Info Retrieval: Sets & Maps

Info Retrieval: Sets & Maps

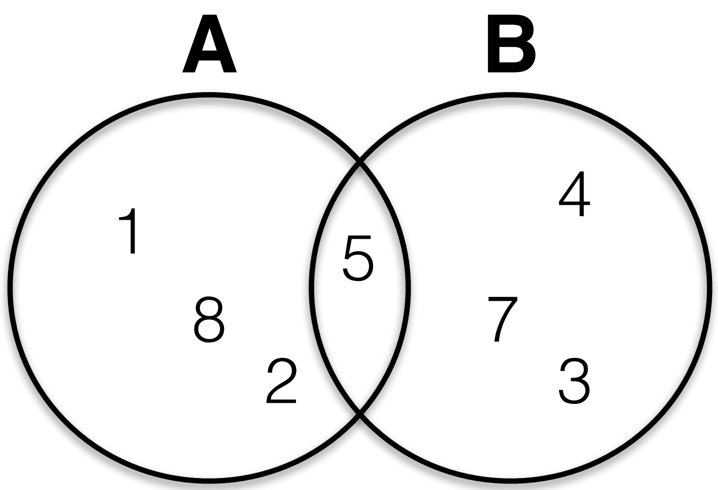
- Information retrieval entails searching through some large set of data ("database") to find things relevant to some query. — e.g., "***find 'good' in document***"
- It's a ubiquitous problem'
  - ***Google***, ***iTunes***, ***Facebook***, ***Twitter***, ***Amazon***, ***CNN***, the list goes on and on …
- We worked with "front-end" stuff when we designed our Flickr application
- **Q:** But how did the backend work? (i.e., actually fetching some images based on a query word)
  - Need to keep track of *all* information that could be retrieved…
  - In a way that makes retrieval relatively efficient…

  [Enter Sets & Maps]

  - we will talk about their APIs (Application Programming Interface), and start building up pieces of a simple info retrieval system!

Sets

Sets



- A set is just a collection of things.
- Set vs. List
  - List has a ***linear ordering*** of items.
  - Set has ***no order***.
  - List can have multiple occurrences of the same items (i.e., duplicates)
  - Set can only have one occurrence of a particular item
    - Q: without ordering, how can we distinguish copies of items?
- Main things we do w/ Sets:
  - Add items to a set
  - See what items are in a set
- If you think about the mathematical use of a set, you can then determine unions, intersections, etc.

$A = \{1, 8, 5, 2\}$
$B = \{5, 3, 7, 4\}$
$A \cup B = \{1, 8, 5, 2, 3, 7, 4\}$
$A \cap B = \{5\}$

Sets: Operations (formal ADT)

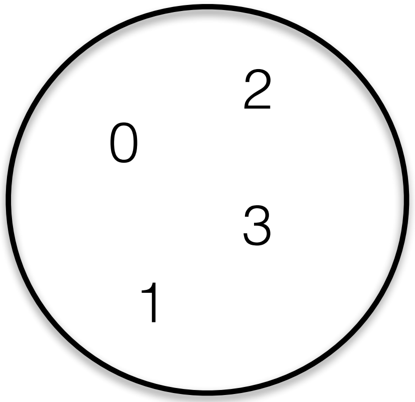
- **boolean add(E o)**
  Adds the specified element to this set if it is not already present. Returns true if it is a new addition.
- **boolean contains(Object o)**
  Returns true if this set contains the specified element.
- **boolean isEmpty()**
  Returns true if this set contains no elements.
- **Iterator<E> iterator()**
  Returns an iterator over the elements in this set.
- **boolean remove(Object o)**
  Removes the specified element from this set if it is present.
- **int size()**
  Returns the number of elements in this set (its cardinality).

Sets: Example

- Simple example:
    - Add some items; try to add some duplicates; check the Set.

```
Set<Integer> numbers = new TreeSet<Integer>();
numbers.add(0);
numbers.add(0); // noop
numbers.add(1);
numbers.add(1); // noop
numbers.add(2);
numbers.add(2); // noop
numbers.add(3);
numbers.add(3); // noop
System.out.println(numbers);                  // [0, 1, 2, 3]
System.out.println(numbers.contains(3)); // true
System.out.println(numbers.contains(4)); // false
```

numbers



Sets: Example

- **UniqueWords.java**
- Create a collection of all unique words
  - ex. compare to other files

```
String page = "... a very long string ...";
String[] allWords = page.split("[ .,?!]+"); // split on punctuation and white space

Set<String> uniqueWords = new TreeSet<String>();

// Loop over all the words split out of the string, adding to set
for (String s: allWords) {
    uniqueWords.add(s.toLowerCase());
}

System.out.println(allWords.length + " words");
System.out.println(uniqueWords.size() + " unique words");
System.out.println(uniqueWords);
```

- [Demo program in class]
- We might want more than just which words show up though — such as words and the number of occurrences of each of those words…

Maps

Maps =)

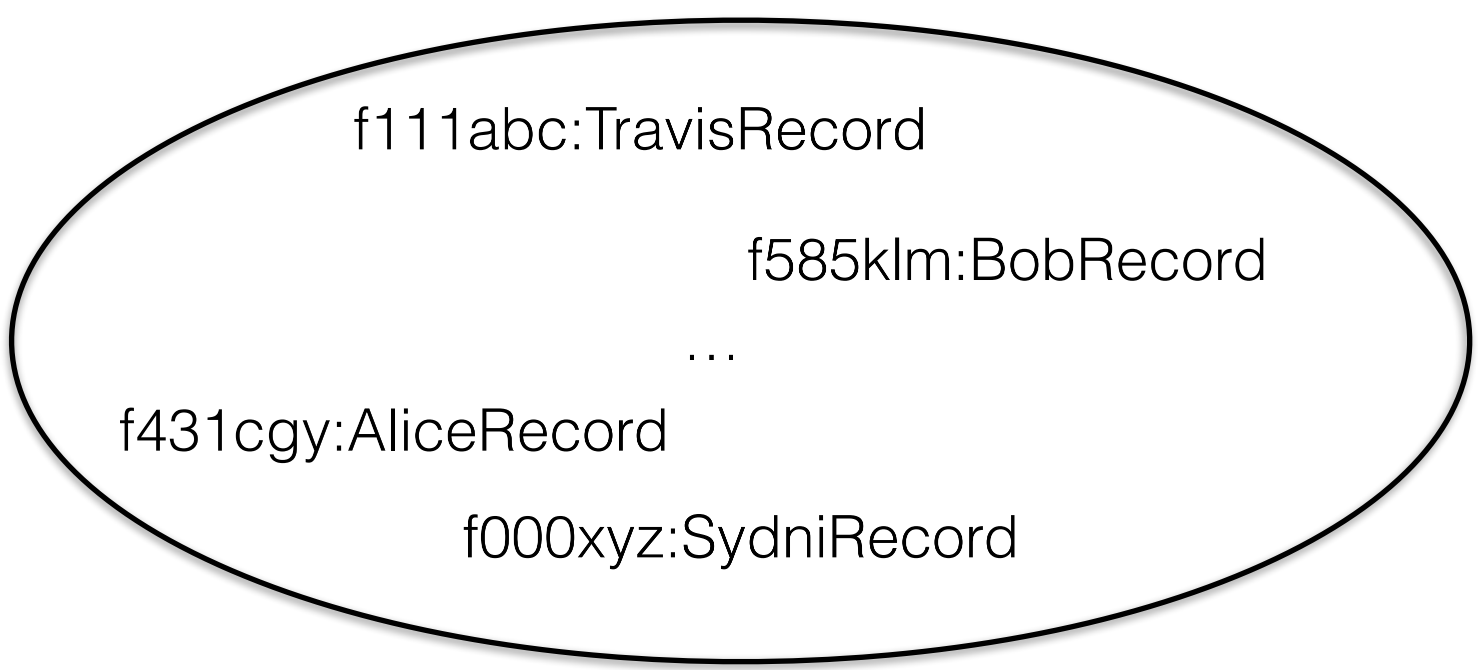
http://classes.bnf.fr/gif/ca-inde.gif

http://www.daylight.com/

Maps

- A map is similar to a map (unique *keys)*, but there is also a corresponding *value* — similar to key/value pairs in BST.
- Ex. student record database
    - key: student ID (i.e., String object)
    - value: student record (i.e., StudentRecord object)

studentDatabase



Maps: Operations (formal ADT)

- **boolean containsKey(Object key)**
  Returns true if this map contains a mapping for the specified key.
- **boolean containsValue(Object value)**
  Returns true if this map maps one or more keys to the specified value.
- **V get(Object key)**
  Returns the value to which this map maps the specified key.
- **boolean isEmpty()**
  Returns true if this map contains no key-value mappings.
- **Set<K> keySet()**
  Returns a set view of the keys contained in this map.
- **V put(K key, V value)**
  Associates the specified value with the specified key in this map.Returns the previous value associated with key, or null if key was not in the map.
- **V remove(Object key)**
  Removes the mapping for this key from this map if it is present. Returns the value associated with key (or null if key is not in the map).
- **int size()**
  Returns the number of key-value mappings in this map.

Maps: Example

- **UniqueWordsCount.java**
  - Collection of word/frequency pairs

```
String page = "... a very long string ...";
String[] allWords = page.split("[ .,?!]+");

Map<String,Integer> wordCounts = new TreeMap<String,Integer>(); // word -> count

// Loop over all the words split out of the string, adding to map or incrementing count
for (String s: allWords) {
    String word = s.toLowerCase();
    if (wordCounts.containsKey(word)) {
        // Increment the count
        wordCounts.put(word, wordCounts.get(word)+1);
    }
    else {
        // Add the new word
        wordCounts.put(word, 1);
    }
}
System.out.println(wordCounts);
```

- [Demo program in class]

Maps: Example

- **UniqueWordsPosition.java**
  - Collection of word/position(s)

```
String page = "... a very long string ...";
String[] allWords = page.split("[ .,?!]+");

// word -> [position1, position2, ...]
Map<String,List<Integer>> wordPositions = new TreeMap<String,List<Integer>>();

// Loop over all the words split out of the string, adding their positions in the string to the map
for (int i=0; i<allWords.length; i++) {
    String word = allWords[i].toLowerCase();
    if (wordPositions.containsKey(word)) {
        // Add the position
        wordPositions.get(word).add(i);
    }
    else {
        // Add the new word with a new list containing just this position
        List<Integer> positions = new ArrayList<Integer>();
        positions.add(i);
        wordPositions.put(word, positions);
    }
}
System.out.println(wordPositions);
```

- [Demo program in class]

Example: Search

Search

- **Search.java** — *very* simple program to search various shakespeare documents for words (ex. “love”, “death”, “witches”)
  - doesn’t handle plurals vs. singulars, verb tenses, etc.
- **Core idea:**
  - build an index, for each document, of the word counts
  - load documents from a file
  - we associate each word count map with its filename
    - i.e. {filename : {word : count}}
  - we also collect some other useful info:
    - total # words in each file,
    - total # of times each word appears,
    - total # of files in which each word appears.
- ***[Demo program in class]***

Search: CLI

- Command Line Interface == CLI
- Search for top/bottom (most/least frequent) *n* words
  - [# *n*] or [# -*n*]
    list the "top" *n* words or the "bottom" *n* words, respectively
  - [# *n filename.txt*] or [# -*n filename.txt*]
    list the "top" *n* words or the "last" *n* words for a particular document, respectively
- Search for # of occurrences of a particular word in all documents
  - [*word*]
    lists the file(s) the word appears in and how many times it appears
- Search for # of occurrences of some set of particular words in all documents
  - [*word1 word2 … wordN*]
    lists only the files that contain all the input words, how many times it appears, in how many files it appears, *and* show the TF-IDF.

Search: Notable Things…

- Search()
  - Standard constructor — initialize the various Maps we need.
- loadFile()
  - Given a filename, read the file, line by line, and build ***word:frequency*** map
  - Also record # of words in that particular file
- computeTotals()
  - Record the total # of occurrences of of each word (across all documents)
  - Record the total # of documents containing each word.

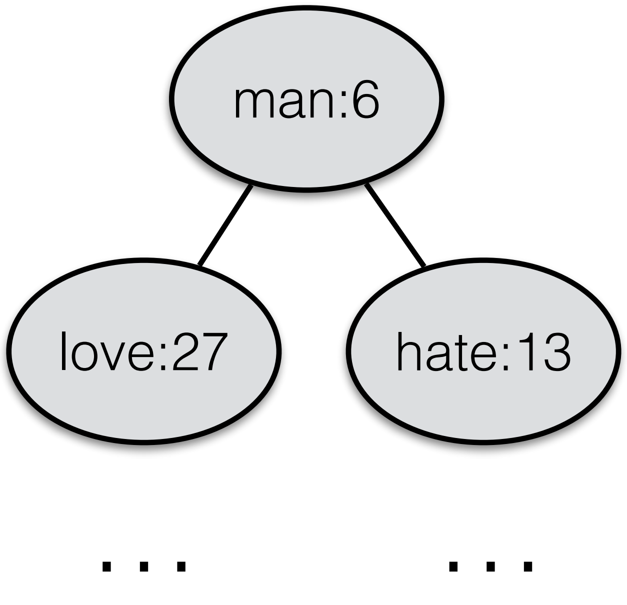
Search: Notable Things…

- printWordCounts()
  - compare() method for Comparator (anonymous class)
- search()
  - use retainAll() to compute the *intersection* of sets — only want files that contain *all* words when given a set of words to search for
- tfIdf()
  - the “tf-idf” stands for “term frequency - inverse document frequency”
  - tf == count the # of occurrences (i.e., frequency)
  - df == weight each term freq. by a notion of how common it is.
    - log(# documents total / # documents with the word)
    - word in all documents ==> weight == 1
    - word in one document ==> weight == log(N), N is the # of documents

Implementations

How do they do it?!

Implementations: Maps



- Maybe you are thinking about BSTs? :)
- Recall that the Map interface has a containsValue() method though, which our BST did not have…
  - **Option 1**: go through the entire tree, checking each node's value — O(n)
  - **Option 2**: keep values in another data structure; values are not necessarily unique
    - Using a **Set** lets us quickly answer query about if the Map contains a certain value, but if we replace/remove a value from the BST, we don't know if we should remove it from this other data structure (maybe some other key still has this value).
    - Maybe we need to use another **Map** to keep track of how many keys have this value? This gets us back to O(lg(n))…
  - **Option 3**: A List. Each element is a key/value pair. The list could be…
    - Sorted ==> quick lookup, slow insert.
    - Unsorted ==> quick insert, slow lookup.

Implementations: Sets

- **<u>Option1</u>**: A List. Again,
  - Sorted ==> quick lookup, slow insert.
  - Unsorted ==> quick insert, slow lookup.
- **<u>Option 2</u>**: Store items as keys in BST
  - don't worry about the values — all we need to know is if the item is in the set.
  - this gets us to O(lg(n))

This is all leading to the fact that we will see a different way to implement Maps and Sets using what are known as <u>Hash Tables</u> in a later lecture.