



计算机领域本科教育教学改革试点
工作计划（“101计划”）研究成果

数据结构

授课教师：

湖南大学 信息科学与工程学院

第 9 章

不相交集

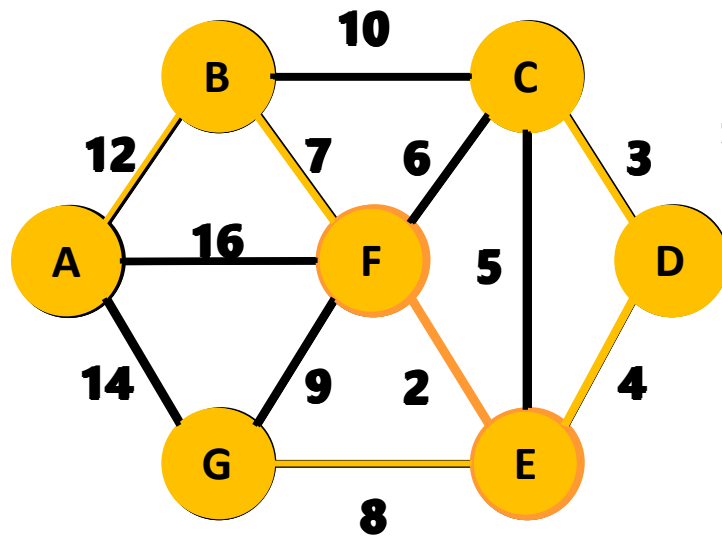
提纲

- 9.1 问题引入
- 9.2 等价关系、等价类和不相交集
- 9.3 不相交集的存储实现
- 9.4 不相交集的基本运算实现
- 9.5 不相交集的应用
- 9.6 拓展延伸
- 9.7 应用场景



9.1 问题引入：Kruskal算法的高效实现

- 求最小生成树的Kruskal算法：
- 按权重递增顺序考虑每条边
- 并把连接不同连通分量的边 $e = (u, v)$ 加入生成树。
- 这需要**动态地维护图的连通性**
- 需要支持下列三种操作：
 - （初始化）初始化一个无向图 $G = (V, E)$ ，其中 $E = \emptyset$ （空集），即每个顶点分属于不同的连通分量。
 - （加边）在图中加入一条边 (u, v) ： $E \leftarrow E \cup \{(u, v)\}$ 。
 - （连通性查询）查询顶点 u 和顶点 v 是否属于同一连通分量，即查询两顶点之间是否有路径相连。



第4步：
选取 $\langle A, B \rangle$



9.1 问题引入：Kruskal算法的高效实现

- 直接做法：利用无向图实现
- 加边操作只需 $O(1)$ 时间；
- 连通性查询可以通过图的遍历实现，在 $n = |V|$ 个顶点的图中，每次查询最坏需要 $O(n)$ 。
- 在Kruskal算法中，对每条边均需要进行一次连通性查询，因此 $m = |E|$ 次查询的总时间复杂度为 $O(nm)$ 。



9.1 问题引入：Kruskal算法的高效实现

- 注意：只需要查询顶点之间**是否连通**，不关心它们具体**通过哪条路径连通**
- 此外，只需要支持**加边操作**，而不需要支持**删边操作**。
- 这种情况下，与其完整地维护无向图的结构，不如直接维护连通分量构成的**集合**
- 每个连通分量用其中顶点的集合表示。

初始时，每个顶点都是独立的连通分量，此时有 n 个仅包含单一顶点的集合： $\{v_1\}, \{v_2\}, \dots, \{v_n\}$



1

每加入一条边 (u, v) ，就将 u 所属的集合和 v 所属的集合合并



2

对于连通性查询 (u, v) ，只需查询 u 和 v 是否在同一集合中



3

连通性查询问题变成了维护若干不相交的集合，并动态地合并、查找的问题。



9.1 问题引入：Kruskal算法的高效实现

- 利用**不相交集**的数据结构，这些集合操作仅需每操作 $O(\alpha(n))$ 的时间复杂度，其中 $\alpha(\cdot)$ 是一个增长极其缓慢的函数，一般可以认为 $\alpha(n) \leq 4$ 。该时间复杂度仅略高于 $O(1)$ ，而**大大低于图的遍历所需的 $O(n)$** 。



9.2 等价关系、等价类和不相交集

- 不相交集与数学中**等价关系、等价类**的概念密切相关
- 元素之间的等价关系自然地定义了若干不相交集的集合
- 因此，不相交集常常用于处理等价性查询的问题
- 例如，两个顶点在同一连通分量中就可以看做一种等价性



9.2 等价关系、等价类和不相交集

等价性查询的一般定义：

定义 9-1. （等价关系）称在集合 X 上的二元关系 \sim 为一个等价关系，若其满足：

- （自反性） $\forall a \in X$, 有 $a \sim a$;
- （对称性） $\forall a, b \in X$, 若 $a \sim b$, 则 $b \sim a$;
- （传递性） $\forall a, b, c \in X$, 若 $a \sim b$, $b \sim c$, 则 $a \sim c$ 。



9.2 等价关系、等价类和不相交集

- 一个最常见的等价关系是定义在整数集 \mathbb{Z} 上的相等关系 $=$ 。不难验证，该关系满足定义9-1中的三条性质。
- 等价关系将“相等”的概念推广到了一般的集合。

例 9.1. 不同问题中的等价关系：

- ✓ 对于平面上全部三角形构成的集合 R^6 ，三角形之间的全等关系 \cong 以及相似关系 \sim 均为等价关系
- ✓ 对于无向图 $G=(V,E)$ 中顶点构成的集合 V ，定义代表 \sim 顶点间的连通性，即 $\forall u, v \in X$, $u \sim v$ 当且仅当 u, v 连通，则顶点间的连通性是一个等价关系。
- ✓ 对于所有生物构成的集合，两种生物是否属于同一科构成一个等价关系。



9.2 等价关系、等价类和不相交集

对集合中的任意元素，称所有与其等价的元素为一个等价类：

- **定义 9-2.** (**等价类**) 给定集合 X 和等价关系 \sim ，定义某一元素 $a \in X$ 的等价类为 $\{x \in X, x \sim a\}$ 。

等价关系把集合划分成若干个不相交的等价类，每个等价类中的元素互相等价。
称所有等价类构成的集合为一个商集。

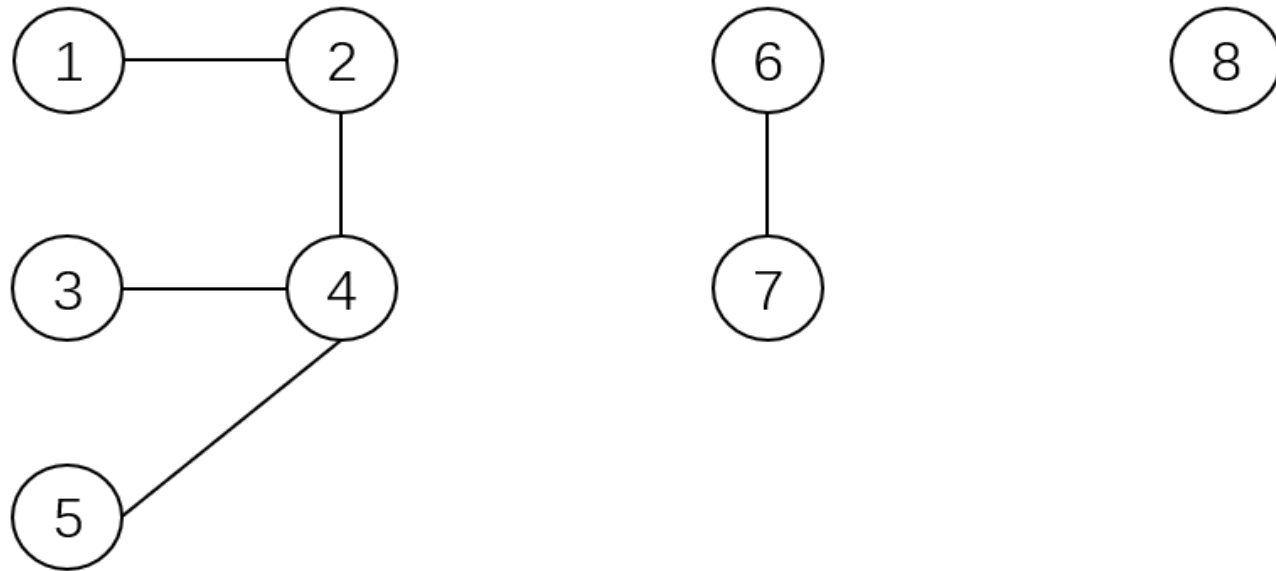
- **定义 9-3.** (**商集**) 集合 X 关于等价关系 \sim 的商集记作 X/\sim ，定义为：
 - $X/\sim := \{\{x \in X, x \sim a\}, a \in X\}$.

商集是一系列集合，这些集合彼此不相交，并且其并集是全集 X 。这与不相交集的概念恰好对应。



9.2 等价关系、等价类和不相交集

- **例 9.2.** 在下图所示的无向图 $G = (V, E)$ 中，考虑顶点间的连通关系 \sim （见例9.1）。
- 该连通关系将点集 V 划分为了三个等价类，分别为 $\{1, 2, 3, 4, 5\}$ 、 $\{6, 7\}$ 和 $\{8\}$ 。这些等价类彼此不相交，且它们的并集为全集 V 。本例中，每个等价类是一个无向图中的连通分量。



无向图 $G = (V, E)$ 中连通性等价关系定义的等价类



9.2 等价关系、等价类和不相交集

- 不相交集可以用于等价性的动态查询。
- 等价性的查询在计算机科学中有广泛应用。例如，在编译器的设计中，用于判断符号地址的等价性。

不相交集维护某集合 X 关于等价关系 \sim 的商集 X/\sim

等价关系的增加对应Union操作，即将两个等价类合并

等价性的查询对应Find操作。若要查询两元素 x 、 y 的等价性，
需判断是否有 $\text{Find}(x) = \text{Find}(y)$



9.3 不相交集的存储实现

- 不相交集数据结构定义：

- 给定 n 个元素组成的集合 X 。不失一般性，令 $X = \{1, 2, \dots, n\}$ 。

不相交集的数据结构动态维护了集合 X 的一组划分，即若干个不相交的集合 X_1, \dots, X_m ，满足：

- $\forall i \neq j, X_i \cap X_j = \emptyset$;
- $\bigcup_{i=1}^m X_i = X$ 。

- 例：考虑集合 $X = \{1, 2, 3, 4, 5, 6, 7, 8\}$ 。
- 合法的划分： $\{\{1, 2, 3, 4, 5\}, \{6, 7\}, \{8\}\}$ 或 $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}\}$;
- 以下不是集合 X 的划分： $\{\{1, 2, 3, 4, 5\}, \{6, 7\}\}$ 和 $\{\{1, 2, 3\}, \{3, 4, 5\}, \{5, 6, 7, 8\}\}$ 。



9.3 不相交集的存储实现

不相交集的数据结构需要动态处理集合的合并和查询操作，其ADT定义如下：

代码 9-1 不相交集的抽象数据类型定义

ADT DisjointSet {

数据对象：

n 个元素构成的全集 X 。

数据关系：

$\{ \langle i, j \rangle \mid i, j \in X \}$ 表示 i 、 j 属于同一个集合。

基本操作：

InitSet(set, n):

建立 n 个不相交的集合 X_1, \dots, X_n ，其中每个集合初始只有一个元素： $X_i = \{i\}, \forall i = 1, \dots, n$ 。

DestorySet(set):

释放不相交集 set 所占用的所有空间。

Find(set, x): 查询元素 x 所在的集合。

Union(set, x, y): 合并元素 x 和元素 y 所在的集合。

}



9.3 不相交集的存储实现

- 思考：Find操作查询元素所在的集合，输入是一个元素，输出是一个集合.....
 - 如何表示一个集合？
- 由于集合是动态变化的，难以实时给集合从1开始按顺序编号
- 技巧：规定每个集合中 x 需要有一个“代表”元素 $a \in X$ ，该元素的编号即为集合编号，即
 - $\text{Find}(x) = a, \forall x \in X$
- 例：对于划分 $\{\{1, 2, 3, 4, 5\}, \{6, 7\}, \{8\}\}$ ，可规定集合 $\{1, 2, 3, 4, 5\}$ 代表元素为1，则 $\text{Find}(2)=\text{Find}(3)=\text{Find}(4)=\text{Find}(5)=1$ 。



9.4 不相交集的基本运算实现

- 不相交集可利用森林的数据结构实现：
 - 每个元素 x 只需维护其父亲结点 $x.parent$ 。
 - 特别地，规定根结点的父亲结点是其本身，即 $x.parent=x$ 。
 - 规定森林中的每棵树代表一个不相交集的集合，且其“代表”元素是其根结点。
 - InitSet操作只需定义 n 棵仅含根结点的树，Find操作只需找到要查询的元素 x 对应的根结点。而Union操作只需将 x 和 y 元素各自对应的根结点合并。



9.4 不相交集的基本运算实现

算法9-1: 初始化不相交集InitSet(set, n)

输入: 元素的数量 n , 不相交集 $set=\{1, 2, \dots, n\}$ 。

输出: 初始化后的集合 set 。

1. **for** 每个元素 $x \in set$ **do**
2. | $x.parent \leftarrow x$
3. **end**

算法9-2: 查找元素所在的集合Find(set, x)

输入: 不相交集 set 中待查找的元素 x 。

输出: 元素 x 所在树的根顶点。

1. **while** $x \neq x.parent$ **do**
2. | $x \leftarrow x.parent$
3. **end**
4. **return** x

算法9-3: 合并两个元素所在的集合Union(set, x, y)

输入: 不相交集 set 中的两个元素 x 和 y 。

输出: 合并 x 和 y 各自所在集合后的不相交集 set 。

1. $i \leftarrow \text{Find}(set, x)$
2. $j \leftarrow \text{Find}(set, y)$
3. $i.parent \leftarrow j$



9.4 不相交集的基本运算实现

- **问题**：在上述实现中，InitSet、Find、Union的操作时间复杂度均可达 $O(n)$
- 考虑森林退化成一条链的情况： $i.parent = i - 1, \forall i = 2, 3, \dots, n$ 。
- 这种情况下，每次Find(set, n)均需要 $O(n)$ 的时间复杂度，而每次Union($set, n - 1, n$)也需要 $O(n)$ 的时间复杂度。



朴素的不相交集合实现的最坏情况



9.4.1 按秩合并

- **按秩合并**策略为每个根结点 x 引入一个秩 $x.rank$ ，**根结点的秩**反映了以该结点为**根的子树“大小”**，考虑利用树的高度作为树的秩：
- 对每个根节点 x ，定义其秩 $x.rank$ 为**以 x 为根的树的高度减一**。
- 特别地，仅由单个结点构成的树的根结点秩为0。
- 在合并时总是把**秩小**的树根**合并到秩大**的树根：增加树的“宽度”而非“高度”
- 在这种情况下，可保证树的“平衡性”：
 - 深度为 l 的子树至少有 $2^l - 1$ 个结点。



9.4.1 按秩合并

- 采取按秩合并策略后，InitSet和Union操作的实现调整为算法9-4、算法9-5：

算法9-4： 初始化采用按秩合并策略的不相交集InitSet(set, n)

输入： 元素的数量 n ，不相交集 $set=\{1, 2, \dots, n\}$ 。
输出： 初始化后的集合 set 。

```
1. for 每个元素  $x \in set$  do
2.   |  $x.parent \leftarrow x$ 
3.   |  $x.rank \leftarrow 0$ 
4. End
```

算法9-5： 利用按秩合并策略合并两个元素所在的集合Union(set, x, y)

输入： 不相交集 set 中的两个元素 x 和 y 。
输出： 合并 x 和 y 各自所在集合后的不相交集 set 。

```
1.  $i \leftarrow \text{Find}(set, x)$ 
2.  $j \leftarrow \text{Find}(set, y)$ 
3. if  $i.rank > j.rank$  then
4.   |  $j.parent \leftarrow i$ 
5. else if  $i.rank < j.rank$  then
6.   |  $i.parent \leftarrow j$ 
7. else //  $i.rank = j.rank$ 
8.   |  $i.parent \leftarrow j$ 
9.   |  $j.rank \leftarrow j.rank + 1$ 
10. end
```



9.4.1 按秩合并

- **引理 9-1.** 在采用按秩合并策略的不相交交集算法运行过程中，对于任意子树，若其根结点秩为 r ，则该子树中至少有 2^r 个结点。

证明.

利用归纳法，对子树根结点的秩 r 进行归纳：

1. $r = 0$ 时子树中仅含一个结点，命题成立。

2. 假设命题对所有不超过 $r - 1$ 的秩成立。考虑根结点 $root$ 的秩为 $r > 0$ 的子树：

由于 $r = root.rank > 0$ ，根结点 $root$ 的秩必然是与另一棵根为 $root'$ 的树进行Union操作时设置而来。



不失一般性，设该次Union前有：

$root.rank = root'.rank = r - 1$

否则该次Union前一定存在某次
 $Union(root, root'')$ ，且

$root.rank = root''.rank = r - 1$ ；

而Union操作不会令树中的结点数目减少。

3. 由归纳假设，可知此时以 $root$ 和 $root'$ 为根的子树中都各有至少 2^{r-1} 个结点，故合并后的子树中至少有 2^r 个结点。

4. 由归纳法，命题对所有 $r \in \mathbb{N}$ 成立。



9.4.1 按秩合并

引理 9-1. 在采用按秩合并策略的不相交集算法运行过程中，对于任意子树，若其根结点秩为 r ，则该子树中至少有 2^r 个结点。

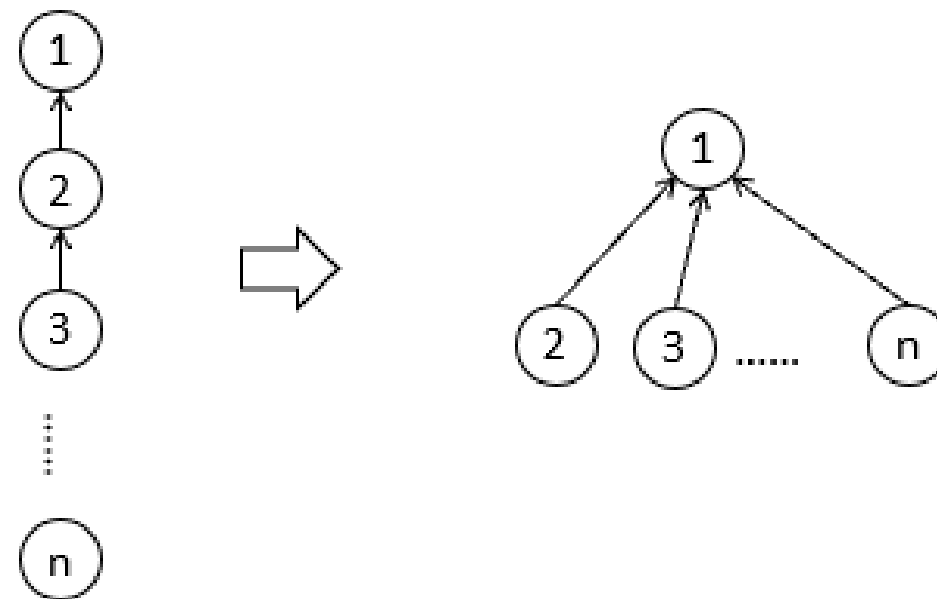
- 每次Find、Union操作的时间复杂度取决于查找链的长度，即Find操作找到的根结点的秩加一。最坏情况下，全部的 n 个结点构成一棵树。根据引理9-1，该树根的秩不超过 $\log n$ 。因此，不相交集每次Find、Union操作的时间复杂度为 $O(\log n)$ 。



9.4.2 路径压缩

- 假设某次查找时，路径深度过大，导致该次查找的时间成本很高。我们只需调整森林的结构，将查找路径上的所有结点直接连接至根。
- 这样，下次查找这条路径上的结点时只需 $O(1)$ 时间即可完成。这个技巧称为**路径压缩 (Path Compression)**。

Find(n)



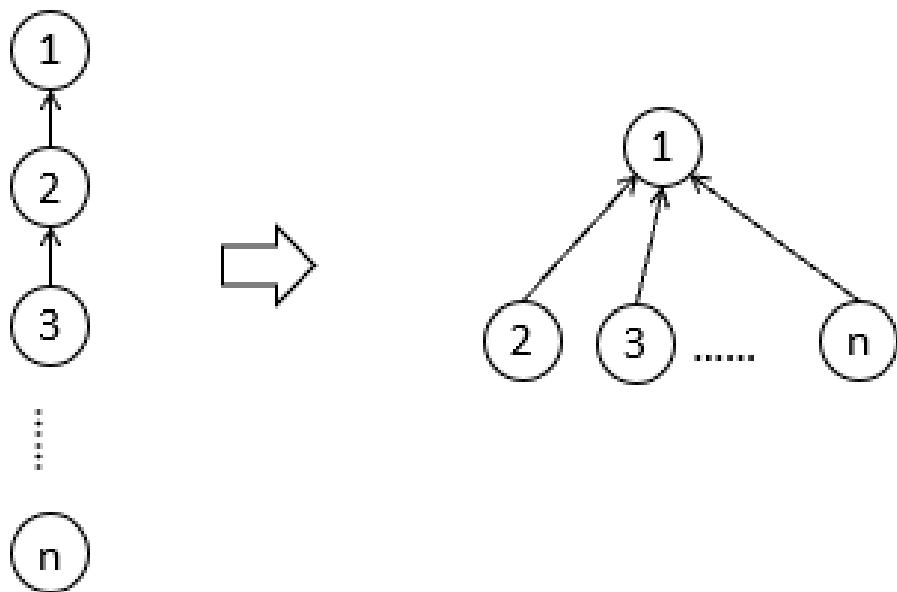
路径压缩



9.4.2 路径压缩

利用路径压缩技巧，不相交集的查找可以重写成为算法9-6。该算法首先递归地查找到根 r ，之后逐层返回，并将路径上所有结点的父结点均设为 r 。

Find(n)



路径压缩

算法9-6: 以路径压缩策略查找元素所在的集合Find(set, x)

输入：不相交集 set 中待查找的元素 x 。

输出：元素 x 所在树的根结点。

```
1. if  $x \neq x.parent$  then
2.   |  $x.parent \leftarrow \text{Find}(set, x.parent)$ 
3. end
4. return  $x.parent$ 
```



9.4.2 路径压缩

- 下图演示了同时采用按秩合并和路径压缩策略时，8个元素构成的不相交集的合并过程。

初始时，所有结点均不相连，结点的秩均为0



Union(5, 1)时，两结点具有相同的秩，此时将5合并到1，并将结点的秩置为1



Union(6, 5)时，两棵秩为1的树合并成为一棵秩为2的树



树有 $2^3=8$ 个结点，在可能出现的秩为3的树中是结点数目最少的



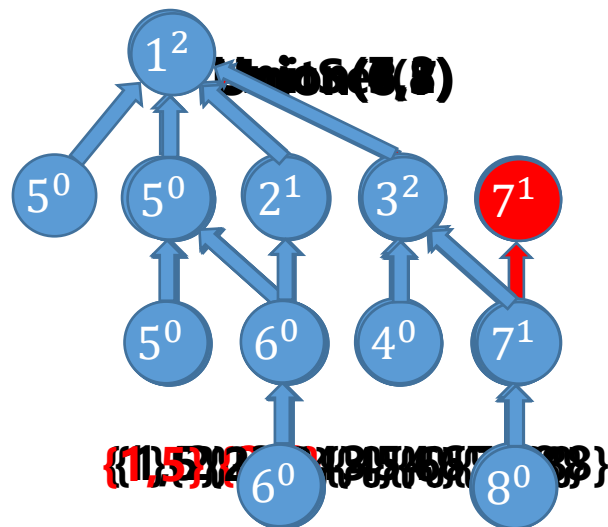
合并操作将两棵秩为2的树合并成一棵秩为3的树



Union(3, 6)时，首先执行的Find(6)进行了路径压缩，将6直接连接至1

*此时2成为叶子结点，但其秩未发生改变。结点1的秩也仍是2

Union(3, 6)



采用了按秩合并和路径压缩的不相交集合运行示例

(*图中 i^r 表示编号为 i ，秩为 r 的顶点)



9.4.2 路径压缩

- 实现不相交集非常简单，但分析其时间复杂度却非常困难。虽然单个操作的时间复杂度较高，最坏可达 $O(\log n)$ 。然而，由于路径压缩策略的存在，每次访问完一条路径后，下次访问该路径上结点的时间复杂度就会降低，因此算法整体的时间复杂度仍然较低。





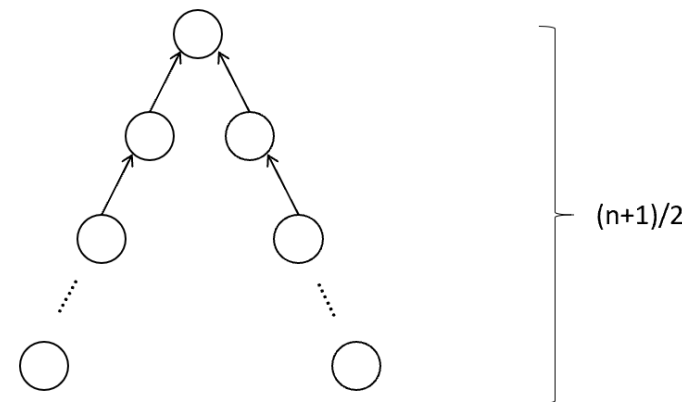
9.4.2 路径压缩

- 使用均摊分析技巧，考虑连续的 m 个操作的总时间复杂度。
- 可证明,同时使用按秩合并和路径压缩后， n 个元素组成的不相交集上进行 m 次Find/Union操作的时间复杂度为 $O(m\alpha(n))$ ，即每次操作的均摊时间复杂度为 $O(\alpha(n))$ 。其中：
 - $\alpha(n)$ 是增长非常缓慢的函数，对 $n < 10^{80}$ 均有 $\alpha(n) \leq 4$ 。
 - 10^{80} 与可观测的宇宙中原子的数目具有相当的数量级。
- 因此，Find/Union操作几乎具有常数时间复杂度。
- 尽管如此，不能称Find/Union具有 $O(1)$ 的时间复杂度，因为
 - $n \rightarrow \infty$ 时, 仍有 $\alpha(n) \rightarrow \infty$
- 可证明 $O(\alpha(n))$ 时间复杂度上界是紧的，即不相交集不存在 $O(1)$ 时间复杂度的算法。



9.5 不相交集的应用：最近公共祖先问题

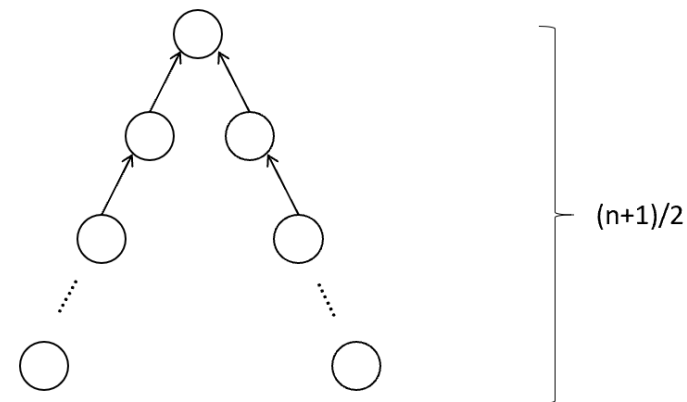
- 在一棵树中，结点 u 与结点 v 的公共祖先是所有同时是 u 和 v 祖先的结点，而其中深度最大（即离根最远）的结点称为**最近公共祖先**。
- 给定任意两个结点 u 和 v ，公共祖先构成一条以根结点为起点的链，而其中最近公共祖先是**唯一的**。
- 最近公共祖先**LCA** (Lowest Common Ancestor) 最近公共祖先问题蛮力算法的坏情况问题考虑 m 个形如 (u, v) 的查询，每次需要查询树 T 中结点 u 和 v 的最近公共祖先。





9.5 不相交集的应用：最近公共祖先问题

- LCA问题可用蛮力算法求解：
 - 首先，从结点 u 回溯至根，将路径上每个结点均做标记；
 - 然后，从结点 v 回溯至根，路径上遇到的第一个有标记的结点即为结点 u 、 v 的最近公共祖先。
- 然而，该蛮力算法的时间复杂度较高。
- 如图，若树 T 中有 n 个结点，则蛮力算法在最坏情况下每次查询的时间复杂度可达 $O(n)$ 。



最近公共祖先问题蛮力算法的坏情况



9.5 不相交集的应用：最近公共祖先问题

- 求解最近公共祖先问题的离线算法（Tarjan算法）
- 一次性返回所有 m 个查询的结果，总时间复杂度仅为 $O(m\alpha(n) + n)$ 。
- Tarjan算法对树进行深度优先遍历，并按遍历的顺序处理询问。
- 若在遍历某结点 v 时，关于该结点的询问 (u, v) 的另一端点 u 已被访问过（“闭询问”），则Tarjan算法立即处理该询问
- 否则，若 u 尚未被访问（“开询问”），则将此询问留待访问 u 时完成。



9.5 不相交集的应用：最近公共祖先问题

- 记号：结点 r 的左子树记为 L ，右子树记为 R 。
- 基本事实： L 中的结点 u 与 R 中的结点 v ，最近公共祖先一定是 r 。
- 关键：对于一批查询，如何批量地找到每一个查询 (u, v) 的最近公共祖先 r 的两个子树？
- 答案：DFS!
- 在DFS中，搜索过程对每个结点 r 有三次“接触”：
 - 第一次：访问该节点 r
 - 第二次：该节点的左子树 L 访问完毕后，第一次回溯到该节点
 - 第三次：该节点的右子树 R 访问完毕后，第二次回溯到该节点



9.5 不相交集的应用：最近公共祖先问题

- 设算法已经访问完了根为 r 的子树 $T_1 \in T$ ，并且处理完毕了所有关于 T_1 **闭询问**。
- 需要保留哪些信息以在未来处理 T_1 的**开询问**呢？
- 不失一般性，考虑开询问 (u, v) ，其中 $u \in T_1$ 已经被访问，而 v 尚未被访问。
- 因为 v 不是子树 T_1 中的结点（否则 (u, v) 应该是闭询问），所以 (u, v) 的 LCA 必然不会是子树 T_1 中的结点，最多只能是子树根 r 的父亲 **$r.parent$** 。
- 因此，对于后续所有关于 T_1 的开询问，保留子树的结构已无意义，我们可以将子树中的所有结点合并到结点 **$r.parent$** 上。
- 在后续遍历中， **$r.parent$** 代表子树 T_1 中的所有结点，无需再考虑子树 T_1 的具体结构。该合并操作可以用**不相交集**实现，如算法9-7。



9.5 不相交集的应用：最近公共祖先问题

- 在DFS中，搜索过程对每个结点 r 有三次接触：
 - 第一次：访问该节点 r
 - 第二次：该节点的左子树 L 访问完毕后，第一次回溯到该节点
 - 第三次：该节点的右子树 R 访问完毕后，第二次回溯到该节点
- 观察访问次序：第二次结束时，可以得知 L 的所有元素！第二次和第三次之间，访问的结点都在 R 中。
- 如果我们在第二次结束时，把所有 L 中的元素都合并到 r 中，然后在访问 R 期间，对于查询($u \in L, v \in R$)，都可以知道它们的最近公共祖先为 r ！
- Tarjan算法如何实现上述思想：
 - 第一次回溯到某个节点时，将它的左子树都合并到该节点上
 - 第二次回溯到某个节点时，将它的右子树也合并到该节点上。
 - 输出LCA(u, v)的时机：若 u 与 v 的最近公共祖先是 r 。1. u 已经访问过，2. u 所在 r 的左子树 L 已经被合并到了 r 上。3. v 的所有子树刚刚合并到 v 上，此时正好第三次接触到 v 。4. 尚未第三次接触到 r ，此时正在访问 r 的右子树 R 。



9.5 不相交集的应用：最近公共祖先问题

算法9-7: Tarjan算法求解最近公共祖先 $LCA(tree, P, u)$

输入: 树 $tree$, 查询集 $P = \{(u, v)\}$, 当前结点 u 。

输出: 查询结果LCA。

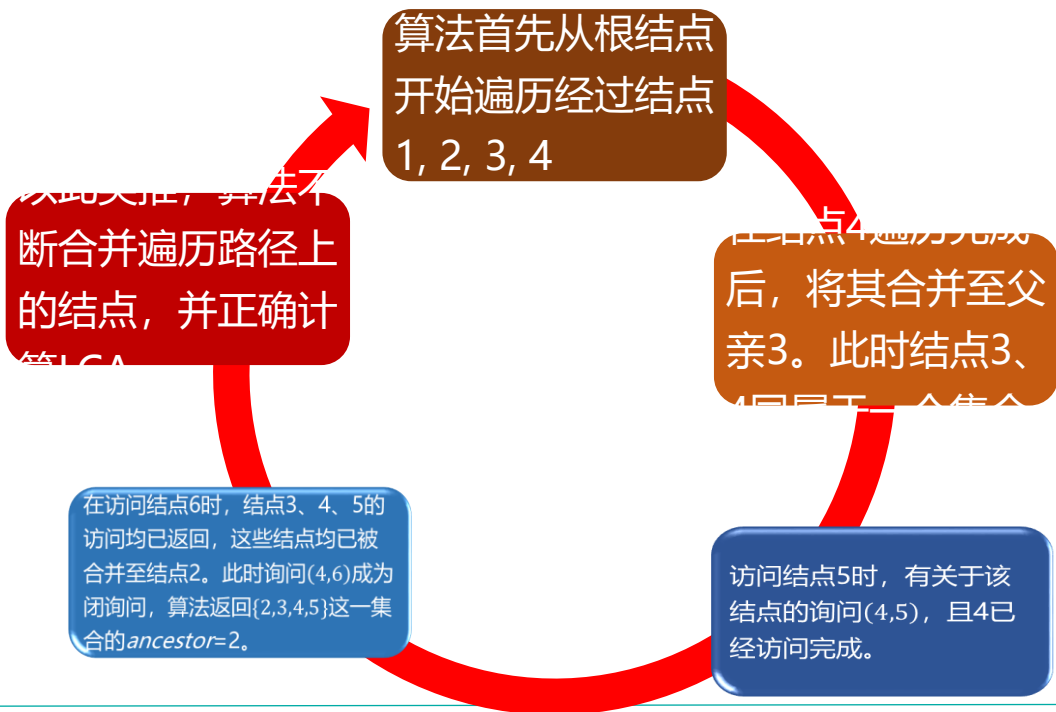
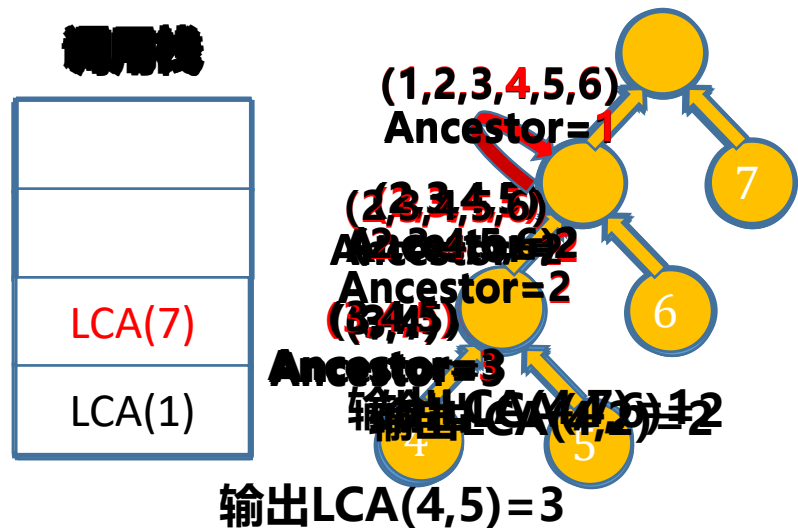
```
1.  $set \leftarrow \text{InitSet}(n)$  // 初始化不相交集
2.  $ancestor[\text{Find}(set, u)] \leftarrow u$  //初始化
3. for  $u$ 的每个孩子  $v$  do
4.     |  $LCA(set, tree, P, v)$  //深度优先遍历  $u$ 的所有子树
5.     |  $\text{Union}(set, u, v)$  //将子树并到根结点  $u$ 
6.     |  $ancestor[\text{Find}(set, u)] \leftarrow u$  //记录这棵树的根是  $u$ 
7. end
8.  $visited[u] \leftarrow \text{true}$ 
9. for  $(u, v) \in P$ 的每个结点  $v$  do
10.    | if  $visited[v]$  then
11.    | | print  $ancestor[\text{Find}(set, v)]$  //输出  $u$  和  $v$ 的LCA
12.    | end
13. End
```

- 算法的执行过程中，我们维护了两个数组visited和ancestor。
- 其中visited表示结点是否访问完成，用以判断询问是否可完成。
- ancestor表示结点集合（子树）的根。这是因为在采用了按秩合并策略后，Find找到的“根”并不一定是子树实际的根，因此需要额外记录实际的根。



9.5 不相交集的应用：最近公共祖先问题

- 右图是算法运行过程的示意图。
- 本例中有4个询问：分别为LCA(4,5)、LCA(4,6)、LCA(4,2)和LCA(4,7)。
- 右图中依次展示了结点4, 5, 3, 6, 2, 7的孩子访问完成，即将输出LCA时刻的场景，对应算法9-7第7行。

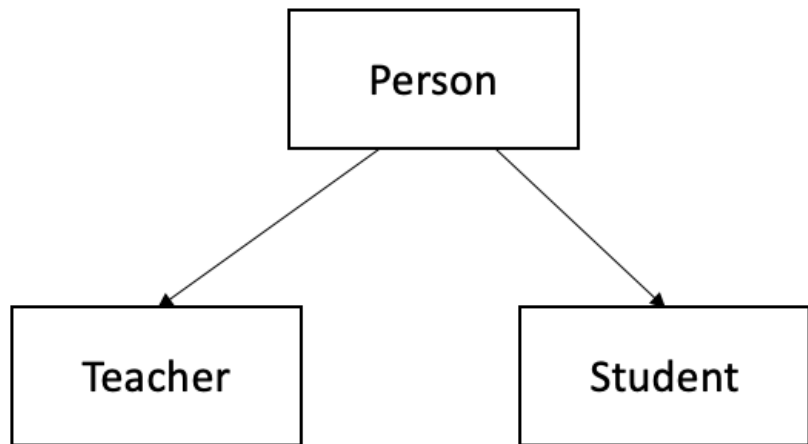


*取决于不相交集的实现，3、4均可能是该集合的代表元素。不失一般性设 $\text{Find}(3) = \text{Find}(4) = 3$ ，则此时 $\text{ancestor}[3] = 3$ 。

*算法输出(4, 5)的LCA为 $\text{ancestor}[\text{Find}(4)] = \text{ancestor}[3] = 3$



9.7 应用场景:面向对象的编程语言



三个类间的继承关系

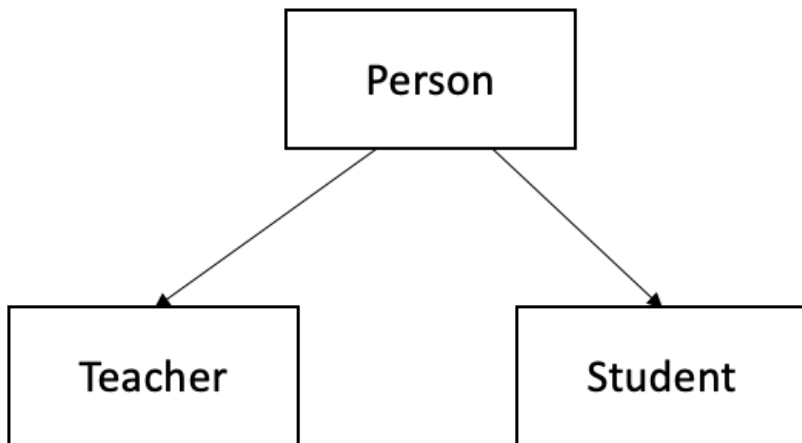
不相交集在许多计算机科学的真实问题上有应用。如离线最小值查询，求控制流图的支配树，类型推断，实现属性文法（Property Grammar）等。近年来，该数据结构在图像处理，数据库和量子计算等领域也有应用。

这里介绍不相交集在面向对象的编程语言中的应用。



9.7 应用场景:面向对象的编程语言

面向对象的编程语言中，变量的类型可以是用户定义的类（class），类之间可以有继承关系。例如说，下列C++代码定义了People、Teacher、Student三个类，其继承关系如图。



三个类间的继承关系

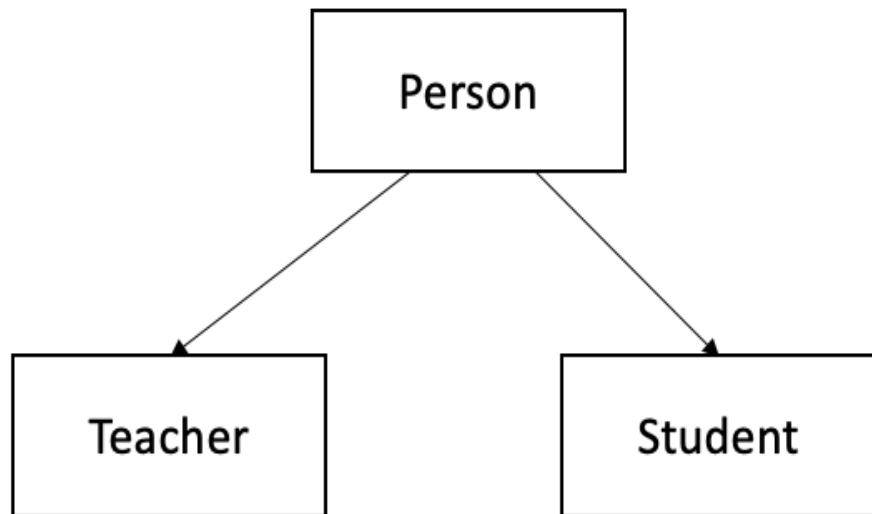
```
class Person { ... };
```

```
class Teacher : public Person { ... };
```

```
class Student : public Person { ... };
```



9.7 应用场景:面向对象的编程语言



三个类间的继承关系

假如我们定义了如下二元运算Talk:

```
string talk(Person a, Person b);
```

```
string talk(Teacher a, Teacher b);
```

```
string talk(Student a, Student b);
```

不同类别之间的Talk函数可以有不同行为。

例如，任意两个Person都可以谈论天气，而只有两个Student才会讨论功课。

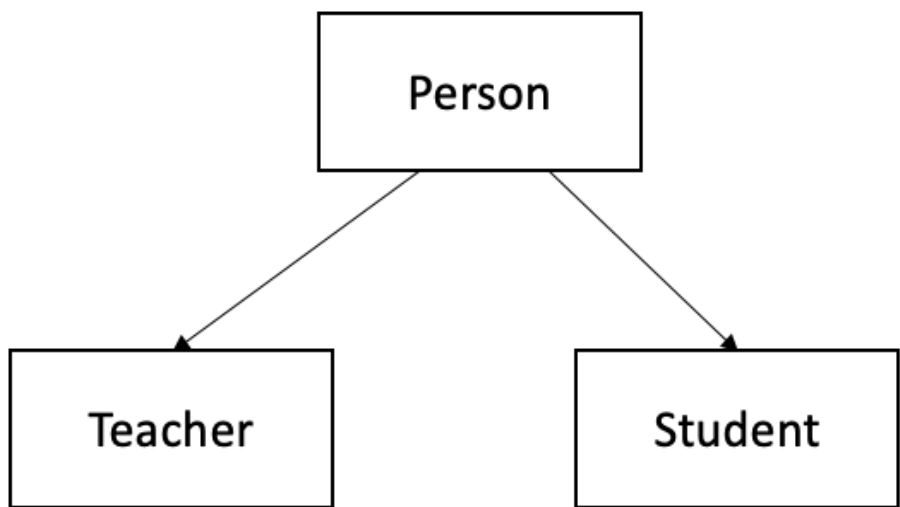


9.7 应用场景:面向对象的编程语言

注意，这里我们没有定义两个参数分别是Teacher和Student的Talk函数。

假如有一个Teacher类型的变量alice和一个Student类型的变量bob，此时若调用talk(alice, bob)会发生什么呢？

实际上，编译器会寻找两个类的最近公共祖先，来调用两个People之间的Talk函数。而最近公共祖先问题又可以用不相交集来解决。



三个类间的继承关系

The background is a solid teal color with a subtle pattern of thin, light teal lines forming a grid and perspective lines. Several 3D cubes of varying sizes are scattered across the scene, some in darker teal and others in a lighter teal, creating a sense of depth and modern design.

谢谢观看