**Problem1:**

**public static** <T> **void** clear(ArrayList<T> l){

**while**(! l.empty())

l.remove();

}

Change the type from ArrayList to LinkedList.

**public static** <T> **void** insertAll(List<T> l1,List <T>l2,**int** i){

l1.findFirst();

**for** (**int** k = 0 ; k < i; k++)

l1.findNext();

l2.findFirst();

**while**(! l2.last()){

l1.insert(l2.retrieve());

l2.findNext();

}

l1.insert(l2.retrieve());

}

**public static** <T>**void** commonE(List<T> l1, List<T> l2, List<T>

cl){

**boolean** found;

l1.findFirst();

**while**(! l1.last()){

l2.findFirst();

found = **false**;

**while**(! l2.last() && ! found){

**if** (l1.retrieve().equals(l2.retrieve()))

found = **true**;

**else**

l2.findNext();

}

**if** (l1.retrieve().equals(l2.retrieve()))

found = **true**;

**if**(found)

cl.insert(l1.retrieve());

l1.findNext();

}

l2.findFirst();

found = **false**;

**while**(! l2.last() && ! found){

**if** (l1.retrieve().equals(l2.retrieve()))

found = **true**;

**else**

l2.findNext();

}

**if** (l1.retrieve().equals(l2.retrieve()))

found = **true**;

**if**(found)

cl.insert(l1.retrieve());

}

**B)**

**public void** removeIth(**int** i)

{

Node<T> prev;

**int** k = 0;

current = head;

prev = **null**;

**while**(current != **null** && k < i)

{

prev = current;

current = current.next;

k++;

}

**if**(current != **null**)

{

**if** (prev == **null**)

head = head.next;

**else**

prev.next = current.next;

**if** (head == **null**)

current = **null**;

**else if** (current.next == **null**)

current = head;

**else**

current = current.next;

}

}

**public void** removeEvenElems()

{

**int** j = 1;

**for** (**int** i = 0 ; i < size/2 ; i++)

{

nodes[i] = nodes[j];

j +=2;

}

size = size/2;

}

**Problem3/1**

public boolean checkListEndsSymmetry(DoubleLinkedList<T> dl, int

k)

{

DNode<T> t1 = head;

DNode<T> t2 = head;

while(t2.next != null)

t2 = t2.next;

int i = 1;

while(i <= k && t1.data.equals(t2.data))

{

t1 = t1.next;

t2 = t2.previous;

i++;

}

if (i > k)

return true;

else

return false;

}

**Problem3/2**

**public void** bubbleSort(DoubleLinkedList<Integer> dl)

{

**int** n = 0;

DNode<Integer> cur1, cur2;

cur1 = dl.head;

**while** (cur1 != **null**)

{

n++;

cur1 = cur1.next;

}

cur1 = dl.head;

**for** (**int** i = 0; i < n - 1; i++)

{

cur2 = dl.head;

**for** (**int** j = 0; j < n - 1 - i; j++)

{

**if** (cur2.data > cur2.next.data)

{

Integer tmp = cur2.data;

cur2.data = cur2.next.data;

cur2.next.data = tmp;

}

cur2 = cur2.next;

}

cur1 = cur1.next;

}

}

**Problem4:**

**public class** DArrayList<T> **implements** List<T>

{

**private** T[] data;

**public int** current, size, maxSize;

**private static final double *minRatio*** = 0.4;

**public** DArrayList()

{

data = (T[]) **ne w** Object[1];

maxSize = 1;

current = -1;

size = 0;

}

**public boolean** full()

{

**return** size == maxSize;

}

**public boolean** empty()

{

**return** size == 0;

}

**public boolean** last()

{

**return** current == size - 1;

}

**public void** findFirst()

{

current = 0;

}

**public void** findNext()

{

current++;

}

**public** T retrieve()

{

**return** data[current];

}

**public void** update(T val)

{

data[current] = val;

}

**public void** insert(T val)

{

**if**(size == maxSize)

{

maxSize = maxSize \* 2;

T[] temp = (T[]) **ne w** Object[ maxSize ];

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

temp[i] = data[i];

data = temp;

}

**for** (**int** i = size - 1; i > current; --i)

{

data[i + 1] = data[i];

}

current++;

data[current] = val;

size++;

}

**public void** remove()

{

**if**(size < maxSize \* ***minRatio***)

{

maxSize = maxSize / 2;

T[] temp = (T[]) **ne w** Object[ maxSize ];

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

temp[i] = data[i];

data = temp;

}

**for** (**int** i = current + 1; i < size; i++)

{

data[i - 1] = data[i];

}

size--;

**if** (size == 0)

current = -1;

**else if** (current == size)

current = 0;

}

}

**Problem5:**

**ADT List: Specification**

**Elements:** The elements are of generic type <Type> are placed in

nodes.

**Structure:** the elements are linearly arranged. The first element is

called head, there is a element called current, last elements point

to first element.

**Domain:** the number of elements in the list is bounded therefore the

domain is finite. Type name of elements in the domain: List

**Operations:** We assume all operations operate on a list L.

**Method FindFirst()**

**requires:** list L is not empty.

**input:** none.

**output:** none.

**results:** first element set as the current element.

**Method FindNext()**

**requires**: list L is not empty. Current is not last.

**input**: none.

**output**: none.

**results**: element following the current element is made the

current element, if the current is last than the current made

the first node.

**Method Retrieve(Type e)**

**requires**: list L is not empty.

**input**: none

**output**: element e.

**results**: current element is copied into e.

**Metho d Update(Type e).**

**requires**: list L is not empty.

**input**: e.

**output**: none.

**results**: the element e is copied into the current node.

**Method Insert(Type e).**

**requires**: list L is not full.

**input**: e.

**output**: none.

**results**: a new node containing element e is created and

inserted after the current element in the list. The new

element e is made the current element. If the list is empty e

is also made the head element.

**Method Remove()**

**requires**: list L is not empty.

**input**: none.

**output**: none.

**results**: the current element is removed from the list. If the

resulting list is empty current is set to NULL. If successor

of the deleted element exists it is made the new current

element otherwise first element is made the new current

element.

**Method Full(boolean flag)**

**requires:** none.

**input**: none.

**output**: flag.

**returns**: if the number of elements in L has reached the

maximum number allowed then flag is set to true otherwise

false.

**Method Empty(boolean flag).**

**Requires**: none.

**input**: none.

**results**: if the number of elements in L is zero, then flag is

set to true otherwise false.

**Output**: flag.

**Method Last (boolean flag).**

**requires**: L is not empty.

**input**: none.

**Output**: flag.

**Results**: if the last element is the current element then flag

is set to true otherwise false.

**public interface** CircularList<T>

{

**public void** findFirst();

**public void** findNext();

**public** T retrieve();

**public void** update(T e);

**public void** insert(T e);

**public void** remove();

**public boolean** full();

**public boolean** empty();

**public boolean** last();

}

User Method

**public static void** print(CircularList<Integer> l)

{

**if** (l.empty())

System.***out***.println("Empty List");

**else**

{

**int** size1 = 0;

System.***out***.println("List Contenets starting from

current");

**while**(! l.last())

{

System.***out***.print(l.retrieve() + ", ");

size1++;

l.findNext();

}

System.***out***.println(l.retrieve());

**int** size2 = 0;

l.findFirst();

**while** (!l.last())

{

size2++;

l.findNext();

}

l.findFirst();

**for**(**int** i = 1 ; i <= size2 - size1 ; i++)

l.findNext();

}

}

Memeber method

**public void** print(CircularList<Integer> l)

{

**if**(current == **null**)

System.***out***.println("Empty List");

**else**

{

System.***out***.println("List Contenets starting from

current");

Node<T> tmp = current;

**while**(! l.last())

{

System.***out***.print(l.retrieve() + ", ");

l.findNext();

}

System.***out***.println(l.retrieve());

current = tmp;

}

**public class** LinkedCircularList<T> **implements** CircularList<T>

{

**private** Node<T> head;

**private** Node<T> current;

**public** LinkedCircularList()

{

head = current = **null**;

}

**public boolean** empty()

{

**return** head == **null**;

}

**public boolean** last()

{

**return** current.next == head;

}

**public boolean** full()

{

**return false**;

}

**public void** findFirst()

{

current = head;

}

**public void** findNext()

{

current = current.next;

}

**public** T retrieve()

{

**return** current.data;

}

**public void** update(T val)

{

current.data = val;

}

**public void** insert(T val)

{

Node<T> tmp;

**if** (empty())

{

current = head = **new** Node<T> (val);

current.next = head;

}

**else**

{

tmp = current.next;

current.next = **new** Node<T> (val);

current = current.next;

current.next = tmp;

}

}

**public void** remove()

{

**if** (head.next == head)

head = current = **null**;

**else if** (current == head)

{

Node<T> tmp = head;

head = head.next;

current = head;

**while**(current.next != tmp)

current = current.next;

current.next = head;

current = head;

}

**else**

{

Node<T> tmp = head;

**while** (tmp.next != current)

tmp = tmp.next;

tmp.next = current.next;

current = current.next;

}

}

}