## Dictionary Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Time to insert</th>
<th>Time to find</th>
<th>Time to delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>Hash Table (Worst case)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Hash Table (Average)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Hash Tables

• Idea:
  • Have a small array to store information
  • Use a **hash function** to convert the key into an index
    • Hash function should “scatter” the keys, behave as if it randomly assigned keys to indices
  • Store key at the index given by the hash function
  • Do something if two keys map to the same place (should be very rare)
    • Collision resolution
What Influences Running time?

• How “spread out” our input keys are
  • How much do keys repeat
• Hash the function itself will take time
• Size of the table relative to the number things inserted
• How well our hash function scatters the keys
• What do we do when two things hash to the same spot
Properties of a “Good” Hash

• Definition: A hash function maps objects to integers

• Should be very efficient
  • Calculating the hash should be negligible

• Should randomly scatter objects
  • Objects that are similar to each other should be likely to end up far away

• Should use the entire table
  • There should not be any indices in the table that nothing can hash to
  • Picking a table size that is prime helps with this

• Should use things needed to “identify” the object
  • Use only fields you would check for a .equals method be included in calculating the hash
  • More fields typically leads to fewer collisions, but less efficient calculation
Collision Resolution

• A Collision occurs when we want to insert something into an already-occupied position in the hash table

• 2 main strategies:
  • Separate Chaining
    • Use a secondary data structure to contain the items
      • E.g. each index in the hash table is itself a linked list
  • Open Addressing
    • Use a different spot in the table instead
      • Linear Probing
      • Quadratic Probing
      • Double Hashing
Separate Chaining Insert

To insert \( k, v \):

• Compute the index using \( i = h(k) \mod \text{size} \)
• Add the key-value pair to the data structure at \( \text{table}[i] \)
Separate Chaining Find

• To find $k$:
  • Compute the index using $i = h(k) \% \text{size}$
  • Call find with the key on the data structure at $\text{table}[i]$
Separate Chaining Delete

• To delete $k$:
  • Compute the index using $i = h(k) \mod \text{size}$
  • Call delete with the key on the data structure at $\text{table}[i]$
Formal Running Time Analysis

- The **load factor** of a hash table represents the average number of items per “bucket”
  - \( \lambda = \frac{n}{\text{size}} \)
- Assume we have a hash table that uses a linked-list for separate chaining
  - What is the expected number of comparisons needed in an unsuccessful find?
    - In general: an unsuccessful find will be linear in the length of the list we hash to
    - \( \lambda \)
  - What is the expected number of comparisons needed in a successful find?
    - \( \frac{\lambda}{2} \)
- How can we make the expected running time \( \Theta(1) \)?
  - We need to make \( \lambda \) constant
  - Make the size of the hash proportional to the number of things in it
Load Factor?

\[ \frac{3}{10} = 0.3 \]
Load Factor?

\[ \frac{8}{10} = 0.8 \]
Load Factor?

\[ \frac{15}{10} = 1.5 \]
Collision Resolution: Linear Probing

• When there’s a collision, use the next open space in the table
Linear Probing: Insert Procedure

- To insert $k, v$
  - Calculate $i = h(k) \mod \text{size}$
  - If $table[i]$ is occupied then try $(i + 1) \mod \text{size}$
  - If that is occupied try $(i + 2) \mod \text{size}$
  - If that is occupied try $(i + 3) \mod \text{size}$
  - …
Linear Probing: Find

• \( i = h(k)\%\text{size} \)
  • If \( i \) has the key or it’s empty, then we’re done
  • Otherwise:
    • Check \((i + 1)\%\text{size}\) if it’s there, done else
    • Check \((i + 2)\%\text{size}\) if it’s there, done else
    • Check \((i + 3)\%\text{size}\)
    • ...
    • Until we hit an empty cell
Linear Probing: Find

• To find key $k$
  • Calculate $i = h(k) \mod size$
  • If $table[i]$ is occupied and does not contain $k$ then look at $(i + 1) \mod size$
  • If that is occupied and does not contain $k$ then look at $(i + 2) \mod size$
  • If that is occupied and does not contain $k$ then look at $(i + 3) \mod size$
  • Repeat until you either find $k$ or else you reach an empty cell in the table
Linear Probing: Delete

• Problem: don’t want to leave an empty space when deleting
• Option 1: when we delete, move the “last thing” with a matching hash to that location
• Option 2: “tombstone” deletion. When deleting something, leave a special marker to indicate something used to be there
Linear Probing: Delete

• Option 1: Find the last thing with a matching hash, move that into the spot you deleted from

• Option 2: Called “tombstone” deletion. Leave a special object that indicates an object was deleted from there
  • The tombstone does not act as an open space when finding (so keep looking after its reached)
  • When inserting you can replace a tombstone with a new item

\[
\begin{array}{cccccccc}
 k, v & k, v & k, v & \text{tombstone} & k, v \\
 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9
\end{array}
\]
Downsides of Linear Probing

• What happens when $\lambda$ approaches 1?
  • Runnings times get longer and longer

• What happens when $\lambda$ exceeds 1?
  • Run out of space

• We need a really small $\lambda$
Quadratic Probing: Insert Procedure

• To insert $k, v$
  • Calculate $i = h(k) \% size$
  • If $table[i]$ is occupied then try $(i + 1^2)\% size$
  • If that is occupied try $(i + 2^2)\% size$
  • If that is occupied try $(i + 3^2)\% size$
  • If that is occupied try $(i + 4^2)\% size$
  • ...

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
Quadratic Probing: Example

- Insert:
  - 76
  - 40
  - 48
  - 5
  - 55
  - 47
Using Quadratic Probing

• If you probe $tablesizetime$ times, you start repeating the same indices

• If $tablesizetime$ is prime and $\lambda < \frac{1}{2}$ then you’re guaranteed to find an open spot in at most $tablesizetime/2$ probes

• Helps with the clustering problem of linear probing, but does not help if many things hash to the same value
Double Hashing: Insert Procedure

• Given $h$ and $g$ are both good hash functions
• To insert $k, v$
  • Calculate $i = h(k) \% \text{size}$
  • If $table[i]$ is occupied then try $(i + g(k)) \% \text{size}$
  • If that is occupied try $(i + 2 \cdot g(k)) \% \text{size}$
  • If that is occupied try $(i + 3 \cdot g(k)) \% \text{size}$
  • If that is occupied try $(i + 4 \cdot g(k)) \% \text{size}$
  • ...
Rehashing

• If your load factor $\lambda$ gets too large, copy everything over to a larger hash table
  • To do this: make a new array with a new hash function
  • Re-insert all items into the new hash table with the new hash function
  • New hash table should be “roughly” double the size (but probably still want it to be prime)