数据结构

- 逻辑结构、物理结构?
  - 逻辑结构：数据之间的逻辑关系，树、图、集合、线性
  - 物理结构：数据结构在计算机中的表示，顺序、链式、索引、散列存储
    - 顺序存储：逻辑上相邻的元素物理也相邻
    - 链式存储：不要求逻辑上相邻的元素物理也相邻，借助指针表明元素之间的逻辑关系
    - 索引存储：存储元素信息，还需要存储一张索引表，检索速度快，但占用额外存储空间
    - 散列存储：可以根据元素关键字直接计算出元素存储地址
- 算法五大特性?
  - 有穷性：有限的步骤
  - 确定性：每条指令有确切含义
  - 可行性：每一步都可通过执行有限次数实现
  - 输入：>=0个输入
  - 输出：>=1个输出
- 什么是好的算法?
  - 正确性、可读性、健壮性（给出非法数据，恰当处理)
  - 高效率、低存储　政客

- **顺序、链式存储结构的区别?**
  -

读取方便 O(1)	读取不方便 需要遍历 O(n)
插入删除 需要移动大量元素	插入删除方便 只需要改变指针
空间分配：一次性	在需要时分配
存储密度 = 1	存储密度 < 1

- **数组、链表区别?**
  -

事先定义长度，不能适应数据动态地增减	动态地进行存储分配，可以适应数据动态地增减
从栈中分配空间	从堆中分配空间
快速访问数据元素，插入删除不方便	查找访问数据不方便，插入删除数据发布

- **头指针、头结点区别?**
  - **头指针：指向链表第一个节点的指针**
  - **头结点：为方便统一操作，放在第一个元素节点之前**
- **介绍KMP算法**
  - **改进的模式匹配算法，模式串失配时，不是从下一个位置匹配，而是跳过一些不可能的位置，以达到快速匹配的目的**
- **栈和队列的异同、用途**
  - **相同：线性结构；插入在表尾；顺序或链式存储；插删时间复杂度相同**
  - **不同：栈先进后出，队列先进先出**
  - **栈可用于函数调用和递归，括号匹配，后缀表达式求值；**
  - **队列，层次遍历，解决主机外设速度不匹配的问题（缓冲队列），解决多用户引起的资源竞争问**

题（等待队列））

- **队列假溢出?**
  - **设队头指针为front，队尾指针为rear，队列的容量为maxsize,有元素入队，若rear=maxsize，上溢——循环队列解决**
  - 队头指针指向队头元素，队尾指针指向队尾元素的下一个位置
- **栈在后缀表达式求值?**
  - **是操作数则进栈**
  - **是运算符则将两个元素出栈并将得到的结果进栈**
  - **表达式扫描完成后，栈顶元素为所求结果**
- **两栈实现队列?**
  - **两个栈**
  - **入队：全部元素入栈1，出栈1压栈2**
  - **出队：栈2依次出栈**
- **树、满二叉树、完全二叉树、BST、AVL?**
  - **树：非线性的数据结构，其元素之间有明显的层次关系，由结点和边组成且不存在环**
  - **满二叉树：树高h，且包含2^h-1个节点的二叉树**
  - **完全二叉树：按层序编号，每个节点都与满二叉树中编号相同**
  - **BST：二叉排序树，左子树结点值<根结点值<右子树结点值，中序遍历得到递增序列**

- AVL：二叉平衡树。左右子树都是空树，或 左右子树都是AVL且高度差绝对值不超过1
- 树、二叉树存储结构?
  - 二叉树的存储结构：
    - 顺序存储结构：用连续的存储单元从上到下、从左到右存一棵完全二叉树；适用于完全二叉树，存储一般的树会浪费大量存储空间
    - 链式存储结构：采用二叉链表，左指针指向左孩子，右指针指向右孩子
  - 树的存储结构：
    - 双亲表示法：一维数组存储，每个节点增设一个伪指针，指示双亲在数组中的位置
    - 孩子表示法：每个结点的孩子结点用单链表连接起来
    - 孩子兄弟表示法：二叉链表存储，左指针指向孩子，右指针指向兄弟
- 二叉树、度为2的树?
  - 二叉树每个节点最多两棵子树，可以为空；度为2的树最少也要有三棵树
  - 二叉树左右子树有顺序
- 哈夫曼树？如何构造?
  - 哈夫曼树带权路径最短（带权路径：树中结点的值乘于结点到根的距离）

- 从集合中选取根结点权值最小的两树组成一颗新树，新树的权值为左右子树权值之和，删除集合选取的两个结点，增加新树的结点，重复上述操作；

- 线索二叉树?
  - 二叉链表存储的二叉树，并将空链域利用起来，指向前驱或者后继
- 红黑树?
  - 一种平衡二叉查找树，结点被标记为红色或者黑色
  - 1、根节点是黑色，每个叶结点是不存数据的空节点
  - 2、相邻的两个节点不能同时为红色（父节点、子节点不同时为红色）
  - 3、树中任一节点到可达的叶节点，中间有相同数量的黑色节点
  - 应用：C++ STL中的set/map
- 图的深度优先遍历、广度优先遍历?
  - 深度优先搜索（先序遍历）：从某个顶点出发，访问这个顶点并设为已访问、访问该顶点的未被访问的邻接结点、重复、直到某一结点的所有邻接点已被访问，退回上一步继续访问其他未被访问的邻接点、直到遍历完成；
  - 广度优先搜索（层序遍历）：从某个顶点出发，访问其所有邻接结点并设为已访问，然后按照顺

序访问邻接结点的邻接结点、直到遍历完成

- 图定义、存储?
  - 由顶点集和边集组成，顶点+联系
  - 存储结构：邻接矩阵法、邻接表、邻接多重表（无向图）、十字链表（有向图）
- 图：求最小生成树?
  - 最小生成树（带权连通图的最小代价生成树，基于贪心）
  - 1、Prim算法：初始任选一个顶点，以后每次选择一个与当前顶点集合距离最近的顶点；适合边稠密图
  - 2、Kruskal算法：按权值递增次序选择合适的边；适合边稀疏图
  - 应用：旅游时规划路径让交通费用最低；建设电力网、公路网、通信网时怎么规划路径可以让建设成本最低
- 图：最短路径算法?
  - 迪杰斯特拉算法（某点到其它节点最短路径）：
    - 从源点出发，每次选择离源点最近的一个顶点前进，然后以该顶点为中心进行扩展，最终得到源点到其余所有点的最短路径。
    - 应用：适合稠密图、不能处理负权图、处理单源最短路径
  - Floyd算法（任意两点间最短路径）

- 1、所有两点之间的距离是边的权，如果两点之间没有边相连，则权为无穷大。
- 2、对于每一对顶点u、v，看看是否存在一个顶点w，使得 u-w-v 比已知的路径更短，更短则更新
- 应用：稠密、稀疏皆可；可处理负权图；处理多源最短路径

- 判断图中是否有环？
  - 深度搜索遍历：若图中有一个顶点被访问两次则证明有环
  - 拓扑排序：查找图中入度为0的顶点，删除它，重复此操作；若图中最后还剩顶点则证明有环
- AOV、AOE？
  - AOV：顶点表示活动的 有向无环图，边无权值
  - AOE：边表活动的 有向无环图，边有权值
- 邻接矩阵、邻接表？
  - 邻接矩阵：矩阵的第i行第j列表示i到j是否连接。可快速添加、删除边，但存稀疏图会造成空间浪费
  - 邻接表：链表后面跟着所有指向的点。节省空间，但涉及度可能要遍历整个链表
- 各类排序算法？
  - 插入排序：每次将一个待排序的关键字插入到已排好序的子序列

- 直接插入
- 折半插入：折半查找找插入位置
- 希尔排序：缩小增量排序

- 交换排序：
  - 冒泡排序：每一趟从前往后、两两比较、逆序交换
  - 快速排序：基于划分的交换，需要选枢轴
- 选择排序：每一趟从待排序元素中选取最小的，加入有序子序列
  - 简单选择排序
  - 堆排序
- 归并排序：二路归并排序(相邻有序表归并成一个新的有序表)
- 基数排序：基于关键字各位的大小排序
- 外部排序——多路归并排序

- 排序算法稳定性、复杂度?

排序方式	时间复杂度			空间复杂度	稳定性
	平均情况	最坏情况	最好情况		
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$			$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	稳定

- 各类查找方法?

- 静态查找：顺序查找、折半查找、分块查找（基于索引表）
- 动态查找：BST，AVL
- 散列查找

- 散列表?
  - 建立了关键字与存储地址直接映射关系的数据结构
  - 冲突：>=2关键字映射到同一地址
  - 冲突解决：开放定址法（空闲地址向其同义词表项或非同义词表项开放）、拉链法（所有同义词存到一个线性链表中）
  - 查找效率取决于：散列函数、处理冲突方法、装填因子
- 贪心算法、动态规划、分治?
  - 贪心：选择每一阶段的局部最优，以达到全局最优
  - 动态规划：拆解子问题，记住过往，减少重复计算
  - 分治：拆解成n个相似子问题，递归解决，合并结果
- 递归、迭代?
  - 递归：自己调用自己
  - 迭代：输出再次作为输入进行处理