CoE 164

Computing Platforms

Midterm Problem

Academic Period: 2nd Semester AY 2022-2023
Workload: 12 hours
Synopsis: Obfuscated, compressed, and executable text

Annex A

The diropql Language

diropql (diro portable query language) is a minimalistic language used to "encrypt" text across some unsecure channel. A diropql program starts with 10000 memory cells with each cell initialized to a value of zero. All cells take in only 1-byte *unsigned* integers. It will also start with a memory pointer `mp`, an instruction pointer `ip`, and an output queue `oq`. `mp` and `ip` are nonnegative integers initialized to zero since diropql is a zero-index language. `oq`, on the other hand, is initially empty and is assumed to have an infinite size.

A diropql source code is a string encoded in ASCII or UTF-8. If saved as a file, it should have the extension `.dpql`. It is run by the diropql interpreter, which starts by looking at the program source code and finding the first valid command in the language by incrementing `ip` one at a time across it. Any invalid commands are ignored by the interpreter. When a valid command is read, the interpreter will process it depending on the command and the value pointed to by `mp`. diropql consists of the following seven commands capable of manipulating the contents of the memory cells.

command	description
`l`	decrement `mp` by one
`r`	increment `mp` by one
`i`	increment content pointed by `mp` by one
`d`	decrement content pointed by `mp` by one
`o`	push content pointed by `mp` to `oq`
`p`	change ip to the index of the matching q command *plus one* if the content pointed by `mp` is zero

`q`	change ip to the index of the matching p command *plus one* if the content pointed by mp is nonzero

Note that in general, the interpreter increments `ip` after each command. However, the `p` and `q` commands modify the `ip`. In this case, the nesting of `p` and `q` follow the same as in math equations (i.e. inner `pq`s match and outer `pq`s match in the following program: `ppidqq`).

If `mp` will point to an invalid memory cell due to an `lr` command, the value of `mp` will wrap around. For example, if `mp` is currently 0 and the next command decrements it by one, the new value of `mp` will be 99999.

If the contents of a memory cell will overflow due to an `id` command, the value will be clamped. For example, if a memory cell currently contains 255 and the next command increments it by one, the value will be unchanged.

Annex B
diropqlz File Format

diropql is a minimalistic language containing only seven characters. Hence, programs in this language can get very large. The diropqlz file format stores a compressed version of a diropql program. Programs written in this format should be decompressed first before being run into the compiler.

A diropqlz file is Base85 encoded so that it can be sent across channels in a human-readable format. In addition, programs that are to be saved in this file format should have the `.dpqlz` extension. To convert a diropql program into its `.dpqlz` equivalent, it has to be first compressed using the obfuscator. Then, the result should be appended with metadata in the following layout written from left to right:

- 8 bytes - 64-bit number denoting the length in bytes of the obfuscated message as the ceiling of the number of bits divided by eight.
- 1 byte - 8-bit number denoting the number of bits to *exclude or ignore* starting at the end of the obfuscated message. It should be a value between 0 and 7.
- 8 bytes - 64-bit number denoting the index I that was part of the output of the Burrows-Wheeler transform
- 16 bytes - 16 1-byte numbers corresponding to the bit lengths of the canonical Huffman codebook derived from the encoding process. The first nine numbers correspond to the alphabet of the run-length encoding output sorted by value. The remaining seven numbers are reserved for future use, and should be set to zero.
- Rest of the binary bits - the obfuscated message itself

All values are written in big-endian (i.e. the most significant bit appears at the leftmost).

This final compressed output may have to be appended with bits set to 0 *after* the obfuscated message to ensure that the total length of the output in bits is divisible by eight.

After the compressed program has been arranged, the program can now be encoded into Base85. Then, a magic string (DIROPQLZ) will be prepended to the encoded program to finish processing.

Annex C
Burrows-Wheeler Transform

The *Burrows-Wheeler transform* (BWT) is a text transform invented in 1994. The most simple algorithm to do the transform is as follows:

1. Get a message M and append a *sentinel character* to it. This sentinel is used to denote the end of the string. In textbooks, they usually use a dollar sign ($), but the `NUL` ASCII character (`\0`, ␀) is most practically used. From now on, this sentinel is part of the original message M.
2. Get M and its $|M| - 1$ other permutations by shifting M to the left one letter at a time, and moving the shifted out letter to the rightmost side of M. Place M and the permutations into an array S.
3. Sort S in lexicographically increasing order (i.e. ascending alphabetical order/ASCII order). Let this sorted array be named S_{sorted}.
4. Get the last letter of each element of S_{sorted} and concatenate them in the order in which permutation the letter belongs. This is now the transformed text. The index I of the row where M is is also recorded.

The transform groups the same occurring letters via permutation. In addition, the transform is reversible, with the simple algorithm as follows:

1. Get the transformed message V and assume that it already contains the sentinel character. Also, initialize a $|V| \times |V|$ matrix C_{BWT}.
2. The following should be done $|V|$ times:
   a. Arrange V like a column vector and append it in the rightmost empty column of C_{BWT}.
   b. Sort C_{BWT} in lexicographically increasing order (i.e. ascending alphabetical order/ASCII order).
3. Find the row in C_{BWT} whose last element, or character, is the sentinel character. That row is the original string.

The transform can be immediately made from a *suffix array*. A suffix array of a string M is a 1D array consisting of the indices of the starting character of a suffix in M. The suffixes are sorted by increasing alphabetical order and then by increasing length. The BWT of the string is the corresponding index of the character pointed to by the suffix array.

Annex D
DC3/skew Algorithm

The DC3 (difference cover mod 3) algorithm builds the suffix array of a string in around linear time. It looks at the characters in the string by threes and does a recursive sort and merge routine to build the array.

To be outlined…

Annex E
Move-To-Front Transform

The second algorithm is the *move-to-front transform*, which is a simple transform relying on the fact that the characters appearing the most in a text are "used" more frequently. If a text is read from left to right, it makes sense to move these characters at the beginning once they appear.

Encoding

The transformation algorithm is as follows:

1. Initialize a list of characters used in the text called the *alphabet* Σ. The characters in it should have been pre-sorted in alphabetical order.
2. Initialize the output queue S where the transformed string will be placed.
3. Get a message M and do the following $|M|$ times, for each character m_i in M
    a. Find the index *from zero* of m_i in Σ. Push this index into S.
    b. Remove that same character from Σ and reinsert it at the front of Σ.
4. The transformed text is given as a series of numbers contained in S.

As an example, we have a string M:

$$M = bananaaa!␀$$

We let ␀ be the *sentinel character* as described in Annex A (BWT). Getting the unique characters and sorting them in ascending ASCII value will yield the ordered set alphabet Σ:

$$\Sigma = \{␀, !, a, b, n\}$$

Now, we let the ordered set S contain the transformed message M. We start with the first character in M and see that it is the 3rd character (starting from zero) in Σ. S will now currently have the value $S = \{3\}$ and Σ will be changed such that this third character is moved at the very front of so that the new Σ becomes $\Sigma = \{b, ␀, !, a, n\}$.

Going through each character of M, we get the following progression:

Iteration Index i	**Letter in Message M_i**	**Current Alphabet Σ_i**	**Transformed Message S**	**New Alphabet Σ_{i+1}**
0	b	␀, !, a, b, n	3	b, ␀, !, a, n
1	a	b, ␀, !, a, n	3, 3	a, b, ␀, !, n
2	n	a, b, ␀, !, n	3, 3, 4	n, a, b, ␀, !

3	a	n, a, b, ␀, !	3, 3, 4, 1	a, n, b, ␀, !
4	n	a, n, b, ␀, !	3, 3, 4, 1, 1	n, a, b, ␀, !
5	a	n, a, b, ␀, !	3, 3, 4, 1, 1, 1	a, n, b, ␀, !
6	a	a, n, b, ␀, !	3, 3, 4, 1, 1, 1, 0	a, n, b, ␀, !
7	a	a, n, b, ␀, !	3, 3, 4, 1, 1, 1, 0, 0	a, n, b, ␀, !
8	!	a, n, b, ␀, !	3, 3, 4, 1, 1, 1, 0, 0, 4	!, a, n, b, ␀
9	␀	!, a, n, b, ␀	3, 3, 4, 1, 1, 1, 0, 0, 4, 4	␀, !, a, n, b

The transformed message is therefore encoded as the sequence:

$$S = \{3,\ 3,\ 4,\ 1,\ 1,\ 1,\ 0,\ 0,\ 4,\ 4\}$$

Decoding

The transform is reversible using the following algorithm:

1. Find a way to get a hold of the original pre-sorted alphabet Σ used in the transformation algorithm above.
2. Initialize the output queue M where the transformed string will be placed.
3. Get an array of numbers S and do the following $|S|$ times, for each value s_i in S
    a. Get the character in Σ corresponding to the index s_i. Push this character into M.
    b. Remove that same character from Σ and reinsert it at the front of Σ.
4. The transformed text is given as a series of characters contained in M.

As an example, we have the following sequence encoded using move-to-front transform:

$$S = \{3,\ 3,\ 4,\ 1,\ 1,\ 1,\ 0,\ 0,\ 4,\ 4\}$$

The original alphabet was also reconstructed or received as:

$$\Sigma = \{␀, !, a, b, n\}$$

Now, we let M contain the reconstructed message from S. We start with the first value in S, treating it as an index, and looking up the corresponding character at that index (starting from zero) in Σ. M will now currently have the value $M = b$ and Σ will be changed such that this third character is moved at the very front of so that the new Σ becomes $\Sigma = \{b, ␀, !, a, n\}$.

Going through each value in S, we get the following progression:

Iteration Index i	Value in Sequence S_i	Current Alphabet Σ_i	Reconstructed Message M	New Alphabet Σ_{i+1}
0	3	␀, !, a, b, n	b	b, ␀, !, a, n
1	3	b, ␀, !, a, n	ba	a, b, ␀, !, n
2	4	a, b, ␀, !, n	ban	n, a, b, ␀, !
3	1	n, a, b, ␀, !	bana	a, n, b, ␀, !
4	1	a, n, b, ␀, !	banan	n, a, b, ␀, !
5	1	n, a, b, ␀, !	banana	a, n, b, ␀, !
6	0	a, n, b, ␀, !	bananaa	a, n, b, ␀, !
7	0	a, n, b, ␀, !	bananaaa	a, n, b, ␀, !
8	4	a, n, b, ␀, !	bananaaa!	!, a, n, b, ␀
9	4	!, a, n, b, ␀	bananaaa!␀	␀, !, a, n, b

The reconstructed message is therefore:

$$M = bananaaa!␀$$

MTF in diropqlz

When implementing this as part of diropqlz compression, the initial alphabet Σ used should be the seven letters sorted in ascending order of their ASCII values $\Sigma = \{d, i, l, o, p, q, r\}$.

Annex F
Run-Length Encoding

The third algorithm is *run-length encoding*, which is another simple transform that encodes a series of the exact same characters as the number of occurrences. Using the previous transforms, the currently compressed data (as an array of numbers) may have a long run of zeros

Encoding

We can use the following algorithm to further compress the data:

1. Initialize a counter N_{zero}, which counts the current number of zeros that will be encountered during the encoding process. Its value should initially be zero.
2. Initialize an output queue L where the encoded data will be stored.
3. Get an array of numbers S and do the following $|S| + 1$ times, for each value s_i in S
    a. Check whether the current loop index i is not greater than $|S| - 1$ and s_i is zero.
        i. If it is, increment N_{zero}.
        ii. Otherwise,
            1. Convert N_{zero} into binary and add one. Push each digit starting from the least significant bit into L *except* the most significant bit.
            2. Reset N_{zero} to zero.
            3. Finally, push $s_i + 2$ into L if s_i exists.

As an example, we have the following sequence S:

$$S = \{3,\ 3,\ 4,\ 1,\ 1,\ 1,\ 0,\ 0,\ 4,\ 4\}$$

Now, we let L contain the encoded sequence from S and $N_{zero} = 0$. We start with the first value in S and checking whether it is a zero. Since it is not, we can already append its value *plus two* directly to L and will now currently have the value $L = \{5\}$.

Going through each value in S, we get the following progression:

Iteration Index i	**Value in Sequence S_i**	**Current number of zeros N_{zero}**	**New number of zeros N_{zero}**	**Encoded Sequence L**
0	3	0	0	5
1	3	0	0	5, 5

2	4	0	0	5, 5, 6
3	1	0	0	5, 5, 6, 3
4	1	0	0	5, 5, 6, 3, 3
5	1	0	0	5, 5, 6, 3, 3, 3
6	0	0	1	5, 5, 6, 3, 3, 3
7	0	1	2	5, 5, 6, 3, 3, 3
8	4	2	0	5, 5, 6, 3, 3, 3, 1, 6
9	4	0	0	5, 5, 6, 3, 3, 3, 1, 6, 6
10		0	0	5, 5, 6, 3, 3, 3, 1, 6, 6

For iteration index 6, note that the current value we are looking for in S is a zero. Therefore, we will not append anything to L, *but* we will increment N_{zero} by one.

For iteration index 8, note that the current value we are looking for in S is *not* a zero. In addition, N_{zero} is greater than one. Hence, we will convert $N_{zero} + 1$ to its binary form ($3_{10} = 11_2$) and append each digit starting from the rightmost digit to L *except* the most significant 1 bit. Finally, we reset the value of N_{zero} to zero. After this, don't forget to append the current value of S *plus one*!

For iteration 10, note that the sequence has already ended in iteration 9. However, this extra iteration is needed so that if N_{zero} is a nonzero value, we can append it at the end of L.

The transformed message is therefore encoded as the sequence:

$$L = \{5, 5, 6, 3, 3, 3, 1, 6, 6\}$$

Decoding

The transform is reversible using the following algorithm:

1. Initialize a stack N_{zero}, which will contain the binary representation of the current number of zeros that were encoded during the encoding process. Its value should initially be zero.
2. Initialize an output queue S where the decoded data will be stored.
3. Get an array of numbers L and do the following $|L| + 1$ times, for each value a_i in L
    a. Check whether the current loop index i is not greater than $|L| - 1$ and whether a_i is zero or one.

i. If it is, push a_i into N_{zero}.

ii. Otherwise,

1. Push a one into N_{zero}, reverse the order of the digits, convert it into its decimal equivalent, and subtract by one. Push the appropriate number of zeros equivalent to the new N_{zero} amount to S.
2. Reset N_{zero} to zero.
3. Finally, push $a_i - 2$ into L if a_i exists.

As an example, we have the following sequence L encoded using run-length encoding:

$$L = \{5, 5, 6, 3, 3, 3, 1, 6, 6\}$$

Now, we let S contain the reconstructed sequence from L and $N_{zero} = 0$. We start with the first value in L and checking whether it is greater than one. Since it is, we can already append its value *minus two* directly to S and will now currently have the value $S = \{3\}$.

Going through each value in L, we get the following progression:

Iteration Index i	**Value in Sequence L_i**	**Current binary digits of N_{zero}**	**Current decimal number of zeros $N_{zero} + 1$**	**New binary digits of N_{zero}**	**Reconstructed Sequence S**
0	5	0	0	0	3
1	5	0	0	0	3, 3
2	6	0	0	0	3, 3, 4
3	3	0	0	0	3, 3, 4, 1
4	3	0	0	0	3, 3, 4, 1, 1
5	3	0	0	0	3, 3, 4, 1, 1, 1
6	1	0	0	1	3, 3, 4, 1, 1, 1
7	6	1	3	0	3, 3, 4, 1, 1, 1, 0, 0, 4
8	6	0	0	0	3, 3, 4, 1, 1, 1, 0, 0, 4, 4
9		0	0	0	3, 3, 4, 1, 1, 1, 0, 0, 4, 4

For iteration index 6, note that the current value we are looking for in S is one. Therefore, we will not append anything to L, *but* we will treat this digit as a binary digit and push it into N_{zero}.

For iteration index 7, note that the current value we are looking for in S is greater than one. In addition, N_{zero} is greater than one. Hence, we will push a 1 into N_{zero}, reverse the digits and add one to get the true binary form ($3_{10} = 11_{2}$). We then convert it to its decimal equivalent and append $N_{zero} - 1$ zeros to S. Finally, we reset the value of N_{zero} to zero. After this, don't forget to append the current value of L *minus one*!

For iteration 9, note that the sequence has already ended in iteration 8. However, this extra iteration is needed so that if N_{zero} is a nonzero value, we can append the appropriate number of zeros at the end of S.

The reconstructed sequence is therefore:

$$S = \{3,\ 3,\ 4,\ 1,\ 1,\ 1,\ 0,\ 0,\ 4,\ 4\}$$

Annex G
Huffman Encoding

The final algorithm is *variable-length encoding*, which is a common transform that encodes all characters in a message into different binary codewords of variable lengths. Characters that more frequently show up are assigned to shorter codewords while those that rarely appear are assigned to longer codewords. A well-known variable length encoder is the *Huffman encoder*, which was published in 1952.

Huffman Tree Building

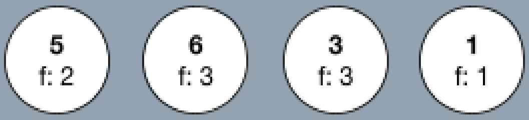
Encoding messages using Huffman encoding utilizes a greedy approach by building a binary tree based on the frequency of the characters that appear in the message. The algorithm for building the tree and encoding map is as follows:

1. Get all of the values uniquely used in the message C. Push them into a priority queue Q of nodes, with each value as its own (leaf) node, and where the frontmost element of it will be the least frequent value, breaking ties with the value itself.
2. Do the following until Q has a single element.
    a. Get the first two elements of Q.
    b. Create a new node from them with these two elements as its children. The frontmost of the two is the left child, and the other is the right child.
    c. Set the frequency of the new node to the sum of the frequencies of the two child elements.
    d. Push this new node into Q.

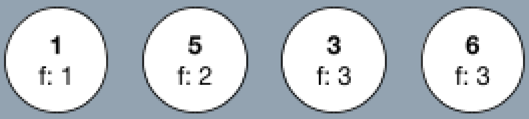
As an example, we have the following sequence C:

$$C = \{5, 5, 6, 3, 3, 3, 1, 6, 6\}$$

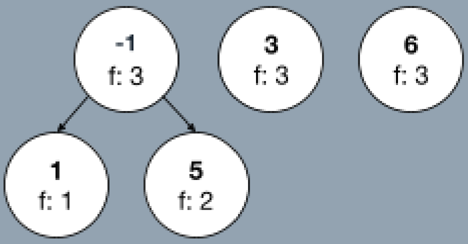
We get the unique values in C and find the frequency of each appearing in it to yield the following forest of nodes:



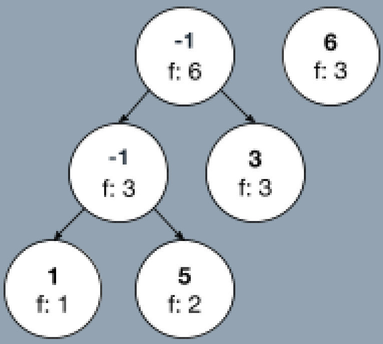
Next, we sort them in increasing frequency, and then increasing value from left to right. We can think of it as these forest of nodes pushed into the priority queue Q, which will automatically sort them. This will yield:



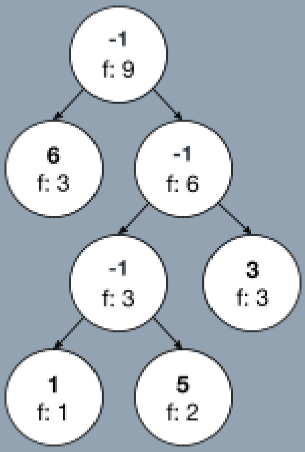
We then start by getting the two leftmost nodes and "merging" them by making a "dummy" node whose value is -1 (or any "null" value) and frequency being the sum of the frequencies of these two leftmost nodes. Then, we connect the leftmost of the two nodes to the left of this dummy node and the other one to the right.



Repeating the same process as outlined above, we sort the nodes in the uppermost row in increasing frequency, and then increasing value. Then, we merge the two leftmost nodes to yield the following arrangement:



Another repetition of the same process will yield the Huffman tree corresponding to the sequence C:

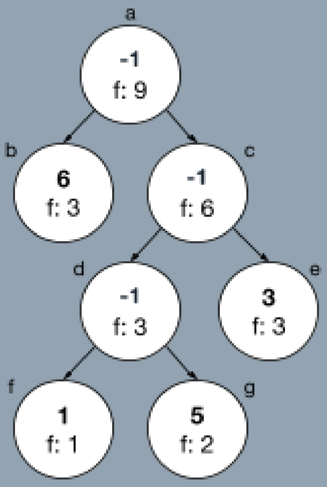


Codeword Mapping

Let the single tree T left in Q be the encoding tree that will be used to encode the message. To find the Huffman codeword for the character using T:

1. Initialize a codeword stack S where the codeword digits will be stored.
2. Push 2 into S. Note that this value will *not* be part of the binary codeword.
3. Start at the root node of T.
4. Do the following until S is empty:
    a. If the current node is not a leaf node (i.e. has a character), traverse downward to the left and push a 0 into S.
    b. If the current node contains the character, then skip to step (4).
    c. Otherwise, if we have already visited the left, but we have not yet done so to the right:
        i. Pop one digit off of S and discard it.
        ii. Traverse downward to the right and push a 1 into S.
    d. Otherwise, if we have already visited the right, too:
        i. Pop two digits off of S.
5. Remove the first element from S and it will now contain the binary codeword.

As an example, we have the following Huffman tree with each node labeled with a letter for reference:



We will traverse the tree in preorder, which will process a node recursively in this order: current, left, right. For the example tree, the nodes will be visited in this order: a, b, c, d, f, g, e.

We let the codeword stack S be empty. Starting in node a, we first check whether the value is not the "null" value. Since it is the "null" value, we will stop processing this node, push a 0 into S and go to node b.

Once we reach node f, S will have the value $S = 100_2$. Since it has a non-"null" value, we save the current value of S and the value in this node to some keyed map (e.g. a hashmap). Then, since f does not have children, we go up the tree by one level and visit the right node. This requires popping off the rightmost digit in S and pushing one to it.

Going through each node in the prescribed visitation order, we get the following progression:

Iteration Index i	**Currently Visited Node**	**Current Codeword Digits in S**	**Current Value in Node**	**Mapped Binary Codeword**
0	a	2	-1	
1	b	2, 0	6	0
2	c	2, 1	-1	
3	d	2, 1, 0	-1	
4	f	2, 1, 0, 0	1	100
5	g	2, 1, 0, 1	5	101
6	e	2, 1. 1	3	11

The value to binary codeword mappings are as follows:

- 1 - 100
- 3 - 11
- 5 - 101
- 6 - 0

Encoding
Encoding a message is now straightforward by just mapping the characters to the corresponding codewords.

As an example, we have the following sequence C:

$$C = \{5, 5, 6, 3, 3, 3, 1, 6, 6\}$$

The value to binary codeword mappings derived from its Huffman tree are as follows:

- 1 - 100
- 3 - 11
- 5 - 101
- 6 - 0

The final encoded sequence of bits corresponding to C is therefore:

101 101 0 11 11 11 100 0 0

Decoding

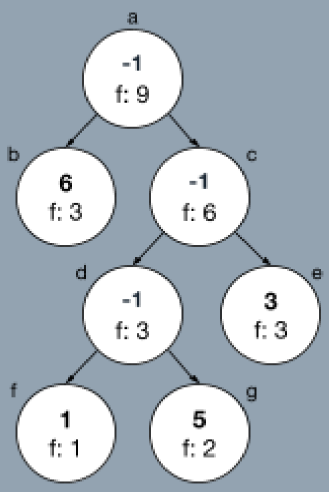
On the other hand, the following algorithm can be used to decode a message encoded using Huffman encoding:

1. Get the tree T associated with encoding the message M.
2. Let t_{idx} be a pointer to the current node in T, and Q be the queue that will contain the decoded message.
3. Let t_{idx} point to the root node of T.
4. Do the following $|M| + 1$ times, with index i:
    a. If t_{idx} points to a leaf node (i.e. has a character):
        i. Push the character onto Q.
        ii. Reset t_{idx} to point it to the root node of T.
    b. Otherwise, if i is less than $|M|$.
        i. Let c be the i_{th} character in M.
        ii. If c is a zero, move t_{idx} to the left.
        iii. Otherwise, if c is a one, move t_{idx} to the right.

As an example, we have the following sequence of bits M encoded using Huffman encoding:

101101011111110000

Additionally, we have reconstructed or received the following Huffman tree corresponding to this sequence of bits with each node labeled with a letter for reference:



Going through each bit in M, we get the following progression:

Iteration Index i	**Current Bit** M_i	**Current Node in** T	**Next Node**	**Value in Next Node**	**Reset Current Node?**	**Reconstructed Message** Q
0	1	a	c	-1	No	
1	0	c	d	-1	No	
2	1	d	g	5	Yes	5
3	1	a	c	-1	No	5
4	0	c	d	-1	No	5
5	1	d	g	5	Yes	5, 5
6	0	a	b	6	Yes	5, 5, 6
7	1	a	c	-1	No	5, 5, 6
8	1	c	e	3	Yes	5, 5, 6, 3
9	1	a	c	-1	No	5, 5, 6, 3
10	1	c	e	3	Yes	5, 5, 6, 3, 3
11	1	a	c	-1	No	5, 5, 6, 3, 3
12	1	c	e	3	Yes	5, 5, 6, 3, 3, 3
13	1	a	c	-1	No	5, 5, 6, 3, 3, 3
14	0	c	d	-1	No	5, 5, 6, 3, 3, 3
15	0	d	f	1	Yes	5, 5, 6, 3, 3, 3, 1
16	0	a	b	6	Yes	5, 5, 6, 3, 3, 3, 1, 6
17	0	a	b	6	Yes	5, 5, 6, 3, 3, 3, 1, 6, 6

We start by letting the current node be the root node a. Since the current node has a "null" value, we then look at the first bit and traverse the tree to the left if the bit is zero, or to the right otherwise.

At iteration 2, we note that the current node has a non-"null" value. So we push this value into Q and reset the current node to the root node a.

The reconstructed sequence is therefore:

$$Q = \{5, 5, 6, 3, 3, 3, 1, 6, 6\}$$

Codebook Canonicalization
The codeword mapping, or codebook, can be stored as a 16-bit pair of binary codeword and character respectively. However, if the alphabet used is known at the decoding side, we can change the binary codewords into something such that we only need to encode the *lengths* of the codewords.

To canonicalize a codebook for such mapping:

1. Sort the mappings by increasing codeword length, and then by ascending mapped values.
2. Let the codeword of the first mapping be equal to a string of zeros of the same length as the codeword. Let this length be equal to L_{canon}.
3. Initialize a value $C_{canon} = 0$ storing the current value of the codeword to assign.
4. For each subsequent mapping in the sorted list $C_{old,i}$:
    a. Increment C_{canon} by one.
    b. Check whether the codeword length of $C_{old,i}$ is greater than L_{canon}.
        i. If it is, then left shift the bits of C_{canon} until C_{canon} now has the same length as $C_{old,i}$. Also set L_{canon} to be equal to the length of $C_{old,i}$.
        ii. Otherwise, replace $C_{old,i}$ with C_{canon} while maintaining the old codeword length of $C_{old,i}$.

As an example, we have the following non-canonical Huffman codebook:

- 1 - 100
- 3 - 11
- 5 - 101
- 6 - 0

We sort the codebook entries to yield:

- 6 - 0
- 3 - 11
- 1 - 100
- 5 - 101

Going through each value in the codebook will yield the following progression:

Value	**Old Codeword**	**Current L_{canon} Length**	**Current C_{canon} Bin Value**	**Left Shift Amount**	**New Codeword**
6	0	1	0	0	0

3	11	1	0	1	10
1	100	2	10	1	110
5	101	3	110	0	111

The new canonical codebook corresponding to the old one is therefore:

- 6 - 0
- 3 - 10
- 1 - 110
- 5 - 111

To encode the codebook for transmission, we first sort the codebook entries by ascending mapped values. Using the canonical codebook derived earlier, the newly-sorted codebook is as follows:

- 1 - 110
- 3 - 10
- 5 - 111
- 6 - 0

Assuming that the original message consists of the mapped values $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the codebook will be transmitted as the following sequence of values:

$$C_{canon,len} = \{0,\ 3,\ 0,\ 2,\ 0,\ 3,\ 1,\ 0,\ 0,\ 0\}$$

This can be interpreted as such: the length of the codeword at the ith element of Σ is the corresponding value at $C_{canon,len}$. So, the first element in Σ has a codeword of length $C_{canon,len}$. Note that the other mapped values that were *not* included in the final codebook are encoded as having a codeword of length zero.

To reconstruct the codebook from the canonical codeword lengths:

1. Obtain the original sorted alphabet Σ used during codebook canonicalization.
2. Create a mapping $\Sigma \rightarrow C_{canon,len}$ of the mapped values and the codeword lengths.
3. Sort the map entries in increasing codeword length, and then by ascending mapped values.
4. Let the codeword of the first mapping be equal to a string of zeros of the specified codeword length. Let this length be equal to L_{canon}.
5. Initialize a value $C_{canon} = 0$ storing the current value of the codeword to assign.

6. For each subsequent mapping in the sorted list $C_{old,i}$:
    a. Increment C_{canon} by one.
    b. Check whether the codeword length of $C_{old,i}$ is greater than L_{canon}.
        i. If it is, then left shift the bits of C_{canon} until C_{canon} now has the same length as $C_{old,i}$. Also set L_{canon} to be equal to the length of $C_{old,i}$.
        ii. Otherwise, replace $C_{old,i}$ with C_{canon} while maintaining the old codeword length of $C_{old,i}$.

As an example, we have the sequence of codeword bit lengths:

$$C_{canon,len} = \{0,\ 3,\ 0,\ 2,\ 0,\ 3,\ 1,\ 0,\ 0,\ 0\}$$

Additionally, we are also given the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ corresponding to these codeword lengths. Mapping these two and removing the values in Σ whose $C_{canon,len}$ lengths are zero yields the following mapping. Note that it is already sorted according to increasing codeword length and then by ascending value:

- 6 - 1
- 3 - 2
- 1 - 3
- 5 - 3

Going through each value in the mapping will yield the following progression:

Value	**Current L_{canon} Length**	**Current C_{canon} Bin Value**	**Left Shift Amount**	**Derived Codeword**
6	1	0	0	0
3	2	0	1	10
1	3	10	1	110
5	3	110	0	111

The reconstructed canonical codebook is therefore:

- 6 - 0
- 3 - 10
- 1 - 110
- 5 - 111

Huffman Encoding in diropqlz

Note that as part of diropqlz compression, the Huffman decoder will receive an array of nonnegative integer values ranging from 0 until 9 (the seven commands of diropql plus two binary digits for the run-length encoding) corresponding to the bit lengths of the canonicalized codebook against some alphabet. Hence, for this purpose, the initial alphabet Σ used should be the integers between 0 and 9 inclusive sorted in ascending order, or $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.