Homework 4 -- Coding Trees

Due Nov 4 by 11:59pm **Points** 100 **Available** after Oct 21 at 12am

> The goal of this assignment is to design and implement a program that can encode and decode data using *prefix encodings*.

Codes

Encoding and decoding are common operations on data. Given data in the form of symbols (e.g. text), it can be encoded by translating each symbol into a unique code, possibly consisting of many symbols. Decoding applies this process in reverse. The unique codes may be made of a different set of symbols (e.g. the original symbols may be characters, but the codes are bits). We call this set "coding symbols".

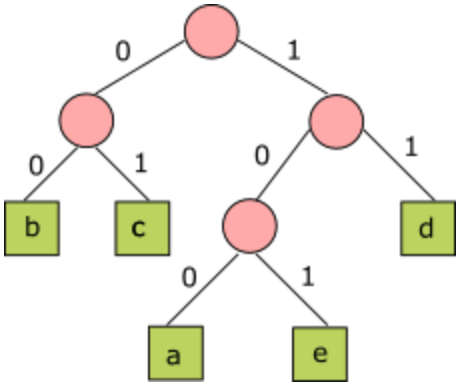
The main component of encoding and decoding can be thought of as a *symbol -> code* dictionary representing a coding scheme. For example, consider that all input messages are made of symbols *{a,b,c,d,e}*, and the coding symbols are *{0,1}*. An example dictionary could be *{a -> 10, b -> 1100, c -> 1101, d -> 01, e -> 00}*. Then, an input message "dab" will be encoded as "01101100" by replacing the symbols 'd', 'a', 'b' with their respective codes from the dictionary. Similarly, an encoded string "11001001" will be decoded to "bad" using the same dictionary.

Coding schemes may have some unique characteristics. For example, certain encoding schemes generate codes such that no code is a prefix of another code (e.g. 10 and 101 cannot both be codes, because the first is a prefix of the second). Such schemes are called *prefix codes* and are often used since a sequence of codes then requires no delimiters between codes, and will have only one possible decoding. You can verify that the dictionary in the above example shows a prefix coding scheme. One way to look at prefix codes is by using *prefix trees*.

As another example, consider the dictionary D:

```
a:100
b:00
c:01
d:11
e:101
```

Formally, in this example, the original symbol set is *{a,b,c,d,e}* and the code symbols are *{0,1}*. We can see that the above code is a prefix code meaning that no code is a prefix of another. Thus it can be represented using a tree *U*:



All the original symbols are leaves in this tree. The others are transition nodes. Since there are only two code symbols, this is a binary tree. Each edge of the tree is annotated (for illustration) with a code symbol (left is 0, right is 1).

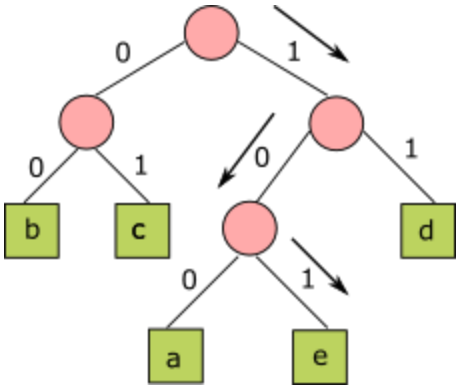
A complete coding tree is full: each transition node has children to its full capacity (2 above, since the tree is binary).

Decoding

Given an encoded message "10001101", we can decode it as follows:

1. Start at the root of the tree.
2. Read the next symbol.
3. Turn "left" or "right" depending on the read symbol.
4. If a leaf is reached, output the character at that leaf, restart at the root.
5. Go to step 2.

This process is illustrated for the decoding "101" into 'e':



Applying this process to the encoded message "10001101": we use the first three symbols "100" to arrive at 'a', the next two symbols "01" to arrive at 'c' and the last three symbols "101" to arrive at 'e'. Thus the encoded message "10001101" decodes to the string "ace".

Encoding

How is a prefix code generated? One can arbitrarily assign a prefix code for a symbol set. However it is possible to generate "better" prefix codes. One way is to use a scheme called *Huffman encoding*. This encoding generates a custom coding scheme for the message to be encoded, such that the length of the encoded message (in terms of number of symbols) is minimized.

The Huffman encoding algorithm to generate a binary prefix code, given the source message `M` is as follows:

1. Create a frequency table *(a,f(a))* where *a* is a symbol in *M* and *f(a)* is the number of times *a* occurs in *M*.
2. Initialize a coding table *(a,g(a))* where *a* is a symbol in *M* and *g(a)* is the desired code for *a*. All codes are initially empty.
3. Add all symbols into a priority queue *Q*, using their frequencies as priorities. When the frequencies are the same, the item that is lexicographically earlier has the lower priority. In general *Q* contains items with one or more symbols in them.
4. Pop *2* items *x* and *y* from the queue with the lowest frequencies *f(x)* and *f(y)* respectively.
5. For each symbol x_i in *x*, add a prefix of *0* to $g(x_i)$. Similarly, for each symbol y_i in *y*, add a prefix of *1* to $g(y_i)$.
6. Add a new entry *(x.y, f(x)+f(y))* to *Q*, where *x.y* is the concatenation of *x* and *y*.
7. If there is more than one item in *Q*, go to step 4. Otherwise, report the resulting coding table.

The above algorithm generates binary codes, i.e. its coding symbol set is *{0,1}*.

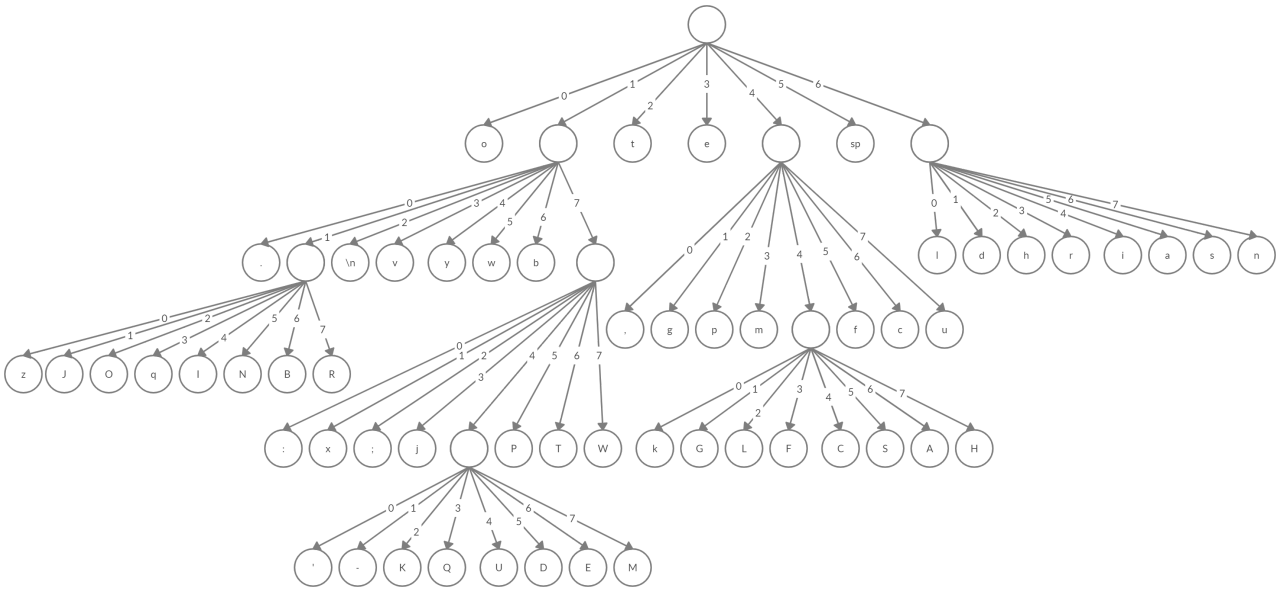
2020fall-cs5010-p4-huffman

Generalizing to larger symbol sets

To generalize the above algorithm to a coding symbol set $C=\{c_1, c_2, \ldots, c_n\}$ change step 4 above to pop *n* items (or all if the *Q* has fewer than *n* entries) from *Q*, and then add prefixes $c_1, c_2, \ldots, c_n$ to them in step 5. This generalization may result in a coding tree that is not full. For example, the **Declaration of Independence (declaration.txt)** has the following frequencies:

\n = 62	A = 21	I = 6	Q = 1	c = 166	k = 13	s = 459
sp = 1290	B = 7	J = 5	R = 8	d = 251	l = 212	t = 627
’ = 1	C = 18	K = 1	S = 19	e = 857	m = 140	u = 208
, = 103	D = 2	L = 16	T = 13	f = 163	n = 478	v = 74
- = 1	E = 2	M = 4	U = 1	g = 116	o = 508	w = 84
. = 37	F = 17	N = 6	W = 13	h = 325	p = 125	x = 9
: = 9	G = 14	O = 5	a = 457	i = 443	q = 5	y = 81
; = 10	H = 24	P = 13	b = 88	j = 11	r = 417	z = 4

Which, when mapped to *C={01234567}* gives the following Huffman tree:



What to do

Design and implement the interfaces/classes to be able to encode and decode these types of coding sets. Your implementation should be able to:

- decode messages given a *symbol->code* dictionary containing a *prefix code*.
- encode messages given a *symbol->code* dictionary containing a *prefix code*.
- be able to handle code sets with any number of symbols, not just binary. For example, a code set using hexadecimal code symbols would have "0123456789abcdef" as the coding symbols, and will result in a tree where each transition node has up to 16 children.
- be able to read a message of arbitrary length and be able to generate a *Huffman encoding* using the algorithm described above.
- be able to handle any character set including punctuation, line breaks, spaces, etc.

You are free to use any existing Java classes and interfaces (within the JDK), as well as any implementations provided to you in this course. You may also design your own classes and interfaces. The only constraint is that you implement the decoding algorithm using a coding tree faithfully (as described above).

In addition, develop a driver program that can:

- read messages from either the keyboard or a file

- write messages to the screen or to a file
- write a binary encoding to a file using either binary or hexadecimal. For example, the binary encoding:

  ```
  11000001011011011010110101101101011011111101011
  ```

  could be written in hexadecimal as:

  ```
  C16DAD6D6B7EB
  ```

- read and write an *prefix encoding* to a human-readable text file.

Create a JAR file of your program

In order to make your application easier to run, you are required to create and submit a `JAR` file.

What to submit

Log on to the **Handins submission server (https://handins.ccs.neu.edu/)** and upload a ZIP file of your assignment. Your ZIP file should contain three folders: src/, test/ and res/ (even if empty).

- All your code should be in src/.
- All your tests should be test/.
- Your original and revised design document should be in res/.
- Submit a correct `JAR` file in the res/ folder. We should be able to run your program from this jar file.
- Submit any supporting documents in the res/ folder. These may include text files that you use in your example runs that contain original messages, encoded messages, and prefix encodings.
- Submit at least two example runs of your program the res/ folder that communicate to us how to specify commands to run your program to verify that it meets all of the above specifications.
  - One of the runs should show the reading of the message from the keyboard
  - One of the runs should show the writing of the message to the screen
  - One of the runs should show the reading of a message from a file
  - One of the runs should show the writing of a message to a file
  - One of the runs should show the reading of the *prefix encoding* from a file and show that it can successfully be used to decode a message
  - One of the runs should show the writing of a *prefix encoding* to a file
- Submit a `README.md` file that documents how to use your program, which parts of the program are complete, and design changes and justifications. You should distinguish the details of the two example so that we know what to look for in each run. The file should also include any necessary citations (see syllabus).

Criteria for grading

You will be graded on:

- Whether your code is well-structured and clean (i.e. not unnecessarily complicating things or using unwieldy logic).
- Correctness of your implementations including whether your example runs contain enough information for us to run your program from the provided `JAR` file and to see that all of the required features are working correctly.
- Whether your code is well documented using well-formed English sentences.
- Whether your code is follows correct style (according to the style grader).
- Self-assessment to be completed on the Handins submission server.