MTH 4300/4299: Programming and Computer Science II

Lecture 19: Hashing

Jaime Abbariao

▓▓ Quiz

Fill in the function below. It should return the reverse of `s` using a `std::stack<char>`.

Example: `reverse("hello")` → `"olleh"`.

```
#include <stack>
#include <string>

std::string reverse(std::string s) {
  // TODO: use a std::stack<char>
}
```

```cpp
std::string reverse(std::string s) {
  std::stack<char> st;
  for (int i = 0; i < s.size(); i++) {
    st.push(s[i]);
  }
  std::string result = "";
  while (!st.empty()) {
    result += st.top();
    st.pop();
  }
  return result;
}
```

Push every character, then pop them all. Since a stack is LIFO, the last character pushed comes out first — that's the reversal. O(n) time, O(n) space.

The Lookup Problem

You have a pile of `(key, value)` pairs and you need fast lookup by key:

- Student ID → grade
- Username → account record
- Word → number of times it appears in a document

What we know so far:

Structure	Lookup	Insert
Unsorted `std::vector`	O(n)	O(1) amortized
Sorted `std::vector` + binary search	O(log n)	O(n)
Linked list	O(n)	O(1) at head

Today: a structure that gives O(1) average for both.

The Big Idea

If the key were already an integer in `[0, N)`, we could just use an array:

```
std::vector<int> grades(100);
grades[42] = 95;                // O(1) write
std::cout << grades[42];        // O(1) read
```

The array index is the lookup. No searching.

Problem: our keys aren't usually small integers. They're strings, big IDs, structs.

Idea: turn any key into a small integer with a function. Call it a hash function.

▓▓ Hash Function

A hash function `h` maps a key to an integer in `[0, M)`, where `M` is the size of our array (called the table).

```
h("apple")  → 3
h("banana") → 7
h("cherry") → 3   ← uh oh
```

Requirements:

- Deterministic — same key always hashes to the same slot.
- Fast — O(1) (or O(length) for strings).
- Spreads keys evenly across `[0, M)`.

When two different keys land in the same slot, that's a collision. We need a plan.

A Simple String Hash

Treat each character as a number and combine them:

```cpp
int hash_string(std::string s, int m) {
  int h = 0;
  for (int i = 0; i < s.size(); i++) {
    h = (h * 31 + s[i]) % m;
  }
  return h;
}
```

Why `31`? It's an odd prime — mixes bits well and is cheap on hardware (`31 * h == (h << 5) - h`). Java's `String.hashCode` uses the same constant.

The `% m` at each step keeps `h` from overflowing.

Trying It Out

```cpp
int main() {
  int m = 10;
  std::cout << hash_string("apple",  m) << std::endl;
  std::cout << hash_string("banana", m) << std::endl;
  std::cout << hash_string("cherry", m) << std::endl;
  std::cout << hash_string("grape",  m) << std::endl;
  return 0;
}
```

Different strings land in different slots — most of the time. With only 10 slots and enough keys, collisions are unavoidable (pigeonhole).

▒▒▒ Collisions Are Inevitable

With `M` slots and more than `M` keys, some slot must hold two keys. Even with far fewer keys, collisions happen surprisingly fast (the birthday paradox).

Two standard strategies:

1. Separate chaining — each slot holds a list of all keys that landed there.
2. Open addressing — on collision, probe other slots in the same table.

We'll focus on separate chaining. It's simpler and maps cleanly onto `std::vector` + `std::list`.

Separate Chaining — Picture

```
index   bucket
  0  →  [ ]
  1  →  [ ("cat", 1) ]
  2  →  [ ]
  3  →  [ ("apple", 5) → ("cherry", 2) ]
  4  →  [ ]
  ...
```

Slot 3 had a collision, so it holds a two-element list.

Lookup: hash the key, walk that one bucket. Insert: hash the key, append (or update) in that bucket. Erase: hash, walk, remove.

▓▓ HashMap Skeleton

We'll build a map from `std::string` to `int`.

```
struct Entry {
  std::string key;
  int value;
};

class HashMap {
public:
  HashMap(int m) : buckets_(m) {}
  void put(std::string key, int value);
  int  get(std::string key);
  bool contains(std::string key);
  void erase(std::string key);
private:
  std::vector<std::vector<Entry>> buckets_;
  int hash(std::string key);
};
```

Each bucket is a `std::vector<Entry>`. A linked list would work too — we use a vector for cache friendliness and simpler code.

Hash and Put

```
int HashMap::hash(std::string key) {
  int m = buckets_.size();
  int h = 0;
  for (int i = 0; i < key.size(); i++) {
    h = (h * 31 + key[i]) % m;
  }
  return h;
}

void HashMap::put(std::string key, int value) {
  int idx = hash(key);
  for (int i = 0; i < buckets_[idx].size(); i++) {
    if (buckets_[idx][i].key == key) {
      buckets_[idx][i].value = value;  // update
      return;
    }
  }
  buckets_[idx].push_back({key, value});  // insert
}
```

Get and Contains

```cpp
int HashMap::get(std::string key) {
  int idx = hash(key);
  for (int i = 0; i < buckets_[idx].size(); i++) {
    if (buckets_[idx][i].key == key) {
      return buckets_[idx][i].value;
    }
  }
  return -1;  // sentinel: "not found"
}

bool HashMap::contains(std::string key) {
  int idx = hash(key);
  for (int i = 0; i < buckets_[idx].size(); i++) {
    if (buckets_[idx][i].key == key) return true;
  }
  return false;
}
```

The inner loop only walks one bucket — not the whole table.

Erase

```
void HashMap::erase(std::string key) {
  int idx = hash(key);
  for (int i = 0; i < buckets_[idx].size(); i++) {
    if (buckets_[idx][i].key == key) {
      buckets_[idx].erase(buckets_[idx].begin() + i);
      return;
    }
  }
}
```

Walk the bucket, find the matching entry, remove it. Done.

```cpp
int main() {
  HashMap m(16);
  m.put("apple",  5);
  m.put("banana", 2);
  m.put("cherry", 9);

  std::cout << m.get("apple")  << std::endl;  // 5
  std::cout << m.get("banana") << std::endl;  // 2

  m.put("apple", 10);                          // update
  std::cout << m.get("apple")  << std::endl;  // 10

  m.erase("banana");
  std::cout << m.contains("banana") << std::endl;  // 0
  return 0;
}
```

▒▒▒ Cost Analysis

Let `n` = number of entries, `M` = number of buckets. Define the load factor `α = n / M`.

With a good hash function (keys spread evenly):

Operation	Average	Worst case
`put`	O(1 + α)	O(n)
`get`	O(1 + α)	O(n)
`erase`	O(1 + α)	O(n)

Average case O(1) as long as we keep `α` small (say, `α ≤ 1`).

Worst case O(n) if every key hashes to the same bucket — either a bad hash function or an adversarial input.

▓▓ Resizing

If we never grow the table, α climbs and everything slows down.

Rule: when α exceeds a threshold (e.g. 1.0), double M and rehash every entry.

```
void rehash(int new_m) {
  std::vector<std::vector<Entry>> old = buckets_;
  buckets_.assign(new_m, {});
  for (int i = 0; i < old.size(); i++) {
    for (int j = 0; j < old[i].size(); j++) {
      put(old[i][j].key, old[i][j].value);
    }
  }
}
```

Rehash costs O(n) — but it happens rarely. Amortized cost per insert stays O(1).

▓▓ std::unordered_map

The standard library already ships a hash-based map.

```cpp
#include <unordered_map>

std::unordered_map<std::string, int> counts;
counts["apple"]  = 5;
counts["banana"] = 2;
counts["apple"]++;                      // now 6

std::cout << counts["apple"] << std::endl;
std::cout << counts.size()  << std::endl;

if (counts.find("cherry") != counts.end()) {
  // found
}
```

- `operator[]` inserts a default (0 for `int`) if the key is missing.
- `find` returns an iterator — `end()` means not found.
- Average O(1) for insert, lookup, erase.

map vs. unordered_map

C++ gives you two associative containers. Pick on purpose.

	`std::map`	`std::unordered_map`
Backed by	balanced BST	hash table
Lookup	O(log n)	O(1) average
Insert	O(log n)	O(1) average
Keys in order?	yes (sorted)	no
Needs `<`	yes	no
Needs hash function	no	yes

Use `unordered_map` by default. Use `map` when you need keys in sorted order (e.g. range queries).

▓▓ Application: Word Count

Count how many times each word appears in a vector of words.

```cpp
std::unordered_map<std::string, int> count_words(
    std::vector<std::string> words) {
  std::unordered_map<std::string, int> counts;
  for (int i = 0; i < words.size(); i++) {
    counts[words[i]]++;
  }
  return counts;
}
```

`counts[word]++` does three things: default-construct `0` if missing, read, write. One line, O(1) average per word. Total: O(n).

▓▓▓ Application: Two Sum

Problem: given `a` and a target `t`, find two indices `i < j` with `a[i] + a[j] == t`.

Brute force: O(n²). With a hash map: O(n).

```cpp
std::vector<int> two_sum(std::vector<int> a, int t) {
  std::unordered_map<int, int> seen;  // value → index
  for (int i = 0; i < a.size(); i++) {
    int need = t - a[i];
    if (seen.find(need) != seen.end()) {
      return {seen[need], i};
    }
    seen[a[i]] = i;
  }
  return {};
}
```

For each element, ask: "have I already seen the number I need?" That's one O(1) lookup per index.

▓▓ Application: First Non-Repeating Character

Problem: in a string, return the first character that appears exactly once.

```cpp
char first_unique(std::string s) {
  std::unordered_map<char, int> counts;
  for (int i = 0; i < s.size(); i++) counts[s[i]]++;
  for (int i = 0; i < s.size(); i++) {
    if (counts[s[i]] == 1) return s[i];
  }
  return '\0';
}
```

Two passes: count everything, then find the first with count 1. O(n).

▓▓ std::unordered_set

Sometimes you only need the keys, not values — `unordered_set` is the map without the value.

```cpp
#include <unordered_set>

std::unordered_set<int> seen;
seen.insert(3);
seen.insert(7);

if (seen.count(3)) {  // 1 if present, 0 if not
  std::cout << "yes" << std::endl;
}
```

Great for: "have I seen this before?", deduplication, set membership tests.

When Hashing Hurts

Hashing is not free:

- Bad hash function → everything collides, O(n) per operation.
- No iteration order — don't rely on the order you get from a loop.
- No range queries — "give me all keys in `[a, b]`" is slow; use `std::map`.
- Rehashing invalidates iterators and references into the container.
- Hashing big objects can be expensive per call.

Default to `unordered_map`, but switch to `map` if you need sorted order or range queries.

Summary

- Hash function turns a key into a small integer → index into an array.
- Collisions are inevitable; separate chaining handles them by storing a list per bucket.
- Average O(1) for `put` / `get` / `erase` when the load factor stays small.
- Resize + rehash when buckets get full. Amortized O(1) stays.
- Use `std::unordered_map` and `std::unordered_set` by default; use `std::map` when you need ordered keys.

Next lecture: trees and binary search trees.

▓▓ Lab: Character Frequency — Setup

Write a program that reads a string from the user and prints the frequency of each character, using `std::unordered_map<char, int>`.

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

std::unordered_map<char, int> char_frequency(std::string s) {
  // TODO
}

int main() {
  std::string s;
  std::cout << "Enter a string: ";
  std::getline(std::cin, s);

  std::unordered_map<char, int> counts = char_frequency(s);
  for (auto it = counts.begin(); it != counts.end(); ++it) {
    std::cout << it->first << ": " << it->second << std::endl;
  }
  return 0;
}
```

▒▒▒ Lab: Expected Behavior

Input:

```
hello world
```

Possible output (order may vary — `unordered_map` makes no promises):

```
h: 1
e: 1
l: 3
o: 2
 : 1
w: 1
r: 1
d: 1
```

Hints:

- `char_frequency` is a single loop over the string.
- `counts[c]++` handles both "first time seen" and "already counted" in one line.
- Don't forget spaces count as characters — that's why `' '` shows up with count 1.