import java.io.BufferedReader;

import java.util.Scanner;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

/\*\*

\* @author Samuel Swedberg

\* @version 10/14/22

\*

\* A client that demonstrates reading text files and determine if they are either a line, word, or character palindrome.

\*/

public class Client {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

String directory = "", answer = "";

boolean tryAgain = true;

while(tryAgain)

{

boolean inValidInput = true;

//Loops until valid input is found

while(inValidInput)

{

System.out.println("Enter starting directory: ");

directory = sc.nextLine();

try {

if(!new File(directory).exists())

throw new FileNotFoundException("You must only enter a directory.");

inValidInput = false;

// FileNotFoundException

} catch (FileNotFoundException fnfe) {

System.out.println(fnfe.getMessage() + " Press a key to continue or N to leave: ");

answer = sc.nextLine();

// Asks user if they want to continue

if(answer.equalsIgnoreCase("N"))

tryAgain = false;

break;

}

}

LinkedQueue<String> txt = new LinkedQueue<>();

LinkedQueue<String> returnLine = new LinkedQueue<>();

LinkedQueue<String> returnWord = new LinkedQueue<>();

LinkedQueue<String> returnChar = new LinkedQueue<>();

// Runs only if valid input is found

if(!inValidInput)

{

File file = new File(directory);

File[] files = file.listFiles();

LinkedQueue<String> queue = new LinkedQueue<>();

// Runs only if files exist

if(files != null)

{

LinkedQueue<String> lineQueue = new LinkedQueue<>();

LinkedQueue<String> wordQueue = new LinkedQueue<>();

LinkedQueue<String> charQueue = new LinkedQueue<>();

// For loop to run through all files

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

{

try

{

if(!new File(directory).exists())

throw new FileNotFoundException();

File f = files[i];

Scanner scanLine = new Scanner( f );

Scanner scanWord = new Scanner( f );

FileReader fr = new FileReader( f );

BufferedReader scanChar = new BufferedReader( fr );

String txtName = f.getName();

txt.enqueue(txtName);

// Scans line

while( scanLine.hasNext() )

{

String stringLine = scanLine.nextLine();

lineQueue.enqueue(stringLine);

}

// Scans words

while( scanWord.hasNext() )

{

String stringWord = scanWord.next();

wordQueue.enqueue(stringWord);

// Cleans up for char check

stringWord = stringWord.replaceAll("\\p{P}", "");

stringWord = stringWord.replaceAll("\\s", "");

stringWord = stringWord.replaceAll("\\n", "");

stringWord = stringWord.replaceAll("\\t", "");

stringWord = stringWord.toLowerCase();

// Splits word into chars

if(!("end".equals(stringWord)))

{

String stringChar[] = stringWord.split("(?!^)");

for(String character : stringChar)

charQueue.enqueue(character);

}

}

returnLine.enqueue(isPalindrome(lineQueue)); // Line palindrome check

returnWord.enqueue(isPalindrome(wordQueue)); // Word palindrome check

returnChar.enqueue(isPalindrome(charQueue)); // Char palindrome check

scanLine.close();

} catch ( FileNotFoundException fnfe )

{

System.out.println("Unable to find file.");

}

catch ( IOException ioe )

{

ioe.printStackTrace();

}

}

}

printTable(txt, returnLine, returnWord, returnChar); // Prints ASCII Table

System.out.println("Press a key to continue or N to leave: ");

answer = sc.nextLine();

// Asks user if they want to continue

if(answer.equalsIgnoreCase("N"))

tryAgain = false;

}

}

}

/\*

Checks if parameter is a palindrome or not

\*/

public static String isPalindrome(LinkedQueue queue)

{

String returnStr = "";

boolean isEqual = false;

int trueCleanSize = queue.size();

// Cleans up string

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

{

String stringRead = ( String ) queue.first();

stringRead = stringRead.replaceAll("\\p{P}", "");

stringRead = stringRead.replaceAll("\\s", "");

stringRead = stringRead.replaceAll("\\n", "");

stringRead = stringRead.replaceAll("\\t", "");

stringRead = stringRead.toLowerCase();

if("end".equals(stringRead))

{

queue.dequeue();

}

else if("".equals(stringRead))

{

queue.dequeue();

}

else

{

queue.dequeue();

//System.out.println(stringRead); //! Remove final

queue.enqueue(stringRead);

}

}

// Holds strings to compare

LinkedStack<String> stack1 = new LinkedStack<>();

LinkedQueue<String> queue2 = new LinkedQueue<>();

int trueQueueSize = queue.size();

// Puts first half of parameter into stack

for(int i=0; i<trueQueueSize/2; i++)

{

String temp = (String) queue.first();

stack1.push(temp);

queue.dequeue();

}

int trueAfter = queue.size();

// Puts the remaining of parameter in queue

while(!(queue.size() == 0))

{

String temp = (String) queue.first();

queue2.enqueue(temp);

queue.dequeue();

}

// Removes one if total is odd

if((trueAfter % 2 == 0) == false)

{

queue2.dequeue();

}

int check = stack1.size() + queue2.size();

// Checks if parameter is a palindrome

for(int i=0; i<check/2; i++)

{

if(stack1.top().equals(queue2.first()))

{

stack1.pop();

queue2.dequeue();

isEqual = true;

}

else

{

isEqual = false;

}

}

// Returns Y

if(isEqual == true)

{

returnStr = "Y";

return returnStr;

}

// Returns N

else

{

returnStr = "N";

return returnStr;

}

}

/\*

Prints out an ASCII table

\*/

public static void printTable( LinkedQueue txt, LinkedQueue character, LinkedQueue word, LinkedQueue line )

{

// Header for ascii table

System.out.printf("+------------+-------------+--------+--------+\n");

System.out.printf("| Filename | Character | Word | Line |\n");

System.out.printf("+------------+-------------+--------+--------+\n");

// Prints out data

int trueSize = txt.size();

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

{

String returnTxt = ( String ) txt.first();

String returnChar = ( String ) character.first();

String returnWord = ( String ) word.first();

String retirnLine = ( String ) line.first();

System.out.printf("| %s | %s | %s | %s |\n", returnTxt, returnChar, returnWord, retirnLine);

txt.dequeue();

character.dequeue();

word.dequeue();

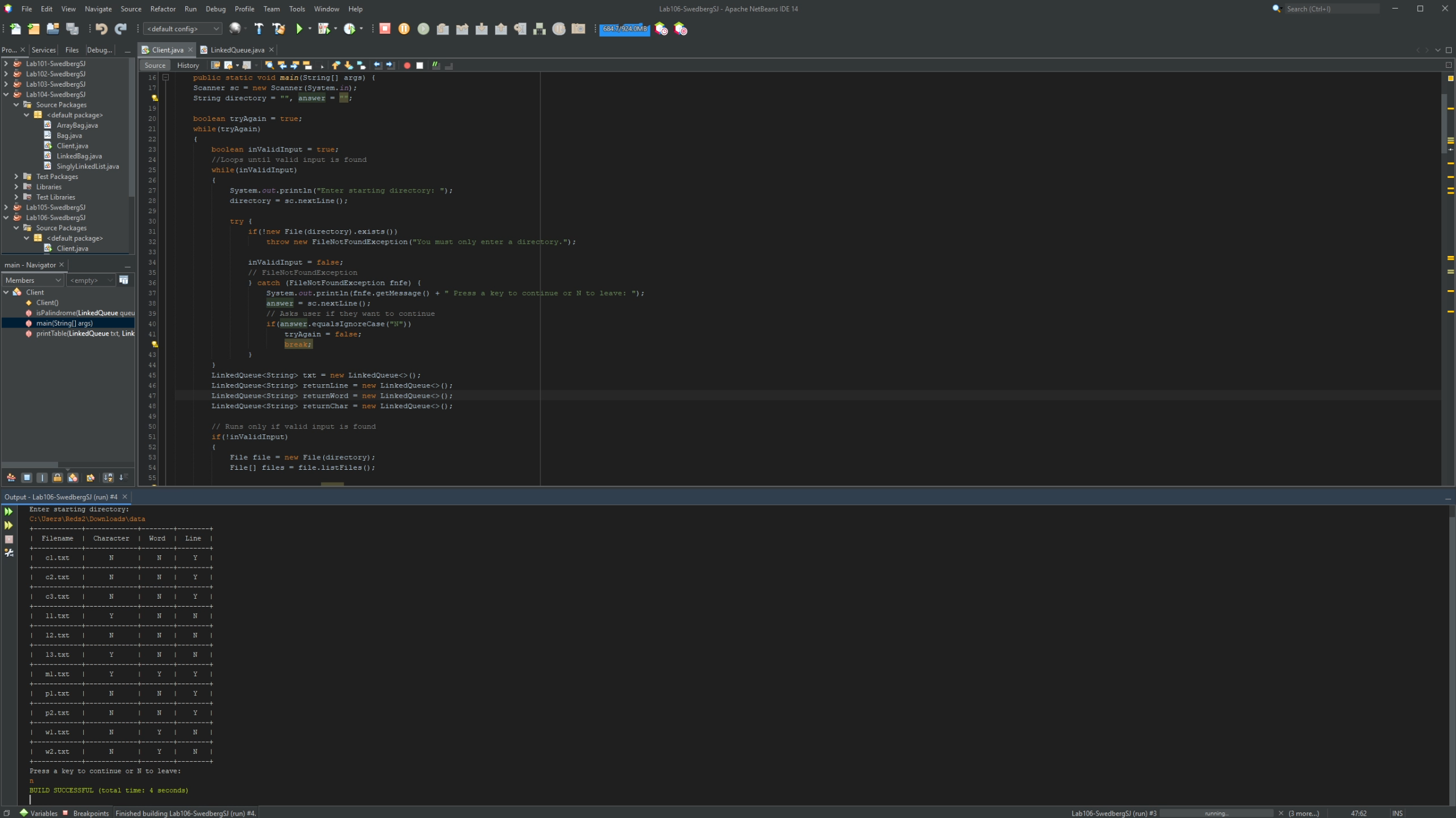
line.dequeue();

System.out.printf("+------------+-------------+--------+--------+\n");

}

}

}



/\*\*

\* LinkedQueue Class

\* Code Fragments 6.11

\* from

\* Data Structures & Algorithms, 6th edition

\* by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 10/14/22

\*/

/\*\* Realization of a FIFO queue as an adaptation of a SinglyLinkedList. \*/

public class LinkedQueue<E> implements Queue<E> {

private SinglyLinkedList<E> list = new SinglyLinkedList<>( ); // an empty list

public LinkedQueue( ) { } // new queue relies on the initially empty list

public int size( ) { return list.size( ); }

public boolean isEmpty( ) { return list.isEmpty( ); }

public void enqueue(E element) { list.addLast(element); }

public E first( ) { return list.first( ); }

public E dequeue( ) { return list.removeFirst( ); }

}

/\*\*

\* LinkedStack Class

\* Code Fragments 6.4

\* from

\* Data Structures & Algorithms, 6th edition

\* by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 10/14/22

\*/

public class LinkedStack<E> implements Stack<E> {

private SinglyLinkedList<E> list = new SinglyLinkedList<>( ); // an empty list

public LinkedStack( ) { } // new stack relies on the initially empty list

public int size( ) { return list.size( ); }

public boolean isEmpty( ) { return list.isEmpty( ); }

public void push(E element) { list.addFirst(element); }

public E top( ) { return list.first( ); }

public E pop( ) { return list.removeFirst( ); }

}

/\*\*

\* Queue Class

\* Code Fragments 6.9

\* from

\* Data Structures & Algorithms, 6th edition

\* by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 10/14/22

\*/

public interface Queue<E> {

/\*\* Returns the number of elements in the queue. \*/

int size( );

/\*\* Tests whether the queue is empty. \*/

boolean isEmpty( );

/\*\* Inserts an element at the rear of the queue. \*/

void enqueue(E e);

/\*\* Returns, but does not remove, the first element of the queue (null if empty). \*/

E first( );

/\*\* Removes and returns the first element of the queue (null if empty). \*/

E dequeue( );

}

/\*\*

\* SinglyLinkedList Class

\* Code Fragments 3.14, 3.15

\* from

\* Data Structures & Algorithms, 6th edition

\* by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 10/14/22

\*/

public class SinglyLinkedList<E> {

//---- nested Node class -----

private static class Node<E> {

private E element; // reference to the element stored at this node

private Node<E> next; // reference to the subsequent node in the list

public Node(E e, Node<E> n) {

element = e;

next = n;

}

public E getElement( ) { return element; }

public Node<E> getNext( ) { return next; }

public void setNext(Node<E> n) { next = n; }

}

// instance variables of the SinglyLinkedList

private Node<E> head = null; // head node of the list (or null if empty)

private Node<E> tail = null; // last node of the list (or null if empty)

private int size = 0; // number of nodes in the lis

public SinglyLinkedList() {} // constructs an initially empty list

// access methods

public int size( ) { return size; }

public boolean isEmpty( ) { return size == 0; }

public E first( ) { // returns (but does not remove) the first element

if (isEmpty( )) return null;

return head.getElement( );

}

public E last( ) { // returns (but does not remove) the last element

if (isEmpty( )) return null;

return tail.getElement( );

}

// update methods

public void addFirst(E e) { // adds element e to the front of the list

head = new Node<>(e, head); // create and link a new node

if (size == 0)

tail = head; // special case: new node becomes tail also

size++;

}

public void addLast(E e) { // adds element e to the end of the list

Node<E> newest = new Node<>(e, null); // node will eventually be the tail

if (isEmpty( ))

head = newest; // special case: previously empty list

else

tail.setNext(newest); // new node after existing tail

tail = newest; // new node becomes the tail

size++;

}

public E removeFirst( ) { // removes and returns the first element

if (isEmpty( )) return null; // nothing to remove

E answer = head.getElement( );

head = head.getNext( ); // will become null if list had only one node

size--;

if (size == 0)

tail = null; // special case as list is now empty

return answer;

}

}

/\*\*

\* A collection of objects that are inserted and removed according to the last-in

\* first-out principle. Although similar in purpose, this interface differs from

\* java.util.Stack.

\*

\* Stack Class

\* Code Fragments 6.1

\* from

\* Data Structures & Algorithms, 6th edition

\* by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 10/14/22

\*/

public interface Stack<E> {

/\*\*

\* Returns the number of elements in the stack.

\* @return number of elements in the stack

\*/

int size( );

/\*\*

\* Tests whether the stack is empty.

\* @return true if the stack is empty, false otherwise

\*/

boolean isEmpty( );

/\*\*

\* Inserts an element at the top of the stack.

\* @param e the element to be inserted

\*/

void push(E e);

/\*\*

\* Returns, but does not remove, the element at the top of the stack.

\* @return top element in the stack (or null if empty)

\*/

E top( );

/\*\*

\* Removes and returns the top element from the stack.

\* @return element removed (or null if empty)

\*/

E pop( );

}