Stack implementation using linked list

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

#define SUCCESS 0
#define FAILURE -1
typedef struct stack{
   int data;
   struct stack *link;
}Stack_l;

void push(Stack_l **head,int data);
Stack_l *createNode(Stack_l *newNode,int data);
void pop(Stack_l **head);
void peep(Stack_l *head);
int peek(Stack_l *head);

int main(){
   int op;
   Stack_l *head = NULL;
   while(1){
      printf("\nChoose the option:\n");
      printf("1.Push\n");
      printf("2.Pop\n");
      printf("3.Peep\n");
      printf("4.Peek\n");
      printf("5.Exit\n");
      scanf("%d",&op);

      switch(op){
         case 1:
            int data;
            printf("Enter the data to push into stack\n");
            scanf("%d",&data);
            push(&head,data);
            break;
         case 2:
            pop(&head);
            break;
         case 3:
            peep(head);
            break;
         case 4:
            int PEEK = peek(head);
            if(PEEK == FAILURE){
               printf("Stack is empty\n");
            }else{
               printf("Peeked data is %d\n",PEEK);
            }
            break;
         case 5:
            printf("Exiting...\n");
            return 0;
            break;
      }
```

```
    }
    return 0;
}

void push(Stack_l **head,int data){
    Stack_l *newNode = createNode(newNode,data);
    if(newNode == NULL){
        printf("Memory allocation failed\n");
        printf("memory is full\n");
        return;
    }
    if(*head == NULL){
        *head = newNode;
        return;
    }else{
        Stack_l *temp = *head;
        newNode->link = temp;
        *head = newNode;
        return;
    }
}

Stack_l *createNode(Stack_l *newNode,int data){
    newNode = (Stack_l *)malloc(sizeof(Stack_l));
    if(newNode == NULL){
        return NULL;
    }
    newNode->data = data;
    newNode->link = NULL;
    return newNode;
}

void pop(Stack_l **head){
    if(*head == NULL){
        printf("Stack is empty\n");
        return;
    }
    Stack_l *temp = *head;
    *head = temp->link;
    free(temp);
    return;
}

void peep(Stack_l *head){
    if(head == NULL){
        printf("Stack is empty\n");
        return;
    }
    else{
        Stack_l *temp = head;

        while(temp){
            printf("%d\n",temp->data);
            temp = temp->link;
        }
        return;
    }
```

```
}

int peek(Stack_l *head){
    if(head == NULL){
        return FAILURE;

    }
    else{
        return head->data;
    }
}
```

write a program in c using stack implement a function to check if an expression is balanced or not

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

#define SUCCESS 0
#define FAILURE -1

typedef struct stack {
    int capacity;
    int top;
    char *stack;
} Stack_t;

int create_stack(Stack_t *, int);
int Push(Stack_t *, char);
int Pop(Stack_t *, char *);
int isBalanced(const char *);

int main() {
    char expression[100];

    printf("Enter an expression: ");
    scanf("%s", expression);

    if (isBalanced(expression)) {
        printf("The expression is balanced.\n");
    } else {
        printf("The expression is not balanced.\n");
    }

    return SUCCESS;
}

int create_stack(Stack_t *stk, int size) {
    stk->stack = (char *)malloc(size * sizeof(char));
    if (!stk->stack) {
        return FAILURE;
    }
    stk->top = -1;
    stk->capacity = size;
    return SUCCESS;
}

int Push(Stack_t *stk, char element) {
    if (stk->top == stk->capacity - 1) {
        return FAILURE;
    }
    stk->stack[++stk->top] = element;
    return SUCCESS;
}

int Pop(Stack_t *stk, char *element) {
    if (stk->top == -1) {
        return FAILURE;
    }
    *element = stk->stack[stk->top--];
    return SUCCESS;
}

int isBalanced(const char *expression) {
    Stack_t stk;
    create_stack(&stk, 100);

    for (int i = 0; expression[i] != '\0'; i++) {
        char ch = expression[i];

        if (ch == '(' || ch == '{' || ch == '[') {
            if (Push(&stk, ch) == FAILURE) {
                printf("ERROR: Stack overflow.\n");
                free(stk.stack);
                return false;
            }
        } else if (ch == ')' || ch == '}' || ch == ']') {
            char topElement;
            if (Pop(&stk, &topElement) == FAILURE ||
                !((topElement == '(' && ch == ')') ||
                  (topElement == '{' && ch == '}') ||
                  (topElement == '[' && ch == ']'))) {
                free(stk.stack);
                return false;
            }
        }
    }

    if(stk.top == -1){
        free(stk.stack);
        return stk.top;
    }
    else
    {
        free(stk.stack);
        return SUCCESS;
    }
}
```

write a programme in c using stack implementation to convert infix expression to postfix expression

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

#define SUCCESS 0
#define FAILURE -1

typedef struct stack {
    int capacity;
    int top;
    char *stack;
} Stack_t;

void converToPostfix(char *infix, char *postfix);
Stack_t *createStack(Stack_t *stk, int size);
void push(Stack_t *stk, char element);
char pop(Stack_t *stk);
char peek(Stack_t *stk);
int precedence(char op);

static int j = 0;

int main() {
    char infix[100], postfix[100];
    printf("Enter the expression: ");
    scanf("%s", infix);

    converToPostfix(infix, postfix);

    printf("The converted expression is: %s\n", postfix);
    return SUCCESS;
}

void push(Stack_t *stk, char element) {
    if (stk->top == stk->capacity - 1) {
        printf("ERROR: Stack overflow\n");
        exit(FAILURE);
    }
    stk->stack[++stk->top] = element;
}

char pop(Stack_t *stk) {
    if (stk->top == -1) {
        printf("ERROR: Stack underflow\n");
        exit(FAILURE);
    }
    return stk->stack[stk->top--];
}
```

```
char peek(Stack_t *stk) {
    if (stk->top == -1) {
        return '\0';
    }
    return stk->stack[stk->top];
}

void converToPostfix(char *infix, char *postfix) {
    Stack_t stk;
    createStack(&stk, 100);

    for (int i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];
        if (isalnum(ch)) {
            postfix[j++] = ch;
        }
        else if(ch == '('){
            push(&stk, ch);
        }else if(ch == ')'){
            while (stk.top != -1 && peek(&stk) != '(') {
                postfix[j++] = pop(&stk);
            }
            if (stk.top != -1 && peek(&stk) == '(') {
                pop(&stk);
            } else {
                printf("ERROR: Mismatched parentheses\n");
                return;
            }
        }
            else {
            while (stk.top != -1 && precedence(ch) <= precedence(peek(&stk))) {
                postfix[j++] = pop(&stk);
            }
            push(&stk, ch);
        }
    }

    while (stk.top != -1) {
        postfix[j++] = pop(&stk);
    }
    postfix[j] = '\0';
}

Stack_t *createStack(Stack_t *stk, int size) {
    stk->stack = (char *)malloc(size * sizeof(char));
    if (stk->stack == NULL) {
        printf("ERROR: Memory allocation failed\n");
        exit(FAILURE);
    }
    stk->top = -1;
    stk->capacity = size;
    return stk;
}

int precedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return 0;
    }
}
```

Reverse a String Using Stack

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

#define SUCCESS 0
#define FAILURE -1

typedef struct stack {
    int capacity;
    int top;
    int *stack;
} Stack_t;

int create_stack(Stack_t *stk, int size);
int Push(Stack_t *stk, int element);
int Pop(Stack_t *stk);
void reverseString(char *str);

int main()
{

    Stack_t stk;

    char str[100];
    printf("Enter a string to reverse: ");
    scanf(" %[^\n]", str);
    reverseString(str);
    printf("Reversed String: %s\n", str);

}

int create_stack(Stack_t *stk, int size)
{
    stk->stack = (int *)malloc(size * sizeof(int));
    if (!stk->stack)
    {
        return FAILURE;
    }
    stk->top = -1;
    stk->capacity = size;
    return SUCCESS;
}

int Push(Stack_t *stk, int element)
{
    if (stk->top == stk->capacity - 1)
    {
        return FAILURE; // Stack is full
    }
    stk->stack[++stk->top] = element;
    return SUCCESS;
}

int Pop(Stack_t *stk)
{
    if (stk->top == -1)
    {
        return FAILURE;
    }
    stk->top--;
    return SUCCESS;
}

void reverseString(char *str)
{
    int length = strlen(str);
    Stack_t stk;
    create_stack(&stk, length);


    for (int i = 0; i < length; i++)
    {
        Push(&stk, str[i]);
    }

    for (int i = 0; i < length; i++)
    {
        str[i] = stk.stack[stk.top];
        Pop(&stk);
    }

    free(stk.stack);
}
```

Queue Implementation:

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

#define SUCCESS 0
#define FAILURE -1

typedef struct{
    int capacity;
    int front;
    int rare;
    int *queue;
}Queue;
int Enqueue(Queue *q,int data);
int Dequeue(Queue *q);
void PrintQueue(Queue *q);
int main(){
    Queue q;
    printf("Enter the size of queue:");
    scanf("%d",&q.capacity);

    q.queue = (int *)malloc(q.capacity *sizeof(int));

    q.front = q.rare = -1;

    int op;
    do{
       printf("\n1.Enqueue\n");
       printf("2.Dequeue\n");
       printf("3.Print Queue\n");
       printf("4.Exit\n");
       printf("Enter your choice :");
       scanf("%d", &op);

       switch(op)
       {
          case 1:
          {
           int data;
           printf("Enter the data to be enquqed: ");
           scanf("%d", &data);
```

```
            if(Enqueue(&q, data) == FAILURE)
            printf("Queue is full\n");
           }
           break;
           case 2:
           {
            int removed = Dequeue(&q);
            if (removed != FAILURE)
            printf("%d is removed from queue.\n", removed);
            else
            printf("Queue is empty.\n");
           }
           break;
           case 3:
           {
           PrintQueue(&q);
           break;
           }
           case 4:
           {
           printf("Exiting...\n");
           break;
           }
           default:
           printf("Invalid option!!\n");
        }
    }while(op != 4);



    return 0;
}

int Enqueue(Queue *q,int data){
    if(q->rare == q->capacity-1){

        return FAILURE;
    }
    else{
        q->queue[++q->rare] = data;
        return SUCCESS;
    }
}

int Dequeue(Queue *q){
    if(q->front == q->rare){
        return FAILURE;
    }
    else{
        q->front++;
        return q->queue[q->front];

    }
}

void PrintQueue(Queue *q) {
    if (q->front == q->rare) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements: ");
        for (int i = q->front + 1; i <= q->rare; i++) {
            printf("%d ", q->queue[i]);
        }
        printf("\n");
    }
}
```

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>

#define FAILURE -1

typedef struct
{
    char name[50];
    char phone[11];
} Call;

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

void create_queue(Queue *q, int size);
int add_call(Queue *q, const char *name, const char *phone);
int remove_call(Queue *q);
int view_calls(Queue q);
void free_queue(Queue *q);

int main()
{
    int size, choice;
```

```c
char name[50];
char phone[11];
Queue queue;

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

create_queue(&queue, size);

do {
    printf("\nCall Center Queue Menu:\n");
    printf("1. Add Call\n");
    printf("2. Remove Call\n");
    printf("3. View Calls\n");
    printf("4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice)
    {
        case 1:
            printf("Enter phone number (10 digits): ");
            scanf("%10s", phone);
            printf("Enter caller name: ");
            scanf(" %[^\n]", name);
            if (add_call(&queue, name, phone) == FAILURE)
                printf("Queue is full. Cannot add call.\n");
            else
                printf("Call added successfully.\n");
            break;

        case 2:
            if (remove_call(&queue) == FAILURE)
                printf("Queue is empty. No call to remove.\n");
            else
                printf("Call handled and removed from queue.\n");
            break;

        case 3:
            if (view_calls(queue) == FAILURE)
                printf("Queue is empty.\n");
            break;

        case 4:
            printf("Exiting Call Center Queue Simulation.\n");
            break;

        default:
            printf("Invalid choice. Please try again.\n");
    }
    } while (choice != 4);

    free_queue(&queue);
    return 0;
}

void create_queue(Queue *q, int size)
{
    q->size = size;
    q->front = -1;
    q->rear = -1;
    q->calls = (Call *)malloc(size * sizeof(Call));
}

int add_call(Queue *q, const char *name, const char *phone)
{
    if (q->rear == q->size - 1)
        return FAILURE;

    if (q->front == -1)
        q->front = 0;

    q->rear++;
    strcpy(q->calls[q->rear].name, name);
    strcpy(q->calls[q->rear].phone, phone);

    return 0;
}

int remove_call(Queue *q)
{
    if (q->front == -1 || q->front > q->rear)
        return FAILURE;
    printf("Handled Call: Phone = %s, Name = %s\n", q->calls[q->front].phone, q-
>calls[q->front].name);
    q->front++;

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

    return 0;
}

int view_calls(Queue q)
{
    if (q.front == -1 || q.front > q.rear)
        return FAILURE;
    printf("\nCurrent Calls in Queue:\n");
    for (int i = q.front; i <= q.rear; i++)
    {
        printf("Phone: %s, Name: %s\n", q.calls[i].phone, q.calls[i].name);
    }

    return 0;
}

void free_queue(Queue *q)
{
    free(q->calls);
}
```

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>

#define FAILURE -1

typedef struct {
    int job_id;
    char description[100];
} PrintJob;

typedef struct {
    int size;
    int front, rear;
    PrintJob *jobs;
} JobQueue;

void create_queue(JobQueue *q, int size);
int add_job(JobQueue *q, int job_id, const char *description);
int cancel_job(JobQueue *q);
int print_jobs(JobQueue *q);
void free_queue(JobQueue *q);

int main()
{
    int size, choice, job_id;
    char description[100];
    JobQueue queue;
```

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

create_queue(&queue, size);

do {
   printf("\nPrint Job Scheduler Menu:\n");
   printf("1. Add New Job\n");
   printf("2. Cancel Job\n");
   printf("3. Print All Jobs\n");
   printf("4. Exit\n");
   printf("Enter your choice: ");
   scanf("%d", &choice);

   switch (choice)
   {
      case 1:
         printf("Enter job ID: ");
         scanf("%d", &job_id);
         printf("Enter job description: ");
         scanf(" %[^\n]", description);
         if (add_job(&queue, job_id, description) == FAILURE)
            printf("Queue is full. Cannot add job.\n");
         else
            printf("Job added successfully.\n");
         break;

      case 2:
         if (cancel_job(&queue) == FAILURE)
            printf("Queue is empty. No job to cancel.\n");
         else
            printf("Job canceled successfully.\n");
         break;

      case 3:
         if (print_jobs(&queue) == FAILURE)
            printf("No jobs in the queue.\n");
         break;

      case 4:
         printf("Exiting Print Job Scheduler.\n");
         break;

      default:
         printf("Invalid choice. Please try again.\n");
   }
} while (choice != 4);

free_queue(&queue);
```

```
    return 0;
}

void create_queue(JobQueue *q, int size)
{
    q->size = size;
    q->front = -1;
    q->rear = -1;
    q->jobs = (PrintJob *)malloc(size * sizeof(PrintJob));
}

int add_job(JobQueue *q, int job_id, const char *description)
{
    if (q->rear == q->size - 1)
        return FAILURE;

    if (q->front == -1)
        q->front = 0;

    q->rear++;
    q->jobs[q->rear].job_id = job_id;
    strcpy(q->jobs[q->rear].description, description);

    return 0;
}

int cancel_job(JobQueue *q)
{
    if (q->front == -1 || q->front > q->rear)
        return FAILURE;

    printf("Canceled Job ID: %d, Description: %s\n", q->jobs[q->front].job_id, q->jobs[q->front].description);

    q->front++;

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

    return 0;
}

int print_jobs(JobQueue *q)
{
    if (q->front == -1 || q->front > q->rear)
        return FAILURE;
    printf("\nCurrent Print Jobs in Queue:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("Job ID: %d, Description: %s\n", q->jobs[i].job_id, q->jobs[i].description);
    }

    return 0;
}

void free_queue(JobQueue *q)
{
    free(q->jobs);
}
```

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 FAILURE -1

// Structure to hold ticket buyer information
typedef struct {
    char name[50];
    int tickets;
} Buyer;

// Queue structure
typedef struct {
    int size;
    int front, rear;
    Buyer *buyers;
} Queue;

// Function declarations
void create_queue(Queue *q, int size);
int join_queue(Queue *q, const char *name, int tickets);
int buy_ticket(Queue *q);
int display_queue(Queue q);
void free_queue(Queue *q);

int main() {
    int size, choice;
    char name[50];
    int tickets;
    Queue queue;
    printf("Enter the maximum number of people in the queue: ");
    scanf("%d", &size);

    create_queue(&queue, size);

    do {
        printf("\nTicketing System Menu:\n");
        printf("1. Join Queue\n");
        printf("2. Buy Ticket\n");
        printf("3. Display Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter name: ");
                scanf(" %[^
]", name);
                printf("Enter number of tickets: ");
                scanf("%d", &tickets);

                if (join_queue(&queue, name, tickets) == FAILURE)
                    printf("Queue is full. Cannot join.");
                else
                    printf("%s joined the queue successfully.\n", name);
                break;

            case 2:
                if (buy_ticket(&queue) == FAILURE)
                    printf("Queue is empty. No tickets to buy.\n");
                else
                    printf("Tickets bought successfully.\n");
                break;

            case 3:
                if (display_queue(queue) == FAILURE)
                    printf("Queue is empty.\n");
                break;

            case 4:
                printf("Exiting Ticketing System Simulation.\n");
                break;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 4);
    free_queue(&queue);
    return 0;
}

// Function to initialize the queue
void create_queue(Queue *q, int size) {
    q->size = size;
    q->front = -1;
    q->rear = -1;
    q->buyers = (Buyer *)malloc(size * sizeof(Buyer));
}

// Function to add a person to the queue
int join_queue(Queue *q, const char *name, int tickets) {
    if (q->rear == q->size - 1)
        return FAILURE;

    if (q->front == -1)
        q->front = 0;

    q->rear++;
    strcpy(q->buyers[q->rear].name, name);
    q->buyers[q->rear].tickets = tickets;

    return 0;
}

// Function to process ticket buying for the person at the front
int buy_ticket(Queue *q) {
    if (q->front == -1 || q->front > q->rear)
        return FAILURE;

    printf("Processed Buyer: Name = %s, Tickets = %d\n", q->buyers[q->front].name,
q->buyers[q->front].tickets);
    q->front++;

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

    return 0;
}

// Function to display the current queue
int display_queue(Queue q) {
    if (q.front == -1 || q.front > q.rear)
        return FAILURE;

    printf("\nCurrent Queue:\n");
    for (int i = q.front; i <= q.rear; i++) {
        printf("Name: %s, Tickets: %d\n", q.buyers[i].name, q.buyers[i].tickets);
    }

    return 0;
}

// Function to free allocated memory for the queue
void free_queue(Queue *q) {
    free(q->buyers);
}
```