redis背后的数据结构-skiplist

讲师：唐僧

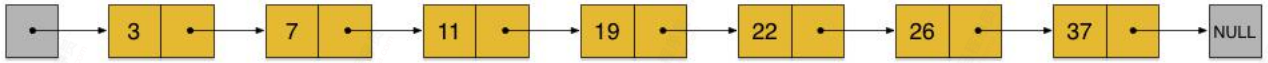
多年一线研发经验，项目覆盖电商，金融，办公自动化，在线教育等，有着非常丰富的项目开发经验，且致力于研究大厂面试算法多年，有着丰富的算法面经，另外在大数据，物联网等方面也有着深入的研究。

跳跃表-skiplist

是由William Pugh发明的，最早出现于他在1990年发表的论文《Skip Lists: A Probabilistic Alternative to Balanced Trees》

ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf

- 普通有序链表：



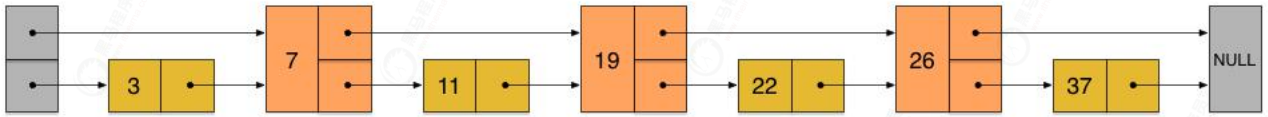
  - 查询复杂度O(n)
  - 插入删除，同样需要先找到要插入的位置和被删除的元素

- 如何提升效率？

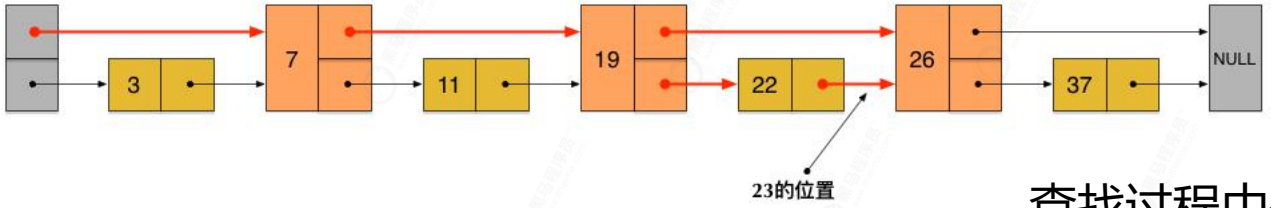
  - 空间换时间，升维

skiplist如何用空间换时间

假如我们每相邻两个节点增加一个指针，让指针指向下下个节点，会发生什么？



- 新增加的指针连成了一个新的链表，包含的节点个数只有原来的一半（7, 19, 26）
- 查找/插入23如何操作？
  - 先沿着这个新链表进行查找，当碰到比待查数据大的节点时，再回到原来的链表中进行查找

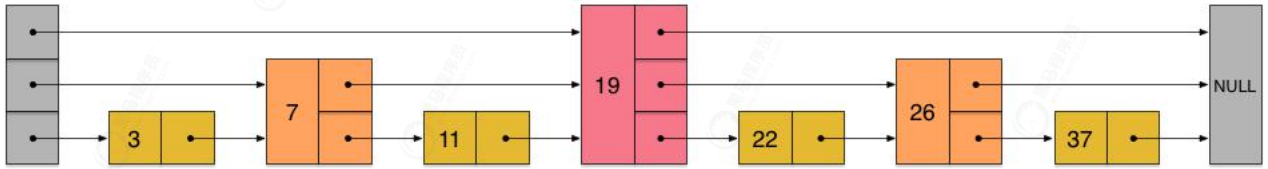


查找过程中需要比较的节点数大概只有原来的一半

skiplist如何用空间换时间

- **如果想继续提高效率？**

  继续为每相邻的两个节点增加一个指针，从而产生第三层链表

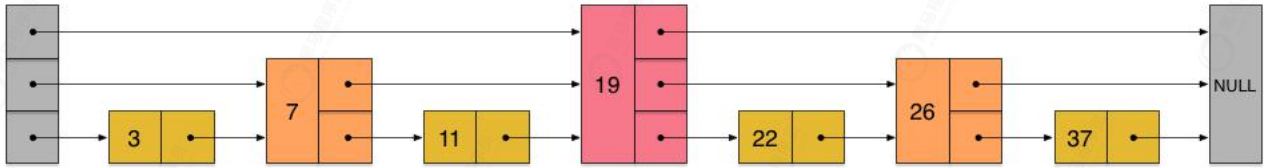


- 查找/插入23如何操作?

可以一次性跳过19前面的元素

当链表足够长的时候，这种多层链表的查找方式能让我们跳过很多下层节点，大大加快查找的速度

查找复杂度分析



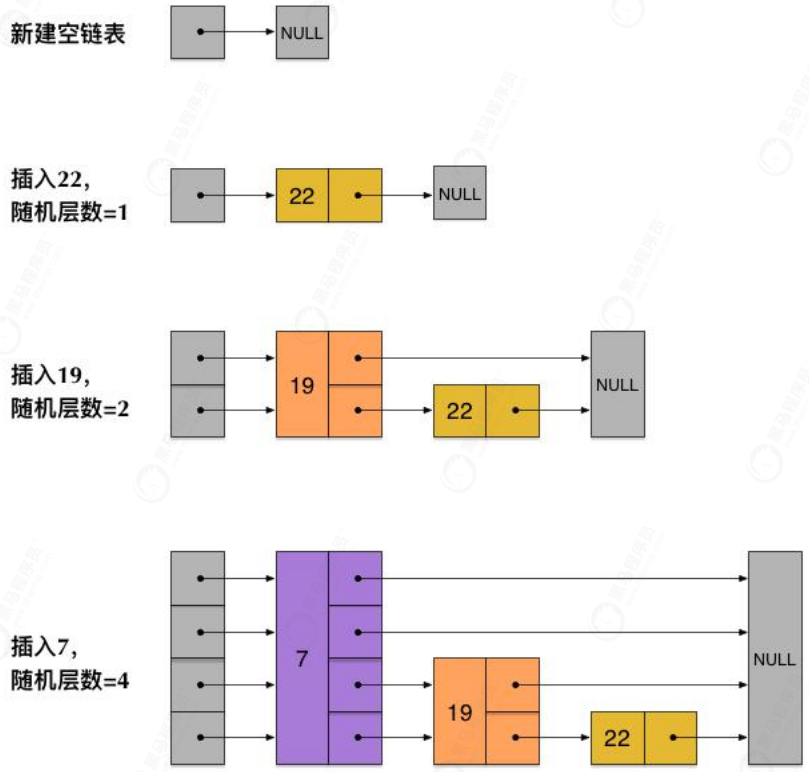
- 按照上面生成链表的方式，上面每一层链表的节点个数，是下面一层的节点个数的一半，这样查找过程就非常类似于一个二分查找，使得查找的时间复杂度可以降低到**O(log n)**

- 弊端：插入/删除元素后，打乱了上下相邻两层链表上节点个数严格的比例关系。为保证这种严格的关系就需要调整，而重新调整链表会让复杂度蜕化成O(n)

如何避免插入/删除复杂度退化?

- 不要求上下相邻两层链表之间的节点个数有严格的对应关系
- 每个节点随机出一个层数(level)

如何做?

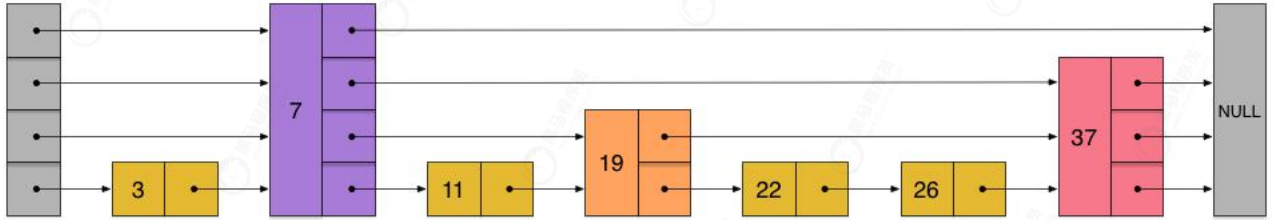


如何避免插入/删除复杂度退化?

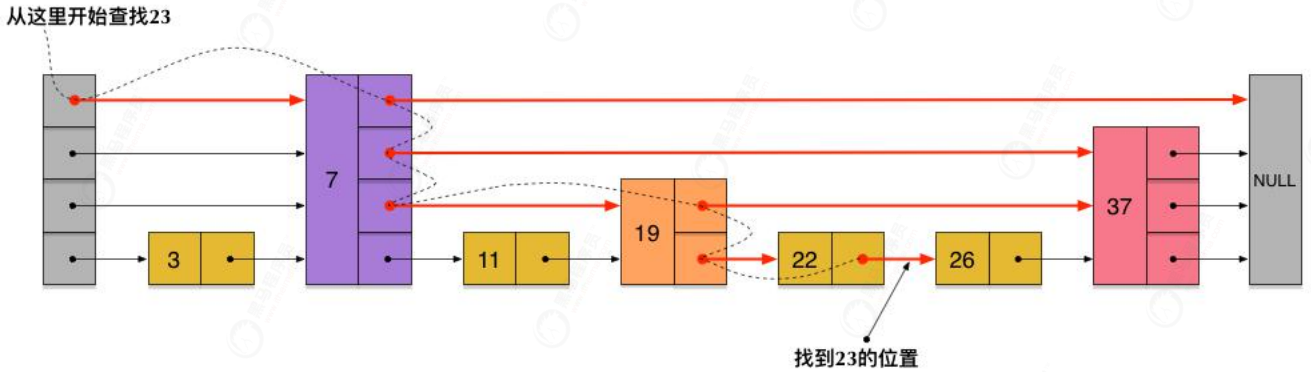
1、节点的层数（level）是随机出来的
2、新插入节点不会影响其它节点的层数，只需要修改插入节点前后的指针

跳表，指的就是除了最下面第1层链表之外，会产生若干层稀疏的链表，这些链表里面的指针跳过了一些节点（而且越高层的链表跳过的节点越多）。这就使得我们在查找数据的时候能够先在高层的链表中进行查找，然后逐层降低，最终降到第1层链表来精确地确定数据位置。在这个过程中，我们跳过了一些节点，从而也就加快了查找速度。

如何查找?

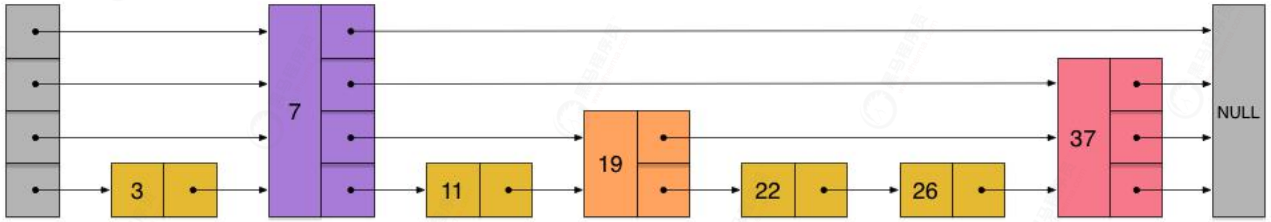


◆ 刚刚创建的这个skiplist总共包含4层链表，如何在它里面查找23？



➢ **对于插入和删除，首先都要经过一个查找过程，找到对应的位置之后在完成插入和删除**

skiplist性能分析



- **节点插入时随机出一个层数，依靠一个简单的随机数操作而构建出来的多层链表结构，能保证良好的查找性能吗?**

```
randomLevel()
    level := 1
    // random()返回一个[0...1)的随机数
    while random() < p and level < MaxLevel do
        level := level + 1
    return level
```

1、首先，每个节点肯定都有第1层指针（每个节点都在第1层链表里）。

2、如果一个节点有第i层(i>=1)指针（即节点已经在第1层到第i层链表中），那么它有第(i+1)层指针的概率为p。

3、节点最大的层数不允许超过一个最大值，记为MaxLevel。

- 经概率统计分析：

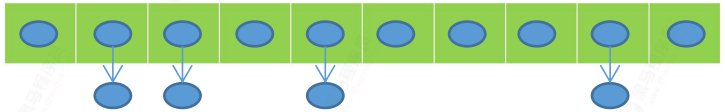
1、一个节点的平均层数（也即包含的平均指针数目）为：1/1-p

2、从第1层到最高层，各层链表的平均节点数是一个指数递减的等比数列

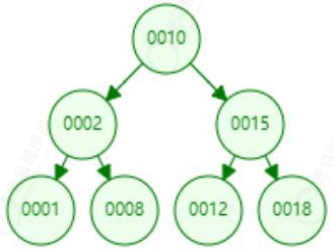
- **查找一个元素的平均时间复杂度为：O(log n)**

skiplist，平衡树，哈希表对比

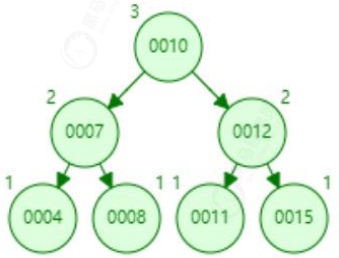
范围查找



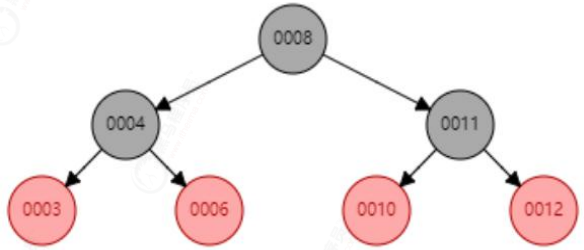
hash表不适合做范围查找



bst

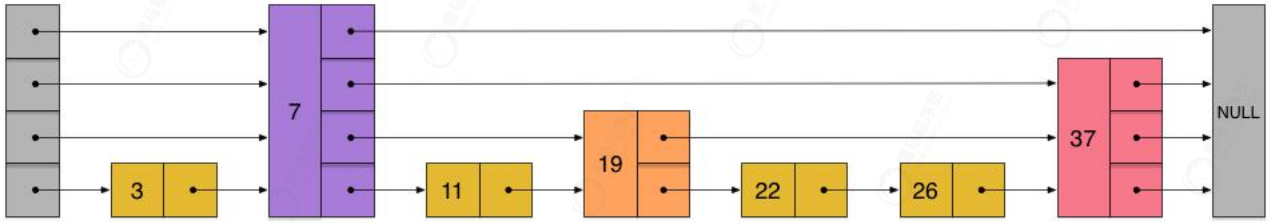


avl



red-black

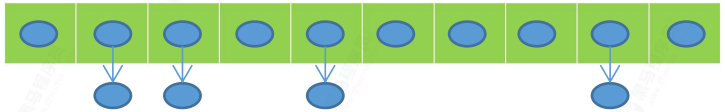
找到指定范围的小值之后，还需要以中序遍历的顺序继续寻找其它不超过大值的节点。如果不对平衡树进行一定的改造，这里的中序遍历并不容易实现



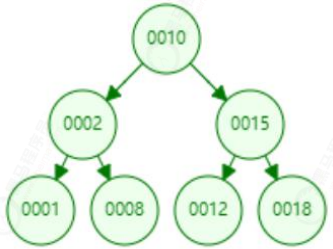
在找到小值之后，对第1层链表进行若干步的遍历就可以实现

skiplist，平衡树，哈希表对比

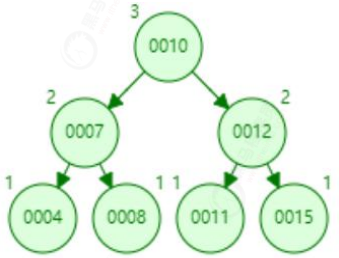
查找操作



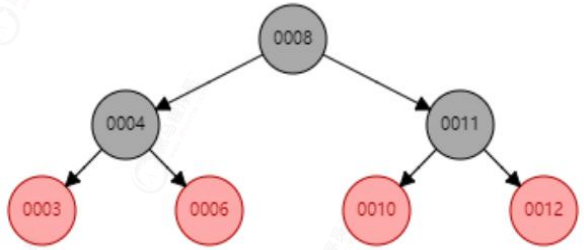
存较低hash冲突情况下
复杂度接近O(1)，性能更高



bst

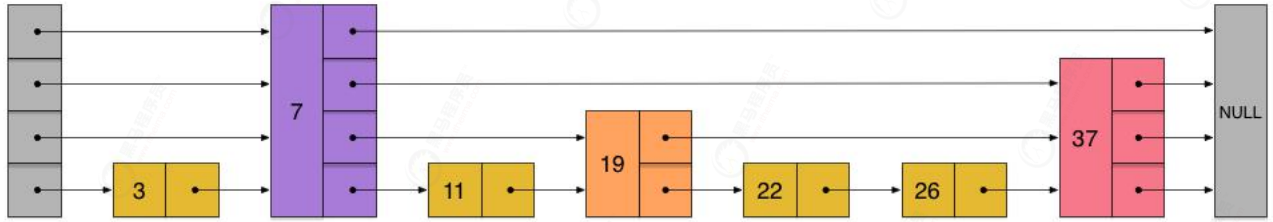


avl



red-black

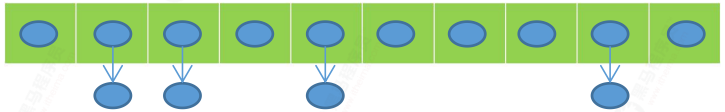
时间复杂度为O(log n)



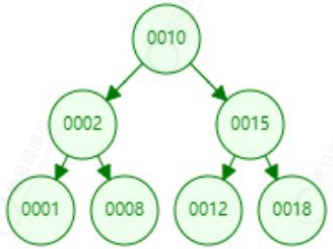
时间复杂度为O(log n)，和平衡树大体相当

skiplist，平衡树，哈希表对比

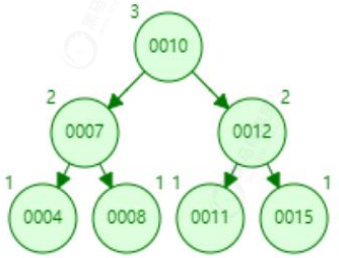
插入删除操作



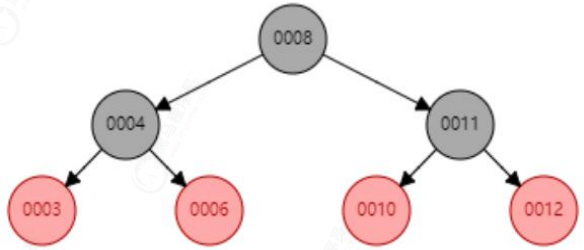
保存较低hash冲突情况下
复杂度接近O(1)，性能更高



bst

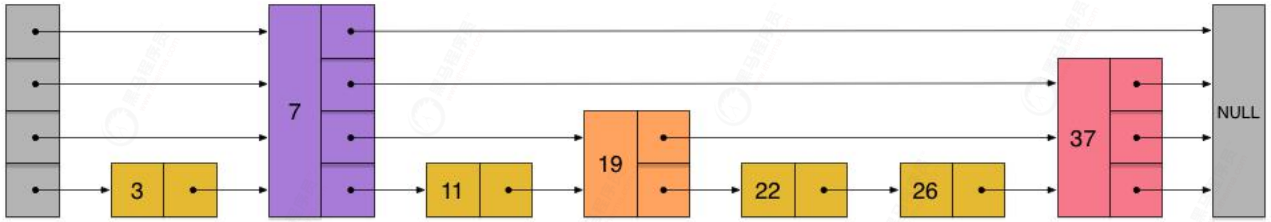


avl



red-black

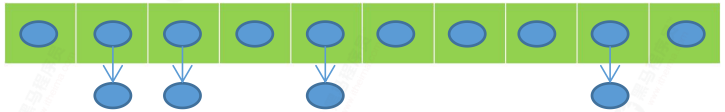
插入和删除操作可能引发子树的调整，逻辑复杂



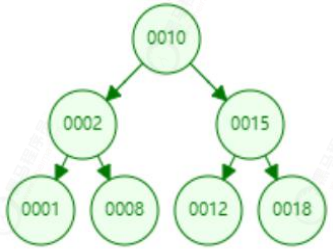
插入和删除只需要修改相邻节点的指针，简单又快速

skiplist，平衡树，哈希表对比

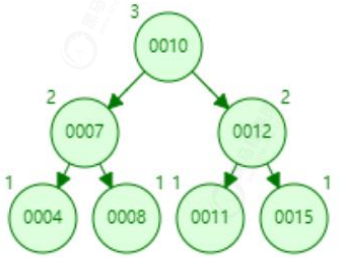
内存占用



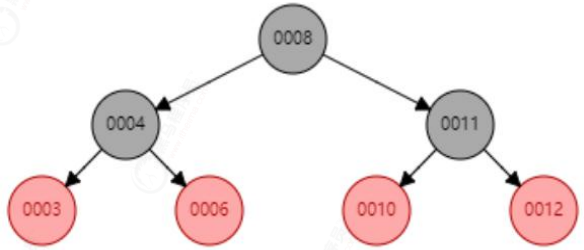
对内存有一定要求



bst

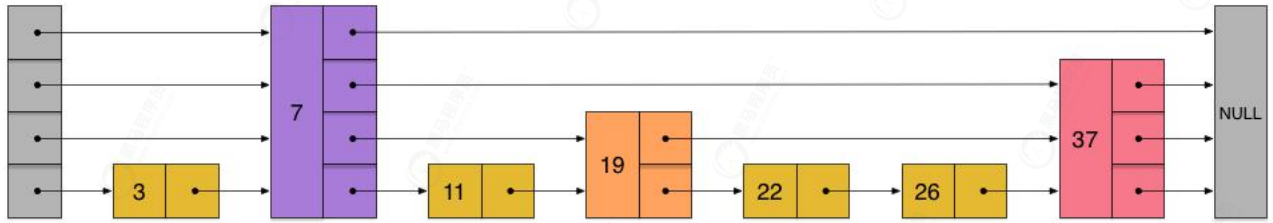


avl



red-black

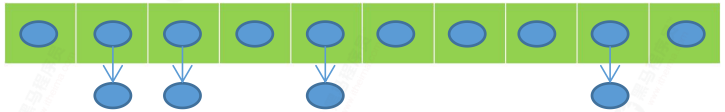
每个节点包含2个指针



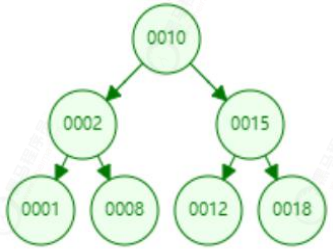
每个节点包含的指针数目平均为1/(1-p)，具体取决于参数p的大小。
如果像Redis里的实现一样，取p=1/4，那么平均每个节点包含1.33个指针，比平衡树更有优势

skiplist，平衡树，哈希表对比

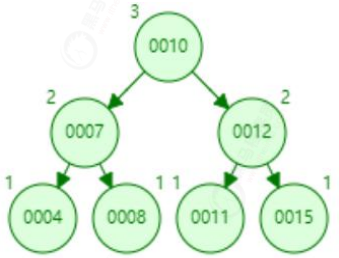
算法实现



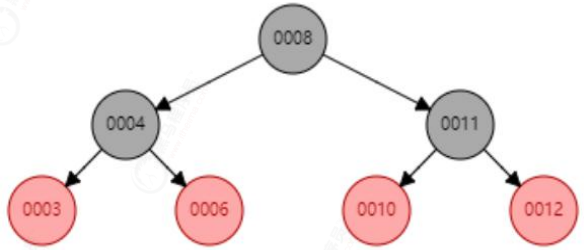
hash函数不太好设计



bst

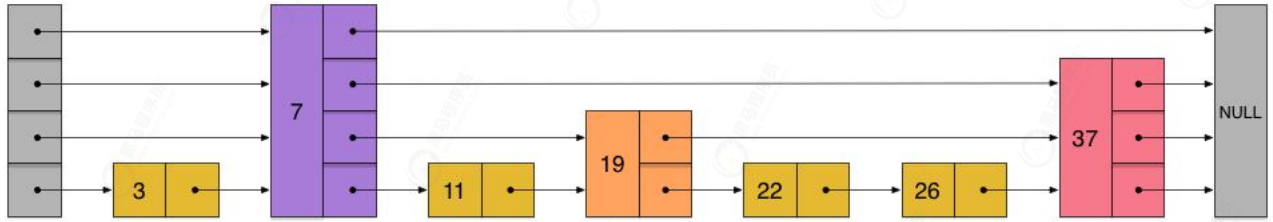


avl



red-black

平衡树的自平衡过程相对复杂



算法实现相对简单

实战题目-1206. 设计跳表

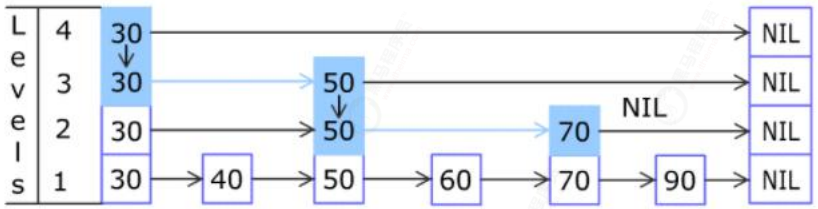
1206. 设计跳表

难度 困难 51

不使用任何库函数，设计一个跳表。

跳表是在 O(log(n)) 时间内完成增加、删除、搜索操作的数据结构。跳表相比于树堆与红黑树，其功能与性能相当，并且跳表的代码长度相较下更短，其设计思想与链表相似。

例如，一个跳表包含 [30, 40, 50, 60, 70, 90]，然后增加 80、45 到跳表中，以下图的方式操作：



Artyom Kalinin [CC BY-SA 3.0], via Wikimedia Commons

跳表中有很多层，每一层是一个短的链表。在第一层的作用下，增加、删除和搜索操作的时间复杂度不超过 O(n)。跳表的每一个操作的平均时间复杂度是 O(log(n))，空间复杂度是 O(n)。

```
class Node{
    //关键字
    int key;
    /*
        表示当前节点在当前层的下一个节点
        next[0] 就是当前节点在第1层的下一个节点
        next[2] 就是当前节点在第3层的下一个节点
    */
    Node[] next;
    public Node(int key,int level) {
        this.key = key;
        next = new Node[level];
    }
}
```

```
class Skiplist {
    int MAX_LEVEL = 32;
    float p = 1/4;
    Random random;
    Node head;//头节点
    int levelCount;//最高层数
    public Skiplist() {
        this.head = new Node(-1,MAX_LEVEL);
        random = new Random();
    }
```

查找/添加

```
public boolean search(int target) {
    Node p = head;
    //从最高层开始查找
    for (int i=levelCount-1;i>=0;i--) { // i--代表移动到下一层
        while (p.next[i] != null && p.next[i].key < target) {
            p = p.next[i];
        }
    }
    //target都在第1层,判断是否找到
    if (p.next[0] != null && p.next[0].key == target) {
        return true;
    }
    return false;
}
```

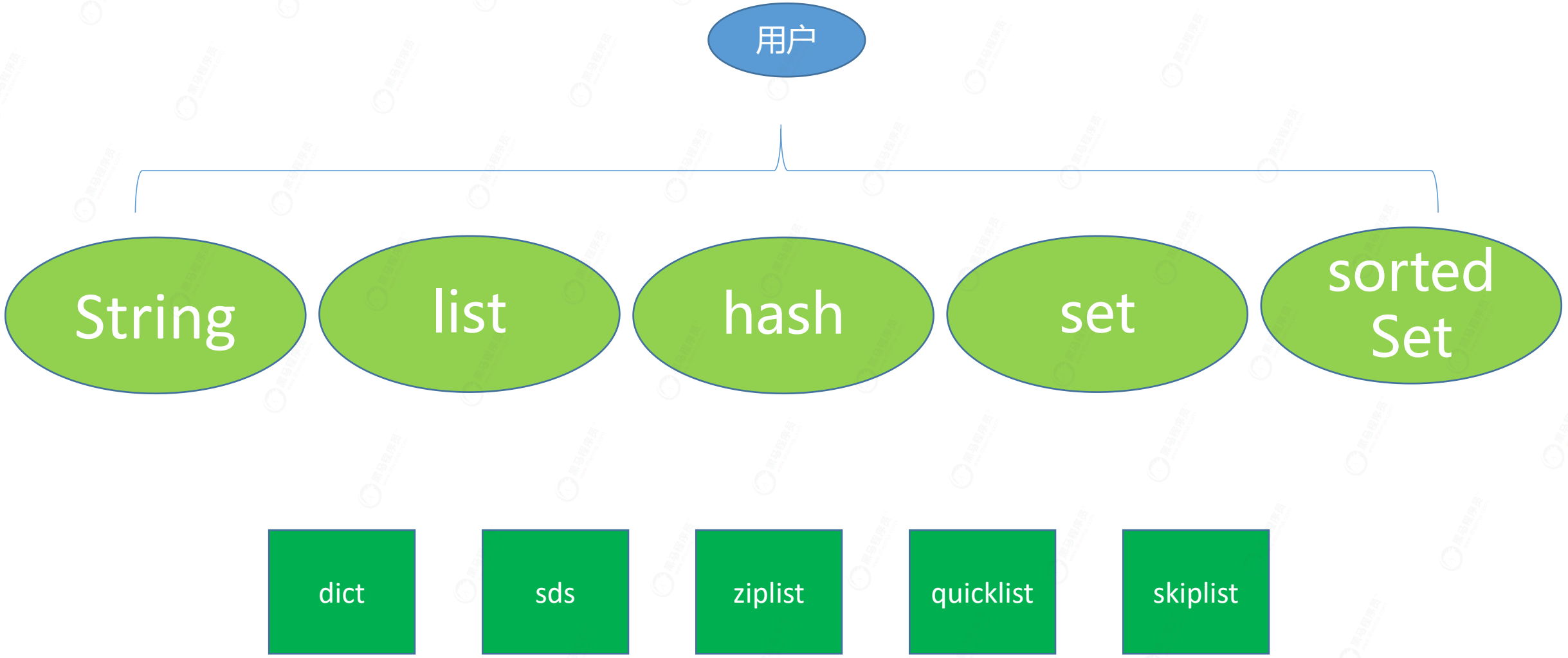
```
//需要一个函数产生随机层数
public int randomLevel() {
    int level = 1;//第一层是必须的
    while (random.nextInt() < p && level < MAX_LEVEL) {
        level++;
    }
    return level;
}
```

```
public void add(int num) {
    //定义新元素应该占几层
    int level = head.next[0] == null ? 1 : randomLevel();
    //如果随机level超过目前最大层数,意味着要上涨很多层，我们选择每次只上涨一层
    if (level > levelCount) {
        level = ++levelCount;
    }
    //创建新节点
    Node newNode = new Node(num,level);
    //从最高层开始查找 找到新节点要插入的位置
    Node p = head;
    for (int i=levelCount-1;i>=0;i--) {
        while (p.next[i] != null && p.next[i].key < num) {
            p = p.next[i];
        }
        if (i < level) { //证明从这层往下,到最底层;某些节点后需要添加一个新节点newNode
            if (p.next[i] == null) {
                //直接在后面接上新节点
                p.next[i] = newNode;
            }else {
                //在当前节点和当前节点后面节点的中间插入新节点
                Node next = p.next[i];
                p.next[i] = newNode;
                newNode.next[i] = next;
            }
        }
    }
}
```

删除参考

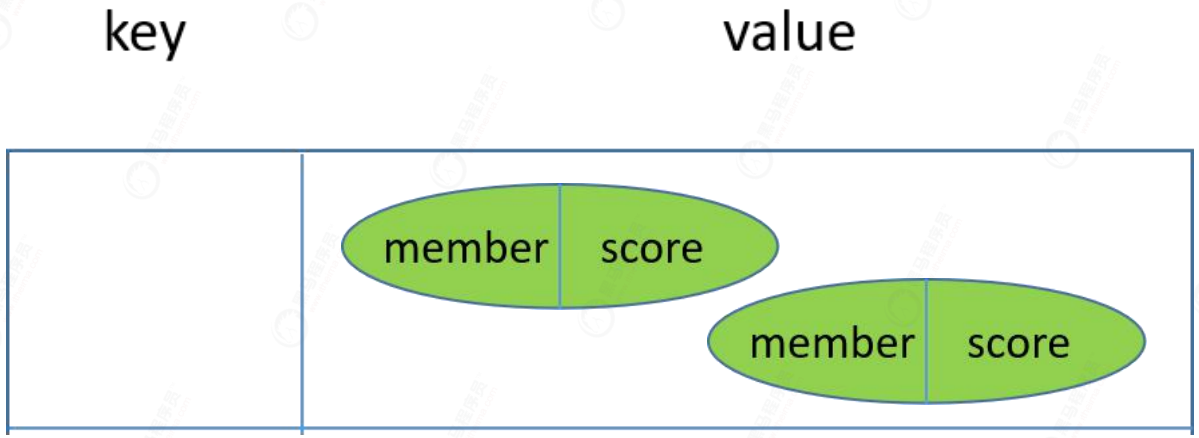
```
public boolean erase(int num) {
    boolean exist = false;
    Node p = head;
    for (int i=levelCount-1;i>=0;i--) {
        while (p.next[i] !=null && p.next[i].key < num) {
            p = p.next[i];
        }
        if (p.next[i] != null && p.next[i].key == num) {
            exist = true;
            p.next[i] = p.next[i].next[i];
        }
    }
    return exist;
}
```

Redis中的数据结构



redis中的sorted set

https://www.redis.net.cn/tutorial/3512.html



- 当数据较少时，sorted set是由一个ziplist来实现的

  ziplist是一个经过特殊编码的双向链表，它的设计目标就是为了提高存储效率

- 当数据多的时候，sorted set是由一个dict + 一个skiplist来实现的

  dict保存member到socre的映射

  skiplist用分数做key，方便按分数查询（如范围查询）