CS 2210 — Data Structures and Algorithms
Assignment 2: Compressing Files using Hash Tables
Due Date: February 22 at 11:55 pm
Total marks: 20

1 Overview

For this assignment you are required to write a Java program that compresses a file to reduce its size using the algorithm described below. You will be provided with an un-compression program, which receives as input a compressed file produced by your program and gives as output the original uncompressed file.

You need 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 or method `hashCode()`.

2 Class DictEntry

This class represents a record to be stored in a dictionary that you must implement using a hash table, as described in the next section. This class will have two instance variables: `key` of type `String`, and `code` of type `int`. Instance variable `key` is the key attribute of a `DictEntry` object.

For this class, you must implement all and only the following `public` methods:

- `public DictEntry(String theKey, int theCode)`: The constructor for the class that stores the values of the parameters in the corresponding instance variables.
- `public String getKey()`: Returns the value of instance variable `key`.
- `public int getCode()`: Returns the value of instance variable `code`.
- `public boolean isEqual(DictEntry secondObject)`: Returns true if `this` object and `secondObject` have the same `key` and `code` attributes; returns false otherwise.

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 Class Dictionary

This class implements a dictionary using a hash table **in which collisions are resolved using double hashing**. You must design your hash functions so that they produce few collisions. Remember that the primary and secondary hash functions **must be different**.

The secondary hash function cannot return the value 0; if it does your algorithm will get trapped in an infinite loop. Bad hash functions that induce many collisions will result in a low mark for you. This class must implement the provided Java interface `DictionaryADT.java`, so the header for this class must be

```
public class Dictionary implements DictionaryADT
```

The public methods that you must implement in this class are the following:

- `public Dictionary(int size)`: The constructor for the class. This method will create an empty hash table of the size specified in the parameter.

- `public int insert(DictEntry pair) throws DictionaryException`: Inserts `pair` in the dictionary. This method must throw a `DictionaryException` (see description of this class below) if the key attribute of `pair`, `pair.getKey()`, is already in the dictionary or if the dictionary is full so that `pair` cannot be inserted in it.

  As previously mentioned, you are required to implement the dictionary using a hash table with double hashing. To determine how good your design is, we will count the number of collisions produced by your hash functions. To that effect, **this method must return the value** 1 if the insertion of object `pair` produces a collision, and it **must return the value** 0 otherwise. In other words, if for example your hash function is `h(k)` and the name of your hash table is `T`, this method will return the value 1 if `T[h(pair.getKey())]` already stores one `DictEntry` object; it will return 0 if `T[h(pair.getKey())]` is null.

- `public void remove(String key) throws DictionaryException`: Removes the `DictEntry` object with the given `key` attribute from the dictionary. This method must throw a `DictionaryException` if no `DictEntry` object in the table has the given `key` attribute.

  To remove a `DictEntry` object from the hash table, you cannot just replace it with the value `null` as explained in the lectures. Instead you will replace with a special `DictEntry` object, that we will call `DELETED`, that has attributes `key = ""` and `code = -1`. Please review the lectures on hash tables to see how information is deleted from a hash table.

- `public DictEntry find(String key)`: Returns the `DictEntry` object stored in the dictionary with the given `key` attribute, or `null` if no entry in the dictionary has the given `key` attribute.

- `public int numElements()`: Returns the number of `DictEntry` objects stored in the dictionary.

You can implement any other methods that you want to in this class, but they must be declared as `private` methods.

4 Class DictionaryException

This is the class implementing the class of exceptions thrown by the `insert` and `remove` methods of class `Dictionary`. The constructor for this clas must receive as parameter a `String` message.

5 Class 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.

The input to the program is the file to be compressed. 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. If you wish to run your program in Eclipse you must set the input file as the command line argument. Please read the tutorials in OWL under the tab FAQ if you need help setting up Eclipse.

Your program must store the compressed file 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`.

5.1 The Compression Algorithm

The compression algorithm will store in a dictionary a set of `DictEntry` objects that will be used to compress the input file. The maximum number of `DictEntry` objects that will be stored in the dictionary is 4096. Therefore, the first thing that your program will do is to create a `Dictionary` of size larger than 4096. Remember that the size of the hash table must be a prime number. You must select for the size of your hash table a prime number larger than 4096, but not too large, say not larger than 10,000.

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. Recall that a byte consists of 8 bits, so there are 256 possible different bytes. Your program must initialize the hash table by storing in it 256 `DictEntry` objects, each one corresponding to a different byte.

Let the different bytes be $b_0 = 00000000$, $b_1 = 00000001$, $b_2 = 00000010, \ldots, b_{255} = 11111111$. Since each `DictEntry` object has a `String` and an `int` attribute, how can we convert a byte into a `String` and into an `int` so we can store the different bytes in the dictionary? Since a byte is just a sequence of 8 bits, it can be readily interpreted as an integer, so b_0 represents the number 0, b_1 represents the number 1, and b_{255} represents the number 255.

Converting a byte into a `String` is just a bit harder. A `char` in java is stored as 8 bits or a byte. Therefore, a byte can be converted into a `char` through casting. So, for example, `(char)`b_0 is the `NULL` character, `(char)`b_1 is the `SOH` character, `(char)`b_{97} is 'a', `(char)`b_{98} is 'b', and so on. An ASCII table contains the bit representation of each `char`; for your reference we posted an ASCII table in OWL.

Once we have converted a byte into a char, it is now easy to convert it into a `String` by using, for example, methods `String.valueOf(char ch)` or `Character.toString(char ch)`. Therefore, to initialize your hash table you could use a procedure like this:

for $i \leftarrow 0$ **to** 255 **do**
    Store in the dictionary a new `DictEntry` object with
    - `code` $= i$ and
    - `key` = `String` obtained from character `(char)`i.

The compression algorithm will map groups of bytes from the input file into integer codes; the compressed file will store these codes. The compression algorithm will open the input file and read it one byte at a time. You can open the input file as follows (Note that you do not have to open the input file as described here, you are free to open the file and read the bytes in it in any way that you like):

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

where `inputFile` is a String storing the name of the input file. To read one byte from the input file you can use method `read()` from the `BufferedInputStream` class; note that this method converts the byte read into an integer value. The end of file is detected when the `read()` method returns the value -1.

Once the dictionary has been initialized and the input file has been opened the compression algorithm works as follows.

1. Assign the value 256 to some variable, say, n_c (this value 256 is the smallest integer that has not been assigned as `code` attribute to any of the `DictEntry` objects stored in the dictionary). This variable n_c stores the code of the next `DictEntry` object that will be inserted in the dictionary.

2. Read the first byte from the input file, convert it to a `String` and store it in some variable, say, s.

3. Repeatedly perform the following steps until the end of the input file has been reached or a string s is found that it is not the `key` attribute of any of the `DictEntry` objects stored in the dictionary:

   - Read the next byte b from the input file
   - Convert b to a character c.
   - Append c to s.

4. If no `DictEntry` object in the dictionary has key s, then let s' to be the string containing all characters from s except the last one; otherwise, set $s' = s$.

5. Find the `DictEntry` object d with `key` attribute equal to s'; let k be the integer code attribute of d.

6. Write k into the compressed file as specified in Section 5.3.

7. If all the input file has been processed, then close the input file and terminate.

8. If the number of `DictEntry` objects stored in the dictionary is less than 4096, then create a new `DictEntry` object d' with `key` attribute s and `code` attribute n_c and insert d' into the dictionary.

9. Delete from s all its characters, except the last one (so now string s has only one character).

10. Increase the value of n_c and then go back to Step 3.

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 `DictEntry` objects corresponding to 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 the string $s =$ `"a"`. The next character `'a'` is read and appended to s to form the string `"aa"`. Since `"aa"` is not in the dictionary, then in Steps 4 and 5 the algorithm creates the string $s' =$ `"a"` and gets from the dictionary the `code` attribute 97 of the `DictEntry` object with `key` attribute `"a"`. In Step 6, the algorithm writes the code 97 into the output file:

Input File	**Compressed File**	**Dictionary**
[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 there are fewer then 4096 object in the dictionary a new `DictEntry` object with `key` attribute $s =$ `"aa"` and `code` attribute $n_c = 256$ is inserted in the dictionary.

Input File	**Compressed File**	**Dictionary**
`a` `aabbbb`	97	···
		(`"a"`,97)
		···
		(`"b"`,98)
		···
		(`"aa"`,256)
		···

In Steps 9 and 10 s takes value `"a"` and n_c is increased to 257. Observe that so far the algorithm has only output code for the first character `'a'` of the input file.

In the second iteration of Step 3, bytes are read from the input file and they are converted to characters which are appended to string s until the string `"aab"` is formed, since `"aab"` is not in the dictionary. String s' is now `"aa"` and the `DictEntry` object with key s' has code $k = 256$, hence the value 256 is written to the output file. A new `DictEntry` object with key s and code 257 is then added to the dictionary. Variable s then takes value `"b"` and n_c is increased 258.

Input File	**Compressed File**	**Dictionary**
`aaa` `bbbb`	97 256	···
		(`"a"`,97)
		···
		(`"b"`,98)
		···
		(`"aa"`,256)
		···
		(`"aab"`,257)
		···

At this point the algorithm has processed the input `"aaa"`. In Step 3 only one character is appended to s to get $s =$ `"bb"` as `"bb"` is not in the dictionary. The `DictEntry` object in the dictionary with key $s' =$ `"b"` has code 98, so the value 98 is written to the output file. Then a `DictEntry` object with key `"bb"` and code 258 is inserted in the dictionary. Finally, s takes value `"b"` and $n_c = 259$.

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

So far the algorithm has already processed `"aaab"` from the input. In Step 3 string s takes value `"bbb"`. The `DictEntry` object in the dictionary with key $s' =$ `"bb"` has code 258, so this value 258 is written to the output file. A `DictEntry` with key `"bbb"` and code 259 is added to the dictionary, s is set to `"b"` and $n_c = 260$.

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

The only character from the input file that has not been processed is the final `'b'`. In Step 3 the algorithm does not change s as the end of the input file has been reached. Therefore $s' = s$ and the code 98 is written to the output file. Finally, the output file is closed and the algorithm terminates. The 7 character input file has been compressed to a sequence of 5 integers.

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 `DictEntry` objects 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.

5.2 Size of the Dictionary

For very long input files the number of `DictEntry` objects `(string,code)` that are inserted 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 `DictEntry` objects that can be stored in the dictionary. Hence, every code that the algorithm writes 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 8 of the algorithm no new data will be inserted in the dictionary.

5.3 Writing Integer Codes to the Compressed File

The output file can be opened as follows:

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

where `outputFile` is the name of the output 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 a file, the above method might temporarily keep in a buffer 4 bits from a code. 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.

6 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 checks whether your dictionary works as specified. We will supply you with a copy of `TestDict` so you can use it to test your implementation. Note that `TestDict` cannot perform an exhaustive testing of your implementation. You are expected to run additional tests on your own to ensure that your program works correctly.

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 to you. 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 or the `fc` windows program to compare the original file with the decompressed one. For example, 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.

7 Coding Style

Your mark will be based partly on your coding style.

- Variable and method names should reflect their purpose in the program.
- Comments and indenting should be used to improve readability.
- No instance variables should be used unless they contain data which needs to be maintained in an object from call to call to tis methods. 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 instance variables should be declared `private`, to maximize information hiding. Any access to these variables should be done with accessor methods (like `getKey()` for `DictEntry`).

8 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.

9 Handing In Your Program

To submit your program, login to OWL and submit your java files from there. **Please make sure you submit your .java files and not your .class files**. DO NOT compress your files. DO NOT add a `package` statement at the top of your java classes.

If you re-submit your program we will only mark the latest version and will deduct marks accordingly if it is late.