1.Balance an expression

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int n;
int top = -1;

void push(int *, char);
void pop(int *);
void isbalance(int *, char *);

int main() {
    char str[30];
    printf("\nEnter the expression::");
    scanf(" %[^\n]", str);
    n = strlen(str);
    int *stack = (int *)malloc(n * sizeof(int));
    isbalance(stack, str);
    free(stack);
    return 0;
}

void push(int *stack, char a) {
    if (top == n - 1) {
        printf("\nStack Overflow");
    } else {
        top++;
        stack[top] = a;
    }
}

void pop(int *stack) {
    if (top == -1) {
        printf("\nStack Underflow");
    } else {
        top--;
    }
}

void isbalance(int *stack, char *str) {
    int i;
    for (i = 0; str[i] != '\0'; i++) {
        if (str[i] == '(') {
```

```
            push(stack, '(');
        } else if (str[i] == ')') {
            pop(stack);
        }
    }
    if (top == -1) {
        printf("\nBalanced");
    } else {
        printf("\nNot Balanced");
    }
}
```

2.Infix to postfix using stack

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int n;
int top = -1;

void push(char *, char);
char pop(char *);
int precedence(char);
void infixtopostfix(char *, char *,int );

int main() {
    char infix[30];
    char postfix[30];
    printf("\nEnter Infix Expression: ");
    scanf(" %[^\n]", infix);
    n = strlen(infix);

    infixtopostfix(infix, postfix,n);
    printf("\nPostfix Expression: %s\n", postfix);

    return 0;
}

void push(char *stack, char a) {
    if (top == n - 1) {
        printf("\nStack Overflow");
```

```
    } else {
        top++;
        stack[top] = a;
    }
}
char pop(char *stack) {
    if (top == -1) {
        printf("\nStack Underflow");
        return '\0';
    } else {
        char temp = stack[top];
        top--;
        return temp;
    }
}
int precedence(char op) {
    if (op == '^')
        return 3;
    else if (op == '*' || op == '/')
        return 2;
    else if (op == '+' || op == '-')
        return 1;
    else
        return 0;
}
void infixtopostfix(char *infix, char *postfix,int n) {
    char *stack = (char *)malloc(n * sizeof(char));
    int j = 0;

    for (int i = 0; i < n; i++) {
        char ch = infix[i];

        if ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z') || (ch
>= '0' && ch <= '9')) {
            postfix[j++] = ch;
        }

        else if (ch == '(') {
            push(stack, ch);

        }

        else if (ch == ')') {
```

```
            while (top != -1 && stack[top] != '(') {
                postfix[j++] = pop(stack);
            }
            pop(stack);

        }

        else {
            while (top != -1 && precedence(stack[top]) >=
precedence(ch)) {
                postfix[j++] = pop(stack);
            }
            push(stack, ch);
        }
    }


    while (top != -1) {
        postfix[j++] = pop(stack);
    }

    postfix[j] = '\0';
    free(stack);
}
```

3.Reverse a string

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int n;
int top = -1;

void push(char *, char);
char pop(char *);
char rev(char* ,int);

int main() {
    char str[30];
```

```
    printf("\nEnter the String::");
    scanf(" %[^\n]", str);
    n = strlen(str);
    char *stack = (char *)malloc(n * sizeof(char));
    for(int i=0;i<n;i++){
        push(stack,str[i]);
    }
    rev(stack,n);
    free(stack);
    return 0;
}

void push(char *stack, char a) {
    if (top == n - 1) {
        printf("\nStack Overflow");
    } else {
        top++;
        stack[top] = a;
    }
}

char pop(char *stack) {

    if (top == -1) {
        printf("\nStack Underflow");
        return '\0';
    } else {
        return stack[top--];
    }

}
char rev(char *stack,int n){

    printf("\nReversed String::");
    for(int i=0;i<n;i++){
        printf("%c",pop(stack));
    }


}
```

4.Queue

```c
#include<stdio.h>
#include<stdlib.h>
typedef struct Queue{
    int size;
    int front;
    int rear;
    int *Q;
}queue;
void enqueue(queue*,int);
void dequeue(queue*);
void display(queue*);
int main(){
    queue q;
    int op,x;
    printf("\nEnter size of queue::");
    scanf("%d",&q.size);
    q.Q=(int *)malloc(q.size*sizeof(int));
    q.front=q.rear=-1;
    while(1){
        printf("\n");
        printf("\n1.Enqueue");
        printf("\n2.Dequeue");
        printf("\n3.Display Queue");
        printf("\n4.Exit");
        printf("\nChoose Option::");
        scanf("%d",&op);
        switch(op){
            case 1:
                printf("\nEnter element to add:");
                scanf("%d",&x);
                enqueue(&q,x);
                break;
            case 2:
                dequeue(&q);
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("\nExiting...");
                free(q.Q);
                return 0;
```

```
                break;
            default:
                printf("\nWRONG INPUT");
                break;
        }
    }

    return 0;

}
void enqueue(queue*q,int x){
    if(q->rear==q->size-1){
        printf("\nQueue is FULL");
    }
    else{
        q->rear++;
        q->Q[q->rear]=x;
    }

}
void dequeue(queue*q){
    int x=-1;
    if(q->front==q->rear){
        printf("\nQueue is empty");
    }
    else{

        q->front++;
    }
}
void display(queue* q){
    if (q->front == q->rear){
        printf("\nQueue is empty");
    }
    else{
        printf("\nQueue elements are: ");
        for (int i = q->front + 1; i <= q->rear; i++){
            printf("%d ", q->Q[i]);
        }
        printf("\n");
    }
}
```

5.Simulate a Call Center Queue

Create a program to simulate a call center where incoming calls are handled on a first-come, first-served basis. Use a queue to manage call handling and provide options to add, remove, and view calls.

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

typedef struct Call {
    int id;
    char callerName[50];
} Call;

typedef struct Queue {
    int size;
    int front;
    int rear;
    Call *calls;
} Queue;

void enqueue(Queue *q, Call newCall);
void dequeue(Queue *q);
void display(Queue *q);

int main() {
    Queue q;
    int op, id = 1;
    char name[50];

    printf("Enter the maximum number of calls the queue can handle: ");
    scanf("%d", &q.size);

    q.calls = (Call *)malloc(q.size * sizeof(Call));
    q.front = q.rear = -1;

    while (1) {
        printf("\nCall Center Queue Options:");
        printf("\n1. Add a Call");
        printf("\n2. Handle a Call");
        printf("\n3. View All Calls");
        printf("\n4. Exit");
        printf("\nChoose an option: ");
        scanf("%d", &op);
```

```
        switch (op) {
            case 1:
                if (q.rear == q.size - 1) {
                    printf("\nQueue is full! Cannot add more
calls.\n");
                } else {
                    printf("\nEnter the caller's name: ");
                    scanf("%s", name);
                    Call newCall = {id++, ""};
                    strcpy(newCall.callerName, name);
                    enqueue(&q, newCall);
                }
                break;
            case 2:
                dequeue(&q);
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("\nExiting... Cleaning up memory.\n");
                free(q.calls);
                return 0;
            default:
                printf("\nInvalid option. Please try again.\n");
        }
    }

    return 0;
}

void enqueue(Queue *q, Call newCall) {
    if (q->rear == q->size - 1) {
        printf("\nQueue is full! Cannot add more calls.\n");
        return;
    }
    q->rear++;
    q->calls[q->rear] = newCall;
    if (q->front == -1) {
        q->front = 0;
    }
```

```
    printf("\nCall from %s (ID: %d) added to the queue.\n",
newCall.callerName, newCall.id);
}

void dequeue(Queue *q) {
    if (q->front == -1) {
        printf("\nQueue is empty! No calls to handle.\n");
        return;
    }
    printf("\nHandling call from %s (ID: %d).\n",
q->calls[q->front].callerName, q->calls[q->front].id);
    q->front++;
    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }
}

void display(Queue *q) {
    if (q->front == -1) {
        printf("\nQueue is empty! No calls to display.\n");
        return;
    }
    printf("\nCurrent Calls in the Queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("ID: %d, Caller: %s\n", q->calls[i].id,
q->calls[i].callerName);
    }
}
```

6.Print Job Scheduler

Implement a print job scheduler where print requests are queued. Allow users to add new print jobs, cancel a specific job, and print jobs in the order they were added.

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

typedef struct PrintJob {
    int id;
    char fileName[100];
} PrintJob;
```

```c
typedef struct Queue {
    int size;
    int front;
    int rear;
    PrintJob *jobs;
} Queue;

void enqueue(Queue *q, PrintJob newJob);
void dequeue(Queue *q);
void display(Queue *q);
void cancelJob(Queue *q, int jobId);

int main() {
    Queue q;
    int op, id = 1;
    char fileName[100];

    printf("Enter the maximum number of print jobs the queue can
handle: ");
    scanf("%d", &q.size);

    q.jobs = (PrintJob *)malloc(q.size * sizeof(PrintJob));
    q.front = q.rear = -1;

    while (1) {
        printf("\nPrint Job Scheduler Options:");
        printf("\n1. Add a Print Job");
        printf("\n2. Process Next Print Job");
        printf("\n3. View All Print Jobs");
        printf("\n4. Cancel a Print Job");
        printf("\n5. Exit");
        printf("\nChoose an option: ");
        scanf("%d", &op);

        switch (op) {
            case 1:
                if (q.rear == q.size - 1) {
                    printf("\nQueue is full! Cannot add more print
jobs.\n");
                } else {
                    printf("\nEnter the file name for the print job:
");
```

```
                    scanf("%s", fileName);
                    PrintJob newJob = {id++, ""};
                    strcpy(newJob.fileName, fileName);
                    enqueue(&q, newJob);
                }
                break;
            case 2:
                dequeue(&q);
                break;
            case 3:
                display(&q);
                break;
            case 4:
                if (q.front == -1) {
                    printf("\nQueue is empty! No jobs to cancel.\n");
                } else {
                    int cancelId;
                    printf("\nEnter the ID of the job to cancel: ");
                    scanf("%d", &cancelId);
                    cancelJob(&q, cancelId);
                }
                break;
            case 5:
                printf("\nExiting... Cleaning up memory.\n");
                free(q.jobs);
                return 0;
            default:
                printf("\nInvalid option. Please try again.\n");
        }
    }

    return 0;
}

void enqueue(Queue *q, PrintJob newJob) {
    if (q->rear == q->size - 1) {
        printf("\nQueue is full! Cannot add more print jobs.\n");
        return;
    }
    q->rear++;
    q->jobs[q->rear] = newJob;
    if (q->front == -1) {
        q->front = 0;
```

```
    }
    printf("\nPrint job '%s' (ID: %d) added to the queue.\n",
newJob.fileName, newJob.id);
}

void dequeue(Queue *q) {
    if (q->front == -1) {
        printf("\nQueue is empty! No jobs to process.\n");
        return;
    }
    printf("\nProcessing print job '%s' (ID: %d).\n",
q->jobs[q->front].fileName, q->jobs[q->front].id);
    q->front++;
    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }
}

void display(Queue *q) {
    if (q->front == -1) {
        printf("\nQueue is empty! No jobs to display.\n");
        return;
    }
    printf("\nCurrent Print Jobs in the Queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("ID: %d, File: %s\n", q->jobs[i].id,
q->jobs[i].fileName);
    }
}

void cancelJob(Queue *q, int jobId) {
    if (q->front == -1) {
        printf("\nQueue is empty! No jobs to cancel.\n");
        return;
    }

    int found = 0;
    for (int i = q->front; i <= q->rear; i++) {
        if (q->jobs[i].id == jobId) {
            found = 1;
            for (int j = i; j < q->rear; j++) {
                q->jobs[j] = q->jobs[j + 1];
            }
```

```
            q->rear--;
            if (q->rear < q->front) {
                q->front = q->rear = -1;
            }
            printf("\nPrint job with ID %d has been canceled.\n",
jobId);
            break;
        }
    }

    if (!found) {
        printf("\nNo print job with ID %d found in the queue.\n",
jobId);
    }
}
```

7.Design a Ticketing System

Simulate a ticketing system where people join a queue to buy tickets. Implement functionality for people to join the queue, buy tickets, and display the queue's current state.

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

typedef struct Ticket {
    int id;
    char customerName[100];
} Ticket;

typedef struct Queue {
    int size;
    int front;
    int rear;
    Ticket *tickets;
} Queue;

void enqueue(Queue *q, Ticket newTicket);
void dequeue(Queue *q);
void display(Queue *q);

int main() {
```

```
    Queue q;
    int op, id = 1;
    char name[100];

    printf("Enter the maximum number of customers the queue can handle:
");
    scanf("%d", &q.size);

    q.tickets = (Ticket *)malloc(q.size * sizeof(Ticket));
    q.front = q.rear = -1;

    while (1) {
        printf("\nTicketing System Options:");
        printf("\n1. Join the Queue");
        printf("\n2. Buy Tickets");
        printf("\n3. View Queue");
        printf("\n4. Exit");
        printf("\nChoose an option: ");
        scanf("%d", &op);

        switch (op) {
            case 1:
                if (q.rear == q.size - 1) {
                    printf("\nQueue is full! Cannot add more
customers.\n");
                } else {
                    printf("\nEnter your name: ");
                    scanf("%s", name);
                    Ticket newTicket = {id++, ""};
                    strcpy(newTicket.customerName, name);
                    enqueue(&q, newTicket);
                }
                break;
            case 2:
                dequeue(&q);
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("\nExiting... Cleaning up memory.\n");
                free(q.tickets);
                return 0;
```

```
            default:
                printf("\nInvalid option. Please try again.\n");
        }
    }

    return 0;
}

void enqueue(Queue *q, Ticket newTicket) {
    if (q->rear == q->size - 1) {
        printf("\nQueue is full! Cannot add more customers.\n");
        return;
    }
    q->rear++;
    q->tickets[q->rear] = newTicket;
    if (q->front == -1) {
        q->front = 0;
    }
    printf("\nCustomer '%s' (ID: %d) joined the queue.\n",
newTicket.customerName, newTicket.id);
}

void dequeue(Queue *q) {
    if (q->front == -1) {
        printf("\nQueue is empty! No customers to serve.\n");
        return;
    }
    printf("\nServing customer '%s' (ID: %d).\n",
q->tickets[q->front].customerName, q->tickets[q->front].id);
    q->front++;
    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }
}

void display(Queue *q) {
    if (q->front == -1) {
        printf("\nQueue is empty! No customers in the queue.\n");
        return;
    }
    printf("\nCurrent Customers in the Queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
```

```
        printf("ID: %d, Name: %s\n", q->tickets[i].id,
q->tickets[i].customerName);
    }
}
```