Sorting and Searching Assignment 3

Yacob Ben Youb en Youssef Ben Youb
500672040 - 500775494

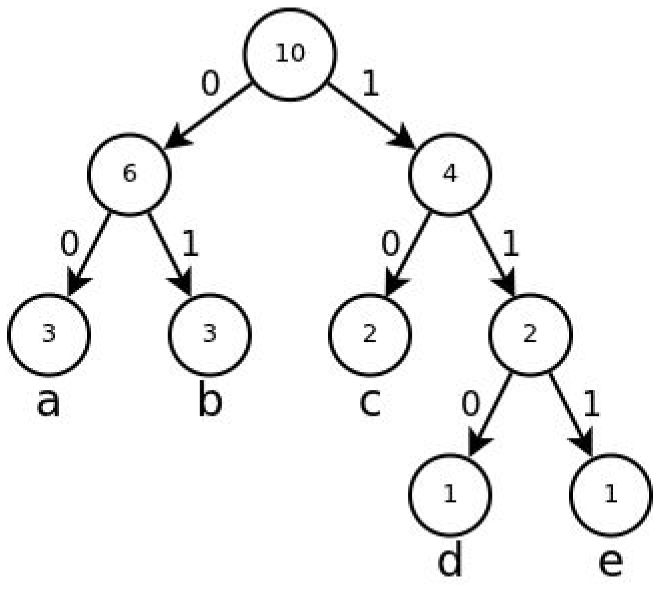


Table of contents:

Introduction **4**

Classes **4**

- HuffmanCompression 4
- Node 4
- HuffmanCompressionTest 4
- ExtendedHuffmanComressionTest 4
- NodeTest 4
- ExtendedNodeTest 4

Methods **5**

- HuffmanCompression 5
  - public HuffmanCompression(String text) 5
  - ArrayList<Node> createNodeList() 5
  - Node getCompressionTree() 6
  - public double getCompressionRatio() 7
  - ArrayList<String> createCodeList(Node Node, StringBuilder str) 8
  - String[] getCodes() 8
  - public double getCompressionRatio() 9
  - Public Node(int weight, character c) 10
  - Node(Node left, Node Right) 10
  - Public int compareTo(Node o) 10
  - void createOutput(ObjectOutputStream output) throws IOException 10
  - public void write(ObjectOutputStream output) throws IOException 11
  - public static Node read(ObjectInputStream input) throws IOException, ClassNotFoundException 11

Test methods: **12**

- HuffmanCompressionTest 12
  - @Before public void setup() 12
  - @Test public void checkWeightSimple() 12
  - @test public void checkUniqueCharacter() 13
  - @test public void checkSimpleCompressionRatio() 13
- ExtendedHuffmanCompressionTest 14
  - @Test public void checkNodeValueDistribution() 14
  - @Test public void evenLeavesMeansDoubleNodes() 14
  - @Testpublic void codeListBinaryValueTest() 14
- NodeTest 15
  - @test public void canReadWhatWasWritten() 15
- ExtendedNodeTest 16
  - @Test public void OutputStreamIsPreOrder() throws IOException, ClassNotFoundException () 16

@Testpublic void canReadWhatWasWrittenBigTree() throws IOException, ClassNotFoundException 17

Output **18**

Conclusion and Discussion **19**

Discussion 19

Introduction

This report will discuss the implementation of the huffman tree encoding algorithm and what the results were of the usage of this algorithm.

Classes

In this chapter the classes in the project are discussed.

HuffmanCompression

This class reads a file, compresses it and generates a Huffman tree. Via the usage of the huffman tree the compressed file can be decompressed.

Node

This class contains the information for Node objects. These Nodes are used to generate a Huffman tree.

HuffmanCompressionTest

Checks whether the huffman compression methods work well such as the amount of leaves on the tree and the total compression rate using these Nodes with their weight.

ExtendedHuffmanComressionTest

Extra methods to check whether the huffman compression methods work according to the actual requirements

NodeTest

Contains methods to test whether Nodes can be written to an output stream and then be decompiled in pre-order

ExtendedNodeTest

Tests extra attributes for Nodes to make sure trees are all well constructed, as well as more specific writing and reading tests which test whether recursive writing and reading works well.

Methods

In this chapter the most important methods which were implemented and added are discussed

HuffmanCompression

public HuffmanCompression(String text)

There are two ways to create a huffmanCompression Object. The first accepts a basic String.

```
public HuffmanCompression(String text) {
   this.text = text;
}
```

The second one uses a Scanner to convert an inputstream to a string, the scanner is then written to the text String which will be used to create the trees

```
public HuffmanCompression(InputStream input) {
   Scanner sc = new Scanner(input);
   sc.useDelimiter("\\Z"); // EOF marker
   text = sc.next();
}
```

ArrayList<Node> createNodeList()

Creates a sorted list of Nodes in descending order based on their amount of occurence. Each Node contains a character with the amount of occurrences it has.

First an array is created with each letter in the 8 bit alphabet (128 letter total). It then scans the text which needs to be converted to Huffman, and counts the amount of occurrences of each character.

```
int maxAscii = 128;
int charsInText = 0;
int letterCount[] = new int[maxAscii];
for (int i = 0; i < maxAscii; i++) {
   letterCount[i] = 0;
}
for (int i = 0; i < text.length(); i++) {
   int currentChar = text.charAt(i);
   letterCount[currentChar]++;
}
for (int i = 0; i < maxAscii; i++) {
   if (letterCount[i] > 0) {
       ++charsInText;
   }
}
```

It then goes through the list and finds each character which has occurred at least once, and puts those characters along with the occurence in a Node, and then adds that Node to a list. This list consists of all leaf Nodes. When all characters are added the list is sorted in descending order.

```
for (Integer c = 0; c <= charsInText; c++) {
   {
       int biggestNumber = 0;
       int biggestIndex = 0;
       for (int i = 0; i < maxAscii; i++) {
           if (letterCount[i] > biggestNumber) {
               biggestIndex = i;
               biggestNumber = letterCount[i];
           }
       }
       if (biggestNumber > 0) {
           Nodelist.add(new Node(letterCount[biggestIndex], (char) biggestIndex));
           letterCount[biggestIndex] = 0;}}
```

Node getCompressionTree()

Creates a huffman tree and returns the root Node.
First a NodeList with all Leaves in descending order is created. While there are more than two Nodes in the list, a new branch is created with the lowest Node at its left and the semi-lowest Node at its right. The weight of this Node is then set to the combined amount of occurences of both characters beneath it.

The lowest two Nodes are taken out of the list and the new combined Node is added back in. The list is then sorted again in descending order. This is done until there are only two Nodes left in the list. A root Node is then created using these two Nodes as its left and right Node, and this root Node is then returned.

```
  ArrayList<Node> NodeList = createNodeList();
       while (NodeList.size() > 2) {
           Node Node1 = NodeList.get(NodeList.size() - 1);
           NodeList.remove(NodeList.get(NodeList.size() - 1));
           Node Node2 = NodeList.get(NodeList.size() - 1);
           NodeList.remove(NodeList.get(NodeList.size() - 1));
           NodeList.add(new Node(Node1, Node2));
           Collections.sort(NodeList, Collections.reverseOrder());
       }
       Node root = new Node(NodeList.get(NodeList.size() - 1), NodeList.get(NodeList.size() - 2));
       return root;
```

public double getCompressionRatio()

```
{
   Node root = getCompressionTree();
   int originalBits = root.getWeight() * 8;// original would be 8 bits per char = total occurrences
*8
   int totalShortenedChar = 0;

   ArrayList<String> nodeCodes = createCodeList(root, new StringBuilder());
   ArrayList<Node> charsWithWeight = createNodeList();

   for (Node node : charsWithWeight) {
       for (String nodeCodeString : nodeCodes) {
           if (nodeCodeString.charAt(1) == node.getCharacter()) {
               totalShortenedChar += node.getWeight() * (nodeCodeString.length() - 6);
           }
       }
   }
   System.out.println("Original amount of bits: " + originalBits);
   System.out.println("Shortened amount of bits: " + totalShortenedChar);
   return (double) totalShortenedChar / (double) originalBits;
}
```

ArrayList<String> createCodeList(Node Node, StringBuilder str)

This is a recursive method which uses a root Node to find out the paths to all its sub-Nodes. An empty stringbuilder needs to be passed as argument when calling this method on a root node.

An arraylist of Strings is made to return. The current Node is checked on character presence to determine whether it's a leaf. If it is not, the createCodeList method is used on the left and right Node, with a 0 or 1 added to their parameters respectively. Whenever it finds a leaf, the Stringbuilder is converted to a normal String which is then added to the addingList in the notation specified in the JavaDocs requirements. When every recursive call has ended the final Arraylist with codes is returned

```
ArrayList<String> createCodeList(Node Node, StringBuilder str) {
   ArrayList<String> addingList = new ArrayList<>();
   StringBuilder leftString = new StringBuilder(str.toString());
   StringBuilder rightString = new StringBuilder(str.toString());
   if (Node.getCharacter() != null) {
       addingList.add("'" + Node.getCharacter() + "'" + "-> " + str);
   } else {
       for (String s : createCodeList(Node.getLeft(), leftString.append("0"))) {
           addingList.add(s);
       }
       for (String s : createCodeList(Node.getRight(), rightString.append("1")))
{
           addingList.add(s);
       }
   }
   return addingList;
}
```

String[] getCodes()

Uses the CreateCodeList() method to create Arraylist of Strings which contain the characters and their huffman codes. This Arraylist is then transferred to a new Array of Strings, which is then returned.

```
{
   Node root = getCompressionTree();
   ArrayList<String> nodeCodes = createCodeList(root, new StringBuilder());
   String codes[] = new String[nodeCodes.size()];
   for (int i = 0; i < nodeCodes.size(); i++) {
       codes[i] = nodeCodes.get(i);
   }
   return codes;
}
```

public double getCompressionRatio()

Returns the compression ratio of a file which is compressed using the huffman algorithm in this class.

A new huffman tree is created using the getCompressionTree() method. The root node of this tree contains the total amount of character occurrences. Because we can assume that every character in the original text is written using with 8 bits, we can simply multiply the amount of occurrences in the root node with 8 to get the original amount of bits.

```
{
   Node root = getCompressionTree();
 int originalBits = root.getWeight() * 8;// original would be 8 bits per char = total occurrences *8
   int totalShortenedChar = 0;
```

Now the compressed bits need to be calculated. The nodeCodes method is used to create an Arraylist containing the code of each character. Then the createNodeList method is used to create a list of every original node with its total amount of occurrences in the text.

Because the requirements state to note every character and code combination in a specific way, the character can be extracted from the codelist by getting the second index of every codeList string.

If this matches the a character in the Nodelist, the amount of bits at the end of the string are counted, (-6 to account for all the letters which are always present) and then multiplied by the occurrences of that character. This is then added to the total amount of bits taken up by shortened characters. The returned value is the compressed amount of bits divided by the original amount of bits.

```
   ArrayList<String> nodeCodes = createCodeList(root, new StringBuilder());
   ArrayList<Node> charsWithWeight = createNodeList();

   for (Node node : charsWithWeight) {
       for (String nodeCodeString : nodeCodes) {
           if (nodeCodeString.charAt(1) == node.getCharacter()) {
               totalShortenedChar += node.getWeight() * (nodeCodeString.length() - 6);
           }
       }
   }
   System.out.println("Original amount of bits: " + originalBits);
   System.out.println("Shortened amount of bits: " + totalShortenedChar);
   return (double) totalShortenedChar / (double) originalBits;
}
```

Public Node(int weight, character c)

This constructor creates a leaf Node. It contains the information of how often a character is used.

```
public Node(int weight, Character c) {
   this.weight = weight;
   this.character = c;
}
```

Node(Node left, Node Right)

This constructor creates a branch Node. This Node contains the information of 2 other Nodes as well as the combined weight of these nodes.

```
public Node(Node left, Node right) {
   this.weight = left.weight + right.weight;
   this.left = left;
   this.right = right;
}
```

Public int compareTo(Node o)

This method compares the weight of one Node to another Node. It is used by the Collections.sort method which is used to sort a list in the getCompressionTree and createNodeList methods.

```
public int compareTo(Node o) {
   return weight - o.getWeight();
}
```

void createOutput(ObjectOutputStream output) throws IOException

Writes the huffman tree to an Outputstream recursively. First a node is checked for characters. If it contains a character, the node's weight (occurrence value), and then the Node's character are written to the outputstream.

If it does not contain any characters a "null" object is written to the Outputstream which indicates that this node is not a Leaf. Then the createOutput method is called on its left sub-node, and then on its right sub-node which recursively writes each node beneath the current node to the output stream **in pre-order.**

```
   try {
       if (character == null){
           output.writeObject(null);
           left.createOutput(output);
           right.createOutput(output);
       }
       else {
           output.writeObject(weight);
           output.writeObject(character);
       };
   } catch (Exception ex) {
       ex.printStackTrace();
   }
}
```

public void write(ObjectOutputStream output) throws IOException

This method uses the createOutput method to write an Outputstream, and then closes this stream. The reason this is in a separate method is because it's not possible to recursively write to an Output stream after one recursive call closes it. The entire Output stream needs to be finished before it can be closed.

```
{
   createOutput(output);
   output.flush();
   output.close();
}
```

public static Node read(ObjectInputStream input) throws IOException, ClassNotFoundException

This method does the same as the output method but instead of creating output from Nodes it takes Input and converts it to Nodes. Because we can assume the data is written to the Outputstream in pre-order, the original tree can be reconstructed from solely the characters, weights and null values.

```
{
  Object temp = input.readObject();
   if (temp == null){
       return new Node(read(input),read(input));
   }
```

Each object in the Inputstream is checked separately. If the object has a null value, it is a node, which means that two lower nodes need to be created upon which this method is called again.

If the object is not null, it means the object and the next one contain data which can be used to create a node. The weight value is the value which was just read, and the character in the Node is the next Object in the Inputstream. Whenever an object is read from the Inputstream is is removed afterwards.

```
else {
       Node newNode = new Node((int) temp, (char)input.readObject());
       return newNode;
   }
}
```

The final return will be the root node with the tree constructed recursively on both sides, **in pre-order.**

Test methods:

This chapter contains the test classes and their methods.

HuffmanCompressionTest

Here the provided Huffman class tests are explained.

@Before public void setup()

This method reads the file "/edu/princeton/cs/algs4/Huffman.java" and saves the text to the Huffman compressor variable before the tests are executed. Other tests can use this huffman tree, or overwrite its value.

```
{
   compressor = new
HuffmanCompression(getClass().getResourceAsStream("/edu/princeton/cs/algs4/Huffman.java"));
}
```

@Test public void checkWeightSimple()

First this test makes the “compressor variable” read a new piece of text which makes the compressor variable save that text. Then it creates a node which contains the information for other nodes. This is made using the text of the compressor variable.

It asserts whether the total weight of the tree is equal to three. Which it should be because it contains three characters.

It then asserts whether the left node has a weight of one. It should have a weight of one because there is one b and the node with the lower weight is stored on the left side of a node.

Finally it asserts whether the right node has a weight of two. There are two a’s and the node with a higher weight is stored on the right side, so it should be equal to two.

```
  compressor = new HuffmanCompression("aba");
   Node compressionTree = compressor.getCompressionTree();
   assertEquals(3, compressionTree.getWeight());
   Node left = compressionTree.getLeft();
   Node right = compressionTree.getRight();
   assertEquals(1, left.getWeight());
   assertEquals(2, right.getWeight());
}
```

@test public void checkUniqueCharacter()

This test checks whether the amount of unique characters in the text is equal to 87 or 88. It does this by asserting whether the size of the array created by the getCodes method is equal to 87 or 88.

```
  // Handle Linux/Mac and Windows end-of-line characters, 87 and 88 are both ok.
   int numberOfChars = compressor.getCodes().length;
   assertTrue("You appear to have some very strange end-of-line configuration on your machine!",
numberOfChars == 87 || numberOfChars == 88);
}
```

@test public void checkSimpleCompressionRatio()

This test compresses the text “aba”. It then asserts whether the compression ratio is equal to 0.125. The compression ratio should be equal to 0.125 as there are only 2 letters. These can be saved in 1 bit: 0 and 1. Normally these are saved in 8 bit so it it should be compressed to 1 eight of its original size.

```
   compressor = new HuffmanCompression("aba");
   assertEquals(0.125,compressor.getCompressionRatio(), 0.0001);
}
```

ExtendedHuffmanCompressionTest

This chapter contains the extra tests which are custom written to test on compliance of the Huffman class requirements.

@Test public void checkNodeValueDistribution()

Tests whether a huffman tree is correctly constructed, and whether the weight of each character and the root node is correct.

```
@Test
public void checkNodeValueDistribution() {
   compressor = new HuffmanCompression("aaabbbbccccccccccc"); // 3*codeListBinaryValueTest, 4*b, 11*c
   Node root = compressor.getCompressionTree();
   assertEquals(18, root.getWeight());
   assertEquals(3, root.getLeft().getLeft().getWeight());
   assertEquals(4, root.getLeft().getRight().getWeight());
   assertEquals(11, root.getRight().getWeight());
}
```

@Test public void evenLeavesMeansDoubleNodes()

This test is a more advanced version of the checkWeightSimple() test. and also tests for Nodes. It checks whether the Huffman tree contains a correct amount of nodes and Leaves, whether the nodes are empty and the Leaves have correct weight.

```
 compressor = new HuffmanCompression("aabbccddeeffgghh"); // 2-2 2-2 2-2 2-2
   Node root = compressor.getCompressionTree();
   assertEquals(16, root.getWeight());
   assertEquals(2, root.getLeft().getLeft().getLeft().getWeight());
   assertEquals(2, root.getLeft().getRight().getRight().getWeight());
   assertEquals(2, root.getRight().getRight().getRight().getWeight());
   assertEquals("isNull", null, root.getRight().getCharacter());
   assertEquals("isNull", null, root.getRight().getRight().getCharacter());
```

@Testpublic void codeListBinaryValueTest()

This test checks whether the binary codes created by the createCodeList() method are correct. It creates a Binary tree with 4 leaves and checks whether first most left navigation value is 00 and the last most right navigation value is 11. This also confirms the codeList is in pre-order.

```
{
   compressor = new HuffmanCompression("aabbccddd"); // 3*codeListBinaryValueTest, 4*b, 7*c
   Node root = compressor.getCompressionTree();
   System.out.println("READING FROM TREE");
   ArrayList<String> testListHolder = compressor.createCodeList(root, new StringBuilder());

   for (String n : compressor.createCodeList(root, new StringBuilder())) {
       System.out.println(n);
   }

   System.out.println("codeListBinaryValueTest");
   System.out.println("Index 0 = "+ testListHolder.get(0).substring(6));
   System.out.println("Index 3 = "+ testListHolder.get(3).substring(6));
   assertEquals(testListHolder.get(0).substring(6), "00");
   assertEquals(testListHolder.get(3).substring(6), "11");

}
```

NodeTest

Here the provided Node class tests are explained.

@test public void canReadWhatWasWritten()

This test checks whether a small Huffman tree can be written and read in pre-order.

First, this test sets the compressor variable’s text equal to “aba”.
It then creates an objectoutputstream with as destination the file “huffman-tree.bin” and writes the Node “tree”(along with its sub Nodes) to the “huffman-tree.bin” file.

Afterwards it reads the node that was written to the “huffman-tree.bin” file and sets a new node equal to its value. Finally it asserts whether the new node’s values are equal to that of the node that was written to the “huffman-tree.bin” file.

```
 compressor = new HuffmanCompression("aba");
   try (ObjectOutputStream output = new ObjectOutputStream(new
FileOutputStream(("huffman-tree.bin")))) {
       Node tree = new Node(new Node(1, 'b'), new Node(2, 'a'));
       tree.write(output);
   }
   Node tree = null;
   try (ObjectInputStream input = new ObjectInputStream(new FileInputStream(("huffman-tree.bin"))))
{
       tree = Node.read(input);
   }

   assertEquals(Character.valueOf('b'), tree.getLeft().getCharacter());
   assertEquals(Character.valueOf('a'), tree.getRight().getCharacter());
}
```

ExtendedNodeTest

@Test public void OutputStreamIsPreOrder() throws IOException, ClassNotFoundException ()

As the name states, this method checks whether the Node's write() method writes an OutputStream in pre-order, and the read()method can reconstruct the tree from this order. as required in the assignment.

This is done by constructing a Huffman Tree with characters which we know are going to end up at certain locations in the tree, and then writing them to an OutputStream using the write()method.

```
{
   compressor = new HuffmanCompression("abbcccccc");
   //should output: null null 1 a 2 b 6 c
   Node root = compressor.getCompressionTree();
   try (ObjectOutputStream output = new ObjectOutputStream(new
FileOutputStream(("huffman-tree.bin")))) {
       root.write(output);
   }
```

This OutputStream is then directly read again as input using the write() method, to recursively construct the Huffman tree in pre-order. The leftern nodes and Leaves should be written and read first. Fort this test the order should be: null(Root node) - null(Node) - Leaf1-a-Leaf2-b - Leaf6-c

```
   try (ObjectInputStream input = new ObjectInputStream(new FileInputStream(("huffman-tree.bin"))))
{
       assertNull(input.readObject()); //a
       assertNull(input.readObject()); //a
       assertEquals(1, (input.readObject())); //1
       assertEquals("a", (input.readObject()).toString()); //a
       assertEquals(2, (input.readObject())); //b
       assertEquals("b", (input.readObject()).toString()); //2
       assertEquals(6, (input.readObject())); //6
       assertEquals("c", (input.readObject()).toString()); //2
   }
}
```

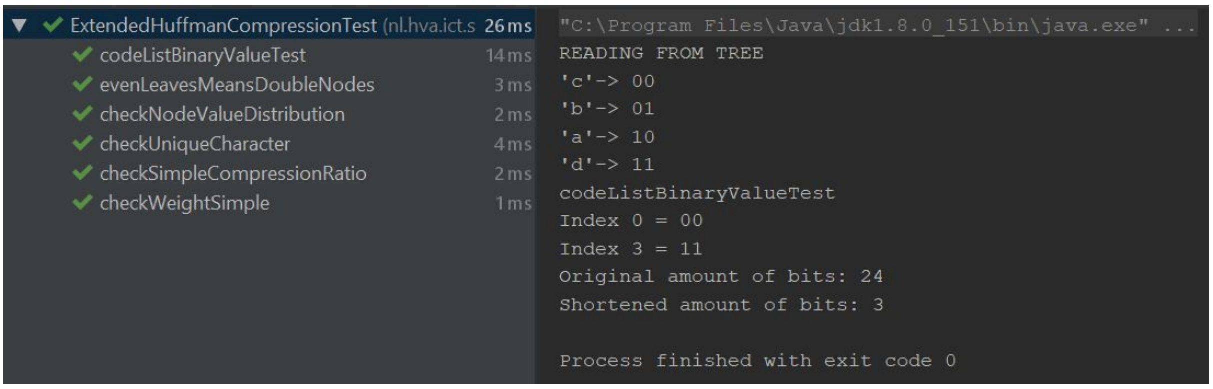
@Testpublic void canReadWhatWasWrittenBigTree() throws IOException, ClassNotFoundException

Almost the same as the ExtendedHuffmanTest “evenLeavesMeansDoubleNodes()” test, except it first writes the tree to a binary file and then reconstructs it before testing on it. This test is to make sure every values is correctly placed. Also the h value is included three times and must end up as the most rightern value to prove the **pre-order** writing and reading.

```
{
   compressor = new HuffmanCompression("aabbccddeeffgghhh"); // 2-2 2-2 2-2 2-3
   Node root = compressor.getCompressionTree();
   try (ObjectOutputStream output = new ObjectOutputStream(new
FileOutputStream(("huffman-tree.bin")))) {
       root.write(output);
   }
   try (ObjectInputStream input = new ObjectInputStream(new FileInputStream(("huffman-tree.bin"))))
{
       root = Node.read(input);
   }
   assertEquals(17, root.getWeight());
   assertEquals(2, root.getLeft().getLeft().getLeft().getWeight());
   assertEquals(2, root.getLeft().getRight().getRight().getWeight());
   assertEquals(3, root.getRight().getRight().getRight().getWeight());
   assertEquals("isNull", null, root.getRight().getRight().getCharacter());
}
```

Output

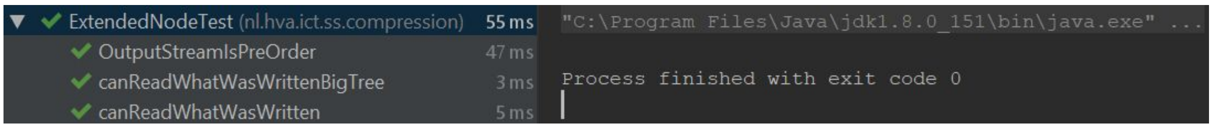
This chapter contains the output of all tests ran on the code we wrote.



```
▼ ✓ ExtendedHuffmanCompressionTest (nl.hva.ict.s 26 ms
    ✓ codeListBinaryValueTest                    14 ms
    ✓ evenLeavesMeansDoubleNodes                  3 ms
    ✓ checkNodeValueDistribution                  2 ms
    ✓ checkUniqueCharacter                        4 ms
    ✓ checkSimpleCompressionRatio                 2 ms
    ✓ checkWeightSimple                           1 ms
```

```
"C:\Program Files\Java\jdk1.8.0_151\bin\java.exe" ...
READING FROM TREE
'c'-> 00
'b'-> 01
'a'-> 10
'd'-> 11
codeListBinaryValueTest
Index 0 = 00
Index 3 = 11
Original amount of bits: 24
Shortened amount of bits: 3

Process finished with exit code 0
```



```
▼ ✓ ExtendedNodeTest (nl.hva.ict.ss.compression)  55 ms
    ✓ OutputStreamIsPreOrder                     47 ms
    ✓ canReadWhatWasWrittenBigTree                3 ms
    ✓ canReadWhatWasWritten                       5 ms
```

```
"C:\Program Files\Java\jdk1.8.0_151\bin\java.exe" ...

Process finished with exit code 0
```

Conclusion and Discussion

This chapter contains the conclusions and thoughts of our Huffman research.
Using the getCompressionRatio() method we can determine how much our Huffman algorithm can compress a given text. Using this method on a text with a large amount of recurring characters, results in a better compression ratio than when used on a text with an evenly distributed amount of characters.

This proofs that the Huffman compression method is more efficient when certain characters are used more often than others in a piece of text. By compressing the most occurring character (such as a whitespace) into a small amount of bytes, a lot of space can be saved. What has not been taken into account for this test is the amount of space the Huffman tree itself takes up.

Concluding: The more often certain characters are used than others, the more efficient the Huffman compression method is.

Discussion

When using this algorithm to compress just a small amount of text, it might end up taking up more space than the original text because of the space the Huffman table takes up. By saving the Huffman tree as characters, occurrences and null values, and then reconstructing it, even more space can possibly be saved. The difference in space between the original node and the written output would still need to be measured in order to calculate that effectiveness.