Hash Tables

Part I

Anton Gerdelan

<gerdela@scss.tcd.ie>

Review

- labs - pointers and pointers to pointers
- memory allocation
- easier if you remember
    - **char*** and **char**** are just basic ptr variables that hold an <u>address</u>
    - **int** and **char*** and **char**** are just conveniences over assembly code
        - e.g. 'what offset size should i use for … [i], +, /, …'
        - e.g. [i] on a 1 byte type says 'get address of start and add i * 1'
        - the i*1 bit is called the '**stride**' - how long is each 'step' in memory

I have a big array of People of size n.

I need to find one holding a name variable “anton”.

Linear search - big-O?
Pre-sorted binary search - big-O?

It would be great If I could just do:

```
Person me = people_array[“anton”]
```

and get **O(1)** indexing.
But this doesn’t work.

How can I make this work?

If You Can Prepare the Data in Advance

- Assign each person that is created an unique index to the array.
    - -> Or create a separate **look-up table** |name|array index|
    - -> Usually you can do this. **Done**.
- If we can't prepare the data for each **key** (names for us) - we need to **search** the data structure.
    - e.g. *"Is there a user called 'anton' in the database?"*
    - -> Difficult. Evaluate **hash table** as an **alternative to searching**. Use name as the **key**.

Can we make a **function** that just turns a **string into an integer**?

How?

Create a **Hash Function**

- return sum of character codes in string?

```
int index = 'a' + 'n' + 't' + 'o' + 'n';
          = 97 + 110 + 116 + 110 = 433;
```

- suggest some improvements to me:
  - what if the sum is bigger than our array size?
  - what if we have e.g. names: ***adi*** and ***ida***?

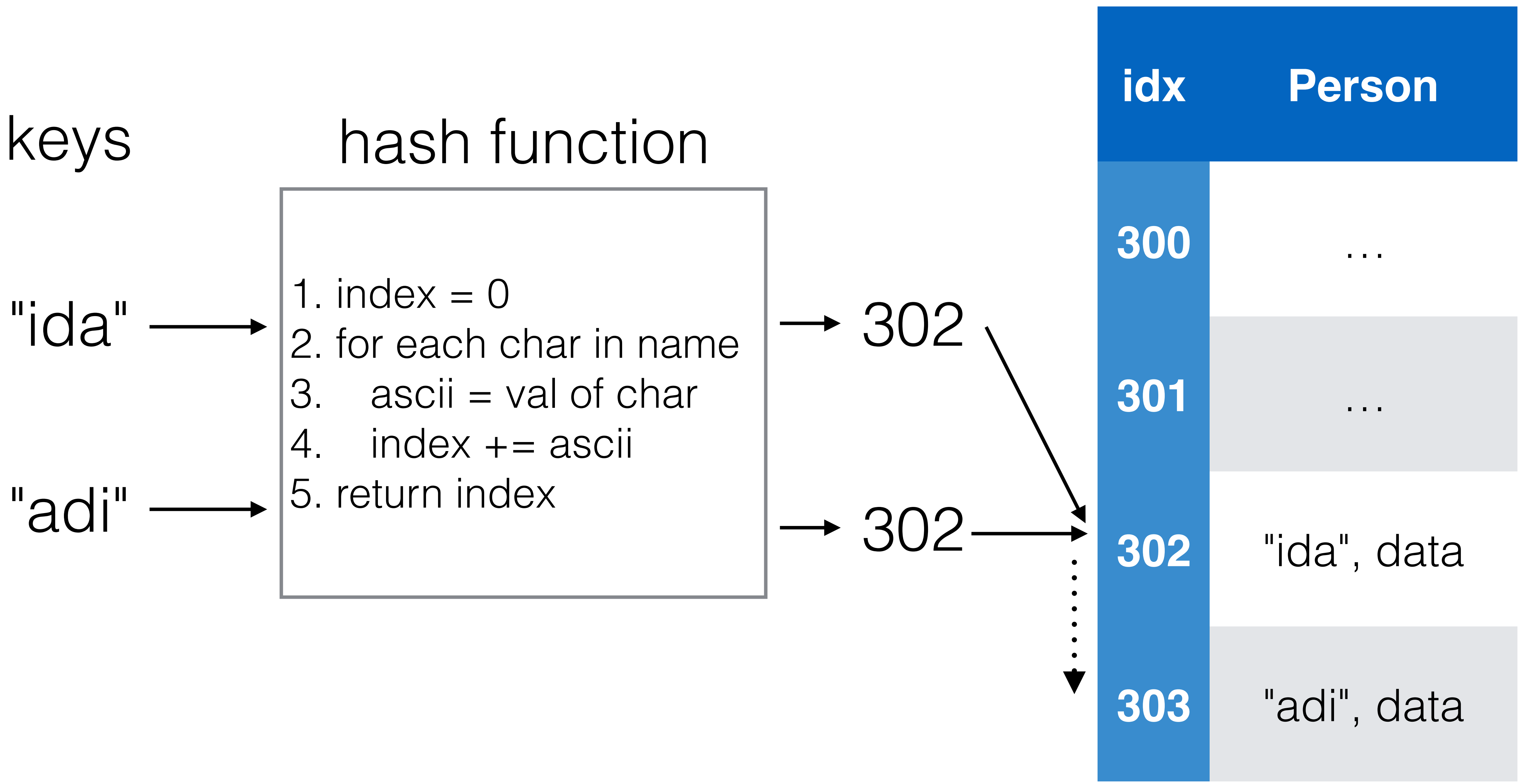
Dealing with Limitations

- Make the array bigger to avoid collisions
  - More wasted space -> **space complexity ++**
- Can't be perfect - allow some collisions
  - More collisions -> **time complexity ++**
- Improve hash function to reduce collisions
  - Hard. May <u>over-fit</u> to *test* input instances.

Allow Collisions

- Must allow some collisions or have infinite storage
- Several strategies exist
  - "Use the next index down"
  - Put a linked list behind every index
  - Cost of each? {Coding, Time, Space}

Use the Next Index Down "Linear Probing"

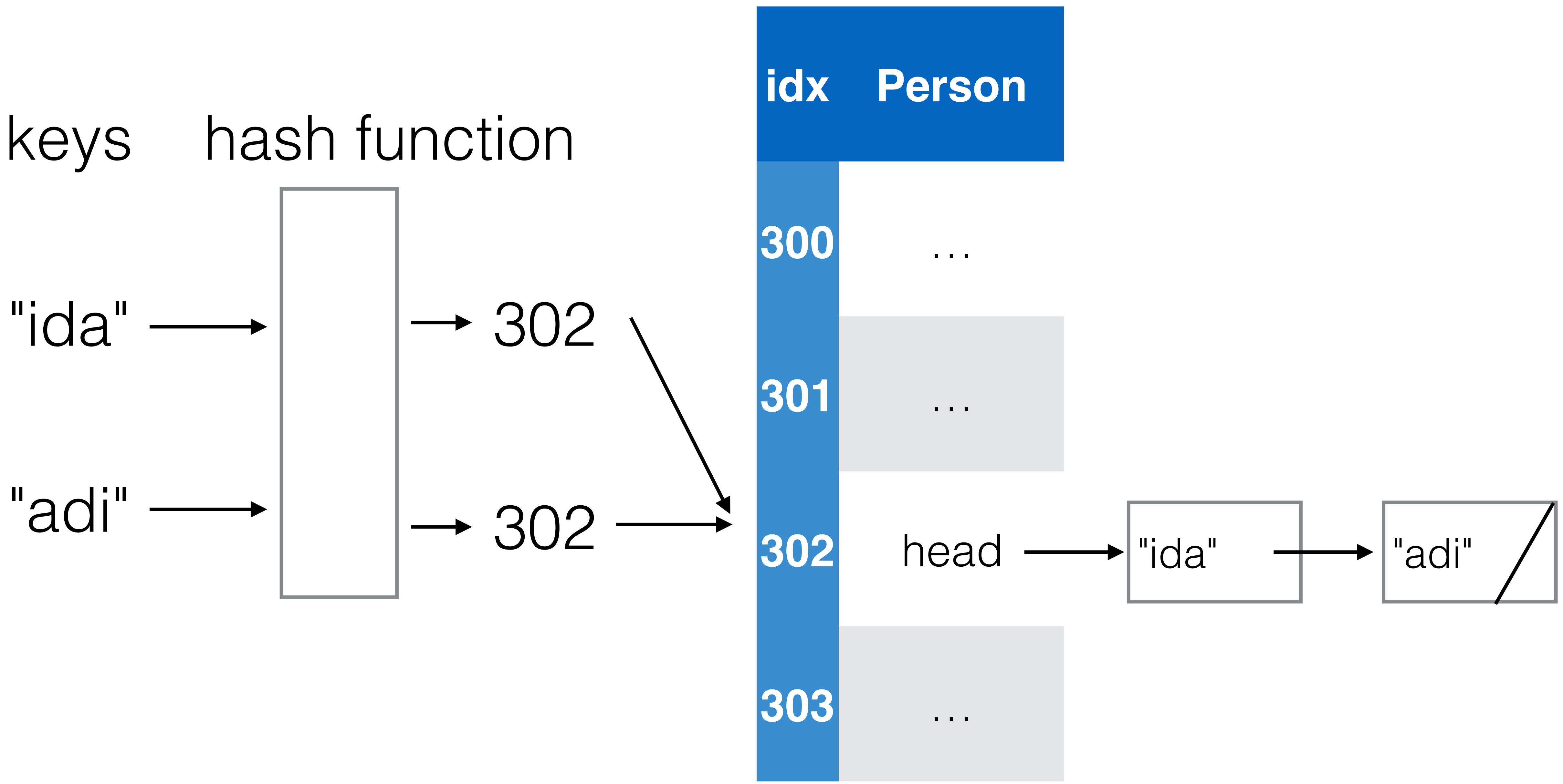


Q. downsides?

Use the Next Index Down

- Relies on keys being mostly evenly distributed with some space in-between
- If keys are clustered
    - Becomes a plain linear array search again
    - tweak hash function
    - enlarge array *S(bigger)*
- Easy to implement (can not be understated)

Chaining Hash Tables



Q. Big-O best/average/worst?

Chaining Hash Tables

- Avoid having to distribute gaps in hash table
- Put a linked list behind each array index
- Inherits pros and cons of linked lists
    - Which are?
    - (what are our criteria for evaluating data structs?)

Part II (lecture 8)

~ Rehash ~

Improve the Hash Function

- Generate more unique values

```
int index = name[0] + name[1]*M^1 + name[2]*M^2 + …
```

  - warning: long strings will get too big for number and ???
    (split them up so exponents don't get too high)

- Fit into a smaller array/table

```
M = 256
my_hash_table[M];
index = index % M;
```

  - Can we do better? Why might **256** be a problem?

Powers of 2 are a problem?

- hash function *h(k) = k % m*
  h(k) is function returning index
  k is key input
  m is max size of table

- if *m* is a power of 2, written *m=2^p*

- books say: then h(k) is just the p lowest-order bits of k

- `~~ int index = lowest M bits of index;`

Improve the Hash Function

- A common strategy uses **prime numbers** - the product of a prime with another number has a <u>very good</u> chance of being [more] unique.

- Choose table size such that it is a prime near the size you expect.

- Choose constant k such that it is the same prime.
  e.g. change table[256] to table[251]

  int index = first letter * 251 + second letter * 251 ^2 …
  index = index % 251;

Different Collision Methods

- Separate **chaining** - our linked lists add-on
- Can also use an array at each table index as "**buckets**" (not as flexible)
- **Open addressing** hashing methods:
  - "**Linear probing**" - our '*use the next value along*'
    - load factor = item_count / array_size
    - when load factor > ~2/3 then perf suffers
    - uses <5 probes on avg. for a table <2/3 full
  - Rehashing and **double hashing**
  - **quadratic probing**
- … there are lots of them! implementations differ between books/programmers etc.

Double Hashing

- h(k,i) = (f(k) + i * g(k)) % M
- where j and k are auxiliary hash functions
- first probe goes to array[f(k)]
- additional probes are offset not by 1, but by the second function
- stepping by >1 means you might miss values. so…

Double Hashing

- to cover entire array g(k) must be *relatively prime* to M
  - M is power of 2 and g(k) always returns odd number
  - or M is prime and g(k) always returns positive number less than M
  - can work with other setups but difficult to predict coverage
- example where M is prime:
  - f(k) = k % M
    g(k) = 1 + (k % M')
    where M' is a slightly smaller M, e.g. M -1
  - will examine e.g. every 257th slot until all slots examined.

Minimum Knowledge

- Read **at least one book**'s summary (some are online) of different hash table methods
- **Implement** your open simple open addressing function (linear probing)
- Know how to **draw/explain** a probing method
- Know when a hash table **is and isn't an advantage**
- **Consider improvements** to code with double hashing or chaining. Read some blogs/code from others.

Comparison

- Time complexity can depend on table load
- for large arrays and input strings at 90% load:
  - linear probe takes avg. **50** probes for unsuccessful search
    - generates O(m) range of values for keys
  - double hashing takes **10**
    - generates O(m^2) values for keys (2 functions)
  - don't let a hash table get 90% full!
  - keep load small or don't use hash tables (space hungry)

Comparison

- open addressing is hard to compare to chaining
  - chaining may be better when memory req. not known in advance
  - otherwise double hashing wins (by a small margin)
- Cormen et. al. "*Algorithms*" have the best (most methods + lengthy + proofs) coverage of hash tables

Are they right?

- Try it!
- I tried w/ short input strings
  - what's the biggest number in a 32-bit unsigned int?
  - what values does pow(120, i) give with a string of length 32?
  - split long strings or replace pow() with something else
- I hashed against: { 8, 16, 32, 64 }
- and then primes: { 7, 17, 31, 61 }

"Rate My Hash Function"

- Ratio of space used - "**load factor**". maximum is ~90%
- Frequency of double-ups
- Spread over table - clustered (~worse) or even (~better)?
- Rate by <u>**Average**</u> time complexity. Is our O(1 + *a*) Closer to O(1) or O(n)?
  - Function must suit actual input instances, not just on paper
- If your data size *n* is small, you may have fallen for a <u>trick question</u>.
- Programmers often refine their own, personal 'awesome simple hash function' in their personal toolkit/header.

Hash Function Strategies

- Division (remainder)   index = key **%** n
- Compression   **sum** or **xor** of large(er) input data
- Extraction
  use only (more unique) **part** of the key as index
- Middle of square   key = key^2.
  key = extract middle part of key (more unique)
- <u>Know what key data looks like</u> to guide you making more efficient function

Hashing Touches other Disciplines

- Hash functions aren't just for tables
  - e.g. SHA algorithm (Secure Hash Algorithm 1)
    - output a **checksum** of particular length
    - run 'checksum myfile' on your computer - compare output
  - Cryptography
  - File integrity
    - download not corrupted
    - this is the original file, nothing injected

Hash Function Algorithm Design

- Input data instance (our key)
  - short string, uint, address, whole file
- Output data permutation
  - table index between 0 and *n* - 1
    - ideally each output index is equally likely (even distribution)
  - or e.g. 20-byte checksum (usually display as hex)
- Algorithm is .: arithmetic and similar to a random number generator
  - -> this is <u>looking for a math. function</u> with even distribution
  - transform keys into numbers first - so we can do arithmetic on them