数据结构基础总结

数据结构

一些概念

> *数据结构就是研究数据的**逻辑结构**和**物理结构**以及它们之间**相互关系**，并对这种结构定义相应的运算，而且确保经过这些运算后所得到的新结构仍然是原来的结构类型。*

1. 数据：所有能被输入到计算机中，且能被计算机处理的符号的集合。是计算机操作的对象的总称。
2. 数据元素：数据（集合）中的一个"个体"，数据及结构中讨论的**基本**单位
3. 数据项：数据的不可分割的最小单位。一个数据元素可由若干个数据项组成。
4. 数据类型：在一种程序设计语言中，变量所具有的数据种类。整型、浮点型、字符型等等
5. 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
6. 物理结构/存储结构：数据在计算机中的表示。物理结构是描述数据具体在内存中的存储（如：顺序结构、链式结构、索引结构、哈希结构）等
7. 在数据结构中,从逻辑上可以将其分为线性结构和非线性结构
8. 数据结构的基本操作的设置的最重要的准则是,**实现应用程序与存储结构的独立**。实现应用程序是"逻辑结构"，存储的是"物理结构"。逻辑结构主要是对该结构操作的设定，物理结构是描述数据具体在内存中的存储（如：顺序结构、链式结构、索引结构、希哈结构）等。
9. 顺序存储结构中，线性表的逻辑顺序和物理顺序总是一致的。但在链式存储结构中，线性表的逻辑顺序和物理顺序一般是不同的。
10. 算法五个特性： 有穷性、确定性、可行性、输入、输出

11. 算法设计要求：正确性、可读性、健壮性、高效率与低存储量需求。(好的算法)
12. 算法的描述有伪程序、流程图、N-S结构图等。E-R图是实体联系模型，不是程序的描述方式。
13. 设计算法在执行时间时需要考虑：算法选用的规模、问题的规模
14. 时间复杂度：算法的执行时间与原操作**执行次数**之和成正比。时间复杂度有小到大：O(1)、O(logn)、O(n)、O(nlogn)、$O(n^2)$、$O(n^3)$。幂次时间复杂度有小到大$O(2^n)$、O(n!)、$O(n^n)$
15. 空间复杂度：若输入数据所占空间只取决于问题本身，和算法无关，则只需要分析**除输入和程序之外的辅助变量所占额外空间。**

线性表

线性表是一种典型的线性结构。头结点无前驱有一个后继，尾节点无后继有一个前驱。链表只能顺序查找，定位一个元素的时间为O(N)，删除一个元素的时间为O(1)

1. 线性表的顺序存储结构：把线性表的结点按逻辑顺序依次存放在一组地址连续的存储单元里。用这种方法存储的线性表简称顺序表。是一种随机存取的存储结构。顺序存储指内存地址是一块的，随机存取指访问时可以按下标随机访问，存储和存取是不一样的。如果是存储，则是指按顺序的，如果是存取，则是可以随机的，可以利用元素下标进行。数组比线性表速度更快的是：原地逆序、返回中间节点、选择随机节点。
   - 便于线性表的构造和任意元素的访问
   - 插入：插入新结点，之后结点后移。平均时间复杂度:O(n)
   - 删除：删除节点，之后结点前移。平均时间复杂度:O(n)
2. 线性链表：用一组任意的存储单元来依次存放线性表的结点，这组存储单元即可以是连续的，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的。因此，链表中结点的逻辑次序和物理次序不一定相同。为了能正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储指示其后继结点的地址。data域是数据域，用来存放结点的值。next是指针域（亦称链域），用来存放结点的直接后继的地址（或位置）。不需要事先估计存储空间大小。
   - **单链表**中每个结点的存储地址是存放在其前趋结点next域中，而开始结点无前趋，故应设头指针head指向开始结点。同时，由于最后一个结点无后继，故结点的指针域为空，即NULL。头插法建表(逆序)、尾插法建表(顺序)。增加头结点的目的是算法实现上的方便，但增大了内存开销。
     - 查找：只能从链表的头指针出发，顺链域next逐个结点往下搜索，直到搜索到第i个结点为止。因此，**链表不是随机存取结构。**
     - 插入：先找到表的第i-1的存储位置，然后插入。新结点先连后继，再连前驱。
     - 删除：首先找到a_{i-1}的存储位置p。然后令p->next指向a_i的直接后继结点，即把a_i从链上摘下。最后释放结点a_i的空间.r=p->next;p->next=r->next;delete r。
     - 判断一个单向链表中是否存在环的最佳方法是快慢指针。
   - 静态链表：用一维数组来实现线性链表，这种用一维数组表示的线性链表，称为静态链表。静态：体现在表的容量是一定的。（数组的大小）；链表：插入与删除同前面所述的动态链表方法相同。静态链表中指针表示的是下一元素在数组中的位置。

- 静态链表是用数组实现的，是顺序的存储结构，在物理地址上是连续的，而且需要预先分配大小。动态链表是用申请内存函数（C是malloc,C++是new）动态申请内存的，所以在链表的长度上没有限制。动态链表因为是动态申请内存的，所以每个节点的物理地址不连续，要通过指针来顺序访问。静态链表在插入、删除时也是通过修改指针域来实现的，与动态链表没有什么分别
- 循环链表：是一种头尾相接的链表。其特点是无须增加存储量，仅对表的链接方式稍作改变，即可使得表处理更加方便灵活。
  - 在单链表中，将终端结点的指针域NULL改为指向表头结点的或开始结点，就得到了单链形式的循环链表，并简单称为**单循环链表**。由于循环链表中没有NULL指针，故涉及遍历操作时，其终止条件就不再像非循环链表那样判断p或p—>next是否为空，而是判断它们是否等于某一指定指针，如头指针或尾指针等。
- 双向链表:在单链表的每个结点里再增加一个指向其直接前趋的指针域prior。这样就形成的链表中有两个方向不同的链。双链表一般由头指针唯一确定的，将头结点和尾结点链接起来构成循环链表，并称之为双向链表。设指针p指向某一结点，则双向链表结构的对称性可用下式描述：p—>prior—>next=p=p—>next—>prior。从两个方向搜索双链表，比从一个方向搜索双链表的方差要小。
  - 插入：先搞定插入节点的前驱和后继，再搞定后结点的前驱，最后搞定前结点的后继。
  - 在有序双向链表中定位删除一个元素的平均时间复杂度为O(n)
  - 可以直接删除当前指针所指向的节点。而不需要像单向链表中，删除一个元素必须找到其前驱。因此在插入数据时，单向链表和双向链表操作复杂度相同，而删除数据时，双向链表的性能优于单向链表

栈和队列

栈

栈(Stack)是限制在表的一端进行插入和删除运算的线性表，通常称插入、删除的这一端为栈顶(Top)，另一端为栈底(Bottom)。先进后出。top= -1时为空栈，top=0只能说明栈中只有一个元素，并且元素进栈时top应该自增

1. 顺序存储栈：顺序存储结构
2. 链栈：链式存储结构。插入和删除操作仅限制在链头位置上进行。栈顶指针就是链表的头指针。通常不会出现栈满的情况。不需要判断栈满但需要判断栈空。
3. 两个栈共用静态存储空间,对头使用也存在空间溢出问题。栈1的底在v[1]，栈2的底在V[m]，则栈满的条件是top[1]+1=top[2]。
4. 基本操作：删除栈顶元素、判断栈是否为空以及将栈置为空栈等
5. 对于n各元素的入栈问题，可能的出栈顺序有C(2n,n)/(n+1)个。
6. 堆栈溢出一般是循环的递归调用、大数据结构的局部变量导致的

应用，代码：

1. 进制转换
2. 括号匹配的检验

3. 行编辑程序
4. 迷宫求解：若当前位置“可通”，则纳入路径，继续前进;若当前位置“不可通”，则后退，换方向继续探索;若四周“均无通路”，则将当前位置从路径中删除出去。
5. 表达式求解：前缀、中缀、后缀。
    - 操作数之间的相对次序不变;
    - 运算符的相对次序不同;
    - 中缀式丢失了括弧信息，致使运算的次序不确定
    - 前缀式的运算规则为:连续出现的两个操作数和在它们之前且紧靠它们的运算符构成一个最小表达式
    - 后缀式的运算规则为:运算符在式中出现的顺序恰为表达式的运算顺序;每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式。
6. 实现递归：多个函数嵌套调用的规则是：后调用先返回。
7. 浏览器历史纪录，Android中的最近任务，Activity的启动模式，CPU中栈的实现，Word自动保存，解析计算式，解析xml/json。解析XML时，需要校验节点是否闭合，节点闭合的话，有头尾符号相对应，遇到头符号将其放入栈中，遇到尾符号时，弹出栈的内容，看是否有与之对应的头符号，栈的特性刚好符合符号匹配的就近原则。

不是所有的递归程序都需要栈来保护现场，比方说求阶乘的，是单向递归，直接用循环去替代从1乘到n就是结果了，另外一些需要栈保存的也可以用队列等来替代。不是所有的递归转化为非递归都要用到栈。转化为非递归主要有两种方法：对于尾递归或单向递归，可以用循环结构算法代替

队列

队列(Queue)也是一种运算受限的线性表。它只允许在表的一端进行插入，而在另一端进行删除。允许删除的一端称为队头(front)，允许插入的一端称为队尾(rear)。先进先出。

1. 顺序队列：顺序存储结构。当头尾指针相等时队列为空。在非空队列里，头指针始终指向队头前一个位置，而尾指针始终指向队尾元素的实际位置
2. 循环队列。在循环队列中进行出队、入队操作时，头尾指针仍要加1，朝前移动。只不过当头尾指针指向向量上界（MaxSize-1）时，其加1操作的结果是指向向量的下界0。除非向量空间真的被队列元素全部占用，否则不会上溢。因此，除一些简单的应用外，真正实用的顺序队列是循环队列。故队空和队满时头尾指针均相等。因此，我们无法通过front=rear来判断队列“空”还是“满”
3. 链队列：链式存储结构。限制仅在表头删除和表尾插入的单链表。显然仅有单链表的头指针不便于在表尾做插入操作，为此再增加一个尾指针，指向链表的最后一个结点。

4. 设尾指针的循环链表表示队列,则入队和出队算法的时间复杂度均为O(1)。用循环链表表示队列，必定有链表的头结点，入队操作在链表尾插入，直接插入在尾指针指向的节点后面，时间复杂度是常数级的；出队操作在链表表头进行，也就是删除表头指向的节点，时间复杂度也是常数级的。

5. 队空条件：rear==front，但是一般需要引入新的标记来说明栈满还是栈空，比如每个位置布尔值

6. 队满条件：(rear+1) % QueueSize==front，其中QueueSize为循环队列的最大长度
7. 计算队列长度：（rear-front+QueueSize）% QueueSize
8. 入队：（rear+1）% QueueSize
9. 出队：（front+1）% QueueSize
10. 假设以数组A[N]为容量存放循环队列的元素,其头指针是front,当前队列有X个元素,则队列的尾指针值为(front+X mod N)

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。

串的表示和实现：

1. 定长顺序存储表示。静态存储分配的顺序表。
2. 堆分配存储表示。存储空间是在程序执行过程中动态分配而得。所以也称为动态存储分配的顺序表
3. 串的链式存储结构。

串匹配：将主串称为目标串，子串称之为模式串。蛮力法匹配。KMP算法匹配。Boyer-Moore算法匹配。

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：

1. 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a^1,a^2,a^3,…,a^n的有限序列，其中a_i或者是原子项，或者是一个广义表。若广义表LS（n>=1)非空，则a^1是LS的表头，其余元素组成的表(a^2,…a^n)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。

例子：

1. A=（）——A是一个空表，其长度为零。
2. B=（e）——表B只有一个原子e，B的长度为1。
3. C=（a,(b,c,d))——表C的长度为2，两个元素分别为原子a和子表(b,c,d)。
4. D=（A，B，C）——表D的长度为3，三个元素都是广义 表。显然，将子表的值代入后，则有D=(( ),(e),(a,(b,c,d)))。
5. E=（a,E）——这是一个递归的表，它的长度为2，E相当于一个无限的广义表E=(a,(a,(a,(a,…)))).

三个结论：

1. 广义表的元素可以是子表，而子表的元素还可以是子表。由此，广义表是一个多层次的结构，可以用图形象地表示
2. 广义表可为其它表所共享。例如在上述例4中，广义表A，B，C为D的子表，则在D中可以不必列出子表的值，而是通过子表的名称来引用。
3. 广义表的递归性

考点：

1. 广义表是0个或多个单因素或子表组成的有限序列，广义表可以是自身的子表，广义表的长度n>=0，所以可以为空表。广义表的**同级**元素(直属于同一个表中的各元素)具有**线性**关系
2. 广义表的表头为空，并不代表该广义表为空表。广义表()和(())不同。前者是长度为0的空表，对其不能做求表头和表尾的运算；而后者是长度为l的非空表(只不过该表中惟一的一个元素是空表)，对其可进行分解，得到的表头和表尾均是空表()
3. 已知广义表LS＝((a,b,c),(d,e,f)),运用head和tail函数取出LS中原子e的运算是head(tail(head(tail(LS)))。根据表头、表尾的定义可知：任何一个非空广义表的表头是表中第一个元素，它可以是原子，也可以是子表，而其**表尾必定是子表**。也就是说，广义表的head操作，取出的元素是什么，那么结果就是什么。但是tail操作取出的元素外必须加一个表——“()”。tail(LS)＝((d,e,f))；head(tail(LS))=(d,e,f)；tail(head(tail(LS)))=(e,f)；head(tail(head(tail(LS))))=e。
4. 二维以上的数组其实是一种特殊的广义表
5. 在（非空）广义表中：1、表头head可以是原子或者一个表 2、表尾tail一定是一个表 3.广义表难以用顺序存储结构 4.广义表可以是一个多层次的结构

树和二叉树

一种**非线性**结构。树是递归结构，在树的定义中又用到了树的概念。

基本术语：

1. 树结点：包含一个数据元素及若干指向子树的分支；
2. 孩子结点：结点的子树的根称为该结点的孩子；
3. 双亲结点：B结点是A结点的孩子，则A结点是B结点的双亲；
4. 兄弟结点：同一双亲的孩子结点；
5. 堂兄结点：同一层上结点；
6. 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；
7. 树的高（深）度：树中最大的结点层
8. 结点的度：结点子树的个数
9. 树的度： 树中最大的结点度。
10. 叶子结点：也叫终端结点，是度为0的结点；
11. 分枝结点：度不为0的结点（非终端结点）；
12. 森林：互不相交的树集合；
13. 有序树：子树有序的树，如：家族树；
14. 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**、**二叉平衡(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：

1. 在二叉树的第 i 层上至多有2^{i-1}个结点。
2. 深度为 k 的二叉树上至多含 2^k-1 个结点（k≥1）
3. 对任何一棵二叉树，若它含有n_0个叶子结点、n_2个度为 2 的结点，则必存在关系式：$n_0= n_2+1$。
4. 具有 n 个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor+1$ 。
5. n个结点的二叉树中，完全二叉树具有最小的路径长度。
6. 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

1. 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。
2. 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
3. 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。
4. 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。

遍历二叉树和线索二叉树

遍历二叉树：使得每一个结点均被访问一次，而且仅被访问一次。非递归的遍历实现要利用栈。

- 先序遍历DLR：根节点->左子树->右子树
- 中序遍历LDR：左子树->根节点->右子树。必须要有中序遍历才能得到一棵二叉树的正确顺序
- 后续遍历LRD：左子树->右子树->根节点。需要栈的支持。
- 层次遍历：用一维数组存储二叉树时,总是以层次遍历的顺序存储结点。层次遍历应该借助队列。

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**。n个结点的线索二叉树上每个结点有2个指针域（指向左孩子和右孩子），总共有2n个指针域；一个n个结点的树有n-1条边，那么空指针域= 2n - (n-1) = n + 1，即线索数为n+1。指针域tag为0，存放孩子指针，为1，存放前驱/后继节点指针。

线索树下结点x的前驱与后继查找：设结点x相应的左（右）标志是线索标志，则lchild(rchild)就是前驱(后继），否则：

- LDR–前驱：左子树中最靠右边的结点；后继：右子树中最靠左边的结点
- LRD–前驱：右子树的根，若无右子树，为左子树跟。后继：x是根，后继是空；x是双亲的右孩子、x是双亲的左孩子，但双亲无右孩子，双亲是后继；x是双亲的左孩子，双亲有右孩子，双亲右子树中最左的叶子是后继
- DLR–对称于LRD线索树一将LRD中所有左右互换，前驱与后继互换，得到DLR的方法。

- 为简化线索链表的遍历算法，仿照线性链表，为线索链表加上一头结点，约定：
  - 头结点的lchild域：存放线索链表的根结点指针；
  - 头结点的rchild域：中序序列最后一个结点的指针；
  - 中序序列第一结点lchild域指向头结点；
  - 中序序列最后一个结点的rchild域指向头结点；

中序遍历的线索二叉树以及线索二叉树链表示意图

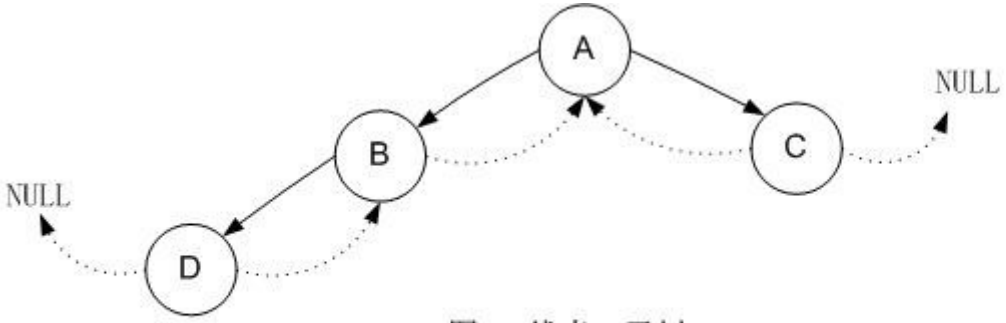


图1：线索二叉树

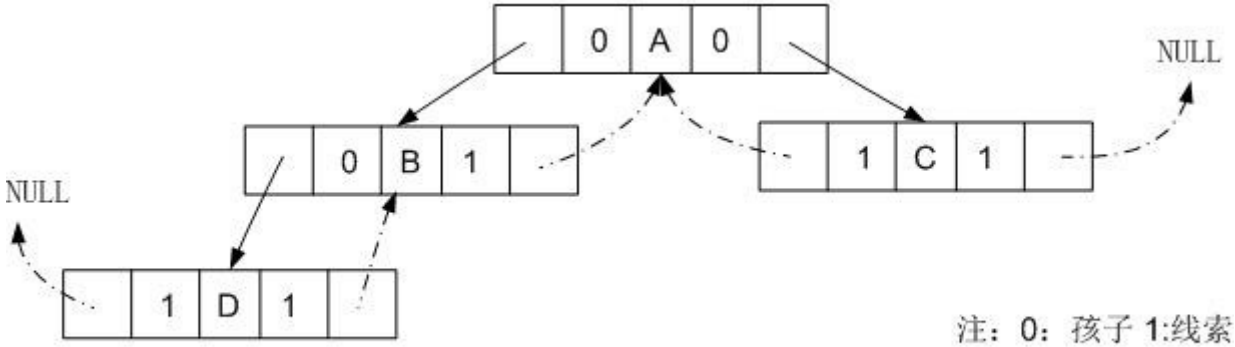


图2:线索二叉链表

一棵左右子树均不空的二叉树在前序线索化后,其中空的链域的个数是1。**前序和后续线索化后空链域个数都是1，中序是2。**二叉树在线索化后，仍不能有效求解的问题是前序求前序先驱，后序求后序后继。

中序遍历的顺序为：左、根、右，所以对于每一非空的线索，左子树结点的后继为根结点，右子树结点的前驱为根结点，再递归的执行上面的过程，可得非空线索均指向其祖先结点。**在中序线索二叉树中,每一非空的线索均指向其祖先结点。**

在二叉树上加上结点前趋、后继线索后，可利用线索对二叉树进行遍历,此时，**不需栈，也不需递归。**基本步骤：

1. p=T->lchild; p指向线索链表的根结点;
2. 若线索链表非空，循环:
   - 循环，顺着p左孩子指针找到最左下结点；访问之;
   - 若p所指结点的右孩子域为线索，p的右孩子结点即为后继结点循环： p=p->rchild； 并访问p所指结点；（在此循环中，顺着后继线索访问二叉树中的结点)
   - 一旦线索"中断"，p所指结点的右孩子域为右孩子指针，p=p->rchild，使 p指向右孩子结点;

树和森林

树的存储结构:

1. 双亲表示法
2. 孩子表示法
3. 利用图表示树
4. 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 $WPL=\sum w_i \times l_i$
7. 哈夫曼树：假设有n个权值($w_1, w_2, \dots, w_n$)，构造有n个叶子结点的二叉树，每个叶子结点有一个 w_i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。

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。

图遍历与回溯

图搜索->形成搜索树

1. 穷举法。
2. 贪心法。多步决策，每步选择使得构成一个问题的可能解，同时满足目标函数。
3. 回溯法。根据题意，选取度量标准，然后将可能的选择方法按度量标准所要求顺序排好，每次处理一个量，得到该意义下的最优解的分解处理。

图

无向图

1. 回路或环：第一个顶点和最后一个顶点相同的路径。
2. 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
3. 连通：顶点v至v’ 之间有路径存在
4. 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。
5. 连通分量：极大连通子图，子图中包含的顶点个数极大
6. 所有顶点度的和必须为偶数

有向图:

1. 回路或环：第一个顶点和最后一个顶点相同的路径。

2. 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。
3. 连通：顶点v至v'之间有路径存在
4. 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。
5. 强连通分量：极大连通子图
6. 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
7. 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。
8. 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。
9. 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。
10. 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。

图的存储形式

1. 邻接矩阵和加权邻接矩阵
   - 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∞
2. 邻接表
   - 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

图的遍历

深度优先搜索利用栈，广度优先搜索利用队列

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。

1. 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。
2. 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)。
3. 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。

双连通图和关节点

若从一个连通图中删去任何一个顶点及其相关联的边，它仍为一个连通图的话，则该连通图被称为**重（双）连通图。**

若连通图中的某个顶点和其相关联的边被删去之后，该连通图被分割成两个或两个以上的连通分量，则称此顶点为**关节点。**

没有关节点的连通图为双连通图

1. 若生成树的根结点，有两个或两个以上的分支，则此顶点(生成树的根)必为关节点；
2. 对生成树上的任意一个非叶“顶点”，若其某棵子树中的所有“顶点”没有和其祖先相通的回边，则该“顶点”必为关节点。

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v_i是v_j前驱，则v_i一定在v_j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：

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(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。~~广度优先搜索~~过程中如果访问到一个已经访问过的节点，可能是多个节点指向这个节点，不一定是存在环。

算法描述：

1. 把邻接表中入度为0的顶点依此进栈
2. 若栈不空，则
   - 栈顶元素v_j退栈并输出；
   - 在邻接表中查找v_j的直接后继v_k，把v_k的入度减1；若v_k的入度为0则进栈
3. 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：

1. 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。
2. 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。
3. 活动a_i由弧

查找

顺序查找、折半查找、索引查找、分块查找是静态查找，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

静态查找表

顺序表的顺序查找：应用范围：顺序表或线性链表表示的表，表内元素之间无序。查找过程：从表的一端开始逐个进行记录的关键字和给定值的比较。

顺序有序表的二分查找。平均查找时间$(n+1)/n \log_2(n+1)$

分块查找：将表分成几块，块内无序，块间有序，即前一块中的最大值小于后一块中的最小值。并且有一张索引表，每一项存放每一块的最大值和指向该块第一个元素的指针。索引表有序，块内无序。所以，块间查找用二分查找，块内用顺序查找，效率介于顺

序和二分之间；先确定待查记录所在块，再在块内查找。因此跟表中元素个数和块中元素个数都有关。

1. 用数组存放待查记录，
2. 建立索引表，由每块中最大（小）的关键字及所属块位置的信息组成。
3. 当索引表较大时，可以采用二分查找
4. 在数据量极大时，索引可能很多，可考虑建立索引表的索引，即二级索引，原则上索引不超过三级

分块查找平均查找长度：$ASL_{bs} = L_b + L_w$。其中，L_b是查找索引表确定所在块的平均查找长度， L_w是在块中查找元素的平均查找长度。在n一定时，可以通过选择s使ASL尽可能小。当s=sqrt(n)时，ASL最小。

1. 时间：顺序查找最差，二分最好，分块介于两者之间
2. 空间：分块最大，需要增加索引数据的空间
3. 顺序查找对表没有特殊要求
4. 分块时数据块之间在物理上可不连续。所以可以达到插入、删除数据只涉及对应的块；另外，增加了索引的维护。
5. 二分查找要求表有序，所以若表的元素的插入与删除很频繁，维持表有序的工作量极大。
6. 在表不大时，一般直接使用顺序查找。

动态查找

二叉排序树的结点删除：

1. x为叶子结点，则直接删除
2. x只有左子树x_L或只有右子树x_R ,则令x_L或x_R直接成为双亲结点f的子树；
3. x即有左子树x_L也有右子树x_R，在x_L中选值最大的代替x，该数据按二叉排序树的性质应在最右边。

平衡二叉树：每个结点的平衡因子都为 1、 - 1、0 的二叉排序树。或者说每个结点的左右子树的高度最多差1的二叉排序树。

平衡二叉树的平衡：

1. 左调整(新结点插入在左子树上的调整)：
   - LL(插入在结点左子树的左子树上)：旋转前后高度都为h+1
   - LR(新插入结点在左子树的右子树上)：旋转前后高度仍为h+1
2. 右调整(新结点插入在右子树上进行的调整)：
   - RR(插入在的右子树的右子树上)：处理方法和 LL对称
   - RL(插入在的右子树的左子树上)：处理方法和 LR对称

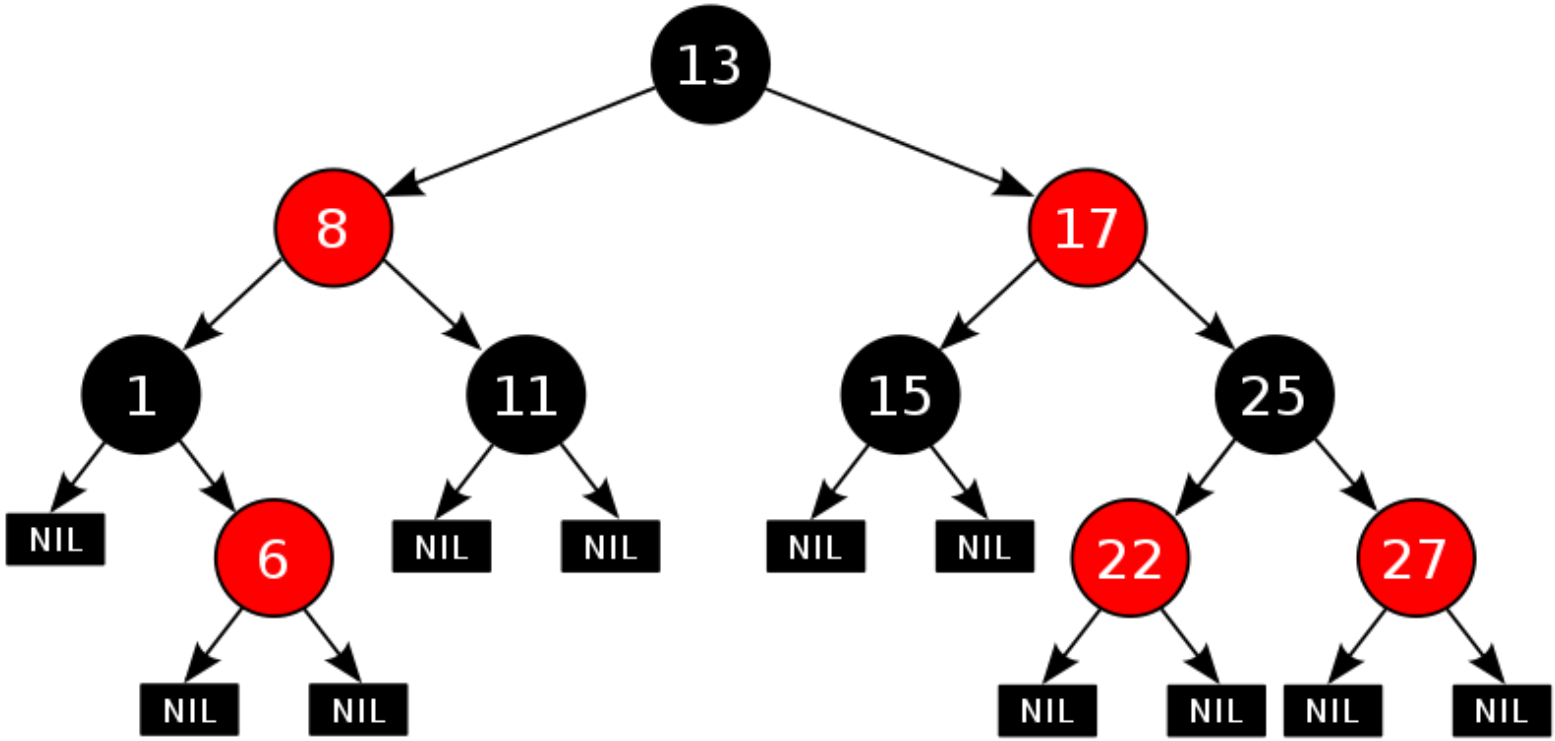
平衡树建立方法：

1. 按二叉排序树插入结点

2. 如引起结点平衡因子变为|2|，则确定旋转点，该点是离根最远（或最接近于叶子的点）
3. 确定平衡类型后进行平衡处理，平衡后以平衡点为根的子树高不变
4. 最小二叉平衡树的节点的公式如下 F(n)=F(n-1)+F(n-2)+1 这个类似于一个递归的数列，可以参考Fibonacci数列，1是根节点，F(n-1)是左子树的节点数量，F(n-2)是右子树的节点数量。

常见的平衡二叉树：

1. 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。
2. avl树也是自平衡二叉树；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
3. 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)；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。
2. 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。
3. 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

1. 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。二叉排序树查找，最优二叉树查找，键树查找，哈希法查找是动态查找。分块、顺序、折半、索引顺序查找均为静态。分块法应该是将整个线性表分成若干块进行保存，若动态变化则可以添加在表的尾部（非顺序结构），时间复杂度是O(1)，查找复杂度为O(n)；若每个表内部为顺序结构，则可用二分法将查找时间复杂度降至O(logn)，但同时动态变化复杂度则变成O(n)；顺序法是挨个查找，这种方法最容易实现，不过查找时间复杂度都是O(n)，动态变化时可将保存值放入线性表尾部，则时间复杂度为O(1)；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）
2. 二叉树的平均查找长度为$O(\log_2 n)$——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$[\log_2 n]+1$,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$[\log_2 n]+1$——n.
3. 链式存储不能随机访问，必须是顺序存储

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：

1. 树中每个结点最多有m棵子树
2. 根结点在不是叶子时，至少有两棵子树
3. 除根外，所有非终端结点至少有$\lceil m/2 \rceil$棵子树
4. 有s个子树的非叶结点具有 n = s-1个关键字，结点的信息组织为:$(n,A_0,K_1,A_1,K_2,A_2 \dots K_n, A_n)$。这里：n为关键字的个数，$k_i$ (i=1,2,…,n)为关键字，且满足K_i小于K_{i+1},，A_i(i=0,1,..n)为指向子树的指针。
5. 所有的叶子结点都出现在同一层上，不带信息（可认为外部结点或失败结点）。
6. 关键字集合分布在整颗树中

7. 任何一个关键字出现且只出现在一个结点中
8. 搜索有可能在非叶子结点结束
9. 其搜索性能等价于在关键字全集内做一次二分查找
10. 只适用于随机检索，不适用于顺序检索。
11. 有结点的平衡因子都为零
12. M阶B-树中含有N个关键字，最大深度为$\log_{\lceil m/2 \rceil}((n+1)/2+2)$

B_树中结点的插入

1. m代表B_树的阶，插入总发生在最低层
2. 插入后关键字个数小于等于 m-1,完成。
3. 插入后关键字个数等于m,结点分裂，以中点数据为界一分为二，中点数据放到双亲结点中。这样就有可能使得双亲结点的数据个数为m,引起双亲结点的分裂，最坏情况下一直波及到根，引起根的分裂——B_树长高。

3阶B_树的插入。每个结点最多3棵子树，2个数据；最少2棵子树，1个数据。所以3阶B_树也称为2-3树。

B_树中结点的删除

1. 删除发生在最底层
   - 被删关键字所在结点中的关键字数目大于等于 m/2 ，直接删除。
   - 删除后结点中数据为$\lceil m/2 \rceil-2$，而相邻的左（右）兄弟中数据大于$\lceil m/2 \rceil-1$，此时左（右兄弟）中最大（小）的数据上移到双亲中，双亲中接（靠）在它后（前）面的数据移到被删数据的结点中
   - 其左右兄弟结点中数据都是$\lceil m/2 \rceil-1$，此时和左（右）兄弟合并，合并时连同双亲中相关的关键字。此时，双亲中少了一项，因此又可能引起双亲的合并，最坏一直到根，使B-树降低一层。
2. 删除不在最底层
   - 在大于被删数据中选最小的代替被删数据，问题转换成在最底层的删除

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。

 B+树

差异:

1. 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。
2. 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)
3. 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)

4. 非叶最底层顺序联结，这样可以进行顺序查找

B+特性

1. 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；
2. 不可能在非叶子结点命中
3. 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
4. 更适合文件索引系统
5. 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)

查找过程

- 在 B+ 树上，既可以进行缩小范围的查找，也可以进行顺序查找；
- 在进行缩小范围的查找时，不管成功与否，都必须查到叶子结点才能结束；
- 若在结点内查找时，给定值≤K_i， 则应继续在 A_i 所指子树中进行查找

插入和删除的操作：类似于B_树进行，即必要时，也需要进行结点的“分裂”或“合并”。

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

1. 在记录的存储地址和它的关键字之间建立一个确定的对应关系；这样不经过比较，一次存取就能得到元素。
2. 哈希函数——在记录的关键字与记录的存储位置之间建立的一种对应关系。是从关键字空间到存储位置空间的一种映象。

3. 哈希表——应用哈希函数，由记录的关键字确定记录在表中的位置信息，并将记录根据此信息放入表中，这样构成的表叫哈希表。
4. Hash查找适合于关键字可能出现的值的集合远远大于实际关键字集合的情形。
5. 更适合查找，不适合频繁更新
6. Hash表等查找复杂依赖于Hash值算法的有效性，在最好的情况下，hash表查找复杂度为O(1)。只有无冲突的hash_table复杂度才是O(1)。一般是O(c)，c为哈希关键字冲突时查找的平均长度。插入，删除，查找都是O(1)。平均查找长度不随表中结点数目的增加而增加,而是随负载因子的增大而增大
7. 由于冲突的产生，使得哈希表的查找过程仍然是一个给定值与关键字比较的过程。

根据抽屉原理，冲突是不可能完全避免的，所以，选择好的**散列函数和冲突处理**方法：

1. 构造一个性能好，冲突少的Hash函数
2. 如何解决冲突

常用的哈希函数

1. 直接定址法。仅适合于：地址集合的大小 == 关键字集合的大小
2. 数字分析法。对关键字进行分析，取关键字的若干位或其组合作哈希地址。仅适合于：能预先估计出全体关键字的每一位上各种数字出现的频度。
3. 平方取中法。以关键字的平方值的中间几位作为存储地址。
4. 折叠法。将关键字分割成位数相同的几部分，然后取这几部分的叠加和（舍去进位）做哈希地址。移位叠加/间界叠加。适合于：关键字的数字位数特别多，且每一位上数字分布大致均匀情况。
5. 除留余数法。取关键字被某个不大于哈希表表长m的数p除后所得余数作哈希地址，即H(key)=key%p，p<=m。
6. 随机数法。取关键字的伪随机函数值作哈希地址，即H(key)=random(key)，适于关键字长度不等的情况。

冲突解决

1. 开放定址法。当冲突发生时，形成一个探查序列；沿此序列逐个地址探查，直到找到一个空位置（开放的地址），将发生冲突的记录放到该地址中。即$H_i=(H(key)+d_i)$ % m，i=1,2,……k(k<=m-1)，H(key)哈希函数，m哈希表长，d_i增量序列。缺点：删除：只能作标记，不能真正删除；溢出；载因子过大、解决冲突的算法选择不好会发生聚集问题。要求装填因子α较小，故当结点规模较大时会浪费很多空间。
   - 线性探测再散列：d_i=1，2，3，…，m-1
   - 二次探测再散列：$d_i=1^2,-1^2,2^2,-2^2,…，±k^2$ (k<=m/2)
   - 伪随机探测再散列：d_i为伪随机数序列
2. 链地址法：将所有关键字为同义词的记录存储在一个单链表中，并用一维数组存放头指针。拉链法中可取α≥1，且结点较大时，拉链法中增加的指针域可忽略不计，因此节省空间。一旦发生冲突，在当前位置给单链表增加结点就行。
3. 其他方法：再哈希法、建立公共溢出区

4. 在用拉链法构造的散列表中，删除结点的操作易于实现。拉链法的缺点是：指针需要额外的空间，故当结点规模较小时，开放定址法较为节省空间。由于拉链法中各链表上的结点空间是动态申请的,故它更适合于造表前无法确定表长的情况。拉链法解决冲突时，需要使用指针，指示下一个元素的存储位置
5. 开哈希表–链式地址法;闭哈希表–开放地址法.开哈希和闭哈希主要的区别在于，随着哈希表的密集度提高，使用闭哈希时，不仅会与相同哈希值的元素发生冲突，还容易与不同哈希值的元素发生冲突；而开哈希则不受哈希表疏密与否的影响，始终只会与相同哈希值的元素冲突而已。所以在密集度变大的哈希表中查找时，显然开哈希的平均搜索长度不会增长。
6. 设有n个关键字具有相同的Hash函数值，则用线性探测法把这n个关键字映射到Hash表中需要做n*(n-1)/2次线性探测。如果使用二次探测再散列法将这n个关键字存入哈希表，至少要进行n*(n+1)/2次探测

Hash查找效率：装填因子=表中记录数/表容量

有B+Tree/Hash_Map/STL Map三种数据结构。对于内存中数据，查找性能较好的数据结构是Hash_Map，对于磁盘中数据，查找性能较好的数据结构是B+Tree。Hash操作能根据散列值直接定位数据的存储地址，设计良好的hash表能在常数级时间下找到需要的数据，但是更适合于内存中的查找。B+树是一种是一种树状的数据结构，适合做索引，对磁盘数据来说，索引查找是比较高效的。STL_Map的内部实现是一颗红黑树，但是只是一颗在内存中建立二叉树树，不能用于磁盘操作，而其内存查找性能也比不上Hash查找。

内部排序

1. 内部排序：全部数据可同时放入内存进行的排序。
2. 外部排序：文件中数据太多，无法全部调入内存进行的排序。

插入类：

1. 直接插入排序。最坏情况是数据递减序，数据比较和移动量最大，达到$O(n^2)$，最好是数据是递增序，比较和移动最少为O(n)。趟数是固定的n-1，即使有序，也要依次从第二个元素开始。排序趟数不等于时间复杂度。
2. 折半插入排序 。由于插入第i个元素到r[1]到r[i-1]之间时，前i个数据是有序的，所以可以用折半查找确定插入位置，然后插入。
3. 希尔排序。缩小增量排序。5-3-1。在实际应用中，步长的选取可简化为开始为表长n的一半（n/2），以后每次减半，最后为1。插入的改进，最后一趟已基本有序，比较次数和移动次数相比直接插入最后一趟更少

交换类：

1. 冒泡排序。$O(n^2)$通常认为冒泡是比较差的，可以加些改进，比如在一趟中无数据的交换，则结束等措施。
    - 在数据已基本有序时，冒泡是一个较好的方法
    - 在数据量较少时（15个左右）可以用冒泡
2. 快速排序。

- 时间复杂度。最好情况：每次支点总在中间，O(nlog$_2$n)，平均O(nlog$_2$n)。最坏，数据已是递增或递减，O(n^2)。pivotkey的选择越靠近中央，即左右两个子序列长度越接近，排序速度越快。越无序越快。
- 空间复杂度。需栈空间以实现递归，最坏情况：S(n)=O(n)；一般情况：S(n)=O(log$_2$n)
- 在序列已是有序的情况下，时间复杂度最高。原因：支点选择不当。改进：随机选取支点或最左、最右、中间三个元素中的值处于中间的作为支点，通常可以避免最坏情况。所以，快速排序在表已基本有序的情况下不合适。
- 在序列长度已较短时，采用直接插入排序、起泡排序等排序方法。序列的个数通常取10左右。

选择类排序：

1. 简单选择排序。O(n^2)。总比较次数n(n-1)/2。
2. 堆排序。建堆 O(n)，筛选排序O(nlogn)。找出若干个数中最大/最小的前K个数，用堆排序是最好。小根堆中最大的数一定是放在叶子节点上，堆本身是个完全二叉树，完全二叉树的叶子节点的位置大于[n/2]。时间复杂度不会因为待排序序列的有序程度而改变，但是待排序序列的有序程度会影响比较次数。
3. 归并排序。时间：与表长成正比，若一个表表长是m，另一个是n，则时间是O(m+n)。单独一个数组归并，时间：O(nlogn)，空间：O(n)，比较次数介于(nlogn)/2和(nlogn)-n+1，赋值操作的次数是(2nlogn)。归并排序算法比较占用内存，但却是**效率高且稳定**的排序算法。在外排序中使用。归并的趟数是logn。
4. 基数排序。在一般情况下，每个结点有 d 位关键字，必须执行 t = d次分配和收集操作。分配的代价：O(n)；收集的代价：O(rd) （rd是基数）；总的代价为：O(d ×(n + rd))。适用于以数字和字符串为关键字的情况。
5. 枚举排序，通常也被叫做秩排序，比较计数排序。对每一个要排序的元素，统计小于它的所有元素的个数，从而得到该元素在整个序列中的位置，时间复杂度为O(n^2)

比较法分类的下界：O(nlogn)

排序算法的一些特点：

1. **堆排序、冒泡排序、快速排序**在每趟排序过程中,都会有一个元素被放置在其最终的位置上。
2. 有字符序列 {Q,H,C,Y,P,A,M,S,R,D,F,X} ,新序列{F,H,C,D,P,A,M,Q,R,S,Y,X}，是快速排序算法一趟扫描的结果。(*拿Q作为分割点,快速排序一轮。二路归并，第一趟排序，得到 n / 2 个长度为 2 的各自有序的子序列，第二趟排序，得到 n / 4 个长度为 4 的各自有序的子序列H Q C Y A P M S D R F X。如果是快速排序的话，第一个元素t将会被放到一个最准确的位置，t前的数均小于t，后面的数均大于t。希尔排序每个小分组内将会是有序的。堆排序，把它构成一颗二叉树的时候，该堆要么就是大根堆，要么就是小根堆，第一趟Y排在最后；冒泡，那么肯定会有数据下沉的动作，第一趟有A在第一位。*)
3. 在文件"局部有序"或文件长度较小的情况下,最佳内部排序的方法是**直接插入排序。** (归并排序要求待排序列已经部分有序，而部分有序的含义是待排序列由若干有序的子序列组成，即每个子序列必须有序，并且其时间复杂度为O(nlog2n)；直接插入排序在待排序列基本有序时，每趟的比较次数大为降低，即n-1趟比较的时间复杂度由O(n^2)降至O(n)。在待排序的元素序列基本有序或者每个元素距其最终位置不远也可用插入排序，效率最高的排序方法是**插入排序**)
4. 排序趟数与序列的原始状态有关的排序方法是优化冒泡和快速排序法。(插入排序和选择排序不管序列的原始状态是什么都要执行n-1趟，优化冒泡和快排不一定。仔细理解排序的次数和比较次数的区别)
5. **不稳定的排序方法：快排，堆排，希尔，选择**

6. 要与关键字的初始排列次序无关,那么就是最好、最坏、一般的情况下排序时间复杂度不变, 总共有堆排序,归并排序,选择排序,基数排序
7. 快速排序、Shell 排序、归并排序、直接插入排序的关键码比较次数与记录的初始排列有关。折半插入排序、选择排序无关。(直接插入排序在完全有序的情况下每个元素只需要与他左边的元素比较一次就可以确定他最终的位置；折半插入排序，比较次数是固定的，与初始排序无关；快速排序，初始排序不影响每次划分时的比较次数，都要比较n次，但是初始排序会影响划分次数，所以会影响总的比较次数，但快排平均比较次数最小；归并排序在归并的时候，如果右路最小值比左路最大值还大，那么只需要比较n次，如果右路每个元素分别比左路对应位置的元素大，那么需要比较2*n-1次，所以与初始排序有关)
8. 精俭排序，即一对数字不进行两次和两次以上的比较，插入和归并是"精俭排序"。插入排序，前面是有序的，后面的每一个元素与前面有序的元素比较，比较过的就是有序的了，不会再比较一次。归并每次合并后，内部都是有序的，内部的元素之间不用再比较。选择排序，每次在后面的元素中找到最小的，找最小元素的过程是在没有排好序的那部分进行，所有肯定会比较多次。堆排序也需比较多次。

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(n^2))；Prim求最小生成树邻接表存储时是O(n+e),图O(n_2)；关键路径及关键活动的求法。
2. 回溯法

3. 分支限界法
4. 分治法。分割、求解、合并。二分查找、归并排序、快速排序。
5. 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$O(n^3)$

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$O(n^2)$，$O(n^3)$等，可以解决相当大的信息量。（数塔在n<=100层时，可以在很短的时间内得到问题解）

- 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：

1. 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。
2. 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。
3. 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

1. 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。一-确定性问题
2. 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$O(n^k)$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。
3. 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。