串
数组和广义表
树和二叉树
    二叉树
    树和森林
    哈弗曼树/霍夫曼树
二叉查找树、红黑树、平衡二叉树
    二叉查找树（BST）
    红黑树
    平衡二叉树（AVL）
图
    图的存储形式
    图的遍历
    生成树和最小生成树
    有向无环图及其应用
查找
    动态查找
    查找总结
B_树的B+树
    B_树
    B+树
哈希表
    哈希函数的构造方法
    哈希冲突的解决方案
    哈希表的查找
    哈希表的删除
    哈希表达查找效率
    Hash的应用
外部排序
有效的算法设计
    高级数据结构
    算法
        回溯法
        动态规划

题目
排序算法
大数据题

“3+2”：3中数据结构：线性结构、树、图；2种算法：查找、排序。其中线性结构又可以分为顺序表、链表、队列、栈，它们在平时有着更加广泛的应用。而树、图则在解决一些特定的算法时更加管用。

- 数据结构三要素：
  1、逻辑结构：线性和非线性
  2、存储结构：顺序，链式，索引，散列
  3、数据运算：算法
- 逻辑结构：数据之间的相互关系。
  - 集合 结构中的数据元素除了同属于一种类型外，别无其它关系。
  - 线性结构 数据元素之间一对一的关系
  - 树形结构 数据元素之间一对多的关系
  - 图状结构或网状结构 结构中的数据元素之间存在多对多的关系

串

串(String)是零个或多个字符组成的有限序列。长度为零的串称为空串(Empty String)，它不包含任何字符。通常将仅由一个或多个空格组成的串称为空白串(Blank String) 注意：空串和空白串的不同，例如“ ”和“”分别表示长度为1的空白串和长度为0的空串。
比较重点的是KMP算法

数组和广义表

数组和广义表可看成是一种特殊的线性表，其特殊在于: 表中的元素本身也是一种线性表。内存连续。根据下标在O(1)时间读/写任何元素。

> 二维数组，多维数组，广义表、树、图都属于非线性结构

- 数组
  数组的顺序存储：行优先顺序；列优先顺序。数组中的任一元素可以在相同的时间内存取，即顺序存储的数组是一个随机存取结构。
  关联数组(Associative Array)，又称映射（ Map ）、字典（ Dictionary ）是一个抽

象的数据结构，它包含着类似于(键，值)的有序对。 不是线性表。

> 矩阵的压缩：对称矩阵、三角矩阵：直接存储矩阵的上三角或者下三角元素。注意区分i>=j和i

- 广义表
  广义表（Lists，又称列表）是线性表的推广。
  广义表是n(n≥0)个元素a1,a2,a3,…,an的有限序列，其中ai或者是原子项，或者是一个广义表。
  若广义表LS（n>=1)非空，则a1是LS的表头，其余元素组成的表(a2,…an)称为LS的表尾。
  广义表的元素可以是广义表，也可以是原子，广义表的元素也可以为空。
  表尾是指除去表头后剩下的元素组成的表，表头可以为表或单元素值。所以表尾不可以是单个元素值。
- 链表和数组有什么区别
  数组和链表有以下几点不同:
  - 存储形式:数组是一块连续的空间,声明时就要确定长度;链表是一块可不连续的动态空间,长度可变,每个结点要保存相邻结点指针;
  - 数据查找:数组的线性查找速度快,查找操作直接使用偏移地址;链表需要按顺序检索结点,效率低;
    数据插入或删除:链表可以快速插入和删除结点,而数组则可能需要大量数据移动;
  - 越界问题:链表不存在越界问题,数组有越界问题;
    > 说明: 在选择数组或链表数据结构时, 一定要根据实际需要进行选择; 数组便于查询, 链表便于插入删除; 数组节省空间但是长度固定, 链表虽然变长但是占了更多的存储空间;

树和二叉树

一种非线性结构。

1. 兄弟结点：同一双亲的孩子结点；
2. 堂兄结点：同一层上结点；
3. 结点层次：根结点的层定义为1；根的孩子为第二层结点，依此类推；
4. 树的高（深）度：树中最大的结点层
5. 结点的度：结点子树的个数

6. 树的度： 树中最大的结点度。
7. 分枝结点：度不为0的结点（非终端结点）；
8. 森林：互不相交的树集合；
9. 有序树：子树有序的树，如：家族树；
10. 无序树：不考虑子树的顺序；

二叉树

二叉树可以为空。二叉树结点的子树要区分左子树和右子树，即使只有一棵子树也要进行区分，说明它是左子树，还是右子树。这是二叉树与树的最主要的差别。

- 注意区分：二叉树、二叉查找树/二叉排序树/二叉搜索树（是一种树）、二叉平衡树（是一棵平衡的二叉查找树）
  二叉平衡树肯定是一颗二叉排序树。堆不是一颗二叉平衡树。
  二叉树与树是不同的，二叉树不等价于分支树最多为二的有序树。当一个结点只包含一个子节点时，对于有序树并无左右孩子之分，而对于二叉树来说依然有左右孩子之分，所以**二叉树与树是两种不同的结构。**
- 性质：
  a. 在二叉树的第 i 层上至多有2i-1个结点。
  b. 深度为 k 的二叉树上至多含 2k-1 个结点（k≥1）
  c. 对任何一棵二叉树，若它含有n0个叶子结点、n2个度为 2 的结点，则必存在关系式：n0= n2+1。
  d. 具有 n 个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor+1$ 。
  e. n个结点的二叉树中，完全二叉树具有最小的路径长度。
  f. 如果对一棵有n个结点的完全二叉树的结点按层序编号,则对任一结点 i（1<=i<=n),有：
  如果i = 1，则结点i无双亲，是二叉树的根；如果i>1，则其双亲的编号是 i/2(整除）。
  如果2i>n，无左孩子；否则，其左孩子是结点2i。
  如果2i + 1>n，则结点i无右孩子；否则，其右孩子是结点2i + 1。
- 二叉树的存储结构
  - 顺序存储结构：仅仅适用于满或完全二叉树，结点之间的层次关系由性质5确定。

    - 二叉链表法：每个节点存储左子树和右子树。三叉链表：左子树、右子树、父节点，总的指针是n+2
    - 在有n个结点的二叉链表中，值为非空的链域的个数为n-1。在有N个结点的二叉链表中必定有2N个链域。除根结点外，其余N-1个结点都有一个父结点。所以，一共有N-1个非空链域，其余2N-(N-1)=N+1个为空链域。
    - 二叉链存储法也叫孩子兄弟法，左指针指向左孩子，右指针指向右兄弟。而中序遍历的顺序是左孩子，根，右孩子。这种遍历顺序与存储结构不同，因此需要堆栈保存中间结果。而中序遍历检索二叉树时，由于其存储结构跟遍历顺序相符，因此不需要用堆栈。

- 遍历二叉树
  先序遍历DLR、中序遍历LDR、后续遍历LRD、层次遍历
- 线索二叉树
  线索二叉树：对二叉树所有结点做某种处理可在遍历过程中实现；检索（查找）二叉树某个结点，可通过遍历实现；如果能将二叉树线索化，就可以简化遍历算法，提高遍历速度，目的是加快查找结点的前驱或后继的速度。
  如何线索化？以中序遍历为例，若能将中序序列中每个结点前趋、后继信息保存起来，以后再遍历二叉树时就可以根据所保存的结点前趋、后继信息对二叉树进行遍历。对于二叉树的线索化，实质上就是遍历一次二叉树，只是在遍历的过程中，检查当前结点左，右指针域是否为空，若为空，将它们改为指向前驱结点或后继结点的线索。**前驱就是在这一点之前走过的点，不是下一将要去往的点。**
  加上结点前趋后继信息（结索）的二叉树称为**线索二叉树。**

树和森林

- 树的存储结构：
    - 双亲表示法
    - 孩子表示法
    - 利用图表示树
    - 孩子兄弟表示法（二叉树表示法）：链表中每个结点的两指针域分别指向其第一个孩子结点和下一个兄弟结点

将树转化成二叉树：右子树一定为空

1. 加线：在兄弟之间加一连线
2. 抹线：对每个结点，除了其左孩子外，去除其与其余孩子之间的关系

3. 旋转：以树的根结点为轴心，将整树顺时针转45°

森林转换成二叉树：

1. 将各棵树分别转换成二叉树
2. 将每棵树的根结点用线相连
3. 以第一棵树根结点为二叉树的根

树与转换后的二叉树的关系：转换后的二叉树的先序对应树的先序遍历；转换后的二叉树的中序对应树的后序遍历

哈弗曼树/霍夫曼树

一些概念

1. 路径：从一个祖先结点到子孙结点之间的分支构成这两个结点间的路径；
2. 路径长度：路径上的分支数目称为路径长度；
3. 树的路径长度：从根到每个结点的路径长度之和。
4. 结点的权：根据应用的需要可以给树的结点赋权值；
5. 结点的带权路径长度：从根到该结点的路径长度与该结点权的乘积；
6. 树的带权路径长度=树中所有叶子结点的带权路径之和；通常记作 WPL=∑wi×li
7. 哈夫曼树：假设有n个权值(w1, w2, … , wn)，构造有n个叶子结点的二叉树，每个叶子结点有一个 wi作为它的权值。则带权路径长度最小的二叉树称为哈夫曼树。最优二叉树。

前缀码的定义：

> 在一个字符集中，任何一个字符的编码都不是另一个字符编码的前缀。
> 霍夫曼编码就是前缀码，可用于快速判断霍夫曼编码是否正确。
> 霍夫曼树是满二叉树，若有n个节点，则共有(n+1)/2个码子

给定n个权值作为n的叶子结点，构造一棵二叉树，若带权路径长度达到最小，称这样的**二叉树为最优二叉树，也称为霍夫曼树(Huffman Tree)。**

> 霍夫曼树是带权路径长度最短的树，权值较大的结点离根较近。

其他性质：

> 假设哈夫曼树是二叉的话，则度为0的结点个数为N，度为2的结点个数为N-1，则结点总数为2N-1。哈夫曼树的结点个数必为奇数。

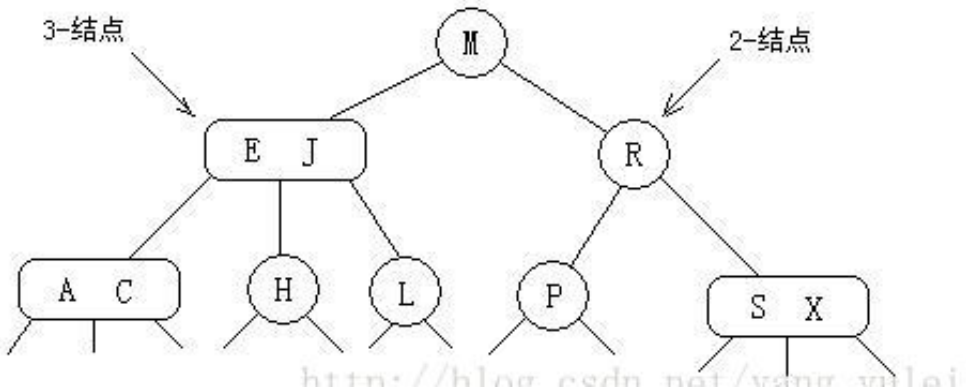
> 哈夫曼树不一定是完全二叉树，但一定是最优二叉树。
> 若度为m的哈夫曼树中,其叶结点个数为n,则非叶结点的个数为[(n-1)/(m-1)]。边的数目等于度。

二叉查找树、红黑树、平衡二叉树

二叉查找树（BST）

特殊的二叉树，又称为排序二叉树、二叉搜索树、二叉排序树。

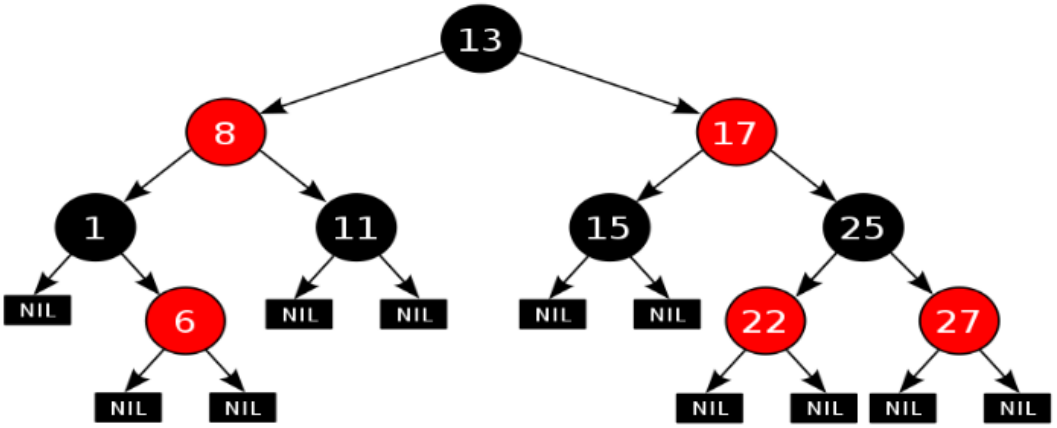
- 性质
  二叉查找树实际上是数据域有序的二叉树，即对树上的每个结点，都满足其
  - 左子树上所有结点的数据域**均小于或等于**根结点的数据域，
  - 右子树上所有结点的数据域**均大于**根结点的数据域。
  - 有序但不平衡。
- 2-3查找树
  2-结点：含有一个键(及值)和两条链接，左链接指向的2-3树中的键都小于该结点，右链接指向的2-3树中的键都大于该结点。
  3-结点：含有两个键(及值)和三条链接，左链接指向的2-3树中的键都小于该结点，中链接指向的2-3树中的键都位于该结点的两个键之间，右链接指向的2-3树中的键都大于该结点。（2-3指的是2叉-3叉的意思）



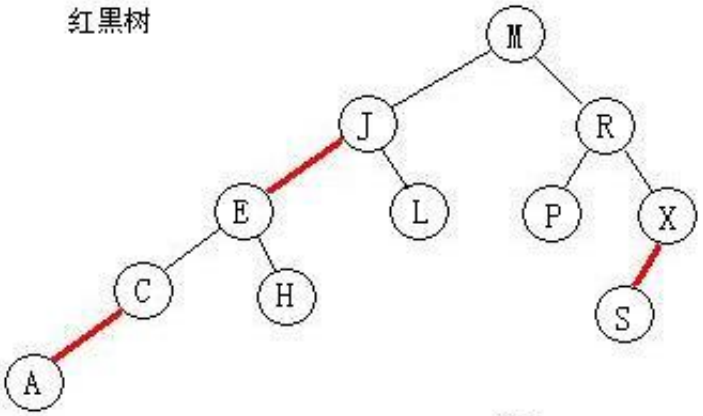
红黑树

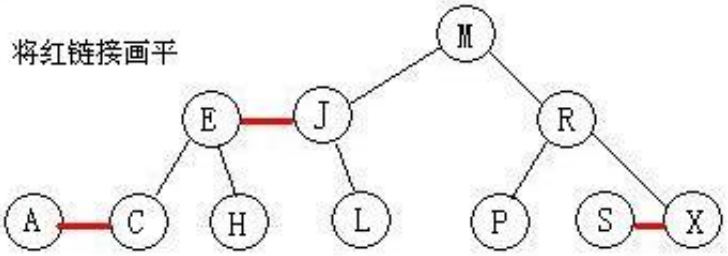
- 性质
  - 是一个BST（是一个高级的二叉查找树）
  - 根节点是黑的，并定义NULL为黑色
  - 每个结点要么是红的，要么是黑的。

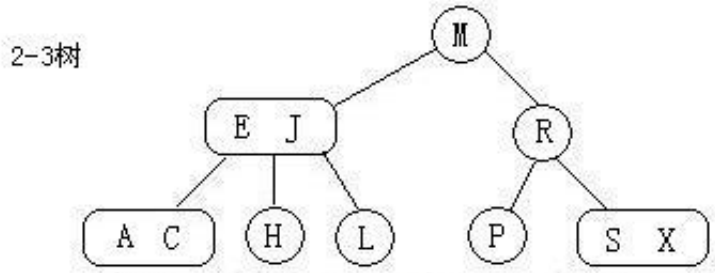
- 如果一个结点是红色，那么两个儿子是黑色，且父节点也是黑色（红黑节点交替）。
- 对于任一结点而言，它到叶结点的每一条路径都包含相同数目的黑色节点，称为黑高（红节点相当于拉到同一层，按2-3查找树理解）。
- 隐含的性质
  - 任意一颗以黑色节点为根的子树也必定是一棵红黑树（递归定义）
  - 左子树和右子树的高度最多差1倍



红色节点与父节点拉到同一水平线，这里应该形同 2-4查找树。





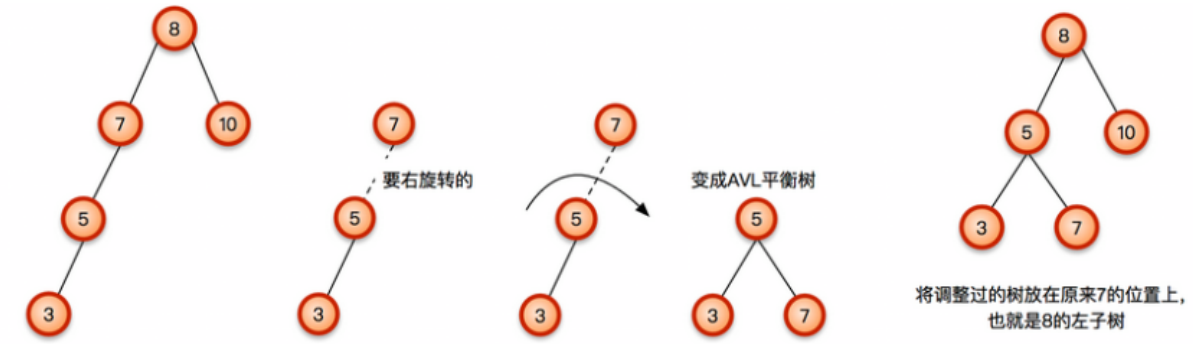


形同 2-3查找树

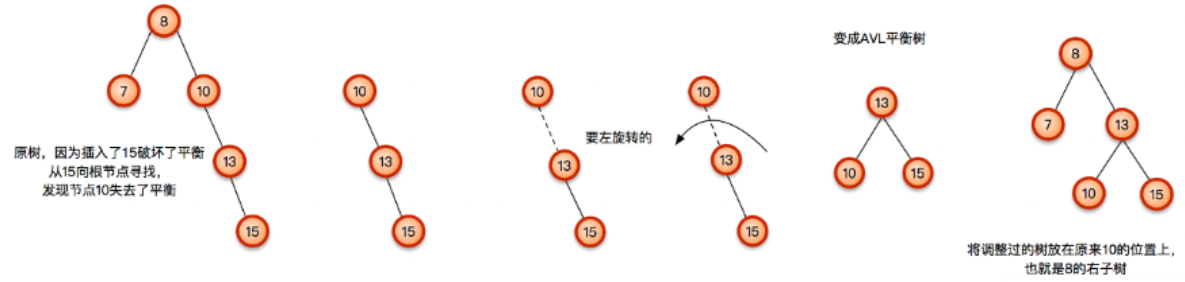
平衡二叉树（AVL）

AVL树的创建是基于二叉查找树的插入代码的基础上，增加平衡操作的。
需要从插入的结点开始从下往上判断结点是否失衡，因此，需要在调用insert函数以后，更新当前子树的高度，并在这之后根据树型来进行相应的平衡操作。

- 右旋

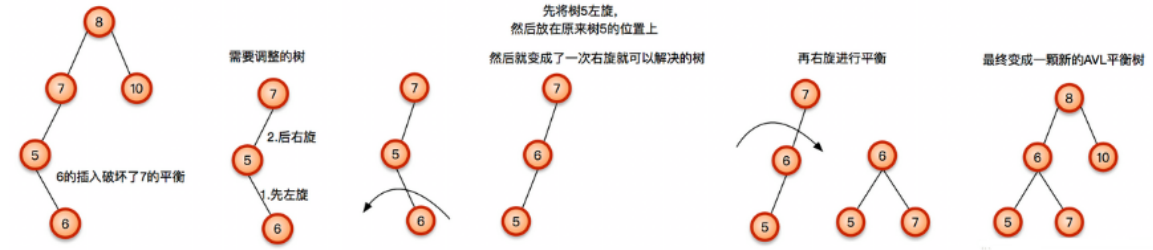


- 左旋

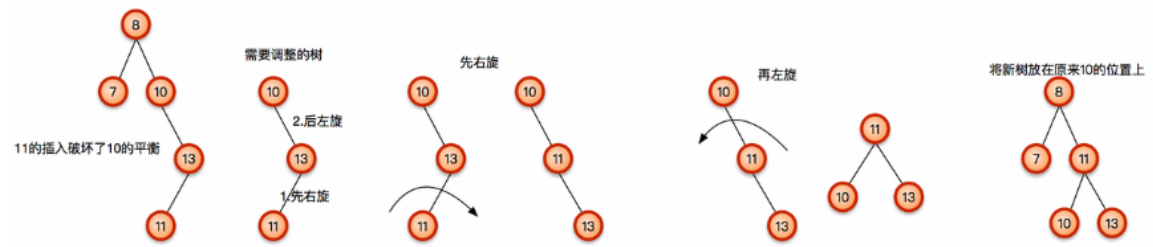


- 先左旋后右旋

LR型



RL型



- 与红黑树的区别

红黑树与AVL树都是平衡树，但是AVL是完全平衡的(平衡就是值树中任意节点的左子树和右子树高度差不超过1)；
红黑树效率更高，因为AVL为了保证其完全平衡，插入和删除的时候在最坏的情况下要旋转logN次，而红黑树插入和删除的旋转次数要比AVL少。

图

- 无向图
    - 连通：顶点v至v' 之间有路径存在
    - 连通图：无向图图 G 的任意两点之间都是连通的，则称G是连通图。
    - 连通分量：极大连通子图，子图中包含的顶点个数极大
    - 所有顶点度的和必须为偶数
    - 回路或环：第一个顶点和最后一个顶点相同的路径。（多个环吧）
    - 简单回路或简单环：除第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路（我理解就是一条回路，即一个环）
- 有向图：
    - 连通：顶点v至v'之间有路径存在
    - 强连通图：有向图G的任意两点之间都是连通的，则称G是强连通图。各个顶点间均可达。
    - 强连通分量：极大连通子图
    - 有向图顶点的度是顶点的入度与出度之和。邻接矩阵中第V行中的1的个数是V的出度
      *生成树：极小连通子图。包含图的所有n个结点，但只含图的n-1条边。在生成树中添加一条边之后，必定会形成回路或环。
    - 完全图：有 n(n-1)/2 条边的无向图。其中n是结点个数。必定是连通图。
      *有向完全图：有n(n-1)条边的有向图。其中n是结点个数。每两个顶点之间都有两条方向相反的边连接的图。
    - 一个无向图 G=(V,E) 是连通的，那么边的数目大于等于顶点的数目减一：|E|>=|V|-1，而反之不成立。如果 G=(V,E) 是有向图，那么它是强连通图的必要条件是边的数目大于等于顶点的数目：|E|>=|V|，而反之不成立。
      没有回路的无向图是连通的当且仅当它是树，即等价于：|E|=|V|-1。
    - 回路或环：第一个顶点和最后一个顶点相同的路径。（多个环吧）
    - 简单回路或简单环：除第一个顶点和最后一个顶点之外，其余顶点不重复出

现的回路。（我理解就是一条回路，即一个环）

图的存储形式

1. 邻接矩阵和加权邻接矩阵（二维矩阵）
   - 无权有向图：出度: i行之和；入度: j列之和。
   - 无权无向图：i结点的度: i行或i列之和。
   - 加权邻接矩阵：相连为w，不相连为∞

	0	1	2	3	4
0	0	1	0	1	1
1	1	0	1	0	1
2	0	1	0	1	0
3	1	0	1	0	1
4	1	1	0	1	0

图2：邻接矩阵

2. 邻接表（数组+链表）
   - 用顶点数组表、边（弧）表表示该有向图或无向图
   - 顶点数组表：用数组存放所有的顶点。数组大小为图顶点数n
   - 边表（边结点表）：每条边用一个结点进行表示。同一个结点的所有的边形成它的边结点单链表。
   - n个顶点的无向图的邻接表最多有n(n-1)个边表结点。有n个顶点的无向图最多有n*(n-1)/2*条边，此时为完全无向图，而在邻接表中每条边存储两次，所以有*n*(n-1)个结点

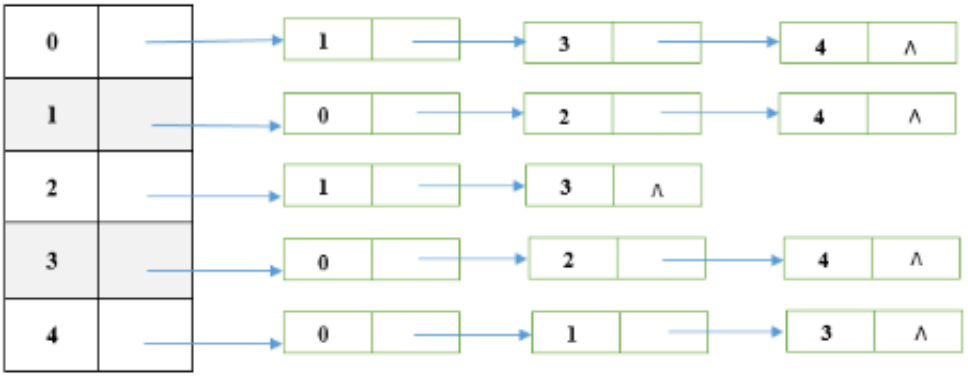


图4 邻接表

图的遍历

深度优先搜索（dfs）利用栈，广度优先搜索（bfs）利用队列
求一条从顶点i到顶点s的简单路径–深搜。
求两个顶点之间的一条长度最短的路径–广搜。
当各边上的权值均相等时,BFS算法可用来解决单源最短路径问题。

生成树和最小生成树

每次遍历一个连通图将图的边分成遍历所经过的边和没有经过的边两部分，将遍历经过的边同图的顶点构成一个子图，该子图称为生成树。因此有DFS生成树和BFS生成树。

> 生成树是连通图的极小子图，有n个顶点的连通图的生成树必定有n-1条边,在生成树中任意增加一条边，必定产生回路。若砍去它的一条边，就会把生成树变成非连通子图

最小生成树：

> 生成树中边的权值(代价)之和最小的树。最小生成树问题是构造连通网的最小代价生成树。

Kruskal算法：

> 令最小生成树集合T初始状态为空，在有n个顶点的图中选取代价最小的边并从图中删去。若该边加到T中有回路则丢弃，否则留在T中；依此类推，直至T中有n-1条边为止。

- Prim算法、Kruskal算法和Dijkstra算法均属于贪心算法。
    a. Dijkstra算法解决的是带权重的有向图上单源最短路径问题，该算法要求所有边的权重都为非负值。
    b. Dijkstra算法解决了从某个原点到其余各顶点的最短路径问题，由循环嵌套可知该算法的时间复杂度为O(N*N*)。若要求任一顶点到其余所有顶点的最短路径，一个比较简单的方法是对每个顶点当做源点运行一次该算法，等于在原有算法的基础上，再来一次循环，此时整个算法的复杂度就变成了*O(N*N*N)。
    c. Bellman-Ford算法解决的是一般情况下的单源最短路径问题，在这里，边的权重可以为负值。该算法返回一个布尔值，以表明是否存在一个从源节点可以到达的权重为负值的环路。如果存在这样一个环路，算法将告诉我们不存

在解决方案。如果没有这种环路存在，算法将给出最短路径和它们的权重。

有向无环图及其应用

拓扑排序。在用邻接表表示图时,对有n个顶点和e条弧的有向图而言时间复杂度为O(n+e)。一个有向图能被拓扑排序的充要条件就是它是一个有向无环图。拓扑序列唯一不能唯一确定有向图。

AOV网(Activity On Vertex)：

> 用顶点表示活动，边表示活动的优先关系的有向图称为AOV网。AOV网中不允许有回路，这意味着某项活动以自己为先决条件。

拓扑有序序列：

> 把AOV网络中各顶点按照它们相互之间的优先关系排列一个线性序列的过程。若vi是vj前驱，则vi一定在vj之前；对于没有优先关系的点，顺序任意。

- 拓扑排序：
  对AOV网络中顶点构造拓扑有序序列的过程。
- 方法：
  a. 在有向图中选一个没有前驱的顶点且输出之
  b. 从图中删除该顶点和所有以它为尾的弧
  c. 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(此时说明图中有环）
- 采用深度优先搜索或拓扑排序算法可以判断出一个有向图中是否有环(回路).
  深度优先搜索只要在其中记录下搜索的节点数n，当n大于图中节点数时退出，并可以得出有回路。
  若有回路，则拓扑排序访问不到图中所有的节点，所以也可以得出回路。
- 算法描述：
  a. 把邻接表中入度为0的顶点依此进栈
  b. 若栈不空，则
  栈顶元素vj退栈并输出；
  在邻接表中查找vj的直接后继vk，把vk的入度减1；若vk的入度为0则进栈
  c. 若栈空时输出的顶点个数不是n，则有向图有环；否则，拓扑排序完毕。

AOE网：

> 带权的有向无环图，其中顶点表示事件，弧表示活动，权表示活动持续时间。在工程上常用来表示工程进度计划。

- 一些定义：
  a. 事件的最早发生时间（ve(j)）：从源点到j结点的最长的路径。意味着事件最早能够发生的时间。
  b. 事件的最迟发生时间（vl(j)）：不影响工程的如期完工，事件j必须发生的时间。
  c. 活动ai由弧

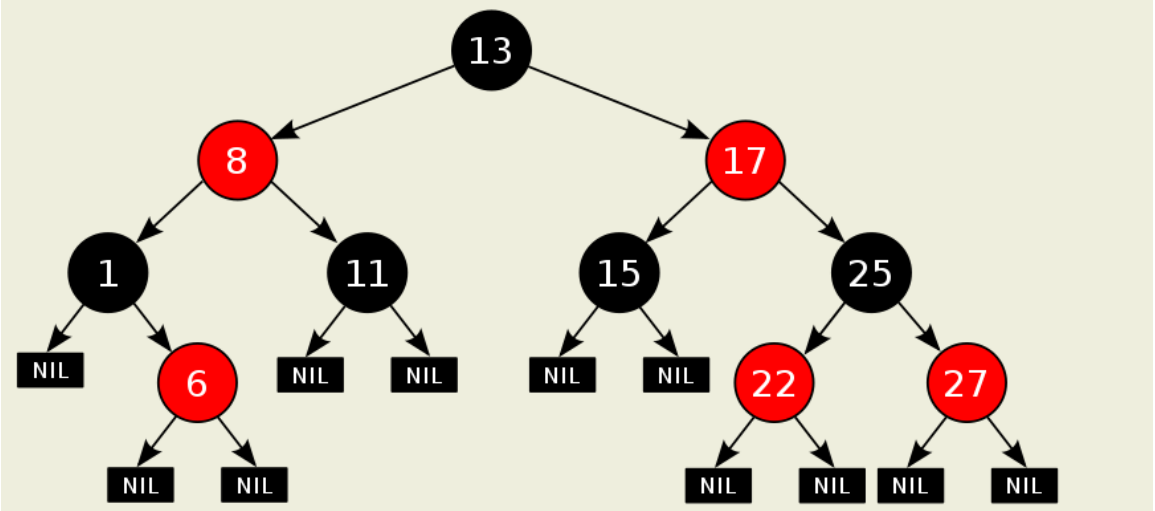
查找

静态查找有：顺序查找、折半查找、索引查找、分块查找，
动态查找有：二叉排序树查找，最优二叉树查找，键树查找，哈希表查找

动态查找

常见的平衡二叉树：

- 红黑树是平衡二叉树，也就是左右子树是平衡的，高度大概相等。

> 这种情况等价于一块完全二叉树的高度，查找的时间复杂度是树的高度，为logn，插入操作的平均时间复杂度为O(logn)，最坏时间复杂度为O(logn)
>
> 
>
>
> - 节点是红色或黑色。
> - 根是黑色。
> - 所有叶子都是黑色（叶子是NIL节点）。
> - 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

> ◦ 从任一节点到其每个叶子的所有简单路径 都包含相同数目的黑色节点。

- avl树(平衡二叉树)

> 红黑树和AVL树查找、插入、删除的时间复杂度相同；包含n个内部结点的红黑树的高度是o(logn); TreeMap 是一个红黑树的实现，能保证插入的值保证排序

- STL和linux多使用红黑树作为平衡树的实现：
  1、如果插入一个node引起了树的不平衡，AVL和RB-Tree都是最多只需要2次旋转操作，即两者都是O(1)；
  2、但是在删除node引起树的不平衡时，最坏情况下，AVL需要维护从被删node到root这条路径上所有node的平衡性，因此需要旋转的量级O(logN)，而RB-Tree最多只需3次旋转，只需要O(1)的复杂度。
  3、其次，AVL的结构相较RB-Tree来说更为平衡，在插入和删除node更容易引起Tree的unbalance，因此在大量数据需要插入或者删除时，AVL需要rebalance的频率会更高。因此，RB-Tree在需要大量插入和删除node的场景下，效率更高。自然，由于AVL高度平衡，因此AVL的search效率更高。
  4、map的实现只是折衷了两者在search、insert以及delete下的效率。总体来说，RB-tree的统计性能是高于AVL的。

查找总结

- 既希望较快的查找又便于线性表动态变化的查找方法是哈希法查找。
- 二分法是基于顺序表的一种查找方式，时间复杂度为O(logn)；
- 通过哈希函数将值转化成存放该值的目标地址，O（1）
- 二叉树的平均查找长度为O(log2n)——O(n).二叉排序树的查找效率与二叉树的高度有关，高度越低，查找效率越高。
- 二叉树的查找成功的平均查找长度ASL不超过二叉树的高度。二叉树的高度与二叉树的形态有关，n个节点的完全二叉树高度最小，高度为[log2n]+1,n个节点的单只二叉树的高度最大，高度为n，此时查找成功的ASL为最大(n+1)/2，因此二叉树的高度范围为[log2n]+1——n.

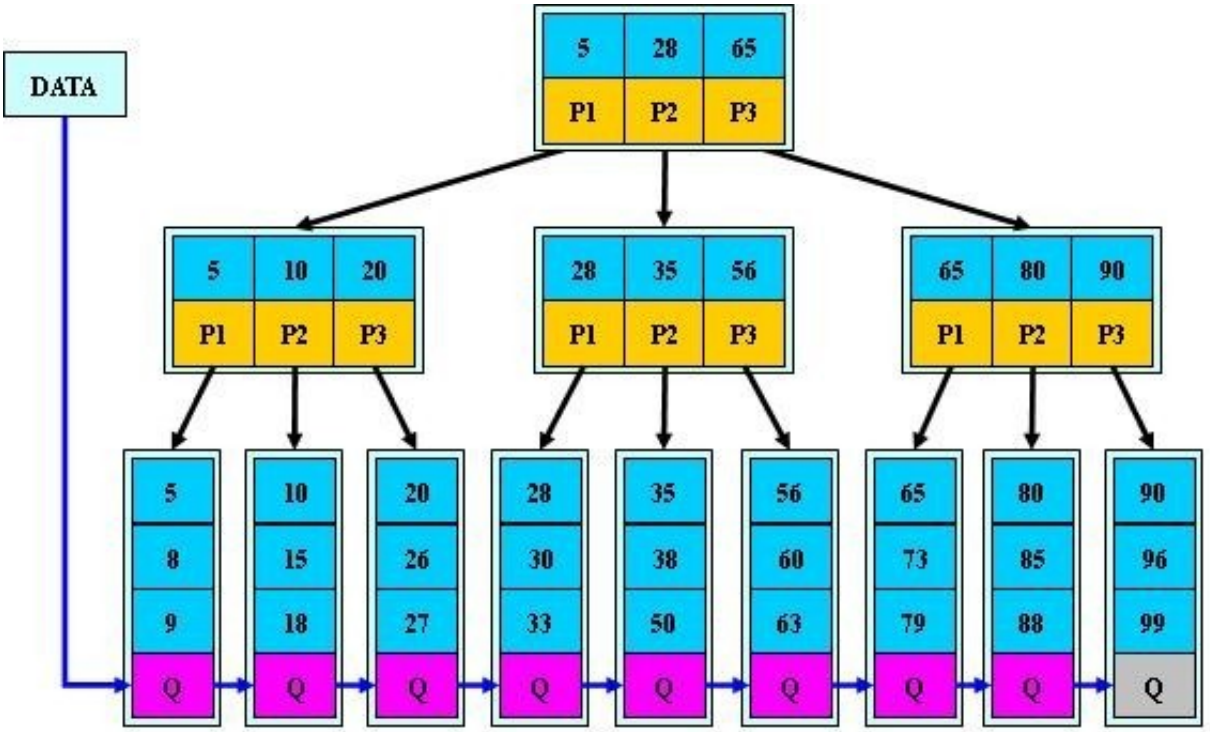
B_树的B+树

B_树

B-树就是B树。m阶B_树满足或空，或为满足下列性质的m叉树：

B+树

在实际的文件系统中，用的是B+树或其变形。有关性质与操作类似与B_树。



差异：

- 有n棵子树的结点中有n个关键字，每个关键字不保存数据，只用来索引，所有数据都保存在叶子节点。
- 所有叶子结点中包含全部关键字信息，及对应记录位置信息及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。(而B树的叶子节点并没有包括全部需要查找的信息)
- 所有非叶子为索引，结点中仅含有其子树根结点中最大（或最小）关键字。 (而B树的非终节点也包含需要查找的有效信息)
- 非叶最底层顺序联结，这样可以进行顺序查找

B+特性

- 所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；
- 不可能在非叶子结点命中
- 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层
- 更适合文件索引系统
- B+树插入操作的平均时间复杂度为O(logn)，最坏时间复杂度为O(logn)

为什么说B+tree比B树更适合实际应用中操作系统的文件索引和数据库索引？

1. B+tree的磁盘读写代价更低
    - B+tree的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。
    - 举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内部结点需要2个盘快。而B+树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B树就比B+树多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。
2. B+tree的查询效率更加稳定
    - 由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

B树和B+树都是平衡的多叉树。B树和B+树都可用于文件的索引结构。B树和B+树都能有效的支持随机检索。B+树既能索引查找也能顺序查找.

哈希表

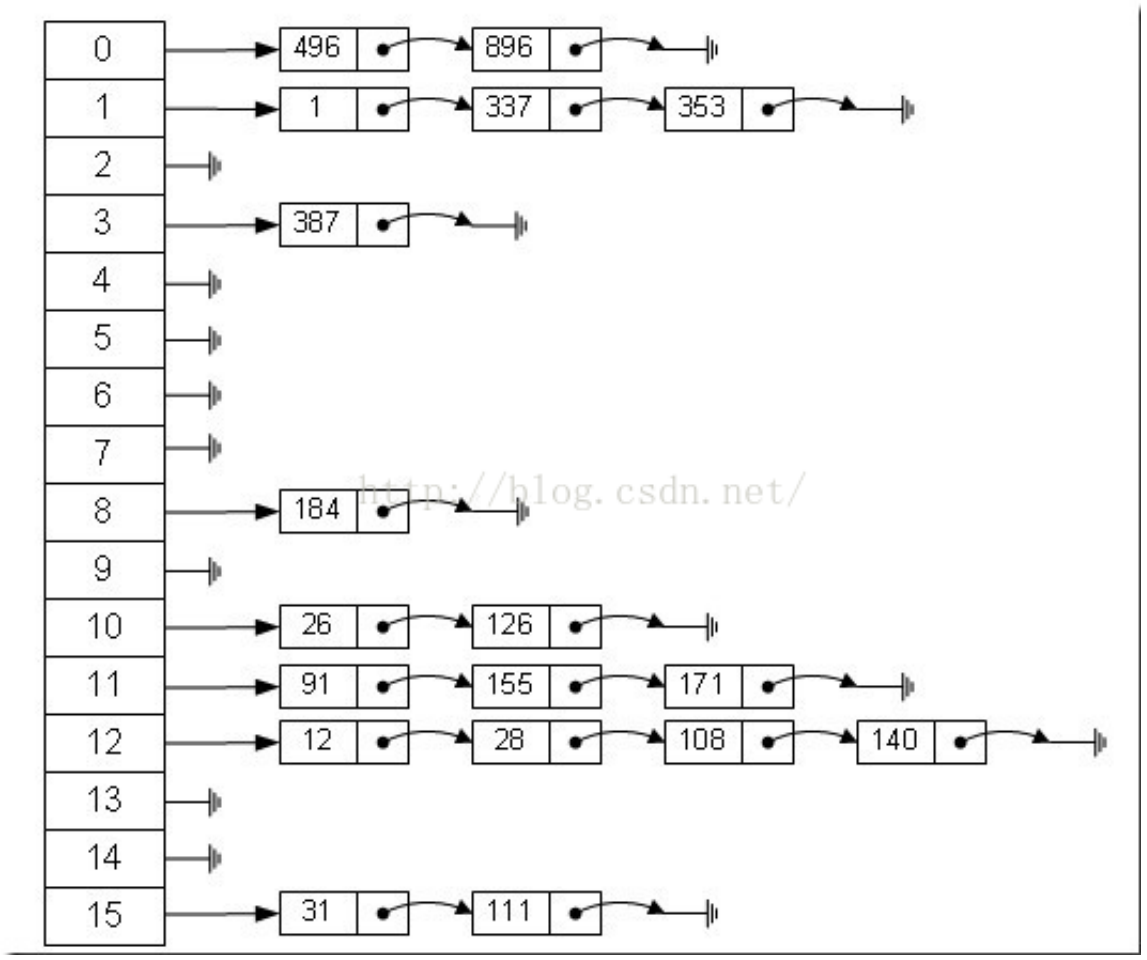
哈希表（Hash table，也叫散列表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。
这个映射函数叫做**散列函数**，存放记录的数组叫做**散列表**。

> 记录的存储位置=f(关键字)
> 这里的对应关系f称为散列函数，又称为哈希（Hash函数），采用散列技术将记录存储在一块连续的存储空间中，这块连续存储空间称为散列表或哈希表（Hash table）。

- 哈希表hashtable(key，value) 就是把Key通过一个固定的算法函数（所谓的哈希函数）转换成一个整型数字，然后就将该数字对数组长度进行取余，取余结果就当作数组的下标，将value存储在以该数字为下标的数组空间里。
- 使用哈希表进行**查询**的时候，就是再次使用哈希函数将key转换为对应的数组下标，并定位到该空间获取value，如此一来，就可以充分利用到数组的定位性能进

行数据定位。

哈希表表示方法（拉链法）：



> 左边很明显是个数组，数组的每个成员包括一个指针，指向一个链表的头，当然这个链表可能为空，也可能元素很多。我们根据元素的一些特征把元素分配到不同的链表中去，也是根据这些特征，找到正确的链表，再从链表中找出这个元素。

哈希函数的构造方法

- 直接定制法
  哈希函数为关键字的线性函数如 H（key）=a*key+b
- 数字分析法
  取分布均匀的若干位或他们的组合构成全体
- 平方取中法
- 折叠法
- 除留余数法（用的较多）
  H（key）=key MOD p （p<=m m为表长），p应为不大于m的质数或是不含20

以下的质因子的合数，这样可以减少地址的重复（冲突）

- 随机数法
  取关键字的随机函数值为它的散列地址

设计的考虑因素

1. 计算散列地址所需要的时间（即hash函数本身不要太复杂）
2. 关键字的长度
3. 表长
4. 关键字分布是否均匀，是否有规律可循
5. 设计的hash函数在满足以上条件的情况下尽量减少冲突

哈希冲突的解决方案

hash函数解决冲突的方法有以下几个常用的方法

1. 开放定制法
2. 链地址法
3. 公共溢出区法
   建立一个特殊存储空间，专门存放冲突的数据。此种方法适用于数据和冲突较少的情况。
4. 再散列法
   准备若干个hash函数，如果使用第一个hash函数发生了冲突，就使用第二个hash函数，第二个也冲突，使用第三个……

> 重点了解一下开放定制法和链地址法

- 开放定制法：
  首先有一个H（key）的哈希函数
  如果H（key1）=H（keyi），那么keyi存储位置Hi=(H(key)+di)MODmm为表长
  di有三种取法
  1）线性探测再散列，di=c∗i
  2）平方探测再散列，di=12,−12,22,−22……
  3）随机探测在散列（双探测再散列），di是一组伪随机数列

> 增量di应该具有以下特点（完备性）：产生的Hi（地址）均不相同，且所产生的s（m-1）个Hi能覆盖hash表中的所有地址
> 平方探测时表长m必须为4j+3的质数（平方探测表长有限制）

> 随机探测时m和di没有公因子（随机探测di有限制）

哈希表的查找

查找过程和造表过程一致，假设采用开放定址法处理冲突，则查找过程为：
1、对于给定的key，计算hash地址index = H（key）
2、如果数组arr【index】的值为空 则查找不成功
3、如果数组arr【index】== key 则查找成功
4、否则 使用冲突解决方法求下一个地址，直到arr【index】== key或者 arr【index】==null

哈希表的删除

首先链地址法是可以直接删除元素的。
但是开放定址法是不行的，拿前面的双探测再散列来说，假如我们删除了元素1，将其位置置空，那 23就永远找不到了。正确做法应该是删除之后置入一个原来不存在的数据，比如-1

哈希表达查找效率

决定hash表查找的ASL因素：
1）选用的hash函数
2）选用的处理冲突的方法
3）hash表的饱和度，装载因子 α=n/m(n表示实际装载数据长度 m为表长)

Hash的应用

1、Hash主要用于信息安全领域中加密算法，它把一些不同长度的信息转化成杂乱的128位的编码,这些编码值叫做Hash值. 也可以说，Hash就是找到一种数据内容和数据存放地址之间的映射关系。
2、查找：哈希表，又称为散列，是一种更加快捷的查找技术。我们之前的查找，都是这样一种思路：集合中拿出来一个元素，看看是否与我们要找的相等，如果不等，缩小范围，继续查找。而哈希表是完全另外一种思路：当我知道key值以后，我就可以直接计算出这个元素在集合中的位置，根本不需要一次又一次的查找！
举一个例子，假如我的数组A中，第i个元素里面装的key就是i，那么数字3肯定是在第3个位置，数字10肯定是在第10个位置。哈希表就是利用利用这种基本的思想，建立

一个从key到位置的函数，然后进行直接计算查找。
3、Hash表在海量数据处理中有着广泛应用。

> Hash Table的查询速度非常的快，几乎是O(1)的时间复杂度。
> hash就是找到一种数据内容和数据存放地址之间的映射关系。

外部排序

1、生成合并段（run）：读入文件的部分记录到内存 - >在内存中进行内部排序 - >将排好序的这些记录写入外存，形成合并段 - >再读入该文件的下面的记录，往复进行，直至文件中的记录全部形成合并段为止。
2、外部合并：将上一阶段生成的合并段调入内存，进行合并，直至最后形成一个有序的文件。
3、外部排序指的是大文件的排序，即待排序的记录存储在外存储器上，待排序的文件无法一次装入内存，需要在内存和外部存储器之间进行多次数据交换，以达到排序整个文件的目的。外部排序最常用的算法是多路归并排序，即将原文件分解成多个能够一次性装入内存的部分，分别把每一部分调入内存完成排序。然后，对已经排序的子文件进行多路归并排序
4、不管初始序列是否有序, 冒泡、选择排序时间复杂度是O(n^2),归并、堆排序时间复杂度是O(nlogn）
5、外部排序的总时间 = 内部排序（产出初始归并段）所需时间 + 外存信息读取时间 + 内部归并所需的时间
6、外排中使用置换选择排序的目的,是为了增加初始归并段的长度。减少外存读写次数需要减小归并趟数
7、根据内存容量设若干个输入缓冲区和一个输出缓冲区。若采用二路归并，用两个输入缓冲。
8、归并的方法类似于归并排序的归并算法。增加的是对缓冲的监视，对于输入，一旦缓冲空，要到相应文件读后续数据，对于输出缓冲，一旦缓冲满，要将缓冲内容写到文件中去。
9、外排序和内排序不只是考虑内外排序算法的性能，还要考虑IO数据交换效率的问题，内存存取速度远远高于外存。影响外排序的时间因素主要是内存与外设交换信息的总次数

有效的算法设计

1、贪心法。Dijkstra的最短路径(时间复杂度O(n2))；Prim求最小生成树邻接表存储时是O(n+e),图O(n2)；关键路径及关键活动的求法。

2、回溯法
4、分支限界法
5、分治法。分割、求解、合并。二分查找、归并排序、快速排序。
6、动态规划。Floyd-Warshall算法求解图中所有点对之间最短路径时间复杂度为O(n3)

动态规划解题的方法是一种高效率的方法，其时间复杂度通常为O(n2)，O(n3)等，可以解决相当大的信息量。

- 适用的原则：原则为优化原则，即整体优化可以分解为若干个局部优化。
- 动态规划比穷举法具有较少的计算次数
- 递归算法需要很大的栈空间，而动态规划不需要栈空间

贪心和动态规划的差别：

- 所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。
- 在动态规划算法中，每步所作的选择往往依赖于相关子问题的解。因而只有在解出相关子问题后，才能作出选择。而在贪心算法中，仅在当前状态下作出最好选择，即局部最优选择。然后再去解作出这个选择后产生的相应的子问题。
- 贪心算法所作的贪心选择可以依赖于以往所作过的选择，但决不依赖于将来所作的选择，也不依赖于子问题的解。正是由于这种差别，动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行,以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为一个规模更小的子问题。

P问题

- P问题，如果它可以通过运行多项式次(即运行时间至多是输入量大小的多项式函数的一种算法获得解决），可以找到一个能在多项式的时间里解决它的算法。—-确定性问题
- NP问题，虽然可以用计算机求解，但是对于任意常数k，它们不能在O(nk)时间内得到解答，可以在多项式的时间里验证一个解的问题。所有的P类问题都是NP问题。
- NP完全问题，知道有效的非确定性算法，但是不知道是否存在有效的确定性算法，同时，不能证明这些问题中的任何一个不存在有效的确定性算法。这类问题称为NP完全问题。

高级数据结构

- 红黑树
- 前缀树

算法

算法	思想	应用
分治法	把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并	循环赛日程安排问题、排序算法（快速排序、归并排序）
动态规划	通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法，适用于有重叠子问题和最优子结构性质的问题	背包问题、斐波那契数列
贪心法	一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法	旅行推销员问题（最短路径问题）、最小生成树、哈夫曼编码

回溯法

回溯法可以被认为是一个有过剪枝的DFS过程
利用回溯法解题的具体步骤

- 首先，要通过读题完成下面三个步骤：
  (1)描述解的形式，定义一个解空间，它包含问题的所有解。
  (2)构造状态空间树。
  (3)构造约束函数（用于杀死节点）。
- 然后就要通过DFS思想完成回溯，完整过程如下：
  (1)设置初始化的方案（给变量赋初值，读入已知数据等）。
  (2)变换方式去试探，若全部试完则转(7)。
  (3)判断此法是否成功（通过约束函数），不成功则转(2)。
  (4)试探成功则前进一步再试探。
  (5)正确方案还未找到则转(2)。
  (6)已找到一种方案则记录并打印。
  (7)退回一步（回溯），若未退到头则转(2)。
  (8)已退到头则结束或打印无解。

动态规划

- 基本思想：问题的最优解如果可以由子问题的最优解推导得到，则可以先求解子问题的最优解，在构造原问题的最优解；若子问题有较多的重复出现，则可以自底向上从最终子问题向原问题逐步求解。
- 使用条件：可分为多个相关子问题，子问题的解被重复使用
    - Optimal substructure（优化子结构）：
        - 一个问题的优化解包含了子问题的优化解
        - 缩小子问题集合，只需那些优化问题中包含的子问题，降低实现复杂性
        - 我们可以自下而上的
    - Subteties（重叠子问题）：在问题的求解过程中，很多子问题的解将被多次使用。
- 动态规划算法的设计步骤：
    - 分析优化解的结构
    - 递归地定义最优解的代价
    - 自底向上地计算优化解的代价保存之，并获取构造最优解的信息
    - 根据构造最优解的信息构造优化解
- 动态规划特点：
    - 把原始问题划分成一系列子问题；
    - 求解每个子问题仅一次，并将其结果保存在一个表中，以后用到时直接存取，不重复计算，节省计算时间
    - 自底向上地计算。
    - 整体问题最优解取决于子问题的最优解（状态转移方程）（将子问题称为状态，最终状态的求解归结为其他状态的求解）
- 背包九讲
- 最大回文子串（动态规划）
- 最长公共子序列（动态规划）
- 最长重复子序列（find和rfind的应用）
- 找零钱问题（动态规划&贪心算法）
- 排列组合问题（递归&回溯）

题目

排序算法

- 手写快速排序(快速排序的基准)

```
/*快速排序函数*/
//输入：待排序的数组,排序其实位置>=0,排序终止位置<=length(a) - 1
void QuickSortHelper(ElemType a[],int low,int high){
    if(low >= high)
        return;
    ElemType temp = a[low]; //存储序列首位元素
    int i = low + 1;
    int j = high;
    while(i != j){
        while(i < j && a[j] > temp)
            j--;
        while(i < j && a[i] < temp)
            i++;
        //交换两个元素
        if(i < j){
            swap(a,i,j);
        }
    }
    a[low] = a[i];
    a[i] = temp;
    QuickSortHelper(a,low,i -1);
    QuickSortHelper(a,i + 1,high);
}

/*快速排序*/
void QuickSort(ElemType a[], int n){
    QuickSortHelper(a,0,n-1);
}
```

- 归并排序
- 堆排序

```c
/*调整为最大堆*/
void PercDown(ElemType a[], int p, int n){
    // 将N个元素的数组中以a[p]为根的子堆调整为关于a[i]的最小堆
    int parent, child;
    int x;
    x = a[p]; //取出根节点的值

    for (parent = p; (parent * 2 + 1) < n; parent = child){
        child = parent * 2 + 1;
        if ((child != n - 1) && (a[child] < a[child + 1]))
            child++;
        if (x >= a[child])
            break;
        else
            a[parent] = a[child];
    }
    a[parent] = x;
}

/*堆排序*/
void HeapSort(ElemType a[], int n){
    for (int i = n / 2; i >= 0; i--)
        PercDown(a, i, n); //建立一个最大堆
    for (int i = n - 1; i > 0; i--){
        Swap(a, 0, i); //交换最大最小元素，把最大元素给a[i]
        PercDown(a, 0, i); //剩下的i个元素调整为最大堆
    }
}
```

大数据题

- 100万个32位整数，如何最快找到中位数。能保证每个数是唯一的，如何实现O(N)算法？
  1).内存足够时：快排
  2).内存不足时：分桶法：化大为小，把所有数划分到各个小区间，把每个数映射到对应的区间里，对每个区间中数的个数进行计数，数一遍各个区间，看看中位数落在哪个区间，若够小，使用基于内存的算法，否则 继续划分
- 十亿整数（随机生成，可重复）中前K最大的数

类似问题的解决方法思路：首先哈希将数据分成N个文件，然后对每个文件建立K个元素最小/大堆（根据要求来选择）。最后将文件中剩余的数插入堆中，并维持K个元素的堆。最后将N个堆中的元素合起来分析。可以采用归并的方式来合并。在归并的时候为了提高效率还需要建一个N个元素构成的最大堆，先用N个堆中的最大值填充这个堆，然后就是弹出最大值，指针后移的操作了。当然这种问题在现在的互联网技术中，一般就用map-reduce框架来做了。

大数据排序相同的思路：先哈希（哈希是好处是分布均匀，相同的数在同一个文件中），然后小文件装入内存快排，排序结果输出到文件。最后建堆归并。

- 十亿整数（随机生成，可重复）中出现频率最高的一千个