CS 2210a — Data Structures and Algorithms
Assignment 2: File Compression with Hash Tables

Due Date: 2013 October 15, 11:59 pm

Total marks: 20

1 Overview

For this assignment you are required to write a Java program that reads a file and compresses it using the algorithm described below. The *compression ratio* (defined as the size of the original file divided by the size of the compressed file) achieved by your program will depend on the input file.

You will be provided with a decompression program, which receives as input a compressed file produced by your program and gives as output the original, uncompressed file.

2 The Compression Algorithm

The input file could be a text file, an image file, an audio file, or any other kind of binary file, so we will consider that the input file is just a sequence of bytes. Each byte can be interpreted in Java as a character. For your reference, a table with all 8-bit characters and their ASCII codes is given in the course's website. An observation that you might find useful is that any integer between 0 and 255 can be stored in one byte and it can be converted into a character through casting. So, for example, `(char) 0` is the `NULL` character, `(char) 1` is the `SOH` character, and `(char) 97` is '*a*'. Similarly, an 8-bit character can be converted to an integer through casting; for example `(int)'a'` is 97.

The compression algorithm will map strings of characters into integer codes; the compressed file will store these codes. The algorithm uses a dictionary, that you have to implement using a hash table. The dictionary will store records of the form `(string, code)`, which are used during the compression process; the `string` field in each record is used as the key for that record. At the beginning, the algorithm will store in the dictionary the following 256 `(string,code)` pairs: $(s_0, 0), (s_1, 1), \dots (s_{255}, 255)$, where each string s_i, $i = 0, \dots, 255$, has length one and it contains the single character `(char)`i. Therefore, string s_0 contains the `NULL` character (as `(char)0 = NULL`), s_1 contains the `SOH` character (as `(char)1 = SOH`), s_{97} is the string "*a*" (as `(char)97 = 'a'`), and so on. The integer code associated with string s_i is the number i.

Once the dictionary has been initialized, the compression algorithm opens the input file and reads its contents one byte at a time. The algorithm repeatedly performs the following steps until all the input file has been processed:

1. Assign the value 256 to variable n_c (this is the smallest code that has not yet been assigned to any of the strings stored in the dictionary). Read the first byte from the input file and store it into variable p_b.

2. Read one by one bytes from the input file and append them to p_b until the **longest** sequence of bytes p_b is read whose corresponding string p (obtained by first converting each byte in p_b to a character and then concatenating all these characters into a string), is already stored in the dictionary. Get the record $(p, code_p)$ from the dictionary.

3. Write p's associated code, $code_p$, to the compressed file.

4. If all the input file has been processed, then terminate. Otherwise, read the next byte b and convert it to a character c.

5. Append character c to string p to form a new string p'. Assign to string p' the integer code n_c, and then insert the pair (p', n_c) in the dictionary. Clear the sequence p_b and then store b in p_b (so p_b contains a single byte). Increase the value of n_c and then go back to Step 2.

To clarify the way in which the algorithm works, let us try it on a text input file that contains the string "*aaabbbb*". Remember that the dictionary initially contains 256 pairs `(string,code)` for the 256 different bytes that might appear in the input file.

The algorithm reads the first byte from the input file and converts it to a character: '*a*'. Note that the longest string from the input that is already stored in the dictionary is "*a*". Hence, in Step 2 of the algorithm the pair ("*a*",97) is extracted from the dictionary, and in Step 3 the code 97 is written in the output file:

Input File	**Compressed File**	**Dictionary**
$\boxed{a}$ *aabbbb*	97	...
		("*a*",97)
		...
		("*b*",98)
		...

In the above figure there is a box surrounding the part of the input that has already been processed.

Since '*a*' is followed in the input file by another '*a*', then in Steps 4-5, the algorithm concatenates them to get the new string "*aa*". This string is assigned the smallest available code: 256 and then, the pair ("*aa*",256) is inserted in the dictionary:

Input File	**Compressed File**	**Dictionary**
$\boxed{a}$ *aabbbb*	97	...
		("*a*",97)
		...
		("*b*",98)
		...
		("*aa*",256)
		...

After step 5 the value of n_c is increased to 257 and p_b contains only the second byte of the input file. Observe that so far the algorithm has only output code for the first character '*a*' of the input.

In the second iteration of the algorithm p_b is converted to string $p =$"*a*"; since p is already in the dictionary the next byte, corresponding to the third '*a*', is read and appended to the end of p_b to get the string $p =$ "*aa*". This string "*aa*" is the longest string that is already stored in the dictionary, so the code 256 associated to "*aa*" is written to the output file.

Since "*aa*" is followed by the character '*b*', in Step 5 of the algorithm the string "*aab*" is assigned code 257, and the pair ("*aab*",257) is entered into the dictionary:

Input File	**Compressed File**	**Dictionary**
$\boxed{aaa}$ *bbbb*	97 256	···
		(“*a*”,97)
		···
		(“*b*”,98)
		···
		(“*aa*”,256)
		···
		(”*aab*”,257)
		···

At this point the algorithm has processed “*aaa*”, so the next character from the input that is read is ’*b*’; “*b*” is the longest string that is already stored in the dictionary, so its code, 98, is output. As “*b*” is followed by ’*b*’, the pair (“*bb*”,258) is entered in the dictionary in Step 5.

Input File	**Compressed File**	**Dictionary**	
$\boxed{aaab}$ *bbb*	97 256 98	···	··· |
		(”*a*”,97)	
		···	
		(”*b*”,98)	
		···	
		(”*aa*”,256)	
		···	
		(”*aab*”,257)	
		···	
		(”*bb*”,258)	
		···	

The algorithm has already processed “*aaab*” from the input. Hence, the longest string from the remaining input that is already in the dictionary is “*bb*”. The algorithm outputs its code, 258, and then the pair (“*bbb*”,259) is stored in the dictionary.

Input File	**Compressed File**	**Dictionary**
$\boxed{aaabbb}$ *b*	97 256 98 258	···
		(”*a*”,97)
		···
		(”*b*”,98)
		···
		(”*aa*”,256)
		···
		(”*aab*”,257)
		···
		(”*bb*”,258)
		···
		(”*bbb*”,259)
		···

The only character that has not been processed is the final ’*b*’; for this, code 98 is written to the compressed file and then he algorithm terminates.

Input File	**Compressed File**	**Dictionary**
aaabbbb	97 256 98 258 98	···
		("*a*",97)
		···
		("*b*",98)
		···
		("*aa*",256)
		···
		("*aab*",257)
		···
		("*bb*",258)
		···
		("*bbb*",259)
		···

Note that in the above example, for illustration purposes, the pairs `(string,code)` were shown in increasing order of code value in the dictionary, but this will not be the case in your program since you will implement the dictionary using a hash table.

2.1 Size of the Dictionary

For very long input files the number of pairs `(string,code)` that might be entered in the dictionary could be very large. In order to keep the size of the integer codes bounded, we impose a limit of 4096 on the maximum number of different records that can be stored in the dictionary. Hence, every code that the algorithm outputs to the compressed file will be 12 bits long (as $2^{12} = 4096$).

Important note. After your program has inserted 4096 records in the dictionary no more records can be added (as there are no more available integer codes). This is important, as otherwise your compressed file will not be decompressed correctly. If the dictionary already contains 4096 codes, then in Step 5 of the algorithm no new record will be inserted in the dictionary.

2.2 Writing Integer Codes to the Compressed File

To simplify your work, we provide a Java class, `MyOutput`, that you can use to write 12-bit codes in the output file. This class contains method

```
MyOutput.output(int code, BufferedOutputStream out);
```

that writes a 12 bit integer code to the output file `out`. Since it is only possible to write whole bytes to the output file, the above method might temporarily keep in a buffer 4 bits from some of the codes. These 4 bits are concatenated with the 12 bits of the next code to produce a 2-byte output. If this is confusing to you, you do not need to understand the explanation of how method `MyOutput.output` works, you just need to understand how to use it.

It is very important that before you close the output file you invoke method

```
MyOutput.flush(BufferedOutputStream out);
```

to send to the output file any bits that are still left in the buffer.

3 Classes to Implement

You are to implement at least 4 Java classes: `DictEntry`, `Dictionary`, `DictionaryException`, and `Compress`. You can implement more classes if you need to. You must write all the code yourself. You cannot use code from the textbook, the Internet, or any other sources. You **cannot** use the standard Java `Hashtable` class.

3.1 DictEntry

This class represents an entry in the dictionary, associating a string key with an integer code. For this class, you must implement all and only the following `public` methods:

- `public DictEntry(String key, int code)`: A constructor which returns a new `DictEntry` with the specified key and code.
- `public String getKey()`: Returns the key in the `DictEntry`.
- `public int getCode()`: Returns the code in the `DictEntry`.

You can implement any other methods that you want to in this class, but they must be declared as `private` methods (i.e. not accessible to other classes).

3.2 Dictionary

This class implements a dictionary. You must implement the dictionary using a hash table. You will decide on the size of the table, keeping in mind that at most 4096 items will be stored in it and the size of the table must be a prime number.

You must design your hash function so that it produces few collisions. (A bad hash function that induces many collisions will result in a lower mark.) Collisions must be resolved using separate chaining.

For this class, you must implement all the `public` methods in the following interface:

```
public interface DictionaryADT {
    public int insert (DictEntry pair) throws DictionaryException;
    public void remove (String key) throws DictionaryException;
    public DictEntry find (String key);
    public int numElements();
}
```

The description of these methods follows:

- `public int insert(DictEntry pair) throws DictionaryException`: Inserts the given `pair` (string,code) in the dictionary. This method must throw a `DictionaryException` (see below) if the key associated with `pair`, `pair.getKey()`, is already in the dictionary.

  You are required to implement the dictionary using a hash table with separate chaining. To determine how good your design is, we will count the number of collisions produced by your hash function. This method will return the value 1 if the insertion of object `pair` produces a collision, and it will return the value 0 otherwise. In other words, if for example your hash function is $h(k)$ and the name of your table is T, this method will return the value 1 if $T[h(\texttt{pair.getKey()})]$ already stores at least one element; it will return 0 if $T[h(\texttt{pair.getKey()})]$ is empty.
- `public void remove(String key) throws DictionaryException`: Removes the entry with the given `key` from the dictionary. Must throw a `DictionaryException` (see below) if the key is not in the dictionary.
- `public DictEntry find(String key)`: A method which returns the `DictEntry` object stored in the dictionary with the given `key`, or `null` if no entry in the dictionary has the given key.
- `public int numElements()`: A method that returns the number of `DictEntry` objects stored in the dictionary.

Since your `Dictionary` class must implement all the methods of the `DictionaryADT` interface, the declaration of your method should be as follows:

```
public class Dictionary implements DictionaryADT
```

You can download the file `DictionaryADT.java` from the course's website and place it in the same directory as your `Dictionary.java` class. This will allow the Java compiler to verify that your class implements all required methods.

The only other public method that you can implement in the `Dictionary` class is the constructor method, which must be declared as follows

```
public Dictionary(int size)
```

this returns an empty dictionary of the specified size.

You can implement any other methods that you want to in this class, but they must be declared as `private` methods (i.e. not accessible to other classes).

3.3 DictionaryException

This is just the class implementing the class of exceptions thrown by the `insert` and `remove` methods of `Dictionary`. See the class notes on exceptions.

3.4 Compress

This class contains the `main` method, declared with the usual method header:

```
public static void main(String[] args)
```

You can implement any other methods that you want to in this class, but they must be declared as `private` methods (i.e. not accessible to other classes).

The input to the program will consist of a file. The output will be the compressed file. To execute your program you must type

```
java Compress file
```

where `file` is the name of the input file. The compressed file must be stored in a file with the same name as the original one, but with extension ".zzz". For example if the name of the input file is `test` then the name of the compressed file must be `test.zzz`.

4 Testing your Program

We will perform two kinds of tests on your program: (1) tests for your implementation of the dictionary, and (2) tests for your file compressor. For testing the dictionary we will run a test program called `TestDict` which verifies whether your dictionary works as specified. We will supply you with a copy of `TestDict` so you can use it to test your implementation.

For testing your file compressor we will give your program some input files, check the sizes of the compressed files, and then we will decompress them and verify that the original files are restored. The Java program to decompress files is given in the course's website. Compile the program and then run it by typing:

```
java Decompress file
```

where `file` is the name of the compressed file **without** the "`.zzz`" extension. The program will produce a file with extension "`.unc`". You can use, for example, the `cmp` UNIX utility to compare the original file with the decompressed one. By typing

```
cmp file1 file2
```

the utility will report the first character where the two files differ, if any. If the files are identical, the utility terminates without producing any output.

5 Coding Style

Your mark will be based partly on your coding style.

- Variable and method names should be chosen to reflect their purpose in the program.
- Comments and indenting should be used to improve readability.
- No variable declarations should appear outside methods ("instance variables") unless they contain data which is to be maintained in the object from call to call. In other words, variables which are needed only inside methods, whose value does not have to be remembered until the next method call, should be declared inside those methods.
- All variables declared outside methods ("instance variables") should be declared `private` (not `protected`), to maximize information hiding. Any access to these variables should be done with accessor methods (like `getKey()` for `DictEntry`)

6 Hints

You can open a file for reading as follows:

```
BufferedInputStream in;
in = new BufferedInputStream(new FileInputStream(inputFile));
```

where `inputFile` is a String storing the name of the input file. The output file can be opened in a similar way:

```
BufferedOutputStream out;
out = new BufferedOutputStream(new FileOutputStream(outputFile));
```

To read one byte (character) from the input file you can use method `read()` from the `BufferedInputStream` class. The end of file is detected when the `read()` method returns the value -1. Note that this method returns an integer value, so you must cast it into a `char`.

7 Marking

Your mark will be computed as follows.

- Program compiles, produces meaningful output: 2 marks.
- `Dictionary` tests pass: 4 marks.
- `Compress` tests pass: 4 marks.
- Coding style: 2 marks.
- Hash table implementation: 4 marks.
- `Compress` program implementation: 4 marks.

8 Handing In Your Program

You are required to fill out and sign the Assignment Submission Form, which can be downloaded from

http://www.csd.uwo.ca/courses/CS2210a/submForm.html

Drop your submission form in the CS210 locker (locker 302 on the third floor of the Middlesex College Building, beside room MC 300) by the due date.

You must submit an electronic copy of your program. To submit your program, login to OWL and submit your java files from there.

Read the tutorials posted in the course's website to learn how to copy your program to your Gaul directory and how to test it there. Remember that the TA's will test your programs on Gaul. Please read also the tutorials on how to configure Eclipse to read command line arguments. Please do not put your code in sub-directories; this will make it easier for the markers.

When you submit your program, we will receive a copy of it with a datestamp and timestamp. If you submit your program more than once please send me an email message to let me know. We will take the latest program submitted as the final version, and will deduct marks accordingly if it is late.

After you submit your assignment you should receive an email message from the submissions system acknowledging the receipt of your work. **It is your responsibility** to ensure that your assignment was received by the system.