Binary Search Tree

Ja-Hee Kim

Agenda

01 ADT of Dictionary

ADT for searching efficiently with a key

02 Binary Search Tree

Definition

03 Implementation of BST

Insert, search, remove

04 AVL Tree

Motivation, definition, insertion

ADT of a Dictionary

ADT: Dictionary

- A dictionary is a collection of ordered items.
- Aliases: map, table, associative array
  - Keyword
    - Search key
    - Example: English word, person's name
  - Value
    - Data associated with that key.
    - Example: definition, address, telephone number
- ADT Dictionary should enable you to locate the desired entry efficiently.

Operations of Dictionary

- Adds the pair (key, value) to the dictionary.
- Removes from the dictionary the entry that corresponds to a given search key.
- Retrieves from the dictionary the value that corresponds to a given search key.
- See whether the dictionary contains a given search key
- Traverse all the search keys in the dictionary
- Traverse all the values in the dictionary

- Detect whether a dictionary is empty
- Get the number of entries in the dictionary
- Remove all entries from the dictionary

Dictionary<K,V>
+add(K key, V value): boolean +remove(K key): boolean +contains(K key): V +toString():String +isEmpty(): boolean +size(): int +remove()

Built-in interface: Map

java.util package contains the interface Map<K,V>

- K - the type of keys maintained by this map
- V - the type of mapped values

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
void	clear() Removes all of the mappings from this map (optional operation).
default V	compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
default V	computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
default V	computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.
boolean	equals(Object o) Compares the specified object with this map for equality.
default void	forEach(BiConsumer<? super K,? super V> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
V	get(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
default V	getOrDefault(Object key, V defaultValue) Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
int	hashCode() Returns the hash code value for this map.
boolean	isEmpty() Returns true if this map contains no key-value mappings.
Set<K>	keySet() Returns a Set view of the keys contained in this map.
default V	merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V	put(K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	putAll(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map (optional operation).
default V	putIfAbsent(K key, V value) If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
V	remove(Object key) Removes the mapping for a key from this map if it is present (optional operation).
default boolean	remove(Object key, Object value) Removes the entry for the specified key only if it is currently mapped to the specified value.
default V	replace(K key, V value) Replaces the entry for the specified key only if it is currently mapped to some value.
default boolean	replace(K key, V oldValue, V newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
default void	replaceAll(BiFunction<? super K,? super V,? extends V> function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
int	size() Returns the number of key-value mappings in this map.
Collection<V>	values() Returns a Collection view of the values contained in this map.

Example

```java
import java.util.*;
public class DictionarayTest {
    public static void main(String[] args) {
        Map<String, Integer> address = new HashMap<>();
        if(address.isEmpty()) System.out.println("No number in my emergency phone address book");
        else System.out.println("I have "+address.size()+" numbers");
        address.put("Korea", 119);
        address.put("SC", 116);
        address.put("EU", 112);
        address.put("USA", 911);
        address.put("Australia", 000);
        address.put("London", 999);
        address.put("France", 17);
        if(address.isEmpty()) System.out.println("No number in my emergency phone address book");
        else System.out.println("I have "+address.size()+" numbers");
        if(address.containsKey("USA")) System.out.println("The emergency phone number in USA is "+ address.get("USA"));
        else System.out.println("We cannot find emergency phone number in U.S.");
        if(address.containsKey("Japan")) System.out.println(address.get("Japan"));
        else System.out.println("We cannot find emergency phone number in Japan");
        System.out.println("Emergency phone number book: "+ address);
        System.out.println("Removing USA: "+ address.remove("USA"));
        System.out.println("Removing SC: "+ address.remove("SC"));
        System.out.println("Removing Korea: "+ address.remove("Korea"));
        System.out.println("Emergency phone number book: "+ address);
    }
}
```

Problems Javadoc Declaration Console

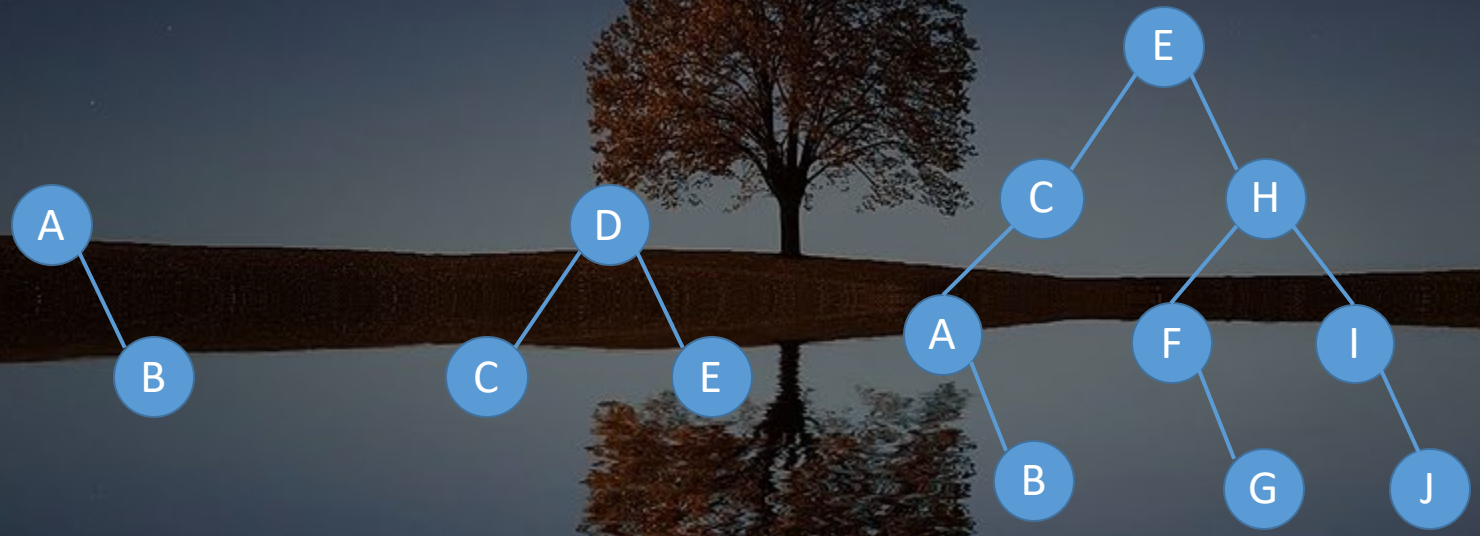
<terminated> DictionarayTest [Java Application] C:₩Program Files₩Java₩jdk-14.0.2₩bin₩javaw.exe (2020. 10. 27. 오후 10:34:28 – 오후 10:34:28)

```
No number in my emergency phone address book
I have 7 numbers
The emergency phone number in USA is 911
We cannot find emergency phone number in Japan
Emergency phone number book: {SC=116, EU=112, USA=911, London=999, Australia=0, France=17, Korea=119}
Removing USA: 911
Removing SC: 116
Removing Korea: 119
Emergency phone number book: {EU=112, London=999, Australia=0, France=17}
```

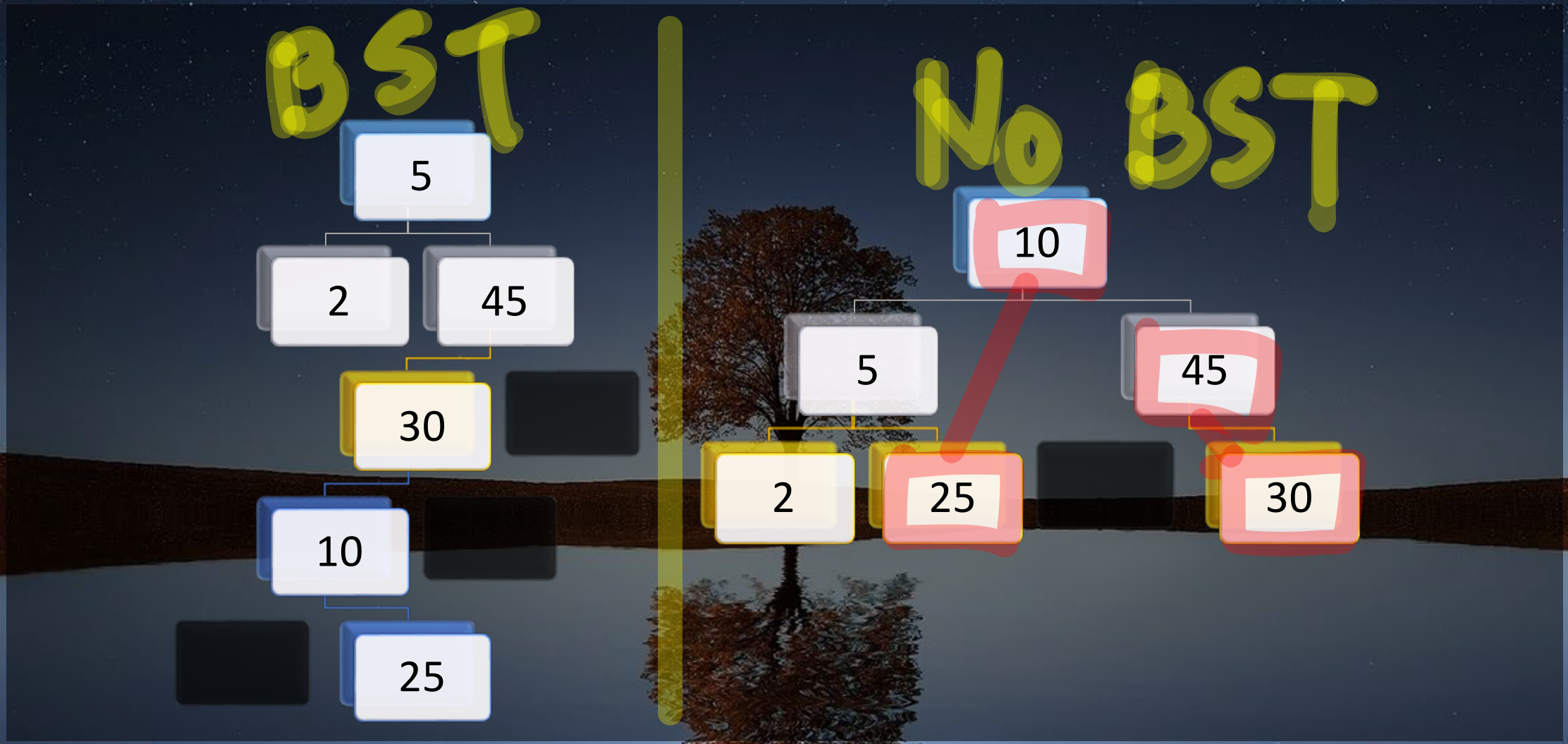
Binary Search Tree

Binary Search Tree (BST)

- Each node can have up to two child nodes.
- All keys should be different.
- Each node has a key
  - All keys of the left subtree is less than the root's
  - All keys of the right subtree is greater than the root's



Q: Which is a BST?



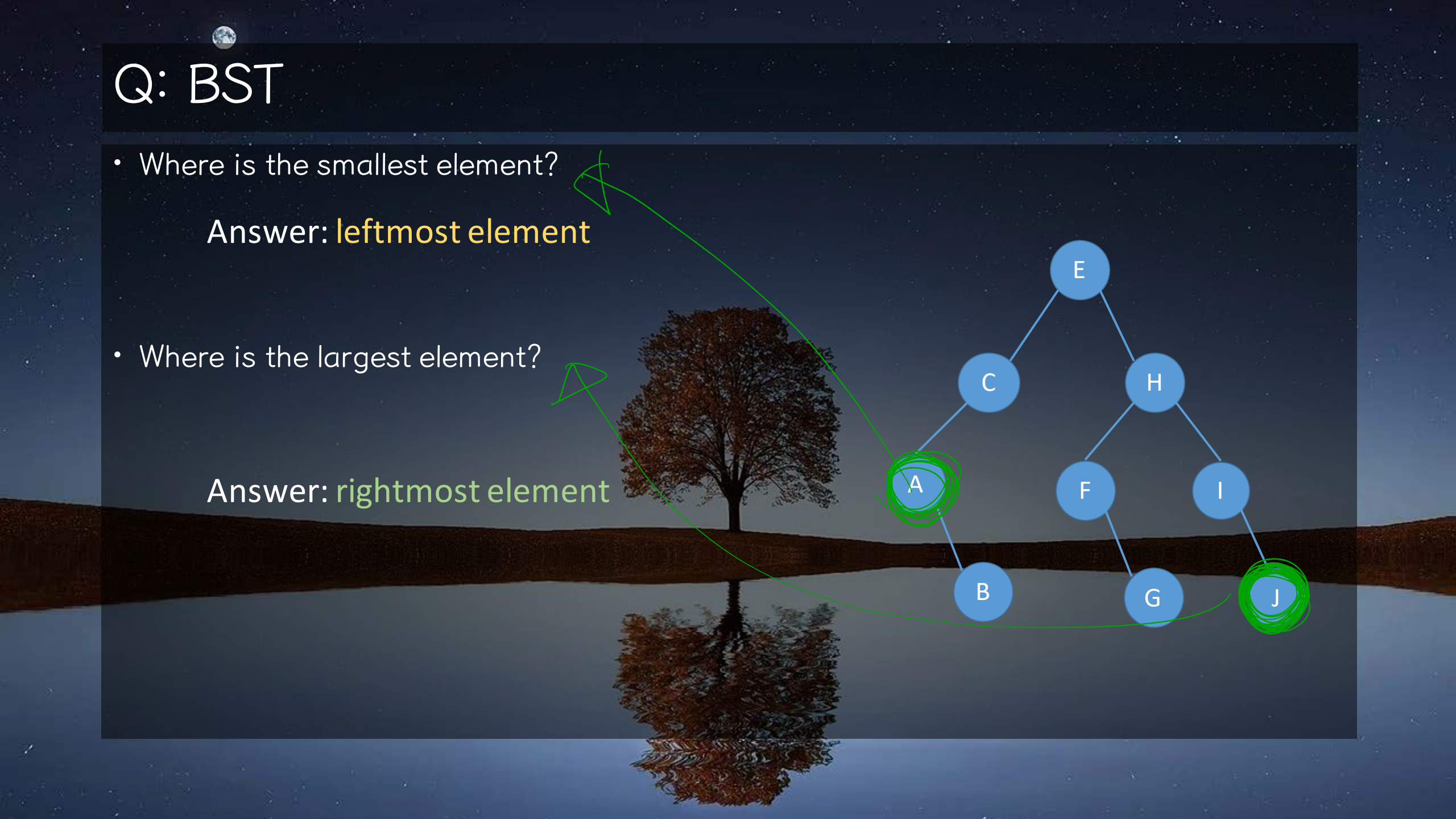
Q: BST

- Where is the smallest element?

  Answer: leftmost element

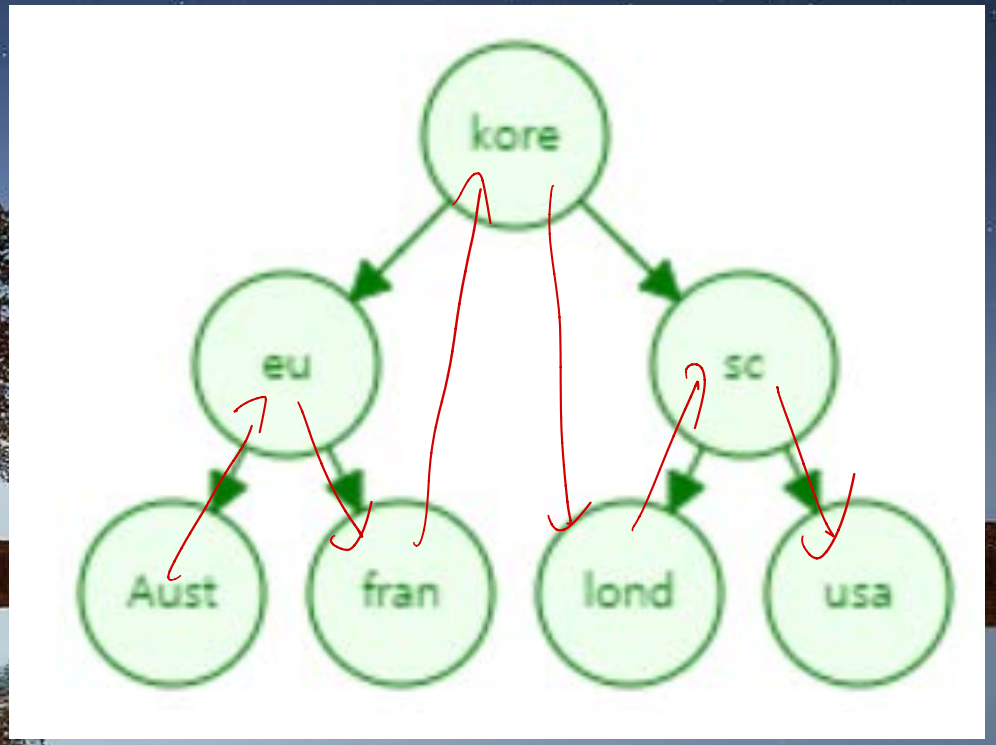
- Where is the largest element?

  Answer: rightmost element



Q: In-order traversal of BST

- In-order traversal of BST = sorted list
- Result of in-order traversal
  - Autralia→EU → France → Korea → Londo n → Southern California → USA



- Visualization tool: https://www.cs.usfca.edu/~galles/visualization/BST.html

Implementation of BST

Insert a node

- Case 1:
  - The Tree is Empty, set the root to a new node containing the item
- Case 2:
  - The Tree is Not Empty, Call a recursive helper method to insert the item
  - Case 2-1: If New data < key
    - Case 2-1-1: if the left is not empty, insert x in the left sub tree
    - Case 2-1-2: else if the left is empty, new data is a left child
  - Case 2-2: If New data > key
    - Case 2-2-1: if the right is not empty, insert x in the right sub tree
    - Case 2-2-2: else if the right is empty, new data is a right child

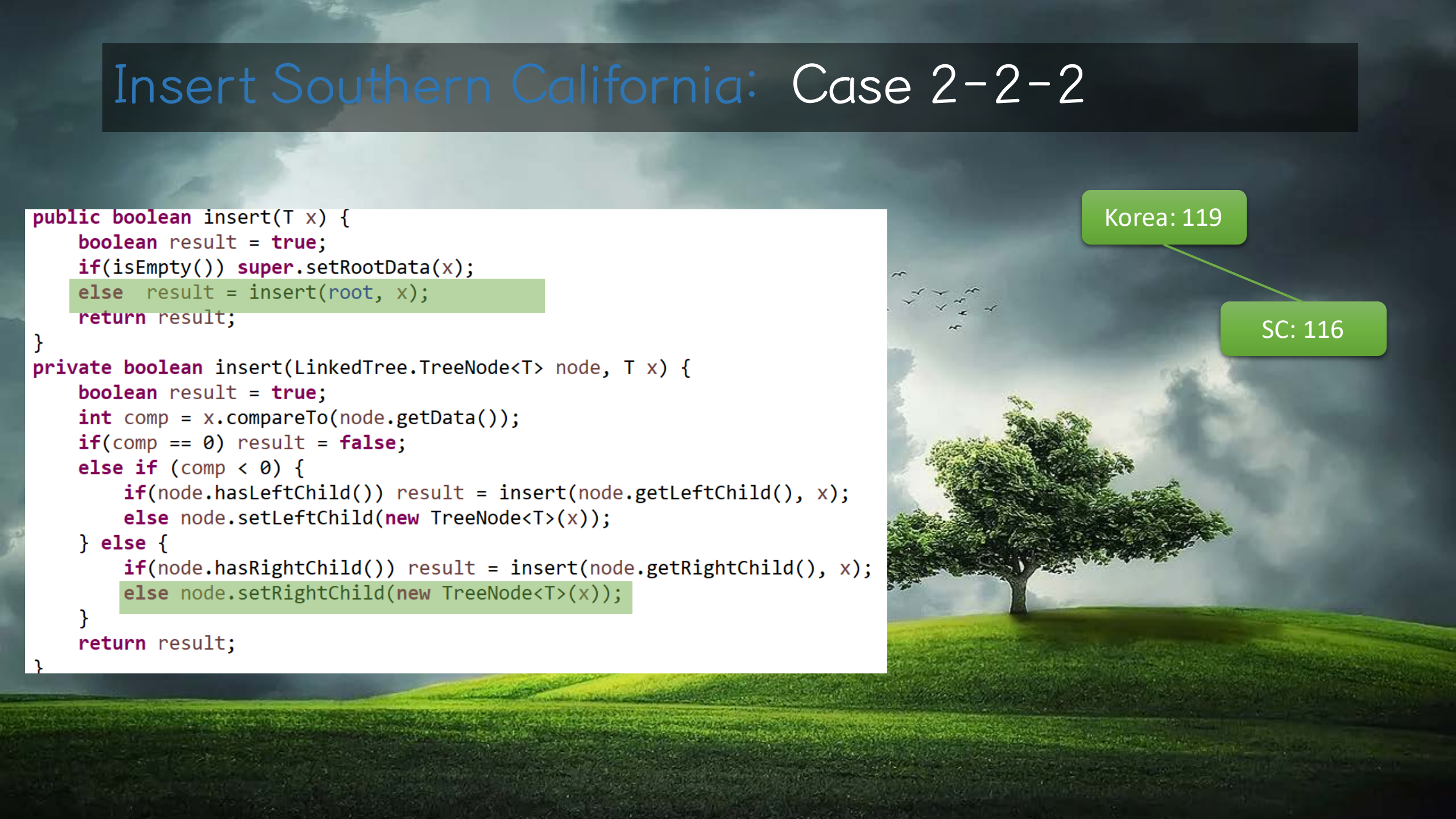
Insert Korea: case 1 - the tree is empty

```
public boolean insert(T x) {
    boolean result = true;
    if(isEmpty()) super.setRootData(x);
    else  result = insert(root, x);
    return result;
}
private boolean insert(LinkedTree.TreeNode<T> node, T x) {
    boolean result = true;
    int comp = x.compareTo(node.getData());
    if(comp == 0) result = false;
    else if (comp < 0) {
        if(node.hasLeftChild()) result = insert(node.getLeftChild(), x);
        else node.setLeftChild(new TreeNode<T>(x));
    } else {
        if(node.hasRightChild()) result = insert(node.getRightChild(), x);
        else node.setRightChild(new TreeNode<T>(x));
    }
    return result;
}
```

Korea: 119

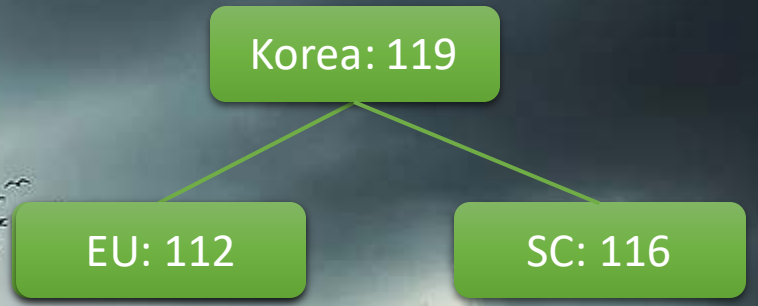
Insert Southern California: Case 2-2-2

```
public boolean insert(T x) {
    boolean result = true;
    if(isEmpty()) super.setRootData(x);
    else  result = insert(root, x);
    return result;
}
private boolean insert(LinkedTree.TreeNode<T> node, T x) {
    boolean result = true;
    int comp = x.compareTo(node.getData());
    if(comp == 0) result = false;
    else if (comp < 0) {
        if(node.hasLeftChild()) result = insert(node.getLeftChild(), x);
        else node.setLeftChild(new TreeNode<T>(x));
    } else {
        if(node.hasRightChild()) result = insert(node.getRightChild(), x);
        else node.setRightChild(new TreeNode<T>(x));
    }
    return result;
}
```



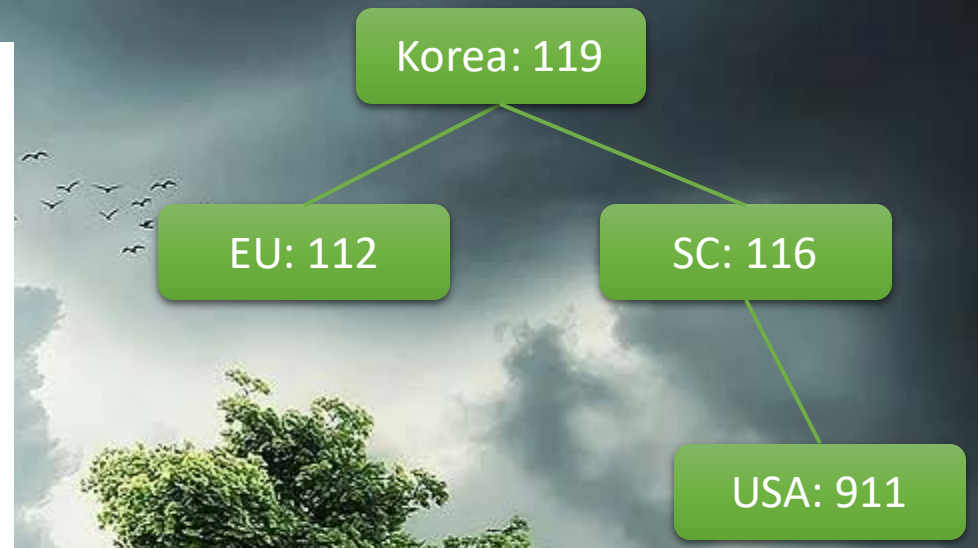
Insert EU: Case 2-1-2

```
public boolean insert(T x) {
    boolean result = true;
    if(isEmpty()) super.setRootData(x);
    else  result = insert(root, x);
    return result;
}
private boolean insert(LinkedTree.TreeNode<T> node, T x) {
    boolean result = true;
    int comp = x.compareTo(node.getData());
    if(comp == 0) result = false;
    else if (comp < 0) {
        if(node.hasLeftChild()) result = insert(node.getLeftChild(), x);
        else node.setLeftChild(new TreeNode<T>(x));
    } else {
        if(node.hasRightChild()) result = insert(node.getRightChild(), x);
        else node.setRightChild(new TreeNode<T>(x));
    }
    return result;
}
```



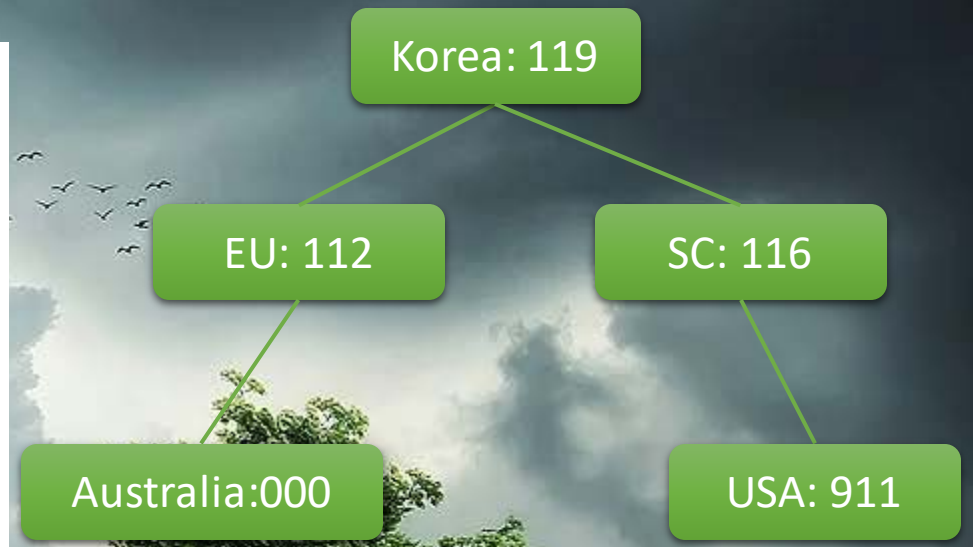
Insert EU: Case 2-2-1→case 2-2-2

```
public boolean insert(T x) {
    boolean result = true;
    if(isEmpty()) super.setRootData(x);
    else  result = insert(root, x);
    return result;
}
private boolean insert(LinkedTree.TreeNode<T> node, T x) {
    boolean result = true;
    int comp = x.compareTo(node.getData());
    if(comp == 0) result = false;
    else if (comp < 0) {
        if(node.hasLeftChild()) result = insert(node.getLeftChild(), x);
        else node.setLeftChild(new TreeNode<T>(x));
    } else {
        if(node.hasRightChild()) result = insert(node.getRightChild(), x);
        else node.setRightChild(new TreeNode<T>(x));
    }
    return result;
}
```



Insert EU: Case 2-1-1→case 2-1-2

```
public boolean insert(T x) {
    boolean result = true;
    if(isEmpty()) super.setRootData(x);
    else  result = insert(root, x);
    return result;
}
private boolean insert(LinkedTree.TreeNode<T> node, T x) {
    boolean result = true;
    int comp = x.compareTo(node.getData());
    if(comp == 0) result = false;
    else if (comp < 0) {
        if(node.hasLeftChild()) result = insert(node.getLeftChild(), x);
        else node.setLeftChild(new TreeNode<T>(x));
    } else {
        if(node.hasRightChild()) result = insert(node.getRightChild(), x);
        else node.setRightChild(new TreeNode<T>(x));
    }
    return result;
}
```



Tree Search

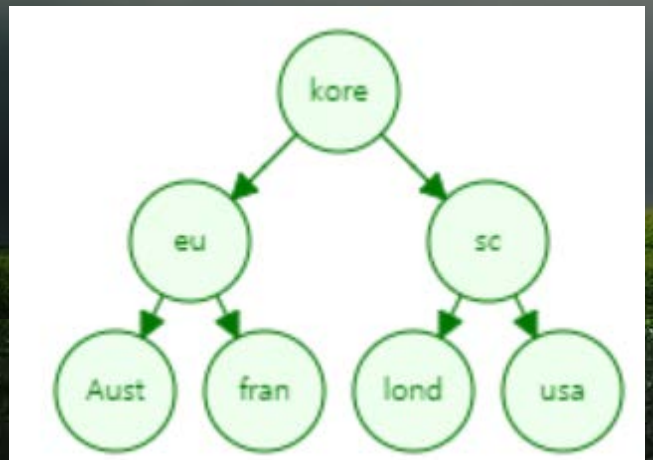
```
search(k)
if key=k: return this
else if k<key:
        if left == null: return null
        else return left.search(k)
else
        if right == null: return null
        else return right.search(k)
```

Running time: *O(h)*

When a node is in the tree

Finding USA

```
27  public T contains(T x) {
28      return find(getRootNode(), x);
29  }
30  private T find(TreeNode n, T x) {
31      T result= null;
32      if(n!=null) {
33          T data = (T) n.getData();
34          if(data.compareTo(x)==0) result = data;
35          else if(data.compareTo(x)<0) result = find(n.getRightChild(), x);
36          else result = find(n.getLeftChild(), x);
37      }
38      return result;
39  }
```

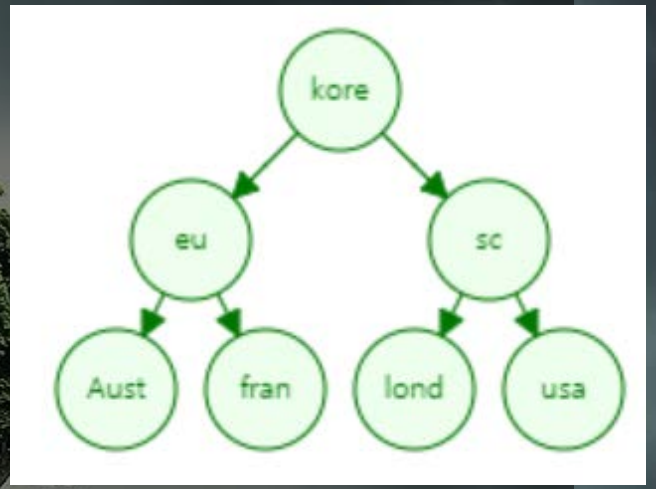



```
public class PhoneDirectory implements Comparable<PhoneDirectory>{
    private String name, phoneNum;
    public int compareTo(PhoneDirectory x) {
        return name.compareTo(x.name);
    }
    public PhoneDirectory(String name, String phoneNum) {
        this.name = name;
        this.phoneNum = phoneNum;
    }   public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPhoneNum() {
        return phoneNum;
    }
    public void setPhoneNum(String phoneNum) {
        this.phoneNum = phoneNum;
    }
    public String toString() {
        return name +": " + phoneNum;
    }
}
```

When a node is not in the tree

Finding Japan

```
27    public T contains(T x) {
28        return find(getRootNode(), x);
29    }
30    private T find(TreeNode n, T x) {
31        T result= null;
32        if(n!=null) {
33            T data = (T) n.getData();
34            if(data.compareTo(x)==0) result = data;
35            else if(data.compareTo(x)<0) result = find(n.getRightChild(), x);
36            else result = find(n.getLeftChild(), x);
37        }
38        return result;
39    }
```



Removing a node

1. if the tree is empty return null
2. Attempt to locate the node containing the target using the binary search algorithm
    1. if the target is not found return null
    2. else the target is found, so remove its node:
        1. Case 1: if the node is a leaf, replace the link in the parent with null
        2. Case 2: if the node has no left child
            1. link the parent of the node
            2. to the right (non-empty) subtree
        3. Case 3: if the node has no right child
            1. link the parent of the target
            2. to the left (non-empty) subtree
        4. Case 4: if the node has a left and a right subtree
            1. replace the node's value with the max value in the left subtree
            2. delete the max node in the left subtree

Running time: *O(h)*

Removing a leaf

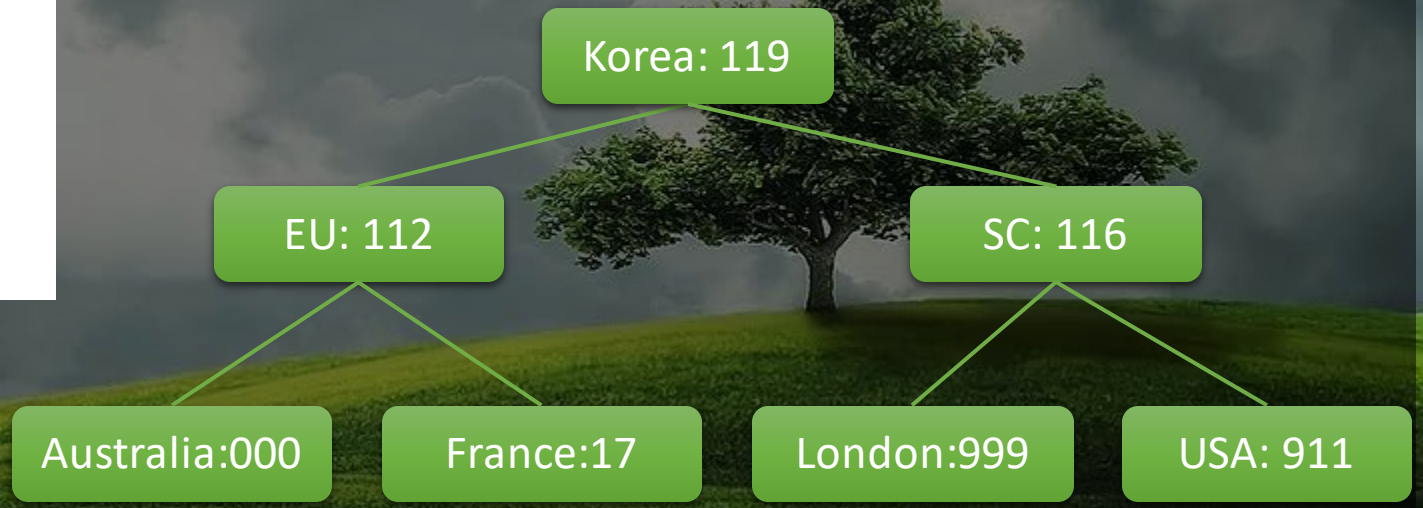
```java
public T remove(T x) {
    NodeNData result = remove(root, x);
    root = result.node;
    return result.data;
}

private NodeNData remove(LinkedTree.TreeNode<T> node, T x) {
    if(node==null) return new NodeNData(node, null);        // x is not in the tree
    NodeNData result = new NodeNData(node, node.getData());
    if(x.compareTo(node.getData())<0) {
        result = remove(node.getLeftChild(), x);
        node.setLeftChild(result.node);
        result.node = node;
    }
    else if(x.compareTo(node.getData())>0) {
        result = remove(node.getRightChild(), x);
        node.setRightChild(result.node);
        result.node = node;
    }
    else {
        if(!node.hasLeftChild()) {
            result.node=node.getRightChild();
        }
        else if(!node.hasRightChild()) {
            result.node = node.getLeftChild();
        } else {
            T temp = findMax(node.getLeftChild());
            node.setData(temp);
            node.setLeftChild(remove(node.getLeftChild(),temp).node);
        }
    }
    return result;
}
private T findMax(LinkedTree.TreeNode<T> node) {
    if(node.hasRightChild()) return findMax(node.getRightChild());
    else return (T) node.getData();
}
```

```java
private class NodeNData{
    LinkedTree.TreeNode<T> node;
    T data;
    private NodeNData(LinkedTree.TreeNode<T> n, T x) {
        node = n; data=x;
    }
}
```

Delete it

Example: removing USA



Removing a node with one child

```
public T remove(T x) {
    NodeNData result = remove(root, x);
    root = result.node;
    return result.data;
}

private NodeNData remove(LinkedTree.TreeNode<T> node, T x) {
    if(node==null) return new NodeNData(node, null);        // x is not in the tree
    NodeNData result = new NodeNData(node, node.getData());
    if(x.compareTo(node.getData())<0) {
        result = remove(node.getLeftChild(), x);
        node.setLeftChild(result.node);
        result.node = node;
    }
    else if(x.compareTo(node.getData())>0) {
        result = remove(node.getRightChild(), x);
        node.setRightChild(result.node);
        result.node = node;
    }
    else {
        if(!node.hasLeftChild()) {
            result.node=node.getRightChild();
        }
        else if(!node.hasRightChild()) {
            result.node = node.getLeftChild();
        } else {
            T temp = findMax(node.getLeftChild());
            node.setData(temp);
            node.setLeftChild(remove(node.getLeftChild(),temp).node);
        }
    }
    return result;
}
private T findMax(LinkedTree.TreeNode<T> node) {
    if(node.hasRightChild()) return findMax(node.getRightChild());
    else return (T) node.getData();
}
```

```
private class NodeNData{
    LinkedTree.TreeNode<T> node;
    T data;
    private NodeNData(LinkedTree.TreeNode<T> n, T x) {
        node = n; data=x;
    }
}
```

Delete it

Example: removing SC

Removing a node with one child

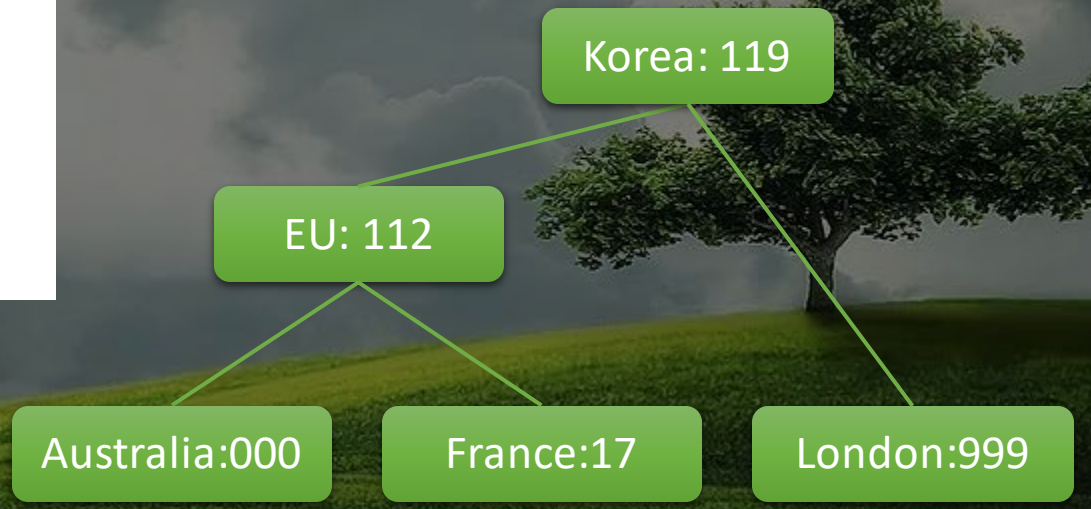
```java
public T remove(T x) {
    NodeNData result = remove(root, x);
    root = result.node;
    return result.data;
}

private NodeNData remove(LinkedTree.TreeNode<T> node, T x) {
    if(node==null) return new NodeNData(node, null);        // x is not in the tree
    NodeNData result = new NodeNData(node, node.getData());
    if(x.compareTo(node.getData())<0) {
        result = remove(node.getLeftChild(), x);
        node.setLeftChild(result.node);
        result.node = node;
    }
    else if(x.compareTo(node.getData())>0) {
        result = remove(node.getRightChild(), x);
        node.setRightChild(result.node);
        result.node = node;
    }
    else {
        if(!node.hasLeftChild()) {
            result.node=node.getRightChild();
        }
        else if(!node.hasRightChild()) {
            result.node = node.getLeftChild();
        } else {
            T temp = findMax(node.getLeftChild());
            node.setData(temp);
            node.setLeftChild(remove(node.getLeftChild(),temp).node);
        }
    }
    return result;
}
private T findMax(LinkedTree.TreeNode<T> node) {
    if(node.hasRightChild()) return findMax(node.getRightChild());
    else return (T) node.getData();
}
```

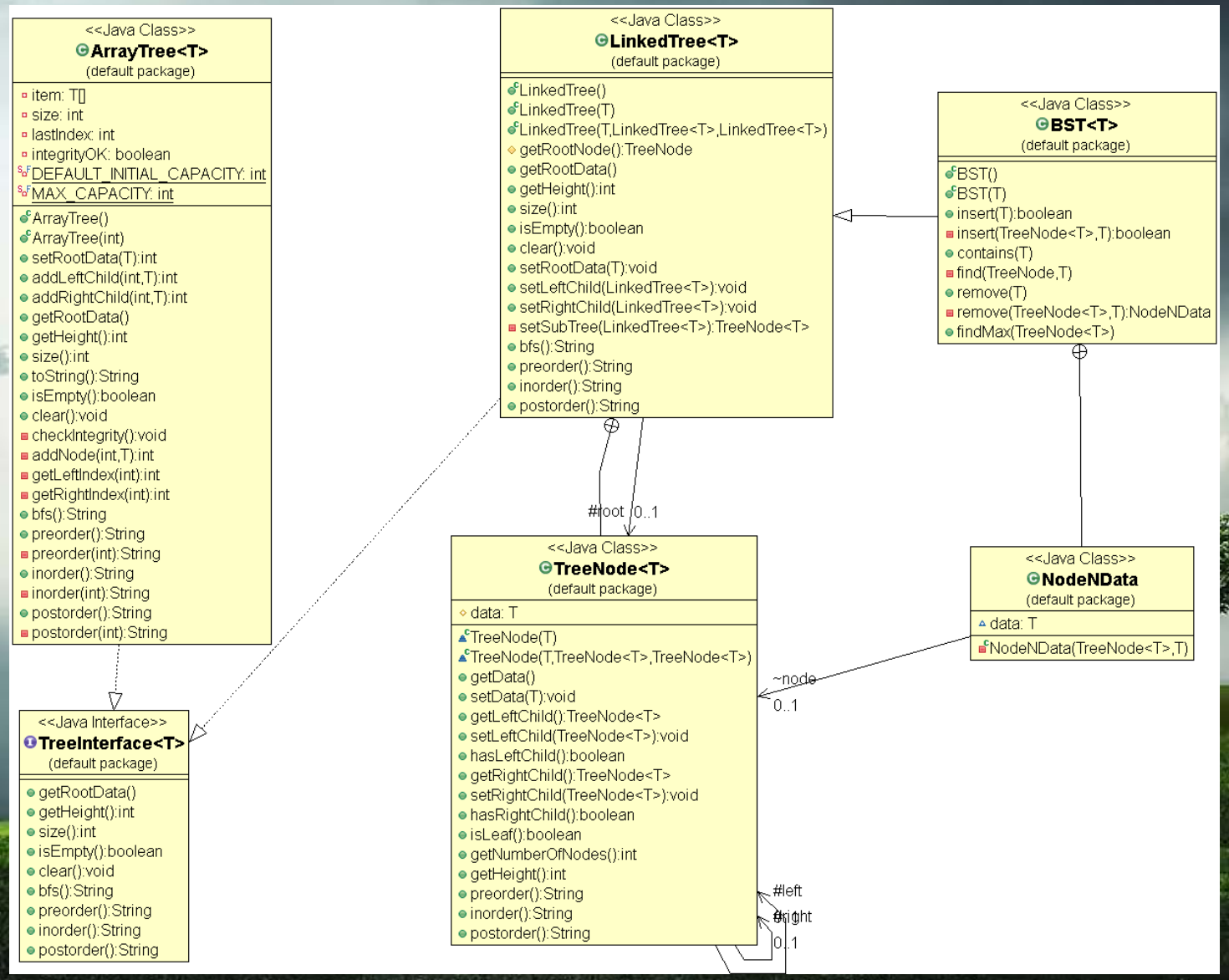
```java
private class NodeNData{
    LinkedTree.TreeNode<T> node;
    T data;
    private NodeNData(LinkedTree.TreeNode<T> n, T x) {
        node = n; data=x;
    }
}
```

Delete it

Example: removing SC



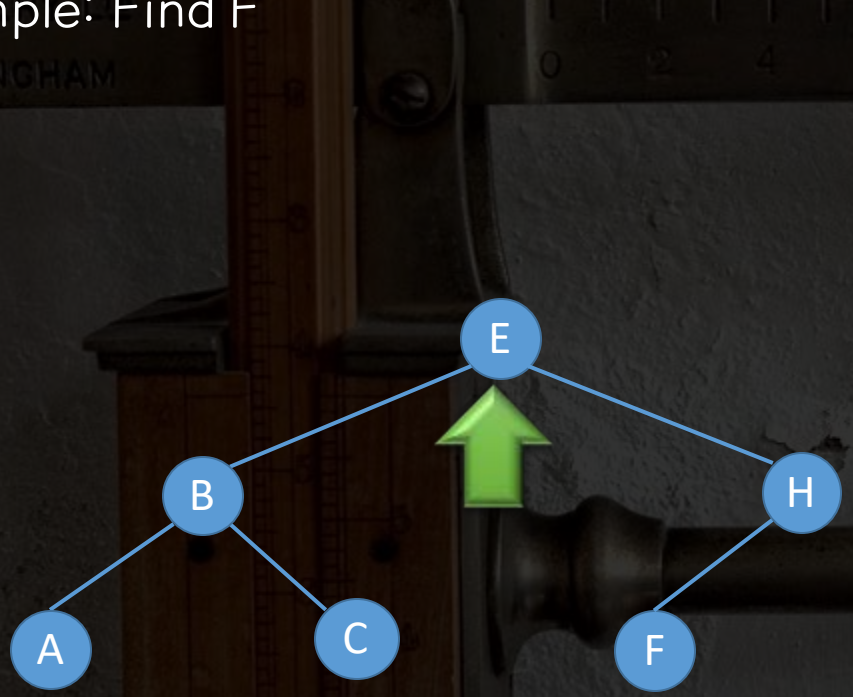
UML model

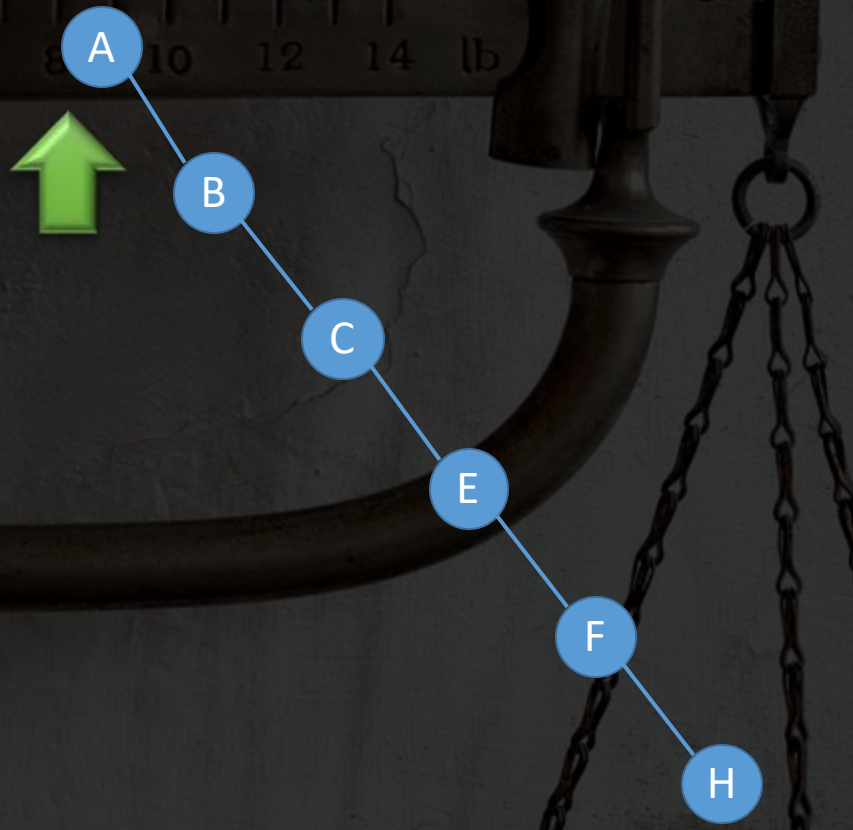

<<Java Class>>
ArrayTree<T>
(default package)
item: T[]
size: int
lastIndex: int
integrityOK: boolean
DEFAULT_INITIAL_CAPACITY: int
MAX_CAPACITY: int
ArrayTree()
ArrayTree(int)
setRootData(T):int
addLeftChild(int,T):int
addRightChild(int,T):int
getRootData()
getHeight():int
size():int
toString():String
isEmpty():boolean
clear():void
checkIntegrity():void
addNode(int,T):int
getLeftIndex(int):int
getRightIndex(int):int
bfs():String
preorder():String
preorder(int):String
inorder():String
inorder(int):String
postorder():String
postorder(int):String
<<Java Class>>
LinkedTree<T>
(default package)
LinkedTree()
LinkedTree(T)
LinkedTree(T,LinkedTree<T>,LinkedTree<T>)
getRootNode():TreeNode
getRootData()
getHeight():int
size():int
isEmpty():boolean
clear():void
setRootData(T):void
setLeftChild(LinkedTree<T>):void
setRightChild(LinkedTree<T>):void
setSubTree(LinkedTree<T>):TreeNode<T>
bfs():String
preorder():String
inorder():String
postorder():String
<<Java Class>>
BST<T>
(default package)
BST()
BST(T)
insert(T):boolean
insert(TreeNode<T>,T):boolean
contains(T)
find(TreeNode,T)
remove(T)
remove(TreeNode<T>,T):NodeNData
findMax(TreeNode<T>)
#root
0..1
<<Java Class>>
TreeNode<T>
(default package)
data: T
TreeNode(T)
TreeNode(T,TreeNode<T>,TreeNode<T>)
getData()
setData(T):void
getLeftChild():TreeNode<T>
setLeftChild(TreeNode<T>):void
hasLeftChild():boolean
getRightChild():TreeNode<T>
setRightChild(TreeNode<T>):void
hasRightChild():boolean
isLeaf():boolean
getNumberOfNodes():int
getHeight():int
preorder():String
inorder():String
postorder():String
<<Java Class>>
NodeNData
(default package)
data: T
NodeNData(TreeNode<T>,T)
~node
0..1
<<Java Interface>>
TreeInterface<T>
(default package)
getRootData()
getHeight():int
size():int
isEmpty():boolean
clear():void
bfs():String
preorder():String
inorder():String
postorder():String
#left
#right
0..1

AVL Tree

Motivation

- Performance of a BST depends on the height of the tree.
- Example: Find F

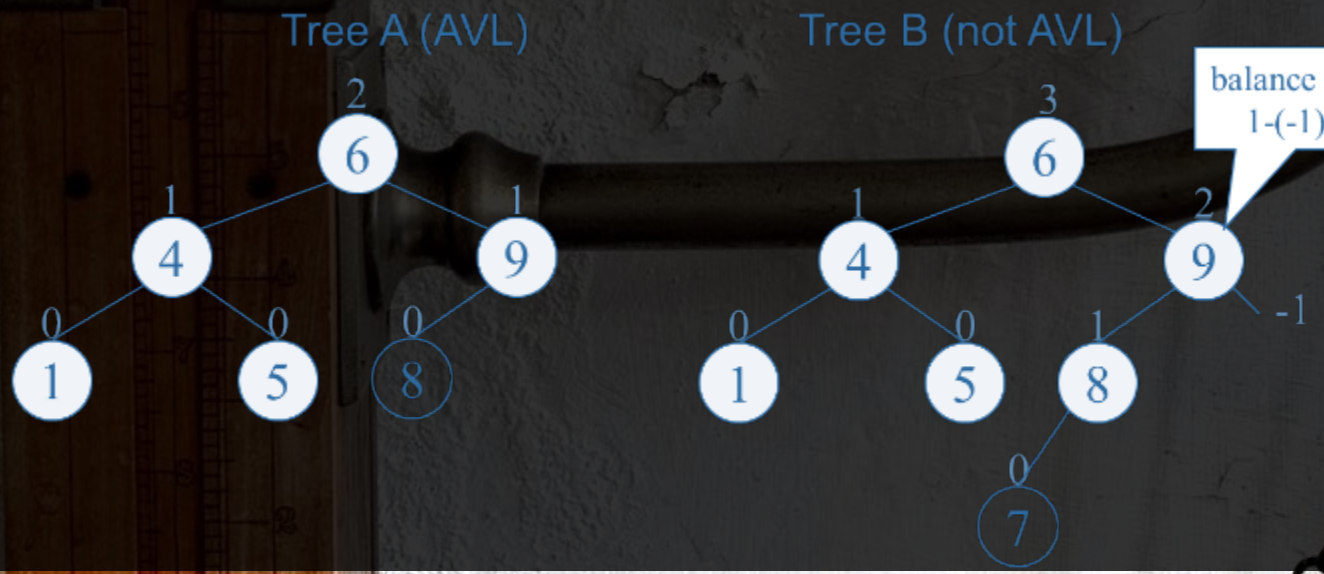




AVL Tree

- Adelson-Velskii and Landis' tree, named after the inventors
- a self-balancing binary search tree
- the heights of the two child subtrees of any node differ by at most one

BF = 왼쪽 subtree 높이 - 오른쪽 subtree 높이



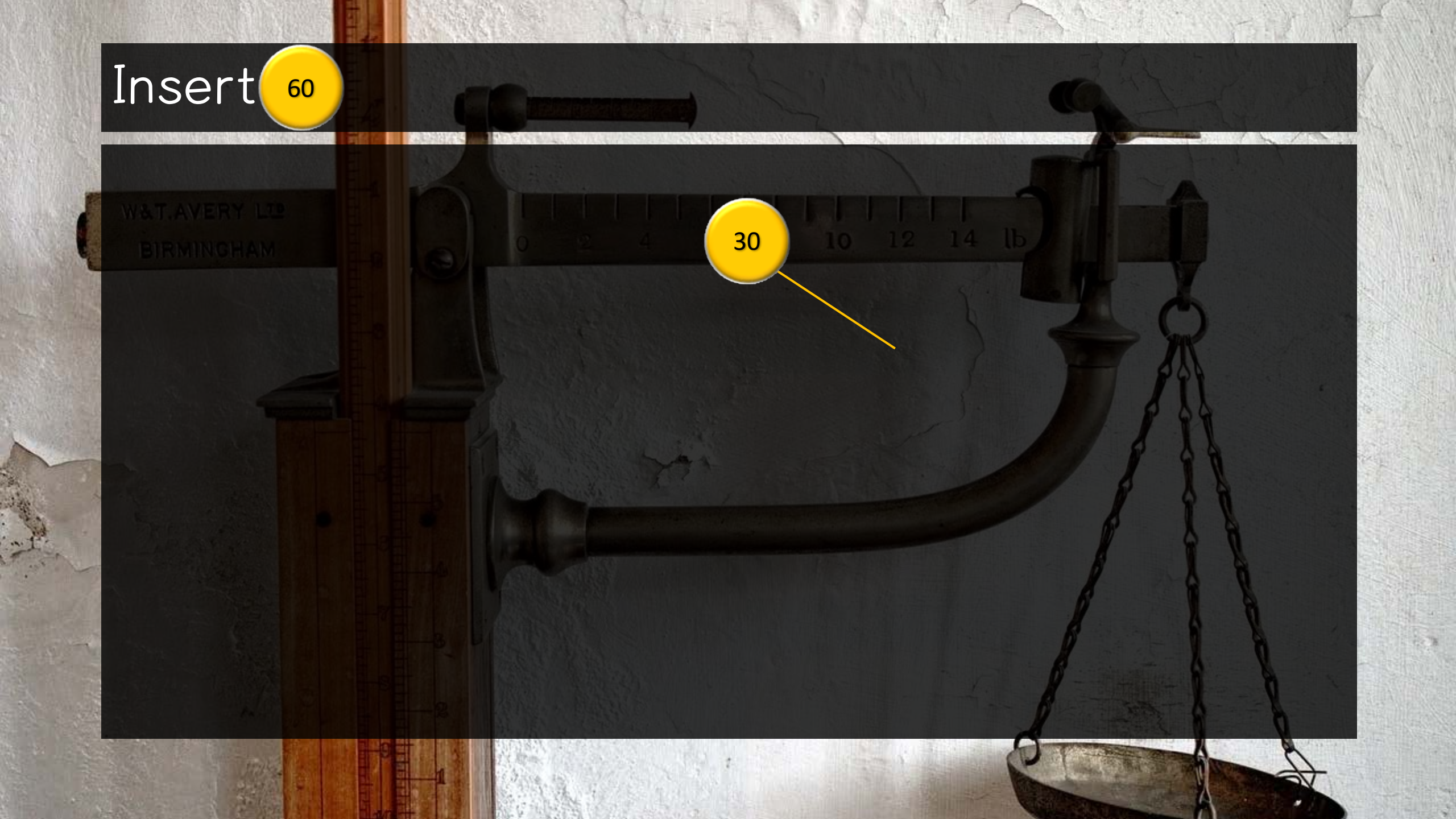
insertion

- Insertion is as in a binary search tree
- Rebalance if it is needed
  - Outside Cases (require single rotation) :
    - Insertion into left subtree of left child of a.
    - Insertion into right subtree of right child of a.
  - Inside Cases (require double rotation) :
    - Insertion into right subtree of left child of a.
    - Insertion into left subtree of right child of a.

- Visualization: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
  - Insert 50→60→70→90→30

Insert 30

Insert

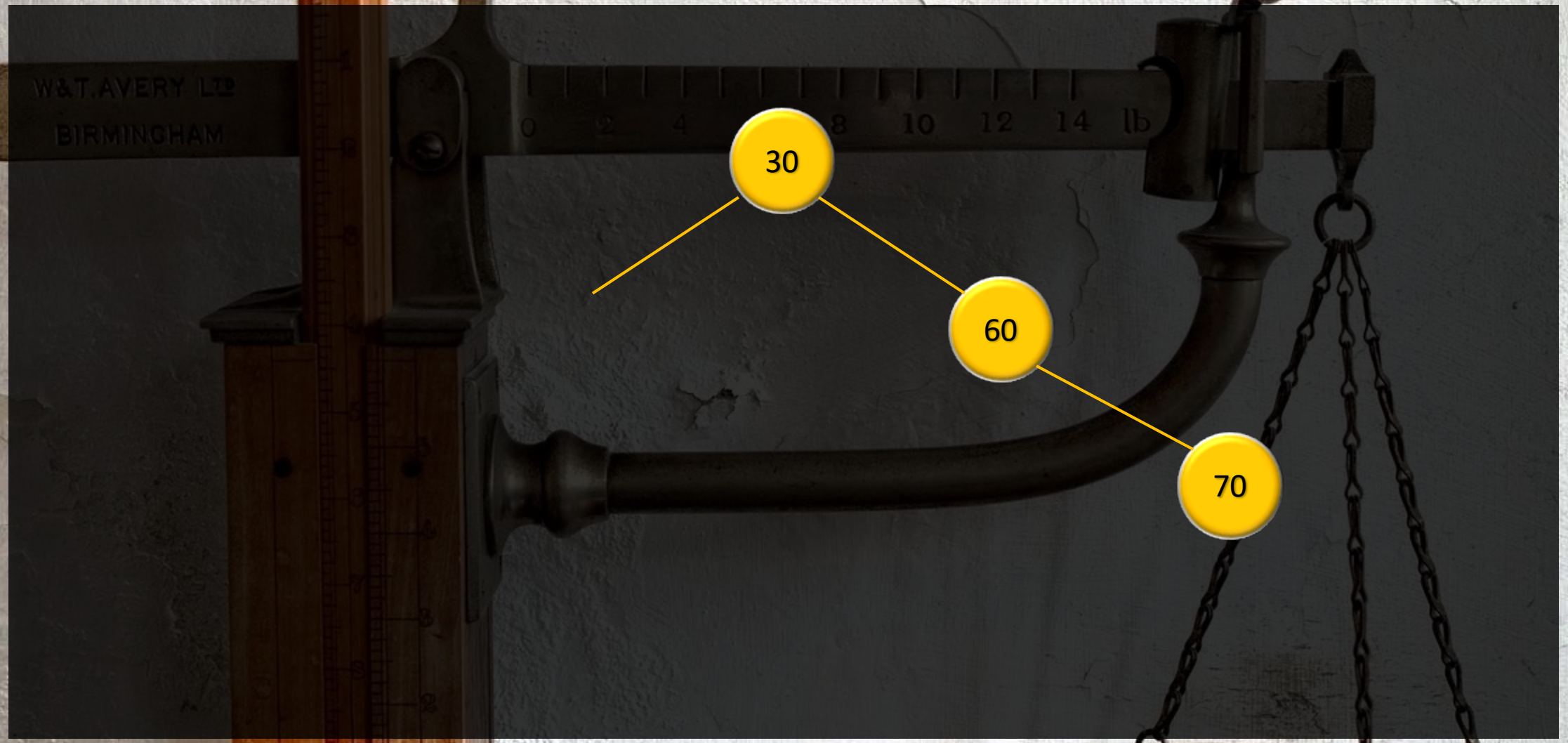


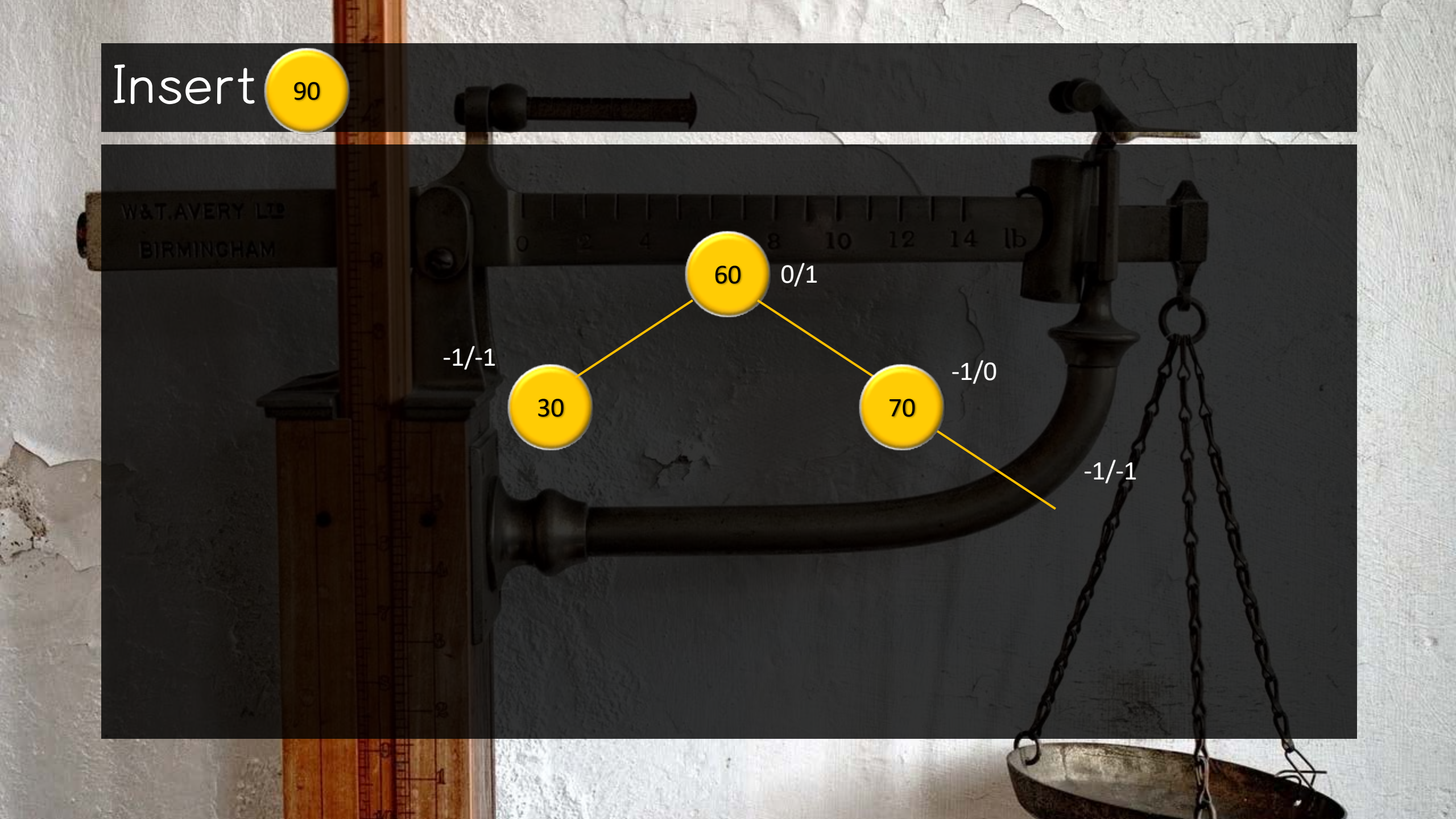
Insert 70

The right child of 30 is heavier than its left.

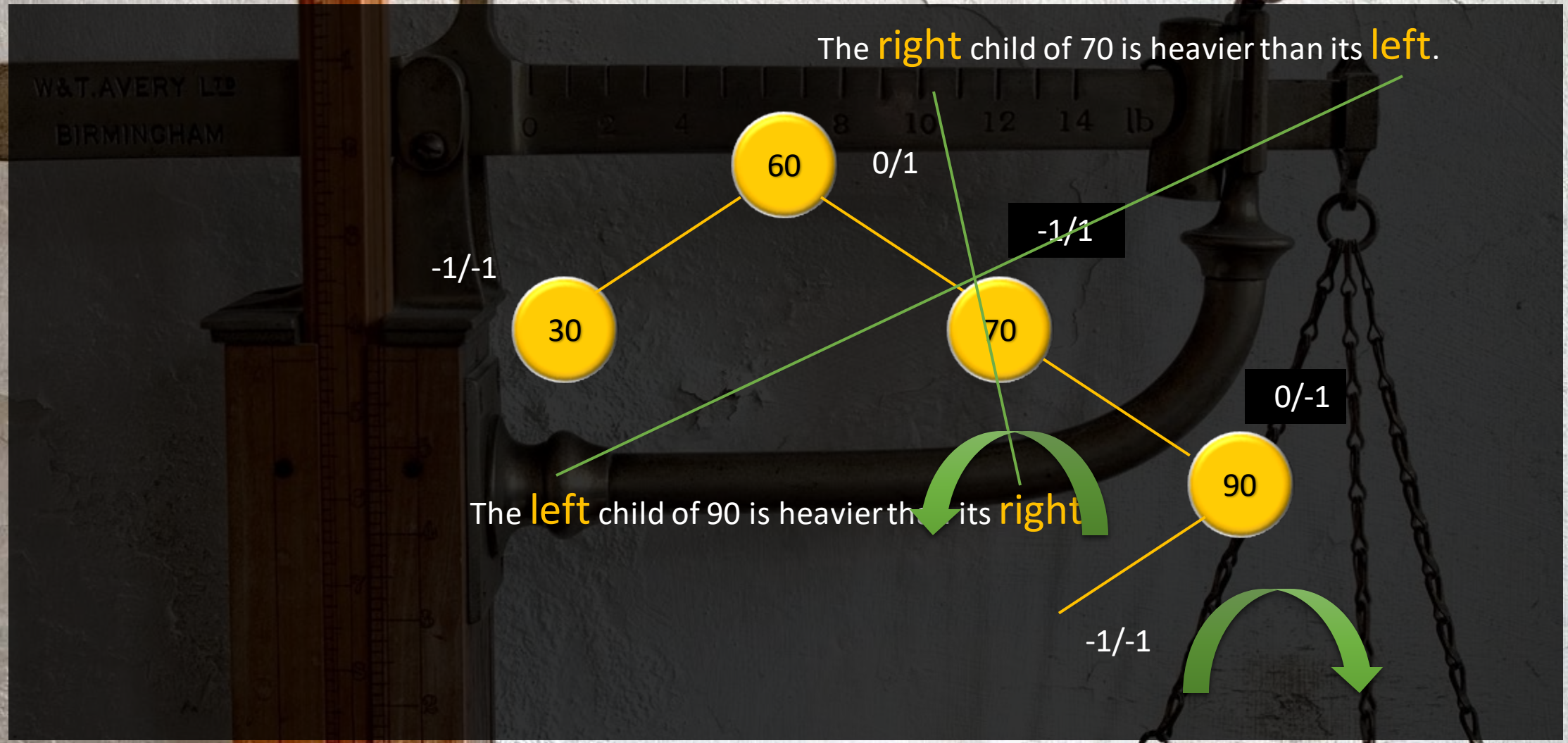
The right child of 60 is heavier than its left.

Insert 70

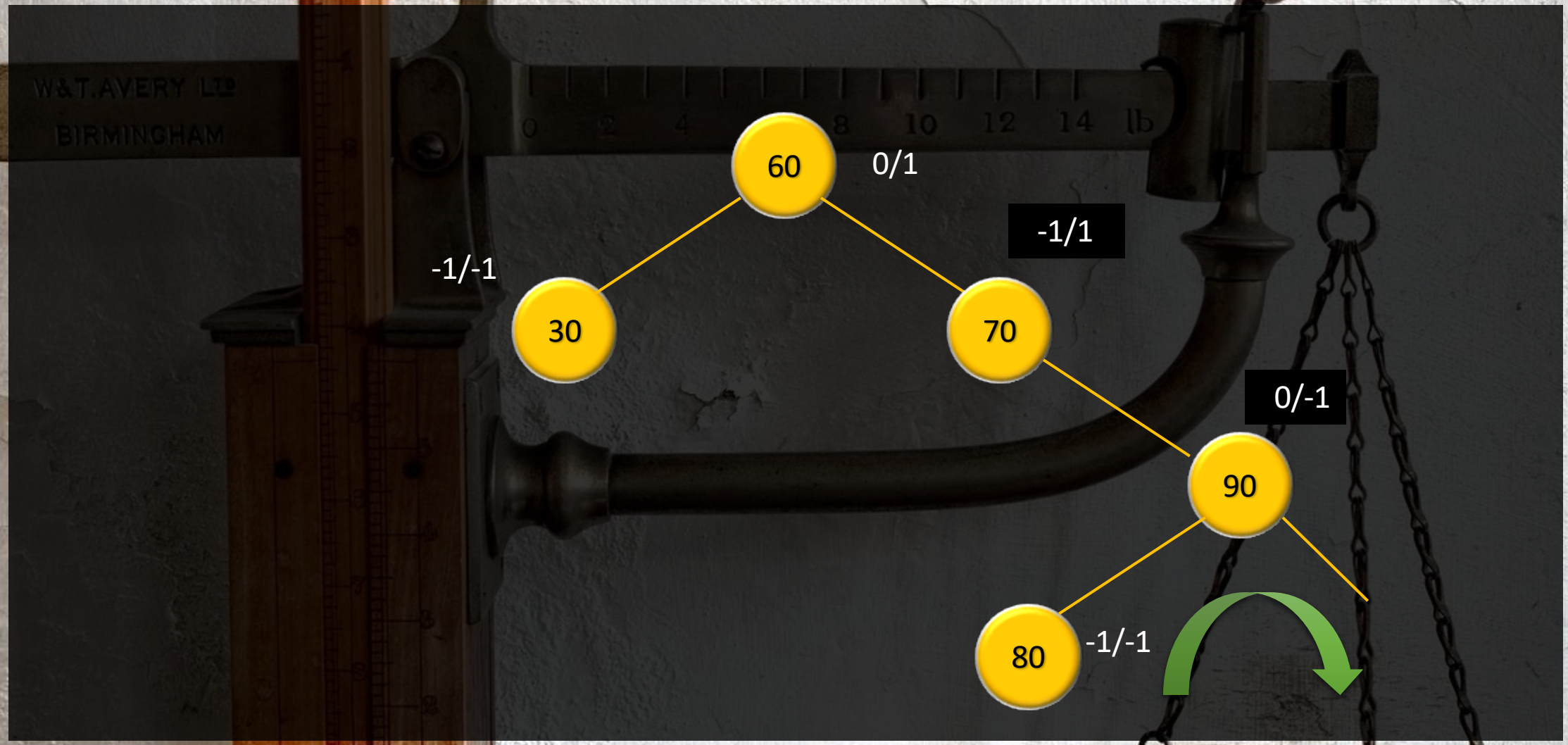



Insert
90
60
0/1
-1/-1
30
70
-1/0
-1/-1

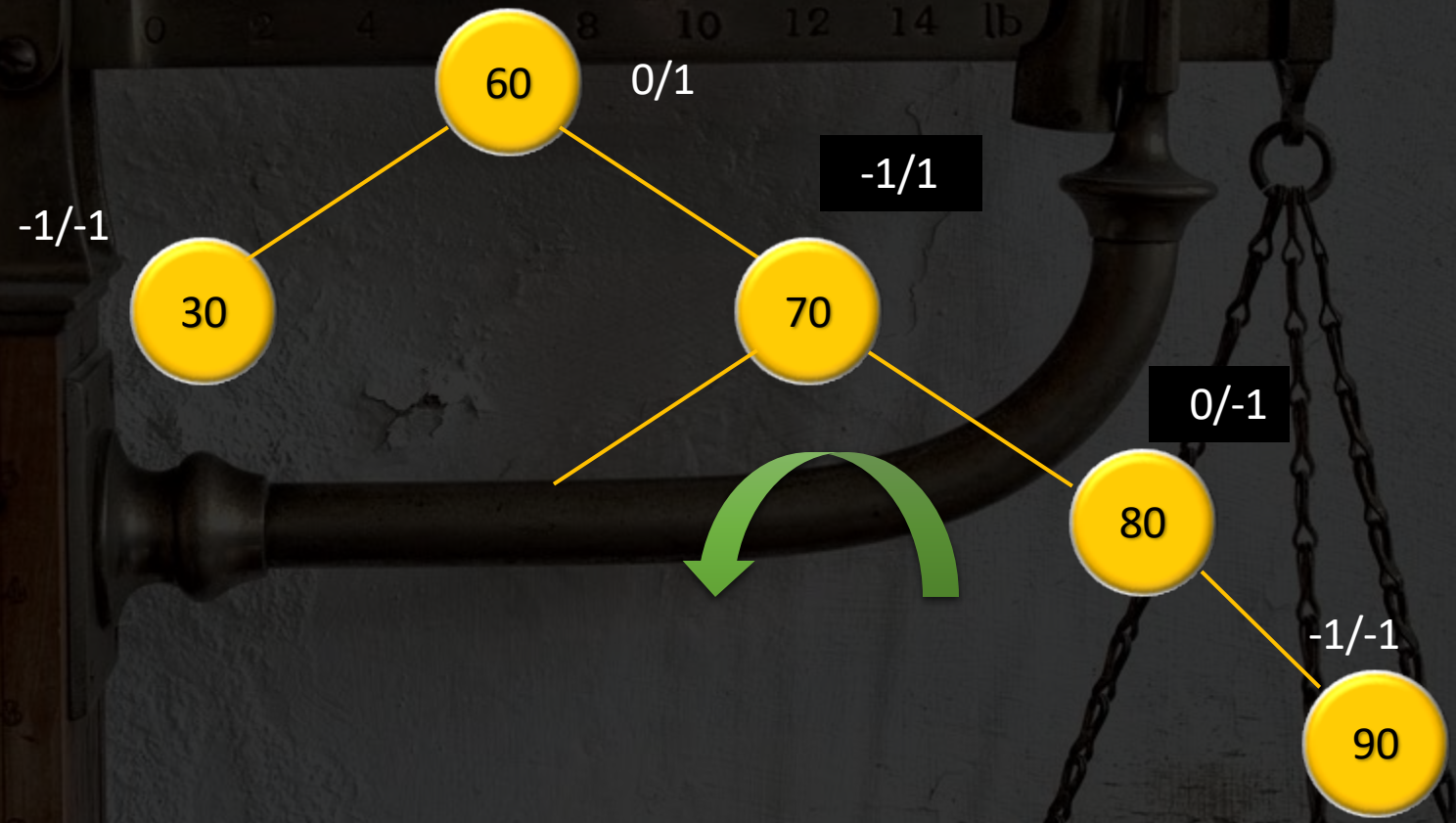
Insert 80


The right child of 70 is heavier than its left.
60
0/1
-1/1
-1/-1
30
70
0/-1
90
The left child of 90 is heavier th its right
-1/-1

Insert 80



Insert 80



https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

Thank you