CSE 326 Data Structures

Dave Bacon

Hashing
Logistics

- Project 26 due at 11:59 pm.
  - Turn in everything.
    - May 20
    - Use zip (no rar)
    - Office 4-5 David O02
- Homework 4 due Friday
- Midterm in section Thur.
  - Given back
# Implementations So Far

<table>
<thead>
<tr>
<th></th>
<th>Unsorted linked list</th>
<th>Sorted Array</th>
<th>BST</th>
<th>AVL</th>
<th>Splay (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>Find</strong></td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

*Worst Case*
Hash Tables

- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:

  ![hash function](image)

  - key space (e.g., integers, strings)
  - TableSize – 1
Example

- key space = integers
- TableSize = 10
- \( h(K) = K \mod 10 \)

- Insert: 7, 18, 41, 94
  - \( 7 \mod 10 = 7 \)
  - \( 18 \mod 10 = 8 \)
  - \( 41 \mod 10 = 1 \)

- Insert 38?
  - \( 38 \mod 10 = 8 \)

- Find (28) → 8?
Another Example

- key space = integers
- TableSize = 6

- \( h(K) = K \mod 6 \)

- Insert: 7, 18, 41, 34

\[
\begin{align*}
7 \mod 6 &= 1 \\
18 \mod 6 &= 0 \\
41 \mod 6 &= 5 \\
34 \mod 6 &= 4
\end{align*}
\]

What do we want from Hash function?
Hash Functions

1. simple/fast to compute,
2. Avoid collisions
3. have keys distributed evenly among cells.

Perfect Hash function:
- No collisions
- No blank locations.

"Ignorance about data"
26 letters 10 digits + 1 "space" = 37

Sample Hash Functions:

- key space = strings
- \( s = s_0 s_1 s_2 \ldots s_{k-1} \)

1. \( h(s) = s_0 \mod \text{TableSize} \)

2. \( h(s) = \left( \sum_{i=0}^{k-1} s_i \right) \mod \text{TableSize} \)

3. \( h(s) = \left( \sum_{i=0}^{k-1} s_i \cdot 37^i \right) \mod \text{TableSize} \)

Table Size = 100

- Spread: 37 0,...,36
- Dumb: 4 collisions
- Dumb: 4 collisions
- Dumb: 4 collisions
- Dumb: 4 collisions

- Spread: 37 0,...,36
- Spread: 37 0,...,36
- Spread: 37 0,...,36
- Spread: 37 0,...,36
- Spread: 37 0,...,36

- Spot, post, h 37k
- Spot, post, h 37k
- Spot, post, h 37k
- Spot, post, h 37k
- Spot, post, h 37k

- Anagram problem 3
- Anagram problem 3
- Anagram problem 3
- Anagram problem 3
- Anagram problem 3
Designing a Hash Function for web URLs

\[ s = s_0 \ s_1 \ s_2 \ \ldots \ s_{k-1} \]

Issues to take into account:
→ huge urls.
→ “www” – “com” – “edu”
→ “,” “?”

simple + fast \[ \text{as much as possible} \]

\[ h(s) = \]

spread large common keys \[ \text{same location} \]
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”).

Find 42

\[ h(42) = 2 \]

\[ 10 \]

\[ 22 \rightarrow 12 \rightarrow 42 \]

\[ 107 \]

Insert:

- 10
- 22
- 107
- 12
- 42
Analysis of find

- Defn: The load factor, $\lambda$, of a hash table is the ratio: $\lambda = \frac{N}{M} \leftarrow \text{no. of elements}$

For separate chaining, $\lambda = \text{average # of elements in a bucket}$

- unsuccessful: $\lambda \leftarrow \text{Avg}$

- successful: $1 + \frac{1}{2}$
How big should the hash table be?

- For Separate Chaining:

\[
\sqrt{N} = \frac{1}{\varepsilon} \quad \text{want}
\]

\[
N = \frac{1}{2} \quad \text{wasting space.}
\]
**tableSize: Why Prime?**

- Suppose
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - `tableSize = 10`
    - data hashes to 0, 3, 0, 5, 1, 0, 0
  - `tableSize = 11`
    - data hashes to 10, 9, 5, 0, 2, 9, 7

Real-life data tends to have a pattern

Being a multiple of 11 is usually *not* the pattern 😊

↑ collision
Open Addressing

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.

Find 8
Terminology Alert!

"Open Hashing" equals "Separate Chaining"

"Closed Hashing" equals "Open Addressing"

\[ h(K) + f(i) \]
Linear Probing

\[ f(i) = i \]

- Probe sequence:
  
  0\textsuperscript{th} probe = \( h(k) \mod \text{TableSize} \)
  
  1\textsuperscript{th} probe = \( (h(k) + 1) \mod \text{TableSize} \)
  
  2\textsuperscript{th} probe = \( (h(k) + 2) \mod \text{TableSize} \)
  
  \ldots
  
  i\textsuperscript{th} probe = \( (h(k) + i) \mod \text{TableSize} \)
Linear Probing – Clustering

[R. Sedgewick]

When are we OK?
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: $\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right)$

• Linear probing suffers from primary clustering
• Performance quickly degrades for $\lambda > 1/2$
Quadratic Probing

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

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

Insert:
89
18
49
58
79
Quadratic Probing Example

insert(76)  insert(40)  insert(48)  insert(5)  insert(55)
76%7 = 6    40%7 = 5    48%7 = 6    5%7 = 5    55%7 = 6

But... insert(47)
47%7 = 5
Quadratic Probing: Success guarantee for $\lambda < \frac{1}{2}$

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

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

- Probe sequence:
  
  \[
  \begin{align*}
  0^{th} \text{ probe} &= h(k) \mod \text{TableSize} \\
  1^{th} \text{ probe} &= (h(k) + g(k)) \mod \text{TableSize} \\
  2^{th} \text{ probe} &= (h(k) + 2 \times g(k)) \mod \text{TableSize} \\
  3^{th} \text{ probe} &= (h(k) + 3 \times g(k)) \mod \text{TableSize} \\
  \vdots \\
  i^{th} \text{ probe} &= (h(k) + i \times g(k)) \mod \text{TableSize}
  \end{align*}
  \]
Double Hashing Example

\[ h(k) = k \mod 7 \text{ and } g(k) = 5 - (k \mod 5) \]

<table>
<thead>
<tr>
<th>Key</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>1</td>
</tr>
<tr>
<td>93</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>55</td>
<td>2</td>
</tr>
</tbody>
</table>

Here, we see how keys are placed and probed into a hash table using the given hash functions. Each row represents a probe step.
Resolving Collisions with Double Hashing

Hash Functions:

\[ H(K) = K \mod M \]
\[ H_2(K) = 1 + ((K/M) \mod (M-1)) \]

\[ M = \]

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

13
28
33
147
43
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?
  – half full ($\lambda = 0.5$)
  – when an insertion fails
  – some other threshold

• Cost of rehashing?
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.