CS310 – Advanced Algorithms and Data Structures

Class 7 Intro PA1, Hashing
Quick intro to pa1

• Parts: Xref, spell checker (like Xref) - All involve Maps.
• Xref: download sources, along with Tokenizer, from Weiss’s site, linked from the class web page.
• Note that Xref is covered in Weiss Chap 12, sec. 12.2, and Tokenizer in Chap 11, Sec. 11.1.2, but the code should be readable if you understand Maps.
• The Tokenizer finds identifiers, including Java keywords, in a Java source code.
• API to get Java identifiers, not looking inside comments or quotes.
  – Actually, it only knows a little Java: comments and quotes.
Trivial example, add this as Tokenizer’s main:

```java
Tokenizer tok = new Tokenizer(new InputStreamReader(System.in));
    String token;
    while ((token = tok.getNextID())!= null)
        System.out.println(token);
```

**Input:**

```java
hi this is fake Java // with comment syntax
and "quoted stuff" /* and internal comment */ so99!
```

**Output:**

```java
hi
this
is
fake
Java
and
so99
```
Quick intro to pa1 (cont.)

For a real Java example, the ids found by getNextID() are underlined here:

```java
import java.util.Map;
// Xref class interface: generate cross-reference
/**
 * Class to perform cross reference
 * generation for Java programs.
 */
public class Xref
{
    /**
     * Constructor.
     * @param inStream the stream containing a program.
     */
    public Xref( Reader inStream )
    {
```
Quick intro to pa1 (cont.)

- Note how both kinds of comments are skipped.
- When the Tokenizer sees // or /* it reads right through to end-of-line or */ without returning anything. Make sure you understand how it does this.
- The Tokenizer also keeps track of what line it is currently processing, so after you call getNextID() and get “import”, you can call getLineNumber() and get its line number.
- A valid Java ID is defined (pg. 7-8) as a letter or underscore followed by characters that are letters, underscores, or decimal digits.
- Xref uses Tokenizer, in particular getNextID and getLineNumber.
Spell checking a text file

- Xref uses Tokenizer, in particular getNextID and getLineNumber. This is not what you want for the spell-checker.
- For the spell-checker, we want “words” from the input text and Tokenizer as-is ignores comments.
- The simplest definition of a word is just a consecutive sequence of letters. You can defend a fancier definition (apostrophes etc.).
- Although you can use ideas from Tokenizer, don’t try to reuse it in the spell-checker. Use Scanner.
Maps of Lists

• Look at code in Xref.java:

```java
Map<String, List<Integer>> theIdentifiers = new TreeMap<String, List<Integer>>();
```

• This is a map of Strings to Lists of Integers. Each string in the domain maps to a List<Integer> in the range.

• Note: we don’t need a concrete class for List here, for example, new TreeMap<String, ArrayList<Integer>>() XXX

• We only need the concrete class on the outside when we create this object. Inside <>, we only need the type. Later, when we create a List to put in this container, we’ll need to use “new ArrayList<Integer>” or “new LinkedList<Integer>”.

import java.util.Map;

// Xref class interface: generate cross-reference
/**
 * Class to perform cross reference generation for Java programs.
 */

public class Xref {

    public Xref( Reader inStream )

    ...

"import" --> (1)
"java"--> (1)
"Xref"--> (7, 10)  <-- this means the id "Xref" shows up on lines 7 and 10
...
Map<List> examples

To add “java” for the first time: it appears on line 1.
List<Integer> value = new ArrayList<Integer>();
value.add(1);
theIdentifiers.put("java", value);

Add “public” the second time, on line 10:
List<Integer> lst = theIdentifiers.get("public");
lst.add(10)

// done, it’s already in the Map!
Also: Lookup an ID, get a List, with get().
Access examples

• When we get a ref on the List with get, we are obtaining the “live” object inside the Map. Not a copy.
• So we don’t have to “put” the List back in the Map after changing it.
• What about equals/hashCode/compareTo here?
  • They are only needed for the domain type, here String, so the JDK has done all the work for us.
• That is very commonly the case: we map from some sort of simple ID in the domain to a more complicated value in the range.
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.

From Wikipedia
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: \( \text{addr} = \text{start-address} + \text{size-of-entry*index} \), 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 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 element by calculating:
  \[
  \text{addr} = \text{start	extunderscore address} + \text{index} \times \text{size	extunderscore of	extunderscore entry}
  \]
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.
  - \( h(4) = 0, \ h(55) = 5, \ h(10) = 1 \)
- 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) -> bucket 0 for key 4
  - \( a[1] = (10, 12) \)
  - \( a[2] = \text{null} \)
  - \( \ldots \)
  - \( a[5] = (55, 44) \)
Example of hashing (cont.)

- Look up 55: \( h(55) = 5 \), \( a[5] \) = ref to \((55, 44)\), 55 matches, so value = 44
- Look up 56: \( h(56) = 5 \), \( a[5] \) = 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 that a hash function is just a computed mapping of some keytype to array spots.
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: \( b = h(x) = h(y) \).
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

Recall FrequencyCounter’s similar key → count Map
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 H
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 \( h(x) \) from chars to integers from 0 up to some limit \( M - 1 \) (table size). We can use the ASCII codes of the chars.
- The Character class has hashCode() already implemented and in use for the code on the last slide. But how does it work?
  - \( h(x) = x \mod M \) would do the trick, where \( x \) is the ASCII code.
  - \((\text{int})'a' = 97, (\text{int})'e' = 101, 'i' \rightarrow 105, 'o' \rightarrow 111, 'u' \rightarrow 117\)
  - 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.
Maps using hashing

- Or is it luck? What's wrong with M=10? It's not a prime. 10 = 2*5. For some reason, the factors of M cause a lot of collisions, especially in biased samples.
- Try M=11. h(x) = x % 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.
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(): Only needs to provide an int. HashMap, etc., will scale it to the right array size.
• 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.
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!
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”.

2/17/2020
Hashing Strings

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

```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 pg. 479 says we can’t use Math.abs here because it can return negative!
✓ Code on pg. 460 avoids overflows by using %m on each pass.
Hashing Strings

- Some powers of 31 exceed the top end of an int: $31^7 > 2G$ = maximum int value – overflow.
- Luckily an overflow does not stop the computation.
- And the contribution of the term with $31^7$ still affects the lower digits of the result because 31 is not a power of 2, so that $(31^7)\%\text{tablesize}$ is non-0.
- 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 tableszie!
Hashing Special Strings

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

Computer mapping that gives a perfect hash

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 39?

Answer, use code = num1*10 + num2  (or num1 + 4*num2)

So the combo maps one-to-one with code = 10*(ch1 – ‘A’) + ch2 – ‘0’

Then

“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
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 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 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 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 larger 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 SSN’s, after first checking for null.
- object with a String id—just use String.hashCode().
Hashing more complex or larger objects - Example

- We are assuming these ids are unique ids.
- 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.
- Java7 new feature: Objects.hash(firstName, lastName) computes a hashCode from instance variables firstName and lastName. See pg. 478.
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. Linear probing. If the first spot is full, use another spot in the hash table
   • look in the next spot down/wrapped in the array.
   • We’ll look at demo at Princeton.
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, including HashMap in the JDK*

In some cases separate chaining may be a little slower than linear probing, because it causes memory references that hop around in memory more. This could be important for very large hash tables.

*In Java 8+, HashMaps may switch a bucket from a linked list to a tree structure if the linked list gets too long. This mitigates bad hash function use.
Separate Chaining Data Structure from S&W website and pg. 464

Hashing with separate chaining for standard indexing client
public class SeparateChainingHashST<Key,Value> {
    private int m;  // HT size
    private SequentialSearchST<Key, Value>[] st; // HT
    // --see pg. 375: singly linked list
    // ... in constructor, create the hash table: needs
    @SuppressWarnings({"rawtype", "unchecked"})
    st = (SequentialSearchST<Key,Value>[])) new SequentialSearchST[m];

    • We can’t directly create an array of generic type in Java. So we create the
      array of raw type and cast it to the generic type, which causes warnings. We can suppress those warnings.

    • Very ugly. Same ugliness shows up in JDK HashMap code. Can’t be avoided if you want the result to be a
generic type, as we definitely do.
Hashing with separate chaining: using array of LinkedList

public class SeparateChainingHashMap<Key, Value> {
    private int m;  // HT size
    private List<Map.Entry<Key, Value>>[] st; // HT
    // ... in constructor, create the hash table: need
    @SuppressWarnings({"rawtype", "unchecked"})
    st = (List<Map.Entry<Key, Value>>[]) new
        LinkedList[m];

    • Again, we can’t directly create an array of generic type in Java. So we create the array of raw type and cast it to the generic type, which causes warnings. We can suppress those warnings. Very ugly.
    • As usual, we use the interface type List for types except for the needed concrete class used with new. We won’t need any of the LinkedList-special methods here. Could replace LinkedList with ArrayList here.
public class SeparateChainingHashST<Key, Value> {
    private int m; // HT size
    private SequentialSearchST<Key, Value>[] st; // HT
    // --see pg. 375: singly linked list
    // ... in constructor, create the hash table: needs
    @SuppressWarnings({"rawtype", "unchecked"})
    st = (SequentialSearchST<Key, Value>[][]) new SequentialSearchST[m];
    // Now create the actual lists in the buckets--
    for (int i=0; i < m; i++)
        st[i] = new SequentialSearchST<Key, Value>();

    • Note that the book code has the raw type here, unnecessary. No array
S&W Hashing with separate chaining: code, pg. 465

```java
public class SeparateChainingHashST<Key, Value> {
    ...
    instance variables, constructor
    private int hash(Key key) {
        return (key.hashCode() & 0x7fffffff) % m;
    }
    public Value get(Key key) {
        return (Value)st[hash(key)].get(key);
    }
    Public void put(Key key, Value val) {
        st[hash(key)].put(key, val);
    }
    ...
}

• Note hash(key) is a private helper method
• Get and put both hash the key, use that hashval to locate the bucket in the hashtable, and thus its own little ST/map, ready for get or put.
• The cast in get can be dropped if that unnecessary raw type is fixed.
```
Hashing with separate chaining: using array of LinkedList

public class SeparateChainingHashMap<Key, Value> {

    private int m;  // HT size
    private List<Map.Entry<Key, Value>>[] st;  // HT

    // ... in constructor, create the hash table: needs
    @SuppressWarnings(\{"rawtype", "unchecked"\})
    st = (List<Map.Entry<Key, Value>>[]) new
        LinkedList[m];

    • With this setup, get and put have more work to do than we saw in the
      S&W code. After they locate the bucket, they need to search the list in it
      for the key. If it’s found, access the value. If not, fail on get or add to the
      list for put.

    • This search code is inside SequentialSearchST for the S&W code.
Linear Probing: pp. 469-474

- If bucket \( b = h(x) \) 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 can suffer because stretches of the array get filled up and probes have to go further and further. Need to resize aggressively.
- 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.
- Look at demo.
As on pg. 469, trace of HT load

![Trace of linear-probing ST implementation for standard indexing client](image-url)