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>$\Theta(\text{height})$</td>
<td>$\Theta(\text{height})$</td>
<td>$\Theta(\text{height})$</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>
Dictionary (Map) ADT

**Contents:**
- Sets of key+value pairs
  - Keys must be comparable

**Operations:**
- `insert(key, value)`
  - Adds the (key,value) pair into the dictionary
  - If the key already has a value, overwrite the old value
    - Consequence: Keys cannot be repeated
- `find(key)`
  - Returns the value associated with the given key
- `delete(key)`
  - Remove the key (and its associated value)
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

Key Object → $h(k)$ → Index between 0 and length-1 → Insert / find / delete → & value
Properties of a “Good” Hash

• Definition: A hash function maps objects to integers

• Should be very efficient
  • Time to calculate the hash should be negligible

• Should “randomly” scatter objects
  • Even similar objects should hash to arbitrarily different values

• 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
    • \{fields used for hashing\} ⊆ \{fields used for .equals\}
  • 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 $i = h(k) \% \text{length}$
  • Add the key-value pair to the data structure at $\text{table}[i]$
Separate Chaining Find

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

- To delete $k$:
  - Compute the index using $i = h(k) \% \text{length}$
  - 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”
  \[ \lambda = \frac{n}{\text{length}} \]
- 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?
Rehashing

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

• What does “too large” mean?
  • For separate chaining, typically we want $\lambda < 2$
  • For open addressing, typically we want $\lambda < \frac{1}{2}$
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 $i = h(k) \mod length$
  • If $table[i]$ is occupied then try $(i + 1)\mod length$
  • If that is occupied try $(i + 2)\mod length$
  • If that is occupied try $(i + 3)\mod length$
  • …
Linear Probing: Find
Linear Probing: Find

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

• Suppose A, B, C, D, and E all hashed to 3
• Now let’s delete B

Before:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

After:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Linear Probing: Delete

• Suppose A, B, and E all hashed to 3, and C and D hashed to 5
• Now let’s delete B

Before:

<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>
</table>

After:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
Linear Probing: Delete

• Suppose A and E hashed to 3, and B, C, and D hashed to 4
• Now let’s delete B

Before:

After:
Linear Probing: Delete

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

• To delete key $k$, where $h(k) = i$
  • Assume it is present
• Beginning at index $i$, probe until we find $k$ (call this location index $j$)
• Mark $j$ as empty (e.g. null), then continue probing while doing the following until you find another empty index
  • If you come across a key which hashes to a value $\leq j$ then move that item to index $j$ and update $j$. 
Linear Probing: Delete

• Option 1: Fill in with items that hashed to before the empty slot
• Option 2: “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
Linear Probing + Tombstone: Find

• To find key $k$
  • Calculate $i = h(k) \% length$
  • While $table[i]$ has a tombstone or a key other than $k$, $i = (i + 1) \% length$
  • If you come across $k$ return $table[i]$
  • If you come across an empty index, the find was unsuccessful
Linear Probing + Tombstone: Insert

- To insert $k, v$
  - Calculate $i = h(k) \% \text{length}$
  - While $table[i]$ has a key other than $k$, $i = (i + 1) \% \text{length}$
    - If $table[i]$ has a tombstone, set $x = i$
      - That is where we will insert if the find is unsuccessful
  - If you come across $k$, set $table[i] = k, v$
  - If you come across an empty index, the find was unsuccessful
    - Set $table[x] = k, v$ if we saw a tombstone
    - Set $table[i] = k, v$ otherwise
Downsides of Linear Probing

• What happens when $\lambda$ approaches 1?
• What happens when $\lambda$ exceeds 1?
Quadratic Probing: Insert Procedure

- To insert \( k, v \)
  - Calculate \( i = h(k) \mod \text{size} \)
  - If \( \text{table}[i] \) is occupied then try \( (i + 1^2) \mod \text{size} \)
  - If that is occupied try \( (i + 2^2) \mod \text{size} \)
  - If that is occupied try \( (i + 3^2) \mod \text{size} \)
  - If that is occupied try \( (i + 4^2) \mod \text{size} \)
  - ...
Quadratic Probing: Example

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

• If you probe `tablesizetime`, 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 $\frac{tablesizetime}{2}$ 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 $i = h(k) \mod \text{size}$
  - If $table[i]$ is occupied then try $(i + g(k)) \mod \text{size}$
  - If that is occupied try $(i + 2 \cdot g(k)) \mod \text{size}$
  - If that is occupied try $(i + 3 \cdot g(k)) \mod \text{size}$
  - If that is occupied try $(i + 4 \cdot g(k)) \mod \text{size}$
  - ...

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |