COMP 2150 - Winter 2020 - Assignment 4

Due April 6, 2020 at 11:59 PM

Updated Mar 19 - replace word "raise" with "throw".

The assignment has two parts. Each part is worth half of the total marks for the assignment. The entire assignment is in javascript.

Part 1: Hash Table Data Structure

Create a javascript class called HashTable that implements a hash table. When initialized, the HashTable should be given a size as a parameter. The HashTable will create an array of that size as a field. When adding or searching for an element in the HashTable, the position of the element in the table will be its hashVal modulo the length of the array (see below for hashVal information).

The HashTable should use separate chaining. If there is a collision between two elements based on their hash code, the HashTable should place both elements in a linked list of elements at the array position. The order of elements in the linked list at a position is not important.

The HashTable should implement the following methods:

- `add(x)`: takes a Hashable object x and adds it to the HashTable in the appropriate position (see below for information on the Hashable class). There is no return value for the method.
- `get(x)`: takes a Hashable object x and returns the first occurrence **equal to** x in the hash table. If the element does not exist, the method should throw an error.
- `remove(x)`: takes a Hashable object x and deletes one occurrence of an object equal to x in the hash table. If the element does not exist, the method should leave the table unchanged. If several elements are equal to x, the method can delete any of them. The method should return a boolean that indicates whether an element was removed or not.
- `contains(x)`: takes a Hashable object x and determines if it is in the hash table. Returns a boolean value.
- `isEmpty()`: returns a boolean value determining whether the hash table is empty or not.

For each of the first four methods, the method should ensure that the parameter has type Hashable (**or** has a hashVal() method, and an equals() method, as appropriate), and it should throw an error if the condition does not hold.

A note on the get() method: you may ask "why does it return the same thing as it's searching for?" Answer: **it's not.** It's searching for the first object that's **equal to** the parameter, which

depends on the definition of equality for the method. It is possible that they are not the same objects, or even have the same contents (see KeyValueHash below).

Hashable

You should also implement a hierarchy of Hashable classes that can be stored in the HashTable. The Hashable class should be abstract (using the techniques shown in class) with an abstract hashVal() and an abstract equals(x) method. You should implement three Hashable subclasses:

- IntHash, whose hash function is the value of the integer. Two IntHash objects are equal if they contain the same integer value.
- StringHash, whose hash function is the following expression:

  s[0]*p^(n-1) + s[1]*p^(n-2) + ... s[n-2]*p + s[n-1]

  where s[i] is the ASCII value of the i-th character of the string, n is the length of the string, and p is a prime (use a prime number less than 1000 of your choice). You can assume that all characters in the strings are ASCII characters. Two StringHash objects are equal if they contain the same strings.

- KeyValueHash, which stores two values, a key and a value. The hashVal for a key-value pair is the hashVal of the key, which must be a Hashable type. Two KeyValueHash objects should be considered equal if they have the same **key.**

The subclasses can have additional methods if you need them, including getters.

Unit Testing

Construct a set of at least **five** different unit tests for your HashTable. To do unit testing in javascript, you should simply write a separate file of tests that can be run (i.e., you should not need to use a testing framework like jest or mocha -- do not assume the markers will have either installed on their system.)

1. Create a test file with "`let assert = require('assert');`" near the top of the file.
2. Write a file with a series of tests in whatever format you would like. Each test should be its own function.
3. Write a main function that calls all of the test methods.
4. Have a single executable line that calls the main function.
5. In your tests, say assert([something]); to give assertions. You need at least one assertion per test method.

At least two of your tests should test boundary conditions: an empty HashTable, and a HashTable with one element. You will be graded on your tests, so write useful tests for all four unit tests.

Save all of your unit tests as part of a single file and submit them with your code. The markers will be running your tests, so ensure that they pass. **Give instructions on how to run them in your readme file.** You are strongly advised to complete the unit tests before starting Part 2.

Part 2: Dictionary and LZW Encoding

Dictionary

Using your HashTable data structure from Part 1, create a dictionary type called Dictionary. The dictionary type must use containment (not inheritance) when using the HashTable.

In the Dictionary, each entry is a Key-Value pair, and there can't be two Key-Value pairs in a dictionary with the same Key (that is, keys are unique in a dictionary). The dictionary should have the following operations:

1. `put(k,v)`: takes a Hashable key k and a value v and adds it to the dictionary, if it does not exist. If a key-value pair with k as the key already exists, the current value is replaced with v. The method does not have a return value.
2. `get(k)`: takes a Hashable key k and returns the value v associated with it, if it appears in the dictionary. Throws an error if the key k is not in the dictionary.
3. `contains(x)`: takes a Hashable key x and determines if it is a key in the dictionary. Returns a boolean value.
4. `isEmpty()`: returns a boolean depending on whether the dictionary is empty or not.

When created, your dictionary should create a HashTable of a fixed size (1000). You do not need to provide unit testing for your dictionary, but of course you are encouraged to do so.

LZW Encoding

Compression is the process of taking a file and using information about the file ("what letters and phrases are more common?") to create a smaller file with the same information. The smaller file (the compressed file) can be decompressed to recover the original file. zip files are created by compression.

Using your Dictionary, you will implement LZW encoding, a compression algorithm. LZW encoding compresses text files by replacing portions of the text with a single integer. As the LZW encoder runs on more and more text, the portions of text replaced with a single integer grow longer and longer.

Before encoding any text, create a dictionary that associates numerical values with each single character. For this assignment, you will be encoding printable ASCII characters in your dictionary as follows:

Character	LZW Encoding value
[space]	0
!	1
"	2
...	...
A	33
B	34
...	...
Z	58
...	...
a	65
...	...
z	90
...	...
~	94

That is, all ASCII characters with values from 32 to 126 should be encoded as **their ASCII value minus 32**. To convert from an integer to a character representation in javascript, use the String.fromCharCode() method, which converts an ASCII value to a character (e.g., String.fromCharCode(33) is a string '!'). **Note** that the newline character (\n) is **NOT** in the list of characters above, so there will be no newlines in the input for this algorithm.

Next, the LZW algorithm looks at the text file and encodes successive portions of the file into an integer according to this pseudocode:

```
1. curr_key = next unseen char from the file
2. while curr_key is in the dictionary:
3.     last_key = curr_key
4.     curr_key += the next char from the file
```

```
5. add curr_key to the dictionary with a new LZW
   encoding value
6. output the value in the dictionary for last_key
7. set the last character of curr_key as the next
   starting point for encoding (i.e., the last character
   of curr_key is the new curr_key next time the
   pseudocode is run)
```

Repeat this code until all of the characters in the file have been considered. Finally, output a stop code (in our case, the stop code will be -1) after all input has been processed. See an example run of the LZW encoding below. (To run this pseudocode multiple times, you can manage your position in the input file with an integer index.)

Your Encoder class should have a constructor and one method called encode(). The constructor should take the name of the file that should be encoded. The encode() method should open a new output file (by adding the ".lzw" extension to the original file name), open the input file, and write the codes for the LZW algorithm to the output file. (You should write the values as **integers and separated by a space**. Do not output any newline symbols. Yes, the output file will be larger than the input file in this implementation.)

You should read the filename from the command-line arguments. Use process.argv (an array) to get the command-line arguments when running node.js.

You do not need to implement a decoder. A LZW decoder will be provided for you shortly before the assignment due date.

Example LZW encoding

Consider the string "TOBEORNOTTOBETHATISTHEBANANA". The LZW algorithm finds the following values of curr_key and last_key at each step. The output (an integer) and the modifications to the dictionary are also shown.

1.	curr_key = "TO",	last_key = "T".	Output = 52,	Add to dictionary: TO:95
2.	curr_key = "OB",	last_key = "O".	Output = 47,	Add to dictionary: OB:96
3.	curr_key = "BE",	last_key = "B",	Output = 34,	Add to dictionary: BE:97
4.	curr_key= "EO",	last_key = "E",	Output = 37,	Add to dictionary: EO:98
5.	curr_key="OR",	last_key = "O",	Output = 47,	Add to dictionary: OR:99
6.	curr_key="RN",	last_key = "R",	Output = 50,	Add to dictionary: RN:100
7.	curr_key="NO",	last_key = "N" ,	Output = 46,	Add to dictionary: NO:101
8.	curr_key="OT",	last_key = "O" ,	Output = 47,	Add to dictionary: OT:102
9.	curr_key="TT",	last_key = "T" ,	Output = 52,	Add to dictionary: TT:103
10.	curr_key="TOB",	last_key = "TO" ,	Output = 95,	Add to dictionary: TOB:104
11.	curr_key="BET",	last_key = "BE" ,	Output = 97,	Add to dictionary: BET:105
12.	curr_key="TH",	last_key = "T" ,	Output = 52,	Add to dictionary: TH:106
13.	curr_key="HA",	last_key = "H" ,	Output = 40,	Add to dictionary: HA:107

14. curr_key=”AT”, last_key = “A” , Output = 33, Add to dictionary: AT:108
15. curr_key=”TI”, last_key = “T” , Output = 52, Add to dictionary: TI:109
16. curr_key=”IS”, last_key = “I” , Output = 41, Add to dictionary: IS:110
17. curr_key=”ST”, last_key = “S” , Output = 51, Add to dictionary: ST:111
18. curr_key=”THE”, last_key = “TH” , Output = 106, Add to dictionary: THE:112
19. curr_key=”EB”, last_key = “E” , Output = 37, Add to dictionary: EB:113
20. curr_key=”BA”, last_key = “B” , Output = 34, Add to dictionary: BA:114
21. curr_key=”AN”, last_key = “A” , Output = 33, Add to dictionary: AN:115
22. curr_key=”NA”, last_key = “N” , Output = 46, Add to dictionary: NA:116
23. curr_key=”ANA”, last_key = “AN” , Output = 115, Add to dictionary: ANA:117
24. curr_key=”A**[EOF]**”, last_key = “A”, Output = 33,
25. output stop code (-1).

Javascript Notes and Hints

1. You must "use strict" in all of your files.
2. You are not required to use #private fields in the assignment, but all fields should be **treated as private.** This means using the tools we have used throughout the term: e.g., proper encapsulation, implementation independent data structures, getters, setters if necessary, and operations on data structures rather than releasing a whole data structure.
3. Use fs.writeFileSync, fs.appendFileSync and fs.readFileSync for file manipulation.
4. Use the substring method on strings to extract a portion of a string, like in Java.
5. Use process.argv (an array) to get command line parameters in javascript.

Hand-in

Submit all source code for all classes, including the unit tests. Submit all files on D2L.

You will be provided a large test file for your encoder before the deadline. Submit your output on that file with your submission.

You **MUST** submit all of your files in a zip file. Additionally, you **MUST** follow these rules:

- All of the files required for your project should be in a single directory.
- Include a README.TXT file that describes exactly how to run your code from the command line. The markers will be using these instructions exactly and if your code does not run as described, you will lose marks.

The easier it is for your assignment to mark, the more marks you are likely to get. Do yourself a favour.