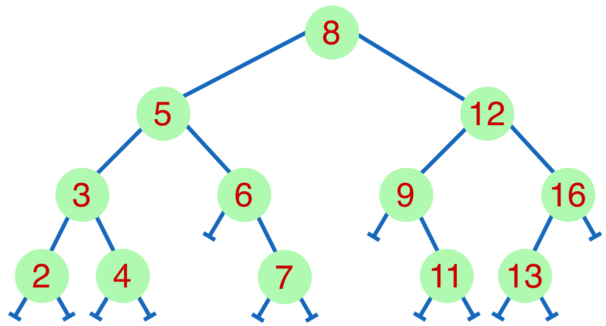
Árboles Binarios de Búsqueda

Profesores Estructuras de Datos, 2024.
Dpto. Lenguajes y Ciencias de la Computación.
University of Málaga

Árboles Binarios de Búsqueda (Binart Search Tree, BST) y la Propiedad de Orden de BST (BSTOP)

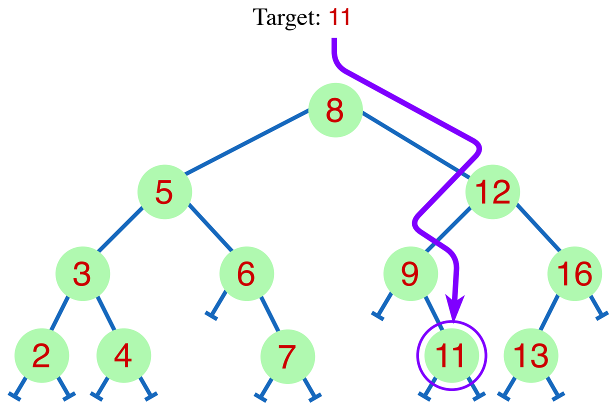
Un Árbol Binario de Búsqueda (BST) es un árbol **binario** de elementos únicos (sin repeticiones) que satisface la Propiedad de Orden de Árbol Binario de Búsqueda (BSTOP):

- Para cada nodo en el árbol:
  - Todos los nodos en su subárbol izquierdo tienen valores menores que el valor del nodo y
  - Todos los nodos en su subárbol derecho tienen valores mayores que el valor del nodo.
- **Consecuencias de BSTOP**:
  - Es posible buscar, insertar y eliminar un elemento en el árbol en tiempo O(h), donde h es la altura del árbol.
  - Los elementos mínimo y máximo son los nodos más a la izquierda y más a la derecha.
  - Un recorrido en orden del árbol visita los nodos en orden ordenado.
- **Aplicaciones de los Árboles Binarios de Búsqueda**:
  - Conjuntos, bolsas y diccionarios pueden ser implementados eficientemente usando BSTs.



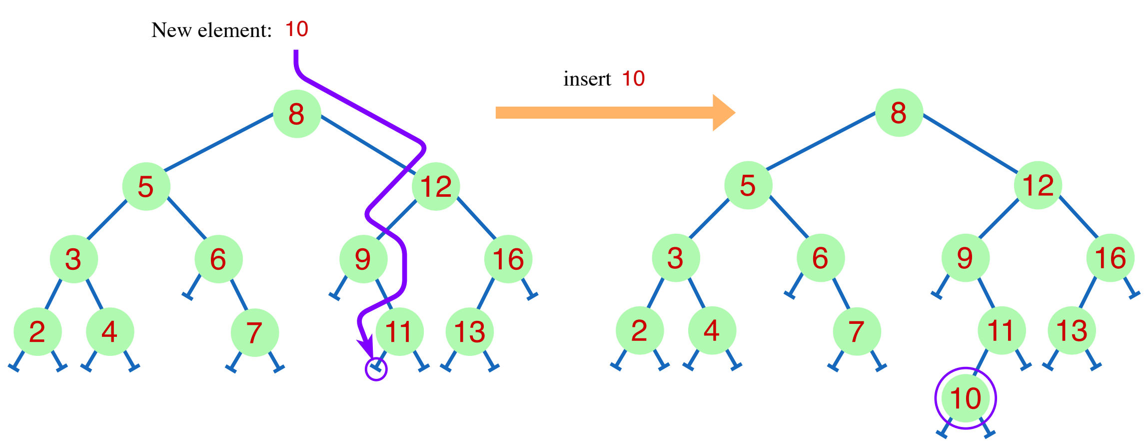
Búsqueda de un Elemento en un BST

- **Objetivo**: Buscar un elemento dado (el objetivo) en un BST.
- **Precondición**: El árbol es un BST (cumple **BSTOP**).
- **Postcondición**: Devolver: encontrado si el objetivo está; si no, no encontrado.
- **Algoritmo**:
  i. Comenzar en la raíz del árbol.
  ii. Si el árbol está vacío, devolver no encontrado.
  iii. Si la raíz es igual al objetivo, devolver encontrado.
  iv. Si el objetivo es menor que raíz, buscar el objetivo en el subárbol **izquierdo**.
  v. Si no, el objetivo es mayor que la raíz; buscar en el subárbol **derecho**.
- **Complejidad**: O(h), donde h es la altura del árbol. Sólo hacemos una única comparación en cada nivel del árbol.



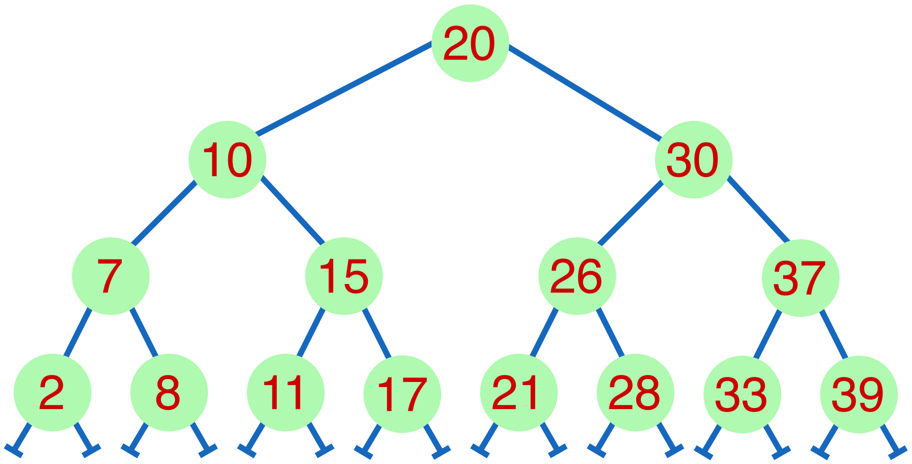
Inserción de un Elemento en un BST

- **Objetivo**: Insertar un nuevo elemento en un BST manteniendo su unicidad de elementos y propiedades de orden.
- **Precondición**: El árbol es un BST.
- **Postcondición**: El árbol sigue siendo un BST después de insertar el nuevo elemento.
- **Algoritmo**:
  i. Comenzar en la raíz del árbol.
  ii. Si el árbol está vacío, insertar el nuevo elemento como raíz e incr. el tamaño.
  iii. Si el nuevo elemento es igual a la raíz, reemplazar la raíz con el nuevo elemento.
  iv. Si el nuevo elemento es menor que la raíz, insertarlo en el subárbol **izquierdo**.
  v. Si no, el nuevo elemento es mayor que la raíz; insertarlo en el subárbol **derecho**.
- **Complejidad**: O(h), donde h es la altura del árbol ya que hacemos una única comparación en cada nivel del árbol.



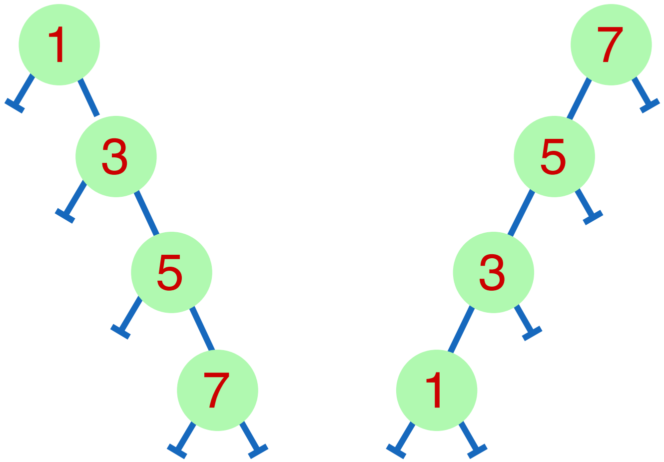
Altura de un BST

- La complejidad de las operaciones en un BST depende de la altura del árbol.
- La altura de un BST impacta significativamente la eficiencia de sus operaciones.
- En el mejor de los casos, el BST es un árbol binario perfecto y su altura es O(log n), donde n es el número de nodos en el árbol.



Altura de un BST (II)

- En el peor de los casos, el BST es un árbol degenerado. Cada nodo tiene solo un hijo, y la altura es O(n), donde n es el número de nodos en el árbol.
- Esto puede suceder si los elementos se insertan en orden.



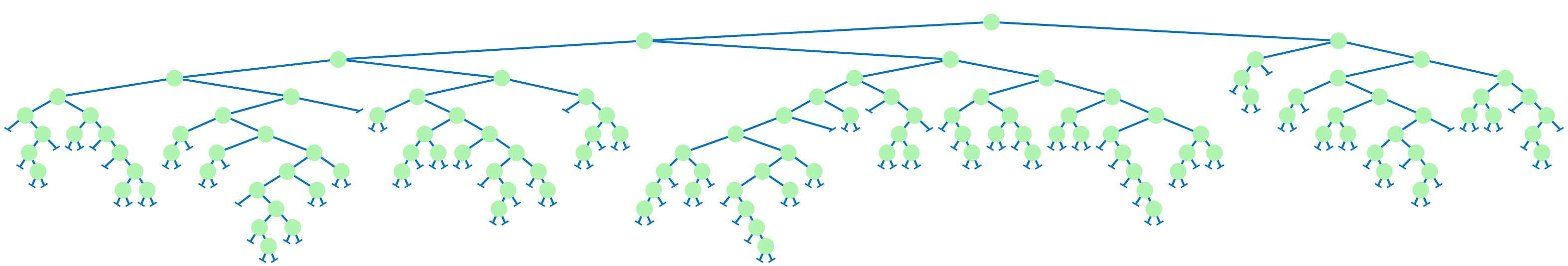
- En este caso, el rendimiento de las operaciones en el árbol se degrada y se comporta en términos de eficiencia similar a una lista enlazada.

Altura de un BST (III)

```
Binario perfecto O(log(n)) <= h <= O(n) Degradado
```

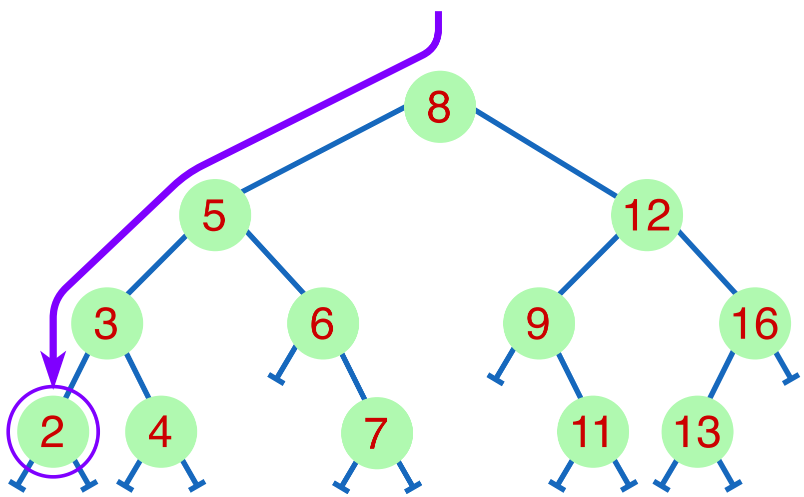
Altura de un BST (IV)

- En la práctica, si insertamos los elementos en un orden aleatorio, la altura del árbol será cercana a O(log n) y el árbol funcionará de manera eficiente.
- Esta imagen muestra un BST construido insertando 128 elementos aleatorios ¡Su altura es solo 13! ... $7 \leq h = 13 \leq 128$



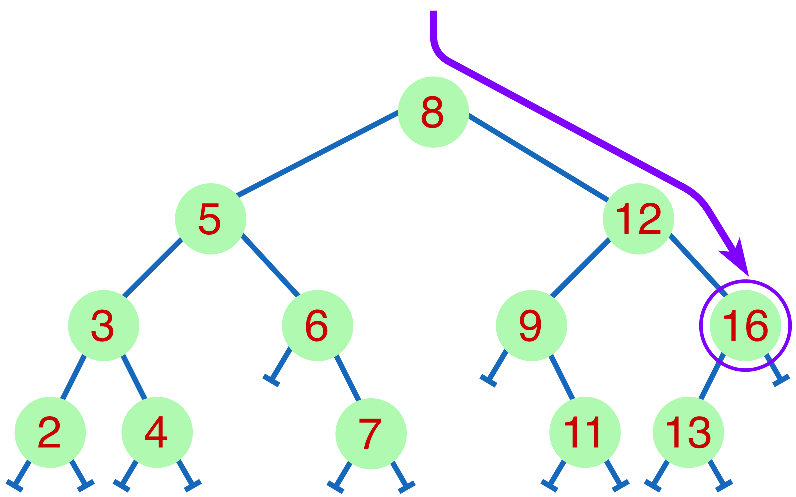
Búsqueda del Elemento Mínimo en un BST

- **Objetivo**: Devolver el elemento mínimo en un BST.
- **Precondición**: El árbol es un BST.
- **Postcondición**: Devolver el elemento mínimo en el árbol.
- **Algoritmo**:
  i. Si el árbol está vacío, devolver no hay elemento mínimo.
  ii. Comenzar en la raíz del árbol.
  iii. Si el subárbol izquierdo está vacío, devolver el valor del nodo.
  iv. De lo contrario, repetir este mismo proceso pero buscando el elemento mínimo en el subárbol izquierdo.
- **Complejidad**: O(h), donde h es la altura del árbol ya que descendemos por la espina izquierda del árbol y su longitud está limitada por la altura del árbol.



Búsqueda del Elemento Máximo en un BST

- **Objetivo**: Devolver el elemento máximo en un BST.
- **Precondición**: El árbol es un BST.
- **Postcondición**: Devolver el elemento máximo en el árbol.
- **Algoritmo**:
  i. Si el árbol está vacío, devolver no hay elemento máximo.
  ii. Comenzar en la raíz del árbol.
  iii. Si el subárbol derecho está vacío, devolver el valor del nodo.
  iv. De lo contrario, repetir este mismo proceso pero buscando el elemento máximo en el subárbol derecho.
- **Complejidad**: O(h), donde h es la altura del árbol ya que descendemos por la espina derecha del árbol y su longitud está limitada por la altura del árbol.

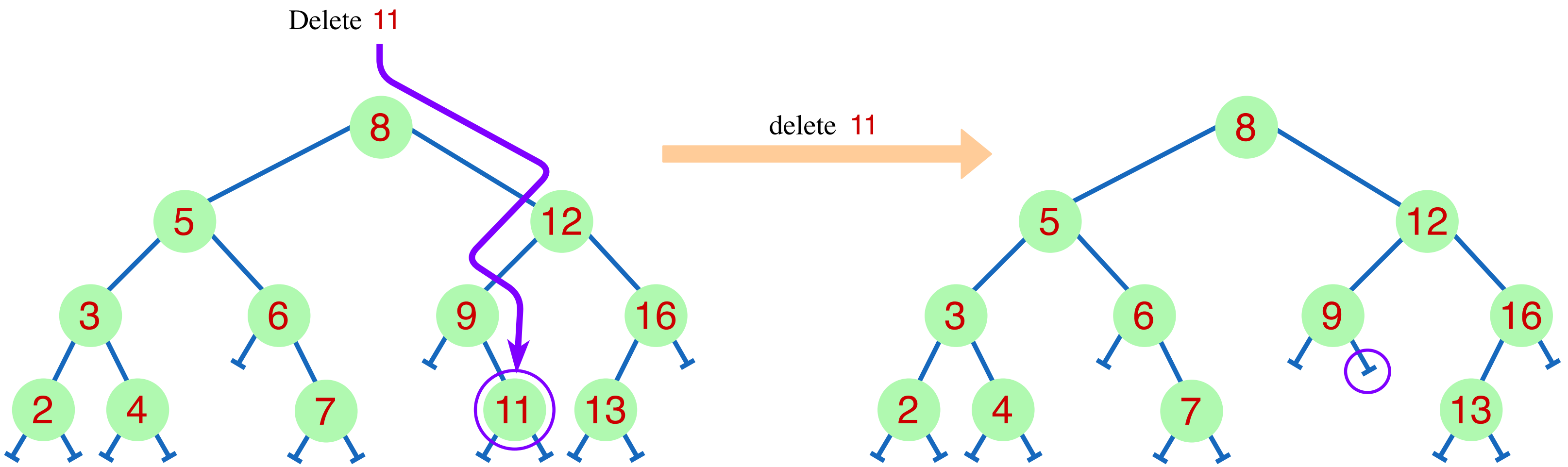


Eliminar un Elemento de un BST

- **Objetivo**: Eliminar un elemento de un BST manteniendo sus propiedades de orden.
- **Precondición**: El árbol es un BST.
- **Postcondición**: El árbol sigue siendo un BST y el elemento es eliminado.
- **Algoritmo**:
    i. Comenzar en la raíz del árbol.
    ii. Si el árbol está vacío, devolver el árbol.
    iii. Si el objetivo es menor que el valor de la raíz, eliminar el objetivo del subárbol izquierdo.
    iv. Si el objetivo es mayor que el valor de la raíz, eliminar el objetivo del subárbol derecho.
    v. Si el objetivo es igual al valor de la raíz:
        - Si la raíz no tiene hijos, eliminar la raíz.
        - Si la raíz tiene un hijo, reemplazar la raíz con su hijo.
        - Si la raíz tiene dos hijos, reemplazar la raíz con el elemento mínimo en el subárbol derecho y eliminar el elemento mínimo del subárbol derecho.
- **Complejidad**: O(h), donde h es la altura del árbol.

Eliminar. Elemento Eliminado sin Hijos

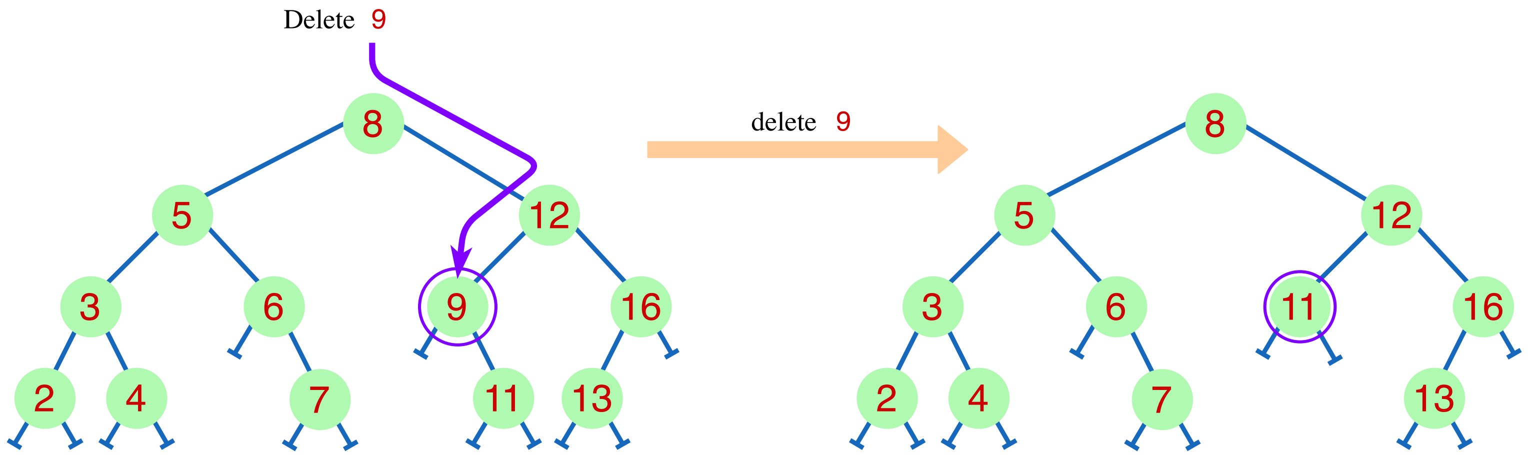
- Si el elemento a eliminar no tiene hijos, simplemente elimine el elemento.



- De manera trivial, mantiene la propiedad de orden del BST.

Eliminar. Elemento Eliminado con un Hijo

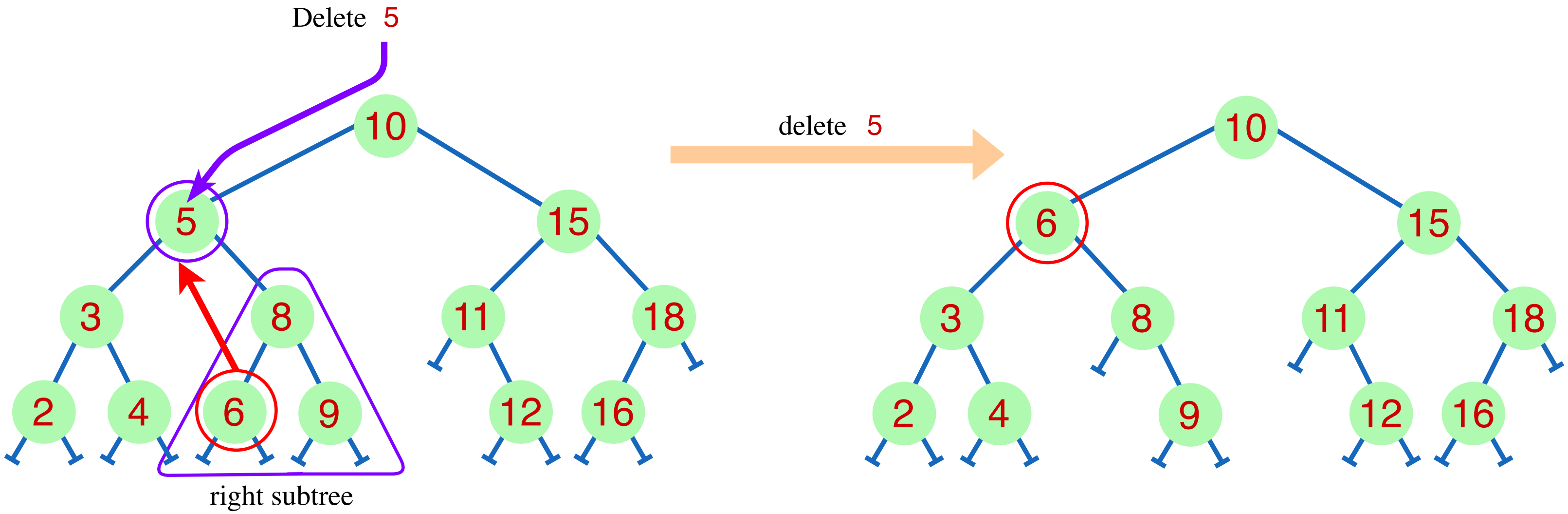
- Si el elemento a eliminar tiene un solo hijo, reemplácelo con su hijo.
- En este ejemplo, vamos a eliminar el elemento 9 del árbol:



- Este algoritmo mantiene la propiedad de orden del BST ya que el hijo está en la posición correcta con respecto al padre del nodo eliminado.

Eliminar. Elemento Eliminado con Dos Hijos

- Si el elemento a eliminar tiene dos hijos, reemplace el elemento con el elemento mínimo en el subárbol derecho y elimine el elemento mínimo del subárbol derecho.
- En este ejemplo, vamos a eliminar el elemento 5 del árbol:



- Este algoritmo mantiene la propiedad de orden del BST ya que el elemento mínimo en el subárbol derecho es mayor que los elementos en el subárbol izquierdo y menor que el resto de los elementos en el subárbol derecho.

El TAD Árbol de Búsqueda en Java

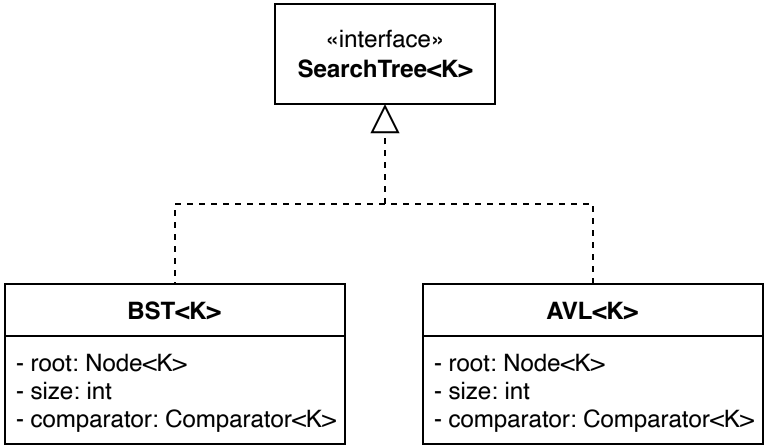
- La interfaz `SearchTree<K>` define un árbol de búsqueda que almacena elementos únicos de tipo `K`.
- El método `comparator` devuelve el comparador utilizado para comparar elementos, determinando el orden de los elementos en el árbol de búsqueda.

```java
package org.uma.ed.datastructures.searchtree;

public interface SearchTree<K> {
    Comparator<K> comparator(); // devuelve el comparador utilizado para comparar elementos
    boolean isEmpty();          // devuelve true si el árbol está vacío
    int size();                 // devuelve el número de elementos en el árbol
    int height();               // devuelve la altura del árbol
    void clear();               // elimina todos los elementos del árbol
    void insert(K key);         // inserta un nuevo elemento o reemplaza un elemento existente con la clave dada
    K search(K key);            // devuelve el elemento con la clave dada o null si no se encuentra
    boolean contains(K key);    // devuelve true si el elemento con la clave dada está en el árbol
    void delete(K key);         // elimina el elemento con la clave dada
    K minimum();                // devuelve el elemento mínimo en el árbol
    K maximum();                // devuelve el elemento máximo en el árbol
    void deleteMinimum();       // elimina el elemento mínimo del árbol
    void deleteMaximum();       // elimina el elemento máximo del árbol
    Iterable<K> inOrder();      // para iterar sobre los elementos en orden (in-order)
    Iterable<K> preOrder();     // para iterar sobre los elementos en preorden
    Iterable<K> postOrder();    // para iterar sobre los elementos en postorden
}
```

Implementaciones del TAD Árbol de Búsqueda

- Un árbol de búsqueda puede ser implementado usando diferentes estructuras de datos.
- Diferentes clases pueden implementar la interfaz `SearchTree<K>`:
  - `BST<K>`: Usa un árbol binario de búsqueda para almacenar elementos. La mayoría de las operaciones son O(h), donde h es la altura del árbol.
  - `AVL<K>`: Usa un árbol balanceado de Adelson-Velsky y Landis (AVL) para almacenar elementos. La mayoría de las operaciones son O(log n), donde n es el número de elementos en el árbol.



La clase `BST`

- `BST<K>` implementa la interfaz `SearchTree<K>` usando un árbol binario de búsqueda como representación.
- La clase anidada `Node` representa un nodo en el BST. Cada nodo almacena:
  - El elemento único (`key`) en el nodo.
  - Referencias a los hijos `left` y `right` .

```java
package org.uma.ed.datastructures.searchtree;

public class BST<K> implements SearchTree<K> {
    private static final class Node<K> {
        K key;                 // elemento en el nodo
        Node<K> left, right; // hijos izquierdo y derecho

        Node(K key) {
            this.key = key;
            this.left = null;
            this.right = null;
        }
    }

    ...
```

La clase `BST` (II)

- La clase `BST` mantiene:
  - Una referencia a `root` del BST. Esta referencia es `null` si el árbol está vacío.
  - Un `comparator` para comparar elementos.
  - El número de elementos (`size`) en el árbol.

```java
...

private final Comparator<K> comparator;
private Node<K> root;
private int size;

private BST(Comparator<K> comparator, Node<K> root, int size) {
    this.comparator = comparator;
    this.root = root;
    this.size = size;
}

public BST(Comparator<K> comparator) {
    this(comparator, null, 0);
}

...
```

Complejidad Computacional en `BST`

Operación	Coste
`BST.empty`	O(1)
`isEmpty`, `size`	O(1)
`clear`	O(1)
`insert`, `search`, `contains`, `delete`	O(h) †
`minimum`, `maximum`	O(h) †
`deleteMinimum`, `deleteMaximum`	O(h) †
`inOrder`, `preOrder`, `postOrder` iteraciones completas	O(n)

† La altura del árbol (h) está entre O(log n) y O(n).

`BSTSet` vs. `SortedLinkedSet` vs. `SortedArraySet`. Comparación Experimental

- Con `BST` se puede implementar un conjunto. Se ha comparado esta implementación con `SortedLinkedSet` y `SortedArraySet`.
- Medimos el tiempo para realizar 50000 operaciones (`insert`, `delete` y `contains`) usando elementos aleatorios en un conjunto inicialmente vacío.
- Usando una CPU Intel i7 860 y JDK 22:
  - `SortedArraySet` fue 8.15 veces más rápido que `SortedLinkedSet`.
  - `BSTSet` fue 304 veces más rápido que `SortedLinkedSet`.
- Luego hicimos la misma comparación pero realizando todas las inserciones en orden ascendente (BST degenerado).
  - `SortedArraySet` fue 28 veces más rápido que `SortedLinkedSet`.
  - `BSTSet` fue 1.8 veces más lento que `SortedLinkedSet`.