IN-COURSE ASSESSMENT (ICA) SPECIFICATION

Name: **Hina Malik**	Student Number: **E4556347**
Module Title: **Algorithmic Problem Solving**	Module Code: **CIS1071-N**
Assignment Title: **APS - Problem Solving & Data Structures ICA**	Date: **07/05/2025**

Part 1 – Problem Solving :

Question 1 – Transitions:

1.

a) If we have n people wishing to cross the bridge with times $t1 \leq t2 \leq t3 \leq \ldots \leq tn$, how many possible initial transitions will there be?

Solution:

Suppose $\binom{n}{1}$ = n transitions {t1, t2, t3, ... , tn }

Since the problem requires two people to cross together at the start, so for 2-person transitions :

The number of ways to choose 2 people from **n** is given by the combination formula:

$$\textit{Number of transitions} = \binom{n}{2} = \frac{n(n-1)}{2}$$

→ {t1, t2} , {t1, t3} , {t1, t4} , ... {t1, tn}
{t2, t3} , {t2, t4} , ... {t2, tn}
.
.
.
{t(n-1), tn}

So for **n** number of people, total number of initial transitions would be :

$$T(n) = \frac{n(n-1)}{2}$$

If we take **4** people ,

$$T(4) = \frac{4 \times 3}{2} = \frac{12}{2} = 6$$

So for *4 number of people* (t4) of people, we have the following **6** possible transitions :

- (t1,t2) , (t1,t3) , (t1,t4)
- (t2,t3) , (t2,t4)
- (t3,t4)

b) If t1 = 2, t2 = 5, t3 = 10, t4 = 20 and tb = 38 then which strategy for crossing the bridge must we use to ensure that the torch doesn't run out of battery, Strategy 1: The fastest person crosses with all the others or Strategy 2: The two slowest cross together.

Strategy Comparison

Given Data:

t1=2 min
t2=5 min
t3=10 min
t4=20 min
tb=Torch battery life =38

Solution: We compare the two strategies S1 and S2:

Strategy 1 (S1): The fastest person crosses with all others.

t1 and t2 cross first:	Time = t2=5 min
t1 returns:	Time = t1=2 min
t1 and t3 cross:	Time = t3=10 min
t1 returns:	Time = t1=2 min
t1 and t4 cross:	Time = t4=20 min

Total time of S1: 5+2+10+2+20= *39 minutes*

Strategy 2 (S2) : The two slowest cross together.

t1 and t2 cross first:	Time = t2=5 min
t1 returns:	Time = t1=2 min
t3 and t4 cross:	Time = t4=20 min
t2 returns:	Time = t2=5 min
t1 and t2 cross again:	Time = t2=5 min

Total time of (S2)**:** 5+2+20+5+5= *37 minutes*

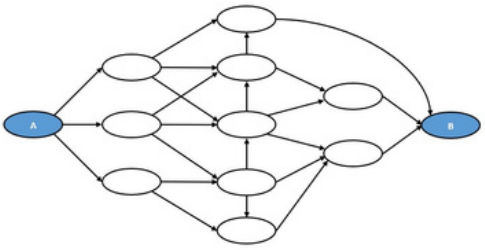
Conclusion:

- Strategy 2 takes *37 minutes,* which is less than the torch's battery life, S1< tb.
- Strategy 1 takes *39 minutes,* which exceeds the battery life, S2 > tb.

Answer**: Strategy 2**

2.

Consider the state transition diagram below.



a) Calculate the number of paths from node A to node B from this diagram, ensuring each node is labelled with the path count

Step 1: The Nodes Directly Connected to A

- **A** (starting node) is the source.
- **X, Y, Z** (directly connected to A) each get 1 (one path from A to each).

Step 2: Middle Nodes (M1–M5)

- **M4** (from Y and Z): 1 (Y) + 1 (Z) = 2
- **M5** (from Z and M4): 1 (Z) + 2 (M4) = 3
- **M3** (from X, Y, and M4): 1 (X) + 1 (Y) + 2 (M4) = 4
- **M2** (from X, Y, and M3): 1 (X) + 1 (Y) + 4 (M3) = 6
- **M1** (from X and M2): 1 (X) + 6 (M2) = 7

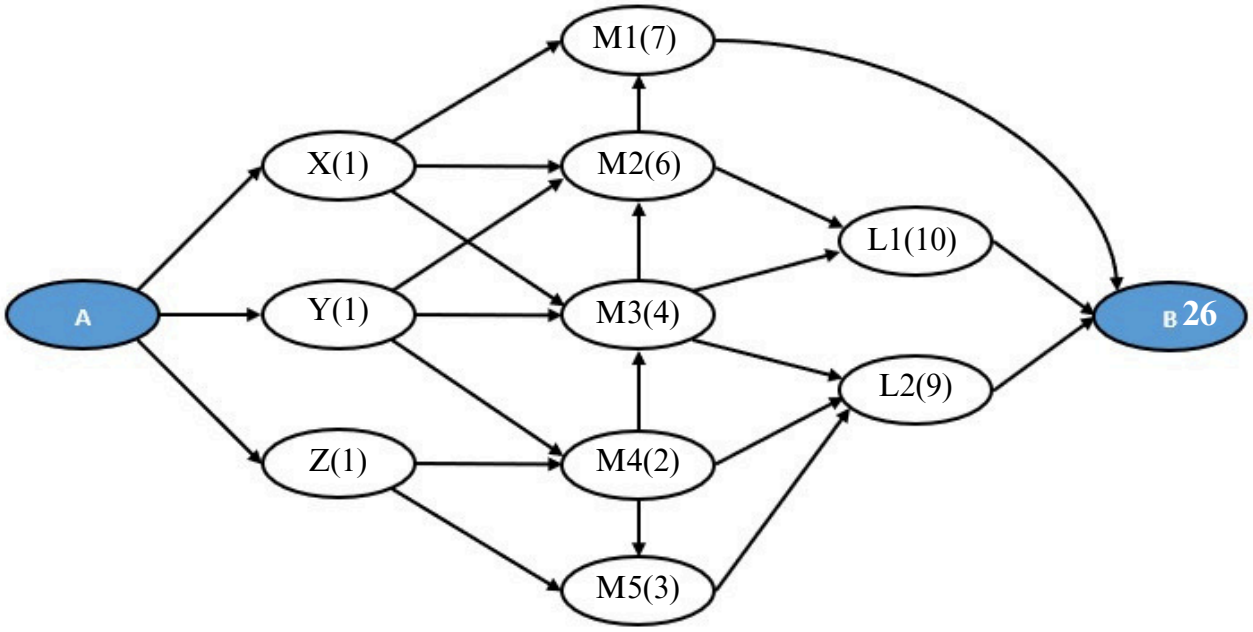
Step 3: Nodes Connected to B (L1 and L2)

- **L1** (from M2 and M3): 6 (M2) + 4 (M3) = 10
- **L2** (from M3, M4, and M5): 4 (M3) + 2 (M4) + 3 (M5) = 9

Step 4: Calculate Total Paths to B

- **B** (from M1, L1, and L2): 7 (M1) + 10 (L1) + 9 (L2) = 26

Final Labeled Diagram:



Final Answer

The number of paths from node A to node B is **26**.
Each node is labeled with its path count as shown in the table and diagram above.

b. State the name of the counting rule you used to obtain this result

This solution uses the Addition Rule (Sum Rule). For each node, the total number of paths is the sum of all paths from its predecessor nodes. Its also known as Path Counting using **Dynamic Programming.**

Question 2 – Invariants:

1.

Assignment	Expression	Invariant
$i, j := i + 5, j - 3$	$5i - 3j$	False
$n, m := n + 10, m + 15$	$3n > 2m$	True
$x, y := x / 2, y * 2 \quad x :=$	$xy = k$	True
$2x$	$even(x)$	False

The function *even*(*x*) returns *True* when *x* is an even number (x mod 2 = 0) and *False* otherwise.

(1)

Assignment: i, j := i + 5, j - 3
Expression: 5i - 3j

Let i' = i + 5 and j' = j - 3

Solution

5i' - 3j' = 5(i + 5) - 3(j - 3)
= 5i + 25 - 3j + 9
= 5i - 3j + 34

Invariant: False / No

(2)

Assignment: n, m := n + 10, m + 15
Expression: 3n > 2m

Let n' = n + 10 and m' = m + 15

Solution

3(n + 10) > 2(m + 15)
=> 3n + 30 > 2m + 30
=> 3n > 2m

Invariant: True / Yes

(3)

Assignment: x, y := x/2, y * 2
Expression: xy = k

Let x' = x/2 and y' = y * 2

Solution

x'y' = (x/2) * (y * 2)
= xy

Since xy = k, then x'y' = k

Invariant: True / Yes

Assignment: x := 2x
Expression: even(x)

Let x = 3, even(3) is false.

Solution:
After the assignment x := 2x, x becomes 6
Then even(6) is True

The expression even(x) becomes True after the assignment, regardless of whether x was initially even or odd. Since the truth value of even(x) changed, it is not an invariant.

Invariant: False / No

2.

A research facility wishes to recruit more staff. It has 12 members of staff who are all researchers. A number of these researchers become supervisors and are assigned 3 new junior researchers to supervise. Some of these junior researchers become supervisors and are assigned 3 student researchers to supervise.

The recruitment processes finishes when the department has 27 members of staff in total made up of supervisors and researchers. **How many members of staff are researchers (with no supervisory duties)?** You should use the following steps in order to solve the problem. The variables *r* and *t* represent the number of researchers and the number of members of staff, respectively.

1) Identify the information that is given about the initial and final values of r and t.

Initial Values:
- r (number of researchers) = 12
- t (total members of staff) = 12

Final Values:
- t (total members of staff) = 27

2) Model the process of assigning researchers as an assignment to r and t.

New staff added (t) :
new staff= $t_{final} - t_{initial} = 27 - 12 = 15$

Every supervisor adds 3 new researchers (r) :
number of supervisors = $\frac{\text{new staff}}{3} = \frac{15}{3} = 5$

3) Identify an invariant of the assignment.

At every step, the number of staff increases in multiples of 3 (since each supervisor brings in 3 new researchers).

Invariant:

$$t-12 \text{ is always a multiple of } 3$$

4) Combine the previous steps to deduce the final value of r.

Deduction of Final Number of Researchers :

Out of 27 total staff, 5 are supervisors.

The rest are researchers with no supervisory duties:

$$\mathbf{r} \text{ final} = \mathbf{t} \text{ final} - \text{number of supervisors} = 27 - 5 = 22$$

Conclusion :

Final number of researchers with no supervisory duties: **22**

Question 3 – Calculating Programs:

A team of programmers are developing software to assist in solving a range of mathematical problems. A component of the software is based around the following code snippet:

```
{Precondition: n > 0}
result := 1
for (i = 2; i < n + 1; i++)
        result := result * i
{Postcondition: result = n!}
```

The loop is designed to operate from states satisfying the precondition $\{n>0\}$ and end in states satisfying the postcondition $\{result = n!\}$. The team wants to ensure that their software operates correctly, and request that you verify the loop using induction. You must explicitly identify the base case and induction hypothesis.

1.

State explicitly the base case of the program and induction hypothesis.

Base Case:
Loop Invariant
At the start of each iteration of the loop:
result = (i-1)!

Base Case (Initialization)

- Before the loop starts:
  - i = 2
  - result = 1
  -
- Invariant check:
  - (i-1)! = (2-1)! = 1! = 1
  - So, result = (i-1)!
  -
- Hoare triple for initialization:

```
{n > 0}
result := 1; i := 2
{result = (i-1)!}
```

Induction Hypothesis :

- Assume: At the start of some iteration,
  - i = k
  - result = (k-1)!
- Hoare triple for induction hypothesis:

```
{result = (i-1)!}
(loop body executes)
{result = (i-1)!
after increment}
```

2.

Show how to use the induction hypothesis to solve the inductive step.

Restate the Loop Invariant
At the start of each iteration of the loop:
result = (i-1)!

Inductive Step
Goal:
Show that if the invariant holds at the start of an iteration (i = k, result = (k-1)!), it also holds at the start of the next iteration (i = k+1, result = k!).

Hoare Triple for the Inductive Step:

```
{result = (i-1)! ∧ i = k}
result := result * i;
i := i + 1
{result = (i-1)! ∧ i = k+1}
```

Step-by-Step Reasoning:

- Assume: At the start of the iteration,

```
result = (k-1)! and i = k.
```

- First statement:

```
result := result * i
Now, result = (k-1)! * k = k!
```

- Second statement:

```
i := i + 1
Now, i = k + 1
```

- After both statements:

```
result = k! and i = k + 1, so
result = (i-1)! (since i-1 = k)
```

Conclusion :
Therefore, the invariant holds at the start of the next iteration. If the invariant holds before an iteration, it holds after the iteration as well.

Part 2 – Data Structures:

Question 4 – Linked Lists:

1.

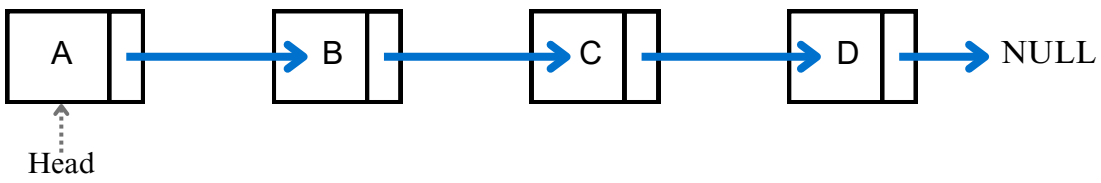
Explain the main difference between singly-linked list, doubly-linked list, and circular-linked list.

Feature	Singly-Linked List	Doubly-Linked List	Circular-Linked List
Direction	Forward only.	Forward and backword.	Forward, forms a cycle.
Advantage	Simple and efficient for basic operations	Facilitates efficient backward traversal.	Useful for cyclical data representation.

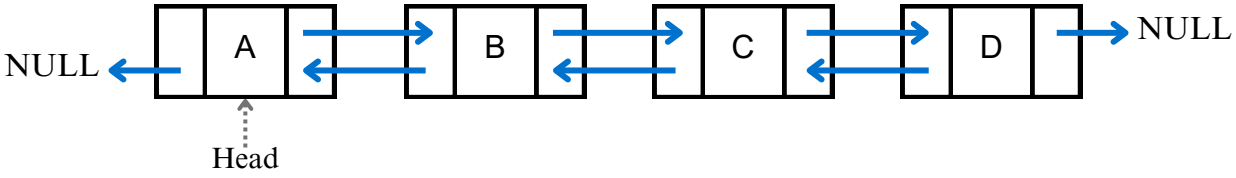
3

Diagram:

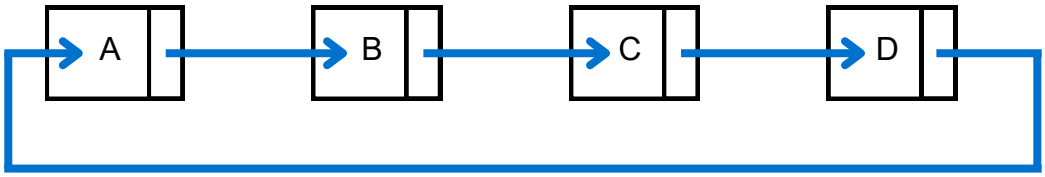
Singly-Linked List



Doubly-Linked List



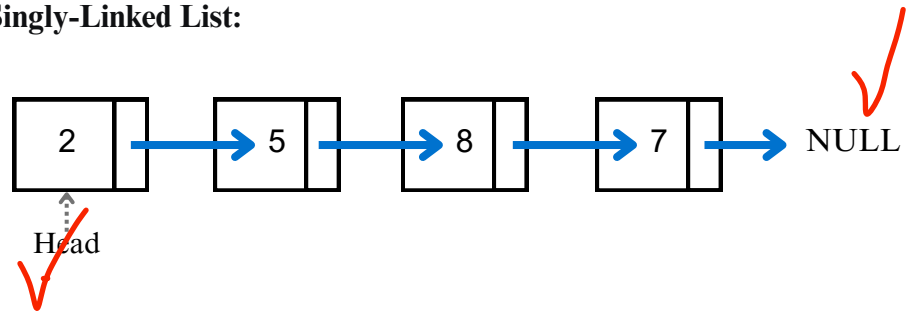
Circular-Linked List



2.

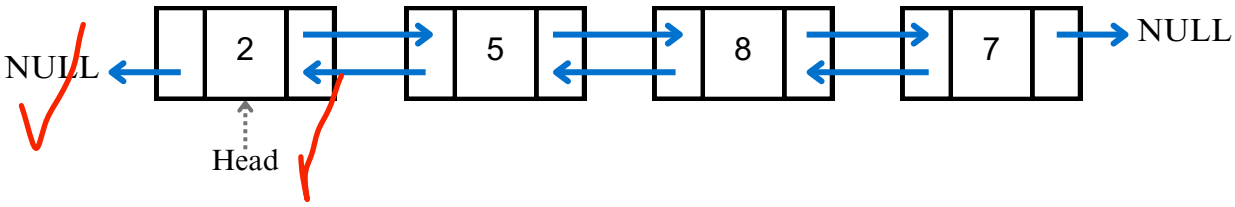
Draw three diagrams for each list type to represent the list containing nodes: 2->5->8->7.

1. Singly-Linked List:



In a singly-linked list, each node points only to the next node in the sequence. The last node points to NULL, indicating the end of the list. The head pointer points to the first node (2).

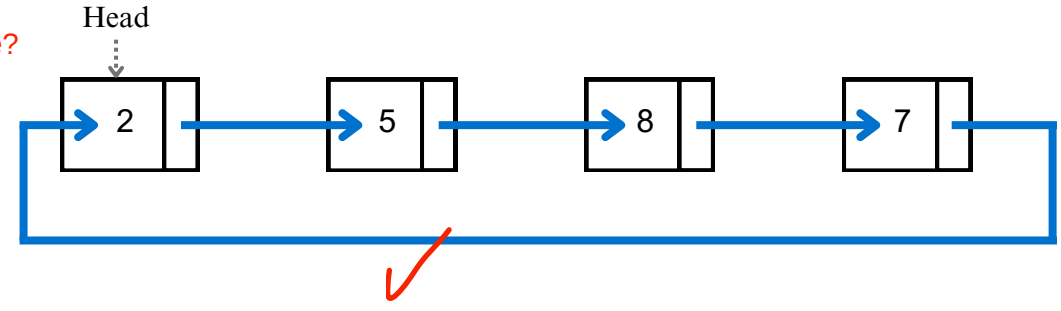
2. Doubly-Linked List:



In a doubly-linked list, each node has pointers to both the next and the previous nodes. The head pointer points to the first node (2).

3. Circularly-Linked List:

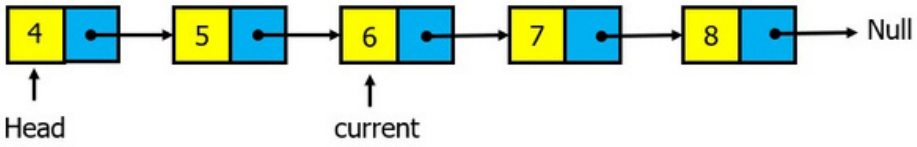
Dummy node?



In a circularly-linked list, the last node points back to the first node, forming a circle. The head pointer still points to the first node (2). There is no NULL at the end.

3.

Given the list below, describe in writing the steps needed to delete node 6. Your answer must also include the coding instructions to add or remove links between nodes (e.g., current.next = .)



Step-by-Step Process:

1. Start at the head node (which contains value 4 in this list)
2. Traverse the list until we find the node just before the one we want to delete (node 5)
3. Modify the next pointer of node 5 to point to node 7 (skipping node 6)
4. Free the memory occupied by node 6 or successfully delete it.

Coding instructions:

```
// We assume a basic node structure for this singly linked list.
   struct Node { int data; Node* next;};
// Function to delete node 6
   void deleteNode6(Node* head)
{
  // Start at the head
   Node* current = head;
  // Traverse to the node before the one to delete
   while (current->next != nullptr &&
          current->next->data != 6)
         {current = current->next;}
  // Check if the node to delete was found and storing a reference to node 6 as temp.
   if (current->next != nullptr) { Node* temp = current->next;
  // Linking the previous node to the next node, skipping the current node(6)
   current->next = current->next->next;
  // Freeing the memory(node 6)
   delete temp;}
}
```

One step missing

Explanation of the Code:

1.
```
Node* current = head;
```

A pointer current is initialized to the head of the list. This pointer will be used to traverse the list.

2.
```
while (current->next != nullptr && current->next->data != 6):
```

This loop does the following:

`current->next != nullptr`
Checks if we have not reached the end of the list

&&
In C++, this && (logical AND) operator has a property called "short-circuiting." This means that if the first operand of && evaluates to false, the second operand is not evaluated.

`current->next->data != 6`
Checks if the next node is not the one we want to delete.

3.
```
current = current->next;
```

Moves the current pointer to the next node in the list.

4.
```
if (current->next != nullptr)
```

This checks that we found the node to delete (i.e., the loop didn't just terminate because we reached the end of the list).

5.
```
Node* temp = current->next;
```

A temporary pointer(temp) stores the address of the node to be deleted (node 6).

6.
```
current->next = current->next->next;
```

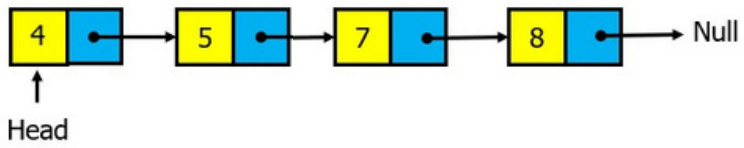
It changes the next pointer of the previous node (node 5) to point to the node after the one being deleted (node 7). This removes node 6 from the linked list.

7.
```
delete temp;
```

This line frees the memory that was allocated to node 6, preventing a memory leak.

Result:

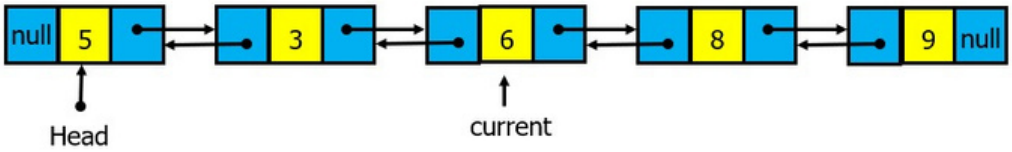
After executing this code, the linked list will be modified to:



Node 6 is no longer part of the list.

4.

Given the list below, describe in writing the steps necessary to add a new node with data=7 after the node indicated by the pointer "current".



Coding Instructions :

//Assume the node structure is

```
struct Node
{ int data; Node* prev; Node* next; };
```

The current is already pointing to the node with value 6.

// Step 1: Create and initialize the new node

```
Node* newNode = new Node;
newNode->data = 7;
```

// Step 2: Point newNode's next to current's next

```
newNode->next = current->next;
```

// Step 3: Point newNode's previous to current

```
newNode->prev = current;
```

// Step 4: If current->next is not null, update its previous to newNode

```
if (current->next != nullptr) {
current->next->prev = newNode; }
```

// Step 5: Finally, point current's next to newNode

```
current->next = newNode;
```

Steps-by- Step Proccess :

1. *Create Node:* Make space for a new node and set its data to 7.
2. *New Node's Next:* Point the new node's next to current's next.
3. *New Node's Previous:* Point the new node's previous to current.
4. *Update Next's Previous:* Update that node's previous to point to the new node.
5. *Current's Next:* Point current's next to the new node.

Explanation of the Code:

1.
```
Node* newNode = new Node;
```
Creates a new node.

2.
```
newNode->data = 7;
```
Assign 7 to the new node's data.

3.
```
newNode->next = current->next;
```
New node's 'next' points to where current's 'next' was pointing.

4.
```
newNode->prev = current;
```
New node's 'prev' points to current.

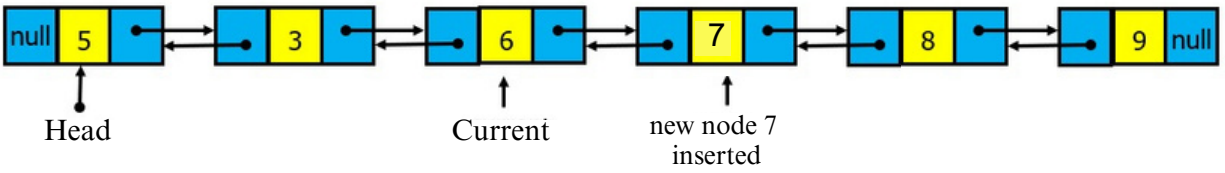
5.
```
if (current->next != nullptr) {
```
If current has a next node:

6.
```
current->next->prev = newNode;
```
Update the previous pointer of the next node

7.
```
} current->next = newNode;
```
Current's 'next' now points to the new node.

Results:

After these steps, the doubly-linked list will look like this;



Question 5 – Stacks and Queues:

1.

If you push the objects 7, 8, and 9 onto a stack that has numbers from 1 to 3 (3 is at the top), in what order will the pop operations remove them from the stack?

(i) **Initial State:**

top ← 3 2 1 → bottom

(ii) **Push the numbers 7, 8, and 9 onto the stack:**

push 7 → 7 3 2 1

push 8 → 8 7 3 2 1

push 9 → 9 8 7 3 2 1

✓ 3

- 9 is at the top
- Stacks operate on a Last-In, First-Out (LIFO) principle.

(iii) **Pop Operation:**

The elements will be removed from the stack in the following order:

$$9 \rightarrow 8 \rightarrow 7 \rightarrow 3 \rightarrow 2 \rightarrow 1$$

- The pop operations will remove the elements in the reverse order of how they were pushed, starting from the top.

2.

What pseudocode statements (push statements) create a stack of the three integers 1,2,3 in that order with 1 at the top?

To create a stack of the integers 1, 2, 3 in that order (with 1 at the top), we push the elements onto the stack in reverse order.

Pseudocode Statement:

```
stack.push(3);
stack.push(2);
stack.push(1);
```

Explanation:

`stack.push(3);` The integer 3 is pushed onto the stack. It becomes the bottom-most element.

`stack.push(2);` The integer 1 is pushed onto the stack. It is placed on top of 2, making it the top-most element.

`stack.push(1);` The integer 2 is pushed onto the stack. It is placed on top of 3.

Result:

Therefore, the stack will contain the elements in the desired order:
1 (top), 2 (middle), 3 (bottom).

3.

Suppose that a, b, and c are empty stacks and 1, 2, 3, and 4 are objects. What do the stacks contain after the following sequence of operations executes?

Initial State:
a: empty
b: empty
c: empty

a) a.push(1):
a: 1
b: empty
c: empty

b) a.push(2)
a: 2 (top), 1
b: empty
c: empty

c) b.push(3)
a: 2, 1
b: 3
c: empty

d) a.pop()
- *Removes 2 from stack a (top element)*

a: 1
b: 3
c: empty

e) c.push(4)
a: 1
b: 3
c: 4

f) c.push(a.pop())
- *a.pop( ) removes 1*
- *c.push(1) pushes 1 on top of c*

a: empty
b: 3
c: 1 (top), 4

g) b.push(c.pop())
- *c.pop( ) removes 1*
- *b.push(1) pushes 1 onto b*

a: empty
b: 1 (top), 3
c: 4

h) b.pop()
- *Removes 1 from stack b*

a: empty
b: 3
c: 4

i) a.push(c.pop())
- *c.pop( ) removes 4*
- *a.push(4) pushes 4 onto a*

a: 4
b: 3
c: empty

Final State:
a: 4
b: 3
c: empty

4.

If you enqueue the objects 7, 8, and 9 onto a list that has numbers from 1 to 3 (1 is at the front), in what order will the dequeue operations remove them from the list?

(i) **Initial State**:

Front ← 1 2 3 → Rear

- 1 at the front

(ii) **Enqueue the numbers 7, 8, and 9**:

1 2 3 4 5 6 7 ← Enqueue 7

1 2 3 4 5 7 8 ← Enqueue 8

1 2 3 7 8 9 ← Enqueue 9

- 7,8 & 9 are added to the rear of the queue.
- Queues operate on a First-In, First-Out (FIFO) principle.

(iii) **Dequeue Order:**

The elements will be dequeued in the following order.
$1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9$

- Therefore, the elements will be dequeued in the order they were enqueued.

5.

What pseudocode statements (enqueue statements) create a list of the three integers 1,2,3 in that order with 1 at the front?

To create a queue of the integers 1, 2, 3 in the order with 1 at the front, we enqueue the elements onto the set in the order we want them.

Pseudocode Statement:

```
myQueue.push(1);
myQueue.push(2);
myQueue.push(3);
```

Explanation:

`myQueue.push(1);` This enqueues the value 1 onto the queue. It becomes the Front of the queue.

`myQueue.push(2);` This enqueues the value 2, after its enqueued it's placed behind 1.

`myQueue.push(3);` This enqueues the value 3, after its enqueued it's placed behind 2. making it the Rear

Result:

Therefore, the queue will contain the integers in the desired order:
1 at the front and 3 at the rear.

6) Suppose that a, b, and c are empty lists and 1, 2, 3, and 4 are objects. What do the lists contain after the following sequence of operations executes?

Initial State:
Queue a. empty
Queue b. empty
Queue c. empty

Final State:
Queue a. 2, 1
Queue b. empty
Queue c. 2, 4, 3

a) a.enqueue (4)
Queue a. 4
Queue b. empty
Queue c. empty

b) b.enqueue (1)
Queue a. 4
Queue b. 1
Queue c. empty

c) b.enqueue (3)
Queue a. 4
Queue b. 1, 3
Queue c. empty

d) c.enqueue (2)
Queue a. 4
Queue b. 1, 3
Queue c. 2

e) a.enqueue[b.dequeue (1)]
Queue a. 4, 1
Queue b. 3
Queue c. 2

f) c.enqueue [a.dequeue (4)]
Queue a. 1
Queue b. 3
Queue c. 2, 4

g) c.enqueue [b.dequeue (3)]
Queue a. 1
Queue b. empty
Queue c. 2, 4, 3

h) a.enqueue(2)
Queue a. 1, 2
Queue b. empty
Queue c. 2, 4, 5

i) a.enqueue [a.dequeue (1)]
Queue a. 2,1
Queue b. empty
Queue c. 2, 4 ,3