9. Huffman and Trees

9.1 Depth

The following program builds a binary tree at random:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>

#define STRSIZE 5000

struct node{
        char c;
        struct node *left;
        struct node *right;
};
typedef struct node Node;

Node *MakeNode(char c);
void InsertRandom(Node *t, Node *n);
char *PrintTree(Node *t);

int main(void)
{

   char c;
   Node *head = MakeNode('A');
   Node *n;

   srand(time(NULL));
   for(c = 'B'; c < 'G'; c++){
      n = MakeNode(c);
      InsertRandom(head, n);
   }
   printf("%s\n", PrintTree(head));
   return 0;
}
```

```
Node *MakeNode(char c)
{

    Node *n = (Node *)calloc(1, sizeof(Node));
    assert(n != NULL);
    n->c = c;
    return n;

}

void InsertRandom(Node *t, Node *n)
{

    if((rand()%2) == 0){ /* Left */
        if(t->left == NULL){
            t->left = n;
        }
        else{
            InsertRandom(t->left, n);
        }
    }
    else{ /* Right */
        if(t->right == NULL){
            t->right = n;
        }
        else{
            InsertRandom(t->right, n);
        }
    }

}

char *PrintTree(Node *t)
{

    char *str;

    assert((str = calloc(STRSIZE, sizeof(char))) != NULL);
    if(t == NULL){
        strcpy(str, "*");
        return str;
    }
    sprintf(str, "%c (%s) (%s)", t->c, PrintTree(t->left), PrintTree(t->right));
    return str;

}
```

Each node of the tree contains one of the characters 'A' ... 'F'. At the end, the tree is printed out in the manner described in the course lectures.

Exercise 9.1 Adapt the code so that the maximum depth of the tree is computed using a recursive function. The maximum depth of the tree is the longest path from the root to a leaf. The depth of a tree containing one node is 1. ■

9.2 Two Trees

Adapt the code shown in Exercise 9.1, so that two random trees are generated.

Exercise 9.2 Write a Boolean function that checks whether two trees are identical or not. ■

9.3 Huffman Encoding

Huffman encoding is commonly used for data compression. Based on the frequency of occurence of characters, you build a tree where rare characters appear at the bottom of the tree, and commonly occuring characters are near the top of the tree.

For an example input text file, a Huffman tree might look something like:

```
010 :       00101 (   5 *  125)
‘ ’ :         110 (   3 *  792)
‘"’ :   111001010 (   9 *   12)
‘’’ :    00100000 (   8 *   15)
‘(’ : 01100000100 (  11 *    2)
‘)’ : 01100001101 (  11 *    2)
‘,’ :     1001001 (   7 *   39)
‘-’ :     0010010 (   7 *   31)
‘.’ :     1001100 (   7 *   40)
‘/’ : 00100110000 (  11 *    1)
‘0’ : 11100110010 (  11 *    3)
‘1’ :    00100010 (   8 *   15)
‘3’ : 01100000101 (  11 *    2)
‘4’ : 01100001001 (  11 *    2)
‘5’ : 11100110011 (  11 *    3)
‘6’ : 01100001000 (  11 *    2)
‘7’ : 01100001100 (  11 *    2)
‘8’ :   001001101 (   9 *    8)
‘9’ :    10010000 (   8 *   18)
‘:’ : 01100001011 (  11 *    2)
‘A’ :    00100111 (   8 *   16)
‘B’ :   111001101 (   9 *   13)
‘C’ :    10011011 (   8 *   22)
‘D’ :   111001110 (   9 *   13)
‘E’ :    10011010 (   8 *   19)
‘F’ :   111001000 (   9 *   11)
‘G’ :  0110000000 (  10 *    4)
‘H’ :  1110011111 (  10 *    7)
‘I’ :  1110010011 (  10 *    6)
‘J’ : 11100111101 (  11 *    3)
‘K’ :  1110010111 (  10 *    6)
‘L’ :    00100011 (   8 *   15)
‘M’ : 11100111100 (  11 *    3)
‘N’ : 01100001010 (  11 *    2)
‘O’ : 01100000111 (  11 *    2)
‘P’ :  1110011000 (  10 *    6)
‘R’ :  0110000111 (  10 *    5)
‘S’ :    10010001 (   8 *   19)
‘T’ :  0010011001 (  10 *    4)
‘U’ :  1110010010 (  10 *    5)
‘W’ :  0110000001 (  10 *    4)
‘a’ :        1010 (   4 *  339)
‘b’ :     1111110 (   7 *   60)
```

```
'c' :       100101 (  6 *   77)
'd' :        01101 (  5 *  143)
'e' :          000 (  3 *  473)
'f' :       100111 (  6 *   84)
'g' :       111000 (  6 *   94)
'h' :        11110 (  5 *  223)
'i' :         0100 (  4 *  266)
'j' :  01100000110 ( 11 *    2)
'k' :     00100001 (  8 *   15)
'l' :        10110 (  5 *  176)
'm' :       101111 (  6 *   92)
'n' :         0111 (  4 *  288)
'o' :         0101 (  4 *  269)
'p' :       101110 (  6 *   89)
'q' :  00100110001 ( 11 *    2)
'r' :        11101 (  5 *  214)
's' :         0011 (  4 *  260)
't' :         1000 (  4 *  305)
'u' :       111110 (  6 *  108)
'v' :      0110001 (  7 *   37)
'w' :      1111111 (  7 *   60)
'x' :   1110010110 ( 10 *    6)
'y' :       011001 (  6 *   72)
2916 bytes
```

Each character is shown, along with its Huffman bit-pattern, the length of the bit-pattern and the frequency of occurrence. At the bottom, the total number of bytes required to compress the file is displayed.

Exercise 9.3 Write a program that reads in a file (argv[1]) and, based on the characters it contains, computes the Huffman tree, displaying it as above. ■

9.4 Binary Tree Visualisation

The course notes showed a simple way to print out integer binary trees in this form :

```
20(10(5(*)(*))(17(*)(*)))(30(21(*)(*))(*))
```

You could also imagine doing the reverse operation, that is reading in a tree in the form above and displaying it in a 'friendlier' style :

```
20----30
|     |
10-17 21
|
5
```

The tree has left branches vertically down the page and right branches horizontally right. Another example is :

```
17(2(*)(3(*)(4(*)(*))))(6(8(*)(*))(*))
```

which is displayed as:

```
17----6
|     |
2-3-4 8
```

The above examples show the most ‘compact’ form of displaying the trees, but you can use simplifying assumptions if you wish:

- The integers stored in the tree are always ≥ 0.
- The integers stored in the tree are 5 characters (or less) in length.
- It is just as valid to print the tree in either of these ways :

```
1-6        00001-00006          00001-------------00006
| |          |     |              |                 |
2 7        00002 00008          00002             00007
|            |                    |
3-4-5      00003-00004-00005    00003-00004-00005
```

Exercise 9.4 Write a program that reads in a tree using `argv[1]` and the tree displayed to `stdout` with no other printing if no error has occurred. ■

Exercise 9.5 Write a program that reads in a tree using `argv[1]` and displays the tree using SDL. ■