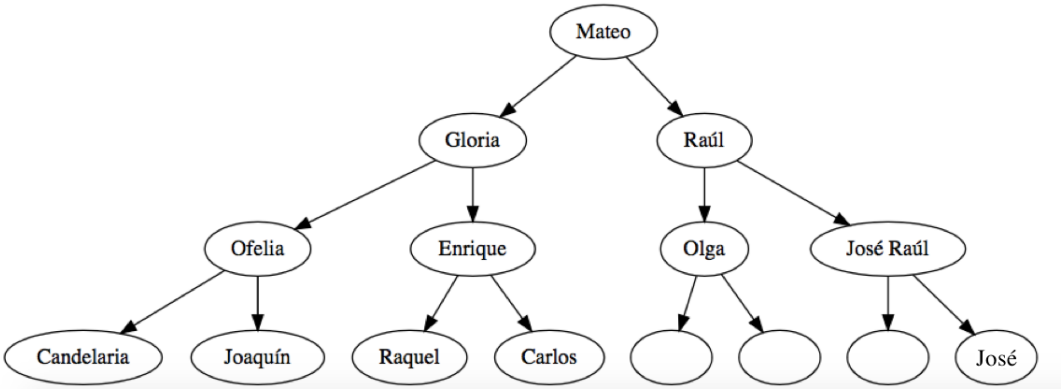
Laboratorio Nro. 5: Árboles binarios

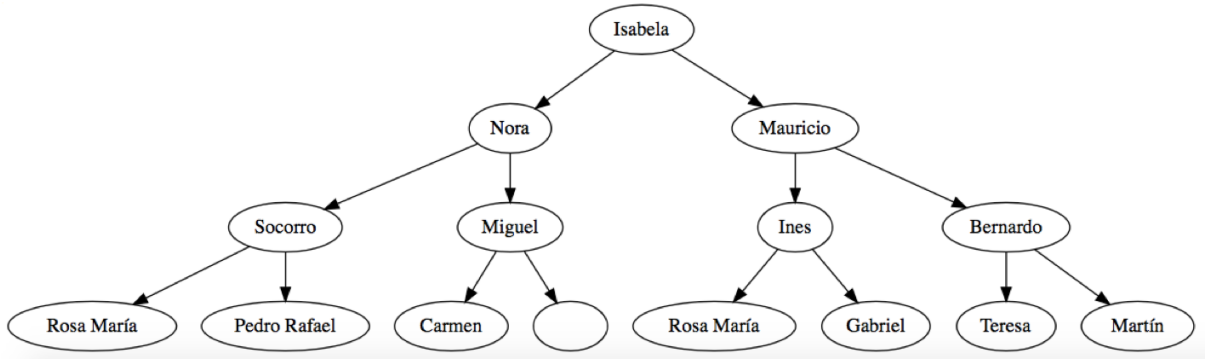
Isabela Muriel Roldán
Universidad Eafit
Medellín, Colombia
imurielr@eafit.edu.co

Mateo Flórez Restrepo
Universidad Eafit
Medellín, Colombia
mflorezr@eafit.edu.co

3) Simulacro de preguntas de sustentación de Proyectos

1.





DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

2. Al utilizar la implementación de java de los arboles binarios se obtiene una complejidad O(log(n)) para los métodos de inserción y búsqueda en el mejor de los casos *(como se muestra en la figura 1)*, lo cual es una complejidad muy baja y esto favorece a que el árbol se genere de forma muy rápida.

Figura 1: Complejidad de un árbol binario de búsqueda

3. **Funcionamiento del ejercicio en línea 2.1**

El ejercicio en línea consiste en dar como entrada los valores de los nodos de un árbol binario en la forma en la que se recorrería en pre-orden (Raíz – Izquierda - Derecha) y el programa devolverá como salida los mismos valores, pero en la forma como se haría en un recorrido en pos-orden (Izquierda – Derecha- Raíz).

El programa cuando se ejecute le pedirá al usuario mediante scanner el número N de nodos o datos que se desea ingresar, y posterior a esto se ingresaran los datos numéricos que contienen los nodos de un árbol binario en forma de recorrido pre-orden, según la cantidad N de datos que el usuario haya ingresado anteriormente. El código lo que hará es leer la cantidad de datos y sus valores, y después creará un nuevo árbol binario de búsqueda, reconstruyendo el árbol original, e insertará cada nodo y su valor, según los datos ingresados y su orden correspondiente (mayores a la raíz, a la derecha, menores a la raíz, a la izquierda), mediante el método "*hacerArbol*" quien contiene el método "*insertar*". Luego de esto hará un recorrido en pos-orden en el árbol por medio del método "*postorden*", quien recorre cada subárbol o nodo de izquierda a derecha y luego a raíz, a su vez imprimiendo el valor que contiene cada uno. Dando como salida los valores de los nodos del árbol binario de búsqueda en forma de recorrido pos-orden.

4. Complejidad del ejercicio 2.1

Se calculará la complejidad por cada método.

```
private static boolean insertar(Node nodo, int n){
    if(nodo.data == n){                    //C1
       return false;                       //C2
    }
    if(nodo.data > n){                     //C3
```

```
            if(nodo.left == null){                    //C4
               nodo.left = new Node(n);               //C5
               return true;                           //C6
            }
            else{
               return insertar(nodo.left, n);         //T(n/2)
            }
         }
         if(nodo.data < n){                           //C7
             if(nodo.right == null){                  //C8
               nodo.right = new Node(n);              //C9
               return true;                           //C10
            }
            else{
               return insertar(nodo.right, n);        //T(n/2)
            }
         }
         return false;                                //C11
      }
```

T(n) = C1+C2 si n==nodo.data

T(n) = C3+C4+C5+C6 si n< nodo.data & nodo.left ==null
T(n) = C3+ T(n/2) si n < nodo.data & nodo.left != null

T(n) = C7+C8+C9+C10 si n> nodo.data & nodo.right==null
T(n) = C3+ T(n/2) si n > nodo.data & nodo.right != null

Resolver por recurrencia, Wolfram:

En el peor de los casos T(n) = C+ T(n/2)
T(n) = C'+C log(n)/log(2) => C'+ C*log2(n)
T(n) = O(C'+C* log2(n))
T(n) = O(C*log2(n))
T(n) = O(log(n))

```
public static  void insertar(int n){
      if(root == null){
         root = new Node(n);
      }
```

```
        insertar(root, n);
    }
```

Este método solo llama al auxiliar de insertar para realizar la función, por ende, su complejidad es la misma O(log(n)).

```
public static void hacerArbol(int n){
        System.out.print("Introduzca "+ n+ " enteros:\n ");    //C1
        for(int i =0; i<n; ++i){                              //C2*n
            Scanner sc = new Scanner (System.in);             //C3*n
            int m = sc.nextInt();                             //C4*n
            insertar(m);                                      //log(n)*n

        }
    }
```

T(n)= C1+(C2+C3+C4+log(n))*n
T(n)= O(C'+(C"+log(n))*n)
T(n)= O(C'+C"n+nlog(n))
T(n)= O(nlog(n))

```
  private static void auxPostorden(Node nodo){
      if (nodo == null){                                  //C1
          return ;                                        //C2
        }
        auxPostorden(nodo.left);                          //T(n/2)
        auxPostorden(nodo.right);                         //T(n/2)
        System.out.println(nodo.data+ " ");               //C3
    }
```

T(n) = C1+C2 si nodo == null
T(n) = T(n/2)+T(n/2)+C3, de lo contrario

En el peor de los casos:
T(n) = T(n/2)+T(n/2)+C3 => 2T(n/2)+C
T(n) = T(n)+C
T(n) = O(n)

```
public static void postorden(){
        auxPostorden(root);
```

}

Este método al igual que insertar solo hace un llamado al auxiliar de postorden para cumplir la función, por lo tanto, conserva la complejidad del auxiliar: O(n).

Método	Complejidad
Insertar	O(log n)
hacerArbol	O(n log n)
postorder	O(n)

Tabla 1. Resumen de complejidades.

5. Variables n y m: En el momento de calcular la complejidad de un algoritmo solo se utiliza n cuando solo hay una entrada, pero cuando hay dos entradas, por ejemplo, en dos ciclos anidados en los que se evalúan dos cosas diferentes se utiliza la m para representar la segunda entrada

4) Simulacro de Parcial

1. 1.1. *altura(raiz.izq)*
   2.2. *altura(raiz.der)*
2. *c*
3. 3.1. *false*
   3.2. *a.dato*
   3.3. *a.izq, suma - a.dato*
   3.4. *a.der, suma - a.dato*
4. 4.1. *c*
   4.2. *a*
   4.3. *d*
   4.4. *a*
5. 5.1. *if (p.dato == toInsert)*
   5.2. *if (toInsert > p.dato)*