**Collision Resolution**

**Collision**: when two keys map to the same location in the hash table.

Two ways to resolve collisions:
1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)

---

**Separate Chaining**

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

---

**Open Hashing (Chaining)**

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

---

**Analysis of find**

- Defn: The load factor, \( \lambda \), of a hash table is the ratio: \( \frac{N}{M} \) → no. of elements → table size

For separate chaining, \( \lambda = \) average # of elements in a bucket

- Unsuccessful find cost:
- Successful find cost:

---

**Closed Hashing (Open Addressing)**

- No chaining, every key fits in the hash table.
- Probe sequence
  - \( h(k) \)
  - \( (h(k) + f(1)) \mod \text{HSIZE} \)
  - \( (h(k) + f(2)) \mod \text{HSIZE} \), ...
- 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.

---
Open Addressing

<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>

**Insert:**
- 38
- 19
- 8
- 109
- 10

- **Linear Probing:**
  - after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Linear Probing

$f(i) = i$

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

Terminology Alert!

“**Open** Hashing” equals “Closed Hashing” equals “Separate Chaining” “**Open** Addressing”

Write pseudocode for find($k$) for Open Addressing with linear probing

- Find($k$) returns $i$ where $T(i) = k$

Linear Probing Example

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

Probes: 1 1 1 3 1 3

Linear Probing – Clustering

[Image of clustering examples]

Student Activity:
Load Factor in Linear Probing

- For any \( \lambda < 1 \), linear probing will find an empty slot.
- Expected \# of probes (for large table sizes):
  - successful search: \( \frac{1}{2} \left( 1 + \frac{1}{1-\lambda} \right) \)
  - unsuccessful search: \( \frac{1}{2} \left( 1 + \frac{1}{\lambda} \right) \)
- 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 = \( (h(k) + 9) \mod \text{TableSize} \)
  . . .
  \( i^{th} \) probe = \( (h(k) + i^2) \mod \text{TableSize} \)

Exercise: Quadratic Probing

<table>
<thead>
<tr>
<th></th>
<th>Insert:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>89</td>
</tr>
<tr>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>49</td>
</tr>
<tr>
<td>3</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td>79</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></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Quadratic Probing Example

<table>
<thead>
<tr>
<th></th>
<th>insert(76)</th>
<th>insert(40)</th>
<th>insert(48)</th>
<th>insert(5)</th>
<th>insert(55)</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>40%7 = 5</td>
<td>48%7 = 6</td>
<td>5%7 = 5</td>
<td>55%7 = 6</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

But...

| 47%7 = 5 |

Quadratic Probing may Fail if \( \lambda > 1/2 \)

- If size is prime and \( \lambda < 1/2 \), then quadratic probing will find an empty slot in size/2 probes or fewer.
  - show for all \( 0 \leq i, j < \text{size}/2 \) and \( i \neq j \):
    \[(h(x) + i^2) \mod \text{size} \neq (h(x) + j^2) \mod \text{size}\]
  - by contradiction: suppose that for some \( i, j \):
    \[(h(x) + i^2) \mod \text{size} = (h(x) + j^2) \mod \text{size}\]
    \[i^2 \mod \text{size} = j^2 \mod \text{size}\]
    \[(i + j)(i - j) \mod \text{size} = 0\]
    BUT size does not divide \((i-j)\) or \((i+j)\).

Quadratic Probing: Success guarantee for \( \lambda < 1/2 \)

- Exercise:
  - Show for all \( 0 \leq i, j < \text{size}/2 \) and \( i \neq j \):
    \[(h(x) + i^2) \mod \text{size} \neq (h(x) + j^2) \mod \text{size}\]
  - By contradiction: suppose that for some \( i, j \):
    \[(h(x) + i^2) \mod \text{size} = (h(x) + j^2) \mod \text{size}\]
    \[i^2 \mod \text{size} = j^2 \mod \text{size}\]
    \[(i + j)(i - j) \mod \text{size} = 0\]
    BUT size does not divide \((i-j)\) or \((i+j)\).

Less likely to encounter Primary Clustering
Quadratic Probing: Properties

- For any \( \lambda < 1/2 \), 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

\[ f(i) = i \cdot g(k) \]
where \( g \) is a second hash function

- Probe sequence:
  - 0th probe = \( h(k) \mod \) TableSize
  - 1st probe = \( (h(k) + g(k)) \mod \) TableSize
  - 2nd probe = \( (h(k) + 2 \cdot g(k)) \mod \) TableSize
  - 3rd probe = \( (h(k) + 3 \cdot g(k)) \mod \) TableSize
  - ...-
  - ith probe = \( (h(k) + i \cdot g(k)) \mod \) TableSize

Double Hashing Example

\( h(k) = k \mod 7 \) and \( g(k) = 5 - (k \mod 5) \)

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

Probes 1: 1 1 2 1 2

Resolving Collisions with Double Hashing

<table>
<thead>
<tr>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

- Insert 13
- Insert 28
- Insert 33
- Insert 147
- Insert 43

Double Hashing is Safe for \( \lambda < 1 \)

- Let \( h(k) = k \mod p \) and \( g(k) = q - (k \mod q) \) where \( 2 < q < p \) and \( p \) and \( q \) are primes. The probe sequence \( h(k) + g(k) \mod p \) probes every entry of the hash table. Let \( 0 < m < p, h = h(k), \) and \( g = g(k). \) We show that \( h+g \mod p = m \) if some \( i \), \( 0 < i < p \), so \( g \) and \( p \) are relatively prime. By extended Euclid's algorithm that are \( s \) and \( t \) such that \( sg + tp = 1. \) Choose \( i = (m-h)ts \mod p \)
  - \( (h + g) \mod p = m \)
  - \( (h + (m-h)sg + (m-h)tp) \mod p = m \)
  - \( (h + (m-h)(sg + tp)) \mod p = m \)
  - \( (h + (m-h)) \mod p = m \mod p = m \)

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.
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?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

Rehashing Example

- Open hashing $- h_1(x) = x \mod 5$ rehashes to $h_2(x) = x \mod 11$.

Rehashing Picture

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

Amortized Analysis of Rehashing

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

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 word are stored as strings and entries in the arrays are pointers to the strings.

Binary search

Open hashing

Closed hashing

N pointers

N/$\lambda$ + 2N pointers

N/$\lambda$ pointers
Analysis

- Binary Search
  - Storage = N pointers + words = 360,000 bytes
  - Time = \( \log_2 N \) < 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
  - \( \lambda = 1/2 \) 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

Splittings

00 001 010 011 100 101 110 111

Insert 11000

Rehashing

00 001 010 101 100 110 111 111

Insert 00111

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?

\[
1 - \frac{2^{128}(2^{128} - 1) L}{(2^{128})^N} \frac{(2^{128} - N + 1)}{(2^{128})^N}
\]

This is essentially zero for N < 2^{40}. 

3 bits of hash value used

Pages

In memory
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.