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 ListImplementation<TaskIF>();
IteratorIF<TaskIF> it = taskScheduler.iterator();
while (it.hasNext()) {
TaskIF task = it.next();
int date = task.getDate(); // Assuming TaskIF has a getDate() method
if (date >= dI && date <= dF) {
result.insert(result.size(), task);
}
}
return result;
}
```

b)
The problem size is the number of tasks in the task scheduler, let's denote it as n.

The operation iterates over all tasks in the task scheduler once, so the time cost is O(n). For each task, it checks if the task date is within the given range and if so, inserts it into the result list. The insertion operation in a list is O(n) in the worst case if the list is implemented as an array and the insertion is not at the end. However, in this implementation, we are inserting at the end of the list, so the insertion operation is O(1) amortized.

Therefore, the overall time cost is O(n) because we are iterating over n tasks and performing a constant amount of work for each task.

The worst-case asymptotic time cost of getTaskBetweenDates(dI, dF) 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)
The problem size is n.
The loop iterates until i is no longer less than n, with i being multiplied by 2 in each iteration.
The sequence of i is 1, 2, 4, 8, ... , 2^k, where k is the number of iterations.
The loop ends when 2^k >= n, so k = ceil(log2(n)).
The time cost is O(log n).

b)
The problem size is n.
The function dum(n) makes recursive calls to dum(n-1), dum(n-2), and dum(n-3).
The time cost can be represented by the recurrence relation: T(n) = T(n-1) + T(n-2) + T(n-3) + c, where c is a constant.
Solving this recurrence relation is complex, but we can see that the number of calls grows exponentially with n.
The time cost is O(3^n), because in the worst case, the number of operations grows like the tribonacci sequence, which has an exponential growth rate related to the roots of the characteristic equation x^3 - x^2 - x - 1 = 0, with the largest root being approximately 1.839, but for simplicity and given the options typically considered in Big O notation, it is more accurately described as being bounded by 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) The most appropriate data structure to manage the order of players is a circularly linked list or a Queue implemented with a circularly linked list, but more simply, a circular linked list where the last node points to the first node. The current player can be identified by a reference to the current node.

b)

```java
public class TurnManager {
private Node current;
private int size;

private class Node {
Player player;
Node next;

Node(Player player) {
this.player = player;
this.next = null;
}
}

public TurnManager() {
this.current = null;
this.size = 0;
}

public void addPlayer(Player player) {
Node newNode = new Node(player);
if (current == null) {
current = newNode;
newNode.next = newNode;
} else {
newNode.next = current.next;
current.next = newNode;
}
```

```
size++;
}

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

public void eliminarJugador() {
if (current != null) {
if (size == 1) {
current = null;
} else {
Node prev = current;
while (prev.next != current) {
prev = prev.next;
}
prev.next = current.next;
current = current.next;
}
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:

To address question 4, we will examine each given sequence, construct the binary search tree, identify the imbalance, and apply the necessary rotations to balance the tree according to AVL tree properties.

Sequence a) 3, 1, 2

1. Insert 3: The tree is `3`.
2. Insert 1: The tree becomes `3 / 1`.
3. Insert 2: The tree becomes `3 / 1 \ 2`. This results in an unbalanced tree because the left subtree of `3` has a height of 2 and the right subtree has a height of 0, making the balance factor of `3` -2.

To balance, we need to perform a right rotation on `3` and then adjust. However, the correct step involves recognizing the imbalance type. Here, it's a case where a left-right insertion causes imbalance, so we first do a left rotation on `1`, resulting in `3 / 2 / 1`, and then a right rotation on `3`, resulting in `2 / 1 \ 3`.

Sequence b) 3, 2, 1

1. Insert 3: The tree is `3`.
2. Insert 2: The tree becomes `3 / 2`.
3. Insert 1: The tree becomes `3 / 2 / 1`. The tree is unbalanced at `3` because the left subtree has a height of 2 and the right subtree has a height of 0.

To balance, we perform a right rotation on `3`. The resulting balanced tree is `2 / 1 \ 3`.

Sequence c) 7, 15, 10

1. Insert 7: The tree is `7`.
2. Insert 15: The tree becomes `7 \ 15`.
3. Insert 10: The tree becomes `7 \ 15 / 10`. The tree is unbalanced at `7` because the right subtree has a height of 2 and the left subtree has a height of 0.

To balance, we need to perform a left rotation on `7`. However, the correct step involves recognizing that the imbalance is due to a right-left insertion. So, we first do a right rotation on `15`, resulting in `7 \ 10 / 15`, and then no further rotation is needed as it's now balanced.

Sequence d) 11, 22, 35

1. Insert 11: The tree is `11`.
2. Insert 22: The tree becomes `11 \ 22`.
3. Insert 35: The tree becomes `11 \ 22 \ 35`. The tree is unbalanced at `11` because the right subtree has a height of 2 and the left subtree has a height of 0.

To balance, we perform a left rotation on `11`. The resulting tree after rotation is `22 / 11 \ 35`.

The final trees after applying the necessary rotations for each sequence are as follows:

- For sequence a) 3, 1, 2: `2 / 1 \ 3`
- For sequence b) 3, 2, 1: `2 / 1 \ 3`
- For sequence c) 7, 15, 10: `10 / 7 \ 15`
- For sequence d) 11, 22, 35: `22 / 11 \ 35`