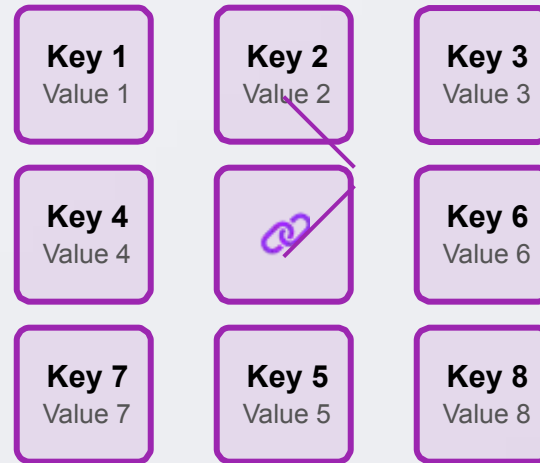


# Data Structures: A Deep Dive into HashMaps

Explanations, Examples, Pros, Cons, Use Cases, & Implementations



Keys



Hashing



Buckets



Collision Resolution

Presented by Dr. Bahadir Aydin

# The Problem with Arrays

## Limitations in Large Datasets

### 🔍 Sequential Search Requirement

In unsorted arrays, finding an element requires checking **every single element** until a match is found.

### ⌚ Linear Time Complexity

This leads to a  **$O(n)$**  time complexity, where 'n' is the number of elements.

### 🎓 Real-World Example

In a large unsorted list of student records, finding a student by name requires checking each record.

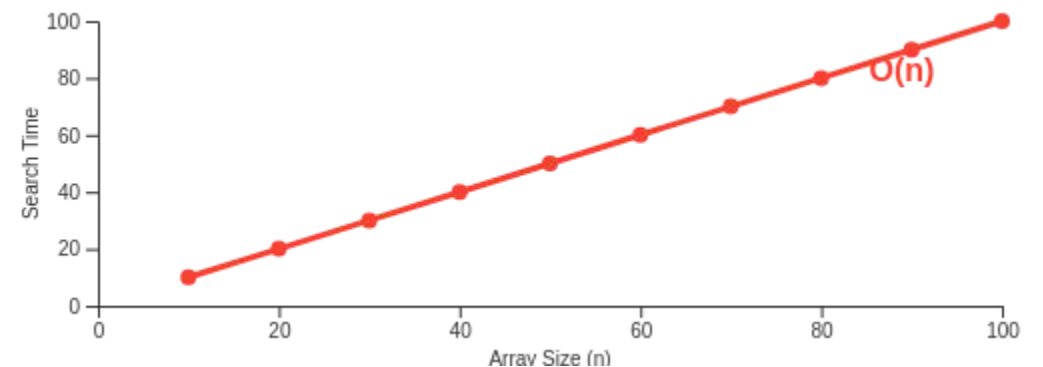
### ⚠️ Growing Bottleneck

As datasets grow, linear search becomes **impractical** for applications requiring fast lookups.

## Visualizing Linear Search



Time Complexity Growth



# What is a HashMap?

## Key-Value Data Structure



### Definition

A HashMap (hash table, dictionary) stores data as **key-value pairs**.



### Mechanism

Uses a **hash function** to convert keys into indices.



### Performance

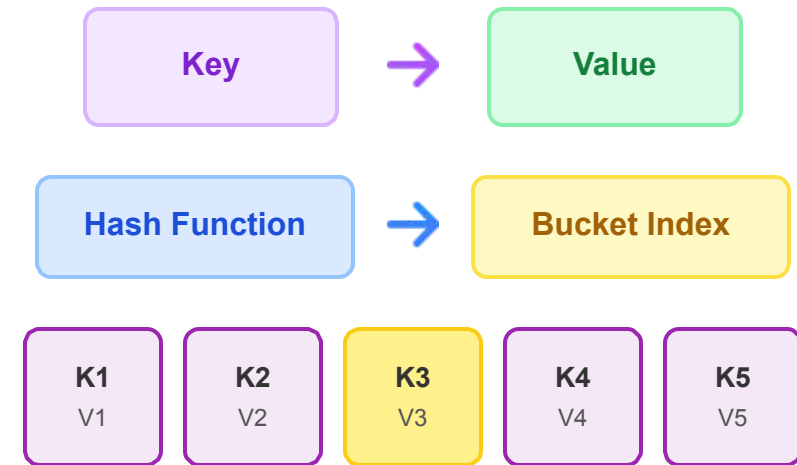
Near-constant time  **$O(1)$**  for operations.



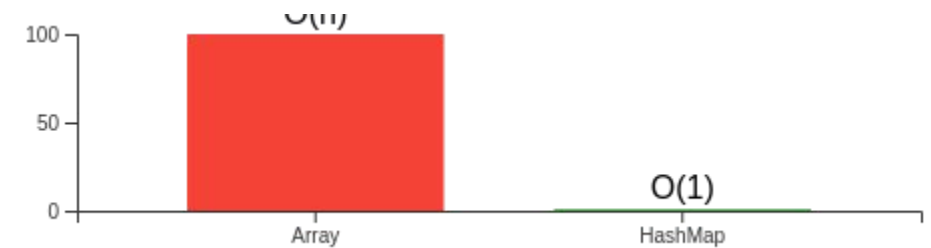
### Value

Ideal for applications requiring **rapid access**.

## How Hashmaps Work



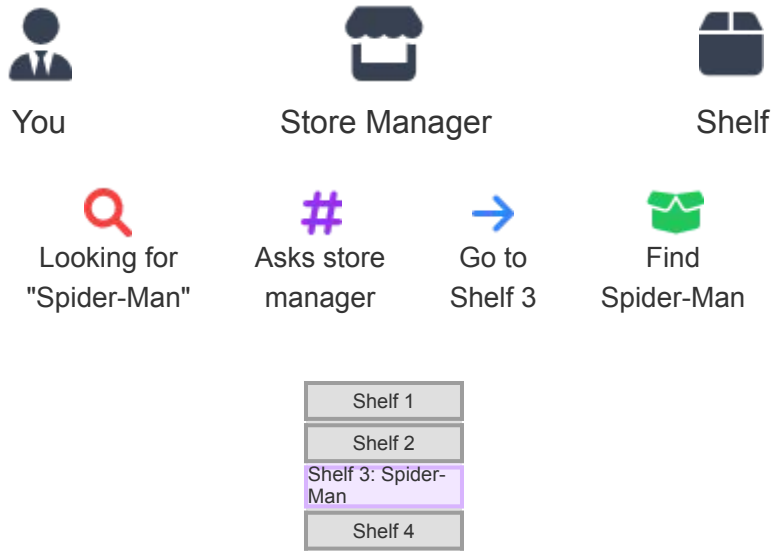
## Time Complexity Comparison



# An Intuitive Analogy

## The Toy Store Analogy

Imagine a giant toy store with hundreds of shelves. If you're looking for a specific superhero bobblehead, you wouldn't want to search every shelf.

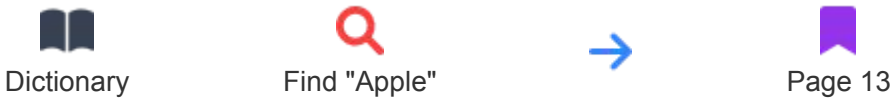


**Store manager** acts like a **hash function**, directing you to the exact shelf

 This direct guidance is the core advantage of a HashMap

## The Dictionary Analogy

In a real-world dictionary, you use a **word** (key) to instantly find its **meaning** (value).






<b>Apple</b>	A fruit that is typically red, green, or yellow
<b>Banana</b>	A long yellow fruit
<b>Orange</b>	A citrus fruit that is typically orange

You don't need to scan every entry — the **word** (key) directly leads you to the **meaning** (value)

# Core Components of HashMaps






## Keys

-  Unique identifiers for data
-  Like "words" in a dictionary
-  Directly link to specific values






## Hash Function

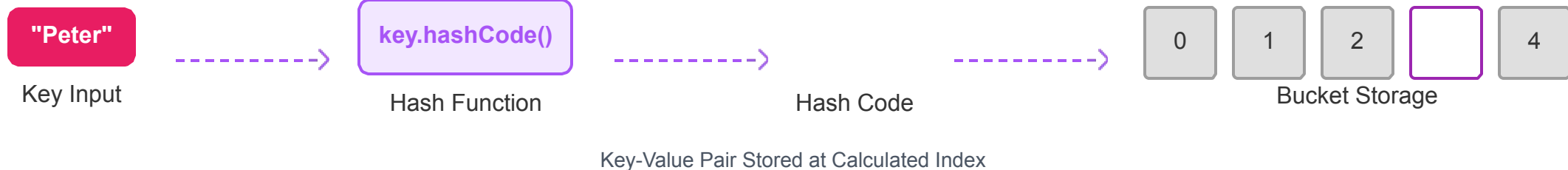
-  "Magical formula" for keys
-  Transforms key to hash code
-  Deterministic: same key = same code



## Buckets

-  Storage locations (array)
-  Hold key-value pairs
-  Distribute keys uniformly

## How These Components Work Together



# The Hashing Process



## 1. Key Input

A key is provided for insertion or retrieval.

"Peter"



## 2. Hash Code Generation

The hash function transforms the key into a hash code.

`key.hashCode()`



## 3. Index Calculation

The hash code is converted to an array index using modulo.

$152 \% 16 = 8$



## 4. Data Storage/Retrieval

The key-value pair is stored at or retrieved from the calculated index.

Bucket 8

## Example Walkthrough

**Key:** "Peter"

We want to store or retrieve the student record for "Peter".

**Hash Code:** 152

The hash function converts "Peter" into the hash code 152.

### Index Calculation

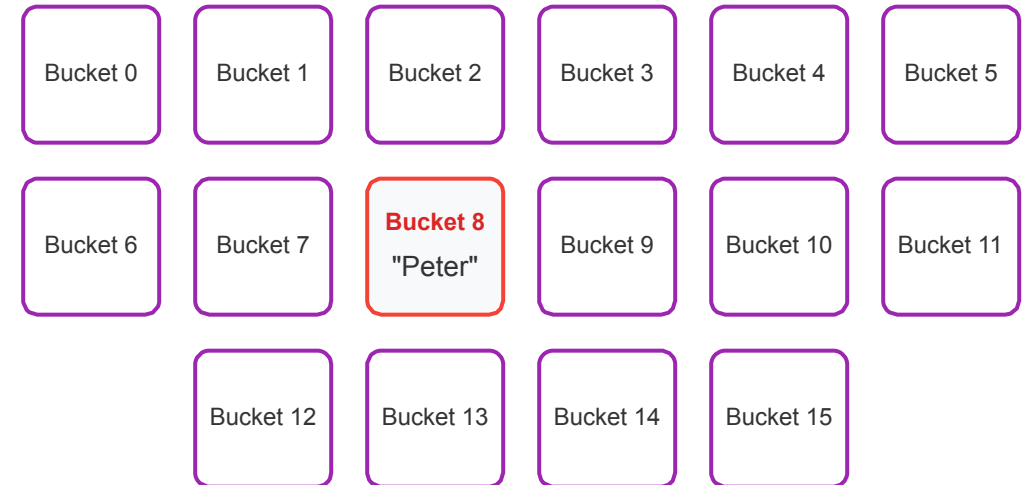
Using modulo operation with array size 16:

$152 \% 16 = 8$

### Direct Access

The system can directly jump to Bucket 8, bypassing the need to search through all elements.

HashMap Buckets (Array of slots)



**$O(1)$  Average Time Complexity**

Direct access to data without sequential search

# The hashCode() and equals() Contract

## The Contract Rules

### Consistency

If **two objects are equal** (`equals()`)

# Properties of a Good Hash Function



## Deterministic

For a given input key, the hash function must **consistently produce the same output** every time it is called.

- ✓ Ensures reliable data retrieval after storage



## Uniform Distribution

The hash function should **distribute hash values evenly** across the entire output range.

- ✓ Minimizes collisions and maintains efficient performance



## Efficiency

The hash function must be **computationally fast** to execute.

- ✓ Since hashing occurs during every insertion, deletion, and lookup operation

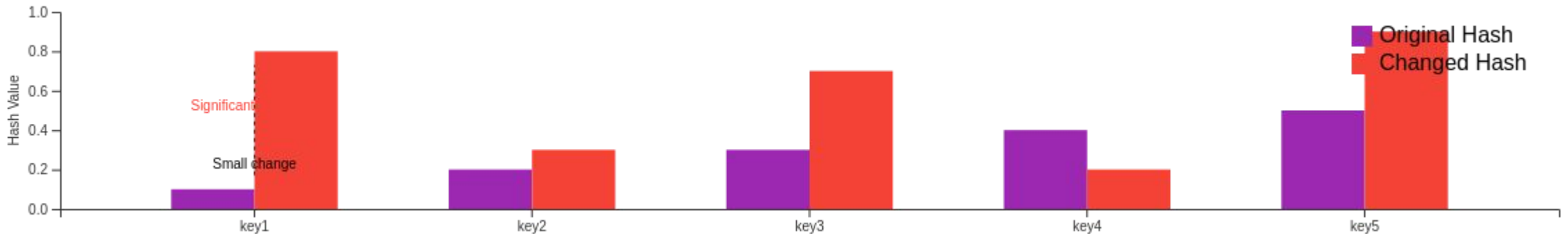


## Avalanche Effect

A small change in the input key (even a single bit) should result in a **significantly different hash value**.

- ✓ Prevents clustering and ensures good distribution

## Avalanche Effect Visualization





# Common Hashing Techniques

Fundamental techniques that illustrate the basic principles of hashing

## ÷ Division Method

$$h(K) = K \bmod M$$

One of the simplest methods, where the hash value is calculated by taking the remainder of the key divided by the size of the hash table.

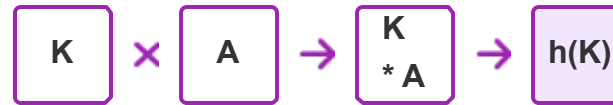


⚠ Can lead to poor distribution if  $M$  is not chosen carefully (e.g., prime numbers are preferred)

## × Multiplication Method

$$h(K) = \text{floor}[M * (K * A \bmod 1)]$$

Involves multiplying the key by a constant  $A$  ( $0 < A < 1$ ), taking the fractional part, then multiplying by table size  $M$ .

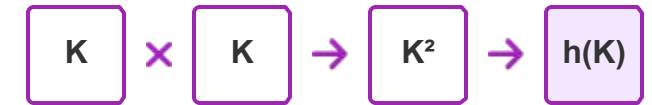


✅ Less sensitive to the choice of table size than division method

## ■ Mid-Square Method

$$h(K) = \text{middle digits of } K^2$$

The key is squared, then a fixed number of middle digits from the result are extracted to form the hash value.



⚠ Can be computationally more intensive than other methods

Modern hash functions are typically more complex and optimized for specific use cases

# Collision Resolution: Chaining

## What is Chaining?



A **collision** occurs when multiple keys hash to the same bucket.

Chaining stores **multiple key-value pairs** at the same bucket location.



### Implementation Methods

- Traditional: **Linked Lists**
- Modern: **Balanced Trees** (Java 8+ uses Red-Black Trees when a bucket has 8+ elements)



### Time Complexity Impact

#### Without Chaining

$O(n)$

Linked List Search



#### With Chaining

$O(\log n)$

Tree-based Search

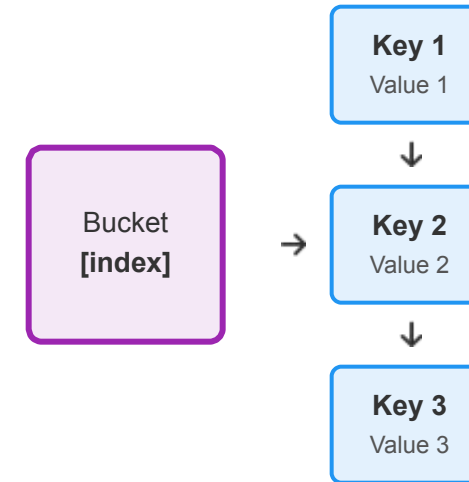
*\*Average time complexity remains  $O(1)$  for both approaches*



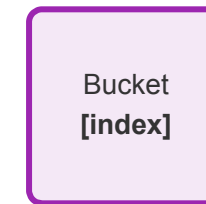
Java's HashMap uses linked lists for collision resolution, with automatic conversion to Red-Black Trees for large buckets.

## Chaining Visualization

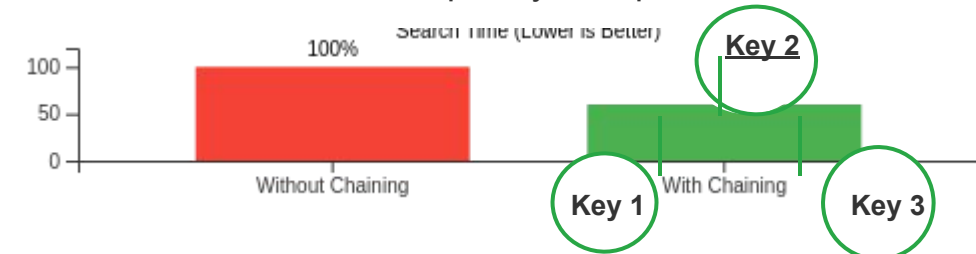
### Traditional Chaining (Linked List)



### Tree-based Chaining (Java 8+)



### Time Complexity Comparison



# Collision Resolution: Open Addressing

When two keys hash to the same index, **open addressing** techniques search for the next available slot in the hash table.

## → Linear Probing

Sequential search for next empty slot.

```
</> probe_seq = i + 1, 2, 3...
```



## X<sup>1</sup> Quadratic Probing

Uses quadratic function for spacing.

```
</> probe_seq = i + c1*i + c2*i2
```



## X<sup>1</sup> Double Hashing

Uses a second hash function for spacing.

```
</> probe_seq = h1(key) + i + h2(key)
```



✓ No additional data structures needed

⚠ Clusters can form, causing performance issues

⚖ Balance between simplicity and performance

# Time and Space Complexity

## HashMap Performance Analysis

### ⌚ Time Complexity

HashMap operations exhibit different performance characteristics based on implementation:

Operations	Average Case	Worst Case (Chaining)	Worst Case (Tree-based)
Insertion	$O(1)$	$O(n)$	$O(\log n)$
Deletion	$O(1)$	$O(n)$	$O(\log n)$
Lookup	$O(1)$	$O(n)$	$O(\log n)$

### 🗄️ Space Complexity

$O(n)$  - where 'n' is the number of key-value pairs

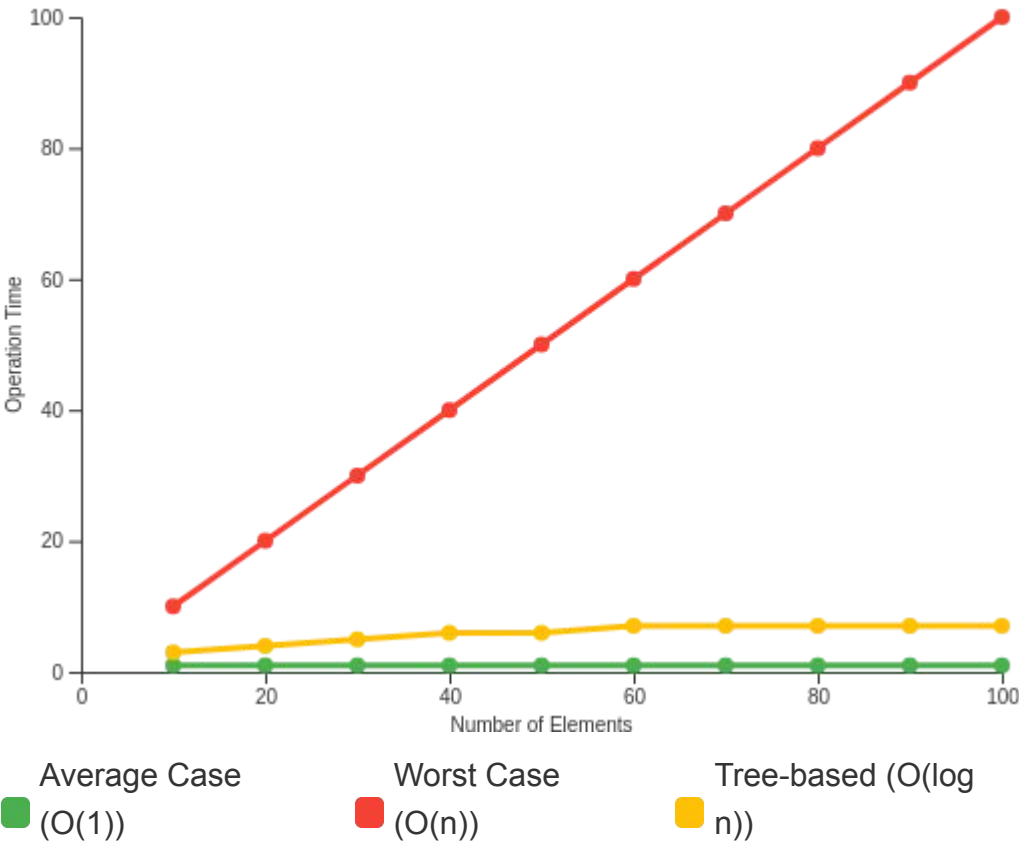
Each pair occupies constant space, and total space grows linearly  
HashMaps often allocate extra space to maintain average  $O(1)$  time complexity

### 📖 Implementation Notes

Java 8+ uses tree-based chaining (Red-Black Trees) when a bucket contains 8+ elements

Python uses SipHash for hash function, with optimized algorithms

## Time Complexity Comparison



### Key Factors Affecting Performance:

- Load Factor: Default 0.75 in Java
- Capacity: Default initial capacity of 16
- Rehashing: Occurs when entries > capacity × load factor

# Load Factor and Rehashing

## Understanding Load Factor

### ⚡ What is Load Factor?

Ratio of **stored entries** to **total buckets**.

Common default: **0.75** (75%)



### Capacity

Number of buckets in the HashMap.

Default initial capacity: **16**



### Rehashing Trigger

Occurs when:

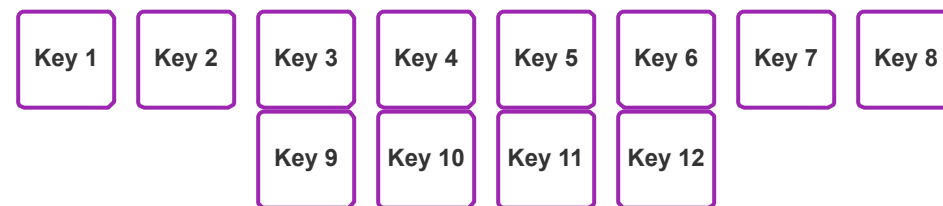
**Entries > Capacity × Load Factor**

### Why This Matters:

- High load increases **collisions**
- Too few buckets causes **inefficient storage**
- Rehashing maintains **O(1)** average time

## Rehashing Process

Current Array (Load Factor = 0.75)

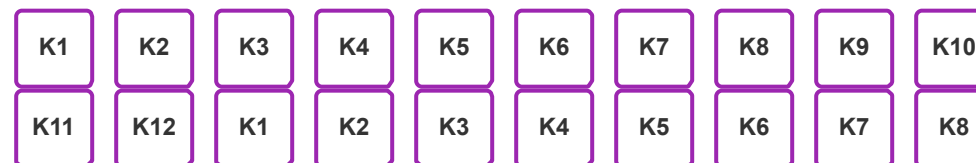


$$12/16 = 0.75$$



Rehashing

New, Larger Array



$$12/32 = 0.375$$

Benefits: Fewer collisions, O(1) performance maintained

# HashMap Implementation in Java

## Java HashMap Example

```
// Create a HashMap
Map<String, String> capitalCities = new HashMap<>();

// Add elements (put) capitalCities.put("England",
"London"); capitalCities.put("India", "New Delhi");
capitalCities.put("Norway", "Oslo");

// Access elements (get)
System.out.println("Capital of England: " + capitalCities.get("England"))

// Update an element
capitalCities.put("Norway", "Bergen"); // Overwrites "Oslo"

// Remove an element
capitalCities.remove("India");

// Iterate over key-value pairs
for (Map.Entry<String, String> entry : capitalCities.entrySet()) {
    System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
}
}
```

## Key Operations



### Adding Elements

Use `put(key, value)`

# HashMap Implementation in Python



## Adding Elements

Uses `dict[key] = value` to add key-value pairs. If the key exists, it updates the value; otherwise, it inserts a new pair.



## Accessing Elements

Uses `dict[key]` to retrieve values by key. This will raise a `KeyError` if the key doesn't exist. Use `dict.get(key)` for safe access that returns `None` by default if the key is not found.



## Removing Elements

Uses the `del` keyword (`del dict[key]`) to remove the key-value pair associated with the specified key.



## Key Existence

Uses the `in` keyword (`key in dict`) to check if a key exists in the map. It returns `True` if the key is present and `False` otherwise.



## Iteration

Uses a standard for loop to iterate over keys by default. Use `.items()` to iterate over (key, value) pairs and `.values()` to iterate over values.

```
# Create a dictionary (Python's HashMap)
capital_cities = {}

# Add elements
capital_cities["England"] = "London"
capital_cities["India"] = "New Delhi"
capital_cities["Norway"] = "Oslo"
print(f"Dictionary after adding elements: {capital_cities}")

# Access elements
print(f"Capital of England: {capital_cities['England']}")

# Update an element
capital_cities["Norway"] = "Bergen" # Overwrites "Oslo"
print(f"Dictionary after updating Norway: {capital_cities}")

# Remove an element
del capital_cities["India"]
print(f"Dictionary after removing India: {capital_cities}")

# Iterate over keys
print("Keys:")
for key in capital_cities.keys():
    print(key)

# Iterate over values
print("Values:")
for value in capital_cities.values():
    print(value)

# Iterate over key-value pairs
print("Key-Value Pairs:")
for key, value in capital_cities.items():
    print(f"{key}: {value}")
```

# HashMap Implementation in C++

## C++ HashMap Overview

## C++ Code Example

### STL Implementation

C++ provides **std::unordered\_map** in the Standard Template Library for hashmap functionality.

### + Adding Elements

Uses **operator[]** to add key-value pairs. If the key exists, it updates the value; otherwise, it inserts a new pair.

### 🔍 Accessing Elements

Uses **operator[]** to retrieve values by key. If the key doesn't exist, it inserts a new default pair.

### 🗑 Removing Elements

Uses **erase(key)** to remove the key-value pair with the specified key.

### ✅ Key Existence

Uses **count(key)** to check if a key exists in the map. Returns 1 if the key exists, 0 otherwise.

### ↔ Iteration

Uses range-based for loop with **auto&** to iterate over key-value pairs.

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

int main() {
    // Create an unordered_map
    std::unordered_map<std::string, std::string> capitalCities;
    // Add elements using operator[]
    capitalCities["England"] = "London";
    capitalCities["India"] = "New Delhi";
    capitalCities["Norway"] = "Oslo";
    std::cout << "Unordered_map after adding elements:" << std::endl;
    for (const auto& pair : capitalCities) {
        std::cout << "      " << pair.first << ": " << pair.second <<
            std::endl;
    }

    // Access elements
    std::cout << "Capital of England: " << capitalCities["England"] << std::endl;

    // Update an element
    capitalCities["Norway"] = "Bergen"; // Overwrites "Oslo"
    std::cout << "Unordered_map after updating Norway:" << std::endl;
    for (const auto& pair : capitalCities) {
        std::cout << "      " << pair.first << ": " << pair.second <<
            std::endl;
    }

    // Remove an element
    capitalCities.erase("India");
    std::cout << "Unordered_map after removing India:" << std::endl;
    for (const auto& pair : capitalCities) {
        std::cout << "      " << pair.first << ": " << pair.second << std::endl;
    }

    // Check if a key exists
    if (capitalCities.count("England")) {
        std::cout << "England is in the map." << std::endl;
    }
    return 0;
}
```



Key → Hash → Bucket → Value



# Solving Problems with HashMaps

## Frequency Counting

HashMaps excel at counting occurrences of elements in a collection.

```
HashMap<String, Integer> frequency = new HashMap<>();  
// Counting logic...
```

## Group Anagrams

Group words that are anagrams of each other from a list.

```
HashMap<String, List<String>> anagrams = new HashMap<>();  
// Grouping logic...
```

## Two-Sum Problem

Find two numbers that add up to a target value efficiently.

```
HashMap<Integer, Integer> map = new HashMap<>();  
// Solution logic...
```

## LRU Cache

Implement a cache that discards the least recently used items first.

```
class LRUCache extends LinkedHashMap<K, V> {  
    // Methods...  
}
```

## Common Algorithm Patterns

### Direct Access Pattern

Using keys for direct access to values.

### Key Transformation

Transforming keys to solve different problems.

### Multi-map Structure

Using multiple hashmaps for complex queries.

### Cache + Verify

Caching results and verifying.

# Real-World Applications

HashMaps are ubiquitous in modern software development due to their efficiency and versatility. Their ability to provide near-constant time access makes them invaluable in various systems:



## Database Indexing

HashMaps accelerate data retrieval in databases by mapping unique keys (e.g., primary keys) to the physical location of data records, allowing for rapid lookup of information.



## Caching

HashMaps are fundamental to caching mechanisms. Results of computationally expensive operations or frequently accessed data can be stored using a key, allowing for quick retrieval without re-computation.



## Compilers & Interpreters

In language processing, compilers and interpreters use HashMaps to implement symbol tables, storing information about variables, functions, and classes for quick lookups during compilation or execution.



## Routers & DNS

Network routers employ hash tables to efficiently route internet traffic by mapping IP addresses to network interfaces. DNS resolvers use HashMaps to quickly translate domain names into their corresponding numerical IP addresses.



## Password Storage

For security, systems store hash values of passwords instead of the actual passwords. When a user logs in, their entered password is hashed and compared against the stored hash value, preventing direct exposure even if the database is compromised.



## Data Integrity

Hash functions are used to generate checksums or hash values for files or messages. By comparing hash values before and after transmission or storage, any unauthorized modifications or data corruption can be detected.

# Pros and Cons of HashMaps

## + Advantages



### Near-Constant Time Operations

$O(1)$  average time complexity for insertion, deletion, and retrieval



### Efficient for Large Datasets

Maintains performance even with millions of elements



### Simple Implementation

Straightforward to use with built-in implementations in most languages



### Fast Data Access

Direct access to data without sequential searching



### Robust Collision Handling

Modern implementations use chaining or tree-based approaches to maintain  $O(1)$  performance

## - Limitations



### Worst-Case Performance

$O(n)$  time complexity when many keys hash to the same bucket



### Degraded Performance

Can revert to linear search in specific scenarios



### Space Overhead

Requires additional space for buckets and hash functions



### Unsorted Data

Not suitable for operations requiring sorted data



### Complex Collision Resolution

Requires additional logic for handling collisions

**Key Takeaway:** HashMaps provide exceptional performance for many use cases, but careful consideration is needed for scenarios with strict performance requirements or limited memory resources.

# Summary and Best Practices

Key takeaways and guidelines for effective use of hashmaps



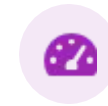
## When to Use

- ✓ When  **$O(1)$**  time complexity is required
- ✓ For large datasets with frequent lookups
- ✓ In database indexing
- ✓ For caching mechanisms



## Implementation

- ⚠ Choose appropriate initial capacity
- ⚠ Understand collision resolution
- ⚠ Implement hash functions properly
- ⚠ Follow hashCode/equals() contract



## Optimization

- ⚡ Optimize hash function
- ⚡ Manage load factor
- ⚡ Use tree-based chaining
- ⚡ Profile and benchmark

💡 Prefer built-in implementations

💡 Choose right hash function

💡 Monitor performance

*"HashMaps provide near-constant time complexity for fundamental operations, making them an essential data structure for efficient information retrieval."*