Project 3: Huffman Coding Trees

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Change Log

Like with Projects 1 and 2, this is a long enough assignment that we expect we will need to occasionally update this PDF as poor wording, or other issues become apparent. We will announce any update using the canvas announcement tools and in-lecture. If you think you have found any issues, please let us know immediately.

2 Essential Information

This is a three week assignment. By doing this assignment we hope you will:

- Demonstrate the ability to apply multiple topics from this course to novel programming problems by designing (without substantial guidelines) a custom purpose data structure
- Implement a tree-based data structure following provided guidelines.
- Implement an interesting algorithm for efficient data storage

2.1 Deadlines

This assignment is due Tuesday Apr 30th at 5pm. You will want to be thoughtful of your own time across your classes to make sure you don't plan on making a major push on this project at the last moment if doing so would also hurt your ability to study for other classes.

2.2 Individual Assignment

Unlike labs, where partner work is allowed, this project is an *individual assignment*. This means that you are expected to solve this problem independently relying only on course resources (zybook, lecture, office hours) for assistance. Inappropriate online resources, or collaboration at any level with another student will result in a grade of 0 on this assignment, even if appropriate attribution is given. Inappropriate online resources, or collaboration at any level with another student without attribution will be treated as an incident of academic dishonesty.

To be very clear, you can ask other students questions only about this document itself ("What is Daniel asking for on page 3?"). Questions such as "how would you approach function X" or "I'm stuck on part B can you give me a pointer" are considered inappropriate collaboration even if no specific code is exchanged. Coming up with general approaches to problems, and finding active ways to become unstuck are all parts of the programming process, and therefore part of the work of this assignment that we are asking you to do independently.

Further note – Huffman coding is a relatively well-known algorithm. Be cautious about what resources you look at to understand these more. Websites like Wikipedia (which do not have

specific final code) are generally OK, but you should not use any website containing specific code to better study these algorithms. If you're having trouble understanding Huffman coding, just in general, please reach out to course staff, we are happy to explain the concepts involved in this assignment if you feel you don't understand them fully.

3 Introduction and Overview

As high-level programming language users (C/C++/Java/Python) we're used to thinking of data in a very particular way. We think of different types of data, each of which can take their own values and have their own rules. An int, we say, is different from a char, or a double. At a low-level (considering the exact behavior of the computer) however, all of these pieces of data are the same – they are simple zeros and ones. This is something most students understand at a cultural level – everything inside the computer is made of patterns of true (one) and false (zero). Often, however, we do not think about the process by which these zero and one *encode* our different data types.

In this project were going to explore two specific ways to encode and decode text files (I.E. very large Strings) into specific patterns of zeros and ones. We will start with the encoding process: turning a String into a pattern of zeros and ones. This process can be done many different ways, so you will be in charge of applying the various ideas from across the semester to design a custom solution to this problem. Then we will explore the decoding process (turning a sequence of zeros and ones into a string) We will do this by looking at Huffman coding schemes, an interesting coding scheme that can sometimes substantially compress text files. This process is naturally suited to implementation with a Tree based data structure – which will be outlined below.

3.1 Concepts of Encoding

While these concepts may seen simple, or irrelevant, this section outlines some important back ground in encoding theory. An understanding of these concepts – to the level that you could per sonally encode and decode text accurately (if slowly) will be vital to your ability to reason about the code you need to write, understand the details that will be presented, and debug your code when things go wrong. Like other projects – make sure you are not skipping these "background" sections and you are testing your knowledge as you go.

3.1.1 Fixed Length Encoding

Encoding text files in a computer may be an every-day task, but it was originally a difficult challenge – one to which many different solutions exist. Normal approaches to encoding text files use what are known as "fixed-length" encoding. This means that each letter corresponds to a fixed number (say 8, or 16) of bits (zeros/ones) in the file. Each 8/16 bit pattern uniquely identifies one single letter. A selection of standard fix-length codes under the ASCII coding scheme can be seen in the table below:

letter	binary	letter	binary
a	01100001	A	01000001
b	01100010	B	01000010
c	01100011	C	01000011
d	01100100	D	01000100
e	01100101	E	01000101
f	01100110	F	01000110
g	01100111	G	01000111
h	01101000	H	01001000
Space	00100000	7	00110111

Fixed-length codes work particularly well in hardware, where the predictable and regular length of data can help make processing this data quick and efficient. Let's take a quick look at an few quick examples:

Given the string "Egg cab" we would encode this as follows:

E	g	g		c	a	b
01000101	01100111	01100111	00100000	01100011	01100001	01100010

So the encoded string would be
01000101011001110110011100100000011000110110000101100010 Going in the reverse

direction we could decode
0100100001100101011000010110010000100000011001110110000101100110011001100110 as follows:

01001000	01100101	01100001	01100100	00100000	01100111	01100001	01100110	01100110
H	e	a	d		g	a	f	f

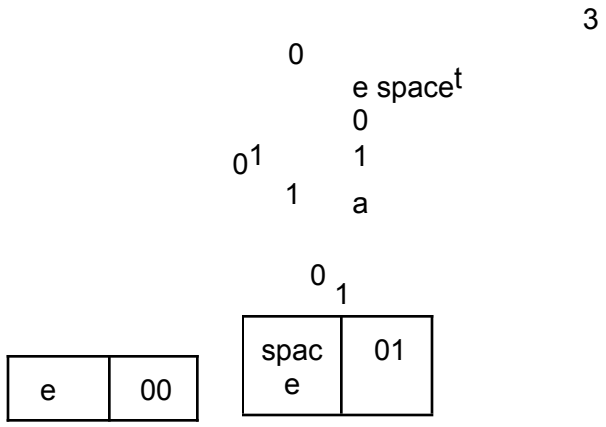
So decoded we would get “Head gaff”

3.1.2 Variable Length Encoding

Fixed-length codes are easy to work with, but are not always the most efficient. Conceptually, it doesn’t make sense that common letters like ‘e’ ‘t’, ‘a’, ‘i’, ‘n’, ‘o’, and ‘s’ take up the same amount of space as more obscure letters like ‘%‘ or ‘q’. Therefore, one solution that’s been explored in the past to compress text is to have a not-fixed-length coding scheme. In such a scheme different letters can encode to shorter or longer sequences of binary. If we make commonly used letters encode to shorter binary sequences (often forcing less common letters to use larger binary sequences) we can get substantially smaller binary encodings.

One of the most common schemes for creating variable length encoding is known as the Huffman coding scheme. A Huffman code scheme is one in which different letters have different length representations, and certain properties are met (which we will explain latter) The different code lengths make the encoding and decoding process more complicated, but also make the files more efficient on disk. The codes we will generate, for example, can reduce the size of an ebook by around 45%.

Before discussing specific details, an example will be useful. Below is a simplified example that only covers the letters ‘e’, ‘t’, ‘a’, ‘b’, ‘k’, and ‘ ‘ (space). A real Huffman code would need to cover all of the letters used in whatever text you wish to compress. This is, obviously, a simplified example (that’s not even all the normal letters, let-alone capitol letters, symbols, and other “letters” we need to represent a whole book) but it should be enough to explain a few core ideas.



t	10
a	111
k	1100

b	1101

0 1 k b

It's common to present Huffman codes in two ways. First as a "look up table" where you can quickly go from a letter to the binary pattern that encodes it, and secondly as a tree. The codebook/look up table makes it easy to encode text following the same process as fixed length codes For example to encode the name "kate" we would simply look up each letter in the codebook:

k	a	t	e
1100	111	10	00

Giving us "11001111000" as our final encoding.

Decoding with only the lookup table is *possible* but not efficient. With fixed length codes we could easily split a long sequence up by length and look up the letter for each 8-bit sub-sequence. With non-fixed huffman codes like this, this isn't possible, so an alternate process would be needed. This is where the tree perspective comes in handy. Decoding text simply becomes the process of letting the binary string guide us as we walk through the tree.

For example given the series "001111001111011000111" we can take each bit one at a time following the edges from the root of the tree based on the next bit. Starting from the root, the first two bits "00" bring us to the first letter 'e'. We then restart from the root. The next three bits 111, bring us to 'a'. We restart again, and 10 brings us to 't'. Following this process bit-by-bit let's us eventually retrieve the full message: "eat a tea" (not a sensible message, but it's what the code says).

Before going forward, as a personal exercise to make sure you personally know what is required here, and you are personally able to encode and decode in this scheme, I recommend encoding the string "take a tea" and decoding the binary sequence "1100111100001111100001111011101111110000". Feel free to compare answers for this exercise with friends, or on slack.

From this example we can see the most important property: Huffman codes are "prefix free". This means that no one code can be the prefix-of (same as the beginning of) a second code. If we had "a" encoded as 01, and "b" encoded as "0100", and "c" and "00", for example, we would not be prefix free, and wouldn't know what "010001" encodes – is it "aca" or "ba"?

Translated structurally, we see that this means that we should only have two types of nodes in our code tree – leaves (store a letter but no next nodes) and internal nodes (have both a zero and one for the next node, but no data) So long as our code tree has this property, we can rely on decoding to be quick and efficient.

(A note on terms: technically, what we have described is known as a variable-length prefix

code. 4

The term Huffman codes specifically refers to algorithms used to build these trees in an optimal way. We won't be requiring this specific algorithm in this assignment.)

3.2 Overview of Requirements

As the Huffman coding process has two core parts (encoding text into the Huffman code, and decoding text out of a Huffman code) our project will have two core parts as well:

- Part 1 will be to make the HuffmanCodeBook class, which can encode text into a Huffman code. This will represent the "code tables" or "code books" seen above – a quick method to translate from a letter to a specific binary sequence.
- Part 2 will be building the HuffmanNode and HuffmanCodeTree classes which allow us to represent the huffman code tree as seen above. Using the general algorithm above, we can efficiently decode text out of a Huffman code, and back into normal text.

While we will not write this program directly, combining these classes with provided testing code should give you all you need to encode and decode ebooks (so long as they only use English letters). To make this easier we have provided a series of files. Your FIRST task, will be to download and familiarize yourself with these classes and understand how to use them.

- BinarySequence – The Binary Sequence class represents a (possibly very large) sequence of binary values. Binary values can be either 1 (true) or 0 (false). This class will be used to represent series of binary values for Huffman codes, as well as larger series of binary values represented full encoded text. This class has helper functions to allow saving and loading binary sequences to a file, which you can use to write or read encoded text from a file.

  As you read this class be sure you know:

  - How to build one of these bit-by-bit
  - How to build one of these out of simpler binary sequences
  - An efficient pattern for looping over the binary sequence bit-by-bit.

  Take notes on each of these as you will need to use this class to perform these tasks later.

- FileIOAssistance – This class has three static methods. A method to read a standard text file as a String, a method to write a string into a standard text file, and a main method which tests these behaviors by making a specific file. The comments in this file can help you make sure you know where files will be read/written from in your project.
- ProvidedHuffmanCodeBook – this class has a single static method that creates a HuffmanCode Book object that has a complete codebook necessary to decode a selection of ebooks.
- StringBuilderDemo this class has a single main method which demonstrates the StringBuilder class and it's use. You will need to use this class to create strings in this project, so I figured an example that demonstrates common use, and has comments explaining why we

use it this way, would be beneficial.

For efficiency reasons you will be required to use StringBuilder make sure you un derstand how to translate common patterns for building strings letter-at-a-time into String Builder code based on this example. If, after review and editing this code, you still feel you don't know how to use StringBuilder let us know and we can walk you though it more.

4 Required Classes

AS A REQUIREMENT FOR ALL OF THE FOLLOWING You cannot use any of the pre built java collection classes. This includes ArrayList, LinkedList, TreeMap, TreeSet, and HashMap. The point of this project is to *build* datastructures, not to *use* datastructures.

You are also not allowed to **directly** use similar classes from lecture or past labs, although you can reference them for inspiration on how to design your own classes. So, for example, if you wanted to use the same *approach* as a past lab you can look at your code from the past lab, and even copy parts of the code into your new datastructures, but you would not be allowed to *use* past data structures in your new code.

To be very direct: Both HuffmanCodeBook and HuffmanCodeTree classes should *directly im plement their respective datastructures* they should not utilize an external implementation of these datastructures.

4.1 HuffmanCodeBook

The Huffman CodeBook class represents the "codebook" of the Huffman coding process, that is, it tells us, for each letter – what is the correct binary sequence. When encoding files, the Huffman CodeBook class is used to encode files – transform them from a series of letters, to a compact binary sequence.

HuffmanCodeBook class has the following required public methods/constructors:

- **public** HuffmanCodeBook() – a 0 argument constructor. When first created a HuffmanCodeBook object contains no letters/sequences.
- **public void** addSequence(**char** c, BinarySequence seq). This method should add a given charac ter/letter and binary sequence into the codeBook. This should be added in such a way that future calls to Contains for this character will return true, and future calls to getSequence with this character will return this sequence. You are not formally required to handle re-adding a given character to the codeBook – you are free to chose any behavior for this case.
- **public boolean** contains(**char** letter). this method should return true/false to indicate if the codebook contains a given letter. A letter is contained if and only if a previous call to addSequence has added this letter.
- **public boolean** containsAll(String letters). This function can be used to see if a given code Book can handle a given piece of text. It should return true if and only if every letter in the input string is contained in the codebook.
- **public** BinarySequence getSequence(**char** c) This method should get the binary sequence associ ated with the given letter. If addSequence was previously called with this letter as a parameter

the BinarySequence added with this letter should be returned. Otherwise (if addSequence has not been called with this letter) the special value **null** should be returned.

- **public** BinarySequence encode(String s) This function should encode the input string into a binary sequence. This process will involve combining, in order, the binary sequence associated with each letter in the string. You will not be tested in a case where encode is called on a string which has characters not contained in the codebook. You are free to chose how your code behaves in this case.

4.1.1 Additional requirements

We have two additional requirements for this project: Efficiency, and looping. These requirements do not directly map to specific required functions and form the *hard part* of your work here. Both of these requirements may involve building *extra* methods, or considering *multiple design alternatives*. You will be in charge of these **additional elements of class-level design**. We will cover sufficient ideas over the semester to reach 100% on these requirements, but not immediately. Therefore you may wish to start with a *sufficient* CodeBook, and then double-back later to optimize it with a more efficient design later in the semester.

efficiency You have 100% control over how the internal structure of this class works. That said, we will be looking for a well-chosen efficient way to organize the data in this class. Due to how the class will likely be used in practice (created once and then used to encode many pieces of text) we want to optimize the performance of the getSequence and contains method. We will want to do this without major sacrifices in terms of memory use.

Approximately 10 points will be allocated to this efficiency requirement.

- For 0 points, contains and getSequence run in time WORSE THAN $O(n)$ such as $O(n \log n)$ or $O(n^2)$ runtime. As an example, many solutions where you store everything in one big string are going to have worse-than $O(n)$ runtime due to the extra length of the binary sequences.
- For 5 points, contains and getSequence run in time $O(n)$ or an excessive amount of memory is used.
- For full points (10) contains and get sequence run in **strictly faster** than $O(n)$. The most common example of this (the expectation) is $O(\log n)$ time. Additionally you need to not use excessive amounts of memory (no 65,536 element arrays!)

As you consider this, remember that the add method is not covered here – some solutions might make the add method less efficient in order to make the contains and getSequence methods more efficient. This is acceptable and reasonable.

looping The assignment as written is not going to be possible with only the public methods that we've required. One of the constructors of the HuffmanCodeTree class will need a way to efficiently loop over all characters in the codeBook class. You will note that the **required** public methods do not have any explicit way to do this. This is because **we want you to design public methods to allow this behavior.** You will be evaluated on both the *design* and *implementation* of these extra methods. As this can be addressed MANY different ways, our evaluation criteria will be a bit more vague than normal – as they must focus on what the methods you decide on

do or do not allow other classes to do.

- The methods you add must make it possible for another class to loop over all characters stored in the CodeBook without repetition.
- No specific order is required for the looping order created by the methods you provide.
- The methods you add must be efficient. As a practical measure of this it should be possible (from a different class) to write a function to print all of the characters in your CodeBook, and such a function should run in time $O(n)$ where n is the number of characters.

- The methods you add should be easy to use. It should not require particularly complicated code to correctly use the method or methods you add
- The way you solve this problem must not breach the principals of abstraction and encapsula tion. That is to say, if the functions you add could be used to modify the private data of the CodeBook you will lose points. (So you can't do this by just making the private data public – your public methods must allow looping over the contained chars without allowing them to be modified.)

To be VERY CLEAR we are not requiring that you have a loopy-function built-into the Code Book class. Nor are we actually asking you to build a function that prints all the characters in a codeBook. We are requiring you add a method or multiple public methods to the CodeBook class that allow other programmers access to the information the would need to loop over the data stored in the CodeBook efficiently.

4.1.2 On using helper-classes

Some solutions to aspects of this problem rely on "helper classes" Any "helper classes" should be included as inner classes of the CodeBook Class. If you need a "helper class" and need assistance making it an inner class, reach out to course staff.

4.2 HuffmanNode

The Huffman Node class ultimately serves as a component of the HuffmanCodeTree class. It's structure is quite typical of a binary tree node, like the ones we will see in lecture. The core design feature of these classes is that they have two "next node" variables. In class we will see these labeled "left" and "right" child variables, but to represent a huffman code tree, we will want to label our "next node" variables as one and zero. Other than that change there are not many tricks to this class.

Your class should have three private variables:

- Two HuffmanNode variables storing the child-nodes of this node, or null (should this node be a leaf).
- a Character – representing the data stored at this node (or null if this node is not a leaf / not storing data) **NOTE** I'm recommending class Character instead of primitive type **char** here – a Character variable can store null to indicate "no data stored here", which a char cannot.

Your class should have the following public properties:

- **public** HuffmanNode(HuffmanNode zero, HuffmanNode one) a constructor that makes a non-leaf node by providing it's two child nodes. The data should be set to null.
- **public** HuffmanNode(**char** data) a constructor that makes a leaf node, specifying the data. The left- and right- child nodes should be set to null.
- As normal, you can add other methods/constructors as you see fit.

- getter and setter methods for the three private variables: **public** HuffmanNode getZero(), **public void** setZero(HuffmanNode zero), **public** HuffmanNode getOne(), **public void** setOne(HuffmanNode one), **public** Character getData(), **public void** setData(**char** data)
- **public boolean** isLeaf() Formally, a node is a leaf if it has no left- or right- children. Given the structure of a Huffman code tree, you could also base this on the data property of the tree, as a Huffman code tree has data on every leaf node (and does not have data on non-leaf nodes.)
- **public boolean** isValidNode() This function should check if this node is "valid" for a huffman code tree. A node is valid if it is either a leaf node (data is not null, and both children variables are null) or an internal node (data is null, both children are not null).
- **public boolean** isValidTree() This function should check if this node **AND ALL DESCEN DANT NODES** are "valid" for a Huffman coding tree. A Huffman code tree should only have two types of nodes: leaf nodes (data is not null, one and zero child node variables are null) and internal nodes (data is null, both one and zero leaf nodes are not null). This func tion should return true if the current node has one of these two valid forms AND ALSO each descendant node (any node you can reach through the one and zero references) has one of the two valid forms. If this node OR ANY DESCENDANT doesn't match this requirement (has data and children, or no data, but only one child) then the function should return false.

  HINT this function is quite easy to define recursively and *very painful* to define without recursion. You are note *required* to use recursion here – but you will find it *quite difficult* to implement correctly without recursion.

4.3 HuffmanCodeTree

The HuffmanCodeTree class uses the node class build and maintain a binary tree that represents a collection of Huffman codes for various letters. While both the HuffmanCodeBook and the HuffmanCodeTree represent the same codes (pairs of letters and binary sequences) the binary code tree is focused on decoding – that is to say, the HuffmanCodeTree is designed to be efficient in computing a char from a binary sequence – while your HuffmanCodeBook should be optimized to compute a binary sequence for a given char. If implemented correctly, the HuffmanCodeBook's main methods should run in time $O(log(n))$.

Requirements:

- You should have a single private variables root – which is of type HuffmanNode – this represents the root node of the Huffman code tree.

- **public** HuffmanCodeTree(HuffmanNode root) a constructor that creates a Huffman code tree using a provided Node as root.

- **public** HuffmanCodeTree(HuffmanCodeBook codebook) – a constructor which should create a Huff man code tree based on the data stored in a Huffman code book. This **should not** directly store the codebook object. Instead, this should create a new root node (make it temporarily an invalid node – with null values for all three private variables) Then it should simply need to loop over the chars in the Huffman code book, get the related sequences and repeatedly call the put method to update the tree with each code one-by-one.

  (This constructor will only be possible if you have a good solution to the looping requirement in the HuffmanCodeBook class. This will not be possible using only the explicitly required public methods)

- **public boolean** isValid() This should check if the tree formed by the root node and it's descen dants is a valid Huffman code tree. The definition for "valid Huffman code tree" is provided above. (**hint** – this method should be one line long.)

- **public void** put(BinarySequence seq, **char** letter) This method should modify the binary tree structure so that the node "addressed" by the binary sequence stores the given char. For example, if the binary sequence is "010" and the char is 'c' then at the end of this method root.getZero().getOne().getZero().getData() should be 'c'. The algorithm for this is straightfor ward, for each boolean in the BinarySequence, follow the appropriate child (I.E. node = node.getZero() or node = node.getOne()). If you ever encounter a null node as you travel, fill that node in with a new empty node (null zero, one, and data) (make sure you add this node to the tree, just creating the node and assigning it to a variable is not good enough – you will need to call setZero or setOne on another node to make it part of the tree.) once you have arrived-at / created the appropriate node, simply store the letter.

  Note – this method is not expected to check if it's made a valid tree. You would commonly use this method to build up a tree one letter at a time, after which you can check if the tree is still valid before proceeding.

- **public** String decode(BinarySequence s) This method should decode a BinarySequence into a string. If this is called you can assume the tree is currently valid, and that the binary sequence is of a correct length (I.E. it will end with the last bit in a code for a letter, not in the middle of the tree) The algorithm for this is quite straightforward:

  1. create a variable "node" and have it store the root node of the tree
  2. for each boolean in the sequence:
  3. if the boolean is true/1, update node = node.getOne()
  4. else if the boolean is false/0 update node = node.getZero()
  5. if you ever arrive at a leaf, add the data from that leaf to your output, and reset node to the root.

A VERY IMPORTANT NOTE for this method to be efficient, you will need to use the StringBuilder class. A demo of this class is provided with the code for this project. As shown in this demo, a StringBuilder is MUCH more efficient for algorithms where you repeatedly add one letter to the end of a string. Normally, this isn't a big deal, but we will be decoding strings with millions of letters, and the efficiency speedup from StringBuilder will be required to keep the code fast.

4.4 README Questions

You are required to prepare a README.txt file. This should be in standard text format (not encoded by your code, or in a document format such as pdf, Microsoft word, or rich-text-format.) (If we can't read it on gradescope you won't get credit for it. I recommend double-checking how gradescope shows your submission) Your file must answer the following questions:

1. What is your name
2. Any other notes for the grader
3. What is the runtime you're claiming for your CodeBook methods (contains and getSequence)? (use *n* = the number of characters in the codebook.)
4. What is the runtime of the HuffmanCodeTree.decode method, use *b* =the number of bits in the binarySequence. (I.E. your answer should not include *n* at all.)

5 Deliverables, Testing, and grading

5.1 Deliverables

Your project will be submitted online through gradescope. You should only need to submit the following files:

- HuffmanNode.java
- HuffmanCodeTree.java
- HuffmanCodeBook.java
- README.txt

If you NEED extra classes, (perhaps a node class for a linked codeBook solution) you will need to make them inner classes. Contact course staff for instructions on this, but the basic steps: 1) change your class from public class NAME to public static class NAME 2) copy the class inside your HuffmanCodeBook class (inside the {}).

No other files will be considered for automatic testing.

5.2 Testing

If you fail the autograded tests but pass on your own computer please contact us We can help debug any cause of issues like this.

5.3 Grading

The general grading rubric will include the following:

- Automatic tests
- Code style
- README
- HuffmanCodeBook (general data structure approach and runtime)
- HuffmanCodeBook
- HuffmanNode

- HuffmanCodeTree

Like usual, the points for automatic grading will not allow any partial credit, but the points of manual review will.