III-I

DATA STRUCTURES

Linear Data Structures

Module 2 QB Solutions Handbook

Ujjwal | Vishal

MODULE 2

PART-A

Q1) The following postfix expression with single digit operands is evaluated using stack. 8 2 3 ↑ 2 3 * + 5 / * - Note that î s exponential operator. Find the top two elements of the stack after the first * Is it evaluated?

A. The top two element of the stack after the first*operator is evaluated are 6 and 1

Q2) Transform the following expression to a postfix expression using stacks. (A+B)*(C- (D-E)+F)-G

A. **Initialize two stacks**: one for operators and one for the output (postfix expression).

1. **Scan the infix expression** from left to right.
2. **Handle operands**: Directly add them to the output.
3. **Handle operators**: Push them onto the operator stack after popping higher or equal precedence operators from the stack to the output.
4. **Handle parentheses**: Push '(' onto the stack and pop all operators until '(' when encountering ')'.
5. **Pop all remaining operators** from the stack to the output after the entire expression is scanned.

PROGRAM

```
import java.util.Stack;

public class InfixToPostfix {
    public static void main(String[] args) {
        String infix = "(A+B)*(C-(D-E)+F)-G";
        System.out.println("Postfix expression: " + toPostfix(infix));
    }

    public static String toPostfix(String infix) {
```

```
    Stack<Character> operators = new Stack<>();
    StringBuilder postfix = new StringBuilder();

    for (int i = 0; i < infix.length(); i++) {
        char symbol = infix.charAt(i);

        if (Character.isLetterOrDigit(symbol)) {
            postfix.append(symbol);
        } else if (symbol == '(') {
            operators.push(symbol);
        } else if (symbol == ')') {
            while (!operators.isEmpty() && operators.peek() != '(') {
                postfix.append(operators.pop());
            }
            operators.pop(); // Remove '(' from stack
        } else {
            while (!operators.isEmpty() && precedence(symbol) <=
precedence(operators.peek())) {
                postfix.append(operators.pop());
            }
            operators.push(symbol);
        }
    }

    while (!operators.isEmpty()) {
        postfix.append(operators.pop());
    }

    return postfix.toString();
  }
```

```
    public static int precedence(char operator) {
        switch (operator) {
            case '+':
            case '-':
                return 1;
            case '*':
            case '/':
                return 2;
            default:
                return -1;
        }
    }
}
```

Q3) Convert the following expression A + (B * C) – ((D * E + F) / G) into postfix form.

A. we can follow these steps:

1. **Scan the infix expression from left to right.**
2. **Use a stack to keep operators and parentheses.**
3. **Output operands (letters) directly to the postfix expression.**
4. **Handle operators and parentheses according to their precedence and associativity.**

Program

```
import java.util.Stack;

public class InfixToPostfix {
    public static void main(String[] args) {
        String infix = "A + (B * C) - ((D * E + F) / G)";
        System.out.println("Postfix: " + infixToPostfix(infix));
```

}

Q4) To implement a queue using PUSH, POP and REVERSE operation, show how to implement ENQUEUE and DEQUEUE operations using a sequence of given operations?

A. ENQUEUE (Add an element to the end of the queue):

- Use the PUSH operation to add the element to the stack.
- If the stack is not empty, reverse the stack, push the element, and reverse it back.

2. DEQUEUE (Remove an element from the front of the queue):
   - Reverse the stack.
   - Use the POP operation to remove the element from the stack.
   - Reverse the stack back.

Program

```
import java.util.Stack;

public class QueueUsingStack {
    private Stack<Integer> stack;

    public QueueUsingStack() {
        stack = new Stack<>();
    }

    // ENQUEUE operation
    public void enqueue(int x) {
        stack.push(x);
    }

    // DEQUEUE operation
```

```
public int dequeue() {
    if (stack.isEmpty()) {
        throw new RuntimeException("Queue is empty");
    }
    reverseStack();
    int front = stack.pop();
    reverseStack();
    return front;
}

// Helper method to reverse the stack
private void reverseStack() {
    if (stack.isEmpty()) {
        return;
    }
    int top = stack.pop();
    reverseStack();
    insertAtBottom(top);
}

// Helper method to insert an element at the bottom of the stack
private void insertAtBottom(int x) {
    if (stack.isEmpty()) {
        stack.push(x);
        return;
    }
    int top = stack.pop();
    insertAtBottom(x);
    stack.push(top);
}
```

```
    public static void main(String[] args) {
        QueueUsingStack queue = new QueueUsingStack();
        queue.enqueue(1);
        queue.enqueue(2);
        queue.enqueue(3);
        System.out.println(queue.dequeue()); // Output: 1
        System.out.println(queue.dequeue()); // Output: 2
        queue.enqueue(4);
        System.out.println(queue.dequeue()); // Output: 3
        System.out.println(queue.dequeue()); // Output: 4
    }
}
```

Q5) The following postfix expression containing single digit operands and arithmetic operators + and * is evaluated using a stack. 5 2 * 3 4 + 5 2 * * + Show the content of the stack after evaluating the above expression.

A. **Initialize an empty stack**.

1. **Scan the postfix expression from left to right**.
2. **Push operands (numbers) onto the stack**.
3. **When an operator is encountered, pop the required number of operands from the stack, perform the operation, and push the result back onto the stack**

Let's go through the expression step by step:

1. **5**: Push 5 onto the stack.
    - Stack: `[5]`
2. **2**: Push 2 onto the stack.
    - Stack: `[5, 2]`
3. ***: Pop 2 and 5, multiply them (5 * 2 = 10), and push the result back onto the stack.
    - Stack: `[10]`
4. **3**: Push 3 onto the stack.

    - Stack: `[10, 3]`
5. **4**: Push 4 onto the stack.
    - Stack: `[10, 3, 4]`
6. **+**: Pop 4 and 3, add them (3 + 4 = 7), and push the result back onto the stack.
    - Stack: `[10, 7]`
7. **5**: Push 5 onto the stack.
    - Stack: `[10, 7, 5]`
8. **2**: Push 2 onto the stack.
    - Stack: `[10, 7, 5, 2]`
9. *: Pop 2 and 5, multiply them (5 * 2 = 10), and push the result back onto the stack.
    - Stack: `[10, 7, 10]`
10. *: Pop 10 and 7, multiply them (7 * 10 = 70), and push the result back onto the stack.
    - Stack: `[10, 70]`
11. **+**: Pop 70 and 10, add them (10 + 70 = 80), and push the result back onto the stack.
    - Stack: `[80]`

So, the final content of the stack after evaluating the expression is `[80]`.

Program

```
import java.util.Stack;


public class PostfixEvaluation {
    public static void main(String[] args) {
        String expression = "5 2 * 3 4 + 5 2 * * +";
        System.out.println("Result: " +
evaluatePostfix(expression));
    }
```

Q6) Evaluate the following postfix operation using a stack. 8 2 3 ↑ 2 3 * + 5 1 * - Where ↑ is the exponentiation operator.

A. To evaluate the given postfix expression `8 2 3 ↑ 2 3 * + 5 1 * -` using a stack in Java, you can follow these steps:

1. **Initialize an empty stack**.
2. **Scan the postfix expression from left to right**.
3. **Push operands (numbers) onto the stack**.
4. **When an operator is encountered**, pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.
5. **Continue until the end of the expression**. The final result will be the only element left in the stack.

Program

```
import java.util.Stack;

public class PostfixEvaluator {
    public static void main(String[] args) {
        String expression = "8 2 3 ↑ 2 3 * + 5 1 * -";
        System.out.println("Result: " + evaluatePostfix(expression));
    }

    public static double evaluatePostfix(String expression) {
        Stack<Double> stack = new Stack<>();
        String[] tokens = expression.split(" ");

        for (String token : tokens) {
            if (isNumeric(token)) {
                stack.push(Double.parseDouble(token));
            } else {
                double operand2 = stack.pop();
```

```
            double operand1 = stack.pop();
            switch (token) {
                case "+":
                    stack.push(operand1 + operand2);
                    break;
                case "-":
                    stack.push(operand1 - operand2);
                    break;
                case "*":
                    stack.push(operand1 * operand2);
                    break;
                case "/":
                    stack.push(operand1 / operand2);
                    break;
                case "^":
                    stack.push(Math.pow(operand1, operand2));
                    break;
                default:
                    throw new IllegalArgumentException("Invalid operator: " + token);
            }
        }
    }
    return stack.pop();
}

private static boolean isNumeric(String str) {
    try {
        Double.parseDouble(str);
        return true;
```

```
        } catch (NumberFormatException e) {
            return false;
        }
    }
}
```

Q7) Convert the following expression from infix to postfix notation. ((A + B) * C – (D – E) (F + G))

A. **Steps to Convert Infix to Postfix**

1. **Initialize an empty stack** for operators and an empty list for the postfix expression.
2. **Scan the infix expression** from left to right.
3. **If the scanned character is an operand**, add it to the postfix expression.
4. **If the scanned character is an operator**:
    - Pop from the stack to the postfix expression until the stack is empty or the top of the stack has an operator of lower precedence.
    - Push the scanned operator to the stack.
5. **If the scanned character is a left parenthesis** (, push it to the stack.
6. **If the scanned character is a right parenthesis**), pop from the stack to the postfix expression until a left parenthesis is encountered.
7. **Pop all the operators** from the stack to the postfix expression.

Program

```
import java.util.Stack;

public class InfixToPostfix {
    public static int precedence(char ch) {
        switch (ch) {
            case '+':
            case '-':
```

```
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
    }
    return -1;
}

public static String infixToPostfix(String expression) {
    StringBuilder result = new StringBuilder();
    Stack<Character> stack = new Stack<>();

    for (int i = 0; i < expression.length(); i++) {
        char c = expression.charAt(i);

        // If the scanned character is an operand, add it to output
        if (Character.isLetterOrDigit(c)) {
            result.append(c);
        }
        // If the scanned character is '(', push it to the stack
        else if (c == '(') {
            stack.push(c);
        }
        // If the scanned character is ')', pop and output from the stack
        // until an '(' is encountered
        else if (c == ')') {
            while (!stack.isEmpty() && stack.peek() != '(') {
```

```
                result.append(stack.pop());
            }
            stack.pop();
        }
        // An operator is encountered
        else {
            while (!stack.isEmpty() && precedence(c) <= precedence(stack.peek())) {
                result.append(stack.pop());
            }
            stack.push(c);
        }
    }

    // Pop all the operators from the stack
    while (!stack.isEmpty()) {
        if (stack.peek() == '(')
            return "Invalid Expression";
        result.append(stack.pop());
    }
    return result.toString();
  }

  public static void main(String[] args) {
    String expression = "((A+B)*C-(D-E)*(F+G))";
    System.out.println("Postfix expression: " + infixToPostfix(expression));
  }
}
```

Q8) Assume that the operators +, -, × are left associative and îs right associative. The order of precedence (from highest to lowest) is ˆx, +, -. The postfix expression corresponding to the infix expression a + b × c – d ê f ˆis

A. Program

```
import java.util.Stack;

public class InfixToPostfix {
    public static void main(String[] args) {
        String infix = "a + b * c - d ^ e ^ f";
        String postfix = infixToPostfix(infix);
        System.out.println("Postfix expression: " + postfix);
    }
```

Q9) Evaluate the postfix expression 1 2 + 3 * 6 + 2 3 + /

A. Program

```
import java.util.Stack;

public class PostfixEvaluation {
    public static void main(String[] args) {
        String expression = "1 2 + 3 * 6 + 2 3 + /";
        System.out.println("Result: " + evaluatePostfix(expression));
    }

    public static int evaluatePostfix(String expression) {
        Stack<Integer> stack = new Stack<>();
```

```
    String[] tokens = expression.split(" ");

    for (String token : tokens) {
        if (isOperator(token)) {
            int b = stack.pop();
            int a = stack.pop();
            int result = applyOperator(token, a, b);
            stack.push(result);
        } else {
            stack.push(Integer.parseInt(token));
        }
    }
    return stack.pop();
}

private static boolean isOperator(String token) {
    return token.equals("+") || token.equals("-") || token.equals("*") || token.equals("/");
}

private static int applyOperator(String operator, int a, int b) {
    switch (operator) {
        case "+":
            return a + b;
        case "-":
            return a - b;
        case "*":
            return a * b;
        case "/":
            return a / b;
```

```
            default:
                throw new IllegalArgumentException("Invalid operator: " + operator);
        }
    }
}
```

Q10) Evaluate the postfix expression 6 2 3 + - 3 8 2 / + * 2 + 3 +

A. Program

```
import java.util.Stack;

public class PostfixEvaluation {
    public static void main(String[] args) {
        String expression = "6 2 3 + - 3 8 2 / + * 2 + 3 +";
        System.out.println("Result: " + evaluatePostfix(expression));
    }

    public static int evaluatePostfix(String expression) {
        Stack<Integer> stack = new Stack<>();
        String[] tokens = expression.split(" ");

        for (String token : tokens) {
            if (isNumeric(token)) {
                stack.push(Integer.parseInt(token));
            } else {
                int operand2 = stack.pop();
                int operand1 = stack.pop();
                switch (token) {
                    case "+":
```

```
                    stack.push(operand1 + operand2);
                    break;
                case "-":
                    stack.push(operand1 - operand2);
                    break;
                case "*":
                    stack.push(operand1 * operand2);
                    break;
                case "/":
                    stack.push(operand1 / operand2);
                    break;
            }
        }
    }
    return stack.pop();
  }

  private static boolean isNumeric(String str) {
    try {
        Integer.parseInt(str);
        return true;
    } catch (NumberFormatException e) {
        return false;
    }
  }
}
```

PART B

Q1) Discuss the various operations performed on stack with examples.
A.

1. Various operations on Stacks:

=> Push (Stack, top, MaxStr, Item): This procedure pushes an Item onto a stack

i) if Top = MaxSize, then print overflow and return

ii) Set Top = Top + 1 [Increase Top by 1]

iii) Set Stack [TOP] = Item [Insert Item in TOP position]

iv) Return.

=> Pop (stack, top, Item): This procedure deletes the top element of stack and assign it to the variable Item.

i) if top=0, then print: underflow and return

ii) Set item = stack[Top]

iii) Set top = top-1 [decrease top by 1]

iv) return

Q2) Write down the algorithm to convert an infix expression to postfix form.
A.

2. Convertion of Infix to postfix:

i) Scan the expression from left to right character by character.

ii) if the scanned character is an operand output it i.e postfix it.

iii) Else:

1) If the precedence of input > precedence of the operator in the stack (precedence of top) or the stack is empty or the stack contains a '(' then push it.

2) Else Pop all the operators from the stack which are greater than or equal to in precedence than that of the inpu

After that push it into the stack.

(If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

an 'r' push it

-After that push it into the stack.
(If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is an '(' push it into the stack.

5. If the scanned character is ')', then pop the elements of the stack until the '(' is encountered and ignore the paranthesis in the output.

6. Repeat the step 2-5 untill the infix expression is scanned.

7. Pop all the elements of the stack and output it into the postfix expression.

Q3) Describe the operations of a stack using stacks using arrays.

A.

1. Various operations on stacks:

=> Push (Stack, top, MaxStr, Item): This procedure pushes an Item onto a stack

i) if Top = MaxSize, then print overflow and return

ii) Set Top = Top + 1 [Increase Top by 1]

iii) Set Stack [TOP] = Item [Insert Item in TOP position]

iv) Return.

=> Pop (stack, top, Item): This procedure deletes the top element of stack and assign it to the variable Item.

i) if top=0, then print: underflow and return

ii) Set item = stack[Top]

iii) Set top = top-1 [decrease top by 1]

iv) return

Q4) Write an algorithm for postfix expression evaluation.

A.

4. Algorith for post-fix expression evaluation:

i) Scan the post-fix expression from left to right untill the stack is empty.

ii) If operand then push it into stack.

iii) If operand is found then pop two operands and operator. After that push the result back to the stack.

iv) Repeat this untill the output is empty.

Example: 6523+8*+3+*

Symbol	op1	op2	value	stack
6				6
5				6,5
2				6,5,2
3				6,5,2,3
+	2	3	5	6,5,5
8				6,5,5,8

Example: 6523+8*+3+*

Symbol	op1	op2	value	stack
6				6
5				6,5
2				6,5,2
3				6,5,2,3
+	2	3	5	6,5,5
8				6,5,5,8
*	5	8	40	6,5,40
+	5	40	45	6,45
3				6,45,3
+	45	3	48	6,48
*	6	48	288	288.

6523+8*+3+* ⇒ 288.

Q5) Write the functional difference between stacks and queues.

A.

5. functional difference b/w stack and queue.

Stack	Queue.
i) The elements are inserted and deleted from same end.	i) The elements are inserted and removed at different ends.
ii) Only one pointer is used and that pointer points to the top of stack.	ii) TWO different pointers are used for pointing the rear and front ends.
iii) It follows last in first out principle.	ii) It follows first in first out principle.
iv) Stack operations are push and pop	iv) Queue operations are enqueue and dequeue.
v) Visualized as vertical collections.	v) Visualized as horizontal collections.
vi) Used for depth first search	vi) Used for breadth first search.

Q6) Compare between linear queue and circular queue? Write down algorithms for insert and delete operations in a circular queue?

A.

6.

Linear Queue:	Circular Queue:
1. In this data and instructions are organized in a sequential order one after one.	1. The data and instructions are organized in a circular order.
2. It follows first in first out.	2. It doesnot have any specific order.
3. linear Queue is not much efficient.	3. Circular queue is efficient

Algorithm for insertion:

Inserting an element into a circular queue results in $R = (R+1)\%$ max where max is the maximum size of the array.

full $\Rightarrow$ $c = max$

$c = c + 1$

Algorithm for deleting:

Deleting an element from a circular queue results in $f = (f+1)\%$ max where max is the maximum size of the array

C = count

if $c = 0$ $\Rightarrow$ empty

$c = c - 1$.

Q7) Define a double ended queue (DEQUE). Explain input restricted and output restricted DEQUE.

A.

7. DEQUE (Double ended queue):

It is a special queue like data structure that supports insertion and deletion at both the front and rear ends. Such an extension for the queue is called double ended queue.

Two types of DEQUE:

=> Input restricted: This allows insertion at only one end but deletion takes place at both ends.

=> Output restricted: This allows insertion at both the ends but deletion at only one end.

Q8) Explain the concept of a linear queue. Write algorithms for performing insert, delete operations using arrays.

A.

8. Linear Queue :

A queue is a data structure where items are inserted and deleted at one end called rear end and deleted at other one end called front end and it follows the principle first in first out.

Operations on Queue :

Enqueue (value) : This is a user defined function used to insert an element into circular queue. In circular queue the new element is always inserted at rear position.

Steps:

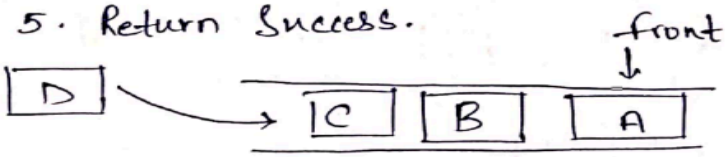
1. check if the queue is full [R = len(queue)]
2. If the queue is full, produe over-flow andexit..
3. If the queue is not full, increment rear pointer to point the next empty space.

Steps:

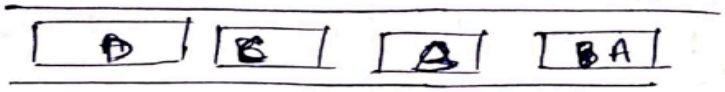
1. check if the queue is full [R = len(queue)]
2. If the queue is full, produe over-flow andexit.
3. If the queue is not full, increment rear pointer to point the next empty space.
4. Add the data to the queue where the rear is pointing.
5. Return Success.

Algorithm:

```
def enqueue:
    if queue isfull:
        return over flow
    else:
        r = r+1
        queue[r] = data
        return true.
```



Before



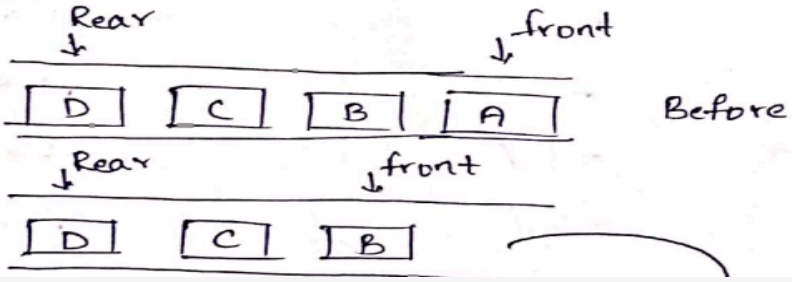
After

Dequeue Operation:

This is used to delete an element from the queue. In a queue, the element is always deleted from the front end.

Steps:

1. check if the queue is empty or not.
2. if the queue is empty, produce underflow and exit.
3. If the queue is not empty, access the data where front is pointing.
4. Increment front pointer to point to the next data element.
5. Return Success.




```
Import java.util.ArrayDeque;
Import java.util.Deque;
Public class DequeCreation{
    Public static void main(String args[]) {
        Deque<String>deque = new ArrayDeque<>();
        System.out.println("Deque elements:"+ deque);
}
}
```

Q9) Write the procedure for Circular Queue full and empty conditions.

A. Circular Queue Full Condition

A circular queue is considered full when the next position of the rear pointer is the front pointer. This can be checked using the following conditions:

Condition 1: (rear + 1) % size == front
Condition 2: front == 0 && rear == size - 1

Circular Queue Empty Condition

A circular queue is considered empty when the front and rear pointers are both set to -1. This can be checked using the following condition:

1. **Condition**: `front == -1 && rear == -1`

Procedures

Enqueue Operation (Insert Element)

1. **Check if the queue is full**:
   - If `(rear + 1) % size == front`, then the queue is full.
2. **If the queue is not full**:
   - If the queue is empty (`front == -1`), set `front = 0`.
   - Increment the rear pointer: `rear = (rear + 1) % size`.
   - Insert the new element at the position pointed to by `rear`.

Dequeue Operation (Remove Element)

1. **Check if the queue is empty**:
   - If `front == -1`, then the queue is empty.
2. **If the queue is not empty**:
   - Retrieve the element at the position pointed to by `front`.
   - If `front == rear`, set both `front` and `rear` to -1 (queue becomes empty).
   - Otherwise, increment the front pointer: `front = (front + 1) % size`.

Q10) Write the equivalent prefix and postfix expression for the given infix expression: (a * b) / 2 – (c / d – e)

A.

10. Given in-fix expression (a * b) / 2 - (c / d - e)

In-fix ⟶ post-fix.

Symbol	stack	Postfix
(	(	
a	(	a
*	(*	a
b		ab
)		ab*
/	/	ab*
2	/	ab*2
-	-	ab*2/
(	-(	ab*2/
c	-(	ab*2/c
/	-(/	ab*2/c
d	-(/	ab*2/cd
-	-(-	ab*2/cd/
e	-(-	ab*2/cd/e
)	-	ab*2/cd/e-
		ab*2/cd/e--

Post fix expression is : ab*2/cd/e--

Q11) Convert following infix expression into postfix form: (A+B) * (C-D/E)* G+H

A.

11. Infix expression: (A+B)*(C-D/E)*G+H.

Infix ⟶ Postfix.

Symbol	Stack	Post-fix
(	(	
A	(	A
+	(+	A
B	(+	AB
)		AB+
*	*	AB+
(	*(	AB+
C	*(	AB+C
-	*(-	AB+C
D	*(-	AB+CD
/	*(-/	AB+CD
E	*(-/	AB+CDE
)	*	AB+CDE/-
*	*	AB+CDE/-*
G		AB+CDE/-*G
+	+	AB+CDE/-*G*
H.	+	AB+CDE/-*G*H

AB+CDE/-*G*H+

The postfix expression is : AB+CDE/-*G*H+

Q12) Evaluate the following postfix notation of expression (Show status of stack after execution of each operations): 5 20 15 - * 25 2 * +

A.

12: Post fix expression. 5 20 15 - * 25 2 * +

Symbol	op1	op2	value	stack.
5				5
20				5, 20
15				5, 20, 15
-	20	15	5	5, 5
*	5	5	25	25
25				25, 25
2				25, 25, 2
*	25	2	50	25, 50
+	25	50	75	75
				75.

5 20 15 - * 25 2 * + = 75.

Q13) Convert the following infix expression to postfix expression using a stack using the usual precedence rule: x + y * z + (p * q + r) * s

A.

13: In-fix expression: $x+y*z+(p*q+r)*s$

In-fix ⟶ post-fix

Symbol	stack	postfix
x		x
+	+	
y	+	xy
*	+*	xy
z	+*	xyz
+	+	xyz*+
(	+(	xyz*+
p	+(	xyz*+p

Scanned by CamScanner

*	+(*	xyz*+p
q	+(*	xyz*+pq
+	+(+	xyz*+pq*
r	+(+	xyz*+pq*r
)	+	xyz*+pq*r+
*	+*	xyz*+pq*r+
s	+*	xyz*+pq*r+s

xyz*+pq*r+s*+

∴ The post-fix expression is : xyz*+pq*r+s*+

Q14) Find the result of evaluating the postfix expression 5, 4, 3, +, *, 4, 9, 3, /,+,*

A.

14: 5 4 3 + * 4 9 3 / + *

Symbol	op1	op2	value	stack
5				5
4				5, 4
3				5, 4, 3
+	4	3	7	5, 7
*	5	7	35	35
4				35, 4
9				35, 4, 9
3				35, 4, 9, 3
/	9	3	3	35, 4, 3
+	4	3	7	35, 7
*	35	7	245	245

∴ 5 4 3 + * 4 9 3 / + * = 245. 245

Q15) Convert following infix expression into postfix form: A + (B*C-D/E*G) + H

A.

15. Infix expression: A + (B * C - D / E * G) + H

Infix → postfix.

Symbol	Stack	postfix
A		A
+	+	A
(	+ (	A
B	+ (	AB
*	+ (*	AB
C	+ (*	ABC
-	+ (-	ABC *
D	+ (-	ABC * D
/	+ (- /	ABC * D
E	+ (- /	ABC * DE
*	+ (- *	ABC * DE /
G	+ (- *	ABC * DE / G
)	+	ABC * DE / G * -
+	+	ABC * DE / G * - +
H	+	ABC * DE / G * - + H
		ABC * DE / G * - + H +

∴ Postfix expression is : ABC * DE / G * - + H +

Q16) Implement an algorithm to DEQUEUE delete from front operation

A.

16. Deque delete from front end:

a) first check deque is empty or not.

b) If deque has only one element

front = -1 ; rear = -1

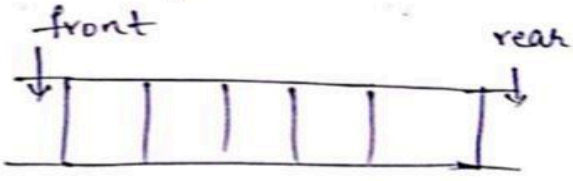
else if front points to the last index of the array it means we have no more elements in array so we have to move front to (points) first index of array

front = 0;

else increment front by 1

front = front + 1;

```
if ( front == -1)
    print ("queue underflow")
    return
print ("Element deleted is :" deque - array [front]);
if (front == rear):
    front = -1 ; rear = -1
else if (front == max-1)
    front = 0
else:
    front = front +1
```



Q17) Implement an algorithm to DEQUEUE delete from rear operation

A.

17) Deque Deletion from Rear end:

a) first check deque is empty or not.

b) If deque has only one element

front = -1; rear = -1

else if rear points to first index of list it means we have to move rear to last index [now first inserted element at front end becomes rear end]

else decrease rear by 1

rear = rear - 1.

Q18) Implement an algorithm to DEQUEUE insert at front operation

A.

18. Deque Insert element at front end:

a) first check deque is full or not.

b) If front == 0 // initial position move front to points last index of list.

front = size - 1

else decrement front by 1 and push current key value into

list [front] = key

Rear remains same.

Q19) Implement an algorithm to DEQUEUE insert at rear operation

A.

19. Insert element at rear end:

a) first we check degue is full or not.

b) If rear = size -1 then intialize

rear = 0 and

else increment rear by 1

and push current key into Arr[rear] = key

front remains same.

Q20) Write the conditions for Queue full and empty conditions.

A.

20. If Queue full:

```
if (rear = N-1)
    print ("overflow")
```

N = len(list) -1

then it shows queue is full.

If Queue is empty:

```
if (front == -1) or (front == 0)
    print ("underflow")
```

then it shows queue is empty.