

3-栈和队列

C生万物 ● 大道至简 ● 鲍鱼科技+v(15339278619)

1、目标



栈的顺序和链式实现

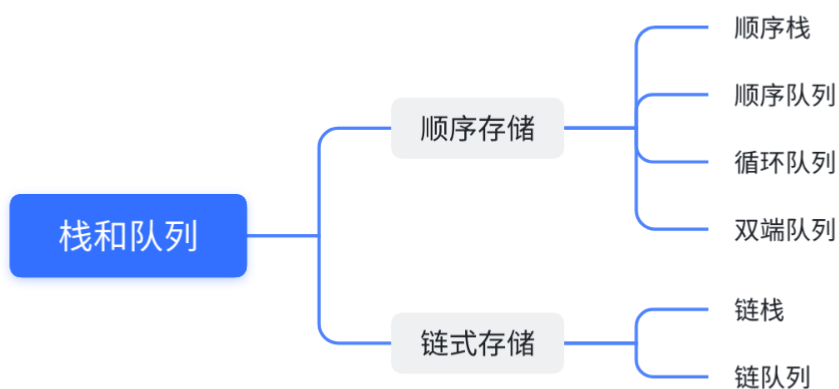
队列的顺序和链式实现

循环队列的实现以及循环条件

栈和队列的应用



栈和队列是操作受限制的线性表




2、栈




栈是先进后出结构，喝多了吐就是一个栈

2.1 顺序栈

 注意top的初始化，两种形式-1，和0，会影响push和pop的实现操作


2.2 顺序栈ADT

```
1 //顺序栈
2 typedef struct SeqStack
3 {
4     SeqStackElem_Type *base;
5     size_t          capacity;
6     int             top;
7 }SeqStack;
8
9 void SeqStackInit(SeqStack *pst);
10 void SeqStackPush(SeqStack *pst, SeqStackElem_Type v);
11 void SeqStackPop(SeqStack *pst);
12 SeqStackElem_Type SeqStackTop(SeqStack *pst);
13 void SeqStackShow(SeqStack *pst);
14 void SeqStackDestroy(SeqStack *pst);
15
16 bool SeqStackFull(SeqStack *pst);
17 bool SeqStackEmpty(SeqStack *pst);
```

 通过实现上述顺序栈接口，需要达到的目的：


- 1、深入掌握顺序栈的管理和实现
- 2、顺序栈是一个数组连续空间结构，入栈出栈需要判满判空
- 3、需要注意top的初始值，可能为0，也可能为-1，将影响出栈入栈的指针操作
- 4、要学会顺序栈操作的各种画图，其中理解顺序栈的管理是核心关键

2.3链栈

 只允许在链表的一头（表头）插入和删除

```
1 typedef struct LinkStackNode
2 {
3     LinkStackElem_Type data;
4     struct LinkStackNode *next;
```

```
5 }LinkStackNode;
6
7 typedef LinkStackNode* LinkStack;
8
9 bool LinkStackEmpty(LinkStack pst);
10 void LinkStackInit(LinkStack *pst);
11 void LinkStackPush(LinkStack *pst, LinkStackElem_Type v);
12 void LinkStackPop(LinkStack *pst);
13 LinkStackElem_Type LinkStackTop(LinkStack pst);
14 void LinkStackDestroy(LinkStack *pst);
```

 通过实现上述链栈接口，需要达到的目的：


- 1、深入链栈的管理方式和实现
- 2、为了效率，一般链栈的入栈和出栈均在头部进行，所以要注意头指针的修改问题
- 3、链栈是通过链表来实现逻辑结构的栈，所以针对链表不可以随意操作，因此栈是操作受限的线性结构

 栈结构必会题型：

- 1、括号匹配检测 <https://leetcode.cn/problems/valid-parentheses/>
- 2、表达式求解 <https://leetcode.cn/problems/evaluate-reverse-polish-notation/>
- 3、进制转换 <https://leetcode.cn/problems/convert-a-number-to-hexadecimal/>
- 4、入栈出栈的顺序组合 <https://leetcode.cn/problems/zhan-de-ya-ru-dan-chu-xu-lie-lcof/>
- 5、最小栈 <https://leetcode.cn/problems/min-stack/>
- 6、用栈实现队列 <https://leetcode.cn/problems/implement-queue-using-stacks/>

3、栈的应用

3.1 表达式的转换

 表达式的表示分为三种：

前缀表达式、中缀表达式、后缀表达式

其中，中缀表达式：是一个通用的算术或逻辑公式表示方法，操作符是以中缀形式处于操作数的中间（例：3 + 4），中缀表达式是人们常用的算术表示方法。

1、一个简单地中缀表达式：a+b

前缀表达式：+ab

后缀表达式：ab+

2、其中a和b分别是一个表达式，而“+”就是运算符

转成前缀表达式就是把运算符放到前面，转成后缀表达式就是把运算符放到后面放

总体的思路：

把每一个表达式先用括号括上，再把运算符提到括号前（后）

a+b

(1) 加括号：(a) + (b) --> ((a) + (b))

(2) 把运算符提到括号前：+ ((a) (b))

(3) 去掉括号：+ab

3、来个复杂的例子：

(a+b) * c+d- (e+g) *h

(1) 加括号：根据运算法则，先*/后+-，有括号的要优先

(a+b) *c+d- (e+g) *h

= ((a+b) *c) +d - ((e+g) *h)

= (((a+b) *c) +d) - ((e+g) *h)

= ((((a+b) *c) +d) - ((e+g) *h))

此时的((((a+b) *c) +d) 就相当于a，((e+g) *h) 相当于b

(2) 提取运算符

第一步：- ((((a+b) *c) +d) ((e+g) *h))

此时再看((((a+b) *c) +d)，((a+b) *c) 相当于a，而d相当于b

提取运算符：

第二步：+ ((((a+b) *c) d)

两步合并

- (+ ((((a+b) *c) d) ((e+g) *h))

为了看这方便，可以去掉提取运算符后的括号：

-+ ((a+b) *c) d ((e+g) *h)

第三步：

$((a+b) * c)$ 的提取结果： $* ((a+b) c)$

向上合并：

$-+ * ((a+b) c) d ((e+g) * h)$

去括号： $-+ * (a+b) c d ((e+g) * h)$

第五步：提取 $(a+b)$

结果： $-+ * + (ab) c d ((e+g) * h)$

去括号： $-+ * + abcd ((e+g) * h)$

第六步： $((e+g) * h)$

过程和上面的一样，这里略过： $* (e+g) h = *+egh$

最终的结果： $-+ * + abcd *+egh$

4，转成后缀的思路和转成前缀的思路一致，这里就直接脱式运算了

$$\begin{aligned} & (((((a+b) * c) + d) - ((e+g) * h)) \\ &= (((((a+b) * c) + d) ((e+g) * h)) - \\ &= ((((a+b) * c) + d) ((e+g) * h) - \\ &= ((a+b) * c) d + ((e+g) * h) - \\ &= ((a+b) c) * d + ((e+g) * h) - \\ &= (a+b) c * d + ((e+g) * h) - \\ &= (ab) + c * d + ((e+g) * h) - \\ &= ab + c * d + ((e+g) * h) - \\ &= ab + c * d + (e+g) h * - \\ &= ab + c * d + (eg) + h * - \\ &= ab + c * d + eg + h * - \end{aligned}$$


 练习：

中缀表达式： $(6+3*(7-4))-8/2$

前缀表达式： $- + 6 * 3 - 7 4 / 8 2$

后缀表达式： $6 3 7 4 - * + 8 2 / -$

• 中缀转前缀栈实现过程

 计算机利用栈，将中缀表达式转化为前缀表达式的过程

1、借助两个栈：操作符栈、结果栈

2、表达式扫描顺序：**从右往左**扫描

3、如果遇到数字直接输出到结果栈

4、如果遇到运算符，则比较优先级

4.1 如果当前运算符的优先级 \geq 栈顶运算符的优先级(**当栈顶是括号时，直接入栈**)，则将运算符直接入栈

4.2 如果当前运算符的优先级 $<$ 栈顶运算符的优先级，则将栈顶运算符出栈并输出到结果栈，直到当前运算符的优先级 \geq 栈顶运算符的优先级(**当栈顶是括号时，直接入栈**)，再将当前运算符入栈

5、如果遇到括号，则根据括号的方向进行处理。


5.1 如果是右括号，则直接入栈；

5.2 如果是左括号，则遇到右括号前将所有的运算符全部出栈并输出至结果栈，最后将左右括号舍去

6、重复上述的3、4、5步骤，直到表达式扫描完毕

7、扫描完成中缀表达式后，结果栈中所保留的数据则为前缀表达式

• 中缀转后缀实现过程

 计算机利用栈，将中缀表达式转化为后缀表达式的过程

1、借助一个栈和一个队列：操作符栈、结果队列

2、表达式扫描顺序：**从左往右**扫描

3、如果遇到数字直接输出到结果队列

4、如果遇到运算符，则比较优先级

4.1 如果当前运算符的优先级 $>$ 栈顶运算符的优先级(**当栈顶是括号时，直接入栈**)，则将运算符直接入栈

4.2 如果当前运算符的优先级 \leq 栈顶运算符的优先级，则将栈顶运算符出栈并输出到结果队列，直到当前运算符的优先级 $>$ 栈顶运算符的优先级(**当栈顶是括号时，直接入栈**)，再将当前运算符入栈

5、如果遇到括号，则根据括号的方向进行处理。

5.1 如果是左括号，则直接入栈

5.2 如果是右括号，则遇到左括号前将所有的运算符全部出栈并输出至结果队列，最后将左右括号舍去

6、重复上述的3、4、5步骤，直到表达式扫描完毕

7、扫描完成中缀表达式后，结果对了中所保留的数据则为后缀表达式

3.2 表达式求值

前缀表达式求值：

前缀表达式又称波兰式,前缀表达式的运算符位于操作数之前

举例说明： $(3+4) \times 5 - 6$ 对应的前缀表达式就是 $- \times + 3 4 5 6$

前缀表达式求值过程：

- 1、借助一个栈：数据栈
- 2、表达式扫描顺序：**从右往左**扫描
- 3、如果遇到操作数，将操作数压入数据栈
- 4、如果遇到运算符，弹出栈顶的两个数，先出栈的为左数，后出栈的为右数，做运算后将结果重新入栈
- 5、重复步骤3、4，直到表达式扫描完毕，则数据栈中保存的数据则为表达式的结果

中缀表达式求值：

中缀表达式，如 $(3+4) \times 5 - 6$

中缀表达式是最熟悉，最符合人的思维习惯的表达式，但计算机本身无法知道表达式的优先级问题，需要在实现过程中加以控制

中缀表达式求值过程：

- 1、需要借助两个栈结构：操作符栈，数据栈
- 2、表达式扫描顺序：**从左往右**扫描
- 3、如果遇到操作数，将操作数压入数据栈
- 4、如果遇到运算符，则比较优先级
 - 4.1 如果当前运算符的优先级 **>** 栈顶运算符的优先级(**当栈顶是括号时，直接入栈**)，则将运算符直接接入栈
 - 4.2 如果当前运算符的优先级 **<** 栈顶运算符的优先级，则将栈顶运算符出栈，并将数据栈出栈，先出的为右值，后出的为左值，将运算之后的结果重新入到数据栈
- 5、如果遇到括号，则根据括号的方向进行处理。

5.1 如果是左括号，则直接入栈

5.2 如果是右括号，则遇到左括号前将所有的运算符全部出栈，并将数据栈两个数出栈，将运算之后的结果重新入到数据栈，直到遇到左括号为止

6、重复上述的3、4、5步骤，直至整个表达式扫描完成

表 3.1 算符间的优先关系

| $\theta_1 \backslash \theta_2$ | + | - | * | / | (|) | # |
|--------------------------------|---|---|---|---|---|---|---|
| + | > | > | < | < | < | > | > |
| - | > | > | < | < | < | > | > |
| * | > | > | > | > | < | > | > |
| / | > | > | > | > | < | > | > |
| (| < | < | < | < | < | = | |
|) | > | > | > | > | > | > | > |
| # | < | < | < | < | < | | = |

后缀表达式求值：


后缀表达式又称逆波兰表达式，运算符位于操作数之后

举例说明： $(3+4) \times 5 - 6$ 对应的前缀表达式就是 $3\ 4 + 5 \times 6 -$


后缀表达式求值过程：

- 1、只需借助一个栈：数据栈
- 2、表达式扫描顺序：**从左往右**扫描
- 3、如果遇到操作数，将操作数压入数据栈
- 4、如果遇到运算符，弹出栈顶的两个数，先出栈的为右数，后出栈的为左数，做运算后并将结果重新入栈
- 5、重复步骤3、4，直到表达式扫描完毕，则数据栈中保存的数据则为表达式的结果

4、队列

 队列是先进先出结构，吃多了拉就是一个队列


4.1 顺序队列

 队列是FIFO,或者LILO结构，队列虽小，却在排队场景中有着重要的地位

普通实现容易造成空间浪费，因此顺序队列往往实现成**循环队列**，数组能够循环的关键在于取模%运算

• 顺序队列ADT

```
1 #define SeqQueueElem_Type int
2 #define SQUEUE_MAX_SIZE 8
3 typedef struct SeqQueue
4 {
5     SeqQueueElem_Type *base; //队列空间
6     size_t capacity; //队列容量
7     int front; //对头指针
8     int rear; //队尾指针
9 }SeqQueue;
10
11 bool SeqQueueFull(SeqQueue *pq);
12 bool SeqQueueEmpty(SeqQueue *pq);
13
14 void SeqQueueInit(SeqQueue *pq);
15 void SeqQueuePush(SeqQueue *pq, SeqQueueElem_Type v);
16 void SeqQueuePop(SeqQueue *pq);
17 SeqQueueElem_Type SeqQueueFront(SeqQueue *pq);
18 SeqQueueElem_Type SeqQueueBack(SeqQueue *pq);
19 void SeqQueueShow(SeqQueue *pq);
```

 通过实现上述顺序队列的接口，需要达到的目的：

- 1、深入掌握顺序队列的管理方式和实现
- 2、顺序队列有空间的限制，所以插入数据要判满，删除数据要判空
- 3、队列管理的特殊性，容易造成空间的假满状态，因此顺序队列的实现一般会实现成循环队列

4.2循环队列

 循环队列主要解决一个问题：**使队列空间能够重复使用**

循环队列面临两个问题：**一是如何循环的问题，二是如何区分空与满的状态**

• 循环队列ADT

```

1 #define CircleQueueElem_Type int
2 #define CIRCLE_QUEUE_MAX_SIZE 8
3
4 typedef struct CircleQueue
5 {
6     CircleQueueElem_Type *base;
7     size_t capacity;
8     int front;
9     int rear;
10 }CircleQueue;
11
12 void CircleQueueInit(CircleQueue *pcq);
13 void CircleQueuePush(CircleQueue *pcq, CircleQueueElem_Type v);
14 void CircleQueuePop(CircleQueue *pcq);
15 CircleQueueElem_Type CircleQueueFront(CircleQueue *pcq);
16 CircleQueueElem_Type CircleQueueBack(CircleQueue *pcq);
17 void CircleQueueShow(CircleQueue *pcq);
18 void CircleQueueDestroy(CircleQueue *pcq);
19
20 bool CircleQueueFull(CircleQueue *pcq);
21 bool CircleQueueEmpty(CircleQueue *pcq);

```



从结构定义来看，循环队列跟顺序队列的定义没有什么区别

循环队列最大的特点就是希望空间能够循环利用，所以如何实现循环就是关键

循环的关键在于：对容量取模， $\%capacity$ ，因此循环队列的编写难点在于条件的判断

4.3 链式队列



只允许在链表的一头插入，另一头删除

• 链式队列ADT

```


1 #define LinkQueueElem_Type int
2
3 typedef struct LinkQueueNode
4 {
5     LinkQueueElem_Type data;
6     struct LinkQueueNode *next;
7 }LinkQueueNode;
8
9 typedef struct LinkQueue

```

```

10 {
11     LinkQueueNode *front;
12     LinkQueueNode *rear;
13 }LinkQueue;
14
15 bool LinkQueueEmpty(LinkQueue *pq);
16 void LinkQueueInit(LinkQueue *pq);
17 void LinkQueuePush(LinkQueue *pq, LinkQueueElem_Type v);
18 void LinkQueuePop(LinkQueue *pq);
19 LinkQueueElem_Type LinkQueueFront(LinkQueue *pq);
20 LinkQueueElem_Type LinkQueueBack(LinkQueue *pq);
21 void LinkQueueShow(LinkQueue *pq);
22 void LinkQueueDestroy(LinkQueue *pq);
23
24 bool LinkQueueEmpty(LinkQueue *pq);

```

 通过实现上述链式队列的接口，需要达到的目的：

- 1、深入掌握链式队列的管理方式和实现
- 2、链式队列具有头尾指针，入队出队需要正确修改指针
- 3、链式队列是通过链表来实现逻辑结构的队列，所以针对链表不可以随意操作，因此队列是操作受限制的线性结构

 队列必会题型：

- 1、设计循环队列 <https://leetcode.cn/problems/design-circular-queue/>
- 2、队列最大值 <https://leetcode.cn/problems/dui-lie-de-zui-da-zhi-lcof/description/>
- 3、用队列实现栈 <https://leetcode.cn/problems/implement-stack-using-queues/>

- 1 1. 一个栈的初始状态为空。现将元素1、2、3、4、5、A、B、C、D、E依次入栈，然后再依次出栈，则元
- 2 栈的顺序是（ ）。
- 3 A 12345ABCDE
- 4 B EDCBA54321
- 5 C ABCDE12345
- 6 D 54321EDCBA
- 7
- 8 2. 若进栈序列为 1,2,3,4，进栈过程中可以出栈，则下列不可能的一个出栈序列是（）
- 9 A 1,4,3,2
- 10 B 2,3,4,1

11 C 3,1,4,2
12 D 3,4,2,1
13
14 3. 循环队列的存储空间为 $Q(1:100)$ ，初始状态为 $front=rear=100$ 。经过一系列正常的入队与退队
15 后， $front=rear=99$ ，则循环队列中的元素个数为 ()
16 A 1
17 B 2
18 C 99
19 D 0或者100
20
21 4. 以下 () 不是队列的基本运算？
22 A 从队尾插入一个新元素
23 B 从队列中删除第 i 个元素
24 C 判断一个队列是否为空
25 D 读取队头元素的值
26
27 5. 现有一循环队列，其队头指针为 $front$ ，队尾指针为 $rear$ ；循环队列长度为 N 。其队内有效长度为？ (1
28 队头不存放数据)
29 A $(rear - front + N) \% N + 1$
30 B $(rear - front + N) \% N$
31 C $(rear - front) \% (N + 1)$
32 D $(rear - front + N) \% (N - 1)$

4.4 双端队列

```
1 #define Deque_Elem_Type int
2
3 typedef struct Deque
4 {
5     Deque_Elem_Type *base;
6     size_t          capacity;
7     int front;
8     int rear;
9 }Deque;
10
11 void DequeInit(Deque *pd);
12 void DequePushFront(Deque *dq, Deque_Elem_Type v);
13 void DequePopFront(Deque *dq);
14 void DequePushBack(Deque *dq, Deque_Elem_Type v);
15 void DequePopBack(Deque *dq);
```