1.4 Data Types, Data Structures and Boolean Algebra

OCR H446 · Calculators not allowed · about 130 minutes · Write-in edition

Total Marks

Name: _____ Date: ____ / ____ / _____

Answer ALL questions. Show all your working.

1 This question is about character encoding.

(a) State the name of the character encoding standard that can represent characters and symbols from all the world's writing systems, in contrast to standard ASCII, which only covers a limited character set. **(1)**

(b) Given that the ASCII code for the character 'A' is 65, state the ASCII code for the character 'H'. **(1)**

(c) A company's customer database must store names containing accented and non-Latin characters (for example, an umlaut in Muller, or a Chinese name). Explain one advantage of storing this data using Unicode rather than ASCII. **(2)**

(d) A new messaging app must support 200 different characters using a fixed-length binary code. Calculate the minimum number of bits required per character, showing your working. **(2)**

(Total for Question 1 is 6 marks)

2 Q = (A AND B) OR (NOT C) is a Boolean expression with three inputs, A, B and C.

(a) Complete a truth table for Q, showing the output for all 8 possible combinations of A, B and C (using the standard ordering 000 to 111). **(4)**

(b) State the number of rows required in a truth table for a Boolean expression with n input variables, and explain why this formula applies. **(2)**

(c) Describe a logic circuit that implements Q = (A AND B) OR (NOT C), stating the logic gates required and how they are connected. **(2)**

(d) A different logic gate has the property that its single output is 1 only when its two inputs, A and B, have different values. Identify this logic gate. **(1)**

(Total for Question 2 is 9 marks)

3 This question is about two's complement representation and binary addition.

(a) Represent the denary number -45 as an 8-bit two's complement binary number. Show your working. **(2)**

(b) Represent the denary number 77 as an 8-bit two's complement binary number. **(1)**

(c) Add your answers to parts (a) and (b) together in binary, showing your working, and state the resulting bit pattern. **(3)**

(d) A programmer adds the 8-bit two's complement numbers 01111111 and 00000001. State the resulting bit pattern, and explain why this result demonstrates arithmetic overflow. **(2)**

(Total for Question 3 is 8 marks)

4 A program uses the following pseudocode:

TYPE StudentRecord

```
    DECLARE studentID : INTEGER
    DECLARE name : STRING
    DECLARE grade : CHAR
```

ENDTYPE

```
DECLARE student1 : StudentRecord
student1.studentID <- 10234
student1.name <- "Amara Okafor"
student1.grade <- 'B'
```

(a) State the term used to describe a data structure, such as StudentRecord above, that groups together related data items, which may be of different data types, under a single identifier. **(1)**

(b) State the value stored in student1.name after the code above has executed. **(1)**

(c) Write pseudocode to (i) declare an array called cohort of 30 StudentRecord elements, and (ii) set the grade field of the 5th student in the array to 'A'. **(3)**

(d) Explain one advantage of using a single array of StudentRecord elements, rather than three separate arrays (one for IDs, one for names and one for grades), to store the same cohort data. **(2)**

(Total for Question 4 is 7 marks)

5 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 5 is 12 marks)

6 This question is about floating point representation.

(a) A computer represents floating point numbers using an 8-bit mantissa and a 4-bit exponent, both stored in two's complement, with the mantissa normalised so that its first two bits differ. Represent the denary integer 22 in this floating point form, showing your working for both the mantissa and the exponent. **(3)**

(b) State what is meant by a normalised mantissa, and explain why floating point numbers are stored in normalised form. **(2)**

(c) Many decimal fractions, such as 0.1, cannot be represented exactly in binary floating point. Explain why this happens, and the effect it can have on a program that repeatedly adds 0.1 to a running total. **(2)**

(d) State the effect on the floating point number system of allocating (i) more bits to the exponent and (ii) more bits to the mantissa, keeping the total number of bits used unchanged. **(2)**

(Total for Question 6 is 9 marks)

7 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)**

(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 7 is 11 marks)

8 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 8 is 11 marks)

9 The values 50, 30, 70, 20, 40, 60, 80, 35 are inserted, in that order, into an initially empty binary search tree (BST), where for each node, values less than the node are inserted into its left subtree and values greater than or equal to the node are inserted into its right subtree.

(a) State the value stored at the root of the tree once all eight values have been inserted. **(1)**

(b) State the left child and the right child of the node containing 30. **(2)**

(c) State the left child and the right child of the node containing 70. **(2)**

(d) A pre-order traversal visits nodes in the order: root, then left subtree, then right subtree. State the sequence of values produced by a pre-order traversal of the completed tree. **(3)**

(e) An in-order traversal visits nodes in the order: left subtree, root, right subtree. State the sequence of values produced by an in-order traversal of the tree, and explain why an in-order traversal of any binary search tree always produces the stored values in ascending order. **(4)**

(f) State one advantage of storing this data in a binary search tree, rather than in a sorted (ordered) one-dimensional array, in terms of the efficiency of inserting new data values. **(2)**

(Total for Question 9 is 14 marks)

10 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 10 is 11 marks)

11 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 11 is 9 marks)

Mark scheme · 1.4 Data Types, Data Structures and Boolean Algebra

Question 1

(a) Unicode **B1**

(a) Answer: Unicode

(b) 72 cao **B1**

(b) Answer: 72

(c) Unicode includes code points for accented characters and non-Latin scripts, whereas standard 7-bit ASCII only encodes basic Latin letters, digits and symbols **B1**

(c) so names can be stored and displayed accurately rather than being corrupted or losing information **B1**

(c) Answer: Unicode covers accented and non-Latin characters that ASCII cannot represent, so names are stored and displayed correctly.

(d) 2^7=128 (insufficient, since 128<200) and 2^8=256 (sufficient, since 256≥200) tested **M1**

(d) 8 bits cao **A1**

(d) Answer: 8 bits

Question 2

(a) A=0,B=0 rows (C=0 then C=1) correctly give Q=1, Q=0 **B1**

(a) A=0,B=1 rows (C=0 then C=1) correctly give Q=1, Q=0 **B1**

(a) A=1,B=0 rows (C=0 then C=1) correctly give Q=1, Q=0 **B1**

(a) A=1,B=1 rows (C=0 then C=1) correctly give Q=1, Q=1 **B1**

(a) Answer: Q values (A,B,C from 000 to 111): 1, 0, 1, 0, 1, 0, 1, 1

(b) 2^n rows **B1**

(b) because each of the n variables can independently take one of 2 values (0 or 1), giving 2 x 2 x ... (n times) = 2^n possible combinations **B1**

(b) Answer: 2^n rows, since each of the n inputs can independently be 0 or 1.

(c) an AND gate with inputs A and B, and a NOT gate with input C **B1**

(c) the outputs of the AND gate and the NOT gate are connected as the two inputs to an OR gate, whose output is Q **B1**

(c) Answer: AND gate (inputs A, B) and NOT gate (input C) feed into an OR gate, whose output is Q.

(d) XOR (exclusive OR) **B1**

(d) Answer: XOR

Question 3

(a) 45 correctly represented in binary (00101101) and correctly inverted (11010010) **M1**

(a) 11010011 cao (invert then add 1) **A1**

(a) Answer: 11010011

(b) 01001101 cao **B1**

(b) Answer: 01001101

(c) correct bit-by-bit binary addition attempted with carries shown **M1**

(c) final carry out of the 8th bit correctly discarded **M1**

(c) 00100000 cao (= 32, since -45 + 77 = 32) **A1**

(c) Answer: 00100000 (= 32)

(d) 10000000 stated **B1**

(d) explanation that both operands are positive (sign bit 0) but the result has sign bit 1 (negative), which is impossible when adding two positive numbers, so overflow has occurred **B1**

(d) Answer: 10000000 (overflow, since 127+1=128 cannot be represented as a positive 8-bit two's complement number)

Question 4

(a) a record **B1**

(a) Answer: A record.

(b) "Amara Okafor" cao **B1**

(b) Answer: Amara Okafor

(c) array declared with correct bounds for 30 elements of type StudentRecord, e.g. DECLARE cohort : ARRAY[0:29] OF StudentRecord **M1**

(c) correct index used for the 5th element (index 4, since arrays are zero-indexed) **M1**

(c) correct dot notation used to access and assign the field, e.g. cohort[4].grade <- 'A' (any correct logic, ignore minor syntax) **A1**

(c) Answer:

```
DECLARE cohort : ARRAY[0:29] OF StudentRecord
cohort[4].grade <- 'A'
```

(d) all the data relating to one student is kept together under one index, so there is no risk of the arrays becoming out of step (e.g. after sorting or deleting an entry) **B1**

(d) makes the code easier to read and maintain, since related fields are accessed via one structure (e.g. cohort[i].grade) rather than needing to keep several array indices synchronised **B1**

(d) Answer: Keeps each student's data together under one index, avoiding the arrays becoming mismatched, and makes the code clearer to maintain.

Question 5

(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 6

(a) 22 written in binary (10110) and expressed as a normalised fraction, e.g. 0.10110×2^5 **M1**

(a) exponent 5 correctly converted to 4-bit two's complement (0101) **M1**

(a) mantissa 01011000 and exponent 0101 both correct (oe equivalent padding) **A1**

(a) Answer: mantissa = 01011000, exponent = 0101

(b) normalised means the mantissa's first two bits differ (no redundant leading 0s or 1s before the significant digits) **B1**

(b) normalisation gives each number a single, unique representation and maximises the number of significant bits available, maximising precision **B1**

(b) Answer: Normalised means the first two mantissa bits differ; this is used to maximise precision and give a unique representation for each value.

(c) 0.1 is a recurring (non-terminating) binary fraction, so it must be truncated/rounded to fit the fixed number of mantissa bits **B1**

(c) the small rounding error accumulates over repeated additions, causing the running total to drift from the mathematically expected value **B1**

(c) Answer: 0.1 cannot be represented exactly in binary so it is rounded; repeated addition accumulates this rounding error, causing the total to drift from the true value.

(d) (i) more exponent bits increases the range of values that can be represented **B1**

(d) (ii) more mantissa bits increases the precision (accuracy) of the values represented **B1**

(d) Answer: (i) increases range; (ii) increases precision.

Question 7

(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.

Question 8

(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 9

(a) 50 **B1**

(a) Answer: 50

(b) left child = 20 **B1**

(b) right child = 40 **B1**

(b) Answer: Left child = 20, right child = 40.

(c) left child = 60 **B1**

(c) right child = 80 **B1**

(c) Answer: Left child = 60, right child = 80.

(d) root 50 visited first, correctly followed by the complete left subtree **M1**

(d) left subtree correctly given as 30, 20, 40, 35 **M1**

(d) full sequence 50, 30, 20, 40, 35, 70, 60, 80 cao **A1**

(d) Answer: 50, 30, 20, 40, 35, 70, 60, 80

(e) sequence 20, 30, 35, 40, 50, 60, 70, 80 attempted with correct method **M1**

(e) sequence fully correct (cao) **A1**

(e) explanation that at every node, all values in its left subtree are smaller and all values in its right subtree are larger (the BST property) **B1**

(e) so visiting left subtree, then the node, then right subtree at every level guarantees values are output smallest-first **B1**

(e) Answer: 20, 30, 35, 40, 50, 60, 70, 80; because at every node all values in its left subtree are smaller and all values in its right subtree are larger, processing left-node-right at each level always yields ascending order.

(f) inserting into a BST only requires traversing a single path from the root (typically O(log n) comparisons for a balanced tree) and attaching a new node **B1**

(f) whereas inserting into a sorted array requires shifting all subsequent elements along by one position to keep the array in order, which is O(n) **B1**

(f) Answer: Inserting into a BST only needs a single root-to-leaf path traversal (no shifting), far more efficient than a sorted array, where all following elements must be shifted along.

Question 10

(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 11

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.