**package** newTest;

**public** **class** Main {

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

**int** n = 0;

n = *test2*(18,5);

System.***out***.println("Return value is " + n);

}

**public** **static** **int** test2(**int** x, **int** y)

{

**if**(x < y)

**return** -5;

**else**

**return**(*test2*(x-y, y+5) +6);

}

}

package edu.wmich.cs1120.spring16.LinkedList07.towerc.application;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.EOFException;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.UTFDataFormatException;

import edu.wmich.cs1120.spring16.LinkedList07.towerc.lists.LinkedList;

import edu.wmich.cs1120.spring16.LinkedList07.towerc.lists.IList;

import edu.wmich.cs1120.spring16.LinkedList07.towerc.stacks.Stack;

public class Application implements IApplication {

/\*\*

\* @param - takes a string to be reversed; sent off to stack class

\* @return - returns the reversed string as a String

\*/

@Override

public String reverseString(String s) {

// Create string builder object to use in reversing strings

StringBuilder str = new StringBuilder();

//Create a new stack object for pushing and popping as we reverse the string.

Stack newStack = new Stack();

// For loop to control pushing string values, one character

// at a time, readying the values in stack class in order to

// pop them off in reverse order, thus REVERSING the string.

// See the nifty push statement using charAt()? :-)

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

{

newStack.push(String.valueOf(s.charAt(i)));

}

//For loop to peek at the value, which obtains it, and allows

// us to append it to the string builder object and then

// we call pop on the stack object, which in turn calls remove

// which removes the character from the string we pushed in the

// code above.

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

{

str.append(newStack.peek());

newStack.pop();

}

// Return the reversed string built with string builder

// but first we must use toString on it because it's an object.

return str.toString();

}

@Override

/\*\*

\* @return - Reads data from file into an object

\* of type IList and returns the IList object.

\*/

public IList readInputFile() {

String line; // A number read from the file

boolean endOfFile = false; // EOF flag

IList listValues = new LinkedList();

try{

// Create the binary file input objects.

FileInputStream fstream = new FileInputStream("input.bin");

DataInputStream inputFile = new DataInputStream(fstream);

System.out.println("Reading data from the file:");

// Read the contents of the file.

while (!endOfFile)

{

try

{

line = inputFile.readUTF();

//System.out.println(line); //tester statement

String[] lines = line.split("\n");

for(String tokens: lines){

//System.out.println(tokens); //tester statement

listValues.add(tokens);

}

}

catch (EOFException e)

{

endOfFile = true;

}

catch (UTFDataFormatException e){

System.out.println("Bytes do not represent a "

+ "valid modified UTF-8 encoding.");

}

}

System.out.println("\nDone.");

System.out.println("Data successfully read from input.bin file.");

// Close the file.

inputFile.close();

} // END BRACKT FOR try-catch file open-close

catch (IOException e)

{

System.out.println("Error in file input/output.");

}

return listValues;

}

/\*\*

\*@param - Method takes the IList output, IE. the reversed strings

\*and writes data to binary output file.

\*

\*/

@Override

public void writeOutputFile(IList output) {

FileOutputStream fstream = null;

try {

fstream = new FileOutputStream("output.bin");

} // END TRY

catch (FileNotFoundException e) {

System.out.println("Could not create/find write file.");

}// END CATCH

DataOutputStream outputFile = new DataOutputStream(fstream);

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

try {

outputFile.writeUTF(output.get(i));

} // END TRY

catch (IndexOutOfBoundsException e) {

System.out.println("Write index is not within boundaries");

} // END CATCH

catch (IOException e){

System.out.println("Error in writting output.");

}// END CATCH

}// END CATCH

try {

outputFile.close();

System.out.println("Data successfully written to output.bin file.");

} // END TRY

catch (IOException e) {

System.out.println("Error in closing output.bin file stream.");

} // END CATCH

} // END BRACKET FOR write output file

} // END BRACKET FOR application class

**package** edu.wmich.cs1120.spring16.LinkedList07.towerc.application;

**import** edu.wmich.cs1120.spring16.LinkedList07.towerc.lists.IList;

**public** **interface** IApplication {

/\*\*

\* Reads the binary file "input.bin" and returns each line

\* as an element in an IList

\* **@return** an IList containing the input.

\*/

**public** IList readInputFile();

/\*\*

\* Writes the reversed string to the binary file "output.bin"

\* **@param** output

\*/

**public** **void** writeOutputFile(IList output);

/\*\*

\* Prints out the input and output strings to the screen.

\* **@param** input the input string

\* **@param** output the output string

\*/

**default** **public** **void** printToScreen(String input, String output){

System.***out***.println("The reverse of string \""+input

+"\" is \""+output+"\".");

}

/\*\*

\* Reverses the String parameter.

\* **@param** s the String to be reversed

\* **@return** the reversed string

\*/

**public** String reverseString(String s);

}

**package** edu.wmich.cs1120.spring16.LinkedList07.towerc.lists;

**public** **interface** INode<E> {

/\*\*

\* Returns the data stored in this node.

\* **@return** Data in this node.

\*/

E getData();

/\*\*

\* Setter for data for this node.

\* **@param** data New data

\*/

**void** setData(E data);

/\*\*

\* Returns the node next to this node.

\* **@return** Node next to this node.

\*/

INode<E> getNext();

/\*\*

\* Sets node received as the next node to this node.

\* **@param** next New next node.

\*/

**void** setNext(INode<E> next);

}

**package** edu.wmich.cs1120.spring16.LinkedList07.towerc.lists;

**public** **interface** IList {

/\*\*

\* Adds the element e to the end of the list.

\* **@param** e element to be added

\*/

**void** add(String e);

/\*\*

\* Adds the element e to the list at the specified index.

\* **@param** index of the location to place the string, starting from 0

\* **@param** e element to be added

\*/

**void** add(**int** index, String e) **throws** IndexOutOfBoundsException;

/\*\*

\* Removes all of the elements from the list

\*/

**void** clear();

/\*\*

\* Checks to see if list contains the parameter s

\* **@param** s parameter to search for.

\* **@return** true if found, false otherwise.

\*/

**boolean** contains(String s);

/\*\*

\* **@return** the element at the front (index 0) of the list

\*/

String getHead();

/\*\*

\* **@return** the element at the back (index size-1) of the list.

\*/

String getTail();

/\*\*

\*

\* **@param** index index of the element to retrieve, starting from 0.

\* **@return** the element at that index.

\* **@throws** IndexOutOfBoundsException

\*/

String get(**int** index) **throws** IndexOutOfBoundsException;

/\*\*

\* Searches for the element s in the list and returns the

\* index of the first occurrence, starting from index 0

\* **@param** s parameter to search for

\* **@return** index of the element, or -1 if not found.

\*/

**int** indexOf(String s);

/\*\*

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

\*/

**boolean** isEmpty();

/\*\*

\* Removes the element at the specified index. Close up the gap

\* in the array by moving all elements forward one step.

\* Amend your “size” attribute to reflect the removal of the element.

\* **@param** index of element to be removed, starting from index 0

\* **@return** The contents of the element that was removed.

\* **@throws** IndexOutOfBoundsException

\*/

String remove(**int** index) **throws** IndexOutOfBoundsException;

/\*\*

\*

\* **@return** the number of elements in this list.

\*/

**int** size();

}

**package** edu.wmich.cs1120.spring16.LinkedList07.towerc.lists;

**public** **class** LinkedList **implements** IList {

**private** INode<String> first; // list head

**private** INode<String> last; // last element in list

**private** **int** size = 0; // size field. We use this field in this class.

/\*\*

Constructor.

We call this when we create a Linked List.

First call is for an empty list and both first and last

are null as there is no data yet.

\*/

**public** LinkedList()

{

first = **null**;

last = **null**;

}

/\*\*

\* **@param** - takes a string that will be added to this Linked List object.

\*

\*/

@Override

**public** **void** add(String e) {

/\*

if (size == 1){ // CODE THAT WE ABANDONED as a secondary check

last = new Node<String>(e);

first.setNext(last);

}

\*/

**if** (isEmpty()) // if list is empty, first is first and last and the only item

{

first = **new** Node<String>(e);

last = first;

}

**else** // else in all other cases add like this

{

last.setNext(**new** Node<String>(e)); // next node is the new node

last = last.getNext(); // sets new last to the end

}

size ++; // we increase size here

}

/\*\*

\* **@param** - integer of index value of element to add to Linked List

\* **@param** - String element to add to Linked List

\* **@throws** - Index out of bounds exception if index is outside Linked List size

\*

\*/

@Override

**public** **void** add(**int** index, String e) **throws** IndexOutOfBoundsException {

**if** (index < 0 || index > size()) // check that index value is in the list bounds

{

String message = String.*valueOf*(index);

**throw** **new** IndexOutOfBoundsException(message);

}

size++;

// Index is at least 0

**if** (index == 0)

{

// New element goes at beginning

first = **new** Node<String>(e, first);

**if** (last == **null**)

last = first;

**return**;

}

// Set a reference pred to point to the node that

// will be the predecessor of the new node

INode<String> pred = first;

**for** (**int** k = 1; k <= index - 1; k++)

{

pred = pred.getNext();

}

// Splice in a node containing the new element

pred.setNext(**new** Node(e, pred.getNext()));

// Is there a new last element ?

**if** (pred.getNext().getNext() == **null**)

last = pred.getNext();

}

/\*\*

\* This method clears Linked List by setting it equal to null

\* and resets size to zero.

\*/

@Override

**public** **void** clear() {

//overwrite head to null

// If there's o value for head, then we cannot traverse list

// as a traverse starts with head; now any previous list is garbage

first = **null**;

size = 0;

}

/\*\*

\* **@param** - String parameter to check if it's in the Linked List.

\* **@return** - boolean value - returns true if element is found;

\* returns false if element is not found.

\*

\*/

@Override

**public** **boolean** contains(String s) {

INode<String> pred = first;

//create a traversal node

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

**if**(s.equals(pred.getData())){

**return** **true**;

}

pred.getNext();

}

//loop through the nodes as well as compare the data

//return true if the node data matches the parameterized version

**return** **false**;

}

/\*\*

\***@return** - Returns String of the element at head (IE. first)

\*in the Linked List.

\*/

@Override

**public** String getHead() {

//If Linked List is empty, there is nothing to return, so return null.

**if**(isEmpty())

**return** **null**;

**return** first.getData();

}

/\*\*

\***@return** - Returns String of the element at tail (IE. current last)

\*in the Linked List.

\*/

@Override

**public** String getTail() {

//If Linked List is empty, there is nothing to return, so return null.

**if**(isEmpty())

**return** **null**;

**return** last.getData();

}

/\*\*

\* **@param** - Method takes an integer value as index of element to

\* get in the Linked List.

\* **@return** - Method returns a string of the value found at the index.

\*

\* **@throws** - throw an out of bounds exception if index is outside

\* the boundaries of the Linked List.

\*

\*/

@Override

**public** String get(**int** index) **throws** IndexOutOfBoundsException {

**int** count = 0;

INode<String> current;

**if**(index < 0 || index > size-1){

**throw** **new** IndexOutOfBoundsException();

}

**for**(current = first; count != index; count++ , current = current.getNext());

//System.out.print(current.getData());

**return** current.getData();

}

/\*\*

\***@param** - method takes a string of element to find in the Linked List.

\*

\***@return** - method returns the index of the string element passed in.

\*

\*/

@Override

**public** **int** indexOf(String s) {

INode<String> pred = first;

**int** counter = 0;

//create a traversal node and a counter

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

**if**(s.equals(pred.getData())){

**return** counter;

}

**else**{

counter++;

pred.getNext();

}

}

//loop through the list and compare the string parameterized

//to the string that the current node is pointing to. The value

//of the current node increases as well as the progression through

//the linked list.

**return** -1;

}

/\*\*

\***@return** - method returns a boolean value.

\*If Linked List is empty, then it returns true, ELSE

\* it returns false.

\*/

@Override

**public** **boolean** isEmpty() {

**if**(size == 0)

**return** **true**;

**else**

**return** **false**;

}

/\*\*

\***@throws** - throw an out of bounds exception if index is outside

\* the boundaries of the Linked List.

\***@param** - index of the item to remove from the Linked List.

\***@return** - Method returns the string value of the item removed

\*from the Linked List.

\*

\*/

@Override

**public** String remove(**int** index) **throws** IndexOutOfBoundsException {

**if** (index < 0 || index >= size())

{

String message = String.*valueOf*(index);

**throw** **new** IndexOutOfBoundsException(message);

}

INode<String> element; // The element to return

**if** (index == 0)

{

size --; // first we decrement

// Removal of first item in the list

element = first;

first = first.getNext();

**if** (first == **null**)

last = **null**;

}

**else**

{

// To remove an element other than the first,

// find the predecessor of the element to

// be removed.

size --;

INode<String> pred = first;

// Move pred forward index - 1 times

**for** (**int** k = 1; k <= index -1; k++)

pred = pred.getNext();

// Store the value to return

element = pred.getNext();

//SAMPLE

// Route link around the node to be removed

//pred.next = pred.next.next;

pred.setNext(pred.getNext().getNext());

// Check if pred is now last

**if** (pred.getNext() == **null**)

last = pred;

}

**return** element.toString();

}

/\*\*

\***@return** - method returns an integer representing the

\*size field, IE. the size of the Linked List.

\*/

@Override

**public** **int** size() {

**return** size;

}

}

**package** edu.wmich.cs1120.spring16.LinkedList07.towerc.lists;

**public** **class** Node<E> **implements** INode<E> {

**private** E stringValue;

**private** INode<E> nextNode;

**public** Node(){}

/\*\*

Constructor

one param

This constructor calls the other constructor

We thought this was a fun way to do it.

Also, in creating first, if there is one node, then next is null.

\*/

**public** Node(E v)

{

**this**(v, **null**);

}

/\*\*

Constructor

two params

String value is the data

next node value is the next node in the linked list

\*/

**public** Node(E v, INode<E> n)

{

stringValue = v;

nextNode = n;

}

/\*\*

\* Returns the data stored in this node.

\* **@return** Data in this node.

\*/

@Override

**public** E getData() {

**return** stringValue;

}

/\*\*

\* Setter for data for this node.

\* **@param** data New data

\*/

@Override

**public** **void** setData(E data) {

stringValue = data;

}

/\*\*

\* Returns the node next to this node.

\* **@return** Node next to this node.

\*/

@Override

**public** INode<E> getNext() {

**return** nextNode;

}

/\*\*

\* Sets node received as the next node to this node.

\* **@param** next New next node.

\*/

@Override

**public** **void** setNext(INode<E> next) {

nextNode = next;

}

}

**package** edu.wmich.cs1120.spring16.LinkedList07.towerc.stacks;

**public** **interface** IStack {

/\*\*

\* Adds the parameter s to the top of the stack.

\* **@param** s the string to be added

\*/

**void** push(String s);

/\*\*

\* Removes the top element from the stack

\*/

**void** pop();

/\*\*

\* Returns the top element without removing it.

\* **@return** the top element in the stack

\*/

String peek();

/\*\*

\*

\* **@return** the number of elements in the stack

\*/

**int** size();

/\*\*

\*

\* **@return** true if the stack contains no elements, false otherwise.

\*/

**boolean** isEmpty();

}

**package** edu.wmich.cs1120.spring16.LinkedList07.towerc.stacks;

**public** **interface** IStack {

/\*\*

\* Adds the parameter s to the top of the stack.

\* **@param** s the string to be added

\*/

**void** push(String s);

/\*\*

\* Removes the top element from the stack

\*/

**void** pop();

/\*\*

\* Returns the top element without removing it.

\* **@return** the top element in the stack

\*/

String peek();

/\*\*

\*

\* **@return** the number of elements in the stack

\*/

**int** size();

/\*\*

\*

\* **@return** true if the stack contains no elements, false otherwise.

\*/

**boolean** isEmpty();

}