Introduction to Mathematical Software Examination Sheet

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Mathematics
PD Dr. Ulf Lorenz
Christian Brandenburg

winter term 2009/2010
25/01/2010

1 The Rules

This is the assignment you have to solve in order to obtain your course attendance certificate, i.e. for receiving the credits for this course. Please read the following instructions carefully so that you fully understand what is asked of you!

- The assignment is due on Monday, February 8, at 8:00am in the exercise session.
- You are encouraged to work in groups of up to three students.
- There will be only the marks "passed" and "not passed".
- In order to pass, it is necessary to solve both exercises!
- In the exercise session on Monday, February 8, the results will be discussed group by group. Only those members of a group can "pass" who actively participate in this short discussion. Every group member should be able to explain any part of the exercise.
- **Visit the course homepage regularly in case there are corrections or additional hints for the exercises!**
- Both the C++ and the Maple exercises must be presented at the workstation computers in the computing room. We will not assess presentations on laptop or similar.

2 C++ Assignment: Huffman Coding

Hufmann coding is one of the oldest and yet most elegant forms of data compression. It is based on minimum redundancy coding, which means that if we know how often different symbols occur in a set of data, we can represent these symbols in a way such that they require less space. This is done by encoding symbols that occur more frequently with fewer bits, whereas symbols occurring less frequently are encoded with more bits. The basis of Huffman coding is the concept of entropy, which we will not dig into here. Although Huffman Coding can be used to compress arbitrary types of code, we will restrict ourselves to text characters.

The Program Code

The basis of your work is the file `huffman.cpp`, which you find at the course web page. It contains all the functions that are already provided as well as skeletons for the functions that you have to implement. Everything is commented so that you should be able to understand what the code does, at least for the parts that affect your work. Don't worry too much if you don't understand the inner workings of some of the functions, it suffices to know how to use the functions.

In this assignment, we will only work with the (extended) standard ASCII character set, see e.g., `http://www.asciitable.com/`. That means that the number of characters, defined by `NUM_CHARS` in `huffman.cpp`, is fixed to 256.

To compile the program, type `g++ huffman.cpp -o huffman` in your source directory. There are two ways to use the program: If you add the name of a text file as a parameter when executing the program, e.g. `./huffman testfile.txt`, the file is read into an array and compressed. If you do not specify any parameters, the program uses a character array that is specified at the beginning of the main routine and is usefull for debugging. From here on, the program flow is the same for both usages: the character array is compressed and the compressed data is written to a binary file. Then, the binary file is read again and the contents is uncompressed and written to a new text file.

character	␣	e	g	i	n	r	s	t
frequency	1	1	1	1	1	1	2	3

Table 1: Table of absolute frequencies for 'test string'

The Assignment

As already mentioned, your assignment is to write the 4 functions that are missing in the implementation in `huffman.cpp`. In the following section, we will provide the necessary information to do this, along with some hints. The assessment of your work will be as follows: Your program has to:

- read a text file we provide
- compress the contained data
- write the compressed data into a binary file
- read the data back from the binary file into a new bitfield
- uncompress the new bitfield
- write the new data into another text file

Note that the `main` function, as it is provided, already does all that for you. You only have to implement the four missing functions. However, you are allowed to make modifications to `main` if that helps you, as long as you do not change the behaviour stated above.

2.1 Compressing Data

Compressing data consists of the following steps:

a) examining the frequencies of the occurring characters,

b) building the Huffman tree,

c) building the table of Huffman codes,

d) generating the compressed data, consisting of a header and the compressed text.

In the following, we will investigate these steps in detail.

a) The Frequency of Characters

The first step in compressing data is to determine the frequencies with which the characters in the given character set occur. These frequencies are stored in a table of integers where each field corresponds to a character from the character set. For the string '`test string`' in the `main` function, the frequencies of occurring characters are given in Table 1.

b) The Huffman Tree

The central element in Huffman coding is the *Huffman tree*, which is a binary tree.

The Huffman tree is built in the following way: First of all, we place all occurring symbols with their frequencies in an own tree (which only consists of this node), see Figure 1. Then, we merge the two trees with the lowest frequencies and store the sum of the frequencies in the new tree's root. We repeat this procedure until we are left with only one tree, see Figure 2.

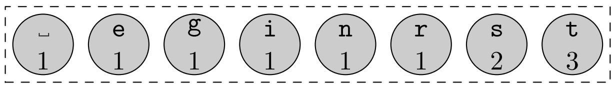

Figure 1: Building a Huffman tree for '`test string`'

The root of this final tree contains the sum of the frequencies of the symbols in the data, and the leaves of the tree contain the original symbols with their frequencies.

Creating the Huffman tree can be done efficiently by using a priority queue. However, you do not need to worry about this since that part of the program is already written. What you should make yourself familiar with is what binary trees are and how you traverse them.

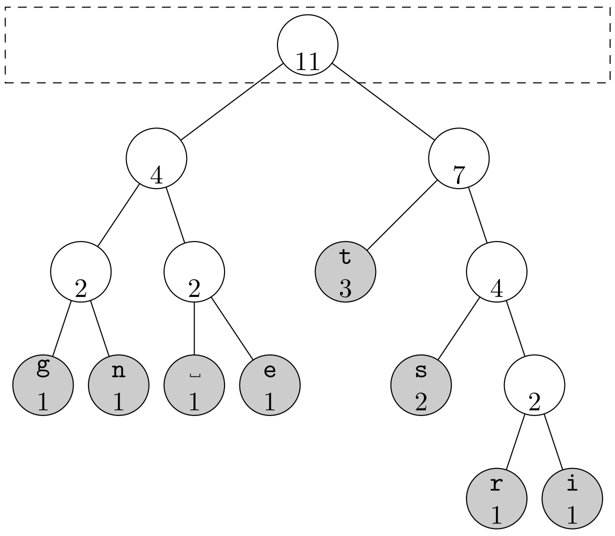

Figure 2: The Huffman tree for 'test string'

c) Huffman Code Generation

Based on the Huffman tree, we can now create the Huffman codes (that is the sequences of zeros and ones that are used to encode the characters) for the characters appearing in the input string.

The Huffman codes are generated as follows: We start at the root of the Huffman tree and descend towards its leaves. Whenever we descend to the left, we append a 0 to the Huffman code, whenever we descend to the right we append a 1, see Figure 3. As soon as we have reached a leave, we store the generated code in the code table at the position corresponding the symbol at the leave. Of course, we have to do this until we have reached all leaves. The Huffman codes for 'test sting' are given in table 2.

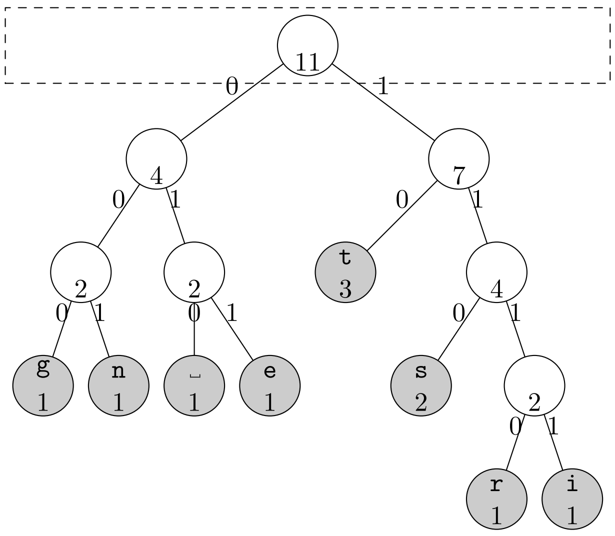

Figure 3: The Huffman tree for 'test string' with Huffman codes

d) Compressing the Data

Compressing the data is now straightforward: We iterate through the input string. For each symbol in the string, we look up the Huffman code in the table and append the corresponding code to the compressed data.

Actually, this is not quite the whole story. In order to uncompress the compressed string again, we not only need the bit sequence of the encoded symbols, but we also need information about the length of the array of the encoded symbols and about the frequencies of all the symbols in the character set. The latter is needed to reconstruct the Huffman tree

Character	Frequency	Huffman Code
␣	1	010
e	1	011
g	1	000
i	1	1111
n	1	001
r	1	1110
s	2	110
t	3	10

Table 2: Generated Huffman Codes for 'test string'

which is needed for uncompressing the compressed sequence of bits. Again, you do not need to worry about the creation of this header, as this is already done for you.

Implementation

Compressing data is done by the function

```
unsigned int huffman_compress(unsigned char** compressed, unsigned char* string,
                              unsigned int size),
```

which is already implemented. To do the actual work, it calls several subfunctions, some of which you have to implement.

Unfortunately, C does not offer a data type that is designed to contain single bits. Therefore, to represent the compressed data, we slightly abuse the data type `unsigned char` to create an array in which we store the bits of the compressed data. The bits in the resulting array of type `unsigned char` can be read and written with the functions `get_bits` and `set_bits`, resp., that you find in the source code file.

Hint: The function `huffman_compress` contains some commented lines that produce more verbose output to the screen. You can uncomment these lines to get some additional debugging information.

Tasks:

a) Implement the function
```
void create_freq_array(unsigned int freqs[NUM_CHARS], unsigned char* string, unsigned int size);.
```

This function determines the frequencies of all occurring characters and scales them to fit within one byte, i.e. it scales them to values between 0 and 255.
`freqs` denotes the array in which we store the scaled frequencies of the characters. The offsets of the array denote the decimal values of the characters
`string` is the input string of text characters
`size` is the size of the input string
Hint: Be carefull that you don't scale any frequencies to zero for symbols that do appear in the string!

b) The function
```
HuffNode* build_Huffman_tree(unsigned int freqs[NUM_CHARS]);
```
is already implemented.

c) Implement the function
```
void create_table_rec(HuffCode table[NUM_CHARS], HuffNode* node,
                      unsigned char* code, unsigned int pos);
```

This function is called by `void create_table(HuffCode table[NUM_CHARS], HuffNode* tree);` and creates the table of Huffman codes from the Huffman tree by recursively calling itself.

The Huffman codes for the symbols are stored in objects of type `HuffCode`, which are arranged in an array of size `NUM_CHARS`. The positions in the array correspond to the ASCII code of the characters. The `HuffCode` objects consist of a fixed-size bitfield `bits` that is large enough to contain all coding lengths that might possibly occur, a counter `size` containing the coding length of the character and a boolean variable `used` that says whether the corresponding symbol occurs in the input string.

create_table traverses the Huffman tree in depth-first order. In each call, code keeps track of the Huffman code that is being created, with pos specifying its length. Whenever we descend to the left, we append a 0 to the code, when we descend to the right, we append 1 accordingly. Whenever we reach the base case, i.e. a node with no children, we write the code and its length to the corresponding HuffCode object in the array table.

d) Implement the function
```
unsigned int compress(HuffCode table[NUM_CHARS], unsigned char* compressed,
                      unsigned char* string, unsigned int size);
```

This function iterates over the input data in string of size size and appends for each symbol the corresponding bit sequence to the bitfield compressed. The function returns the size of the bitfield **in bytes** (so you will have to round upwards when the number of bits is not a multiple of 8).
Hints: the bitfield compressed is chosen large enough to contain all bits, so you do not need to worry about memory. You should use a counter that remembers at which position you are in the bitfield.

2.2 Uncompressing Data

Uncompressing data consists of the following steps:

e) reading the header from the compressed data,

f) reconstructing the Huffman tree from the header,

g) reconstructing the compressed text.

e) Reading the Header Information

The first step in the uncompressing process is to retrieve the header information, which consist of the (uncompressed) size of the encoded string in bytes and the scaled frequencies of *all* symbols in the character set. The frequencies are read into an array of the same type as in part a). This part of the code is already implemented.

f) Reconstructing the Huffman Tree

To uncompress the encoded data, we need to restore the Huffman tree. With the information from step e), this can be accomplished with the function build_Huffman_tree from part b).

g) Reconstructing the compressed text

Finally, using the Huffman tree from part f), we reconstruct the encoded data. We iterate over the compressed bitfield, and read the bits at the corresponding positions. Starting at the root of the Huffman tree, we descend to the left whenever we encounter a 0 and we descend to the right when we encounter a 1. As soon as we reach a node, we have found the next encoded symbol. We append the symbol to the uncompressed string, return to the root of the Huffman tree, and start to descend again in the same fashion, starting at the next position in the bitfield, until we have restored all symbols.

Implementation

Uncompressing data is done by the function
unsigned int huffman_uncompress(unsigned char* compressed, unsigned char** string); which is also already implemented. To do the actual work, it also calls several subfunctions. As we can reuse the functions for creating the frequency table and the Huffman tree, there is only one thing left:

g) Implement the function
```
void uncompress(HuffNode* root, unsigned char* compressed,
                unsigned char* string, unsigned int size);
```

Hints: You have to keep track of your position in the bitfield as well as the number of symbols that have already been restored. This function can be implemented iteratively (i.e. you don't need recursion). How do you know when to stop reading bits?

3 Maple Assignment: Design of a Beer Glass

Notice: If not stated otherwise, you are supposed to solve the problems analytically.

A brewery would like to design their own beer glass. It is your job to help them design the glass by using solids of revolution. More information on solids of revolution can be found at http://en.wikipedia.org/wiki/Solid_of_revolution.

The solid of revolution around the x-axis is supposed to be given in terms of a function $p(x)$. A designer has given the following requirements (with lengths in cm):

$$p(0) = \frac{5}{2}, \quad p(6) = \frac{19}{10}, \quad p(18) = \frac{9}{2}, \quad p(20) = 4, \quad p'(6) = p'(18) = 0.$$

Although polynomial interpolation is often not a very good approach (interpolating large point sets can lead to strong oscillations and spline interpolation should be used instead), it is sufficient for what we want to achieve here. As you know, a polynomial of order n is uniquely defined by $n + 1$ function constraints (which might be function or derivative values). As we are given 6 values, we are looking for a polynomial of degree 5.

a) Find the polynomial $p(x)$ of degree 5 that satisfies the specification above.

b) The glass should have a height of 20cm. Plot the function $p(x)$ (not the solid of revolution) on the interval $[0, 20]$. Make sure both axes are scaled identically.

Next, we would like to visualize the solid of revolution. To this end, the Maple method `VolumeOfRevolution` is helpful. Before you continue, find out which package you have to include and what the parameter `output` does.

c) Plot the solid of revolution on the interval $[0, 20]$.
Hint: store the plot in a variable for the upcoming tasks and draw it using `display` from the `plot` package. There are faster/easier ways than to use `3dplot`.

The glass is to be calibrated to 500ml.

d) Determine the x-position of the calibration mark. You can do this numerically as the calibration mark cannot be positioned exactly anyway due to its thickness.

e*) (not necessary for passing) Plot the calibration mark and the bottom of the glass. *Hint:* the Maple command `spacecurve` might come in handy.

You want to hide a cylinder with height $20 - L$ inside the glass. In particular, you would like to know the volume of the largest cylinder you can put the glass over (which need not be the largest cylinder fitting inside the glass). This means that the cylinder ranges from L to 20 in x-direction.

f) Find $L \in [0, 20]$, such that the volume of the cylinder fitting under the glass becomes maximal. You can compute the extremal values using `fsolve`, if necessary. A sketch with a not yet maximal cylinder is given in Figure 4.

g) Visualize the glass with the inscribed cylinder.

h) Compute the volume of the cylinder. (It should be larger than 230).

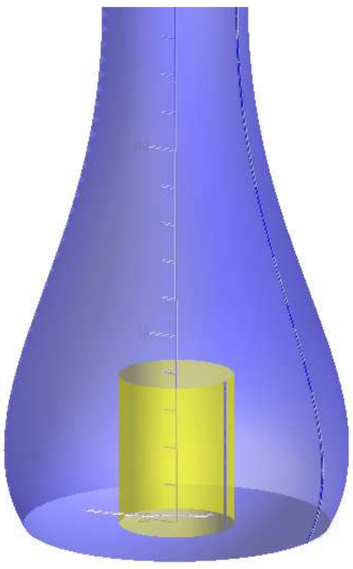

Figure 4: Cylinder under the glass