课程要求

平时作业+每周上机+每月竞赛+大作业：40%

期中考试：20%

期末考试：40%

Chapter1 数据结构概论

1.1-1.2 数据结构

数据，数据元素

Data_Structure = { D , R } **数据元素 + 数据元素之间的关系**

数据的逻辑结构（**数据结构**）：用户视角，面向问题

数据的物理结构（**存储结构**）：数据逻辑结构在计算机中的物理存储方式，逻辑结构的存储映像 (数组，链表.....)

数据结构分类：

线性结构：除第一个和最后一个元素之外，每个元素都有其前驱与后继 “一对一”

非线性结构：（1）层次结构 “一对多”（如树） （2）网状结构 “多对多”（如图） （3）松散结构（如集合）

存储结构分类：顺序结构，链式结构，索引结构，散列结构

常见数据运算：插入运算，删除运算，修改运算，查找运算，排序运算

数据类型：一种类型，以及定义于这个值集合上一系列操作的总称

抽象数据类型：使用与实现分离，实现封装与信息屏蔽（使用者利用公共接口，不用关心物理实现）

断言：assert.h，例：assert (x>0) ，如果x > 0，继续执行；反之报错：源文件名称，行号。

1.4-1.5 算法

算法：输入（可以为0个），输出（至少为1个），确定性（结果确定，不会不同次执行结果不一样），有穷性（不能陷入无限循环，如：求解Π），有效性（每一步足够基本）

评价算法：正确性（结果正确），可使用性（用户友好性），可读性（便于理解修改），效率（包括时间代价和空间代价），健壮性（对错误情况的处理），简单性（所用数据结构和方法的简单程度）

时间代价：算法执行时所耗费的时间

空间代价：算法执行过程中所占用的存储空间

算法效率的衡量方法：事前估计/后期测试

后期测试：

（1）插装时间函数 time();

（2）计算程序步：注释、声明语句（0）；表达式、赋值语句（1）；循环语句：每执行一次，程序步数为1；

渐进的时间复杂度：用规模的数量级来表示时间复杂度

大O渐进表示：取时间复杂度 T(n) 中增长最快的一项 *f(n)*，使得T(n) 与 *f(n)* 增长率相同，表示为 O(*f(n)*)

加法规则：针对并列程序段：取多个程序段中，复杂度级别较高的程序段的复杂度

乘法规则：针对嵌套程序段：每层循环的渐进时间复杂度相乘

乘法规则中的常数无关项：O(C) → O(1)；O(1) 表示常数计算时间

大O表示法: T(n) = O(f(n)) 的几条规则

- **加法规则　针对并列程序段**

  $T(n, m) = T1(n) + T2(m)$

  $= O(\max(f(n), g(m)))$

- **乘法规则　针对嵌套程序段**

  $T(n, m) = T1(n) * T2(m)$

  $= O(f(n)*g(m))$

- **乘法规则中的常数无关项　$O(C) \rightarrow O(1)$**

  $T(n) = O(c * f(n)) = O(f(n))$

- **O(1)表示常数计算时间**

Chapter2 线性表

线性表

顺序表，单链表（线性表的物理实现）

循环链表，双向链表（单链表的变化形式）

多项式（链表的应用）

LinearList：线性表　SeqList：顺序表　List：链表　LinkNode：链表节点

CircList：循环链表（first，last）　DblList：双向链表（llink，rlink）　Polynomal：多项式

2.1 线性表

线性表的定义：n(>=0) 个数据元素的一个有限序列，记为 L =（a_1,...,a_i,a_{i+1},...a_n）

–为简单起见，假定表中元素的数据类型相同

– 线性表中，结点和结点间的关系是一对一的

2.2 顺序表

顺序表是线性表基于数组的存储表示

将线性表中的元素相继存放在一个连续的存储空间中

特点：

（1）各表项的逻辑顺序和物理顺序保持一致

（2）各表项可以顺序访问，也可以随机访问

2.3 单链表

特点：适应表的动态增长和删除

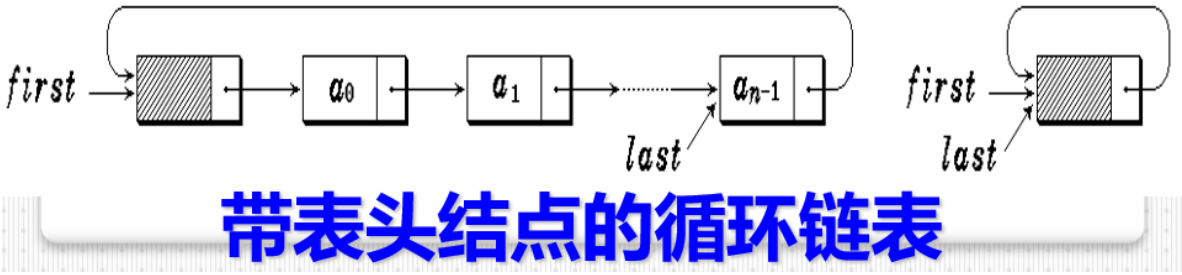
缺点：需要额外的指针存储空间

带附加头结点（表头结点）的单链表：表头结点位于表的最前端，本身不带数据，仅标志表头。

设置表头结点的目的是统一空表与非空表的操作，简化链表操作的实现

2.4 循环链表/双向链表

2.4.1 循环链表



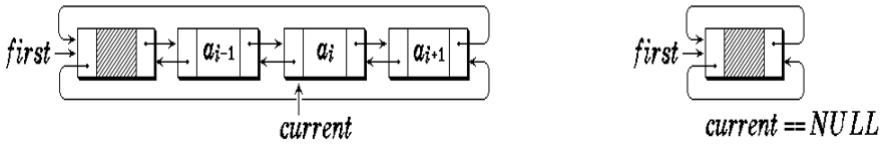
带表头结点的循环链表

- 循环链表最后一个结点的 link 指针不 为 0 (NULL)，而是指向了表的前端。
- 只要知道表中某一结点的地址，就可搜寻到所有其他结点的地址。

2.4.2 双向链表

前驱方向 ← → 后继方向

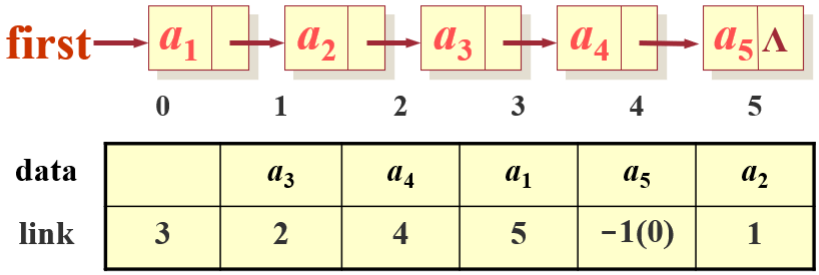
• 带头节点的双向链表表示



2.6 静态链表

用结构数组实现链表

静态链表的结构



data		a_3	a_4	a_1	a_5	a_2
link	3	2	4	5	−1(0)	1

- **0号是表头结点，link给出首元结点地址。**
- **循环链表收尾时link = 0，回到表头结点。如果不是循环链表，收尾结点指针link = −1。**
- **link指针是数组下标，因此是整数。**

Chapter3 栈与队列

Stack：栈 SeqStack：顺序栈 LinkedStack：链式栈 DualStack：双栈

Queue：队列 SeqQueue：顺序队列（循环队列）

QueueNode：队列结点类定义 LinkedQueue：链式队列 PQueue：优先级队列

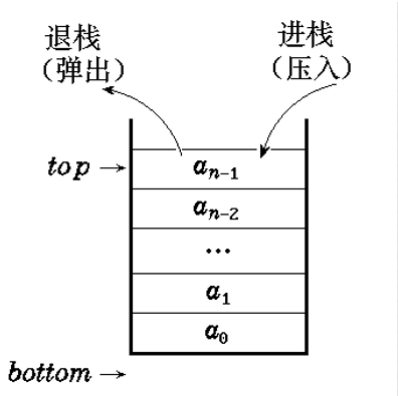
3.1 栈

只允许在一端插入和删除的线性表

允许插入和删除的一端称为栈顶 (top)，另一端称为栈底(bottom)

特点：**后进先出(LIFO)**

3.1.1 栈的数组表示 — 顺序栈



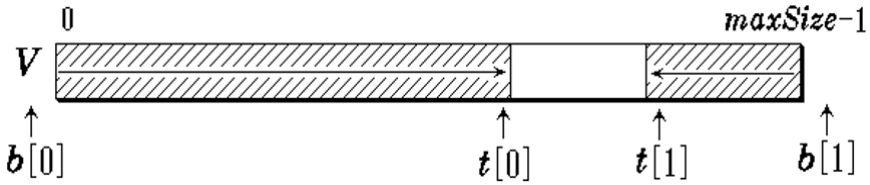
常用操作：

```
template <class Type> class Stack {
public:
    Stack ( int=10 );              //构造函数
    void Push ( const Type & item); //进栈
    Type Pop ( );                  //出栈
    Type GetTop ( );               //取栈顶元素
    void MakeEmpty ( );            //置空栈
    int IsEmpty ( ) const;         //判栈空否
    int IsFull ( ) const;          //判栈满否
}
```

private成员：

```
private:
   int top;                    //栈顶指针
   Type *elements;             //栈元素数组
   int maxSize;                //栈最大容量
   void overflowProcess();     //栈的溢出处理
```

双栈共享一个栈空间：



两个栈共享一个数组空间V[maxSize]

设立栈顶指针数组 t[2] 和栈底指针数组 b[2]

t[i]和b[i]分别指示第 i 个栈的栈顶与栈底

初始 t[0] = b[0] = -1，t[1] = b[1] = maxSize

栈满条件：t[0]+1 == t[1] //栈顶指针相遇

栈空条件：t[0] = b[0]或t[1] = b[1] //栈顶指针退到栈底

3.1.2 栈的链接表示 — 链式栈

链式栈无栈满问题，空间可扩充；插入与删除仅在栈顶处执行

链式栈的栈顶在链头；适合于多栈操作

3.1.3 栈的应用：表达式的计算

1. 后缀表达式的计算

在后缀表达式的计算顺序中已隐含了加括号的优先次序，括号在后缀表达式中不出现。

（1）从左向右顺序地扫描表达式，并用一个栈暂存扫描到的操作数或计算结果。

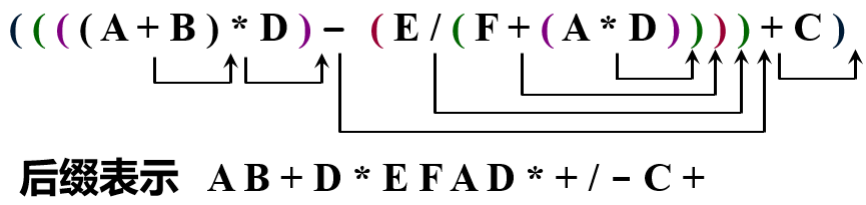
（2）扫描中遇操作数则压栈；遇操作符则从栈中退出两个操作数，计算后将结果压入栈

（3）最后计算结果在栈顶

（碰到双目运算符时，先弹出来的在运算符后面）

2. 中缀表示转后缀表示

- **先对中缀表达式按运算优先次序加上括号，再把操作符后移到右括号的后面并以就近移动为原则，最后将所有括号消去。**
- **如中缀表示 (A+B)*D−E/(F+A*D)+C，其转换为后缀表达式的过程如下：**



3. 利用栈将中缀表示转换为后缀表示

优先级	操作符
1	单目-、！
2	*、/、%
3	+、-
4	<、<=、>、>=
5	==、!=
6	&&
7	||

操作符 ch	;	(	*, /, %	+, −	)
isp (栈内)	0	1	5	3	6
icp (栈外)	0	6	4	2	1

□ isp叫做栈内(in stack priority)优先数

□ icp叫做栈外(in coming priority)优先数。

□ 操作符优先数相等的情况只出现在括号配对或栈底的“;”号与输入流最后的“;”号配对时。

4. 中缀表示转换为后缀表示的算法

1. 操作符栈初始化，将结束符‘;’进栈。然后读入中缀表达式字符流的首字符ch。
2. 重复执行以下步骤，直到ch = ‘;’，同时栈顶的操作符也是‘;’，停止循环。
3. 若ch是操作数直接输出，读入下一个字符ch。
4. 若ch是操作符，判断ch的优先级icp和位于栈顶的操作符op的优先级isp：
   （1）若 icp(ch) > isp(op)，令ch进栈，读入下一个字符ch。
   （2）若 icp(ch) < isp(op)，退栈并输出。
   （3）若 icp(ch) == isp(op)，退栈但不输出，若退出的是“(”号读入下一个字符ch。

操作符进入栈内之后，优先级+1；左括号栈外最高，入栈最低；右括号反之

3.2 栈与递归

3.2.1 递归的定义

若一个对象部分地包含它自己，或用它自己给自己定义，则称这个对象是递归的；若一个过程直接地或间接地调用自己，则称这个过程是递归的过程。

以下三种情况常常用到递归方法。

（1） 定义是递归的

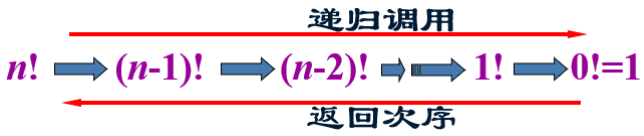
（2）数据结构是递归的

（3） 问题的解法是递归的

3.2.2 递归过程与递归工作栈

递归过程在实现时，需要自己调用自己。

层层向下递归，退出时的次序正好相反：



主程序第一次调用递归过程为**外部调用**

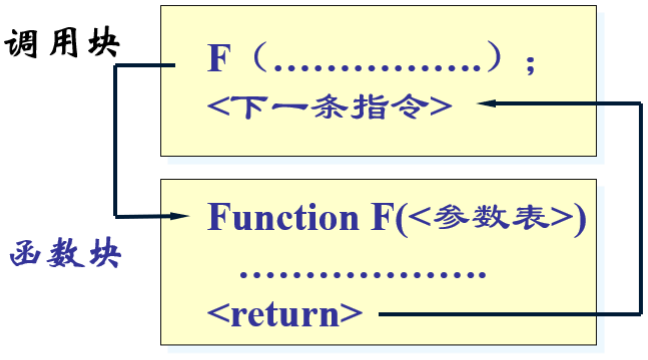
递归过程每次递归调用自己为**内部调用**

它们返回调用它的过程的地址不同

递归工作栈

每一次递归调用时，需要为过程中使用的参数、局部变量等另外分配存储空间。

每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织



递归改非递归的方法

（1）通过迭代实现非递归过程

（2）借助栈实现非递归过程

3.3 队列

3.3.1 队列的定义和特性

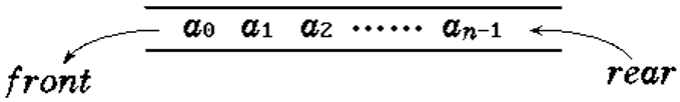
- **定义**

队列是只允许在一端删除，在另一端插入的顺序表

允许删除的一端叫做队头(front)，允许插入的一端叫做队尾(rear)。

- **特性**

先进先出(FIFO, First In First Out)



3.3.2 队列的抽象数据类型

队列的抽象数据类型

```
template <class E>
class Queue {
public:
   Queue() { };          //构造函数
   ~Queue() { };         //析构函数
   virtual bool EnQueue(E x) = 0;        //进队列
   virtual bool DeQueue(E& x) = 0;       //出队列
   virtual bool getFront(E& x) = 0;      //取队头
   virtual bool IsEmpty() const = 0;     //判队列空
   virtual bool IsFull() const = 0;      //判队列满
};
```

3.3.3 队列的数组存储表示 — 顺序队列

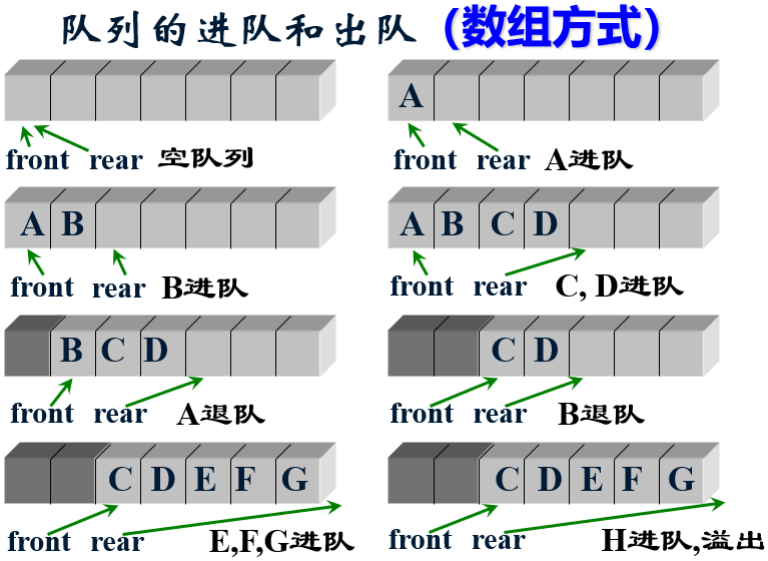
```
protected:
   int rear, front;              //队尾与队头指针
   E *elements;                  //队列存放数组
   int maxSize;                  //队列最大容量
public:
   SeqQueue(int sz = 10);    //构造函数

~SeqQueue() { delete[ ] elements; }  //析构函数
bool EnQueue(E x);        //新元素进队列
bool DeQueue(E& x);       //退出队头元素
bool getFront(E& x);      //取队头元素值
void makeEmpty() { front = rear = 0; }
bool IsEmpty() const { return front == rear; }
bool IsFull() const
   { return ((rear+1)% maxSize == front); }
int getSize() const
   { return (rear−front+maxSize) % maxSize; }
```

进队：新元素在rear处加入，rear = rear + 1。

出队：取出下标为 front 的元素，front = front + 1。

队空时：rear == front

队满时：rear == maxSize （假溢出）



解决假溢出的办法之一：将队列元素存放数组首尾相接，形成循环队列

3.3.4 循环队列

队列存放数组被当作首尾相接的表处理。

队头、队尾指针加1时从maxSize-1直接进到0,可用语言的取模(余数)运算实现。

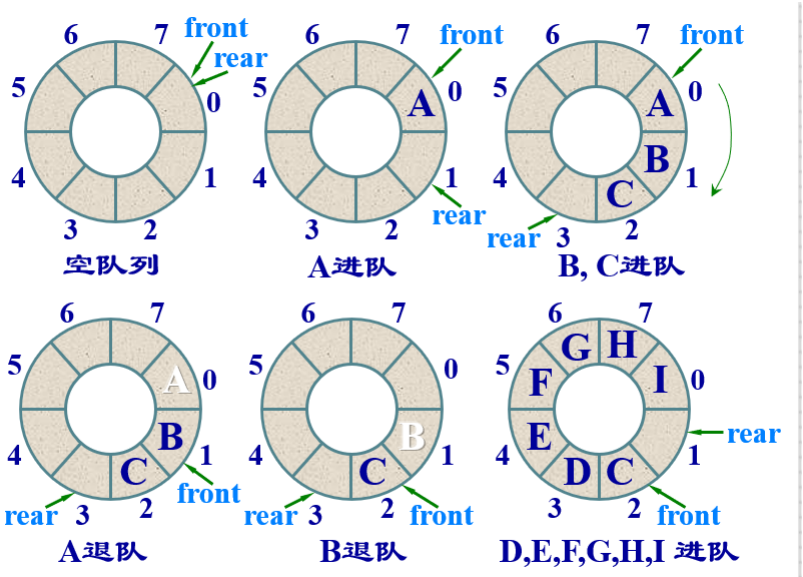
队头指针进1: front = (front+1) % maxSize;

队尾指针进1: rear = (rear+1) % maxSize;

队列初始化：front = rear = 0;

队空条件：front == rear;

队满条件：(rear+1) % maxSize == front（实际上为了区分队空，空了1个元素）

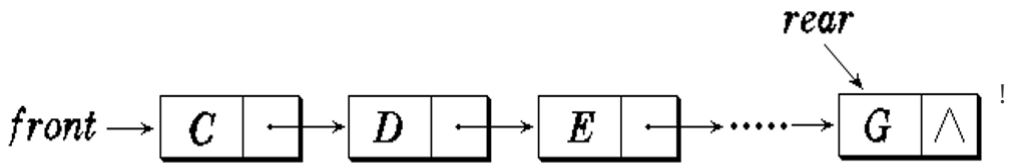


3.3.5 队列的链接表示 — 链式队列

队头在链头，队尾在链尾。

链式队列在进队时无队满问题，但有队空问题。

队空条件为 front == NULL




```
template <class E>
class LinkedQueue {
private:
    QueueNode<E> *front, *rear;   //队头、队尾指针
public:
   LinkedQueue() : rear(NULL), front(NULL) { }
   ~LinkedQueue();
   bool EnQueue(E x);
   bool DeQueue(E& x);
   bool GetFront(E& x);
   void MakeEmpty();          //实现与~LinkedQueue()同
   bool IsEmpty() const { return front == NULL; }
};
```

3.4 优先级队列

是不同于先进先出队列的另一种队列。每次从队列中取出的是具有最高优先权的元素

任务编号	1	2	3	4	5
优先权	20	0	40	30	10
执行顺序	3	1	5	4	2

数字越小，优先权越高

出现相同的优先级的元素时，按FIFO的方式处理

3.5 递归的内部实现原理

3.5.1 递归调用和返回

（1）调用时：

1. 保留现场（局部变量、返回地址、形参→入栈）
2. 虚实结合（计算实参→形参）
3. 转入执行

（2）返回时：

1. 恢复现场（局部变量{赋给相应变量} 、返回地址{赋给L}、形参{赋给相应变量}→ 出栈）
2. Goto L

3.5.2 递归程序的验证

1. 察看n = 0 时，结果是否正确（在递归出口时，功能正确）
2. 假设n < k 时，满足功能（子程序的功能正确）
3. 证明当n = k 时，功能实现

3.5.3 递归程序的模拟（递归 → 非递归）

转换规则：

1. 设一个栈，并置为空
2. 程序入口处设立标号L0
3. 用以下操作替换递归调用

   a. 保留现场（局变量，形参，返回地址等 → 入栈）

   b. 计算实参值 → 形参

   c. 转入 L0
4. 返回时，执行

   a. 恢复现场（出栈 → 局变量，形参，返回地址等；其中X = 返回地址 ）

   b. 转入 X

   c. 变参，函数值 → 回传变量
5. 其他的非递归的部分可照搬

简化规则：

1. 若递归程序中只有一处递归调用，在转换时，返址不必入栈（返址唯一）
2. 在尾递归调用时，入栈操作等可以不必执行（入栈的内容是为了返回后继续使用，若没有后续操作则不必保存上述内容了）

（例子见第3章ppt！）