Hashing

Overview and Analysis

Hashing

Hashing converts a value to some other consistent form

- for example, we can hash an object's key to directly obtain its index in an array
- other hashing algorithms are useful in databases, similarity comparisons, and elsewhere

Hashing is great for searching

- worst-case complexity is still linear
- however, expected average case complexity under most circumstances is constant!
- similarly, insertion and deletion can be done in average-case constant time

Hashing is not limited to numeric values

- countless hash algorithms exist to convert strings (can be entire files!) into a numbers with some number of bits (binary digits: 1's and 0's)
- we will talk about keys that are already numeric values, to make life simpler =)

A Simple Hashing Scheme

Using the book's example, let's say you're storing tractors... O_o

- a company makes lots of different types of tractors, each with different names, specifications, prices, and ids

An example data type:

```
struct Tractor {
    int key;
    double cost;
    int horsepower;
};
```

A Simple Hashing Scheme

Using the book's example, let's say you're storing tractors...

- a company makes lots of different types of tractors, each with different names, specifications, prices, and ids

Goal:

- we want to store the tractors so that searching (and, coincidentally, insertion and deletion) is typically a constant-time operation

Use a table-like data structure:

- store Tractors in an array of 50 elements

```
Tractor data[50];
```

- hash each tractor's key to determine its index in the array

```
data[ hash(my_tractor) ] = my_tractor;
```

A Simple Hashing Scheme

Let's say we know that Tractor ids have this pattern:

- `0, 100, 200, ..., 4800, 4900`

In this case, an ideal choice for a hashing algorithm would be this:

```
size_t hash(const Tractor& t) {
    return tractor.key / 100;
}
```

Each tractor's key directly converts to a unique index in the array

- how nice!
- of course, we are seldom this fortunate...

A Simple Hashing Scheme

What if we don't know the pattern of keys?

- `12, 78346, 47, 745, 974, ...`

In this case, we could use something like this:

```
size_t hash(const Tractor& t) {
    return tractor.key % 50;
}
```

Each tractor's key still directly converts to an index in the array

- however, it is not guaranteed to be unique!
- duplicate hash values for different keys are called collisions

Handling Collisions

In most practical applications of a table:

- the number of values a table can store is commonly significantly less than the total number of *possible* values
- this means multiple values will hash to the same index in the array
- these collisions must be handled somehow

Multiple different schemes

- open-address hashing
- double-hashing
- chained hashing

Open-Address Hashing

Algorithm for storage in open-address hashing:

- compute the index of each value as `hash(key)`
- if `data[hash(key)]` does not already contain a record, then store the new value in that location; all done!
- however, if `data[hash(key)]` already contains a value (a collision), then try the next spot, `data[hash(key) + 1]`
- keep repeating this process until a vacant spot is found, wrapping around to the beginning of the array as needed

This process is called *linear probing*

- a potential problem with this scheme is clustering of records, leading to worse performance for operations

Double Hashing

Double hashing is another approach:

- use a second hash function to determine how to move through the array in the event of collisions

Example:

- assume there are two hash functions, `hash1` and `hash2`
- when an item is inserted, double hashing starts by hashing the key to an index using `hash1(key)`
- If there is a collision, then calculate `hash2(key)`, which tells us how many spots to jump during linear probing
- if `hash2(key)` is `7`, then we look for a vacant array spot every `7` elements, starting at `hash1(key) + 7`, then `hash1(key) + 14`, and so forth...

Double Hashing

Double hashing is another approach:

- use a second hash function to determine how to move through the array in the event of collisions

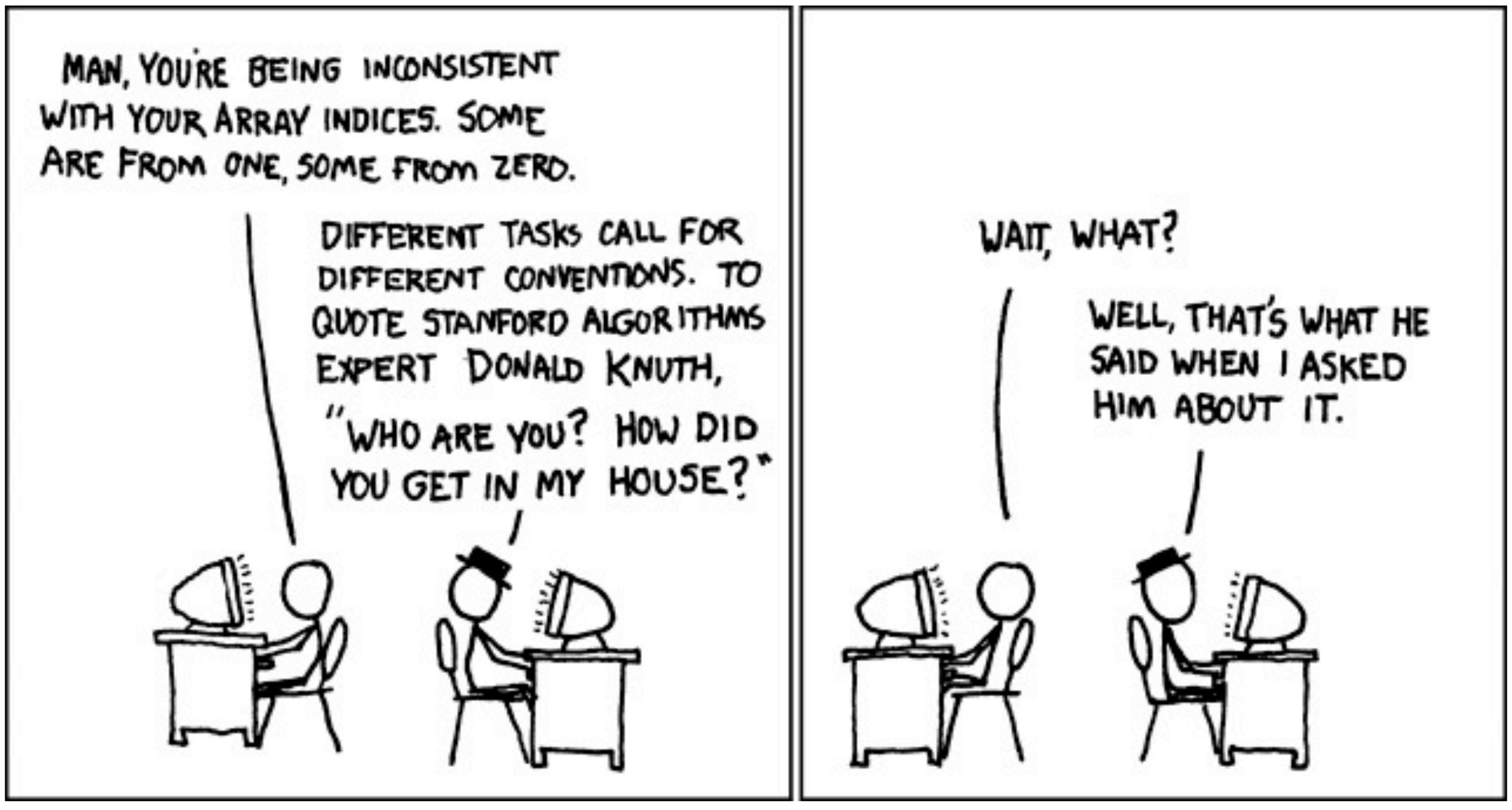
So sayeth Donald Knuth:

- both `CAPACITY` and `CAPACITY-2` should be prime numbers (such as `811` and `809`); this ensures that visiting every `xth` element will still visit all the spots in the array
- `hash1(key) = key % CAPACITY`
- `hash2(key) = 1 + (key % (CAPACITY - 2))`

Double Hashing

Double hashing is another approach:

- use a second hash function to determine how to move through the array in the event of collisions



Chained Hashing

In the other two addressing schemes:

- collisions were handled by probing the array for an unused position
- each array element could hold just one record

Chained hashing works differently:

- each array element can hold more than one record
- still hash the key of an entry to determine where it goes
- however, in the event of a collision, simply add the new entry to the other one at the correct spot
- this can be done, for example, by having each array element be the head pointer to a linked list of values whose keys map to that index

Choosing a Hash Function

We saw a simple hash algorithm for integers:

```
key % 50
```

For a division-based hash function like this:

- certain table sizes are better than others to reduce collisions
- for example, some guy named Radke did a study that suggests a good table size is a prime number in the form of 4k+3, like `811` (`4 * 202 + 3`)

Other types of hash functions:

- mid-square functions, where the key is multiplied by itself and some of the middle digits of the result are used as the hash
- multiplicative hash functions, where the key is multiplied by a constant less than one and some of the first few digits of the fractional part of the result are used as the hash

Time Analysis of Hashing

The performance of a table with hashing depends on its load

- the load of a table is how many items are in the table, relative to its total capacity
- basically, what percentage of the table is occupied?

Lower loads lead to better performance

- you never want to have a full or nearly full table; performance degenerates to linear time for searching, inserting, and deleting

With a load of less than 0.8:

- average-case complexity is 3
- constant-time complexity for average case!