



# Estructura de Datos

Maria C. Torres

# Maria C. Torres

Ing. Electrónica (UNAL)

M.E. Ing. Eléctrica (UPRM)

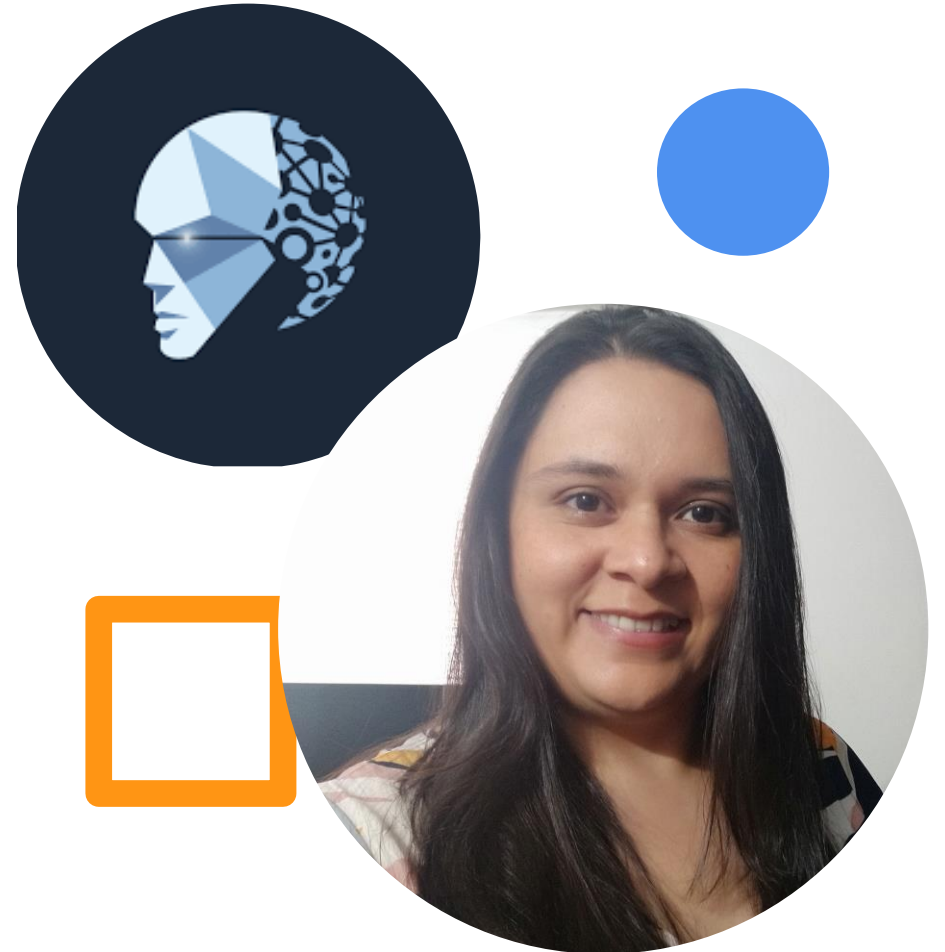
Ph.D. Ciencias e Ingeniería de la Computación y la Información (UPRM)


Profesora asociada

Dpto. Ciencias de la Computación y la Decisión


[mctorresm@unal.edu.co](mailto:mctorresm@unal.edu.co)

HORARIO DE ATENCIÓN: Martes 10:00 am a 12:00 m – Oficina 313 M8A





# Contenido del Curso

- 
- ☐ Introducción: revisión fundamentos y POO
  - ☐ Análisis de complejidad
  - ☐ Arreglos
  - ☐ Listas enlazadas
  - ☐ Pilas y colas
  - ☐ Heap
  - ☐ **Arboles binarios**
  - ☐ Tablas hash
  - ☐ Grafos



# Árbol Binario de Búsqueda

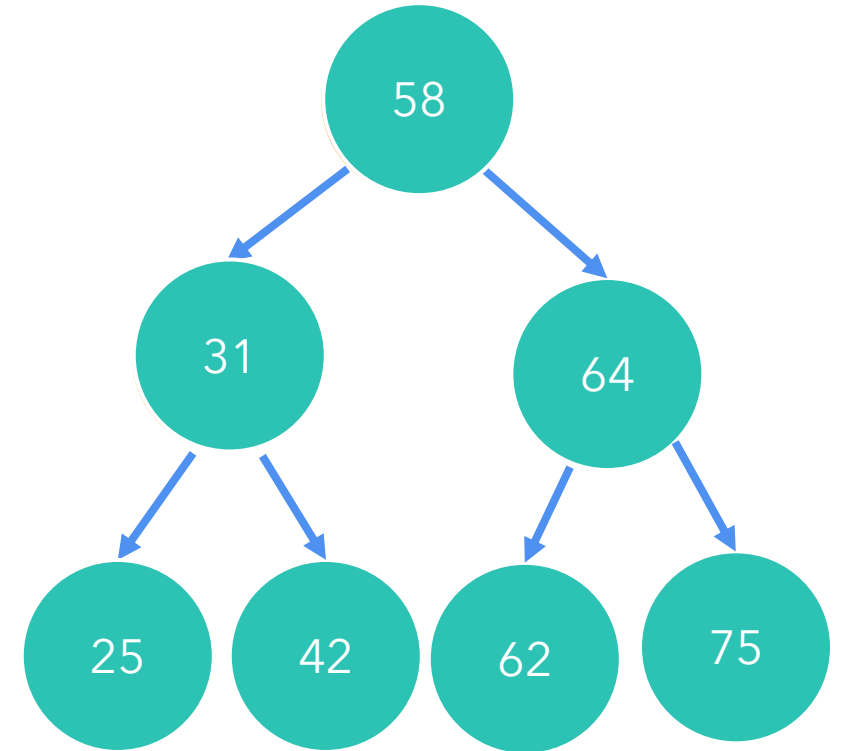
- ☐ Árboles
- ☐ Árboles binarios
- ☐ **Árboles binarios de búsqueda**
- ☐ Árboles balanceados

# Arboles binarios de búsqueda - ABB

## Definición:

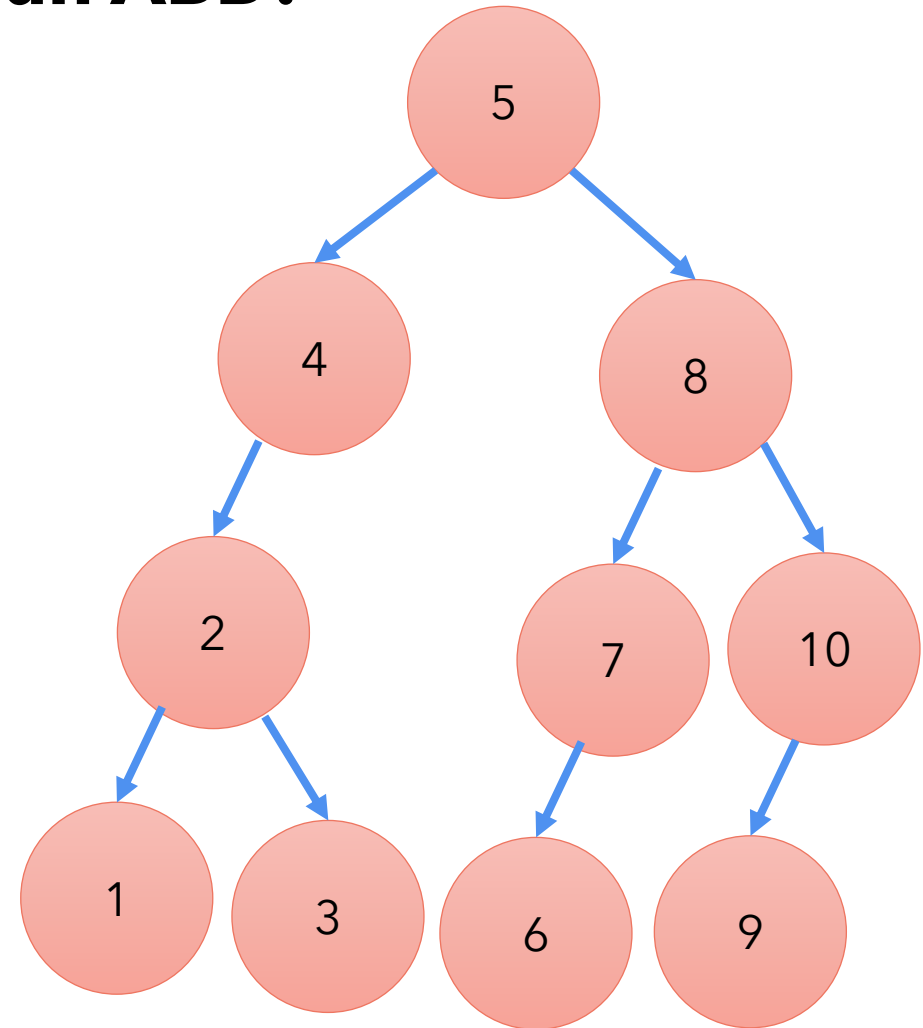
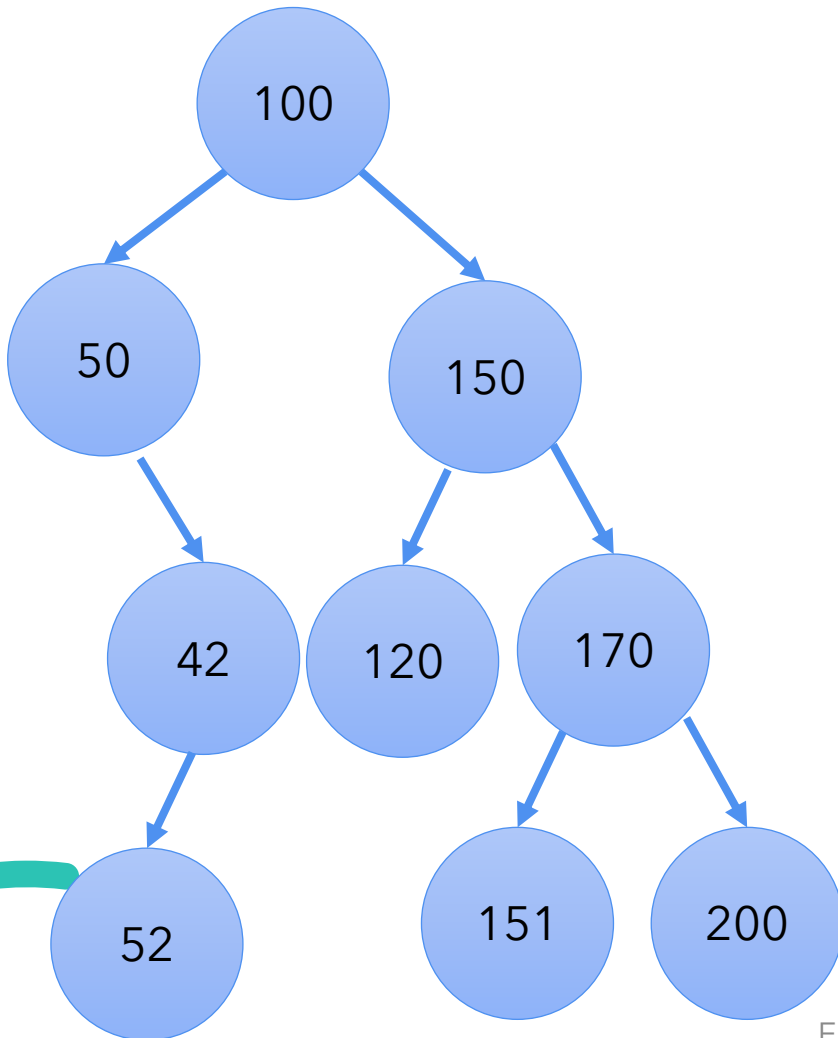
Dado un conjunto  $S$  tal que sus elementos tienen una relación de orden. Un árbol binario de búsqueda (abb) es un árbol  $T$  que:

- ❑ En cada nodo  $v$  almacena un elemento de  $S$
- ❑ Todos los elementos almacenados en el subárbol izquierdo de  $v$  son menores o iguales al elemento de  $v$
- ❑ Todos los elementos almacenados en el subárbol derecho de  $v$  son mayores o iguales al elemento de  $v$



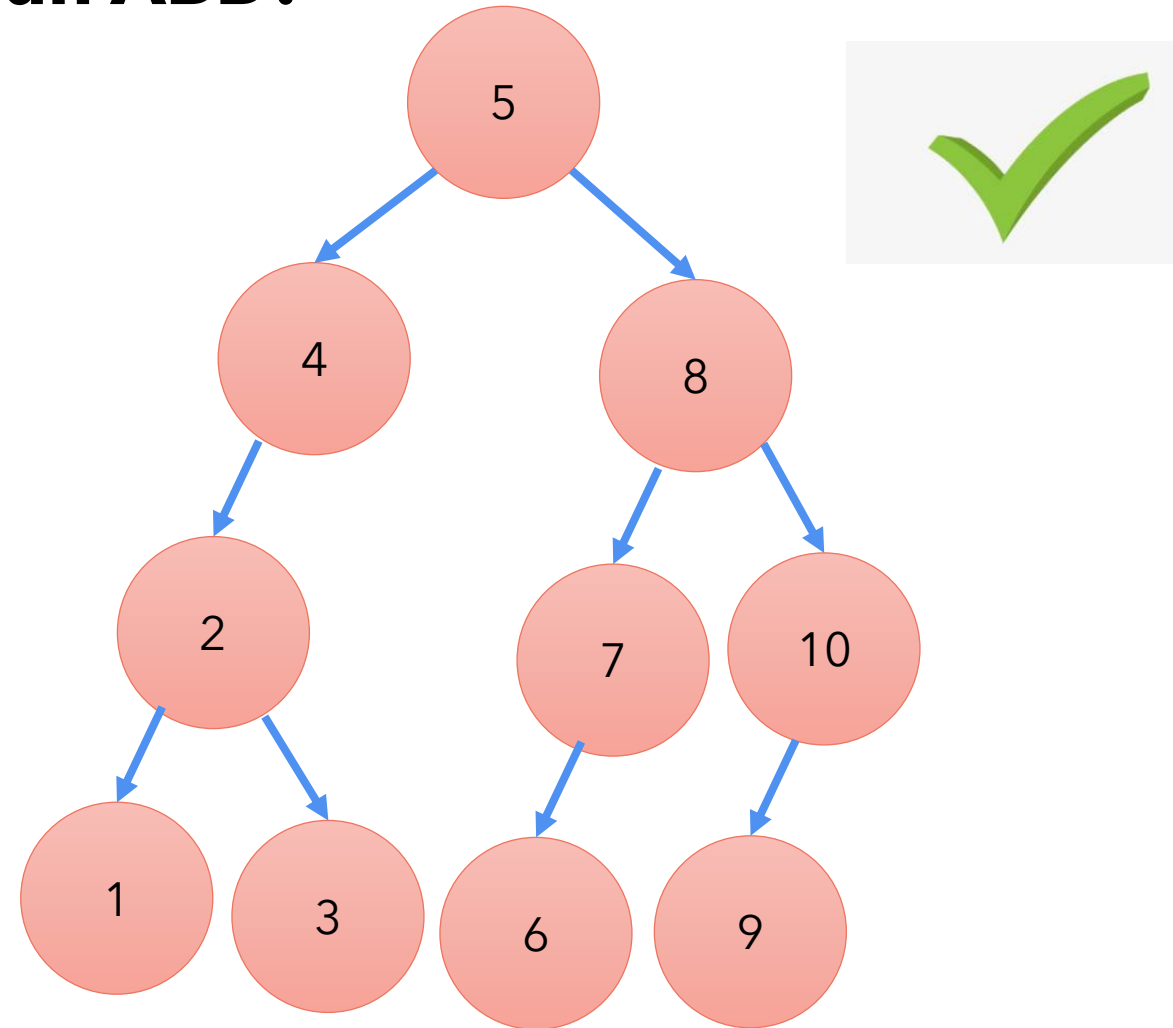
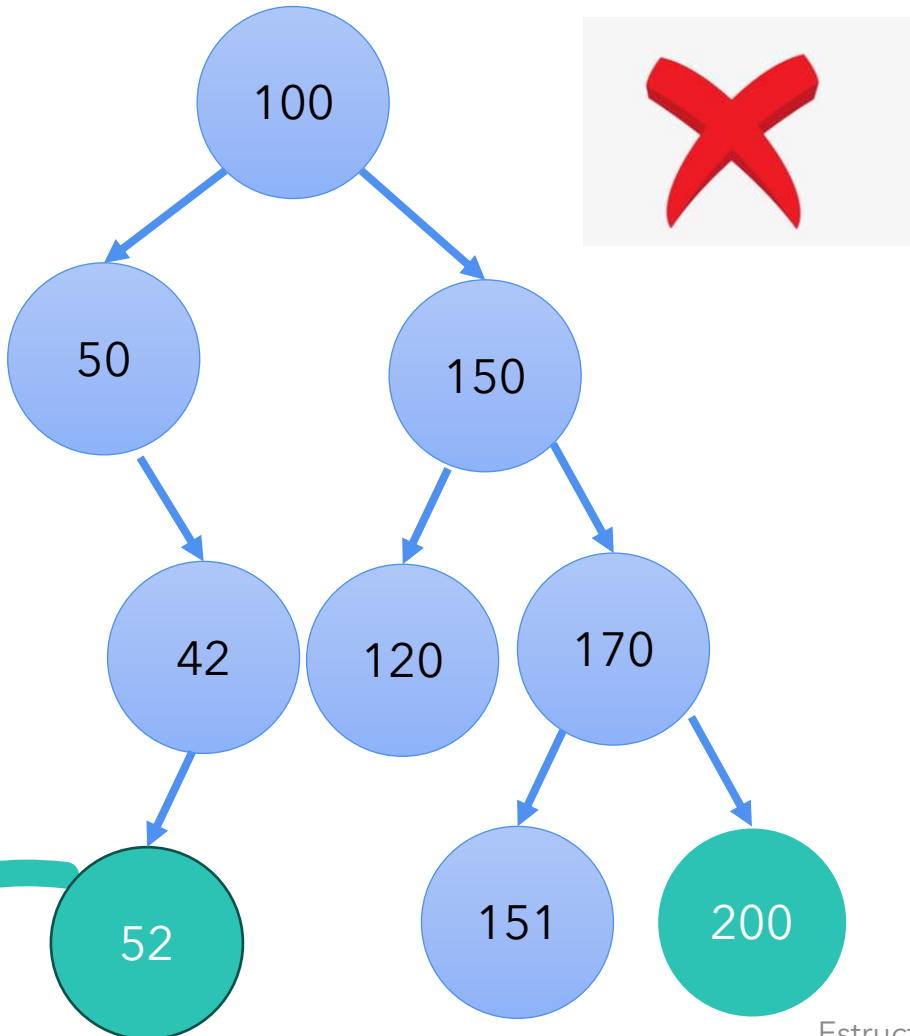
# Arboles binarios de búsqueda - ABB

**Cual de los siguientes arboles es un ABB?**



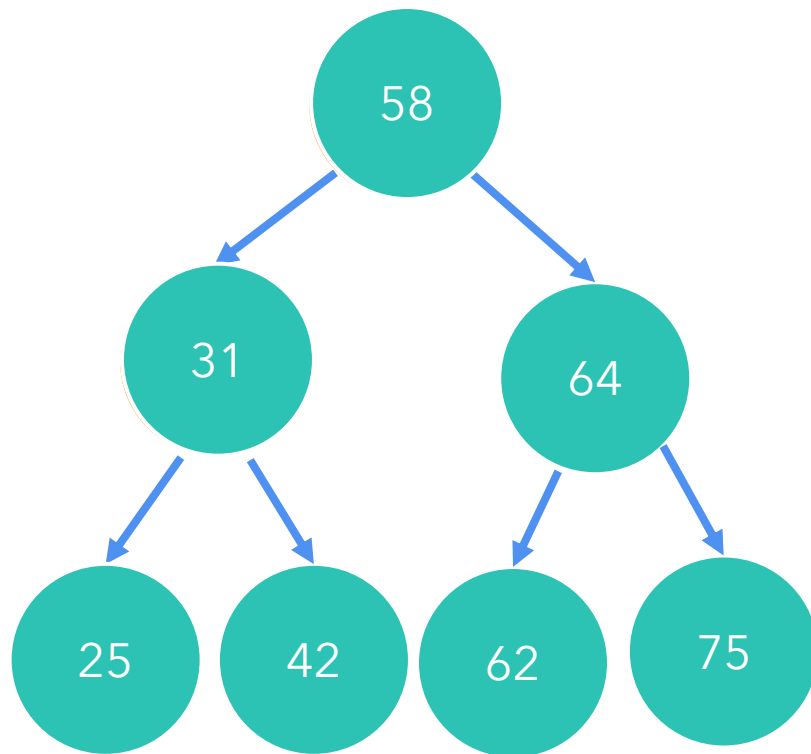
# Arboles binarios de búsqueda - ABB

**Cual de los siguientes arboles es un ABB?**

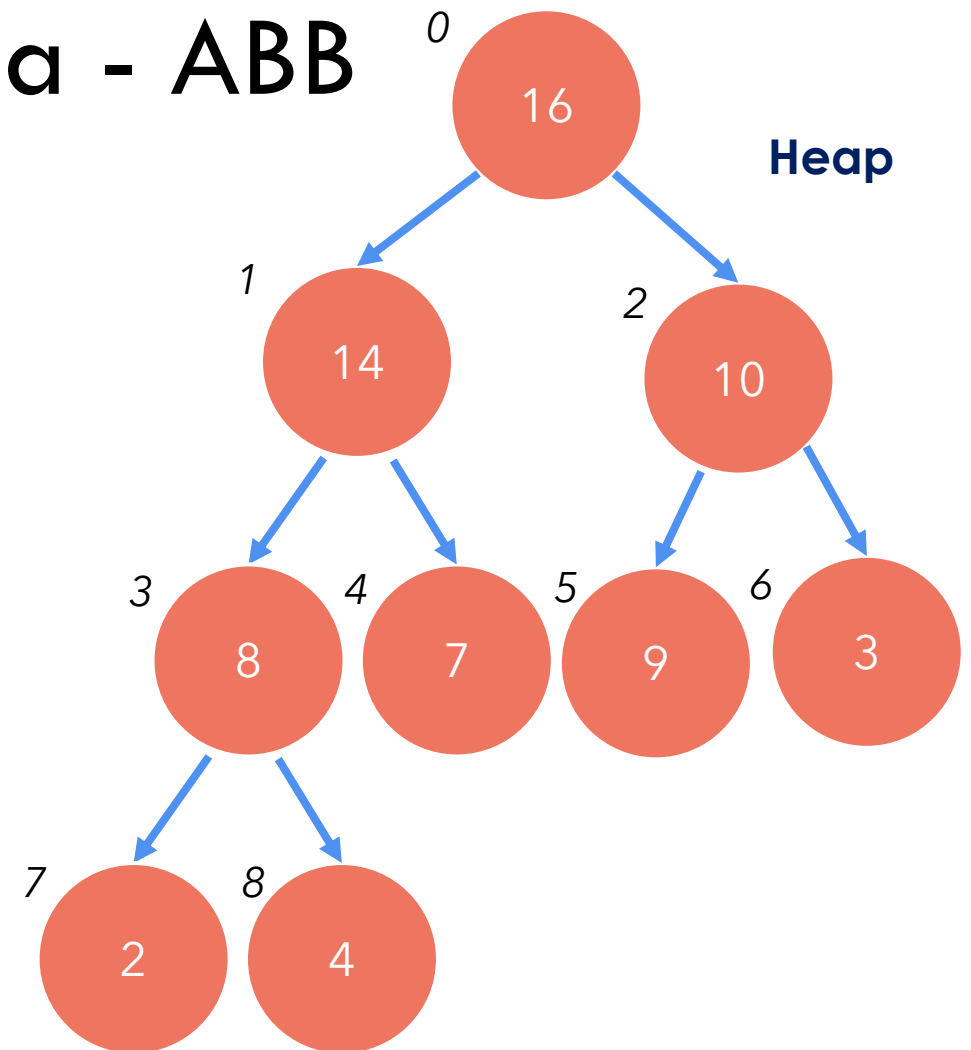


# Arboles binarios de búsqueda - ABB

## Diferencia entre ABB y HEAP



**ABB**



**Heap**

16	14	10	8	7	9	3	2	4	
0	1	2	3	4	5	6	7	8	9



# Arboles binarios de búsqueda - ABB

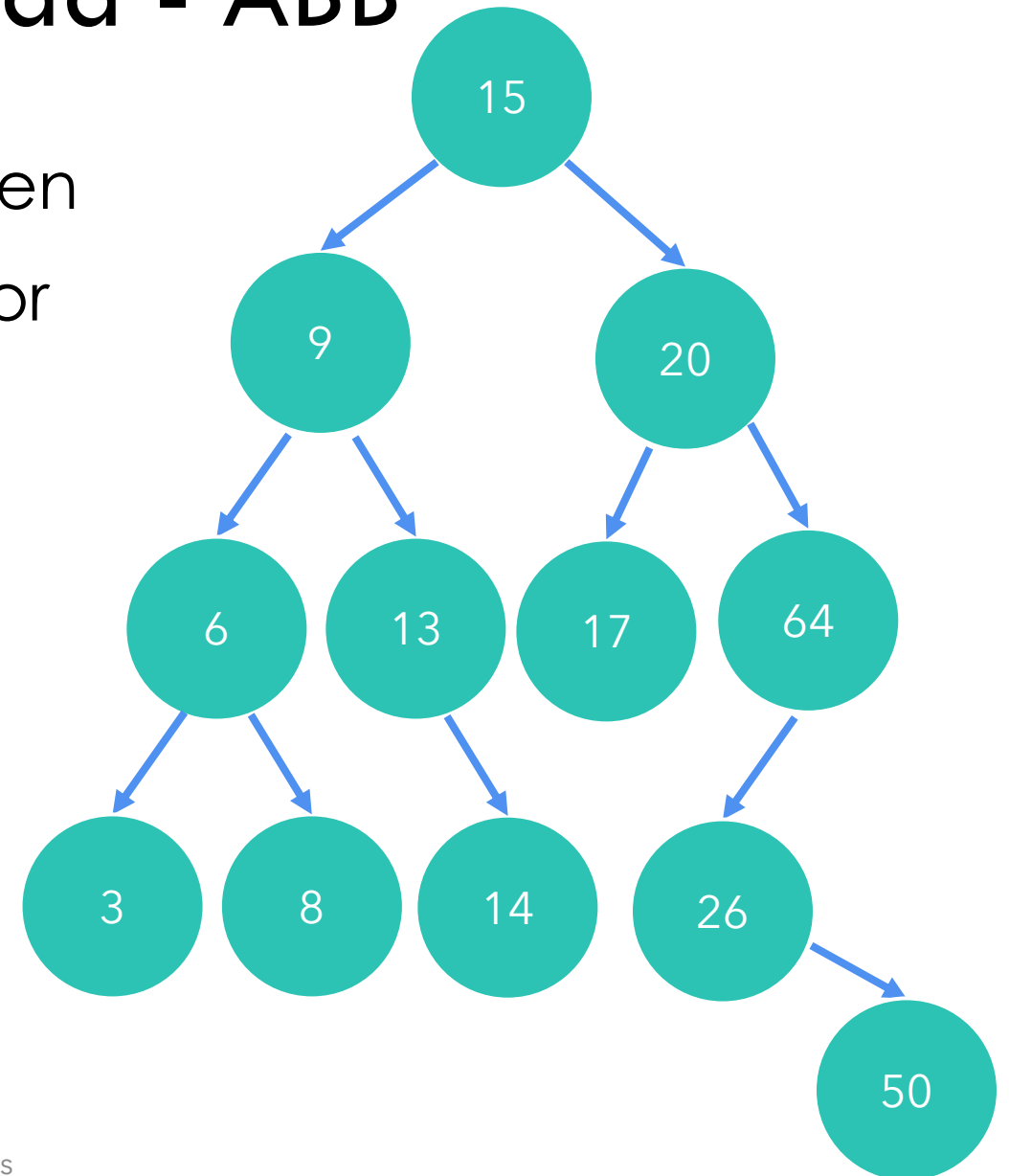
## Propiedades

- ❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=



# Arboles binarios de búsqueda - ABB

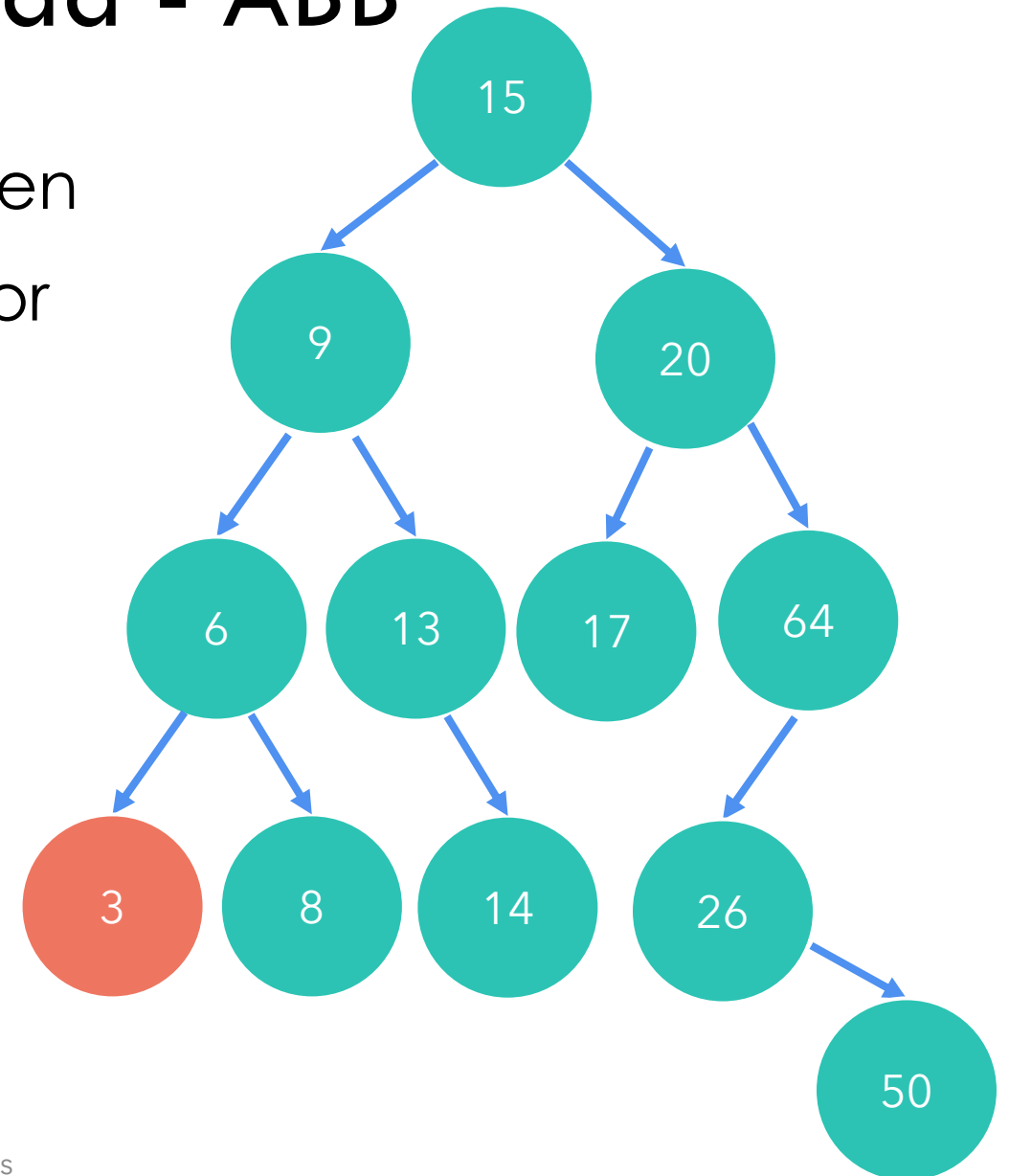
## Propiedades

❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=3-



# Arboles binarios de búsqueda - ABB

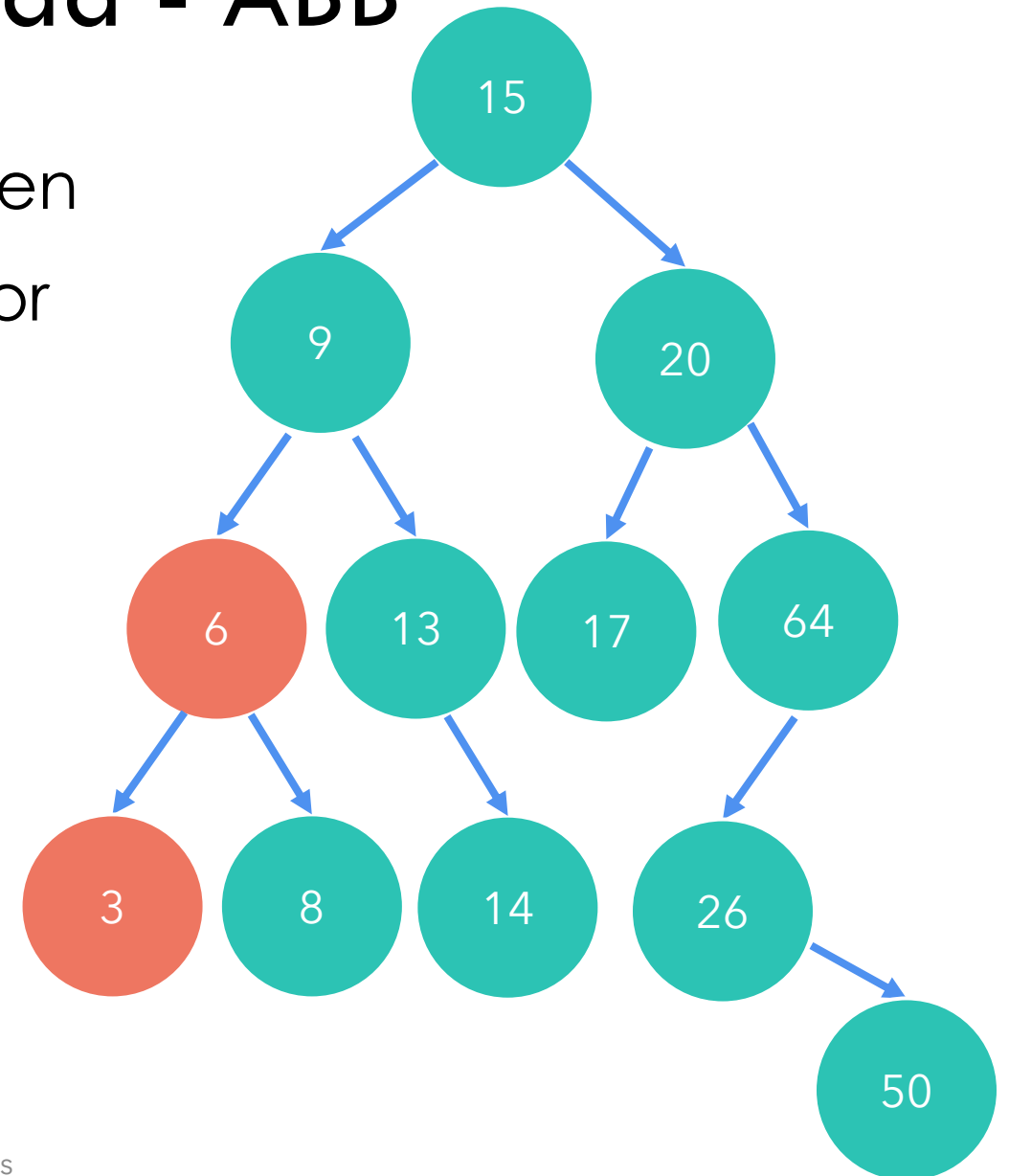
## Propiedades

- ❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=3-6-



# Arboles binarios de búsqueda - ABB

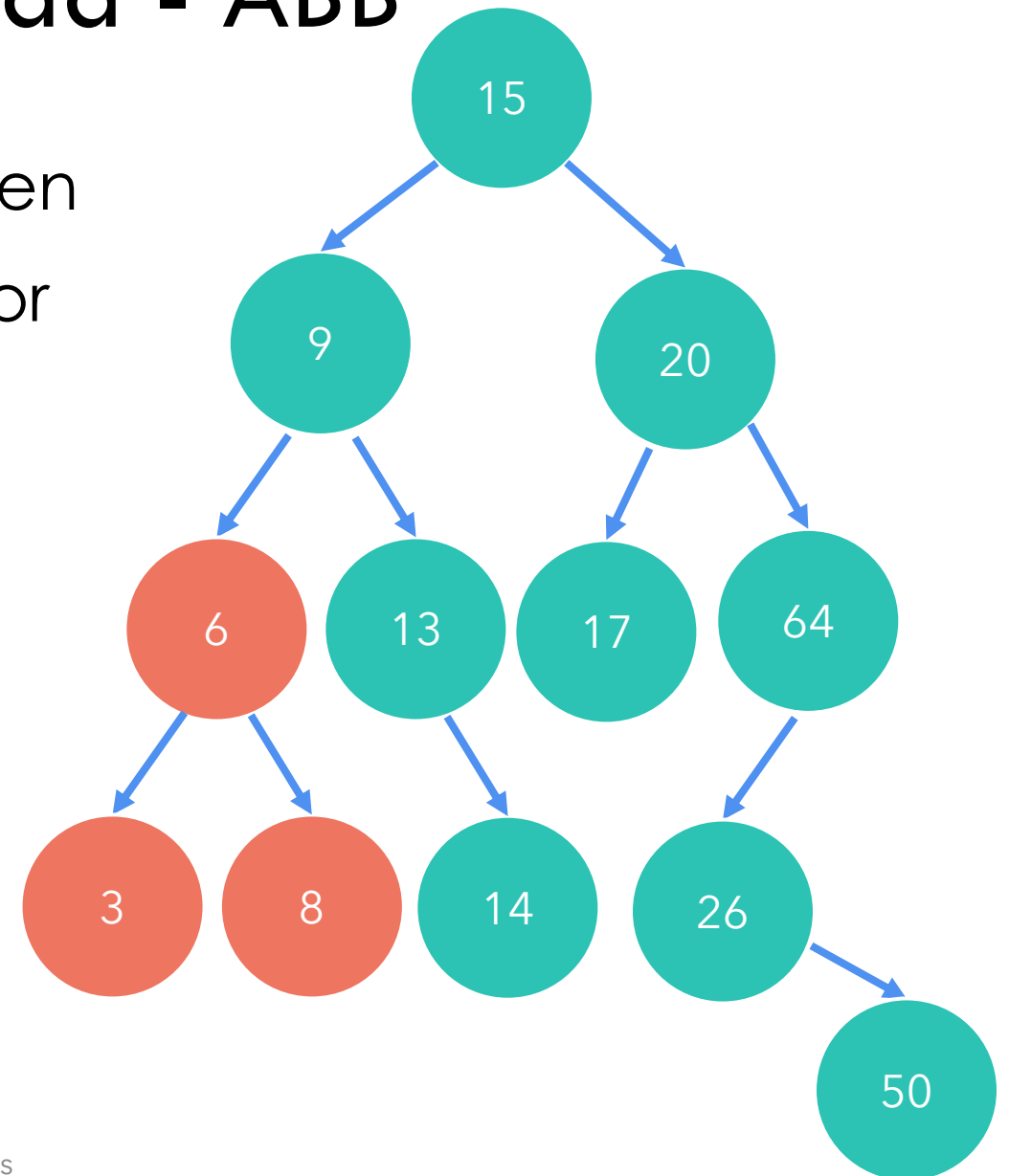
## Propiedades

❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=3-6-8-



# Arboles binarios de búsqueda - ABB

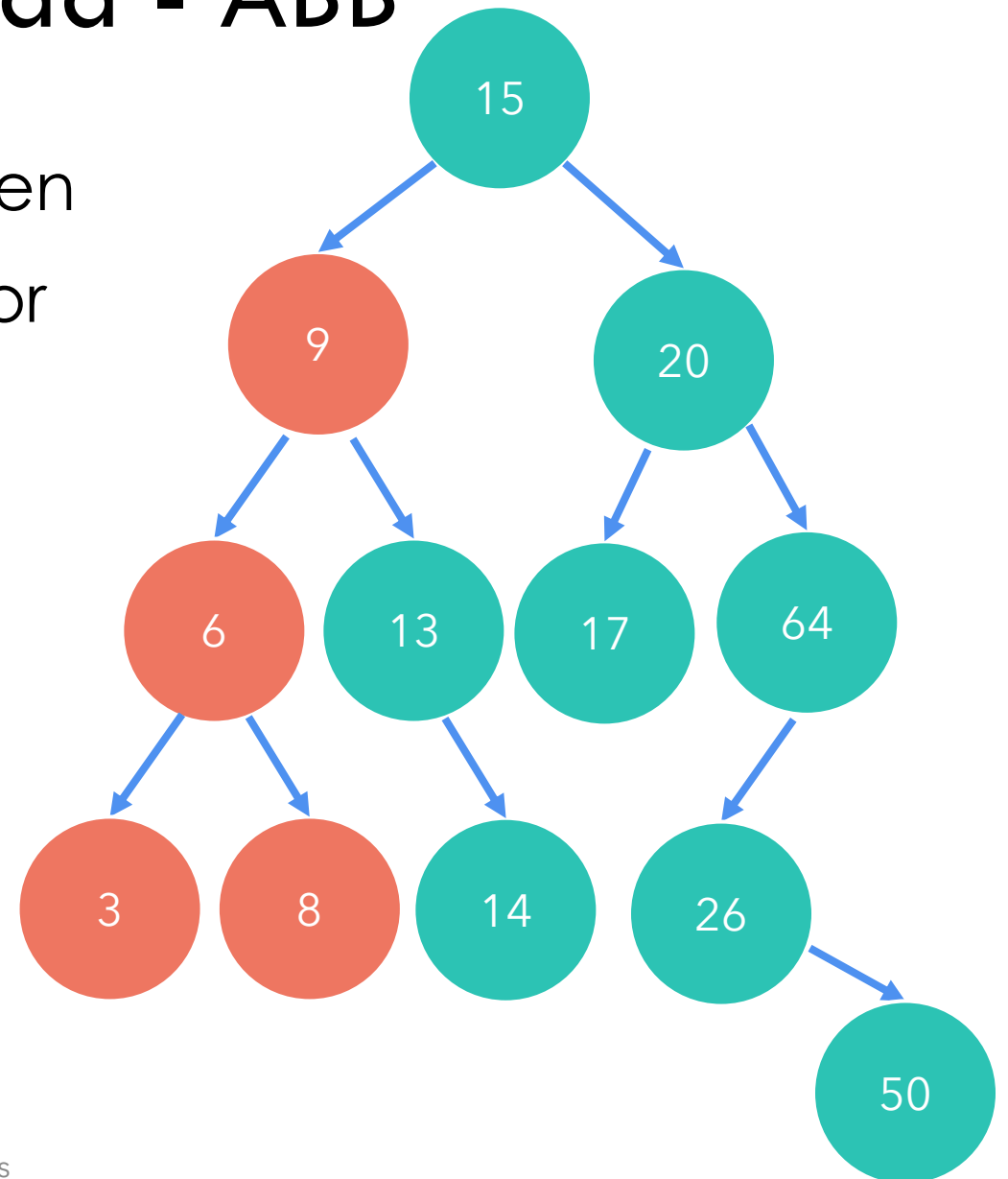
## Propiedades

❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=3-6-8-9-



# Arboles binarios de búsqueda - ABB

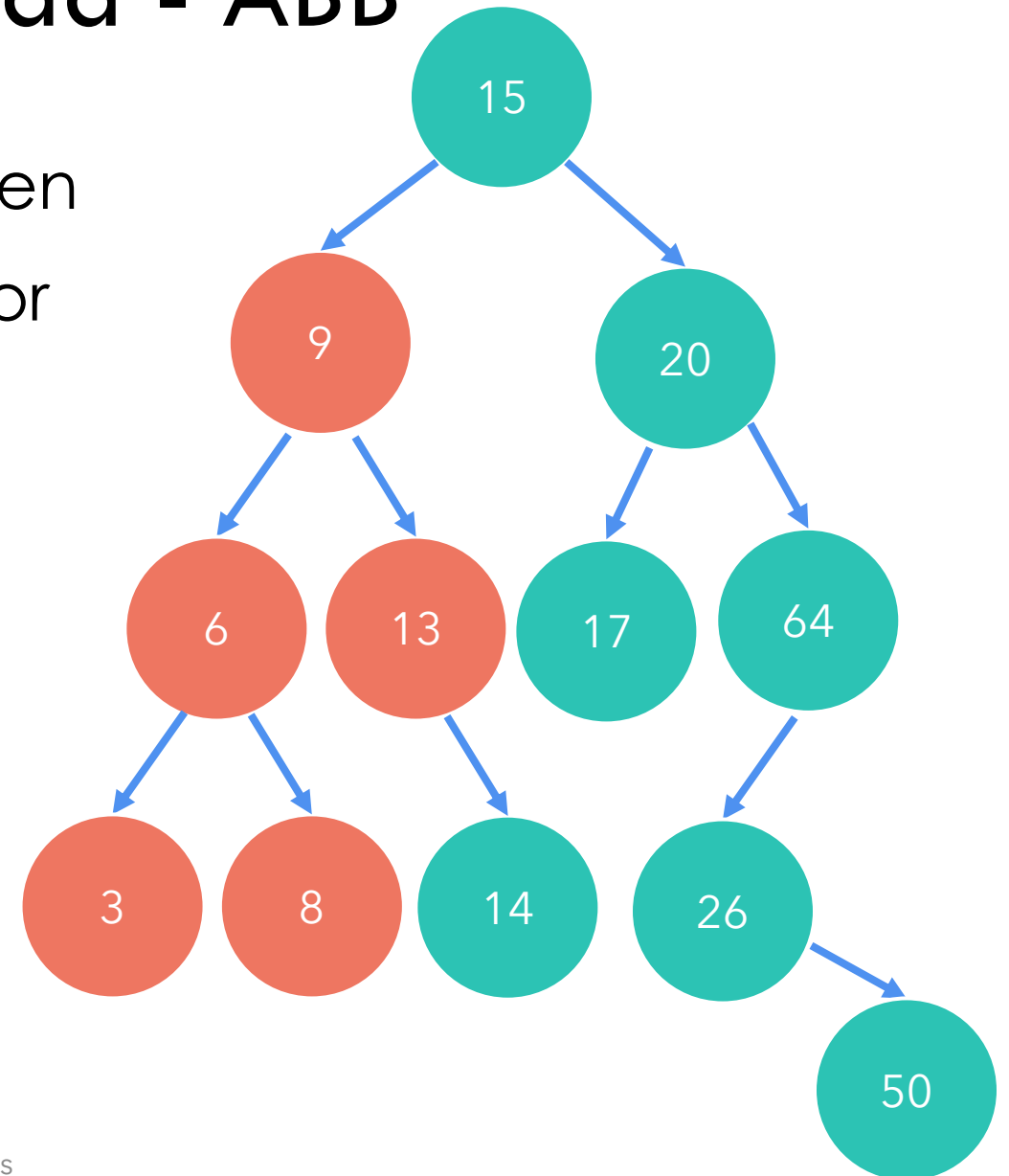
## Propiedades

❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

`Inorder(BinaryTree T, Node v)`

1. `If T.hasLeft(v)`
2.     `Inorder(T,T.left(v))`
3. `visit(v)`
4. `If T.hasRight(v)`
5. `Inorder(T,T.right(v))`

`Inorder(T)=3-6-8-9-13`



# Arboles binarios de búsqueda - ABB

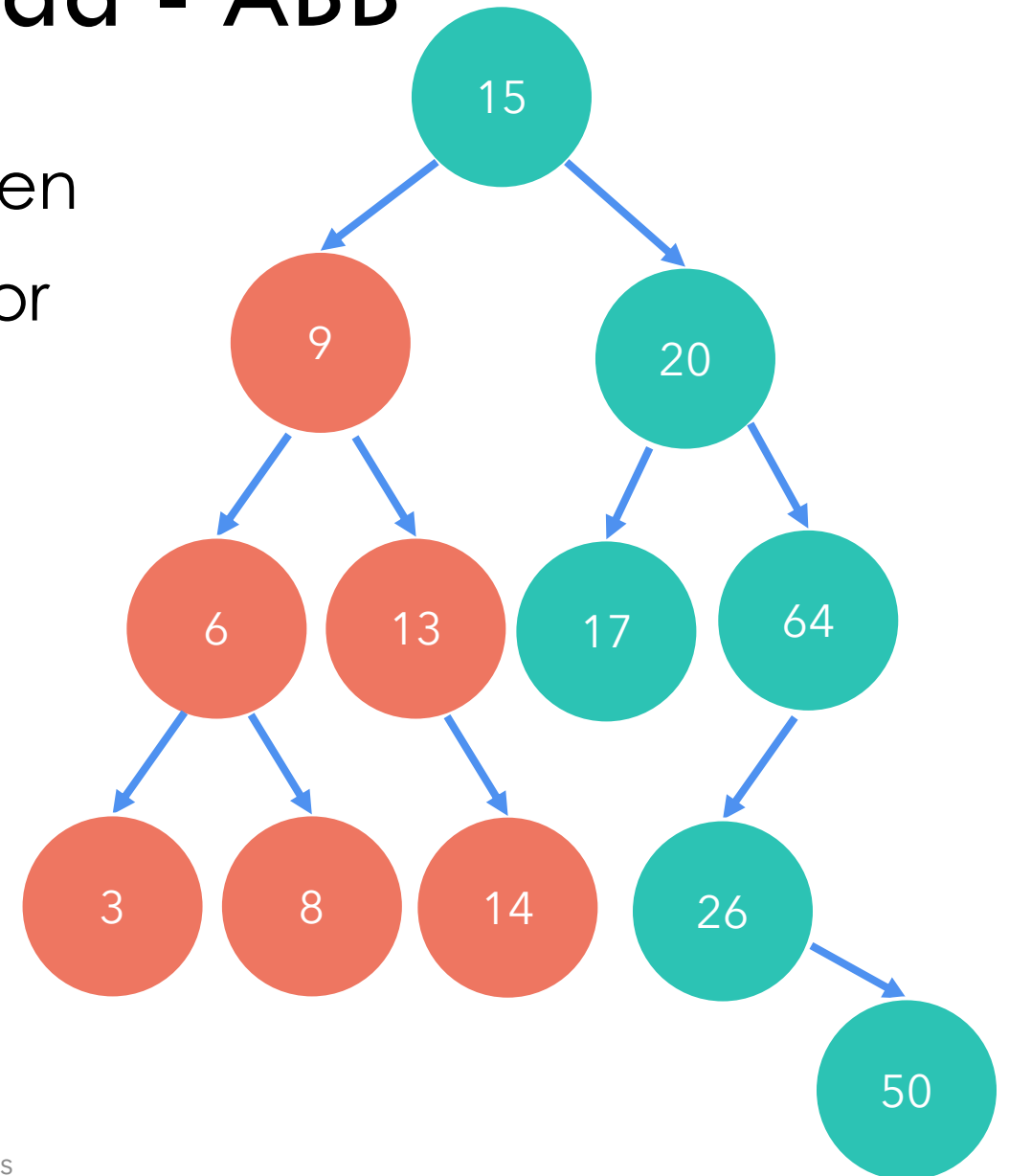
## Propiedades

❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=3-6-8-9-13-14-



# Arboles binarios de búsqueda - ABB

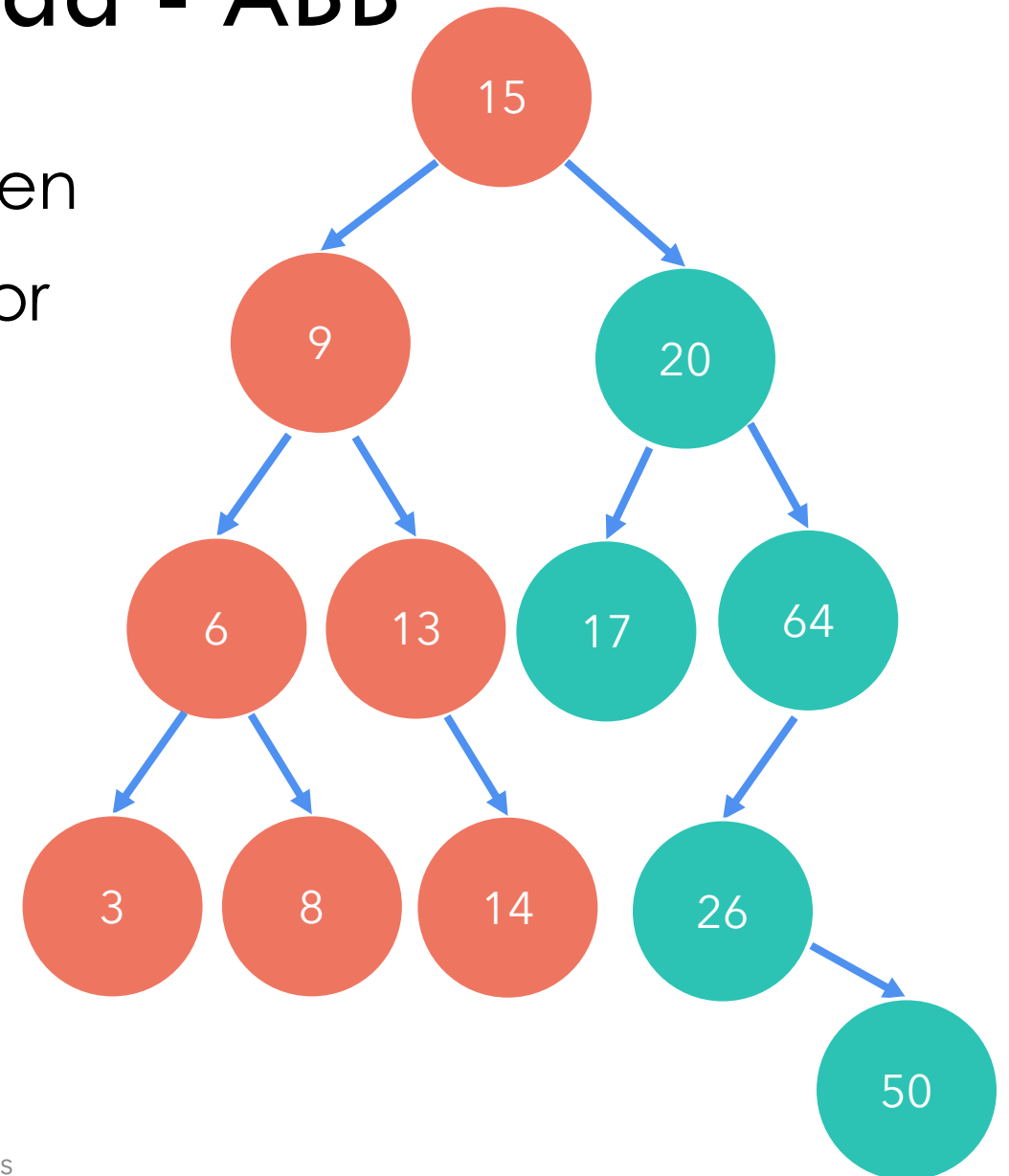
## Propiedades

- ❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=3-6-8-9-13-14-15





# Arboles binarios de búsqueda - ABB

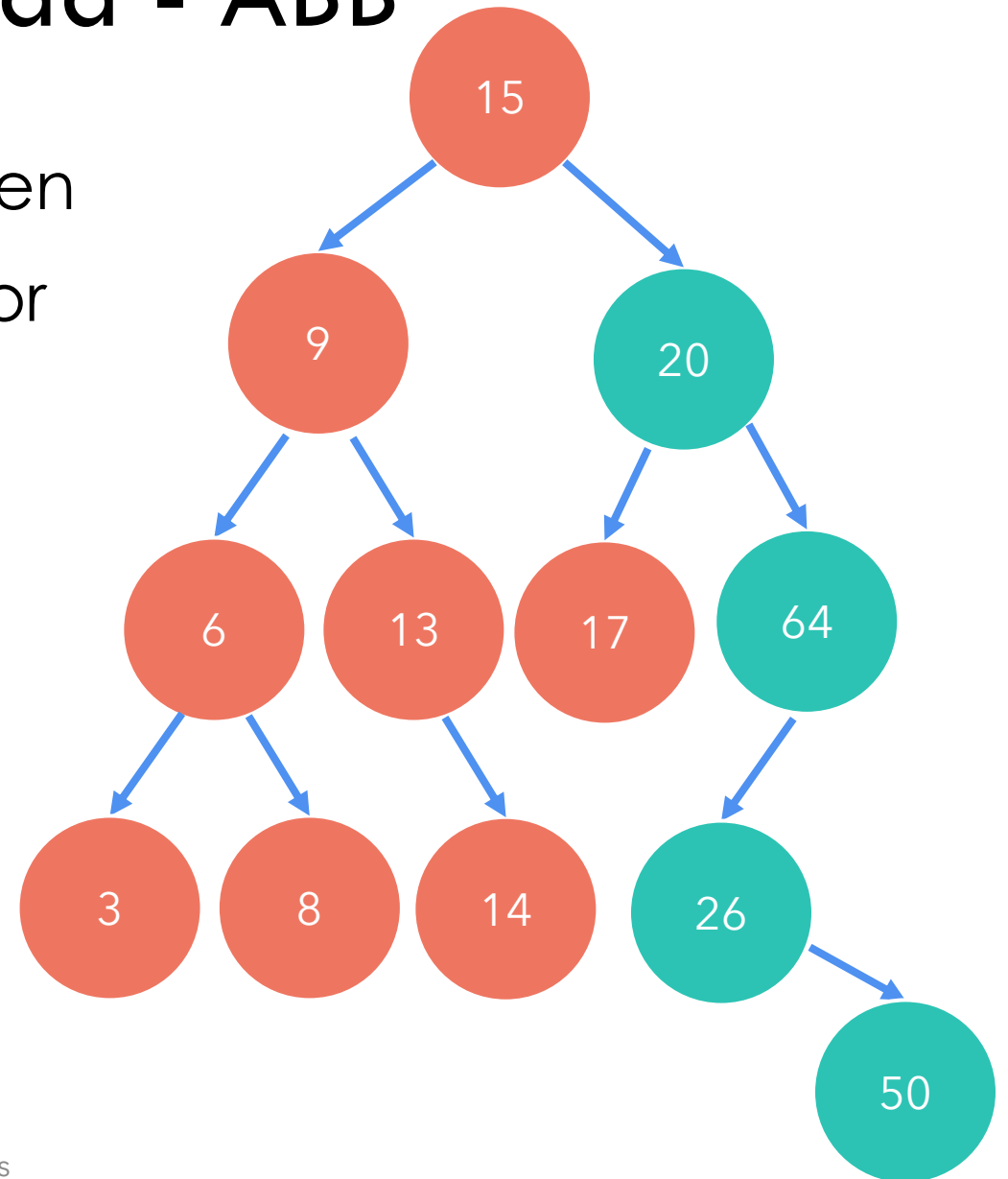
## Propiedades

❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=3-6-8-9-13-14-15-17-



# Arboles binarios de búsqueda - ABB

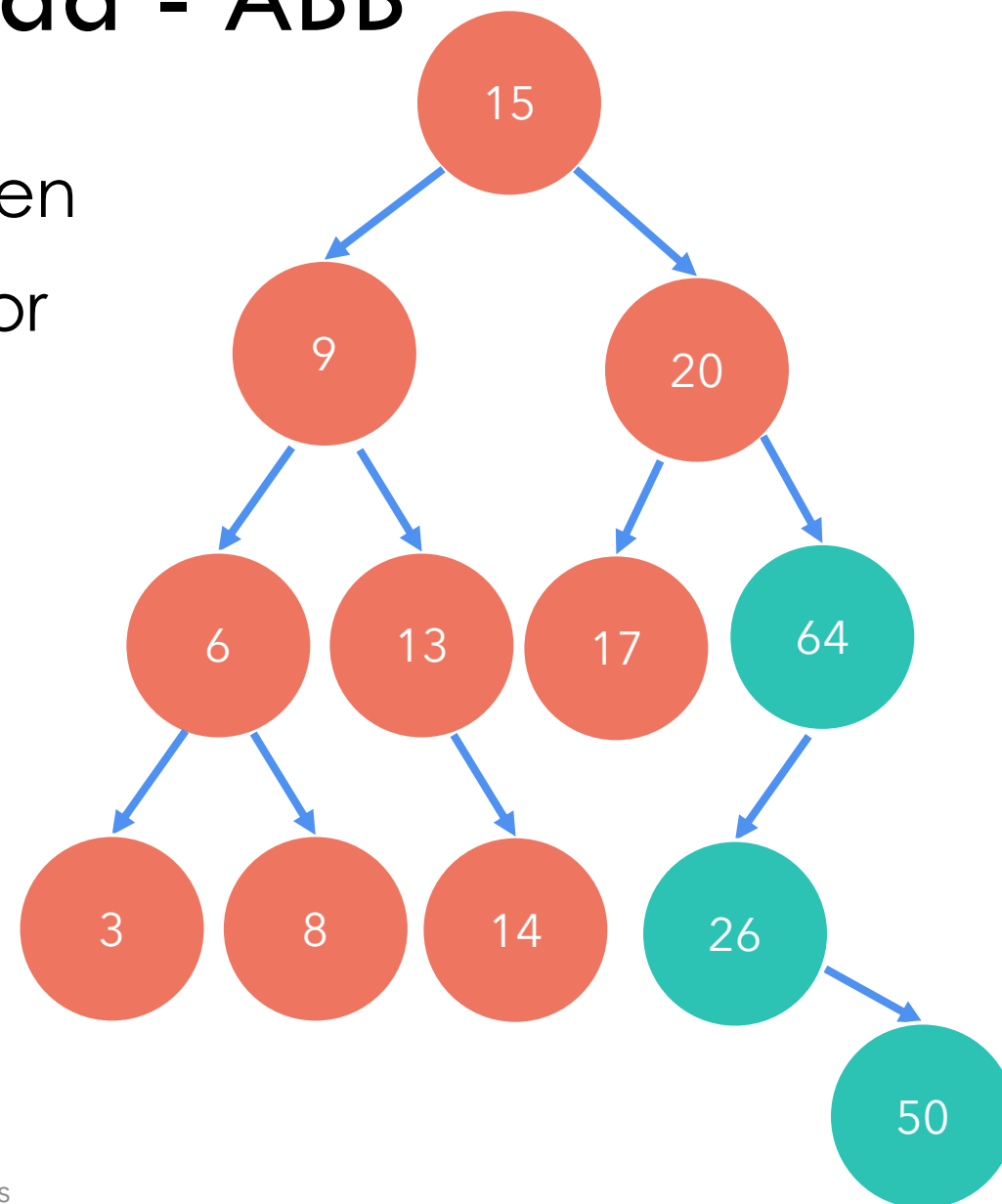
## Propiedades

❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=3-6-8-9-13-14-15-17-20-



# Arboles binarios de búsqueda - ABB

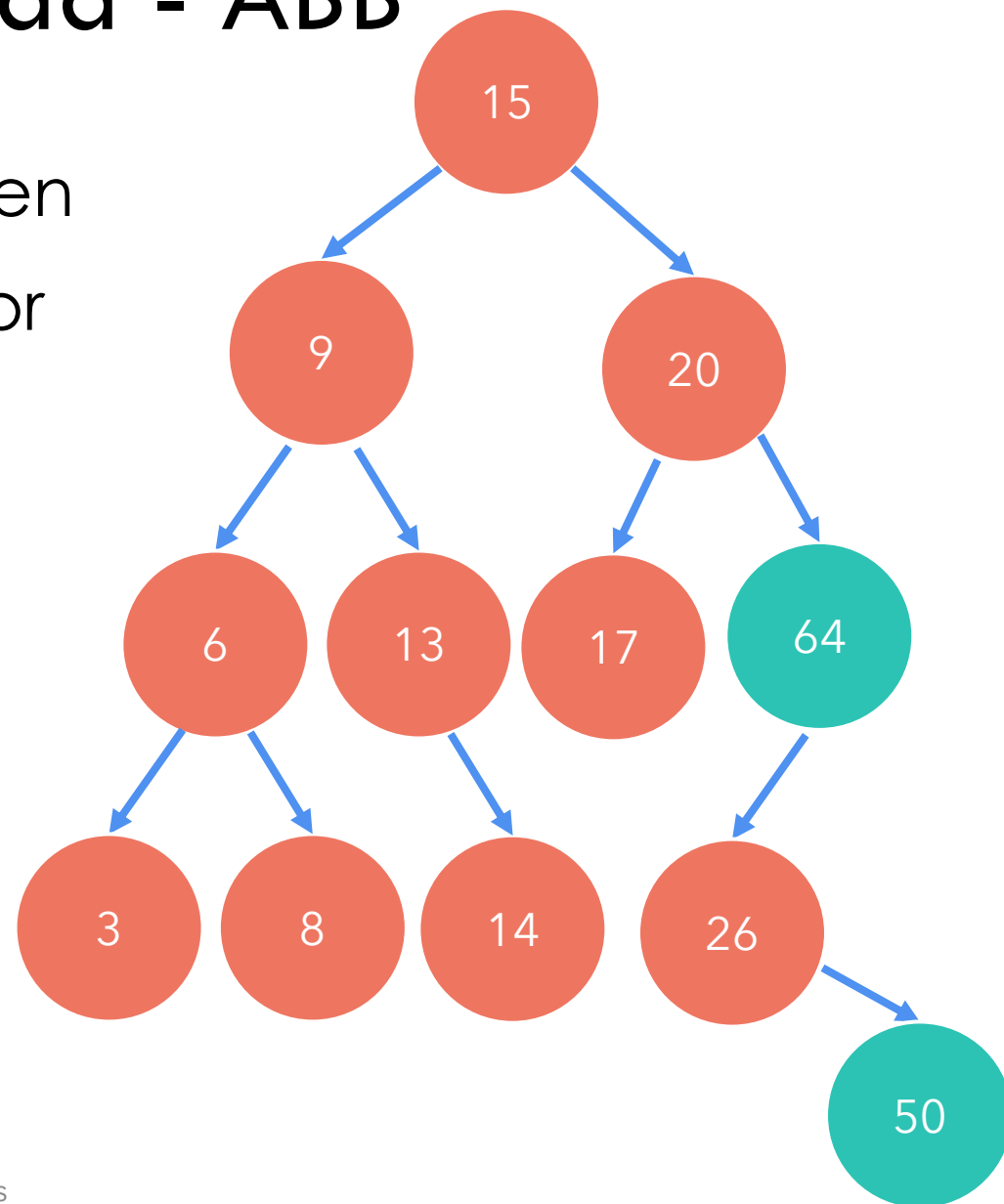
## Propiedades

❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=3-6-8-9-13-14-15-17-20-26-



# Arboles binarios de búsqueda - ABB

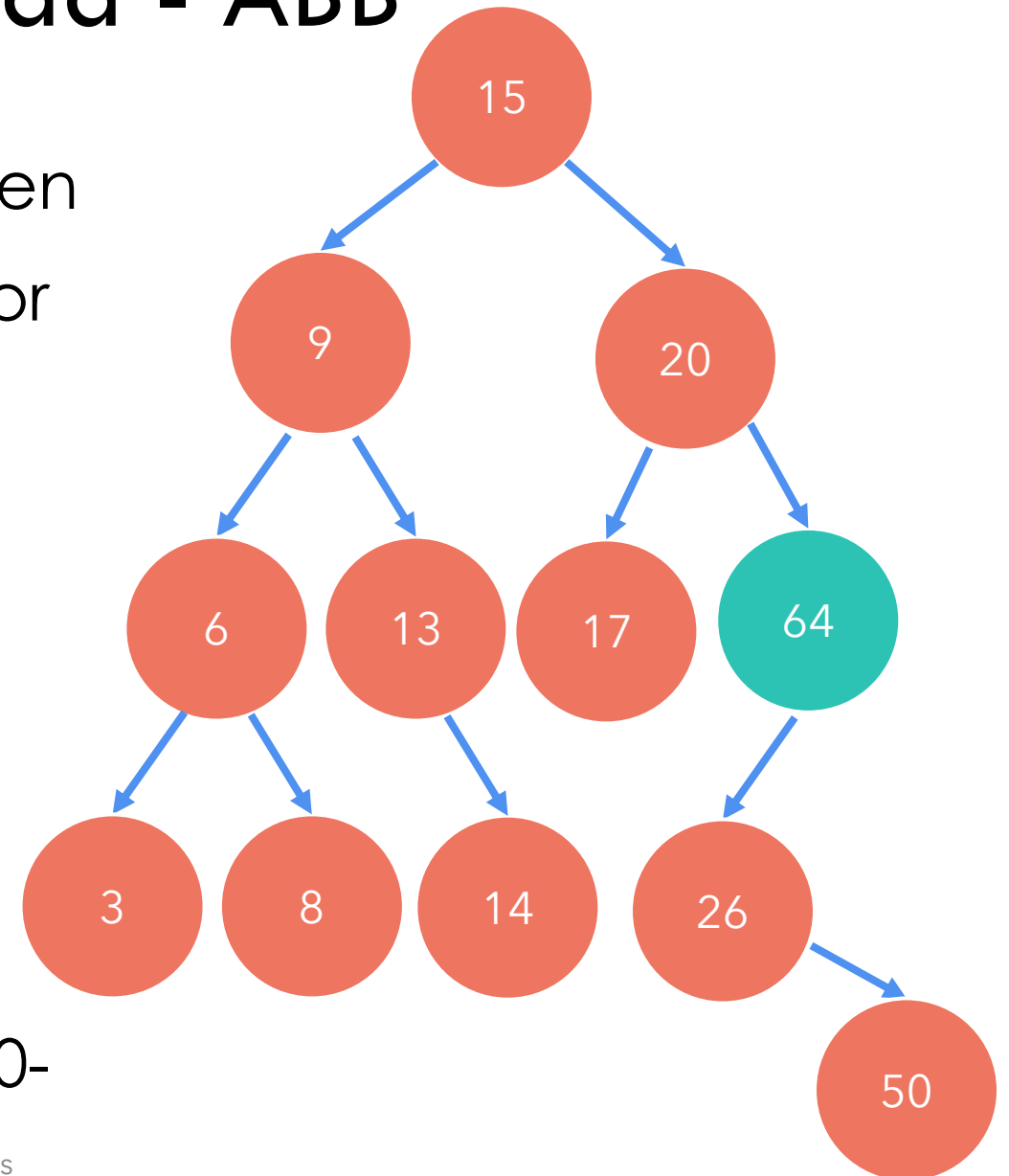
## Propiedades

❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

Inorder(BinaryTree T, Node v)

1. If T.hasLeft(v)
2.     Inorder(T,T.left(v))
3. visit(v)
4. If T.hasRight(v)
5.     Inorder(T,T.right(v))

Inorder(T)=3-6-8-9-13-14-15-17-20-26-50-



# Arboles binarios de búsqueda - ABB

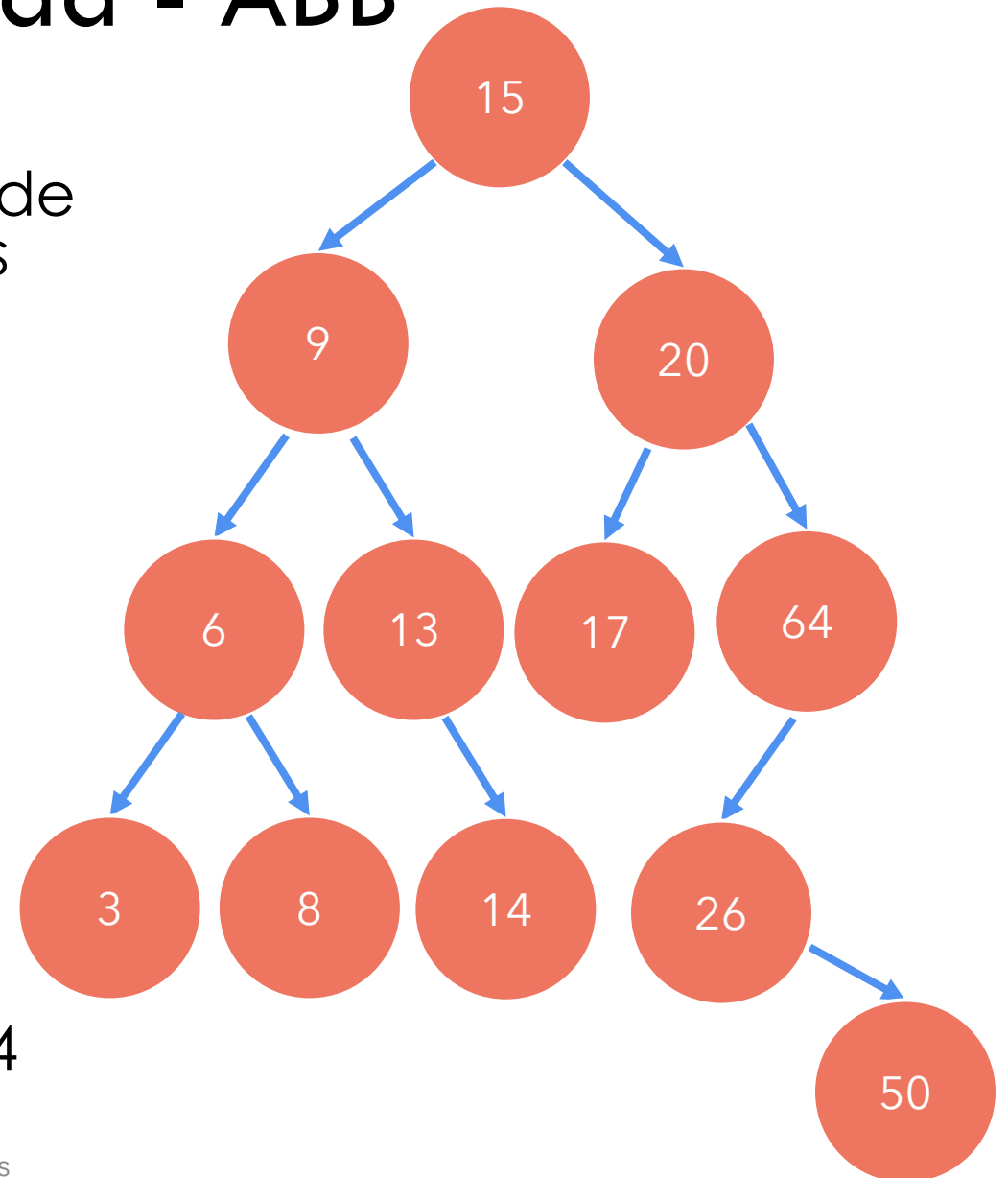
## Propiedades

❑ Recorrido inorder: El recorrido inorden de un abb retorna los datos almacenados organizados de menor a mayor

`Inorder(BinaryTree T, Node v)`

1. `If T.hasLeft(v)`
2.     `Inorder(T,T.left(v))`
3. `visit(v)`
4. `If T.hasRight(v)`
5. `Inorder(T,T.right(v))`

`Inorder(T)=3-6-8-9-13-14-15-17-20-26-50-64`



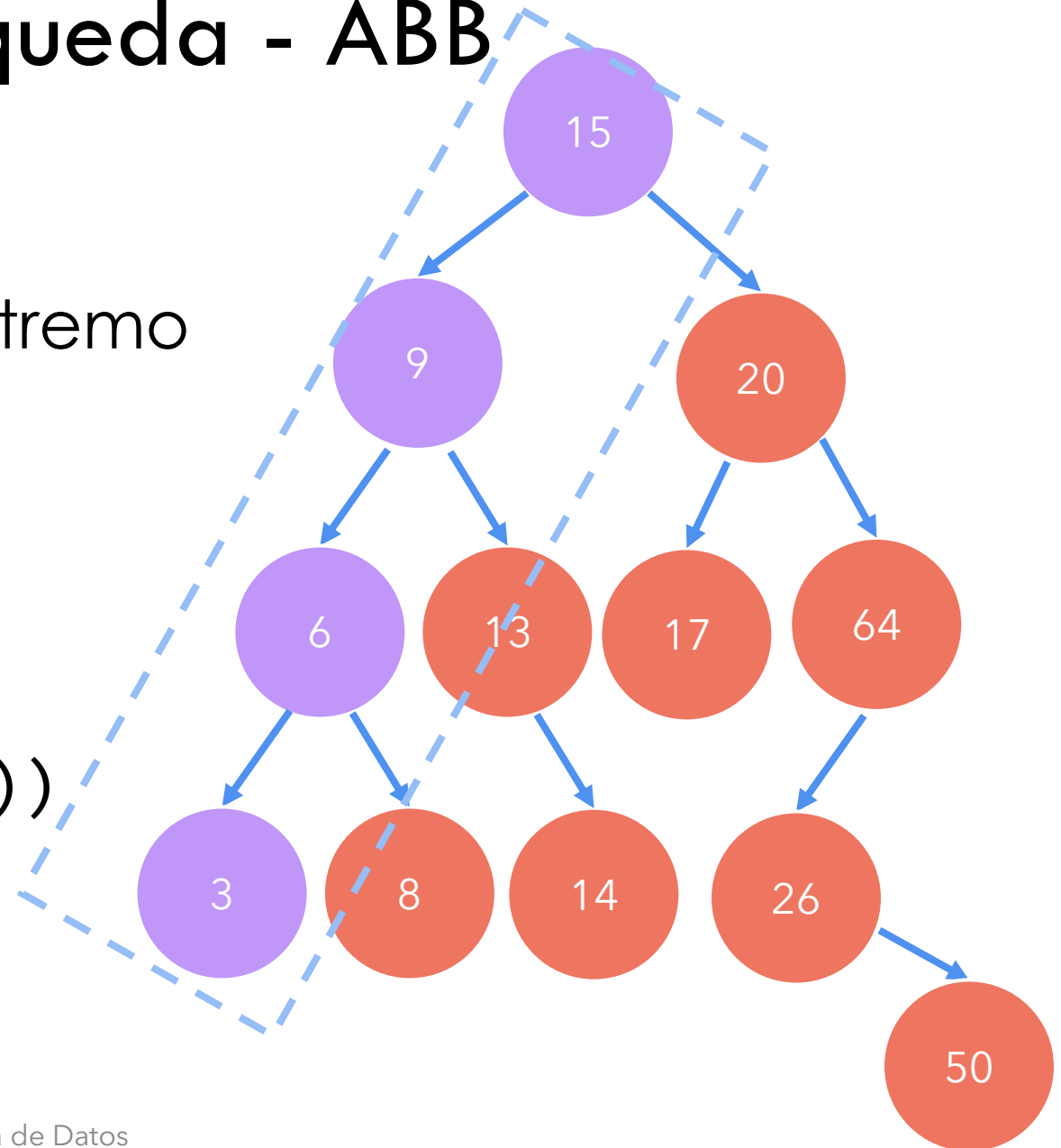
# Arboles binarios de búsqueda - ABB

## Propiedades

- ❑ El valor mínimo en un ABB se encuentra en el nodo más extremo de la rama izquierda

`min(BinaryTree T, Node v)`

1. `If T.hasLeft(v)`
2. `return min(T, T.left(v))`
3. `else`
4. `return v.getData()`



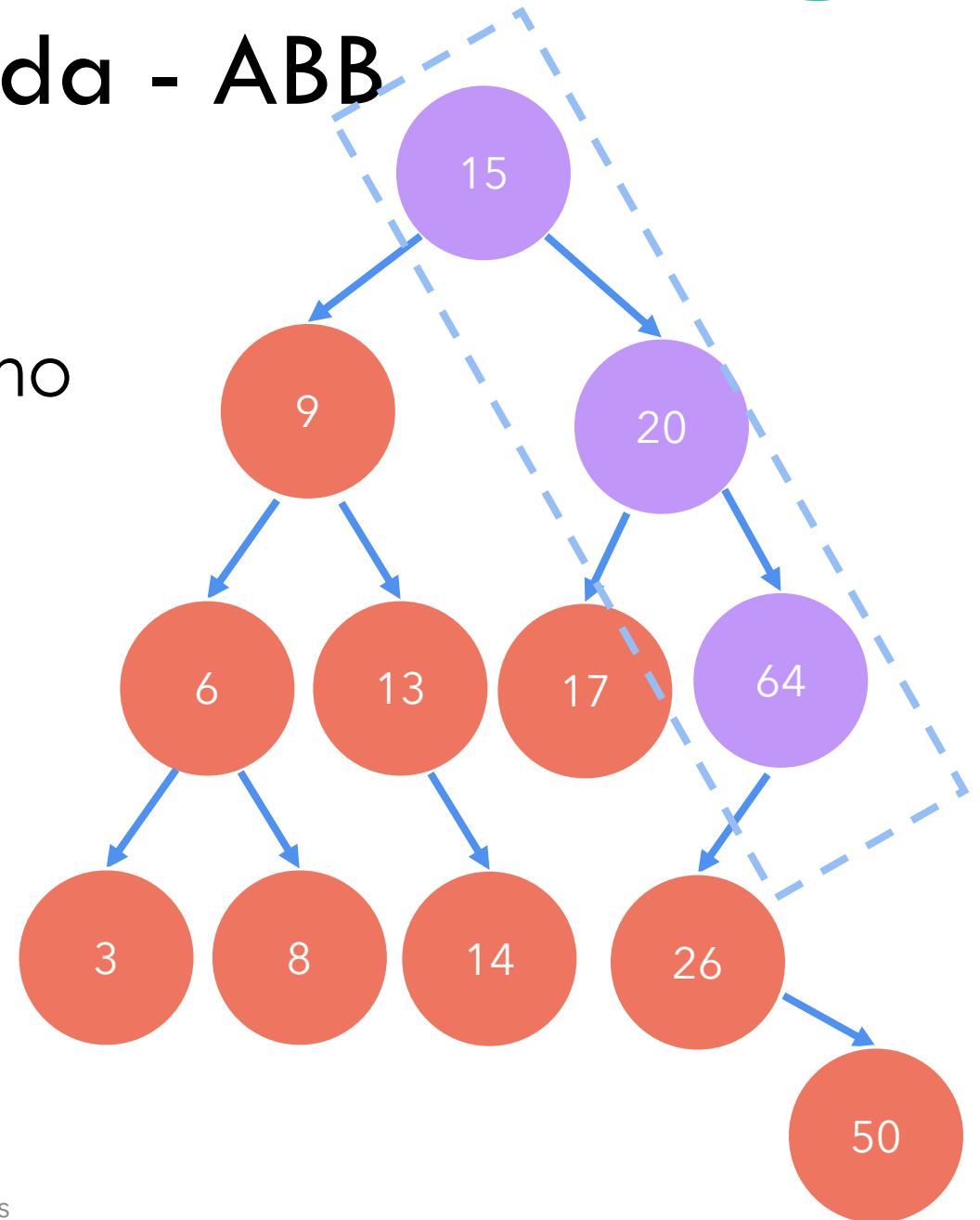
# Arboles binarios de búsqueda - ABB

## Propiedades

- ❑ El valor máximo en un ABB se encuentra en el nodo más extremo de la rama derecha

`max(BinaryTree T, Node v)`

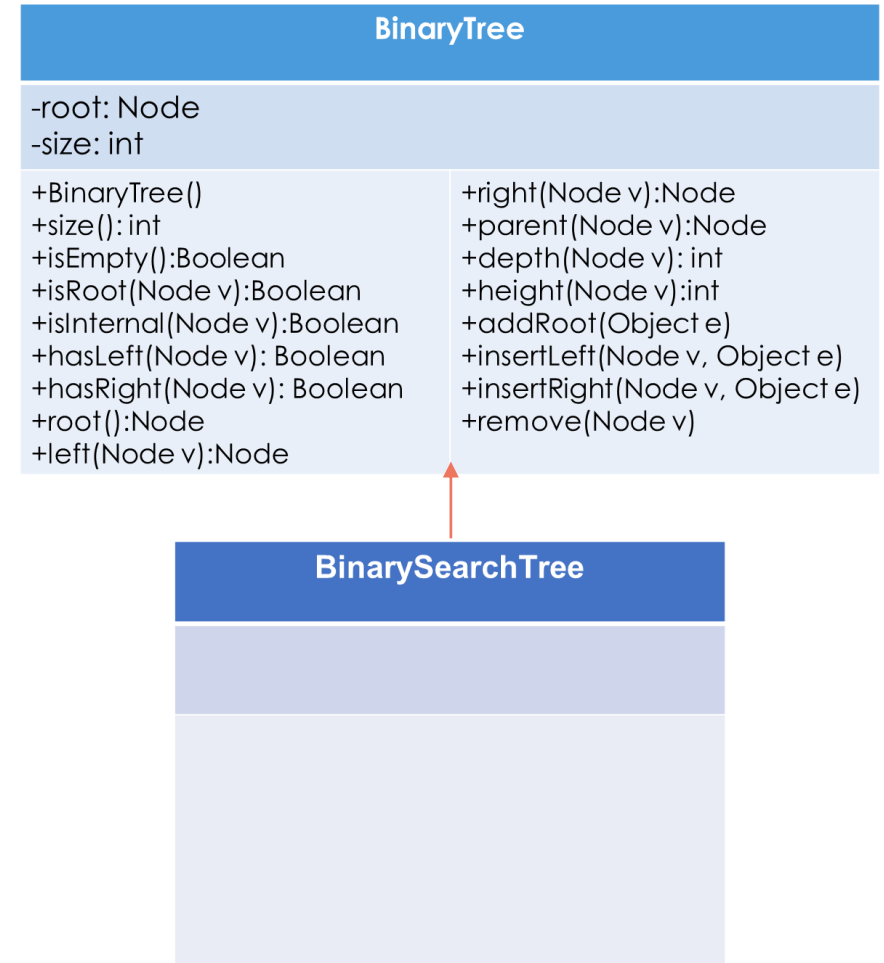
1. `If T.hasRight(v)`
2.     `return max(T, T.right(v))`
3. `else`
4.     `return v.getData()`



# Arboles binarios de búsqueda - ABB

## Implementación

- ❑ La clase BinarySearchTree es una extensión de la clase BinaryTree, heredando todos sus atributos y métodos





# Arboles binarios de búsqueda - ABB

## Implementación

- ❑ Un nodo en un árbol binario de búsqueda almacena un objeto y una clave k asociada al mismo. La clave k permite ordenar los objetos.
- ❑ Para manejar estos datos creamos la clase BSTEntry

```
BSTEntry(Object e, int k)
```

```
    data = e
```

```
    this.k = k
```

```
getData()
```

```
    return data
```

```
setData(Object d)
```

```
    data = d
```

```
getKey()
```

```
    return key
```

```
setKey(int k)
```

```
    this.k = k
```

### BSTEntry

```
#data: Object
```

```
#k:int
```

```
+BSTEntry(Object d, int k)
```

```
+getData(): Object
```

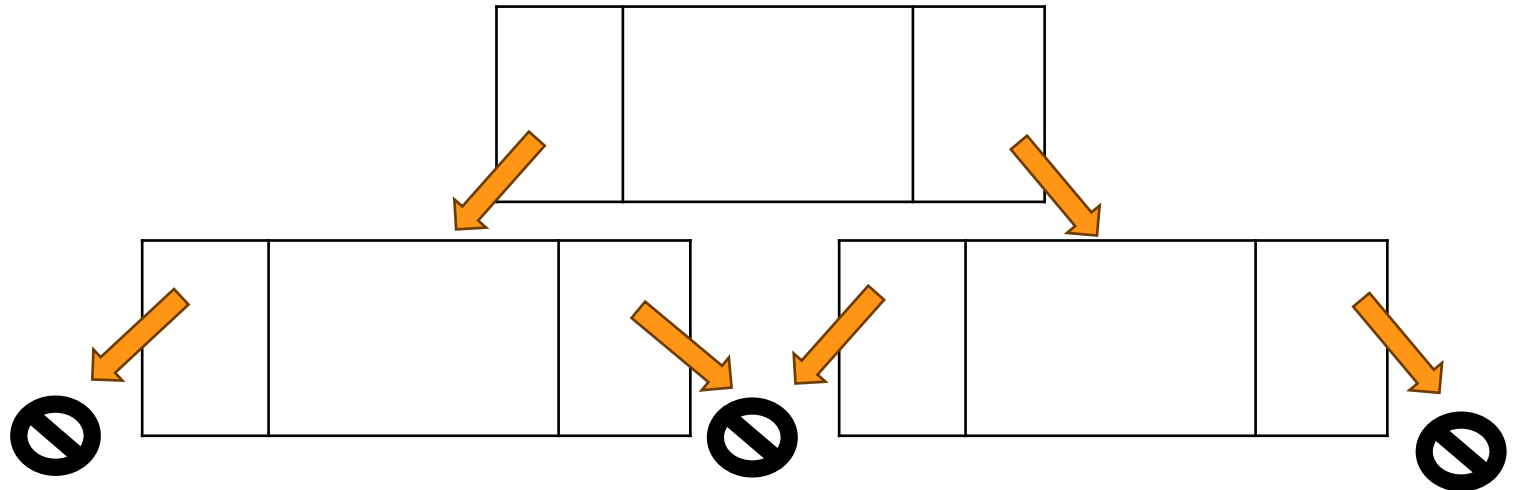
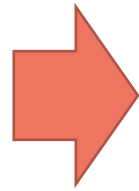
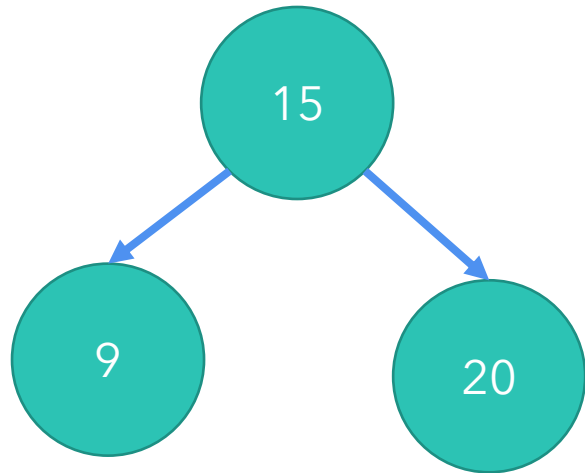
```
+getKey(): int
```

```
+setData(Object d)
```

```
+setKey(int k)
```

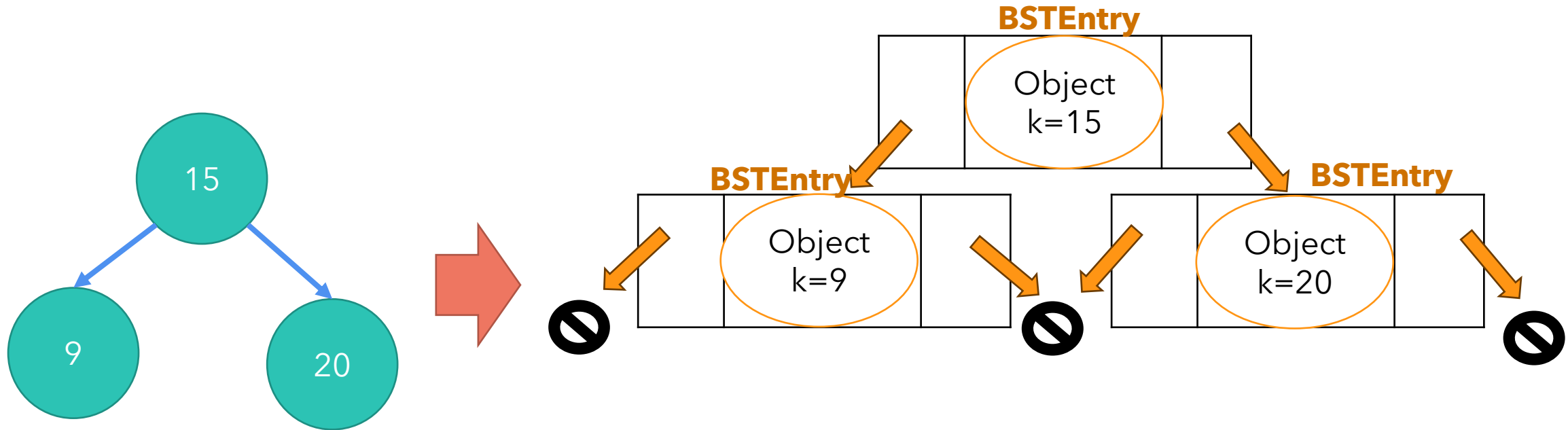
# Arboles binarios de búsqueda - ABB

## Implementación



# Arboles binarios de búsqueda - ABB

## Implementación



# Arboles binarios de búsqueda - ABB

## Implementación

Métodos:

- ❑ `BinarySearchTree()`: Constructor vacío
- ❑ `find(int k)`: busca el nodo con clave `k`
- ❑ `insert(Object e, int k)`: inserta en el árbol binario de búsqueda un nodo con el elemento `e` y clave `k`
- ❑ `Remove(int k)`: eliminar del árbol binario de búsqueda el nodo con clave `k` y retorna el objeto almacenado

BinaryTree	
-root: Node -size: int	
+BinaryTree() +size(): int +isEmpty(): Boolean +isRoot(Node v): Boolean +isInternal(Node v): Boolean +hasLeft(Node v): Boolean +hasRight(Node v): Boolean +root(): Node +left(Node v): Node	+right(Node v): Node +parent(Node v): Node +depth(Node v): int +height(Node v): int +addRoot(Object e) +insertLeft(Node v, Object e) +insertRight(Node v, Object e) +remove(Node v)

BinarySearchTree
+BinarySearchTree() +find(int k): Node +insert(Object e, int k) +remove(int k): Object

# Arboles binarios de búsqueda - ABB

## Implementación

Métodos:

❑ find(int k): busca el nodo con clave k

## Algoritmo recursivo

1. Se compara k con la raíz
2. Si k es menor al valor de la raíz, se recorre el subárbol izquierdo de forma recursiva
3. Si k es mayor al valor de la raíz, se recorre el subárbol derecho

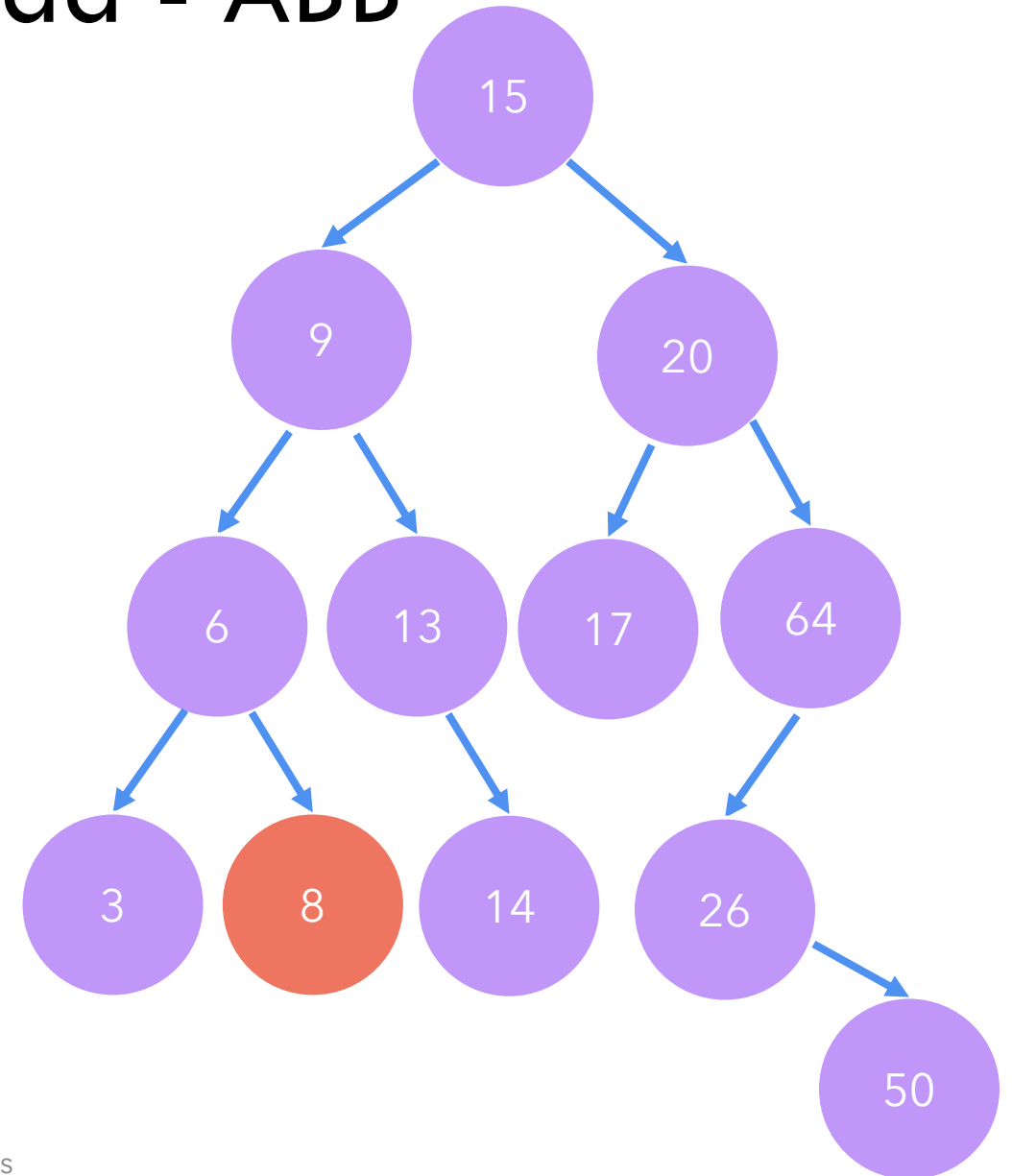
BinaryTree	
-root: Node -size: int	
+BinaryTree() +size(): int +isEmpty(): Boolean +isRoot(Node v): Boolean +isInternal(Node v): Boolean +hasLeft(Node v): Boolean +hasRight(Node v): Boolean +root(): Node +left(Node v): Node	+right(Node v): Node +parent(Node v): Node +depth(Node v): int +height(Node v): int +addRoot(Object e) +insertLeft(Node v, Object e) +insertRight(Node v, Object e) +remove(Node v)

BinarySearchTree
+BinarySearchTree() +find(int k): Node +insert(Object e, int k) +remove(int k): Object

# Arboles binarios de búsqueda - ABB

## Ejemplo:

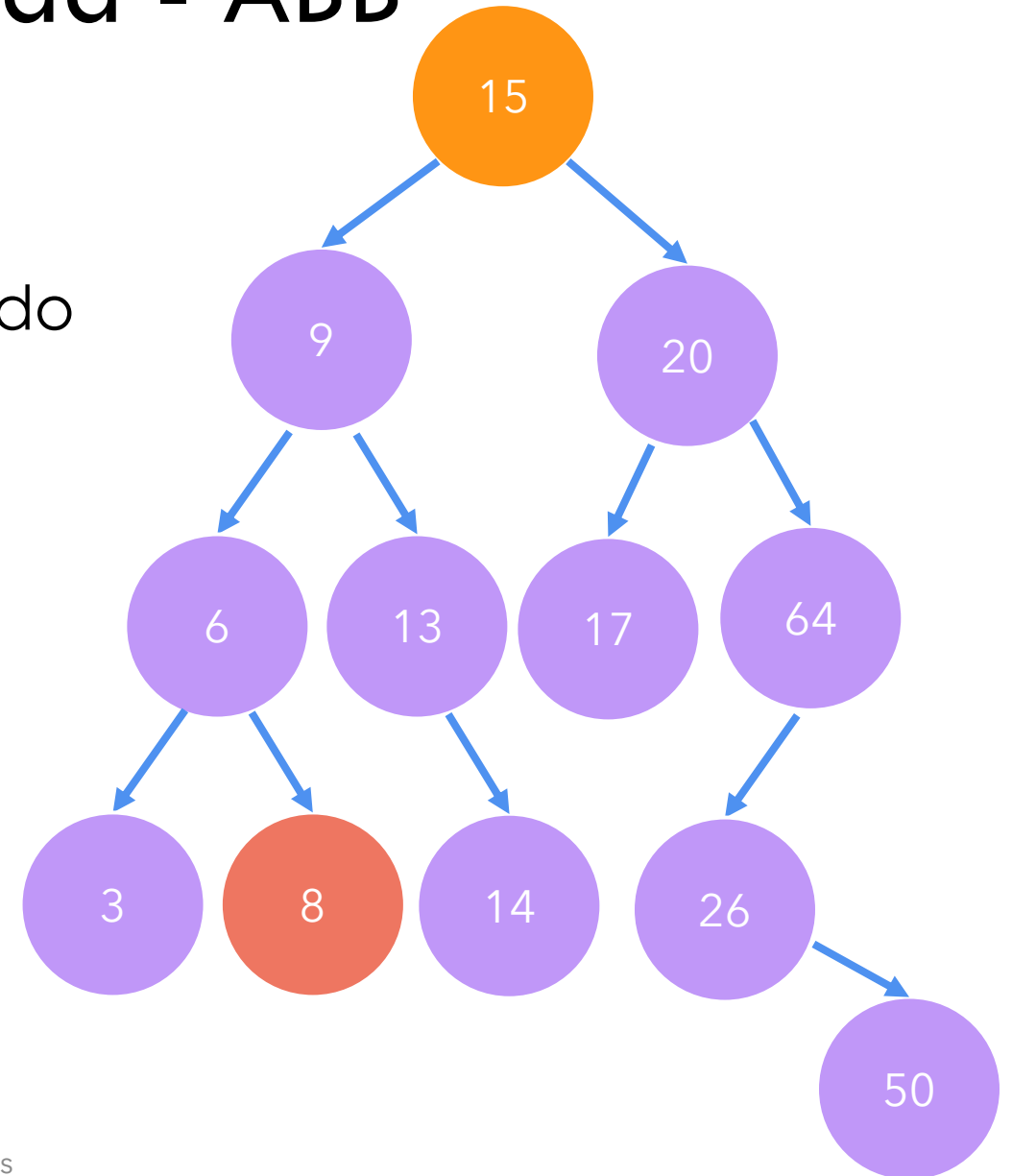
❑ Buscar nodo con clave igual a 8



# Arboles binarios de búsqueda - ABB

## Ejemplo:

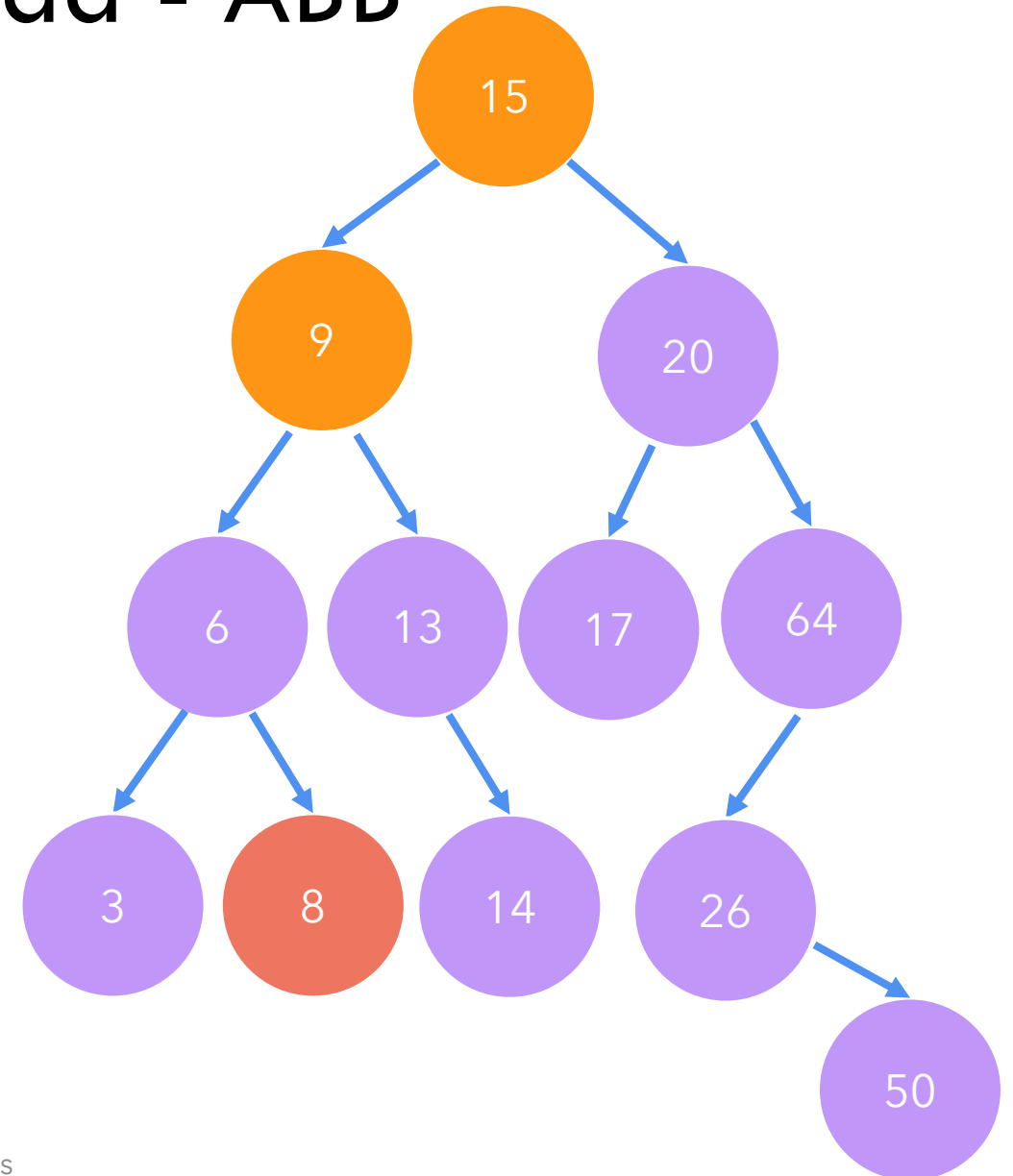
- ❑ Buscar nodo con clave igual a 8
- $8 < 15$  por tanto se busca al lado izquierdo del árbol



# Arboles binarios de búsqueda - ABB

## Ejemplo:

- ❑ Buscar nodo con clave igual a 8
- $8 < 9$  por tanto se continua por el lado izquierdo del árbol

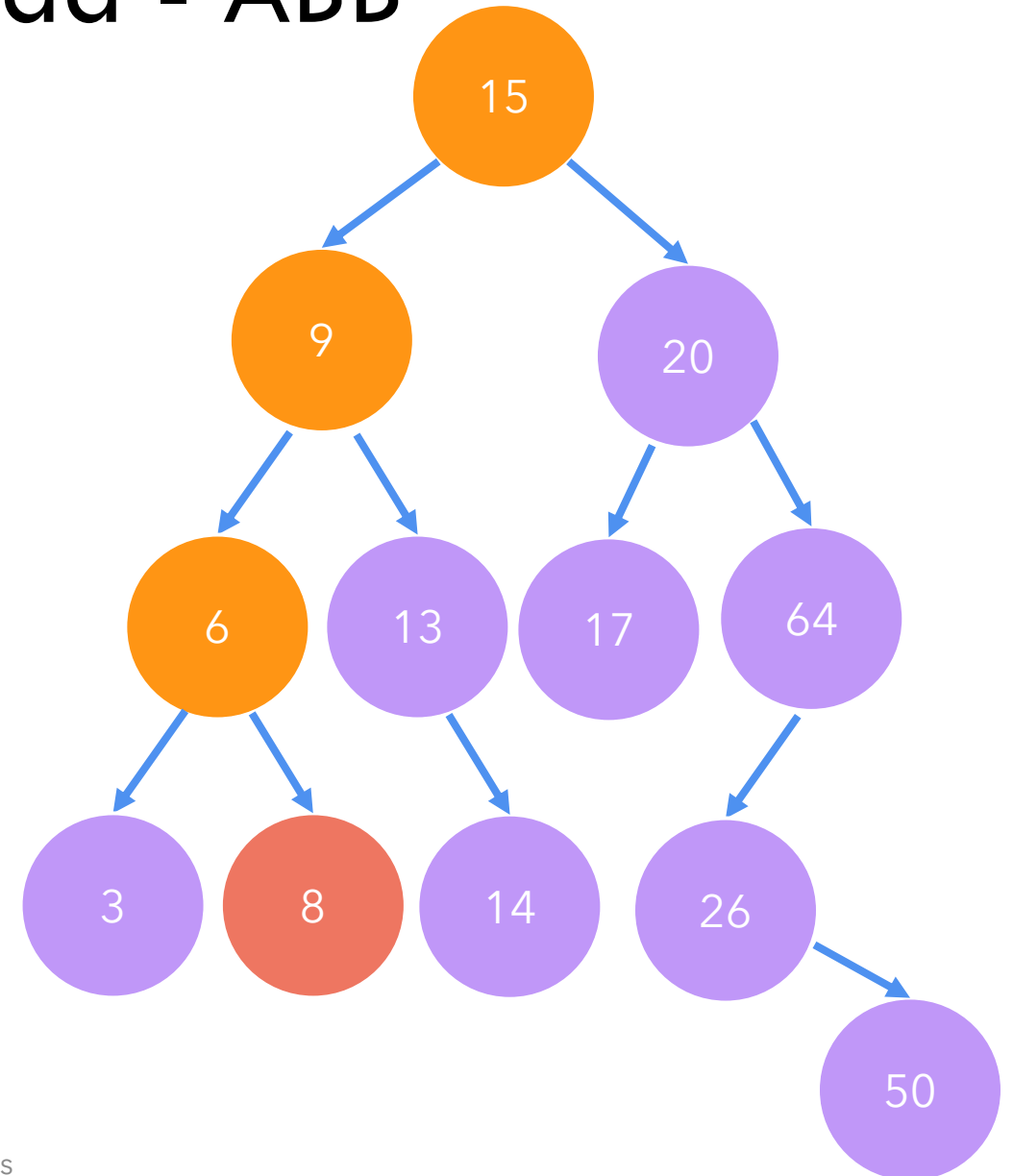




# Arboles binarios de búsqueda - ABB

## Ejemplo:

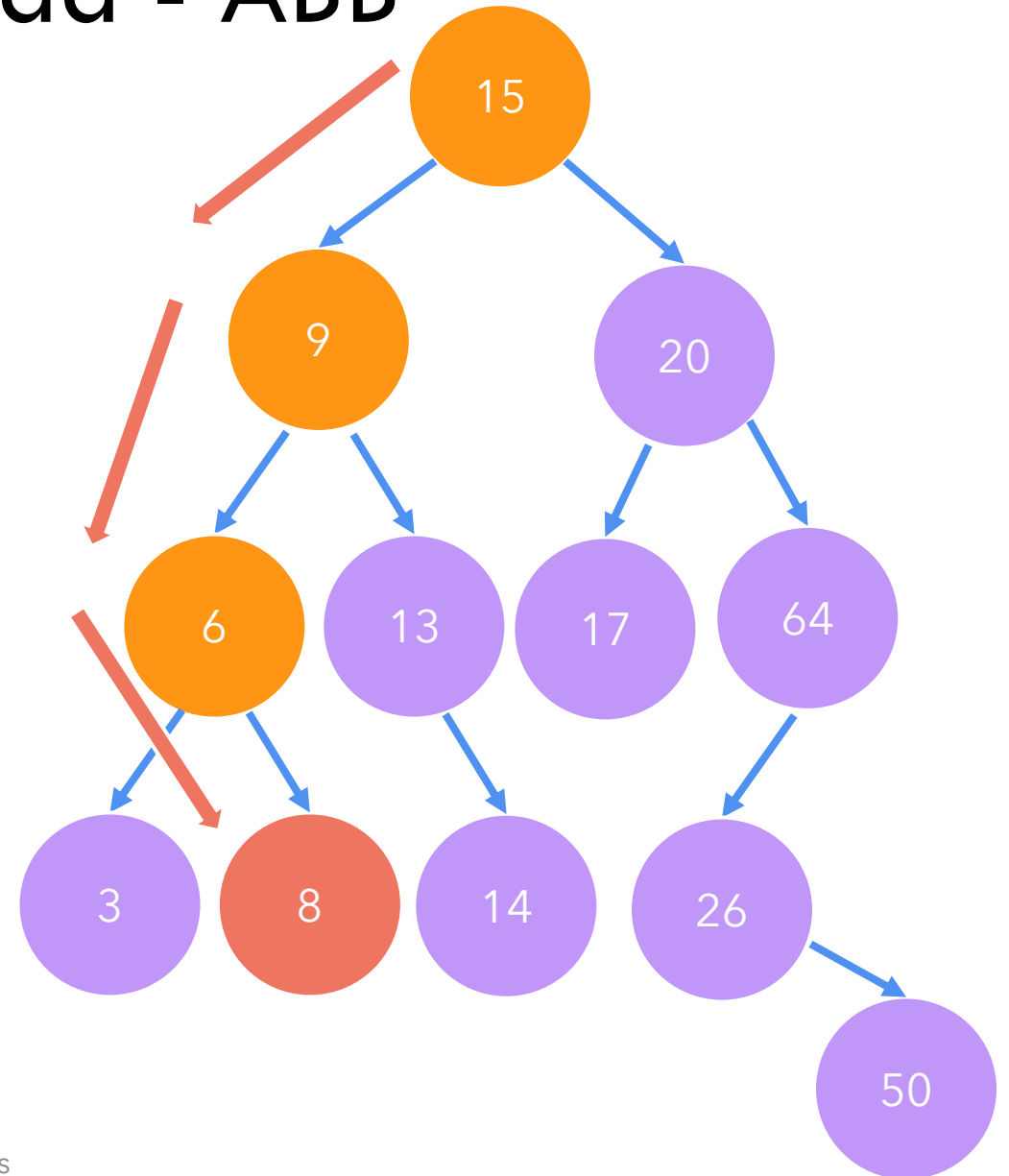
- ❑ Buscar nodo con clave igual a 8
- $8 > 6$  por tanto se continua por el lado derecho del árbol



# Arboles binarios de búsqueda - ABB

## Ejemplo:

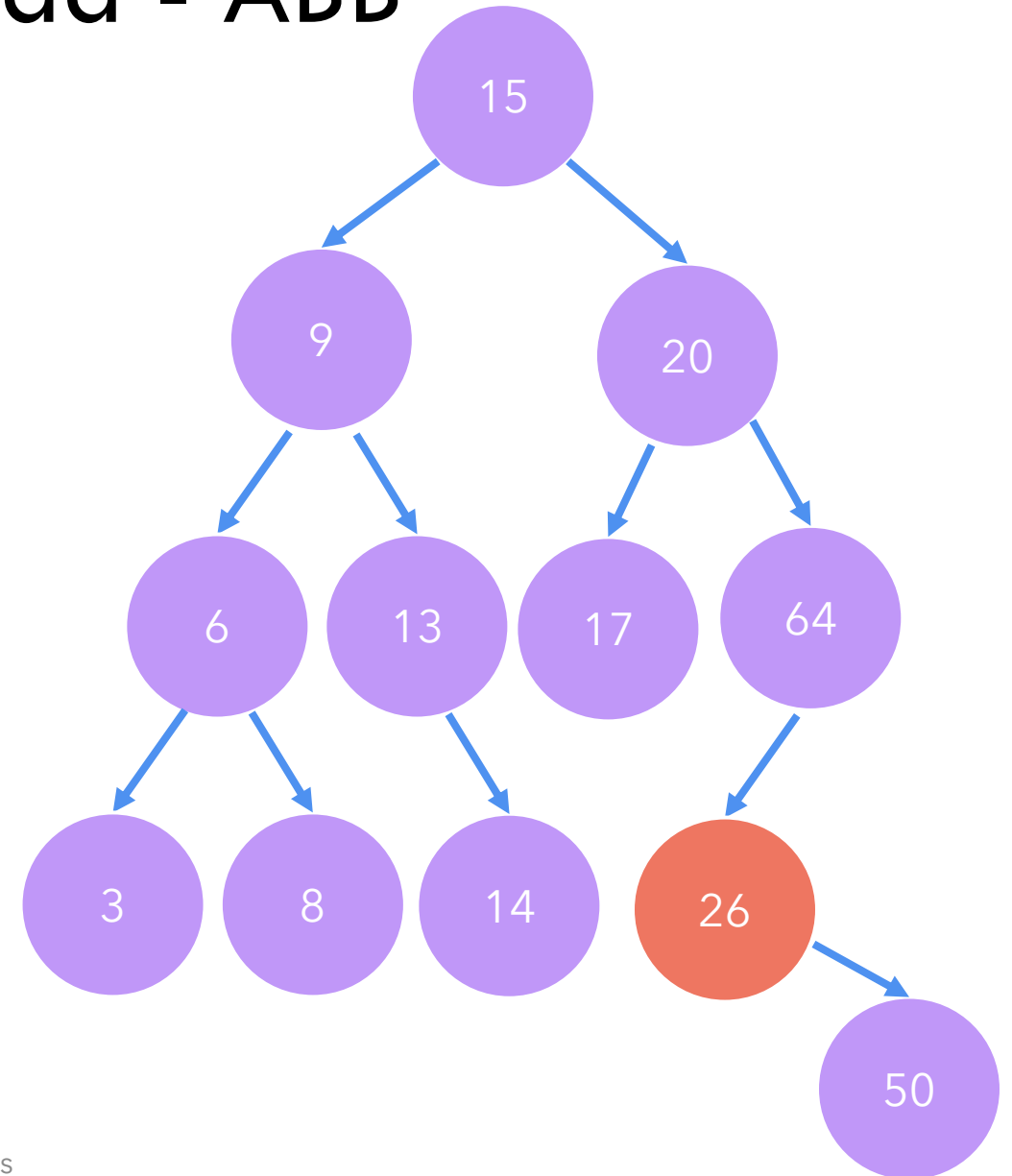
- ❑ Buscar nodo con clave igual a 8
- Se encuentra el nodo con clave 8



# Arboles binarios de búsqueda - ABB

## Ejemplo:

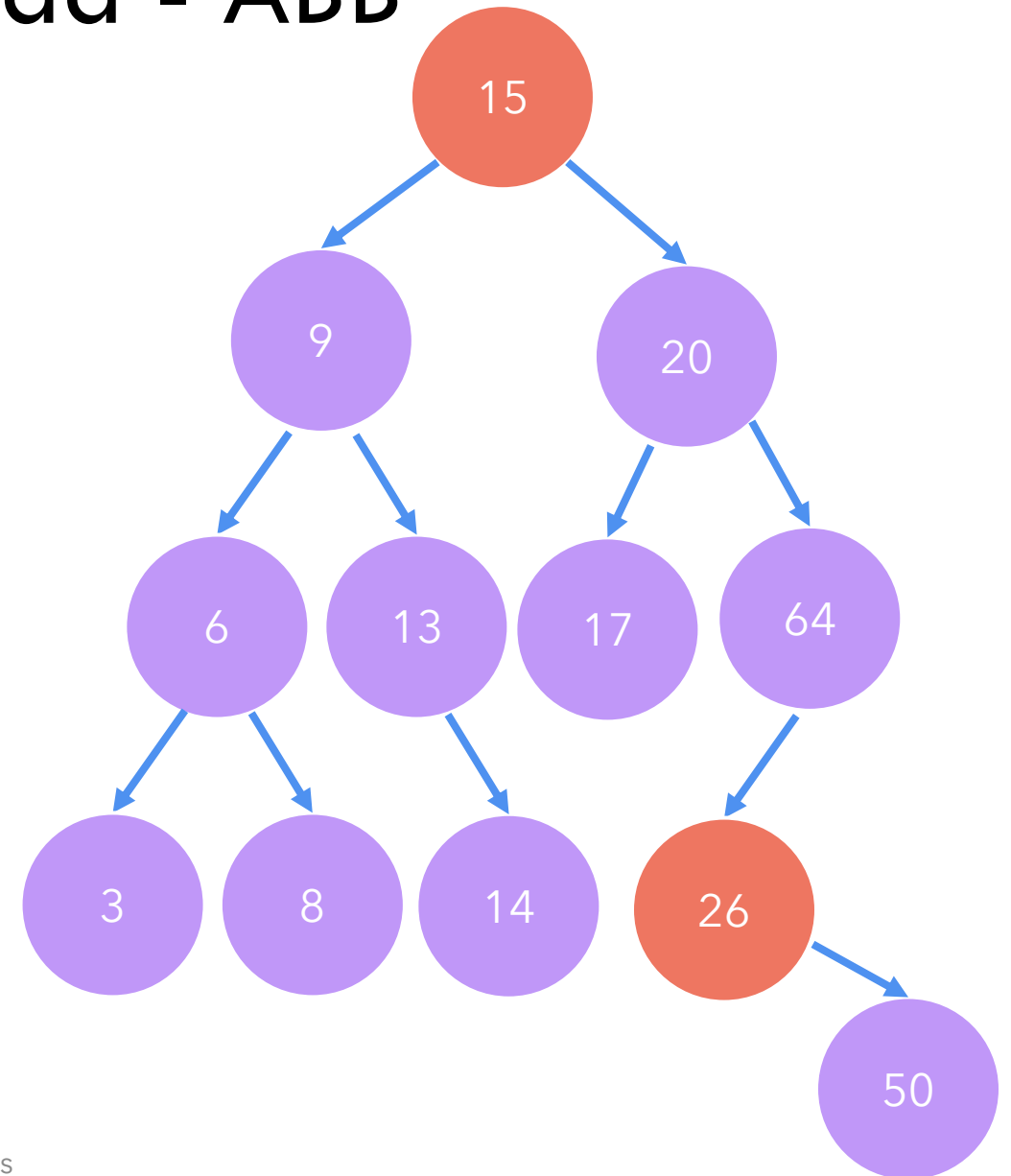
❑ Buscar nodo con clave igual a 26



# Arboles binarios de búsqueda - ABB

## Ejemplo:

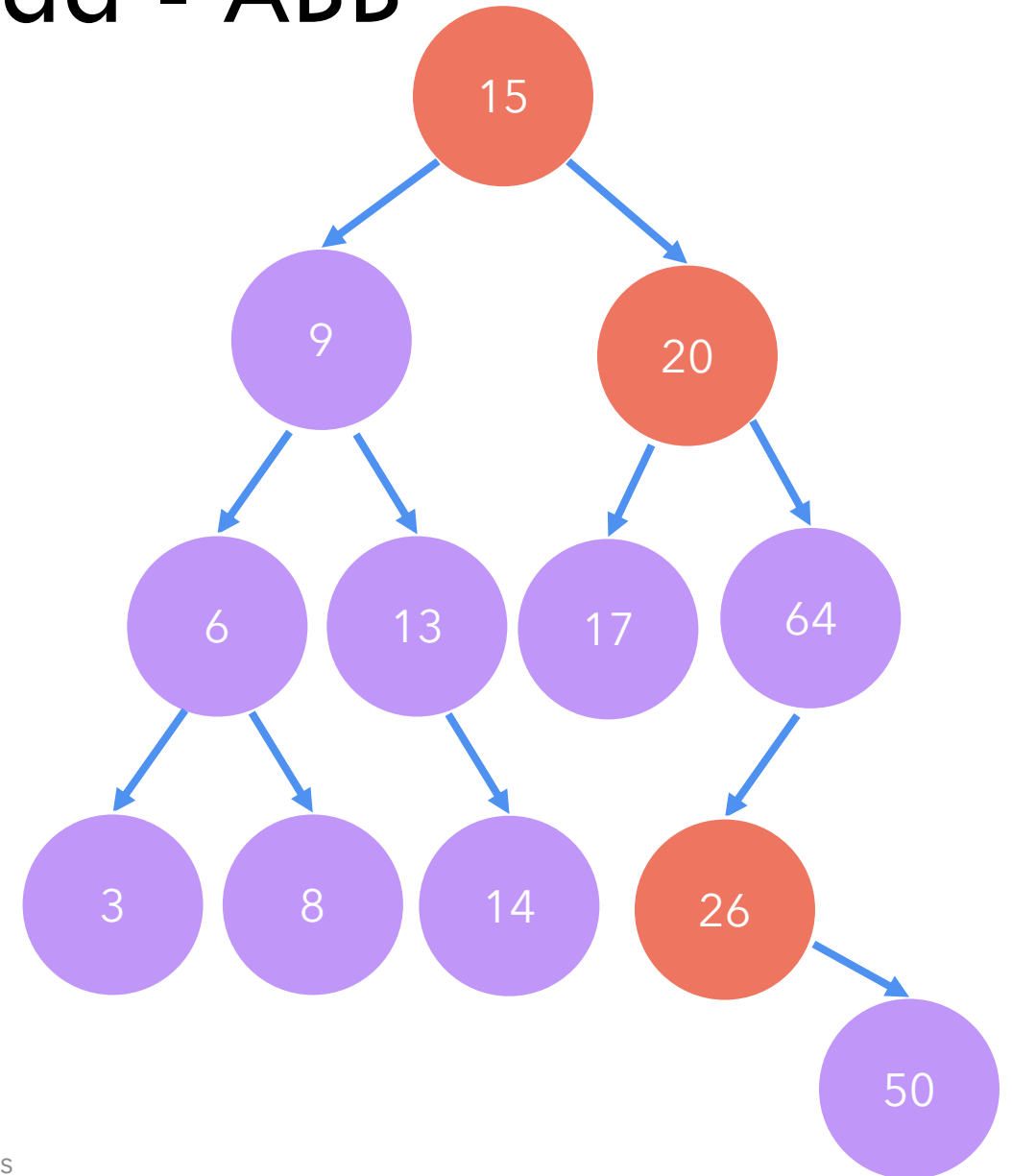
❑ Buscar nodo con clave igual a 26



# Arboles binarios de búsqueda - ABB

## Ejemplo:

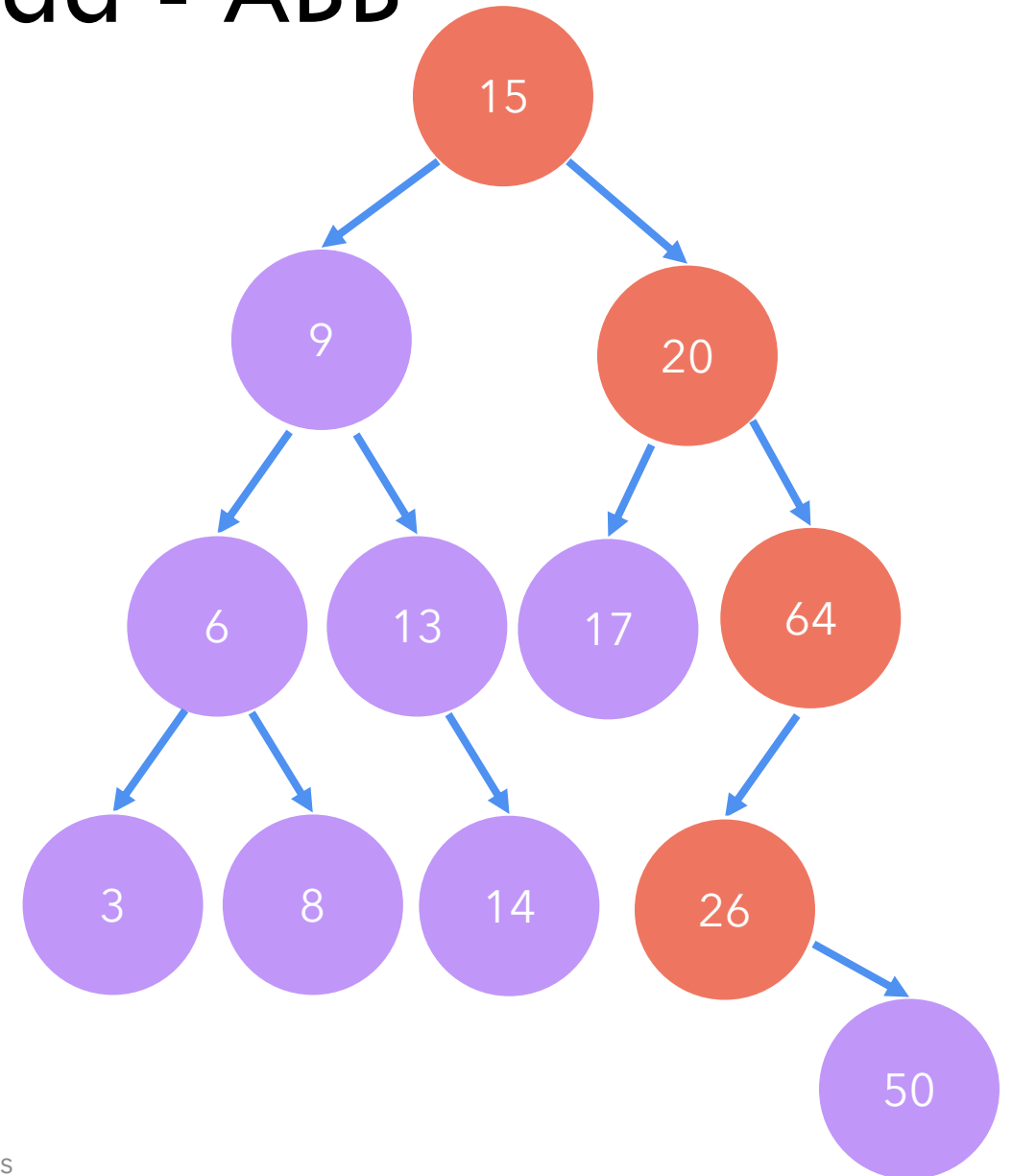
❑ Buscar nodo con clave igual a 26



# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Buscar nodo con clave igual a 26



# Arboles binarios de búsqueda - ABB

## Implementación

Métodos:

❑ `find(int k)`: busca el nodo con clave `k`

## Algoritmo recursivo

1. Se compara `k` con la raíz
2. Si `k` es menor al valor de la raíz, se recorre el subárbol izquierdo de forma recursiva
3. Si `k` es mayor al valor de la raíz, se recorre el subárbol derecho

## Seu código

```
find(int k)
```

```
1. return searchTree(k,root)
```

```
searchTree(int key, Node v)
```

```
1. BSTEntry u = v.getData()
```

```
2. if k==u.getKey() //Caso base
```

```
3.     return v
```

```
4. elseif k<u.getKey()
```

```
5.     return searchTree(k,v.getLeft())
```

```
6. else
```

```
7.     return searchTree(k,v.getRight())
```

# Arboles binarios de búsqueda - ABB

## Implementación

insert(Object e, int k): inserta en el árbol binario de búsqueda un nodo con el elemento e y clave k

Algoritmo:

1. Crear un nuevo BSTEntry O con dato = e y key=k
2. Si el árbol está vacío, O será la raíz; por tanto, se invoca el método addRoot(Object e) de la clase BinaryTree
3. En caso contrario, de forma recursiva, se recorre el subárbol izquierdo si la clave es menor a la raíz, o el subárbol derecho si la clave es mayor a la raíz

BinaryTree	
-root: Node -size: int	
+BinaryTree() +size(): int +isEmpty(): Boolean +isRoot(Node v): Boolean +isInternal(Node v): Boolean +hasLeft(Node v): Boolean +hasRight(Node v): Boolean +root(): Node +left(Node v): Node	+right(Node v): Node +parent(Node v): Node +depth(Node v): int +height(Node v): int +addRoot(Object e) +insertLeft(Node v, Object e) +insertRight(Node v, Object e) +remove(Node v)

BinarySearchTree
+BinarySearchTree() +find(int k): Node +insert(Object e, int k) +remove(int k): Object





# Arboles binarios de búsqueda - ABB

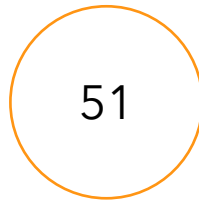
## Ejemplo:

❑ Crear el árbol con las claves: 51 – 23 – 54 – 68 – 22 – 73 – 59 – 30

# Arboles binarios de búsqueda - ABB

## Ejemplo:

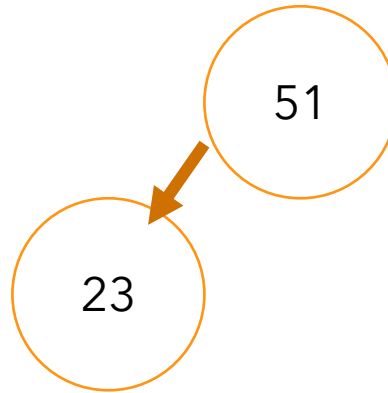
❑ Crear el árbol con las claves: 51 – 23 – 54 – 68 – 22 – 73 – 59 – 30



# Arboles binarios de búsqueda - ABB

## Ejemplo:

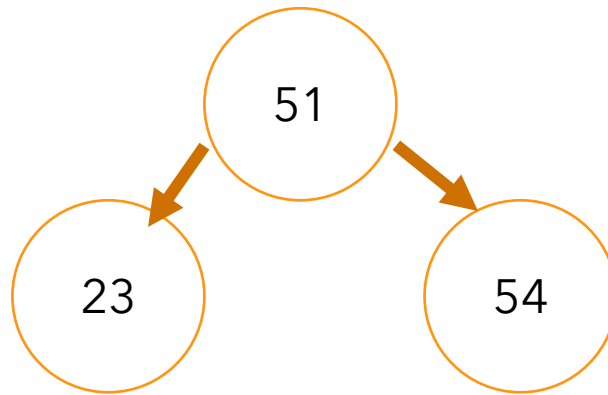
❑ Crear el árbol con las claves: 51 – 23 – 54 – 68 – 22 – 73 – 59 – 30



# Arboles binarios de búsqueda - ABB

## Ejemplo:

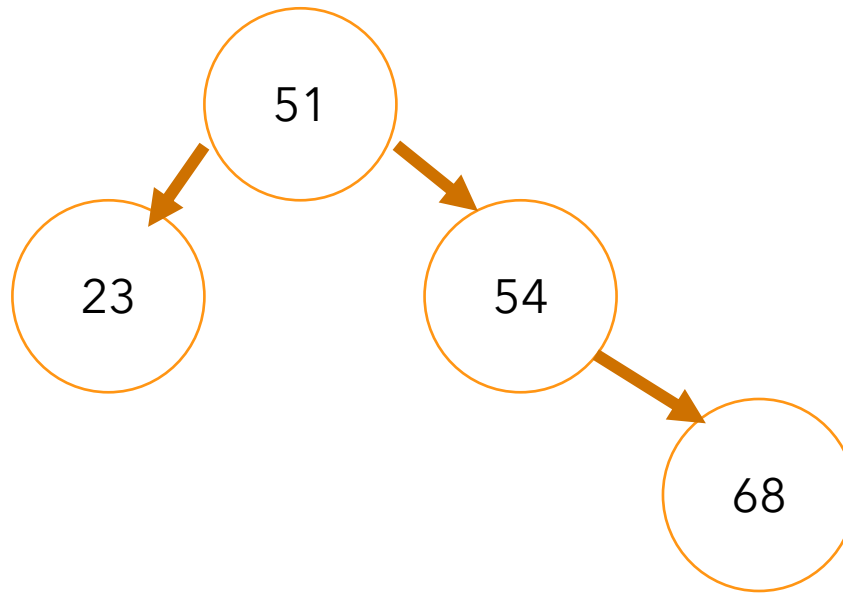
❑ Crear el árbol con las claves: 51 – 23 – 54 – 68 – 22 – 73 – 59 – 30



# Arboles binarios de búsqueda - ABB

## Ejemplo:

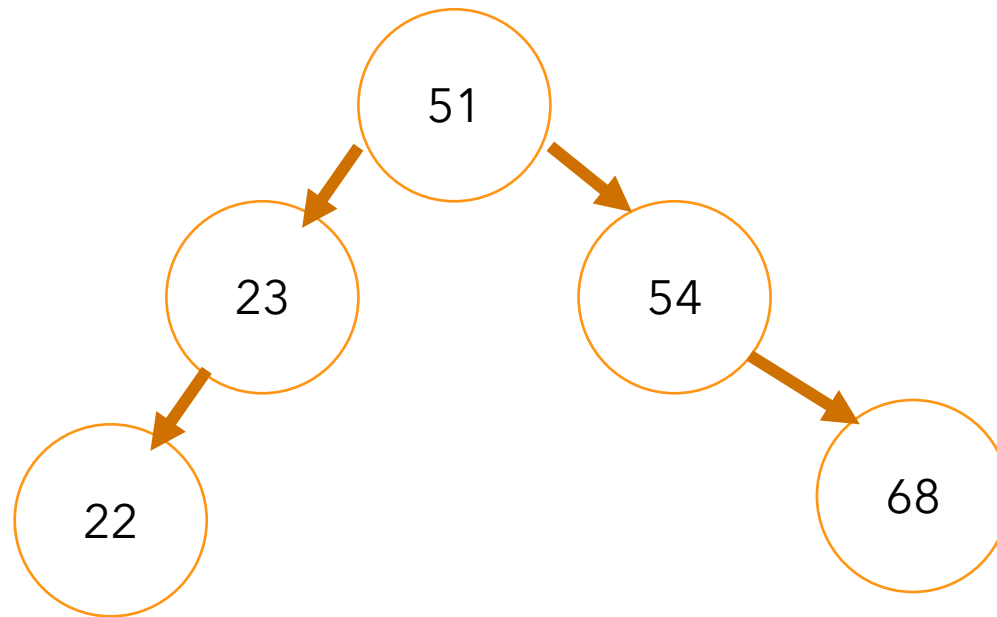
❑ Crear el árbol con las claves: 51 – 23 – 54 – 68 – 22 – 73 – 59 – 30



# Arboles binarios de búsqueda - ABB

## Ejemplo:

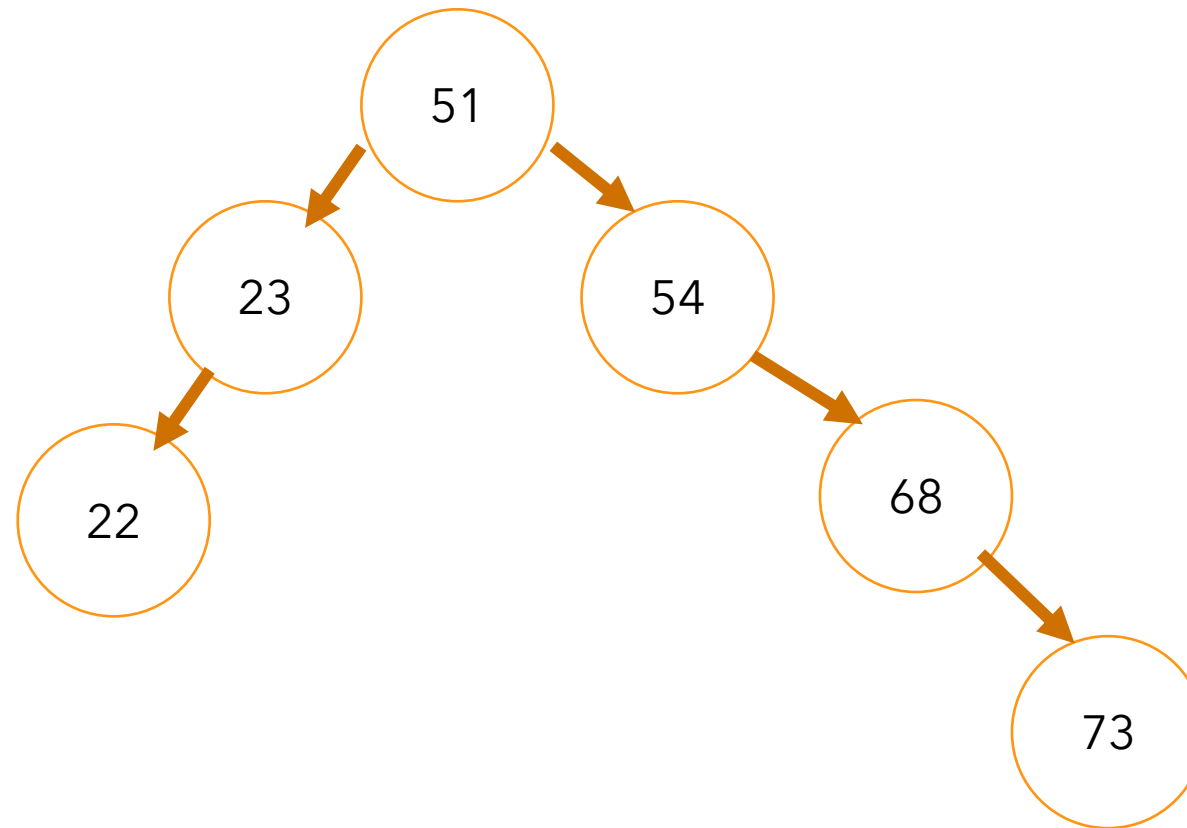
❑ Crear el árbol con las claves: 51 – 23 – 54 – 68 – 22 – 73 – 59 – 30



# Arboles binarios de búsqueda - ABB

## Ejemplo:

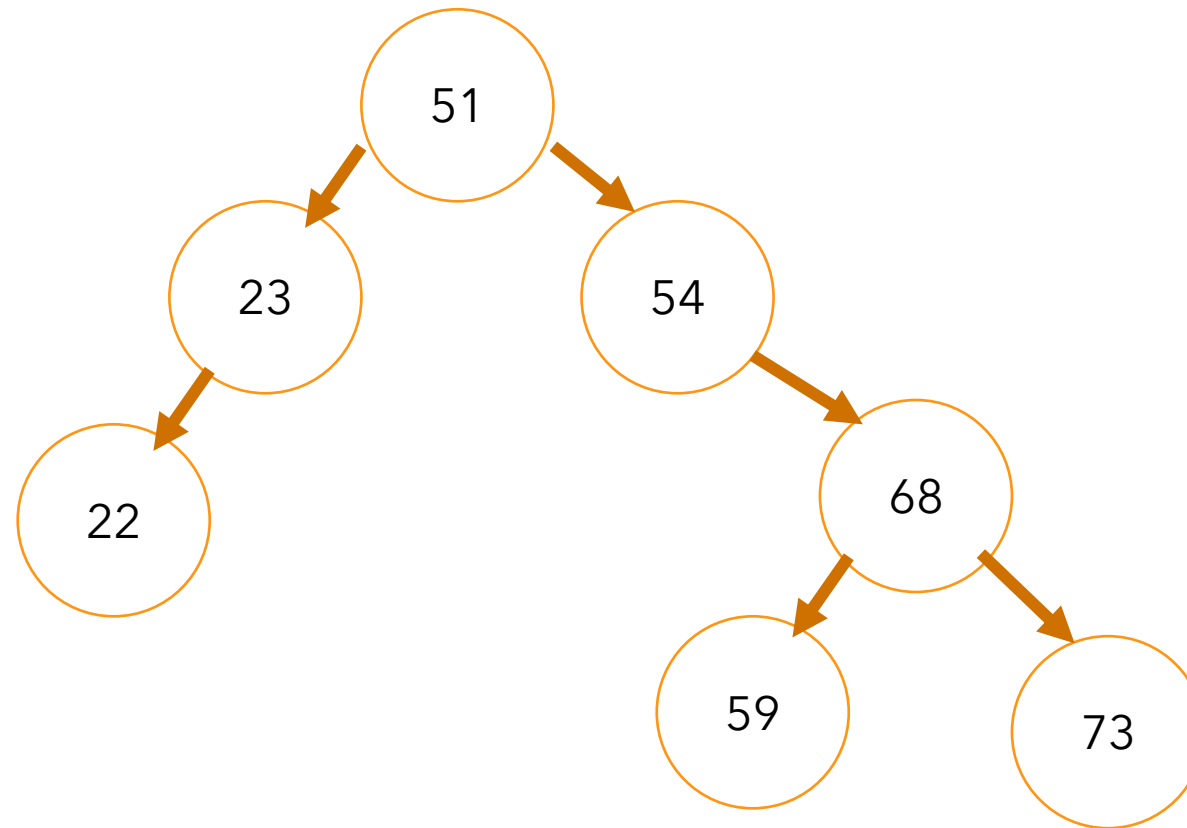
❑ Crear el árbol con las claves: 51 – 23 – 54 – 68 – 22 – 73 – 59 – 30



# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Crear el árbol con las claves: 51 – 23 – 54 – 68 – 22 – 73 – 59 – 30

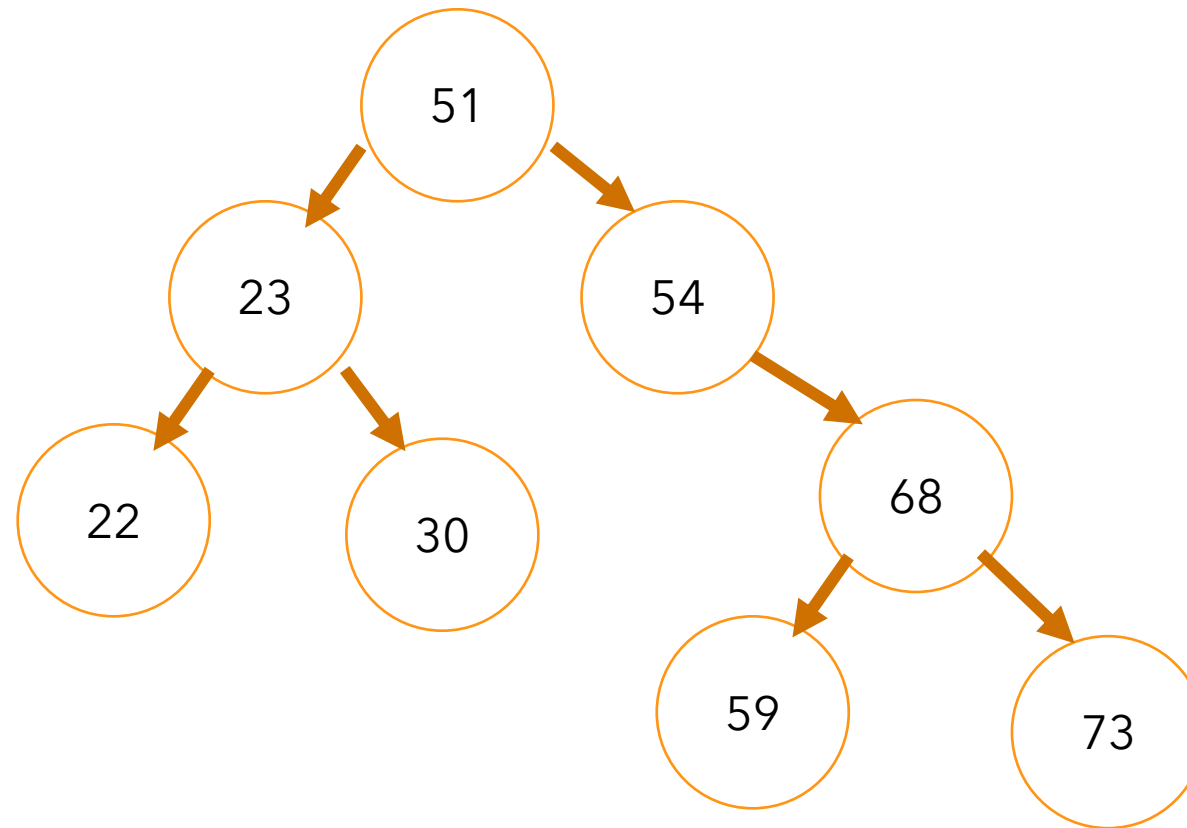




# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Crear el árbol con las claves: 51 – 23 – 54 – 68 – 22 – 73 – 59 – 30



# Arboles binarios de búsqueda - ABB

## Implementación

Algoritmo:

1. Crear un nuevo BSTEntry O con dato = e y key=k
2. Si el árbol está vacío, O será la raíz; por tanto, se invoca el método addRoot(Object e) de la clase BinaryTree
3. En caso contrario, de forma recursiva, se recorre el subárbol izquierdo si la clave es menor a la raíz, o el subárbol derecho si la clave es mayor a la raíz

```
insert(Object e, int k)
1. BSTEntry O = new BSTEntry(e,k)
2. if isEmpty()
3.     super.addRoot(0)
4. else
5.     addEntry(root,0)
```

Creamos BSTEntry con el objeto y la clave ingresadas como parámetros

Si no hay nodos en el árbol, O será la raíz

addEntry: método auxiliar recursivo

# Arboles binarios de búsqueda - ABB

```
insert(Object e, int k)
```

1. BSTEntry 0 = new BSTEntry(e,k)
2. if isEmpty()
3.     super.addRoot(0)
4. else
5.     addEntry(root,0)

```
addEntry(Node v, BSTEntry o)
```

1. BSTEntry temp = v.getData()
2. Node nD = new Node(o)
3. if o.getKey() < temp.getKey()
4.     if hasLeft(v)
5.         addEntry(left(v), o)
6.     else
7.         v.setLeft(nD)
8. else
9.     if hasRight(v)
10.         addEntry(right(v), o)
11.     else
12.         v.setRight(nD)

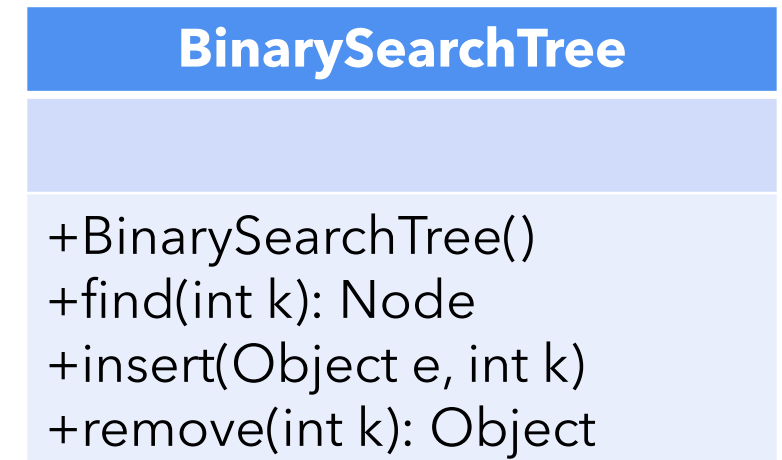
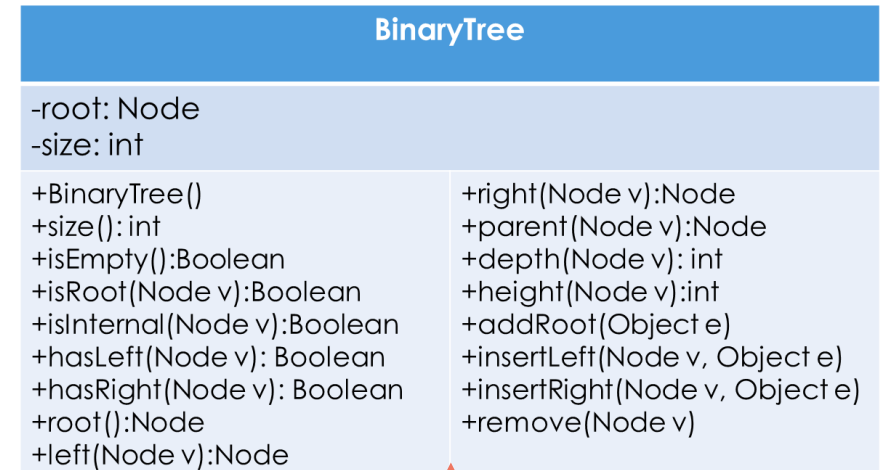
# Arboles binarios de búsqueda - ABB

## Implementación

Remove(int k): eliminar del árbol binario de búsqueda el nodo con clave k y retorna el objeto almacenado

### Algoritmo

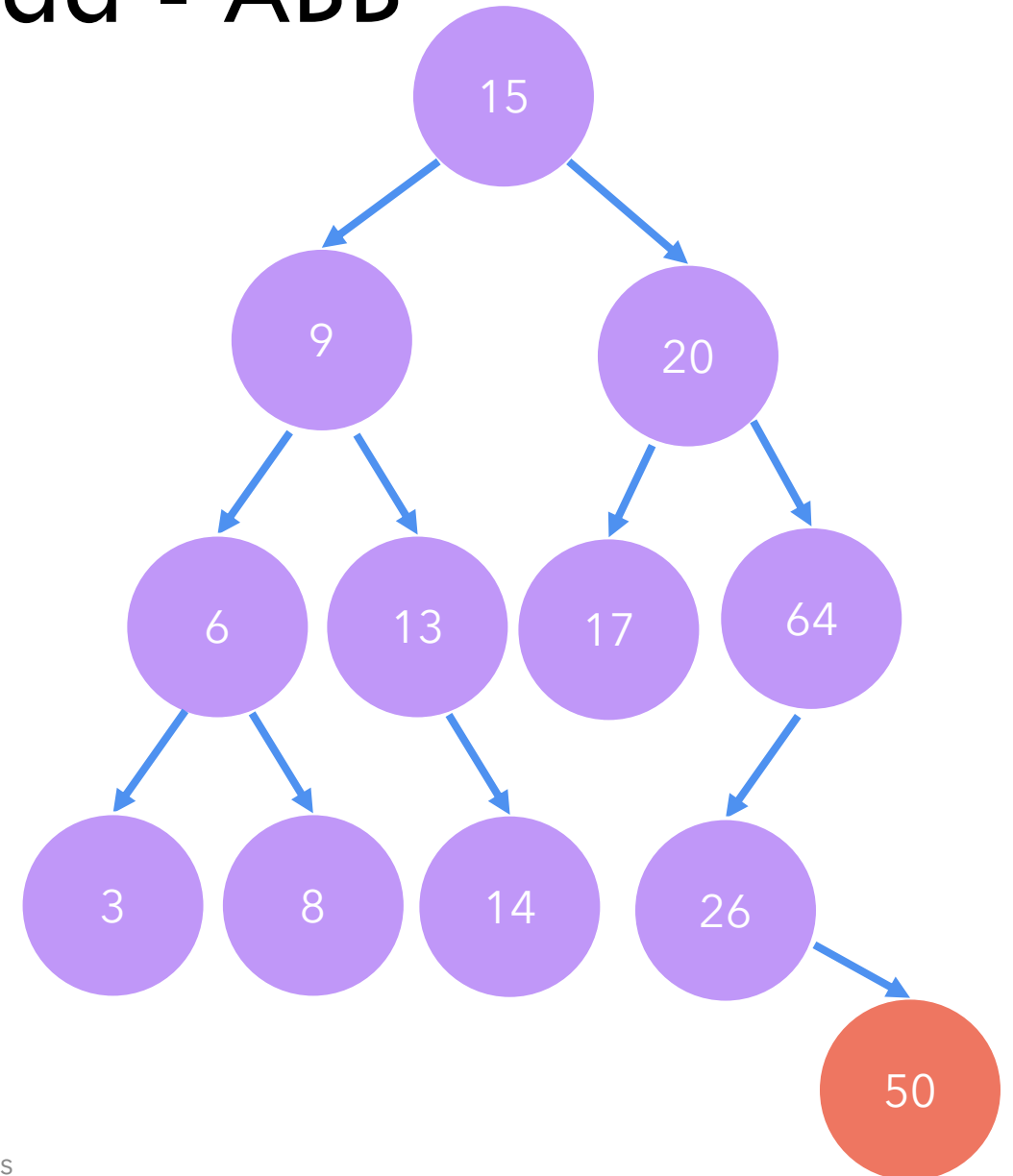
1. Se busca el nodo v con clave k
2. Si el nodo v tiene un solo hijo o no tiene hijos, empleamos el método remove() en la clase BinaryTree
3. Si el nodo v tiene dos hijos, debemos reemplazar el nodo v por su predecesor



# Arboles binarios de búsqueda - ABB

## Ejemplo:

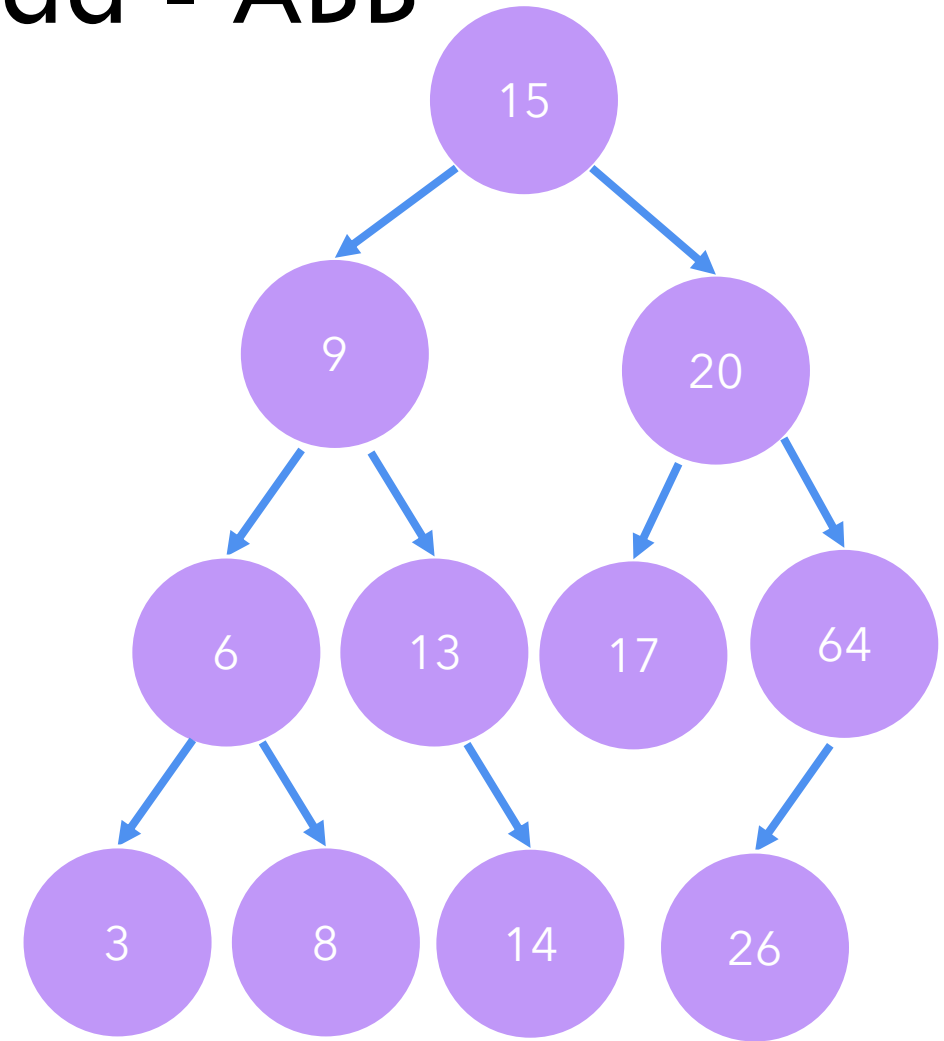
❑ Eliminar nodo con clave igual a 50



# Arboles binarios de búsqueda - ABB

## Ejemplo:

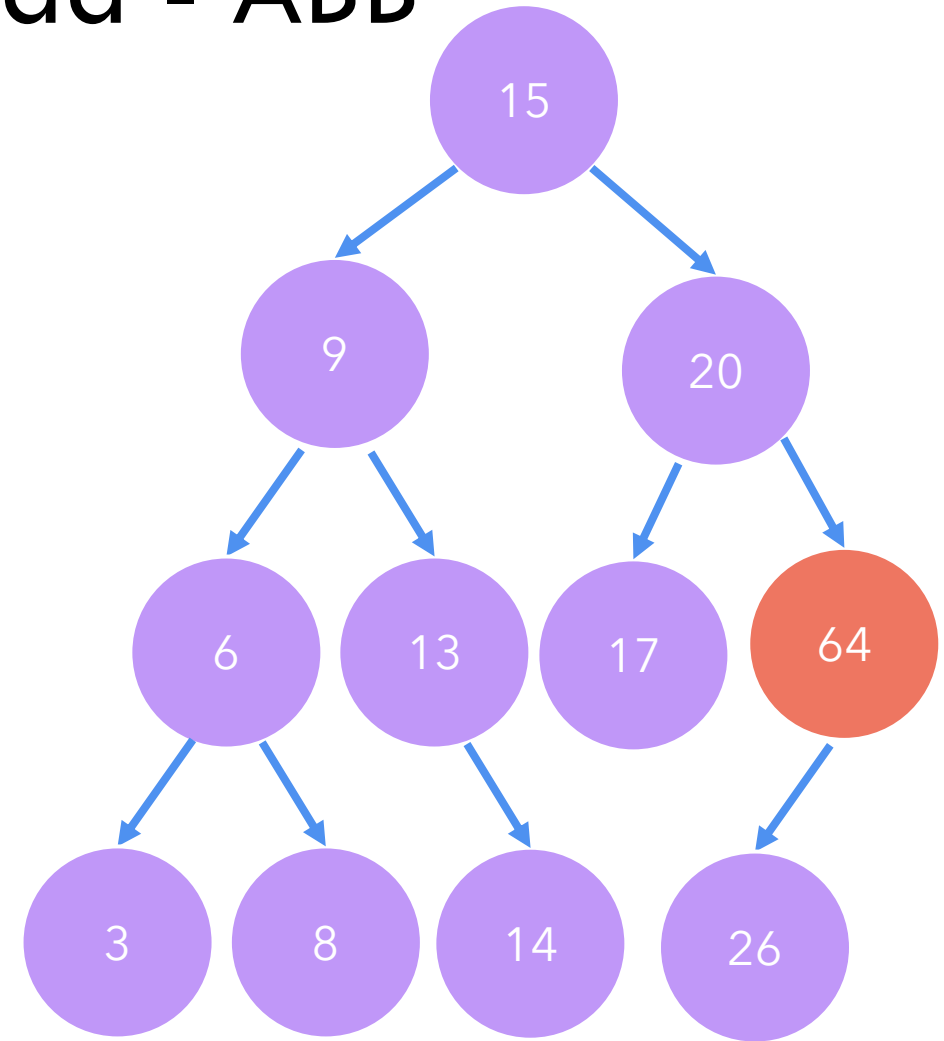
❑ Eliminar nodo con clave igual a 50



# Arboles binarios de búsqueda - ABB

## Ejemplo:

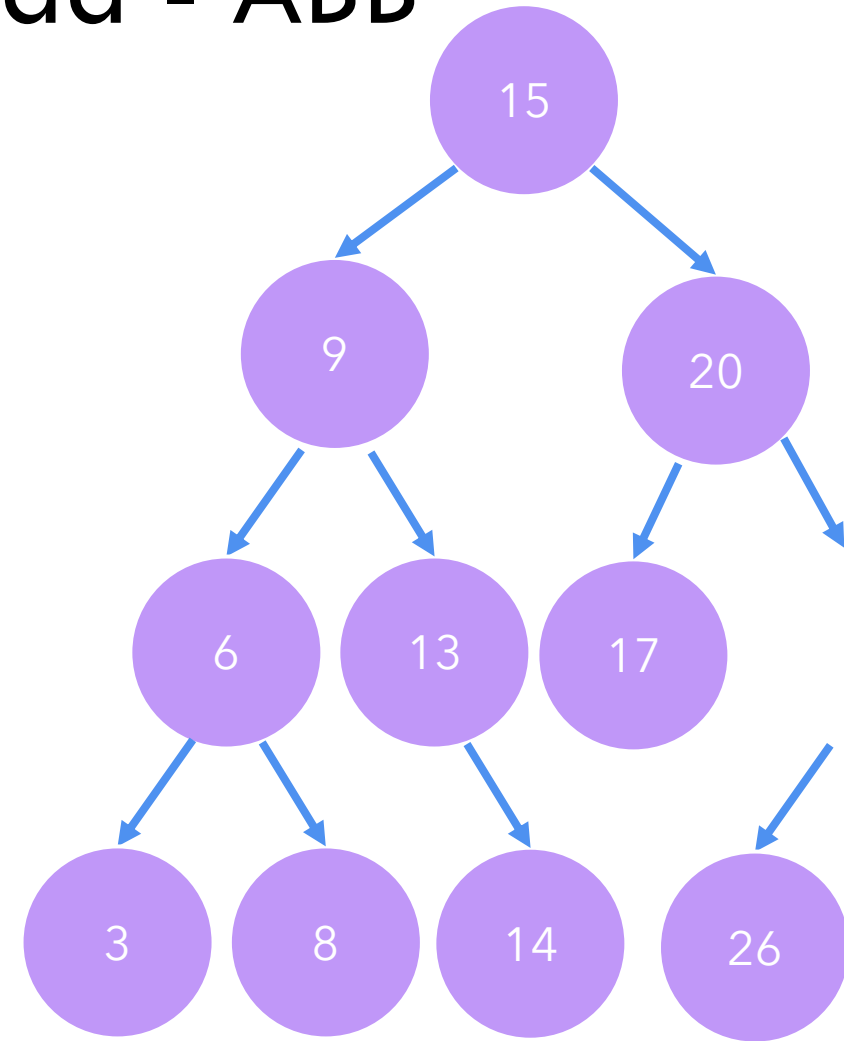
❑ Eliminar nodo con clave igual a 64



# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Eliminar nodo con clave igual a 64

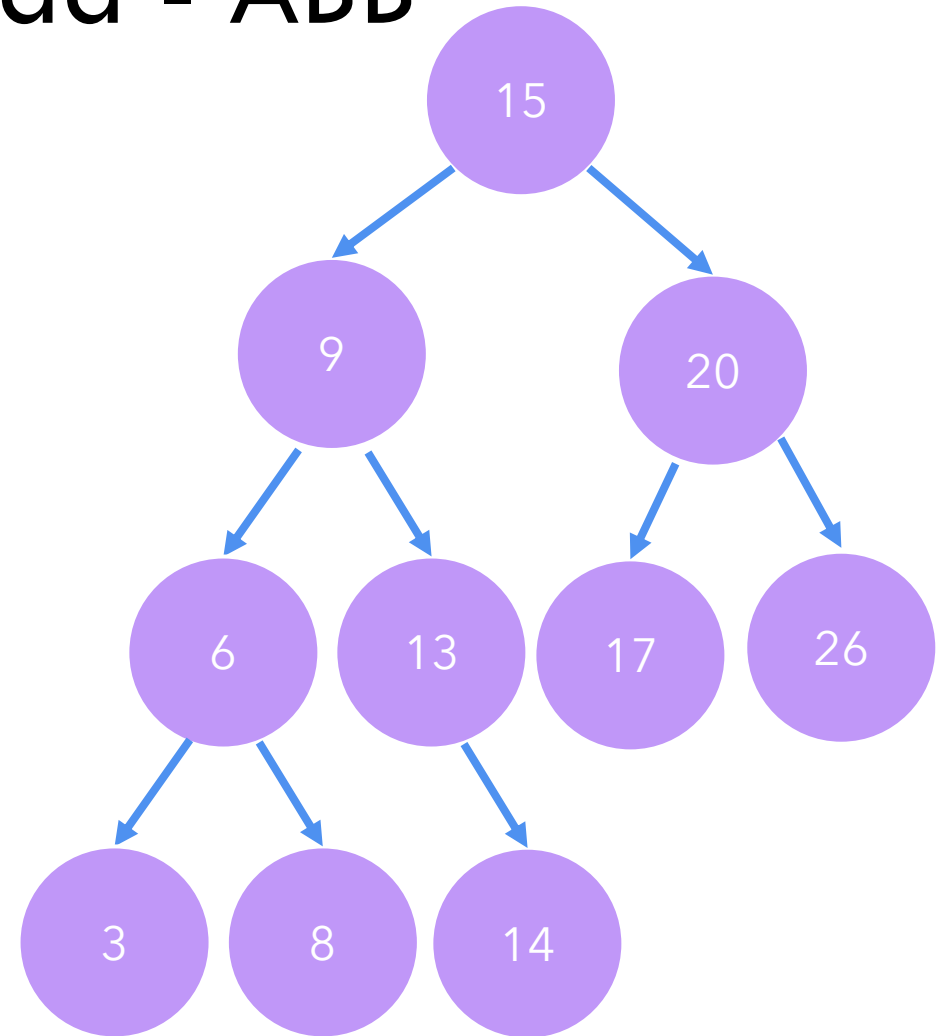




# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Eliminar nodo con clave igual a 64



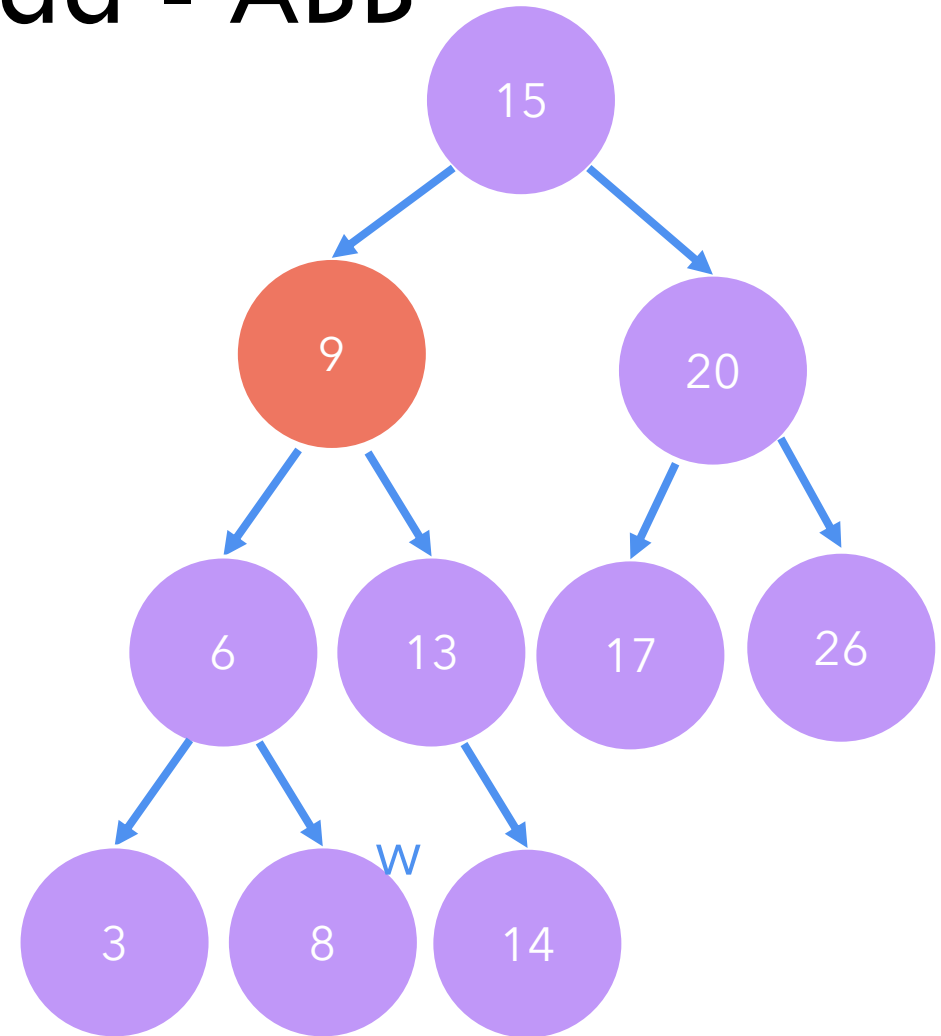
# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Eliminar nodo con clave igual a 9

### Algoritmo

- Si el nodo  $v$  tiene dos hijos, debemos remplazar el nodo  $v$  por su predecesor
1. Si se organiza los nodos de menor a mayor, el predecesor es el nodo anterior a  $v$
  2. El nodo  $w$  predecesor de  $v$  es el valor máximo del subárbol izquierdo de  $v$
  3. El predecesor  $w$  puede tener un hijo izquierdo, en tal caso, el hijo izquierdo toma la posición de  $w$



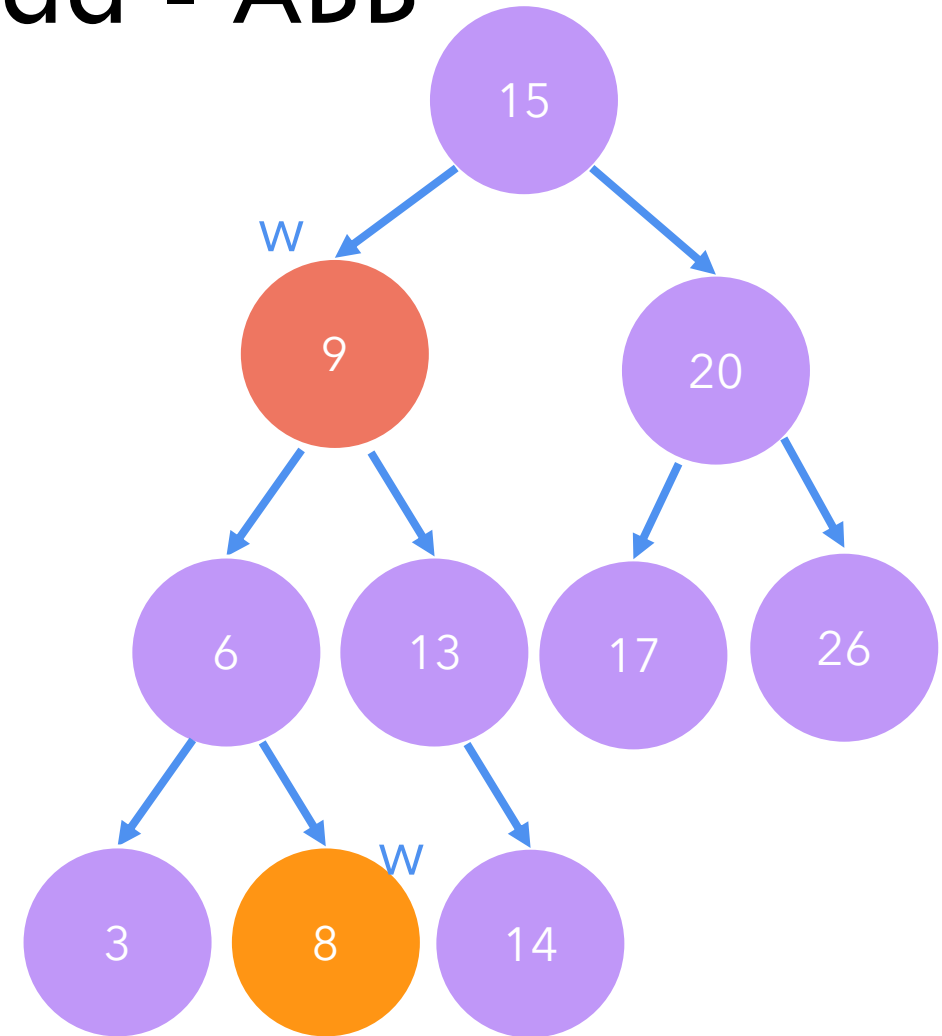
# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Eliminar nodo con clave igual a 9

### Algoritmo

- Si el nodo  $v$  tiene dos hijos, debemos remplazar el nodo  $v$  por su predecesor
1. Si se organiza los nodos de menor a mayor, el predecesor es el nodo anterior a  $v$
  2. El nodo  $w$  predecesor de  $v$  es el valor máximo del subárbol izquierdo de  $v$
  3. El predecesor  $w$  puede tener un hijo izquierdo, en tal caso, el hijo izquierdo toma la posición de  $w$



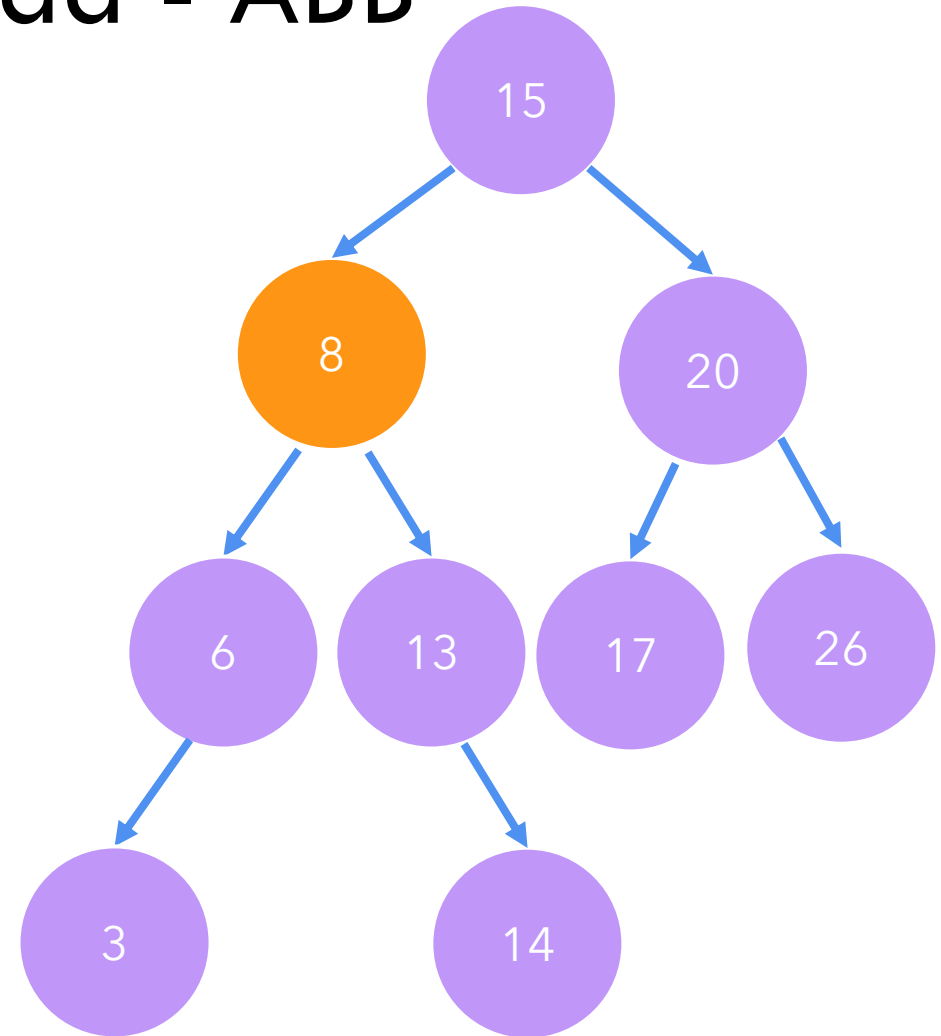
# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Eliminar nodo con clave igual a 9

### Algoritmo

- Si el nodo  $v$  tiene dos hijos, debemos remplazar el nodo  $v$  por su predecesor
1. Si se organiza los nodos de menor a mayor, el predecesor es el nodo anterior a  $v$
  2. El nodo  $w$  predecesor de  $v$  es el valor máximo del subárbol izquierdo de  $v$
  3. El predecesor  $w$  puede tener un hijo izquierdo, en tal caso, el hijo izquierdo toma la posición de  $w$



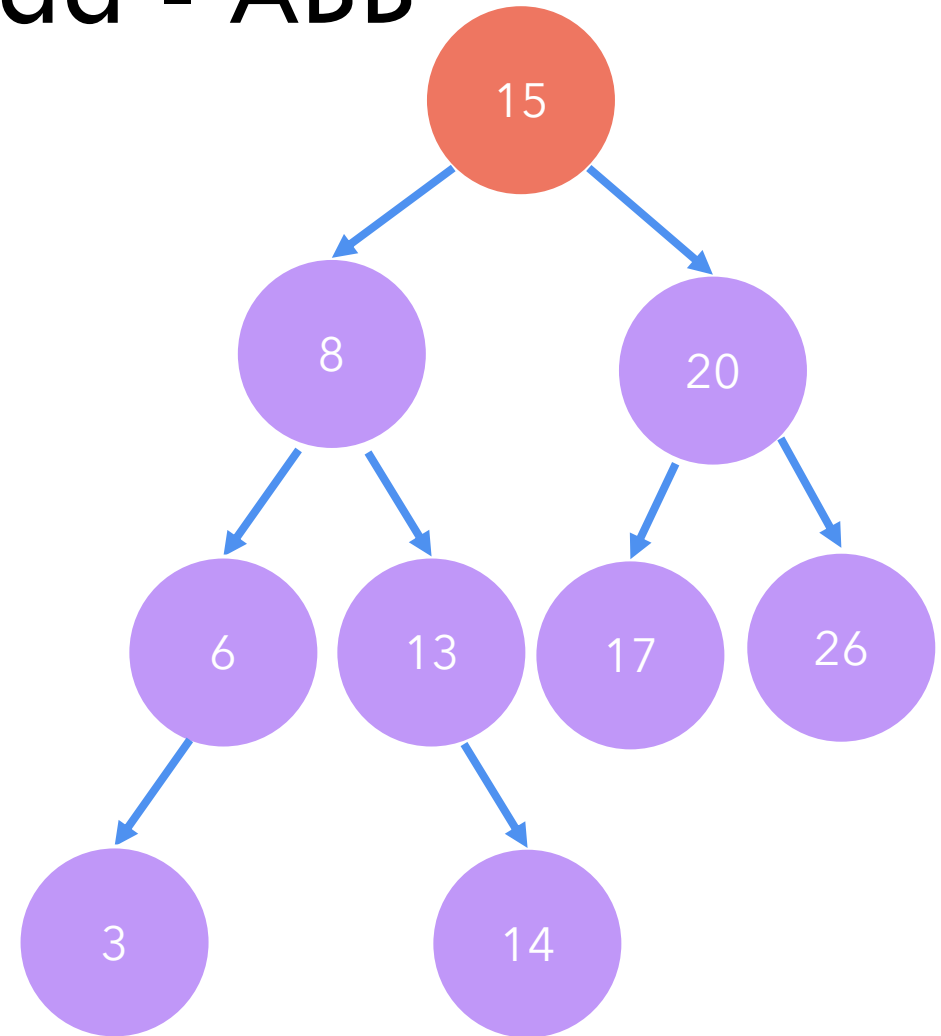
# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Eliminar nodo con clave igual a 15

### Algoritmo

- Si el nodo  $v$  tiene dos hijos, debemos remplazar el nodo  $v$  por su predecesor
1. Si se organiza los nodos de menor a mayor, el predecesor es el nodo anterior a  $v$
  2. El nodo  $w$  predecesor de  $v$  es el valor máximo del subárbol izquierdo de  $v$
  3. El predecesor  $w$  puede tener un hijo izquierdo, en tal caso, el hijo izquierdo toma la posición de  $w$



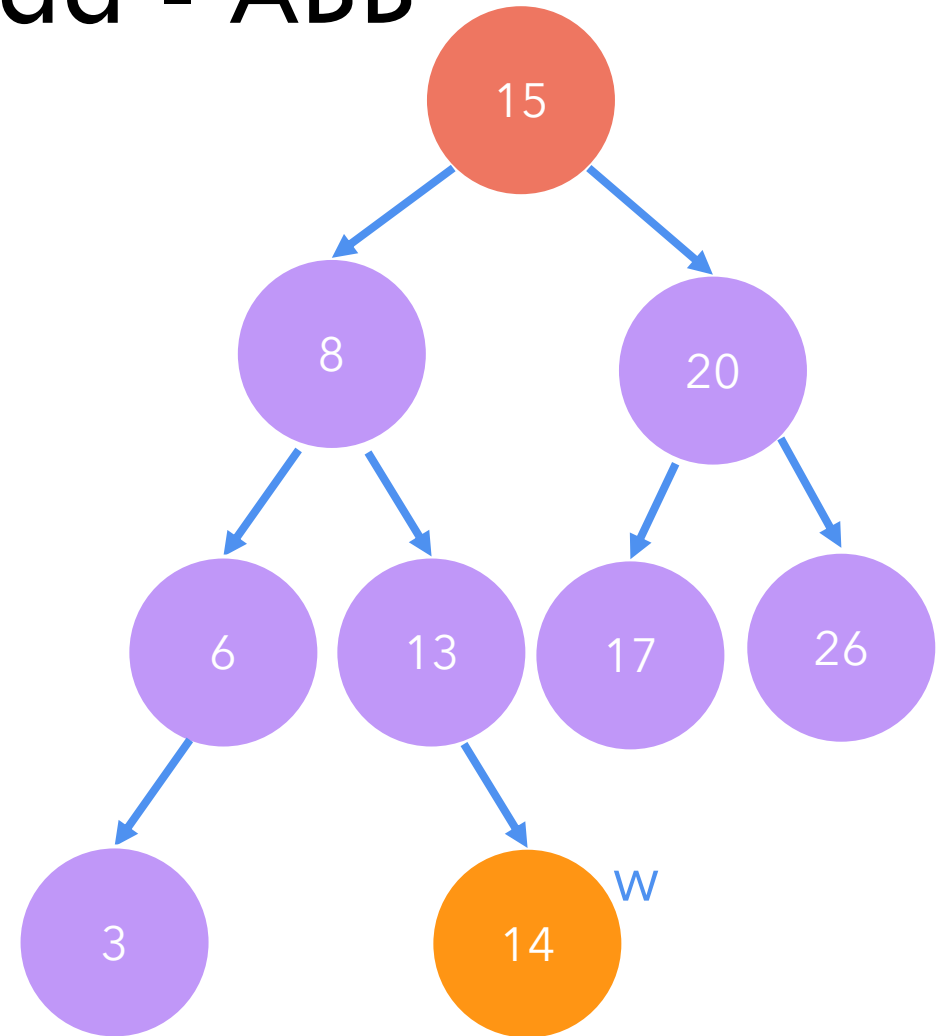
# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Eliminar nodo con clave igual a 15

### Algoritmo

- Si el nodo  $v$  tiene dos hijos, debemos remplazar el nodo  $v$  por su predecesor
1. Si se organiza los nodos de menor a mayor, el predecesor es el nodo anterior a  $v$
  2. El nodo  $w$  predecesor de  $v$  es el valor máximo del subárbol izquierdo de  $v$
  3. El predecesor  $w$  puede tener un hijo izquierdo, en tal caso, el hijo izquierdo toma la posición de  $w$



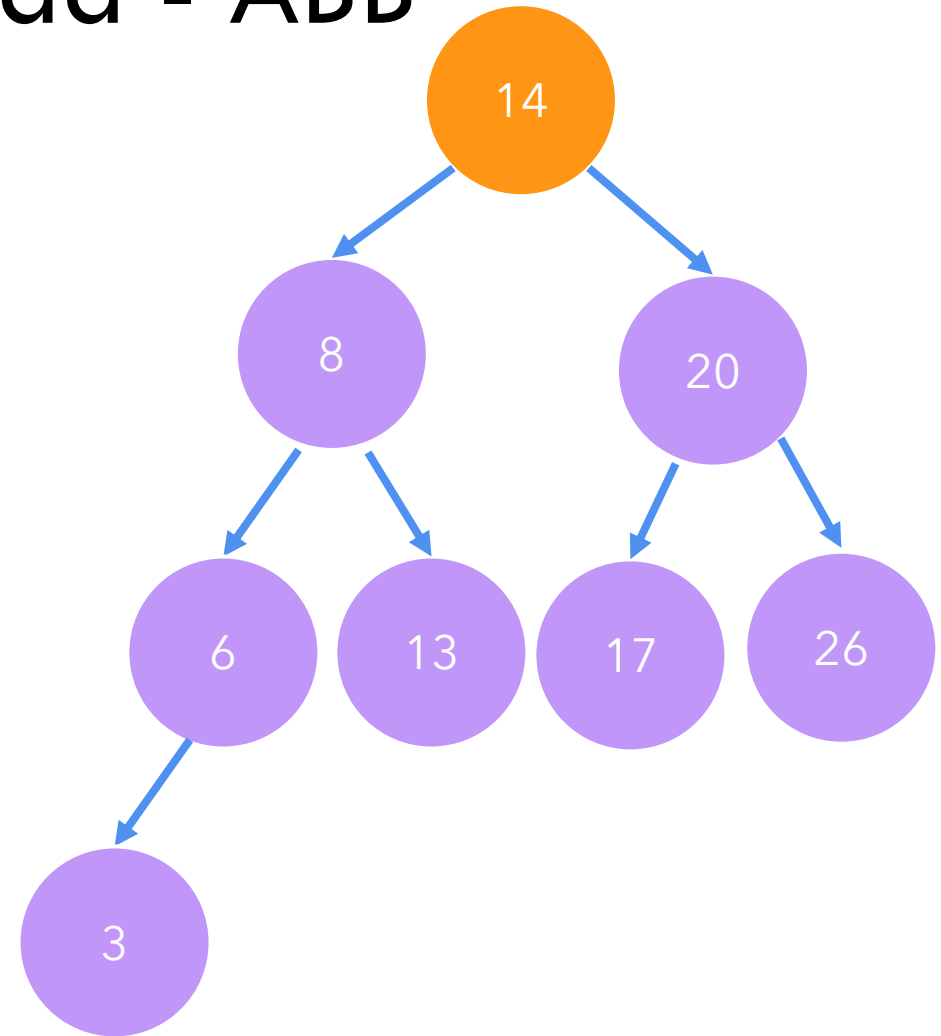
# Arboles binarios de búsqueda - ABB

## Ejemplo:

❑ Eliminar nodo con clave igual a 15

### Algoritmo

- Si el nodo  $v$  tiene dos hijos, debemos remplazar el nodo  $v$  por su predecesor
1. Si se organiza los nodos de menor a mayor, el predecesor es el nodo anterior a  $v$
  2. El nodo  $w$  predecesor de  $v$  es el valor máximo del subárbol izquierdo de  $v$
  3. El predecesor  $w$  puede tener un hijo izquierdo, en tal caso, el hijo izquierdo toma la posición de  $w$



# Arboles binarios de búsqueda - ABB

## Implementación

Remove(int k)

### Algoritmo

1. Se busca el nodo v con clave k
2. Si el nodo v tiene un solo hijo o no tiene hijos, empleamos el método remove() en la clase BinaryTree
3. Si el nodo v tiene dos hijos, debemos reemplazar el nodo v por su predecesor

```
Remove(int k)
```

```
1.Node v = find(k)
```

```
2.temp = v.getData()
```

```
3.if hasLeft(v)&&hasRight(v) //caso 2
```

```
...
```

```
4.else //caso 1
```

```
5.    super.remove(v)
```

```
6.return temp
```



# Arboles binarios de búsqueda - ABB

Remove(int k)

1. Node v = find(k)
2. temp = v.getData()
3. if hasLeft(v)&&hasRight(v) //caso 2
4.     Node w = predecesor(v)
5.     v.setData(w.getData())
6.     super.remove(w)
7. else //caso 1
8.     super.remove(v)
9. return temp

predecesor(Node v)

1. Node temp = temp.getLeft()
2. return maxNode(temp)

maxNode(Node temp)

1. if hasRight(temp)
2.     return maxNode(right(temp))
3. else
4.     return temp

# Arboles binarios de búsqueda - ABB

Operación	Complejidad
parent()	$O(n)$
depth()	$O(n^2)$ Árbol completo: $O(n \lg n)$
height()	$O(n)$ Árbol completo: $O(\lg n)$
addRoot()	$\Theta(1)$
insertLeft()	$\Theta(1)$
insertRight()	$\Theta(1)$
remove()	$O(n)$ Árbol completo: $O(\lg n)$
Find()	Árbol completo: $O(h)$ Árbol completo: $O(\lg n)$
Insert()	Árbol completo: $O(h)$ Árbol completo: $O(\lg n)$

BinaryTree	
-root: Node -size: int	
+BinaryTree() +size(): int +isEmpty(): Boolean +isRoot(Node v): Boolean +isInternal(Node v): Boolean +hasLeft(Node v): Boolean +hasRight(Node v): Boolean +root(): Node +left(Node v): Node	+right(Node v): Node +parent(Node v): Node +depth(Node v): int +height(Node v): int +addRoot(Object e) +insertLeft(Node v, Object e) +insertRight(Node v, Object e) +remove(Node v)

BinarySearchTree
+BinarySearchTree() +find(int k): Node +insert(Object e, int k) +remove(int k): Object