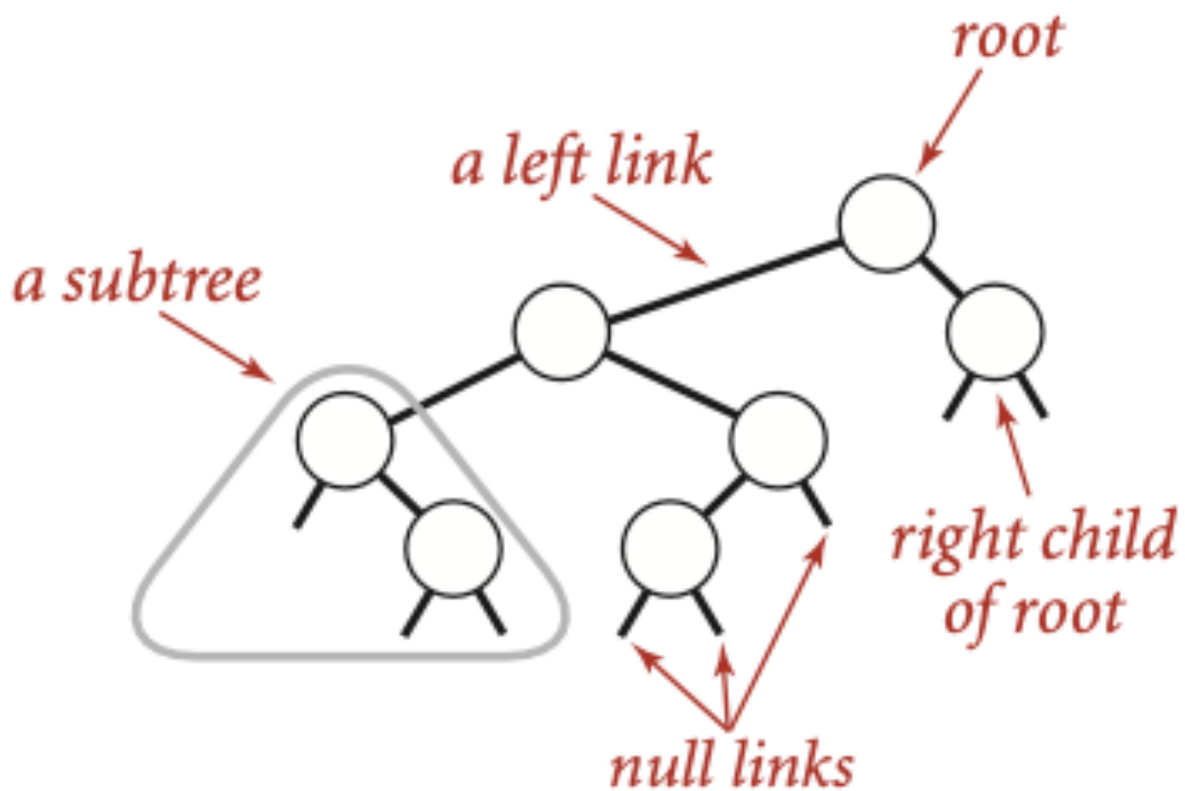
Árboles Binarios de Búsqueda

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II
Año 2025
Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET

Árboles Binarios

Los árboles binarios son aquellos que:

- Cada nodo tiene a lo sumo dos hijos,
- Cada hijo de un nodo es llamado hijo izq. o hijo derecho.



Árboles Binarios el Java

Una implementación muy básica de árboles binarios consta de las siguientes operaciones:

- raiz: Retorna la raíz del árbol,
- hi: retorna el subárbol izquierdo,
- hd: retorna el subárbol derecho,
- preorder: recorrido preorder,
- inorder: recorrido inorder,
- posorder: recorrido posorder.

Esta implementación básica no tiene mucha utilidad en la práctica

Implementación básica en JAVA

Primero definimos una interface con operaciones básicas:

```
// Ejemplo de una interfaz basica para arboles contiene la funcionalidad minima para este tipo
// de estructuras, puede ser enriquecida con mas operaciones
public interface BinaryTreeBasis{

   // Devuelve el elemento de la raiz
   public Object getRoot();

   // Setea la raiz
   public void setRoot(Object item);

   // Dice si el arbol es vacio
   public boolean isEmpty();

   //Remueve todo los nodos del arbol
   public void makeEmpty();

   // recorrido preOrder
   public List preOrder();

   // recorrido postOrder
   public List postOrder();

   // recorrido inOrder
   public List inOrder();
}
```

Implementación con Memoria Dinámica

Una posible implementación con memoria dinámica:

```
public class TreeNode{
   private Object element; // elemento del nodo
   private TreeNode left;  // hijo izquierdo
   private TreeNode right; // hijo derecho

   // constructor del NodoArbol por defecto
   public  TreeNode(){
    element = null;
    left = null;
    right = null;
   }
// implementar el resto..
}
```

Parecida a Node de LinkedList

Los recorridos preorder, inorder, posorder pueden ser implementados acá

```
public class LinkedBinaryTree implements BinaryTreeBasis{
   // raiz del arbol
   private TreeNode root;

   public  LinkedBinaryTree(){
    root = null;
   }
// Implementar el resto…
}
```

Un LinkedBinaryTree tiene como raíz un TreeNode

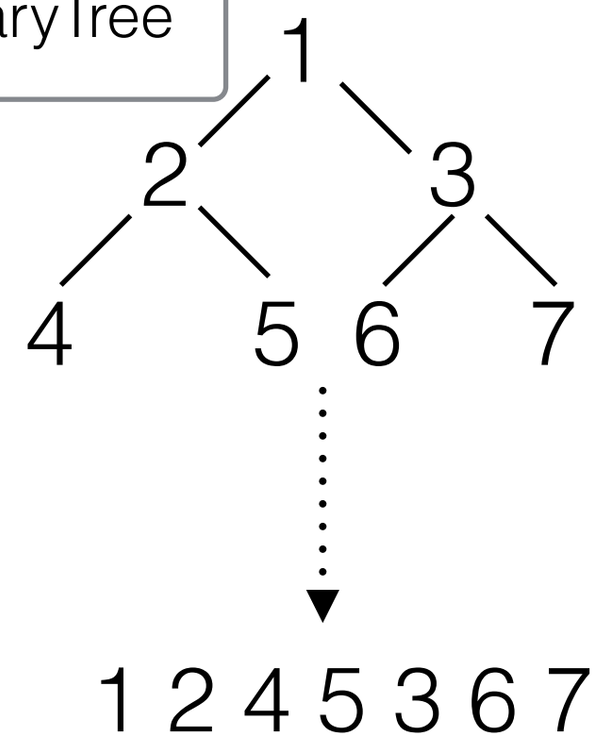
Preorder

Con preorder se recorre primero la raíz, después el hi y finalmente el hd.

```
// Recorrido preorder
   public List preOrder(){
    List result = new LinkedList();
     // se agrega la raiz
    result.add(element);
     // se recorre el hi
    if (left != null){
        result.addAll(left.preOrder());
    }
     // se recorre el hd
    if (right != null){
        result.addAll(right.preOrder());
    }
    return result;
  }
```

Implementado en TreeNode, se puede llamar desde LinkedBinaryTree

El Tiempo de ejecución es O(n), para n la cantidad de nodos (recorre todos los nodos)

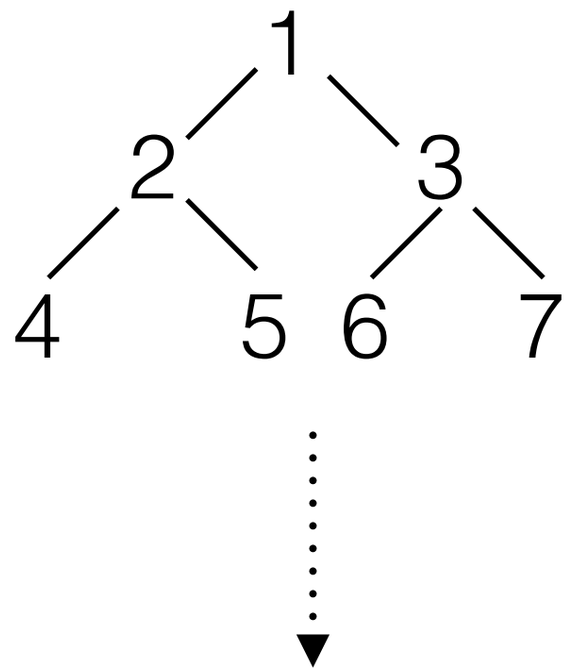


Inorder

Primero el hi, después la raíz y finalmente el hd.

```
 // Recorrido inorder
public List inOrder(){
 List result = new LinkedList();
 // agrega el hi
 if (left != null){
     result.addAll(left.InOrder());
 }
 // agrega la raíz
 list.add(element);
 // agrega el hd
 if (right != null){
     result.addAll(right.inOrder());
 }
 return result;
}
```

El Tiempo de ejecución es O(n)



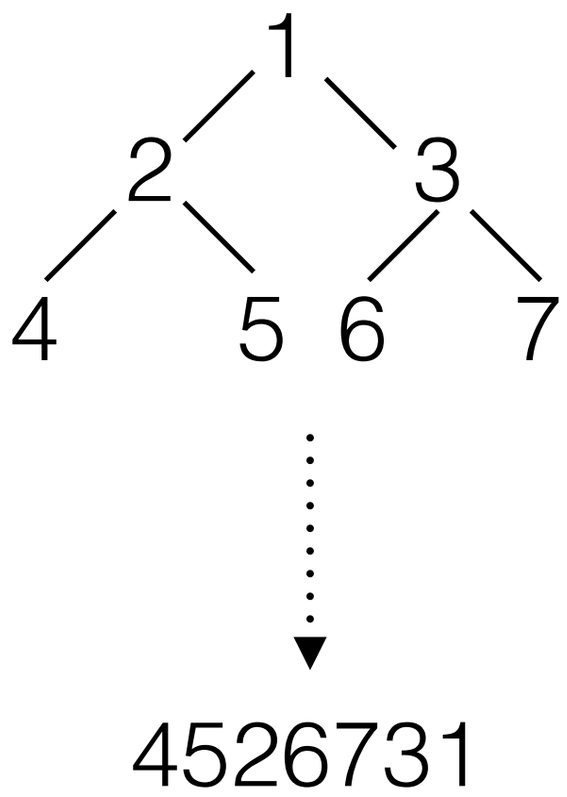
4251637

Posorder

La raíz se recorre al último:

```
// Recorrido posorder
   public List postOrder(){
     List result = new LinkedList();
     // se recorre hi
    if (left != null){
        result.addAll(left.postOrder());
    }
     // se recorre hd
    if (right != null){
        result.addAll(right.postOrder());
    }
     // se agrega la raiz
    result.add(element);
    return result;
  }
```

El Tiempo de ejecución es O(n)



Propiedades

Sea *t* un árbol binario, en donde *size(t)* es su tamaño, *alt(t)* su altura

full(t) dice si el árbol está lleno: tiene todos los nodos en todos los niveles

Propiedad 1: $size(t) \leq 2^{alt(t)} - 1$

Propiedad 2: $log_2(size(t) + 1) \leq alt(t)$

Propiedad 3: $full(t) \Rightarrow size(t) = 2^{alt(t)} - 1$

Propiedad 4: $full(t) \Rightarrow alt(t) = log_2(size(t) + 1)$

Propiedades

Sea *t* un árbol binario, en donde *size(t)* es su tamaño, *alt(t)* su altura

full(t) dice si el árbol está lleno: tiene todos los nodos en todos los niveles

Propiedad 1: $size(t) \leq 2^{alt(t)} - 1$

Propiedad 2: $log_2(size(t) + 1) \leq alt(t)$

Propiedad 3: $full(t) \Rightarrow size(t) = 2^{alt(t)} - 1$

Propiedad 4: $full(t) \Rightarrow alt(t) = log_2(size(t) + 1)$

Ejercicio: Probar estas propiedades usando inducción

Repaso: TAD Set

- Un conjunto es una colección que almacena cada elemento a lo sumo una vez (no admite repetición), y no mantiene un orden específico para los elementos
  - Matemáticamente, un conjunto s se denota como $s = \{e_1, e_2, \ldots, e_n\}\ (e_1 \neq e_2 \neq \ldots \neq e_n)$
- Son apropiados cuando tenemos colecciones de elementos sin repeticiones
- Pueden implementarse de manera que las operaciones de agregar elementos, eliminar elementos y consultar sobre la pertenencia de un elemento sean eficientes

Repaso: Algunas operaciones de Sets

boolean	**add**(**E** e) Adds the specified element to this set if it is not already present.
boolean	**contains**(**Object** o) Returns `true` if this set contains the specified element.
Iterator<**E**>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(**Object** o) Removes the specified element from this set if it is present.
int	**size**() Returns the number of elements in this set (its cardinality).

Repaso: TAD Map

- Un map es una colección que almacena pares de claves y valores, donde cada clave deben tener un único valor asociado
  - Matemáticamente, un map m es una función parcial:
    $m = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\} \ \ (k_1 \neq k_2 \neq \ldots \neq k_n)$
- Los maps son útiles cuando queremos asociar claves con valores (ej. agenda telefónica)
- Las operaciones de maps pueden implementarse de manera eficiente: asociar claves con valores, eliminar claves, y obtener el valor asociado a una clave
- Los maps se suelen llamar también diccionarios, tablas de símbolos, etc...

Repaso: Algunas operaciones de Maps

boolean	**containsKey**(**Object** key) Returns true if this map contains a mapping for the specified key.
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.
Set<**K**>	**keySet**() Returns a **Set** view of the keys contained in this map.
V	**put**(**K** key, **V** value) Associates the specified value with the specified key in this map.
V	**remove**(**Object** key) Removes the mapping for the specified key from this map if present.
int	**size**() Returns the number of key-value mappings in this map.

TAD SortedSet

```java
/**
 * SortedSet are unbounded sets of objects of type T.
 * A typical SortedSet is {o1, . . . , on}.
 *
 * SortedSet requires that the key type T implements the
 * Comparable interface.
 *
 * The methods use compareTo to determine equality of elements.
 */
public interface SortedSet<T extends Comparable<? super T>> {
    /**
     * @post Adds 'x' to the elements of 'this'.
     *   More formally, it satisfies: this = old(this) U {e}.
     */
    public boolean add(T x);
    /**
     * @post Removes 'x' from 'this'. Returns true iff 'x'
     *   was removed. More formally, it satisfies:
     *      result = (e in old(this)) && this = old(this) \ {e}.
     */
    public boolean remove(T x);
    /**
     * @post Returns true iff 'x' is in 'this'.
     */
    public boolean contains(T x);
    /**
     * @post Returns the cardinality of 'this'.
     *   More formally, it satisfies: #this.
     */
    public int size();
```

TAD SortedSet

```java
/**
 * SortedSet are unbounded sets
 * A typical SortedSet is {o1, .
 *
 * SortedSet requires that the k
 * Comparable interface.
 *
 * The methods use compareTo to
 */
public interface SortedSet<T extends Comparable<? super T>> {
    /**
     * @post Adds 'x' to the elements of 'this'.
     *   More formally, it satisfies: this = old(this) U {e}.
     */
    public boolean add(T x);
    /**
     * @post Removes 'x' from 'this'. Returns true iff 'x'
     *   was removed. More formally, it satisfies:
     *      result = (e in old(this)) && this = old(this) \ {e}.
     */
    public boolean remove(T x);
    /**
     * @post Returns true iff 'x' is in 'this'.
     */
    public boolean contains(T x);
    /**
     * @post Returns the cardinality of 'this'.
     *   More formally, it satisfies: #this.
     */
    public int size();
```

```java
public class Person {}

public class Student extends Person implements Comparable<Person> {
    @Override public int compareTo(Person that) {
        // ...
    }
}
```

TAD SortedSet

```
/**
 * SortedSet are unbounded sets
 * A typical SortedSet is {o1, .
 *
 * SortedSet requires that the k
 * Comparable interface.
 *
 * The methods use compareTo to
 */
public interface SortedSet<T extends Comp
    /**
     * @post Adds 'x' to the elements of
     *   More formally, it satisfies: thi
     */
    public boolean add(T x);
    /**
     * @post Removes 'x' from 'this'. Ret
     *   was removed. More formally, it s
     *      result = (e in old(this)) &&
     */
    public boolean remove(T x);
    /**
     * @post Returns true iff 'x' is in '
     */
    public boolean contains(T x);
    /**
     * @post Returns the cardinality of '
     *   More formally, it satisfies: #th
     */
    public int size();
```

```
public (

public (
    @Ov(

    }
}
```

```
    /**
     * @pre !isEmpty()
     * @post Returns the smallest element of 'this'.
     */
    public T min();
    /**
     * @pre !isEmpty()
     * @post Returns the largest element of 'this'.
     */
    public T max();
    /**
     * @post Deletes the smallest element of 'this'.
     */
    public void removeMin();
    /**
     * @post Deletes the largest element of 'this'.
     */
    public void removeMax();
    /**
     * @post Returns true if and only if the structure is a
     *   valid set.
     */
    public boolean repOK();
    /**
     * @post Returns a string representation of the set. Implements
     *   the abstraction function. It represents the set by showing
     *   its elements in increasing order "{o1, o2,..., on}".
     */
    public String toString();
```

Implementaciones

Algunas observaciones:

- Sobre estructuras ordenadas la búsqueda se puede hacer eficientemente en O(log n)
- Los elementos deben tener una clave Comparable para que los elementos se puedan ordenar
- En general, si se manejan una gran cantidad de elementos, las operaciones O(n) sobre colecciones son demasiado costosas
  - Veremos como implementar versiones más eficientes de las operaciones

Implementación de Set/Map con arreglos ordenados

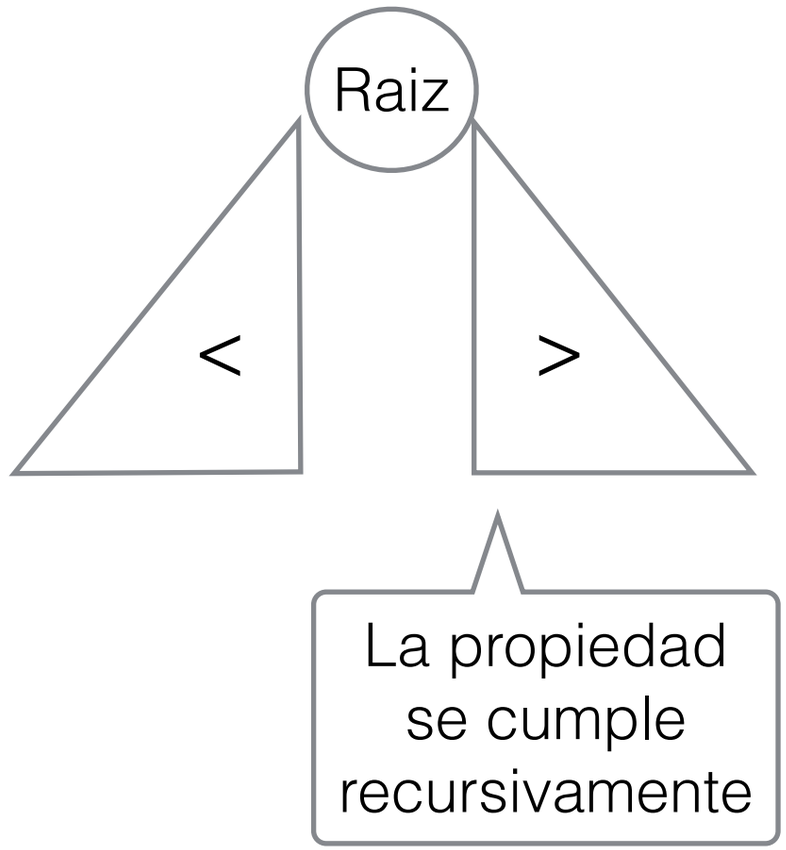
- La búsqueda es O(log n) (búsqueda dicotómica)
- El insertar es O(n), se busca la posición, y luego se tienen que correr n elementos
- El eliminar es O(n) se busca el elemento a eliminar y luego se corren n elementos
- ¡Mejores implementaciones son posibles!

Árboles Binarios de Búsqueda (ABB)

Son arboles binarios cuyas claves están ordenadas

Es decir, cumplen con las siguientes propiedades:

- Los nodos del subárbol izquierdo son menores a la raíz,
- Los nodos del subárbol derecho son mayores a la raíz,
- Los hijos izquierdo y derecho son ABB's



Podemos usarlos para implementar Sets/Maps

Implementación en JAVA

```java
/**
 * BSTSet is an implementation of unbounded sets of
 * objects of type T, based on Binary Search Trees.
 * A typical BSTSet is {o1, . . . , on}.
 *
 * BSTSet requires that the key type T implements the
 * Comparable interface. BSTSet calls the compareTo method
 * to compare two keys in many of the operations.
 *
 * The methods use compareTo to determine equality of elements.
 */
public class BSTSet<T extends Comparable<? super T>> implements SortedSet<T>
{
    private Node root;          // root of BST
    private int size;           // number of nodes in subtree
```

Implementación en JAVA

```java
/**
 * BSTSet is an implementation of unbounded sets of
 * objects of type T, based on Binary Search Trees.
 * A typical BSTSet is {o1, . . . , on}.
 *
 * BSTSet requires that the key type T implements the
 * Comparable interface. BSTSet calls the compareTo method
 * to compare two keys in many of the operations.
 *
 * The methods use compareTo to determine equality of elements.
 */
public class BSTSet<T extends Comparable<? super T>> implements SortedSet<T>
{
    private Node root;          // root of BST
    private int size;           // number of nodes in subtree
```

```java
private class Node {
    private T key;              // sorted by key
    private Node left, right;  // left and right subtrees

    public Node(T key) {
        this.key = key;
    }
```

Implementación en JAVA

```java
/**
 * BSTSet is an implementation of unbounded sets of
 * objects of type T, based on Binary Search Trees.
 * A typical BSTSet is {o1, . . . , on}.
 *
 * BSTSet requires that the key type T implements the
 * Comparable interface. BSTSet calls the compareTo method
 * to compare two keys in many of the operations.
 *
 * The methods use compareTo to determine equality of elements.
 */
public class BSTSet<T extends Comparable<? super T>> implements SortedSet<T>
{
    private Node root;          // root of BST
    private int size;           // number of nodes in subtree
```

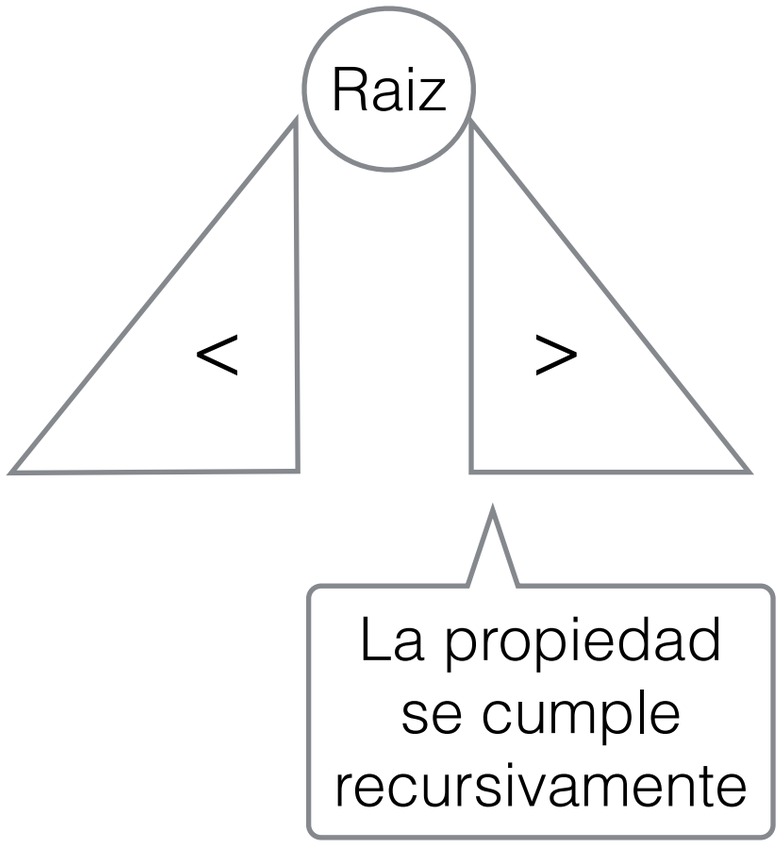
```java
private class Node {
    // ...
    }
```

Ejercicio: Especifique el TAD SortedMap y provea una implementación con ABBs.
Ayuda: Notar que la implementación es muy similar a SortedSet, pero se requiere almacenar pares (key, value) en los nodos.

Árboles Binarios de Búsqueda: repOK

```java
/**
 * @post Returns true if and only if the structure is a
 *   valid Binary Search Tree.
*/
public boolean repOK() {
    return isBST(root, null, null);
}
```

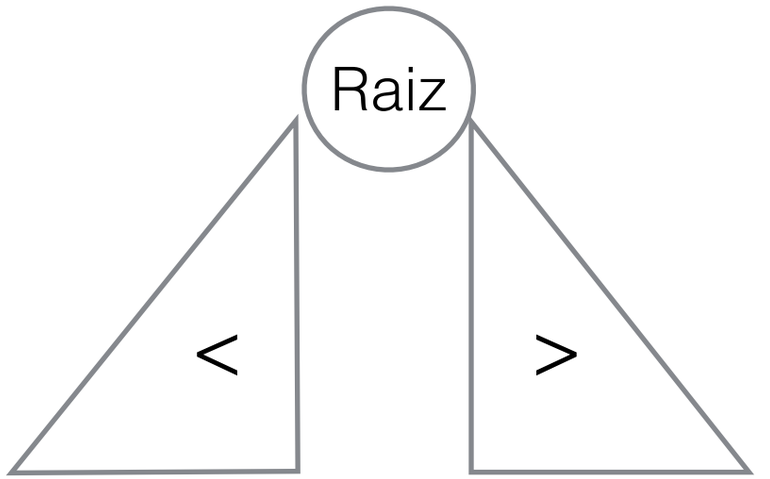
```java
/**
 * @post Returns true iff all the keys in the subtree with
 *   root x are larger than min and smaller than max.
 */
private boolean isBST(Node x, T min, T max) {
    if (x == null)
        return true;

    if (min != null && x.key.compareTo(min) <= 0)
        return false;
    if (max != null && x.key.compareTo(max) >= 0)
        return false;

    return isBST(x.left, min, x.key) &&
           isBST(x.right, x.key, max);
}
```



Árboles Binarios de Búsqueda: repOK

```java
/**
 * @post Returns true if and only if the structure is a
 *   valid Binary Search Tree.
*/
public boolean repOK() {
    return isBST(root, null, null);
}
```

```java
/**
 * @post Returns true iff all the keys in the subtree with
 *   root x are larger than min and smaller than max.
 */
private boolean isBST(Node x, T min, T max) {
    if (x == null)
        return true;

    if (min != null && x.key.compareTo(min) <= 0)
        return false;
    if (
        ret
}
```



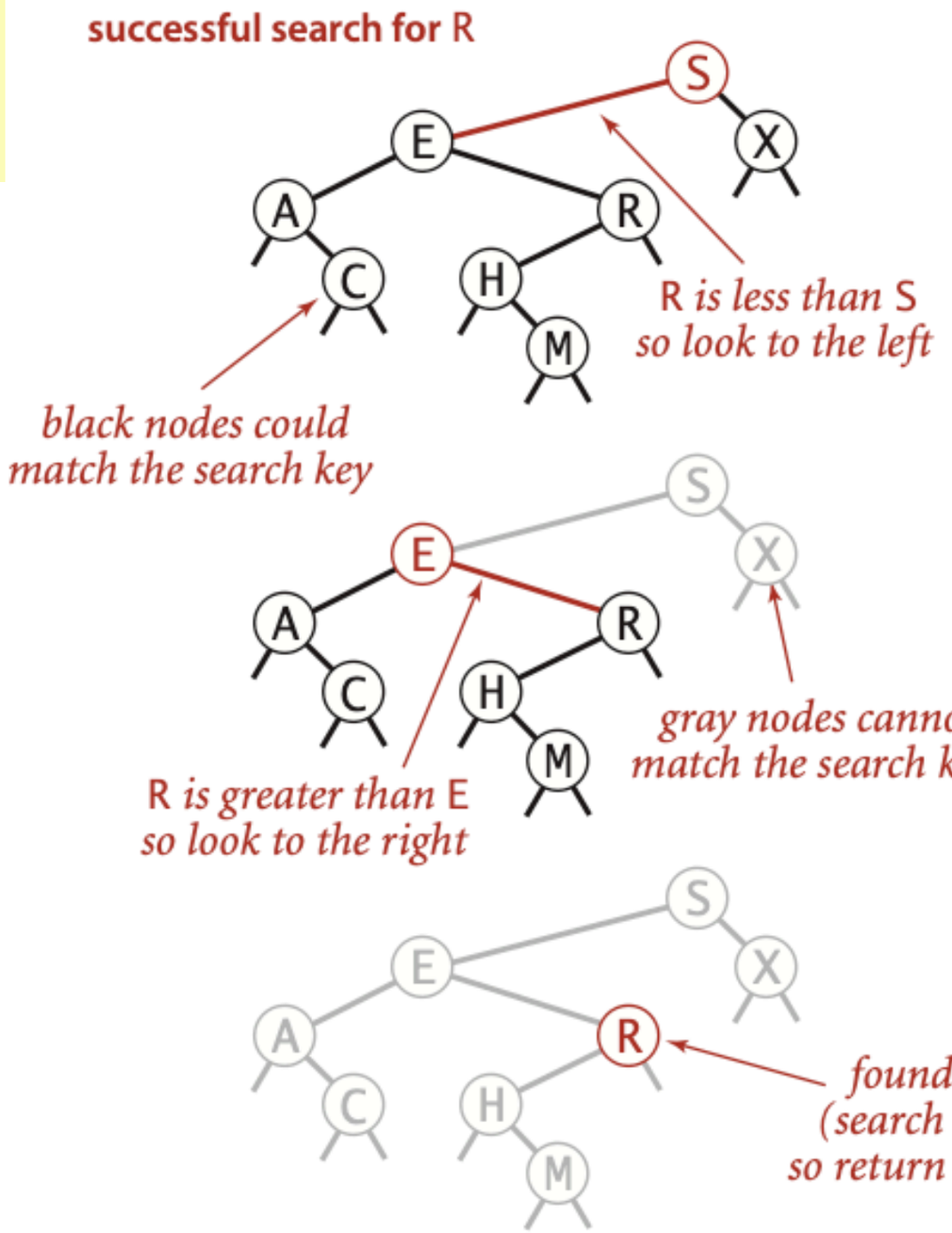
La propiedad se cumple recursivamente

Ejercicio: Verificar las propiedades restantes que deben cumplir los ABBs: que size sea igual a la cantidad de nodos, que no haya ciclos en el árbol, y que los subárboles izquierdos y derechos nunca comparten nodos.

Búsqueda en ABBs

```java
/**
 * @post Returns true iff 'key' is in 'this'.
 */
public boolean contains(T key) {
```

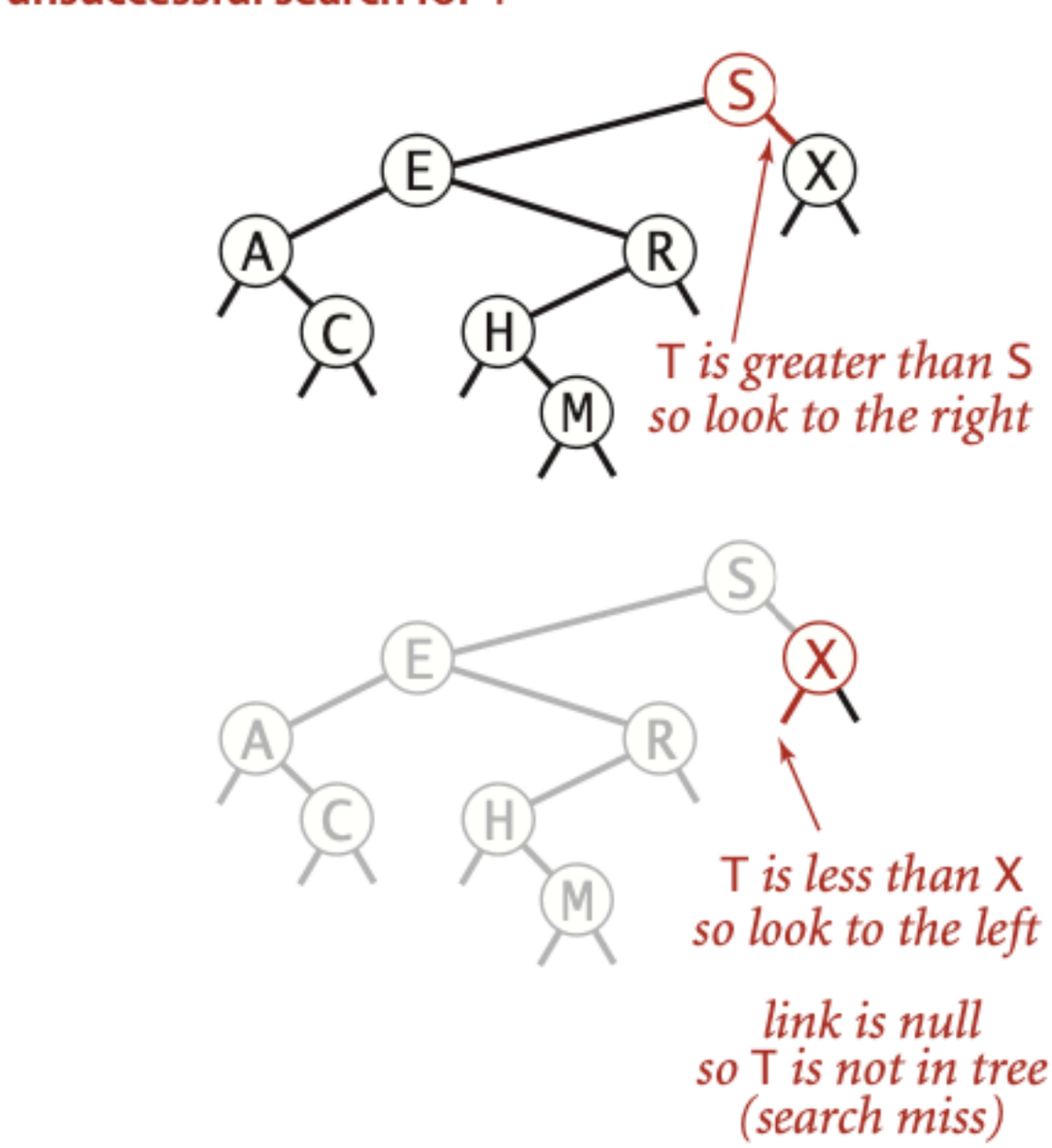
- Idea:
  - Comenzar desde el root
  - Si key es menor que la clave del root, buscar recursivamente en el subárbol izquierdo
  - Si key es mayor que la clave del root, buscar recursivamente en el subárbol derecho
- Casos base:
  - Si llegamos a un nodo que contiene key, retornamos true
  - Si llegamos a null, la clave no está en el ABB, y retornamos false



Búsqueda en ABBs

```java
/**
 * @post Returns true iff 'key' is in 'this'.
 */
public boolean contains(T key) {
```

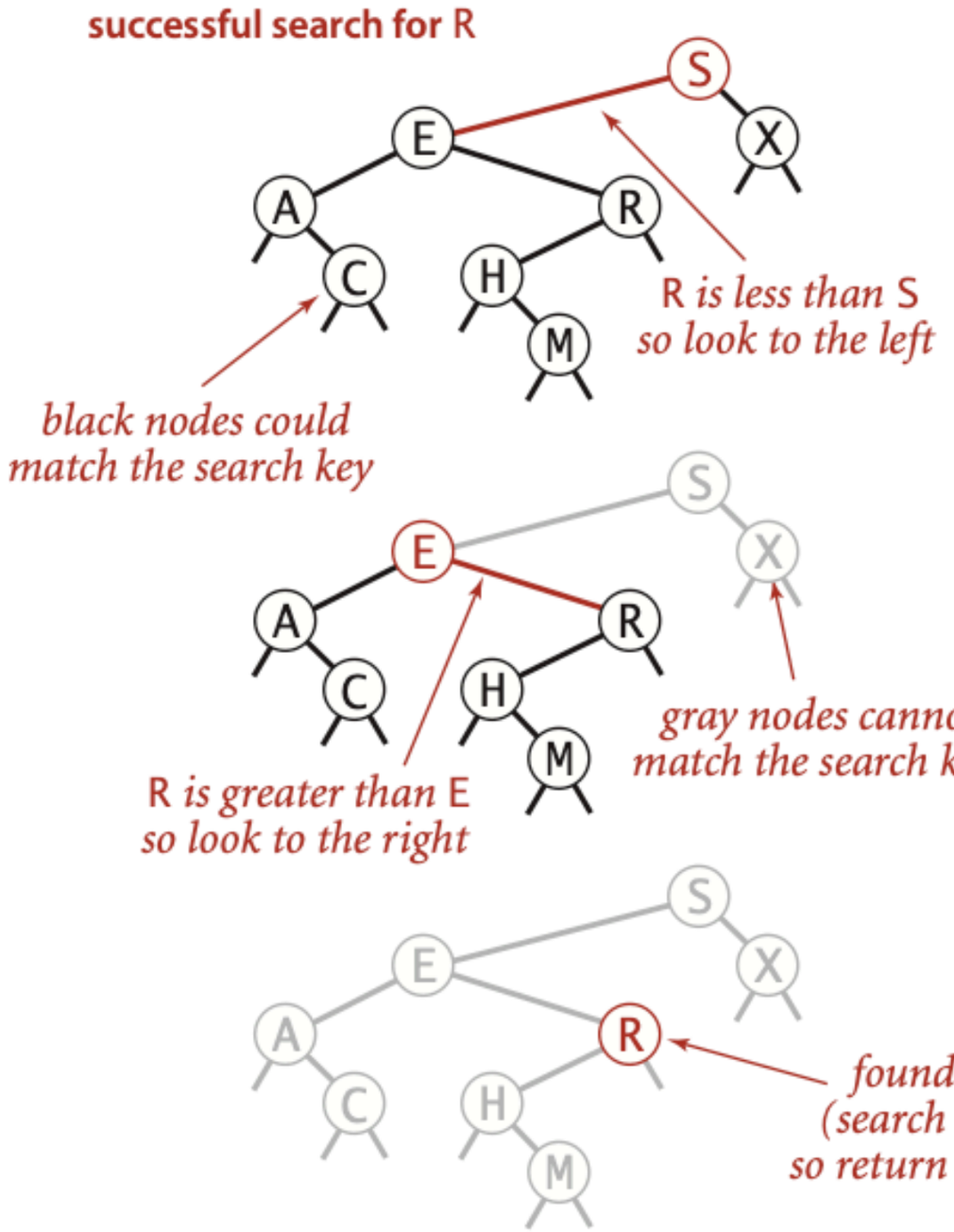
- Idea:
  - Comenzar desde el root
  - Si key es menor que la clave del root, buscar recursivamente en el subárbol izquierdo
  - Si key es mayor que la clave del root, buscar recursivamente en el subárbol derecho
- Casos base:
  - Si llegamos a un nodo que contiene key, retornamos true
  - Si llegamos a null, la clave no está en el ABB, y retornamos false



Búsqueda en ABBs

```java
/**
 * @post Returns true iff 'key' is in 'this'.
 */
public boolean contains(T key) {
    return get(root, key) != null;
}
```

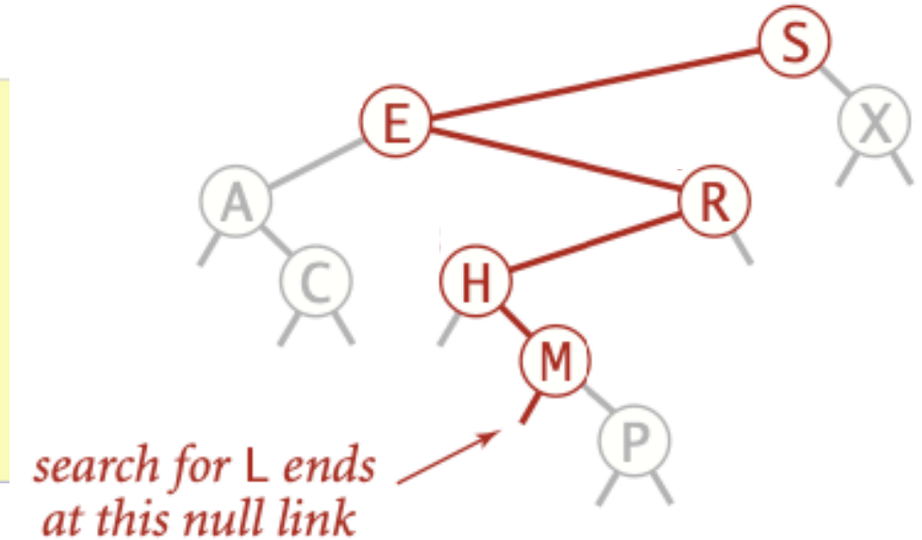
```java
/**
 * @post Returns true iff 'key' can be reached
 *   from node 'x'.
 */
private Node get(Node x, T key) {
    if (x == null)
        return null;

    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        return get(x.left, key);
    else if (cmp > 0)
        return get(x.right, key);
    else
        return x;
}
```

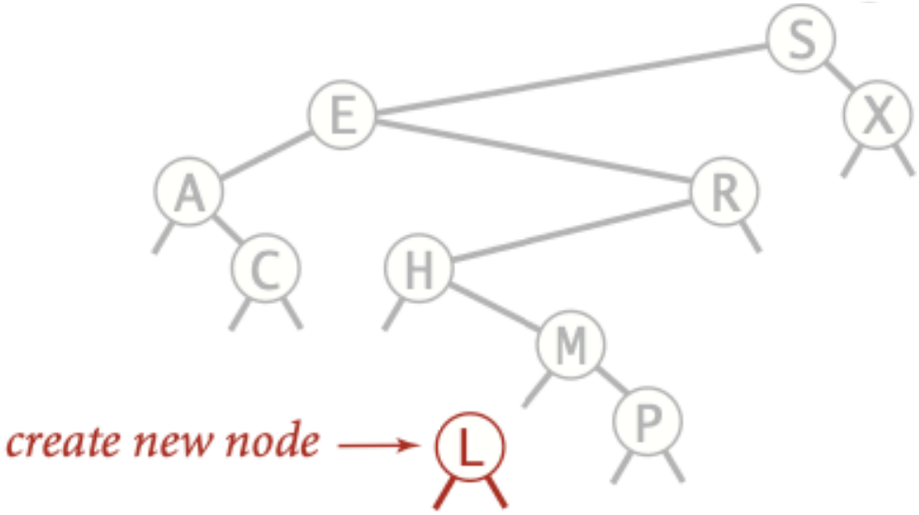


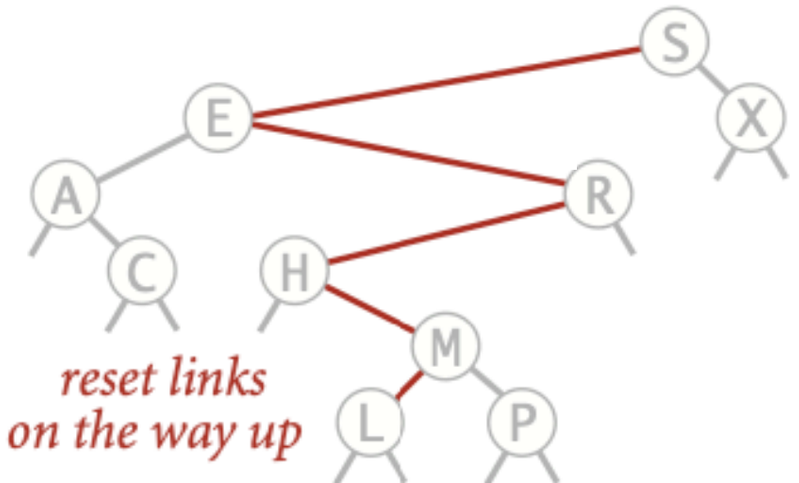
Inserción en ABBs

```
/**
 * @post Adds 'key' to the elements of 'this'. Returns
 *   true iff 'key' was added.
 *   More formally, it satisfies:
 *      result = !(key in old(this)) && this = old(this) U {key}.
 */
public boolean add(T key) {
```

- Hacer una búsqueda recursiva para llegar a la posición donde tenemos que insertar el elemento
- Cuando llegamos a null crear un nodo con la nueva clave y retornarlo
- Actualizar los enlaces cuando las llamadas recursivas retornan:
  - En cada llamada recursiva retornar la raíz del árbol luego de insertar el nuevo nodo
  - Cuando la llamada recursiva retorna, cambiar el subárbol previo por el árbol retornado



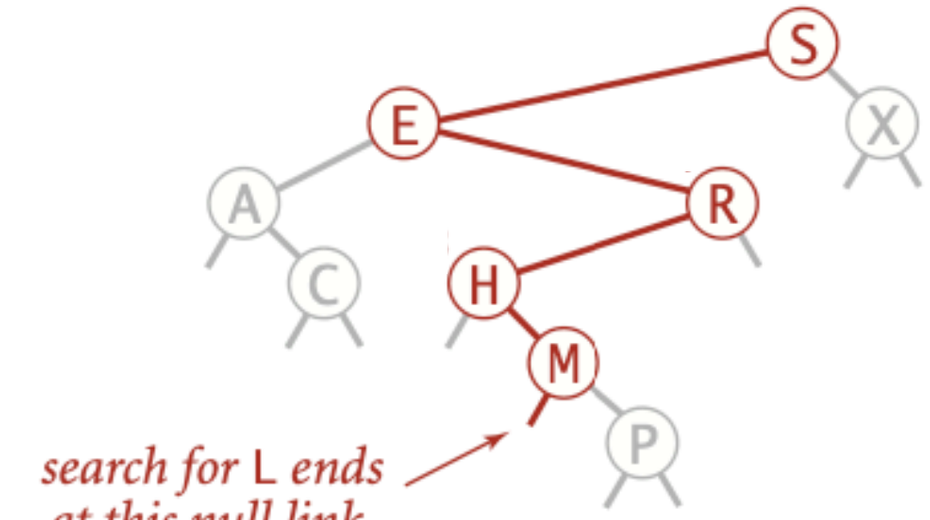


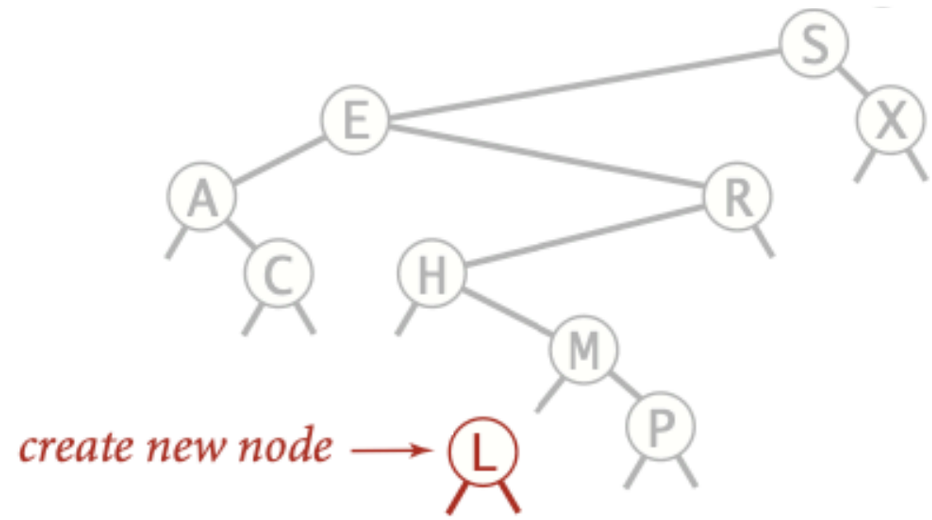


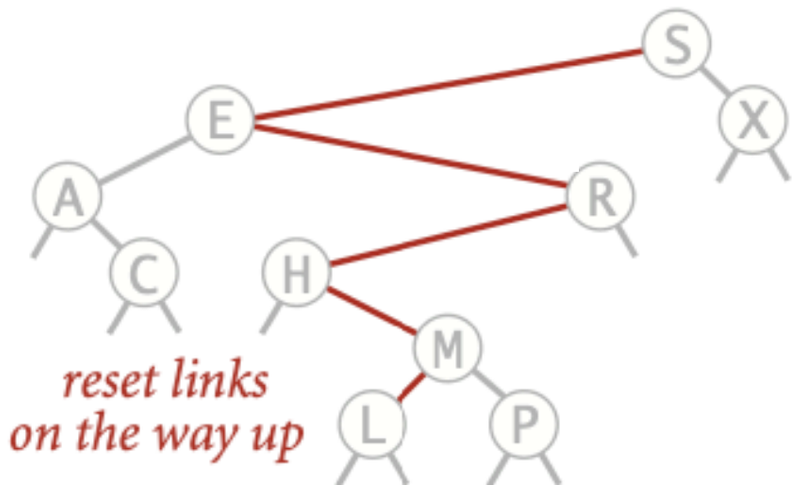
Inserción en ABBs

```
public boolean add(T key) {
    if (contains(key))
        return false;
    root = add(root, key);
    size++;
    return true;
}
```

```
/**
 * @post Inserts 'key' to the tree with root 'x',
 *   and returns the root of the resulting tree.
 */
private Node add(Node x, T key) {
    if (x == null)
        return new Node(key);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0)
        x.left  = add(x.left, key);
    else if (cmp > 0)
        x.right = add(x.right, key);
    else
        // Should never happen!
        assert false;
    return x;
}
```





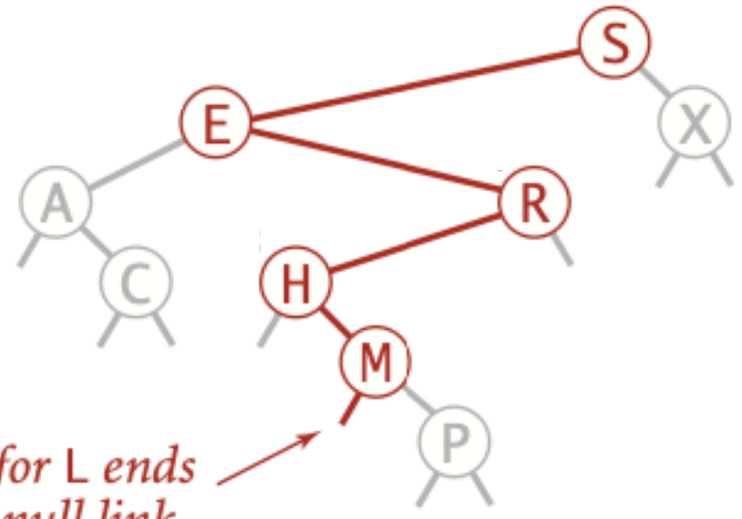


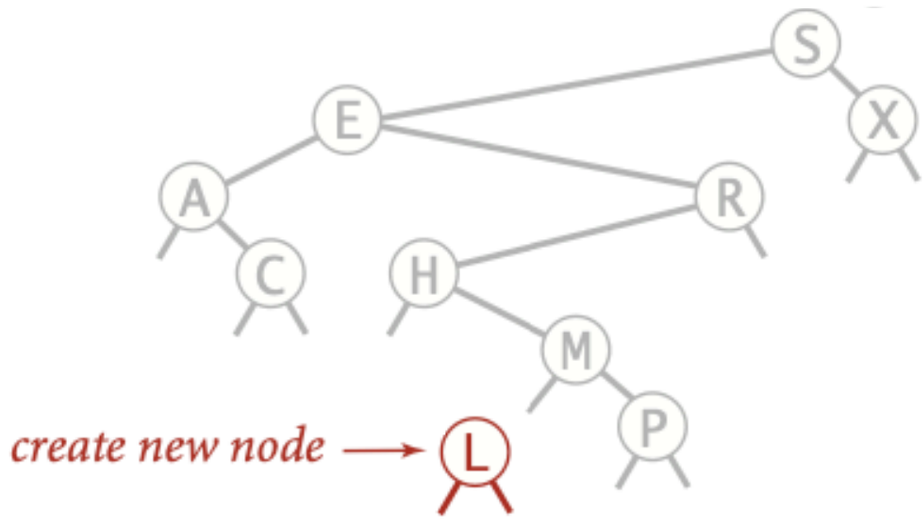
Inserción en ABBs

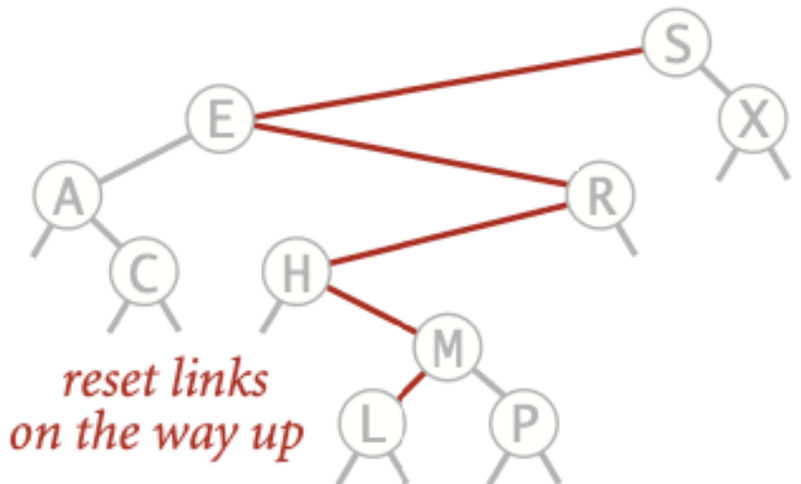
```
public boolean add(T key) {
    if (contains(key))
        return false;
    root = add(root, key);
    size++;
    return true;
}
```

Estamos insertando en un set, no permitimos duplicados

```
/**
 * @post Inserts 'key' to the tree with root 'x',
 *   and returns the root of the resulting tree.
 */
private Node add(Node x, T key) {
    if (x == null)
        return new Node(key);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0)
        x.left  = add(x.left, key);
    else if (cmp > 0)
        x.right = add(x.right, key);
    else
        // Should never happen!
        assert false;
    return x;
}
```







Retornar el mínimo

- El mínimo de un ABB está en el nodo más a la izquierda del árbol
- Caso base: Si el subárbol izquierdo es null, el mínimo es la raíz del árbol
- Caso recursivo: Continuar la búsqueda por el subárbol izquierdo

```java
/**
 * @pre !isEmpty()
 * @post Returns the smallest element of 'this'.
 */
public T min() {
    if (isEmpty())
        throw new NoSuchElementException("Empty tree");
    return min(root).key;
}

/**
 * @post Returns the node that holds the smallest element of the
 *   tree with root x.
 */
private Node min(Node x) {
    if (x.left == null)
        return x;
    else
        return min(x.left);
}
```

Retornar el mínimo

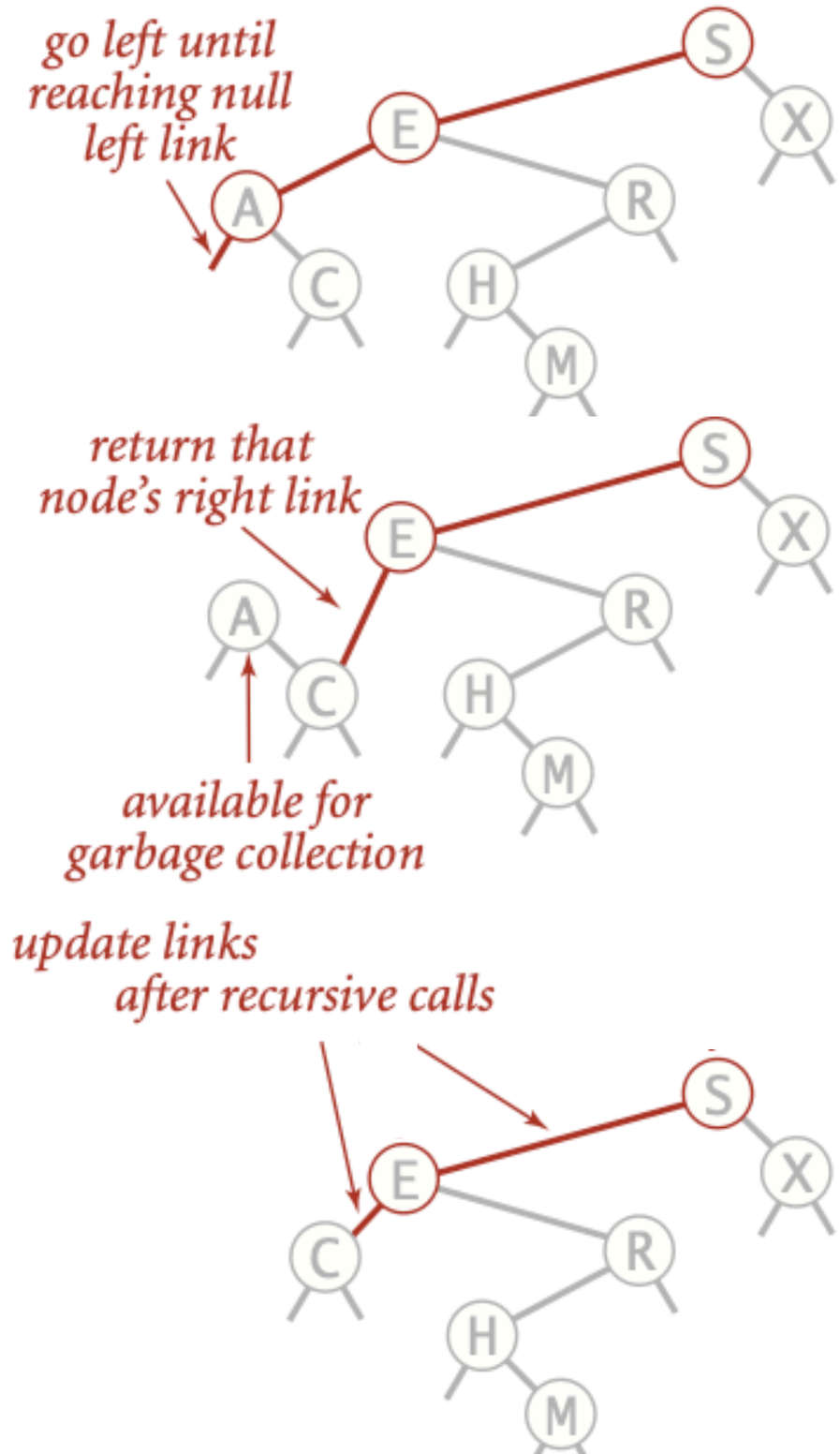
- El mínimo de un ABB está en el nodo más a la izquierda del árbol
- Caso base: Si el subárbol izquierdo es null, el mínimo es la raíz del árbol
- Caso recursivo: Continuar la búsqueda por el subárbol izquierdo

```java
/**
 * @pre !isEmpty()
 * @post Returns the smallest element of 'this'.
 */
public T min() {
    if (isEmpty())
        throw new NoSuchElementException("Empty tree");
    return min(root).key;
}

/**
 * @post Returns the node that holds the smallest element of the
 *   tree with root x.
 */
private Node min(Node x) {
    if (x.left == null)
        return x;
    else
```

Ejercicio: Implementar el método max, que retorna el máximo del árbol. El máximo es el caso dual del min, es decir, está contenido en el nodo más a la derecha en el árbol

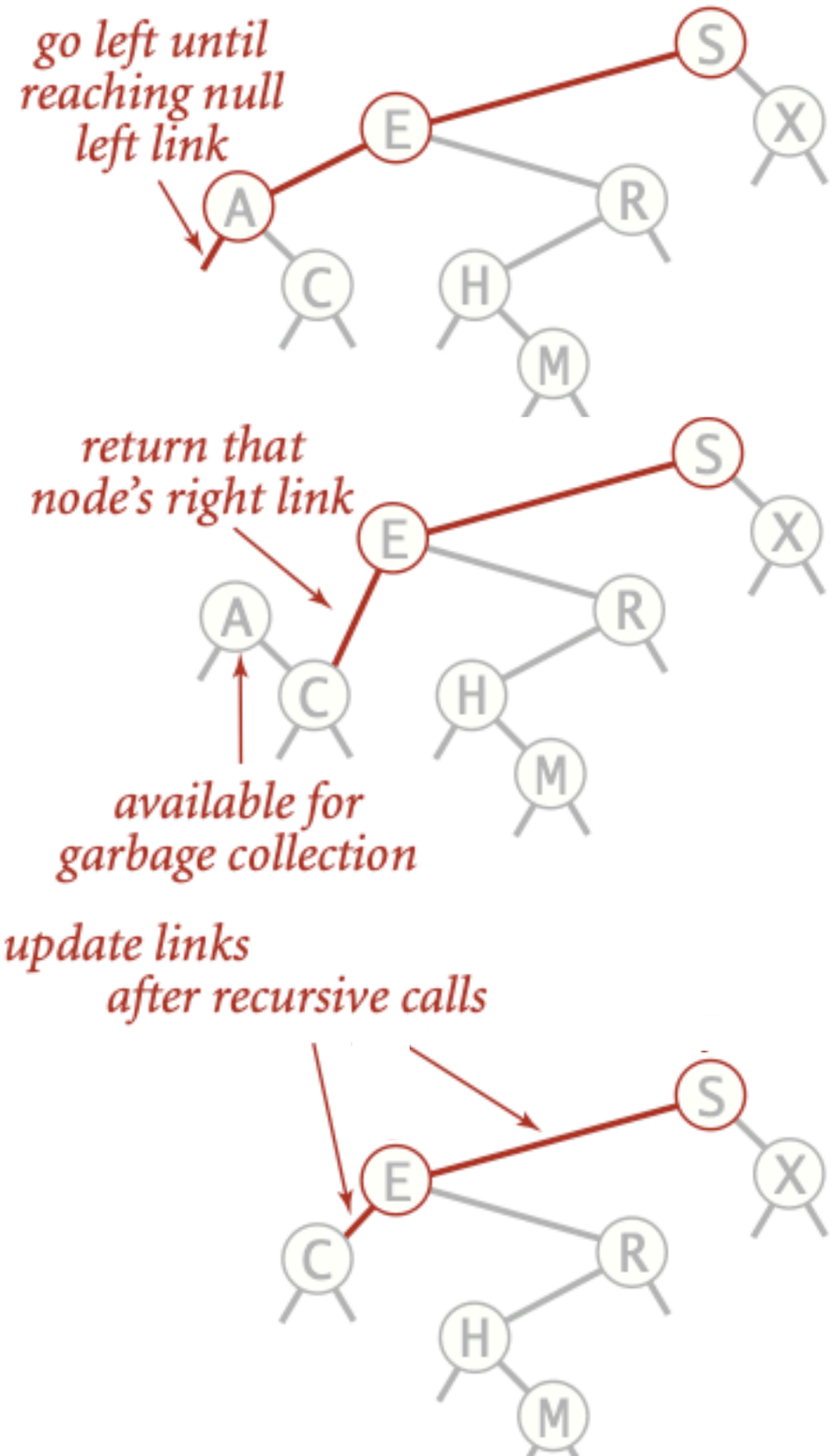
Borrar el mínimo

- Moverse recursivamente hacia la izquierda, hasta encontrar el primer nodo con hijo izquierdo null
- Retornar el hijo derecho de ese nodo
- Actualizar los enlaces cuando las llamadas recursivas retornan:
  - En cada llamada recursiva retornar la raíz del árbol luego de eliminar el nodo con el mínimo
  - Cuando la llamada recursiva retorna, cambiar el subárbol previo por el árbol retornado



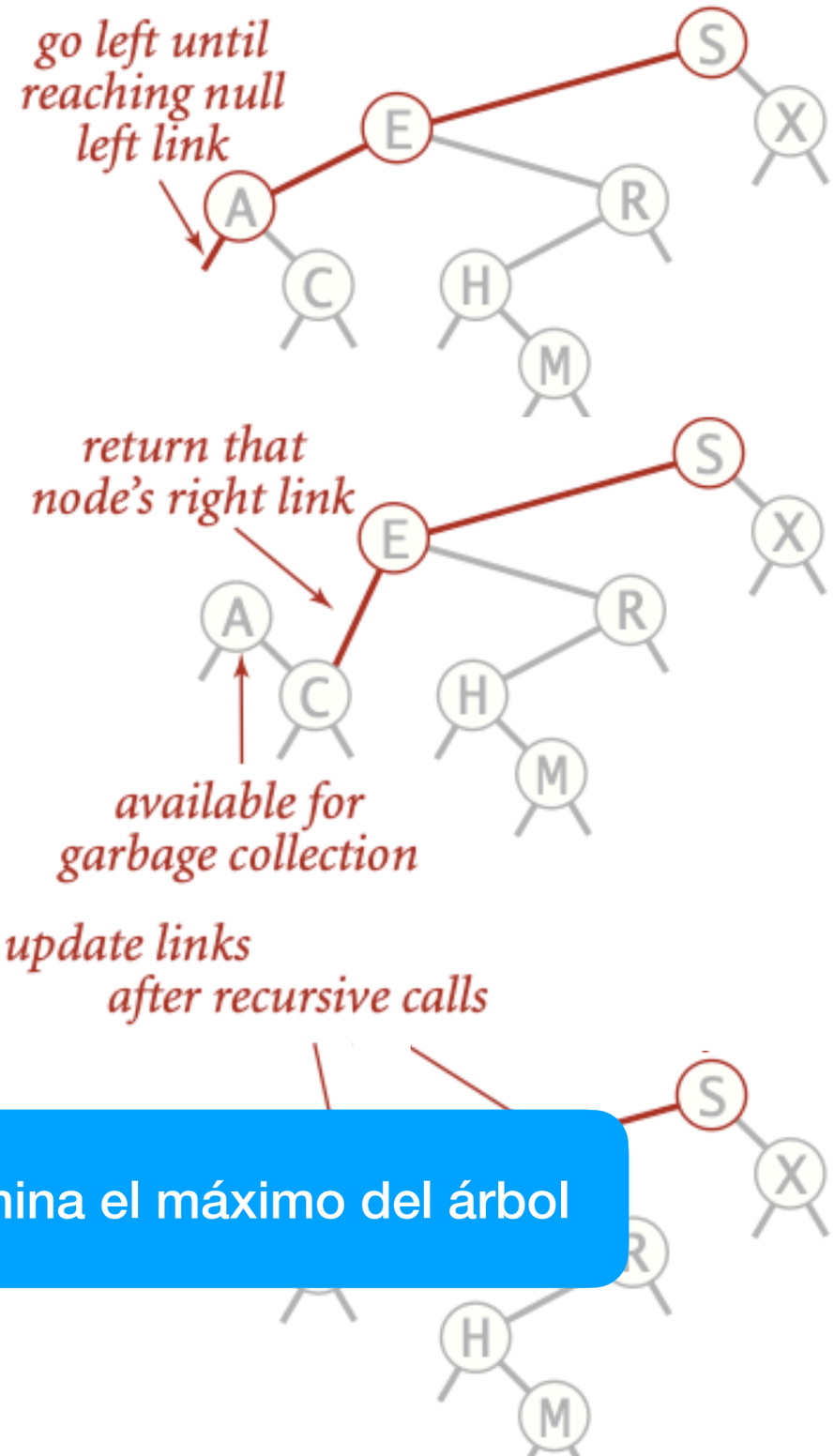
Borrar el mínimo

```java
/**
 * @pre !isEmpty()
 * @post Deletes the smallest element of 'this'.
 */
public void removeMin() {
    if (isEmpty())
        throw new NoSuchElementException("Empty tree");
    root = removeMin(root);
    size--;
}

/**
 * @post Deletes the smallest element of the tree with root 'x',
 *   and returns the root of the resulting tree.
 */
private Node removeMin(Node x) {
    if (x.left == null)
        return x.right;
    x.left = removeMin(x.left);
    return x;
}
```



Borrar el mínimo

```java
/**
 * @pre !isEmpty()
 * @post Deletes the smallest element of 'this'.
 */
public void removeMin() {
    if (isEmpty())
        throw new NoSuchElementException("Empty tree");
    root = removeMin(root);
    size--;
}

/**
 * @post Deletes the smallest element of the tree with root 'x',
 *   and returns the root of the resulting tree.
 */
private Node removeMin(Node x) {
    if (x.left == null)
        return x.right;
    x.le
    retu
}
```



Ejercicio: Implementar el método removeMax, que elimina el máximo del árbol

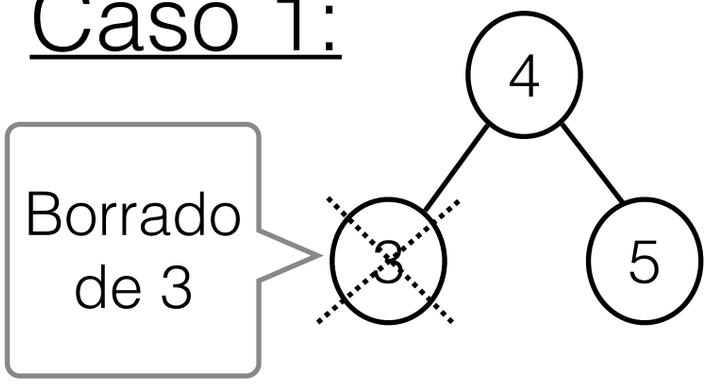
Borrar un elemento de un ABB

Hay que tener cuidado de preservar el invariante de ABB's:

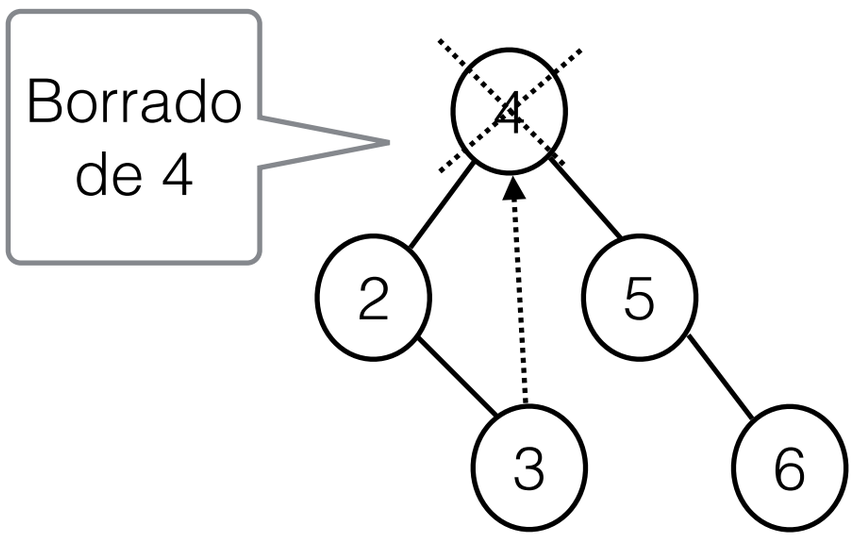
- Se busca el elemento a borrar.
- Si no se encuentra, se termina,
- En otro caso, se elimina el elemento, hay varios casos:
  - El elemento es un hoja, fácil el elemento se borra,
  - El elemento no tiene hi ó hd, se reemplaza el elemento por la raíz del hijo que existe,
  - El elemento tiene ambos hijos, se lo reemplaza por el máximo de la izq. o mínimo de la derecha.

Borrar un elemento de un ABB

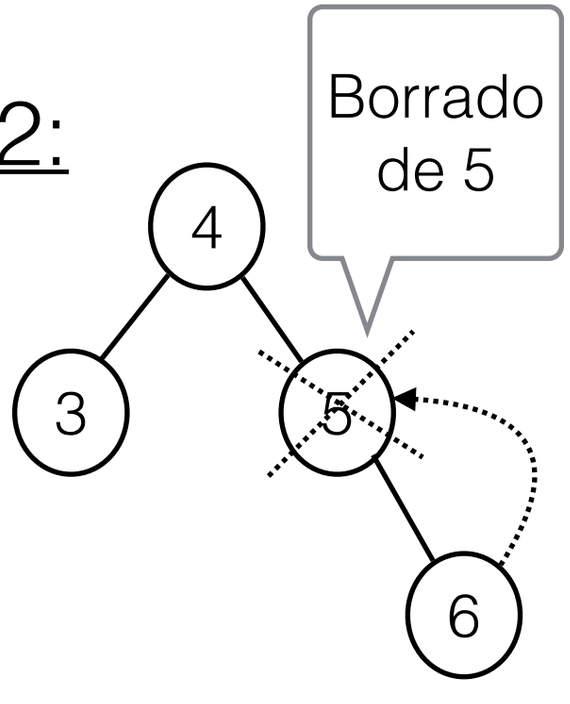
Caso 1:



Caso 3:

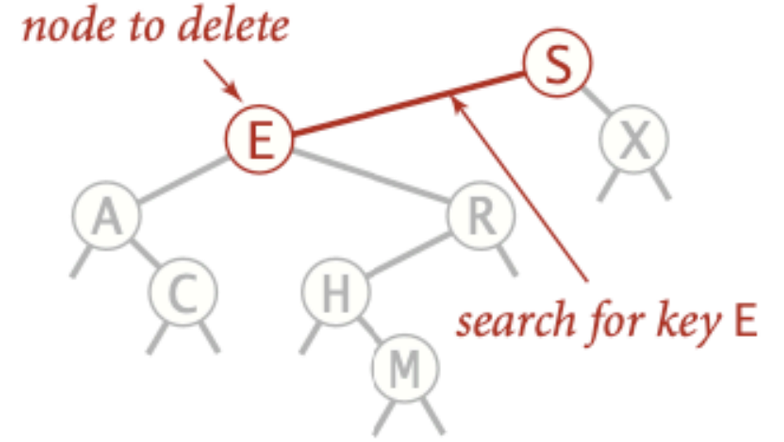


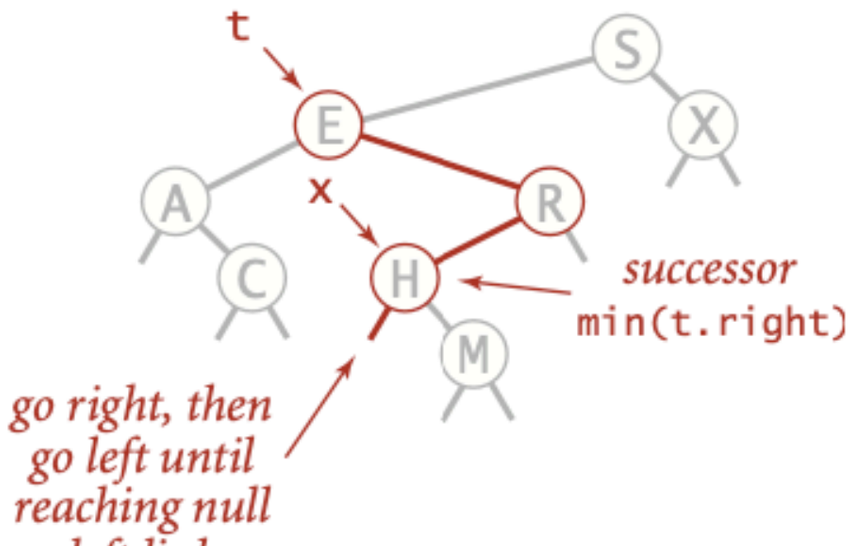
Caso 2:

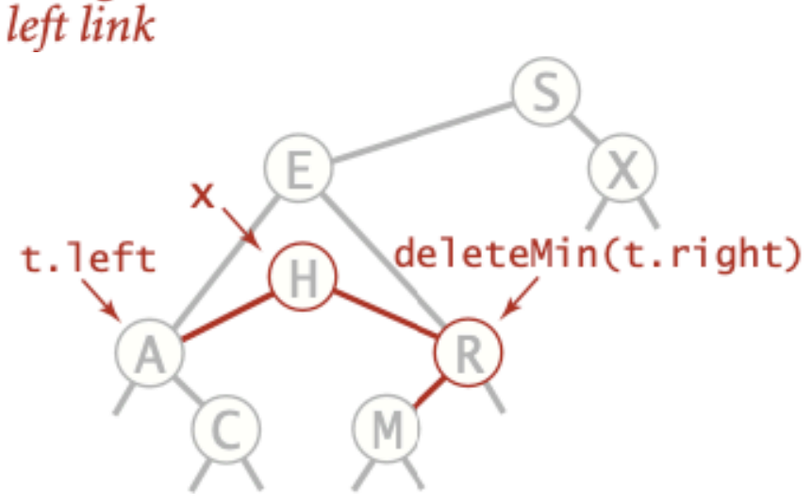


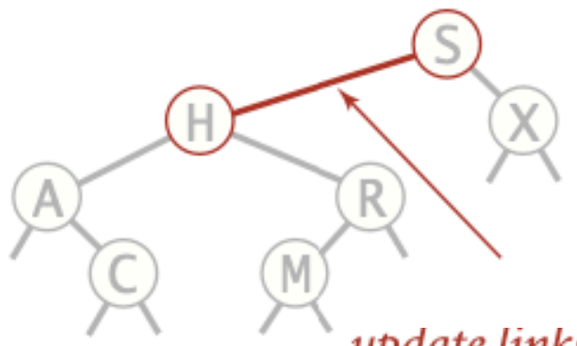
```java
/**
 * @post Removes 'x' from 'this'. Returns
 *   true iff 'x' was removed.
 *   More formally, it satisfies:
 *      result = (e in old(this)) && this = old(this) \ {e}.
 */
public boolean remove(T key) {
    if (!contains(key))
        return false;
    root = remove(root, key);
    size--;
    return true;
}
```

```java
/**
 * @pre key belongs to the tree with root x.
 * @post Removes element key from the tree with root 'x',
 *   and returns the root of the resulting tree.
 */
private Node remove(Node x, T key) {
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left  = remove(x.left,  key);
    else if (cmp > 0)
        x.right = remove(x.right, key);
    else {
        if (x.right == null)
            return x.left;
        if (x.left  == null)
            return x.right;
        Node t = x;
        x = min(t.right);
        x.right = removeMin(t.right);
        x.left = t.left;
    }
    return x;
}
```





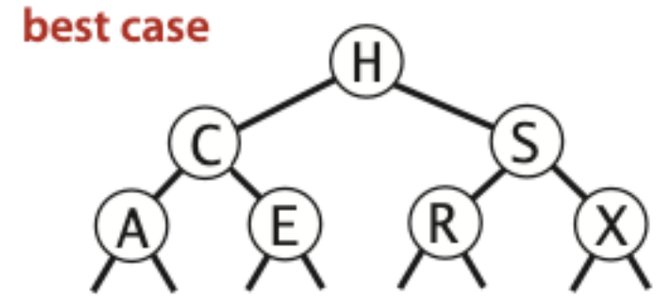


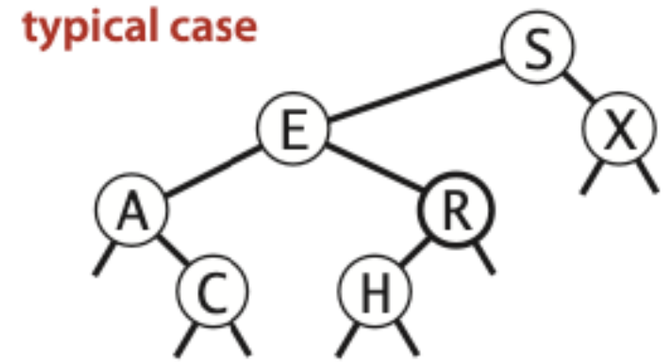


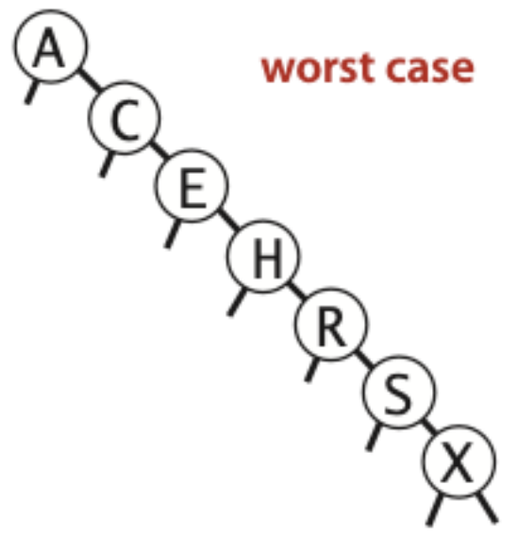
Tiempo de Ejecución en ABB's

- Las operaciones de búsqueda, inserción y eliminación que vimos están acotadas por la altura del árbol (h), es decir, son O(h)
  - Esto se debe a que en el peor caso estos métodos recorren uno o dos veces un camino desde la raíz hasta una hoja del árbol
- En el peor caso h = n, y los métodos son O(n).
- En el mejor caso, el árbol está completo, h = log (n+1), y los métodos son O(log n)
- Si el árbol está balanceado h es O(log n), y los métodos son O(log n)

Si los elementos se insertan en un ABB en orden aleatorio uniforme, en promedio la altura del árbol es O(log n), y las operaciones en el caso promedio son O(log n)







BST possibilities

Función de abstracción

- Aprovechando que los elementos tienen un orden, podemos definir la la función abstracción de manera que retorne una representación canónica del conjunto: el conjunto donde los elementos aparecen en orden

```java
/**
 * @post Returns a list containing all the keys of the tree, where
 *   nodes are visited in inorder.
 */
public List<T> inorder() {
    if (root == null)
        return new LinkedList<T>();

    return root.inorder();
}
```

Función de abstracción

- Aprovechando que los elementos tienen un orden, podemos definir la la función abstracción de manera que retorne una representación canónica del conjunto: el conjunto donde los elementos aparecen en orden

```java
/**
 * @post Returns a list containing al
 *   nodes are visited in inorder.
 */
public List<T> inorder() {
    if (root == null)
        return new LinkedList<T>();

    return root.inorder();
}
```

```java
/**
 * @post Returns a string representation of the set. Implements
 *   the abstraction function. It represents the set by showing
 *   its elements in increasing order "{o1, o2,..., on}".
 */
public String toString() {
    String res = "{";
    boolean first = true;
    List<T> elems = inorder();
    for (T item: elems)
    {
        if (!first)
            res += ", ";
        res += item.toString();
        first = false;
    }
    res += "}";
    return res;
}
```

Función de abstracción

- Aprovechando que los elementos
  función abstracción de manera q
  del conjunto: el conjunto donde l

```java
@Test
public void test2()
{
    BSTSet<Integer> set = new BSTSet<Integer>();
    set.add(0);
    set.add(-2);
    set.add(3);
    set.add(1);
    set.add(-1);
    assertEquals(5, set.size());
    assertEquals("{-2, -1, 0, 1, 3}", set.toString());
    assertTrue(set.repOK());
}
```

```java
/**
 * @post Returns a list containing al
 *   nodes are visited in inorder.
 */
public List<T> inorder() {
    if (root == null)
        return new LinkedList<T>();

    return root.inorder();
}
```

```java
/**
 * @post Re
 *   the ab
 *   its el
 */
public String toString() {
    String res = "{";
    boolean first = true;
    List<T> elems = inorder();
    for (T item: elems)
    {
        if (!first)
            res += ", ";
        res += item.toString();
        first = false;
    }
    res += "}";
    return res;
}
```

Actividades

- Leer el capítulo 12 del libro "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009

Bibliografía

- "Algorithms (4th edition)". R. Sedgewick, K. Wayne. Addison-Wesley. 2016
- "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009
- "Data Structures and Algorithms". A. Aho, J. Hopcroft, J. Ullman. Addison-Wesley. 1983