CSC 212: Data Structures and Abstractions

Hash Tables (part 2)

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

THINK BIG WE DO℠

Resizing a hash table

- Growing to a larger array when α exceeds a threshold
  - ✓ create a new table with larger capacity and rehash all the keys

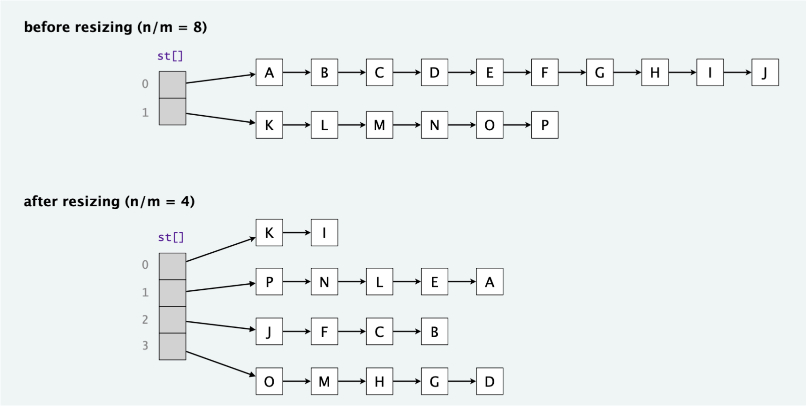

Image credit: COS 226 @ Princeton

Practice

- Insert the following keys into a hash of size M=4
  - 4, 2, 1, 10, 21, 32, 43, 3, 51, 71
- Resize the table to M=11

Open addressing

Open addressing

- Collision resolution mechanism
  - searching for next available slot (*probing*)
  - single-element per slot constraint, however requires careful deletion handling
  - assume duplicated keys are not allowed and $M \geq N$
- Core operations (assume a hash function h)
  - **insert**: if $h(key)$ is empty, place the new key (or key/value pair) there, otherwise, probe the table using a predetermined sequence until a slot is found
  - **search**: if $h(key)$ contains the key then return successfully, if not, probe the table using a predetermined sequence until either finding the key or an empty slot, which indicates that the key is not present in the table
  - **delete**: upon finding the key, **cannot mark the slot as empty**, as this would disrupt future search operations by prematurely terminating probe sequences, instead, mark the slot as deleted
- Comments
  - approach is more space-efficient than chaining, but it can be slower (better with $\alpha \approx 0.5$)

Probing

- Linear probing
  - probes next available index sequentially
  - $h(k, i) = (h'(k) + i) \mod m$
- Quadratic probing
  - probes next available index using a quadratic function
  - $h(k, i) = (h'(k) + i^2) \mod m$
- Double hashing
  - probes next available index using a secondary hash function h_2 (should not evaluate to 0)
  - $h(k, i) = (h'(k) + i \cdot h_2(k)) \mod m$

- m: table size
- i: probe number ($i = 0,1,2,\ldots$)
- $h'(k)$: initial hash value of key k
- $h(k, i)$: position for the i-th probe
- $h_2(k)$: secondary hash function

Practice

- Insert the following keys into a hash of size M=13
  - 4, 2, 1, 10, 21, 32, 43, 3, 51, 71, 17
  - linear probing
  - quadratic probing
  - double hashing
    - $h_2(k) = 1 + (k \mod 10)$

Image credit: CS106B @ Stanford

Data Structure	Worst-case			Average-case			Ordered?
	insert at	delete	search	insert at	delete	search	
sequential (unordered)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	No
sequential (ordered) binary search	O(n)	O(n)	O(log n)	O(n)	O(n)	O(log n)	Yes
BST	O(n)	O(n)	O(n)	O(log n)	O(log n)	O(log n)	Yes
2-3-4	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes
Red-Black	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes
Hash table (separate chaining)	O(n)	O(n)	O(n)	O(1)*	O(1)*	O(1)*	No
Hash table (open addressing)	O(n)	O(n)	O(n)	O(1)*	O(1)*	O(1)*	No

(*) assumes uniform hashing and appropriate load factor

Unordered associative containers (STL)

Unordered associative containers implement data structures that can be quickly searched – $O(1)$ average-case complexity

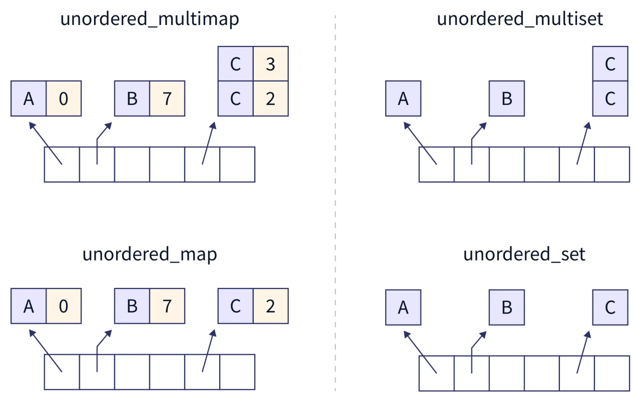

Image credit: https://www.scaler.com/topics/cpp/containers-in-cpp/

Practice

- Consider the code below that finds duplicate tokens on an input text
  - ✓ modify it in a way that it finds **rare tokens** (less than k occurrences)

```cpp
#include <iostream>
#include <unordered_set>
#include <string>
#include <sstream>

int main() {
    std::unordered_set<std::string> uniqueToks;
    std::unordered_set<std::string> duplicates;
    std::string line, token;

    while (std::getline(std::cin, line)) {
        std::istringstream stream(line);
        while (stream >> token) {
            if (!uniqueToks.insert(token).second) {
                duplicates.insert(token);
            }
        }
    }

    std::cout << duplicates.size() << " duplicate tokens found:\n";
    for (const auto& token : duplicates) {
        std::cout << token << " ";
    }
    std::cout << std::endl;

    return 0;
}
```


Comparing efficiency

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <unordered_set>
#include <string>
#include <chrono>
#include <algorithm>
#include <iomanip>

void testVector(const std::vector<std::string>& words) {
    auto start = std::chrono::high_resolution_clock::now();
    std::vector<std::string> vec;
    int duplicateCount = 0;
    for (const auto& word : words) {
        if (std::find(vec.begin(), vec.end(), word) != vec.end()) duplicateCount ++;
        else vec.push_back(word);
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Vector: " << duplicateCount << " duplicates, " << std::fixed << std::setprecision(6) << duration.count() << " seconds\n";
}

void testSet(const std::vector<std::string>& words) {
    auto start = std::chrono::high_resolution_clock::now();
    std::set<std::string> s;
    int duplicateCount = 0;
    for (const auto& word : words)
        if (!s.insert(word).second)
            duplicateCount++;
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Set: " << duplicateCount << " duplicates, " << std::fixed << std::setprecision(6) << duration.count() << " seconds\n";

}

void testUnorderedSet(const std::vector<std::string>& words) {
    auto start = std::chrono::high_resolution_clock::now();
    std::unordered_set<std::string> us;
    int duplicateCount = 0;
    for (const auto& word : words)
        if (!us.insert(word).second)
            duplicateCount++;
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Unordered Set: " << duplicateCount << " duplicates, " << std::fixed << std::setprecision(6) << duration.count() << " seconds\n";
}

int main() {
    std::vector<std::string> testWords;
    for (int i = 0 ; i < 1000000 ; i++)
        testWords.push_back("word" + std::to_string(i % 100000));
    std::cout << "Processing " << testWords.size() << " words...\n\n";
    testVector(testWords);
    testSet(testWords);
    testUnorderedSet(testWords);
    return 0;
}
```