CS310 – Advanced Algorithms and Data Structures

Class 8 More on Sets

Sets

- Consider set of integers: \( A = \{1, 5, 3, 96\} \), or \( B = \{17, 5, 1, 96\} \) and a set of strings, \{“Mary”, “contrary”, “quite”\}.
- Sets have no duplicates and order doesn’t count. (vs. List allows duplicates and order does count.)
- The Set API is just the Collection API.
- So Sets have methods isEmpty, size, add, contains, remove, clear, toArray, iterator
- We will see the Collection API in fact has more methods, but first look at basic OO considerations.

How does a Set hold its element objects?

Set of Integer objects: Each number here is in an Integer object, hanging off the set object by its ref:

Ref to HashSet object:
\[ a = \{1, 3, 5, 96\} \]

How does a Set hold its element objects?

In general, we may be able to change the element inside the set by using this “extra” ref, but in this case Integers are immutable, so with Integer (or String) elements this can’t actually happen.

Ref from variable:
\[ x = 5; \]
\[ a.add(x); \]
\[ a = \{1, 3, 5, 96\} \]

P.S. Same is true of elements of Lists, Maps, etc.

How does a Set hold its element objects?

Each element here is in an Integer object, with a ref that can be used in multiple places (“aliasing”):

Ref from variable:
\[ x = 5; \]
\[ a.add(x); \]
\[ a = \{1, 3, 5, 96\} \]

We programmers need to keep in mind that our Set elements can be shared. The encapsulation of Sets covers their handling of the their element refs, but not the element objects themselves.
More Collection/Set operations

- The Collection interface we’ve been using contains just the basic set operations add, contains, remove, etc.
- We would like to use Set Union, Intersection, and Difference.
- There are additional Collection methods that allow us to do these operations…

Basic set operations - reminder

\[ A \cup B = \{1, 3, 5, 17, 96\} \]
\[ A \cap B = \{1, 3, 5\} \]
\[ A - B = \{3\} \]
\[ B - A = \{17\} \]

Set operation examples: union

- Consider Sets made with HashSet:
  ```java
  Set<String> a = new HashSet<String>(){
  ...}
  ```
- Union of Set<String> a and Set<String> b: we can “addAll” of b to a or vice versa, to form the union set:
  ```java
  a.addAll(b) // a is the union of a and b
  b.addAll(a) // turns b into the union
  ```
- If we don’t want to change a or b, make a new set from one of them first:
  ```java
  Set<String> union = new HashSet<String>(a);
  union.addAll(b); // form a UNION b in own set
  ```

Intersection example

```java
a.retainAll(b);
// turns a into intersection of a and b
b.retainAll(a);
// turns b into intersection
Set intersection = new HashSet<T>(a);
intersection.retainAll(b);
// form a INTERSECT b in own set
```

Subset Containment example

- For subset containment, we just use containsAll: The following returns true if b is a subset of a:
  ```java
  a.containsAll(b);
  ```

Difference example

```java
Notice that A – B and B – A are different.
a.removeAll(b); // turns a into a – b
b.removeAll(a); // turns b into b – a
Set<T> diffAB = new HashSet<T>(a);
// form a – b in own set:
diffAB.removeAll(b);
// form b – a in own set
Set<T> diffBA = new HashSet<T>(b);
diffBA.removeAll(a);
```
Note: \((a - b) \cup (b - a)\) measures how different sets a and b are.
Implementation of union, intersection

**UNION via a.addAll(b):**
- Iterate through b, checking elements for membership in a, an O(1) op for HashMap, O(log n) for TreeMap. Any that aren’t there, add (by ref copy, as usual) to result set, also O(1)/O(log n). For N elements of b, O(N) operation for HashMap, O(NlogN) for TreeMap.

**INTERSECTION via a.retainAll(b):**
- Iterate through a, checking elements for membership in b. If not in b, remove it (O(1)/O(log n)). For N elements of A, an O(N) operation for HashMap, O(NlogN) for TreeMap.

How does it look with the element objects?
- In Java we allow many refs to one object, and when we call “new HashSet(a)” we are not copying the element objects but just their refs, a “shallow” copy.
- Then the addAll copies more element refs to the new set.
- Element matching is done via element.equals().
- And if we’re using HashSet, we need a consistent hashCode for the elements as well.

How does it look with the element objects?

Disjoint sets and their union:
- A={1,5}  
- B={17,96}  
- AUB={1,5,17,96}  

Sets with shared elements and their union:
- A={1,5}  
- B={1,17,96}  
- AUB={1,5,17,96}  

How does it look with the element objects?

Sets with equal but not shared elements and their union:
- A={1,5}  
- B={1,17,96}  
- AUB={1,5,17,96}  

Implementation of difference

- For a.removeAll(b) it is possible to be a little smarter.

```
if (a.size() < b.size())
    iterate through a, removing (from a) those that are in b
else
    iterate through b, removing those from a.
```

This way, only the smaller set is scanned.
Toy example revisited: vowels

- The original pseudocode said
  ```java
  if (c is a vowel)
  count it
  ```
- How did we code the condition here?
  ```java
  String vowelStr = "aeiou";
  Set vowels = new HashSet();
  for (int i=0; i<vowelStr.length(); i++)
    vowels.add(new Character(vowelStr.charAt(i)));
  ```
- Later:// c a character: O(1) lookup
  ```java
  if (vowels.contains(c))
  count++;
  ```

Of course for 5 items this is not important, but for large sets, it really matters.
- We have gone from an O(N) search to an O(1) lookup, a big savings.
- So when you see a loop of tests, think “could I use a Set here?”
- We just saw how set-thinking allowed us to replace an O(N) search with an O(1) lookup when we needed to classify something.
- But to see the full pay-off of set-thinking we need to use the set-crunching operations of union, intersection, etc.

Example: monitoring users

- Suppose we have 500 office connections, or “lines”, each line has a Set of usernames seen on the line, so this defines 500 sets of usernames.
- For example, the set of usernames (Strings) seen on line 1, set of users seen on line 2, etc.
- We could hold this all in an ArrayList<Set<String>> if we wanted.
- Or in an array of Set<String>, using the trick of casting from raw type to generic type:
  ```java
  Set<String> lines[] = new (Set<String>[]) HashSet[500];
  ```
- We have 500 sets of String usernames: users on line 1, users on line 2, etc.
- How do you find all users ever seen on any line?
  - Answer: take the union of all the sets.
- How do you find if anyone (or several users) logged in on all lines?
  - Suppose one line is for a public station suspected of break-in attempts. The set of users who logged in on that line are considered suspicious. How do you find all the lines that any of these users logged in on?

Application idea – movie rentals

- Netflix and other webapps offer movies for rent. Thinking of new features…
- Suppose we get our patrons to rate movies they have just seen: buttons for “liked movie” and “disliked movie”.
- Then our webapp could enter this data into the customer movie database. Movies have well-defined identifiers, movie titles, or title and year if needed.
- We want our webapp to output suggestions, based on our like-dislike data from our own customers, or choose the next movie for binge-watching.
- These suggestions are predictions of liking a certain movie Y, given knowledge of previous movie likes X.

Application – movie rentals

- 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:
  ```java
  popularity = Size(likeset)/[Size(likeset) + Size(dislikeset)].
  ```
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 predictions. 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 (statistically)

\[ \text{probability(like Y | like X)} = \frac{\text{Size(like(X) and like(Y))}}{\text{Size(like(X))}} \]

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 \( \rightarrow \{ \text{substitutes} \} \), i.e., a Map.
- We also have movie status, whether or not it’s available. We can combine these into one master Map, using notation \((x,y)\) for an object with fields \(x,y\):

\[ \text{movie} \rightarrow (\text{status}, \{\text{substitutes}\}) \]

- Here the domain element is a movie, id’d by String movie title. The range element is an object with an Boolean status 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, MovieData> where MovieData is a class which has a Set<String> as well as a status (true = available).

Program: Counting "gaps" in our movie collection

- If a movie isn’t available, 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: first attempt

```java
// Counting "gaps" in our movie collection: we don't have the movie or any of its substitutes.
int gapCount = 0;
for (MovieData movieData : M.values()) {
    if (!movieData.status) {
        for (String subst : movieData.substitutes) {
            if (M.get(subst).status) {
                break;
            }
        }
        if (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 (status, {subs})
movieData = M.get(movie); // look up movie in Map
if (movieData's status = true)
    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); // get subs
availSubs = subs INTERSECT avail if (availSubs.isEmpty())
    gapCount++;

Comments

• 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>(subs);
    availSubs.retainAll(avail); // intersect subs and avail
    if (availSubs.isEmpty()) // if none of the subs are avail
        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.

TreeMap and HashMap implementations

• We have been mostly 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).
• 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 features

• 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.
• Java 8 has easy Comparators, as we saw earlier when using a PriorityQueue.

TreeMap orderings for Set<Student>

// default: id order, since Student.compareTo uses id
Set<Student> s = new TreeSet<Student>();
// For reverse order: Java 8+
// TreeSet<Student>(Collections.reverseOrder());
// For order by gpa: Java 8+ for use lambda function
// TreeSet<Student>(Comparator.comparing(Student::getGpa));
s.add(new Student("Bob", 10, 3.2)); // name, id, gpa
s.add(new Student("Dave", 11, 3.0));
s.add(new Student("Jane", 12, 3.8));
System.out.println(s);

// Note: This Student.java is in PriorityQueue.zip
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.

Set 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

<table>
<thead>
<tr>
<th></th>
<th>Hashing</th>
<th>BitMap (IntSet only)</th>
<th>Tree</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Contains, get, add, remove</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(logN)</td>
<td>O(N)</td>
</tr>
<tr>
<td>union, intersection, difference</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N+2)</td>
</tr>
<tr>
<td>Next, nextKey</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(logN)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Set Min, Max</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(logN)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

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 resizing 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.
Bringing the 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.

Performance analysis for movie rental

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