SkipList

初始的想法

在每一层我们每隔一个元素取一个为索引上升到上一层，构成一个带索引的有序链表，这样增删查的时间复杂度都为O(logn)，空间复杂度为O(n) 。 另外，我们规定最底下一层为第0层，亦即第0级索引，往上依次为第1级索引，第2级索引，……

随机化算法改进

我们发现上面的想法难以维护有大量元素插入或删除的情况。如果放任不管，那么增删查的效率会变低；如果要继续维护上面提到的性质，那么维护成本会很大，增删的效率还是很低。因此我们决定使用随机化的方法来决定一个节点最高能够上升到哪一层。
随机化的具体实现：

```
const int S=0xffff;
const int P=4;
/*没有索引的概率为3/4，最高索引为一级索引的概率为3/16，
最高索引为二级索引的概率为9/64……*/
const int SP=S/P;
int randLevel(){
    int l=1;
    while(rand()&S<SP&&l<MaxLevel)l++;
    return l;
}
```

怎么表示跳表及其节点

比较难理解的是固定有两个哨兵节点，一个头节点，一个尾节点。头节点指向尾节点，要求头节点在每层都有索引。二者的键值都为MAX_INT(或相应键值的类型的最大值)，每次查找都从头节点开始，当然，但第一个比较的不是头节点本身，而是其下一个元素。

增删查

1. 查找：从最目前高一层开始，找到当前层小于目标键值的最后一个也即是最大的一个元素，并从该级索引往下一级，继续上述操作，直至最底层。 代码如下：

```
    V find(K key){
        NODE* p=head;
        for(int i=level;i>=0;i--){
            while(key>p->forward[i]->key)p=p->forward[i];
        }
        p=p->forward[0];//Do not forget this
        if(p->key==key)return p->val;
```

```
        else return INVALID;
    }
```

2. 插入：与插入一样，先找到在最底层的插入位置，然后通过随机函数确定新节点的最高的索引层次，然后一层一层地插入索引节点。注意1要用update数组记录要插入的位置2要维护整个跳表的最大高度与长度。

```
    void insert(const K &key,const V &val)  {
        NODE* update[level+1];
        NODE* p=head;
        for(int i=level;i>=0;i--){//The lowest level is 0!
            while(key>p->forward[i]->key)p=p->forward[i];//保证能在tail之前停下来
            update[i]=p;
        }
        p=p->forward[0];//Do not forget this!
        if(p->val==val){
            p->val=val;
            return;
        }
        int lv=randLevel();
        if(lv>level){//DO NOT FORGET THIS!
            lv=++level;
            update[lv]=head;
        }
        NODE *newNode=new NODE(key,val,lv);
        for(int i=0;i<=lv;i++){
            p=update[i];
            newNode->forward[i]=p->forward[i];
            p->forward[i]=newNode;
        }
        length++;
    }
```

3. 删除：与插入类似，自己去看完整程序

时空复杂度

见oiwiki

跳表的应用

①跳表与红黑树一样支持增删查操作，空间时间复杂度相近。但是跳表支持高效的区间查找，这是红黑树没办法做到的。比如Redis就是使用了跳表。
②摘自博客：

> HBase MemStore 内部存储数据就使用的跳表。为什么呢？HBase 属于 LSM Tree 结构的数据库，LSM Tree 结构的数据库有个特点，实时写入的数据先写入到内存，内存达到阈值往磁盘 flush 的时候，会生成类似于 StoreFile 的有序文件，而跳表恰好就是天然有序的，所以在 flush 的时候效率很高，而且跳表查找、插入、删除性能都很高，这应该是 HBase MemStore 内部存储数据使用跳表的原因之一。HBase

> 使用的是 java.util.concurrent 下的 ConcurrentSkipListMap()。 Google 开源的 key/value 存储引擎 LevelDB 以及 Facebook 基于 LevelDB 优化的 RocksDB 都是 LSM Tree 结构的数据库，他们内部的 MemTable 都是使用了跳表这种数据结构。