南開大學
Nankai University

数据结构实验报告 5

散列

姓　　名：南亚宏

学　　号：2311788

目录

2025 年 11 月 11 日

一、 题目一

1.1 题目表述

开发一个基于线性探查的散列表类，要求用 neverUsed 思想进行删除操作。为每个方法编写 C++代码。其中有一个方法，它在 60%的空桶的 neverUsed 域的值为 false 时，重新组织散列表。重新组织散列表的过程要在必要时移动记录，重新组织之后，每个空桶的 neverUsed 域的值为 true。测试代码的正确性。

1.2 代码 1 的测试用例:

容量与哈希：cap = 7，h(k) = k % 7，以下分别是输入输出
返回格式（示例）：

- insert(k,v) -> {inserted|updated|full, index:i}
- find(k) -> {found|not_found, index:i[, value:...]}
- erase(k) -> {removed|not_found, index:i}：

1. insert(1,"a") {inserted, index:1}
2. insert(8,"b") {inserted, index:2}
3. insert(15,"c") {inserted, index:3}
4. find(22) {not_found, index:4}
5. erase(8) {removed, index:2}
6. find(15) {found, index:3, value:"c"}
7. insert(22,"d") {inserted, index:2}
8. insert(0,"t0") {inserted, index:0}
9. insert(2,"t2") {inserted, index:4} // 2 需线性探查：2→3 占用，落在 4
10. insert(4,"t4") {inserted, index:5} // 4 与 idx4 冲突，落在 5
11. erase(0) {removed, index:0}
12. erase(2) {removed, index:4}
13. erase(4) {removed, index:5}
14. insert(5,"e") **触发重组织后插入** → {inserted, index:5}
15. find(1) {found, index:1, value:"a"}
16. find(22) {found, index:2, value:"d"}
17. find(0) {not_found, index:0}

1.3 思路:

每个桶存 key、value、inUse（是否有有效数据）、neverUsed（是否从未使用），用 neverUsed 区分 “从未用” 和 “已删除”，避免删除后探查链断裂。
用 k%cap 计算初始索引，线性探查（索引 +1 循环）解决冲突，探查时跳过 “从未使用” 的桶（后续无目标数据）。

插入逻辑是优先复用“已删除桶”（inUse=false 但 neverUsed=false），无则用“从未使用桶”；插入前检查空桶中 neverUsed=false 比例，≥60% 则先重组织。
删除逻辑是不物理删除数据，仅将 inUse 设为 false，保留 neverUsed=false，确保后续探查链不中断。
查找逻辑是从哈希初始索引开始探查，遇“从未使用桶”终止（无目标数据），返回最终探查索引；找到目标 key 且 inUse=true 时返回对应值。
空桶碎片（neverUsed=false）过多时触发重组织机制，重置表（所有桶设为“从未使用”），重新插入所有有效数据（inUse=true），清理碎片并恢复探查效率。

1.4 代码:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class LinearProbingHashTable {
private:
    struct Entry {
        int key;
        string value;
        bool inUse;     // 表示该桶当前是否有有效数据（未被删除）
        bool neverUsed; // 表示该桶是否从未被使用过（包括未插入和未删除）
    };

    vector<Entry> table;
    int capacity;
    int size; // 当前有效数据个数（inUse=true 的桶数）

    int hash(int key) {
        return key % capacity;
    }

    // 计算空桶中 neverUsed=false 的比例（用于触发重组织）
    double getNeverUsedFalseRatio() {
        int emptyCount = 0;          // 空桶总数（inUse=false 的桶）
```

```
        int neverUsedFalseCount = 0; // 空桶中 neverUsed=false
的数量
        for (const auto& entry : table) {
            if (!entry.inUse) {
                emptyCount++;
                if (!entry.neverUsed) {
                    neverUsedFalseCount++;
                }
            }
        }
        return emptyCount == 0 ? 0.0 :
(double)neverUsedFalseCount / emptyCount;
    }

    // 重组织散列表：重新插入所有有效数据，空桶 neverUsed 设为 true
    void reorganize() {
        vector<Entry> oldTable = table;
        // 重置新表：所有桶 neverUsed=true，inUse=false
        table.assign(capacity, {0, "", false, true});
        size = 0;

        // 重新插入旧表中所有有效数据（inUse=true）
        for (const auto& entry : oldTable) {
            if (entry.inUse) {
                insert(entry.key, entry.value, true); // 内部调
用，跳过重组织检查
            }
        }
    }

    // 内部插入接口：skipReorganize=true 时跳过重组织检查（避免递
归）
    string insert(int key, const string& value, bool
skipReorganize) {
        if (!skipReorganize && getNeverUsedFalseRatio() >= 0.6)
{
            reorganize();
        }

        int index = hash(key);
        int start = index;
```

```
        int firstDeletedIndex = -1; // 记录第一个遇到的已删除桶（inUse=false）

        do {
            // 情况 1：桶从未使用过（neverUsed=true），直接插入
            if (table[index].neverUsed) {
                // 优先使用之前找到的已删除桶（如果有）
                if (firstDeletedIndex != -1) {
                    index = firstDeletedIndex;
                }
                table[index].key = key;
                table[index].value = value;
                table[index].inUse = true;
                table[index].neverUsed = false;
                size++;
                return "{inserted, index:" + to_string(index) + "}";
            }
            // 情况 2：桶已使用过（neverUsed=false）
            else {
                // 找到相同 key，更新值
                if (table[index].key == key) {
                    table[index].value = value;
                    return "{updated, index:" + to_string(index) + "}";
                }
                // 遇到已删除的桶，记录第一个位置（用于后续插入）
                if (!table[index].inUse && firstDeletedIndex == -1) {
                    firstDeletedIndex = index;
                }
            }

            index = (index + 1) % capacity;
        } while (index != start);

        // 循环结束：表满（无从未使用的桶，且无已删除的桶）
        return "{full, index:" + to_string(start) + "}";
    }

public:
    LinearProbingHashTable(int cap) : capacity(cap), size(0) {
```

```
        table.assign(capacity, {0, "", false, true});
    }

    // 外部插入接口
    string insert(int key, const string& value) {
        return insert(key, value, false);
    }

    // 查找接口：返回探查终止时的索引（而非初始哈希索引）
    string find(int key) {
        int index = hash(key);
        int start = index;
        int finalIndex = start; // 记录探查终止时的索引

        do {
            finalIndex = index; // 更新当前探查索引为最终索引
            // 桶从未使用过，无需继续探查（后续桶也不可能有目标 key）
            if (table[index].neverUsed) {
                break;
            }
            // 找到目标 key，返回结果
            if (table[index].key == key && table[index].inUse)
{
                return "{found, index:" + to_string(index) + ",
value:" + table[index].value + "}";
            }

            index = (index + 1) % capacity;
        } while (index != start);

        // 未找到：返回探查终止时的索引
        return "{not_found, index:" + to_string(finalIndex) +
"}";
    }

    // 删除接口：仅标记 inUse=false，不改变 neverUsed
    string erase(int key) {
        int index = hash(key);
        int start = index;

        do {
            // 桶从未使用过，无需继续探查
```

```
            if (table[index].neverUsed) {
                break;
            }
            // 找到目标 key 且有效，标记为删除
            if (table[index].key == key && table[index].inUse)
{
                table[index].inUse = false;
                size--;
                return "{removed, index:" + to_string(index) +
"}";
            }

            index = (index + 1) % capacity;
        } while (index != start);

        // 未找到
        return "{not_found, index:" + to_string(start) + "}";
    }
};

int main() {
    LinearProbingHashTable ht(7);

    cout << ht.insert(1, "a") << endl;    // {inserted,
index:1}
    cout << ht.insert(8, "b") << endl;    // {inserted,
index:2}
    cout << ht.insert(15, "c") << endl;   // {inserted,
index:3}
    cout << ht.find(22) << endl;          // {not_found,
index:4}
    cout << ht.erase(8) << endl;          // {removed, index:2}
    cout << ht.find(15) << endl;          // {found, index:3,
value:"c"}
    cout << ht.insert(22, "d") << endl;   // {inserted,
index:2}
    cout << ht.insert(0, "t0") << endl;   // {inserted,
index:0}
    cout << ht.insert(2, "t2") << endl;   // {inserted,
index:4}
    cout << ht.insert(4, "t4") << endl;   // {inserted,
index:5}
```

```
    cout << ht.erase(0) << endl;          // {removed, index:0}
    cout << ht.erase(2) << endl;          // {removed, index:4}
    cout << ht.erase(4) << endl;          // {removed, index:5}
    cout << ht.insert(5, "e") << endl;    // 触发重组织后插入 → {inserted, index:5}
    cout << ht.find(1) << endl;           // {found, index:1, value:"a"}
    cout << ht.find(22) << endl;          // {found, index:2, value:"d"}
    cout << ht.find(0) << endl;           // {not_found, index:0}

    return 0;
}
```

1.5 测试用例截图

```
PS C:\Users\Macro\Desktop\hw5> cd .\hw5.1
PS C:\Users\Macro\Desktop\hw5\hw5.1> ./main.exe
{inserted, index:1}
{inserted, index:2}
{inserted, index:3}
{not_found, index:4}
{removed, index:2}
{found, index:3, value:c}
{inserted, index:2}
{inserted, index:0}
{inserted, index:4}
{inserted, index:5}
{removed, index:0}
{removed, index:4}
{removed, index:5}
{inserted, index:5}
{found, index:1, value:a}
{found, index:2, value:d}
{not_found, index:0}
PS C:\Users\Macro\Desktop\hw5\hw5.1>
```

二、 题目二

2.1 题目表述

设计一个类 hashChainsWithTail，其中每个散列链表都是一个有尾节点的有序链表，而且所有链表在物理上都共享一个尾节点。不使用任何链表类的方法实现插人和删除。和类 hashChains（见程序 10-20）比较时间性能。

程序 10-20 hashChains 的一些方法

```
template<class K, class E>
pair<const K, E>* find(const K& theKey) const
   {return table[hash(theKey) % divisor].find(theKey);}

void insert(const pair<const K, E>& thePair)
{
   int homeBucket = (int) hash(thePair.first) % divisor;
   int homeSize = table[homeBucket].size();
   table[homeBucket].insert(thePair);
   if (table[homeBucket].size() > homeSize)
      dSize++;
}

void erase(const K& theKey)
   {table[hash(theKey) % divisor].erase(theKey);}
```

2.2 代码 2 的测试用例:

约定：cap=5，h(k)=k%5；链表**按 key 升序**；重复键**不插入**并返回 exists。每条链都以同一个全局尾哨兵 T 结束。
返回格式：

- insert(k,v) -> {inserted|exists, bucket:b, pos:i}（pos 为链内位置，从 0 开始；T 不计入）
- find(k) -> {found|not_found, bucket:b, pos:i}
- erase(k) -> {removed|not_found, bucket:b}：

1. insert(6,"a") {inserted, bucket:1, pos:0}
2. insert(1,"b") {inserted, bucket:1, pos:0}
3. insert(11,"c") {inserted, bucket:1, pos:2}
4. find(6) {found, bucket:1, pos:1}
5. erase(1) {removed, bucket:1}
6. erase(7) {not_found, bucket:2}
7. insert(16,"d") {inserted, bucket:1, pos:2}
8. insert(6,"A'") {exists, bucket:1, pos:0}
9. find(99) {not_found, bucket:4}
10. **遍历检查尾哨兵** 对每个桶：tail.next==tail 且 &tail 同 addr(T)（同一地址）

2.3 思路:

使用散列表来管理一系列有序链表，每个链表都有一个尾哨兵（sentinel）。尾哨兵的 next 指针指向自身，用于标记链表的结束。所有链表共享同一个尾哨兵节点，减少内存使用。
关键在于正确地管理链表节点和尾哨兵，确保在插入、查找和删除操作中，链表的结构始终保持正确。此外，checkTailSentinel 函数用于验证链表的完整性，确保没有破坏尾哨兵的链接。

2.4 代码:

```
#include <iostream>
#include <string>
using namespace std;

// 链表节点结构（包含尾哨兵所需的 next 指针）
template <class K, class E>
struct ChainNode {
    K key;
    E value;
    ChainNode* next;

    // 普通节点构造函数（默认指向尾哨兵）
    ChainNode(const K& k, const E& v, ChainNode* n = nullptr) :
key(k), value(v), next(n) {}
    // 尾哨兵专用构造函数（无 key/value，next 指向自身）
    ChainNode() : next(this) {}
};

// 全局尾哨兵（单例，所有链表共享）
ChainNode<int, string> T;

template <class K, class E>
class hashChainsWithTail {
private:
    ChainNode<K, E>** table; // 散列表
    int divisor;             // 桶数量
    ChainNode<K, E>* tail;   // 指向全局尾哨兵（类型匹配）

    int hash(const K& k) const {
        return k % divisor;
```

```
    }

public:
    hashChainsWithTail(int cap) : divisor(cap) {
        table = new ChainNode<K, E>*[divisor];
        // 所有空桶直接指向尾哨兵
        for (int i = 0; i < divisor; ++i) {
            table[i] = reinterpret_cast<ChainNode<K, E>*>(&T);
        }
        tail = reinterpret_cast<ChainNode<K, E>*>(&T);
    }

    ~hashChainsWithTail() {
        for (int i = 0; i < divisor; ++i) {
            ChainNode<K, E>* cur = table[i];
            while (cur != tail) {
                ChainNode<K, E>* temp = cur;
                cur = cur->next;
                delete temp;
            }
        }
        delete[] table;
    }

    pair<string, pair<int, int>> insert(const K& k, const E& v)
{
        int b = hash(k);
        ChainNode<K, E>* prev = nullptr;
        ChainNode<K, E>* cur = table[b];
        int pos = 0;

        while (cur != tail && cur->key < k) {
            prev = cur;
            cur = cur->next;
            pos++;
        }

        if (cur != tail && cur->key == k) {
            return {"exists", {b, pos}};
        }
```

```
        ChainNode<K, E>* newNode = new ChainNode<K, E>(k, v,
cur);
        if (prev == nullptr) {
            table[b] = newNode;
        } else {
            prev->next = newNode;
        }
        return {"inserted", {b, pos}};
    }

    pair<string, pair<int, int>> find(const K& k) const {
        int b = hash(k);
        ChainNode<K, E>* cur = table[b];
        int pos = 0;

        while (cur != tail && cur->key != k) {
            cur = cur->next;
            pos++;
        }

        if (cur != tail && cur->key == k) {
            return {"found", {b, pos}};
        } else {
            return {"not_found", {b, -1}};
        }
    }

    pair<string, int> erase(const K& k) {
        int b = hash(k);
        ChainNode<K, E>* prev = nullptr;
        ChainNode<K, E>* cur = table[b];

        while (cur != tail && cur->key != k) {
            prev = cur;
            cur = cur->next;
        }

        if (cur == tail || cur->key != k) {
            return {"not_found", b};
        }

        if (prev == nullptr) {
```

```
            table[b] = cur->next;
        } else {
            prev->next = cur->next;
        }
        delete cur;
        return {"removed", b};
    }

    // 修复尾哨兵检查逻辑
    bool checkTailSentinel() const {
        // 1. 检查尾哨兵自身的 next 是否指向自己
        if (T.next != &T) {
            return false;
        }

        // 2. 检查每个桶的链表末尾是否指向尾哨兵
        for (int i = 0; i < divisor; ++i) {
            ChainNode<K, E>* cur = table[i];
            // 遍历到链表最后一个节点（next 为尾哨兵）
            while (cur->next != tail) {
                // 防止死循环
                cur = cur->next;
            }
            // 确认最后一个节点的 next 是尾哨兵
            if (cur->next != tail) {
                return false;
            }
        }

        return true;
    }
};

void test() {
    hashChainsWithTail<int, string> hc(5);

    // 测试用例
    auto res1 = hc.insert(6, "a");
    cout << "{" << res1.first << ", bucket:" <<
res1.second.first << ", pos:" << res1.second.second << "}" <<
endl;
```

```
    auto res2 = hc.insert(1, "b");
    cout << "{" << res2.first << ", bucket:" <<
res2.second.first << ", pos:" << res2.second.second << "}" <<
endl;

    auto res3 = hc.insert(11, "c");
    cout << "{" << res3.first << ", bucket:" <<
res3.second.first << ", pos:" << res3.second.second << "}" <<
endl;

    auto res4 = hc.find(6);
    cout << "{" << res4.first << ", bucket:" <<
res4.second.first << ", pos:" << res4.second.second << "}" <<
endl;

    auto res5 = hc.erase(1);
    cout << "{" << res5.first << ", bucket:" << res5.second <<
"}" << endl;

    auto res6 = hc.erase(7);
    cout << "{" << res6.first << ", bucket:" << res6.second <<
"}" << endl;

    auto res7 = hc.insert(16, "d");
    cout << "{" << res7.first << ", bucket:" <<
res7.second.first << ", pos:" << res7.second.second << "}" <<
endl;

    auto res8 = hc.insert(6, "A'");
    cout << "{" << res8.first << ", bucket:" <<
res8.second.first << ", pos:" << res8.second.second << "}" <<
endl;

    auto res9 = hc.find(99);
    cout << "{" << res9.first << ", bucket:" <<
res9.second.first << "}" << endl;

    // 输出尾哨兵检查结果
    bool tailCheck = hc.checkTailSentinel();
    cout <<  (tailCheck ? "pass" : "not pass") << endl;
}
```

```
int main() {
    test();
    return 0;
}
```

和类 hashChains（程序 10-20）比较时间性能：
在时间性能上，hashChains 和 hashChainsWithTail 在插入、查找和删除操作上的时间复杂度相同：都是 O(n)，其中 n 是桶中元素的数量。
hashChainsWithTail 在内存使用上更高效，并且提供了额外的尾哨兵检查功能。

2.5 测试用例截图

```
PS C:\Users\Macro\Desktop\hw5\hw5.2> ./main.exe
{inserted, bucket:1, pos:0}
{inserted, bucket:1, pos:0}
{inserted, bucket:1, pos:2}
{found, bucket:1, pos:1}
{removed, bucket:1}
{not_found, bucket:2}
{inserted, bucket:1, pos:2}
{exists, bucket:1, pos:0}
{not_found, bucket:4}
pass
PS C:\Users\Macro\Desktop\hw5\hw5.2>
```