

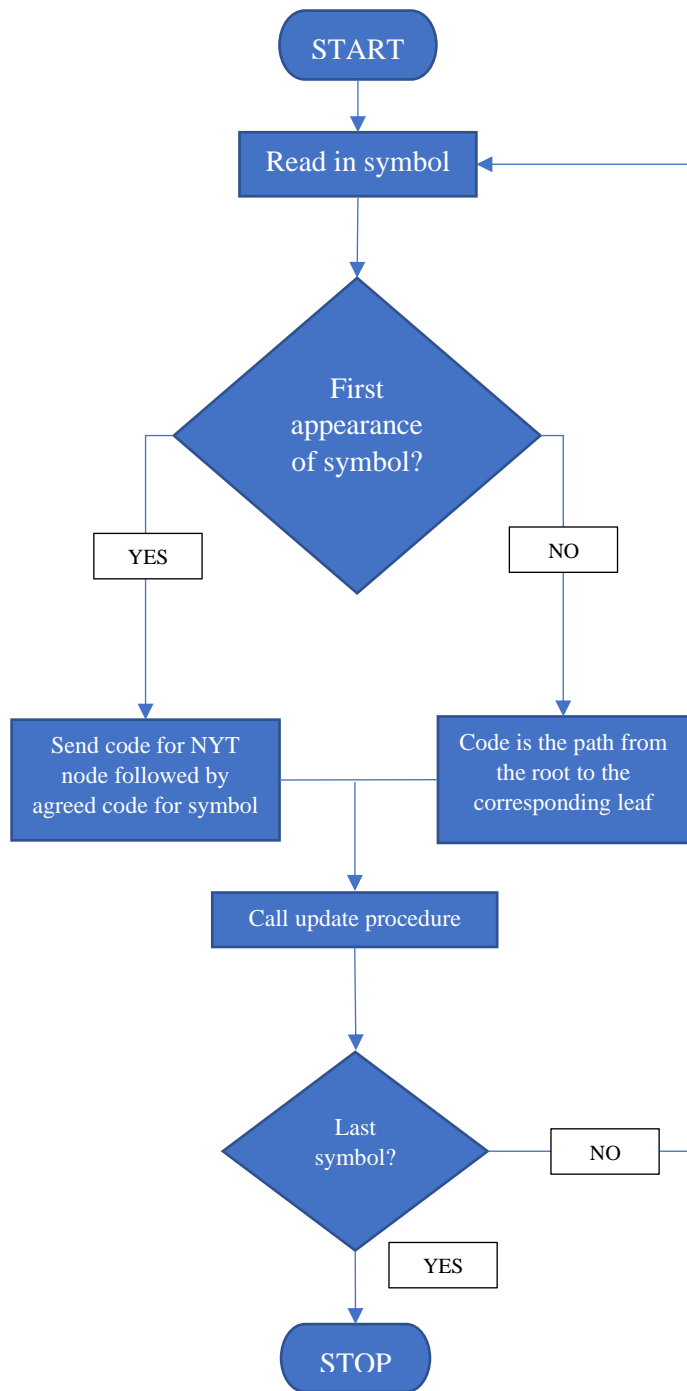
Subject: Implementation of String Compression Algorithms

Before reviewing the classes and methods implemented for encoding and decoding text with the methods requested by questions 1 and 2, a brief presentation of the Huffman encoding method is considered useful. In particular, Huffman encoding is achieved by constructing the Huffman Tree which has properties favourable for encoding.

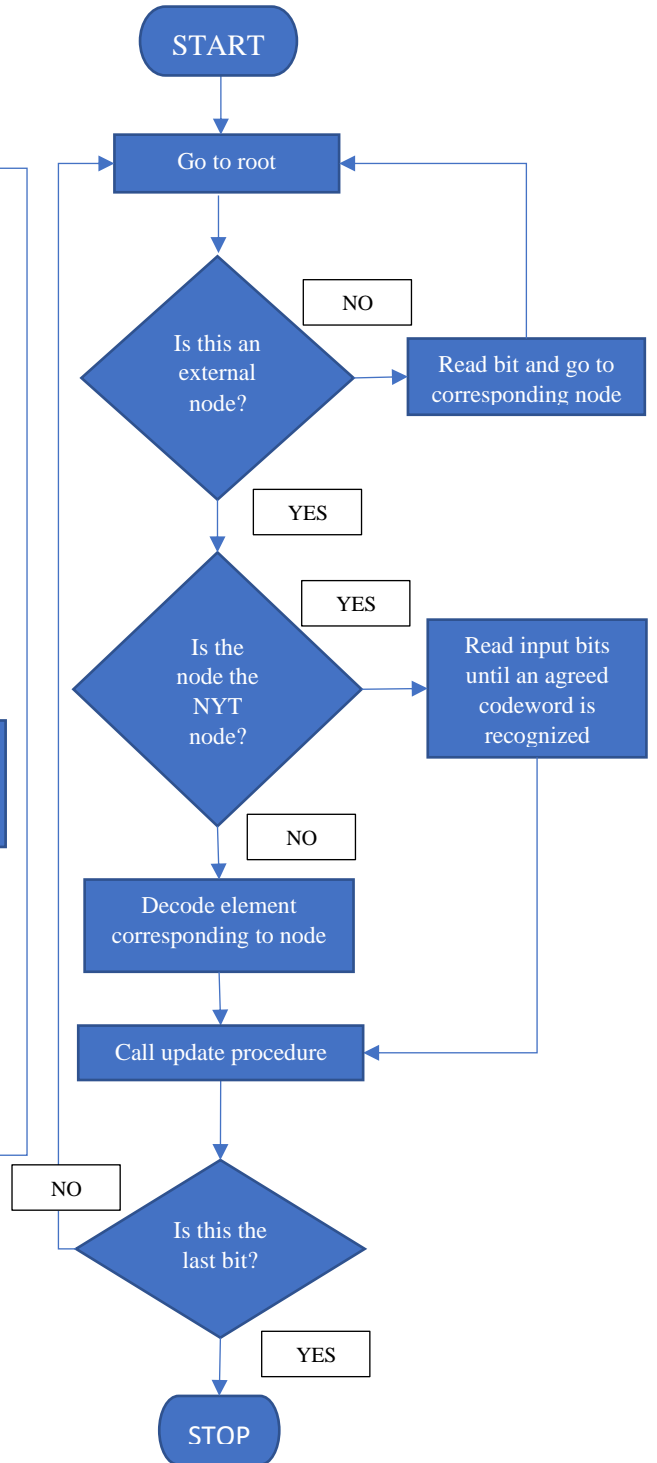
As far as the static Huffman algorithm is concerned, the coding process starts with an initial scan of the text (text to be encoded) to create an "alphabet" of frequencies for each character, i.e., a list of characters is constructed which is accompanied by their respective frequencies of occurrence in the given text (input text). Then, we insert the elements of the alphabet with the frequencies into a min heap and perform two deletions of the least element from the heap (delete min) and obtain as output the elements with the two smallest frequency fields, which could be achieved in many ways but for the implementation of the static Huffman algorithm the use of a heap was preferred. When executing the command delete min from the heap (delete min) the Huffman Tree is constructed, specifically for the pair of elements deleted from the min heap a node - element with a frequency field equal to the sum of the frequencies of the deleted pair of elements is created which has as right child the node - element with the lowest frequency of the two elements deleted from the heap and as left child the other node - element, the character field (char) remains empty in the new node. Next, we insert the new parent node into the min heap and follow the same steps by performing two successive min heap deletions, thus the Huffman Tree will start to form. In this way, we manage to select each time the nodes - elements with the smallest frequency fields and assign to them as parent a node - element with frequency field the sum of their frequencies. By iteratively executing the above procedure until we reach the root of the tree, i.e. until we arrive at a heap with two nodes - elements and perform delete min, we obtain the completed Huffman Tree. Finally, to assign a codeword to each character of the alphabet it is enough for each internal node of Huffman Tree to attach the value "0" to the edges leading to the left child and the value "1" to the other edges (the attachment of values to the edges can be done arbitrarily). The decoding process is made obvious and easy by using the Huffman tree constructed for the input text.

Regarding the dynamic self-adaptive Huffman algorithm, the following flowcharts make it easy to create classes and methods for text encoding and decoding. That is, the flowcharts adequately describe the dynamic Huffman algorithm as well as partition the encoding problem into smaller tractable subproblems. In addition, the dynamic Huffman algorithm has the property of generating a Huffman Tree on each character-symbol it encounters while scanning the text.

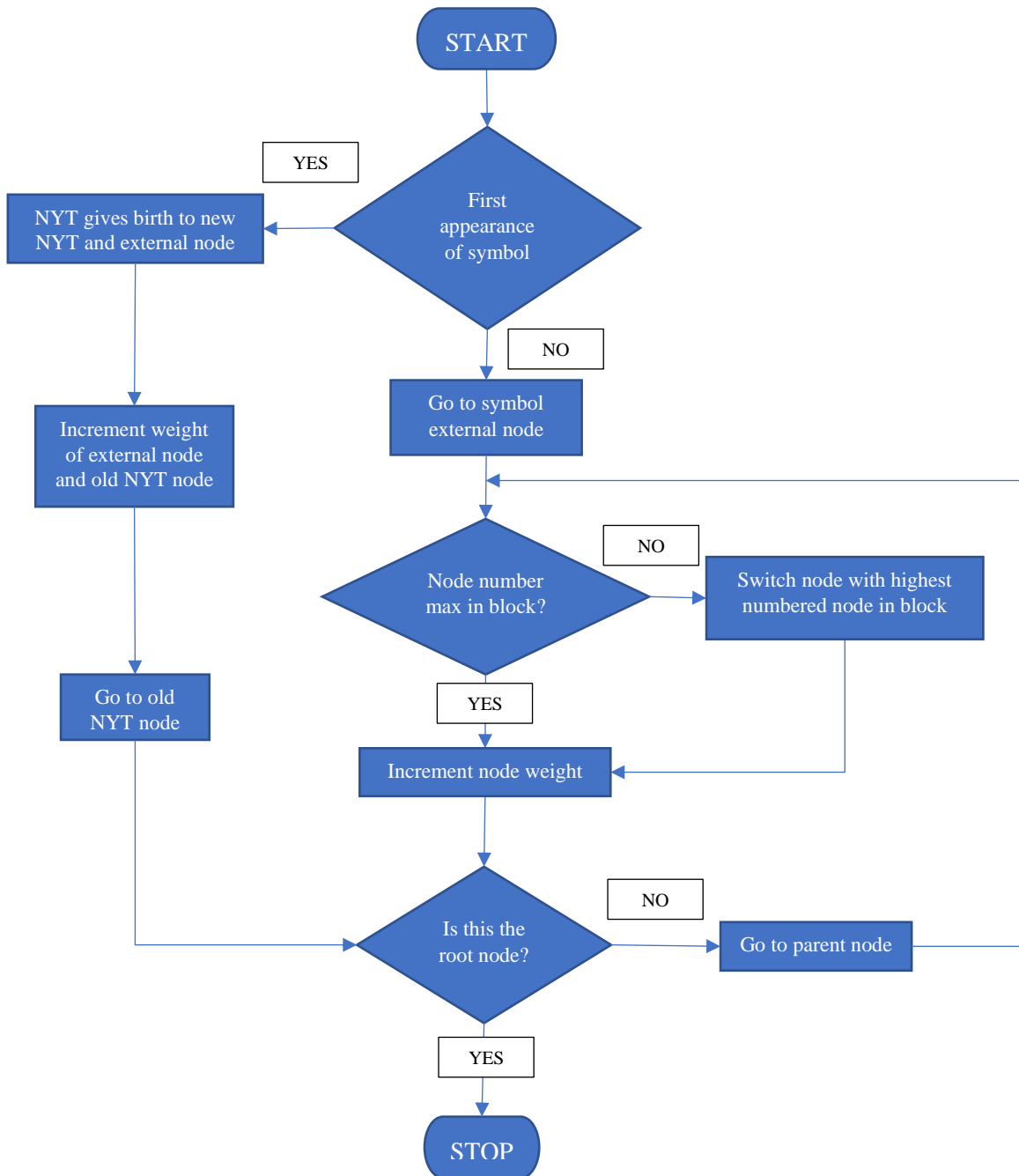
Encoding Procedure



Decoding Procedure



Update Procedure



Question 1

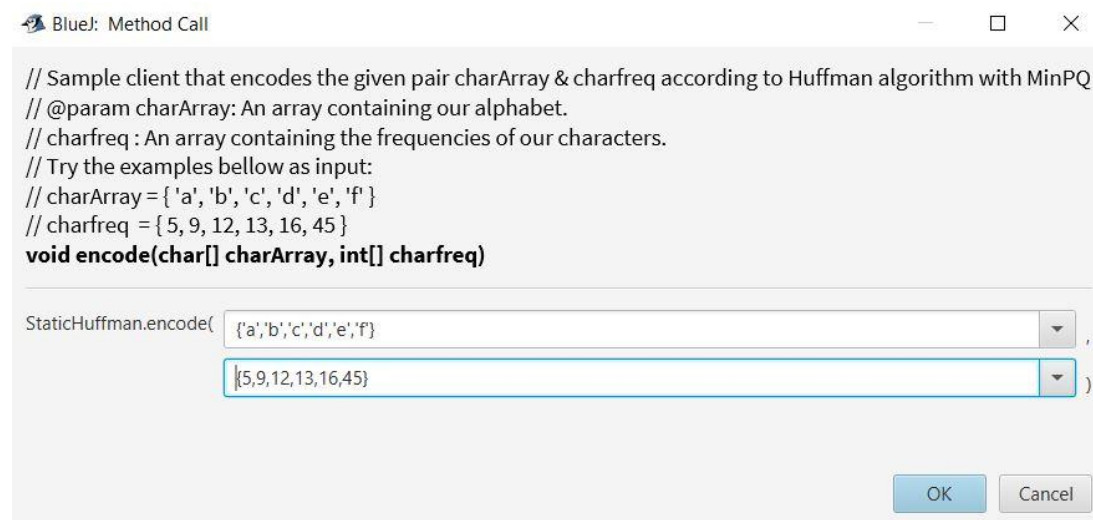
Write an encoder and decoder in Java for Huffman's static algorithm (for the case where the frequencies of each letter of the alphabet are known in advance).

Answer 1

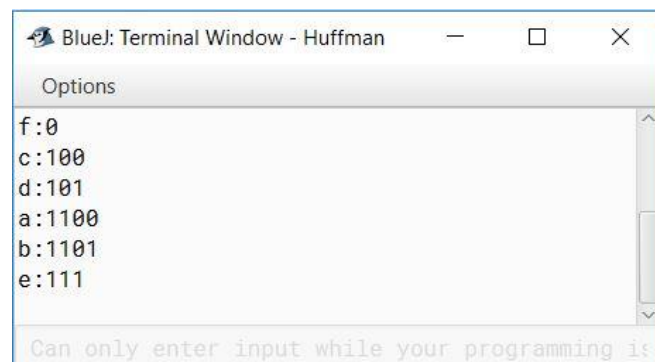
The case of Huffman's static algorithm was implemented by defining 3 classes, HuffmanNode, MyComparator and StaticHuffman. The MyComparator and HuffmanNode classes are secondary classes, the first one serves to construct heap-minimum in terms of the frequency domain of the nodes (i.e. nodes are inserted into the heap-minimum according to their frequency domain) and the second one serves to construct nodes as objects with fields the frequency, the character to be encoded, the left child of that node and the right child of that node. The StaticHuffman class is the base class in which the encoding and decoding of the input-text is done as described above in the reference to Huffman's static algorithm. In particular, the StaticHuffman class includes 3 methods, printCode, encode and decode each performs the obvious function indicated by its signature.

Example of encoding text with known frequencies of each letter:

[Input]:

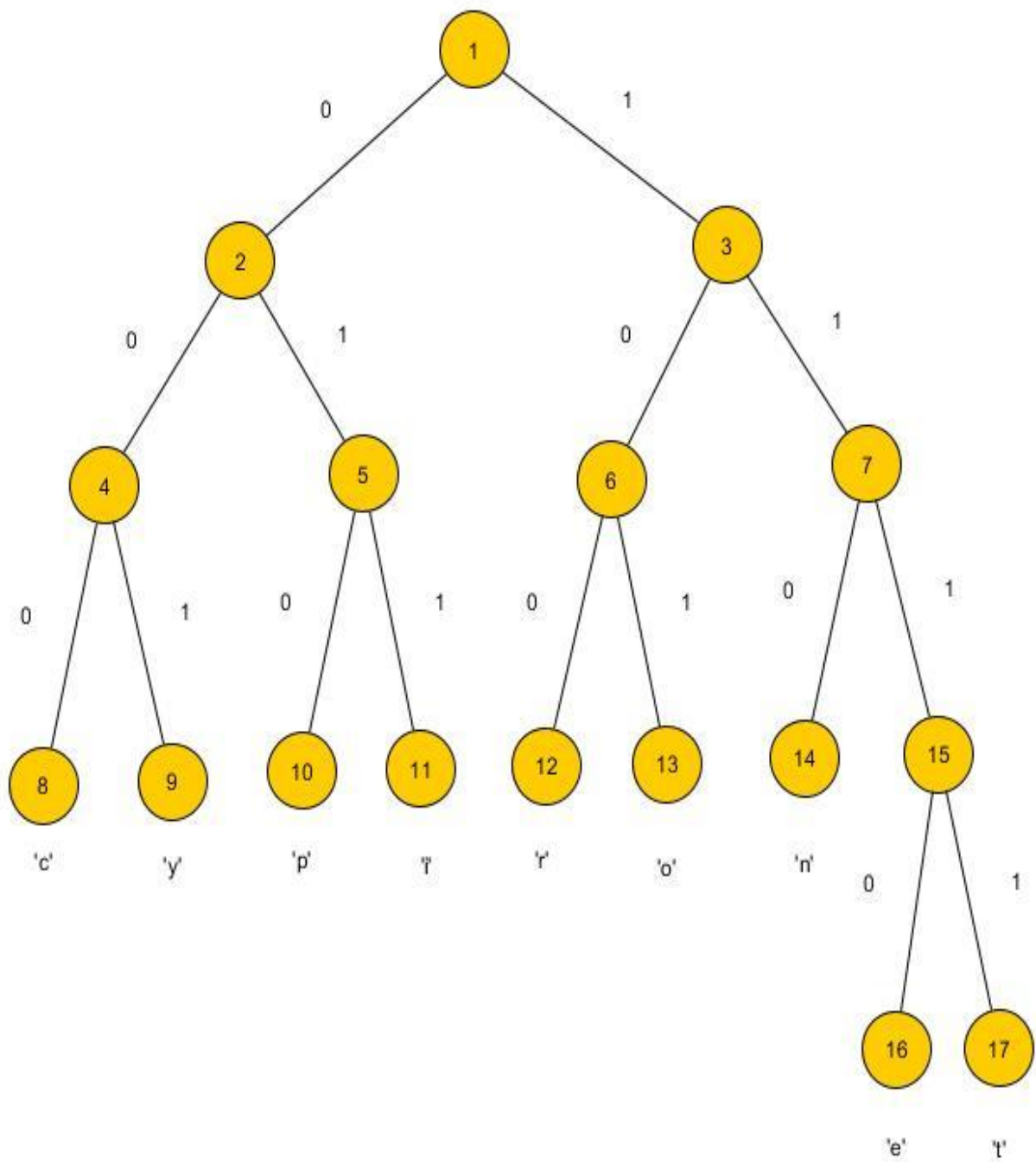


[Output]:

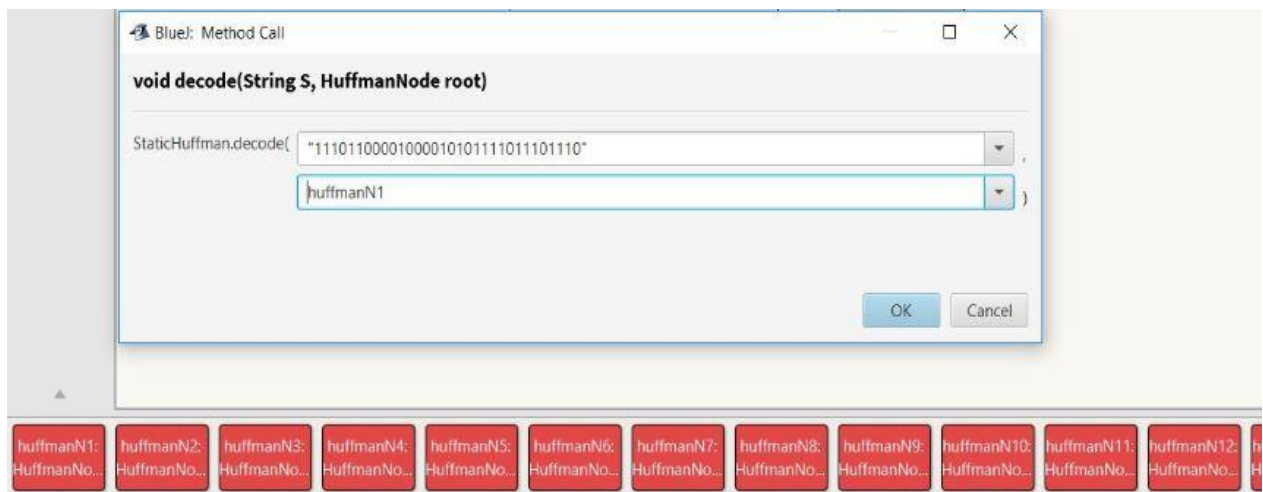


Example of text decoding given the corresponding Huffman tree:

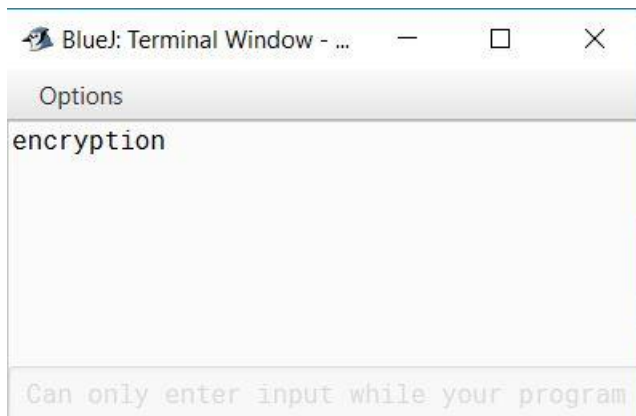
In this example, the possibility of decoding text using the corresponding Huffman tree is represented. In particular, the following Huffman tree is constructed for the word "encryption" and with its help the sequence "0" and "1" is decoded as shown in the call to the decode method.



[Input]:



[Output]:



Question 2

Write an encoder and decoder in Java for dynamic self-adaptive Huffman coding.

Answer 2

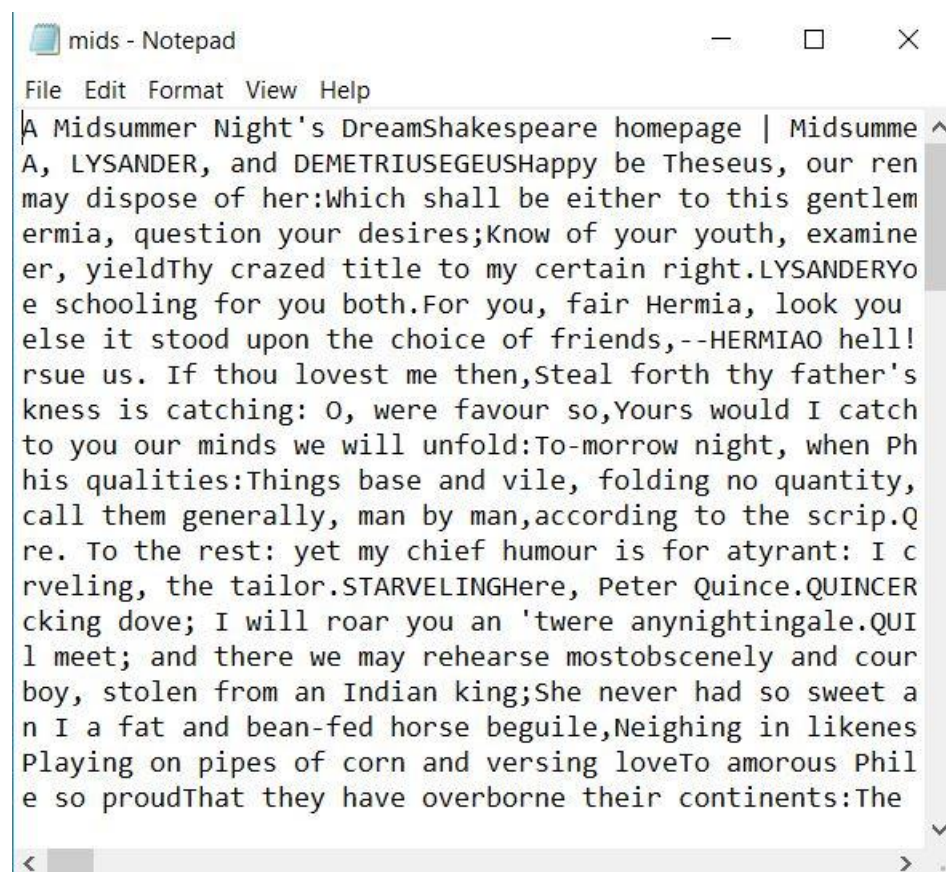
The case of the dynamic self-adaptive Huffman algorithm (adaptive Huffman algorithm) was not as obvious to implement as the previous case of the static algorithm. In particular, several attempts were made to create a project in BlueJ that closely follows the dynamic algorithm as shown in the introductory flowcharts. In fact, one of the attempts involved implementing the dynamic Huffman algorithm using heaps-minimum, as in the case of the static Huffman algorithm. The rationale of the aforementioned implementation was to construct a Huffman tree according to the static algorithm for each character scanned in the input - text, i.e. when the next character is scanned in the input-text the problem receives one more data by adding the new character scanned in the alphabet with the frequencies and for the new alphabet a Huffman tree is constructed (the Huffman tree is constructed according to the static algorithm, using heaps - minimum). According to this procedure a Huffman tree is constructed for each character scanned and inserted into the alphabet with frequencies, it becomes obvious that this way of working for encoding a text is quite costly.

The implementation that will be presented in this report is the product of web research on codes written in Java that compress .txt files using the adaptive Huffman method. For the implementation of the dynamic algorithm, it was necessary to create 6 classes, Node, Tree, Encoder, Decoder, BitInputStream and BitByteOutputStream of which the first 4 are the main ones, the last two contribute to reading the input files and creating the compressed - encoded files, respectively. In more detail, the Node class includes set and get type methods which are associated with its fields, parent, left, right, isNYT, isLeaf, weight, index and value each field has the obvious function indicated by its name. The Tree class handles the construction of a binary tree and its configuration to meet the properties of a Huffman Tree. That is, as shown in the flowchart of the Update Procedure mainly the Tree class contains methods such as findHighestIndexWeight, swap and updateTree which follow the flowchart to the letter. The Encoding and Decoding Procedures call the Update Procedure; in addition to this call, they perform various checks on the first appearance of symbol and whether the last symbol of the text is scanned (the Decoding Procedure also includes checks related to the Huffman tree).

Example of text encoding:

The text is within a .txt file which is attached below (mids.txt).

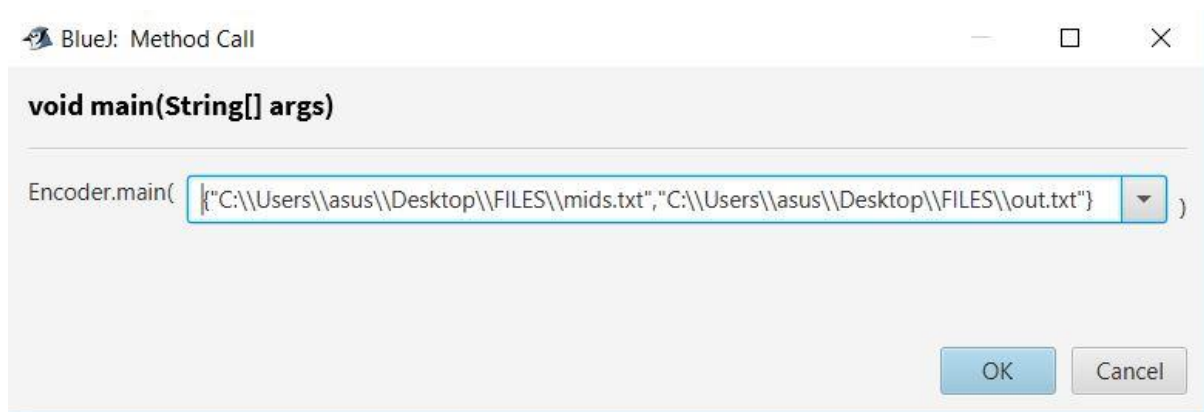
[Original Text]:



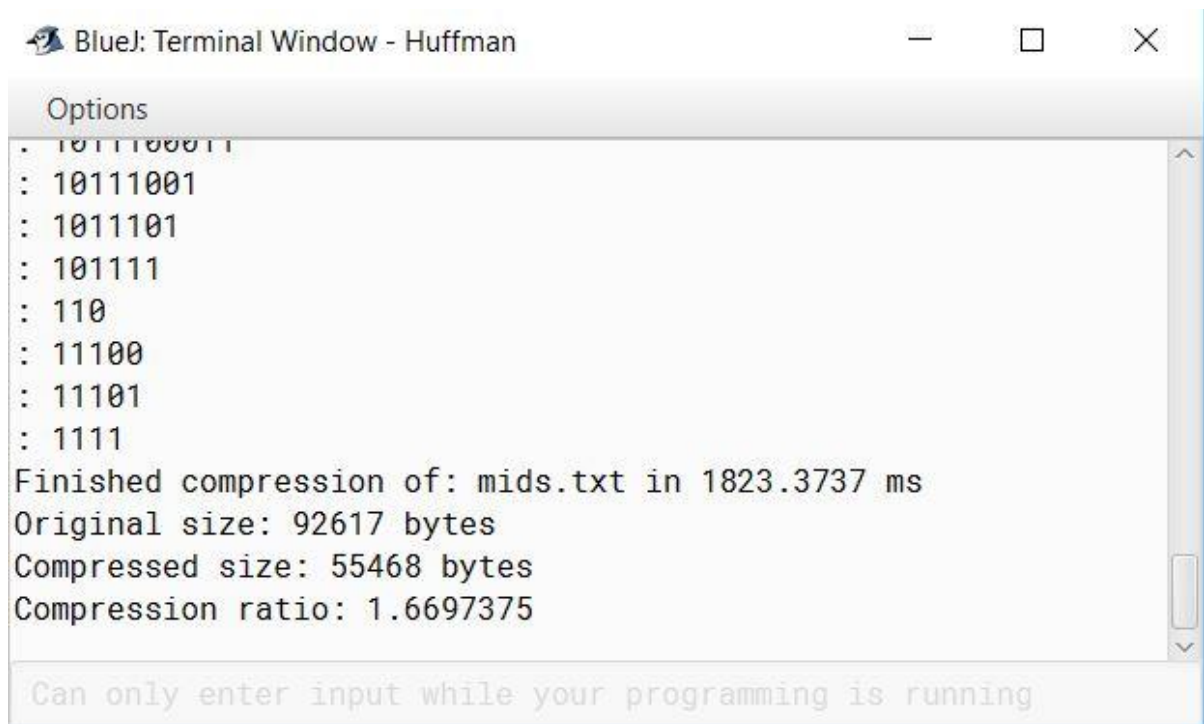
```
File Edit Format View Help
A Midsummer Night's DreamShakespeare homepage | Midsumme
A, LYSANDER, and DEMETRIUSEGEUSHappy be Theseus, our ren
may dispose of her:Which shall be either to this gentlem
ermia, question your desires;Know of your youth, examine
er, yieldThy crazed title to my certain right.LYSANDERYo
e schooling for you both.For you, fair Hermia, look you
else it stood upon the choice of friends,--HERMIAO hell!
rsue us. If thou lovest me then,Steal forth thy father's
kness is catching: O, were favour so,Yours would I catch
to you our minds we will unfold:To-morrow night, when Ph
his qualities:Things base and vile, folding no quantity,
call them generally, man by man,according to the scrip.Q
re. To the rest: yet my chief humour is for atyrant: I c
rveling, the tailor.STARVELINGHere, Peter Quince.QUINCER
cking dove; I will roar you an 'twere anynightingale.QUI
l meet; and there we may rehearse mostobscenely and cour
boy, stolen from an Indian king;She never had so sweet a
n I a fat and bean-fed horse beguile,Neighing in likenes
Playing on pipes of corn and versing loveTo amorous Phil
e so proudThat they have overborne their continents:The
```

We execute the main method of the Encode class by introducing as input a vector of Strings of length 2 positions. In the first position we enter the location of the .txt file to be encoded and in the second position we enter the location of the .txt file which will be the encoded - compressed file.

[Input]:



[Output]:



The above Terminal Window shows an extract of the encodings of each character encountered in the text within the mids.txt file, as well as the run time, the original and compressed size of the files and their compression ratio.

Attached below is the out.txt file which was compressed - encrypted according to Huffman's dynamic algorithm.

[Encrypted – Compressed Text]:



The decoding process works and can be verified by inserting in the main method of the Decoder class a vector of Strings of length 2 positions where the first position contains the file out.txt which is the encoding of the aforementioned mids.txt file and as the second argument an empty file, let's say test.txt, which will be the decoding of out.txt. If after the decoding process the files mids.txt and test.txt result identical, then we verify that decoding works smoothly.

Question 3

Write an encoder and decoder in Java for the LZ77 method and [optionally] for the LZ78 and LZW methods.

Answer 3

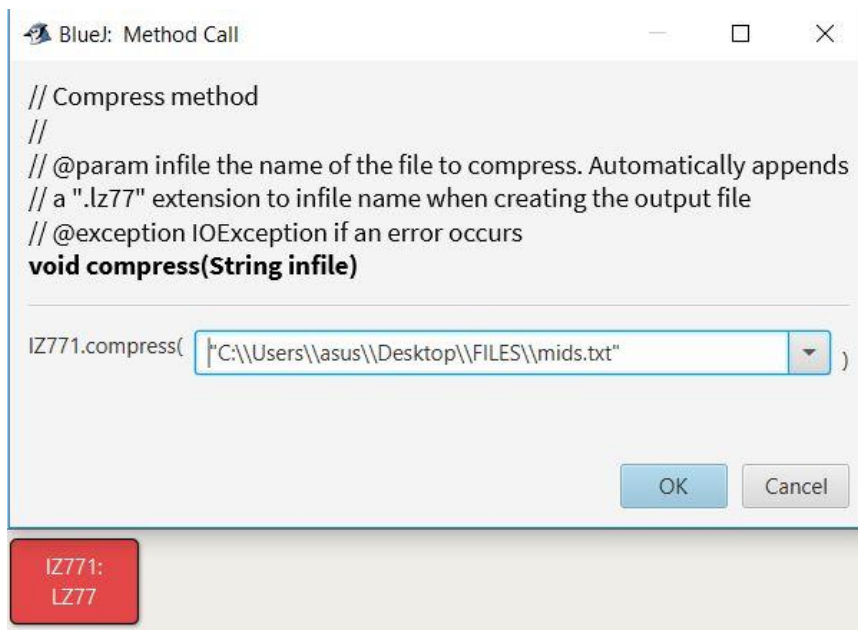
This answer includes only an encoder and decoder for the LZ77 method. The implementation of the LZ77 method has also been developed with the contribution of the Internet to a large extent in order to create compressed .txt files of selected texts which are given as input to the program. From theory we know that the process of encoding the input text is achieved with the help of a sliding window which consists of the search buffer and the look ahead buffer. A match is a String, which is created when two identical Strings are found in the search buffer and look ahead buffer, respectively. The first String of the two, starts *offset* positions to the left of the

look ahead buffer and the second String is located in the leftmost length positions of the look ahead buffer. The procedure followed by the LZ77 encoding algorithm is simple and repeats the detection of the largest match, the transmission of the triplet $\langle o, l, c \rangle$ (where the symbols represent $o = offset, l = length$ και $c = character$) and the transfer of the sliding window $l + 1$ positions to the right.

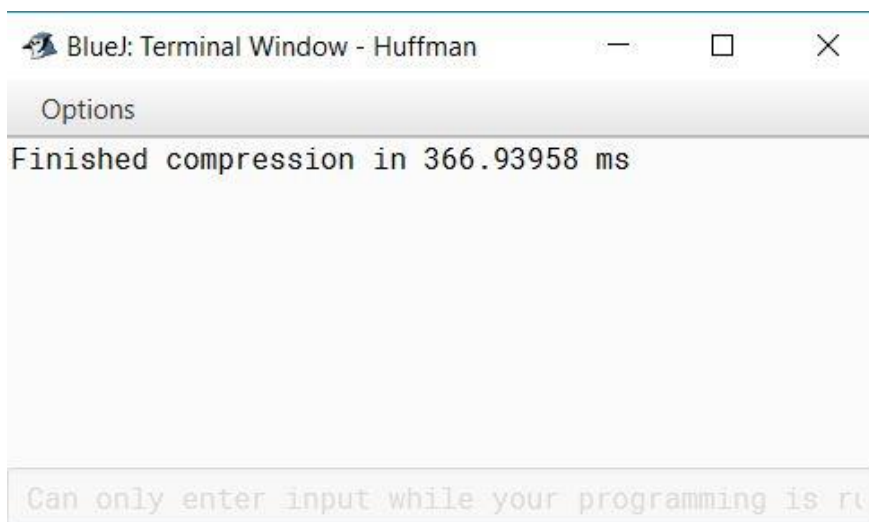
Example of text encoding:

The text is within a .txt file which was also encoded previously (mids.txt). Just construct an LZ77 object and execute the compress command, as shown below.

[Input]:



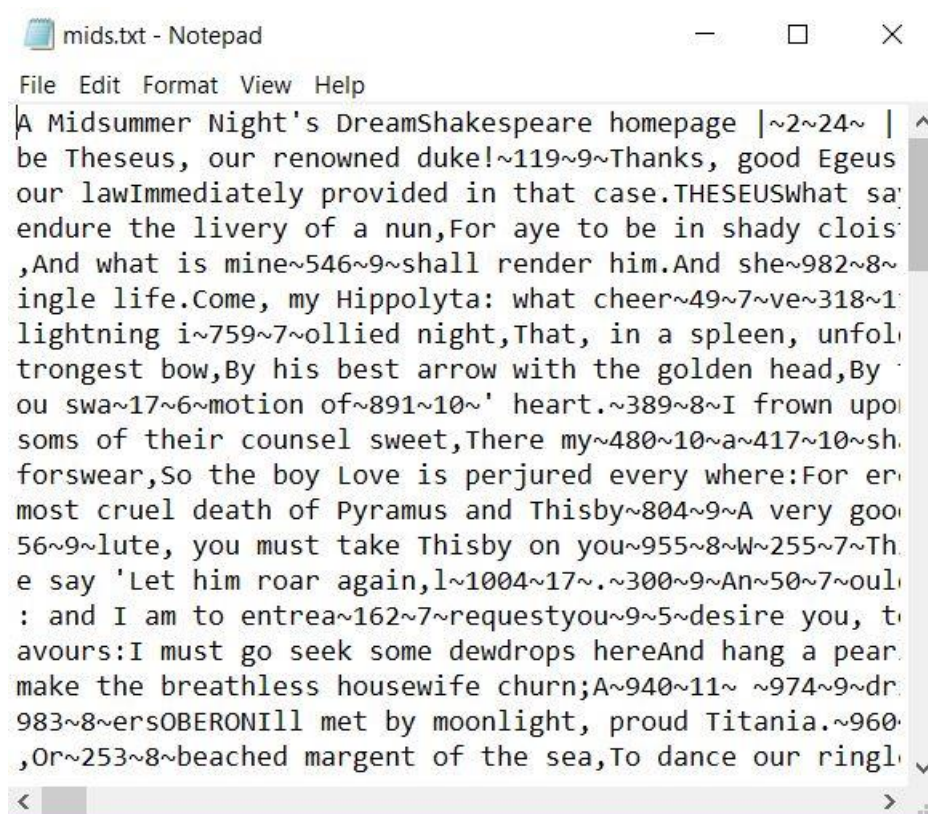
[Output]:



We observe that the execution time of the LZ77 compression method compared to the Adaptive Huffman compression method is significantly shorter. However, if we observe the compression efficiency by comparing initial and final file sizes we see that the Adaptive Huffman method outperforms LZ77.

COMPARISON	ADAPTIVE HUFFMAN	LZ77
INITIAL SIZE	91 KB	91 KB
FINAL SIZE	55 KB	88 KB

[Encrypted – Compressed Text]:



Question 4

Experimentally study the time required to run the above algorithms and their compression performance. Include the "zip" compression program in the above comparison.

Answer 4

In answers 2 and 3, the mids.txt file was compressed using Adaptive Huffman and LZ77 methods respectively and the results can be summarized in the table below as to the exact size of the final compressed file. The data for the compression with the "zip" program is again for the mids.txt file and was obtained by "compressing the file to zip".

COMPARISON	ADAPTIVE HUFFMAN	LZ77	ZIP
INITIAL SIZE	92,617 bytes	92,617 bytes	92,617 bytes
FINAL SIZE	55,468 bytes	89,209 bytes	38,153 bytes