Algorithms and Data Structures in an Object Oriented Framework - Coursework

Ryan Welch

December 16, 2016

Contents

Introduction

The task was to create an abstract dictionary data structure defined by an interface and implemented by an efficient data structure. The requirements are to implement a dictionary that can insert, remove and count the occurrences or words. I started with a reference 'array and count' implementation where the words are stored in a virtual array which doubles in size every time it runs out of space in the real array. This is a simple way to check more complicated implementations for correctness as there is very little complexity in this reference implementation.

```
public interface WordStore {
    void add(String word);

    int count(String word);

    void remove(String word);
}
```

Figure 1: WordStore interface

I chose to use a hash table as it provides in the average case $O(1)$ lookup, insertion and removal. Whereas the reference implementation is an abysmal $O(n)$ for insertion, removal and lookup. While in theory a hash table is actually $O(n)$, in the rare case where all elements have the same hash, in practice it performs closer to $O(1)$. Another data structure I considered was a binary tree, in particular a self-balancing binary tree such as a red-black tree which has a guaranteed $O(\log n)$ worst case however the hash table appears to be a better, more performant choice [1]. One disadvantage with a hash table is it wastes some memory as the array will never be full, however we are not very concerned with it's memory use as long as it is not excessive.

Structure

A hash table is simply an array, however instead of placing items linearly we insert them at a position determined by a hash function. The hash function aims for a uniform distribution of the items across all buckets. In the case where the hash function chooses the same bucket for two different elements we must store both elements in the same bucket. There are two design choices for dealing with collisions: open addressing and separate chaining. Separate chaining is where each bucket contains a list structure to store all elements at the same position, however requiring the additional step of iterating the list to find the element, usually $O(n)$. Open addressing is where if the bucket is already occupied you traverse the array until you find the next unoccupied slot, this is also $O(n)$ however incurs a hidden penalty in which you are occupying another position in the table and increasing the chances of another collision more rapidly.

I chose to use separate chaining with a linked list data structure for storing collisions, this keeps the complexity low. It would be more efficient in most cases to use a better performing list structure such as self-balancing binary trees to store the collisions however it increases a lot of additional code and complexity for a small increase in performance, it also may reduce performance on smaller hash table due to the increased insertion cost of a balancing binary tree. It would reduce the worst case in theory from $O(n)$ to $O(\log n)$ however I found there were not many collisions and hence only a small cost in traversing the list.

Insertion

Insertion is a worst-case $O(n)$ operation however it is in most cases $O(1)$, the code works by generating the hash for the word and mapping it to a position in the array, and then inserting the word into that position. Each position contains a linked list to deal with possible collisions of a different word having the same hash, if the list already contains the same word the count of the node which is our linked list cell is increased to account for the same word being added again. If the word is not found it is added to the end of the linked list.

```
int position = getIndex(word, array.length);

if(array[position] == null) {
    // Create start of linked list as this node
    array[position] = new WordNode(word);
    items++;
} else {
    // Insert node at end of linked list, or increment count
    WordNode node = array[position];

    while(node != null) {
        if(word.equals(node.getWord())) {
            node.incrementCount();
            break;
        } else if(!node.hasNextNode()) {
            node.setNextNode(new WordNode(word));
            items++;
            break;
        }

        node = node.getNextNode();
    }
}
```

Figure 2: Code fragment from the add method

In the implementation, the WordNode class is a the cell that represents the linked lists in the hash table, each WordNode can hold a word, the count of how many times it has been added and an optional pointer to another WordNode. The bucket array is simply an array of pointers to the root WordNodes which form the linked lists for each bucket.

Deletion

Removing a word works very similar to insertion but in reverse. First we find the bucket for the word by calculating the index of the bucket it is stored at using its hash. If the bucket is not empty then the linked list at the bucket is traversed until it finds the node for the word. If the node has a count greater than one we remove the word by simply decreasing the count which represents the number of times it has been added to the word store. If however the count is 1 we remove the node completely by altering the pointers in the linked list, the parent's next node is set to the next node of the one we are removing (it may be null). The deletion method has the same time complexity as the addition, with $O(1)$ on average and $O(n)$ at worst.

```
int position = getIndex(word, array.length);

if(array[position] != null) {
    WordNode parent = null;
    WordNode node = array[position];

    while(node != null) {
        if(word.equals(node.getWord())) {
            if(node.getCount() > 1) {
                node.decrementCount();
                break;
            } else {
                if(parent != null) {
                    parent.setNextNode(node.getNextNode());
                } else {
                    array[position] = node.getNextNode();
                }
                items--;
                break;
            }
        }

        parent = node;
        node = node.getNextNode();
    }
}
```

Figure 3: Code fragment from the remove method

Retrieval

Retrieval from the hash table again has the same big-O categories with $O(1)$ on average and $O(n)$ at worst case. It works again very similarly, it calculates the hash to access the correct bucket. If the bucket contains the word in the linked list, it returns the value of the count stored in that node as it may represent more than one copy of the same word. If it is not found in the list or the bucket is empty the count is 0.

```
int position = getIndex(word, array.length);

if(array[position] != null) {
    WordNode node = array[position];

    while(node != null) {
        if(word.equals(node.getWord())) return node.getCount();

        node = node.getNextNode();
    }
}

return 0;
```

Figure 4: Code fragment from the count method

Collisions

Resizing

An additional step is performed before addition or removal, a call is made to the resize method. A resize is needed to prevent the number of collisions from becoming too high. For example, if the array of linked lists is created initially with a size of 100 and 1 million items are added there will be almost 1 million collisions as there is not enough space to store the 1 million items.

In order to prevent this we resize the array when the array is almost full, we do this by calculating a load factor which is the number of items (WordNode's) in the hash table over the number of buckets (our array size). Then if the load factor is over an arbitrary value, such as 0.7 we double the array size and move all the items to their new hash positions. The arbitrary value of 0.7 worked well for me, it equates to when the array is 70%, the closer the load factor gets to 1 the more likely you are to collide as in a perfect distribution it is when there are no more spaces in the array, 0.7 provides a nice buffer as the hash function can not be perfect.

```
float currentLoadFactor = items / array.length;
float minLoadFactor = items / (array.length / 2);

boolean shouldShrink = array.length > minSize && minLoadFactor <
  ↪  MAX_LOAD_FACTOR;
boolean shouldGrow = currentLoadFactor > MAX_LOAD_FACTOR;

if(!shouldGrow && !shouldShrink) return; // No need to resize
```

Figure 5: Code fragment calculating load factor and deciding whether to resize

Whenever we resize the bucket array it becomes necessary to recompute the position for all the items as the position depends on the size of the array as we map the hash (which is the always the same) to the buckets. This means we take a large performance hit on insertion or removal whenever we need to resize, however despite this it provides better performance for future insertion and removal due to the reduced number of collisions. This means that when we resize, our big-O is $O(n)$ which is the same as the formal big-O of the hash table.

In order to avoid resizing if you know the size of the data in advance, the WordStore provides a constructor which accepts the number of items that will be added. This can be used as the initial value for the bucket array which means we are guaranteed to not need to resize when adding the given number of elements. Additionally, we round the array size up to the next power of two and double it on resize to ensure it is still a power of two, the array size is required to be a power of two due to our hash function which is explained later.

The Hash Function

The hash function is the most important part of the hash table, it should distribute the items uniformly across all the buckets in order to avoid collisions, our items in this case are strings so we must find a way to represent a string as a 32bit integer. It is not possible to have a unique integer for every string as the number of possible strings is infinitely larger than a 32bit integer. However

```java
private int hashDijb2(byte[] data) {
    int hash = 5381;

    for(byte b : data) {
        hash = ((hash << 5) + hash) + b; /* hash * 33 + b */
    }

    return hash;
}

private int getIndex(String word, int size) {
    // Dijb2
    int hash = hashDijb2(word.getBytes());

    // Requires table to be a PoT
    return hash & (size-1);
}
```

Figure 6: The hash function and mapping to bucket array

we can try to randomize the distribution of 32bit integers so that the probability of another string having the same hash is $\frac{1}{2^{32}}$.

The first attempt at a hash function was simply adding the ascii value of each character multiplied by a prime to ensure that the same letters but in different orders resulted in different hashes. This worked okay however later tried more algorithms for generating a hash, two of which where djb2 [2] and FNV [3], both offer an improvement however I chose djb2 as it resulted in slightly less collisions on my test data. It operates on bytes of data rather than characters and follows a similar pattern by multiplying by a number before adding the next byte. However it uses bit shifts to achieve the multiplication as it is much faster. Computing the hash is often the slowest part of a hash table and hence we aim to make it as fast as possible but still providing a good enough result to avoid collisions.

Once we have a 32bit integer as a hash for our word, we need to map the integer to a bucket in our hash table. Taking the modulus of the integer would work however it is relatively slow so instead we use a bitwise AND operation on the lower bits of the integer, this can mean we do not always use the full information available in the hash but it is faster than using modulus and in this case we care about performance over a perfect hash. A requirement of doing a bitwise AND instead of modulus is making sure that table size is a power of two. This works as to map the hash to the bucket we take the n rightmost number of bits, each additional bit increases our output size by double, so if the table size is 2^n in size, we use the n most significant bits.

Comparison

In order to show the hash table's performance and verify its big-O category I compared it against a simple Array and count data structure and a Binary Tree data structure. The Array implementation was also used as a comparison whilst developing the hash table version. In the graphs below I tested each data structure with a fixed number of insertions, deletions and retrievals whilst changing the size of previously stored data. This should show the big-O category as its category defines how long an individual operation will take depending on the how much data is already in the structure. I measured the time taken with as much precision as possible using javas `System.nanoTime()`.

The first test is shown in Figure 7, it shows the trend of all three data structures roughly constant, however noticibly the Binary Tree has a larger gradient than the other two. This is because the binary tree has a big-O of $O(\log(n))$ whilst the insertion for the Array and insertion for the Hash Table are $O(1)$. The Hash Table takes a constant amount of time longer than the Array to insert which is most likely due to the higher costs in insertion from calculating the hash of each word.

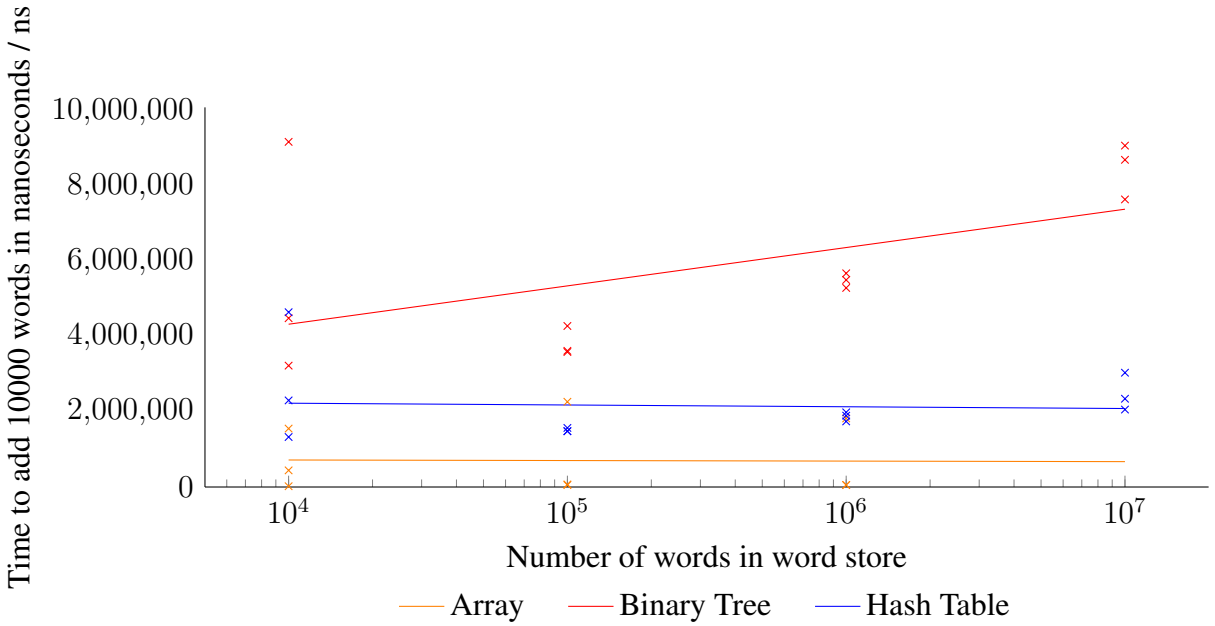

Figure 7: Graph of adding n words to word store[1]

The count test data is shown in Figure 8, the Array data is not included in the graph as it is much slower however it is included in Table 2 in the appendix for completeness. The data shows a similar trend as in the addition test, the Hash Table remains constant at around 2ms but the Binary Tree increases linearly in $O(\log(n))$ time. The Array performs at a very slow $O(n)$ and struggles at anything larger than an initial size of 1 million as it must check every position in the array in contrast to the Hash Table which simply looks up the position for the node.

Finally, the remove method test in Figure 9 shows again the same patterns, the Hash Table performs at $O(1)$ and the Binary Tree at $O(\log(n))$. This shows just how efficient the hash table is, and is capable to deal with vast amount of data, more than I have tested. The hash table is more likely to be bounded by memory than processing time on larger data sets. It does however also incur a a large penalty when resizing to accommodate more elements which is not represented accurately in this data, however at a large scale this would be an issue with the hash table which could be mitigated by, after a resize, maintaining more than one active hash function and on every insert or delete operation a portion of data is relocated to the new array [4]. Hash Tables are a large improvement over a simple 'array and count' structure, there are still improvements that can be made to my implementation however it would also increase the complexity and my implementation still performs as expected.

[1]Tests performed on an Intel Core i7-3770k @ 3.5Ghz

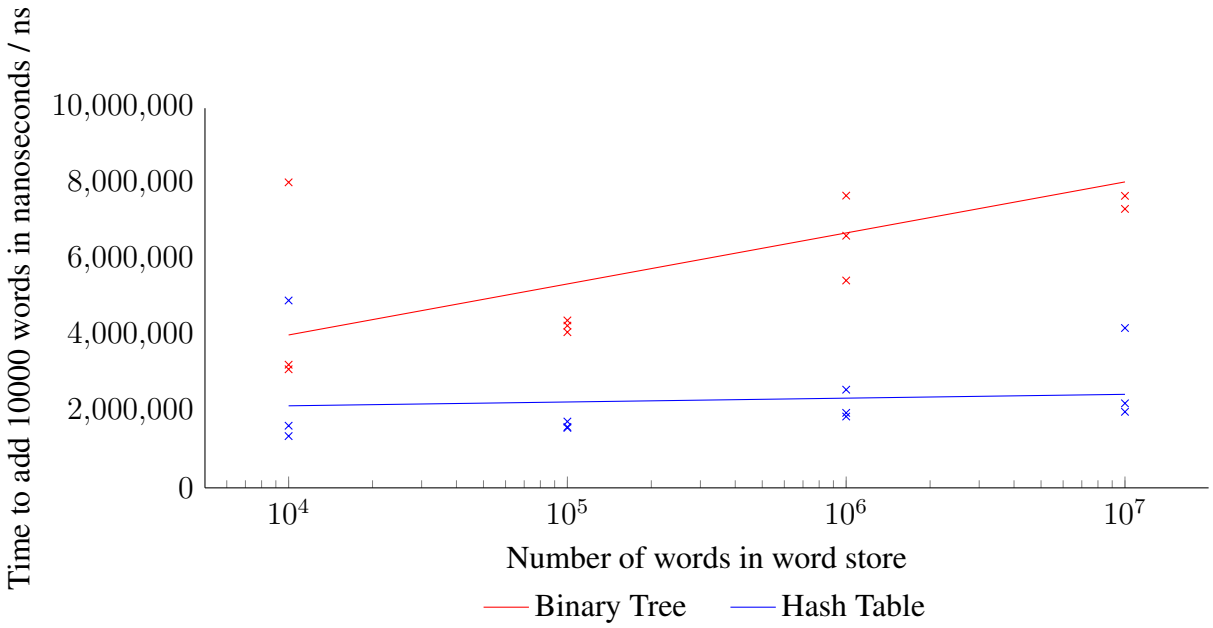

Figure 8: Graph of counting n words in word store[1]

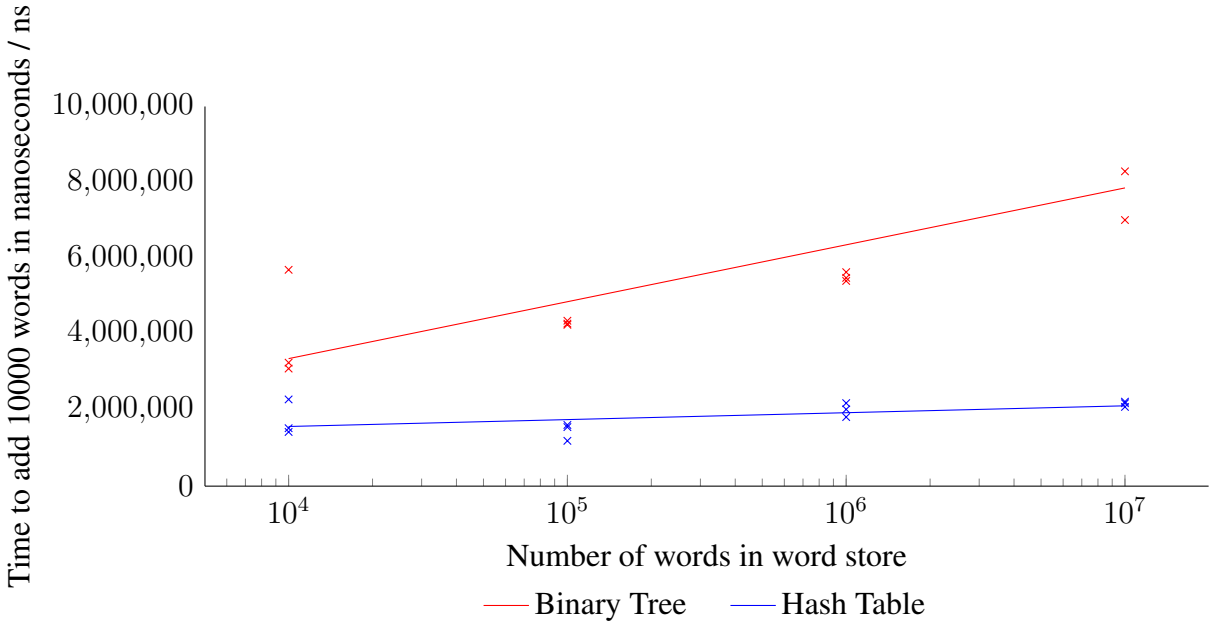

Figure 9: Graph of removing n words from word store[1]

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms: Third Edition*. MIT Press, 2009. ISBN: 9780262033848.

[2] *Hash Functions DJB2*. URL: http://www.cse.yorku.ca/~oz/hash.html.

[3] Landon Curt Noll. *FNV Hash*. URL: http://www.isthe.com/chongo/tech/comp/fnv/index.html.

[4] *Resizing Hash Tables*. 2011. URL: https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf.

Appendix

Timing data

Table 1: Add n words to word store

Number of words	Array / ns	Binary Tree / ns	Hash Table / ns
10000	443,403	9,197,108	4,658,064
10000	1,557,160	4,497,913	1,332,251
10000	21,003	3,234,507	2,307,445
100000	68,844	3,597,981	1,574,372
100000	2,271,273	3,628,611	1,487,733
100000	52,800	4,291,090	1,486,275
1000000	53,384	5,516,864	1,747,649
1000000	1,827,286	5,694,518	1,989,478
1000000	53,092	5,305,373	1,901,673
10000000		8,718,700	2,352,369
10000000		7,663,285	3,046,061
10000000		9,097,635	2,063,865

Table 2: Count n words in word store

Number of words	Array / ns	Binary Tree / ns	Hash Table / ns
10000	922,217,516	8,047,469	4,935,482
10000	893,515,350	3,244,133	1,636,507
10000	1,038,367,764	3,123,365	1,363,464
100000	4,088,574,092	4,100,893	1,604,418
100000	4,424,993,451	4,273,295	1,579,915
100000	5,296,396,127	4,410,108	1,746,774
1000000	50,014,403,299	7,697,706	2,586,322
1000000	50,179,007,559	5,461,731	1,879,211
1000000	50,763,058,943	6,641,416	1,975,184
10000000		7,347,652	2,226,641
10000000		7,691,873	4,210,285
10000000		10,476,267	2,003,189

Table 3: Remove n words from word store

Number of words	Array / ns	Binary Tree / ns	Hash Table / ns
10000	425,449,689	5,698,310	2,284,108
10000	619,336,040	3,096,527	1,526,239
10000	701,767,252	3,252,302	1,427,349
100000	2,055,966,773	4,357,017	1,193,396
100000	2,283,816,614	4,281,462	1,555,410
100000	2,337,686,850	4,250,249	1,611,419
1000000	19,547,127,303	5,486,235	2,186,677
1000000	19,851,233,681	5,405,722	1,819,993
1000000	20,392,864,249	5,637,050	2,021,859
10000000		7,009,557	2,183,759
10000000		8,291,925	2,225,766
10000000		10,520,900	2,087,203

WordStoreHashTableImp

```
1  package com.ryanwelch.wordstore;
2
3  /**
4   * Copyright 2016 (C) Ryan Welch
5   *
6   * @author Ryan Welch
7   */
8  public class WordStoreHashTableImp implements WordStore {
9
10     // Maximum load factor allowed before resize
11     private static final float MAX_LOAD_FACTOR = 0.7f;
12     // Smallest size of hash table
13     private static final int MINIMUM_TABLE_SIZE = 8;
14
15     private WordNode[] array;
16     private int items;
17     private int minSize;
18
19     public WordStoreHashTableImp(int minSize) {
20         this.minSize = getNextPowerOf2(minSize < MINIMUM_TABLE_SIZE ?
           ↪  MINIMUM_TABLE_SIZE : minSize);
21         this.array = new WordNode[this.minSize];
```

```
22          this.items = 0;
23      }
24
25      public WordStoreHashTableImp() {
26          this(MINIMUM_TABLE_SIZE);
27      }
28
29      private int getNextPowerOf2(int n) {
30          int res = 1;
31          while (res < n) res = res << 1;
32          return res;
33      }
34
35      /**
36       * Resizes the array if necessary
37       */
38      private void resize() {
39          float currentLoadFactor = items / array.length;
40          float minLoadFactor = items / (array.length / 2);
41
42          boolean shouldShrink = array.length > minSize && minLoadFactor <
              ↪  MAX_LOAD_FACTOR;
43          boolean shouldGrow = currentLoadFactor > MAX_LOAD_FACTOR;
44
45          if(!shouldGrow && !shouldShrink) return; // No need to resize
46
47          WordNode[] newArray;
48          if(shouldGrow) newArray = new WordNode[2 * array.length];
49          else newArray = new WordNode[array.length / 2];
50
51          // Move all nodes to their new positions, and relink collisions
52          for(int i = 0; i < array.length; i++) {
53              WordNode node = array[i];
54              WordNode nextNode;
55
56              while(node != null) {
57                  int newIndex = getIndex(node.getWord(), newArray.length);
58
59                  if(newArray[newIndex] == null) { // If empty place in array
                      ↪  set the node
60                      newArray[newIndex] = node;
61                  } else { // If there is a collision, add to the end of the
                      ↪  linked list
62                      WordNode listNode = newArray[newIndex];
63                      while(listNode != null) {
64                          if(!listNode.hasNextNode()) {
65                              listNode.setNextNode(node);
66                              break;
67                          }
68                          listNode = listNode.getNextNode();
69                      }
70                  }
71
72                  nextNode = node.getNextNode();
73                  node.setNextNode(null);
74                  node = nextNode;
75              }
76          }
77
78          array = newArray;
79      }
80
81      /**
82       * Dijb2 hash
83       */
84      private int hashDijb2(byte[] data) {
85          int hash = 5381;
86
87          for(byte b : data) {
88              hash = ((hash << 5) + hash) + b; /* hash * 33 + b */
89          }
90
91          return hash;
92      }
```

```
93
94      private int getIndex(String word, int size) {
95          // Dijb2
96          int hash = hashDijb2(word.getBytes());
97
98          // Requires table to be a PoT
99          return hash & (size-1);
100     }
101
102     @Override
103     public void add(String word) {
104         resize(); // Resize if needed
105
106         int position = getIndex(word, array.length);
107
108         if(array[position] == null) {
109             // Create start of linked list as this node
110             array[position] = new WordNode(word);
111             items++;
112         } else {
113             // Insert node at end of linked list, or increment count
114             WordNode node = array[position];
115
116             while(node != null) {
117                 if(word.equals(node.getWord())) {
118                     node.incrementCount();
119                     break;
120                 } else if(!node.hasNextNode()) {
121                     node.setNextNode(new WordNode(word));
122                     items++;
123                     break;
124                 }
125
126                 node = node.getNextNode();
127             }
128         }
129     }
130
131     @Override
132     public int count(String word) {
133         int position = getIndex(word, array.length);
134
135         if(array[position] != null) {
136             WordNode node = array[position];
137
138             while(node != null) {
139                 if(word.equals(node.getWord())) return node.getCount();
140
141                 node = node.getNextNode();
142             }
143         }
144
145         return 0;
146     }
147
148     @Override
149     public void remove(String word) {
150         resize(); // Resize if needed
151
152         int position = getIndex(word, array.length);
153
154         if(array[position] != null) {
155             WordNode parent = null;
156             WordNode node = array[position];
157
158             while(node != null) {
159                 if(word.equals(node.getWord())) {
160                     if(node.getCount() > 1) {
161                         node.decrementCount();
162                         break;
163                     } else {
164                         if(parent != null) {
165                             parent.setNextNode(node.getNextNode());
166                         } else {
```

```
167                             array[position] = node.getNextNode();
168                         }
169                         items--;
170                         break;
171                     }
172                 }
173 
174                 parent = node;
175                 node = node.getNextNode();
176             }
177         }
178     }
179 
180     private class WordNode {
181         private String word;
182         private int count;
183         private WordNode nextNode;
184 
185         WordNode(String word, int count) {
186             this.word = word;
187             this.count = count;
188         }
189 
190         WordNode(String word) {
191             this(word, 1);
192         }
193 
194         String getWord() {
195             return word;
196         }
197 
198         boolean hasNextNode() {
199             return nextNode != null;
200         }
201 
202         WordNode getNextNode() {
203             return nextNode;
204         }
205 
206         void setNextNode(WordNode node) {
207             nextNode = node;
208         }
209 
210         int getCount() {
211             return count;
212         }
213 
214         void incrementCount() {
215             this.count++;
216         }
217 
218         void decrementCount() {
219             this.count--;
220         }
221     }
222 
223 }
```

WordStoreArrayImp

```java
package com.ryanwelch.wordstore;

/**
 * Copyright 2016 (C) Ryan Welch
 *
 * @author Ryan Welch
 */
public class WordStoreArrayImp implements WordStore {

    private String[] array;
    private int length = 0;
    private int actualLength;

    public WordStoreArrayImp(int initialArraySize) {
        this.actualLength = initialArraySize;
        this.array = new String[initialArraySize];
    }

    public WordStoreArrayImp() {
        this(20);
    }

    private void growArray() {
        if(length < actualLength) return; // No need to grow, space still left

        actualLength *= 2;
        String[] newArray = new String[actualLength];
        for(int i = 0; i < length; i++) {
            newArray[i] = array[i];
        }
        array = newArray;
    }

    @Override
    public void add(String word) {
        if(length >= actualLength) growArray();

        array[length++] = word;
    }

    @Override
    public int count(String word) {
        int count = 0;
        for(int i = 0; i < length; i++) if(array[i].equals(word)) count++;
        return count;
    }

    @Override
    public void remove(String word) {
        int index = -1;
        for(int i = 0; i < length; i++) {
            if(array[i].equals(word)) {
                index = i;
                break;
            }
        }
        if(index == -1) return;

        length--;
        for(int i = index; i < length; i++) {
            array[i] = array[i+1];
        }
    }
}
```