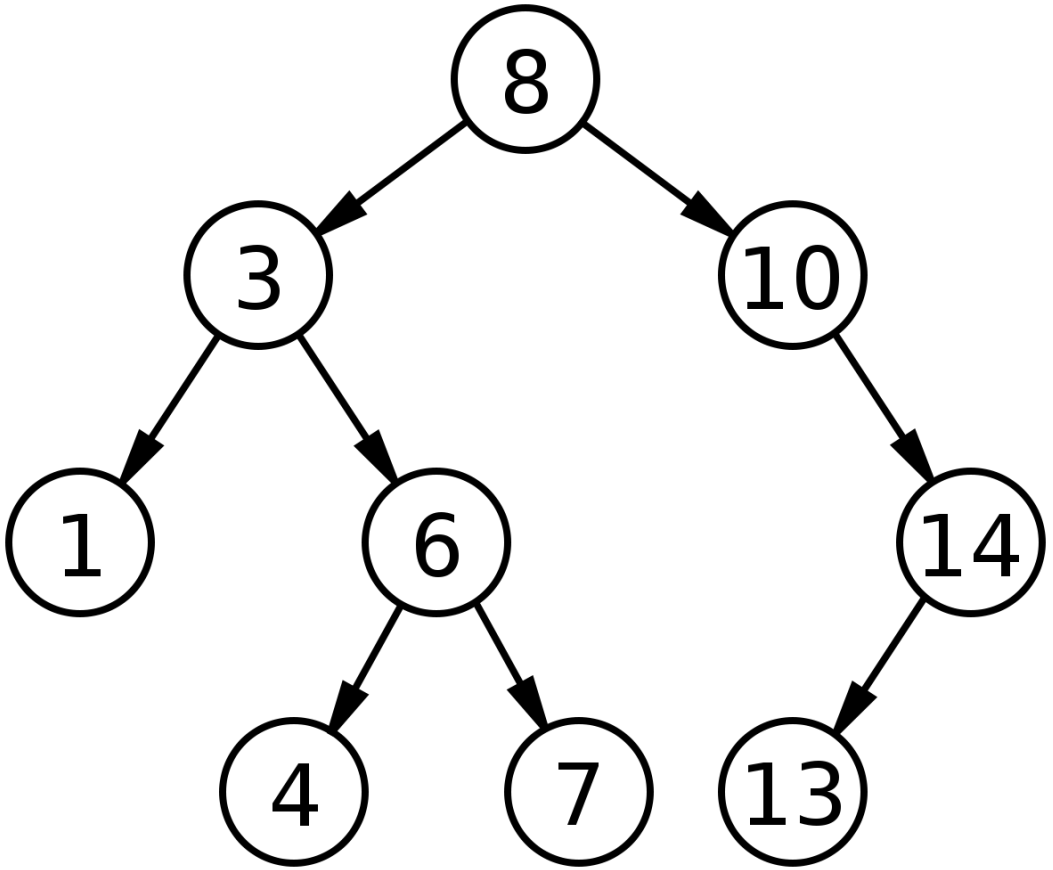
Árboles Binarios de Búsqueda

Carlos Andrés Lozano Garzón
calozanog@uniandes.edu.co

Agenda

- Árboles Binarios
  - Árboles binarios de búsqueda (BST)
  - Búsqueda en BST
  - *Delete* en BST
  - Operaciones ordenadas



Tablas de Símbolos

Definición

Utilizamos el término *Tabla de Símbolos* para describir un mecanismo abstracto donde se almacena información (un valor) que luego podemos buscar y recuperar especificando una *llave*.

La naturaleza de las llaves y los valores depende de la aplicación.

Puede haber una gran cantidad de llaves y una gran cantidad de información, por lo que **implementar una tabla de símbolos eficiente** es un desafío computacional significativo.

Tabla de símbolos

Tipo **abstracto de dato** para guardar objetos en una estructura contenedora

- Tabla de **llaves** y **valores**
- Se añade un objeto a la colección suministrando su llave
  - `void put(Key k, Value v)`
- Se permite buscar por llave para obtener el valor buscado
  - `value get(Key k)`
- Se pueden eliminar los valores de la contenedora
  - `void delete(Key k)`

API Tabla de Símbolos

Abstracción de un arreglo asociativo: Asocia un valor con cada llave.

```
public class ST<Key, Value>
```

	`ST()`	*create an empty symbol table*
`void`	`put(Key key, Value val)`	*put key-value pair into the table* ← `a[key] = val;`
`Value`	`get(Key key)`	*value paired with key* ← `a[key]`
`boolean`	`contains(Key key)`	*is there a value paired with key?*
`void`	`delete(Key key)`	*remove key (and its value) from table*
`boolean`	`isEmpty()`	*is the table empty?*
`int`	`size()`	*number of key-value pairs in the table*
`Iterable<Key>`	`keys()`	*all the keys in the table*

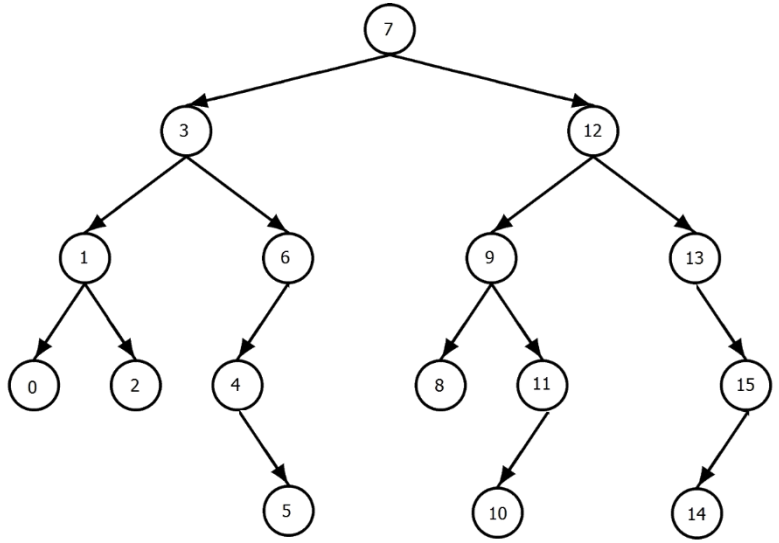
Tablas de símbolos vistas hasta ahora

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`

Tablas de símbolos con árboles binarios

Tablas de símbolos con árboles binarios

Se busca una implementación de tabla de símbolos que combina la flexibilidad de inserción en una lista enlazada con la eficiencia de búsqueda en un arreglo ordenado



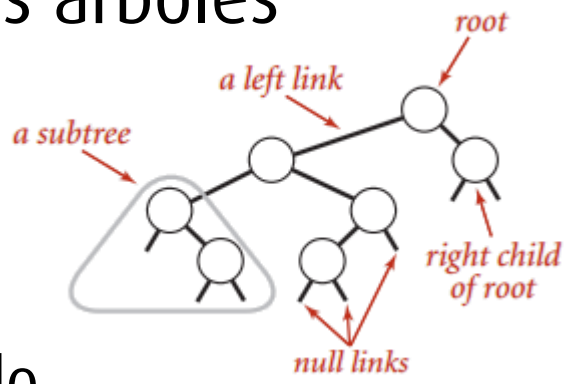
Árboles Binarios de Búsqueda (BST)

Definición

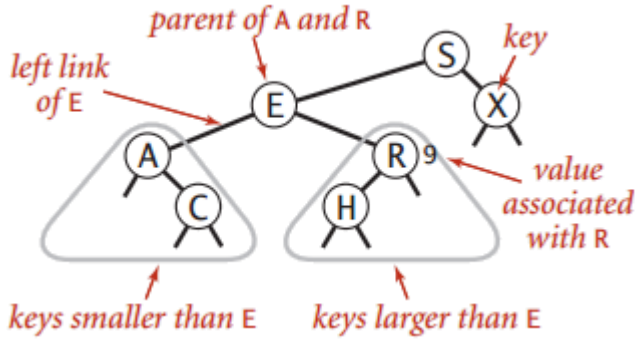
Un árbol binario de búsqueda (BST) es un árbol binario donde cada nodo tiene una clave `Comparable` (y un valor asociado) y satisface la restricción de que la clave en cualquier nodo es más grande que las claves en todos los nodos en el subárbol izquierdo de ese nodo y más pequeña que las claves en todos los nodos en el subárbol derecho de ese nodo.

Árboles binarios de búsqueda (BST)

- Un BST es un árbol binario en **orden simétrico**
- Un árbol binario es un árbol vacío o un nodo con dos árboles disyuntos (izquierdo y derecho)

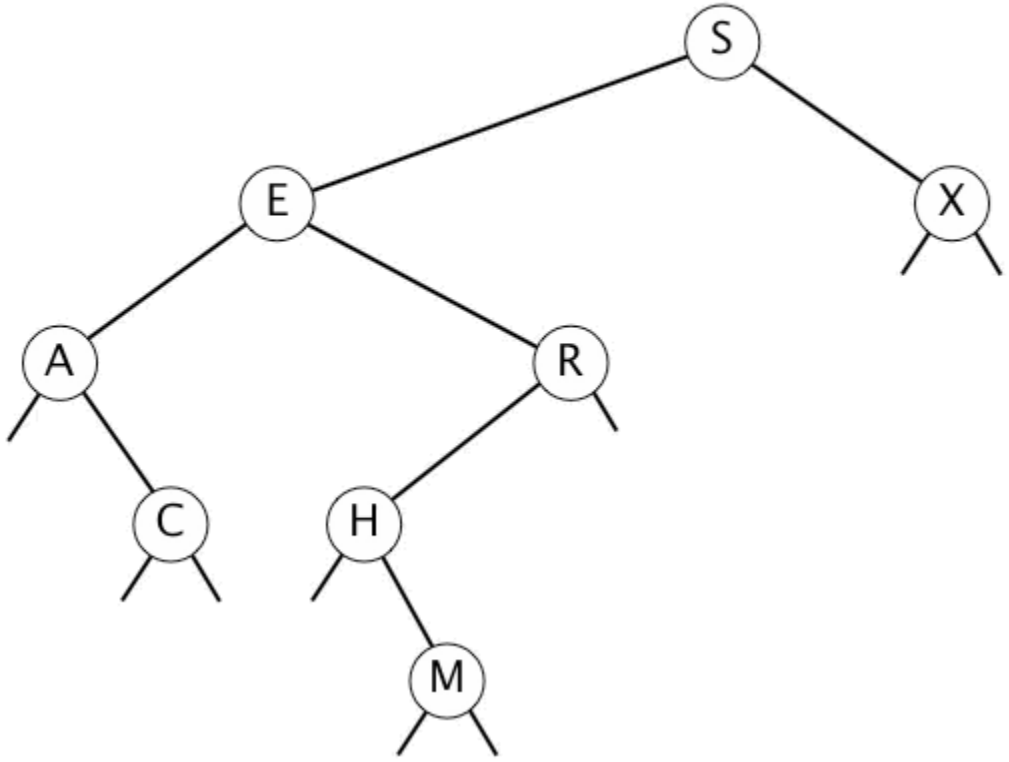


- Orden simétrico: Cada nodo tiene una clave
  - La clave del nodo es mayor a todas las del árbol izquierdo
  - La clave del nodo es menor a todas las del árbol derecho

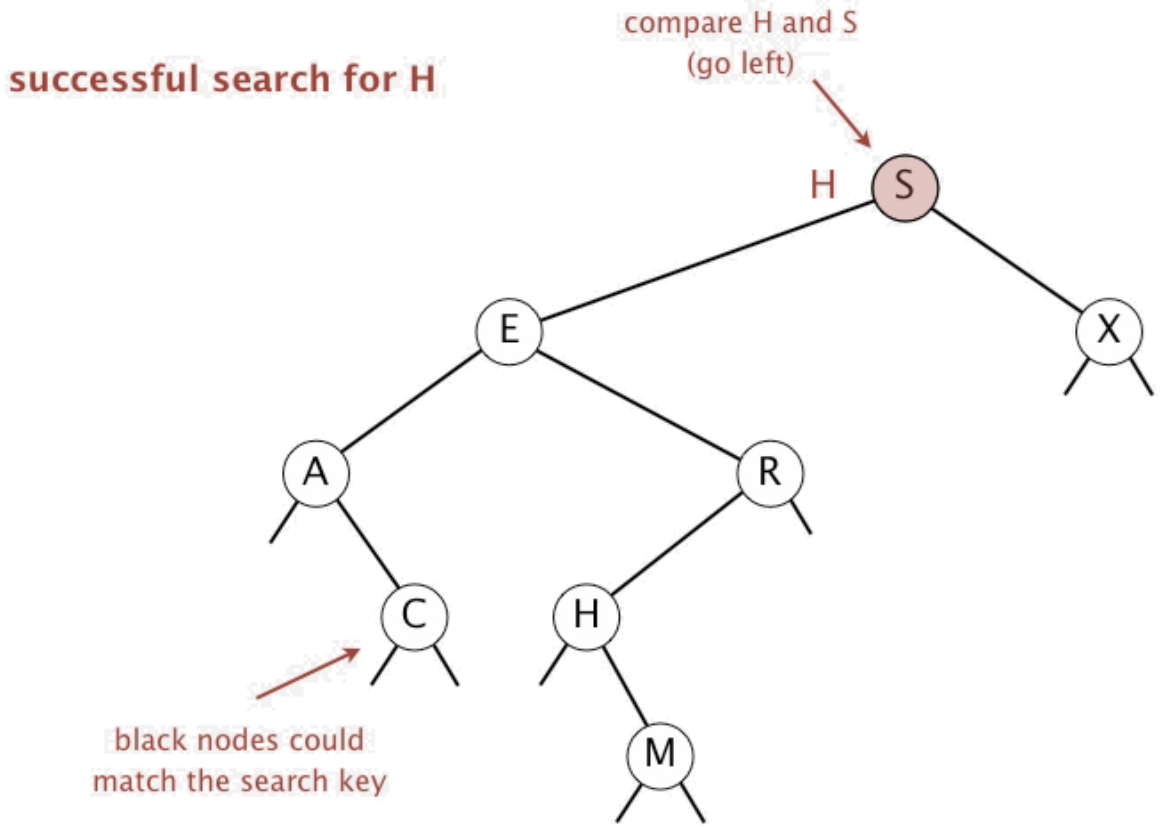


Árboles binarios de búsqueda (BST)

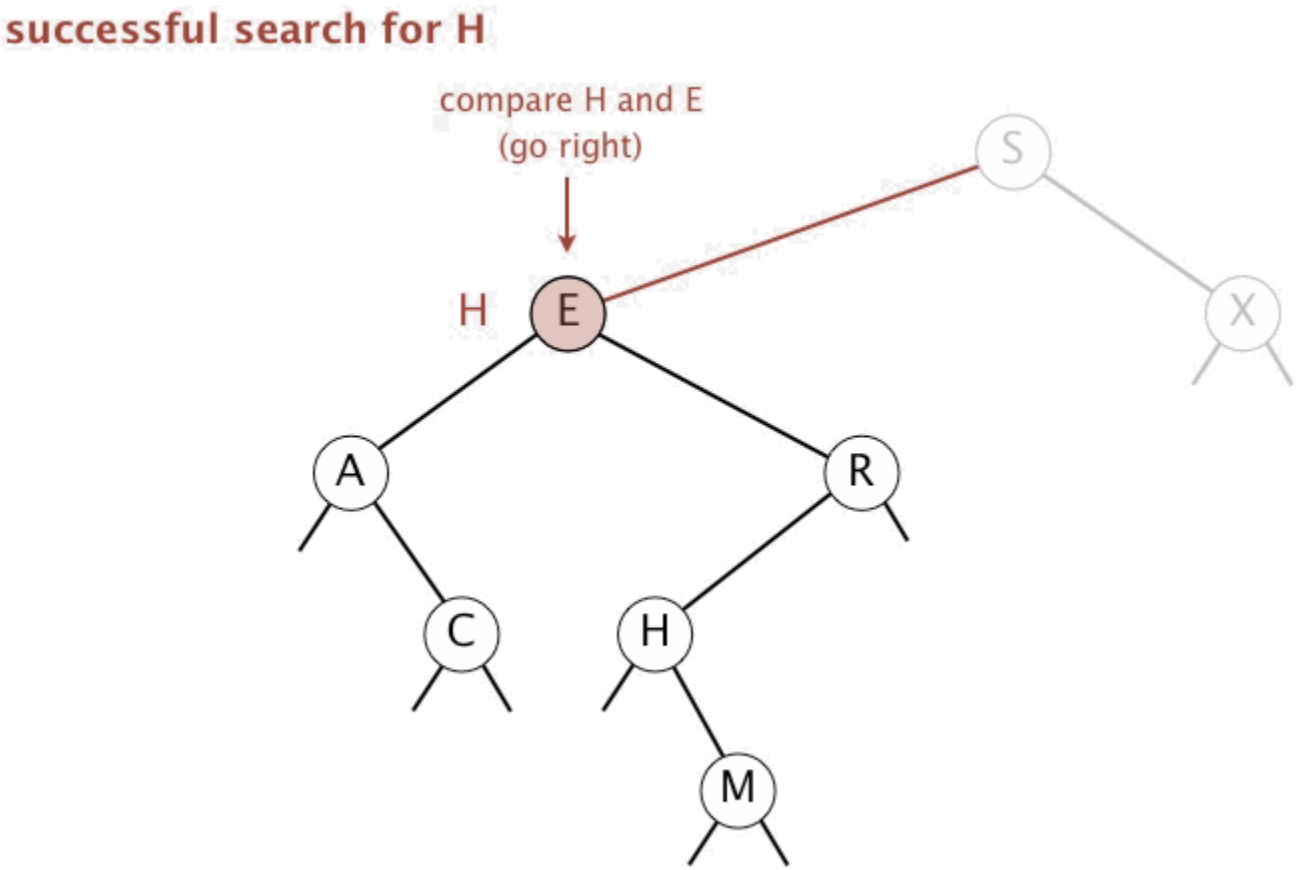
successful search for H



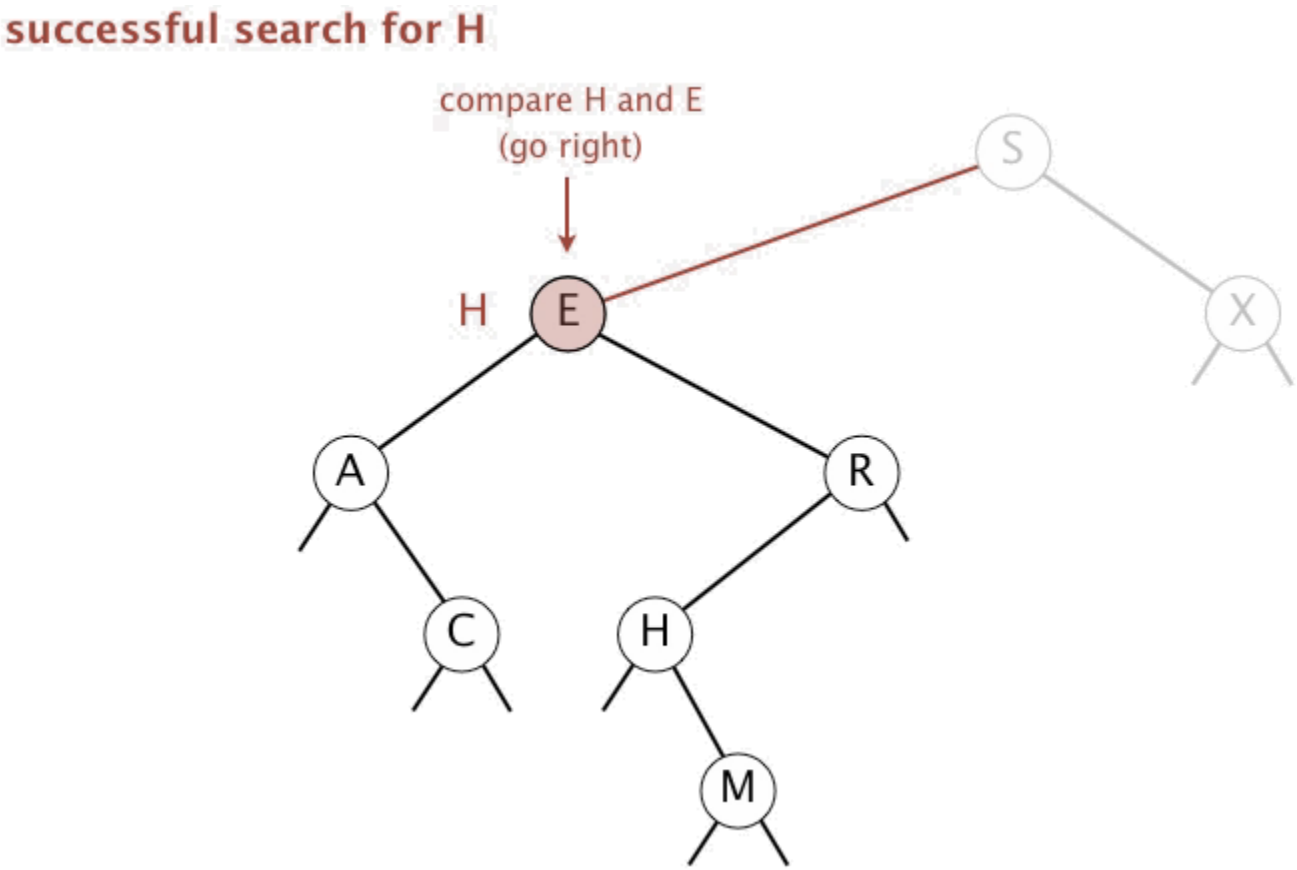
Árboles binarios de búsqueda (BST)



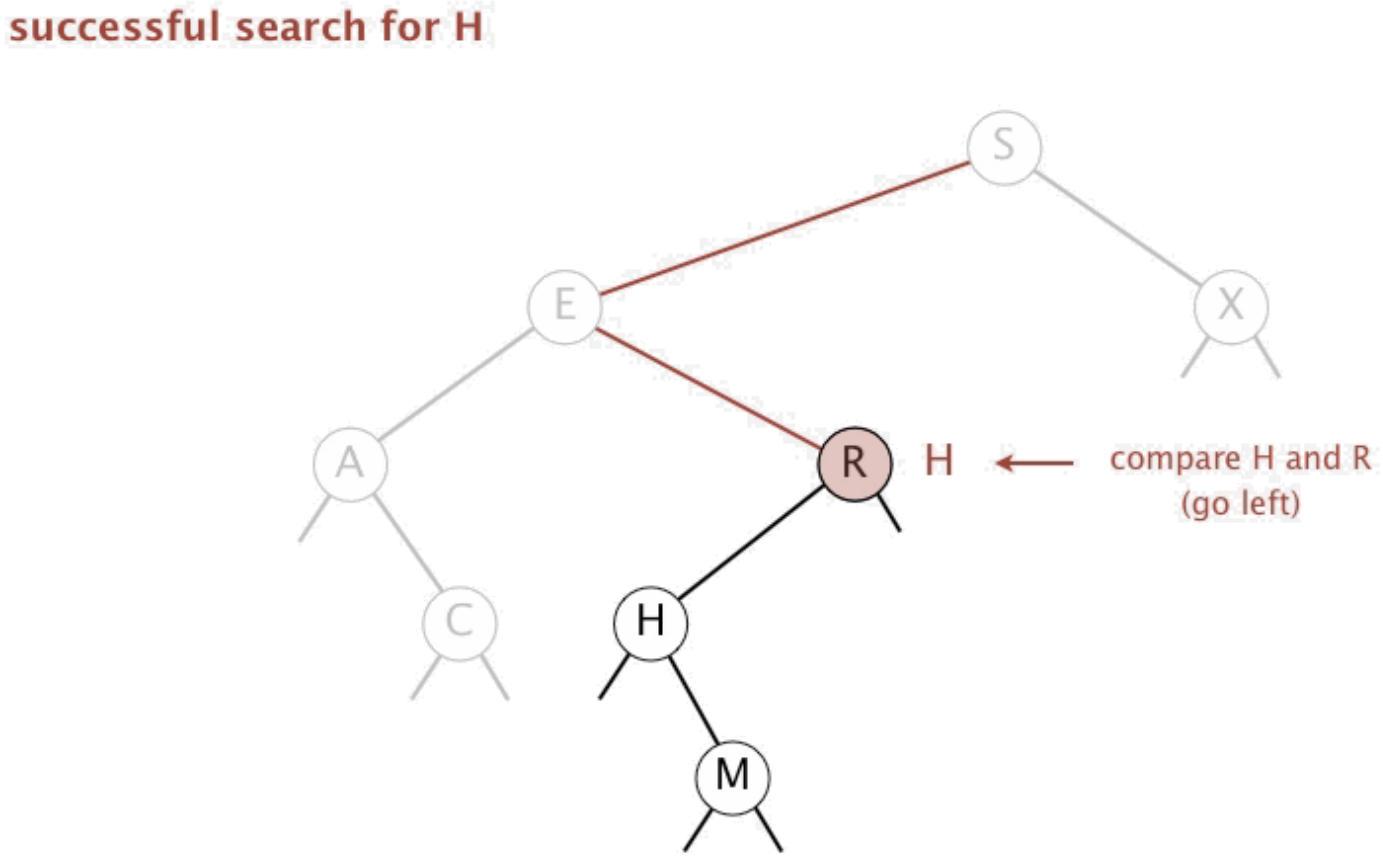
Árboles binarios de búsqueda (BST)



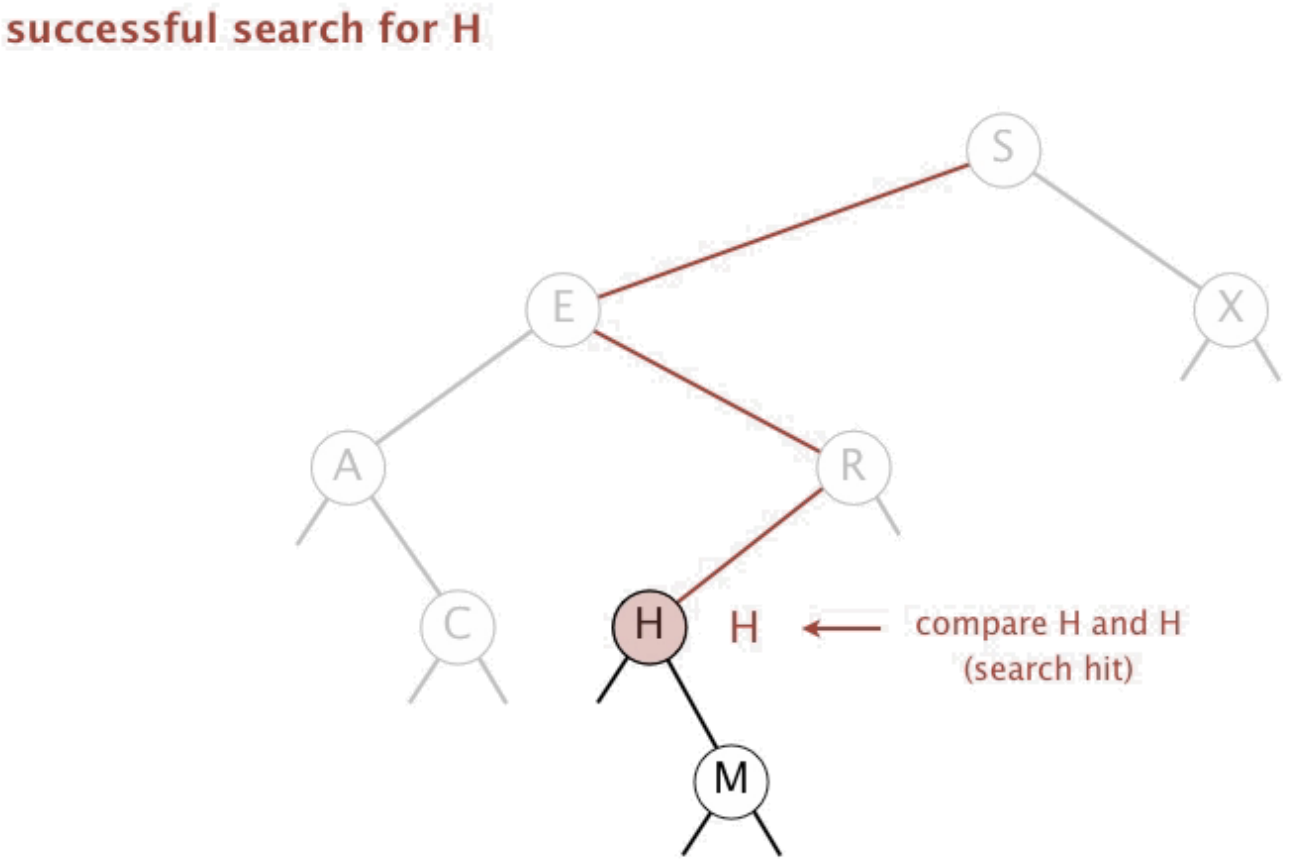
Árboles binarios de búsqueda (BST)



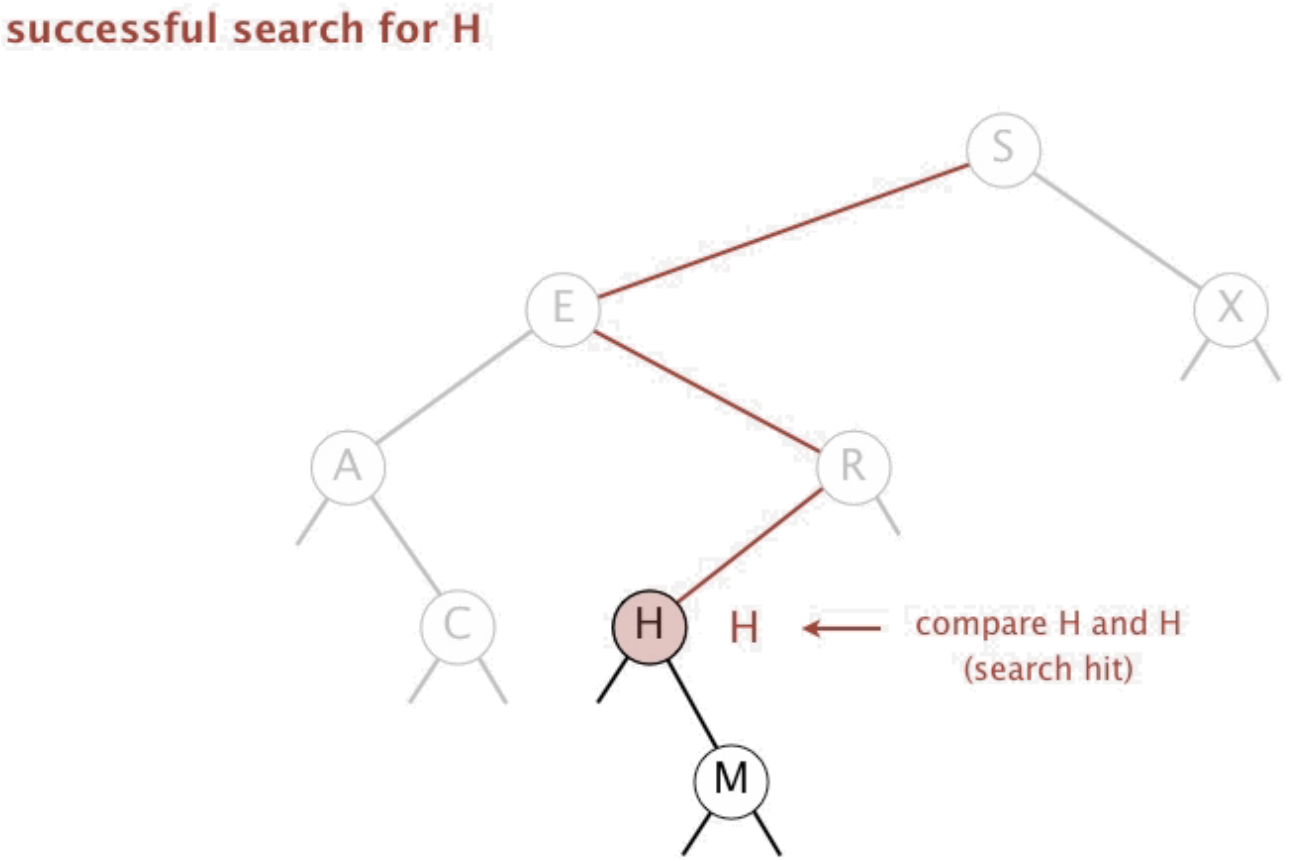
Árboles binarios de búsqueda (BST)



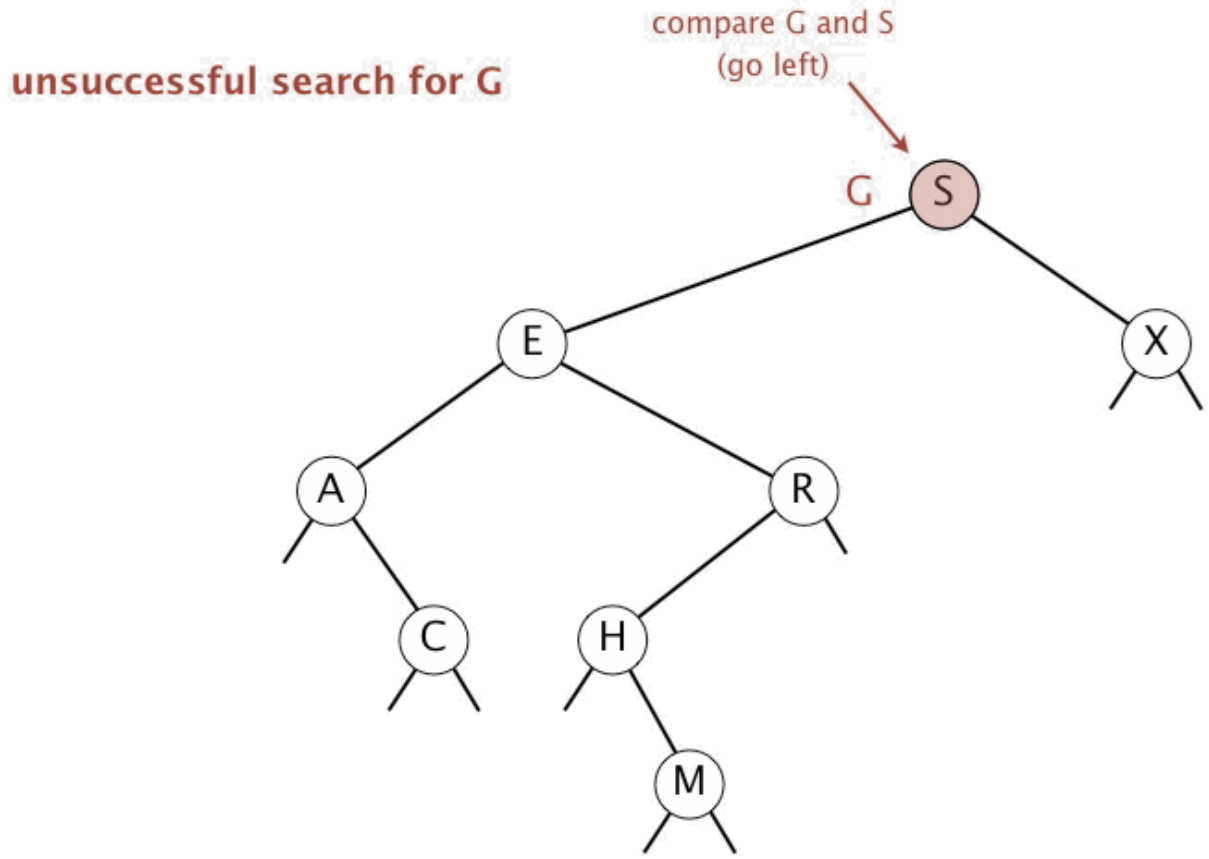
Árboles binarios de búsqueda (BST)



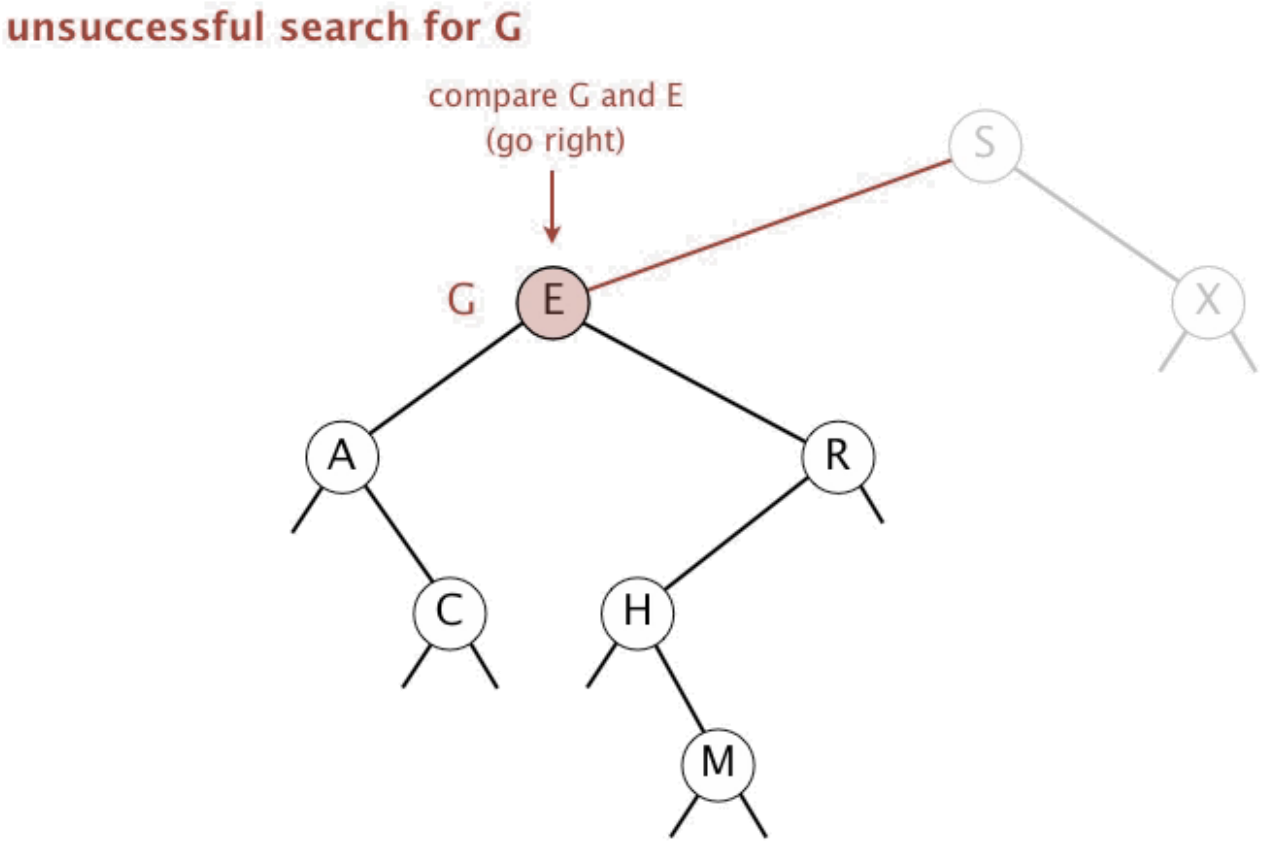
Árboles binarios de búsqueda (BST)



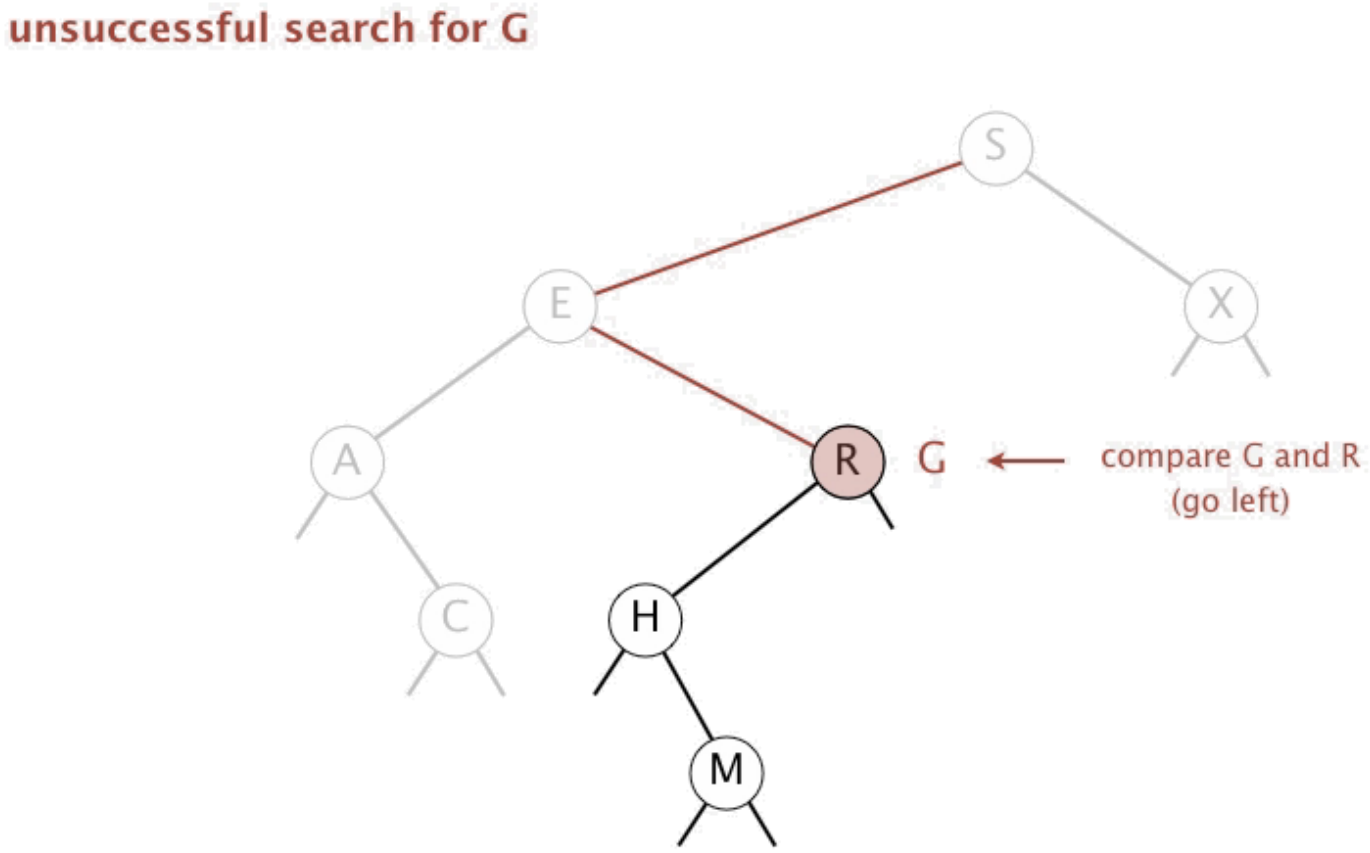
Árboles binarios de búsqueda (BST)



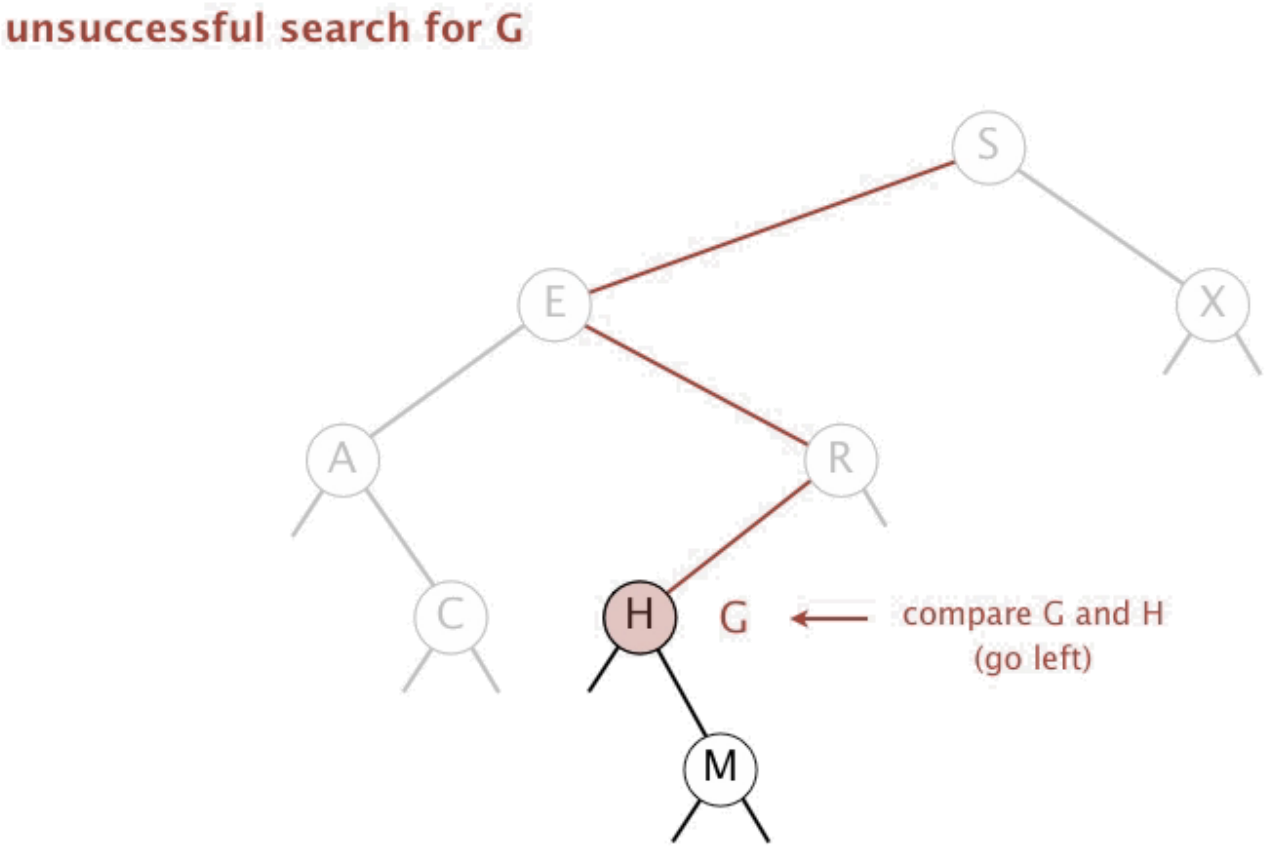
Árboles binarios de búsqueda (BST)



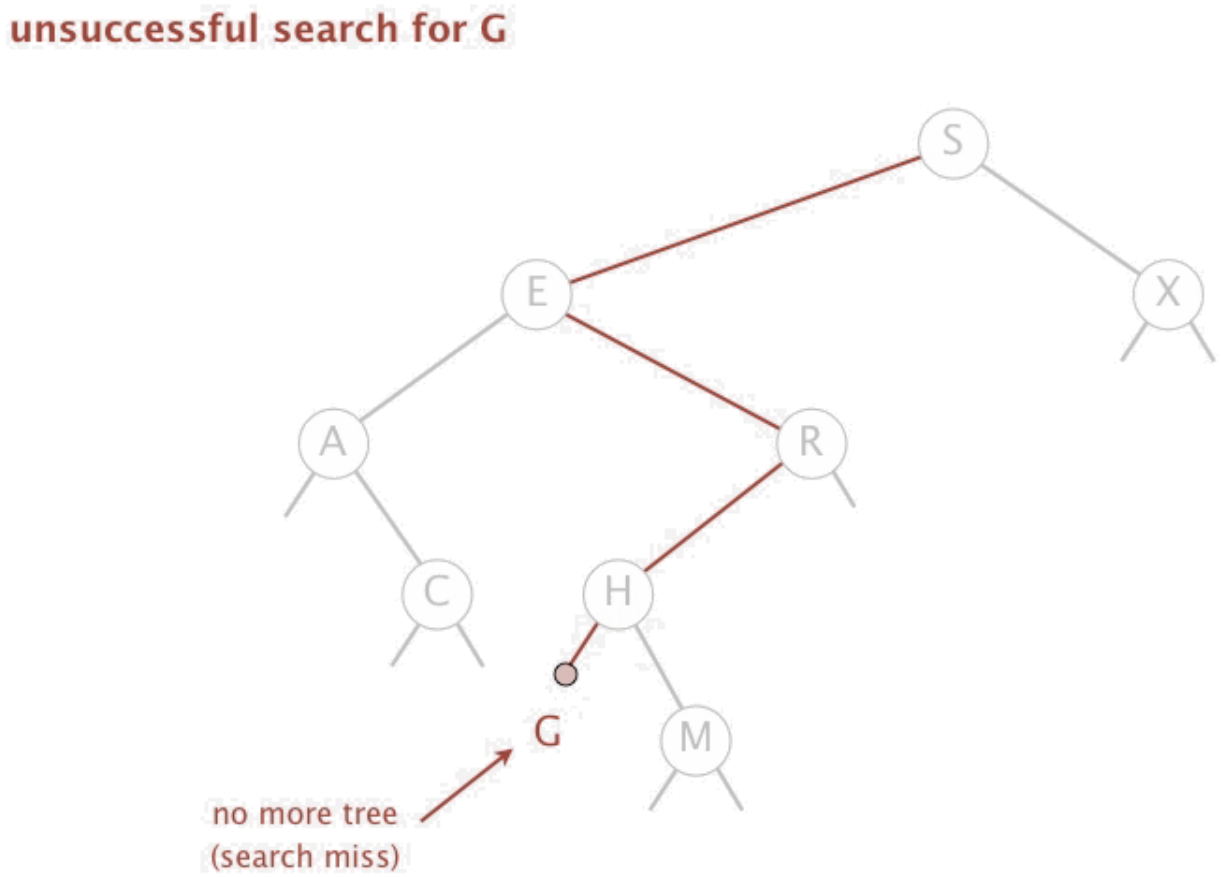
Árboles binarios de búsqueda (BST)



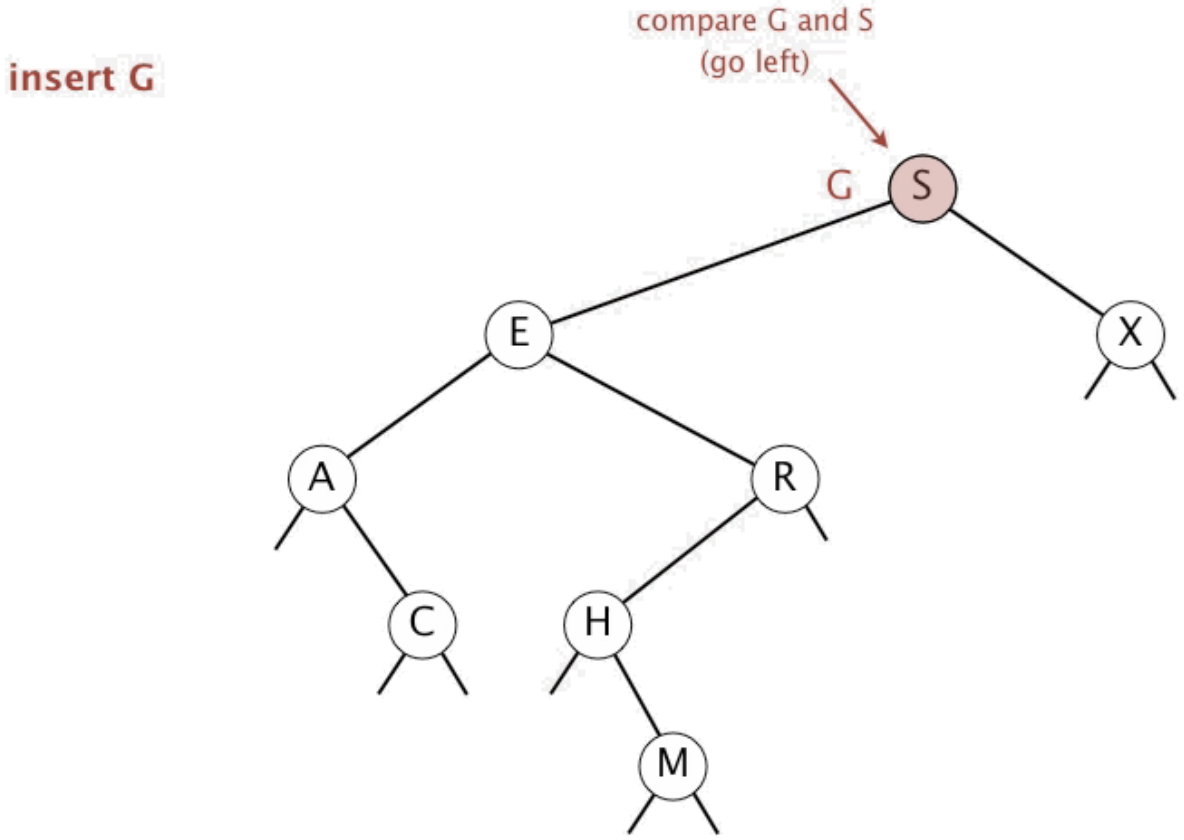
Árboles binarios de búsqueda (BST)



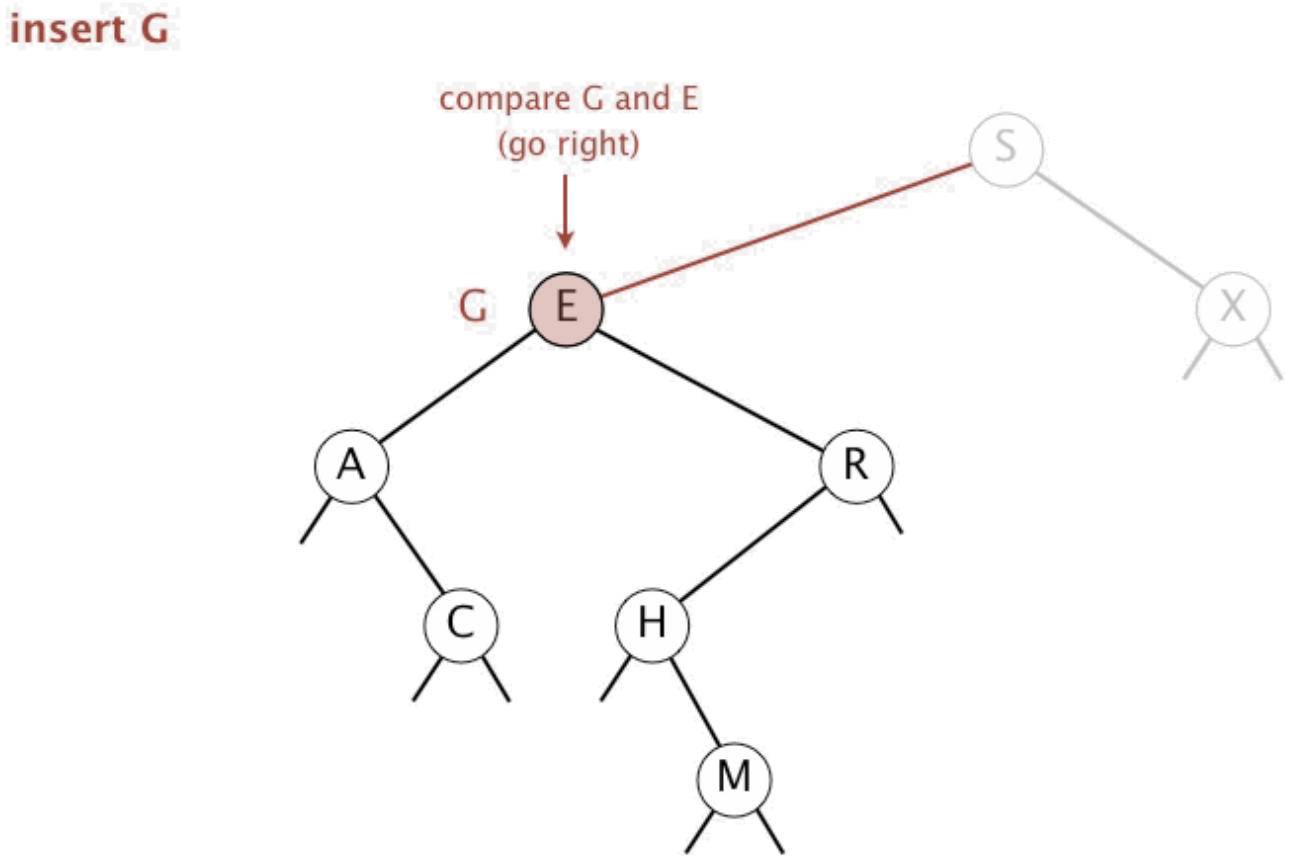
Árboles binarios de búsqueda (BST)



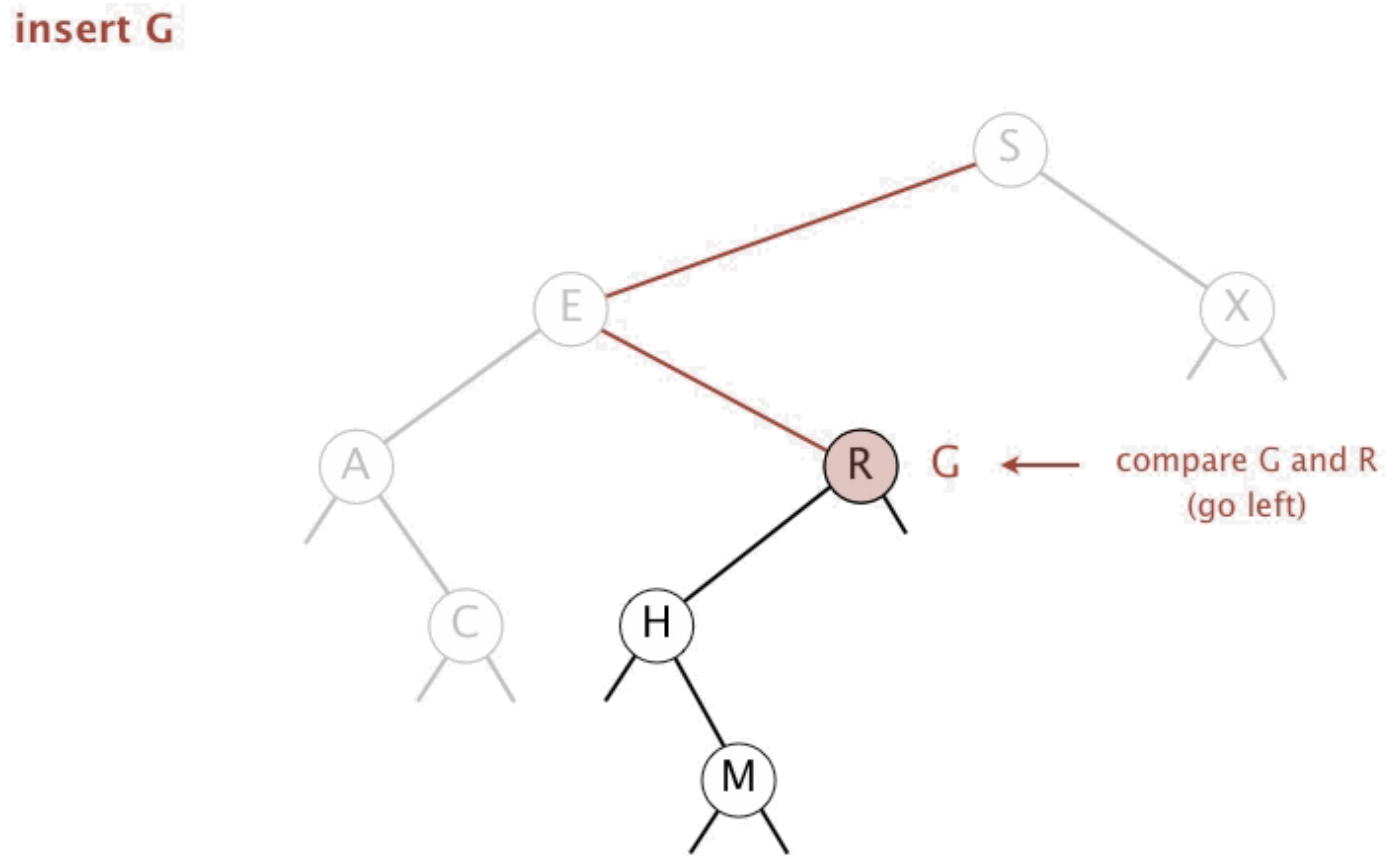
Árboles binarios de búsqueda (BST)



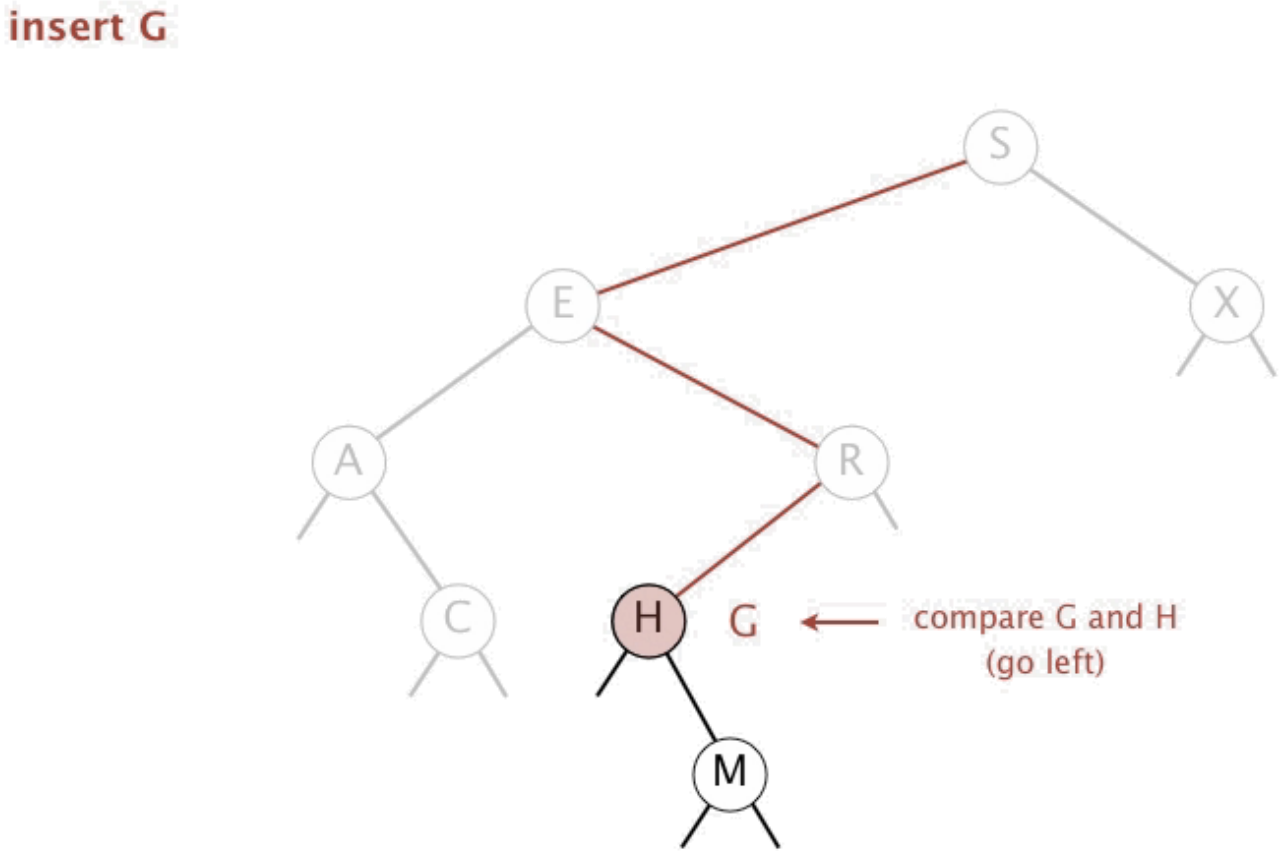
Árboles binarios de búsqueda (BST)



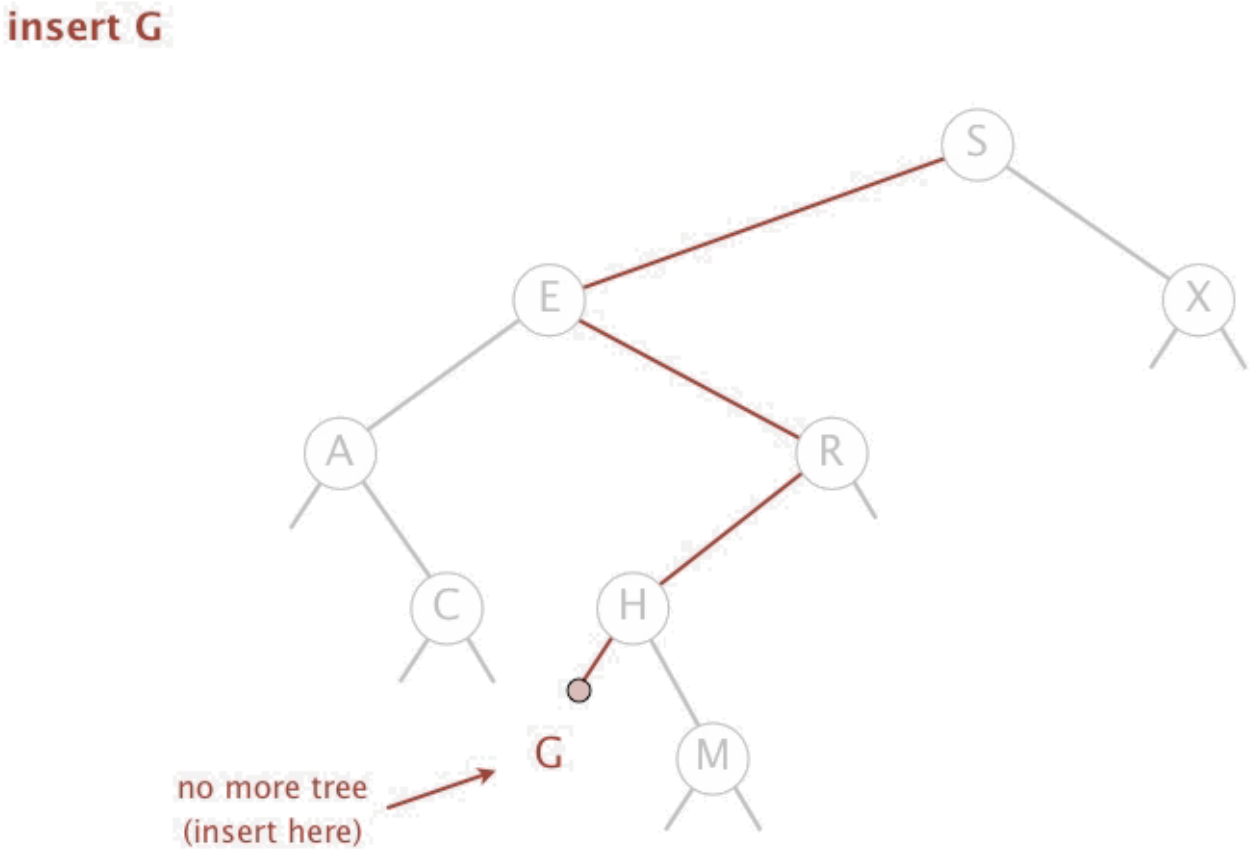
Árboles binarios de búsqueda (BST)



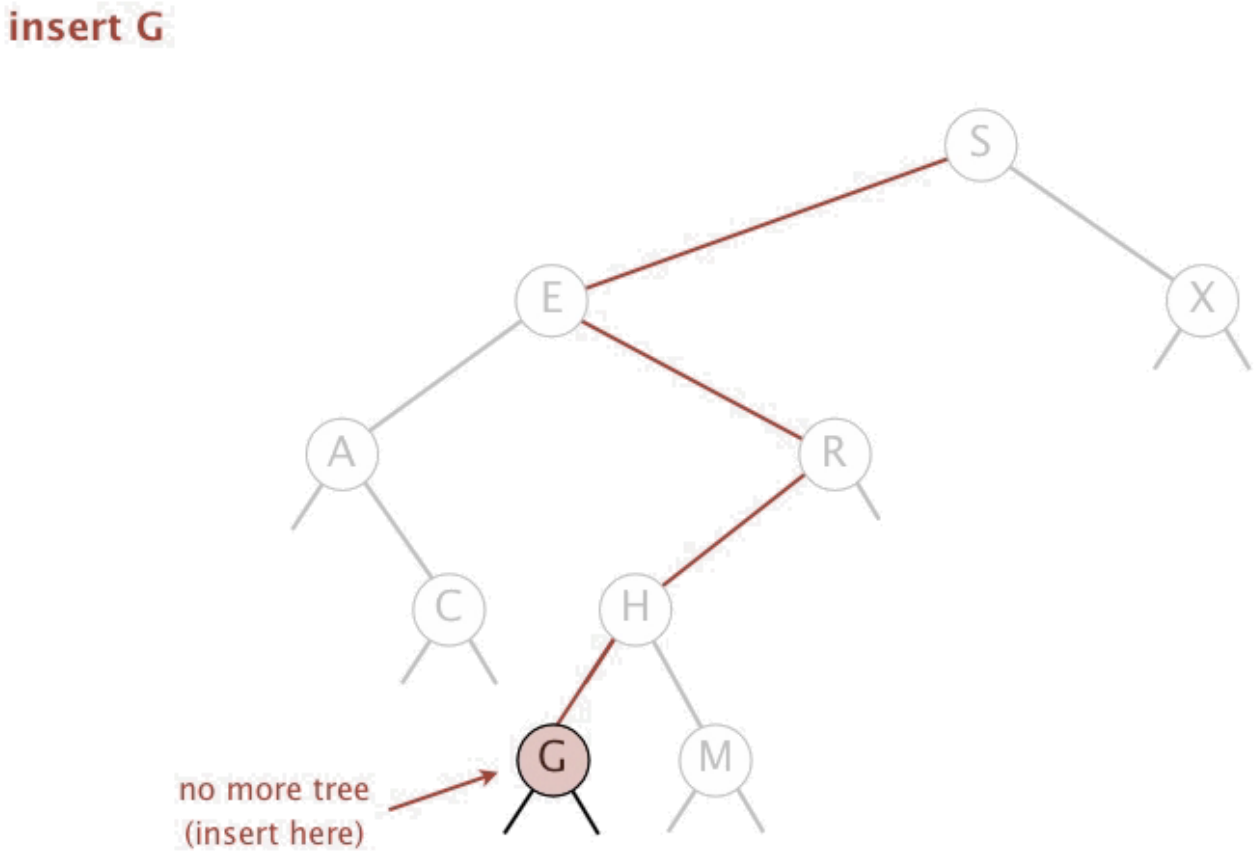
Árboles binarios de búsqueda (BST)



Árboles binarios de búsqueda (BST)



Árboles binarios de búsqueda (BST)

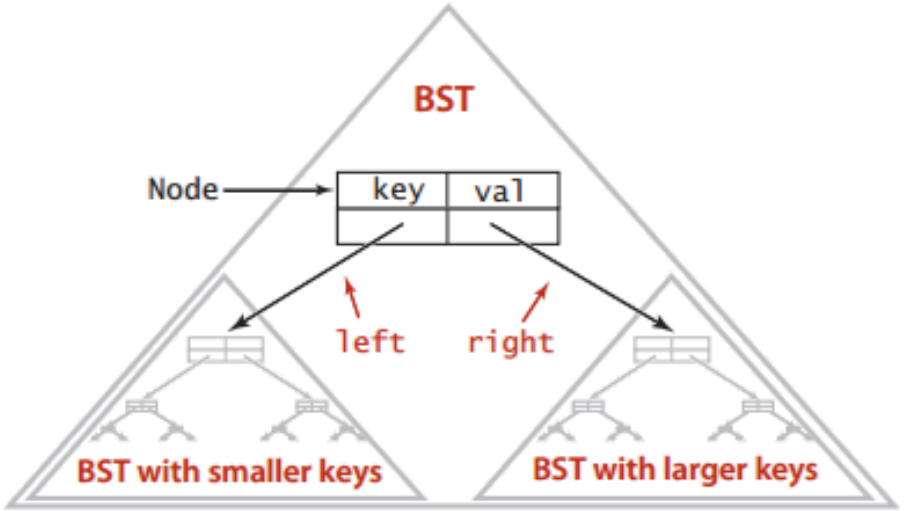


BST en Java

BST en Java

- Árbol es un nodo raíz (Node)

- **Node tiene 4 atributos**
  - Llave (Comparable)
  - Valor (no null)
  - left (Node)
  - Right (Node)



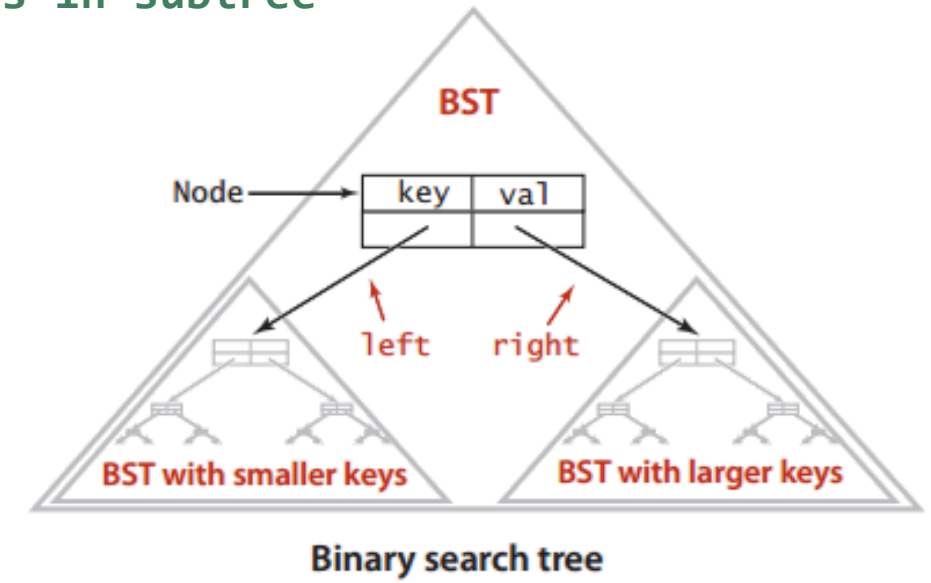
Binary search tree

BST en Java

```
private class Node
{
        private Key key;           // sorted by key
        private Value val;         // associated data
        private Node left, right;  // left and right subtrees
        private int size;          // number of nodes in subtree

        public Node(Key key, Value val, int size) {
            this.key = key;
            this.val = val;
            this.size = size;
        }
}
```

Key y Value son generic; Key es Comparable



Binary search tree

BST search: en Java

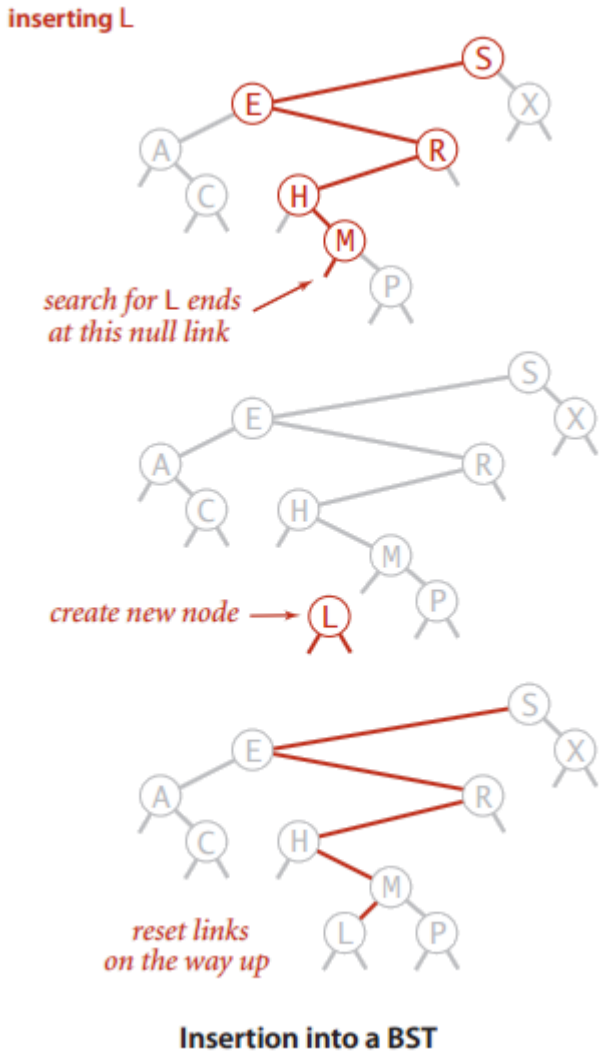
- Comparo llaves con la de nodo actual
- **Si son iguales , retorno valor**
  - Si es menor, nodo actual es el de la izquierda
  - Si es mayor, nodo actual es el de la derecha
- Si el nodo actual es null, quiere decir que la búsqueda falló, retorno null

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    {
            int cmp = key.compareTo(x.key);
            if (cmp < 0) x = x.left;
            else if (cmp > 0) x = x.right;
            else if (cmp == 0) return x.val;
    }
    return null;
}
```

El número de comparaciones es igual a 1 + profundidad del nodo.

BST insert: en Java

- Dos casos:
  - Sobrescribir valor si existe
  - Añadir nodo si no se encontró

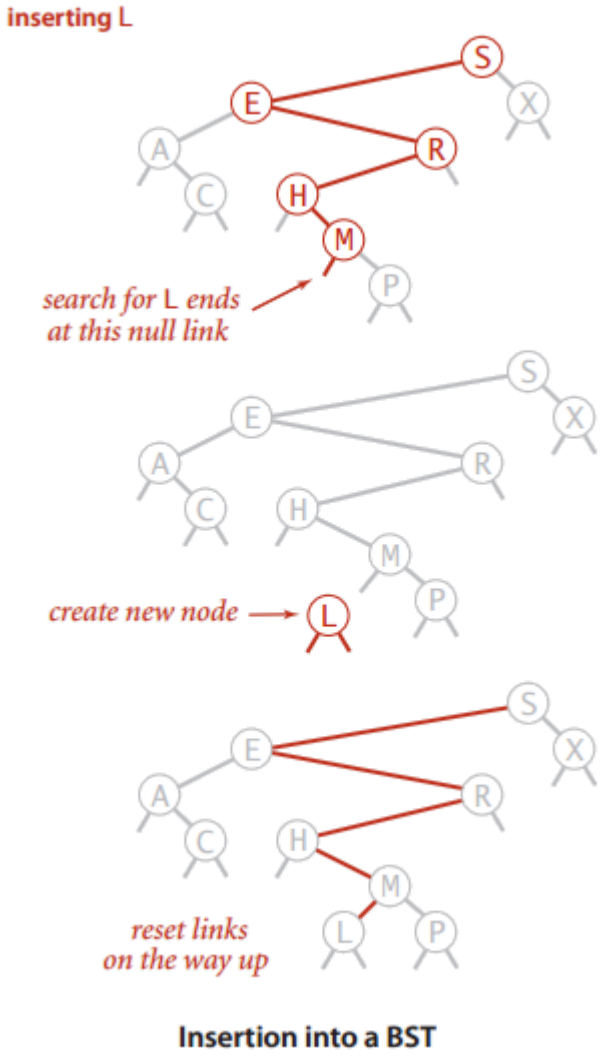


Insertion into a BST

BST insert: en Java

```
public void put(Key key, Value val)
       {root = put(root, key, val); }

private Node put(Node x, Key key, Value val)
{
       if (x == null) return new Node(key, val);
       int cmp = key.compareTo(x.key);
       if (cmp < 0)
              x.left = put(x.left, key, val);
       else if (cmp > 0)
              x.right = put(x.right, key, val);
       else if (cmp == 0)
              x.val = val;
    return x;
}
```

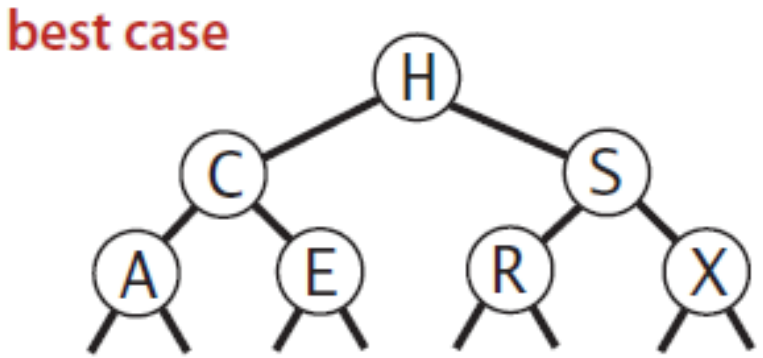


Insertion into a BST

El número de comparaciones es igual a 1 + profundidad del nodo.

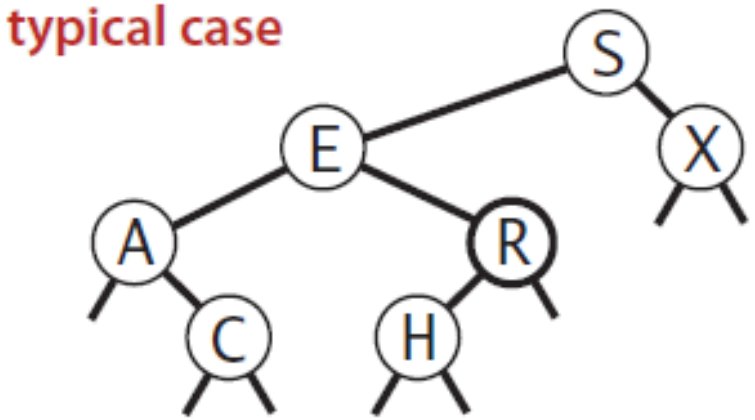
BST insert: en Java

- Mejor caso
  - Las llaves llegan en orden que dejan el árbol balanceado



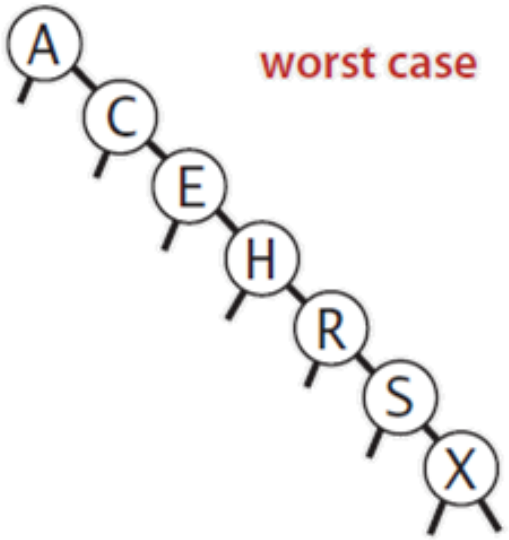
BST insert: en Java

- Caso promedio
  - Las llaves llegan en orden aleatorio

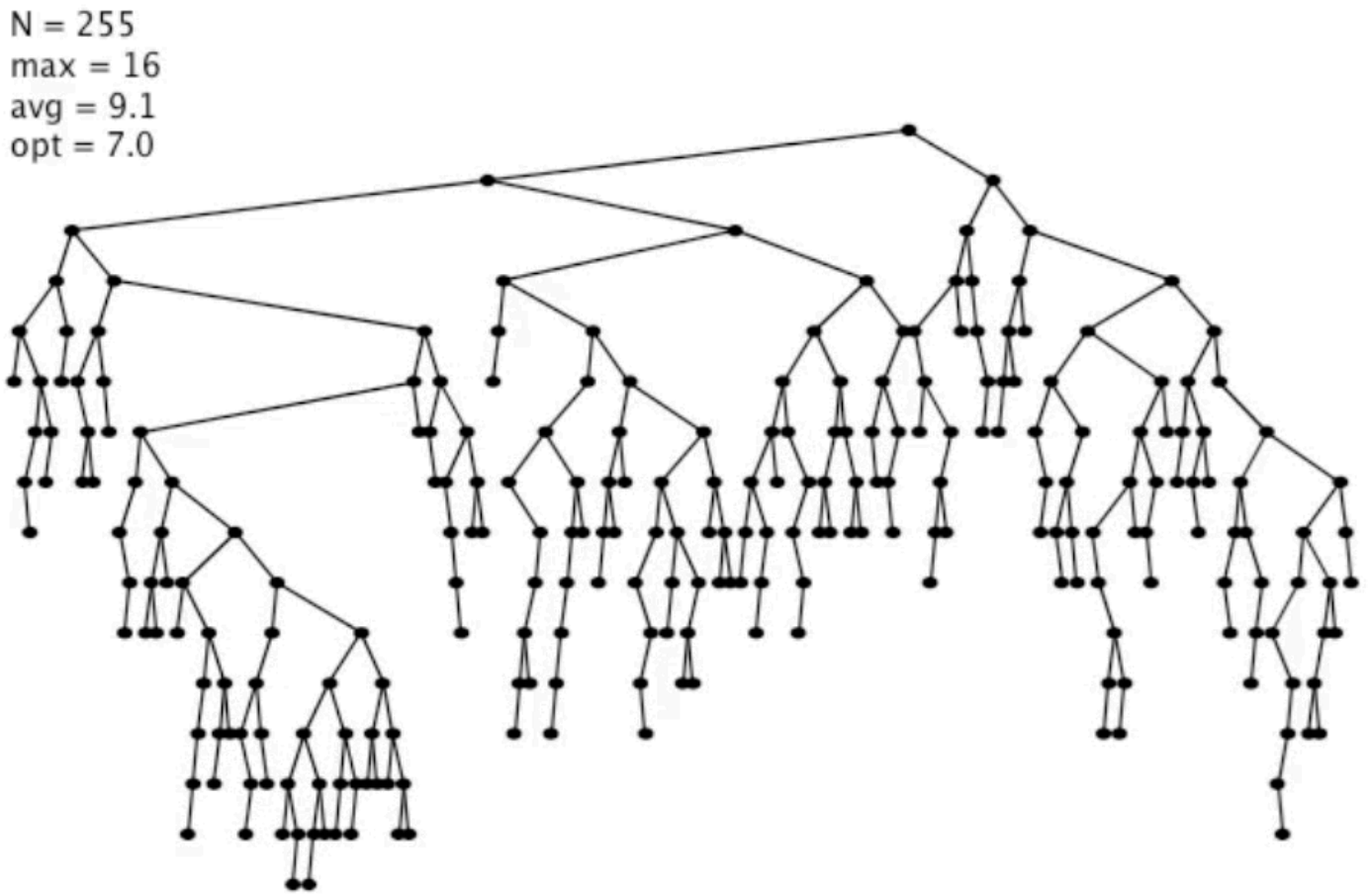


BST en Java: put(Key k, Value v)

- Peor caso
  - Las llaves llegan en orden



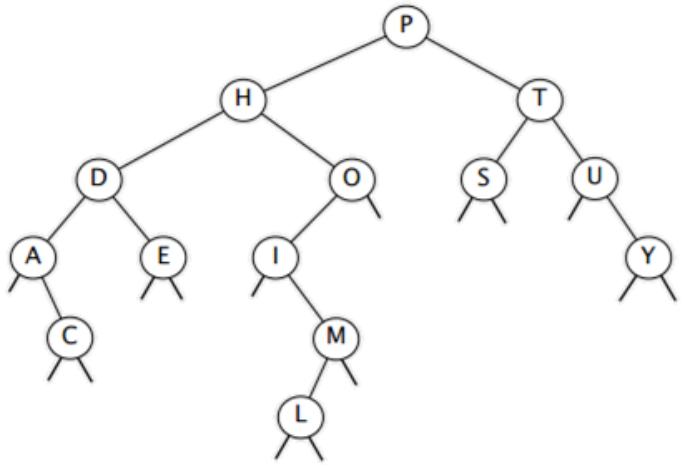
BST en Java: put(Key k, Value v)



Quicksort y BST

– Comparten mismo análisis matemático

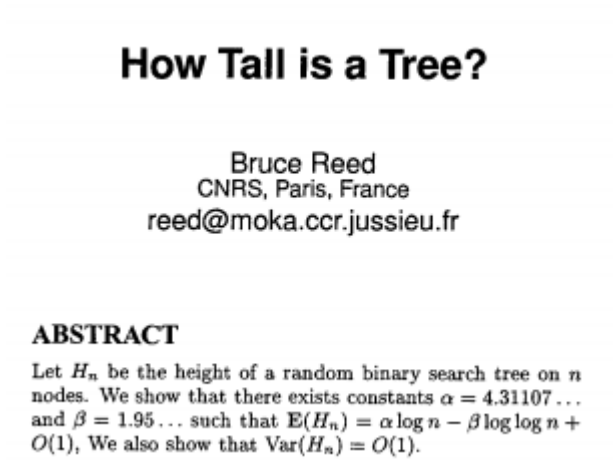
0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
H	L	E	A	D	O	M	C	I	P	T	Y	U	S
D	C	E	A	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y



La correspondencia es 1-1 si el arreglo no tiene claves duplicadas.

BSTs: análisis matemático

- Número esperado de comparaciones: $\sim 2 \cdot \ln N$ (quicksort)
- [Reed,2003] Altura esperada: ~ $4.311 \cdot \ln N$



How Tall is a Tree?

Bruce Reed
CNRS, Paris, France
reed@moka.ccr.jussieu.fr

ABSTRACT

Let H_n be the height of a random binary search tree on n nodes. We show that there exists constants $\alpha = 4.31107\ldots$ and $\beta = 1.95\ldots$ such that $\mathbf{E}(H_n) = \alpha \log n - \beta \log \log n + O(1)$, We also show that $\mathrm{Var}(H_n) = O(1)$.

- Peor caso $N - 1$ (mismas probabilidades que en quicksort)

Resumen implementaciones ST

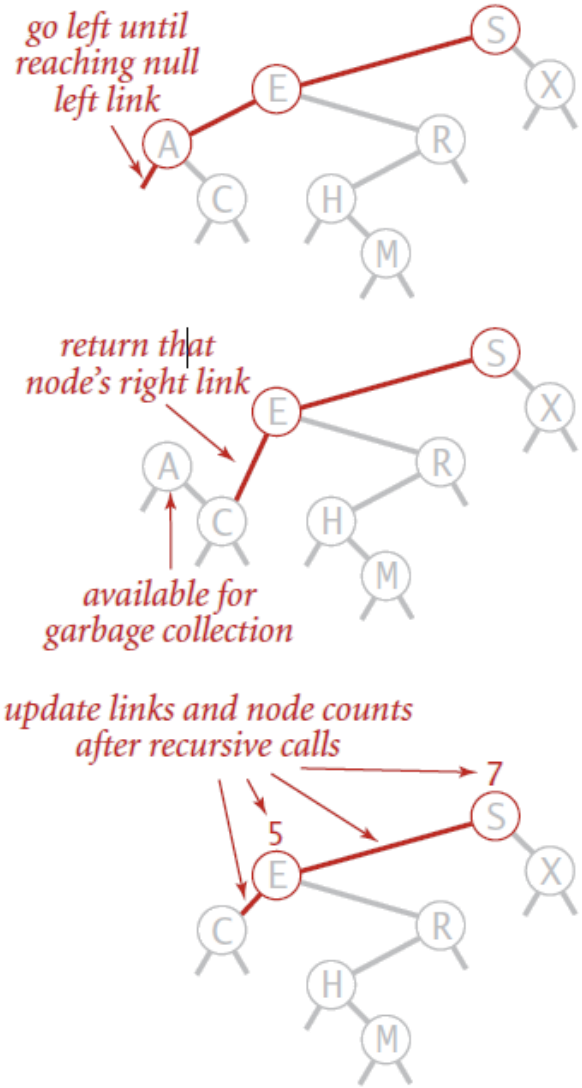
implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	???	✔	`compareTo()`

Delete en BST

Delete en BST

Para eliminar la clave mínima:

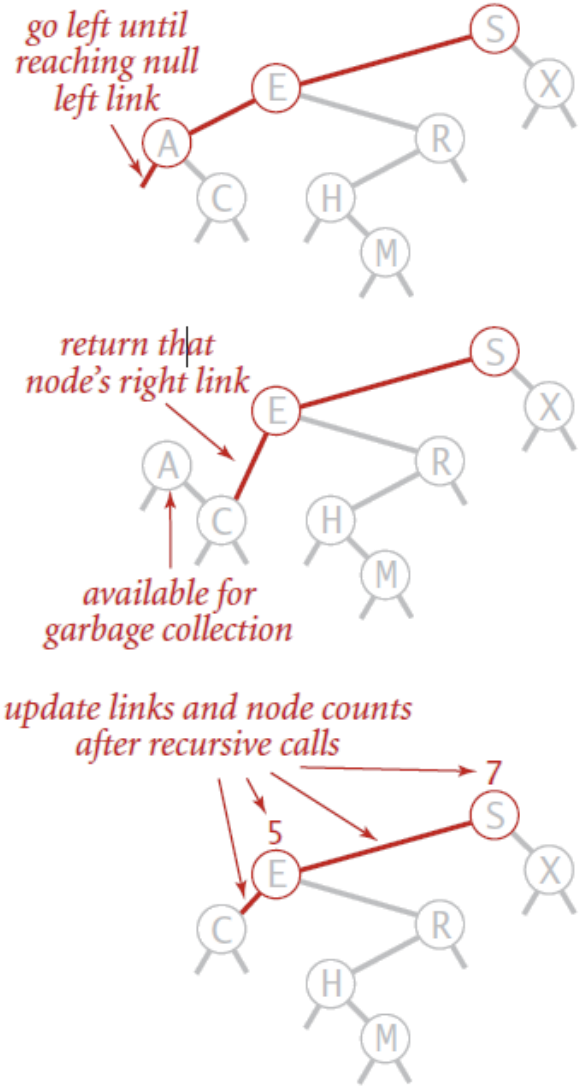
- Gire a la izquierda hasta encontrar un nodo con un enlace nulo a la izquierda.
- Reemplace ese nodo por su enlace derecho.
- Actualizar los conteos del subárbol.



Delete en BST

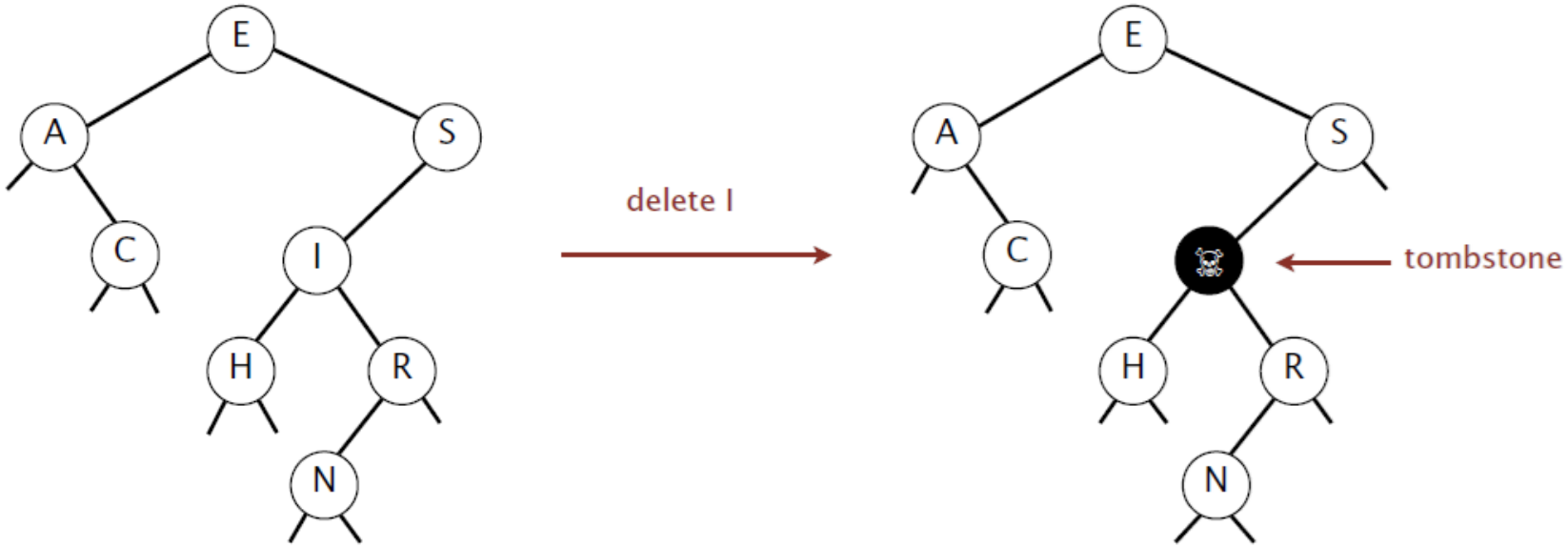
```
public void deleteMin()
{ root = deleteMin(root); }

private Node deleteMin(Node x)
{
        if (x.left == null) return x.right;
        x.left = deleteMin(x.left);
        x.count = 1 + size(x.left) + size(x.right);
        return x;
}
```



Delete en BST

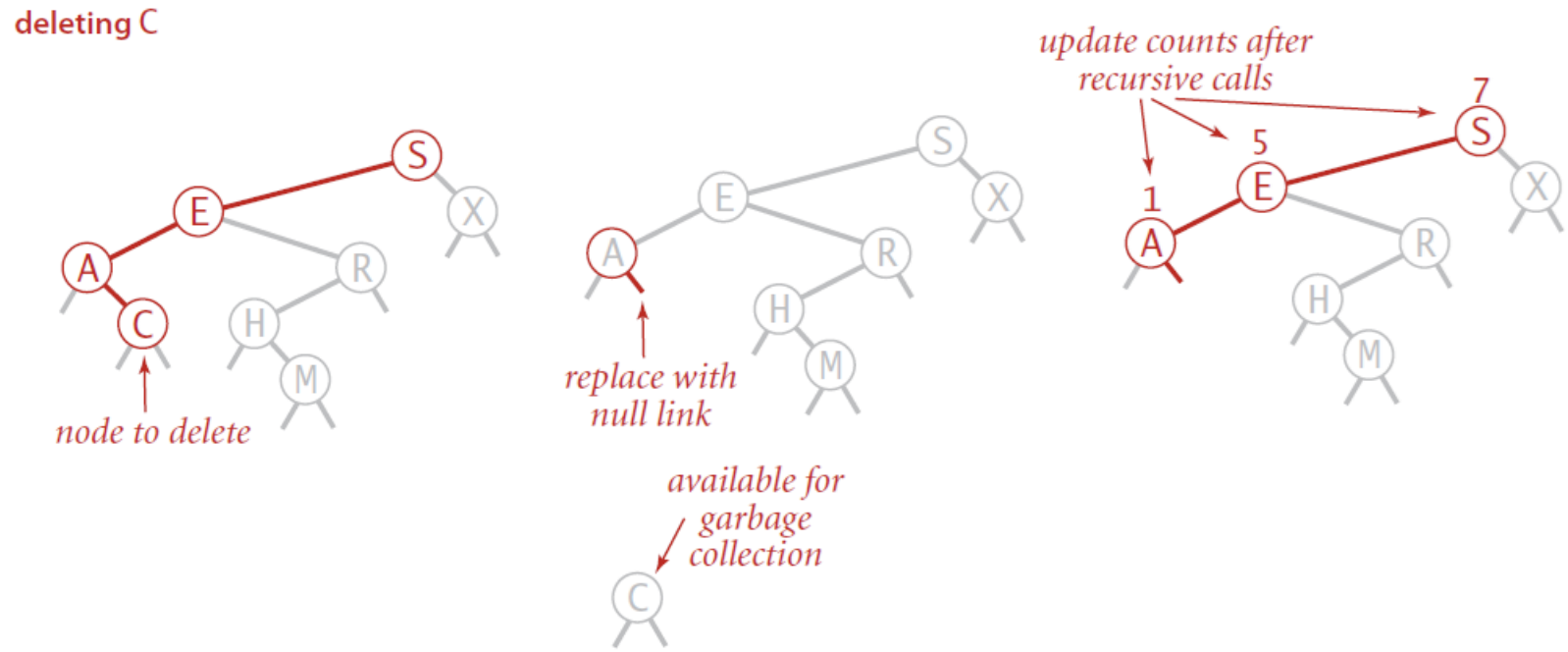
– Opción 0: Dejar el valor en **null**, la clave queda en el árbol para guiar la búsqueda



$\sim 2 \cdot \ln N'$ por inserción, búsqueda y eliminación (si las claves están en orden aleatorio), donde N' es el número de pares clave-valor insertados en el BST.

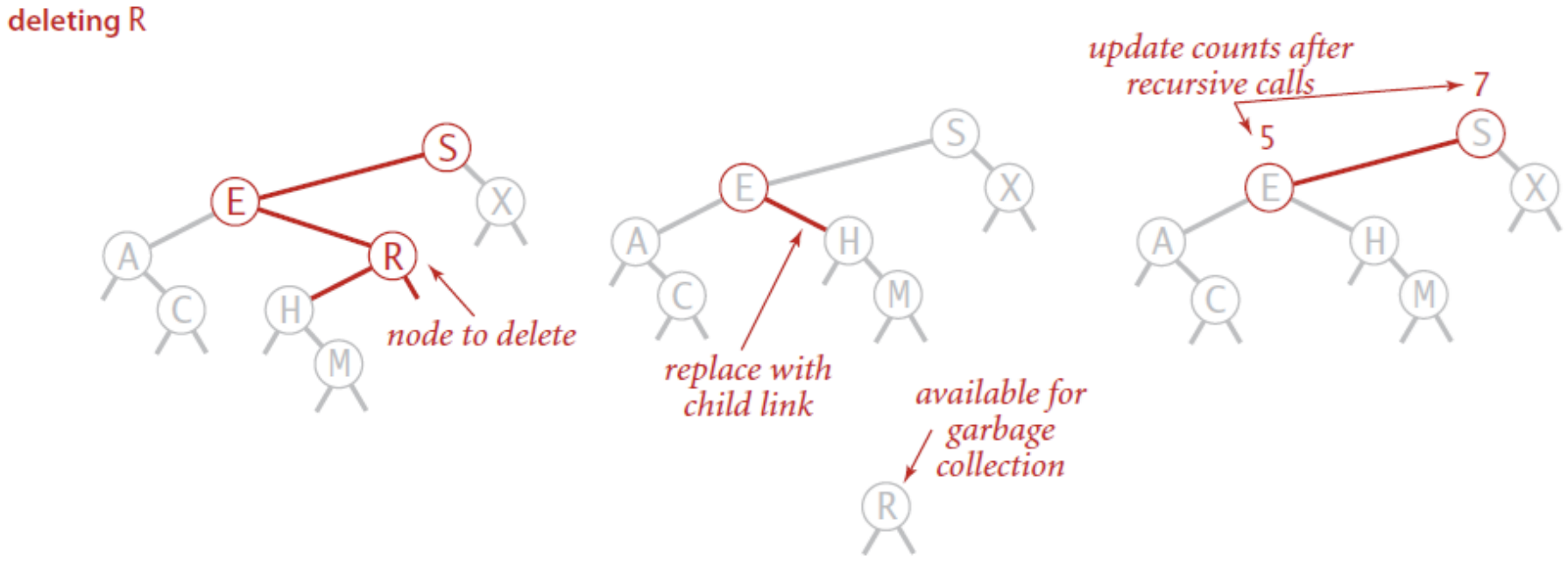
Delete en BST

- Opción 1: Borrar el nodo
  - Caso 0: No tiene hijos, cambiar referencia del papá a null



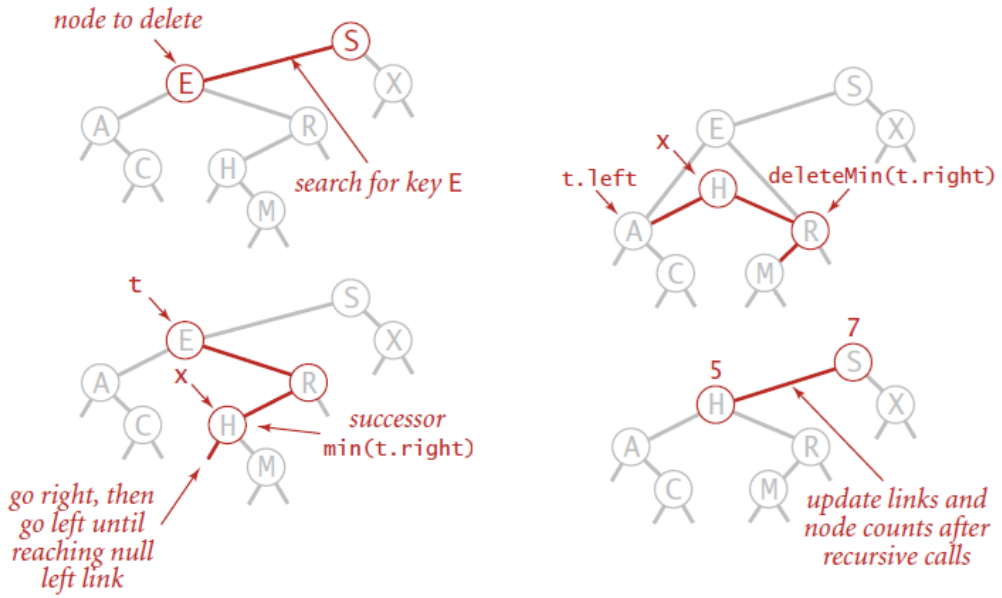
Delete en BST

- Opción 1: Borrar el nodo
  - Caso 1: Tiene 1 hijo, "darle el hijo al abuelo"



Delete en BST

- Opción 1: Borrar el nodo
  - Caso 2: Tiene 2 hijos
    - Encontrar el sucesor, nodo que le sigue en el orden (nodo más a la izquierda en hijo derecho)
    - Saco el nodo sucesor del árbol y lo reemplazo por el nodo que se quiere eliminar
    - Actualizo referencias del nodo



Delete en BST

```
public void delete(Key key) { root = delete(root, key); }
private Node delete(Node x, Key key) {
        if (x == null) return null;
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x.left = delete(x.left, key);
        else if (cmp > 0) x.right = delete(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 = deleteMin(t.right);
                x.left = t.left;
        }
        x.count = size(x.left) + size(x.right) + 1;
        return x;
}
```

buscar clave

Sin hijo derecho

Sin hijo izquierdo

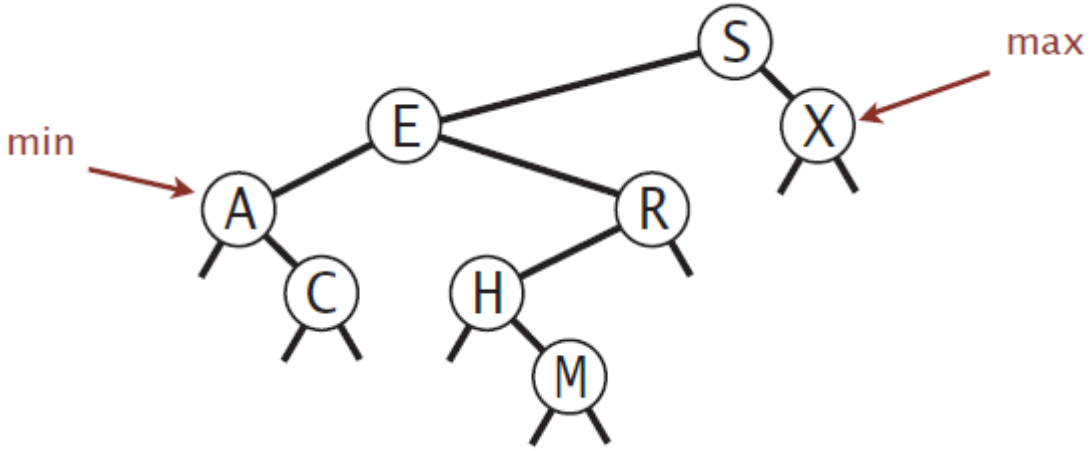
reemplazar con sucesor

actualizar los conteos del subárbol

Operaciones Ordenadas

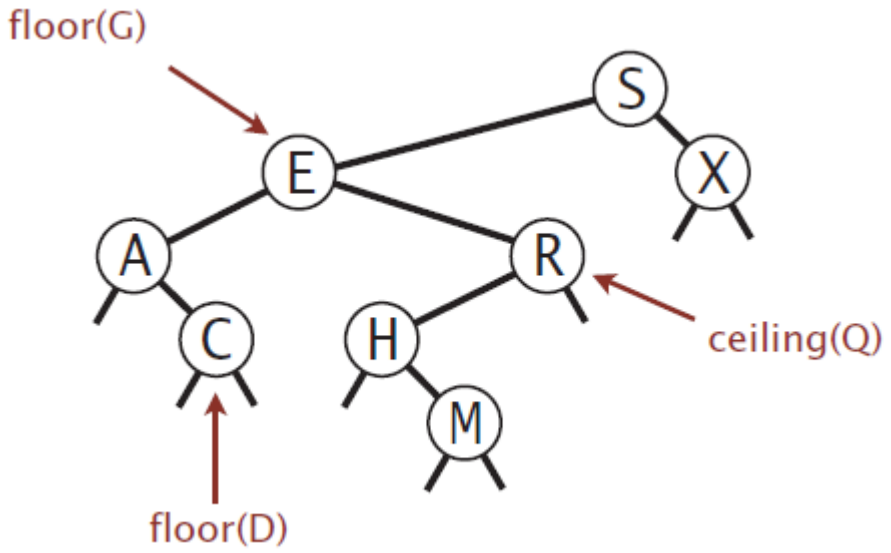
Mínimo y Máximo

- **Mínimo**. La clave más pequeña en la tabla.
- **Máximo**. La clave más grande en la tabla.



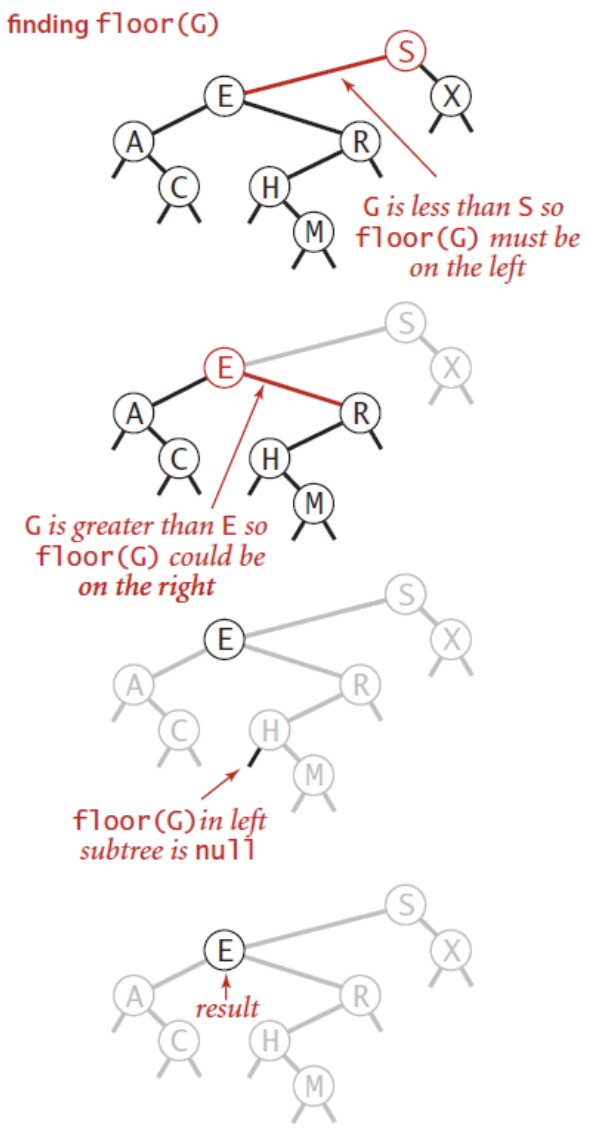
Floor and ceiling

- **Floor**. Clave más grande ≤ una clave dada.
- **Ceiling**. La clave más pequeña ≥ una clave dada.



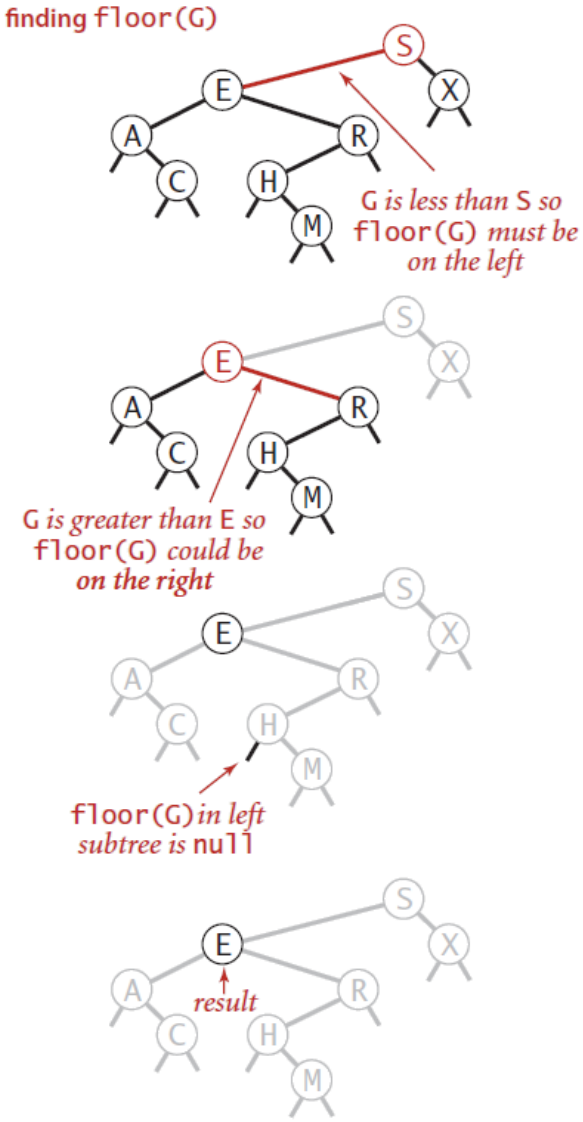
Floor

- Caso 1: La clave k es k
- Caso 2: La clave k esta en el árbol izquierdo
- Caso 3: La clave esta en el árbol derecho
  - Si k no existe en el árbol derecho, el nodo actual es el buscado



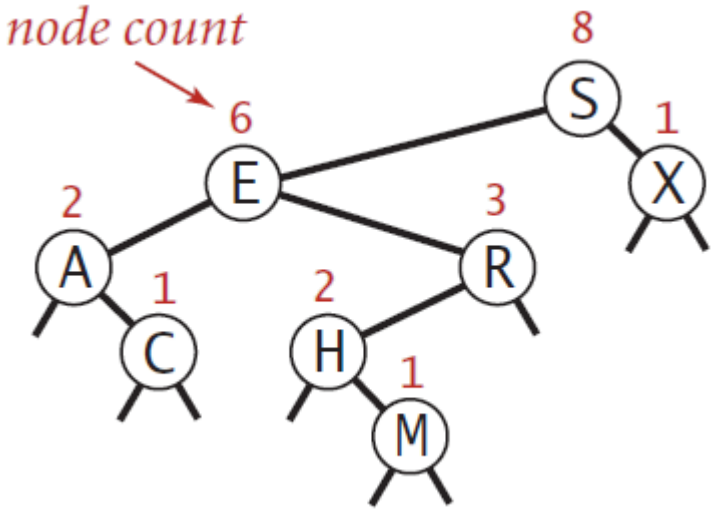
Floor

```
public Key floor(Key key)
{
        Node x = floor(root, key);
        if (x == null) return null;
        return x.key;
}
private Node floor(Node x, Key key)
{
        if (x == null) return null;
        int cmp = key.compareTo(x.key);
        if (cmp == 0) return x;
        if (cmp < 0) return floor(x.left, key);
        Node t = floor(x.right, key);
        if (t != null) return t;
        else return x;
}
```



Rank and select

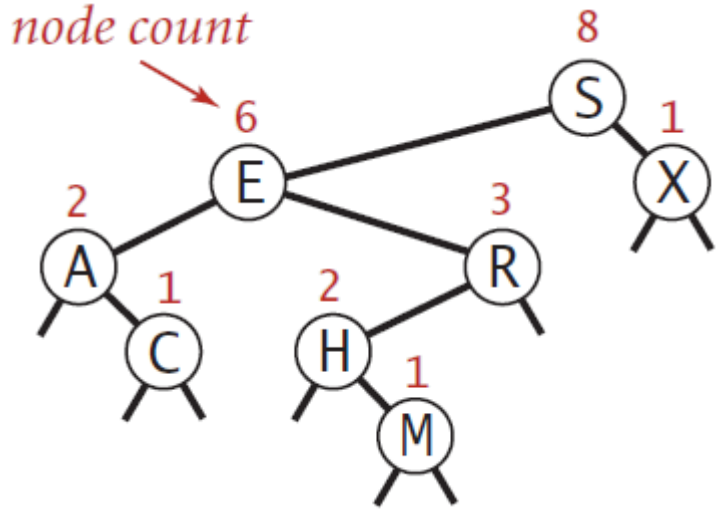
En cada nodo, se almacena la cantidad de nodos en el subárbol enraizado en ese nodo; para implementar `size()`, devuelve el conteo en la raíz.



BST size: Implementación

- Opción 1: Contar cuantos hijos tiene recursivamente (potencialmente puede ser de complejidad N)

- Opción 2: Manejar un atributo



BST size: Implementación

```
private class Node
{
        private Key key;
        private Value val;
        private Node left;
        private Node right;
        private int count;
}
```

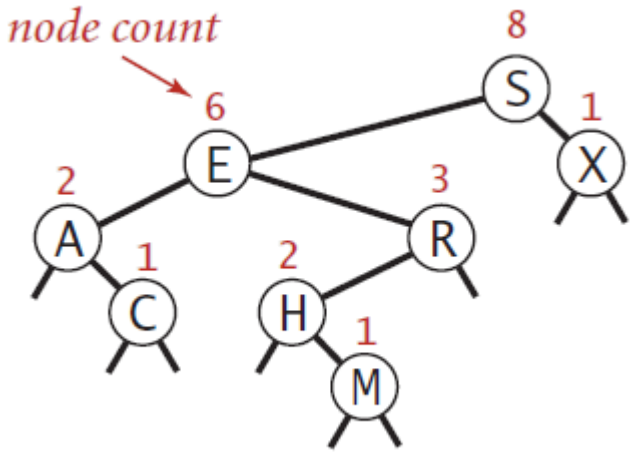
```
public int size()
{ return size(root); }

private int size(Node x)
{
        if (x == null) return 0;
        return x.count;
}
```

```
private Node put(Node x, Key key, Value val) {
        if (x == null) return new Node(key, val, 1);
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x.left = put(x.left, key, val);
        else if (cmp > 0) x.right = put(x.right, key, val);
        else if (cmp == 0) x.val = val;
        x.count = 1 + size(x.left) + size(x.right);
        return x;
}
```

BST rank: Implementación

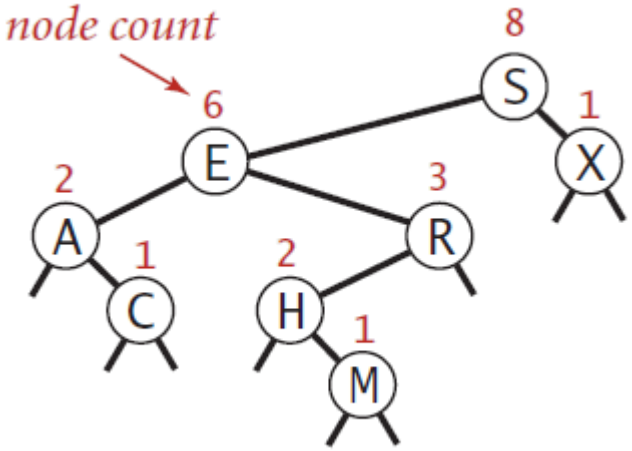
–Cuantas claves son menores que la clave dada por parámetros



BST rank: Implementación

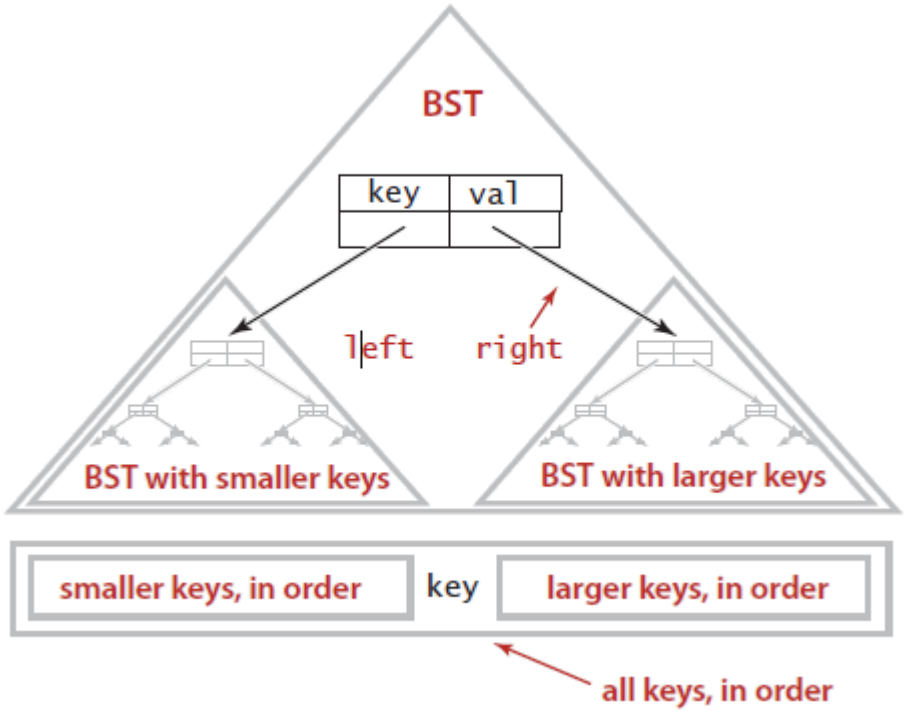
```
public int rank(Key key)
{ return rank(key, root); }


private int rank(Key key, Node x)
{
        if (x == null) return 0;
        int cmp = key.compareTo(x.key);
        if (cmp < 0) return rank(key, x.left);
        else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
        else if (cmp == 0) return size(x.left);
}
```



Iterar claves en orden

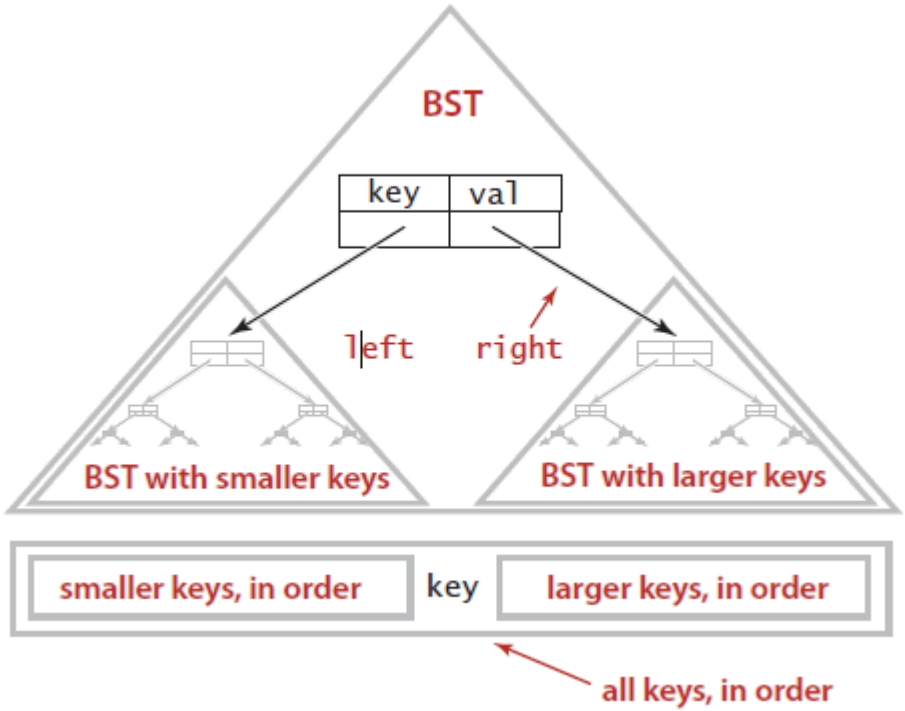
- Añadir llaves a una cola recorriendo en inorden el árbol
  - Recorra recursivamente nodo de la izquierda
  - Visite elemento actual
  - Recorra recursivamente nodo de la derecha



Iterar claves en orden

```
public Iterable<Key> keys()
{
        Queue<Key> q = new Queue<Key>();
        inorder(root, q);
        return q;
}

private void inorder(Node x, Queue<Key> q)
{
        if (x == null) return;
        inorder(x.left, q);
        q.enqueue(x.key);
        inorder(x.right, q);
}
```



Resumen implementaciones ST

implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`

Preguntas?