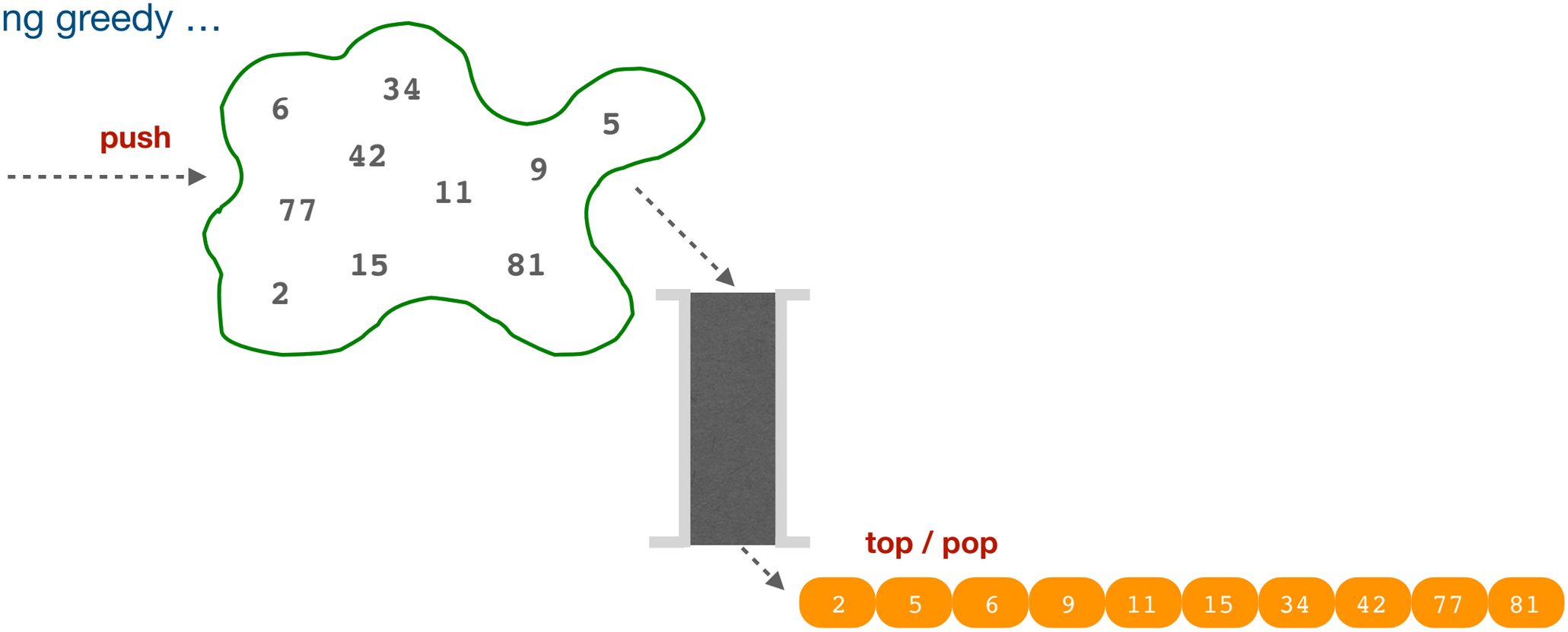
数据结构实验（6）

树的扩展（1）

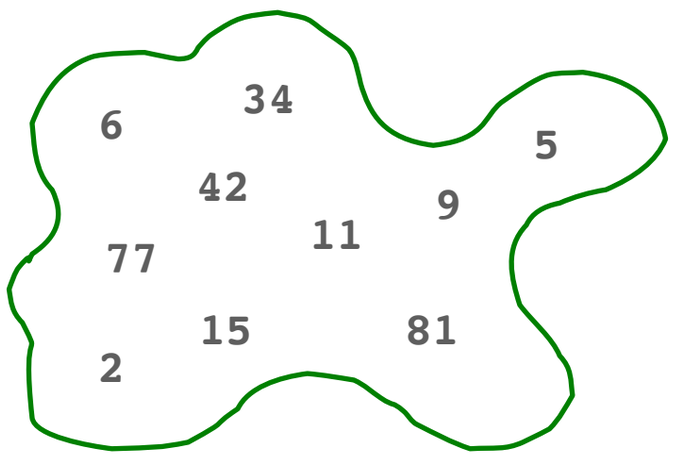
目录

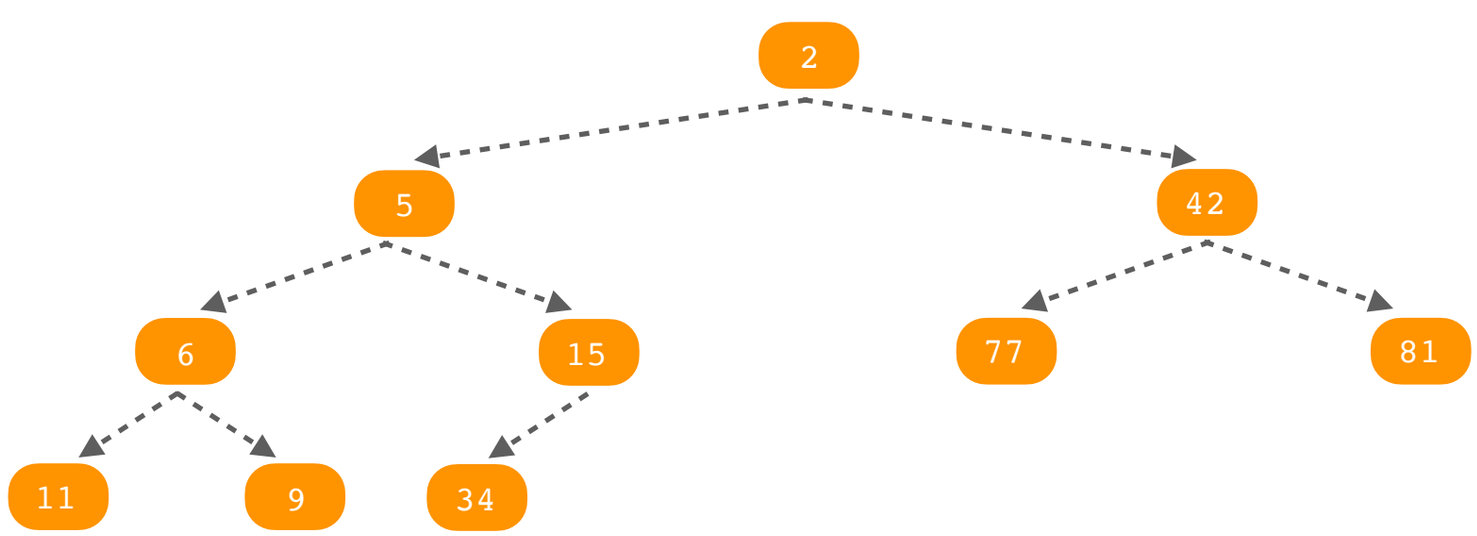
- 优先队列
  - 优先队列（Priority Queue）是一种抽象数据类型（ADT）。
    - 可以向优先队列中插入元素（push）；每个元素都有优先级，而优先级高（或者低）的将会先出队（pop）；同时也能获取优先队列中优先级最高（或者最低）的元素（top）
  - 优先队列的应用
    - 排序
    - Dijkstra 最短路径算法
    - Prim 最小生成树算法
    - 哈夫曼树
    - Anything greedy …

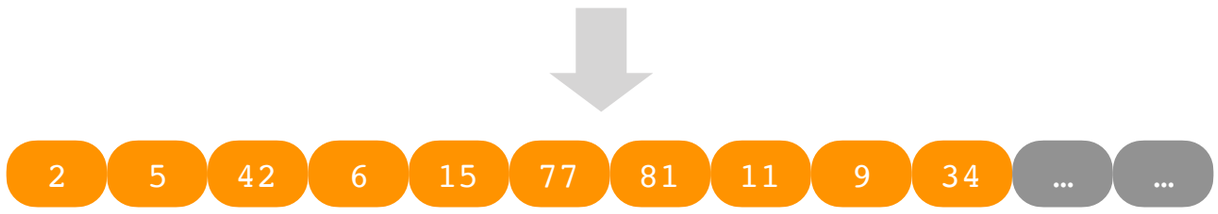


https://zh.wikipedia.org/wiki/二叉堆

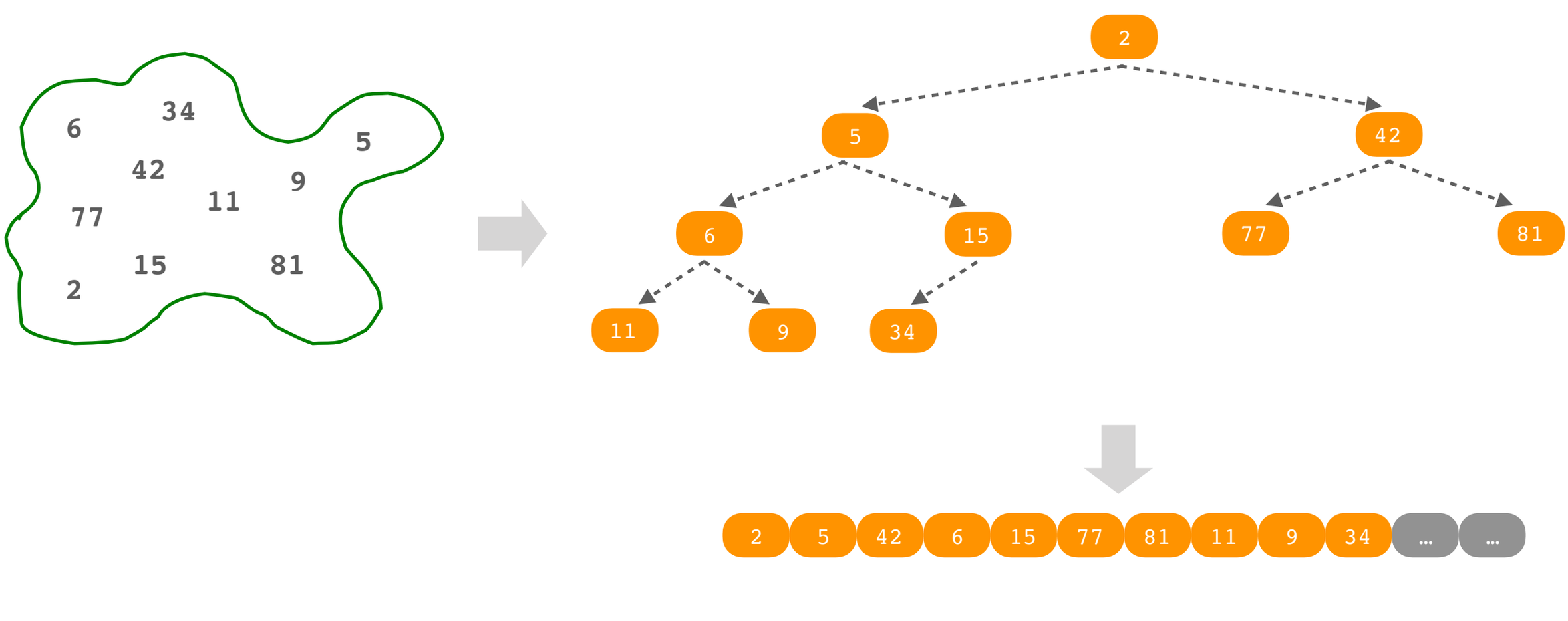
- 优先队列
  - 二叉堆（Binary Heap）
    - 二叉堆是一棵满二叉树
    - 二叉堆中每一个节点（的优先级）均比它的儿子小（或大）
    - 用顺序表实现







- 优先队列
  - 二叉堆（Binary Heap）
    - 二叉堆是一棵满二叉树
      - N 个节点的二叉堆，树的深度是 $\log_2 N$
    - 二叉堆中每一个节点（的优先级）均比它的儿子小（或大）
      - 二叉树的根节点就是 最小（或最大）节点 —— O(1) 获取 Top
    - 用顺序表实现
      - h[i] < h[i*2] && h[i] < h[i*2+1]

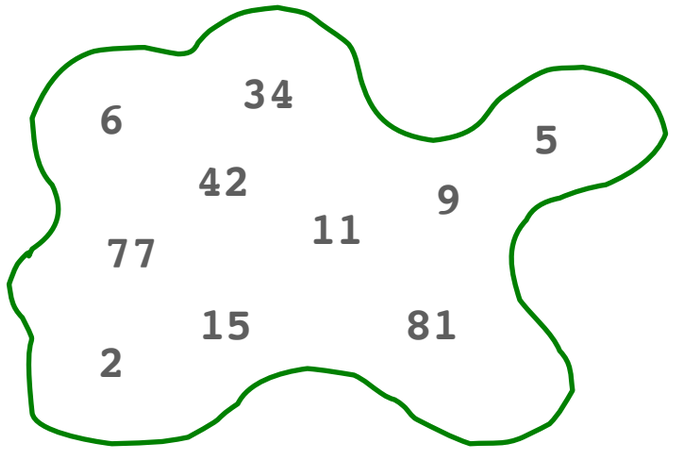


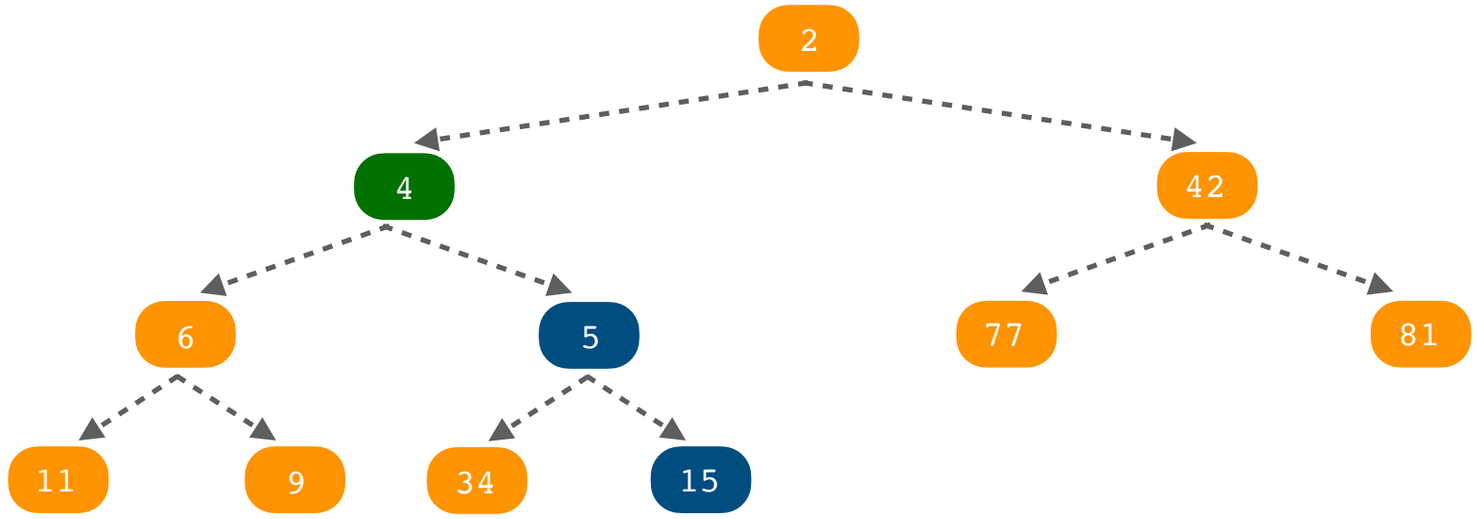
- 优先队列
  - 二叉堆（Binary Heap）
    - Push 操作
      - 将新元素作为满二叉树下一个元素，写入顺序表末尾
      - 如果新元素比它的父节点小（或大），则将它与父亲交换

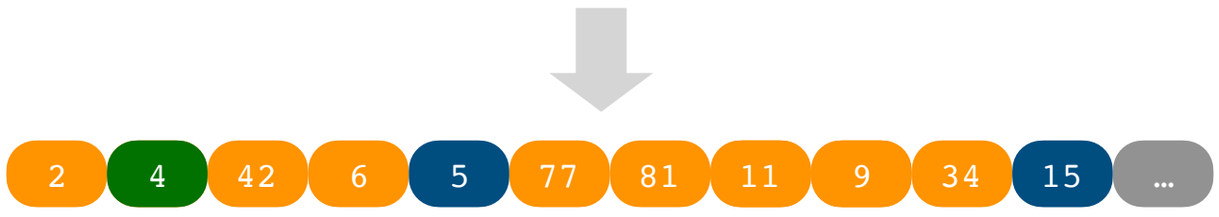
- 优先队列
  - 二叉堆（Binary Heap）
    - Push 操作
      - 将新元素作为满二叉树下一个元素，写入顺序表末尾
      - 如果新元素比它的父节点小（或大），则将它与父亲交换

- 优先队列
  - 二叉堆（Binary Heap）
    - Push 操作
      - 将新元素作为满二叉树下一个元素，写入顺序表末尾
      - 如果新元素比它的父节点小（或大），则将它与父亲交换

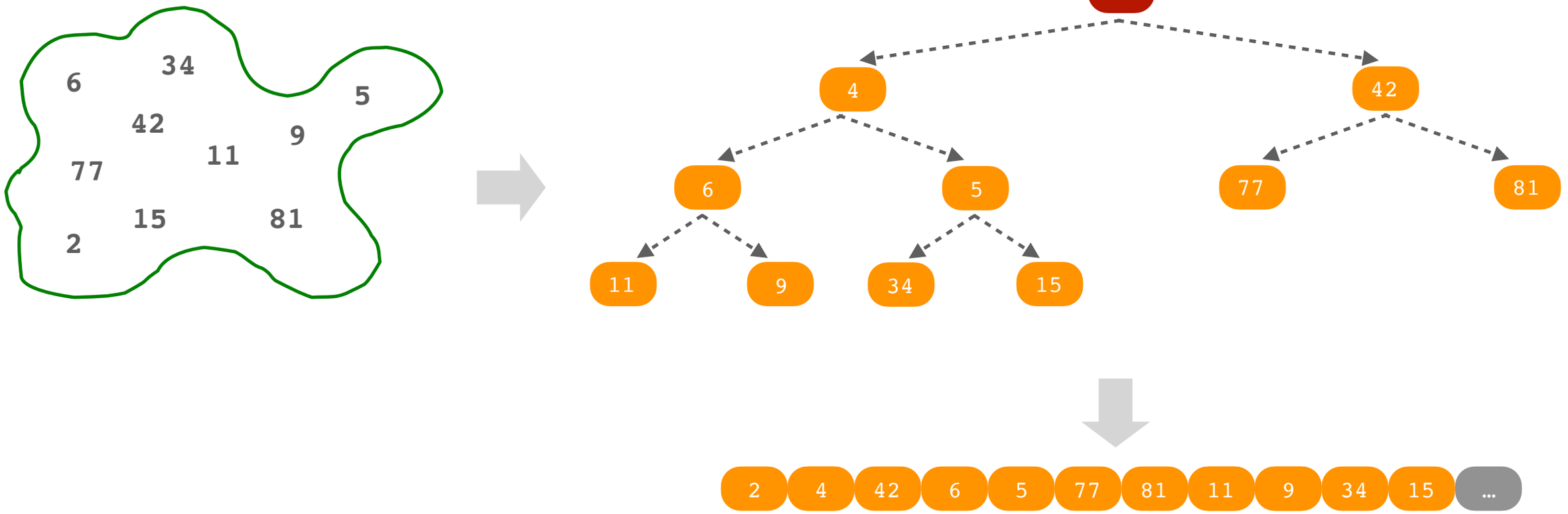
- 优先队列
  - 二叉堆（Binary Heap）
    - Push 操作
      - 将新元素作为满二叉树下一个元素，写入顺序表末尾
      - 如果新元素比它的父节点小（或大），则将它与父亲交换
      - 时间复杂度 O(logN)



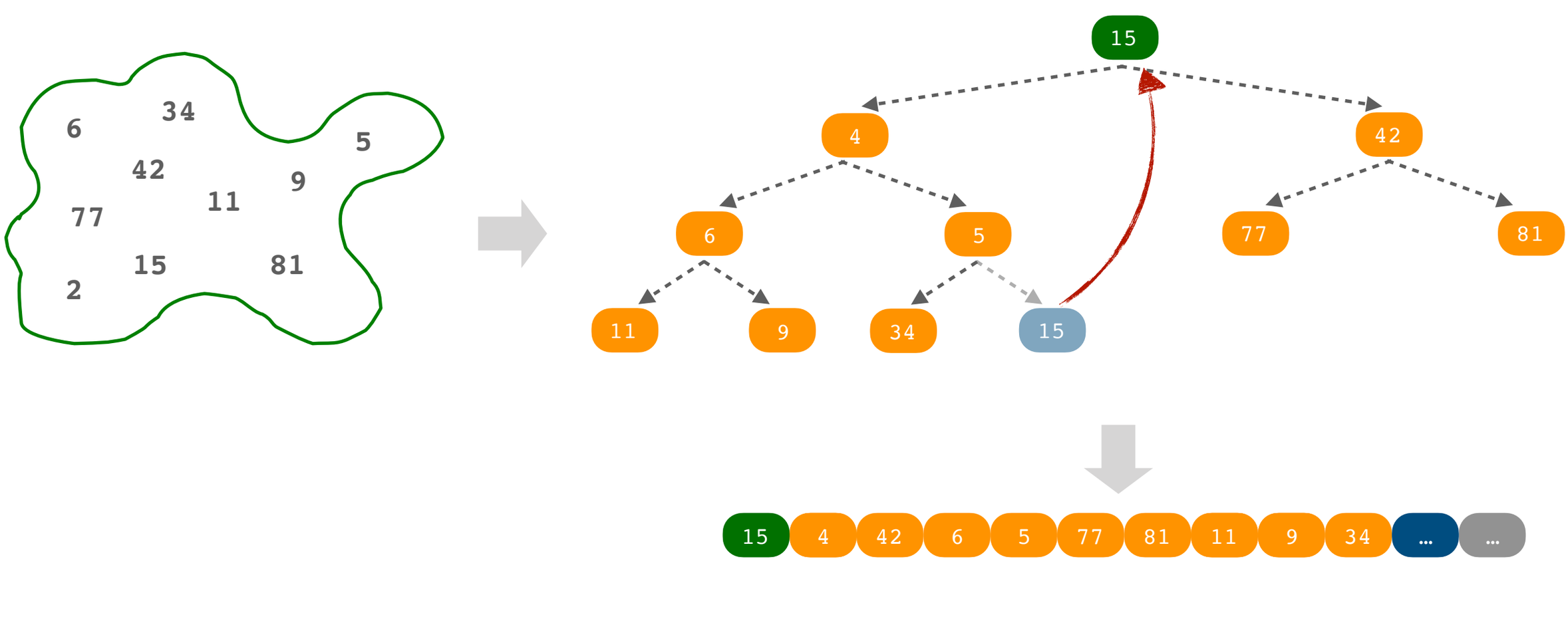




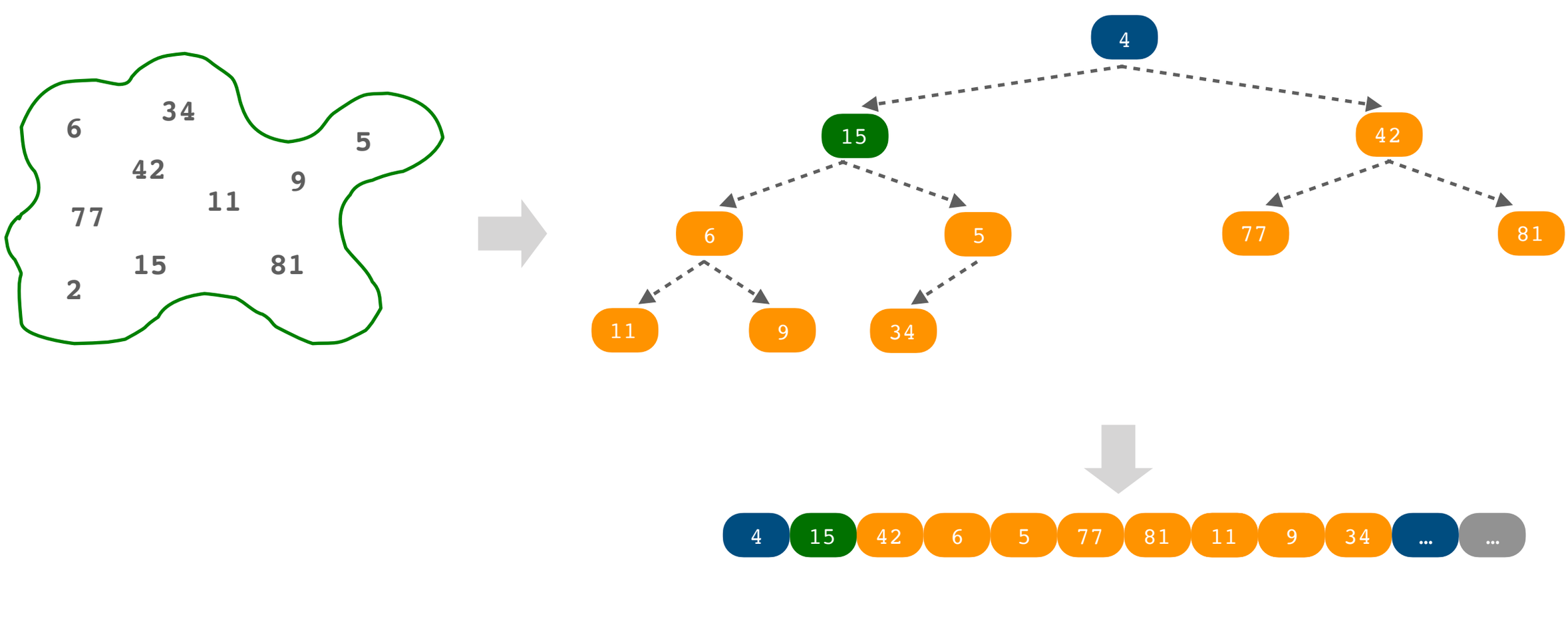
- 优先队列
  - 二叉堆（Binary Heap）
    - Pop 操作
      - 根（Root）元素出队，将满二叉树中 “最后” 一个元素移动到根（Root）
      - 如果新的根节点比它的任意一个儿子节点小（或大），则将它与该儿子交换



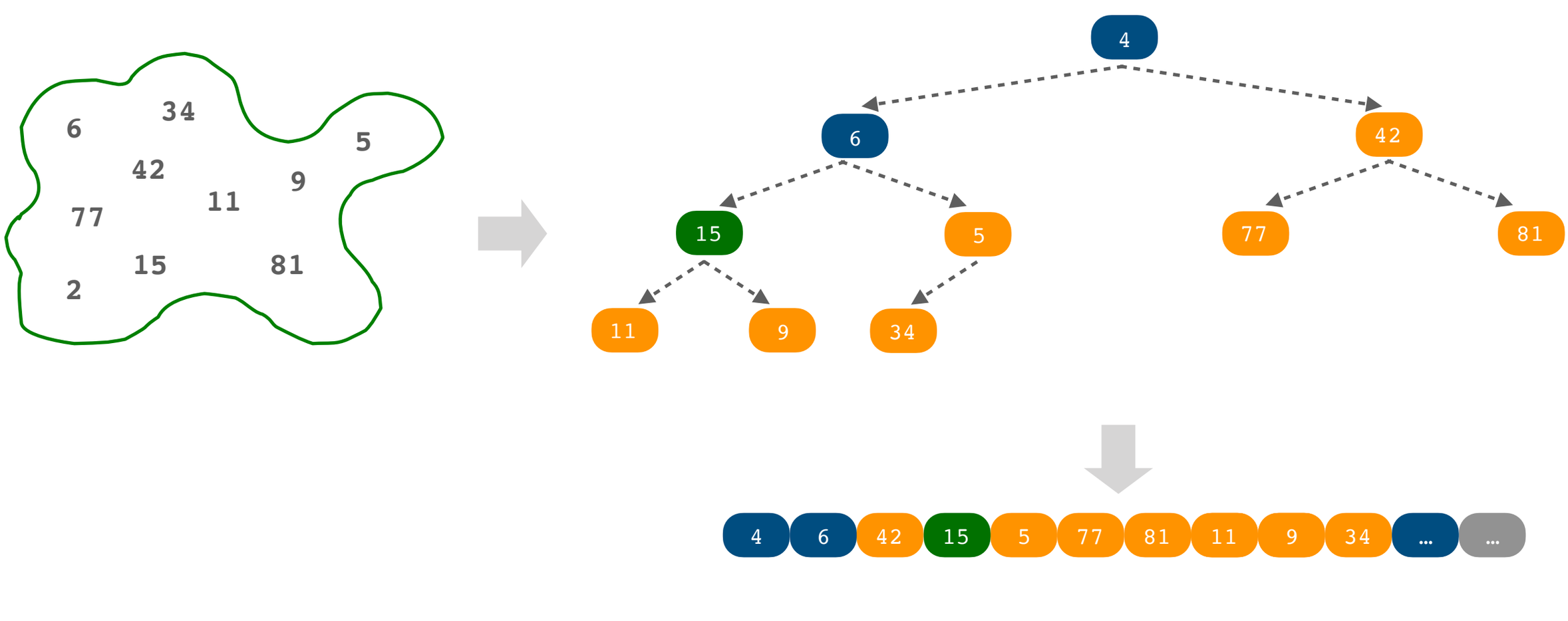
- 优先队列
  - 二叉堆（Binary Heap）
    - Pop 操作
      - 根（Root）元素出队，将满二叉树中 “最后” 一个元素移动到根（Root）
      - 如果新的根节点比它的任意一个儿子节点小（或大），则将它与该儿子交换



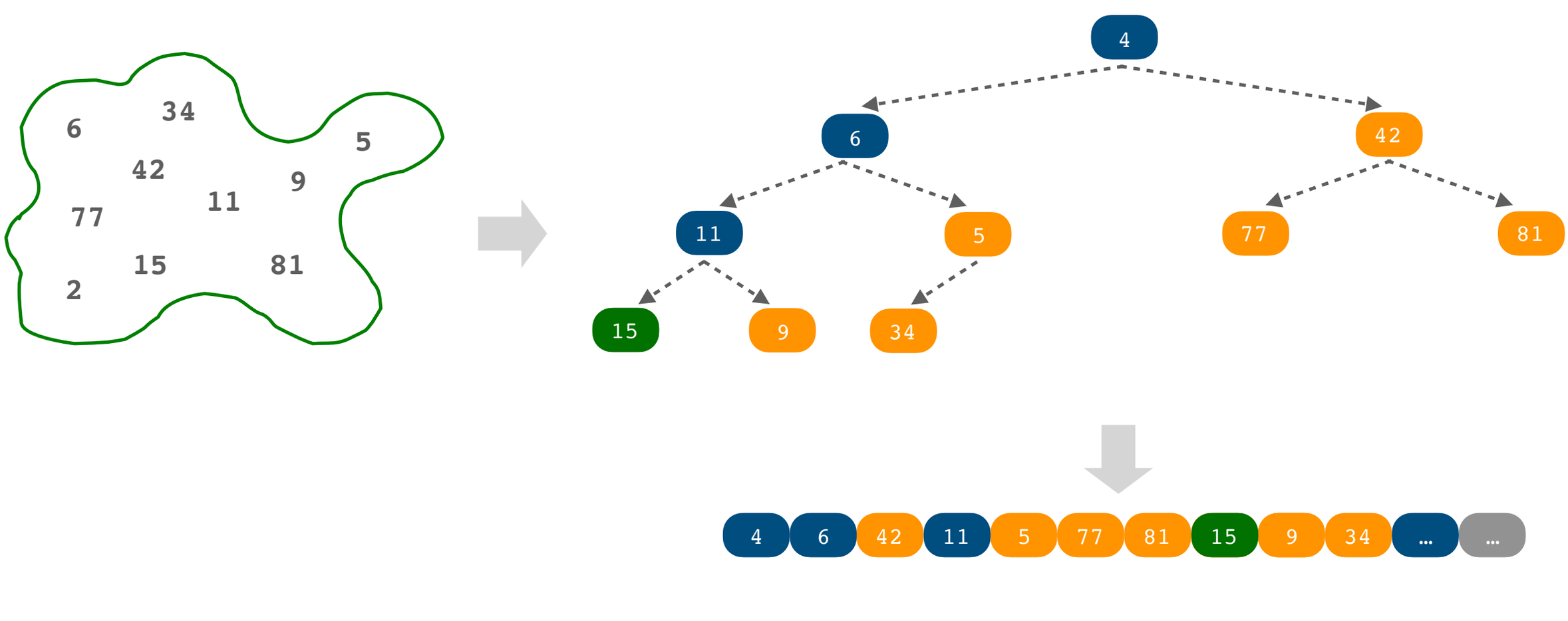
- 优先队列
  - 二叉堆（Binary Heap）
    - Pop 操作
      - 根（Root）元素出队，将满二叉树中 “最后” 一个元素移动到根（Root）
      - 如果新的根节点比它的任意一个儿子节点小（或大），则将它与该儿子交换



- 优先队列
  - 二叉堆（Binary Heap）
    - Pop 操作
      - 根（Root）元素出队，将满二叉树中 “最后” 一个元素移动到根（Root）
      - 如果新的根节点比它的任意一个儿子节点小（或大），则将它与该儿子交换



- 优先队列
  - 二叉堆（Binary Heap）
    - Pop 操作
      - 根（Root）元素出队，将满二叉树中 “最后” 一个元素移动到根（Root）
      - 如果新的根节点比它的任意一个儿子节点小（或大），则将它与该儿子交换

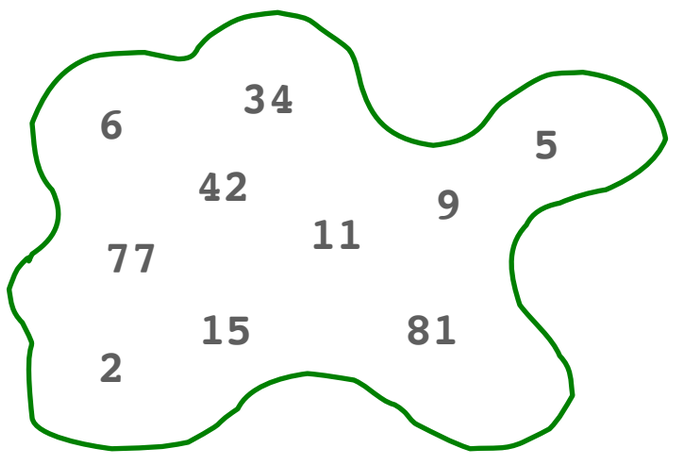


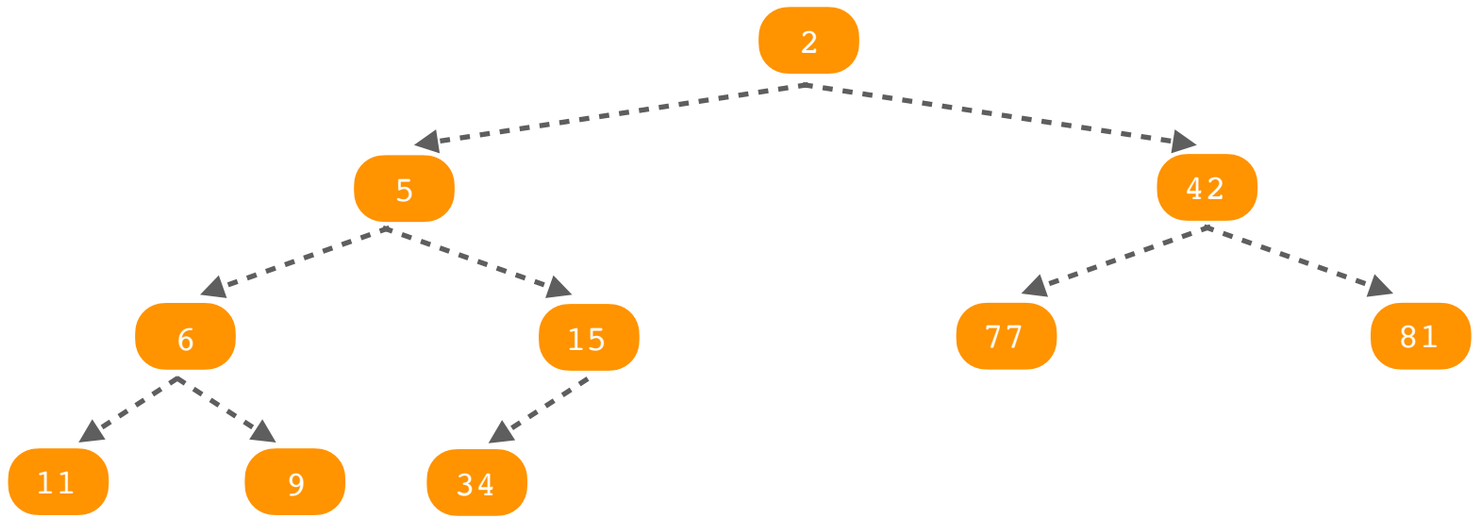
- 优先队列
  - 二叉堆（Binary Heap）

```cpp
#include<queue>

int main(int, char**) {
    // 最大堆
    std::priority_queue<int> q;
    // 最小堆
    std::priority_queue<int, std::vector<int>, std::greater<int> >q;

    q.top();
    q.pop();
    q.push(123);
    q.empty();
    q.size();
    return 0;
}
```

目录

- 优先队列
  - 应用
    - 堆排序



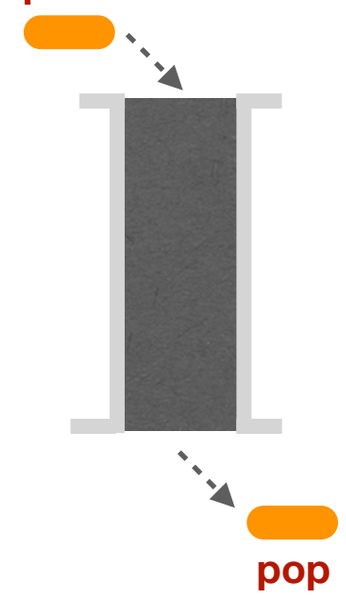


时间复杂度：O(NlogN)

- 优先队列
  - 应用
    - TopK 问题

LeetCode 703. <Kth Largest Element in a Stream>
https://leetcode.com/problems/kth-largest-element-in-a-stream/

算法：
1）定义优先队列，以元素大小作为优先级，值较小的元素优先级较高：
  1）push(k) – 插入元素
  2）pop() – 删除队列中最小的元素
  3）size() – 返回队列中元素数目
  4）top() – 返回队列中最小的元素
2）遍历整数序列
  1）将序列元素 push() 进优先队列
  2）若队列中的元素数目 size() <= k，则重复 1)；否则，pop() 删除优先级最高的元素（值最小的元素）
3）返回队列中优先级最高的元素（值最小的元素）



pop() 返回优先队列中最小的元素

扩展：中位数问题?

- 优先队列
  - 应用
    - 定时器

Redis Setex 命令

Redis 字符串(string)

Redis Setex 命令为指定的 key 设置值及其过期时间。如果 key 已经存在， SETEX 命令将会替换旧的值。

语法

redis Setex 命令基本语法如下：

```
redis 127.0.0.1:6379> SETEX KEY_NAME TIMEOUT VALUE
```

可用版本

>= 2.0.0

返回值

设置成功时返回 OK 。

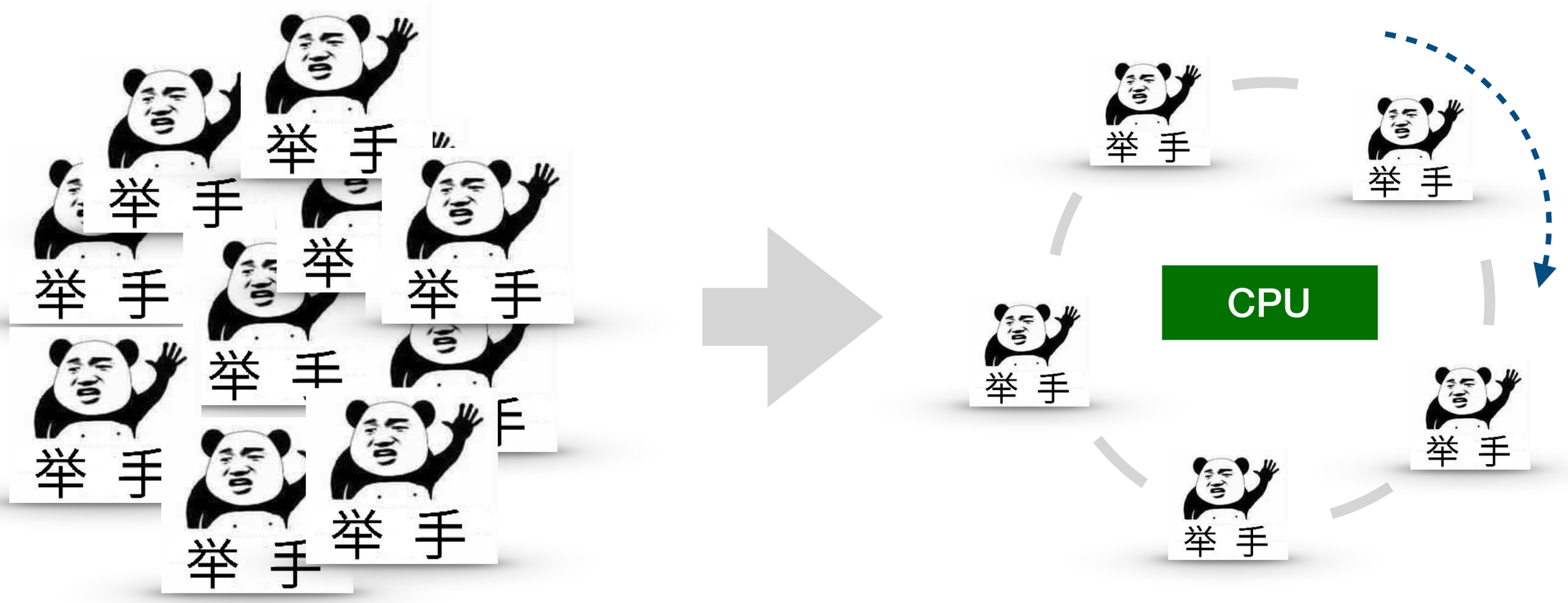
实例

```
redis 127.0.0.1:6379> SETEX mykey 60 redis
OK
redis 127.0.0.1:6379> TTL mykey
60
redis 127.0.0.1:6379> GET mykey
"redis
```

- 优先队列
  - 应用
    - 调度问题

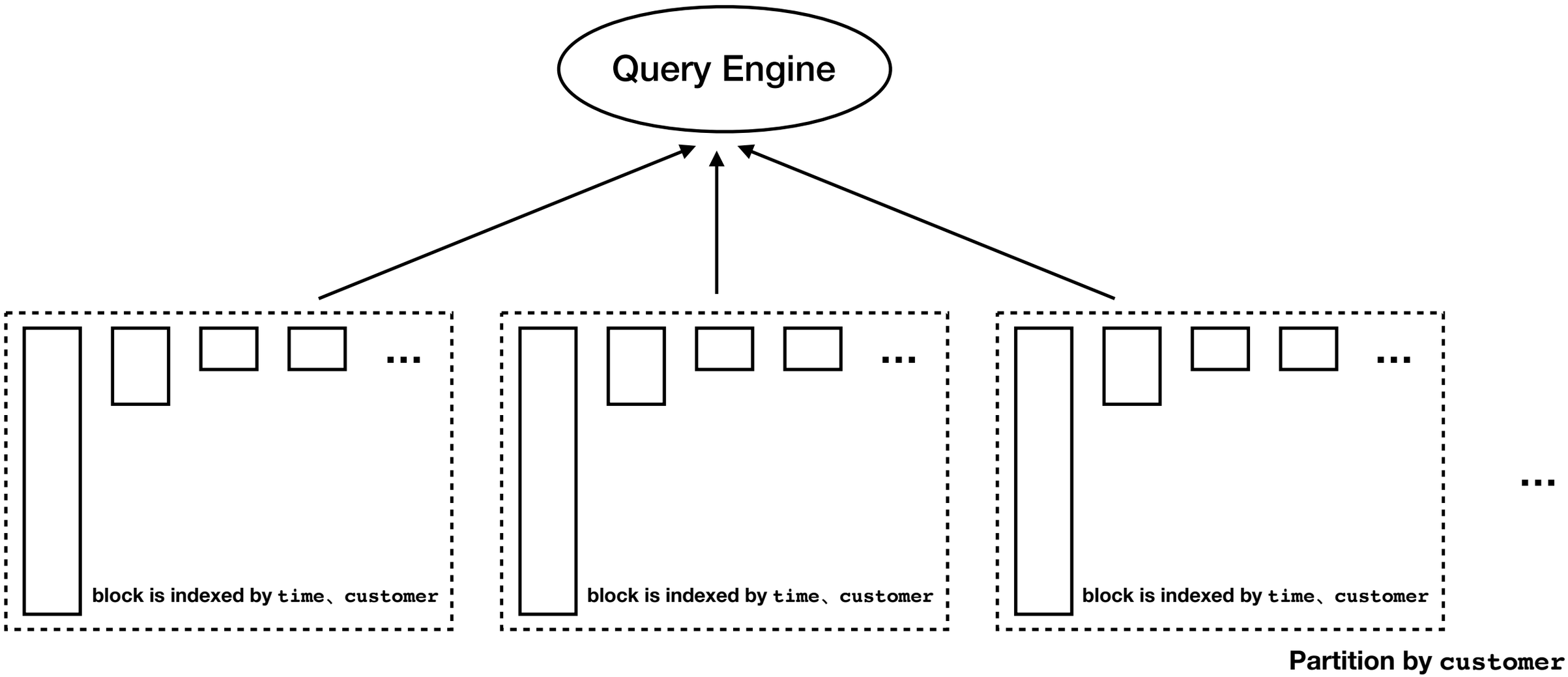
```
// 轮询调度算法(Round-Robin Scheduling)
// https://en.wikipedia.org/wiki/Round-robin_scheduling
RoundRobin {
    // 初始化队列，并将共享资源的所有使用者插入队列中
    Queue q(clients);
    // 反复执行任务，直到在服务终止
    while !serviceTerminated() {
        client = q.pop();     // 删除并获取队列首的使用者
        serve(client.task);   // 执行任务
        q.push(client);       // 将使用者放入队列末尾，等待下一次轮训
    }
}
```

- 多线程程序在单一 CPU 上的执行；多个 IO 请求在磁盘执行；多个 Web 服务程序相应网络请求
- 如果每个请求都有优先级，需按优先级排序执行?



- 优先队列
  - 应用
    - 多路归并排序

```
SELECT SUM(clicks)
FROM tblClick
WHERE
   time >= ‘xxx’ AND
   time <= ‘xxx’ AND
   customer in (‘xxx’, ‘yyy’, ‘zzz’)
```



- 优先队列
  - 应用
    - 工程中的复杂排序问题

      10 亿条搜索关键词日志记录中找出 top 10 最热门的关键词
      假设平均每个搜索关键词占 50 字节

目录

- 优先队列
  - 堆的优化

	Binary Heap	Binomial Heap	Fibonacci Heap
make_heap	O(1)	O(1)	O(1)
push	O(logN)	O(logN)	O(1)
top	O(1)	O(logN)	O(1)
pop	O(logN)	O(logN)	O(logN)
union	O(N)	O(logN)	O(1)

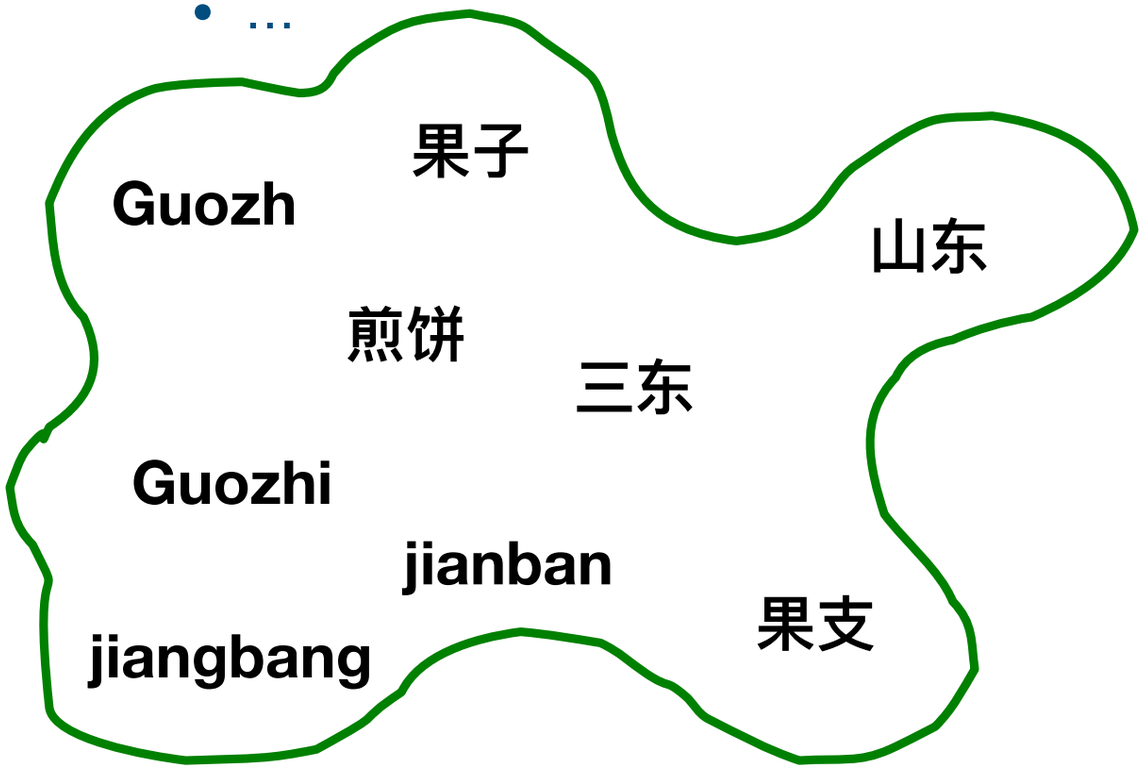
Binomial Heap: https://zh.wikipedia.org/zh-hans/二项堆

Fibonacci Heap: https://zh.wikipedia.org/zh-hans/斐波那契堆

目录

https://zh.wikipedia.org/wiki/并查集

- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）
  - 并查集的应用
    - 求连通子图
    - Kruskal 最小生成树算法
    - 最近公共祖先（Least Common Ancestors）
    - 搜索引擎关键词改写
    - …

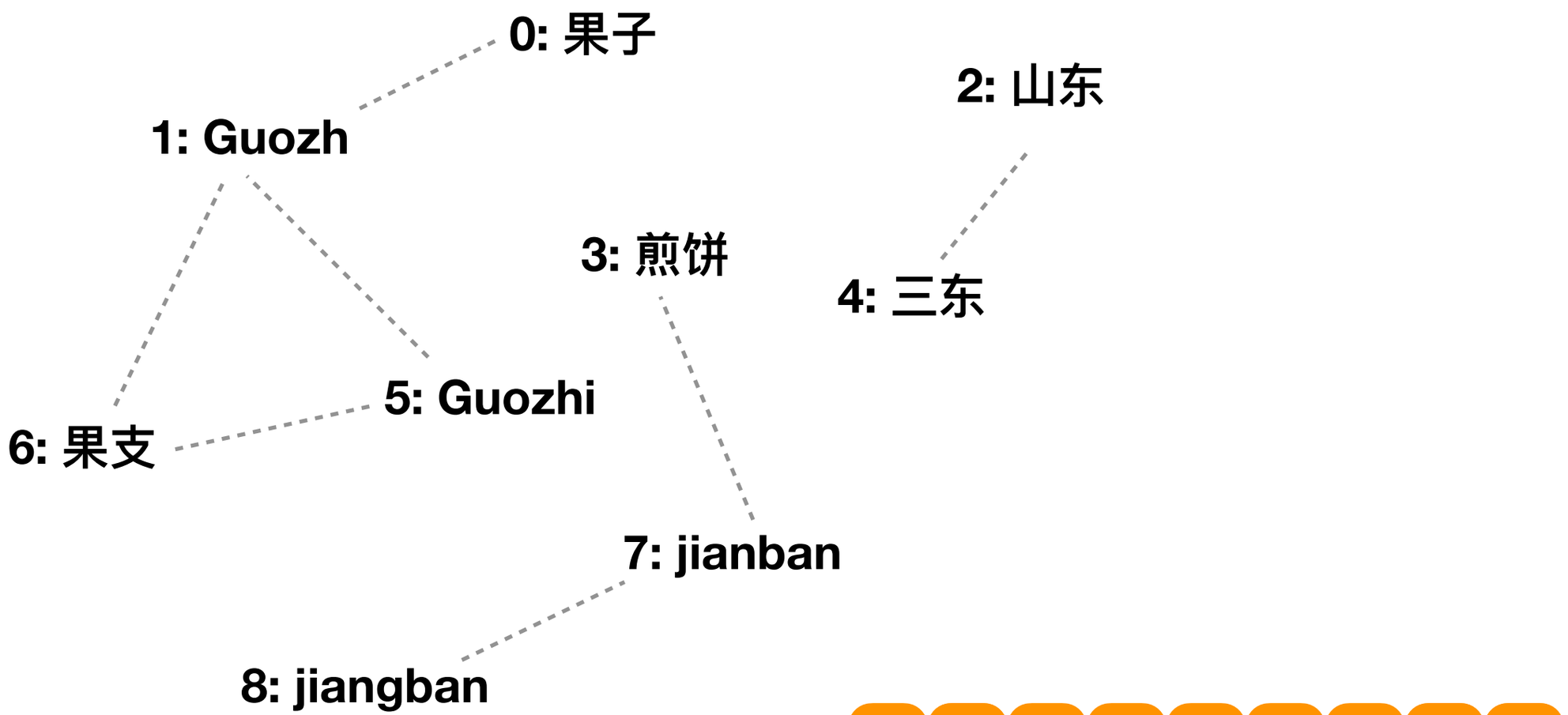


Guozh 果子 Guozhi

三东 山东

jianban 煎饼 jiangbang

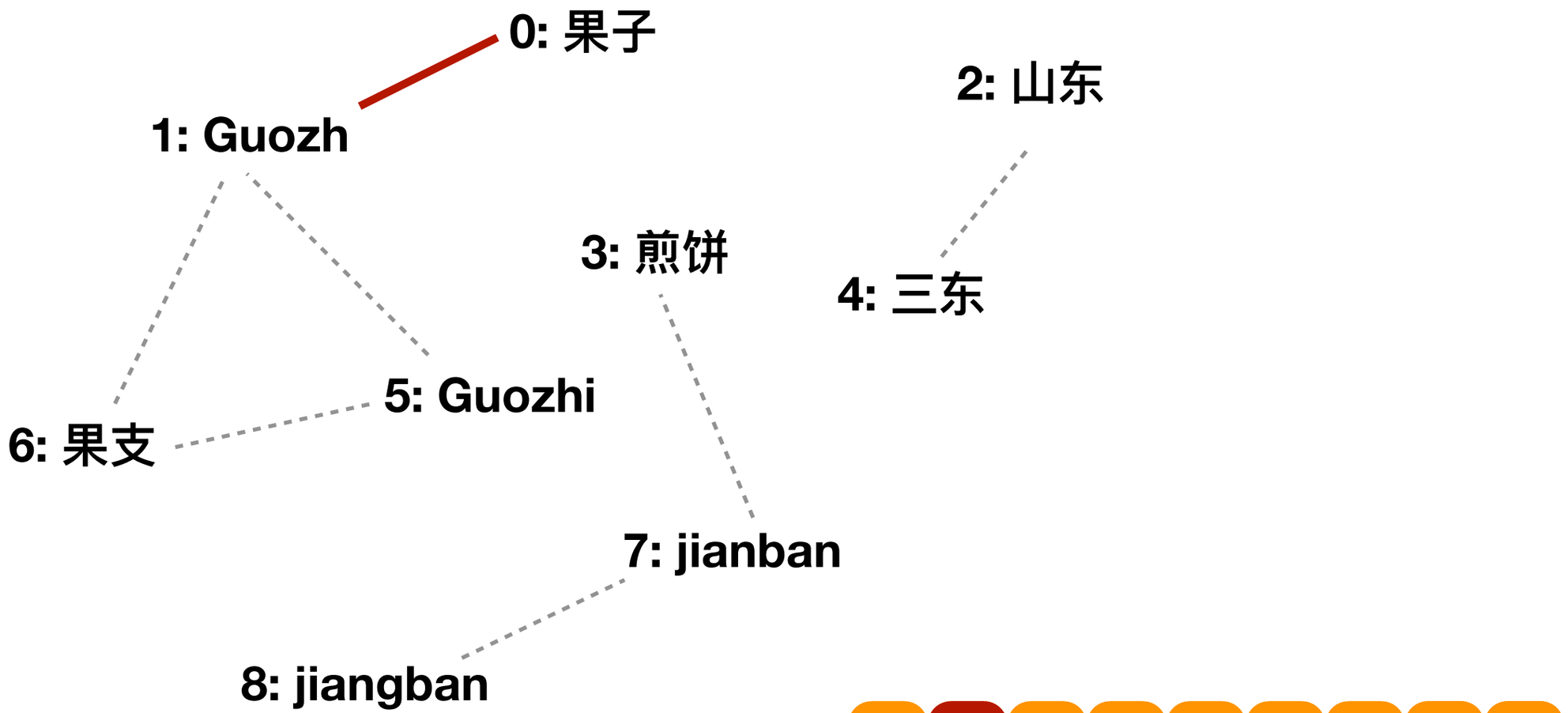
- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）



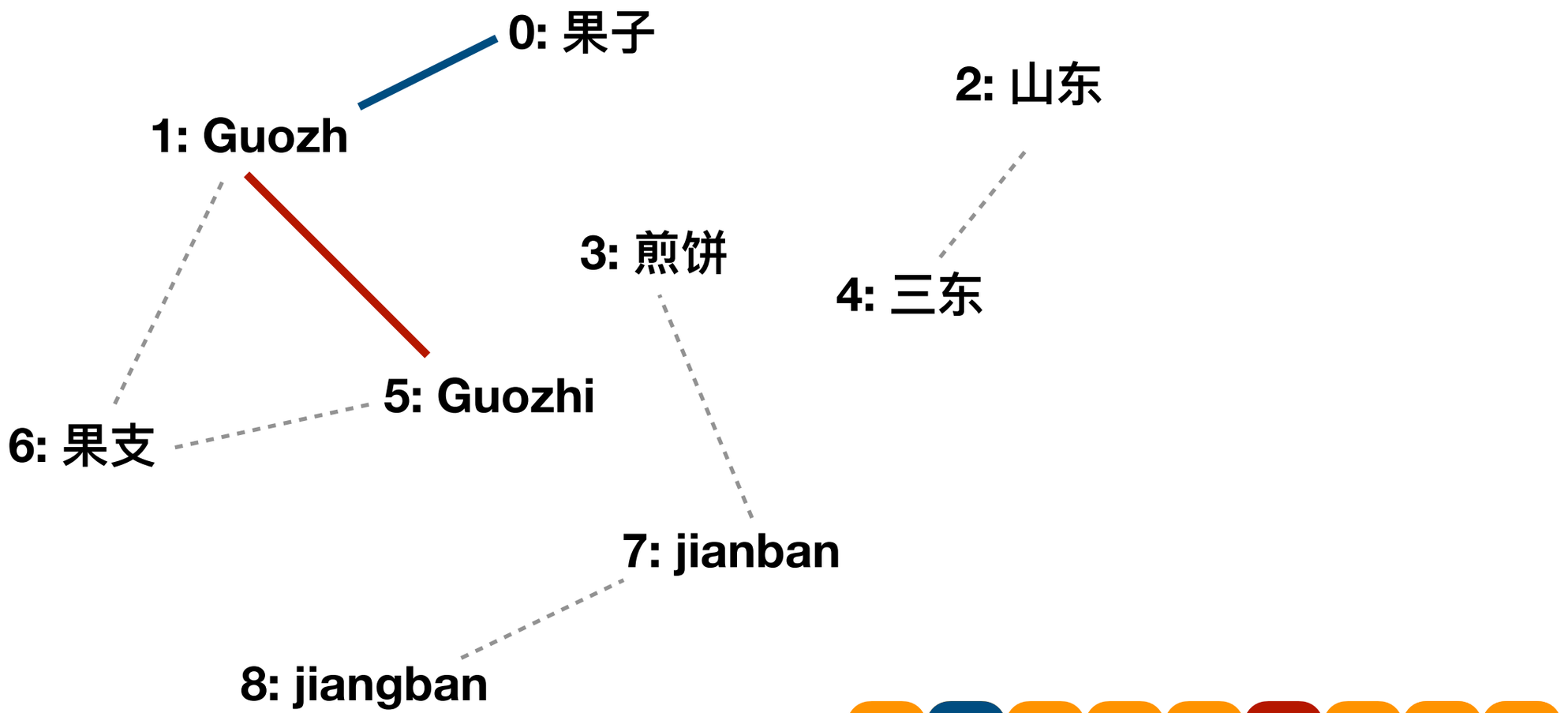
- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）

```
std::vector<int> make_disjoint_set(int size) {
    std::vector<int> d(size);
    for (int i = 0; i < size; ++i)
        d[i] = i;
    return std::move(d);
}

int find(const std::vector<int> &d, int x) {
    if (x != d[x])
        return find(d[x]);
    else
        return x;
}

void union(int x, int y, std::vector<int> &d) {
    x_root = find(d, x);
    y_root = find(d, y);
    if (x_root != y_root)
        d[x_root] = y_root;
}
```

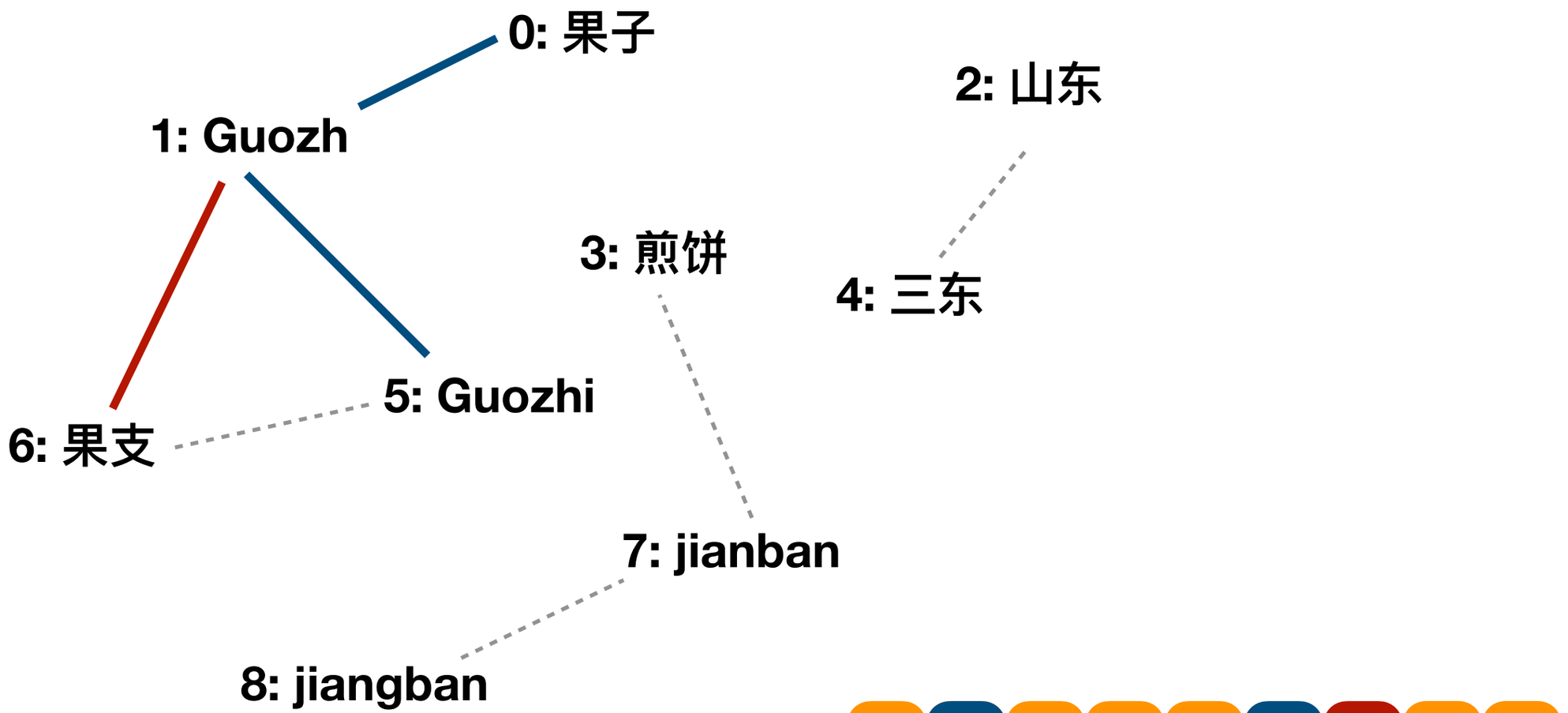
- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）



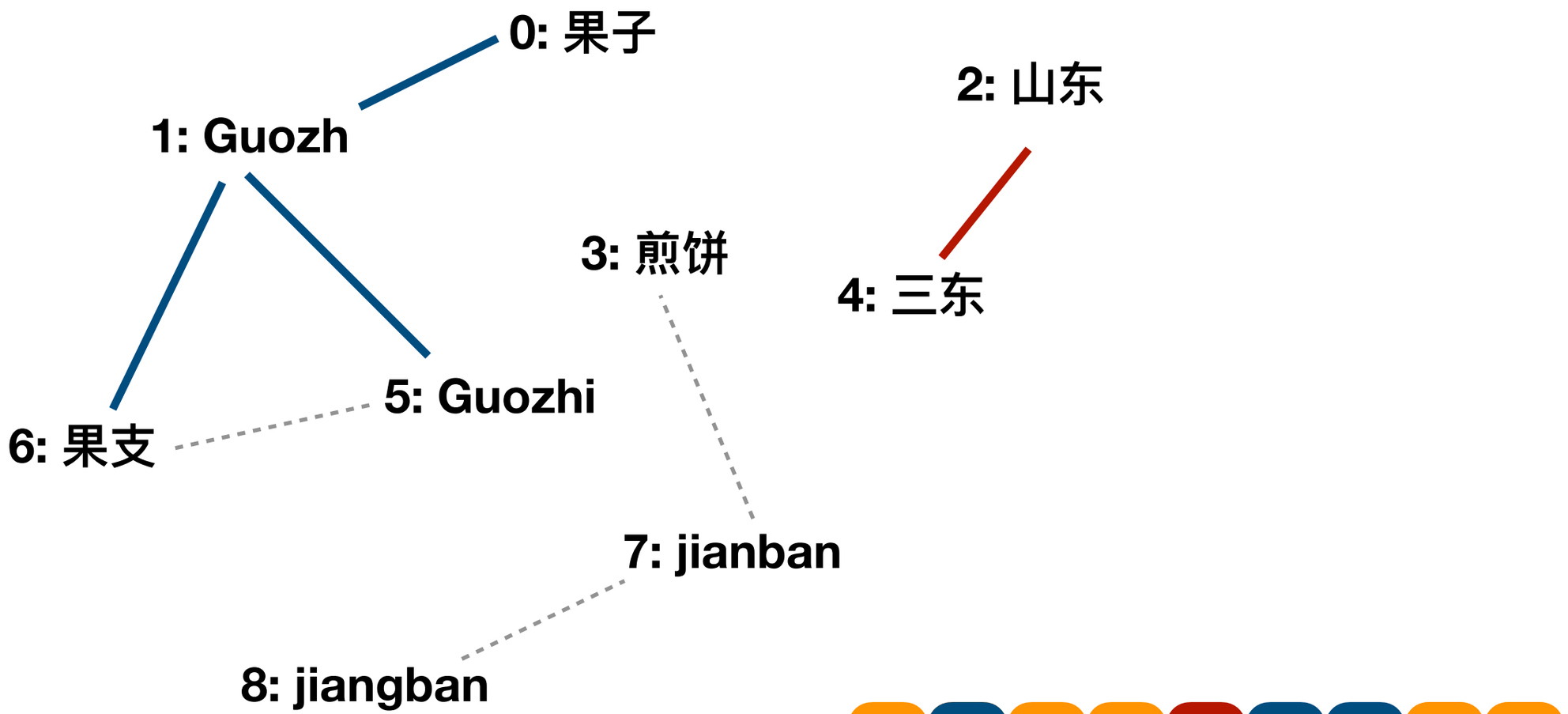
- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）



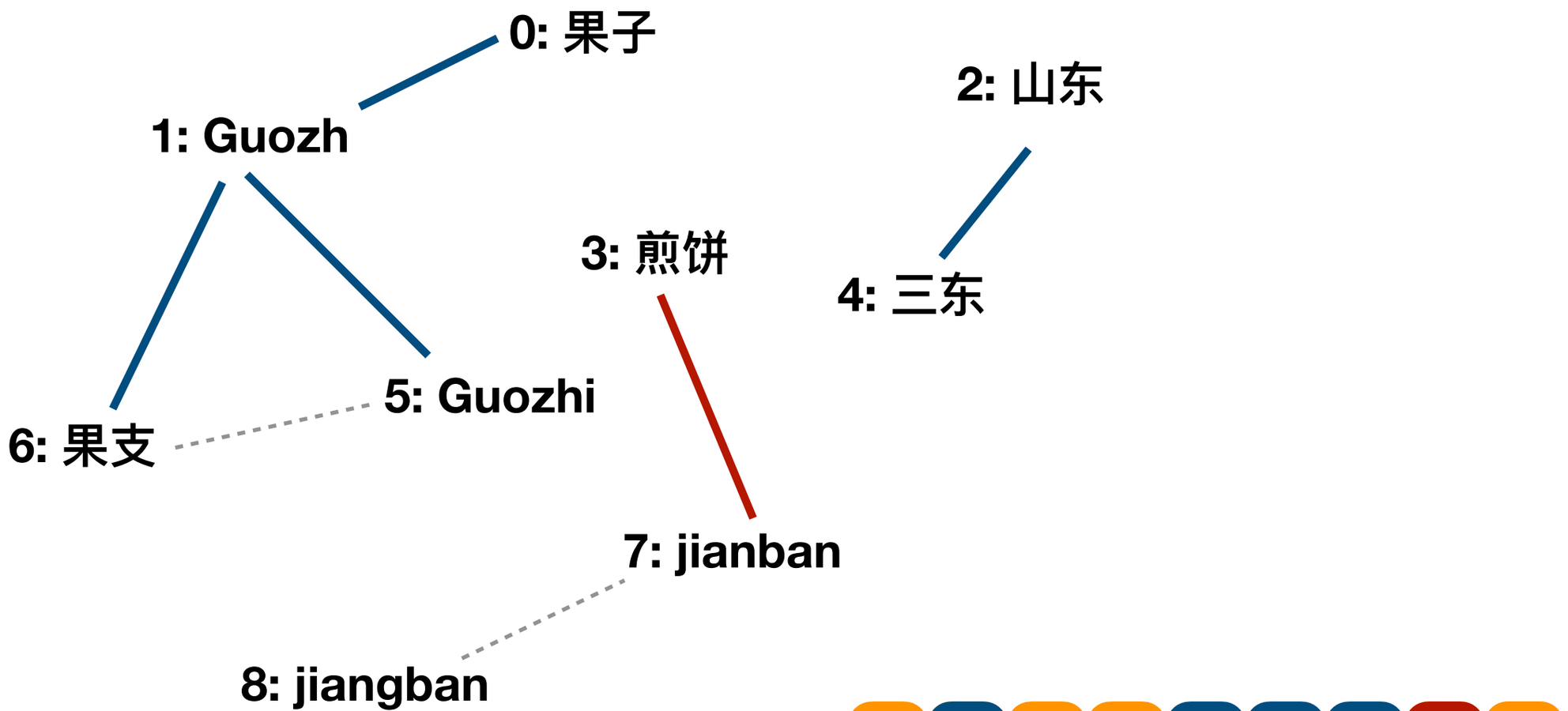
- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）



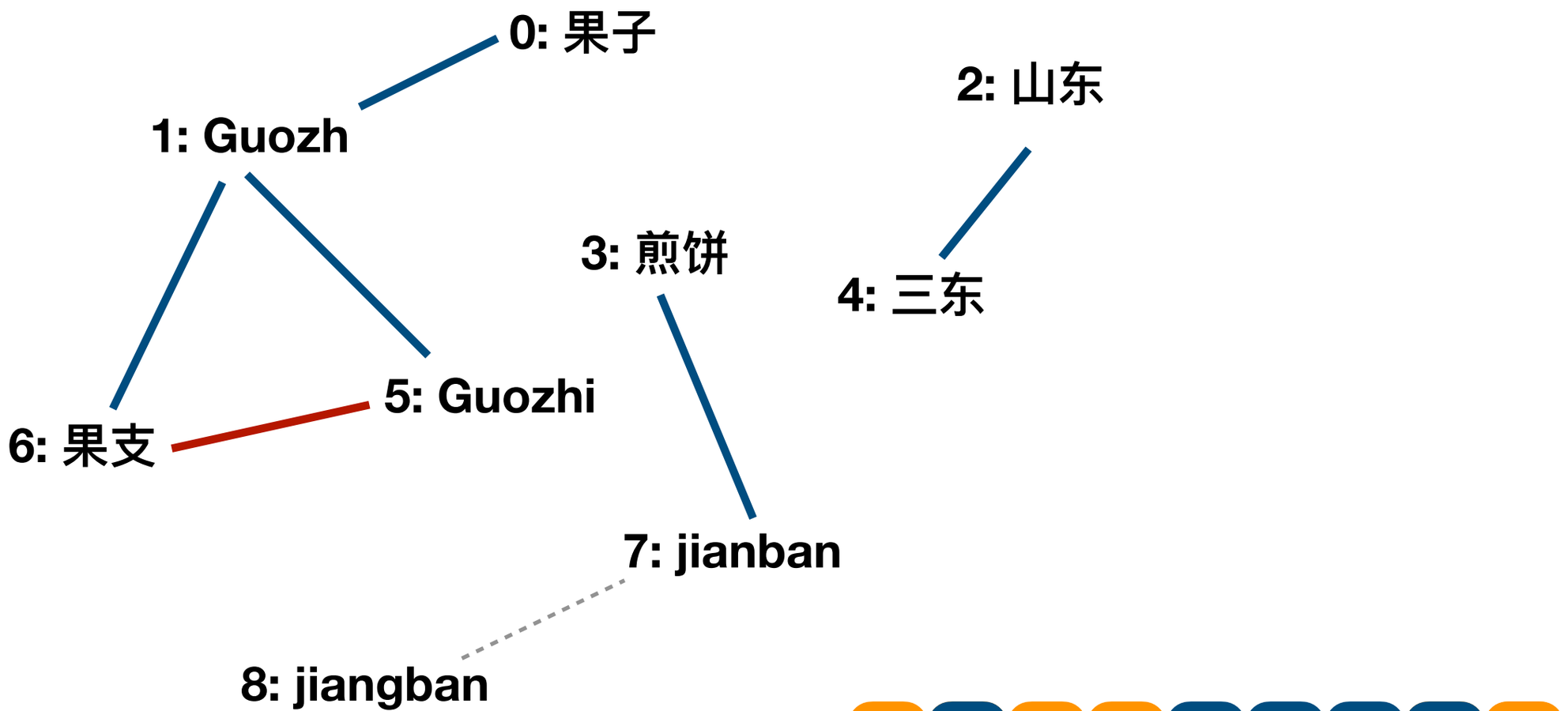
- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）



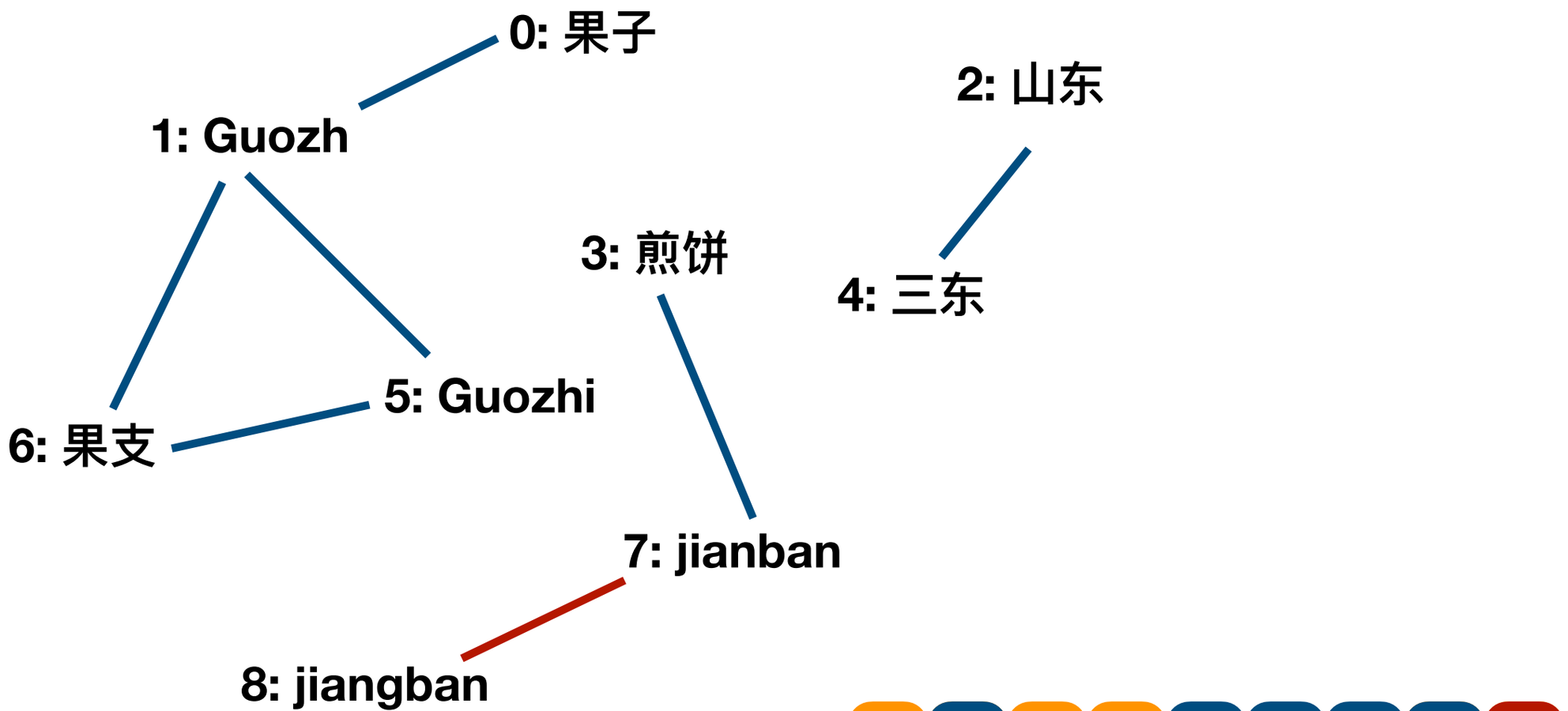
- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）



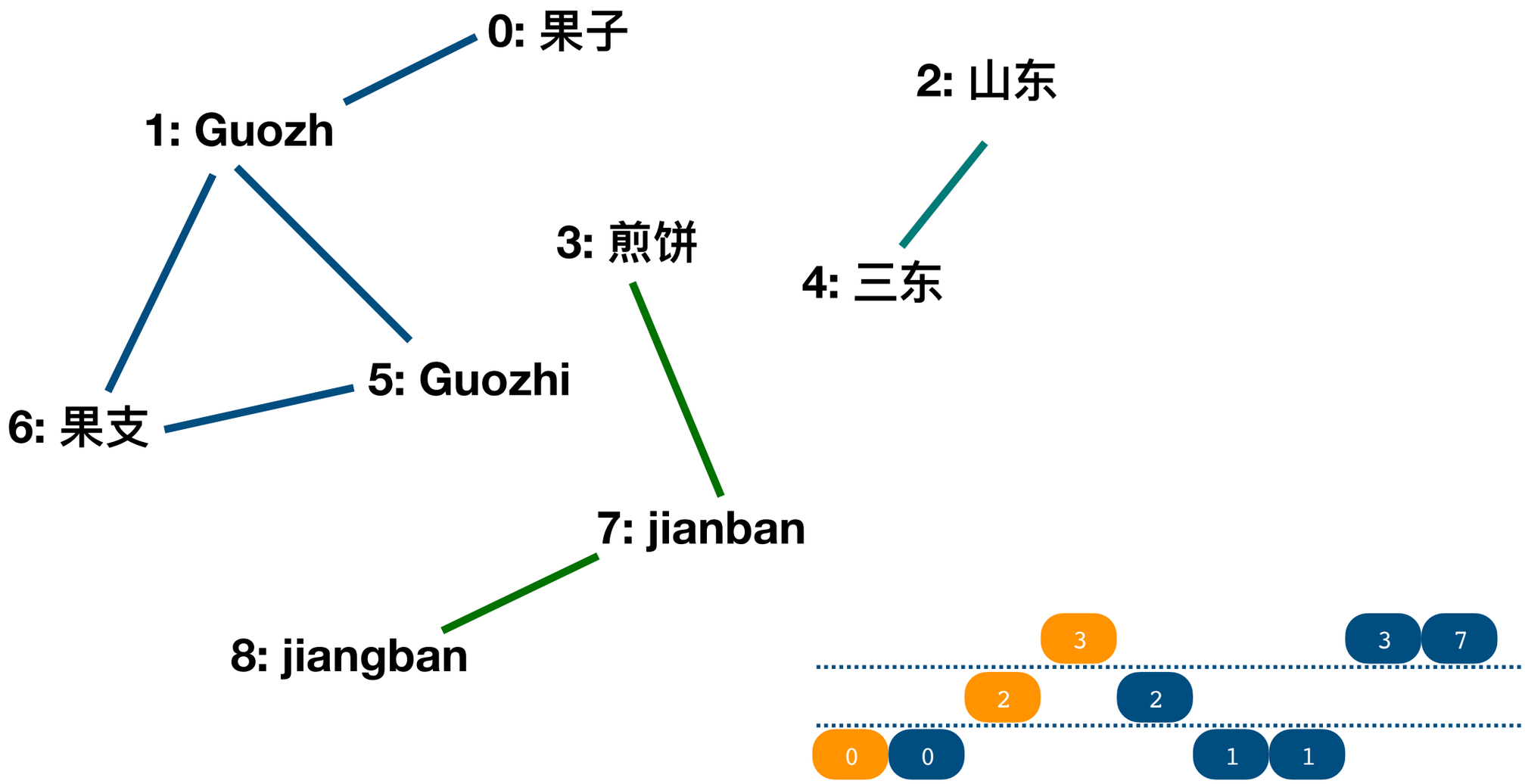
- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）



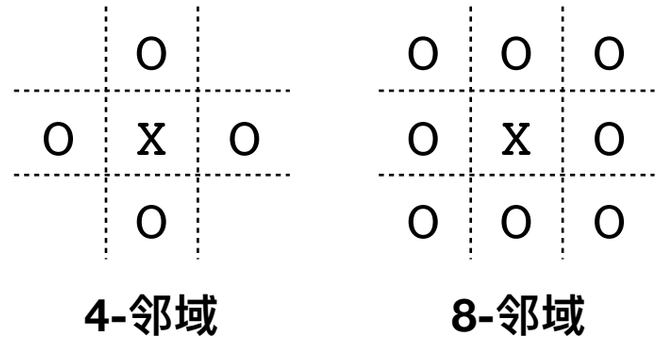
- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）

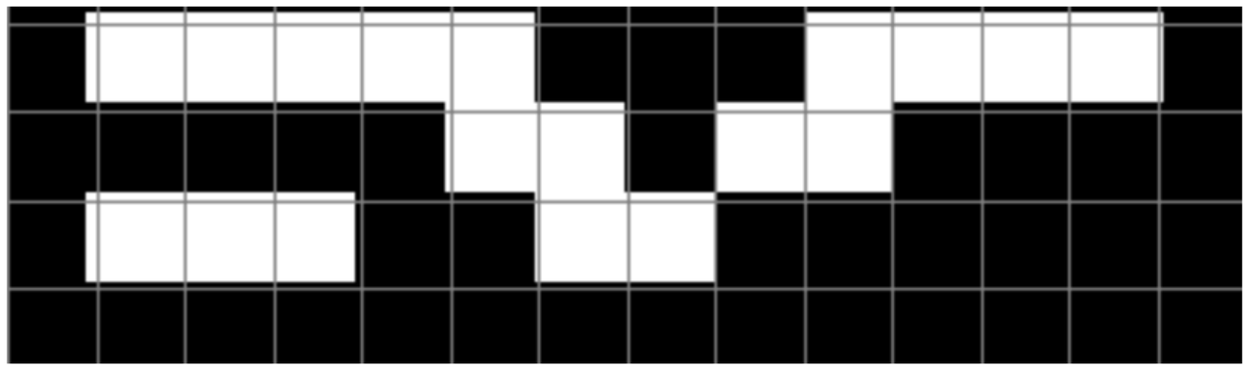


- 并查集
  - 并查集（Disjoint Set）是一种抽象数据类型（ADT）。
    - 可以将一系列不相交的（包含一个或多个元素的）集合进行合并（union），也可以获取某个元素所在的集合信息（find）

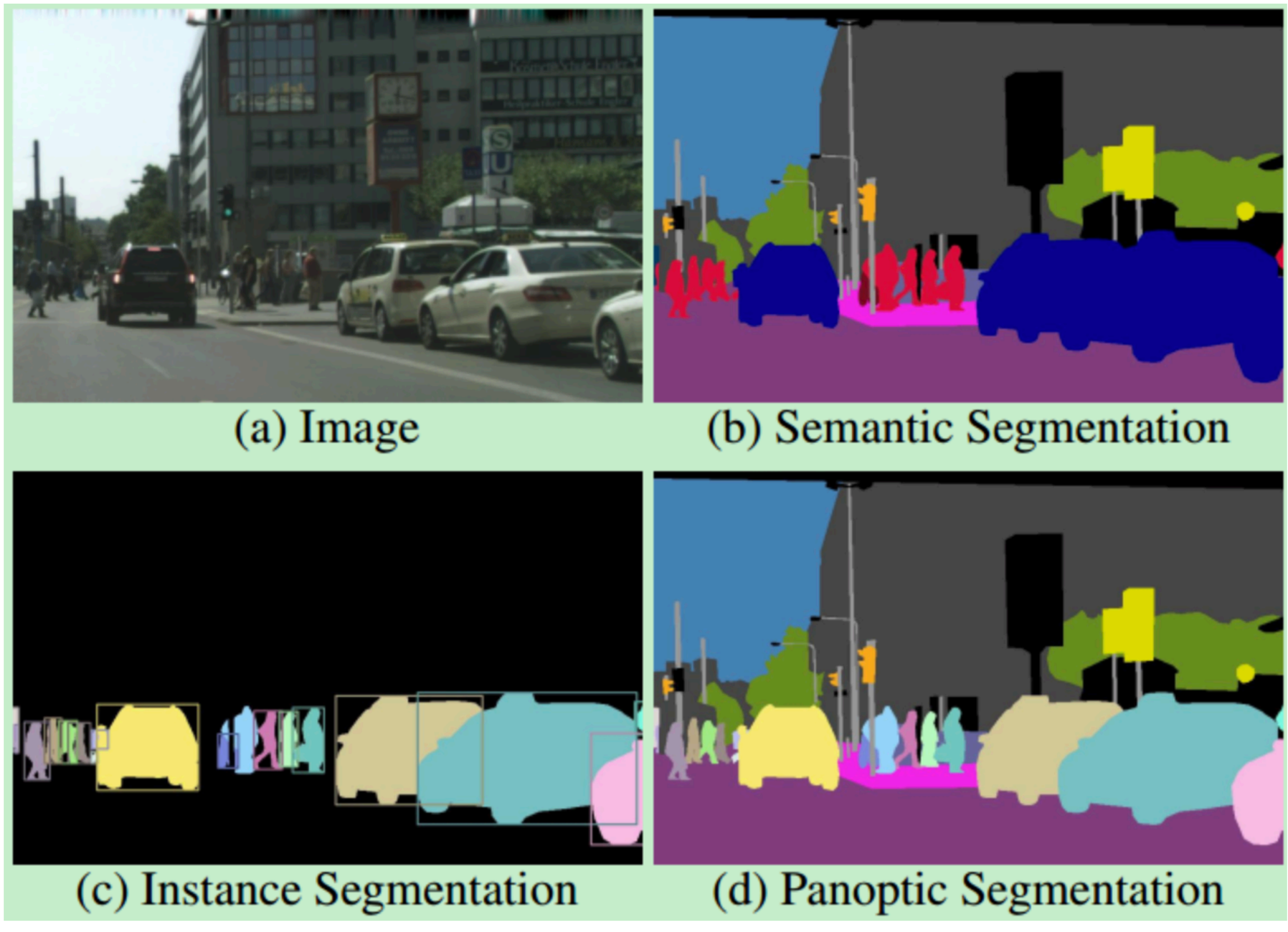


- 并查集
  - 应用
    - 二值图的连通区域





- 并查集
  - 应用
    - 实例分割



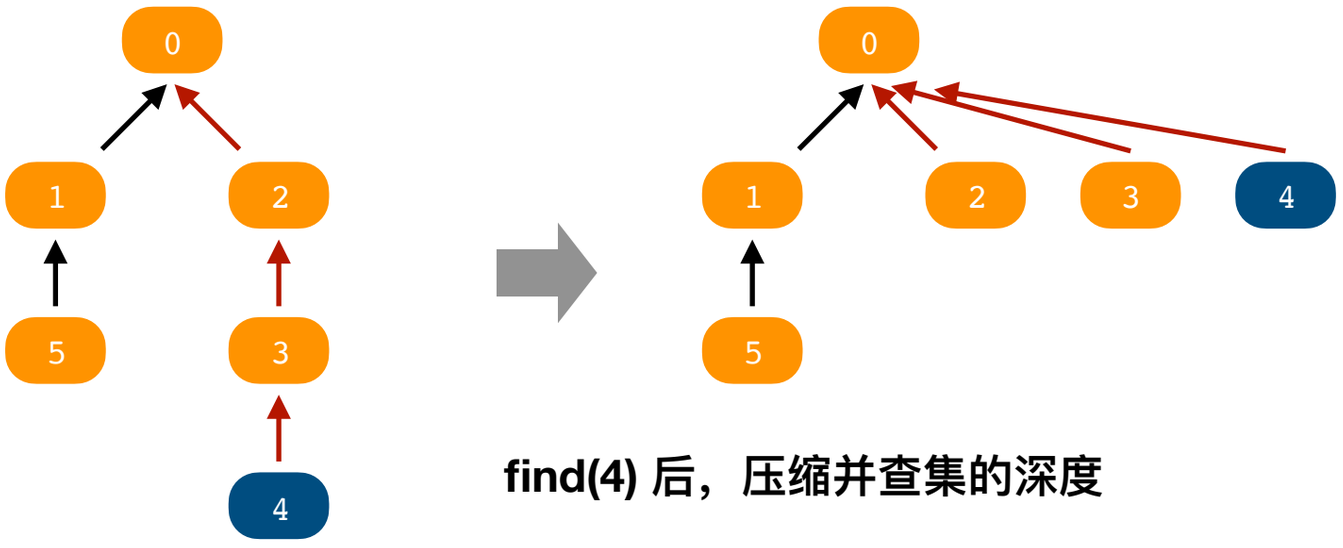
(a) Image (b) Semantic Segmentation

(c) Instance Segmentation (d) Panoptic Segmentation

目录

- 并查集
  - 并查集的优化
    - **路径压缩**
    - 按秩（集合深度）合并

```
int find(const std::vector<int> &d, int x) {
    if (x != d[x]) {
        d[x] = find(d[x]);
    }
    return d[x];
}
```



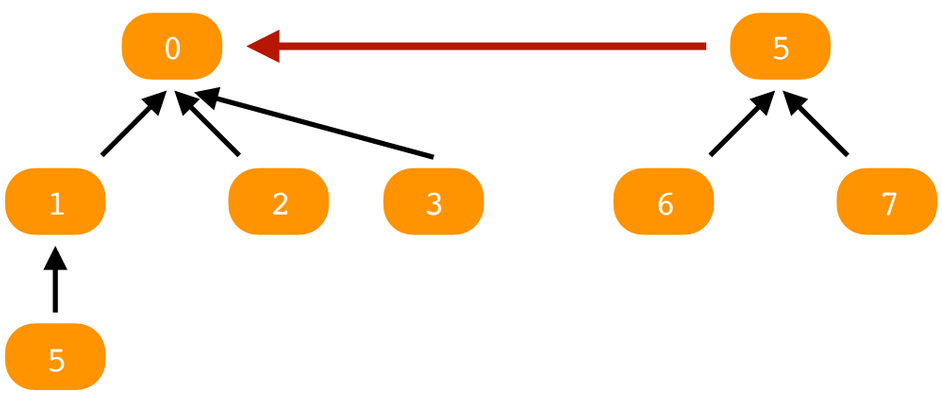
find(4) 后，压缩并查集的深度

- 并查集
  - 并查集的优化
    - 路径压缩
    - **按秩（集合深度）合并**

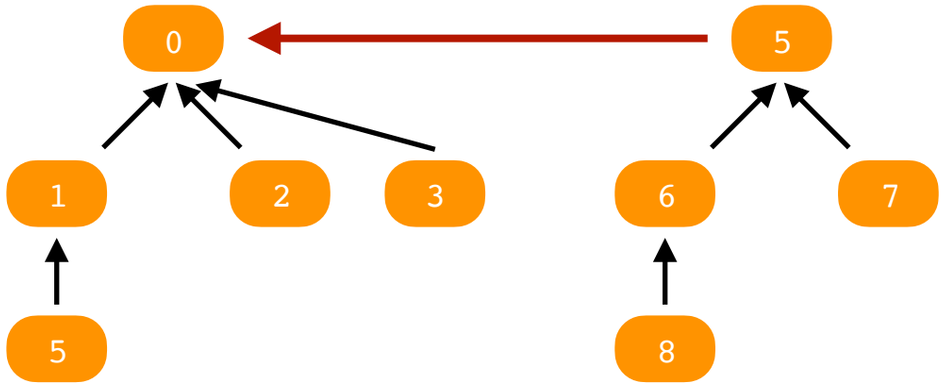
```
std::pair<std::vector<int>, std::vector<int> > make_disjoint_set(int size) {
    std::vector<int> d(size);
    std::vector<int> rank(size);
    for (int i = 0; i < size; ++i) {
        d[i] = i;
        rank[i] = 0;
    }
    return std::make_pair(std::move(d), std::move(rank));
}
```

- 并查集
  - 并查集的优化
    - 路径压缩
    - **按秩（集合深度）合并**

```
void union(int x, int y, std::vector<int> &d, std::vector<int> &rank) {
    x_root = find(d, x);
    y_root = find(d, y);
    if (x_root != y_root) {
        if (rank[x_root] > rank[y_root])
            d[y_root] = x_root;
        else if (rank[x_root] < rank[y_root])
            d[x_root] = y_root;
        else { // rank[x_root] == rank[y_root]
            d[y_root] = x_root;
            rank[x_root] += 1;
        }
    }
}
```



rank[5] < rank[0]，置 d[5] = 0，秩不变



rank[5] == rank[0]，置 d[5] = 0，且 rank[0] += 1

- 并查集
  - 并查集的扩展
    - **增加 min(x) / max(x) 操作，获取 X 所在集合的最小值/最大值**
    - **增加 de_union() 操作，允许撤回一次或多次 union 操作**