CS 202 – Data Structures (Spring 2007)

Homework Assignment 4: Data Compression

Assigned on 1 May, 2007 — Due by 23:00:00 on Thursday 17 May, 2007
Graded by Deniz Türdü (denizturdu@...)

PLEASE NOTE:

- The deadline is hard: No exceptions will be made.
- Solutions must be your own: No cooperation is permitted.
- Late solutions will be penalised by 10 points for each day of delay, but solutions that are late by more than 2 days will get 0 points.
- Solutions must be submitted via the WebCT server (whose clock may differ from yours): No other method of submission will be accepted.

1 Introduction

The purpose of *data compression* is to take an input file A and, within a reasonable amount of time, transform it into an output file B in such a way that B is smaller than A and it is possible to reconstruct A from B. A program that converts A into B is called a *compressor*, and one that undoes this operation is called a *decompressor*. Programs such as *WinZip* perform this function, among others. Compression enables us to store data more efficiently on storage devices or transmit data faster using communication facilities, since fewer bits are needed to represent the actual data.

A compressor cannot guarantee that B will always be smaller than A. Indeed, if this were possible, then what would happen if one just kept compressing the output of the compressor?! Any compressor that compresses some files must thus also actually enlarge some files. Nevertheless, compressors tend to work pretty well on the kinds of files (especially those generated by Micro$oft) that are typically found on computers, and they are widely used in practice.

2 The Ziv-Lempel Algorithm

The considered algorithm is a version of the *Ziv-Lempel data compression algorithm*, which is the basis for must popular compression programs, such as *WinZip*, *zip*, and *gzip*. You may find this algorithm a little difficult to understand at first, but your program could be quite short.

It is an example of an *adaptive* data compression algorithm: the code used to represent a particular sequence of bytes in the input file may be different for distinct input files, and may even be different if the same sequence appears in more than one place in the input file.

2.1 Compressor

The Ziv-Lempel compressor maps strings of input characters into numeric codes. To begin with, each character of the set of characters that may occur in the text file, called the *alphabet*, is assigned a code. For example, suppose the input file starts with the string:

aaabbbbbbaabaaba

This string is composed of the characters **a** and **b**. Assuming the alphabet is just $\{\mathbf{a}, \mathbf{b}\}$, initially **a** is assigned the code 0 and **b** the code 1. The mapping between character strings and their codes is stored in a dictionary. Each dictionary entry has two fields: a *code* and a *string*. The character string represented by the field *code* is stored in the field *string*. The initial dictionary for our example is given by the first two columns below:

code	0	1	2	3	4	5	6	7
string	**a**	**b**	**aa**	**aab**	**bb**	**bbb**	**bbba**	**aaba**

Beginning with the dictionary initialised as above, the Ziv-Lempel compressor repeatedly finds the longest prefix p of the unprocessed part of the input file that is in the dictionary and outputs its code. Furthermore, if there is a next character c in the input file, then pc (denoting the string p followed by the character c) is assigned the next available code and inserted into the dictionary. This strategy is called the *Ziv-Lempel rule*.

Example 1 Let us apply the Ziv-Lempel rule on the example string **aaabbbbbbaabaaba** above. The longest prefix of the input that is in the initial dictionary is **a**. Its code 0 is output and the string **aa** (for $p = \mathbf{a}$ and $c = \mathbf{a}$) is assigned the code 2 and entered into the dictionary. Now, **aa** is the longest prefix of the remaining string that is in the dictionary. Its code 2 is output and the string **aab** (for $p = \mathbf{aa}$ and $c = \mathbf{b}$) is assigned the code 3 and entered into the dictionary. *Even though* **aab** *has the code* 3 *assigned to it, the code* 2 *for* **aa** *is actually output! The suffix* **b** *will be a prefix of the string corresponding to the next output code. The reason for not outputting* 3 *is that the dictionary is* not *part of the compressed file. Instead, the dictionary has to be reconstructed during decompression using the compressed file. This reconstruction is possible only if we adhere strictly to the Ziv-Lempel rule: see the next subsection.* Following the output of the code 2, the code 1 for **b** is output and **bb** is assigned the code 4 and entered into the dictionary. Then, the code 4 for **bb** is output and **bbb** is entered into the dictionary with code 5. Next, the code 5 is output and **bbba** is entered into the dictionary with code 6. Then, the code 3 is output for **aab** and **aaba** is entered into the dictionary with code 7. Finally, the code 7 is output for the entire remaining string **aaba**. The example string is thus encoded as the sequence 0214537 of codes.

2.2 Uncompressor

For decompression, we read the codes one at a time and replace them by the strings they denote. The dictionary can be dynamically reconstructed as follows. The codes assigned for single character strings are entered as (*code*, *string*) pairs into the dictionary at the initialisation (just as for compression). This time, however, the dictionary is searched for an entry with a given code (rather than with a given string). The first code in the compressed file necessarily corresponds to a single character and so may be replaced by that character, which is already in the dictionary. For all other codes x in the compressed file, we have two cases to consider:

1. If the code x is already in the dictionary, then the corresponding string, denoted by $\mathit{string}(x)$, is extracted from the dictionary and output. Furthermore, we know that for the code q that precedes x in the compressed file the compressor created a new code for the string $\mathit{string}(q)$ followed by the first character of $\mathit{string}(x)$, denoted by $\mathit{fc}(x)$. So we also enter the pair $(q+1, \mathit{string}(q)\mathit{fc}(x))$ into the dictionary.

2. If the code x is not yet in the dictionary, then the uncompressed text segment corresponding to the compressed file segment qx has the form $\mathit{string}(q)\mathit{string}(q)\mathit{fc}(q)$, where q is the code that precedes x in the compressed file. Indeed, during compression, the string $\mathit{string}(q)\mathit{fc}(q)$ was assigned the new code x in the dictionary and the code x was output. So we output $\mathit{string}(q)\mathit{fc}(q)$ and enter the pair $(x, \mathit{string}(q)\mathit{fc}(q))$ into the dictionary.

Example 2 Let us apply this decompression scheme on the string **aaabbbbbbaabaaba**, which was compressed in Example 1 into the code sequence 0214537. The dictionary is initialised with the pairs $(0, \mathbf{a})$ and $(1, \mathbf{b})$. The first code is 0, so its string **a** is output. The next code, 2, is still undefined. Since the previous code 0 has $\mathit{string}(0) = \mathbf{a}$ and $\mathit{fc}(0) = \mathbf{a}$, we have $\mathit{string}(2) = \mathit{string}(0)\mathit{fc}(0) = \mathbf{aa}$, so **aa** is output and $(2, \mathbf{aa})$ is entered into the dictionary. The next code, 1, triggers output **b** and $(3, \mathit{string}(2)\mathit{fc}(1)) = (3, \mathbf{aab})$ is entered into the dictionary. The next code, 4, is not yet in the dictionary. The preceding code is 1, so $\mathit{string}(4) = \mathit{string}(1)\mathit{fc}(1) = \mathbf{bb}$. The pair $(4, \mathbf{bb})$ is entered into the dictionary and **bb** is output. Similarly for the next code, 5, where $(5, \mathbf{bbb})$ is entered into the dictionary and **bbb** is output. The next code is 3, which is already in the dictionary, so $\mathit{string}(3) = \mathbf{aab}$ is output and the pair $(6, \mathit{string}(5)\mathit{fc}(3)) = (6, \mathbf{bbba})$ is entered into the dictionary. Finally, when the code 7 is read, the pair $(7, \mathit{string}(3)\mathit{fc}(3)) = (7, \mathbf{aaba})$ is entered into the dictionary and **aaba** is output.

3 Implementing the Ziv-Lempel Algorithm

The data structure to be used by the *compressor* is a *hash table* storing codes. New codes are entered into the hash table for given strings, and the hash table is queried with strings for the corresponding codes, if any. *For this homework, we limit ourselves to the codes 0 through 4095.* The ASCII codes 0 through 255 will be used for single character strings, even though the character with ASCII value 0 will never be encountered in the input file. So the first new code that the compressor actually assigns will be 256. If more than 4096 codes are needed, then do not generate new codes but use the available ones; this may just give less compression, but otherwise is not a problem.

The *decompressor* can actually be simpler. Since we just query on codes, rather than on strings, use an *array* of 4096 strings and initialise it for the first 256 single character strings. Beware of negative character values. For instance, array position 65 (which corresponds to ASCII symbol **A**) *must* contain (or point to) the string "**A**". Starting with position 256, new strings are added to this array.

4 Work To Be Done

First design and implement a `HashTable` class, using *open addressing* with *linear* probing. You may modify the code of a former edition of the textbook (see the course webpage) for hash tables with quadratic probing, otherwise you design your own class. Make sure it is general and works for an arbitrary object class that provides an `operator==` method.

Second, write a program called `compress` that performs compression. To avoid a number of problems, make this program actually print out the sequence of codes as integers with one space between each integer, with no spaces at the beginning of the file, and exactly one space at the end of the file after the last integer code. Actual compression will involve rather advanced C++ details, such as binary input/output, bit packing, etc, which is really beyond the scope of this homework. There may thus not be any actual compression here. The compression program reads the input from a file called `compin` and produces a file called `compout`.

Third, write a program called `decompress` that performs decompression. It reads the file `compout` and produces a file `decompout`. The contents of `compin` and `decompout` should be equal if your programs are working correctly.

Example 3 The files `compin` and `decompout` are

aaabbbbbbaabaaba

if and only if the file `compout` is

0□2□1□4□5□3□7□

where □ denotes the space character.

What and How to Submit When?

Take our warning on plagiarism **very seriously**. We assume that by submitting your solution, you are certifying that you are submitting your own work. Take the following steps:

1. Prepare two project folders, called *compress-program* and *decompress-program*. Put your respective code into the suitable folder and then put the two folders into a folder called *LastnameName-StudentID-hw4*. Do not use any special Turkish characters in the folder name. Remove any executables (debug or release), as they take up too much space.

2. Compress your folder, giving *MehmetogluAli-15432-hw4.zip* for example. *Make sure it decompresses properly, reproducing your folder exactly, and actually corresponds to assignment* 4. Do not use the compression program you have just written!

3. Submit this compressed file via WebCT by the deadline given on the first page.

Your solution will be graded in the following way:

- If your `compress` program does not use a *general* `HashTable` class using *open addressing* with *linear* probing, then you get 0 points, even if your programs work correctly otherwise.

- For each program that compiles without error, you get 15 points and the corresponding test(s) below will be made.

- We will run three tests on your programs:

  - We will decompress with *your* `decompress` program a file obtained with *our* `compress` program. If the input and output are identical, then you get 20 points.
  - We will decompress with *our* `decompress` program two files obtained with *your* `compress` program. For each identical input/output pair, you get 15 points.

  Your outputs must be *exactly* in the format described above, without a single additional character: otherwise your programs are considered to function incorrectly.

- The style of your own code will be graded on clarity, comments, etc. This will cover 20 points. You will however not get any of these points if your programs fail in *all* the tests we perform. A nice looking but useless program is just a useless program, no matter how nice it is.

For late submissions, we will first grade your solution as indicated above and then discount accordingly. So, assuming you got 90 points, if you were late at least one second but at most one day, then your actual grade will be 80 points; if you were late more than one day but at most two days, then your actual grade will be 70 points; if you were late more than two days, then your actual grade will be 0 points and your solution will not be graded.

Have fun!

Hints

You can perform character input and output (for reading from `compin` and writing to `decompout`) as done in the example code below:

```
/*
  Read file "tale" character by character and write them to file "yaz".
  So in the end "yaz" is a copy of "tale".
*/

#include <iostream>
#include <fstream>

using namespace std;

int main()
{
  char ch;
  ifstream deneme("tale");
  ofstream sonuc("yaz");
  deneme.get(ch);
  while(!deneme.eof()) // eof returns true if next character is eof
  {
    sonuc << ch;
    deneme.get(ch);  // get reads next character unless at the end of the file
  }
  deneme.close();
  sonuc.close();
  return 0;
}
```

For dealing with strings, you may use the `string` class you learned about in CS 201, or use the `mystring` class of a former edition of the textbook (see the course webpage), or just use plain character arrays. The only needed methods (during decompression) are to make a (deep) copy of an existing string, to append a symbol to its end, and to store it somewhere in the array of strings that you are maintaining.