

# 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。

```
1 // 支持动态增长的栈
2 typedef int DataType;
3 typedef struct Stack
4 {
5     DataType* arr;
6     int top; //栈顶指针,初始top为0, 指向栈顶的下一个位置
7     int capacity; //容量
8 }Stack;
```

### 入栈

```
1 // 入栈
2 void StackPush(Stack* ps, DataType data)
3 {
4     //判断栈是否已满
5     if (ps->top == ps->capacity)
6     {
7         //将栈的容量加倍
8         ps->capacity = ps->capacity * 2;
9         //申请新的空间并将之前的数据复制过去
10        ps->arr = (DataType*)realloc(ps->arr, sizeof(DataType)*ps->capacity);
11    }
12    //将数据入栈
13    ps->arr[ps->top] = data;
14    ps->top++;
15 }
```

### 出栈

```

1 // 出栈
2 void StackPop(Stack* ps)
3 {
4     //判断栈是否为空
5     if (ps->top == 0)
6     {
7         return;
8     }
9     //栈顶指针减一
10    ps->top--;
11 }

```

## 基于动态数组实现的顺序栈

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // 支持动态增长的栈
5 typedef int DataType;
6 typedef struct Stack
7 {
8     DataType* arr;
9     int top; //栈顶指针,初始top为0, 指向栈顶的下一个位置
10    int capacity; //容量
11 }Stack;
12 // 初始化栈
13 void StackInit(Stack* ps)
14 {
15     ps->top = 0;
16     ps->capacity = 2;
17     ps->arr = (DataType*)malloc(sizeof(DataType)*ps->capacity);
18 }
19
20 // 入栈
21 void StackPush(Stack* ps, DataType data)
22 {
23     //判断栈是否已满
24     if (ps->top == ps->capacity)
25     {
26         //将栈的容量加倍
27         ps->capacity = ps->capacity * 2;
28         //申请新的空间并将之前的数据复制过去
29         ps->arr = (DataType*)realloc(ps->arr, sizeof(DataType)*ps->
30                                     >capacity);
31     }
32     //将数据入栈
33     ps->arr[ps->top] = data;
34     ps->top++;
35 }
36
37 // 出栈
38 void StackPop(Stack* ps)
39 {
40     //判断栈是否为空
41     if (ps->top == 0)
42     {
43         return;
44     }
45 }

```

```

42     }
43     //栈顶指针减一
44     ps->top--;
45 }
46 // 获取栈顶元素
47 DataType StackTop(Stack* ps)
48 {
49     //返回栈顶数据元素
50     return ps->arr[ps->top-1];
51 }
52 // 获取栈中有效元素个数
53 int StackSize(Stack* ps)
54 {
55     return ps->top;
56 }
57 // 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
58 int StackEmpty(Stack* ps)
59 {
60     if (ps->top == 0)
61     {
62         return 1;
63     }
64     else
65     {
66         return 0;
67     }
68 }
69 // 销毁栈
70 void StackDestroy(Stack* ps)
71 {
72     free(ps->arr);
73     ps->arr = NULL;
74     ps->top = ps->capacity = 0;
75 }

```

## 用栈模拟实现队列

思路：

分别申请两个栈，pushst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。

```

1 //队列的结构体定义
2 typedef struct {
3     Stack pushst;
4     Stack popst;
5 } MyQueue;
6
7 //队列的初始化
8 MyQueue* myQueueCreate() {
9     MyQueue* q=(MyQueue*)malloc(sizeof(MyQueue));
10    StackInit(&q->pushst);
11    StackInit(&q->popst);
12    return q;

```

```

13 }
14 //入队操作
15 void myQueuePush(MyQueue* obj, int x) {
16     StackPush(&obj->pushst,x);
17 }
18 //出队操作
19 int myQueuePop(MyQueue* obj) {
20     if(StackEmpty(&obj->popst))
21     {
22         while(!StackEmpty(&obj->pushst))
23         {
24             StackPush(&obj->popst,StackTop(&obj->pushst));
25             StackPop(&obj->pushst);
26         }
27     }
28     int front=StackTop(&obj->popst);
29     StackPop(&obj->popst);
30     return front;
31 }
32 //返回队头元素
33 int myQueuePeek(MyQueue* obj) {
34     if(StackEmpty(&obj->popst))
35     {
36         while(!StackEmpty(&obj->pushst))
37         {
38             StackPush(&obj->popst,StackTop(&obj->pushst));
39             StackPop(&obj->pushst);
40         }
41     }
42     return StackTop(&obj->popst);
43 }
44 //判断队列是否为空
45 bool myQueueEmpty(MyQueue* obj) {
46     return StackEmpty(&obj->pushst)&&StackEmpty(&obj->popst);
47 }
48 //销毁队列
49 void myQueueFree(MyQueue* obj) {
50     StackDestroy(&obj->popst);
51     StackDestroy(&obj->pushst);
52     free(obj);
53 }

```

## 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。

```

1  typedef int DataType;
2  //链式结构：表示队列
3  typedef struct QListNode
4  {
5      struct QListNode* pNext;
6      DataType data;
7  }QueueNode;
8  //队列的结构
9  typedef struct Queue
10 {
11     QueueNode* front;//队头指针
12     QueueNode* rear;//队尾指针
13 }Queue;

```

## 入队

```

1  // 队尾入队列
2  void QueuePush(Queue* q, DataType data)
3  {
4      //申请结点空间
5      QueueNode* newnode = (QueueNode*)malloc(sizeof(QueueNode));
6      //结点赋值
7      newnode->data = data;
8      newnode->next;
9      //判断队列是否为空
10     if (q->front == NULL)
11     {
12         //为空，直接插入
13         q->front = q->rear = newnode;
14     }
15     else
16     {
17         //不为空，将新节点插在队尾，重置队尾指针
18         q->rear->next = newnode;
19         q->rear = newnode;
20     }
21 }

```

## 出队

```

1  // 队头出队列
2  void QueuePop(Queue* q)
3  {
4      //判断队列是否为空
5      if (q->front == NULL)
6      {
7          //为空直接return
8          return;
9      }
10     //定义tmp指针暂存队头的下一个指针
11     QueueNode* tmp = q->front->next;
12     //释放队头结点
13     free(q->front);
14     //重置新的队头
15     q->front = tmp;
16 }

```

## 基于单链表实现的链队列

```
1  typedef int DataType;
2  //链式结构：表示队列
3  typedef struct QListNode
4  {
5      struct QListNode* next;
6      DataType data;
7  }QueueNode;
8  //队列的结构
9  typedef struct Queue
10 {
11     QueueNode* front;//队头指针
12     QueueNode* rear;//队尾指针
13 }Queue;
14
15 // 初始化队列
16 void QueueInit(Queue* q)
17 {
18     q->front = NULL;
19     q->rear = NULL;
20 }
21 // 队尾入队列
22 void QueuePush(Queue* q, DataType data)
23 {
24     //申请结点空间
25     QueueNode* newnode = (QueueNode*)malloc(sizeof(QueueNode));
26     //结点赋值
27     newnode->data = data;
28     newnode->next;
29     //判断队列是否为空
30     if (q->front == NULL)
31     {
32         //为空，直接插入
33         q->front = q->rear = newnode;
34     }
35     else
36     {
37         //不为空，将新节点插在队尾，重置队尾指针
38         q->rear->next = newnode;
39         q->rear = newnode;
40     }
41 }
42 // 队头出队列
43 void QueuePop(Queue* q)
44 {
45     //判断队列是否为空
46     if (q->front == NULL)
47     {
48         //为空直接return
49         return;
50     }
51     //定义tmp指针暂存队头的下一个指针
52     QueueNode* tmp = q->front->next;
53     //释放队头结点
54     free(q->front);
```

```

55     //重置新的队头
56     q->front = tmp;
57 }
58 // 获取队列头部元素
59 DataType QueueFront(Queue* q)
60 {
61     return q->front->data;
62 }
63 // 获取队列队尾元素
64 DataType QueueBack(Queue* q)
65 {
66     return q->rear->data;
67 }
68 // 获取队列中有效元素个数
69 int QueueSize(Queue* q)
70 {
71     QueueNode* cur;
72     //计数器
73     int count = 0;
74     //循环遍历队列
75     for (cur = q->front; cur; cur = cur->next)
76     {
77         count++;
78     }
79     return count;
80 }
81 // 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
82 int QueueEmpty(Queue* q)
83 {
84     return q->front == NULL;
85 }
86 // 销毁队列
87 void QueueDestroy(Queue* q)
88 {
89     if (q->front == NULL)
90     {
91         return;
92     }
93     while (q->front)
94     {
95         QueuePop(q);
96     }
97 }

```

## 循环队列

```

1  typedef struct {
2      int* array;
3      int maxsize; //数组的空间大小
4      int front; //队头指针
5      int rear; //队尾指针, 若队列不空, 指向队尾元素的下一个位置
6  } MyCircularQueue;
7
8
9  //构造器, 设置循环队列长度为k
10 MyCircularQueue* myCircularQueueCreate(int k) {

```

```

11     MyCircularQueue* cq =
(MyCircularQueue*)malloc(sizeof(MyCircularQueue));
12     cq->array = (int*)malloc(sizeof(int)*(k + 1));
13     cq->maxsize = k + 1;
14     cq->front = 0;
15     cq->rear = 0;
16     return cq;
17 }
18 //检查队列是否为空
19 bool myCircularQueueIsEmpty(MyCircularQueue* obj) {
20     //队头指针和队尾指针相遇则队列为空
21     return obj->front == obj->rear;
22 }
23 //检查队列是否为满
24 bool myCircularQueueIsFull(MyCircularQueue* obj) {
25     //当(rear+1)%size==front时队列为满
26     return (obj->rear + 1) % obj->maxsize == obj->front;
27 }
28 //向队列插入一个元素
29 bool myCircularQueueEnQueue(MyCircularQueue* obj, int value) {
30     //队列满的判断
31     if (myCircularQueueIsFull(obj))
32     {
33         return false;
34     }
35     //将元素e赋值给队尾
36     obj->array[obj->rear] = value;
37     //rear指针向后移一位，若到最后则转到数组头部
38     obj->rear = (obj->rear + 1) % obj->maxsize;
39     return true;
40 }
41 //从队列删除一个元素
42 bool myCircularQueueDeQueue(MyCircularQueue* obj) {
43     //队列为空的判断
44     if (myCircularQueueIsEmpty(obj))
45     {
46         return false;
47     }
48     //front指针后移，若到最后则转到数组头部
49     obj->front = (obj->front + 1) % obj->maxsize;
50     return true;
51 }
52 //获取队首元素
53 int myCircularQueueFront(MyCircularQueue* obj) {
54     if (myCircularQueueIsEmpty(obj))
55     {
56         return -1;
57     }
58     return obj->array[obj->front];
59 }
60 //获取队尾元素
61 int myCircularQueueRear(MyCircularQueue* obj) {
62     if (myCircularQueueIsEmpty(obj))
63     {
64         return -1;
65     }
66     int prerear = obj->rear - 1;
67     if (obj->rear == 0)

```



```
68     {
69         prerear = obj->maxsize - 1;
70     }
71     return obj->array[prerear];
72 }
73 //销毁队列
74 void myCircularQueueFree(MyCircularQueue* obj) {
75     free(obj->array);
76     free(obj);
77 }
```