15-122: Principles of Imperative Computation, Fall 2010
Assignment 5: Heaps, BSTs, and Secret Codes

William Lovas (wlovas@cs) Tom Cortina (tcortina@cs)

Out: Friday, October 15, 2010
Due (Written): Thursday, October 21, 2010 (before lecture)
Due (Programming): Friday, October 22, 2010 (11:59 pm)

1 Written: (30 points)

The written portion of this week's homework will give you some practice reasoning about heaps and 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

Refer to the heap implementation given on the course website for Lecture 14.

Exercise 1 (10 pts)**.** Consider the `build_heap` function shown on the next page that takes an array of n integers and returns a heap by rearranging the elements in the array in place. (You may assume that the index 0 in the array is not used.) The function also includes a new version of `sift_down` implemented recursively.

(a) What is the base case for the `sift_down_recursive` function? (That is, what must be true about the element at position *i* to cause the recursive calls to stop?) [1 pt]

(b) Assume that the heap array starts with the following values in the order shown (stored in positions 1 to 7):

```
72  51  32  14  93  20  48
```

Show the contents of the array after each iteration of the `for` loop is completed in the `build_heap` function. [2 pts]

(c) Write a specification function `is_heap_after(H, n)` that returns true if the array in heap H satisifies the heap invariants for all cells in the heap array after index n, false otherwise. [2 pts]

(d) In `build_heap`, show that the following partial invariant holds for each iteration of the `for` loop:

```
//@loop_invariant is_heap_after(H, i);
```

Remember: Show that the invariant is true when the loop condition is tested the first time. Then show that if the invariant is true at the start of an iteration, it is true at the end of that iteration. [3 pts]

(e) Argue using the preconditions and the loop invariant that the postcondition is satisfied. [2 pts]

```
void sift_down_recursive(heap H, int i)
{   int smallest;
    if (2*i < H->limit && H->heap[2*i] < H->heap[i])
        smallest = 2*i;
    else
        smallest = i;
    if (2*i+1 < H->limit && H->heap[2*i+1] < H->heap[smallest])
        smallest = 2*i+1;
    if (smallest != i) {
        swap(H->heap, i, smallest);
        sift_down_recursive(H, smallest);
    }
}

heap build_heap(int[] A, int n)
//@requires n > 0;
//@requires \length(A) == n;
//@ensures is_heap(\result);
{   int i;
    heap H = alloc(struct heap);
    H->limit = n;
    H->next = n;
    H->heap = A;
    for (i = (n-1)/2; i >= 1; i--) {
        sift_down_recursive(H, i);
    }
    return H;
}
```

Exercise 2 (2 pts)**.** Describe how to implement a FIFO queue using a heap so the enqueue and dequeue operations are both as efficient as possible. What is the runtime complexity of `enq` and `deq` in your design? You do not have to write code here, but your description must be clear and concise.

1.2 Binary Search Trees

Refer to the binary search tree code posted on the course website for Lecture 15. The binary search tree (BST) stores elements of type `elem` consisting of a `word` and an associated `count`. The elements are stored into the tree using the `word` as the key.

Exercise 3 (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 4 (3 pts)**.** 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;
}
```

Exercise 5 (3 pts)**.** The insert function for a binary search tree could be written iteratively instead of recursively. Describe the three logical errors in the `bst_insert` function below. Show how to fix each error by adding or changing code.

```
void bst_insert(bst B, elem x)
//@requires is_bst(B);
//@ensures bst_search(B, elem_key(x)) != NULL;
{   key kt;   // key in tree node T, if any
    key k = elem_key(x);
    tree newT = alloc(struct tree);
    tree T = B->root;
    newT->data = x; newT->left = NULL; newT->right = NULL;
    if (T == NULL) {
        T = newT; return;
    }
    while (T->left != NULL && T->right != NULL)
    //@loop_invariant T != NULL;
    {
        kt = elem_key(T->data);
        if (compare(k,kt) == 0) {
            T = newT;
            return;
        } else if (compare(k,kt) < 0) {
            if (T->left != NULL) {
                T = T->left;
            } else {
                T->left = newT;
                return;
            }
        } else {
            if (T->right != NULL) {
                T = T->right;
            } else {
                T->right = newT;
                return;
            }
        }
    }
}
```

Exercise 6 (9 pts)**.** Consider extending the BST implementation with the following function which deletes a given key from the tree. (Annotations not shown.)

```
void bst_delete(bst B, key k) {
    B->root = tree_delete(B->root, key k);
}
```

(a) Complete the code for the recursive function `tree_delete` which is used by the `bst_delete` function. This function should return a pointer to the tree rooted at `T` once the key is deleted (if it is in the tree). [6 pts]

```
tree tree_delete(tree T, key k)
{   key kt;    // key for node in tree
    if (T == NULL) {   // key is not in the tree
        return _____________________;
    }
    kt = elem_key(T->data);
    if (compare(k, kt) < 0) {          // correction made below
       ________________ = tree_delete(T->left, k); return T;
    } else if (compare(k, kt) > 0) {   // correction made below
       ________________ = tree_delete(T->right, k); return T;
    } else {      // key is in current tree node T
        if (T->left == NULL)
            return _________________________;
        else if (T->right == NULL)
            return _________________________;
        else {         // Node to be deleted has two children
            if (T->left->right == NULL) {
                // Replace the data in T with the data in the left child.
                ___________________________________________;
                // Replace the left child with its left child.
                ___________________________________________;
               return T;
            } else {
                // Search for the "inorder predecessor" of T and
                // replace the deleted node's data with this data.
                T->data = findLargestChild(T->left);
                return T;
            }
        }
    }
}
```

(b) Write the BST support function `findLargestChild` that removes and returns the largest child rooted at a given tree node. **Hint:** Think recursively! [3 pts]

2 Programming: Huffman Coding (20 points)

For the programming portion of this week's homework, you'll write a single C_0 file `huffman.c0` implementing a popular compression technique called Huffman coding.

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 `hw5-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. For this assignment, you will need to use libraries for file I/O (`file`), console I/O (`conio`), parsing (`parse`) and strings (`string`). 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 hw5 <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 1: 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 1, 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 2: 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 2 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.

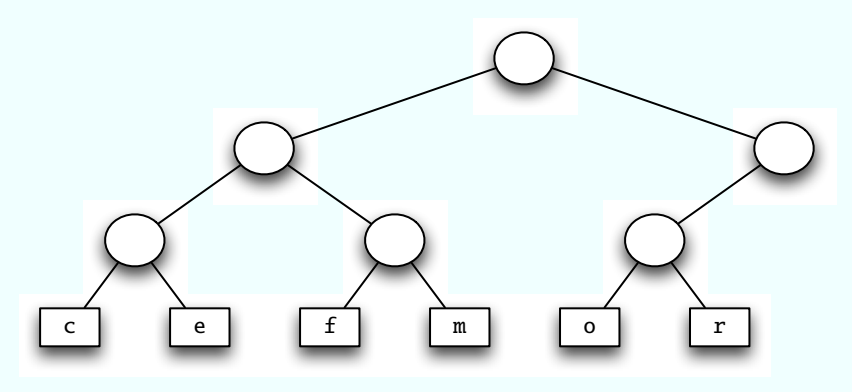

Figure 3: The custom encoding from Figure 1 as a binary tree.

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 1 and 2 are represented by the binary trees in Figures 3 and 4, respectively.

Recall that the variable-length encoding represented by the tree in Figure 4 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 5. 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`:

```
typedef struct htree* htree;
```

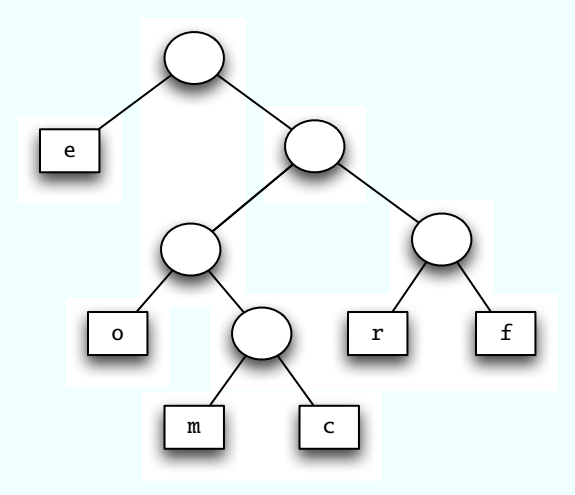

Figure 4: The custom encoding from Figure 2 as a binary tree.

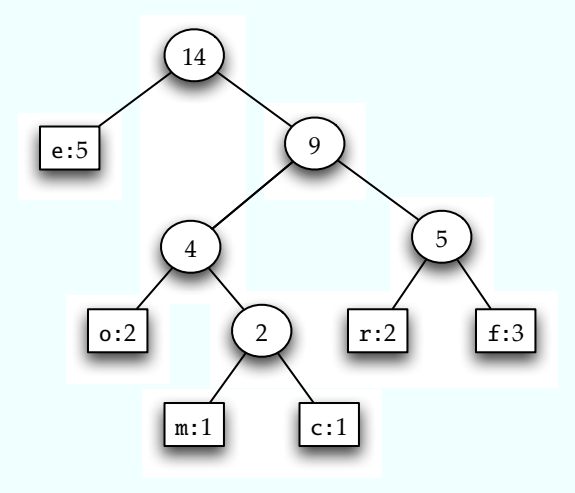

Figure 5: The custom encoding from Figure 2 as a binary tree annotated with frequencies, i.e., a Huffman tree.

```
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 6.

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 `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.

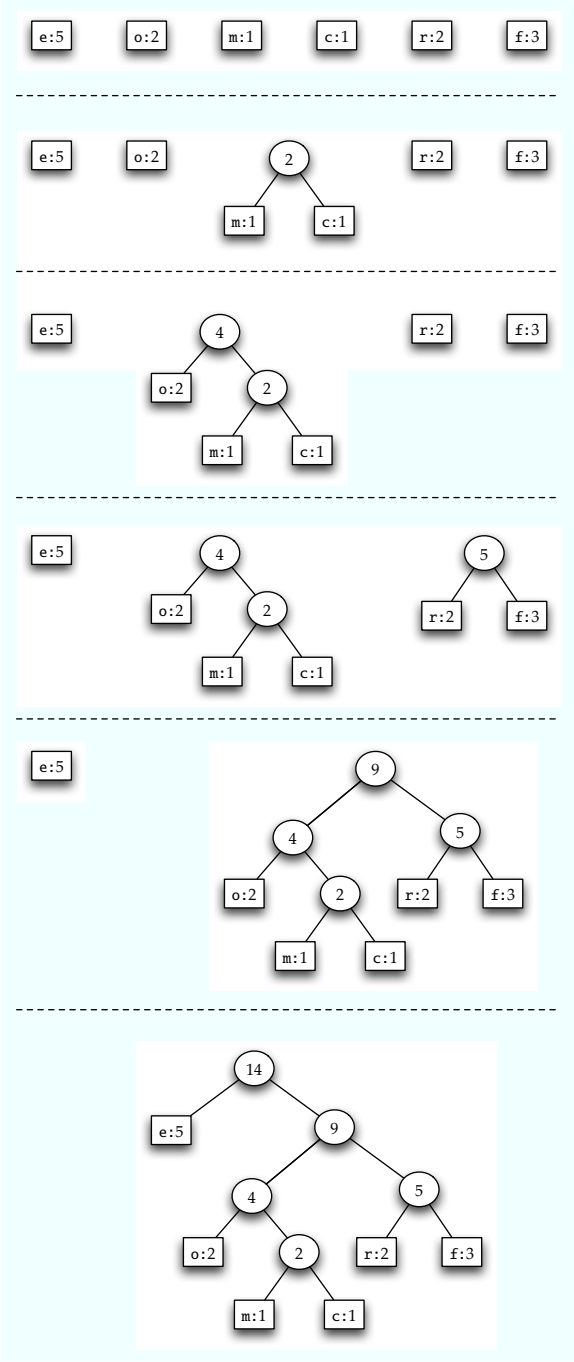

Figure 6: Building an optimal encoding using Huffman's algorithm.

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.

As an example, we can use the encoding from Figure 5 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 crème?

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;
}
```

Task 3 (6 pts)**.** Write a function `string decode(htree encoding, bitstring bits)` that decodes `bits` based on the `encoding`. If there are any trailing characters in `bits`, you should signal an error. Use any functions from the `string` library to build the result string.