Announcements (2/11/09)

- Project 2B due today.
- Homework #4 due on Friday, start of class
- Project #3 assigned on Friday
  - Partner signups NOW - midnight Friday
- Section: project warm-up, midterms returned, …
- Reading for this lecture: Chapter 5.

Hash Tables

- Find, insert, delete: constant time on average!
- A **hash table** is an array of some fixed size.
- General idea:

  ```
  TableSize – 1
  0
  index = h(K)
  hash table
  ...
  key space (e.g., integers, strings)
  ```

Key space of size $M$, but we only want to store subset of size $N$, where $N << M$.

- Keys are identifiers in programs. Compiler keeps track of them in a symbol table.
- Keys are student names. We want to look up student records quickly by name.
- Keys are chess configurations in a chess playing program.
- Keys are URLs in a database of web pages.
Simple Integer Hash Functions

- key space = integers
- TableSize = 10
- \( h(K) = \)
- **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, \( \text{function}(K) = K \).)

It’s worth noting a couple properties of the mod function:
- \((a + b) \% c = [(a \% c) + (b \% c)] \% c\)
- \((a \cdot b) \% c = [(a \% c) \cdot (b \% c)] \% c\)
- \(a \% c = b \% c \implies (a - b) \% c = 0\)

String Hash Functions?

key space = strings

\( K = s_0 \ s_1 \ s_2 \ldots \ s_{m-1} \) (where \( s_i \) are chars: \( s_i \in [0, 128] \))
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 some desirable properties for a hash function?

Designing Hash Functions

We’ve seen a few possibilities. The simplest is **modular hashing**:

\[ h(K) = K \mod P \]

where \( P \) is usually just 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.

Instead, we can work on designing a really good hash function:

```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 can we cope with collisions?

---

Separate Chaining

- **Separate Chaining**: 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

$$\lambda = \frac{N}{\text{TableSize}} \leftarrow \text{no. of elements}$$

Separate chaining: $\lambda =$ average # of elems per bucket

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

---

Alternative: Use Empty Space in the Table

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**:
- 0th probe = \( h(k) \mod \text{TableSize} \)
- 1st probe = \( (h(k) + f(1)) \mod \text{TableSize} \)
- 2nd 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…

Terminology Alert!

- **Separate chaining** is sometimes called **open hashing**.
- **Open addressing** is sometimes called **closed hashing**.

Open Addressing Example, Revisited

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

**What is \( f(i) \)?**

Linear Probing

\( f(i) = i \)

- Probe sequence:
  - 0th probe = \( h(K) \mod \text{TableSize} \)
  - 1st probe = \( (h(K) + 1) \mod \text{TableSize} \)
  - 2nd probe = \( (h(K) + 2) \mod \text{TableSize} \)
  - Etc…
  - \( i^{th} \) probe = \( (h(K) + i) \mod \text{TableSize} \)
Linear Probing – Clustering

- 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 $\lambda > 1/2$

Analysis of Linear Probing

Quadratic Probing

$$f(i) = i^2$$

- Probe sequence:
  0th probe = $h(K) \% \text{TableSize}$
  1th probe = $(h(K) + 1) \% \text{TableSize}$
  2th probe = $(h(K) + 4) \% \text{TableSize}$
  3th probe = $(h(K) + 9) \% \text{TableSize}$
  \ldots
  ith probe = $(h(K) + i^2) \% \text{TableSize}$

Less likely to encounter Primary Clustering
Quadratic Probing Example

Insert:
0
89
18
49
58
79

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 $\lambda < \frac{1}{2}$

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

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

\[
(h(K) + i^2) \% T \neq (h(K) + j^2) \% T
\]

 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:

\[
(h(K) + i^2) \% T = (h(K) + j^2) \% T
\]
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 are not bad.

• But what about keys that hash to the same spot? — 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 = $h(K) \% \text{TableSize}$
  - $1^{st}$ probe = $(h(K) + g(K)) \% \text{TableSize}$
  - $2^{nd}$ probe = $(h(K) + 2\times g(K)) \% \text{TableSize}$
  - $3^{rd}$ probe = $(h(K) + 3\times g(K)) \% \text{TableSize}$
  - ...  
  - $i^{th}$ probe = $(h(K) + i\times g(K)) \% \text{TableSize}$

Another Example of Double Hashing

<table>
<thead>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

Hash Functions:
- $T = \text{TableSize} = 10$
- $h(K) = K \% T$
- $g(K) = 1 + (K/T) \% (T-1)$

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 at least one 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: $\frac{1}{\lambda} \log_\lambda \left( \frac{1}{1-\lambda} \right)$

Deletion in Separate Chaining

How do we delete an element with separate chaining?

Deletion in Open Addressing

Can we do something similar for open addressing?

- Delete
- Find
- Insert

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

Rehashing

**Idea:** 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 Example

• Separate chaining example:
  \( h_1(x) = x \mod 5 \) rehashes to \( h_2(x) = x \mod 11 \).

\[
\begin{array}{cccccc}
\text{λ = 1} & 0 & 1 & 2 & 3 & 4 \\
25 & 37 & 83 & 52 & 98
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
\text{λ = 5/11} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
25 & 37 & 83 & 52 & 98
\end{array}
\]

Rehashing Picture

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

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 textbook.)

Amortized Analysis of Rehashing

• Cost of inserting \( n \) keys is < 3\( n \)
• \( 2^k + 1 \leq n \leq 2^{k+1} \)
  – Hashes = \( n \)
  – Rehashes = \( 2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2 \)
  – Total = \( n + 2^{k+1} - 2 < 3n \)
• Example
  – \( n = 33 \), Total = \( 33 + 64 - 2 = 95 < 99 \)
Java hashCode() Method

- Class Object defines a hashCode method
  - Intent: returns a suitable hashcode for the object
  - Result is arbitrary int; must scale to fit a hash table (e.g. obj.hashCode() % nBuckets)
  - Used by collection classes like HashMap
- Classes should override with calculation appropriate for instances of the class
  - Calculation should involve semantically “significant” fields of objects

hashCode() and equals()

- To work right, particularly with collection classes like HashMap, hashCode() and equals() must obey this rule:
  if a.equals(b) then it must be true that a.hashCode() == b.hashCode()
  - Why?
- Reverse is not required