15-122: Principles of Imperative Computation, Fall 2010
Assignment 6: Trees and Secret Codes

Karl Naden (kbn@cs)

Out: Tuesday, March 22, 2010
Due (Written): Tuesday, March 29, 2011 (before lecture)
Due (Programming): Tuesday, March 29, 2011 (11:59 pm)

1 Written: (25 points)

The written portion of this week's homework will give you some practice reasoning about variants of binary search trees. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

1.1 Heaps and BSTs

Exercise 1 (3 pts)**.** Draw a tree that matches each of the following descriptions.

a) Draw a heap that is not a BST.

b) Draw a BST that is not a heap.

c) Draw a non-empty BST that is also a heap.

1.2 Binary Search Trees

Refer to the binary search tree code posted on the course website for Lecture 17 if you need to remind yourself how BSTs work.

Exercise 2 (4 pts)**.** The height of a binary search tree (or any binary tree for that matter) is the number of levels it has. (An empty binary tree has height 0.)

(a) Write a function `bst_height` that returns the height of a binary search tree. You will need a wrapper function with the following specification:

```
int bst_height(bst B);
```

This function should not be recursive, but it will require a helper function that will be recursive. See the BST code for examples of wrapper functions and recursive helper functions.

(b) Why is recursion preferred over iteration for this problem?

Exercise 3 (5 pts)**.** In this exercise you will re-implement `bst_search` in an iterative fashion.

a) Complete the `bst_search` function below that uses iteration instead of recursion.

```
elem bst_search(bst B, key k)
//@requires is_bst(B);
//@ensures \result == NULL || compare(k, elem_key(\result)) == 0;
{
    tree T = B->root;
    while (_____________________________________________)
    {
        if (_______________________________)
            T = T->left;
        else
            T = T->right;
    }
    if (T == NULL) return NULL;
    return T->data;
}
```

b) Using your loop condition, prove that the `ensures` clause of `bst_search` must hold when it returns.

1.3 Rotations and AVL Trees

Exercise 4 (3 pts)**.** Consider the following function to perform a rotate right operation:

```
tree tree_rotate_right(tree T) {
    tree newRight = alloc(Struct tree);
    newRight-> data = T->data;
    newRight->right = T->right;
    newRight->left = T->left->right;

    tree newRoot = alloc(struct tree);
```

```
    newRoot->data = T->left->data;
    newRoot->left = T->left->left;
    newRoot->right = newRight;

    return newRoot;
}
```

Give `requires` and `ensures` annotations for the `tree_rotate_right` function. Your annotations should include tests on the ordering of the tree and the height of the tree, before and after the rotation. You may assume the following functions are defined for you:

```
bool is_ordered(tree T, elem min, elem max);
int height(tree T);
```

Exercise 5 (3 pts)**.** Draw the AVL tree that results after successively inserting the following keys (in the order shown) into an initially empty tree, rebalancing if necessary after every insert.

```
7  12  10  16  4  2
```

Your answer should show the tree after each key is successfully inserted. Also be sure to label each node with its *balance factor*, which is defined as the height of the right subtree minus the height of the left subtree.

Exercise 6 (5 pts)**.** During lecture 18, we showed that each rotation of a binary tree takes constant time. Since the height of an AVL tree is guaranteed to be $O(\log n)$ and the balance invariant can only be broken along the path from the location a new element was inserted to the root, we concluded that to restore the balance invariant we will need to do at most $O(\log n)$ rotations. However, it is an important property of AVL trees that restoring the balance invariant is achieved in one (double or single) rotation. It takes $O(\log n)$ time to find the place in the tree to insert the new element, so the `insert` operation is still $O(\log n)$ overall, but fewer rotations mean a smaller constant factor of overhead, which is an important practical consideration.

In this exercise we will go through several steps to prove that we only need to perform at most one rotation following an insertion into an AVL tree. The basic idea is to show that any rotation of an unbalanced tree created from an insertion of a new element into a balanced AVL tree must reduce the height of the rotated tree. In lecture, we considered three cases after insertion on the left. In two of these (not shown here), the height of the tree is reduced by one during the rotation. The third has the form show in Figure 1.

a) Draw the result of rotating the tree in Figure 1 to the right and explain why the overall height of the tree is not reduced.

b) Explain why the case depicted above could not occur after the insertion of a single element into a balanced AVL tree prior to any rebalancing.

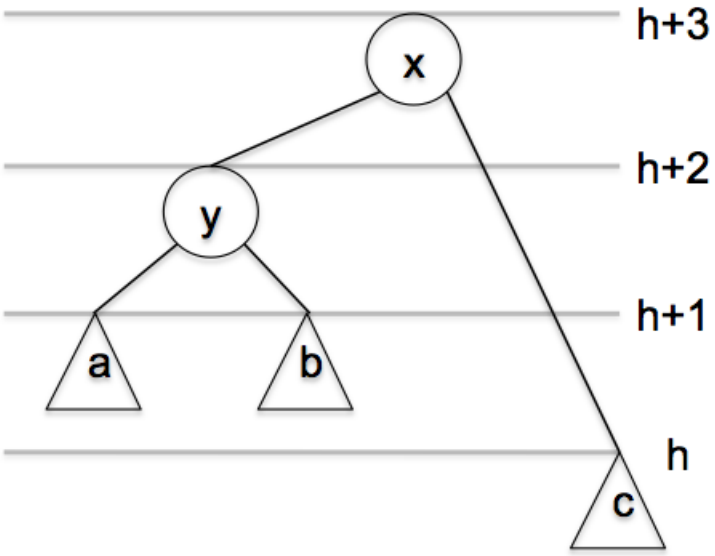

Figure 1: An unbalanced AVL tree

c) Given that the other rotations reduce the height of the tree, argue that the parent of the rotated tree cannot be unbalanced. (**Hint:** Think about how the height of the subtree has changed between during the insertion and rotation)

1.4 Testing

Exercise 7 (2 pts)**.** Identify the primary difference between black box and glass box testing in terms of what information is used to generate tests. How does this difference impact what can be effectively tested with each approach?

2 Programming: Huffman Coding (25 points)

For the programming portion of this week's homework, you'll write a C_0 implementation of a popular compression technique called Huffman coding. In addition, you will be asked to generate a suite of examples using ideas from testing methodologies presented in lecture 19.

You should submit your code electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Starter code. Download the file `hw6-starter.zip` from the course website. It contains some code for reading frequency tables (`freqtable.c0`), described below, as well as some sample inputs.

Compiling and running. For this homework, you will use the standard `cc0` command with command line options. Don't forget to include the `-d` switch if you'd like to enable dynamic annotation checking.

For details on how we will compile your code, see the file `COMPILING.txt` included in the starter code. **Warning:** *You will lose credit if your code does not compile.*

Submitting. Once you've completed some files, you can submit them by running the command

```
handin -a hw6 <file1>.c0 ... <fileN>.c0
```

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

Character	Code
`'c'`	000
`'e'`	001
`'f'`	010
`'m'`	011
`'o'`	100
`'r'`	101

Figure 2: A custom fixed-length encoding for the non-whitespace characters in the string "`more free coffee`".

2.1 Data Compression Primer

Whenever we represent data in a computer, we have to choose some sort of *encoding* with which to represent it. When representing strings in C_0, for instance, we use ASCII codes to represent the individual characters. Under the ASCII encoding, each character is represented using 7 bits, so a string of length n requires $7n$ bits of storage, which we usually round up to $8n$ bits or n bytes. Other encodings are possible as well, though. The UNICODE standard defines a variety of character encodings with a variety of different properties. The simplest, UTF-32, uses 32 bits per character.

Sometimes when we wish to store a large quantity of data, or to transmit a large quantity of data over a slow network link, it may be advantageous to seek a specialized encoding that requires less space to represent our data by taking advantage of redundancies inherent in it. For example, consider the string "`more free coffee`"; ignoring spaces, it can be represented in ASCII as follows with $14 \times 7 = 98$ bits:

$$1101101 \cdot 1101111 \cdot 1110010 \cdot 1100101 \cdot 1100110 \cdot$$
$$1110010 \cdot 1100101 \cdot 1100101 \cdot 1100011 \cdot 1101111 \cdot$$
$$1100110 \cdot 1100110 \cdot 1100101 \cdot 1100101$$

This encoding of the string is rather wasteful, though. In fact, since there are only 6 distinct characters in the string, we should be able to represent it using a custom encoding that uses only $\lceil \lg 6 \rceil = 3$ bits to encode each character. If we were to use the custom encoding shown in Figure 2, the string would be represented with only $14 \times 3 = 42$ bits:

$$011 \cdot 100 \cdot 101 \cdot 001 \cdot 010 \cdot$$
$$101 \cdot 001 \cdot 001 \cdot 000 \cdot 100 \cdot$$
$$010 \cdot 010 \cdot 001 \cdot 001$$

In both cases, we may of course omit the separator "$\cdot$" between codes; they are included only for readability.

If we confine ourselves to representing each character using the same number of bits, i.e., a *fixed-length encoding*, then this is the best we can do. But if we allow

Character	Code
`'e'`	0
`'o'`	100
`'m'`	1010
`'c'`	1011
`'r'`	110
`'f'`	111

Figure 3: A custom variable-length encoding for the non-whitespace characters in the string `"more free coffee"`.

ourselves a *variable-length encoding*, then we can take advantage of special properties of the data: for instance, in the sample string, the character `'e'` occurs very frequently while the characters `'c'` and `'m'` occur very infrequently, so it would be worthwhile to use a smaller bit pattern to encode the character `'e'` even at the expense of having to use longer bit patterns to encode `'c'` and `'m'`. The encoding shown in Figure 3 employs such a strategy, and using it, the sample string can be represented with only 34 bits:

$$1010 \cdot 100 \cdot 110 \cdot 0 \cdot 111 \cdot$$
$$110 \cdot 0 \cdot 0 \cdot 1011 \cdot 100 \cdot$$
$$111 \cdot 111 \cdot 0 \cdot 0$$

Since this encoding is *prefix-free*—no code word is a prefix of any other code word—the "$\cdot$" separators are redundant here, too.

It can be proven that this encoding is optimal for this particular string: no other encoding can represent the string using fewer than 34 bits. Moreover, the encoding is optimal for *any* string that has the same distribution of characters as the sample string. In this assignment, you will implement a method for constructing such optimal encodings due to David Huffman.

2.2 Huffman Coding

2.2.1 A Brief History

"Huffman coding" is an algorithm for constructing optimal encodings given a frequency distribution over characters. It was developed in 1951 by David Huffman when he was a Ph.D student at MIT taking a course on information theory taught by Robert Fano. It was towards the end of the semester, and Fano had given his students a choice: they could either take a final exam to demonstrate mastery of the material, or they could write a term paper on something pertinent to information theory. Fano suggested a number of possible topics, one of which was efficient binary encodings: while Fano himself had worked on the subject with his colleague Claude Shannon, it was not known at the time how to efficiently construct optimal encodings.

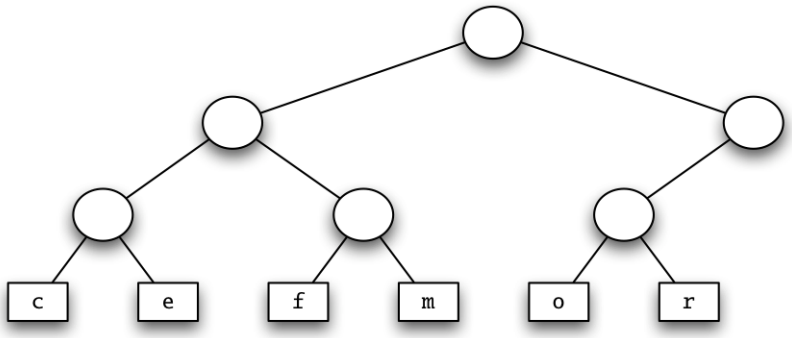

Figure 4: The custom encoding from Figure 2 as a binary tree.

Huffman struggled for some time to make headway on the problem and was about to give up and start studying for the final when he hit upon a key insight and invented the algorithm that bears his name, thus outdoing his professor, making history, and attaining an "A" for the course. Today, Huffman coding enjoys a variety of applications: it is used as part of the DEFLATE algorithm for producing ZIP files and as part of several multimedia codecs like JPEG and MP3.

2.2.2 Huffman Trees

Recall that an encoding is *prefix-free* if no code word is a prefix of any other code word. Prefix-free encodings can be represented as binary trees with characters stored at the leaves: a branch to the left represents a 0 bit, a branch to the right represents a 1 bit, and the path from the root to a leaf gives the code word for the character stored at that leaf. For example, the encodings from Figures 2 and 3 are represented by the binary trees in Figures 4 and 5, respectively.

Recall that the variable-length encoding represented by the tree in Figure 5 is an optimal encoding. The tree representation reflects the optimality in the following property: frequently-occuring characters have short paths to the root. We can see this property clearly if we label each subtree with the total frequency of the characters occurring at its leaves, as shown in Figure 6. A frequency-annotated tree is called a *Huffman tree*.

Huffman trees have a recursive structure: a Huffman tree is either a leaf containing a character and its frequency, or an interior node containing the combined frequency of two child Huffman trees. Since only the leaves contain character data, we draw them as rectangles to distinguish them from the interior nodes, which we draw as circles.

We represent both kinds of Huffman tree nodes in C_0 using a `struct htree`:

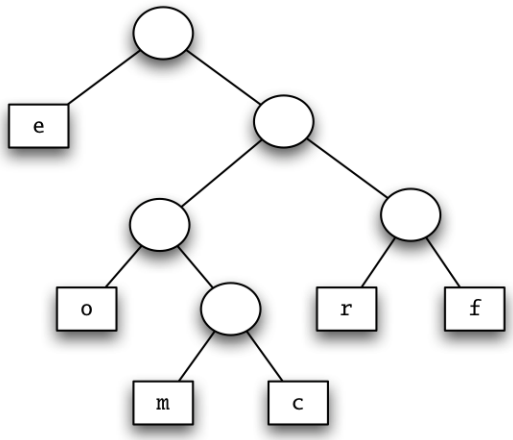

Figure 5: The custom encoding from Figure 3 as a binary tree.

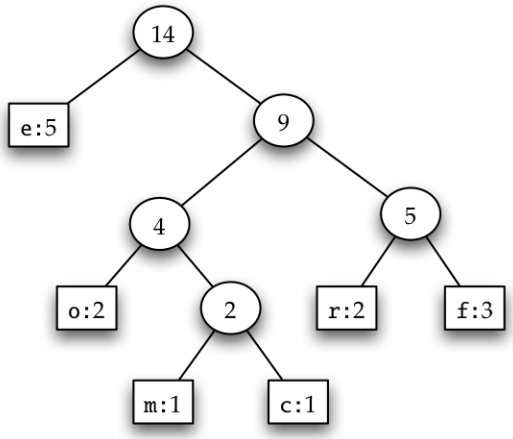

Figure 6: The custom encoding from Figure 3 as a binary tree annotated with frequencies, i.e., a Huffman tree.

```
typedef struct htree* htree;

struct htree {
    char character;  // '\0' except at leaves
    int frequency;
    htree left;
    htree right;
};
```

The `character` field of an `htree` should consist of a NUL character `'\0'` everywhere except at the leaves of the tree, and every interior node should have exactly two children. These criteria give rise to the following recursive definitions:

> An `htree` is a *valid htree* if it is non-`NULL`, its `frequency` is strictly positive, and it is either a *valid htree leaf* or a *valid htree interior node*.
>
> An `htree` is a *valid htree leaf* if its `character` is *not* `'\0'` and its `left` and `right` children are `NULL`.
>
> An `htree` is a *valid htree interior node* if its `character` is `'\0'` and its `left` and `right` children are *valid htrees*.

Task 1 (6 pts)**.** Write a specification function `bool is_htree(htree H)` that returns `true` if `H` is a *valid htree* and `false` otherwise.

2.2.3 Finding Optimal Encodings

Huffman's key insight was to use the frequencies of characters to build an optimal encoding tree from the bottom up. Given a set of characters and their associated frequencies, we can build an optimal Huffman tree as follows:

1. Construct leaf Huffman trees for each character/frequency pair.
2. Repeatedly choose two minimum-frequency Huffman trees and join them together into a new Huffman tree whose frequency is the sum of their frequencies.
3. When only one Huffman tree remains, it is an optimal encoding.

This is an example of a *greedy algorithm* since it makes locally optimal choices that nevertheless yield a globally optimal result at the end of the day. Selection of a minimum-frequency tree in step 2 can be accomplished using a *priority queue* based on a heap. A sample run of the algorithm is shown in Figure 7. (**Implementation Note:** when joining two Huffman trees put the tree removed from the priority queue first as the left child of the new Huffman tree).

Task 2 (8 pts)**.** Write a function `htree build_htree(char[] chars, int[] freqs, int n)` that constructs an optimal encoding for an `n`-character alphabet using Huffman's algorithm. The `chars` array contains the characters of the alphabet and the

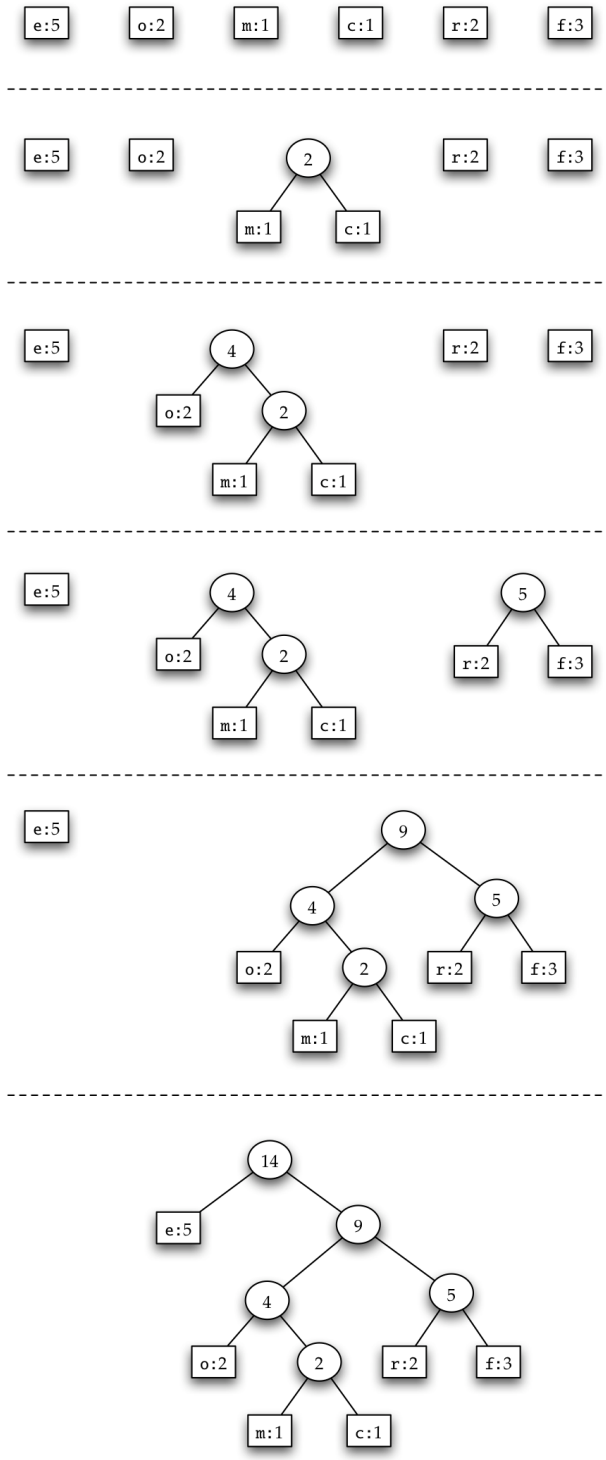

Figure 7: Building an optimal encoding using Huffman's algorithm.

`freqs` array their frequencies, where `chars[i]` occurs with frequency `freqs[i]`. Use the code in the included `heaps.c0` as your implementation of priority queues.

To test your implementation, you may use the code in `freqtable.c0`, which reads a character frequency table from a file in the following format:

```
<character 1> <frequency 1>
<character 2> <frequency 2>
...
```

2.3 Decoding Bitstrings

Huffman trees are a data structure well-suited to the task of decoding encoded data. Given an encoded bitstring and the Huffman tree that was used to encode it, decode the bitstring as follows:

1. Initialize a pointer to the root of the Huffman tree.
2. Repeatedly read bits from the bitstring and update the pointer: when you read a 0 bit, follow the left branch, and when you read a 1 bit, follow the right branch.
3. Whenever you reach a leaf, output the character at that leaf and reset the pointer to the root of the Huffman tree.

If the bitstring was properly encoded, then when all of the bits are exhausted, the pointer should once again point to the root of the Huffman tree. If this is not the case, then the decoding fails for the given inputs.

As an example, we can use the encoding from Figure 6 to decode the following message:

```
1 1 0 1 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 1 1 0 1 0 1 1 1 1 0 0 1 0 1 0 0
   r     o     o       m     f     o     r       c     r e       m e
```

Room for creme?

Bitstrings can be represented in C_0 as ordinary strings composed only of characters `'0'` and `'1'`.

```
typedef string bitstring;

bool is_bitstring(bitstring s) {
    int len = string_length(s);
    int i;
    for (i = 0; i < len; i++) {
        char c = string_charat(s, i);
        if (!(c == '0' || c == '1')) return false;
    }
    return true;
}
```

You will implement this algorithm in a function `string decode(htree encoding, bitstring bits)` that decodes `bits` based on the `encoding`. The function should return the decoded string if the bitstring can be decoded and should signal an error otherwise. Remember from the lecture 19 on testing that black box testing attempts to reduce the number of different inputs to test the code on by dividing all the possible inputs to the function into groups on which the function should act in the same way and then testing only one input from each group. Before you implement the `decode` function, you will use this methodology to implement a suite of examples to test your implementation.

Task 3 (5 pts)**.** For this task, write a series of tests that you will use to be confident that your implementation of `decode` is correct based on your reading of the spec. Following black box testing methodology, write tests that are representative of a large class of inputs and think about the different kind of results you expect from different inputs.

For each test you will submit a test file and potentially some input files. Each individual test should be given a unique number used in the naming conventions below. There should be two primary components of each test:

- **Inputs:** The main portion of each test is a well chosen set of inputs that represents the behavior of the function on a large class of possible inputs. You should either hard-code these inputs into the testing function (see below) or store them in files that you submit along with your tests.

  If you choose to place the inputs in files, put the specification of the Huffman Tree and the bitstring in different files. Name them using the convention:

  `input#-<inputType>.txt`

  where # is the test number and `<inputType>` is `FT` for the frequency table and `BS` for the bitstring (For example, the bitstring input file for your second test should be named `input2-BS.txt`).

- **Main function:** You should write a main function that runs the test on your inputs. The main function should do three things:

  1. *Test Description:* Before doing anything else, you should print a short description of the test to the screen including the class of inputs it is testing and the expected result.
  2. *Build/Import Inputs:* Next you should build or read in the inputs to `decode` for this test. If you import the inputs from files, read in the bitstring with the `read_words` function defined in the `readfile.c0` file and import the frequency table used to build your Huffman Tree using the `read_freqtable` function defined in the file `freqtable.c0`. Call these functions with the bare file name (For example, load the bitstring for your second test by calling `read_words("input2-BS.txt")`).

3. *Call* `decode`*:* Call `decode` on your inputs.
4. *Verify Results:* If the test should result in an error, there will be no output to test. Otherwise compare the output to the output you expect for the test and return `0` if the test succeeded and `1` if it did not (recall that 0 is the return value of choice when a function succeeds).

You should name the file containing the main function for the test using the following format:

`test#-<expectedOutput>.c0`

where # is the test number and `<expectedOutput>` is `F` if the function should end in an assertion or annotation failure or `S` if the function should succeed (return `0`). (For example, if your second test should succeed, you would name the file `test2-S.c0`)

Submit all test and input files. We will be looking for a set of tests that check all different possible outputs of the `decode` function without duplicate tests for inputs in the same equivalence class. (**Hint:** you should be writing a relatively small number of tests).

Task 4 (6 pts)**.** Write a function `string decode(htree encoding, bitstring bits)` that decodes `bits` based on the `encoding`. Use any functions from the `string` library to build the result string. Use the tests you developed to check your program. Add additional tests if you notice new classes of inputs that were not tested by your original suite.