Hashing

What is it?

A form of narcotic intake?

A side order for your eggs?

A combination of the two?
Problem

• RT&T is a large phone company, and they want to provide caller ID capability:
  - given a phone number, return the caller’s name
  - phone numbers are in the range $R=0$ to $10^7-1$
  - want to do this as efficiently as possible ($$$)

• A few suboptimal ways to design this dictionary:
  - an array indexed by key: takes $O(1)$ time, $O(N+R)$ space -- huge amount of wasted space

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(null)</td>
<td>(null)</td>
<td>...</td>
<td>Roberto</td>
<td>...</td>
<td>(null)</td>
</tr>
<tr>
<td>000-0000</td>
<td>000-0001</td>
<td>...</td>
<td>863-7639</td>
<td>...</td>
<td>999-9999</td>
</tr>
</tbody>
</table>

  - a linked list: takes $O(N)$ time, $O(N)$ space

```
863-7639   863-9350
Roberto    Gordon
```

  - a balanced binary tree: $O(lg N)$ time, $O(N)$ space
    (you want fancy pictures here too? so read the slides from the RedBlack help session).
Another Solution

• We can do better, with a *Hashtable* -- O(1) expected time, O(N+M) space, where M is table size

• Like an array, but come up with a function to map the large range into one which we can manage
  - e.g., take the original key, modulo the (relatively small) size of the array, and use that as an index

• Insert (863-7639, Roberto) into a hashed array with, say, five slots
  - \( 8637639 \mod 5 = 4 \), so (863-7639, Roberto) goes in slot 4 of the hash table

• A lookup uses the same process: hash the query key, then check the array at that slot

• Insert (863-9350, Gordon)

• And insert (863-2234, Gordon). Don’t skip this example!
Collision Resolution

• How to deal with two keys which hash to the same spot in the array?

• Use *chaining*
  - Set up an array of links (a *table*), indexed by the keys, to *lists* of items with the same key

• Most efficient (time-wise) collision resolution
  - we’ll talk about others later which use less space
Pseudo-code

• Any dictionary has 3 basic methods, and the constructor:
  init
  insert
  find
  remove

• Init
  create table of M lists

• Insert(K)
  index = h(K)
  insert into table[index]

• Find(K)
  index = h(K)
  walk down list at table[index], looking for a match
  return what was found (or error)

• Remove(K)
  index = h(K)
  walk down list at table[index], looking for a match
  remove what was found (or error)
Hash Functions

• Need to choose a good hash function
  - quick to compute
  - distributes keys uniformly throughout the table

• How to deal with hashing non-integer keys:
  - find some way of turning the keys into integers
    - in our example, remove the hyphen in 863-7639 to get 8637639!
    - for a string, add up the ASCII values of the characters of your string
  - then use a standard hash function on the integers

• Use the remainder
  - \( h(K) = K \mod M \)
  - \( K \) is the key, \( M \) the size of the table

• Need to choose \( M \)

• \( M = b^e \) (bad)
  - if \( M \) is a power of two, \( h(K) \) gives the \( e \) least significant bits of \( K \)
  - all keys with the same ending go to the same place

• \( M \) prime (good)
  - helps ensure uniform distribution
  - take a number theory class to understand why
Hash Functions (cont.)

- **Mid-Square**
  - \( h(K) = \) middle digits of \( K^2 \)

- **I.E. Table size power of 10**
  - \( h(4150130) = 21526 \, 4436 \, 17100 \)
  - \( h(415013034) = 526447 \, 3522 \, 151420 \)
  - \( h(1150130) = 13454 \, 2361 \, 7100 \)

- **I.E. Table power is power of 2**
  - \( h(1001) = 10 \, 100 \, 01 \)
  - \( h(1011) = 11 \, 110 \, 01 \)
  - \( h(1101) = 101 \, 010 \, 01 \)
More on Collisions

• A key is mapped to an already occupied table location
  - what to do?!?

• Use a collision handling technique

• We’ve seen *Chaining*

• Can also use *Open Addressing*
  - Double Hashing
  - Linear Probing

Man, that’s a lot of hash!

Watch out for the legal probe
Linear Probing

- If the current location is used, try the next table location

```plaintext
linear_probing_insert(K)
if (table is full) error

probe = h(K)

while (table[probe] occupied)
    probe = (probe + 1) mod M

    table[probe] = K
```

- Lookups walk along table until the key or an empty slot is found

- Uses less memory than chaining
  - don’t have to store all those links

- Slower than chaining
  - may have to walk along table for a long way

- A real pain to delete from
  - either mark the deleted slot
  - or fill in the slot by shifting some elements down
Linear Probing Example

- $h(K) = K \mod 13$
- Insert keys:

18  41  22  44  59  32  31  73

0  1  2  3  4  5  6  7  8  9  10  11  12
Double Hashing

- Use two hash functions
- If $M$ is prime, eventually will examine every position in the table

```python
def double_hash_insert(K):
    if(table is full) error

    probe = h1(K)
    offset = h2(K)

    while (table[probe] occupied)
        probe = (probe + offset) mod M

    table[probe] = K
```

- Many of same (dis)advantages as linear probing
- Distributes keys more uniformly than linear probing does
Double Hashing Example

- \( h1(K) = K \mod 13 \)
  \( h2(K) = 8 - K \mod 8 \)
  - we want \( h2 \) to be an offset to add

18  41  22  44  59  32  31  73
Theoretical Results

• Let $\alpha = N/M$
  - the load factor: average number of keys per array index

• Analysis is probabilistic, rather than worst-case

**Expected Number of Probes**

<table>
<thead>
<tr>
<th></th>
<th>not found</th>
<th>found</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Chaining</strong></td>
<td>$1 + \alpha$</td>
<td>$1 + \frac{\alpha}{2}$</td>
</tr>
<tr>
<td><strong>Linear Probing</strong></td>
<td>$\frac{1}{2} + \frac{1}{2(1 - \alpha)^2}$</td>
<td>$\frac{1}{2} + \frac{1}{2(1 - \alpha)}$</td>
</tr>
<tr>
<td><strong>Double Hashing</strong></td>
<td>$\frac{1}{(1 - \alpha)}$</td>
<td>$\frac{1}{\alpha \ln(1 - \alpha)}$</td>
</tr>
</tbody>
</table>
Expected Number of Probes vs. Load Factor

Number of Probes vs. Load Factor, showing the expected number of probes for different hash table methods: Linear Probing, Double Hashing, and Chaining. The graph illustrates how the number of probes increases with the load factor $\alpha$. The y-axis represents the number of probes, and the x-axis represents the load factor $\alpha$. Different lines indicate successful and unsuccessful searches for each method.