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:

Para la pregunta 1, se ha programado la operación getTaskBetweenDates(dI, dF) de forma que sea independiente de la estructura escogida para implementar el planificador de tareas. La operación se ha implementado de tal manera que se garantiza que el coste asintótico temporal en el caso peor sea mínimo.

Para la pregunta 2, se ha calculado el coste asintótico temporal en el caso peor de la operación dum(n). El coste asintótico temporal en el caso peor es de O(n^2). Esto se debe a que la operación se ejecuta n veces, donde n es el tamaño de la lista.

Para la pregunta 3, se ha propuesto un tipo de datos que permite gestionar el orden en el que los jugadores van a tener su turno. Se ha utilizado una estructura de datos que consiste en una lista ordenada de jugadores. La operación eliminarJugador() se utiliza para eliminar al jugador actual del juego y la operación pasarTurno() se utiliza para avanzar el turno al siguiente jugador.

Para la pregunta 4: Se han construido árboles binarios de búsqueda no equilibrados mediante rotaciones. Para cada secuencia, se ha seleccionado un árbol y se ha aplicado una rotación necesaria para que el árbol vuelva a cumplir las condiciones para ser un árbol AVL.

Pregunta 5: Los problemas de este examen se pueden clasificar en función de los requisitos de tiempo de ejecución, es decir, el tiempo que tarda el programa en ejecutarse.

a) (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);
}
}
```

b) (1,5 puntos)

```java
public static int dum(int n) {
if (n == 0) {
return 0;
trees:
for (int i = 1; i <= n; i++) {
if (i < 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:

En la pregunta 2, el costo asintótico temporal en el caso peor se puede calcular mediante el siguiente algoritmo:

1. La primera parte del fragmento, `while (i < n)` se ejecuta `n` veces, donde `i` comienza en 1 y aumenta en 2 en cada iteración. Por lo tanto, el tiempo total para ejecutar esta parte es:

$$
\text{Tiempo total} = \frac{1}{n} \cdot n = \frac{1}{n} \cdot n = n
$$

2. La segunda parte del fragmento, `int i = 1;` se ejecuta una vez, donde se establece `i` en 1. Por lo tanto, el tiempo total para ejecutar esta parte es:

$$
\text{Tiempo total} = 1 \cdot 1 = 1
$$

3. La tercera parte del fragmento, `int dum(int n)` se ejecuta `n` veces, donde `n` puede variar entre 0 y 3. Por lo tanto, el tiempo total para ejecutar esta parte es:

$$
\text{Tiempo total} = n \cdot n
$$

Por lo tanto, el costo asintótico temporal en el caso peor es `n` veces el tiempo total.

Sección 1: Estrategias de Programación y Estructuras de Datos del Grado en Ingeniería Informática

Los objetivos de esta sección son:

- Entender los conceptos básicos de estructuras de datos y programación recursiva.
- Aprender a utilizar las estructuras de datos básicas de Java, como colecciones, listas, pilas y árboles.
- Comprender cómo se pueden implementar algoritmos y procesos en Java utilizando diferentes estructuras de datos.
- Analizar y resolver problemas relacionados con la programación recursiva, anillos y bucles.
- Estudiar la teoría de algoritmos y sus aplicaciones en situaciones prácticas.

Para alcanzar estos objetivos, se espera que el estudiante comprenda los siguientes conceptos:

- Estructuras de datos: tipos de datos y sus características.
- Algoritmos: definición, pasos y resultados.
- Programación recursiva: concepto y ejemplos prácticos.
- Anillos y bucles: conceptos y aplicaciones.
- Teoría de algoritmos: fundamentos y aplicaciones en situaciones prácticas.

Es importante destacar que la comprensión de estos conceptos permitirá al estudiante resolver problemas y aplicar conocimientos en situaciones prácticas. Además, la capacidad de utilizar diferentes estructuras de datos para resolver problemas es esencial para el desarrollo de habilidades en programación y resolución de problemas.

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:

En la primera parte de esta pregunta, es importante elegir una estructura de datos adecuada para almacenar el orden de los jugadores y para representar las acciones que cada jugador realiza. Una posible estructura de datos que cumple con estas condiciones es una lista enlazada o linked list.

Para pasar el turno al siguiente jugador, podríamos utilizar una operación llamada "borrar" o "eliminar" que elimine al jugador actual del juego. La operación "eliminarJugador()" podría estar implementada en la clase "Player" o en una clase relacionada.

Para eliminar al jugador actual del juego, podríamos utilizar una operación llamada "quitar" o "remove" que elimine al jugador actual del juego. La operación "qu

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:

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

En primer lugar, construimos un árbol binario de búsqueda no equilibrado utilizando las secuencias dadas:
Secuencia a: 3, 1, 2
Secuencia b: 3, 2, 1
Secuencia c: 7, 15, 10
Secuencia d: 11, 22, 35

A continuación, aplicamos la rotación a la secuencia a:
1. Removemos el elemento 3 de la secuencia a, ya que no está en ninguna rama del árbol.
2. Dividimos la secuencia a en dos ramas: una rama a *que contiene 1 y una rama b* que contiene 2.
3. Aplicamos la rotación a la rama a *a la rama b*, haciendo que la rama b* se mueva hacia la izquierda.

Luego, aplicamos la rotación a la secuencia b:
1. Dividimos la secuencia b en dos ramas: una rama b *que contiene 2 y una rama c* que contiene 1.
2. Aplicamos la rotación a la rama c *a la rama b*, haciendo que la rama b* se mueva hacia la derecha.

Finalmente, aplicamos la rotación a la secuencia c:
1. Dividimos la secuencia c en dos ramas: una rama c *que contiene 1 y una rama d* que contiene 10.
2. Aplicamos la rotación a la rama d *a la rama c*, haciendo que la rama c* se mueva hacia la izquierda.

Con esto, hemos vuelto a tener un árbol binario de búsqueda no equilibrado que cumple con las condiciones para ser un árbol AVL.