Announcements (1/29/14)

- HW #3 due now
- HW #4 out today
- Project 2A due Thursday night.
- Reading for this lecture: Chapter 5.

AVL find, insert, delete: $O(\log n)$

Suppose (unique) keys between 0 and 1000.
- Can we do better than $O(\log n)$?

Arrays for Dictionaries

Now suppose keys are first, last names
- how big is the key space?

But keyspace is sparsely populated
- $<10^5$ active students

Hash Tables

- Map keys to a smaller array called a hash table
  - via a hash function $h(K)$
  - Find, insert, delete: $O(1)$ on average!

Simple Integer Hash Functions

- key space $K = \text{integers}$
- TableSize = 10

- $h(K) =$

- Insert: 7, 18, 41, 34
Simple Integer Hash Functions

- key space $K = \text{integers}$
- TableSize = 7
- $h(K) = K \mod 7$
- **Insert**: 7, 18, 41, 34

Aside: Properties of Mod

To keep hashed values within the size of the table, we will generally do:

$h(K) = \text{function}(K) \mod \text{TableSize}$

(In the previous examples, function($K$) = $K$.)

Useful properties of mod:

- $(a + b) \mod c = [(a \mod c) + (b \mod c)] \mod c$
- $(a \times b) \mod c = [(a \mod c) \times (b \mod c)] \mod c$
- $a \mod c = b \mod c \rightarrow (a - b) \mod c = 0$

String Hash Functions?

What's a good hash function for a string?

Some String Hash Functions

key space = strings

$K = s_0, s_1, s_2, \ldots, s_{m-1}$ (where $s_i$ are chars: $s_i \in [0, 128]$)

1. $h(K) = s_0 \mod \text{TableSize}$
2. $h(K) = \left(\sum_{i=0}^{m-1} s_i\right) \mod \text{TableSize}$
3. $h(K) = \left(\sum_{i=0}^{m-1} s_i \times 128^i\right) \mod \text{TableSize}$

Hash Function Desiderata

What are good properties for a hash function?

Designing Hash Functions

Often based on modular hashing:

$h(K) = f(K) \mod P$

$P$ is typically the TableSize

$P$ is often chosen to be prime:

- Reduces likelihood of collisions due to patterns in data
- Is useful for guarantees on certain hashing strategies (as we'll see)

But what would be a more convenient value of $P$?
A Fancier Hash Function

Some experimental results indicate that modular hash functions with prime tables sizes are not ideal.
Lots of better solutions, e.g.,

```java
jenkinsOneAtATimeHash(String key, int keyLength) {
  hash = 0;
  for (i = 0; i < key_len; i++) {
    hash += key[i];
    hash += (hash << 10);
    hash ^= (hash >> 6);
  }
  hash += (hash << 3);
  hash ^= (hash >> 11);
  hash += (hash << 15);
  return hash % TableSize;
}
```

Collision Resolution

**Collision**: when two keys map to the same location in the hash table.
How handle this?

Separate Chaining

Insert:

| 0 | 10 |
| 1 | 22 |
| 2 | 107 |
| 3 | 12 |
| 4 | 42 |

All keys that map to the same hash value are kept in a list (or "bucket").

Analysis of Separate Chaining

The **load factor**, \( \lambda \), of a hash table is

\[
\lambda = \frac{N}{\text{TableSize}}
\]

Average cost of:
- Unsuccessful find?
- Successful find?
- Insert?

Alternative: Use Empty Space in the Table

Insert:

| 0 | 38 |
| 1 | 19 |
| 2 | 8 |
| 3 | 109 |
| 4 | 10 |

Try \( h(K) \).
If full, try \( h(K)+1 \).
If full, try \( h(K)+2 \).
If full, try \( h(K)+3 \).
Etc...
Open Addressing

The approach on the previous slide is an example of **open addressing**:

After a collision, try “next” spot. If there’s another collision, try another, etc.

Finding the next available spot is called **probing**:

- $0^{th}$ probe = $h(k) \% \text{TableSize}$
- $1^{st}$ probe = $(h(k) + f(1)) \% \text{TableSize}$
- $2^{nd}$ probe = $(h(k) + f(2)) \% \text{TableSize}$

... $i^{th}$ probe = $(h(k) + f(i)) \% \text{TableSize}$

$f(i)$ is the probing function. We’ll look at a few...

Linear Probing

$$f(i) = i$$

• Probe sequence:
  - $0^{th}$ probe = $h(K) \% \text{TableSize}$
  - $1^{st}$ probe = $(h(K) + 1) \% \text{TableSize}$
  - $2^{nd}$ probe = $(h(K) + 2) \% \text{TableSize}$
  - ... $i^{th}$ probe = $(h(K) + i) \% \text{TableSize}$

Linear Probing – Clustering

[Diagram showing clustering in linear probing]

Analysis of Linear Probing

- For any $\lambda < 1$, linear probing will find an empty slot
- Expected # of probes (for large table sizes)
  - unsuccessful search:
    $$\frac{1}{2} \left( 1 + \frac{1}{1-\lambda} \right)$$
  - successful search:
    $$\frac{1}{2} \left( 1 + \frac{1}{\lambda - 1} \right)$$
- Linear probing suffers from **primary clustering**
- Performance quickly degrades for $\lambda > 1/2$
Quadratic Probing

Quadratic Probing Example

• Probe sequence:
  0th probe = h(K) % TableSize
  1st probe = (h(K) + 1) % TableSize
  2nd probe = (h(K) + 4) % TableSize
  3rd probe = (h(K) + 9) % TableSize
  . . .
  ith probe = (h(K) + i^2) % TableSize

Another Quadratic Probing Example

TableSize = 7
h(K) = K % 7
insert(76)  76 % 7 = 6
insert(40)  40 % 7 = 5
insert(48)  48 % 7 = 6
insert(5)   5 % 7 = 5
insert(55)  55 % 7 = 6
insert(47)  47 % 7 = 5

Quadratic Probing: Success guarantee for λ < ½

We can prove assertion #2 by contradiction. Suppose that for some i ≠ j, 0 ≤ i, j ≤ T/2, prime T:

(h(K) + i^2) % T ≠ (h(K) + j^2) % T

Quadratic Probing: Properties

• For any λ < ½, quadratic probing will find an empty slot; for bigger λ, quadratic probing may find a slot.

• Quadratic probing does not suffer from primary clustering: keys hashing to the same area is ok

• But what about keys that hash to the same slot? – Secondary Clustering!
Double Hashing

Idea: given two different (good) hash functions $h(K)$ and $g(K)$, it is unlikely for two keys to collide with both of them.

So...let's try probing with a second hash function:

$$f(i) = i \cdot g(K)$$

- Probe sequence:
  - 0th probe = $h(K) \mod \text{TableSize}$
  - 1st probe = $(h(K) + g(K)) \mod \text{TableSize}$
  - 2nd probe = $(h(K) + 2g(K)) \mod \text{TableSize}$
  - 3rd probe = $(h(K) + 3g(K)) \mod \text{TableSize}$
  - ... 
  - $i$th probe = $(h(K) + ig(K)) \mod \text{TableSize}$

Double Hashing Example

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table Size = 7

$h(K) = K \mod 7$
$g(K) = 5 - (K \mod 5)$

- Insert(76) $76 \mod 7 = 6$ and $5 - 76 \mod 5 = 5$
- Insert(93) $93 \mod 7 = 2$ and $5 - 93 \mod 5 = 5$
- Insert(40) $40 \mod 7 = 5$ and $5 - 40 \mod 5 = 5$
- Insert(47) $47 \mod 7 = 5$ and $5 - 47 \mod 5 = 5$
- Insert(10) $10 \mod 7 = 3$ and $5 - 10 \mod 5 = 5$
- Insert(55) $55 \mod 7 = 6$ and $5 - 55 \mod 5 = 5$

Another Example of Double Hashing

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hash Functions:

- $T = \text{TableSize} = 10$
- $h(K) = K \mod T$
- $g(K) = 1 + (K/T) \mod (T-1)$

- Insert these values into the hash table in this order. Resolve any collisions with double hashing:
  - 13
  - 28
  - 33
  - 147
  - 43

Analysis of Double Hashing

- Double hashing is safe for $\lambda < 1$ for this case:
  - $h(k) = k \mod p$
  - $g(k) = q - (k \mod q)$
  - $2 < q < p$, and $p$, $q$ are primes

- Expected # of probes (for large table sizes)
  - unsuccessful search:
    $$\lambda = \frac{1}{1 - \lambda}$$
  - successful search:
    $$\frac{1}{\lambda} \log \left( \frac{1}{1 - \lambda} \right)$$

Deletion in Separate Chaining

How do we delete an element with separate chaining?

Deletion in Open Addressing

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>59</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>76</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$h(k) = k \mod 7$

Linear probing

Deleter(23)
Find(59)
Insert(30)

Need to keep track of deleted items... leave a "marker"
Rehashing

When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - Separate chaining: full ($\lambda = 1$)
  - Open addressing: half full ($\lambda = 0.5$)
  - When an insertion fails
  - Some other threshold
- Cost of a single rehashing?

Amortized Analysis of Rehashing

- Cost of inserting $n$ keys is $< 3n$
- suppose $2^k + 1 \leq n \leq 2^{k+1}$
  - Hashes = $n$
  - Rehashes = $2 + 2^2 + ... + 2^k = 2^{k+1} - 2$
  - Total = $n + 2^{k+1} - 2 < 3n$
- Example
  - $n = 33$, Total = $33 + 64 - 2 = 95 < 99$

Equal objects must hash the same

- The Java library (and your project hash table) make a very important assumption that clients must satisfy...
  - If $c\text{.compare}(a, b) == 0$, then we require $h\text{.hash}(a) == h\text{.hash}(b)$
- If you ever override equals
  - You need to override $hashCode$ also in a consistent way
  - See CoreJava book, Chapter 5 for other "gotchas" with equals

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
  - But what is the cost of doing, e.g., findMin?
- Can use:
  - Separate chaining (easiest)
  - Open hashing (memory conservation, no linked list management)
  - Java uses separate chaining
- Rehashing has good amortized complexity.
- Also has a big data version to minimize disk accesses: extendible hashing. (See book.)

Terminology Alert!

- We (and the book) use the terms
  - "chaining" or "separate chaining"
  - "open addressing"
- Very confusingly
  - "open hashing" is a synonym for "chaining"
  - "closed hashing" is a synonym for "open addressing"