**Práctico 2.**

**INF310-SX. Estructuras de Datos II. Gestión 2-2019**

***Todos los problemas de este práctico, deben ser implementados en computadora.***

|  |  |
| --- | --- |
| 1. En la class ArbolM escriba el método  public boolean gemelos(int a, int b)  El cual devuelva **true** si los datas a y b están alojados en un **mismo** nodo del árbol. Si a o b no existen, o a y b están alojados en nodos distintos, devuelve **false**.  2. En la class ArbolM escriba el método  **public boolean hnoCercano(int a, int b)**  //Por comodidad, asuma que a < b  el cual devuelve true si el data a es hermano cercano del data b. Si a o b no están en el árbol; o a y b no son hermanos cercanos, devuelve false.  Dos datas son *hermanos cercanos* si no comparten el mismo nodo, pero en el nodo padre tienen un **data** padre en común P.  […| P |…]  […| a |…]      […| b |…]      3. En la class ArbolM escriba el método  public int cantDatas()  El cual devuelva la cantidad de datas (números) alojados en el árbol. Recuerde que en las hojas, pueden existir datas vacías.  *La class NodoM tiene los métodos cantDatasUsadas y cantDatasVacias, que le podrían servir.*  4. En la class ArbolM escriba el método  public int nivel(int x)  El cual devuelva el nivel donde se encuentra el data x. Si x no existe en el árbol, devuelve 0.  Recuerde que la raíz está en el primer nivel, los hijos de ésta en el segundo nivel y así sucesivamente.  5. En la class ArbolM escriba el método  public void podar()  El cual elimine **todas** las hojas del árbol.  (Si el árbol es vacío, “no pasa nada”. Si todo el árbol es una hoja, es decir la Raíz es una hoja, entonces Raiz=null). | 6. En la class Grafo (no-dirigido), escriba el método  public boolean hayCamino(int u, int v)    el cual devuelva true sii existe un camino desde u hasta v.  7. En la class Grafo (no-dirigido), escriba el método  public boolean isLineal()    el cual devuelva true sii el grafo es lineal. Un grafo es *lineal* sii   1. El Grafo tiene a lo sumo un vértice, o 2. *Si el grafo tiene dos o más vértices:* El grafo es conexo, dos vértices tienen exactamente una arista, mientras que los demás (si existen) tienen exactamente dos aristas.   (a los vértices que tienen una sola arista, le dicen “extremos”)  Grafo Q    Grafo P    Grafo G    G.isLineal() = true  //0 y 2 son los extremos  P.isLineal() = false    Q.isLineal() = true  //0 y 1 son los extremos    *Los grafos lineales no tienen lazos*  8. En la class Grafo (dirigido), escriba usted el procedimiento  public void dfsInv(int v)    el cual realiza el DFS(v), pero yendo en contrasentido (contraflecha).  Por ejemplo:  dfsInv(0) ⇒ 0, 3, 1, 4  dfsInv(1) ⇒ 1  dfsInv(2) ⇒ 2  dfsInv(3) ⇒ 3, 1  dfsInv(4) ⇒ 4  dfsInv(5) ⇒ 5, 2  dfsInv(6) ⇒ 6, 2 |

9. En la class Grafo (dirigido), escriba el método

**public void delLazos(int u, int v)**

el cual elimine los lazos que se encuentran al transitar desde el vértice u hasta el vértice v, usando el DFS. Si no hay camino desde u hasta v, éste algoritmo no elimina ningún lazo.

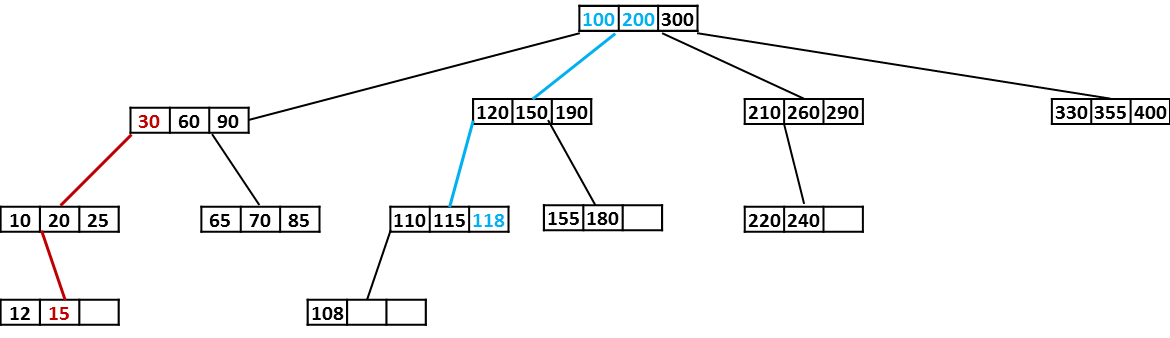
*Por comodidad, asuma que u es diferente de v.*

10. Se dice que un data x, es un super-parent de un data z, si z se encuentra en la rama izquierda o en la rama derecha de x (i.e. si descendemos por el hijo-izquierdo o el hijo-derecho de x, y encontramos a z, entonces x es un super-parent de z)

En la class **ArbolM**, escriba la función

public boolean superParent(int x, int z)

la cual devuelva true, si x es un superParent de z. Si x o z no existen o x no es un super-parent de z, devuelve false. Por ejemplo:



superParent(30, 15) =true //Porque partiendo del hijo izquierdo del 30, podemos llegar al 15

superParent(100,118)=true //Porque partiendo del hijo derecho del 100, podemos llegar al 118

superParent(200,118)=true //Porque partiendo del hijo izquierdo del 200, podemos llegar al 118

superParent(15,30)=false //Porque yendo por el hijo izq o por el hijo der del 15, no llegamos al 30

superParent(100,290)=false //Porque yendo por el hijo izq o por el hijo der del 100, no llegamos al 290

superParent(200,290)=true //Porque yendo por el hijo derecho del 200, llegamos al 290

superParent(330,400)=false //Porque yendo por el hijo izq o por el hijo der del 330, no llegamos al 400 (330 está en el mismo nodo del 400).

superParent(**600**,150)=false //600 no existe.

superParent(30, **8**) =false //8 no existe.

11. Se dice que un data **t**, es **tío** de un data **s**, si y solo si, s está alojado en un nodo hijo, hermano del nodo donde está t.

[…| |…]

[…| |…]

[…| **t** |…]

[…| **s** |…]

[…| |…]

[…| **t** |…]

[…| |…]

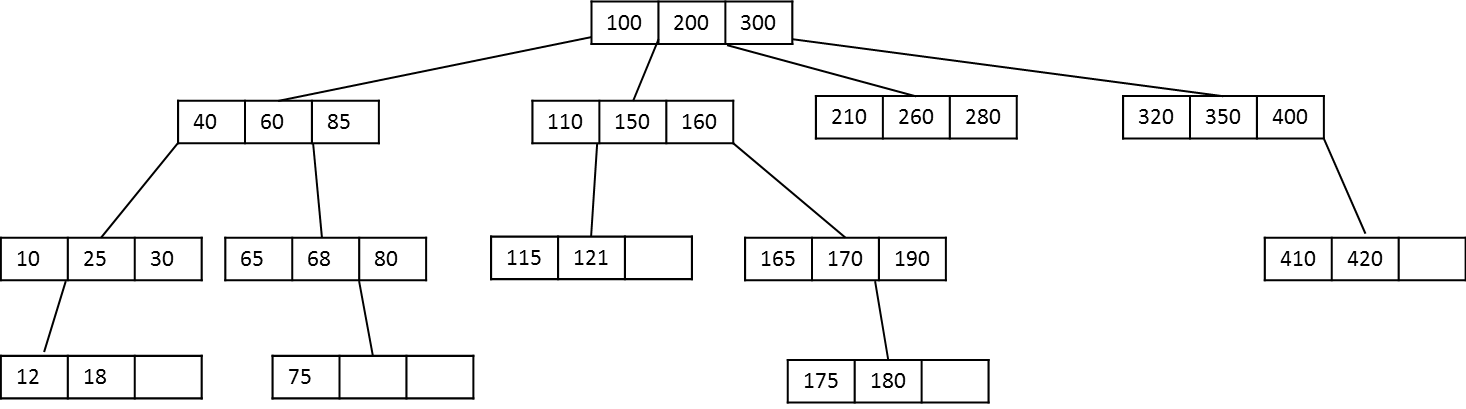
[…| **s** |…]

En la class **ArbolM**, escriba la función:

public boolean tio(int t, int s)

la cual devuelva true, si t es un tío de s. Si t o s no existen o t no es un tío de s, devuelve false.

Por ejemplo:



tio(10, 75)=true //Porque el nodo padre del 75, es hermano del nodo donde está el 10

tio(75, 10)=false

tio(500,210)=false //El 500 no existe.

tio(350,80)=true

tio(260,195)=false //195 no existe

tio(100,85)=false //100 es padre del 85, no su tío.

12. En la class Grafo (dirigido), escriba usted un DFS especial, con la función

public String path(int a)

la cual partiendo de un vértice a, continúa con el vértice adyacente (no-marcado) cuyo número de vértice sea el menor. Finalmente, devuelve la ruta efectuada.

Por ejemplo:

path(0)= “0 2”

path(1)= “1 6 2”

path(2)= “2”

path(3)= “3 1 6 2”

path(4)= “4 0 2”

path(5)= “5 3 1 6 2”

path(6)= “6 1”

path(7)= “7”

path(8)= “8 7”

**Tip:** *Cada vez que usted visite un vértice u, márquelo usando el método private marcar(u)*

**Explicando path(3)**

Visitamos el vértice 3 (y lo marcamos). Observamos que 3 tiene como adyacentes no marcados a: 1, 4 y 6. Como el **1** es el vértice menor, entonces visitamos el 1.

Visitamos el vértice 1 (y lo marcamos). El vértice 1 tiene como **único** adyacente no marcado al 6. No tenemos más opción que visitar el **6**.

Visitamos el vértice 6 (y lo marcamos). El vértice 6 tiene un adyacente no marcado: el **2** (el 1 es adyacente del 6, pero está marcado)

Visitamos el vértice 2 (y lo marcamos). El vértice 2 NO tiene adyacentes no marcados. Finalizamos.

La ruta seguida fue: 3 ---> 1 ---> 6 ---> 2

**Recuerde:** *Un DFS jamás visita un vértice más de una vez (por eso se usa el marcado de vértices).*