# Hashing

## Data Structures and Algorithms Tutorial

IIITH - S26

Hash Tables, Collision Resolution, Chaining

# Agenda

1. Overview
2. Open Addressing
3. Separate Chaining
4. Comparison
5. Rehashing
6. Practice Problems
7. Summary

# Overview

# What is Hashing?

Main Idea

- Map large key space to small array indices
- $O(1)$ average case for search, insert, delete!

Components

- Hash function: Key $\rightarrow$ Index
- Hash table: Array of fixed size
- Collision handling: What when two keys map to same index?

The Trade-off

- Time vs Space
- Larger table = fewer collisions but more memory

# Hash Functions

Properties of Good Hash Function

- Deterministic: Same key always gives same hash
- Uniform: Distributes keys evenly across indices
- Fast: $O(1)$ computation

Common Methods

- Division: h(key) = key % table_size
- Multiplication: h(key) = floor(table_size * (key * A mod 1)) ( A = (sqrt(5) - 1)/2 )
- For strings: Sum ASCII values, then modulo

Note:

- Use prime number for table_size (reduces patterns in keys, prime numbers form algebric fields)

# The Collision Problem

What is Collision?

- Two different keys hash to same index
- Pigeonhole principle: With enough keys, collisions inevitable

Example

- Table size = 10
- h(15) = 15 % 10 = 5
- h(25) = 25 % 10 = 5
- Both want index 5!

Collision Resolution Strategies

- Open Addressing: Linear, Quadratic, Double Hashing
- Separate Chaining

# Open Addressing

# Open Addressing: Overview

Main Idea

- All elements stored in the table itself
- When collision: find another empty slot
- Table must be larger than number of elements

Load Factor

$$\alpha = \frac{n}{m}$$

- $n$ = elements stored, $m$ = table size
- Keep $\alpha < 0.7$, performance can degrade rapidly after that

Probe Sequence

- Order of slots checked when collision occurs

# Linear Probing

## Algorithm

- If slot h(key) is occupied: try h(key) + 1, then h(key) + 2, ...
- Wrap around to beginning if needed

## Insert Example (table size = 10)

insert(15) -> h(15) = 5 -> slot 5
insert(25) -> h(25) = 5 -> slot 6 (collision!)
insert(35) -> h(35) = 5 -> slot 7 (collision!)

## Pros/Cons

1. Simple to implement
2. Cache friendly (sequential access)

- Clustering: Occupied slots group together

# The Clustering Problem

What happens?

- Once a cluster forms, it grows
- Larger clusters = higher collision probability
- Performance degrades as load factor increases

Linear Probing Search

- Best case: $O(1)$ - found at hash index
- Worst case: $O(n)$ - entire table is one cluster
- Average: $\approx .\frac{1}{1-\alpha}$

Solution?

- Quadratic probing
- Double hashing

# Quadratic Probing

- Probe sequence: h(key), h(key) + 1^2, h(key) + 2^2, …
- Formula: (h(key) + i^2) % table_size

## Insert Example (table size = 10)

insert(15) -> h(15) = 5 -> slot 5
insert(25) -> h(25) = 5 -> slot 6 (5 + 1^2)
insert(35) -> h(35) = 5 -> slot 9 (5 + 2^2)
insert(45) -> h(45) = 5 -> slot 4 (5 + 3^2 = 14 -> 4)

## Pros

- Reduces primary clustering

## Cons

- Secondary clustering: Keys with same hash follow same sequence
- May not visit all slots even if table not full

# Double Hashing

Algorithm

- Use TWO hash functions
- Probe: (h1(key) + i * h2(key)) % table_size
- h2(key) must never be 0 (and ideally coprime with table_size)

Pros

- No clustering (different keys have different probe sequences)
- Sorta guarantees better distribution of probes ( Check Selection of h2(k) )

Cons

- More computation (two hash functions)

# Separate Chaining

# Separate Chaining

## Main Idea

- Each table slot is a linked list (or vector)
- Collisions go into same list

## Example (table size = 10)

Index 0: NULL
Index 1: 11 -> 21 -> 31 -> NULL
Index 2: NULL
Index 5: 5 -> 15 -> 25 -> NULL
...

## Key Insight

- Load factor can exceed 1.0!
- Performance depends on average list length

# Chaining: Operations

Insert: $O(1)$ average

- Compute hash, go to index
- Append to list (or insert at front)

Search: $O(1 + \alpha)$ average

- Compute hash, go to index
- Traverse list
- With small load factor: lists are short!

Delete: $O(1 + \alpha)$ average

- Search for element, remove from list
- Easier than open addressing!

# Comparison

# When to Use What?

Open Addressing when:

- Memory is constrained
- Know max number of elements upfront
- Want cache-friendly access
- Rarely delete elements

Separate Chaining when:

- Don’t know size in advance
- Frequent deletions
- Load factor might exceed 1
- Want simpler implementation

# Rehashing

# Rehashing

When to Rehash?

- Load factor exceeds threshold (e.g., 0.7 for open addressing, 1.0 for chaining)
- Performance starts degrading

How?

1. Create new table (typically 2x size)
2. Rehash ALL existing elements
3. Free old table

Cost

- $O(n)$ one-time cost
- Amortized $O(1)$ per operation (like dynamic arrays)

# Practice Problems

# Problem 1: Group Anagrams

Problem Statement Group strings that are anagrams of each other.

# Group Anagrams: Intuition

Key Insight

- Anagrams have same sorted characters
- “eat”, “tea”, “ate” → all sort to “aet”
- Use sorted string as hash key!

Alternative

- Character count as key: “a -> 1 e -> 1 t -> 1” for “eat”
- Avoids sorting overhead

Data Structure

- Hash map: key → list of strings

# Group Anagrams: Solution

Algorithm

1. For each word: sort characters to get key
2. If key exists in map: append word to list
3. Else: create new list with this word
4. Return all lists

```
for each string s:
    key = sort(s)
    map[key].push(s)
return map.values()
```

Time: $O(n * k \log k)$ where k = max word length Space: $O(n * k)$

# Problem 2: Two Sum

Problem Statement

Find two numbers that add to target.

# Two Sum: Intuition

Hashing Idea

- For each number `x`, we need `target - x`
- Check if complement exists in hash set
- $O(1)$ lookup per element!

Single Pass

- Check complement, then insert current

# Two Sum: Solution

Algorithm

1. Create empty hash map: value $\rightarrow$ index
2. For each element:
   - Calculate complement = target - current
   - If complement in map: return both indices
   - Else: store current $\rightarrow$ index in map

```
for i = 0 to n-1:
    complement = target - arr[i]
    if map.has(complement):
        return [map[complement], i]
    map[arr[i]] = i
```

Time: $O(n)$ | Space: $O(n)$

# Problem 3: Subarray Sum Equals K

Problem Statement

Given an integer array nums and an integer k, return the total number of subarrays whose sum equals k.

Example

- Input: nums = [3, 4, 7, 2, -3, 1, 4, 2], k = 7
- Output: 4
- Subarrays: [3,4], [7], [7,2,-3,1], [1,4,2]

# Subarray Sum Equals K: Intuition

Prefix Sum Idea

- Let `prefix[i]` = sum of `nums[0..i]`
- Subarray `[l+1 .. r]` has sum = `prefix[r] - prefix[l]`
- We want: `prefix[r] - prefix[l] = k`
- Rearrange: `prefix[l] = prefix[r] - k`

Hashing the DP state

- As we compute `prefix[r]`, look up how many times `prefix[r] - k` has appeared before
- Store prefix sum frequencies in a hash map
- `map[0] = 1` (empty prefix, sum = 0 seen once)

Single pass, no need to store the full prefix array!

# Problem 4: Count Nice Subarrays

Problem Statement Given an array of integers nums and an integer k, return the number of contiguous subarrays that contain exactly k odd numbers.

Example

- Input: nums = [2, 2, 2, 1, 2, 2, 1, 2, 2, 2], k = 2
- Output: 16

Constraint: $1 \leq n \leq 50000$, values up to $10^5$

# Count Nice Subarrays: Intuition

Transform the Problem

- Replace each number: odd → 1, even → 0
- Now: “subarrays with exactly k odd numbers” = “subarrays with sum = k”
- This is exactly Subarray Sum Equals K!

Prefix Odd Count + Hashing

- prefix[i] = number of odd elements in nums[0..i]
- We want prefix[r] - prefix[l] = k
- i.e., look up prefix[r] - k in our hash map

# Problem 5: Longest Consecutive Sequence

Problem Statement

Given an unsorted array of integers, find the length of the longest sequence of consecutive integers. Must run in $O(n)$.

Example

- Input: nums = [100, 4, 200, 1, 3, 2]
- Output: 4
- Sequence: 1, 2, 3, 4

Constraint: $O(n)$

# Longest Consecutive Sequence: Intuition

- We need a way to check “is this number’s neighbour present?” in $O(1)$
- Hash set gives exactly that!

Key Idea

- Insert all numbers into a hash set
- For each number `n`, only start counting if `n - 1` is not in the set
- This ensures we only begin at sequence starts.
- From there, keep checking `n+1`, `n+2`, … until the chain breaks

```c
int longestConsecutive(int *nums, int numsSize) {
    // build hash set: presence only, value = 1
    // ... insert all nums into set ...
    int max_len = 0;
    for (int i = 0; i < numsSize; i++) {
        if (has(nums[i] - 1)) continue; // not a sequence start

        int cur = nums[i], len = 1;
        while (has(cur + 1)) { cur++; len++; }
        if (len > max_len) max_len = len;
    }
    return max_len;
}
```

- Every number is visited at most twice (once on insert, once during a streak)
- The contains(n - 1) guard is what keeps it $O(n)$ overall

# Summary

# Summary: Hash Table Operations

| Operation | Average | Worst | Notes |
|---|---|---|---|
| Search | $O(1)$ | $O(n)$ | With good hash function |
| Insert | $O(1)$ | $O(n)$ | Assuming resizing |
| Delete | $O(1)$ | $O(n)$ | Chaining easier |

# Takeaway Problems!

**LRU Cache**

Design cache that evicts least recently used item when full. Hash map + doubly linked list for $O(1)$ operations.

**Circle Killing**

Hard counting problem with coordinate compression.

**Sum of Three Values**

Find three elements that sum to target. Hash map + sorting.

# More you know ..

**Rabin-Karp String Matching**

# Questions?

Thank you!