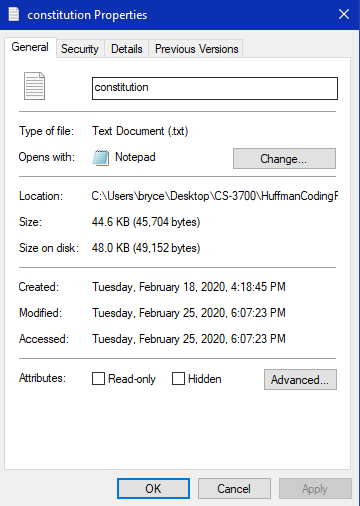
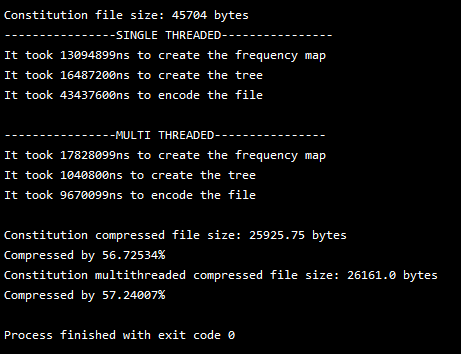
Bryce Callender

<https://github.com/BryceCallender/CS-3700/tree/master/HuffmanCodingProject>

1. 

PROGRAM OUTPUT



1. Constitution compressed file size: 25925.75 bytes

Compressed by 56.72534%

Constitution multithreaded compressed file size: 26161.0 bytes

Compressed by 57.24007%

1. Decompression of the human encoding can be multithreaded, but you would have to go through a lot to make it so. It is inherently single threaded as the prefix for each character is a unique sequence so you will not really be able to know when a character end or where it starts based on the length of the sequence and until you traverse the tree.

CODE

Main.java

import java.io.File**;**import java.io.IOException**;**public class Main {  
 public static void main(String[] args) throws IOException {  
  
 File constitutionFile = new File("constitution.txt")**;** System.*out*.println("Constitution file size: " + constitutionFile.length() + " bytes")**;** System.*out*.println("----------------SINGLE THREADED----------------")**;** HuffmanEncoding huffmanEncoding = new HuffmanEncoding(constitutionFile)**;** huffmanEncoding.createHuffmanTree()**;** File newFile = huffmanEncoding.encodeFile()**;** huffmanEncoding.decodeFile()**;** System.*out*.println("----------------MULTI THREADED----------------")**;** ParallelHuffmanEncoding parallelHuffmanEncoding = new ParallelHuffmanEncoding(constitutionFile)**;** parallelHuffmanEncoding.createHuffmanTree()**;** File parallelFile = parallelHuffmanEncoding.encodeFile()**;** parallelHuffmanEncoding.decodeFile()**;** System.*out*.println("Constitution compressed file size: " + (newFile.length()/**8.0f**) + " bytes")**;** System.*out*.println("Compressed by " + ((newFile.length() / **8.0f**)/constitutionFile.length() \* **100**) + "%")**;** System.*out*.println("Constitution multithreaded compressed file size: " + (parallelFile.length()/**8.0f**) + " bytes")**;** System.*out*.println("Compressed by " + ((parallelFile.length() / **8.0f**)/constitutionFile.length() \* **100**) + "%")**;** }  
}

HuffmanEncoding.java

import java.io.\***;**import java.util.Comparator**;**import java.util.HashMap**;**import java.util.Map**;**import java.util.PriorityQueue**;**class HuffmanNode {  
 int frequency**;** char character**;** HuffmanNode leftChild**;** HuffmanNode rightChild**;**}  
  
class HuffmanComparator implements Comparator<HuffmanNode> {  
 @Override  
 public int compare(HuffmanNode huffmanNode1**,** HuffmanNode huffmanNode2) {  
 return Integer.*compare*(huffmanNode1.frequency**,** huffmanNode2.frequency)**;** }  
}  
  
public class HuffmanEncoding {  
 private File file**;** private Map<Character**,** Integer> frequencyMap**;** private Map<Character**,** String> binaryRepresentations**;** PriorityQueue<HuffmanNode> priorityQueue**;** HuffmanEncoding(File file) {  
 this.file = file**;** frequencyMap = new HashMap<>()**;** priorityQueue = new PriorityQueue<>(**1,** new HuffmanComparator())**;** binaryRepresentations = new HashMap<>()**;** }  
  
 public void createHuffmanTree() throws IOException {  
 try {  
 long start = System.*nanoTime*()**;** BufferedReader bufferedReader = new BufferedReader(new FileReader(file))**;** int c**;** while((c = bufferedReader.read()) != -**1**) {  
 Character character = (char) c**;** if(!frequencyMap.containsKey(character)) {  
 frequencyMap.put(character**, 1**)**;** }else {  
 frequencyMap.put(character**,** frequencyMap.get(character) + **1**)**;** }  
 }  
 long end = System.*nanoTime*()**;** System.*out*.println("It took " + (end-start) + "ns to create the frequency map")**;**// for (Map.Entry<Character,Integer> entry: frequencyMap.entrySet()) {  
// System.out.println(entry.getKey() + ": " + entry.getValue());  
// }  
  
 start = System.*nanoTime*()**;** for (Map.Entry<Character**,**Integer> entry: frequencyMap.entrySet()) {  
 HuffmanNode huffmanNode = new HuffmanNode()**;** huffmanNode.character = entry.getKey()**;** huffmanNode.frequency = entry.getValue()**;** huffmanNode.leftChild = null**;** huffmanNode.rightChild = null**;** priorityQueue.add(huffmanNode)**;** }  
  
 while(priorityQueue.size() > **1**) {  
 HuffmanNode node1 = priorityQueue.poll()**;** HuffmanNode node2 = priorityQueue.poll()**;** HuffmanNode newNode = new HuffmanNode()**;** newNode.frequency = node1.frequency + node2.frequency**;** newNode.character = '\_'**;** //Means that this node just holds the 2 nodes and is not an actual value node  
  
 newNode.leftChild = node1**;** newNode.rightChild = node2**;** priorityQueue.add(newNode)**;** }  
  
 generateBinaryCodes(priorityQueue.peek()**,** "")**;** end = System.*nanoTime*()**;** System.*out*.println("It took " + (end-start) + "ns to create the tree")**;**// System.out.println("Encoding Key Output");  
// for (Map.Entry<Character, String> entry: binaryRepresentations.entrySet()) {  
// System.out.println(entry.getKey() + ": " + entry.getValue());  
// }  
  
 } catch (FileNotFoundException e) {  
 e.printStackTrace()**;** e.printStackTrace()**;** }  
 }  
  
 public File encodeFile() {  
 long start = System.*nanoTime*()**;** File outputFile = new File("compressed\_constitution.txt")**;** StringBuilder string = new StringBuilder()**;** try {  
 FileWriter fileWriter = new FileWriter(outputFile)**;** BufferedReader bufferedReader = new BufferedReader(new FileReader(file))**;** int c**;** while((c = bufferedReader.read()) != -**1**) {  
 Character character = (char) c**;** fileWriter.write(binaryRepresentations.get(character))**;** string.append(binaryRepresentations.get(character))**;** }  
  
 fileWriter.close()**;** } catch (IOException e) {  
 e.printStackTrace()**;** }  
 long end = System.*nanoTime*()**;** System.*out*.println("It took " + (end-start) + "ns to encode the file")**;** return outputFile**;** }  
  
 public void decodeFile() {  
 File fileToRead = new File("compressed\_constitution.txt")**;** try {  
 BufferedReader bufferedReader = new BufferedReader(new FileReader(fileToRead))**;** HuffmanNode current = priorityQueue.peek()**;** int c**;** while((c = bufferedReader.read()) != -**1**) {  
 Character character = (char) c**;** if(character.equals('0')) {  
 current = current.leftChild**;** }else {  
 current = current.rightChild**;** }  
  
 if(current.leftChild == null && current.rightChild == null) {  
 //System.out.print(current.character);  
 current = priorityQueue.peek()**;** }  
 }  
 System.*out*.println()**;** } catch (IOException e) {  
 e.printStackTrace()**;** }  
 }  
  
 private void generateBinaryCodes(HuffmanNode root**,** String binary) {  
 if(root == null) {  
 return**;** }  
  
 if(root.character != '\_') {  
 binaryRepresentations.put(root.character**,** binary)**;** }  
 generateBinaryCodes(root.leftChild**,** binary + "0")**;** generateBinaryCodes(root.rightChild**,** binary + "1")**;** }  
  
}

ParallelHuffman.java

import java.io.\***;**import java.nio.ByteBuffer**;**import java.nio.CharBuffer**;**import java.nio.channels.AsynchronousFileChannel**;**import java.nio.charset.Charset**;**import java.nio.file.\***;**import java.util.\***;**import java.util.concurrent.\***;**public class ParallelHuffmanEncoding {  
 private File file**;**

This code below takes advantage of the AsyncFileChannel class to asynchronously read a file and then once it is done reading, we can use the futures to get the buffer data and store it in a string. This will take a bit of time, thus the slowdown compared to single thread, but it allows better performance for the threads in the encoding stage. So, it’s a tradeoff for more performance later.

private Map<Character**,** Integer> frequencyMap**;** private Map<Character**,** String> binaryRepresentations**;** private String fileData**;** private ExecutorService threadPool**;** private int coreCount**;** private PriorityQueue<HuffmanNode> priorityQueue**;** ParallelHuffmanEncoding(File file) {  
 this.file = file**;** frequencyMap = new ConcurrentHashMap<>()**;** priorityQueue = new PriorityQueue<>(**1,** new HuffmanComparator())**;** binaryRepresentations = new HashMap<>()**;** coreCount = Runtime.*getRuntime*().availableProcessors()**;** threadPool = Executors.*newFixedThreadPool*(coreCount)**;** }  
  
 public void createHuffmanTree() throws IOException {  
 try {  
 long start = System.*nanoTime*()**;** Path path = Paths.*get*("constitution.txt")**;** StringBuilder fileContents = new StringBuilder()**;** AsynchronousFileChannel asynchronousFileChannel = AsynchronousFileChannel.*open*(path**,** EnumSet.*of*(StandardOpenOption.*READ*)**,** threadPool)**;** int bufferSize = (int)path.toFile().length() / coreCount + **50;** // some padding from bad division offsets  
  
 List<ByteBuffer> buffers = new ArrayList<>()**;** List<Future<?>> futures = new ArrayList<>()**;** long position = **0;** for(int i = **0;** i < coreCount**;** i++) {  
 ByteBuffer buffer = ByteBuffer.*allocate*(bufferSize)**;** futures.add(asynchronousFileChannel.read(buffer**,** position))**;** position += bufferSize**;** buffers.add(buffer)**;** }  
  
 for(Future<?> future: futures) {  
 future.get()**;** }  
  
 for (ByteBuffer byteBuffer: buffers) {  
 byteBuffer.position(**0**)**;** CharBuffer charBuffer = Charset.*defaultCharset*().decode(byteBuffer)**;** fileContents.append(charBuffer)**;** for (char character: charBuffer.array()) {  
 if(!frequencyMap.containsKey(character)) {  
 frequencyMap.put(character**, 1**)**;** }else {  
 frequencyMap.put(character**,** frequencyMap.get(character) + **1**)**;** }  
 }  
 }  
  
 long end = System.*nanoTime*()**;** System.*out*.println("It took " + (end-start) + "ns to create the frequency map")**;** fileData = fileContents.toString()**;** start = System.*nanoTime*()**;** for (Map.Entry<Character**,**Integer> entry: frequencyMap.entrySet()) {  
 HuffmanNode huffmanNode = new HuffmanNode()**;** huffmanNode.character = entry.getKey()**;** huffmanNode.frequency = entry.getValue()**;** huffmanNode.leftChild = null**;** huffmanNode.rightChild = null**;** priorityQueue.add(huffmanNode)**;** }  
  
 while(priorityQueue.size() > **1**) {  
 HuffmanNode node1 = priorityQueue.poll()**;** HuffmanNode node2 = priorityQueue.poll()**;** HuffmanNode newNode = new HuffmanNode()**;** newNode.frequency = node1.frequency + node2.frequency**;** newNode.character = '\_'**;** //Means that this node just holds the 2 nodes and is not an actual value node  
  
 newNode.leftChild = node1**;** newNode.rightChild = node2**;** priorityQueue.add(newNode)**;** }  
 generateBinaryCodes(priorityQueue.peek()**,** "")**;** end = System.*nanoTime*()**;** System.*out*.println("It took " + (end-start) + "ns to create the tree")**;**// System.out.println("Encoding Key Output");  
// for (Map.Entry<Character, String> entry: binaryRepresentations.entrySet()) {  
// System.out.println(entry.getKey() + ": " + entry.getValue());  
// }  
  
 } catch (FileNotFoundException | InterruptedException | ExecutionException e) {  
 e.printStackTrace()**;** }  
 }

This code below is sped up by using an executor service that is submitted a callable (code at the end) that will read the file and use memorization to quickly write to the file with little to no computation.

public File encodeFile() {  
 long start = System.*nanoTime*()**;** File outputFile = new File("constitution\_multithread.txt")**;** List<Future<String>> futures = new ArrayList<>()**;** try {  
 FileWriter fileWriter = new FileWriter(outputFile)**;** for(int i = **0;** i < coreCount**;** i++) {  
 int startLine = i \* Math.*floorDiv*(fileData.length()**,** coreCount)**;** int endLine = startLine + Math.*floorDiv*(fileData.length()**,** coreCount)**;** //System.out.printf("Start: %d End: %d%n", startLine, endLine);  
  
 futures.add(threadPool.submit(new CallableEncoder(binaryRepresentations**,** fileData**,** startLine**,** endLine)))**;** }  
  
 for (Future<String> future: futures) {  
 try {  
 fileWriter.write(future.get())**;** } catch (InterruptedException | ExecutionException e) {  
 e.printStackTrace()**;** }  
 }  
  
 fileWriter.close()**;** } catch (IOException e) {  
 e.printStackTrace()**;** }  
 threadPool.shutdown()**;** long end = System.*nanoTime*()**;** System.*out*.println("It took " + (end-start) + "ns to encode the file")**;** return outputFile**;** }  
  
 private void generateBinaryCodes(HuffmanNode root**,** String binary) {  
 if(root == null) {  
 return**;** }  
  
 if(root.character != '\_') {  
 binaryRepresentations.put(root.character**,** binary)**;** }  
 generateBinaryCodes(root.leftChild**,** binary + "0")**;** generateBinaryCodes(root.rightChild**,** binary + "1")**;** }  
  
 public void decodeFile() {  
 File fileToRead = new File("constitution\_multithread.txt")**;** try {  
 BufferedReader bufferedReader = new BufferedReader(new FileReader(fileToRead))**;** HuffmanNode current = priorityQueue.peek()**;** int c**;** while((c = bufferedReader.read()) != -**1**) {  
 Character character = (char) c**;** if(character.equals('0')) {  
 current = current.leftChild**;** }else {  
 current = current.rightChild**;** }  
  
 if(current.leftChild == null && current.rightChild == null) {  
 //System.out.print(current.character);  
 current = priorityQueue.peek()**;** }  
 }  
 System.*out*.println()**;** } catch (IOException e) {  
 e.printStackTrace()**;** }  
 }  
  
}

CallableEncoder.java

import java.util.Map**;**import java.util.concurrent.Callable**;**public class CallableEncoder implements Callable<String> {  
 private Map<Character**,** String> binaryRepresentations**;** private String fileData**;** private int start**;** private int end**;** CallableEncoder(Map<Character**,** String> binaryRepresentations**,** String fileData**,** int startLine**,** int endLine) {  
 this.binaryRepresentations = binaryRepresentations**;** this.fileData = fileData**;** this.start = startLine**;** this.end = endLine**;** }  
  
 @Override  
 public String call() throws Exception {  
 StringBuilder data = new StringBuilder()**;** for(int i = start**;** i < end**;** i++) {  
 char character = fileData.charAt(i)**;** data.append(binaryRepresentations.get(character))**;** }  
 return data.toString()**;** }  
}