**Question 1: (chapter 27)**

Write a program that is the beginning of a spell-checker.

First, your program will create the dictionary with the following animal names:

Second, The program will read from the user a word, parse the word, see if it is in the hash table. It if is, display a message saying "Correct", and if not display the message "Not Correct", print the word and print the closest word in the dictionary as a potential correction (which is the word in the entry that this word parsed/hashed to). You may use any hashing function to parse the word, and you may use any table size.

Code

***MyHashSet Class***

// based on lecture 27 slides

import java.util.\*;

public class MyHashSet<E> implements Collection<E> {

private final static int DEFAULT\_INITIAL\_CAPACITY = 4;

private final static int MAXIMUM\_CAPACITY = 1 << 30;

private int capacity;

private final static float DEFAULT\_MAX\_LOAD\_FACTOR = 0.75f;

private float loadFactorThreshold;

private int size = 0;

private LinkedList<E>[] table;

public MyHashSet() {

this(DEFAULT\_INITIAL\_CAPACITY, DEFAULT\_MAX\_LOAD\_FACTOR);

}

public MyHashSet(int initialCapacity) {

this(initialCapacity, DEFAULT\_MAX\_LOAD\_FACTOR);

}

public MyHashSet(int initialCapacity, float loadFactorThreshold) {

if (initialCapacity > MAXIMUM\_CAPACITY)

this.capacity = MAXIMUM\_CAPACITY;

else

this.capacity = trimToPowerOf2(initialCapacity);

this.loadFactorThreshold = loadFactorThreshold;

table = new LinkedList[capacity];

}

@Override

public void clear() {

size = 0;

removeElements();

}

@Override

public boolean contains(Object e) {

int bucketIndex = hash(e.hashCode());

if (table[bucketIndex] != null) {

LinkedList<E> bucket = table[bucketIndex];

return bucket.contains(e);

}

return false;

}

@Override

public boolean add(E e) {

if (contains(e))

return false;

if (size + 1 > capacity \* loadFactorThreshold) {

if (capacity == MAXIMUM\_CAPACITY)

throw new RuntimeException("Exceeding maximum capacity");

rehash();

}

int bucketIndex = hash(e.hashCode());

if (table[bucketIndex] == null) {

table[bucketIndex] = new LinkedList<E>();

}

table[bucketIndex].add(e);

size++;

return true;

}

@Override

public boolean remove(Object e) {

if (!contains(e))

return false;

int bucketIndex = hash(e.hashCode());

if (table[bucketIndex] != null) {

LinkedList<E> bucket = table[bucketIndex];

bucket.remove(e);

}

size--;

return true;

}

@Override

public boolean isEmpty() {

return size == 0;

}

@Override

public int size() {

return size;

}

@Override

public java.util.Iterator<E> iterator() {

return new MyHashSetIterator(this);

}

private class MyHashSetIterator implements java.util.Iterator<E> {

private java.util.ArrayList<E> list;

private int current = 0;

private MyHashSet<E> set;

public MyHashSetIterator(MyHashSet<E> set) {

this.set = set;

list = setToList();

}

@Override

public boolean hasNext() {

return current < list.size();

}

@Override

public E next() {

return list.get(current++);

}

}

private int hash(int hashCode) {

return hashCode & (capacity - 1);

}

private int trimToPowerOf2(int initialCapacity) {

int capacity = 1;

while (capacity < initialCapacity) {

capacity <<= 1;

}

return capacity;

}

private void removeElements() {

for (int i = 0; i < capacity; i++) {

if (table[i] != null) {

table[i].clear();

}

}

}

private void rehash() {

java.util.ArrayList<E> list = setToList();

capacity <<= 1;

table = new LinkedList[capacity];

size = 0;

for (E element: list) {

add(element);

}

}

private java.util.ArrayList<E> setToList() {

java.util.ArrayList<E> list = new java.util.ArrayList<>();

for (int i = 0; i < capacity; i++) {

if (table[i] != null) {

for (E e: table[i]) {

list.add(e);

}

}

}

return list;

}

@Override

public String toString() {

java.util.ArrayList<E> list = setToList();

StringBuilder builder = new StringBuilder("[");

for (int i = 0; i < list.size() - 1; i++) {

builder.append(list.get(i) + ", ");

}

if (list.size() == 0)

builder.append("]");

else

builder.append(list.get(list.size() - 1) + "]");

return builder.toString();

}

@Override

public boolean addAll(Collection<? extends E> arg0) {

return false;

}

@Override

public boolean containsAll(Collection<?> arg0) {

return false;

}

@Override

public boolean removeAll(Collection<?> arg0) {

return false;

}

@Override

public boolean retainAll(Collection<?> arg0) {

return false;

}

@Override

public Object[] toArray() {

return null;

}

@Override

public <T> T[] toArray(T[] arg0) {

return null;

}

}

***MySet Interface***

// based on lecture 27 slides

public interface MySet<E> extends java.lang.Iterable<E> {

public void clear();

public boolean contains(E e);

public boolean add(E e);

public boolean remove(E e);

public boolean isEmpty();

public int size();

}

***Dictionary Class***

package Question1;

import java.util.LinkedList;

import java.util.Scanner;

public class Dictionary {

private static MyHashSet<String> set = new MyHashSet<>();

private static LinkedList<String> suggest = new LinkedList<String>();

public static void main(String[] args) {

Scanner input = new Scanner (System.in);

addValues();

System.out.println("Please enter string to look up");

String value = input.nextLine();

checkValues(value);

}

private static void checkValues(String string) {

if (set.contains(string)) {

System.out.println("Correct -> " + string);

}

else if (!set.contains(string)){

System.out.println("Not Correct -> " + string);

swap(string);

capz(string);

lwrz(string);

if (!suggest.isEmpty()) {

System.out.println(string + " for suggestion");

for (int i = 0; i<suggest.size();i++) {

System.out.println(suggest.get(i) + ",");

}

}

}

}

//ant

private static void lwrz(String string) {

MyHashSet<String> lwSet2 = lowSet();

String word = "";

word += Character.toUpperCase(string.charAt(0));

if(lwSet2.contains(string) ) {

for(int i = 1; i<string.length();i++) {

word += string.charAt(i);

}

suggest.add(word);

}

}

//ANT

private static void capz(String string) {

MyHashSet<String> hwSet2 = hwSet();

String word = "";

word += string.charAt(0);

if(hwSet2.contains(string) ) {

for(int i = 1; i<string.length();i++) {

word += Character.toLowerCase(string.charAt(i));

}

suggest.add(word);

}

}

private static MyHashSet<String> hwSet() {

MyHashSet<String> set2 = new MyHashSet<>();

for (String s: set) {

set2.add(s.toUpperCase());

}

return set2;

}

//laligator

private static void swap(String string) {

MyHashSet<String> LwSet2 = lowSet();

String newWord;

StringBuffer buffer = new StringBuffer(string);

StringBuffer swap = new StringBuffer(string);

for (int i=0; i<string.length()-1; i++) {

swap = new StringBuffer(buffer.substring(i, i+2)).reverse();

buffer.delete(i,i+2);

newWord = buffer.insert(i, swap).toString();

if(LwSet2.contains(newWord)) {

lwrz(newWord);

}

buffer = new StringBuffer(string);

}

}

private static MyHashSet<String> lowSet() {

MyHashSet<String> set2 = new MyHashSet<>();

for (String s: set) {

set2.add(s.toLowerCase());

}

return set2;

}

private static void addValues() {

set.add("Aardvark");

set.add("Abyssinian");

set.add("Adelie Penguin");

set.add("Affenpinscher");

set.add("Afghan Hound");

set.add("African Bush Elephant");

set.add("African Civet");

set.add("African Clawed Frog");

set.add("African Forest Elephant");

set.add("African Palm Civet");

set.add("African Penguin");

set.add("African Tree Toad");

set.add("African Wild Dog");

set.add("Ainu Dog");

set.add("Airedale Terrier");

set.add("Akbash");

set.add("Akita");

set.add("Alaskan Malamute");

set.add("Albatross");

set.add("Aldabra Giant Tortoise");

set.add("Alligator");

set.add("Alpine Dachsbracke");

set.add("American Bulldog");

set.add("American Cocker Spaniel");

set.add("American Coonhound");

set.add("American Eskimo Dog");

set.add("American Foxhound");

set.add("American Pit Bull Terrier");

set.add("American Staffordshire Terrier");

set.add("American Water Spaniel");

set.add("Anatolian Shepherd Dog");

set.add("Angelfish");

set.add("Ant");

set.add("Anteater");

set.add("Antelope");

set.add("Appenzeller Dog");

set.add("Arctic Fox");

set.add("Arctic Hare");

set.add("Arctic Wolf");

set.add("Armadillo");

set.add("Asian Elephant");

set.add("Asian Giant Hornet");

set.add("Asian Palm Civet");

set.add("Asiatic Black Bear");

set.add("Australian Cattle Dog");

set.add("Australian Kelpie Dog");

set.add("Australian Mist");

set.add("Australian Shepherd");

set.add("Australian Terrier");

set.add("Avocet");

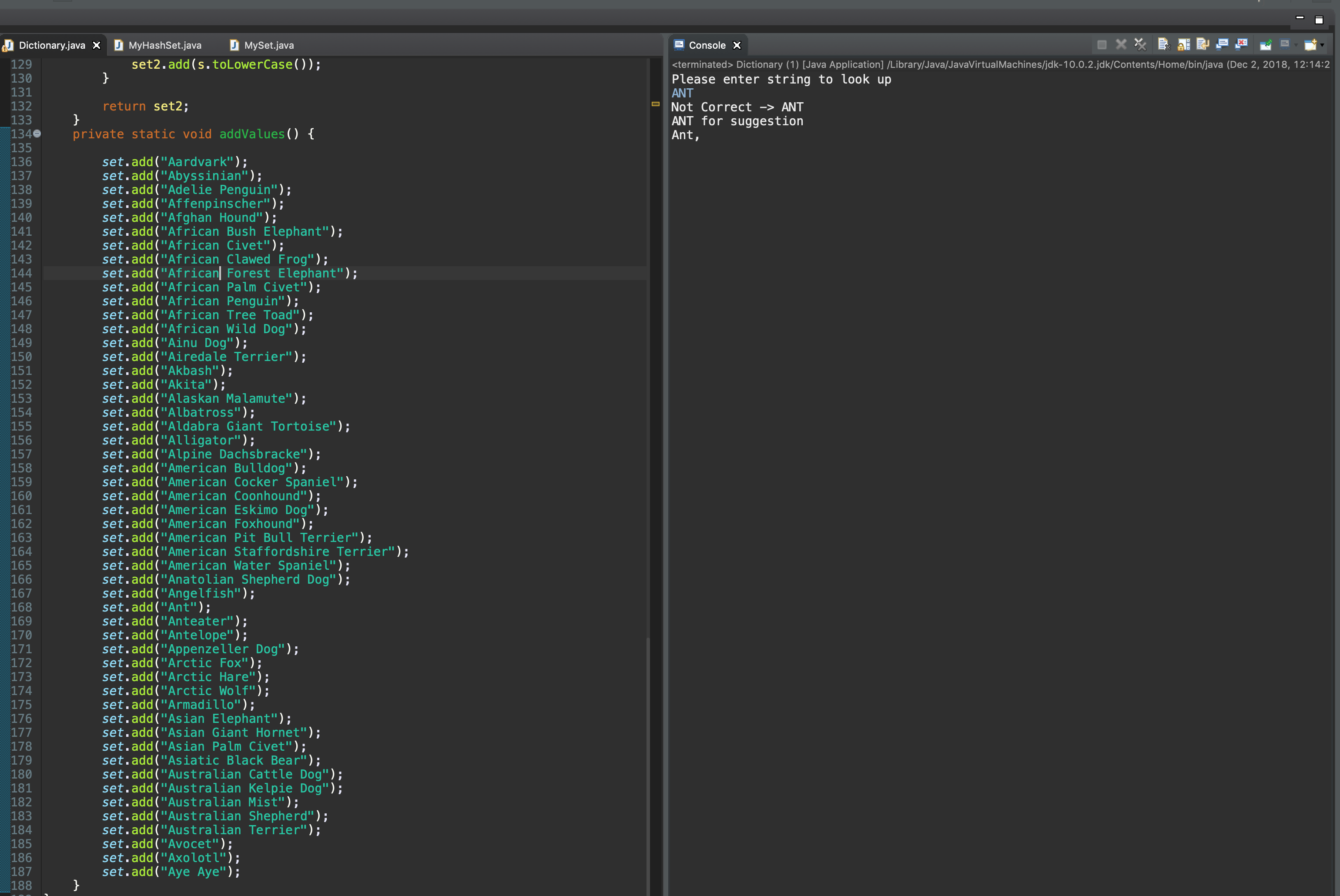
set.add("Axolotl");

set.add("Aye Aye");

}

}

**Screenshots**



**Question 2: (Chapter 28)**

Write a program that reads a connected graph from a user. Then the program will prompt the user to enter the name of two vertices, check if they exist and if so, the program will display the shortest path between these two vertices.

***UnweightedGraph.class***

// code from Lecture Slides 28

import java.util.\*;

public class UnweightedGraph<V> implements Graph<V> {

protected List<V> vertices = new ArrayList<>();

protected List<List<Edge>> neighbors

= new ArrayList<>();

public UnweightedGraph() {

}

public UnweightedGraph(V[] vertices, int[][] edges) {

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

addVertex(vertices[i]);

createAdjacencyLists(edges, vertices.length);

}

public UnweightedGraph(List<V> vertices, List<Edge> edges) {

for (int i = 0; i < vertices.size(); i++)

addVertex(vertices.get(i));

createAdjacencyLists(edges, vertices.size());

}

public UnweightedGraph(List<Edge> edges, int numberOfVertices) {

for (int i = 0; i < numberOfVertices; i++)

addVertex((V)(new Integer(i)));

createAdjacencyLists(edges, numberOfVertices);

}

public UnweightedGraph(int[][] edges, int numberOfVertices) {

for (int i = 0; i < numberOfVertices; i++)

addVertex((V)(new Integer(i)));

createAdjacencyLists(edges, numberOfVertices);

}

private void createAdjacencyLists(

int[][] edges, int numberOfVertices) {

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

addEdge(edges[i][0], edges[i][1]);

}

}

private void createAdjacencyLists(

List<Edge> edges, int numberOfVertices) {

for (Edge edge: edges) {

addEdge(edge.u, edge.v);

}

}

@Override

public int getSize() {

return vertices.size();

}

@Override

public List<V> getVertices() {

return vertices;

}

@Override

public V getVertex(int index) {

return vertices.get(index);

}

@Override

public int getIndex(V v) {

return vertices.indexOf(v);

}

@Override

public List<Integer> getNeighbors(int index) {

List<Integer> result = new ArrayList<>();

for (Edge e: neighbors.get(index))

result.add(e.v);

return result;

}

@Override

public int getDegree(int v) {

return neighbors.get(v).size();

}

@Override

public void printEdges() {

for (int u = 0; u < neighbors.size(); u++) {

System.out.print(getVertex(u) + " (" + u + "): ");

for (Edge e: neighbors.get(u)) {

System.out.print("(" + getVertex(e.u) + ", " +

getVertex(e.v) + ") ");

}

System.out.println();

}

}

@Override

public void clear() {

vertices.clear();

neighbors.clear();

}

@Override

public boolean addVertex(V vertex) {

if (!vertices.contains(vertex)) {

vertices.add(vertex);

neighbors.add(new ArrayList<Edge>());

return true;

}

else {

return false;

}

}

@Override

public boolean addEdge(Edge e) {

if (e.u < 0 || e.u > getSize() - 1)

throw new IllegalArgumentException("No such index: " + e.u);

if (e.v < 0 || e.v > getSize() - 1)

throw new IllegalArgumentException("No such index: " + e.v);

if (!neighbors.get(e.u).contains(e)) {

neighbors.get(e.u).add(e);

return true;

}

else {

return false;

}

}

@Override

public boolean addEdge(int u, int v) {

return addEdge(new Edge(u, v));

}

@Override

public SearchTree dfs(int v) {

List<Integer> searchOrder = new ArrayList<>();

int[] parent = new int[vertices.size()];

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

parent[i] = -1;

boolean[] isVisited = new boolean[vertices.size()];

dfs(v, parent, searchOrder, isVisited);

return new SearchTree(v, parent, searchOrder);

}

private void dfs(int v, int[] parent, List<Integer> searchOrder,

boolean[] isVisited) {

searchOrder.add(v);

isVisited[v] = true;

for (Edge e : neighbors.get(v)) {

int w = e.v;

if (!isVisited[w]) {

parent[w] = v;

dfs(w, parent, searchOrder, isVisited);

}

}

}

@Override

public SearchTree bfs(int v) {

List<Integer> searchOrder = new ArrayList<>();

int[] parent = new int[vertices.size()];

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

parent[i] = -1;

java.util.LinkedList<Integer> queue =

new java.util.LinkedList<>();

boolean[] isVisited = new boolean[vertices.size()];

queue.offer(v);

isVisited[v] = true;

while (!queue.isEmpty()) {

int u = queue.poll();

searchOrder.add(u);

for (Edge e: neighbors.get(u)) {

int w = e.v;

if (!isVisited[w]) {

queue.offer(w);

parent[w] = u;

isVisited[w] = true;

}

}

}

return new SearchTree(v, parent, searchOrder);

}

public class SearchTree {

private int root;

private int[] parent;

private List<Integer> searchOrder;

public SearchTree(int root, int[] parent,

List<Integer> searchOrder) {

this.root = root;

this.parent = parent;

this.searchOrder = searchOrder;

}

public int getRoot() {

return root;

}

public int getParent(int v) {

return parent[v];

}

public List<Integer> getSearchOrder() {

return searchOrder;

}

public int getNumberOfVerticesFound() {

return searchOrder.size();

}

public List<V> getPath(int index) {

ArrayList<V> path = new ArrayList<>();

do {

path.add(vertices.get(index));

index = parent[index];

}

while (index != -1);

return path;

}

public void printPath(int index) {

List<V> path = getPath(index);

System.out.print("A path from " + vertices.get(root) + " to " +

vertices.get(index) + ": ");

for (int i = path.size() - 1; i >= 0; i--)

System.out.print(path.get(i) + " ");

}

public void printTree() {

System.out.println("Root is: " + vertices.get(root));

System.out.print("Edges: ");

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

if (parent[i] != -1) {

System.out.print("(" + vertices.get(parent[i]) + ", " +

vertices.get(i) + ") ");

}

}

System.out.println();

}

}

@Override

public boolean remove(V v) {

return true;

}

@Override

public boolean remove(int u, int v) {

return true;

}

}

***Edge.class***

*//code from chapter 28*

public class Edge {

int u;

int v;

public Edge (int u, int v){

this.u = u;

this.v =v;

}

public boolean equals(Object o){

return u == ((Edge)o).u && v == ((Edge)o).v;

}

}

***Graph interface***

// code from Lecture Slides 28

public interface Graph<V> {

public int getSize();

public java.util.List<V> getVertices();

public V getVertex(int index);

public int getIndex(V v);

public java.util.List<Integer> getNeighbors(int index);

public int getDegree(int v);

public void printEdges();

public void clear();

public boolean addVertex(V vertex);

public boolean addEdge(int u, int v);

public boolean addEdge(Edge e);

public boolean remove(V v);

public boolean remove(int u, int v);

public UnweightedGraph<V>.SearchTree dfs(int v);

public UnweightedGraph<V>.SearchTree bfs(int v);

}

***TestGraph.class***

*/\* Similar to ex 28.1 from Book*

*\* txt file taken from http://liveexample.pearsoncmg.com/test/GraphSample1.txt*

*code from Lecture Slides 28*

*\*/*

*import java.io.File;*

*import java.util.Scanner;*

*import java.util.ArrayList;*

*public class TestGraph {*

*public static void main(String[] args) throws Exception {*

*Scanner input = new Scanner (System.in);*

*System.out.println("Enter the file name to import the data");*

*String s = input.nextLine();*

*//Users/techyouknow/eclipse-workspace/Assigment13/dat.txt*

*File file = new File (s);*

*String[] vertices = assignVertices(file);*

*ArrayList<Edge> edgeList = new ArrayList<>();*

*edgeList = assiginEdges(file,vertices);*

*//System.out.println(Arrays.toString(vertices));*

*Graph<String> graph2 = new UnweightedGraph<>*

*(java.util.Arrays.asList(vertices), edgeList);*

*/\*System.out.println("\nThe number of vertices in graph: "*

*+ graph2.getSize());*

*System.out.println("The edges for graph2:");*

*graph2.printEdges();\*/*

*System.out.println("Checking for Two Vertices");*

*System.out.println("Please enter the first Vertice");*

*String x = String.valueOf(input.nextInt());*

*System.out.println("Please enter the second Vertice");*

*String y = String.valueOf(input.nextInt());*

*System.out.println("Path");*

*shortPath(x,y,graph2);*

*/\*for (int i = 0 ; i<vertices.length; i++) {*

*String str= read.nextLine();*

*System.out.println(str);*

*String [] line = str.split("[\\s+]");*

*System.out.println(Arrays.toString(line));*

*//String u = line [0];*

*//vertices[i] = u;*

*}*

*System.out.println(vertices[3]);*

*\*/*

*/\**

*\**

*System.out.println("Enter the no of vertices");*

*String[] vertices = new String [input.nextInt()];*

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

*System.out.println("Enter each "+ i + " vertices");*

*vertices [i] = input.next();*

*}*

*\**

*\*/*

*/\* Working condition program*

*\**

*\**

*\* String[] vertices = {"0","1","2","3","4","5"};*

*int[][] edges = {*

*{0, 1}, {0, 2},*

*{1, 0}, {1, 3},*

*{2, 0}, {2, 3},{2,4},*

*{3, 1}, {3, 2},{3,4},{3,5},*

*{4, 2}, {4, 3}, {4,5},*

*{5, 3}, {5, 4},*

*};*

*Graph<String> graph1 = new UnweightedGraph<>(vertices, edges);*

*System.out.println("The number of vertices in graph1: "*

*+ graph1.getSize());*

*\*/*

*}*

*private static void shortPath(String x, String y, Graph<String> graph2 ) {*

*boolean a = validVertices(x,graph2);*

*boolean b = validVertices(y,graph2);*

*if (a && b) {*

*UnweightedGraph<String>.SearchTree dfs = graph2.dfs(graph2.getIndex(x));*

*java.util.List<Integer> searchOrders = dfs.getSearchOrder();*

*for (int i = 0; i < searchOrders.size(); i++) {*

*System.out.print(graph2.getVertex(searchOrders.get(i)) + " ");*

*if (graph2.getVertex(searchOrders.get(i)).equals(y)) {*

*break;*

*}*

*}*

*}*

*}*

*private static boolean validVertices(String x, Graph<String> graph2) {*

*boolean set = false;*

*if (graph2.getIndex(x) == -1) {*

*System.out.println("Vertice " + x + " doesnt exists");*

*set = false;*

*}*

*else {*

*System.out.println("Vertice " + x + " does exists ");*

*set = true;*

*}*

*return set;*

*}*

*private static ArrayList<Edge> assiginEdges(File file, String[] vertices) throws Exception {*

*ArrayList<Edge> edgeList = new ArrayList<>();*

*Scanner read = new Scanner (file);*

*read.nextLine();*

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

*String s = read.nextLine();*

*String [] str = s.split("[\\s+]");*

*// System.out.println(Arrays.toString(str));*

*// if (vertices[i] == str[0]) {*

*for(int j = 1; j<str.length;j++) {*

*edgeList.add(new Edge(Integer.parseInt(vertices[i]),Integer.parseInt(str[j])));*

*}*

*}*

*read.close();*

*return edgeList;*

*}*

*private static String[] assignVertices(File file) throws Exception {*

*Scanner read = new Scanner (file);*

*String[] vertices = new String [read.nextInt()];*

*read.nextLine();*

*while(read.hasNextLine()) {*

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

*vertices [i] = String.valueOf(read.nextInt());*

*read.nextLine();*

*}*

*}*

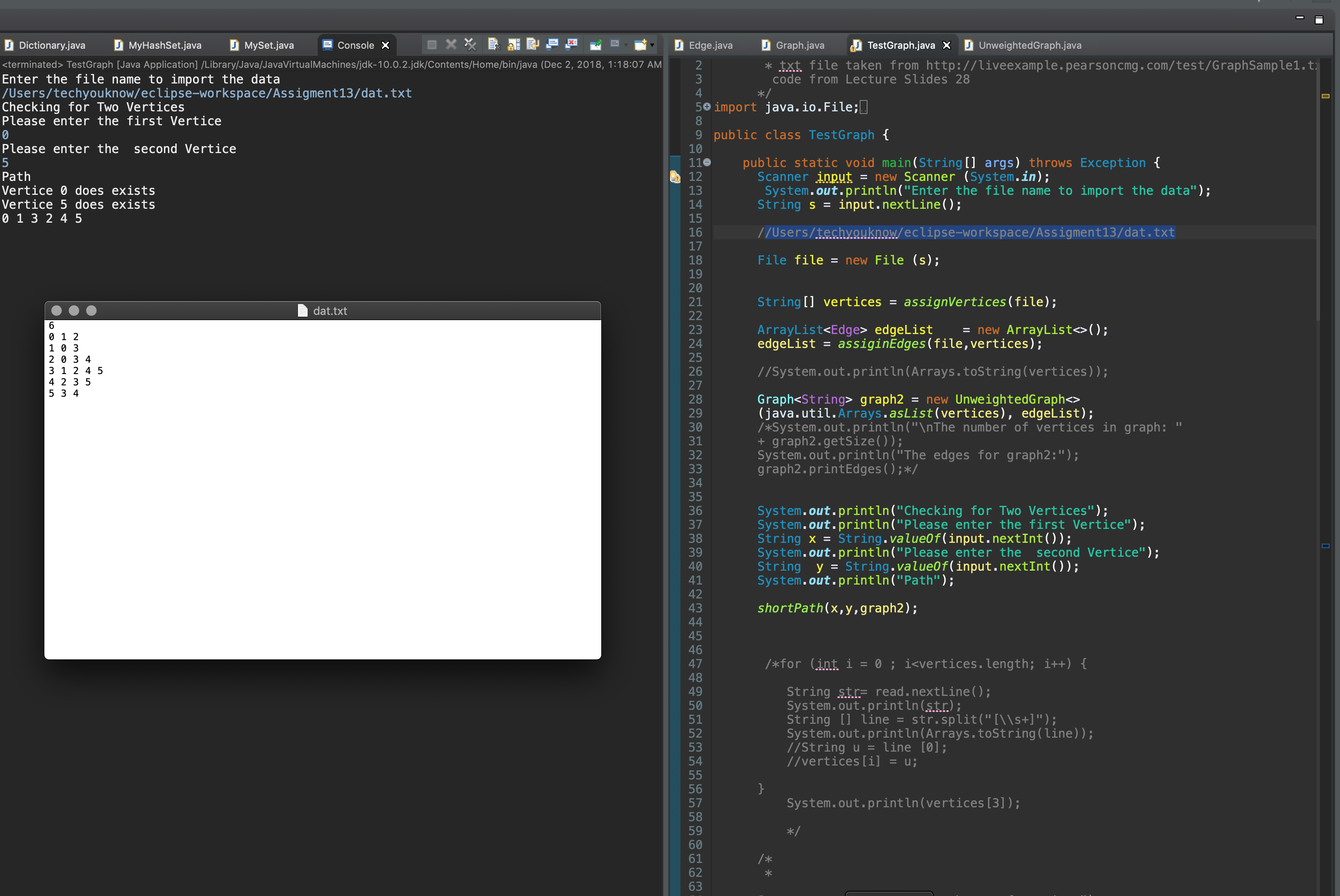
*read.close();*

*return vertices;*

*}*

*}*

**Screenshot**

**