* 数据结构知识点
* 数据对象、数据结构、存储结构的概念

数据对象是所有能够被输入到计算机中，且被计算机处理的符号的集合。它是计算机程序加工处理的对象。数据对象的基本单位是数据元素，数据元素内部的最小项是数据项。

数据结构是相互之间存在一种或多种特定关系的数据元素的集合。包含三个概念：数据的逻辑结构、数据的物理存储结构和对数据的各种操作。

存储结构是数据结构再计算机中的表示。我们使用二进制位的位串表示数据元素。数据元素之间的关系分为：顺序存储 和 链式（非顺序）存储。

* 顺序存储、链式存储（单链、循环链表、十字链式）等存储结构的特点及作用

顺序存储的特点：

顺序存储时，相邻数据元素的存放地址也相邻（逻辑与物理统一）；要求内存中可用存储单元的地址必须是连续的，并且，顺序表的存储空间需要预先分配。

顺序存储的作用：若线性表的长度变化不大，而且主要操作是查找、修改，则采用此存储结构。

顺序存储的优缺点:

（1） 优点：

1） 随机存取元素容易实现

2） 简单直观

（2） 缺点：

1） 插入或删除结点困难。

2） 扩展不灵活

3） 容易造成浪费

链式存储的特点：链式存储时，相邻数据元素可随意存放，在链表中逻辑上相邻的数据元素，物理存储位置不一定相邻，它使用指针实现元素之间的逻辑关系。并且，链表的存储空间是动态分配的。

链式存储的作用：若线性表长度变化较大，且主要操作是插入、删除，则采用此存储结构。

链式存储的优缺点：

（1） 优点：

1） 插入、删除运算方便

（2） 缺点：

1） 要占用额外的存储空间存储元素之间的关系，存储密度降低

2） 链表不是一种随机存储结构，不能随机存取元素，查找麻烦

* 稀疏矩阵/特殊矩阵压缩存储的特点

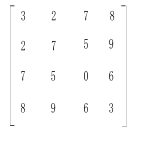
（1） 对有规律的元素和值相同的元素只分配一个存储单元，对于零元素不分配空间。

（2） 相较于使用多维数组存储，矩阵压缩存储可以节省更多存储空间，减少使用不必要的存储空间。

（3） 使用矩阵压缩存储之后，线性表失去了原有的随机存取功能。

* 矩阵压缩存储地址的计算

以下图为例：

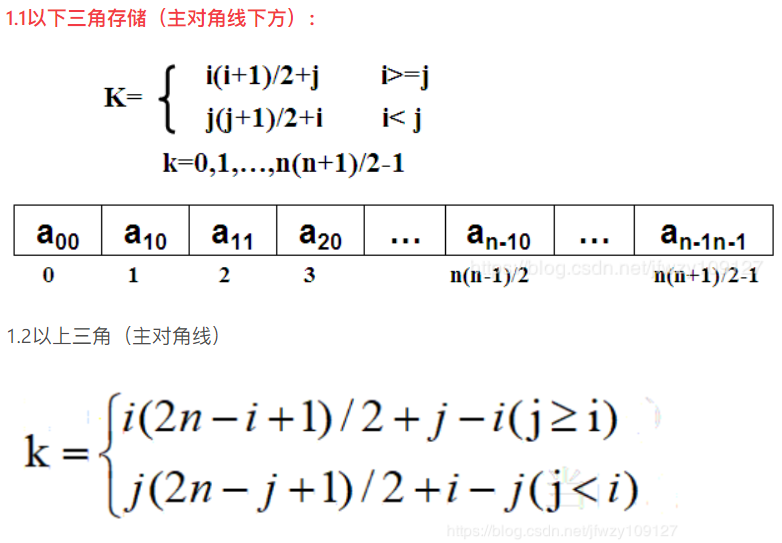


在M[n(n+1)/2]数组中存储为：

下标 ：00 01 11 20 21 22 30 31 32 33

值 ： 3 2 7 7 5 0 8 9 6 3

计算公式如下：



* 循环队列的队头队尾与存储空间的关系

非循环队列使用sq.rear == MAX\_SIZE来判断队列是否为满，但是这样通常会产生假溢出情况。

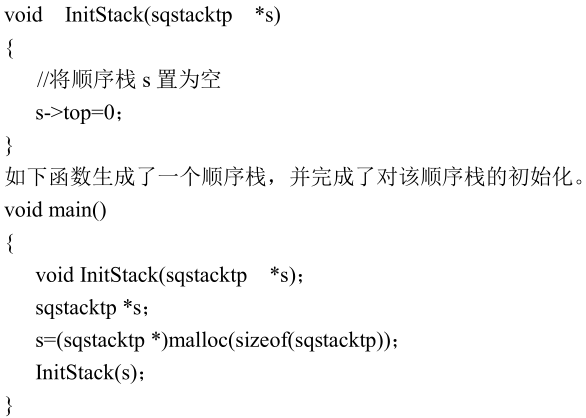
我们可以使用循环队列来存储队列元素，设想sq.data[0]接在sq.data[MAX\_SIZE - 1]之后。但这样又会产生新的问题，即队列空和队列满都是判断条件 sq.front == sq.rear。

所以我们通常约定队头指针指示的位置不用来存放元素，只标志队列的开始。这样，队满的条件为：(sq.rear + 1) % MAX\_SIZE == sq.front。队空的条件为：sq.rear == sq.front.

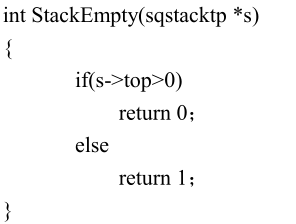
* 栈和队列操作（已知输入输出序列，写出操作过程/空间大小）

顺序栈：

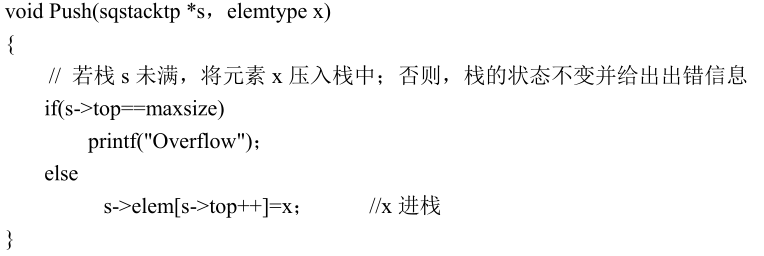
1. 栈的初始化



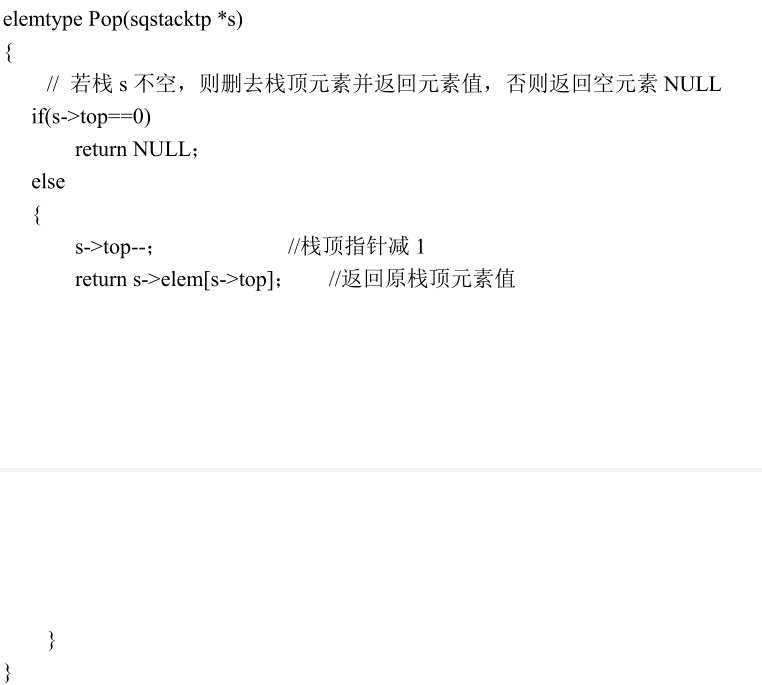
1. 判断栈空



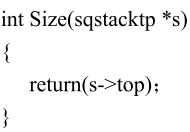
1. 压栈



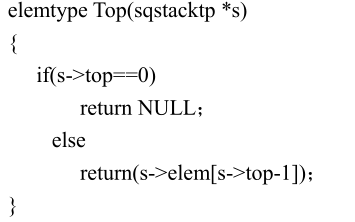
1. 出栈



1. 求栈深

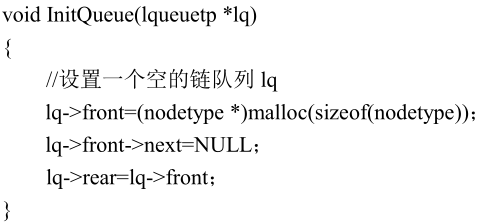


1. 读取栈顶元素

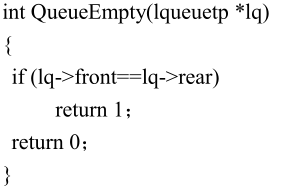


链队列：

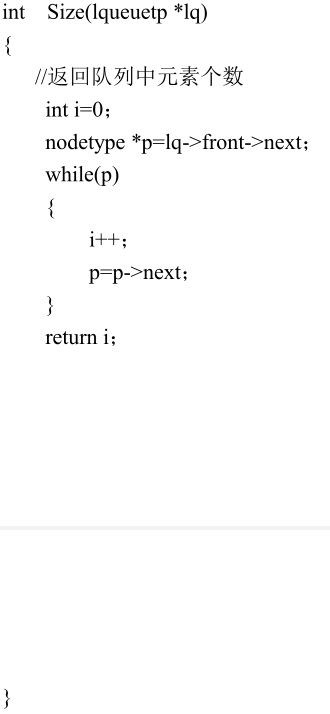
1. 初始化



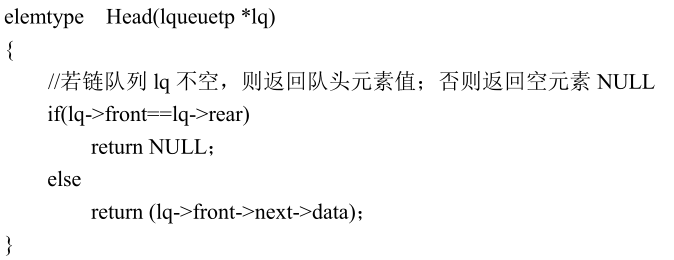
1. 判断队列为空



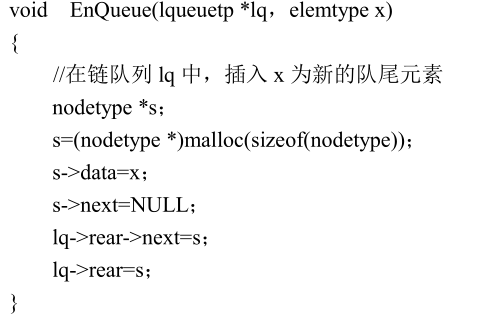
1. 求队列长度



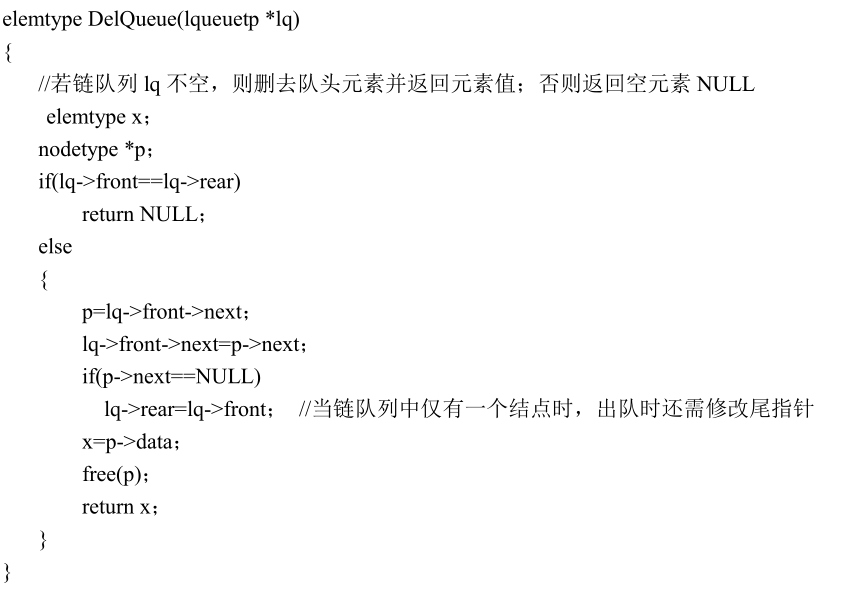
1. 读取队头



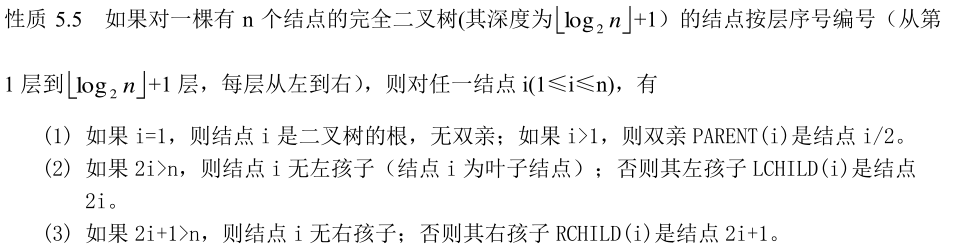
1. 入队



1. 出队



* 二叉树的性质以及二叉树的性质（1-5性质）的应用

1. 在二叉树的第 i 层上至多有 2^（i-1） 个结点(i≥1)。
2. 深度为 k 的二叉树至多有 2^k - 1 个结点(k≥1)。
3. 对任何一颗二叉树 T，如果其终端结点(叶子结点)数为 n0，度为 2 的结点数为 n2，则 n0=n2+1。
4. 
5. 

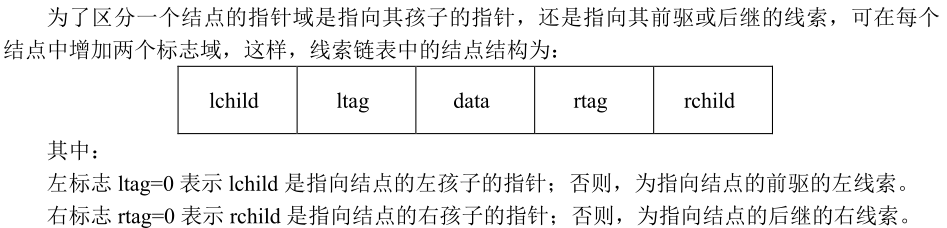
例题：

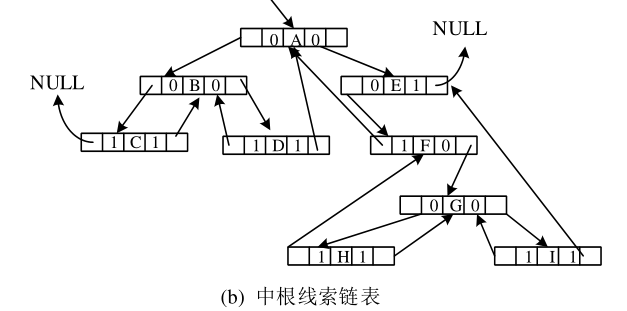
1. 对于一棵具有n个结点的树，树中所有度数之和为：n - 1
2. 一棵具有124个叶子结点的完全二叉树，最多有 248 个结点。（度为2 的 123个，度为 1 的 1个，度为 0 的124个）
3. 深度为5的二叉树至多有 31 个结点。
4. 对于一个满二叉树，m个叶子，n个结点，深度为h，则 n = 2 \* h – 1
5. 深度为k的完全二叉树至少有2^(k-1)个结点，至多有2^k – 1个结点，若从上到下，从左到右的次序给结点编号（从1开始），则编号最小的叶子结点的编号是2^(k-2) + 1
6. 具有n个结点的完全二叉树的深度为：floor(以2为底n的对数) + 1

* 二叉树线索化含义

线索化的实质：将二叉链表中的空指针改为指向前驱或后继的线索。由于前驱和后继信息只有在遍历该二叉树时才能得到，所以，线索化的过程就是在遍历的过程中修改空指针的过程。

线索二叉树的作用：将二叉树线索化后，减少空指针数量，赋予二叉树新的意义。如：在中序线索化后，可以用以近队列存储的方式访问二叉树，加快访问速度和查找效率。



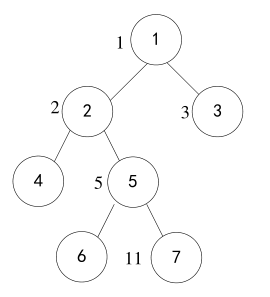


二叉树的线索化的含义：

将二叉树转化成线索二叉树的过程称为线索化。

按照某种次序将二叉树线索化，只要按照该次序遍历二叉树，在遍历过程中用线索取代空指针即可。为此，附加一个指针 pre 始终指向刚访问过的结点，而指针 p 指向当前正在访问的结点。显然结点\*pre 是结点\*p 的前驱，而\*p 是\*pre 的后继。

* 已知二叉树的前/中/后序遍历结果中的二个，写出另一个结果



先根序：

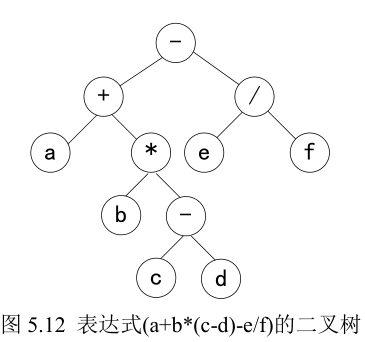
1 2 4 5 6 7 3

后根序：

4 6 7 5 2 3 1

中根序：

4 2 6 5 7 1 3



先根序：

* + a \* b – c d / e f

中根序：

a + b \* c – d – e / f

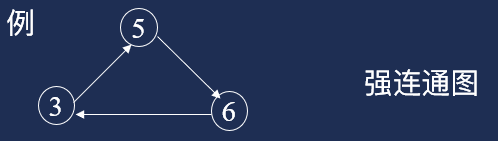
后根序：

a b c d - \* + e f / -

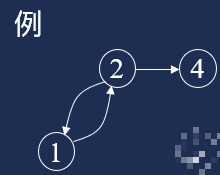
* 有向图中强连通、出入度等有关概念

强连通有向图：

在有向图G中，如果对于每一对<vi，vj>∈V，vi!=vj，从vi到vj和从vj到vi都存在路径，则称G是强连通图。



<3, 5> : 3->5 <3, 6> : 3->5->6 <5, 6> : 5->6

 非连通图，4无法到1

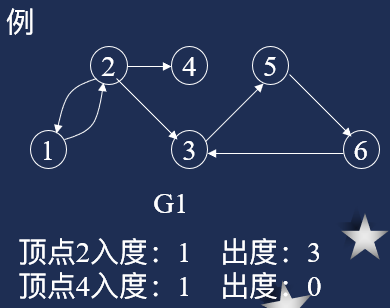
入度：

有向图中，以顶点v为头的弧的数目称为v的入度，记为ID(V)；

出度：

有向图中，以顶点v为尾的弧的数目称为v的出度，记为OD(v)；

**顶点v的度为： TD(v)=ID(v)+OD(v)**

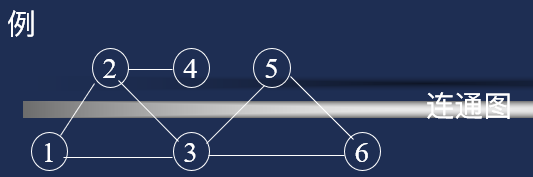


* 无向连通图、连通分量、完全图的概念

无向连通图：如果对于图中任意两个顶点vi，vj∈V，vi，vj都是连通的，则称G是连通图。

连通分量：指的是无向图中的极大连通子图。

连通图：



连通分量：



* 二分（折半）查找法的查找成功所需的平均比较次数ASL/某个元素的比较次数

折半查找中元素的比较次数及判断数

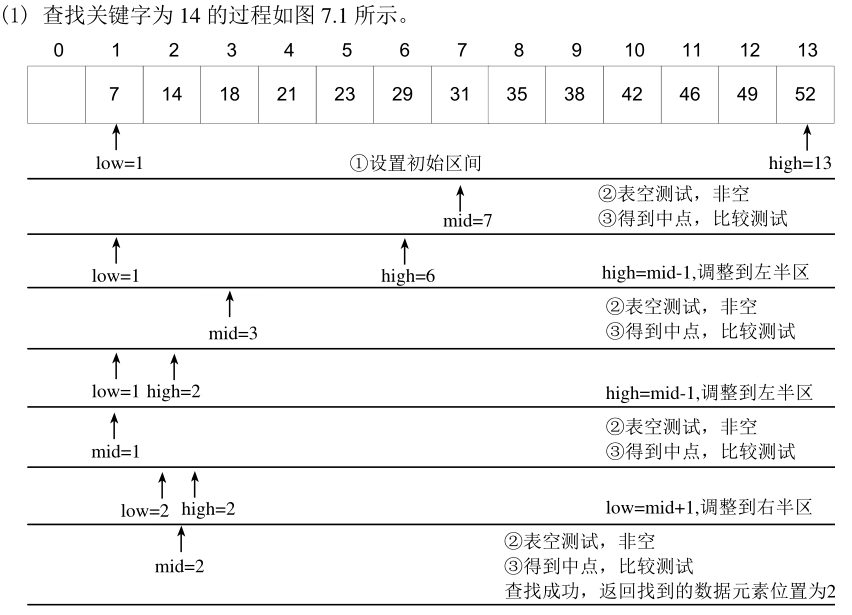
二分查找的前提条件：

有序表——表中数据元素按关键字升序或降序排列。

二分查找的主要思想：

每次将待查记录所在区间缩小一半。

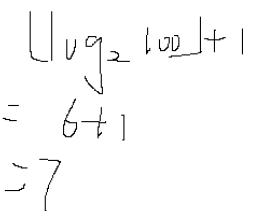
例如：



折半查找的元素比较次数最多不超过判定树的高度，即floor(以2为底n的对数) + 1.

例如：

对100个元素进行折半查找，在查找成功的情况下，比较次数最多是：7

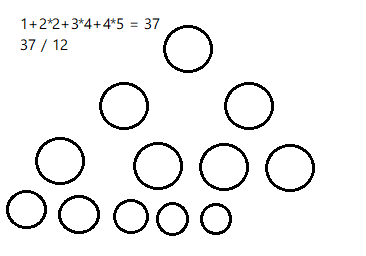


顺序查找长度为n的线性表时，每个元素的平均查找长度为：(n + 1) /2，时间复杂度为：O(n)

折半查找长度为n的线性表时，每个元素的平均查找长度为：O(以2为底n的对数)

例如：

有一个长度为12的有序表，按照二分查找法对该表进行查找，在表内各元素等概率情况下，查找成功所需的平均比较次数为：37 / 12



* 算法分析的意义

（1） 算法分析可以对一个算法需要多少计算时间和存储空间作定量的分析。

（2） 可以让我们对当前算法所需要的计算时间和存储空间进行大致的判断，根据实际情况牺牲更多空间节约时间，牺牲更多时间节省空间。

（3） 对于一个实际问题的解决，我们可以提出若干种算法，进行算法分析可以帮助我们选择执行时间更短、占用存储空间更少的更优算法。

* 哈夫曼树的定义、含义

已知权值，构造哈夫曼树、带权路径长度WPL、哈夫曼编码。

哈夫曼树的定义：

假设有 n 个权值 W1，W2，……，Wn ，试构成一棵有 n 个叶子结点的二叉树，每个叶子结点权值为 Wi，则其中带权树路径长度 WPL 最小的二叉树称作哈夫曼树（或最优二叉树）。

哈夫曼算法：

(1) 根据给定的 n 个权值：W1，W2，……，Wn，构成 n 棵二叉树的集合 F＝{T1，T2，……，Tn}，其中每棵二叉树 Ti 中只有一个权为 Wi 的根结点，其左、右子树均为空。

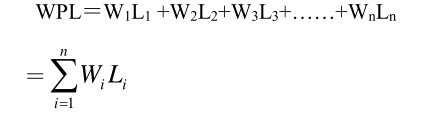
(2) 在 F 中任选两棵根结点的权值最小的树作为左右子树，构成一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

(3) 从 F 中删除这两棵树，同时将新得到的二叉树加入到 F 中。

(4) 重复⑵和⑶步，直到 F 中只含一棵树为止。这棵树便是哈夫曼树。

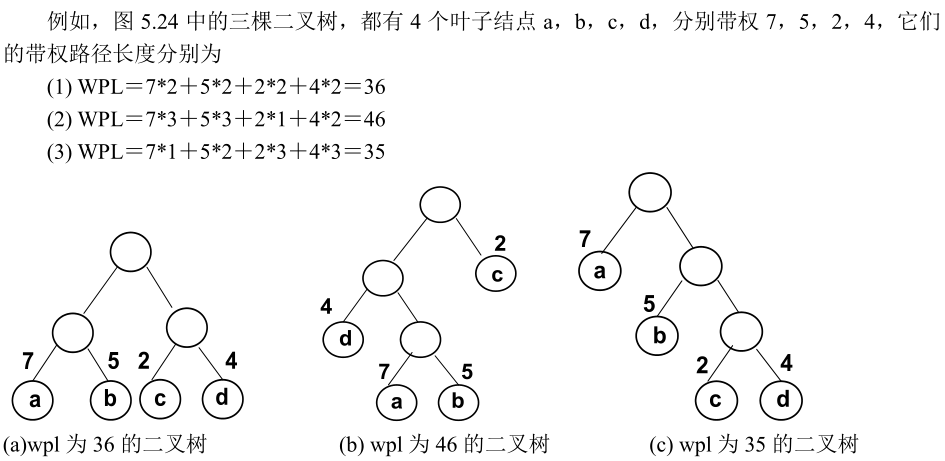
WPL:

树的带权路径长度：为树中所有叶子结点的带权路径长度之和，通常记作：



其中，n 为二叉树的叶子结点的个数，Wi 为第 i 个叶子结点的权值，Li 为从根结点到第 i 个叶子结点的路径长度。

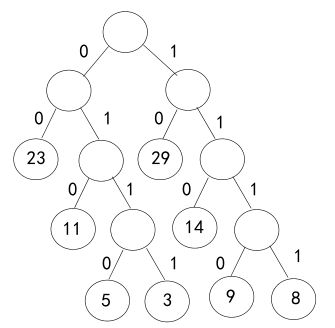
例：



例题：

8种字符（A B C D E F G H）,对应8种频率（5 29 8 9 14 23 3 11）

画出哈夫曼树，写出各字符的哈夫曼编码。



左0右1依次根据遍历写出编码即可。

在F中选出两个最小的结点，合并成一个树，删除这两个结点，再将这棵树的根结点放到F中。如此循环往复。

* 一维/二维数组元素存储地址计算

C语言采用以行序为主序的存储方式。

假设数据有n列 m行，每个数组元素占用s个存储单元，设元素Aij在存储器中的地址为LOC(i,j)，A00的存储地址是LOC(0,0)，即二维数组存储的起始位置，也叫首地址或基地址。Aij以上有i行数据，Aij以前有j个数据，所以它前面共存放了(i\*n+j)个元素，则可得公式：

LOC(i，j) = LOC(0,0) + (n\*i+j) \* s

* 二路归并排序每一趟的结果

二路归并排序的主要思想：

二路归并排序主旨是“分解”与“归并”

分解：

1.将一个数组分成两个数组，分别对两个数组进行排序。

2.循环第一步，直到划分出来的“小数组”只包含一个元素，只有一个元素的数组默认为已经排好序。

归并：

1.将两个有序的数组合并到一个大的数组中。

2.从最小的只包含一个元素的数组开始两两合并。此时，合并好的数组也是有序的。

举例说明二路归并排序算法过程：

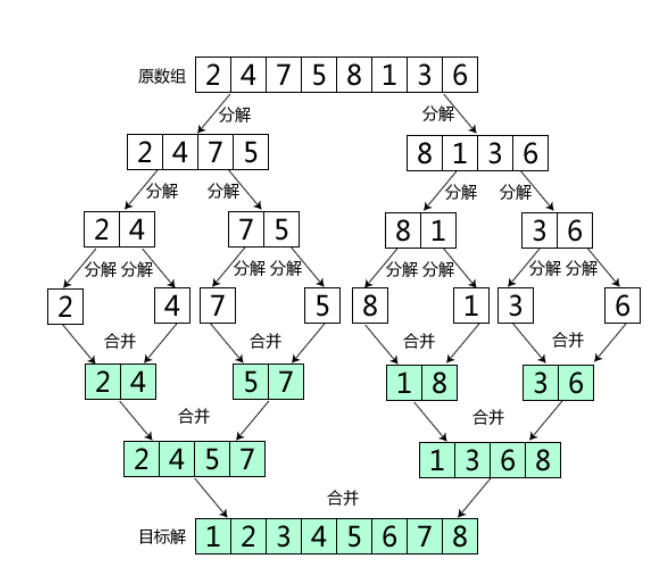
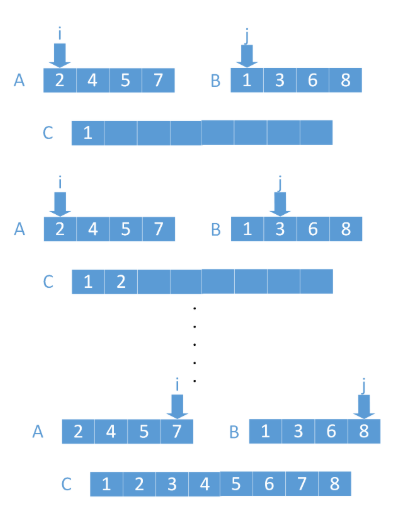
1.图中原始数组为{2,4,7,5,8,1,3,6}，数组中元素的个数为8个。首先将8个元素的数组二分，每次分解后，数组中元素的数目为原数组的一半。直到分解为只含有一个元素的数组。

2.将小的数组按序合并，每次合并后数组的大小为上层数组的一倍。此时数组中的元素都是按序排列的。

3.在合并两个有序数组。如右图。

（1） 合并时，有两个指针分别指向有序数组A和B的起始元素，比较指针所指元素的大小，如果A[i]较小，则将A[i]存入数组C中，并且将i后移。循环比较i和j所指的元素。

（2）当一个数组A的元素全部排之后，数组B中的指针就并没有指向B的末尾位置，将B中剩余元素全部存入到C中。

* 二叉树和树的遍历结果、树与二叉树的转换。

先根序遍历（与对应的二叉树的先根遍历序列一致）：

若树非空，则：

访问根结点

依次先根遍历根的各个子树

后根序遍历（与对应的二叉树的中根遍历序列一致）：

若树非空，则：

依次后根遍历根的各个子树

访问根结点

层次遍历：

若树非空，访问根结点。

若第1，…i(i≥1)层结点已被访问，且第i+1层结点尚未访问，则从左到右依次访问第i+1层。

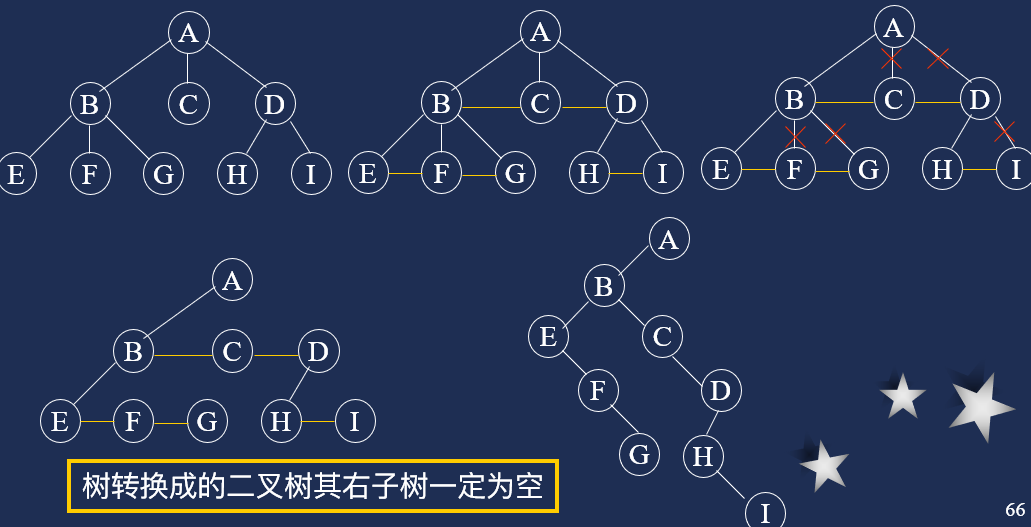
一般树 与 二叉树的转换

1. 一般树转成二叉树

加线：在兄弟之间加一连线

抹线：对每个结点，除了其第一孩子外，去除其与其余孩子之间的关系

旋转：以树的根结点为轴心，将整树顺时针转45°

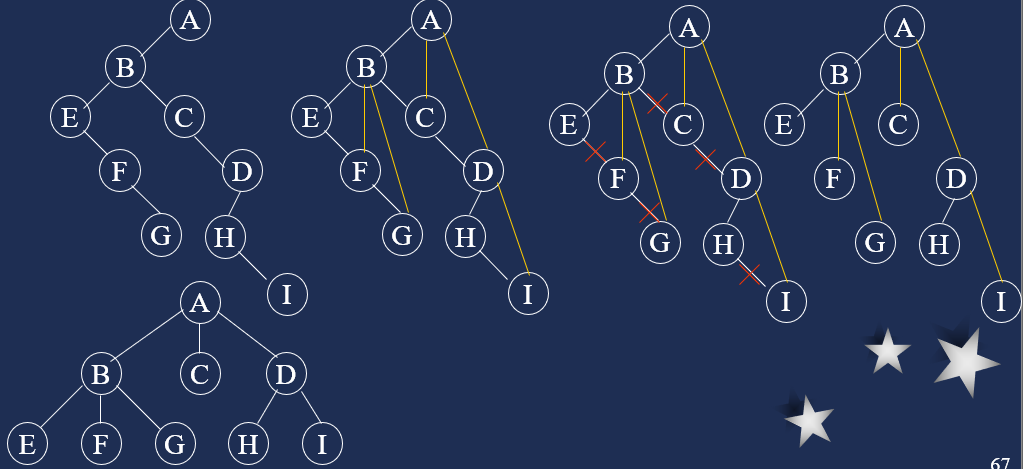


1. 二叉树转成一般树

加线：若p结点是双亲结点的左孩子，则将p的右孩子，右孩子的右孩子，……沿分支找到的所有右孩子，都与p的双亲用线连起来

抹线：抹掉原二叉树中双亲与右孩子之间的连线

调整：将结点按层次排列，形成树结构



* 二叉排序树（BST树）的建立、平衡二叉树（AVL树）的判断。

二叉排序树：

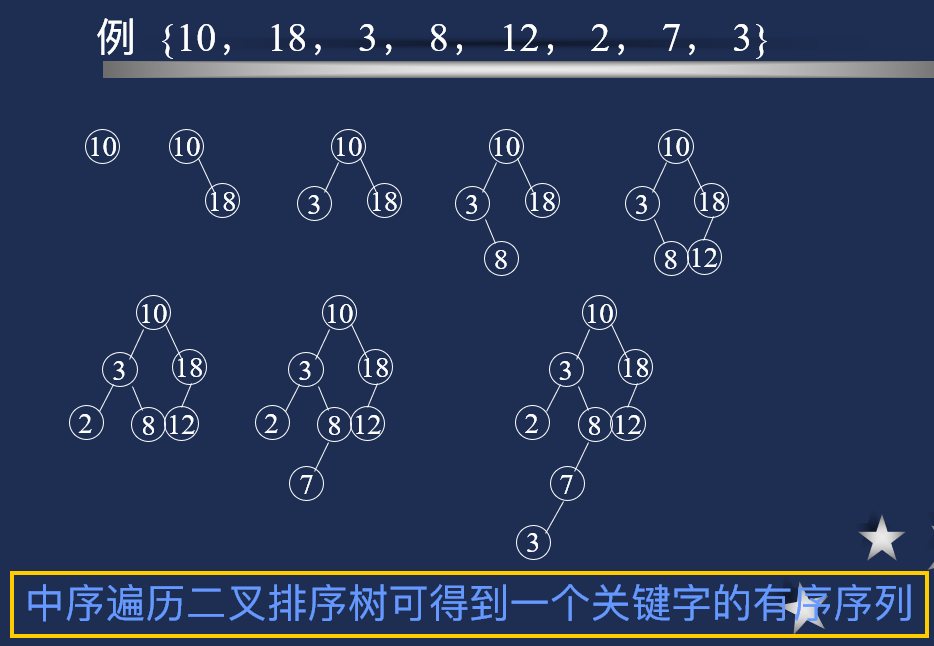
**二叉排序树（Binary Sort Tree）**或者是一棵空树；或者是具有下列性质的二叉树：

⑴ 若左子树不空，则左子树上所有结点的值均小于根结点的值；

若右子树不空，则右子树上所有结点的值均大于根结点的值。

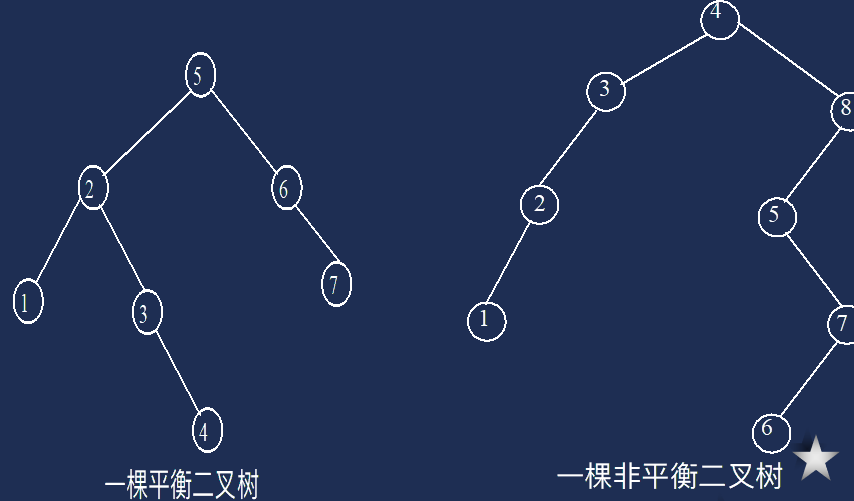
⑵ 左右子树也都是二叉排序树。

创建二叉排序树的过程如下：



二叉排序树：

平衡二叉树或者是一棵空树，或者是具有下列性质的二叉排序树：它的左子树和右子树都是平衡二叉树，且左子树和右子树高度之差的绝对值（平衡因子）不超过1。



如果根的层次是1，则n层结点的平衡二叉树（AVL树）至少有F(h + 2) – 1

这个F（）函数为斐波那契数列。

1 1 2 3 5 8 13 21 34 55……

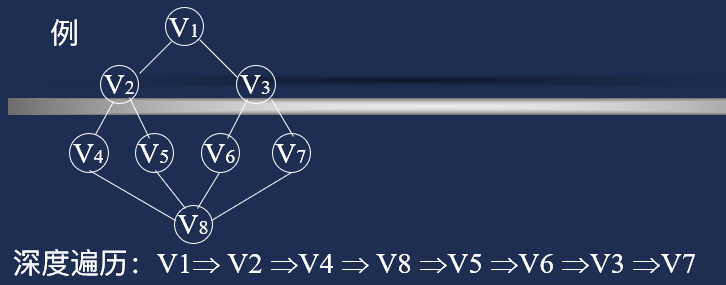
* 已知有向图的存储结构，写出图的遍历结果；拓扑排序。

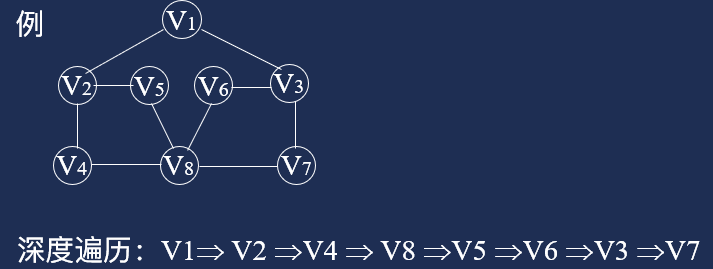
深度优先遍历（depth first search/DFS）：

假设初始状态是图中所有顶点都没被访问过，则DFS可从图中某一顶点v0出发，访问v0，然后去访问与邻接但未被访问过的任一顶点v1，之后再去访问与v1邻接但未被访问过的任一顶点v2时，重复这一过程，当达到一个所有邻接顶点被访问过的顶点时，则依次退回到最近被访问过的顶点，若它还有邻接点未被访问过，从这些被访问过的顶点中，取其中的一顶点开始重复这一过程；若所有邻接顶点被访问过，则依次回退，...， 直到所有顶点被访问过为止。

简单地说就是，一条路走到黑，碰到墙就回到上一个岔路口看看有没有路还没走，如果有就继续走下去，如果没有就再回到上一个岔路口，如此重复，直到全部走完所有结点。

例子：

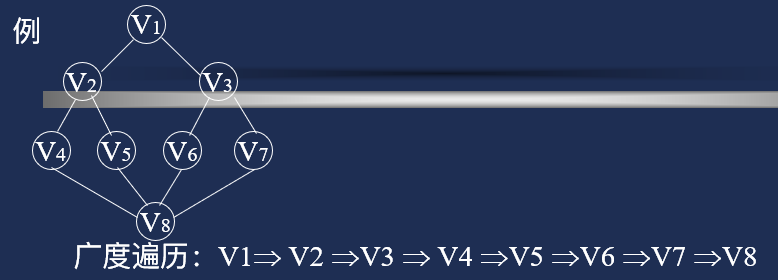


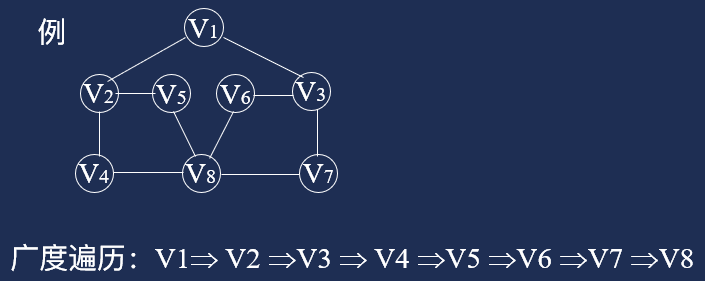


广度优先遍历（breadth-first search/BFS）：

假设初始状态是图中所有顶点都没被访问过，则BFS可以从图中某一指定顶点v0出发，访问v0，然后访问v0的全部邻接点w1，w2，...，wt，再依次访问与w1，w2，...，wt邻接的全部邻接点( 已被访问的顶点除外)，再从这些被访问的顶点出发，逐次访问它们的邻接点(已被访问的顶点除外)。依此类推，直到所有顶点都被访问为止。

例子：





拓扑（pu）排序 与 AOV网（activity on vertex network/活动依附于结点组成的网）：

AOV：用顶点表示活动，用弧表示活动间优先关系的有向图称为顶点表示活动的网，简称AOV网. 若<vi,vj>是图中有向边，则vi是vj的直接前驱；vj是vi的直接后继。AOV网中不允许有回路。前面的结点始终是指向后面的结点的。

拓扑排序：把AOV网络中各顶点按照它们相互之间的优先关系排列成一个线性序列的过程叫拓扑排序。

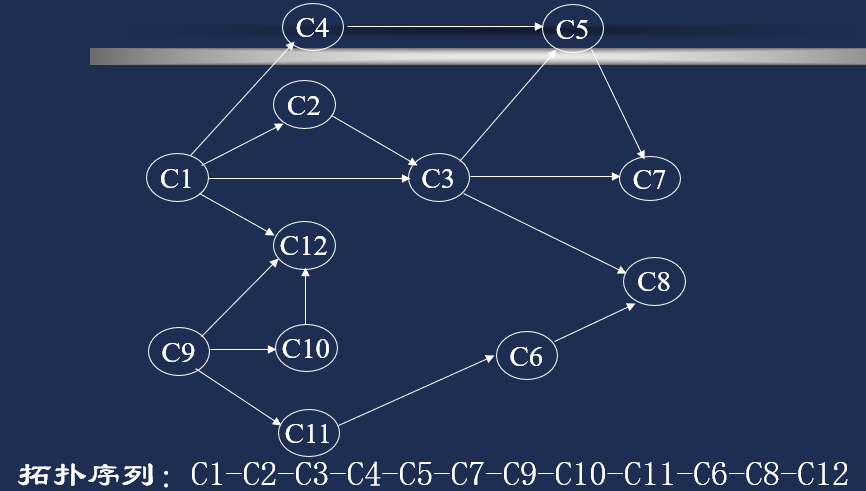
拓扑排序方法：

1. 在有向图中选一个没有前驱的顶点且输出之
2. 从图中删除该顶点和所有以它为尾的弧
3. 重复上述两步，直至全部顶点均已输出；或者当图中不存在无前驱的顶点为止（即： AOV网中存在环）

其中，没有前驱的顶点===入度为零的顶点

删除顶点及以它为尾的弧===弧头顶点的入度减1

拓扑排序例子：



拓扑排序不是唯一的，我们一般按照增序对其进行排序。

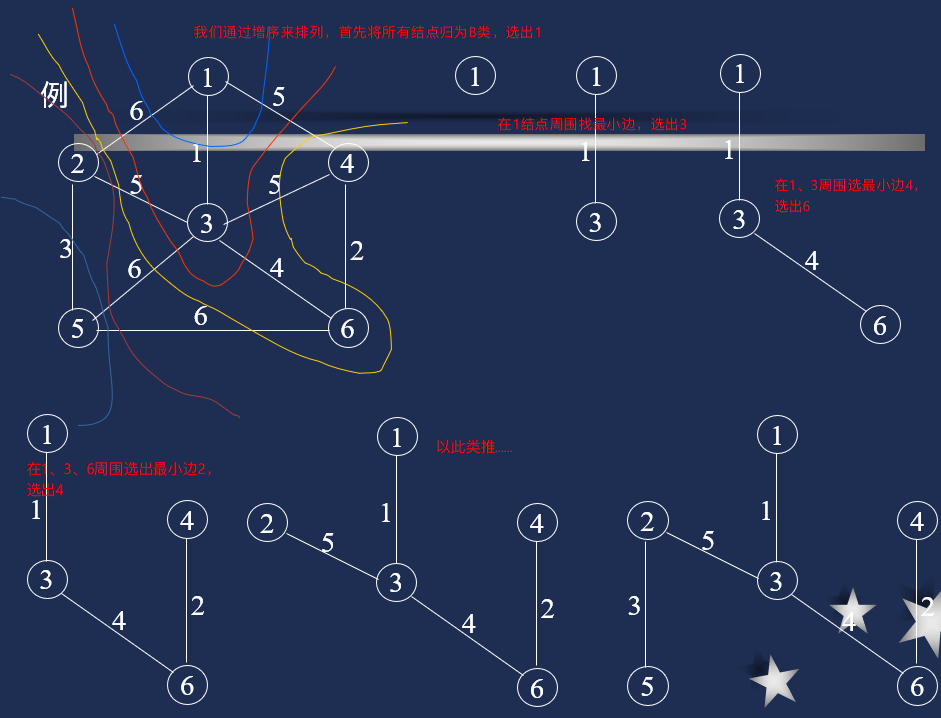
* 已知一个无向带权网络，用克鲁斯卡尔(Kruskal)算法/Prim算法生成其最小生成树的每一步骤。

普利姆算法（Prim）：

普里姆算法在找最小生成树时，将顶点分为两类，一类是在查找的过程中已经包含在树中的（假设为 A 类），剩下的是另一类（假设为 B 类）。

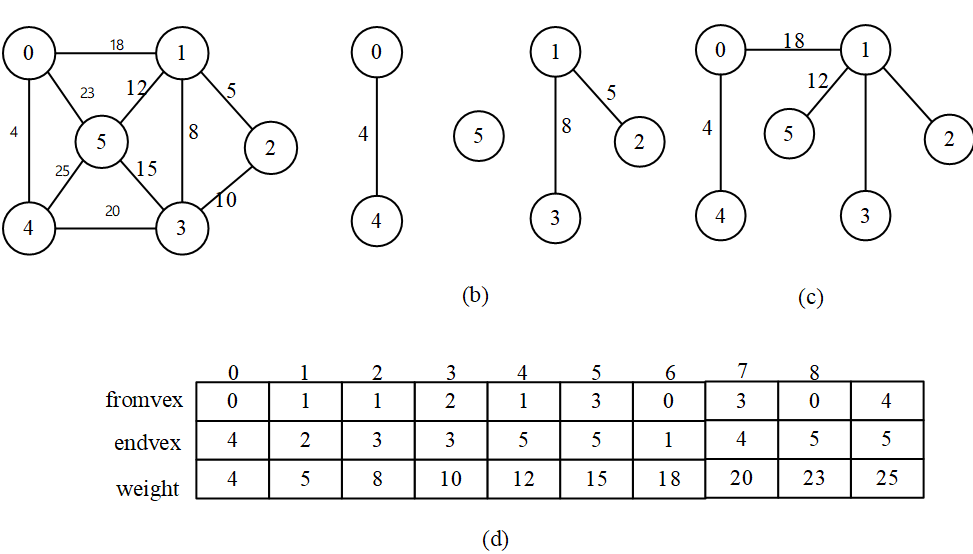
对于给定的连通网，起始状态全部顶点都归为 B 类。在找最小生成树时，选定任意一个顶点作为起始点，并将之从 B 类移至 A 类；然后找出 B 类中到 A 类中的顶点之间权值最小的顶点，将之从 B 类移至 A 类，如此重复，直到 B 类中没有顶点为止。所走过的顶点和边就是该连通图的最小生成树。

我们常用邻接表来作为Prim算法的数据结构。



克鲁斯卡尔算法（Kruskal）：

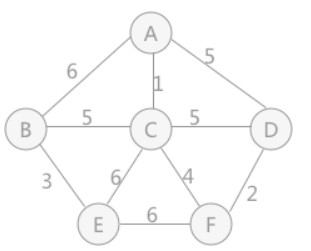
将所有边按照权值的大小进行升序排序，然后从小到大一一判断，条件为：如果这个边不会与之前选择的所有边组成回路，就可以作为最小生成树的一部分；反之，舍去。直到具有 n 个顶点的连通网筛选出来 n-1 条边为止。筛选出来的边和所有的顶点构成此连通网的最小生成树。

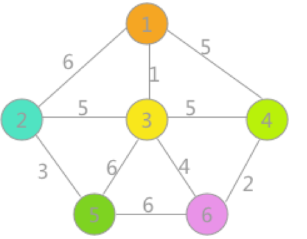
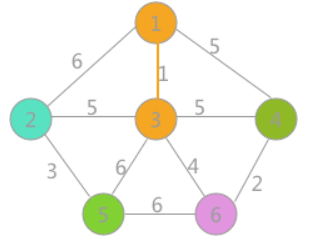
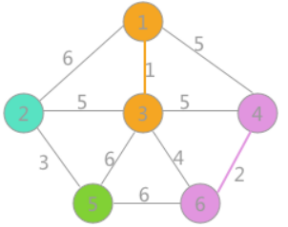


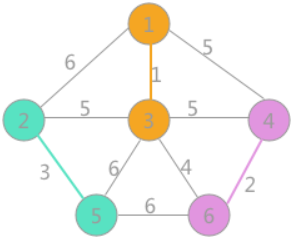
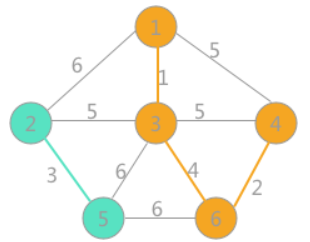
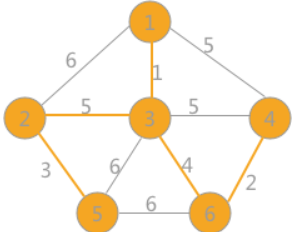
我们使用边集数组来存作为Kruskal算法的数据结构，最好对边集数组按照权重weight来排好序。

再例：

使用克鲁斯卡尔算法构建最小生成树。



123 

45 6 

* 已知数据序列，写出复杂排序（快速排序、堆排序、SHELL排序）的每一步结果。

快速排序

快速排序主要思想：

通过一趟排序，将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录进行排序，以达到整个序列有序

具体描述如下：

从数列中挑出一个元素，称为 “基准”（pivot）；

重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；

递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

堆排序

堆排序主要思想：

1.首先将待排序的数组构造成一个大根堆，此时，整个数组的最大值就是堆结构的顶端

2.将顶端的数与末尾的数交换，此时，末尾的数为最大值，剩余待排序数组个数为n-1

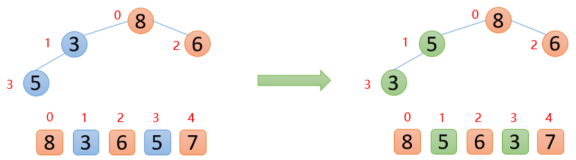
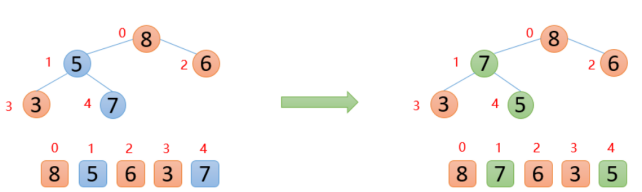
3.将剩余的n-1个数再构造成大根堆，再将顶端数与n-1位置的数交换，如此反复执行，便能得到有序数组

例子:

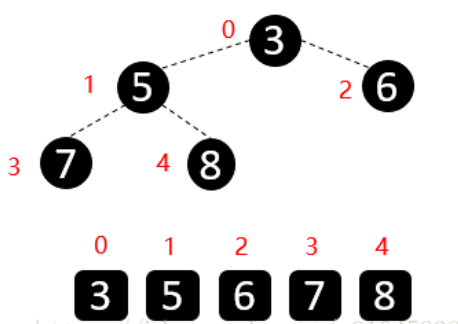
假设存在下列数组：



构造堆

12  3 4

开始排序

5 6 7 

希尔排序

希尔排序主要思想：

选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

按增量序列个数k，对序列进行k 趟排序；

每趟排序，根据00对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

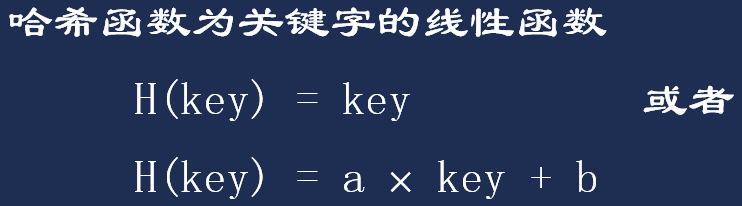
例子：



* 已知数据类型，编写函数完成顺序表操作（插入、删除、遍历等）（也可以伪代码）
* 已知数据类型，编写函数完成单链表、单循环链表的操作（插入、删除、遍历等）（也可以伪代码）
* 散列查找方法中常用散列函数、解决冲突的方法、影响散列方法ASL的因素、散列表的建立、求ASL等。

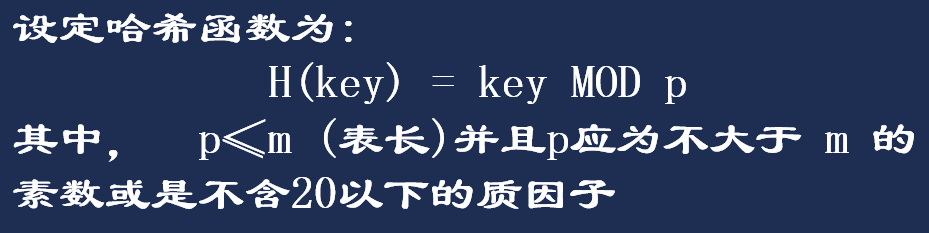
我们常用的散列函数是：直接定址法 和 除留余数法。

直接定址法:



对于较大的关键字集合不适用。

除留余数法：



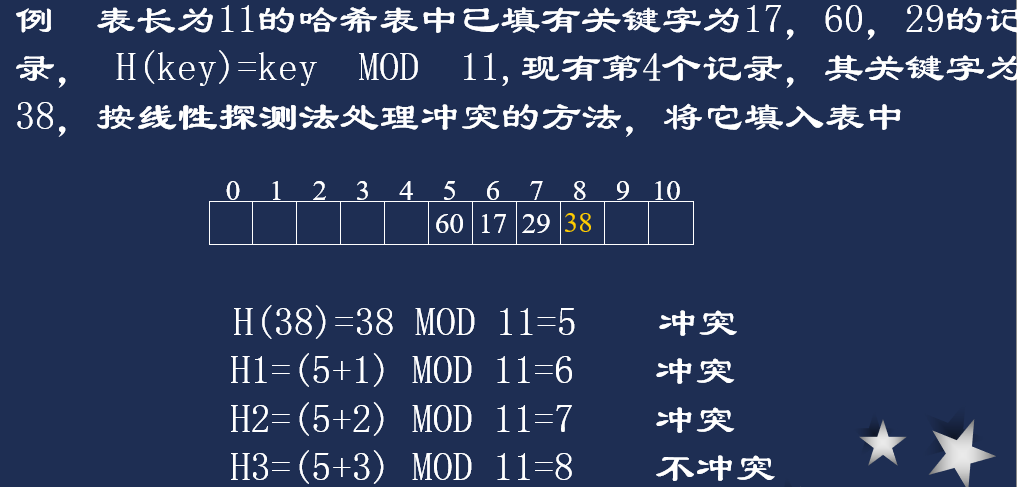
注意这个p是素数。

解决冲突的方法：

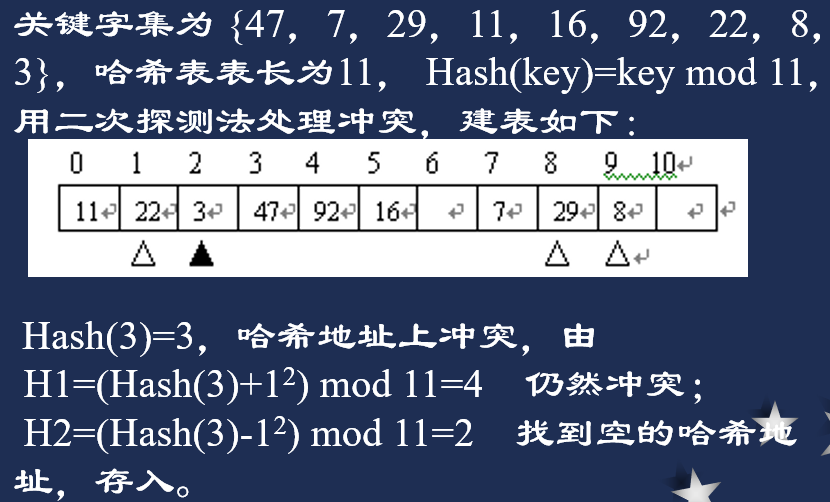
1. 开放定址法：

开放定址法，即是由关键字得到的哈希地址一旦产生了冲突，也就是说，该地址已经存放了数据元素，就去寻找下一个空的哈希地址，只要哈希表足够大，空的哈希地址总能找到，并将数据元素存入。

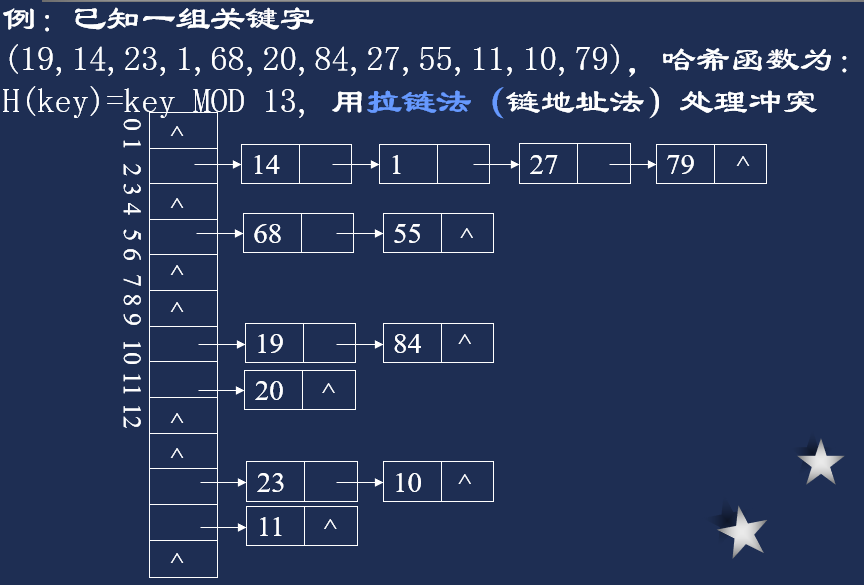
1. 线性探测法：



1. 二次探测法：



1. 拉链法：



影响ASL的因素：

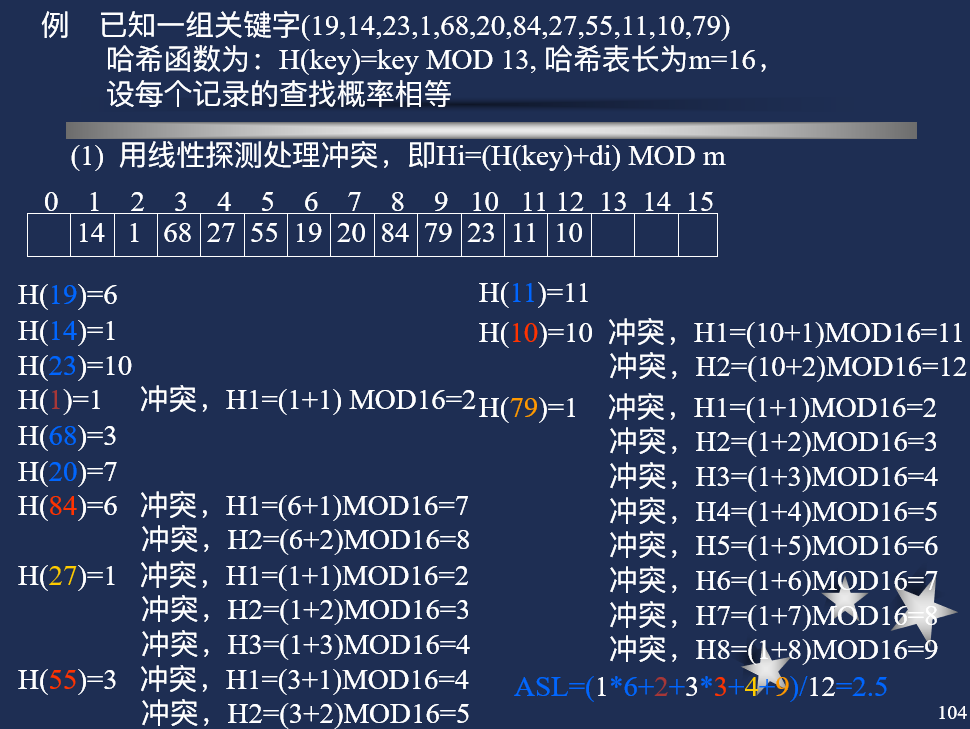
1. 哈希函数是否均匀。
2. 处理冲突的办法。
3. 哈希表的装填因子。

装填因子阿尔法定义为：填入表中的元素个数 / 哈希表的长度。

装填因子越大，填入表中的元素越多，产生冲突的可能性就越大，装填因子越小，填入表中的元素越少，产生冲突的可能性就越小。

拉链法的查找成功的平均查找长度为：1 + 阿尔法 / 2

线性探测法处理冲突，除留余数法建立散列表：



拉链法处理冲突，除留余数法建立散列表：



* 排序算法的选择：初始有序/无序、稳定/不稳定选择，时间、空间复杂性简单/复杂；排序算法稳定的原因。

|  |  |  |  |
| --- | --- | --- | --- |
| **方法** | **平均时间** | **最坏情况** | **辅助存储** |
| **简单排序** | **O(n2)** | **O(n2)** | **O(1)** |
| **快速排序** | **O(nlog2n)** | **O(n2)** | **O(nlog2n)** |
| **堆排序** | **O(nlog2n)** | **O(nlog2n)** | **O(1)** |
| **归并排序** | **O(nlog2n)** | **O(nlog2n)** | **O(n)** |

从时间上来看：

快速排序最佳。但在最坏情况下时间性能不如堆排序和归并排序。

从稳定性看：

直接插入排序、冒泡排序和归并排序是稳定的；而希尔排序、直接选择排序、快速排序和堆排序是不稳定排序。

选择排序方法：

（1）当待排序记录数n 较大，若要求排序稳定，则采用归并排序。

（2）当待排序记录数n 较大，关键字分布随机，而且不要求稳定时，可采用快速排序；

（3）当待排序记录数n 较大，关键字会出现正、逆序情形，可采用堆排序（或归并排序）。

（4）当待排序记录数n 较小，记录已接近有序或随机分布时，又要求排序稳定，可采用直接插入排序。

（5）当待排序记录数n 较小，且对稳定性不作要求时，可采用直接选择排序。

排序算法稳定的原因：

(1)冒泡排序

冒泡排序就是把小的元素往前调或者把大的元素往后调。比较是相邻的两个元素比较，交换也发生在这两个元素之间。所以，如果两个元素相等，我想你是不会再无聊地把他们俩交换一下的；如果两个相等的元素没有相邻，那么即使通过前面的两两交换把两个相邻起来，这时候也不会交换，所以相同元素的前后顺序并没有改变，所以冒泡排序是一种稳定排序算法。

(2)选择排序

选择排序是给每个位置选择当前元素最小的，比如给第一个位置选择最小的，在剩余元素里面给第二个元素选择第二小的，依次类推，直到第n - 1个元素，第n个元素不用选择了，因为只剩下它一个最大的元素了。序列5 8 5 2 9，我们知道第一遍选择第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了，所以选择排序不是一个稳定的排序算法。

(3)插入排序

插入排序是在一个已经有序的小序列的基础上，一次插入一个元素。当然，刚开始这个有序的小序列只有1个元素，就是第一个元素。比较是从有序序列的末尾开始，也就是想要插入的元素和已经有序的最大者开始比起，如果比它大则直接插入在其后面，否则一直往前找直到找到它该插入的位置。如果碰见一个和插入元素相等的，那么插入元素把想插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，所以插入排序是稳定的。

(4)快速排序

快速排序有两个方向，左边的i下标一直往右走，当a[i] <= a[center\_index]，其中center\_index是中枢元素的数组下标，一般取为数组第0个元素。而右边的j下标一直往左走，当a[j] > a[center\_index]。如果i和j都走不动了，i <= j，交换a[i]和a[j],重复上面的过程，直到i > j。 交换a[j]和a[center\_index]，完成一趟快速排序。在中枢元素和a[j]交换的时候，很有可能把前面的元素的稳定性打乱，比如序列为5 3 3 4 3 8 9 10 11，现在中枢元素5和3（第5个元素，下标从1开始计）交换就会把元素3的稳定性打乱，所以快速排序是一个不稳定的排序算法，不稳定发生在中枢元素和a[j] 交换的时刻。

(5)归并排序

归并排序是把序列递归地分成短序列，递归出口是短序列只有1个元素（认为直接有序）或者2个序列（1次比较和交换），然后把各个有序的短序列合并成一个有序的长序列，不断合并直到原序列全部排好序。可以发现，在1个或2个元素时，1个元素不会交换，2个元素如果大小相等也没有人故意交换，这不会破坏稳定性。那么，在短的有序序列合并的过程中，稳定是是否受到破坏？没有，合并过程中我们可以保证如果两个当前元素相等时，我们把处在前面的序列的元素保存在结果序列的前面，这样就保证了稳定性。所以，归并排序也是稳定的排序算法。

(6)希尔排序(shell)

希尔排序是按照不同步长对元素进行插入排序，当刚开始元素很无序的时候，步长最大，所以插入排序的元素个数很少，速度很快；当元素基本有序了，步长很小， 插入排序对于有序的序列效率很高。所以，希尔排序的时间复杂度会比O(n^2)好一些。由于多次插入排序，我们知道一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以shell排序是不稳定的。

(8)堆排序

我们知道堆的结构是节点i的孩子为2 \* i和2 \* i + 1节点，大顶堆要求父节点大于等于其2个子节点，小顶堆要求父节点小于等于其2个子节点。在一个长为n 的序列，堆排序的过程是从第n / 2开始和其子节点共3个值选择最大（大顶堆）或者最小（小顶堆），这3个元素之间的选择当然不会破坏稳定性。但当为n / 2 - 1， n / 2 - 2， ... 1这些个父节点选择元素时，就会破坏稳定性。有可能第n / 2个父节点交换把后面一个元素交换过去了，而第n / 2 - 1个父节点把后面一个相同的元素没 有交换，那么这2个相同的元素之间的稳定性就被破坏了。所以，堆排序不是稳定的排序算法。