1- Check if expression is balanced

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

struct Stack
{
  int size;
  int top;
  char *S;
};

void create(struct Stack *, char *expr);
void push(struct Stack *,char);
void pop(struct Stack*);
int isbalanced(struct Stack*,char *);

int main()
{
  char expr[20]={"(((a+b)*((-d))))"};
  struct Stack *st = (struct Stack *)malloc(sizeof(struct Stack));
  create(st,expr);

  if(isbalanced(st,expr))
  {
    printf("The expr is balanced");
  }
  else
  {
    printf("Not balanced");
  }

  printf("\n");


  return 0;
}

void create(struct Stack *st,char *expr)
{
```

```
    st->size=strlen(expr);
    st->top=-1;
    st->S=(char *)malloc(st->size * sizeof(char));

}


void push(struct Stack *st, char x)
{
    if(st->top == st->size -1)
    {
        printf("Stack overflow");
    }
    else
    {
        st->top++;
        st->S[st->top]=x;
    }
}


void display(struct Stack *st)
{
    for(int i=st->top;i>=0;i--)
    {
        printf("%c ",st->S[i]);

    }
}


void pop(struct Stack *st)
{

    if(st->top==-1)
    {
        printf("Stack is empty");
    }
    else
    {
```

```
        st->top--;

    }

}
```

```
int isbalanced(struct Stack *st,char *expr)
{
    for(int i=0;expr[i]!='\0';i++)
    {
        if(expr[i]=='(')
        {
            push(st,expr[i]);
        }
        else if(expr[i]==')')
        {
            if(st->top==-1)
            {
                return 0;
            }
            pop(st);
        }
    }

    if(st->top ==-1)
    {
        return 1;
    }
    return 0;
}
```

2- Infix to postfix

```
#include<stdio.h>

#include<stdlib.h>
```

```
#include<string.h>

struct Stack
{
   int size;
   int top;
   char *S;
};

void create(struct Stack *, char *expr);
void push(struct Stack *,char);
int priority(char op);
void infixtopostfix(struct Stack *st,char *expr,char * post);
char pop(struct Stack*);
int isbalanced(struct Stack*,char *);

int main()
{
   char expr[20]={"(A+B)*(C+D)"};
   struct Stack *st = (struct Stack *)malloc(sizeof(struct Stack));
   char post[20];
   create(st,expr);

   if(isbalanced(st,expr))
   {
      printf("The expr is balanced");
   }
   else
```

```
    {
        printf("Not balanced");
    }

    printf("\n");
    infixtopostfix(st,expr,post);
    printf("Postfix expression: %s\n", post);

    return 0;
}

void create(struct Stack *st,char *expr)
{

    st->size=strlen(expr);
    st->top=-1;
    st->S=(char *)malloc(st->size * sizeof(char));

}


void push(struct Stack *st, char x)
{
    if(st->top == st->size -1)
    {
        printf("Stack overflow");
    }
    else
```

```
  {
    st->top++;
    st->S[st->top]=x;
  }
}




char pop(struct Stack *st)
{
  char x;

  if(st->top==-1)
  {
    printf("Stack is empty");

  }
  else
  {
    x=st->S[st->top];
    st->top--;
    return x;

  }

}
```

```
int isbalanced(struct Stack *st,char *temp)
{
   for(int i=0;temp[i]!='\0';i++)
   {
      if(temp[i]=='(')
      {
         push(st,temp[i]);
      }
      else if(temp[i]==')')
      {
         if(st->top==-1)
         {
            return 0;
         }
         char a=pop(st);
      }
   }

   if(st->top ==-1)
   {
      return 1;
   }
   return 0;
}



void infixtopostfix(struct Stack *st,char *expr,char * post)
```

```
{
   int j=0;
   for(int i=0;expr[i]!='\0';i++)
   {
      if(expr[i]=='(')
      {
         push(st,expr[i]);
      }

      else if (expr[i] == ')')
      {
         while(st->top != -1 && st->S[st->top] != '(' )
         {
            post[j++]=pop(st);

         }
         if (st->top != -1)
         { // Pop the opening parenthesis
            pop(st);
         }
      }

      else if(expr[i] == '+' || expr[i] == '-' || expr[i] == '*' || expr[i] == '/' || expr[i] == '^')
      {
         while (st->top != -1 && priority(expr[i]) <= priority(st->S[st->top]))
         {
            post[j++] = pop(st);
```

```
        }
        push(st, expr[i]);
      }
      else
      {
        post[j++]=expr[i];

      }
    }
    while (st->top != -1)
    {
      post[j++] = pop(st);
    }
    post[j]='\0';

}



int priority(char op)
{
    if (op == '+' || op == '-')
    {
      return 1;
    }
    if (op == '*' || op == '/')
    {
      return 2;
    }
```

```
    if (op == '^')
    {
       return 3;
    }
    return 0;
}
```

3- Reverse a string using stack

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


struct Stack
{
   int size;
   int top;
   char *S;
};

void create(struct Stack * st, char * expr);
void push(struct Stack* st,char  x);
char pop(struct Stack * st);


int main()
{
```

```
    int l;
    char expr[20];
    char new[20];

    struct Stack * st=(struct Stack*)malloc(sizeof(struct Stack));
    printf("Enter the expression: ");
    scanf("%[^\n]",expr);
    l=strlen(expr);
    create(st,expr);

    for(int i=0;expr[i]!='\0';i++)
    {
       push(st,expr[i]);
    }

    for(int j=0;j<l;j++)
    {
       new[j]=pop(st);
    }
    printf("\n");
    printf("the reversed string is: %s",new);
    return 0;

}


void create(struct Stack * st, char * expr)
{
```

```
    st->size=strlen(expr);
    st->top=-1;
    st->S=(char *)malloc(st->size * sizeof(char));
}

void push(struct Stack* st,char  x)
{
    if(st->top == st->size -1)
    {
        printf("full");
    }
    else
    {
        st->top++;
        st->S[st->top]=x;
    }

}

char pop(struct Stack * st)
{
    char x;
    if(st->top ==-1)
    {
        printf("empty");
    }
    else
    {
```

```
        x=st->S[st->top];
        st->top--;
        return x;
      }
    }
```

4- Queue implementation

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

struct Queue
{
  int size;
  int front;
  int rear;
  int *Q;
};

void create(struct Queue *);
void enqueue(struct Queue *, int);
int dequeue(struct Queue *);
void display(struct Queue *);

int main()
{
  struct Queue *q = (struct Queue *)malloc(sizeof(struct Queue));
```

```
    create(q);

    enqueue(q, 10);
    enqueue(q, 20);
    enqueue(q, 30);
    enqueue(q, 40);
    enqueue(q, 50);

    printf("Queue elements: ");
    display(q);
    printf("\n");

    int x = dequeue(q);
    if (x == -1)
    {
        printf("Queue is empty\n");
    }
    else
    {
        printf("Element dequeued: %d\n", x);
    }

    printf("Queue after dequeue: ");
    display(q);

    return 0;
}
```

```
void create(struct Queue *q)
{
   printf("Enter the size of the queue: ");
   scanf("%d", &q->size);
   q->front = q->rear = -1;
   q->Q = (int *)malloc(q->size * sizeof(int));
}

void enqueue(struct Queue *q, int x)
{
   if (q->rear == q->size - 1)
   {
      printf("Queue overflow\n");
   }
   else
   {
      if (q->front == -1)
      {
         q->front = 0;
      }
      q->rear++;
      q->Q[q->rear] = x;
   }
}

int dequeue(struct Queue *q)
{
   int x = -1;
```

```
    if (q->front == -1 || q->front > q->rear)
    {
        printf("Queue is empty\n");
        return x;
    }
    else
    {
        x = q->Q[q->front];
        q->front++;
        if (q->front > q->rear)
        {
            q->front = q->rear = -1;
        }
    }
    return x;
}

void display(struct Queue *q)
{
    if (q->front == -1 ||q->front > q->rear)
    {
        printf("Queue is empty");
        return;
    }

    for (int i = q->front; i <= q->rear; i++)
    {
        printf("%d ", q->Q[i]);
```

```
    }
  }
```

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.

```
#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 createQueue(Queue *q);
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];

   createQueue(&q);

   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("EnterCallID:");
            scanf("%d",&id);
            printf("Enter the caller's name: ");
            scanf("%s", name);
            Call newCall;
            strcpy(newCall.callerName, name);
            newCall.id=id;
            enqueue(&q, newCall);
        }
        break;

    case 2:
        dequeue(&q);
        break;

    case 3:
        display(&q);
        break;

    case 4:
        printf("\nExiting... \n");
        free(q.calls);
        return 0;

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

```
    return 0;
}

void createQueue(Queue *q)
{
    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;
}

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;
    }
```

```
}

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- 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 jobName[50];
} PrintJob;

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

```
void createQueue(Queue *q);
void enqueue(Queue *q, PrintJob newJob);
void dequeue(Queue *q);
void display(Queue *q);

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

   createQueue(&q);

   while (1)
   {
      printf("\nPrint Job Scheduler Options:");
      printf("\n1. Add a Print Job");
      printf("\n2. Cancel a Print Job");
      printf("\n3. View All Print Jobs");
      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 jobs.\n");
        }
        else
        {
            printf("Enter the job name: ");
            scanf("%s", jobName);
            printf("Enter Job ID:");
            scanf("%d",&id);
            PrintJob newJob;
            newJob.id = id;
            strcpy(newJob.jobName, jobName);
            enqueue(&q, newJob);
        }
        break;

    case 2:
        dequeue(&q);
        break;

    case 3:
        display(&q);
        break;

    case 4:
        printf("\nExiting...\n");
        free(q.jobs);
        return 0;
```

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

   return 0;
}

void createQueue(Queue *q)
{
   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;
}

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

   if (q->front == -1)
   {
```

```
        q->front = 0;
    }
}

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

    printf("\nCanceling print job: %s (ID: %d).\n", q->jobs[q->front].jobName, 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, Job Name: %s\n", q->jobs[i].id, q->jobs[i].jobName);
    }
}
```

7- 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 Person
{
    int id;
    char name[50];
} Person;

typedef struct Queue
{
    int size;
    int front;
    int rear;
    Person *people;
```

```
} Queue;

void createQueue(Queue *q);
void joinQueue(Queue *q, Person newPerson);
void buyTicket(Queue *q);
void displayQueue(Queue *q);

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

   createQueue(&q);

   while (1)
   {
      printf("\nTicketing System Options:");
      printf("\n1. Join the Queue");
      printf("\n2. Buy Ticket");
      printf("\n3. Display 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! No more people can join.\n");
    }
    else
    {
        printf("Enter your name: ");
        scanf("%s", name);
        Person newPerson;
        newPerson.id = id++;
        strcpy(newPerson.name, name);
        joinQueue(&q, newPerson);
    }
    break;

case 2:
    buyTicket(&q);
    break;

case 3:
    displayQueue(&q);
    break;

case 4:
    printf("\nExiting...\n");
    free(q.people);
    return 0;
```

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

   return 0;
}

void createQueue(Queue *q)
{
   printf("Enter the maximum number of people the queue can handle: ");
   scanf("%d", &q->size);
   q->people = (Person *)malloc(q->size * sizeof(Person));
   q->front = q->rear = -1;
}

void joinQueue(Queue *q, Person newPerson)
{
   if (q->rear == q->size - 1)
   {
      printf("\nQueue is full! No more people can join.\n");
      return;
   }
   q->rear++;
   q->people[q->rear] = newPerson;

   if (q->front == -1)
   {
```

```
        q->front = 0;
    }

    printf("\n%s has joined the queue with ID: %d.\n", newPerson.name, newPerson.id);
}

void buyTicket(Queue *q)
{
    if (q->front == -1)
    {
        printf("\nThe queue is empty! No one to buy tickets.\n");
        return;
    }

    printf("\n%s (ID: %d) has bought a ticket.\n", q->people[q->front].name, q->people[q->front].id);
    q->front++;

    if (q->front > q->rear)
    {
        q->front = q->rear = -1;
    }
}

void displayQueue(Queue *q)
{
    if (q->front == -1)
    {
```

```
        printf("\nThe queue is empty! No one is waiting.\n");
        return;
    }

    printf("\nCurrent Queue State:\n");
    for (int i = q->front; i <= q->rear; i++)
    {
        printf("ID: %d, Name: %s\n", q->people[i].id, q->people[i].name);
    }
}
```