栈和队列——数据结构

栈

栈的定义

栈(stack)是限定仅在表尾进行插入和删除操作的线性表。

我们把允许数据插入和删除的一端称为栈顶(top)，另一端称为栈底(bottom)，不含任何数据元素的栈称为空栈。栈又称为后进先出(Last In First Out)的线性表。

栈的顺序存储结构

栈的实现一般可以使用数组或者链表实现，相对而言数组的结构实现更优一些。因为数组在尾上插入数据的代价比较小。

```
// 支持动态增长的栈
typedef int DataType;
typedef struct Stack
{
    DataType* arr;
    int top; //栈顶指针,初始top为0，指向栈顶的下一个位置
    int capacity; //容量
}Stack;
```

入栈

```
// 入栈
void StackPush(Stack* ps, DataType data)
{
    //判断栈是否已满
    if (ps->top == ps->capacity)
    {
        //将栈的容量加倍
        ps->capacity = ps->capacity * 2;
        //申请新的空间并将之前的数据复制过去
        ps->arr = (DataType*)realloc(ps->arr,sizeof(DataType)*ps->capacity);
    }
    //将数据入栈
    ps->arr[ps->top] = data;
    ps->top++;
}
```

出栈

```c
// 出栈
void StackPop(Stack* ps)
{
    //判端栈是否为空
    if (ps->top == 0)
    {
        return;
    }
    //栈顶指针减一
    ps->top--;
}
```

基于动态数组实现的顺序栈

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

// 支持动态增长的栈
typedef int DataType;
typedef struct Stack
{
    DataType* arr;
    int top; //栈顶指针,初始top为0，指向栈顶的下一个位置
    int capacity; //容量
}Stack;
// 初始化栈
void StackInit(Stack* ps)
{
    ps->top = 0;
    ps->capacity = 2;
    ps->arr = (DataType*)malloc(sizeof(DataType)*ps->capacity);

}
// 入栈
void StackPush(Stack* ps, DataType data)
{
    //判断栈是否已满
    if (ps->top == ps->capacity)
    {
        //将栈的容量加倍
        ps->capacity = ps->capacity * 2;
        //申请新的空间并将之前的数据复制过去
        ps->arr = (DataType*)realloc(ps->arr,sizeof(DataType)*ps->capacity);
    }
    //将数据入栈
    ps->arr[ps->top] = data;
    ps->top++;
}
// 出栈
void StackPop(Stack* ps)
{
    //判端栈是否为空
    if (ps->top == 0)
    {
        return;
```

```
    }
    //栈顶指针减一
    ps->top--;
}
// 获取栈顶元素
DataType StackTop(Stack* ps)
{
    //返回栈顶数据元素
    return ps->arr[ps->top-1];
}
// 获取栈中有效元素个数
int StackSize(Stack* ps)
{
    return ps->top;
}
// 检测栈是否为空，如果为空返回非零结果，如果不为空返回0
int StackEmpty(Stack* ps)
{
    if (ps->top == 0)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
// 销毁栈
void StackDestroy(Stack* ps)
{
    free(ps->arr);
    ps->arr = NULL;
    ps->top = ps->capacity = 0;
}
```

用栈模拟实现队列

思路：

分别申请两个栈，puststt用来实现入队操作，popst用来实现出队操作；

入队时，直接将数据元素放入栈pushst；

出队时，先判断栈popst是否为空，若不为空则直接执行栈的出栈操作，若为空将pushst 的数据元素执行出栈操作压入栈popst中，知道pushst为空，popst执行栈的出栈操作.

```
//队列的结构体定义
typedef struct {
    Stack pushst;
    Stack popst;
} MyQueue;

//队列的初始化
MyQueue* myQueueCreate() {
    MyQueue* q=(MyQueue*)malloc(sizeof(MyQueue));
    StackInit(&q->pushst);
    StackInit(&q->popst);
    return q;
```

```c
}
//入队操作
void myQueuePush(MyQueue* obj, int x) {
    StackPush(&obj->pushst,x);
}
//出队操作
int myQueuePop(MyQueue* obj) {
    if(StackEmpty(&obj->popst))
    {
        while(!StackEmpty(&obj->pushst))
        {
            StackPush(&obj->popst,StackTop(&obj->pushst));
            StackPop(&obj->pushst);
        }
    }
    int front=StackTop(&obj->popst);
    StackPop(&obj->popst);
    return front;
}
//返回队头元素
int myQueuePeek(MyQueue* obj) {
    if(StackEmpty(&obj->popst))
    {
        while(!StackEmpty(&obj->pushst))
        {
            StackPush(&obj->popst,StackTop(&obj->pushst));
            StackPop(&obj->pushst);
        }
    }
    return StackTop(&obj->popst);
}
//判断队列是否为空
bool myQueueEmpty(MyQueue* obj) {
    return StackEmpty(&obj->pushst)&&StackEmpty(&obj->popst);
}
//销毁队列
void myQueueFree(MyQueue* obj) {
    StackDestroy(&obj->popst);
    StackDestroy(&obj->pushst);
    free(obj);
}
```

队列

队列的定义

队列(queue)是只允许在一端进行插入操作，另一端进行删除操作的线性表。

队列是一种先进先出(First In First Out)的线性表，允许插入的一段称为队尾，允许删除的一端称为队头。

队列的链式存储结构

队列也可以数组和链表的结构实现，使用链表的结构实现更优一些，因为如果使用数组的结构，出队列在数组头上出数据，效率会比较低 。

```c
typedef int DataType;
//链式结构：表示队列
typedef struct QListNode
{
    struct QListNode* pNext;
    DataType data;
}QueueNode;
//队列的结构
typedef struct Queue
{
    QueueNode* front;//队头指针
    QueueNode* rear;//队尾指针
}Queue;
```

入队

```c
// 队尾入队列
void QueuePush(Queue* q, DataType data)
{
    //申请结点空间
    QueueNode* newnode = (QueueNode*)malloc(sizeof(QueueNode));
    //结点赋值
    newnode->data = data;
    newnode->next;
    //判断队列是否为空
    if (q->front == NULL)
    {
        //为空，直接插入
        q->front = q->rear = newnode;
    }
    else
    {
        //不为空，将新节点插在队尾，重置队尾指针
        q->rear->next = newnode;
        q->rear = newnode;
    }
}
```

出队

```c
// 队头出队列
void QueuePop(Queue* q)
{
    //判断队列是否为空
    if (q->front == NULL)
    {
        //为空直接return
        return;
    }
    //定义tmp指针暂存队头的下一个指针
    QueueNode* tmp = q->front->next;
    //释放队头结点
    free(q->front);
    //重置新的队头
    q->front = tmp;
}
```

基于单链表实现的链队列

```c
typedef int DataType;
//链式结构：表示队列
typedef struct QListNode
{
    struct QListNode* next;
    DataType data;
}QueueNode;
//队列的结构
typedef struct Queue
{
    QueueNode* front;//队头指针
    QueueNode* rear;//队尾指针
}Queue;

// 初始化队列
void QueueInit(Queue* q)
{
    q->front = NULL;
    q->rear = NULL;
}
// 队尾入队列
void QueuePush(Queue* q, DataType data)
{
    //申请结点空间
    QueueNode* newnode = (QueueNode*)malloc(sizeof(QueueNode));
    //结点赋值
    newnode->data = data;
    newnode->next;
    //判断队列是否为空
    if (q->front == NULL)
    {
        //为空，直接插入
        q->front = q->rear = newnode;
    }
    else
    {
        //不为空，将新节点插在队尾，重置队尾指针
        q->rear->next = newnode;
        q->rear = newnode;
    }
}
// 队头出队列
void QueuePop(Queue* q)
{
    //判断队列是否为空
    if (q->front == NULL)
    {
        //为空直接return
        return;
    }
    //定义tmp指针暂存队头的下一个指针
    QueueNode* tmp = q->front->next;
    //释放队头结点
    free(q->front);
```

```
    //重置新的队头
    q->front = tmp;
}
// 获取队列头部元素
DataType QueueFront(Queue* q)
{
    return q->front->data;
}
// 获取队列队尾元素
DataType QueueBack(Queue* q)
{
    return q->rear->data;
}
// 获取队列中有效元素个数
int QueueSize(Queue* q)
{
    QueueNode* cur;
    //计数器
    int count = 0;
    //循环遍历队列
    for (cur = q->front; cur; cur = cur->next)
    {
        count++;
    }
    return count;
}
// 检测队列是否为空，如果为空返回非零结果，如果非空返回0
int QueueEmpty(Queue* q)
{
    return q->front == NULL;
}
// 销毁队列
void QueueDestroy(Queue* q)
{
    if (q->front == NULL)
    {
        return;
    }
    while (q->front)
    {
        QueuePop(q);
    }
}
```

循环队列

```
typedef struct {
    int* array;
    int maxsize;//数组的空间大小
    int front;//队头指针
    int rear;//队尾指针,若队列不空，指向队尾元素的下一个位置
} MyCircularQueue;


//构造器，设置循环队列长度为k
MyCircularQueue* myCircularQueueCreate(int k) {
```

```
    MyCircularQueue* cq = (MyCircularQueue*)malloc(sizeof(MyCircularQueue));
    cq->array = (int*)malloc(sizeof(int)*(k + 1));
    cq->maxsize = k + 1;
    cq->front = 0;
    cq->rear = 0;
    return cq;
}
//检查队列是否为空
bool myCircularQueueIsEmpty(MyCircularQueue* obj) {
    //队头指针和队尾指针相遇则队列为空
    return obj->front == obj->rear;
}
//检查队列是否为满
bool myCircularQueueIsFull(MyCircularQueue* obj) {
    //当(rear+1)%size==front时队列为满
    return (obj->rear + 1) % obj->maxsize == obj->front;
}
//向队列插入一个元素
bool myCircularQueueEnQueue(MyCircularQueue* obj, int value) {
    //队列满的判断
    if (myCircularQueueIsFull(obj))
    {
        return false;
    }
    //将元素e赋值给队尾
    obj->array[obj->rear] = value;
    //rear指针向后移一位，若到最后则转到数组头部
    obj->rear = (obj->rear + 1) % obj->maxsize;
    return true;
}
//从队列删除一个元素
bool myCircularQueueDeQueue(MyCircularQueue* obj) {
    //队列为空的判断
    if (myCircularQueueIsEmpty(obj))
    {
        return false;
    }
    //front指针后移，若到最后则转到数组头部
    obj->front = (obj->front + 1) % obj->maxsize;
    return true;
}
//获取队首元素
int myCircularQueueFront(MyCircularQueue* obj) {
    if (myCircularQueueIsEmpty(obj))
    {
        return -1;
    }
    return obj->array[obj->front];
}
//获取队尾元素
int myCircularQueueRear(MyCircularQueue* obj) {
    if (myCircularQueueIsEmpty(obj))
    {
        return -1;
    }
    int prerear = obj->rear - 1;
    if (obj->rear == 0)
```

```
    {
        prerear = obj->maxsize - 1;
    }
    return obj->array[prerear];
}
//销毁队列
void myCircularQueueFree(MyCircularQueue* obj) {
    free(obj->array);
    free(obj);
}
```