数据结构实验（10）

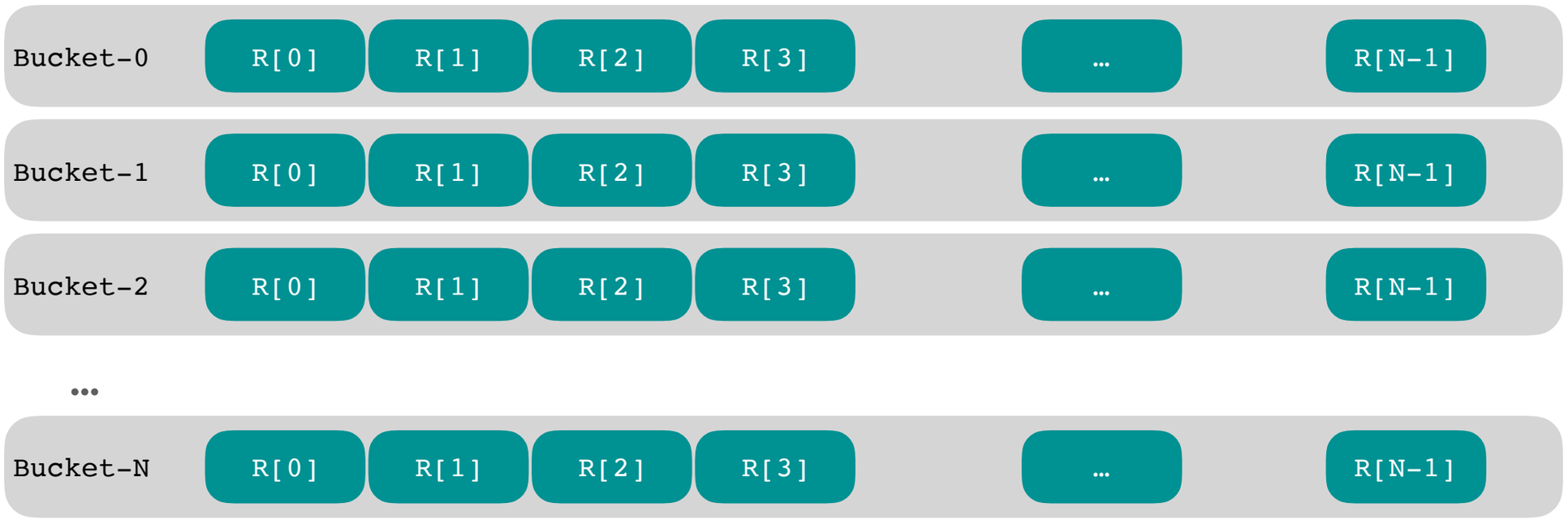
查找（1）

目录

- 神人 Robert Tarjan
  - https://www.cs.princeton.edu/~ret/
  - 他发明了：
    - 连通图：并查集（Disjoint Set）
    - 优先队列：Pairing Heap
    - 动态查找树：
      - Splay Tree
      - Top Trees（《Self-adjusting top trees》）

- 大段维护，局部朴素
  - 精致的算法往往很难准确实现
  - 别太在意算法的理论时间复杂度和问题的 Worst Case
    - “贪心过样例，暴力出奇迹，枚举枚上天，打表拿省一！”

- 大段维护，局部朴素
  - Bucket +
    - 数组
    - 链表
    - XX树
    - ...



- 大段维护，局部朴素
  - LeetCode 164. <Maximum Gap>

164. Maximum Gap

Hard 625 152 Favorite Share

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Return 0 if the array contains less than 2 elements.

Example 1:

```
Input: [3,6,9,1]
Output: 3
Explanation: The sorted form of the array is [1,3,6,9], either
             (3,6) or (6,9) has the maximum difference 3.
```

- Bucket Sort + Quick Sort

Bucket Size = (max_element - min_element) / nums

Bucket Count = (max_element - min_element) / Bucket Size

- 大段维护，局部朴素
  - LeetCode 164. <Maximum Gap>

```cpp
#include <algorithm>
#include <vector>

class Solution {
public:
    int maximumGap(std::vector<int>& nums) {
        int max_element = *std::max_element(nums.begin(), nums.end());
        int min_element = *std::min_element(nums.begin(), nums.end());

        int bucket_size = int((max_element - min_element) / nums.size()) + 1;
        int bucket_count = int((max_element - min_element) / bucket_size) + 1;

        std::vector<std::vector<int> > bucket(bucket_count);
        for (int k : nums)
            bucket[int((k - min_element) / bucket_size)].push_back(k);

        int max_gap = 0;
        int last = -1;
        for (auto it = bucket.begin(); it != bucket.end(); ++it) {
            sort(it->begin(), it->end());
            for (int k : *it) {
                if (last != -1 && k - last > max_gap)
                    max_gap = k - last;
                last = k;
            }
        }

        return max_gap;
    }
};
```

- 大段维护，局部朴素
  - LeetCode 169. <Majority Element>
  - LeetCode 229. <Majority Element II>
  - 扩展问题：
    - 输入 k 个正整数 N[0], N[1], …, N[k]，求任意区间 [a, b) 内出现次数最多的数
    - 输入 k 个正整数 N[0], N[1], …, N[k]，支持两种操作：
      - 1. 求任意区间 [a, b) 内的数值总和
      - 2. 修改任意区间 [a, b) ，区间内所有数字加上或减去M

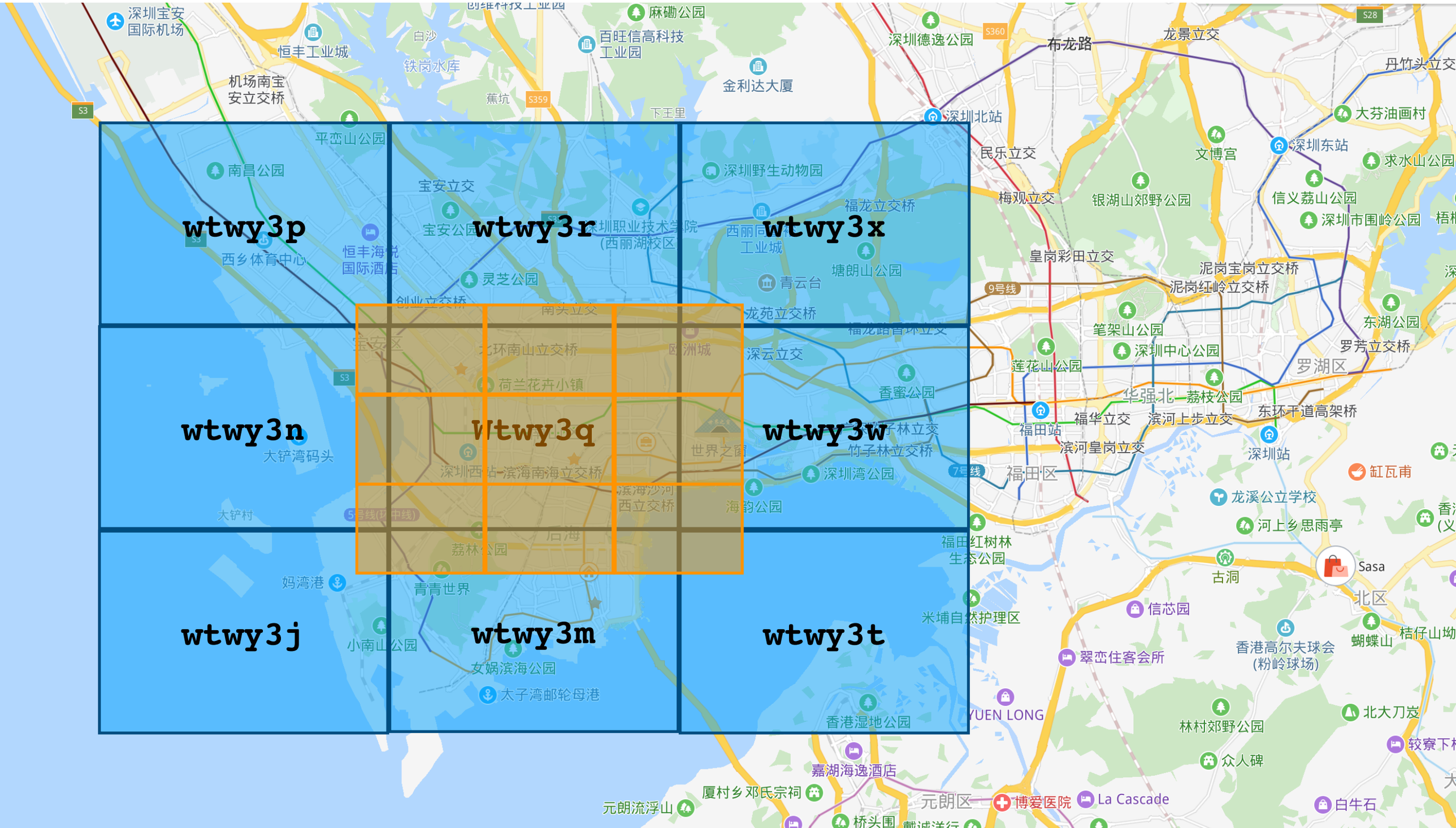
- 大段维护，局部朴素
  - 梓豪的问题：在平面上给定 N 个点（[x, y] 表示），输入其中任意一点 P，求离它最近的 K 个点
  - 扩展问题：
    - 附近的人
    - 附近的餐厅

共找到"美食"相关149个结果

- 大段维护，局部朴素
  - GeoHash
    - 一种地理位置编码方法，把空间分隔为网格
    - 分级划分，每一级的网格大小如右图；
    - 按照经度+纬度编码

字符串长度		cell 宽度		cell 高度
1	≤	5,000km	×	5,000km
2	≤	1,250km	×	625km
3	≤	156km	×	156km
4	≤	39.1km	×	19.5km
5	≤	4.89km	×	4.89km
6	≤	1.22km	×	0.61km
7	≤	153m	×	153m
8	≤	38.2m	×	19.1m
9	≤	4.77m	×	4.77m
10	≤	1.19m	×	0.596m
11	≤	149mm	×	149mm
12	≤	37.2mm	×	18.6mm

- 大段维护，局部朴素
  - GeoHash



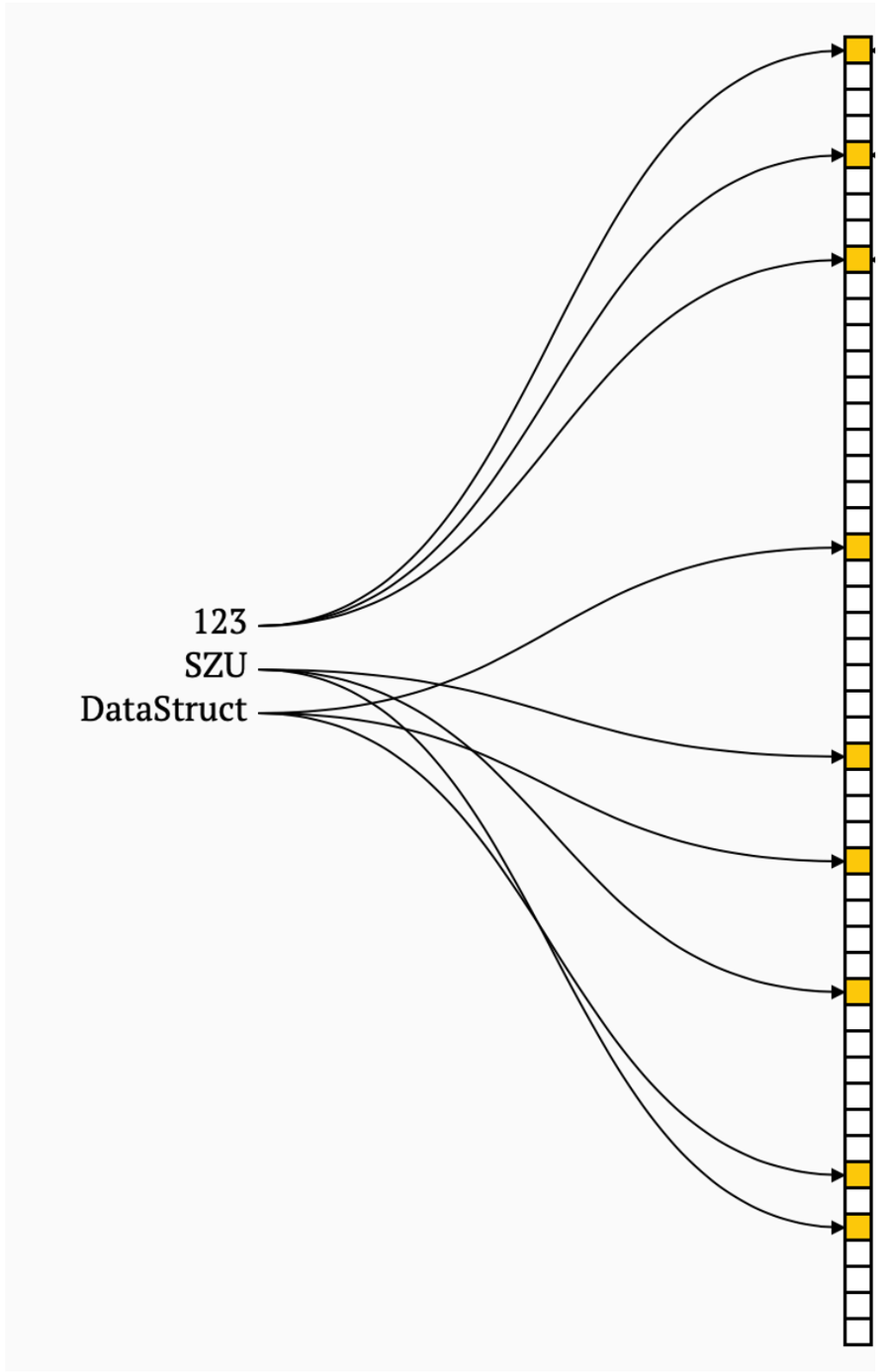
- 大段维护，局部朴素
  - GeoHash
    - 将二维空间映射到一维
    - 查找临近的人
      - 搜索拥有相同公共前缀的字符串对应的网格里的所有点
  - 扩展
    - Annoy：LSH（局部敏感哈希），将向量A、B映射到低维空间中的两个签名向量，并且近似保持A、B之间的相似度
    - Google S2 Library：球面算法库

Google's S2 library is a real treasure, not only due to its capabilities for spatial indexing but also because it is a library that was released more than 4 years ago and it didn't get the attention it deserved

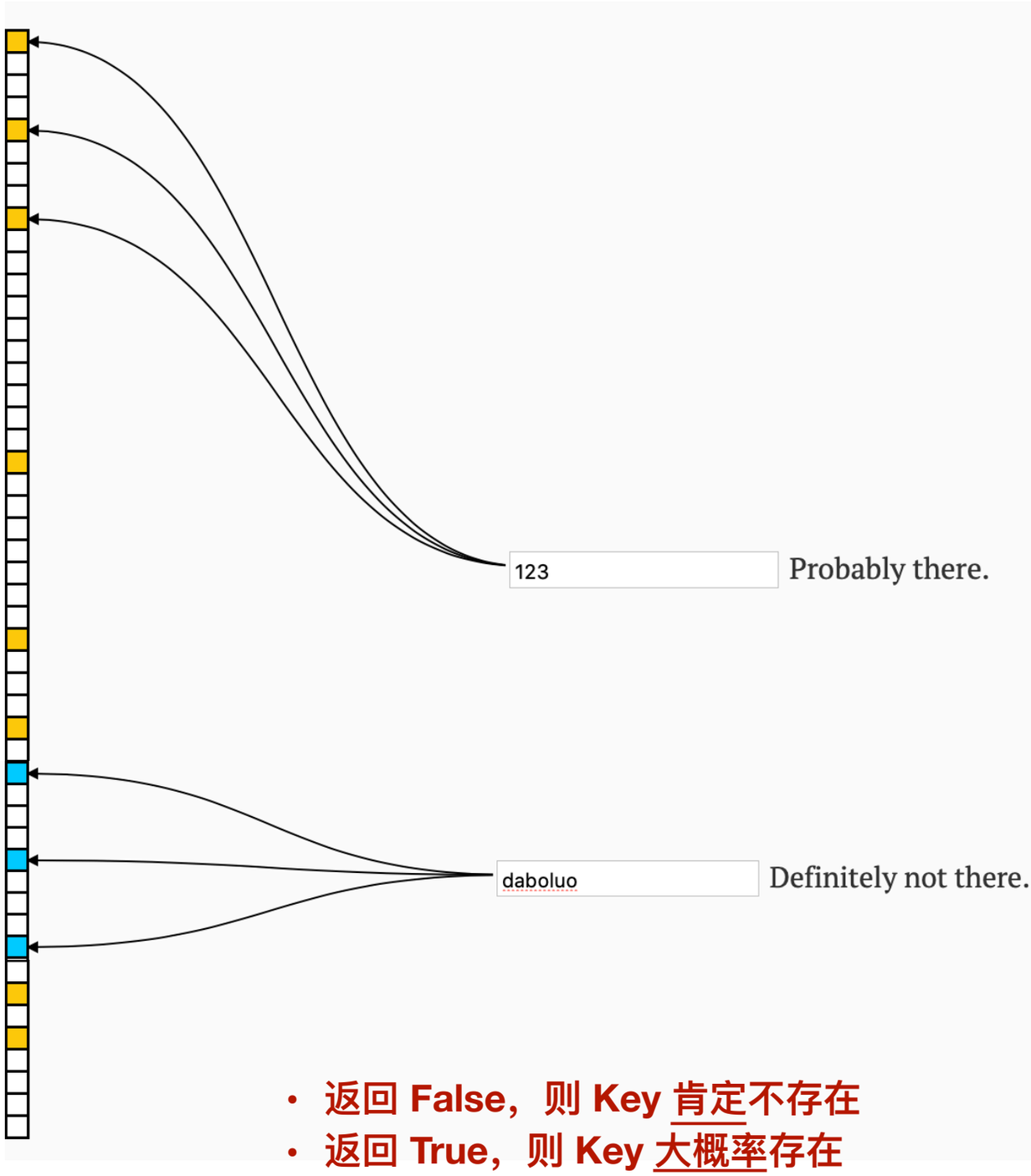
- 大段维护，局部朴素
  - Bucket + Bucket + Bucket ..
  - Hash 表的存储效率通常只有 50%
  - 改进方案：
    - Bloom Filter：组合使用 Bucket
    - Cuckoo Hash：提高 Hash 表存储效率

- 大段维护，局部朴素
  - Bloom Filter
    - 使用场景：
      - 判断一个元素是否在某个集合（海量数据）中，例如：
        - Email / URL / 用户ID / 手机号码 黑名单
        - 数据库索引
    - 单机的内存无法使用 HashTable 或动态查找树存储集合
    - 需支持：set() 和 exists() 操作

- 大段维护，局部朴素
  - Bloom Filter
    - 根据集合数据量和单机内存，设置长度为 N 的 0/1 Bucket，默认值为 0
    - 选择一组 Hash 函数
  - set(key)：
    - 设置 h1(key) mod N, …, hk(key) mod N 为 1
  - exists(key)：
    - 返回 h1(key) mod N & … & hk(key) mod N



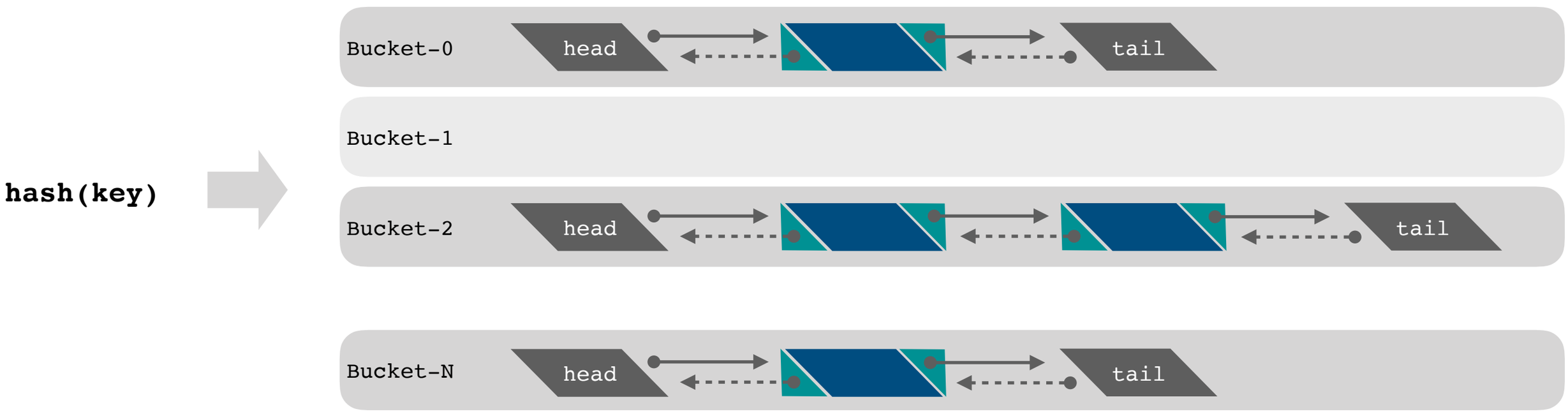
- 大段维护，局部朴素
  - Bloom Filter
    - 根据集合数据量和单机内存，设置长度为 N 的 0/1 Bucket，默认值为 0
    - 选择一组 Hash 函数
    - set(key)：
      - 设置 h1(key) mod N, …, hk(key) mod N 为 1
    - exists(key)：
      - 返回 h1(key) mod N & … & hk(key) mod N



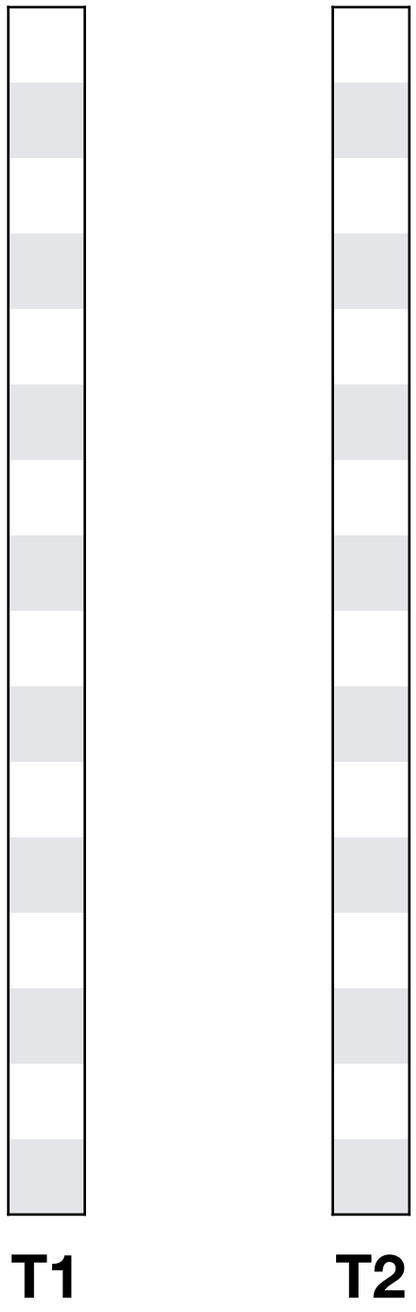
- **返回 False，则 Key 肯定不存在**
- **返回 True，则 Key 大概率存在**
- **不能删除 BloomFilter 的数据**

- 大段维护，局部朴素
  - Bloom Filter
    - 扩展：
      - 改进：
        - 支持删除 - 《Bloom Filters via d-Left Hashing and Dynamic Bit Reassignment Extended Abstract》
    - 工程应用：
      - 《Approximately detecting duplicates for streaming data using stable bloom filters》
      - 《Mutable strings in Java: design, implementation and lightweight text-search algorithms》
      - 《Informed content delivery across adaptive overlay networks》

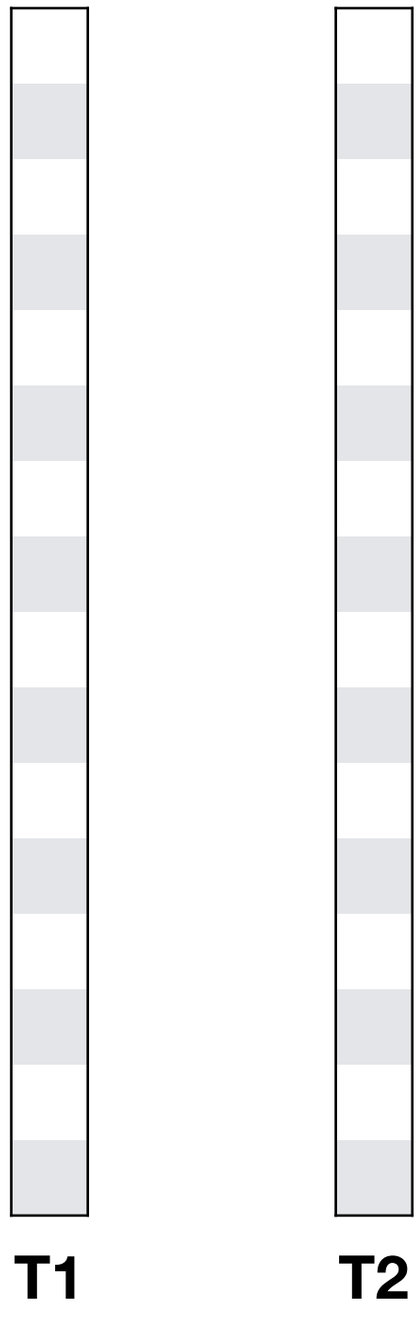
- 大段维护，局部朴素
  - Cuckoo Hash
    - 普通 Hash Table 通常用拉链法解决 Hash 冲突，缺点：
      - 查找操作的最坏时间复杂度 O(N)
      - Hash 表存储效率低 50%（满足一定冲突率下，元素数量 / Bucket 大小）



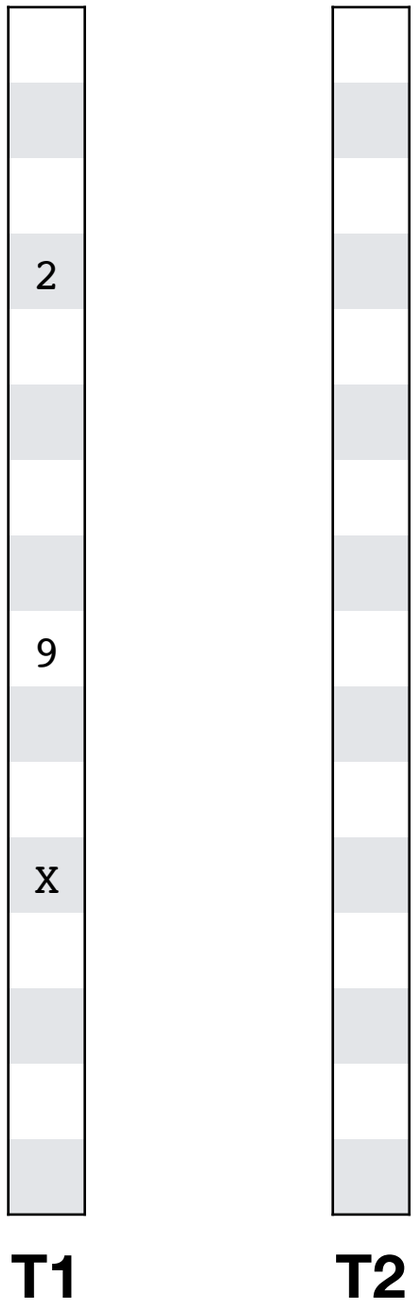
- 大段维护，局部朴素
  - Cuckoo Hash
    - 使用两个不同的 Hash 函数 h1 和 h2
    - 使用两个长度为 N 的 Bucket T1 和 T2
    - 对于每个被插入元素 x，要么在 T1 的 `h1(x)`，要么在 T2 的 `h2(x)` 处
      - exists(x)、get(x) 操作时间复杂度 O(1)



- 大段维护，局部朴素
  - Cuckoo Hash
    - `insert(x)`：
      - 将 `x` 插入 `T1` 的 `h1(x)` 位置
        - 若 `T1 h1(x)` 为空，则插入成功；
        - 若 `T1` h1(x) 不为空，则将该位置上的元素 `y`，移动到 `T2 h2(y)` 处；
          - 若 `T2 h2(y)`为空，则插入成功；
          - 若 `T2 h2(y)`不为空，则将该位置上的元素 `z`，移动到 `T1 h1(z)` 处；
            - 若 `T1 h1(z)` 为空...
            - 反复移动，直到不再有冲突

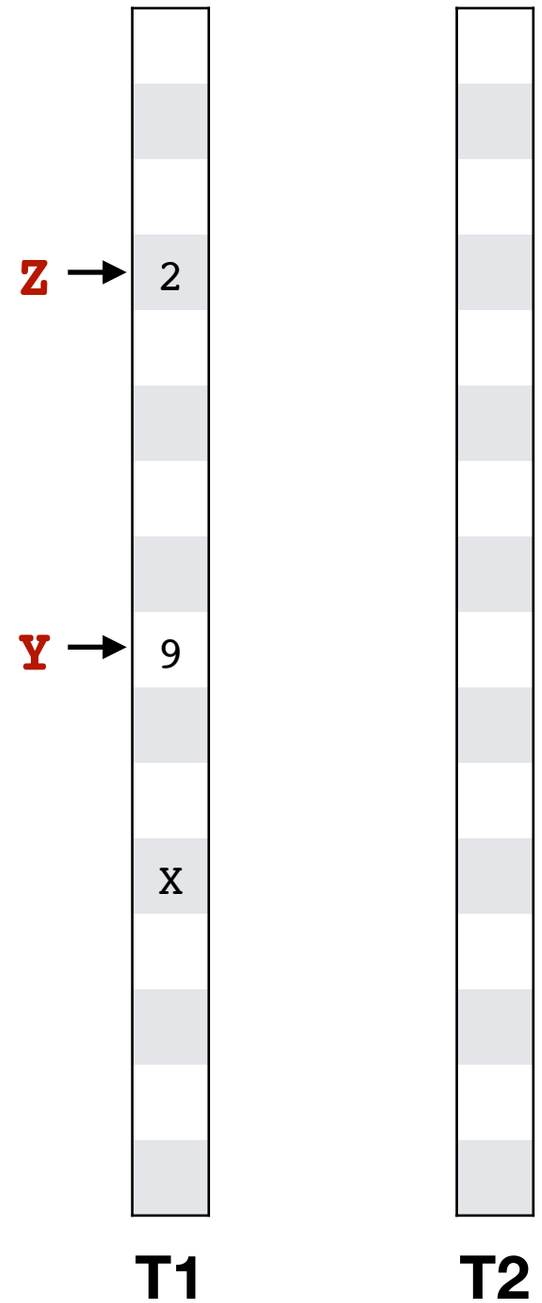


- 大段维护，局部朴素
  - Cuckoo Hash
    - `insert(x)`：
      - 将 `x` 插入 `T1` 的 `h1(x)` 位置
        - 若 `T1 h1(x)` 为空，则插入成功；
        - 若 `T1` h1(x) 不为空，则将该位置上的元素 `y`，移动到 `T2 h2(y)` 处；
          - 若 `T2 h2(y)`为空，则插入成功；
          - 若 `T2 h2(y)`不为空，则将该位置上的元素 `z`，移动到 `T1 h1(z)` 处；
            - 若 `T1 h1(z)` 为空...
            - 反复移动，直到不再有冲突



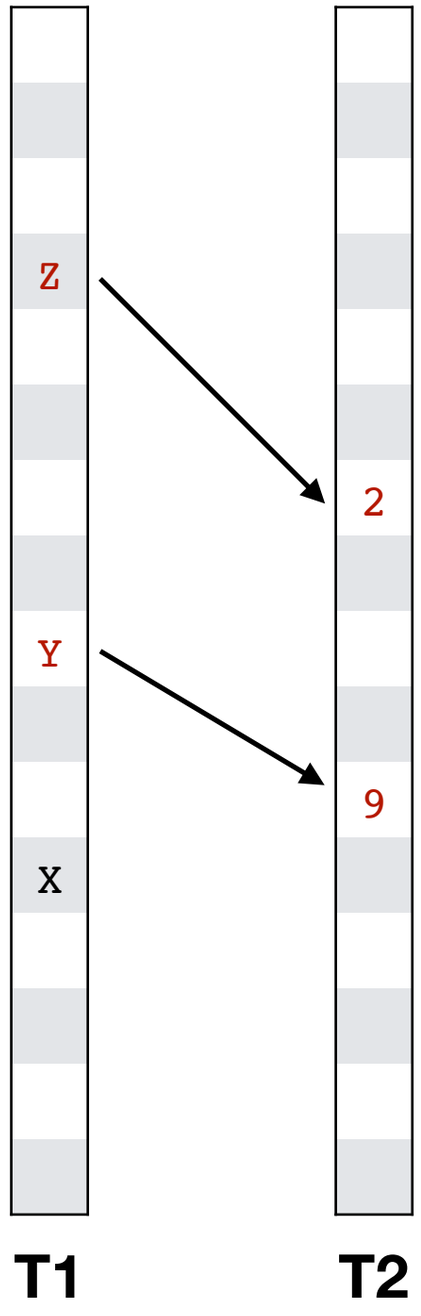

```
insert("2")
insert("9")
insert("X")
```

- 大段维护，局部朴素
  - Cuckoo Hash
    - `insert(x)`：
      - 将 `x` 插入 `T1` 的 `h1(x)` 位置
        - 若 `T1 h1(x)` 为空，则插入成功；
        - 若 `T1` h1(x) 不为空，则将该位置上的元素 `y`，移动到 `T2 h2(y)` 处；
          - 若 `T2 h2(y)`为空，则插入成功；
          - 若 `T2 h2(y)`不为空，则将该位置上的元素 `z`，移动到 `T1 h1(z)` 处；
            - 若 `T1 h1(z)` 为空...
            - 反复移动，直到不再有冲突



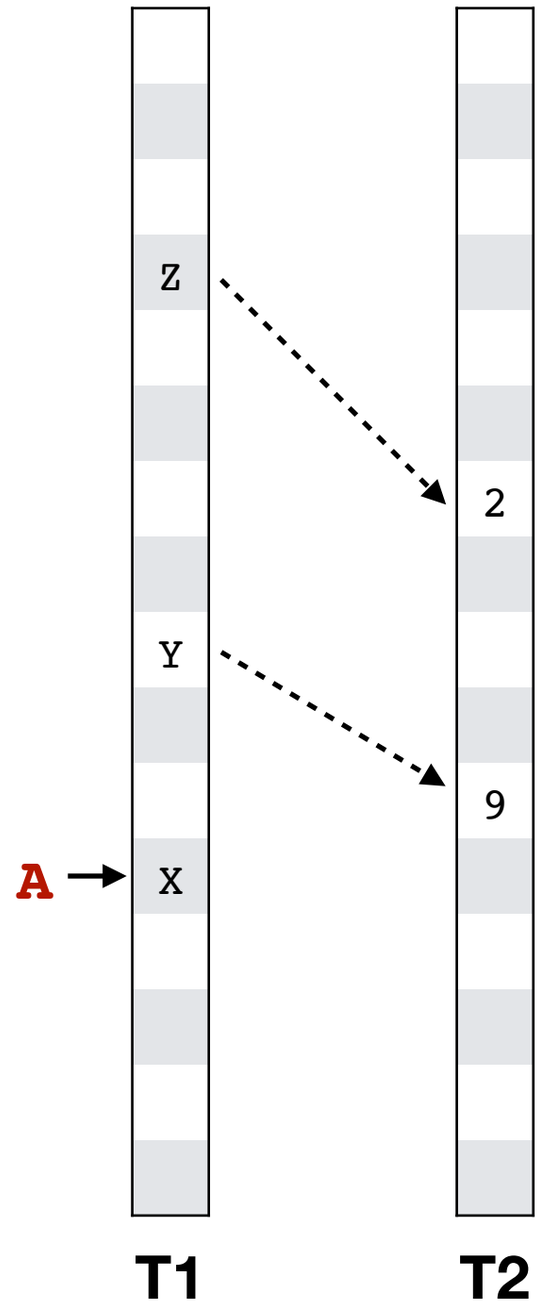

```
insert("Y")
insert("Z")
```

- 大段维护，局部朴素
  - Cuckoo Hash
    - `insert(x)`：
      - 将 `x` 插入 `T1` 的 `h1(x)` 位置
        - 若 `T1 h1(x)` 为空，则插入成功；
        - 若 `T1` h1(x) 不为空，则将该位置上的元素 `y`，移动到 `T2 h2(y)` 处；
          - 若 `T2 h2(y)`为空，则插入成功；
          - 若 `T2 h2(y)`不为空，则将该位置上的元素 `z`，移动到 `T1 h1(z)` 处；
            - 若 `T1 h1(z)` 为空...
            - 反复移动，直到不再有冲突



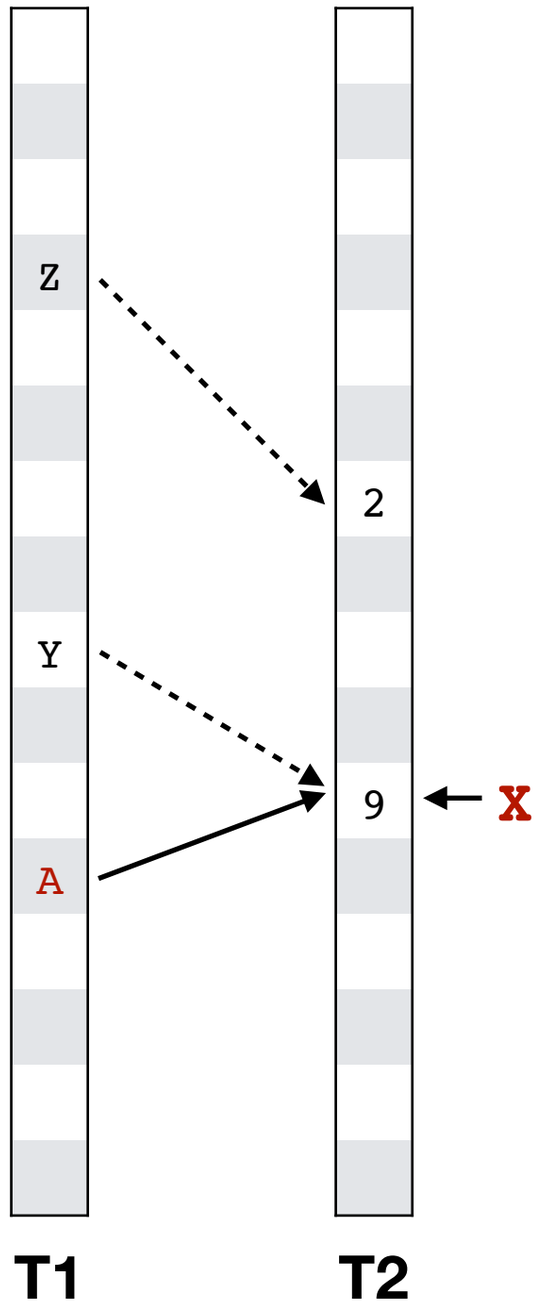

```
insert("Y")
insert("Z")
```

- 大段维护，局部朴素
  - Cuckoo Hash
    - `insert(x)`：
      - 将 `x` 插入 `T1` 的 `h1(x)` 位置
        - 若 `T1 h1(x)` 为空，则插入成功；
        - 若 `T1` h1(x) 不为空，则将该位置上的元素 `y`，移动到 `T2 h2(y)` 处；
          - 若 `T2 h2(y)`为空，则插入成功；
          - 若 `T2 h2(y)`不为空，则将该位置上的元素 `z`，移动到 `T1 h1(z)` 处；
            - 若 `T1 h1(z)` 为空...
            - 反复移动，直到不再有冲突



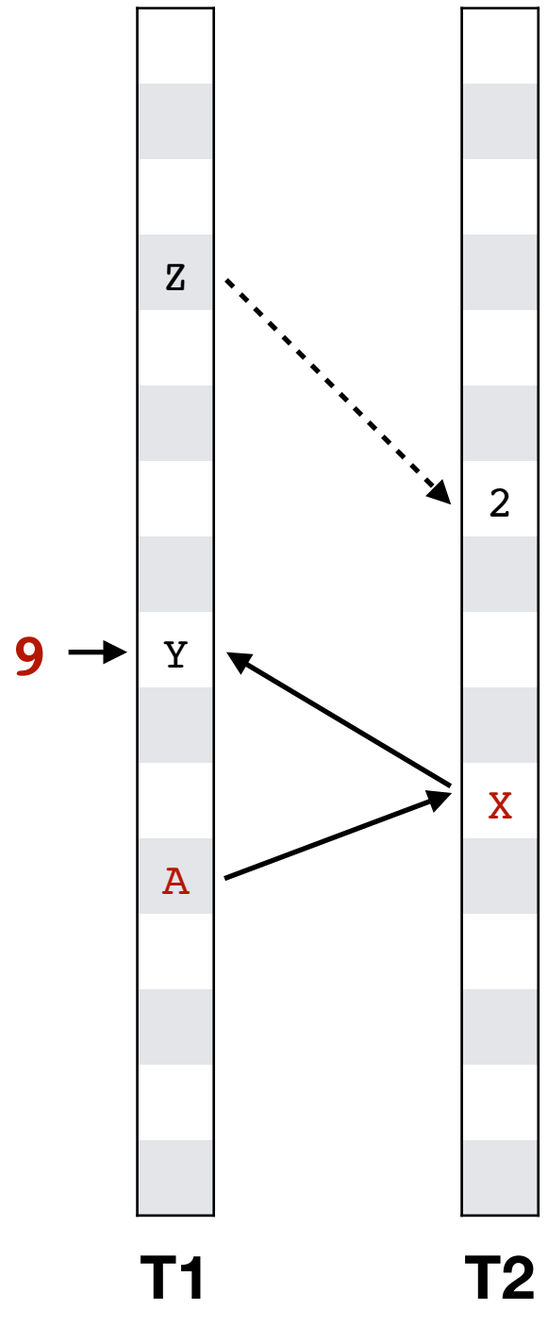
insert("A")

- 大段维护，局部朴素
  - Cuckoo Hash
    - `insert(x)`：
      - 将 `x` 插入 `T1` 的 `h1(x)` 位置
        - 若 `T1 h1(x)` 为空，则插入成功；
        - 若 `T1` h1(x) 不为空，则将该位置上的元素 `y`，移动到 `T2 h2(y)` 处；
          - 若 `T2 h2(y)`为空，则插入成功；
          - 若 `T2 h2(y)`不为空，则将该位置上的元素 `z`，移动到 `T1 h1(z)` 处；
            - 若 `T1 h1(z)` 为空...
            - 反复移动，直到不再有冲突



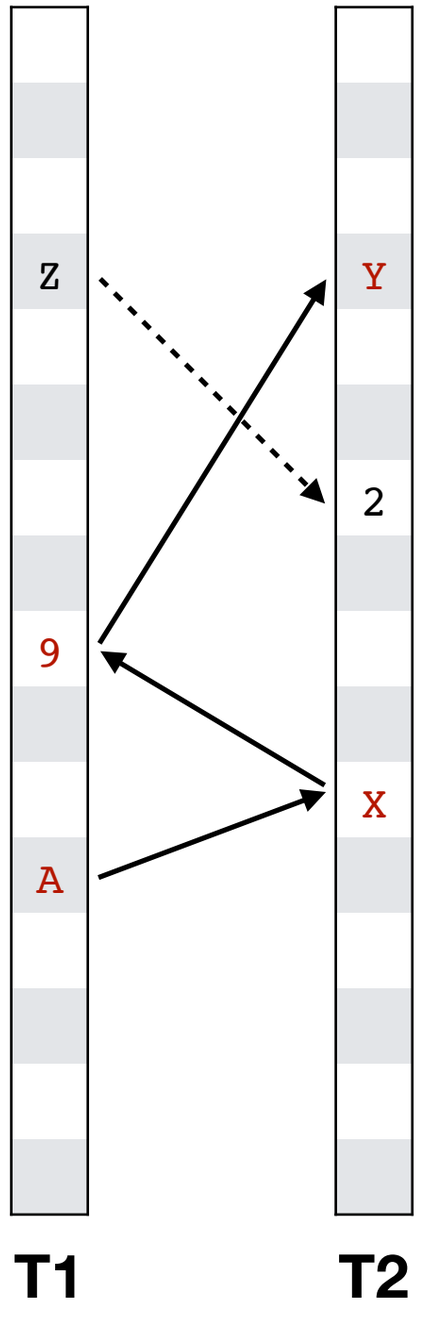
insert("A")

- 大段维护，局部朴素
  - Cuckoo Hash
    - `insert(x)`：
      - 将 `x` 插入 `T1` 的 `h1(x)` 位置
        - 若 `T1 h1(x)` 为空，则插入成功；
        - 若 `T1` h1(x) 不为空，则将该位置上的元素 `y`，移动到 `T2 h2(y)` 处；
          - 若 `T2 h2(y)`为空，则插入成功；
          - 若 `T2 h2(y)`不为空，则将该位置上的元素 `z`，移动到 `T1 h1(z)` 处；
            - 若 `T1 h1(z)` 为空...
            - 反复移动，直到不再有冲突



- 大段维护，局部朴素
  - Cuckoo Hash
    - `insert(x)`：
      - 将 `x` 插入 `T1` 的 `h1(x)` 位置
        - 若 `T1 h1(x)` 为空，则插入成功；
        - 若 `T1` h1(x) 不为空，则将该位置上的元素 `y`，移动到 `T2 h2(y)` 处；
          - 若 `T2 h2(y)`为空，则插入成功；
          - 若 `T2 h2(y)`不为空，则将该位置上的元素 `z`，移动到 `T1 h1(z)` 处；
            - 若 `T1 h1(z)` 为空...
            - 反复移动，直到不再有冲突

- **定义一个移动次数上限**
- **若移动次数达到上限，则重新定义 `Hash` 函数 `h1`，`h2`，并 `rehash`**



`insert("A")`

- 大段维护，局部朴素
  - Cuckoo Hash
    - 优点：
      - exists(x)、get(x) 操作时间复杂度 O(1)
      - insert(x) 操作时间复杂度 O(1)
      - Hash 表存储效率达到 80%
- 操作样例：
  - Cuckoo Hashing Visualization