Dictionary 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>Insert</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Find</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hash Tables

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

General idea:

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

```
TableSize - 1
```

```
hash function: h(K)
```

```
hash table
```

Example

- key space = integers
- TableSize = 10
  - $h(K) = K \mod 10$
  - **Insert**: 7, 18, 41, 94
Another Example

- key space = integers
- TableSize = 6
- \( h(K) = K \mod 6 \)
- Insert: 7, 18, 41, 34

Hash Functions

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

Perfect Hash function:

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

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”).

<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:
- 10
- 22
- 107
- 12
- 42

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:

- Successful find:

How big should the hash table be?

- For Separate Chaining:

\[ \text{tableSize: Why Prime?} \]

- 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, 2, 7

Real-life data tends to have a pattern

Being a multiple of 11 is usually not the pattern ☹
Open Addressing

• **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.

Terminology Alert!

“Open Hashing” equals “Closed Hashing”

“Separate Chaining” “Open Addressing”

Linear Probing

f(i) = i

• Probe sequence:
  0th probe = h(k) mod TableSize
  1st probe = (h(k) + 1) mod TableSize
  2nd probe = (h(k) + 2) mod TableSize
  ...
  ith probe = (h(k) + i) mod TableSize

Linear Probing – Clustering

[R. Sedgewick]
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:
  0th probe = $h(k) \mod \text{TableSize}$
  1st probe = $(h(k) + 1) \mod \text{TableSize}$
  2nd probe = $(h(k) + 4) \mod \text{TableSize}$
  3rd probe = $(h(k) + 9) \mod \text{TableSize}$
  \ldots
  $i$th probe = $(h(k) + i^2) \mod \text{TableSize}$

Quadratic Probing Example

Insert:
- Insert 89
- Insert 18
- Insert 49
- Insert 58
- Insert 79

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!

Quadratic Probing Works for $\lambda < \frac{1}{2}$

- 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
  \[(h(x) + i^2) \mod \text{HSize} = (h(x) + j^2) \mod \text{HSize}\]
  \[(h(x) + i^2) - (h(x) + j^2) \mod \text{HSize} = 0\]
  \[i^2 - j^2 \mod \text{HSize} = 0\]
  \[(i-j)(i+j) \mod \text{HSize} = 0\]
  \[\Rightarrow \text{HSize does not divide } (i-j) \text{ or } (i+j)\]

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

- Probe sequence:
  \[0^{\text{th}} \text{ probe} = h(k) \mod \text{TableSize}\]
  \[1^{\text{st}} \text{ probe} = (h(k) + g(k)) \mod \text{TableSize}\]
  \[2^{\text{nd}} \text{ probe} = (h(k) + 2g(k)) \mod \text{TableSize}\]
  \[3^{\text{rd}} \text{ probe} = (h(k) + 3g(k)) \mod \text{TableSize}\]
  \[\ldots\]
  \[i^{\text{th}} \text{ probe} = (h(k) + ig(k)) \mod \text{TableSize}\]
Double Hashing Example

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

Resolving Collisions with Double Hashing

<table>
<thead>
<tr>
<th>Hash Functions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H(K) = K \mod M )</td>
</tr>
<tr>
<td>( H_2(K) = 1 + ((K/M) \mod (M-1)) )</td>
</tr>
<tr>
<td>( M = )</td>
</tr>
</tbody>
</table>

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?**

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. \( \text{obj}.hashCode() \% \text{nBuckets} \))
  - 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

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.