极客时间算法训练营
第十四课
字典树、并查集

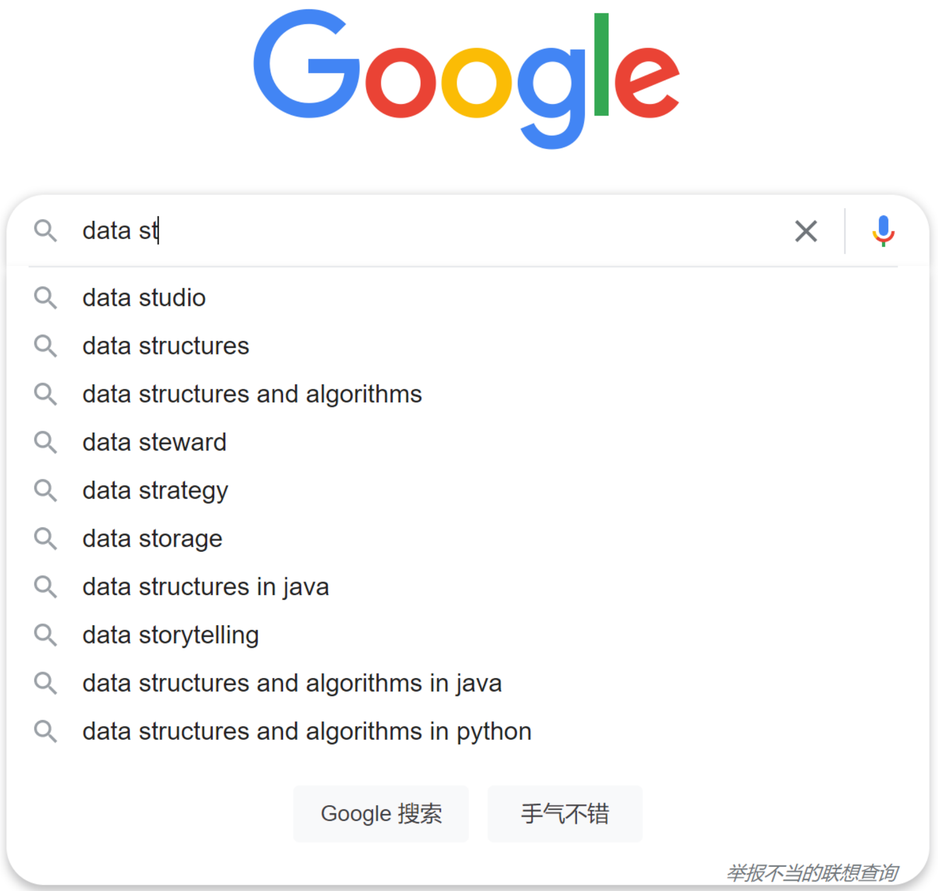
李煜东

《算法竞赛进阶指南》作者

目录

字典树（Trie）

经典面试题：搜索提示（自动补全）

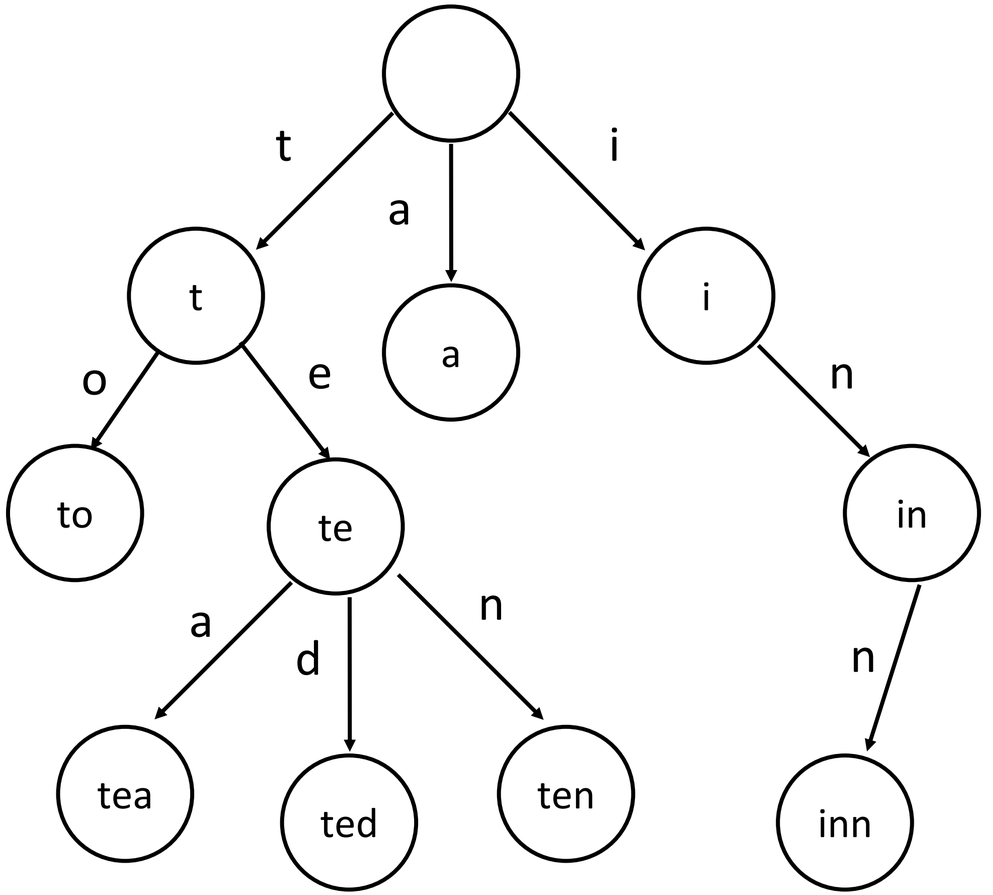


基本结构

字典树（Trie 树）是一种由“结点”和“带有字符的边”构成的树形结构。

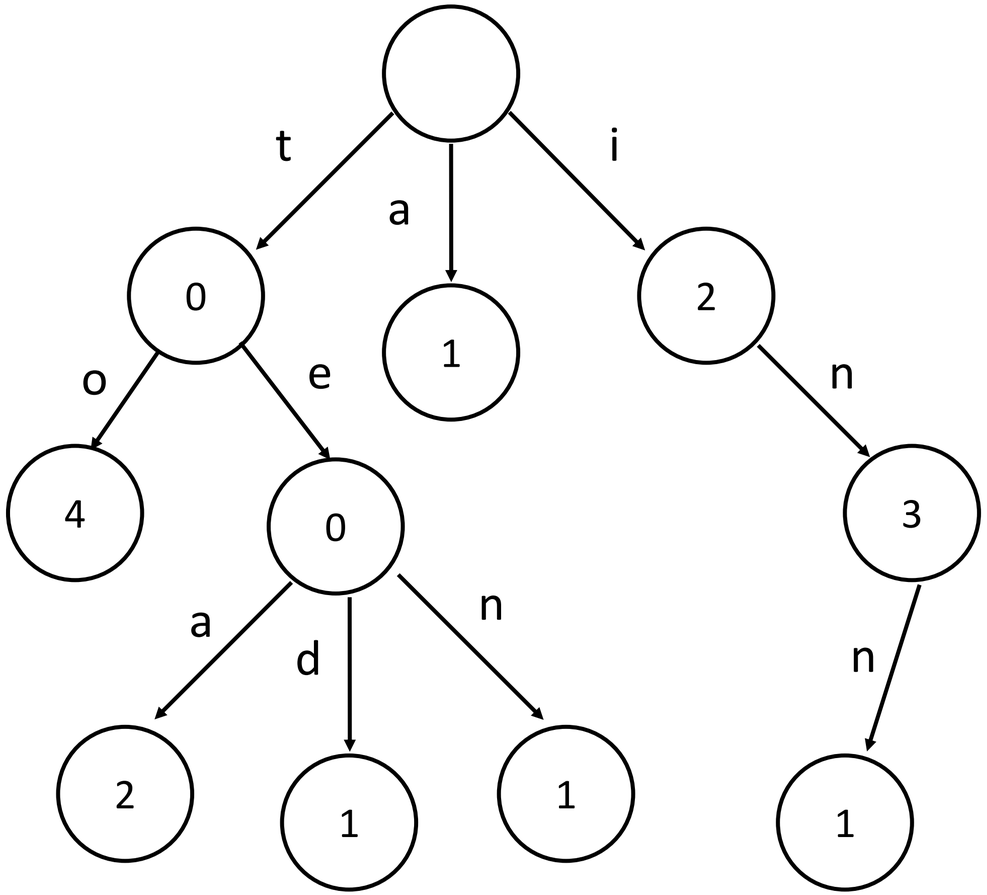
典型应用是用于统计和排序大量的字符串（但不仅限于字符串），经常被搜索引擎系统用于文本词频统计。

它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高。



基本性质

1. 结点本身不保存完整单词。
2. 从根结点到某一结点，路径上经过的字符连接起来，为该结点对应的单词。
3. 每个结点出发的所有边代表的字符都不相同。
4. 结点用于存储单词的额外信息（例如频次）。



右边的字典树存储了8个单词及其出现频次

to: 4次，tea：2次，ted：1次，ten：1次，
a：1次，i：2次，in：3次，inn：1次

内部实现

字符集数组法（简单）

每个结点保存一个长度固定为字符集大小（例如26）的数组，以字符为下标，保存指向的结点

空间复杂度为 O(结点数 * 字符集大小)，查询的时间复杂度为 O(单词长度)

适用于较小字符集，或者单词短、分布稠密的字典

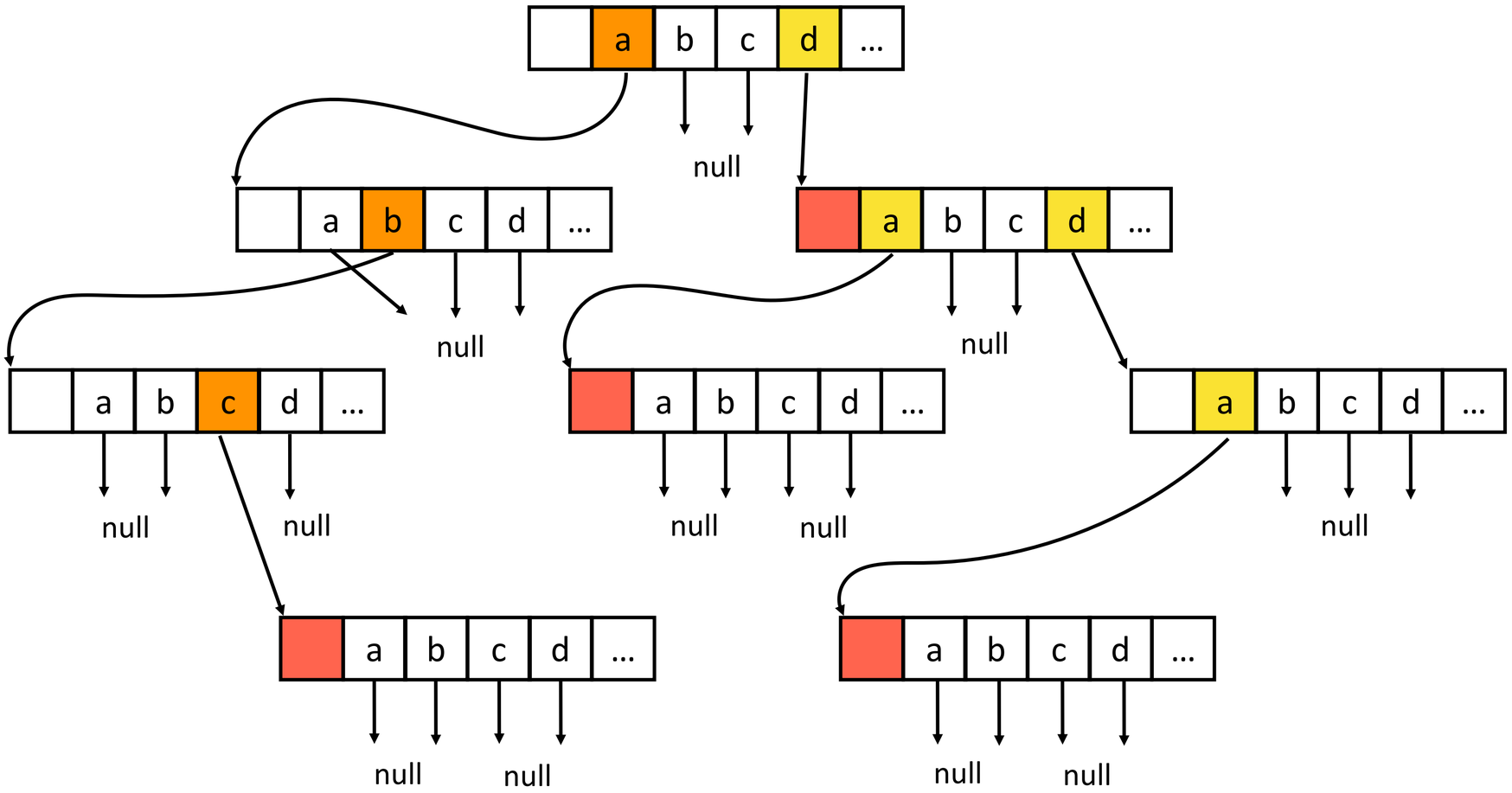
字符集映射法（优化）

把每个结点上的字符集数组改为一个映射（词频统计：hash map，排序：ordered map）

空间复杂度为 O(文本字符总数)，查询的时间复杂度为 O(单词长度)，但常数稍大一些

适用性更广

内部实现


a b c d ...
null
a b c d ...
null
a b c d ...
null
a b c d ...
null
null
a b c d ...
null
null
a b c d ...
null
null
a b c d ...
null
null

核心思想

Trie 树的核心思想是空间换时间

无论是保存树的结构、字符集数组还是字符集映射，都需要额外的空间

利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的

分组思想——前缀相同的字符串在同一子树中

实战

实现 Trie

https://leetcode-cn.com/problems/implement-trie-prefix-tree/

单词搜索 II

https://leetcode-cn.com/problems/word-search-ii/

思考题

系统设计：搜索提示（自动补全）

C++ Code

```cpp
class Trie {
public:
    Trie() { root = new Node(); }
    void insert(string word) { find(word, true, true); }
    bool search(string word) { return find(word, true, false); }
    bool startsWith(string prefix) { return find(prefix, false, false); }

private:
    struct Node {
        int count;
        unordered_map<char, Node*> child;
        Node(): count(0) {}
    };
    Node* root;

    bool find(const string& s, bool exact_match, bool insert_if_not_exist) {
        Node* curr = root;
        for (char c : s) {
            if (curr->child.find(c) == curr->child.end()) {
                if (!insert_if_not_exist) return false;
                curr->child[c] = new Node();
            }
            curr = curr->child[c];
        }
        if (insert_if_not_exist) curr->count++;
        return exact_match ? curr->count > 0 : true;
    }
};
```

Java Code

```java
class Trie {
    public Trie() { root = new Node(); }
    public void insert(String word) { find(word, true, true); }
    public boolean search(String word) { return find(word, true, false); }
    public boolean startsWith(String prefix) { return find(prefix, false, false); }

    class Node {
        public int count;
        public HashMap<Character, Node> child;
        public Node() { count = 0; child = new HashMap<>(); }
    }
    Node root;

    boolean find(String s, boolean exact_match, boolean insert_if_not_exist) {
        Node curr = root;
        for (Character c : s.toCharArray()) {
            if (!curr.child.containsKey(c)) {
                if (!insert_if_not_exist) return false;
                curr.child.put(c, new Node());
            }
            curr = curr.child.get(c);
        }
        if (insert_if_not_exist) curr.count++;
        return exact_match ? curr.count > 0 : true;
    }
}
```

Python Code

```python
class Trie:
    def __init__(self):
        self.root = [0, {}]  # [count, child]

    def insert(self, word: str) -> None:
        self.find(word, True, True)

    def search(self, word: str) -> bool:
        return self.find(word, True, False)

    def startsWith(self, prefix: str) -> bool:
        return self.find(prefix, False, False)

    def find(self, s, exact_match, insert_if_not_exist):
        curr = self.root
        for ch in s:
            if ch not in curr[1]:
                if not insert_if_not_exist:
                    return False
                curr[1][ch] = [0, {}]
            curr = curr[1][ch]
        if insert_if_not_exist:
            curr[0] += 1
        return curr[0] > 0 if exact_match else True
```

并查集（Disjoint Set）

基本用途

处理不相交集合（disjoint sets）的合并和查询问题

处理分组问题

维护无序二元关系

基本操作

MakeSet(s)：

建立一个新的并查集，其中包含 s 个集合，每个集合里只有一个元素。

UnionSet(x, y)：

把元素 x 和元素 y 所在的集合合并。

要求 x 和 y 所在的集合不相交，如果相交则无需合并。

Find(x)：

找到元素 x 所在的集合的代表。

该操作也可以用于判断两个元素是否位于同一个集合，只要将它们各自的代表比较一下就可以了。

内部实现

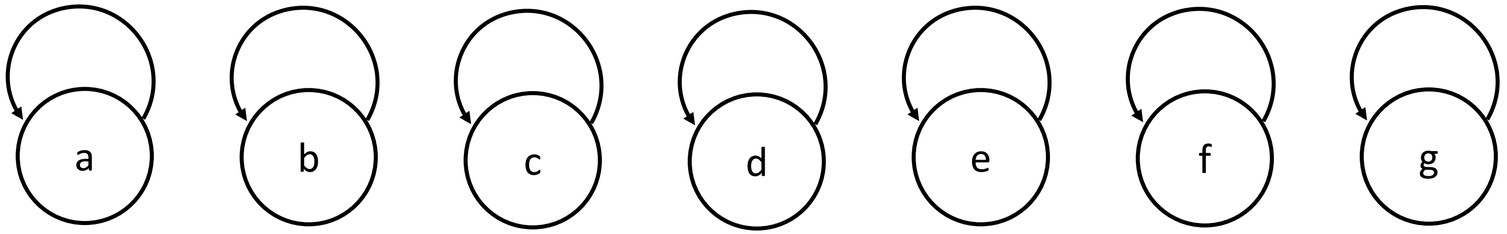
每个集合是一个树形结构

每个结点只需要保存一个值：它的父结点

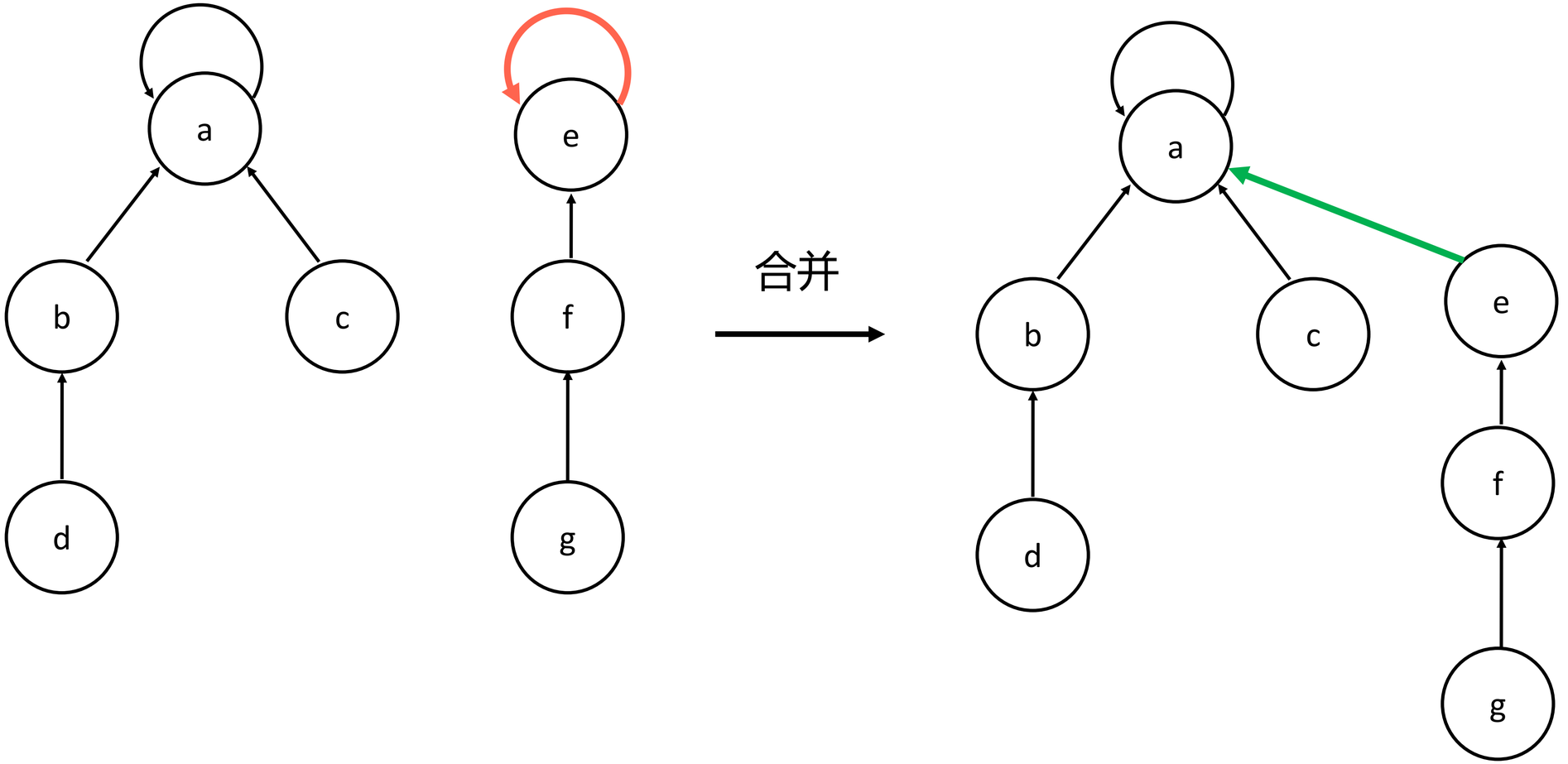
最简单的实现是只用一个 int 数组 fa，fa[x] 表示编号为 x 的结点的父结点

根结点的 fa 等于它自己

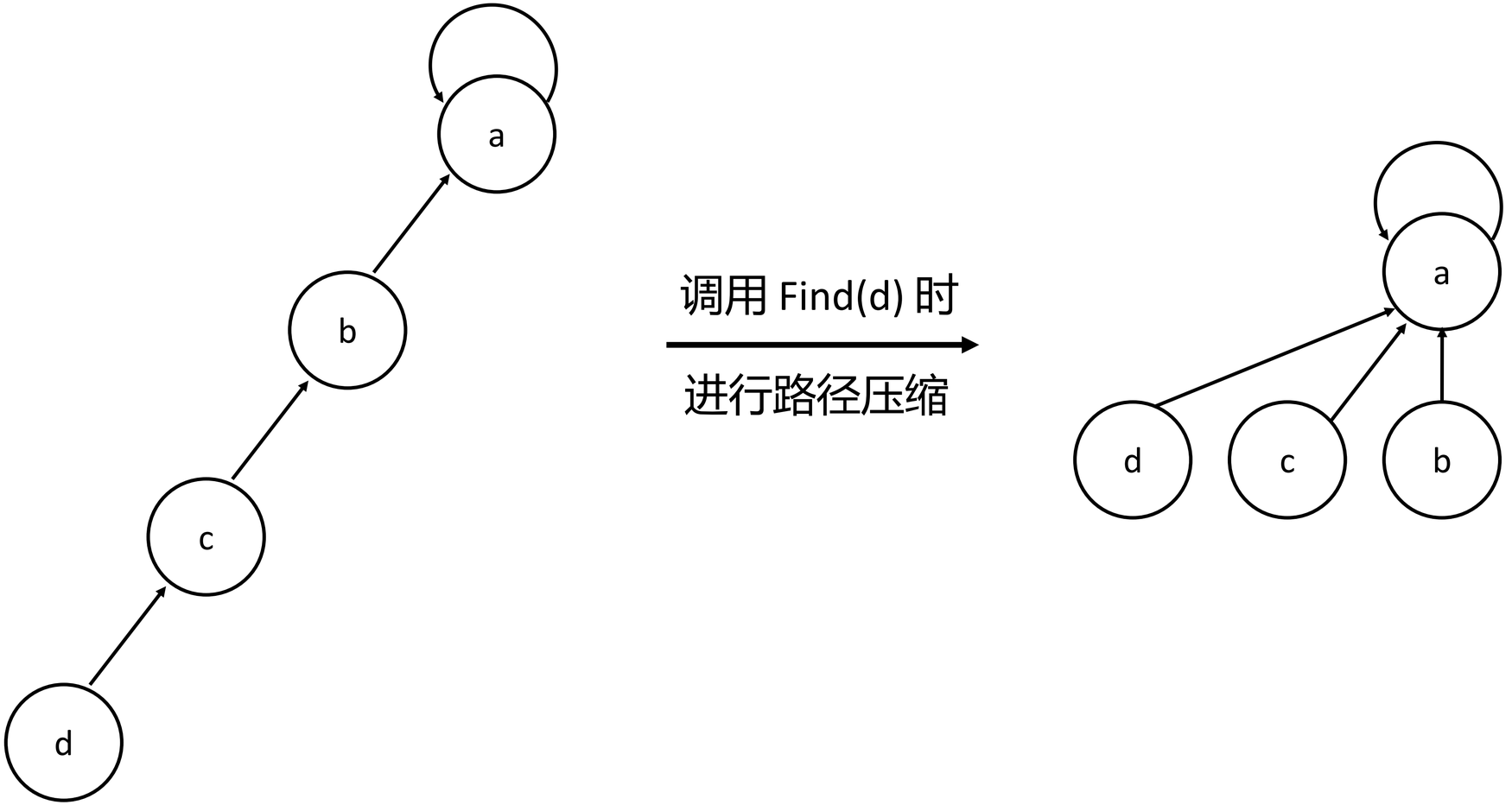
初始化 MakeSet



合并 UnionSet



查询 Find + 路径压缩



路径压缩

并查集本质上只关心每个结点所在的集合，不关心该集合对应的树形结构具体是怎样的

而一个结点所在的集合由根结点确定

因此在 Find(x) 的同时把 x 和 x 的所有祖先直接连到根结点上，下一次就可以一步走到根了

并查集还有一个优化叫做按秩合并（合并时把较深的树合并到较浅的上面）或者启发式合并（合并时把较大的树合并到较小的树上面）

同时采用路径压缩 + 按秩合并优化的并查集，单次操作的均摊复杂度为 $O(\alpha(n))$

只采用其中一种，$O(\log(n))$

$\alpha(n)$ 是反阿克曼函数，是一个比 $\log(n)$ 增长还要缓慢许多的函数，一般 $\alpha(n) \leq 5$，近似常数

通常实现中为了简便，我们只使用路径压缩

C++ Code

```cpp
class DisjointSet {
public:
    DisjointSet(int n) {
        fa = vector<int>(n, 0);
        for (int i = 0; i < n; i++) fa[i] = i;
    }

    int find(int x) {
        if (x == fa[x]) return x;
        return fa[x] = find(fa[x]);
    }

    void unionSet(int x, int y) {
        x = find(x), y = find(y);
        if (x != y) fa[x] = y;
    }

private:
    vector<int> fa;
};
```

Java Code

```
class DisjointSet {
    public DisjointSet(int n) {
        fa = new int[n];
        for (int i = 0; i < n; i++) fa[i] = i;
    }

    public int find(int x) {
        if (x == fa[x]) return x;
        return fa[x] = find(fa[x]);
    }

    public void unionSet(int x, int y) {
        x = find(x);
        y = find(y);
        if (x != y) fa[x] = y;
    }

    int[] fa;
};
```

Python Code

```python
class DisjointSet:
    def __init__(self, n):
        self.fa = [i for i in range(n)]

    def find(self, x):
        if x == self.fa[x]:
            return x
        self.fa[x] = self.find(self.fa[x])
        return self.fa[x]

    def unionSet(self, x, y):
        x = self.find(x)
        y = self.find(y)
        if x != y:
            self.fa[x] = y
```

实战

省份数量

https://leetcode-cn.com/problems/number-of-provinces/

被围绕的区域

https://leetcode-cn.com/problems/surrounded-regions/

Homework

冗余连接

https://leetcode-cn.com/problems/redundant-connection/

岛屿数量

https://leetcode-cn.com/problems/number-of-islands/

要求：使用并查集而非DFS/BFS实现

THANKS