LECTURE 32

Hashing and Efficient Data Access

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

WHAT THIS LECTURE COVERS

Agenda

1. The need for O(1) data access, and the hashing concept
2. Hash tables, hash functions, and what makes one "good"
3. Collisions, and open vs. closed hashing
4. Separate chaining, linear probing, and double hashing
5. Rehashing, worked through with real before/after table sizes
6. Chaining vs. probing performance, and real applications of hashing

THE FASTEST DATA ACCESS THIS COURSE COVERS

Hashing Concept

- A BST (Unit 5) gets search to O(log n). Hashing asks a bolder question: what if you could compute exactly where a value belongs, and skip comparisons entirely?


```
table[5] = ("apple", value)
```

- A hash table stores key-value pairs, using a hash function to convert a key directly into an array index — the same O(1) array indexing from Lecture 4, now applied to arbitrary keys.

TURNING A KEY INTO AN INDEX

Hash Function

hash_table_chaining.cpp

```cpp
int hashFunction(const string& key, int tableSize) {
    int sum = 0;
    for (char c : key) sum += c;
    return sum % tableSize;
}
```

```
hash("apple", 7) =          5

hash("cherry", 7) =         2

hash("fig", 7) =            2

hash("grape", 7) =          2
```

- Sums ASCII values of every character, then takes modulo the table size.
- A simple hash — easy to understand, but poor in practice.
- Notice: "cherry", "fig", and "grape" all hash to index 2 — a COLLISION.

AND WHAT A COLLISION IS

Properties of a Good Hash Function

✓ **Deterministic**

The same key always produces the same index.

✓ **Fast to compute**

The whole point is speed — a slow hash defeats the purpose.

✓ **Uniform distribution**

Keys should spread evenly, not cluster in a few indices.

A collision happens when two different keys hash to the same index.

This is COMMON, not rare — "cherry"/"fig"/"grape" collided at table size 7. A second, independent check confirms it isn’t a fluke: "lemon"/"melon"/"cocoa" all hash to 0 at table size 11, despite sharing no letters and different lengths.

TABLE SIZE 7: INSERT APPLE, CHERRY, FIG, GRAPE

A Collision, Visualized

0	(empty)
1	(empty)
2	**cherry → fig → grape** ← *3-entry chain (collision)*
3	(empty)
4	(empty)
5	apple
6	(empty)

Five of seven buckets sit empty while bucket 2 holds a three-entry chain — exactly the "clustering" a good hash function tries to avoid.

HANDLING COLLISIONS WITH A LIST PER BUCKET

Open Hashing: Separate Chaining

hash_table_chaining_full.cpp

```
void insert(const string& key, int value) {
    int index = hashFunction(key);
    table[index].push_back({key, value});
}

bool search(const string& key, int& valueOut) {
    int index = hashFunction(key);
    for (auto& p : table[index])
        if (p.first == key) { valueOut = p.second; return true; }
    return false;
}
```

- Every key that hashes to an index gets appended to that index's list.
- search() still finds a match correctly — it just checks each entry in the bucket instead of one direct step.
- Found "fig": 40 (correctly found despite the collision).

KEEPING EVERY ENTRY IN THE ARRAY ITSELF

Closed Hashing: Linear Probing

- On a collision, checks index+1, then index+2, wrapping around, until an empty slot is found.

"cherry", "fig", and "grape" all "want" index 2 — instead of a list, linear probing physically walks forward: "fig" lands at 3 (1 probe), "grape" at 4 (2 probes).

IMPLEMENTATION, PLUS TWO ALTERNATIVES

Linear Probing — Code, and Beyond

hash_table_probing.cpp

```
int index = hashFunction(key);
while (occupied[index]) {
    index = (index + 1) % tableSize;  // wrap around
}
keys[index] = key; occupied[index] = true;
```

- Linear probing's weakness: CLUSTERING — once keys land near each other, new collisions land there too, more often.
- Quadratic probing and double hashing both address clustering by spreading probes out more.

Quadratic probing

Checks index+1^2, +2^2, +3^2 — spreads probes out faster.

Double hashing

A second hash function computes the probe step size itself.

GROWING THE TABLE BEFORE LOAD FACTOR GETS TOO HIGH

Rehashing

- When load factor (entries ÷ table size) exceeds a threshold, a new, larger table is built and every key re-inserted.

Before rehash — size 4, load factor 1.00

0	(empty)
1	cherry
2	apple→fig
3	grape

table size
4 → 8

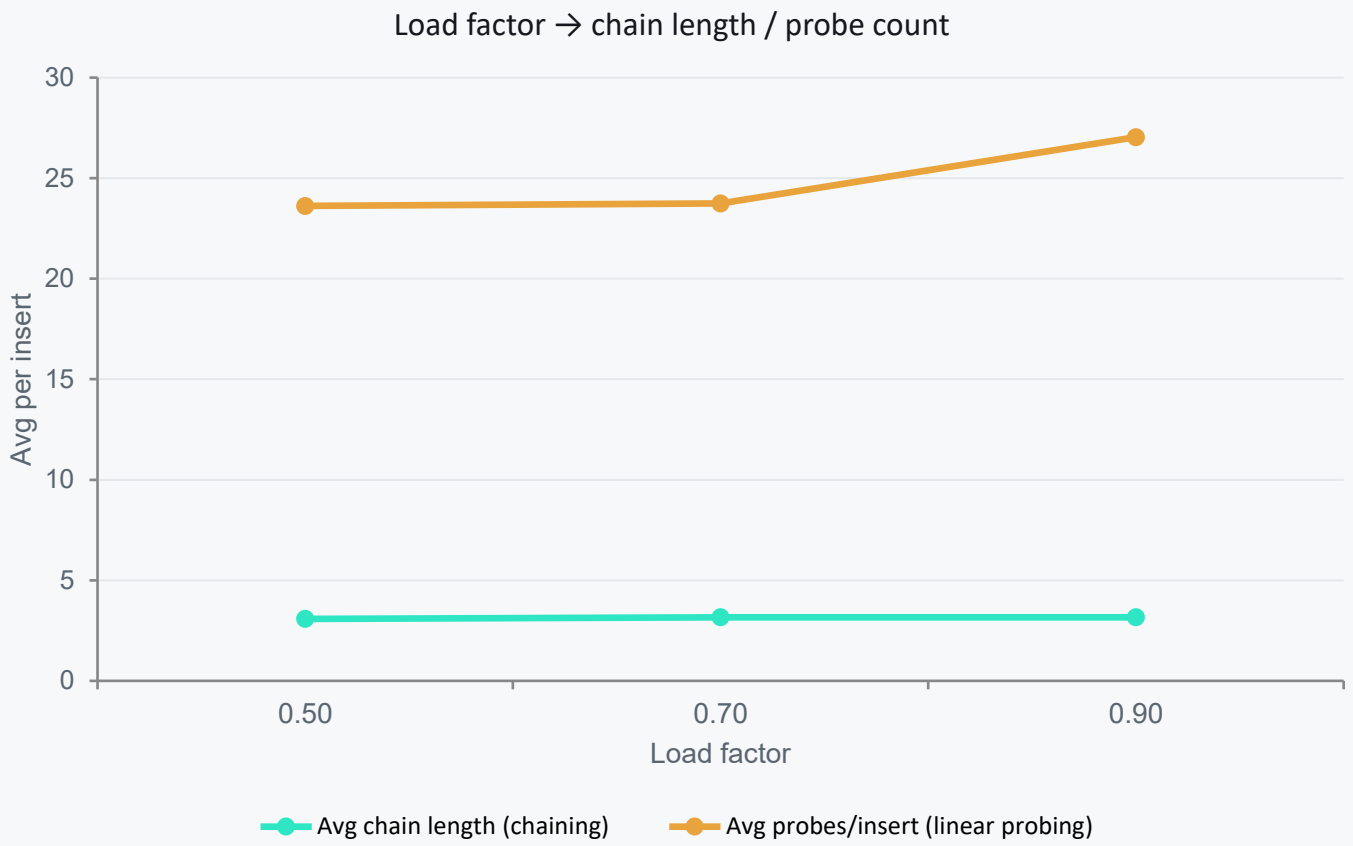
After rehash — size 8

0	(empty)
1	(empty)
2	apple→mango
3	lemon
4	kiwi
5	cherry
6	fig
7	grape

Every key’s index is recomputed from scratch at the new size — "fig" moved to bucket 6 alone, while "apple" stayed at bucket 2 (later joined by "mango"). Cost amortizes to O(1) per insertion.

80 KEYS, WEAK HASH FUNCTION, THREE LOAD FACTORS

Chaining vs. Probing: Measured Data



- Chaining's average chain length stays a modest ~3 across all load factors.
- Linear probing's average probe count balloons past 20 and keeps climbing.
- Probing is far more sensitive to a poor hash function than chaining is.

WHERE THIS SHOWS UP EVERY DAY

Applications of Hashing

1

Dictionaries & maps

Python dict, JS objects, C++ unordered_map.

2

Caching

Browser cache, CPU cache, CDN edge cache — "do I already have this?" in O(1).

3

Password storage

Passwords are hashed so the original is never stored in plain text.

4

Duplicate detection

Checking if a value has been seen before, O(1) per check.

CLOSING

Key Takeaways

- Hashing computes an index directly from a key — O(1) average-case access, at the cost of giving up ordering.
- A collision — two keys hashing to the same index — is common, not rare; every real hash table needs a resolution strategy.
- Separate chaining stores a list per index; linear probing keeps every entry in the array, searching forward for empty slots.
- Rehashing grows the table and re-inserts everything once the load factor gets too high, keeping O(1) average performance.
- Chaining tolerates a weak hash function far better than probing does — measured data showed chain length ~3 vs. probes climbing past 20.

COURSE COMPLETE

You’ve Reached the End of CSC211

Algorithms and Data Structures

- You started with one question: "how should data be organized?" — and built, in real compiled C++, every major answer to it.
- Linear structures that trade access speed against insertion cost; trees that turn search into a logarithmic walk; graphs that model any network of relationships; and now hashing, which sidesteps search entirely with direct computation.
- Every structure here reappears constantly ahead: databases index tables with trees and hash tables, operating systems schedule with queues, compilers parse with trees and stacks.

The goal was never to memorize implementations — it was to build the habit of asking: what does this problem need to do quickly, and which structure gives me that?