QUEUE and Stack

1) Array Implementation:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5
typedef struct Queue {
    int items[MAX];
    int front;
    int rear;
} Queue;
void initQueue(Queue* q) {
    q->front = -1;
    q->rear = -1;
}
int isEmpty(Queue* q) {
    return q->front == -1;
}
int isFull(Queue* q) {
    return (q->rear + 1) % MAX == q->front;
}
void enqueue(Queue* q, int value) {
    if (isFull(q)) {
        printf("Queue is full! Cannot enqueue %d\n", value);
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
    }
    q->rear = (q->rear + 1) % MAX;
    q->items[q->rear] = value;
    printf("Enqueued: %d\n", value);
}
int dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty! Cannot dequeue\n");
        return -1;
    }
    int value = q->items[q->front];
    if (q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX;
    }
```

```
    printf("Dequeued: %d\n", value);
    return value;
}
void display(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = q->front; i != q->rear; i = (i + 1) % MAX) {
        printf("%d ", q->items[i]);
    }
    printf("%d\n", q->items[q->rear]);
}
int main() {
    Queue q;
    initQueue(&q);
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    display(&q);
    dequeue(&q);
    display(&q);
    enqueue(&q, 40);
    enqueue(&q, 50);
    display(&q);
    enqueue(&q, 60);
    dequeue(&q);
    dequeue(&q);
    display(&q);
    return 0;
}
```

OUTPUT:

```
Enqueued: 10
Enqueued: 20
Enqueued: 30
Queue elements: 10 20 30
Dequeued: 10
Queue elements: 20 30
Enqueued: 40
Enqueued: 50
Queue elements: 20 30 40 50
Enqueued: 60
Dequeued: 20
Dequeued: 30
Queue elements: 40 50 60
```

2) Linked List Implementation:

Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int data;
    struct Node* next;
} Node;
typedef struct Queue {
    Node* front;
    Node* rear;
} Queue;
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
Queue* createQueue() {
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    if (!queue) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    queue->front = NULL;
    queue->rear = NULL;
    return queue;
}
int isEmpty(Queue* queue) {
    return queue->front == NULL;
}
void enqueue(Queue* queue, int data) {
    Node* newNode = createNode(data);
    if (isEmpty(queue)) {
        queue->front = newNode;
        queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
    printf("%d enqueued to queue\n", data);
}
```

```
int dequeue(Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty, cannot dequeue\n");
        return -1;
    }
    Node* temp = queue->front;
    int data = temp->data;
    queue->front = queue->front->next;
    if (queue->front == NULL) {
        queue->rear = NULL;
    }
    free(temp);
    printf("%d dequeued from queue\n", data);
    return data;
}
int front(Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        return -1;
    }
    return queue->front->data;
}
void freeQueue(Queue* queue) {
    while (!isEmpty(queue)) {
        dequeue(queue);
    }
    free(queue);
}
int main() {
    Queue* queue = createQueue();
    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    printf("Front item is %d\n", front(queue));
    dequeue(queue);
    dequeue(queue);
    printf("Front item is %d\n", front(queue));
    dequeue(queue);
    dequeue(queue);
    freeQueue(queue);
    return 0;
}
```

OUTPUT:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
```

```
Front item is 10
10 dequeued from queue
20 dequeued from queue
Front item is 30
30 dequeued from queue
Queue is empty, cannot dequeue
```

3) Write c program code to implement infix prefix and postfix rotation for arithmetic expression using stack

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
typedef struct {
    int top;
    unsigned capacity;
    char *array;
} Stack;
Stack* createStack(unsigned capacity) {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (char*)malloc(stack->capacity * sizeof(char));
    return stack;
}
int isFull(Stack* stack) {
    return stack->top == stack->capacity - 1;
}
int isEmpty(Stack* stack) {
    return stack->top == -1;
}
void push(Stack* stack, char item) {
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
}
char pop(Stack* stack) {
    if (isEmpty(stack))
        return '\0';
    return stack->array[stack->top--];
}
char peek(Stack* stack) {
    if (isEmpty(stack))
        return '\0';
    return stack->array[stack->top];
```

```
}
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}
void infixToPostfix(char* infix, char* postfix) {
    Stack* stack = createStack(strlen(infix));
    int i = 0, k = 0;
    for (i = 0; infix[i]; i++) {
        if (isalnum(infix[i])) {
            postfix[k++] = infix[i];
        } else if (infix[i] == '(') {
            push(stack, infix[i]);
        } else if (infix[i] == ')') {
            while (!isEmpty(stack) && peek(stack) != '(') {
                postfix[k++] = pop(stack);
            }
            pop(stack);
        } else if (isOperator(infix[i])) {
            while (!isEmpty(stack) && precedence(peek(stack)) >= precedence(infix[i])) {
                postfix[k++] = pop(stack);
            }
            push(stack, infix[i]);
        }
    }
    while (!isEmpty(stack)) {
        postfix[k++] = pop(stack);
    }
    postfix[k] = '\0';
    free(stack->array);
    free(stack);
}
void reverse(char* str) {
    int length = strlen(str);
    for (int i = 0; i < length / 2; i++) {
        char temp = str[i];
        str[i] = str[length - i - 1];
        str[length - i - 1] = temp;
    }
}
void infixToPrefix(char* infix, char* prefix) {
    int length = strlen(infix);
    reverse(infix);
    for (int i = 0; i < length; i++) {
```

```
        if (infix[i] == '(') {
            infix[i] = ')';
        } else if (infix[i] == ')') {
            infix[i] = '(';
        }
    }
    char postfix[length + 1];
    infixToPostfix(infix, postfix);
    reverse(postfix);
    strcpy(prefix, postfix);
}
int main() {
    char infix[100], postfix[100], prefix[100];
    printf("Enter infix expression: ");
    scanf("%s", infix);
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);
    infixToPrefix(infix, prefix);
    printf("Prefix expression: %s\n", prefix);
    return 0;
}
```

OUTPUT:

```
Enter infix expression: *+5
Postfix expression: *5+
Prefix expression: +*5
```