Estructura de Datos y Algoritmos
19 de Junio de 2018
2017-2018

Nombre y Apellido: **Grupo:**

Problema 1 - (2 puntos): Escribe la clase **DList** que implementa una lista doblemente enlazada sin nodos centinelas para manejar enteros, con los siguientes requisitos:

1. (0.25) Escriba la cabecera de la clase, los atributos y el método constructor.
2. (0.75) Implementa el método **getAt** que tome un entero como parámetro de entrada y devuelva el elemento que está en esa posición en la lista.

   list: 5<->3<->2<->1
   list.getAt(0)->5
   list.getAt(3)->1
3. (0.75) Implementa el método **insertSort** que tome un entero como parámetro de entrada y lo inserte en la ubicación correspondiente, suponiendo que la lista está ordenada en orden descendente y que no existen elementos repetidos.

   list: 5<->3<->2<->1,
   list.insertSort(4) resultados: list:5<->4<->3<->2<->1.
4. (0.25) ¿Cuál es la complejidad de los métodos **getAt** y **insertSort**? Justifícalo.
   Nota: si necesitas cualquier otro método de la clase (por ejemplo, **isEmpty(),...**), también debes implementarlo.

SOLUCIÓN:

1.

```
public class DList {

	public DNode first;
	public DNode last;

	public DList() {
		first = null;
		last = null;
	}
```

2.

```
public int getAt(int position) {
	int counter = 0;
	for (DNode nodeIt = first; nodeIt != null; nodeIt = nodeIt.next) {
		if (position == counter)
			return nodeIt.element;

		counter++;
	}
	return -1;
}
```

3.

```
public void insertSort(int element) {
	DNode newNode = new DNode(element);
	// List empty
	if (first == null && last == null) {
		first = newNode;
		last = newNode;
		size++;
		return;
	}

	// otherwise
	for (DNode nodeIt = first; nodeIt != null; nodeIt = nodeIt.next) {
		if (element > nodeIt.element) {
			if (nodeIt==first) {
				newNode.next = nodeIt;
				nodeIt.previous = newNode;
				first=newNode;
			} else {
				newNode.next = nodeIt;
				newNode.previous = nodeIt.previous;
				nodeIt.previous.next = newNode;
				nodeIt.previous = newNode;
			}
			size++;
			return;
		}
	}

	last.next =newNode;
	newNode. previous =last;
	last=newNode;
	size++;

}
```

5.

Ambos métodos tienen complejidad temporal de O(n) porque en el peor caso hay que recorrer la lista para recoger un elemento (**getAt**), o para insertar un elemento (**insertSort**).

Problema 2 - (1.5 puntos): Escribe un método estático y recursivo para invertir una lista doblemente enlazada de enteros (debes utilizar la clase implementada en el problema anterior). El método toma una lista doblemente enlazada como parámetro de entrada y no devuelve nada.

list: 3<->2<->1<->0 *reverse(list)-> list: 0<->1<->2<->3*

SOLUCIÓN:

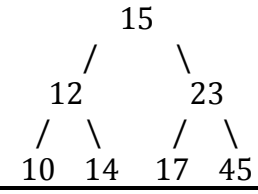
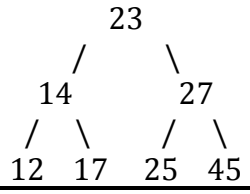
```
public static void reverse(DList list) {
      if (list==null) return;
      if (list.size>0) {
            int elem=list.removeFirst();
            reverse(list);
            list.addLast(elem);
      }
}
```

Problema 3 - (1,5 puntos): Escribe un método estático y recursivo que tome un array de cadenas de caracteres como parámetro de entrada y devuelva la cadena de mayor longitud en el array. El algoritmo debe estar basado en el enfoque de divide y vencerás.

SOLUCIÓN:

```
public static String largest(String[] array) {
      if (array!=null && array.length!=0)
            return largest(array, 0, array.length - 1);
      else
            return null;
}

public static String largest(String[] array, int start, int end)
{
      if (start > end ) return “”;


      if (start == end)
            return array[start]
;
      int m = (start + end) / 2;
      String left = largest(array, start, m);
      String right = largest(array, m + 1, end);
      return (left.length() > right.length()) ? left : right;
}
```

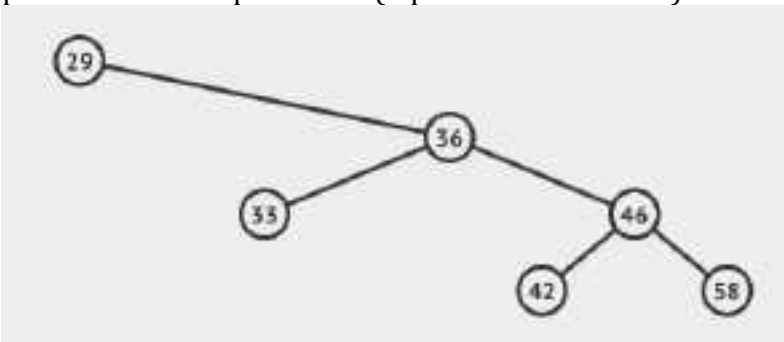
Problema 4 - (2 puntos): Escribe un método estático que tome dos árboles binarios de búsqueda como parámetros de entrada y devuelva una lista doblemente enlazada de las claves que ambos árboles tienen en común. Si una clave está en ambos árboles, debes agregarla solo una vez.

tree1	tree2
	
common (tree1, tree2)-> list: 12<->14<->17<->23<->45	

SOLUCIÓN:

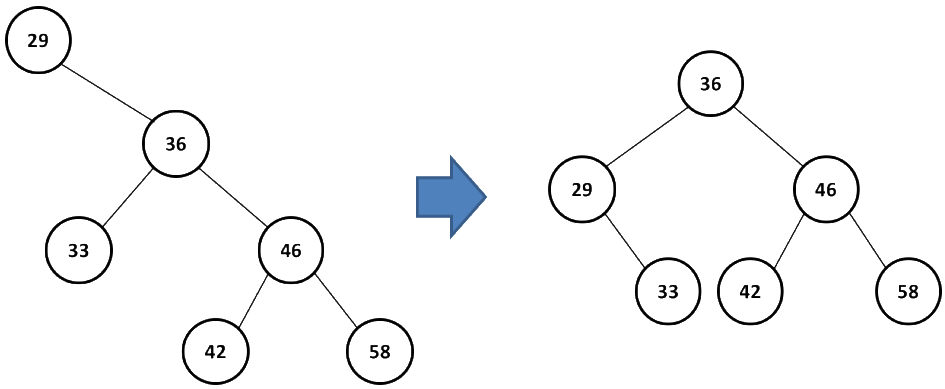
```
public static DList Merge(BSTree tree1, BSTree tree2) {
      DList l1 = tree1.getInOrderList();
      DList l2 = tree2.getInOrderList();

      DList merge=mergeList(l1,l2)
}

public DList getInOrderList() {
      DList l=new DList();
      if (root!= null) {
            getInOrderList(root,l);
      }
      return l;
}

public static void getInOrderList(BSTNode node, DList l) {
      if (node.left!=null) getInOrderList(node.left, l);
      l.addLast(node.elem);
      if (node.right!=null) getInOrderList(node.right, l);
}


public static DList mergeList(DList l1, DList l2) {
      DList merge=new DList();
      if (l1==null || l2== null) return merge;

      for (int i=0; i<l1.getSize(); i++) {
            int elem=l1.getAt(i);
            if (l2.contains(elem))
                  merge.addLast(elem);
      }
      return merge;
}
```

Problema 5 (1 punto): Dado el siguiente árbol binario de búsqueda, dibuja paso a paso sus equilibrados para que sea; 1) AVL (equilibrio de altura), y 2) perfectamente equilibrado (equilibrio en tamaño).

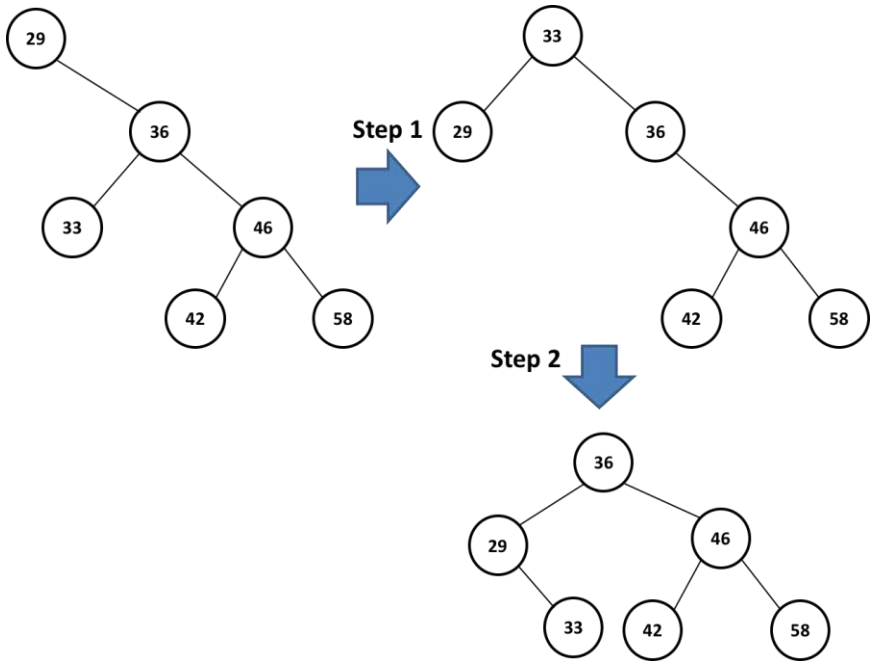


SOLUCIÓN:

1) AVL (equilibrio de altura)



2) Perfectamente equilibrado (equilibrio en tamaño)



Problema 6 (2 puntos): supongamos que una ciudad tiene una red de metro con 100 estaciones.

1. (0.25) ¿Qué estructura de datos es la más adecuada para representar la red de metro (estaciones y sus conexiones)?
2. (0.25) ¿Cuál es la cantidad máxima de conexiones posibles (directas) entre las estaciones?
3. (0.25) Si el número de conexiones es mucho menor que el número máximo posible, ¿cuál es la mejor implementación? Explíquelo.
4. (0.25) Para implementar dicha estructura, escribe la cabecera de la clase, los atributos y un método constructor.
5. (0.40) Escriba un método **checkConnection** que tome dos cadenas (que son nombres de estaciones) y devuelva **true** si hay una conexión directa entre ellas, o **false** si no hay esa conexión ¿Cuál es su complejidad?
6. (0.60) Escriba un método **checkPath** que tome dos cadenas (que son nombres de estaciones) y devuelva **true** si hay una ruta entre ambas estaciones o **false** si no hay esa ruta.

   Nota: En este problema, puedes utilizar clases Java tales como LinkedList <String> o ArrayList <String> para implementar su solución. Si lo prefieres, puedes usar nuestras clases SList, DList, SQueue o Stack (no necesitas implementarlas, solo debes recordar el nombre de sus métodos y debes saber cómo usarlos).

SOLUCIÓN:

1) Un grafo es una estructura de datos adecuada para representar las estaciones y las conexiones entre esas estaciones. El grafo sería no ponderado (las conexiones no tienen ningún peso) y NO dirigido (ya que las conexiones son simétricas).
2) La cantidad máxima de conexiones posibles (directas) entre las estaciones son 10000 conexiones.
3) Respecto a la implementación, el uso de una lista de adyacencias sería la opción óptima tanto en complejidad espacial (sólo usas la memoria necesaria) y temporal (las mayorías de las operaciones tendrían complejidad linear).

4)

```
public class Metro {
public String[] stations;
public LinkedList<String>[] listMetro;
public int MAX_STATIONS = 100;

public Metro(String[] stations) {
this.stations = new String[MAX_STATIONS];
listMetro = new LinkedList[MAX_STATIONS];
for (int i = 0; i < stations.length && i < MAX_STATIONS; i++) {
            this.stations[i] = stations[i];
            listMetro[i] = new LinkedList<String>();
      }
}
```

5)

```
public boolean checkConnection(String station1, String station2)
{
	boolean check = false;
	for (String station : listMetro[getIndex(station1)])
		if(station.equals(station2))
			check = true;
	return check;
}
```

la complejidad es O(n)

6)

```
public boolean checkPath(String station1, String station2) {
	Queue<String> queue = new LinkedList<String>();
	boolean[] visited = new boolean[stations.length];
	queue.add(station1);
	while (!queue.isEmpty()) {
		String station = queue.poll();
		System.out.print(station + "->");
		int idx = getIndex(station);
		visited[idx] = true;
		for(String connection : listMetro[idx]) {
			if (!visited[getIndex(connection)]) {
				queue.add(connection);
				if (connection.equals(station2)) {
					System.out.println(connection);
					return true;
				}
			}
		}
	}
	System.out.println("...No path!");
	return false;
}
```