**Approach**

**Algorithms**

*Encode (encode.js)*

I implemented static Huffman algorithm and then used it in adapting Huffman by reconstructing tree each step. I am updating tree after encoding because in decoding I would only have access to previous tree.

Each step, it starts from reading one character from string, encoding the character with the current state of the Huffman tree and updating the alphabet by either incrementing this character’s value or adding a new character and reconstructing the tree with alphabet.

*Decoding (decode.js)*

My implementation is a bit the other way around comparing to encoding algorithm.

I look through the input by each symbol. Only when encoding reaches the leaf of the tree I do something – if it is a dugger, I read next 8 characters, decode them and add character to alphabet and output. In another case, I just read its tag, which represents the character, and increment that character’s value in the alphabet. In the end, I reconstruct the tree by running static Huffman.

*Static Huffman (updateTree.js)*

In my algorithm, I am using static Huffman algorithm to reconstruct my tree.

The way how it works that I have an alphabet. I am sorting my alphabet by frequency values. After that, it is getting the smallest two values in the alphabet, sum them together, add them back to alphabet and repeat the same operation until I get only one element in alphabet.

*Drawing a Huffman Tree*

I only used one external algorithm in my program to draw a Huffman tree in console. This is commented in my code.

<https://www.npmjs.com/package/treeify>

**Implementation**

1. Alphabet

My alphabet is an array called *nodes* where each element in array is a *HuffmanNode.*  I am using array *nodes* to save and keep track each read character and building my Huffman tree. I chose array because it was convenient for implementing sorting and building a *Huffman tree.*

If character has not been seen before in *Huffman tree*, it saves it in *nodes* array**,** calculates the dugger from previous tree and adds the ASCII code to overall output. I am using a *dugger*, which is the way to identify that a new character has to be read during decoding.

1. Huffman tree

My Huffman tree represented as a binary tree. Each node is an object class *HuffmanNode* that has value, tag, left and right. Value defines the frequency of each character appeared in the output string. Tag is the symbol of the character from input string. I am storing the concatenation of the characters in the tag because it is the way how I know if the node is in the left or right child. Tag helps me to navigate in the tree and know the connection between children and the parent node. The left and right are representing the split of my concatenated string same as tag helping with navigation in the tree.

I chose this representation of the Huffman tree because it is easy way to navigate in tree.

1. Building Huffman tree

As I mentioned above, I am using static Huffman algorithm to reconstruct my tree. For implementing these, *nodes* array is being sorted. The smallest two elements are summed by theirs value, tag and then removed from array. After that, I create the new node, which becomes the parent of those two smallest elements. The node tag is the concatenations of its children tags and value is the sum of values of children. Node gets added back to array and this operation keeps repeated until the last element in array.

1. Dugger

For *dugger* implementation, I am using the ASCII of 255 because this value is less frequently used. However, in some languages *dugger* could be represented, as empty value of string but unfortunately, JS does not allow me to do this.

**Demonstration**

**Set up**

1. Install node js

<https://nodejs.org/en/download/>

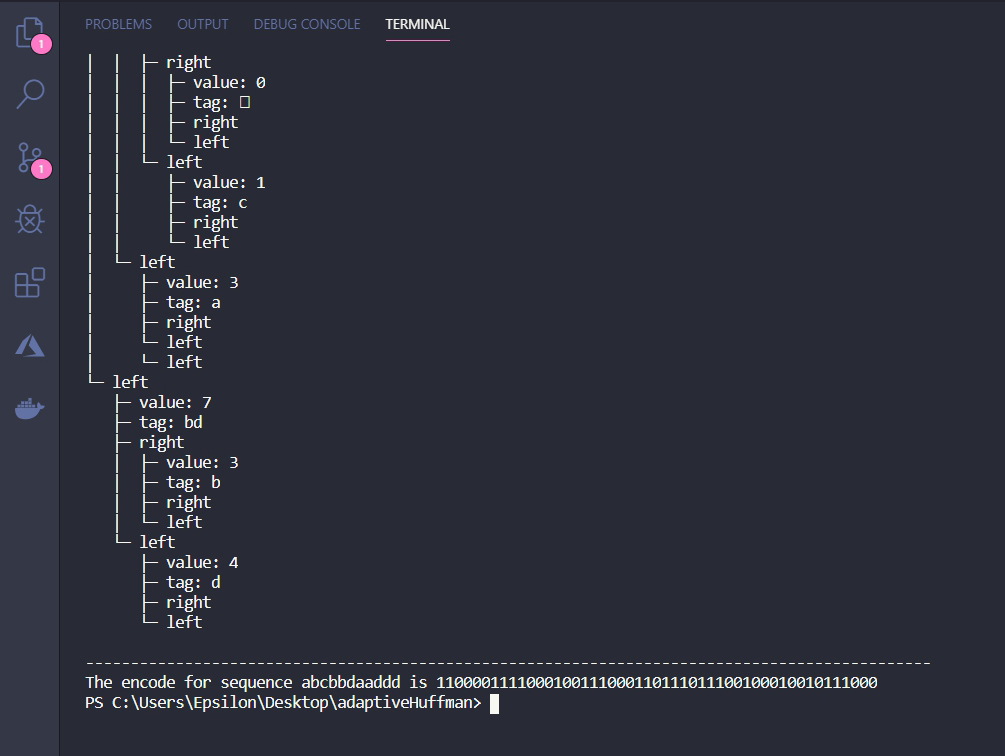
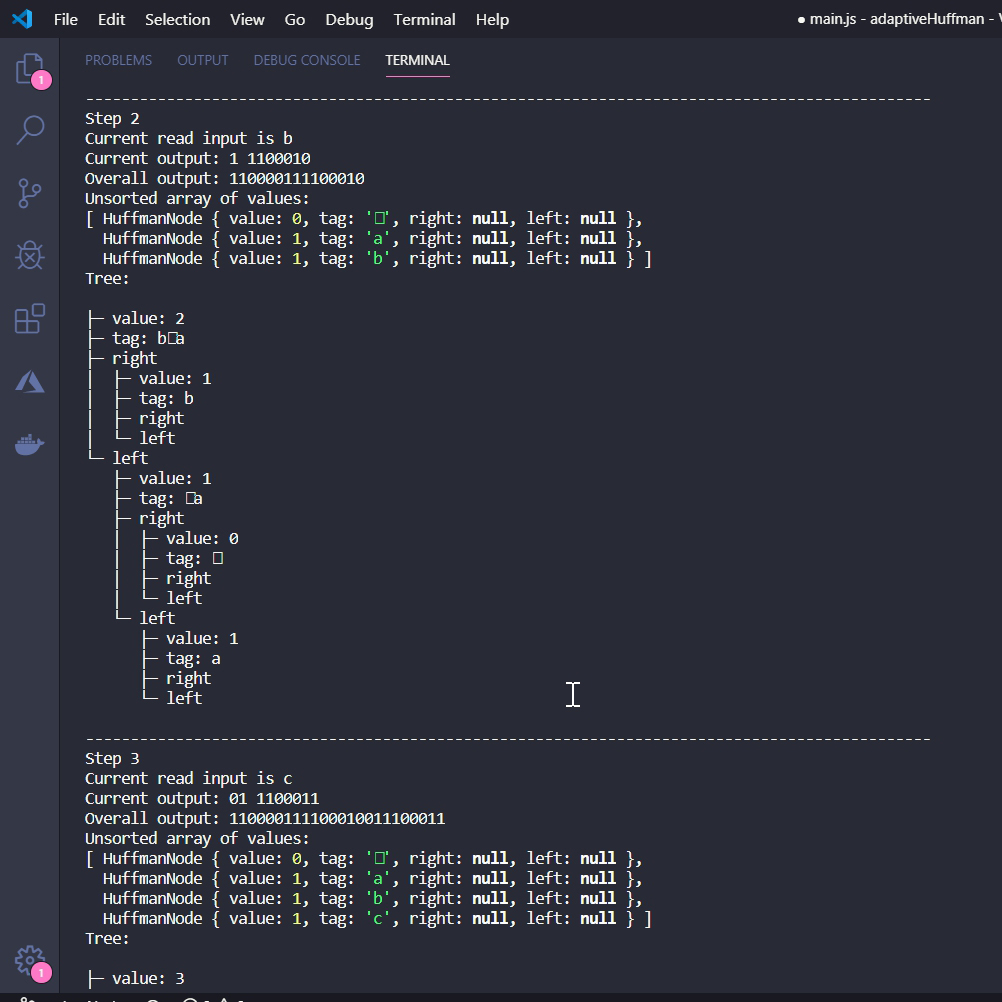
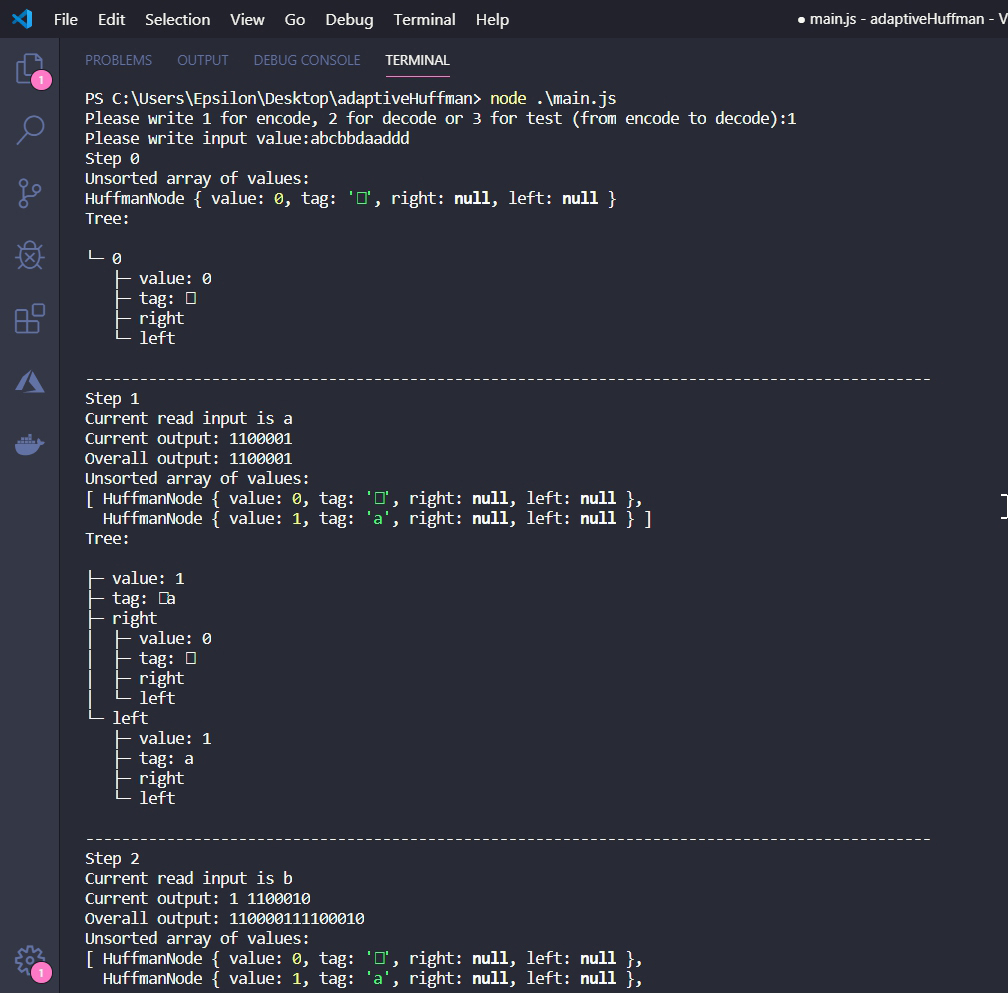
download the right version

1. Run **node -v** to check successful installation

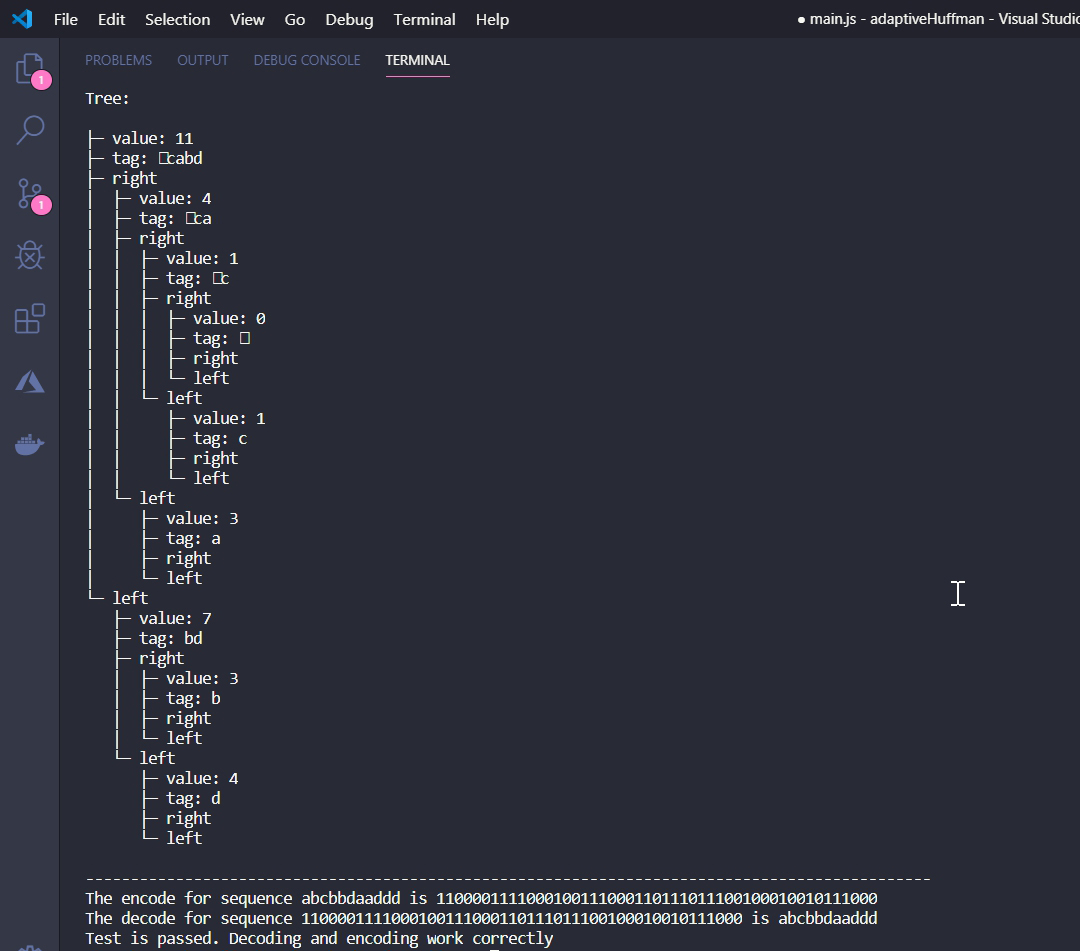
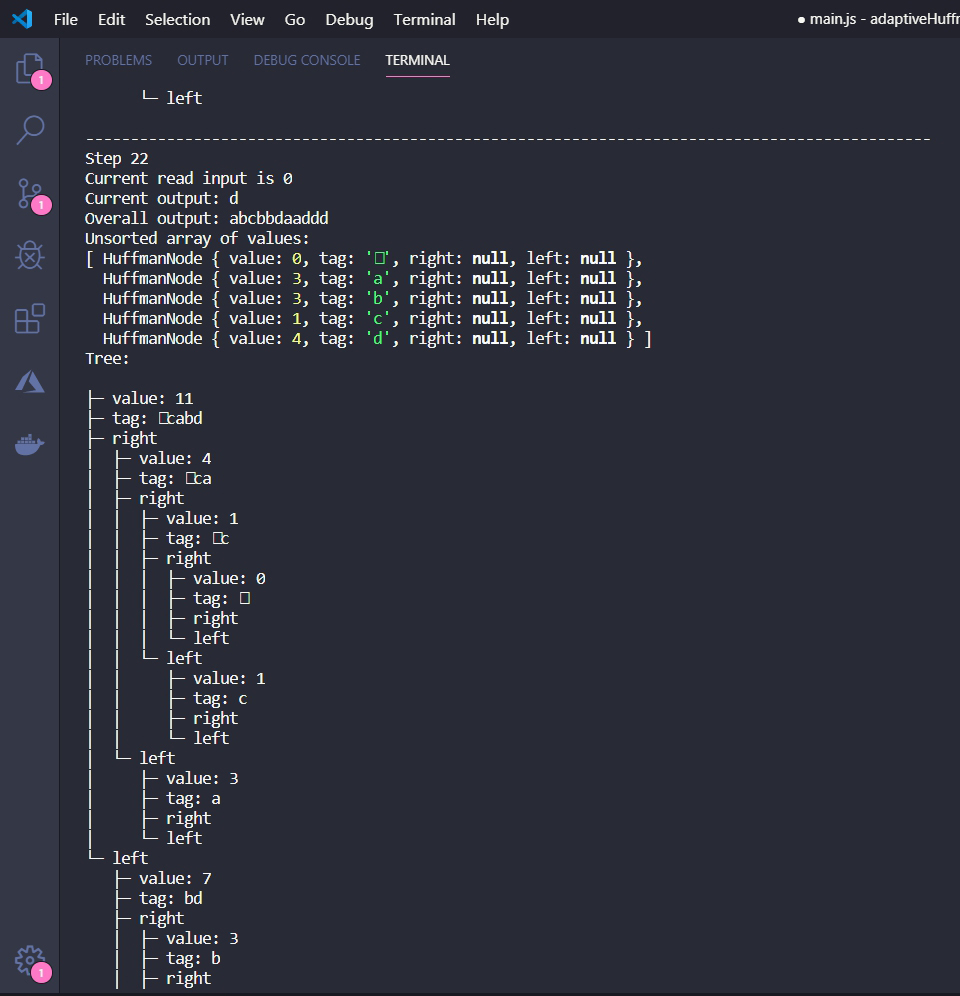
**Run**

1. Go to root of the folder **cd adaptiveHuffman**
2. Run command **node .\main.js**
3. For encoding write 1 and then your output
4. Run test to see all step for encoding and decoding. Write 3 and then input string. You will be able to see at the bottom the result

For encoding **abcbbdaaddd**



Test for **abcbbdaaddd** is doing both encoding and decoding



**Discussion**

My aim was to write the program with my own implementation of algorithms without using others people code because I wanted to learn how Adoptive Huffman algorithm works by myself and become better in writing software. This was really challenging because of my small background in algorithms. Even my implementation is not that efficient and basic, I am pleased that I achieved my personal goals and learnt new approaches.

**Improvements and suggestions**

1. In my approach, I reconstructing the tree each iteration, which makes it less efficient. There are two examples of algorithms that implements the updating the tree - *FGK* and *Vitter* algorithms.
2. Representation of Huffman tree

Array representation of the tree would be more efficient than in my situation. In array, you would not need to have left and right because you can easily access to these value by *left(i)=2i+1*. As a result, array representation of the tree would have better access to values.

1. Sorting the values for constructing tree

For more efficient way to sort my array, it would be better to do it as heap because in heap I would have better access for max and min values.

1. Dugger

My limitation is that if dugger appears in my string it will break my code. However, I can improve this situation by creating the algorithm to correctly changing the dugger. I would still need to choose any character for dugger which has value 0. This value would help to identify if it is a dugger. Let’s use example when my dugger is ‘x’. When the same character appears in the output, I increment value for ‘x’ and create a new dugger character by randomly choosing a new character and checking it against my tree.