1.4 Data Types, Data Structures and Boolean Algebra: 1-hour Pack

OCR H446 · Calculators not allowed · about 60 minutes

Total Marks

Name: _____ Date: ____ / ____ / _____

Answer ALL questions. Show all your working.

1 A programmer is implementing a hash table.

(a) Explain what is meant by a hash table, and describe the role played by a hash function when storing and retrieving data in one. **(3)**

(b) A hash table has 7 slots, indexed 0 to 6, and uses the hash function h(key) = key MOD 7. State the index that each of the following keys initially hashes to: 15, 22, 9, 30, 16. Identify which of these keys collide with an earlier key (hash to a slot already used by a key inserted before it), assuming they are inserted in the order given. **(3)**

(c) Collisions are resolved using linear probing: if a key's home slot is already occupied, the next slot is checked (wrapping from slot 6 back to slot 0), and so on, until an empty slot is found. Using this method, insert the five keys from part (b), in the order given, into the hash table. State the final contents of all 7 slots. **(4)**

(d) Explain one disadvantage of using linear probing to resolve collisions when a hash table becomes heavily loaded (close to full). **(2)**

(Total for Question 1 is 12 marks)

© Revision Library | revisionlibrary.co.uk

2 A program implements a stack using the following pseudocode:

```
DECLARE stack : ARRAY[0:9] OF INTEGER
DECLARE top : INTEGER
top <- -1

PROCEDURE Push(value : INTEGER)
    top <- top + 1
    stack[top] <- value
ENDPROCEDURE

FUNCTION Pop() RETURNS INTEGER
    DECLARE value : INTEGER
    value <- stack[top]
    top <- top - 1
    RETURN value
ENDFUNCTION
```

(a) State the name of the abstract data type shown above, and state its access principle (the order in which items are added and removed). **(2)**

(b) The following sequence of calls is made on an initially empty stack: Push(6), Push(2), Push(9), Pop(), Push(4), Pop(), Pop(). Complete a trace table showing the value of top and the contents of the stack array after each call, and the value returned by each Pop(). **(6)**

Call	top	Stack contents	Returned value
Push(6)			
Push(2)			
Push(9)			
Pop()			
Push(4)			
Pop()			
Pop()			

(c) State the value returned by the final Pop() call in the sequence. **(1)**

© Revision Library | revisionlibrary.co.uk

(d) A text editor uses a stack to implement its 'undo' feature, where the most recently made change is the first to be undone. Explain why a stack, rather than a queue, is the appropriate data structure for this purpose. **(2)**

(Total for Question 2 is 11 marks)

---

3 A circular queue is implemented as an array of 5 elements (indices 0 to 4), using two pointers: rear points to the next free slot for insertion, and front points to the next item to be removed. Both pointers start at index 0.

Enqueue(value): queue[rear] <- value, then rear <- (rear+1) MOD 5
Dequeue(): value <- queue[front], then front <- (front+1) MOD 5, RETURN value

(a) Explain why a linear queue implemented using a fixed-size array can appear to be 'full', and reject further enqueue operations, even when it currently holds far fewer items than the array's capacity. **(2)**

(b) Explain how implementing the queue as a circular queue overcomes this problem. **(2)**

(c) The following operations are carried out in order on the circular queue described above: Enqueue(15), Enqueue(23), Enqueue(8), Dequeue(), Enqueue(41), Enqueue(6), Dequeue(). Complete a trace showing the value of front and rear after each operation, and the value returned by each Dequeue(). **(5)**

(d) State the final values of front and rear, and list the values currently stored in the queue in order from front to rear. **(2)**

(Total for Question 3 is 11 marks)

© Revision Library | revisionlibrary.co.uk

4 This question is about the laws of Boolean algebra.

(a) State De Morgan's first law, expressing NOT(A AND B) in terms of NOT A, NOT B and OR. **(1)**

(b) State De Morgan's second law, expressing NOT(A OR B) in terms of NOT A, NOT B and AND. **(1)**

(c) Simplify the Boolean expression Q = NOT(NOT A OR NOT B) OR (A AND B), showing each step of your working and naming the law used at each step. **(4)**

(d) Simplify the Boolean expression Q = (A AND B) OR (A AND NOT B), showing each step of your working and naming the law used at each step. **(3)**

(e) A student claims that (A OR B) AND (A OR C) is equivalent to A OR (B AND C). State whether the student is correct, and name the Boolean law that justifies your answer. **(2)**

(Total for Question 4 is 11 marks)

---

5 Priya, a software engineer, is designing two separate programs: (1) a program to manage a print queue for a school's network printer, where print jobs must be handled in the order they are submitted, and (2) a program to store and quickly search a large, frequently-changing dictionary of 50000 words for a spell-checker.

Discuss which data structure(s) would be most appropriate for (1) the print queue and (2) the spell-checker's word list, justifying your choice by comparing at least two suitable data structures for each scenario in terms of how their operations are performed and how efficient they are.

(Total for Question 5 is 9 marks)

© Revision Library | revisionlibrary.co.uk

Mark scheme · 1.4 Data Types, Data Structures and Boolean Algebra: 1-hour Pack

Question 1

(a) a hash table is a data structure that stores key-value data in an array-like structure, using a function to determine where a given key's data should be stored **B1**

(a) the hash function takes a key as input and returns an index (address) within the table, so the same key always maps to the same index **B1**

(a) this allows data to be stored and retrieved directly, typically in constant / O(1) average time, without searching through the whole structure **B1**

(a) Answer: A hash table stores data using a hash function that maps each key to an index in the table, allowing fast (average O(1)) storage and retrieval.

(b) correct MOD 7 calculation shown or implied for at least 3 of the 5 keys **M1**

(b) all 5 indices correct: 15->1, 22->1, 9->2, 30->2, 16->2 **A1**

(b) collisions correctly identified: 22 collides with 15 (both index 1); 30 and 16 each collide, since index 2 is already used by 9 (and then 30) **A1**

(b) Answer: 15->1, 22->1, 9->2, 30->2, 16->2. Collisions: 22 (with 15); 30 (with 9); 16 (with 9 and 30).

(c) 15 placed at slot 1 (home slot, no collision) **M1**

(c) 22 probes from slot 1 (occupied) to slot 2 (empty), placed at slot 2; 9 then probes from slot 2 (occupied) to slot 3 (empty), placed at slot 3 **M1**

(c) 30 probes slots 2 and 3 (occupied) to slot 4 (empty), placed at slot 4; 16 probes slots 2, 3 and 4 (occupied) to slot 5 (empty), placed at slot 5 **M1**

(c) final table fully correct: slot0=empty, 1=15, 2=22, 3=9, 4=30, 5=16, 6=empty **A1**

(c) Answer: Slot0: empty, Slot1: 15, Slot2: 22, Slot3: 9, Slot4: 30, Slot5: 16, Slot6: empty.

(d) keys that collide tend to cluster together in consecutive slots (primary clustering) **B1**

(d) this means later insertions and searches may need to probe through many occupied slots before finding an empty slot or the target key, increasing average access time towards O(n) **B1**

(d) Answer: Linear probing causes clustering of colliding keys in consecutive slots, so as the table fills, insertions and searches must probe through more slots, increasing average access time.

Question 2

(a) Stack **B1**

(a) LIFO (Last In, First Out) **B1**

(a) Answer: Stack; LIFO (Last In, First Out).

(b) Push(6): top=0, stack[0]=6 **B1**

(b) Push(2): top=1, stack[1]=2 **B1**

(b) Push(9): top=2, stack[2]=9 **B1**

(b) Pop() returns 9, top becomes 1 **B1**

(b) Push(4): top=2, stack[2]=4 **B1**

(b) Pop() returns 4 (top becomes 1), then Pop() returns 2 (top becomes 0) **B1**

(b) Answer: Returned values in order: 9, 4, 2. Final top = 0.

(c) 2 cao **B1**

(c) Answer: 2

(d) a stack is LIFO, so the last (most recent) action pushed onto the stack is the first one popped off / undone, matching the required behaviour **B1**

(d) a queue is FIFO and would instead undo the oldest recorded action first, which would not match how 'undo' is expected to behave **B1**

(d) Answer: A stack's LIFO order means the most recent action is undone first, matching the required behaviour; a FIFO queue would undo the oldest action first instead.

© Revision Library | revisionlibrary.co.uk

Question 3

(a) as items are dequeued from the front, the front pointer moves forward but the freed slots at the start of the array are not reused **B1**

(a) so once the rear pointer reaches the end of the array, no more items can be enqueued even though slots earlier in the array are empty **B1**

(a) Answer: Freed slots at the front of the array cannot be reused once the rear pointer reaches the end, so the queue appears full despite having empty slots.

(b) the rear (and front) pointer wraps around to index 0 once it passes the last index, e.g. using MOD arithmetic **B1**

(b) so slots freed near the start of the array can be reused, meaning the queue is only full when all slots are genuinely occupied **B1**

(b) Answer: The pointers wrap around to the start of the array using MOD arithmetic, so freed slots are reused and the queue is only full when genuinely at capacity.

(c) Enqueue(15): queue[0]=15, rear=1 **B1**

(c) Enqueue(23) and Enqueue(8): queue[1]=23, queue[2]=8, rear=3 **B1**

(c) Dequeue() returns 15, front becomes 1 **B1**

(c) Enqueue(41) and Enqueue(6): queue[3]=41, queue[4]=6, rear wraps to 0 (since (4+1) MOD 5=0) **B1**

(c) Dequeue() returns 23, front becomes 2 **B1**

(c) Answer: Dequeue() calls return 15, then 23. After all operations: front=2, rear=0.

(d) front=2, rear=0 **B1**

(d) values in order: 8, 41, 6 **B1**

(d) Answer: front=2, rear=0; queue contents (front to rear): 8, 41, 6.

Question 4

(a) NOT(A AND B) = (NOT A) OR (NOT B) **B1**

(a) Answer: NOT(A AND B) = (NOT A) OR (NOT B)

(b) NOT(A OR B) = (NOT A) AND (NOT B) **B1**

(b) Answer: NOT(A OR B) = (NOT A) AND (NOT B)

(c) De Morgan's second law applied to NOT(NOT A OR NOT B), giving NOT(NOT A) AND NOT(NOT B) **M1**

(c) double negation (involution) law applied: NOT(NOT A) AND NOT(NOT B) = A AND B **dM1**

(c) expression now reads Q = (A AND B) OR (A AND B); idempotent law applied (X OR X = X) **dM1**

(c) Q = A AND B cao **A1**

(c) Answer: Q = A AND B

(d) A factorised out using the distributive law: A AND (B OR NOT B) **M1**

(d) complement law applied: B OR NOT B = 1, giving A AND 1 **dM1**

(d) identity law applied: A AND 1 = A, so Q = A cao **A1**

(d) Answer: Q = A

(e) the student is correct **B1**

(e) this is the distributive law (OR distributes over AND, just as AND distributes over OR): A OR (B AND C) = (A OR B) AND (A OR C) **B1**

(e) Answer: Correct; this is an application of the distributive law.

Question 5

Level 3 Discusses both scenarios in detail, comparing at least two suitable data structures for each with accurate, specific justification based on how operations (insertion, deletion, search/access) are performed and their efficiency (e.g. time complexity or real-world behaviour). Recognises the importance of the required order of operations (FIFO for the print queue) and the effect of frequent updates on the choice of data structure for the spell-checker. The answer is coherent, well-structured and uses correct technical terminology throughout. **7-9**

Level 2 Discusses both scenarios with some relevant comparison of data structures, but justification may be partial, one-sided, or lack detail on efficiency/complexity for one of the two scenarios. Some correct use of technical terminology; reasoning is mostly clear but may lack depth or contain minor inaccuracies. **4-6**

Level 1 Identifies at least one appropriate data structure for at least one scenario, with limited or no justification, or justification that is largely inaccurate. The answer may address only one of the two scenarios, or lack coherence. **1-3**

Level 0 No relevant content, or the response does not address data structures in the context given. **0**

Indicative content:

- Print queue: a queue (FIFO) is appropriate because print jobs must be processed in the order they are received, which is fair to users and matches how a physical printer works.
- Print queue: a stack (LIFO) would be inappropriate, since the most recently submitted job would be printed first, which is unfair and does not reflect the required behaviour.
- Print queue: a (circular) queue implemented with an array or linked list allows efficient O(1) enqueue and dequeue operations at each end, without needing to shift other jobs.
- Spell-checker word list: an unsorted array requires a linear search, O(n), to check whether a word exists, which becomes slow for 50000 words.
- Spell-checker word list: a binary search tree allows searching in O(log n) time if the tree remains reasonably balanced, and also supports straightforward insertion of new words as the dictionary changes.
- Spell-checker word list: a hash table allows average O(1) lookup and insertion using a hash function, making it very efficient for a large, frequently-changing word list, though collisions must be managed (e.g. by chaining or probing).
- A hash table (or a well-balanced BST) is likely preferable to an unsorted array for the spell-checker because lookups will be performed very frequently and must be fast; a BST may degrade towards O(n) if it becomes unbalanced, whereas a well-designed hash table maintains near O(1) average performance.
- Valid alternative structures and justifications should be credited if properly explained and appropriate to the scenario, e.g. a sorted array with binary search, O(log n), for the spell-checker if insertions are rare.

© Revision Library | revisionlibrary.co.uk