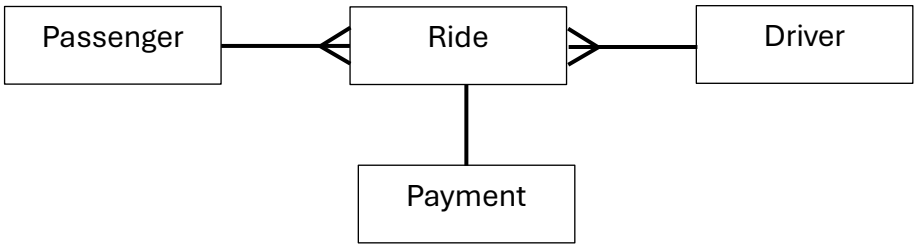
1

(a)



[1m] All four entities shown

[1m] Correct 1:1 relationship shown (Ride - Payment)

[1m] Correct 1:M relationships (e.g., one Driver has many Rides, One Passenger has many Rides)

(b)

Ride (RideID, PickupLocation, DropoffLocation, RideDateTime, Status, DriverID, PassengerID)

Driver (DriverID, FullName, VehicleNumber, LicenseExpiryDate, ContactNumber)

Passenger (PassengerID, FullName, PhoneNumber, EmailAddress)

Payment (PaymentID, RideID, PaymentMethod, FareAmount, PaymentStatus)

(c)

(i)

```
SELECT DriverID, SUM(FareAmount) AS TotalEarnings
FROM Ride
JOIN Payment ON Ride.RideID = Payment.RideID
WHERE PaymentStatus = 'paid'
GROUP BY DriverID
ORDER BY TotalEarnings DESC;
```

(ii)

```
UPDATE Payment
SET PaymentStatus = 'paid'
WHERE RideID = 'R7645';
```

(d)
Reasons why NoSQL is suitable for storing customer reviews:

1. **Unstructured/semi-structured data** – Reviews may contain ratings, free text, emojis, images, etc., which NoSQL can store more naturally than SQL.
2. **Schema flexibility** – Different reviews can have different fields (e.g. some with upvotes, some without) without needing database redesign, supporting future feature expansion.
3. **Horizontal scalability** – NoSQL can distribute data across multiple servers, handling thousands/millions of reviews efficiently as the platform grows.

4. **Fault tolerance** – Reviews can be replicated across multiple servers; if one server fails, data is still accessible. This ensures reliability and availability for both passengers and drivers.
5. **Hierarchical data storage** – NoSQL supports nested structures. For example, a single review record could include:
    - Review text and rating
    - Driver replies
    - User upvotes
    - Timestamps

    This avoids the need to split into multiple relational tables and makes retrieving full review threads faster.

6. **Performance on queries** – Fetching a review document is faster (no complex joins across multiple tables as in SQL), which improves the app's responsiveness.

(e)
(Any 2 of the following, or any other reasonable answers):
1. Location data misuse:
   Concern that real-time or historical pickup/drop-off locations might be shared with third parties or used to track personal habits.
2. Unconsented data sharing:
   Worry that personal information (e.g., name or contact number) may be shared with external advertisers or third-party apps without consent.
3. Data retention:
   Concern about how long the company keeps ride history and whether old data is securely deleted.

(f)
(Any 2 of the following, or any other reasonable answers):
1. Do not misuse passenger contact details:
   A driver must not use the passenger's phone number (e.g., for personal messages or marketing) after the ride is completed.

2. Keep information confidential:
   A driver must not disclose passenger details (e.g., drop-off location, name) to others or post them online.
3. Report data breaches:
   If a driver's device is lost or compromised, they must report it to the platform if passenger data is affected.

(g)
Actions the company should take:

1. **Anonymise or pseudonymise** data before sharing (e.g., remove names, mask phone numbers)
2. **Obtain informed consent** from passengers OR ensure a **data sharing agreement** is in place that complies with data protection laws (e.g., PDPA, GDPR)
3. Limit Data to the Minimum Necessary: e.g., transaction IDs, device fingerprints) — not unnecessary details like full names or contact numbers.

2a)

- `AddOrder(orderID)`: Hash table, O(1) **average** case insertion time
- `CheckOrderExists(orderID)`: Hash table, O(1) **average** case lookup time
- `GetMostRecentOrder()`: Stack, O(1) to access top of stack
- `GetAllOrdersSorted()`: Binary Search Tree, O(n) in-order traversal returns sorted list in O(n)

AddOrder - Should not be Unordered List even as insert/delete can be O(1), as the system will also need to support CheckOrderExists, which makes Hash Table more suitable

(b)
A linked list does not support direct access to elements by key.
To check whether an order exists, the system must traverse the list from start to end, resulting in O(n) time complexity. This becomes inefficient as the number of orders grows.

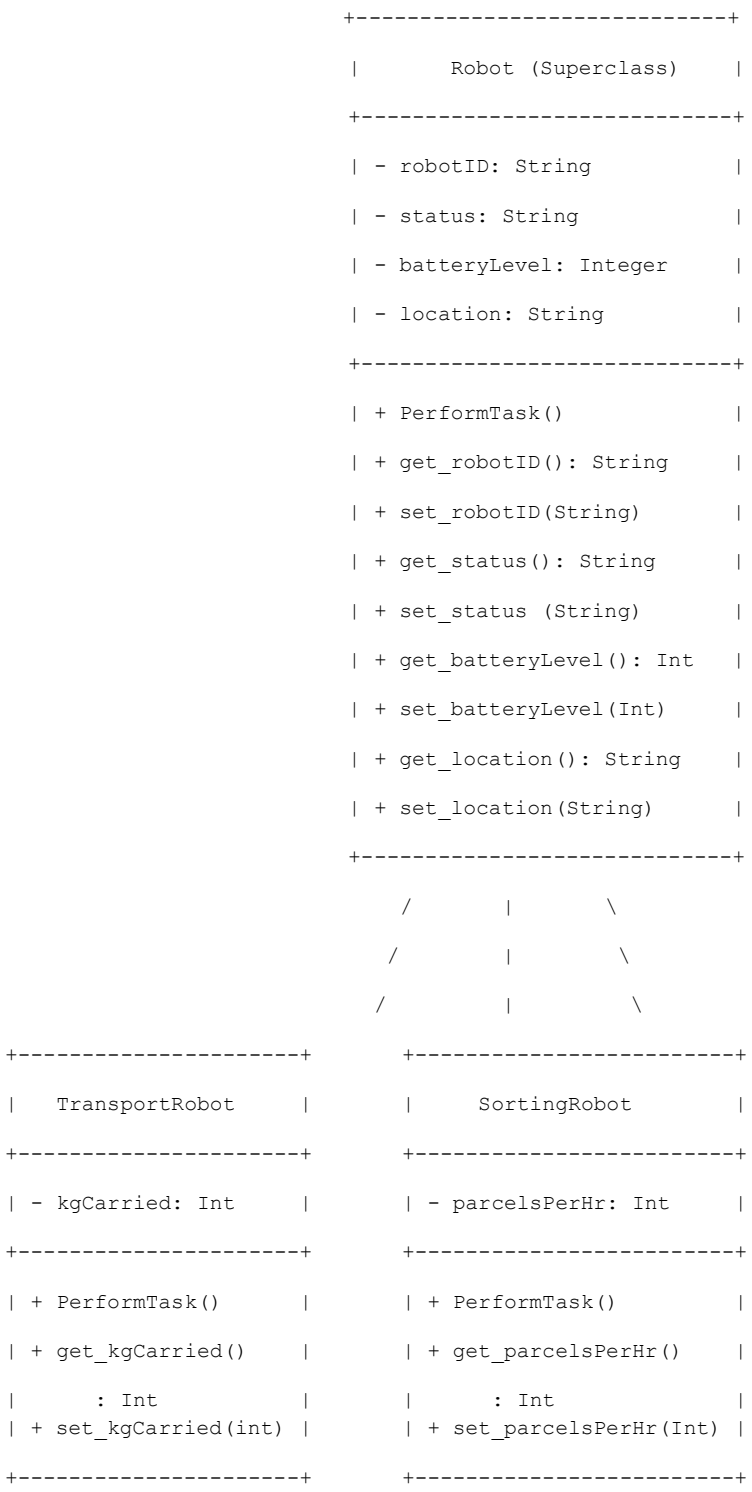
(c)
The company wants to display all active orders sorted by orderID or order time on a dashboard for warehouse staff.

1m-Weakness of HT
Hash tables do not maintain any inherent order of keys.
1m – Explain the inefficiency
To retrieve active orders in a sorted order, the system must first extract all values and then sort them manually, which adds an extra O(n log n) operation.
1m-Proposing suitable alternative
This reduces efficiency compared to using a data structure that maintains order, such as a binary search tree or a priority queue.

(d)

Use a queue to manage orders. A queue naturally supports first-come, first-served processing, which matches the requirement to fulfil orders in chronological order. Insertion of new orders is O(1) at the rear, and fulfilling orders is O(1) at the front, making it efficient even under high sales volumes.
1m – identify queue (FIFO characteristics)
1m – Justifies enqueue and dequeue operation efficiency

3a)



Names of superclass and subclasses [1]
Superclass attributes [1]
Superclass getters and setters [1]
Subclasses all attributes [1]
Subclasses all getters and setters [1]
Polymorphic method in super and subclasses [1]
Arrows must point to superclass [1]
Private minus and public plus [1]

b)

(i) Instantiation is the process of **creating an object from a class**. For example, creating a new TransportRobot object called robot1 is instantiating the TransportRobot class.

(ii) Encapsulation is the concept of **hiding internal data** of an object and allowing access only through methods. *Accept: bundling of private data and public methods in a single class*

In the simulation, the robot's batteryLevel can be made private, and only modified via the PerformTask() method.

(iii) Inheritance allows a subclass to **reuse attributes and methods** from a superclass. In this case, TransportRobot, SortingRobot, and RestockRobot inherit common attributes and methods from the Robot superclass.

(iv) Polymorphism allows the same method name to have **different implementations** in different subclasses. Each subclass defines its own version of PerformTask(), which behaves differently depending on the robot type.

(c)

Type of test	Test data	Reasoning
Normal	30	A typical value within the expected range of carried weight
Abnormal	-5	Invalid: negative weight not allowed
Extreme	0 or maximum allowed (i.e.., 100)	Edge cases that test no load and upper limit of carrying capacity

4a)

Token	Action	Stack (Top → Bottom)	
4	Push 4	4	
5	Push 5	5, 4	
6	Push 6	6, 5, 4	
*	Pop 6 and 5 → 5 * 6 = 30 → Push 30	30, 4	1m
+	Pop 30 and 4 → 4 + 30 = 34 → Push 34	34	1m
2	Push 2	2, 34	
-	Pop 2 and 34 → 34 - 2 = 32 → Push 32	32	1m

Answer: 32

***1 mark** – Correctly showing the **first three pushes** (4, 5, 6) onto the stack.*

1 mark *– Correctly handling the first operator * (pop 6 and 5, compute 30, push result).*

1 mark *– Correctly handling the + operator (pop 30 and 4, compute 34, push result).*

1 mark *– Correctly handling the final two tokens (2 and -) and reaching the final answer 32.*

4b)

Operands must be popped in reverse order: first A, then B, so the operation is performed as B op A, with B as the first operand and A being the second operand.

If reversed, A op B would yield incorrect results, especially for non-commutative operations like subtraction and division.

4c)

Modified lines (only change needed is adding another ELSE IF block):

```
16      ELSE IF Token = '/' THEN
17          PUSH (B DIV A) TO Stack
18      ELSE IF Token = '^' THEN
19          PUSH (B ^ A) TO Stack
```

Mark scheme (3 marks):

- 1 mark – Identifies correct position for insertion (between line 17 and line 18).
- 1 mark – Correct conditional test (Token = '^').
- 1 mark – Correct push statement showing B ^ A.

4d)

A **stack underflow** error may occur when the function attempts to **POP from an empty stack**, or when there are **fewer than 2 items** in the stack before an operator is processed.

It happens when the postfix expression is **invalid or malformed**, such as:

Tokens = [3, "+"]

Pushed '3' into the stack, reads '+', attempts to pop twice - but stack only have one single item, which will cause a stack underflow error

Insert a condition before popping to verify that the stack has at least 2 items:

```
07          ELSE
08              IF Stack.LENGTH < 2 THEN
09                  OUTPUT "Error: Not enough operands"
10                  RETURN -1
11              ENDIF
12              A ← POP Stack
13              B ← POP Stack
```

Component	Marks
1. Correctly **identifies** the runtime error as a **stack underflow**	1
2. **Explains how it could happen**, e.g., processing an operator when the stack has < 2 items	1
3. Proposes a valid **safeguard**, e.g., check Stack.LENGTH < 2	1
4. Shows **correct placement or logic** of the safeguard in the code	1

5a)

(i)

- 178 ÷ 2 = 89, remainder **0**
- 89 ÷ 2 = 44, remainder **1**
- 44 ÷ 2 = 22, remainder **0**
- 22 ÷ 2 = 11, remainder **0**
- 11 ÷ 2 = 5, remainder **1**
- 5 ÷ 2 = 2, remainder **1**
- 2 ÷ 2 = 1, remainder **0**
- 1 ÷ 2 = 0, remainder **1**

Binary (read from bottom to top): 10110010

Answer: 10110010

(ii)

Hexadecimal

- 178 ÷ 16 = 11 remainder **2**
- 11 in hexadecimal is **B**

Answer: B2

(b)

```
result ← STR(remainder) + result
```

Prepending ensures most significant digit is on the left.

(c)

```
FUNCTION ConvertToBinary(number: INTEGER) RETURNS STRING
    IF number = 0 THEN
        RETURN ""
    ELSE
        RETURN ConvertToBinary(number DIV 2) + STR(number MOD 2)
    ENDIF
ENDFUNCTION
```

- 1 mark for correct base case
- 1 mark for correct recursive call
- 1 mark for converting remainder to string
- 1 mark for **returning** the concatenated result in correct order

6a)

CONDITIONS	**OUTCOMES**							
	1	2	3	4	5	6	7	8
Is a Member	Y	Y	Y	Y	N	N	N	N
Is Tuesday	Y	Y	N	N	Y	Y	N	N
Spends $50 or more	Y	N	Y	N	Y	N	Y	N
ACTIONS								
10% discount	X	X	X	X				
$5 dining voucher	X	X			X	X		
Additional $10 dining voucher					X		X	

b)

CONDITIONS	**OUTCOMES**							
	1+2	3+4	5	6	7			
Is a Member	Y	Y	N	N	N			
Is Tuesday	Y	N	Y	Y	N			
Spends $50 or more	-	-	Y	N	Y			
ACTIONS								
10% discount	X	X						

$5 dining voucher	X		X	X				
Additional $10 dining voucher			X		X			

7

(a) The Domain Name System (DNS) translates the human-readable web address api.foodtrack.sg into its corresponding IP address (e.g. 203.0.113.15) so the rider's device can locate and communicate with the server.

- 1 mark for identifying that DNS performs a translation/lookup
- 1 mark for specifying the result is an IP address

(b)

(i)
MAC: A unique identifier hard-coded into the network interface card (NIC) of a device. Used for communication within the local network segment (e.g., Wi-Fi or LAN).

IP: A logical address assigned to a device that identifies its location in a larger network (Internet). Used for routing data between networks.

(ii)
Both are needed because data travels across different networks using IP addresses for global delivery, but once the data reaches the destination network, the MAC address is used for final delivery to the correct device.

(c) (Any three, correct component and brief explanation of purpose)

1. Destination Address
    - Identifies the IP address of the receiving device (e.g. the central server)
    - Used by routers to ensure the packet reaches the correct destination
2. Sequence Number
    - Indicates the position of the packet in a sequence of packets from a larger message
    - Helps the recipient reassemble data in the correct order
3. Checksum (Error-checking field)
    - A calculated value based on the data in the packet
    - Used by the receiver to verify integrity and detect any transmission errors
4. Source Address – IP of the sender (rider's device); allows reply to be sent back
5. Payload – The actual data, e.g. GPS coordinates, delivery status
6. Protocol field – Identifies the type of data or how it should be processed
7. Time to Live (TTL) – Prevents packets from circulating endlessly by limiting hops

(d)
Advantage 1: Riders can share updates (e.g., delivery status, traffic alerts) directly with nearby peers without internet, which is helpful in areas with weak connectivity.

Advantage 2: Reduces reliance on the central server, potentially decreasing latency and improving resilience if the server is temporarily unreachable.

Disadvantage: P2P requires devices to be physically near each other for communication and can lead to inconsistent or delayed updates if devices don't sync promptly.

Security - Difficult to monitor the flow of data and ensure security, especially as more devices are added. Raises risk of malicious attacks, which could hinder delivery operations as riders may be unable to communicate with one another.

2m per advantage (1 for idea, 1 for clear explanation)
2m for disadvantage (1 for idea, 1 for clear explanation)