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:

a)
```java
public ListIF<TaskIF> getTasksBetweenDates(int dI, int dF) {
ListIF<TaskIF> result = new List<TaskIF>();
IteratorIF<TaskIF> it = scheduler.iterator();
while (it.hasNext()) {
TaskIF t = it.getNext();
int d = t.getDate();
if (d >= dI && d <= dF) {
result.insert(result.size(), t);
}
}
return result;
}
```

b)
Let n be the total number of tasks stored in the scheduler.
In the worst case, all tasks are traversed once and each iteration performs constant-time operations. Therefore, the worst-case asymptotic time cost is 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) Worst-case asymptotic time cost: Θ(log n).

b) Worst-case asymptotic time cost: Θ(α^n), where α is the real root greater than 1 of the equation x^3 = x^2 + x + 1 (approximately α ≈ 1.839).

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)
The most appropriate data structure is a circular queue.
The current player is identified as the element at the front of the queue.

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

private QueueIF<E> players;

public TurnManager(QueueIF<E> players) {
this.players = players;
}

public void pasarTurno() {
if (!players.isEmpty()) {
E current = players.getFirst();
players.dequeue();
players.enqueue(current);
}
}

public void eliminarJugador() {
if (!players.isEmpty()) {
players.dequeue();
}
}
}
```

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) Sequence 3, 1, 2
Insertion produces the tree: root 3, left child 1, and right child of 1 equal to 2.
This is a left-right imbalance.
Required rotation: double rotation, first left rotation on node 1, then right rotation on node 3.
Resulting AVL tree: root 2, left child 1, right child 3.

b) Sequence 3, 2, 1
Insertion produces the tree: root 3, left child 2, and left child of 2 equal to 1.
This is a left-left imbalance.
Required rotation: single right rotation on node 3.
Resulting AVL tree: root 2, left child 1, right child 3.

c) Sequence 7, 15, 10
Insertion produces the tree: root 7, right child 15, and left child of 15 equal to 10.
This is a right-left imbalance.
Required rotation: double rotation, first right rotation on node 15, then left rotation on node 7.
Resulting AVL tree: root 10, left child 7, right child 15.

d) Sequence 11, 22, 35
Insertion produces the tree: root 11, right child 22, and right child of 22 equal to 35.
This is a right-right imbalance.
Required rotation: single left rotation on node 11.
Resulting AVL tree: root 22, left child 11, right child 35.