Hash Tables

Authors: Refat Othman and Diaeddin Rimawi

Lecture 1: Fundamentals of Hash Tables

1. Definition

A **hash table** is a data structure that stores key–value pairs and uses a **hash function** to map keys to indices in an array, enabling fast data retrieval.

2. Purpose of Hash Tables

The main goal of a hash table is to provide **efficient insertion, deletion, and lookup operations**. In an ideal case, these operations run in **O(1)** time.

3. Main Components of a Hash Table

Below are the main components of a hash table with detailed explanations and examples:

- **Array**: The underlying storage where elements are placed at positions determined by the hash function. Each index can store one or more items depending on the collision resolution method.
  - *Example*: Suppose we have an array of size 10 and we apply a hash function to the numeric key 57. If `57 % 10` gives 7, then the pair `(57, someValue)` is stored at index 7 of the array. This means when we later search for 57, we compute its hash again, get index 7, and directly access that position to retrieve the value.
- **Hash Function**: A function that converts a key into an integer index within the bounds of the array. It ensures that the same key always maps to the same index.
  - *Example*: For a numeric key 92 and table size 10, `92 % 10` gives 2, meaning 92 is stored at index 2.
- **Buckets**: Containers at each array index that hold entries when collisions occur. Depending on the design, a bucket can be a linked list, dynamic array, or other structure.
  - *Example*: At index 3, the bucket could contain both `(92, 100)` and `(85, 200)` if they share the same index, for example when using the hash function `key % 7` for table size 7, both 92 and 85 give remainder 1 and then are placed in the same bucket due to collision resolution.
- **Collision Resolution Strategy**: The method used to handle cases when two different keys map to the same index.
  - *Example*: Using *separate chaining*, both items (92, 100) and (85, 200) would be linked together in a list at that index. For instance, with table size 7 and hash function `key % 7`, both 92 and 85

produce remainder 1, so they go into bucket at index 1, forming a linked list: `(92, 100) -> (85, 200)`.
- Using *linear probing*, 92 is placed at index 1 and 85 would be placed in the next available index (index 2 in this example) after detecting the collision.

4. Attributes of a Good Hash Function

A good hash function should meet the following criteria. We illustrate each with concise numeric examples.

- **Minimize collisions**: Different keys should rarely map to the same index.
  - Example (bad choice): With table size M = 10 and h(k) = k % 10, the keys 1001, 1011, 1021 all map to index 1 (three collisions).
  - Example (improved): With M = 11, indices become 1001 % 11 = 0, 1011 % 11 = 10, 1021 % 11 = 9, which spreads the keys across the table.
- **Be fast to compute**: Use simple arithmetic and bitwise operations so hashing does not dominate runtime.
  - Example (integers): Multiplicative hashing h(k) = floor(M * frac(A * k)) with A ≈ 0.618033 uses one multiply and a fractional extraction. For M = 16 and k = 92: A * 92 ≈ 56.86, frac = 0.86, 16 * 0.86 ≈ 13.76, so h = 13.
  - Example (strings): Horner's method runs in O(L) for length L: for key "abc" with base B = 31, compute (((0*31 + 97)*31 + 98)*31 + 99) = 96354, then compress with 96354 % M.
- **Distribute keys uniformly**: Indices should be as evenly spread as possible to keep the load per bucket balanced.
  - Example (poor distribution): h(k) = floor(k/10) % 10 on keys 0…99 puts exactly 10 keys per index 0…9, but if your workload is 0…49 only, indices 5…9 are never used.
  - Example (better): h(k) = k % 10 on 0…49 yields about 5 keys per index 0…9. Using prime M and a good compressor further reduces skew for non-consecutive keys.
- **Be deterministic**: The same key must always hash to the same index for consistent retrieval.
  - Example: With M = 7 and h(k) = k % 7, key 92 always hashes to 1. Repeated inserts/lookups of 92 will always target index 1.

> Implementation tip (compression choice): Avoid M as a power of 2 with simple modulo on structured keys, since patterns in keys (for example many even numbers) can amplify collisions. Prefer a prime M and pair it with either multiplicative hashing (integers) or Horner's method + modulo (strings).

5. Building a Hash Function

Example walk-through: Suppose we have key = 92 and table size M = 7.

1. **Convert the key to an integer**: Since 92 is already numeric, the hash code is 92.
2. **Compress the hash code**: 92 % 7 = 1, which maps it into the valid index range 0…6.
3. **Choose M wisely**: We used M = 7 (a prime number) to help distribute keys more evenly. The result tells us to store the key–value pair at index 1.

6. Types of Hashing

Below are the main types of hashing with full descriptions and numeric examples:

1. **Separate Chaining**: Each table index stores a bucket (often a linked list) of all key–value pairs that hash to that index. Collisions are handled by simply adding the new pair to the bucket.
   - *Example*: M = 5, keys 12 and 22 both give 12 % 5 = 2, so index 2 holds a list: (12, v1) → (22, v2).
2. **Open Addressing**: All elements are stored directly in the array. When a collision occurs, probe other indices to find an empty slot.
   - **Linear Probing**: On collision at index i, check i+1, i+2, etc., wrapping around if needed.
     - *Example*: M = 7, insert 15 → index 1. Insert 22 → index 1 (collision), try 2 (empty), store at index 2.
   - **Quadratic Probing**: Probe at distances of 1^2, 2^2, 3^2, ... from the original index.
     - *Example*: M = 7, insert 15 → index 1. Insert 22 → index 1 (collision), try $1+1^2=2$, then $1+2^2=5$, etc., until empty slot found.
   - **Double Hashing**: Use a second hash function to determine the probe step size.
     - *Example*: M = 7, h1(k) = k % 7, h2(k) = 5 - (k % 5). For key 22: h1 = 1, h2 = 3
3. **Rehashing**: When the load factor becomes high, create a larger table (usually about twice the size, preferably prime) and insert all existing elements using a new hash function.
   - *Example*: Table size 5, keys 1, 6, 11 all at index 1. The **load factor** is computed as N / M, where N is the number of stored keys and M is the table size. For N = 3 and M = 5, load factor = 3/5 = 0.6. If our chosen threshold is, for example, 0.7, we do not rehash yet; if N grew to 4, load factor = 0.8 > 0.7, so we would resize to 11 and rehash: 1 % 11 = 1, 6 % 11 = 6, 11 % 11 = 0.

7. Example with Visualization

Let table size `M = 7` and rehash when the load factor exceeds 0.5 (i.e., when more than 3 positions are filled). Keys: 50, 700, 76, 85, 92, 73, 101.

Hash Function: `index = key % 7`

```
Insert 50 → index 1 (N=1, load factor=1/7≈0.14)
Insert 700 → index 0 (N=2, load factor≈0.29)
Insert 76 → index 6 (N=3, load factor≈0.43)
Insert 85 → index 1 (collision) → Linear probing → index 2 (N=4, load factor≈0.57
> 0.5) → Trigger rehash to M=14, reinsert all keys.
After rehash to the first prime number after M*2 (M=7 → M*2=14 → next prime is
17):
    50 → 50%17=16 (N=1, load factor≈1/17≈0.06)
    700 → 700%17=3 (N=2, load factor≈0.12)
    76 → 76%17=8 (N=3, load factor≈0.18)
    85 → 85%17=0 (N=4, load factor≈0.24)
```

```
Continue inserting:
Insert 92 → 92%17=7 (N=5, load factor≈0.29)
Insert 73 → 73%17=5 (N=6, load factor≈0.35)
Insert 101 → 101%17=16 (collision) → Linear probing → index 0 is occupied → next available index 1 (N=7, load factor≈0.41)
```

Visualization after all insertions (final table size 17):

Index	Value
0	85
1	101
3	700
5	73
6	76
7	92
8	76
16	50

8. Implementing a Hash Table in Python

> Linear probing implementation with a simple `Node` (key, value) structure and **rehashing when load factor > 0.7** to the **first prime after 2×M**.

```python
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value

class HashTable:

    def __init__(self, size=11, threshold=0.7):
        self.size = self._next_prime(size)
        self.threshold = threshold
        self.table = [None] * self.size  # slots hold: None | Node
        self.count = 0                   # number of active nodes

    # --- Hash & load factor ---
    def _hash(self, key):
        return hash(key) % self.size

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

    # --- Core operations (linear probing) ---
    def insert(self, key, value):
```

```
        # Rehash BEFORE insertion if the next insert would exceed the threshold
        self.count += 1
        if self.load_factor() > self.threshold:
            self._rehash(self._next_prime(self.size))
        idx = self._hash(key)
        probes = 0
        while probes < self.size:
            slot = self.table[idx]
            if slot is None:
                self.table[idx] = Node(key, value)
                return
            idx = (idx + 1) % self.size
            probes += 1

    def search(self, key):
        idx = self._hash(key)
        probes = 0
        while probes < self.size:
            slot = self.table[idx]
            if slot is None:
                return None  # key not present
            if slot.key == key:
                return slot.value
            idx = (idx + 1) % self.size
            probes += 1
        return None

    def delete(self, key):
        idx = self._hash(key)
        probes = 0
        while probes < self.size:
            slot = self.table[idx]
            if slot is None:
                return False
            if slot.key == key:
                self.table[idx] = None
                self.count -= 1
                return True
            idx = (idx + 1) % self.size
            probes += 1
        return False

    def _rehash(self, new_size):
        old_table = self.table
        self.size = new_size
        self.table = [None] * self.size
        old_count = self.count
        self.count = 0
        for slot in old_table:
            if slot is not None:
                self.insert(slot.key, slot.value)

    def _is_prime(self, n):
        if n < 2:
```

```
            return False
        for i in range(2, sqrt(n)):
            if n%i == 0:
                return False
        return True

    def _next_prime(self, n):
        new_size = n * 2
        while not self._is_prime(new_size):
            new_size += 1
        return new_size

# Example: size=7, threshold=0.7; will rehash to first prime > 14 when needed
ht = HashTable(size=7, threshold=0.7)
for k, v in [(50, 'A'), (700, 'B'), (76, 'C'), (85, 'D'), (92, 'E')]:
    ht.insert(k, v)
print('92 ->', ht.search(92))
ht.delete(76)
print('76 ->', ht.search(76))
```

> Implementation below uses linear probing and triggers a rehash when load factor exceeds 0.7.

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None

class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        node = self.table[index]
        if node is None:
            self.table[index] = Node(key, value)
        else:
            while node.next:
                if node.key == key:
                    node.value = value
                    return
                node = node.next
            node.next = Node(key, value)

    def delete(self, key):
        index = self._hash(key)
        node = self.table[index]
```

```
        prev = None
        while node:
            if node.key == key:
                if prev:
                    prev.next = node.next
                else:
                    self.table[index] = node.next
                return True
            prev, node = node, node.next
        return False

    def search(self, key):
        index = self._hash(key)
        node = self.table[index]
        while node:
            if node.key == key:
                return node.value
            node = node.next
        return None
```

Lecture 2: Exercises

Exercise 1 — Pair sum (Two-Sum)

Explanation. Given an array of integers and a target T, determine whether there exist two indices i ≠ j such that a[i] + a[j] = T. A hash table lets us check for each element x whether the complement (T − x) has already been seen.

Exercise 2 — First non-repeating character

Explanation. Given a string s, find the first character with frequency 1 when scanning left to right.

Exercise 3 — Duplicate detection

Explanation. Given a list A, determine if any value appears at least twice. Insert each element into a hash set and check membership before insertion. Expected time O(n).

Exercise 4 — Phonebook (dictionary) operations

Explanation. Implement a simple phonebook that maps names to phone numbers using a hash table. Support insert/update, lookup, and delete.

```
class PhoneBook:
    init(capacity, threshold):
        T = new HashTable(capacity, threshold)

    insert_or_update(name, number):
        T.insert(name, number)

    lookup(name):
```

```
        return T.search(name)  # returns number or None

    remove(name):
        return T.delete(name)  # True if removed
```

Assignment

Implement a hash table from scratch without using built-in dictionary types, supporting the following operations:

- `insert(key, value)`
- `delete(key)`
- `search(key)`
- Automatic `rehashing` when the load factor exceeds 0.7.
- Use **quadratic probing** as the collision resolution strategy, ensuring that probing steps follow the quadratic sequence (i^2) from the initial hash index.