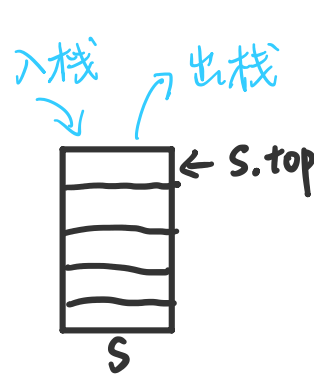


逻辑结构知识点

栈	• 只允许在 一端进行插入或删除 操作的线性表
栈顶	• 线性表允许进行插入或删除的那一段
栈底	• 固定的、不允许进行插入或删除的那一段
操作特性	• 先进后出
数学性质	• n 个不同元素进栈，出栈元素的不同排列的个数为 $\frac{1}{n+1} C_{2n}^n$ (卡特兰数)




存储结构知识点

1、栈的顺序存储结构

顺序栈的定义	<ul style="list-style-type: none"> 采用顺序存储的栈 利用一组地址连续的存储单元存放自栈底到栈顶的数据元素 		
顺序栈的实现	<pre>#define Maxsize 50 typedef struct { Elemtype data[Maxsize]; //存放栈中元素 int top; //栈顶指针 } SqStack;</pre> <p>栈顶指针 S.top, 初始时设置 S.top=-1; 栈顶元素: S.data[S.top]</p> <p>进栈操作 栈不满时, 栈顶指针先加1, 再送值到栈顶元素</p> <p>栈空条件 S.top== -1</p>	<p>出栈操作 栈非空时, 先找栈顶元素值, 再将栈顶指针减1</p> <p>栈满条件 S.top==Maxsize-1; 栈长: S.top+1</p>	
顺序栈基本运算代码	<p>初始化</p> <pre>void InitStack(SqStack &S) { S.top = -1; }</pre> <p>进栈</p> <pre>bool Push(SqStack &S, ElemType x) { if (S.top == Maxsize - 1) //栈满 return false; S.data[++S.top] = x return true; }</pre> <p>读栈顶元素</p> <pre>bool GetTop(SqStack &S, ElemType x) { if (S.top == -1) //栈空 return false; x = S.data[S.top] return true; }</pre>	<p>判栈空</p> <pre>bool StackEmpty(SqStack S) { if (S.top == -1) //栈空 return true; else return false; }</pre> <p>出栈</p> <pre>bool Pop(SqStack &S, ElemType x) { if (S.top == -1) //栈空 return false; S.data[S.top--] = x return true; }</pre>	
共享栈	<p>定义</p> <ul style="list-style-type: none"> 利用栈底位置相对不变的特性, 可让两个顺序栈共享一个一维数组空间 将两个栈的栈底分别设置在共享空间的两端, 两个栈顶向共享空间的中间延伸 <p>特点</p> <ol style="list-style-type: none"> ① top0=-1时, 0号栈长度为 0; top1=Maxsize-1时, 1号栈长度为 0 ② top1-top0=1时, 栈满 ③ 0号栈进栈, top0加1, 再取值; 1号栈进栈, top1减1, 再取值 <p>目的</p> <ul style="list-style-type: none"> 更有效地利用存储空间 两个栈的空间相互调节, 只有在整个存储空间都被占满时才发生上溢 		

2、栈的链式存储结构

概念	<ul style="list-style-type: none"> 采用链式存储的栈 
优点	<ul style="list-style-type: none"> 便于多个栈共享存储空间和提高其效率, 且不存在栈满上溢的情况
特点	<ul style="list-style-type: none"> 通常采用单链表实现, 并且所有操作都是在单链表的表头进行; 这里规定链表没有头结点, Lhead指向栈顶元素
链式栈的实现	<pre>typedef struct LinkNode { ElemType data; //存放栈中元素 struct LinkNode *next; //栈顶指针 } * LinkStack;</pre>

栈的运算/操作

InitStack(&S)	初始化栈	• 初始化一个空栈S
StackEmpty(S)	判断栈是否为空	• 空则返回True
Push(&S,x)	进栈	• 若S未满,则将x加入使之成为新栈顶
Pop(&S,&x)	出栈	• 若S非空,则弹出栈顶元素,并用x返回
GetTop(S,&x)	读栈顶元素	• 若栈非空,则用x返回栈顶元素
DestroyStack(&S)	销毁栈	• 销毁并释放栈S占用的存储空间

栈的应用

<p>括号匹配</p> <ul style="list-style-type: none"> • 初始设置一个空栈，顺序读入括号 • 若是右括号，置于栈顶 • 若是左括号，压入栈中 • 算法结束时，栈为空，否则括号序列不匹配(具体例子见下面考点3的例题) <p>表达式求值</p> <ul style="list-style-type: none"> • 考点1: 后缀表达式和正常表达式的相互转换 • 考点2: 涉及到编译原理的求值过程 	<p>考点3: 中缀到后缀表达式转换的过程(见王道P98)</p> <p>12 【2014 统考真题】设栈初始为空，将中缀表达式 $a*b*(c-d)-e/f$ 转换为等价的后缀表达式的过程中，当扫描到 f 时，栈中的元素依次是()。</p> <p>A. $a, c, *$ B. $a, c, *, d, -$ C. $a, c, *, d, -, /$ D. $a, c, *, d, -, /, *$</p> <p>解析:</p> <p>将中缀表达式转换为后缀表达式的方法:</p> <ol style="list-style-type: none"> ① 从左到右扫描表达式 ② 遇到运算符，压入栈中 ③ 遇到左括号，压入栈中 ④ 遇到右括号，弹出栈顶运算符，直到遇到左括号为止 <p>对于表达式 $a*b*(c-d)-e/f$，扫描到 f 时，栈中的元素依次是 $a, c, *, d, -, /$。</p>
---	--

队列

逻辑结构知识点

队列	操作受限的线性表
队头	允许删除的一端
队尾	允许插入的一端
操作特性	先进先出

存储结构知识点



1、队列的顺序存储结构

顺序队列的定义	<ul style="list-style-type: none"> • 分配一块连续的存储空间存放队列中的元素 • 设两个指针 <ul style="list-style-type: none"> ◦ 队头指针<code>front</code>指向队头元素 ◦ 队尾指针<code>rear</code>指向队尾元素的下一个位置 		
顺序队列的实现	<pre>#define MaxSize 50 typedef struct { ElemType data[MaxSize]; int front, rear; } SqQueue;</pre>	<p>初始状态 $Q.front == Q.rear == 0$</p> <p>进队操作 队不满时，先送值到队尾元素，再将队尾指针加1</p> <p>出队操作 队不空时，先取队头元素值，再将队头指针加1</p>	<p>队满操作</p> <ul style="list-style-type: none"> • $Q.rear == \text{MaxSize}$ 不能作为队列满的条件 • 只有一个元素仍满足该条件（假退出）
循环队列的定义	• 把存储队列元素的表从逻辑上视为一个环		
循环队列的实现 (以下方法一为例)	<p>① 初始: $Q.front = Q.rear = 0$</p> <p>② 入队: 队尾加: $Q.rear = (Q.rear + 1) \% \text{MaxSize}$</p> <p>③ 出队: 队头减: $Q.front = (Q.front + 1) \% \text{MaxSize}$</p> <p>④ 队列已满: $(Q.rear + \text{MaxSize} - Q.front) \% \text{MaxSize}$</p> <p>⑤ 队空条件: $Q.front == Q.rear$</p> <p>⑥ 队满条件: $(Q.rear + 1) \% \text{MaxSize} == Q.front$</p>		
如何区分循环队列 队空还是队满	<p>方法一〔常用方法, 示例图如上〕</p> <p>定义 牺牲一个单元来区分队空和队满 入队时少用一个队列单元 约定以“队头指针在队尾指针的下一位置作为队满的标志”</p> <p>队空条件 $Q.front == Q.rear$</p> <p>队满条件 $(Q.rear + 1) \% \text{MaxSize} == Q.front$</p> <p>元素个数 $(Q.rear - Q.front + \text{MaxSize}) \% \text{MaxSize}$</p>	<p>方法二</p> <p>类型中增设表示元素个数的数据成员</p> <p>$Q.size == 0$</p> <p>$Q.size == \text{MaxSize}$</p>	<p>方法三</p> <p>类型中增设tag数据成员，以区分是队满还是队空</p> <p>$tag == 0$且因删除导致$Q.front == Q.rear$</p> <p>$tag == 0$且因插入导致$Q.front == Q.rear$</p>
循环队列的操作代码	<p>判队空</p> <pre>bool isEmpty(SqQueue &Q) { if (Q.rear == Q.front) return true; else return false; }</pre> <p>判队空</p> <pre>bool DeEmpty(SqQueue &Q, ElemType &x) { if (Q.rear == Q.front) return false; x = Q.data[Q.front]; Q.front = (Q.front + 1) % MaxSize; return true; }</pre>		

2、队列的链式存储结构

链式队列的定义	<ul style="list-style-type: none"> • 实质上是一个同时带有队头指针和队尾指针的单链表 • 头指针指向队头节点，尾指针指向队尾节点，即单链表的最后一个节点 • 删除操作时，通常仅需修改头指针 • 当队列只有一个元素时，删除后队列为空，修改尾指针为read=front 			
链式队列的实现	<pre>typedef struct LinkNode { ElemType data; struct LinkNode *next; } LinkNode; typedef struct { LinkNode *front, *rear; } LinkQueue;</pre>			
链式队列的操作代码	<p>初始化</p> <pre>void InitQueue(LinkQueue &Q) { Q.front = Q.rear = (LinkNode *) malloc(sizeof(LinkNode)); Q.front->next = NULL; }</pre>	判队空	<pre>bool isEmpty(LinkQueue Q) { if (Q.rear == Q.front) return true; else return false; }</pre>	
	<p>入队</p> <pre>bool EnQueue(LinkQueue &Q, ElemType x) { LinkNode *s = (LinkNode *)malloc(sizeof(LinkNode)); s->data = x; s->next = NULL; Q.rear->next = s; Q.rear = s; }</pre>	出队	<pre>bool DeQueue(LinkQueue &Q, ElemType &x) { if (Q.rear == Q.front) return false; LinkNode *p = Q.front->next; x = p->data; Q.front->next = p->next if (Q.rear == p) Q.rear = Q.front; free(p); return true; }</pre>	

3、双端队列

定义	允许两端都可以进行入队和出队操作的队列 将队列的两端分别称为前队和后队	
特点	其元素的逻辑结构仍是线性结构	
分类	输出受限的双端队列 允许在一段进行插入和删除，但在另一端只许插入的双端队列 做删的时候输出固定，看输入	
	输入受限的双端队列 允许在一段进行插入和删除，但在另一端只许删除的双端队列 做删的时候输入固定，看输出	

队列的运算/操作

InitQueue(&Q)	初始化队列	• 构造一个空队列Q
QueueEmpty(Q)	判队列为空	• 空则返回True
EnQueue(&Q,x)	入队	• 若队列未满, 将x加入, 使之成为新的队尾
DeQueue(&Q,&x)	出队	• 若队列非空, 删除表头元素, 并用x返回
GetHead(&Q,&x)	读队头元素	• 若队列非空, 则将队头元素赋值给x

队列的应用

层次遍历

• 使用队列是为了保存下一步的处理顺序

	T=0	T=1	T=2
1	A	A	
2	D, E, F, H	B, C	A
3	G, I, J, K, L, M, N, O	C, D	A, B
4	E, F, H	D, E, F	A, B, C
5	D, E, F, H	E, F	A, B, C, D
6	G, I, J, K, L, M, N, O	F, G, H, I	A, B, C, D, E, F
7	F, G, I	G, H, I, J	A, B, C, D, E, F, G, H, I, J
8	G, H, I, J, K, L, M, N, O		

在计算机系统中的应用

- 解决主机与外部设备之间速度不匹配的问题（如打印机与主机，设置一个**打印数据缓冲区**）
- 解决由多个用户引起的资源竞争问题（如**CPU资源的竞争**）
- 页面替换算法(FIFO算法)

数组和特殊矩阵

本部分重点	如何将矩阵更有效地存储在内存中，并能方便地提取矩阵中的元素
数组的定义	由n个相同类型的数据元素构成的有限序列
数组的特定	<ul style="list-style-type: none"> 数组一旦被定义，其维数和维界就不再改变 除结构的初始化和销毁外，数组只会有存储元素和修改元素的操作
数组的数据结构	<p>一个数组的所有元素在内存中占用一段连续的存储空间</p> <p>对于多维数组，有两种映射方法</p> <p>按行优先 先行后列，先存储行号较小的元素，行号相等先存储列号较小的元素</p> <p>按列优先 先列后行，先存储列号较小的元素，列号相等先存储行号较小的元素</p>
特殊矩阵相关概念	<p>压缩存储 为多个值相同的元素只分配一个空间，对零元素不分配存储空间，其目的是节省存储空间</p> <p>特殊矩阵 具有许多相同矩阵元素或者零元素，并且这些相同矩阵元素或零元素的分布有一定规律性的矩阵</p> <p>常见的特殊矩阵有对称矩阵、三角矩阵（第一行和最后一行两个元素，其余行3个元素）、对角矩阵</p> <p>通常每个值对应的行列用特殊代号记好，不用记那么多公式，可以用线性代数的知识解决</p>
特殊矩阵的压缩存储方法	<ul style="list-style-type: none"> 找到特殊矩阵中值相同的矩阵元素的分布规律 把这些呈现规律性分布的、值相同的多个矩阵元素存储到一个存储空间中
稀疏矩阵的定义	非0元素非常少的矩阵
稀疏矩阵的特点	<ul style="list-style-type: none"> 稀疏矩阵压缩存储后便失去了随机存取特性 可以使用三元组表和十字链表法压缩存储稀疏矩阵