CS310 - Advanced Data Structures and Algorithms

Fall 2013 – Class 9

October 1, 2013

Application – Movie Rental

- Movie Rentals, inspired by sign in a store: "If you liked X, try Y"
- Suppose we get our patrons to mark watched movies as one of two options: "liked movie" and "disliked movie".
- Then our media handler could enter this data into the customer-movie database.
- Movies have well-defined identifiers, movie titles, or title-year if needed.
- We want our computer to print out these prediction signs, based on our like-dislike data from our own customers.

Application – Movie Rental

- After a while we would have access to the movie like-sets and dislike sets of our customers.
- This is now happening in a lot of industries, and is summarized by the phrase "market of one" data.
- Looking from the movie angle, we have sets of customers who liked it and disliked it.
- The simplest analysis is simple movie popularity: popularity = Size(likeset)/[Size(likeset) + Size(dislikeset)].

Application – Movie Rental

- But what about these predictions from X to Y?
- In the population, some people liked both X and Y, some liked X only and some liked Y only and some disliked both.
- If the first and last of these are relatively large compared to the middle ones, we have correlation between them: they tend to go together, like or dislike.
- But we don't actually care about the dislike-X case for our prediction signs.
- We are interested in all the cases where people liked X – then what about Y?The chance that such a person will like Y is: probability(like Y | like X) = Size(like(X) $\cap$ like(Y)) / Size(like(X))

Application – Movie Rental

- Similarly if an individual customer wants a recommendation for a movie they haven't yet seen we could use these ratios to reach out from each movie that the customer has seen and liked and find the most predictably likable movie.
- We could also use dislike prediction as well. At some point we would need to bring in a real statistician!
- After a while our customers think we're great and really come to depend on our predictions.Then they become more predictable themselves.
- Although it is impossible to keep enough movies on hand to keep everything available, we now have a notion of a good substitute: if X predicts Y, then Y is a good substitute for X.
- Each movie has a substitute list based on some predictability level we've established.

How Do We Implement This?

- Each movie has a substitute set.movie$\rightarrow$ {substitute movies} i.e., a Map.
- We also have movie quantity-on-hand, its inventory.
- We can combine these into one master Map,using notation (x,y) for an object with fields x,y:
  movie $\rightarrow$ (qoh, {substitutes})
- Here the domain element is a movie, id'd by String movie title. The range element is an object with an int qoh and a Set of movies, i.e., Set of Strings.
- Pretty easy to work with since String comes with equals and hashCode, ready for HashMap.
- Implementation: `HashMap<String, InventoryData>` where InventoryData is a class which HASA Set<String> as well as an int "qoh".

Program: Counting "gaps" in our movie collection

- We don't have the movie or any of its substitutes.
- If a movie is all rented out, and all its substitutes are also all rented out, let's call it a "gap" in our collection.
- Given the master map M, count the gaps in the movie inventory.
- First try at an algorithm for this follows.

Implementation

```
 // Counting ‘‘gaps’’ in our movie collection:
// we don’t have the movie or any of its substitutes.
// First attempt
int gapCount = 0;
 loop through movies in M, a Map of String to (qoh, subs)
    movieData = M.get(movie); // look up movie in Map
     if (movie’s qoh == 0) {
     loop through movies subs, Set of Strings
         subData = M.get(sub);
        // look up subs info via Map
         if (subDatas qoh>0)
            // found substitute, stop searching
             break;
         if (no sub found) gapCount++;
 }
```

Set Oriented Implementation

- Note that in the above code we're searching through a set – that's often avoidable.
- We are trying to determine availability – we can make a Set of available movies and use that, or maintain these sets in the database along with the other data that supports.
- Here's an algorithm that first computes these availability sets and then uses them in the main computation.

Set Oriented Implementation

```
// Counting ‘‘gaps’’ in our movie collection:
// version 2 using more set-thinking
// first compute avail and unavail movie sets
 loop through movies in M, a Map of String to (qoh, subs)
     movieData = M.get(movie); // look up movie in Map
     if (movieData.qoh > 0)
         avail.add(movie); // compute set of available movies
     else
         unavail.add(movie); // and unavailable ones
 // second, check out unavail movies:
 // look at (their subs) intersect (avail movies)
 int gapCount = 0;
 loop through unavail Set of movie title strings
     movieData = M.get(movie);
     availSubs = movieData.subs INTERSECT avail
     if (availSubs.isEmpty())
        gapCount++
```

Comment

- We need to use retainAll to do the intersection, but retainAll modifies its own object, so we need a new Set to use as a temporary, throw-away Set here:

```
// calculate new set = intersection of subs and avail
 Set<String> availSubs = new
Set<String>(movieData.subs);
availSubs.retainAll(avail);// intersect subs and avail
if (availSubs.isEmpty())
// if none of the subs are available
    gapCount++;
```

- We have not yet done the full performance analysis to see if these examples in fact save time this way.
- Sometimes set crunching does pay off generously in applications.
- Also, some things are greatly simplified by this approach.

More on Inner Classes

- Handout "Example of a private inner class providing an iterator"
- For hw3, you tried out some uses of nested classes.
- Why do we need private inner classes?
- Answer: We could live without them, but they are convenient if the detail they represent has a close relationship to the outer class, and is suited to share the outer class type parametrization.
- For an iterator over HashSet<T> for example, we need an Iterator<T> object to return to the client, so we set up the private inner class HashSetIterator.
- But we need to return this Iterator object to the caller **outside HashSet**, so how can we restrict ourselves to a private inner class?

More on Inner Classes

- Answer: With the power of Java interfaces. The Iterator<T> interface is public. We define a private object of type that ISA Iterator<T>, and return it to the caller as an Iterator<T>.
- The caller never knows the real type of the object, and doesn't care either. It works fine as an iterator.
- Look at MyContainer, pg. 578(522). Not parametrized, but we can fairly easily fix it. See the handout. Add <T> to lines 3, 10, end of 14, and on next page, end of line 2.
- Change Object to T on lines 21 and 9 on the next page, and a cast to T, (T), on lines 22 and 10 on the next page.
- Note that LocalIterator is written two ways, the second one using the special relation an inner class has to the outer class fields.

More on Inner Classes

- It needs a "remove" method to implement the JDK Iterator<T> interface. See the handout.
- We added <T> to MyContainer's declaration, but not to LocalIterator. LocalIterator gets the parameter automatically because it's non-static.
- In fact, adding <T> to LocalIterator causes an unwanted parametrization shadowing the original one. Nasty bug. Need a good IDE to show warning right away.
- Now understand line 6, pg. 789(728) HashSetIterator: an inner class of HashSet<AnyType>, so itself parametrized by <AnyType>.

More on Inner Classes

- Note that inner class code can use outer class fields as easily as its own fields. Look at the line of code in LocalIterator's hasNext():
  current $<$ size
- Here current is a field of the inner class and size is a field of the outer class. It's that easy.
- This is implemented by the inner class having a "secret" reference to the outer class object that it must have. See pg. 577.

More on Inner Classes

- What about nested classes – can we refer to the outer object's fields?
- Yes, if we have a ref to the outer object, but that requires some programming on our part.
- For Book of SimpleStudent, we could add a field outerRef and invent a constructor for Book that set it.
  public Book (SimpleStudent ss) { outerRef = ss; }
- Then the SimpleStudent object code that created a Book could use Book(this) and get it set up.
- The Book code could then use outerRef.name to access the student's name. (Probably should call the outer ref "student" here.)

More on Inner Classes

- Don't forget that static inner classes don't get outer class type parameters–
- HashEntry is a static inner class, so lack of <> after HashEntry means it has no type parametrization.
- The code in HashEntry can use no outer-class parametric types, and has none of its own.

More on Inner Classes

- In general, take union of parameters of the inner class and its outer class:
  `public class Foo<X,Y> ...`
  `private class Bar<Z> ...`
  This means types X,Y, and Z are usable in Bar's code.
- Bar can implement or extend Mumble<X>– Mumble<X>'s API is followed–some types show up in the API, others are involved in implementation of Foo, or both.
- More challenging case – in TreeMap code, but useful for 5b, the challenge option of pa2.
- Cases involving multiple parametric types:
- pg. 753 (695) ViewClass<AnyType>, inner class of MapImpl<KeyType, ValueType>
- This class is parametrized 3 ways! That means the code in this class is allowed to use these 3 types.

TreeMap and HashMap Implementations

- We have been using HashMap for Maps so far, but sometimes TreeMap is a better fit.
- Both HashMap and TreeMap implement the Map interface.
- HashMap has slightly faster lookup (get), O(1) vs. O(log N) for TreeMap, but log(N) is very good.
- The TreeMap (and TreeSet) maintains its keys in an order, whereas hashing randomizes the keys.
- Thus we can avoid a sort by using a TreeMap in some cases.

Spell Checker Example

- Think about the SpellChecker example (pa1, q4).
- There no special order was specified in the report. So a TreeMap wouldn't help in that case.
- If we had asked for the final report to be in alphabetical order of misspelled words, then a TreeMap would keep the appropriate order for us, saving a sort at the end.

TreeMap and HashMap Implementations

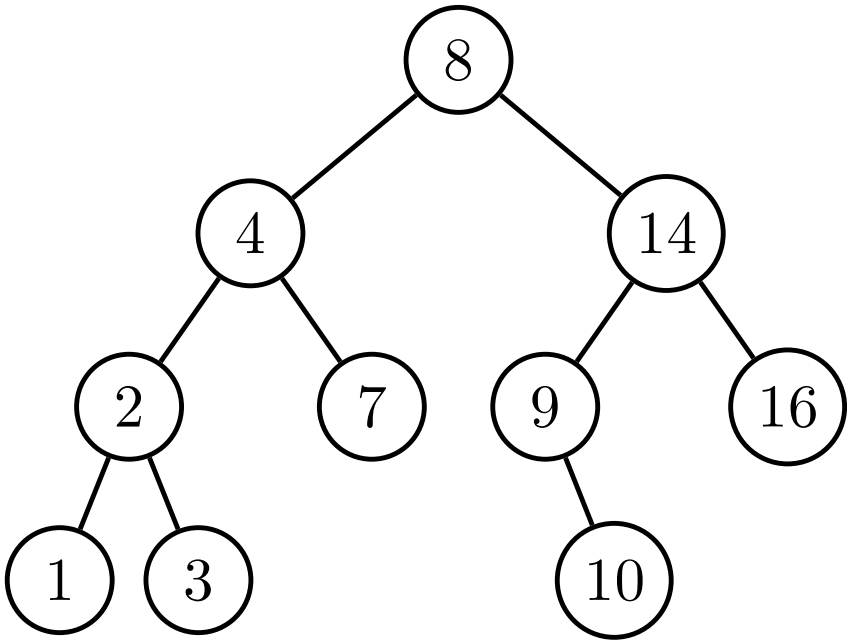
- You don't have to use the natural (compareTo) order of elements in a TreeMap.
- Instead, you can create a Comparator object to be used in the comparison, and pass it to the TreeMap constructor.
- A comparator takes two objects and returns 0 if they are equal, positive or negative value otherwise.
- Example: Strings in reverse order shown earlier in the course.

TreeMap and HashMap Implementations

```
public static void main(String[] args)
{     Set<String> s = new TreeSet<String>();
    // For reverse order:
    // TreeSet<String>(Collections.reverseOrder());
    s.add("joe");
    s.add("bob");
    s.add("hal");
    printCollection(s);
}
```

How Does a TreeMap Work?

- Basic idea: binary search trees, (fancier trees like AVL or red-black trees available).
- In the binary search tree the keys are held in the tree nodes and guided the lookup down the tree.
- A tree of N nodes has height about O(log N), and it takes about log N decisions to find the right leaf node.



Sets Using Bit Vectors

- Bit-vectors (also known as bitmaps) are a very simple and very fast implementation for an important kind of sets, that is, sets of moderate-size ints.
- We simply set up an array of bits in effect, bits numbered 0 to n-1, and each bit tells whether that number is in the set or not.
- This is really easy in Java, which supports array of boolean, but only slightly harder in C, where we can use an array of int and for each int, store membership info on 32 set elements, for 32-bit ints.
- IntSet implemented by Array of Boolean. Not a JDK Set, works with primitive arguments.

IntSet Implementation

- IntSet constructor: take in max size, make an array of Boolean of that size, all false.
- s.add(i): bits[i] = true;
- s.remove(i); bits[i] = false;
- s.contains(i): return bits[i];
-
```
void retainAll(IntSet b)
{
    for (int i = 0;i < NBITS;i++)
        bits[i] = bits[i] && b.bits[i];
}
```

Runtime Analysis

- This is O(N), but hopefully Java realizes it can be done 32 bits at a time, by CPUs bitwise AND.
- Clearly contains, add, and remove are O(1), while union, intersection, difference, min, and equals are O(N).
- As in the hash table analysis, we can assume the array sizes track actual set sizes in use.
- Thus we say these are O(n) where n is the set size, but we know they are "fast O(n)" because of the small constant.

Comparison of Different Methods

	Hashing	Bitmaps (IntSet only)	Tree	List
Construct	O(1)	O(N)	O(1)	O(1)
Contains, get, add, remove	O(1)	O(1)	O(log N)	O(N)
Union, intersection, difference	O(N)	O(N)	O(N)	$O(N^2)$
Next, nextKey	O(1)	O(1)	O(log N)	O(1)
Set Min, Max	O(N)	O(N)	O(log N)	O(N)

Limitations, Disadvantages, etc.

- Hashing: Assumes "good" hashing function, and rehashing, so the table can start small. Need to code hashCode for user-defined elements.
- When rehashing occurs, the add that caused it takes much longer than the other adds: the O(1) for add is an average.
- BitMap: Only for Sets of moderate-sized ints.
- "Fast" O(n) because 32 elements are processed together, small constant in T(n) formula.
- Trees: Requires the elements or domain elements have an ordering.Min/Max for Set is very fast, but that speed applies only to the ordering used in building the tree.

BitMap Implementation to More Cases

- another use of computed mappings to 0..n-1
- Since the bitmap gives us the best-looking union, intersection, and difference performance results, we would like to use it over more cases if possible.
- If we have a one-one computed mapping from an element type to the ints 0 to N-1, and its inverse as well, then we can map from e, an element, to its i, an int, use that as a bit number, and determine membership.
- To return an element, we use the inverse mapping to reconstruct e from i.
- Now we have enough info to do a performance analysis of a Set app.

BitMap Implementation to More Cases

- Recall Map of string to object movie (qoh, subs)
- subs = Set of all movies this one can substitute for, assume O(1):

```
for all movie in M.keySet():
    M.get(movie) to get (qoh, subs) :  O(1)
    Set<String> availSubs =
        cover-set INTERSECT avail
    if empty, gapcount++
```

- N movies, O(1) in each subs, INTERSECT with avail: needs O(1) contains, so O(1)
- isEmpty: O(1), so O(1) per movie, O(N) in all.
- So T(N) = O(N), assuming O(1) in cover-sets