COL 106 : Data Structures and Algorithms
Semester II, 2022-23, IIT Delhi

Assignment - 2 (due on 03rd February, 11:00 PM)

Important Guidelines:

- You must ensure that your submitted code runs in the JDK 11.0.17 environment.
- You are not allowed to share your code with anyone. Cheating of any form will lead to strict disciplinary action. Typical penalty would include Fail grade in the course.
- The assignment must be done individually, and no group submissions are allowed.
- All starter code files must be included in the submission, which must be uploaded on the gradescope without compressing the files.
- The names of files and method signatures must not be changed in the starter code.
- You should write the code in the portion of the starter code labelled **"To be filled in by the student"**.

1 Polynomial Implementation using Linked Lists

1.1 Creating a Linked List for representing Polynomials

In this part of the assignment, you will use linked lists for implementing basic polynomial operations such as addition and multiplication. Each node of the linked list will be used to represent an individual term of the polynomial.

The Linked List will have the following features:-

1. Each node in the linked list will contain a single integer representing the coefficient of the polynomial term
2. The nodes of the linked will be arranged in the decreasing order of the exponent of the term.
3. Each node will contain a reference to the next node.

Your task is to implement the 'insert' and the 'len' function in the file **LinkedList.java**. You can create other helper functions if you like, but don't change the signature of the given functions.

1. The **insert** function takes an integer as a parameter(representing the coefficient of a term of the polynomial) and inserts the integer at the end of the linked list.
2. The **len** function should return the length of the linked list.

The given functions will be called in the following manner:-

```
TestCase1:-
LinkedList l = new LinkedList();
l.insert(2);
l.insert(0);
l.insert(7);


Final Linked List:-
 2 -> 0 -> 7
```

The above testcase represents the polynomial $2x^2 + 7$

```
TestCase2:-
LinkedList l = new LinkedList();
l.insert(-7);
l.insert(0);
l.insert(0);
l.insert(15);
l.insert(0)

Final Linked List:-
 -7 -> 0 -> 0 -> 15 -> 0
```

This testcase represents the polynomial $-7x^4 + 15x$

NOTE:- Proceed to the next part of the question only if you have completed this part. We will be using your implementation of the Linked List for the next part of the question.

1.2 Addition and multiplication of polynomials

In this part, you will have to perform addition and multiplication of two polynomials(which you should have learned in school level algebra) given in the form of Linked Lists and return the final Linked Lists representing the resultant polynomial.

You will have to implement two functions 'add' and 'mult' in the file **Polynomial.java**. Again, you can create other helper functions, but don't change the already existing function signatures.

1. **add**:- For two polynomials p1 and p2, the usage of add function is:-

   Polynomial sum = p1.add(p2);

   This function adds the two polynomials p1 and p2 represented as Linked Lists and returns the polynomial representing their sum.(Please note that this function will have only parameter, the polynomial p2, which needs to be added to the polynomial in which this function is called).

2. **mult**:- For two polynomials p1 and p2, the usage of mult function is:-

   Polynomial prod = p1.mult(p2);

   This function multiplies the two polynomials p1 and p2 represented as Linked Lists and returns the polynomial representing their product.(Please note that this function will have only parameter, the polynomial p2, which needs to be multiplied to the polynomial in which this function is called).

The functions will be called as follows:-

```
TestCase1:-
First Linked List:-
1 -> -1
Second Lined List:-
1 -> 1


Return List for add:-
2 -> 0
Return List for mult:-
1 -> 0 -> -1
```

The two given linked list represents $x-1$ and $x+1$. Their summation should be $2x$ and multiplication should be x^2-1.

```
TestCase2:-
First Linked List:-
3 -> 1 -> 0 -> 2
Second Lined List:-
2 -> 1


Return List for add:-
3 -> 1 -> 2 -> 3
Return List for mult:-
6 -> 5 -> 1 -> 4 -> 2
```

In this case the two polynomials are $3x^2+x^2+2$ and $2x+1$. Their summation should be $3x^3+x^2+2x+3$ and product should be $6x^4+5x^3+x^2+4x+2$.

2 Implementing Dictionary

A Dictionary is a list of Key-Value pairs, such that each key has a single associated value. And when presented with a key the dictionary returns the associated value. A dictionary uses a hash-table to keep a record of addresses where the associated values are stored in the linked-list, to make lookup time complexity $O(1)$.

Using the Java's in-built Linked-List (`java.util.LinkedList`) and Array, build a Generic Dictionary by defining the following methods in `COL106Dictionary.java`:-

1. `void insert(K key, V value) throws KeyAlreadyExistException, NullKeyException`:*
   Inserts the key-value pair in the dictionary and doesn't return anything.
   NOTE: Insertion in the Dictionary implies that the key-value pair will be inserted into the list and also the address of the node will be stored in the hash-table.

   Input Format:
   The method will be argumented with a key of type K and a value of type V.

   Return Format:
   Nothing is to be returned by this method.

   Example:
   Consider the following dictionary where hash-table of size `7` is already built. We next insert a new key-value pair `k, val` into the dictionary, where `k` is a string and `val` is an integer.

   Linked-list of key-value pairs:
   ```
   Head -> (0x123121)["Hello", 2] -> (0x0019ff)["COL106", 5] ->
   (0x34ef17)["IITD", 10] -> null
   ```

   Hash-table:
   ```
   0 | Head -> null
   1 | Head -> ["IITD", 0x34ef17] -> ["COL106", 0x0019ff] -> null
   2 | Head -> null
   3 | Head -> null
   4 | Head -> ["Hello", 0x123121] -> null
   5 | Head -> null
   6 | Head -> null
   ```

   NOTE:- Above Hash-table is just for illustration and not the actual hash-table that will be created. The actual hash table will depend on the hash function.

   Input:

```
["Assignment2", 9]
```

Suppose the hash of string `"Assignment2"` is `4`.

Then, the Dictionary after the insertion will be as follows:

Linked-list:

```
Head -> (0x098fd1)["Assignment2", 9] -> (0x123121)["Hello", 2] ->
(0x0019ff)["COL106", 5] -> (0x34ef17)["IITD", 10] -> null
```

Hash-table:

```
0 | Head -> null
1 | Head -> ["IITD", 0x34ef17] -> ["COL106", 0x0019ff] -> null
2 | Head -> null
3 | Head -> null
4 | Head -> ["Hello", 0x123121] -> ["Assignment2", 0x098fd1] ->
null
5 | Head -> null
6 | Head -> null
```

2. `V delete(K key) throws KeyNotFoundExecption, NullKeyException`:* Deletes the key and associated value pair from the dictionary and returns the associated value.

Input Format:
The method will be argumented with a key of type `K`.

Return Format:
The value of type `V` corresponding to the argumented key is to be returned.

Example:
Consider the following dictionary where a hash-table of size is `7` is already built. We next delete an element from the Dictionary.

Linked-list:

```
Head -> (0x098fd1)["Assignment2", 9] -> (0x123121)["Hello", 2] ->
(0x0019ff)["COL106", 5] -> (0x34ef17)["IITD", 10] -> null
```

Hash-table:

```
0 | Head -> null
1 | Head -> ["IITD", 0x34ef17] -> ["COL106", 0x0019ff] -> null
2 | Head -> null
3 | Head -> null
```

```
4 | Head -> ["Hello", 0x123121] -> ["Assignment2", 0x098fd1] ->
null
5 | Head -> null
6 | Head -> null
```

NOTE: Above Hash-table is just for illustration and not the actual hash-table that will be created. The actual hash table will depend on the hash function.

Input 1:
`["COL106"]`
In the example, the hash of string `"COL106"` is `1`, so will we go to index `1` of hash-table and start searching for `"COL106"` in the linked list. We will next go to address of node of linked-list corresponding to key `"COL106"`. Finally, we will delete the node from linked-list as well as remove the corresponding entry from the Hash-table.
Hence, the Dictionary after the deletion will be as follows:

Linked-list:
```
Head -> (0x098fd1)["Assignment2", 9] -> (0x123121)["Hello", 2] ->
(0x34ef17)["IITD", 10] -> null
```

Hash-table:
```
0 | Head -> null
1 | Head -> ["IITD", 0x34ef17] -> null
2 | Head -> null
3 | Head -> null
4 | Head -> ["Assignment2", 0x098fd1] -> ["Hello", 0x123121] ->
null
5 | Head -> null
6 | Head -> null
```

And before deletion we found that the value corresponding to key "COL106" was 5. So,

Return 1:
`5`

Input 2:
`["COL100"]`
Let the hash of string `"COL100"` be `3`, and since index `3` of hash-table is `null`, we will throw exception.

Return 2:
`KeyNotFoundExecption`

3. `V update(K key, V value) throws KeyNotFoundExecption, NullKeyException`:* Updates the value associated with the argumented key with the arguemented value and returns the previously stored value associated with the argumented key.

4. `V get(K key) throws KeyNotFoundExecption, NullKeyException`:* Returns the value associated with the argumented key.

5. `int size()`:* Returns the number of key-value pairs stored in the dictionary.

6. `K[] keys()`:** Returns array of keys stored in dictionary.

7. `V[] values()`:** Returns array of values stored in dictionary.

8. `long hash(K key)`: Returns the hash of the argumented key using the polynomial rolling hash function.
   NOTE: To use the polynomial rolling hash function, convert the type of `key` i.e. `K` to String say `s`, and perform the following mathematical equation to compute the hash.

$$hash(s) = s[0] + s[1].p + s[2].p^2 + ... + s[n-1].p^{n-1} \mod m$$

   or we can say,

$$hash(s) = \sum_{i=0}^{n-1} s[i].p^i \mod m$$

   where, $s[i]$ =ASCII-value($s[i]$) + 1, m will be the size of your hash-table and consider $p = 131$, since we are allowing all ASCII characters (128 characters) in string `s`.

*Time-complexity should be $O(1)$.
**Order of elements should be FIFO, i.e. the element inserted first in the dictionary should be the first element of the array and so on.

NOTE:- Hash-table uses **Chain-Hashing** as collision resolution technique.

NOTE:- Proceed to the next question only if you have completed this part. We will be using your implementation of the `COL106Dictionary` for the next question.

3 Frequency Analysis

Given a text containing alphabets and special characters, your task is to perform a frequency analysis of the words by ignoring the special characters in the given string and considering `[SPACE]` as the only separator of the words.
To perform the frequency analysis of the given string use the `COL106Dictionary` implemented in the previous part of the question, to store the word and frequency as key-value pairs.

Your task is to define the following methods in `FrequencyAnalysis.java`:

1. `void fillDictionaries(String inputString) throws KeyNotFoundException, KeyAlreadyExistException, NullKeyException`: Fill the three dictionaries with hash-table of sizes 101, 503 and 1009, where the key is the unique words in the given input string and value is the frequency of occurrence of those words in the given input text.

   Input Format :
   A string consisting of alphabets and special characters only. Please consider `[SPACE]` as the only separator for words and ignore the special characters and consider string to be case-insensitive while performing frequency analysis.

2. `long[] profile()`: Returns an array of long of size 4 that represents the average time taken to lookup the hash-table for every word in the given string if a hash-table of size 101, 503, 1009 is computed. Compare the time with Built-in Java Dictionary (`java.util.Dictionary`) respectively are used.
   NOTE: No marks will be allotted for the definition of this method, it is **Just for Practice** and understanding the benchmarks analysis for different Dictionaries by the student.

3. `int[] noOfCollisions()`: Returns an array of int of size 3 that represents the number of collisions in the hash-table if a hash-table of size 101, 503 and 1009 respectively are used for frequency analysis.

   Input Format :
   This method doesn't take any inputs, and uses the hash-table of the dictionaries used for frequency analysis.

   Return Format :
   An array of int of size 3 is to be returned.

   Example:
   Consider the hash-table to be of size 11, (This is just for the example, your dictionaries will be having hash-tables of sizes 101, 503 and 1009.)

Let the hash-table be:

```
0 -> 0x0090cc -> 0x198233
1 -> null
2 -> null
3 -> 0x11bd34
4 -> 0x78ee12 -> 0x943ff1
5 -> null
6 -> 0xdf123e
7 -> 0x9087cc
8 -> null
9 -> 0x08fed1 -> 0x43e441 -> 0x1902ac
10 -> null
```

Now, since at index `0`, hash-table is not null and points to a linked-list of size `2`, hence, the number of collisions here is `1`. And at index `2`, hash-table is null, so number of collisions here is `0` and so on.
So, the number of collisions will be `1+0+0+0+1+0+0+0+0+2+0 = 4`. Hence, the number of collisions to be returned for this hash-table is `4`.

4. `String[] hashTableHexaDecimalSignature()`: Returns an array of size 3 representing the hexa-decimal signature of the hash-tables of the dictionary after the frequency analysis when hash-table of size 101, 503, 1009 respectively are used. The hexa-decimal signature of a hash-table can be computed as follows: For each index of the hash-table, if there exist any element at that index, then its value will be considered as 1 else 0. Now, considering all the indices of hash-table form a Binary number of 0's and 1's such that index 0 of hash-table represents the MSB of the Binary number. Now, convert this Binary number to a Hexa-decimal and store it in the string (Only use upper-case English alphabets while computing the Hexa-decimal string).

Input Format :
This method doesn't take any inputs, and uses the hash-table of the dictionaries used for frequency analysis.

Return Format:
An array of strings of size 3 is to be returned.

Example:
Consider the hash-table to be of size 11, (This is just for the example, your dictionaries will be having hash-tables of sizes 101, 503 and 1009.)

Let the hash-table be:

```
0 -> 0x0090cc -> 0x198233
```

```
1 -> null
2 -> null
3 -> 0x11bd34
4 -> 0x78ee12 -> 0x943ff1
5 -> null
6 -> 0xdf123e
7 -> 0x9087cc
8 -> null
9 -> 0x08fed1 -> 0x43e441 -> 0x1902ac
10 -> null
```

Now, since at index `0`, hash-table is not null, so, it will correspond to `1`. And at index `2`, hash-table is null, so it will correspond to `0` and so on.
So, the binary signature hence formed will be `10011011010`. And corresponding hexa-decimal number will be `0x4da`. Hence, the hexa-decimal signature of hash-table to be returned will be `4DA`.

5. `String[] distinctWords()`: Returns an array of string containing distinct words in the same order as they appear in the input text.

Return Format:
An array of strings.

Example 1 :
```
The Indian Institute of Technology Delhi (IIT Delhi) is a public
institute of technology located in New Delhi, India.  It is one
of the twenty-three Indian Institutes of Technology created
to be Centres of Excellence for India's training, research and
development in science, engineering and technology.
```

Return 1 :
```
["the", "indian", "institute", "of", "technology", "delhi", "iit",
"is", "a", "public", "located", "in", "new", "india", "it", "one",
"twentythree", "institutes", "created", "to", "be", "centres",
"excellence", "for", "indias", "training", "research", "and",
"development", "science", "engineering"]
```

6. `Integer[] distinctWordsFrequencies()`: Returns an array of integers containing frequencies of distinct words in the same order as they appear in the input text.

Return Format:
An array of integers.

Example 1 :

```
The Indian Institute of Technology Delhi (IIT Delhi) is a public
institute of technology located in New Delhi, India.  It is one
of the twenty-three Indian Institutes of Technology created
to be Centres of Excellence for India's training, research and
development in science, engineering and technology.
```

Return 1 :

```
[2 ,2 ,2 ,5 ,4 ,3 ,1 ,2 ,1 ,1 ,1 ,2 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1
,1 ,1 ,1 ,1 ,1 ,2 ,1 ,1 ,1]
```