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 课程实验报告

学号：202200400053	姓名： 王宇涵	班级： 2202
上机学时：4	实验日期： 2024-03-25	

课程设计题目：

跳表实现与分析

软件开发环境：

Clion 2023.1.1

报告内容：

1. 需求描述

1.1 问题描述

实现并分析跳表结构。

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 ：

设计并生成一个操作序列，操作序列中包含插入、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、查找操作，可设总共操作的次数是 M 次（操作数据随机生成）；

随机产生 N 个数据并将其初始化为严格跳表；

在以上跳表的基础上，依次执行 M 次操作，统计单个操作序列中各个操作执行所需的平均时间（以元素的比较次数衡量），获得随着 M 增加而导致操作时间的变化情况。分析产生这样变化的原因。当操作时间大到一定程度后应进行跳表的整理操作，设计相应的整理算法，并从数量上确定何时较为合适。观察在添加整理操作后执行时间的变化情况。

1.3 输入说明

第一行两个数 M , N，分别表示操作个数和初始化跳表长度 第二行 N 个数为初始化跳表所用元素，保证没有重复数据。接下来 M-1 行分别为跳表各个操作。

具体操作为:

1 num，查找跳表中是否含有元素 num，含有则输出 YES，否则输入 NO

2 num，向跳表中插入元素 num，并输出跳表中所有元素的异或和

3 num，将跳表中的元素 num 删除，并输出跳表中所有元素的异或和

4，删除跳表中的最小元素，并将该元素输出

5，删除跳表中的最大元素，并将该元素输出

1.4 输出说明

按要求输出，每个操作输出一行

测试数据仅限于评测跳表正确性，由于代码实现不同，具体的每个操作所需时间会有变化，因此没有设置验证，代码可设置各个操作执行所需的元素比较次数的输出用于实验验收时展示。

2. 分析与设计

2.1 问题分析

本实验是针对跳表数据结构的分析和操作，我们首先需要定义和实现跳表的基本操作，包括插入，删除，查找等，此外我们需要分析跳表的性能，通过动态生成操作序列和分析函数运行时间来分析跳表的性能随着操作数目增长而产生的影响.

2.2 主程序设计

我们设计跳表分为 skipList.h, skipList.cpp, main.cpp 三个文件，分别定义了跳表的类，函数具体实现，测试函数，最终通过在测试函数中生成动态操作序列，读入文件，输出文件来分析跳表性能.

2.3 设计思路

主体思路是先实现基础功能，再通过给定的样例进行测试功能的正确性，最终自己实现动态操作序列，并调整输入数据的规模，来观察跳表性能的变化规律.

2.4 数据及数据类型定义

cutOff：浮点数，用于确定跳表的层数。

levers：整数，表示当前最大的非空链表层数。

dSize：整数，表示元素个数。

maxLevel：整数，表示最大层数。

tailKey：模板类型 K，表示最大关键字。

headerNode：指向 skipNode<K> 类型的指针，表示头节点。

tailNode：指向 skipNode<K> 类型的指针，表示尾节点。

last：指向 skipNode<K> 指针的指针数组，last[i] 表示第 i 层链表的最后一个节点。

2.5. 算法设计及分析

1. 构造函数 skipList(K largeKey, int maxSize, float p)

平均时间复杂度：O(log(maxSize))

算法设计：在构造函数中，首先根据给定的参数计算了跳表的最大层数 maxLevel，然后初始化了头结点、尾结点和 last 数组.

2. 查找操作 bool find(const K &theKey) const

平均时间复杂度：O(log N)，其中 N 是跳表的元素个数

算法设计：从顶层开始，逐层向右查找，直到找到等于或大于目标值的节点

3. 插入操作 void insert(const K &theKey)

平均时间复杂度：O(log N)，其中 N 是跳表的元素个数

算法设计：首先进行查找操作定位到插入位置，然后确定新节点的层数，逐层插入节点

4. 删除操作 void erase(const K &theKey)

平均时间复杂度：O(log N)，其中 N 是跳表的元素个数

算法设计：首先进行查找操作定位到要删除的节点，然后逐层删除该节点

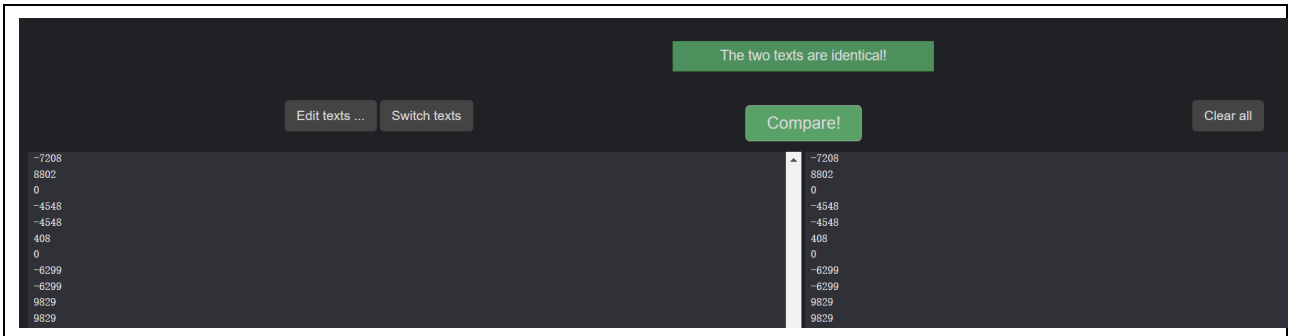
5. 整理操作 void reBuild()

平均时间复杂度：O(N)，其中 N 是跳表的元素个数

算法设计：初始化 last 指针数组为头节点，从头节点开始按顺序遍历元素，通过与 3 的次方关系来判断该节点所属的最大层数，并将对应的层数均插入该节点，最后将 last 指针数组指向尾节点

3. 测试

基础功能测试：通过输入 10 个样例，将输出数据保存到文件中，在网站 https://text-compare.com/中进行文件的对比(下图以 outPut07 为例)，可以发现输出结果正确.



动态序列测试：设定 n = 1000. 通过规定不同规模的 m， 随机生成初始化数据和操作数数据，来测试各个操作的平均运行时间和总平均运行时间.

M = 100, n =1000

```
(查找操作)执行时间为: 0ms; 执行次数为17; 平均执行时间为0ms
(插入元素+求异或和)操作执行时间为: 0ms; 执行次数为20; 平均执行时间为0ms
(删除元素+求异或和)操作执行时间为: 2ms; 执行次数为24; 平均执行时间为0.0833333ms
(删除最小元素)操作执行时间为: 0ms; 执行次数为16; 平均执行时间为0ms
(删除最大元素)操作执行时间为: 0ms; 执行次数为23; 平均执行时间为0ms
初始数据规模为: 1000; 总执行时间为: 2ms; 总操作次数为100; 平均执行时间为0.02ms
```

m = 1000, n = 1000;

```
(查找操作)执行时间为: 1ms; 执行次数为195; 平均执行时间为0.00512821ms
(插入元素+求异或和)操作执行时间为: 3ms; 执行次数为190; 平均执行时间为0.0157895ms
(删除元素+求异或和)操作执行时间为: 2ms; 执行次数为203; 平均执行时间为0.00985222ms
(删除最小元素)操作执行时间为: 0ms; 执行次数为208; 平均执行时间为0ms
(删除最大元素)操作执行时间为: 4ms; 执行次数为204; 平均执行时间为0.0196078ms
初始数据规模为: 1000; 总执行时间为: 10ms; 总操作次数为1000; 平均执行时间为0.01ms
```

m = 5000, n = 1000;

```
(查找操作)执行时间为: 9ms; 执行次数为965; 平均执行时间为0.00932642ms
(插入元素+求异或和)操作执行时间为: 6ms; 执行次数为1003; 平均执行时间为0.00598205ms
(删除元素+求异或和)操作执行时间为: 10ms; 执行次数为1016; 平均执行时间为0.00984252ms
(删除最小元素)操作执行时间为: 3ms; 执行次数为1011; 平均执行时间为0.00296736ms
(删除最大元素)操作执行时间为: 9ms; 执行次数为1005; 平均执行时间为0.00895522ms
初始数据规模为: 1000; 总执行时间为: 37ms; 总操作次数为5000; 平均执行时间为0.0074ms
```

M = 10000, n = 1000;

```
(查找操作)执行时间为: 11ms; 执行次数为2028; 平均执行时间为0.00542406ms
(插入元素+求异或和)操作执行时间为: 9ms; 执行次数为1985; 平均执行时间为0.00453401ms
(删除元素+求异或和)操作执行时间为: 13ms; 执行次数为1960; 平均执行时间为0.00663265ms
(删除最小元素)操作执行时间为: 9ms; 执行次数为2026; 平均执行时间为0.00444225ms
(删除最大元素)操作执行时间为: 18ms; 执行次数为2001; 平均执行时间为0.0089955ms
初始数据规模为: 1000; 总执行时间为: 60ms; 总操作次数为10000; 平均执行时间为0.006ms
```

M = 30000, n = 1000;

```
(查找操作)执行时间为: 30ms; 执行次数为6070; 平均执行时间为0.00494234ms
(插入元素+求异或和)操作执行时间为: 42ms; 执行次数为5936; 平均执行时间为0.00707547ms
(删除元素+求异或和)操作执行时间为: 38ms; 执行次数为5998; 平均执行时间为0.00633545ms
(删除最小元素)操作执行时间为: 29ms; 执行次数为5977; 平均执行时间为0.00485193ms
(删除最大元素)操作执行时间为: 39ms; 执行次数为6019; 平均执行时间为0.00647948ms
初始数据规模为: 1000; 总执行时间为: 179ms; 总操作次数为30000; 平均执行时间为0.00596667m
```

M = 60000, n = 1000;

```
(查找操作)执行时间为: 78ms; 执行次数为12106; 平均执行时间为0.00644309ms
(插入元素+求异或和)操作执行时间为: 84ms; 执行次数为11953; 平均执行时间为0.00702752ms
(删除元素+求异或和)操作执行时间为: 75ms; 执行次数为12023; 平均执行时间为0.00623804ms
(删除最小元素)操作执行时间为: 68ms; 执行次数为11919; 平均执行时间为0.00570518ms
(删除最大元素)操作执行时间为: 76ms; 执行次数为11999; 平均执行时间为0.00633386ms
初始数据规模为: 1000; 总执行时间为: 382ms; 总操作次数为60000; 平均执行时间为0.00636667ms
```

可以看出随着 M 的增大，总体的平均执行时间大体是逐渐降低的，则跳表会更加体现优势.
当输入规模相较于 n 大到一定程度后，平均执行时间会上升，则此时需要整理算法帮助我们调整跳表结构，优化时间.
可看出，如果随机操作数生成 1-5 的话，整体的数值小且不稳定，我们为了防止随机数中删除操作占比过多导致的影响，统一将 op 改为 2，即插入操作，这样得出的平均执行时间相对稳定.

我设计了自己的整理算法,从头节点开始按顺序遍历元素，通过与 3 的次方关系来判断该节点所属的最大层数，并将对应的层数均插入该节点，使得跳表更为有序，进行操作更高效.

M=30000, n = 1000, insert 操作，不使用整理算法

```
(查找操作)执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(插入元素+求异或和)操作执行时间为: 3956ms; 执行次数为30000; 平均执行时间为0.131867ms
(删除元素+求异或和)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(删除最小元素)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(删除最大元素)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
初始数据规模为: 1000; 总执行时间为: 3957ms; 总操作次数为30000; 使用整理算法平均执行时间为0.1319ms
```

M=30000, n = 1000, insert 操作，使用整理算法

```
(查找操作)执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(插入元素+求异或和)操作执行时间为: 3753ms; 执行次数为30000; 平均执行时间为0.1251ms
(删除元素+求异或和)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(删除最小元素)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(删除最大元素)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
初始数据规模为: 1000; 总执行时间为: 3754ms; 总操作次数为30000; 使用整理算法平均执行时间为0.125133ms
```

M = 60000, n = 1000, insert 操作，不使用整理算法

```
(查找操作)执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(插入元素+求异或和)操作执行时间为: 17063ms; 执行次数为60000; 平均执行时间为0.284383ms
(删除元素+求异或和)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(删除最小元素)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(删除最大元素)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
初始数据规模为: 1000; 总执行时间为: 17064ms; 总操作次数为60000; 使用整理算法平均执行时间为0.2844ms
```

M = 60000, n = 1000, insert 操作，使用整理算法

```
(查找操作)执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(插入元素+求异或和)操作执行时间为: 14529ms; 执行次数为60000; 平均执行时间为0.24215ms
(删除元素+求异或和)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(删除最小元素)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
(删除最大元素)操作执行时间为: 0ms; 执行次数为0; 平均执行时间为nanms
初始数据规模为: 1000; 总执行时间为: 14529ms; 总操作次数为60000; 使用整理算法平均执行时间为0.24215ms
```

使用整理算法之后，大数据的平均执行时间明显下降，显示了整理算法的优势.

4. 分析与探讨

本次实验我通过学习跳表的数据结构，了解了它类似于二分的思想，并实现了跳表的基础功能，成功完成了测试样例.

此外，我也成功通过自己生成动态序列，进行文件的输入输出操作，改变输入数据的规模，成功简洁地完成了跳表的性能测量与分析.

遇到的问题如下

1. 测试输出样例的时候错误

解决 ： 通过调试发现 n, m 的位置输入反了，改正即可.

2. 测试运行时间以 s 尾单位时发现数据过小无法显示

解决 ： 将 s 单位改为 ms 单位，使得数据更为明显.

3. **自定义随机生成数据，测试运行时间时发现很多操作显示 0ms 运行时间**

解决 ： 发现 n 的值设置太小，如 10，导致初始数据太少，导致随机生成的删除操作因为没有相关元素直接返回，通过增大 n 值为 1000 解决.

4. 数值很小，测试不稳定

解决 ： 统一将 op 改为 2，即插入操作，这样得出的平均执行时间相对稳定.

5. 附录：实现源代码

SkipList 类

```
template <class K>
class skipList{
public :
    skipList(K largeKey, int maxSize, float p); //  构造函数
    bool find (const K & theKey) const ; //查找对应的元素
    int level() const ;
    skipNode<K> * search(const K & theKey) const;
    void insert(const K & theKey);
    void erase (const K & theKey);
    int size();
    bool empty();
    void output(ostream & out) const;
    int xorNum();
    int eraseMinElement();
    int eraseMaxElement();
    void rebuild();
private:
    float cutOff; //  确定层数
    int levers; //当前最大的非空链表
```

```
    int dSize;//元素个数
    int maxLevel; //  最大层数
    K tailKey; //最大关键字
    skipNode<K> * headerNode; //  头节点
    skipNode<K> * tailNode; //  尾节点
    skipNode<K> ** last; // last[i]表示 i 层链表最后的节点
};
template<class K>
ostream & operator<<(ostream &out, const skipList<K> & x) {
    x.output(out);
    return out;
}
```

查找操作

```
template<class K>
bool skipList<K>::find(const K &theKey) const {
    if (theKey >= tailKey) {
        return false;
    }
    //  关键字  < key 的最大的位置
    skipNode<K> * beforeNode = headerNode;
    //  从最高层开始搜索
    for (int i = levers; i >= 0; i--) {
        while (beforeNode->next[i]->element < theKey) {
            beforeNode = beforeNode->next[i];
        }
    }

    //如果第一层有
    if (beforeNode->next[0]->element == theKey) {
        return true;
    }
    return false;
}
```

确定级数操作

```
template<class K>
int skipList<K>::level() const {
    int lev = 0 ;
    while (rand() <= cutOff) {
        lev++;
    }
    return (lev <= maxLevel) ? lev : maxLevel;
}
```

搜寻最后一个小于关键字的元素操作

```
template<class K>
skipNode<K>* skipList<K>::search(const K &theKey) const {
```

```
	skipNode<K> * beforeNode = headerNode;
	// last[i]更新为关键字的前一个结点
	for (int i = levers; i >= 0; i--) {
		while (beforeNode->next[i]->element < theKey) {
			beforeNode = beforeNode->next[i];
		}
		last[i] = beforeNode;
	}

	return beforeNode->next[0];
}
```

插入元素操作

```
template<class K>
void skipList<K>::insert(const K &theKey) {
	if (theKey >= tailKey) {
		return;
	}

	//此时存储了每一级需要插入的后一个结点
	skipNode<K> * theNode = search(theKey);
	if (theNode->element == theKey) {
		return;
	}

	//确定新节点的链表级数
	int theLevel = level();
	if (theLevel > levers) {
		//比如说现有 3 级 ，加到了 5 级，则设为 4 级
		theLevel = ++levers;
		last[theLevel] = headerNode;
	}

	// 比如说属于 3 级链表，则创建 4 个 next 即可
	skipNode<K> * newNode = new skipNode<K> (theKey, theLevel + 1);
	for (int i = 0 ;i <= theLevel; i++) {
		newNode->next[i] = last[i] ->next[i];
		last[i]->next[i] = newNode;
	}

	//插入后更新
	dSize++;
	return ;
}
```

删除元素操作

```
template<class K>
void skipList<K>::erase(const K &theKey) {
```

```
	if (theKey >= tailKey) {
		return;
	}

	skipNode<K>* theNode = search(theKey);
	if (theNode->element != theKey) {
		return;
	}

	//删除结点，此时不知道该结点对应的级数?
	for (int i = 0 ; i <= levers && last[i]->next[i] == theNode; i++) {
		last[i] -> next[i] = theNode->next[i];
	}

	//更新链表级
	while (levers > 0 && headerNode->next[levers] == tailNode) {
		levers--;
	}

	//释放结点空间
	delete theNode;
	dSize--;
}
```

异或和操作

```
template<class K>
int skipList<K>::xorNum() {
	int ans = 0;
	for (auto t = headerNode->next[0]; t != tailNode; t = t->next[0]) {
		ans ^= t->element;
	}
	return ans;
}
```

删除最小元素并输出

```
template<class K>
int skipList<K>::eraseMinElement() {
	int ele = headerNode->next[0]->element;
	erase(ele);
	return ele;
}
```

删除最大元素并输出

```
template<class K>
int skipList<K>::eraseMaxElement() {
	if (empty()) {
		return -1;
	}
```

```
    auto t = headerNode[0].next[0];
    while(t -> next[0] != tailNode) {
        t = t ->next[0];
    }

    int ele = t->element ;
    erase(ele);
    return ele;
}
```

跳表整理

```
template<class K>
void skipList<K>::reBuild()
{
    if (dSize == 0) {
        return;
    }
    int val = 3;
    // 找到第一个节点
    skipNode<K>* nowNode = headerNode->next[0];
    //初始化 last 数组为头结点
    for (int i = 0; i <= maxLevel; i++) {
        last[i] = headerNode;
    }
    // 从第二个节点开始遍历
    for (int index = 1; index <= dSize; index++)
    {
        int cur = index;
        int cur_level = 0;//记录当前节点应该有的最高级数
        while (cur % val ==0) {
            cur_level++;
            cur /= val;
        }
        skipNode<K> *nextNode = nowNode->next[0];// 保存下一个节点
        //将级数置为 cur_level + 1
        nowNode->next = new skipNode<K> * [cur_level + 1];
        //最高级数之下的进行重构，使用尾插法
        for (int i = 0; i <= cur_level; i++) {
            last[i]->next[i] = nowNode;
            last[i] = nowNode;
        }
        nowNode = nextNode;// 移到下一个节点
    }

    // 连接尾结点
    for (int i = 0; i <= maxLevel; i++)
    {
```

```
            last[i]->next[i] = tailNode;
        }
}
```

随机生成动态序列

```
void initData() {
        ofstream generateFile("input.txt", ios::trunc);
        generateFile << m << " " << n << endl; //输入 n, m
        //生成-100000 + 100000 内的随机数
        for (int i = 0 ; i < n ; i++) {
            int x = rand() % 200001 - 100000;
            generateFile << x << " ";
        }
        generateFile << endl;
        //生成操作序列
        for (int i = 0 ; i < m; i++) {
            int op = rand() % 5 + 1;
            int num = rand() % 200001 - 100000;
            generateFile << op << " ";
            if (op == 1 || op == 2 || op == 3) {
                generateFile << num;
            }
            generateFile << endl;
        }
        generateFile.close();
}
```