Hash Tables

Books

Everyday Data Structures
https://www.packtpub.com/application-development/everyday-data-structures
https://www.packtpub.com/application-development/
everyday-data-structures
Examples using C#, Java, Objective-C, and Swift

Learning Functional Data Structures and Algorithms
https://www.packtpub.com/application-development/learning-functional-data-structu
https://www.packtpub.com/application-development/
learning-functional-data-structures-and-algorithms
Examples using Scala and Clojure.

Beginning Haskell: A Project-Based Approach
https://www.apress.com/us/book/9781430262503
Introduction to Haskell.

1 Hash Tables

Searching for an element in a tree can be very fast, but slows down as the number of items increases (at best $\mathcal{O}(\log N)$).

Why?

- Tree imbalance
- No direct access

What about using an array?

- Direct access
- Not working with just numbers

Need/Want some sort of conversion scheme to use an array.

imbalance/unbalance — Merriam Webster

```
http://www.merriam-webster.com/dictionary/imbalance
Main Entry:  im·bal·ance

Pronunciation: \(.)im-?ba-len(t)s\
Function: noun
Date: circa 1890
: lack of balance : the state of being out of
equilibrium or out of proportion <a vitamin imbalance>
<racial imbalance in schools>

Main Entry: 2 unbalance
Function: noun
Date: 1855
: lack of balance : imbalance
```

1.1 Hashing

A **Hash function** transforms keys into an array index.

One possible implementation is to sum the ASCII values of each character in a string and *mod* it with the table size. For example, given an array with 120 elements and the string JONES:

J	74
O	79
N	78
E	69
S	83

$74 + 79 + 78 + 69 + 83 = 383$

$383 \ \% \ 120 = 23$

One problem with all hash functions is that they lead to **collisions**. A collision is the result of two or more keys hashing to the same value (location)[1].

How to handle collisions?

[1]Physics: Two objects cannot occupy the same space at the same time.

1.2 Collision Resolution

Need some sort of **collision resolution strategy** to handle collisions when they occur.

Two common strategies:

- Linear resolution
- Open hashing

1.2.1 Linear Resolution

The simplest strategy is **linear resolution**:

If $h(x)$, for some key x, points to a location that is already occupied, inspect the next location in the array. If that location is full, try the one after that, and so on, until we find a vacant location or find that the array (**hash table**) is completely full.

Example The hash function is given by:

$$h(x) = x \,\%\, 7$$

(the table has seven locations).

Insert the integers (keys) $23, 14, 9, 6, 30, 12$, and 18 into the table, T (recall that its indices are $0, \ldots, 6$).

1. $h(23) = 2$, so key 23 would be stored in $T[2]$.
2. $h(14) = 0$, so key 14 would be stored in $T[0]$.
3. $h(9) = 2$, but $T[2]$ is already occupied, so key 9 would be stored in $T[3]$.
4. $h(6) = 6$, so key 6 would be stored in $T[6]$.
5. $h(30) = 2$, but $T[2]$ and $T[3]$ are already occupied, so key 30 would be stored in $T[4]$.
6. $h(12) = 5$, so key 12 would be stored in $T[5]$.
7. $h(18) = 4$, but $T[4]$, $T[5]$, and $T[6]$ are already occupied, so we *wrap around* the array indices and go back to $T[0]$, which is also occupied, so key 18 would be stored in $T[1]$.

0	1	2	3	4	5	6

It took 14 **probes** into the hash table to fill it in this example: Seven (7) initial probes to hash the elements into the table and seven (7) more probes to resolve the collisions.

The other problem that we encountered was **clustering**. Keys form clusters when several elements whose keys hash to the same/nearly the same value are inserted into the table.

1.2.2 Open Hashing

Another way to resolve the collision/clustering problem is to use **Open Hashing** (also referred to as *chaining*). We ignore the collision and simply place the value in a *bucket* (which is what table cells are usually called), along with the values that are already there. This technique is usually accomplished by using a linked list for every bucket, accessed by pointers in the hash table.

Example The hash function is given by:

$$h(x) = x \,\%\, 7$$

(the table has seven locations).

Insert the integers (keys) $23, 14, 9, 6, 30, 12$, and 18 into the table, T (recall that its indices are $0, \ldots, 6$).

1. $h(23) = 2$, so key 23 would be stored in $T[2]$.
2. $h(14) = 0$, so key 14 would be stored in $T[0]$.
3. $h(9) = 2$, so key 9 would be stored in $T[2]$.
4. $h(6) = 6$, so key 6 would be stored in $T[6]$.
5. $h(30) = 2$, so key 30 would be stored in $T[2]$.
6. $h(12) = 5$, so key 12 would be stored in $T[5]$.
7. $h(18) = 4$, so key 18 would be stored in $T[4]$.

0	1	2	3	4	5	6