Hashing

CSE 326
Data Structures
Lecture 15

Hashing

• Hashing is a family of data structures used to efficiently support insert, delete, find.
• It cannot be used efficiently for other operations where the order of data is important. No list-all, range queries, successor, predecessor.

Hashing Properties

• Load Factor $\lambda = \frac{N}{\text{HSize}}$
  – Hash tables may have unused entries $\lambda < 1$
• Good quality hash function distribute data as evenly as possible over the keys.
• Collisions: $h($inserted key$) = h($existing key$)$.
  – Open hashing - linked lists
  – Closed hashing - find a new place to put inserted key

Readings and References

• Reading
  – Chapter 5

Simple Hash Table

<table>
<thead>
<tr>
<th>T</th>
<th>Hash function:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$h: U \rightarrow {0, 1, \ldots, \text{Hsize} - 1}$</td>
</tr>
<tr>
<td>2</td>
<td>$U$ is the universe of keys</td>
</tr>
<tr>
<td>3</td>
<td>$h($&quot;name&quot;$)$ is the hash value of &quot;name&quot;</td>
</tr>
<tr>
<td>4</td>
<td>$h($&quot;Judy Jones&quot;$) = 4$</td>
</tr>
<tr>
<td>5</td>
<td>$h($&quot;Jerry Lee&quot;$) = 7$</td>
</tr>
<tr>
<td>6</td>
<td>Find(&quot;name&quot;) = T[h(&quot;name&quot;)]</td>
</tr>
</tbody>
</table>
Good Hash Functions

- Integers: Division method
  - Choose Hsize to be a prime
  - Example. Hsize = 23, h(50) = 4, h(1257) = 15

- Character Strings
  - $x = a_0a_1a_2...a_m$ is a character string. Define $int(x) = a_0 + a_1128 + a_2128^2 + ... + a_m128^{m-1}$
  - Compute $h(x) = int(x) \mod Hsize$
  - Compute $h(x)$ using Horner’s Rule
    
    ```
    h := 0
    for i = m to 0 by -1 do h := (a_i + 128h) \mod Hsize
    return h
    ```

A Bad Hash Function

- Keys able1, able2, able3, able4
  - Hsize = 128
  - $int(ablex) \mod 128 = int(a) = 97$
  - $h(ablex) = h(abley)$ for all $x$ and $y$

- Why use primes for hash table sizes?
  - Primes have no nontrivial divisors
  - Numbers relatively prime to 128 will also work for character strings

Multiplication Method

- Hash function defined by HSize and a floating point number $A$.
  - Integer case
  - $h(k) = \lfloor HSize \times (k \times A \mod 1) \rfloor$
  - Example: HSize = 10, $A = 0.485$
    
    ```
    h(50) = \lfloor 10 \times (50 \times 0.485 \mod 1) \rfloor
    = \lfloor 10 \times (24.25 \mod 1) \rfloor
    = \lfloor 10 \times 0.25 \rfloor
    = 2
    ```
  - HSize need not be prime
  - More computation than division method
  - Another alternative – Universal Hashing

What about Collisions?

- Open Hashing - Collisions overflow into linked lists.
  - Load factors > 1 are possible
- Closed Hashing - find another place in the hash table for the entry.
  - Load factor must be $\leq 1$

Open Hashing (Chaining)

- $h(a) = h(b)$ and $h(d) = h(g)$
  - Chains may be ordered or unordered. Little advantage to ordering.
- $h(a) \neq h(b)$ and $h(d) \neq h(g)$
- $h(a) \neq h(b)$ and $h(d) = h(g)$
- $h(a) = h(b)$ and $h(d) \neq h(g)$

Open Hashing Properties

- Load factor = $\lambda$
  - Unsuccessful searches cost $\lambda$ comparisons on average
  - Successful searches cost $1 + \lambda/2$ comparisons on average
- Comparisons can be expensive so choosing $\lambda$ between 1/2 and 1 is wise.
Closed Hashing (Open Addressing)

• No chaining, every key fits in the hash table.
• Probe sequence
  – \( h(k) \)
  – \( (h(k) + f(1)) \mod HSize \)
  – \( (h(k) + f(2)) \mod HSize, \ldots \)
• Insertion: Find the first probe with an empty slot.
• Find: Find the first probe that equals the query or is empty. Stop at HSize probe, in any case.
• Deletion: lazy deletion is needed. That is, mark locations as deleted, if a deleted key resides there.

Linear Probing

• \( f(i) = i \)
• Probe sequence
  \( h(k) \)
  \( (h(k) + 1) \mod HSize \)
  \( (h(k) + 2) \mod HSize, \ldots \)
• Insertion (assuming \( \lambda < 1 \))
  \( h := h(k) \)
  while \( T(h) \) not empty do
  \( h := (h + i) \mod HSize; \)
  insert \( k \) in \( T(h) \)

Performance of Linear Probing

• If there is an available slot linear probing will find it.
• For large hash tables the expected number of probes on insertion is:
  \[
  \frac{1}{2} \left( \frac{1}{1 - \lambda^2} \right) - 1
  \]
• The expected number of probes on successful searches is:
  \[
  \frac{1}{2} \left( \frac{1}{1 - \lambda^2} \right)
  \]
• Linear probing suffers from primary clustering.
• Not a good idea to use linear probing with \( \lambda > \frac{1}{2} \).
• Lazy deletion needed.

Linear Probing Example

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

Quadratic Probing

• \( f(i) = \lambda \)
• Probe sequence
  \( h(k) \)
  \( (h(k) + 1) \mod HSize \)
  \( (h(k) + 4) \mod HSize, \ldots \)
• Insertion (assuming \( \lambda < 1/2 \))
  \( h := h(k) \)
  while \( T(h) \) not empty do
  \( h := (h + 2i + 1) \mod HSize; \)
  insert \( k \) in \( T(h) \)
  \( i := i + 1 \)

Note: \( (i + 1)^2 - 1 = 2i + 1 \)
Quadratic Probing Works for $\lambda < 1/2$

- If HSize is prime then $(h(x) + i^2) \mod \text{HSize} \neq (h(x) + j^2) \mod \text{HSize}$ for $i \neq j$ and $0 \leq i,j < \text{HSize}/2$.

- Proof

\[
(h(x) + i^2) \mod \text{HSize} = (h(x) + j^2) \mod \text{HSize} \\
(h(x) + i^2 - j^2) \mod \text{HSize} = 0 \\
(i-j)(i+j) \mod \text{HSize} = 0
\]

$\Rightarrow \text{HSize does not divide (i-j) or (i+j)}$

Quadratic Probing may Fail if $\lambda > 1/2$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$51 \mod 7$</th>
<th>2</th>
<th>$i = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>93</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>40</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>76</td>
<td>6</td>
</tr>
</tbody>
</table>

Performance of Quadratic Probing

- Although quadratic probing can fail for $\lambda > 1/2$, it is not likely to do so. We can use load factors greater than 1/2, but load factors close to 1 should be avoided.
- Quadratic hashing does not suffer from primary clustering, but has only minor secondary clustering.
- With load factors near 1/2 the expected number of probes per successful search is about 1.5.
- Lazy deletion must be used.

Double Hashing

- $f(i) = i \ g(k)$ where $g$ is a second hash function
- Probe sequence

\[
h(k) \mod \text{HSize} \\
(h(k) + 2g(k)) \mod \text{HSize} \\
(h(k) + 3g(k)) \mod \text{HSize}, \ldots
\]

- In choosing $g$ care must be taken so that it never evaluates to 0.
- A good choice for $g$ is to choose a prime $R < \text{HSize}$ and let $g(k) = R - (k \mod R)$.

Double Hashing Example

- Let $h(k) \mod p$ and $g(k) = q - (k \mod q)$ where $2 < q < p$ and $p$ and $q$ are primes. The probe sequence $h(k) + ig(k) \mod p$ probes every entry of the hash table.

Double Hashing is Safe for $\lambda < 1$

- Let $h(k) \mod p$ and $g(k) = q - (k \mod q)$ where $2 < q < p$ and $p$ and $q$ are primes. The probe sequence $h(k) + ig(k) \mod p$ probes every entry of the hash table.

Let $0 \leq m < p$, $h = h(k)$, and $g = g(k)$. We show that $h+ig \mod p = m$ for some $i$. 0 < $q < p$, so $q$ and $p$ are relatively prime. By extended Euclid's algorithm that are $s$ and $t$ such that

\[
sq + tp = 1.\]

Choose $i = (m-h)s \mod p$

\[
(h + iq) \mod p = \\
(h + (m-h)sg) \mod p = \\
(h + (m-h)(g + (m-h)p)) \mod p = \\
(h + (m-h)(g + tp)) \mod p = \\
(h + (m-h)(g + m)) \mod p = m
\]
Deletion in Hashing

- Open hashing (chaining) — no problem
- Closed hashing — must do lazy deletion. Deleted keys are marked as deleted.
  - Find: done normally
  - Insert: treat marked slot as an empty slot and fill it.

Rehashing

- Build a bigger hash table of approximately twice the size when \( \lambda \) exceeds a particular value
  - Go through old hash table, ignoring items marked deleted
  - Recompute hash value for each non-deleted key and put the item in new position in new table
  - Cannot just copy data from old table because the bigger table has a new hash function
- Running time is \( O(N) \) but happens very infrequently
  - Not good for real-time safety critical applications

Rehashing Example

- Open hashing — \( h_1(x) = x \mod 5 \) rehashes to \( h_2(x) = x \mod 11 \).
  - \( \lambda = 1 \)
  - \( \lambda = 5/11 \)

Rehashing Picture

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

Amortized Analysis of Rehashing

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

Case Study

- Spelling Dictionary - 30,000 words
- Goals
  - Fast spell checking
  - Minimal storage
- Possible solutions
  - Sorted array and binary search
  - Open hashing (chaining)
  - Closed hashing with linear probing
- Notes
  - Almost all searches are successful
  - 30,000 word average 8 bytes per word, 240,000 bytes
  - Pointers are 4 bytes
Storage

- Assume words are stored as strings and entries in the arrays are pointers to the strings.

Binary search

- \( N \) pointers

Open hashing

- \( N + N/\lambda + 2N \) pointers

Closed hashing

- \( N/\lambda \) pointers

Analysis

- Binary Search
  - Storage = \( N \) pointers + words = 360,000 bytes
  - Time = \( \log_2 N \leq 15 \) probes in worst case

- Open hashing
  - Storage = \( 2N + N/\lambda \) pointers + words
  - \( \lambda = 1 \) implies 600,000 bytes
  - Time = 1 + \( \lambda/2 \) probes per access
  - \( \lambda = 1 \) implies 1.5 probes per access

- Closed hashing
  - Storage = \( N/\lambda \) pointers + words
  - \( \lambda = 1/2 \) implies 480,000 bytes
  - Time = \( (1/2)(1+1/(1-\lambda)) \) probes per access
  - \( \lambda = 1/2 \) implies 1.5 probes per access

Extendible Hashing

- Extendible hashing is a technique for storing large data sets that do not fit in memory.
- An alternative to B-trees

In memory

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>0001</td>
<td>0011</td>
<td>0010</td>
<td>0110</td>
<td>0100</td>
<td>0101</td>
<td>1010</td>
<td>1011</td>
</tr>
</tbody>
</table>

Pages

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0011</td>
<td>0010</td>
<td>0110</td>
</tr>
<tr>
<td>0100</td>
<td>0101</td>
<td>1010</td>
<td>1011</td>
</tr>
</tbody>
</table>

Rehashing

- On deletion neighbors can be merged.
- If table uses \( k \) bits but all pages use \( k-1 \) bits then rehashing to a smaller table can be done. Not normally an issue with large databases.
- Rehashing does not touch pages.
- Splitting and merging touch only two pages.
**Fingerprints**

- Given a string $x$ we want a fingerprint $x'$ with the properties.
  - $x'$ is short, say 128 bits
  - Given $x \neq y$ the probability that $x' = y'$ is infinitesimal (almost zero)
  - Computing $x'$ is very fast
- MD5 - Message Digest Algorithm 5 is a recognized standard
- Applications in databases and cryptography

**Fingerprint Math**

Given 128 bits and $N$ strings what is the probability that the fingerprints of two strings coincide?

$$1 - \frac{2^{128} (2^{128} - 1) \cdots (2^{128} - N + 1)}{(2^{128})^N}$$

This is essentially zero for $N < 2^{35}$.

**Hashing Summary**

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
- Dynamic hash tables have good amortized complexity.
- Extendible hashing is useful in databases.
- Fingerprints good for databases and crypto.