# 第3章 表、栈和队列

## 1 抽象数据类型（ADT）

是一些操作的集合（并、交、查找、测定大小、取余），是数学的抽象、模块化设计的扩充。

## 2.表ADT

大小为0的是空表。有操作：Empty、PrintList、MakeEmpty、Insert、Delete、FindKth（返回k位置的元素）、Find（返回关键词首次出现的位置）、Next、Precious（上一个）。

### 2.1表的简单数组实现

连续存储，数组的大小估计得大一些，表的大小必须事先已知。Find、PrintList、Insert、Delete需要线性时间，FindKth需要常数时间。

### 2.2链表

不连续存储，最后一个单元的Next指针指向NULL（ASCI C规定NULL为0），第0个单元的元素为空，只存指针，作为表头/哑节点。因为：①不存在从所给定义出发在表的前面插入元素的真正显性的方法，②从表的前面实行删除时一个特殊情况。

Find、PrintList、FindKth需要线性时间，Delete、Insert需要常数时间。Insert需要一次malloc 命令调用从系统得到的一个新单元，并执行两次指针调整。Delete需要free命令释放动态空间，执行一次指针调整。

Find：while(p!=NULL && p->Next->Element != X)

p = p->Next;

### 2.3常见错误

1.变量初始化失败：无论何时只要确定一个指向，那么必须保证该指针不是NULL，比如：p->Next。P不可以为NULL，p->next可以为NULL。

2.何时使用或何时不使用malloc来获取一个新的单元:malloc需要包含stdlib.h。如果从未对一个链表进行过删除操作，那么调用malloc的次数应该等于表的大学，有表头再加1。Malloc(sizeof(ptrToNode))是合法的，但是它并不给结构体分配足够的空间，只给指针分配一个空间。

3.当空间不需要时，可以使用free命令通知系统来收回它，free(p)收回:p在指向的地址没变。

### 2.4双（向）链表

增加了空间需求，使得插入和删除的开销增加一倍，因为有更多的指针需要定位。简化了删除操作，因为不再被迫使用一个指向前驱元的指针来访问一个关键字。

### 2.5循环链表

可以有表头，也可以无表头。

### 2.6例子

#### 2.6.1一元多项式ADT

1.使用结构体

typedef struct{

int CoeffArray[MaxDegree+ 1];//系数数组，MaxDegree是最大系数

int HighPower;//最高幂次

}\*Polynomial;

2.使用单链表:以次数递减的顺序排序

typedef struct Node{

int Coefficient;//系数

int Exponent;//指数

PtrToNode Next;

] Node \*PtrToNode

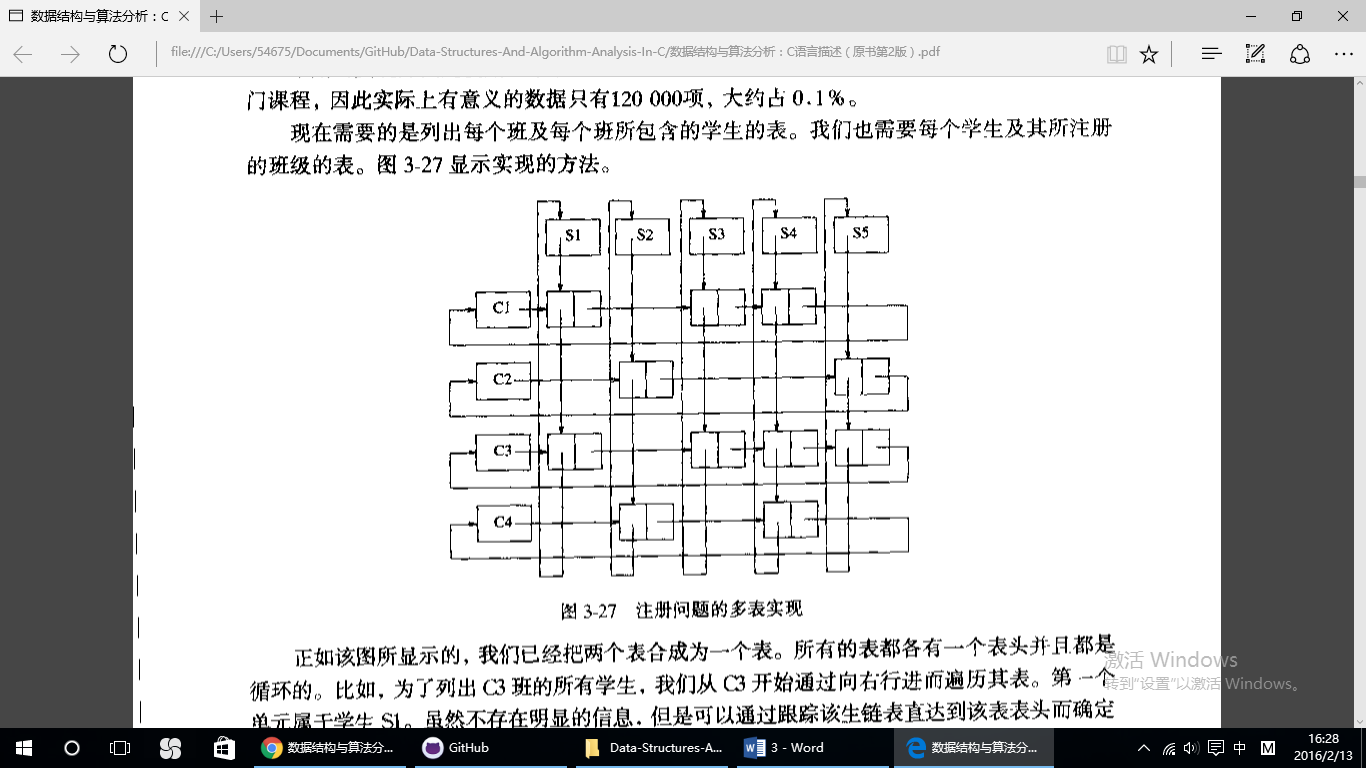
#### 2.6.2基数排序/卡式排序

桶式排序：对N个整数排序，范围从1-M。设置一个数组Count[M]，初始化为0，Count有M个桶。当Ai被读入时Count[Ai]增加1.最后扫描数组打印出排好序的表。花费O(M+N)。

基数排序是桶式排序的推广，有的数范围太大，桶的数量太多。采用多趟桶式排序，使用最低有效位优先的方式进行排序，每一位排序完后重新串联、对下一位进行排序。唯一可能出错的是两个数如果出自同一个桶但顺序却是错误的，不过前面各趟排序保证了当几个数进入一个桶的时候，它们是以顺序的顺序进入的。话费O(P(N+M))，P是排序的趟数。

#### 2.6.3多重表

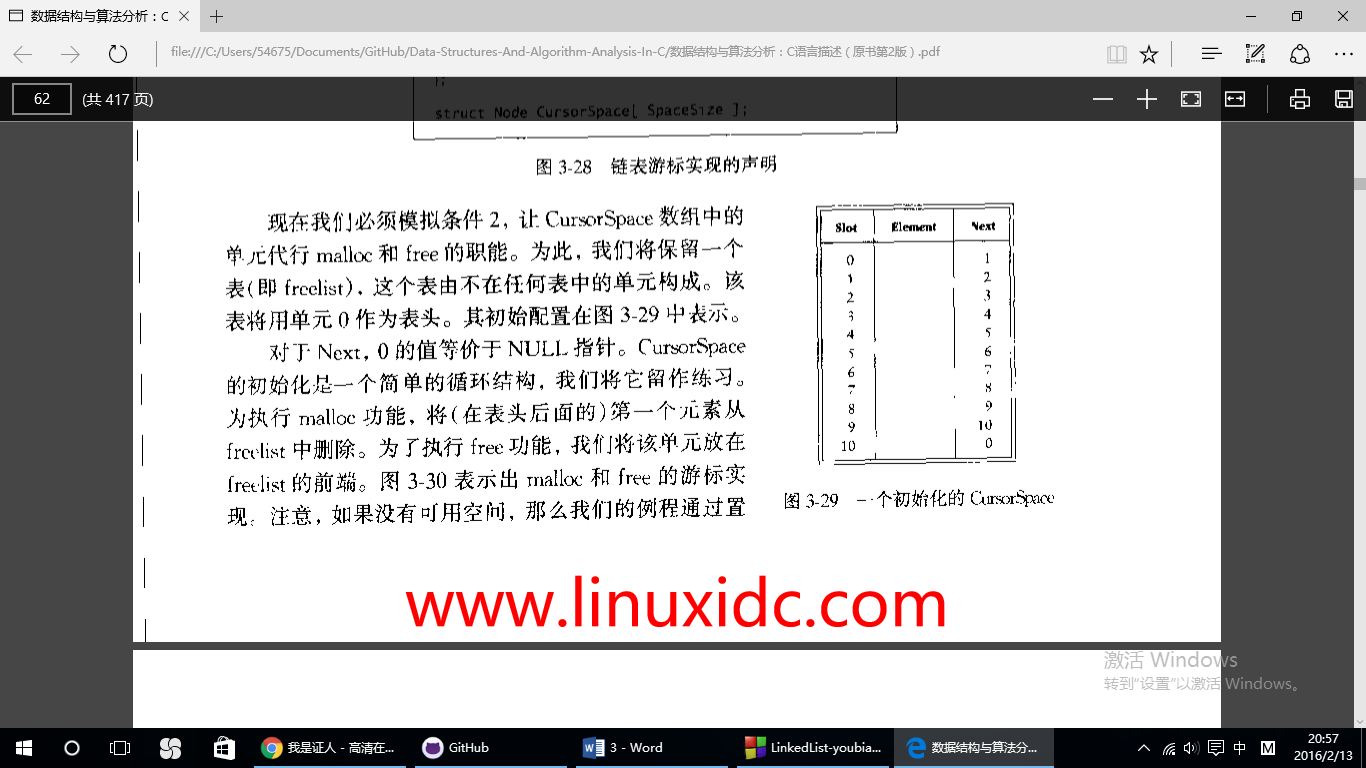
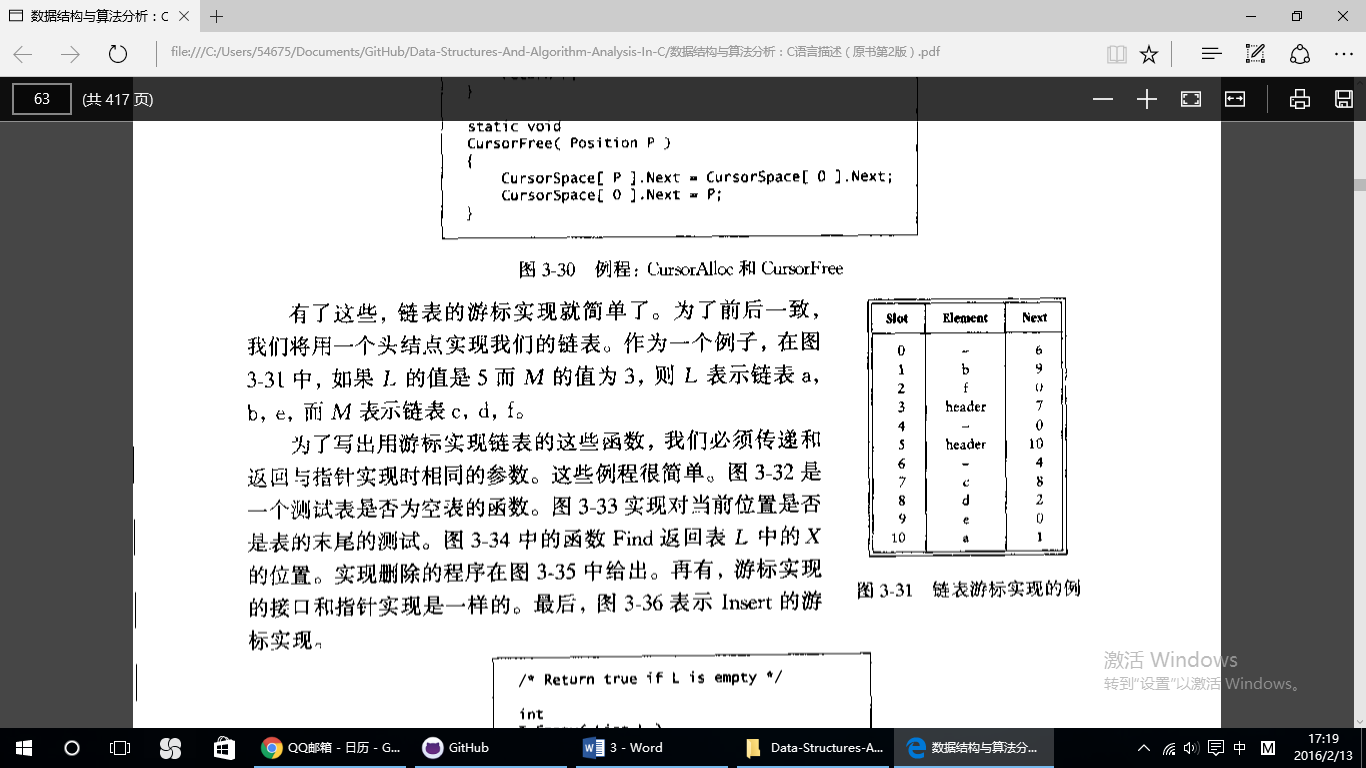
假设：大学有40000名学生、2500门课程，需要生成2种类型报告：列出每个课程的学生、列出每个学生的课程。①使用二位数组，将有2500\*40000项。②我们使用多重表：二维表的实现，接近数组，但只留下了有用的项。课程从左往右遍历时，虽然不存在明显学生信息，但是可以通过跟踪该链表直达到该表表头而确定该生信息。



### 2.7链表的游标实现

需要链表又不能使用指针，将利用游标实现。指针链表的特性：①数组存储在一组结构体中，每隔结构体包含数据及指向下一个结构体的指针；②一个新的结构体可以通过调用malloc而从系统全局内存得到，并可通过free释放，游标指针也要符合这两个特性。

我们保留一个表（freelist），这个表由不在任何表中的单元构成，将用CursorSpace中的单元0作为表头。对于Next，0的值等价于NULL指针，CursorSpace初始化是一个简单的循环结构。为执行malloc功能，将在表头后面的第一个元素从freelist中删除；为了执行free功能，我们将该单元放在freelist的前端。从freelist删除的单元是刚刚由free放在那里的单元，最后被放在freelist的单元是被最先拿走的单元。



## 3.栈ADT

### 3.1栈模型/LIFO（先进后出）表

Push进栈，pop出栈，top取栈顶元素。

## 3.2栈的实现

栈是一个表，人和实现表的方法都能实现栈。一种是使用指针，一种是使用数组。

#### 3.2.1栈的链表实现

在表的顶端插入来实现push，删除表顶端来实现pop。这种方法的缺点是对于malloc和free的调用的开销是昂贵的。

#### 3.2.2栈的数组实现

每个栈有一个TopOfStack，对于空栈，它等于-1（空栈的初始化）。由于存在多个栈，因此Stack数组和TopOfStack是代表一个栈的结构的一部分。使用全局变量和固定名字来表示这种数据结构几总是有害的，因为大多数情况下总是存在多于一个的栈。这些操作不仅以常数时间运行，而且是以非常快的常数时间运行。在某些机器上，若在带有自增和自减寻址功能的寄存器上操作，则Push和pop都可以写成一条机器指令。栈很有可能是在数组之后最基本的数据结构。

影响栈的执行效率的问题是错误检测。

1.结构：

struct Stack{

int val[105];

int capacity;//容量

int TopOfStack;

};

2.初始化：（不是说好的不用指针吗！骗子！）

Stack \*CreateStack(int MaxElements){

if(MaxElements >= 105){

cout<<"这个栈太大了，无法建立！";

return;

}

Stack \*S = new Stack;

if(!S){

cout<<"越界";

return;

}

S->val = new [105] int;

if(!(S->val)){

cout<<"越界";

return;

}

S->capacity = MaxElements;

S.TopOfStack = -1;

}

#### 3.2.3应用

**1.平衡符号：**

做一个空栈。读入字符知道文件尾。如果字符是一个开放符号，则将其推入栈中。如果字符是一个封闭符号，则当栈空时报错。否则，将栈元素弹出。如uotanchude符号是开放符号，也报错。在文件尾，如果栈非空则报错。

**2.后缀表达式的计算：**

前中后缀表达式：区别在于运算符相对与操作数的位置不同，前缀表达式的运算符位于与其相关的操作数之前。

计算方法：当见到一个数时就把它推入栈；在遇到一个运算符时该运算符就作用于从该栈弹出的两个数（符号）上，将所得结果推入栈中。

时间复杂度：O(N)，因为对输入中的每个元素的处理都是由一些栈操作组成从而花费常数时间。

算法优点：简单，当一个表达式以后缀记号给出时，没有必要知道任何优先原则。

**3.中缀到后缀的转换：**

当读到操作数：直接输出。

当读到操作符、括号：压入栈。如果压入一个右括号，则将栈元素输出，将弹出的符号写出知道我们遇到一个左括号，左右括号不输出。如果压入其他符号，那么从栈中弹出栈元素直到发现更低优先级的元素为止（这里不移走左括号，左括号的优先级最高）。当从栈弹出元素的工作完成后，我们再将操作符压入栈中。

当读到输入末尾：把栈元素弹出直至为空。

需要O(N)时间并经过一趟输入后工作完成。

**4.函数调用：**

函数调用过程：主函数将所有例如局部变量的值和返回地址等存储起来。这些变量一般由编译器指派给机器的寄存器，但存在某些冲突（所有的过程豆浆某些变量指派给1#寄存器）。都以抽象的形式存在一张纸上并被置于一个堆的顶部。然后控制转移到新函数，该函数用自己的一些值代替父函数的寄存器。当该函数要返回的时候，它查看堆顶部的那张纸病复原所有的寄存器。所有全部工作可以由栈来完成，所存储的信息或称为活动记录或成为栈帧。

栈的溢出：由于有太多同时在运行着的函数，用尽栈空间的情况总是可能发生的。当栈太大的时候，它可能触及到你的程序部分，这部分也许是主程序，也许是数据部分。发生这种情况通常是由失控递归（忘记基准情形）的指向引起的；某些完全合法并且表面上无害的程序也可以造成栈溢出。比如尾递归，尾递归是设计在最后一行的递归调用。尾递归可以通过将递归调用变成goto语句并在其前加上对函数各个参数的赋值语句而手工消除（即递归变迭代）。

递归总能被彻底除去，但这么做是相当冗长乏味的，一般方法是要求使用一个栈，而且仅当你能够把最低限度的最小值放到栈上时，这个方法才值得一用。虽然迭代程序一般说来比等价的递归程序快，但是速度优势的代价却是由于去除递归而使得程序清晰性下降。

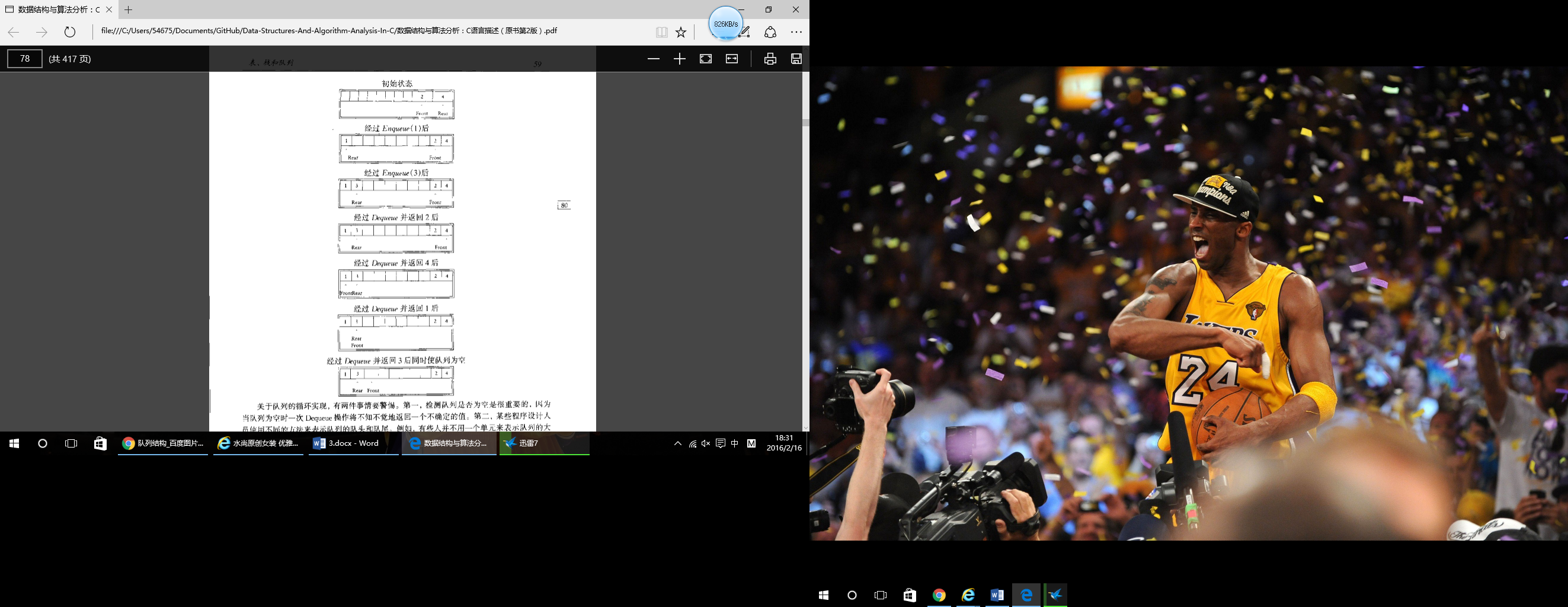
## 3.4队列

### 3.4.1队列模型

队列也是表。基本操作是：Enqueue（入队）、Deque（出队）。队尾Rear、队头Front，具体结构如下：

### 3.4.2队列的数组实现

只要Front或Rear到达数组的尾端，它就又绕回到开头，这叫循环数组实现。



关于循环队列的实现，有两件事需要警惕：第一，检测队列是否为空是很重要的，因为当队列为空时一次Dequeue操作将返回一个不确定的值。第二，某些程序设计人员使用不同的方法来表示队列的队头和队尾。注意！当队列为空的时候Rear = Front – 1。