Data Structures (CSC 151)
Fall 2024
Project 5: We the People **Due: Tuesday, 11/12/2024**

Learning Objectives:

1. Practice with binary search trees
2. To learn when a binary search tree makes a good choice for a data structure
3. More hands-on practice with generic classes and interfaces

1 Introduction

At the end of most textbooks is an index that tells you on what pages you can find certain keywords. Most modern word processors can automatically create an index for you by scanning the text, picking out significant words, and keeping track of the pages on which those words occur.

2 Your Mission

Your assignment is to write a program that will automatically create such an index for any given text file.

One issue to consider is what happens with words like "an" and "the" that occur so frequently? We'll solve that using two techniques.

1. First, if a given word is 2 characters or less, we'll ignore it altogether. That gets rid of words like "a" and "an" from the index.
2. Second, we'll keep a *dictionary* of words that we don't want to place in the index. For every word in the text, we first check the dictionary to see if the word occurs there. Only if it's not in the dictionary will we place it in the index. Words will be placed into the dictionary if they occur on too many pages. So once we've seen "the" on, say, 5 *different* pages, we'll remove it from the index and place it into the dictionary. All subsequent occurrences of "the" will then be ignored. (Note: this dictionary has the same functionality as the Python construct of the same name, but we're going to implement it differently.)

Another issue is that for each word in the index, we must keep track of the page(s) on which it occurs. Therefore, each index word will have an associated *pagelist* listing the page numbers on which that word is found. Each pagelist will be able to hold 4 page numbers. Thus, the 5th attempt to insert a page number into a given pagelist will find the pagelist full. And that's how we'll know when to delete that word from the index and place it into the dictionary. We also want to make sure that if we see two occurrences of the same word on the same page, we don't want to insert the same page number twice into that word's pagelist.

3 The Data Structures

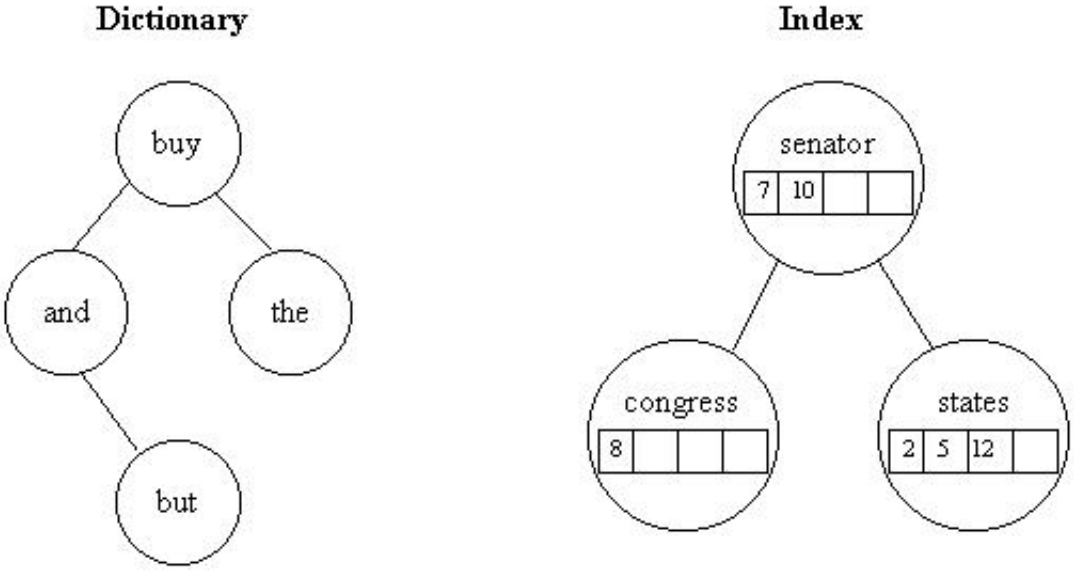
An important part of program design is deciding which data structures to use. As we study more and more ADTs, it is important for you to know the strengths and weaknesses of each one so you can pick the right structure for the job.

For this project, both the dictionary and the index itself will be searched a lot. The dictionary gets searched for every word scanned, and if not found, the index will be searched to see if the current word is already in the index. So an ADT with a good running time for searches is preferable. A binary search tree is a good choice since it has an average case running time of O(log n) for the search operation compared to O(n) average case search time for an array or linked list.

We'll use two binary search trees: one for the index and one for the dictionary. You are required to use the linked implementation of a binary search tree for this project (i.e. not an array).

Now for the pagelists. At the end of the program, we'll want to print the pagelist for each word with the page numbers in sorted order. But we're already going to encounter the pages in sorted order since we will be scanning each word starting from the beginning of the text file. Therefore we just need to save the pages in the same order that we encounter them. Pick an appropriate ADT for the pagelists. Feel free to reuse code you already have. Or build something new. How will you choose? You'll need to determine what ADT comes closest to giving you the behavior that you're looking for while weighing the pros and cons. None of them will be perfect. **You are required to create a short write-up justifying your choice while acknowledging the disadvantages. Include this write-up as a pdf file in your Gradescope upload.** As always, you may NOT use Java's built-in List, Vector, or other Collection classes.

For those who like to see it in pictures, here's what the dictionary and index might look like after a few insertions:



So each node in the dictionary just holds a word, while each node in the index holds a word and a pagelist showing where that word can be found.

4 Algorithm

We now have enough detail to write an algorithm in pseudocode. You should flesh out this algorithm further using top-down design before beginning to write any code.

4.1 Input

Any text file. Pages will be delimited by a pound symbol (#).

4.2 Output

Your program should print out three things:

1. For any word deleted from the index and inserted into the dictionary, print out the word along with its (full) pagelist. Use this format:

   ```
   Deleting 'Dick {1, 2, 3, 4}' from index.
   ```

2. The complete index, with case preserved. All words should be in ASCII alphabetical order which means the words that start with a capital letter will come before the words that are all lowercase (e.g. "Zero" will come before "apple"). You shouldn't have to do anything special to accomplish this since the String class's compareTo method already works like this. Each set of page references should be in increasing order. Use this format:

   ```
   like {3, 8}
   ran {8}
   run {3, 4, 5}
   ```

3. The complete dictionary. All words should be in ASCII alphabetical order. One word per line

4.3 Pseudo-code

See next page.

```
while (there are still more words to process) {
    get a word
    if (word is mpre than 2 characters long && word is not in the dictionary) {
        if (word is already in the index) {
            if (word's page list doesn't yet have this page number) {
                if (page list isn't full) {
                    insert page number into page list
                } else { //page list is full
                    print word and pagelist
                    delete word from index
                    insert word into dictionary
                }
            }
        } else { //word isn't in index yet
            insert word into index
        }
    }
}
print index
print dictionary
```

5 Files to Download

I'm providing you with some files to help you get started, which you can download from Nexus:

- `input.txt` and `usconst.txt`: These are two sample input files with which you can test your code. Page breaks are indicated by a # symbol. The first is a small set of sentences that you can (and should) process by hand so you can verify that your program is producing correct output. The second is the US Constitution which should give your code quite a workout. I'll be using (at least) both of these when I grade your projects. You could also create your own small input files for additional testing.
- `BinarySearchTree.java` and `BSTNode.java`: Partially completed classes for the binary search tree. In particular, the `toString` method is already completed. Please don't change that method.
- `Client.java`: You have to complete the `makeIndex` method.
- `FileReadingDemo.java`: This file will not be part of the final solution. It's purpose is to show you how to set up and use the Scanner class to read the input files word by word, while stripping away punctuation and numeric characters.

6 Getting Started

I won't be telling you every single class you need to write. You already know about some of them though:

- You'll need a `BinarySearchTree` class to handle tree operations like insertion, deletion, searching, toString, etc. Your `BinarySearchTree` should be usable for both the dictionary

and the index. Do this by making it store Comparables (the interface Java provides; see the Java API documentation). You are required to use both Java generics and the Comparable interface so that each BST instance is a homogeneous collection of general Elements, as long as Element implements Comparable. The starter files for BinarySearchTree already declares the class to store elements that implement Java's Comparable interface. You'll have to make sure that any object types you want to store using this class actually implement the Comparable interface. Note: Java's String class does implement the Comparable interface. GRADESCOPE ALERT: Gradescope will be expecting the class names and method names from the starter code. The BinarySearchTree class should have (at least) the following public methods: insert, search, delete, toString. **Start by writing JUnit tests for these methods.**

- You will need classes that implement Java's Comparable interface that you can use for the objects you need to store in a binary search tree.
- The rest of the design is in your hands. Remember to be modular. Decide what other classes you need, what public methods they should have, and how they interact, before starting to implement anything.

7 Submit

Be sure to include the honor code affirmation in the comments of one of the Client class: *I affirm that I have carried out the attached academic endeavors with full academic honesty, in accordance with the Union College Honor Code and the course syllabus.*

Submit the following files to Gradescope by **Tuesday, 11/15/2022, 10pm**.

- *all* of your .java files
- the pdf explaining your design choice for the pagelist

You don't have to submit the input files.
Before you submit, check that your code satisfies the following requirements:

1. Are all of your files properly documented?
2. Are the invariants for the BinarySearchTree class explained in a comment? Do all additional classes contain comments that explain their instance variables?
3. Are your files formatted neatly and consistently?
4. Did you clean up your code once you got it to work? It should not contain any code snippets that don't contribute to the purpose of the program, commented out code from earlier attempts, or unnecessary comments. In particular, make sure to get rid of any print statements left over from your debugging.
5. Do you use variable and method names that are informative?
6. Did you get rid of all magic numbers?
7. Does your code practice good information hiding?

8. Does your code make use of already existing public methods or private helper methods to modularize complex tasks and to minimize the number of methods that have to access instance variables?

Finally, submit your files to Gradescope (*Project 5: We the People*).

8 Gentle Reminder

Programming assignments are *individual* projects. I encourage you to talk to others about the general nature of the project and ideas about how to pursue it. However, the technical work, the writing, and the inspiration behind these must be substantially your own. If any person besides you contributes in any way to the project, you must credit their work on your project. Similarly, if you include information that you have gleaned from other published sources, you must cite them as references. Looking at, and/or copying, other people's programs or written work is inappropriate, and will be considered cheating.

9 Grading guidelines

- Correctness: Your program does what the problem specifications require. (Based on Gradescope tests.)
- Testing: The evidence submitted shows that the program was tested thoroughly. The tests run every line of the code at least once. Different input scenarios, and especially border cases, were tested.
- Documentation: Every public method and class is documented in the appropriate format (Javadoc) and provides the necessary information. The information provided would enable a user to use the class/method effectively in their code. **The ADTs' invariants are documented.**
- Programming techniques: Programming constructions have been used appropriately in such a way that make the code efficient and easy to read and maintain. If the project required the use of a specific technique or algorithm, this has been correctly implemented. For this project, I will particularly look for good information hiding and modularity, robustness, and that the stated invariant is coherently implemented.
- Style: The program is written and formatted to ensure readability. For example, naming conventions are obeyed, whitespace (blanks, line breaks, indentation) is used help structure the code, formatting is consistent and the code is well organized.