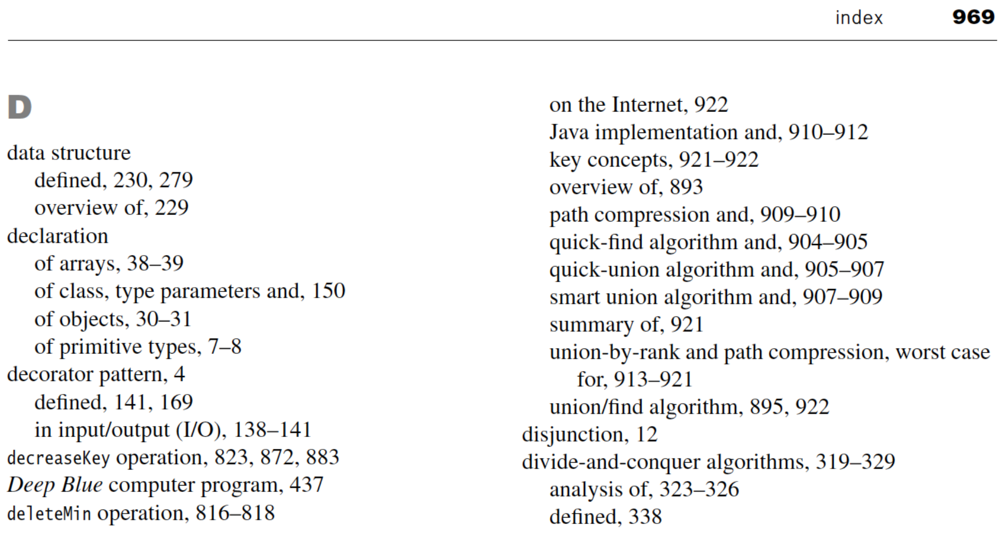
CS 150: Project III

Project Description

Implement a program that creates a book index of any given book. A typical book index appears at the end of a book and lists words used in the book and the page numbers they appear in. The following is a page from the index of our textbook.



index **969**

D

data structure
    defined, 230, 279
    overview of, 229
declaration
    of arrays, 38–39
    of class, type parameters and, 150
    of objects, 30–31
    of primitive types, 7–8
decorator pattern, 4
    defined, 141, 169
    in input/output (I/O), 138–141
`decreaseKey` operation, 823, 872, 883
Deep Blue computer program, 437
`deleteMin` operation, 816–818
    on the Internet, 922
    Java implementation and, 910–912
    key concepts, 921–922
    overview of, 893
    path compression and, 909–910
    quick-find algorithm and, 904–905
    quick-union algorithm and, 905–907
    smart union algorithm and, 907–909
    summary of, 921
    union-by-rank and path compression, worst case
        for, 913–921
    union/find algorithm, 895, 922
disjunction, 12
divide-and-conquer algorithms, 319–329
    analysis of, 323–326
    defined, 338

A real book index only lists "important" words. Our book index will list all English words that appear in a book. Assume that the book is given in a plain text file (txt) as those downloaded from Project Gutenberg. Your program will create an index that lists each word appear in it and the line numbers where the word appears (since there are no page numbers, we will use line numbers). For example, the following is a portion of an index for the book "The Complete Works of William Shakespeare":

```
bechance [149617, 164034]
bechanced [54969]
beck [1170, 10194, 11030, 30899, 54586, 124958]
becking [152565]
beckon [133425]
beckoning [144460]
beckons [29105, 29108, 29163, 48371, 105480]
becks [62854, 133943]
```

From this partial index, we know that the word "bechance" appears in lines 149617 and 164034, and the word "bechanced" appears in line 54969 and so on.

For the purpose of this program, ignore the cases of the words. In other words, “Frank” and ”frank” are considered the same words. Furthermore, you only need to consider English words consisting of upper- or lower-case letters. Therefore, you may read the input file line by line and use the following method to extract the words in each line:

```
String line = in.readLine();
String[] words = line.split("[^A-Za-z]+");
```

After each word is extracted, you will convert it to all lower-case. You then search for it in a given dictionary of English words (using binary search) and if word exists in the dictionary, you add it to the index. If the word is already in the index, you add the line number to the set of line numbers associated with the word; if the word is not in the index, create a new entry containing the word and the line number in the index. The words are listed in an alphabetical order and the line number are listed in an increasing order.

There are three options for the data structure of the index:

1. Sorted List: use a sorted `ArrayList` for the words and their associated sets of line numbers. In other words, the index is a `ArrayList<Entry>`, where each `Entry` object has a `String` for the word and a `TreeSet<Integer>` for the set of line numbers. When a new word `w` is found on line `k`, you will run a binary search for the word `w` to find if it exits in the `ArrayList`. If it already exists in the `ArrayList`, you will add the line number `k` to its associated set; if not, you will create a new Entry object containing `w` and `k` and insert it into the appropriate position in the sorted `ArrayList`.

2. Tree Map: use a `TreeMap` for the words and their associated sets of line numbers. In other words, the index is a `TreeMap<String, TreeSet<Integer>>`. When a new word `w` is found on line `k`, you will search for the word `w` to find if it exits in the `TreeMap`. If it already exists in the `TreeMap`, you will add the line number `k` to its associated set; if not, you will put a new entry containing `w` and `k` in the `TreeMap`.

3. Hash Map: use a `HashMap` for the words and their associated sets of line numbers. In other words, the index is a `HashMap<String, TreeSet<Integer>>`. When a new word `w` is found on line `k`, you will search for the word `w` to find if it exits in the `HashMap`. If it already exists in the `HashMap`, you will add the line number `k` to its associated set; if not, you will put a new entry containing `w` and `k` in the `HashMap`.

Assignment

Your assignments are:

1. Perform a theoretical analysis of the above three options. Compare and contrast their relative strengths/weaknesses and hypothesize their relative performance under difference scenario.

2. Write a program to implement the three options and create a testing facility. You implementation should be efficient. It should use a good object-oriented design. For example, it is advisable to create an interface class `Index` which is implemented by three subclasses `ListIndex`, `TreeIndex` and `HashIndex`. As usual, create JavaDocs and run unit tests on your program.

3. You will be provided with a book in plain text, a dictionary containing 194k English words, and a sample index file for the book. Test your program on the provided input data to make sure that the results of your program match the output data.

4. Compare the performance of the three approaches by running experiments on books of different sizes (downloadable from Project Gutenberg). Create a plot of three curves representing the running time of the three approaches as a function of the book size. You should run at least 5 experiments for each data point on the curves.

5. Compare your simulation results with your theoretical analysis and test your hypothesis.

Guidelines

1. You should use the appropriate data structures and efficient algorithms. Make appropriate use of object-oriented design concepts. Avoid large classes and methods and run thorough unit testing.

2. Your analysis and conclusions should be supported by mathematical reasoning and experimental data.

Report

The guideline for wiring the project report is the same as that of Project 1 and 2.

Grading

The project is to be completed individually. Unless otherwise approved by the instructor, the only person you can consult is the instructor. Your project will be graded on the following criteria (assuming the program compiles and runs):

1. soundness of the theoretical analysis

2. correctness of the program

3. documentation (methods and classes) including javadoc

4. unit testing and the ability to allow more testing

5. object oriented design

6. quality of the experiments and the data analysis

7. quality of the project report