山东大学计算机科学与技术学院

数据结构与算法课程设计报告

学号：202000130143	姓名： 郑凯饶	班级： 20.1
上机学时： 6		日期：2022-3-24
课程设计题目：跳表的实现与分析		
软件环境：Windows 10 家庭中文版 64 位（10.0，版本 18363） Microsoft VS Code cmake version 3.23.0-rc3		

报告内容：

1.需求描述

1.1 问题描述

实现并分析跳表。

1.2 基本要求

1. 构造并实现跳表 ADT，跳表 ADT 中应包括初始化、查找、插入、删除 指定关键字的元素、删除关键字最小的元素、删除关键字最大的元素等 基本操作。 2. 生成测试数据并验证你所实现的跳表结构的正确性。 3. 分析各基本操作的时间复杂性。 4. 对跳表维护动态数据集合的效率进行实验验证。

1.3 输入说明

输入界面设计

无（重定向至输入文件，并重定向输出和标准输出对比）

输入样例

1.4 输出说明

输出界面设计

无（重定向输出程序的运行情况）

输出样例

2.分析与设计

2.1 问题分析

跳表是由有序链表拓展而来，利用随机函数来实现“平衡性”，增删查改的操作复杂度和平衡树相同。和链表相比，插入时需要进行节点定级,插删时要处理多级链表指针。

2.2 主程序设计

```
// O(n)
void calXor(skipList<int, int> &a) {…

// O(1)
void outMin(skipList<int, int> &a) {…

// O(n)
void outMax(skipList<int, int> &a) {…

void solve() {…

/* 利用所给数据测试正确性 */
void testForCorrection() {…

/* 随机增删查改测试 */
void testForCapability() {…

/*
重构： 将有序序列初始化为理想跳表
*/
```

主程序包含测试正确性的 testForCorrection() 及测试性能的 testForCapability()，以及利用跳表特定功能的函数。

2.3 设计思路

通过面向对象的程序设计思维，设计并实现类 SkipList，并对其进行性能和正确性的测试。

2.4 数据及数据类(型)定义

跳表节点定义：

```
template <class K, class E>
struct skipNode
{
    typedef pair<K, E> pairType;

    pairType ele;
    skipNode<K, E> **nxt;
    skipNode(const pairType &thePair, int sz)
        :ele(thePair) { nxt = new skipNode<K, E>* [sz]; }
};
```

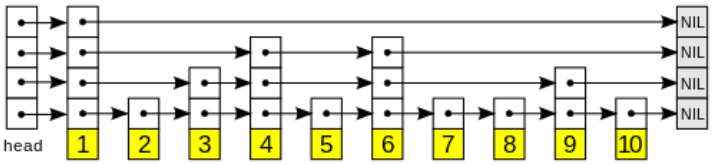
跳表定义：

```
template <class K, class E>
class skipList
{
    public:
        skipList(K, int, float);
        pair<K, E>* find(const K& theKey) const;                // O(logn)
        int level() const;
        skipNode<K, E>* search(const K& theKey) const;          // O(logn)
        void insert(const pair<K, E>& thePair);                 // O(logn)
        void erase(const K& theKey);                            // O(logn)
        // 链表成员类iterator
        class iterator…

        iterator begin() { return iterator(head); }
        iterator end() { return iterator(tail); }
    private:
        float cutOff;
        int levels, dSize, maxLevel;
        K tailKey;                  // 最大关键字
        skipNode<K, E>* head;
        skipNode<K, E>* tail;
        skipNode<K, E>** last;      // last[i]表示第i层的最后节点
};
```

2.5. 算法设计及分析

Search()函数的实现：

```
template <class K, class E>
skipNode<K, E>* skipList<K, E>::search(const K& theKey) const
{
    // 搜索theKey同时将每一级要查看的最后一个节点存至last
    skipNode<K, E>* pre = head;
    for (int i = levels; i >= 0; i--) {
        while (pre->nxt[i]->ele.first < theKey)
            pre = pre->nxt[i];
        last[i] = pre;
    }

    return pre->nxt[0];
}
```



从高级链表向下查找，并保存每一级链表最后访问过的节点，最后返回并查找节点小的最大节点，辅助后面插删操作。

Insert()函数：

```cpp
template <class K, class E>
void skipList<K, E>::insert(const pair<K, E>& thePair)
{
    if (thePair.first >= tailKey) {
        // ostringstream s;
        // s << "Key = " << thePair.first << " Must be < " << tailKey;
        return;
    }

    skipNode<K, E>* theNode = search(thePair.first);
    if (theNode->ele.first == thePair.first) {
        theNode->ele.second = thePair.second;
        return;
    }

    // 不存在
    int theLevel = level();
    if (theLevel > levels) {
        theLevel = ++levels;        // 增长1级
        last[theLevel] = head;      // 最高级中最后访问的是头结点
    }

    // 在theNode之后插入新节点
    skipNode<K, E>* newNode = new skipNode<K, E>(thePair, theLevel + 1);
    for (int i = 0; i <= theLevel; i++) {
        newNode->nxt[i] = last[i]->nxt[i];
        last[i]->nxt[i] = newNode;
    }

    dSize++;
    return;
}
```

先判断 search()返回的下一节点是否是查找节点，有则进行替换直接返回。无则 new 一个节点，定级，在查找过程中每一级链表最后访问的节点后插入新节点。

3. 测试

运行法官程序 localJudge.cpp 将 myoutputx.txt 和标准输出 outputx.txt 进行比对：

```
The result is as follows:
Test point 1 : Accept
Test point 2 : Accept
Test point 3 : Accept
Test point 4 : Accept
Test point 5 : Accept
Test point 6 : Accept
Test point 7 : Accept
Test point 8 : Accept
Test point 9 : Accept
Test point 10 : Accept
```

性能测试：

使用 1e8 级别的数据进行性能评估，发现跳表性能随使用次数的增加逐渐下降。

```
Round 1 time cost 39288 ms
Round 2 time cost 39278 ms
Round 3 time cost 39280 ms
Round 4 time cost 40744 ms
Round 5 time cost 41828 ms
Round 6 time cost 40427 ms
Round 7 time cost 42333 ms
Round 8 time cost 40222 ms
Round 9 time cost 40168 ms
Round 10 time cost 40962 ms
```

4. 分析与探讨

跳表是一种十分巧妙的数据结构，利用随机函数进行“平衡”，相比于理想跳表，免去动态维护严格结构的麻烦。在区间查找上甚至更优于红黑树，找到最小值后，对第 1 级链表进行若干步遍历即可。这也使得 redis 数据库在有序集合 zset 的实现上采用了它。

基于随机性，在长期使用后性能难免下降，这时我们可以通过将其重构为理想链表的方式优化性能。

这次我使用的测试方式比较简单，希望之后可以在此基础上进行升级，最好能封装成一个鲁棒的测试类使用，可以应对对不同程序的性能及正确性测试。

5. 附录：实现源代码

（另附）