

Exercise #7, Joongle

Due Wednesday, December 23rd, 14:00
2 weeks, not too long, no lab this Tuesday



Boaz Kantor
Introduction to Computer Science
Fall semester 2009-2010
IDC Herzliya

Background



- Consider the following text:
 - *“Hence, when able to attack, we must seem unable;*
 - *when using our forces, we must seem inactive;*
 - *when we are near, we must make the enemy believe we are far away;*
 - *when far away, we must make him believe we are near.”*

The Art of War / Sun Tzu

- Search for the word “near”

Sequential search



- Sequential search means iterating the collection
- The first instance of “near” is the 21st word
- Searching for “near” takes 21 instructions
- Searching for any word at index k takes k instructions.
- What if the text is 200,000 characters long?
- What if we have 25 Billion pages to search in with 100 words in average per page? We will then have $O(2,500,000,000,000)$. Efficient?

Building an index



- In order for a search to be faster, we can:
 - Sort the input (exercise 7)
 - Run advanced search algorithms (not part of the exercise)
- But should we sort 2.5 trillion words? NO!
- We create an index of words, where each word appears only once. This index should be much shorter than 2.5 trillion.

The art of index



Word

“Hence,”

“when”

“we”:

“to”

“attack,”

.

.

“near”

Reference (index in text)

1

2, 10, 18, 32

6, 14, 19, 22, 28, 35, 40

4

5

.

.

21, 42

Searching the word



- We can still search for our word sequentially:

```
public int getWordIndex(String word) {
    for (int i = 0; i < index.length; ++i) {
        if (index[i].equals(word)) {
            return index[i].getReferences();
        }
    }
    return ERROR_WORD_NOT_FOUND;
}
```

Complexity improvement



- Have we improved anything?
 - A bit. We reduced the size of text by removing duplicate words.
- Is this enough?
 - No.
- How about sorting the index alphabetically?
 - That's one way to improve.
 - We can then use binary search to find a word in $O(\log n)$
- So why not?
 - It would make the exercise too difficult... 😊

Indexing for sequential search



- Assume we have to search the word in sequential search. How can we improve our search efficiency?
- Word rank: a number, indicating the chances that a word will be searched.
- By sorting the index by Rank, sequential search should ultimately run in $O(1)$, in theory (why?).

WordRank



- A word rank is comprised of two factors:
 - Popularity: how many times, so far, has this word been searched for?
 - Prevalence: how many times does this word appear in our text?

$$\text{Rank} = (\text{prevalence} * 1) + (\text{popularity} * 10)$$

- Popularity is more important!
- However, the prevalence/popularity ratio may change in the future.
- A higher WordRank means the word is more likely to be searched for.
- In order to improve our sequential search, we can now sort our index by rank, in descending order.

Calculating ranks (first 9 words)



“Hence”

Prevalence: 1
Popularity: 0
Rank: 1
References:
1

“when”

Prevalence: 4
Popularity: 0
Rank: 4
References:
2, 10, 18, 32

“able”

Prevalence: 1
Popularity: 0
Rank: 1
References:
3

“to”

Prevalence: 1
Popularity: 0
Rank: 1
References:
4

“attack”

Prevalence: 1
Popularity: 0
Rank: 1
References:
5

“we”

Prevalence: 7
Popularity: 0
Rank: 7
References:
6, 14, 19, 22,
28, 35, 40

“must”

Prevalence: 4
Popularity: 0
Rank: 4
References:
7, 15, 23, 36

“seem”

Prevalence: 2
Popularity: 0
Rank: 2
References:
8, 16

“unable”

Prevalence: 1
Popularity: 0
Rank: 1
References:
9

Index, sorted by ranks



“we”

Prevalence: 7
Popularity: 0
Rank: 7
References:
6, 14, 19, 22,
28, 35, 40

“when”

Prevalence: 4
Popularity: 0
Rank: 4
References:
2, 10, 18, 32

“must”

Prevalence: 4
Popularity: 0
Rank: 4
References:
7, 15, 23, 36

“seem”

Prevalence: 2
Popularity: 0
Rank: 2
References:
8, 16

“to”

Prevalence: 1
Popularity: 0
Rank: 1
References:
4

“attack”

Prevalence: 1
Popularity: 0
Rank: 1
References:
5

“able”

Prevalence: 1
Popularity: 0
Rank: 1
References:
3

“Hence”

Prevalence: 1
Popularity: 0
Rank: 1
References:
1

“unable”

Prevalence: 1
Popularity: 0
Rank: 1
References:
9

Searching for a word



- Searching for a word increases its popularity.
- This, of course, changes the rank (remember? $\text{Rank} = \text{prevalence} + 10 \times \text{Popularity}$).
- For example, if we search for the word “when”, its popularity will be 1 and its new rank will be 14. that should put it right at the front of the index.
- If we then search for the word “seem”, it should move it up to the 3rd place in the index (why?). The next time we will search for it, it will move to the first cell in the index. Then **the next search for “seem” will be done in $O(1)$.**
- This means that after a word has been searched, we have to sort the index again.
- The idea behind this exercise is to improve our sequential search efficiency for words that are common and words that are searched for many times.

What you should do



- In order to “solve the exercise”, you will need to implement:
 - The index element, which is a class called “WordStats”. This class includes references to files and the statistics of that word (popularity, prevalence, rank, etc).
 - The index itself, which is an array of WordStats elements.
 - Sorting algorithms (in a utility class).
 - Something that the user can work with, which has both functionality to index a word and functionality to search for a word.

Where to begin

A hand holding a globe, symbolizing global reach or a starting point.

- The following instructions are good for each and every project you develop, in this semester and for all eternity.
- At each stage, make sure your code compiles.
 1. Start by creating files for all your classes. Fill these files with empty classes with the proper names. Add your data structures (instance variables).
 2. Create all the methods written in the API, only leave them empty. Add the following code to each method:

```
// TODO implement!
```

```
throw new UnsupportedOperationException();
```

the first line will be used by Eclipse to remind you where to implement your code. The second line will make sure that during runtime your code will not work without implementing the method.

Where to continue



3. Write a main method in each class. This method will be used to test your class. **Make sure to delete the main method before submitting your solution!**
4. Create a new object of your class (if possible), several objects (if needed) and use all the constructors the class has.
5. Call each method at least once, even if it's private. Try both valid parameters and invalid ones. Try to think of all the special cases your method should handle.
6. Add debug printing. This kind of printing should be different from other output messages (for example, prefix it with the word `*** DEBUG***:`). It should also include what is being tested, method results, variable names, etc.
7. Implement your methods one by one. Don't implement a method before the previous one has been tested successfully.

Where to end



8. To implement a method, make sure you understand what you want it to do. Write the algorithm in the form of code comments. Test it in your head. If you're sure it's ok, add code implementation behind each comment. DO NOT remove these comments. If they're good, they will probably serve you well as your final code comments.
9. I suggest that before transforming code comments to code, write those comments for all the methods. This will provide you a good picture of your solution, and will provide you with enough information to decide on additional private methods.
10. To debug your code, use both the output of your debug messages and the debugging features of Eclipse.
11. Once implemented and tested, go over all your literals. Find the ones that have a meaning and create a constant for them.

Polishing your solution



- If you followed all the instructions above, you will have a well documented and well designed working code. Congratulations!
- Now go over the entire solution and find mishaps, like missing constants, duplicated code (which requires additional private methods), missing documentation, styling errors, etc.

Good luck!!



Boaz Kantor
Introduction to Computer Science
Fall semester 2009-2010
IDC Herzliya