/\* Arthur Flores

masc0809

\*/

package data\_structures;

import java.util.Iterator;

import java.util.NoSuchElementException;

public class LinearList<E> implements LinearListADT<E> {

private int currentSize;

private Node<E> head, tail;

public LinearList(){

currentSize=0;

head=tail=null;

}

// Adds the Object obj to the end of list.

public void addLast(E obj){

Node<E> newNode= new Node(obj);

if(tail == null)

head = tail = newNode;

else{

tail.next = newNode;

tail = newNode;

}

currentSize++;

}

// Adds the Object obj to the beginning of list.

public void addFirst(E obj){

Node<E> newNode = new Node(obj);

if(head == null)

head = tail = newNode;

else{

newNode.next = head;

head = newNode;

}

currentSize++;

}

// Inserts the Object obj at the position indicated. If there is an element at

// that location, all elements from that location to the end of the list are

// shifted down to make room for the new insertion. The location is one based.

// If the location > size()+1 then a RuntimeException is thrown. List elements

// must be contiguous.

public void insert(E obj, int location){

if(location>currentSize+1 || location<1)

throw new RuntimeException("invalid input");

Node<E> newNode = new Node(obj);

Node<E> current=head, previous=null;

int where=1;

while(current!=null && location > where){

previous=current;

current=current.next;

where++;

}

if(previous==null){

newNode.next=head;

head=tail=newNode;

}

else{

previous.next=newNode;

newNode.next=current;

}

currentSize++;

}

// Removes the object located at the parameter location (one based).

// Throws a RuntimeException if the location does not map to a valid position within the list.

public E remove(int location){

if(location>currentSize || location<1)

throw new RuntimeException("invalid input");

Node<E> current=head, previous=null;

int where=1;

while(current!=null && location>where){

previous=current;

current=current.next;

where++;

}

if(current==null)

throw new NoSuchElementException();

else if(current==head)

head=head.next;

else if(current==tail){

previous.next=null;

tail=previous;

}else

previous.next=current.next;

E tmp=current.data;

if(head==null)

tail=null;

currentSize--;

return tmp;

}

// Removes and returns the parameter object obj from the list if the list contains it, null otherwise.

// The ordering of the list is preserved. The list may contain duplicate elements. This method

// removes and returns the first matching element found when traversing the list from first position.

public E remove(E obj){

Node<E> current=head;

Node<E> previous=null;

while(current!=null && ((Comparable<E>)obj).compareTo(current.data) !=0){

previous=current;

current=current.next;

}

if(current==null)

return null;

else if(current==head)

head=head.next;

else if(current==tail){

previous.next=null;

tail=previous;

}else

previous.next=current.next;

E tmp = current.data;

if(head==null)

tail=null;

currentSize--;

return tmp;

}

// Removes and returns the parameter object obj in first position in list if the list is not empty,

// null if the list is empty. The ordering of the list is preserved.

public E removeFirst(){

if(head==null)

return null;

E tmp = head.data;

head=head.next;

currentSize--;

return tmp;

}

// Removes and returns the parameter object obj in last position in list if the list is not empty,

// null if the list is empty. The ordering of the list is preserved.

public E removeLast(){

Node<E> current=head,previous=null;

if(current==null)

return null;

while(current.next != null){

previous=current;

current=current.next;

}

E tmp= current.data;

if(current==head)

head=tail=null;

else{

previous.next=null;

tail=previous;

}

currentSize--;

return tmp;

}

// Returns the parameter object located at the parameter location position (one based).

// Throws a RuntimeException if the location does not map to a valid position within the list.

public E get(int location){

if(location>currentSize || location<1)

throw new RuntimeException("invalid input");

Node<E> current=head, previous=null;

int where=1;

while(current!=null && location>where){

previous=current;

current=current.next;

where++;

}

if(previous==null)

throw new NoSuchElementException();

E tmp=current.data;

return tmp;

}

// Returns true if the parameter object obj is in the list, false otherwise.

public boolean contains(E obj){

Node<E> tmp=head;

while(tmp!= null){

if(((Comparable<E>)obj).compareTo(tmp.data)==0)

return true;

tmp=tmp.next;

}

return false;

}

// Returns the one based location of the parameter object obj if it is in the list, -1 otherwise.

// In the case of duplicates, this method returns the element closest to position #1.

public int locate(E obj){

Node<E> current=head,previous=null;

int where=1;

while(current!=null && ((Comparable<E>) obj).compareTo(current.data) !=0){

previous=current;

current=current.next;

where++;

}

if(current==null)

return -1;

return where;

}

// The list is returned to an empty state.

public void clear(){

head=tail=null;

currentSize=0;

}

// Returns true if the list is empty, otherwise false

public boolean isEmpty(){

return currentSize==0;

}

// Returns the number of Objects currently in the list.

public int size(){

return currentSize;

}

// Returns an Iterator of the values in the list, presented in

// the same order as the underlying order of the list. (position #1 first)

public Iterator<E> iterator(){

return new IteratorHelper();

}

class IteratorHelper implements Iterator<E> {

Node<E> iterIndex;

public IteratorHelper() {

iterIndex=head;

}

public boolean hasNext() {

return iterIndex != null;

}

public E next() {

if(!hasNext())

throw new NoSuchElementException();

E tmp = iterIndex.data;

iterIndex = iterIndex.next;

return tmp;

}

public void remove() {

throw new UnsupportedOperationException();

}

}

class Node<T>{

T data;

Node<T> next;

public Node(T obj){

data=obj;

next=null;

}

}

}