# 数据结构基础 - 课程内容大纲（完整版）

---

1. 第一节课介绍分数构成、作业形式等重要内容！
2. 复杂度分析
    - 大 O：上限（**logN is O(N)**，大 O 只规定上限，这句话是对的）
    - 大 Ω：下限
    - 大 θ：上限 + 下限
    - 小 o：无穷小量（大 O 的反向操作）
    - **if/else**：选较大的那个分支算大 **O** 复杂度
3. 栈和队列
    - 中缀表达式转后缀表达式
        i. 如果是操作数, 则直接压进输出队列中
        ii. 如果是运算符, 分以下情况:
            a. 运算符堆栈为空, 或者运算符栈顶元素为'(', 则直接将运算符压进栈
            b. 运算符如果是')', 则将运算符栈中的元素都压进去输出队列中并将其弹出运算符栈, 直到遇到'('为止
            c. 运算操作符是'+' '-' '*' '/' 之一的时候, 将其与运算符栈顶元素作比较,如果栈顶的优先级较小（如果运算符是左结合的则优先级相等也要出栈）, 则将运算符压入栈中。否则, 将栈顶元素弹出并压入输出队列中, 然后继续比较栈顶, 直到运算符被压入栈.
    - 中缀表达式转前缀表达式
        i. 存两个堆栈, 一个存放操作数, 一个存放运算符
        ii. 由于要转换为prefix, 运算符在操作数前, 所以扫描从中缀表达式从右往左 扫描.
        iii. 如果是操作数, 则直接压进操作数栈中
        iv. 如果是运算符, 分以下情况:
            a. 运算符堆栈为空, 或者运算符栈顶元素为')', 则直接将运算符压进栈
            b. 运算符如果是'(', 则将运算符栈中的元素都压进去操作数栈中并将其弹出运算符栈, 直到遇到')'为止
            c. 运算操作符是'+' '-' '*' '/' 之一的时候, 将其与运算符栈顶元素作比较,如果栈顶的优先级较小,（如果运算符是右结合的则优先级相等也要出栈） 则将运算符压入栈中. 否则, 将栈顶元素弹出并压入操作数栈中, 然后继续比较栈顶, 直到运算符被压入栈。
        v. 扫描完一遍后, 将运算符堆栈剩余的元素都压入操作数栈中。
        vi. 将操作数栈从栈顶到栈底输出就是prefix.
        - 前缀表达式的计算：从前往后读，把遇到的运算符放到运算符栈，遇到的数字放到数字栈，一旦数字有两个就拿出来用运算符栈顶的运算符计算，把结果放回数字栈。
    - 表达式树(expression tree)：后缀表达式是表达式树的后序遍历，前缀表达式是表达式树的前序遍历。
4. 树
    - traversal:（名字容易记错）

    - 前序遍历(**preorder**): 根左右
    - 中序遍历(**inorder**): 左根右
    - 后序遍历(**postorder**): 左右根
- **degree of tree: max degree of node; degree of node on tree:** 节点拥有的儿子个数（父亲不算），概念很容易忘
- 任意树变成二叉树：倾斜 45 度，兄弟变儿子；i.e.左儿子是第一个儿子，右儿子是兄弟
- threaded binary tree 线索二叉树
    - 目的：n 节点二叉树有 n+1 个儿子指针是 NULL，利用这些指针来使遍历更加方便
    - 分类：前序 / 中序 / 后序 threaded binary tree
    - 构造：NULL 的左儿子换成（前序 / 中序 / 后序）遍历中的前驱节点，右儿子换成后继。
    - 遍历：不再需要回溯，只需要判断自身和左右儿子的顺序即可
- 完全二叉树、满二叉树

5. 二叉搜索树: 参考 这个博客 或者随便搜一个博客看
    - 查找、插入
    - 删除根节点
    - 支持删除指定节点(带 lazy tag 后的查找和删除)
6. 堆
    - property：
        - 每个节点都比儿子大（左右儿子之间没有限制）
        - 一定是 complete binary tree
        - 查找只能 $O(N)$
        - 编号从 $\lfloor \frac{N}{2} \rfloor + 1$ 开始就都没有儿子
    - operation
        - 线性建堆(linear xxx)：保证操作每个节点时，他的两个儿子子树都是堆，然后将这个节点往下推。有 $\frac{N}{2}$ 个节点需要往下推至少一次，$\frac{N}{4}$ 个节点往下至少两次，以此类推 $T(N) = \frac{N}{2} + \frac{N}{4} + ... = O(N)$
        - push：在最大编号后面插入，依次往上交换
        - pop：把编号最大的放到根的位置（否则无法保证完全二叉树的性质），左右儿子挑一个大（小）的提上来
    - d-heaps:
        - 单次操作 $O(d \log_d N)$，d 为 3 时时间复杂度最低
        - 父亲：$father(i)$ 是一个阶梯函数，$father(1) = 0$，每过 d 个数函数值 +1，$father(i) = \lfloor (i + d - 2)/d \rfloor$
        - 最大的儿子：把最大的儿子后面的节点全去掉，则树上除了叶子之外全都是满儿子，$father(son_{max}(i)) = [son_{max}(i) - 1]/d = i$ 所以 $son_{max}(i) = id + 1$
        - 最小的儿子： $son_{min}(i) = (i - 1)d + 2$
7. 并查集
    - union-by-size 及其复杂度证明：小树合并做大树的儿子，查询 $O(\log_2 N)$。因为从任意节点每往上爬一层，子树大小至少翻一倍
    - union-by-depth 及其复杂度证明：浅树合并做深树的儿子，查询 $O(\log_2 N)$，因为一棵深度为 n 的树需要 2 棵深度为 n-1 的树合并得到，所以深度为 n 的树大小至少为 $2^n$，树深度为 $O(\log_2 N)$ 级

- 路径压缩：查询和合并的复杂度都是 $O(1)$，下面是一种非递归写法

```
SetType  Find ( ElementType  X, DisjointSet  S )
{
    ElementType  root,  trail,  lead;
    for ( root = X; S[ root ] > 0; root = S[ root ]
);
    for ( trail = X; trail != root; trail = lead ) {
    lead = S[ trail ] ;
    S[ trail ] = root ;
    }
    return  root ;
}
```

8. 图
   - tips1: 单讲 connected 一般是无向图，有向图要分强联通和弱联通
   - tips2: 有向图的 adjacent 有 from 和 to 之分
   - 一种图的存储方式：adjacent multilist，就是同一条边存两个next，分别是对于出点的 next 和对于入点的 next，方便找入度
9. 最短路算法
   a. Floyd $O(N^3)$
   b. Dijkstra $O(V^2+E)$
      - 堆优化：$O((V+E)\log V)$，log 后面是 E 还是 V 并不关键因为 E 最多是 V 的平方。
      - 可以处理负权边吗？不能
   c. Bellman-Ford & SPFA
      - SPFA 平均情况下 $O(kE)$，其中 $k$ 是所有顶点进队的平均次数，一般满足 $k<2$。最坏情况退化为 Bellman-Ford 算法，复杂度 $O(VE)$
      - 用数组 dis 记录每个结点目前的最短路径值，用邻接表来存储图 G。设立一个队列用来保存待优化的结点，优化时每次取出队首结点 u，并且用 u 点当前的最短路径值对 u 点能到达的所有结点 v 进行松弛操作，如果 v 点的最短路径估计值有所调整，且 v 点不在当前的队列中，就将 v 点放入队尾。这样不断从队列中取出结点来进行松弛操作，直至队列空为止。
   d. 拓扑排序
10. 其他图论算法
    a. 最小生成树
       0. 性质：边权最小的边一定在最小生成树中（用于证明两个算法的正确性）
       i. **Kruskal**（Kruskal 基于边，Prim 基于点，区分一下）：
          - 做法：按边权从小到大排序 + 取边做并查集
          - 证明：用性质
       ii. **Prim**：
          - 做法：以某个点为初始点集，每次选点集和外界连边中最小的边，把那个点加入点集
          - 证明：用性质
    b. 最大流

i. 定义：有源有汇，其他点流入等于流出。 定义参考博客

ii. 定理：最大流 = 最小割

i. 平面图网络流：

- 把源汇连边，找 对偶图 （面作为点，面之间相邻就连 2 条 有向边，正向保留边权，反向设为 0，走一条有向边表示把边左边的点放进 s 的集合中，右边的点放进 t 的集合中）
- 求最短路（本质上来说对偶图中的环相当于一个割。为了保证源汇在不同的集合中，强制选取了源汇之间连的虚拟边）

iii. 常见增广路算法：

i. 核心思路： 引入反向边

- 反向边小技巧：正向边存在数组的偶数位，反向边存在奇数位，则取反向边只需要 `i^1`

ii. Ford-Fulkerson：找到增广路径就更新

iii. Edmonds-Karp (EK 算法)：BFS 找边数最少的增广路径更新，$O(NM^2)$

iv. Dinic：在 EK 的基础上，分层 (BFS) + 多路增广 (DFS)，复杂度 $O(N^2M)$

- 当前弧优化：已经增广过的边不再增广，引用写法 `for (int &i = cur[u]; i; i = g.nxt[i])`

11. DFS 的应用：

a. 欧拉路径（回路）和哈密尔顿路径

- 欧拉回路 dfs： $O(V+E)$

b. 无向图的双连通分量(biconnectivity)

i. 定义：

- **articulation point**: 关节点，去掉这个点图变得不连通（注意记名词，并且关节点出现表示这张图一定是无向图）
- biconnected graph: 不存在关节点
- biconnected component: maximal biconnected subgraph

ii. tarjan 算法：

i. 生成一颗 dfs 树，遍历顺序记为 `dfn[v]`

ii. 除了树边之外仅可能存在一种边：连接 u 和 u 子树中的节点 v 的边。记 `low[v]` 为从 **v** 和 **v** 子树中的节点出发走 **1** 条 非树边能够到达的最小 dfn，更准确地说，

$$low[u] = \min \left\{ \begin{matrix} dfn[u] & \\ low[v] & ,v \text{ is a son of } u \\ dfn[w] & ,(u,w) \text{is a back edge} \end{matrix} \right\}$$

当 u 的某个儿子 v `low[v]>=dfn[u]` 时 u 是关节点。

iii. 想要记录每个双联通分量中的点有哪些：将遍历到的点都入栈，在找到关节点的时候不断出栈直到关节点出

栈，然后再把关节点入栈。（因为每个关节点可能会被包含在多个点双中）

iv. 注意：对于图的所有不连通的分量都要搜索，注意特判孤立节点的情况

c. 有向图的强联通分量：

i. 生成一颗 dfs 树，遍历顺序记为 `dfn[v]`

ii. 除了树边之外可能存在 3 种边：

i. 前向边：可以忽略
ii. 后向边：可以形成强连通
iii. 横插边：从 dfn 大的子树插到 dfn 小 的子树

iii. `low[u]` 表示 从 **u** 出发可以到达的最小的 **dfn**（和无向图双联通分量区分），其中 w 必须是还未确定在哪个强连通分量的点（即需要在栈中，用 `inq[w]` 来判断）

$$low[u] = \min \begin{cases} dfn[u] & \\ low[v] & , v \text{ is a s} \\ low[w] & , (u \rightarrow w) \text{ is a non-tree-edge and w} \in \text{undetermined com} \end{cases}$$

iv. 当节点 u 满足 `low[u]==dfn[u]` 时说明 u 和他的父亲属于不同的强连通分量，弹栈直到弹出 u

12. 排序：

a. 插入排序：

- 插入排序（以及任何交换相邻元素的排序）的交换次数 = 逆序对个数(inversion count)
- 最大比较和交换次数 $\frac{n(n-1)}{2}$，最小比较和交换次数 $n-1, 0$
- stable sort

b. 希尔排序 Shell Sort：

- 取步长(也称增量，**increment**)，进行分组插入排序，并逐步缩小步长，直到步长为 1，变为插入排序。
- 步长可以取 `floor(n/2)`，每次除 2。
- 平均复杂度取决于步长的选取，在随机条件下效果较好，$O(N^{\frac{3}{2}})$ 或者 $O(N^{\frac{5}{4}})$
- unstable sort

c. 堆排序：

- $O(N)$ 建堆，每次把堆顶的元素和最后一个元素交换，$O(logN)$ 更新堆。理论总复杂度不到 $O(NlogN)$，但实际效果并不好。

d. 快速排序：

- 方法：

i. 首先，在这个序列中随便找一个数作为基准数（只取其值不取其具体元素）
ii. 这里可以用两个变量 i 和 j ，分别指向序列最左边和最右边，称作"哨兵 i "和"哨兵 j "
iii. 左边的哨兵向右移动直到找到第一个大于基准数的数，右边的哨兵向左移动直到找到第一个小于基准数的数，交换这两个数
iv. 若 i > j 则说明 j 以及 j 左边都是小于 pivot 的数；i 以及 i 右边都是大于 pivot 的数；i 和 j 中间（不包含 i，j）

得数都等于 pivot。所以递归处理 j 的左边和 i 的右边部分（包含i，j）

- 一些问题：

  i. 如何选择 pivot 值？：

  - 问题：选择固定值最坏情况复杂度 $O(N^2)$
  - 解决方法1：随机 pivot，但是生成随机数复杂度较大
  - 解决方法2：使用头、中间、尾部三个数的中位数作为 pivot

  ii. 遇到等于 pivot 交换不交换？：遇到 pivot 直接 pass，等于 pivot 的不进入下一层递归，这样就不会被 `1, 1, ..., 1` 数据卡成 $O(N^2)$

  iii. 在 N 较小时（N<20）快速排序的效率不如插入排序：在 N 较小时用插排解决。

- 复杂度分析：$T(N) = T(i) + T(N-i-1) + cN$，最坏 $O(N^2)$，最好 $O(N\log N)$，平均 $T(N) = \frac{2}{N}\sum_{i=0}^{N-1} T(i) + (N-1) = O(N\log N)$，可用错位相减法求通项。参考博客

e. 归并排序：

  i. 从中间分成两段，分别递归

  ii. 在临时数组中进行归并，需要 $O(n)$ 的空间

f. table sort（名字容易忘）：用 `table[i]` 表示第 i 大的元素的下标，用于交换复杂度较高的场景

g. bucket sort（桶排序） & radix sort（基数排序）：

- **LSD** (Least Significant Digit)：即从最低位开始排序，这样一定是按照低位从小到大的顺序放到高位的桶里，保证排序。
- **MSD** (Most Significant Digit): 每个 run 之后对每个 bucket 单独排序，所以复杂度会更大一些。
- 复杂度：$O(N+B)$ & $O(P(N+B))$，其中 B 是基数，P 是重复次数，N 是桶或者基数个数

h. 其他

- **stable sort**(易考): 大小相同的元素不会交换位置
- unstable sort：会交换位置
- 基于交换的排序的复杂度下界：排序树的深度 $k \geq \log_2(N!)$，又有 $log_2(\frac{N}{2}\log_2\frac{N}{2} = \frac{N}{2}^{\frac{N}{2}}) \leq \log_2(N!) \leq log_2(N^N) = N\log_2 N$，所以 $\log_2(N!) = \Theta(N\log N)$

13. Hash

0. 专有名词：

- collision：表项已经存在
- overflow：表放满了
- loading density = 已经放了几个数到 hash table 里 / 总共可以放几个（常见题目：第一次发生冲突时的 **loading density**）
- identifier density = 已经放了几个数到 hash table 里 / 可能放到 table 里的数的总个数
- Hash Function & **key & hash value** (注意区分) : H(key) = hash value.

a. 时间复杂度：没有冲突 $O(1)$

b. 哈希函数：自变量是整数，自变量是字符串（选择部分字符看成 32 进制数）

c. 开放寻址法 open addressing：

   i. linear probing 循环找下一个位置直到找到空位

      - linear probing 的期望 probe 次数：对于插入或者不成功的查询 $\frac{1}{2}(1+\frac{1}{(1-\lambda)^2})$，对于成功的查询 $\frac{1}{2}(1+\frac{1}{1-\lambda})$

   ii. quadratic probing: 往后找 1, 2, 4, ... 个位置

      - 定理：如果使用平方探测，且表的规模是素数，那么当表至少有一半是空的时候，总能插入新的元素。
      - 拓展：如果表的规模是形如 $4k+3$ 的素数，则平方探测法可以探测到整张表。

   iii. double hashing: `f(i)=i*hash2(x)` 探测的步长与 key 值有关，解决聚集问题（**double hashing** 和 **rehashing** 的名字容易记混）

      - 二次哈希函数需要满足的条件：不会映射到 0，能够探测整个表。例如 `hash2(x)=R-(x%R)`，其中 R 是比 tablesize 小的质数（tablesize 也是质数）

   iv. rehashing：

      - rehashing 的条件：如果表已经有一半满或达到一定的插入率，或者某次插入失败了
      - 操作：把已经插入的 key 用新的哈希函数插入到一张新的表里，新表的大小是比当前大小的两倍大的最小质数，不是直接乘 2，否则非质数的表会导致效率降低。
      - 复杂度：因为之前的插入已经是 O(N) 的了，重新哈希一遍只会给每次插入的时间复杂度加一个常数。但交互的时候重新哈希速度慢会被用户感知。

d. seperate chaining: 对相同哈希值用链表存储，简单粗暴

e. 如何删除：先标记删除。频繁删除会降低插入的效率