is not empty, the function returns the next element in the queue. The function updates the appropriate pointer(s) and updates `NumberItems`
Write program code for `Dequeue()`
Save your program.
Copy and paste the program code into part **2(c)** in the evidence document. **[3]**

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data. The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string `""`
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

Question 2

(d) The text file `BinaryData.txt` stores individual binary digits, '1' and '0'. Each digit is on a new line. For example, the first line in the text file stores '1', the second line stores '1'.
The procedure `ReadData()` reads in each line from the text file `BinaryData.txt` and inserts it into the queue using the appropriate method.
The procedure needs to work for a text file with any number of lines up to a maximum of 100.
Write program code for `ReadData()`

Save your program.
Copy and paste the program code into part **2(d)** in the evidence document.

Question 2

[5]

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data. The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string `""`
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

Question 2

(e) The string data in the text file is compressed.
The compression algorithm counts the number of times each binary digit appears consecutively, then stores the binary digit followed by the number of times it appears.
The algorithm stores the compressed data in a single string.
For example, if the text file contains the data:
1 1 0 0 0 1 1 1
The compression algorithm will create the string "120313" because there are two '1' digits, followed by three '0' digits, followed by three '1' digits.

Question 2

The procedure `Compress()` uses `Dequeue()` to remove each element from the queue in turn. The procedure then compresses the data following the compression algorithm described. The new compressed string is stored in the global variable `NewString`
You can assume that one binary digit will never appear more than nine times consecutively in the sequence.
You can assume that there will always be at least one item in the queue.
Write program code for `Compress()`

Save your program. Copy and paste the program code into part **2(e)** in the evidence document.

Question 2

[6]

Question 2

9618/43 N25 ▪ Q2 ▪ 24 marks

A program reads string data from a text file into a linear queue and then compresses the data. The queue is created using the global 1D array `Queue`. The queue can store up to 100 elements. Each element is initialised to the empty string `""`
The queue has two pointers that are global to the program:

- `QueueHead` that points to the index of the first element in the queue, initialised to -1
- `QueueTail` that points to the index of the last element in the queue, initialised to -1

The global variable `NumberItems` stores the number of elements stored in the queue, initialised to 0

(f)(i) Write program code for the main program to:

- call `ReadData()`
- call `Compress()`

Question 2

- output the content of the compressed string.

Save your program.
Copy and paste the program code into part **2(f)(i)** in the evidence document. **[2]**
(f)(ii) Test your program.
Take a screenshot of the output(s).
Save your program.
Copy and paste the screenshot(s) into part **2(f)(ii)** in the evidence document. **[1]**

Solution 2

(a) Worked steps

Four things are declared, and each has a starting value the rest of the question depends on.

```
DECLARE Queue : ARRAY[0:99] OF STRING
DECLARE QueueHead, QueueTail, NumberItems : INTEGER

FOR Index <- 0 TO 99
    Queue[Index] <- ""
NEXT Index

QueueHead <- -1
QueueTail <- -1
NumberItems <- 0
```

Solution 2

- The array is **global**, 100 elements, of type STRING, and every element starts as the empty string — that is one mark, and the loop is what earns it.
- The pointers start at **−1**, not 0, because 0 is a valid index: -1 is the value that means *this queue is empty*. `NumberItems` starts at 0.

Solution 2

If your language indexes from 0, declare 100 elements as 0 to 99; if from 1, use 1 to 100 and keep the rest of the question consistent with that choice. **Mark scheme**

- 1 mark each
- ▪ Queue declared as a (global) 1D array of 100 string elements all initialised to ""
- ▪ QueueHead and QueueTail initialised with –1, NumberItems initialised to 0
- Example program code
- Java public static String[] Queue = new String[100];
- public static Integer QueueHead;
- public static Integer QueueTail;
- public static Integer NumberItems;
- for(int x = 0; x < 100; x++){
- Queue[x] = "";
- }
- QueueHead = -1;
- QueueTail = -1;

Solution 2

(b) Worked steps

Enqueue has three jobs in a fixed order: refuse if full, store, then update the bookkeeping.

```
FUNCTION Enqueue(TheData : STRING) RETURNS BOOLEAN
    IF QueueTail >= 99 THEN
        RETURN FALSE
    ENDIF
    Queue[QueueTail + 1] <- TheData
    QueueTail <- QueueTail + 1
    NumberItems <- NumberItems + 1
    IF QueueHead = -1 THEN
        QueueHead <- 0
    ENDIF
    RETURN TRUE
ENDFUNCTION
```

Solution 2

The five marks map to five separate things, so check for each one:

1. Function header with one string parameter, the full test, and `RETURN FALSE`.
2. Storing the parameter at `QueueTail + 1`.
3. Incrementing **both** `QueueTail` and `NumberItems`.
4. The **first-element case**: when the queue was empty, `QueueHead` is still -1 and must become 0, or the first item can never be dequeued.
5. `RETURN TRUE` on every path where the item was inserted.

Solution 2

That fourth mark is the one most often lost. It is the only place where the head pointer moves during an insertion, and it is easy to overlook because the queue works for the second item onward without it. **Mark scheme**

- 1 mark each
- ▪ Function header (and end where appropriate) taking one (string) parameter and checking full and returning FALSE
- ▪ (otherwise) Storing parameter in QueueTail + 1
- ▪ Incrementing QueueTail and NumberItems
- ▪ (Dealing with first element) If QueueHead = –1 store 0 in QueueHead / increment QueueHead
- ▪ Return TRUE in all cases when inserted
- Example program code.
- Java public static Boolean Enqueue(String TheData){ if(QueueHead == -1){
- Queue[0] = TheData;
- QueueHead = 0;
- QueueTail = 0;
- NumberItems = 1;

Solution 2

(c) Worked steps

Dequeue is the mirror image, and the empty test is what makes it safe.

```
FUNCTION Dequeue() RETURNS STRING
    DECLARE ReturnValue : STRING
    IF NumberItems = 0 THEN
        RETURN "False"
    ENDIF
    ReturnValue <- Queue[QueueHead]
    QueueHead <- QueueHead + 1
    NumberItems <- NumberItems - 1
    RETURN ReturnValue
ENDFUNCTION
```

Solution 2

- The empty test may be written as `NumberItems = 0` or as `QueueHead > QueueTail`; both are accepted, and both are true of an empty queue.
- **Read the element before moving the pointer.** Incrementing `QueueHead` first returns the wrong item, and the mark for returning `Queue[QueueHead]` goes with it.
- `QueueHead` goes **up** and `NumberItems` goes **down** — the two move in opposite directions, which is worth checking before you leave the question.

Solution 2

The string `"False"` here is a sentinel value, not a Boolean: the function's return type is STRING, so it must return one. Returning the Boolean `FALSE` is a type error in most of the languages this paper allows. **Mark scheme**

- 1 mark each
- ▪ Function header (and end), checking if Queue is empty (NumberItems = 0 or QueueHead > QueueTail) and returning "False"
- ▪ (otherwise) returning Queue[QueueHead]
- ▪ Incrementing QueueHead, decrementing NumberItems
- Example program code
- Java public static String Dequeue(){
- String ReturnValue;
- if(NumberItems == 0){ return "False";
- }else{
- ReturnValue = Queue[QueueHead];
- QueueHead++;
- NumberItems--;

Solution 2

(d) Worked steps

`ReadData()` is a procedure — it returns nothing, it fills the queue as a side effect. Five marks, five separate things, so build it in five visible pieces.

```
PROCEDURE ReadData()
    DECLARE Line : STRING
    DECLARE Ok   : BOOLEAN
    TRY
        OPENFILE "BinaryData.txt" FOR READ
        WHILE NOT EOF("BinaryData.txt")
            READFILE "BinaryData.txt", Line
            Ok <- Enqueue(Line)
        ENDWHILE
        CLOSEFILE "BinaryData.txt"
    EXCEPT
        OUTPUT "The file could not be read"
    ENDTRY
ENDPROCEDURE
```

Solution 2

1. Procedure header and end.
2. The file **opened and closed**, with the close in the right place — after the loop, inside the successful path.
3. A loop **until end of file**, one line per pass.
4. Each value **read into a variable** inside that loop.
5. `Enqueue()` called with the value, **and its return value stored or used**.

Solution 2

Two details carry the marks people miss. `Enqueue()` is a **function**: it returns FALSE when the queue is full, so assigning its result — and ideally stopping when it is FALSE — is what the fifth bullet asks for. And the exception handling must wrap **all** the file access, not just the `OPENFILE`; a file that vanishes mid-read fails on the `READFILE`.

Mark scheme

- 1 mark each to max 5
- ▪ Procedure header (and end where appropriate)
- ▪ Opening the file and closing the file (in appropriate place)
- ▪ Looping until EOF // looping through each line …
- ▪ … read in each value …
- ▪ … calling Enqueue() with read in value and store/use return value
- ▪ Exception handling with try except and appropriate output with all file access within try
- Example program code
- Java public static void ReadData(){
- Boolean ReturnValue;
- Boolean FinishLoop = false;
- try{

Solution 2

(e) Worked steps

Run-length encoding: walk the queue, count how many times each value repeats, and append the value followed by its count.

```
PROCEDURE Compress()
    DECLARE Current, NextValue : STRING
    DECLARE Count : INTEGER
    NewString <- ""
    Current <- Dequeue()
    WHILE Current <> "False"
        Count <- 1
        NextValue <- Dequeue()
        WHILE NextValue = Current
            Count <- Count + 1
            NextValue <- Dequeue()
        ENDWHILE
        NewString <- NewString & Current & NUM_TO_STR(Count)
        Current <- NextValue
    ENDWHILE
ENDPROCEDURE
```

Solution 2

Six marks, and they map to the six things this has to do:

1. Procedure header and end, with the finished string kept in the **global** variable the next part will output.
2. `Dequeue()` called and its return value **stored**.
3. Called **repeatedly until it returns `"False"`** — that sentinel is how the empty queue announces itself, which is why `Dequeue` returns a STRING rather than a Boolean.
4. Each new value **compared with the previous** one.
5. A **counter** of consecutive matches, reset to 1 at the start of each run.
6. The digit and its count **appended** to the string — `NewString <- NewString & …`, never `NewString <- …`, or every run overwrites the last.

Solution 2

The subtle part is the value that ends a run: it has already been dequeued, so it must become the `Current` of the next run rather than being read again. Dequeuing it twice loses one item from every run boundary. **Mark scheme**

- 1 mark each
- ▪ Procedure header (and end where appropriate), storing final string compressed data in global variable
- ▪ Call Dequeue() and storing return value ...
- ▪ ... repeatedly until the return value == "False"
- ▪ Comparing new value with previous ...
- ▪ ... maintaining counter of number of occurrences ...
- ▪ ... appending digit and number of occurrences to a string without overriding
- Example program code
- Java public static void Compress(){
- String First = Dequeue();
- String NewLine = "";
- Integer Count;

Solution 2

(f)(i) Worked steps

The main program wires the two procedures together in order.

```
NewString <- ""
// initialise Queue, QueueHead, QueueTail and NumberItems as in part (a)
CALL ReadData()
CALL Compress()
OUTPUT NewString
```

Solution 2

Two marks: **calling `ReadData()` and then `Compress()`**, and **outputting the compressed string**.

Solution 2

The order is not arbitrary — `Compress()` empties the queue that `ReadData()` fills, so reversing them compresses nothing and outputs an empty string. And the globals must be initialised before either runs, or `QueueHead` still holds whatever the last run left in it. **Mark scheme**

- 1 mark each
- ▪ Calling ReadData() then Compress()
- ▪ Outputting compressed string
- Example program code
- Java public static void main(String args[]){
- NewString = "";
- for(int x = 0; x < 100; x++){
- Queue[x] = "";
- }
- QueueHead = -1;
- QueueTail = -1;
- NumberItems = 0;

Solution 2

(f)(ii)

Mark scheme

- 1 mark for screenshot showing output
- Example

Question 5

9618/21 J25 ▪ Q5 ▪ 7 marks

Stacks and queues are both abstract data types.
A stack uses a top-of-stack pointer to indicate the location of the last item added to the stack.
A queue uses two pointers:

- a front pointer to indicate the location of the next item to be removed from the queue
- a rear pointer to indicate the location of the next item to be added to the queue.

A queue can be used to reverse the items stored on a stack.
For example, if a stack contains six items:

Question 5

Initial state of the stack:
top-of-stack pointer $\longrightarrow$

item 6
item 5
item 4
item 3
item 2
item 1

Question 5

Final state of the stack when the items have been reversed:
top-of-stack pointer $\longrightarrow$

item 1
item 2
item 3
item 4
item 5
item 6

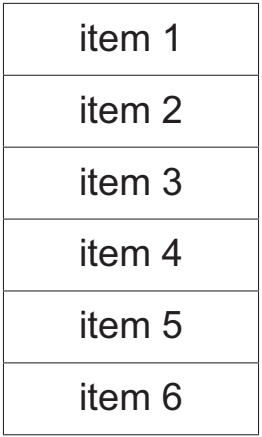
Question 5

Describe how the queue could be used to reverse the items that are currently stored on the stack.
Your description must include how the pointers are used in both the stack and queue.
Assume:

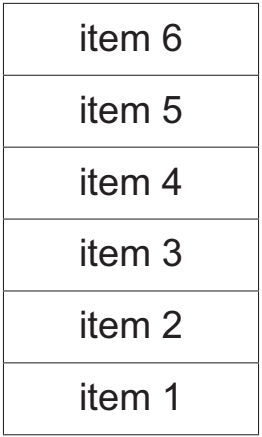
- The stack initially contains an unknown number of items.
- The queue can store all the items currently stored on the stack.
- The queue is initially empty.

[7]

Question 5



Question 5



Solution 5

- Chars array final values
- 'D'
- 'H'
- 'R'
- 'T'
- Mark as follows:
- MP1, MP2, MP3, MP4, MP5
- 1 mark per enclosure

Solution 5

- 6
- MP1
- MP2
- MP3
- MP4
- MP6
- MP5
- 5
- MP1

Solution 5

- Pop an item off the stack
- MP2
- Update Stack pointer
- MP3
- Add it to the queue
- MP4
- Update Rear pointer
- MP5
- Repeat until the stack is empty

Solution 5

- MP6
- Remove an item from the queue
- MP7
- Update Front pointer
- MP8
- Push it onto the stack
- MP9
- Update the Stack pointer
- MP10

Solution 5

- Repeat until the queue is empty
- Max 7

Question 2

9618/23 J25 ▪ Q2 ▪ 11 marks

A programmer uses a number of Abstract Data Types (ADT) in the programs that she is developing.

(a)(i) Complete the answer column in the following table:

	Answer
the memory location of the value that has been on the stack for the longest time	
the number of consecutive `push` operations that will result in the `TopOfStack` variable containing 409	

Question 2

[2]

(a)(ii) The diagram shows the current state of the stack.

Stack

Memory location	Value	
409		
408		
407	'A'	← `TopOfStack`
406	'C'	
405	'F'	
404	'K'	
403	'B'	
402	'S'	
401	'R'	
400	'D'	

Question 2

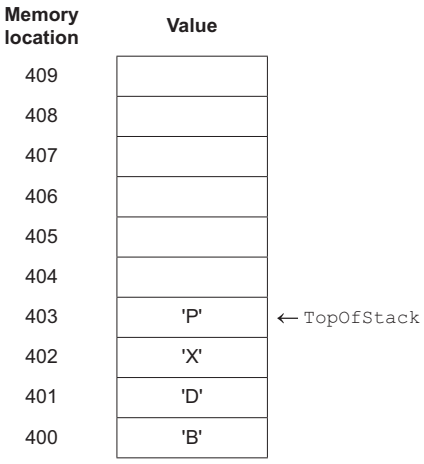
The following sequence of operations are performed:
PUSH 'T' POP POP PUSH 'Z' PUSH 'X' POP PUSH 'Y'
Complete the following diagram to show the state of the stack after the operations have been performed.
Stack

Memory location	Value
409	
408	
407	
406	
405	
404	
403	
402	
401	
400	

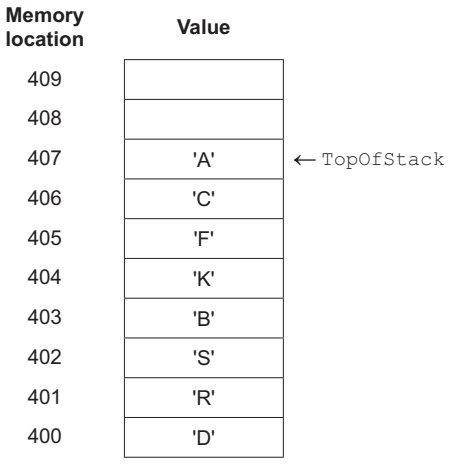
Question 2

[3]

Question 2


Memory location
Value
409
408
407
406
405
404
403
'P'
← TopOfStack
402
'X'
401
'D'
400
'B'

Question 2


Memory location
Value
409
408
407
'A'
← TopOfStack
406
'C'
405
'F'
404
'K'
403
'B'
402
'S'
401
'R'
400
'D'

Question 2

	Answer
the memory location of the value that has been on the stack for the longest time	
the number of consecutive **push** operations that will result in the `TopOfStack` variable containing 409	

[2]

Question 2

9618/23 J25 ▪ Q2 ▪ 11 marks

A programmer uses a number of Abstract Data Types (ADT) in the programs that she is developing.

(b)(i) The array is represented in the diagram.

A pointer value of -1 indicates the end of the linked list.

Complete the pointer field to produce a linked list that is in alphabetical order.

Index	Data	Pointer
0	"Neptune"	
1	"Jupiter"	
2	"Saturn"	
3	"Earth"	
4	"Mercury"	
5	"Uranus"	

Question 2

[3]

(b)(ii) The variable `StartPointer` contains the index of the first item in the linked list.

State the value of the variable `StartPointer` **[1]**

Question 2

9618/23 J25 ▪ Q2 ▪ 11 marks

A programmer uses a number of Abstract Data Types (ADT) in the programs that she is developing.

(c) A third program is also being created to manage a queue.

Describe **two** features of a queue.

Feature one

Feature two **[2]**

Solution 2

(a)(i)

Mark scheme

- MP1
- 400
- MP2
- 6

Solution 2

(a)(ii) Mark scheme

- Stack
- Memory location
- Value
- 409
- 408
- 'Y'
- TopOfStack
- 407

Solution 2

- 'Z'
- 406
- 'C'
- 405
- 'F'
- 404
- 'K'
- 403
- 'B'
- 402

Solution 2

- 'S'
- 401
- 'R'
- 400
- 'D'
- MP1
- TopOfStack pointing to 'Y' and value 'Y' in location 408
- MP2
- Values 'C' and 'Z' in 406 and 407
- MP3

Solution 2

- Values 'D' to 'F' unchanged in 400 to 405 and 409 is empty
- 3

Solution 2

(b)(i) Mark scheme

- Index
- Data
- Pointer
- 0
- "Neptune"
- 2
- MP3
- 1

Solution 2

- "Jupiter"
- 4
- 2
- "Saturn"
- 5
- 3
- "Earth"
- 1
- MP1
- 4

Solution 2

- "Mercury"
- 0
- 5
- "Uranus"
- -1
- MP2
- MP1
- Earth pointer 1
- MP2
- Uranus pointer -1

Solution 2

- MP3
- Other four correct pointers

Solution 2

(b)(ii)

Mark scheme

- 3

Solution 2

(c) Mark scheme

- MP1
- First value added to queue is the first value removed // FIFO // LILO
- MP2
- Two pointers needed to indicate the start and end of the queue
- MP3
- May behave as a circular queue
- MP4
- Manipulated using enqueue()/ dequeue() operations // by description

Solution 2

- Max 2

Question 4

9618/31 J25 ▪ Q4 ▪ 7 marks

A linked list of nodes is used to store an ordered list of integers. Each node consists of the data, a left pointer and a right pointer, for example:

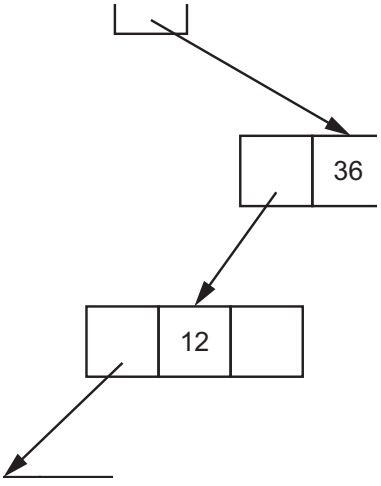
Question 4

(a) The linked list will be organised as a binary tree.
-1 is used to represent a null pointer.
Complete the binary tree, including null pointers, to show how the data will be organised after the following integers have been added:
6, 15, 41, 66 **[4]**

(b) Describe what is meant by recursion. **[2]**

(c) A binary tree is a suitable Abstract Data Type (ADT) that a designer can implement using recursive algorithms.
Identify **one other** ADT that a designer can implement using recursive algorithms. **[1]**

Question 4



Solution 4

(a)

Mark scheme

- One mark per mark point
- MP1 Any two nodes added correctly with correct data (6, 15, 41, 66) and arrows
- MP2 Remaining two nodes added with correct data (6, 15, 41, 66) and all nodes with connecting arrows starting from pointer boxes
- MP3 Correct null pointers (-1) added throughout
- MP4 ... with no entries in other pointer boxes and all nodes correctly positioned and connected

Solution 4

(b)

Mark scheme

One mark per mark point

- MP1 A technique used to solve problems using a function/procedure/subroutine that calls itself (general case)
- MP2 ... until the terminating condition / base case is achieved, when no further recursive calls are made

Solution 4

(c)

Mark scheme

One mark

- Stack

Question 3

9618/22 N24 ▪ Q3 ▪ 7 marks

A program uses a stack to hold up to 60 numeric values. The stack is implemented using two integer variables and a 1D array.
The array is declared in pseudocode as shown:
DECLARE ThisStack : ARRAY[1:60] OF REAL
The stack operates as follows:

- Global variable `SP` acts as a stack pointer that points to the next available stack location. The value of `SP` represents an array index.
- Global variable `OnStack` represents the number of values currently on the stack.
- The stack grows upwards from array element index 1.

(a)(i) Give the initial values that should be assigned to the two variables. **[1]**

(a)(ii) Explain why it is not necessary to initialise the array elements before the stack is used. **[2]**

Question 3

9618/22 N24 ▪ Q3 ▪ 7 marks

A program uses a stack to hold up to 60 numeric values. The stack is implemented using two integer variables and a 1D array.
The array is declared in pseudocode as shown:
DECLARE ThisStack : ARRAY[1:60] OF REAL
The stack operates as follows:

- Global variable `SP` acts as a stack pointer that points to the next available stack location. The value of `SP` represents an array index.
- Global variable `OnStack` represents the number of values currently on the stack.
- The stack grows upwards from array element index 1.

Question 3

(b) A function to add a value to `ThisStack` is expressed in pseudocode as shown. The function will return a value to indicate whether the operation was successful or not. Complete the pseudocode by filling in the gaps.

```
FUNCTION Push(ThisValue : REAL) RETURNS BOOLEAN
DECLARE ReturnValue : BOOLEAN
IF ..... THEN
RETURN ..... // stack is already full
ENDIF
SP <- .....
OnStack <- OnStack + 1
RETURN TRUE
ENDFUNCTION
```

[4]

Solution 3

(a)(i) Worked steps

Read the design note carefully: `SP` points to the **next available location**, not to the last item pushed. That one word decides both values.

- `SP` = **1** — the next free slot is the first element of `ARRAY[1:60]`.
- `OnStack` = **0** — nothing has been pushed yet.

One mark for both together, so check them against each other: if `SP` started at 0 it would point outside the array, and if `OnStack` started at 1 the stack would claim an item that was never pushed.

Solution 3

△ This is the opposite convention from the papers where `SP` holds the index of the *last* item and starts at 0. Always take the convention from the question in front of you — the two differ by one everywhere.

Mark scheme

- SP: 1
- OnStack: 0
- One mark for both correct values

Solution 3

(a)(ii) Worked steps

Two points, one mark each, and the second is the one that completes the argument.

- An element that has never been written **cannot be read**: the stack can only pop a location that is below `SP`, and nothing is below `SP` until something is pushed.
- A push **writes over** whatever was in the location before it, so any initial value would be destroyed before it could ever be seen.

Solution 3

So initialising the array would be work whose result is guaranteed never to be used. The pointer, not the array contents, is what defines which elements are part of the stack.

Mark scheme

- MP1 Unused values cannot be popped / taken off the stack // initialised values would never be used / unused elements cannot be accessed
- MP2 ... until a value has first been pushed / written // overwrites previous value

Solution 3

(b) Worked steps

Push refuses when the stack is full, otherwise stores, moves the pointer and counts.

```
FUNCTION Push(ThisValue : REAL) RETURNS BOOLEAN
    IF OnStack = 60 THEN
        RETURN FALSE
    ENDIF
    ThisStack[SP] <- ThisValue
    SP      <- SP + 1
    OnStack <- OnStack + 1
    RETURN TRUE
ENDFUNCTION
```

Solution 3

One mark per gap:

1. The **full test** — `OnStack = 60`, or equivalently `SP > 60`, with `RETURN FALSE`.
2. Storing the parameter at `ThisStack[SP]` — with this convention `SP` is already the free slot, so there is no `+ 1` here.
3. **Incrementing both** `SP` and `OnStack`.
4. `RETURN TRUE` on the path where the value was stored.

Solution 3

The check that catches convention errors: after the first push, `SP` is 2 and `OnStack` is 1, and `ThisStack[1]` holds the value. If your code leaves `SP` at 1 the next push overwrites it; if it stores at `ThisStack[SP + 1]` the first element is never used. **Mark scheme**

- Example solution:
- FUNCTION Push(ThisValue : REAL) RETURNS BOOLEAN
- DECLARE ReturnValue : BOOLEAN
- IF OnStack = 60 / >59 // SP = 61 / SP > 60 //
- SP outside the range 1 to 60 THEN
- RETURN FALSE // Stack is already full
- ENDIF
- ThisStack[SP] <- ThisValue
- SP <- SP + 1
- OnStack <- OnStack + 1
- RETURN TRUE
- ENDFUNCTION1

Question 3

9618/23 N24 ▪ Q3 ▪ 8 marks

3 The implementation of a linked list uses an integer variable and a 1D array `List` of type `Node`.

Record type `Node` is declared in pseudocode as follows:

```
TYPE Node
   DECLARE Data : STRING
   DECLARE Pointer : INTEGER
ENDTYPE
```

The array `List` is declared in pseudocode as follows:

```
DECLARE List : ARRAY[1:200] OF Node
```

The linked list will operate as follows:

- Integer variable `HeadPointer` will store the array index for the first node in the linked list.
- The `Pointer` field of a node contains the index value of the next array element (the next node) in the linked list.

Question 3

- The value 0 is used as a null pointer.

(a) (i) State why the value 0 has been selected as the null pointer.

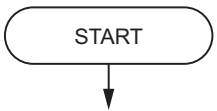
Question 3

(ii) Give the range of valid values that could be assigned to variable `HeadPointer`.

Question 3

(b) The array `List` will be initialised so that each node points to the following node. The last node will contain a null pointer.

Complete the program flowchart to represent the algorithm for this operation.

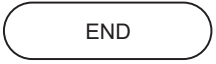


Question 3

Question 3

Question 3

Question 3



[4]

(c) An algorithm outputs the `Data` field from all nodes in the array `List`. The order the `Data` is output should be the same order it is stored in the linked list.

Question 3

output should be the same order it is stored in the linked list.

Question 3

Describe the algorithm in **four** steps.

Do **not** use pseudocode statements in your answer.

Question 3

Question 3

Question 3

Question 3

[4]

Solution 3

(a)(i)

Worked steps

- **0 is not a valid index** into the array — there is no element 0 — so it can never be mistaken for a real node.

That is exactly what a null pointer has to be: a value the pointer field can hold that could not possibly point at anything. If the array were indexed from 0, some other value (-1 is the usual choice) would have to be used instead.

Mark scheme

- 0 is not a valid array / List index value // No 0th element in the array
- 1

Solution 3

(a)(ii)

Worked steps

- The pointer field holds **0 to 200**.

It is 0 for the null pointer, and 1 to 200 for the 200 array elements. Reading the range off this way — one value per thing the field can mean — is more reliable than trying to remember a convention.

Mark scheme

- 0 to 200
- 1
- 9618/23
- October/November
- 2024

Solution 3

(b) Worked steps

Initialising the **free list** means linking every element to the next one, so an unused node can be found and reclaimed.

```
Index <- 1
FOR Index <- 1 TO 199
    List[Index].Pointer <- Index + 1
NEXT Index
List[200].Pointer <- 0
```

Solution 3

Four marks:

1. A variable used as the **array index**, starting at 1 — the first element.
2. A loop of **199 iterations** (or 200, with care at the end).
3. `Index + 1` **assigned to each pointer field** inside the loop.
4. **0 assigned to the 200th pointer field**.

Solution 3

That last line is the whole point of the exercise and it sits **outside** the loop. Element 200 has no successor, so its pointer is the null value from part (a). A loop that runs all 200 times sets `List[200].Pointer` to 201 — an index that does not exist — and the free list then runs off the end of the array. **Mark scheme**

- Mark as follows:
- MP1 Use of variable as array index and initialisation to 1 / to first element of
- array
- MP2 Loop for 199 / 200 iterations
- MP3 Assign Index+1 to each pointer field in a loop
- MP4 Assign 0 to 200th pointer field
- 4
- 9618/23
- October/November

Solution 3

(c) Worked steps

Traversal is the standard three-step pattern for any linked list.

```
ThisPointer <- HeadPointer
WHILE ThisPointer <> 0
    OUTPUT List[ThisPointer].Data
    ThisPointer <- List[ThisPointer].Pointer
ENDWHILE
```

Solution 3

One mark per step:

1. **Start at the head**: assign `HeadPointer` to a working pointer.
2. **Loop until the pointer is 0** — the null value, meaning the end of the list.
3. **Output the data** of the element at the current index.
4. **Follow the link**: assign that element's pointer field to the working pointer.

The order of the last two is what makes it work. Reading the data **before** moving on means the final node is printed; advancing first skips the first node and runs one element past the end.

Solution 3

Note the list is traversed by **following pointers**, not by counting from 1 to 200. The array is only storage; the order of the list is held entirely in the pointer fields, which is why the items need not be stored in order at all. **Mark scheme**

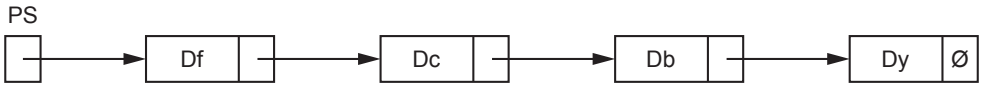
- One mark per step:
- MP1 Assign HeadPointer to ThisPointer / Current Pointer // Identify
- first node using headpointer
- MP2 Loop the following until ThisPointer value is 0 (zero) / null pointer
- reached
- MP3 … Output data (field) from array element at index ThisPointer //
- Output data (field) of the current node / array element
- MP4 … Assign pointer field from current array element / index / to
- ThisPointer / Current Pointer // Assign pointer field from current
- node to ThisPointer / Current Pointer

Question 3

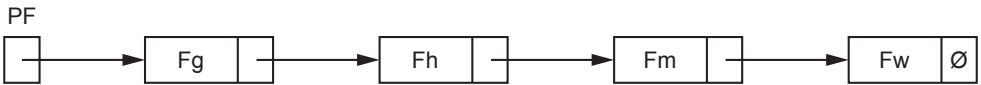
9618/21 J24 ▪ Q3 ▪ 11 marks

3 The diagram shows an Abstract Data Type (ADT) representation of a linked list after data items have been added.

- PS is the start pointer.
- PF is the free list pointer.
- Labels Df, Dc, Db and Dy represent the data items of nodes in the list.
- Labels Fg, Fh, Fm and Fw represent the data items of nodes in the free list.
- The symbol Ø represents a null pointer.



Question 3



(a) Describe the linked list immediately after initialisation, before **any** data items are added.

Question 3

(b) A program will be written to include a linked list to store alphanumeric user IDs.

The design uses two variables and two 1D arrays to implement the linked list.
Each array element contains data of a single data type and **not** a record.

The statements below describe the design.

Complete the statements.

Question 3

[5]

Solution 3

(a) Mark scheme

- One mark per point:
- 1
- The PS contains a null pointer
- 2
- The PF points to the first element on the free list
- 3
- All the nodes are on the free list
- 3

Solution 3

(b) Worked steps

The implementation needs **two integer variables and two 1D arrays**, and each mark is for saying what one of them *is* and what it is *for*.

The two variables

- Both are of type **INTEGER**.
- Both are used as **pointers — indexes into the arrays**.
- Their values give the **first element of each list**: `PS` the start of the data list, `PF` the start of the free list.

The first array

- Of type **STRING**.
- Holds the **data items** — the user IDs.

Solution 3

The second array

- Of type **INTEGER**.
- Holds the **pointers**: for each element, the index of the **next** item in whichever list it belongs to.

Give the type **and** the use for every one. A list of four declarations with no explanation answers only half of what is asked.

The idea worth stating explicitly is that **one pair of arrays holds two lists at once**. Every element is on exactly one of them — the data list reached from `PS`, or the free list reached from `PF` — and adding an item means unlinking a node from the free list and linking it into the data list. That is why both need a start pointer and why the pointer array serves both. **Mark scheme**

Solution 3

- Max 2 marks for 'Variables':
- The two variables will be of type Integer
- The two variables will be used as pointers / indexes to the
- arrays.
- The values stored in the two variables will indicate the first
- element in each list
- The first 1D array will be of type String
- The first 1D array will be used to store the values // data items
- // User IDs
- The second 1D array will be of type Integer

Solution 3

- The second 1D array will be used to store the pointers // point
- to next item
- Mark as follows:
- One mark for each of the first three rows
- One mark for both Array 1 rows
- One mark for both Array 2 rows
- 5
- 9618/21
- May/June 2024
- 4

Solution 3

- Example:
- FUNCTION Check() RETURNS STRING
- DECLARE Odd, Even, Index : INTEGER
- Odd <- 0
- Even <- 0
- FOR Index <- 1 TO 100
- IF Index MOD 2 = 0 THEN
- Even <- Even + Data[Index]
- ELSE
- Odd <- Odd + Data[Index]

Solution 3

- ENDIF
- NEXT Index
- ENDFUNCTION
- Mark as follows:
- 1. Function heading, ending and return type
- 2. Declare local variables Odd, Even and Index as integers
- 3. Initialise Odd and Even
- 4. Loop for 100 iterations // through array
- 5. Sum Odd and Even element values in a loop
- 6. Compare Odd and Even after the loop and Return appropriate string

Solution 3

- 6
- 9618/21
- May/June 2024