

UNIVERSIDAD AUTONOMA GABRIEL RENE MORENO

*FACULTAD DE INGENIERIA Y CIENCIAS DE LA
COMPUTACION Y TELECOMUNICACIONES*

ESTRUCTURA DE DATOS II

CONTENIDO:

TAREA-1. ARBOLES BINARIOS DE BÚSQUEDA.

PORCENTAJE TERMINADO: 100%

Grupo 14

Garcia Taboada Brayan Albaro

Fecha de presentación : Lunes , 06 de mayo de 2024

Fecha Presentada : Domingo , 05 de mayo de 2024

Días de Atraso : 0

ÁRBOLES BINARIOS DE BÚSQUEDA.

TRABAJO INDIVIDUAL.

Sea A1, un Árbol Binario de Búsqueda. Implementar los siguientes métodos:

1. **A1.generarElem(n, a, b)** : Método que genera n elementos aleatorios enteros diferentes entre a y b inclusive.
2. **A1.insertar(x)** : Método que inserta el elemento x, en el árbol A1 en su lugar correspondiente.
3. **A1.preOrden()** : Método que muestra los elementos del árbol A1 en preOrden.
4. **A1.inOrden()** : Método que muestra los elementos del árbol A1 en inOrden.
5. **A1.postOrden()** : Método que muestra los elementos del árbol A1 en postOrden.
6. **A1.desc()**: Método que muestra los elementos del árbol A1 de mayor a menor.
7. **A1.seEncuentra(x)** : Métodos lógico que devuelve True, si el elemento x, se encuentra en el árbol A1.
8. **A1.cantidad()** : Método que devuelve la cantidad de nodos del árbol A1.
9. **A1.suma()** : Método que devuelve la suma de los elementos del árbol A1.
10. **A1.menor()** : Método que devuelve el elemento menor del árbol A1.
11. **A1.mayor()** : Método que devuelve el elemento mayor del árbol A1.
12. **A1.mostrarTerm()**: Método que muestra los elementos de los nodos terminales del árbol A1. Mostrar los elementos de menor a mayor.
13. **A1.cantidadTerm()**: Método que devuelve la cantidad de nodos terminales del árbol A1.
14. **A1.lineal()** : Método lógico que devuelve True, si el árbol A1 es un árbol degenerado o llineal. (Se puede dar cuando se genera el árbol con una secuencia ordenada de elementos)
15. **A1.inmediatoSup(x)** : Método que devuelve el elemento inmediato superior a x, si x se encuentra en A1, caso contrario devuelve el mismo elemento.
16. **A1.inmediatoInf(x)** : Método que devuelve el elemento inmediato inferior a x, si x se encuentra en A1, caso contrario devuelve el mismo elemento.

17. Implementar al menos 5 Ejercicios adicionales cualesquiera, de consultas sobre uno o más árboles binarios de búsqueda. Citar fuentes./**

*

* @author braya

*/

```
public class Arbol {
```

```
    public Nodo raiz;
```

```
    public Arbol() {
```

```
        this.raiz = null;
```

```
    }
```

//1. A1.generarElem(n, a, b) : Método que genera n elementos aleatorios enteros diferentes entre a y b inclusive.

```
    public void generarElem(int n, int a, int b) {
```

```
        for (int i = 0; i < n; i++) {
```

```
            insertar((int) Math.floor(a + Math.random() * (b - a)));
```

```
        }
```

```
    }
```

//2. A1.insertar(x) : Método que inserta el elemento x, en el árbol A1 en su lugar correspondiente.

```
    public void insertar(int x) {
```

```
        raiz = insertar(x, raiz);
```

```
    }
```

```
    private Nodo insertar(int x, Nodo p) {
```

```
        if (p == null) {
```

```
            return new Nodo(x);
```

```
        }
```

```
        if (x < p.elem) {
```

```

        p.izq = insertar(x, p.izq);
    } else {
        p.der = insertar(x, p.der);
    }
    return p;
}

```

//3. A1.preOrden() : Método que muestra los elementos del árbol A1 en preOrden.

```

public void preOrden() {
    preOrden(raiz);
}

```

```

private void preOrden(Nodo p) {
    if (p == null) {
        return;
    }
    System.out.println(p.elem);
    preOrden(p.izq);
    preOrden(p.der);
}

```

//4. A1.inOrden() : Método que muestra los elementos del árbol A1 en inOrden.

```

public void inOrden() {
    inOrden(raiz);
}

```

```

private void inOrden(Nodo p) {
    if (p == null) {
        return;
    }
}

```

```
        inOrden(p.izq);  
        System.out.println(p.elem);  
        inOrden(p.der);  
    }  
}
```

//5. A1.postOrden() : Método que muestra los elementos del árbol A1 en postOrden.

```
public void postOrden() {  
    postOrden(raiz);  
}
```

```
private void postOrden(Nodo p) {  
    if (p == null) {  
        return;  
    }  
    postOrden(p.izq);  
    postOrden(p.der);  
    System.out.println(p.elem);  
}
```

//6. A1.desc(): Método que muestra los elementos del árbol A1 de mayor a menor.

```
public void desc() {  
    desc(raiz);  
}
```

```
private void desc(Nodo p) {  
    if (p == null) {  
        return;  
    }  
    desc(p.der);  
}
```

```
System.out.println(p.elem);  
  
desc(p.izq);  
  
}
```

//7. A1.seEncuentra(x) : Métodos lógico que devuelve True, si el elemento x, se encuentra en el árbol A1.

```
public boolean seEncuentra(int x) {  
    return seEncuentra(x, raiz);  
}
```

```
private boolean seEncuentra(int x, Nodo p) {  
    if (p == null) {  
        return false;  
    }  
    if (x == p.elem) {  
        return true;  
    }  
    if (x < p.elem) {  
        return seEncuentra(x, p.izq);  
    } else {  
        return seEncuentra(x, p.der);  
    }  
}
```

//8. A1.cantidad() : Método que devuelve la cantidad de nodos del árbol A1.

```
public int cantidad() {  
    return cantidad(raiz);  
}
```

```
private int cantidad(Nodo p) {  
    return p == null ? 0 : cantidad(p.izq) + cantidad(p.der) + 1;  
}
```

//9. A1.suma() : Método que devuelve la suma de los elementos del árbol A1.

```
public int suma() {  
    return suma(raiz);  
}
```

```
private int suma(Nodo p) {  
    if (p == null) {  
        return 0;  
    } else {  
        return suma(p.izq) + suma(p.der) + p.elem;  
    }  
}
```

//10. A1.menor() : Método que devuelve el elemento menor del árbol A1.

```
public int menor() {  
    return menor(raiz);  
}
```

```
private int menor(Nodo p) {  
    if (p.izq == null) {  
        return p.elem;  
    } else {  
        return menor(p.izq);  
    }  
}
```

//11. A1.mayor() : Método que devuelve el elemento mayor del árbol A1.

```
public int mayor() {  
    return mayor(raiz);  
}
```

```
private int mayor(Nodo p) {  
    if (p.der == null) {  
        return p.elem;  
    } else {  
        return menor(p.der);  
    }  
}
```

//12. A1.mostrarTerm(): Método que muestra los elementos de los nodos terminales del árbol A1. Mostrar los elementos de menor a mayor.

```
public void mostrarTerm() {  
    mostrarTerm(raiz);  
}
```

```
private void mostrarTerm(Nodo p) {  
    if (p.izq == null && p.der == null) {  
        System.out.println(p.elem);  
    } else {  
        mostrarTerm(p.der);  
        mostrarTerm(p.izq);  
    }  
}
```

//13. A1.cantidadTerm(): Método que devuelve la cantidad de nodos terminales del árbol A1.

```
public int cantidadTerm() {  
    return cantidadTerm(raiz);  
}
```



```
private int cantidadTerm(Nodo p) {
    return (p.izq == null && p.der == null) ? 1 : cantidadTerm(p.der) + cantidadTerm(p.izq);
}
```

//14. A1.lineal() : Método lógico que devuelve True, si el árbol A1 es un árbol degenerado o lineal. (Se puede dar cuando se genera el árbol con una secuencia ordenada de elementos)

```
public boolean lineal() {
    return !(linealDer(raiz.der) && lineallzq(raiz.izq));
}
```

```
private boolean linealDer(Nodo p) {
    if (p == null) {
        return true;
    }
    if (p.izq != null) {
        return false;
    }
    return linealDer(p.der);
}
```

```
private boolean lineallzq(Nodo p) {
    if (p == null) {
        return true;
    }
    if (p.der != null) {
        return false;
    }
    return lineallzq(p.izq);
}
```

//15. A1.inmediatoSup(x) : Método que devuelve el elemento inmediato superior a x, si x se encuentra en A1, caso contrario devuelve el mismo elemento.

```
public int inmediatoSup(int x) {  
    return seEncuentra(x) ? inmediatoSup(raiz, x) : x;  
}
```

```
private int inmediatoSup(Nodo p, int x) {  
    if (p.elem == x) {  
        return superior(p.der);  
    }  
    if (p.elem > x) {  
        return inmediatoSup(p.izq, x);  
    }  
    return inmediatoSup(p.der, x);  
}
```

```
private int superior(Nodo p) {  
    return p.izq == null ? p.elem : superior(p.izq);  
}
```

//16. A1.inmediatoInf(x) : Método que devuelve el elemento inmediato inferior a x, si x se encuentra en A1, caso contrario devuelve el mismo elemento.

```
public int inmediatoInf(int x) {  
    return seEncuentra(x) ? inmediatoInf(raiz, x) : x;  
}
```

```
private int inmediatoInf(Nodo p, int x) {  
    if (p.elem == x) {  
        return inferior(p.izq);  
    }  
    if (p.elem > x) {  
        return inmediatoInf(p.izq, x);  
    }  
}
```

```

    }
    return inmediatoInf(p.der, x);
}

```

```

private int inferior(Nodo p) {
    return p.der == null ? p.elem : inferior(p.der);
}

```

//17. Implementar al menos 5 Ejercicios adicionales cualesquiera, de consultas sobre uno o más árboles binarios de búsqueda. Citar fuentes.

```

public int size() {
    return size(raiz);
}

```

```

private int size(Nodo p) {
    if (p == null) {
        return 0;
    }
    int sizeXlq = size(p.izq);
    int sizeXDer = size(p.der);
    return sizeXDer + sizeXlq + 1;
}

```

```

public int altura() {
    return altura(raiz);
}

```

```

protected int altura(Nodo p) {
    if (p != null) {
        return 0;
    }
}

```

```

    }

    int alturaXlq = altura(p.izq);
    int alturaXDer = altura(p.der);
    return alturaXlq > alturaXDer ? alturaXlq + 1 : alturaXDer + 1;

}

public int nivel() {
    if(this.altura()==0){
        return 0;
    }
    return this.altura()-1;
}

public String imprimirArbol() {
    String espacio = "";
    return imprimir(raiz, espacio);
}

protected String imprimir(Nodo p, String espacio) {
    String arbol = "";
    if (p != null) {
        arbol = espacio + "L" + p.elem + "\n";
        espacio += " ";
        arbol += imprimir(p.izq, espacio);

        arbol += imprimir(p.der, espacio);
    }
    return arbol;
}

public int hijosEnNivelN(int n) {

```

```
        return hijosEnNivelN(n, raiz);  
    }  
}
```

```
private int hijosEnNivelN(int n, Nodo p) {  
    if (p == null || (n < 0)) {  
        return 0;  
    }  
    int c = hijosEnNivelN(n - 1, p.der)  
        + hijosEnNivelN(n - 1, p.izq);  
    if (p.izq != null && (n == 0)) {  
        c++;  
    }  
    return c;  
}
```

```
public int sumaPares() {  
    return sumaPares(raiz);  
}
```

```
private int sumaPares(Nodo p) {  
    if (p == null) {  
        return 0;  
    } else {  
        return p.elem % 2 == 0  
            ? sumaPares(p.izq) + suma(p.der) + p.elem  
            : sumaPares(p.izq) + suma(p.der);  
    }  
}
```

```
public int cantidadPares() {
```

```

        return cantidadPares(raiz);

    }

    private int cantidadPares(Nodo p) {
        if (p == null) {
            return 0;
        }
        return (p.elem % 2 == 0) ? cantidadPares(p.der) + cantidadPares(p.izq) + 1
            : cantidadPares(p.der) + cantidadPares(p.izq);
    }
}

class Nodo {

    public Nodo izq;
    public Nodo der;
    public int elem;

    public Nodo(int elem) {
        this.elem = elem;
        izq = der = null;
    }

}

```