Hash Functions

Hash function:

```
int hash(keytype key);
```

maps item's key to an array slot

```
A[ hash(item->key) ] = item;
```

Desirable features and requirements of a hash function:

- Output value within bounds of the array
- Should minimize collisions, as far as possible
- Should spread items throughout the table

Hash Functions

Some bad hash functions

- `A[100]; hash(key) = key%10`
- `A[100]; hash(key) = key%100`

Better:

- `A[97]; hash(key) = (key * BIGPRIME)%97`

Prime numbers:

- disrupt patterns in data
- spread it throughout the table.

Hash Functions

Student numbers example:

- 3 first numbers
- 3 last numbers
- 0-9 buckets
- Key: First digit

Hash Functions

3 first numbers

Bucket	Key
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Functions

`3 last numbers`

Bucket	Key
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Quiz: Hash function properties

Suppose we have b buckets and k keys (k > b). Which properties should the hash function have?

a) Output a unique number between 0 and k-1

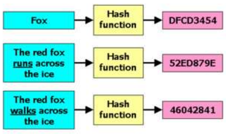
b) Output a number between 0 and b-1

c) Map approximately k/b keys to bucket 0

d) Map approximately k/b keys to bucket 1

e) Map more keys to bucket 0 than to bucket 1

Hash functions for strings



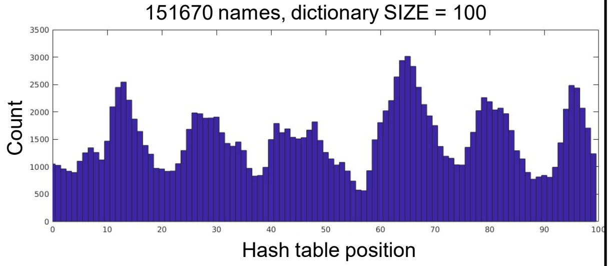
Record with string key *s*, array dictionary size **SIZE** might be stored in location:

Example:

```
A[ (17*s[0] + 37*s[1] +101*length(s))%SIZE ]
```

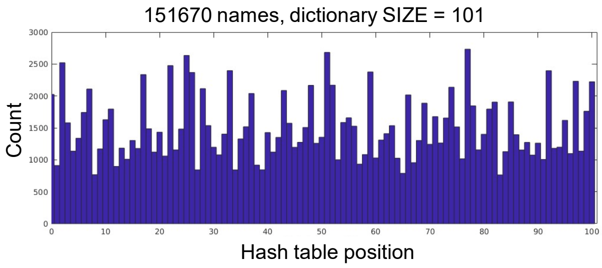

Hash table: US surnames

151670 names, dictionary SIZE = 100



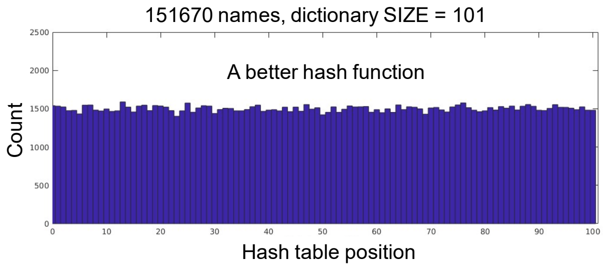
US surname dataset: https://data.world/fivethirtyeight/most-common-name

Hash table: US surnames



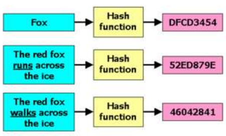
US surname dataset: https://data.world/fivethirtyeight/most-common-name

Hash table: US surnames



US surname dataset: https://data.world/fivethirtyeight/most-common-name

Hash functions for strings

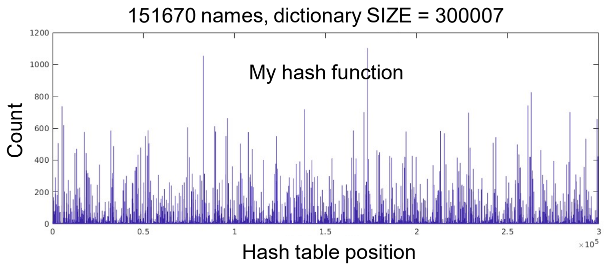


Use bigger prime numbers?

Example:

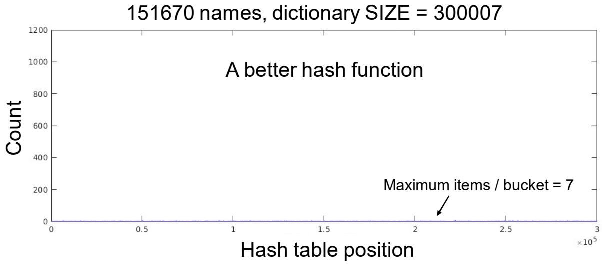
```
A[ (76771*s[0] +
    44773*s[1] +
    99017*length(s))%SIZE ]
```


Hash table: US surnames



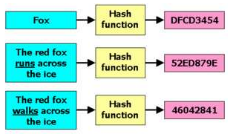
US surname dataset: https://data.world/fivethirtyeight/most-common-name

Hash table: US surnames



US surname dataset: https://data.world/fivethirtyeight/most-common-name

Hash functions for strings



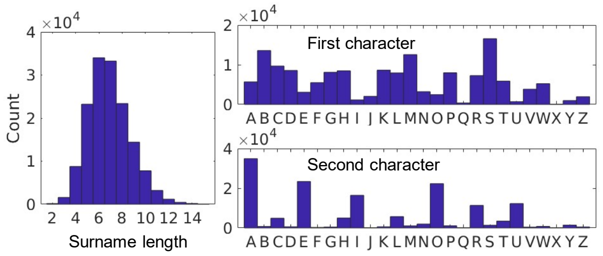
Why doesn't it work?

Example:

```
A[ (76771*s[0] +
    44773*s[1] +
    99017*length(s))%SIZE ]
```


Hash functions for strings

Strings have patterns that are hard to break!



US surname dataset: https://data.world/fivethirtyeight/most-common-name

Hash functions for strings

- Skiena (p.89) shows mapping strings to number, base alphabet size α:

$$H(S) = \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(s_i)$$

- **H("cat") = 26^2 * 3 + 26 * 1 + 1 * 20**
- Does this work for longer strings?
- Is this efficient?

Hash functions for strings

More efficient:

- Use a power of 2 instead of alphabet size:

Implementation:

```
H("cat") = 32² * 3 + 32 * 1 + 1 * 20;
      hashcat = (('c')*(1<<10)) +
                (('a')* (1<<5)) +
                 ('t'))%TABLESIZE;
         OR
      hashcat = ((‘c’ << 10) +
                 (‘a’ <<5)  +
                 (‘t’))%TABLESIZE;
```


Hash functions for strings

More efficient *and* prevent overflow:

- Use a power of 2 instead of alphabet size:

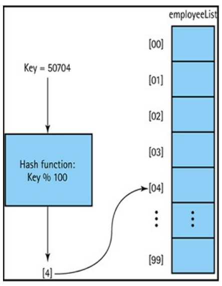
```
H("cat") = 32² * 3 + 32 * 1 + 1 * 20;
        hashcat =((((('c'* 1<<10) %TABLESIZE)
                +('a' * 1 <<5)%TABLESIZE)
                +'c')%TABLESIZE;
```

Principle:

$$(a+b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$$

Hash tables: key idea

- Huge range of possible keys
  - e.g. space of possible surnames: 26^n
    - 26^{10} = 141,167,095,653,376
- Map to a smaller set of array indexes, 0..m-1
  - hash function: *h*
  - easily computed
  - even distribution



Collisions

Collision: Two keys map to the same array index

$$h(k_1) == h(k_2)$$

When array **SIZE** < number of records

- *definitely* have collisions

When array **SIZE** > number of records

- often have collisions – and we *must* handle them

Good hash functions have fewer collisions, but *we can never assume there will be none*

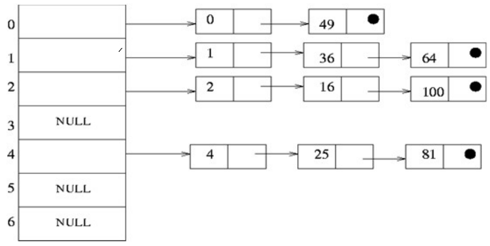
Collision Resolution Methods

1. Chaining

2. Open addressing methods
   - Linear probing
   - Double hashing

Chaining

Average length of the lists?



Demo: https://www.cs.usfca.edu/~galles/visualization/OpenHash.html


```
void insert( HT, item )
{
   new newnode = /* ... make a list node */
   /* put --item-- in the list node */

   index = hash(item->key);

   if( HT[ index ] == NULL)
      HT[ index ] = newnode;
   else
   {
      newnode->next = HT[ index ]->node;
      HT[ index ] = newnode;
   }
}
```


Linear chaining

What happens if:

- you forget to null the table initially?
- all the items hash to the same location?
- number of items is much bigger than the table?

Chaining: analysis

- Insertion:
  - Best case
  - Worst case
  - Average Case
- Search
  - Best case
  - Worst case
  - Average Case

Chaining: analysis

Average case:

- fast lookup when table is not heavily loaded

Performance degrades when table gets crowded

- Eventually degenerates to a linked list

Extra time and space for pointers

Open addressing: Linear probing

If there is a collision, put the item in the next available slot

```
while( HT[ index ] != NULL )
  index = (index + 1)%TABLESIZE
  /* only get out of this loop when
  get to a vacant spot */
```


Open addressing: linear probing

m = 20, f(k) = k % m

Initial Situation

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

After inserting 34, 55, 12, 8, 45, 37, 32, 88, 98, 54

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
–1	–1	–1	–1	–1	45	–1	–1	8	88	–1	–1	12	32	34	55	54	37	98	–1
					0			0	1			0	1	0	0	2	0	0	

After inserting 21, 42, 56, 74, 52, 33, 16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
74	21	42	52	33	45	16	–1	8	88	–1	–1	12	32	34	55	54	37	98	56
6	0	0	11	11	0	10		0	1			0	1	0	0	2	0	0	3

Linear probing

- What happens when:
  - Hash table is lightly loaded?
  - Hash table is heavily loaded?
  - Hash table is full?

Linear probing: Biggest problems

Catastrophic failure when table full

Clustering:

- Some parts of the table may fill up before other parts

Double hashing

Instead of shifting +1, choose a second hash function

```
jumpnum = hash2(key);
while (HT[index] != NULL)
  index=(index+jumpnum)%TABLESIZE
```

Example hash2 function:

```
hash2(key) = key%SMALLNUMBER + 1;
```

- Reduces clustering