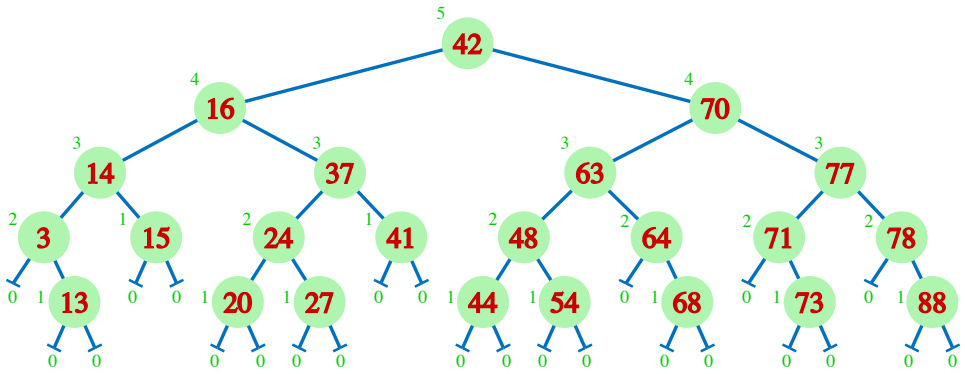
Árboles Andelsón-Velski y Landis (AVL)

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

Árboles AVL

- Los árboles AVL son un tipo de árbol binario de búsqueda autoequilibrado.
- Fueron la primera estructura de datos autoequilibrada (Adelson-Velsky y Landis, 1962).
- Los árboles AVL mantienen tanto la propiedad de orden de BST como la propiedad de equilibrio AVL:
  - Para cada nodo, las alturas de los subárboles izquierdo y derecho difieren en máximo 1.
- Como resultado, la altura de un árbol AVL es O(log n), donde n es el número de nodos.
- La siguiente imagen muestra un ejemplo de un árbol AVL:

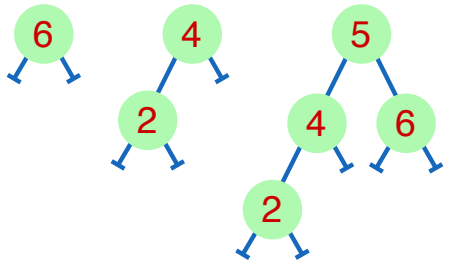


- Las alturas de los nodos se muestran en verde. Observe que tanto la propiedad de orden de BST como la propiedad de equilibrio AVL se cumplen.

Altura Logarítmica de los Árboles AVL

- Sea $N(h)$ el número mínimo de nodos en un árbol AVL de altura h.
- Esto se da por la siguiente relación de recurrencia:

$$\begin{aligned} N(1) &= 1 \\ N(2) &= 2 \\ N(h) &= 1 + N(h-1) + N(h-2), \text{ si h} > 2 \end{aligned}$$



AVL trees with minimum number of nodes for heights 1, 2, and 3

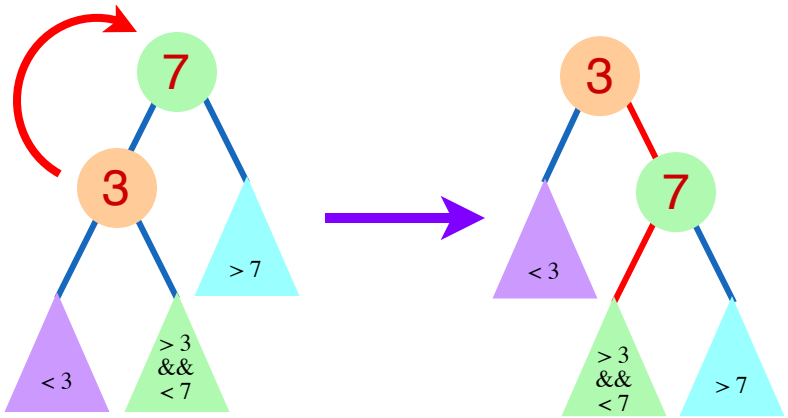
- Esta fórmula se deriva del hecho de que para mantener la propiedad de equilibrio AVL, un subárbol puede tener una altura de (h-1) y el otro de (h-2). El “+1” cuenta para el nodo raíz.
- Esta recurrencia es similar a la recurrencia de Fibonacci, y su solución es $N(h)$ = O(ϕ^h), donde $\phi = \frac{1+\sqrt{5}}{2}$ es el número áureo, por lo tanto:
  - h = O($log_\phi\ n$) y hemos demostrado que la altura de un árbol AVL es logarítmica.

Rotaciones en Árboles AVL

- Una rotación es una operación local que cambia la estructura del árbol mientras preserva la propiedad de orden de BST.
- Las rotaciones se utilizan para mantener la propiedad de equilibrio AVL al insertar o eliminar nodos.
- Hay dos tipos simples de rotaciones:
  - Rotación a la derecha: Mueve el hijo izquierdo de un nodo a la posición de su padre.
  - Rotación a la izquierda: Mueve el hijo derecho de un nodo a la posición de su padre.
- Estas rotaciones se pueden implementar eficientemente actualizando solo dos referencias en los nodos involucrados en la rotación (tiempo O(1)).

Rotación a la derecha

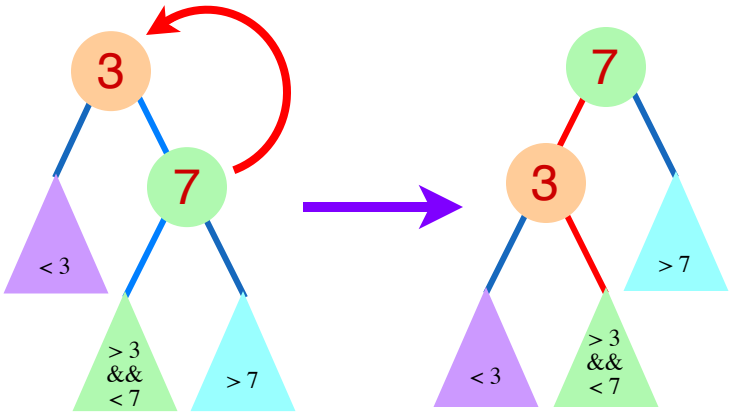
- En una rotación a la derecha, el hijo izquierdo del nodo se convierte en la nueva raíz del subárbol:



- El hijo derecho del subárbol izquierdo original se convierte en el hijo izquierdo del nuevo subárbol derecho.
- La rotación a la derecha mantiene la propiedad de orden del BST.

Rotación a la izquierda

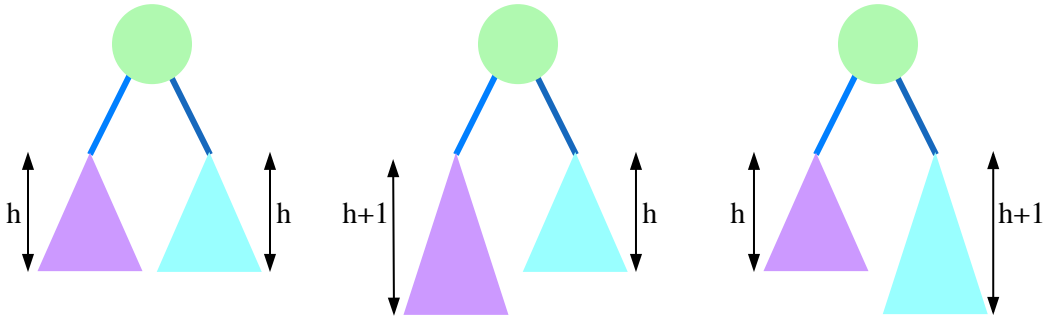
- En una rotación a la izquierda, el hijo derecho del nodo se convierte en la nueva raíz del subárbol:



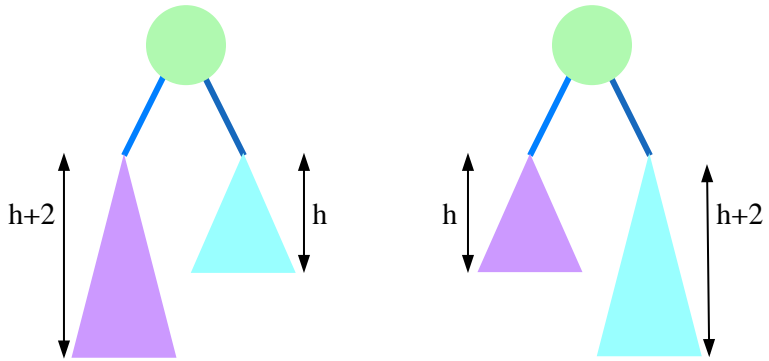
- El hijo izquierdo del subárbol derecho original se convierte en el hijo derecho del nuevo subárbol izquierdo.
- La rotación a la izquierda mantiene la propiedad de orden del BST.

Inserción en un Árbol AVL

- **Proceso de Inserción**: Inserta el nuevo elemento como lo harías en un BST.
- **Propiedad de Equilibrio**: Como partimos de un árbol AVL, cada nodo debe satisfacer la propiedad de equilibrio AVL. Las posibles configuraciones para cada nodo antes de insertar (donde (h) es la altura del subárbol correspondiente) se muestran a continuación:



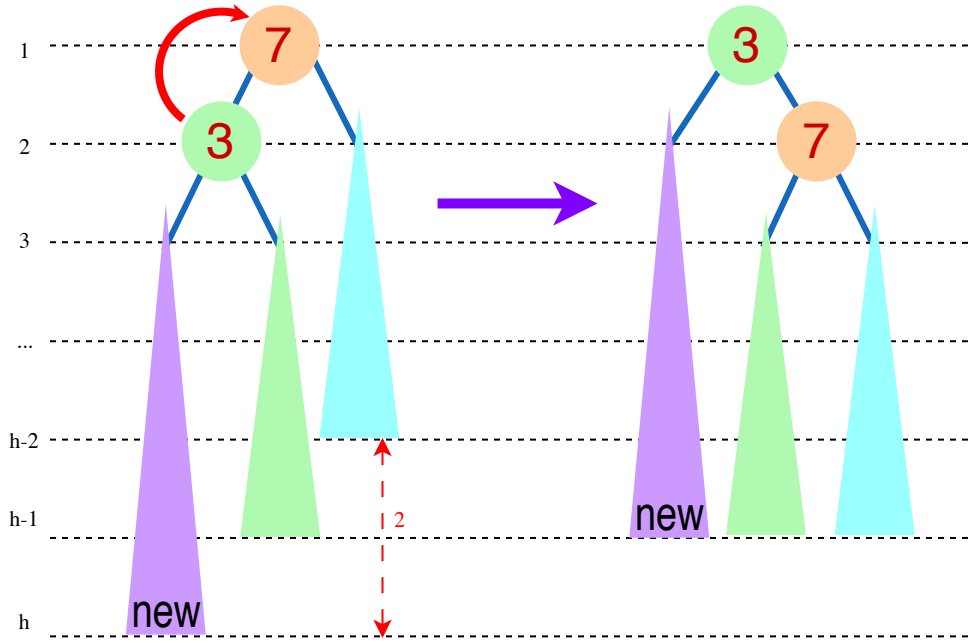
- **Análisis de Casos**:
  - **Caso 1**: Esta configuración permanece equilibrada después de la inserción.
  - **Caso 2**: Esta configuración puede volverse desequilibrada si se inserta en el subárbol izquierdo (ver figura izquierda abajo).
  - **Caso 3**: Esta configuración puede volverse desequilibrada si se inserta en el subárbol derecho (ver figura derecha abajo).



Inserción en un Árbol AVL: caso 2

La inserción ha aumentado la altura del subárbol izquierdo y el nodo se ha vuelto desequilibrado

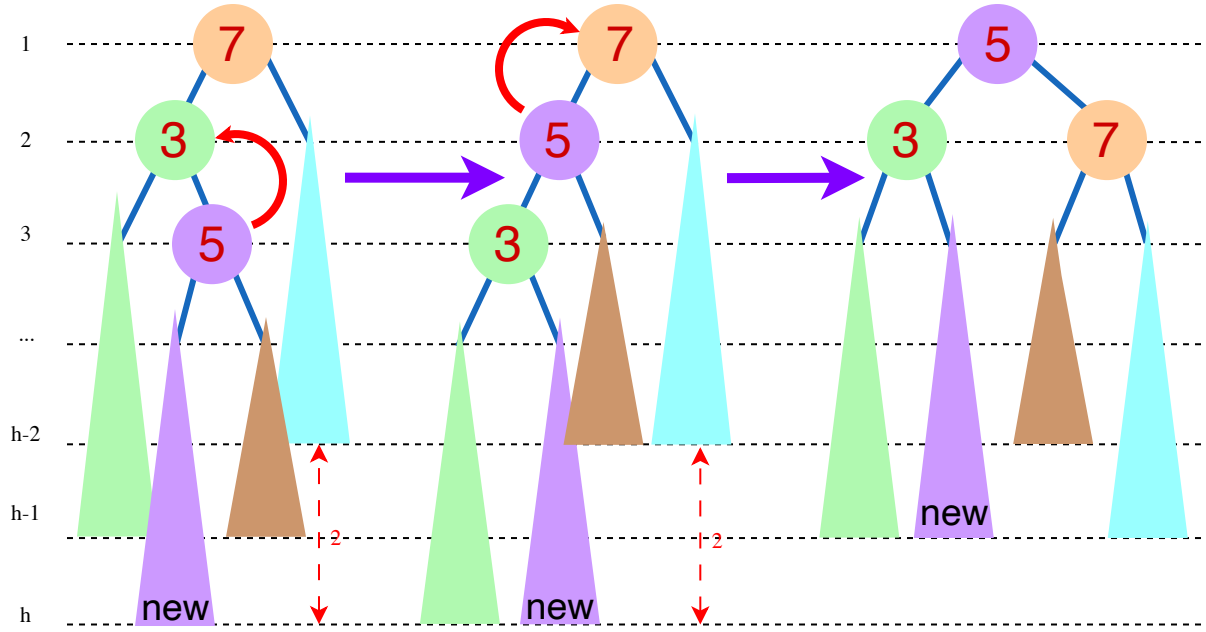
- **Escenario 1. Caso Izquierda-Izquierda**: El nuevo elemento se insertó en el subárbol izquierdo del hijo izquierdo, causando un desequilibrio.
- **Impacto**: Esta inserción aumenta la altura del subárbol izquierdo (destacado en verde), llevando a un árbol desequilibrado.
- **Solución**: Para restaurar el equilibrio, se requiere una rotación simple a la derecha:



Inserción en un Árbol AVL: caso 2

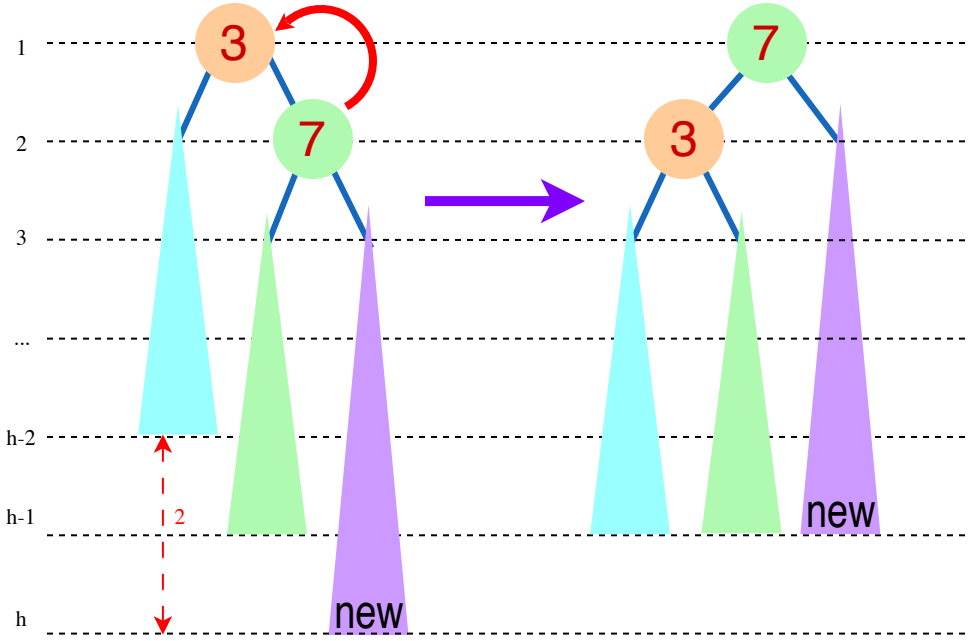
La inserción ha aumentado la altura del subárbol izquierdo y el nodo se ha vuelto desequilibrado

- **Escenario 2. Caso Izquierda-Derecha**: El nuevo elemento se insertó en el subárbol derecho del hijo izquierdo, causando un desequilibrio.
- **Impacto**: Esta inserción aumenta la altura de **uno** de los subárboles (destacado en púrpura), llevando a un árbol desequilibrado.
- **Solución**: Para restaurar el equilibrio, se requiere una doble rotación:
  i. Realizar una rotación a la izquierda en el hijo izquierdo.
  ii. Seguir con una rotación a la derecha en la raíz.



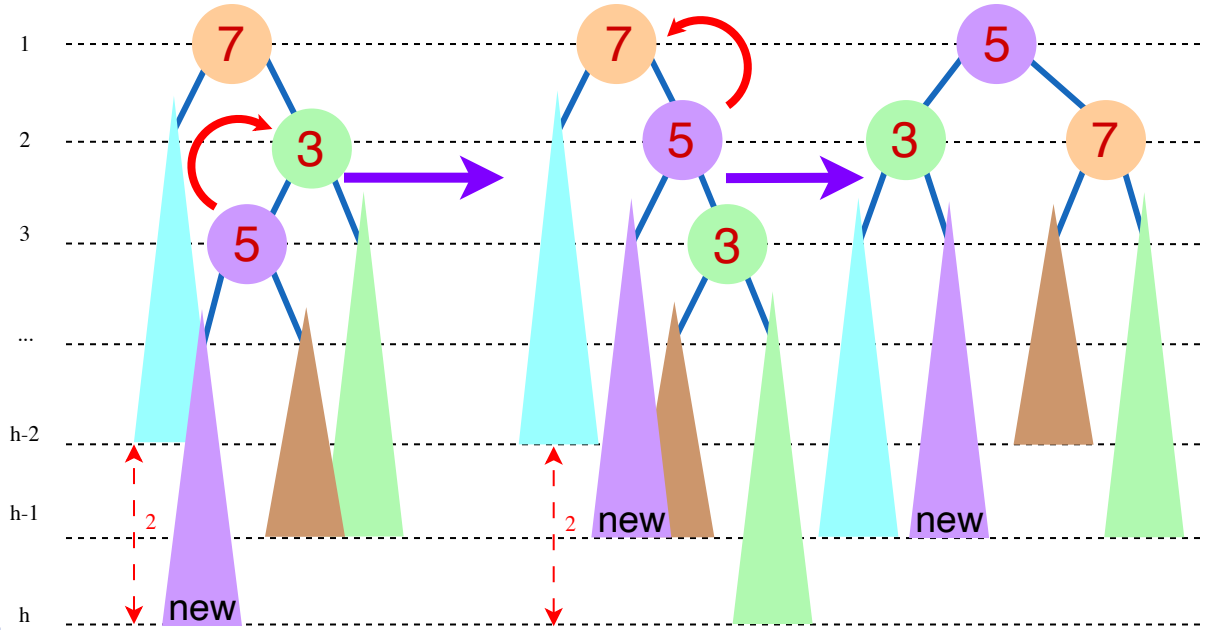
Inserción en un Árbol AVL (VI)

- Consideremos el caso en el que la inserción ha aumentado la altura del subárbol derecho y el nodo se ha vuelto desequilibrado (la diferencia en altura entre los subárboles izquierdo y derecho se ha vuelto mayor que 1).
- **Escenario 3. Caso Derecha-Derecha**: El nuevo elemento se insertó en el subárbol derecho del hijo derecho, causando un desequilibrio.
- **Impacto**: Esta inserción aumenta la altura del subárbol derecho (destacado en verde), llevando a un árbol desequilibrado.
- **Solución**: Para restaurar el equilibrio, se requiere una rotación simple a la izquierda:



Inserción en un Árbol AVL (V)

- **Escenario 4. Caso Derecha-Izquierda**: El nuevo elemento se insertó en el subárbol izquierdo del hijo derecho, causando un desequilibrio.
- **Impacto**: Esta inserción aumenta la altura de **uno** de los subárboles (destacado en púrpura), llevando a un árbol desequilibrado.
- **Solución**: Para restaurar el equilibrio, se requiere una doble rotación:
  i. Realizar una rotación a la derecha en el hijo derecho.
  ii. Seguir con una rotación a la izquierda en la raíz.



La Clase `AVL`

- La clase `AVL<K>` implementa la interfaz `SearchTree<K>` usando un árbol AVL balanceado como representación.
- La clase anidada `Node` representa un nodo en el árbol AVL. Cada nodo almacena:
  - El elemento único (`key`) en el nodo.
  - La `altura` del nodo en el árbol.
  - Referencias a los hijos `izquierdo` y `derecho`.

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

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

        Node(K key) { // Construye un nodo singleton
            this.key = key;
            this.height = 1;
            this.left = null;
            this.right = null;
        }

        static int height(Node<?> node) { // Devuelve la altura del nodo
            return node == null ? 0 : node.height;
        }

        void setHeight() { // Actualiza la altura del nodo receptor
            height = 1 + Math.max(height(left), height(right));
        }
        ...
    }
...
```

La Clase `AVL` (II)

- La clase `Node` proporciona métodos para realizar rotaciones:

```
private static final class Node<K> {
    ...

    Node<K> rightRotated() { // Rota el nodo receptor a la derecha. Devuelve la nueva raíz del árbol rotado
        Node<K> left = this.left;

        this.left = left.right;
        this.setHeight();

        left.right = this;
        left.setHeight();

        return left;
    }

    Node<K> leftRotated() { // Rota el nodo receptor a la izquierda. Devuelve la nueva raíz del árbol rotado
        ...
    }
    ...
}
```

La Clase `AVL` (III)

- La clase `Node` también proporciona métodos para realizar el balanceo del árbol:

```java
private static final class Node<K> {
    ...
    // Devuelve el factor de balance del nodo.
    // Negativo si el nodo está inclinado a la derecha, positivo si está inclinado a la izquierda
    static int balance(Node<?> node) {
        return node == null ? 0 : height(node.left) - height(node.right);
    }

    // Balancea el nodo receptor y establece la nueva altura. Devuelve el nodo ya balanceado
    Node<K> balanced() {
        int balance = balance(this);
        Node<K> balanced;

        if (balance > 1) { // inclinado a la izquierda
            if (balance(left) < 0) { // el hijo izquierdo está inclinado a la derecha. Se necesita doble rotación
                left = left.leftRotated();
            }
            balanced = this.rightRotated();
        } else if (balance < -1) { // inclinado a la derecha
            if (balance(right) > 0) { // el hijo derecho está inclinado a la izquierda. Se necesita doble rotación
                right = right.rightRotated();
            }
            balanced = this.leftRotated();
        } else {
            balanced = this; // el nodo está balanceado. No se necesita rotación
            balanced.setHeight();
        }
        return balanced;
    }
}
```

La Clase `AVL`. Inserción

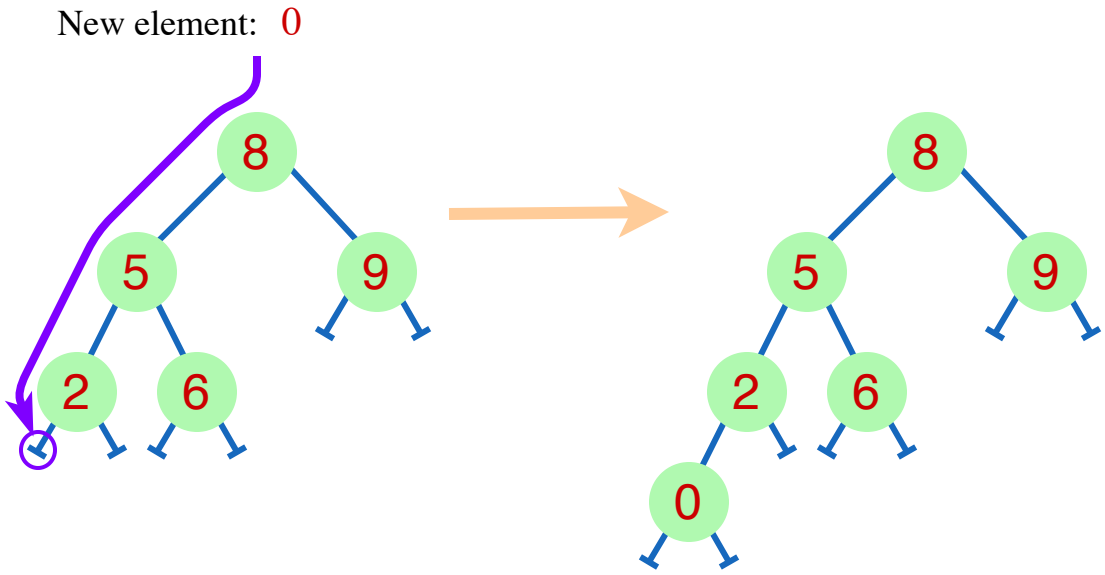
- La inserción en un árbol AVL es similar a la inserción en un BST, pero necesitamos verificar y balancear cada nodo que haya modificado sus hijos.

```
public class AVL<K> implements SearchTree<K> {
    ...
    public void insert(K key) {
        root = insert(root, key);
    }

    private Node<K> insert(Node<K> node, K key) {
        if (node == null) {
            node = new Node<>(key);
            size++;
        } else {
            int cmp = comparator.compare(key, node.key);
            if (cmp < 0) {
                node.left = insert(node.left, key);
                node = node.balanced(); // hijo izquierdo modificado. Verificar y balancear nodo
            } else if (cmp > 0) {
                node.right = insert(node.right, key);
                node = node.balanced(); // hijo derecho modificado. Verificar y balancear nodo
            } else {
                node.key = key;
            }
        }
        return node;
    }
    ...
}
```

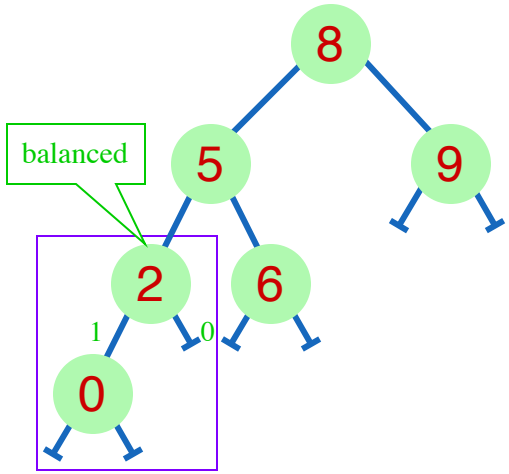
Inserción en un Árbol AVL: Restaurando el Balance con una Rotación Simple

- Comienza con el árbol AVL mostrado en la figura de la izquierda.
- Inserta el nuevo elemento `0`. Los nodos a lo largo del camino desde la raíz hasta el nuevo elemento (8, 5, 2) pueden volverse desequilibrados.
- Verifica y rebalancea cada nodo si es necesario, comenzando desde el nodo más bajo y subiendo hasta la raíz.



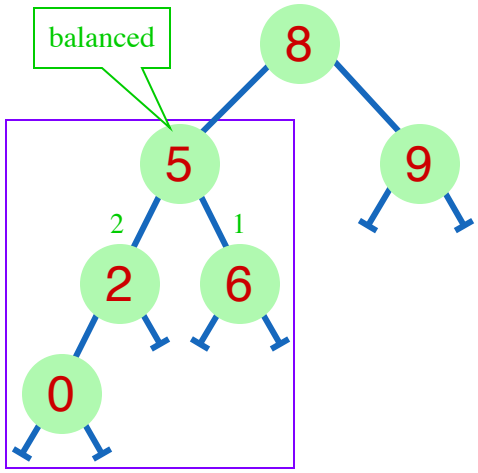
Inserción en un Árbol AVL: Restaurando el Balance con una Rotación Simple (II)

- El nodo 2 permanece balanceado, ya que la diferencia de altura entre sus subárboles izquierdo y derecho es 1. No se necesita rebalanceo para este nodo.



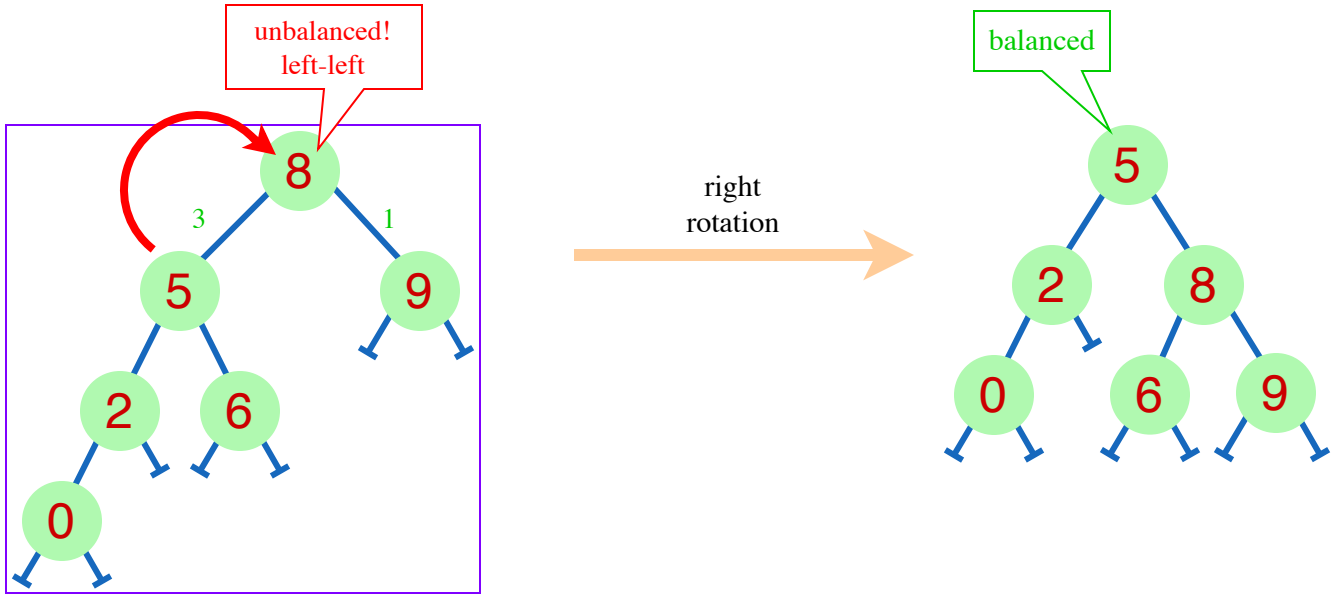
Inserción en un Árbol AVL: Restaurando el Balance con una Rotación Simple (III)

- El nodo 5 permanece balanceado, ya que la diferencia de altura entre sus subárboles izquierdo y derecho es 1. No se necesita rebalanceo para este nodo.



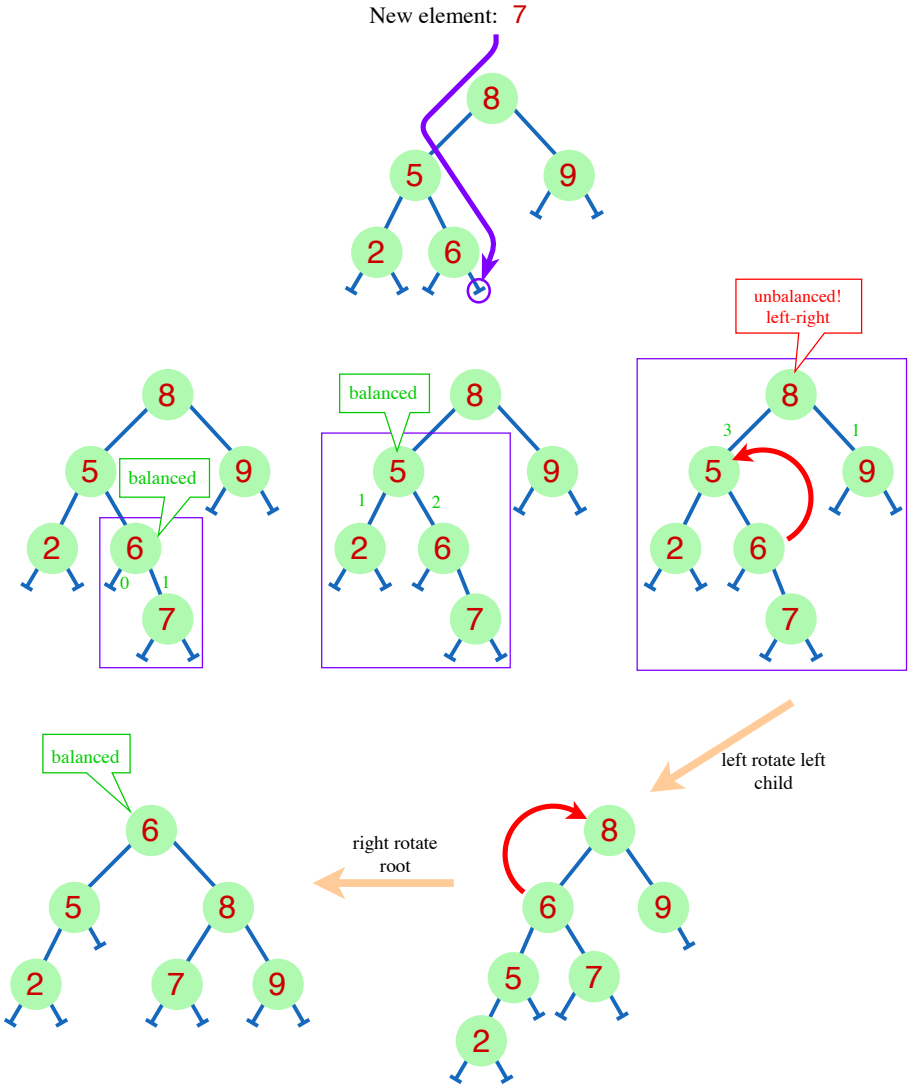
Inserción en un Árbol AVL: Restaurando el Balance con una Rotación Simple (IV)

- El nodo 8 se vuelve desequilibrado con una diferencia de altura de 2 entre sus subárboles izquierdo y derecho.
- Dado que el nodo 8 está inclinado a la izquierda y su hijo izquierdo también está inclinado a la izquierda, este es un caso izquierda-izquierda. Realiza una rotación a la derecha en el nodo 8 para restaurar el balance.



Inserción en un Árbol AVL: Restaurando el Balance con una Doble Rotación

- Después de insertar el elemento 7 en el árbol AVL, los nodos 6 y 5 permanecen balanceados, pero el nodo raíz 8 se vuelve desequilibrado.
- Dado que el nodo raíz está inclinado a la izquierda y su hijo izquierdo está inclinado a la derecha, este es un caso izquierda-derecha. Para restaurar el balance, se realiza una rotación a la izquierda en el hijo izquierdo, seguida de una rotación a la derecha en el nodo raíz para restaurar el balance.



La Clase `AVL`. Búsqueda y Eliminación

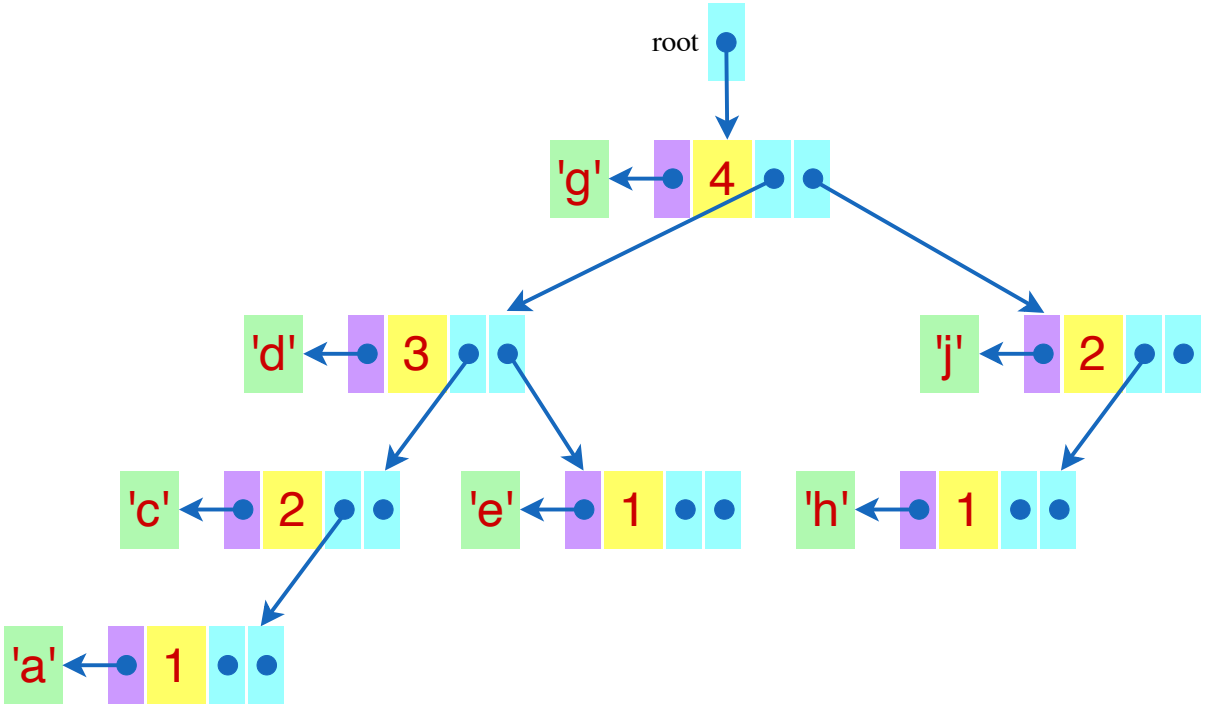
- La búsqueda en un árbol AVL es igual que la búsqueda en un BST, pero ahora la complejidad es O(log n) debido a la altura del árbol.
- La eliminación en un árbol AVL es similar a la eliminación en un BST, pero necesitamos verificar y balancear cada nodo que haya modificado sus hijos.

Complejidad Computacional de las Operaciones de `AVL`

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

La Clase `AVLSet`

- **Implementación**: La clase `AVLSet<T>` implementa la interfaz `SortedSet<T>` usando un árbol AVL para almacenar los elementos dentro del conjunto.
- **Comparador**: Un `Comparator<T>`, proporcionado en la construcción, se utiliza para definir la igualdad y el orden de los elementos.
- **Visualización**: El siguiente diagrama ilustra la estructura de un `AVLSet` que contiene caracteres. Los caracteres se muestran en cajas verdes, mientras que las alturas de los nodos se muestran en cajas amarillas. El árbol está ordenado según el orden natural de los caracteres y permanece balanceado:



Complejidad Computacional de las Operaciones de `AVLSet`

Dado que la altura de un árbol AVL es logarítmica, la mayoría de las operaciones en un `AVLSet` son O(log n):

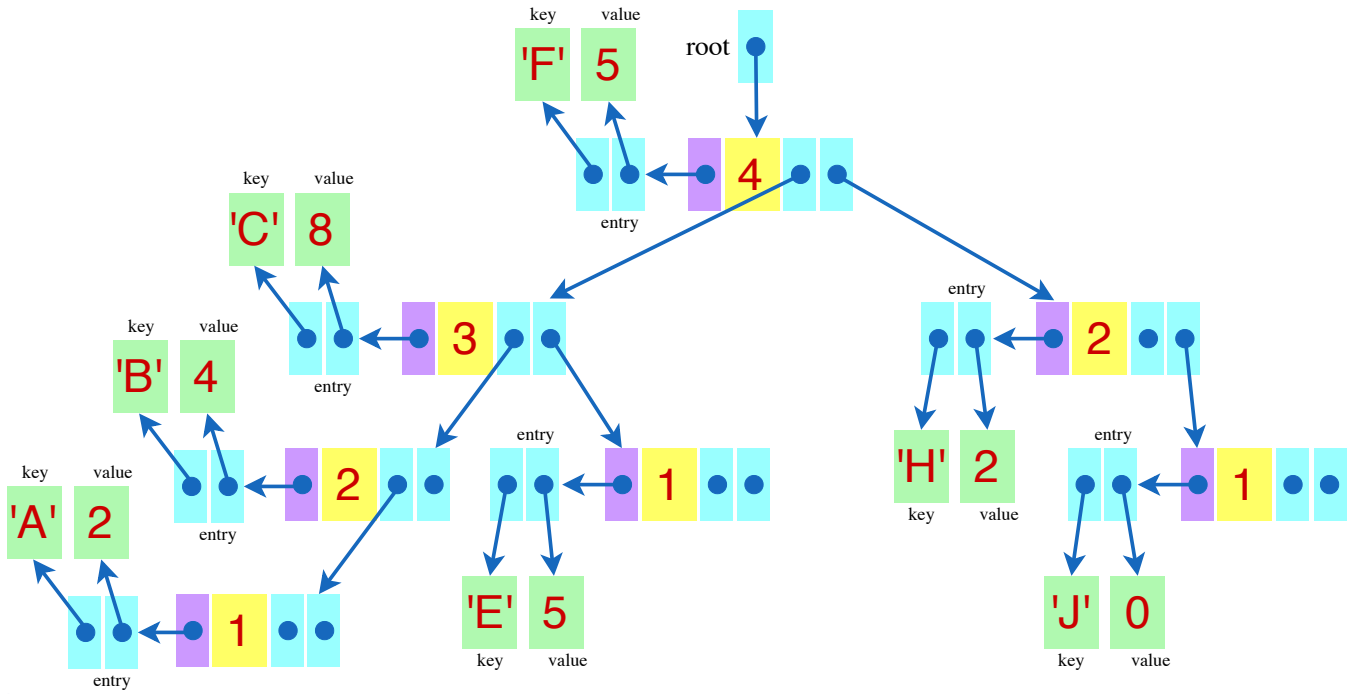
Operación	Coste
`AVLSet.empty`	O(1)
`insert`	O(log n)
`delete`	O(log n)
`contains`	O(log n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(1)

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

- Hemos implementado un conjunto usando un árbol AVL (`AVLSet`) y lo hemos comparado con las implementaciones lineales ordenadas (`SortedLinkedSet` y `SortedArraySet`) y la que usa un BST (`BSTSet`).
- Medimos el tiempo de ejecución 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 aproximadamente 8.15 veces más rápido que `SortedLinkedSet` .
  - `BSTSet` fue aproximadamente 304 veces más rápido que `SortedLinkedSet` .
  - `AVLSet` fue aproximadamente 220 veces más rápido que `SortedLinkedSet` .
- Luego hicimos la misma comparación pero realizando todas las inserciones en orden ascendente (lo que lleva a un árbol degenerado BST).
  - `SortedArraySet` fue aproximadamente 28 veces más rápido que `SortedLinkedSet` .
  - `BSTSet` fue aproximadamente 1.8 veces más lento que `SortedLinkedSet` (árbol degenerado).
  - `AVLSet` fue aproximadamente 113 veces más rápido que `SortedLinkedSet` .

La Clase `AVLDictionary`

- **Implementación**: La clase `AVLDictionary<K, V>` implementa la interfaz `SortedDictionary<K, V>` usando un árbol AVL para almacenar los pares clave-valor (un `Entry`) dentro del diccionario.
- **Comparador**: Un `Comparator<K>`, proporcionado en la construcción, se utiliza para definir la igualdad y el orden de las claves.
- **Visualización**: El siguiente diagrama ilustra la estructura de un `AVLDictionary` que contiene pares clave-valor. Las claves son caracteres y los valores son enteros (ambos mostrados en cajas verdes), mientras que las alturas de los nodos se muestran en cajas amarillas. El árbol está ordenado según el orden natural de las claves y permanece balanceado:



Complejidad Computacional de las Operaciones de `AVLDictionary`

Operación	Coste
`AVLDictionary.empty`	O(1)
`insert`	O(log n)
`delete`	O(log n)
`isDefinedAt`	O(log n)
`valueOf`	O(log n)
`valueOfOrDefault`	O(log n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(1)