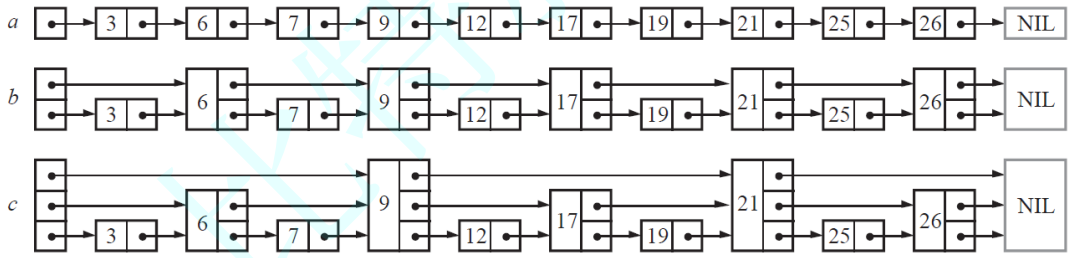
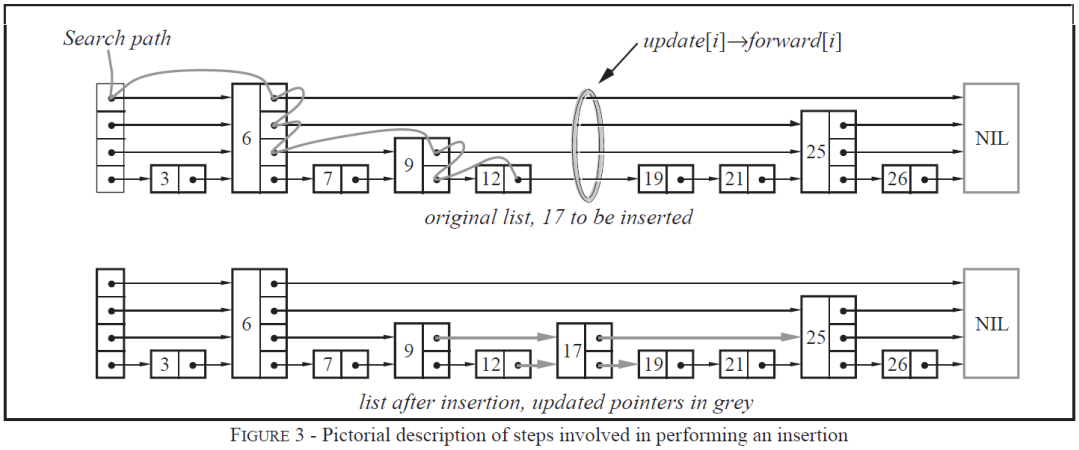
Lessson05---跳表

1.什么是跳表-skiplist

> skiplist本质上也是一种查找结构，用于解决算法中的查找问题，跟平衡搜索树和哈希表的价值是一样的，可以作为key或者key/value的查找模型。那么相比而言它的优势是什么的呢？这么等我们学习完它的细节实现，我们再来对比。
>
> 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》。对细节感兴趣的同学可以下载论文原文来阅读。
>
> skiplist，顾名思义，首先它是一个list。实际上，它是在有序链表的基础上发展起来的。如果是一个有序的链表，查找数据的时间复杂度是O(N)。
>
> William Pugh开始的优化思路：
>
> 1. 假如我们**每相邻两个节点升高一层，增加一个指针，让指针指向下下个节点**，如下图b所示。这样所有新增加的指针连成了一个新的链表，但它包含的节点个数只有原来的一半。由于新增加的指针，我们不再需要与链表中每个节点逐个进行比较了，需要比较的节点数大概只有原来的一半。
> 2. 以此类推，我们可以在第二层新产生的链表上，继续为每相邻的两个节点升高一层，增加一个指针，从而产生第三层链表。如下图c，这样搜索效率就进一步提高了。
> 3. skiplist正是受这种多层链表的想法的启发而设计出来的。实际上，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**类似二分查找**，使得查找的时间复杂度可以降低到**O(log n)**。但是这个结构在插入删除数据的时候有**很大的问题**，插入或者删除一个节点之后，就会打乱上下相邻两层链表上节点个数严格的2:1的对应关系。如果要维持这种对应关系，就必须把新插入的节点后面的所有节点（也包括新插入的节点）重新进行调整，这会让时间复杂度重新蜕化成O(n)。
>
> 
>
>
> 4. skiplist的设计为了避免这种问题，做了一个大胆的处理，不再严格要求对应比例关系，而是插入一个节点的时候随机出一个层数。这样每次插入和删除都不需要考虑其他节点的层数，这样就好处理多了。细节过程入下图：
>
> 
>
> FIGURE 3 - Pictorial description of steps involved in performing an insertion

2.skiplist的效率如何保证?

上面我们说到，skiplist插入一个节点时随机出一个层数，听起来怎么这么随意，如何保证搜索时的效率呢？

这里首先要细节分析的是这个随机层数是怎么来的。一般跳表会设计一个最大层数maxLevel的限制，其次会设置一个多增加一层的概率p。那么计算这个随机层数的伪代码如下图：

```
randomLevel()
    lvl := 1
    -- random() that returns a random value in [0...1)
    while random() < p and lvl < MaxLevel do
        lvl := lvl + 1
    return lvl
```

FIGURE 5 - Algorithm to calculate a random level

在Redis的skiplist实现中，这两个参数的取值为：

```
p = 1/4
maxLevel = 32
```

根据前面randomLevel()的伪码，我们很容易看出，产生越高的节点层数，概率越低。定量的分析如下：

- 节点层数至少为1。而大于1的节点层数，满足一个概率分布。
- 节点层数恰好等于1的概率为1-p。
- 节点层数大于等于2的概率为p，而节点层数恰好等于2的概率为p(1-p)。
- 节点层数大于等于3的概率为p^2，而节点层数恰好等于3的概率为p^2*(1-p)。
- 节点层数大于等于4的概率为p^3，而节点层数恰好等于4的概率为p^3*(1-p)。
- ……

因此，一个节点的平均层数（也即包含的平均指针数目），计算如下：

$$1\times(1-p)+2p(1-p)+3p^2(1-p)+4p^3(1-p)+\ldots=(1-p)\sum_{k=1}^{+\infty}kp^{k-1}=(1-p)\cdot\frac{1}{(1-p)^2}=\frac{1}{1-p}$$

现在很容易计算出：

- 当p=1/2时，每个节点所包含的平均指针数目为2；
- 当p=1/4时，每个节点所包含的平均指针数目为1.33。

跳表的平均时间复杂度为O(logN)，这个推导的过程较为复杂，需要有一定的数据功底，有兴趣的老铁，可以参考以下文章中的讲解：

铁蕾大佬的博客：http://zhangtielei.com/posts/blog-redis-skiplist.html

William_Pugh大佬的论文：ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf

3.skiplist的实现

https://leetcode.cn/problems/design-skiplist/

```
struct SkiplistNode
```

```
{
    int _val;
    vector<SkiplistNode*> _nextV;

    SkiplistNode(int val, int level)
        :_val(val)
        , _nextV(level, nullptr)
    {}
};

class Skiplist {
    typedef SkiplistNode Node;
public:
    Skiplist() {
        srand(time(0));

        // 头节点，层数是1
        _head = new SkiplistNode(-1, 1);
    }

    bool search(int target) {
        Node* cur = _head;
        int level = _head->_nextV.size() - 1;
        while (level >= 0)
        {
            // 目标值比下一个节点值要大，向右走
            // 下一个节点是空(尾)，目标值比下一个节点值要小，向下走
            if (cur->_nextV[level] && cur->_nextV[level]->_val < target)
            {
                // 向右走
                cur = cur->_nextV[level];
            }
            else if (cur->_nextV[level] == nullptr || cur->_nextV[level]-
>_val > target)
            {
                // 向下走
                --level;
            }
            else
            {
                return true;
            }
        }

        return false;
    }

    vector<Node*> FindPrevNode(int num)
    {
        Node* cur = _head;
        int level = _head->_nextV.size() - 1;

        // 插入位置每一层前一个节点指针
        vector<Node*> prevV(level + 1, _head);

        while (level >= 0)
        {
            // 目标值比下一个节点值要大，向右走
```

```
            // 下一个节点是空(尾)，目标值比下一个节点值要小，向下走
            if (cur->_nextV[level] && cur->_nextV[level]->_val < num)
            {
                // 向右走
                cur = cur->_nextV[level];
            }
            else if (cur->_nextV[level] == nullptr
                || cur->_nextV[level]->_val >= num)
            {
                // 更新level层前一个
                prevV[level] = cur;

                // 向下走
                --level;
            }
        }

        return prevV;
    }

    void add(int num) {
        vector<Node*> prevV = FindPrevNode(num);

        int n = RandomLevel();
        Node* newnode = new Node(num, n);

        // 如果n超过当前最大的层数，那就升高一下_head的层数
        if (n > _head->_nextV.size())
        {
            _head->_nextV.resize(n, nullptr);
            prevV.resize(n, _head);
        }

        // 链接前后节点
        for (size_t i = 0; i < n; ++i)
        {
            newnode->_nextV[i] = prevV[i]->_nextV[i];
            prevV[i]->_nextV[i] = newnode;
        }

       // Print();
    }

    bool erase(int num) {
        vector<Node*> prevV = FindPrevNode(num);

        // 第一层下一个不是val，val不在表中
        if (prevV[0]->_nextV[0] == nullptr || prevV[0]->_nextV[0]->_val !=
num)
        {
            return false;
        }
        else
        {
            Node* del = prevV[0]->_nextV[0];
            // del节点每一层的前后指针链接起来
            for (size_t i = 0; i < del->_nextV.size(); i++)
            {
```

```
                prevV[i]->_nextV[i] = del->_nextV[i];
            }
            delete del;

            // 如果删除最高层节点，把头节点的层数也降一下
            int i = _head->_nextV.size()-1;
            while(i >= 0)
            {
                if(_head->_nextV[i] == nullptr)
                    --i;
                else
                    break;
            }
            _head->_nextV.resize(i+1);

            return true;
        }
    }

    int RandomLevel()
    {
        size_t level = 1;
        // rand() ->[0, RAND_MAX]之间
        while (rand() <= RAND_MAX*_p && level < _maxLevel)
        {
            ++level;
        }

        return level;
    }

    /*int RandomLevel()
    {
        static std::default_random_engine
generator(std::chrono::system_clock::now().time_since_epoch().count());
        static std::uniform_real_distribution<double> distribution(0.0,
1.0);

        size_t level = 1;
        while (distribution(generator) <= _p && level < _maxLevel)
        {
            ++level;
        }

        return level;
    }*/

    void Print()
    {
        /*int level = _head->_nextV.size();
        for (int i = level - 1; i >= 0; --i)
        {
            Node* cur = _head;
            while (cur)
            {
                printf("%d->", cur->_val);
                cur = cur->_nextV[i];
            }
```

```
            printf("\n");
        }*/

        Node* cur = _head;
        while (cur)
        {
            printf("%2d\n", cur->_val);
            // 打印每个每个cur节点
            for (auto e : cur->_nextV)
            {
                printf("%2s", "↓");
            }
            printf("\n");

            cur = cur->_nextV[0];
        }
    }

private:
    Node* _head; // 头节点
    size_t _maxLevel = 32;
    double _p = 0.25;
};
```

4.skiplist跟平衡搜索树和哈希表的对比

1. skiplist相比平衡搜索树(AVL树和红黑树)对比，都可以做到遍历数据有序，时间复杂度也差不多。skiplist的优势是：a、skiplist实现简单，容易控制。平衡树增删查改遍历都更复杂。b、skiplist的额外空间消耗更低。平衡树节点存储每个值有三叉链，平衡因子/颜色等消耗。skiplist中p=1/2时，每个节点所包含的平均指针数目为2；skiplist中p=1/4时，每个节点所包含的平均指针数目为1.33；
2. skiplist相比哈希表而言，就没有那么大的优势了。相比而言a、哈希表平均时间复杂度是O(1)，比skiplist快。b、哈希表空间消耗略多一点。skiplist优势如下：a、遍历数据有序 b、skiplist空间消耗略小一点，哈希表存在链接指针和表空间消耗。c、哈希表扩容有性能损耗。d、哈希表再极端场景下哈希冲突高，效率下降厉害，需要红黑树补足接力。