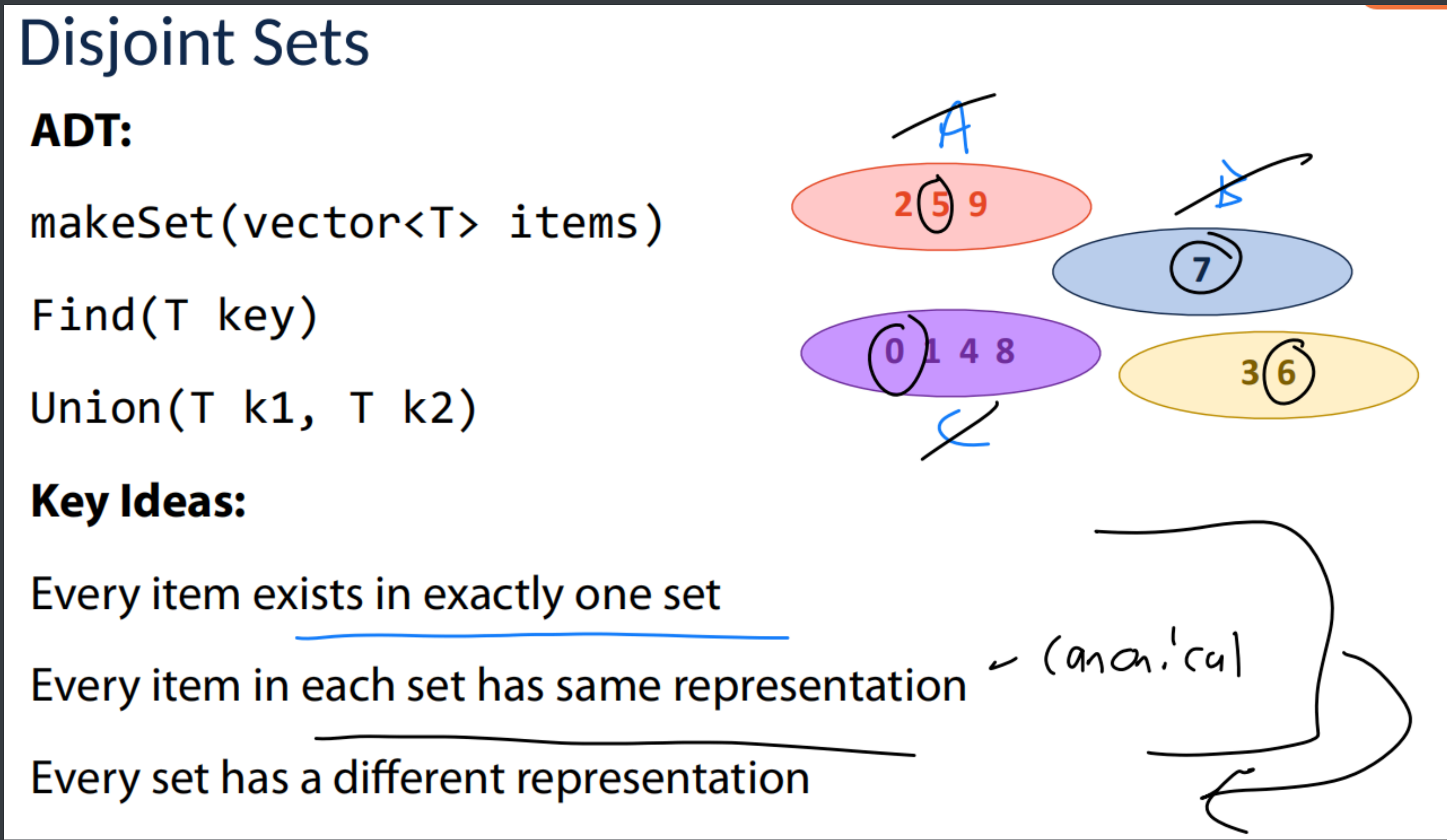


这张图是关于**并查集（Disjoint Sets）**的数据结构的介绍。它是一种用于处理集合合并与查询操作的高效结构，常用于图的连通性判断、Kruskal 最小生成树算法等。

一、图中术语解释：

1. ADT (抽象数据类型)

图中列出了并查集的三种基本操作：

- **makeSet(vector items)**
  为每个元素创建一个独立的集合（每个元素最开始都是一个集合的代表）。
- **Find(T key)**
  查找元素 `key` 所属集合的“代表元素”或“根”。
- **Union(T k1, T k2)**
  将两个不同的集合合并成一个集合。

二、图中集合划分说明：

右边用颜色区分了四个集合（红色、蓝色、紫色、黄色）：

- **集合 A**：{2, 5, 9}，代表元素是 **5**
- **集合 B**：{7}，代表元素是 **7**
- **集合 C**：{0, 1, 4, 8}，代表元素是 **0**
- **集合 D**：{3, 6}，代表元素是 **6**

> 每个集合都有一个**代表元素**（用黑圈圈出来了），也称为**canonical element**（规范代表）。

三、Key Ideas（核心思想）详解：

1. **Every item exists in exactly one set**
   每个元素只属于一个集合，集合之间不交叉。
2. **Every item in each set has same representation**
   每个集合中的元素都有相同的“代表元素”。例如在集合 A 中，`Find(2)`、`Find(5)`、`Find(9)` 都会返回 **5**。
3. **Every set has a different representation**
   每个集合的代表元素都不同，这有助于我们区分不同集合。

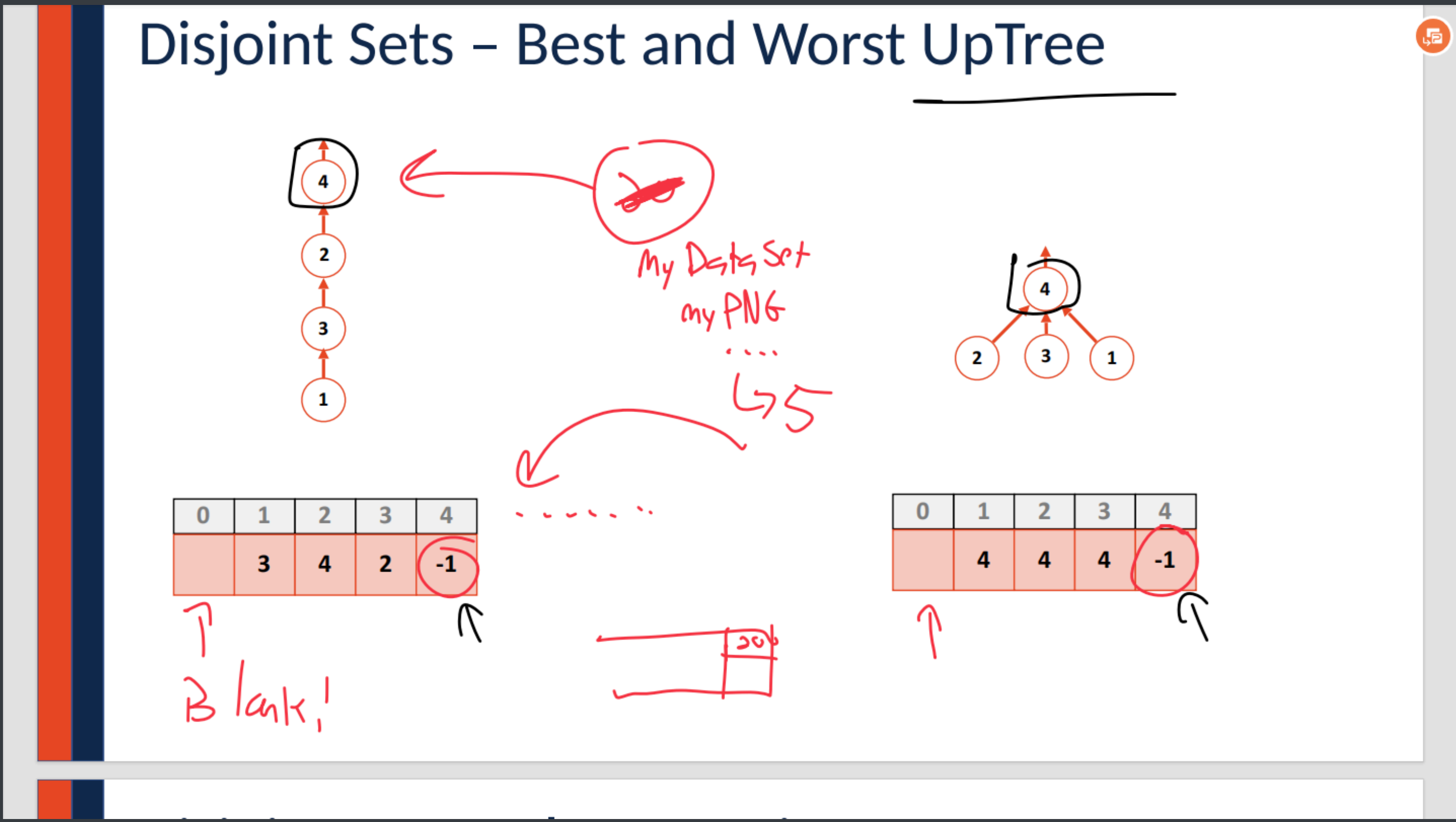
图中右侧还手写注释了 `canonical`，表示**“规范代表”**，即代表元素的含义。

四、用途和应用场景：

并查集最常见于以下场景：

- 网络连通性判断
- 图中的连通分量问题
- Kruskal 算法用于生成最小生成树
- 动态集合合并问题

如果你想我用代码演示这些操作，也可以告诉我！



这张图片展示了**并查集（Disjoint Set）*的*最佳**和**最坏**情况的树结构（UpTree），并介绍了数组如何表示这些树结构。

最佳情况（左图）：

- 在最佳情况下，**每个元素都是自己的根节点**，即每个元素都单独构成一个集合。
- 这种情况下，数组的值都为负数，表示每个集合的代表元素（根节点），例如：

```
Index: 0  1  2  3  4
Value: -1 -1 -1 -1 -1
```

- 图示说明：
  - 每个集合中的元素没有父节点，且每个元素都自己指向自己作为根节点。这意味着树的高度为 1，所有的 `Find` 操作都是 **O(1)**。
- 这种结构是最理想的状态，但通常在实际使用中难以保持，因为集合之间会经常进行合并操作。

最坏情况（右图）：

- 在最坏情况下，所有元素会形成一个链式结构，即每个元素只指向下一个元素作为父节点，导致树的高度最大化。
- 这种情况是 **O(n)**，其中 `n` 是元素数量，因为每次执行 `Find` 时，都需要沿着链遍历所有元素。

- 例如：

  ```
  Index: 0  1  2  3  4
  Value: 4  4  4  4  -1
  ```

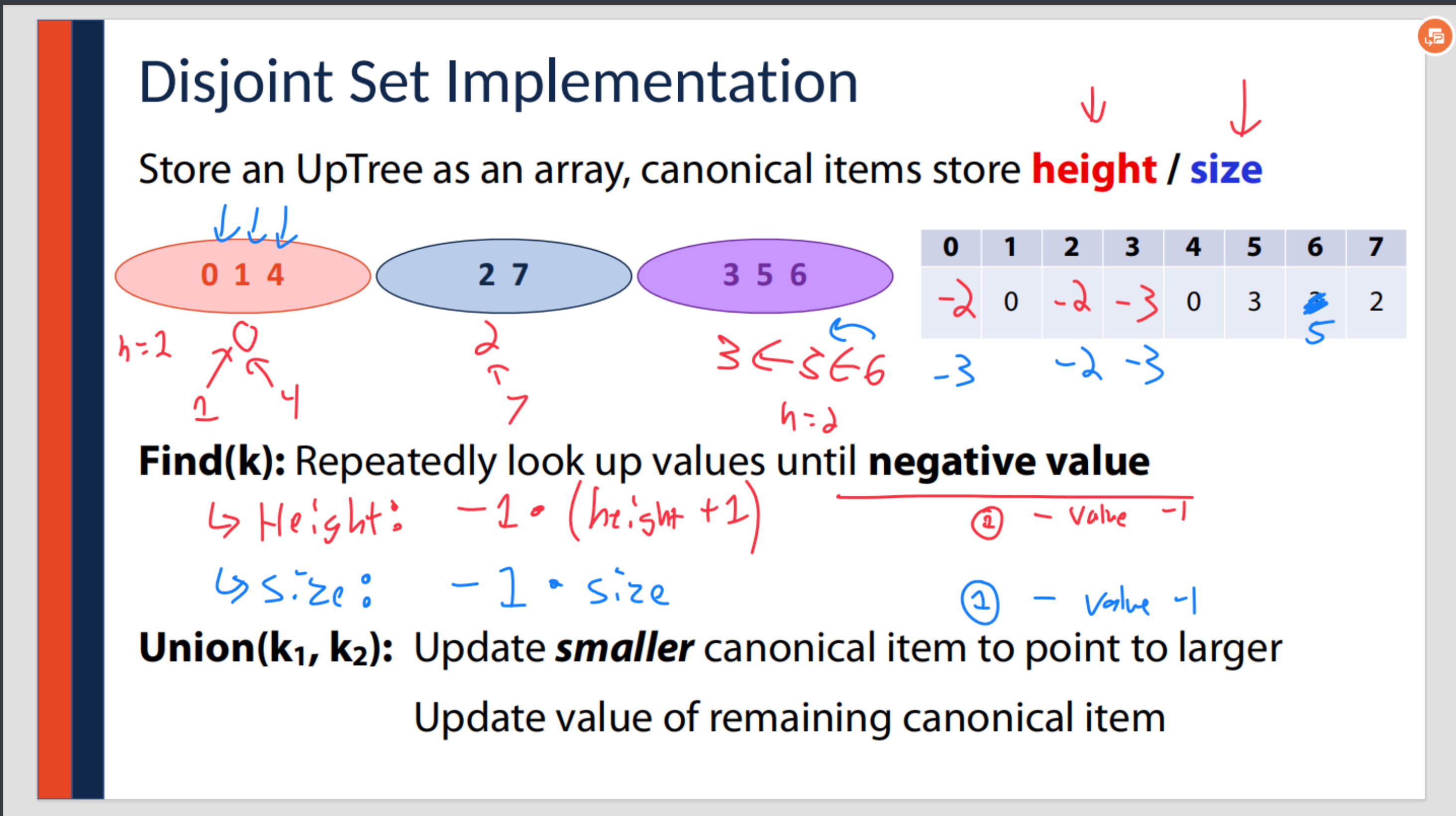
  - 在这个示例中，元素 `0`、`1`、`2` 和 `3` 都指向 `4`，而 `4` 是根节点，表示所有元素都属于同一个集合，根节点为 `4`。
- **合并后的树结构**：
  - 所有元素都形成一条链式结构，树的高度为 `n`，`Find` 操作的时间复杂度为 **O(n)**。
  - 在这种情况下，性能很差，因为每次查询都需要遍历整个链。

优化方法：

- **按高度合并（Union by Height）**：合并树时，将较小的树合并到较大的树上，从而避免形成链式结构。
- **路径压缩（Path Compression）**：在执行 `Find` 操作时，压缩路径，即将查询路径上的所有节点直接指向根节点，从而使树的结构更加扁平，进一步优化后续的查询。

总结：

- **最佳情况**是每个元素独立成集合，查找操作非常高效（O(1)）。
- **最坏情况**是所有元素形成链式结构，查找操作非常低效（O(n)）。
- 通过**按高度合并**和**路径压缩**等优化策略，可以避免最坏情况，保持树的高度较小，从而提高查询操作的效率。



这张图展示了**并查集（Disjoint Set）***的*UpTree实现方式，重点介绍了如何通过高度或大小存储树结构，并优化 `Find` 和 `Union` 操作。

UpTree 实现解析：

1. **树的表示**：
   - 每个集合使用树结构表示，并通过数组存储。在数组中，每个根节点（代表元素）存储了一个负值，表示树的高度或大小。
   - 非根节点存储的是指向父节点的索引。
2. **Find(k) 操作**：
   - `Find(k)` 操作通过查找元素 `k` 的父节点，直到找到根节点（即数组中值为负数的元素），并返回该根节点。
   - **Find** 操作的时间复杂度是 **O(h)**，其中 `h` 是树的高度。每次查找都需要沿着树的路径向上查找，直到遇到根节点。

   在图中，**Find(k)** 的操作通过递归查找元素的父节点，直到找到代表元素（即值为负数的元素）。这时：

   - **高度**（height）是用来表示树的深度（通过 `-1 * (height + 1)` 表示）。
   - **大小**（size）表示集合中元素的数量（通过 `-size` 来表示）。
3. **Union(k1, k2) 操作**：

- **Union** 操作将两个集合合并。首先，需要查找 `k1` 和 `k2` 所在集合的代表元素，然后将**较小的树**（较短或元素较少的集合）合并到**较大的树**中。
- 合并时，更新新集合的代表元素（根节点）的值，表示树的高度或大小。

图中展示的例子：

- **初始状态**：
  - 集合 `{0, 1, 4}`，树的高度为 2，根节点是 `4`。
  - 集合 `{2, 7}`，树的高度为 2，根节点是 `7`。
  - 集合 `{3, 5, 6}`，树的高度为 2，根节点是 `6`。

  对应的数组表示：

  ```
  Index:  0  1  2  3  4  5  6  7
  Value:  -2 0  -2 -3  0  3  2  7
  ```

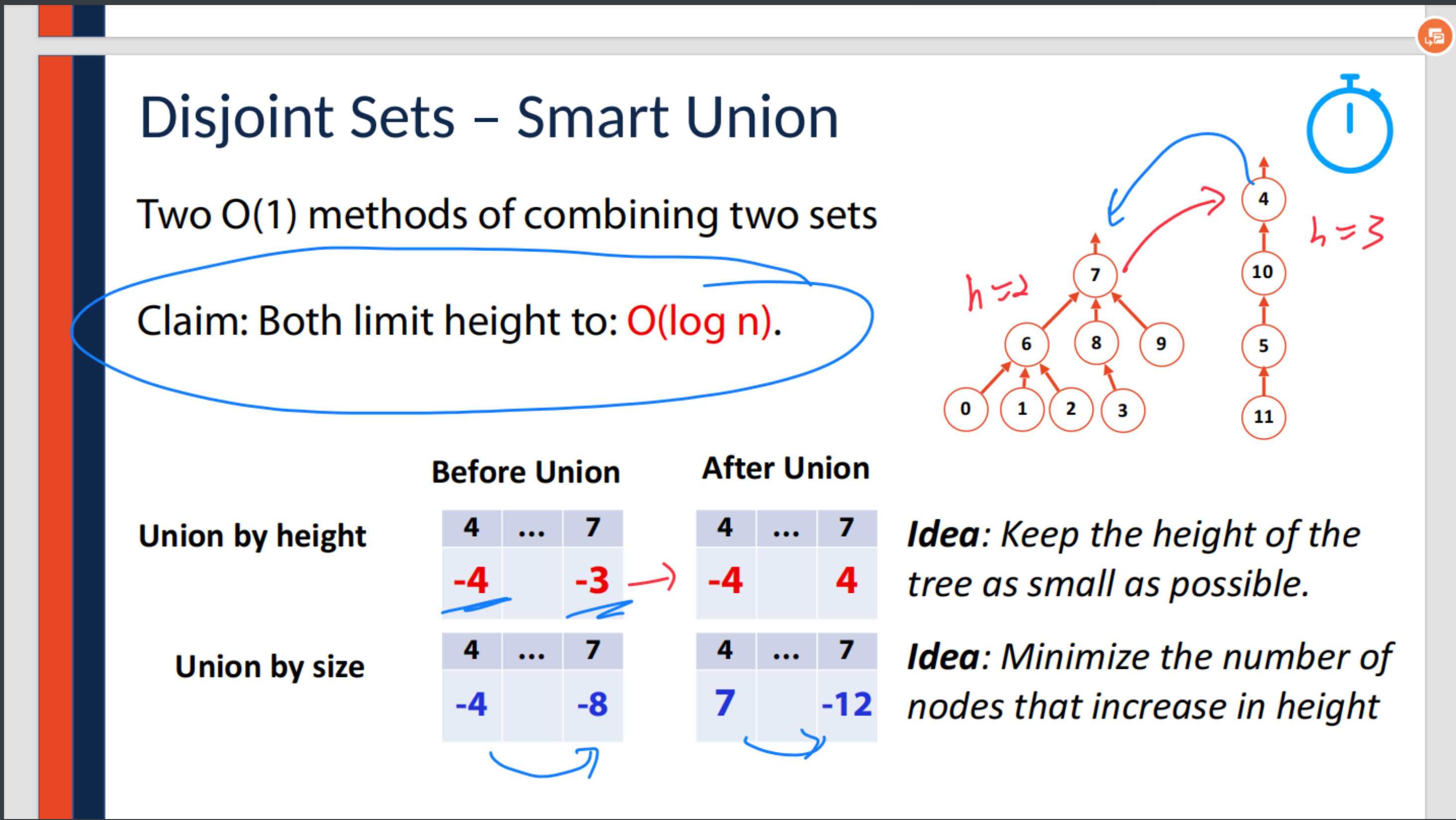
  - `s[0] = -2` 表示 `0` 是根节点，树的高度为 2。
  - `s[2] = -2` 表示 `2` 是根节点，树的高度为 2。
  - `s[4] = -2` 表示 `4` 是根节点，树的高度为 2。
- **Find(0)** 操作：
  - `Find(0)` 会沿着父节点查找，直到找到根节点。在这个例子中，根节点是 `4`，返回 `4` 作为集合的代表元素。
- **Union(4, 7)** 操作：
  - `Union(4, 7)` 操作会将两个集合合并，合并后的集合的代表元素是 `7`，并更新相应的数组值。

  合并后数组可能如下：

  ```
  Index:  0  1  2  3  4  5  6  7
  Value:  -3 0  -3 -3  0  3  2  7
  ```

总结：

- **Find(k)** 操作通过不断查找父节点，直到找到根节点，其时间复杂度为树的高度 **O(h)**。
- **Union(k1, k2)** 操作通过将较小的树合并到较大的树中，来保持树的平衡，减少查找操作的时间复杂度。



这张图展示了 **并查集（Disjoint Sets）** 中 **智能合并（Smart Union）** 的两种优化方法，分别是 **按高度合并（Union by Height）** 和 **按大小合并（Union by Size）**。图中的重点是 **限制树的高度**，确保树的高度不会增长到超过 **O(log n)**，从而优化查找操作。

两种合并方法:

1. **按高度合并（Union by Height）**：
   - 在合并两个集合时，将 **高度较小的树** 合并到 **高度较大的树** 下方，这样可以减少树的高度增长。
   - 在图中，**合并前**，树的高度为 2（`h=2`），**合并后**，树的高度为 3（`h=3`）。
   - 数组表示如下：
     - **合并前**:
       ```
       4   -4
       7   -3
       ```
     - **合并后**:
       ```
       4   -4
       7   4
       ```

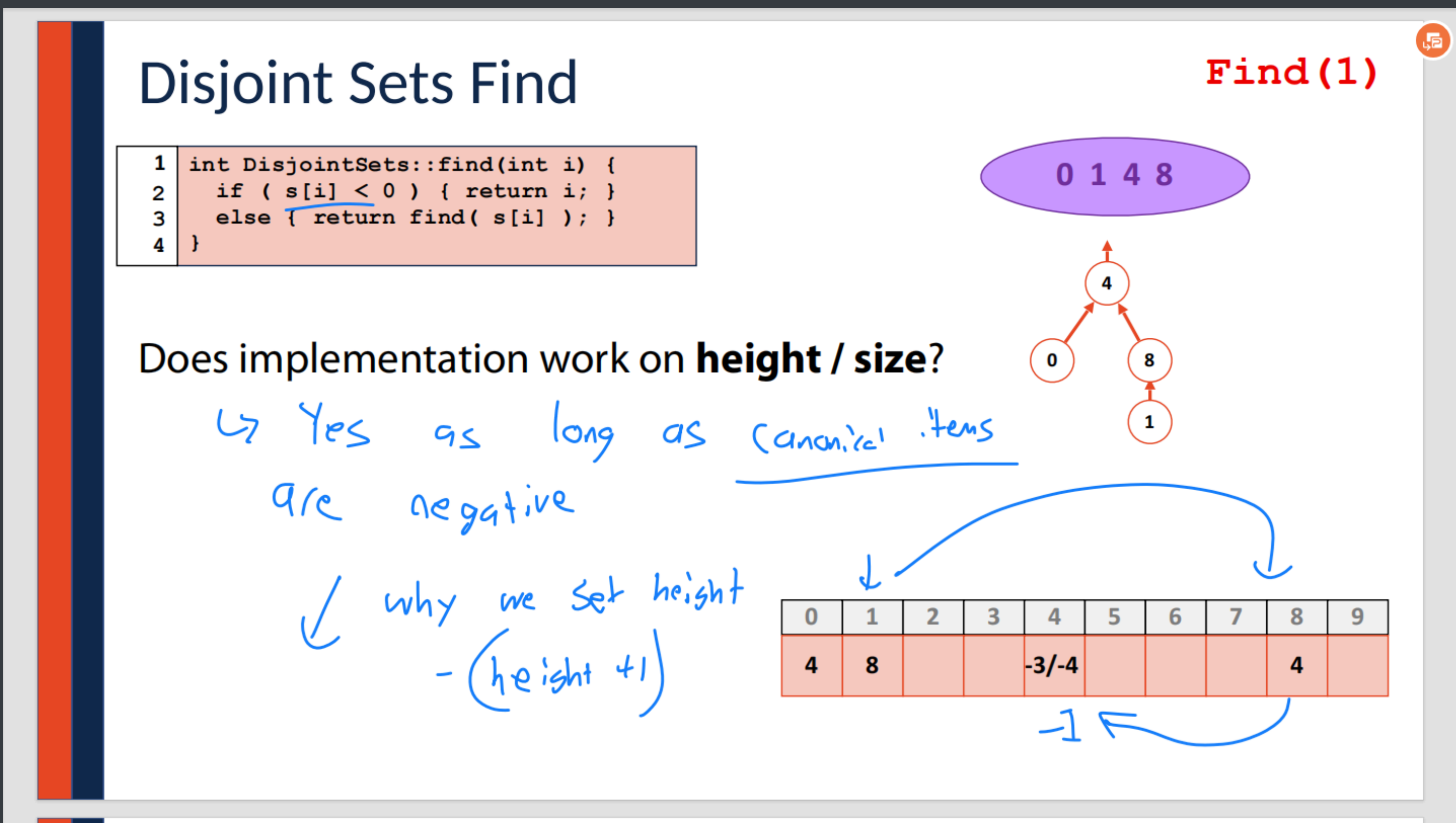
2. **按大小合并（Union by Size）**：
   - 在合并两个集合时，将 **节点数较少的树** 合并到 **节点数较多的树** 下方，这样可以避免树的高度增加。
   - 在图中，**合并前**，树的大小为 4（`-4`），**合并后**，树的大小变为 12（`-12`）。

图示说明:

- **合并前**：集合 `{4, 7}` 和集合 `{3, 5, 6}` 具有不同的高度和大小，合并前我们会选择按某种策略将它们合并。
- **合并后**：无论是按高度合并还是按大小合并，都会导致根节点的值更新，反映出树的新高度或新大小。

重点:

- **目标**：保持树的高度尽可能小，避免树的高度增长至线性。通过按高度或大小合并，可以在合并操作中避免增加不必要的树深度。
- **Claim**（声明）：无论是按高度合并还是按大小合并，都能将树的高度限制为 **O(log n)**，大大提高 **Find** 操作的效率。



这张图展示了**并查集（Disjoint Sets）***的 **Find** 操作的实现，以及如何通过*高度**或**大小**来表示树的结构。

Find 操作解析：

1. **Find 函数实现**：

    ```
    int DisjointSets::find(int i) {
        if (s[i] < 0) {
            return i;
        }
        else {
            return find(s[i]);
        }
    }
    ```

    - `find(i)` 用于查找元素 `i` 所在集合的代表元素（根节点）。
    - **递归查找**：如果 `s[i] < 0`，说明 `i` 是根节点，直接返回 `i`；如果 `s[i] >= 0`，则递归查找 `s[i]`，即查找 `i` 的父节点，直到找到根节点。

2. **高度/大小存储**：

    - 根节点存储的是一个**负数**，表示树的高度（如果按高度合并）或者集合的大小（如果按大小合并）。
    - 在图中，根节点存储的是树的**高度**，值为负数，`- (height + 1)`。
    - 非根节点存储其**父节点**的索引。

3. **数组表示**：

    - 数组 `s[]` 中，根节点的值为负数，表示树的高度或大小；非根节点存储的是指向父节点的索引。
    - 在图中，`s[0] = -4` 表示元素 `0` 是根节点，树的高度为 `3`，即高度加 1 为 `4`；`s[1] = 8` 表示元素 `1` 的父节点是 `8`。

4. **Find(1) 操作**：

    - 在图中的例子，执行 `Find(1)` 操作时，首先检查 `s[1] = 8`，然后继续查找 `s[8]`，最终找到根节点 `4`。
    - 因为根节点的值为负数（`s[4] = -1`），所以 `Find(1)` 会返回根节点 `4`。

高度/大小的选择：

- 该实现可以用于**高度**或**大小**两种方式来表示集合的结构。
- **关键点**：无论是按高度还是按大小，只要**根节点**的值是负数（表示树的高度或大小），`Find` 操作就能正确返回集合的代表元素。
- **为什么根节点存储负值**：
  - 根节点存储负值是为了区分它与非根节点（非根节点指向父节点）。通过负值可以方便地表示树的高度或集合的大小，且便于在 `Find` 操作时直接识别。

图示：

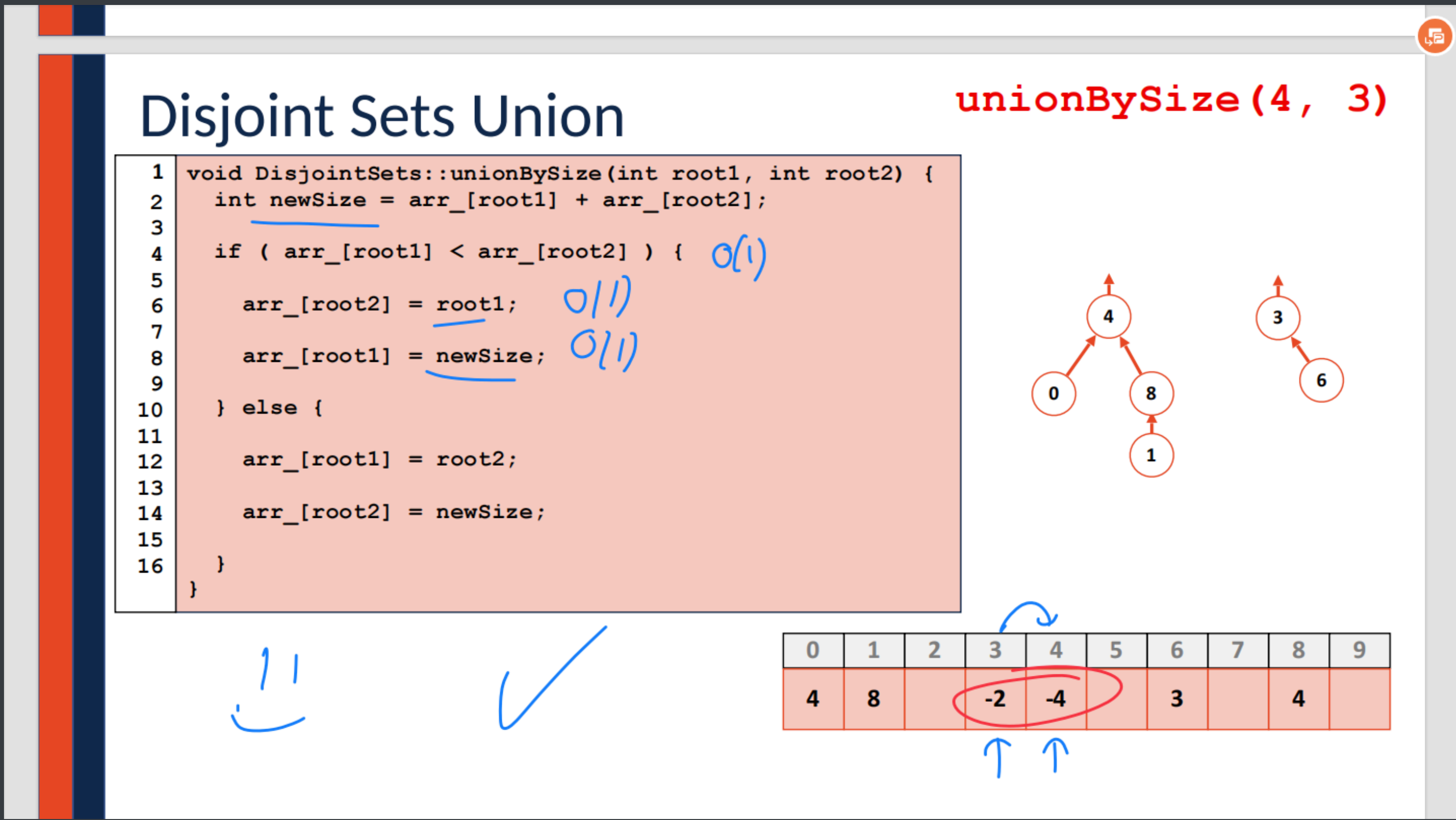
- **数组表示**：

  ```
  Index:  0  1  2  3  4  5  6  7  8  9
  Value: -4  8  -3  -4  -1  4  4  4  4
  ```

  - `s[0] = -4`：表示元素 `0` 是根节点，树的高度为 `3`。
  - `s[1] = 8`：表示元素 `1` 的父节点是 `8`，并不是根节点。

总结：

- **Find** 操作通过递归查找父节点直到根节点，利用数组中负数的值来识别根节点，并返回该根节点作为集合的代表元素。
- 通过将树的高度或集合的大小存储为负数，可以有效地进行查找操作，且实现灵活，可以用于按**高度**或**大小**合并。



这张图展示了 **并查集（Disjoint Sets）** 中的 **按大小合并（Union by Size）** 方法的实现，以及相应的代码和图示。

Union by Size 代码解释：

```
void DisjointSets::unionBySize(int root1, int root2) {
    int newSize = arr[root1] + arr[root2];  // 计算合并后树的大小

    if ( arr[root1] < arr[root2] ) {       // 如果 root1 的树比 root2 的树小
        arr[root2] = root1;                 // 将 root2 的根指向 root1
        arr[root1] = newSize;               // 更新 root1 树的大小
    } else {                                // 如果 root2 的树比 root1 的树小
        arr[root1] = root2;                 // 将 root1 的根指向 root2
        arr[root2] = newSize;               // 更新 root2 树的大小
    }
}
```

1. `newSize = arr[root1] + arr[root2]`：
   - 计算合并后的新树的大小。`arr[root1]` 和 `arr[root2]` 是根节点的值，表示这两棵树的大小（树的大小以负值表示，负值的绝对值是树的大小）。

2. **按大小合并**:
   - 如果 `root1` 的树比 `root2` 小（即 `arr[root1] < arr[root2]`），那么将 `root2` 合并到 `root1` 下，并更新 `root1` 树的大小。
   - 否则，将 `root1` 合并到 `root2` 下，并更新 `root2` 树的大小。
   - 在 `arr` 数组中，根节点的值表示树的大小（负值）。

代码执行图示:

- **合并前的树结构**:
  - 左侧的树表示元素 `0, 4, 8, 1` 属于一个集合，根节点是 `4`，树的大小是 4。
  - 右侧的树表示元素 `3, 6` 属于另一个集合，根节点是 `3`，树的大小是 2。
- **数组表示**:

```
Index:   0   1   2   3   4   5   6   7   8   9
Value:  -2  8  -3  -4   4  3  4  4  -1
```

  - `arr[0] = -2` 表示元素 `0` 是根节点，树的大小为 2。
  - `arr[4] = -4` 表示元素 `4` 是根节点，树的大小为 4。
  - `arr[3] = -4` 表示元素 `3` 是根节点，树的大小为 4。
- **Union(4, 3)**:
  - `Union(4, 3)` 操作会将两棵树合并。
  - 在这个例子中，`root1` 是 `4`，`root2` 是 `3`，因为树 `4` 的大小（4）大于树 `3` 的大小（2），所以将树 `3` 合并到树 `4` 下。
  - 合并后的数组会变为:

```
Index:   0   1   2   3   4   5   6   7   8   9
Value:  -4  8  -3  -4  -8  3  4  4  -1
```

总结:

- **Union by Size** 优化了树的结构，通过总是将小树合并到大树下，从而有效限制树的高度，避免形成链式结构。
- 通过**按大小合并**，每次合并时都会选择较小的树作为子树，从而使树的高度保持在 **O(log n)** 以内，优化了后续的查找操作（`Find`）。