Hashes

Another data structure optimized for efficient searches
What’s a Hash Table?

1. A collection of objects.
2. Easily found by using a key word.
   - i.e. search for object by keyword

Examples:
- Find names by searching on social security #.
- Find IP addresses by searching on computer name.
- Find car owner’s address by license plate #.
- Find lunch menu by day of the week.
- Find class grade by student name.
Hash Table Example

- Let’s map SSN’s to people names.

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>422-22-2232</td>
<td>John Doe</td>
</tr>
<tr>
<td>433-21-2456</td>
<td>Suzy Que</td>
</tr>
<tr>
<td>511-34-8946</td>
<td>Auntie Em</td>
</tr>
</tbody>
</table>

- We say that 422-22-2232 is the “key”
- We say that John Doe is the “value”
- We say that 422-22-2232 “hashes” to John Doe.
Kind of Like an Array

- Notice that an array **IS** a hash table.
- `char[] alphabet = {'a','b','c','d',..., 'z'};`
- This is a hash table with keys 0 – 25 that represent **values which are the letters of the alphabet.**
- e.g., `alphabet[3]` is ‘d’
- Similarly, **anything** stored in an array is a hash table with keys 0 to `sizeOfArray - 1`. 
But 1 – 26 Would Be More Convenient Than 0-25

• Letters are more naturally numbered from 1-26 than 0-25. So let’s make the keys 1-26.
• Then we will map the keys 1-26 to 0-25 using a hash function.

Huh? What’s a hash function? Just a mapping from the keys we would like onto a more reasonable set of array numbers.

• In this case, the hash function is
  • hash(n) = n-1.
Alphabet Hash Table

- So now we can have a hash table data structure

\[
\text{hash(n)} = n - 1
\]

<table>
<thead>
<tr>
<th>key</th>
<th>hash(key)</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>z</td>
</tr>
</tbody>
</table>

key | hash(key) | value
But What If I Want To Look Up Alphabet Number From Letter?

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>z</td>
<td>26</td>
</tr>
</tbody>
</table>

hash function???
Possible Answer

• Make the hash function take the letter (character) and convert to its ASCII equivalent!

• In computers every letter translates to a number via ASCII (or its extension Unicode).
  - a = 97
  - b = 98
  - c = 99
  - d = 100
  - etc.
Possible Letter Hash Function

- hash(character) = ASCII equivalent – 97
- public int asciiHash(char letter)
  {
    return (int) letter – 97;
  }

…yeah, that works.
But What About Hash Function If Keys Are Strings

- Suppose we are hashing names to SSNs?
  abby 311-79-8456

- Want to store in array.

- One (bad) possibility: Convert string names into sum of ascii equivalents.
  - hashFunction(“abby”) = 97+98+98+121

- So store 311-79-8456 in the 414th array position.
  - hashArray[414] = “311-79-8456”
  - When you want to retrieve this value later, just say
    - hashArray[hashFunction(“abby”)]
But ASCII Sum Not Great. Why?

• Let’s consider the ideal hash table.
  – Simple finite sized array.
  – Will not be a wastefully large array.
  – Will not have too many “collisions.”
    • When use hashFunction, don’t want it to return the same value for two different keys.
    • For example, the ASCII sum gives the same answer for “abby” and “baby”. Oops!
  – Will distribute keys evenly in the array to avoid collisions.
    • The ASCII sum is NOT uniform. Stay tuned for reason.
Example of Even Distribution

- SSN’s evenly distributed. Reduces chance of collision when hash the next name.
- Lot’s of room to put new SSN somewhere.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>433-13-4573</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>555-55-5535</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>444-64-2163</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Names Are Not Random Letters

- How many three letter words are there? $26 \times 26 \times 26 = 17,576$ possibilities.
  - But only 2851 are used in a big dictionary.
  - So even if use a big array with 20,000 spaces, only 14% gets filled.
  - So even with no collisions, get lots of wasted space in array.
But Most Keys Are Strings. So What Hashing Works Well?

- Use this function (where key is the string and key[i] is the ith character of the string, and keyLength is the length of the string).

\[
\text{hashValue} = \sum_{i=0}^{\text{keyLength}-1} \text{key}[i] \cdot 37^i
\]

- Take hashValue \%= hashTableArraySize;
  - This fits it into the array.
  - Making the array size prime helps avoid collisions.

Think of this as the string “123” being mapped to the number 123 by using key[i]*10^i. But we use 37 instead so that it covers 26 characters plus 10 numbers plus a space.
This Works Because…

- Spreads out the impact of each character.
  - abby and baby will not collide.

- The 37 is arbitrary but
  1. Is prime which helps avoid collisions.
     - Why? The modulus operator wraps around. If prime, doesn’t wrap onto itself as frequently.
  2. Covers all the letters, numbers, and a space.
  3. Making as large or larger than the alphabet helps avoid collisions.
     - e.g., suppose we are mapping numbers like “41”. Have an “alphabet” of size 10 (i.e., 0 – 9). But if we use 2 instead of 37, then will get collisions.
       - e.g., “41” and “9” will collide. 41 maps to $1*2^0 + 4 * 2^1 = 9$. And 9 maps to $9*2^0=9$.
     - If we want to include caps, punctuation, etc., then might consider a prime larger than 37.
But What If Want to Have Keys That Are Floats or Etc.?

- Ok, already. Just get creative! Find a way to convert it to a hash function!

- Java uses an object’s “hashcode” which is usually the memory address of the object.
  - Does not guarantee no collisions.
  - If hash table is size 10, then memory addresses 9 and 19 will collide.
Hash Table ADT

- **Operations**
  - void **put**(key, object)
    - insert
  - object **get**(key)
    - find
  - int **size**()
    - number of elements in the hash
  - int **capacity**()
    - max number that can be stored
  - boolean **containsKey**(key)
    - true if contains object with this key
  - object **remove**(key)
    - delete… if object in table has this key, then removes the object and returns it
Collisions: Separate Chaining

• They are inevitable. Two keys will map to the same array index sometimes.

• Solution 1: “Separate Chaining”

Use a linked list to store duplicates. (Store both key and value.) Array stores pointer to linked list. Used by Java’s hash table!
Collisions: Open Addressing

• But the additional linked lists are sometimes annoying.

• Solution 2: “Open Addressing”
  • Store both the key and the value in array.
  • And when collide, just move to the next available position in the array.
If try to put value “mouse” into position 0, it will instead get bumped up to position 1.

key = squeak
value = mouse

0 | bark, dog
1 |
2 |
3 | meow, cat
4 |

NOTE: This assumes that hashFunction(bark) = 0 and hashFunction(squeak) = 0 and hashFunction(meow) = 3
Linear Collision Function

- Linear probing modifies the hash function.

- newHash(x) = hash(x) + f(i)
  - where f(i) is the collision function
  - where f(i) = i
  - where i is the ith attempt to do an insert
    - start with i = 0
    - if collision, then increment i and try again.
Linear Problem: Clustering

So where is the next one going to go if have a collision?

Why bad? Because can take a long time to search or insert.
Clustering Speed Impacts

• Suppose we chose a really bad hash function
  • For example, every key hashes to array index 0.
    – Oooh. That’s bad!
  • Then all keys are clustered.
• With linear probing, get $O(N)$ time to insert/search an element.
  – $O(N^2)$ time for $N$ inserts.
  – But would it be any better with separate chaining?
Quadratic Collision Function

- newHash(x) = hash(x) + f(i) where f(i) = i^2

- This helps prevent clustering.
  - Try it! Draw a picture. First it tries the next cell. Then, 4 cells away, then 9, etc., with wraparound.

- AND could create even fancier collision functions.
  - But quadratic will usually do fine…
Code For Hash Table

- We need to store both the key as well as the value.

**In Java:**

```java
public class HashTable{
    public int[] key = new int[100];
    public int[] value = new int[100];
}
```

**In C:**

```c
struct HashTable{
    int key[100];
    int value[100];
};
```
But Problem With That Code?

• Can’t resize our hash tables. They are fixed to have size 100.

• Solutions:
  – In Java, pass in the array size to a constructor.
  – In C, use malloc/calloc to create variable size arrays. Maybe don’t use a struct. Just use an array and typedef it to call it a hashTable.