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:

\[ h(K) = \text{function}(K) \% \text{TableSize} \]

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

Useful properties of mod:

- \((a + b) \% c = [(a \% c) + (b \% c)] \% c\)
- \((a \cdot b) \% c = [(a \% c) \cdot (b \% c)] \% c\)
- \(a \% c = b \% c \rightarrow (a - b) \% c = 0\)
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. \[ h(K) = \left( \sum_{i=0}^{m-1} s_i \cdot 128^i \right) \mod \text{TableSize} \]
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

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

Insert:
10
22
107
12
42
Analysis of Separate Chaining

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

$$\lambda = \frac{\text{average # of elems per bucket}}{\text{TableSize}}$$

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>01 / 23 / 4 / 5 / 6 / 7 / 8 / 9 / 10</th>
<th>42 → 12 → 22 /</th>
<th>86 /</th>
<th>12</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis of Separate Chaining

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

$$\lambda = \frac{N}{\text{TableSize}}$$

$\lambda$ = average # of elems per bucket

Average cost of:
- Unsuccessful find?
- Successful find?
- Insert?
Alternative: Use Empty Space in the Table

<table>
<thead>
<tr>
<th></th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>109</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

Try $h(K)$. If full, try $h(K)+1$. If full, try $h(K)+2$. If full, try $h(K)+3$. Etc…

Insert:
- 38
- 19
- 8
- 109
- 10
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 = $h(k) \mod \text{TableSize}$
- $1^{th}$ probe = $(h(k) + f(1)) \mod \text{TableSize}$
- $2^{th}$ probe = $(h(k) + f(2)) \mod \text{TableSize}$
- ...

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

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

\[ f(i) = i \]

- Probe sequence:
  
  \[
  0^{th} \text{ probe} = h(K) \mod \text{TableSize} \\
  1^{th} \text{ probe} = (h(K) + 1) \mod \text{TableSize} \\
  2^{th} \text{ probe} = (h(K) + 2) \mod \text{TableSize} \\
  \ldots \\
  i^{th} \text{ probe} = (h(K) + i) \mod \text{TableSize}
  \]
Linear Probing

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>109</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
<td>19</td>
</tr>
</tbody>
</table>

**Insert:**

38
19
8
109
10

Try \( h(K) \)
If full, try \( h(K)+1 \).
If full, try \( h(K)+2 \).
If full, try \( h(K)+3 \).
Etc…
Linear Probing – Clustering

no collision → collision in small cluster

no collision → collision in large cluster

[R. Sedgewick]
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)^2} \right)
  
  - successful search: $\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)} \right)$

- Linear probing suffers from primary clustering.

- Performance quickly degrades for $\lambda > 1/2$
Quadratic Probing

\[ f(i) = i^2 \]

- Probe sequence:
  - 0\(^{th}\) probe = \( h(K) \mod \text{TableSize} \)
  - 1\(^{st}\) probe = \( (h(K) + 1) \mod \text{TableSize} \)
  - 2\(^{nd}\) probe = \( (h(K) + 4) \mod \text{TableSize} \)
  - 3\(^{rd}\) probe = \( (h(K) + 9) \mod \text{TableSize} \)
  - \( \ldots \)
  - \( i^{th}\) probe = \( (h(K) + i^2) \mod \text{TableSize} \)

Less likely to encounter Primary Clustering
Quadratic Probing Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Insert:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>49 (+1)</td>
<td>89</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>58 (+4)</td>
<td>49</td>
</tr>
<tr>
<td>3</td>
<td>79 (+4)</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>79</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>89</td>
<td></td>
</tr>
</tbody>
</table>

Note: The table and diagram illustrate the insertion process using quadratic probing.
Another Quadratic Probing Example

TableSize = 7

\( h(K) = K \mod 7 \)

- insert(76): \( 76 \mod 7 = 6 \)
- insert(40): \( 40 \mod 7 = 5 \)
- insert(48): \( 48 \mod 7 = 6 \)
- insert(5): \( 5 \mod 7 = 5 \)
- insert(55): \( 55 \mod 7 = 6 \)
- insert(47): \( 47 \mod 7 = 5 \)

\[ i^2 = 1 4 9 16 25 \]
\[ i^2 \mod 7 = 1 4 2 2 4 \]
Quadratic Probing:
Success guarantee for $\lambda < \frac{1}{2}$

Assertion #1: If $T = \text{TableSize}$ is prime and $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in $\leq \frac{T}{2}$ probes.

Assertion #2: For prime $T$ and all $0 \leq i, j \leq \frac{T}{2}$ and $i \neq j$,

$$ (h(K) + i^2) \mod T \neq (h(K) + j^2) \mod T $$

Assertion #3: Assertion #2 proves assertion #1.
Quadratic Probing:
Success guarantee for $\lambda < \frac{1}{2}$

We can prove assertion #2 by contradiction.
Suppose that for some $i \neq j$, $0 \leq i, j \leq T/2$, prime $T$:

$$(h(K) + i^2) \mod T = (h(K) + j^2) \mod T = 0$$

$$
\begin{align*}
(h(K) + i^2 &\neq h(K) - j^2) \mod T \\
i^2 - j^2 &\mod T = 0 \\
(i - j)(i + j) &\mod T = 0 \\
i = j &\quad i + j = T
\end{align*}
$$
Quadratic Probing: Properties

• For any $\lambda < \frac{1}{2}$, quadratic probing will find an empty slot; for bigger $\lambda$, 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:

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

- Probe sequence:
  
  $0^{th}$ probe = $h(K) \mod \text{TableSize}$
  
  $1^{th}$ probe = $(h(K) + g(K)) \mod \text{TableSize}$
  
  $2^{th}$ probe = $(h(K) + 2\times g(K)) \mod \text{TableSize}$
  
  $3^{th}$ probe = $(h(K) + 3\times g(K)) \mod \text{TableSize}$
  
  $\ldots$
  
  $i^{th}$ probe = $(h(K) + i\times g(K)) \mod \text{TableSize}$
### Double Hashing Example

TableSize = 7

- \( h(K) = K \% 7 \)
- \( g(K) = 5 - (K \% 5) \)

<table>
<thead>
<tr>
<th>Index</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>47</td>
</tr>
<tr>
<td>2</td>
<td>93</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>76</td>
</tr>
</tbody>
</table>

- Insert(76) \( 76 \% 7 = 6 \) and \( 5 - 76 \% 5 = 3 \)
- Insert(93) \( 93 \% 7 = 2 \) and \( 5 - 93 \% 5 = 3 \)
- Insert(40) \( 40 \% 7 = 5 \) and \( 5 - 40 \% 5 = 5 \)
- Insert(47) \( 47 \% 7 = 5 \) and \( 5 - 47 \% 5 = 3 \)
- Insert(10) \( 10 \% 7 = 3 \) and \( 5 - 10 \% 5 = 5 \)
- Insert(55) \( 55 \% 7 = 6 \) and \( 5 - 55 \% 5 = 5 \)
Another Example of Double Hashing

Hash Functions:
\[ T = \text{TableSize} = 10 \]
\[ h(K) = K \mod T \]
\[ g(K) = 1 + \left(\frac{K}{T}\right) \mod (T-1) \]

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:
  - $h(k) = k \% p$
  - $g(k) = q - (k \% q)$
  - $2 < q < p$, and $p, q$ are primes

- Expected # of probes (for large table sizes)
  - unsuccessful search:
    $$\frac{1}{1 - \lambda}$$
  - successful search:
    $$\frac{1}{\lambda} \log_e \left( \frac{1}{1 - \lambda} \right)$$
Deletion in Separate Chaining

How do we delete an element with separate chaining?
Deletion in Open Addressing

h(k) = k % 7
Linear probing

Delete(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?**
Rehashing Picture

• Starting with table of size 2, double when load factor > 1.
Amortized Analysis of Rehashing

- Cost of inserting $n$ keys is $< 3n$
- Suppose $2^k + 1 \leq n \leq 2^{k+1}$
  - Hashes = $n$
  - Rehashes = $2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2$
  - 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…
  
  \[
  \text{If} \; \text{c.compare}(a,b) \; \text{==} \; 0, \text{then we require} \\
  h.\text{hash}(a) \; \text{==} \; h.\text{hash}(b)
  \]

• 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”