# Next topic: Hash Tables

<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>
Two Different ideas of “Average”

• **Expected Time**
  • The expected number of operations a randomly-chosen input uses
  • Assumed randomness from somewhere
    • Most simply: from the input
    • Preferably: from the algorithm/data structure itself
  • \( f(n) = \) sum of the running times for each input of size \( n \) divided by the number of inputs of size \( n \)

• **Amortized Time**
  • The long-term average per-execution cost (in the worst case)
  • Rather than look at the worst case of one execution, look at the total worst case of a sequential chain of many executions
    • Why? The worst case may be guaranteed to be rare
  • \( f(n) = \) the sum of the running times from a sequence of \( n \) sequential calls to the function divided by \( n \)
Amortized Example

- ArrayList Insert:
  - Worst case: $\Theta(n)$
Amortized Example

• ArrayList Insert:
  • First 8 inserts: 1 operation each
  • 9th insert: 9 operations
  • Next 7 inserts: 1 operation each
  • 17th insert: 17 operations
  • Next 15 inserts: 1 operation each
  • …

Do $x$ operations with cost 1
Do 1 operation with cost $x$
Do $x$ operations with cost 1
Do 1 operation with cost $2x$
Do $2x$ operations with cost 1
Do 1 operation with cost $4x$
Do $4x$ operations with cost 1
Do 1 operation with cost $8x$
…
Amortized: each operation cost 2 operations
$\Theta(1)$
Hash Tables

• Motivation:
  • Why not just have a gigantic array?
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
Example

- Key: Phone Number
- Value: People
- Table size: 10
- $h(phone) = \text{number as an integer} \mod 10$
- $h(8675309) = 9$
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
A Bad Hash (and phone number trivia)

- $h(phone) = \text{the first digit of the phone number}$
  - No US phone numbers start with 1 or 0
  - If we’re sampling from this class, 2 is by far the most likely
Compare These Hash Functions (for strings)

- Let $s = s_0s_1s_2 \ldots s_{m-1}$ be a string of length $m$
  - Let $a(s_i)$ be the ASCII encoding of the character $s_i$
- $h_1(s) = a(s_0)$
- $h_2(s) = \sum_{i=0}^{m-1} a(s_i)$
- $h_3(s) = \sum_{i=0}^{m-1} a(s_i) \cdot 37^i$
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 $table[i]$
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 $i = h(k) \% \text{size}$
  • Call delete with the key on the data structure at $table[i]$
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?
  • What is the expected number of comparisons needed in a successful find?

• How can we make the expected running time $\Theta(1)$?
Load Factor?
Load Factor?
Load Factor?
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 \( i = h(k) \mod size \)
  • If \( table[i] \) is occupied then try \((i + 1) \mod size\)
  • If that is occupied try \((i + 2) \mod size\)
  • If that is occupied try \((i + 3) \mod size\)
  • ...
Linear Probing: Find

• Let’s do this together!
Linear Probing: Find

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

• Let’s do this together!
Linear Probing: Delete

• Let’s do this together!