Hash Day 2, with new book

Outline:

Review topics

Introduce open addressing (linear probing), quadratic probing, chaining.

done!

REVIEW

* Illustrate idea of hash function.
* How to create a hash function.
  + Preface – in real world, you rarely have to generate a hash code completely from scratch. Most modern PLs, including Java, C++, and Python, all have many hash code functions built-in.
  + For instance, in Java, since hashCode is in object, every built-in object in java already has a hashcode function built-in. i.e., if you have a String s, you can call s.hashCode(), if you have an ArrayList<Integer> list, you can call list.hashCode and everything just normally works.
  + The problem is what happens when you are defining your own objects. If you want to store your own objects into hashtables (as the key), then you must define your own hashcode function.
  + The usual way this is done is to come up with a function that combines as many different pieces of the object as possible, but still is fast to compute, and generates a wide range of integers for different inputs.
  + Example with Strings.
    - In modern programming language, string are represented using something called Unicode, which means for every character in a string, there's a number that corresponds to it.
    - For instance, the capital letter A is 65, and the lowercase letter A is 97.
    - Example hash functions.
    - Unicode number for the first letter in string (fast, lots of collisions).
    - "" for all letters, summed together (still pretty fast, but now anagrams collide).
    - show what java actually does, which is position based.
  + mod size of table
  + pick a prime number for size of table.-> avoids patterns
* Get pet names from students, Use hashcodes: 41, 73, 50, 26, 92, 34, 18, 65.
* Illustrate with a table of size 5, then table of size 10.
  + [storing dogs and their ages]
* **Open Addressing**
  + There are two strategies for organizing hash tables: open addressing and chaining, and each strategy deals with collisions in different ways.
  + In **open addressing**, each slot in the array of our hash table always stores exactly one key-value pair. In **chaining**, we will let each slot in the array store multiple key values pairs.
  + First we will talk about open addressing.
  + Recall that a collision is where two items that we wish to store in a hash table end up with the same hash code. Since we use this code to give us the index in the array that we store the item at, if two hash codes are equal, this effectively tells us that these two items need to be stored in the same position, and we can't do this.
  + Open addressing resolves this situation by allowing us to store one item in the correct slot in the array, and the 2nd item in a different slot in the array. Here's how one way to determining the "different slot" works.
  + This is called **linear probing**: If an item's hash code ever says that our item should be stored at index [i] in the array, and we look at index [i] and there's a different item there, we will check index [x+1] instead. If the item isn't at [i+1], then we'll check [i+2]. And we will keep doing this until we either find the item or we reach a null entry. We stop when we find a null because null means the item isn't in the table at all.  
    - Run example:
    - Get pet names from students, Use hashcodes: 41, 73, 26, 92.
    - Table of 5 locations: 0—4.
    - They will hash to locations 1, 3, 1 (goes to 2), 2 (goes to 4).
    - insert alg:
    - algorithm:
    - compute index by taking the items hashcode % table size.
    - if table[i] == null
      * put item at this location
    - else if table[i] == item
      * item is already in table (don't add twice)
    - else
      * continue to search table by incrementing index (with wraparound) until we find null, and put the item where the null is.
    - search algorithm:
    - compute index by taking the items hashcode % table size.
    - if table[i] == null
      * item not in table
    - else if table[i] == item
      * item is in table (return value for map)
    - else
      * continue to search table by incrementing index (with wraparound) until either the item is found or a null is found.
    - Last step might cause infinite loop:
      * either stop when we wrap all the way around
      * or, we can rehash the whole table to a bigger table
  + Deletion in open addressing.
  + When we want to delete an item using open addressing, we follow the same basic idea above. Compute hash code, % table size, check at that index, and traverse the table with wraparound until you find the item. When you find the item, you can't replace it with null! Why not? (if you do, then when you search for an item with the same hash code as the deleted item, you may not find it, because a search may end prematurely).
  + So instead, we store a "dummy value" that slot instead of replacing the item with null. Unfortunately, these dummy values have to be treated like real values in the table, and not like empty slots that we can just immediately overwrite.   
    - example – suppose we have three items that all hash to the same value. Insert them all. Now delete the one that was inserted first. Now try to re-insert the third one. If we assume that we can overwrite the "deleted" spot, then we'll end up having two copies of the third item in our table.
    - So these dummy items can never be overwritten, and they just end up wasting space. So what happens is as your table fills up and/or you have dummy values, your efficiency degrades from O(1) to as bad as linear O(n).
  + So what are some remedies to this?
  + Use a larger table -> rehash everything. One big O(n) rehash.
  + The problem with linear probing is that it tends to cause "clusters" of keys in the table, causing very long search chains sometimes.
  + example: make table of size 10, insert hash code 5, 6, then 5, 6, 7.
* QUADRATIC PROBING  
  + One way to fix this is to replace LINEAR PROBING with QUADRATIC PROBING.
  + The idea is that instead of incrementing the index by 1 each time, we increment it by a factor that keeps getting larger and larger. A common way to do this is use the sequence of square numbers 1, 4, 9, 16, 25, etc. [hence the name quadratic].
  + So if we want to store something at hash index 5, if it's full, we will then try index 6 (5 + 1), then index 9 (5 + 4), index 14 (5 + 9), instead of 6, 7, 8.
  + Reasoning – this breaks up clusters more because for every time we find another slot that is full, we jump farther and farther ahead, hopefully to a location in the array that hasn't been touched yet.
* Any open -addressing strategy, no matter how good it is at resolving collisions and avoiding clustering, still has the problem of that colliding values are stored in locations that should be reserved for items that hash direction to those locations. We’re kicking the can down the road --- just stalling for time and in a sense, creating the chance for more collisions.
* There’s a different policy we can use, that results in making the hash table itself slightly more complicated, but eliminates the previous problem of “borrowing” table slots from other items.
* We can make the array store a linked list of items, and whenever there is a collision, we add it to the linked list.

WRAP-UP of Hashing:

* Insert into hash table/search hash table/delete from hash table -> O(1) in average case, can degrade to O(n) in the worst case.
  + Worst case would be [for open addressing] when the table is very full and you have to cycle around it, or when everything hashes to the same slot (or a very small number of slots).
* So the insert/search/delete functions for BSTs are O(log n) average case, and for hash tables, these are O(1). So why would you ever want to use BSTs over Hash tables, since hashing is faster (on average)?
* The main advantage is that BSTs keep their keys in sorted order, and hash tables do not.  
  + This is useful when you want to have your map or set answer additional questions besides these three basic operations.
  + For instance, some set/map implementations allow you to ask, "What is the smallest/largest key in the map/set? Or "what is the smallest key that is larger than a given key?" Or "give me a traversal of the keys in the map in sorted order."
  + In a BST these operations are fast, because the BST keeps the keys in sorted order. For instance, finding the smallest/largest key in BST would be O(log n) [avg case]. Traversing the keys in sorted order would be O(n) [worst case]
  + In a hash table, these operations would be slower. Finding the smallest/largest would be O(n) [have to look at the whole table]. Traversing the keys in sorted order -> no good way to do that, aside from copying the whole hash table into an array and sorting it.
* So, the TAKEAWAY for using these in real world situations.  
  + Most modern PLs have one or both of these implementations. Java has both, for both maps and sets. (TreeMap/TreeSet/HashMap/HashSet). Python only has hash tables. C++ has both.
  + If you can easily create a hash function for your keys, and you don't need "min/max/nearest" queries, then the hash table implementation will usually be faster.
  + If you need min/max/nearest queries, choose BST [or a better BST with built-in balancing, like a red-black tree]