5.4: 不使用链表的哈希表

概述

与分离链表哈希表相比，分离链表哈希表的缺点是使用了链表。这可能会因为分配新单元所需的时间而稍微降低算法的速度，并且实际上需要实现第二种数据结构。不使用链表来解决冲突的替代方法是尝试替代的单元，直到找到一个空单元。更正式地说，单元 h0(x), h1(x), h2(x), ... 被连续尝试，其中 h(x) = (hash(x) + f(i)) mod TableSize，f(0) = 0。函数 f 是冲突解决策略。由于所有数据都进入表中，因此需要一个比分离链表哈希表更大的表。通常，对于不使用分离链表的哈希表，负载因子应该低于 λ = 0.5。我们称这样的表为探测哈希表。我们现在来看三种常见的冲突解决策略。

—

1 5.4.1 线性探测法（Linear Probing）

线性探测法从发生冲突的槽位开始，逐个检查下一个槽位，直到找到空槽。

算法步骤

- **插入**：从初始哈希值开始，检查每个槽位，找到空槽插入。
- **查找**：从初始槽位依次检查，直到找到目标或遇到空槽。
- **删除**：标记为“已删除”（lazy deletion），避免影响其他元素。

优缺点

- **优点**：实现简单，元素存储在同一数组中。
- **缺点**：容易产生“聚集现象”，降低性能。

—

2 5.4.2 二次探测法（Quadratic Probing）

二次探测法通过非线性跳跃减少聚集现象。

算法公式

$$f(i) = i^2 \bmod M$$

其中，i 是探测次数，M 是哈希表的大小。

优缺点

- **优点**：减少聚集现象。
- **缺点**：需要仔细选择 M，否则可能无法插入。

定理5.1

定理5.1 如果使用二次探测，并且表大小是质数，那么如果表至少半空，总是可以插入新元素。(证明详见课本)

3 5.4.3 双重哈希（Double Hashing）

双重哈希使用两个独立的哈希函数，进一步优化冲突处理。

算法公式

$$f(i) = ih_2(x)$$

举例

如果选择不当，hash2(x) 可能会导致灾难性的后果。例如，如果插入了99，那么显而易见的选择$hash2(x) = xmod9$就不会有帮助。因此，该函数必须永远不会评估为零。同样重要的是要确保所有的单元格都可以被探测到（在下面的例子中这是不可能的，因为表大小不是质数）。一个函数如 $hash2(x) = R - (xmodR)$，其中 R 是一个小于 TableSize 的质数，将会工作得很好。

优缺点

- **优点**：分布更均匀，进一步减少聚集。
- **缺点**：实现复杂，需要设计两个独立的哈希函数。

---

4 方法对比

方法	探测公式	优点	缺点
线性探测法	$h_i(k) = (h(k) + i) \bmod M$	实现简单	易出现聚集现象
二次探测法	$h_i(k) = (h(k) + i^2) \bmod M$	减少聚集现象	插入失败可能性更高
双重哈希	$h_i(k) = (h_1(k) + i \cdot h_2(k)) \bmod M$	分布更均匀	实现复杂，需要两个哈希

表 1: 哈希表冲突处理方法对比

---

5 示例代码：线性探测法

以下代码实现了线性探测法的插入、查找和删除功能：

Listing 1: 基于线性探测法的哈希表实现

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

class HashTable {
private:
    vector<string> table;
    int size;
    string DELETED = "__DELETED__";

public:
    HashTable(int size) : size(size) {
        table.resize(size, "");
    }

```

```
    int hashFunction(string key) {
        int hash = 0;
        for (char ch : key) {
            hash = (hash * 31 + ch) % size;
        }
        return hash;
    }

    void insert(string key) {
        int index = hashFunction(key);
        while (table[index] != "" && table[index] != DELETED) {
            index = (index + 1) % size;
        }
        table[index] = key;
    }

    bool search(string key) {
        int index = hashFunction(key);
        int start = index;
        while (table[index] != "") {
            if (table[index] == key) return true;
            index = (index + 1) % size;
            if (index == start) break;
        }
        return false;
    }

    void remove(string key) {
        int index = hashFunction(key);
        int start = index;
        while (table[index] != "") {
            if (table[index] == key) {
                table[index] = DELETED;
                return;
            }
            index = (index + 1) % size;
            if (index == start) break;
        }
    }
```

```
56
57      void display() {
58          for (int i = 0; i < size; ++i) {
59              cout << i << ": " << table[i] << endl;
60          }
61      }
62  };
63
64  int main() {
65      HashTable hashTable(7);
66
67      hashTable.insert("Alice");
68      hashTable.insert("Bob");
69      hashTable.insert("Charlie");
70
71      hashTable.display();
72
73      cout << "Search Bob: " << hashTable.search("Bob") << endl;
74
75      hashTable.remove("Bob");
76      cout << "After removing Bob:" << endl;
77      hashTable.display();
78
79      return 0;
80  }
```

—