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)
# 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>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td>$\Theta(1)$</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>Heap</td>
<td>$\Theta(\log n)$</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>
</tbody>
</table>
BSTs and AVL Trees

• Binary Search Tree:
  • A binary tree where for each node, all keys in its left subtree are smaller and all keys in its right subtree are larger
  • Find:
    • If it matches, return the value.
    • If the search key is less than the current node, look left. If it’s greater, look right.
    • If we reach an empty spot, find was unsuccessful
  • Insert:
    • Do a find, if it was successful then update the value
    • If it was unsuccessful, add a new node to the empty spot we found.
  • Delete:
    • If the deleted node is a leaf, just remove it
    • If the deleted node had one child, replace it with that one child
    • If the deleted node had 2 children, replace it with the largest key to the left

• AVL Tree:
  • A binary search tree where for each node, the height of its left subtree and the height of its right subtree are off by at most 1.
  • Find:
    • Same as BST
  • Insert:
    • Do a BST insert, then rotate if tree is unbalanced (apply one LL, RR, LR, RL case)
  • Delete:
    • Do a BST delete, then rotate if the tree is unbalanced (apply LL, RR, LR, RL cases as needed from leaf to root)
Other Tree-based Dictionaries

• Red-Black Trees
  • Similar to AVL Trees in that we add shape rules to BSTs
  • More “relaxed” shape than an AVL Tree
    • Trees can be taller (though not asymptotically so)
    • Needs to move nodes less frequently
  • This is what Java’s TreeMap uses!

• Tries
  • Similar to a Huffman Tree
  • Requires keys to be sequences (e.g. Strings)
  • Combines shared prefixes among keys to save space
  • Often used for text-based searches
    • Web search
    • Genomes
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)
The Best Data Structure!

- Think of every key as a number
- Give each key its own index in an array

```plaintext
insert(key, value){
    arr[key]=value;
}
find(key){
    return arr[key];
}
delete(key){
    arr[key] = null;
}
```
Problem? eXuby xcs
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 long hashing itself takes

• Likelihood of collisions
  • Size of the array vs number of values in the array
  • “quality” of our hash function

• What we do when we have a collision
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
    • \( \{ \text{fields used for hashing} \} \subseteq \{ \text{fields used for .equals} \} \)
    • More fields typically leads to fewer collisions, but less efficient calculation
A Bad Hash (and phone number trivia)

• $h(phone) = \text{the first digit of the phone number}$
  • Assume 10-digit format
  • 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 $s = s_0s_1s_2 ... s_{m-1}$ be a string of length $m$
  • Let $a(s_i)$ be the ascii encoding of the character $s_i$

• $h_1(s) = a(s_0)$
  • Way more items map to $h(e)$ than to $h(q)$

• $h_2(s) = \left( \sum_{i=0}^{m-1} a(s_i) \right)$
  • Feels pretty random
  • Relatively efficient
  • Use the whole string
  • dog and god map to the same spot
  • Might be different likelihoods of even vs odd

• $h_3(s) = \left( \sum_{i=0}^{m-1} a(s_i) \cdot 37^i \right)$
  • Shares benefits with the last one
  • Slower than the last because there’s more math
  • Uses character’s position
  • 37 is prime (important?)
    • May be a problem if length is also 37
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 $\text{table}[i]$
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 $\text{table}[i]$
Formal Running Time Analysis

• The **load factor** of a hash table represents the average number of items per “bucket”
  • $\lambda = \frac{n}{\text{length}}$

• Assume we have a has 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?

$$k, v$$

$$k, v$$

0 1 2 3 4 5 6 7 8 9
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 $i = h(k) \% \text{length}$
  • If $table[i]$ is occupied then try $(i + 1)\% \text{length}$
  • If that is occupied try $(i + 2)\% \text{length}$
  • If that is occupied try $(i + 3)\% \text{length}$
  • ...
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>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, 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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After:  

<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>
Linear Probing: Delete

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

<table>
<thead>
<tr>
<th>Before:</th>
<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

• 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 $i = h(k) \mod \text{length}$
  • While $table[i]$ has a tombstone or a key other than $k$, $i = (i + 1) \mod \text{length}$
  • 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