# 基本概念

* 1. 数据项（字段） → 数据元素（记录） → 数据对象（表）。三者都是数据
  2. 数据结构：逻辑结构和存储结构
     1. 逻辑结构表示：图表、二元组
        1. 二元组：B=(D, R)
           1. D：数据元素集合 R：关系序偶集合，前/后驱元素、第一/二元素、开始/终端元素。
           2. 例如：D = { 010, 021, 027 } R = { r } r = {<010, 021>,<021, 027>}
     2. 逻辑结构类型：集合（数据元素无关系）、线性结构（一对一）、树形结构（一对多）、图形结构（多对多）
     3. 存储结构类型：顺序存储结构（逻辑相邻元素物理也相邻）、链式存储结构（逻辑相邻元素物理不一定相邻）~~、~~
     4. ~~索引存储结构、哈希（散列）存储结构~~
  3. 数据类型（值的集合和一组操作的总称）、抽象数据类型（数据对象、数据关系、基本操作）
  4. 算法五特性：有穷性、确定性、可行性、输入、输出。
  5. 算法分析：
     1. 时间复杂度：所有语句频度（语句在算法中被重复执行的次数）之和，用表示
        1. 求和定理、求积定理
     2. 空间复杂度：形参在调用函数时计算，函数体内不计。O(1)算法称为原地/就地工作算法

# 线性表

* 1. 线性表的每个数据元素由逻辑序号唯一确定。有穷性、一致性、序列性。位序从1开始
  2. 顺序表：一块连续的存储空间（逻辑相邻的存储位置相邻、线性表的直接映射）
     1. 特点：存储密度大、随机存取特性；插入删除要移动大量元素、分配空间固定
  3. 链表：空间动态管理的链式存储结构（逻辑相邻的存储不一定相邻）
     1. 结点：LinkNode、后继结点（next）、前驱结点（prior）
     2. 单链表：
     3. 首结点数据域一般无意义、一般不删除
     4. 单链表增加首结点的优点：便于首元结点的处理、空/非空表处理过程统一
     5. 单链表建立：头插法（倒序）、尾插法（顺序）
     6. 特点：存储密度小、无随机存取特性；插入删除不移动元素、动态分配空间
     7. 双链表、循环单链表、循环双链表（尾节点连到首结点，为空时首节点的前驱和后继都是自己）

# 栈

* 1. 栈为只能在一端进行插入和删除的线性表
  2. 在**栈顶**进栈、出栈（push、pop）、后进先出。栈顶指针指向空
  3. ~~n个元素的出栈序列可能数：~~
  4. 顺序栈：顺序存储结构的栈
     1. 共享栈：两头作栈底，用两个指针指示栈顶。
  5. 链栈：链式存储结构的栈（不存在栈满上溢出），采用头插法创建。
  6. 作用：对称匹配、中缀表达式转换为后缀表达式、后缀表达式的运算、回溯已经过的数组（迷宫问题）

# 队列

* 1. 队列为只能在一端进行插入、在另一端删除的线性表
  2. 从**队尾(rear)**进队\入队(enqueue)、从**队首\头(front)**出队\离队(dequeue)、先进先出。rear指向空
  3. 顺序队：顺序存储结构的队列、存在假溢出
  4. 环形队（循环队）：顺序存储结构的环形队列
     1. 队首指针-队尾指针法：为判定队满队空而始终空出一个元素空间
     2. 队首指针-元素个数法：无需空出元素空间
  5. 链队：链式存储结构的队列
     1. 单链表：需要队首指针指向头结点和队尾指针指向尾结点
     2. ~~循环链表：仅需队尾指针指向链表尾~~
  6. ~~双端队列：队列两端都可进队与出队~~

# 串

* 1. 串(string)由零个或多个字符组成的有限序列，用s表示。
  2. 串一般表现为、空串、子串、串相等。空串是任意串的子串，一个串是自身的子串
  3. 顺序串：串的顺序存储结构
     1. ~~紧缩格式（存储密度大、运算效率低）、非紧缩格式（存储密度小、运算效率高）~~
     2. 静态存储（定长顺序存储）、动态存储（堆分配存储）
  4. 链串：串的链式存储结构
  5. 串的模式匹配：在目标串**(target string)**中找到模式串**(pattern string)**
     1. Brute-Force（暴力）算法：简单穷举匹配
     2. KMP算法

# 递归

* 1. 直接递归、间接递归
  2. 尾递归：函数的最后一条语句是递归语句。可用循环语句转化为非递归函数、其他递归算法可用栈转化。
  3. 使用场景：定义是递归的、数据结构是递归的、求解方法是递归的
  4. 递归模型：由递归出口、递归体组成
  5. 栈帧：单个函数调用操作所使用的函数调用帧。函数调用时创建栈帧，保存数据并压入调用栈。

# 数组

* 1. n维数组可以看作是由n-1维数组作为数据元素的线性表。
  2. 基本操作：初始化、销毁、读取、写入
  3. 二维数组：按行优先存放、按列优先存放
  4. 特殊矩阵的压缩存放：对称矩阵、上下三角矩阵、对角矩阵、稀疏矩阵。
  5. 广义表：非线性，是线性表的推广，有限元素的序列。
     1. 原子：广义表中不可分解的元素，如整型等
     2. 子表：广义表中的广义表
     3. 广义表中的元素有相对次序
     4. 广义表的长度为最外层包含元素的个数、深度为所含括弧的重数
     5. 广义表可以共享，广义表可以是自己的子表（可递归）
     6. 广义表只能采用链式存储

# 树

* 1. D是包含n个结点的有限集合（n≥0）。R为关系序偶集合
  2. **空树**：n=0 **根节点**：无前驱结点的结点 其他结点有且仅有一个前驱结点 后继结点无限制
  3. 树的逻辑表示：树形表示法、文氏图表示法、凹入表示法、括号表示法
  4. **结点的度**：结点的子树的个数。 **树的度**：树中各结点的度的最大值。 **n次(叉)树**：度为n的树
  5. **分支（非终端）结点**：度不为0的结点 **叶子结点**：度为0 **路径**：由结点与其后继结点连接的序列
  6. **路径长度**：路径所通过的结点数减1 **孩子结点、兄弟结点、双亲结点、子孙结点、祖先结点**
  7. **结点层次（深度）**：根节点为1，其孩子结点为2……树的最大层次称为**树的高度（深度）**
  8. **有序树/无序树**：树中各节点的子树的相对次序不能/能随意更换的树。一般情况下树为有序树
  9. **森林：**n个不相交的树的集合
  10. ~~树的性质：~~
      1. ~~树中的结点数等于所有结点的度数之和加1~~
      2. ~~度为m的树中第i层上最多有个结点（i>=1）~~
      3. ~~高度为h的m次树最多有个结点~~
      4. ~~具有n个结点的m次树的最小高度为 顶角括号表示大于等于。。的最小整数~~
  11. ~~遍历：~~
      1. ~~先根遍历：访问根结点，对子树递归~~
      2. ~~后根遍历：对子树递归，访问根结点~~
      3. ~~层次遍历：从上到下、从左到右的次序访问结点~~
  12. 存储结构：
      1. 双亲表示法：顺序存储结构，结点类中包含数据、双亲的位置。
      2. 孩子表示法：
         1. 多重链表法：在树的链表表示法中做修改，使结点包含数据和若干个孩子结点
         2. 孩子链表表示法：主体为存放树的数据的一维数组，每个数组由数据域和指针域组成，指针域指向孩- 子组成的单链表的首位置，单链表的结点单元由该结点在一维数组的序号（下标）和指向下一个- 孩子的指针构成
         3. 双亲孩子表示法：在孩子链表表示法的主体部分增加一个双亲域
      3. 孩子兄弟表示法：链式存储结构，结点类中包含数据、指向下一个兄弟的指针、指向长子的指针

## 二叉树

* + 1. 二叉树是有限的结点集合，集合或为空，或由一个根节点和两个互不相交的左子树和右子树的二叉树组成
    2. 二叉树严格区分左右子树
    3. **满二叉树**：所有分支节点都有左右孩子结点、叶子结点都集中在最下一层(结点数为)
    4. **层序编号**：从树根为1开始，从上到下从左到右依次编号
    5. **完全二叉树**：只有最下面两层的结点的度数可以小于2，且最下面一层的叶子结点依次排列在最左边
       1. 层序编号时，编号为i的结点若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
       2. 结点总数为奇数时，单分支结点个数，否则
    6. **正则二叉树**：二叉树所有结点的度等于0或等于2
    7. 性质：

       2. 非空二叉树的第i层上最多有个结点（i>=1）
       3. 高度为h的二叉树最多有个结点（h>=1）
       4. 完全二叉树中层序编号为i的结点有以下性质：
          1. ~~若，即，则此结点为分支结点，否则为叶子结点~~
          2. ~~若n为奇数，每个分支节点都有左右孩子；若为偶数，编号最大的分支结点(=)只有左孩子~~
          3. 如果2i>n，则结点i无左孩子，否则其左孩子结点为2i
          4. 如果2i+1>n，则结点i无右孩子，否则其右孩子结点为2i+1
          5. 除根节点外，此结点的双亲结点的编号为
       5. 具有n个结点的完全二叉树的高度为~~或~~
    8. 二叉树与树、森林之间的转换：
       1. 森林、树转换为二叉树：
          1. 树：树中所有相邻兄弟之间加一条连线，树中所有结点只保留与长子之间的连线，删除与其他孩----子之间的连线，原兄弟结点为右孩子，原孩子结点为左孩子（图像顺时针转动45°）。
          2. 森林：先把所有的树转换为二叉树，依次把后面的二叉树的根节点作为前树根节点的右孩子
       2. 二叉树还原为树、森林：
          1. 树：若某结点为左孩子，把它的右孩子、右孩子的右孩子。。。都与该节点的双亲连起来，删除所----有双亲与右孩子之间的连线，逆时针转动45°
          2. 森林：抹除根节点右链上所有双亲与右孩子之间的连线，将各树还原
    9. 存储结构：
       1. 顺序存储结构：用~~#~~0填补空结点，使其成为满二叉树，再从上到下，从左到右依次存入数组
       2. 链式存储结构：
          1. 二叉链表：由包含数据、指向左孩子的指针、指向右孩子的指针的类组成的链
          2. 三叉链表：比二叉链的类多一个指向双亲的指针
          3. 链式存储结构更节约存储空间
    10. 基本运算：
        1. CreateBTree(b, str) 用括号表示法字符串创建二叉树 DestroyBTree(&b) 销毁二叉树
        2. FindNode(b, x) 查找结点 L / RchildNode 找左/右孩子
        3. BTHeight(b) 求二叉树的高度 DispBTree(b) 以括号表示法输出二叉树
    11. 遍历：
        1. 先序遍历：访问根节点、先序遍历左子树、先序遍历右子树
        2. 中序遍历：中序遍历左子树、访问根节点、中序遍历右子树
        3. 后序遍历：后序遍历左子树、后序遍历右子树、访问根节点
        4. ~~层次遍历：根节点进队，队不空时循环，出队并访问一个结点，若有孩子，依次进队左右孩子~~
        5. 由先序序列和中序序列，或由中序序列和后序序列可以唯一确定一颗二叉树

## 线索二叉树

* + 1. 在空链域存放指向结点的前驱结点和后继结点的地址
    2. **线索**：指向该线性序列的前驱结点和后继结点的指针
    3. 存储结构：ltag（左孩子存在时为0）、lchild（指向左孩子的指针）、data、rchild、rtag
    4. 头结点：ltag为0，rtag为1，data为空，lchild指向无线索时的根节点，rchild指向遍历的最后结点
    5. 可以提高遍历效率

## 哈夫曼树

* + 1. 是只有叶子结点有值的二叉树
    2. **带权路径长度（WPL）**：从根节点到该结点之间的路径长度与该结点上权的乘积
    3. 树的带权路径长度：所有叶子结点的带权路径长度之和
    4. **哈夫曼树（最优二叉树）**：在个带权叶子结点构成的所有二叉树中，WPL最小的二叉树
    5. 构造算法（哈夫曼算法）：结点由权重、左孩子、右孩子、双亲构成
       1. 根据给定的个权值构成森林
       2. 森林中取两棵结点的权值最小的子树作为左右子树构造一个权值为子树之和的新二叉树
       3. 用新二叉树替代被取走的两棵树（值为两个结点的和）
       4. 重复前两步至只剩一棵树为止
    6. 定理：
       1. 对于具有个叶子结点的哈夫曼树，共有个结点
    7. 哈夫曼编码（前缀编码）：
       1. 字符出现的次数为权重，字符集合为结点数据，构建哈夫曼树，给出现频率最高的字符最短的编码
       2. 哈夫曼树所有左分支代表0，右分支代表1，从根节点到叶子结点的路径组成的01序列为编码

# 图

* 1. V为顶点的有限集合，E连接V中两个不同顶点的边的有限集合，记为V(G) 、E(G)
  2. **有向图**：表示边的顶点对（或序偶）是有序的。有向边用尖括号表示
     1. **起始端点（起点）**：边从此点出，称该边为此点的一条**出边**
     2. **终止端点（终点）**：边从此点进入，称该边为此点的一条**入边**
     3. **出/入边邻接点**：在一条有向边（**弧**）上有起点i终点j，则i是j的入边邻接点，j是i的出边邻接点
  3. **无向图**：表示边的顶点对（或序偶）是无序的。无向边用圆括号表示
     1. **端点、邻接点**：无向图中，同一条边上的两个端点互为**邻接点**
  4. **顶点的度**：无向图中为一个顶点所关联的边数；有向图中，顶点的度=入度+出度，入/出度=顶点的入/出边数
  5. **完全图**：无向图中，每两个顶点之间都有一条边；有向图中，每两个顶点之间都有两条方向相反的边
     1. 无向完全图边数=n(n-1)/2 有向完全图边数=n(n-1)
  6. **稠密图**：接近完全图的图 **稀疏图**：含有较少的边数的图（如，e为边数，n为顶点数）
  7. **子图**：V’（不能为空）和E’都为另一个图V、E的子集的图
  8. **路径**：是一个顶点序列(i, i1, ... , j)，顶点之间所经边必须属于E(G) **路径长度**：路径上所经边的数目
  9. **简单路径**：除开始点和结束点可以相同外，其余顶点均不相同的一条路径
  10. **回路/环**：开始点和结束点相同的路径 **简单回路/环**：开始点和结束点相同的简单路径
  11. **连通**：两个顶点之间有路径，称这两个顶点连通 **连通图**：无向图中任意两个顶点都连通，否则为**非连通图** **连通分量**：无向图的极大连通子图，连通图只有自身这一个连通分量
  12. **强连通图**：有向图中任意两个顶点都连通，否则为非强连通图
  13. **强连通分量**：有向图的极大连通子图，强连通图只有自身这一个强连通分量
  14. 称为极大是因为如果此时加入任何一个不在图的点集中的点都会导致它不再连通
  15. **权**：每个边上可以附有的一个对应的数值 **网（带权图）**：边上带有权的图
  16. 存储结构：
      1. 邻接矩阵存储方法：含有n个顶点（编号为0、1 …… n-1）的图G，其邻接矩阵A为n阶方阵
         1. 不带权图：当(i, j)或<i, j>是G中的边时，A[i][j]为1，否则为0
         2. 带权图：当(i, j)或<i, j>是G中的边时，A[i][j]为边的权w；i=j时，A[i][j]为∞；其他情况为∞
         3. 特性：
            1. 图的邻接矩阵是唯一的
            2. 邻接矩阵的存储空间为，与边数无关，所以邻接矩阵适合于稠密图
            3. 无向图的邻接矩阵为对称矩阵，可以采用压缩存储
            4. 对于无向图，第i行或第i列非零元素、非∞元素的个数正好是顶点i的度
            5. 对于有向图，第i行（第i列）非零元素、非∞元素的个数正好是顶点i的出度（入度）
            6. 此方法中判断两顶点是否有边或求权的执行时间为，需要提取边权的算法中通常使用此方法
      2. 邻接表存储方法：是一种顺序与链式存储相结合的存储方法
         1. 创建：
            1. 每个顶点建立一个单链表：

顶点结点（顶点信息data、指向单链表的指针firstarc）

边结点（与顶点邻接的顶点（有向图为出边邻接点）的编号adjvex、指向下一个边结点的指- 针nextarc、权值等与该边相关的信息weight）

* + - * 1. 再将所有头结点构成一个数组adjlist，完成邻接表
      1. 对于无向图，此类结点数为边数的两倍；对于有向图，等于边数
      2. 逆邻接表：有向图中边结点存储的是入边邻接点而非出边邻接点
      3. 特性：
         1. 邻接表不唯一，因为各结点链接次序可以是任意的
         2. 对于无向图，共有n+2e个结点；对于有向图，共有n+e个结点。特别适合稀疏图存储
         3. 对于无向图（有向图），表中顶点i对应的单链表的边结点数为i的度（出度）
         4. 此方法一般用于需要提取某个顶点的所有邻接点的算法中
    1. ~~其他存储方法：~~
       1. ~~十字链表：头结点（data，firstin，firstout）边结点（tailvex，headvex，hlink，tlink，weight）~~
          1. ~~tailvex（headvex）为起（终）点编号，hlink指向相同起点的下一个边结点~~
          2. ~~通过firstout->hlink->下一个hlink……的方式找所有出边，找入边方式同理~~
          3. ~~仅适用于有向图，方便找顶点的所有邻接点以及求度~~
       2. ~~邻接多重表：头结点（data，firstarc）边结点（mark，i，ilink，j，jlink，weight）~~
          1. ~~firstarc指向第一条依附于此点的边、mark为标识域（是否被搜索过）、ij表示边的两个顶点~~
          2. ~~ilink指向下一条依附于i的边结点，jlink指向下一条依附于j的边结点~~
          3. ~~仅适用于无向图~~
  1. 算法设计：创建图的邻接表、输出图、销毁图

## 图的遍历

* + 1. 深度优先遍历（DFS）：距离初始顶点越远越优先访问
       1. 从图中某个初始顶点v出发，首先访问v。选择一个与顶点v相邻且没被访问过的顶点w，再从w出发进行深度优先遍历，直到图中与当前顶点v邻接的所有顶点都被访问过为止。
       2. 邻接表总时间：、邻接矩阵总时间：、后进先出采用栈、适合用于寻找两点之间的路径
    2. 广度优先遍历（BFS）：距离初始顶点越近越优先访问
       1. 访问初始点v，接着访问v的所有未被访问过的邻接点v1，v2，…，vt，再按照v1，v2，…，vt的次序，对每一个顶点进行广度优先遍历，直到图中所有和点v有连通的顶点都被访问过为止。
       2. 邻接表总时间：、邻接矩阵总时间：、先进先出采用队列、适合用于寻找最短路径
    3. 对于非连通图需要在每个连通分量中选择初始点进行遍历
    4. 寻找两顶点之间的所有简单路径：带有回溯的DFS（一个顶点找完所有路径后置该顶点为未访问）
    5. BFS找到的路径一定是最短路径，而DFS则不一定。DFS能找所有路径，而BFS难以实现。
  1. 生成树：一个连通图的一个极小连通子图
     1. 极小是因为此时如果删除一条边，就无法构成生成树，也就是说给极小连通子图的每个边都是不可少的
     2. 通过DFS、BFS遍历可以得到**深度优先生成树**、**广度优先生成树**
     3. 各个连通分量的生成树组成非连通图的生成森林
     4. **最小生成树**：在带权图中，所有生成树中边上的权值之和最小的树，可用于规划铁路
     5. 普里姆Prim算法：
        1. ①对于带权图G=（V，E），创建一个树T=（U，TE），在U中放入一个V里的顶点作初始点，TE为空
        2. ②在点集U和V-U之间的边中选择权值最小的边，将边所连的不在U中的点加入U，将边加入TE
        3. ③重复第②步直至所有顶点都被加入U中，此时T为G的最小生成树
        4. closest[ j ]数组表示离j最近且属于U的点，lowcost[ j ]表示j到U的最小边的权值
        5. lowcost[ j ]为0时表示j属于U，不为零时它属于V-U
        6. 此算法频繁取用权值，G适合采用邻接矩阵。时间复杂度为，与边数e无关，适合稠密图
     6. 克鲁斯卡尔Kruskal算法：
        1. ①对于带权图G=（V，E），创建一个树T=（U，TE），U的初值为V，TE的初值为空
        2. ②从E中按从小到大的顺序选取边，若此边不使T生成回路，则加入TE，直至TE中的边达到n-1条
        3. 此算法频繁取用权值，G适合采用邻接矩阵。时间复杂度为，与顶点数无关，适合稀疏图
        4. 判断（i，j）是否构成回路：
           1. ①设置辅助数组vset[ i ]表示顶点i所在的连通分量编号，初值为自身顶点编号
           2. ②加入边时将两顶点所在连通分量中所有顶点的连通分量编号改为相同
           3. 当vset[ i ]==vset[ j ]时，构成回路
        5. 改进：将边集排序改为堆排序、利用并查集进行连通分量合并。优化后时间复杂度为
     7. 最短路径：**最短路径长度**、**带权最短路径长度**
        1. 单源最短路径长度：源点v到带权有向图G中其他顶点的最短路径，**迪克斯特拉(Dijkstra)算法**求解：
           1. ①对于带权有向图G=（V，E），从V中取源点v建立S={v}，设U=V-S，表示未被取用的点集
           2. ②从U中选择点u，使u到v的最短路径最小，将u加入S
           3. ③以u为新的考虑点，修改v到U的最短路径，重复②③直至S包含所有顶点

（加入u后寻找u的出边邻接点a，若a不在U中则令dist[a]为MIN{dist[a], dist[u]+ua}）

（若发生修改，则将path[a] = u）

* + - * 1. 数组path[ i ]表示从源点到i点的最短路径上顶点i的前一个顶点的编号
        2. 数组dist[ i ]表示从源点到i的最短路径长度
        3. 时间复杂度为
      1. 每对顶点之间的最短路径：带权有向图G中任意不同顶点i j之间的最短路径，**弗洛伊德(Floyd)算法**：
         1. 设G=（V，E）采用邻接矩阵g表示，二维数组AK[ i ][ j ]表示AK-1[ i ][ j ]考虑k后修正的顶点间的最短路径长度，A-1[ i ][ j ]=g，二维数组pathk[ i ][ j ] 表示i到j最短路径上j的前一个顶点的编号
         2. ①根据g求得A-1[ i ][ j ]和path-1[ i ][ j ]
         3. ②考虑所有经过顶点0的路径，计算A0[ i ][ j ]和path0[ i ][ j ]，若A-1[i][0]+ A-1[0][ j ]< A-1[ i ][ j ] 则更新A0[ i ][ j ]，重复此步直至求出An和pathn
         4. 时间复杂度为

## 拓扑排序

* + 1. 在一个有向图中找一个拓扑序列的过程
    2. 拓扑序列：顶点序列v1,v2...vn为拓扑序列，若<vi,vj>是有向图G中的一条边或之间有路径，则vi必须排-----------在vj前面
    3. AOV网(activity on vertex newtork)：用顶点表示活动，用有向边表示活动之间的优先关系的有向图
    4. 方法：
       1. ①从有向图中选择一个入度为0的顶点并输出它
       2. ②从图中删去该顶点及从该顶点的出边，重复①②直至图中不存在入度为0的顶点
    5. 结果：若所有顶点都被输出，说明图中无回路，称为**有向无环图**；反之说明图中有回路

## AOE网

* + 1. 以顶点表示事件，有向边表示活动，边e的权c表示完成活动所需时间来描述工程的有向无环图
    2. **开始/结束事件**：入/出度为0的顶点 **起点/终点事件**：活动从起点事件开始，到终点事件结束
    3. **源/汇点**：若网中只有一个开始/结束事件，则称其为源/汇点
    4. 若有多个开始事件，则可以设一个虚拟源点，使其长度为0的出边指向所有开始事件，汇点同理
    5. **关键路径**：从源点到汇点路径长度最长的路径 **关键活动**：关键路径上的活动
    6. 关键路径的长度为完成工程的最短时间 **富余时间**
    7. **前驱/后继活动**：当两条边首尾相接时，称它们所代表的活动为前驱/后继活动
    8. **触发事件**：只有当事件w发生后，活动a才能开始，称w为触发事件
    9. 事件的最早开始时间ve(j)：等于从源点到事件j之间的最长路径长度，此路径的事件和活动必须刻不容缓
    10. 事件的最迟开始时间vl(j) (late event)：等于ee(汇点)-j到汇点的最长路径长度
    11. 事件ve(j)的推算：计算所有j的前驱结点的ve + 前驱结点到j的活动的长度，它们的最大值为ve(j)
    12. 事件vl(j)的推算：计算所有j的后继结点的vl +j到后继结点的活动的长度，它们的最小值为vl(j)
    13. 活动的e/l (early/late)：e等于起点事件的ve，l等于终点事件的vl - 活动所需的时间

# 查找

* 1. **动态/静态查找表**：在查找的同时涉及/不涉及对表的修改操作，所对应的查找表
  2. **内/外查找**：整个查找过程都在内存中进行/需要访问外存
  3. 关键字：查找时用来给数据元素排序的数据项
  4. 平均查找长度，为查找第i个元素的概率，一般设为，为找到第i个元素所需比较次数
  5. 顺序查找：用顺序表或线性链表来表示静态查找表时，只能用顺序查找。在第0个位置设置哨兵，从后往前找
     1. 过程：从表的一端开始，逐个比较记录的关键字与给定值。
     2. ASL：当n很大时可以忽略不成功。此时，若不忽略，假设成功不成功的可能性相同，
  6. 折半查找（二分查找）：要求查找表为有序排列顺序存储结构。
  7. 分块查找：建立一个索引表，为有序表，数据元素包含每个分块的最大关键字和分块的起始地址
     1. 对索引表可以使用顺序查找也可以使用折半查找。分块内的记录任意排列，在分块内只能顺序查找
  8. 折半查找效率最高，但只适合顺序存储结构，不方便增删。
  9. 并查集
     1. 用于求解等价关系的问题，将所有元素每个元素作为一个集合
     2. 通过给出的包含两个元素的无序对，将涉及的两个元素所在的集合合并，重复此步直至无序对穷尽
     3. 此时可以查找任意两个元素是否属于同一集合
  10. 动态查找（树表）：
      1. 二叉排序树BST：左子树所有结点均小于根节点，右子树所有结点均大于根节点，左右子树都是BST
         1. 该树的中序序列是递增有序序列、该树的形态与数据输入次序相关
         2. 删除结点：若被删除结点为：
            1. 叶结点：直接删除
            2. 只有左子树，没有右子树：直接将左子树的根节点替代被删除结点的位置
            3. 只有右子树，没有左子树：直接将右子树的根节点替代被删除结点的位置
            4. 既有左子树，又有右子树：用中序序列的前驱或后继结点替代被删除结点的位置
      2. 平衡二叉树AVL：具有BST特性，左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过1
         1. 平衡因子（平衡度）：左子树高度-右子树高度
         2. 调整方法：根节点为A，插入到所属A的子树的根节点为B
            1. LL型平衡旋转：顺时针旋转，B为根节点，A为B的右子树，Br为A的左子树
            2. LR型平衡旋转：B的右子树为C，使C为根节点，A为C右子树，B为C左子树，C的左右孩子
            3. - 分别给B和A
            4. RR型平衡旋转：逆时针旋转，B为根节点，A为B左子树，Bl为A的右子树
            5. RL型平衡旋转：B的左子树为C，是C为根节点，A为C左子树，B为C右子树，C的左右孩子
            6. - 分别给A和B
      3. 散列查找：
         1. 散列表构造方法：
            1. 直接地址法：取关键字为散列地址。
            2. 折叠法：关键字分割为位数相同的几个部分，去这几部分的叠加和（舍去进位）为散列地址
            3. 数字分析法：取关键字的分布均匀的位，组成散列地址
            4. 平方取中法：去关键字平方后的中间几位为散列地址
            5. 除留余数法：关键字除以p得到的余数作为散列地址
            6. 随机数法：选择一个随机函数，取关键字的一个随机函数值为它的散列地址
         2. 冲突处理方法：
            1. 开放定址法：冲突时按某种方法继续探测表中其他单元，直到找到一个空位置

探测地址=(直接散列地址+偏移)mod 散列表长

线性探测再散列：偏移=1，2，3。。。

二次探测再散列：偏移=1，-1，4，-4，9，-9。。。

随机探测再散列：偏移=随机数序列

* + - * 1. 再散列法：当冲突时，采用另一个散列函数计算地址，直至不冲突
        2. 链地址法：散列表为由m个头指针组成的一维指针数组，指向单链表，所有关键字为同义词的结- 点在同一单链表中。采用头插法。
        3. ASL=

# 排序

* 1. 
  2. 稳定性分析：排序码相同的记录，若经过排序后，这些记录仍保持原来的相对次序不变，称这个排序
     + - 1. 算法是稳定的。否则，称为排序算法是不稳定的
     1. 邻接表的生成不唯一
     2. 深度优先遍历仅在确定邻接表的结构时，遍历结果确定。
     3. 拓扑排序结果不唯一
  3. 内部排序：排序过程中所有数据均放在内存中，不需要外存
     1. 内部排序的记录分为有序序列区和无序序列区。逐步扩大有序序列区，直至所有记录都有序。
     2. 内部排序算法：
        1. 插入排序：将待排序表中的记录逐个插入有序表【有序表初始为原表第一位】
           1. 直接插入排序：无序表按顺序取，顺序查找有序表，找到应在位置，插入指定处
           2. 折半插入排序：无序表按顺序取，折半查找有序表，找到应在位置，插入指定处
           3. 希尔排序：对n个记录进行排序时，首先取一个整数d<n【一般取n的一半】，将这n个记录分
           4. 成d组，所有位置相差为d的倍数的记录分在同一组，在每组中使用直接插入排序进行组内排序，然后缩小d的值，重复进行分组和组内排序，一直到d=1结束。因此又称“缩小增量排序”
        2. 交换排序：
           1. 冒泡排序：对所有记录从左到右每相邻两个记录的关键字进行比较，如果这两个记录的关键字不

符合排序要求，则进行交换，这样一趟做完，关键字最大者将处于最后一个位置；反复进行n-1次排序后，序列有序。（如果在某一趟中没有发生交换，说明已经有序）

* + - * 1. 快速排序：通过一趟排序将待排序的记录分割成独立的两部分，其中一部分记录的关键字均小于
        2. 另一部分记录的关键字，然后分别对这两部分记录继续进行排序，以达到整个序列有序
      1. 选择排序：
         1. 简单选择排序：遍历找到最小/最大值，与第一个记录交换；再在剩下的记录中选出。。。
         2. 树型选择排序（锦标赛排序）：按淘汰赛思想，两两比较，较小/较大者胜出，继续比较直至最后

堆排序：堆采用一维数组存储，但自上而下，从左到右，视为完全二叉树，任一双亲结点的

- 值都小于（大于）其孩子结点的值，则根节点为最小（最大值），该堆为小跟堆（大- 根堆）

过程：生成递减序列，需要小根堆。将堆顶元素与堆尾元素互换，原堆顶元素移除堆，- 再调整堆，使其恢复堆属性。重复

* + - 1. 归并排序：将两个或以上的有序表组合成新的有序表
         1. 2路归并排序：将初始序列中的所有记录看作长度为1的序列，将前后两个相邻的有序表归成一
         2. 个有序表，直至只剩1组
      2. 基数排序：无需比较关键字，通过“分组”与“收集”两个过程来完成排序任务
         1. 1）将待排序的关键字编成统一长度的排序码：024, 002, 056, 102, 098, 004, 081, 112；并根据基数设置10个队列（分别为0~9）；
         2. 2）根据各关键字的个位数，将各关键字依次进入相应的队列，完成第一次“分组”；然后将0~9个队列中的关键字依次出队，完成第一次“收集”；
         3. 3）重复第（2）步的过程，依次完成对“十位数”、“百位数”的“分组”与“收集”，完成整个排序。
      3. 计数排序：将输入的数据值转化为键存储在额外开辟的数组空间中，牺牲空间换时间
         1. 1）函数需要接受原始数组arr，值的最小值min，值的最大值max【或通过遍历获取最值】
         2. 2）创建长度为max-min+1的计数数组count【数组元素初始值为0】，遍历原始数组arr，使count[arr[i]-min]++
         3. 3）创建长度为max-min+1的加总数组sum【初始0】，使得sum[0]=count[0]，sum[i-min]=sum[i-min-1]+count[i-min]，每一项表示小于等于该项所代表数字的元素个数
         4. 4）创建长度为max-min+1的存储数组res，从后往前遍历arr，
         5. 使res[(sum[arr[i]-min]--)-1]=arr[i]
      4. 桶排序：将数据分到有限数量的桶里，每个桶再分别排序
         1. 1）设置一个指定长度的数组作为空桶。数组的元素为队列
         2. 2）遍历原序列，将各个元素放入指定桶
         3. 3）对非空的桶排序
         4. 4）从非空桶中依次出队并拼接数据

# 其他

* 1. LRU缓存：一种数据结构，容量为正整数。有get和put方法。
     1. get（key） 如果key存在于缓存中，返回关键子的值，否则返回-1
     2. put（key，value） 若关键字已存在，变更其数据值。若关键字不存在，添加该组「关键字-值」。
        1. 若缓存容量已达上限，则在写入前删除最久未使用的数据，然后写入