Hash Tables

A **hash table** is an **associative data structure** that maps **keys to values**. Unlike sequential data structures such as arrays or linked lists, which organize elements by position, hash tables provide access to values via arbitrary keys. This allows for efficient average-case performance for **insertion**, **deletion**, and **lookup** operations—typically in $\mathcal{O}(1)$ time.

Hash tables are widely used in compilers, databases, caches, and dictionaries, where fast retrieval by key is critical.

1 Use Cases

- Implementing associative arrays or maps
- Caching computed results (memoization)

2 Time Complexity

Operation	Average Case	Worst Case
Insert	$\mathcal{O}(1)$	$\mathcal{O}(n)$ (with poor hash function or high load)
Lookup	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Update	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Delete	$\mathcal{O}(1)$	$\mathcal{O}(n)$

3 Underlying Implementation

Hash tables are typically backed by an array. A **hash function** maps a key to an index in this array:

$$\texttt{index} = \texttt{hash(key)} \mod \texttt{table\_size}$$

Example Hash Function (for Strings):

The following hash function maps a string key to an array index by summing the ASCII values of its characters:

```
int hash(const string& key, int table_size) {
  int sum = 0;
  for (char c : key)
    sum += static_cast<int>(c);
  return sum % table_size;
}
```

Why use `% table_size`?

Hash functions often produce large integers. The modulo operation ensures the result is a valid index within the bounds of the underlying array (typically of size `table_size`). For example:

- If $\texttt{sum} = 615$ and $\texttt{table\_size} = 10$, then the index is `615 % 10 = 5`.
- This maps the key to index 5 in the internal array.

This hash function is simple but not collision-resistant. For example:

- `"top"` → ASCII sum = `116 + 111 + 112 = 339`
- `"pot"` → ASCII sum = `112 + 111 + 116 = 339`

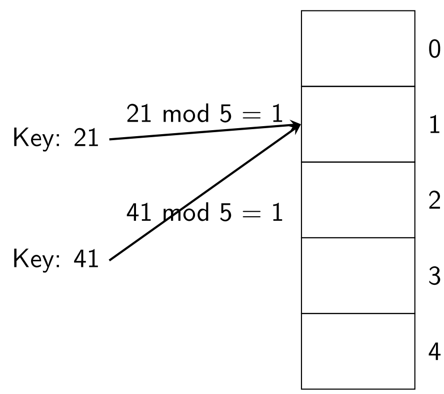
Both strings produce the same hash value:

$$\texttt{339 \% table\_size} = \texttt{same index}$$

Conclusion: Anagrams like `"top"` and `"pot"` will collide under this hash function. This illustrates the importance of using more sophisticated hash functions that consider character positions and order.

3.1 Visualizing a Hash Collision

The diagram below illustrates two different keys mapping to the same index, resulting in a collision:



Observation: Keys 21 and 41 both hash to index 1, resulting in a collision that must be resolved.

4 Why Hash Tables Can Perform Poorly

While hash tables offer average-case constant time, performance degrades in the following scenarios:

- **High Load Factor:** As the number of entries approaches table size, collisions increase and probing chains grow longer.
- **Poor Hash Functions:** If a hash function does not distribute keys uniformly, many values may cluster at the same index.

Mitigation Strategies:

- Use a well-designed hash function (e.g., MurmurHash, SipHash).
- Resize the table dynamically and maintain a low load factor (e.g., below 0.75).
- Choose collision resolution wisely (e.g., quadratic probing, double hashing, cuckoo hashing).

5 Collision Resolution Strategies

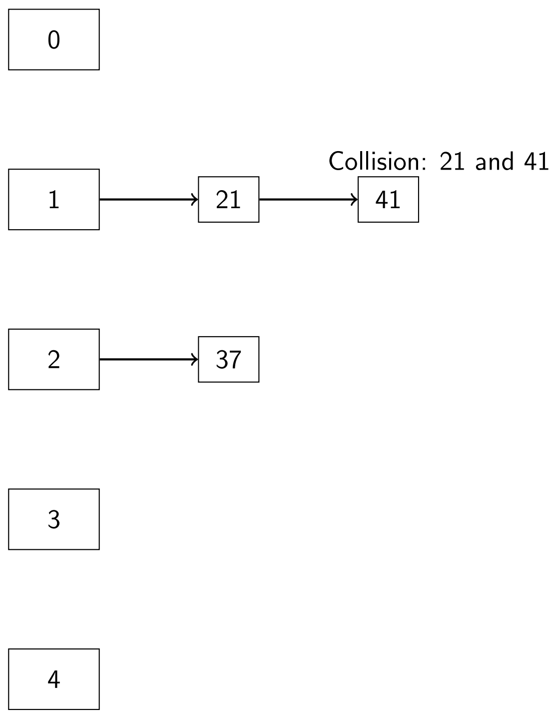
Since different keys may hash to the same index (a collision), hash tables use various strategies to resolve them:

5.1 Chaining (Separate Chaining)

In separate chaining, each hash table slot holds a pointer to a linked list (or another dynamic structure) of entries that hash to the same index. New entries are appended to the list when collisions occur.

Pros: Simple and flexible; handles collisions gracefully even when the load factor exceeds 1. **Cons:** Requires additional memory for pointers; performance degrades if lists grow too long.

C++ STL: `unordered_map` uses separate chaining with buckets internally.



Explanation:

- Keys 21 and 41 both hash to index 1, forming a linked list.
- Key 37 hashes to index 2 and occupies that slot alone.
- Each slot acts as the head of a chain.

5.2 Linear Probing

If a collision occurs at index i, search sequentially at $i+1, i+2, \ldots$ until an empty slot is found.

```
int probe(int key, int table_size, int i) {
  return (hash(key, table_size) + i) % table_size;
}
```

Pros: Simple, good cache performance. **Cons:** Prone to clustering, performance degrades as table fills.

5.3 Cuckoo Hashing

Uses two hash functions and two tables. If insertion causes a collision, evict the resident key and relocate it using the other hash function.

Pros: Constant-time worst-case lookup. **Cons:** Complex insertion logic and potential for infinite loops (requiring rehashing).

5.4 Modern Hash Functions

Robust hash functions are essential to minimize collisions and ensure uniform distribution:

- `std::hash` (C++ STL): Simple, type-specialized
- **MurmurHash3**: Non-cryptographic, fast and well-distributed
- **CityHash**, **FarmHash** (Google): Optimized for speed on large inputs
- **SipHash**: Secure hash for short strings; used in Python dictionaries

6 STL unordered_map

C++ provides a standard hash table implementation via `unordered_map`:

```
#include <unordered_map>
using namespace std;

unordered_map<string, int> freq;
freq["apple"] = 2;
freq["banana"] = 1;
cout << freq["apple"]; // prints 2
```

Key Characteristics:

- Backed by a hash table
- Average $\mathcal{O}(1)$ for insert, find, and erase
- Allows custom hash functions via `std::hash` specializations

7 STL unordered_set

The C++ Standard Template Library provides `std::unordered_set`, an associative container that stores unique elements using a hash table for fast access.

7.1 Key Features

- Stores only keys (no associated values)
- Ensures uniqueness of elements
- Average-case $\mathcal{O}(1)$ time complexity for insert, erase, and find
- Backed by hash tables with separate chaining

7.2 Basic Usage

```
#include <unordered_set>
#include <iostream>
using namespace std;

int main() {
  unordered_set<string> dict;
  dict.insert("apple");
  dict.insert("banana");
  dict.insert("apple"); // duplicate, ignored

  if (dict.find("banana") != dict.end())
    cout << "banana exists\n";

  dict.erase("apple");
}
```

8 Custom Hash Functions

To use custom types (e.g., structs), you must define both a hash function and an equality comparator:

```
struct Point {
  int x, y;
  bool operator==(const Point& other) const {
    return x == other.x && y == other.y;
  }
};

struct PointHash {
  size_t operator()(const Point& p) const {
    return hash<int>()(p.x) ^ (hash<int>()(p.y) << 1);
  }
};

unordered_set<Point, PointHash> points;
```

9 Practice problem

Leetcode Problem 1: *Two sum - O(n)*
`https://leetcode.com/problems/two-sum/description/`