C Practice Test
Huffman coding

Tuesday 10th June 2014, 14:00

THREE HOURS

(including 15 minutes planning time)

- Please make your swipe card visible on your desk.
- After the planning time log in using your username as **both** your username and password.

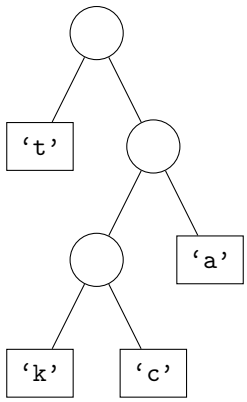
The maximum mark is 40.

Credit will be awarded throughout for clarity, conciseness and *useful* commenting.

Important note: FOUR MARKS will be deducted from solutions that do not compile. Comment out any code that does not compile before you leave.

1 Huffman coding

The standard ASCII code uses 7 bits to encode each character, so a string with 6 characters requires a minimum of 42 bits. Huffman coding uses variable length codes in order to reduce the number of bits required to represent a string. The code is defined by a binary tree with characters at its leaves, called a Huffman tree. The code for a particular character, 'c' say, is a path through the tree from the root to the leaf containing 'c'. For example, using the following Huffman coding tree:



the string "attack" would be coded as:

`RRLLRRRLRRLL`

where `L` and `R` respectively mean "go left" and "go right" through the tree. A Huffman tree is generated from a given *source* string and is designed so that the most frequently occurring characters in the string appear near the top of the tree so the paths to these leaves are as short as possible. In this exercise we will represent a Huffman tree with the following C structure:

```
typedef struct huffman_tree {
  int count;
  char letter;
  struct huffman_tree *left, *right;
} huffman_tree_t;
```

There are two types of node stored in a Huffman tree: *leaf nodes* store a letter and a frequency count while *internal nodes* store a left and right subtree and a count containing the sum of the frequency counts stored in the two subtrees. For example, the Huffman tree corresponding to the source string "aatatttactaccattatktk" (9 't's, 7 'a's, 3'c's and 2 'k's) is given in Figure 1.

2 Building a Huffman tree

The algorithm you are going to use for building a Huffman tree requires the construction of a *list* of Huffman trees which are subsequently reduced into a single tree. We will define such a list thus:

```
typedef struct huffman_tree_list {
  huffman_tree_t *tree;
  struct huffman_tree_list *next;
} huffman_tree_list_t;
```

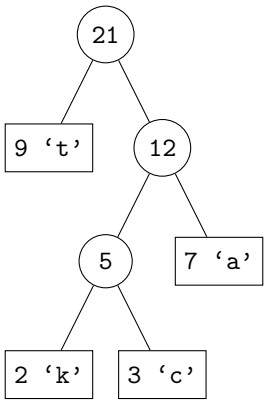


Figure 1: The Huffman tree corresponding to the source string “aatatttactaccattatktk”.

The process for constructing a Huffman tree then proceeds as follows:

1. A source string containing the complete set of characters to be used in a given Huffman code is accepted. Note that any future encoding and decoding is restricted to the characters present in the source string.

2. A table of unique characters is determined by removing all duplicates from the source string.

3. A list of leaf nodes is constructed from the table of unique characters and the source string. The list *must* be ordered such that the frequency counts of each leaf node are in ascending order. As an example, the input “aatatttactaccattatktk” would produce the following list of leaf nodes:

   ```
   [{2, ‘k’, {}, {}}, {3, ‘c’, {}, {}}, {7, ‘a’, {}, {}}, {9, ‘t’, {}, {}}]
   ```

   where the expression `{2, ‘k’, {}, {}}` denotes a Huffman tree encoding the character ‘k’ with frequency count 2. Since the node is a leaf, its subtrees are both empty, represented by the expression {}.

4. The list of Huffman trees is reduced to a single tree as follows:

   (a) The two trees at the head of the list (here `{2, ‘k’, {}, {}}` and `{3, ‘c’, {}, {}}`) are *removed* from the list and *combined* into a single tree:

   ```
   {5, ‘’, {2, ‘k’, {}, {}}, {3, ‘c’, {}, {}}}
   ```

   Since this new node is an internal node and not a leaf, its frequency count is the sum of its children’s frequency counts and its character value is undefined.

   (b) The merged tree is added to the list *in the correct position*, i.e., the resulting list’s trees are still in ascending order of frequency count:

   ```
   [{5, ‘’, {2, ‘k’, {}, {}}, {3, ‘c’, {}, {}}},
     {7, ‘a’, {}, {}}, {9, ‘t’, {}, {}}]
   ```

(c) Since the resulting list still contains more than one tree, the above step is repeated to yield the list:

```
[{9, 't', {}, {}}, {12, '', {5, '', {2, 'k', {}, {}},
  {3, 'c', {}, {}}}, {7, 'a', {}, {}}}]
```

Note that the merged tree's frequency count is now higher than that of the tree {9, 't', {}, {}}; it is thus positioned at the end of the list.

(d) Applying the process once more produces the singleton list:

```
[{21, '', {9, 't', {}, {}}, {12, '', {5, '', {2, 'k', {}, {}},
  {3, 'c', {}, {}}}, {7, 'a', {}, {}}}}]
```

which is the Huffman tree shown in Figure 1. An illustration of these steps is given in Figure 2.

3 What to do

You are provided with a main program file `main.c`, a header file `exam.h` and a program file `exam.c`. The `exam.c` file is where you will write your code while the file `main.c` contains a definition of a `main` function which will call the functions you are required to complete during this test. At present these functions are nothing more than stubs; you should fill out each as you write and test your functions.

The file `exam.h` contains prototypes of all the functions you are expected to write, as well as some functions that are provided for you to use. In particular, the functions `print_huffman_tree`, `print_huffman_tree_list` and `print_huffman_tree_codes` have been supplied to aid you in developing and testing your code. Do *not* modify the definitions in `exam.h` as this will impede our ability to evaluate and test your code.

You may assume that all inputs to the program are valid and, for example, that any text to be encoded or decoded contains characters present in the supplied Huffman tree. For example, given the input string "aatatttactaccattatktk" your program should print out something like:

```
Please enter a string for processing: aatatttactaccattatktk
Derived lookup table: catk

Huffman tree list:
Huffman tree:
  Leaf: 'k' with count 2
Huffman tree:
  Leaf: 'c' with count 3
Huffman tree:
  Leaf: 'a' with count 7
Huffman tree:
  Leaf: 't' with count 9

Huffman tree:
  Node: accumulated count 21
    Leaf: 't' with count 9
```

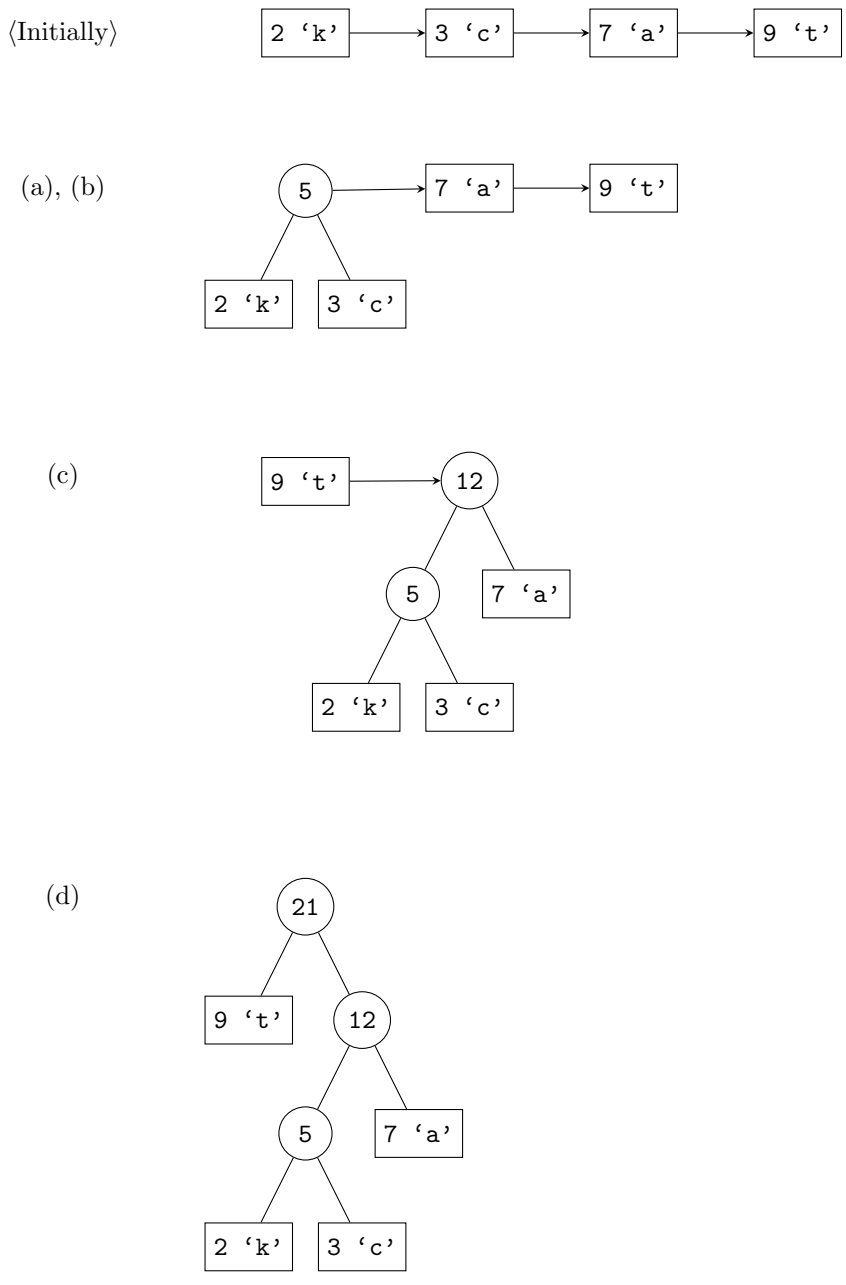


Figure 2: A reduction sequence for constructing the Huffman tree shown in Figure 1.

```
    Node: accumulated count 12
      Node: accumulated count 5
        Leaf: 'k' with count 2
        Leaf: 'c' with count 3
      Leaf: 'a' with count 7

Huffman tree codes:
't' has code "L"
'k' has code "RLL"
'c' has code "RLR"
'a' has code "RR"
```

Your program should use the structures, constants and function templates provided in the header file `exam.h`. You are of course free to define and use auxiliary functions but these should not be visible outside the scope of `exam.c`. You may compile your code using the `Makefile` supplied. By default your code will be compiled with debugging symbols (via `gcc`'s `-g` option) so you will be able to use `gdb` to debug your answers should you wish.

Part I – Core functions

1. Write a function:

   ```
   int contains(char *s, char c);
   ```

   which returns 1 if the character `c` is present in the string `s` and 0 if not.

   [3 marks]

2. Write a function:

   ```
   int frequency(char *s, char c);
   ```

   which returns the number of occurrences of the character `c` in the string `s`.

   [3 marks]

3. Write a function:

   ```
   char *nub(char *s);
   ```

   which given a string `s` produces a string containing only the unique characters of `s` (c.f. Haskell's `nub` function). Use the `contains` function as an aid. You should allocate the result on the heap, and may assume that any string will have fewer than or equal to `MAX_STRING_LENGTH - 1` characters.

   [7 marks]

4. Write a function:

   ```
   huffman_tree_list_t *huffman_tree_list_add(huffman_tree_list_t *l,
                                              huffman_tree_t *t);
   ```

which adds the Huffman tree `t` to the list `l`. A pre- and post-condition is that the list `l` is ordered according to the frequency counts of the trees it contains. The return value is the modified list.

[14 marks]

5. Write a function:

```
huffman_tree_list_t *huffman_tree_list_build(char *s, char *t);
```

which given two strings, `s` and `t`, returns a list of Huffman trees containing leaf nodes for all the characters contained in `t`. The leaf nodes' frequency counts will be derived from the string `s`. You should use the `huffman_tree_list_add` and `frequency` functions to do this. A pre-condition is that `t` is a duplicate-free version of `s` (such as that which could be produced by `nub`, for example).

[7 marks]

6. Write a function:

```
huffman_tree_list_t *huffman_tree_list_reduce(huffman_tree_list_t *l);
```

which reduces a list of Huffman trees to a single, correctly formed tree according to the rules outlined in Section 2. A pre-condition is that the list `l` is non-empty and sorted according to the frequency counts of the trees it contains. Correspondingly the post-condition is that the resulting lists contains a single, correctly-formed tree. This recursive function should make use of the `huffman_tree_list_add` function. **Hint**: When removing the first two elements of the list `l`, consider that you only need free the memory of the list elements, since the trees themselves will still be referenced as the subtrees of the new element.

[6 marks]

Part II – Encoding and decoding (bonus marks)

1. Write a pair of functions:

```
char *huffman_tree_encode(huffman_tree_ *t, char *s);
char *huffman_tree_decode(huffman_tree_t *t, char *code);
```

which encode or decode a value according to a supplied Huffman tree. Both functions should allocate their results on the heap. Encoded strings should consist of only the characters `'L'` and `'R'` as outlined in the introduction, and may be assumed to consist of no more than `MAX_CODE_LENGTH - 1` characters.

[4 marks]