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

The implementation of `getTaskBetweenDates` should not depend on the specific data structure used to store the tasks. Here's a general approach:

```java
public class TaskScheduler {
// Assume `tasks` is a data structure representing the task scheduler.
private CollectionIF<TaskIF> tasks;

public TaskScheduler() {
this.tasks = new ArrayList<>();
}

public ListIF<TaskIF> getTasksBetweenDates(int dI, int dF) {
ListIF<TaskIF> result = new ArrayList<>();

// Loop through all tasks in the scheduler.
for (TaskIF task : tasks) {
// Check if the task falls within the date range [dI, dF].
if (task.getDate() >= dI && task.getDate() <= dF) {
result.add(task);
}
}

return result;
}
}
```

This implementation uses a generic `CollectionIF` for the task scheduler, allowing it to work with any data structure that implements `CollectionIF`. The method iterates over all tasks in the scheduler and adds those that fall within the specified date range to the result list.

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

The time complexity of `getTaskBetweenDates` depends on the specific data structure used to represent the task scheduler. If we assume that the scheduler is implemented as a simple array, the time complexity would be `O(n)` where `n` is the number of tasks in the scheduler. This is because the method iterates over all tasks in the scheduler, checking each one to see if it falls within the specified date range.

However, if the scheduler is implemented using a more efficient data structure such as a balanced binary search tree or a hash map, the time complexity could be improved to `O(log n)` or `O(1)`, respectively.

In general, the time complexity of `getTaskBetweenDates` depends on the underlying data structure and the specific implementation details. It is important to consider the trade-offs between performance and space complexity when choosing a data structure for the task scheduler.

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:

The code fragment in part a) uses a while loop where the condition is `i < n`. This loop iterates until `i` equals `n`, which means that the loop runs exactly `n` times. The body of the loop prints `i` and multiplies `i` by `2`. Since `i` starts at `1` and doubles each iteration, the value of `i` after `n` iterations is `2^n - 1`. The total time complexity of the loop is thus `O(n)`.

For the code fragment in part b), we need to analyze the recursive function `dum()`. When `n == 0`, the base case returns `0`, which takes constant time. When `n > 0`, the function makes three recursive calls: `dum(n-1)`, `dum(n-2)`, and `dum(n-3)`. The time complexity of each of these recursive calls is proportional to the time complexity of the entire function up to that point. Therefore, the total time complexity of `dum(n)` is proportional to the sum of the time complexities of all the recursive calls made, which is exponential in the input size `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) The most appropriate data structure to manage the order in which players take their turns is a circular linked list. In a circular linked list, each node contains a reference to the next node in the sequence, and the last node contains a reference to the first node. This allows us to easily advance the turn to the next player by updating the reference to the next node in the sequence. Additionally, we can efficiently identify the current player by keeping track of the position of the current node in the circular linked list.

b) Here's an implementation of the pasarTurno() and eliminarJugador() methods using a circular linked list:

```java
class CircularLinkedListNode {
// ... fields ...
CircularLinkedListNode next;
}

class CircularLinkedList {
// ... fields ...
private CircularLinkedListNode currentNode;

// ... methods ...

public void pasarTurno() {
if (currentNode != null) {
// Advance to the next node in the sequence
currentNode = currentNode.next;
}
}

public void eliminarJugador() {
// Remove the current node from the sequence
// ... implementation ...
}
}
```

In this implementation, the currentNode field stores the reference to the current node in the circular linked list. To advance the turn to the next player, we simply update the currentNode reference to point to the next node in the sequence. Similarly, to remove a player from the game, we can remove the corresponding node from the circular linked list.

Note that in this implementation, we assume that there is always at least one player in the game. If the number of players decreases during the game, we need to adjust the implementation accordingly. For example, we could keep track of the total number of players and ensure that the currentNode reference remains valid even when there are fewer players than nodes in the circular linked list.

We have provided a simple implementation of the circular linked list and the operations required for managing the order of players in a turn-based game. However, it is important to note that this implementation assumes certain conditions, such as the existence of at least one player, and does not handle edge cases or error conditions. It is recommended to review and test the implementation thoroughly to ensure its correctness and robustness.

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:

Answer:

The sequence a) corresponds to the following tree:

```
3
/ \
/ \
1 2
```

To make it an AVL tree, we need to perform a left rotation at level 1:

```
3
/ \
/ \
1 2
|
4
```

The sequence b) corresponds to the following tree:

```
3
/ \
/ \
2 1
```

To make it an AVL tree, we need to perform a right rotation at level 1:

```
3
/ \
/ \
2 1
```

The sequence c) corresponds to the following tree:

```

7
/ \
/ \
15 10
```

To make it an AVL tree, we need to perform a left rotation at level 2:

```

7
/ \
/ \
15 10
/
15
```

Then, we need to perform a right rotation at level 1:

```

7
/ \
/ \
15 10
/
15
|
8
```

The sequence d) corresponds to the following tree:

```

11
/ \
/ \
22 35
```

To make it an AVL tree, we need to perform a left rotation at level 2:

```

11
/ \
/ \
22 35
```

/
22
```

Then, we need to perform a right rotation at level 1:

```
11
/ \
/ \
22 35
/
22
|
22
```
```