

**Sabanci University**  
**Faculty of Engineering and Natural Sciences**

**CS 300 Data Structures**

**Homework 2**

**Assigned: November 3, 2016**

**Due: November 15, 2016 @ 23:55**

**PLEASE NOTE:**

- **SOLUTIONS HAVE TO BE YOUR OWN. NO COLLABORATION OR COOPERATION AMONG STUDENTS IS PERMITTED.**
- **LATE SUBMISSIONS WILL BE PENALIZED 10 POINTS FOR EACH DAY OR PART OF A DAY DELAY. SUBMISSIONS THAT ARE LATE BY MORE THAN 3 DAYS WILL NOT GET ANY CREDITS.**
- **SUBMISSIONS WILL BE MADE TO THE SUCOURSE SERVER. NO OTHER METHOD OF SUBMISSION WILL BE ACCEPTED.**

**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 since 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, then 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. (Understand why this should be the case!) Nevertheless, compressors tend to work pretty well on the kinds of files that are typically found on computers, and they 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 most 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 1. 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 1):

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

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' ( $p = \text{"a"}$  and  $c = \text{'a'}$ ) 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" ( $p = \text{"aa"}$ ,  $c = \text{'b'}$ ) 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.

The decompression algorithm proceeds as follows: For decompression, we input the codes one at a time and replace them by 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:

- (i) **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,  $text(q)fc(p)$ ) into the dictionary.
- (ii) **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  $\text{text}(3) = \text{'aab'}$  is output to the decompressed file, and the pair  $(6, \text{text}(5)\text{fc}(3)) = (6, \text{'bbba'})$  is entered into the dictionary.
8. Finally, when the code 7 is encountered,  $(7, \text{text}(3)\text{fc}(3)) = (7, \text{'aaba'})$  is entered into the dictionary and 'aaba' output.

### Implementing the Ziv-Lempel Algorithm

We will use binary search trees as the basic dictionary data structure in the Ziv-Lempel compression algorithm. We will assume that our alphabet consists of 256 characters with values ranging from 0 to 255. The first 256 integer codes 0 to 255, correspond to these characters respectively. Since this is a very simple mapping, we do not have to store these; we will only store in the binary search tree, all strings with length 2 or more, along with their codes (which we know will start with 256), as we construct them during compression as discussed above.<sup>1</sup>

As you read new characters from the input file you create longer and longer strings which you search in the binary search tree. If you find any string (of length  $\geq 2$ ) in the tree but the next longer string is NOT in the tree, you do the following:

- 1) You print out the code associated with the longest string you found to the output file.
- 2) Then, unless you are at the end of the file, you create a string longer than the one you found using the next character read in (the character right after the string you found). You then associate the next available code (that you can keep track of with a counter) with the new string, inserting them to the tree.

Note once again that strings of length one will not be stored in the tree; you directly know what the codes for these are directly.

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 code. Please note that codes 0 through 255 will be used for single character strings, so the first code that YOU will assign will be 256. If during compression, you find that you need more codes, you do not generate new strings 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 character strings, so that, for instance, array position, 65 (which corresponds to ASCII symbol A) **points** to a 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 or a string of length 0 to detect that a string for a code is not yet composed.

---

<sup>1</sup> You can also assume that you will never encounter a byte with value 0 in the input, which may interfere with normal I/O on some platforms. Just assume you have an English text file.

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 set of class for the data objects that you will store in the binary search tree. You should figure out what this looks like and what methods you need. You can use the code for the binary search tree class provided in the textbook.

You will write two programs: The first one 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** after each integer.<sup>2</sup> Thus, you will not actually compress but print the sequence of integer codes for the file to be compressed. The compression program will read the input from a file called **compin** and will produce a file called **compout**.

The second program will perform decompression. It should be able to read the file **compout** and produce the file **decompout** if it is functioning correctly. Obviously the contents of **compin** and **decompout** should be equal.

You will turn in your documented code for any classes, and for the compression and decompression programs. Your code will be tested on the sample files we will use.

Your code should be submitted to SUCOURSE at the deadline given on the first page. You should follow the following steps:

- Name the two folders containing your source files as *XXXXX-NameLastname-compress* and *XXXXX-NameLastname-decompress* where *XXXXX* is your student number. Put all the related files for the two programs into the respective directories. Make sure you do NOT use any Turkish characters in the folder name. **You should remove any folders containing executables (Debug or Release), since they take up too much space.**
- Use the Winzip program to compress your folders to a compressed file named *05432-AliMehmetoglu.zip*. After you compress, please make sure it uncompresses properly and the uncompress should produce the two folders above.
- You then submit this compressed file in accordance with the deadlines above.

If the file you submitted is not labeled properly as above, if it does not uncompress, if the source code(s) do(es) not compile and if they do not work *for the input we will use*, as specified, we regretfully will have to give you 0 credits for the homework. So please make sure all these steps work properly.

---

<sup>2</sup> Actual compression will involve rather disgusting C++ code for dealing with details such as binary input output, bit packing, etc. which is really beyond the scope of this homework.