Programming Assignment 2

● Graded

Student

Mohammed Ali Abdul-nabi

Total Points

18.8831 / 20 pts

Autograder Score

8.8831 / 10.0

Passed Tests

Compiled Succesfully

Interface Tests Small

Interface Tests Large

Dictionary Tests

Question 2

Ordered Dictionary Implementation **4** / 4 pts

2.1 **Classes Key, Record, BSTNode properly implemented** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts some omissions in the class implementation.

– 1 pt No Classes implemented, or all classes have severe omissions.

2.2 **Classes BSTDictionary and BinarySearchTree, no use of additional data structures to find successors/predecessors** **3** / 3 pts

✔ **– 0 pts** Correct

– 1.5 pts Missing/wrong implementation for BinarySearchTree

– 1.5 pts Missing/wrong implementation for BSTDictionary

Question 3

Interface Implementation **4** / 4 pts

3.1 **Main either contains all logic or delegates to smaller task specific PRIVATE methods** **4** / 4 pts

✔ **– 0 pts** Correct

– 1 pt Methods are not private.

– 2 pts Missing some methods.

– 4 pts No implementation.

– 0.5 pts hardcoded input file

– 1 pt Command parser is not working correctly / it is always exiting the program

– 0 pts Click here to replace this description.

Question 4

Coding Style **2** / 2 pts

4.1 **Meaningful names for variables and constants, all instance variables are private, only public methods are as specified in assignment** **0.5** / 0.5 pts

✔ **– 0 pts** Correct

– 0.5 pts repetitive/meaningless names

– 0.5 pts Some helper methods and variables are public

4.2 **Readability, Good and consistent indentations** **0.5** / 0.5 pts

✔ **– 0 pts** Correct

– 0.5 pts inconsistent indentation/difficult to read

4.3 **Comments, somewhat descriptive, everywhere where necessary (accessors not needed)** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts important segments left without comment

– 1 pt very sparse comments or no comments at all

Autograder Results

Autograder Output

{"score": 2, "output": "Code compiled successfully", "output_format": "simple_format", "visibility": "visible", "st

Compiled Succesfully

Interface Tests Small

Test 4:failed, expected output = a device that makes noise. | your output was : at interface.main(interface.jav

Interface Tests Large

Test 2:failed, expected output = dummy name for an algorithm. | your output was : at interface.main(interfac

Dictionary Tests

Submitted Files

BinarySearchTree.java Download

```java
public class BinarySearchTree {

    // Root node of the binary search tree
    private BSTNode root;

    // Constructor to creat an empty tree with a null root
    public BinarySearchTree() {
        this.root = null;
    }

    // Returns the root node of this binary search tree
    public BSTNode getRoot() {
        return root;
    }

    // Adds a record to the binary search tree with root r
    // Throws DictionaryException if the key already exsists
    public void insert(BSTNode r, Record d) throws DictionaryException
    {
        if (r == null)
        {
            root = new BSTNode(d);
        }
        else
        {
            insertRecursively(r, d);
        }
    }

    // Helper method to insert a record into the tree recursivly
    private void insertRecursively(BSTNode r, Record d) throws DictionaryException
    {
        // Compare the key of the new record with the key of the current node's record
        if (d.getKey().compareTo(r.getRecord().getKey()) < 0)
        {
            // If the key of the new record is smaller, check if the left child is null
            if (r.getLeftChild() == null)
            {
                // If left child is null, insert the new record here
                r.setLeftChild(new BSTNode(d));
            }
            else
            {
                // If left child is not null, continue recursion on the left subtree
                insertRecursively(r.getLeftChild(), d);
            }
        }
        else if (d.getKey().compareTo(r.getRecord().getKey()) > 0)
        {
```

```
50          // If the key of the new record is greater, check if the right child is null
51          if (r.getRightChild() == null)
52          {
53              // If right child is null, insert the new record here
54              r.setRightChild(new BSTNode(d));
55          }
56          else
57          {
58              // If right child is not null, continue recursion on the right subtree
59              insertRecursively(r.getRightChild(), d);
60          }
61      }
62      else
63      {
64          // If the key of the new record matches the current node's key, throw an exception
65          throw new DictionaryException("Key already exsists in the tree");
66      }
67  }
68
69  // Returns the node storing the given key; returns null if the key is not found
70  public BSTNode get(BSTNode r, Key k)
71  {
72      if (r == null)
73      {
74          return null;
75      }
76      if (k.compareTo(r.getRecord().getKey()) < 0)
77      {
78          return get(r.getLeftChild(), k);
79      }
80      else if (k.compareTo(r.getRecord().getKey()) > 0)
81      {
82          return get(r.getRightChild(), k);
83      }
84      else
85      {
86          return r;
87      }
88  }
89
90  // Removes the node with the given key from the tree
91  public void remove(BSTNode r, Key k) throws DictionaryException
92  {
93      root = removeRecursively(r, k);
94  }
95
96  private BSTNode removeRecursively(BSTNode r, Key k) throws DictionaryException
97  {
98      if (r == null)
99      {
100         throw new DictionaryException("Key not found in the tree");
101     }
```

```
102         if (k.compareTo(r.getRecord().getKey()) < 0)
103         {
104            r.setLeftChild(removeRecursively(r.getLeftChild(), k));
105         }
106         else if (k.compareTo(r.getRecord().getKey()) > 0)
107         {
108            r.setRightChild(removeRecursively(r.getRightChild(), k));
109         }
110         else
111         {
112            if (r.getLeftChild() == null)
113            {
114               return r.getRightChild();
115            }
116            else if (r.getRightChild() == null)
117            {
118               return r.getLeftChild();
119            }
120            BSTNode smallestNode = smallest(r.getRightChild());
121            r.setRecord(smallestNode.getRecord());
122            r.setRightChild(removeRecursively(r.getRightChild(), smallestNode.getRecord().getKey()));
123         }
124         return r;
125      }
126
127      // Returns the node with the smallest key in tree with root r
128      public BSTNode smallest(BSTNode r)
129      {
130         if (r == null || r.getLeftChild() == null)
131         {
132            return r;
133         }
134         return smallest(r.getLeftChild());
135      }
136
137      // Returns the node with the largest key in tree with root r
138      public BSTNode largest(BSTNode r)
139      {
140         if (r == null || r.getRightChild() == null)
141         {
142            return r;
143         }
144         return largest(r.getRightChild());
145      }
146   }
147
148
```

BSTDictionary.java Download

```java
public class BSTDictionary implements BSTDictionaryADT {

    private BSTNode root;

    // Constructor to create an empty BSTDictionary with a null root
    public BSTDictionary()
    {
        this.root = null;
    }

    // Method to insert a Record into the dictionary
    public void put(Record d) throws DictionaryException
    {
        if (root == null)
        {
            root = new BSTNode(d);
        }
        else
        {
            insertRecursively(root, d);
        }
    }

    private void insertRecursively(BSTNode node, Record d) throws DictionaryException {
        if (node.getRecord() == null)
        {
            // This is a leaf node, replace with a new internal node
            node.setRecord(d);
            node.setLeftChild(new BSTNode(null)); // Create new leaf nodes
            node.setRightChild(new BSTNode(null));
        }
        else
        {
            int comparison = d.getKey().compareTo(node.getRecord().getKey());
            if (comparison < 0)
            {
                if (node.getLeftChild() == null)
                {
                    node.setLeftChild(new BSTNode(d));
                    node.getLeftChild().setLeftChild(new BSTNode(null));
                    node.getLeftChild().setRightChild(new BSTNode(null));
                }
                else
                {
                    insertRecursively(node.getLeftChild(), d);
                }
            } else if (comparison > 0) {
                if (node.getRightChild() == null) {
                    node.setRightChild(new BSTNode(d));
```

```
50              node.getRightChild().setLeftChild(new BSTNode(null));
51              node.getRightChild().setRightChild(new BSTNode(null));
52           }
53           else
54           {
55              insertRecursively(node.getRightChild(), d);
56           }
57        }
58        else
59        {
60           throw new DictionaryException("Key already exists in the dictionary");
61        }
62     }
63  }
64
65  // Method to get the Record associated with a given Key
66  public Record get(Key k)
67  {
68     BSTNode node = getRecursively(root, k);
69     if (node != null && node.getRecord() != null)
70     {
71        return node.getRecord();
72     }
73     return null; // Key not found
74  }
75
76  private BSTNode getRecursively(BSTNode node, Key k)
77  {
78     if (node == null || node.getRecord() == null)
79     {
80        return null;
81     }
82     int comparison = k.compareTo(node.getRecord().getKey());
83
84     if (comparison < 0)
85     {
86        return getRecursively(node.getLeftChild(), k);
87     }
88     else if (comparison > 0)
89     {
90        return getRecursively(node.getRightChild(), k);
91     }
92     else
93     {
94        return node; // Found the node
95     }
96  }
97
98  // Removes a node with the specified key from the tree
99  public void remove(Key k) throws DictionaryException
100 {
101 root = removeRecursively(root, k);
```

```
102     }
103
104     private BSTNode removeRecursively(BSTNode node, Key k) throws DictionaryException
105     {
106     if (node == null || node.getRecord() == null)
107     {
108        throw new DictionaryException("Key not found in the dictionary");
109     }
110
111     int comparison = k.compareTo(node.getRecord().getKey());
112     if (comparison < 0)
113     {
114        node.setLeftChild(removeRecursively(node.getLeftChild(), k));
115     }
116     else if (comparison > 0)
117     {
118        node.setRightChild(removeRecursively(node.getRightChild(), k));
119     }
120     else
121     {
122        // Node with the key found
123        if (node.getLeftChild() == null || node.getLeftChild().getRecord() == null)
124        {
125           return node.getRightChild(); // Replace with right child if left is null or a leaf
126        }
127        else if (node.getRightChild() == null || node.getRightChild().getRecord() == null)
128        {
129           return node.getLeftChild(); // Replace with left child if right is null or a leaf
130        }
131
132        // Node with two children: Find the smallest node in the right subtree
133        BSTNode current = node.getRightChild();
134        while (current.getLeftChild() != null && current.getLeftChild().getRecord() != null)
135        {
136           current = current.getLeftChild();
137        }
138
139        // Replace node's record with the smallest node's record
140        node.setRecord(current.getRecord());
141
142        // Remove the smallest node in the right subtree
143        node.setRightChild(removeRecursively(node.getRightChild(), current.getRecord().getKey()));
144     }
145
146     return node;
147
148     }
149
150
151     public Record successor(Key k)
152     {
153     BSTNode current = root;
```

```
154     BSTNode successor = null;
155
156     while (current != null && current.getRecord() != null)
157     {
158         int comparison = k.compareTo(current.getRecord().getKey());
159         if (comparison < 0)
160         {
161             // Possible successor; move to the left subtree
162             successor = current;
163             current = current.getLeftChild();
164         }
165         else
166         {
167             // Move to the right subtree
168             current = current.getRightChild();
169         }
170     }
171
172     // If we found a successor, return the associated Record
173     if (successor != null)
174     {
175         return successor.getRecord();
176     }
177
178     return null; // No successor found
179 }
180
181      // Finds and returns the predecessor Record of the given key
182     public Record predecessor(Key k)
183     {
184     BSTNode current = root;
185     BSTNode predecessor = null;
186
187     while (current != null && current.getRecord() != null)
188     {
189         int comparison = k.compareTo(current.getRecord().getKey());
190         if (comparison > 0)
191         {
192             // Possible predecessor; move to the right subtree
193             predecessor = current;
194             current = current.getRightChild();
195         }
196         else
197         {
198             // Move to the left subtree
199             current = current.getLeftChild();
200         }
201     }
202
203     // If we found a predecessor, return the associated Record
204     if (predecessor != null)
205     {
```

```
206         return predecessor.getRecord();
207     }
208 
209     return null; // No predecessor found
210     }
211 
212     // Finds and returns the node with the smallest key in the subtree with root r
213     public Record smallest()
214     {
215         if (root == null || root.getRecord() == null)
216         {
217             return null; // Handle empty tree case
218         }
219         BSTNode current = root;
220         while (current.getLeftChild() != null && current.getLeftChild().getRecord() != null)
221         {
222             current = current.getLeftChild();
223         }
224 
225         return current.getRecord(); // Return the smallest node's record
226     }
227 
228 
229     public Record largest()
230     {
231         if (root == null || root.getRecord() == null)
232         {
233             return null; // Handle empty tree case
234         }
235         BSTNode current = root;
236         while (current.getRightChild() != null && current.getRightChild().getRecord() != null)
237         {
238             current = current.getRightChild();
239         }
240         return current.getRecord(); // Return the largest node's record
241     }
242 
243 }
244 
```

BSTNode.java Download

```java
public class BSTNode {

    // Attribute Declaration
    private Record node;
    private BSTNode rightChild;
    private BSTNode leftChild;
    private BSTNode parent;

    // Constructor
    public BSTNode(Record node)
    {
        this.node = node;
        this.rightChild = null;
        this.leftChild = null;
        this.parent = null;
    }

    // Getters
    public Record getRecord()
    {
        return node;
    }

    public BSTNode getRightChild()
    {
        return rightChild;
    }

    public BSTNode getLeftChild()
    {
        return leftChild;
    }

    public BSTNode getParent()
    {
        return parent;
    }

    // Setters
    public void setRecord(Record node)
    {
        this.node = node;
    }

    public void setRightChild(BSTNode rightChild)
    {
        this.rightChild = rightChild;
    }

```

```
50      public void setLeftChild(BSTNode leftChild)
51      {
52          this.leftChild = leftChild;
53      }
54
55      public void setParent(BSTNode parent)
56      {
57          this.parent = parent;
58      }
59
60      // Method to check if the node is a leaf
61      public boolean isLeaf() {
62          return this.leftChild == null && this.rightChild == null;
63      }
64  }
65
```

Interface.java Download

```
1  import java.io.BufferedReader;
2  import java.io.File;
3  import java.io.FileNotFoundException;
4  import java.io.FileReader;
5  import java.io.IOException;
6  import java.util.ArrayList;
7  import java.util.StringTokenizer;
8
9  public class Interface{
10
11
12    private static boolean commands(BSTDictionary dictionary, String command, ArrayList<String>
   splitUserInput)
13    {
14
15       if (command.equals("define"))
16       {
17             // Check size of arguments inputted and print invalid command if less than 2
18             if (splitUserInput.size() != 2)
19             {
20                System.out.println("Invalid command.");
21             }
22
23             //Get the word we want to add (label) from the input.
24             String label = splitUserInput.get(1);
25             //Create a new object with the word and the type 1
26             Key newKey = new Key(label, 1);
27             //Find the newRecord using the key
28             Record newRecord = dictionary.get(newKey);
29             // Check to see if word exists in dictionary, if not return null. Otherwise return the data stored.
30             if(newRecord == null)
31             {
32                System.out.println("the word " + label + " is not in the dictionary");
33
34             } else
35             {
36                String data = newRecord.getDataItem();
37                System.out.println(data);
38             }
39       }
40       else if(command.equals("translate"))
41       {
42          // Check size of arguments inputted and print invalid command if less than 2
43          if (splitUserInput.size() != 2)
44          {
45             System.out.println("Invalid input");
46          }
47
```

```
48          //Get the label from the command, add it with the corresponding type and find the key from the
    dictionary.
49          String label = splitUserInput.get(1);
50          Key newkey = new Key(label, 2);
51          Record newRecord = dictionary.get(newkey);
52
53          //If we dont find the word return a statement. Otherwise output the data we found.
54          if(newRecord == null)
55          {
56             System.out.println("There is no definition for the word " + label);
57          } else
58          {
59             String data = newRecord.getDataItem();
60             System.out.println(data);
61          }
62       }
63       else if (command.equals("sound"))
64       {
65          // Check size of arguments inputted and print invalid command if less than 2
66          if (splitUserInput.size() != 2)
67          {
68             System.out.println("Invalid input");
69          }
70
71          //Get the label from the command, add it with the corresponding type and find the key from the
    dictionary.
72          String label = splitUserInput.get(1);
73          Key newkey = new Key(label, 3);
74          Record newRecord = dictionary.get(newkey);
75
76          //If we dont find the file print an error message
77          if(newRecord == null)
78          {
79             System.out.println("There is no sound file for " + label);
80          }
81          //If we find the file start playing using SoundPlayer.java
82          else
83          {
84             String filePath = newRecord.getDataItem();
85             SoundPlayer player = new SoundPlayer();
86
87             try
88             {
89                player.play(filePath);
90             }
91             catch (MultimediaException e)
92             {
93                System.out.println(e.getMessage());
94             }
95          }
96       }
97       else if(command.equals("play"))
```

```
98      {
99          // Check size of arguments inputted and print invalid command if less than 2
100         if (splitUserInput.size() != 2)
101         {
102             System.out.println("Invalid input");
103         }
104
105         //Get the label from the command, add it with the corresponding type and find the key from the
    dictionary.
106         String label = splitUserInput.get(1);
107         Key finderKey = new Key(label, 4);
108         Record newRecord = dictionary.get(finderKey);
109
110         //If we dont find the file print an error message
111         if(newRecord == null)
112         {
113             System.out.println("There is no music file for " + label);
114         }
115         else
116         {
117             // Play the music file if it exists and handle any exceptions that may arise
118             String filePath = newRecord.getDataItem();
119             SoundPlayer player = new SoundPlayer();
120             try
121             {
122                 player.play(filePath);
123             }
124             catch (MultimediaException e)
125             {
126                 System.out.println(e.getMessage());
127             }
128         }
129     }
130     else if(command.equals("say"))
131     {
132         //Check size of arguments inputted and print invalid command if less than 2
133         if (splitUserInput.size() != 2)
134         {
135             System.out.println("Invalid input");
136         }
137
138         //Get the label from the command, add it with the corresponding type and find the key from the
    dictionary.
139         String label = splitUserInput.get(1);
140         Key finderKey = new Key(label, 5);
141         Record newRecord = dictionary.get(finderKey);
142
143         //When we cant find the file we print an error message.
144         if(newRecord == null)
145         {
146             System.out.println("There is no voice file for " + label);
147         }
```

```
148         else
149         {
150             // Play the audio file if it exists and handle any exceptions that may happen
151             String filePath = newRecord.getDataItem();
152             SoundPlayer player = new SoundPlayer();
153             try
154             {
155                 player.play(filePath);
156             }
157             catch (MultimediaException e)
158             {
159                 System.out.println(e.getMessage());
160             }
161         }
162     }
163     else if (command.equals("show"))
164     {
165 
166         //Check size of arguments inputted and print invalid command if less than 2
167         if (splitUserInput.size() != 2)
168         {
169             System.out.println("Invalid input");
170         }
171         else
172         {
173             //Get the label from the command, add it with the corresponding type and find the key from
    the dictionary.
174             String label = splitUserInput.get(1);
175             Key finderKey = new Key(label, 6);
176             Record newRecord = dictionary.get(finderKey);
177 
178             //When we cant find the file we print an error message.
179             if (newRecord == null)
180             {
181                 System.out.println("There is no image file for " + label);
182             }
183             //If we can find it then we get the picture and open it, handling exceptions that could happen
184             else
185             {
186                 String filePath = newRecord.getDataItem();
187                 PictureViewer viewer = new PictureViewer();
188                 try
189                 {
190                     viewer.show(filePath);
191                 }
192                 catch (MultimediaException e)
193                 {
194                     System.out.println(e.getMessage());
195                 }
196             }
197         }
198     }
```

```
199         else if (command.equals("animate")) {
200             // Check if the correct number of arguments is provided
201             if (splitUserInput.size() != 2)
202             {
203                 System.out.println("Invalid command.");
204             }
205             else
206             {
207                 String label = splitUserInput.get(1);
208                 Key finderKey = new Key(label, 7); // Create a key to search for animated image files
209                 Record newRecord = dictionary.get(finderKey);
210 
211                 // Check if the animated image file exists in the dictionary
212                 if (newRecord == null)
213                 {
214                     System.out.println("There is no animated image file for " + label);
215                 }
216                 else
217                 {
218                     // Display the animated image if it exists
219                     String filePath = newRecord.getDataItem();
220                     PictureViewer viewer = new PictureViewer();
221 
222                     try {
223 
224                         viewer.show(filePath);
225                     }
226                     catch (MultimediaException e)
227                     {
228                         // Handle any exceptions that occur while showing the file
229                         System.out.println(e.getMessage());
230                     }
231                 }
232             }
233         }
234 
235         else if (command.equals("browse"))
236         {
237         // Check if the correct number of arguments is provided
238             if (splitUserInput.size() != 2)
239             {
240                 System.out.println("Invalid command.");
241             }
242             else
243             {
244                 String label = splitUserInput.get(1);
245                 Key finderKey = new Key(label, 8); // Create a key to search for a webpage
246                 Record newRecord = dictionary.get(finderKey);
247 
248                 // Check if the webpage exists in the dictionary
249                 if (newRecord == null)
250                 {
```

```
                System.out.println("There is no webpage called " + label);
            }
            else
            {
                // Display the webpage if it exists
                String filePath = newRecord.getDataItem();
                ShowHTML htmlViewer = new ShowHTML();
                htmlViewer.show(filePath);
            }
        }
    }
    else if (command.equals("delete"))
    {
        // Check if the correct number of arguments is provided
        if (splitUserInput.size() != 3)
        {
            System.out.println("Invalid command.");
        }
        else
        {
            // Pull the type from the input
            String label = splitUserInput.get(1);
            int type = Integer.parseInt(splitUserInput.get(2));

            try
            {
                // Attempt to remove the element with the specified key from the dictionary
                dictionary.remove(new Key(label, type));
            }
            catch (DictionaryException e)
            {
                // Handle any exceptions during the deletion process
                System.out.println(e.getMessage());
            }
        }
    }

    else if (command.equals("add"))
    {
        // Ensure at least 4 elements are provided in the input
        if (splitUserInput.size() < 4)
        {
            System.out.println("Invalid command.");
        }
        else
        {
            String label = splitUserInput.get(1);
            int type = Integer.parseInt(splitUserInput.get(2)); // Parse the type from input
            // Combine the remaining input elements into a single data string
            String data = String.join(" ", splitUserInput.subList(3, splitUserInput.size()));

            Key newKey = new Key(label, type);
```

```
303             Record newRecord = new Record(newKey, data);
304
305             try
306             {
307                 // Add the new newRecord to the dictionary
308                 dictionary.put(newRecord);
309             }
310             catch (DictionaryException e)
311             {
312                 // Handle any exception during insertion
313                 System.out.println(e.getMessage());
314             }
315         }
316     }
317
318     else if (command.equals("list"))
319     {
320         // Check if exactly 2 elements are provided in the input
321         if (splitUserInput.size() != 2)
322         {
323             System.out.println("Invalid command.");
324         }
325         else
326         {
327             String prefix = splitUserInput.get(1);
328             ArrayList<String> matching = new ArrayList<>();
329             Key prefixKey = new Key(prefix, 1); // Create a key to search for matching prefixes
330
331             // Check if a newRecord with the exact prefix exists and add it to the list
332             if (dictionary.get(prefixKey) != null)
333             {
334                 matching.add(prefix);
335             }
336
337             // Find and add all successor keys that start with the same prefix
338             Record successor = dictionary.successor(prefixKey);
339             while (successor != null && successor.getKey().getLabel().startsWith(prefix))
340             {
341                 matching.add(successor.getKey().getLabel());
342                 successor = dictionary.successor(successor.getKey());
343             }
344
345             // Output matching elements or a message if none are found
346             if (matching.isEmpty())
347             {
348                 System.out.println("No label attributes in the ordered dictionary start with prefix " + prefix);
349             }
350             else
351             {
352                 for (int i = 0; i < matching.size(); i++)
353                 {
354                     String word = matching.get(i);
```

```
                if (i < matching.size() - 1)
                {
                    System.out.print(word + ", ");
                }
                else
                {
                    System.out.println(word);
                }
            }
        }
    }

    else if (command.equals("first"))
    {
        // Check if the command input size is exactly 1, otherwise print "Invalid command"
        if (splitUserInput.size() != 1) {
            System.out.println("Invalid command.");
        }
        else
        {
            // Retrieve the smallest newRecord in the dictionary
            Record smallest = dictionary.smallest();
            if (smallest != null)
            {
                // If a newRecord exists, retrieve and print its label, type, and data
                String label = smallest.getKey().getLabel();
                int type = smallest.getKey().getType();
                String data = smallest.getDataItem();
                System.out.println(label + ',' + type + ',' + data + '.');
            }
        }
    }
    else if (command.equals("last"))
    {
        // Check if the command input size is exactly 1, otherwise print "Invalid command"
        if (splitUserInput.size() != 1)
        {
            System.out.println("Invalid command.");
        }
        else
        {
            // Retrieve the largest newRecord in the dictionary
            Record largest = dictionary.largest();
            if (largest != null)
            {
                // If a newRecord exists, retrieve and print its label, type, and data
                String label = largest.getKey().getLabel();
                int type = largest.getKey().getType();
                String data = largest.getDataItem();
                System.out.println(label + ',' + type + ',' + data + '.');
            }
```

```
407         }
408      }
409      else if (command.equals("exit"))
410      {
411         // Check if the command input size is exactly 1, otherwise print "Invalid command"
412         if (splitUserInput.size() != 1)
413         {
414            System.out.println("Invalid command.");
415         }
416         else
417         {
418            // Return true to indicate that the program should exit
419            return true;
420         }
421      }
422      else
423      {
424         // Handle any unrecognized commands by printing "Invalid command"
425         System.out.println("Invalid command.");
426      }
427      return false;
428   }
429
430   public static void main(String[] args) throws IOException
431   {
432      BSTDictionary dictionary = new BSTDictionary();
433      String filename = args[0];
434      try
435      {
436         File myObj = new File(filename);
437         BufferedReader myReader = new BufferedReader(new FileReader(myObj));
438         String label;
439
440         while (myReader.ready())
441         {
442
443            label = myReader.readLine();
444            if (label == null)
445            {
446               break;
447            }
448
449
450            String nextLineData = myReader.readLine();
451            if (nextLineData == null)
452            {
453               break;
454               }
455
456            int type = checkFistLetter(nextLineData);
457            String data = nextLineData;
458
```

```
459             if (type == 2 || type == 3 || type == 4 || type == 5)
460             {
461                 data = nextLineData.substring(1);
462             }
463 
464             Key newKey = new Key(label.toLowerCase(), type);
465             Record newRecord = new Record(newKey, data.toLowerCase());
466 
467             try
468             {
469                 dictionary.put(newRecord);
470             }
471             catch (DictionaryException e)
472             {
473                 System.out.println("Error: Cannot insert data into dictionary key "+ "(" +label.toLowerCase()
    + ", "+ type +")" +" already exists");
474             }
475         }
476         myReader.close();
477     }
478     catch (FileNotFoundException e)
479     {
480         System.out.println("An error occurred.");
481         e.printStackTrace();
482     }
483 
484     while (true)
485     {
486         StringReader keyboard = new StringReader();
487         String line = keyboard.read("Enter next command: ");
488         ArrayList<String> splitUserInput = splitUserInput(line.trim());
489         String command = splitUserInput.get(0);
490 
491         boolean output = commands(dictionary, command, splitUserInput);
492         if (output) break;
493     }
494 }
495 
496 
497 
498 //Checks the index 0 of each line to find out what if statement will be executed.
499 private static int checkFistLetter(String data)
500 {
501     if(data.charAt(0) == '/')
502     {
503         //French
504         return 2;
505     }
506     else if(data.charAt(0) == '-')
507     {
508         //Sound
509         return 3;
```

```
510         }
511         else if(data.charAt(0) == '+')
512         {
513             //Music
514             return 4;
515         }
516         else if(data.charAt(0) == '*')
517         {
518             //Voice
519             return 5;
520         }
521 
522         else if (data.endsWith("jpg"))
523         {
524             //image
525             return 6;
526         }
527         else if (data.endsWith("gif"))
528         {
529             //gif
530             return 7;
531         }
532         else if (data.endsWith("html"))
533         {
534             //browse
535             return 8;
536         }
537         //define
538         return 1;
539     }
540 
541     /**
542      * This helper function splitUserInputs an input string into individual words based on spaces.
543      * It returns an ArrayList containing each token as a separate element.
544      *
545      * @param input The input string to be splitUserInput
546      * @return An ArrayList of strings containing each tokn separated with spaces in the input.
547      */
548     private static ArrayList<String> splitUserInput(String input) {
549         // Create an ArrayList to store the splitUserInput tokens from the input string
550         ArrayList<String> store = new ArrayList<>();
551         // Use a StringTokenizer to splitUserInput the input string by spaces
552         StringTokenizer tokenizer = new StringTokenizer(input, " ");
553 
554         // Iterate threw the tokens and add each to the ArrayList
555         while (tokenizer.hasMoreTokens())
556         {
557             store.add(tokenizer.nextToken());
558         }
559 
560         // Return the list of splitUserInput tokens
561         return store;
```

```
562     }
563
564 }
```

Record.java Download

```
1  public class Record
2  {
3
4     // Private fields for storing the key and data associated with the record
5     private Key theKey;
6     private String data;
7
8     // Constructor to create a new record with a specified key and data
9     public Record(Key theKey, String data)
10       {
11       this.theKey = theKey;
12       this.data = data;
13    }
14
15    // Getter method to retrieve the key of the record
16    public Key getKey()
17       {
18       return theKey;
19    }
20
21    // Getter method to retrieve the data associated with the record
22    public String getDataItem()
23       {
24       return data;
25    }
26 }
27
28
```

Key.java

Download

```java
public class Key {
   //Declate the instance variables
   private String label;
   private int type;

   //A constructor which initializes a new Keyobject with the specified parameters
   public Key(String theLabel, int theType)
   {
      this.label = theLabel.toLowerCase();
      this.type = theType;
   }

   public String getLabel()
   {
      return label;
   }

   public int getType()
   {
      return type;
   }

   //Two Key objects A and B are equal if A.label = B.label and A.type = B.type.
   public int compareTo(Key k) {

      int compared_label = this.label.compareTo(k.getLabel());
   int compared_type = Integer.compare(this.type, k.getType()); // Using Integer.compare for clarity

   // Compare labels first
   if (compared_label < 0)
   {
      return -1;
   }
   else if (compared_label > 0)
   {
      return 1;
   }
   else
   {
      // If labels are equal, compare types
      if (compared_type < 0)
      {
         return -1;
      }
      else if (compared_type > 0)
      {
         return 1;
      }
      // Both label and type are equal
```

```
50          else
51          {
52              return 0;
53          }
54      }
55  }
56
57  }
58
59
60
```