/\*\*

\* Test data structures

\* @author Zhenhua.Yang.1

\*/

public class Client {

public static void main(String[] args) {

// create Arrays to store titles and headers for the table

String[] titles = new String[4];

String[] header1 = {"N", "push(nsec)", "pop(nsec)"};

String[] header2 = {"N", "enqueue(nsec)", "dequeue(nsec)"};

titles[0]="ArrayStack Test";

titles[1]="LinkedStack Test";

titles[2]="ArrayQueue Test";

titles[3]="LinkedQueue Test";

long[][] data = new long[7][3]; // 2-D array that save calculated data.

int n = 10;

/\*\* Test ArrayStack \*/

ArrayStack <Integer> stackArr = new ArrayStack<>(); // create new ArrayStack object

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

long startTime = System.nanoTime();

repeatPush\_as(stackArr, 1, n); // repeatly push data to the ArrayStack for n times.

long endTime = System.nanoTime();

long elapsed = endTime - startTime; // calculate the elapsed time.

long startTime2 = System.nanoTime();

repeatPop\_as(stackArr, n); // repeatly pop data from the ArrayStack for n times.

long endTime2 = System.nanoTime();

long elapsed2 = endTime2 - startTime2; // calculate the elapsed time.

data[i][0] = n; // add n to the row i of the data array

data[i][1] = elapsed; // add elapsed time of push() to the row i of the data array

data[i][2] = elapsed2; // add elapsed time of pop() to the row i of the data array

n = n \* 10; // multiplied n by 10;

}

printASCII(titles[0], header1, data);

/\*\* Test LinkedStack \*/

LinkedStack <Integer> stackLink = new LinkedStack<>();

n = 10;

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

long startTime = System.nanoTime();

repeatPush\_ls(stackLink, 1, n); // repeatly push data to the LinkedStack for n times.

long endTime = System.nanoTime();

long elapsed = endTime - startTime; // calculate the elapsed time.

long startTime2 = System.nanoTime();

repeatPop\_ls(stackLink, n); // repeatly pop data from the LinkedStack for n times.

long endTime2 = System.nanoTime();

long elapsed2 = endTime2 - startTime2; // calculate the elapsed time.

data[i][0] = n; // add n to the row i of the data array

data[i][1] = elapsed; // add elapsed time of push() to the row i of the data array

data[i][2] = elapsed2; // add elapsed time of pop() to the row i of the data array

n = n \* 10; // multiplied n by 10;

}

printASCII(titles[1], header1, data);

/\*\* Test ArrayQueue \*/

ArrayQueue <Integer> queueArr = new ArrayQueue<>();

n = 10;

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

long startTime = System.nanoTime();

repeatEnqueue\_aq(queueArr, 1, n); // repeatly enqueue data to the ArrayQueue for n times.

long endTime = System.nanoTime();

long elapsed = endTime - startTime; // calculate the elapsed time.

long startTime2 = System.nanoTime();

repeatDequeue\_aq(queueArr, n); // repeatly dequeue data from the ArrayQueue for n times.

long endTime2 = System.nanoTime();

long elapsed2 = endTime2 - startTime2; // calculate the elapsed time.

data[i][0] = n; // add n to the row i of the data array

data[i][1] = elapsed; // add elapsed time of enqueue() to the row i of the data array

data[i][2] = elapsed2; // add elapsed time of dequeue() to the row i of the data array

n = n \* 10; // multiplied n by 10;

}

printASCII(titles[2], header2, data);

/\*\* Test LinkedQueue \*/

LinkedQueue <Integer> queueLink = new LinkedQueue<>();

n = 10;

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

long startTime = System.nanoTime();

repeatEnqueue\_lq(queueLink, 1, n); // repeatly push data to the LinkedQueue for n times.

long endTime = System.nanoTime();

long elapsed = endTime - startTime; // calculate the elapsed time.

long startTime2 = System.nanoTime();

repeatDequeue\_lq(queueLink, n); // repeatly dequeue data from the LinkedQueue for n times.

long endTime2 = System.nanoTime();

long elapsed2 = endTime2 - startTime2; // calculate the elapsed time.

data[i][0] = n; // add n to the row i of the data array

data[i][1] = elapsed; // add elapsed time of enqueue() to the row i of the data array

data[i][2] = elapsed2; // add elapsed time of dequeue() to the row i of the data array

n = n \* 10; // multiplied n by 10;

}

printASCII(titles[3], header2, data);

/\*\* Test ArrayList \*/

String listFormat = "| %12s | %12s | %12s |\n";

System.out.println("+--------------------------------------------------+\n"

+ "| ArrayList Test |\n"

+ "+----------------+----------------+----------------+\n"

+ "| Method | Return Value | List Contents |\n"

+ "+----------------+----------------+----------------+");

ArrayList<Character> listArr = new ArrayList<>();

listArr.add(0, 'A');

System.out.printf(listFormat, "add(0, A)", "-", listArr.toString());

listArr.add(0, 'B');

System.out.printf(listFormat, "add(0, B)", "-", listArr.toString());

System.out.printf(listFormat, "get(1)", listArr.get(1), listArr.toString());

try{

listArr.set(2, 'C');

System.out.printf(listFormat, "set(2, C)", "-", listArr.toString());

}catch(IndexOutOfBoundsException ex){

System.out.printf(listFormat, "set(2, C)", "\"error\"", listArr.toString());

}

listArr.add(2, 'C');

System.out.printf(listFormat, "add(2, C)", "-", listArr.toString());

try{

listArr.add(4, 'D');

System.out.printf(listFormat, "add(4, D)", "-", listArr.toString());

}catch(IndexOutOfBoundsException ex){

System.out.printf(listFormat, "add(4, D)", "\"error\"", listArr.toString());

}

System.out.printf(listFormat, "reomve(1)", listArr.remove(1), listArr.toString());

listArr.add(1, 'D');

System.out.printf(listFormat, "add(1, D)", "-", listArr.toString());

listArr.add(1, 'E');

System.out.printf(listFormat, "add(1, E)", "-", listArr.toString());

try{

listArr.get(4);

System.out.printf(listFormat, "get(4)", "-", listArr.toString());

}catch(IndexOutOfBoundsException ex){

System.out.printf(listFormat, "get(4)", "\"error\"", listArr.toString());

}

listArr.add(4, 'F');

System.out.printf(listFormat, "add(4, F)", "-", listArr.toString());

System.out.printf(listFormat, "set(2,G)", listArr.set(2,'G'), listArr.toString());

System.out.printf(listFormat, "get(2)", listArr.get(2), listArr.toString());

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

}

public static void repeatPush\_as(ArrayStack list,int num, int n){

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

list.push(num);

}

}

public static void repeatPop\_as(ArrayStack list,int n){

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

list.pop();

}

}

public static void repeatPush\_ls(LinkedStack list,int num, int n){

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

list.push(num);

}

}

public static void repeatPop\_ls(LinkedStack list,int n){

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

list.pop();

}

}

public static void repeatEnqueue\_aq(ArrayQueue list,int num, int n){

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

list.enqueue(num);

}

}

public static void repeatDequeue\_aq(ArrayQueue list,int n){

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

list.dequeue();

}

}

public static void repeatEnqueue\_lq(LinkedQueue list,int num, int n){

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

list.enqueue(num);

}

}

public static void repeatDequeue\_lq(LinkedQueue list,int n){

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

list.dequeue();

}

}

public static void printASCII(String title, String[] header, long [][] data){

int longest = Long.toString(data[data.length-1][1]).length() + 6;

printHr(longest);

String headerFormat = "| %" + longest + "s | %" + longest + "s | %" + longest + "s |\n";

String dataFormat = "| %," + longest + "d | %," + longest + "d | %," + longest + "d |\n";

String titleFormat = "| %" + longest + "s %" + longest + "s %" + longest + "s |\n";

System.out.printf(titleFormat, " ", title, " ");

printHr(longest);

System.out.printf( headerFormat, header[0], header[1], header[2]);

printHr(longest);

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

System.out.printf( dataFormat, data[i][0], data[i][1], data[i][2] );

}

printHr(longest);

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

}

public static void printHr(int maxNum){

String str = "+";

for(int i = 0; i < (maxNum\*3+14); i++){

str += "-";

}

System.out.println(str + "+");

}

}

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

public static final int CAPACITY = 100; // 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 capacity

data = (E[]) new Object[capacity]; // safe cast; compler 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) {

E[] temp = (E[]) new Object[data.length \* 2];

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

temp[i] = data[i];

data = temp;

temp = null;

}

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 garbage collection

t--;

return answer;

}

/\*\* toString \*/

public String toString(){

String result = "[";

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

result += data[i] + ",";

if( i == data.length - 1 ){

result += data[i] + "]";

}

}

return result;

}

}

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

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

// instance variables

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

public static final int CAPACITY = 100;

// constructors

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

public ArrayQueue(int 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 ) {

E[] temp = (E[]) new Object[data.length \* 2];

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

temp[i] = data[i];

}

data = temp;

temp = null;

};

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 returns 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;

}

}

/\*\*

\* Data Structure & Algorithms 6th Edition

\* Goodrick, Tamassia, GoldWasser

\* Code Fragments 7.2, 7.3, 7.4 and 7.5

\*

\* An implementation of a generic 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; compiler 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 do 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 the element at index i with e, 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);

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++;

}

/\*\* Remove / 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 holl

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);

}

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

protected void resize(int capacity){

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

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

temp[k] = data[k];

data = temp;

}

/\*\* toString method \*/

public String toString(){

String str = "";

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

if(i<size-1)

str += data[i] + ",";

else

str += data[i];

}

return "(" + str + ")";

}

}

/\*\* 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 e) { list.addLast(e); }

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

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

}

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 elemtn){list.addFirst(elemtn);}

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

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

}

/\*

\* A simplified version of the java.util.List interface.

\*/

public interface List<E> {

/\*\* Returns the number of elements in this 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;

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

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

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

E remove(int i)throws IndexOutOfBoundsException;

}

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 joseph.latimer

\*/

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 the 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();

}

// update methods

public void addFirst(E e){

head = new Node<>(e, head); //public Node(E e, Node<E> n){ element = e; next = n;}

if(size == 0)

tail = head;

size++;

}

public void addLast(E e){ // add 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.

\*

\* @author Michael T. Goodrich

\* @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) throws IllegalStateException;

/\*\*

\* 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 is empty)

\*/

E pop();

}

**run:**

**+-----------------------------------------------------------+**

**| ArrayStack Test |**

**+-----------------------------------------------------------+**

**| N | push(nsec) | pop(nsec) |**

**+-----------------------------------------------------------+**

**| 10 | 25,814 | 5,574 |**

**| 100 | 31,144 | 12,418 |**

**| 1,000 | 298,011 | 385,974 |**

**| 10,000 | 736,610 | 353,859 |**

**| 100,000 | 8,254,838 | 2,766,176 |**

**| 1,000,000 | 19,628,023 | 5,828,412 |**

**| 10,000,000 | 144,791,020 | 12,459,262 |**

**+-----------------------------------------------------------+**

**+--------------------------------------------------------------+**

**| LinkedStack Test |**

**+--------------------------------------------------------------+**

**| N | push(nsec) | pop(nsec) |**

**+--------------------------------------------------------------+**

**| 10 | 829,218 | 25,482 |**

**| 100 | 157,466 | 39,051 |**

**| 1,000 | 597,170 | 440,239 |**

**| 10,000 | 3,230,637 | 410,127 |**

**| 100,000 | 6,000,760 | 1,888,837 |**

**| 1,000,000 | 63,443,293 | 2,830,367 |**

**| 10,000,000 | 2,824,642,285 | 55,349,298 |**

**+--------------------------------------------------------------+**

**+--------------------------------------------------------+**

**| ArrayQueue Test |**

**+--------------------------------------------------------+**

**| N | enqueue(nsec) | dequeue(nsec) |**

**+--------------------------------------------------------+**

**| 10 | 16,740 | 15,053 |**

**| 100 | 12,165 | 19,447 |**

**| 1,000 | 527,719 | 667,102 |**

**| 10,000 | 789,281 | 392,788 |**

**| 100,000 | 5,931,675 | 2,152,660 |**

**| 1,000,000 | 21,709,859 | 9,542,134 |**

**| 10,000,000 | 79,683,128 | 87,322,774 |**

**+--------------------------------------------------------+**

**+--------------------------------------------------------------+**

**| LinkedQueue Test |**

**+--------------------------------------------------------------+**

**| N | enqueue(nsec) | dequeue(nsec) |**

**+--------------------------------------------------------------+**

**| 10 | 149,987 | 7,509 |**

**| 100 | 20,711 | 6,463 |**

**| 1,000 | 297,325 | 84,551 |**

**| 10,000 | 2,512,309 | 1,120,646 |**

**| 100,000 | 6,153,646 | 2,594,637 |**

**| 1,000,000 | 130,777,112 | 8,301,506 |**

**| 10,000,000 | 2,920,151,406 | 45,914,296 |**

**+--------------------------------------------------------------+**

**+--------------------------------------------------+**

**| ArrayList Test |**

**+----------------+----------------+----------------+**

**| Method | Return Value | List Contents |**

**+----------------+----------------+----------------+**

**| add(0, A) | - | (A) |**

**| add(0, B) | - | (B,A) |**

**| get(1) | A | (B,A) |**

**| set(2, C) | "error" | (B,A) |**

**| add(2, C) | - | (B,A,C) |**

**| add(4, D) | "error" | (B,A,C) |**

**| reomve(1) | A | (B,C) |**

**| add(1, D) | - | (B,D,C) |**

**| add(1, E) | - | (B,E,D,C) |**

**| get(4) | "error" | (B,E,D,C) |**

**| add(4, F) | - | (B,E,D,C,F) |**

**| set(2,G) | D | (B,E,G,C,F) |**

**| get(2) | G | (B,E,G,C,F) |**

**+--------------------------------------------------+**