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)
        {
            // If the key of the new record is greater, check if the right child is null
            if (r.getRightChild() == null)
            {
                // If right child is null, insert the new record here
                r.setRightChild(new BSTNode(d));
            }
            else
            {
                // If right child is not null, continue recursion on the right subtree
                insertRecursively(r.getRightChild(), d);
            }
        }
        else
        {
            // If the key of the new record matches the current node's key, throw an exception
            throw new DictionaryException("Key already exsists in the tree");
        }
    }

    // Returns the node storing the given key; returns null if the key is not found
    public BSTNode get(BSTNode r, Key k)
    {
        if (r == null)
        {
            return null;
        }
        if (k.compareTo(r.getRecord().getKey()) < 0)
        {
            return get(r.getLeftChild(), k);
        }
        else if (k.compareTo(r.getRecord().getKey()) > 0)
        {
            return get(r.getRightChild(), k);
        }
        else
        {
            return r;
        }
    }

    // Removes the node with the given key from the tree
    public void remove(BSTNode r, Key k) throws DictionaryException
    {
        root = removeRecursively(r, k);
    }

    private BSTNode removeRecursively(BSTNode r, Key k) throws DictionaryException
    {
        if (r == null)
        {
            throw new DictionaryException("Key not found in the tree");
        }
```

```java
        if (k.compareTo(r.getRecord().getKey()) < 0)
        {
            r.setLeftChild(removeRecursively(r.getLeftChild(), k));
        }
        else if (k.compareTo(r.getRecord().getKey()) > 0)
        {
            r.setRightChild(removeRecursively(r.getRightChild(), k));
        }
        else
        {
            if (r.getLeftChild() == null)
            {
                return r.getRightChild();
            }
            else if (r.getRightChild() == null)
            {
                return r.getLeftChild();
            }
            BSTNode smallestNode = smallest(r.getRightChild());
            r.setRecord(smallestNode.getRecord());
            r.setRightChild(removeRecursively(r.getRightChild(), smallestNode.getRecord().getKey()));
        }
        return r;
    }

    // Returns the node with the smallest key in tree with root r
    public BSTNode smallest(BSTNode r)
    {
        if (r == null || r.getLeftChild() == null)
        {
            return r;
        }
        return smallest(r.getLeftChild());
    }

    // Returns the node with the largest key in tree with root r
    public BSTNode largest(BSTNode r)
    {
        if (r == null || r.getRightChild() == null)
        {
            return r;
        }
        return largest(r.getRightChild());
    }
}
```

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));
```

```java
                    node.getRightChild().setLeftChild(new BSTNode(null));
                    node.getRightChild().setRightChild(new BSTNode(null));
                }
                else
                {
                    insertRecursively(node.getRightChild(), d);
                }
            }
            else
            {
                throw new DictionaryException("Key already exists in the dictionary");
            }
        }
    }

    // Method to get the Record associated with a given Key
    public Record get(Key k)
    {
        BSTNode node = getRecursively(root, k);
        if (node != null && node.getRecord() != null)
        {
            return node.getRecord();
        }
        return null; // Key not found
    }

    private BSTNode getRecursively(BSTNode node, Key k)
    {
        if (node == null || node.getRecord() == null)
        {
            return null;
        }
        int comparison = k.compareTo(node.getRecord().getKey());

        if (comparison < 0)
        {
            return getRecursively(node.getLeftChild(), k);
        }
        else if (comparison > 0)
        {
            return getRecursively(node.getRightChild(), k);
        }
        else
        {
            return node; // Found the node
        }
    }

    // Removes a node with the specified key from the tree
    public void remove(Key k) throws DictionaryException
    {
    root = removeRecursively(root, k);
```

```
}

private BSTNode removeRecursively(BSTNode node, Key k) throws DictionaryException
{
if (node == null || node.getRecord() == null)
{
    throw new DictionaryException("Key not found in the dictionary");
}

int comparison = k.compareTo(node.getRecord().getKey());
if (comparison < 0)
{
    node.setLeftChild(removeRecursively(node.getLeftChild(), k));
}
else if (comparison > 0)
{
    node.setRightChild(removeRecursively(node.getRightChild(), k));
}
else
{
    // Node with the key found
    if (node.getLeftChild() == null || node.getLeftChild().getRecord() == null)
    {
        return node.getRightChild(); // Replace with right child if left is null or a leaf
    }
    else if (node.getRightChild() == null || node.getRightChild().getRecord() == null)
    {
        return node.getLeftChild(); // Replace with left child if right is null or a leaf
    }

    // Node with two children: Find the smallest node in the right subtree
    BSTNode current = node.getRightChild();
    while (current.getLeftChild() != null && current.getLeftChild().getRecord() != null)
    {
        current = current.getLeftChild();
    }

    // Replace node's record with the smallest node's record
    node.setRecord(current.getRecord());

    // Remove the smallest node in the right subtree
    node.setRightChild(removeRecursively(node.getRightChild(), current.getRecord().getKey()));
}

return node;

}


public Record successor(Key k)
{
BSTNode current = root;
```

```
    BSTNode successor = null;

    while (current != null && current.getRecord() != null)
    {
        int comparison = k.compareTo(current.getRecord().getKey());
        if (comparison < 0)
        {
            // Possible successor; move to the left subtree
            successor = current;
            current = current.getLeftChild();
        }
        else
        {
            // Move to the right subtree
            current = current.getRightChild();
        }
    }

    // If we found a successor, return the associated Record
    if (successor != null)
    {
        return successor.getRecord();
    }

    return null; // No successor found
}

     // Finds and returns the predecessor Record of the given key
    public Record predecessor(Key k)
    {
    BSTNode current = root;
    BSTNode predecessor = null;

    while (current != null && current.getRecord() != null)
    {
        int comparison = k.compareTo(current.getRecord().getKey());
        if (comparison > 0)
        {
            // Possible predecessor; move to the right subtree
            predecessor = current;
            current = current.getRightChild();
        }
        else
        {
            // Move to the left subtree
            current = current.getLeftChild();
        }
    }

    // If we found a predecessor, return the associated Record
    if (predecessor != null)
    {
```

```java
        return predecessor.getRecord();
    }

    return null; // No predecessor found
    }

    // Finds and returns the node with the smallest key in the subtree with root r
    public Record smallest()
    {
        if (root == null || root.getRecord() == null)
        {
            return null; // Handle empty tree case
        }
        BSTNode current = root;
        while (current.getLeftChild() != null && current.getLeftChild().getRecord() != null)
        {
            current = current.getLeftChild();
        }

        return current.getRecord(); // Return the smallest node's record
    }


    public Record largest()
    {
        if (root == null || root.getRecord() == null)
        {
            return null; // Handle empty tree case
        }
        BSTNode current = root;
        while (current.getRightChild() != null && current.getRightChild().getRecord() != null)
        {
            current = current.getRightChild();
        }
        return current.getRecord(); // Return the largest node's record
    }

}

```

▾ **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;
    }

    public void setLeftChild(BSTNode leftChild)
    {
        this.leftChild = leftChild;
    }

    public void setParent(BSTNode parent)
    {
        this.parent = parent;
    }

    // Method to check if the node is a leaf
    public boolean isLeaf() {
        return this.leftChild == null && this.rightChild == null;
    }
}

```

Interface.java Download

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.StringTokenizer;

public class Interface{


    private static boolean commands(BSTDictionary dictionary, String command, ArrayList<String> splitUserInput)
    {

        if (command.equals("define"))
        {
                // Check size of arguments inputted and print invalid command if less than 2
                if (splitUserInput.size() != 2)
                {
                    System.out.println("Invalid command.");
                }

                //Get the word we want to add (label) from the input.
                String label = splitUserInput.get(1);
                //Create a new object with the word and the type 1
                Key newKey = new Key(label, 1);
                //Find the newRecord using the key
                Record newRecord = dictionary.get(newKey);
                // Check to see if word exists in dictionary, if not return null. Otherwise return the data stored.
                if(newRecord == null)
                {
                    System.out.println("the word " + label + " is not in the dictionary");

                } else
                {
                    String data = newRecord.getDataItem();
                    System.out.println(data);
                }
        }
        else if(command.equals("translate"))
        {
            // Check size of arguments inputted and print invalid command if less than 2
            if (splitUserInput.size() != 2)
            {
                System.out.println("Invalid input");
            }

            //Get the label from the command, add it with the corresponding type and find the key from the dictionary.
            String label = splitUserInput.get(1);
            Key newkey = new Key(label, 2);
            Record newRecord = dictionary.get(newkey);

            //If we dont find the word return a statement. Otherwise output the data we found.
            if(newRecord == null)
            {
                System.out.println("There is no definition for the word " + label);
            } else
            {
                String data = newRecord.getDataItem();
                System.out.println(data);
            }
        }
        else if (command.equals("sound"))
        {
            // Check size of arguments inputted and print invalid command if less than 2
            if (splitUserInput.size() != 2)
            {
                System.out.println("Invalid input");
            }

            //Get the label from the command, add it with the corresponding type and find the key from the dictionary.
            String label = splitUserInput.get(1);
            Key newkey = new Key(label, 3);
            Record newRecord = dictionary.get(newkey);

            //If we dont find the file print an error message
            if(newRecord == null)
            {
                System.out.println("There is no sound file for " + label);
            }
            //If we find the file start playing using SoundPlayer.java
            else
            {
                String filePath = newRecord.getDataItem();
                SoundPlayer player = new SoundPlayer();

                try
                {
                    player.play(filePath);
                }
                catch (MultimediaException e)
                {
                    System.out.println(e.getMessage());
                }
            }
        }
        else if(command.equals("play"))
        {
            // Check size of arguments inputted and print invalid command if less than 2
            if (splitUserInput.size() != 2)
            {
                System.out.println("Invalid input");
            }

            //Get the label from the command, add it with the corresponding type and find the key from the dictionary.
            String label = splitUserInput.get(1);
            Key finderKey = new Key(label, 4);
            Record newRecord = dictionary.get(finderKey);

            //If we dont find the file print an error message
            if(newRecord == null)
            {
                System.out.println("There is no music file for " + label);
            }
            else
            {
                // Play the music file if it exists and handle any exceptions that may arise
                String filePath = newRecord.getDataItem();
                SoundPlayer player = new SoundPlayer();
                try
                {
                    player.play(filePath);
                }
                catch (MultimediaException e)
                {
                    System.out.println(e.getMessage());
                }
            }
        }
        else if(command.equals("say"))
        {
            //Check size of arguments inputted and print invalid command if less than 2
            if (splitUserInput.size() != 2)
            {
                System.out.println("Invalid input");
            }

            //Get the label from the command, add it with the corresponding type and find the key from the dictionary.
            String label = splitUserInput.get(1);
            Key finderKey = new Key(label, 5);
            Record newRecord = dictionary.get(finderKey);

            //When we cant find the file we print an error message.
            if(newRecord == null)
            {
                System.out.println("There is no voice file for " + label);
            }
            else
            {
                // Play the audio file if it exists and handle any exceptions that may happen
                String filePath = newRecord.getDataItem();
                SoundPlayer player = new SoundPlayer();
                try
                {
                    player.play(filePath);
                }
                catch (MultimediaException e)
                {
                    System.out.println(e.getMessage());
                }
            }
        }
        else if (command.equals("show"))
        {

            //Check size of arguments inputted and print invalid command if less than 2
            if (splitUserInput.size() != 2)
            {
                System.out.println("Invalid input");
            }
            else
            {
                //Get the label from the command, add it with the corresponding type and find the key from the dictionary.
                String label = splitUserInput.get(1);
                Key finderKey = new Key(label, 6);
                Record newRecord = dictionary.get(finderKey);

                //When we cant find the file we print an error message.
                if (newRecord == null)
                {
                    System.out.println("There is no image file for " + label);
                }
                //If we can find it then we get the picture and open it, handling exceptions that could happen
                else
                {
                    String filePath = newRecord.getDataItem();
                    PictureViewer viewer = new PictureViewer();
                    try
                    {
                        viewer.show(filePath);
                    }
                    catch (MultimediaException e)
                    {
                        System.out.println(e.getMessage());
                    }
                }
            }
        }
```

```java
        else if (command.equals("animate")) {
            // Check if the correct number of arguments is provided
            if (splitUserInput.size() != 2)
            {
                System.out.println("Invalid command.");
            }
            else
            {
                String label = splitUserInput.get(1);
                Key finderKey = new Key(label, 7); // Create a key to search for animated image files
                Record newRecord = dictionary.get(finderKey);

                // Check if the animated image file exists in the dictionary
                if (newRecord == null)
                {
                    System.out.println("There is no animated image file for " + label);
                }
                else
                {
                    // Display the animated image if it exists
                    String filePath = newRecord.getDataItem();
                    PictureViewer viewer = new PictureViewer();

                    try {

                        viewer.show(filePath);
                    }
                    catch (MultimediaException e)
                    {
                        // Handle any exceptions that occur while showing the file
                        System.out.println(e.getMessage());
                    }
                }
            }
        }

        else if (command.equals("browse"))
        {
        // Check if the correct number of arguments is provided
            if (splitUserInput.size() != 2)
            {
                System.out.println("Invalid command.");
            }
            else
            {
                String label = splitUserInput.get(1);
                Key finderKey = new Key(label, 8); // Create a key to search for a webpage
                Record newRecord = dictionary.get(finderKey);

                // Check if the webpage exists in the dictionary
                if (newRecord == null)
                {
```

```
                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);
```

```
                Record newRecord = new Record(newKey, data);

                try
                {
                    // Add the new newRecord to the dictionary
                    dictionary.put(newRecord);
                }
                catch (DictionaryException e)
                {
                    // Handle any exception during insertion
                    System.out.println(e.getMessage());
                }
            }
        }

        else if (command.equals("list"))
        {
            // Check if exactly 2 elements are provided in the input
            if (splitUserInput.size() != 2)
            {
                System.out.println("Invalid command.");
            }
            else
            {
                String prefix = splitUserInput.get(1);
                ArrayList<String> matching = new ArrayList<>();
                Key prefixKey = new Key(prefix, 1); // Create a key to search for matching prefixes

                // Check if a newRecord with the exact prefix exists and add it to the list
                if (dictionary.get(prefixKey) != null)
                {
                    matching.add(prefix);
                }

                // Find and add all successor keys that start with the same prefix
                Record successor = dictionary.successor(prefixKey);
                while (successor != null && successor.getKey().getLabel().startsWith(prefix))
                {
                    matching.add(successor.getKey().getLabel());
                    successor = dictionary.successor(successor.getKey());
                }

                // Output matching elements or a message if none are found
                if (matching.isEmpty())
                {
                    System.out.println("No label attributes in the ordered dictionary start with prefix " + prefix);
                }
                else
                {
                    for (int i = 0; i < matching.size(); i++)
                    {
                        String word = matching.get(i);
```

```java
                        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 + '.');
                }
            }
        }
        else if (command.equals("exit"))
        {
            // Check if the command input size is exactly 1, otherwise print "Invalid command"
            if (splitUserInput.size() != 1)
            {
                System.out.println("Invalid command.");
            }
            else
            {
                // Return true to indicate that the program should exit
                return true;
            }
        }
        else
        {
            // Handle any unrecognized commands by printing "Invalid command"
            System.out.println("Invalid command.");
        }
        return false;
}

    public static void main(String[] args) throws IOException
    {
        BSTDictionary dictionary = new BSTDictionary();
        String filename = args[0];
        try
        {
            File myObj = new File(filename);
            BufferedReader myReader = new BufferedReader(new FileReader(myObj));
            String label;

            while (myReader.ready())
            {

                label = myReader.readLine();
                if (label == null)
                {
                    break;
                }


                String nextLineData = myReader.readLine();
                if (nextLineData == null)
                {
                    break;
                    }

                int type = checkFistLetter(nextLineData);
                String data = nextLineData;

                if (type == 2 || type == 3 || type == 4 || type == 5)
                {
                    data = nextLineData.substring(1);
                }

                Key newKey = new Key(label.toLowerCase(), type);
                Record newRecord = new Record(newKey, data.toLowerCase());

                try
                {
                    dictionary.put(newRecord);
                }
                catch (DictionaryException e)
                {
                    System.out.println("Error: Cannot insert data into dictionary key "+ "(" +label.toLowerCase() + ", "+ type +")" +" already exists");
                }
            }
            myReader.close();
        }
        catch (FileNotFoundException e)
        {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }

        while (true)
        {
            StringReader keyboard = new StringReader();
            String line = keyboard.read("Enter next command: ");
            ArrayList<String> splitUserInput = splitUserInput(line.trim());
            String command = splitUserInput.get(0);

            boolean output = commands(dictionary, command, splitUserInput);
            if (output) break;
        }
    }



    //Checks the index 0 of each line to find out what if statement will be executed.
    private static int checkFistLetter(String data)
    {
            if(data.charAt(0) == '/')
            {
                //French
                return 2;
            }
            else if(data.charAt(0) == '-')
            {
                //Sound
                return 3;
```

```java
            }
            else if(data.charAt(0) == '+')
            {
                //Music
                return 4;
            }
            else if(data.charAt(0) == '*')
            {
                //Voice
                return 5;
            }

            else if (data.endsWith("jpg"))
            {
                //image
                return 6;
            }
            else if (data.endsWith("gif"))
            {
                //gif
                return 7;
            }
            else if (data.endsWith("html"))
            {
                //browse
                return 8;
            }
            //define
            return 1;
    }

    /**
     * This helper function splitUserInputs an input string into individual words based on spaces.
     * It returns an ArrayList containing each token as a separate element.
     *
     * @param input The input string to be splitUserInput
     * @return An ArrayList of strings containing each tokn separated with spaces in the input.
     */
    private static ArrayList<String> splitUserInput(String input) {
        // Create an ArrayList to store the splitUserInput tokens from the input string
        ArrayList<String> store = new ArrayList<>();
        // Use a StringTokenizer to splitUserInput the input string by spaces
        StringTokenizer tokenizer = new StringTokenizer(input, " ");

        // Iterate threw the tokens and add each to the ArrayList
        while (tokenizer.hasMoreTokens())
        {
            store.add(tokenizer.nextToken());
        }

        // Return the list of splitUserInput tokens
        return store;
    }

}
```

▾ **Record.java**

Download

```java
public class Record
{

    // Private fields for storing the key and data associated with the record
    private Key theKey;
    private String data;

    // Constructor to create a new record with a specified key and data
    public Record(Key theKey, String data)
        {
        this.theKey = theKey;
        this.data = data;
    }

    // Getter method to retrieve the key of the record
    public Key getKey()
        {
        return theKey;
    }

    // Getter method to retrieve the data associated with the record
    public String getDataItem()
        {
        return data;
    }
}


```

▾ **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
        else
        {
            return 0;
        }
    }
}

}



```