Hash Tables

Warmup Question

Perform the following operations on a binary search and sketch the tree at each step

```
insert(5)
insert(3)
insert(4)
insert(2)
insert(1)
delete(2)
delete(3)
```

Another implementation of the `Map< K, V >` ADT

- We actually have a REALLY fast and simple implementation of the `Map` ADT for a particular type of key
- When keys are small, positive integers, a plain old Java Array is great!
- Can we use that as a starting point for a map that can hold any kind of keys?

Hash Table Outline

- The hash table map implementation is pretty simple!
- We convert out key's into small positive integers somehow, and use that into an index
- When store keys/values in an array and use that index
- Let's look at the details

Hash Functions

- A **Hash Function** is a function that takes in an object and returns an integer
- A good hash function should turn different objects into different integers (why?)
- It's unavoidable for some objects to “hash” to the same value (why?)
- In Java, the `Object` class includes the method `int hashcode()` so ALL objects are hashable.
- The implementation in `Object` just looks at the object's address. You should override the `hashcode` method in your classes if you want to use them with hash tables
- Note: if you override `hashCode` or `equals` you should override both of them

Hash Tables

- A hash table implements the `Map` and/or `Set` interfaces, we'll consider `Map` first
- The storage is just a basic array
- When we store a `K, V` pair, we hash the key to get an integer which will be used as the index into the array (almost...)
- Each used entry in the array will store a the `K` and `V`
- That's basically it!

Compression Functions

- The `hashcode` method in `Object` can't know the size of our array, so it almost certainly won't return an integer which is a valid index into our array
- A **Compression Function** takes an arbitrary integer and returns one in the range [0, N) so we can use it as an index
- The simplest compression function is modulo (the `%` operator) which just wraps the number around if it's greater >= the array size
- Note: in Java we have to be careful when our hashcode is negative, because `%` can return a negative number
- In one line: `put(K k, V v) → arr[compress(hash(k))] = value`

Collisions

- Let's say our basic array has `10` elements, our key type is `String` and our value type is `Integer`
- If we try to store `11` `K, V` pairs in our hash table, we're guaranteed to have 2 strings hash/compress to the same index
- No matter how big we make the array, and no matter how we choose the hash function it will be possible (and in fact likely) we'll try to store multiple objects at the same index
- This is called a **collision**
- We should design our hash functions to avoid collisions, but we can't eliminate them entirely and must handle them in our hash table implementation

Solution 1: Chaining

- This is perhaps the conceptually simplest solution
- If there are multiple items that should be stored in the same “bucket” ... fine, we'll support storing multiple values at each array index
- A common approach is to have the array not store (K,V) pairs, but LinkedList's of Key/Value pairs

Runtime Analysis

- All 3 of the map operations (put, get, remove) require us to hash the key and either access/remove a value from the appropriate “bucket”
- The runtime of the hash function doesn't depend on the number of items in our hash table (it might depend on the size of the object, for example, the length of a string), so it's O(1)
- Compression and array access are also O(1)
- So, the runtime depends on the length of the list we use to store values in each “bucket”

Load Factor

- The **load factor** of a hash table, which we often use the greek lambda (λ) for is just numberOfElements/arrayLength
- An empty hash table has load factor 0, a hash table where each bucket has exactly 1 value has a load factor of 1
- For a chaining hash table, λ tells us the average length of each linked list
- So, if the load factor is O(1) (it's common to keep it below 1), all operations are expected to be O(1)!
- How do we keep the load factor small? Why isn't O(1) behavior guaranteed in this case?

HashSet

- To implement the `Set` ADT, we just store the keys in our hash table, and there are no values
- If you only have a HashMap available, you can make a HashSet from `HashMap< K, Boolean>` which is useful in languagues that have a built in hash map (like perl, python, D, ...)
- `Java's` `HashMap` and `HashSet` are hash table based implementations of the `Map` and `Set` ADTs with O(1) expected runtime for the operations we discussed

Hash Table Iterators

- A hash table iterator stores it's position in the backing array (like an arraylist iterator)
- next() will skip over any unused array spots and find the next one that's in use
- When we use chaining, we'll use a `LinkedList Iterator` to iterate through each “bucket”
- So a chaining iterator would store an index + a LLIterator as members
- What is the runtime of iterating through the whole hash table?

Handling collisions with probing

- Instead of storing multiple items with the same hash in the same “bucket” we could “try another spot”
- This is called “probing” and basically when we insert, if the slot we want to use is full, we find another one
- When we're searching (for get, or remove) we need to be sure we look in the same spots that insertion did!

Linear Probing: Insertion

- If there's a collision when inserting a value, we iterate through the array from that spot until we find an unused one, wrapping around if we hit the end
- If our hash function is "good" and spreads the keys out well, then probably we won't have to look very far
- Example: let's consider an array with 10 slots and the "identity" hash function (`h(x) == x`). What happens when we add keys 8, 9, 88, 89, 90, 91, 92?
- What's our final load factor, λ?

Linaer Probing: Searching

- Now that a key can be at many indices, we have to perform a search starting at `c(h(k))`
- We need to keep searching until we find the key we're looking for or we hit an empty spot in the array
- How many elements would we look at if we searched for 38 in the previous example?

Linear Probing: Removing items

- Removing starts with a search like the previous slide What should happen when we find what we're looking for?
- In the previous example, if we delete 9 by marking it's array slot as unused, what happens when search for 89 in the future?
- We can't mark spots as unused when we delete because they might be on the “probe path” for other entries. In general it's hard to check to see if this is the case, so we assume that it is
- Instead, we perform a “lazy delete” where we just mark that element as “deleted”. Future searches must keep going past “delete” nodes, but future inserts can use this slot
- A common term for the “deleted” marker is a “tombstone”

Linear Probing: Analysis

- How far do we have to search?
- Assuming items are distributed randomly, the probability of any spot being occupied is λ
- So the probability of hitting k cells in a row that are full is λ^k
- The expected number of spots we'll have to check is requires a bit of probability knowledge, but turns out to be $1/(1-\lambda)$
- Gut check: does that make sense when λ is 0, or 1? 0.5?

Bad News: Primary Clustering

- Unfortunately, the cells are NOT occupied at random. When there are collisions, we iterate forward to the next empty slot, which means it's common to have long runs of occupied slots
- With some more analysis we can see that the actual expected number of probes is $(1+1/(1-\lambda)^2)/2$
- If lambda is .9, the naive estimate is 10, and the more accurate value is 50!

Linear Probing Wrapup

- If lambda is < 0.5, things are probably OK, so we will rehash when lambda gets around there
- Recall: rehashing means allocate a larger array and insert all the elements into it
- A nice property is that if $\lambda < 1$ we're guaranteed to find a slot (seems obvious, but not all probing schemes guarantee this!)
- Performance also degrades if we add/delete lots of items so there are lots of tombstone elements (which operations are affected by tombstones?)
- We may also want to compute a modified load factor and use it to trigger a rehash

Avoiding Primary Clustering

- Linear Probing can lead to long runs of full slots that we have to search through
- We can (hopefully) avoid this by changing where we look for alternative locations
- There are MANY schemes (double hashing, cuckoo hashing, ...)
- We'll look at the next simplest scheme called “quadratic probing”

Quadratic probing

- Instead of searching `c(h(k)) + i` where i = 0..k to find a spot, we'll use a different set of locations
- Quadratic probing checks `c(h(k)) + i^2` for i = 0..k
- In other words, we look at the expected index first. If there's a collision we look at the next slot (like in linear probing)
- After that, though, we look at the original slot + 4, slot + 9, slot + 16, etc (all wrapping around as needed)
- Because we're jumping farther and farther away, we expect to avoid the problem of primary clustering
- Let's try this with the same sequence as before: 8, 9, 88, 89, 90, 91, 92

Issues with table size

- Assume h(k) = 0 and our table has size 16. Which indices would our probe sequence look at?
- 0, 1, 4, 9, 25%16 == 9, 36 % 16 = 4, 49 % 16 == 1, ...
- We don't ever see index 2!
- To guarantee that we hit every index in our sequence, we need the table size to be a prime number
- When we rehash, we can double the array size, then find the next prime number above that (Java's `BigInteger` class has a method `nextProbablePrime`)
- If the size is prime and $\lambda < 0.5$ we'll find a slot without checking any slots multiple times

Problems solved?

- Quadratic probing can have “secondary clustering” but we expect fewer probes than linear probing if λ is small
- The cost of probes is slightly higher (computing indices is a teeny bit more complex than adding 1), and finding a prime size might be a bit more expensive than doubling, but those are likely negligible costs
- In general, if we can overallocate our array (keeping $\lambda < 0.5$ quadratic probing hash tables work well

Double hashing

- A similar approach is “double hashing” where we have 2 different hash functions
- Our search locations are h1(x) + i * h2(x)
- In other words, our “step size” is determined by a second hash function, which should ideally be different for different keys