Computer Science (H046, H446)

Queues / Stacks / Lists
Mr. Montgomery

32/45

Please note that you may see slight differences between this paper and the original.

Candidates answer on the Question paper.

OCR supplied materials:
Additional resources may be supplied with this paper.

Other materials required:
- Pencil
- Ruler (cm/mm)

Duration: Not set

Candidate forename	Daichen	Candidate surname	

Centre number						Candidate number				

INSTRUCTIONS TO CANDIDATES

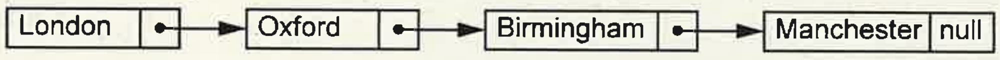
- Write your name, centre number and candidate number in the boxes above. Please write clearly and in capital letters.
- Use black ink. HB pencil may be used for graphs and diagrams only.
- Answer **all** the questions, unless your teacher tells you otherwise.
- Read each question carefully. Make sure you know what you have to do before starting your answer.
- Where space is provided below the question, please write your answer there.
- You may use additional paper, or a specific Answer sheet if one is provided, but you must clearly show your candidate number, centre number and question number(s).

INFORMATION FOR CANDIDATES

- The quality of written communication is assessed in questions marked with either a pencil or an asterisk. In History and Geography a *Quality of extended response* question is marked with an asterisk, while a pencil is used for questions in which *Spelling, punctuation and grammar and the use of specialist terminology* is assessed.
- The number of marks is given in brackets **[ ]** at the end of each question or part question.
- The total number of marks for this paper is **45**.
- The total number of marks may take into account some 'either/or' question choices.

1 A coach company offers tours of the UK.

A linked list stores the names of cities on a coach tour in the order they are visited.



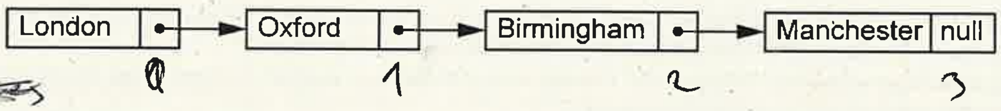
(i) Describe what is meant by the term 'linked list'.

A List which has a poiter pointing to the start of the list, ✓ a pointer pointing to the next spot in the list that is free and can have an item added to the spot. Each ~~item~~ spot in the list is called a node, each node containing the data you added to the list, and a pointer ✓ which points to the next node in the list's order (whether the order is alphabetical, numerical, etc.) **[3]**

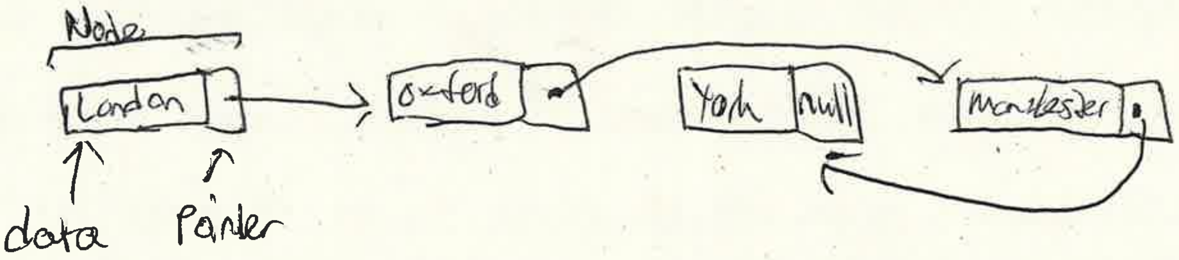
2

(ii) The tour is amended. The new itinerary is: London, Oxford, Manchester then York. Explain how Birmingham is removed from the linked list and how York is added. You may use the diagram below to illustrate your answer.

L, O, ~~B~~, M
Y



To remove Birmingham, we don't have to literally get rid of it, what we can do instead is, because Oxford is pointing to Birmingham, which is pointing to Manchester, Change Oxford's pointer to ~~point~~ straight to Manchester, skipping Birminham ~~entirely~~, ~~making~~ making it so Birmingham is never referenced or passed through. Then, we can change Manchester's pointer, since York is added, Manchester is no longer the last item, to point to Birmingham's ~~node~~ node. Now change the data in Birmingham's node to say "York". Then change the changed ~~next~~ ~~established~~ node's pointer to equal "null" since York is the last data in the list now. Result should look like this: [4]



4

2(a) A program stores a queue of mathematical questions to be asked to a user. The questions are asked in the order they are added. Once a question has been asked it cannot be asked again. New questions are continually added to the end of the queue.

The program will use a non-circular queue, `questions`, (implemented using an array) to store the questions.
The pointer, `head`, stores the index of the first element in the queue.
The pointer, `tail`, stores the index of the last element in the queue.

Describe why a queue is a suitable structure for this program.

~~Queue~~ Queues ~~use~~ are "First in first out" (FIFO) ✓ data structures, meaning the ~~whichever~~ questions ~~is~~ answered by the program are answered in the order of which the questions are inputted ✓. This is suitable because, it wouldn't ~~only~~ make sense if the data structure was not FIFO, which would result in the answers given by the program to be all mixed up with their corresponding question, for example if you didn't use a queue, you might get an answer that doesn't match the answer of one question, but instead is the answer to another question inputted at a different time. Whereas with a queue, the answer given will be the answer to the question that was input ~~in~~ in the index of the answer [3]

2

(b) Fig. 4.1 shows an example of the data in the queue. `head` is currently 0, `tail` is currently 4.

“2*3”	“1+4”	“3–1”	“10/2”	“3+6”			

Fig. 4.1

(i) Show the contents of the queue shown in Fig. 4.1, after the following code is run.

```
add("6+1")
```

"2*3"	"1+4"	"3-1"	"10/2"	"3+6"	"6+1"		

[2]

(ii) State the values stored in `head` and `tail` after the code in **part (i)** has run.

`head` ~~[illegible]~~ 0

`tail` ~~[illegible]~~ 5

[2]

(c) Complete the following algorithm, to remove, and output, the first element in the queue.

```
procedure remove()   <- non circular

    removedItem = questions[head]
    head +=1
    ~~print(removedItem)~~
    ~~length(questions) -= 1~~
    print(removedItem)

endprocedure
```

need to check if queue is empty

3 [4]

(d) Complete the following algorithm, to ask the user to input a new question and then either add it to the queue, or report that the queue is full.

```
procedure add()
    maxElements = 10
    if length(questions) < maxElements then
            inputQ = input("enter a question: ")
            questions[tail+1] = inputQ
            tail += 1
    else
          print("queue is already full")
    endif

endprocedure
```

[4]

3(a) A programmer is developing an ordering system for a fast food restaurant. When a member of staff inputs an order, it is added to a linked list for completion by the chefs.

Explain why a linked list is being used for the ordering system.

A linked list has pointers ~~at~~ in each node which allows the list to be in a desired order regardless of how the items are inputted ~~into~~ into the list. This makes it so if a chef completes an order, they can easily ~~see~~ see what the next order is once the current one has been removed since the current order will point to what order needs to be done next.

[2]

(b) Each element in a linked list has:

- a pointer, `nodeNo`, which gives the number of that node
- the order number, `orderNo`
- a pointer, `next`, that points to the next node in the list

Fig. 2.1 shows the current contents of the linked list, `orders`.

index

nodeNo	orderNo	next
0	154	1
1	157	2
2	155	3
3	156	∅

Fig. 2.1

∅ represents a null pointer.

(i) Order 158 has been made, and needs adding to the end of the linked list.

Add the order, 158, to the linked list as shown in Fig. 2.1. Show the contents of the linked list in the following table.

nodeNo	orderNo	next
0	154	1
1	157	2
2	155	3
3	156	4
4	158	∅

2

[2]

(ii) Order 159 has been made. This order has a high priority and needs to be the second order in the linked list.

Add the order, 159, to the original linked list as shown in Fig. 2.1. Show the contents of the linked list in the following table.

nodeNo	orderNo	next
0	154	1
1	159	2
2	157	3
3	155	4
4	156	∅

✗ [3]

(c) The linked list is implemented using a 2D array, `theOrders`:

- Row 0 stores `orderNo`
- Row 1 stores `next`

The data now stored in `theOrders` is shown in Fig. 2.2.

184	186	185	187
1	2	3	

Fig. 2.2

`theOrders [1,0]` would return `1`

The following algorithm is written:

```
procedure x()
    finished = false
    count = 0
    while NOT(finished)
        if theOrders[1,count] == null then
            finished = true
        else
            output = theOrders[0,count]
            print(output)
            count = theOrders[1,count]
        endif
    endwhile
    output = theOrders[0,count]
    print(output)

endprocedure
```

(i) Outline why nodeNo does not need to be stored in the array.

~~[illegible]~~ nodeNo is just the index of ~~the~~ a node. The Index of each node is it's node No, meaning you can get access to the node No simply by finding ~~its~~ the node's position in the array. Storing it in array just wastes memory. [1]

(ii) Complete the trace table for procedure x, for the data shown in Fig. 2.2.

finished	count	output
false	0	184
false	1	166
false	2	165
false	3	187
~~[illegible]~~ true	~~null~~ 3	187

X

[3]

(iii) Describe the purpose of procedure x.

To print all the current orders in ~~order of first to be done~~ Chronological order, to last, then output the very last, most recently added order.

1 [2]

(iv) A new order, 190, is to be added to `theOrders`. It needs to be the third element in the list.

The current contents of the array are repeated here for reference:

0	1	2	3	4	5
184	186	185	187		
1	2	3			

↑ 3rd element, 2nd index

Describe how the new order, 190, can be added to the array, so the linked list is read in the correct order, without rearranging the array elements.

First, add the order 190 to the next free spot in the list, which is index 4, the node right after order 187's node. Now change the ~~pointer~~ value of index 1's pointer to point to index 4 (index 1 is order number 186's node). This way, the orders will go from 184 → 186 → 190 then → 185 → 187 so that 190 is third ~~[illegible]~~ to be read. Now, change order 190's node's pointer's value to point to index 2, (order 185) so that the ~~list~~ rest of the orders are read in the same order as before.

[4]

4(a) Stacks and queues are both data structures.

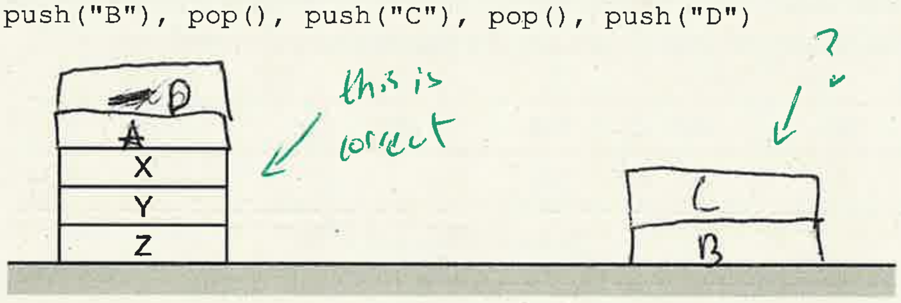
State which of a stack or queue would be considered as a 'First In First Out' data structure.

Queue [1]

(b) A stack is shown in Fig. 4.1 before a set of operations are carried out on it.

Draw what the stack shown in Fig. 4.1 would look like after the following operations:

```
push("A"), push("B"), pop(), push("C"), pop(), push("D")
```



Before operations **After operations**

Fig. 4.1

[2]

(c) Fig. 4.2 shows a stack in two states: State One and State Two.

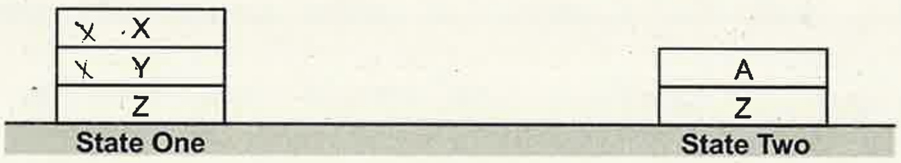


Fig. 4.2

List the operations needed to get the stack from State One to State Two.

remove("x"), remove("y"), pop("Z"), push("A"), pop("A") ✗

[3]

END OF QUESTION PAPER