**数据结构**

**预备知识**

**什么是数据结构？**

**我们如何把现实中大量而反复的问题以特定的数据类型和特定的存储结构保存到计算机的主存储器(内存)中，以及在此基础上为实现某个功能（如查找某个元素，删除某个元素，对所有元素进行排序）而执行的相应的操作，这个相应的操作也叫做算法。**

**狭义：  
 数据结构是专门研究数据存储的学科  
 数据存储包含两方面：个体的存储和个体之间关系的存储**

**广义:**

**数据结构既包含数据的存储也包含数据的操作**

**对存储数据的操作就是算法**

**数据的逻辑结构:**

**数据的逻辑结构是从数据元素的逻辑关系上来描述数据的，数据的逻辑结构与数据的存储无关，是独立于计算机的，它是从具体问题中抽象出来的数学模型**

**数据的逻辑结构分3类  
线性结构  
定义：该结构中数据的元素之间存在一对一的关系  
特点：开始元素没有前驱元素，终端元素没有后继元素，其余元素有且仅有一个前驱元素和一个后继元素，比如线性表**

**树形结构  
定义：该结构中数据的元素之间存在一对多的关系  
特点：除了开始元素以外，每个元素有且仅有一个前驱元素，除了终端元素以外，每个元素有一个或多个后继元素，比如二叉树**

**图形结构  
该结构中数据的元素之间存在多对多的关系  
特点  
每个元素的前驱和后继元素可以有任意多个，该结构可能没有开始和终端元素，也可能有多个开始和终端元素**

**注: 树形结构和图形结构又统称为非线性结构**

**数据的存储结构  
 定义：**

**数据的逻辑结构在计算机存储器中的存储表示称为数据的存储结构，简而**

**言之，就是数据的逻辑结构在计算机中的存储实现**

**4中常见的存储结构  
顺序存储  
原理：用一组地址连续的存储单元来存放所有的数据元素  
特点：  
 1.所有元素在存储器中占有一整块存储空间  
 2.逻辑相邻的元素在计算机存储器中的存储位置也相邻**

**优点：**

1. **因为每个元素对应一个逻辑序号，由该序号可以直接计算出对应元素的 存储地址，所以可以实现随机存取，即对元素进行存取操作快**
2. **存储效率高，即系统分配给元素的存储空间完全用来存储元素，元素之间的关系不占存储空间**

**缺点：**

1. **不便于数据修改，即对元素的删除或插入，往往需要移动一系列元素**

**链式存储  
原理：每个数据元素用一个内存结点存放  
特点：**

1. **内存结点是单独且随机分配的，所有结点的地址不一定连续，这样导致了 逻辑相邻的元素在计算机存储器中的存储位置不一定连续**
2. **每个内存结点包括数据域和指针域，通过指针域将所有结点链接起来**

**优点：**

1. **便于数据修改，对插入元素和删除元素，只需修改相应结点的指针域，不用移动元素**

**缺点：**

1. **由于都每个元素分配的存储单元地址不一定连续，所以不能实现随机存取，即对元素进行存取操作慢**
2. **存储空间利用率较低，即用该结构存储数据，占内存较大**

**索引存储  
优点：查找效率高  
缺点：因为需要建立索引表，从而增加了空间的开销**

**哈希存储  
根据元素的关键字通过哈希函数直接计算出一个值，将这个值作为该元素的存储地址  
优点：查找速度快  
缺点：哈希存储只存储元素的数据，不存储元素之间的关系**

**注：数据的逻辑结构采用不同的存储方法可以得到不同的存储结构  
如：线性表既有顺序存储结构也有链式存储结构.......**

**线性表(重点)  
 定义：**

**具有相同特性的数据元素组成的有限序列  
 可以为空，线性表中的元素个数也是线性表的长度**

**线性表的顺序存储结构称为顺序表  
所谓顺序表，就是把线性表中的所有元素按其逻辑顺序依次存储到计算机存储器中指定的一块地址连续的存储空间中。  
在c或c++语言中顺序表可以用数组来实现，即用数组存放线性表中的元素及逻辑关系，但数组的大小必须大于等于线性表的长度**

**顺序表的类型**

**typedef struct**

**{**

**int data[MaxSize]；//存放线性表中元素**

**int length；//记录线性表的长度(也是对应顺序表的长度)**

**}SqList；**

**typedef 用来重新定义一个新的数据类型**

**即将某个数据类型重新换一个名字**

**优点：  
 1.因为顺序表采用的是顺序存储结构，所以具有随机存取特性，即对元素**

**的存取操作很快**

**缺点：  
 1.插入，删除元素很慢，即对数据进行修改的操作很慢  
 2.必须是事先知道顺序表的大小  
 3.存储空间通常是有限的  
 4.需要大块的连续存储空间，降低了存储空间的利用率  
  
线性表的链式存储结构称为链表**

**所谓链表，就是把线性表中的所有元素，用一个个内存结点存放，结点之间通过指针相连  
链表可以有单链表，双链表，循环单链表，循环双链表，链表中的元素是采用结点来存放**

**什么是结点？  
结点实际上是内存中的一块存储空间，该存储空间被分为两个部分，一部分用来存储数据，另一部分用来存储地址，对应的就是结点的数据域和指针域，每个结点之间通过指针相连**

**什么是指针域和数据域**

**简单的说数据域用来存放数据，指针域用来存放地址**

**具体的来讲数据域用来存储数据元素本身的信息，指针域用来存储数据元素之间的关系**

**结点的指针域实际上就是与结点数据类型相同的一个指针变量，用来存放该结点的后继结点的地址，从而指向它的后继结点**

**链表中的结点可以分为  
头结点：链表最前面的那个结点，它不存储有效的数据，即数据域为空。指向头结点的指针称为头指针，用来唯一标识一个链表。头结点的作用是方便对链表进行操作。  
首结点：链表中第一个数据域不为空的结点，指向它的指针称为首指针  
尾结点：链表中的最后一个结点，指向它的指针称为尾指针  
  
优点：  
 1.对数据元素的修改操作很快，比如删除，插入元素的操作，只需修改相**

**关结点的指针域即可**

**缺点  
1.链表占用的存储空间比顺序表大，多了个指针域  
2.因为结点是单独且随机分配的，结点的地址是不一定连续，所以线性表采用链**

**式存储时，逻辑相邻的元素存储在链表中的存储地址不一定相邻，即可能相**

**邻，也可能不相邻，因此链表不具备随机存取特性，所以当要在链表中找某个结点时，需要先定义一个指向头结点的指针(准确来说是定义一个指向链表第一结点的指针)，再将该指针逐步后移，直到该指针指向要找的结点为止，因此链表具有顺序存取特性**

**单链表**

**单链表中的结点只包含一个数据域和一个指针域，单链表中的指针域只能存放后继结点的地址，并且单链表中尾结点的指针域为空**

**单链表中结点的类型**

**typedef struct node**

**{**

**int data；//存放元素值**

**struct node \*next；//存放后继结点的地址，从而指向它**

**}LinkNode；**

**循环单链表**

**将单链表中尾结点原来为空的指针域改为指向头结点，从而构成了一个单向循环，这就是循环单链表**

**双链表**

**双链表中结点包含一个数据域和二个指针域，一个指针域(prior域)存放前驱结点的地址，另一个(next域)存放后继结点的地址，并且双链表中头结点的prior域为空，尾结点的next域为空**

**双链表中结点的类型**

**typedef struct node**

**{**

**int data；//存放元素值**

**struct node \*prior；//存放前驱结点的地址，从而指向它**

**struct node \*next；//存放后继结点的地址，从而指向它**

**}DLinkNode；**

**循环双链表**

**将双链表中头结点的prior域由原来的空，改为指向尾结点，将尾结点的next域也由原来的空，改为指向头结点，从而构成了一个双向循环，这就是循环双链表**

**拓展:**

1. **在单链表中某个位置插入一个结点，或删除某个结点，必须找到该位置前一个位置的结点，比如删除或插入单链表中的第i个结点必需找到第i-1个结点。**

**但在双链表中某个位置插入一个结点，或删除某个结点，可以直接找该位置的结点，因为双链表有两个指针域，分别指向其前驱和后继**

**（2）与单链表相比，双链表访问一个结点的前，后结点更方便**

**（3）单链表的数据类型和循环单链表完全相同，所占存储空间大小也完全相同。**

**双链表的数据类型和循环双链表完全相同，所占存储空间大小也完全相同。**

**(4) 在链表中插入或删除一个结点，需要特别注意结点丢失的问题，尤其是单链表**

**（5）在循环单链表/双链表L中判断尾结点p的条件是p->next = = L**

**(7) 循环单链表/双链表L初始化为分别为：L->next = L**

**L->next = L->prior = NULL**

**(8) 计算链表的长度不包括头结点**

**(9) 建立链表有两种方法，头插法和尾插法。用头插法建立的单链表中的数据结点的顺序与之前插入元素的顺序相反，用尾插法建立的单链表中的数据结点的顺序与之前插入元素的顺序相同，即一个正序一个反序。在跟链表有关的算法中要灵活运用头插法和尾插法，比如在链表中某个位置插入一个元素，用的就是头插法**

**(10) 假设指针p用来指向链表中的结点**

**p->next == NULL**

**表示p所指向的结点的指针域为空**

**P == NULL**

**表示指针p没有存放任何结点的地址，即指向为空**

**有序表**

**有序表是指这样的线性表，其中所有元素递增或递减方式有序排列**

**即线性表中的元素有序排列，这样的线性表又称为有序表**

**注:**

**有序表的一些基本算法与线性表一样，只有插入元素的操作与线性表有些差异，在有序表中插入一个元素，需要将该元素与有序表中的元素进行比较，再插入**

**栈和队列(重点)**

**栈的定义:**

**后进先出表（Last In First Out，LIFO），一种能实现后进先出(先进后出)的存储结构，实际上就是操作受限的线性表，即只能在栈顶进行插入和删除元素，栈中不含任何元素称为空栈。**

**特点:**

**后进先出(先进后出)：后进栈的元素先出栈(先进栈的元素后出栈)，每次进栈的元素都放在原来栈顶元素之间，成为新的栈顶元素，并且出栈的元素一定是栈顶元素**

**两个重要的操作：**

**进栈(入栈，压栈)：即往栈中插入元素**

**出栈(退栈)：即删除栈中的元素**

**栈的链式存储结构称为链栈(动态栈)**

**栈的链式存储结构实际上也和线性表的链式存储结构一样，用链表来实现，这里采用带头结点的单链表来实现链栈，不使用栈顶指针**

**注意：采用本方法实现链栈(用一个带头结点的单链表，不设立栈顶指针)**

**元素进栈就是用头插法在头结点处将元素一个个插入到单链表中，就好比头插法建立单链表**

**元素出栈就是删除单链表的首结点，因为首结点就是栈顶结点**

**并且首结点即栈顶结点，尾结点即栈底结点**

**链栈中结点的类型**

**typedef struct node**

**{**

**int data；//数据域**

**struct node \*next //指针域**

**}LinkStNode；**

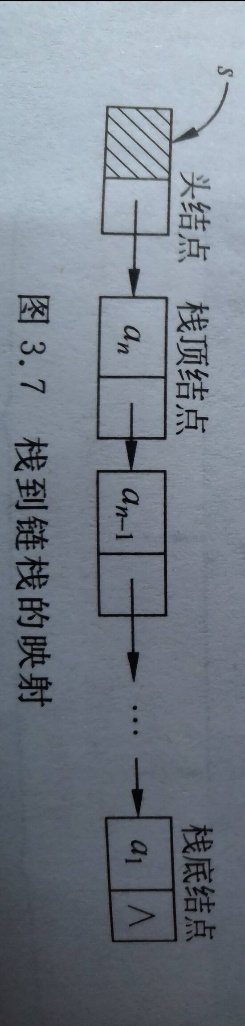
**假设有一个链栈S**

1. **判断链栈S为空的条件：S->next == NULL**

**若采用的是不带头结点的单链表，但有一个栈顶指针(S)，这样的单链表来实现链栈，该链栈为空的条件是：S == NULL；**

1. **栈满的条件：因为内存溢出才会出现栈满的情况，所以理论上链栈不存在栈满的情况**
2. **进栈：定义一个结点存放要进栈的元素，再采用头插法插入到对应单链表头结点后面**
3. **出栈：取出首结点的值域，再把它删除**
4. **销毁栈的算法和销毁单链表的算法完全相同**

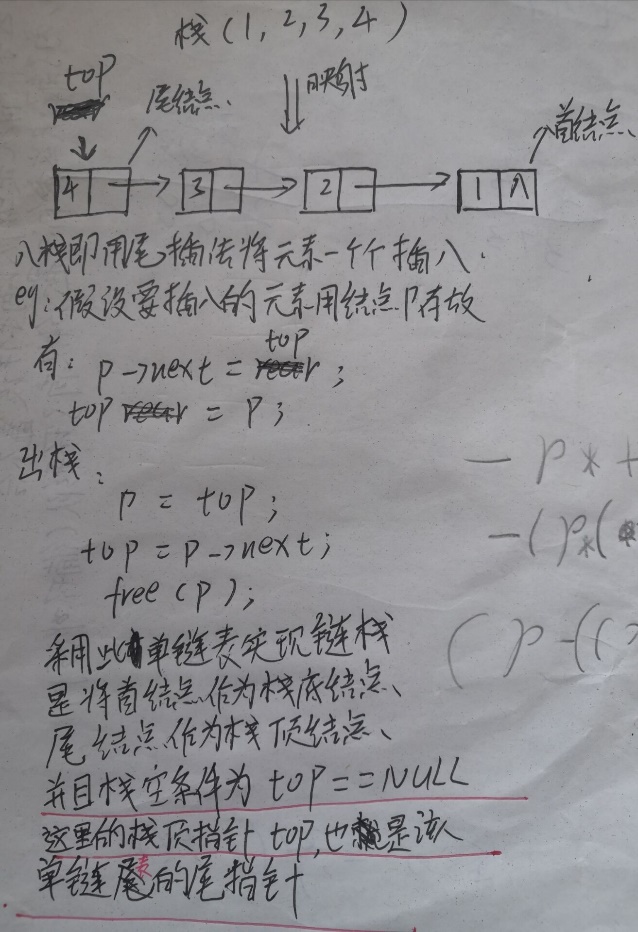
**图解：**

****

**拓展：**

**其实还可以不带头结点，仅设立一个栈顶指针(尾指针)的单链表来实现链栈**

**图解：**

****

**还可以用不带头结点，仅有一个尾指针(栈顶指针)rear的循环单链表来实现链栈，这样的链栈，栈空条件也是 rear == NULL**

**栈的顺序存储结构称为顺序栈(静态栈)**

**栈的顺序存储结构实际上也和线性表的顺序存储结构一样，用数组来实现，并且定义了一个栈顶指针用来存放栈顶元素的下标，这里设栈顶指针初始时从-1开始，数组下标从0开始**

**顺序栈的类型**

**typedef struct**

**{**

**int data[MaxSize]；//存放栈中的数据元素**

**int top；//存放栈顶元素在数组中的下标**

**}SqStack；**

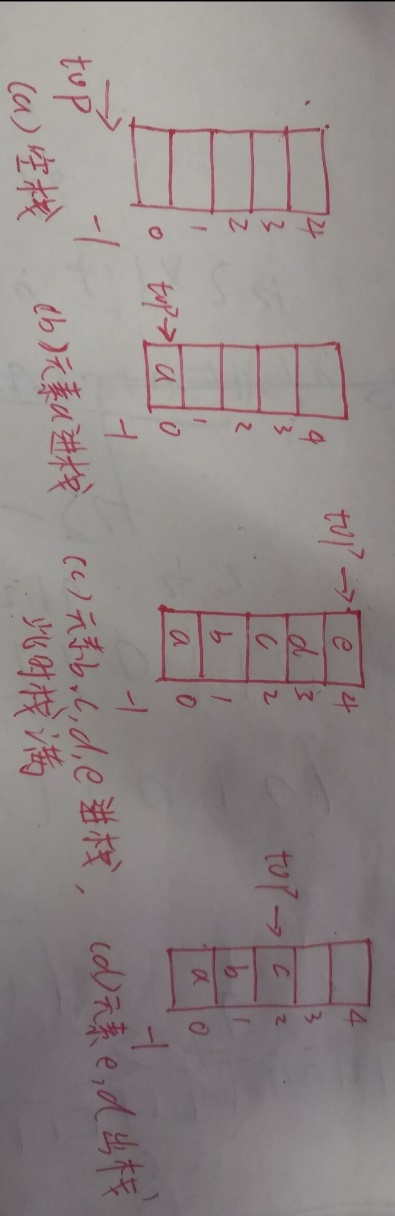
**假设有一个顺序栈S，栈顶指针为top**

1. **判断顺序栈S为空的条件：S->top == -1**
2. **栈满的条件：因为顺序栈本质是用数组来实现，当插入顺序栈的元素达到**

**数组最大下标时即为栈满，即S->top == MaxSize-1(数组的最大下标)**

1. **进栈：先将栈顶指针top增1，再将要进栈的元素放在栈顶指针处(即栈顶指针存放(指向)的数组下标内)**
2. **出栈：先将栈顶指针top处的元素取出，再将栈顶指针减1**

**图解：**

****

**注意设计顺序栈时，栈顶指针不一定从-1开始，也可以从0开始，若是从0开始，进/出队操作会有所不同，顺序栈空条件变为S->top == 0，顺序栈满条件变为S->top == MaxSize**

**共享栈(也属于顺序栈)**

**定义：用一个数组实现两个栈，即两个栈共用一个数组**

**假设有个数组data[MaxSize]，两个栈(栈1，栈2)有两个栈底，让一个栈底为数组的初始端，即下标为0出，另一个栈底为数组的末端，即下标为MaxSize-1处，并且定义两个栈顶指针(top1，top2)，一个初始为0，一个初始为MaxSize**

**共享栈的数据类型**

**typedef struct**

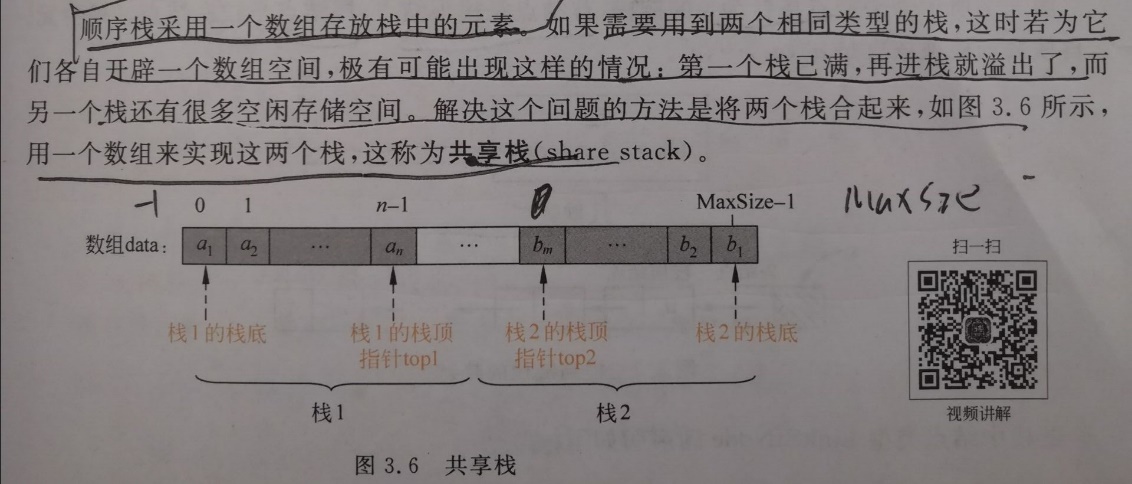
**{**

**int data[MaxSize]；**

**int top1，top2；**

**}DStack；**

**图解**

****

**共享栈的4个要素**

1. **栈空条件：栈1空为top1 == -1；栈1空为top1 == MaxSize；**
2. **栈满条件：top1 == top2 -1 或 top2 == top1 + 1；**
3. **元素x进栈操作：进栈1操作为 top1++；data[top1] = x；**

**进栈2操作为 top2--；data[top2] = x；**

**（4）元素x出栈操作：出栈1操作为 x = data[top1]；top1--；**

**出栈2操作为 x = data[top2]；top1++；**

**拓展：**

**顺序栈与链栈的对比**

1. **链栈理论不存在栈满溢出的情况(这是较顺序栈最大的优点)**
2. **因为规定链栈的所有操作都在栈顶进行的，即在对应单链表的表头进行，比如往链栈中插入或删除元素都十分方便，因为都是在单链表的头结点处进行操作，所以不存在往顺序栈中插入或删除元素比链栈快，这一问题**
3. **顺序栈和链栈中出栈，取栈顶元素都需要先判断栈是否为空**
4. **顺序栈入栈需要判断先栈是否已满，链栈不用**

**栈的应用**

**中缀/后缀/前缀表达式**

**什么是中缀/后缀/前缀表达式？**

**算术表达式中，运算符位于两个操作数中间的算术表达式称为中缀表达式**

**例如: 1 + 2 \* 3，运算符+，\*在两个操作数的中间**

**算术表达式中，运算符位于操作数后面的算术表达式称为后缀表达式，又称逆波兰表达式**

**例如:1 2 3 \* +，运算符+，\*在操作数的后面**

**算术表达式中，运算符位于操作数前面的算术表达式称为前缀表达式**

**例如:+ 1 \* 2 3，运算符+，\*在操作数的前面**

**eg:**

**2 \* (1 + 3) - 4 //这是一个中缀表达式**

**转化成前缀表达式为：- \* 2 + 1 3 4**

**转化成后缀表达式为：2 1 3 + \* 4 -**

**拓展：**

1. **中缀表达式是最常用的表达式，日常生活中表达式一般都是中缀表达式**
2. **中缀表达式的运算，遵循“先乘除后加减，从左到右，先括号内再括号外”的规则，因此中缀表达式不仅要考虑运算符优先级，还要处理括号**
3. **后缀表达式中已经考虑了运算符的优先级，所以该表达式中没有括号只有操作数和运算符，并且越靠前的运算符越优先执行**
4. **前缀表达式中已经考虑了运算符的优先级，所以该表达式中没有括号只有操作数和运算符，并且越靠后的运算符越优先执行**
5. **后缀表达式是一种十分有用的表达式，它将复杂的表达式转换为可以依靠简单操作得到计算结果的表达式**
6. **由(5)得出，对中缀表达式的计算，是先把它转换成后缀表达式，再计算**

**队列的定义：**

**先进先出表(First In First Out，FIFO)，一种能实现先进先出的存储结构，它也是一种操作受限的线性表，即只能表的一端进行插入操作，在另一端进删除操作**

**特点：**

**先进先出(后进后出)，即先进队的元素先出，后进的元素后出，每次出队的元素总是队首元素，入队的元素总是作为队尾元素。比如：进队入队就相当于几个人过一个独木桥，上桥顺序和下桥顺序相同**

**两个重要的操作：**

**进队(入队)：即往队列中插入元素，且只能从队尾插入**

**出队(离队)：即删除队列中的元素，且只能从队首删除**

**队首(头)元素：队列中的第一个元素，指向它的指针叫队首指针，但在顺序队中队首指针不一定指向队首元素**

**队尾元素：队列中的最后一个元素，指向队尾元素的指针叫队尾指针**

**队列的顺序存储结构称为顺序队**

**顺序队实际上也和线性表的顺序存储结构一样，用数组来实现，并定义了两个指针，分别为队首指针(front)，队尾指针(rear)，初始时都为-1**

**注：顺序队的队首指针，和队尾指针初始化也可以为0，即从0开始，这里采用队首指针，和队尾指针初始化为-1**

**顺序队的数据类型**

**typedef struct**

**{**

**int data[MaxSize]；**

**int front，rear；**

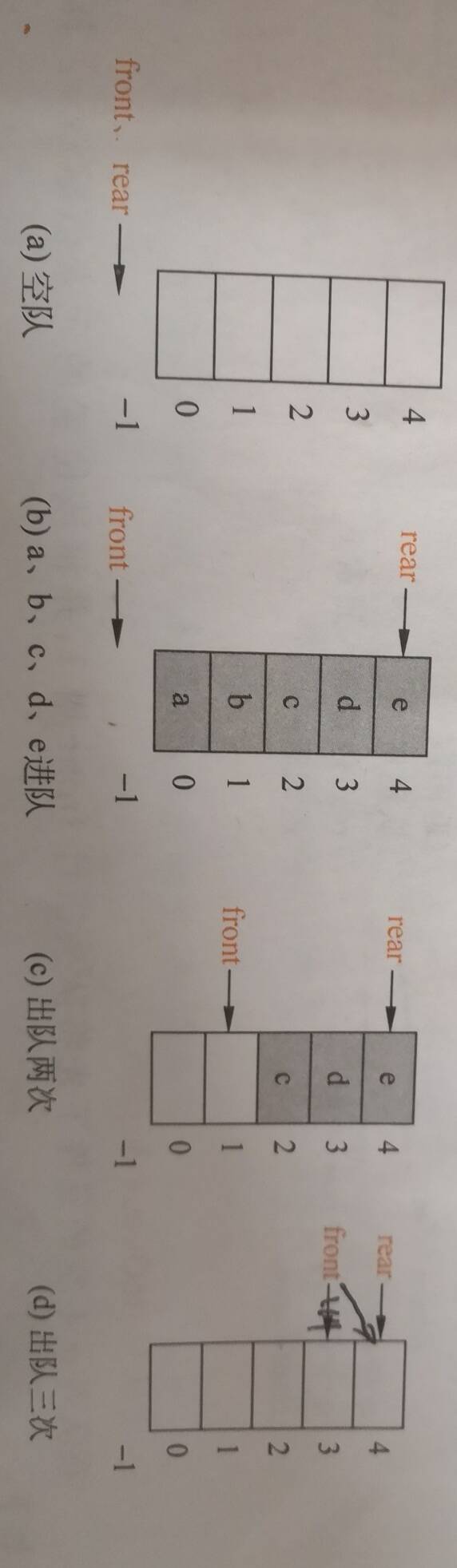
**}SqQueue；**

**假设有个顺序队q**

1. **队空条件：q->front == q->rear**
2. **队满条件：q->rear == MaxSize-1(数组的最大下标)**
3. **进队：先将q->rear 增1，然后将元素e放在q->rear处（即队尾指针存放(指向)的数组下标内）**
4. **出队：先将q->front 增1，然后取出q->front处的元素(即将队首指针存放(指向)的数组下标内的元素取出)**

**注：在顺序队中队首指针指向当前队首元素的前一个位置(即存放队首元素前一个位置的下标)**

**图解：**

****

**由上图中的(c)可以看出此时的队尾指针(rear)已到达最大数组下标位置，即队列已满，但队列仍有空位置，即队列中仍有空闲的存储空间没有用来存放数据，这种情况叫假溢出，造成了存储空间的浪费，这也是顺序队最大的缺陷，所以在其基础上改进有了循环队列**

**循环队列(采用的也是顺序存储结构，也属于顺序队)**

**定义：**

**循环队列也叫环形队列，实际上将数组的前后两端连接起来，形成一个环形数组，即把存储队列元素的数组从逻辑上看成是一个环**

**当队尾指针rear = MaxSize-1后，再前进一个位置就到达0，于是就可以使用另一端的空位置存放元素**

**循环队列的数据类型和上述的顺序队相同**

**假设有一个循环队列q**

1. **环形队列的队首指针(front)和队尾指针(rear)，初始时都为0，原来为-1，但队首指针没变仍指向当前队首元素的前一个位置**
2. **在进队出队时，队尾指针(rear)和队首指针(front)都循环增1**

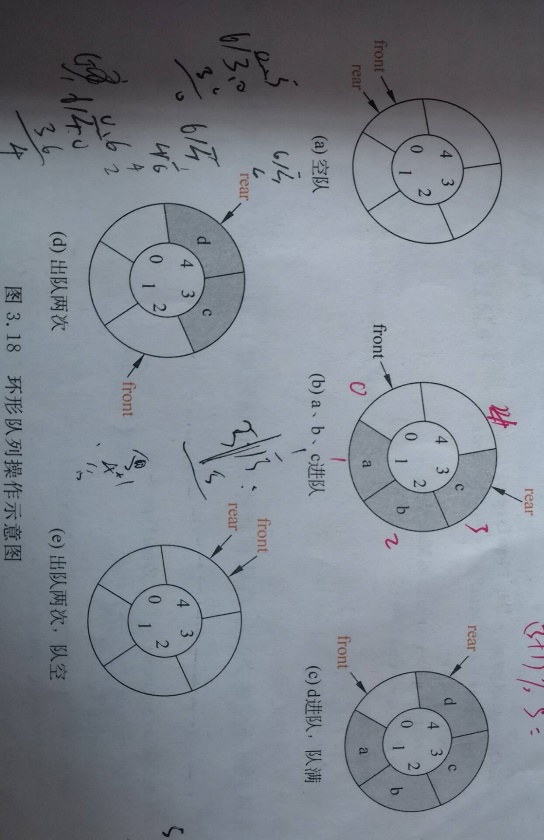
**即：q->front = (q->front + 1) % MaxSize**

**q->rear = (q->rear + 1) % MaxSize**

1. **队空条件没变，仍为 q->rear == q->front**
2. **当元素进队的速度快于元素出队速度时，队尾指针很可能会追赶上队首指针，很可能导致队满时队尾指针和队首指针指向数组中相同的位置，即q->rear == q->front ，这里有个问题，那岂不是无法判断队空和队满，所以循环队列采用另一种方法作为队满的条件，即以“队尾指针循环增1时等于队首指针”作为队满条件，即尝试进队一次，若到达队头，就认为队满，不能再进队，这样的方法导致循环队列少用了一个元素空间，所以循环队列中最多只能存放MaxSize-1个元素，当循环队列中存放的元素很多时，浪费的这一个元素空间，可以忽略不记**

**所以队满条件为：若 (q->rear + 1) % MaxSize == q->front，队满**

**图解：**

****

**拓展：**

1. **对顺序队的进行进队操作，需要要先判断队是否已满，链队不需要**
2. **对顺序队的进行出队操作，需要要先判断队是否为空**

**队列的链式存储结构称为链队**

**链队和线性表的链式存储结构一样，用链表来实现，这里采用一个带头结点单链表来实现链队，链队中的头结点(链队头结点)包含两个指针，分别是队首指针和队尾指针，队首指针(front)用来指向队首元素(区别于顺序队)，队尾指针(rear)用来指向队尾元素，front 和 rear 初始时都为空(NULL)**

**链队中包含两种结点，分别是链队头结点，和数据结点**

**链队中数据结点的数据类型**

**typedef struct qnode**

**{**

**int data；**

**struct qnode \*next；**

**}DataNode；**

**链队中头结点的数据类型**

**typedef struct**

**{**

**DataNode \*front；**

**DataNode \*rear；**

**}LinkQuNode；**

**注：**

**因为这里是用带头结点的单链表来实现链队**

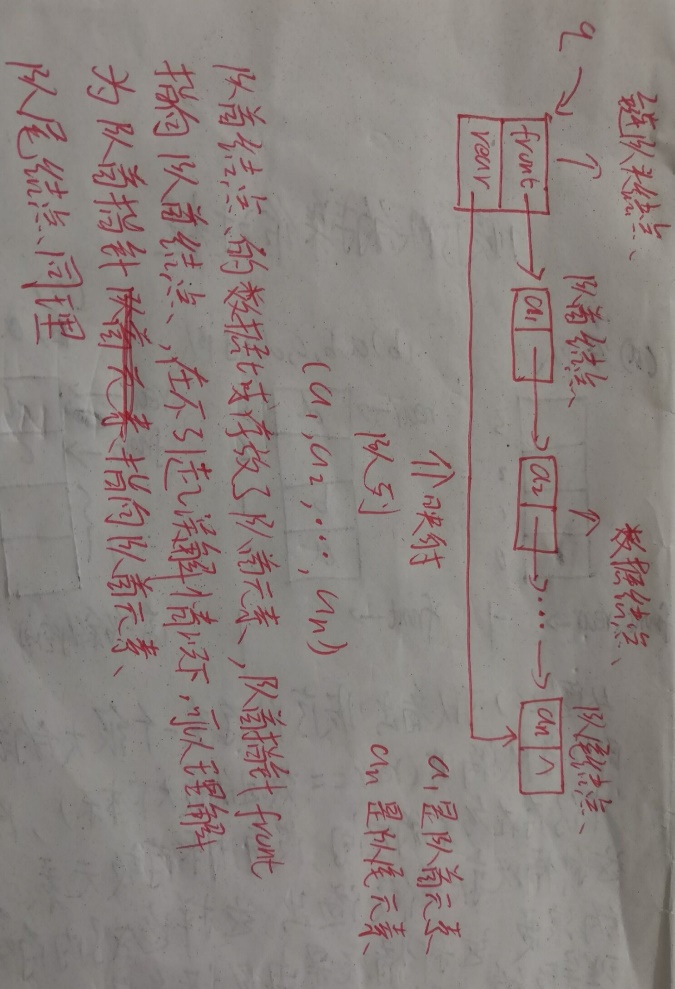
**进队操作就是用尾插法将元素一个个插入，类似于尾插法建立单链表**

**出队操作就是删除单链表中的首结点，首结点的数据域存放的就是队列中 的队首元素**

**假设有一个链队q**

1. **队空的条件：q->rear == NULL 或 q->front == NULL**
2. **队满的条件：因为链表中的结点时动态分配的，只有当内存出现溢出时，才会队满，所以不考虑队满的情况，原因同链栈一样**
3. **若q->front == q->rear 说明链队中只有一个元素**
4. **进队操作：新建一个结点存放要插入的元素，再尾插法将该结点插入，作为尾结点**
5. **出队操作：取出队首元素的data值并将其删除**

**图解：**

****

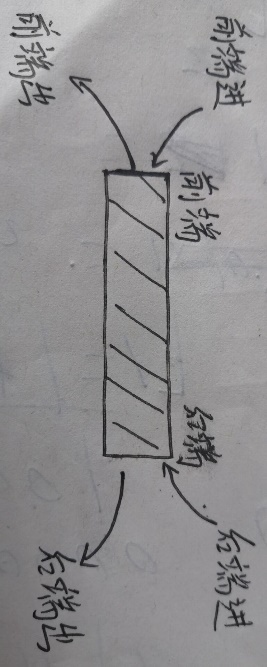
**拓展：**

1. **链队中进队操作：进队前不需要先判断队满情况，但需要注意当队空时，即q->rear或q->front 为空(NULL)，这时进队的元素是第一个元素，此时将rear和front 都指向它**
2. **链队中出队操作：出队前也要先判断链队中是否只包含一个元素，若只有一个元素将q->front 和 q->rear 都置空即可**
3. **链队较顺序队，最大优点是理论上不存在队满的情况**

**双端队列**

**定义：两端都可以进队和出队的队列**

**图解：**

****

**特点：**

1. **在双端队列中，前端进的元素，出队时的位置一定是排在后端进的元素前面，后端进的元素，出队时的位置一定是排在前端进的元素后面，与进队顺序无关**
2. **一端进一端出体现出先进后出的特点(比如前端进前端出或后端进后端出)，就像栈一样**

**一端进另一端出体现出先进先出的特点(比如前端进后端出或后端进前端出)**

1. **在双端队列中出队时，无论是前端出还是后端出，先出的元素排列在后出的元素前面**

**eg:**

**有元素a，b，c，d按顺序进队，能否得到dcab的出队顺序呢？普通的队列一定不能实现，但双端队列能**

**具体操作：**

**a 前端进，b后端进，c前端进，d前端进，再全部从前端出队，便可得到这样的出队序列**

**注意：通过双端队列不能得到任意次序的出队序列**

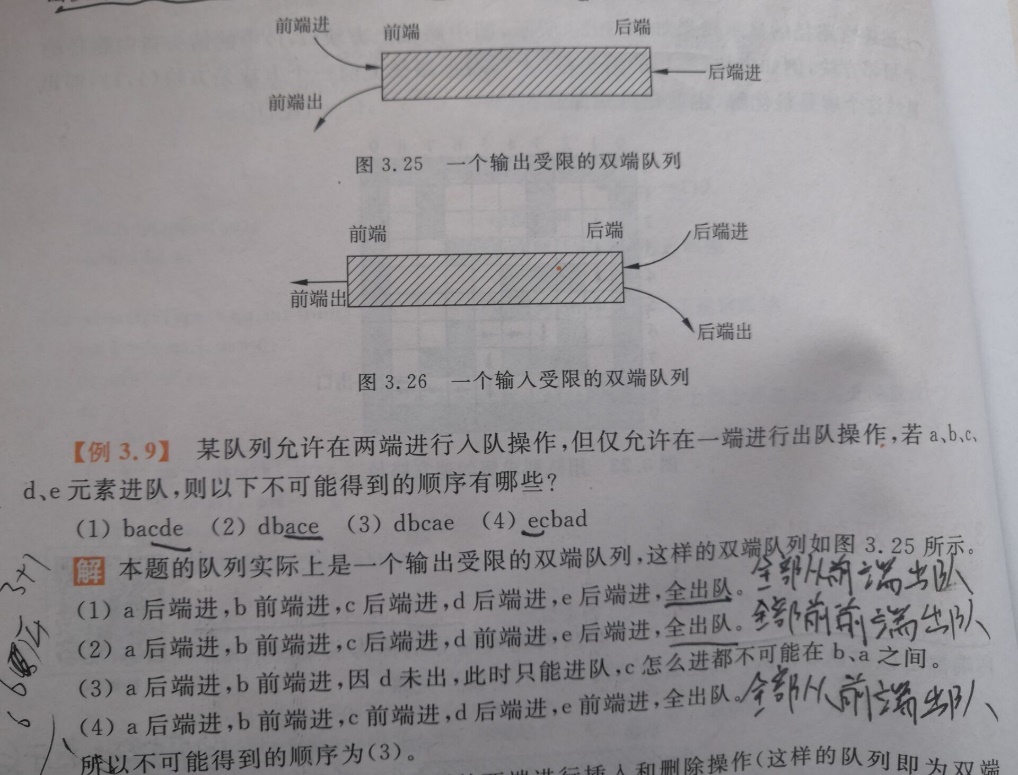
**eg:**

**a，b，c，d依次进队就不能得到dacb这样的出队序列，因为a前端进，b后端进，c无论从哪端进队，出队时都不可能在a和b的中间**

**拓展：**

1. **若规定双端队列某端进只能从该端出，则双端队列就蜕变成两个栈底相邻的栈了**
2. **存在输入受限的双端队列，即只能从一端进队，但能从两端出队**
3. **存在输出受限的双端队列，即能从两端进队，但只能从一端出队**

**eg:**

****

**栈和队列共同之处**

1. **都是在端点处进行插入删除操作**
2. **栈和队列本质上都是线性表(操作受限的线性表)，都是线性结构**

**树(重点)**

**树的定义：**

**由n (n>=0) 个结点组成的有限集合**

**若n = 0，这是一棵空树，为树的特**

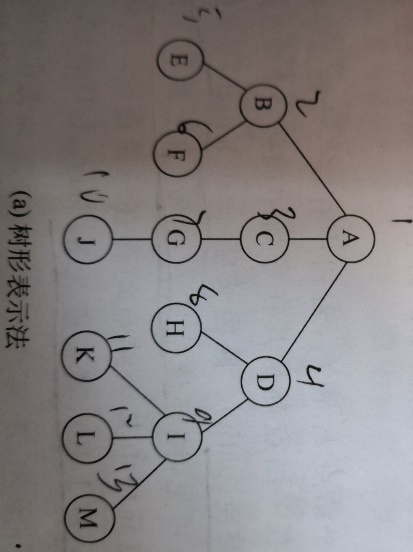
**、、**

**例，一般不允许为空**

**若n>0 ，这样的树，有且只有一个根结点，有多个互不相交的子树，每个子树，也是一棵树**

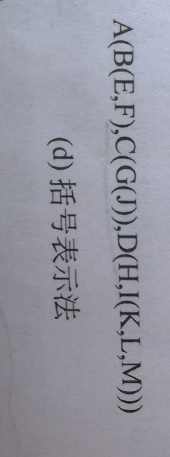
**树的逻辑表示方法(这里介绍4种)**

**树形表示法**

****

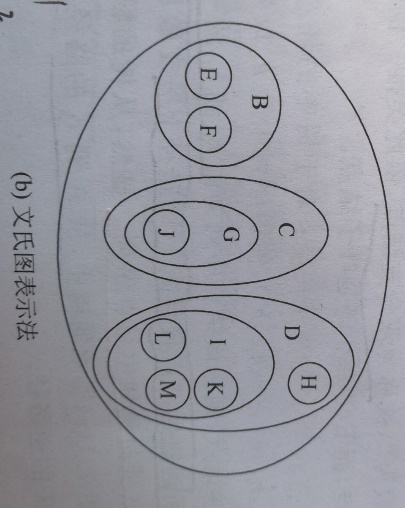
**树形表示法最常用，结点之间通过直线相连，连线上方的结点是下方结点的前驱结点，连线下方的结点是上方结点的后继结点，**

**括号表示法**

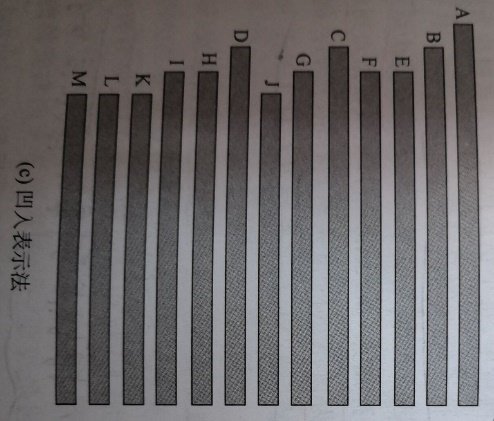
****

**各子树之间用逗号分隔，结点之间的关系用括号表示**

**文氏图表示法**

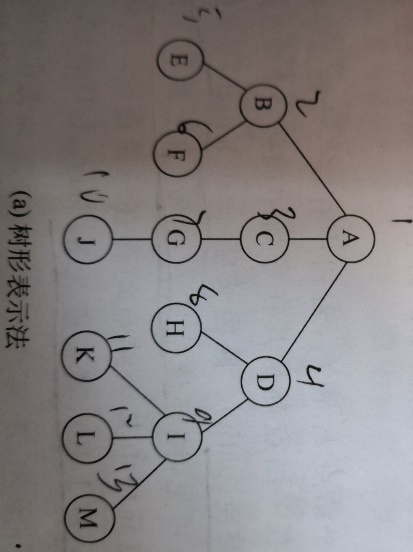
****

**凹入表示法**

****

**树的常用术语**

**以这棵树为例**

****

1. **结点的度：某个结点的子树个数或某个结点包含的分支数，称为该结点的度**

**eg: 上图的结点D，有两个子树(H，I)，该结点的度为2**

1. **树的度：该树中所有结点的度中的最大值为树的度，并且通常称度为m的树，为m次树**

**eg：上图中结点I的度最大为3，所以树的度是3，又称3次树**

1. **分支节点：树中度不为0的结点，每个结点的分支数就是该结点的度，度为 1的结点称为单分支结点，度为 2的结点称为双分支结点，依此类推，分支结点也叫非终端结点**
2. **叶子结点：度为0的结点，又叫终端结点**

**eg：上图中ABCDEFGI为分支结点，HJKLM为叶子结点**

1. **路径：树中任意两结点，如上图的结点A和结点J，存在一条路径能使从A出发到达J，这个路径是 (A，C，G，J)，不难看出A到J的路径实际上就是A到J经过哪些结点，路径就是结点序列**
2. **路径长度：还是以结点A和J为例，A到J的路径上所经过的结点数-1或A到j的路径上所包含的分支数，为结点A到J之间的路径长度，从图看出A到J的路径长度为3**
3. **孩子结点：每个结点的后继结点被称为这个结点的孩子结点**
4. **双亲结点：每个结点的前驱结点被称为这个结点的双亲结点**
5. **兄弟结点：双亲结点相同的结点之间互称兄弟结点**
6. **子孙结点：某个结点子树中包含的所有结点称为该结点的子孙结点**
7. **祖先结点：从根结点到某个结点所经过的所有结点(除自身外)，称为该 结点的祖先结点**

**eg: 以上图为例**

**A是B，C，D的双亲结点；B，C，D是A的孩子结点；**

**B，C，D之间又互为兄弟结点**

**A的子孙结点有C，G，I；L的祖先结点有I，D，A**

1. **结点层次：树中的每个结点都处在一定的层次上，结点的层次是从树根开始定义的，根结点的层次为1，依此类推，结点层次又叫结点深度**
2. **树的高度：树中结点的最大层次称为树的高度，即从根结点到最后一层结点的高度，又叫树的深度**

**eg: 上图中根结点A的层次为1，结点G的层次为3**

**该图中树的高度为4**

1. **森林：n (n>0) 个互不相交的树组成的集合，称为森林**

**eg: 将上图中树的根结点A删除就变成森林，这个森林由3棵互不相交的树组成，分别是B，C，D；**

**若又加上根结点A，森林又变回原来的一棵树**

1. **有序树和无序树：树中每个结点的子树按照次序排列(从左至右)，且每个子树的位置不能交换，这样的树称为有序树，否则称为无序树，没有特别说明，默认都为有序树**

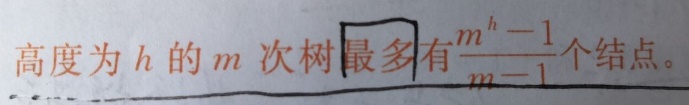
**树的4个重要性质**

1. **树中所有结点的数目等于所有结点的度数之和加1**

**证明：除根结点以外的结点数等于所有结点分支数之和，而所有结点分支数之和恰好等于所有结点的度数之和，所以有树中所有结点的数目等于所有结点的度数之和加1**

1. **度为*m*的树中第*i*层上至多有*mi*-1个结点（*i*≥1）。**

**（3）**

****

**(4)具有*n*个结点的*m*次树的最小高度为**⎡**log*m* (*n*(*m*-1)+1)**⎤

**最大高度为n – m + 1**

**拓展：**

**具有n个结点，度为*m*的树(m次树)的其他重要特性**

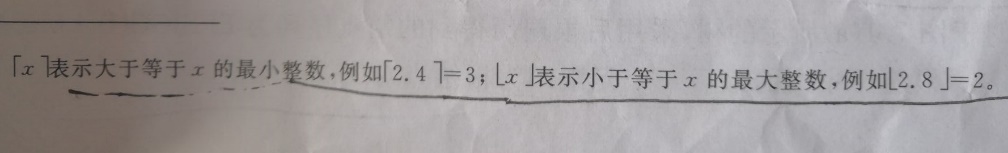
1. ***n*=*n*0+*n*1+…+*nm***

***n*为总结点个数，*ni*为度为*i*（0≤*i*≤*m*）的结点个数**

1. **所有结点度之和 = *n*-1**

***n*为总结点个数**

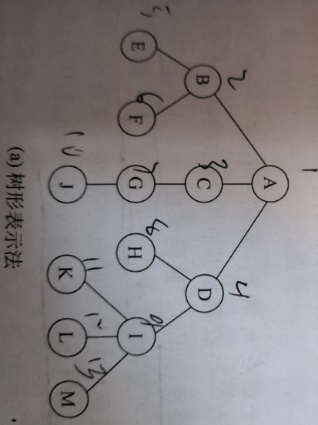
**拓展**

****

**树的遍历**

**定义：按某种遍历方法访问树中的所有结点且每一个结点只被访问一次**

**以这棵树为例**

****

**先根遍历**

1. **先访问根结点**
2. **再按照从左到右的顺序先根遍历根结点的每一棵子树**

**eg: 上图先根遍历得到的结点序列为：ABEFCGJDHIKLM**

**先根遍历所得序列中的第一个元素为根结点的值**

**后根遍历**

1. **先按照从左到右的顺序后根遍历根结点的每一棵子树**
2. **再访问根结点**

**eg: 上图后根遍历得到的结点序列为：EFBJGCHKLMIDA**

**后根遍历所得序列中的最后一个元素为根结点的值**

**层次遍历**

**从根结点开始，自上而下、自左至右访问树中每个结点。**

**即从根结点到最后一层结点，一层一层遍历**

**eg: 上图层次遍历得到的结点序列为：ABCDEFGHIJKLM**

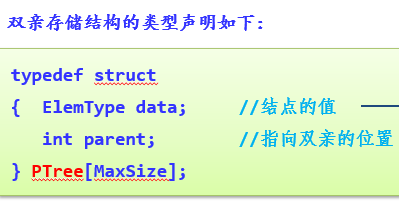
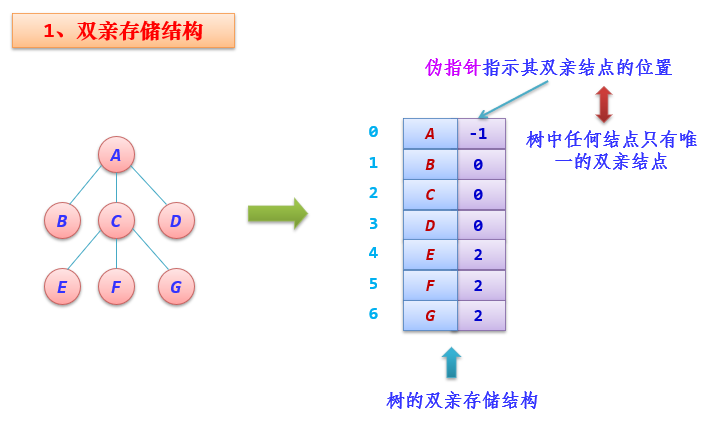
**层次遍历所得序列中的第一个元素为根结点的值**

**树的存储结构**

**树的存储结构有很多种，这里介绍常用的3种**

1. **双亲存储结构**

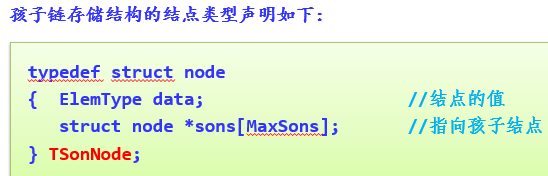
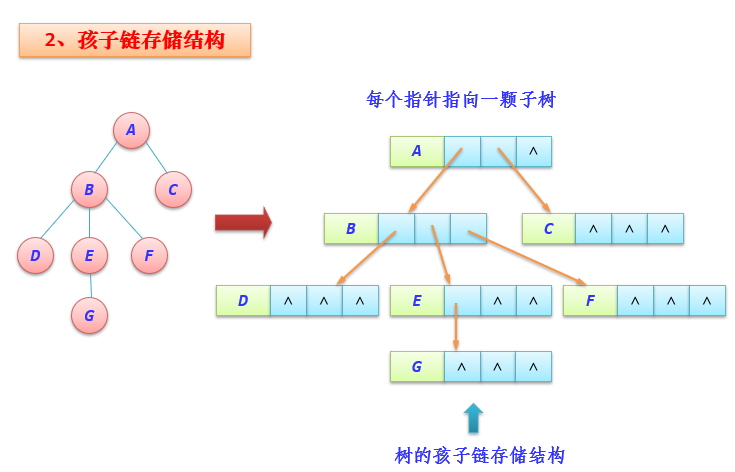
**采用的是顺序存储结构，该存储结构中的结点包含一个伪指针，用来指向该结点的双亲结点**

****

**优点：访问某个结点的双亲结点十分容易**

**缺点：但访问其孩子结点费时，需要遍历整个存储结构**

1. **孩子链式存储结构**

****

**由该树中所有结点的度的最大值来设计该存储结构中指针域的个数**

**即MaxSons是结点度的最大值，或树的度**

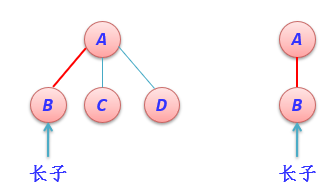
**优点：访问某个结点的孩子结点十分容易**

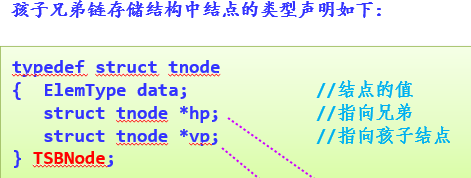
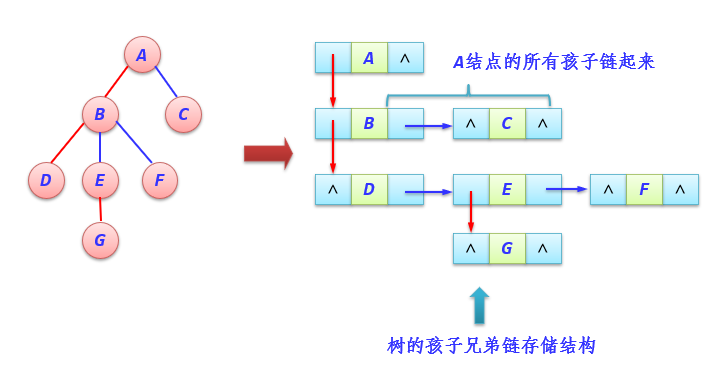
**缺点：**

1. **访问其双亲结点费时**
2. **当树的度较大时，该存储结构中的结点存在较多空指针域**
3. **孩子兄弟链式存储结构**

**该存储结构中的结点有3个域，一个是数据域，两个指针域(两个指针)，左指针域(左指针)，用来存放该结点第一个孩子(长子)的地址(指向它第一个孩子)，右指针域(右指针)存放该结点兄弟结点的地址(指向它兄弟结点)，但根结点比较特殊，它只有孩子结点，没有兄弟结点，所以它的右指针域为空**

**所以孩子兄弟链式存储结构实际上就是把一棵转化成二叉树的存储结构**

****

****

**优点：**

1. **可以方便的实现树和二叉树的相互转换**
2. **访问某个结点的孩子结点和兄弟结点很容易**

**缺点：访问其双亲结点费时，需要遍历整个存储结构**

**拓展：**

1. **包含n(n>2)个结点的树，该树的最大高度为n，即该树的每一层只包含一个结点；该树的最小高度为2，即该树第一层是根结点，其它结点都处于第2层**

**二叉树**

**定义：该树中的任意一个结点最多只能有两个孩子结点，分别为左孩子(左子树)，和右孩子(右子树)，并且左，右子树互不相交，位置也不能更改**

**即二叉树规定了树的度最大只能为2**

**二叉树与树的区分**

1. **二叉树允许为空，树一般不允许，除非特例**
2. **二叉树严格区分左右子树，树没有左右子树这一概念，所以不区分**
3. **二叉树规定了树的度最大只能为2，树没有这种规定**

**满二叉树**

**定义：除了叶子结点，每个结点都有两个孩子结点的二叉树，称为满二叉树**

**若一棵高度为*h*的二叉树恰好有2*h*-1 个结点，这样的二叉树就为满二叉 树**

**满二叉树的特点**

1. **叶子结点都在最底层**
2. **只有度为0和度为2的结点，并且度为0的结点数比度为2的结点数多1**

**完全二叉树**

**定义：实际上就是删除满二叉树最底层(叶子结点层)最右边的连续若干个结点得到的一种二叉树，称为完全二叉树，不难看出来满二叉树就是完全二叉树的特例，其实也属于完全二叉树**

**满二叉树的特点**

1. **叶子结点只可能在最下面两层出现**
2. **度小于2的结点同样也只能在最下面两层出现**
3. **如果有度为1的结点，只能可能有一个，且该结点只有左孩子无右孩子**
4. **按层序编号时，一旦出现编号为i的结点是叶子结点或只有左孩子，则编号大于i的结点均为叶子结点**
5. **当完全二叉树中结点总数n为奇数时，度为1的结点个数为0；当n为偶数时，度为1的结点个数为1**

**二叉树的性质**

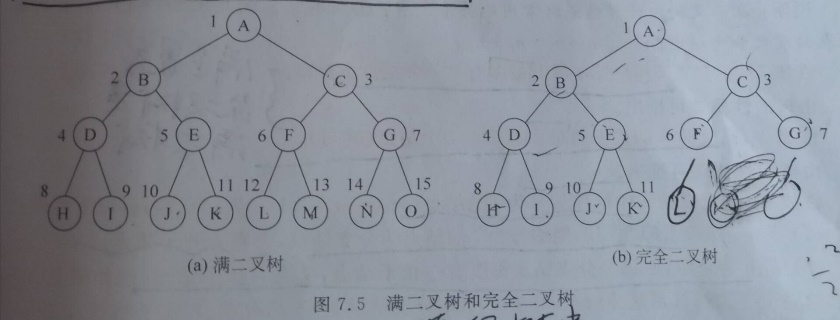
1. **二叉树上叶结点数等于双分支结点数加1。即：*n*0=*n*2+1。**
2. **二叉树上第i层上至多有2*i*-1个结点（i>=1）**
3. **高度为*h*的二叉树至多有2*h*-1个结点（*h*≥1），这样的二叉树为满二叉树**

**完全二叉树的性质**

**完全二叉树中编号为i的结点(1<=i<=n，n是结点数)，由以下性质**

1. **若i<=⎣n/2⎦，则编号为i的结点为分支结点，否则为叶子结点**
2. **除树根结点外，若一个结点的编号为i，则它的双亲结点的编号为⎣i/2⎦**
3. **若编号为i的结点有左孩子结点，则左孩子结点的编号为2i；若编号为i的结点有右孩子结点，则右孩子结点的编号为2i+1**
4. **若n为奇数，则每个分支结点既有左孩子，又有右孩子；若n为偶数，则编号最大的分支结点(即编号为⎣n/2⎦的结点)，只有左孩子，没有右孩子，其余分支结点都有左，右孩子结点**
5. **完全二叉树的高**
6. **度为：**⎡**log2(*n*+1)**⎤ **或者** ⎣**log2*n***⎦**+1**

**图解：**

****

**树转化成二叉树**

**将树中任意一个结点的左指针域(左指针)指向它的第一个孩子，作为它的左子树，右指针域(右指针)指向它兄弟，作为它的右孩子，注意根结点无兄弟，只需将左指针指向它的第一个孩子即可**

**将二叉树还原回树，倒推即可**

**将森林(n棵树组成)转换成二叉树**

**从森林中第一棵树的根结点开始，根结点的左指针域(左指针)指向它的第一个孩子，作为它左子树，右指针域(右指针)指向与它相邻的一棵树的根结点，作为它的右孩子，这棵树的其余结点的左指针指向它的第一个孩子，右指针指向它的兄弟，第二棵也是如此，依此类推，直到森林(n棵树组成)变成一个二叉树**

**注：其实简单点来说，就是先把森林中n棵树转化成n棵二叉树，在按从左到右的顺序将n棵二叉树的根结点连接起来**

**二叉树的顺序存储结构**

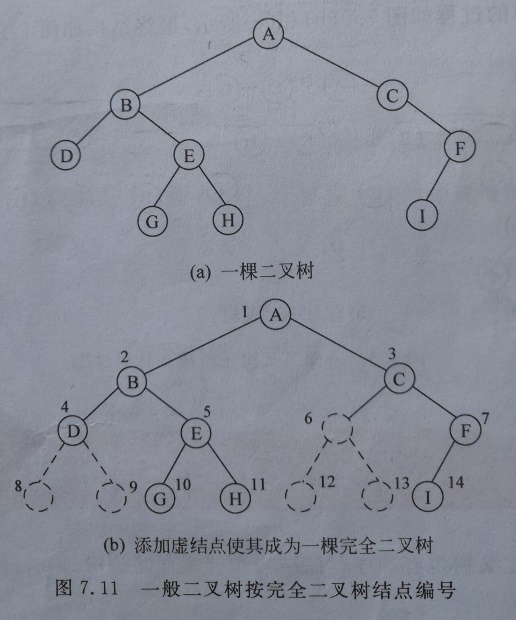
**即采用一组地址连续的存储单元来存放二叉树中的元素，实际就是用数组来存放二叉树中的元素**

**什么的二叉树适合采用顺序存储？**

**完全二叉树和满二叉树适合采用顺序存储结构存储，原因是既能节省存储空间，又能通过数组下标确定完全二叉树或满二叉树中的每个结点在这棵树中对应的层次，以及它与同一棵树中其他结点的关系是什么**

**如果二叉树的树形接近完全二叉树，可以通过构造空结点，使之成为一棵完全二叉树，这里的空结点存放在数组中用“#”来表示**

**图解**

****

**因为完全二叉树或满二叉树，可以通过树的层次将树中每个结点进行编号，在存放到数组对应的下标中，结点的层次编号与存放在数组中的下标一致，即结点编号是几，就把它存放在数组中下标为几的元素中，注意数组下标为0的元素不使用，因为结点是从1开始编号**

**拓展：**

**一般的二叉树(即树形不接近完全二叉树)，仍要采用顺序存储结构存储，会造成存储空间的大量浪费，因为要保证结点之间的关系就要构造大量的空结点，使它成为一棵完全二叉树，在将经二叉树转化而来的完全二叉树存放到数组中，空结点也是占存储空间的，所以会浪费数组中的一些存储单元**

**eg: 当一棵二叉树高度也为h，并且也只有h个结点，即它是一棵右单支树，但要分配2*h*-1个存储单元去存放它，这种情况是最坏的情况**

**所以一般的二叉树，通常采用链式存储结构**

**图**

**定义：无论多么复杂的图都是由顶点和边构成**

**eg：**

**图G是由两个集合V(vertex)，E(edge) 组成，记为G=(V，E)，其中V是图G中所有顶点的有限集合，E是图G中所有顶点之间边的有限集合**

**图的术语**

1. **完全图**

**无向图中，每两个顶点之间都存在着一条边，称为完全无向图， 包含有*n*(*n*-1)/2条边。有向图中，每两个顶点之间都存在着方向相反的两条边，称为完全有向图，包含有*n*(*n*-1)条边。n是顶点数**

1. **路径和路径长度**

**在一个图G中，从顶点*i*到顶点*j*的一条路径(*i*，*i*1，*i*2，…，*im*，*j* )**

**可以看出图的路径和树的路径一样，是个顶点序列**

**路径长度是指顶点i到顶点j的路径上经过的边的数目。**

**若一条路径上除开始点和结束点可以相同外，其余顶点均不相同，则称此路径为简单路径。**

1. **回路或环**

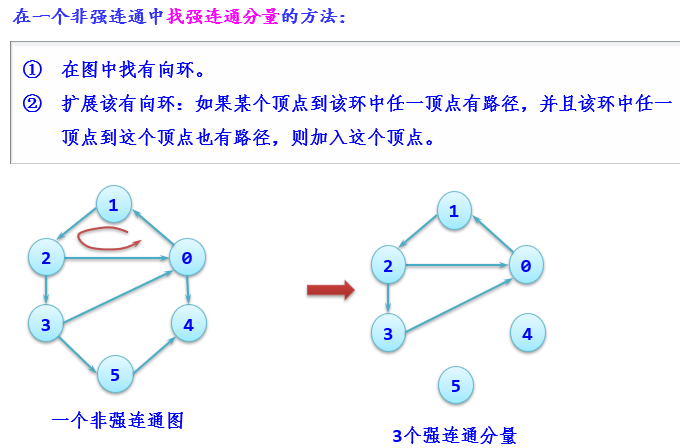
**若一条路径上的开始顶点与结束顶点相同，其余顶点均不同或可能存在相同的情况，此路径被称为回路或环。**

**一条路径上开始点与结束点相同，其余结点不同也被称为简单回路或简单环。**

1. **…………………**

**拓展：**

1. **连通图/强连通图只有一个连通/强连通分量，就是它本身**

****