CS61B Lecture 24

Wednesday, March 18, 2020

Hashing

Hashing is another form of searching. The idea is that a linear search over a list of things is slow; so, it would be nice if we could quickly split our search into many little pieces, because small searches are fine with linear search.

Suppose that in constant time, we could put any item in our data set into a numbered bucket, where the number of buckets stays within a constant factor of the number of keys. We should also suppose that each bucket has roughly the same number of keys. In this scenario, the search becomes constant time.

Hash functions

To do this, we must have a way to convert key to a bucket number, which is called a hash function.

For example, we have

data items. Keys are `longs`, evenly spread over the range 0 to

We want to keep the maximum search to L = 2 items. We can use a hash function:

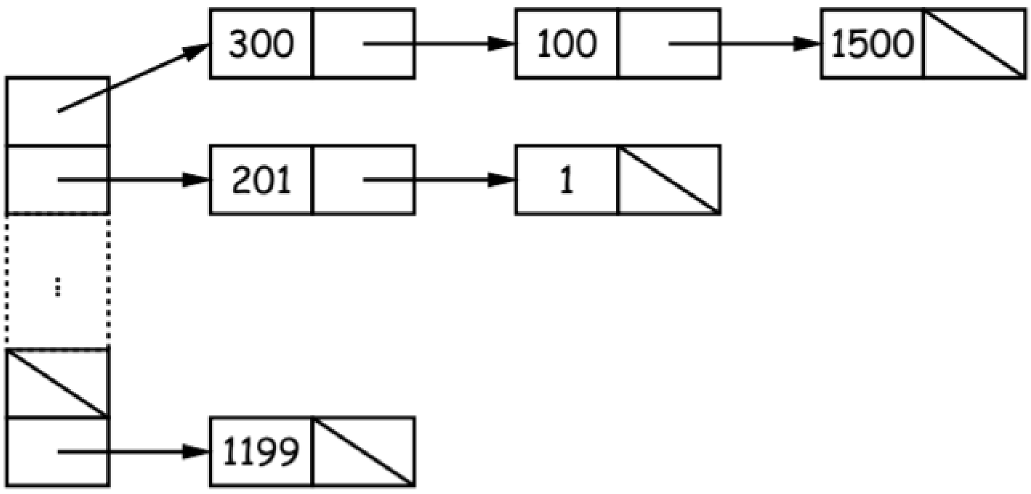
where

is the number of buckets:

So the numbers 100232, 433, and 100233482 would go into different buckets, but 10, 400210, and 210 all go into the same bucket.

External chaining

Say that we have an array of M buckets, and each bucket is a list of data items.



Not all buckets have the same length, but the average is or otherwise called the load factor.

To work well, the hash function must avoid **collisions**: keys that "hash" to equal values.

In this case, there is a possibility some buckets are much larger than others. The idea is to pick one that "smooths out" the distribution of items among buckets so that no one bucket gets too large. It is a similar problem to BSTs, where we want to keep trees bushy.

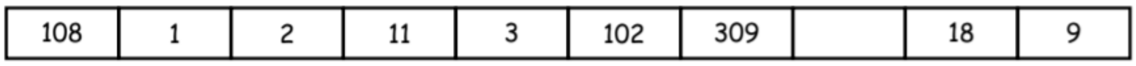
Open Addressing

So how can we solve this problem? Here's an idea: we can put one data item in each bucket. When there is a collision, and the bucket is full, we can just use another bucket, in some systematic way.

There are various ways of doing this:

- Linear probes: If there is a collision at h(K), try h(K) + m, h(K) + 2m etc. (wrap around at end)
- Quadratic probes: h(K) + m^2, h(K) + 2m^2, ...
- Double hashing: h(K) + h'(K), h(K) + 2h'(K)

Let's take an example where we have the same hash function as before based on modulo, with M = 10, and linear probes with m (offset) = 1. We then add 1, 2, 11, 3, 102, 9, 18, 108, 309 to an empty table.



So first we put 1, which works fine, and 2, which is also fine. When we try to add 11, we see the bucket at 1 is already filled, so we add the linear offset m, find the bucket 2 occupied, and add the offset again, where we find an empty 3 bucket. When we try to add the element 3, its bucket is occupied and we move it to the next bucket.

Things can get kind of slow with open addressing hashes, even when the table is far from full. There is lots of literature on this technique, but Professor Hilfinger usually settles for external chaining.

Filling the Table

To get constant-time lookup, we need to keep the number of buckets within some constant factor of the number of items. Thus, when the load factor gets higher than some limit, we should resize the table. In general, we must rehash all table items.

Still, this operation must be constant time per item, which we can achieve by doubling the table size each time, such that we can amortize the time for insertion and lookup, assuming our hash function is good.

String Hash Functions

For some String "s0 s1 ... s(n-1)", we want a function that takes all characters and their positions into account.

Here's an idea: why don't we just add up the numerical values of the individual `char`s?

One possible problem is that permutations of strings have the exact same value -- "cat" and "act" for example.

But more importantly, due to probability and statistics (the Central Limit Theorem specifically), a large majority of words will end up right in the middle, in which we get a normal distribution where the keys are clustered around the average value of the alphabet, instead of the equal distribution we want for each bucket.

For strings, Java itself uses something slightly different:

The number 31 is selected such that to fit under the `int` maximum value.

To convert a table index in {0, ..., N-1}, compute

(but don't use a table size that's a multiple of 31!)

This is not as hard to compute as you might think, as you do not even need multiplication!

```
int r = 0;
for (int i = 0; i < s.length(); i++) {
    r = (r << 5) - s.charAt(i);
}
```

Another way of saying this:

```
r = 31 * r + s.charAt(i);
```

Hash Functions

Lists (`ArrayList`, `LinkedList`) are analogous to strings. For example, Java uses:

```
hashCode = 1; Iterator i = list.iterator();
while (i.hasNext()) {
    Object obj = i.next();
    hashCode = 31 * hashCode
        + (obj==null > 0 : obj.hashCode());
}
```

We can limit time spent on computing hash function by not looking at the entire list. For example, look at only the first few items if you are dealing with a `List` or `SortedSet`. This causes more collisions, but it does not cause equal things to go to different buckets.

A recursively defined data structure is also a recursively defined hash function. For example, on a binary tree, one can use something like:

```
hash(T):
    if (T == null):
        return 0
    else:
        return someHashFunction(T.label())
            ^ hash(T.left()) ^ hash(T.right())
```

This Python hash function recurses down the tree.

Identity Hash Functions

What Java does by default is to use the address of an object, or the hash code value provided by the class of the object. We can do this is distinct objects are never considered equal.

The problem with this scenario is that it won't work for Strings, because `.equal` Strings could be in different buckets:

```
String H = "Hello", S1 = H + ", world", H2 = "Hello, world"
```

Here, `S1.equals(S2)`, but `S1 != S2`.

We can override this behavior for classes like String, or your particular class, using the function `hashCode()`, which returns the identity hash function by default.

For reasons we detailed above, this function is overridden for String, as well as many types like `List`s. The types `Hashtable`, `HashSet` and `HashMap` use `hashCode` to give you fast lookup of objects.

```
HashMap<KeyType,ValueType> map = newn HashMap<>(approximate size, load factor);
map.put(key, value);     // Map KEY -> VALUE
... map.get(someKey);    // VALUE last mapped to by SOMEKEY
... map.containsKey(someKey); // Is SOMEKEY mapped?
... map.keySet();        // All keys in a MAP (a Set)
```

Python does rather some interesting: the hash code is not automatically defined for a type, and the dictionary will actually refuse to take in items of some type if the hash code method is not defined for that type.

Monotonic Hash Functions

Suppose our hash function is monotonic: either nonincreasing or nondecreasing, such that if key k1 is greater than k2, then the value of h(k1) is greater than or equal to h(k2).

An example of this is that the items are time-stamped records and the key is the time, and our hash function is to have one bucket for every hour.

In this case, can you use a hash table to speed up range queries?Hash maps in general are not particularly good at range queries, so this would improve its ability to do so. As for the how, it's pretty obvious: we just need to find what hours are in that range, and we'll just look in those buckets.

Could this be applied to strings, and when would it work well?

Perfect Hashing

Here's another variation on hashing. Suppose our set of keys is fixed. A tailor-made hash function might then hash every key to a different value; this is called **perfect hashing**.

In this case, there is no search along a chain, or in an open-address table: either the element at the hash value is or is not equal to the target key.

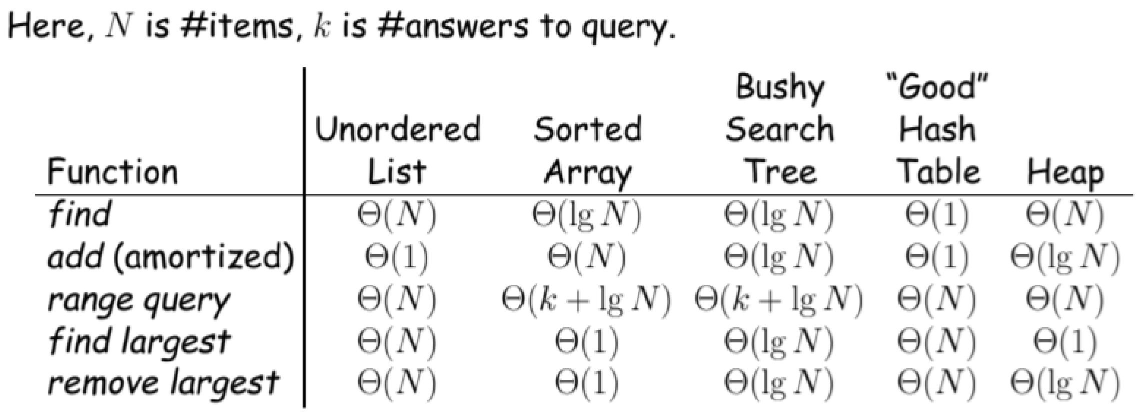
For example, we might use the first, middle and last letters of a string, read as a 3-digit base-26 numeral. This might work is those letters differed among all the strings in the set. Or we might use the Java method, but tweak the multipliers until all strings gave different results. The idea is to ensure there are no collisions between different values.

There is a lot of literature on this, but a good place to start if you're interested is the "The Art of Computer Programming: Volume 3" by Donald Knuth.

Characteristics

Assuming we have a good hash function, add, lookup and deletion take constant amortized time. It is good for cases where one looks up among equal keys. But they are usually bad for range queries, except in the special case mentioned before, and it is bad for small sets you rapidly create and discard, because there is a certain amount of overhead in creating a hash map, and this can be avoided by using a simple linear sequence.

And finally, here is a longer summary of all search operations we have studied so far:



Here, N is #items, k is #answers to query.

			Bushy	“Good”	
	Unordered	Sorted	Search	Hash	
Function	List	Array	Tree	Table	Heap
find	$\Theta(N)$	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(N)$
add (amortized)	$\Theta(1)$	$\Theta(N)$	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(\lg N)$
range query	$\Theta(N)$	$\Theta(k + \lg N)$	$\Theta(k + \lg N)$	$\Theta(N)$	$\Theta(N)$
find largest	$\Theta(N)$	$\Theta(1)$	$\Theta(\lg N)$	$\Theta(N)$	$\Theta(1)$
remove largest	$\Theta(N)$	$\Theta(1)$	$\Theta(\lg N)$	$\Theta(N)$	$\Theta(\lg N)$