Chapter 27 Hashing

Objectives

- To know what hashing is for (§27.3).
- To obtain the hash code for an object and design the hash function to map a key to an index (§27.4).
- To handle collisions using open addressing (§27.5).
- To know the differences among linear probing, quadratic probing, and double hashing (§27.5).
- To handle collisions using separate chaining (§27.6).
- To understand the load factor and the need for rehashing (§27.7).
- To implement MyHashMap using hashing (§27.8).
Why Hashing?

The preceding chapters introduced search trees. An element can be found in $O(\log n)$ time in a well-balanced search tree. Is there a more efficient way to search for an element in a container? This chapter introduces a technique called hashing. You can use hashing to implement a map or a set to search, insert, and delete an element in $O(1)$ time.

Map

A map is a data structure that stores entries. Each entry contains two parts: key and value. The key is also called a search key, which is used to search for the corresponding value. For example, a dictionary can be stored in a map, where the words are the keys and the definitions of the words are the values.

A map is also called a dictionary, a hash table, or an associative array. The new trend is to use the term map.
What is Hashing?

If you know the index of an element in the array, you can retrieve the element using the index in $O(1)$ time. So, can we store the values in an array and use the key as the index to find the value? The answer is yes if you can map a key to an index.

The array that stores the values is called a *hash table*. The function that maps a key to an index in the hash table is called a *hash function*.

*Hashing* is a technique that retrieves the value using the index obtained from key without performing a search.

Hash Function and Hash Codes

A typical hash function first converts a search key to an integer value called a *hash code*, and then compresses the hash code into an index to the hash table.
Collision: two keys map to the same index

Hash function: key%101

both 4567 and 7597 map to 22

The Birthday Problem

- What is the minimum number of people so that the probability that at least two of them have the same birthday is greater than ½?

- Assumptions:
  - Birthdays are independent
  - Each birthday is equally likely
The Birthday Problem

What is the minimum number of people so that the probability that at least two of them have the same birthday is greater than \( \frac{1}{2} \)?

Assumptions:
- Birthdays are independent
- Each birthday is equally likely

\( p_n \) – the probability that all people have different birthdays

\[
p_n = \frac{365}{366} \times \frac{364}{366} \times \cdots \times \frac{366-(n-1)}{366}
\]

at least two have same birthday:

\[ n = 23 \rightarrow 1 - p_n \approx 0.506 \]

The Birthday Problem: Probabilities

<table>
<thead>
<tr>
<th>N: # of people</th>
<th>P(N): probability that at least two of the N people have the same birthday.</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11.7 %</td>
</tr>
<tr>
<td>20</td>
<td>41.1 %</td>
</tr>
<tr>
<td>23</td>
<td>50.7 %</td>
</tr>
<tr>
<td>30</td>
<td>70.6 %</td>
</tr>
<tr>
<td>50</td>
<td>97.0 %</td>
</tr>
<tr>
<td>57</td>
<td>99.0%</td>
</tr>
<tr>
<td>100</td>
<td>99.999997%</td>
</tr>
<tr>
<td>200</td>
<td>99.999999999999999999999999999998%</td>
</tr>
<tr>
<td>366</td>
<td>100%</td>
</tr>
</tbody>
</table>
Probability of Collision

- How many items do you need to have in a hash table, so that the probability of collision is greater than $\frac{1}{2}$?

- For a table of size 1,000,000 you only need 1178 items for this to happen!

Collisions

**Collision**: two keys map to the same index

Hash function: key%101

both 4567 and 7597 map to 22
Methods for Handling Collisions

- Approach 1: Open addressing
  - Probe for an empty (open) slot in the hash table

- Approach 2: Restructuring the hash table
  - Change the structure of the array table: make each hash table slot a collection (e.g. ArrayList, or linked list), often called separate chaining
  - Extendable dynamic hashing

Open addressing

- When colliding with a location in the hash table that is already occupied
  - Probe for some other empty, open, location in which to place the item.
  - Probe sequence
    - The sequence of locations that you examine
    - Linear probing uses a constant step, and thus probes loc, (loc+step)%size, (loc+2*step)%size, etc.

In the sequel we use step=1 for linear probing examples.
Linear Probing, step = 1

- Use first char. as hash function
  - Init: ale, bay, egg, home
- Where to search for
  - egg hash code 4
  - ink hash code 8
- Where to add
  - gift 6 empty
  - age 0 full, 1 full, 2 empty

**Question:** During the process of linear probing, if there is an empty spot,
A. Item not found?
or
B. There is still a chance to find the item?

Open addressing: Linear Probing

- **Deletion:** The empty positions created along a probe sequence could cause the retrieve method to stop, incorrectly indicating failure.

- **Resolution:** Each position can be in one of three states **occupied, empty, or deleted.** Retrieve then continues probing when encountering a deleted position. Insert into empty or deleted positions.
Linear Probing (cont.)

- **insert**
  - *bay*
  - *age*
  - *acre*

- **remove**
  - *bay*
  - *age*

- **retrieve**
  - *acre*

**Question:** Where does almond go now?

---

**Linear Probing Animation**

For simplicity, only the keys are shown and the values are not shown. Here N is 11 and index = key % N.
Quadratic Probing

Quadratic probing can avoid the clustering problem in linear probing. Linear probing looks at the consecutive cells beginning at index k. Quadratic probing increases the index by \( j^2 \) for \( j = 1, 2, 3, \ldots \). The actual index searched are \( k, k + 1, k + 4, \ldots \)

www.cs.armstrong.edu/liang/animation/web/QuadraticProbing.html

Double Hashing

Double hashing uses a secondary hash function on the keys to determine the increments to avoid the clustering problem.

\[
h'(k) = 7 - k \mod 7;
\]
Handling Collisions Using Separate Chaining

The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a *bucket*. A bucket is a container that holds multiple entries.

Implementing Map Using Hashing

```
interface MyMap<K, V>
{
    void clear(); // Removes all entries from this map
    boolean containsKey(K key); // Returns true if this map contains an entry for the specified key.
    boolean containsValue(V value); // Returns true if this map maps one or more keys to the specified value.
    String entrySet(); // Returns a set consisting of the entries in this map.
    boolean isEmpty(); // Returns true if this map contains no mappings.
    String keySet(); // Returns a set consisting of the keys in this map.
    String remove(K key); // Removes the entries for the specified key.
    int size(); // Returns the number of mappings in this map.
    String values(); // Returns a set consisting of the values in this map.
}

MyHashMap<K, V>
{
    MyHashMap();
    MyHashMap(int capacity); // Creates a map with a specified capacity and default load factor threshold.
    MyHashMap(int capacity, float loadFactor); // Creates a map with a specified capacity and load factor threshold.
}

MyMap.Entry<K, V>
{
    K key;
    V value;
    String remove(); // Returns the key in the entry.
    String get(); // Returns the value in the entry.
    String setValue(V value); // Returns the value in the entry.
}

MyMap
MyHashMap
TestMyHashMap
Run
```
Implementing Set Using Hashing

```java
import java.lang.Iterable;
import java.util.Iterator;

public interface MySet<E> {
    Iterator<E> iterator();
}

public abstract class MyHashSet<E> implements MySet<E> {
    public abstract void clear();
    public abstract boolean contains(E e);
    public abstract boolean add(E e);
    public abstract boolean remove(E e);
    public abstract boolean isEmpty();
    public abstract int size();
}

MySet MySet
MyHashSet MyHashSet
TestMyHashSet
Run
```