Hash Notes, Day 2

REVIEW Concepts

* Hash table: Data structure used to implement a symbol table.
* Central idea:
  + Suppose we are storing Keys and Values
  + We store key-value pairs in an array (or vector). Something with constant-time indexing.
  + We develop a function hash(key k) returns an integer.
  + The value that is returned by this function is called a hash value or hash code.
  + We interpret the hash value as an index into the array usually by computing   
    hash value % array size.
  + We use this array index to do all searches, insertions, and deletions.
* Things to be decided ahead of time for a hashtable:
  + What is the hash function? [can be programmer of the HT or user of HT]
  + What is the size of the array? [usually programmer, hidden from user]
  + What is the collision-resolution strategy? [always programmer]
* Show pseudocode for search, insert, delete
* Show example.
* Pros/Cons
  + Pro – for most real-world hashtables, operations are all O(1) ON AVERAGE.
    - However, for poor choices of hash function, size of array, or collision resolution, can degrade to O(n) in worst case.
  + Con – Uses more space than most other implementations of a symbol table, because there is a high chance of lots of empty (wasted) space in the array.
  + Con – does not support any sort of range query (find closest key > or < a certain value, all k-v pairs where they key is in a certain range, etc). Only supports exact searches.
  + This is a time-space trade-off.

NEW MATERIAL

* Let’s explore the hash function and its interaction with the size of the array a little bit.
* Basic goal – get the keys to be roughly equally distributed into each index of the array.
* hash value % array size. 🡨-- called modular hashing
* If we use modular hashing, prime numbers work best for the size of the array. That’s because if we use a size of the array that is divisible by something, it often reveals patterns in the keys that are being hashed.  
  + Example: (from last time). Suppose all the keys end up hashing to even numbers. Then if you use a size of an array that is also even, you’ll never hash to any odd values.
  + Example (from book): Suppose all the keys are phone numbers, and you use the area code. Area codes usually have a 0 or 1 in the middle. So if you pick an array size that is 100, then all the hash keys will be less than 20.
  + Because in general, hash tables are built to be as flexible as possible (and usually people don’t want to worry about manually picking a size), prime numbers work well for hash table sizes because they don't have any factors other than themselves and 1. So they are more likely to equally distribute keys into the indices of the array.
* More about hash codes.
  + Mantra: Try to use the whole key for computing the hash value.
  + Example: we talked about creating a hash function for strings by using the code for just the first char in a string. But what if we hash on Rhodes R#’s? All start with the same code.
  + Best practice: Use built-in hash functions as much as possible. If you are hashing any basic data type (string, int, float, double), then use the built-in hash code functions. All major programming languages, Python, C++, Java, etc have these.
  + Best practice: If you are hashing an a user-defined object, you will usually take the hash code of the individual components and then combine them into a single hash code. Again, most PLs have functions to combine multiple hash codes into one in an easy way.
  + Best practice: Don’t write your own Hashcode function from scratch. It’s easy to fall into the trap of making a code that has some kind of hidden pattern in it that will mess up the distribution of hash codes into the array indices.
* To be clear, if you pick a bad hash function or bad array size, everything will work, but much more slowly than you’d hope. This is the O(1) performance degrading to O(n).

Last part to talk about is collision resolution. -> What do we do when the hash function tells us that two k-v pairs should go into the same array index?   
  
Remember what a collision is. A collision is where two different keys hash to the same index in the array. In other words, a collision is where we are inserting k1, v1 and k2, v2, and we have hash(k1) == hash(k2). [But obviously k1 != k2].   
  
In order to study this in more depth, we define  
  
A critical statistic for a hash table is the *load factor*, defined as LOAD FACTOR = n / k

{\displaystyle {\text{load factor}}={\frac {n}{k}},}

where

* *n* is the number of entries occupied in the hash table.
* *k* is the number of buckets.  
    
  The higher the load factor, the more collisions we have, and the slower our hash table will run. That’s can happen in two ways:  
    
  either n is too big (too many things in the hash table).  
  Or k is too small (table is too small).  
    
  Perfect analogy is finding something in your dorm room. When do you start to lose things in your dorm room? That means either you have too much stuff, or your dorm room is too small! (Or it’s too messy, which means you should rewrite your hash function.)

As the load factor grows larger, the hash table becomes slower, and it may even fail to work (depending on the method used). The expected [constant time](https://en.wikipedia.org/wiki/Constant_time) property of a hash table assumes that the load factor is kept below some bound. For a *fixed* number of buckets, the time for a lookup grows with the number of entries and therefore the desired constant time is not achieved. As a real-world example, the default load factor for a HashMap in Java 10 is 0.75, which "offers a good tradeoff between time and space costs."[[10]](https://en.wikipedia.org/wiki/Hash_table#cite_note-JavadocHashmap10-10)

* Most strategies fall into one of two campus: SEPARATE CHAINING or OPEN ADDRESSING.
* We can make the array store a linked list of items, and whenever there is a collision, we add it to the linked list.
* This is called **Separate chaining**.
* Do Insert 5, 11, 18, 23, 28, 13, 25 with chaining.
* Use % 10 hash function (poor function, but easy to use for example).
* Sep chaining is probably the most widely used and fastest implementation of a HT.
  + OPEN ADDRESSING  
      
    linear probing
    - insert-keep going linearly in table until we find an empty spot.
    - lookup: straightforward. keep going linearly until we find the item, or a -1.
      * case 1 – item is at first location: we’re done (positive)
      * case 2 – “-1” is at location: we’re done (negative)
      * case 3 – something else is there, search the table. until we reach a -1 or wrap all the way around.
    - deletions – common strategy is to use a DELETED flag.

* Assume hash table with 7 locations. h(x) = x % 7.
  + Insert 5, 11, 18, 23, 28, 13, 25 with linear probing.