Sabancı University
Faculty of Engineering and Natural Sciences

CS 300 Data Structures - Homework 2
Ziv-Lempel Compression Algorithm

Assigned: March 21, 2025
Due: April 07, 2025 @ 23:55

Introduction

In this assignment, you will implement an algorithm for **data compression**. The purpose of data compression is to take a file **A** and, within a reasonable amount of time, transform it into another file **B** in such a way that: (i) B is smaller than A, and (ii) 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 an *uncompressor*.

Popular programs such as WinZip perform this function, among others. Compressing data enables us to store data more efficiently on our storage devices or transmit data faster using communication facilities because fewer bits are needed to represent the actual data.

Of course, a compressor cannot guarantee that the file B will always be smaller than A. It is easy to see why: If this were possible, imagine what would happen if you just kept iterating the method, compressing the output of the compressor. In fact, with a little bit of thinking, you should be able to convince yourself that any compressor that compresses some files must also actually enlarge some files. Nevertheless, compressors tend to work pretty well on the types of files that are typically found on computers, and hence are widely used in practice.

The Ziv-Lempel Algorithm

The algorithm that you will implement is a version of the *Ziv-Lempel data compression algorithm*, which is the basis for must popular compression programs such as *winzip*, *zip* or *gzip*. At first, you may find this algorithm a little difficult to understand, but in the end, the program that you will write to implement it should not be very long.

Ziv-Lempel is an example of an adaptive data compression algorithm. What this means is that the code that the algorithm uses to represent a particular sequence of bytes in the input file will be different for different input files, and may even be different if the same sequence appears in more than one place in the input file.

This is how the algorithm operates: The Ziv-Lempel compression method maps strings of input characters into numeric codes. To begin with, all characters that may occur in the text file (that is, the alphabet) are assigned a code. For example, suppose the input file to be compressed has the string:

aaabbbbbbaabaaba

This string is composed of the characters ’a’ and ’b’. Assuming our alphabet of symbols is just a, b, initially ’a’ is assigned the code 0 and ’b’ the code l. The mapping between character strings and their codes is stored in a *dictionary.* Each dictionary entry has two fields: *key* and *code.* The character string represented by *code* is stored in the field *key.* The initial dictionary for our example is given by the first two columns below (i.e., the shaded part with codes 0 and l):

Code	0	1	2	3	4	5	6	7
Key	a	b	aa	aab	bb	bbb	bbba	aaba

Table 1: Code to Key Mapping

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

Let us try the Ziv-Lempel method on our example string. The longest prefix of the input that is in the initial dictionary is ’a’. Its code, 0, is output, and the string ’aa’ (**p** = ‘a’ and **c** = ‘a’) is assigned the code 2 and entered into the dictionary. ’aa’ is the longest prefix of the remaining string that is in the dictionary. Its code, 2, is output; the string ’aab’ (**p** = ‘aa’, **c** =’b’) is assigned the code 3 and entered into the dictionary.

Notice that even though ’aab’ has the code 3 assigned to it, only the code 2 for ’aa’ is output. The suffix ’b’ will be part of the next code output. The reason for not outputting 3 is that the code table is not part of the compressed file. Instead, the code table has to be reconstructed during decompression using the compressed file. This reconstruction is possible only if we adhere strictly to the Lempel-Ziv rule.

Following the output of the code 2, the code for ’b’ is output; ’bb’ is assigned the code 4 and entered into the code dictionary. Then, the code for ’bb’ is output, and ’bbb’ is entered into the table with code 5. Next, the code 5 is output, and ’bbba’ is entered 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 remaining string ’aaba’. Our sample string is thus encoded as the sequence of codes: 0 2 1 4 5 3 7.

For decompression, we input the codes one at a time and replace them with the texts they denote. The code-to-text mapping can be reconstructed in the following way. The codes assigned for single-character texts are entered into the dictionary at the initialization (just as we did for compression). As before, the dictionary entries are code-text pairs. 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 must correspond to a single character (why?) and so may be replaced by the corresponding character (which is already in the dictionary!) For all other codes x in the compressed file, we have two cases to consider:

1. **The code x is already in the dictionary:** When **x** is in the dictionary, the corresponding text, *text(**x**)*, to which it corresponds, is extracted from the dictionary and output. Also, from the working of the compressor, we know that if the code that precedes x in the compressed file is **q** and *text(**q**)* is the corresponding text, then the compressor would have created a new code for the text *text(**q**)* followed by the first character (that we will denote by *fc(**x**)*), of *text(**x**)* (Understand why this is the case!) So, we enter the pair *(next code, text(**q**)fc(**x**))* into the dictionary.
2. **The code x is NOT in the dictionary:** This case arises only when the current text segment has the form *text(**q**)text(**q**)fc(**q**)* and *text(**x**)* = *text(**q**)fc(**q**)* (Understand why this is the case!) The corresponding compressed file segment is **qx**. During compression, *text(**q**)fc(**q**)* is assigned the code **x**, and the code **x** is output for the text *text(**q**)fc(**q**)*. During decompression, after **q** is replaced by *text(**q**)*, we encounter the code **x**. However, there is no code-to-text mapping for **x** in our table. We are able to decode **x** using the fact that this situation arises only when the decompressed text segment is *text(**q**)text(**q**)fc(**q**)*. When we encounter a code **x** for which the code-to-text mapping is undefined, the code-to-text mapping for **x** is *text(**q**)fc(**q**)*, where **q** is the code that precedes **x** in the file.

Let us try this decompression scheme on our earlier sample string

aaabbbbbbaabaaba

which was compressed into the code sequence 0 2 1 4 5 3 7.

1. To begin, we initialize the dictionary with the pairs (0, a) and (1, b), and obtain the first two entries in the dictionary above.
2. The first code in the compressed file is 0. It is replaced by the text 'a'.
3. The next code, 2, is undefined. Since the previous code 0 has *text(*0*)* = 'a', *fc(*0*)*='a' then *text(*2*)* = *text(*0*)fc(*0*)* = 'aa'. So, for code 2, 'aa' is output, and (2, 'aa') is entered into the dictionary.
4. The next code, 1, is replaced by *text(*1*)* = ‘b' and (3, *text(*2*)fc(*1*)*) = (3, 'aab') is entered into the dictionary.
5. The next code, 4, is not in the dictionary. The code preceding it is 1 and so *text(*4*)*= *text(*1*)fc(*1*)*= 'bb'. The pair (4, 'bb') is entered into the dictionary, and 'bb' is output to the decompressed file.
6. When the next code, 5, is encountered, (5, 'bbb') is entered into the dictionary and 'bbb' is output to the decompressed file.

7. The next code is 3 which is already in the dictionary so *text(3)* = 'aab' is output to the decompressed file, and the pair (6, *text (5)fc(3)*) = (6, 'bbba') is entered into the dictionary.
8. Finally, when the code 7 is encountered, (7, *text(3)fc(3)*) = (7, 'aaba') is entered into the dictionary and 'aaba' output.

Implementing the Ziv-Lempel Algorithm

The basic data structure used by the Ziv-Lempel compression algorithm is a *hash table.* The hash table stores *objects* that consist of a string and the corresponding code that is assigned. During compression you will insert new (string, code) pairs to the hash table (obviously hashing on just the string part), or query the hash table with a string and if it is there, obtain the code associated so that you can output it.

One implementation restriction is that you should be able to assign a maximum number codes. Let us assume for this homework that *you will need at most 4096 codes.* (This means that you will be able to store only 4095 distinct strings including the single character strings.) Please note that codes 0 through 255 will be used for single character strings (although you will never encounter the character with value 0 in your input.) So the first code that YOUR PROGRAM will assign will be 256. If during compression you find that you need more codes, you do not generate new codes but just use the available codes. This will give you less compression but otherwise it is not a problem.

The algorithm for uncompressing a file is actually simpler. You need to keep an array (of size 4096) of strings. This array is initialized for the first 256 singles character strings, so that for instance, array position, 65 (which corresponds to ASCII symbol A) **contains or points** to the string "A". Starting with position 256, new character strings that are composed as described above for decompression are added to this table. You can use a null pointer to detect that a string for a code is not yet composed.

You should convince yourself that these two data structures should be sufficient for you to implement the compression and decompression algorithms.

Your Task

You will complete the following for this homework:

- You will design and implement a hash table class using open addressing with linear probing. You can use the code available on the lecture slides for hash tables with quadratic probing if you want, and modify it (hopefully after understanding how it works) or you can design you own hash table class. **But please make sure that your hash table is general and works for any arbitrary object class that overloads the operator == and/or != . You will lose points if your hash-table class is not general.**

- You will write two programs: The first one that you will call **compress**, will perform compression. To avoid a number of problems we will 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 written.* Actual compression will involve rather disgusting C++ details such a binary input/output, bit packing, etc., which is really beyond the scope of this homework. Thus, you will not actually compress but pretend you are compressing and print the sequence of integer codes for the file to be compressed. The compression program will read the input from the command line, and output the compressed version of it back to the command line.

- The second program that you will call **decompress** will perform decompression. Similarly, it should read the compressed characters from the command line, and output the decompressed string back to the command line. Obviously the uncompressed string in the above section and the decompressed string in this section should be equal if your programs are working correctly.

- You should first prompt the user on which program they would like to utilize. Based on their answer, you should then ask for a string to compress, or a compressed input to decompress. Following their answer, you can use the answer as an input for your compression or your decompression programs. Once you output the answer to the console, you can terminate the program.

- You can find sample runs at the end of the document, showcasing how this process should work.

- Your code should be submitted to SUCourse at the deadline given on the first page.

- You then submit this compressed file in accordance with the deadlines above.

Your homework will be graded in the following way:

- We will run 6 tests on your homework and check the results.
  - We will first test your decompress program with a file that we compress. We will thus make sure that your decompress program is general and not geared towards your own compressor. If this test fails, the other tests will not be done and you will not receive any credits from any of the tests.
  - We will compress 5 text files with your version of the compress program and then decompress it with your decompress program and compare the input and the output. If they are the same they the test succeeds otherwise the test fails.

- Note that your compression output file should be in the exact same format we described above.

- The style of your OWN code will be graded on clarity, commenting, etc. This will cover 10 points. You will however NOT get this credit if your program fails in ALL the tests we perform. A nice looking but useless program is just a useless program not matter how nice it is.

You should populate your hash table with all ASCII characters in its initialization (so that it can process all possible inputs) as done in the example code below:

```
/*
* This piece of code populates the hash table with all
* ASCII characters.
*/
#include <string>
#include ”HashTable.h”
using namespace std;
int main()
{
    // Example hash table with fixed size
    HashTable<DictItem> t( DictItem( ””, −1 ), true, 4096 );

    // Insert ASCII characters
    string current;
    int counter;
    for (counter = 0; counter < 256; ++counter) {
      current = char( counter );
      t.insert( DictItem( current, counter ) );
    }
    return 0;
}
```

For dealing with strings you may use the **string** class that you learned about in CS 201 (#include <string>), or just use plain character arrays. The only interesting thing with strings that you will do (during decompression) is to make a (deep) copy of an existing string, and append a symbol to the end of it and store it somewhere in the array of strings you are maintaining.

Examples

- String to compress: ”aaabbbbbbaabaaba”

  Console output:

  ```
  To compress a file, press 1. To decompress a file, press 2: 1
  Enter the input string: aaabbbbbbaabaaba
  Compressed output: 97 256 98 258 259 257 261
  ```

  Note that ”Compressed output” has one extra space character at the end.

- String to decompress: ”97 256 98 258 259 257 261 ”

  Console output:

  ```
  To compress a file, press 1. To decompress a file, press 2: 2
  Enter the compressed string: 97 256 98 258 259 257 261
  Decompressed string: aaabbbbbbaabaaba
  ```

  Note that when entering the compressed string, we will use the same compressed output. So there will be a space character at the end of the string.

- String to compress: ”aa aa aab aaaab aaaab aaaa aaa aa aaaabaa”

  Console output:

  ```
  To compress a file, press 1. To decompress a file, press 2: 1
  Enter the input string: aa aa aab aaaab aaaab aaaa aaa aa aaaabaa
  Compressed output: 97 97 32 256 258 97 98 260 256 262 256 264 263 257 266 260 268 98 256
  ```

- String to decompress: ”97 97 32 256 258 97 98 260 256 262 256 264 263 257 266 260 268 98 256 ”

  Console output:

  ```
  To compress a file, press 1. To decompress a file, press 2: 2
  Enter the compressed string: 97 97 32 256 258 97 98 260 256 262 256 264 263 257 266 260 268 98 256
  Decompressed string: aa aa aab aaaab aaaab aaaa aaa aa aaaabaa
  ```

- String to compress: ”abcdef eghijklmnop qrstuvwxyz ABC.,-()DEF”

  Console output:

  ```
  To compress a file, press 1. To decompress a file, press 2: 1
  Enter the input string: abcdef eghijklmnop qrstuvwxyz ABC.,-()DEF
  Compressed output: 97 98 99 100 101 102 32 101 103 104 105 106 107 108 109 110 111 112 32 113 114 115 116 117 118 119 120 121 122 32 65 66 67 46 44 45 40 41 68 69 70
  ```

- String to decompress: ”97 98 99 100 101 102 32 101 103 104 105 106 107 108 109 110 111 112 32 113 114 115 116 117 118 119 120 121 122 32 65 66 67 46 44 45 40 41 68 69 70 ”

  Console output:

```
To compress a file, press 1. To decompress a file, press 2: 2
Enter the compressed string: 97 98 99 100 101 102 32 101 103 104 105 106 107 108 109 110 111 112 32 113 114 115 116 117 118 119 120 121 122 32 65 66 67 46 44 45 40 41 68 69 70
Decompressed string: abcdef eghijklmnop qrstuvwxyz ABC.,-()DEF
```