CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 5

February 11, 2014

Database-Like Lookup

- Often in applications we use database-like data, a bunch of objects with the same fields, and one of them is an identifier (or key.)
- Example: A simple inventory system where part names are keys to inventory objects holding (name, quantity, bin#):
  ("pencil", 120, 42) 120 pencils in bin 42
  ("tape", 44, 11) 44 rolls of tape in bin 11
  ...
- Suppose the file inventory.dat looks like this:

```
120|42|pencil
44|11|tape
10|47|quadrille-pad
```

Reading Structured Data

- Similar to the qualifying exam, we see a pattern:
  - get a Scanner on the input file
  - use it to read lines
  - for each line, use split, or another Scanner based on the String line
- In the inventory example, one line of a file corresponds to one instance of InventoryItem.
- We said the file inventory.dat looks like this:

```
120|42|pencil
44|11|tape
10|47|quadrille-pad
```

Reading Structured Data

- The first call to nextLine() returns String line = "`120|42|pencil`".
- No end-of-line is included, probably because that varies by OS and so causes portability problems if included.
- `String parts[] = line.split("|");` But | is a special character to regular expressions, needs quoting with \
- `String parts[] = line.split(" \|");` But \ is a special char in Strings, needs quoting with \\
- `String parts[] = line.split("\\|");` This works: uses | as separator

Reading Structured Data

- Note: if we used a comma, it would be just line.split(“,”);
- parts[0] = “120”, ..., should check that parts.length == 3
- Or use another Scanner for the line string, with non-default delimiter:

```
Scanner lineScanner = new Scanner(line);
lineScanner.useDelimiter(‘‘\\|’’);
// same regular expression
int quantity = lineScanner.nextInt();
int bin = lineScanner.nextInt();
String name = lineScanner.next();
```

Database-Like Lookup

- Scanner and String.split() can handle any delimiter specified by a regular expression.
- Once this data is pulled into memory, our users want to look up facts, for example, how many pencils there are.
- **Query:** how many pencils?

Inventory Setup

```
 // This class contains all the information about
// a single inventory item.
public class InventoryItem {
  // constructors...
  public InventoryItem();
  public InventoryItem(String name, int bin, int qoh)
    { ... }
  public String getName() {...};
  public int getBin() { ... };
  public int getQOH() { ... };
  public void setQOH(int newQOH) {...};
  private String name;
  private int bin;
  private int qoh;
}
```

What is the best data structure to store items and conduct a search?

Inventory Example – Set of Items

1. Create Set: `Set<InventoryItem> invDB = new HashSet<InventoryItem>();`
   - Need to add an equals, hashCode method to InventoryItem for this, or leave both out.
2. Load the Set: For each line in the file, loop over:
   ```
   InventoryItem newItem = new InventoryItem(name, bin, qoh);
   invDB.add(newItem);
   ```
3. Lookup: For example, look up "pencil" by `invDB.contains(``pencil'');`.
   - No efficient way to pull out the whole record for pencil based on key-match.Only to know if it exists.
   - We could iterate over the whole Set, and find the match, but that's $O(N)$, and we want something faster.

Inventory Example – Map of Items

1. Create Map: `Map<String, InventoryItem> invDB = new HashMap<String,InventoryItem>();`
   - No need for equals/hashCode for InventoryItem, only for String, already there.
2. load the Map: loop over:
   ```
   InventoryItem newItem = new InventoryItem(name, bin, qoh);
   invDB.put(name, newItem);
   ```
3. Do query: how many pencils? $O(\log N)$ for TreeMap, $O(1)$ for HashMap:
   ```
   InventoryItem item = invDB.get(name)
   // (no cast needed, thanks to generic collections)
   ```

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: addr = start-address + size-of-entry*index.
- This is a lookup in O(1) time.put is also in time O(1), as is remove (set a[i] to null).

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: addr = start_address+index*size_of_entry

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] = null
- ...
- a[5] = (55, 44)

Example of Hashing

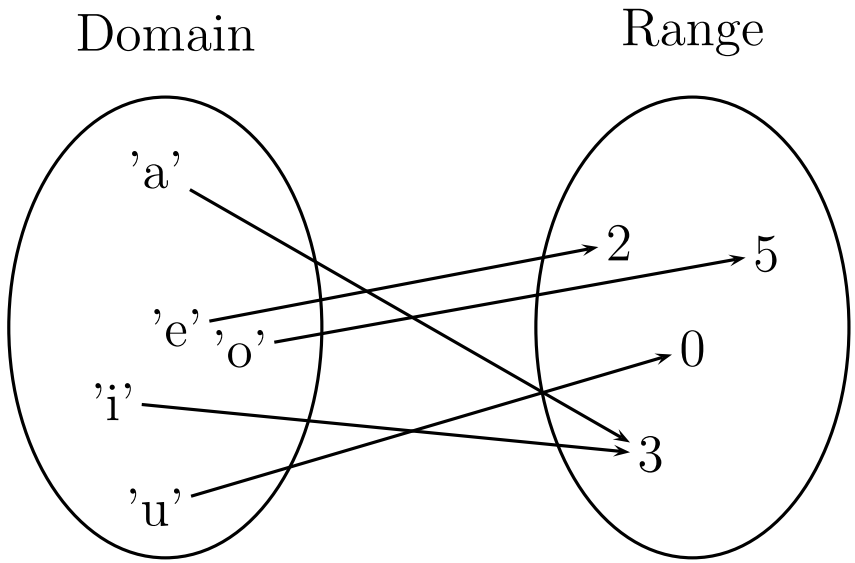
- Look up 55: h(55) = 5, a[5] = ref to (55, 44), 55 matches, so value = 44
- Look up 56: h(56) = 5, a[5] = ref to (55, 44), no match so value not there
- Luckily, the quick example has no collisions (two keys hashing to the same bucket).
- The above example is "hashing integers" .Similarly we can hash 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 $\rightarrow$ count of a's
- e $\rightarrow$ count of e's
- ...
- u $\rightarrow$ count of u's



Vowel Example

```
  String s = this is a test; // to count vowels in
// set up HashMap stats
Map<Character,Integer> stats = new HashMap<Character,Integer>();
// with 5 puts, 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 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%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%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.
- 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 hashing itself is hidden inside the HashMap implementation.
- Note: there might be collisions in the HashMap case, since were not taking control of the exact hash function.
- Its 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

- Strings of less 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!

Bad Hashing Function

```
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.

Hashing Strings

It's better to slide the contributions of characters over by multiplications by a prime (say 37):

```
public static int hash(String key, int tableSize)
{
  int hashVal = 0;
  for(int i=0;i<key.length;i++)
    hashVal += 37*hashVal + key.charAt(i);
  hashVal %= tableSize;
  if(hashVal < 0) hashVal += tableSize;
  return hashVal;
}
```

Hashing Strings

- Some powers of 37 exceed the top end of an int: $37^7 > 2G =$ maximum int value – overflow.
- However the contribution of the term with 37^7 still affects the lower digits of the result because 37 is not a power of 2, so that (37^7)%tablesize is non-0.
- You could replace the 37 with another prime, but not another number with factors of 2 or other small primes in it.
- Similarly, avoid 37 as a tablesize!

Quick Intro Into PA1

- Parts: Xref, redo qual, spell checker (like Xref)- All involve Maps.
- Xref: download sources, along with Tokenizer, from Weisss site, linked from the class web page.
- Note that Xref is covered in Chap 12, sec. 12.2, and Tokenizer in Chap 11, Sec. 11.1.2.
- The Tokenizer finds identifiers, including Java keywords, in a Java source code.
- API, pg 446 (not including constructor): get Java identifiers OR get brackets, not looking inside comments or quotes.
- Actually, it only knows a little Java: comments and quotes.

Quick Intro Into PA1

Trivial example, add this as Tokenizers main:

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

Input:

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

Output:

```
hi
this
is
fake
Java
and
so99
```

Quick Intro Into PA1

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

```
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 Into PA1

- 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, dont try to reuse it in the spell-checker. Use Scanner.

Map of Lists

- Look at pg. 498 in Xref.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>>();.
- 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>".

Example

```
 1 import java.util.Map;
2 // Xref class interface:  generate cross-reference
3 /**
4 * Class to perform cross reference
5 * generation for Java programs.
6 */
7 public class Xref
8
9 {
10 public Xref( Reader inStream )  ‘‘import’’ --> (1)
‘‘java’’--> (1)
‘‘Xref’’--> (7, 10)
  <-- this means the id ‘‘Xref’’ shows up on lines 7 and
10
...
‘‘public’’-->(7,10)
```

Access Examples

- Add "java" for the first time.

```
List<Integer> value = new ArrayList<Integer>();
value.add(1);
theIdentifiers.put(``java'',value);
```

- Add "public" the second time:

```
List<Integer> l = theIdentifiers.get(``public'');
l.add(10)
// done, its 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 dont 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.