UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Estrategias de Programación y Estructuras de Datos

Idioma: ES

INSTRUCCIONES:
Estrategias de Programación y Estructuras de Datos. Junio 2025 · 2ª Semana

Los ejercicios que requieran programación se deberán realizar en lenguaje Java, utilizando los TADs de la asignatura (adjuntas a este enunciado están las interfaces de dichos TADs).

Los ejercicios de cálculo de coste requieren que se explicite cuál es el tamaño del problema. Si no se hace, la respuesta no se evaluará.

Todas las respuestas deberán justificarse; no se evaluarán respuestas sin justificar.

Interfaces de los TADs

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();
}
```

Pregunta 1

Pregunta sobre la práctica.
Se desea programar una operación:
```java
ListIF getTasksBetweenDates(int dI, int dF)
```
que devuelva la lista de tareas que deben realizarse entre las fechas dI y dF, ambas incluidas, indicadas por los parámetros y que se encuentran almacenadas en el planificador de tareas futuras.
Como precondición asumiremos que dI < dF.

a) (1 punto) Programe la operación getTaskBetweenDates(dI, dF) de forma que sea independiente de la estructura escogida para implementar el planificador de tareas.

b) (1 punto) Calcule el coste asintótico temporal en el caso peor de la operación getTaskBetweenDates(dI, dF).

RESPUESTA DEL ESTUDIANTE:

a) La operación getTaskBetweenDates(dI, dF) debe recorrer el planificador de tareas futuras, iterando sobre cada tarea t almacenada en él. Para ello, podemos utilizar la interfaz SequenceIF. La operación de iteración deberá comprobar que la fecha de inicio de la tarea t está comprendida entre las fechas dI y dF. La operación de iteración debe devolver una lista de todas las tareas t que cumplan esta condición.

La operación de iteración debe aplicarse a todos los planificadores de tareas futuras, lo cual implica que la operación getTaskBetweenDates(dI, dF) debe ser independiente de la estructura escogida para implementar el planificador de tareas.

b) El coste asintótico temporal en el caso peor de la operación getTaskBetweenDates(dI, dF) depende del tamaño del planificador de tareas futuras. Si el tamaño del planificador es N, entonces el coste asintótico temporal en el caso peor de la operación getTaskBetweenDates(dI, dF) será O(N).

Pregunta 2

Analice los siguientes fragmentos de código y determine su coste asintótico temporal en el caso peor:

a) (1,5 puntos)
```java
int i = 1;
while (i < n) {
System.out.println(i);
i *= 2;
}
```

b) (1,5 puntos)
```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);
}
}
```

RESPUESTA DEL ESTUDIANTE:

a) (1,5 puntos) El coste asintótico temporal en el caso peor del fragmento de código es O(n^2), ya que el bucle while se ejecuta n veces y dentro de él se realiza una multiplicación por 2 que también se repite n veces.
b) (1,5 puntos) El coste asintótico temporal en el caso peor del método dum es O(2^n), ya que la recursividad se repite hasta llegar al límite inferior de n <= 3 y luego se suma tres veces el valor de dum(n-1).

Pregunta 3

Supongamos un juego por turnos con varios jugadores cuyo número puede disminuir durante el juego cuando alguno queda eliminado durante su turno. Cada jugador realiza una acción y luego pasa el turno al siguiente, en orden circular.

Se desea programar un Tipo de Datos que nos permita gestionar el orden en el que los jugadores van a tener su turno. Para ello, se necesitan dos operaciones:
- pasarTurno(): avanza el turno al siguiente jugador.
- eliminarJugador(): elimina al jugador actual del juego.

a) (0,5 puntos) Indique qué estructura de datos de las estudiadas en la asignatura sería la más adecuada, de forma que se optimice el coste de las dos operaciones anteriores. Indique, además, cómo se puede identificar al jugador actual en dicha estructura.

b) (1,5 puntos) Programe las operaciones pasarTurno() y eliminarJugador().

RESPUESTA DEL ESTUDIANTE:

a) La estructura de datos más adecuada para este caso sería una cola circular. En esta estructura, cada elemento representa a un jugador y el último elemento en entrar en la cola es el primer jugador en salir y tomar su turno. De esta manera, se puede avanzar el turno al siguiente jugador de manera eficiente mediante la inserción y extracción de elementos.

Para identificar al jugador actual, se puede utilizar el elemento que está en la parte frontal de la cola. Al avanzar el turno, simplemente se extrae ese elemento y se inserta al final de la cola, lo que indica que el siguiente jugador tomará su turno.

b) La implementación de las operaciones pasarTurno() y eliminarJugador() se pueden realizar de la siguiente manera:

```java
import java.util.Deque;
import java.util.LinkedList;

class ColaCircular {
private Deque<Jugador> cola;

public ColaCircular() {
this.cola = new LinkedList<>();
}

// Método para avanzar el turno al siguiente jugador
public void pasarTurno() {
if (!cola.isEmpty()) {
Jugador jugadorActual = cola.removeFirst();
cola.addLast(jugadorActual);
}
}

// Método para eliminar al jugador actual del juego
```

```
public void eliminarJugador() {
if (!cola.isEmpty()) {
cola.removeFirst();
}
}

// Método para agregar un nuevo jugador a la cola
public void agregarJugador(Jugador jugador) {
cola.addLast(jugador);
}

// Método para obtener el jugador actual
public Jugador obtenerJugadorActual() {
return cola.peekFirst();
}
}

class Jugador {
private String nombre;

public Jugador(String nombre) {
this.nombre = nombre;
}

public String getNombre() {
return nombre;
}
}
```

En este ejemplo, utilizamos una cola circular implementada usando la clase `Deque` de Java, la cual permite la inserción y extracción de elementos desde ambos extremos. Los métodos `pasarTurno()` y `eliminarJugador()` realizan las operaciones correspondientes, mientras que el método `obtenerJugadorActual()` retorna al jugador actual.

Esta implementación asegura que las operaciones `pasarTurno()` y `eliminarJugador()` sean eficientes, ya que solo involucran movimientos de elementos dentro de la cola.

Pregunta 4

Para que un árbol binario de búsqueda no equilibrado vuelva a cumplir las condiciones para ser un árbol AVL, es necesario aplicar rotaciones.

Secuencias:
a) 3, 1, 2
b) 3, 2, 1
c) 7, 15, 10
d) 11, 22, 35

Se pide construir los árboles, explicar la rotación necesaria y aplicarla.

RESPUESTA DEL ESTUDIANTE:

a) 3, 1, 2

El árbol inicial es:

```
3
/ \
1 2
```

La rotación necesaria es:

```
1
/ \
3 2
```

b) 3, 2, 1

El árbol inicial es:

```
3
/ \
2 1
```

La rotación necesaria es:

```
3
/ \
1 2
```

c) 7, 15, 10

El árbol inicial es:

```
7
/ \
15 10
```

La rotación necesaria es:

```
15
/ \
7 10
```

d) 11, 22, 35

El árbol inicial es:

```
11
/ \
22 35
```

La rotación necesaria es:

```
22
/ \
11 35
```