Hashing

Presented by Yasin Ceran

March 14, 2024

Introduction to HashSet and HashMap

Hashset and HashMap

- Focusing on the utility of ‘HashSet‘ and ‘HashMap‘.
- Essential for coding interviews, especially for problems involving uniqueness, counting, and frequency.
- Sets and Maps differ in that Sets do not associate keys to values, whereas Maps maintain key-value pairs.

Motivation: Comparing HashMaps, TreeMaps, and Arrays

- Understanding trade-offs among data structures is crucial.
- HashMaps offer significant efficiency for certain operations compared to TreeMaps and sorted/unsorted arrays.

Time Complexity Comparison

Operation	TreeMap	HashMap	Array
Insert	$O(\log n)$	$O(1)$	$O(n)$
Remove	$O(\log n)$	$O(1)$	$O(n)$
Search	$O(\log n)$	$O(1)$	$O(\log n)$ / $O(n)$
Inorder Traversal	$O(n)$	-	-

Table: Operation Time Complexity for TreeMap, HashMap, and Array

- The listed time complexity for HashMaps is average-case.
- In interviews, it's common to assume constant time for HashMap operations.

Tree Maps vs. Hash Maps

- **Ordering:** Hash Maps do not maintain order, unlike Tree Maps or arrays.
- To iterate through Hash Map keys in a specific order, sorting is necessary, leading to $O(n \log n)$ complexity.
- **Use Case for Hash Maps:**
  - Ideal for counting frequencies or occurrences due to their key-value storage model.
  - Example: Counting names in a phonebook can be efficiently done by mapping each name to its frequency.
- While Hash Maps offer speed and efficiency for insert, remove, and search operations, the lack of inherent order is a trade-off.

Counting Frequencies with HashMap

- Utilizing a **HashMap** efficiently counts the occurrences of elements in an array.
- **Example Array:** ["*alice*", "*brad*", "*collin*", "*brad*", "*dylan*", "*kim*"]
- In a HashMap:

Key	**Value**
Alice	1
Brad	2
Collin	1
Dylan	1
Kim	1

- **Algorithm:**
  - If a name is encountered for the first time, add it to the HashMap with a frequency of 1.
  - If a name already exists, increment its frequency by 1.

Efficiency

Efficiency

- TreeMap: Insertion operation costs $O(\log n)$, leading to $O(n \log n)$ for all insertions.
- HashMap: Overall operation costs $O(n)$, showcasing its efficiency over TreeMap for this task.

```
names = ["alice", "brad", "collin", "brad", "dylan", "kim"]

countMap = {}
for name in names:
    # If countMap does not contain name
    if name not in countMap:
        countMap[name] = 1
    else:
        countMap[name] += 1
```

Hash Implementation: Understanding the Basics

- **Under the Hood:** Hash maps utilize arrays for storage. An empty hash map does not equate to a zero-sized array.
- **Example:** Inserting key-value pairs into a hash map.
  - `hashmap["Alice"] = "NYC"`
  - `hashmap["Brad"] = "Chicago"`
  - `hashmap["Collin"] = "Seattle"`
- **Hashing and Hash Function:**
  - A *hash function* processes a key (e.g., a string) to produce a consistent integer output.
  - This integer determines where to store the key-value pair in the array.
  - **Property:** The same input always yields the same output, ensuring consistency in data retrieval.
- The effectiveness of a hash map hinges on the *hash function's* ability to:
  - Distribute keys evenly across the array.
  - Minimize collisions (instances where different keys hash to the same index).

Insertion and Hashing Process

- **Hash Function Example:** Converting "Alice" to an array index.
- Characters are converted to ASCII codes and summed. For "Alice", assume the sum is 25.
- **Determining Position:** Sum mod Array Size. With an array size of 2, $25 \mod 2 = 1$.
- This method is known as pre-hashing, effectively distributing keys across the array space.

Handling Collisions

- Collisions occur when two keys hash to the same index.
- **Inevitable** due to the finite size of arrays and the diverse range of keys.
- Strategies like chaining or open addressing are employed to resolve collisions.
- **Example:** If both "Alice" and another key hash to 1, a collision management strategy is needed.

Array Resizing and Rehashing

- **Resizing:** When the array becomes half full, its size is doubled to minimize collisions.
- This preemptive resizing occurs before new insertions once the threshold is reached.
- **Rehashing:** All keys are rehashed according to the new array size.
  - Necessary to maintain $O(1)$ time complexity for lookups.
  - Ensures keys remain in correct positions relative to the new size.
- **Example:** After inserting "Alice", doubling array size from 2 to 4 requires rehashing "Alice" to determine its new position.

Visualization of Insertions

- Initial array size: 2. After inserting "Alice": size doubles to 4.
- "Alice" hashes to index 1. "Brad" hashes to 27, and 27 $\mod 4 = 3$, so "Brad" is placed at index 3.
- Upon reaching half capacity again, array size doubles to 8, necessitating rehashing.
- **Importance:** Efficient insertions and searches rely on dynamic resizing and proper collision handling.

Understanding Collisions in Hash Maps

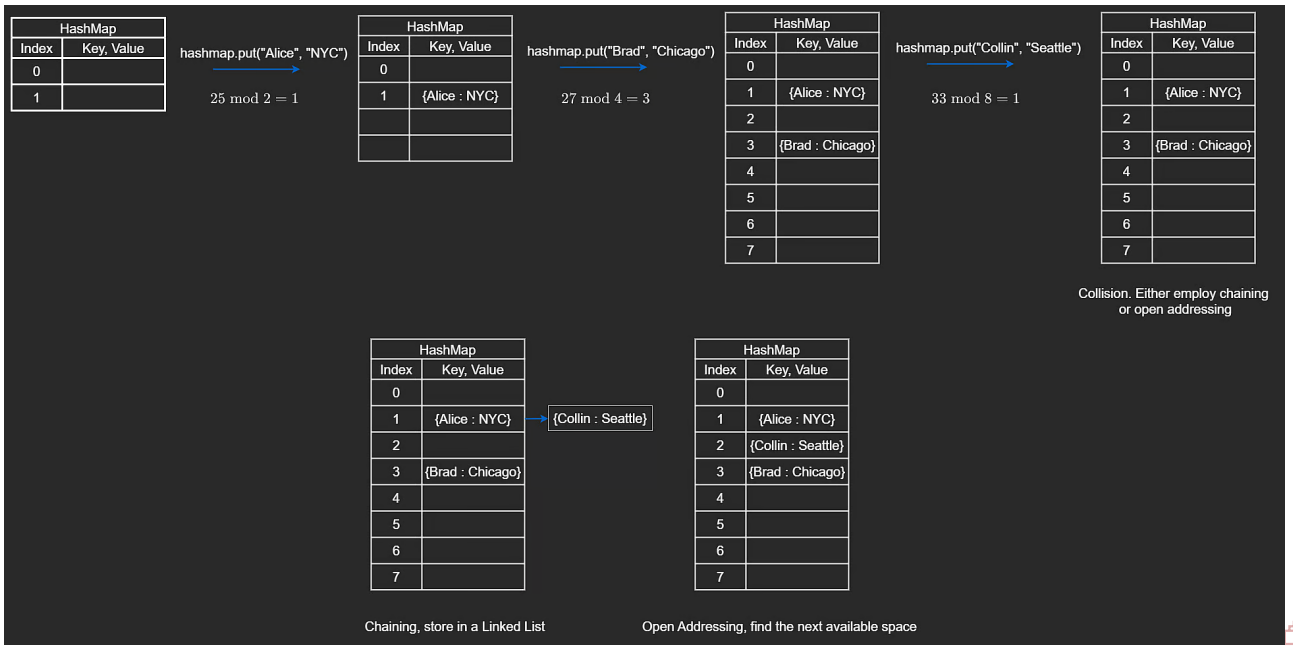
- Collisions occur when two keys hash to the same index.
- Example: "Collin" hashes to 33, and $33 \mod 8 = 1$, causing a collision with "Alice" at index 1.
- Resolving collisions is crucial to maintain the integrity and performance of a hash map.

Chaining to Resolve Collisions

- **Chaining:** Links key-value pairs at the same index using a linked list.
- Allows multiple entries to occupy the same index without loss of data.
- **Example:** "Alice" and "Collin" can be stored in a linked list at index 1.
- **Time Complexity:** Becomes $O(n)$ in the worst case when searching, inserting, or deleting.
- Chaining is effective for handling multiple collisions, ensuring data is not overwritten.

Open Addressing for Collision Resolution

- **Open Addressing:** Searches for the next available slot for inserting the collided key-value pair.
- Prevents storing more than one entry at each index, optimizing memory usage.
- More efficient than chaining when the number of collisions is low.
- **Limitation:** The total number of entries is capped by the array size, possibly necessitating array resizing and rehashing.
- Open addressing ensures a compact storage but requires careful management to avoid performance degradation.



Initialization

We will store a list of Pairs in our array, for which we declare a class.

```
class Pair:
    def __init__(self, key, val):
        self.key = key
        self.val = val
```

We initialize a size, capacity and the map itself in our constructor. Here, size refers to the size of the hash map and capacity refers to size of the array under the hood.

```
class HashMap:
    def __init__(self):
        self.size = 0
        self.capacity = 2
        self.map = [None, None]
```

Hash function

The hash function below iterates through each of the characters in a given key, sums up their ASCII code and finds the position of the key in the array.

```
def hash(self, key):
    index = 0
    for c in key:
        index += ord(c)
    return index % self.capacity
```

To retrieve the value, we first need to retrieve the position and check if the value exists in that position. If it does, we can return that value. Otherwise, we can perform open addressing and look for it in the next available index.

```
def get(self, key):
    index = self.hash(key)

    while self.map[index] != None:
        if self.map[index].key == key:
            return self.map[index].val
        index += 1
        index = index % self.capacity
    return None
```

Adding to the Map

To add to the map, we first compute the hash of the key and find the position. Once this is calculated, there are three scenarios.

- The index is occupied
- The index is occupied with the same key
- The index is vacant

```
def put(self, key, val):
    index = self.hash(key)

    while True:
        if self.map[index] == None:
            self.map[index] = Pair(key, val)
            self.size += 1
            if self.size >= self.capacity // 2:
                self.rehash()
            return
        elif self.map[index].key == key:
            self.map[index].val = val
            return

        index += 1
        index = index % self.capacity
```

Removing from the Map

To remove, we find the index, remove the key, and set the index to null.

```
def remove(self, key):
    if not self.get(key):
        return

    index = self.hash(key)
    while True:
        if self.map[index].key == key:
            # Removing an element using open-addressing actually causes a bug,
            # because we may create a hole in the list, and our get() may
            # stop searching early when it reaches this hole.
            self.map[index] = None
            self.size -= 1
            return
        index += 1
        index = index % self.capacity
```

Removing from the Map

To remove, we find the index, remove the key, and set the index to null.

```
def remove(self, key):
    if not self.get(key):
        return

    index = self.hash(key)
    while True:
        if self.map[index].key == key:
            # Removing an element using open-addressing actually causes a bug,
            # because we may create a hole in the list, and our get() may
            # stop searching early when it reaches this hole.
            self.map[index] = None
            self.size -= 1
            return
        index += 1
        index = index % self.capacity
```

Rehasing

When we perform re-hashing, we double the capacity, copy our previous map's values into our new map and set the size to be zero.

```
def rehash(self):
    self.capacity = 2 * self.capacity
    newMap = []
    for i in range(self.capacity):
        newMap.append(None)

    oldMap = self.map
    self.map = newMap
    self.size = 0
    for pair in oldMap:
        if pair:
            self.put(pair.key, pair.val)
```

If we wish to print all of the pairs, it would look like the following.

```
def print(self):
    for pair in self.map:
        if pair:
            print(pair.key, pair.val)
```

Summary of Hashing Concepts

- Hashing transforms keys into array indices using hash functions.
- A good hash function distributes keys evenly across the array, minimizing collisions.
- Collisions occur when different keys hash to the same index.

Collision Resolution Techniques

- **Chaining:** Resolves collisions by linking entries at the same index into a linked list. Effective for handling high collision rates but increases search time to $O(n)$ in the worst case.
- **Open Addressing:** Finds the next available slot for collided entries, keeping the array compact. Efficient with fewer collisions but requires rehashing when resizing.

Efficiency of Hash Maps

- Hash maps offer $O(1)$ average time complexity for insert, search, and delete operations, assuming minimal collisions.
- Efficiency depends on:
  - The quality of the hash function.
  - The chosen method to resolve collisions.
  - The load factor and how the hash map handles resizing.
- Hash maps are a versatile tool for various programming challenges, especially those requiring fast access to elements.

Practical Applications of Hashing

- Hashing underpins many data structures, including hash tables, caches, and distributed storage systems.
- Essential for algorithms requiring fast data retrieval, uniqueness checks, and data aggregation.
- Key to implementing efficient associative arrays, sets, and maps in many programming languages.