NATIONAL UNIVERSITY OF DISTANCE EDUCATION

# Estrategias de Programación y Estructuras de Datos

**Idioma:** EN

**INSTRUCTIONS:**
Programming Strategies and Data Structures. June 2025 · 2nd Week

Exercises that require programming must be done in Java, using the course ADTs (the interfaces for these ADTs are attached to this statement).

Cost-calculation exercises require explicitly stating the problem size. If this is not done, the answer will not be evaluated.

All answers must be justified; answers without justification will not be evaluated.

ADT Interfaces

CollectionIF
```java
public interface CollectionIF {
public int size();
public boolean isEmpty();
public boolean contains(E e);
public void clear();
}
```

SequenceIF
```java
public interface SequenceIF extends CollectionIF {
public IteratorIF iterator();
}
```

ListIF
```java
public interface ListIF extends SequenceIF {
public E get(int pos);
public void set(int pos, E e);
public void insert(int pos, E elem);
public void remove(int pos);
}
```

StackIF
```java
public interface StackIF extends SequenceIF {
public E getTop();
public void push(E elem);
public void pop();
}
```

QueueIF
```java
public interface QueueIF extends SequenceIF {
public E getFirst();
```

public void enqueue(E elem);
public void dequeue();
}

TreeIF
```java
public interface TreeIF extends CollectionIF {
public E getRoot();
public boolean isLeaf();
public int getNumChildren();
public int getFanOut();
public int getHeight();
public IteratorIF iterator(Object mode);
}
```

GTreeIF
```java
public interface GTreeIF extends TreeIF {
enum IteratorModes { PREORDER, POSTORDER, BREADTH }
public void setRoot(E e);
public ListIF> getChildren();
public GTreeIF getChild(int pos);
public void addChild(int pos, GTreeIF e);
public void removeChild(int pos);
}
```

BTreeIF
```java
public interface BTreeIF extends TreeIF {
enum IteratorModes { PREORDER, POSTORDER, BREADTH, INORDER, RLBREADTH }
public BTreeIF getLeftChild();
public BTreeIF getRightChild();
public void setRoot(E e);
public void setLeftChild(BTreeIF child);
public void setRightChild(BTreeIF child);
public void removeLeftChild();
public void removeRightChild();
}
```

BSTreeIF
```java
public interface BSTreeIF> extends TreeIF {
enum IteratorModes { DIRECTORDER, REVERSEORDER }
enum Order { ASCENDING, DESCENDING }

public BSTree getLeftChild();
public BSTree getRightChild();
public void add(E e);
public void remove(E e);
public Order getOrder();
}
```

## Question 1

Practice question.
It is required to program an operation:
```java
ListIF getTasksBetweenDates(int dI, int dF)
```
that returns the list of tasks to be performed between dates dI and dF, both included, specified by the parameters and stored in the future task scheduler.
As a precondition, assume that dI < dF.

a) (1 point) Implement getTaskBetweenDates(dI, dF) so that it is independent of the structure chosen to implement the task scheduler.

b) (1 point) Compute the worst-case asymptotic time cost of getTaskBetweenDates(dI, dF).

**STUDENT RESPONSE:**

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class TaskScheduler {

private List<TaskIF> tasks;

public TaskScheduler(List<TaskIF> tasks) {
this.tasks = tasks;
}

public ListIF<TaskIF> getTasksBetweenDates(int dI, int dF) {
List<TaskIF> result = new ArrayList<>();
for (TaskIF task : tasks) {
int taskDate = task.getDate();
if (taskDate >= dI && taskDate <= dF) {
result.add(task);
}
}
return new ArrayListWrapper(result);
}

private static class ArrayListWrapper implements ListIF<TaskIF> {
private List<TaskIF> list;

public ArrayListWrapper(List<TaskIF> list) {
this.list = list;
}

@Override
public int size() {
return list.size();
}
@Override
public boolean isEmpty() {
return list.isEmpty();
}

@Override
public boolean contains(TaskIF e) {
return list.contains(e);
}

@Override
public void clear() {
list.clear();
}

@Override
public Iterator<TaskIF> iterator() {
return list.iterator();
}
}
}

interface TaskIF {
int getDate();
}

// Worst-case asymptotic time cost: O(n)
```

## Question 2

Analyze the following code fragments and determine their worst-case asymptotic time cost:

a) (1.5 points)

```java
int i = 1;
while (i < n) {
System.out.println(i);
i *= 2;
}
```

b) (1.5 points)

```java
public static int dum(int n) {
if (n == 0) {
return 0;
} else if (n <= 3) {
return 1;
} else {
return dum(n-1) + dum(n-2) + dum(n-3);
}
}
```

**STUDENT RESPONSE:**

a) O(log n)
b) O(3^n)

## Question 3

Suppose a turn-based game with several players whose number can decrease during the game when someone is eliminated during their turn. Each player performs an action and then passes the turn to the next player, in circular order.

You are asked to program a Data Type that lets us manage the order in which players take their turns. For this, two operations are needed:
- pasarTurno(): advances the turn to the next player.
- eliminarJugador(): removes the current player from the game.

a) (0.5 points) Indicate which data structure studied in the course would be the most appropriate, so that the cost of both operations is optimized. Also indicate how the current player can be identified in that structure.

b) (1.5 points) Implement pasarTurno() and eliminarJugador().

**STUDENT RESPONSE:**

a) A circular linked list is the most appropriate data structure. The current player can be identified by maintaining a reference to the current node in the list.

b)
```java
public class CircularLinkedList<E> {

private Node<E> current;
private Node<E> tail;
private int size;

private static class Node<E> {
E data;
Node<E> next;

Node(E data) {
this.data = data;
}
}

public CircularLinkedList() {
current = null;
tail = null;
size = 0;
}

public void add(E element) {
Node<E> newNode = new Node<>(element);
if (isEmpty()) {
current = newNode;
tail = newNode;
current.next = current;
} else {
```

```
tail.next = newNode;
tail = newNode;
tail.next = current;
}
size++;
}

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

public void pasarTurno() {
if (!isEmpty()) {
current = current.next;
}
}

public void eliminarJugador() {
if (!isEmpty()) {
if (size == 1) {
current = null;
tail = null;
} else {
Node<E> nextNode = current.next;
if (current == tail) {
tail = nextNode;
}
current = nextNode;
tail.next = current;
}
size--;
}
}

public E getCurrentPlayer() {
return (current != null) ? current.data : null;
}

public int getSize() {
return size;
}
}
```

## Question 4

Given the following tree, indicate the sequence of nodes visited by preorder, postorder, and breadth-first traversals (from left to right).

1
/ | \n 2 3 4
/ \ |
5 6 7

Explain how each traversal is performed.

**STUDENT RESPONSE:**

a) Initial tree: 3 as root. 1 inserted as left child. 2 inserted as right child of 1.
Unbalanced at 3. Right rotation at 3 needed.
Resulting tree: 1 as root. 3 as right child. 2 as left child of 3.

b) Initial tree: 3 as root. 2 inserted as left child. 1 inserted as left child of 2.
Unbalanced at 3. Right rotation at 3 needed.
Resulting tree: 2 as root. 3 as right child. 1 as left child of 2.

c) Initial tree: 7 as root. 15 inserted as right child. 10 inserted as left child of 15.
Unbalanced at 7. Left rotation at 15 needed.
Resulting tree: 10 as root. 7 as left child. 15 as right child.

d) Initial tree: 11 as root. 22 inserted as right child. 35 inserted as right child of 22.
Unbalanced at 11. Left rotation at 22 needed.
Resulting tree: 22 as root. 11 as left child. 35 as right child.