Advanced programming – Assignment 3
SortUniq Commando

What to create

Create a program that implements the UNIX-command

```
      sort [options] <files> | uniq
```

for only the identifiers in the given files.

This program should find all identifiers in the supplied files that
occur an uneven number of times, sort these, and then print them on standard
output (the screen).
By default the program should sort identifiers monotonically non-decreasing,
differentiating between upper and lower case letters. In other words, the program
should be case sensitive by default.

How to make it

The program will learn through its command-line arguments (found in the "String[] argv" parameter of the method "public static void main()") which file(s) need to be read. When starting the program, at least one file should be given to the program via a command-line argument.

Passing **options** to the program should modify its behavior:

- When the option "-i" (case insensitive) is passed, the program should not differentiate between upper and lower case letters. Output should be given in lowercase.
- When the option "-d" (descending) is passed, the program should sort the identifiers contained in the files in monotonically non-increasing order.

The syntax of the command-line arguments is as follows:

```
commandLineArguments = [ commandLineOptions ] files ;
```
It is possible that no command-line options are passed. Files are always passed.

```
commandLineOptions = option { option } ;
```
The command line options consist of one or more options.

```
option = '-i' | '-d' ;
```
An option is either '-i' or '-d'

```
files = <file> { <file name> } ;
```
At least one file should be passed. Files are passed by filename.

Reading 1 file

The text in a file consists of words and delimiters, used to divide the words. A word is an identifier or a non-identifier.
An identifier is a row of characters consisting of letters and digits, starting with a letter. A row

of characters consisting of letters and digits, but starting with a digit, is a non-identifier and should be skipped as a whole.
All non-alphanumeric characters (including <eoln>) serve as delimiters. The text "N.B. watch it: 8 identifiers in this sentence.", therefore contains nine words of which eight are identifiers. (The word "8" is a non-identifier).
Reading and processing a text (i.e. finding identifiers among delimiters and non-identifiers in that text) is harder than it seems. It is therefore convenient to construct syntax-rules for the notions of "text", "identifier", "non-identifier" and "delimiters" before you start on the design. These syntax-rules for parsing the content of a file will be discussed at the design meeting, before the design itself is reviewed. The design is supposed to match the constructed syntax-rules.
When in possession of the correct syntax-rules, the reading and processing of a text is easily implemented by programming a method for each constructed syntax-rule.

Printing the output

The identifiers to be printed, should be printed on standard output (the screen). Each identifier should be printed as a separate line. Nothing is printed in front of each identifier. Each identifier is directly follow by an end-of-line. Do not use any other output than lines with identifiers or the automatic test will fail.
When the option "-i" is passed on invocation of the program, all identifiers are printed in lower case.

Implementation

Use a binary searchtree as the datastructure for storing identifiers. Create an interface for this to show at the design meeting.
Use the following abstract methods in this interface:

```
    /**
     @postcondition
      The data stored in the binary search tree was iterated in
      monotonically non-decreasing order and was added in this
      order to an object of the type Iterator<E>.
      This  object  of  the  type  Iterator<E>  was  subsequently
      returned.
  **/
    Iterator<E> ascendingIterator ()
```

and

```
    /**
     @postcondition
      The data stored in the binary search tree was iterated in
      monotonically non-increasing order and was added in this
      order to an object of the type Iterator<E>.
      This object of the type Iterator<E> was subsequently
      returned.

         **/
    Iterator<E> descendingIterator ()
```

An object of the type Iterator<E> can be created using an object of the type ArrayList as explained in the lecture. De specification of the class ArrayList and the interface Iterator<E> can be found with the URL http://docs.oracle.com/javase/7/docs/api/ .

Example

The contents of `file1` are the words on the following two lines:

```
one two 2, two three 3, Four Four five Six Six.
seven eight 8; eight
```

The contents of `file2` are the words on the following two lines:

```
one two 2, three 3, Four five Six.
seven eight 8;
```

When the contents of the two files with names `file1` and `file2` are passed without options to the finished program, then the output should be:

```
Four
Six
eight
two
```

The identifiers "one", "three", "five" en "seven" are not present in the output because they occur an even number of times in the input and the program will only print identifiers occurring an odd number of times.

Submission

When the program is completely finished and thoroughly tested (do not only use the example above), you can submit all Java files of your program, combined into one zip file, via Practool.