# 第一讲 基本概念

## 什么是数据结构

官方统一定义——

没有…… 

“数据结构是数据对象，以及存在于该对象的实例和 组成实例的数据元素之间的各种联系。这些联系可以 通过定义相关的函数来给出。” 

Sartaj Sahni，《数据结构、算法与应用》 

“数据结构是ADT（抽象数据类型 Abstract Data Type）的物理实现。” 

Clifford A.Shaffer，《数据结构与算法分析》 

“数据结构（data structure）是计算机中存储、组织 数据的方式。通常情况下，精心选择的数据结构可以 带来最优效率的算法。” 

中文维基百科

解决问题方法的效率， 跟数据的组织方式有

解决问题方法的效率， 跟空间的利用效率有关

解决问题方法的效率， 跟算法的巧妙程度有关

clock()：捕捉从程序开始运行到clock()被调用时所耗费的时间。这个 时间单位是clock tick，即“时钟打点”。 常数CLK\_TCK(或CLOCKS\_PER\_SEC)：机器时钟每秒所走的时钟打点数

所以到底什么是数据结构？？？ 

数据对象在计算机中的组织方式 

逻辑结构 

物理存储结构 

数据对象必定与一系列加在其上的操作相关联 

完成这些操作所用的方法就是算法

数据类型 

数据对象集 

数据集合相关联的操作集 

抽象：描述数据类型的方法不依赖于具体实现 

与存放数据的机器无关 

与数据存储的物理结构无关 

与实现操作的算法和编程语言均无关

只描述数据对象集和相关操作集 “是什么 ”，并不涉及 “如何做到 ”的问题

## 什么是算法

定义 

算法（Algorithm） 

一个有限指令集 

接受一些输入（有些情况下不需要输入） 

产生输出 

一定在有限步骤之后终止 

每一条指令必须 

有充分明确的目标，不可以有歧义 

计算机能处理的范围之内 

描述应不依赖于任何一种计算机语言以及具体的实现手段

什么是好的算法？ 

空间复杂度 S ( n ) —— 根据算法写成的程序在执行时 占用存储单元的长度。这个长度往往与输入数据的 规模有关。空间复杂度过高的算法可能导致使用的 内存超限，造成程序非正常中断。 

时间复杂度 T( n ) —— 根据算法写成的程序在执行时 耗费时间的长度。这个长度往往也与输入数据的规 模有关。时间复杂度过高的低效算法可能导致我们在有生之年都等不到运行结果。

在分析一般算法的效率时，我们经常关注下面 两种复杂度 

最坏情况复杂度 Tworst( n ) 

平均复杂度 Tavg( n ) Tavg( n )  Tworst( n )

复杂度的渐进表示法

T( n) = O(f( n)) 表示存在常数C>0, n 0>0 使得当 n >=n 0 时有 T( n) <= C·f( n ) 

T( n) = Ω (g ( n)) 表示存在常数C >0, n 0>0 使得当n>=n0时有 T( n)>=C·g ( n ) 

T(n)=Θ(h(n)) 表示同时有 T(n)=O(h(n))和T(n)= Ω(h(n))

复杂度分析小窍门 

若两段算法分别有复杂度 T1(n)=O(f1(n)) 和 T2 ( n) = O(f2(n))，则 

T1 ( n) + T2 ( n) = max( O(f1( n)), O(f2(n)) ) 

T1 ( n)\*T2 ( n) = O(f1(n)\*f2(n) ) 

若T(n)是关于n的 k阶多项式,那么 T(n)=Θ(nk ) 

一个for循环的时间复杂度等于循环次数乘以循环体代码的复杂度 

if-else 结构的复杂度取决于if的条件判断复杂度和两个分枝部分的复杂度，总体复杂度取三者中最大

## 应用实例： 最大子列和问题

给定N个整数的序列{ A1, A2, …, AN}，求函数的最大值。

### 算法1

int MaxSubseqSum1( int A[], int N )

{ int ThisSum, MaxSum = 0;

int i, j, k;

for( i = 0; i < N; i++ ) { /\* i是子列左端位置 \*/

for( j = i; j < N; j++ ) { /\* j是子列右端位置 \*/

ThisSum = 0; /\* ThisSum是从A[i]到A[j]的子列和 \*/

for( k = i; k <= j; k++ )

ThisSum += A[k];

if( ThisSum > MaxSum ) /\* 如果刚得到的这个子列和更大 \*/

MaxSum = ThisSum; /\* 则更新结果 \*/

} /\* j循环结束 \*/

} /\* i循环结束 \*/

return MaxSum;

} T( N ) = O( N3 )

### 算法2

int MaxSubseqSum2( int A[], int N ) {

int ThisSum, MaxSum = 0;

int i, j;

for( i = 0; i < N; i++ ) { /\* i是子列左端位置 \*/

ThisSum = 0; /\* ThisSum是从A[i]到A[j]的子列和 \*/

for( j = i; j < N; j++ ) { /\* j是子列右端位置 \*/

ThisSum += A[j];

/\*对于相同的i，不同的j，只要在j-1次循环的基础上累加1项即可\*/

if( ThisSum > MaxSum ) /\* 如果刚得到的这个子列和更大 \*/

MaxSum = ThisSum; /\* 则更新结果 \*/

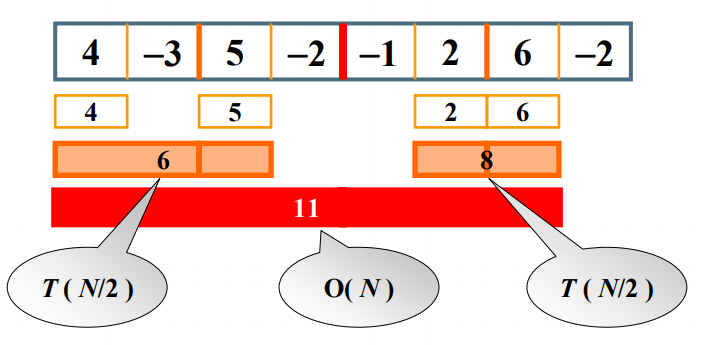
} /\* j循环结束 \*/

} /\* i循环结束 \*/

return MaxSum;

} T( N ) = O( N2 )

### 算法3：分而治之



T(N) = 2T(N/2)+c N , T(1) = O(1)

= 2 [2 T( N/22 ) + c N/2] + c N

= 2k O(1)+c k N 其中 N/2k = 1

= O( N log N )

int Max3( int A, int B, int C ) {

/\* 返回3个整数中的最大值 \*/

return A > B ? A > C ? A : C : B > C ? B : C;

}

int DivideAndConquer( int List[], int left, int right ) {

/\* 分治法求List[left]到List[right]的最大子列和\*/

     int MaxLeftSum, MaxRightSum; /\* 存放左右子问题的解 \*/

 int MaxLeftBorderSum, MaxRightBorderSum; /\*存放跨分界线的结果\*/

int LeftBorderSum, RightBorderSum;     int center, i;

if( left == right )  { /\* 递归的终止条件，子列只有1个数字 \*/

if( List[left] > 0 )  return List[left];

else return 0;

 }

/\* 下面是"分"的过程 \*/

center = ( left + right ) / 2; /\* 找到中分点 \*/

 /\* 递归求得两边子列的最大和 \*/

MaxLeftSum = DivideAndConquer( List, left, center );

MaxRightSum = DivideAndConquer( List, center+1, right );

/\* 下面求跨分界线的最大子列和 \*/

MaxLeftBorderSum = 0; LeftBorderSum = 0;

 for( i=center; i>=left; i-- ) { /\* 从中线向左扫描 \*/

LeftBorderSum += List[i];

if( LeftBorderSum > MaxLeftBorderSum )

MaxLeftBorderSum = LeftBorderSum;

} /\* 左边扫描结束 \*/

 MaxRightBorderSum = 0; RightBorderSum = 0;

for( i=center+1; i<=right; i++ ) { /\* 从中线向右扫描 \*/

RightBorderSum += List[i];

 if( RightBorderSum > MaxRightBorderSum )

MaxRightBorderSum = RightBorderSum;

} /\* 右边扫描结束 \*/

 /\* 下面返回"治"的结果 \*/

return Max3( MaxLeftSum, MaxRightSum, MaxLeftBorderSum + MaxRightBorderSum );

}

int MaxSubseqSum3( int List[], int N ) {

/\* 保持与前2种算法相同的函数接口 \*/

return DivideAndConquer( List, 0, N-1 );

}

### 算法4：在线处理

int MaxSubseqSum4( int A[], int N ) {

int ThisSum, MaxSum;

int i;

ThisSum = MaxSum = 0;

for( i = 0; i < N; i++ ) {

ThisSum += A[i]; /\* 向右累加 \*/

if( ThisSum > MaxSum )

MaxSum = ThisSum; /\* 发现更大和则更新当前结果 \*/

else if( ThisSum < 0 ) /\* 如果当前子列和为负 \*/

ThisSum = 0; /\* 则不可能使后面的部分和增大，抛弃之 \*/

} return MaxSum;

} T( N ) = O( N )

“在线”的意思是指每输入一个数据就进行即时处理，在任 何一个地方中止输入，算法都能正确给出当前的解。 

# 第二节 线性结构

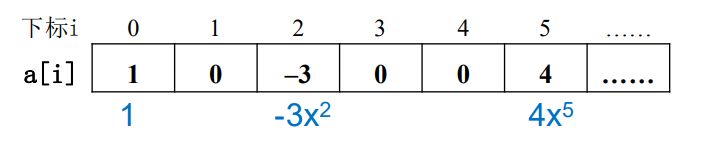
## 2.1 线性表及其实现

### 方法1：顺序存储结构直接表示

数组各分量对应多项式各项：

a[i]:项xi的系数ai

例如:f(x) = 4x5-3x2+1

表示成： 

两个多项式相加： 两个数组对应分量相加

问题：如何表示多项式 x+3x2000 ?

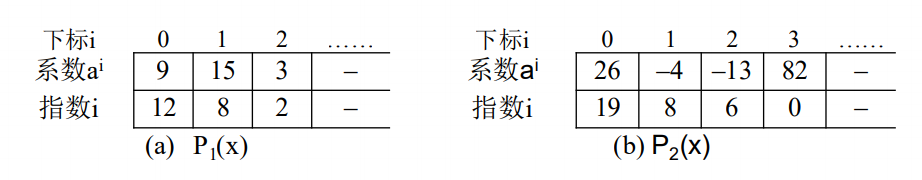
### 方法2：顺序存储结构表示非零项

每个非零项aixi涉及两个信息：系数ai和指数i

可以将一个多项式看成是一个 (ai,i) 二元组的集合。

用结构数组表示：数组分量是由系数ai、指数i组成的结构，对应一个非零项

例如： P1(x)=9x12+15x8+3x2 和P2(x)=26x19-4x8-13x6+82



按指数大小有序存储！

相加过程：从头开始，比较两个多项式当前对应项的指数

P1: (9,12), (15,8), (3,2)

P2: (26,19), (-4,8), (-13,6), (82,0)

P3: (26,19) (9,12) (11,8) (-13,6) (3,2) (82,0)

P3(x)=26x19+9x12-11x8-13x6+3x2+82

### 方法3：链表结构存储非零项

链表中每个结点存储多项式中的一个非零项，包括系数和指数两个数据域以及一个指针域

typedef struct PolyNode \*Polynomial;

struct PolyNode {

int coef;

int expon;

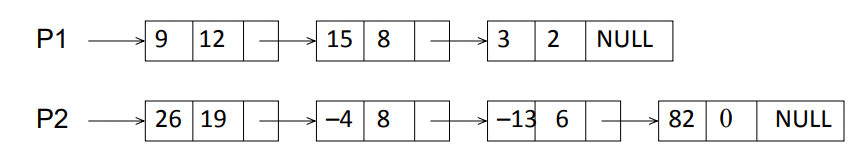
Polynomial link;}

例如:

P1(x)=9x12+15x8+3x2

P2(x)=26x19-4x8-13x6+82

链表存储形式为：



### 什么是线性表

“线性表(Linear List)”：由同类型数据元素构成有序序列的线性结构 

表中元素个数称为线性表的长度 

线性表没有元素时，称为空表 

表起始位置称表头，表结束位置称表尾

多项式表示问题的启示：

1.同一个问题可以有不同的表示（存储）方法

2.有一类共性问题：有序线性序列的组织和管理

### 线性表的抽象数据类型描述

类型名称：线性表（List）

数据对象集：线性表是 n (≥0)个元素构成的有序序列( a1 , a2 , ,an )

操作集：线性表L∈List,整数i表示位置，元素X∈ElementType,

线性表基本操作主要有：

1.List MakeEmpty()：初始化一个空线性表L；

2.ElementType FindKth( int K, List L )：根据位序K，返回相应元素 ；

3.int Find( ElementType X, List L )：在线性表L中查找X的第一次出现位置；

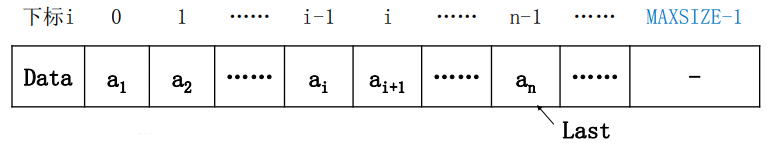
4.void Insert( ElementType X, int i, List L)：在位序i前插入一个新元素X；

5.void Delete( int i, List L )：删除指定位序i的元素；

6.int Length( List L )：返回线性表L的长度n。

### 线性表的顺序存储实现

利用数组的连续存储空间顺序存放线性表的各元素



typedef int Position;

typedef struct LNode \*List;

define MAXSIZE 50

struct LNode{

ElementType Data[MAXSIZE];

int Last; } ;

/\*

\*struct LNode L;

\*List PtrL;

\*访问下标为 i 的元素：L.Data[i] 或 PtrL->Data[i]

\*线性表的长度：L.Last+1 或 PtrL->Last+1

\*/

实现：

typedef int Position;

typedef struct LNode \*List;

struct LNode {

     ElementType Data[MAXSIZE];

     Position Last;

};

/\* 初始化 \*/

List MakeEmpty() {

     List L;

     L = (List)malloc(sizeof(struct LNode));

     L->Last = -1;

     return L;

}

/\* 查找 \*/

#define ERROR -1

Position Find( List L, ElementType X ) {

      Position i = 0;

     while( i <= L->Last && L->Data[i]!= X )

         i++;

     if ( i > L->Last )

return ERROR; /\* 如果没找到，返回错误信息 \*/

     else

return i;  /\* 找到后返回的是存储位置 \*/

}

/\* 插入 \*/

/\*是P存储下标位置（从0开始）\*/

bool Insert( List L, ElementType X, Position P )  {

/\* 在L的指定位置P前插入一个新元素X \*/

     Position i;

     if ( L->Last == MAXSIZE-1) {

         /\* 表空间已满，不能插入 \*/

         printf("表满");

           return false;

       }

      if ( P<0 || P>L->Last+1 ) { /\* 检查插入位置的合法性 \*/

         printf("位置不合法");

         return false;      }

     for( i=L->Last; i>=P; i-- )

        L->Data[i+1] = L->Data[i]; /\* 将位置P及以后的元素顺序向后移动 \*/

    L->Data[P] = X;  /\* 新元素插入 \*/

    L->Last++;       /\* Last仍指向最后元素 \*/

    return true;

}

/\* 删除 \*/

bool Delete( List L, Position P ) {

/\* 从L中删除指定位置P的元素 \*/

     Position i;

     if( P<0 || P>L->Last ) { /\* 检查空表及删除位置的合法性 \*/

         printf("位置%d不存在元素", P );

         return false;

       }

     for( i=P+1; i<=L->Last; i++ )

         L->Data[i-1] = L->Data[i]; /\* 将位置P+1及以后的元素顺序向前移动 \*/

     L->Last--; /\* Last仍指向最后元素 \*/

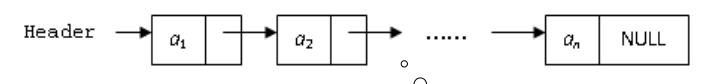
     return true;

}

### 线性表的链式存储实现

不要求逻辑上相邻的两个元素物理上也相邻；通过“链”建 立起数据元素之间的逻辑关系。

• 插入、删除不需要移动数据元素，只需要修改“链”。



实现：

typedef struct LNode \*List;

struct LNode{

ElementType Data;

List Next; };

struct Lnode L;

List PtrL;

实现：

typedef struct LNode \*PtrToLNode;

struct LNode {

    ElementType Data;

  PtrToLNode Next; };

typedef PtrToLNode Position;

typedef PtrToLNode List;

/\* 查找 \*/

#define ERROR NULL

Position Find( List L, ElementType X ) {

     Position p = L; /\* p指向L的第1个结点 \*/

while ( p && p->Data!=X )

       p = p->Next;

    /\* 下列语句可以用 return p; 替换 \*/

 if ( p )

         return p;

     else

         return ERROR;

}

/\* 带头结点的插入 \*/

/\*注意:在插入位置参数P上与课程视频有所不同，课程视频中i是序列位序（从1开始），这里P是链表结点指针，在P之前插入新结点 \*/

bool Insert( List L, ElementType X, Position P ) {

/\* 这里默认L有头结点 \*/

     Position tmp, pre;     /\* 查找P的前一个结点 \*/

     for ( pre=L; pre&&pre->Next!=P; pre=pre->Next ) ;

if ( pre==NULL ) { /\* P所指的结点不在L中 \*/

         printf("插入位置参数错误\n");

          return false;

}else { /\* 找到了P的前一个结点pre \*/

  /\* 在P前插入新结点 \*/

   tmp = (Position)malloc(sizeof(struct LNode)); /\* 申请、填装结点 \*/

    tmp->Data = X;

    tmp->Next = P;

pre->Next = tmp;

    return true;

     }

}

/\* 带头结点的删除 \*/

/\*注意:在删除位置参数P上与课程视频有所不同，课程视频中i是序列位序（从1开始），这里P是拟删除结点指针 \*/

bool Delete( List L, Position P ) {

/\* 这里默认L有头结点 \*/

  Position pre;     /\* 查找P的前一个结点 \*/

    for ( pre=L; pre&&pre->Next!=P; pre=pre->Next ) ;

   if ( pre==NULL || P==NULL) { /\* P所指的结点不在L中 \*/

         printf("删除位置参数错误\n");

         return false;

 } else { /\* 找到了P的前一个结点pre \*/

  /\* 将P位置的结点删除 \*/

         pre->Next = P->Next;

         free(P);

         return true;

     }

}

### 广义表

广义表(Generalized List) 

广义表是线性表的推广 

对于线性表而言，n个元素都是基本的单元素； 

广义表中，这些元素不仅可以是单元素也可以是另一个广义表。

typedef struct GNode \*GList;

struct GNode{

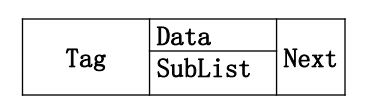
int Tag; /\*标志域：0表示结点是单元素，1表示结点是广义表 \*/

union { /\*子表指针域Sublist与单元素数据域Data复用，即共用存储空间\*/

ElementType Data;

GList SubList; } URegion;

GList Next; /\* 指向后继结点 \*/ };



### 多重链表

多重链表：链表中的节点可能同时隶属于多个链 

多重链表中结点的指针域会有多个，如前面例子包含了Next和 SubList两个指针域； 

但包含两个指针域的链表并不一定是多重链表，比如在双向链表 不是多重链表。 

多重链表有广泛的用途：

基本上如树、图这样相对 复杂的数据结构都可以采用多重链表方式实现存储。

## 2.2 堆栈

### 什么是堆栈

#### 后缀表达式

中缀表达式：运算符号位于两个运算数之间。如 ，a+b\*c-d/e 

后缀表达式：运算符号位于两个运算数之后。如， a\*b\*c+d\*e 

后缀表达式求值策略：从左向右“扫描”，逐个处理运算数和运算符号

1. 遇到运算数怎么办？如何“记住”目前还不未参与运算的数？
2. 遇到运算符号怎么办？对应的运算数是什么？

启示：需要有种存储方法，能顺序存储运算数， 并在需要时“倒序”输出！

#### 堆栈的抽象数据类型描述

堆栈（Stack）：具有一定操作约束的线性表

只在一端（栈顶，Top）做 插入、删除

插入数据：入栈（Push） 

删除数据：出栈（Pop） 

后入先出：Last In First Out（LIFO）

类型名称: 堆栈（Stack）

数据对象集：一个有0个或多个元素的有穷线性表。

操作集：长度为MaxSize的堆栈S∈Stack，堆栈元素item∈ElementType

1. Stack CreateStack( int MaxSize )： 生成空堆栈，其最大长度为MaxSize；
2. int IsFull( Stack S, int MaxSize )：判断堆栈S是否已满；
3. void Push( Stack S, ElementType item )：将元素item压入堆栈；
4. int IsEmpty ( Stack S )：判断堆栈S是否为空；
5. ElementType Pop( Stack S )：删除并返回栈顶元素；

### 栈的顺序存储实现

栈的顺序存储结构通常由一个一维数组和一个记录 栈顶元素位置的变量组成。

#define MaxSize <储存数据元素的最大个数>

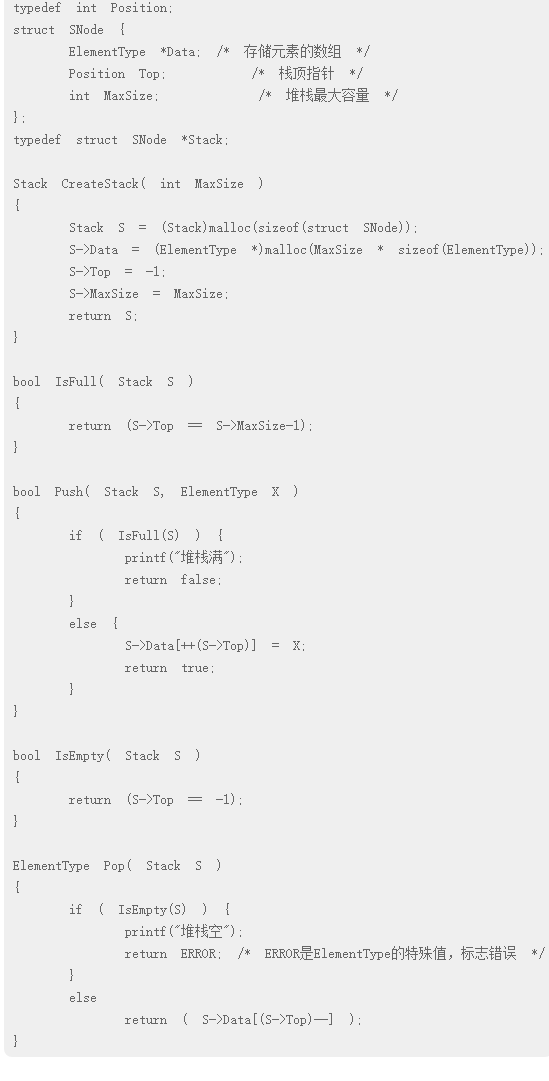
typedef struct SNode \*Stack;

struct SNode{

ElementType Data[MaxSize];

int Top;

};



### 栈的链式存储实现

栈的链式存储结构实际上就是一个单链表，叫做链栈。插入和删 除操作只能在链栈的栈顶进行。栈顶指针Top应该在链表的哪一头？

typedef struct SNode \*Stack;

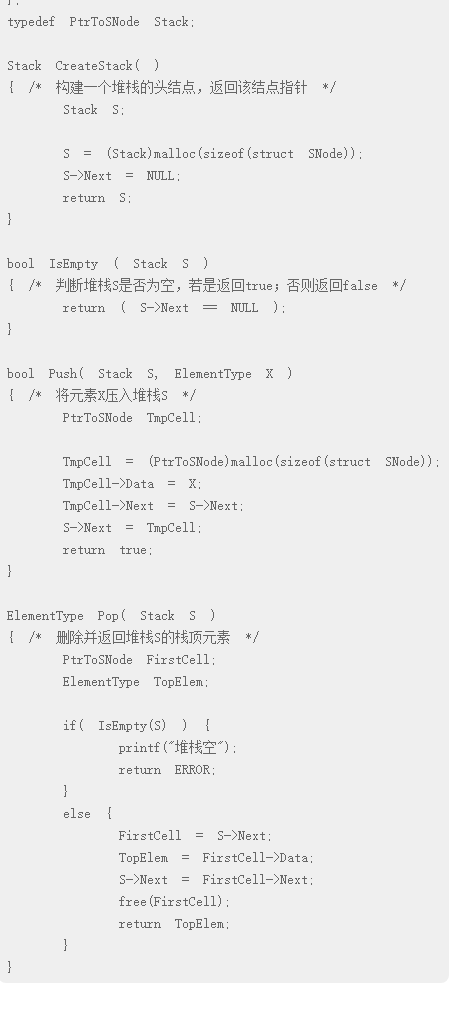
struct SNode{

ElementType Data;

struct SNode \*Next; } ;

(1)堆栈初始化（建立空栈）

(2) 判断堆栈S是否为空



## 2.3 队列及实现

### 什么是队列

队列(Queue)：具有一定操作约束的线性表 

插入和删除操作：只能在一端插入，而在另一端删除。

数据插入：入队列（AddQ） 

数据删除：出队列（DeleteQ） 

先来先服务 

先进先出：FIFO

### 队列的抽象数据类型描述

类型名称：队列(Queue)

数据对象集：一个有0个或多个元素的有穷线性表。

操作集：长度为MaxSize的队列Q  Queue，队列元素item  ElementType

1. Queue CreatQueue( int MaxSize )：生成长度为MaxSize的空队列；
2. int IsFullQ( Queue Q, int MaxSize )：判断队列Q是否已满；
3. void AddQ( Queue Q, ElementType item )： 将数据元素item插入队列Q中；

4、int IsEmptyQ( Queue Q )： 判断队列Q是否为空；

5、ElementType DeleteQ( Queue Q )：将队头数据元素从队列中删除并返回。

### 队列的顺序存储实现

队列的顺序存储结构通常由一个一维数组和一个记录队列头元素位置的变量front以及一个记录队列尾元素位置的变量rear组成

#define MaxSize <储存数据元素的最大个数>

struct QNode {

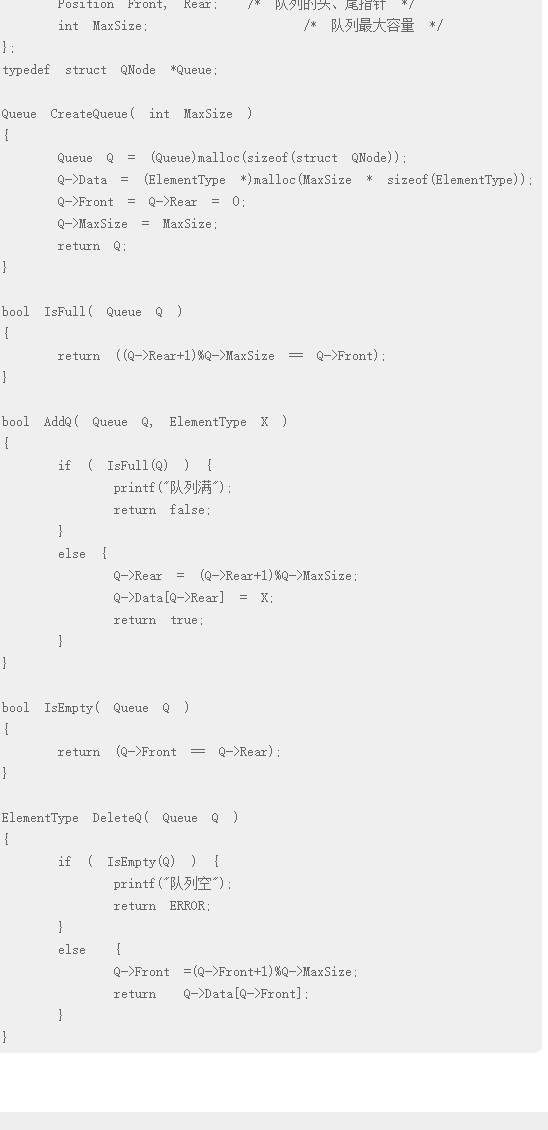
ElementType Data[ MaxSize ];

int rear;

int front;

};

typedef struct QNode \*Queue;



### 队列的链式存储实现

队列的链式存储结构也可以用一个单链表实现。插入和删除操作 分别在链表的两头进行；队列指针front和rear应该分别指向链表的哪一头？

struct Node{

ElementType Data;

struct Node \*Next; };

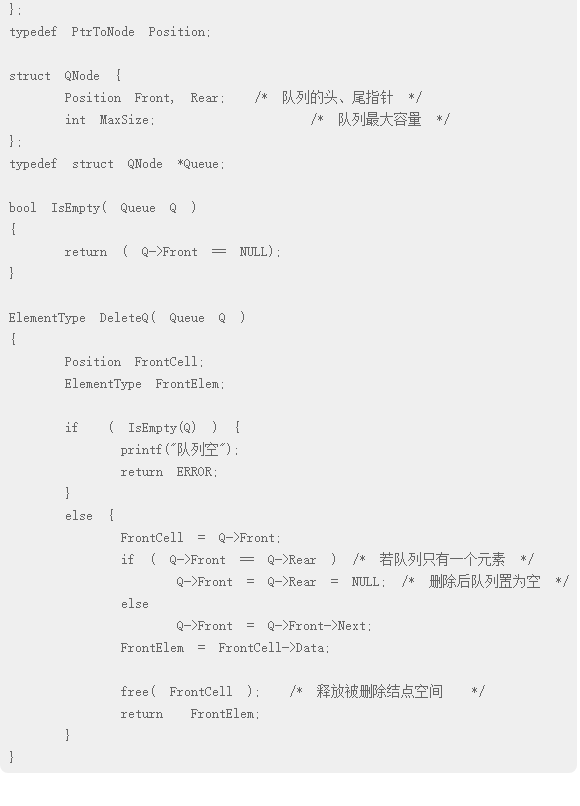
struct QNode{ /\* 链队列结构 \*/

struct Node \*rear; /\* 指向队尾结点 \*/

struct Node \*front; /\* 指向队头结点 \*/ };

typedef struct QNode \*Queue;

Queue PtrQ;

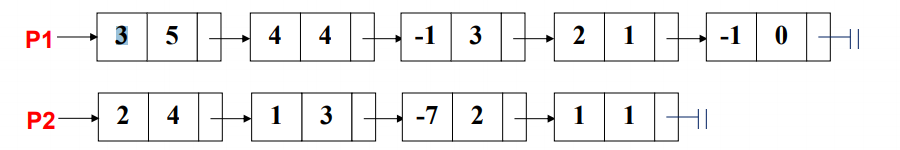


## 2.4 应用: 多项式加法运算

主要思路：相同指数的项系数相加，其余部分进行拷贝。

### 多项式加法运算

采用不带头结点的单向链表，按照指数递减的顺序排列各项



struct PolyNode {

int coef; // 系数

int expon; // 指数

struct PolyNode \*link; // 指向下一个节点的指针 };

typedef struct PolyNode \*Polynomial;

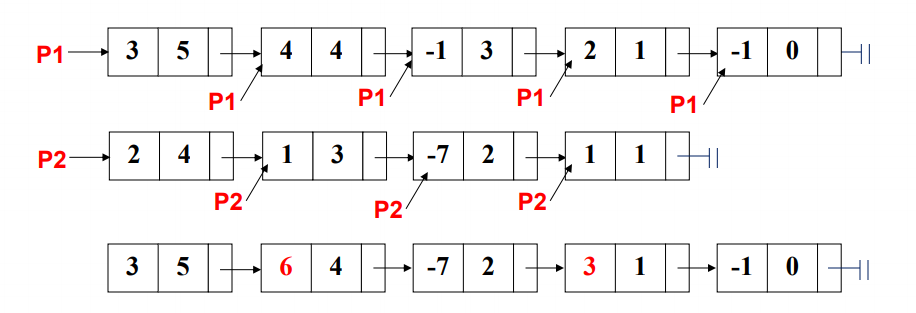
Polynomial P1, P2;

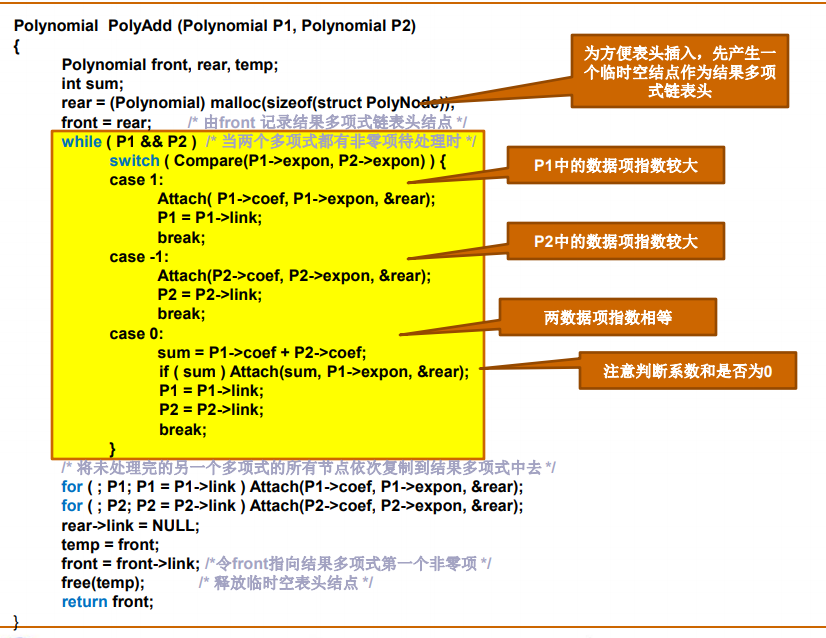
算法思路：两个指针P1和P2分别指向这两个多项式第一个结点，不断循环：

P1->expon==P2->expon: 系数相加，若结果不为0，则作为结果多项式对应项 的系数。同时，P1和P2都分别指向下一项； 

P1->expon>P2->expon: 将P1的当前项存入结果多项式，并使P1指向下一项； 

P1->exponexpon: 将P2的当前项存入结果多项式，并使P2指向下一项； 当某一多项式处理完时，将另一个多项式的所有结点依次复制到结果多项式中去





void Attach( int c, int e, Polynomial \*pRear ) {

/\* 由于在本函数中需要改变当前结果表达式尾项指针的值， \*/

/\* 所以函数传递进来的是结点指针的地址，\*pRear指向尾项\*/

Polynomial P;

P =(Polynomial)malloc(sizeof(struct PolyNode)); /\* 申请新结点 \*/

1. >coef = c; /\* 对新结点赋值 \*/

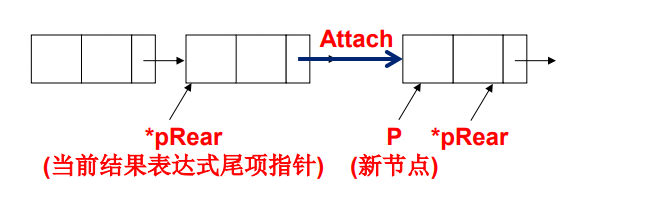
P->expon = e;

P->link=NULL; /\* 将P指向的新结点插入到当前结果表达式尾项的后面 \*/

(\*pRear)->link = P;

\*pRear = P; /\* 修改pRear值 \*/

}



# 树（上）

## 3.1 树与树的表示

查找：根据某个给定关键字K ，从集合R中找出关键字与K相同的记录

静态查找：集合中记录是固定的没有插入和删除操作，只有查找

动态查找：集合中记录是动态变化的除查找，还可能发生插入和删除

### 树的定义

树（Tree）: n（n≥0）个结点构成的有限集合。

当n=0时，称为空树；

对于任一棵非空树（n> 0），它具备以下性质： 

树中有一个称为“根（Root）”的特殊结点，用 r 表示； 

其余结点可分为m(m>0)个互不相交的有限集T1，T2，... ，Tm，其 中每个集合本身又是一棵树，称为原来树的“子树（SubTree）”

### 树与非树?

子树是不相交的； 

除了根结点外，每个结点有且仅有一个父结点； 

一棵N个结点的树有N-1条边。

### 树的一些基本术语

1. 结点的度（Degree）：结点的子树个数
2. 树的度：树的所有结点中最大的度数

3 叶结点（Leaf）：度为0的结点

4.父结点（Parent）：有子树的结点是其子树 的根结点的父结点

5. 子结点（Child）：若A结点是B结点的父结 点，则称B结点是A结点的子结点；子结点也 称孩子结点。

6. 兄弟结点（Sibling）：具有同一父结点的各 结点彼此是兄弟结点。

7. 路径和路径长度：从结点n1到nk的路径为一 个结点序列n1 , n2 ,… , nk , ni是 ni+1的父结 点。路径所包含边的个数为路径的长度。

8. 祖先结点(Ancestor)：沿树根到某一结点路 径上的所有结点都是这个结点的祖先结点。

9. 子孙结点(Descendant)：某一结点的子树 中的所有结点是这个结点的子孙。

10. 结点的层次（Level）：规定根结点在1层， 其它任一结点的层数是其父结点的层数加1。

12. 树的深度（Depth）：树中所有结点中的最 大层次是这棵树的深度。

### 二叉树的链表结构（儿子--兄弟表示法）

typedef struct TNode \*Position;

typedef Position BinTree; /\* 二叉树类型 \*/

struct TNode{ /\* 树结点定义 \*/

    ElementType Data; /\* 结点数据 \*/

    BinTree Left;     /\* 指向左子树 \*/

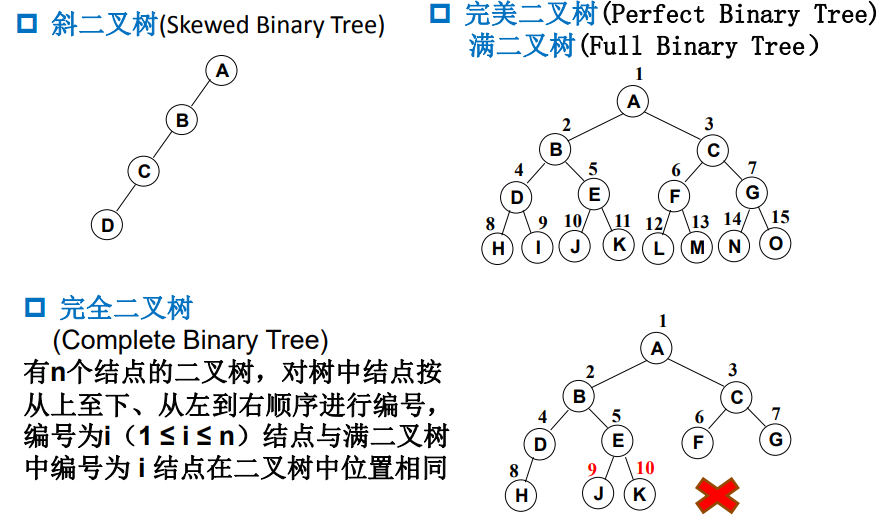
    BinTree Right;    /\* 指向右子树 \*/ };

## 3.2二叉树及其储存结构

### 二叉树的定义

二叉树T：一个有穷的结点集合。 这个集合可以为空 若不为空，则它是由根结点和称为其左子树TL和右子树TR的 两个不相交的二叉树组成。

### 特殊二叉树



二叉树几个重要性质 

一个二叉树第 i 层的最大结点数为：2 i-1 ，i>=1。

深度为k的二叉树有最大结点总数为： 2 k -1，k>=1 

对任何非空二叉树 T，若n0表示叶结点的个数、n2是度为2的非叶结点个数，那么两者满足关系n0 = n2 +1。

二叉树的抽象数据类型定义

类型名称：二叉树

数据对象集：一个有穷的结点集合。若不为空，则由根结点和其左、右二叉子树组成。

操作集： BT∈BinTree, Item∈ElementType，重要操作有：

1、Boolean IsEmpty( BinTree BT )： 判别BT是否为空；

1. void Traversal( BinTree BT )：遍历，按某顺序访问每个结点；
2. BinTree CreatBinTree( )：创建一个二叉树。

常用的遍历方法有： 

void PreOrderTraversal( BinTree BT )：先序----根、左子树、右子树；

void InOrderTraversal( BinTree BT )： 中序---左子树、根、右子树； 

void PostOrderTraversal( BinTree BT )：后序---左子树、右子树、根 

void LevelOrderTraversal( BinTree BT )：层次遍历，从上到下、从左到右

### 二叉树的存储结构

#### 顺序存储结构

完全二叉树：

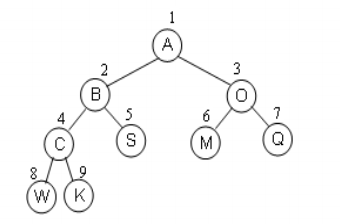
按从上至下、从左到右顺序存储 n个结点的完全二叉树的结点父子关系：  非根结点（序号 i > 1）的父结点的序号是 i / 2; 

结点（序号为 i ）的左孩子结点的序号是 2i，

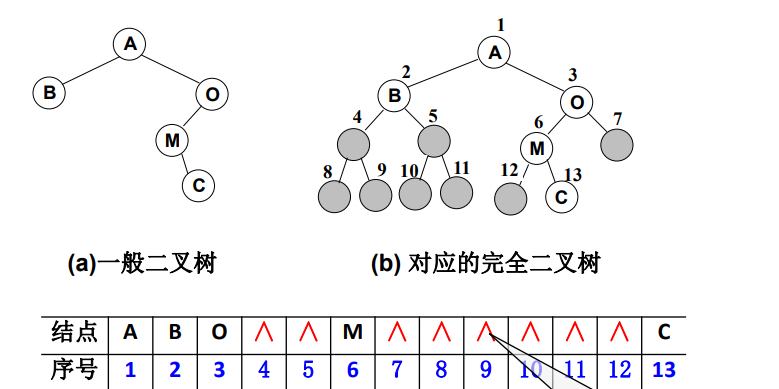
(若2 i <= n，否则没有左孩子）; 

结点（序号为 i ）的右孩子结点的序号是 2i+1，

(若2 i +1<= n，否则没有右孩子）;



一般二叉树也可以采用这种结构，但会造成空间浪费……



#### 2. 链表存储

typedef struct TreeNode \*BinTree;

typedef BinTree Position;

struct TreeNode{

ElementType Data;

BinTree Left;

BinTree Right;

}