CS 310 – Advanced Data Structures and Algorithms

Hashing

June 5, 2018
Hashing

- Hashing is probably one of the greatest programming ideas ever. It solves one of the most basic problems in computing: the need to efficiently store and lookup big (sometimes huge) amounts of data.
- It allows us to do lookup, insert and delete operations in expected (average) constant time.
- The lookup time does not depend on the size of the input.
Hashing

- A technique for fast lookup by key.
- Keeping an array (lookup table) with a subscript for every possible value we might want to look up.
- Say we have a Map with 2000 integers in the domain, with values 0 .. 1999.
- We can create a 2000 element array a[ ] and look up the range entry for value i in a single reference to the array, a[i], itself a pointer or reference.
- Array lookup is done by computed address: \( \text{addr} = \text{start-address} + \text{size-of-entry} \times \text{index} \).
- This is a lookup in \( O(1) \) time.
Less Trivial Example

- For large, sparse domains, this plain-array approach is impractical.
  - With a larger domain, like 1..1000000 with only 100 values in use we can still set up an array.
  - Wastes memory but gives us O(1) lookup, Insert, and Delete.
- What if the domain is not integers at all?
- Solution: We map the domain to addresses with a more complicated function called the hash function.
- The hash function computes the “bucket number”, itself an array index, and we find the array index by calculating:
  \[ \text{addr} = \text{start address} + \text{index} \times \text{size of entry} \]
A quick way to do lookup: O(1) insert, delete and find.

A hash table is a fancy array of “buckets” containing the data.

The hash function maps key values to array entries.

From Wikipedia
Hashing – Illustration

- Hash function properties:
- Map key elements to integers.
- Fast to calculate.
- Minimize collisions – Not many different keys hash to the same value.

From Wikipedia
Hashing Terminology

- **Keys:** each value of type keytype can be called a key. It just means that we’re going to do a look-up using this value.

- **Hash table:** the array in use, of some size $M$.

- **Hash bucket or hash slot:** a subscript in the hash table array, these are numbered from 0 to $M-1$. $M$ is the number of buckets.

- **Hash function:** a function from the keytype to a bucket (array entry) number: $b = h(x)$, where $x$ is of type keytype and $0 \leq b < M$ is the bucket number. We say “$x$ hashes to $b$”.

- **Collision:** when two keys $x$ and $y$ hash to the same bucket: $b = h(x) = h(y)$. 

$h(x)$ is a computed mapping and is expected to take $O(1)$ computation time.
Example of Hashing

- We have a map of int to int with $4 \rightarrow 100, 55 \rightarrow 44, 10 \rightarrow 12$
- Here 4, 55, and 10 are the keys.
- The hash function is $h(x) = x/10$, for hashing the keys.
- $h(4) = 0, h(55) = 5, h(10) = 1$
- 4 hashes to 0, 55 hashes to 5, and 10 hashes to 1.
- Hash table: Set up array of 10 spots, put the (key, value) pairs in the array by hash bucket:
  - $a[0] = (4, 100)$ (ref to object containing 4, 100) $\rightarrow$ bucket 0 for key 4
  - $a[1] = (10, 12)$
  - $a[2] = \text{null}$
  - ...
  - $a[5] = (55, 44)$
Look up 55: $h(55) = 5$, $a[5] = \text{ref to } (55, 44)$, 55 matches, so value $= 44$

Look up 56: $h(56) = 5$, $a[5] = \text{ref to } (55, 44)$, no match so value not there

Luckily, the quick example has no “collisions” (two keys hashing to the same bucket).

The above example is “hashing integers”. Similarly we can hash strings by coming up with a function that maps strings into bucket numbers.

We see again that a hash function is just a computed mapping of some keytype to array spots.
Implementing Maps Using Hashing

Example: Given a string, count the occurrence of the 5 English vowels, using map from chars to ints.
- 'a' → count of a’s
- 'e' → count of e’s
- ...
- 'u' → count of u’s
String s = "this is a test"; // to count vowels in
// set up HashMap stats
Map<Character, Integer> stats = new HashMap<Character, Integer>();
// with 5 put's, add ('a',0) ('e', 0), ('i', 0), ('o', 0), and ('u', 0)
// to map
for (int i=0; i<s.length(); i++) {
    c = s.charAt(i);
    Integer count = stats.get(c);
    // get Object, so can test if null
    if (count != null) // if vowel - found in map
        stats.put(c, count.intValue() + 1);
}
print "a's: " + stats.get('a');
print "e's: " + stats.get('e');
...
Maps Using Hashing – Vowel Example

- How do we implement a map with characters as domainType?
- We need a hash function from chars to integers from 0 up to some limit M-1 (table size).
- We can use the ASCII codes of the chars.
- \( h(x) = x \mod M \) does the trick, where \( x \) is the ASCII code.
- 'a' = 97, 'e' = 101, 'i' = 105, 'o' = 111, 'u' = 117
- Simplest M to figure with is M=10, the doubled size of the domain. Then \( x \mod M \) is just the last decimal digit of \( x \).
- Then \( h('a') = 7, h('e') = 1, h('i') = 5, h('o') = 1, h('u') = 7 \).
- Two 2-way collisions! What bad luck to use only 3 slots out of the 10 we have here.
Or is it luck? What’s wrong with M=10? It’s not a prime.

For some reason, the factors of M cause a lot of collisions, especially in biased samples.

Try M=11. $h(x) = x \mod 11$.

Then $h(‘a’) = 9, h(‘e’) = 2, h(‘i’) = 6, h(‘o’) = 1, h(‘u’) = 7, h(‘y’) = 0$.

No collisions! A prime does not guarantee this perfection, but tends to give better results than a number with factors, esp. lots of different factors, and factors of 2 or 5, used in our number base.
The hashing itself is hidden inside the HashMap implementation.

Note: there might be collisions in the HashMap case, since we’re not taking control of the exact hash function.

It’s OK, though, because HashMap takes appropriate action.

hashCode(): Only needs to provide an int. HashMap, etc., will scale it to the right array size.

hashCode() will always return the same value for the same input, but due to scaling it will not necessarily always end up at the same array entry in the end.
Strings of fewer than 5 chars can be assembled into an int $x$ by left-shifting the chars of $s$ by 0, 7, 14, and 21 bits and combining (4 bytes = integer).

Longer strings: it's very important to let all parts of the string contribute to the result.

Think of hashing URLs, for ex., "http://www." Better not be using just the first 12 chars!
Possible Hashing Function

```java
public static int hash(String key, int tableSize) {
    int hashVal = 0;
    for(int i=0; i<key.length; i++)
        hashVal += key.charAt(i);
    return hashVal % tableSize;
}
```

Advantages:

1. Uses all the available information.
2. Simple to calculate.

Disadvantages:

1. Returns same value for words like “bat” and “tab”.
2. Limited to values between 0 and 127*key.length % tableSize.
It’s better to slide the contributions of characters over by multiplications by a prime (say 31, this is what the Java hash function does. Other primes are OK too):

```java
public static int hash(String key, int tableSize)
    int hashVal = 0;
    for(int i=0;i<key.length;i++)
        hashVal += 31*hashVal + key.charAt(i);
    hashVal %= tableSize;
    if(hashVal < 0) hashVal += tableSize;
    return hashVal;
}
```
Some powers of 31 exceed the top end of an int: $31^7 > 2G =$ maximum int value – overflow.

You could replace the 31 with another prime, but not another number with factors of 2 or other small primes in it.

Similarly, avoid 31 as a table size!
Example: Employee record containing first name, last name, SSN, address, dept . . . .

We hash by SSN. They have max $999 - 99 - 9999 = 999,999,999 < 1G$, so they fit nicely in 32-bit numbers.

For hashCode(), just return the int SSN, and for equals, compare int SSN’s, after first checking for null.

object with a String id – just use String.hashCode().
We are assuming these id’s are unique id’s.

Another source of unique ids, if the data is coming from a database, are the primary key values, since they are guaranteed unique by the database.

If using firstname, lastname as id, with equals requiring both to match, we could concatenate the two Strings and then do hashCode(), or add up the two hashCodes or XOR them.
A collision happens when two keys hash to the same bucket, i.e., the same hash-value. What to do?

1. Separate chaining. Make the hash table an array of linked lists.
2. Closed hashing. If the first spot is full, use another spot in the hash table
   - 1. linear probing: look in the next spot down/wrapped in the array.
   - 2. quadratic probing: look in quadratically-determined places in the array.

There are other ways we won’t cover.
Separate Chaining

- Each hash array element is a linked list holding all the keys that hash to that bucket – all the collision participants.
- No further probing is needed, just list operations.
- This is the simplest way to make a hash table that works decently.
- Many hash tables work this way.
- In some cases separate chaining may be a little slower than quadratic probing, because it causes memory references that hop around in memory more.
- This could be important for very large hash tables.
Separate Chaining

```
0 → 0
1 → 1
2 → 2
... →...
N → N
```
Assuming a good hashing function:

**Question:** For $N$ keys and $M$ buckets, what is the average run time for lookup?
Linear Probing

- If \( b = h(x) \) is already in use, try \( b+1 \), then \( b+2 \), etc., wrapping around to \( b = 0 \) if you hit \( M \), the table size. \( b = (b + 1) \mod M \).
- As soon as you find an empty spot, take it.
- On lookup, hash the key and check if it matches the one in the hash slot, if not, try the next (wrapping if necessary), etc., until you find the matching one or an empty one.
- The performance isn’t great, because stretches of the array get filled up and probes have to go further and further.
- Also, you can’t delete entries in a simple way, just removing them, because they have been used as stepping stones in other key’s probing sequences.
Quadratic Probing

- Quadratic probing fixes the local-fill-up problem with linear probing.
- We assume $M$ is prime.
- Instead of plodding one by one through the array from the original bucket, jump by bigger and bigger steps: $b + 1, b + 4, b + 9, \ldots b + i^2$.
- Wrap around to $b = 0$ if necessary.
- Because of jumping further and further, the clustering effect is lessened.
Collisions: Various Solutions

- General rules of thumb:
  - Keep hash tables no more than half full, for good performance.
  - Rehashing: every time the data doubles in size, double the hash table size to keep under the half-full rule (rehashing).
    - Turns out you can still maintain $O(1)$ lookup.
  - Use a prime for the hash table size.