Dylan Carlson

1161653

Lab 105

10/02/18

**Client Class:**

/\*\*

\* This Client class tests all of the stack and queue

\* data structures, and ensures their correctness. Then it tests

\* the time it takes to add elements to the ArrayList, specifically

\* when the array has to double.

\*

\* @author dylca

\*/

public class Client {

public static void main (String[] args) {

arrayStackTest();

linkedStackTest();

arrayQueueTest();

linkedQueueTest();

arrayListTest();

}

/\*\*

\* This method Tests the ArrayStack

\*/

public static void arrayStackTest() {

Stack<Integer> S = new ArrayStack<>(4);

System.out.println("Testing the ArrayStack: ");

System.out.println("Pushing 1 onto stack");

S.push(1);

System.out.println("Pushing 2 onto stack");

S.push(2);

System.out.println("Pushing 3 onto stack");

S.push(3);

System.out.println("Pushing 4 onto stack");

S.push(4);

System.out.println( "Size = " + S.size() );

//Pushing onto a full stack

System.out.println("Trying to push onto a full stack: ");

try{

S.push(5);

}

catch(IllegalStateException ise){

System.out.println("Stack is full");

}

System.out.println("Popping off the stack: ");

//Popping all off

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

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

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

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

System.out.println("Is the stack empty?");

System.out.println( S.isEmpty() );

//Popping an empty Stack

System.out.println("Popping off an empty stack: ");

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

}

/\*\*

\* This method Tests the LinkedStack

\*/

public static void linkedStackTest() {

Stack<Integer> L = new LinkedStack<>();

System.out.println("\n\nTesting the Linked Stack: ");

System.out.println("Pushing 1 onto stack");

L.push(1);

System.out.println("Pushing 2 onto stack");

L.push(2);

System.out.println("Pushing 3 onto stack");

L.push(3);

System.out.println("Pushing 4 onto stack");

L.push(4);

System.out.println( "Size = " + L.size() );

System.out.println("Popping off the stack: ");

//Popping 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("Is the stack empty?");

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

//Popping an empty Stack

System.out.println("Popping off an empty stack: ");

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

}

/\*\*

\* This method Tests the ArrayQueue

\*/

public static void arrayQueueTest() {

Queue<Integer> Q = new ArrayQueue<>(4);

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

System.out.println("Entering 1 in queue");

Q.enqueue(1);

System.out.println("Entering 2 in queue");

Q.enqueue(2);

System.out.println("Entering 3 in queue");

Q.enqueue(3);

System.out.println("Entering 4 in queue");

Q.enqueue(4);

System.out.println( "Size = " + Q.size() );

//Pushing into a full queue

System.out.println("Trying to push into a full queue: ");

try{

Q.enqueue(5);

}

catch(IllegalStateException ise){

System.out.println("Queue is full");

}

System.out.println("Deueueing all elements: ");

//Dequeueing all out

System.out.println( Q.dequeue() );

System.out.println( Q.dequeue() );

System.out.println( Q.dequeue() );

System.out.println( Q.dequeue() );

System.out.println("Is the queue empty?");

System.out.println( Q.isEmpty() );

//Popping an empty Stack

System.out.println("Dequeueing an empty queue: ");

System.out.println( Q.dequeue() );

}

/\*\*

\* This method Tests the LinkedQueue

\*/

public static void linkedQueueTest() {

Queue<Integer> Q = new LinkedQueue<>();

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

System.out.println("Entering 1 in queue");

Q.enqueue(1);

System.out.println("Entering 2 in queue");

Q.enqueue(2);

System.out.println("Entering 3 in queue");

Q.enqueue(3);

System.out.println("Entering 4 in queue");

Q.enqueue(4);

System.out.println( "Size = " + Q.size() );

System.out.println("Deueueing all elements: ");

//Dequeueing all out

System.out.println( Q.dequeue() );

System.out.println( Q.dequeue() );

System.out.println( Q.dequeue() );

System.out.println( Q.dequeue() );

System.out.println("Is the queue empty?");

System.out.println( Q.isEmpty() );

//Popping an empty Stack

System.out.println("Dequeueing an empty queue: ");

System.out.println( Q.dequeue() );

}

/\*\*

\* This method Tests the ArrayList and catches if

\* there is no more available memory.

\*/

public static void arrayListTest() {

try{

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

int k = 67108864;

long startTime;

long endTime;

long elasped;

System.out.println("==== Test adding "+ k +" items to the list ====");

List<Boolean> L1 = new ArrayList<>();

for (int i = 0; i < k+3; i++){

if(i>=k-2){

System.out.println("Size = "+ L1.size());

startTime = System.nanoTime();

L1.add(i, true);

endTime = System.nanoTime();

elasped = endTime - startTime;

System.out.print(" Time = "+ elasped +" nanoseconds" );

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

}

else

L1.add(i, true);

}

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

k \*= 2;

System.out.println("==== Test adding "+ k +" items to the list ====");

List<Boolean> L2 = new ArrayList<>();

for (int i = 0; i < k+3; i++){

if(i>=k-2){

System.out.println("Size = "+ L2.size());

startTime = System.nanoTime();

L2.add(i, true);

endTime = System.nanoTime();

elasped = endTime - startTime;

System.out.print(" Time = "+ elasped +" nanoseconds" );

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

}

else

L2.add(i, true);

}

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

k \*= 2;

System.out.println("==== Test adding "+ k +" items to the list ====");

List<Boolean> L3 = new ArrayList<>();

for (int i = 0; i < k+2; i++){

if(i>=k-2){

System.out.println("Size = "+ L3.size());

startTime = System.nanoTime();

L3.add(i, true);

endTime = System.nanoTime();

elasped = endTime - startTime;

System.out.print(" Time = "+ elasped +" nanoseconds" );

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

}

else

L3.add(i, true);

}

}

catch(OutOfMemoryError oome){

System.out.println("Out Of memory");

}

}

}

**Stack Interface:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 6.1

\*

\* An implementation of the generic Stack interface

\*/

/\*\*

\* 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.until.Stack.

\*

\* @author Michael T. Goodrick

\* @author Roberto Tamassia

\* @author Michael H. Goldwasser

\*/

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();

}

**ArrayStack Class:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 6.2

\*

\* An implementation of the ArrayStack class

\*/

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

public static final int CAPACITY = 1000; //default array capacity

private E[] data; //generic array used for storage

private int t = -1; //index of the top element in stack

public ArrayStack() { this(CAPACITY); } //constructs stack with default capacity

public ArrayStack(int capacity) { //constructs stack with given capcity

data = (E[]) new Object[capacity]; //safe case; complier may give warning

}

public int size() { return (t + 1); }

public boolean isEmpty() { return (t == -1); }

public void push(E e) throws IllegalStateException {

if(size() == data.length) throw new IllegalStateException("Stack is full");

data[++t] = e; //increment t before storing new item

}

public E top() {

if(isEmpty()) return null;

return data[t];

}

public E pop() {

if(isEmpty()) return null;

E answer = data[t];

data[t] = null; //dereference to help garabage collection

t--;

return answer;

}

}

**SinglyLinkedList Class:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 3.14 and 3.15

\*

\* An implementation of a SinglyLinkedList class

\*/

public class SinglyLinkedList<E> {

//------Nested Node class------

private static class Node<E>{

private E element;

private Node<E> next;

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; }

}

//------End of Nested Node class------

//Instance variables of SinglyLinkedList

private Node<E> head = null;

private Node<E> tail = null;

private int size = 0;

public SinglyLinkedList(){}

//access methods

public int size(){ return size; }

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

public E first(){

if( isEmpty() ) return null;

return head.getElement();

}

public E last(){

if( isEmpty() ) return null;

return tail.getElement();

}

//Update methods

public void addFirst(E e){

head = new Node<>(e, head);

if( size == 0)

tail = head;

size++;

}

public void addLast(E e){

Node<E> newest = new Node<> (e, null);

if ( isEmpty() )

head = newest;

else

tail.setNext(newest);

tail = newest;

size++;

}

public E removeFirst(){

if ( isEmpty() ) return null;

E answer = head.getElement();

head = head.getNext();

size--;

if( size == 0 )

tail = null;

return answer;

}

}

**LinkedStack Class:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 6.4

\*

\* An implementation of the LinkedStack class

\*/

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 Interface:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 6.9

\*

\* An implementation of the Queue interface

\*/

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();

}

**ArrayQueue:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 6.10

\*

\* An implementation of the ArrayQueue class

\*/

/\*\* Implementation of the Queue ADT using a fixed-length array. \*/

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

//instance variables

public static final int CAPACITY = 1000; //default array capacity //Added by Me.

private E[] data; // generic array used for storage

private int f = 0; // index of the front element

private int sz = 0; // current number of elements

//constructors

public ArrayQueue() { this(CAPACITY); } // Constructs queue with default capacity

public ArrayQueue( int capacity ) { // Constructs queue with given capacity

data = (E[]) new Object[capacity]; // safe cast; compiler may give warning

}

//methods

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

public int size() { return sz; }

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

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

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

public void enqueue(E e) throws IllegalStateException{

if ( sz == data.length ) throw new IllegalStateException("Queue if full");

int avail = (f + sz) % data.length; // use modular arithmetic

data[avail] = e;

sz++;

}

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

public E first(){

if(isEmpty()) return null;

return data[f];

}

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

public E dequeue(){

if(isEmpty()) return null;

E answer = data[f];

data[f] = null; //dereference to help garbage collection

f = (f + 1) % data.length;

sz--;

return answer;

}

}

**LinkedQueue:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 6.11

\*

\* An implementation of the LinkedQueue class

\*/

/\*\* 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 iitially 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(); }

}

**List Interface:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 7.1

\*

\* An implementation of the List interface.

\*/

/\*\* A simplified version of the java.until.List interface. \*/

public interface List<E> {

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

int size();

/\*\* Returns whether the list is empty. \*/

boolean isEmpty();

/\*\* Returns (but does not remove) the element at index i. \*/

E get(int i) throws IndexOutOfBoundsException;

/\*\* Replaces the element at index i with e, and returns the replaced element. \*/

E set(int i, E e) throws IndexOutOfBoundsException;

/\*\* Inserts element e to be at index i, shifting all subsequent elements later. \*/

void add(int i, E e) throws IndexOutOfBoundsException;

/\*\* Removes/ returns the element at index i, shifting subsequent elements earlier. \*/

E remove(int i) throws IndexOutOfBoundsException;

}

**ArrayList:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragments 7.2, 7.3, 7.4, 7.5

\*

\* An implementation of the ArrayList class.

\*/

public class ArrayList<E> implements List<E> {

// instance variables

public static final int CAPACITY = 16; // default array capacity

private E[] data; //generic array used for storage

private int size = 0; // current number of elements

// constructors

public ArrayList() { this(CAPACITY); } //constructs list with default capacity

public ArrayList(int capacity) { //constructs list with given capacity

data = (E[]) new Object[capacity]; //safe cast; complier may give warning

}

//public methods

/\*\* Returns the number of elements in the array list. \*/

public int size() { return size; }

/\*\* Returns whether the array list is empty. \*/

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

/\*\* Returns (but does not remove) the element at index i. \*/

public E get(int i) throws IndexOutOfBoundsException {

checkIndex(i, size);

return data[i];

}

/\*\* Replaces the element at index i with e, and returns the replaced element. \*/

public E set(int i, E e) throws IndexOutOfBoundsException {

checkIndex(i, size);

E temp = data[i];

data[i] = e;

return temp;

}

/\*\* Inserts element e to be at index i, shifting all subsequent elements later. \*/

public void add(int i, E e) throws IndexOutOfBoundsException {

checkIndex(i, size + 1);

if(size == data.length) // not enough capacity

resize(2 \* data.length); // so double the current capacity

for (int k = size - 1; k >= i; k--) // start by shifting rightmost

data[k+1] = data[k];

data[i] = e; // ready to place the new element

size++;

}

/\*\* Removes/returns the element at index i, shifting subsequent elements earlier. \*/

public E remove(int i) throws IndexOutOfBoundsException {

checkIndex(i, size);

E temp = data[i];

for(int k = i; k < size -1; k++) // shift elements to fill hole

data[k] = data[k+1];

data[size-1] = null; // help garbage collection

size--;

return temp;

}

// utility method

/\*\* Checks whether the given index is in the range[0, n-1]. \*/

protected void checkIndex( int i, int n) throws IndexOutOfBoundsException {

if (i < 0 || i >= n)

throw new IndexOutOfBoundsException("Illegal index: "+ i);

}

//Code fragment 7.4

/\*\* Resizes internal array to have given capacity >= size. \*/

protected void resize(int capacity){

E[] temp = (E[]) new Object[capacity]; // safe cast; compiler may give warning

for(int k = 0; k < size; k++)

temp[k] = data[k];

data = temp; // start using the array

}

}

**Output of Program:**

run:

Testing the ArrayStack:

Pushing 1 onto stack

Pushing 2 onto stack

Pushing 3 onto stack

Pushing 4 onto stack

Size = 4

Trying to push onto a full stack:

Stack is full

Popping off the stack:

4

3

2

1

Is the stack empty?

true

Popping off an empty stack:

null

Testing the Linked Stack:

Pushing 1 onto stack

Pushing 2 onto stack

Pushing 3 onto stack

Pushing 4 onto stack

Size = 4

Popping off the stack:

4

3

2

1

Is the stack empty?

true

Popping off an empty stack:

null

Testing the ArrayQueue:

Entering 1 in queue

Entering 2 in queue

Entering 3 in queue

Entering 4 in queue

Size = 4

Trying to push into a full queue:

Queue is full

Deueueing all elements:

1

2

3

4

Is the queue empty?

true

Dequeueing an empty queue:

null

Testing the LinkedQueue:

Entering 1 in queue

Entering 2 in queue

Entering 3 in queue

Entering 4 in queue

Size = 4

Deueueing all elements:

1

2

3

4

Is the queue empty?

true

Dequeueing an empty queue:

null

==== Test adding 67108864 items to the list ====

Size = 67108862

Time = 1459 nanoseconds

Size = 67108863

Time = 365 nanoseconds

Size = 67108864

Time = 199148197 nanoseconds

Size = 67108865

Time = 19327 nanoseconds

Size = 67108866

Time = 364 nanoseconds

==== Test adding 134217728 items to the list ====

Size = 134217726

Time = 1458 nanoseconds

Size = 134217727

Time = 365 nanoseconds

Size = 134217728

Time = 402035020 nanoseconds

Size = 134217729

Time = 729 nanoseconds

Size = 134217730

Time = 365 nanoseconds

==== Test adding 268435456 items to the list ====

Size = 268435454

Time = 19693 nanoseconds

Size = 268435455

Time = 0 nanoseconds

Size = 268435456

Out Of memory

BUILD SUCCESSFUL (total time: 24 seconds)