1) Trace the call f(16) to the following method by showing a stack of activation records:

As shown below, f(16) will return 3.



2) Creativity Exercise: C – 5.11:

Pseudocode:

**Algorithm:** log\_2( n )

**Input:** int n of the operation log2n

**Output:** the integer part of the result of the operation log2n

**Code:**

**if** n < 2 **then**

**return** 0

**else**

**return** 1 + log­\_2(n/2)

Java implementation:

/\*\*

\* Basic algorithm to compute the integer part of log\_2(n)

\* @author Jacob Huesman

\*/

public class log\_2 {

public static int log\_2(int n) {

if(n < 2) {

return 0;

} else {

return 1 + log\_2(n / 2);

}

}

}

3) Reinforcement Exercise: R – 6.7:

Suppose an initially empty queue Q has performed a total of 32 enqueue operations, 10 first operations and 15 dequeue operations, 5 of which returned null to indicate an empty queue. What is the current size of Q?

There were 32 enqueue operations. Which indicates that 32 objects were added to Q.

The 10 first operations can be disregarded, since they don’t remove the object from the queue.

There were only 10 dequeue operations that actually removed something from the queue.

So the the size of the queue has to be (32 – 10) or 22.

4) Creativity Exercise C- 6.19:

**Algorithm:** postfix(exp)

**Input:** exp is a String representation of the postfix expression to be evaluated

**Output:** the result of the expression

**Throws**:NotAnIntegerException

**Code:**

{ Covert exp to a char array and remove any spaces }

set char[] arr = exp.toCharArray()

set char[] pos = new char[arr.length()]

**for** (i = 0, j=0; i < arr.length(); i++) **do**

**if** (arr[i] != ‘ ‘) **then**

set pos[j] = arr[i]

increment j

{ Evaluate the expression }

declare int back, x

declare stack to store integers

**for** (i = 0; i < array.length(); i++) **do**

**switch** (array[i])

**case** ‘+’:

stack.push(stack.pop() + stack.pop())

**break**

**case** ‘-‘:

back = stack.pop();

stack.push(stack.pop() - back)

**break**

**case** ‘\*’:

stack.push(stack.pop() \* stack.pop())

**break**

**case** ‘/’:

back = stack.pop()

stack.push(stack.pop() / back)

**break**

**default**:

set x = array[i].toInt()

**throw** NotAnIntegerException if conversion is unsuccessful

stack.push( x )

**return** stack.pop()

5) Creativity Exercise C- 6.34:

**34 min way:**

Take Maizie and Daisy: 4 min

Return with Maizie: 2 min

Take Crazy and Lazy: 20 min

Return with Daisy: 4 min

Take Daisy and Maizie: 4 min

Total: 34 min

**Faster way:**

Take Crazy and Lazy: 20 min

Return: 2 min (assuming Bob moves at the rate of the fastest cow and doesn’t need the yoke to actually transport the cow)

Take Maizie: 2 min

Return: 2 min

Take Daisy: 4 min

Total: 30 min

6) Programming Project:

Note: I did not include the GFile Package as it has a single purpose unrelated to the main purpose of the assignment. Which is to get files. I also didn’t include log\_2.java, since the code is already in task 2.

Client:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package hom03;

import GFile.\*;

import java.io.File;

import java.util.Scanner;

/\*\*

\* Client program that tests to see if given files read the same top to bottom as they do bottom to top

\* @author Jacob Huesman

\*/

public class Client {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

/\* Declare and initialize objects and variables \*/

Scanner scan = new Scanner(System.in);

Scanner file;

String[] files;

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

try {

do {

/\*\*

\* Allow the user to specify the name (and location) of one or more input file(s)

\*/

System.out.println("Please enter your files...");

files = GFile.getFiles();

System.out.println("List of files selected:");

for (String file1 : files) {

System.out.println(file1);

}

/\*\*

\* Process each input file in the order in which they were entered by the user.

\*/

for (String filePath : files) {

/\* Create a new scanner object to scan lines from \*/

file = new Scanner(new File(filePath));

/\* Tell user current file \*/

System.out.println("\nCurrent file: " + filePath);

/\* Prime the while loop \*/

boolean cont = true, reads = false;

String line;

/\*\*

\* This while loop will loop through the file adding each line to a stack until two lines in a row match, at which point a nested loop takes over.

\*/

while(cont) {

if(file.hasNextLine()){

line = file.nextLine();

/\* Stop execution if "end" is encountered \*/

if(line.trim().equalsIgnoreCase("end")){

break;

}

/\* Empty lines are ignored logically. However they are printed to retain proper formatting. \*/

else if(line.trim().equals("")){

System.out.println(line);

continue;

}

/\* Print out line being added to the stack \*/

System.out.println(line);

/\* This decision structure adds lines to the stack until two lines match up. \*/

if(stack.first() == null){

stack.push(line);

} else if(StringOps.equalsIgnoreCaseAndPunctuation(stack.first(), line)){

stack.pop();

/\* Loop through the rest of the file comparing to the entries in the stack. If they all match up then the file reads the same forward and back. \*/

while(cont){

line = file.nextLine();

if(line.trim().equals("")){

System.out.println(line);

} else if(file.hasNextLine() && StringOps.equalsIgnoreCaseAndPunctuation(line, stack.pop())){

System.out.println(line);

} else if(line.equalsIgnoreCase("end")){

reads = true;

cont = false;

break;

} else {

System.out.println(line);

while(file.hasNextLine()){

line = file.nextLine();

if(!line.trim().equalsIgnoreCase("end")){

System.out.println(line);

}

}

break;

}

}

} else {

stack.push(line);

}

} else {

break;

}

}

/\* Display this message if the file reads the same forward and back \*/

if( reads ){

System.out.println("\nThe passage reads the same top to bottom (or bottom to top).");

} else {

System.out.println("\nThe passage did not read the same from the top to bottom.");

}

}

/\* Ask the user if they would like to process more files. \*/

System.out.print("\nWould you like to process additional files? (yes/no): ");

String response = scan.nextLine();

if(!response.trim().equalsIgnoreCase("yes")){

System.out.println("");

break;

}

} while(true);

System.out.println("");

} catch (Exception e) {

System.out.println(e);

}

}

}

Linked Stack:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package hom03;

/\*\*

\* SLL implementation of the stack data structure.

\* @author Jacob Huesman

\* @param <T> Type of objects to be contained in the stack

\*/

public class LinkedStack <T> {

/\*\*

\* The objects of this class form the nodes of the singly linked list.

\* @param <T> Type of the object the node will hold

\*/

private static class Node<T> {

private final T element;

private Node<T> next;

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

element = e;

next = n;

}

public T getElement(){ return element; }

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

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

}

private Node<T> stackHead;

int count;

/\*\*

\* Default constructor.

\*/

public LinkedStack(){

stackHead = null;

count = 0;

}

/\*\*

\* Returns a count of the items in the stack.

\* @return A count of items in stack

\*/

public int getCurrentSize() {

return count;

}

/\*\*

\* Checks if the stack is empty.

\* @return true - if empty; false - otherwise

\*/

public boolean isEmpty() {

return count <= 0;

}

/\*\*

\* Adds an element to the beginning of the list.

\* @param item The item the node will point to

\* @return true - when the operation is successful

\*/

public boolean push(T item) {

stackHead = new Node(item, stackHead);

count++;

return true;

}

/\*\*

\* Removes and returns the first node in the list.

\* @return The element, or null if the list is empty

\*/

public T pop() {

if(stackHead == null){

return null;

}

T element = stackHead.getElement();

stackHead = stackHead.getNext();

count--;

return element;

}

/\*\*

\* Clears the stack.

\*/

public void clear() {

stackHead = null;

count = 0;

}

/\*\*

\* Returns the first item in the stack.

\* @return First item in the stack

\*/

public T first(){

if (stackHead!= null){

return stackHead.getElement();

}

return null;

}

}

StringOps:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package hom03;

/\*\*

\* Convenient String Operations

\* @author Jacob Huesman

\*/

public class StringOps {

/\*\*

\* Eliminates all punctuation and spacing from two strings before comparing the two ignoring the case.

\* @param str1 First string for comparison

\* @param str2 Second string for comparison

\* @return true - if the strings match in the absence of case and punctuation; false - otherwise

\*/

public static boolean equalsIgnoreCaseAndPunctuation(String str1, String str2){

String str1mod, str2mod;

//Get rid of spaces

str1mod = str1.replaceAll(" ", "");

str2mod = str2.replaceAll(" ", "");

//Get rid of punctuation

str1mod = str1mod.replaceAll("!", "");

str2mod = str2mod.replaceAll("!", "");

str1mod = str1mod.replaceAll("\\?", "");

str2mod = str2mod.replaceAll("\\?", "");

str1mod = str1mod.replaceAll(",", "");

str2mod = str2mod.replaceAll(",", "");

str1mod = str1mod.replaceAll("\\.", "");

str2mod = str2mod.replaceAll("\\.", "");

str1mod = str1mod.replaceAll(":", "");

str2mod = str2mod.replaceAll(":", "");

str1mod = str1mod.replaceAll(";", "");

str2mod = str2mod.replaceAll(";", "");

str1mod = str1mod.replaceAll("-", "");

str2mod = str2mod.replaceAll("-", "");

str1mod = str1mod.replaceAll("\\(", "");

str2mod = str2mod.replaceAll("\\(", "");

str1mod = str1mod.replaceAll("\\)", "");

str2mod = str2mod.replaceAll("\\)", "");

str1mod = str1mod.replaceAll("\\{", "");

str2mod = str2mod.replaceAll("\\{", "");

str1mod = str1mod.replaceAll("\\}", "");

str2mod = str2mod.replaceAll("\\}", "");

str1mod = str1mod.replaceAll("\\[", "");

str2mod = str2mod.replaceAll("\\[", "");

str1mod = str1mod.replaceAll("\\]", "");

str2mod = str2mod.replaceAll("\\]", "");

str1mod = str1mod.replaceAll("\\'", "");

str2mod = str2mod.replaceAll("\\'", "");

str1mod = str1mod.replaceAll("\"", "");

str2mod = str2mod.replaceAll("\"", "");

str1mod = str1mod.replaceAll("\\\*", "");

str2mod = str2mod.replaceAll("\\\*", "");

str1mod = str1mod.replaceAll("\\\_", "");

str2mod = str2mod.replaceAll("\\\_", "");

str1mod = str1mod.replaceAll("\\/", "");

str2mod = str2mod.replaceAll("\\/", "");

//Return the result of the comparison

return str1mod.equalsIgnoreCase(str2mod);

}

}

Note: 1st and 3rd files read the same. The second is the same as the third file however it has a word added to make the test fail.



