Parenthesis matching using stack

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include<string.h>


struct Stack {

   int size;

   int top;

   char *S;

};


void create(struct Stack *, int);

void push(struct Stack *, char);

char pop(struct Stack *);

int isEmpty(struct Stack);

int isFull(struct Stack);

int isBalanced(char *, struct Stack);


int main() {

   struct Stack st;

   char expression[50];

   printf("Enter the expression: ");

   scanf("%s", expression);

   create(&st, strlen(expression));


   if (isBalanced(expression, st)) {

      printf("The expression is balanced.\n");

   } else {

      printf("The expression is not balanced.\n");

   }
```

```
    free(st.S);

    return 0;
}

void create(struct Stack *st, int size) {
    st->size = size;
    st->top = -1;
    st->S = (char *)malloc(st->size * sizeof(char));
    if (!st->S) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
}
void push(struct Stack *st, char x) {
    if (st->top == st->size - 1) {
        printf("Stack overflow\n");
    } else {
        st->top++;
        st->S[st->top] = x;
    }
}
char pop(struct Stack *st) {
    char x = -1;
    if (st->top == -1) {
        printf("Stack underflow\n");
    } else {
        x = st->S[st->top];
        st->top--;
    }
    return x;
```

```
}
int isEmpty(struct Stack st) {
    return st.top == -1;
}
int isFull(struct Stack st) {
    return st.top == st.size - 1;
}
int isBalanced(char *expression, struct Stack st) {
    for (int i = 0; expression[i] != '\0'; i++) {
        if (expression[i] == '(') {
            push(&st, expression[i]);
        } else if (expression[i] == ')') {
            if (isEmpty(st)) {
                return false;
            }
            pop(&st);
        }
    }
    return isEmpty(st);
}
```

Infix to postfix using stack

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

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

```
};

void create(struct Stack *, int);
void push(struct Stack *, char);
char pop(struct Stack *);
char stackTop(struct Stack st);
int isEmpty(struct Stack);
int isFull(struct Stack);
int precedence(char);
int isOperand(char);
void infixToPostfix(struct Stack,char *,char *);
int main() {
   struct Stack st;
   char infix[50],postfix[50];
   printf("Enter the size of stack: ");
   int size;
   scanf("%d",&size);
   printf("Enter the infix expression: ");
   scanf("%s",&infix);
   create(&st,size);
   infixToPostfix(st,infix,postfix);
   printf("Postfix expression: %s",postfix);

   return 0;
}

void create(struct Stack *st, int size) {
   st->size = size;
   st->top = -1;
   st->S = (char *)malloc(st->size * sizeof(char));
   if (!st->S) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
}
void push(struct Stack *st, char x) {
    if (st->top == st->size - 1) {
        printf("Stack overflow\n");
    } else {
        st->top++;
        st->S[st->top] = x;
    }
}
char pop(struct Stack *st) {
    char x = -1;
    if (st->top == -1) {
        printf("Stack underflow\n");
    } else {
        x = st->S[st->top];
        st->top--;
    }
    return x;
}
int isEmpty(struct Stack st) {
    return st.top == -1;
}
int isFull(struct Stack st) {
    return st.top == st.size - 1;
}
char stackTop(struct Stack st){
    if(isEmpty(st)){
        return -1;
    }else{
        return st.S[st.top];
    }
}
int precedence(char c){
    if(c=='+'||c=='-'){
        return 1;
    }else if(c=='*'||c=='/'){
        return 2;
    }else if(c=='^'){
        return 3;
    }
    return 0;
}
int isOperand(char c){
    return isalpha(c) || isdigit(c);
}
void infixToPostfix(struct Stack st,char *infix,char *postfix){
    int i=0,k=0;
    char symbol;
    while ((symbol = infix[i++]) != '\0'){
        if(isOperand(symbol)){
            postfix[k++]=symbol;
        }else if(symbol=='('){
            push(&st,symbol);
        }else if(symbol==')'){
            while(!isEmpty(st)&&stackTop(st)!='('){
                postfix[k++]=pop(&st);
            }
            pop(&st);
        }else{
            while(!isEmpty(st)&& precedence(stackTop(st))>=precedence(symbol)){
                postfix[k++]=pop(&st);
            }
            push(&st,symbol);
        }
    }
    while(!isEmpty(st)){
        postfix[k++]=pop(&st);
    }
    postfix[k]='\0';
}
```

Reverse a string

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

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

// Function to initialize the stack
void createStack(struct Stack *stack, int size) {
    stack->size = size;
    stack->top = -1;
    stack->arr = (char *)malloc(stack->size * sizeof(char));
    if (!stack->arr) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
```

```
}

// Function to push an element onto the stack
void push(struct Stack *stack, char c) {
    if (stack->top == stack->size - 1) {
        printf("Stack overflow\n");
    } else {
        stack->arr[++stack->top] = c;
    }
}

// Function to pop an element from the stack
char pop(struct Stack *stack) {
    if (stack->top == -1) {
        printf("Stack underflow\n");
        return -1;
    } else {
        return stack->arr[stack->top--];
    }
}

// Function to reverse a string using the stack
void reverseString(char *str) {
    int len = strlen(str);
    struct Stack stack;
    createStack(&stack, len);

    // Push all characters of the string onto the stack
    for (int i = 0; i < len; i++) {
        push(&stack, str[i]);
    }
    // Pop the characters from the stack to reverse the string
    for (int i = 0; i < len; i++) {
        str[i] = pop(&stack);
    }

    // Free the allocated memory for the stack
    free(stack.arr);
}

int main() {
    char str[100];

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);  // Read input string, including spaces
    str[strcspn(str, "\n")] = '\0';  // Remove newline character at the end

    reverseString(str);

    printf("Reversed string: %s\n", str);

    return 0;
}
```

QUEUE

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

struct Queue{
    int size;
    int front;
    int rear;
    int *Q;
};
void enqueue(struct Queue *,int);
int dequeue(struct Queue *);
int main(){
    struct Queue q;
    printf("Enter the size: ");
    scanf("%d",&q.size);
    q.Q = (int *)malloc(q.size*sizeof(int));
    q.front=q.rear=-1;
    enqueue(&q,12);
    enqueue(&q,8);
    enqueue(&q,9);
    enqueue(&q,10);
    enqueue(&q,11);
    int x = dequeue(&q);
    printf("Dequeue element = %d",x);
}
void enqueue(struct Queue *q,int x){
    if(q->rear==q->size-1){
        printf("Queue is full");
    }else{
        q->rear++;
        q->Q[q->rear]=x;
    }
}
int dequeue(struct Queue *q){
    int x=-1;
   if(q->front==q->rear){
       printf("Queue is empty");
   }else{
      q->front++;
      x=q->Q[q->front];
   }
   return x;
}
```

1.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>
struct call{
    int callID;
    char callerName[50];
};

struct Queue{
    int size;
    int front;
    int rear;
    struct call *calls;
};
void enqueue(struct Queue *,int,char *);
void dequeue(struct Queue *);
void view(struct Queue *);
int main(){
    struct Queue q;
    printf("Enter the size of queue: ");
    scanf("%d",&q.size);
    q.calls = (struct call *)malloc(q.size*sizeof(struct call));
    q.front=q.rear=-1;
```

```
int choice,callID;
char callerName[50];

while(1){
    printf("1.Add a call\n");
    printf("2.Remove a call\n");
    printf("3.View the current queue\n");
    printf("4.Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1:
            printf("Enter the call ID: ");
            scanf("%d",&callID);
            printf("Enter the caller name: ");
            scanf("%s",&callerName);
            enqueue(&q,callID,callerName);
            break;
        case 2:
            dequeue(&q);
            break;
        case 3:
            view(&q);
            break;
        case 4:
            free(q.calls);
            exit(0);
        default:
            printf("Invalid option");
            break;
    }
```

```
   }
}
void enqueue(struct Queue *q,int callID,char *callerName){
   if(q->rear==q->size-1){
      printf("Queue is full");
   }else{
      q->rear++;
      q->calls[q->rear].callID = callID;
      strcpy(q->calls[q->rear].callerName,callerName);
      printf("Successfully added");
   }
}
void dequeue(struct Queue *q){
   int x=-1;
  if(q->front==q->rear){
     printf("Queue is empty");
  }else{
     q->front++;
     printf("Call ID %d is handled.",q->calls[q->front].callID);
  }
}
void view(struct Queue *q){
   int i=q->front;
   while(i!=q->rear){
      i++;
      printf("Call ID: %d, Caller Name: %s\n",q->calls[i].callID,q->calls[i].callerName);
   }
}
```

2.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>
struct Printjob{
    int jobID;
    char jobName[50];
};

struct Queue{
    int size;
    int front;
    int rear;
    struct Printjob *jobs;
};
void enqueue(struct Queue *,int,char *);
void dequeue(struct Queue *);
void view(struct Queue *);
void cancelJob(struct Queue *,int);
int main(){
    struct Queue q;
    printf("Enter the size of queue: ");
    scanf("%d",&q.size);
    q.jobs = (struct Printjob *)malloc(q.size*sizeof(struct Printjob));
    q.front=q.rear=-1;
    int choice,jobID;
    char jobName[50];

    while(1){
        printf("1.Add a new print job\n");
        printf("2.Cancel a specific print job\n");
        printf("3.View all print jobs\n");
        printf("4.Process the next job\n");
        printf("5.Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1:
                printf("Enter the job ID: ");
                scanf("%d",&jobID);
                printf("Enter the job name: ");
                scanf("%s",&jobName);
                enqueue(&q,jobID,jobName);
                break;
            case 2:
                printf("Enter the job ID to cancel: ");
                scanf("%d",&jobID);
                cancelJob(&q,jobID);
                break;
            case 3:
                view(&q);
                break;
            case 4:
                dequeue(&q);
                break;
            case 5:
                free(q.jobs);
                exit(0);
            default:
                printf("Invalid option");
                break;
        }
    }
}
```

```
void enqueue(struct Queue *q,int jobID,char *jobName){
    if(q->rear==q->size-1){
        printf("Queue is full. Cannot add new job");
    }else{
        q->rear++;
        q->jobs[q->rear].jobID = jobID;
        strcpy(q->jobs[q->rear].jobName,jobName);
        printf("Successfully added");
    }
}
void dequeue(struct Queue *q){
   if(q->front==q->rear){
       printf("Queue is empty. No jobs to process");
   }else{
       q->front++;
       printf("Job ID %d is done.",q->jobs[q->front].jobID);
   }
}
void view(struct Queue *q){
    int i=q->front;
    while(i!=q->rear){
        i++;
        printf("Job ID: %d, Job Name: %s\n",q->jobs[i].jobID,q->jobs[i].jobName);
    }
}
void cancelJob(struct Queue *q, int jobID) {
    if(q->front==q->rear){
       printf("Queue is empty. No jobs to process");
   }
    int i = q->front;
    int found = 0;
    // Search for the job to cancel
    while (i != q->rear) {
        if (q->jobs[i].jobID == jobID) {
            found = 1;
            break;
        }
        i = (i + 1) % q->size;
    }

    // Check if the job is found at the last position
    if (!found && q->jobs[i].jobID == jobID) {
        found = 1;
    }

    if (found) {
        printf("Job ID %d ('%s') has been canceled.\n", jobID, q->jobs[i].jobName);
        // Shift jobs to fill the canceled job's position
        while (i != q->rear) {
            int next = (i + 1) % q->size;
            q->jobs[i] = q->jobs[next];
            i = next;
        }
        q->rear = (q->rear - 1 + q->size) % q->size;
        if (q->front == q->rear) {
            q->front = q->rear = -1;  // Queue is empty
        }
    } else {
        printf("Job ID %d not found in the queue.\n", jobID);
    }
}
```

3.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>

// Define the structure for a person in the queue
struct Person {
    int personID;
    char personName[50];
};

// Define the structure for the queue
struct Queue {
    int size;
    int front;
    int rear;
    struct Person *persons;
};

// Function declarations
void enqueue(struct Queue *, int, char *);
void dequeue(struct Queue *);
int isEmpty(struct Queue *);
int isFull(struct Queue *);
void view(struct Queue *);

int main() {
    struct Queue q;

    // Initialize the queue
```

```
printf("Enter the size of the queue: ");
scanf("%d", &q.size);
q.persons = (struct Person *)malloc(q.size * sizeof(struct Person));
q.front = q.rear = -1;

int choice, personID;
char personName[50];

// Main menu loop
while (1) {
   printf("\nTicketing System Menu:\n");
   printf("1. Join the queue\n");
   printf("2. Buy a ticket\n");
   printf("3. View the current queue\n");
   printf("4. Exit\n");
   printf("Enter your choice: ");
   scanf("%d", &choice);

   switch (choice) {
      case 1:
         printf("Enter the person ID: ");
         scanf("%d", &personID);
         printf("Enter the person name: ");
         scanf(" %[^\n]", personName);  // To handle spaces in person names
         enqueue(&q, personID, personName);
         break;
      case 2:
         dequeue(&q);
         break;
      case 3:
         view(&q);
```

```
                break;
            case 4:
                free(q.persons);
                exit(0);
            default:
                printf("Invalid option\n");
                break;
        }
    }
}

// Function to add a person to the queue (join the queue)
void enqueue(struct Queue *q, int personID, char *personName) {
    if (isFull(q)) {
        printf("Queue is full. Cannot join the queue.\n");
    } else {
        q->rear = (q->rear + 1) % q->size;
        q->persons[q->rear].personID = personID;
        strcpy(q->persons[q->rear].personName, personName);
        if (q->front == -1) {
            q->front = 0; // First person joins the queue
        }
        printf("Person ID %d ('%s') has joined the queue.\n", personID, personName);
    }
}

// Function to remove a person from the front of the queue (buy a ticket)
void dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No one to buy a ticket.\n");
    } else {
        printf("Person ID %d ('%s') has bought a ticket.\n", q->persons[q->front].personID, q->persons[q->front].personName);

        q->front = (q->front + 1) % q->size;

        if (q->front == (q->rear + 1) % q->size) { // Queue is empty after the operation

            q->front = q->rear = -1;

        }

    }

}


// Function to check if the queue is empty

int isEmpty(struct Queue *q) {

    return q->front == -1;

}


// Function to check if the queue is full

int isFull(struct Queue *q) {

    return (q->rear + 1) % q->size == q->front;

}


// Function to view all people in the queue

void view(struct Queue *q) {

    if (isEmpty(q)) {

        printf("The queue is empty.\n");

    } else {

        int i = q->front;

        printf("Current people in the queue:\n");

        while (i != q->rear) {

            printf("Person ID: %d, Person Name: %s\n", q->persons[i].personID, q->persons[i].personName);

            i = (i + 1) % q->size;

        }

        // Print the last person in the queue
        printf("Person ID: %d, Person Name: %s\n", q->persons[i].personID, q->persons[i].personName);
    }
}
```