Homework 6: External Chaining HashMaps

Graded

Student

Vidit Dharmendra Pokharna

Total Points

95 / 100 pts

Autograder Score

100.0 / 100.0

Question 2

Feedback & Manual Grading

-5 / 0 pts

✔ **– 5 pts** Efficiency 1

[-5] efficiency: In resizeBackingTable methods, should keep a size counter and stop after size elements have been seen
Great work :) -Isabelle □‿□

Autograder Results

Autograder Output

If you're seeing this message, everything compiled and ran properly!
-CS1332 TAs

Submitted Files

ExternalChainingHashMap.java Download

```java
import java.util.HashSet;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.Set;
import java.util.ArrayList;

/**
 * Your implementation of a ExternalChainingHashMap.
 *
 * @author Vidit Pokharna
 * @version 1.0
 * @userid vpokharna3
 * @GTID 903772087
 *
 * Collaborators: LIST ALL COLLABORATORS YOU WORKED WITH HERE
 *
 * Resources: LIST ALL NON-COURSE RESOURCES YOU CONSULTED HERE
 */
public class ExternalChainingHashMap<K, V> {

    /*
     * The initial capacity of the ExternalChainingHashMap when created with the
     * default constructor.
     *
     * DO NOT MODIFY THIS VARIABLE!
     */
    public static final int INITIAL_CAPACITY = 13;

    /*
     * The max load factor of the ExternalChainingHashMap.
     *
     * DO NOT MODIFY THIS VARIABLE!
     */
    public static final double MAX_LOAD_FACTOR = 0.67;

    /*
     * Do not add new instance variables or modify existing ones.
     */
    private ExternalChainingMapEntry<K, V>[] table;
    private int size;

    /**
     * Constructs a new ExternalChainingHashMap.
     *
     * The backing array should have an initial capacity of INITIAL_CAPACITY.
     *
```

```
47      * Use constructor chaining.
48      */
49     public ExternalChainingHashMap() {
50         this(INITIAL_CAPACITY);
51     }
52 
53     /**
54      * Constructs a new ExternalChainingHashMap.
55      *
56      * The backing array should have an initial capacity of capacity.
57      *
58      * You may assume capacity will always be positive.
59      *
60      * @param capacity the initial capacity of the backing array
61      */
62     public ExternalChainingHashMap(int capacity) {
63         table = new ExternalChainingMapEntry[capacity];
64     }
65 
66     /**
67      * Adds the given key-value pair to the map. If an entry in the map
68      * already has this key, replace the entry's value with the new one
69      * passed in.
70      *
71      * In the case of a collision, use external chaining as your resolution
72      * strategy. Add new entries to the front of an existing chain, but don't
73      * forget to check the entire chain for duplicate keys first.
74      *
75      * If you find a duplicate key, then replace the entry's value with the new
76      * one passed in. When replacing the old value, replace it at that position
77      * in the chain, not by creating a new entry and adding it to the front.
78      *
79      * Before actually adding any data to the HashMap, you should check to
80      * see if the array would violate the max load factor if the data was
81      * added. Resize if the load factor is greater than max LF (it is okay
82      * if the load factor is equal to max LF). For example, let's say the
83      * array is of length 5 and the current size is 3 (LF = 0.6). For this
84      * example, assume that no elements are removed in between steps. If
85      * another entry is attempted to be added, before doing anything else,
86      * you should check whether (3 + 1) / 5 = 0.8 is larger than the max LF.
87      * It is, so you would trigger a resize before you even attempt to add
88      * the data or figure out if it's a duplicate. Be careful to consider the
89      * differences between integer and double division when calculating load
90      * factor.
91      *
92      * When regrowing, resize the length of the backing table to
93      * 2 * old length + 1. You must use the resizeBackingTable method to do so.
94      *
95      * Return null if the key was not already in the map. If it was in the map,
```

```
96      * return the old value associated with it.
97      *
98      * @param key   the key to add
99      * @param value the value to add
100     * @return null if the key was not already in the map. If it was in the
101     * map, return the old value associated with it
102     * @throws IllegalArgumentException if key or value is null
103     */
104    public V put(K key, V value) {
105        if (key == null || value == null) {
106            throw new IllegalArgumentException("Either the key or value is null and cannot be added to the
    map");
107        }
108        if (((size + 1.0) / table.length) > MAX_LOAD_FACTOR) {
109            resizeBackingTable(table.length * 2 + 1);
110        }
111        int index = Math.abs(key.hashCode() % table.length);
112        ExternalChainingMapEntry<K, V> list = table[index];
113        if (list != null) {
114            while (list != null) {
115                if (list.getKey().equals(key)) {
116                    V remove = list.getValue();
117                    list.setValue(value);
118                    return remove;
119                }
120                list = list.getNext();
121            }
122            list = new ExternalChainingMapEntry<>(key, value);
123            list.setNext(table[index]);
124            table[index] = list;
125            size++;
126            return null;
127        }
128        table[index] = new ExternalChainingMapEntry<>(key, value, table[index]);
129        size++;
130        return null;
131    }
132
133    /**
134     * Removes the entry with a matching key from the map.
135     *
136     * @param key the key to remove
137     * @return the value previously associated with the key
138     * @throws java.lang.IllegalArgumentException if key is null
139     * @throws java.util.NoSuchElementException   if the key is not in the map
140     */
141    public V remove(K key) {
142        if (key == null) {
143            throw new IllegalArgumentException("Key is null and therefore cannot be removed");
```

```
144         } else {
145             int index = Math.abs(key.hashCode() % table.length);
146             ExternalChainingMapEntry<K, V> prev = null;
147             ExternalChainingMapEntry<K, V> list = table[index];
148             if (list != null) {
149                 while (list != null) {
150                     if (list.getKey().equals(key)) {
151                         V remove = list.getValue();
152                         if (prev != null) {
153                             prev.setNext(list.getNext());
154                         } else {
155                             table[index] = list.getNext();
156                         }
157                         size--;
158                         return remove;
159                     }
160                     prev = list;
161                     list = list.getNext();
162                 }
163             }
164         }
165         throw new NoSuchElementException("Key cannot be found and therefore cannot be removed");
166     }
167 
168     /**
169      * Gets the value associated with the given key.
170      *
171      * @param key the key to search for in the map
172      * @return the value associated with the given key
173      * @throws java.lang.IllegalArgumentException if key is null
174      * @throws java.util.NoSuchElementException   if the key is not in the map
175      */
176     public V get(K key) {
177         if (key == null) {
178             throw new IllegalArgumentException("Key is null and therefore cannot be found");
179         }
180         int index = Math.abs(key.hashCode() % table.length);
181         ExternalChainingMapEntry<K, V> list = table[index];
182         while (list != null) {
183             if (list.getKey().equals(key)) {
184                 return list.getValue();
185             }
186             list = list.getNext();
187         }
188         throw new NoSuchElementException("The key was not found in the map");
189     }
190 
191     /**
192      * Returns whether or not the key is in the map.
```

```
193      *
194      * @param key the key to search for in the map
195      * @return true if the key is contained within the map, false
196      * otherwise
197      * @throws java.lang.IllegalArgumentException if key is null
198      */
199     public boolean containsKey(K key) {
200         if (key == null) {
201             throw new IllegalArgumentException("Key is null and therefore cannot be found");
202         }
203         int index = Math.abs(key.hashCode() % table.length);
204         ExternalChainingMapEntry<K, V> list = table[index];
205         while (list != null) {
206             if (list.getKey().equals(key)) {
207                 return true;
208             }
209             list = list.getNext();
210         }
211         return false;
212     }
213 
214     /**
215      * Returns a Set view of the keys contained in this map.
216      *
217      * Use java.util.HashSet.
218      *
219      * @return the set of keys in this map
220      */
221     public Set<K> keySet() {
222         Set<K> set = new HashSet<K>();
223         int index = 0;
224         while (set.size() < size) {
225             ExternalChainingMapEntry<K, V> list = table[index];
226             if (list == null) {
227                 index++;
228             } else {
229                 while (list != null) {
230                     set.add(list.getKey());
231                     list = list.getNext();
232                 }
233                 index++;
234             }
235         }
236         return set;
237     }
238 
239     /**
240      * Returns a List view of the values contained in this map.
241      *
```

```
242     * Use java.util.ArrayList or java.util.LinkedList.
243     *
244     * You should iterate over the table in order of increasing index and add
245     * entries to the List in the order in which they are traversed.
246     *
247     * @return list of values in this map
248     */
249    public List<V> values() {
250        List<V> list1 = new ArrayList<V>();
251        int index = 0;
252        while (list1.size() < size) {
253            ExternalChainingMapEntry<K, V> list = table[index];
254            if (list == null) {
255                index++;
256            } else {
257                while (list != null) {
258                    list1.add(list.getValue());
259                    list = list.getNext();
260                }
261                index++;
262            }
263        }
264        return list1;
265    }
266
267    /**
268     * Resize the backing table to length.
269     *
270     * Disregard the load factor for this method. So, if the passed in length is
271     * smaller than the current capacity, and this new length causes the table's
272     * load factor to exceed MAX_LOAD_FACTOR, you should still resize the table
273     * to the specified length and leave it at that capacity.
274     *
275     * You should iterate over the old table in order of increasing index and
276     * add entries to the new table in the order in which they are traversed.
277     *
278     * Since resizing the backing table is working with the non-duplicate
279     * data already in the table, you shouldn't explicitly check for
280     * duplicates.
281     *
282     * Hint: You cannot just simply copy the entries over to the new array.
283     *
284     * @param length new length of the backing table
285     * @throws java.lang.IllegalArgumentException if length is less than the
286     *                                  number of items in the hash
287     *                                  map
288     */
289    public void resizeBackingTable(int length) {
290        if (length < size) {
```

```
            throw new IllegalArgumentException("Cannot resize as length is less than original size");
        }
        ExternalChainingMapEntry<K, V>[] newTable = new ExternalChainingMapEntry[length];
        int index = 0;
        while (index < table.length) {
            ExternalChainingMapEntry<K, V> list = table[index];
            if (list == null) {
                index++;
            } else {
                while (list != null) {
                    int index1 = Math.abs(list.getKey().hashCode() % length);
                    if (newTable[index1] == null) {
                        newTable[index1] = new ExternalChainingMapEntry<>(list.getKey(), list.getValue());
                    } else {
                        ExternalChainingMapEntry<K, V> temp = newTable[index1];
                        newTable[index1] = new ExternalChainingMapEntry<>(list.getKey(), list.getValue(), temp);
                    }
                    list = list.getNext();
                }
                index++;
            }
        }
        table = newTable;
    }


    /**
     * Clears the map.
     *
     * Resets the table to a new array of the initial capacity and resets the
     * size.
     */
    public void clear() {
        table = new ExternalChainingMapEntry[INITIAL_CAPACITY];
        size = 0;
    }

    /**
     * Returns the table of the map.
     *
     * For grading purposes only. You shouldn't need to use this method since
     * you have direct access to the variable.
     *
     * @return the table of the map
     */
    public ExternalChainingMapEntry<K, V>[] getTable() {
        // DO NOT MODIFY THIS METHOD!
        return table;
    }
```

```
340
341     /**
342      * Returns the size of the map.
343      *
344      * For grading purposes only. You shouldn't need to use this method since
345      * you have direct access to the variable.
346      *
347      * @return the size of the map
348      */
349     public int size() {
350         // DO NOT MODIFY THIS METHOD!
351         return size;
352     }
353 }
354
```