Arrays

1. Insertion

```
void insert(int arr[], int size,int n,int index){
    if (size>=100){
        printf("Insertion not possible");
    }else{
        for(int i=size-1;i>=index;i--){
            arr[i+1]=arr[i];
        }
        arr[index]=n;
    }

}
```

2. Deletion

```
void delete(int arr[],int size,int index){
    for(int i=index;i<=size-1;i++){
        arr[i]=arr[i+1];
    }
}
```

Linked Lists

1. Insertion

```
struct Node* insertAtFirst(struct Node* head, int data){
    struct Node* ptr=(struct Node*)malloc(sizeof(struct Node*));
    ptr->next=head;
    ptr->data=data;
    return ptr;
}

struct Node* insertAtIndex(struct Node* head, int data,int
index){
    struct Node* ptr=(struct Node*)malloc(sizeof(struct Node*));
    ptr->data=data;
    struct Node* p=head;
    int i=0;
    while(i!=index-1){
        p=p->next;
        i++;
    }
    ptr->next=p->next;
    p->next=ptr;
    return head;
}

struct Node* insertAtEnd(struct Node* head, int data){
    struct Node* ptr=(struct Node*)malloc(sizeof(struct Node*));
    ptr->data=data;
    struct Node* p=head;
    while(p->next!=NULL){
        p=p->next;
    }
    ptr->next=NULL;
    p->next=ptr;
    return head;
}
```

2. Deletion

```
struct Node* deleteAtFirst(struct Node* head){
    struct Node* ptr=head;
    head=head->next;
    free(ptr);
    return head;
}

struct Node* deleteAtIndex(struct Node* head,int index){
    struct Node* p=head;
    struct Node* q=head->next;
    for(int i=0;i<index-1;i++){
        q=q->next;
        p=p->next;
    }
    p->next=q->next;
    free(q);
    return head;
}

struct Node* deleteAtEnd(struct Node* head){
    struct Node* p=head;
    struct Node* q=head->next;
    while(q->next!=NULL){
        q=q->next;
        p=p->next;
    }
    p->next=NULL;
    free(q);
    return head;
}
```

3. Traversal

```
void display(struct Node* ptr){
    while(ptr!=NULL){
        printf("%d\t",ptr->data);
        ptr = ptr->next;
    }
}
```

Stack

1. Insertion

```
void push(struct stack* ptr,char val){
    if(isFull(ptr)==1){
        printf("Stack is full");
    }else{
        ptr->top++;
        ptr->arr[ptr->top]=val;
    }
}
```

2. Deletion

```
int pop(struct stack* ptr){
    if(isEmpty(ptr)==1){
        printf("Stack is Empty");
        return -1;
    }else{
        int val=ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
```

3. Empty/Full

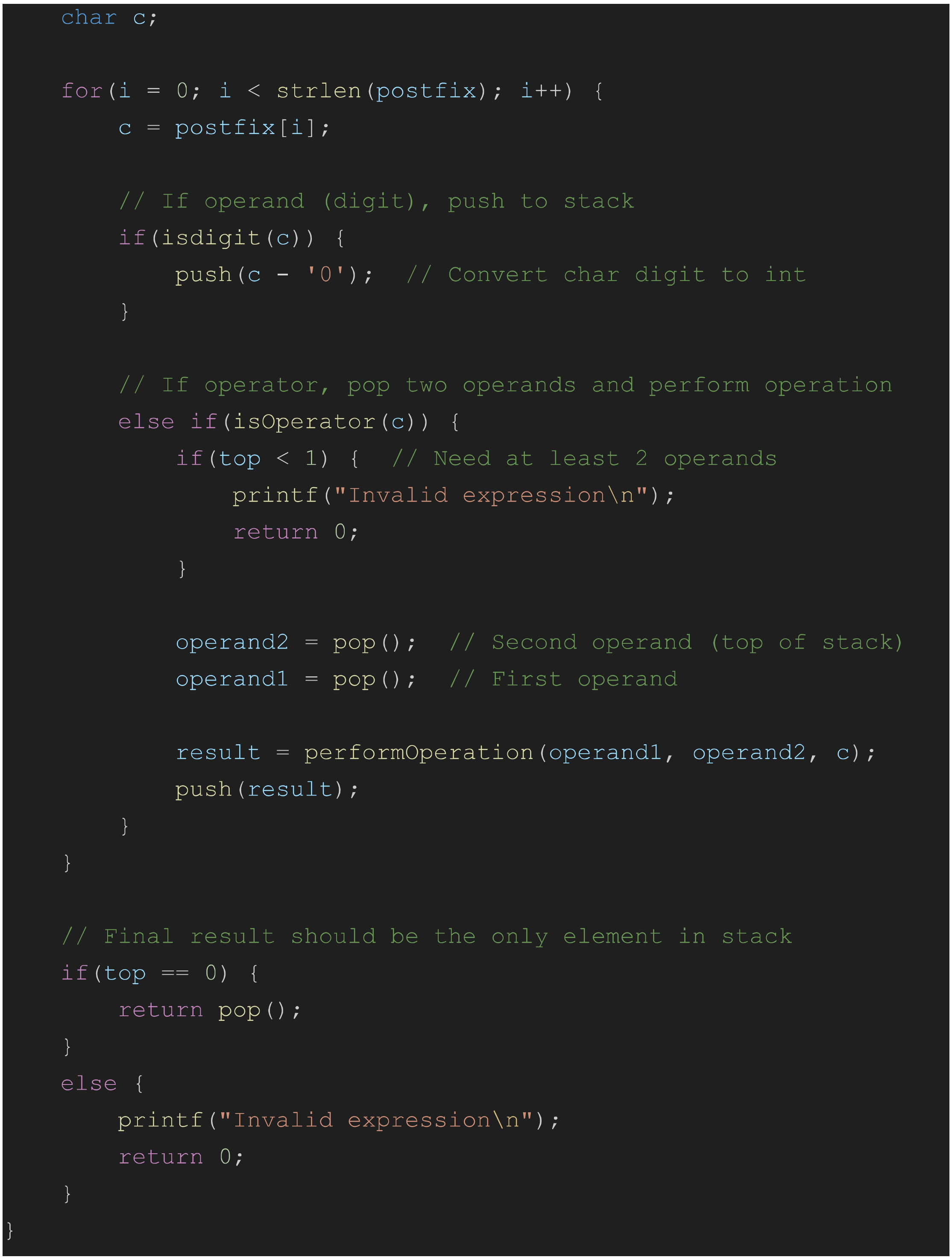
```
int isEmpty(struct stack* ptr){
    if(ptr->top==-1){
        return 1;
    }else{
        return 0;
    }
}

int isFull(struct stack* ptr){
    if(ptr->top==(ptr->size)-1){
        return 1;
    }else{
        return 0;
    }
}
```

4. Infix to Postfix Conversion

```
int prec(char ch){
    if(ch=='*' || ch=='/'){
        return 3;
    }else if(ch=='+' || ch=='-'){
        return 2;
    }else{
        return 0;
    }
}

int isOperator(char ch){
    if(ch=='/' || ch=='*' || ch=='+' || ch=='-'){
        return 1;
    }else{
        return 0;
    }
}

char* infixConversion(char* infix){
    struct stack* sp=(struct stack*)malloc(sizeof(struct stack));
    sp->size=100;
    sp->top=-1;
    sp->arr=(char*)malloc(sp->size*sizeof(char));
    char* postfix=(char*)malloc(100*sizeof(char));
    int i=0; //Track infix expression
    int j=0; //track postfix operations
    while(infix[i]!='\0'){
        if(isOperator(infix[i])!=1){
            postfix[j]=infix[i];
            i++;
            j++;
        }else{
            if(prec(infix[i])>prec(stackTop(sp))){
                push(sp,infix[i]);
                i++;
            }else{
                postfix[j]=pop(sp);
                j++;
            }
        }
    }
    while(isEmpty(sp)!=1){
        postfix[j]=pop(sp);
        j++;
    }
    postfix[j]='\0';
    return postfix;
}
```

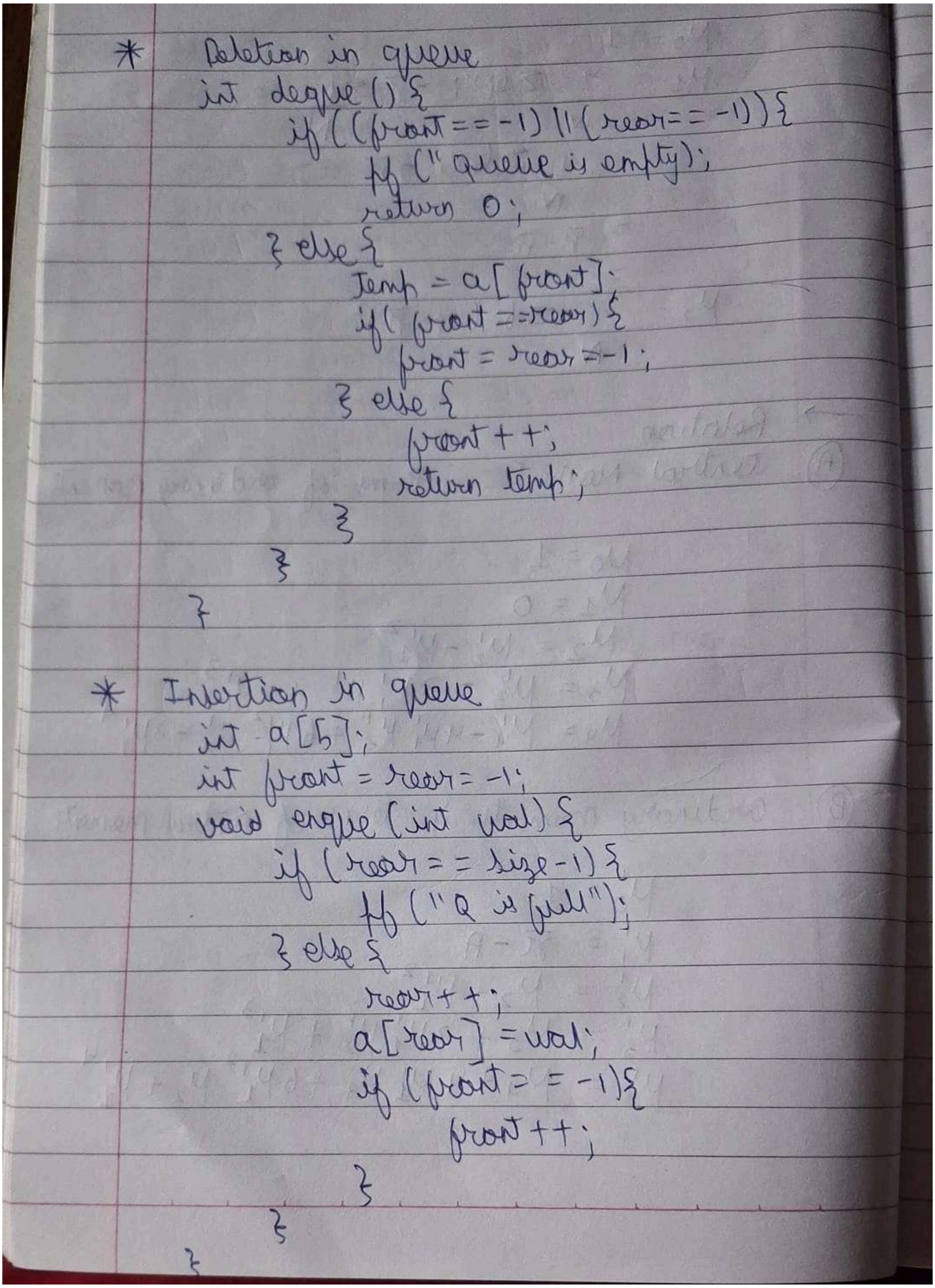
5. Postfix Evaluation

```
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/' || c ==
'^');
}

int performOperation(int op1, int op2, char operator) {
    switch(operator) {
        case '+': return op1 + op2;
        case '-': return op1 - op2;
        case '*': return op1 * op2;
        case '/':
            if(op2 == 0) {
                printf("Division by zero error\n");
                return 0;
            }
            return op1 / op2;
        case '^': return (int)pow(op1, op2);
        default: return 0;
    }
}

int evaluatePostfix(char postfix[]) {
    int i, operand1, operand2, result;
```



```c
    char c;

    for(i = 0; i < strlen(postfix); i++) {
        c = postfix[i];

        // If operand (digit), push to stack
        if(isdigit(c)) {
            push(c - '0');  // Convert char digit to int
        }

        // If operator, pop two operands and perform operation
        else if(isOperator(c)) {
            if(top < 1) {  // Need at least 2 operands
                printf("Invalid expression\n");
                return 0;
            }

            operand2 = pop();  // Second operand (top of stack)
            operand1 = pop();  // First operand

            result = performOperation(operand1, operand2, c);
            push(result);
        }
    }

    // Final result should be the only element in stack
    if(top == 0) {
        return pop();
    }
    else {
        printf("Invalid expression\n");
        return 0;
    }
}
```

Queue



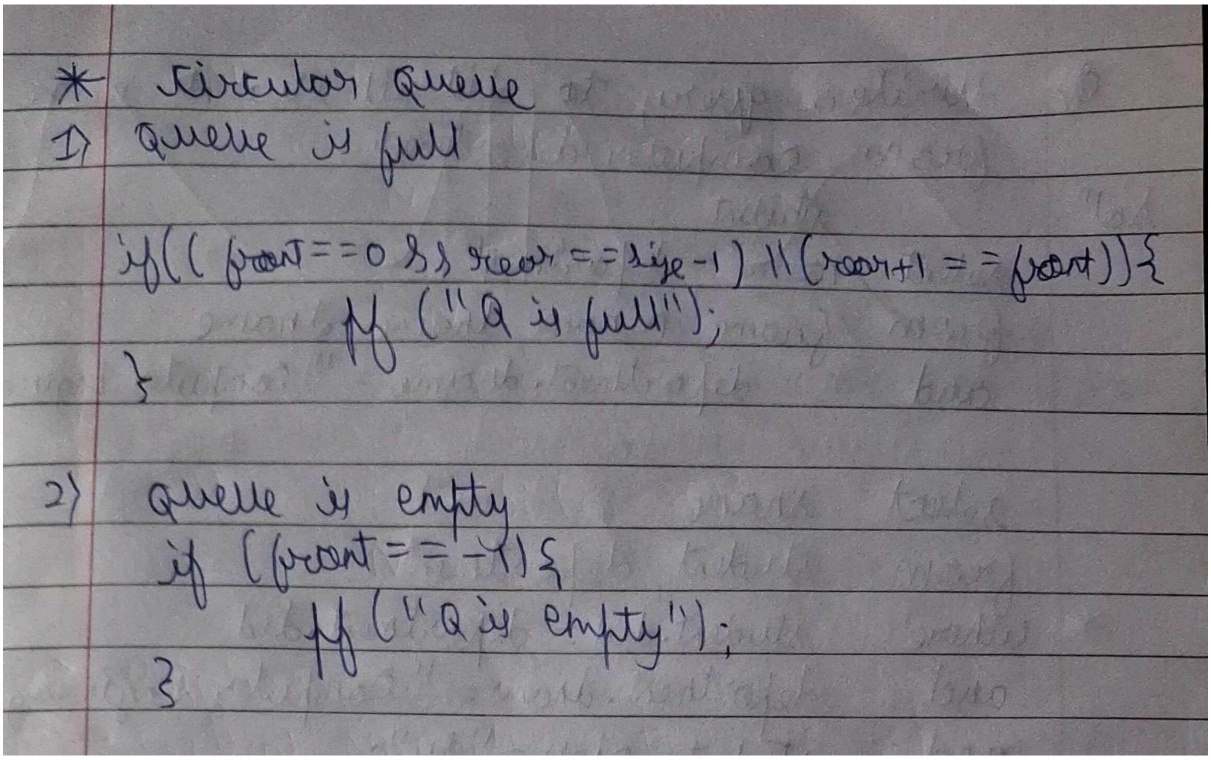
* Deletion in queue

```
int deque(){
    if((front==-1)||(rear==-1)){
        pf("queue is empty);
        return 0;
    }else{
        Temp = a[front];
        if(front==rear){
            front = rear =-1;
        }else{
            front++;
            return temp;
        }
    }
}
```

* Insertion in queue

```
int a[5];
int front = rear = -1;
void enque(int val){
    if(rear == size-1){
        pf("Q is full");
    }else{
        rear++;
        a[rear] = val;
        if(front = = -1){
            front++;
        }
    }
}
```

Circular Queue



* Circular queue

1) Queue is full

```
if((front==0 && rear==size-1)||(rear+1==front)){
        pf("Q is full");
}
```

2) queue is empty

```
if ((front==-1){
        pf("Q is empty");
}
```

* Insertion in circular queue

```
void enque (int val) {
        if ( (rear+1) % size == front) {
              pf ("Queue is full");
        } else {
              if (front == -1) {
                    front = rear = 0;
                    a[rear] = val;
              } else {
                    rear = (rear+1) % size;
                    a[rear] = val;
              }
        }
}
```

* Deletion in circular queue

```
int deque () {
     int temp;
     if (front == -1) {
          printf ("Queue is empty"); return 0;
     } else {  temp = a[front];
          if (front == rear) {
               front = rear = -1;
          } else {
               front = (front+1) % size
          } return temp; } }
```

* Display in circular queue

```
int display(){
    if (front == -1){
        printf ("Queue is empty");
    } else {
      if ((front == size-1) {
        for(i= front; i <= size-1; i++){
            printf ("%d", a[i]);
        }
        for(i=0; i<= rear; i++) {
            printf (a[i]);
        }
      } else {
        for(i= front; i <= rear; i++){
            pf (a[i]);
        }
      }
    }
}
```