DAY-4

NAME:GOWTHAMI MOPURI

REG.NO:192311287

1.INFIX TO POSTFIX:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}

char associativity(char c) {
    if (c == '^')
        return 'R';
    return 'L'; // Default to left-associative
}
```

```
void infixToPostfix(char s[]) {
    char result[1000];
    int resultIndex = 0;
    int len = strlen(s);
    char stack[1000];
    int stackIndex = -1;

    for (int i = 0; i < len; i++) {
        char c = s[i];



        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9')) {
            result[resultIndex++] = c;
        }

        else if (c == '(') {
            stack[++stackIndex] = c;
        }

        else if (c == ')') {
            while (stackIndex >= 0 && stack[stackIndex] != '(') {
                result[resultIndex++] = stack[stackIndex--];
            }
            stackIndex--;
        }

        else {
            while (stackIndex >= 0 && (prec(s[i]) < prec(stack[stackIndex]) ||
```

```
                                                prec(s[i]) == prec(stack[stackIndex]) &&
                                                    associativity(s[i]) == 'L')) {
                        result[resultIndex++] = stack[stackIndex--];
                    }
                    stack[++stackIndex] = c;
                }
            }


            while (stackIndex >= 0) {
                result[resultIndex++] = stack[stackIndex--];
            }

            result[resultIndex] = '\0';
            printf("%s\n", result);
}



int main() {
            char exp[] = "a+b*(c^d-e)^(f+g*h)-i";



            infixToPostfix(exp);

            return 0;
}
```

Output:

abcd^e-fgh*+^*+i-

2.ARRAY IMPLEMENTATION FOR QUEUE:

```
#include <stdio.h>
#include <stdlib.h>

struct Queue {
    int front, rear, capacity;
    int* queue;
};

// Function to initialize the queue
struct Queue* createQueue(int capacity) {
    struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
    q->capacity = capacity;
    q->front = 0;
    q->rear = -1;
    q->queue = (int*)malloc(q->capacity * sizeof(int));
    return q;
}

// Function to insert an element at the rear of the queue
void enqueue(struct Queue* q, int data) {
    // Check if the queue is full
    if (q->rear == q->capacity - 1) {
        printf("Queue is full\n");
        return;
```

```
    }

    // Insert element at the rear
    q->queue[++q->rear] = data;
}

// Function to delete an element from the front of the queue
void dequeue(struct Queue* q) {
    // If the queue is empty
    if (q->front > q->rear) {
        printf("Queue is empty\n");
        return;
    }

    // Shift all elements from index 1 till rear to the left by one
    for (int i = 0; i < q->rear; i++) {
        q->queue[i] = q->queue[i + 1];
    }

    // Decrement rear
    q->rear--;
}

// Function to print queue elements
void display(struct Queue* q) {
    if (q->front > q->rear) {
        printf("Queue is Empty\n");
        return;
```

```
    }

    // Traverse front to rear and print elements
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d <-- ", q->queue[i]);
    }
    printf("\n");
}

// Function to print the front of the queue
void front(struct Queue* q) {
    if (q->rear == -1) {
        printf("Queue is Empty\n");
        return;
    }
    printf("Front Element is: %d\n", q->queue[q->front]);
}

// Driver code
int main() {
    // Create a queue of capacity 4
    struct Queue* q = createQueue(4);

    // Print queue elements
    display(q);

    // Insert elements in the queue
    enqueue(q, 20);
```

```
enqueue(q, 30);
enqueue(q, 40);
enqueue(q, 50);

// Print queue elements
display(q);

// Insert element in the queue
enqueue(q, 60);

// Print queue elements
display(q);

// Dequeue elements
dequeue(q);
dequeue(q);

printf("After two node deletions\n");

// Print queue elements
display(q);

printf("After one insertion\n");
enqueue(q, 60);

// Print queue elements
display(q);
```

```
    // Print front of the queue
    front(q);

    // Free the allocated memory
    free(q->queue);
    free(q);

    return 0;
}
```

Output:

```
Queue is Empty
 20 <--  30 <--  40 <--  50 <--

Queue is full
 20 <--  30 <--  40 <--  50 <--

After two node deletions
 40 <--  50 <--

After one insertion
 40 <--  50 <--  60 <--

Front Element...
```

3.LINKED LIST IMPLEMENTATION FOR QUEUE:

```
// A C program to demonstrate linked list based
// implementation of queue
#include <stdio.h>
```

```
#include <stdlib.h>

// A linked list (LL) node to store a queue entry
struct QNode {
	int key;
	struct QNode* next;
};

// The queue, front stores the front node of LL and rear
// stores the last node of LL
struct Queue {
	struct QNode *front, *rear;
};

// A utility function to create a new linked list node.
struct QNode* newNode(int k)
{
	struct QNode* temp
		= (struct QNode*)malloc(sizeof(struct QNode));
	temp->key = k;
	temp->next = NULL;
	return temp;
}

// A utility function to create an empty queue
struct Queue* createQueue()
{
	struct Queue* q
```

```
                = (struct Queue*)malloc(sizeof(struct Queue));
        q->front = q->rear = NULL;
        return q;
}

// The function to add a key k to q
void enQueue(struct Queue* q, int k)
{
        // Create a new LL node
        struct QNode* temp = newNode(k);

        // If queue is empty, then new node is front and rear
        // both
        if (q->rear == NULL) {
                q->front = q->rear = temp;
                return;
        }

        // Add the new node at the end of queue and change rear
        q->rear->next = temp;
        q->rear = temp;
}

// Function to remove a key from given queue q
void deQueue(struct Queue* q)
{
        // If queue is empty, return NULL.
        if (q->front == NULL)
```

```
        return;

    // Store previous front and move front one node ahead
    struct QNode* temp = q->front;

    q->front = q->front->next;

    // If front becomes NULL, then change rear also as NULL
    if (q->front == NULL)
        q->rear = NULL;

    free(temp);
}

// Driver code
int main()
{
    struct Queue* q = createQueue();
    enQueue(q, 10);
    enQueue(q, 20);
    deQueue(q);
    deQueue(q);
    enQueue(q, 30);
    enQueue(q, 40);
    enQueue(q, 50);
    deQueue(q);
    printf("Queue Front : %d \n", ((q->front != NULL) ? (q->front)->key : -1));
    printf("Queue Rear : %d", ((q->rear != NULL) ? (q->rear)->key : -1));
```

```
        return 0;
}
```

Output:

queue front: 40

queue rear: 50