CS310 – Advanced Algorithms and Data Structures

Spring, 2021

Intro Hashing: Hash Functions
First, more on FrequencyCounter: Setting up a WordCounter service

- **FrequencyCounter** uses a HashMap<String, Integer> to store words and counts for them right in the one program
- Let’s split this up (“refactor it”) into two parts:
  - Top-level code (the main()), now FrequencyCounter1.java
  - A WordCounter service that does the detail work.
- Project 0 note: WordCounter, like LineUsage, has an instance variable of type HashMap<String, Integer>
Note on example

- Admittedly, this is not needed for such a simple program, but it shows how to do such a refactoring.
- It shows to use a Map<String, Integer> as an instance variable in a class to allow its object to hold significant information, easily.
- Note that pa0’s LineUsage uses such a map to hold, for one terminal line, all the usernames and counts for them.
What is the API between the parts?

• WordCounter will hold the HashMap<String, Integer> with all the details
• What does the top-level code need?
• It needs the Set of words and the count of each.
• So we have an API to invent…
What is the API between the parts?

• The top-level code needs the Set of words and the count of each. Also it needs a chance to input the file before providing this info.

• We can use a constructor to do the setup part. It needs something like System.in (seen in FrequencyCounter.java) to get access to the incoming data.

• What is System.in? It’s an object named “in” of class System, itself in the JDK. It has type InputStream.

• So we need an object of type InputStream, and then we can read from it using Scanner just as before.
What is the API between the parts?

- The top-level code needs the Set of words and the count of each. Also it needs a chance to input the file before providing this info.
- We need a constructor that accepts an InputStream and uses that input to fill a Map.
- We need a method to return the Set<String> for all the words
- We need a method to return a count for a word.
What is the API between the parts?

• We need a constructor that accepts an InputStream and uses that input to fill a Map.
  • public WordCounter(InputStream in)
• We need a method to return the Set<String> for all the words
  • public Set<String> getWords()
• We need a method to return a count for a word.
  • public int getCount(String word)
Simpler top-level code

```java
public static void main(String[] args) {
    int minLen = Integer.parseInt(args[0]); // key-len cutoff
    WordCounter wordCounter = new WordCounter(System.in);
    // find a key with highest frequency count
    int maxCount = 0;
    String maxWord = "";
    for (String word : wordCounter.getWords()) {
        int count;
        if (word.length() < minLen)
            continue;
        if ((count = wordCounter.getCount(word)) > maxCount) {
            maxCount = count;
            maxWord = word;
        }
    }
    System.out.println(maxWord + " " + maxCount);
}
```
Like LineUsage of pa0, WordCounter has a Map<String, Integer> inside it as an instance variable, private of course, to hold all the words/ usernames and their counts.

That means the class looks like this:

```java
public class WordCounter {
    private Map<String, Integer> wordCounts;
    //... constructor, methods
}
```
WordCounter API in code skeleton

```
// read from in, fill map
public WordCounter(InputStream in) {
    ...
}
// return the set of words
public Set<String> getWords() {
    ...
}
// return a count for a word
public int getCount(String word) {
    ...
}
```

Note that these differ from pa0’s LineUsage quite a bit. LineUsage takes in the data one element at a time.
WordCounter API in code skeleton

```java
private Map<String, Integer> wordCounts;
// read from in, fill map
public WordCounter(InputStream in) {
    ...
}
// return the set of words
public Set<String> getWords() {
    what goes here? // Needs to use the wordCounts map
}
// return a count for a word
public int getCount(String word) {
    what goes here?
}
```
private Map<String, Integer> wordCounts;
// read from in, fill map
public WordCounter(InputStream in) {
    // what goes here? Needs to load up wordCounts map from input
}
// return the set of words
public Set<String> getWords() {
    return wordCounts.keySet()
}
// return a count for a word
public int getCount(String word) {
    return wordCounts.get(word);
}
WordCounter Constructor

```java
private Map<String, Integer> wordCounts;
// read from in, fill map
public WordCounter(InputStream in) {
    wordCounts = new HashMap<String, Integer>();
    Scanner scan = new Scanner(in); // uses whitespace delim.
    while (scan.hasNext()) { // Build symbol table and count frequencies
        String word = scan.next();
        // if (word.length() < minLen) continue;
        if (!wordCounts.containsKey(word))
            wordCounts.put(word, 1);
        else
            wordCounts.put(word, wordCounts.get(word) + 1);
    }
    scan.close();
}
```

Same as original input loop in FrequencyCounter, except it’s dropping word-length filtering (see commented-out line above), and the map variable has a different name.
Hashing – quick intro

- 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 intro

• Hash function properties:
  1. Map key elements to integers.
  2. Fast to calculate.
  3. Try to minimize collisions – when different keys hash to the same value.
Hashing: uses array $O(1)$ lookup

- Hashing is a technique for fast lookup by key.
- It’s related to the idea of 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: $addr = \text{start-address} + \text{size-of-entry} \times \text{index}$, so $O(1)$ performance.
- This is a lookup in $O(1)$ time. put is also in time $O(1)$, as is remove (set $a[i]$ to null).
Hashing – less trivial examples

- 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 objects to integers with a more complicated function called the hash function.
- The hash function computes the “bucket number”, itself an array index, and the compiled code will find the array element by calculating:
  $$\text{addr} = \text{start\_address} + \text{index} \times \text{size\_of\_entry}$$
Simple Example of hashing.

- We have a map of int to int with 4 -> 100, 55 -> 44, 10 -> 12
  - Here 4, 55, and 10 are the keys.
- The hash function is \( h(x) = x/10 \), for hashing the keys.
  - So \( h(4) = 0 \), \( h(55) = 5 \), \( h(10) = 1 \) --bucket numbers 0, 5 and 1
- 4 hashes to bucket 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) --bucket 0 for key 4
  - \( a[1] = (10, 12) \)
  - \( a[2] = \text{null} \)
  - ...
  - \( a[5] = (55, 44) \)
Example of hashing (cont.)

- Look up 55: \( h(55) = 55/10 = 5 \), \( a[5] = \text{ref to (55, 44)} \), 55 matches, so value = 44
- Look up 56: \( h(56) = 56/10 = 5 \), \( a[5] = \text{ref to (55, 44)} \), no match so value not there (value = null)
- 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.
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 computable expression (some formula) and is expected to take \( O(1) \) computation time.
- **Collision**: when two keys `x` and `y` hash to the same bucket: \( b = h(x) = h(y) \).
Implementing maps using hashing

- Example: Given a string, count the occurrence of the 5 English vowels, using map from chars to ints.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
</tr>
<tr>
<td>i</td>
<td>0</td>
</tr>
<tr>
<td>o</td>
<td></td>
</tr>
<tr>
<td>u</td>
<td></td>
</tr>
</tbody>
</table>

‘a’ → count of a’s
‘e’ → count of e’s
...
‘u’ → count of u’s

Recall FrequencyCounter’s similar key → count Map
Vowel example

String s = "this is a test"; // string to count vowels in
// set up HashMap for stats
Map<Character, Integer> stats = new HashMap<Character, Integer>();
// with 5 put's, add ('a',0) ('e', 0), ... to stats here
for (int i=0; i < s.length(); i++) {
    c = s.charAt(i);
    Integer count = stats.get(c);
    // get Integer object, so can test if null
    if (count != null) // if vowel - found in map
        stats.put(c, count + 1); // count is auto-unboxed, then
        // 1 is added, result boxed up
}
print "a's: " + stats.get('a')  a's: 1
print "e's: " + stats.get(e') e's: 1
... i's: 2
  o's: 0
  u's: 0
Maps using hashing – vowel example

- How does HashMap work in this case?
- The Character class has hashCode() already implemented and in use for the code on the last slide. But how does it work?
- In homework 1, we saw that x.hashCode() = x for Integer
- x.hashCode() = x for Character too, and that value is the ASCII code.
- 'a' = 97 = 0x61, 'e' = 101 = 0x65, 'i' = 0x69, 'o' = 0x6f, 'u' = 0x75
- Use M=16, the default table size of HashMap and h(x) = x%M, as done in HashMap. Then x%M = x%16 is just the last hex digit of x.
  So, inside HashMap: h('a') = 1, h('e') = 5, h('i') = 9, h('o') = 0xf, h('u') = 5.
- One collision! Not wonderful, but not too bad.
Rough picture of HashMap holding Map of Characters to Integers: each bucket has a list of pairs
Maps using hashing

• The hash table itself is hidden inside the HashMap implementation.
• Note: there might be collisions in the HashMap. It’s OK, though, because HashMap takes appropriate action.
• hashCode() for key type: Only needs to provide an int. HashMap will scale it to the right array size.
• hashCode() vs. h(): hashCode yields an int, h() yields a bucket number, so h(x) = x.hashCode() % M for HashMap/HashSet with hash table of size M.
• Rule of Thumb: Try for only half-full (or less) hash tables to minimize collision on one hand and save space on the other hand. HashMap automatically grows its table as needed.
Hashing 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!
Somewhat bad hashing function

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

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

Problems:
1. Returns same value for words like “bat” and “tab”.
Hashing Strings: Java’s way

• It’s better to slide the contributions of characters over by multiplications by a prime (say 31), as done by Java in `String.hashCode()`.

```java
int hash = 0;
for(int i=0; i<key.length(); i++)
    hash = 31*hash + key.charAt(i);
```

• Then inside HashMap:

```java
hashVal %= tableSize;
hashVal &= 0x7fffffff; // make positive
```

✓ Note on S&W pg. 479 says we can’t use `Math.abs` here because it can return negative!

✓ Code on pg. 460 uses `%tablesize` during loop, but Java doesn’t (in `hashCode()` for String). It’s HashMap itself that does the `%tableSize`, modifying the `hashCode` it gets from the key.
Hashing Strings

• Some ordinary powers of 31 exceed the top end of an int: $31^7 > 2G = \text{maximum int value}$.
• This can cause an overflow in the CPU register doing the work.
• Luckily an overflow does not stop the computation in the CPU, which continues to have the 32 least significant bits of the exact answer.
• 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 tablesize if using this hash function!
FYI: Hashing Special Strings

How to get O(1) lookup performance by almost magical means

Consider id’s A₁, A₂, ... A₉, B₁, B₂, ..., B₉, C₁, ..., C₉, D₁, ..., D₉

Also ‘1’ -> 1, ‘2’ -> 2, ... ‘9’ -> 9 by num₂ = ch₂ – ‘0’;

Here num₁ goes from 0 to 3, num₂ goes from 1 to 9, so 36 cases in all. How can we combine them into one number that goes from 0 to 35, or 0 to 39?

Answer, use code = num₁*10 + num₂  (or num₁ + 4*num₂)
i.e., code = 10*(ch₁ – ‘A’) + ch₂ – ‘0’

Then each id maps to a unique number: that’s a “perfect hash”:
“A₁” -> 10 * (‘A’ – ‘A’) + ‘1’ – ‘0’ = 1, “A₂” -> 2, ... “A₉” -> 9,
“B₁” -> 10 * (‘B’ – ‘A’) + ‘1’ – ‘0’ = 11, “B₂” -> 12, ... “B₉” -> 19, ...
“D₁” -> 31. ... “D₉” -> 39
FYI: 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.

• We have a perfect hash, so a simple array can be used as a mapping table, since no collisions occur.
  
  • For example, to map “B2” to 100, set a[12] = 100

• Note: in Java applications, 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 larger objects

• Graphics bitmaps are sometimes hashed to identify and classify them—think of them as strings with binary codes.

• Complex Java objects 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 larger objects.

- Example: Employee record containing first name, last name, SSN (or empid, assigned), 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() of Employee, just return the int SSN, and for equals, compare int SSNs
- object with a String id—just use String’s hashCode().
Hashing more complex or larger objects - Example

- Another source of unique ids, if the data is coming from a database, are the database “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 hashCode()s or XOR them.
- Java7 new feature: `Objects.hash(firstName, lastName)` computes a hashCode from instance variables firstName and lastName. See S&W pg. 478.