Dylan T Carlson

1161653

Lab106

10/08/18

**Client:**

/\*\*

\* This client class is a thorough test of each of the stacks,

\* LeakyStack and DoubleStack.

\*

\* @author Dylan Carlson

\*/

public class Client {

/\*\*

\* The main method simply calls two methods that performs

\* multiple tests of the two stacks.

\* @param args

\*/

public static void main(String[] args){

redAndBlueTest();

leakyTest();

}

/\*\*

\* redAndBlueTest tests the DoubleStackArray

\*/

public static void redAndBlueTest(){

DoubleStack<Integer> N = new DoubleStackArray<>();

System.out.println("===== Testing DoubleStackArray =====");

System.out.println("\nIs the blue stack empty? "+ N.blueIsEmpty());

System.out.println("Is the red stack empty? "+ N.redIsEmpty());

System.out.println("\nBlue size: "+ N.blueSize());

System.out.println("Red size: "+ N.redSize());

System.out.println("\nTop of Blue and Red: ");

System.out.println("Top of blue: "+ N.blueTop());

System.out.println("Top of red: "+ N.redTop());

System.out.println("\n----Pushing 6 onto blue----");

System.out.println("Pushing 1 on blue: ");

N.bluePush(1);

System.out.println("Pushing 2 on blue: ");

N.bluePush(2);

System.out.println("Pushing 3 on blue: ");

N.bluePush(3);

System.out.println("Pushing 4 on blue: ");

N.bluePush(4);

System.out.println("Pushing 5 on blue: ");

N.bluePush(5);

System.out.println("Pushing 6 on blue: ");

N.bluePush(6);

System.out.println("\n----Pushing 4 onto red----");

System.out.println("Pushing 1 on red: ");

N.redPush(1);

System.out.println("Pushing 2 on red: ");

N.redPush(2);

System.out.println("Pushing 3 on red: ");

N.redPush(3);

System.out.println("Pushing 4 on red: ");

N.redPush(4);

System.out.println("\n\nTesting toString:");

System.out.println(N.toString());

System.out.println("\nIs the blue stack empty? "+ N.blueIsEmpty());

System.out.println("Is the red stack empty? "+ N.redIsEmpty());

System.out.println("\nBlue size: "+ N.blueSize());

System.out.println("Red size: "+ N.redSize());

System.out.println("\nTop of Blue and Red: ");

System.out.println("Top of blue: "+ N.blueTop());

System.out.println("Top of red: "+ N.redTop());

System.out.println("\nPopping off each: ");

System.out.println("\nPopping all off of blue: ");

System.out.println(N.bluePop());

System.out.println(N.bluePop());

System.out.println(N.bluePop());

System.out.println(N.bluePop());

System.out.println(N.bluePop());

System.out.println(N.bluePop());

System.out.println("\nPopping all off of red: ");

System.out.println(N.redPop());

System.out.println(N.redPop());

System.out.println(N.redPop());

System.out.println(N.redPop());

System.out.println("\nBlue size: "+ N.blueSize());

System.out.println("Red size: "+ N.redSize());

System.out.println("\nIs the blue stack empty? "+ N.blueIsEmpty());

System.out.println("Is the red stack empty? "+ N.redIsEmpty());

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

System.out.println("Top of blue: "+ N.blueTop());

System.out.println("Top of red: "+ N.redTop());

}

/\*\*

\* leakyTest tests the LeakyStackArray

\*/

public static void leakyTest() {

LeakyStack<Integer> L = new LeakyStackArray<>();

System.out.println("\n\n===== Testing LeakyStack =====");

System.out.println("\nIs the stack empty? "+ L.isEmpty() );

System.out.println("\nElement on top: " + L.top() );

System.out.println("\nSize of stack: " + L.size() );

System.out.println("\nPushing 11 elements into the stack: ");

System.out.println("pushing 1:");

L.push(1);

System.out.println("pushing 2:");

L.push(2);

System.out.println("pushing 3:");

L.push(3);

System.out.println("pushing 4:");

L.push(4);

System.out.println("pushing 5:");

L.push(5);

System.out.println("pushing 6:");

L.push(6);

System.out.println("pushing 7:");

L.push(7);

System.out.println("pushing 8:");

L.push(8);

System.out.println("pushing 9:");

L.push(9);

System.out.println("pushing 10:");

L.push(10);

System.out.println("pushing 11:");

L.push(11);

System.out.println("\nIs the stack empty? "+ L.isEmpty() );

System.out.println("\nElement on top: " + L.top() );

System.out.println("\nSize of stack: " + L.size() );

System.out.println("\nTesting toString: \n"+ L.toString() );

System.out.println("\nPopping all off");

System.out.println(L.pop());

System.out.println(L.pop());

System.out.println(L.pop());

System.out.println(L.pop());

System.out.println(L.pop());

System.out.println(L.pop());

System.out.println(L.pop());

System.out.println(L.pop());

System.out.println(L.pop());

System.out.println(L.pop());

System.out.println("\nIs the stack empty? "+ L.isEmpty() );

System.out.println("\nElement on top: " + L.top() );

System.out.println("\nSize of stack: " + L.size() );

}

}

**DoubleStack ADT:**

/\*\*

\* DoubleStack is a generic interface that contains the methods for

\* a blue stack and a red stack.

\*

\* @author Dylan Carlson

\*/

public interface DoubleStack<E> {

/\*\* Blue Stack \*\*/

/\*\*

\* blueSize returns the size of the blue stack to the user.

\* @return

\*/

int blueSize();

/\*\*

\* blueIsEmpty returns true if the blue stack is empty and false otherwise.

\* @return

\*/

boolean blueIsEmpty();

/\*\*

\* bluePush is passed an object and pushes it onto the top

\* of the blue stack.

\* @param e

\*/

void bluePush(E e);

/\*\*

\* blueTop returns the element on the top of the blue stack.

\* @return

\*/

E blueTop();

/\*\*

\* bluePop removes an element from the top of the blue stack

\* and returns the element removed.

\* @return

\*/

E bluePop();

/\*\* Red Stack \*\*/

/\*\*

\* redSize returns the size of the red stack to the user.

\* @return

\*/

int redSize();

/\*\*

\* redIsEmpty returns true if the red stack is empty and false otherwise.

\* @return

\*/

boolean redIsEmpty();

/\*\*

\* redPush is passed an object and pushes it onto the top

\* of the red stack.

\* @param e

\*/

void redPush(E e);

/\*\*

\* redTop returns the element on the top of the red stack.

\* @return

\*/

E redTop();

/\*\*

\* redPop removes an element from the top of the red stack

\* and returns the element removed.

\* @return

\*/

E redPop();

/\*\*

\* toString returns all the elements in each stack, the class name,

\* and size.

\* @return

\*/

String toString();

/\*\*

\* equals is passed an object and returns whether it is equal to

\* the doublestack.

\* @param o

\* @return

\*/

boolean equals(Object o);

}

**DoubleStackArray (implementation):**

/\*\*

\* DoubleStackArray is a generic implementation of the DoubleStack

\* interface. It uses a single array to hold both stacks. It adds blue

\* elements from the left, and red elements from the right.

\*

\* @author Dylan Carlson

\*/

public class DoubleStackArray<E> implements DoubleStack<E>{

private static final int CAPACITY = 10;

private E[] stack;

private int blueSize;

private int redSize;

private int last;

/\*\*

\* Default constructor for DoubleStackArray.

\* Has a default capacity of 10.

\*/

public DoubleStackArray() {

this(CAPACITY);

}

/\*\*

\* Overload constructor for DoubleStackArray

\* @param capacity

\*/

public DoubleStackArray( int capacity ){

stack = (E[]) new Object[capacity];

blueSize = 0;

redSize = 0;

last = capacity-1;

}

/\*\* Blue Methods \*\*/

/\*\*

\* blueSize returns the size of the blue stack to the user.

\* @return

\*/

public int blueSize(){

return blueSize;

}

/\*\*

\* blueIsEmpty returns true if the blue stack is empty and false otherwise.

\* @return

\*/

public boolean blueIsEmpty(){

return blueSize == 0;

}

/\*\*

\* bluePush is passed an object and pushes it onto the top

\* of the blue stack.

\* @param e

\*/

public void bluePush(E e){

if( (blueSize + redSize) == stack.length )

throw new IllegalStateException("Stack is full");

else{

stack[blueSize++] = e;

}

}

/\*\*

\* bluePop removes an element from the top of the blue stack

\* and returns the element removed.

\* @return

\*/

public E bluePop(){

if( blueIsEmpty() ){

return null;

}

E answer = stack[--blueSize];

stack[blueSize] = null;

return answer;

}

/\*\*

\* blueTop returns the element on the top of the blue stack.

\* @return

\*/

public E blueTop(){

if( blueIsEmpty() ){

return null;

}

E answer = stack[blueSize - 1];

return answer;

}

/\*\* Red Methods \*\*/

/\*\*

\* redSize returns the size of the red stack to the user.

\* @return

\*/

public int redSize(){

return redSize;

}

/\*\*

\* redIsEmpty returns true if the red stack is empty and false otherwise.

\* @return

\*/

public boolean redIsEmpty(){

return redSize == 0;

}

/\*\*

\* redPush is passed an object and pushes it onto the top

\* of the red stack.

\* @param e

\*/

public void redPush(E e){

if( (blueSize + redSize) == stack.length )

throw new IllegalStateException("Stack is full");

else{

redSize++;

stack[last--] = e;

}

}

/\*\*

\* redPop removes an element from the top of the red stack

\* and returns the element removed.

\* @return

\*/

public E redPop(){

if( redIsEmpty() ){

return null;

}

E answer = stack[++last];

stack[last] = null;

redSize--;

return answer;

}

/\*\*

\* redTop returns the element on the top of the red stack.

\* @return

\*/

public E redTop(){

if( redIsEmpty() ){

return null;

}

E answer = null;

if(stack.length == last+1)

answer = stack[last];

else{

answer = stack[last + 1];

}

return answer;

}

/\*\*

\* toString returns all the elements in each stack, the class name,

\* and size, and the index of the last element in the array.

\* @return

\*/

public String toString(){

String blueElements = "";

String redElements = "";

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

blueElements = blueElements + stack[i];

}

for(int i = (last+1); i < (blueSize + redSize); i++){

redElements = redElements + stack[i];

}

return getClass().getName() + " Blue elements: " + blueElements + "--Blue Size = " + blueSize+ " Red Elements: "+ redElements + "--Red Size = "+ redSize + " Last index: " + last;

}

/\*\*

\* equals is passed an object and returns whether it is equal to

\* the doublestack.

\* @param o

\* @return

\*/

public boolean equals( Object o ){

if(!(o instanceof DoubleStackArray))

return false;

DoubleStackArray e = (DoubleStackArray) o;

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

if ( !(stack[i] == e.stack[i]) );

return false;

}

for(int i = last; i > redSize; i--){

if ( !(stack[i] == e.stack[i]) );

return false;

}

return blueSize == e.blueSize

&& redSize == e.redSize;

}

}

**LeakyStack ADT:**

/\*\*

\* LeakyStack is a generic interface that contains the methods

\* for a stack that has a unique property. Once it is full and

\* another element is pushed onto it, it drops or "leaks" out

\* the last element.

\*

\* @author Dylan Carlson

\*/

public interface LeakyStack<E> {

/\*\*

\* size returns the amount of elements in the stack.

\* @return

\*/

int size();

/\*\*

\* isEmpty returns true if the stack is empty, and false otherwise.

\* @return

\*/

boolean isEmpty();

/\*\*

\* push is passed an element and pushes it on top of the stack.

\* If full, it drops the last element in the stack, or the first added.

\* @param e

\*/

void push(E e);

/\*\*

\* pop removes the top element from the stack and returns it.

\* @return

\*/

E pop();

/\*\*

\* top returns the element at the top of the stack.

\* @return

\*/

E top();

/\*\*

\* toString returns all the elements in the stack, the size,

\* and the default capacity.

\* @return

\*/

String toString();

/\*\*

\* equals is passed an Object and returns true if it is equal to

\* the leaky stack, and false otherwise.

\* @param o

\* @return

\*/

boolean equals(Object o);

}

**LeakyStackArray (implementation):**

/\*\*

\* LeakyStackArray is a static implementation of the LeakyStrack

\* interface. It holds elements in an array and when it is full it

\* "leaks" or drops the last element.

\*

\* @author Dylan Carlson

\*/

public class LeakyStackArray<E> implements LeakyStack<E> {

public static final int CAPACITY = 10;

private E[] stack;

private int size;

private int t = 0; //top of stack

/\*\*

\* Default constructor.

\* Sets default capacity to 10.

\*/

public LeakyStackArray() { this(CAPACITY); }

/\*\*

\* Overload constructor.

\* @param capacity

\*/

public LeakyStackArray( int capacity){

stack = (E[]) new Object[capacity];

size = 0;

}

/\*\* Methods \*\*/

/\*\*

\* size returns the amount of elements in the stack.

\* @return

\*/

public int size(){

return size;

}

/\*\*

\* isEmpty returns true if the stack is empty, and false otherwise.

\* @return

\*/

public boolean isEmpty(){

return size == 0;

}

/\*\*

\* push is passed an element and pushes it on top of the stack.

\* If full, it drops the last element in the stack, or the first added.

\* @param e

\*/

public void push(E e){

stack[ (t%stack.length) ] = e;

t++;

if(size != stack.length)

size++;

}

/\*\*

\* pop removes the top element from the stack and returns it.

\* @return

\*/

public E pop(){

if( isEmpty() ){

return null;

}

E answer = stack[ (t-1)%stack.length ];

t--;

stack[ t%stack.length ] = null;

size--;

return answer;

}

/\*\*

\* top returns the element at the top of the stack.

\* @return

\*/

public E top(){

if(size == 0)

return null;

E answer = stack[ (t-1)%stack.length ];

return answer;

}

/\*\*

\* toString returns all the elements in the stack, the size,

\* and the default capacity.

\* @return

\*/

public String toString(){

String elements = " ";

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

elements = elements + stack[i] + " ";

}

return getClass().getName() + " Default Capacity: " + CAPACITY + " Size of stack: "+ size + " Elements: "+ elements;

}

/\*\*

\* equals is passed an Object and returns true if it is equal to

\* the leaky stack, and false otherwise.

\* @param o

\* @return

\*/

public boolean equals(Object o) {

if( !(o instanceof LeakyStackArray) )

return false;

LeakyStackArray L = (LeakyStackArray) o;

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

if( !(stack[i] == L.stack[i]) ){

return false;

}

}

return size == L.size;

}

}

**Output:**

run:

===== Testing DoubleStackArray =====

Is the blue stack empty? true

Is the red stack empty? true

Blue size: 0

Red size: 0

Top of Blue and Red:

Top of blue: null

Top of red: null

----Pushing 6 onto blue----

Pushing 1 on blue:

Pushing 2 on blue:

Pushing 3 on blue:

Pushing 4 on blue:

Pushing 5 on blue:

Pushing 6 on blue:

----Pushing 4 onto red----

Pushing 1 on red:

Pushing 2 on red:

Pushing 3 on red:

Pushing 4 on red:

Testing toString:

DoubleStackArray Blue elements: 123456--Blue Size = 6 Red Elements: 4321--Red Size = 4 Last index: 5

Is the blue stack empty? false

Is the red stack empty? false

Blue size: 6

Red size: 4

Top of Blue and Red:

Top of blue: 6

Top of red: 4

Popping off each:

Popping all off of blue:

6

5

4

3

2

1

Popping all off of red:

4

3

2

1

Blue size: 0

Red size: 0

Is the blue stack empty? true

Is the red stack empty? true

Top of blue: null

Top of red: null

===== Testing LeakyStack =====

Is the stack empty? true

Element on top: null

Size of stack: 0

Pushing 11 elements into the stack:

pushing 1:

pushing 2:

pushing 3:

pushing 4:

pushing 5:

pushing 6:

pushing 7:

pushing 8:

pushing 9:

pushing 10:

pushing 11:

Is the stack empty? false

Element on top: 11

Size of stack: 10

Testing toString:

LeakyStackArray Default Capacity: 10 Size of stack: 10 Elements: 11 2 3 4 5 6 7 8 9 10

Popping all off

11

10

9

8

7

6

5

4

3

2

Is the stack empty? true

Element on top: null

Size of stack: 0

BUILD SUCCESSFUL (total time: 0 seconds)