EECS 233 Introduction to Algorithms and Data Structures
Fall 2014, Programming Assignment 2 (“P2”)

Drafts: New for this assignment is that you can submit drafts (via blackboard) to get feedback from the testing suite once a day until the final due date. Note that your code needs to implement the fundamental functionality and have skeleton methods in order to be checked against the full test suite. Please don’t submit until your code passes the fundamental and skeleton JUnit tests supplied with the assignment. The TA will contact you if there are issues with your submission that prevent us from the running the test suite.

Final Due: Sat Oct 18 by midnight.

Total Points: 111 points

In this assignment, you will write a compression program using the Huffman coding algorithm which uses a combined linked list and binary tree data structure. When you are finished your code will be able to read any file and compress it to a binary file. This will work best (i.e. compress the most) for text files with an uneven character frequency, e.g. text files. The `BinaryWriter` class to write binary codes to files is provided on blackboard with the assignment, along with the JUnit tests for fundamental functionality and skeleton methods.

1. Byte Counter

Define a (public) class `ByteCounter` that computes the count of each byte in from a byte array or file and stores the result in a data structure. What data structure you use is up to you, but the class must have the following methods.

- ▷ A constructor with a `byte` array argument indicates that the counts are to be computed from that array. 2 P.
- ▷ A constructor with a `String` argument specifies the name of a file that the counts are to be computed from. The file could be text or binary, but you *must read it in binary mode* regardless of its form. Otherwise there could be problems comparing results on different platforms. In Java, a convenient way to read a file into a byte array is the `readAllBytes` method in the `Files` class. You can use the `Paths` class to obtain the system path to a local file. 5 P.
- ▷ `int getCount(byte b)` - returns an integer value that contains the number of occurrences of `b`. 5 P.
- ▷ `int[] getCount(byte[] b)` - returns an integer array that contains the number of occurrences of the corresponding bytes in the array `b`. 2 P.
- • `void setOrder(String order)` - Defines the order for the current object, which controls the ordering of the arrays returned by `toString` and `getElements` (defined below). If `order` equals `byte` (the default) the ordering is in terms of increasing byte value. If `order` equals `countInc` or `countDec` the order is in terms of increasing or decreasing count, respectively. For counts that are equal, the order should be in increasing byte order. If `order` is not one of `byte`, `countInc`, or `countDec` throw an `IllegalArgumentException`. 5 P.
- ▷ `byte[] getElements()` - returns a `byte` array of the bytes that have been counted, i.e. those with non-zero counts. The order of the array is specified by the current order. 2 P.
- ▷ `String toString()` - returns a `String` containing the current bytes and counts in the current order. It should have the format `byte:count`, separated by spaces, with no leading or trailing spaces. The `byte` should be formatted as a (signed) integer. For example: 2 P.

  ```
  32:3 44:1 63:1 72:1 97:1 101:2 104:1 108:2 111:3 114:1 117:1 119:1 121:1
  ```

  Note that in Java bytes are signed (and there is no unsigned byte), so that the implicit integer range of a byte is $[-128, 127]$. When you convert a byte to the integer, it should be negative for byte values 0x80 to 0xFF.
- • `String toString(String format)` - If `format` equals `char`, returns a `String` containing the counts formatted as above, but with the bytes as ASCII characters. For example (using the same ASCII values above): 2 P.

  ```
   :3 ,:1 ?:1 H:1 a:1 e:2 h:1 l:2 o:3 r:1 u:1 w:1 y:1
  ```

  Note that 32 is the ASCII value for a space.

2. Building a Huffman tree

A Huffman code for a set of characters (or bytes in our case) is constructed using their relative frequencies which are specified by the counts. These counts are used to construct a Huffman tree from which an efficient binary code is obtained for each character (byte). Please refer to lecture slides covered in class for a detailed illustration of how to build a Huffman tree. The basic steps of constructing a Huffman code are as follows:

- Read the file to determine the byte counts (or specify the counts directly). The `ByteCounter` class should come in handy here.
- Store each byte and its count in a `HuffmanNode` and create a linked list in increasing order of the counts. For testing consistency, *bytes with the same count should have their nodes ordered by the byte*. The list should *not* contain bytes with zero counts.
- The `HuffmanCode` class is constructed by creating a Huffman tree from the linked list, reusing the same nodes. The specific steps to create the Huffman tree are:
  1. Create a new node by merging the first two nodes in the list (i.e., the two nodes with lowest counts).
  2. Set the left child to the lowest count node, and the right child to the next lowest count node.
  3. Initialize the count of the new node to the sum of the counts of its children.
  4. Insert the new node into the linked list, maintaining the list order by count. Again for testing consistency, *new nodes with the same count as existing nodes should be inserted in front of the other nodes*.
  5. Repeat from step 1 until all the nodes in the list have been combined.
  6. Finally, traverse the tree to construct the binary codes for each byte. The binary code is defined along the path from the root to the leaf, which contains the byte. A left branch indicates a binary zero; a right branch, a binary one. See the lecture slides for an illustration.

Follow the specifications below to define the Huffman coder. You will first define the `HuffmanNode` class which will be used to create the linked list and, from that, the Huffman tree.

`public class HuffmanNode`

▷ Define a `HuffmanNode` class with the following fields: 2 P.

`public byte b` – the byte denoted by the node. This is meaningful only for the leaf nodes in a Huffman tree.

`public int count` – For a leaf node, this is the count of the byte `b`; for an interior node, count is the sum of the counts the node's children.

`public boolean[] code` – a boolean array representing the binary encoding for byte `b`. This is only defined for leaf nodes and ignored for interior nodes.

`public HuffmanNode next` – (optional) the link field used by the linked list from which the Huffman code is generated. In its initial state, this list is in order of increasing byte counts (i.e. their relative frequency in the file or array). In final Huffman tree, `next` should be null for all nodes. Note that if you are using Java's `LinkedList` class, this field is unnecessary, because it is provided by the class.

`public HuffmanNode left` – left child of a node in the Huffman tree

`public HuffmanNode right` – right child of a node in the Huffman tree

Note that the class and its fields need to be public for testing purposes. In a more realistic setting, `HuffmanNode` would be a private subclass of `HuffmanCode`.

▷ The constructor `public HuffmanNode(byte b, int c)` should initialize the `count` and `b` fields and set the left and right fields to `null`. 2 P.

`public class HuffmanList`

For testing purposes, implement a `HuffmanList` class that constructs the initial linked list of `HuffmanNode`s to build the Huffman tree. You are encouraged to use the built-in `LinkedList` class in Java. This is a generic class, which is declared as follows

```
 private LinkedList<HuffmanNode> list = new LinkedList<HuffmanNode>();
```

You can read the Java API for more information.

- ▷ A constructor with a `byte` array argument constructs a linked list from the supplied byte array. The list should use the `HuffmanNode` class and should be in increasing order of the byte counts obtained from the array. For bytes with the same count, the nodes should ordered by the byte. The list should not contain bytes with zero counts. 2 P.
- ▷ A constructor with a `String` argument constructs a `HuffmanList` as above, but obtains the bytes and their counts from the file referred to by the string. 2 P.
- A constructor with two arguments, a byte array and an integer array of equal length, returns a `HuffmanList` using the counts defined in the integer array for each of the corresponding bytes. This constructor should also 5 P.
  - allow the elements in the arrays to be specified in arbitrary (but corresponding) order 3 P.
  - throw an `IllegalArgumentException` if the bytes are not unique or if any of the counts are negative. 3 P.
- ▷ Provide an `Iterator` for the list and – if you are not using Java's built-in `LinkedList` class – implement the `hasNext` and `next` methods (if you use the `LinkedList` class, these methods are provided automatically). 5 P.

`public class HuffmanCode`

- The constructors for the `HuffmanCode` class are the same as for the `HuffmanList` class, described above, except that they construct the full Huffman tree and resulting code. When constructing the Huffman tree from the linked list, new nodes should be inserted in front of other nodes with the same count. Note that you don't have to use `HuffmanList` in your implementation of `HuffmanCode`.
  - a constructor with a `byte` array makes the code using the byte counts in the array 2 P.
  - a constructor with a `String` argument makes the code from the byte counts of a file 2 P.
  - a constructor with two arguments makes the code from a `byte` array and their corresponding counts 2 P.
- `boolean[] code(byte b)` – returns the binary encoding for byte `b`. Throw an `IllegalArgumentException` if the byte does not exist in the Huffman code. 5 P.
- `String codeString(byte b)` – returns the binary encoding for byte `b`, but in the form of a `String`, i.e. 1's and 0's for `true` and `false`, respectively. The string should contain no spaces. Throw an `IllegalArgumentException` if the byte does not exist in the Huffman code. 3 P.
- `String toString()` – returns a `String` containing the table of the binary encodings of each byte in the `HuffmanTree`. Each encoding should be separated by a newline (`\n`). The byte encodings in the table should be: 5 P.
  - in decreasing order of the byte counts , i.e. from the shortest to the longest codes. For grading consistency, the lesser of the two counts should be assigned to the left-subtree while merging. 5 P.
  - For bytes with the same frequency, the order should be in terms of increasing byte value. 3 P.

  The format should follow the example below (with a space after the colon, but with no leading or trailing spaces):

```
69: 01
84: 00
79: 100
83: 111
```

3. Compressing Files

In this part, you write a `HuffmanCoder` class to compress files. It should use the `HuffmanCode` class you have already defined. The public interface should contain the following methods:

- The constructor takes two `String` arguments `inputFile` and `outputFile` which defines the files to be read and written. 5 P.
- `void compress()` – should define a `HuffmanCode` for the `inputFile` and write (in binary form) the `HuffmanCode` object and the compressed `inputFile` to the `outputFile`. 5 P.

A Binary Writer Class

Along with the JUnit fundamental test suite for the draft, we also provide a `BinaryWriter` class that you can use. It has the following functionality:

- a constructor with a string argument specifies the binary file to write to, e.g.
  `BinaryWriter writer = new BinaryWriter(outputFile)`
- The method `writeBinaryArray(boolean[])` writes the boolean array to the file. Any last bits of the binary array that do not fill a complete byte will be buffered and written during the next call.
- The `close()` writes any remaining bits to the file and closes it.

4. Other grading considerations

We will also take into account the following when grading your assignment. Because many these considerations are subjective, one TA will be responsible for assigning the grades in each of these areas to help ensure consistency.

a) *Design* (5 P.) Design an implementation[1] considers how you chose to structure your program to implement the specification. At a high level, the assignment itself dictates much of the structure, e.g. the methods you need to provide, but there are still choices that need to be made regarding your choice of helper methods. At a lower level, design also concerns how you structure the code within a method and your choice of data structures and algorithms. For grading we will consider things like, how you chose to break down a problem into smaller components, how elegantly your code handles edge cases, etc. As much of the "design" is prescribed from the assignment specifications, the design constitutes only a small portion of the overall grade.

b) *Style* (10 P.) Style considerations include (from *Practice of Programming*):

   - Variable name choices. Are they descriptive and do they convey information about their purpose? Note for simple concepts, it perfectly acceptable, even preferable to use, for example, `i` for an index or `n` for an integer.
   - Expressions and statements. Are they clear? Do they make the meaning as transparent as possible? Well-written code should not need much commenting, if any. Is it formatted in a way that aides readability?
   - Consistency and Idiom. Is formatting consistent? Are common operations consistent with programming idiom? (the textbook, for example)
   - Comments. Do they aid the reader in understanding the code? Comments that restate what is already clear the code are redundant and not helpful. Nor are comments that are not consistent with the code. For more complex methods, the comment should state the running time of the method (or running times if there are different cases).
   - Exceptions. When you throw an exception, you should choose an appropriate name for the exception and print and informative error message.

c) *Testing* (10 P.) Your code should include JUnit tests to check that the code is correct and performs as specified. Your tests should be well thought-out with clear reasoning behind them, which you should explain in your comments. Having a large number of tests that are essentially equivalent will receive deductions. Remember that the whole goal of testing is to make the process of implementation and debugging faster and more efficient.

[1]The design, style, and testing considerations follow the guidelines and ideas in *The Practice of Programming* by Kernighan and Pike.

So don't waste your time writing a bunch of unnecessary tests. Here are some guidelines from the *Practice of Programming*:

- Test as you write the code. This goes along with writing incrementally. Once you implement a part of the assignment, write a test to check its correctness.
- Test code at its boundaries. This is also referred to as testing "corner cases." Do conditions branch in the right way? Do loops execute the correct number of times? Does it work for an empty input? A single input? An exactly full array? Etc.
- Test systematically. This goes along with testing as you write the code. Have you checked that every line is correct? Do you have test code that executes every line and condition? Test the simple parts first, especially those parts you're going to reuse. You want to know that they're working when you use them. If you don't, debugging is much more difficult.
- Test automatically. Have test suite that automatically runs all the tests. JUnit helps provides this functionality.
- Stress test. Does your code work on large inputs? Random inputs? It's easy to make implicit assumptions that aren't always valid. Stress testing can help uncover these.

The same book also provides an amusing quote from the famous computer scientist and Case alum Don Knuth (the "father" of the analysis of algorithms among many other things) on how he writes test code:

> I get into the meanest, nastiest frame of mind that I can imagine, and write the nastiest [testing] code I can think of; then I turn around and embed that into even nastier constructions that are almost obscene.

General Instructions and Guidelines

You must submit your draft and final via blackboard by the due dates listed at the top of the assignment. Late assignments cannot be accepted, so start early and use the feedback from the draft to get help if you need it.

Turning in your code. You should submit a zipped folder of your java source and JUnit tests files via Blackboard. Follow these instructions:

- Make a separate submission folder with the following name: `EECS233_abc123_P2` where `abc123` is replaced by your own Case Network ID.
- The folder should contain your java source code, your JUnit tests, and any additional files (e.g. if you have a README.txt).
- Your JUnit filenames should have the form `ClassNameTest.java`.
- Do *not* include the java `.class` files. They take up too much space, and we will compile your files ourselves.
- Your submission folder should not contain subfolders.
- Compress your submission folder using the `.zip` format (do not use `.rar` or some other format)
- The compressed filename should be the same as the folder name, but with the `.zip` extension.
- Submit this file via Blackboard.

Drafts. Draft submissions allow you to check your code for correctness against our own suite of JUnit tests. This will be the same test suite used for the final version.[2] In general, there will be set of tests for each required component. The draft also helps ensure that you get started early, which is vital for completing programming assignments on time.

[2]Although we reserve the right to correct errors or shortcomings in the test suites when we find them. If this happens, we will try to be fair, so that students are not led to have a false sense of completion. I will even consider awarding bonus points to those who identify useful test cases that we have overlooked.

It is in your best interest that your draft be as complete possible. This will allow you to benefit from feedback from the full test suite and give you time to identify problems and fix them or get help.

- **Fundamental items**. Assignment items marked with "▷" are *fundamental*. These should be simple and straightforward to implement and represent the minimal functionality we need to test your code. We will provide you with our JUnit tests for these items, so that you can check the correctness of your code yourself. Needless to say, do not modify them, but if you find errors or potential issues, please bring them to our attention as soon as possible. You can use these JUnit tests as templates to write your own test suite for the rest of the assignment.
- **Use exact method names**. Use the exact method names specified, including the character case. This is required to check your code. Please be aware that we cannot make modifications to your code for grading purposes. Also, do not declare your source as a package. This isn't necessary and interferes with the test suite.
- **Complete skeleton methods are required**. In order for the JUnit tests to compile, you need to write skeletons for *all* the required methods. These can be empty, i.e. without any code, but they must have properly typed arguments and return values. If you do not this, we will not be able to test your code.
- **Ensure your code compiles**. You must ensure that your code is in a stable state and compiles before submitting your code. Our test suite script will automatically using the latest submitted version. We will only run the test suites on new submitted drafts once a day, so do not ask for your code to be regradedunless there is a specific issue you are resolving with the TA. You should be writing your own JUnit test suites as you code to check correctness.
- **Throw unchecked exceptions**. Java has two types of exceptions: checked and unchecked. Unless otherwise specified, you should throw unchecked exceptions, which means you do not put a "throws" statement in your method declaration. The assignment will specify which type of exception you should throw.

Version Control. You should always write your code incrementally, with a short write-test-debug cycle. Once you have a component written (which could be just a single method), write JUnit tests that check that your implementation is correct. Since we cannot grade what we cannot compile, you need to use some kind of version control to ensure that you have a saved working version once each step is completed. This could be as simple as copying the directory or using a sophisticated version control system like "git". Anything is fine, as long as you always maintain a working copy of your code that you can turn in (or revert to if you introduce a bug that you can't track down).

Use an Automated Backup System. Everyone should be using a backup system that backs up your files automatically and seamlessly, such as Dropbox or Google Drive, among many others. They are free services for small space requirements, so there is no excuse anymore for loosing your work to a hard drive failures or accidentally deleted files. If you haven't done so already, set up a such a system now.

Final Version. Like the draft, make sure your code compiles and is in a stable state (i.e. your best working version) before the deadline. We will run the test suite on the version you have labeled *final* following the deadline.

General Strategies. It is helpful to work through the logic of the implementation for the problems in the assignment on a piece of paper, *before* you write any code. This can include things like data structure design, implementation, and small examples you use to work through the steps of your algorithms. You do not need to turn this in, but it can be helpful for grading in you include a README.txt file or a pdf scan of your notes that explains your design choices to the grader. Your code comments should describe clearly your high-level logic, design choices, and the running time and/or memory complexity of your algorithms.

Cheating and Plagiarism. It should go without saying that you should solve the problems and write the code on your own. You can consult the book and the internet for java questions, but do not look for solutions to the problems given in the assignments. You are encouraged to get help from your classmates regarding basic programming issues, debugging problems, features of Eclipse, etc., but like I mentioned in class, be careful about giving (or receiving) too much help. Not only might it spoil someone's learning efforts, it may appear as plagiarism. To ensure fairness, we will run software to check for cheating, so please do not copy code. Also keep in mind that your goal should be to develop the ability to think through problems, find solutions, and implement them in code. Some exam questions will be designed to test to what extent you have achieved this using examples drawn from the assignments.