Backwards and Huffman

Assignment 4 for Sorting & Searching

This assignment is ONLY for students who took the course Sorting & Searching. That is students who started Software Engineering during the academic year 2017-2018 or before!

Backward search

You have to implement a text-based searching algorithm that finds the right most occurrence of the text (the needle) searched for. This in contrast to most algorithms that find the left most occurrence. This brings you to a beautiful insight. Why not use a reverse version of the Boyer-Moore algorithm. This way you will always find the right most occurrence as fast as possible. Besides determining where the needle can be found, your code must be able to return the number of character comparisons made during the the last search.

You may assume that all text is in lowercase and does not contain spaces, punctuations or special characters.
No clever tricks such as reversing the haystack and/or the needle, converting to StringBuffer, Streams or whatever and using the standard Java search implementations are allowed!

You are allowed to look for examples on the internet, if you use code from any external source you must clearly state the original code's origin!

When looking for '`needle`' in '`needleinthehaystack`' after comparing the n and the y you should shift left the pattern 6 characters.

```
needleinthehaystack
      needle
```
there is mismatch on the first character and they `y` is not in the pattern -> shift 6.

```
needleinthehaystack
    needle
```
there is a mismatch on the second charcter and they `t` is not in the pattern -> shift 5

```
needleinthehaystack
 needle
```
there is a mismatch on the first charcter and they `e` is in the pattern -> shift 1

```
needleinthehaystack
needle
```
there is a mismatch on the first charcter and they `e` is in the pattern -> shift 1, etc

Deliverables

- The code that implements the reverse Boyer-Moore search
- Describe clearly what you have changed compared to the original Boyer-Moore implementation and most importantly why. (you are allowed to use the code from Sedgewick)
- Provide tests that clearly show that your reverse Boyer-Moore implementation is more efficient (needs less character comparisons) than the original Boyer-Moore code, which seaeches from left to right. (You might need to find some large bodies of text that can be used for these tests.)

Tips

Although the implementation can be done without a lot of modifications here are some tips to help you.

- Since you can assume only lowercase characters are used in both the pattern and the text you can place the shift value for ‘a’ at index 0.
- Since you are implementing the algorithm in reverse order you might need to change some minuses into pluses and vice versa and change the initial value and condition in some loops.
- The initial values for the shifts are different from the standard implementation. Start by investigating with what should happen when the first character (the left most) results in a mismatch with the mismatching character does not appear in the pattern. How many characters should the patterns shift to the left? Once you figured that out, what should the shift value be when the mismatch happens somewhere else.

HuffmanCompression

With this assignment you need to determine the compression-ratio for a Java source file using the Huffman compression method and compress the text. On the start project which can be found on the DLO contains all you need to get you started. The class `HuffmanCompression` has some methods that you must implement. You are allowed to add extra methods, please be careful with the access-modifiers. Don’t make every method `public` just because you are used to. You are NOT allowed to change the signature of any of the provided classes! There is also a `Node` class which you must use when constructing the Huffman tree. This class also needs some code to work properly.

For calculating the compression-ratio you need to know many occurrences there are for every unique character in the text and how long (measured in bits) the code is for that character. Please keep in mind that everything IS a character when compressing text. Not only the characters form the alphabet, but also slashes, exclamation points, carriage-return, line-feeds, and horizontal tabs are characters. You don’t have to consider the translation table itself!
You can assume that 1 character needs 8 bits in the original text!

Once you have determined the number of occurrences for every character you can build the Huffman-tree. For this you must use the provided `Node` class.
With the Huffman-tree at hand you should be able to determine the compressed code that will be used when compressing the file.

Knowing the number of occurrences of each character and the code that will used when compressing the text you can determine the compression-ratio and compress the text.

One of your tasks is adding extra tests that will ensure that your code works properly.

When nodes have the same weight and both of them are leafs, the you must sort then alphabetically. If one of them is a node with only a weight and the other is leaf then the leaf comes first. If neither of them is a leaf the order is undetermined.

Note

Since Java is not particularly well suited to handle single bits we use Strings instead. So if the code for a character should be 0110 (e.g. 4 bits) your code can use the String “0110”.

Report

The report contains a brief description of the written code in plain English[1] or Dutch[2] (avoid technical mambo-yambo). Also clearly state the encoding of “YES, we made it!” using the Huffman codes that you derived from the provided file.
And of course you must upload a ZIP-file containing all the Java source files including your unit tests.

Tips

- There are some methods in the class `HuffmanCompression` that have the default-scope (package private), you can use these methods to test parts of your code during development.
- When implementing the methods in the `Node` class there is a tricky difference between a leaf and a node. A node only has a weight and a left and right node, while a leaf only has a weight and a character value.
- Try to use recursion since this tends to make your code very elegant and keeps the methods short.

[1] If your lecturer doesn’t speak Dutch fluently.
[2] If your lecturer does speak Dutch fluently.