Lecture 08: Hash Tables

C++ Code Samples — Sedgwick Algorithms Course — lecture-08-samples.cpp

```cpp
// ============================================================================
// Lecture 08: Hash Tables  --  Sedgwick Algorithms Course
//
// Topics: polynomial rolling hash, separate chaining, linear probing,
//         insert/search/delete, collision analysis at various load factors
// ============================================================================

#include <iostream>
#include <vector>
#include <string>
#include <list>
#include <optional>
#include <functional>
#include <algorithm>
#include <iomanip>

using namespace std;

// === SECTION: Hash Function ===
// Polynomial rolling hash: h(s) = (s[0]*p^(n-1) + s[1]*p^(n-2) + ... + s[n-1]) mod m
// This is a widely-used hash for strings. The prime base (31) distributes well.

size_t polynomialHash(const string& key, int tableSize) {
    const int p = 31;           // Small prime base
    const int mod = 1e9 + 7;    // Large prime to avoid overflow
    size_t hash = 0;
    size_t pPow = 1;
    for (int i = key.size() - 1; i >= 0; i--) {
        hash = (hash + (key[i] - 'a' + 1) * pPow) % mod;
        pPow = (pPow * p) % mod;
    }
    return hash % tableSize;
}

// === SECTION: Separate Chaining Hash Table ===
// Each bucket is a linked list. On collision, we append to the list.
// Load factor = N / M (items / buckets). Works well even for alpha > 1.

class ChainingHashTable {
    struct Entry {
        string key;
        int value;
    };

    vector<list<Entry>> table;
    int tableSize;
    int count;

    int hash(const string& key) const {
        return polynomialHash(key, tableSize);
    }

public:
    ChainingHashTable(int size = 11) : tableSize(size), count(0) {
        table.resize(tableSize);
    }

    void insert(const string& key, int value) {
        int idx = hash(key);
        // Update if key exists
        for (auto& entry : table[idx]) {
```

```
            if (entry.key == key) { entry.value = value; return; }
        }
        table[idx].push_back({key, value});
        count++;
    }

    optional<int> search(const string& key) const {
        int idx = hash(key);
        for (const auto& entry : table[idx]) {
            if (entry.key == key) return entry.value;
        }
        return nullopt;
    }

    bool remove(const string& key) {
        int idx = hash(key);
        for (auto it = table[idx].begin(); it != table[idx].end(); ++it) {
            if (it->key == key) {
                table[idx].erase(it);
                count--;
                return true;
            }
        }
        return false;
    }

    double loadFactor() const { return (double)count / tableSize; }
    int size() const { return count; }

    void printTable(const string& label) const {
        cout << label << " (size=" << count << ", buckets=" << tableSize
             << ", load=" << fixed << setprecision(2) << loadFactor() << "):" << endl;
        for (int i = 0; i < tableSize; i++) {
            cout << "  [" << setw(2) << i << "]: ";
            if (table[i].empty()) {
                cout << "(empty)";
            } else {
                bool first = true;
                for (const auto& entry : table[i]) {
                    if (!first) cout << " -> ";
                    cout << "\"" << entry.key << "\":" << entry.value;
                    first = false;
                }
            }
            cout << endl;
        }
    }

    // Return the distribution of chain lengths for analysis
    vector<int> chainLengths() const {
        vector<int> lengths;
        for (const auto& chain : table) {
            lengths.push_back(chain.size());
        }
        return lengths;
    }
};

// === SECTION: Linear Probing Hash Table ===
// Open addressing: on collision, probe the next slot linearly.
// Uses a sentinel for deleted slots (lazy deletion).
// Load factor must stay below ~0.75 for good performance.

class LinearProbingHashTable {
```

```cpp
    struct Entry {
        string key;
        int value;
        bool occupied = false;
        bool deleted = false;     // Tombstone marker for lazy deletion
    };

    vector<Entry> table;
    int tableSize;
    int count;

    int hash(const string& key) const {
        return polynomialHash(key, tableSize);
    }

public:
    LinearProbingHashTable(int size = 11) : tableSize(size), count(0) {
        table.resize(tableSize);
    }

    void insert(const string& key, int value) {
        int idx = hash(key);
        int start = idx;
        // Linear probe: find an empty or deleted slot, or update existing
        do {
            if (!table[idx].occupied || table[idx].deleted) {
                table[idx] = {key, value, true, false};
                count++;
                return;
            }
            if (table[idx].key == key) {
                table[idx].value = value;   // Update existing
                return;
            }
            idx = (idx + 1) % tableSize;
        } while (idx != start);
        // Table is full (should not happen with proper load factor management)
        cout << "  WARNING: Table full, cannot insert \"" << key << "\"" << endl;
    }

    optional<int> search(const string& key) const {
        int idx = hash(key);
        int start = idx;
        do {
            if (!table[idx].occupied && !table[idx].deleted) return nullopt;
            if (table[idx].occupied && !table[idx].deleted && table[idx].key == key)
                return table[idx].value;
            idx = (idx + 1) % tableSize;
        } while (idx != start);
        return nullopt;
    }

    bool remove(const string& key) {
        int idx = hash(key);
        int start = idx;
        do {
            if (!table[idx].occupied && !table[idx].deleted) return false;
            if (table[idx].occupied && !table[idx].deleted && table[idx].key == key) {
                table[idx].deleted = true;   // Mark as tombstone
                count--;
                return true;
            }
            idx = (idx + 1) % tableSize;
        } while (idx != start);
```

```cpp
        return false;
    }

    double loadFactor() const { return (double)count / tableSize; }
    int size() const { return count; }

    void printTable(const string& label) const {
        cout << label << " (size=" << count << ", capacity=" << tableSize
             << ", load=" << fixed << setprecision(2) << loadFactor() << "):" << endl;
        for (int i = 0; i < tableSize; i++) {
            cout << "  [" << setw(2) << i << "]: ";
            if (table[i].deleted) {
                cout << "(deleted)";
            } else if (!table[i].occupied) {
                cout << "(empty)";
            } else {
                cout << "\"" << table[i].key << "\":" << table[i].value;
            }
            cout << endl;
        }
    }

    // Count clusters (consecutive occupied slots) for analysis
    int countClusters() const {
        int clusters = 0;
        bool inCluster = false;
        for (int i = 0; i < tableSize; i++) {
            if (table[i].occupied && !table[i].deleted) {
                if (!inCluster) { clusters++; inCluster = true; }
            } else {
                inCluster = false;
            }
        }
        return clusters;
    }
};

// === SECTION: Utility ===

void printHashValues(const vector<string>& keys, int tableSize) {
    cout << "  Hash values (table size " << tableSize << "):" << endl;
    for (const auto& key : keys) {
        cout << "    \"" << key << "\" -> " << polynomialHash(key, tableSize) << endl;
    }
}

// === MAIN ===

int main() {
    cout << "========================================" << endl;
    cout << " Lecture 08: Hash Tables" << endl;
    cout << "========================================" << endl;

    // Test data: names and associated values
    vector<pair<string, int>> data = {
        {"alice", 85}, {"bob", 92}, {"charlie", 78}, {"diana", 95},
        {"eve", 88}, {"frank", 72}, {"grace", 91}, {"henry", 84}
    };

    vector<string> keys;
    for (auto& [k, v] : data) keys.push_back(k);

    // --- Demo 1: Hash function ---
    cout << "\n--- Polynomial Rolling Hash ---" << endl;
    printHashValues(keys, 11);
```

```
    cout << endl;
    printHashValues(keys, 7);

    // --- Demo 2: Separate Chaining ---
    cout << "\n--- Separate Chaining Hash Table ---" << endl;
    ChainingHashTable cht(7);  // Small table to show collisions

    for (auto& [key, val] : data) {
        cht.insert(key, val);
        cout << "  Inserted \"" << key << "\":" << val
             << " (load=" << fixed << setprecision(2) << cht.loadFactor() << ")" << endl;
    }
    cout << endl;
    cht.printTable("  Chaining table");

    cout << "\n  Search tests:" << endl;
    for (const string& k : {"alice", "bob", "zach"}) {
        auto result = cht.search(k);
        cout << "    \"" << k << "\": "
             << (result.has_value() ? to_string(result.value()) : "NOT FOUND") << endl;
    }

    cout << "\n  Delete \"charlie\"..." << endl;
    cht.remove("charlie");
    auto r = cht.search("charlie");
    cout << "  Search \"charlie\" after delete: "
         << (r.has_value() ? to_string(r.value()) : "NOT FOUND") << endl;

    // --- Demo 3: Linear Probing ---
    cout << "\n--- Linear Probing Hash Table ---" << endl;
    LinearProbingHashTable lpt(17);  // Larger table for open addressing

    for (auto& [key, val] : data) {
        lpt.insert(key, val);
        cout << "  Inserted \"" << key << "\":" << val
             << " (load=" << fixed << setprecision(2) << lpt.loadFactor()
             << ", clusters=" << lpt.countClusters() << ")" << endl;
    }
    cout << endl;
    lpt.printTable("  Linear probing table");

    cout << "\n  Search tests:" << endl;
    for (const string& k : {"diana", "frank", "zach"}) {
        auto result = lpt.search(k);
        cout << "    \"" << k << "\": "
             << (result.has_value() ? to_string(result.value()) : "NOT FOUND") << endl;
    }

    cout << "\n  Delete \"bob\"..." << endl;
    lpt.remove("bob");
    lpt.printTable("  After deleting \"bob\"");

    // Verify that search still works past a tombstone
    auto rr = lpt.search("eve");
    cout << "\n  Search \"eve\" (past tombstone): "
         << (rr.has_value() ? to_string(rr.value()) : "NOT FOUND") << endl;

    // --- Demo 4: Collision patterns at different load factors ---
    cout << "\n--- Collision Analysis ---" << endl;
    vector<string> manyKeys = {
        "apple", "banana", "cherry", "date", "elderberry",
        "fig", "grape", "honeydew", "kiwi", "lemon",
        "mango", "nectarine", "orange", "papaya", "quince"
    };
```

```cpp
    for (int tableSize : {7, 11, 17, 31}) {
        ChainingHashTable ct(tableSize);
        for (int i = 0; i < (int)manyKeys.size(); i++) {
            ct.insert(manyKeys[i], i);
        }
        auto lengths = ct.chainLengths();
        int maxChain = *max_element(lengths.begin(), lengths.end());
        int emptyBuckets = 0;
        for (int l : lengths) if (l == 0) emptyBuckets++;

        cout << "  Table size " << setw(2) << tableSize
             << ": load=" << fixed << setprecision(2) << ct.loadFactor()
             << ", max chain=" << maxChain
             << ", empty buckets=" << emptyBuckets
             << "/" << tableSize << endl;
        cout << "    Chain lengths: [";
        for (int i = 0; i < (int)lengths.size(); i++) {
            if (i > 0) cout << ", ";
            cout << lengths[i];
        }
        cout << "]" << endl;
    }

    cout << "\n  Key insight: lower load factor = fewer collisions, but more wasted space." << endl;
    cout << "  Chaining tolerates load > 1. Linear probing degrades sharply past ~0.75." << endl;

    return 0;
}
```