

Los usos de árboles son multiples. Además de los árboles de expresiones, usar una estructura de árbol ordenada para buscar elementos suena interesante:

De la **lista** toma lo mejor: **encadenar** los elementos con punteros y no tener que alocar zona contigua.

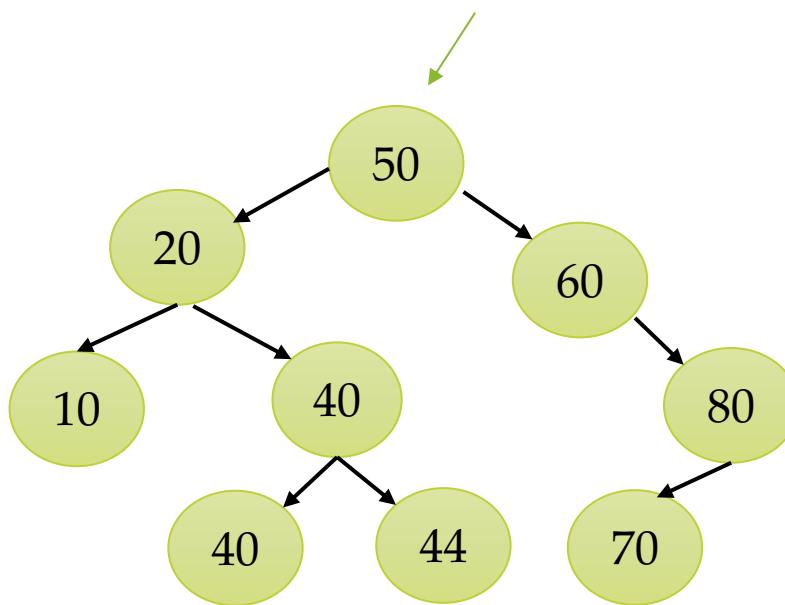
De los **arreglos ordenados** toma lo mejor: la posibilidad de aplicar **búsqueda binaria** (es un árbol binario...)

Árbol Binario de Búsqueda u árbol binario ordenado (Binary Search Tree o BST)

Es un árbol binario donde cada nodo no vacío cumple la siguiente condición: todos los datos de su **subárbol izquierdo** son menores o iguales que su dato, y todos los datos de su **subárbol derecho** son mayores que su dato.

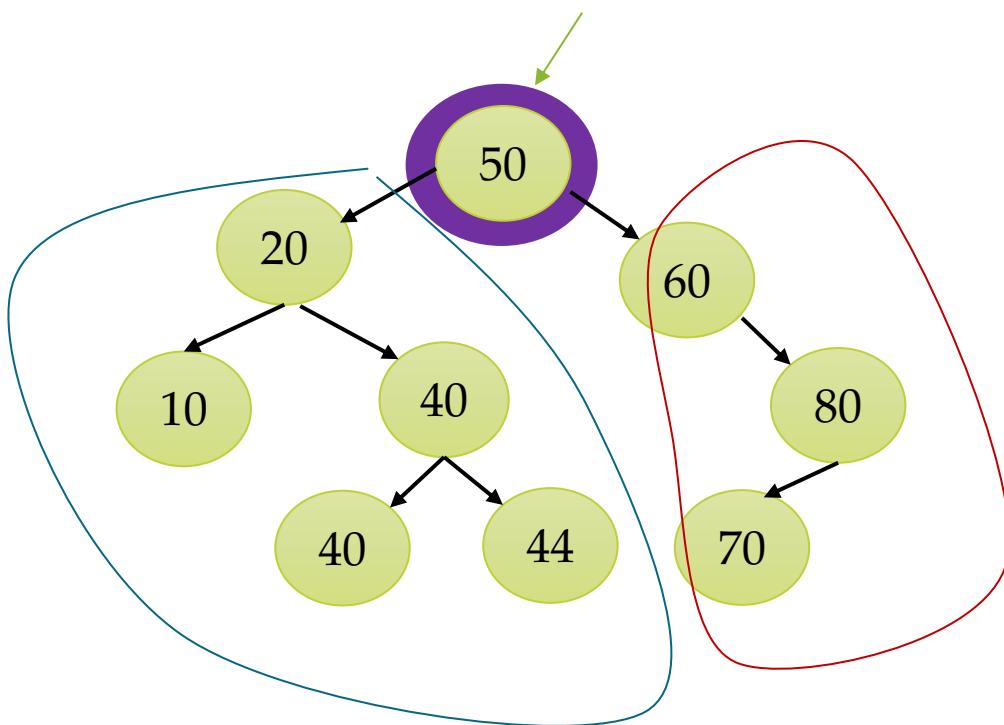
BST

Ej:



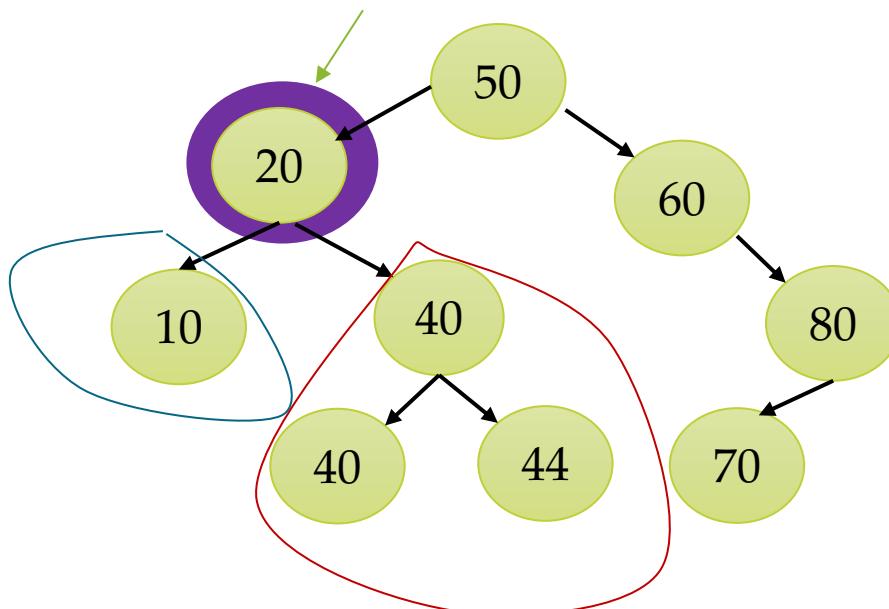
BST

Ej:



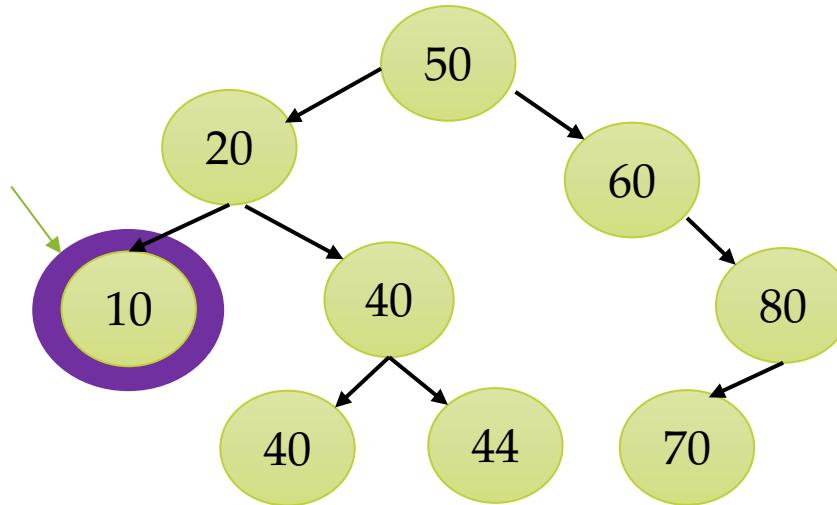
BST

Ej:

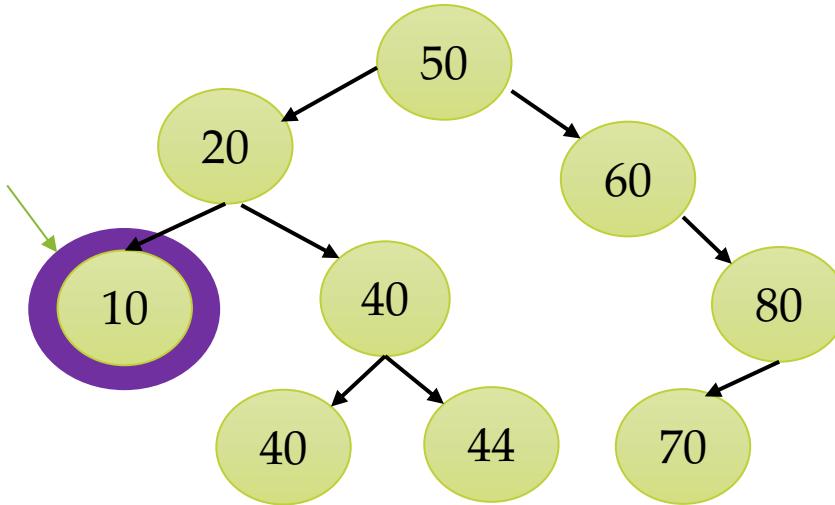


BST

Ej:

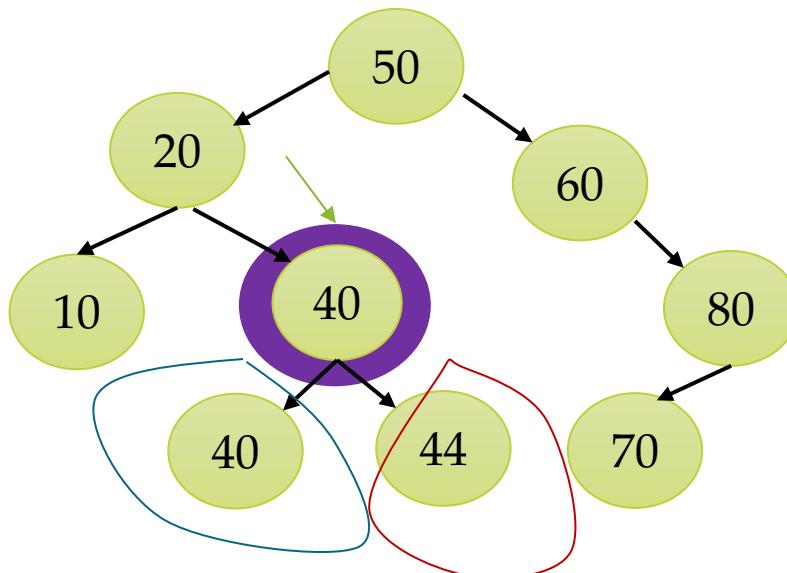


Ej:



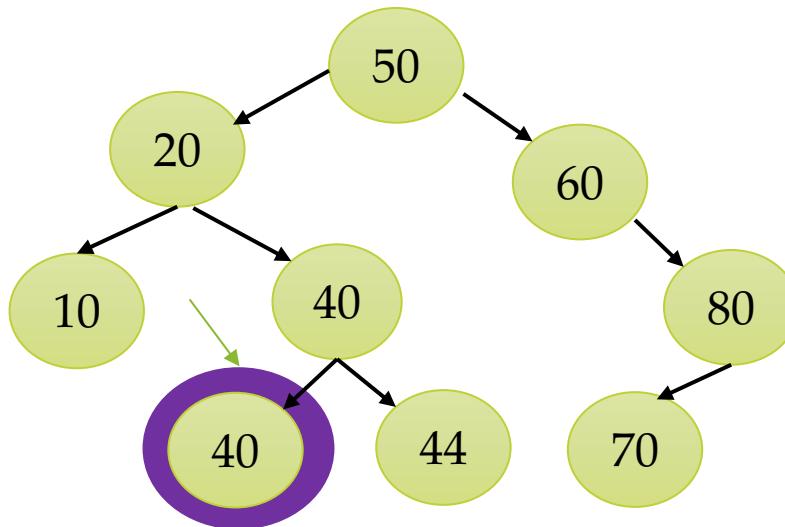
BST

Ej:



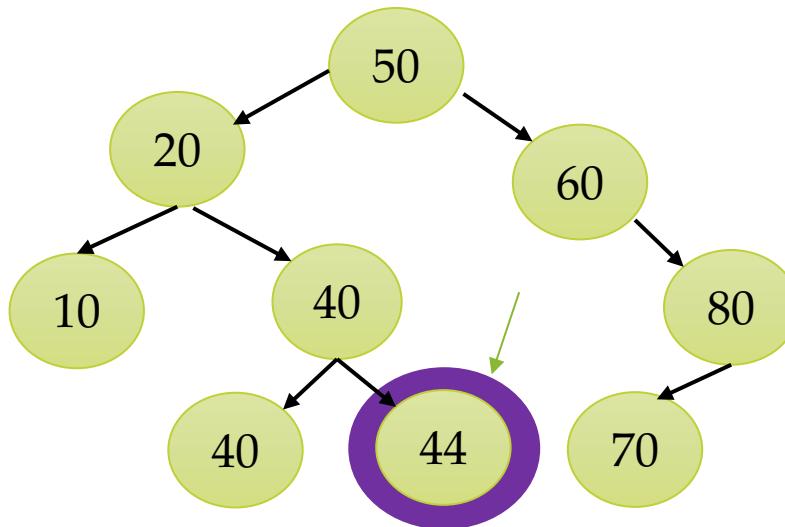
BST

Ej:



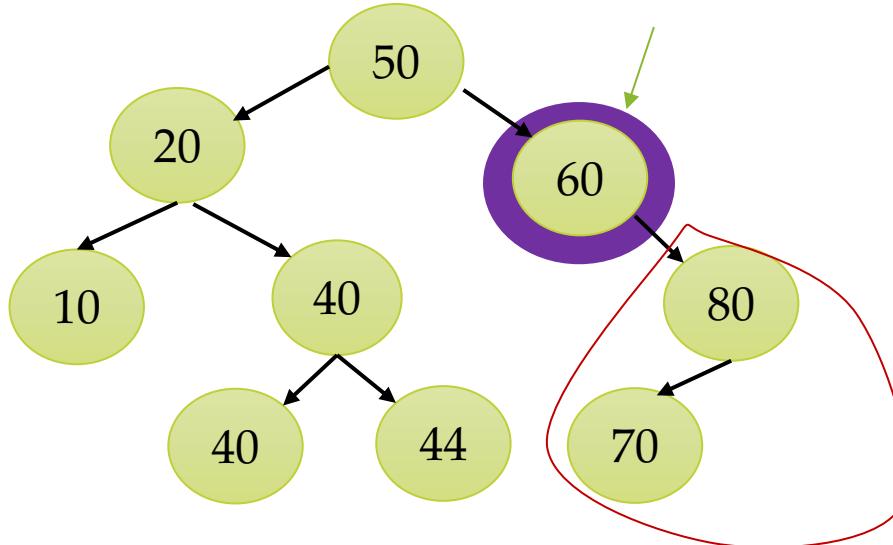
BST

Ej:



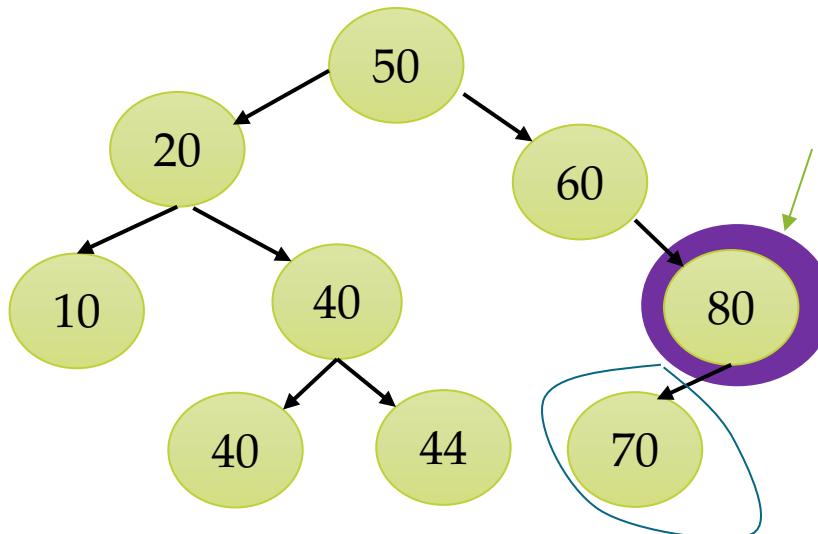
BST

Ej:



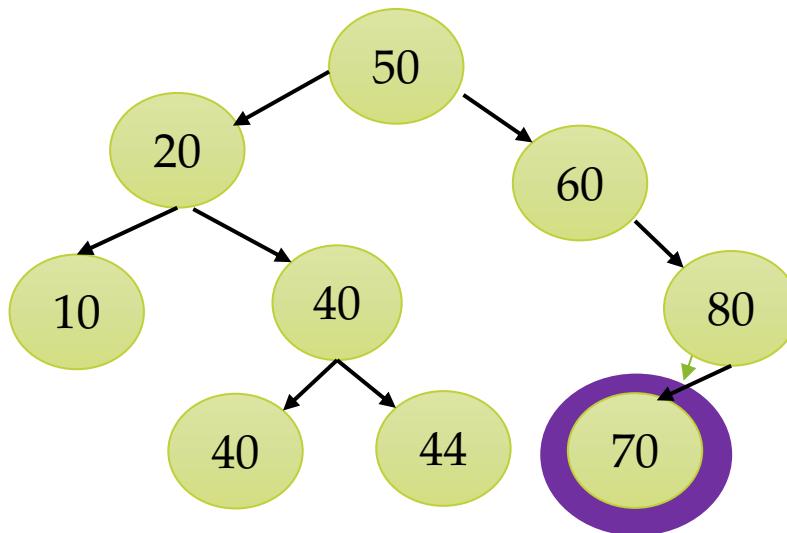
BST

Ej:



BST

Ej:



BST

Operaciones sobre un BST

- 1) Insertar: un BST crece desde las hojas.

Ej:

```
BST<Integer> myTree = new BST<>(); // root null
```

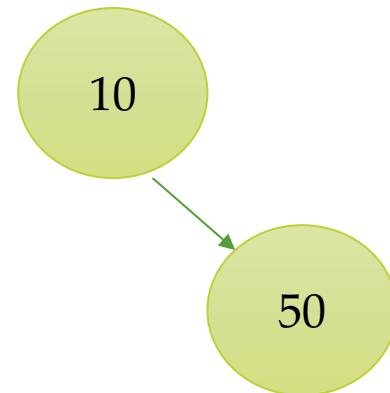
BST

```
myTree.insert(10);      // apareció el root.
```



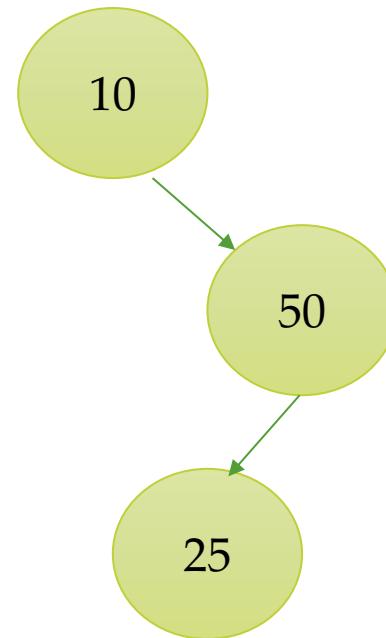
BST

```
myTree.insert(50);
```



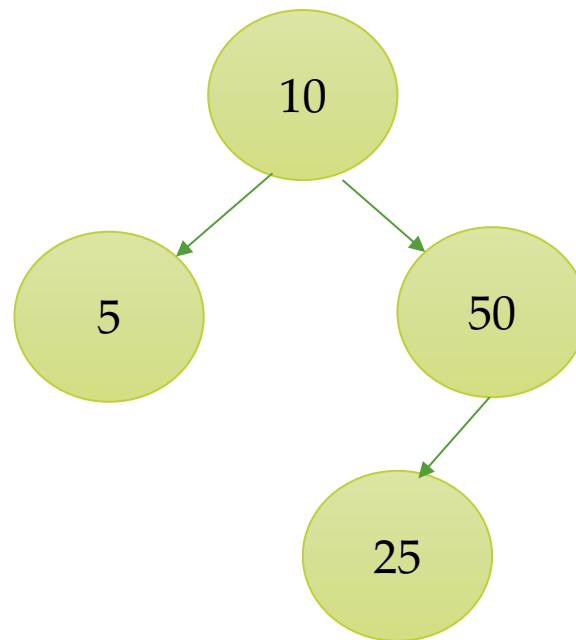
BST

```
myTree.insert(25);
```



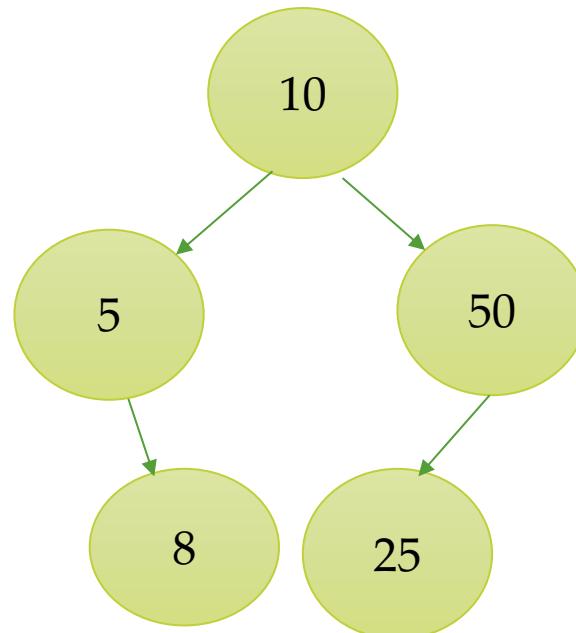
BST

```
myTree.insert(5);
```



BST

```
myTree.insert(8);
```



TP 5C – Ejer 1

Mostrar gráficamente, paso a paso, cómo quedaría la inserción en un BST si los datos se insertan en el siguiente orden: 50 60 80 20 70 40 44 10 40

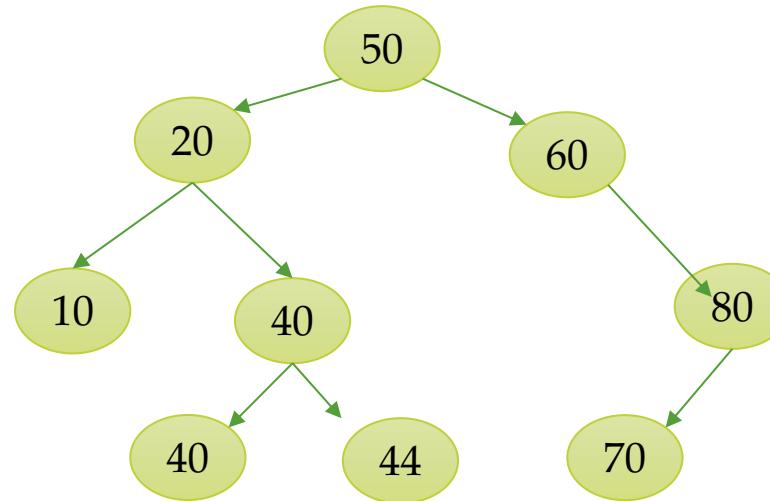


BST

Ejercicio 1

Mostrar gráficamente, paso a paso, cómo quedaría la inserción en un BST si los datos se insertan en el siguiente orden: 50 60 80 20 70 40 44 10 40

Rta final



TP 5C – Ejer 2

Armar un Proyecto Maven con las interfaces (Campus):
BSTreeInterface
NodeTreeInterface

Y la clase BST que implementa BSTreeInterface y la clase Node inner que implementa NodeTreeInterface.

Implementar la clase BST<T>



(bajar de campus)

Como mínimo, tendría el siguiente contrato que ofrecer:

```
package core;

public interface BSTreeInterface<T extends Comparable<? super T>> {

    void insert(T myData);

    void preOrder();

    void postOrder();

    void inOrder();

    NodeTreeInterface<T> getRoot();

    int getHeight();
}
```

(bajar de campus)

Y para el nodo interno también vamos a generar un contrato (getters) para luego ofrecer su graficación

```
package core;

public interface NodeTreeInterface<T extends Comparable<? super T>> {

    T getData();

    NodeTreeInterface<T> getLeft();

    NodeTreeInterface<T> getRight();

}
```

Implementar y chequear estas 2 técnicas para el método insert en el BST:

Insert se lo resuelve desde BST (sin precisar métodos de Node)

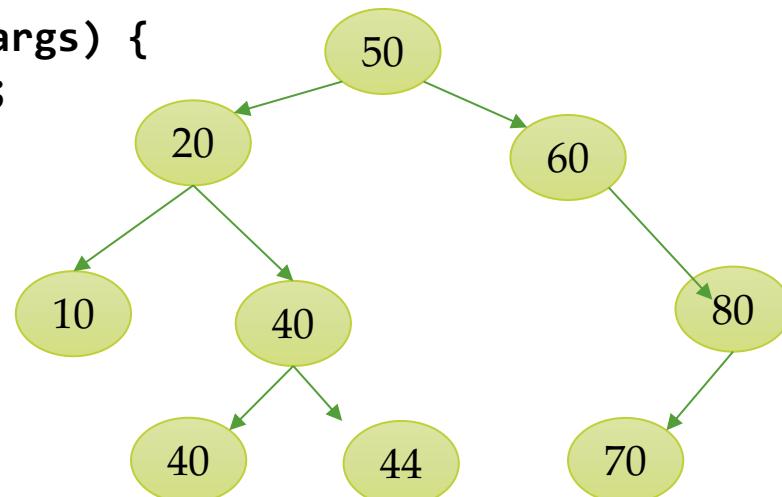
Insert se resuelve invocando desde BST al insert del Node

```
public class BST<T extends Comparable<? super T>> implements BSTreeInterface<T> {  
    ...  
  
    class Node implements NodeTreeInterface<T> {  
  
        private T myData;  
        private Node left;  
        private Node right;  
        ...  
    }  
}
```

BST

Caso de uso:

```
public static void main(String[] args) {  
    BST<Integer> myTree = new BST<>();  
    myTree.insert(50);  
    myTree.insert(60);  
    myTree.insert(80);  
    myTree.insert(20);  
    myTree.insert(70);  
    myTree.insert(40);  
    myTree.insert(44);  
    myTree.insert(10);  
    myTree.insert(40);  
  
    myTree.inOrder();  
    myTree.preOrder();  
    myTree.postOrder();
```



10 20 40 40 44 50 60 70 80

50 20 10 40 40 44 60 80 70

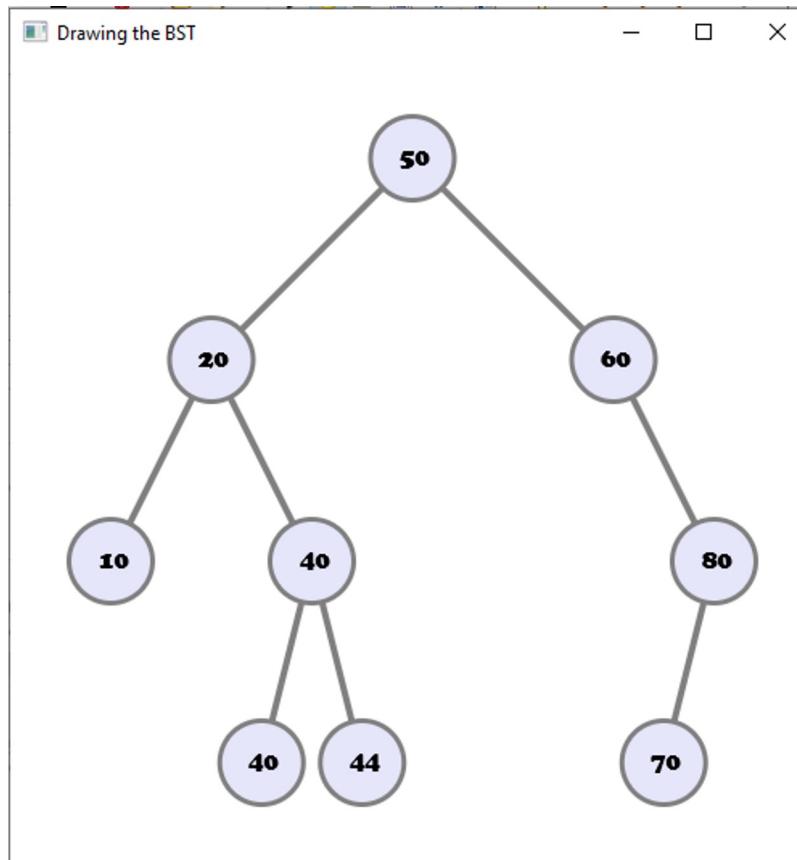
10 40 44 40 20 70 80 60 50

TP 5C – Ejer 3

Visualización javafx con
MVC



BST



BST

Asegurarse que tiene implementado los métodos (`insert`) y
`@Override`

```
public NodeTreeInterface<T> getRoot() {  
    ...  
}
```

`@Override`

```
public int getHeight() {  
    ...  
}
```

BST

Agregar al pom las dependencias:

```
<dependencies>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-fxml</artifactId>
    <version>11</version>
  </dependency>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-base</artifactId>
    <version>11</version>
  </dependency>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-controls</artifactId>
    <version>11</version>
  </dependency>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-graphics</artifactId>
    <version>11</version>
  </dependency>
</dependencies>
```

BST

Las interfaces deben estar dentro del package core

Ahora agregar al proyecto (Bajar de Campus):

GraphicsTree.java (en package controller)

Circle.java y Line.java (en package shape)

TestGUI.java (podría estar en package default). Es la aplicación de ejemplo javafx.

```
src/main/java
  (default package)
    > TestGUI.java
  controller
    > GraphicsTree.java
  core
    > BST.java
    > BSTreeInterface.java
    > NodeTreeInterface.java
  shape
    > Circle.java
    > Line.java
> TestGUI.java
```

Bajar <https://gluonhq.com/products/javafx/> la versión 11

Descompactar, por ejemplo en

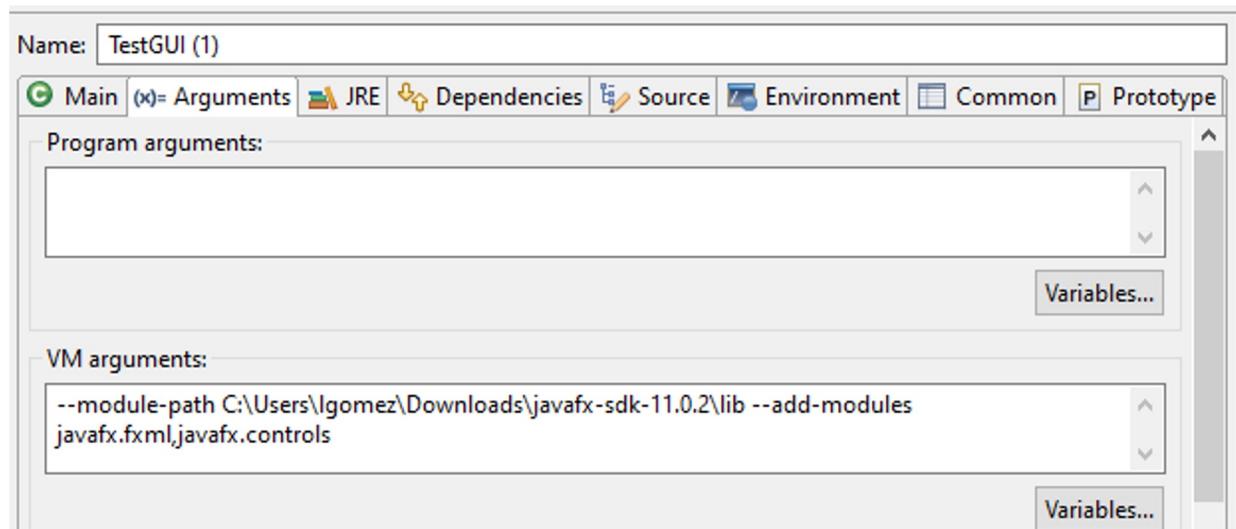
C:\Users\lgomez\Downloads\javafx-sdk-11.0.2

O para Linux/Mac:

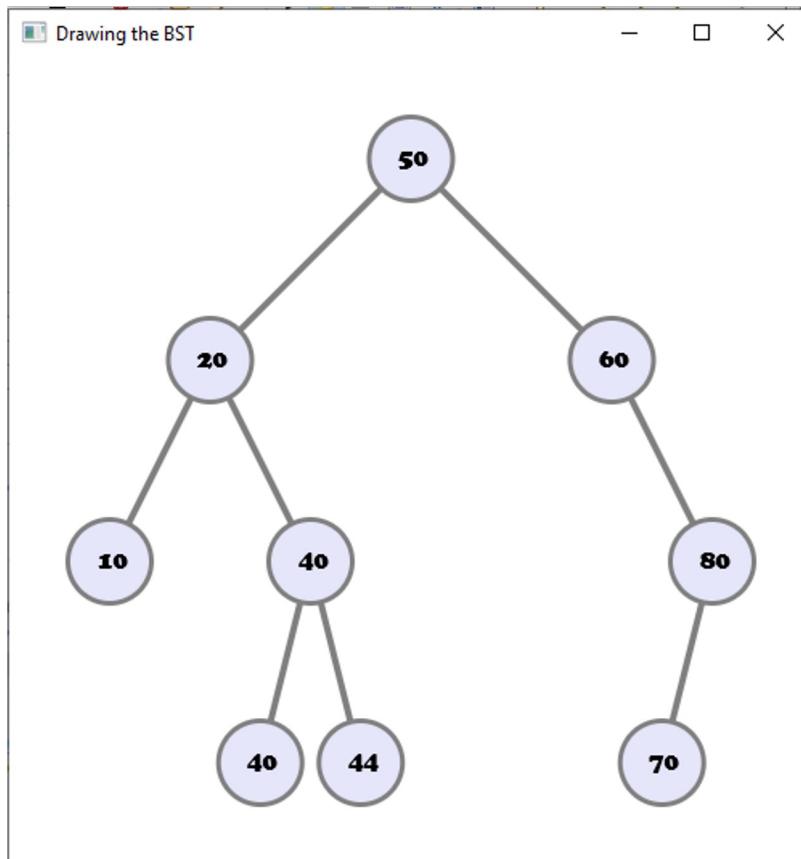
/Users/jabu/Downloads/javafx-sdk-11.0.2

Setear VM options la ejecutar (a donde está el lib):

```
--module-path C:\Users\lgomez\Downloads\javafx-sdk-11.0.2\lib --add-modules javafx.fxml,javafx.controls
```



Ejecutarlo.



TP 5C – Ejer 4

El método `createModel()` nos permite cambiar lo que queremos visualizar.



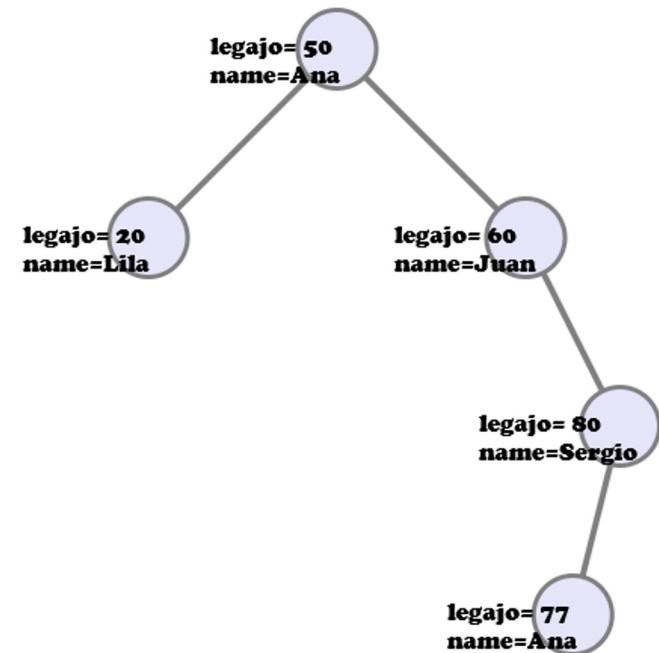
Crear un TAD Person: legajo, nombre. Participará del BST ordenado por legajo

BST

Caso de uso. Se crea el modelo y se lo invoca desde el GUI

```
private BST<Person> createModel() {  
    BST<Person> myTree = new BST<>();  
    myTree.insert(new Person(50, "Ana"));  
    myTree.insert(new Person(60, "Juan"));  
    myTree.insert(new Person(80, "Sergio"));  
    myTree.insert(new Person(20, "Lila"));  
    myTree.insert(new Person(77, "Ana"));  
    return myTree;  
}
```

Drawing the BST



BST

Operaciones sobre un BST

2) Borrar: no basta con eliminar un elemento, se debe mantener la forma del original (no deformarse)

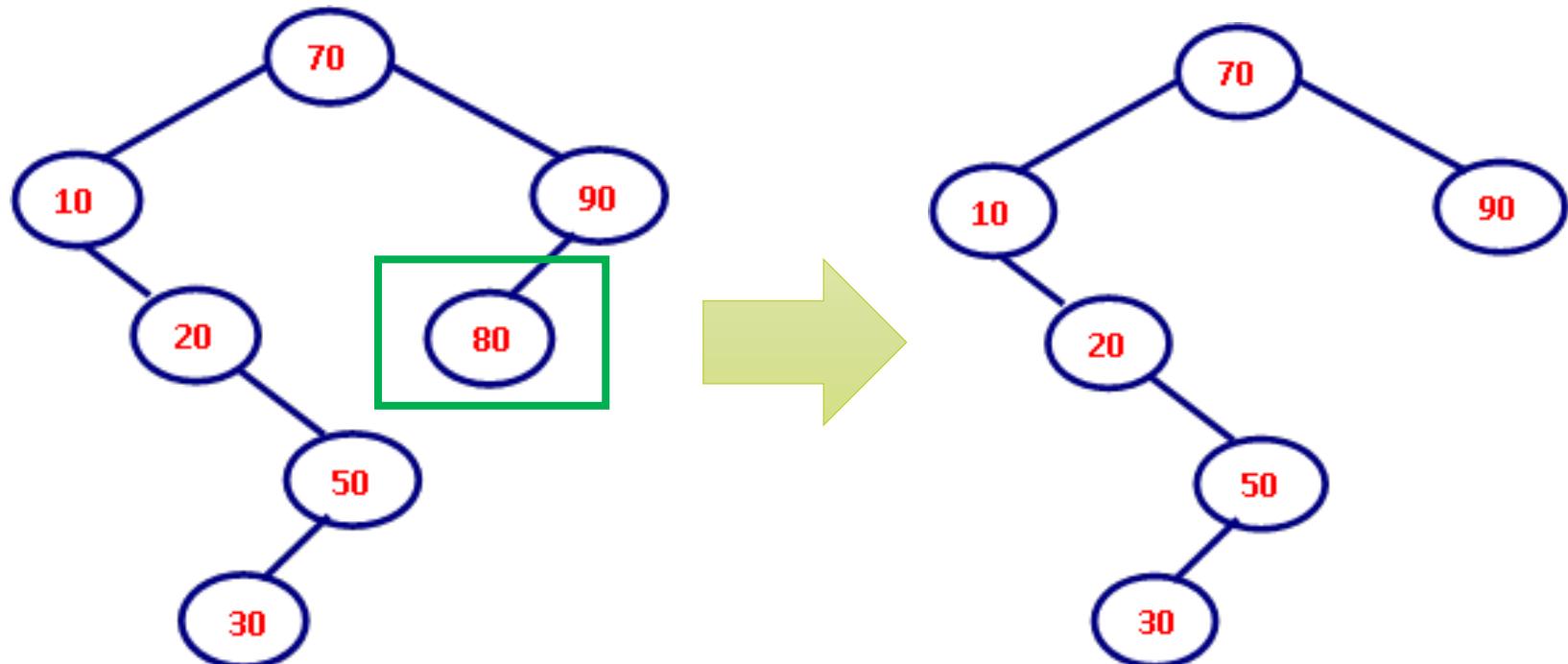
El algoritmo es el siguiente:

BST

- R1: Si el nodo a eliminar es hoja, actualizar quien lo apunta a él (puntero a raíz o su antecesor inmediato) para que ya no lo apunte más a él y pase a apuntar a NULL.
- R2: Si el nodo a eliminar tiene un solo hijo, actualizar quien lo apunta a él (puntero a raíz o su antecesor inmediato) para que en vez de apuntarlo a él lo haga al hijo del que se borra.
- R3: Si el nodo a eliminar tiene dos hijos se procede en dos pasos: primero se lo **reemplaza** por un *nodo lexicográficamente adyacente* (su *predecesor inorder* o sea el más grande de los nodos de su subárbol izquierdo, o bien su *sucesor inorder* o sea el más chico de los nodos de su subárbol derecho), y finalmente se borra al nodo que lo reemplazó (seguro que dicho nodo tiene a lo sumo un solo hijo, sino no sería el lexicográficamente adyacente, y por lo tanto es fácil de borrar)

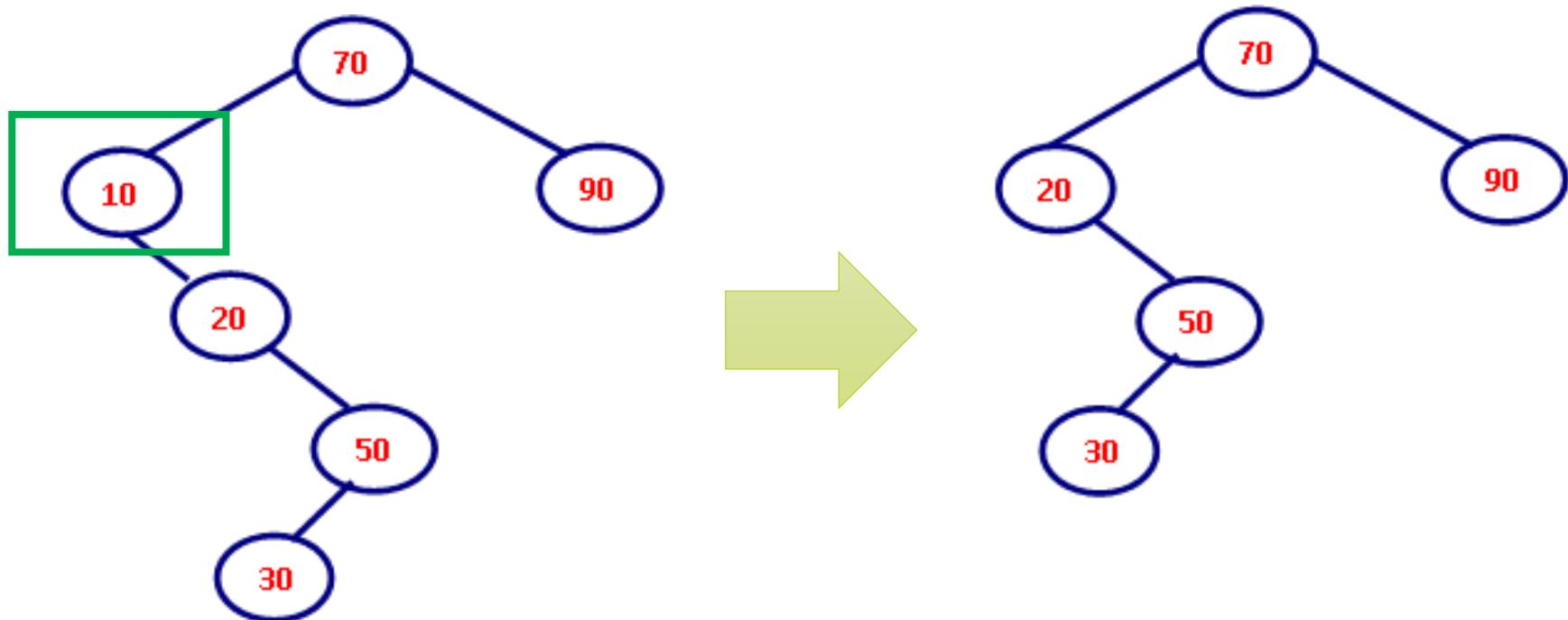
BST

Ej: Se quiere borrar el 80 (R1)



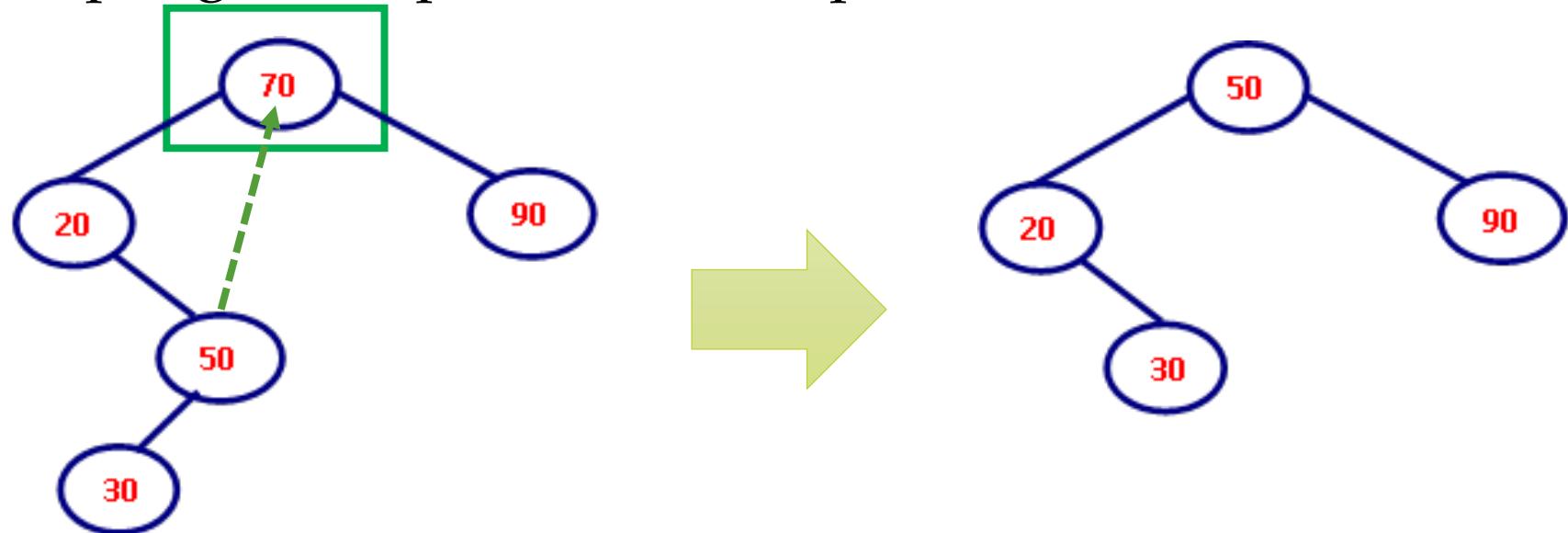
BST

A partir de lo obtenido se quiere borrar 10 (R2)



BST

A partir de lo obtenido se quiere borrar 70 (R3).
Supongamos que usamos su predecessor inorder



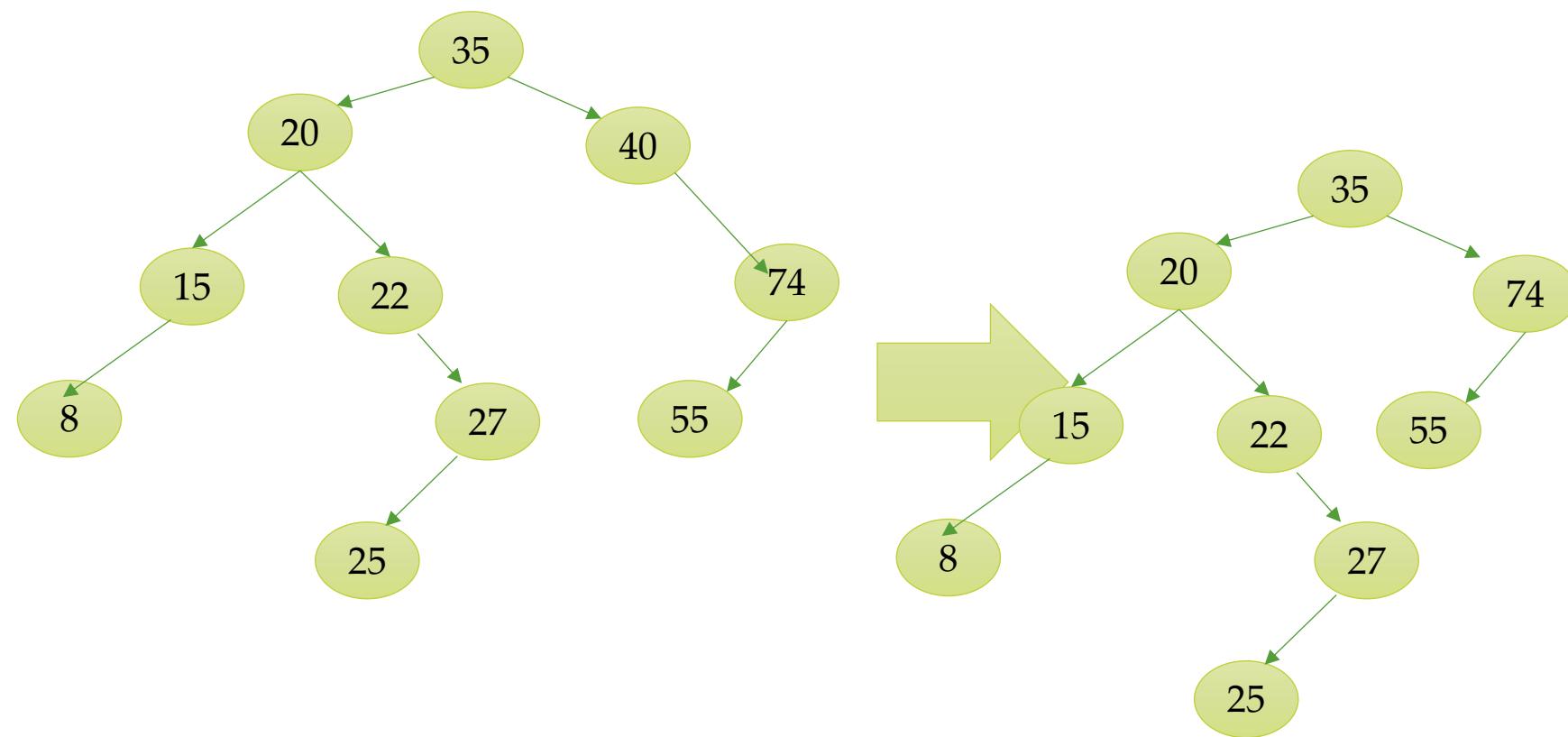
TP 5C – Ejer 6.1

A partir del siguiente BST, mostrar gráficamente cómo queda paso a paso el BST luego de aplicar las siguientes operaciones y qué reglas se usaron.

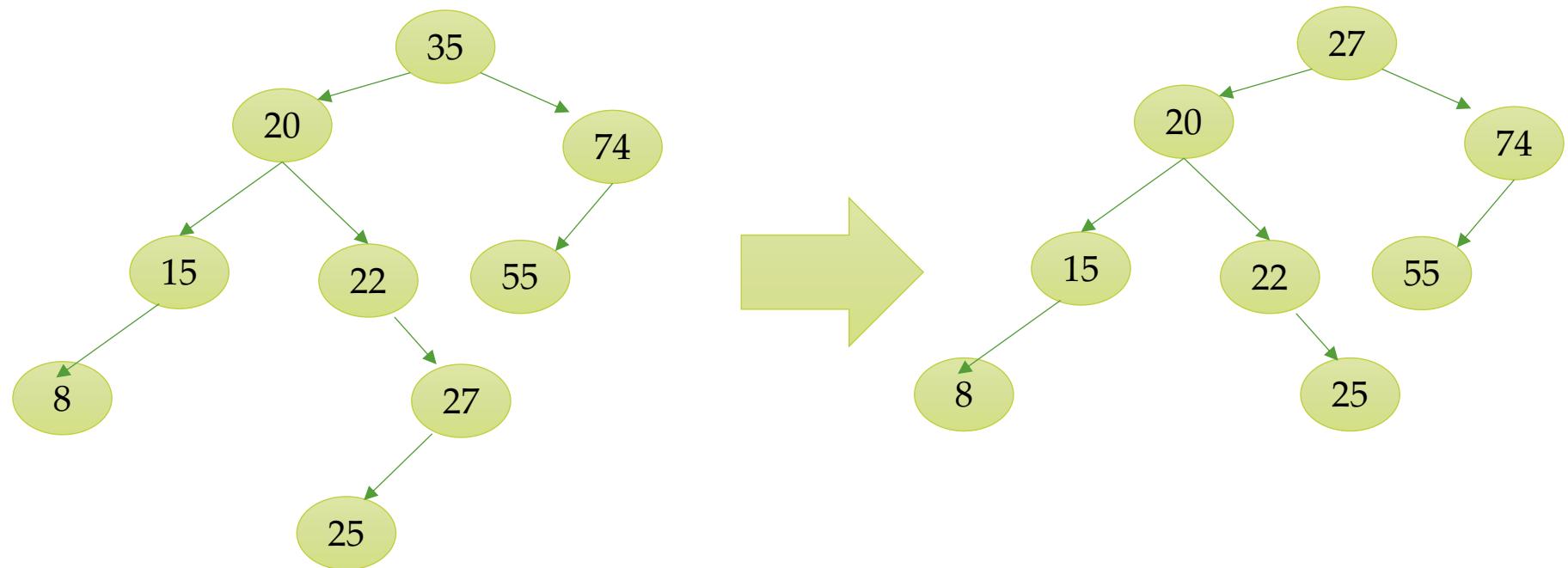
Usar predecesor inorder: 40, 35 y 8 (en ese orden)



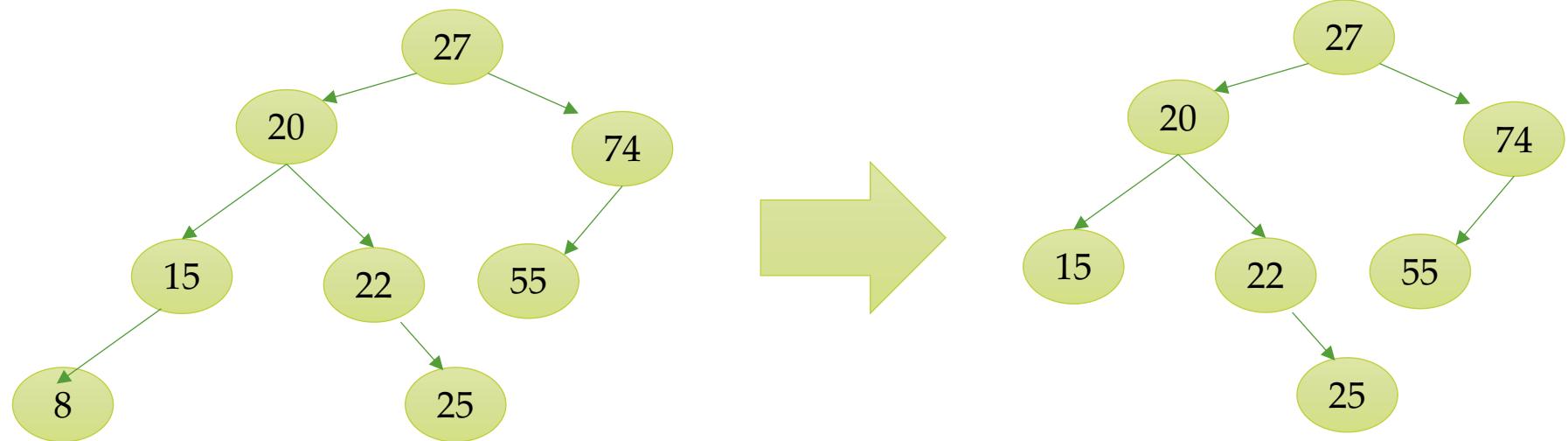
Borrar 40



Borrar 35



Borrar 8



TP 5C – Ejer 6.2

Implementar el borrado según lo explicado



Ejercicio 8.2

El borrado puede implementarse en forma recursiva

En BST

```
@Override  
public void delete(T myData) {  
    if (myData == null)  
        throw new RuntimeException("element cannot be null");  
  
    if (root != null) //delegates in Node class |  
        root= root.delete(myData);  
}
```

En Node

```
private Node delete(T myData) {
    if (myData.compareTo(this.data) < 0) {
        if (left != null) {
            left= left.delete(myData);
        }
        return this;
    }

    if (myData.compareTo(this.data) > 0) { //greater
        if (right != null) {
            right= right.delete(myData);
        }
        return this;
    }

    // found!
    //lexicographically replacement
    if (left == null) {
        return right;
    }

    if (right == null) {
        return left;
    }

    T replacement= lexiAdjacent(left);
    this.data= replacement;
    left= left.delete(this.data);
    return this;
}
```

```
private T lexiAdjacent(Node candidate) {
    Node auxi= candidate;
    // look forward
    while(auxi.right != null) {
        auxi= auxi.right;
    }

    return auxi.data;
}
```



Hacer un seguimiento de su funcionamiento.
Implementarlo y testearlo

TP 5C – Ejer 7.1

Cambiar la interface
BSTreeInterface para que sea
iterable



BST

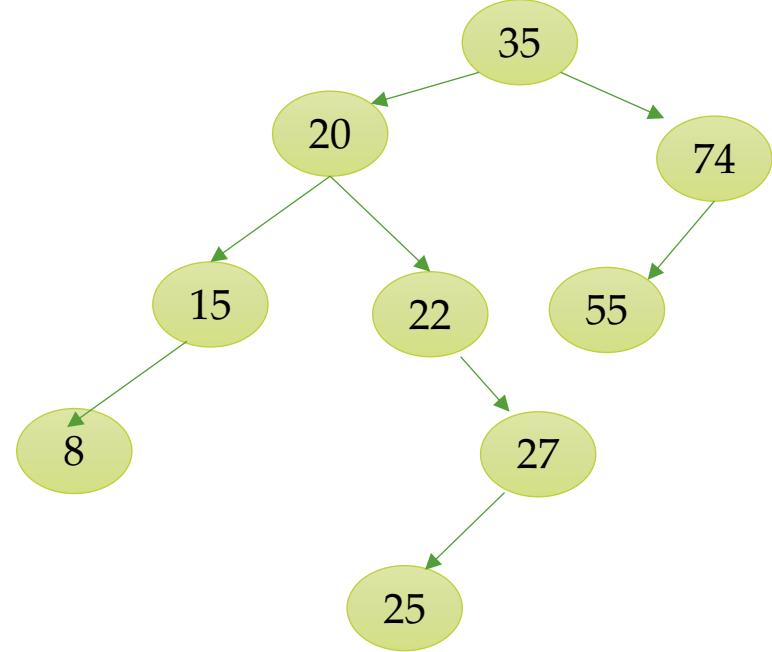
Para ello la interfaz `BSTreeInterface` extiende de `Iterable`.

El iterador devolverá los **elementos por niveles**, como lo hicimos previamente.

No asumir que pueden generar de un BST su información en una estructura auxiliar completamente. Tratar de “recorrer en el momento” o con algún element forward.

Caso de Uso:

```
BST<Integer> myTree = new BST<>();  
myTree.insert(35);  
myTree.insert(74);  
myTree.insert(20);  
myTree.insert(22);  
myTree.insert(55);  
myTree.insert(15);  
myTree.insert(8);  
myTree.insert(27);  
myTree.insert(25);  
  
for (Integer data : myTree) {  
    System.out.print(data + " ");  
}  
  
// Puedo hacerlo múltiples veces...  
System.out.println("\n\nUna vez más...\n");  
  
myTree.forEach( t-> System.out.print(t + " ") );
```



35 20 74 15 22 55 8 27 25

Una vez más

35 20 74 15 22 55 8 27 25

Tip: tomar la versión (iterativa) y transformarla.

```
// es iterativa
@Override
public void printByLevels() {
    if (root == null) {
        return;
    }

    // create an empty queue and enqueue the root node
    Queue<NodeTreeInterface<T>> queue = new LinkedList<>();
    queue.add(root);

    NodeTreeInterface<T >currentNode;

    // hay elementos?
    while (!queue.isEmpty())
    {
        currentNode = queue.remove();
        System.out.print(currentNode.getData() + " ");

        if (currentNode.getLeft() != null) {
            queue.add(currentNode.getLeft());
        }

        if (currentNode.getRight() != null) {
            queue.add(currentNode.getRight());
        }
    }

    System.out.println();
}
```

```
class BSTByLevelIterator implements Iterator<T>{

    private Queue<NodeTreeInterface<T>> queue;

    private BSTByLevelIterator() {
        // create an empty queue and enqueue the root node
        queue = new LinkedList<>();

        if (root!= null)
            queue.add(root);
    }

    @Override
    public boolean hasNext() {
        return !queue.isEmpty();
    }

    @Override
    public T next() {
        NodeTreeInterface<T> currentNode = queue.remove();

        if (currentNode.getLeft() != null) {
            queue.add(currentNode.getLeft());
        }

        if (currentNode.getRight() != null) {
            queue.add(currentNode.getRight());
        }

        return currentNode.getData();
    }
}
```

TP 5C – Ejer 7.2

Hacer que el iterador lo haga inOrder.

Para ello, conviene escribir el método inOrder en formato no recursivo.

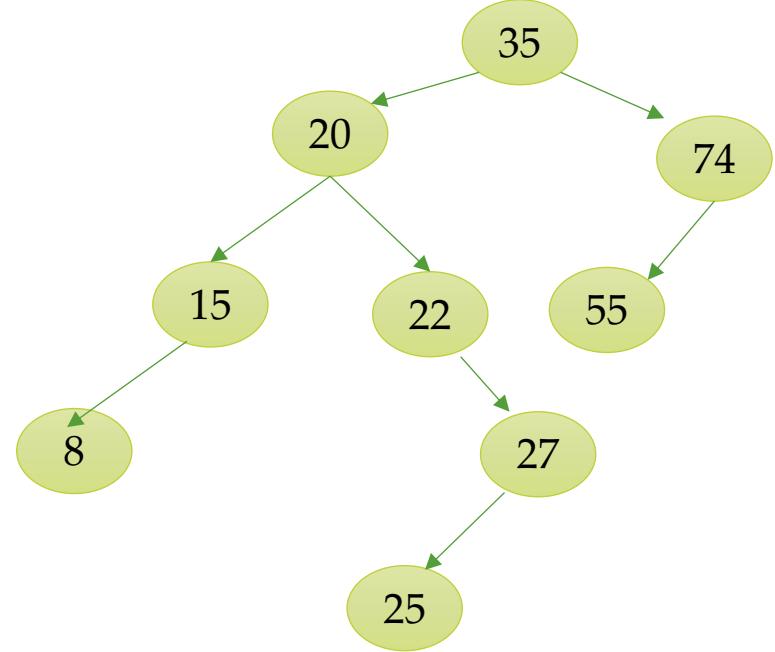


```
// versão iterativa
public void inOrderIter() {
    Stack<NodeTreeInterface<T>> stack= new Stack<>();

    NodeTreeInterface<T> current = root;
    while ( ! stack.isEmpty() || current != null) {
        if (current != null) {
            stack.push(current);
            current= current.getLeft();
        }
        else {
            NodeTreeInterface<T> elementToProcess = stack.pop();
            System.out.print(elementToProcess.getData() + "\t");
            current= elementToProcess.getRight();
        }
    }
}
```

Caso de Uso:

```
BST<Integer> myTree = new BST<>();  
myTree.insert(35);  
myTree.insert(74);  
myTree.insert(20);  
myTree.insert(22);  
myTree.insert(55);  
myTree.insert(15);  
myTree.insert(8);  
myTree.insert(27);  
myTree.insert(25);  
  
for (Integer data : myTree) {  
    System.out.print(data + " ");  
}  
  
// Puedo hacerlo múltiples veces...  
System.out.println("\n\nUna vez más...\n");  
  
myTree.forEach( t-> System.out.print(t + " ") );
```



debería obtenerse
8 15 20 22 25 27 35 55 74

Una vez más
8 15 20 22 25 27 35 55 74

```

class BSTInOrderIterator implements Iterator<T> {
    Stack<NodeTreeInterface<T>> stack;
    NodeTreeInterface<T> current;

    public BSTInOrderIterator() {
        stack= new Stack<>();
        current= root;
    }

    @Override
    public boolean hasNext() {
        return ! stack.isEmpty() || current != null;
    }

    @Override
    public T next() {
        while (current != null) {
            stack.push(current);
            current= current.getLeft();
        }
        else {
            NodeTreeInterface<T> elementToProcess = sta
            System.out.print(elementToProcess.getData())
            current= elementToProcess.getRight();
        }
    }
}

// version iterativa
public void inOrderIter() {
    Stack<NodeTreeInterface<T>> stack= new Stack<>();

    NodeTreeInterface<T> current = root;
    while ( ! stack.isEmpty() || current != null) {
        if (current != null) {
            stack.push(current);
            current= current.getLeft();
        }
        else {
            NodeTreeInterface<T> elementToProcess = sta
            System.out.print(elementToProcess.getData())
            current= elementToProcess.getRight();
        }
    }
}

```

TP 5C – Ejer 7.3

Dejar que el usuario elija el iterador que desea, tantas veces lo quiera



BST

El problema es que no se puede cambiar el método `iterator()` parametrizandolo por la forma en que se lo quiere recorrer.

```
@Override  
public Iterator<T> iterator( ?? ) {  
    ...  
}
```

BST

Possible solución:

Agregar un método que hay que invocar “antes” de tomar el iterador que indique de qué manera se lo quiere recorrer. El default, de no invocarse, es iteración byLevels.

Chequearlo con múltiples invocaciones de formas de recorrido!!!

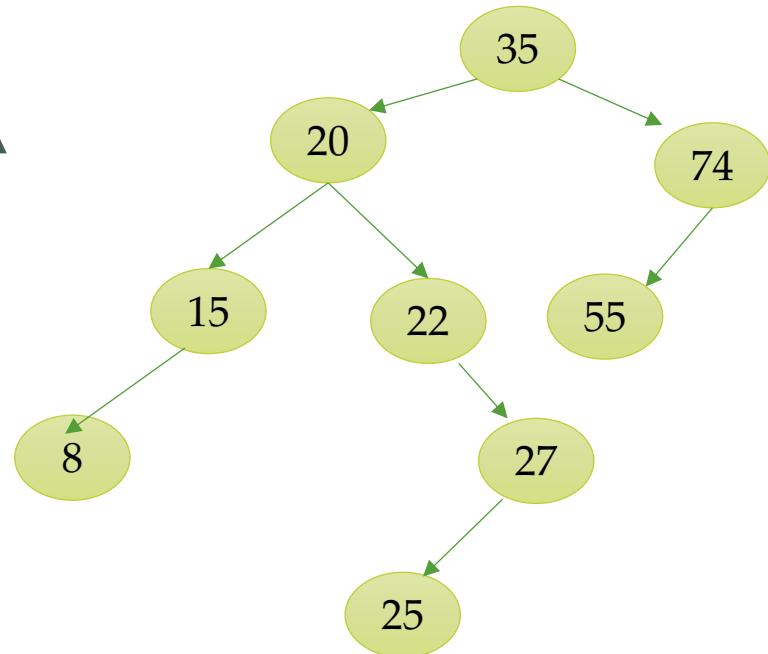
Básicamente:

Sacar comentario en la interface...

```
public interface BSTreeInterface<T extends Comparable<? super T>> ...{  
    enum Traversal { BYLEVELS, INORDER}  
  
    void setTraversal(Traversal traversal);  
  
    ...  
}
```

Caso de uso A

```
public static void main(String[] args) {  
    BST<Integer> myTree= new BST<>();  
    myTree.insert(35);  myTree.insert(74);  
    myTree.insert(20);  myTree.insert(22);  
    myTree.insert(55);  myTree.insert(15);  
    myTree.insert(8);   myTree.insert(27);  
    myTree.insert(25);  
  
    System.out.println("\n\nDefault Traversal...\n");  
    myTree.forEach( t-> System.out.print(t + " ") );  
  
    myTree.setTraversal(Traversal.INORDER);  
    System.out.println("\n\nUna vez más INORDER\n");  
    myTree.forEach( t-> System.out.print(t + " ") );  
  
    myTree.setTraversal(Traversal.BYLEVELS);  
    System.out.println("\n\nUna vez más BYLEVELS\n");  
    myTree.forEach( t-> System.out.print(t + " ") );  
  
    myTree.setTraversal(Traversal.INORDER);  
    System.out.println("\n\nUna vez más INORDER\n");  
    myTree.forEach( t-> System.out.print(t + " ") );  
}  
}
```



Default Traversal

35 20 74 15 22 55 8 27 25

Una vez más INORDER

8 15 20 22 25 27 35 55 74

Una vez más BYLEVELS

35 20 74 15 22 55 8 27 25

Una vez más INORDER

8 15 20 22 25 27 35 55 74

Possible Implementación

```
@Override  
public Iterator<T> iterator() {  
  
    switch (aTraversal) {  
        case BYLEVELS: return new BSTByLevelIterator();  
        case INORDER: return new BSTInOrderIterator();  
  
    }  
    throw new RuntimeException("invalid traversal parameter");  
}
```