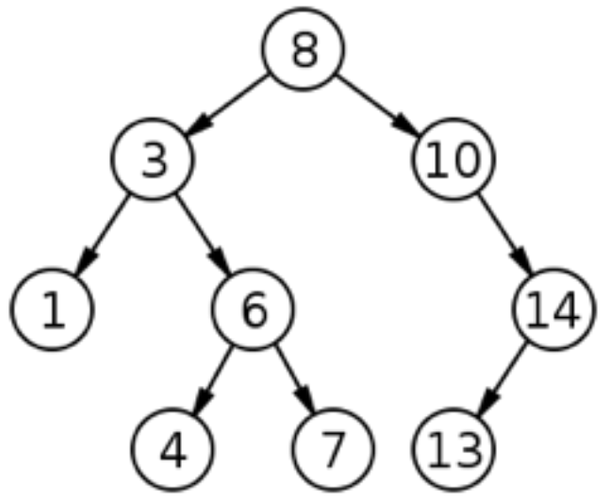
Árboles binarios de búsqueda (BST)

mat-151

Arbol Binario de Búsqueda

- Un árbol binario de búsqueda (Binary Search Tree [BST]) es un árbol binario definido de la siguiente forma:

  - Todo árbol vacío es un árbol binario de búsqueda.
  - Un árbol binario no vacío, de raíz R, es un árbol binario de búsqueda si:
    - En caso de tener subárbol izquierdo, la raíz R debe ser mayor que el valor máximo almacenado en el subárbol izquierdo, y que el subárbol izquierdo sea un árbol binario de búsqueda.
    - En caso de tener subárbol derecho, la raíz R debe ser menor que el valor mínimo almacenado en el subárbol derecho, y que el subárbol derecho sea un árbol binario de búsqueda.



Notacion de psudocódigos

- Dado un nodo x y un arbol T
  - root[T] regresa el nodo raiz del arbol
  - key[x] regresa la llave (contenido) de x
  - left[x] regresa el apuntador al hijo izquierdo de x
  - right[x] regresa el apuntador al hijo derecho de x
  - p[x] regresa el apuntador al padre de x

Operaciones en BST: **búsqueda recursiva**

Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.

Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.

Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

TREE-SEARCH(x, k)

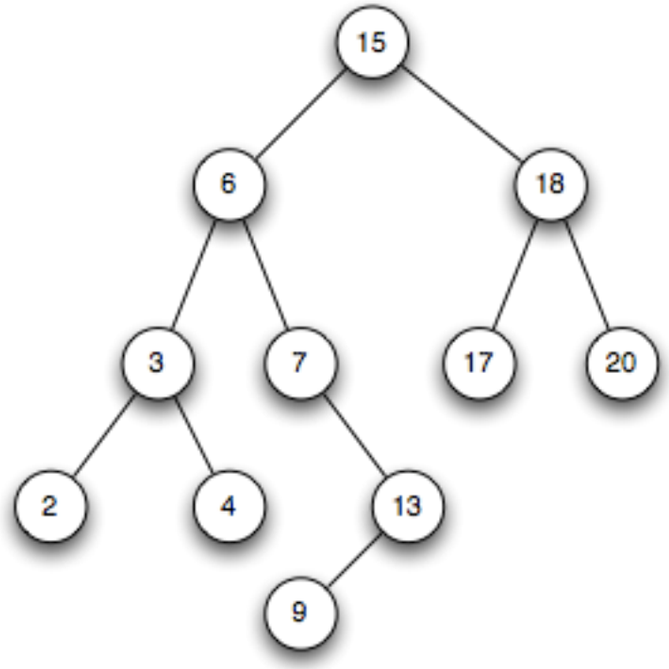
```
1. if x = NULL o k = key[x]
2.       then return x
3. if k < key[x]
4. then return TREE-SEARCH( left[x], k )
5. else return TREE-SEARCH( right[x], k )
```

Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

TREE-SEARCH(x, k)

```
1. if x = NULL o k = key[x]
2.       then return x
3. if k < key[x]
4. then return TREE-SEARCH( left[x], k )
5. else return TREE-SEARCH( right[x], k )
```

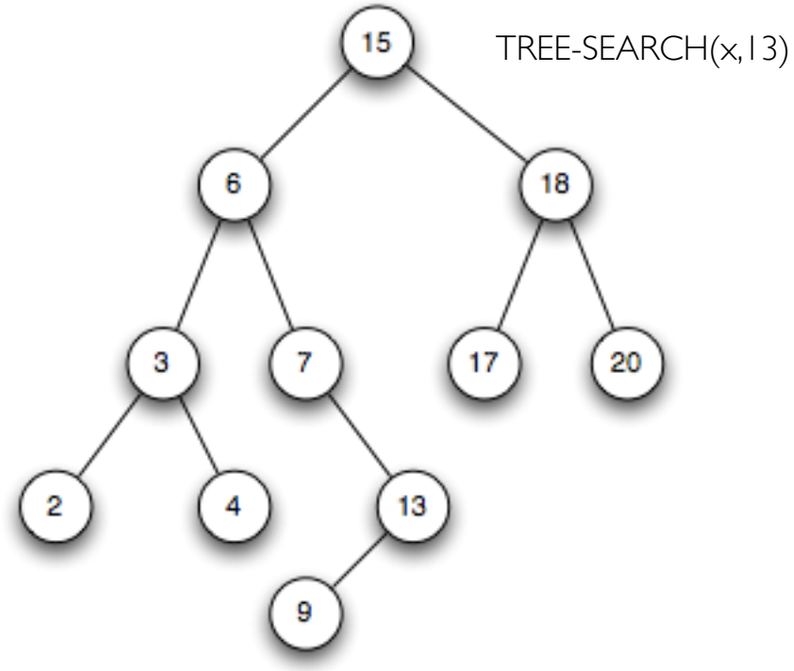


Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

TREE-SEARCH(x, k)

```
1. if x = NULL o k = key[x]
2.       then return x
3. if k < key[x]
4. then return TREE-SEARCH( left[x], k )
5. else return TREE-SEARCH( right[x], k )
```

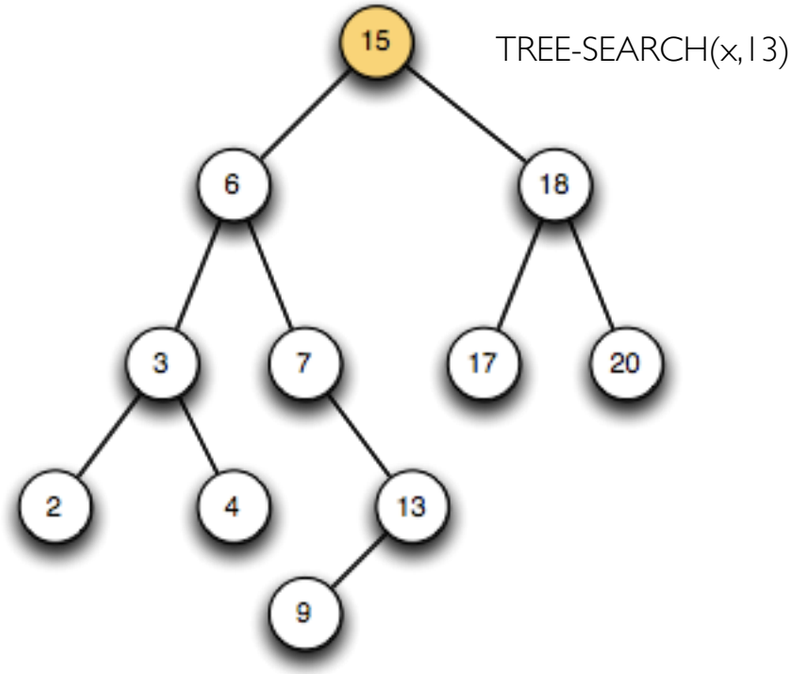


Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

TREE-SEARCH(x, k)

```
1. if x = NULL o k = key[x]
2.       then return x
3. if k < key[x]
4. then return TREE-SEARCH( left[x], k )
5. else return TREE-SEARCH( right[x], k )
```

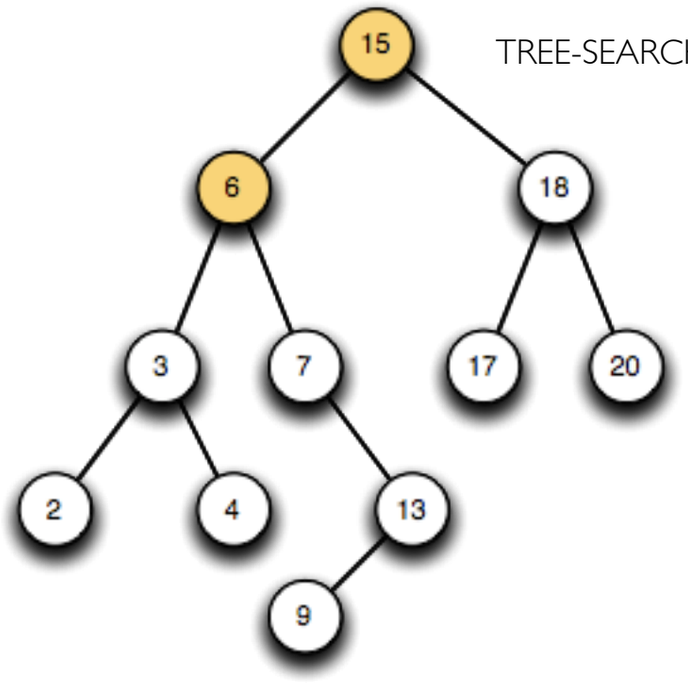


Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

TREE-SEARCH(x, k)

```
1. if x = NULL o k = key[x]
2.       then return x
3. if k < key[x]
4. then return TREE-SEARCH( left[x], k )
5. else return TREE-SEARCH( right[x], k )
```

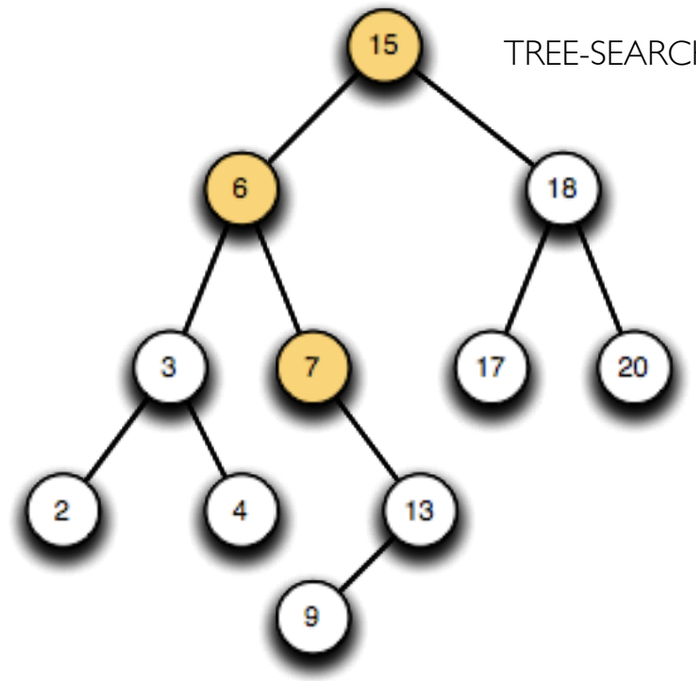


Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

TREE-SEARCH(x, k)

```
1. if x = NULL o k = key[x]
2.       then return x
3. if k < key[x]
4. then return TREE-SEARCH( left[x], k )
5. else return TREE-SEARCH( right[x], k )
```

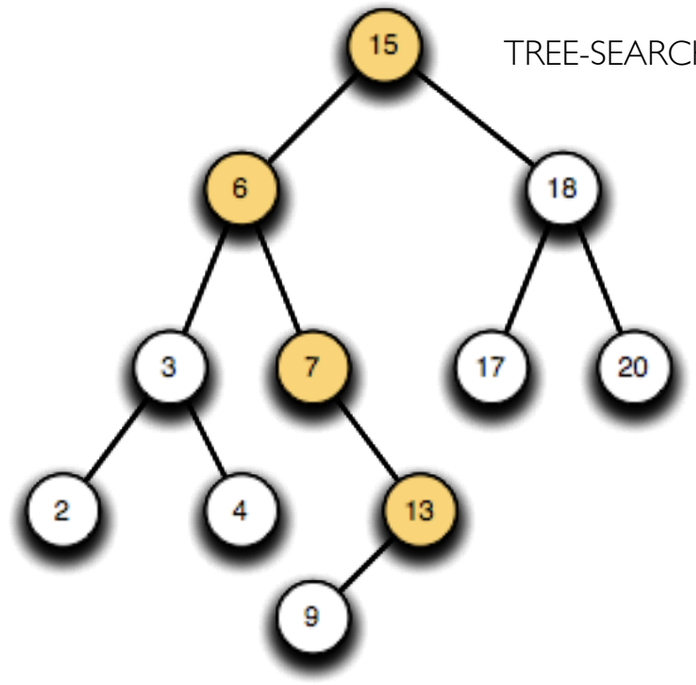


Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

TREE-SEARCH(x, k)

```
1. if x = NULL o k = key[x]
2.       then return x
3. if k < key[x]
4. then return TREE-SEARCH( left[x], k )
5. else return TREE-SEARCH( right[x], k )
```



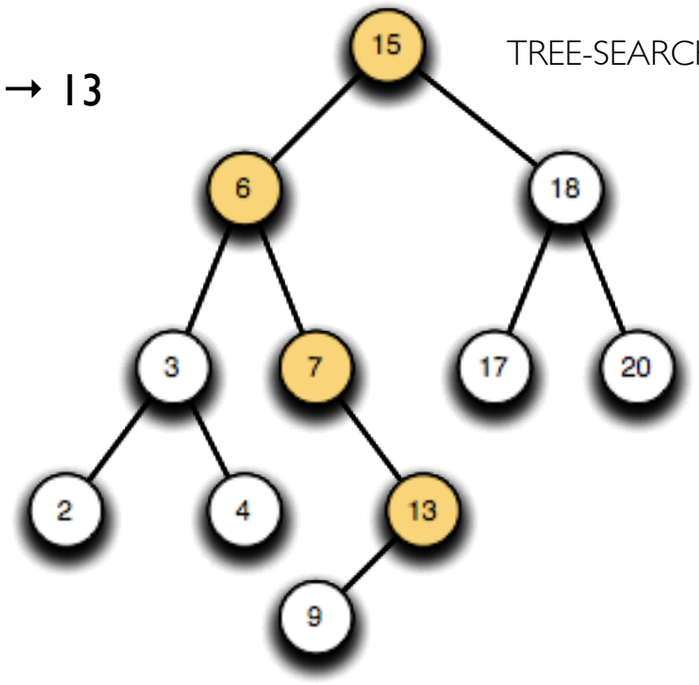
Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

TREE-SEARCH(x, k)

```
1. if x = NULL o k = key[x]
2.       then return x
3. if k < key[x]
4. then return TREE-SEARCH( left[x], k )
5. else return TREE-SEARCH( right[x], k )
```

15 → 6 → 7 → 13



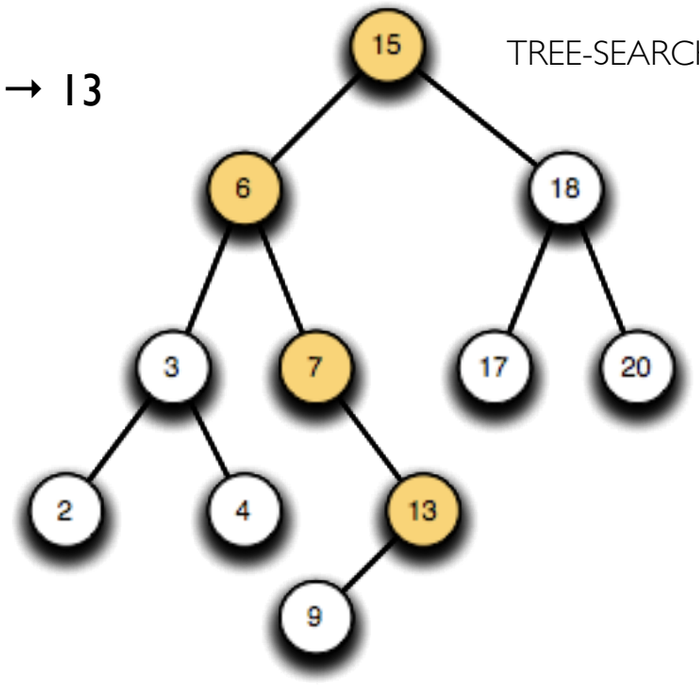
Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

TREE-SEARCH(x, k)

```
1. if x = NULL o k = key[x]
2.       then return x
3. if k < key[x]
4. then return TREE-SEARCH( left[x], k )
5. else return TREE-SEARCH( right[x], k )
```

15 → 6 → 7 → 13



- Tiempo de ejecución de TREE-SEARCH?:

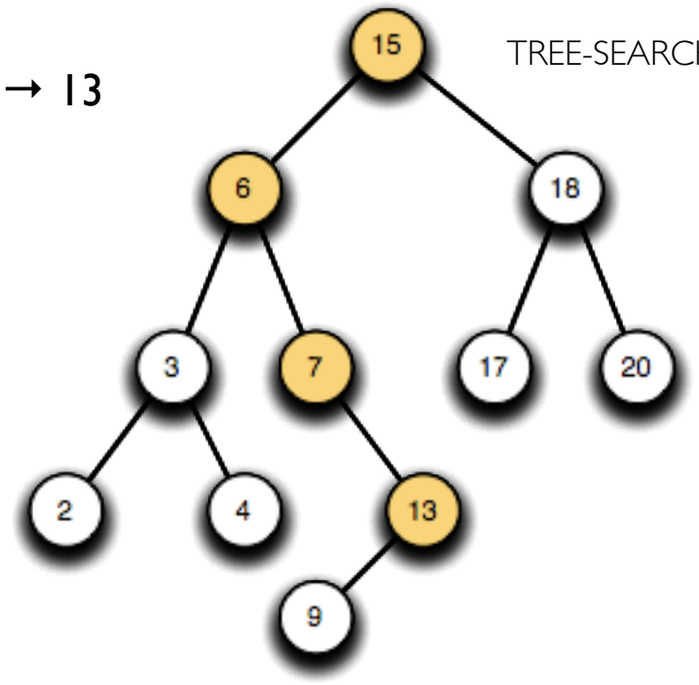
Operaciones en BST: **búsqueda recursiva**

- Dada una llave en un árbol binario de búsqueda, regresar el valor de un nodo.
- Entrada: apuntador a la raíz del árbol y la llave k.
- Salida: apuntador al nodo con la llave k, si existe y si no regresa NULL.

TREE-SEARCH(x, k)

```
1. if x = NULL o k = key[x]
2.       then return x
3. if k < key[x]
4. then return TREE-SEARCH( left[x], k )
5. else return TREE-SEARCH( right[x], k )
```

15 → 6 → 7 → 13



- Tiempo de ejecución de TREE-SEARCH?: O(h) donde h es la altura del árbol.

Operaciones en BST: **búsqueda iterativa**

Operaciones en BST: **búsqueda iterativa**

- Mismo principio “desenrollando” la recursión en un ciclo while.

Operaciones en BST: **búsqueda iterativa**

- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

Operaciones en BST: **búsqueda iterativa**

- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

ITERATIVE-TREE-SEARCH(x, k)

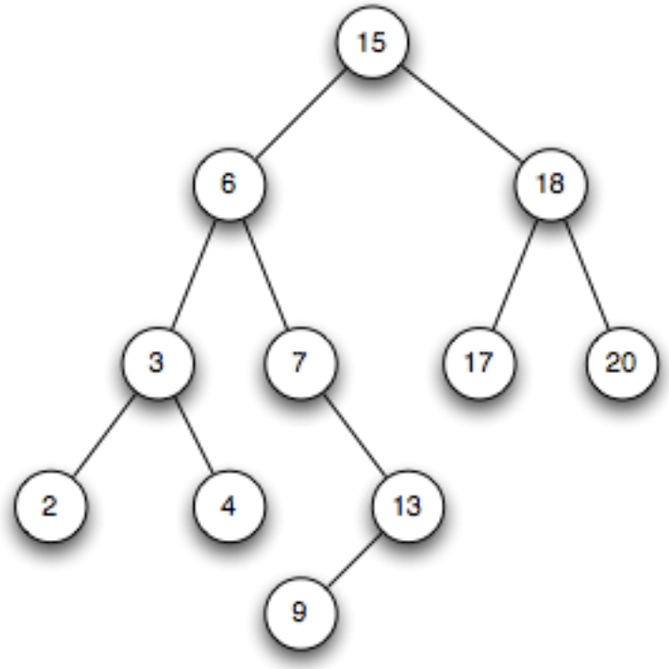
```
1. while x ≠ NULL y k ≠ key[x]
2.       do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5. return x
```

Operaciones en BST: **búsqueda iterativa**

- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

ITERATIVE-TREE-SEARCH(x, k)

```
1. while x ≠ NULL y k ≠ key[x]
2.       do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5. return x
```



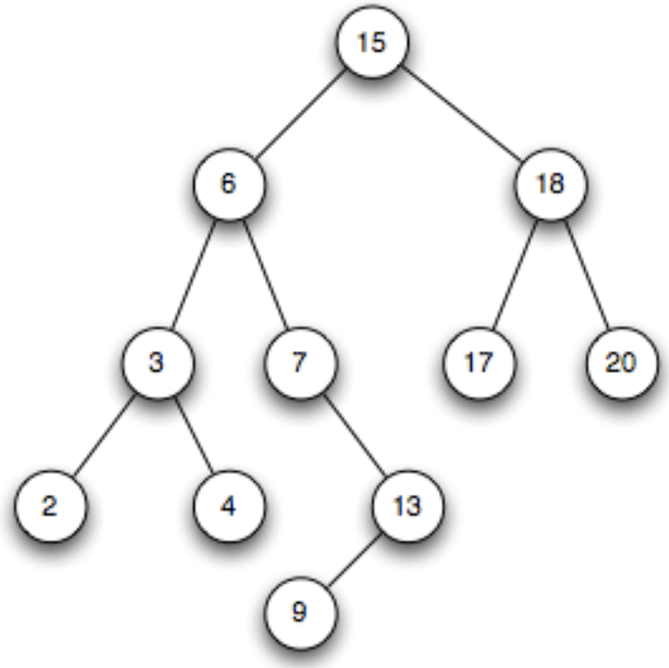
Operaciones en BST: **búsqueda iterativa**

- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

ITERATIVE-TREE-SEARCH(x, k)

```
1. while x ≠ NULL y k ≠ key[x]
2.       do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5. return x
```

ITERATIVE-TREE-SEARCH(x,13)



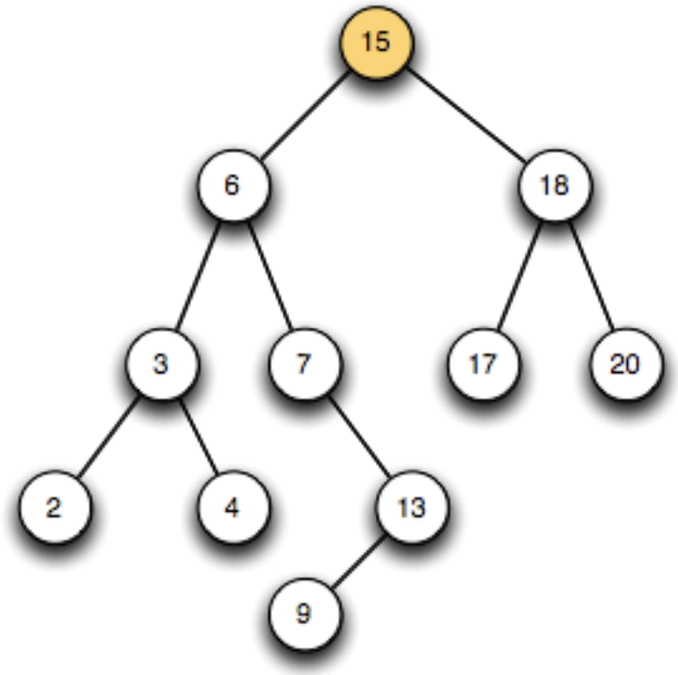
Operaciones en BST: **búsqueda iterativa**

- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

ITERATIVE-TREE-SEARCH(x, k)

```
1. while x ≠ NULL y k ≠ key[x]
2.       do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5. return x
```

ITERATIVE-TREE-SEARCH(x,13)



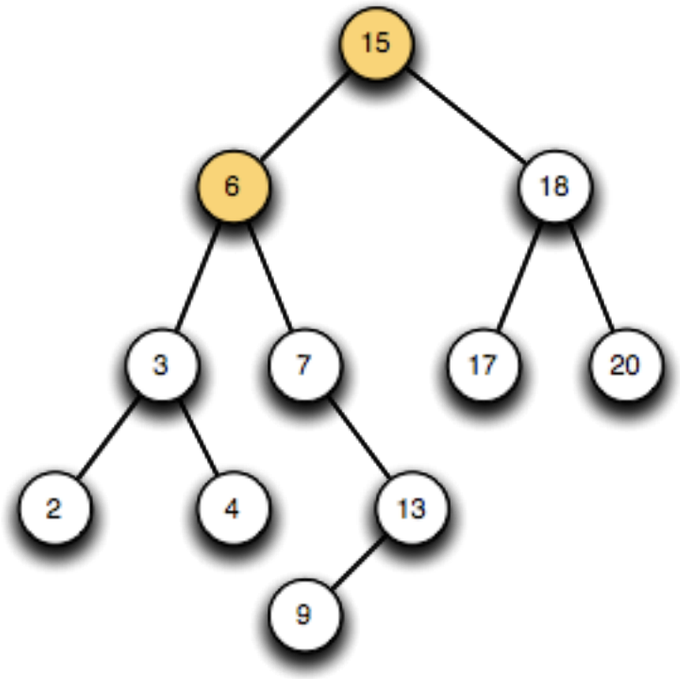
Operaciones en BST: **búsqueda iterativa**

- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

ITERATIVE-TREE-SEARCH(x, k)

```
1. while x ≠ NULL y k ≠ key[x]
2.       do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5. return x
```

ITERATIVE-TREE-SEARCH(x,13)



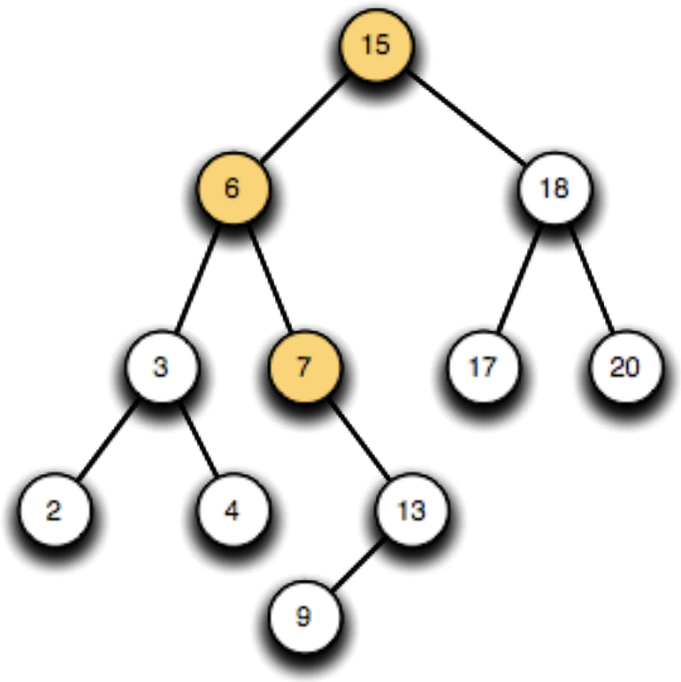
Operaciones en BST: **búsqueda iterativa**

- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

ITERATIVE-TREE-SEARCH(x, k)

```
1. while x ≠ NULL y k ≠ key[x]
2.       do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5. return x
```

ITERATIVE-TREE-SEARCH(x,13)



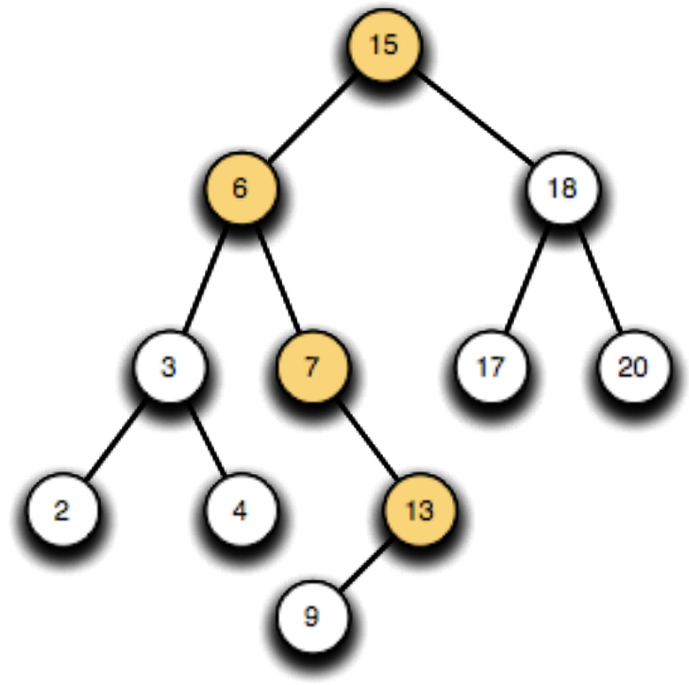
Operaciones en BST: **búsqueda iterativa**

- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

ITERATIVE-TREE-SEARCH(x, k)

```
1. while x ≠ NULL y k ≠ key[x]
2.       do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5. return x
```

ITERATIVE-TREE-SEARCH(x,13)



Operaciones en BST: **búsqueda iterativa**

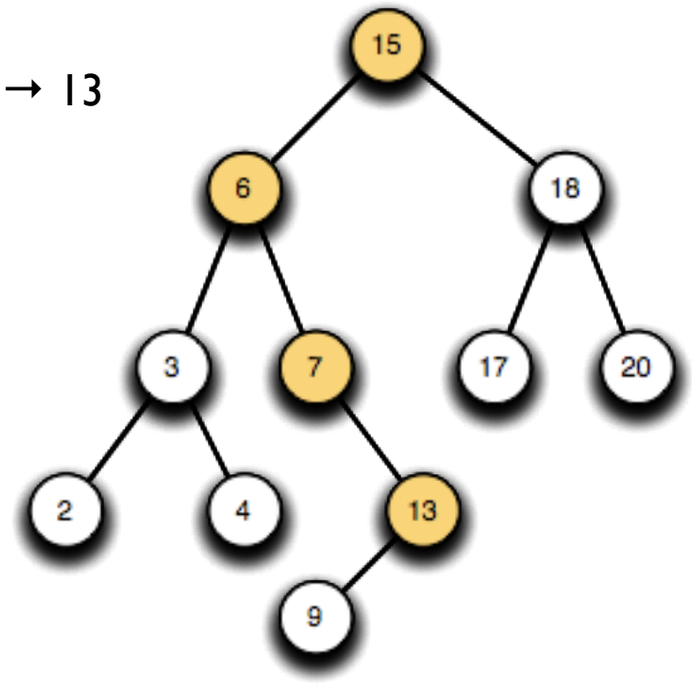
- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

ITERATIVE-TREE-SEARCH(x, k)

```
1. while x ≠ NULL y k ≠ key[x]
2.       do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5. return x
```

ITERATIVE-TREE-SEARCH(x,13)

15 → 6 → 7 → 13



Operaciones en BST: **búsqueda iterativa**

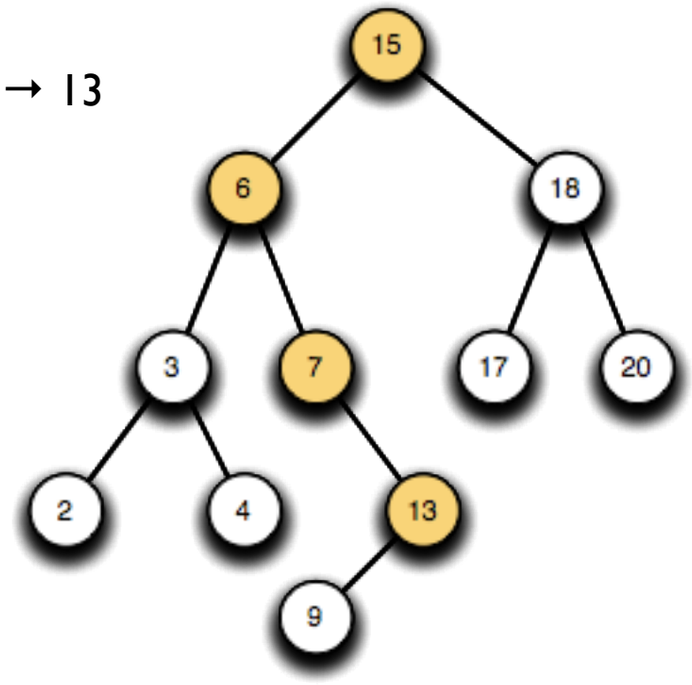
- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

ITERATIVE-TREE-SEARCH(x, k)

```
1. while x ≠ NULL y k ≠ key[x]
2.       do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5. return x
```

ITERATIVE-TREE-SEARCH(x,13)

15 → 6 → 7 → 13



- Tiempo de ejecución de ITERATIVE-TREE-SEARCH?:

Operaciones en BST: **búsqueda iterativa**

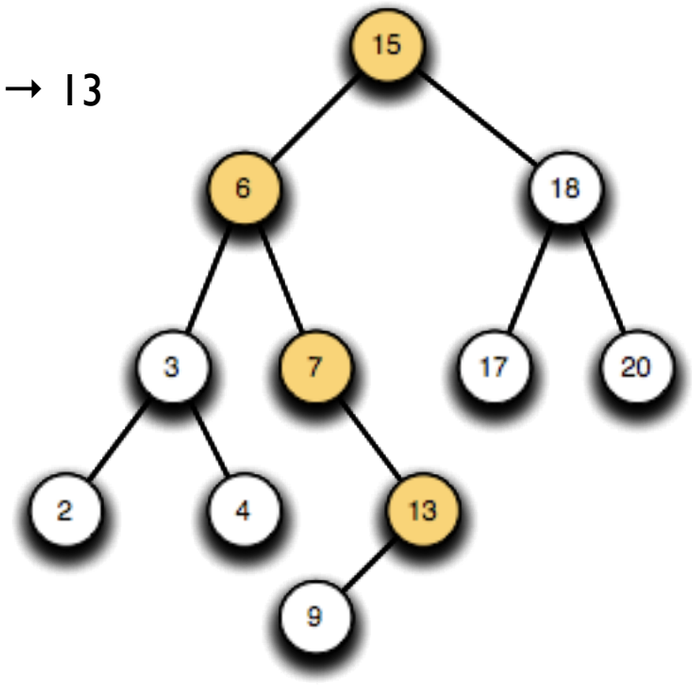
- Mismo principio “desenrollando” la recursión en un ciclo while.
- En la mayoría de las computadoras esta versión es más eficiente.

ITERATIVE-TREE-SEARCH(x, k)

```
1. while x ≠ NULL y k ≠ key[x]
2.       do if k < key[x]
3.             then x ← left[x]
4.             else x ← right[x]
5. return x
```

ITERATIVE-TREE-SEARCH(x,13)

15 → 6 → 7 → 13



- Tiempo de ejecución de ITERATIVE-TREE-SEARCH?: O(h)

Operaciones en BST: **mínimo y máximo**

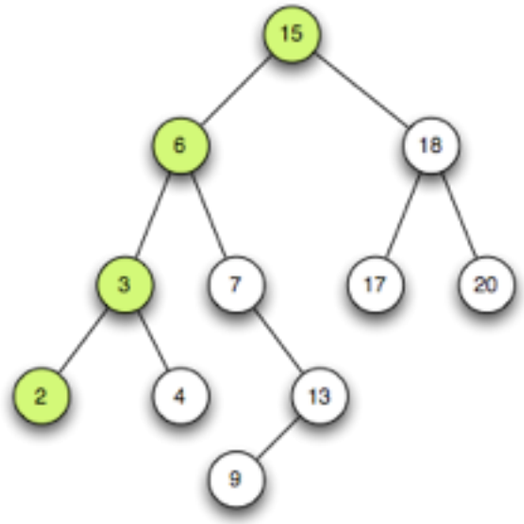
- Se puede encontrar el **elemento mínimo** de un BST siguiendo los **hijos izquierdos** a partir de la raíz hasta encontrar NULL.

Operaciones en BST: **mínimo y máximo**

- Se puede encontrar el elemento mínimo de un BST siguiendo los hijos izquierdos a partir de la raíz hasta encontrar NULL.

TREE-MINIMUM(x)

```
1. while left[x] ≠ NULL
2.       do x ← left[x]
3. return x
```

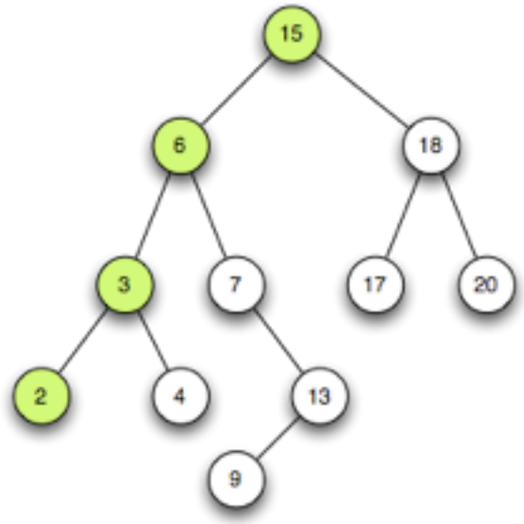


Operaciones en BST: **mínimo y máximo**

- Se puede encontrar el elemento mínimo de un BST siguiendo los hijos izquierdos a partir de la raíz hasta encontrar NULL.

TREE-MINIMUM(x)

```
1. while left[x] ≠ NULL
2.       do x ← left[x]
3. return x
```



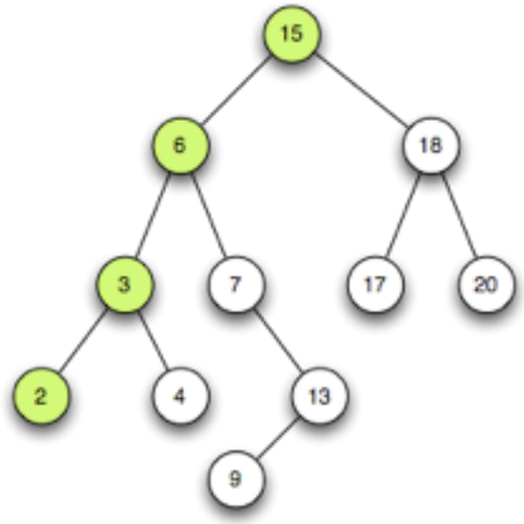
- Simétricamente se puede encontrar el elemento máximo de un BST siguiendo los hijos derechos a partir de la raíz hasta encontrar NULL.

Operaciones en BST: **mínimo y máximo**

- Se puede encontrar el elemento mínimo de un BST siguiendo los hijos izquierdos a partir de la raíz hasta encontrar NULL.

TREE-MINIMUM(x)

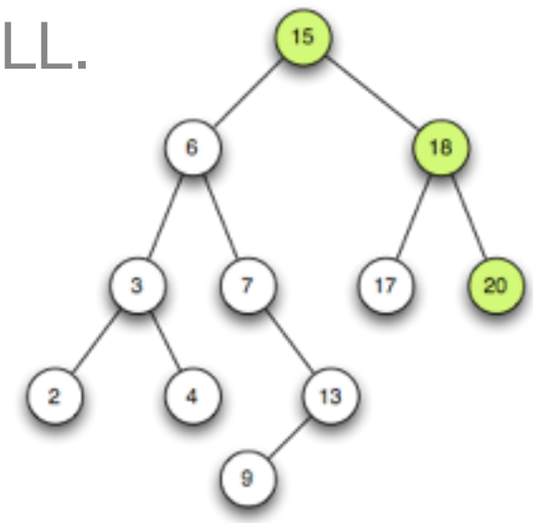
```
1. while left[x] ≠ NULL
2.       do x ← left[x]
3. return x
```



- Simétricamente se puede encontrar el elemento máximo de un BST siguiendo los hijos derechos a partir de la raíz hasta encontrar NULL.

TREE-MAXIMUM(x)

```
1. while right[x] ≠ NULL
2.       do x ← right[x]
3. return x
```

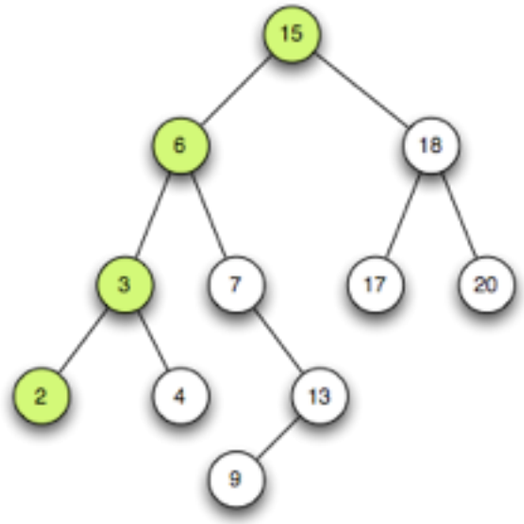


Operaciones en BST: **mínimo y máximo**

- Se puede encontrar el elemento mínimo de un BST siguiendo los hijos izquierdos a partir de la raíz hasta encontrar NULL.

TREE-MINIMUM(x)

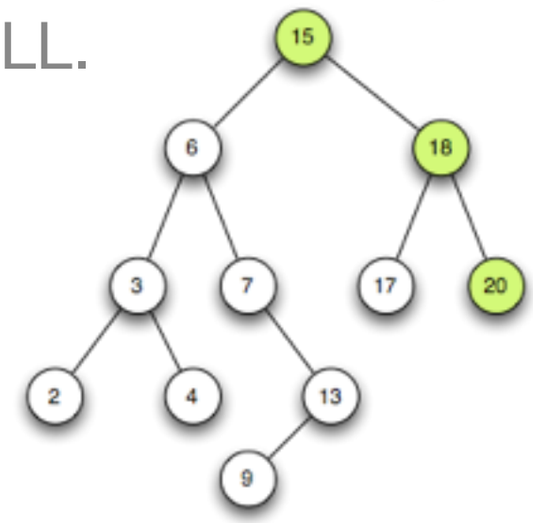
```
1. while left[x] ≠ NULL
2.       do x ← left[x]
3. return x
```



- Simétricamente se puede encontrar el elemento máximo de un BST siguiendo los hijos derechos a partir de la raíz hasta encontrar NULL.

TREE-MAXIMUM(x)

```
1. while right[x] ≠ NULL
2.       do x ← right[x]
3. return x
```



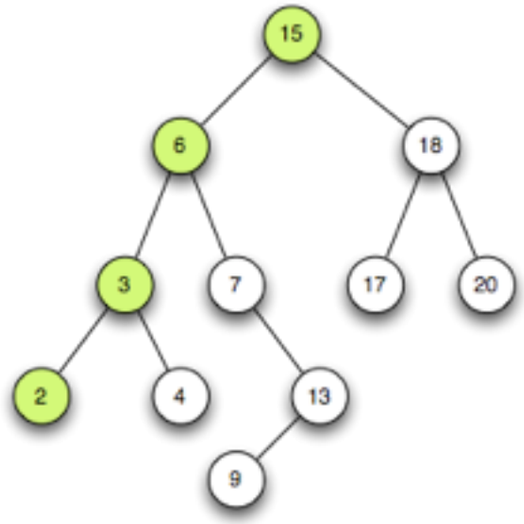
- Tiempo de ejecución?:

Operaciones en BST: **mínimo y máximo**

- Se puede encontrar el elemento mínimo de un BST siguiendo los hijos izquierdos a partir de la raíz hasta encontrar NULL.

TREE-MINIMUM(x)

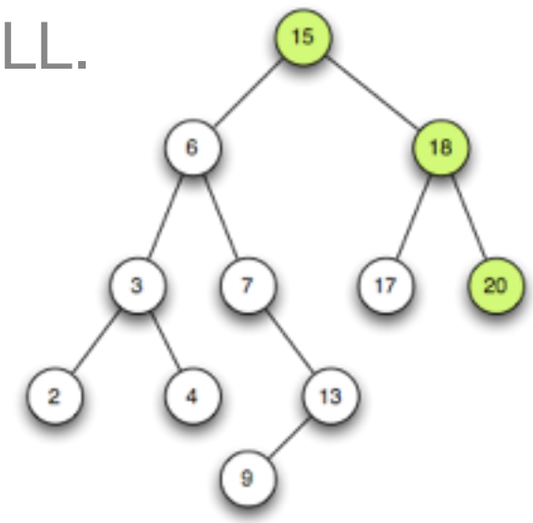
```
1. while left[x] ≠ NULL
2.       do x ← left[x]
3. return x
```



- Simétricamente se puede encontrar el elemento máximo de un BST siguiendo los hijos derechos a partir de la raíz hasta encontrar NULL.

TREE-MAXIMUM(x)

```
1. while right[x] ≠ NULL
2.       do x ← right[x]
3. return x
```



- Tiempo de ejecución?:O(h)

Operaciones en BST: **sucesor**

Operaciones en BST: **sucesor**

- Dado un nodo en un BST, encontrar su sucesor en el orden determinado por un recorrido en orden.

Operaciones en BST: **sucesor**

- Dado un nodo en un BST, encontrar su sucesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el sucesor de un nodo x es el nodo con la llave más pequeña mayor que key[x].

Operaciones en BST: **sucesor**

- Dado un nodo en un BST, encontrar su sucesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el sucesor de un nodo x es el nodo con la llave más pequeña mayor que key[x].

TREE-SUCCESSOR(x)

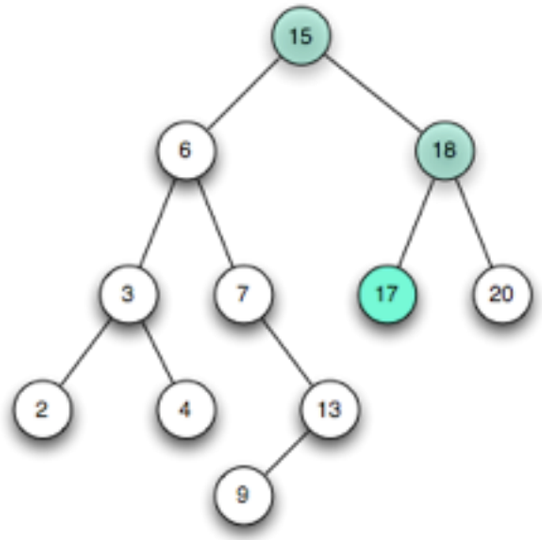
```
1. if right[x] ≠ NULL
2.       then return TREE-MINIMUM(right[x])
3. y ← p[x]
4. while y ≠ NULL y x=right[y]
5.       do x ← y
6.             y ← p[y]
7. return y
```

Operaciones en BST: **sucesor**

- Dado un nodo en un BST, encontrar su sucesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el sucesor de un nodo x es el nodo con la llave más pequeña mayor que key[x].

TREE-SUCCESSOR(x)

```
1. if right[x] ≠ NULL
2.       then return TREE-MINIMUM(right[x])
3. y ← p[x]
4. while y ≠ NULL y x=right[y]
5.       do x ← y
6.             y ← p[y]
7. return y
```



caso 1: sub-árbol derecho diferente a NULL, minimo del sub-árbol derecho.

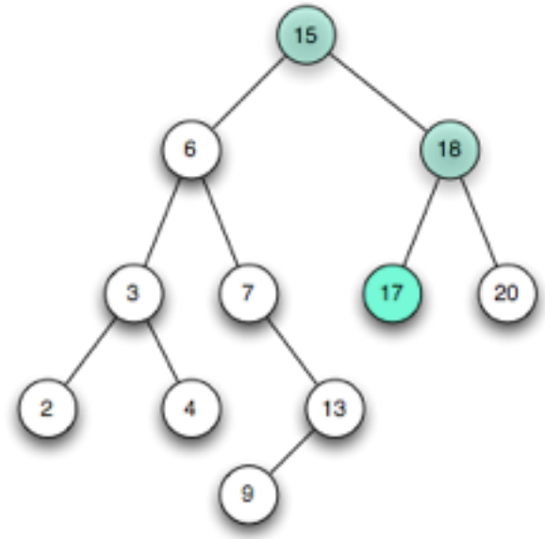
TREE-SUCCESSOR(15)

Operaciones en BST: **sucesor**

- Dado un nodo en un BST, encontrar su sucesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el sucesor de un nodo x es el nodo con la llave más pequeña mayor que key[x].

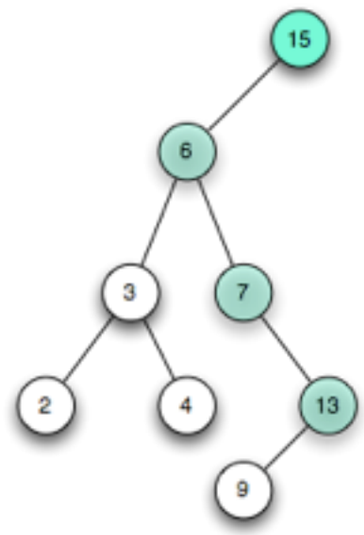
TREE-SUCCESSOR(x)

```
1. if right[x] ≠ NULL
2.       then return TREE-MINIMUM(right[x])
3. y ← p[x]
4. while y ≠ NULL y x=right[y]
5.       do x ← y
6.             y ← p[y]
7. return y
```



caso 1: sub-árbol derecho diferente a NULL, minimo del sub-árbol derecho.

TREE-SUCCESSOR(15)



caso 2: sub-árbol derecho igual a NULL, subir hasta encontrar un nodo que sea el hijo izquierdo de su padre, el padre es el sucesor.

TREE-SUCCESSOR(13)

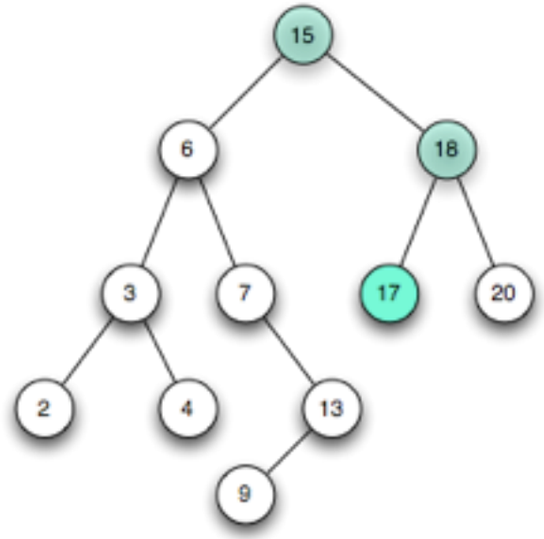
Operaciones en BST: **sucesor**

- Dado un nodo en un BST, encontrar su sucesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el sucesor de un nodo x es el nodo con la llave más pequeña mayor que key[x].

TREE-SUCCESSOR(x)

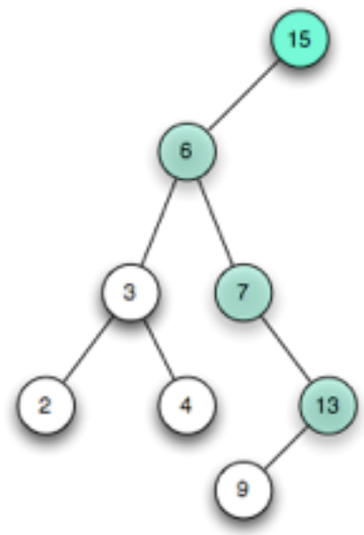
```
1. if right[x] ≠ NULL
2.       then return TREE-MINIMUM(right[x])
3. y ← p[x]
4. while y ≠ NULL y x=right[y]
5.       do x ← y
6.             y ← p[y]
7. return y
```

- Tiempo de ejecución?:



caso 1: sub-árbol derecho diferente a NULL, minimo del sub-árbol derecho.

TREE-SUCCESSOR(15)



caso 2: sub-árbol derecho igual a NULL, subir hasta encontrar un nodo que sea el hijo izquierdo de su padre, el padre es el sucesor.

TREE-SUCCESSOR(13)

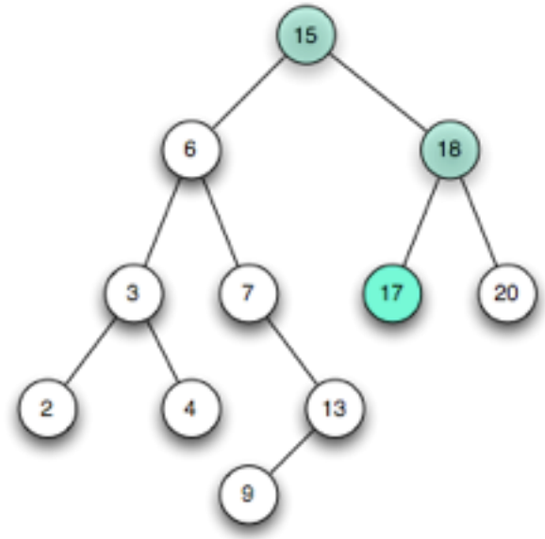
Operaciones en BST: **sucesor**

- Dado un nodo en un BST, encontrar su sucesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el sucesor de un nodo x es el nodo con la llave más pequeña mayor que key[x].

TREE-SUCCESSOR(x)

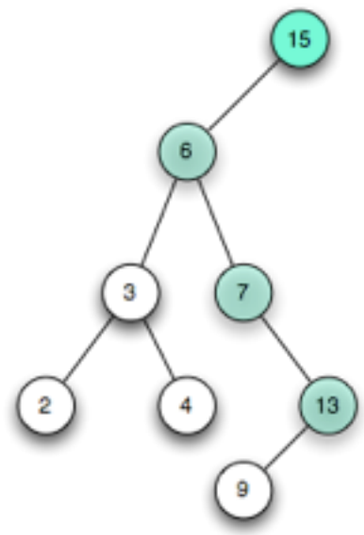
```
1. if right[x] ≠ NULL
2.       then return TREE-MINIMUM(right[x])
3. y ← p[x]
4. while y ≠ NULL y x=right[y]
5.       do x ← y
6.             y ← p[y]
7. return y
```

- Tiempo de ejecución?:O(h)



caso 1: sub-árbol derecho diferente a NULL, minimo del sub-árbol derecho.

TREE-SUCCESSOR(15)



caso 2: sub-árbol derecho igual a NULL, subir hasta encontrar un nodo que sea el hijo izquierdo de su padre, el padre es el sucesor.

TREE-SUCCESSOR(13)

Operaciones en BST: **antecesor**

Operaciones en BST: **antecesor**

- Simétricamente, dado un nodo en un BST, encontrar su antecesor en el orden determinado por un recorrido en orden.

Operaciones en BST: **antecesor**

- Simétricamente, dado un nodo en un BST, encontrar su antecesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el antecesor de un nodo x es el nodo con la llave más grande menor que key[x].

Operaciones en BST: **antecesor**

- Simétricamente, dado un nodo en un BST, encontrar su antecesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el antecesor de un nodo x es el nodo con la llave más grande menor que key[x].

TREE-PREDECESSOR(x)

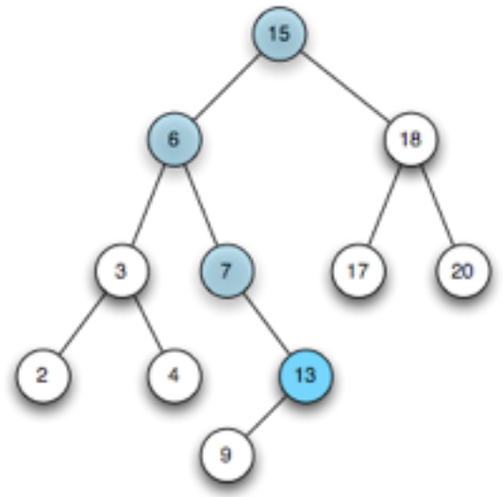
```
1. if left[x] ≠ NULL
2.       then return TREE-MAXIMUM(left[x])
3. y ← p[x]
4. while y ≠ NULL y x=left[y]
5.       do x ← y
6.             y ← p[y]
7. return y
```

Operaciones en BST: **antecesor**

- Simétricamente, dado un nodo en un BST, encontrar su antecesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el antecesor de un nodo x es el nodo con la llave más grande menor que key[x].

TREE-PREDECESSOR(x)

```
1. if left[x] ≠ NULL
2.       then return TREE-MAXIMUM(left[x])
3. y ← p[x]
4. while y ≠ NULL y x=left[y]
5.       do x ← y
6.             y ← p[y]
7. return y
```



caso 1: sub-árbol izquierdo diferente a NULL, máximo del sub-árbol izquierdo.

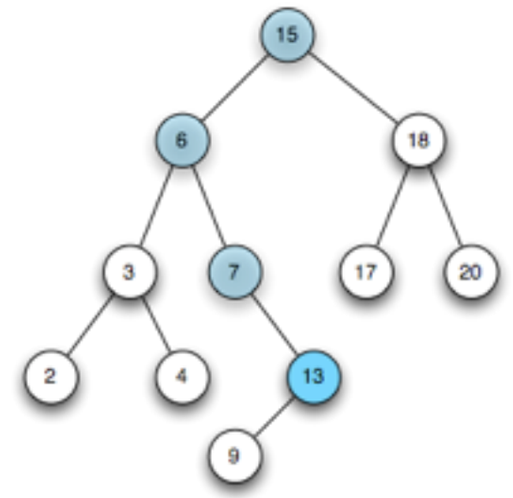
TREE-PREDECESSOR(15)

Operaciones en BST: **antecesor**

- Simétricamente, dado un nodo en un BST, encontrar su antecesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el antecesor de un nodo x es el nodo con la llave más grande menor que key[x].

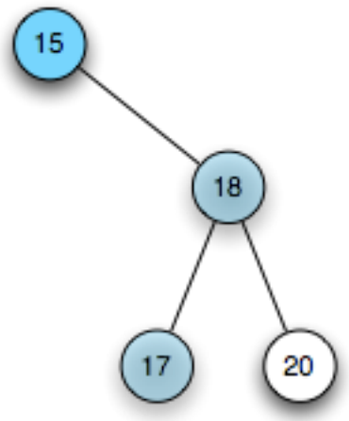
TREE-PREDECESSOR(x)

```
1. if left[x] ≠ NULL
2.       then return TREE-MAXIMUM(left[x])
3. y ← p[x]
4. while y ≠ NULL y x=left[y]
5.       do x ← y
6.             y ← p[y]
7. return y
```



caso 1: sub-árbol izquierdo diferente a NULL, máximo del sub-árbol izquierdo.

TREE-PREDECESSOR(15)



caso 2: sub-árbol izquierdo igual a NULL, subir hasta encontrar un nodo que sea el hijo derecho de su padre. El padre es el antecesor.

TREE-PREDECESSOR(17)

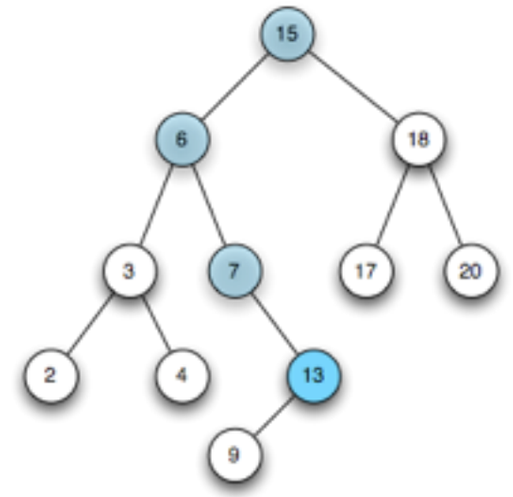
Operaciones en BST: **antecesor**

- Simétricamente, dado un nodo en un BST, encontrar su antecesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el antecesor de un nodo x es el nodo con la llave más grande menor que key[x].

TREE-PREDECESSOR(x)

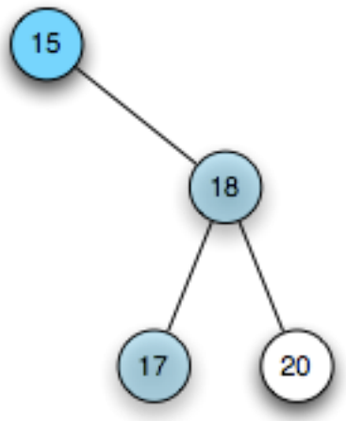
```
1. if left[x] ≠ NULL
2.       then return TREE-MAXIMUM(left[x])
3. y ← p[x]
4. while y ≠ NULL y x=left[y]
5.       do x ← y
6.             y ← p[y]
7. return y
```

- Tiempo de ejecución?:



caso 1: sub-árbol izquierdo diferente a NULL, máximo del sub-árbol izquierdo.

TREE-PREDECESSOR(15)



caso 2: sub-árbol izquierdo igual a NULL, subir hasta encontrar un nodo que sea el hijo derecho de su padre. El padre es el antecesor.

TREE-PREDECESSOR(17)

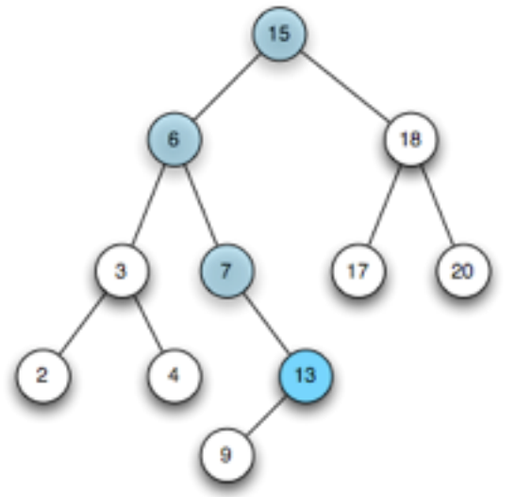
Operaciones en BST: **antecesor**

- Simétricamente, dado un nodo en un BST, encontrar su antecesor en el orden determinado por un recorrido en orden.
- Si todas las llaves son diferentes, el antecesor de un nodo x es el nodo con la llave más grande menor que key[x].

TREE-PREDECESSOR(x)

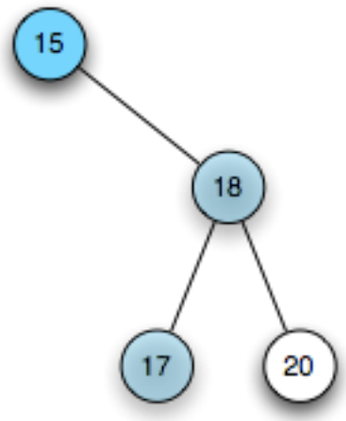
```
1. if left[x] ≠ NULL
2.       then return TREE-MAXIMUM(left[x])
3. y ← p[x]
4. while y ≠ NULL y x=left[y]
5.       do x ← y
6.             y ← p[y]
7. return y
```

- Tiempo de ejecución?:O(h)



caso 1: sub-árbol izquierdo diferente a NULL, máximo del sub-árbol izquierdo.

TREE-PREDECESSOR(15)



caso 2: sub-árbol izquierdo igual a NULL, subir hasta encontrar un nodo que sea el hijo derecho de su padre. El padre es el antecesor.

TREE-PREDECESSOR(17)

Operaciones en BST

Operaciones en BST

- En resumen:

Operaciones en BST

- En resumen:
  - las operaciones de **búsqueda**, encontrar **mínimo**, encontrar **máximo**, encontrar el **sucesor** de un nodo y encontrar el **antecesor** de un nodo pueden implementarse de tal manera que se ejecuten en un **tiempo O(h)** en un árbol binario de búsqueda de **altura h**.

Operaciones en BST

- En resumen:
  - las operaciones de **búsqueda**, encontrar **mínimo**, encontrar **máximo**, encontrar el **sucesor** de un nodo y encontrar el **antecesor** de un nodo pueden implementarse de tal manera que se ejecuten en un **tiempo O(h)** en un árbol binario de búsqueda de **altura h**.

Operaciones en BST

- En resumen:
  - las operaciones de **búsqueda**, encontrar **mínimo**, encontrar **máximo**, encontrar el **sucesor** de un nodo y encontrar el **antecesor** de un nodo pueden implementarse de tal manera que se ejecuten en un **tiempo O(h)** en un árbol binario de búsqueda de **altura h**.

- Las operaciones de inserción y eliminación causan una modificación del árbol binario de búsqueda.

Operaciones en BST

- En resumen:
  - las operaciones de **búsqueda**, encontrar **mínimo**, encontrar **máximo**, encontrar el **sucesor** de un nodo y encontrar el **antecesor** de un nodo pueden implementarse de tal manera que se ejecuten en un **tiempo O(h)** en un árbol binario de búsqueda de **altura h**.

- Las operaciones de inserción y eliminación causan una modificación del árbol binario de búsqueda.
- El árbol debe variar de tal manera que su propiedad de orden se preserve (recordar montículos)

Operaciones en BST: **inserción**

Operaciones en BST: **inserción**

- Dado un árbol binario de búsqueda T insertar un nuevo valor v.

Operaciones en BST: **inserción**

- Dado un árbol binario de búsqueda T insertar un nuevo valor v.
- Se da como entrada un nuevo nodo z para el cual:

Operaciones en BST: **inserción**

- Dado un árbol binario de búsqueda T insertar un nuevo valor v.
- Se da como entrada un nuevo nodo z para el cual:
  - key[z] = v,

Operaciones en BST: **inserción**

- Dado un árbol binario de búsqueda T insertar un nuevo valor v.
- Se da como entrada un nuevo nodo z para el cual:
  - key[z] = v,
  - left[z]=NULL,

Operaciones en BST: **inserción**

- Dado un árbol binario de búsqueda T insertar un nuevo valor v.
- Se da como entrada un nuevo nodo z para el cual:
  - key[z] = v,
  - left[z]=NULL,
  - right[z]=NULL,

Operaciones en BST: **inserción**

- Dado un árbol binario de búsqueda T insertar un nuevo valor v.
- Se da como entrada un nuevo nodo z para el cual:
  - key[z] = v,
  - left[z]=NULL,
  - right[z]=NULL,
  - p[z] = NULL.

Operaciones en BST: **inserción**

- Dado un árbol binario de búsqueda T insertar un nuevo valor v.
- Se da como entrada un nuevo nodo z para el cual:
  - key[z] = v,
  - left[z]=NULL,
  - right[z]=NULL,
  - p[z] = NULL.
- Buscamos insertar z en la posición apropiada del árbol.

Operaciones en BST: **inserción**

- Dado un árbol binario de búsqueda T insertar un nuevo valor v.
- Se da como entrada un nuevo nodo z para el cual:
  - key[z] = v,
  - left[z]=NULL,
  - right[z]=NULL,
  - p[z] = NULL.
- Buscamos insertar z en la posición apropiada del árbol.

TREE-INSERT(T, z)

```
•  y ← NULL
•  x ← root[ T ]
1.  while x ≠ NULL
2.        do y ← x
3.           if key[z] < key[x]
4.              then x ← left[x]
5.              else x ← right[x]
•  p[z] ← y
•  if y = NULL
•     then root[ T ] ← z          —— El árbol T estaba vacío
•     else if key[z] < key[y]
•          then left[y] ← z
•          else right[y] ← z
```

Operaciones en BST: **inserción**

TREE-INSERT(T, z)

- y ← NULL
- x ← root[T]

1. while x ≠ NULL
2. do y ← x
3. if key[z] < key[x]
4. then x ← left[x]
5. else x ← right[x]

- p[z] ← y
- if y = NULL
- then root[T] ← z
- else if key[z] < key[y]
- then left[y] ← z
- else right[y] ← z

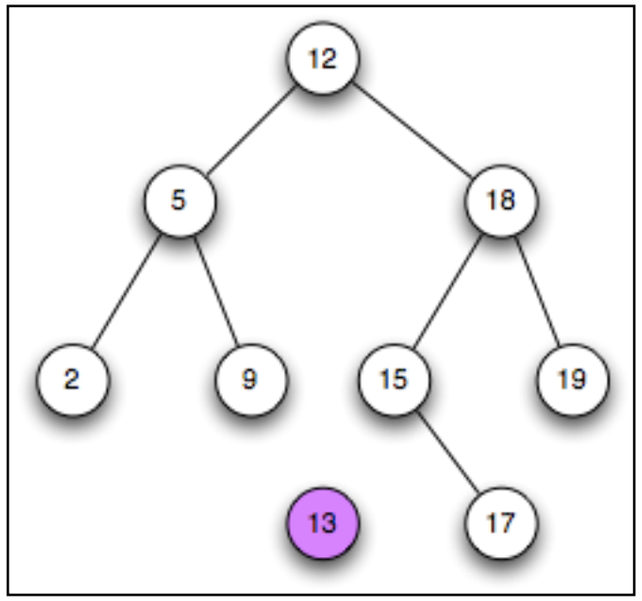
Operaciones en BST: **inserción**

TREE-INSERT(T,13)

TREE-INSERT(T, z)

```
•  y ← NULL
•  x ← root[ T ]
1.  while x ≠ NULL
2.        do y ← x
3.           if key[z] < key[x]
4.              then x ← left[x]
5.              else x ← right[x]
•  p[z] ← y
•  if y = NULL
•     then root[ T ] ← z
•     else if key[z] < key[y]
•        then left[y] ← z
•        else right[y] ← z
```

Operaciones en BST: **inserción**



TREE-INSERT(T,13)

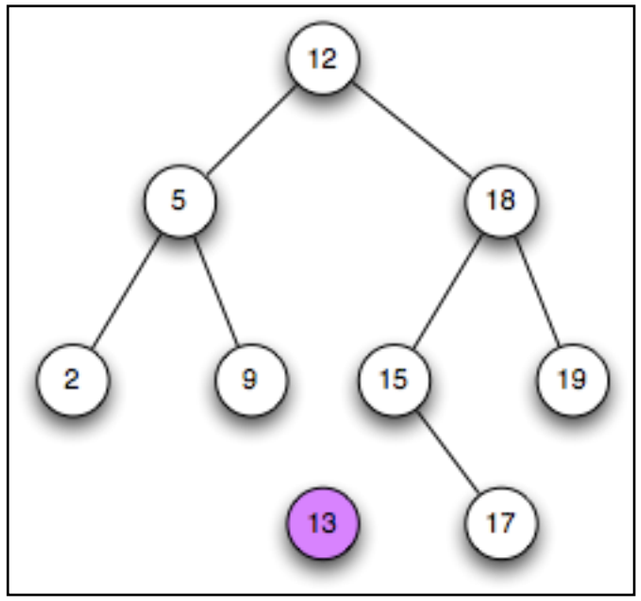
TREE-INSERT(T, z)

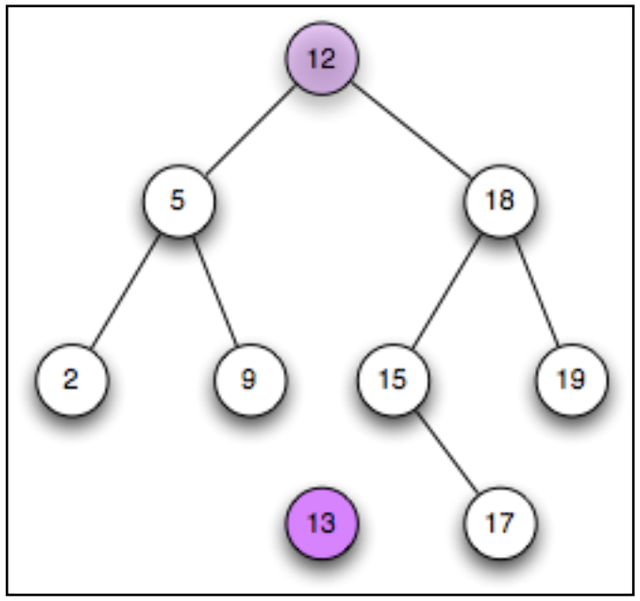
- y ← NULL
- x ← root[T]

1. while x ≠ NULL
2. do y ← x
3. if key[z] < key[x]
4. then x ← left[x]
5. else x ← right[x]

- p[z] ← y
- if y = NULL
- then root[T] ← z
- else if key[z] < key[y]
- then left[y] ← z
- else right[y] ← z

Operaciones en BST: **inserción**





TREE-INSERT(T,13)

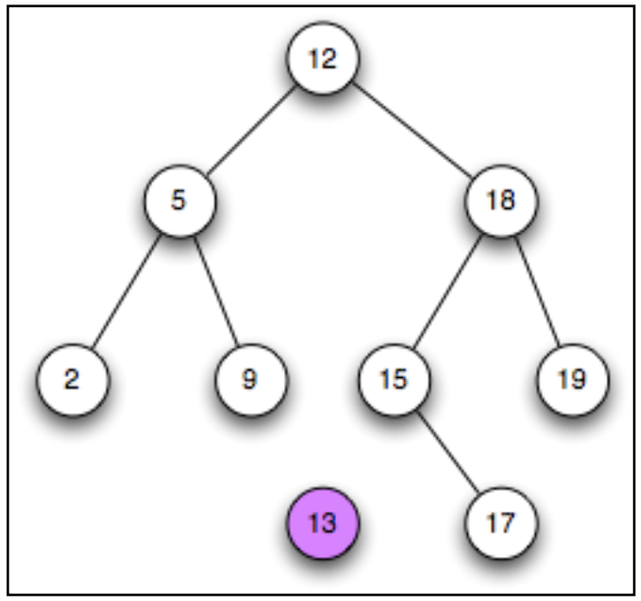
TREE-INSERT(T, z)

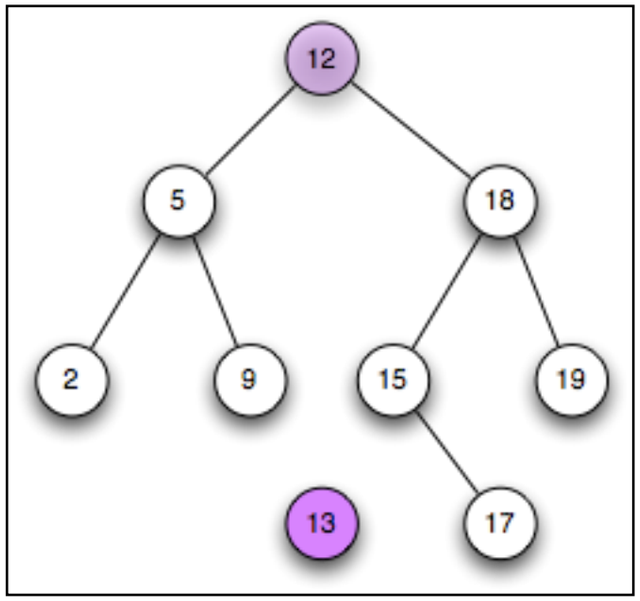
- y ← NULL
- x ← root[T]

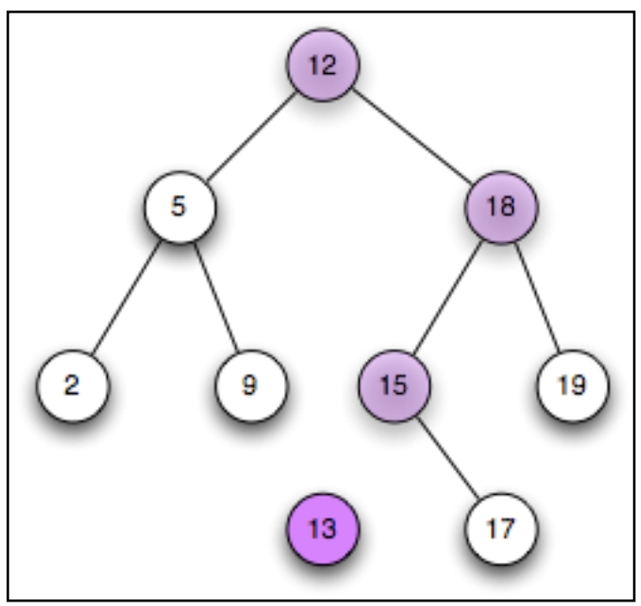
1. while x ≠ NULL
2. do y ← x
3. if key[z] < key[x]
4. then x ← left[x]
5. else x ← right[x]

- p[z] ← y
- if y = NULL
- then root[T] ← z
- else if key[z] < key[y]
- then left[y] ← z
- else right[y] ← z

Operaciones en BST: **inserción**







TREE-INSERT(T,13)

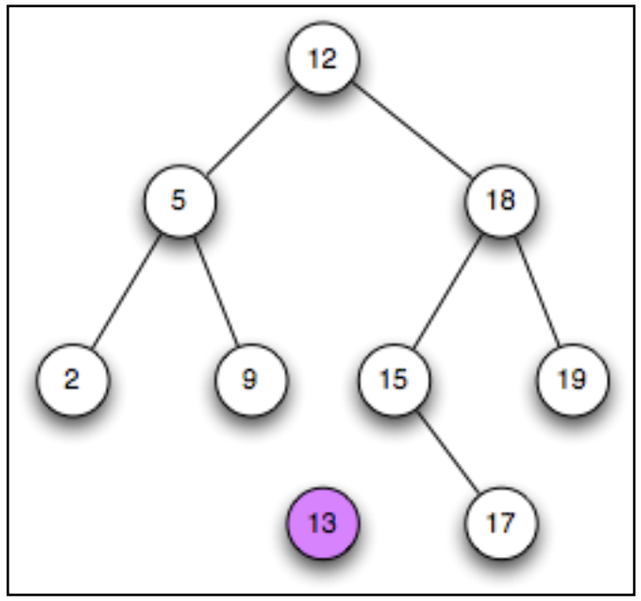
TREE-INSERT(T, z)

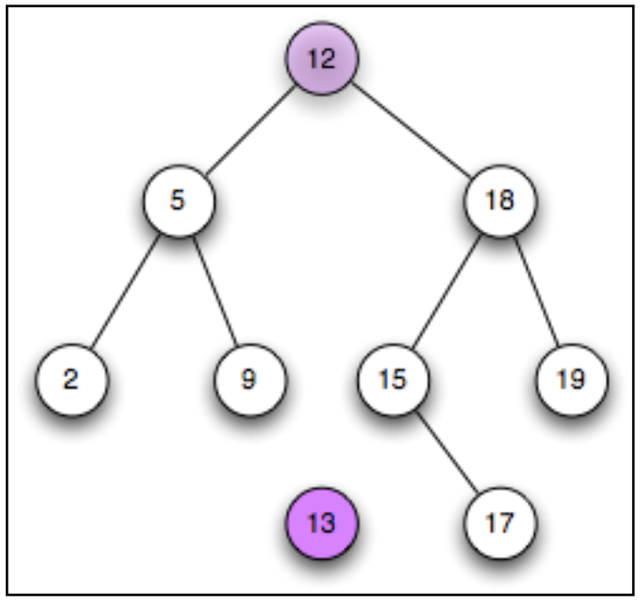
- y ← NULL
- x ← root[T]

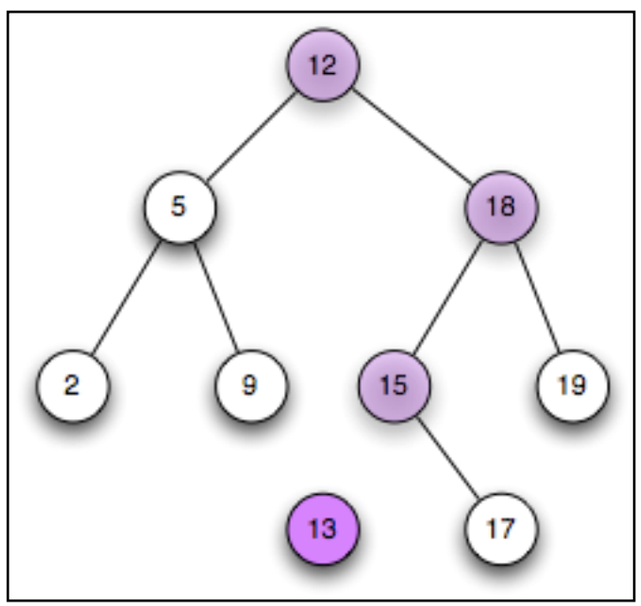
1. while x ≠ NULL
2. do y ← x
3. if key[z] < key[x]
4. then x ← left[x]
5. else x ← right[x]

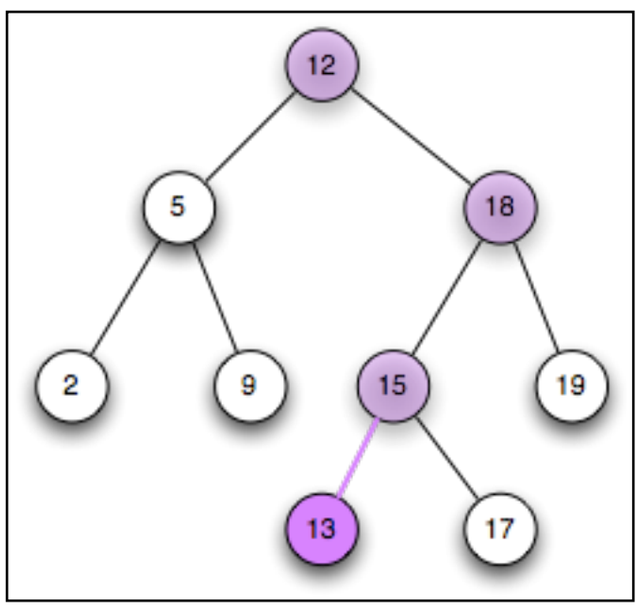
- p[z] ← y
- if y = NULL
- then root[T] ← z
- else if key[z] < key[y]
- then left[y] ← z
- else right[y] ← z

Operaciones en BST: **inserción**









TREE-INSERT(T,13)

TREE-INSERT(T, z)

- y ← NULL
- x ← root[T]

1. while x ≠ NULL
2. do y ← x
3. if key[z] < key[x]
4. then x ← left[x]
5. else x ← right[x]

- p[z] ← y
- if y = NULL
- then root[T] ← z
- else if key[z] < key[y]
- then left[y] ← z
- else right[y] ← z

Operaciones en BST: **eliminación**

Operaciones en BST: **eliminación**

- Dado un árbol binario de búsqueda T eliminar un nodo determinado.

Operaciones en BST: **eliminación**

- Dado un árbol binario de búsqueda T eliminar un nodo determinado.
- Se da como entrada un apuntador al nodo z a borrar.

Operaciones en BST: **eliminación**

- Dado un árbol binario de búsqueda T eliminar un nodo determinado.
- Se da como entrada un apuntador al nodo z a borrar.
- Buscamos borrar z y arreglar el árbol para preservar sus propiedades.

Operaciones en BST: **eliminación**

- Dado un árbol binario de búsqueda T eliminar un nodo determinado.
- Se da como entrada un apuntador al nodo z a borrar.
- Buscamos borrar z y arreglar el árbol para preservar sus propiedades.

TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z
3.       else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y]
7.  if x ≠ NULL
8.       then p[x] ← p[y]
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x
•            else right[p[y]] ← x
• if y ≠ z
•     then key[z] ← key[y]
1.
2. return y
```

Operaciones en BST: **eliminación**

y

TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z //aqui
3.       else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y] //aqui, es null
7.  if x ≠ NULL
8.       then p[x] ← p[y]
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x
•            else right[p[y]] ← x // aqui
•   if y ≠ z
•     then key[z] ← key[y]
1.         copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**

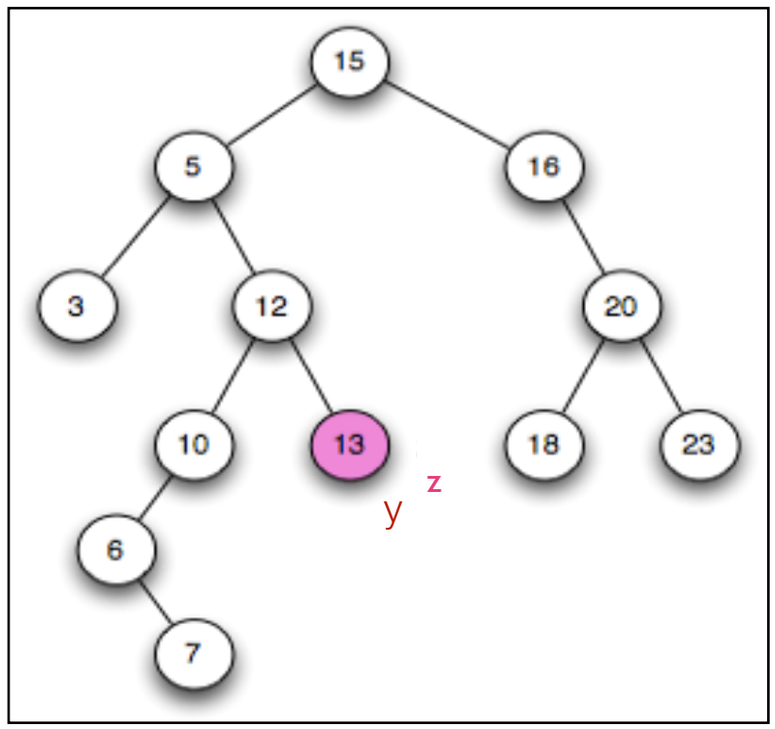
TREE-DELETE(T,13)

y

TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z //aqui
3.       else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y] //aqui, es null
7.  if x ≠ NULL
8.       then p[x] ← p[y]
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x
•            else right[p[y]] ← x // aqui
•   if y ≠ z
•     then key[z] ← key[y]
1.         copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**

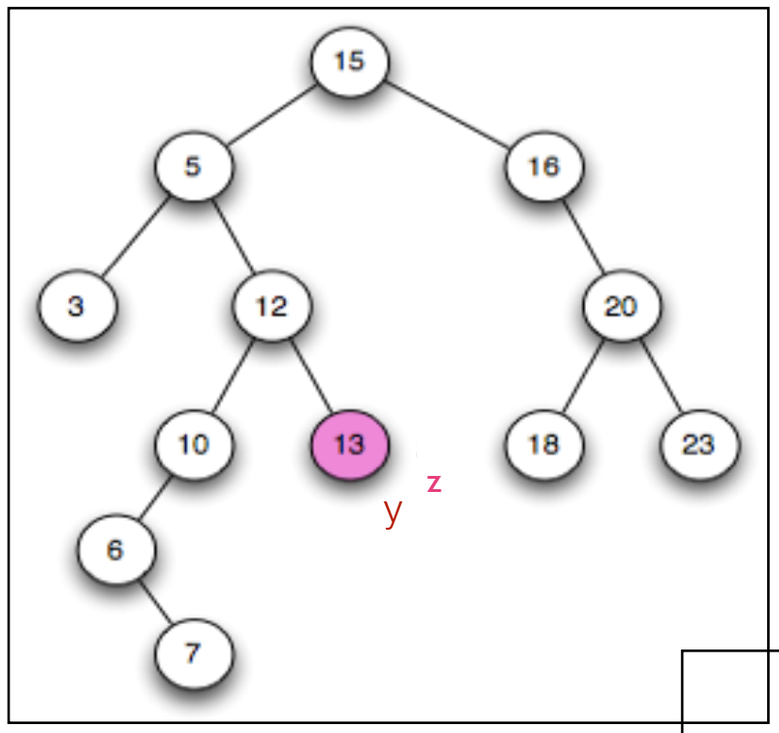


TREE-DELETE(T,13)

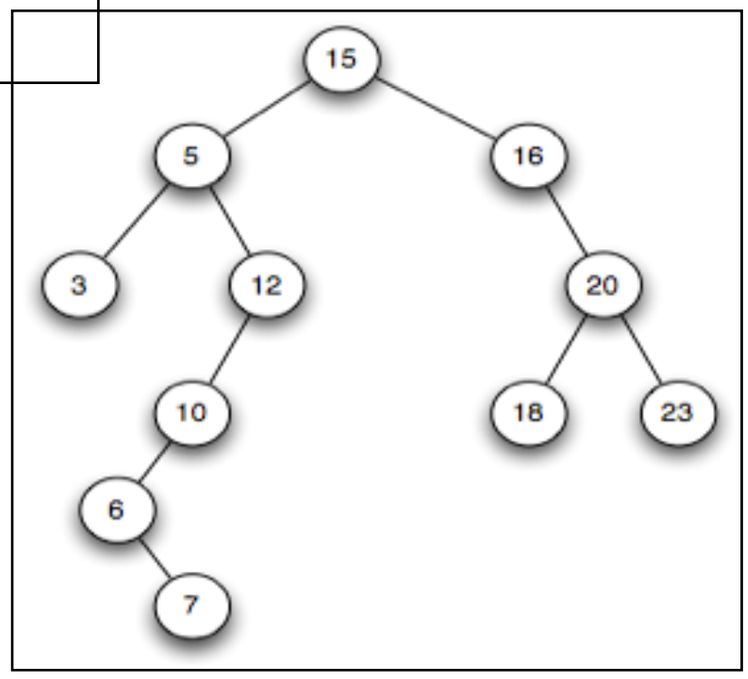
TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.        then y ← z //aqui
3.        else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ NULL
5.        then x ← left[y]
6.        else x ← right[y] //aqui, es null
7.  if x ≠ NULL
8.        then p[x] ← p[y]
9.  if p[y] = NULL
10.       then root[ T ] ← x
11.       else if y = left[p[y]]
12.             then left[p[y]] ← x
•             else right[p[y]] ← x // aqui
•   if y ≠ z
•     then key[z] ← key[y]
1.          copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**



TREE-DELETE(T,13)



TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.        then y ← z //aqui
3.        else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ NULL
5.        then x ← left[y]
6.        else x ← right[y] //aqui, es null
7.  if x ≠ NULL
8.        then p[x] ← p[y]
9.  if p[y] = NULL
10.       then root[ T ] ← x
11.       else if y = left[p[y]]
12.             then left[p[y]] ← x
•             else right[p[y]] ← x // aqui
•   if y ≠ z
•     then key[z] ← key[y]
1.          copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**

y

x

TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z  // aqui
3.       else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y]  //aqui
7.  if x ≠ NULL
8.       then p[x] ← p[y] //aqui
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x
•            else right[p[y]] ← x //aqui
• if y ≠ z
•     then key[z] ← key[y]
1.          copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**

TREE-DELETE(T,16)

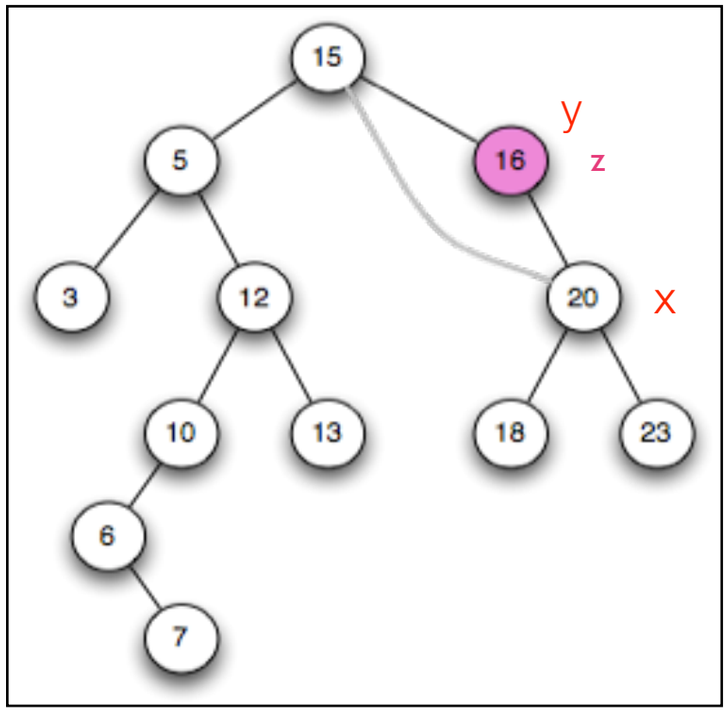
y

x

TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z  // aqui
3.       else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y]  //aqui
7.  if x ≠ NULL
8.       then p[x] ← p[y] //aqui
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x
•            else right[p[y]] ← x //aqui
• if y ≠ z
•     then key[z] ← key[y]
1.          copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**

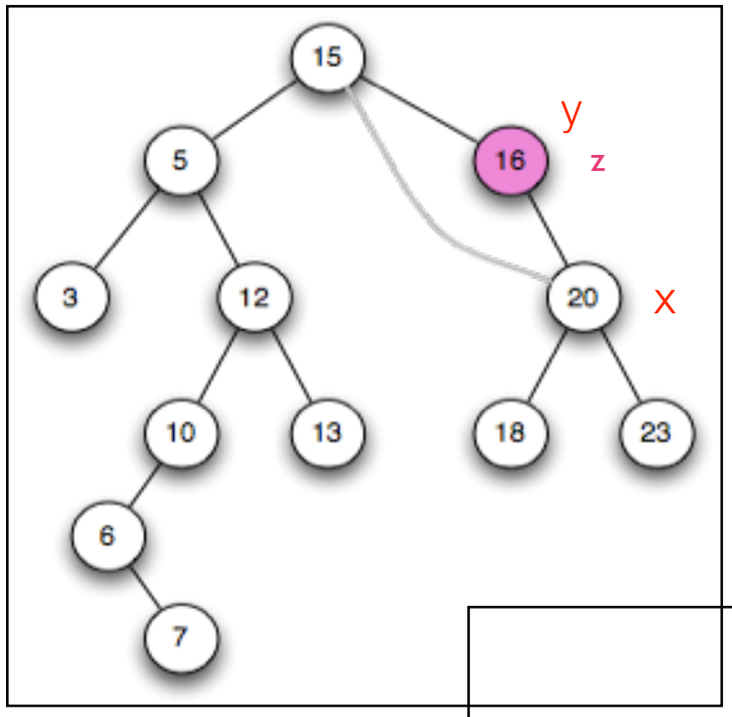


TREE-DELETE(T,16)

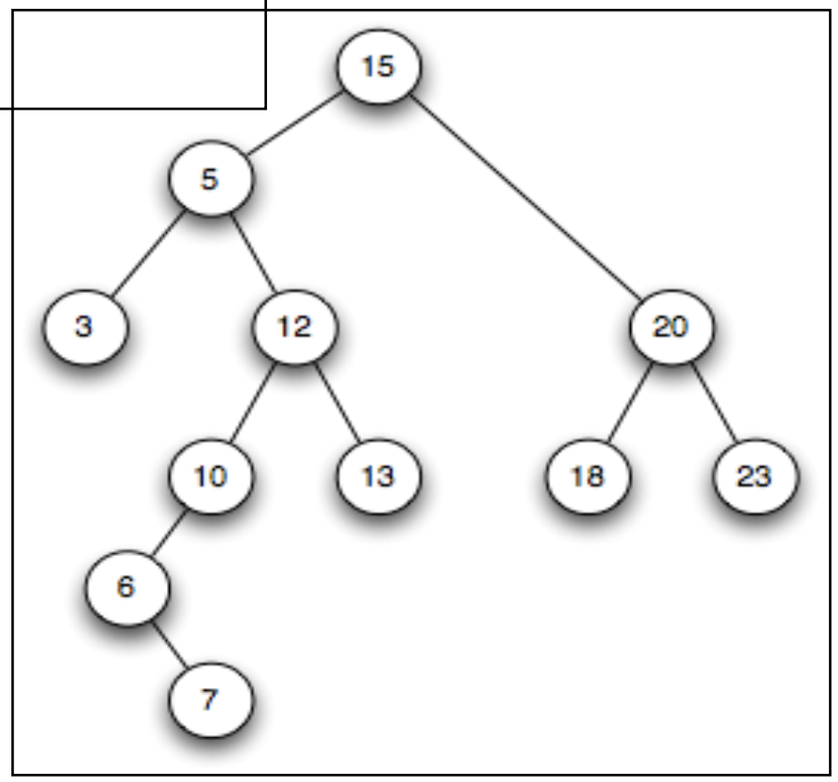
TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z  // aqui
3.       else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y]  //aqui
7.  if x ≠ NULL
8.       then p[x] ← p[y] //aqui
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x
•          else right[p[y]] ← x //aqui
• if y ≠ z
•     then key[z] ← key[y]
1.         copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**



TREE-DELETE(T,16)



TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z  // aqui
3.       else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y]  //aqui
7.  if x ≠ NULL
8.       then p[x] ← p[y] //aqui
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x
•            else right[p[y]] ← x //aqui
• if y ≠ z
•     then key[z] ← key[y]
1.         copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**

x

TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z
3.       else y ← TREE-SUCCESSOR(z) //aqui
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y] //aqui
7.  if x ≠ NULL
8.       then p[x] ← p[y] // aqui
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x //aqui
•            else right[p[y]] ← x
• if y ≠ z
•     then key[z] ← key[y] //aqui
1.          copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**

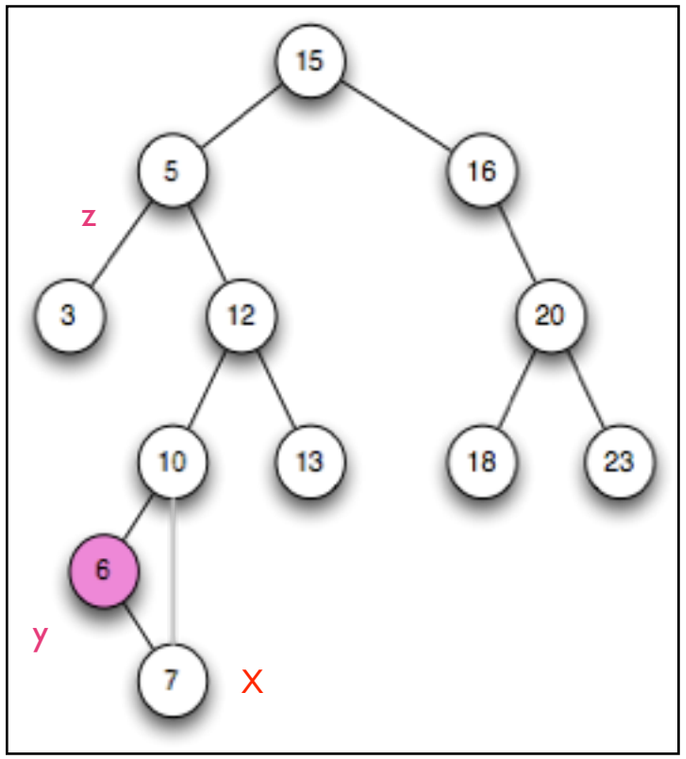
x

TREE-DELETE(T,5)

TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z
3.       else y ← TREE-SUCCESSOR(z) //aqui
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y] //aqui
7.  if x ≠ NULL
8.       then p[x] ← p[y] // aqui
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x //aqui
•            else right[p[y]] ← x
• if y ≠ z
•     then key[z] ← key[y] //aqui
1.          copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**

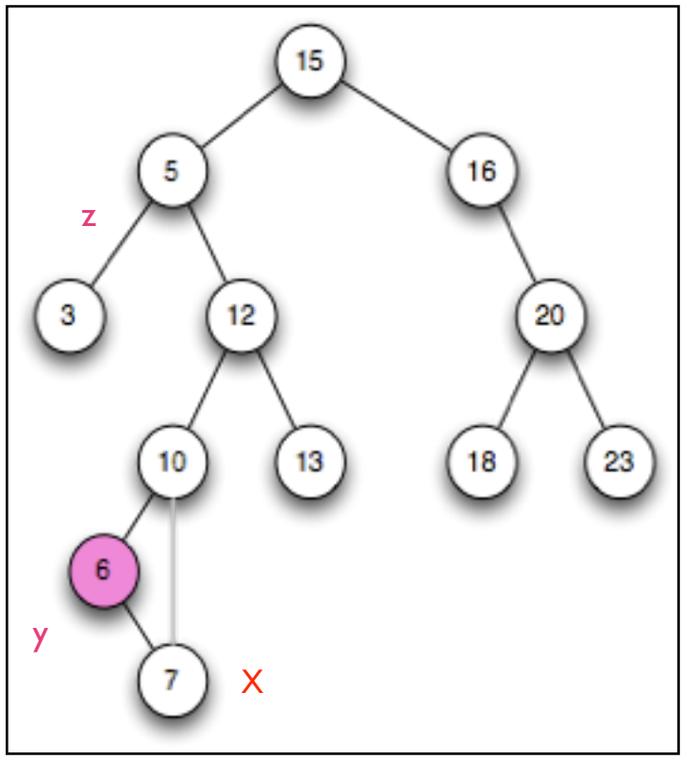


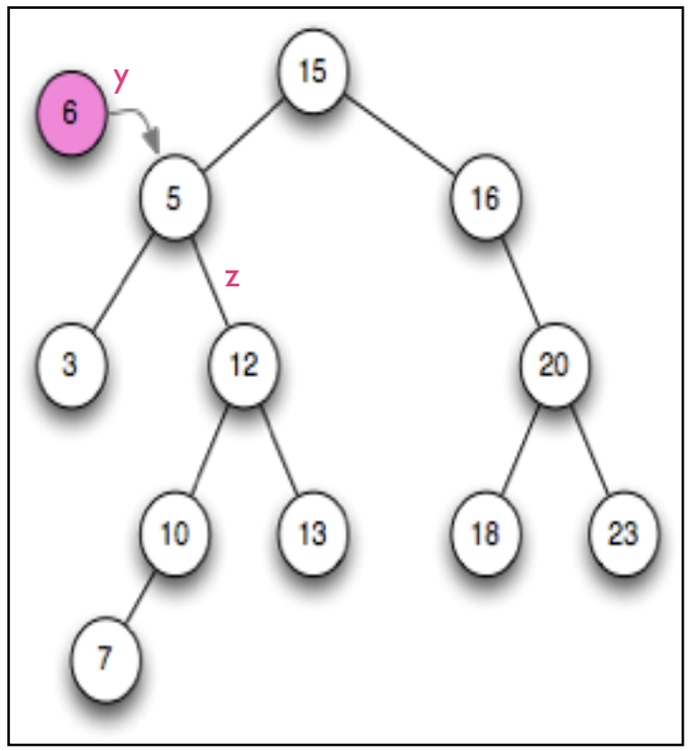
TREE-DELETE(T,5)

TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z
3.       else y ← TREE-SUCCESSOR(z) //aqui
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y] //aqui
7.  if x ≠ NULL
8.       then p[x] ← p[y] // aqui
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x //aqui
•            else right[p[y]] ← x
• if y ≠ z
•     then key[z] ← key[y] //aqui
1.          copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**



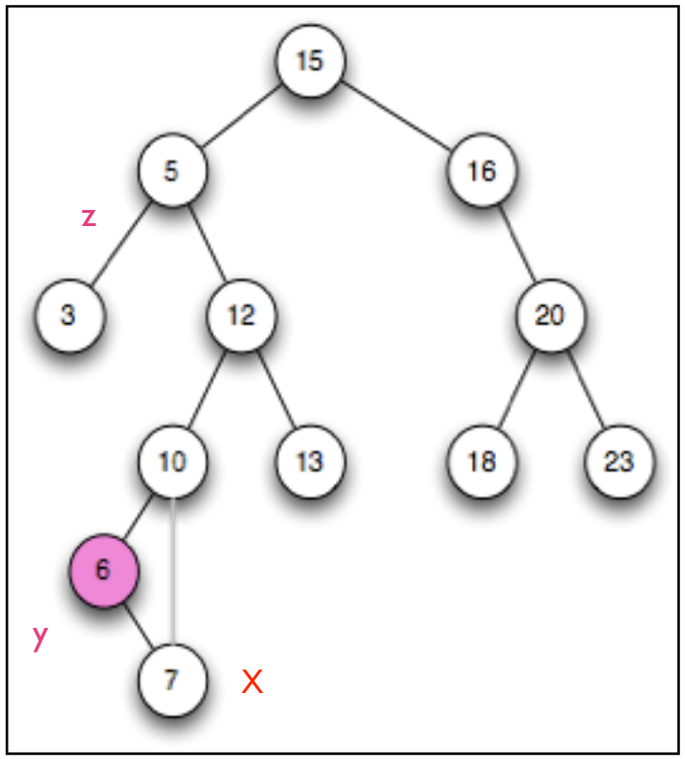


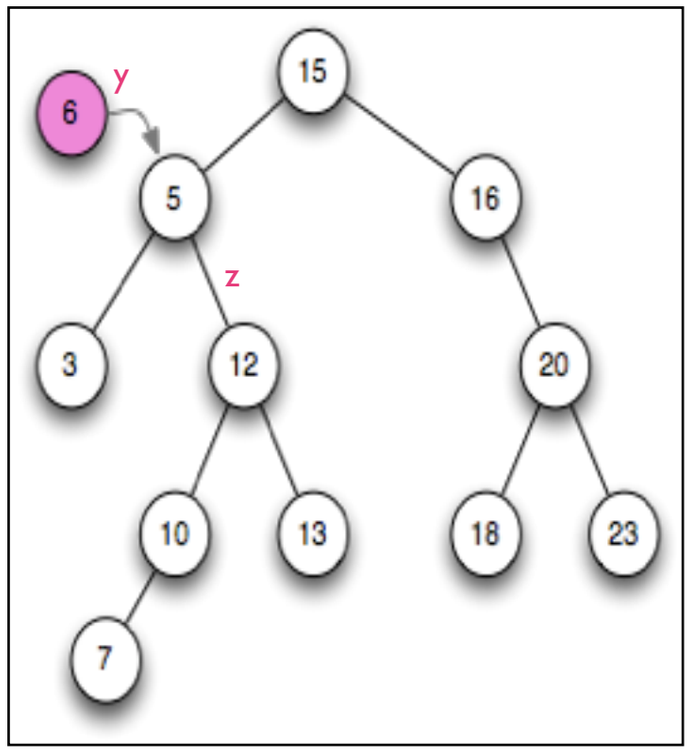
TREE-DELETE(T,5)

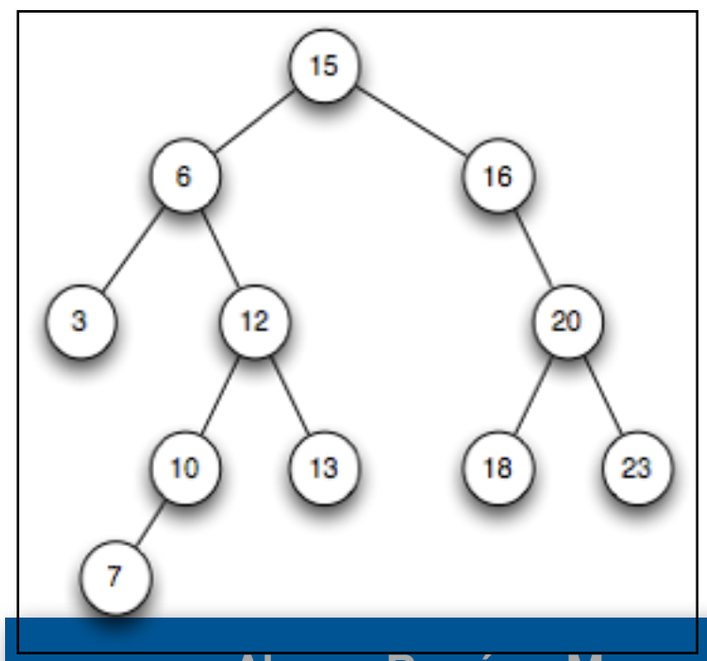
TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z
3.       else y ← TREE-SUCCESSOR(z) //aqui
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y] //aqui
7.  if x ≠ NULL
8.       then p[x] ← p[y] // aqui
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x //aqui
•            else right[p[y]] ← x
• if y ≠ z
•      then key[z] ← key[y] //aqui
1.          copiar los datos de y a z
2. return y
```

Operaciones en BST: **eliminación**







TREE-DELETE(T,5)

TREE-DELETE(T, z)

```
1.  if left[z] = NULL o right[z] = NULL
2.       then y ← z
3.       else y ← TREE-SUCCESSOR(z) //aqui
4.  if left[y] ≠ NULL
5.       then x ← left[y]
6.       else x ← right[y] //aqui
7.  if x ≠ NULL
8.       then p[x] ← p[y] // aqui
9.  if p[y] = NULL
10.      then root[ T ] ← x
11.      else if y = left[p[y]]
12.           then left[p[y]] ← x //aqui
•            else right[p[y]] ← x
• if y ≠ z
•     then key[z] ← key[y] //aqui
1.         copiar los datos de y a z
2. return y
```

Desempeño de búsqueda secuencial en diccionarios

	peor caso			caso promedio		
	insert	search	select Regresa el k-ésimo	insert	hit	miss
arreglo indexado por llaves	I	I	M	I	I	I
arreglo ordenado	N	N	I	N/2	N/2	N/2
lista ligada ordenada	N	N	N	N/2	N/2	N/2
arreglo no-ordenado	I	N	?	I	N/2	N
lista ligada no ordenada	I	N	?	I	N/2	N
búsqueda binaria	N	lg N	I	N/2	lg N	lg N
árboles binarios de búsqueda	N	N	N	lg N	lg N	lg N

Árboles binarios de búsqueda (BST) y Arboles balanceados (o equilibrados)

mat-151

Desempeño de los BST

Desempeño de los BST

- Las operaciones presentadas para los BST funcionan bien para una gran variedad de aplicaciones pero tienen el problema de su funcionamiento en el peor caso.

Desempeño de los BST

- Las operaciones presentadas para los BST funcionan bien para una gran variedad de aplicaciones pero tienen el problema de su funcionamiento en el peor caso.

- Peor caso: datos ordenados, datos con un gran número de llaves repetidas , datos en orden inverso, datos con llaves pequeñas y grandes alternadas.

Desempeño de los BST

- Las operaciones presentadas para los BST funcionan bien para una gran variedad de aplicaciones pero tienen el problema de su funcionamiento en el peor caso.
- Peor caso: datos ordenados, datos con un gran número de llaves repetidas , datos en orden inverso, datos con llaves pequeñas y grandes alternadas.
- En el caso ideal quisiéramos tener arboles perfectamente balanceados que:

Desempeño de los BST

- Las operaciones presentadas para los BST funcionan bien para una gran variedad de aplicaciones pero tienen el problema de su funcionamiento en el peor caso.

- Peor caso: datos ordenados, datos con un gran número de llaves repetidas , datos en orden inverso, datos con llaves pequeñas y grandes alternadas.

- En el caso ideal quisiéramos tener arboles perfectamente balanceados que:

  - corresponden a una busqueda binaria completada en menos de lg N+1 comparaciones pero que

Desempeño de los BST

- Las operaciones presentadas para los BST funcionan bien para una gran variedad de aplicaciones pero tienen el problema de su funcionamiento en el peor caso.

- Peor caso: datos ordenados, datos con un gran número de llaves repetidas , datos en orden inverso, datos con llaves pequeñas y grandes alternadas.

- En el caso ideal quisiéramos tener arboles perfectamente balanceados que:
  - corresponden a una busqueda binaria completada en menos de lg N+1 comparaciones pero que
  - es costoso mantener por sus inserciones y eliminaciones.

Desempeño de los BST

- Las operaciones presentadas para los BST funcionan bien para una gran variedad de aplicaciones pero tienen el problema de su funcionamiento en el peor caso.
- Peor caso: datos ordenados, datos con un gran número de llaves repetidas , datos en orden inverso, datos con llaves pequeñas y grandes alternadas.

- En el caso ideal quisiéramos tener arboles perfectamente balanceados que:
  - corresponden a una busqueda binaria completada en menos de lg N+1 comparaciones pero que
  - es costoso mantener por sus inserciones y eliminaciones.

Desempeño de los BST

- Las operaciones presentadas para los BST funcionan bien para una gran variedad de aplicaciones pero tienen el problema de su funcionamiento en el peor caso.
- Peor caso: datos ordenados, datos con un gran número de llaves repetidas , datos en orden inverso, datos con llaves pequeñas y grandes alternadas.

- En el caso ideal quisiéramos tener arboles perfectamente balanceados que:
  - corresponden a una busqueda binaria completada en menos de lg N+1 comparaciones pero que
  - es costoso mantener por sus inserciones y eliminaciones.

Árboles balanceados

Árboles balanceados

- Un BST de **altura h** puede contener **hasta 2^h-1 valores**.

Árboles balanceados

- Un BST de altura h puede contener hasta 2^h-1 valores.
- Dados n valores, siempre podemos construir un árbol binario de búsqueda de altura máxima de $1+\log_2(n)$.

Árboles balanceados

- Un BST de altura h puede contener hasta 2^h-1 valores.
- Dados n valores, siempre podemos construir un árbol binario de búsqueda de altura máxima de $1+\log_2(n)$.
  - Ordenar los n valores en orden ascendente y ponerles en un arreglo A[0...n-1].

Árboles balanceados

- Un BST de altura h puede contener hasta 2^h-1 valores.
- Dados n valores, siempre podemos construir un árbol binario de búsqueda de altura máxima de $1+\log_2(n)$.
  - Ordenar los n valores en orden ascendente y ponerles en un arreglo A[0...n-1].
  - Hacer al elemento A[n/2] la raíz.

Árboles balanceados

- Un BST de altura h puede contener hasta 2^h-1 valores.
- Dados n valores, siempre podemos construir un árbol binario de búsqueda de altura máxima de $1+\log_2(n)$.
  - Ordenar los n valores en orden ascendente y ponerles en un arreglo A[0...n-1].
  - Hacer al elemento A[n/2] la raíz.
  - Construir el sub-árbol izquierdo con los elementos A[0...n/2-1]

Árboles balanceados

- Un BST de altura h puede contener hasta 2^h-1 valores.
- Dados n valores, siempre podemos construir un árbol binario de búsqueda de altura máxima de $1+\log_2(n)$.
  - Ordenar los n valores en orden ascendente y ponerles en un arreglo A[0...n-1].
  - Hacer al elemento A[n/2] la raíz.
  - Construir el sub-árbol izquierdo con los elementos A[0...n/2-1]
  - Construir el sub-árbol derecho con los elementos A[n/2+1...n].

Árboles balanceados

- Un BST de altura h puede contener hasta 2^h-1 valores.
- Dados n valores, siempre podemos construir un árbol binario de búsqueda de altura máxima de $1+\log_2(n)$.
  - Ordenar los n valores en orden ascendente y ponerles en un arreglo A[0...n-1].
  - Hacer al elemento A[n/2] la raíz.
  - Construir el sub-árbol izquierdo con los elementos A[0...n/2-1]
  - Construir el sub-árbol derecho con los elementos A[n/2+1...n].
- Pero también se puede construir un BST de altura n-1.

Árboles balanceados

- Un BST de altura h puede contener hasta 2^h-1 valores.
- Dados n valores, siempre podemos construir un árbol binario de búsqueda de altura máxima de $1+\log_2(n)$.
  - Ordenar los n valores en orden ascendente y ponerles en un arreglo A[0...n-1].
  - Hacer al elemento A[n/2] la raíz.
  - Construir el sub-árbol izquierdo con los elementos A[0...n/2-1]
  - Construir el sub-árbol derecho con los elementos A[n/2+1...n].
- Pero también se puede construir un BST de altura n-1.
  - Hacer A[0] la raiz, hacer A[1] el sub-árbol derecho, ...

Árboles balanceados

Árboles balanceados

- Para conseguir tiempos de ejecución de O(log n) es necesario tener arboles balanceados: i.e., todas las hojas deben tener casi la misma distancia a la raíz.

Árboles balanceados

- Para conseguir tiempos de ejecución de O(log n) es necesario tener arboles balanceados: i.e., todas las hojas deben tener casi la misma distancia a la raíz.
- Esto es fácil si el contenido del árbol esta fijo (el numero de elementos).

Árboles balanceados

- Para conseguir tiempos de ejecución de O(log n) es necesario tener arboles balanceados: i.e., todas las hojas deben tener casi la misma distancia a la raíz.
- Esto es fácil si el contenido del árbol esta fijo (el numero de elementos).
- No tan fácil si se insertan y eliminan elementos dinámicamente.

Árboles balanceados

- Para conseguir tiempos de ejecución de O(log n) es necesario tener arboles balanceados: i.e., todas las hojas deben tener casi la misma distancia a la raíz.
- Esto es fácil si el contenido del árbol esta fijo (el numero de elementos).
- No tan fácil si se insertan y eliminan elementos dinámicamente.
  - podemos intentar tener un árbol balanceado en promedio, i.e., la probabilidad de tener un árbol que no este balanceado tiende a cero mientras $n \to \infty$.

Árboles balanceados

- Para conseguir tiempos de ejecución de O(log n) es necesario tener arboles balanceados: i.e., todas las hojas deben tener casi la misma distancia a la raíz.
- Esto es fácil si el contenido del árbol esta fijo (el numero de elementos).
- No tan fácil si se insertan y eliminan elementos dinámicamente.
  - podemos intentar tener un árbol balanceado en promedio, i.e., la probabilidad de tener un árbol que no este balanceado tiende a cero mientras $n \to \infty$.
  - se puede tener balance determinista que garantiza balance en el peor caso: árboles rojo negros, arboles AVL, arboles 2-3, B-trees, s

Desempeño de diccionarios implementados con BST

Desempeño de diccionarios implementados con BST

- El problema de garantizar el desempeño para diccionarios con BST nos permite describir 3 tipos de soluciones generales dentro del diseño de algoritmos:

Desempeño de diccionarios implementados con BST

- El problema de garantizar el desempeño para diccionarios con BST nos permite describir 3 tipos de soluciones generales dentro del diseño de algoritmos:
  - aleatorios (randomized)

Desempeño de diccionarios implementados con BST

- El problema de garantizar el desempeño para diccionarios con BST nos permite describir 3 tipos de soluciones generales dentro del diseño de algoritmos:
  - aleatorios (randomized)
  - amortizados (amortized)

Desempeño de diccionarios implementados con BST

- El problema de garantizar el desempeño para diccionarios con BST nos permite describir 3 tipos de soluciones generales dentro del diseño de algoritmos:
  - aleatorios (randomized)
  - amortizados (amortized)
  - optimizados (optimized)

Desempeño de diccionarios implementados con BST

- El problema de garantizar el desempeño para diccionarios con BST nos permite describir 3 tipos de soluciones generales dentro del diseño de algoritmos:
  - aleatorios (randomized)
  - amortizados (amortized)
  - optimizados (optimized)

Desempeño de diccionarios implementados con BST

- El problema de garantizar el desempeño para diccionarios con BST nos permite describir 3 tipos de soluciones generales dentro del diseño de algoritmos:
  - aleatorios (randomized)
  - amortizados (amortized)
  - optimizados (optimized)

- Los algoritmos aleatorios introducen decisiones aleatorias en el algoritmo mismo, reduciendo dramáticamente la probabilidad de un peor caso sin importar la entrada.

Desempeño de diccionarios implementados con BST

Desempeño de diccionarios implementados con BST

- Los algoritmos amortizados implican hacer trabajo extra en algún momento del algoritmos para evitar hacerlo después. De esta manera permite garantizar cotas superiores sobre el costo promedio por operación.

Desempeño de diccionarios implementados con BST

- Los algoritmos amortizados implican hacer trabajo extra en algún momento del algoritmos para evitar hacerlo después. De esta manera permite garantizar cotas superiores sobre el costo promedio por operación.

- El enfoque de optimización consiste en proporcionar garantías de desempeño para cada operación. Estos algoritmos requieren mantener algún tipo de información estructural en los árboles.

Operaciones en BST: **rotación**

Operaciones en BST: **rotación**

- Una rotación es una operación que cambia la estructura de un árbol binario de búsqueda sin interferir con el orden de sus elementos.

Operaciones en BST: **rotación**

- Una rotación es una operación que cambia la estructura de un árbol binario de búsqueda sin interferir con el orden de sus elementos.
- Una rotación mueve un nodo hacia arriba en el árbol y uno hacia abajo.

Operaciones en BST: **rotación**

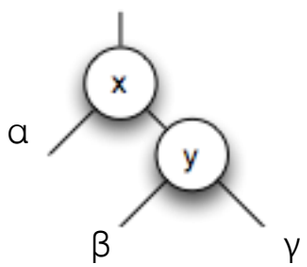
- Una rotación es una operación que cambia la estructura de un árbol binario de búsqueda sin interferir con el orden de sus elementos.
- Una rotación mueve un nodo hacia arriba en el árbol y uno hacia abajo.
- Se utilizan para cambiar la forma del árbol y reducir la altura, lo que resulta en una mejora en el desempeño de las otras operaciones.

Operaciones en BST: **rotación**

- Una rotación es una operación que cambia la estructura de un árbol binario de búsqueda sin interferir con el orden de sus elementos.
- Una rotación mueve un nodo hacia arriba en el árbol y uno hacia abajo.
- Se utilizan para cambiar la forma del árbol y reducir la altura, lo que resulta en una mejora en el desempeño de las otras operaciones.
- Hay dos tipos de rotaciones: hacia la derecha y hacia la izquierda.

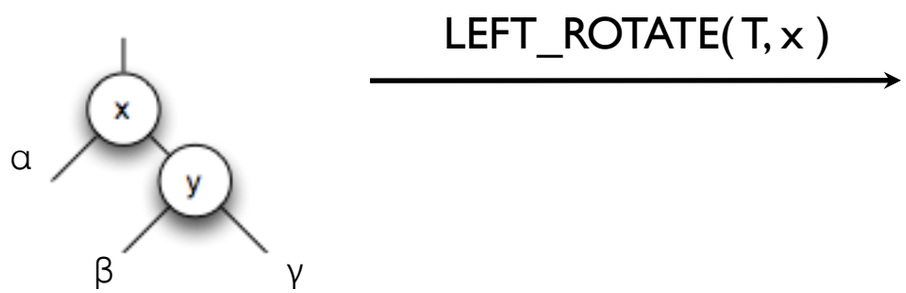
Operaciones en BST: **rotación**

- Una rotación es una operación que cambia la estructura de un árbol binario de búsqueda sin interferir con el orden de sus elementos.
- Una rotación mueve un nodo hacia arriba en el árbol y uno hacia abajo.
- Se utilizan para cambiar la forma del árbol y reducir la altura, lo que resulta en una mejora en el desempeño de las otras operaciones.
- Hay dos tipos de rotaciones: hacia la derecha y hacia la izquierda.



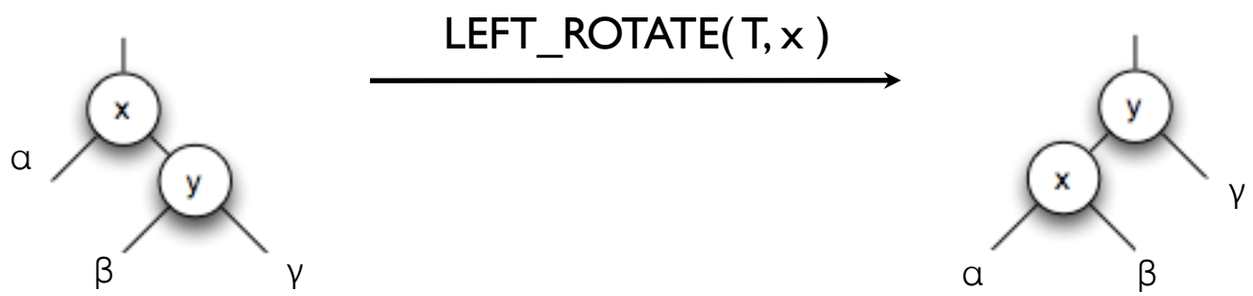
Operaciones en BST: **rotación**

- Una rotación es una operación que cambia la estructura de un árbol binario de búsqueda sin interferir con el orden de sus elementos.
- Una rotación mueve un nodo hacia arriba en el árbol y uno hacia abajo.
- Se utilizan para cambiar la forma del árbol y reducir la altura, lo que resulta en una mejora en el desempeño de las otras operaciones.
- Hay dos tipos de rotaciones: hacia la derecha y hacia la izquierda.



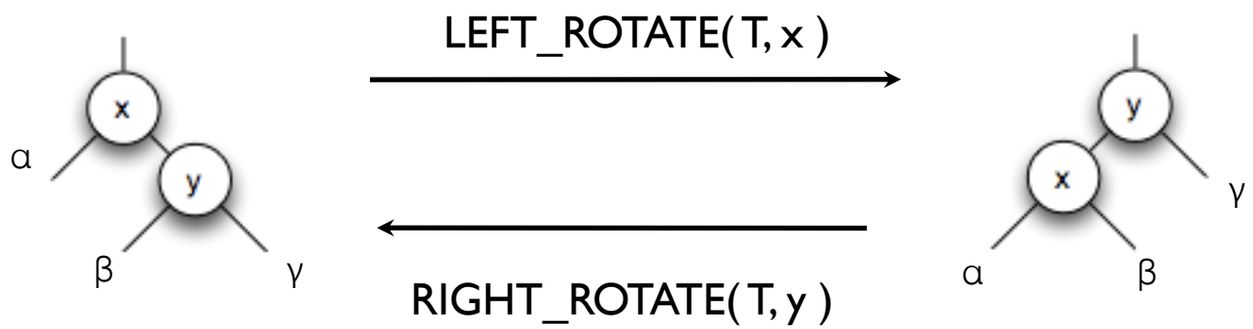
Operaciones en BST: **rotación**

- Una rotación es una operación que cambia la estructura de un árbol binario de búsqueda sin interferir con el orden de sus elementos.
- Una rotación mueve un nodo hacia arriba en el árbol y uno hacia abajo.
- Se utilizan para cambiar la forma del árbol y reducir la altura, lo que resulta en una mejora en el desempeño de las otras operaciones.
- Hay dos tipos de rotaciones: hacia la derecha y hacia la izquierda.

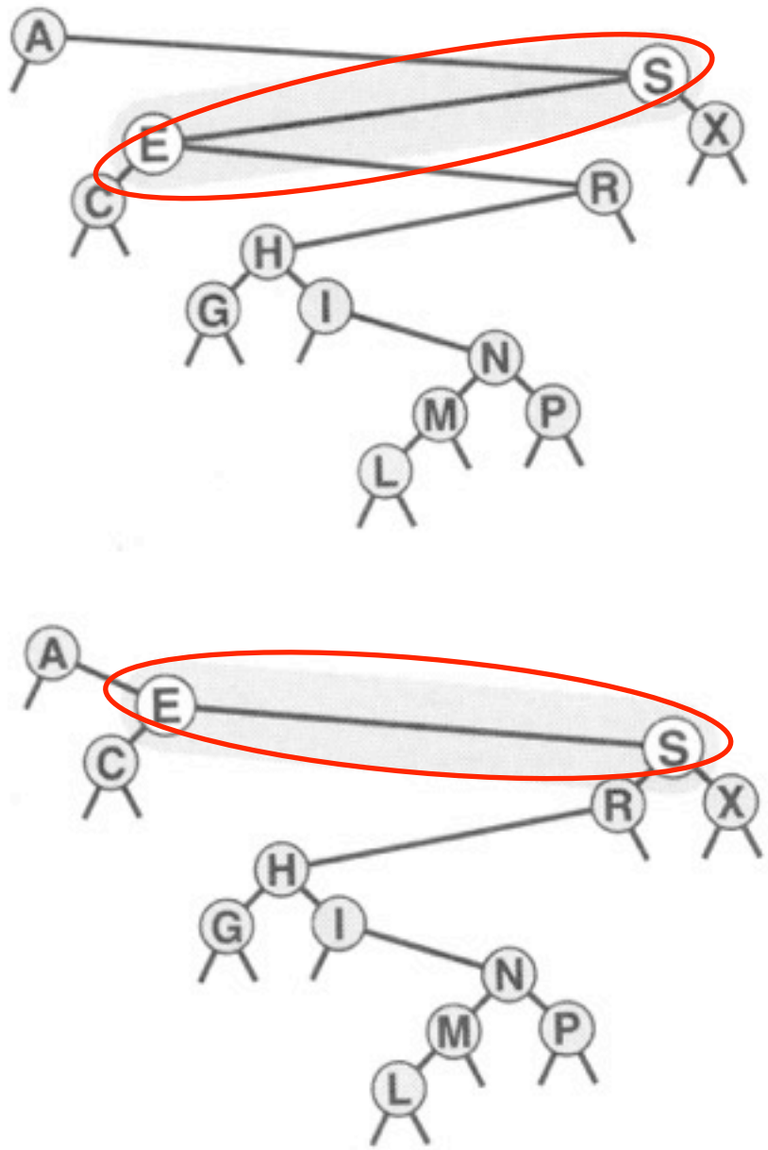


Operaciones en BST: **rotación**

- Una rotación es una operación que cambia la estructura de un árbol binario de búsqueda sin interferir con el orden de sus elementos.
- Una rotación mueve un nodo hacia arriba en el árbol y uno hacia abajo.
- Se utilizan para cambiar la forma del árbol y reducir la altura, lo que resulta en una mejora en el desempeño de las otras operaciones.
- Hay dos tipos de rotaciones: hacia la derecha y hacia la izquierda.



Operaciones en BST: **rotación a la derecha**



Operaciones en BST: **rotación**

Sea A un árbol binario no-vacío

$A = (x, Z, T)$ donde x es la raíz de A, Z y T son los sub-árboles izquierdo y derecho respectivamente.

Operaciones en BST: **rotación**

Sea A un árbol binario no-vacío

$A = (x, Z, T)$ donde x es la raíz de A, Z y T son los sub-árboles izquierdo y derecho respectivamente.

Sea $A = (x, X, B)$ donde B es un árbol no-vacío y sea $B = (y, Y, Z)$.

Operaciones en BST: **rotación**

Sea A un árbol binario no-vacío

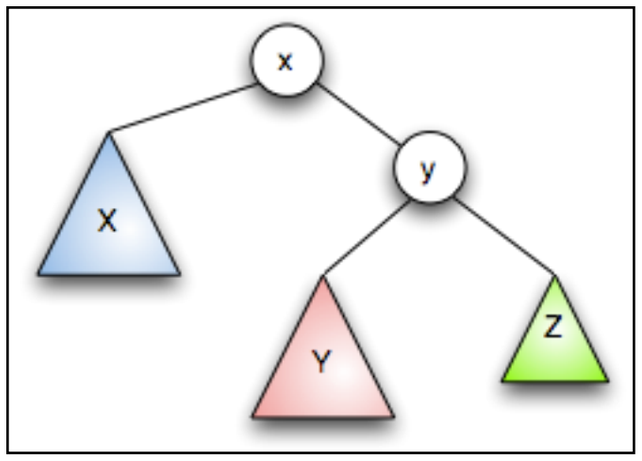
$A = (x, Z, T)$ donde x es la raíz de A, Z y T son los sub-árboles izquierdo y derecho respectivamente.

Sea $A = (x, X, B)$ donde B es un árbol no-vacío y sea $B = (y, Y, Z)$.

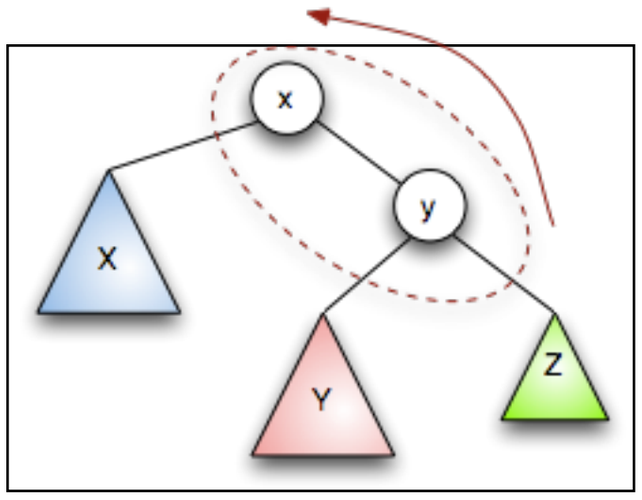
La operación de *rotación izquierda* de A es la operación:

$$A = (x, X, (y, Y, Z)) \rightarrow \mathcal{RI}(A) = (y, (x, X, Y), Z).$$

Operaciones en BST: **rotación izquierda**

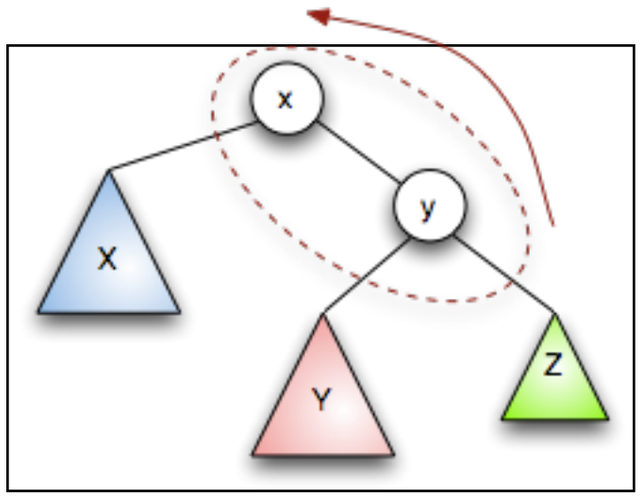


Operaciones en BST: **rotación izquierda**



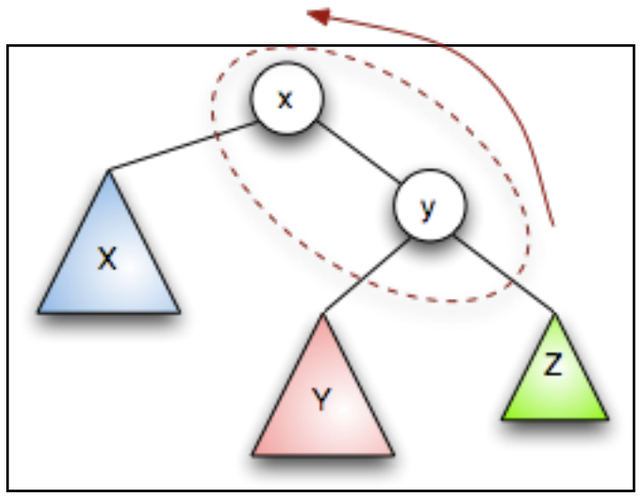
Operaciones en BST: **rotación izquierda**

- Reemplazamos el nodo raíz **x** por el nodo **y**.



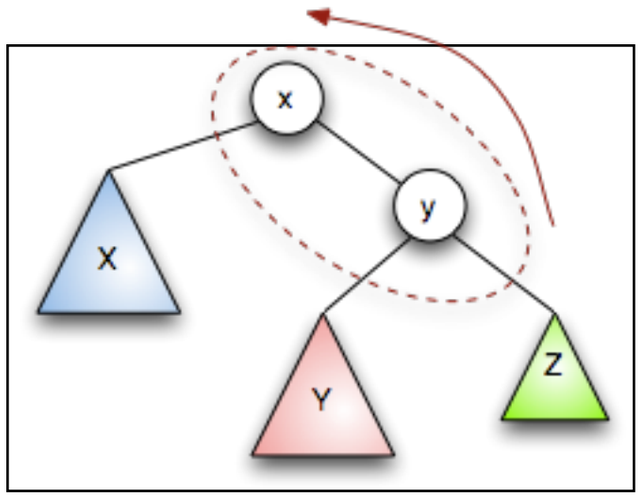
Operaciones en BST: **rotación izquierda**

- Reemplazamos el nodo raíz **x** por el nodo **y**.
- El hijo izquierdo de **y** apunta al nodo **x**.



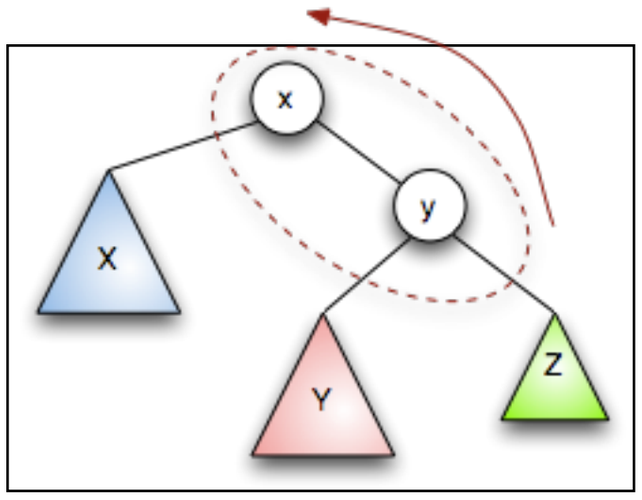
Operaciones en BST: **rotación izquierda**

- Reemplazamos el nodo raíz **x** por el nodo **y**.
- El hijo izquierdo de **y** apunta al nodo **x**.
- El hijo derecho de **y** queda intacto y apunta al sub-árbol **Z**.



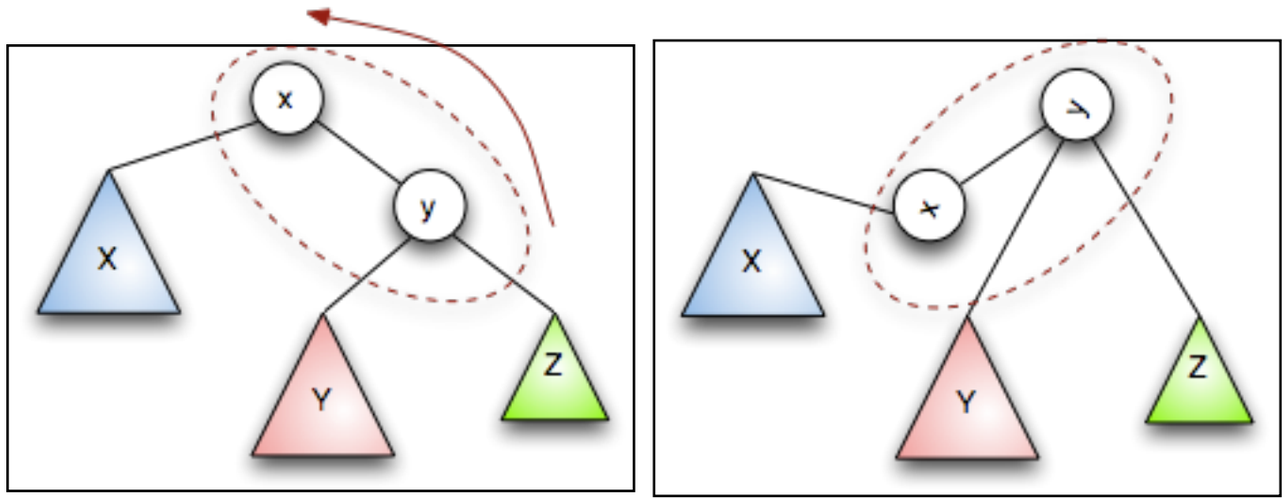
Operaciones en BST: **rotación izquierda**

- Reemplazamos el nodo raíz **x** por el nodo **y**.
- El hijo izquierdo de **y** apunta al nodo **x**.
- El hijo derecho de **y** queda intacto y apunta al sub-árbol **Z**.
- El hijo derecho del nodo **x** cambia su apuntador al sub-árbol **Y**.



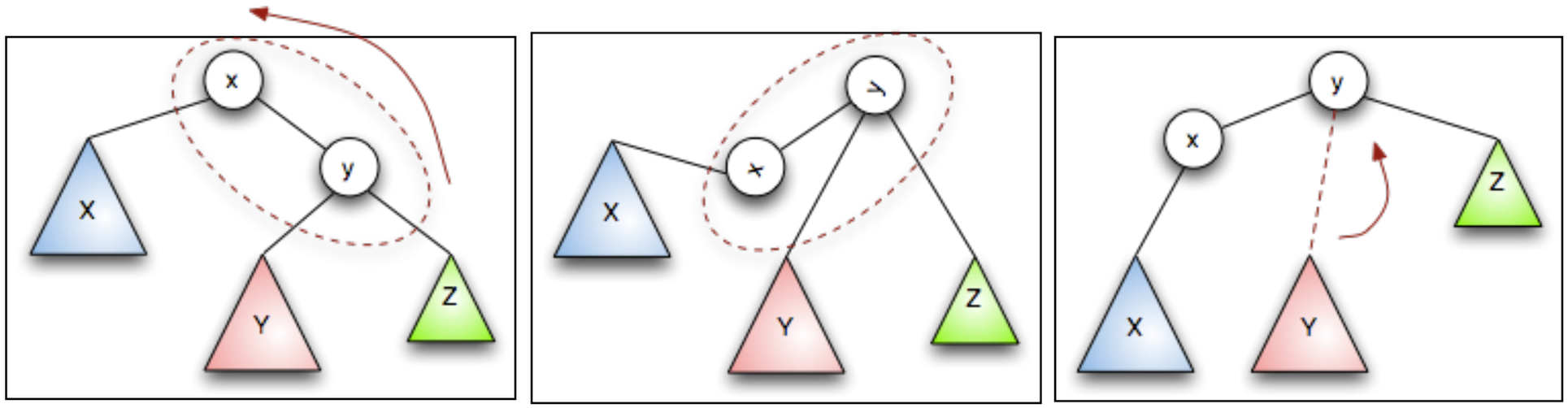
Operaciones en BST: **rotación izquierda**

- Reemplazamos el nodo raíz **x** por el nodo **y**.
- El hijo izquierdo de **y** apunta al nodo **x**.
- El hijo derecho de **y** queda intacto y apunta al sub-árbol **Z**.
- El hijo derecho del nodo **x** cambia su apuntador al sub-árbol **Y**.



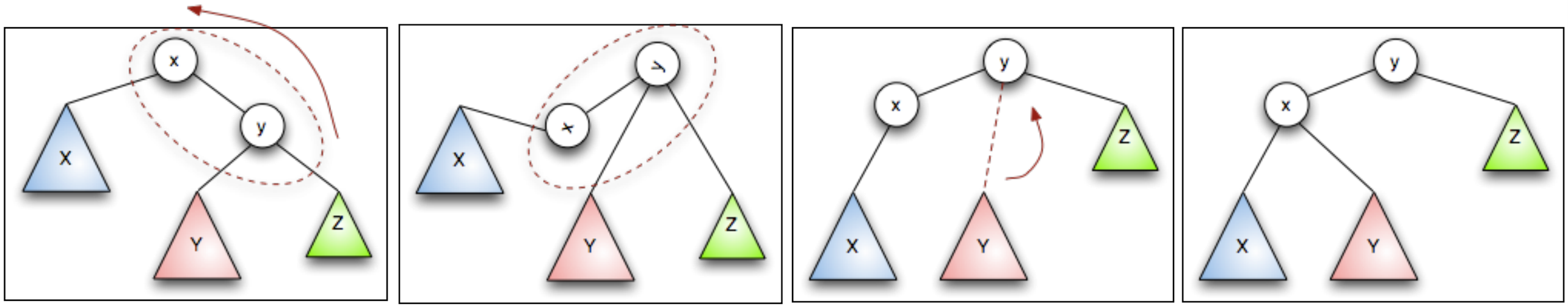
Operaciones en BST: **rotación izquierda**

- Reemplazamos el nodo raíz **x** por el nodo **y**.
- El hijo izquierdo de **y** apunta al nodo **x**.
- El hijo derecho de **y** queda intacto y apunta al sub-árbol **Z**.
- El hijo derecho del nodo **x** cambia su apuntador al sub-árbol **Y**.



Operaciones en BST: **rotación izquierda**

- Reemplazamos el nodo raíz **x** por el nodo **y**.
- El hijo izquierdo de **y** apunta al nodo **x**.
- El hijo derecho de **y** queda intacto y apunta al sub-árbol **Z**.
- El hijo derecho del nodo **x** cambia su apuntador al sub-árbol **Y**.



Operaciones en BST: **rotación**

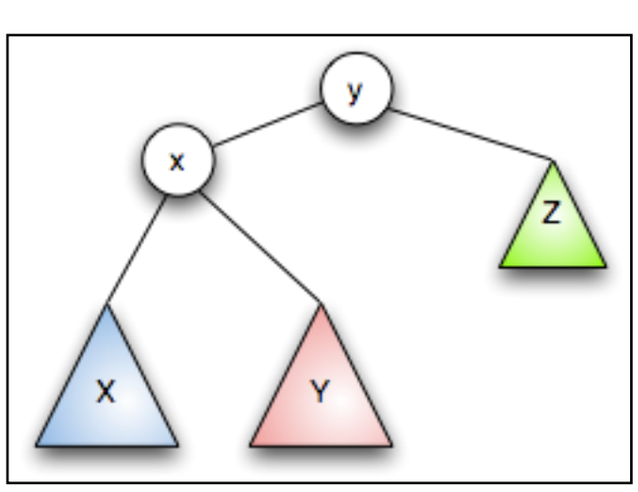
La *rotación derecha* es la operación inversa:

$$A = (y, (x, X, Y), Z) \rightarrow \mathcal{RD}(A) = (x, X, (y, Y, Z)).$$

Operaciones en BST: **rotación**

La *rotación derecha* es la operación inversa:

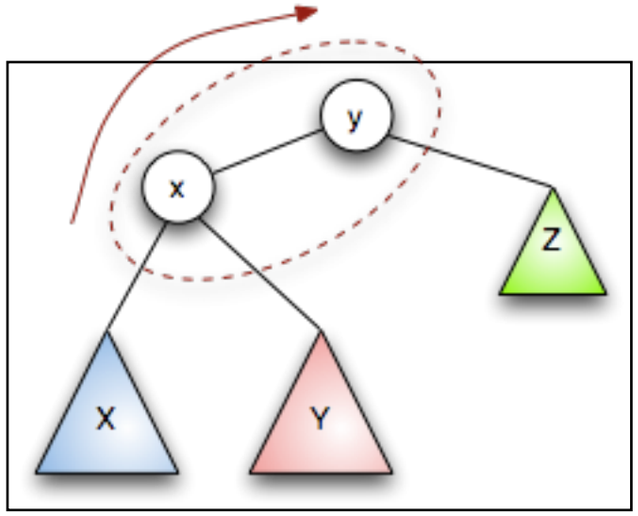
$$A = (y, (x, X, Y), Z) \rightarrow \mathcal{RD}(A) = (x, X, (y, Y, Z)).$$



Operaciones en BST: **rotación**

La *rotación derecha* es la operación inversa:

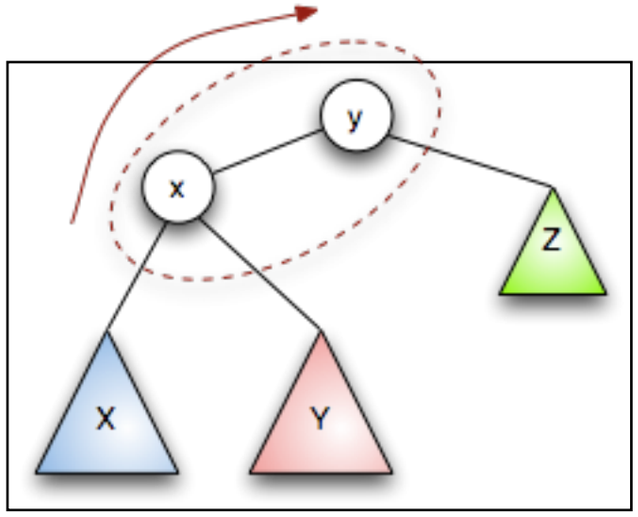
$$A = (y, (x, X, Y), Z) \rightarrow \mathcal{RD}(A) = (x, X, (y, Y, Z)).$$

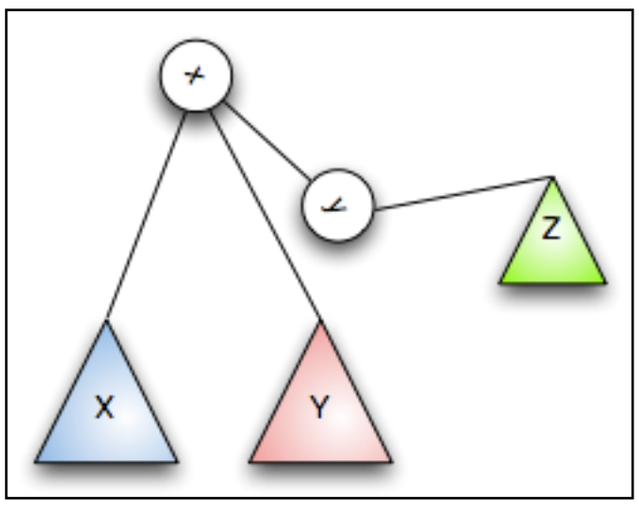


Operaciones en BST: **rotación**

La *rotación derecha* es la operación inversa:

$$A = (y, (x, X, Y), Z) \rightarrow \mathcal{RD}(A) = (x, X, (y, Y, Z)).$$

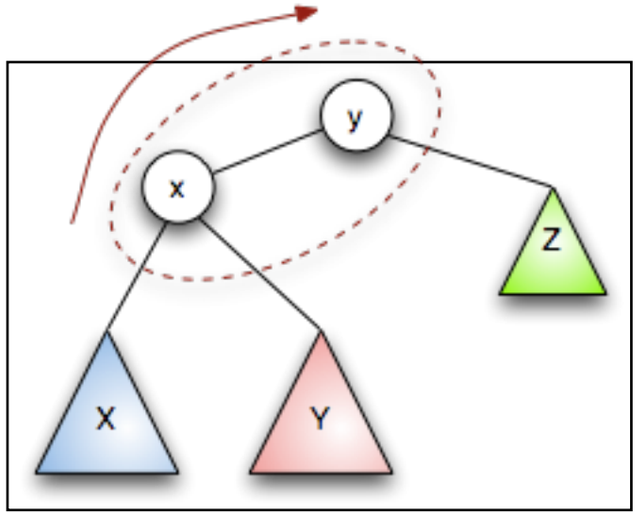


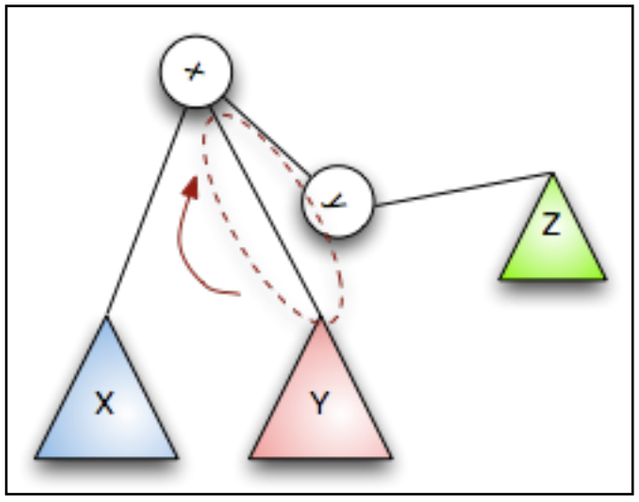


Operaciones en BST: **rotación**

La *rotación derecha* es la operación inversa:

$$A = (y, (x, X, Y), Z) \rightarrow \mathcal{RD}(A) = (x, X, (y, Y, Z)).$$

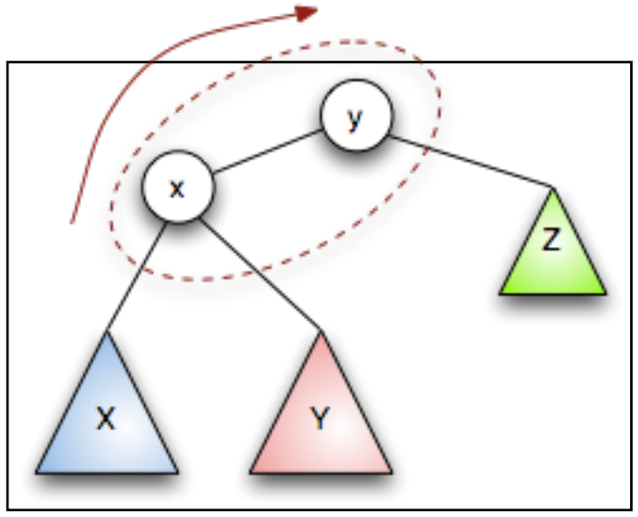


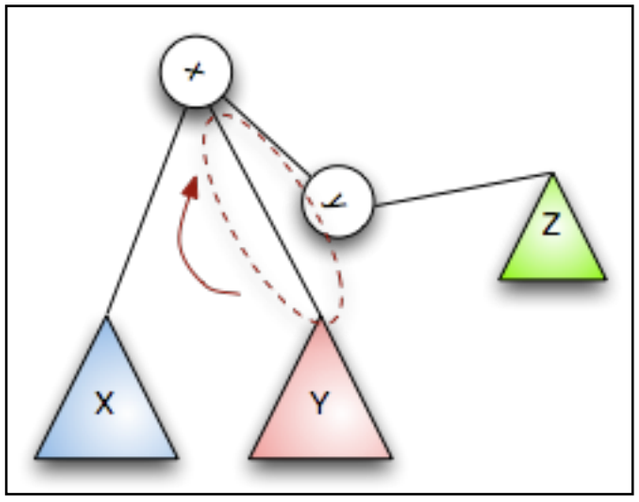


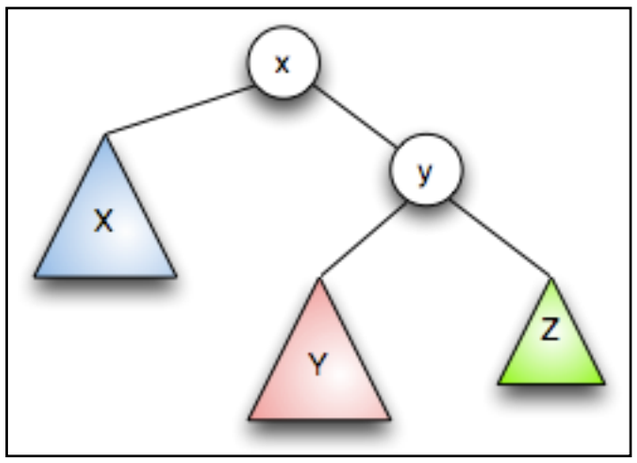
Operaciones en BST: **rotación**

La *rotación derecha* es la operación inversa:

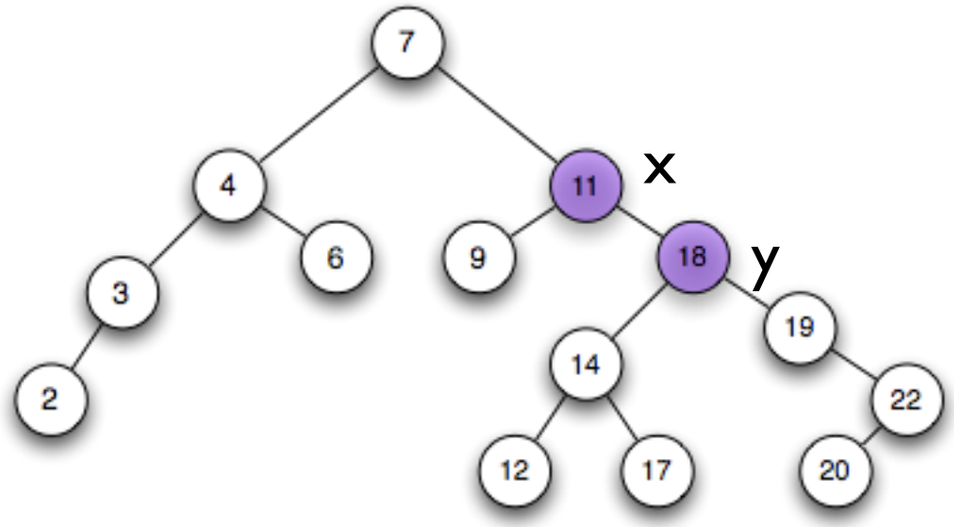
$$A = (y, (x, X, Y), Z) \rightarrow \mathcal{RD}(A) = (x, X, (y, Y, Z)).$$







Operaciones en BST: **rotación**



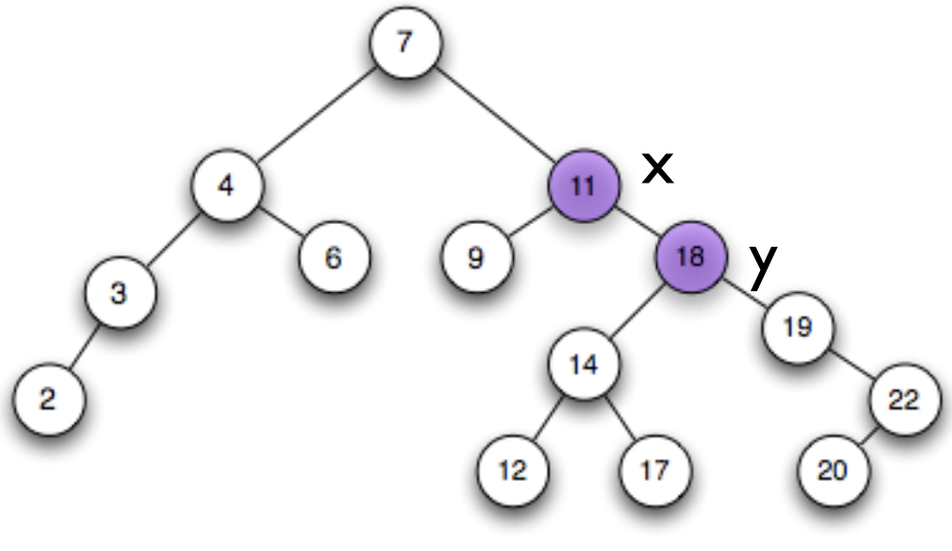
LEFT-ROTATE(T, x)

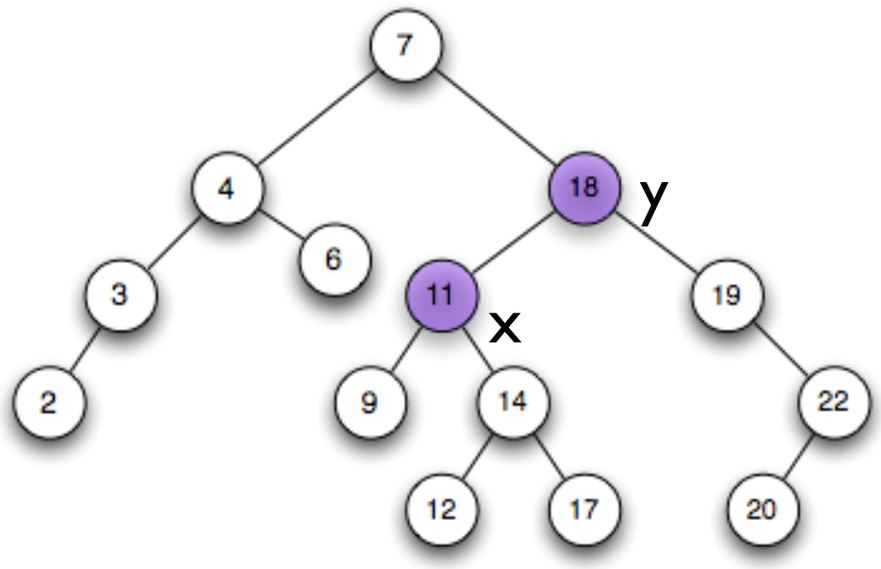
- y ← right[x]
- right[x] ← left[y]

1. if left[y] ≠ NULL
2. then p[left[y]] ← x
3. p[y] ← p[x]
4. if p[x] = NULL

- then root[T] ← y
- else if x = left[p[x]]
- then left[p[x]] ← y
- else right[p[x]] ← y
- left[y] ← x
- p[x] ← y

Operaciones en BST: **rotación**





LEFT-ROTATE(T, x)

- y ← right[x]
- right[x] ← left[y]

1. if left[y] ≠ NULL
2. then p[left[y]] ← x
3. p[y] ← p[x]
4. if p[x] = NULL

- then root[T] ← y
- else if x = left[p[x]]
- then left[p[x]] ← y
- else right[p[x]] ← y
- left[y] ← x
- p[x] ← y

Operaciones en BST: **rotación**

Operaciones en BST: **rotación**

- El código para **RIGHT-ROTATE** es simétrico.

Operaciones en BST: **rotación**

- El código para **RIGHT-ROTATE** es simétrico.
- Ambos, **LEFT-ROTATE** y **RIGHT-ROTATE** tienen un tiempo de ejecución asintótico de O(1).

Operaciones en BST: **rotación**

- El código para **RIGHT-ROTATE** es simétrico.
- Ambos, **LEFT-ROTATE** y **RIGHT-ROTATE** tienen un tiempo de ejecución asintótico de O(1).
- Solo cambian apuntadores durante la rotación, los otros campos de la estructura nodo quedan constantes.

Operaciones en BST: **rotación**

- El código para **RIGHT-ROTATE** es simétrico.
- Ambos, **LEFT-ROTATE** y **RIGHT-ROTATE** tienen un tiempo de ejecución asintótico de O(1).
- Solo cambian apuntadores durante la rotación, los otros campos de la estructura nodo quedan constantes.
- Invariante al recorrido en orden.

Operaciones en BST: **rotación**

- El código para **RIGHT-ROTATE** es simétrico.
- Ambos, **LEFT-ROTATE** y **RIGHT-ROTATE** tienen un tiempo de ejecución asintótico de O(1).
- Solo cambian apuntadores durante la rotación, los otros campos de la estructura nodo quedan constantes.
- Invariante al recorrido en orden.
- Las rotaciones sirven para poder re-equilibrar un BST..

Operaciones en BST: **inserción en la raíz**

Operaciones en BST: **inserción en la raíz**

- Insertar un nodo nuevo y hacerlo la nueva raíz.

Operaciones en BST: **inserción en la raíz**

- Insertar un nodo nuevo y hacerlo la nueva raíz.
- Idea:

Operaciones en BST: **inserción en la raíz**

- Insertar un nodo nuevo y hacerlo la nueva raíz.
- Idea:
  - Insertar el nodo al final

Operaciones en BST: **inserción en la raíz**

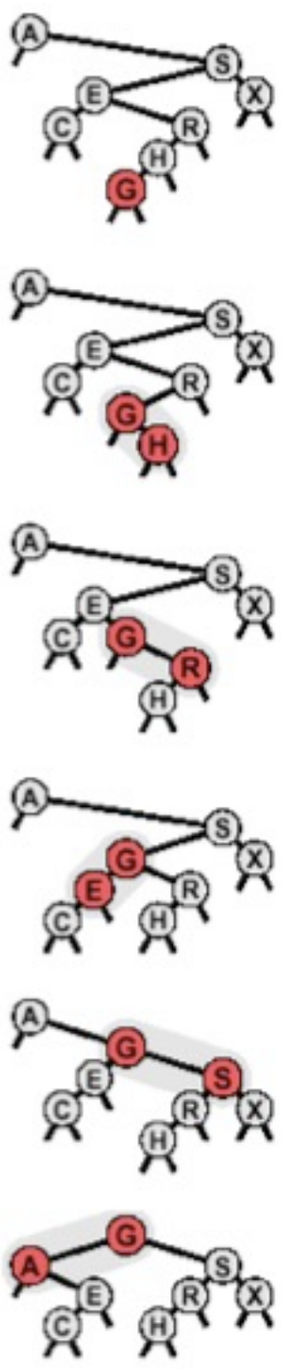
- Insertar un nodo nuevo y hacerlo la nueva raíz.
- Idea:
  - Insertar el nodo al final
  - Rotar el nodo insertado hasta la raíz

Operaciones en BST: **inserción en la raíz**

- Insertar un nodo nuevo y hacerlo la nueva raíz.
- Idea:
  - Insertar el nodo al final
  - Rotar el nodo insertado hasta la raíz
  - Implementación recursiva compacta.

Operaciones en BST: **inserción en la raíz**

- Insertar un nodo nuevo y hacerlo la nueva raíz.
- Idea:
  - Insertar el nodo al final
  - Rotar el nodo insertado hasta la raíz
- Implementación recursiva compacta.

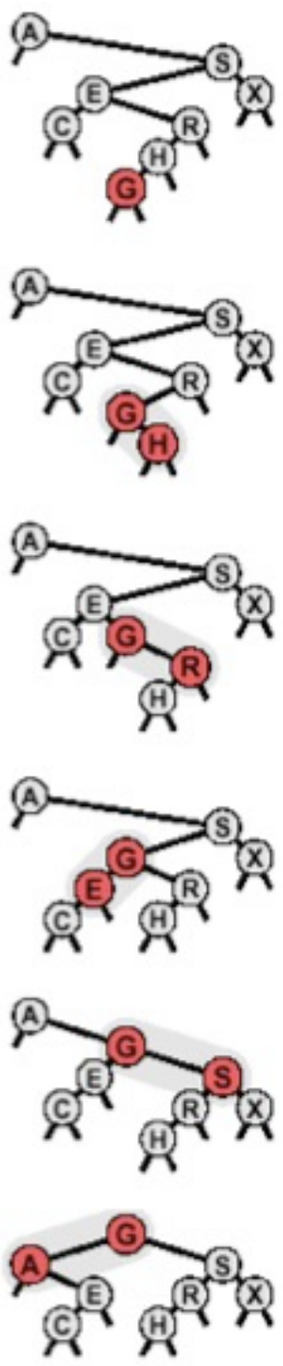


Operaciones en BST: **inserción en la raíz**

```
private:
  void insertT(link& h, Item x)
    { if (h == 0) { h = new node(x); return; }
      if (x.key() <  h->item.key())
           { insertT(h->l, x); rotR(h); }
      else { insertT(h->r, x); rotL(h); }
    }
public:
  void insert(Item item)
    { insertT(head, item); }
```

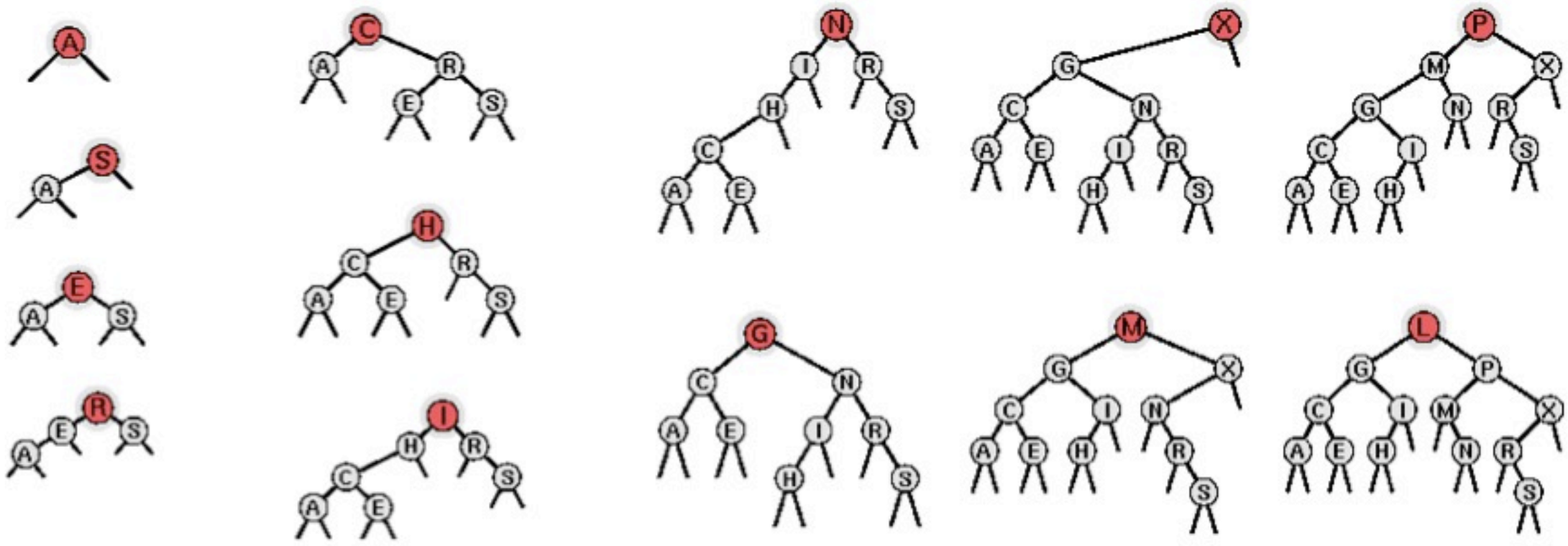
Operaciones en BST: **inserción en la raíz**

```
private:
  void insertT(link& h, Item x)
    { if (h == 0) { h = new node(x); return; }
      if (x.key() <  h->item.key())
           { insertT(h->l, x); rotR(h); }
      else { insertT(h->r, x); rotL(h); }
    }
public:
  void insert(Item item)
    { insertT(head, item); }
```



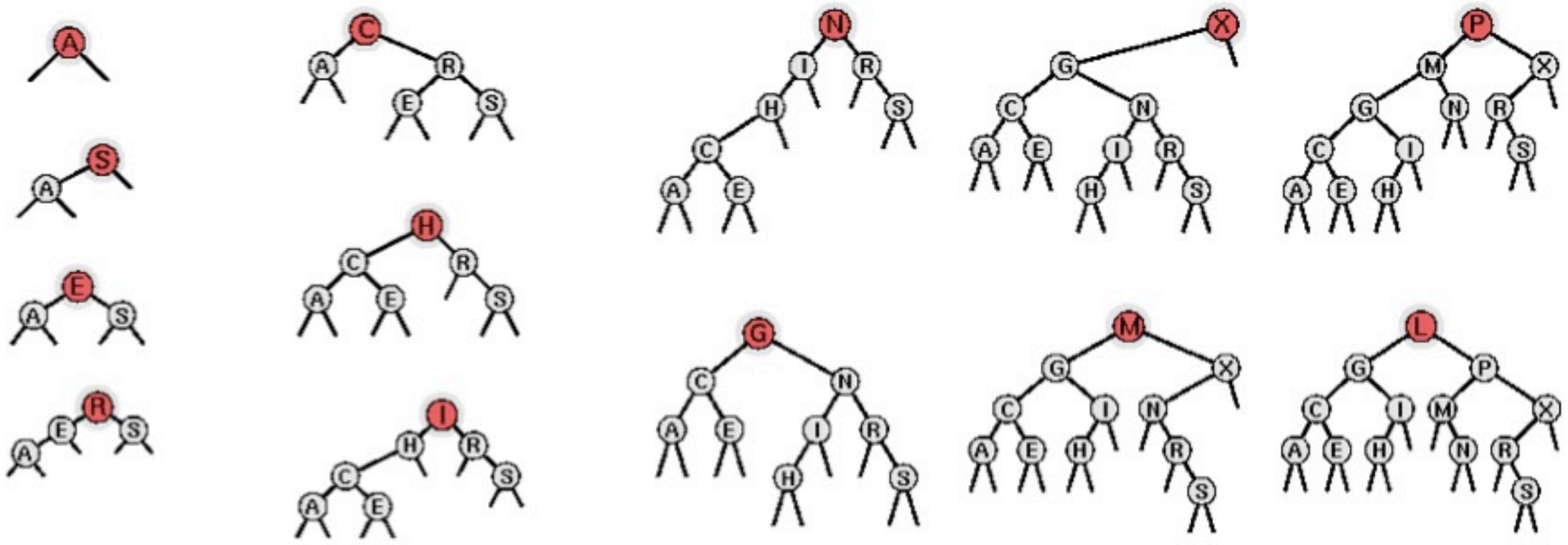
Operaciones en BST: **inserción en la raíz**

A S E R C H I N G X M P L



Operaciones en BST: **inserción en la raíz**

A S E R C H I N G X M P L



- Las llaves insertadas recientemente se encuentran arriba (esto es bueno para algunas aplicaciones)
- También es bueno para una construcción aleatoria.

BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

TREE-INSERT(A)

BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

TREE-INSERT(A)

BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

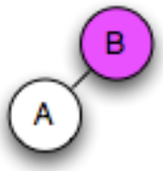
TREE-INSERT(A)

TREE-ROOT-INSERT(B)

BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

TREE-INSERT(A)

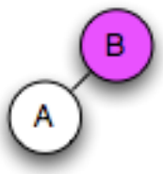


TREE-ROOT-INSERT(B)

BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

TREE-INSERT(A)



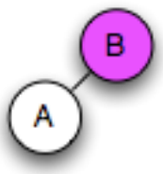
TREE-ROOT-INSERT(B)

TREE-INSERT(C)

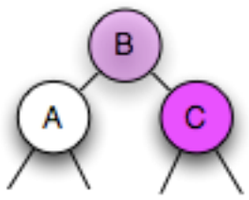
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

TREE-INSERT(A)



TREE-ROOT-INSERT(B)

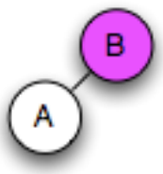


TREE-INSERT(C)

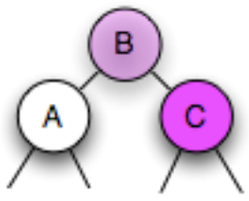
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

TREE-INSERT(A)



TREE-ROOT-INSERT(B)



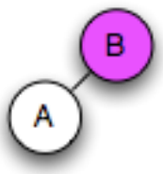
TREE-INSERT(C)

TREE-INSERT(D)

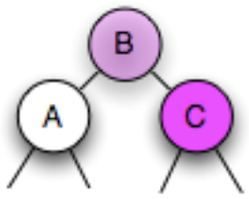
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

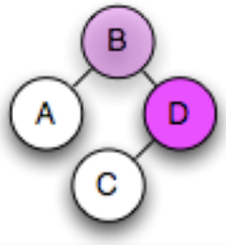
TREE-INSERT(A)



TREE-ROOT-INSERT(B)



TREE-INSERT(C)



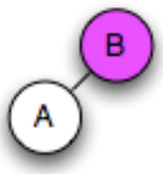
TREE-INSERT(D)

BST de construcción aleatoria

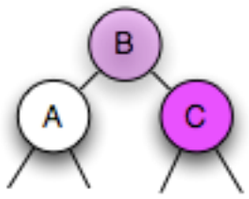
- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

TREE-INSERT(A)

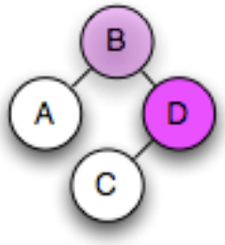
TREE-INSERT(E)



TREE-ROOT-INSERT(B)



TREE-INSERT(C)

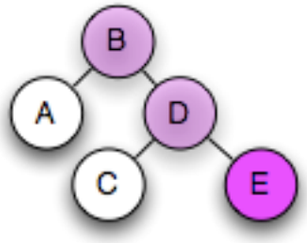


TREE-INSERT(D)

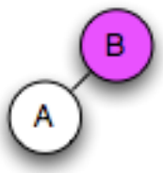
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

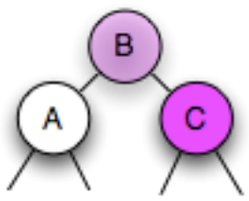
TREE-INSERT(A)



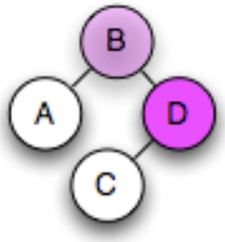
TREE-INSERT(E)



TREE-ROOT-INSERT(B)



TREE-INSERT(C)

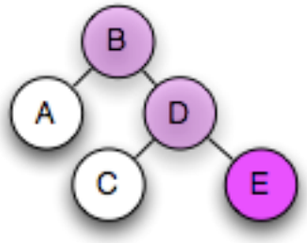


TREE-INSERT(D)

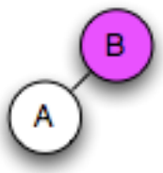
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

TREE-INSERT(A)

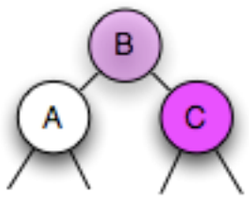


TREE-INSERT(E)

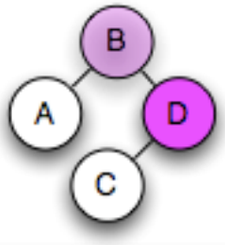


TREE-ROOT-INSERT(B)

TREE-ROOT-INSERT(F)



TREE-INSERT(C)

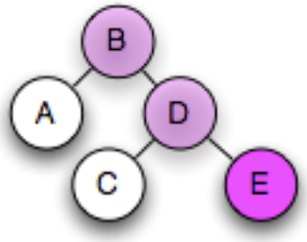


TREE-INSERT(D)

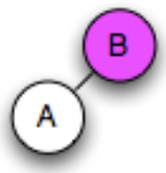
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

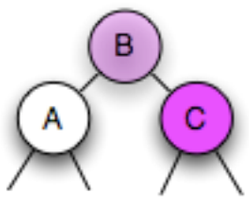
TREE-INSERT(A)



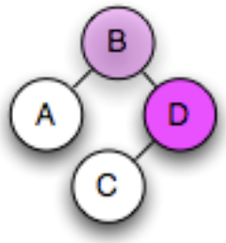
TREE-INSERT(E)



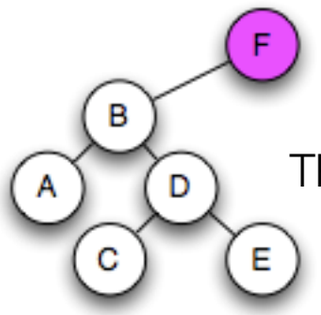
TREE-ROOT-INSERT(B)



TREE-INSERT(C)



TREE-INSERT(D)

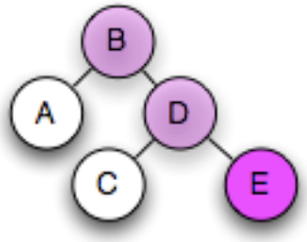


TREE-ROOT-INSERT(F)

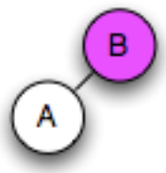
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

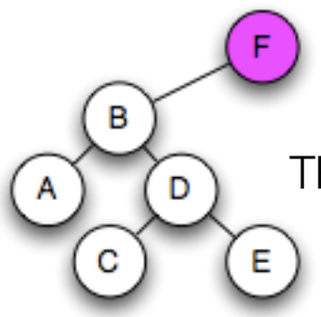
TREE-INSERT(A)



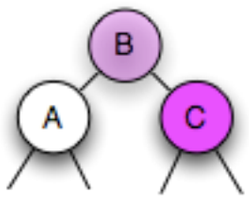
TREE-INSERT(E)



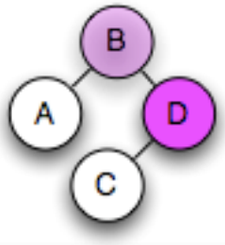
TREE-ROOT-INSERT(B)



TREE-ROOT-INSERT(F)



TREE-INSERT(C)



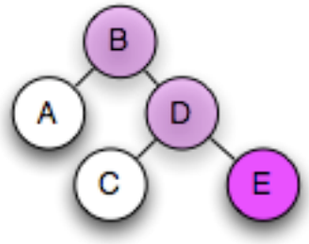
TREE-INSERT(D)

TREE-INSERT(G)

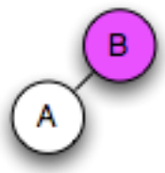
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

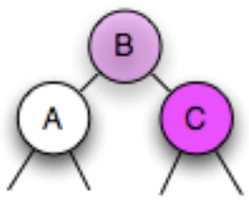
TREE-INSERT(A)



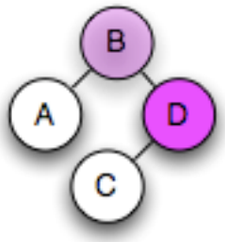
TREE-INSERT(E)



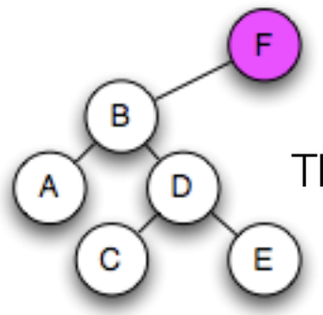
TREE-ROOT-INSERT(B)



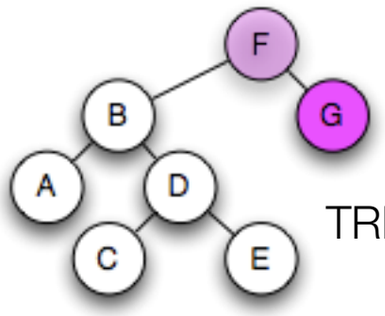
TREE-INSERT(C)



TREE-INSERT(D)



TREE-ROOT-INSERT(F)

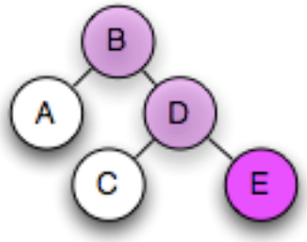


TREE-INSERT(G)

BST de construcción aleatoria

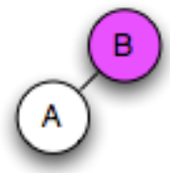
- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

TREE-INSERT(A)

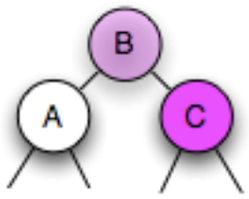


TREE-INSERT(E)

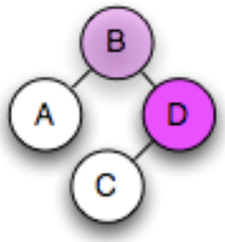
TREE-INSERT(H)



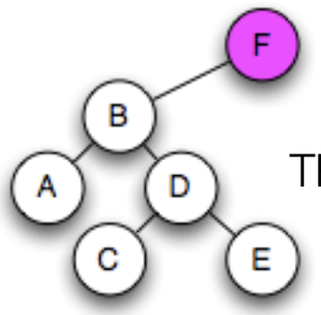
TREE-ROOT-INSERT(B)



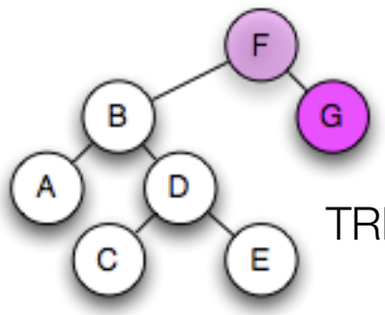
TREE-INSERT(C)



TREE-INSERT(D)



TREE-ROOT-INSERT(F)

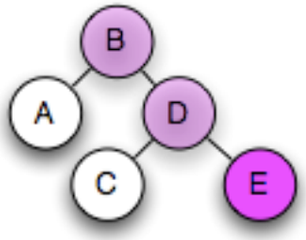


TREE-INSERT(G)

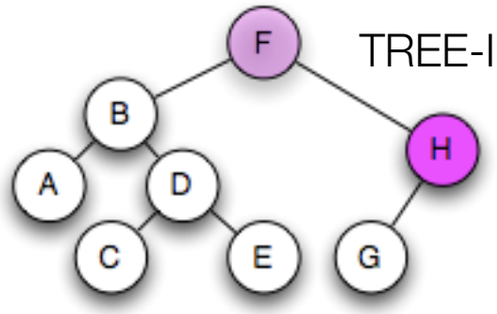
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

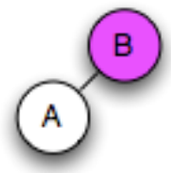
TREE-INSERT(A)



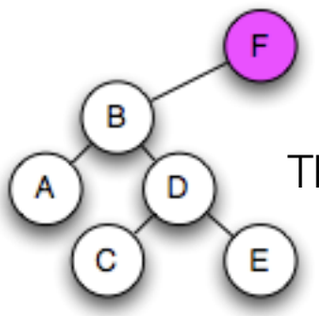
TREE-INSERT(E)



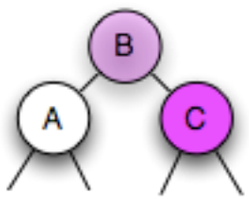
TREE-INSERT(H)



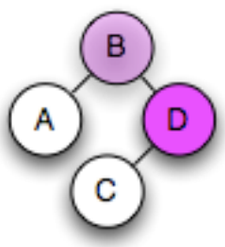
TREE-ROOT-INSERT(B)



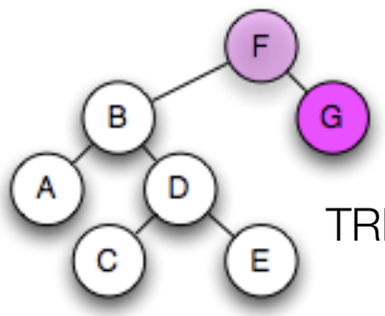
TREE-ROOT-INSERT(F)



TREE-INSERT(C)



TREE-INSERT(D)

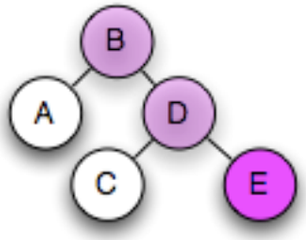


TREE-INSERT(G)

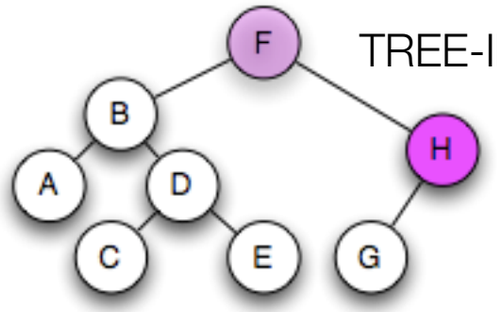
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

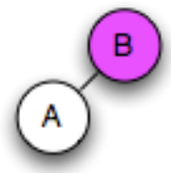
TREE-INSERT(A)



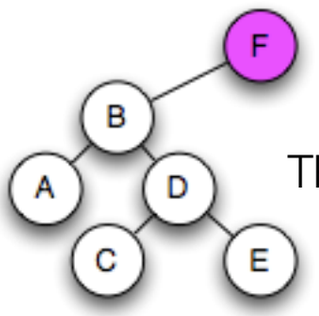
TREE-INSERT(E)



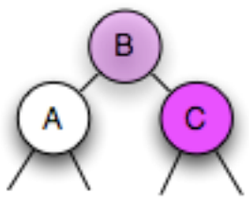
TREE-INSERT(H)



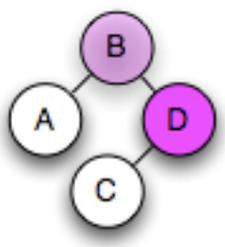
TREE-ROOT-INSERT(B)



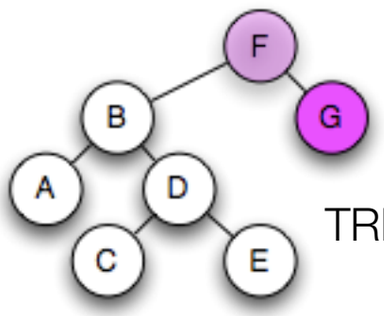
TREE-ROOT-INSERT(F)



TREE-INSERT(C)



TREE-INSERT(D)



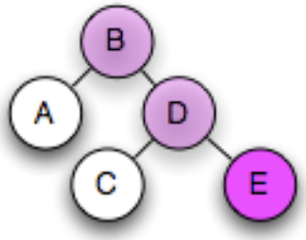
TREE-INSERT(G)

TREE-INSERT(I)

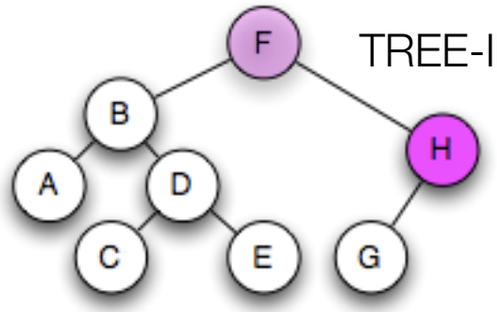
BST de construcción aleatoria

- Un árbol binario de búsqueda construido de forma aleatoria es aquel que resulta de insertar las llaves en orden aleatorio en un árbol inicialmente vacío, y donde cada una de las n! permutaciones tiene la misma probabilidad de ocurrir.

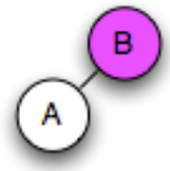
TREE-INSERT(A)



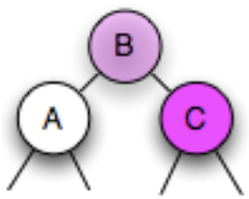
TREE-INSERT(E)



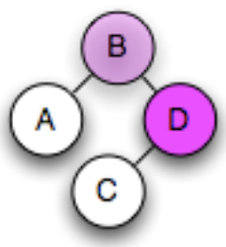
TREE-INSERT(H)



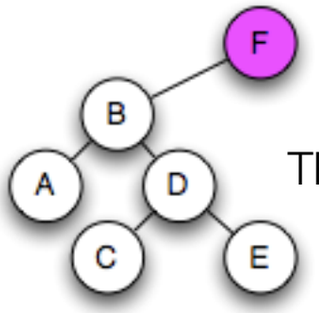
TREE-ROOT-INSERT(B)



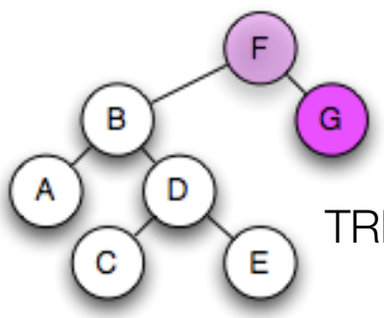
TREE-INSERT(C)



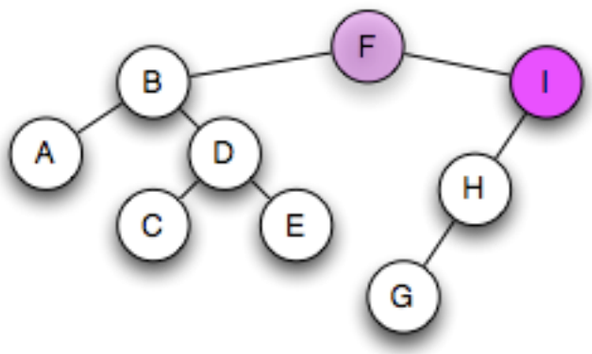
TREE-INSERT(D)



TREE-ROOT-INSERT(F)



TREE-INSERT(G)



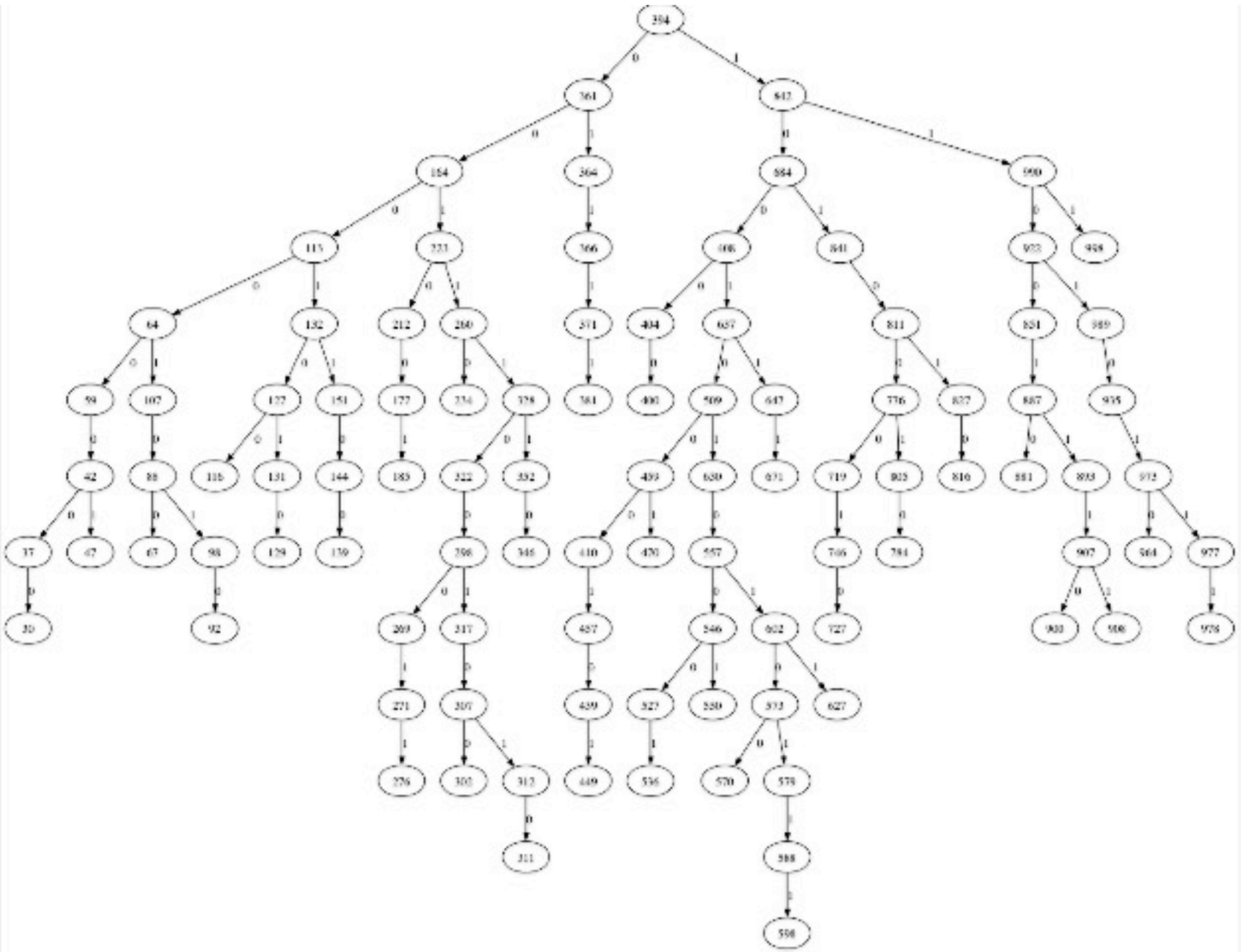
TREE-INSERT(I)

BST de construcción aleatoria

BST de construcción aleatoria

- Hay una probabilidad que el generador de números aleatorios pueda llevar a la mala decisión en cada oportunidad y construir un árbol mal balanceado pero esta probabilidad es muy pequeña.

BST de construcción aleatoria



Resultado de la inserción aleatoria de 100 nodos.

Dado un BST

¿ Cómo implementamos la operación ***select***? (la que regresa el k-ésimo elemento)