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.

- (November 26th) Version 1.0
- (November 27th) Version 1.1 added "reasonable memory" requirement that forbids using arrays hundreds of times bigger than needed for HuffmanCodebook. It's a very clever solution, but doesn't really demonstrate the mastery we're looking for here (and also, it's just really wasteful)
- (December 8th) Version 1.2 added clarification to the readme requirement questing 4 (we're only looking for runtime of the contains/getSequence methods)

2 Essential Information

This is a two-and-a-half 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 has two deadlines:

- "Check in" deadline (Monday December 6 at 6pm) This deadline requires completing a first-draft of HuffmanCodeBook class. The efficiency of your code will not be judged at this deadline, only it's ability to meet functional requirements. (See the grading section for more information)
- Final deadline (Wednesday December 15 at 6pm – last day of class) This deadline requires completing the entire assignment.

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.

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 a specific way to encode and decode text files (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. 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.

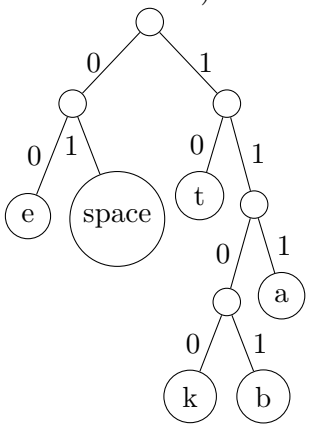
Encoding text files in a computer may be an every-day task, but it was originally a difficult challenge – one to which many different solutions exist. Normal approaches to encoding text files use what are known as "fixed-length" encoding. This means that each letter corresponds to a fixed number (say 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	01100010
c	01100011	C	01100011
d	01100100	D	01100100
e	01100101	E	01100101
f	01100110	F	01100110
g	01100111	G	01100111
h	01101000	H	01101000
Space	00100000	7	00110111

Fix-length codes like this are GREAT because they are easy for computer hardware to support, but they are often inefficient in terms of storage size. One way to make files smaller is to recognize that different letters are used with different frequencies. For example, in English text letters like 'e' 't', 'a', 'i', 'n', 'o', and 's' are used FAR more frequently than more obscure letters like 'z', 'j', 'x', and 'q'. As such, if we were to use a not-fixed-length encoding scheme, where 'e', 't', 'a', etc. have shorter codes and 'z', 'j', 'x', 'q' etc. have longer codes, we can save substantial storage space.

This second idea known as variable-length codes leads to the idea of a 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.



e	00
space	01
t	10
a	111
k	1100
b	1101

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. The tree structure is useful when decoding, for example given the series "00111100111011000111" 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. The term Huffman codes specifically refers to algorithms used to build these trees in an optimal way For more information on this I recommend doing the extra credit.)

3.1 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.
- `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 HuffmanCodeBook 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.

4 Required Classes

AS A REQUIREMENT FOR ALL OF THE FOLLOWING You cannot use any of the prebuilt java collection classes. This includes ArrayList, LinkedList, TreeMap, TreeSet, and HashMap. 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.

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 character/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 codeBook 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 associated 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. Notice, we do not require ANY specific internal datastructure here – You are in charge of designing how to structure this data to achieve the goals set out in the above methods. We have implemented datastructures in lecture and lab (at the time of assignment) that should be sufficient for these requirements. To reach 100% credit, however, you will either need to add ideas from elsewhere in class, or topics we will be covering at the very end of the semester. My recommendation here, is to implement a *sufficient* data structure, then double-back and optimize it as a secondary task.

looping One of the constructors of the HuffmanCodeTree class will need a way to efficiently loop over the characters in your codeBook class. We have NOT included methods designed to allow this. There are several ways to do this, ranging from the simple (list-style indexed methods) to the elegant (implementing an iterator). As the best, and most efficient, solution will vary class-by-class, we will not mandate any specific functions here. However, you are **REQUIRED** to provide some public method or methods that will allow other classes to loop over the set of characters contained in the codebook. It must be possible to use this loop to print all characters in the codebook without repetition in time $O(n)$ where n is the number of characters in the codeBook.

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 function. The specific efficiency focus of this class should be the `getSequence` method, as that will be used frequently during the encoding process. We will also look for efficiency for the contains method (as this should be at least as fast as the getSequence method)

Roughly 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.
- For 5 points, contains and getSequence run in time $O(n)$
- 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.
- For full points (10) and bragging rights – professor Kluver believes this is technically possible in runtime $O(1)$, but it's complex enough that Kluver didn't implement the reference solution to do it.

(added in version 1.1) Additional to these runtime efficiency goals we also require memory efficiency. We will not be evaluating this particularly rigidly, all of the approaches we've seen or will see in lecture (array doubling, linked lists, binary trees) are reasonable here. Solutions that use substantially more memory than needed however, will not be accepted. As an example of what is expressly not allowed here: An $O(1)$ solution can be made easily with a size 65,536 BinarySequence array. While fast, such a solution is quite wasteful in terms of memory efficiency and therefore would only be worth 5 points.

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 isValid()` This function should check if this node **AND ALL DESCENDANT 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 function 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 Huffman 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.
- `public boolean isValid()` This should check if the tree formed by the root node and it's descendants 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 straightforward, 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 common 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. Did you attempt the extra credit, and if so, where should we look for it in grading?
3. Any other notes for the grader
4. What is the runtime you're claiming for your CodeBook methods (contains and getSequence)? (use n = the number of characters in the codebook.)
5. 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 Extra Credit

The algorithm for creating optimal Huffman code trees is a challenging and interesting task in it's own right. For up to 5 points extra credit, independently research and implement this algorithm. The class staff can *help* you with this research/design, but we will not *tell you* how to do it. Part of the point of this extra credit is the actual task of independently researching an algorithm, understanding it, and designing code to implement it yourself.

Hints:

- You can use built-in java data types for this part of the project.
- The algorithm should start by reading 1 or more text files to decide what letters it needs to encode
- You can use the built-in java HashMap class to count occurrences of letters.

- You may want to modify HuffmanNode to store a "count" or "weight" on it and be comparable by that weight. If you do this, you should be able to use a PriorityQueue (built in java class) to get the 2 lowest weight nodes at each step.

6 Deliverables, Testing, and grading

6.1 First Deadline

For the Monday December 6 at 6pm "Check in" deadline, you are expected to have a complete HuffmanCodeBook class.

This checkin will be worth 15 points.

- 5 points from an autograder. This will directly map to the behavioral tests in the HuffmanCodeBookTester class provided on canvas.
- 2 points to the contains methods
- 2 points to the get and add methods
- 2 points to the encode method
- 2 points to manual evaluation of your strategy for looping over the chars in the datastructure
- 2 points to manual evaluation of the overall design of the class. **NOTE** at this point, we are looking only for *functional* code – and will not be evaluating the runtime of your class methods.

These manual reviews will be high-level, graded on the following rubric:

- 0 points (no attempt)
- 1 point (issue found, or incomplete)
- 2 points (no issue found, or easy-to-fix issue found.)

6.2 Second Deadline

For the second (and final) submission (Wednesday December 15 at 6pm) you are expected to implement all 3 classes listed above, and prepare a README file to assist grading.

The second deadline will be worth 85 points:

- CodeBook 30 points
  - 10 points on the design of the class as a whole – for full credit here your `getSequence(char)` and `contains(char)` methods should run in time $O(logn)$ or better (where n is the number of chars in the codebook) Running in time $O(n)$ will be worth 5 points, and running in time worse than $O(n)$ will result in 0 points here.
  - 5 points on the design and implementation of code to allow looping over the chars in your codebook. You should be able to print each char in the codebook in time $O(n)$.

  - 6 points on autograder (there will be one additional test in phase 2 – nothing secret, just checking that encode and decode can act as reverses)
  - 9 points (roughly 3 each) on the encode methods, get and add methods, and two contains methods.

- HuffmanNode 10 points
  - 5 points from autograder
  - 3 points for the `isValid` method
  - 2 points for all other aspects of this class

- HuffmanCodeTree 30 points. (Tentatively:)
  - 15 points from autograder tests.
  - 5 points for the put method
  - 5 points for the decode method
  - 5 points for all other methods

- Code Style 10 points
  - Your code should be relatively readable (any particularly confusing lines of code should be explained by comment, variable names should explain the intent of the code, etc.)
  - Each class should have a high-level javadoc explaining it's purpose
  - Each method longer than 1 line should have a javadoc explaining it's purpose
  - Good code style choices should be made around public/private. Instance variables should not be public without very good reason.
  - Good indentation should be practiced.
  - A consistent variable naming scheme should be used (you can choose camelCase or snake_case, but you should keep whatever you choose for the whole project.)

- README file 5 points
  - 1 point for simply submitting a file with at least your name in it
  - 2 point for correctly estimating the runtime of your codeBook methods
  - 2 points for correctly estimating the runtime of the decode method