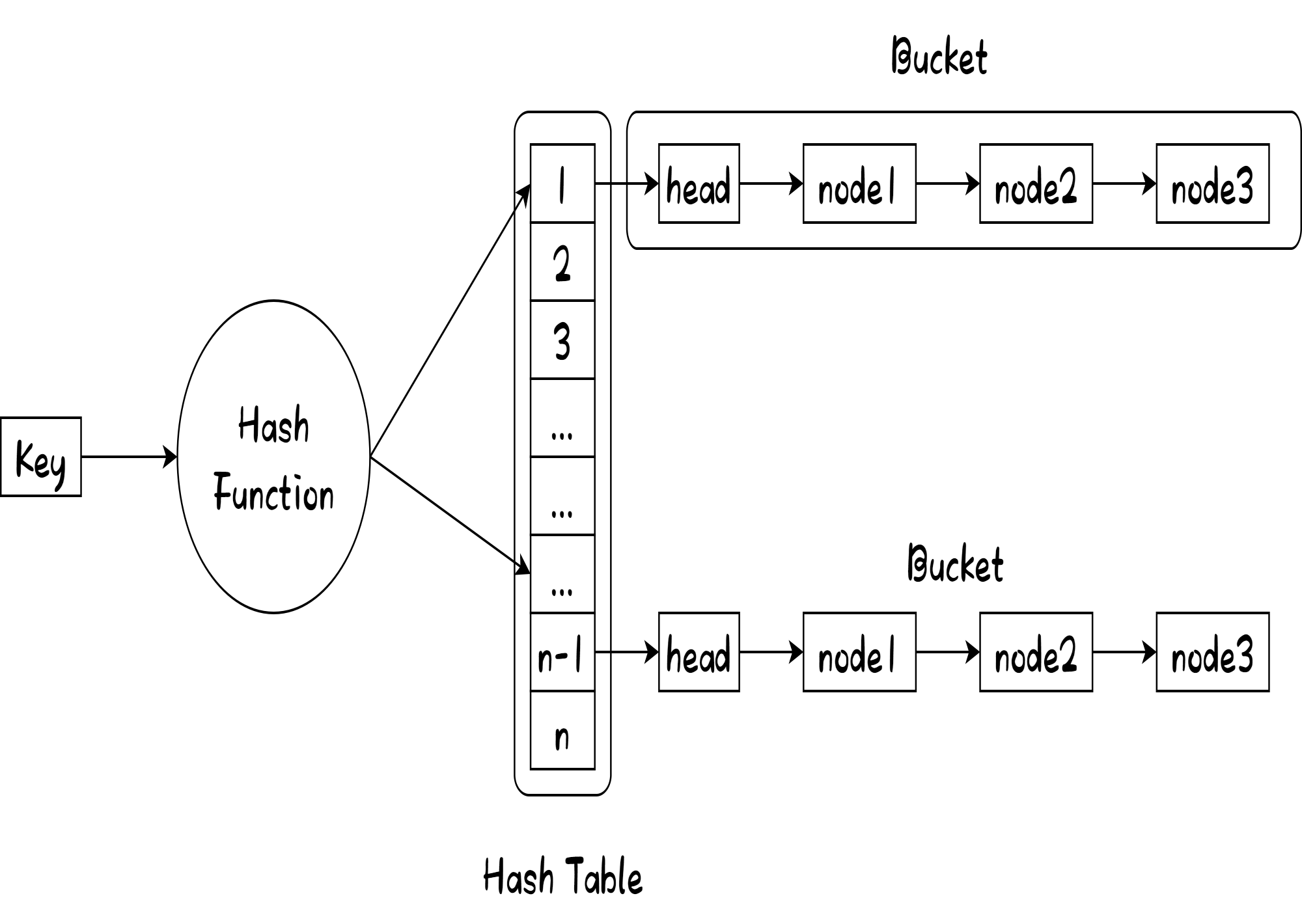
# 可拓展哈希

## 原理

### 所解决的问题

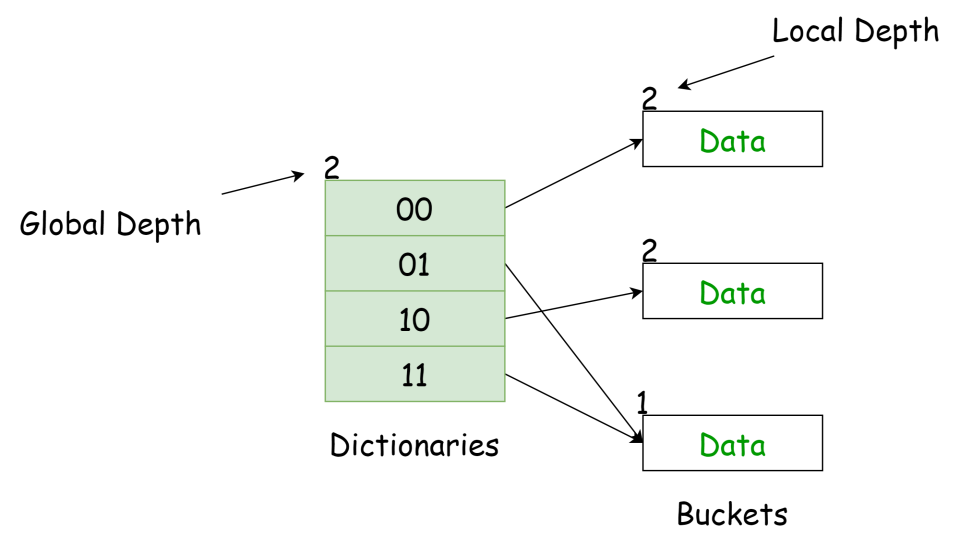
可扩展hash (Extendible Hashing ) 是一种动态 hash 方法. 可扩展 Hash 是从链式 hash 方法延伸的.链式 hash 中, Directories 大小是固定的, 当不同的元素的 key 映射到相同的slot 之后, 相同的Key会放入相同的 Bucket,由于 Bucket 没有大小限制，会随着元素的写入无限增大. 这样的无限扩张会随着元素的写入导致 查找效率越来越低.



### 所提出的方法：可拓展哈希

可拓展Hash 函数会随着数据结构的调整而动态调整.

可扩展Hash的 Directories 与链式 Hash类似, Directories 中的每一个slot存储的是 Bucket的起始地址, 可扩展Hash的一大特点就是它的Directories 会随着元素的写入而改变大小，Bucket的数量会自动增加, 但是 Bucket的大小上限一定是2^global\_depth. 可扩展Hash的Buckets是存储数据 value 的地方, 但是通常还需要存储这个Bucket的大小等信息.



## 示例

假设我们的插入的元素形成Hash的序列为 [16,4,6,22,24,10,31,7,9,20,26]. Bucket Size 大小为 3.

根据给定的键计算它在目录中的索引（index of 函数）示例中使用二进制的最后的X位表示目录的Index.数据的二进制如下：

16- 10000

4- 00100

6- 00110

22- 10110

24- 11000

10- 01010

31- 11111

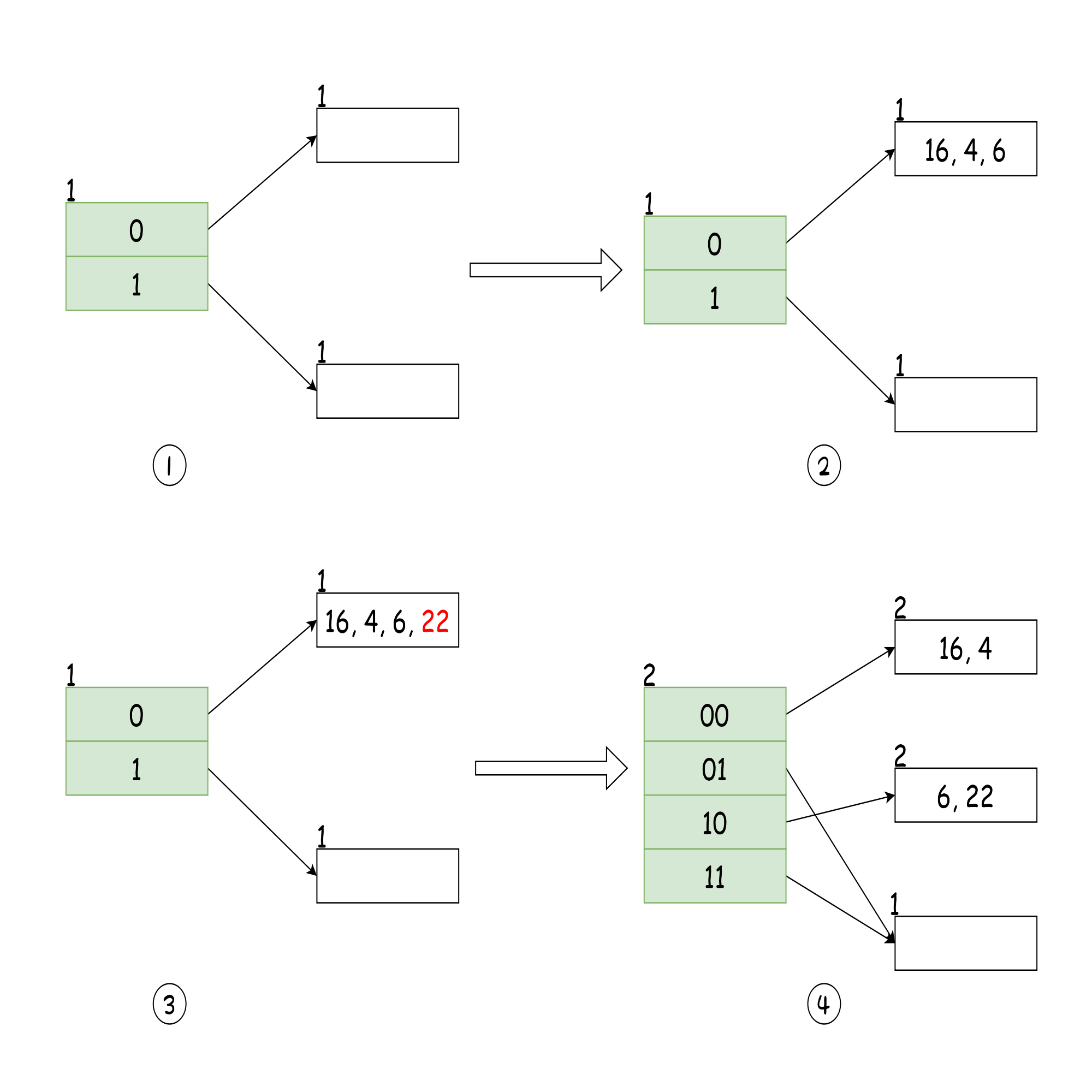
7- 00111

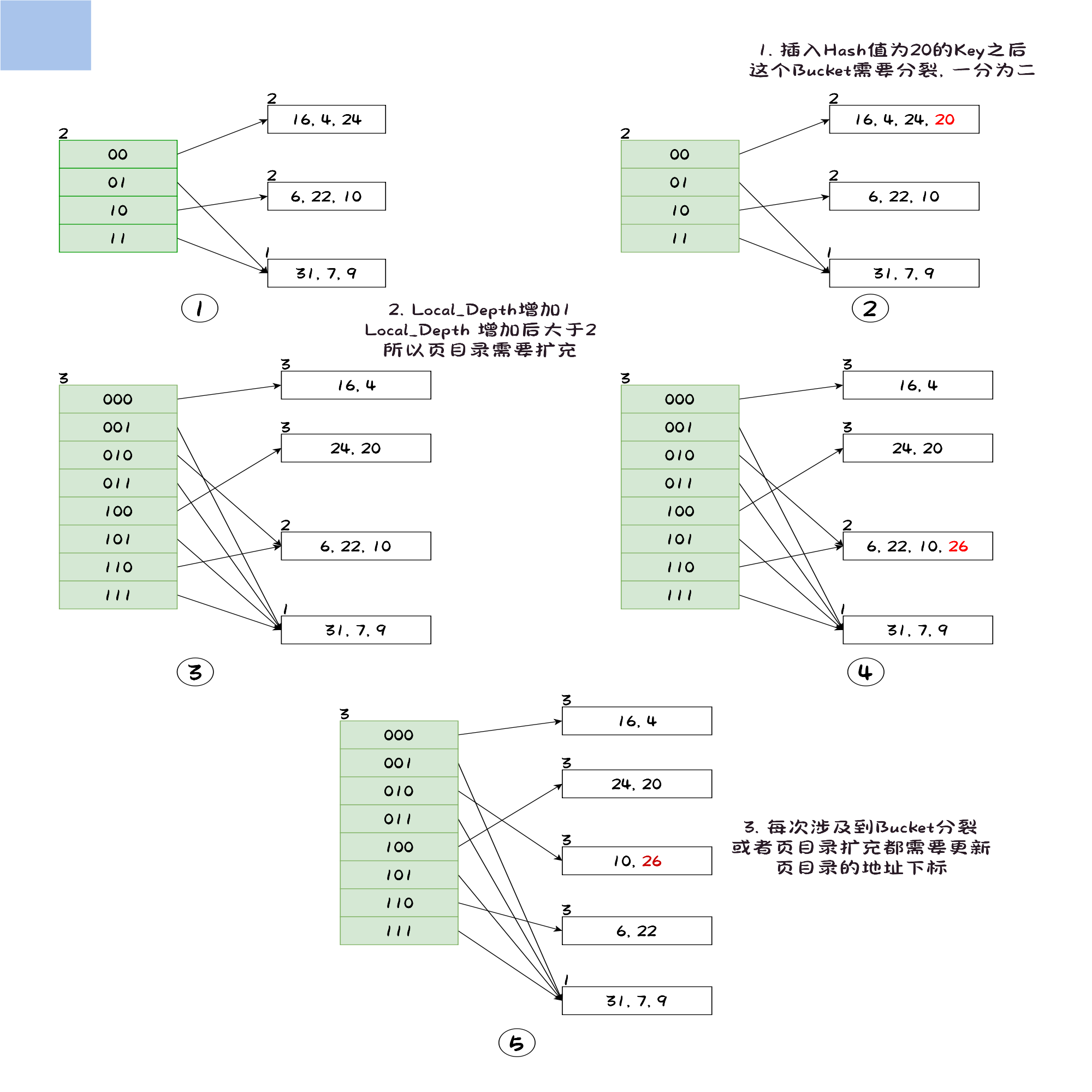
9- 01001

20- 10100

26- 11010

（这些index在后面贴两张图时候也放在边上，具体描述不用放很多，我来讲就行）





## 实现ExtendibleHashTable 类

### Index of函数

该函数用于根据给定的键计算它在目录中的索引。通过计算键的哈希值，使用掩码（mask）与哈希值进行按位与操作来确定桶的索引。掩码大小由全局深度（global\_depth\_）决定。

---（代码贴图）---

### Insert函数

Insert 函数首先检查对应桶是否已经满了。如果桶满了，它会触发“再哈希”（rehashing）过程，通过增加全局深度，扩展目录大小，并重新分配桶中的元素。对于插入操作本身，先检查当前桶是否已经包含该键，如果有则更新值；否则，直接将键值对插入桶中。

---（代码贴图）---

### Find函数

Find 函数查找哈希表中是否存在指定的键，并将对应的值返回。它通过计算键的索引，定位到相应的桶，然后在桶中查找键。

——（）——

### Remove 函数

Remove 函数从哈希表中删除一个指定的键值对。与 Find 类似，它通过计算键的索引来定位桶，并从桶中删除键值对。

——（）——

### GetGlobalDepth 和 GetLocalDepth 函数

这两个函数用于获取哈希表的全局深度和局部深度（桶的深度）。GetGlobalDepth 是对外暴露的接口，而 GetGlobalDepthInternal 和 GetLocalDepthInternal 是内部使用的辅助函数。

——（）——

## 实现Bucket类

### Find函数

Find 函数通过遍历桶中的元素，查找是否存在指定的键，如果找到，则将其值返回。

### Remove函数

Remove 函数通过遍历桶中的元素，查找指定的键并删除对应的键值对。

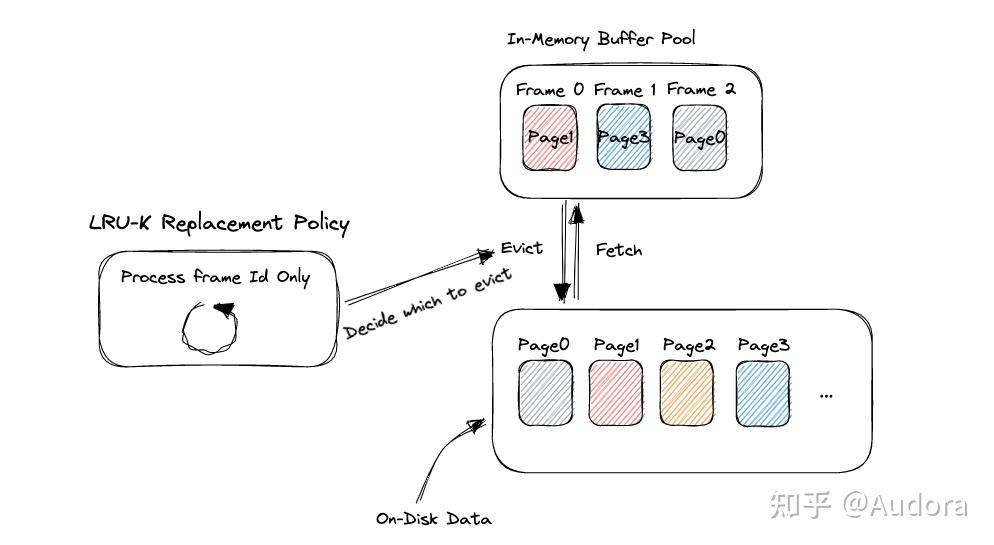
### Insert函数

Insert 函数检查桶是否已满，如果未满，则将键值对插入到桶中（使用 std::list 来存储）。如果插入成功，返回 true，否则返回 false。

# LRU-K页面置换

## 原理

### 应用场景



Buffer Pool就是缓存数据的池子，最大容量为3个page。Disk上面的Data数不胜数，当需要数据时系统会从Buffer Pool检索，如果检索到了，则直接从内存中提出数据，并且通知LRU-K Replacement Policy组件调用了这个page。如果没检索到，就从硬盘中读取一块。如果此时Buffer Pool满了，那就询问LRU-K Replacement Policy组件，索要一个可以驱逐的frame id号，系统再根据这个frame id从内存中把对应的page重新写回硬盘

### 所解决的问题

LRU算法存在的问题是，当存在大量的一次性操作时，会把历史的缓存冲刷掉，而新进入buffer pool的page有可能之后不会再访问了，被冲刷掉的page是之前保留下来的比较“有用”的page，这就是缓存污染问题。

### 所提出的方法

LRU-K的思路是，永远最先驱逐访问次数小于K次的page。页面必须被访问 k 次才会被视为 "最近访问" 的概念。它通过维护两个队列来实现这一目标：history\_queue 和 last\_accesed\_queue。

## 示例

见附图，主要我讲即可

## 实现

Evict 函数用于驱逐一个页面框架。如果 history\_queue 和 last\_accesed\_queue 都为空，返回 false 表示无法驱逐。如果历史队列中存在页面框架，并且它符合可驱逐的条件（is\_evictable 中标记为可驱逐），则驱逐该页面框架；否则，尝试从最近访问队列中驱逐。驱逐后，移除该页面框架的所有记录。

RecordAccess 函数记录某个页面框架的访问事件。它首先增加时间戳，然后检查页面框架是否已经在最近访问队列中。如果在最近访问队列中，更新其访问时间戳；如果不在历史队列中，则将其加入历史队列。

SetEvictable 函数设置一个页面框架是否可驱逐。传入的 set\_evictable 参数决定是标记该页面为可驱逐（true），还是将其从可驱逐页面框架集合中移除（false）。

Remove 函数用于从所有队列中移除页面框架。无论该页面框架是否可驱逐，都将其从历史队列、最近访问队列和 is\_evictable 集合中删除。

Size 函数返回当前可驱逐页面框架的数量。即 is\_evictable 集合中页面框架的数量