

Unless specifically told otherwise, you may not import anything into your projects. That means you may not use any built-in data structures like ArrayList or Vector. This rule applies to all projects in this course.

**Note:** You are permitted to use the `java.lang.Object` class versions of primitive data types in this and future projects (e.g. Integer). For this project you are also allowed to use the Iterator and Iterable interfaces, which can be imported from `java.util.Iterator`.

### **Learning Objectives:**

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

**Reminder:** Programming assignments are *individual* projects. I encourage you to talk to others about the general nature of the project and ways to approach it; however, the technical work (the code writing, and inspiration behind it) 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 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.

## 1 Introduction

At the end of many books is an index that tells you on which 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 goal is to write a program that will automatically create such an index for any given text file.

One issue to consider is what to do with words like “an” and “the” that occur very frequently. We'll handle that using two techniques.

1. First, if a given word is 2 characters or less, we'll ignore it completely. This 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. If the word is **not** in the dictionary, we'll place it in the index. Words will be placed into the dictionary if they occur on too

many pages. For example, once we've seen "the" on, say, 5 different pages, we'll remove it from the index and add it to 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*, which lists 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 list already full. When this happens, we'll know 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 into the word's pagelist twice.

### 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's 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 will be searched a lot. The dictionary gets searched once for every word scanned, and if the word isn't 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, which is better than the  $O(n)$  average search time for an array or linked list.

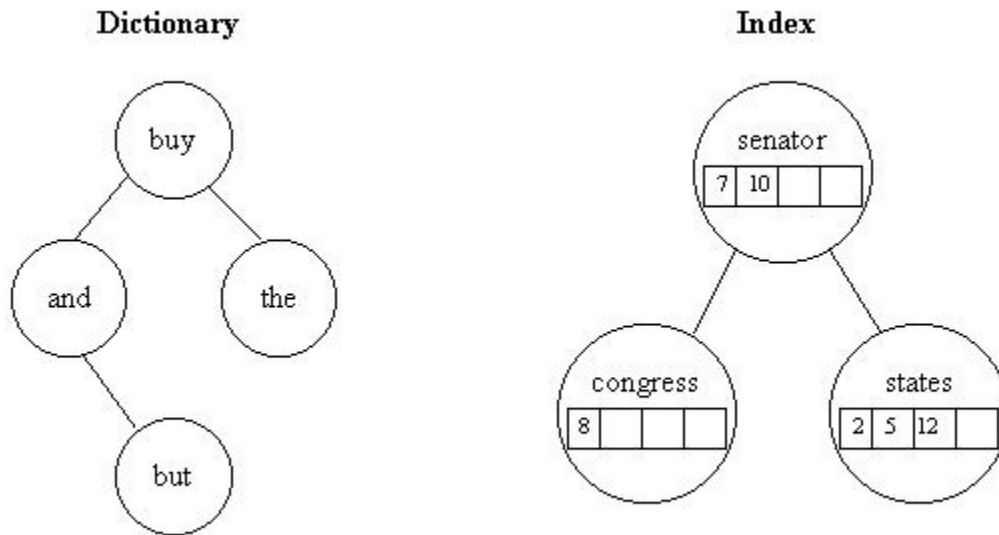
We'll use two binary search trees: one for the index and one for the dictionary. You are required to use the linked structure implementation of a binary search tree for this project (i.e. not an array).

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'll be scanning each word starting from the beginning of the text file. Thus we just need to save the pages in the same order that we encounter them. Pick an appropriate ADT for the pageslists. Feel free to reuse code that you already have or, if you prefer, build something new; the decision is yours to make. How will you choose? You'll need to figure out which 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 submission.**

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:



Each node in the dictionary just holds a single word, while each node in the index holds a word as well as a pagelist showing where that word can be found.

## 4 The Algorithm

Now we've got enough details 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 the following format:

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

2. The complete index, with case preserved. All words should be in ASCII alphabetical order, meaning that words which start with a capital letter will come before words that are all lowercase (e.g. "Zero" comes 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 the following 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 Pseudocode

```
while (there are still words to process){
    get a word
    if (word is in the index) {
        if (word's pagelist doesn't have this page number) {
            if (pagelist isn't full {
                insert page number into next position of
                pagelist
            } else { // pagelist is full
                print word and pagelist
                delete word from index
                insert word into dictionary
            }
        }
    } else { // word isn't in index
        insert word into index (don't forget to add the page
        number)
    }
}
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 a good 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. You should not change this method.
- `Client.java`: You have to complete the `makeIndex` method.
- `FileReadingDemo.java`: This file will not be part of the final solution. Its purpose is to show you how to set up and use the `Scanner` class to read input files word by word, while stripping away punctuation and numeric characters.

## 6 Getting started

I won't tell you every single class you need to write, though you already know about some of them:

- 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` (an interface that Java provides; see the Java API documentation for more information). 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.

**Note 2:** 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'll 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 up to you. Remember to be modular. Decide what other classes you need, what public methods they should have, and how they should interact before starting to implement anything.

## 7 Submission

**Be sure to include the honor code affirmation in the comments of one of the classes:**

*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 a file called `YourNameBSTTest.java` (where `YourName` is your name, like `TJSchluterBSTTest.java`) containing your JUnit tests for the `BinarySearchTree` class to *Project 5: BST Tests* on Gradescope by **Friday, 3/10/2023, 11:59pm**.

Before you submit, check to make sure that your code satisfies the following requirements:

1. All files are properly commented

2. All files are formatted neatly and consistently
3. Your code has been cleaned up. Your code does not contain snippets that don't contribute to the purpose of the program (e.g. commented out code from earlier attempts or comments that aren't helpful).
4. Your variable and method names are informative.
5. There are no magic numbers (or other magic values).
6. Your code practices good information hiding
7. Your code makes use of already existing public methods or private helper methods to make complex tasks more modular and to minimize the number of methods that access instance variables directly.

Submit:

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

to *Project 5: Creating an Index* on Gradescope by **Tuesday, 3/14/2023, 11:59pm**.

You do **not** need to submit the input files.

## 6 Grading guidelines

- **Correctness:** programs do what the problems require. (Based on Gradescope tests)
- **Testing:** The evidence submitted shows that the program was tested thoroughly. The tests run every line of 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.  
**ADT invariants should be documented.**
- **Programming techniques:** Programming constructions have been used appropriately in a way that makes the code efficient and easy to read and maintain. If the project required use of a specific technique or algorithm, this has been correctly implemented. For this project, I'll be paying special attention to good information hiding, modularity, robustness , and that the stated invariants are 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 to help structure the code, formatting is consistent, and the code is well organized.