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：201700140056	姓名：李港	班级：17.4 班
上机学时：2h	日期：2020.03.04	
课程设计题目：跳表的实现与分析		
软件开发环境：Visual Studio 2019		

报告内容：

一. 需求描述

1. 问题描述

   实现并分析跳表结构

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. 分析各基本操作的时间复杂性。
   3. 利用相关应用示例（自选）完成基本功能的实现演示。
   4. 能对跳表维护动态数据集合的效率进行实验验证，获得一定量的实验数据，如给定随机产生 1000 个数据并将其初始化为严格跳表，在此基础上进行一系列插入、删除、查找操作（操作序列也可以随机生成），获得各种操作的平均时间（或统计其基本操作个数）；获得各操作执行时间的变化情况。

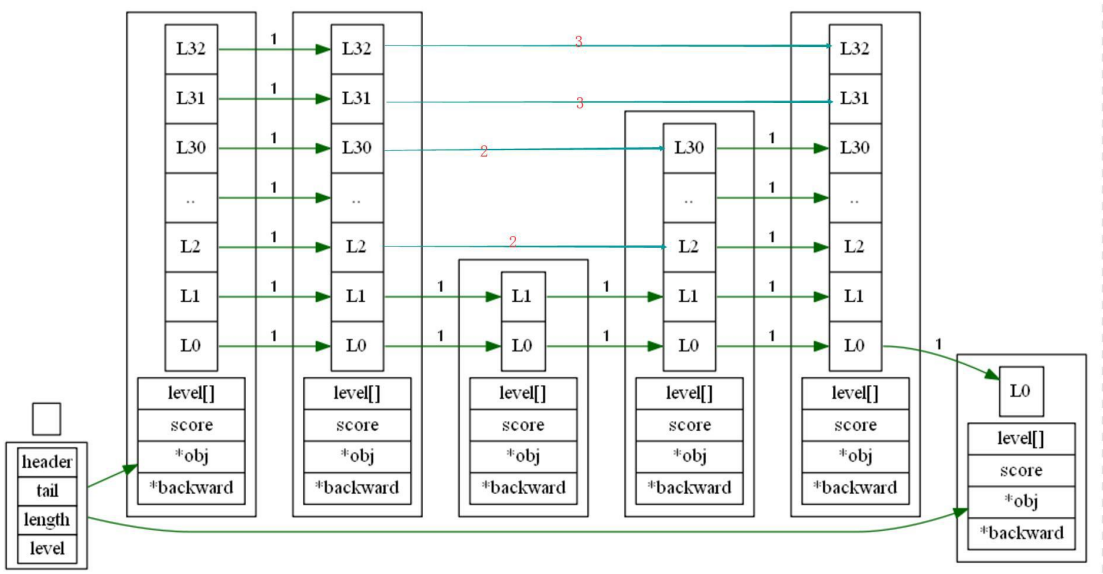
3. 输入输出说明

   1. 输入包含 1+n+m 行数据
   2. 第一行包含两个数字 n, m
   3. 之后的 n 行，每行包含一个数字与一个字符串，表示一个元素，数字为跳表的关键字，字符串为元素的值。
   4. 之后的 m 行表示 m 个跳表 ADT 操作。每行第一个数字 a 用来区别操作。
      1. a 为 1 时，表示查找操作， b 为要查询的元素的关键字， 输出 该元素的值，不存在则输出 -1。
      2. a 为 2 时，表示插入操作， b 为要插入的元素，一个字符串 c 为对应的值， 不输出 。
      3. a 为 3 时，表示删除操作， b 为要删除的关键字， 输出 该关键字与其对应的元素值，若删除关键字不存在或删除失败，输出 -1。
      4. a 为 4 时，表示删除最小元素操作。 输出 跳表内最小关键字 c 与其对应的元素值，并删除
      5. a 为 5 时，表示删除最大元素操作。 输出 跳表内最大关键字 c 与其对应的元素值，并删除

二. 设计

1. 设计思路

   1. 本跳表设计参考内存数据库 Redis,一大特色是添加 backward 指针,使跳表形成双向链表。
   2. 节点，指针数组，跳表 均由单独的类来表示。
   3. 跳表提供根据关键字搜索内容，插入，删除，范围删除，获取最大最小关键字等功能。
   4. 跳表的高度是由随机函数确定的，为了使其高度分布能够形成类似二叉树的状态，我们的随机数函数设计为特定高度出现频率与其高度值呈负指数关系，高度越高，出现次数越少。
   5. 本文设计的跳表内存布局类似下图：



2. 数据及数据类(型)定义

1. 跳表类

```
template<typename K, typename E>
class skiplist {
public:
    static const int _SKIPLIST_MAXLEVEL = 32;   //最大层数
protected:
    skiplistNode<K, E>* _header;//指向空头
    skiplistNode<K, E>* _tail;//指向有数据的尾部
    unsigned long _length;// 节点数量
    int _level;// 目前表内节点的最大层数
    int _getRandomLevel () ;
    skiplistNode<K, E>* _createNode (int  level, K score, E ele);
    skiplistNode<K, E>* _createNode (int  level);
    void _skiplistDeleteNode (skiplistNode<K, E>* x, skiplistNode<K, E>** update);

public:
    explicit skiplist ();    //构造函数
    bool empty ();          //返回是否为空
    K* getMinScore ();       //获取最大关键词
    K* getMaxScore ();       //获取最小关键词
    skiplistNode<K, E>* insert (double score, E data);//插入
    unsigned long deleteRangeByRank (unsigned long long start, unsigned long long end);
    bool deleteByScore (double score);
    E* getDataByScore (double score);

    ostream& output (ostream& out);
    friend ostream& operator <<(ostream& out, skiplist<K, E>& item);
}
```

2. 跳表节点类

```
template<typename K, typename E>
class skiplistNode {
public:
    E data;     // 值
    K score;    // 关键词
    skiplistNode<K, E>* backward;       // 后退指针
    skiplistLevel<K, E> level[_SKIPLIST_MAXLEVEL];  // 各层指针
};
```

3. 跳表节点指针数组类

```
template<typename K, typename E>
class skiplistLevel {
public:
    skiplistNode<K, E>* forward;  // 前进指针
    unsigned long long span;      // 节点在该层和前向节点的距离。可用于获取元素排名
};
```

3. 算法设计及分析（各模块算法及类内函数的算法伪码表示）

1. 搜索：

1. 跳表为空，关键词大于最大值，小于最小值 这三种情况都会直接返回空指针。

2. 取当前节点 x 初值为头结点，取当前层数 i 为最高层索引数。
3. 在 x 节点的 i 层不断向后遍历，遇到为空或节点关键词大于等于目标关键词则进行下一层判断
    1. 若关键词等于目标关键词则返回节点数据
    2. 否则在当前节点处下降一层。
4. 核心代码：

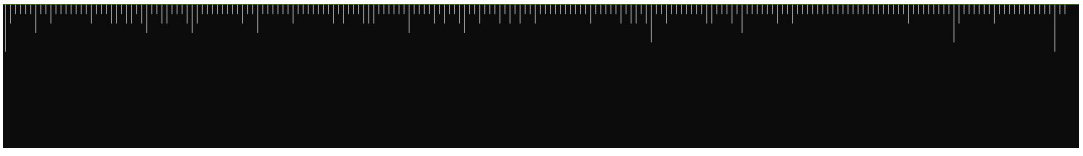
```
for (i = this->_level - 1; i >= 0; i--) {
    while (x->level[i].forward && x->level[i].forward->score <= score) {
        rank += x->level[i].span;
        //排名累加
        x = x->level[i].forward;
    }

    if (x->score == score) {
        return &x->data;
    }
}
```

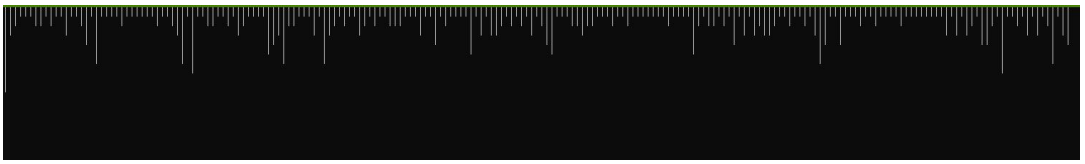
2. 插入：插入操作较为复杂，但比红黑树简单多了
    1. 寻找目标位置，类似搜索，不过要保存目标位置的所有前置节点。
    2. 获取随机高度
    3. 如果新高度大于跳表已有高度，则对前置节点表进行修补，所有现有高度与新高度范围之间的部分都要将前置节点设置为头结点 。
    4. 创建新节点
    5. 将新节点按层插入
        1. 新节点的后继节点设置为前置节点的后继节点
        2. 前置节点的后继节点设置为新节点
        3. 设置新节点的前置节点指针
            1. 若新节点的前置节点是头结点，则指向空，而不是指向头结点
            2. 反之指向前置节点
    6. 如果新节点有后继节点，则将其后继节点的前置节点指针指向新节点。
3. 删除：删除较为简单，比红黑树的删除简单多了
    1. 先找到目标节点，并统计前置节点
    2. 对前置节点的每一层，将其后继节点指针指向被删除节点的后继节点。
    3. 如果被删除节点有后继节点，则后继节点的前置节点设置为被删除节点。
    4. 否则将尾节点指向被删除节点的前置节点
4. 随机高度：核心代码是 `while (rand () %4)k++;`，这句代码使高度出现几率随值得增加而减小。

三. 测试结果

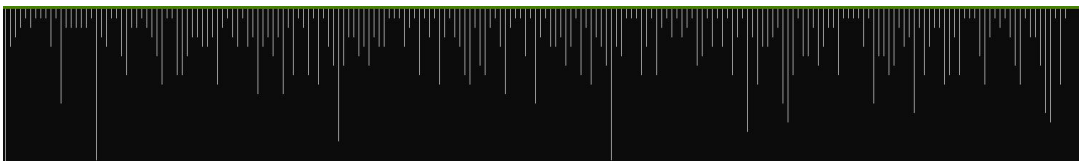
1. 本地动画测试：
    1. p=0.25



    2. p=0.5



3. p=0.75



2. 0J 平台测试

#	状态	耗时	内存占用
#1	✓ Accepted	1ms	344.0 KiB
#2	✓ Accepted	1ms	348.0 KiB
#3	✓ Accepted	1ms	348.0 KiB
#4	✓ Accepted	1ms	348.0 KiB
#5	✓ Accepted	2ms	348.0 KiB
#6	✓ Accepted	2ms	348.0 KiB
#7	✓ Accepted	1ms	512.0 KiB
#8	✓ Accepted	2ms	348.0 KiB
#9	✓ Accepted	2ms	384.0 KiB
#10	✓ Accepted	2ms	348.0 KiB
#11	✓ Accepted	1ms	408.0 KiB
#12	✓ Accepted	1ms	384.0 KiB
#13	✓ Accepted	2ms	512.0 KiB
#14	✓ Accepted	1ms	476.0 KiB
#15	✓ Accepted	2ms	384.0 KiB
#16	✓ Accepted	2ms	464.0 KiB
#17	✓ Accepted	7ms	896.0 KiB
#18	✓ Accepted	4ms	1.0 MiB
#19	✓ Accepted	6ms	896.0 KiB
#20	✓ Accepted	9ms	988.0 KiB

四. 分析与探讨

1. 本代码的妥协：指针数组是固定长度，这是为了简化跳表整理操作，但同时也带来了空间的消耗。
2. 虽然没有红黑树那种繁杂的约束规则，但跳表依然需要精心设计的随机函数；理想跳表应该呈现大山峰套小山峰的分形模样；根据打印的图形来看，本跳表的随机函数设计得不错，比较逼近分形图案。
3. 复杂度分析
    1. 在理想状态下，跳表高度 h = log2n
    2. 在理想状态下，跳表的任何操作都可以总结为一个类似树的搜索操作，复杂度是 O(log2n)
    3. 空间复杂度：
        1. 在本代码中，所有节点都申请了相同长度的指针数组
4. 若优化指针数组所占空间，则总的指针数可以用等比数列求和公式求出。为 n/p，其中 n 代表节点数，p 代表每个节点有多大几率被拔到高层去。可知空间复杂度为 O(n)。

心得体会：

1. 跳表是用来替代红黑树的一种选择，它和红黑树各有优劣。
   1. 如果单纯比较性能，跳跃表和红黑树相差不大
   2. 但是在高并发环境中，跳表的插入删除操作涉及节点较少，加锁少
   3. 红黑树的 rebalance 可能涉及大量操作，需要加很多锁，造成巨大性能损失。
   4. 跳表的范围操作优于红黑树。
2. 不同数据结构有不同的应用场景，我们应牢记其特性，对数据结构进行合理选择。

附录：实现源代码

```
#pragma once
#include <cstdlib>
#include <ctime>
#include<iostream>
using namespace std;


/*
template<typename K, typename E>
class skiplistLevel {
public:
    skiplistNode<K, E>* forward;  // 前进指针
    unsigned long long span;      // 节点在该层和前向节点的距离。可用于获取元素排名
};

template<typename K, typename E>
class skiplistNode {
public:
    E data;     // 值
    K score;    // 关键词
    skiplistNode<K, E>* backward;       // 后退指针
    skiplistLevel<K, E> level[_SKIPLIST_MAXLEVEL];  // 各层指针
};

template<typename K, typename E>
class skiplist {
public:
    static const int _SKIPLIST_MAXLEVEL = 32;   //最大层数
protected:
    skiplistNode<K, E>* _header;//指向空头
    skiplistNode<K, E>* _tail;//指向有数据的尾部
    unsigned long _length;// 节点数量
    int _level;// 目前表内节点的最大层数
    int _getRandomLevel () ;
    skiplistNode<K, E>* _createNode (int  level, K score, E ele);
    skiplistNode<K, E>* _createNode (int  level);
    void _skiplistDeleteNode (skiplistNode<K, E>* x, skiplistNode<K, E>** update);

public:
    explicit skiplist ();    //构造函数
    bool empty ();           //返回是否为空
    K* getMinScore ();       //获取最大关键词
    K* getMaxScore ();       //获取最小关键词
    skiplistNode<K, E>* insert (double score, E data);//插入
    unsigned long deleteRangeByRank (unsigned long long start, unsigned long long end);
    bool deleteByScore (double score);
    E* getDataByScore (double score);

    ostream& output (ostream& out);
    friend ostream& operator <<(ostream& out, skiplist<K, E>& item);
}

*/
template<typename K, typename E>
class skiplist {
public:
    static const int _SKIPLIST_MAXLEVEL = 32;//最大层数，足够用
    static const int _SKIPLIST_P = 0.25;//随机数

    template<typename K, typename E>
    class skiplistNode {
```

```
    public:
        E data;// member 对象
        K score;// 分值
        skiplistNode<K, E>* backward;// 后退指针
        template<typename K, typename E>
        class skiplistLevel {
        public:
            skiplistNode<K, E>* forward; // 前进指针
            unsigned long long span; // 节点在该层和前向节点的距离。可用于获取元素排名
        };
        skiplistLevel<K, E> level[_SKIPLIST_MAXLEVEL];// 层
    };


protected:
    skiplistNode<K, E>* _header;//指向空头
    skiplistNode<K, E>* _tail;//指向有数据的尾部
    unsigned long _length;// 节点数量
    int _level;// 目前表内节点的最大层数


    /*获取随机层数，每个高度出现的概率都指数降低*/
    int _getRandomLevel () {
        int k = 1;
        while (rand () %4)k++;//概率随高度指数降低
        k = (k < _SKIPLIST_MAXLEVEL) ? k : _SKIPLIST_MAXLEVEL-1;
        return k;
    }
    /*根据输入的层数和键值对构建节点*/
    skiplistNode<K, E>* _createNode (int  level, K score, E ele) {
        skiplistNode<K, E>* zn = new skiplistNode<K, E>;
        zn->score = score;
        zn->data = ele;
        return zn;
    }
    /*根据层数构建空节点*/
    skiplistNode<K, E>* _createNode (int  level) {
        skiplistNode<K, E>* zn = new skiplistNode<K, E>;
        return zn;
    }
    /*删除功能的辅助函数*/
    void _skiplistDeleteNode (skiplistNode<K, E>* x, skiplistNode<K, E>** update) {
        long long i;
        for (i = 0; i < this->_level; i++) {
            //如果待更新节点（待删除节点的前一节点）的后继节点是待删除节点，则需要处理待更新
节点的后继指针
            if (update[i]->level[i].forward == x) {
                update[i]->level[i].span += x->level[i].span - 1;

                //这里有可能为 nullptr，比如删除最后一个节点
                update[i]->level[i].forward = x->level[i].forward;

                //待删除节点没有出现在此层--跨度减 1 即可
            } else {
                update[i]->level[i].span -= 1;
            }
        }
        //处理待删除节点的后一节点（如果存在的话）
        if (x->level[0].forward) {
            x->level[0].forward->backward = x->backward;
        } else {
            this->_tail = x->backward;
        }
        //跳跃表的层数处理，如果表头层级的前向指针为空，说明这一层已经没有元素，层数要减一
        while (this->_level > 1 && this->_header->level[this->_level - 1].forward ==
nullptr)
            this->_level--;

        //跳跃表长度减一
        this->_length--;

        delete x;
    }

public:
    explicit skiplist () {
        long long j;

        // 初始化跳跃表属性,层数初始化为 1，长度初始化为 0
        this->_level = 1;
```

```
        this->_length = 0;

        // 创建一个层数为 32，分值为 0，成员对象为 nullptr 的表头结点
        this->_header = _createNode (_SKIPLIST_MAXLEVEL);
        for (j = 0; j < _SKIPLIST_MAXLEVEL; j++) {
            // 设定每层的 forward 指针指向 nullptr
            this->_header->level[j].forward = nullptr;
            this->_header->level[j].span = 0;
        }
        // 设定 backward 指向 nullptr
        this->_header->backward = nullptr;
        this->_tail = nullptr;
    }

    bool empty () { return _length == 0; }

    /*获取最小关键词*/
    K* getMinScore () {
        if (empty ())return nullptr;
        if (_header) {
            if (_header->level[0].forward) {
                return &_header->level[0].forward->score;
            }
        }
        return nullptr;
    }
    /*获取最大关键词*/
    K* getMaxScore () {
        if (empty ())return nullptr;
        if (_tail) {
            return &_tail->score;
        }
        return nullptr;
    }
    /*插入新的键值对*/
    skiplistNode<K, E>* insert (K score, E data) {
        // updata[]数组记录每一层位于插入节点的前一个节点
        skiplistNode<K, E>* update[_SKIPLIST_MAXLEVEL];

        // rank[]记录每一层位于插入节点的前一个节点的排名
        //在查找某个节点的过程中，将沿途访问过的所有层的跨度累计起来，得到的结果就是目标节点
在跳跃表中的排位
        unsigned long long rank[_SKIPLIST_MAXLEVEL];

        long long i, level;

        // 表头节点
        skiplistNode<K, E>* x = this->_header;

        // 从最高层开始查找(最高层节点少，跳越快)
        for (i = this->_level - 1; i >= 0; i--) {
            /* store rank that is crossed to reach the insert position */

            //rank[i]用来记录第 i 层达到插入位置的所跨越的节点总数,也就是该层最接近(小于)给
定 score 的排名
            //rank[i]初始化为上一层所跨越的节点总数,因为上一层已经加过
            rank[i] = i == (this->_level - 1) ? 0 : rank[i + 1];

            //后继节点不为空，并且后继节点的 score 比给定的 score 小
            while (x->level[i].forward &&
                ((x->level[i].forward)->score < score)) {
                //记录跨越了多少个节点
                rank[i] += x->level[i].span;

                //查找下一个节点
                x = x->level[i].forward;
            }
            // 存储当前层上位于插入节点的前一个节点,找下一层的插入节点
            update[i] = x;
        }

         // 此处假设插入节点的成员对象不存在于当前跳跃表内，即不存在重复的节点
         // 随机生成一个 level 值
        level = _getRandomLevel ();


        // 如果 level 大于当前存储的最大 level 值
        // 设定 rank 数组中大于原 level 层以上的值为 0--为什么设置为 0
        // 同时设定 update 数组大于原 level 层以上的数据
        if (level > this->_level) {
```

```
            for (i = this->_level; i < level; i++) {

                //因为这一层没有节点，所以重置 rank[i]为 0
                rank[i] = 0;
                //因为这一层还没有节点，所以节点的前一个节点都是头节点
                update[i] = this->_header;

                // 在未添加新节点之前，需要更新的节点跨越的节点数目自然就是
skiplist->length---因为整个层只有一个头结点----->言外之意头结点的 span 都是链表长度
                update[i]->level[i].span = this->_length;
            }
            // 更新 level 值（max 层数）
            this->_level = level;
        }

        // 创建插入节点
        x = _createNode (level, score, data);
        for (i = 0; i < level; i++) {

            // 针对跳跃表的每一层，改变其 forward 指针的指向
            x->level[i].forward = update[i]->level[i].forward;

            //插入位置节点的后继就是新节点
            update[i]->level[i].forward = x;

            //rank[i]: 在第 i 层，update[i]->score 的排名
            //rank[0] - rank[i]: update[0]->score 与 update[i]->score 之间间隔了几个数

            // A3 ---------------------------- [I] -> F3
            // A2 -----------------> D2 -------- [I] -> F2
            // A1 ---------> C1 --> D1 -------- [I] -> F1
            // A0 --> B0 --> C0 --> D0 --> E0 - [I] -> F0

            //x->level[i].span = 从 x 到 update[i]->forword 的 span 数目，
            //   原   来   的   update[i]->level[i].span   =   从   update[i]   到
update[i]->level[i]->forward 的 span 数目
            // 所以 x->level[i].span = 原来的 update[i]->level[i].span - (rank[0] -
rank[i]);
            x->level[i].span = update[i]->level[i].span - (rank[0] - rank[i]);


            // 对于 update[i]->level[i].span 值的更新由于在 update[i] 与
update[i]->level[i]->forward 之间又添加了 x，
            //update[i]->level[i].span = 从 update[i]到 x 的 span 数目，
            //由于 update[0]后面肯定是新添加的 x，所以自然新的 update[i]->level[i].span =
(rank[0] - rank[i]) + 1;

            //提示： update[i]和 x[i]之间肯定没有节点了
            update[i]->level[i].span = (rank[0] - rank[i]) + 1;
        }

        //另外需要注意当 level > skiplist->level 时，update[i] = skiplist->header 的 span
处理
        // 更新高层的 span 值
        for (i = level; i < this->_level; i++) {
            //因为下层中间插入了 x，而高层没有，所以多了一个跨度
            update[i]->level[i].span++;
        }

        // 设定插入节点的 backward 指针
        //如果插入节点的前一个节点都是头节点，则插入节点的后向指针为 nullptr？
        x->backward = (update[0] == this->_header) ? nullptr : update[0];

        //如果插入节点的 0 层存前向节点则前向节点的后向指针为插入节点
        if (x->level[0].forward)
            x->level[0].forward->backward = x;
        else
            //否则该节点为跳跃表的尾节点
            this->_tail = x;

        // 跳跃表长度+1
        this->_length++;

        //返回插入的节点
        return x;
    }
```

```
    /*通过排名范围删除数据*/
    unsigned long deleteRangeByRank (unsigned long long start, unsigned long long end)
{
        skiplistNode<K, E>* update[_SKIPLIST_MAXLEVEL], * x;
        unsigned long traversed = 0, removed = 0;
        long long i;

        x = this->_header;
        //寻找待更新的节点
        for (i = this->_level - 1; i >= 0; i--) {
            //指针前移的必要条件是前继指针不为空
            while (x->level[i].forward && (traversed + x->level[i].span) < start) {
                //排名累加
                traversed += x->level[i].span;
                x = x->level[i].forward;
            }
            update[i] = x;
        }

        //下面的节点排名肯定大于等于 start
        traversed++;
        x = x->level[0].forward;
        while (x && traversed <= end) {
            //逐个删除后继节点,直到 end 为止
            skiplistNode<K, E>* next = x->level[0].forward;
            _skiplistDeleteNode (x, update);

            removed++;
            //每删除一个节点,排名加 1
            traversed++;
            x = next;
        }
        return removed;
    }

    /*通过关键词删除数据*/
    bool deleteByScore (K score) {
        skiplistNode<K, E>* update[_SKIPLIST_MAXLEVEL], * x;
        long long i;

        x = this->_header;
        // 遍历所有层，记录删除节点后需要被修改的节点到 update 数组
        for (i = this->_level - 1; i >= 0; i--) {
            //指针前移首要条件是前向节点指针不为空，次要条件是分数小于指定分数，或即使分数相
等，节点成员对象也不相等------> 前向指针前移的必要条件：分数小于或等于指定分数
            while (x->level[i].forward && //前向指针不为空
                x->level[i].forward->score < score//前向节点分数小于指定分数
                )//前向节点成员对象不相同
                x = x->level[i].forward;
            //保存待删除节点的前一节点指针
            update[i] = x;
        }
        // 因为多个不同的 member 可能有相同的 score
        // 所以要确保 x 的 member 和 score 都匹配时，才进行删除

        x = x->level[0].forward;

        if (x && score == x->score) {
            _skiplistDeleteNode (x, update);
            return true;
        }
        return false; /* not found */
    }
    /*通过关键词获取数据*/
    E* getDataByScore (K score) {
        if (empty ())return nullptr;
        if (score < _header->level[0].forward->score)return nullptr;
        if (_tail && score > _tail->score)return nullptr;

        skiplistNode<K, E>* x;
        unsigned long rank = 0;
        long long i;

        x = this->_header;
        for (i = this->_level - 1; i >= 0; i--) {
            //指针前移的必要条件是后继指针不为空
            while (x->level[i].forward && x->level[i].forward->score <= score) {
                rank += x->level[i].span;
                //排名累加
                x = x->level[i].forward;
```

```
            }

            if (x->score == score) {
                return &x->data;
            }
        }
        return nullptr;
    }
    /*图形化输出跳表*/
    ostream& output (ostream& out) {
        char map[1000][32] = { 0 };
        for (int y = 0; y < 1000; y++) {
            for (int x = 0; x < 32; x++) {
                map[y][x] = ' ';
            }
        }
        skiplistNode<K, E>* p = _header;
        int i = 0;
        while (p) {//
            int j = -1;
            while (p->level[++j].forward) {
                map[i][j] = '|';
            }

            i++;

            p = p->level[0].forward;
        }

        for (int y = 0; y < 30; y++) {
            for (int x = 0; x < 1000; x++) {
                out << map[x][y];
            }
            out << "\n";
        }
        return out;
    }
    /*输出跳表的包装函数*/
    friend ostream& operator <<(ostream& out, skiplist<K, E>& item) {
        item.output (out);
        return out;
    }
};
```