# Hash、树

七月算法 邹博

2015年4月16日

# 一种NoSQL数据库 - Riak

从事过建筑的人都知道，有种用来加强混凝土的钢条，称作钢筋。正如 Riak（读作”Ree-ahck”）一样，钢筋不会单独使用，它往往用于互相作用的各个部分，以使整个系统持久耐用。于是，系统中的每个组件都廉价又不起眼，但只要使用得当，就可构建起足够简单且牢固的基础结构。

Riak 是一种分布式的键-值（key-value）数据库。其中，值可以是任何类型的数据，如普通文本、JSON、XML、图片，甚至视频片段；而所有这些都可以通过普通的 HTTP 接口访问。你有什么，Riak 就能存什么。

容错是 Riak 的另一特性。服务器可在任何时刻启动或者停止，而不会引起任何单点故障。不管是增加或者移除服务器，甚至有节点崩溃（谁都不想这样），集群依然可以持续忙碌地运行。Riak 让你不再整夜无眠担心集群，某个节点失效不再是紧急事件，完全可以等到第二天早晨处理。Riak 的核心开发者 Justin Sheehy 曾提到：“（Riak 团队）非常注重可写入性……为的是可以回家睡觉。”

然而万事都有利弊取舍，Riak 的灵活性自有其代价。对于自由定义的（ad hoc）查询，Riak 缺乏有力支持；而键-值存储的设计，使得数据值无法相互连接（即，Riak 没有外键）。Riak 试图将这些问题各个击破，我们会在后面几天读到相关内容。

# 节选自《七周七数据库》

本章会探究 Riak 如何存储并检索数据，以及如何使用链接将数据捆绑。然后会探索本书中大量使用的数据检索概念：映射-归约（mapreduce）。此外，也会看到 Riak 如何把节点服务器组成集群，并且在节点发生故障时处理请求。最后，我们看看 Riak 如何处理因写入分布式服务器而产生的冲突，以及基本服务器的一些扩展。

# 字典

- 字典，又称符号表（symbol table）、关联数组（associative array）或者映射（map），是一种用于保存键值对（key-value pair）的抽象数据结构。
- 在字典中，一个键（key）可以和一个值（value）进行关联（或者说将键映射为值），这些关联的键和值就被称为键值对。
- 字典中的每个键都是独一无二的，程序可以在字典中根据键查找与之关联的值，或者通过键来更新值，又或者根据键来删除整个键值对，等等。
- 字典经常作为一种数据结构内置在很多高级编程语言里面，但Redis所使用的C语言并没有内置这种数据结构，因此Redis构建了自己的字典实现。

# 字典

- 字典在Redis中的应用相当广泛，比如Redis的数据库就是使用字典来作为底层实现的，对数据库的增、删、查、改操作也是构建在对字典的操作之上的。
- 除了用来表示数据库之外，字典还是哈希键的底层实现之一：当一个哈希键包含的键值对比较多，又或者键值对中的元素都是比较长的字符串时，Redis就会使用字典作为哈希键的底层实现。

# 字典的实现

- Redis的字典使用哈希表作为底层实现，一个哈希表里面可以有多个哈希表节点，而每个哈希表节点就保存了字典中的一个键值对。

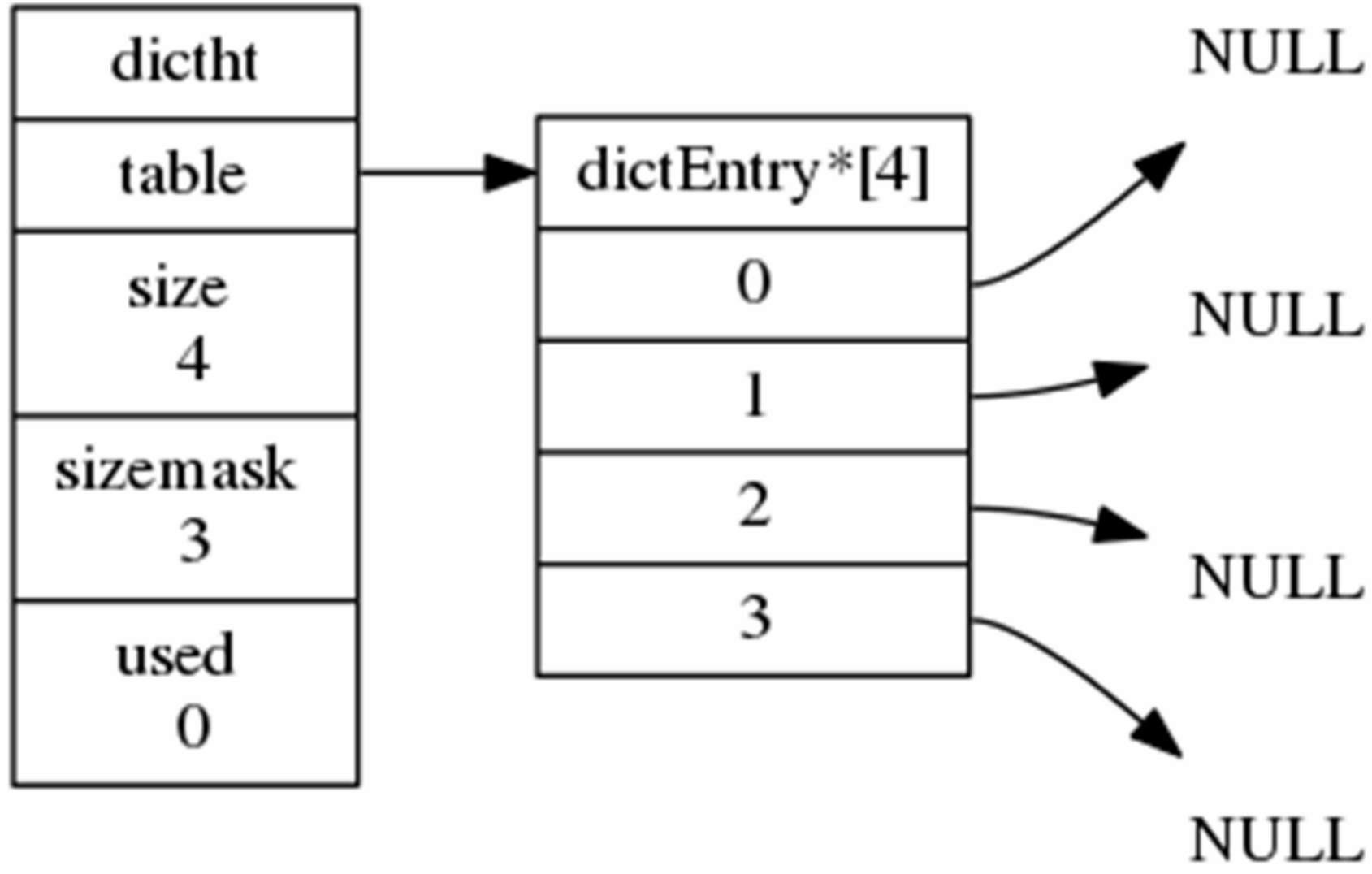
# 哈希表

```
typedef struct dictht {

    // 哈希表数组
    dictEntry **table;

    // 哈希表大小
    unsigned long size;

    // 哈希表大小掩码，用于计算索引值
    // 总是等于 size - 1
    unsigned long sizemask;

    // 该哈希表已有节点的数量
    unsigned long used;

} dictht;
```

# 哈希表

- table属性是一个数组，数组中的每个元素都是一个指向dictEntry结构的指针，每个dictEntry结构保存着一个键值对。
- size属性记录了哈希表的大小，也即是table数组的大小，而used属性则记录了哈希表目前已有节点（键值对）的数量。
- sizemask属性的值总是等于size-1，这个属性和哈希值一起决定一个键应该被放到table数组的哪个索引上面。

# 一个大小为 4 的空哈希表



# 哈希表节点

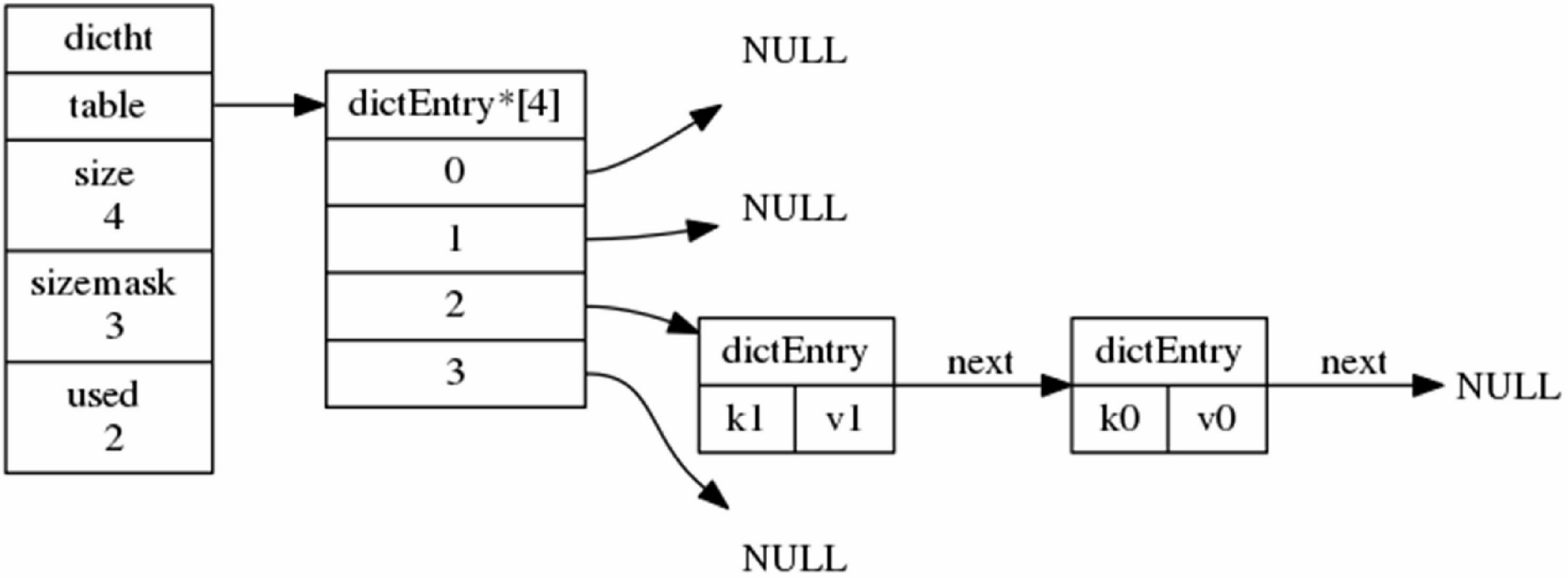
- 哈希表节点使用dictEntry结构表示，每个dictEntry结构都保存着一个键值对。

```
typedef struct dictEntry {

    // 键
    void *key;

    // 值
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
    } v;

    // 指向下个哈希表节点，形成链表
    struct dictEntry *next;

} dictEntry;
```

# 哈希表节点

- key属性保存着键值对中的键，而val属性则保存着键值对中的值，其中键值对的值可以是一个指针，或者是一个uint64_t整数，又或者是一个int64_t整数。
- next属性是指向另一个哈希表节点的指针，这个指针可以将多个哈希值相同的键值对连接在一次，以此来解决键冲突（collision）的问题。

# 两个索引值相同的键k1和k0



# 字典

```
typedef struct dict {

    // 类型特定函数
    dictType *type;

    // 私有数据
    void *privdata;

    // 哈希表
    dictht ht[2];

    // rehash 索引
    // 当 rehash 不在进行时，值为 -1
    int rehashidx;

} dict;
```

# 字典的多态

- type属性和privdata属性是针对不同类型的键值对，为创建多态字典而设置的：
- type属性是一个指向dictType结构的指针，每个dictType结构保存了一簇用于操作特定类型键值对的函数，Redis会为用途不同的字典设置不同的类型特定函数。
- privdata属性则保存了需要传给那些类型特定函数的可选参数。

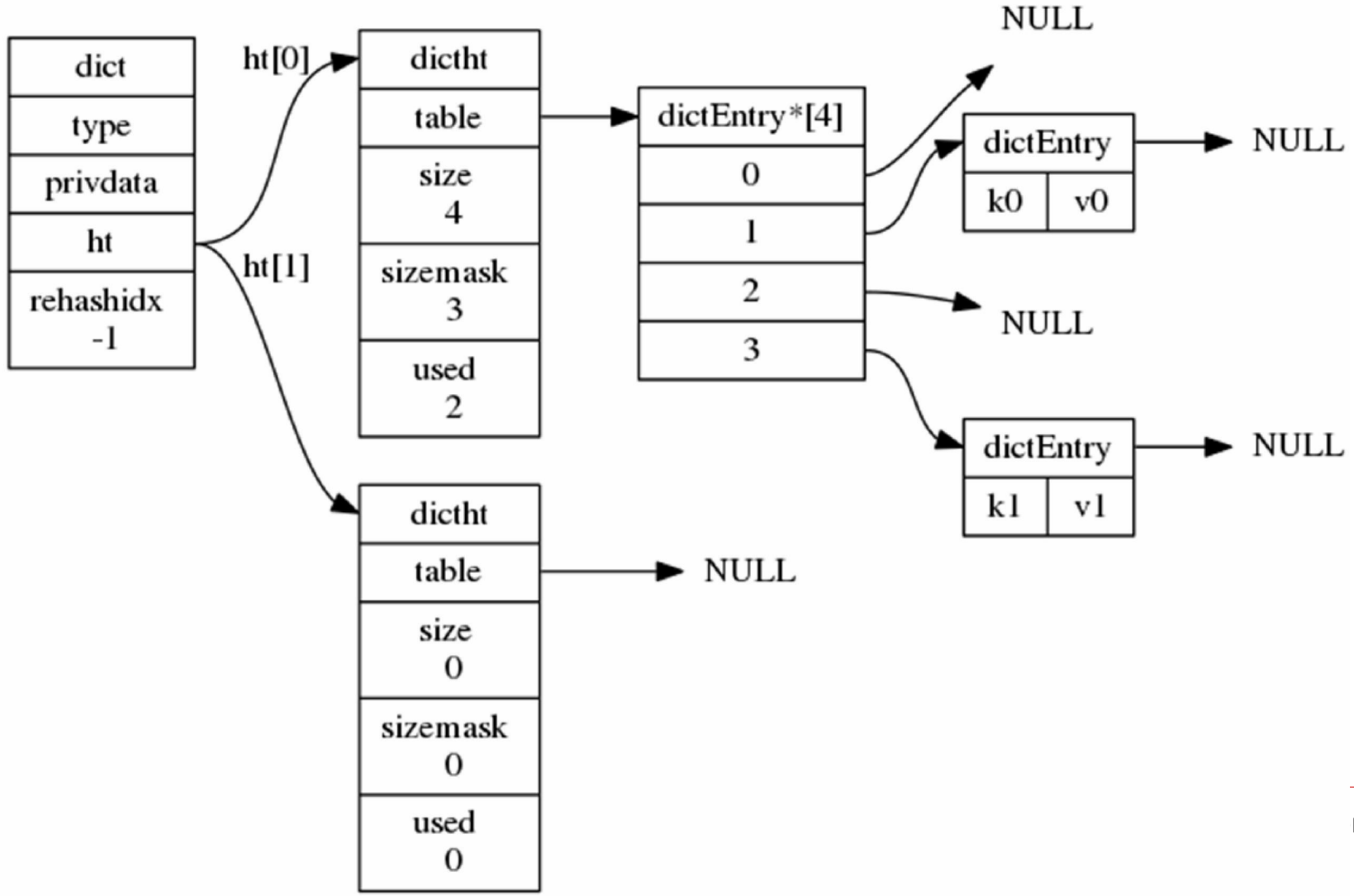
# 字典的多态

```
typedef struct dictType {

    // 计算哈希值的函数
    unsigned int (*hashFunction)(const void *key);

    // 复制键的函数
    void *(*keyDup)(void *privdata, const void *key);

    // 复制值的函数
    void *(*valDup)(void *privdata, const void *obj);

    // 对比键的函数
    int (*keyCompare)(void *privdata, const void *key1, const void *key2);

    // 销毁键的函数
    void (*keyDestructor)(void *privdata, void *key);

    // 销毁值的函数
    void (*valDestructor)(void *privdata, void *obj);

} dictType;
```

# Rehash

- ht属性是一个包含两个项的数组，数组中的每个项都是一个dictht哈希表，一般情况下，字典只使用ht[0]哈希表，ht[1]哈希表只会在对ht[0]哈希表进行rehash时使用。
- 除了ht[1]之外，另一个和rehash有关的属性就是rehashidx：它记录了rehash目前的进度，如果目前没有在进行rehash，那么它的值为-1。

# 普通状态下的字典


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
2
dictEntry*[4]
0
1
2
3
NULL
dictEntry
k0
v0
NULL
NULL
dictEntry
k1
v1
NULL
dictht
table
NULL
size
0
sizemask
0
used
0

# 哈希算法

- 当要将一个新的键值对添加到字典里面时，程序需要先根据键值对的键计算出哈希值和索引值，然后再根据索引值，将包含新键值对的哈希表节点放到哈希表数组的指定索引上面。
- 当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis使用MurmurHash2算法来计算键的哈希值。

# Redis计算哈希值和索引值的方法

```
# 使用字典设置的哈希函数，计算键 key 的哈希值
hash = dict->type->hashFunction(key);

# 使用哈希表的 sizemask 属性和哈希值，计算出索引值
# 根据情况不同， ht[x] 可以是 ht[0] 或者 ht[1]
index = hash & dict->ht[x].sizemask;
```

# MurmurHash

- MurmurHash是一种非加密型哈希函数，适用于一般的哈希检索操作。由AustinAppleby在2008年发明，并出现了多个变种，都已经发布到了公有领域(public domain)。与其它流行的哈希函数相比，对于规律性较强的key，Murmur Hash的随机分布特征表现更良好。
- 当前的版本是MurmurHash3，能够产生出32-bit或128-bit哈希值。
- 较早的MurmurHash2能产生32-bit或64-bit哈希值。对于大端存储和强制对齐的硬件环境有一个较慢的MurmurHash2可以用。MurmurHash2A变种增加了Merkle–Damgård构造，所以能够以增量方式调用。有两个变种产生64-bit哈希值：MurmurHash64A，为64位处理器做了优化；MurmurHash64B，为32位处理器做了优化。MurmurHash2-160用于产生160-bit哈希值，而MurmurHash1已经不再使用。

# MurmurHash

```
Murmur3_32(key, len, seed)
    c1 ← 0xcc9e2d51
    c2 ← 0x1b873593
    r1 ← 15
    r2 ← 13
    m ← 5
    n ← 0xe6546b64

    hash ← seed

    for each fourByteChunk of key
        k ← fourByteChunk

        k ← k * c1
        k ← (k << r1) OR (k >> (32-r1))
        k ← k * c2

        hash ← hash XOR k
        hash ← (hash << r2) OR (hash >> (32-r2))
        hash ← hash * m + n

    with any remainingBytesInKey
        remainingBytes ← SwapEndianOrderOf(remainingBytesInKey)
        remainingBytes ← remainingBytes * c1
        remainingBytes ← (remainingBytes << r1) OR (remainingBytes >> (32 - r1))
        remainingBytes ← remainingBytes * c2

        hash ← hash XOR remainingBytes

    hash ← hash XOR len

    hash ← hash XOR (hash >> 16)
    hash ← hash * 0x85ebca6b
    hash ← hash XOR (hash >> 13)
    hash ← hash * 0xc2b2ae35
    hash ← hash XOR (hash >> 16)
```

# djb2

- this algorithm (k=33) was first reported by dan bernstein many years ago in comp.lang.c. another version of this algorithm (now favored by bernstein) uses xor: hash(i) = hash(i - 1) * 33 ^ str[i]; the magic of number 33 (why it works better than many other constants, prime or not) has never been adequately explained.
- 这个算法(k=33)是多年前首先由dan bernstein 在comp.lang.c中提出的，这个算法的另外一个版本（bernstein的贡献）是使用异或方式：hash(i) = hash(i - 1) * 33 ^ str[i];魔数33尚未得到足够的解释（为何它能够比其他很多素数的效果更好）

# djb2 Code

```
unsigned long
hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
}
```

# sdbm Code(65599)

```
static unsigned long
sdbm(str)
unsigned char *str;
{
    unsigned long hash = 0;
    int c;

    while (c = *str++)
        hash = c + (hash << 6) + (hash << 16) - hash;

    return hash;
}
```

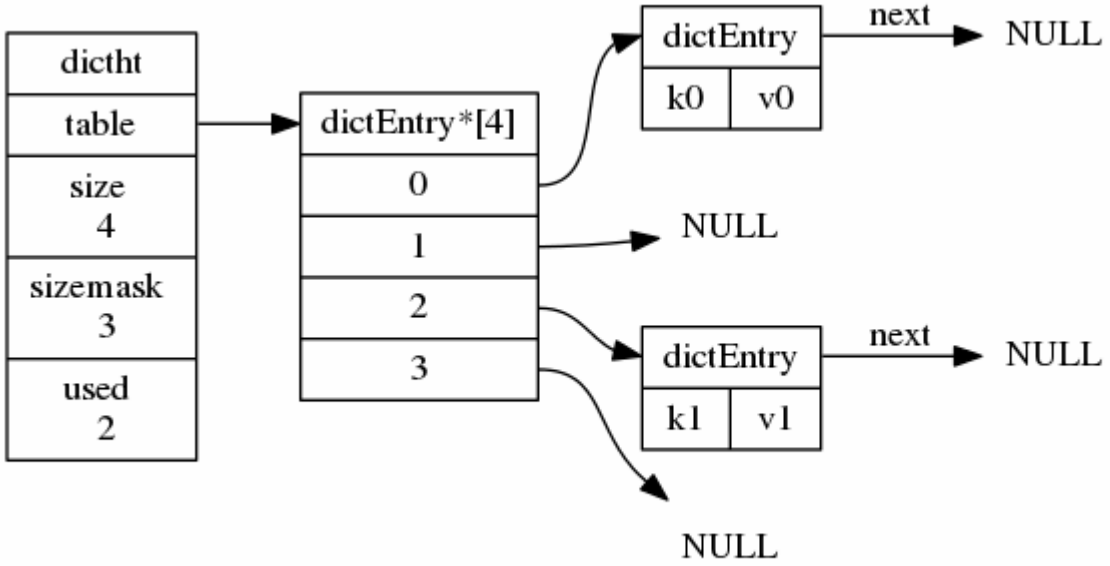
# sdbm的说明

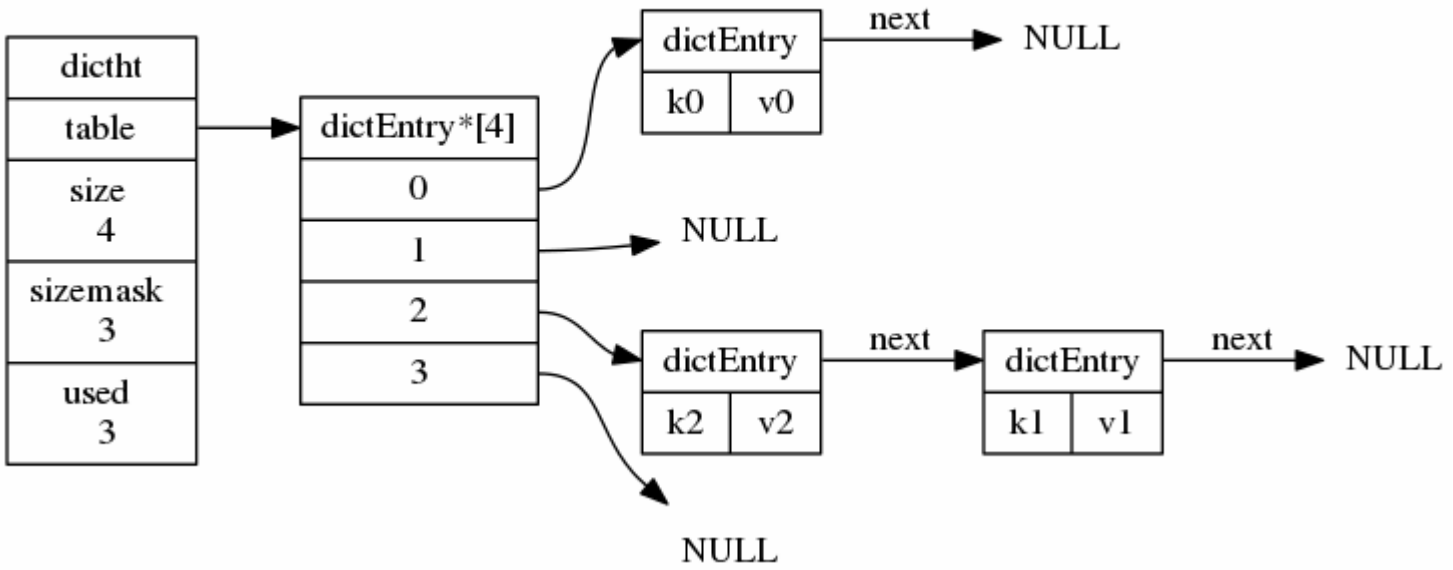
- This algorithm was created for sdbm (a public-domain reimplementation of ndbm) database library. it was found to do well in scrambling bits, causing better distribution of the keys and fewer splits.
- It also happens to be a good general hashing function with good distribution. the actual function is hash(i) = hash(i - 1) * 65599 + str[i]; which is the faster version used in gawk. The magic constant 65599 was picked out of thin air while experimenting with different constants, and turns out to be a prime. this is one of the algorithms used in berkeley db and elsewhere.

# 哈希冲突

- 当有两个或以上数量的键被分配到了哈希表数组的同一个索引上面时，我们称这些键发生了冲突（collision）。
- Redis的哈希表使用链地址法（separate chaining）来解决键冲突：每个哈希表节点都有一个next指针，多个哈希表节点可以用next指针构成一个单向链表，被分配到同一个索引上的多个节点可以用这个单向链表连接起来，这就解决了键冲突的问题。

# 冲突


dictht
table
size
4
sizemask
3
used
2
dictEntry*[4]
0
1
2
3
dictEntry
k0
v0
next
NULL
NULL
dictEntry
k1
v1
next
NULL
NULL

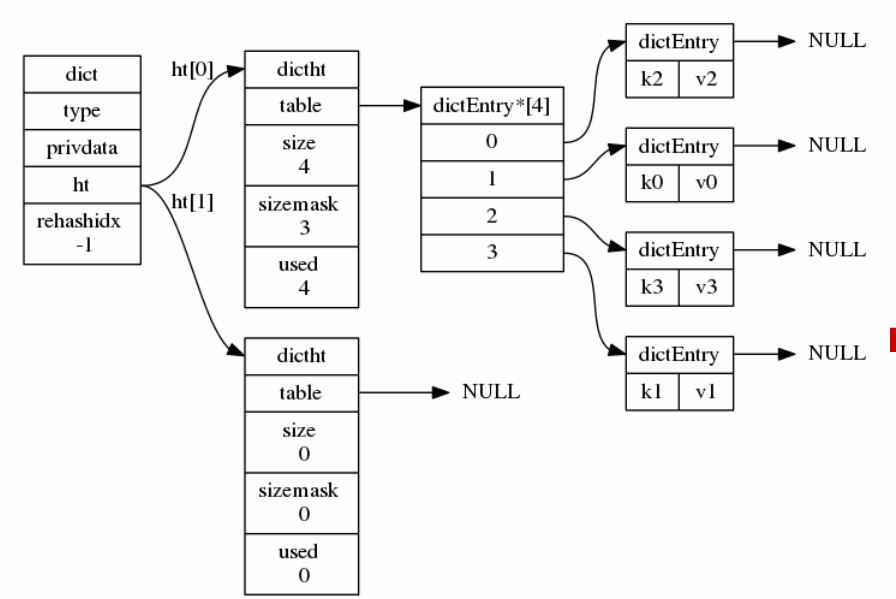

dictht
table
size
4
sizemask
3
used
3
dictEntry*[4]
0
1
2
3
dictEntry
k0
v0
next
NULL
NULL
dictEntry
k2
v2
next
dictEntry
k1
v1
next
NULL
NULL

# Rehash

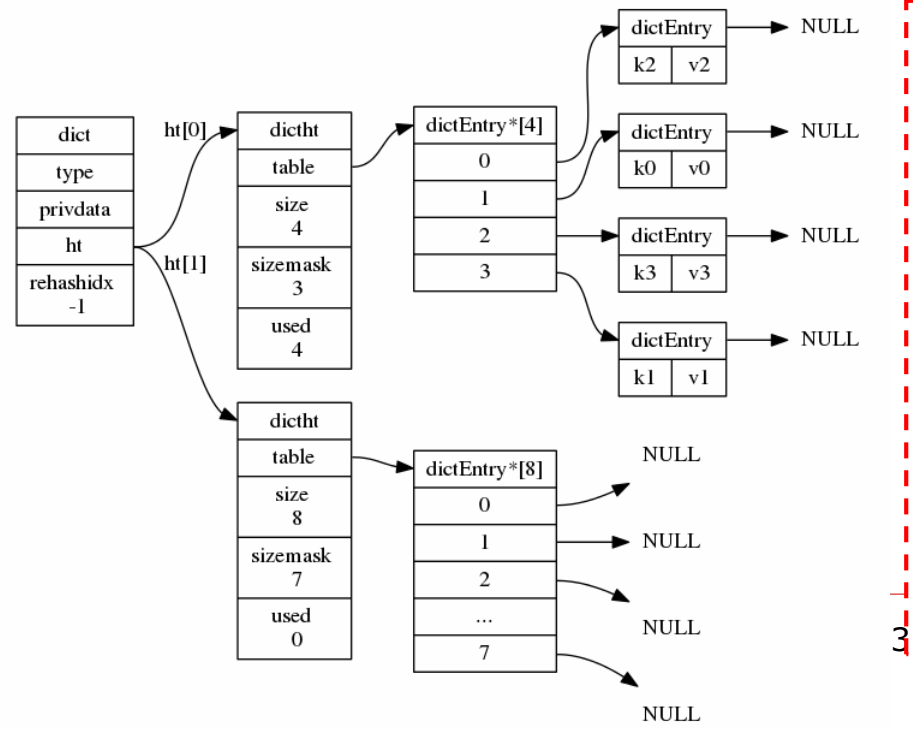
- rehash是在hash table的大小不能满足需求，造成过多hash碰撞后需要进行的扩容hash table的操作，其实通常的做法确实是建立一个额外的hash table，将原来的hash table中的数据在新的数据中进行重新输入，从而生成新的hash表。
- 优先使用0号hash table，当空间不足时会调用dictExpand来扩展hash table，此时准备1号hash table用于增量的rehash使用。rehash完成后把0号释放，1号保存到0号。
- rehashidx是下一个需要rehash的项在ht[0]中的索引，不需要rehash时置为-1。也就是说-1时，表示不进行rehash。

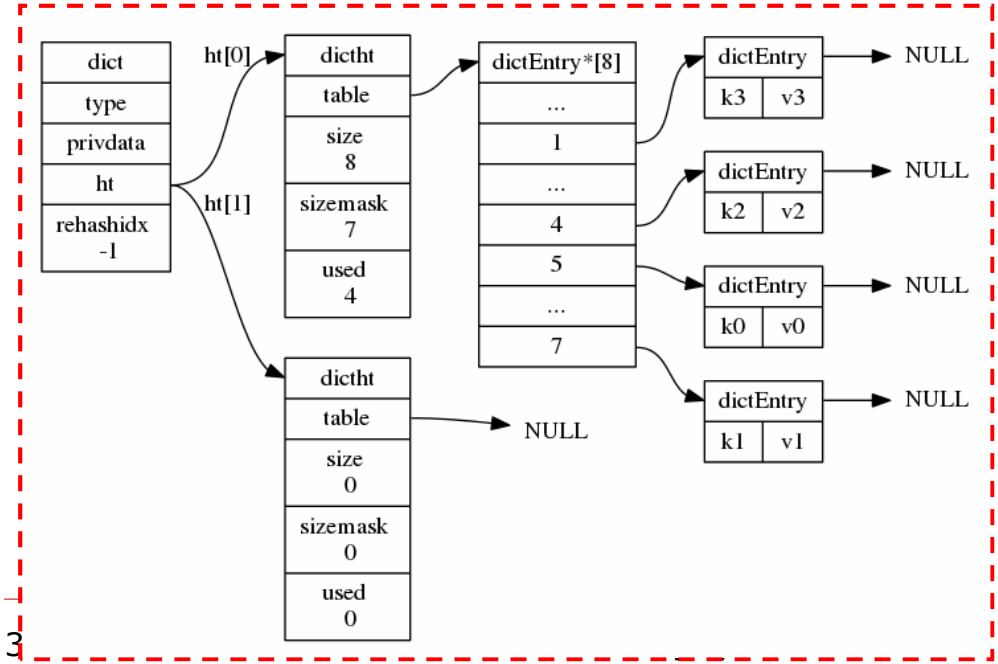
# Rehash

- 随着操作的不断执行，哈希表保存的键值对会逐渐地增多或者减少，为了让哈希表的负载因子（load factor）维持在一个合理的范围之内，当哈希表保存的键值对数量太多或者太少时，程序需要对哈希表的大小进行相应的扩展或者收缩。
- 扩展和收缩哈希表的工作可以通过执行rehash（重新散列）操作来完成，Redis对字典的哈希表执行rehash的步骤如下：
  - 为字典的ht[1]哈希表分配空间，这个哈希表的空间大小取决于要执行的操作，以及ht[0]当前包含的键值对数量（也即是ht[0].used属性的值）：
    - 如果执行的是扩展操作，那么ht[1]的大小为大于等于ht[0].used*2的2的n次方幂的最小值；
    - 如果执行的是收缩操作，那么ht[1]的大小为大于等于ht[0].used的2的n次方幂的最小值。
  - 将保存在ht[0]中的所有键值对rehash到ht[1]上面：rehash指的是重新计算键的哈希值和索引值，然后将键值对放置到ht[1]哈希表的指定位置上。
  - 当ht[0]包含的所有键值对都迁移到了ht[1]之后（ht[0]变为空表），释放ht[0]，将ht[1]设置为ht[0]，并在ht[1]新创建一个空白哈希表，为下一次rehash做准备。


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
4
dictEntry*[4]
0
1
2
3
dictEntry
k2
v2
dictEntry
k0
v0
dictEntry
k3
v3
dictEntry
k1
v1
NULL
NULL
NULL
NULL
dictht
table
size
0
sizemask
0
used
0
NULL

# Rehash


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
4
dictEntry*[4]
0
1
2
3
dictEntry
k2
v2
dictEntry
k0
v0
dictEntry
k3
v3
dictEntry
k1
v1
NULL
NULL
NULL
NULL
dictht
table
size
8
sizemask
7
used
0
dictEntry*[8]
0
1
2
...
7
NULL
NULL
NULL
NULL


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
8
sizemask
7
used
4
dictEntry*[8]
...
1
...
4
5
...
7
dictEntry
k3
v3
dictEntry
k2
v2
dictEntry
k0
v0
dictEntry
k1
v1
NULL
NULL
NULL
NULL
dictht
table
size
0
sizemask
0
used
0
NULL

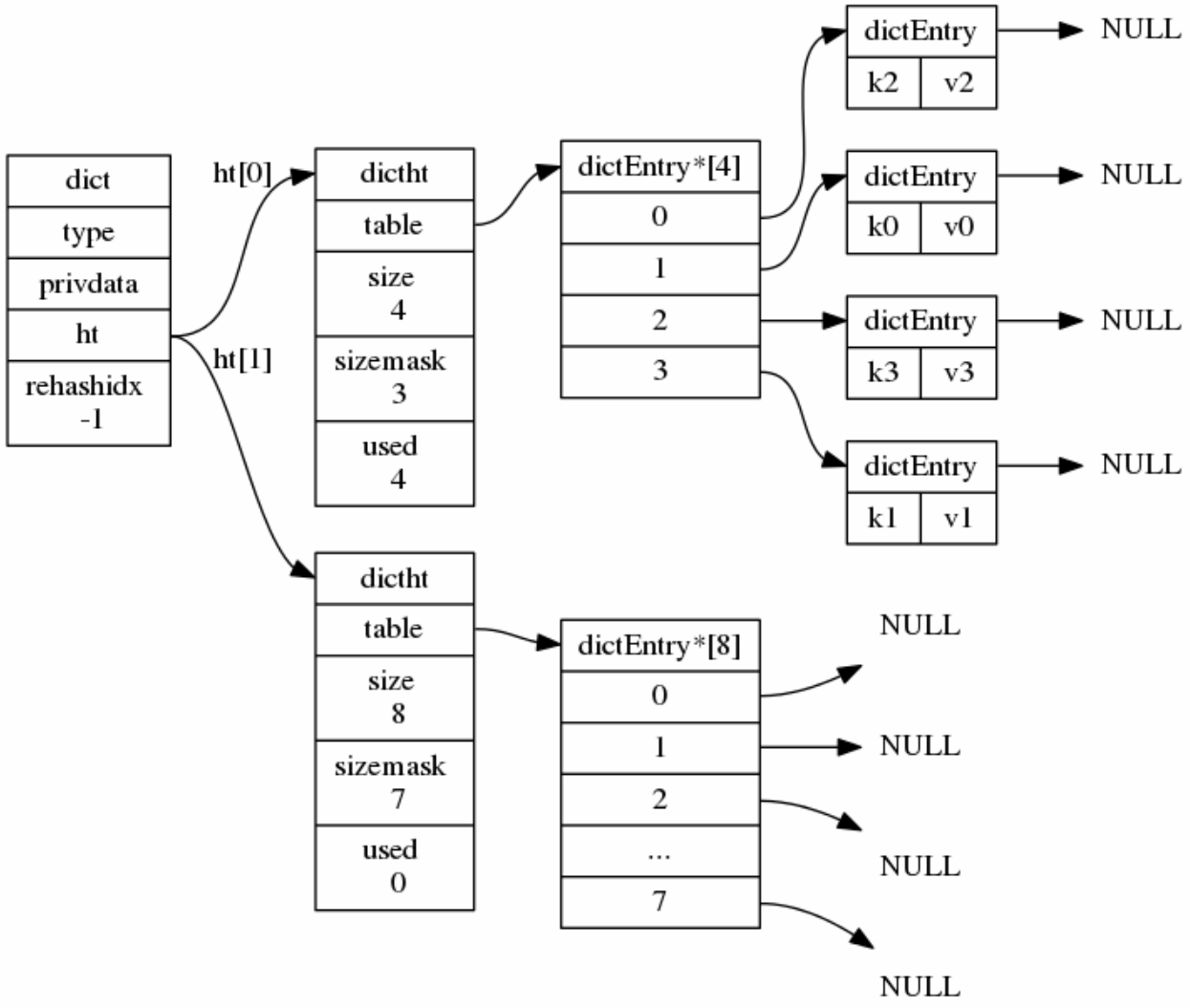
# 渐进式rehash

- 扩展或收缩哈希表需要将ht[0]里面的所有键值对rehash到ht[1]里面，但是，这个rehash动作并不是一次性、集中式地完成的，而是分多次、渐进式地完成的。
- 这样做的原因在于，如果ht[0]里只保存着四个键值对，那么服务器可以在瞬间就将这些键值对全部rehash到ht[1]；但是，如果哈希表里保存的键值对数量不是四个，而是四百万、四千万甚至四亿个键值对，那么要一次性将这些键值对全部rehash到ht[1]的话，庞大的计算量可能会导致服务器在一段时间内停止服务。
- 因此，为了避免rehash对服务器性能造成影响，服务器不是一次性将ht[0]里面的所有键值对全部rehash到ht[1]，而是分多次、渐进式地将ht[0]里面的键值对慢慢地rehash到ht[1]。

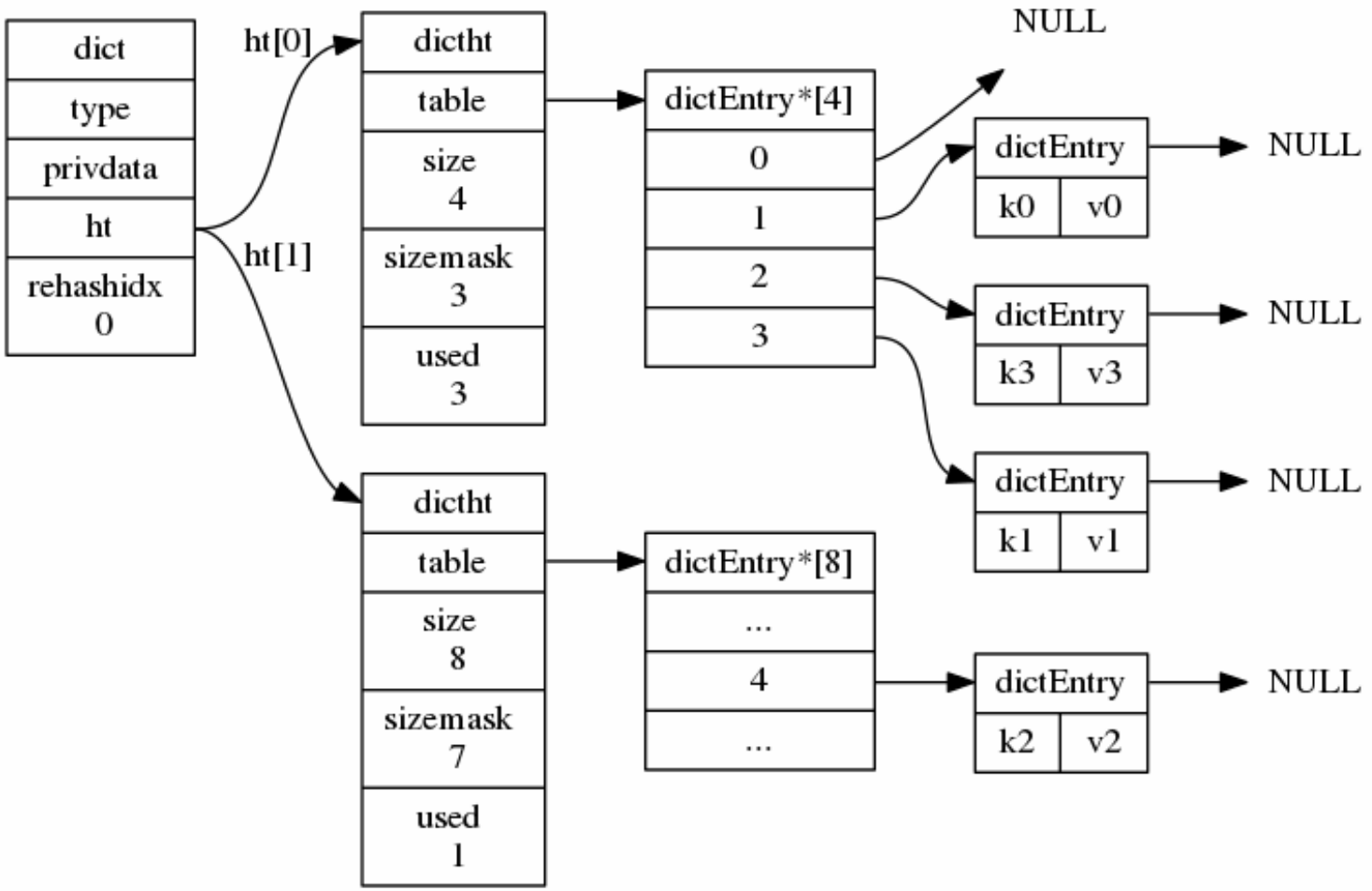
# 哈希表渐进式rehash的详细步骤

- 1.为ht[1]分配空间，让字典同时持有ht[0]和ht[1]两个哈希表。
- 2.在字典中维持一个索引计数器变量rehashidx，并将它的值设置为0，表示rehash工作正式开始。
- 3.在rehash进行期间，每次对字典执行添加、删除、查找或者更新操作时，程序除了执行指定的操作以外，还会顺带将ht[0]哈希表在rehashidx索引上的所有键值对rehash到ht[1]，当rehash工作完成之后，程序将rehashidx属性的值增一。
- 4.随着字典操作的不断执行，最终在某个时间点上，ht[0]的所有键值对都会被rehash至ht[1]，这时程序将rehashidx属性的值设为-1，表示rehash操作已完成。
- 渐进式rehash的好处在于它采取分而治之的方式，将rehash键值对所需的计算工作均摊到对字典的每个添加、删除、查找和更新操作上，从而避免了集中式rehash而带来的庞大计算量。

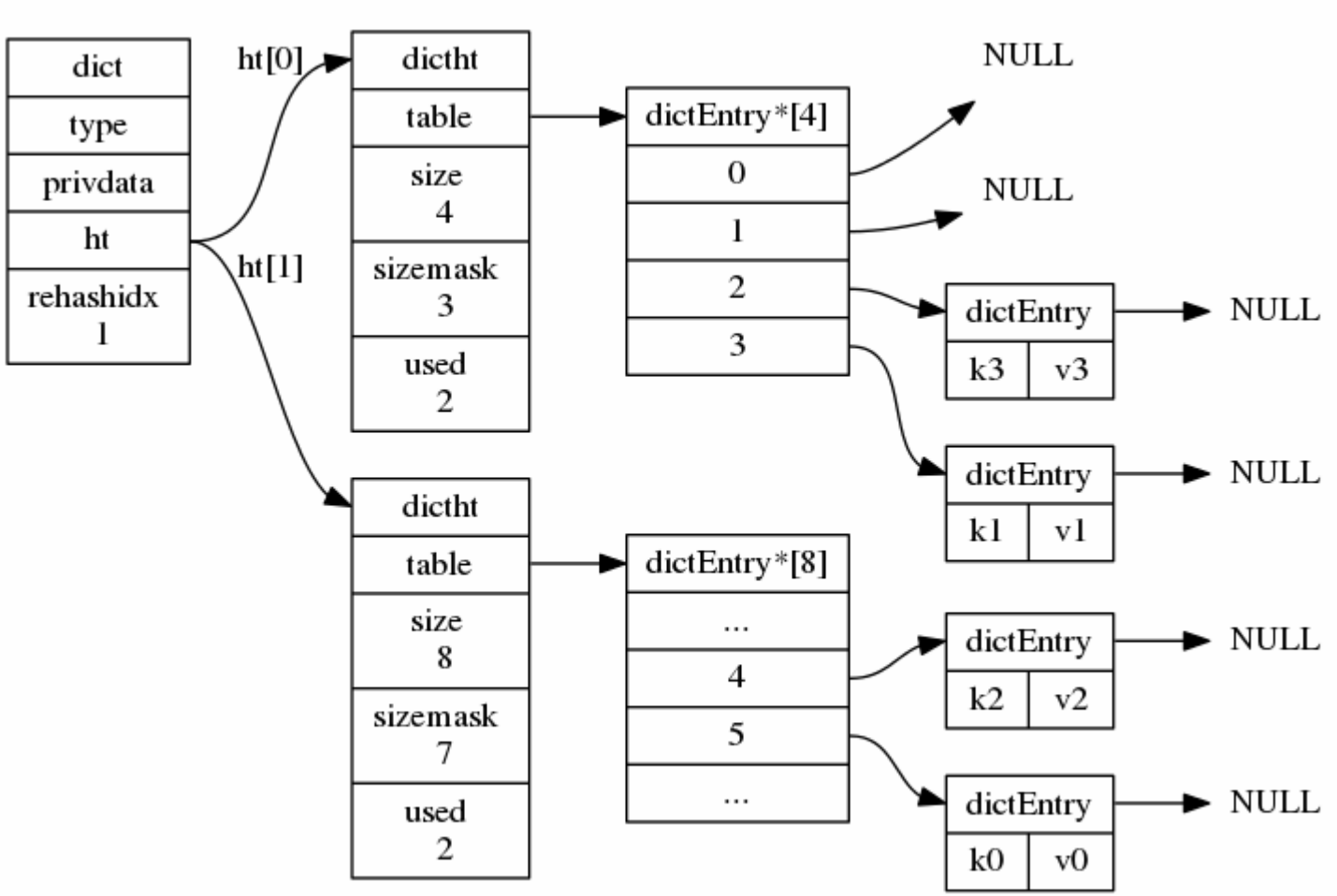
# 准备开始Rehash


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
4
dictEntry*[4]
0
1
2
3
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL
dictEntry
k3
v3
NULL
dictEntry
k1
v1
NULL
dictht
table
size
8
sizemask
7
used
0
dictEntry*[8]
0
1
2
...
7
NULL
NULL
NULL
NULL

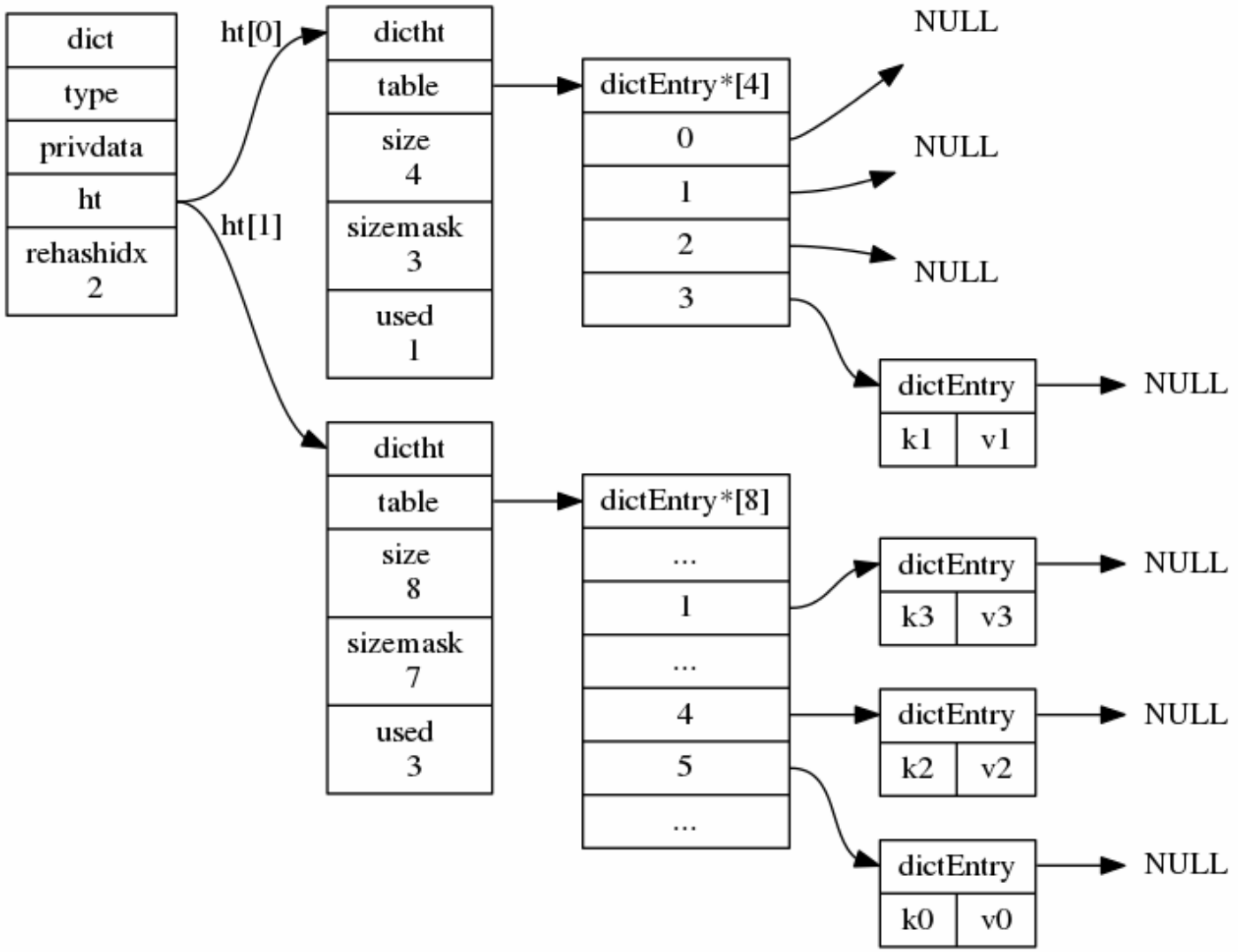
# 索引0上的键值对


dict
type
privdata
ht
rehashidx
0
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
3
dictEntry*[4]
0
1
2
3
NULL
dictEntry
k0
v0
NULL
dictEntry
k3
v3
NULL
dictEntry
k1
v1
NULL
dictht
table
size
8
sizemask
7
used
1
dictEntry*[8]
...
4
...
dictEntry
k2
v2
NULL

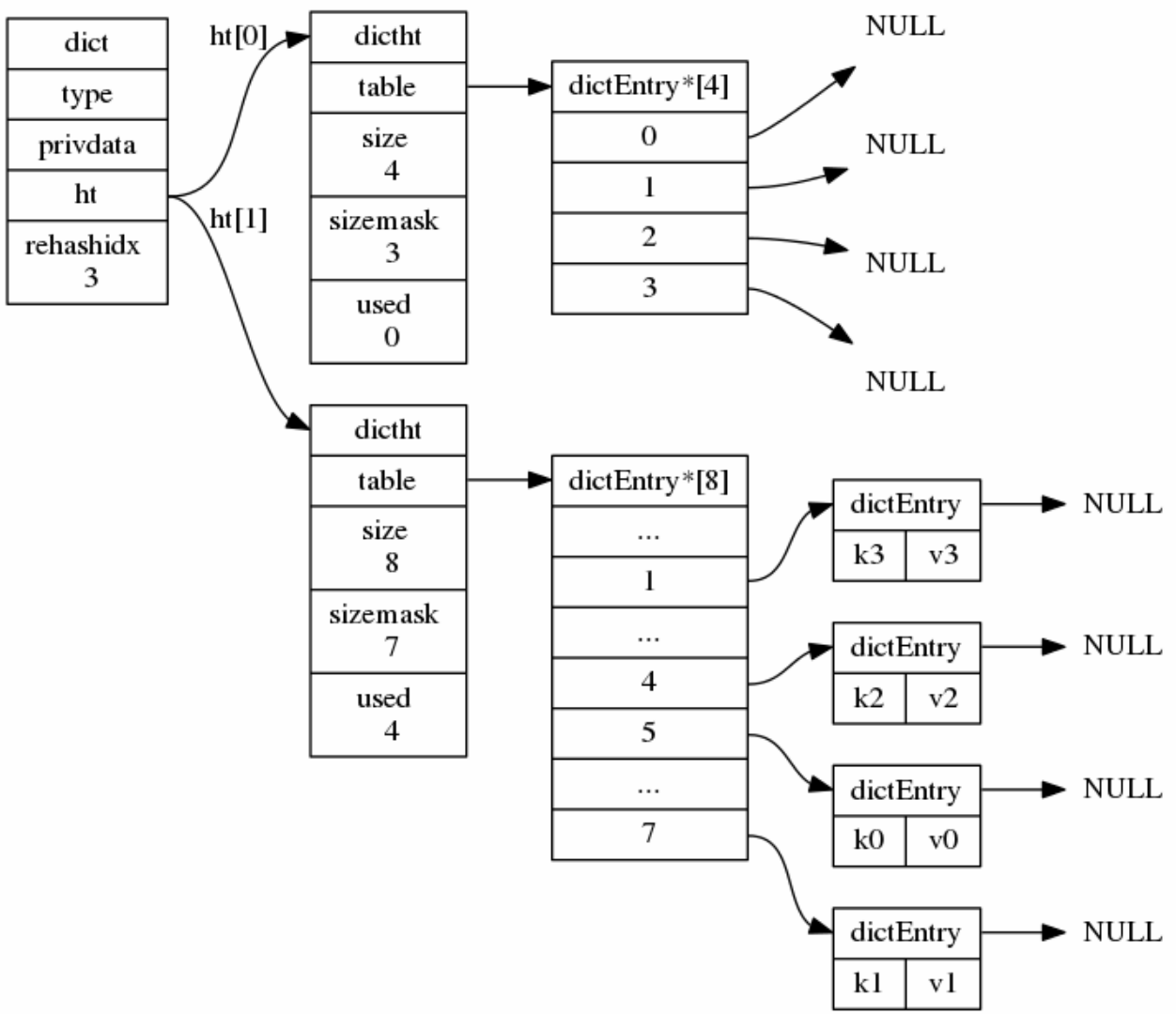
# 索引1上的键值对


dict
type
privdata
ht
rehashidx
1
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
2
dictEntry*[4]
0
1
2
3
NULL
NULL
dictEntry
k3
v3
NULL
dictEntry
k1
v1
NULL
dictht
table
size
8
sizemask
7
used
2
dictEntry*[8]
...
4
5
...
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL

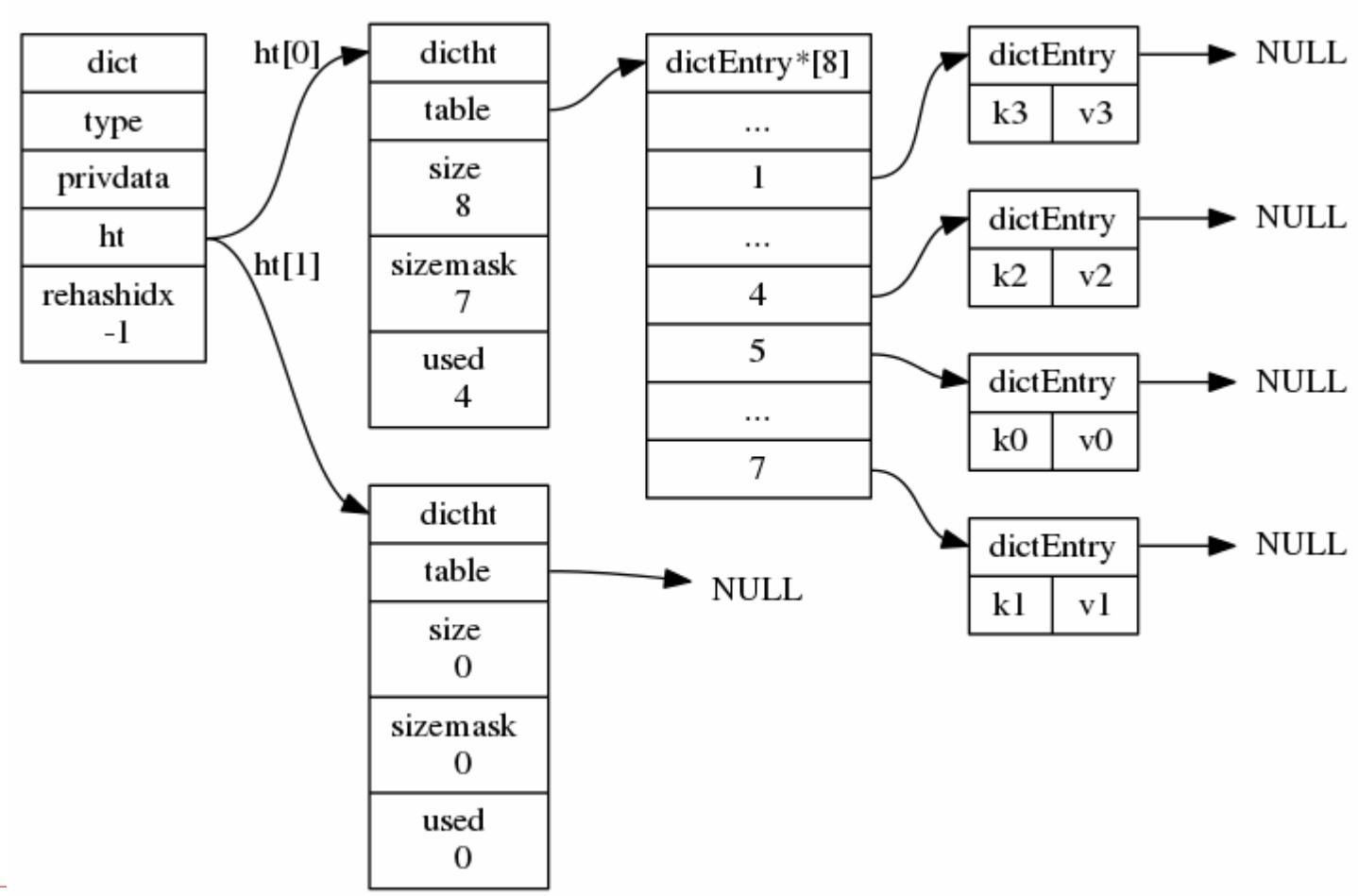
# 索引2上的键值对



# 索引3上的键值对


dict
type
privdata
ht
rehashidx
3
ht[0]
ht[1]
dictht
table
size
4
sizemask
3
used
0
dictEntry*[4]
0
1
2
3
NULL
NULL
NULL
NULL
dictht
table
size
8
sizemask
7
used
4
dictEntry*[8]
...
1
...
4
5
...
7
dictEntry
k3
v3
NULL
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL
dictEntry
k1
v1
NULL

# Rehash执行完毕


dict
type
privdata
ht
rehashidx
-1
ht[0]
ht[1]
dictht
table
size
8
sizemask
7
used
4
dictEntry*[8]
...
1
...
4
5
...
7
dictEntry
k3
v3
NULL
dictEntry
k2
v2
NULL
dictEntry
k0
v0
NULL
dictEntry
k1
v1
NULL
dictht
table
size
0
sizemask
0
used
0
NULL

# 渐进式Rehash的说明

- 因为在进行渐进式rehash的过程中，字典会同时使用ht[0]和ht[1]两个哈希表，所以在渐进式rehash进行期间，字典的删除（delete）、查找（find）、更新（update）等操作会在两个哈希表上进行：比如说，要在字典里面查找一个键的话，程序会先在ht[0]里面进行查找，如果没找到的话，就会继续到ht[1]里面进行查找，诸如此类。
- 另外，在渐进式rehash执行期间，新添加到字典的键值对一律会被保存到ht[1]里面，而ht[0]则不再进行任何添加操作：这一措施保证了ht[0]包含的键值对数量会只减不增，并随着rehash操作的执行而最终变成空表。

# Redis数据字典总结

- 字典被广泛用于实现Redis的各种功能，其中包括数据库和哈希键。
- Redis中的字典使用哈希表作为底层实现，每个字典带有两个哈希表，一个用于平时使用，另一个仅在进行rehash时使用。
- 当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis使用MurmurHash2算法来计算键的哈希值。
- 哈希表使用链地址法来解决键冲突，被分配到同一个索引上的多个键值对会连接成一个单向链表。
- 在对哈希表进行扩展或者收缩操作时，程序需要将现有哈希表包含的所有键值对rehash到新哈希表里面，并且这个rehash过程并不是一次性地完成的，而是渐进式地完成的。

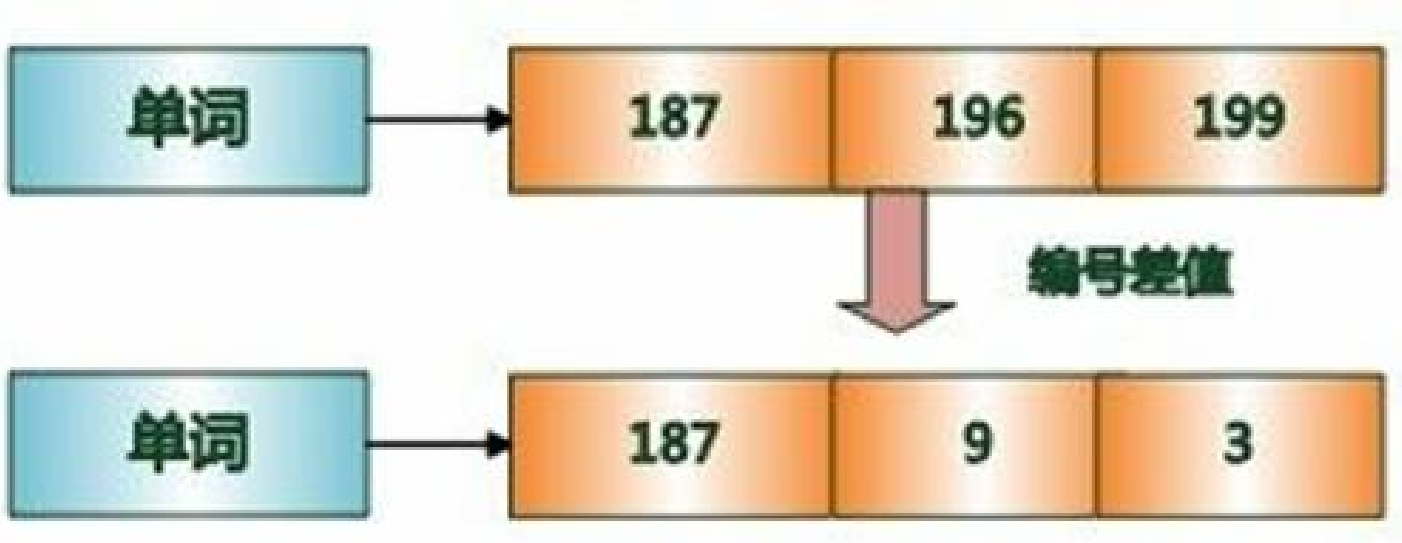
# 使用Hash建立倒排索引

- 倒排索引源于实际应用中需要根据属性的值来查找记录。这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。带有倒排索引的文件称为倒排索引文件，简称倒排文件(inverted file)。

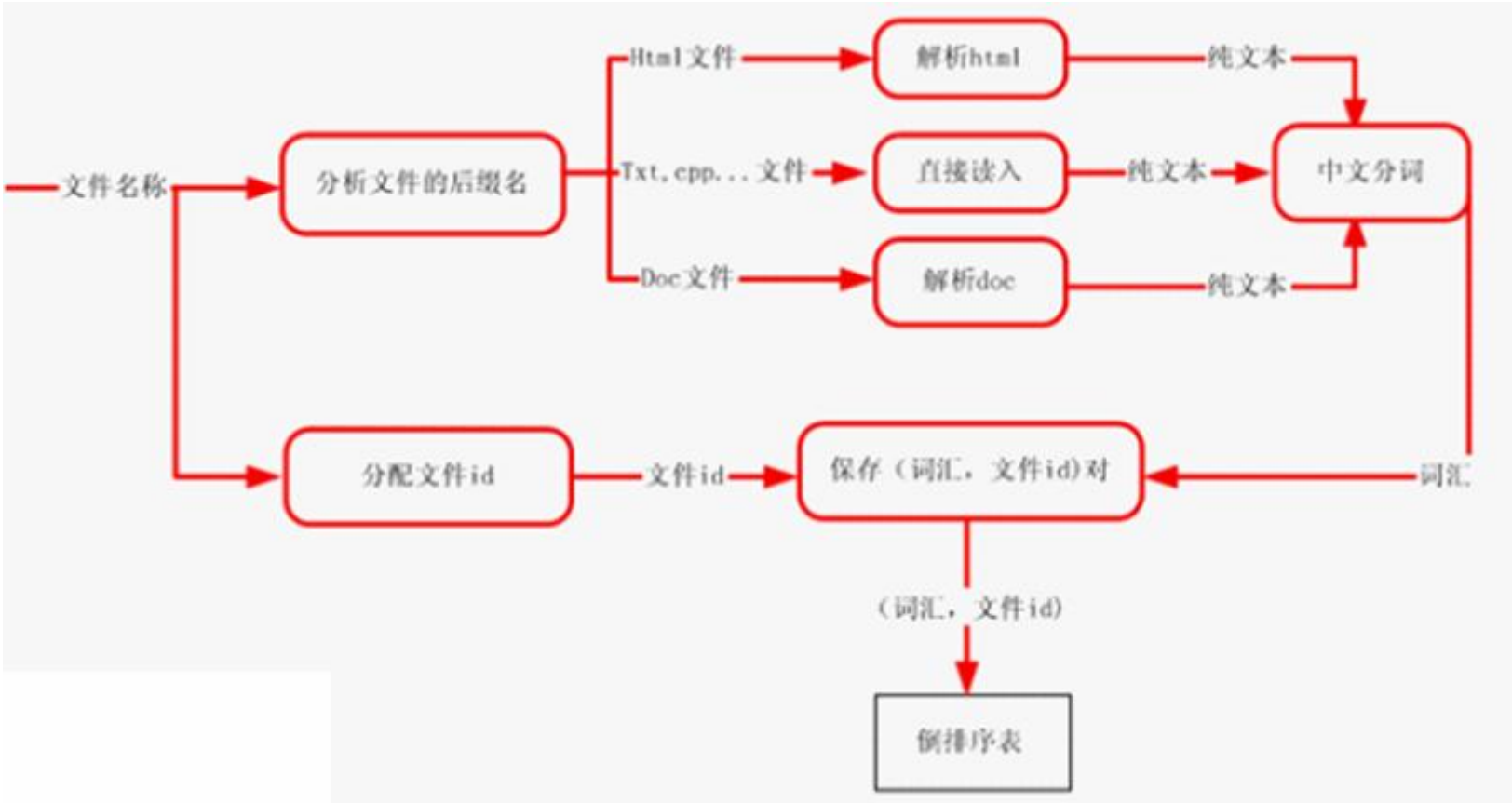
# 倒排列表

- 倒排列表记录了某个单词位于哪些文档中。一般在文档集合里会有很多文档包含某个单词，每个文档会记录文档编号（DocID），单词在这个文档中出现的次数（TF）及单词在文档中哪些位置出现过等信息，这样与一个文档相关的信息被称做倒排索引项（Posting），包含这个单词的一系列倒排索引项形成了列表结构，这就是某个单词对应的倒排列表。

# 倒排列表



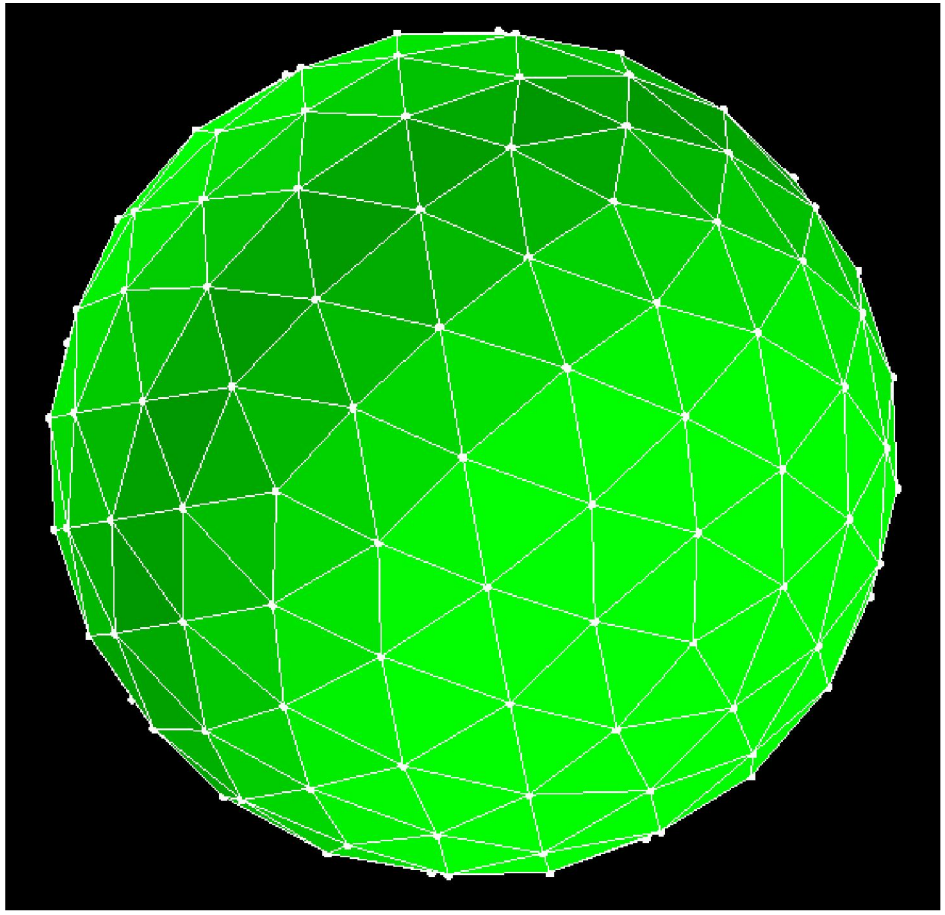
# 倒排索引

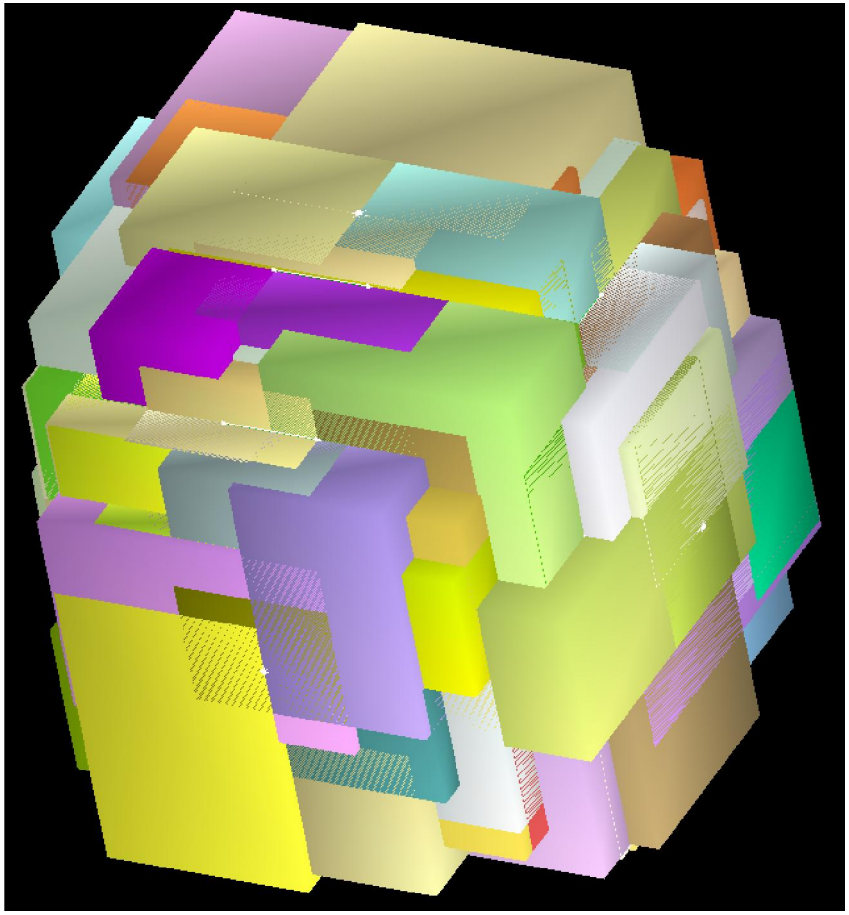

文件名称
分析文件的后缀名
Html文件
Txt,cpp...文件
Doc文件
解析html
直接读入
解析doc
纯文本
纯文本
纯文本
中文分词
分配文件id
文件id
保存（词汇，文件id)对
词汇
（词汇，文件id)
倒排序表

# 更新策略

- 完全重建策略：当新增文档到达一定数量，将新增文档和原先的老文档整合，然后利用静态索引创建方法对所有文档重建索引，新索引建立完成后老索引会被遗弃。此法代价高，但是主流商业搜索引擎一般是采用此方式来维护索引的更新。
- 再合并策略：当新增文档进入系统，解析文档，之后更新内存中维护的临时索引，文档中出现的每个单词，在其倒排表列表末尾追加倒排表列表项；一旦临时索引将指定内存消耗光，即进行一次索引合并，这里需要倒排文件里的倒排列表存放顺序已经按照索引单词字典顺序由低到高排序，这样直接顺序扫描合并即可。其缺点是：因为要生成新的倒排索引文件，所以对老索引中的很多单词，尽管其在倒排列表并未发生任何变化，也需要将其从老索引中取出来并写入新索引中，这样对磁盘消耗是没必要的。
- 原地更新策略：试图改进再合并策略，在原地合并倒排表，这需要提前分配一定的空间给未来插入，如果提前分配的空间不够了需要迁移。实际显示，其索引更新的效率比再合并策略要低。
- 混合策略：出发点是能够结合不同索引更新策略的长处，将不同索引更新策略混合，以形成更高效的方法。

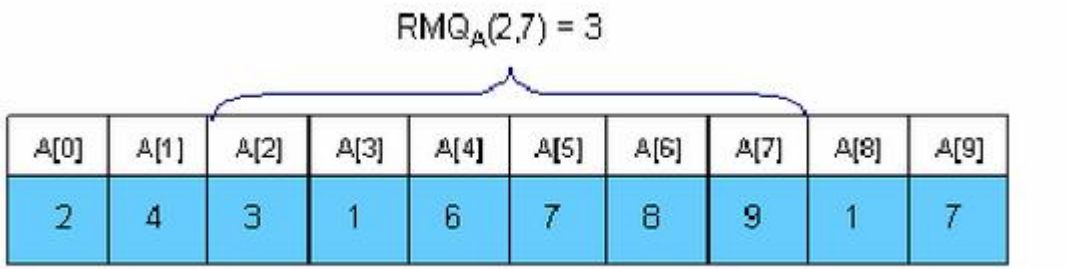
# 三维空间中的R索引





# RMQ

- Range Minimum/Maximum Query：给定数组A[0,N-1]找出给定的两个索引间的最小值的位置。通常用在需要多次询问一些区间的最值的问题中。



# 问题分析

- 最简单的算法是O(n)的，但是对于查询次数m很多（假设有100万次），则这个算法的时间复杂度为O(mn)，显然时间效率太低。
- 可以对数组进行预处理f(n)，然后再查询g(n)：记<f(n),g(n)>
- 可以用线段树将查询算法优化到O（logn)（在线段树中保存线段的最值），而线段树的预处理时间复杂度为O(n)，线段树整体复杂度为<O(n),O(logn)>。
- 不过，Sparse Table算法（ST算法）才是最好的：它可以在O(nlogn)的预处理以后实现O(1)的查询效率，即整体时间复杂度为<O(nlogn),O(1)>。
  - 何为最好？

# 最直观的动态规划方案

- 用dp[i,j]表示数组A[i,i+1…,j]的最小值，i≤j
- dp[i,i]=A[i];
- dp[i,j]=min{dp[i,j-1], A[j]}
- i从0到n-1，j从i到n-1
- 时间复杂度O(N^2)

# Straight DP for Code

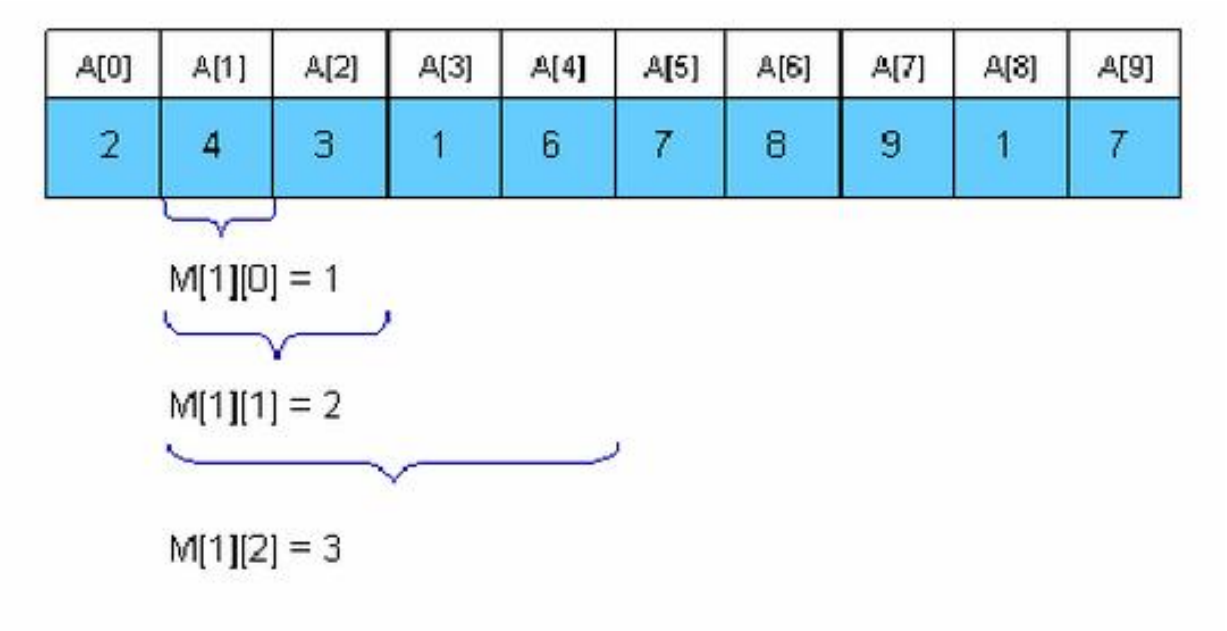
```
void process1(int M[MAXN][MAXN], int A[MAXN], int N)
{
    int i, j;
    for (i =0; i < N; i++)
        M[i][i] = i;
    for (i = 0; i < N; i++)
        for (j = i + 1; j < N; j++)
            if (A[M[i][j - 1]] < A[j])
                M[i][j] = M[i][j - 1];
            else
                M[i][j] = j;
}
```

# 更好的动态规划方案

- 对2^k的长度的子数组进行动态规划。
- dp[i,j]表示[i,i+2^j-1]区间中的最小值，即dp[i,j]表示从第i个数起连续2^j个数中的最小值。
  - dp(1,0)表示[1,1]之间的最小值，即A[1]
  - dp(1,2)表示[1,4]之间的最小值
  - dp(2,4)表示[2,17]之间的最小值
- 初值dp(i,0)=A[i]
- dp(i,j)可以由dp(i,j-1)和dp(i+2^(j-1),j-1)导出

# 状态转移方程

- dp(i,0)=A[i]
- dp(i,j)=min(dp(i,j-1), dp(i+2^(j-1),j-1) )

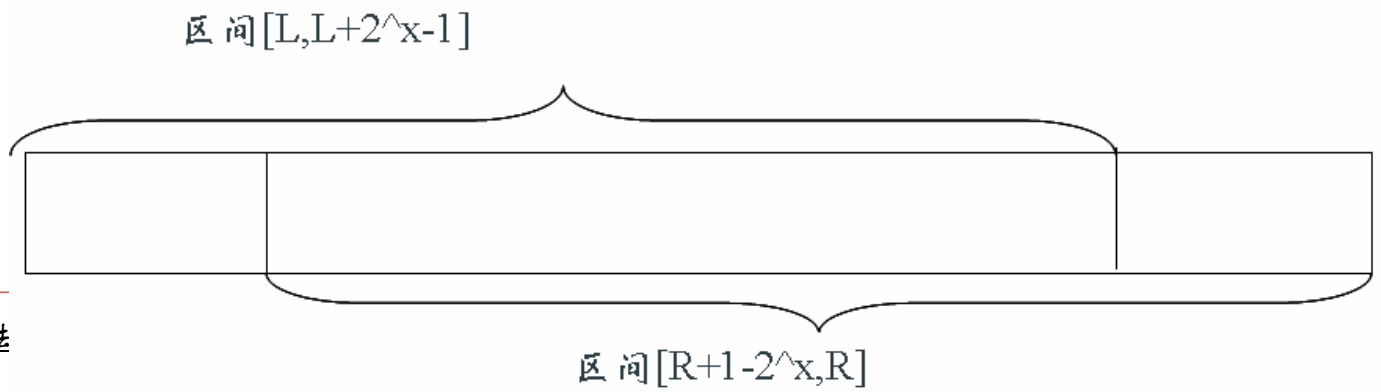


# ST算法预处理部分

```
void process2(int M[MAXN][LOGMAXN], int A[MAXN], int N)
{
    int i, j;
    //initialize M for the intervals with length 1

    for (i = 0; i < N; i++)
        M[i][0] = i;

    //compute values from smaller to bigger intervals
    for (j = 1; 1 << j <= N; j++)
        for (i = 0; i + (1 << j) - 1 < N; i++)
            if (A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])
                M[i][j] = M[i][j - 1];
            else
                M[i][j] = M[i + (1 << (j - 1))][j - 1];
}
```

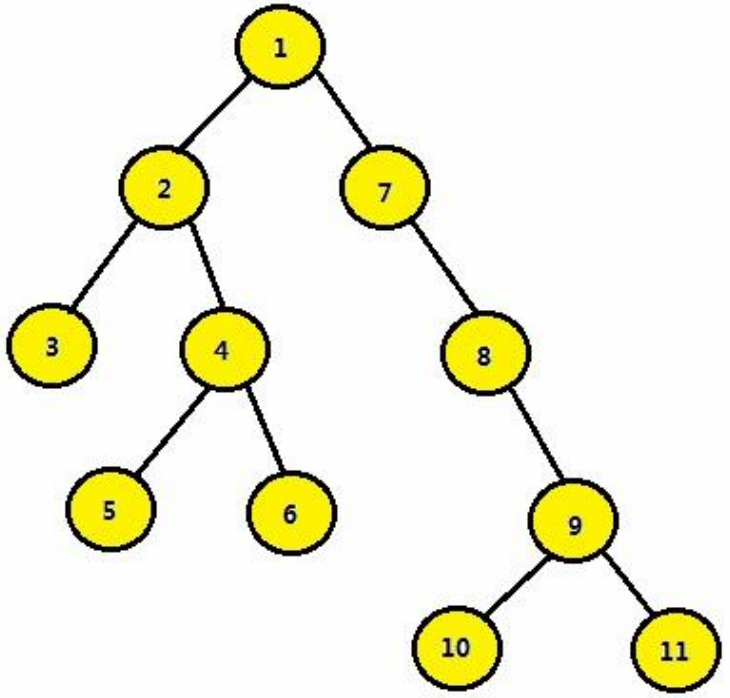
# 如何使用数组dp[i,j]进行查询

- 假设要查询A[L…R]的最小值，那么先求出一个最大的x，使得x满足2^x≤(R-L+1)；
- 于是把区间[L, R]分成两个(部分重叠的)长度为2^x的区间：[L,L+2^x-1]和[R-2^x+1,R]；
- 而dp(L,x)为[L,L+2^x-1]的最小值，dp(R-2^x+1,x)为[R-2^x+1,x]的最小值；
- 返回其中更小的那个，就是A[L…R]的最小值，时间复杂度是O(1)。



# LCA

- 最近公共祖先(Lowest Common Ancestor, LCA)：给定一棵树 T 和两个节点 u 和 v，找出 u 和 v 离根节点最远的公共祖先。
- LCA(3,4)=2
- LCA(3,2)=2
- LCA(3,6)=4
- LCA(6, 10)=1



# 问题转化

- 在有父指针的前提下，该问题即为寻找两个单向链表的第一个公共结点。
- 两个单链表的第一个公共结点问题，下文不妨简称单链公共结点问题。

# 单链公共结点问题

```
typedef struct tagListNode
{
   int m_nKey;
   tagListNode* m_pNext;
}ListNode;
```

- 如果两个单向链表有公共的结点，也就是说两个链表从某一结点开始，它们的m_pNext都指向同一个结点。但由于是单向链表的结点，每个结点只有一个m_pNext，因此从第一个公共结点开始，之后它们所有结点都是重合的，不可能再出现分叉。所以，两个有公共结点而部分重合的链表，拓扑形状看起来像一个Y，而不可能像X。
- 蛮力法：在第一个链表上顺序遍历每个结点。每遍历一个结点的时候，在第二个链表上顺序遍历每个结点。如果此时两个链表上的结点是一样的，说明此时两个链表重合，于是找到了它们的公共结点。如果第一个链表的长度为m，第二个链表的长度为n，显然，该方法的时间复杂度为O(mn)。

# 单链公共结点问题

- 分析：如果第一个链表的长度为m，第二个链表的长度为n，不妨认为m≥n，由于两个链表从第一个公共结点到链表的尾结点是完全重合的。所以前面的(m-n)个结点一定没有公共结点。
- 算法：先分别遍历两个链表得到它们的长度m，n。在长的链表上先遍历|m-n|次后，再同步遍历两个链表，直到找到相同的结点，或者一直到链表结束。时间复杂度为O(m+n)。
- 进一步的问题：如果两个链表可能有环，如何判断两个链表是否相交？以及找到两个链表的第一个公共点？
  - 快慢指针

# 一般LCA

- 在没有父指针的情况下，可以通过从根到v，u的递归查找，找到最近公共祖先。

# Code

```
node* getLCA(node* root, node* node1, node* node2)
{
    if(root == null)
        return null;
    if(root== node1 || root==node2)
        return root;

    node* left = getLCA(root->left, node1, node2);
    node* right = getLCA(root->right, node1, node2);

    if(left != null && right != null)
        return root;
    else if(left != null)
        return left;
    else if (right != null)
        return right;
    else
        return null;
}
```

# 递归代码详解

```
if(root== node1 || root==node2)
    return root;
```

- 因为函数要返回node1和node2的最近祖先，但事实上，此处返回的，并不一定是“最正”的结论。
- 比如，node1==root，同时，node1不是node2的祖先，那么，“正”结论应该是null，但该代码返回的是node1

# 递归代码详解

node* left = getLCA(root->left, node1, node2);

node* right = getLCA(root->right, node1, node2);

- 第一句，会返回(node1,node2)的“潜在祖先”
  - ①如果node1，node2分立在root的左、右子树中，不妨认为node1在左、node2在右，返回的是node1；
  - ②如果node1，node2都在root的左子树中，将返回node1，node2的LCA
  - ③如果node1，node2都在root的右子树中，将返回null
- 第二句，返回的情况和第一句对称，不再赘述
- ②、③的情况，可以认为返回的是(node1,node2)的LCA，但①的情况，不是LCA。但是，暂时不是，没关系。

# 递归代码详解

```
if(left != null && right != null)
    return root;
```

- 如果发现left和right都不空，说明前面的第一、二句都返回了非空结果，那必然是node1在root的一侧，node2在另外一侧。root就是它们俩的LCA。

# 递归代码详解

```
else if(left != null)
        return left;
else if (right != null)
        return right;
```

# 递归代码详解

```
else if(left != null)
        return left;
else if (right != null)
        return right;
else
        return null;
```

- “最正”的情况：第一句的②情况： node1，node2都在root的左子树中，因此，只有left为非空（第二个else if情况对称，不再赘述）
- return null：就是node1和node2，两个都没有在树root中，显然应该返回null

# 该递归代码的进一步思考

- 这个函数其实有个“bug”：如果node1在树中，node2不在。按照LCA的定义，应该返回null，但它返回的是node1；
- 但这个“bug”其实无所谓。两个不在一颗树下的结点，非要算他们的共同祖先...
- 可以通过增加标记的方法，去除这个bug；同时带来的好处是，如果node1，node2在root的左孩子中，将不必调用root的右孩子。

```
node* getLCA(node* root, node* node1, node* node2, int& nodeSize)
{
        if (root == NULL)
        {
                nodeSize = 0;
                return NULL;
        }
        int temp1 = 0;
        node *left = getLCA(root->left, node1, node2,temp1);
        if (temp1 == 2)
                return left;

        int temp2 = 0;
        node *right = getLCA(root->right, node1,node2, temp2);
        if (temp2 == 2)
                return right;

        nodeSize = temp1 + temp2;
        if (root == node1)
                ++nodeSize;
        if (root == node2)
                ++nodeSize;
        if ((nodeSize == 2) || (root == node1) || (root == node2))
                return root;
        else if (temp1 == 1)
                return left;
        else if (temp2 == 1)
                return right;
        return NULL;
}
```

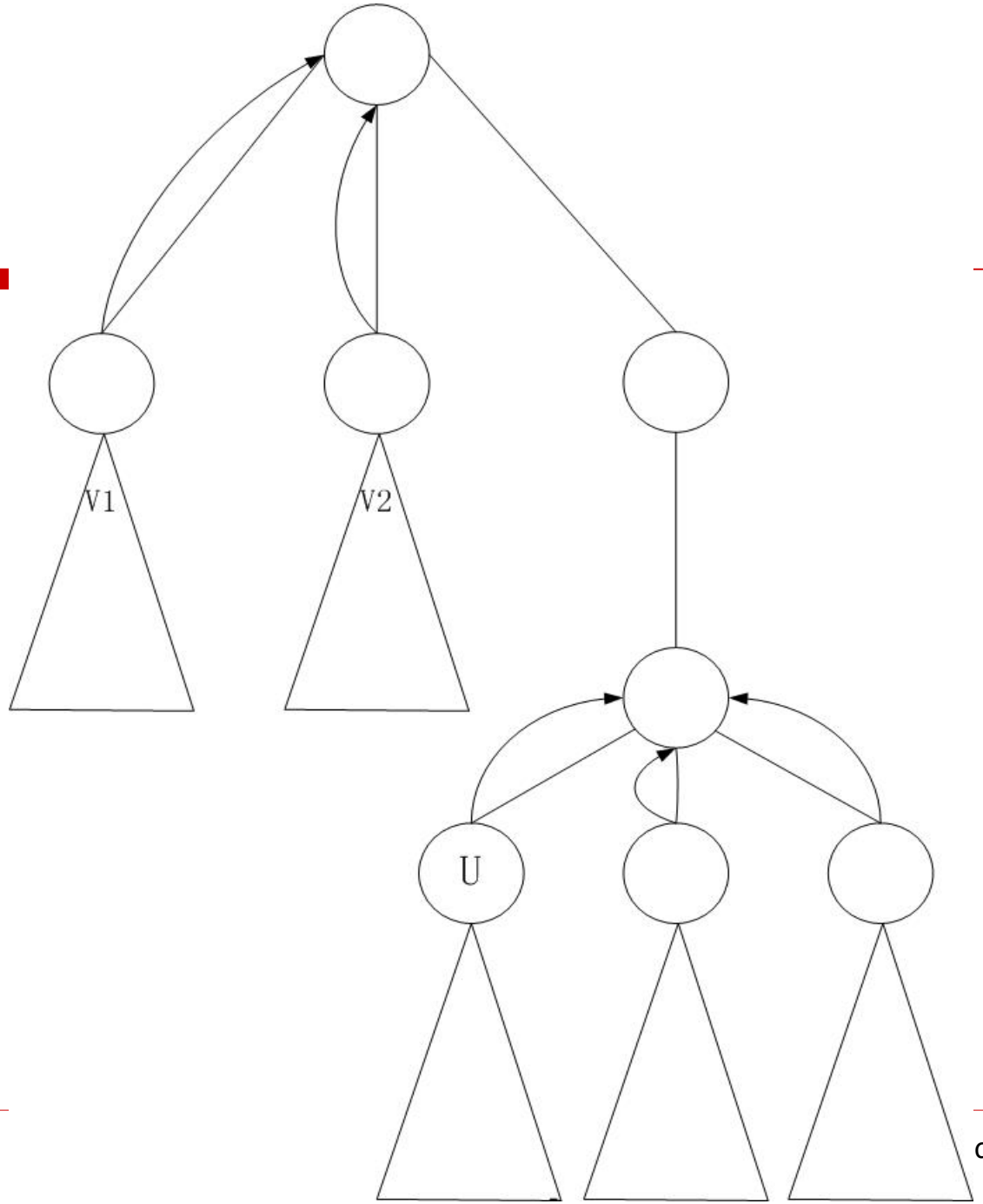
# Tarjan算法

- Tarjan算法是由Robert Tarjan在1979年发现的一种高效的离线算法，也就是说，它要首先读入所有的询问（求一次LCA叫做一次询问），然后并不一定按照原来的顺序处理这些询问，该算法的时间复杂度O(N*α(N)+Q)，其中，α(x)不大于4，N表示问题规模，Q表示询问次数。

# Tarjan概览

- Tarjan算法基于深度优先搜索，对于新搜索到的一个结点u，首先创建由这个结点u构成的集合setU，再对当前结点u的每一个子树subTree进行搜索，每搜索完一棵子树sub，则可确定子树sub内的LCA询问都已解决。其他的LCA询问的结果必然在这个子树sub之外，这时把子树sub所形成的集合setSub与当前结点的集合setU合并成set，并将当前结点u设为这个集合set的祖先Ua。之后继续搜索下一棵子树subNext，直到当前结点u的所有子树搜索完。这时把当前结点u设为checked，同时可以处理有关当前结点u的LCA询问，如果有一个从当前结点u到结点v的询问，且v已被检查过，则由于进行的是深度优先搜索，当前结点u与v的最近公共祖先LCA一定是未checked，而这个最近公共祖先LCA包含v的子树subV一定已经搜索过了，那么LCA一定是v所在集合的祖先Va。

# Tarjan

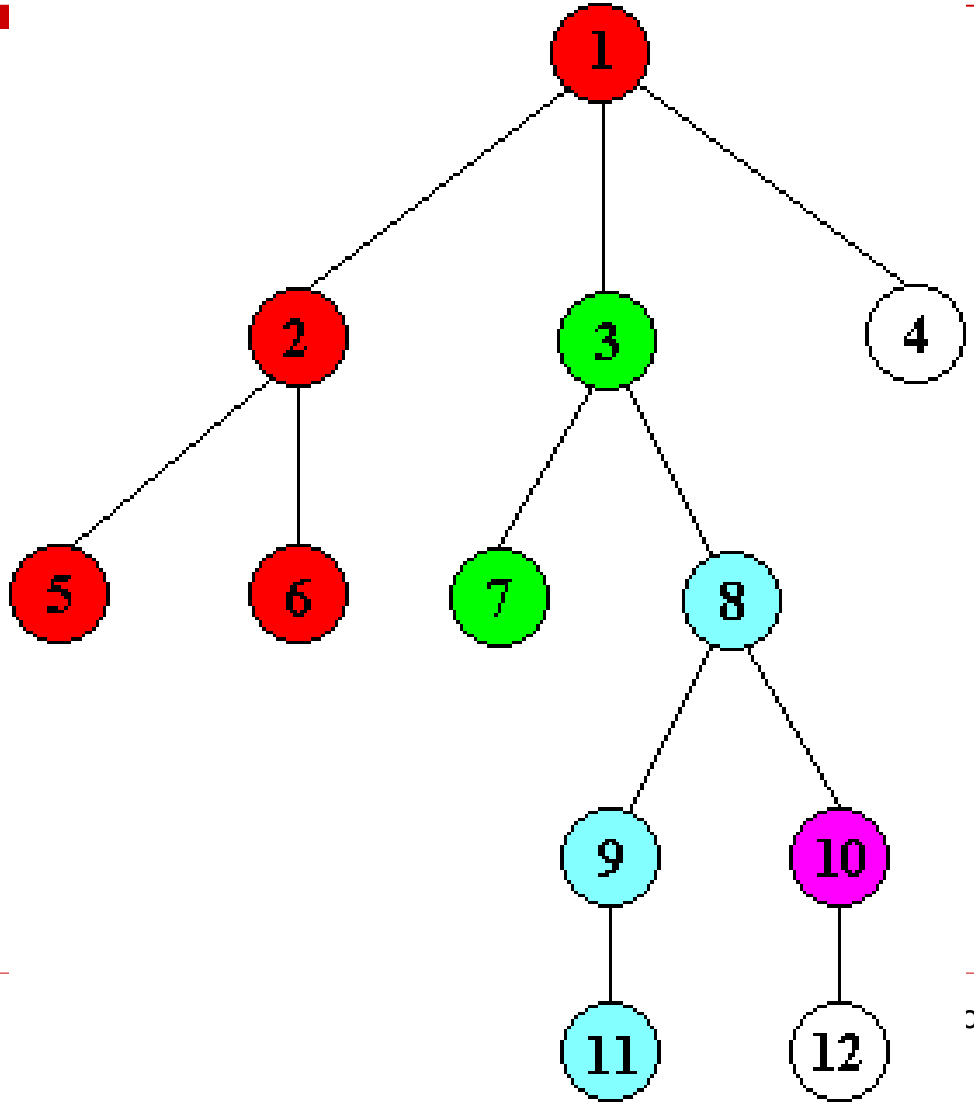


# Tarjan

```
function TarjanOLCA(u)
  MakeSet(u);
  u.ancestor := u;    // 将集合u的祖先指向自己
  for each v in u.children do  // DFS所有孩子
     TarjanOLCA(v);
     Union(u,v);     // 与根结点U合并（并查集操作）
     Find(u).ancestor := u;    // 将u所在集合根的祖先指向u
  u.colour := black;  // 当所有孩子都已遍历，则标记根已完成
  for each v such that {u,v} in P do  // 找所有与 u相关的查询
    if v.colour == black  // 如果另一结点 v是前面标记过的,则输出递归向上返回根的祖先
       print "Tarjan's Least Common Ancestor of " + u +
            " and " + v + " is " + Find(v).ancestor + ".";
```

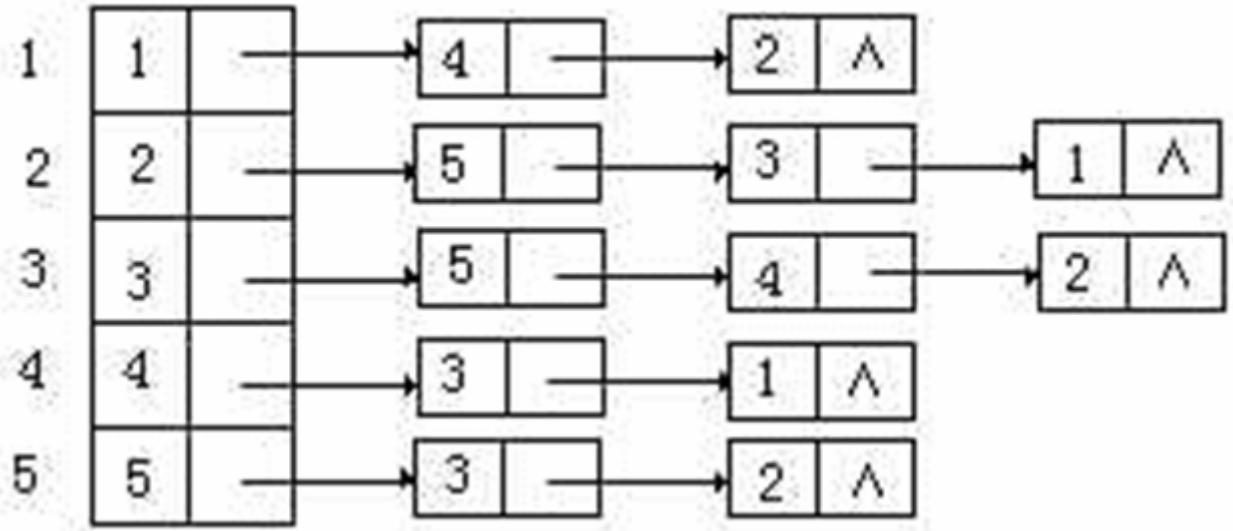
# Tarjan

- (2,10)
- (10,7)
- (9,10)
- (10,8)



# 处理查询的方法 -邻接表

- 遍历P中的查询(a,b)，将a插入到b的列表中，b插入到a的列表中；
- 常见的空间检索方案。

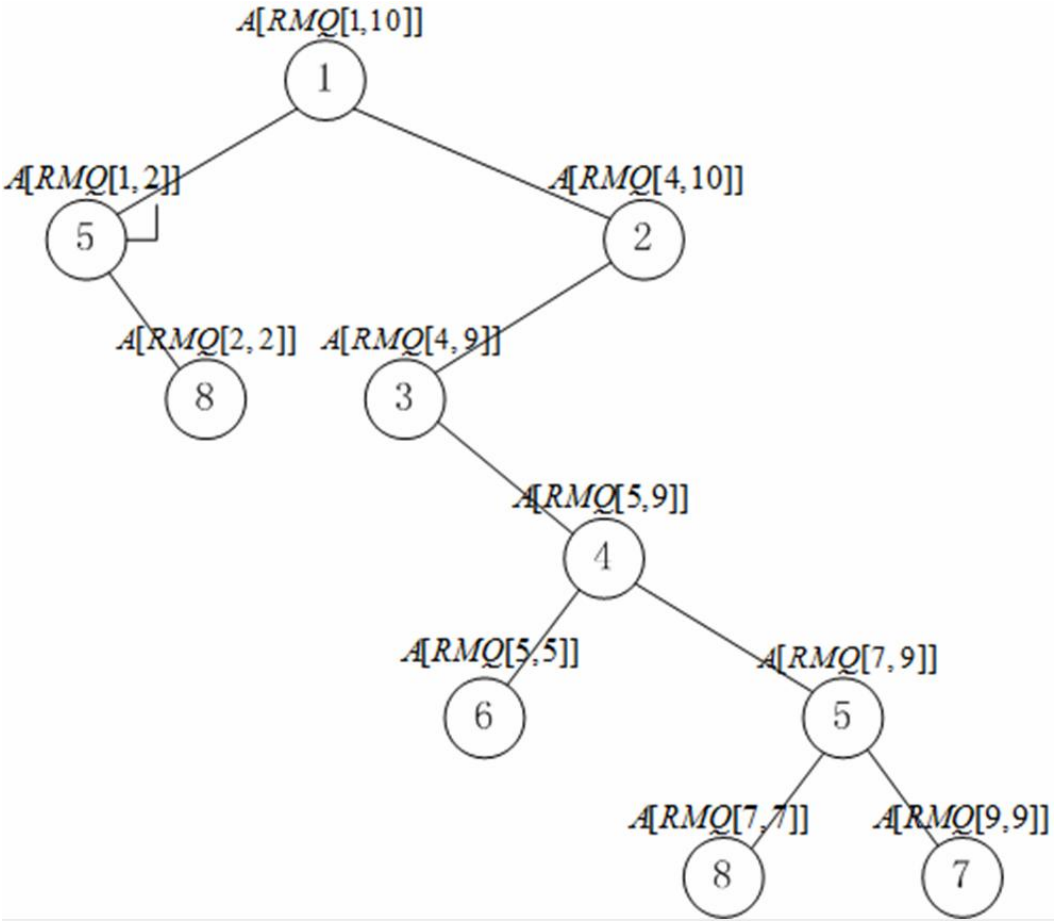


# RMQ可以转换为LCA

- 借助笛卡尔树(CartesianTree)
- 笛卡尔树是一个特殊的堆，它根据一个长度为n的数组A建立。它的根是A的最小元素位置i,而左子树和右子树分别为A[1...i-1]和A[i+1...n]的笛卡尔树。
- 中序遍历笛卡尔树就可以得到原数列。

# 使用构造法求解问题

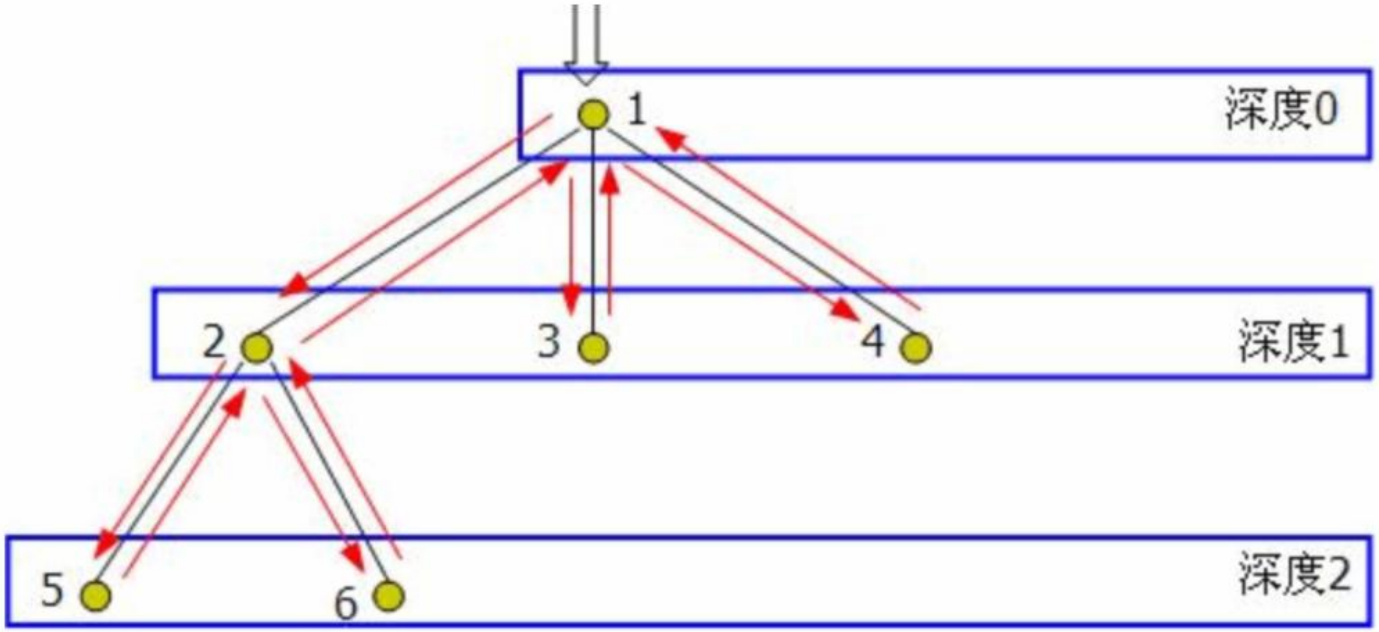
- 5 8 1 3 6 4 8 5 7 2
- 根据定义，A[m...n]中的最小值，就是以该最小值为根的一颗子树



# LCA转换为RMQ

- 欧拉序列：对有根树T进行DFS，将遍历到的结点按照顺序记下，将得到一个长度为2N–1的序列，称之为T的欧拉序列F。
- 通过欧拉序列，能够将一颗树映射成一个数组。

# 考察F的第一次出现




DFS 遍历

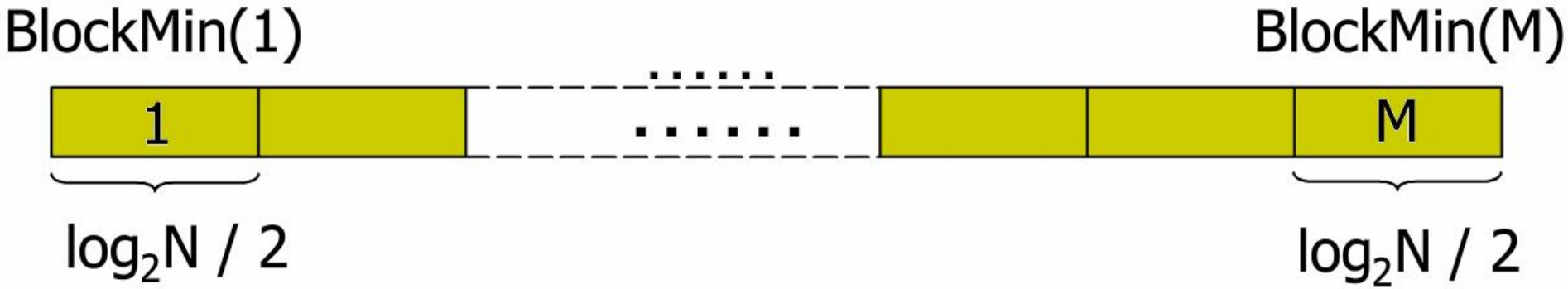
| 欧拉序列 F | 1 | 2 | 5 | 2 | 6 | 2 | 1 | 3 | 1 | 4 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 深度序列 B | 0 | 1 | 2 | 1 | 2 | 1 | 0 | 1 | 0 | 1 | 0 |

# LCA转换成RMQ之后

- 将LCA问题转换为RMQ问题后，利用ST算法来求解，时间复杂度是O(nlogn)+Q*O(1)，反而不及直接利用Tarjian算法求解LCA问题时间效率O(N*α+Q)高。
- ±1RMQ算法的时间复杂度为<O(n),O(1)>，而上述转换中RMQ(B,pos(u),pos(v))深度序列B数组中任意相邻两个数的差是±1，正好利用“±1RMQ”算法来求解。

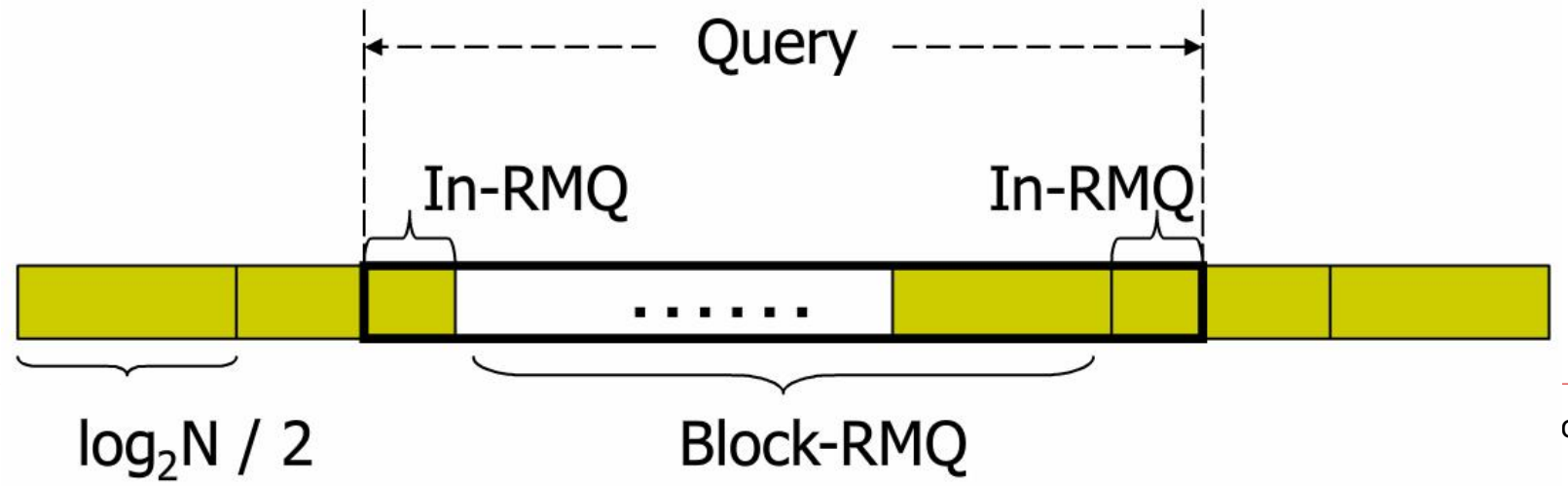
# ±1RMQ

- 算法的核心思想是分块；
- 以L=$\log_2 N/2$的块长，把B划分成M=N/L段，记录第k块的最小元素为BlockMin(k)，把M块的最小值组成序列Blocks，利用分块思想，可以把询问分成两个部分：



# ±1RMQ

- 求解方法：
  - 连续的BlockMin取最小值，即Block-RMQ；
    - ST算法
  - 两端块中某一部分取最小值，即In-RMQ；
- 这两个问题都可以O(N)内实现。



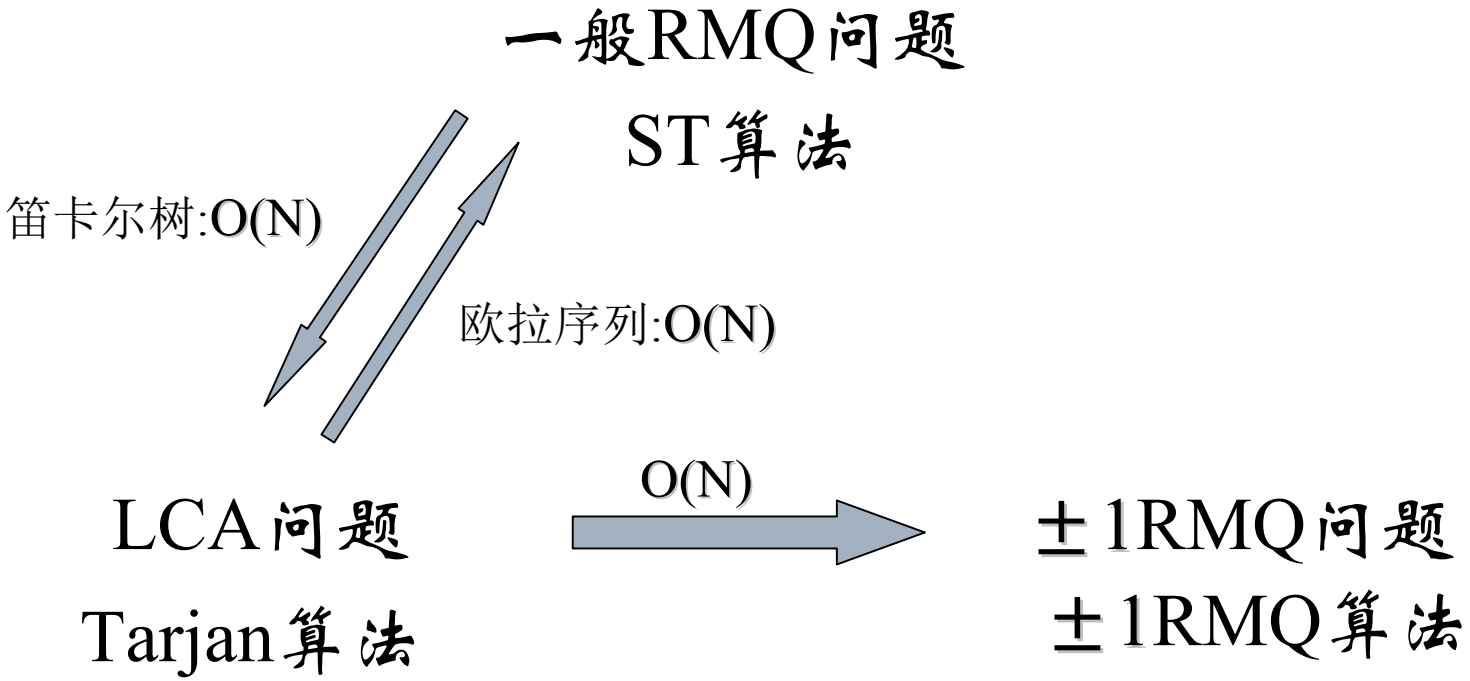
# Block-RMQ

- Block-RMQ采用ST算法，时间复杂度为$O(M\log_2 M)$
- 因为$M\log_2 M<2N/\log_2 N*\log_2 N=O(N)$，所以，Block-RMQ的时间复杂度不大于O(N)

# In-RMQ

- B中任意两个相邻数为±1，所以，本质上，不同的块至多有$2^L$种；
- 对于每一种块上的询问最多只有$O(L^2)$种；
- 所以，本质上不同的询问数最多有
  - $O(2^{L*}L^2)<O(N^{0.5}\log_2 N*\log_2 N)<O(N)$
- 完成In-RMQ只需要预处理本质不同的询问，然后对应作答即可，时间复杂度为O(N)

# LCA & RMQ & ±1RMQ 的相互转换



# 由于±1RMQ线性时间带来的结论

- 因为LCA问题可以转换成生成序列的RMQ问题，并且问题规模不变，所以，LCA问题可以在O(N)的时间内解决；
- 因为RMQ问题可以转换成LCA问题，并且问题规模不变，所以，RMQ问题可以在O(N)的时间内解决；

# 参考文献


- 陈海波，王申康，RTree的查询代价模型分析及算法改进，计算机辅助设计与图形学学报[J]，15,2003(3)
- 程昌秀，矢量数据多尺度空间索引方法的研究，武汉大学学报·信息科学版[J]，34,2009(5)
- Eric Redmond, etc, 王海鹏等 译，七周七数据库[M]，7th,2013
- http://m.blog.csdn.net/blog/zh_qd1014/6879083(LCA&RMQ)
- http://www.redisbook.com/en/latest/toc.html(Redis&Hash)
- http://www.cse.yorku.ca/~oz/hash.html(djb2 Hash)
- http://blog.csdn.net/jsjwk/article/details/7964108(rehash)
- http://zh.wikipedia.org/wiki/Murmur%E5%93%88%E5%B8%8C(MurmurHash)
- http://baike.baidu.com/view/676861.htm(倒排索引)
- http://www.coderplusplus.com/?p=393(笛卡尔树)
- http://wenku.baidu.com/view/c197275e804d2b160b4ec0c4.html（LCA&RMQ）
- http://blog.163.com/clevertanglei900@126/blog/static/111352259201191414846 7/(单链公共结点问题)
- http://baike.baidu.com/view/6667519.htm(笛卡尔树)
- http://zh.wikipedia.org/wiki/%E7%AC%9B%E5%8D%A1%E5%B0%94%E6%A0%91 (笛卡尔树)
- http://m.blog.csdn.net/blog/zh_qd1014/6879083 （LCA&RMQ）

# 我们在这里

- 更多算法面试题在 7 七月算法 官网
  - http://www.julyedu.com/
    - 免费视频
    - 直播课程
    - 问答社区

- contact us：微博
  - @七月问答
  - @七月算法

感谢大家

恳请大家批评指正！