**数据结构与算法笔记**

**目录**

[**第一章 绪论 3**](#_Toc19626)

[**1. 程序的本质 3**](#_Toc24436)

[**2. 数据、数据元素、数据项和数据对象 3**](#_Toc24694)

[**3. 数据结构 3**](#_Toc23300)

[**4. 逻辑结构的种类（两种划分方法） 4**](#_Toc13710)

[**5. 存储结构的种类 4**](#_Toc16004)

[**6. 数据类型 4**](#_Toc13398)

[**7. 抽象数据类型（Abstract Data Type,DAT） 5**](#_Toc20849)

[**8. 算法 6**](#_Toc11353)

[**9. 算法分析 6**](#_Toc15146)

[**10. 算法的效率 7**](#_Toc19852)

[**11. 时间效率的度量 7**](#_Toc16789)

[**12. 时间复杂度T(n) 7**](#_Toc27874)

[**13. 空间复杂度S(n) 9**](#_Toc31219)

[**第二章 线性表 10**](#_Toc30275)

[**1. 定义 10**](#_Toc25976)

[**2. 逻辑特征 10**](#_Toc16221)

[**3. 图示 10**](#_Toc32595)

[**4. 线性表的例子 10**](#_Toc18928)

[**5. 线性表的存储结构（顺序存储结构和链式存储结构） 10**](#_Toc12366)

[**（1） 顺序存储结构 10**](#_Toc14720)

[**（2） 链式存储结构 13**](#_Toc31800)

[**（3） 单链表 14**](#_Toc8083)

[**（4） 循环链表 17**](#_Toc8105)

[**（5） 双向链表 17**](#_Toc6885)

[**（6） 顺序表与线性表的比较 18**](#_Toc18661)

[**第三章 栈和队列 19**](#_Toc26938)

[**1. 栈的定义 19**](#_Toc11502)

[**2. 栈的顺序存储结构 19**](#_Toc18608)

[**3. 栈的链式存储结构(栈顶放在单链表的头部且不需要头结点) 21**](#_Toc510)

[**4. 栈的应用：递归 22**](#_Toc11462)

[**5. 栈的应用：四则运算表达式求值 （如计算“9+（3+1）\*3+10/2”的值） 22**](#_Toc29445)

[**6. 队列的定义 23**](#_Toc11377)

[**7. 队列的顺序存储结构（循环队列） 23**](#_Toc31711)

[**8. 队列的链式存储结构（链队列） 24**](#_Toc31217)

[**第四章 串 26**](#_Toc23637)

[**1. 串的定义 26**](#_Toc31308)

[**2. 基础概念 26**](#_Toc50)

[**3. 串的比较 26**](#_Toc400)

[**4. 查找字符串：（朴素模式匹配） 26**](#_Toc27888)

[**5. 串的顺序存储结构 27**](#_Toc8791)

[**6. 串的链式存储结构 27**](#_Toc2497)

[**7. KMP模式匹配算法 27**](#_Toc12822)

[**8. KMP算法改进 29**](#_Toc32010)

[**第五章 树 31**](#_Toc28995)

[**1. 树的定义 31**](#_Toc29320)

[**2. 树的基本术语 31**](#_Toc32090)

[**3. 结点间的关系 32**](#_Toc30640)

笔记内容：第一部分：数据结构的基本概念

第二部分：基本的数据结构：①线性结构：线性表、栈和队列、串

②非线性结构：树、图

第三部分：基本的数据处理技术：查找技术、排序技术

（数据结构是一门研究非数值运算的程序设计问题中的操作对象，以及它们之间的关系和操作等相关问题的学科）

1. 绪论
2. **程序的本质**

程序 = 数据结构 + 算法

1. **数据、数据元素、数据项和数据对象**

①**数据**：能输入计算机且能被计算机处理的各种符号的集合，是信息的载体，是对客观 事物符号化的表示，能够被计算机识别、存储和加工。

（如：数值、文字、图像、声音等。）

②**数据元素**：是数据的基本单位，在计算机中通常作为一个整体考虑和处理。也称为元 素、记录、节点或顶点。一个数据元素可由若干个数据项组成。

③**数据项**：构成数据元素的不可分割的最小单位。

④**数据对象**：是性质相同的数据元素的集合，是数据的子集。

（如：自然数数据对象：N={0，1，2，3，...};字母字符数据对象：C={‘A’,’B’,’C’,...}）

**注**：前三者的关系为： 数据 > 数据元素 > 数据项

（如：学生表是数据，则个人纪录是数据元素，学号、姓名、年龄等是数据项）

1. **数据结构**

①**定义**

数据元素不是孤立存在的，他们之间存在着某种关系，数据元素相互之间的关系称为数据结构。数据结构是指相互之间存在一种或多种特定关系的数据元素的集合，或者说数据结构是带结构的数据元素的集合。

②**分类**

**逻辑结构**：描述数据元素间的逻辑关系，与数据的存储无关，独立于计算机，是从具体 问题抽象出来的数学模型。

**物理结构（存储结构）**：数据元素及其关系在计算机存储器中的结构（存储方式），是数 据结构在计算机中的表示。

**注**：二者关系：存储结构是逻辑关系的映像与元素本身的映像，逻辑结构是数据结构的抽象， 存储结构是数据结构的实现，两者综合起来建立了数据元素之间的结构关系。

1. **逻辑结构的种类（两种划分方法）**

①**线性结构**：有且仅有一个开始和一个终端结点，并且所有节点都最多只有一个直接前 驱和一个直接后继。（如：线性表、栈、队列和串）

**非线性结构**：一个节点可能有多个直接前驱和直接后继。（如：树、图）

②**集合结构**：结构中的数据元素之间除了同属于一个集合外，无任何其他关系。

**线性结构**：结构中的数据元素之间存在着一对一的线性关系。

**树形结构**：结构中的数据元素之间存在着一对多的层次关系。

**图/网状结构**：结构中的数据元素之间存在着多对多的任意关系。

1. **存储结构的种类**

①**顺序存储结构**：用一组连续的存储单元依次存储数据元素，数据元素之间的逻辑关系 由元素的存储位置来表示，C语言中用数组来实现顺序存储结构。

②**链接存储结构**：用一组任意的存储单元存储数据元素，数据元素之间的逻辑关系用指 针来表示，C语言中用指针来实现链式存储结构。

③**索引存储结构**：在存储结点信息的同时，还建立附加的索引表。索引表中的每一项成 为索引项，索引项的一般形式为：（关键字，地址），关键字是能唯一标识一个结点 的那些数据项。（如通讯录中的姓名索引）

④**散列存储结构**：根据结点的关键字直接计算出该结点的存储地址。（如散列表）

1. **数据类型**

①**定义**：一组性质相同的值的集合以及定义于这个值集合上的一组操作的总称。

（即：数据类型 = 值的集合 + 值集合上的一组操作）

②**作用**：约束变量或常量的取值范围和操作。

（如：C语言中定义变量i为int类型，就是表示i是[min,max]范围的整数，在这 个整数集上进行+、-、\*、\、%等操作）

**注**：使用高级程序设计语言时，必须对程序中出现的每个变量、常量或表达式明确说明他们 所属的数据类型。如C语言中提供的int,char,float等基本数据类型，数组，结构体，共 用体等构造数据类型，还有指针类型等。

1. **抽象数据类型（Abstract Data Type,DAT）**

①**定义**：一个数学模型以及定义在此数学模型上的一组操作。其由用户定义，从问题抽 象出数据模型（逻辑结构），还包括定义在数据模型上的一组抽象运算（相关操作）， 不考虑计算机内的具体存储结构与运算的具体实现方法。

②**形式定义**：抽象数据类型可用三元组（D ,S ,P）表示。

（其中：D是数据对象；S是D上的关系集；P是对D的基本操作集。）

③**抽象数据类型的定义格式如下：**

ADT 抽象数据类型名{

数据对象<数据对象的定义>

数据关系<数据关系的定义>

基本操作<基本操作的定义>

}ADT 抽象数据类型名

**注**：其中，数据对象，数据关系的定义为伪代码描述。

④**基本操作的定义格式如下：**

基本操作名（参数表）

初始条件： <初始条件描述>

操作结果： <操作结果描述>

**注**：**参数表**：赋值参数，只为操作提供输入值。引用参数以&打头，除可提供输入值外，还将返回操作结果。

**初始条件**：描述操作执行之前数据结构和参数应满足的条件，若不满足，则操作失败，并返回相应出错信息。若初始条件为空，则省略之。

**操作结果**：说明操作正常完成之后数据结构的变化状况和返回的结果。

**举个例子**：Circle的定义

ADT Circle{

数据对象：D = {r,x,y|r,x,y均为实数}

数据关系：R = {<r,x,y>|r是半径，<x,y>是圆心坐标}

基本操作：Circle(&C,r,x,y)

操作结果：构造一个圆。

double Area(C)

初始条件：圆已存在。

操作结果：计算面积。

double Circumference(C)

初始条件：圆已存在。

操作结果：计算周长。

......

}ADT Circle

1. **算法**
2. **定义**

①**通俗的定义**：对待定问题求解方法和步骤的一种描述，它是指令的有限序列。其中， 每个指令表示一个或多个操作。简言之，就是解决问题的方法和步骤。

②**狭义的定义**：对存储数据的操作。

③**广义的定义**：广义的算法也叫泛型，即无论数据如何存储，对数据的操作都是一样的。

1. **算法的描述工具**

①**自然语言**：中文，英文等，...

②**流程图**：传统流程图，NS流程图，...

③**伪代码/类语言**：类C语言，...

④**程序代码**：C语言程序，JAVA语言程序，...

1. **算法的特性**

①**有穷性**：必须在执行有穷步后结束，且每步都在有穷时间内完成。

②**确定性**：每条指令必须有确切的含义，没有二义性，任何条件下只有唯一一条执行路 径。

③**可行性**：算法是可执行的，算法描述的操作可以通过已经实现的基本操作执行有限次 来实现。

④**输入/输出**：一个算法，有零个或多个输入，有一个或多个输出。

1. **算法设计的要求**

①**正确性**：程序对于精心选择的、典型的、苛刻且带有刁钻性的几组输入数据能够得出 满足要求的结果。

②**可读性**：算法应易于人的理解，晦涩难读的算法易于隐藏较多错误而难以调试。

③**健壮性**：当输入非法数据时，算法恰当的做出反应或进行相应处理，而不是产生莫名 其妙的输出结果。处理出错的方法不应是中断程序的执行，而应是返回一个表示错 误或错误性质的值，以便在更高的抽象层次上进行处理。

④**高效性**：花费尽量少的时间和尽量低的存储需求。

1. **算法分析**

目的是看算法实际是否可行，并在处理同一问题时存在较多算法的情况下进行性能上的比较，以便从中挑选出比较优的算法。

1. **算法的效率**

①**时间效率**：算法所消耗的时间

②**空间效率**：算法执行过程中所消耗的存储空间

1. **时间效率的度量**
2. **事后统计**：将算法实现，利用计算机计时器测算其时间和空间开销。（不采纳）

**缺陷**：①必须依据算法事先编好程序，需花费大量时间和精力。

②依赖计算机硬件和软件等因素，有时会掩盖算法本身的优劣。

③算法的测试数据设计困难，且程序运行时间往往与数据的规模有关。

1. **事前分析**：对算法所消耗资源的一种估算方法，把程序看成独立于程序设计语言的算 法或一系列步骤。
2. **事前分析方法**：算法运行时间 = 一个简单操作所需时间 简单操作次数，即算法中 每条语句的执行时间之和。每条语句的执行次数又称为语句频度，通常假设每条语 句的执行时间为单位时间。

如：for( i = 1; i <= n; i++ ) //执行n+1次

{

for( j = 1; j <= n; j++ ) //执行n(n+1)次

{

C[i][j] = 0; //执行n\*n次

for( k = 0; k < n; k ++ ) //执行n\*n\*(n+1)次

{

C[i][j] = c[i][j] + a[i][k] \* b[k][j]; //执行n\*n\*n次

}

}

}

则上述算法的时间消耗为：T(n) = 

**注**：判断一个算法的效率时，函数中的常数和其他次要项常常可以忽略，而更应关注最高阶 项的阶数。

1. **时间复杂度T(n)**
2. **定义**

为了方便比较不同算法的时间效率，通常仅比较它们的数量级。若有某个辅助函数f(n)， 当n趋近于无穷大时，T(n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函 数。记作T(n) =O(f(n))，称O(f(n))为算法的渐进时间复杂度（O是数量级符号），简称时间 复杂度。

1. **分析方法**

①找到语句频度最大的那条语句作为基本语句。

②计算基本语句的频度得到问题规模n的某个函数f(n)。

③取其数量级用符号“O”表示。

如：

x = 0; //1次

y = 0; //1次

for (int k = 0; k < n; k++) //执行n+1次

{

x++; //执行n次

}

for (int i = 0; i < n; i++) //执行n+1次

{

for (int j = 0; j < n; j++) //执行n(n+1)次

{

y++; //执行n×n次

}

}

则f(n) = n(n+1), T(n) = O()

1. **运算法则**：对于复杂的算法，可以将其分成几个容易估算的部分，然后利用O的加法法则和乘法法则计算时间复杂度。

①**加法规则**：T(n) = T1(n) + T2(n) = O(f(n)) + O(g(n)) = O(max(f(n), g(n)))

②**乘法规则**：T(n) = T1(n) × T2(n) = O(f(n)) × O(f(n)) = O(f(n) × g(n))

1. **常见的时间复杂度**

|  |  |  |
| --- | --- | --- |
| **执行次数函数** | **阶数** | **非正式术语** |
| 1 2 3 4 ...... | O(1) | 常数阶 |
| 2n+3 | O(n) | 线性阶 |
|  | O() | 平方阶 |
|  | O() | 对数阶 |
|  | O(n) | n阶 |
|  | O() | 立方阶 |
|  | O() | 指数阶 |

O(1) < O() < O(n) < O(n) < O() < O() < O() < O(n!) < O()

1. **最坏运行时间**

一种保证，即运行时间不会再坏了，是应用中一种最重要的需求。通常提到的运行时间都是最坏情况的运行时间。

1. **平均运行时间**

期望的运行时间，是所有情况中最有意义的。

1. **空间复杂度S(n)**
2. **定义**：算法所需存储空间的度量，记作：S(n) = O(f(n))。
3. **算法要占据的空间**

①算法本身要占据的空间：输入/输出、指令、常数和变量等。

②算法要使用的辅助空间。

**如**：将一维数组a中的n个数逆序放到原数组中

算法一：

for (i = 0; i < n / 2; i++) //空间复杂度为：S(n) = O(1) (原地工作)

{ //只使用一个辅助变量t

t = a[i];

a[i] = a[n - i - 1];

a[n - i - 1] = t;

}

算法二：

for (i = 0; i < n; i++) //空间复杂度为： S(n) = O(n)

{ //数组b需要和数组a一样大，使用n个空间

b[i] = a[n - i - 1];

}

for (i = 0; i < n; i++)

{

a[i] = b[i];

}

1. 线性表
2. **定义**

线性表是具有相同特性的数据元素的一个有限序列,是一种典型的线性结构。

1. **逻辑特征**

非空线性表中，有且仅有一个开始结点和一个终端结点。没有直接前趋，仅有一个直接后继；没有直接后继，仅有一个直接前趋；其余内部结点都仅有一个直接前趋和一个直接后继.

1. **图示**

（，，...，，，，...，）

1. **线性表的例子**

①**英文字母表**：（A, B, C, D, ......, Z）

②**十二星座**：（白羊座，金牛座，双子座，巨蟹座，狮子座，处女座，天枰座，天蝎座， 射手座，魔蝎座，水瓶座，双鱼座）

......

**注**：线性表中数据元素可以为简单类型，也可以为复杂类型。许多实际应用问题所涉及的基本操作有很大相似性（如对线性表进行增删查改等操作），不应为每个具体应用单独编写一个程序。从具体应用中抽象出共性的逻辑结构和基本操作（抽象数据类型），然后实现其存储结构和基本操作。

1. **线性表的存储结构**（顺序存储结构和链式存储结构）
2. **顺序存储结构**

①**定义**：把逻辑上相邻的数据元素存储在物理上相邻的存储单元中的存储结构。

②**特点**：线性表顺序存储结构占用一篇连续的存储空间，知道某个元素的存储位置就可以计算其他元素的存储位置，即任一元素可随机存取。（如C语言中的数组）

③**顺序表基本操作**：

常用设置：

#define OK 1

#define ERROR 0

#define TRUE 1

#define FALSE 0

typedef int Status; //Status 是函数类型，其值是函数结果状态代码，如OK等

1. **顺序存储的结构代码**

#define CAPACITY 20 //动态数组容量

typedef int ElemType; //ElemType类型根据实际情况而定，这里假设为int

typedef struct Sqlist

{

ElemType\* data; //创建一个动态数组指针

int length; //统计线性表当前长度

}SqList;

1. **顺序表L的初始化**

void SqListInit(SqList\* L)

{

int i = 0;

L->length = 0;

L->data = (ElemType\*)malloc(sizeof(ElemType)\*CAPACITY);

for (i = 0; i < 10; i++)

{

L->data[i] = i+1;

L->length++;

}

}

1. **顺序表L的取值(读取L中第i个元素)**

Status GetElem(SqList\* L, int i, ElemType\* e)

{

if (L->length == 0 || i<1 || i>L->length) //判断线性表是否为空，判断查找位

{ //置是否合法

return ERROR;

}

\*e = L->data[i - 1];

return OK;

}

1. **顺序表L的插入：(在L中第i个位置之前插入新的数据元素e,L的长度加一)**

Status ListInsert(SqList\* L, int i, ElemType e)

{

int k = 0;

if (L->length == CAPACITY) //判断线性表是否满了

{

return ERROR;

}

if (i<1 || i>L->length + 1) //判断插入位置是否合法

{

return ERROR;

}

if (i <= L->length) //若插入位置不在表尾

{

for (k = L->length - 1; k >= i - 1; k--) //插入位置后的元素向后移动一位

{

L->data[k + 1] = L->data[k];

}

}

L->data[i - 1] = e; //插入e

L->length++;

return OK;

}

1. **删除顺序表L中元素(删除L中第i个元素，并用e返回其值，L的长度减一)**

Status ListDelete(SqList\* L, int i, ElemType\* e)

{

int k = 0;

if (L->length == 0) //判断是否为空表

{

return ERROR;

}

if (i<1 || i>L->length) //判断删除位置是否合法

{

return ERROR;

}

\*e = L->data[i - 1];

if (i < L->length)

{

for (k = i; k < L->length; k++) //将删除位置后的元素迁移一位

{

L->data[k - 1] = L->data[k];

}

}

L->length--;

return OK;

}

④**查找算法平均查找长度（ASL）**

为确定记录在表中的位置，需要与给定值进行比较的关键字个数的期望值叫做查找算法的平均查找长度。（平均时间复杂度O（n））

如：在有7个元素的顺序表中进行查找，ASL = (1+2+3+4+5+6+7)/7 = 4

则：在有n个元素的顺序表中进行查找，ASL = (1+2+3+...+n)/n = (n+1)/2

⑤**插入算法平均移动次数（）**

若插入在尾结点则无需移动，若插入在首结点之前则全部元素均需后移，若插入第i个位置，则需移动n+1-i个元素，插入位置共有n+1种可能。

则：= (0+1+2+...+n) / (n+1) = n/2。（平均时间复杂度O（n））

⑥**删除算法平均移动次数（）**

若删除尾接结点则无需移动，若删除首结点则其余n-1个元素均需前移，若删除第i个位置，则需移动n - i个元素，删除位置共有n种可能。

则：= (0+1+2+...+(n-1)) / n = (n-1) / 2。（平均时间复杂度O（n））

1. **链式存储结构**

①**定义**

结点在存储器中的位置是任意的，即逻辑上相邻的数据元素在物理上不一定相邻的存储结构。

②**特点**

用一组物理位置任意的存储单元来存放线性表的数据元素，这组存储单元既可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上。访问时只能通过头指针进入链表，并通过每个节点的指针域依次向后顺序扫描其余结点，所以寻找第一个结点和最后一个结点所花费的时间不等。

③**相关术语**

1. **结点**：数据元素的存储映像，由数据域和指针域两部分组成。
2. **数据域**：存储元素数值数据的区域。
3. **指针域**：存储直接后继结点的地址的区域。
4. **头指针**：指向链表中第一个结点的指针。
5. **首元结点**：链表中存储第一个数据元素的结点。
6. **头结点**：在链表的首元结点之前预设的一个结点。
7. **链表**：n个结点由指针链组成一个链表，它是线性表的链式存储映像。
8. **单链表**：结点中只有一个指针域的链表（存放直接后继的地址），单链表由头指 针唯一确定，因此单链表可以用头指针的名字来命名。
9. **双链表**：结点中有两个指针域的链表（存放直接前趋和直接后继的地址）。
10. **循环链表**：首尾相接的链表。

**注**：有两种链式存储结构：带头结点和不带头结点。头结点的数据域可以为空，也可以存放 线性表长度等附加信息，但此结点不计入链表长度值。

④**空表的表示方式**

无头结点时，头指针为空表示空表；有头结点时，头结点的指针域为空表示空表。

⑤**设置头结点的好处**

1. 便于首元结点的处理。首元结点的地址保存在头结点的指针域中，所以在链表的

第一个位置上的操作和其他位置一致，无需进行特殊处理。

1. 便于空表和非空表的统一处理。无论链表是否为空，头指针都是指向头结点的非空指针，因此空表和非空表的处理也统一了。
2. **单链表**

①**结点类型定义**

typedef struct Lnode {

ElemType data; //数据域

struct Lnode\* next; //指针域

}Lnode；

typedef struct Lnode\* LinkList;

②**操作算法**

1. **建立单链表L**

void CreatListHead(LinkList\* L, int n) //头插法

{

LinkList p;

int i = 0;

srand(time(0)); //初始化随机数种子

\*L = (LinkList)malloc(sizeof(Node));

(\*L)->next = NULL; //建立一个带头结点的单链表

for (i = 0; i < n; i++)

{

p = (LinkList)malloc(sizeof(Node)); //生成新结点

p->data = rand() % 100 + 1; //随机生成100以内的数字

p->next = (\*L)->next;

(\*L)->next = p; //插入到表头

}

}

void CreatListTail(LinkList\* L, int n) //尾插法

{

LinkList p, r;

int i = 0;

srand(time(0)); //初始化随机数种子

\*L = (LinkList)malloc(sizeof(Node));

r = \*L; //r为指向尾部的结点

for (i = 0; i < n; i++)

{

p = (Node\*)malloc(sizeof(Node)); //生成新结点

p->data = rand() % 100 + 1; //随机生成100以内的数字

r->next = p; //将表尾终端结点的指针指向新结点

r = p; //将当前的新结点定义为表尾终端结点

}

r->next = NULL; //链表创建完成

}

1. **单链表L取值(取第i个元素的内容)**

Status GetElem(LinkList L, int i, ElemType\* e)

{

int j = 0;

LinkList p; //声明一个结点p

p = L->next; //让p指向链表L的第一个结点

j = 1; //计数器

while (p && j < i) //p不为空或计数器j还没有等于i时，循环继续

{

p = p->next; //p指向下一个结点

j++;

}

if (!p || j > i)

{

return ERROR; //第i个元素不存在

}

\*e = p->data; //取第i个元素赋值给e

return OK;

}

1. **单链表L插入(在第i个结点前插入新结点)**

Status ListInsert(LinkList\* L, int i, ElemType e)

{

int j = 1;

LinkList p = \*L;

static LinkList s = NULL;

while (p && j < i) //寻找第i个结点

{

p = p->next;

j++;

}

if (!p || j > i)

{

return ERROR; //第i个元素不存在

}

s = (LinkList)malloc(sizeof(Node)); //生成新结点

s->data = e;

s->next = p->next; //将p后继结点赋值给s的后继

p->next = s; //将新结点s赋值给p的后继

return OK; //插入完成

}

1. **单链表L删除(删除第i个结点)**

Status ListDelete(LinkList\* L, int i, ElemType\* e)

{

int j = 1;

LinkList p = \*L;

LinkList q = NULL;

while (p->next && j < i) //遍历寻找第i个元素

{

p = p->next;

j++;

}

if (!(p->next) || j > i)

{

return ERROR; //第i个元素不存在

}

q = p->next; //将p的后继赋值给q

p->next = q->next; //将q的后继赋值给p的后继

\*e = q->data; //将q的数据赋值给e

printf("%d处的数据%d已删除\n", i, \*e);

free(q);

q = NULL;

return OK;

}

1. **清空单链表L**

Status ClearList(LinkList\* L)

{

LinkList p = (\*L)->next; //p指向第一个结点

LinkList q = NULL;

while (p) //p没到表尾

{

q = p->next;

free(p);

p = q;

}

(\*L)->next = NULL; //头结点指针域设为空

printf("链表已清空\n");

return OK;

}

1. **循环链表**

①**定义**：循环链表是一种头尾相接的链表（即表中最后一个结点的指针域指向头结点， 整个链表形成一个环）。

②**优点**：从表中任一结点出发均可找到表中其他结点。

③**尾指针**：由于表的操作常常在表的首尾位置上进行，用头指针表示单循环链表不方便， 所以用尾指针rear表示单循环链表。此时开始结点为：rear->next->next。

④**带尾指针循环链表的合并算法**

p = rearA->next; //保存A表的头结点

rearA->next = rearB->next->next; //将指向B表的第一个结点赋值给rearA->next

rearB->next = p; //将A表的头结点赋值给rearB->next

free(p); //释放p

p=NULL；

**注**：其实循环链表和单链表的主要差异就在于循环的判断条件上，原来是判断p->next 是否为空，现在则是判断p->next是否等于头结点。

1. **双向链表**

①**定义**：在单链表的每个结点里再增加一个指向其直接前驱的指针域prior，这样链表 中就形成了有两个方向不同的链，故称为双向链表。

②**双向链表的结构定义**

typedef struct DulNode{

ElemType data;

struct DulNode\* prior; //直接前驱指针

struct DulNode\* next; //直接后驱指针

}DulNode,\*DuLinkList;

③**对称性**（设指针p指向某一结点）：p->prior->next = p = p->next->prior

④**操作算法**

在双链表中的操作（如List Length，GetElem等）因仅涉及一个方向的指针，故它们的算法与线性链表的相同。但在插入、删除时，则需同时修改两个方向上的指针，两者的操作的时间复杂度均为O(n)。

1. **双向链表的插入**（将结点s插入到p和p->next之间）

s->prior = p; //把p赋值给a的前驱

s->next = p->next; //把p->next赋值给s的后继

p->next->prior = s; //把s赋值给p->next的前驱

p->next = s; //把s赋值给p的后继

B. **双向链表的删除**

p->prior->next = p->next; //把p->next赋值给p->prior的后继

p->next->prior = p->prior; //把p->prior赋值给p->next的前驱

free(p); //释放结点p

p=NULL；

1. **顺序表与线性表的比较**

|  |  |  |  |
| --- | --- | --- | --- |
| 存储结构  比较项目 | | 顺序表 | 链表 |
| 空间 | 存储空间 | 预先分配，会导致空间闲置或溢出现象 | 动态分配，不会出现存储空间限制或溢出现象 |
| 存储密度 | 不用为表示节点间的逻辑关系而增加额外的存储开销，存储密度等于1 | 需要借助指针来实现元素间的逻辑关系，存储密度小于1 |
| 时间 | 存取元素 | 随机存取，按位置访问元素的时间复杂度为O(1) | 顺序存储，按位置访问元素的时间复杂度为O(n) |
| 插入、删除 | 平均移动约表中一半元素，时间复杂度为O(n) | 不需要移动元素，确定插入、删除位置后，时间复杂度为O(1) |
| 适用情况 | | ①表长变化不大，且事先确定变化范围  ②很少进行插入或删除操作，经常按元素位置序号访问数据元素 | ①长度变化不大  ②频繁进行插入或删除操作 |

**注：**

1. 栈和队列
2. **栈的定义**

栈是一个特殊的线性表，是限定仅在一端(通常是表尾)进行插入和闪出操作的线性表。又称为后进先出的线性表，简称LIFO结构。把允许插入和删除的一端称为栈顶，另一端称为栈底，不含任何元素的栈称为空栈。

1. **栈的顺序存储结构**

①**结构定义**

typedef struct{

SElemType data[MAXSIZE];

int top; //栈顶指针

}SqStack;

②**入栈操作**（向s中插入新元素e）

Status Push(SqStack\* s, SElemType e)

{

if (s->top == MAXSIZE - 1) //栈满

{

return ERROR;

}

s->top++;

s->data[s->top] = e;

return OK;

}

③**出栈操作**（删除s的栈顶元素，用e返回其值）

Status Pop(SqStack\* s, SElemType\* e)

{

if (s->top == -1)

{

return ERROR;

}

\*e = s->data[s->top];

s->top--;

return OK;

}

④**两栈共享空间结构定义**(适用于两栈的数据类型相同且空间需求为相反关系时)

typedef struct

{

SElemType data[MAXSIZE];

int top1; //栈1栈顶指针

int top2; //栈2栈顶指针

}SqDoubleStack;

⑤**两栈共享空间进栈操作**（向s中插入元素e，stackNumber判断是栈1还是栈2）

Status Push(SqDoubleStack\* s, SElemType e, int stackNumber)

{

if (s->top1 + 1 == s->top2) //栈满

{

return ERROR;

}

if (stackNumber == 1) //栈1有元素进栈

{

s->data[++s->top1] = e; //top1加一后赋值

}

else if (stackNumber == 2) //栈2有元素进栈

{

s->data[--s->top2] = e; //top2减一后赋值

}

return OK;

}

⑥**两栈共享空间出栈操作**（删除S的栈顶元素，用e返回其值，stackNumber判断是栈1

还是栈2）

Status Pop(SqDoubleStack\* s, SElemType\* e, int stackNumber)

{

if (stackNumber == 1)

{

if (s->top1 == -1) //栈空

{

return ERROR;

}

\*e = s->data[s->top1--];

}

else if (stackNumber == 2)

{

if (s->top2 == MAXSIZE) //栈空

{

return ERROR;

}

\*e = s->data[s->top2++];

}

return OK;

}

1. **栈的链式存储结构**(栈顶放在单链表的头部且不需要头结点)

①**结构定义**

typedef struct StackNode{

SElemType data;

struct StackNode\* next;

}StackNode,\*LinkSackPtr;

typedef struct LinkStack

{

LinkStackPtr top;

int count;

}LinkStack;

②**进栈操作**（向S中插入元素为e的新元素）

Status Push(LinkStack \* S, SElemType e)

{

LinkStackPtr s = (LinkStackPtr)malloc(sizeof(StackNode));

s->data = e;

s->next = S->top; //当前栈顶值赋值给新结点的直接后继

S->top = s; //将新结点s赋值给栈顶指针

S->count++;

reurn OK;

}

③**出栈操作**（删除S栈顶元素，用e返回其值）

Status Pop(LinkStack \* S, SElemType \* e)

{

LinkStackPtr P;

if (StackEmpty(\*s))

{

return ERROR;

}

\*e = S->top->data;

p = S->top; //栈顶结点赋值给p

S->top = S->top->next; //栈顶指针下移一位，指向后一结点

free(p); //释放结点p

S->count--;

return OK;

}

**注**：如果栈的使用过程中元素的变化不可预料，有时很小有时很大，则最好使用链栈；反之，如果它的变化在可控范围内，建议使用顺序栈。

1. **栈的应用：递归**

①递归的定义

把一个直接调用自己或通过一系列的调用语句间接地调用自己的函数称做递归函数。每个递归定义必须至少有一个条件，满足时递归不再进行，即不再引用自身而是返回值退出。

②打印前40个斐波那契数列数

int Fbi(int i)

{

if (i < 2)

{

return i == 0 ? 0 : 1;

}

return Fbi(i - 1) + Fbi(i - 2); //调用函数自己

}

int main()

{

int i;

for (int i = 0; i < 40; i++)

{

printf("%d", Fbi(i));

}

return 0;

}

**注**：前行阶段对于每层递归，函数的局部变量、参数值以及返回地址都被压入栈中。退回阶段位于栈顶的局部变量、参数值和返回地址被弹出，用于返回调用层次中执行代码的其余部分，也就是恢复了调用的状态。

1. **栈的应用：四则运算表达式求值** （如计算“9+（3+1）\*3+10/2”的值）

①**原理**

我们把上述表达式叫做中缀表达式，即所有运算符都在两数字中间。而我们需要将其转化为后缀表达式进行入栈出栈操作。从左到右遍历后缀表达式的每个数字和符号，遇到数字就进栈，遇到是符号就将栈顶两个数字出栈，进行运算，运算结果进栈，一直到最终获得结果。

②**后缀表达式转化规则**

从左到右遍历中缀表达式的每个数字和符号，若是数字就输出，即成为后缀表达式的一部分；若是符号则判断其与栈顶符号的优先级，是右括号或优先级低于栈顶符号则栈顶元素依次出栈并输出，并将当前符号进栈，一直到最终输出后缀表达式为止。（则：“9+（3-1）\*3+10/2” -> “9 3 1 - 3 \* + 10 2 / +”）

|  |  |  |
| --- | --- | --- |
| 对栈操作 | 栈内元素 | 输出元素 |
| ①初始化空栈 | 空 | 无 |
| ②输出9，“+”入栈 | + | 9 |
| ③ “(”入栈，输出3，“-”入栈，输出1 | + ( - | 9 3 1 |
| ④“)”与“(”匹配，输出“-”，“(”出栈 | + | 9 3 1 - |
| ⑤“\*”进栈，输出3 | + \* | 9 3 1 - 3 |
| ⑥“\*”出栈，第一个“+”出栈， | 空 | 9 3 1 - 3 \* + |
| ⑦第二个“+”进栈，输出10，“/”进栈，输出2 | + / | 9 3 1 - 3 \* + 10 2 |
| ⑧“/”出栈，“+”出栈 | 空 | 9 3 1 - 3 \* + 10 2 / + |

注：简述上述表达式转化步骤：（“+”优先级最低，遍历后栈内元素全部出栈，如⑥）

③利用后缀表达式计算原理

从左到右遍历表达式的数字和符号，遇到数字就进栈，遇到符号就将处于栈顶两个数字出栈，进行运算，运算结果出栈，一直到最终获得结果。

（后缀表达式：“9 3 1 - 3 \* + 10 2 / +”）

|  |  |
| --- | --- |
| 对栈操作 | 栈内元素（栈底->栈顶） |
| ①初始化空栈 | 空 |
| ②9、3、1进栈 | 9 3 1 |
| ③1、3出栈，3-1=2，2入栈 | 9 2 |
| ④3入栈 | 9 2 3 |
| ⑤3、2出栈，2\*3=6，6入栈 | 9 6 |
| ⑥6、9出栈，9+6=15，15入栈 | 15 |
| ⑦10、2入栈 | 15 10 2 |
| ⑧2、10出栈，10/2=5，5入栈 | 15 5 |
| ⑨5、15出栈，15+5=20，20进栈 | 20 |
| ⑩遍历结束，20出栈，结果为20 | 空 |

1. **队列的定义**

只允许在一端进行插入操作，而在另一端进行删除操作的线性表。队列是一种先进先出的线性表，简称FIFO。允许插入的一端称为队尾，允许删除的一端称为队头。

1. **队列的顺序存储结构**（循环队列）

①**结构代码**

typedef struct

{

QelemType data[MAXSIZE];

int front; //头指针

int rear; //尾指针，若队列不空，则指向队尾元素的下一个位置

}SqQueue;

②**初始化**

int InitQueue(SqQueue \*Q)

{

Q->front = 0;

Q->rear = 0;

return OK;

}

③**计算队列长度**

int QueueLength(SqQueue Q)

{

return (Q.rear - Q.front + MAXSIZE) % MAXSIZE;

}

④**入队列操作**

Status EnQueue(SqQueue\* Q, QElemType e)

{

if ((Q->rear + 1) % MAXSIZE == Q->front) //判断队列是否已满

return ERROR;

Q->data[Q->rear] = e; //将e赋值给队尾

Q->rear = (Q->rear + 1) % MAXSIZE; //rear后移一位，若到最后则转到队头

return OK;

}

⑤**出队列操作**

Status DeQueue(SqQueue\* Q, QElemType\* e)

{

if (Q->front == Q->rear) //判断队列是否为空

{

return ERROR;

}

\*e = Q->data[Q->fron]; //队头元素赋值给e

Q->front = (Q->front + 1) % MAXSIZE; //front指针后移一位，若到最后则转 到队头

return OK;

}

1. **队列的链式存储结构**（链队列）

①**结构代码**

typedef struct QNode //结点结构

{

QElemType data;

struct QNode\* next;

}QNode,\*QueuePtr;

typedef struct //队列的链表结构

{

QueuePtr front, rear;

}LinkQueue;

②**入队操作**

Status EnQueue(LinkQueue \* Q, QElemType e)

{

QueuePtr s = (QueuePtr)malloc(sizeof(QNode));

if (!s) //存储分配失败

{

exit(OVERFLOW);

}

s->data = e;

s->next = NULL;

Q->rear->next = s; //把拥有元素e的新节点s赋值给原队尾结点的后继

Q->rear = s; //把当前的s设置为队尾结点，rear指向s

return OK;

}

③**出队操作**

Status DeQueue(LinkQueue \* Q, QElemType \* e)

{

QueuePtr p;

if (Q->front == Q->rear)

{

return ERROR;

}

p = Q->front->next; //把要删除的队头结点暂存给p

\*e = p->data; //把要删除的队头结点的值赋值给e

Q->front->next = p->next; //将原队头结点的后继赋值给头结点的后继

if (Q->rear == p) //若队头是队尾，则删除后将rear指向头结点

{

Q->rear = Q->front;

}

free(p);

return OK;

}

**注**：在可以确定队列长度最大值的情况下，建议使用循环队列，如果无法预估队列长度时， 则用链队列。

1. 串
2. **串的定义**

由零个或多个字符组成的有限序列，又名字符串。（如：S=“a1a2a3......an”）

1. **基础概念**

①**空串**：零个字符的串，它的长度为零，可直接用“”或Φ表示。

②**空格串**：只包含空格的串，它是有长度的，而且可以不止一个空格。

③**字串与主串**：串中任意个数的连续字符组成的子序列转为该串的子串，相应的，包含 字串的串称为主串。

④**真字串**：不包含自身的所有字串。

1. **串的比较**

给定两个串：S=（S=“”,T=“”）。

①若n < m,且ai = bi(i=1,2, ...... ,n)。（如：“hap”<“happy”）

②若存在K<min(m,n),使得ai = bi(i=1,2, ...... ,k-1),ak<bk。（如：“happen”<“happy”）

③当且仅当n=m,且ai=bi(i = 1,2, ...... ,n)时，S = T。

1. **查找字符串：（朴素模式匹配）**

//S中第pos个字符后查找T,并返回位置

int Index(String S, String T, int pos)

{

int n, m, i;

String sub;

if (pos > 0)

{

n = Strlength(S); //字符串S的长度

m = StrLength(T); //字符串T的长度

i = pos;

while (i <= n - m + 1)

{

//取主串S第i个位置，长度与T相等，赋值给sub

SubString(sub, s, i, m);

if (StrCompare(sub, T) != 0) //若sub与T不相等

{

++i; //向后平移，继续比较

}

else //若sub与T相等

{

return i; //返回位置

}

}

return 0; //无字符串与T相等

}

1. **串的顺序存储结构**

用一组地址连续的存储单元来存储串中的字符序列。一般用定长数组来定义。通常用malloc()和free()来动态分配和管理。

1. **串的链式存储结构**

与线性表相似，但一个结点可以存放一个或多个字符，最后一个结点未被占满时用“#”或其他字符补全。具体存放多少字节根据实际情况选择。

1. **KMP模式匹配算法**

（S=“”,T=“”）

①**原理**：当T的第j位字符与S的对应位置不同时，可将最长相等前后缀的前缀移到后 缀对应的位置继续进行比较。

**如**：若T=“abcabcd”,S=“abcaebd”。T的第5位与S的第5位不相等，而第二个‘b’ 之前的最长相等前后缀为a,所以可将第一个‘a’平移到第二个‘a’的位置继续与 S相比较。

**再如**：T=“abcabca”,S=“abcabbd”。T的第6位与S的第6位不相等，而第三个‘b’之前的最长相等前后缀为‘ab’,则可将第一个‘ab’平移到第二个‘ab’的位置继续与S相比较。

**注**：如abcdcabcd的最长前后缀为abcd；aaaaaa的最长前后缀为aaaaa。为方便计算机处理实现KMP，建立next数组，具体规则如下。

②**next数组**

把T串各个位置j值的变化定义为一个数组next，那么next的长度就是T串的长度。若T第j位字符与S不匹配时，则T的next[j]位字符与S的当前位进行比较。（next[j]取决于当前不匹配字符之前的串的前后缀相似度）

0, j=1

next[j]= MAX{k | 1<k<j,且=}, 集合不为空

1. 其他情况

**如**：字符串T=“abcdex”,j=（1,2,3,4,5,6）时，对应的next[j]=（0,1,1,1,1,1）；

字符串T=“abcabx”,j=（1,2,3,4,5,6）时，对应的next[j]=（0,1,1,1,2,3）；

字符串T=“ababaaaba”,j=（1,2,3,4,5,6,7,8,9）时，对应的next[j]=（0,1,1,2,3,4,2,2,3）；

**注**：可观察到规律：T的j位字符前字符串的最长相等前后缀长度为n时，next[j]=n+1。

③**代码**

void get\_next(String T, int\* next) //通过计算返回子串T的next数组

{

int i, j;

i = 1;

j = 0;

next[1] = 0;

while (i < T[0]) //此处T[0]存放串T的长度，即i小于串T的长度时进入循环

{

if (j == 0 || T[i] == T[j]) //T[i]:后缀的单个字符 T[j]:前缀的单个字符

{

++i;

++j;

next[i] = j;

}

else

j = next[j]; //若字符不同，j值回溯

}

}

int Index\_KMP(String S, String T, int pos) //返回子串T在主串S中第pos个字符后的位置

{ //若不存在，则返回0

int i = pos; //i为主串S当前位置的下标

int j = 1; //j为子串T中当前位置下标

int next[255]; //定义next数组

get\_next(T, next); //对串T进行分析，得到next数组

while (i <= S[0] && j <= T[0]) //i < S长度且j < T长度时

{

if (j == 0 || S[i] == T[j])

{

++i;

++j;

}

else //指针后退，重新开始匹配

j = next[j]; //平移T串，即j退回合适位置，i不变

}

if (j > T[0])

return i - T[0]; //返回子串T在主串S中第pos个字符后的位置

else

return 0;

}

1. **KMP算法改进**

①**缺陷**：在主串S=“aaaabcde”中寻找字子串T=“aaaaax”时，发现第5位不相等：

1. 将T的第4位与S的第5位进行比较，发现又不相等；
2. 接着用T的第3、2、1位与S的第5位进行比较，发现都不相等；
3. 根据算法接着用T的第1位于S的第6位进行比较。

可见，由于T的第1、2、3、4位字符相等，所以第2）步的操作是多余的。

②**改良**：构造nextval数组如下

1）字符串T=“ababaaaba”，j=（1,2,3,4,5,6,7,8,9）时：

对应的next[j]=（0,1,1,2,3,4,2,2,3）；对应的nextval[j]=（0,1,0,1,0,4,2,1,0）

注：j=1时，nextval[1]=0；

j=2时，第2位字符b的next值为1，而第1位字符a与b不相等，所以

nextval[2] = next[2] = 1；

j=3时，第3位字符a的next值为1，而第1位字符a与a相等，所以

nextval[3] = nextval[1] = 0；

j=4时，第4位字符b的next值为2，而第2位字符b与b相等，所以

nextval[4] = nextval[2] = 1；

j=5时，第5位字符a的next值为3，而第3位字符a与a相等，所以

nextval[5] = nextval[3] = 0；

j=6时，第6位字符b的next值为4，而第4位字符a与b不相等，所以

nextval[6] = next[6] = 4；

j=7时，第7位字符a的next值为2，而第2位字符a与b不相等，所以

nextval[7] = next[7] = 2；

j=8时，第8位字符b的next值为2，而第2位字符b与b相等，所以

nextval[8] = nextval[2] = 1；

j=9时，第9位字符a的next值为3，而第3位字符a与a相等，所以

nextval[9] = nextval[3] = 0；

1. 字符串T=“aaaaaaaab”，j=（1,2,3,4,5,6,7,8,9）时：

对应的next[j]=（0,1,2,3,4,5,6,7,8）；对应的nextval[j]=（0,0,0,0,0,0,0,0,8）

注：j=1时，nextval[1]=0；

j=2时，第2位字符a的next值为1，而第1位字符a与a相等，所以

nextval[2] = nextval[1] = 0；

j=3时，第3位字符a的next值为1，而第1位字符a与a相等，所以

nextval[3] = nextval[1] = 0；

……

j=9时，第9位字符b的next值为8，而第8位字符a与b不相等，所以

nextval[9] = next[9] =8；

③**代码**

void get\_nextval(String T, int\* nextval) //通过计算返回子串T的nextval数组

{

int i, j;

i = 1;

j = 0;

nextval[1] = 0;

while (i < T[0]) //此处T[0]存放串T的长度，即i小于串T的长度时进入循环

{

if (j == 0 || T[i] == T[j]) //T[i]:后缀的单个字符 T[j]:前缀的单个字符

{

++i;

++j;

if(T[i] != T[j]) //若当前字符与前缀字符不同

nextval[i] = j; //当前的j为nextval在i位的值

else

nextval[i]=nextval[j]; //若当前字符与前缀字符相同，则将前缀字符的nextval值赋值给nextval在i位的值

}

else

j = nextval[j]; //若字符不同，j值回溯

}

}

int Index\_KMP(String S, String T, int pos) //返回子串T在主串S中第pos个字符后的位置

{ //若不存在，则返回0

int i = pos; //i为主串S当前位置的下标

int j = 1; //j为子串T中当前位置下标

int nextval[255]; //定义nextval数组

get\_nextval(T, nextval); //对串T进行分析，得到nextval数组

while (i <= S[0] && j <= T[0]) //i < S长度且j < T长度时

{

if (j == 0 || S[i] == T[j])

{

++i;

++j;

}

else //指针后退，重新开始匹配

j = nextval[j]; //平移T串，即j退回合适位置，i不变

}

if (j > T[0])

return i - T[0]; //返回子串T在主串S中第pos个字符后的位置

else

return 0；

}

1. 树
2. **树的定义**

树是n(n≥0)个结点的有限集。

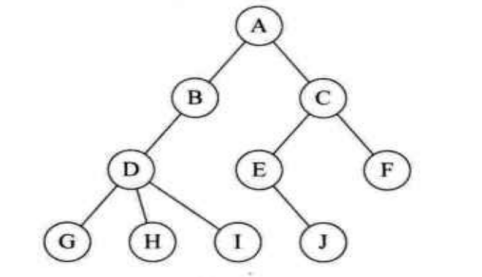
①若n = 0，称为空树。

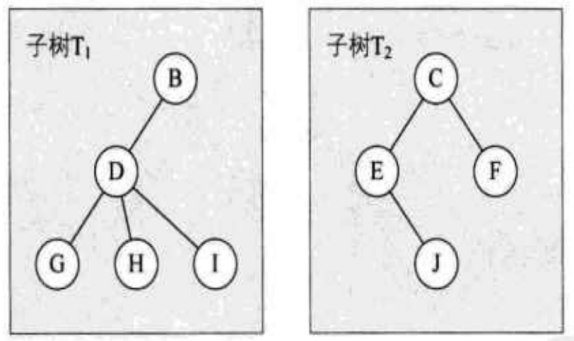
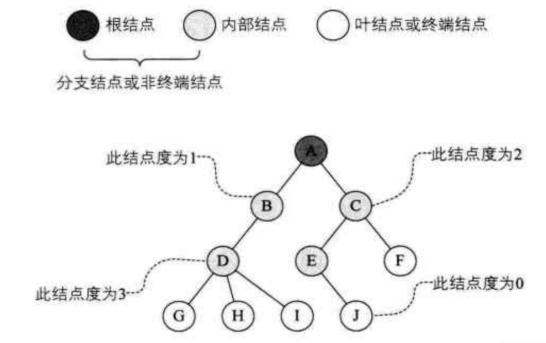
②若n＞0，则满足如下条件：

1）有且仅有一个特定的称为根的结点。

2）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，并称为根的子树。

1. **树的基本术语**



①**子树**：如：T1, T2。

②**根结点**：非空树中无前继结点的结点。

③**结点的度**：结点拥有的子树数称为结点的度。

④**叶结点**：度为零的结点，也叫终端结点。

⑤**分支结点及内部结点**：度不为零的结点，也叫非终端结点。除根结点外的分支结点称 为内部结点。

⑥**树的度**：树内各结点的度的最大值。

⑦**树的深度**：树中结点的最大层次称为树的深度或高度。

⑧**有序树及无序树**：将树中结点的各子树看成从左到右是有次序的，不能互换的，则称 该树为有序树，否则称为无序树。

⑨**森林**：m(m≥0)课互不相交的树的集合。对树中每个结点而言其子树的集合即为森林。

⑩**结点的层次**：从根开始定义，根为第一层，根的孩子为第二层，……

1. **结点间的关系**

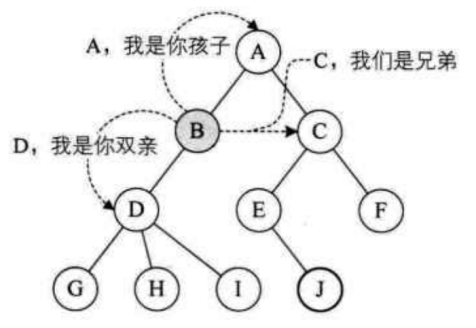
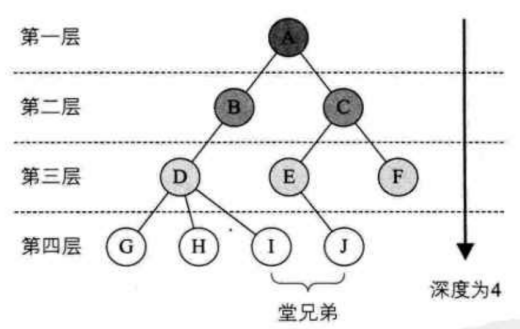
①**孩子及双亲**：结点的子树的根称为该结点的孩子；该结点称为孩子的双亲。

②**兄弟**：同一双亲的孩子之间互称兄弟。

③**祖先**：从根到该结点所经分支上的所有结点称为该结点的祖先。

④**子孙**：以某结点为根的子树中的任一结点都称为该结点的子孙。

⑤**堂兄弟**：其双亲在同一层的结点互为堂兄弟。

**注**：n>0时，根结点时唯一的，不可能存在多个根结点。m>0时，子树的个数没有限制，但 它们一定是互不相交的。

1. 树的顺序存储结构（双亲表示法）

①结点结构定义：

1. 树的链式存储结构：（孩子表示法、孩子兄弟表示法）