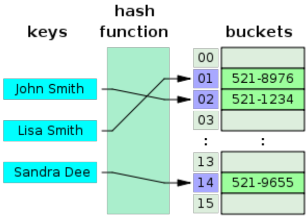
CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 6

February 18, 2014

Hashing – Quick Reminder

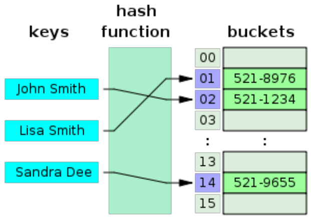
- A quick way to do lookup: O(1) insert, delete and find.
- A hash table is a fancy array of "buckets" containing the data.
- The hash function maps key values to array entries.



From Wikipedia

Hashing – Quick Reminder

- 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-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".
- h(x) is a computed mapping and is expected to take O(1) computation time.
- **Collision:** when two keys x and y hash to the same bucket: b = h(x) = h(y).

Hashing Special Strings

- Get $O(1)$ lookup performance by computing a perfect hash
- Consider id's A1, A2, ... A9, B1, B2, ..., B9, C1, ..., C9, D1, ..., D9
- Easy to map $'A' \to 0,' B' \to 1,' C' \to 2,' D' \to 3$ by $num1 = ch1 -' A'$.
- Also $'1' \to 1,' 2' \to 2, \ldots '9' \to 9$ by $num2 = ch2 -' 0'$;
- Here num1 goes from 0 to 3, num2 goes from 1 to 9, so 36 cases in all.

Hashing Special Strings

- Combine them into one number that goes from 0 to 39.
- Use code = num1*10 + num2 (or num1 + 4*num2)
- So the combo maps one-to-one with $code = 10 * (ch1 -' A') + ch2 -' 0'$

$“A1'' \rightarrow 10 * ('A' -' A') +' 1' -' 0' = 1, “A2'' \rightarrow 2, \ldots “A9'' \rightarrow 9,$

$“B1'' \rightarrow 10 * ('B' -' A') +' 1' -' 0' = 11, \ldots “B9'' \rightarrow 19,$

...

$“D1'' \rightarrow 31 \ldots . “D9'' \rightarrow 39$

Finding the Perfect Hash

- We have mapped all these patterns into $[1, 39]$, with all-different hash function values.
- Some hash function values aren't used, but that's OK, since there aren't a lot of them.
- We can use these range values as unique id's of these strings and we have a perfect hash, so a simple array can be used as a mapping table as we did with the vowel count problem.
- Suppose we have part names A123, B456, up to C999.
- We could find a computed mapping from these to the numbers 0 to 2999 with no holes (a "one-to-one" mapping).
- This would be another perfect hash!

Finding the Perfect Hash

- Another example of a set of special strings is the set of keywords for a computer language.
- As the compiler runs, these keywords (if, for, while, static,. . . , for C), are a fixed set that get accessed over and over.
- A compiler writer might find a perfect hash for these strings by experimenting with different formulas.
- Note: in Java, we usually don't worry about finding a perfect hash, since the non-perfect ones work so well, and we have HashSet and HashMap to do all the collision handling for us.
- Only if we need extremely high performance: nothing beats a simple array!

Hashing More Complex or Large Objects

- Graphics bitmaps are sometimes hashed to identify and classify them – think of them as strings with binary codes.
- Complex objects like classes often have an identifier in them, and that is what is hashed.
- Hashing implements fast look-up, so we only want to hash the things we want to look up by.
- Note that graphics bitmaps often don't have a natural identifier, so we use their contents to id them for want of a better method.

Hashing More Complex or Large Objects

- 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 SSNs, after first checking for null.
- object with a String id – just use String.hashCode().

Hashing More Complex or Large Objects – Example

- 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.

Collisions:Various Solutions

- 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.

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) % 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.

Is This Any Better?

- We need to show that these sequences dont start cycling around some small set of values.
- Suppose they did. Then, for M a prime table size:

$$\begin{aligned} &b + i^2 = b + j^2 \% M \qquad \text{for some i and j, different probe numbers.} \\ &i^2 = j^2 \% M \\ &i^2 - j^2 = 0 \% M \\ &(i - j)(i + j) = 0 \% M, \end{aligned}$$

- (i - j)(i + j) is divisible by M, a prime, i.e. the prime factors of (i - j)(i + j) contain M.

Is This Any Better?

- Any number can be written uniquely as a product of primes.
- if the product of two numbers contains a prime factor of M, one of the two factors must contain it.
- Thus we have the conclusion that (i - j) or (i + j) has a prime factor of M.
- If we restrict ourselves to M/2 probes, so $i \leq M/2$ and $j \leq M/2$, neither of these can happen.
- If we keep the hash table no more than half full (a good idea anyway), we can always insert another key using quadratic probing.
- With up to M/2 different probes to work with, weve got to find an empty slot!

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.

HashSet Implementation

- Chapter 20 implements a HashSet<T> by quadratic probing.
- HashSet can be a basis for HashMap with small changes.
- Also, HashSet is a useful collection class by itself.
- HashSet has an array of HashEntry, which is an internal class of hashed object type.
- Class definition:

```
public class HashSet<T> extends
AbstractCollection<T> implements Set<T> { ...
```

HashSet Partial Implementation

```
public class HashSet<AnyType> extends
AbstractCollection<AnyType> implements Set<AnyType>
{
    ...// Class implementation here
    private static final int DEFAULT_TABLE_SIZE = 101;
    private HashEntry [] array;
}
```

- The array is the actual table.
- It is private because it is an internal implementation detail external users shouldn't care about or access.
- The access is handled through API functions – contains, add, remove, findPos etc.

The Inner Data Structures

- HashEntry is an element (set element) plus a flag “isActive”, so that entries can be marked deleted.
- The HashEntry class is a private static class of HashSet. Its definition is inside the HashSet class definition:

```
public class HashSet<AnyType> ...  {
   private static class HashEntry {
   ...
   }
}
```