Estructura de Datos y Algoritmos

31 de mayo de 2019

2018-2019

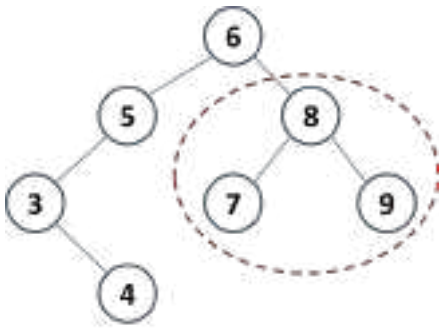
Nombre y Apellido: **Grupo:**

Problema 1 (2 puntos) – Dadas las clases BSNode (nodo binario de búsqueda) y BSTree (árbol binario de búsqueda) estudiadas durante el curso, implementa los dos métodos siguientes:

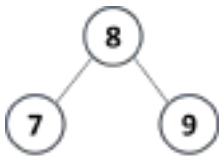
A) (1) En la clase BSTree, el método no estático **getLeaves().** Dicho método devuelve una lista doblemente enlazada (DList) con las claves (**key**) de los **nodos hojas** del árbol. La lista debe estar ordenada de forma descendente. No está permitido utilizar ningún algoritmo de ordenación para ordenar la lista. Se recomienda que el método sea recursivo. ¿Cuál es la complejidad del método? Justifica la respuesta.

B) (1) También en la clase BSTree, el método no estático **isIncluded** recibe un objeto de un árbol binario de búsqueda y devuelve true si el árbol de entrada está incluido en el árbol invocador, y false en caso contrario.

Ejemplo: en el siguiente ejemplo el método devuelva true



Objeto invocador



Objeto de entrada

Nota: En el apartado A, puede utilizar la clase DList. Si utilizas algún método de la clase DList, deberás incluir su implementación.

SOLUCIÓN: Problema 1

A)

```
static DList getLeaves()  {

     if (this.root==null){ System.out.println("Tree is empty");

      return null; }

      DList dlist = new DList();

      return getLeaves(tree.root, dlist);
}

public static DList getLeaves(BSTNode node, DList dlist) {
      // TODO Auto-generated method stub
      if (node!=null) {
            getLeaves(node.left, dlist);
            if (node.left==null && node.right==null) {
                  // ADD FIRST
                  DNode newNode = new DNode(node.key);
                  newNode.next = dlist.header.next;
                  newNode.prev= dlist.header;
                  dlist.header.next.prev= newNode;
                  dlist.header.next = newNode;
                  dlist.size++;
            }
            getLeaves(node.right, dlist);
      }
      return dlist;
}
```

==========

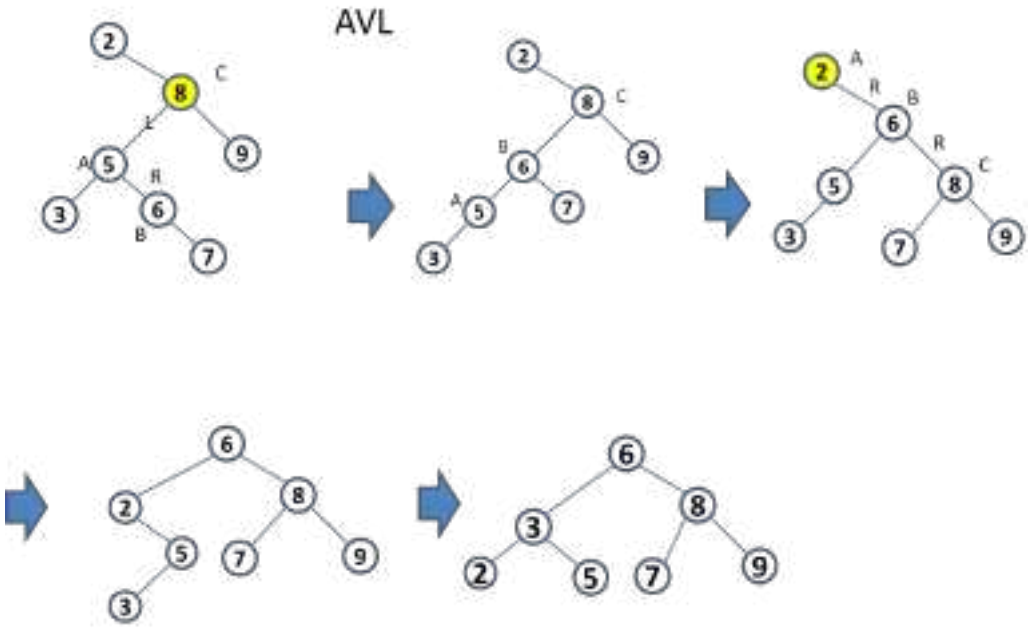
B)

```
public boolean  isIncluded(BSTree tree) {
      if (tree==null)
            return false;
      return isIncluded(this.root, tree.root);
}

public static boolean isIncluded(BSTNode n1,BSTNode n2) {

     if (n1 == null && n2 == null)

      return true;

     if (n1 != null && n2 != null){

      if (n1.key== n2.key)
                              return    (isIncluded(n1.left,    n2.left)    &&
isIncluded(n1.right, n2.right));

      else if(n1.key < n2.key){
                  return isIncluded(n1.right,n2);

      else if(n1.key > n2.key){
                        return isIncluded(n1.left,n2);
            }
      }
```

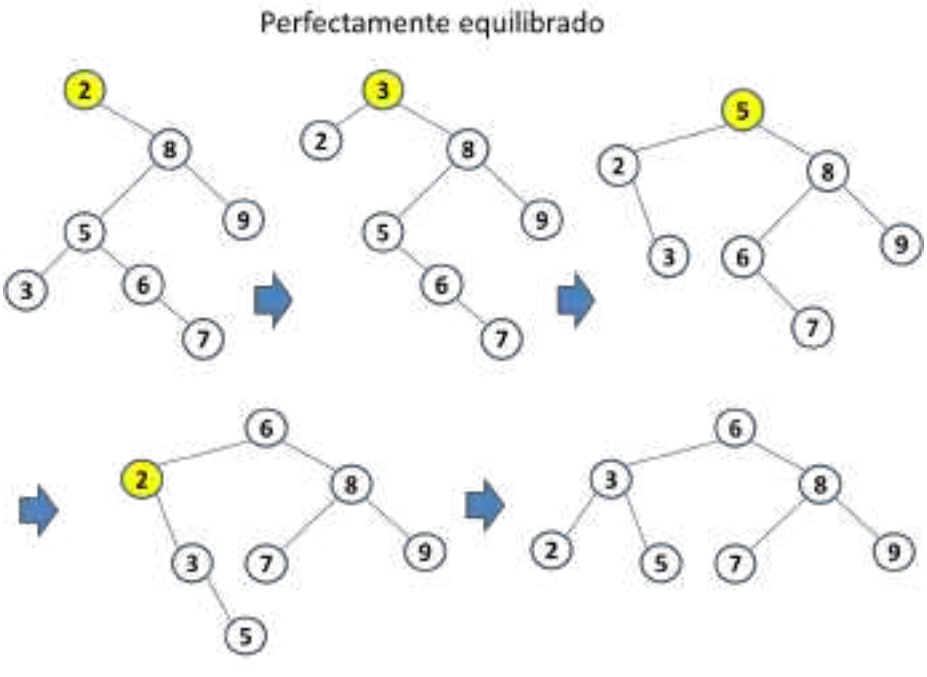
```
    return false;
}
```

Problema 2 (0,5 puntos) –
A) (0.25) Equilibra (paso a paso) el siguiente árbol para que satisfaga la propiedad AVL (equilibrio de altura).
B) (0.25) Equilibra (paso a paso) para que sea un árbol perfectamente equilibrado (equilibrio en tamaño).

SOLUCIÓN:



==========



Problema 3 (2,5 puntos) – Una pequeña compañía aérea de ámbito nacional necesita una aplicación que gestione su red de líneas aéreas. El número de aeropuertos en los que opera esta compañía se estima que no va a variar mucho. Actualmente, el número de aeropuertos es 50.

Cada aeropuerto está identificado por el nombre de la localidad en la que se encuentra, y existen conexiones entre dos aeropuertos, por ejemplo, una conexión entre Madrid e Ibiza, significa que existen vuelos que operan entre Madrid e Ibiza.

A) (0,25) ¿Qué estructura de datos es la más adecuada para representar la red de líneas aéreas (aeropuertos y sus conexiones)? ¿qué implementación consideras que es mejor?

B) (0,25) Para implementar dicha estructura, escribe la cabecera de la clase, los atributos y un método constructor.

```
public class NetworkAirlines  {



	public String[] citiesAirport;
	public int num;
	LinkedList<Adjacent>[] lstAdjacents;


	public NetworkAirlines(String[] citiesAirport) {
			this.citiesAirport=citiesAirport;
			this.num=citiesAirport.length;
			lstAdjacents=new LinkedList[num];
			for (int i=0; i<num; i++) {
				lstAdjacents[i]=new LinkedList<Adjacent>();
			}
	}
```

C) (0,25) Escribe un método **checkConnection** que tome dos atributos de tipo String, que son los nombres de los aeropuertos, y devuelva true si hay una conexión directa entre ellos, o false si no lo hay. ¿Cuál es la complejidad del método? Justifica la respuesta.

```
public Boolean checkConectionn(String cityAirport1, String
cityAirport2) {
		int index1=getIndex(cityAirport1);
		int index2=getIndex(cityAirport2);
		if (index1==-1) return null;
		if (index2==-1) return null;

		int sizeList=lstAdjacents[index1].size();
		if (sizeList==0) return null;

		for (int i=0; i<lstAdjacents[index1].size();i++) {
			int adj=lstAdjacents[index1].get(i).vertex;
			if (adj==index2) return true;
		}
		return false;
	}
```

D) (0,50) Un método, **addConnection**, que tome dos atributos de tipo String, representando los nombres de dos aeropuertos existentes y añada una conexión entre ambos. ¿Cuál es la complejidad del método? Justifica la respuesta.

```
private void addConnection(String cityAirport1, String cityAirport2)
{
                int i=getIndex(cityAirport1);
                int j=getIndex(cityAirport2);

                boolean contains=false;
                for (int k=0; k<lstAdjacents[i].size() &&
contains==false;k++) {
                      Adjacent obj=lstAdjacents[i].get(k);
                      if (obj.vertex==j) {
                            contains=true;
                      }
                }
                if (!contains) {
                      lstAdjacents[i].add(new Adjacent(j));
                }
          }
```

E) (0,50) Escribe un método **findConnections** que tome un String, nombre de un aeropuerto, y devuelva una lista con todos los aeropuertos que estén conectados directamente con este aeropuerto. ¿Cuál es la complejidad del método? Justifica la respuesta.

```
public LinkedList<String>  findConnection(String cityAirport) {
          int index=getIndex(cityAirport);
          LinkedList<String> lst = new LinkedList<String>();
          if (index!=-1) {

                for (int i=0; i<num;i++) {
                      for (int j=0; j<lstAdjacents[i].size();j++) {
                            int adj=lstAdjacents[i].get(j).vertex;
                            if (index==adj) {
                                  lst.add(checkVertex(i));
                            }
                      }
                }
          }
          return lst;
          }
```

F) (0,75) Un método, **getAllAirports**, que devuelva todos los aeropuertos del grafo mediante un recorrido en amplitud.

```
public void getAllCitiesAirports() {
            System.out.println("breadth traverse of the graph:");

            boolean visited[]=new boolean[this.citiesAirport.length];

            for (int i=0;i<this.citiesAirport.length;i++) {
                  if (!visited[i]) {
                        getAllCitiesAirports(i,visited);
                  }
            }
            System.out.println();

      }



      protected void getAllCitiesAirports(int i, boolean[] visited) {
            Queue<Integer> q=new LinkedList<Integer>();
            q.add(i);
            while (!q.isEmpty()) {
                  int vertex=q.poll();
                  String cityAirport=this.checkVertex(vertex);
                  System.out.print(cityAirport+",");
                  visited[vertex]=true;

                  int[] adjacents=getAdjacents(vertex);
                  for(int adjVertex:adjacents) {
                              if (!visited[adjVertex] &&
!q.contains(adjVertex)) {
                                    q.add(adjVertex);
                              }
                  }
            }
      }
```

Nota: En este problema, puedes utilizar clases Java tales como LinkedList. Si lo prefieres, puedes usar nuestras clases SList, DList (no necesitas implementarlas, solo debes recordar el nombre de sus métodos y debes saber cómo usarlos)

Problema 4 (1 punto) – Implementar un método Java para ordenar una lista de palabras alfabéticamente. El método de ordenación debe basarse en el algoritmo mergesort. Se pueden crear métodos auxiliares adicionales según sea necesario. La cabecera del método debe ser el siguiente:

```
public static void orderWords(DList words){…
      }
```

Nota: Hay que utilizar una lista doble DList, no se puede utilizar LinkedList

Ejemplo:

- Input = {'sword','throne','game','end','dragon'}
- Output = {'dragon', 'end','game','sword','throne'}

SOLUCIÓN:

```
public static void orderWords(DList words){

            if((words != null)&&(words.size > 0))
                  orderWords(words,0,words.size - 1);
            else{
                  System.out.println("[orderWords] Error. Input parameter
is null or empty)");
            }

      }

      private static void orderWords(DList words, int start, int end){

            if(start != end) {
                  int imiddle = (start + end)/2;
                  orderWords(words,start,imiddle);
                  orderWords(words,imiddle + 1,end);

                  mergeWords(words,start,imiddle,end);
            }

      }

      private static void mergeWords(DList words, int left, int middle, int
right) {
            //sizes of two sublists to be merged
          int n1 = middle - left + 1;
          int n2 = right - middle;

          /* Create temp lists and copy */
          DList list1 = new DList();
          DList list2 = new DList();

          for (int i=0; i<n1; ++i) list1.addLast(words.getAt(left+i));
          for (int j=0; j<n2; ++j) list2.addLast(words.getAt(middle+1+j));

          /* Merge the temp lists */
          // Initial indexes of first and second subarrays
          int i = 0, j = 0;
          // Initial index of merged subarry array
          int k = left;
          while (i < n1 && j < n2) {
              if (list1.getAt(i).compareTo(list2.getAt(j)) <= 0){
                  words.removeAt(k);
                  words.insertAt(k, list1.getAt(i));
                  i++;
              } else {
                  words.removeAt(k);
                  words.insertAt(k, list2.getAt(j));
                  j++;
              }
              k++;
          }

          /* Copy remaining elements of list1 if any */
          while (i < n1)    {
            words.removeAt(k);
              words.insertAt(k, list1.getAt(i));
              i++;
```

```
            k++;
        }

        /* Copy remaining elements of list2 if any */
        while (j < n2)    {
          words.removeAt(k);
            words.insertAt(k, list2.getAt(j));
            j++;
            k++;
        }
    }
```