Introduction

Suppose that we have an alphabet of *n* symbols and a long message consisting of symbols from this alphabet. How can we encode the message so that it's shorter and we can save space?

For example, suppose that there are 4 characters in our alphabet: A, B, C, and D, and that the message is BAACABAD.

Method 1: Use ASCII code

Each character needs 8 bits, so a total of 64 bits (8 * 8) is required.

Method 2: Use binary

Since there are only 4 symbols, 2 bits are enough to distinguish them:

A	00
B	01
C	10
D	11

The message can be encoded as 0100001000010011. A total of 16 bits (8 * 2) is required, and we save 75% space.

But, can we do better?

Huffman Coding

Huffman Coding is an encoding algorithm that uses **variable length codes** so that the symbols appearing **more frequently will be given shorter codes**. It was developed by David A. Huffman and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

Suppose we use the following code:

A	**0**
B	**10**
C	**110**
D	**111**

Then **BAACABAD** can be encoded as **10001100100111**, and we only need 14 bits.

The code for one symbol should ***not*** be a prefix of the code for another symbol, because it would create ambiguity. For example, if the code for A in the example above was 1 (instead of 0), then 111 could be interpreted as D or AAA.

Applications

- The DEFLATE lossless compressed data format, used by different file compression utilities/formats (such as zip and gzip), uses a combination of the LZ77 algorithm and Huffman coding.
- The bzip2 compression utility uses a combination of the Burrows-Wheeler compression algorithm and Huffman coding. It is considered to be more effective at compression than zip or gzip, although it is slower.
- Google created its own lossless compression algorithm, called Brotli, which also uses a combination of LZ77 and Huffman coding.
- The JPEG file format uses Huffman coding as part of its compression algorithm. The PNG file format also indirectly uses it, since it uses the DEFLATE compression algorithm.
- Some communications protocols use Huffman coding, such as HPACK, the header compression technique of http/2.
- Some fax machines use “Modified Huffman coding” (a combination of run-length encoding and Huffman coding) to encode black on white images.

How to construct Huffman Code

From the previous example assume the message is: **BAACABAD**
The steps are:

1. **Calculate** the frequency distribution of the symbols. In our example above, we have:

Symbol	Frequency
A	4
B	2
C	1
D	1

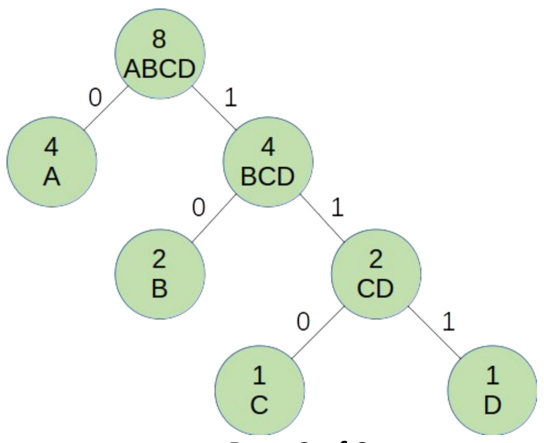
2. **Find 2 symbols** that appear least frequently. In our example, these symbols are **C and D**.
3. The **last bit** of their code will differentiate between them, say 0 for C, and 1 for D.
4. **Combine** these 2 symbols into one single symbol, whose code represents the knowledge that a symbol is either C or D. Let's call this new symbol **CD** and whose frequency is 2, which is the **sum** of the frequencies of C and D.
5. Now there are three symbols left:

Symbol	Frequency
A	4
B	2
CD	2

6. **Repeat steps 2-5 until there is only 1 symbol left.**

Huffman Tree

This process can be best described by using a binary tree. Note that the code of *any* symbol can be determined by **starting at the leaf that represents that symbol** and **traversing up to the root**. The code is constructed from right to left.



For example, with **C** we would obtain:

0 → 10 → 110.

For consistency, when constructing the Huffman tree assume that a node with a **lower frequency is the left child** and is assigned a **0 bit**, and a **node with higher frequency is the right child** and is assigned a **1 bit**. If they both have the same frequency, compare the symbols as a tiebreaker, so that the "smaller" symbol is the left child. In practice, this tiebreaker is unnecessary; it's only added here for consistency among projects.

Notice in the previous Figure that the nodes of the Tree contain the symbols and their respective frequency. The bits associated with each symbol are *implied* by their placement in the Tree. Hence how C's code was calculated.

General Specifications

You will finish the implementation of the `HuffmanCode.java` such that it does the following:

1. Read an input file with a single line of text.
2. Determine the frequency distribution of every symbol in the line of text.
3. Construct the Huffman tree.
4. Create a mapping between every symbol and its corresponding Huffman code.
5. Encode the line of text using the corresponding code for every symbol.
6. Display the results on the screen.

One of the reasons why design is such an important aspect of Software Engineering is that it helps you organize your logic. By following the 6 steps above, you should be able to neatly organize your code into the following 4 methods:

1. **`load_data`**: Receives a file name (including its path) and returns a single string with the contents. This method is already done for consistency amongst projects.
2. **`compute_fd`**: Receives a string and returns a Map with the symbol frequency distribution.
3. **`huffman_tree`**: Receives a Map with the frequency distribution and returns the root node of the corresponding Huffman tree.
4. **`huffman_code`**: Receives the root of a Huffman tree and returns a mapping of every symbol to its corresponding Huffman code.
5. **`encode`**: Receives the Huffman code map and the input string and returns the encoded string.
6. **`process_results`**: Receives the frequency distribution map, the Huffman code map, the input string, and the output string, and prints the results to the screen (per specifications). This method is also done for consistency amongst projects.

Note that this design should allow you to conveniently test each method after it's implemented, which helps detect errors in the most recent code. You may include additional helper/auxiliary methods, but these methods in particular are **required,** and they must be *public* so that they show up in the Javadoc documentation (discussed later).

There's a different and simpler strategy for building the Huffman code than starting at the root and traversing to the leaves. (**Hint**: you would have to change the return type of the **`huffman_tree`** method). You may use that other strategy if you want, but I'll leave it to you to think about what that strategy is.

Input/Output Specifications

1. The input file will use UTF-8 encoding, which has a character set of 1,112,064 characters, so you may need to specify this in the file-handling methods you use. However, for the purpose of this project you may safely assume that the input file will have no more than 1,000 distinct characters. You will be provided a sample input file.
2. The input file will be named **`stringData.txt`** and it will reside in a directory named **`inputData`**, which resides at the same level as the **`src`** directory for your project.
3. The results to be displayed are as follows:
   a) Frequency distribution table, **including the Huffman code,** in non-ascending order by frequency.
   b) The original message.
   c) The encoded message.
   d) The amount of bytes needed to store the encoded text and the savings percentage compared to the amount of bytes needed for the original text. Here's sample output (you can use tabs to align your columns):

```
Symbol      Frequency   Code
------      ---------   ----
A           4           0
B           2           10
C           1           110
D           1           111

Original string:
BAACABAD
Encoded string:
10001100100111

The original string requires 8 bytes.
The encoded string requires 2 bytes.
Difference in space required is 75%.
```

Note: Technically, if we compress a file/message using Huffman coding, we will also need to include a Huffman table in the file, so that the user or software on the other end may know how to decode the file/message. However, we will ***disregard*** *this aspect in this project.*

Note: This part of the specifications is already done for you to maintain consistency amongst all projects, but it is presented here so you have an idea of what happens and what each method of input/output is supposed to do.

Implementation Ideas

First, let's mention some Java classes that you should be using in this project:

- **Class `BTNode<K, V>`:** The nodes of our binary tree will use the **frequency as the key** and the **symbol(s) as the value**.
- **Class `SortedLinkedList<E>`:** You will implement and use a **`SortedList`** to easily retrieve the next symbols with the lowest frequency. What data type will the SortedList store? BTNode! This means that your **`BTNode`** must be **`Comparable`**. The way you will compare two nodes is by comparing their keys, which means that the keys must also be **`Comparable`**. This time we'll just incorporate the comparisons directly into the BTNode class because our **`SortedList`** class requires the data type to be **`Comparable`**. Note that for most implementations of this algorithm a PriorityQueue is used, but why is a SortedList a good substitute for it? **Watch this video to see how this modified version of the List ADT works so you can efficiently implement your `SortedLinkedList` class. Also be sure to review the `Comparable<E>` interface documentation from the Java API as you will need it to implement this ADT as well. Here is a resource on Comparable vs. Comparator in Java.**
- **Class `HashTableSC<K, V>`:** By now you should be well-aware of the ideal data structure to use for counting frequency distributions. This time, however, we'll use our own implementation (including the rehash method). Note that we can use the same class to later map every symbol to its Huffman Code.

Note that technically we don't need a Binary Tree class since we're creating the tree **from the leaves up to the root**. Here's the algorithm for building the Huffman tree:

```
Huffman-Tree(FD): # Receives FD data and returns the root of a Huffman Tree
1  size ← | FD |
2  Create SortedList SL of nodes
3  SL ← FD # Store the frequencies and corresponding symbols in freq order
4  for i ← 1 to size - 1 do
5     Allocate new node N
6     N.left ← x ← MIN-FREQ-REMOVE(SL)
```

```
    N.right ← y ← MIN-FREQ-REMOVE(SL)
    N.freq ← x.freq + y.freq
    N.symbol ← x.symbol + y.symbol
    SL.add(N)
# SL should only have one node, the “root” of the tree
return MIN-FREQ-REMOVE(SL)
```

After the tree is finished, we now need to *map* every symbol to its Huffman Code, according to the tree. Then just loop over every symbol of the input text and print its corresponding code.

Documentation & Comments

You must properly document your code using the Javadoc format. Use the tools in Eclipse to generate the Javadoc documentation; it must be included in a directory named `doc` that resides within your project directory (at the same level as the `inputData` directory). **Your entire `HuffmanCoding` & `SortedLinkedList` classes must be thoroughly documented using Javadoc comments, so that anyone reading the HTML documentation knows exactly what the class represents and knows exactly what each *public* method does.** You may use the official `ArrayList` documentation as a reference guide. If you have never generated Javadoc documentation before, there are great resources online that show you how to do so in Eclipse, but it’s very straightforward.

Additionally, since different people will have different implementations, it’s important that within your methods your code is well documented and commented so that we may understand your intention. This includes block comments that may take 2-4 lines explaining what the next several lines intend to do; hopefully the code you have seen so far this semester has provided good examples of that. This is an important part of your formation as a future professional in the computer science/engineering industry. You will lose points if you don’t provide Javadoc-style comments and additional comments explaining your logic, or if we deem the amount and/or frequency of comments to be inadequate. If you’re unsure whether you have enough comments, I encourage you to meet with your TA and seek his/her input.

Submission

The *official* final date to submit your program will be Friday**, April 21, 2023,** at 11:59pm. The project will be submitted through GitHub Classroom as specified before for Project 1. The deadline specified in this document is a **hard deadline**, meaning that **there will be no deadline extensions granted under any circumstance**.

Academic Integrity

Do NOT share your code! You may discuss design/implementation strategies, but *if we find projects that are too similar for it to be a coincidence, all parties involved will receive a grade of 0*. Don't cheat yourself out of a learning experience; seek our help if you need it.

Final comments

The specifications of a project are the first, and arguably the most important, part of a software development project. Therefore, it's crucial that you read these specifications thoroughly so that you understand what is being asked of you. These are skills that you will need to succeed in your professional career, so it's imperative that you start applying and improving them now. If your program runs successfully, but does not adhere to the specifications, it is of no use. *Before you submit your project, review these specifications one last time and make sure you meet all of the requirements that have been imposed.*

If your code does not compile properly, your grade will be 0, NO EXCEPTIONS!