Assignment 1

My Huffman Tree Encoder

Changelog

All important changes to the assignment specification and files will be listed here.

- `[17/06 17:00]` Assignment released
- `[23/06 20:40]` Added a few hints for meeting the time limits

Admin

Marks	contributes 15% towards your final mark (see Assessment section for more details)
Submit	see the Submission section
Deadline	8pm on Monday of Week 7
Late penalty	0.2% of the maximum mark per hour or part thereof deducted from the attained mark, submissions later than 5 days not accepted

Background

What is Encoding?

Encoding is the process of converting data into a different format.

In the real world, data often needs to be converted into different formats for efficient processing, storage or transmission, so there are many examples of encoding:

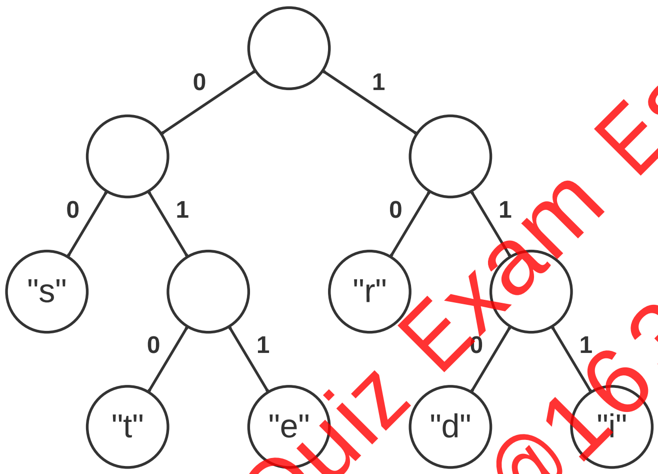
- ASCII encoding, which converts commonly used characters to a 7 bit code
- UTF-8 encoding, which converts a much wider range of characters to a 8-32 bit code
- Base64 encoding, which converts binary data (often images) to Base64 text
- SHA-256 converts a text/byte sequence to a hash which can be used for security purposes

- QR codes, which convert various kinds of information (often a link) to a grid of black/white pixels

The aim of this assignment is to encode and decode text into/from a compressed format using an algorithm known as Huffman coding.

Huffman Coding

Huffman coding is an encoding method used for data compression where each character is encoded as a sequence of bits (0's and 1's). The result of running the algorithm is a Huffman tree, which is used for both encoding and decoding. Here is an example of a Huffman tree:



In a Huffman tree, all leaf nodes (and *only* leaf nodes) contain a character, and the encoding of each character, also known as the *code word* (or simply code) is defined by the path from the root to the character, with left and right corresponding to 0 and 1 respectively. For the above example, this means the code of "s" is "00", the code of "t" is "010", and so on. The text "stirred", which consists of multiple characters, would be encoded as "00010111101001110".

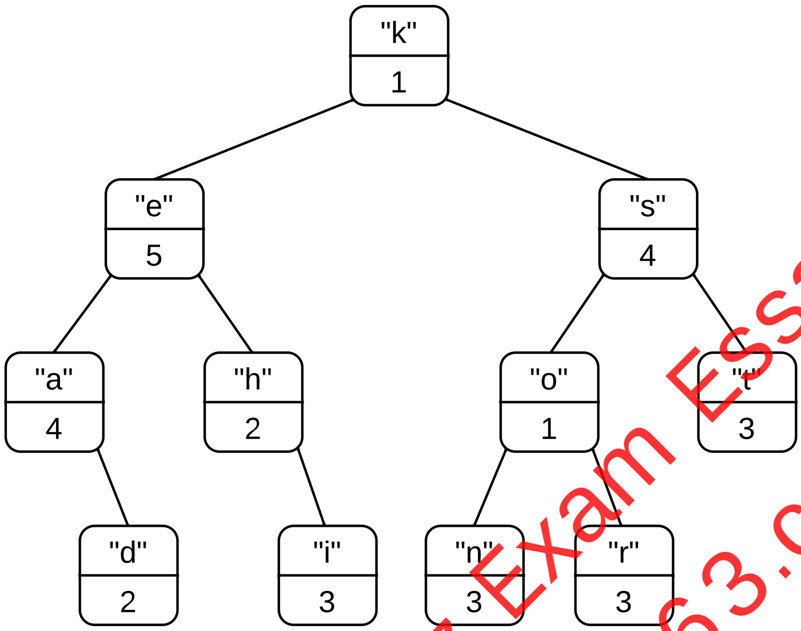
Since only the leaves of a Huffman tree contain characters, a Huffman code is known as a "prefix code", which is a type of code system where no code word is a prefix of another code word. This means that special markers are not required between code words (unlike Morse code, where long pauses are used to separate letters and words), and each encoding can be uniquely decoded – in other words, it is not possible for an encoding to be decoded to two different texts.

What makes Huffman coding particularly notable is that it produces an *optimal* prefix code. This means that out of all possible prefix codes, Huffman coding will always produce the encoding with the shortest length. This is achieved by making sure that characters that occur more frequently in the text will have shorter code words.

Binary Search Tree

A binary search tree is a binary tree which is ordered such that for each node, all items in its left subtree are less than the item in the node, and all items in its right subtree are greater than the item in the node. In the lectures and labs, we have been using binary search trees of integers – however in this assignment, you will be using binary search trees of strings.

You will use a binary search tree to implement a Counter ADT, which keeps track of the frequencies of characters. To achieve this, each node in the binary search tree will contain not just a character and left and right pointers, but an integer which indicates the number of occurrences of the character. For example, consider the following binary search tree of characters:



This shows, for example, that the character "e" occurs 5 times, the characters "a" and "s" occur 4 times, the characters "i", "n", "r" and "t" occur 3 times, and so on.

The World of Characters

So far in your computer science degree, you have likely mainly dealt with ASCII characters. There are 128 ASCII characters, which means an ASCII character can be represented by 7 bits (as $2^7 = 128$). Most computers represent an ASCII character using 1 byte (8 bits), as it is easier for computers to read one byte at a time (as opposed to 7 bits).

In the real world however, there are many more than 128 characters, such as mathematical symbols (e.g., ∀, ∃), Chinese characters (e.g., 日, 月), Greek letters (e.g., Φ, Ψ) and emoji (e.g., 🐶 , 🐱). Unicode, which is a text encoding standard maintained by the Unicode Consortium, defines almost 150,000 characters that are used in various contexts around the world. Since it is clearly not possible to represent all of these characters using just one byte, Unicode characters are typically represented by a series of one to four bytes, depending on the character.

In this assignment, we will ensure that our Huffman coding algorithm is capable of encoding all kinds of text by representing a character as a string of one to four `char`s. This means that ASCII characters will be represented by a string with one `char` (plus the null terminating byte `'\0'`), whereas other characters (such as Greek letters) will be represented by a string of two, three or four `chars` (plus `'\0'`). You will not need to know or learn exactly how these characters are represented, as you will be provided with an ADT to read characters one at a time from a file. When working with these characters, you can treat them as regular C strings.

In this specification and the provided files, the word "character" will always be used to refer to a string of `char`s.

Setting Up

Change into the directory you created for the assignment and run the following command:

```
$ unzip /web/cs2521/24T2/ass/ass1/downloads/files.zip
```

If you're working on your own machine, download `files.zip` by clicking on the above link and then unzip the downloaded file.

You should now have the following files:

`Makefile`	to help you compile your code
`Counter.h`	interface to the Counter ADT
`Counter.c`	implementation of the Counter ADT (incomplete)
`File.h`	interface to the File ADT
`File.c`	implementation of the File ADT (complete)
`huffman.h`	interface to the Huffman module
`huffman.c`	implementation of the Huffman module (incomplete)
`encode.c`	main program for encoding a text (complete)
`decode.c`	main program for decoding an encoded text (complete)
`testCounter.c`	main program for testing the Counter ADT
`task[134]/`	subdirectories containing files for testing tasks 1, 3 and 4
`autotest`	a script that runs provided tests

Note that the only files you are allowed to submit are `Counter.c` and `huffman.c`, so you must write all your code in these files. You shouldn't modify any other files unless you are doing it for testing purposes or if you know what you're doing.

Types of Files

If you look around the provided directories you will find a few different types of files (with different extensions):

`*.txt` files

These files contain regular, human-readable text. You will be encoding these files in the tasks below. It is also very easy to create new files for testing.

`*.enc` files

These files contain encodings of `.txt` files, and consist entirely of the characters `0` and `1`. You will be producing these encodings and decoding them in the tasks below. You must not modify any of these files, otherwise they won't get decoded to the correct text.

Note that in reality, these files would be approximately eight times smaller, because each of the `0`'s and `1`'s would be stored as a single bit. In this assignment, these files contain explicit `0` and `1` characters so you can see the encodings.

`*.tree` files

These files contain Huffman trees in a format that is easy for computers to understand (i.e., they contain serialisations of Huffman trees). The format can be described recursively as follows:

- The serialisation of a leaf node (which will always contain a character) is simply the character in the leaf node. For example, a leaf node which contains the character "a" is serialised as `a`.
- The serialisation of a tree that contains two smaller subtrees is `(`A`,`B`)`, where A is the serialisation of the left subtree, and B is the serialisation of the right subtree.
- The characters `(`, `,`, `)` and `\` are escaped with a `\`. This is so that these characters do not confuse a program that is trying to parse a tree from this format.

As an example, the serialisation of the Huffman tree in the Background section is:

```
((s,(t,e)),(r,(d,i)))
```

This format is very difficult for humans to read, so we have provided a website to help you visualise the Huffman trees stored in a `.tree` file. To use it, click the "Browse..." button and then select a `.tree` file.

Task 1: Decoding

Given a Huffman tree and an encoded text, decoding is very easy, so we've designated this as the first task.

Your task is to implement the following function in `huffman.c`:

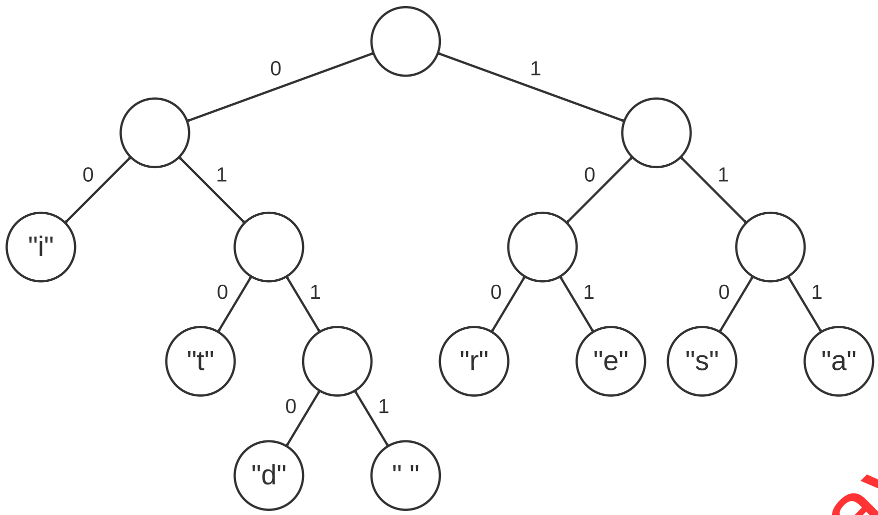
```
void decode(struct huffmanTree *tree, char *encoding, char *outputFilename);
```

This function takes a Huffman tree and an encoding, and writes the decoded text to a file with the given name. The encoding is a string consisting entirely of the characters `'0'` and `'1'`. You must use the File ADT to open and write to the file.

Note that `struct huffmanTree` contains a `freq` field, but this field is not used during decoding, so you should simply ignore it.

Example

Consider the following Huffman tree:



Here are some example encodings and what they would be decoded to:

Encoding	Decoded text
`101111010`	`eat`
`010100101101110`	`trees`
`1101011110111010000110101`	`sea tide`
`01100010010101111100101001110001011 0`	`dire straits`

Testing

The `task1/` directory contains pairs of `.tree` and `.enc` files, and a subdirectory `expected_texts/` which contain the expected texts for each of the pairs of files.

To test your `decode` function, compile your program with `make` and then run the `decode` program, passing it a `.tree` file, the matching `.enc` file and a filename for the output to go to. For example:

```
$ ./decode task1/sea_shells.tree task1/sea_shells.enc task1/sea_shells.txt
```

If you've implemented the function correctly, then the text in the output file you created (e.g., `task1/sea_shells.txt`) should match the expected text in the `expected_texts/` directory (e.g., `task1/expected_texts/sea_shells.txt`). You can check that the files are the same using the `diff` command:

```
$ diff task1/sea_shells.txt task1/expected_texts/sea_shells.txt
```

If the files are identical, the `diff` command will produce no output.

Note that you should only run the `./decode` program on matching `.tree` and `.enc` files. If you run the program on non-matching `.tree` and `.enc` files then either the program will encounter an error, or you will get the wrong output.

You can run all provided tests using the command:

```
$ ./autotest 1
```

Time Limits

When we test your decode function, we will impose a 5 second user-time limit on each run of the decode program. The program will be compiled using make asan. The largest .enc file we will use is the provided war_and_peace.enc file.

Hint: If your program is running too slowly, one possible reason is your use of string library functions such as strlen and strcat. Think carefully about how these functions work.

Task 2: Counter ADT

Encoding is much more difficult than decoding, so we've divided it into three tasks.

The first step in encoding a text is to count the occurrences of each character. Therefore, your task is to implement a Counter ADT which records the frequency of characters **using a binary search tree**.

Remember that in this assignment, a character is represented by a string of one to four chars. If you are confused by this, then you should go back and read the Background section.

Here are the operations of the Counter ADT:

Operation	Description
CounterNew	Returns a new empty counter
CounterFree	Frees all memory allocated to the given counter
CounterAdd	Adds an occurrence of the given character to the counter
CounterNumItems	Returns the number of distinct characters added to the counter
CounterGet	Returns the frequency of the given character
CounterItems	Returns a dynamically allocated array containing a copy of each distinct character in the counter and its count (in any order), and sets *numItems to the number of distinct characters.

Constraints

You must implement the Counter ADT using a binary search tree. You will not receive marks for this task if you use any other data structure.

The binary search tree does not need to be balanced.

Testing

We have provided a program testCounter.c for you to test your implementation of the Counter ADT.

To run the tests, compile with make and then run ./testCounter. The tests are assert-based, which means the program will terminate with an error message if a test fails. If your implementation passes all the tests, it will simply output:

```
$ ./testCounter
Test 1 passed!
Test 2 passed!
Test 3 passed!
```

The program contains only three simple tests, so it is recommended that you add your own tests. You can add new tests by creating a new test function (e.g., `test4()`), and calling it from the `main` function.

Efficiency

To receive full marks for this task, all Counter ADT functions must be $O(n)$ or faster, where n is the number of distinct items that have been added to the counter.

Task 3: Constructing a Huffman Tree

After you've recorded the frequencies of all characters, the next step is to construct the Huffman tree.

Your task is to implement the following function in `huffman.c`:

```c
struct huffmanTree *createHuffmanTree(char *inputFilename);
```

This function takes the name of a text file and constructs a Huffman tree from the characters in the file. You must use the File ADT to read characters from the file, and use the Counter ADT to count the number of occurrences of each character.

After you have counted all the characters, you must follow this algorithm to construct the Huffman tree:

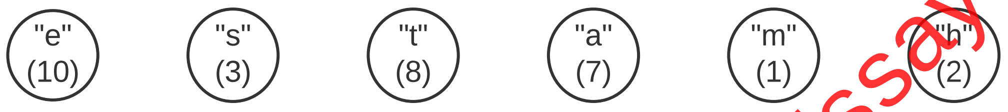
- For each character, create a leaf node that contains the character and its frequency. This creates n Huffman trees (each consisting of one node) where n is the number of characters.
- Until there is only one Huffman tree remaining:
  - Take the two Huffman trees with the smallest total frequency
    - If there are multiple possibilities for which pair of trees to take, you may select any of these pairs
  - Combine the two trees, i.e., create a new node that has the two trees as subtrees, with frequency equal to the sum of their frequencies. The `character` field of the new node should be set to `NULL`, as only leaf nodes contain characters.

Example

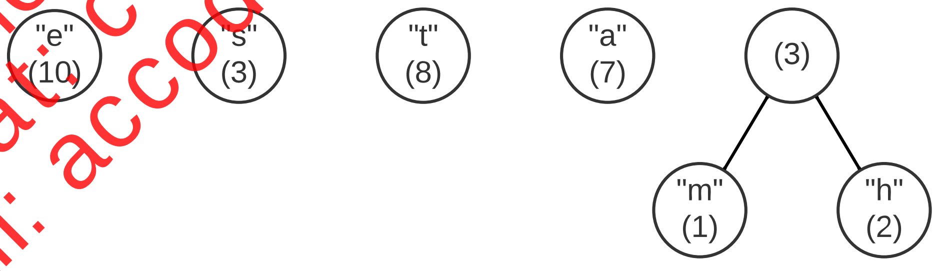
Suppose you have processed all the characters in the text file and these are their frequencies:

Character	Frequency
"e"	10
"s"	3
"t"	8
"a"	7
"m"	1
"h"	2

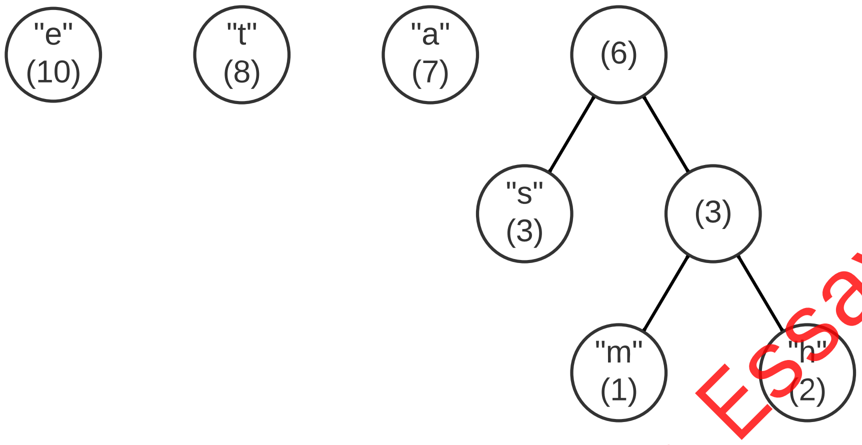
The following images demonstrate the process of constructing the Huffman tree (click the arrows on the sides to navigate between steps):



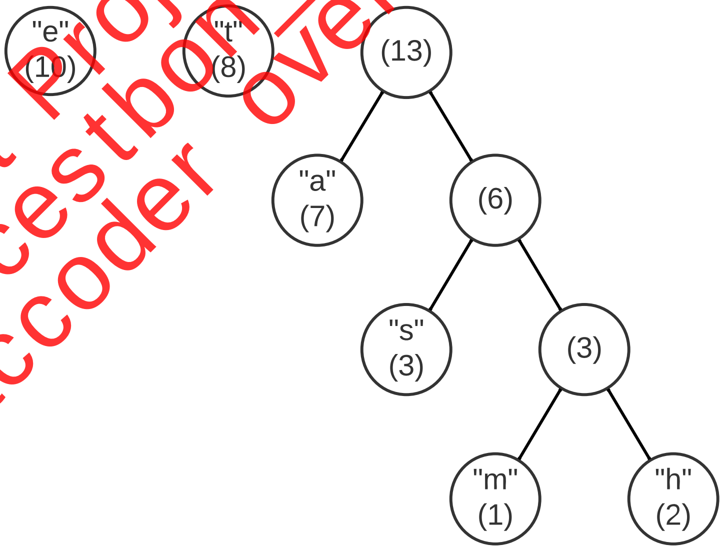
Initially, there are n Huffman trees.



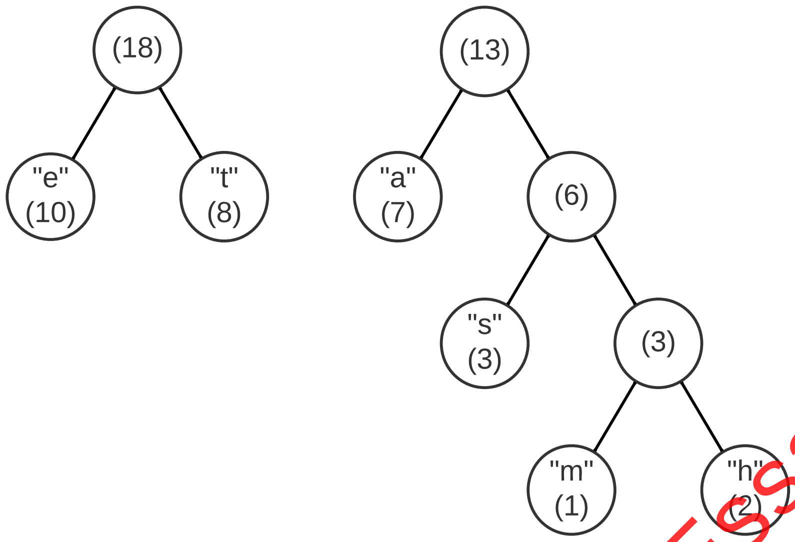
The two Huffman trees with the lowest frequencies (1 and 2) are merged into a new Huffman tree with frequency 3.



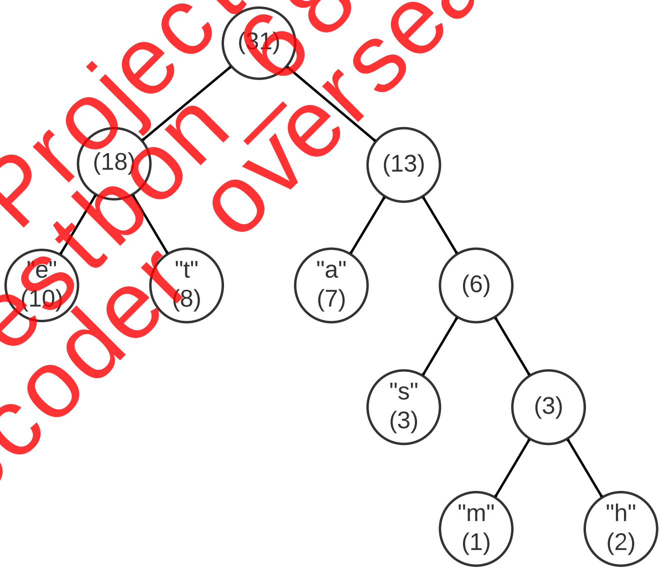
The two Huffman trees with the lowest frequencies (3 and 3) are merged into a new Huffman tree with frequency 6.



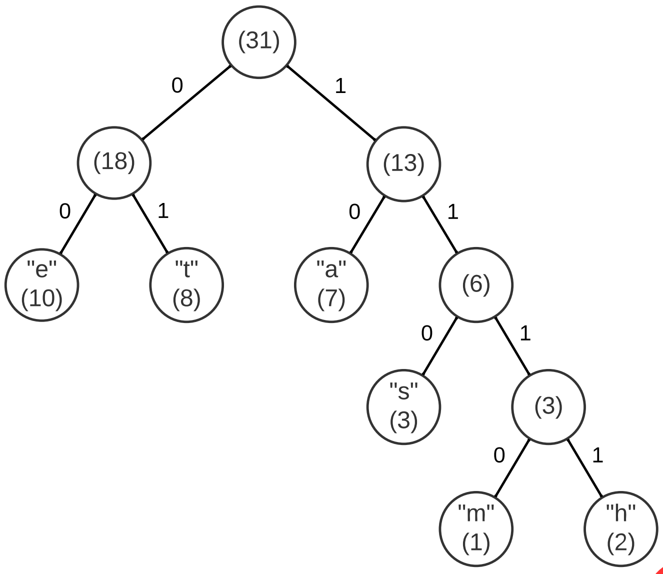
The two Huffman trees with the lowest frequencies (6 and 7) are merged into a new Huffman tree with frequency 13.



The two Huffman trees with the lowest frequencies (8 and 10) are merged into a new Huffman tree with frequency 18.



The two Huffman trees with the lowest frequencies (13 and 18) are merged into a new Huffman tree with frequency 31.



The Huffman tree is complete.

Previous Next

Notes

You can assume that the text file contains at least two distinct characters.

Note that when you are choosing two subtrees to combine, if there are multiple eligible pairs, then you may choose any of them. When combining the two subtrees, it does not matter which one goes on the left and which one goes on the right. Because of this, it is possible to construct many different correct Huffman trees from the same text.

Testing

The `task3/` directory contains some text files, and a subdirectory `sample_trees/` which contains *one possible Huffman tree* for each of the text files. Note that there are multiple possible Huffman trees for each text file, so the trees you obtain may be different from the given sample trees.

To test your `createHuffmanTree` function, compile with `make` and then run the `./encode` program with two command-line arguments: the name of the text file, and the name of a new `.tree` file that you want your Huffman tree to be printed to. For example:

```
$ ./encode task3/sea_shells.txt task3/sea_shells.tree
```

You may want to use the Huffman Tree Viewer to visualise your Huffman tree so that you can more easily verify whether it is correct.

Reference Program

Note that even though multiple Huffman trees are possible, all correct Huffman trees will result in encodings of the same length (*for the file that they were created from*), which also happens to be the minimal encoding length. To help you check whether your function has produced a correct Huffman tree, we have provided the following reference program:

```
/web/cs2521/24T2/ass/ass1/references/encodingLength
```

The reference program behaves as follows:

- If the program is run with one command-line argument, a valid text file, then the program will output the encoding length that would result from a correct Huffman tree. For example:

  ```
  $ /web/cs2521/24T2/ass/ass1/references/encodingLength task3/sea_shells.txt
  Minimal encoding length for task3/sea_shells.txt: 112
  ```

- If the program is run with two command-line arguments, a valid text file and a matching `.tree` file produced by your `encode` program, then the program will output the encoding length that would result from using the given Huffman tree. For example:

  ```
  $ /web/cs2521/24T2/ass/ass1/references/encodingLength task3/sea_shells.txt
  task3/sea_shells.tree
  Encoding length: 112
  ```

  If the encoding length shown here is the same as the minimal encoding length shown above, then this means your program has produced a correct Huffman tree.

You can run all provided tests using the command:

```
$ ./autotest 3
```

Time Limits

When we test your `createHuffmanTree` function, we will impose a 5 second user-time limit on each run of the `encode` program. The program will be compiled using `make asan`. The largest `.txt` file we will use is the provided `war_and_peace.txt` file.

Task 4: Encoding

The final step is to use the Huffman tree to encode the text.

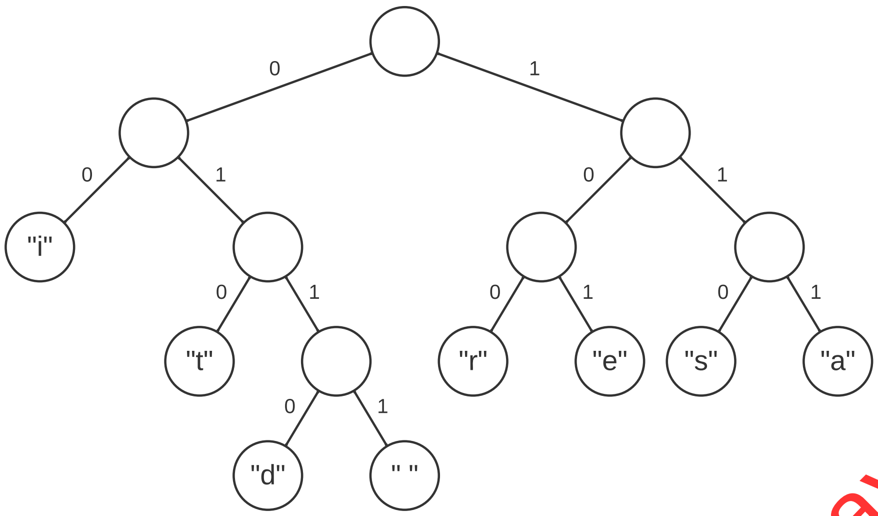
Your task is to implement the following function in `huffman.c`:

```
char *encode(struct huffmanTree *tree, char *inputFilename);
```

This function takes a Huffman tree and the name of a text file, and encodes the text file. It should return a string containing the encoding of the file, and the string should consist entirely of the characters `'0'` and `'1'`. You must use the File ADT to read characters from the file.

Example

Encoding is just the opposite of decoding. Consider the following Huffman tree (same as above):



Here are some examples of texts and their encodings:

Text	Encoding
`tires`	`01000100101110`
`reader`	`1001011110110101100`
`iterate`	`00010101100111010101`
`red star`	`1001010110011111001011100`

Testing

The `task4/` directory contains some pairs of `.txt` and `.tree` files, and a subdirectory `expected_encodings/` which contain the expected encoding for each of the pairs of files.

To test your `encode` function, compile with `make` and then run the `./encode` program with **three** command-line arguments: the name of the text file, the name of the matching `.tree` file, and the name of a new file where you want the encoding to be printed to. For example:

```
$ ./encode task4/sea_shells.txt task4/sea_shells.tree task4/sea_shells.enc
```

Then use the `diff` command to compare the encoding your function produced with the expected encoding in the `expected_encodings/` directory. For example:

```
$ diff task4/sea_shells.enc task4/expected_encodings/sea_shells.enc
```

If the files are identical, the `diff` command will produce no output.

You can run all provided tests using the command:

```
$ ./autotest 4
```

Time Limits

When we test your `encode` function, we will impose a 5 second user-time limit on each run of the `encode` program. The program will be compiled using `make asan`. The largest text file we will use is the provided `war_and_peace.txt` file.

Hint: If your program is running too slowly, one possible reason is your use of string library functions such as `strlen` and `strcat`. Think carefully about how these functions work.

Frequently Asked Questions

- **Are we allowed to create our own functions?** You are always allowed to create your own functions. All additional functions you create should be made `static`.
- **Are we allowed to create our own `#defines` and structs?** Yes.
- **Are we allowed to `#include` any other libraries?** No. All the libraries required for this assignment are provided already.
- **Are we allowed to change the signatures of the given functions?** No. If you change these, your code won't compile and we won't be able to test it.
- **What errors do we need to handle?** You should handle common errors such as `NULL` returned from `malloc` by printing an error message to `stderr` and terminating the program. You are not required to handle other errors.
- **What invalid inputs do we need to handle?** You are not required to handle invalid inputs, such as `NULL` being passed in as a counter. It is the user's responsibility to provide valid inputs.
- **Will we be assessed on our tests?** No. You will not be submitting any test files, and therefore you will not be assessed on your tests.
- **Are we allowed to share tests?** No. Testing is an important part of software development. Students that test their code more will likely have more correct code, so to ensure fairness, each student should independently develop their own tests.

Submission

You must submit the files `Counter.c` and `huffman.c`. You can submit via the command line using the `give` command:

```
$ give cs2521 ass1 Counter.c huffman.c
```

You can also submit via give's web interface. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

WARNING:

After you submit, you must check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Compilation

You must ensure that your final submission compiles on CSE machines. Submitting non-compiling code leads to extra administrative overhead and will result in a 10% penalty.

Every time you make a submission, a dryrun test will be run on your code to check that it compiles. Please ensure that your final submission successfully compiles, even for parts that you have not completed.

Assessment Criteria

This assignment will contribute 15% to your final mark.

Correctness

80% of the marks for this assignment will be based on the correctness of your code, and will be mostly based on autotesting. Marks for correctness will be distributed as follows:

Task	Weighting
Task 1 – Decoding	10%
Task 2 – Counter ADT	25%
Task 3 – Constructing a Huffman Tree	25%
Task 4 – Encoding	20%

Time limits

There will be a 5 second user-time limit applied to **each test**. This means each time the `decode`, `encode` or `testCounter` program is run, a 5 second user-time limit will be applied to it. The programs will be compiled with the same flags as `make asan`.

The largest `.enc` file that will be used for testing task 1 is `war_and_peace.enc`. The largest text file that will be used for testing tasks 3 and 4 is `war_and_peace.txt`. For task 2, a Counter ADT that is implemented sensibly with no function slower than $O(n)$ will run in time.

Memory errors/leaks

You must ensure that your code does not contain memory errors or leaks, as code that contains memory errors is unsafe and it is bad practice to leave memory leaks in your program.

Our tests will include memory error/leak test(s) for each task. If a memory error/leak arises from your code, you will receive a penalty which is 10% of the marks for that task. For example, the penalty for having a memory error/leak in the Counter ADT will be 2.5%. Note that our tests will always free memory as necessary at the end of the test.

Style

20% of the marks for this assignment will come from hand marking of the readability of the code you have written.
These marks will be awarded on the basis of clarity, commenting, elegance and style. The following is an indicative list of characteristics that will be assessed, though your program will be assessed wholistically so other characteristics may be assessed too (see the style guide for more details):

- Consistent and sensible indentation and spacing
- Using blank lines and whitespace
- Using functions to avoid repeating code
- Decomposing code into functions and not having overly long functions
- Using comments effectively and not leaving planning or debugging comments

The course staff may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.

Originality of Work

This is an individual assignment. The work you submit must be your own work and only your work apart from the exceptions below. Joint work is not permitted. At no point should a student read or have a copy of another student's assignment code.

You may use any amount of code from the lectures and labs of the **current iteration** of this course. You must clearly attribute the source of this code in a comment. Please see the referencing guide.

You may use small amounts (< 10 lines) of general purpose code (not specific to the assignment) obtained from sites such as Stack Overflow or other publicly available resources. You must clearly attribute the source of this code in a comment.

You are not permitted to request help with the assignment apart from in the course forum, help sessions or from course lecturers or tutors.

Do not provide or show your assignment work to any other person (including by posting it publicly on the forum) apart from the teaching staff of COMP2521. When posting on the course forum, teaching staff will be able to view the assignment code you have recently submitted with give.

The work you submit must otherwise be entirely your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such issues.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, then you may be penalised, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.