# Week 10: Hash Table

## Readings
- **Required**
  - [Weiss] ch20
- **Exercise**
  - 20.5

## Recap

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unsorted Array/List</th>
<th>Sorted Array</th>
<th>BST</th>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(log N)</td>
<td>O(1) avg</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(log N)</td>
<td>O(1) avg</td>
</tr>
<tr>
<td><strong>Find</strong></td>
<td>O(N)</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td>O(1) avg</td>
</tr>
<tr>
<td><strong>findMin</strong></td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(log N)</td>
<td>O(N)</td>
</tr>
<tr>
<td><strong>findMax</strong></td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(log N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

Hash Table is a data structure that supports the most common dynamic set operations in constant time on average. It has many applications.

Direct address table is a simplified version of hash table.
Consider the problem of maintaining information about SBS (and TIBS) bus services. We want to support three operations find, insert and delete.

Since bus numbers are integers between 0 – 999, we can create an array with 1000 booleans, initialized to false. If bus service N exists, just set position N to true. All find, delete, and insert can be done in O(1) time.

We can extends this idea, if we want to maintain additional data about a bus. Use an array of 1000 slots, each can reference to an Object.
Restrictions

- Keys must be integer
- Range of keys must be small

This works only if keys are integers, (cannot keep track of bus no NR10, 162M) and the range for the keys must be small (if keys are phone numbers, you need an array of size 10 million).

Hash Table

Hash Table is a generalization of direct addressing table, to remove these restrictions.

Idea

- Map non-integer keys to integers
- Map large integers to smaller integers

The idea is to map any keys to small integers. We call this hashing. The function that map keys to integers are call hash function.

Hash Table

h is a hash function. This example shows how we map phone numbers to slot numbers between 0 and 999.
Here is the pseudocode: notice that we have replaced key with h(key).
(This does not work! See next slide)

But a hash function does not guarantee that two different keys goes into different slots! This is called a “collision”.

To implement hash table, we need to answer two questions: how to define a hash function and how to resolve collision. They are important issues that can affect the efficiency of hash table.
Hash Functions

Good Hash Functions
- appear random
- fast
- depends on all information in the key
- keys that are close have hash values that are far apart

Perfect Hashing Function
- One-to-one mapping between keys and hash values.
- Maybe possible if all keys are known

Uniform Hashing Function
- Distributes keys evenly
- Example
  - if k are integers uniformly distributed among 0 and X-1
  - \[ k \in [0, X) \]
  - \[ \text{hash}(k) = \left\lfloor \frac{km}{X} \right\rfloor \]

It is possible to have a perfect hash function: where collision is guaranteed not to occur.

A uniform hashing function put a key into a slot with equal probability.
Hashing Integers

There are many ways to hash an integer.

The most popular one is the division method: where we use the mod operator (% in Java) to map an integer to values between 0 and m-1 (inclusive).

\[ hash(k) = k \mod m \]

The choice of m (or hash table size) is important. If m is power of two, say \(2^n\), then key modulo of m is the same of last n bits of the key.
If m is \(10^n\), then our hash values is the last n digit of keys.
We usually pick m to be a prime number close to a power of two.
Rule

- Pick \( m \) to be a prime number **not too** close to power of two.

Multiplication Method

1. Multiply by a number \( 0 \leq A < 1 \)
2. Extract the fractional part
3. Multiply by \( m \)

\[
hash(k) = \left\lfloor m(kA - \left\lfloor kA \right\rfloor) \right\rfloor
\]

Another method is the multiplication method. The golden ratio \( = \frac{\sqrt{5} - 1}{2} \) seems to be a good choice for \( A \).

Hashing Strings

To hash a string, we can just sum up all ascii values of each character.

Hashing of Strings

\[
\text{hash}(s, m) = \frac{\sum \text{ascii values of } s}{m} \mod m
\]
= (84 + 97 + 110 + 32 + 65 + 104 + 32 + 84 + 101 + 99 + 107) % 11
= 825 % 11
= 0

This only depends on the characters that are present in a string, not their positions.

A better way is to “shift” the sum everytime, so that the position affects the calculated hash values. (Note: Java’s String.hashCode() uses 31 instead of 37)
Probability of Collision

- von Mises Paradox: "How many people must be in a room before the probability that some share a birthday, ignoring the year and leap days, becomes at least 50 percent?"

\[
Q(n) = \text{Probability of unique birthday for } n \text{ people} = \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \ldots \times \frac{365-n+1}{365}
\]

\[
P(n) = \text{Probability of collisions for } n \text{ people} = 1 - Q(n)
\]

\[
P(23) = 0.507
\]

Probability of Collision

Collision is very likely!

If we more than 23 keys into a table with 365 slots, more than half of the time we get collision.

Collision Resolutions

- Separate Chaining
- Linear Probing
- Quadratic Probing
- Double Hashing
Separate Chaining is the most straightforward method, using a linked-list to store the collided keys.

Insertion can be done in O(1) time. But deletion and search takes O(n) time where n is the length of the list.
Average Running Time

- Search: $O(1 + L)$
- Insert: $O(1)$
- Delete: $O(1 + L)$

If $L$ is bounded by some constant, then all three operations are $O(1)$

Rehashing

To keep $L$ bounded, we may need to reconstruct the whole table when the load factor exceeds the bound. We need to rehash all keys into a bigger table (increase $m$ to reduce $L$)

Linear Probing

In linear probing, when we get a collision, we scan through the table looking for an empty slot (wrapping around when we reach the last slot)

However, we can bound the length of the chain by a constant.
21 collides with 14. Look for the next empty slot.

1 collided with 21. Look for an empty slot.

Find a values is similar to find. We probe the array starting from the original hash position (in this case hash(35) = 0).
When probing, if we reach an empty slot, we know that the value does not exist in the hash table.

To delete, we first find the value, and remove it from the table.

We cannot simply remove a value, because it can affect \text{find}(\ ) !
How to delete?

- Lazy Deletion
- Three different states
  - occupied
  - occupied but mark as deleted
  - empty

When a value is removed from linear probed hash table, we just mark it as “deleted”, instead of emptying the slot.

Delete 21

Find 35

Insert 15

When we insert, we can put a value into either an empty slot, or a slot that has been marked as deleted.
The problem with linear probing is that it can create many consecutive occupied slots, increasing the running time of find/insert/delete. This is called primary clustering.

An improvement to linear probing is quadratic probing.

The probe sequence for linear probing is this.
For quadratic probing, we use this probe sequence.

\[
\text{hash(key)}\equiv (\text{hash(key)} + 1) \mod m \\
(\text{hash(key)} + 4) \mod m \\
(\text{hash(key)} + 9) \mod m \\
\vdots
\]

Notice that the calculation of \(+1 +4 +9 \ldots\) starts from the original hash position. If we were to start from the previous probe position, the probe sequence should be \(+1 +3 +5 \ldots + (2i -1)\).

(Q: Show mathematically that they are the same)

How can we be sure that quadratic probing always terminate? Insert 12 into the previous example, follow by 10. See what happen?
Problems

- If two keys have the same initial position, their probe sequence is the same.
- Secondary clustering.

Double Hashing

Double hashing uses a second hash function to calculate the probe sequence, so unless two keys have the same hash values for both hash functions, they have different probe sequences.

Double Hashing

\[
\begin{align*}
\text{hash}(\text{key}) &= \text{hash}_1(\text{key}) \\
(\text{hash}(\text{key}) + \text{hash}_2(\text{key})) \mod m &= \text{hash}_1(\text{key}) \\
(\text{hash}(\text{key}) + 2\times\text{hash}_2(\text{key})) \mod m &= \text{hash}_1(\text{key}) \\
(\text{hash}(\text{key}) + 3\times\text{hash}_2(\text{key})) \mod m &= \text{hash}_1(\text{key}) \\
\end{align*}
\]

hash_2(key) is the secondary hash function.

Insert 21

We use k\%5 as the secondary hash function in this example. Can you give two keys that have the same probe sequence in this example?

If we insert 21, the probe sequence is the same as linear probing.

Using quadratic probing requires more careful design of hash table. It also suffers from a (less minor) problem – if two keys has the same initial position, they have the same probe sequence.
If we insert 4, the probe sequence is 4, 8, 12 … (from the first probe position) or 4, 4, 4, … (from the previous probe position).

But if we insert 35, the probe sequence is 0, 0, 0, …
What is wrong?

Warning
- Secondary hash function must not evaluate to 0!
- Change hash₂(key) to
  \[ \text{hash₂(key)} = 5 - \text{(key \% 5)} \]

Good Collision Resolution
- Minimize clustering
- Can find an empty slot if \( L \) is small
- Give different probe sequence when initial probe is the same
- Fast