Adaptive Huffman Coding

Elena Martina

1. Introduction

A static Huffman coding algorithm would firstly need to collect information of a given sequence before being able to construct a Huffman tree and encode the above-mentioned sequence. The basic Huffman technique requires indeed the encoding procedure to be divided in two different steps.

More dynamic algorithms were independently developed by Faller and Gallagher, then later improved by Knuth and Vitter (Sayood, 2006). Those algorithms, based on the statistics of the symbols already encountered, made it possible to transform encoding into a one-pass procedure, using what was then named Adaptive Huffman Coding, likewise the algorithm that I implemented and that will be described in this report.

1. Algorithms

1.1 Huffman Tree Update Procedure

The Update Procedure of the Huffman Tree is shown in the flowchart in Figure 1, and implemented in the code in the function "updateTree()".

The process of updating the Huffman Tree represents the core of the algorithm: its is needed to preserve the sibling property of the Tree after every iteration in both compression and decompression. The sibling property states that "each node (except the root) has a sibling and the nodes can be listed in order of non-increasing weight with each node adjacent to its siblings" (Ics.uci.edu, 2018).

In order to preserve this property, each node of the Tree has a weight and an order:

- ➢ The *weight* corresponds to the frequency of the symbol contained in the node for the leaf nodes, and to the sum of its two children nodes for the internal nodes.
- ➢ The *order* represents the position of a node in the Tree structure. The order values in the Tree increase left to right, top to bottom; the nodes with higher weight have a higher order, and the nodes with the same weight (that belong to the same weight-class) are in consecutive order.

Every Tree contains a ROOT node and a DAG node, that have respectively the highest and lowest order in the Tree.
When reading a symbol, the Tree is first checked to see if it is the first time encountering that symbol: if so, two new nodes are created from the DAG node. The new right child will contain the symbol and the new left child will be the new DAG node.
If a character is already in the tree, the weight of it's node is updated, then the weights the rest of the nodes in the Tree above are checked and updated. Each node is checked to see if it is the highest order node in its weight class; if not, the node is swapped with the node with the highest order; the weight of the node is then incremented.

1.2 Compression and Decompression

The Compression process is shown in the flowchart in Figure 2, and implemented in the code in the functions "encode()".
The Decompression process is shown in the flowchart in Figure 3, and implemented in the code in the functions "decode()".

At the beginning of both the encoding and decoding processes, the Tree consists of a single node, DAG.
Whenever the symbol to be encoded is already in the tree, the code for the symbol is generated by traversing the tree from the external node corresponding to the symbol to the ROOT.

Otherwise, if the symbol is being encountered for the first time, it will be encoded as the path from DAG to the ROOT, followed by the ASCII code of that particular symbol. The code of the path from the DAG node is obtained by traversing the Huffman Tree from the DAG node to the ROOT: if the node under consideration is the left child, "0" is added to the output sequence; "1" if it is the right child.
Encoding a new node with the path from DAG is necessary for one reason: During the decoding process, finding DAG alerts the decoder of the fact that the code that follows corresponds to the ASCII encoding of that symbol.

In the decompression procedure indeed, the algorithm receives a binary string; the tree is traversed in the opposite way to that used in the compression procedure, starting from the ROOT. Once a leaf node is found, the value corresponding to that leaf is decoded. If the leaf node is DAG instead, as mentioned previously, the symbol is decoded from its ASCII code.
Once the symbol has been encoded/decoded, the tree is updated as previously described in paragraph 1.1.

2. Implementation

The algorithm was implemented in the Java programming language.

The Menu class is responsible of running different programs with different functionalities. The different functionalities implemented are:

- ➢ Read an input File and compress it into an output File;
- ➢ Read a compressed input File and decompress it into an output File;
- ➢ Input different characters or strings to encode and add sequentially to a unique Huffman Tree;
- ➢ Input different binary code sequences to decode and add sequentially to an Huffman Tree;

The main functionalities of the program are enclosed in the HuffmanTree class. This class represents the structure of a tree composed of nodes and contains other two Data Structures useful for computational purposes:

- ➢ An ArrayList, "orders", that stores the nodes depending on their order: their index in the array corresponds to their order in the Tree. As ArrayLists are easy to manipulate, having this data structure simplified the process of checking and updating the orders of the nodes, as well as finding highest-order nodes in the weight classes.
- ➢ A HashMap, "alphabet", which maps each character to its corresponding leaf node. This data structure turned out to be really useful in order to easily and efficiently "get()" the node containing a particular character, as it is characterized by a $\Theta(1)$ access time;

Other two classes, "Compressor" and "Decompressor" take care respectively of the compressing and decompressing procedures, taking the inputs given in the Menu and calling the function in the HuffmanTree class to encode and decode the inputs.

3. Demonstration

The platform used to develop and run the program was IntelliJ Idea, Jetbrains' Java IDE.
The tests I run were the following:

- ➢ Compression and Decompression of the sequence "abcbbdaaddd" given in the Assignment; the screenshots of the output are shown in Figure 4.
- ➢ Compression and Decompression of all the different ASCII characters.
- ➢ Compression and Decompression of a long text.

4. Discussion

The algorithm follows correctly the Adaptive Huffman Coding method and updates the Huffman tree dynamically as each input is given in both the compression and decompression processes.
It was implemented and tested on different text files and character sequences and it is proven to work for any input in the ASCII character set. This is demonstrated from the tests described in section 3.

The performance of the algorithm is calculated with the compression ratio, calculated as the length of the compressed file over the length of the input file ("outputFile.length()/inputFile.length()"). The performance is higher for longer text files: the longest the sequence encoded, the best the compression ratio.

In order to improve the performance, many data structures were tested; in the end, the best way to encode the binary file was to create an array of bytes, and assign to each byte in the array a group of 8 bits.
This outperformed saving the encoded bits in a String by resulting in a four times better compression ratio.

References

[1] Sayood, K. (2006). Introduction to Data Compression (3rd ed.). 3rd ed.San Francisco: Morgan Kaufmann Publishers, pp.58-65.
[2] Ics.uci.edu. (2018). Data Compression - Section 4. Adaptive Huffman Coding. [online]
Available at: https://www.ics.uci.edu/~dan/pubs/DC-Sec4.html [Accessed 25 Nov. 2018].

List of Figures

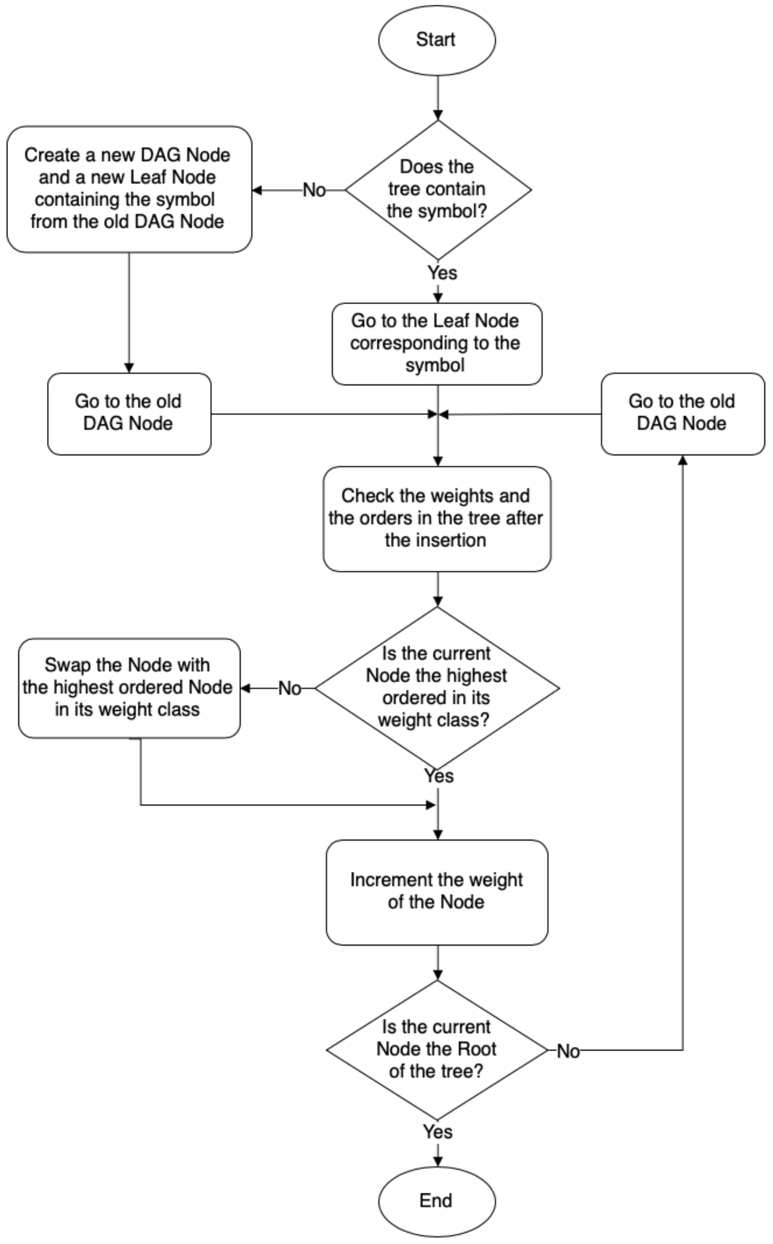

Start
Does the tree contain the symbol?
No
Create a new DAG Node and a new Leaf Node containing the symbol from the old DAG Node
Yes
Go to the Leaf Node corresponding to the symbol
Go to the old DAG Node
Go to the old DAG Node
Check the weights and the orders in the tree after the insertion
Is the current Node the highest ordered in its weight class?
No
Swap the Node with the highest ordered Node in its weight class
Yes
Increment the weight of the Node
Is the current Node the Root of the tree?
No
Yes
End

Figure 1. Flowchart showing the Update Tree Procedure.

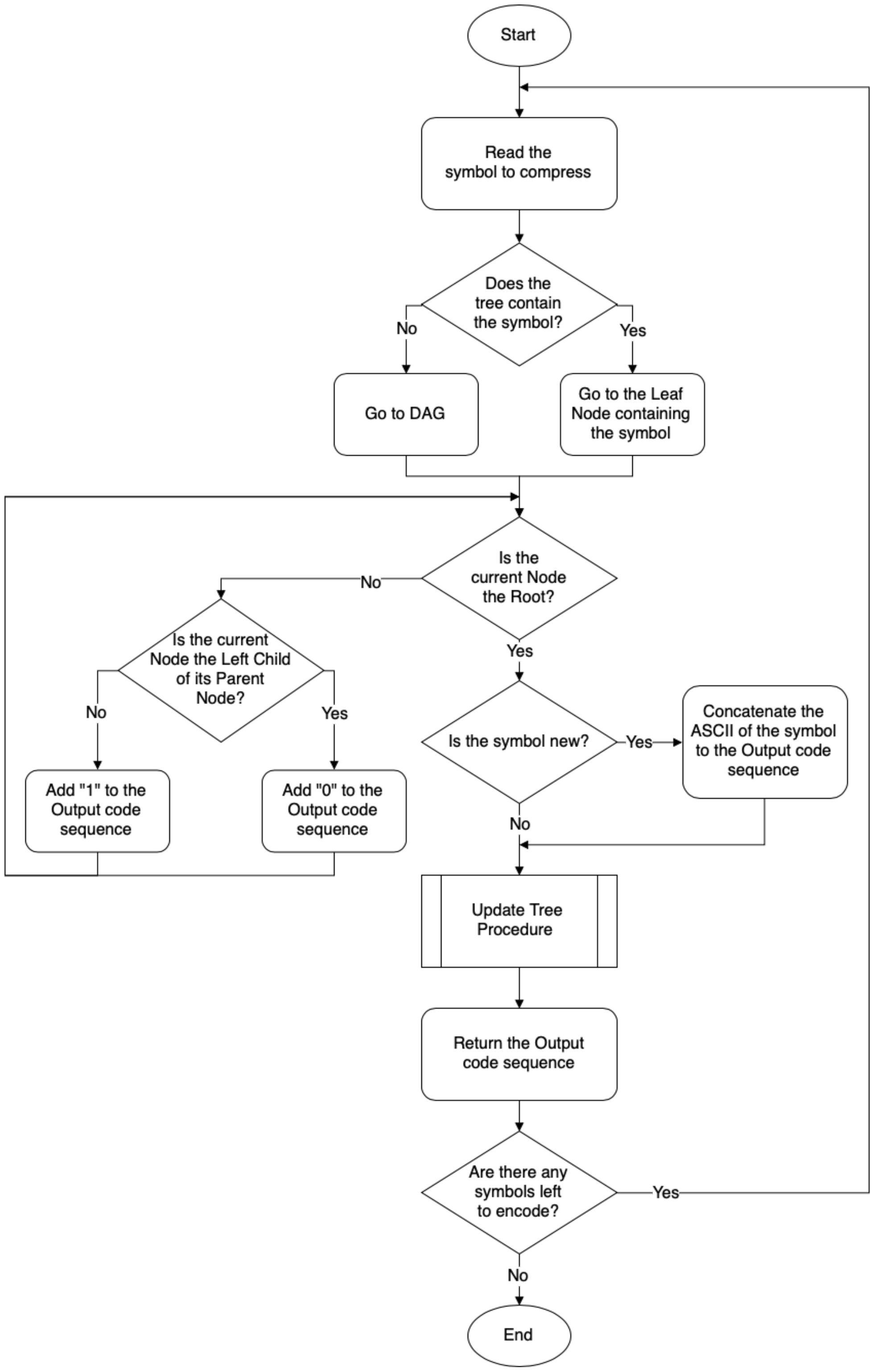
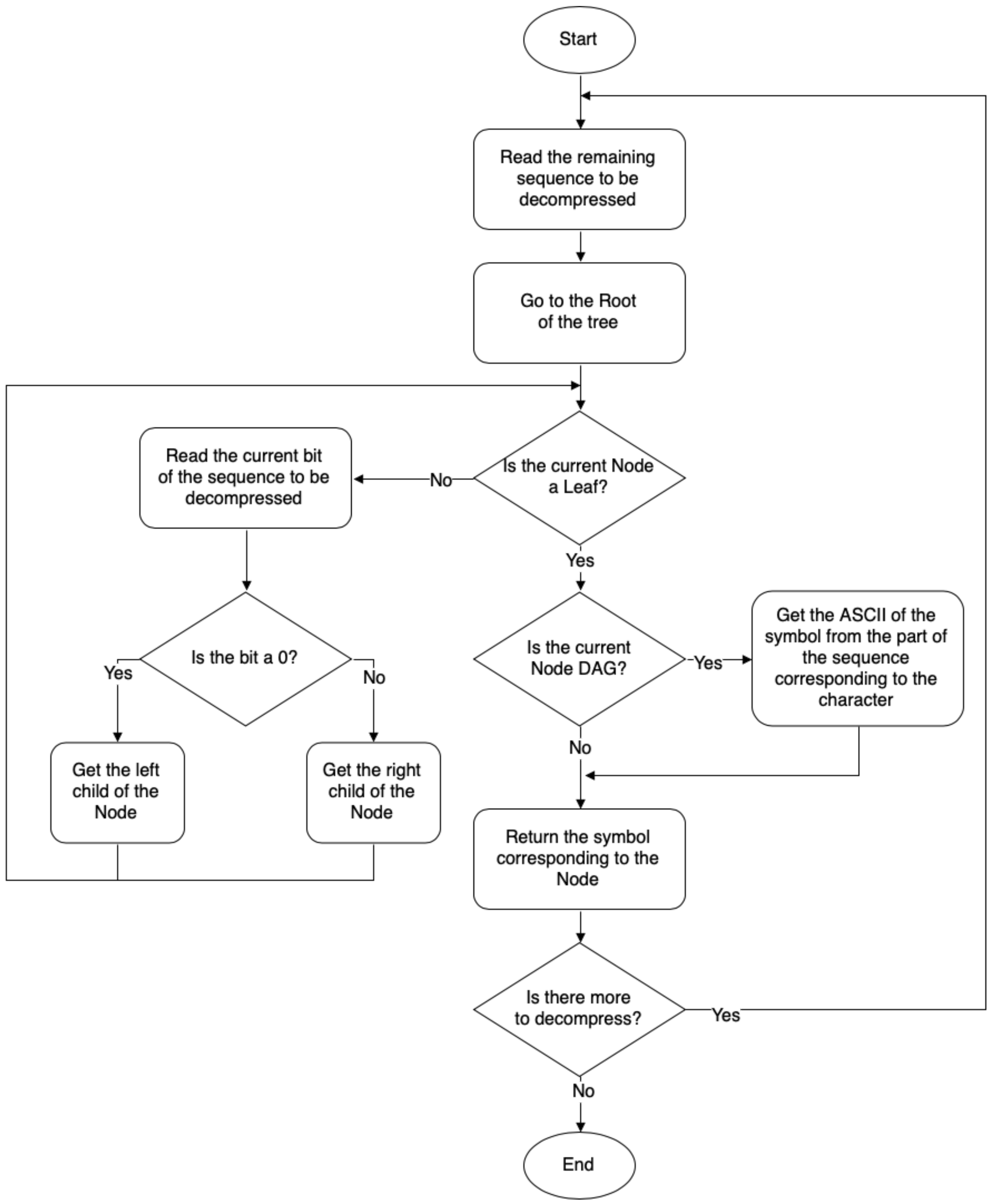
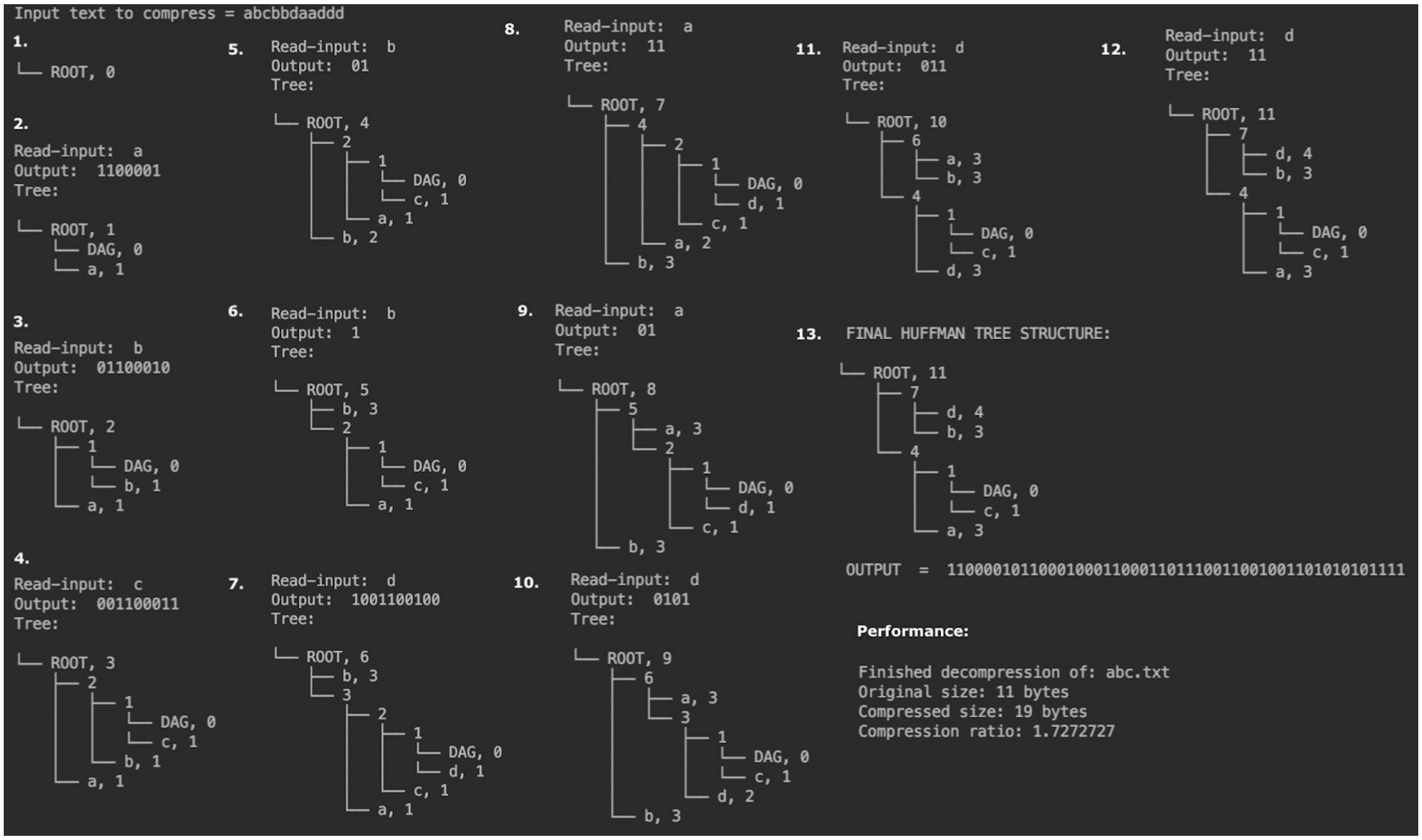

Start
Read the symbol to compress
Does the tree contain the symbol?
No
Yes
Go to DAG
Go to the Leaf Node containing the symbol
Is the current Node the Root?
No
Yes
Is the current Node the Left Child of its Parent Node?
No
Yes
Add "1" to the Output code sequence
Add "0" to the Output code sequence
Is the symbol new?
Yes
Concatenate the ASCII of the symbol to the Output code sequence
No
Update Tree Procedure
Return the Output code sequence
Are there any symbols left to encode?
Yes
No
End

Figure 2. Flowchart showing the Compression Procedure.



Start
Read the remaining sequence to be decompressed
Go to the Root of the tree
Is the current Node a Leaf?
No
Read the current bit of the sequence to be decompressed
Is the bit a 0?
Yes
No
Get the left child of the Node
Get the right child of the Node
Yes
Is the current Node DAG?
Yes
Get the ASCII of the symbol from the part of the sequence corresponding to the character
No
Return the symbol corresponding to the Node
Is there more to decompress?
Yes
No
End

Figure 3. Flowchart showing the Decompression Procedure.



```
Input text to compress = abcbbdaaddd
1.

└── ROOT, 0

2.
Read-input:  a
Output:  1100001
Tree:

└── ROOT, 1
    └── DAG, 0
    └── a, 1

3.
Read-input:  b
Output:  01100010
Tree:

└── ROOT, 2
    ├── 1
    │   └── DAG, 0
    │   └── b, 1
    └── a, 1

4.
Read-input:  c
Output:  001100011
Tree:

└── ROOT, 3
    ├── 2
    │   ├── 1
    │   │   └── DAG, 0
    │   │   └── c, 1
    │   └── b, 1
    └── a, 1

5.  Read-input:  b
    Output:  01
    Tree:

    └── ROOT, 4
        ├── 2
        │   ├── 1
        │   │   └── DAG, 0
        │   │   └── c, 1
        │   └── a, 1
        └── b, 2

6.  Read-input:  b
    Output:  1
    Tree:

    └── ROOT, 5
        ├── b, 3
        └── 2
            ├── 1
            │   └── DAG, 0
            │   └── c, 1
            └── a, 1

7.  Read-input:  d
    Output:  1001100100
    Tree:

    └── ROOT, 6
        ├── b, 3
        └── 3
            ├── 2
            │   ├── 1
            │   │   └── DAG, 0
            │   │   └── d, 1
            │   └── c, 1
            └── a, 1

8.  Read-input:  a
    Output:  11
    Tree:

    └── ROOT, 7
        ├── 4
        │   ├── 2
        │   │   ├── 1
        │   │   │   └── DAG, 0
        │   │   │   └── d, 1
        │   │   └── c, 1
        │   └── a, 2
        └── b, 3

9.  Read-input:  a
    Output:  01
    Tree:

    └── ROOT, 8
        ├── 5
        │   ├── a, 3
        │   └── 2
        │       ├── 1
        │       │   └── DAG, 0
        │       │   └── d, 1
        │       └── c, 1
        └── b, 3

10. Read-input:  d
    Output:  0101
    Tree:

    └── ROOT, 9
        ├── 6
        │   ├── a, 3
        │   └── 3
        │       ├── 1
        │       │   └── DAG, 0
        │       │   └── c, 1
        │       └── d, 2
        └── b, 3

11. Read-input:  d
    Output:  011
    Tree:

    └── ROOT, 10
        ├── 6
        │   ├── a, 3
        │   └── b, 3
        └── 4
            ├── 1
            │   └── DAG, 0
            │   └── c, 1
            └── d, 3

12. Read-input:  d
    Output:  11
    Tree:

    └── ROOT, 11
        ├── 7
        │   ├── d, 4
        │   └── b, 3
        └── 4
            ├── 1
            │   └── DAG, 0
            │   └── c, 1
            └── a, 3

13.  FINAL HUFFMAN TREE STRUCTURE:

    └── ROOT, 11
        ├── 7
        │   ├── d, 4
        │   └── b, 3
        └── 4
            ├── 1
            │   └── DAG, 0
            │   └── c, 1
            └── a, 3

    OUTPUT  =  11000010110001000110001101110011001001101010101111

     Performance:

     Finished decompression of: abc.txt
     Original size: 11 bytes
     Compressed size: 19 bytes
     Compression ratio: 1.7272727
```

Figure 4. Output of the program compressing the character sequence "abcbbdaaddd". The output of the decompression creates the same tree, therefore is the same.