CS 10:
Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson
260 (255) Sudikoff

Day 10 – Info Retrieval

Agenda

1. Sets
2. Maps
3. Search

Sets are an unordered collection of items without duplicates

Set

- Model for mathematical definition of a Set
- Like a List, but:
  - Unordered (no 0^{th} item, can't set/get by position)
  - No duplicates allowed
- Operations:
  - `add(E e)` – adds `e` to set if not already present
  - `contains(E e)` – returns true if `e` in Set
  - `isEmpty()` – true if no elements in Set, else false
  - `Iterator<E> iterator()` – returns iterator over Set
  - `remove(E e)` – removes `e` from Set
  - `size()` – returns number of elements in Set

Trees are one way to implement Sets

Sets implemented with Trees

- Could implement as a List, but linear search time
- Trees are a natural way to think about implementation
- Given key, easy and fast to determine if item in list as in the `contains` method
- `add` must make sure duplicates are not allowed (Java documentation cautions that behavior is undefined if elements are mutable)
- Soon we will see another way to implement a Set using a hash table

We can use a Set to easily count unique words in a body of text

UniqueWords.java

- Pretend *page* was loaded from a web page
- *allWords* holds each word from page after tokenizing
- Loop over each word in *allWords* and add to Set *uniqueWords*
- Duplicates overwritten
- Print results

Agenda

1. Sets
2. Maps
3. Search

Maps associate a key with a value

Maps

- Python people think Dictionary
- Key is something used to look up a value (ex., student ID)
- Value could be an object (e.g., a person object)
- Operations:
  - `containsKey(K key)` – true if `key` in Map, else false
  - `containsValue(V value)` – true if one or more keys contain `value`
  - `get(K key)` – returns value for specified `key` or null
  - `isEmpty()` – true if no elements in Map
  - `keySet()` – returns Set of keys in Map
  - `put(K key, V value)` – store `key`/`value` in Map; overwrite existing
  - `remove(K key)` – removes `key` from Map
  - `size()` – returns number of elements in Map

Trees are one way to implement Maps

Maps implemented with Trees

- Could implement as a List, but linear search time
- Like Sets, trees are natural way to think about implementation
- Problem: no easy way to implement `containsValue()` (but `containsKey()` is easy!)
  - Could search entire tree for value,
    - Problem: linear time
  - Could keep a Set of values and search it
    - Problem: a value could be stored with multiple keys, so if delete key, can't delete value from Set
  - Solution: keep another Map with counts of values
    - When adding a value, increment its count
    - When deleting a key, decrement value count
    - Now have log time search for value (if tree kept balanced)

We can use a Map count how many times a word appears in a body of text

UniqueWordCounts.java

- Count how many times each word appears
- Pretend *page* was loaded from a web page
- *wordCounts* maps String (word) to Integer (count of each word)
- *allWords* holds each word from page after tokenizing
- Loop over each word in *allWords*
  - Check if word already in Map
    - True: increment count by getting value, then add 1
    - False: put word with count 1
- Print results

A Map can also contain a List associated with each key

UniqueWordPositions.java

- Count what position where each word appears
- Pretend *page* was loaded from a web page
- *wordPositions* maps String (word) to List of Integers (so we can have more than one integer per key)
- *allWords* holds each word from page after tokenizing
- Loop over each word in *allWords*
  - Check if word already in Map
    - True: add position *i* to List for this key
    - False: create new ArrayList, add *i* to it, store in Map
- Print results

The same concept can apply to reading data from different files

UniqueWordPositionsFile.java

- Same as uniqueWordPositions.java except reads from file
- *loadFileIntoString(filename)* returns text from *filename* into a string
  - Create BufferedRader *in*
  - Initialize *str* (accumulator) and *line*
  - Read filename line by line
    - Assigns line in the while loop expression (yuck)
    - Tests for null (end of file)
    - Appends line read onto *str*
  - Returns *str*

Agenda

1. Sets
2. Maps
3. Search

The same concept can apply to reading data from different files

Search.java

- Reads text from several Shakespeare works
- Create 4 four Maps:
  - *file2WordCounts*: filename->(map word->count)
  - *numWords*: filename-># words in file
  - *totalCounts*: word-> count over all files
  - *numFiles*: word -> # files containing it
- loadFile(File file) – fill *file2WordCounts* and *numWords* for each file
- *computeTotals() – fill totalCounts* and *numFiles* (must be done after all files have been loaded!)

The same concept can apply to reading data from different files

Search.java

- User interface
  - Type a word to see how many times it appears in each file (e.g., nay or love)
  - *# n* to get *n* most common words
  - Can restrict to just a single file with *# n (e.g., # 10 hamlet.txt)*
  - # -n to get the least common
  - printWordCounts() sorts and prints
    - Custom comparator, *WordCountComparator()* to sort entries to print
    - Looks at sign of number to print and gets top or tail of list

The same concept can apply to reading data from different files

Search.java

- User interface
- Can search for multiple words (forsooth and forbear)