Hash Tables

Expected O(1) Key-Value Operations

Minseok Jeon
DGIST

November 2, 2025

Outline

Introduction to Hash Tables

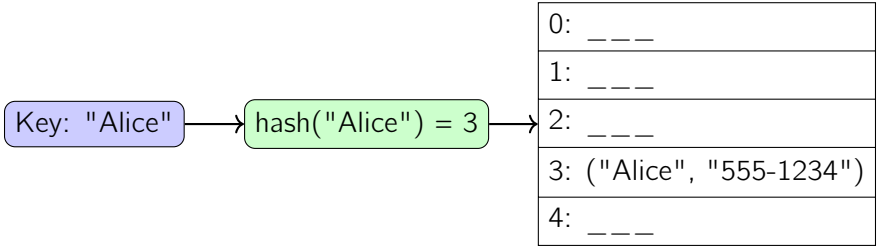
What is a Hash Table?

Definition: A data structure that maps keys to values using a hash function.

Key Components:

- **Hash function:** Converts keys to array indices
- **Array (table):** Stores key-value pairs
- **Collision handling:** Manages keys that hash to same index

Basic Idea:



Performance: Expected O(1) for insert, search, delete

Why Use Hash Tables?

Advantages:

- **Fast lookups:** Average O(1) vs O(log n) for trees
- **Simple interface:** Natural key-value mapping
- **Flexible keys:** Strings, numbers, tuples, etc.
- **Widely used:** Python dicts, Java HashMap, C++ unordered_map

Common Applications:

- Dictionaries/maps (phone book, student records)
- Caching (LRU cache, memoization)
- Sets (unique element tracking)
- Frequency counting (word count)
- Database indexing
- Deduplication

Hash Functions

What Makes a Good Hash Function?

Essential Properties:

1. **Deterministic:** Same key always produces same hash
2. **Uniform distribution:** Keys spread evenly across table
3. **Fast to compute:** O(1) or O(k) where k = key length
4. **Minimize collisions:** Different keys rarely map to same index
5. **Avalanche effect:** Small key change → large hash change

Hash Function Signature:

$$\text{hash(key)} \rightarrow \text{integer in range [0, table\_size - 1]}$$

Goal: Distribute keys uniformly to minimize collisions

Common Hash Functions: Division Method

Simplest approach: Use modulo operator

```
def hash_division(key, table_size):
    return key % table_size
```

Example:

- Key = 42, Table size = 13
- hash(42) = 42 % 13 = 3

Best Practice: Use prime table sizes

- Prime numbers: 53, 97, 193, 389, 769, 1543, ...
- Better distribution
- Fewer collisions

Trade-off: Powers of 2 are faster (bitwise AND) but worse distribution

Common Hash Functions: Multiplication Method

Idea: Multiply by constant, extract fractional part

```
def hash_multiplication(key, table_size):
    A = 0.6180339887  # (sqrt(5) - 1) / 2, golden ratio
    return int(table_size * ((key * A) % 1))
```

Advantages:

- Works well with any table size
- Good distribution with golden ratio constant
- Independent of table size choice

Example:

- Key = 123, A = 0.618, Table size = 100
- $123 \times 0.618 = 76.014$
- Fractional part: 0.014
- Index: $\lfloor 100 \times 0.014 \rfloor = 1$

String Hashing: Polynomial Rolling Hash

Challenge: Hash strings efficiently

```python
def hash_string(s, table_size):
    hash_val = 0
    p = 31  # Prime base
    p_pow = 1

    for char in s:
        hash_val = (hash_val + ord(char) * p_pow) % table_size
        p_pow = (p_pow * p) % table_size

    return hash_val
```

Example: "hello"

- h: 104×31^0
- e: 101×31^1
- l: 108×31^2

Bad Hash Function Example

What NOT to do:

```
# BAD: Only uses first character
def bad_hash(s, table_size):
    return ord(s[0]) % table_size
```

Problem: Many collisions!

- "apple" → ord('a') = 97
- "ant" → ord('a') = 97
- "arrow" → ord('a') = 97
- All hash to same index!

Another bad example:

```
# BAD: All 5-letter words collide
def bad_hash2(s, table_size):
    return len(s) % table_size
```

Load Factor and Resizing

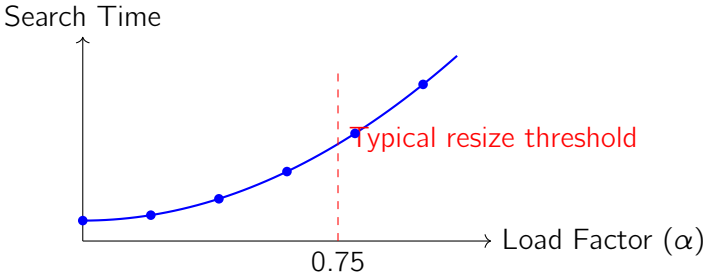
Load Factor (α)

Definition: $\alpha = \frac{n}{m}$ where

- n = number of elements
- m = table size

Meaning: Average number of elements per slot

Impact on Performance:



Resizing Strategy

```python
class HashTable:
    def __init__(self, initial_size=16):
        self.size = initial_size
        self.count = 0
        self.table = [[] for _ in range(self.size)]

    def load_factor(self):
        return self.count / self.size

    def resize(self):
        # Double the size
        old_table = self.table
        self.size *= 2
        self.table = [[] for _ in range(self.size)]
        self.count = 0

        # Rehash all elements
        for bucket in old_table:
            for key, value in bucket:
```

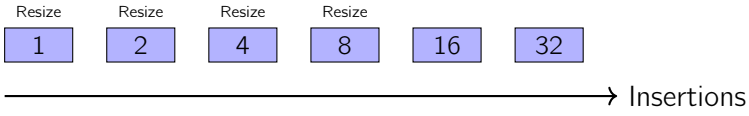
Amortized Analysis of Resizing

Question: Is resizing expensive?

Analysis:

- Insert n elements
- Resizes occur at: 1, 2, 4, 8, 16, ..., n
- Total rehashing cost: $1 + 2 + 4 + \ldots + n = 2n - 1 = O(n)$
- Amortized cost per insertion: $O(n) / n =$ **O(1)**

Visualization:

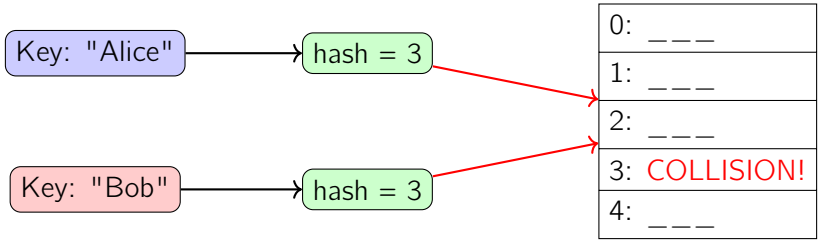


Key insight: Expensive resizes are rare, so average cost is O(1)

Collision Handling

Collision: When Two Keys Hash to Same Index

Problem: Different keys can hash to the same index

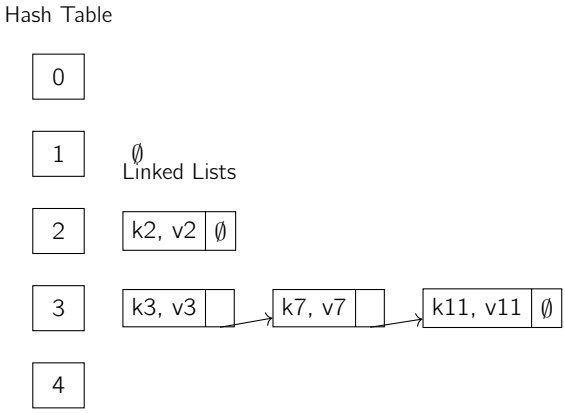


Two Main Solutions:

1. **Separate Chaining:** Store multiple elements at each index (linked list)
2. **Open Addressing:** Find another empty slot in the table

Separate Chaining

Idea: Each table slot stores a list of colliding elements



Advantages:

- Simple to implement

Separate Chaining Implementation

```python
class HashTableChaining:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def insert(self, key, value):
        index = hash(key) % self.size
        bucket = self.table[index]

        # Update if key exists
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return

        # Add new key-value pair
        bucket.append((key, value))

    def search(self, key):
```

Open Addressing

Idea: All elements stored in table, probe for empty slots

New key hashes to index 1 (occupied)
Probe sequence:
$1 \rightarrow 2 \rightarrow 3 \rightarrow \mathbf{4}$

0: (k0, v0)
1: OCCUPIED
2: (k2, v2)
3: Try next $\rightarrow$
4: Empty
5: (k5, v5)
6: Empty
7: Empty

Advantages:

- Better cache locality

Comparison: Chaining vs Open Addressing

Feature	Chaining	Open Addressing
Memory	More (pointers)	Less (no pointers)
Cache locality	Poor	Good
Load factor	Can exceed 1.0	Must stay < 0.7
Deletion	Simple	Complex (tombstones)
Clustering	No	Yes
Implementation	Easier	Harder
Best for	General use	Cache-friendly, known size

Recommendation:

- Use chaining for most applications (simpler, more flexible)
- Use open addressing for performance-critical, cache-sensitive code

Open Addressing Strategies

Linear Probing

Formula: h(k, i) = (h(k) + i) % m
Probe sequence: h(k), h(k)+1, h(k)+2, h(k)+3, ...

Example: Key hashes to index 3, table size = 10

$$\text{Probe: } 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 0 \to 1 \to 2$$

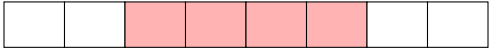
Advantages:

- Simple to implement
- Good cache locality (sequential access)

Disadvantages:

- **Primary clustering:** Long runs of occupied slots form

Clustering Example:



Cluster grows!

Quadratic Probing

Formula: $h(k, i) = (h(k) + i^2) \% m$
Probe sequence: h(k), h(k)+1, h(k)+4, h(k)+9, h(k)+16, ...

Example: Key hashes to index 3, table size = 10

i	Offset	Index
0	0	3
1	1	4
2	4	7
3	9	2
4	16	9

Advantages:

- Reduces primary clustering
- Better distribution than linear

Disadvantages:

- **Secondary clustering:** Keys with same hash follow same probe sequence

Double Hashing

Formula: $h(k, i) = (h_1(k) + i \times h_2(k)) \% m$

Two hash functions:

- $h_1(k)$: Initial position
- $h_2(k)$: Step size (must be coprime with m)

Example: $h_1(k) = 3$, $h_2(k) = 7$, table size = 10

i	Offset	Index
0	0	3
1	7	0
2	14	7
3	21	4
4	28	1

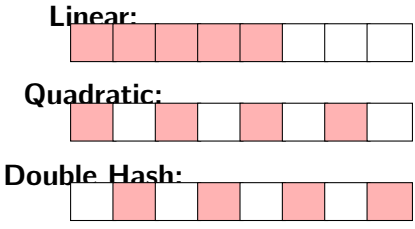
Advantages:

- Eliminates both primary and secondary clustering
- Best distribution among open addressing methods

Comparison of Probing Methods

Method	Clustering	Complexity	Distribution
Linear	Primary	Simple	Poor
Quadratic	Secondary	Moderate	Good
Double Hashing	None	Complex	Excellent

Clustering Visualization:



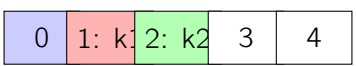
Deletion in Open Addressing

The Deletion Problem

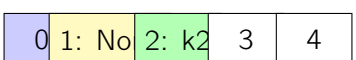
Challenge: Cannot simply set slot to None

Example Problem:

1. Insert k1 at index 1, k2 at index 2 (k2 collided, probed to 2)
2. Delete k1 (set index 1 to None)
3. Search for k2:
   - Start at index 1
   - Find None → stop searching
   - k2 is "lost" even though it's at index 2!



Before delete



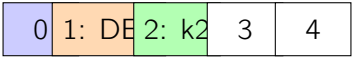
After delete - k2 unreachable!

Solution: Tombstones (Lazy Deletion)

Idea: Mark deleted slots with special DELETED marker

Rules:

- **Insert:** Can place at None or DELETED slots
- **Search:** Skip over DELETED, continue probing
- **Delete:** Mark slot as DELETED (not None)



With tombstone - k2 still reachable

Tombstone Issues:

- DELETED markers accumulate over time
- Degrade search performance
- Waste space

Deletion Implementation

```python
class HashTableWithDeletion:
    DELETED = object()  # Sentinel value

    def delete(self, key):
        for i in range(self.size):
            index = (hash(key) + i) % self.size

            if self.table[index] is None:
                return False  # Not found

            if self.table[index] is not self.DELETED:
                if self.table[index][0] == key:
                    self.table[index] = self.DELETED
                    return True

        return False

    def search(self, key):
        for i in range(self.size):
```

Applications

Application: Dictionaries / Maps

Most common use case: Key-value storage

```python
# Python dict (hash table implementation)
phonebook = {
    "Alice": "555-1234",
    "Bob": "555-5678",
    "Charlie": "555-9012"
}

# O(1) average case operations
phone = phonebook["Alice"]  # Lookup
phonebook["David"] = "555-3456"  # Insert
del phonebook["Bob"]  # Delete
exists = "Charlie" in phonebook  # Membership test
```

Real-world examples:

- Student records (ID → student info)

Application: Caching (LRU Cache)

```python
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key):
        if key not in self.cache:
            return -1
        # Move to end (most recently used)
        self.cache.move_to_end(key)
        return self.cache[key]

    def put(self, key, value):
        if key in self.cache:
            self.cache.move_to_end(key)
        self.cache[key] = value
        if len(self.cache) > self.capacity:
```

Application: Frequency Counting

```python
from collections import Counter

# Count word frequencies
text = "the quick brown fox jumps over the lazy dog"
word_count = Counter(text.split())
print(word_count)
# Counter({'the': 2, 'quick': 1, 'brown': 1, 'fox': 1, ...})

# Most common words
print(word_count.most_common(3))
# [('the', 2), ('quick', 1), ('brown', 1)]

# Manual implementation
def count_frequencies(items):
    freq = {}
    for item in items:
        freq[item] = freq.get(item, 0) + 1
```

Application: Two Sum Problem

Problem: Find two numbers that sum to target

```python
def two_sum(nums, target):
    """
    Given array and target, return indices of two numbers
    that add up to target.

    Time: O(n), Space: O(n)
    """
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
    return None

# Example
```

Application: Deduplication

```python
def remove_duplicates(arr):
    """Remove duplicates while preserving order"""
    seen = set()
    result = []
    for item in arr:
        if item not in seen:
            seen.add(item)
            result.append(item)
    return result

# Example
arr = [1, 2, 3, 2, 4, 1, 5]
print(remove_duplicates(arr))
# [1, 2, 3, 4, 5]

# Using set (loses order)
unique = list(set(arr))
```

Complexity and Pitfalls

Time Complexity

Operation	Average	Worst	Notes
Insert	O(1)	O(n)	Worst: all keys collide
Search	O(1)	O(n)	Worst: all keys collide
Delete	O(1)	O(n)	Worst: all keys collide
Resize	O(n)	O(n)	Amortized O(1) per insert

Space Complexity:

- **Chaining:** O(n + m) where n = elements, m = table size
- **Open addressing:** O(m), must keep load factor low

Key Point: Expected O(1) depends on:

- Good hash function (uniform distribution)
- Reasonable load factor (< 0.75)
- Proper collision handling

Common Pitfall 1: Mutable Keys

Problem: Using mutable objects as keys

```
# BAD: Lists are mutable, can't be hashed
d = {[1, 2]: "value"}  # TypeError: unhashable type: 'list'

# BAD: Dictionaries are mutable
d = {{1: 2}: "value"}  # TypeError: unhashable type: 'dict'

# GOOD: Use immutable types
d = {(1, 2): "value"}              # Tuples work
d = {frozenset([1, 2]): "value"}   # Frozen sets work
d = {"key": "value"}               # Strings work
d = {42: "value"}                  # Integers work
```

Rule: Keys must be immutable and hashable

- **Hashable:** int, float, str, tuple, frozenset
- **Not hashable:** list, dict, set

Common Pitfall 2: Poor Hash Function

```python
# BAD: All strings of same length collide
def bad_hash(s):
    return len(s) % 10

# All 5-letter words map to index 5!
# "hello", "world", "apple" -> all collide

# BAD: Only uses first character
def bad_hash2(s):
    return ord(s[0]) % 10

# "ant", "apple", "arrow" -> all collide
```

Consequences:

- Many collisions
- Performance degrades to O(n)

Common Pitfall 3: Ignoring Load Factor

```python
# BAD: Never resize
class BadHashTable:
    def __init__(self):
        self.table = [[] for _ in range(10)]
        # Fixed size!

    def insert(self, key, value):
        index = hash(key) % 10
        self.table[index].append((key, value))
        # Just keep inserting...
        # Performance degrades to O(n)!
```

Problem: With 1000 elements in size-10 table:

- Load factor = 100
- Average chain length = 100
- Search time = O(100) = O(n)

When NOT to Use Hash Tables

Hash tables are NOT suitable when you need:

- **Ordered iteration:** Use BST or sorted array
  - Hash tables don't maintain order
- **Range queries:** Use BST or B-tree
  - "Find all keys between 10 and 20"
- **Minimum/maximum:** Use heap
  - Hash tables need O(n) to find min/max
- **Worst-case guarantees:** Use balanced trees
  - Hash tables can degrade to O(n)
- **Memory constrained:** Consider alternatives
  - Hash tables waste space (load factor < 1)

Summary

Key Concepts Recap

Hash Table Fundamentals:

- Hash function maps keys to array indices
- Expected O(1) insert, search, delete
- Trade space for time

Good Hash Function:

- Deterministic, uniform, fast
- Minimize collisions
- Common methods: division, multiplication, polynomial

Load Factor Management:

- $\alpha = n/m$ (elements / table size)
- Resize when $\alpha > 0.75$ (chaining) or $\alpha > 0.5$ (open addressing)
- Amortized O(1) resizing cost

Collision Handling Recap

Separate Chaining:

- Store lists at each index
- Simple, never fills up
- Extra memory, poor cache locality

Open Addressing:

- Probe for next empty slot
- Better cache locality, no pointers
- Complex deletion (tombstones)
- Three methods:
  - Linear probing (simple, primary clustering)
  - Quadratic probing (better, secondary clustering)
  - Double hashing (best, no clustering)

Applications Recap

Common Uses:

- Dictionaries/maps (key-value storage)
- Caching (LRU cache)
- Sets (unique elements)
- Frequency counting
- Database indexing
- Deduplication
- Two sum and related problems

Best Practices:

- Use immutable keys only
- Choose good hash function
- Monitor and maintain load factor
- Resize when needed
- Use chaining for most cases

Practice Problems

Basic:

- Implement hash table with chaining
- Implement hash table with linear probing
- Design hash function for strings

Intermediate:

- LRU Cache (LeetCode 146)
- Two Sum (LeetCode 1)
- Group Anagrams (LeetCode 49)
- First Unique Character (LeetCode 387)
- Implement resizing with load factor

Advanced:

- Design hashmap with all operations (LeetCode 706)
- LFU Cache (LeetCode 460)
- Implement quadratic probing
- Implement double hashing

Further Learning

Advanced Topics:

- Universal hashing families
- Perfect hashing
- Cuckoo hashing
- Robin Hood hashing
- Bloom filters (probabilistic hash structures)

Resources:

- CLRS: Chapter 11 (Hash Tables)
- Practice on LeetCode hash table tag
- Study Python dict implementation (CPython source)
- Analyze hash functions with test data

Projects:

- Build your own hash table library
- Implement LRU cache from scratch
- Create spell checker with hash table

Thank You!

Questions?

Hash Tables: Fast Average-Case Performance