**CSCI 3302 Programming Assignment 07 (100 Points)**

**Due: Dec 8, 8:00 AM**

**GITHUB Link:** [**Program 07**](https://classroom.github.com/a/DkxgR4Ex)

**Objectives:**

* Demonstrate understanding of a hash table.
* Demonstrate how to implement operations on a hash table.
* Demonstrate handling of hash table collisions.

**Assignment Assistance:**

* This homework assignment is due prior to the date and time specified above.
* This assignment is restricted to individual effort. As per our syllabus, the use of AI is prohibited. You may not receive help from any other person except the instructor or the AARC (help from the AARC must be well documented!).
* Any resource used (other than Dr. Becnel or the course text) must be documented in the code (as comments) detailing the source and describing exactly what was learned and how that information was used. Submissions will be severely penalized if copied in part or in whole from any source.
* If you need help, visit your instructor during his posted office hours. If your schedule cannot accommodate any of these times, then email your instructor to schedule a different time.

**Problem Description:**

1. A Dictionary ADT (also called an associative array) is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection. Usually, the Dictionary ADT is implemented as a hash table because a hash table provides near-constant time operations for adding, removing, and retrieving values.
2. In this assignment, you will implement a Dictionary ADT using an array-based hash table.
   1. You will write the HashTable class in a file called HashTable.java.
   2. Your hash table should be implemented as a jagged array of Node objects.
   3. In each (key, value) pair, the key is of type int and the value is of type String.
   4. Each (key, value) pair is stored in an object of type Node. A listing for the Node class can be found in Appendix B. The provided code should not be modified.
   5. Your class should contain an attribute called table which is the array used for storing (arrays of) Node objects.
   6. Your class should define a constant of type int called NUM\_BINS. The NUM\_BINS constant determines the number of bins in the hash table (the size of the array). This value should be prime. In your submission, this value should be set to 101.
   7. Your class should implement the IDictionary interface found in Appendix A. The provided code should not be modified.
   8. Notice that this implementation does not use generics. Working with an array of generic types in Java is very difficult and beyond the scope of this assignment.
3. The methods that you are required to implement should, for the most part, be self-explanatory from examining the IDictionary interface and the Node class. The comments within the code listings provide some guidance. Additional guidance for each part follows:
   1. Your add , remove, and retrieve methods should use a private getHashCode method to hash the integer value of the key (not of the Node object).

* You will need to then convert the key value into an integer between 0 and NUM\_BINS – 1 to specify the index of where in the hash table array to store the string by using modulo (as shown in class).
  1. Each (key, value) pair will be stored in a Node object. For the add method, Node objects will be stored within the appropriate bucket at the correct location within the hash table.
  2. Collisions can result when different keys hash to the same bin. For the add method, collisions will be handled using **buckets**. Thus, each bin will contain an array of Node objects.
* Each bucket should start off with a length of two (2). If the bucket is full, then a new bucket should be created that is twice as large and all of the Node objects from the old bucket should be copied into the new bucket.
* If the bucket length is greater than two, and a remove operation results in the array being less than ¼ full, then a new bucket should be created that is half as large as the existing bucket, and all of the Node objects from the old bucket should be copied into the new bucket.
* Notice that, in these cases, there is no need to recalculate the hash values since the size of the hash table (number of bins) is not changed.
  1. The keys should be unique for every (key, value) pair. Thus, in the add method, the bucket found in the appropriate bin should be searched to determine if a (key, value) pair already exists for the given key.
* If the key of the (key, value) pair being added to the hash table is identical to the key of a (key, value) pair already located within the bin, then the old (key, value) pair should be overwritten. That is, the old pair is removed and the new (key, value) pair is added.
* If a collision occurs, but the keys are unique, then the new (key, value) pair is added to the bin with no (key, value) pair being removed.

1. To make sure that your implementation is working correctly, you need to create a DictionaryException class as an extension of a RuntimeException. This exception needs to be thrown if any attempt is made to access any (key, value) pair not contained in the dictionary (key is not found).
2. Your program should work in the GitHub codespace (Linux environment) and locally (Windows environment).
3. You may write any private helper methods, if needed.

**Hints:**

* While testing your code, it is recommended that you make the size of your array (NUM\_BINS) fairly small (e.g., 11) to determine if your code is working correctly. However, your submitted code should assign NUM\_BINS = 101.
* For this programming assignment, you do not dynamically resize the hash table array. The hash table is never ‘full’, but as more and more items are stored in a particular bin, the time complexity for the add , remove, and retrieve methods increase.
* Your hash calculation should result in a relatively uniform distribution of index values. Thus, all strings should not end up in a small portion of the bins. However, the number of collisions is also dependent on the actual key values used, so it is impossible to guarantee that all bins will contain a single element before the first collision.
* A toString() method is in the starter code for your convenience. However, I do not recommend using it for a large number of bins.
* No tests are provided for you. Use what you learned this semester to test your program. You are welcome to use JUnit testing (like in previous assignments) or a simple test program that makes use of a main method as we have done in some paired programming assignments and in class. Include your test file in your submission.

**Submission:**

* Review the Evaluation below to ensure you have met all the requirements.
* Commit DictionaryException.java, HashTable.java, IDictionary.java, Node.java to GitHub. Upload a backup copy to D2L. Your submitted files should NOT contain a main method or testing code. You may include testing files in your repository; however, these will not be considered when grading. If you wish to include non-working code for insight into your thought process, make sure to contain it within comment blocks and ensure that submission successfully compiles.

**Evaluation**

|  |  |
| --- | --- |
| * **Automatic Deductions:** |  |
| Late/Not Submitted | -100 |
| Code not submitted to GitHub | -30 |
| Code does not run/compile | -50 |
|  |  |
| **Earn Points for the following:** |  |
| Code has comment header with name, section, date | 5 pts |
| Code organization, structure, and indention is appropriate (SHFT + ALT + F in VS Code) | 5 pts |
| Code is well and meaningfully commented. | 5 pts |
| Appropriate variable and method names that follow Java conventions | 5 pts |
| Instructions correctly followed for fields, class, methods | 10 pts |
| isEmpty and clear | 10 pts |
| add, remove, retrieve | 20 pts each |

**Appendix A: The interface used for this assignment:**

public interface **IDictionary** {

    // Determines whether the dictionary is empty.

    public boolean isEmpty();

    // Adds the provided (key, item) pair to the dictionary.

    // If the key is already associated with a value, the

    // old key, value pair are replaced.

    public void add(int key, String value);

    // Removes the (key, value) pair specified by the given

    // key from the dictionary.  Throws an exception if the

    // (key, value) pair is not contained in the dictionary.

    public void remove(int key) throws DictionaryException;

    // Returns the value associated with the given key

    // from the dictionary. Does not modify the dictionary.

    // Throws an exception if the (key, value) pair is not

    // contained in the dictionary.

    public String retrieve(int key) throws DictionaryException;

    // Removes all (key, value) pairs in the dictionary.

    public void clearDictionary();

}

**Appendix B: The implementation of the Node class used for this assignment:**

// Node class to store a key of type int and

// a value that is also of type String.

public class **Node** {

    // Attributes

    private int key;

    private String value;

    // Constructor for a node.

    // Call setter to assign next.

    public Node(int newKey, String newValue) {

**this**.key = newKey;

**this**.value = newValue;

    }

    // Getter for key within the node.

    public int getKey() {

        return **this**.key;

    }

    // Getter for value within the node.

    public String getValue() {

        return **this**.value;

    }

} // end class