package shake\_n\_bacon;

import providedCode.\*;

/\*\*

\* @author Daniel Nakamura

\* @UWNetID nakamd

\* @studentID 0962452

\* @email nakamd@uw.edu

\* @author Ryan Rinaldi

\* @UWNetID ryanr12

\* @studentID 1125570

\* @email ryanr12@uw.edu

\*

\* Implements a hash table that extends the DataCounter class. Is implemented with open addressing,

\* resolving collisions by using quadratic probing to find open indices.

\*/

public class HashTable\_OA extends DataCounter {

DataCount[] table;

private int size;

Comparator<String> c;

Hasher h;

public static final int DEFAULT\_SIZE = 137;

public static final int[] PRIMES = {

281, 563, 1103, 2273, 4561, 10007, 20021, 40111, 80387, 200003, 400009

};

/\*\*

\* Takes in a Comparator and a Hasher and creates the initial table with default size

\*/

public HashTable\_OA(Comparator<String> c, Hasher h) {

table = new DataCount[DEFAULT\_SIZE];

this.c = c;

this.h = h;

size = 0;

}

/\*\*

\* Increment the count for a particular data element.

\*

\* @param data

\* data element whose count to increment.

\*/

@Override

public void incCount(String data) {

if(size + 1 > table.length \* 2 / 4) {

resize();

}

insertUpdate(new DataCount(data, 1), this.table);

}

/\*\*

\* Searches for the given word, if found increments the count of that word

\* by one. If not found, adds the new word to the table.

\*/

private void insertUpdate(DataCount data, DataCount[] tableUsed) {

int index = findWord(data.data, tableUsed);

DataCount word = tableUsed[index];

if(word != null) {

word.count++;

} else {

tableUsed[index] = data;

size++;

}

}

/\*\*

\* resizes the hash table to the next size up (using a list of prime numbers)

\* so there will be less collisions. Takes all the elements in the hash table

\* and rehashes them to a new hash table.

\*/

private void resize() {

int newSize = PRIMES[PRIMES.length - 1];

for(int i = 0; i < PRIMES.length; i++) {

if(table.length < PRIMES[i]) {

newSize = PRIMES[i];

break;

}

}

DataCount[] newTable = new DataCount[newSize];

SimpleIterator it = getIterator();

size = 0;

while(it.hasNext()) {

DataCount cur = it.next();

insertUpdate(cur, newTable);

}

this.table = newTable;

}

/\*\*

\* The number of unique data elements in the structure.

\*

\* @return the number of unique data elements in the structure.

\*/

@Override

public int getSize() {

return size;

}

/\*\*

\* Finds the given word in the hash table and returns its location.

\*/

private int findWord(String data, DataCount[] tableUsed) {

int original = h.hash(data) % tableUsed.length;

int hashBucket = original;

DataCount cur = tableUsed[hashBucket % tableUsed.length];

int constant = 1;

while(cur != null) {

if(c.compare(cur.data, data) == 0) {

return hashBucket % tableUsed.length;

}

hashBucket = original + (int) Math.pow(constant, 2);

cur = tableUsed[hashBucket % tableUsed.length];

constant++;

}

return hashBucket % tableUsed.length;

}

/\*\*

\* The current count for the data, 0 if it is not in the counter.

\*/

@Override

public int getCount(String data) {

int index = findWord(data, table);

DataCount word = table[index];

if(word != null) {

return word.count;

}

return 0;

}

/\*\*

\* Clients must not increment counts between an iterator's creation and its

\* final use. Data structures need not check this.

\*

\* @return an iterator for the elements.

\*/

@Override

public SimpleIterator getIterator() {

return new HashIterator(table, size);

}

/\*\*

\* Creates an iterator class that implements the SimpleIterator interface

\* to iterate through the hash table.

\*/

public class HashIterator implements SimpleIterator {

private int currentBucket;

private int size;

private int curSize;

private DataCount[] list;

/\*\*

\* Takes the list of HashNodes and the number of elements in the list

\*/

public HashIterator(DataCount[] list, int size) {

this.list = list;

this.size = size;

currentBucket = 0;

}

/\*\*

\* @Returns whether or not there are more elements to iterate through

\*/

public boolean hasNext() {

return size > curSize;

}

/\*\*

\* @Returns the next element in the collection. @Returns null if corrupted.

\* @Throws java.Util.NoSuchElementException if no next element.

\*/

public DataCount next() {

if (!hasNext())

throw new NullPointerException("Nothing left to iterate over");

while (currentBucket < list.length) {

if (list[currentBucket] != null) {

DataCount result = list[currentBucket];

curSize++;

currentBucket++;

return result;

}

currentBucket++;

}

return null;

}

}

}