

IIC-2133 — Estructuras de Datos y Algoritmos

Árboles Binarios de Búsqueda

Jorge A. Baier

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

Santiago, Chile



Árboles Binarios de Búsqueda (ABB)

- Un ABB se puede ver como un grafo conexo, *dirigido* y acíclico
- Un nodo del ABB T es la *raíz del árbol*; lo anotamos como $root[T]$.
- Si r es la raíz del ABB:
 - 1 $left[r]$ y $right[r]$ denotan los hijos izquierdo y derecho de r , respectivamente (cada uno puede ser NIL)
 - 2 Se entiende que hay un arco entre r y cada uno de sus hijos
 - 3 No hay más arcos que salgan desde r aparte de los descritos arriba
 - 4 Para cada hijo h de r , tenemos que $p[h] = r$
 - 5 De haber hijo izquierdo, $key[left[r]] \leq key[r]$
 - 6 De haber hijo derecho, $key[r] \leq key[right[r]]$
 - 7 Los subárboles izquierdo y derecho son ABB



- El nodo y es hijo de x si $y = left[x]$ o $y = right[x]$
- La relación “alcanzable desde” es la clausura refleja y transitiva de “es hijo de”
- Un subárbol con raíz en x de un ABB A es el subgrafo inducido en A por los nodos alcanzables desde x
- El subárbol izquierdo de x es el subárbol con raíz en $left[x]$.
- El subárbol derecho de x es el subárbol con raíz en $right[x]$.



In-Order, Pre-Order, Post-Order

```
1 procedure Inorder-Tree-Walk( $x$ )
2   if  $x = \text{NIL}$  then return
3   Inorder-Tree-Walk( $\text{left}[x]$ )
4   print  $x$ 
5   Inorder-Tree-Walk( $\text{right}[x]$ )
```

```
1 procedure Preorder-Tree-Walk( $x$ )
2   if  $x = \text{NIL}$  then return
3   print  $x$ 
4   Preorder-Tree-Walk( $\text{left}[x]$ )
5   Preorder-Tree-Walk( $\text{right}[x]$ )
```

```
1 procedure Postorder-Tree-Walk( $x$ )
2   if  $x = \text{NIL}$  then return
3   Postorder-Tree-Walk( $\text{left}[x]$ )
4   Postorder-Tree-Walk( $\text{right}[x]$ )
5   print  $x$ 
```

Propiedad: Si x tiene n nodos bajo él, estas operaciones son $\Theta(n)$.



Buscando un elemento en un árbol

Para buscar un elemento de clave k en un árbol de raíz r hacemos *Iterative-Tree-Search*(r, k).

```
1 procedure Iterative-Tree-Search( $x, k$ )
2   while  $x \neq \text{NIL}$  and  $k \neq \text{key}[x]$  do
3     if  $k < \text{key}[x]$  then
4        $x \leftarrow \text{left}[x]$ 
5     else
6        $x \leftarrow \text{right}[x]$ 
```

Propiedad: Esta operación es



Buscando un elemento en un árbol

Para buscar un elemento de clave k en un árbol de raíz r hacemos *Iterative-Tree-Search*(r, k).

```
1 procedure Iterative-Tree-Search( $x, k$ )
2   while  $x \neq \text{NIL}$  and  $k \neq \text{key}[x]$  do
3     if  $k < \text{key}[x]$  then
4        $x \leftarrow \text{left}[x]$ 
5     else
6        $x \leftarrow \text{right}[x]$ 
```

Propiedad: Esta operación es $O(h)$, donde h es la altura del árbol de raíz x .



Supuesto: z es la celda a agregar y es tal que
 $left[z] = right[z] = \text{NIL}$

```
1 procedure Tree-Insert( $T, z$ )
2    $y \leftarrow \text{NIL}$ 
3    $x \leftarrow root[T]$ 
4   while  $x \neq \text{NIL}$  do
5      $y \leftarrow x$ 
6     if  $key[z] < key[x]$  then  $x \leftarrow left[x]$ 
7     else  $x \leftarrow right[x]$ 
8    $p[z] \leftarrow y$ 
9   if  $y = \text{NIL}$  then  $root[T] \leftarrow z$ 
10  else
11    if  $key[z] < key[y]$  then  $left[y] \leftarrow z$ 
12    else  $right[y] \leftarrow z$ 
```

Propiedad: Tiempo de ejecución es



Supuesto: z es la celda a agregar y es tal que
 $left[z] = right[z] = \text{NIL}$

```
1 procedure Tree-Insert( $T, z$ )
2    $y \leftarrow \text{NIL}$ 
3    $x \leftarrow \text{root}[T]$ 
4   while  $x \neq \text{NIL}$  do
5      $y \leftarrow x$ 
6     if  $\text{key}[z] < \text{key}[x]$  then  $x \leftarrow \text{left}[x]$ 
7     else  $x \leftarrow \text{right}[x]$ 
8    $p[z] \leftarrow y$ 
9   if  $y = \text{NIL}$  then  $\text{root}[T] \leftarrow z$ 
10  else
11    if  $\text{key}[z] < \text{key}[y]$  then  $\text{left}[y] \leftarrow z$ 
12    else  $\text{right}[y] \leftarrow z$ 
```

Propiedad: Tiempo de ejecución es $O(h)$.



Mínimo y Máximo

```
1 function Tree-Min(x)
2   while left[x]  $\neq$  NIL do
3      $x \leftarrow \textit{left}[x]$ 
4   return x
```

```
1 function Tree-Max(x)
2   while right[x]  $\neq$  NIL do
3      $x \leftarrow \textit{right}[x]$ 
4   return x
```



El sucesor de x es el nodo cuya clave es la más pequeña de entre todos los nodos cuya clave es mayor o igual que la de x .



El sucesor de x es el nodo cuya clave es la más pequeña de entre todos los nodos cuya clave es mayor o igual que la de x .

```
1 function Successor( $x$ )
2   if  $right[x] \neq \text{NIL}$  then
3     return Tree-Min( $x$ )
4    $y \leftarrow p[x]$ 
5   while  $y \neq \text{NIL}$  and  $x = right[y]$  do
6      $x \leftarrow y$ 
7      $y \leftarrow p[y]$ 
8   return  $y$ 
```

Propiedad: Tiempo de ejecución $O(h)$

Propiedad: Retorna una hoja ssi x tiene hijo derecho



El predecesor de x es el nodo cuya clave es la más grande de entre todos los nodos cuya clave es menor o igual que la de x .

[Pseudo-código de tarea]



Eliminación

- El caso más sencillo: el nodo a eliminar tiene un solo hijo
- En ese caso, hacemos un *splice out* del nodo, con su hijo:

```
1 procedure Splice-Out( $x, y$ )  
   //  $y$  es el nodo a eliminar;  $x$  su hijo  
2   if  $y = \text{left}[p[y]]$  then  
3      $\text{left}[p[y]] \leftarrow x$   
4   else  
5      $\text{right}[p[y]] \leftarrow x$   
6   if  $x \neq \text{NIL}$  then  
7      $p[x] \leftarrow p[y]$ 
```

- En caso contrario, hacemos *splice out* del sucesor del nodo a eliminar.



Pseudo Código de Eliminación

```
1 procedure Tree-Delete( $T, z$ )  
    //  $y$  es el nodo que será spliced out  
2    if  $left[z] = \text{NIL}$  or  $right[z] = \text{NIL}$  then  
3         $y \leftarrow z$   
4    else  
5         $y \leftarrow \text{Tree-Successor}(z)$   
6    if  $left[y] \neq \text{NIL}$  then  $x \leftarrow left[y]$   
7    else  $x \leftarrow right[y]$   
8    Splice-Out( $x, y$ )  
9    if  $x \neq \text{NIL}$  and  $p[x] = \text{NIL}$  then  $root[T] \leftarrow x$   
10   if  $y \neq z$  then  
11       Copiar datos del nodo  $y$  al nodo  $z$   
12       Eliminar nodo  $y$ 
```



Objetivo: garantizar que operaciones tomen $O(\log n)$

Idea Principal: Usar *rotaciones* para mantener el balance.

[rotaciones presentadas en pizarra]

Tarea: escribir pseudo-código de operaciones Left-Rotate y Right-Rotate



Propiedad de árbol AVL: cada nodo interno v es tal que la altura de sus hijos no difiere en más de 1.

Teorema: La altura de un árbol AVL de n datos es $O(\log n)$.

Punto de partida de la demostración: definir $n(h)$ como el número mínimo de nodos internos de un árbol de altura h .



- 1 El primer paso es ejecutar la operación tal como en un ABB estándar.
- 2 Luego se recorre el árbol hacía arriba, revisando si hay desbalance.
- 3 Si lo hay, sea z el primer nodo desbalanceado, y su hijo de mayor altura, y x el hijo de y de altura mayor.
- 4 Llamar ahora a `restructure(x,y,z)`.
- 5 Continuar al paso 2 hasta llegar a la raíz (en el caso de eliminación).



restructure(x, y, z)

- 1 (a, b, c) denotan a (x, y, z) en in-orden.
- 2 Sean T_0, T_1, T_2, T_3 , de izquierda a derecha, los subárboles de a, b, c que no contienen a a, b, c .
- 3 Reemplazar el árbol cuya raíz es z por otro cuya raíz es b .
- 4 Establecer a y c como hijos izquierdo y derecho de b , respectivamente.
- 5 Establecer a T_0 y T_1 como hijos izq y der (resp) de a .
- 6 Establecer a T_2 y T_3 como hijos izq y der (resp) de c .

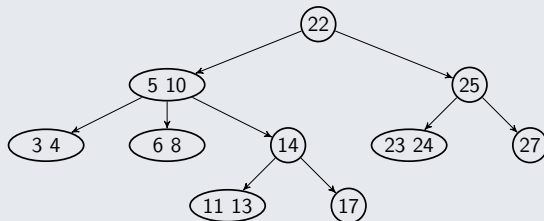


Propiedades:

- 1 reestructure es correcto y tal que el árbol resultante es AVL
- 2 a lo más es necesario un llamado a *restructure* por cada inserción
- 3 el número de llamados a *reestructure* por una eliminación es $O(h)$



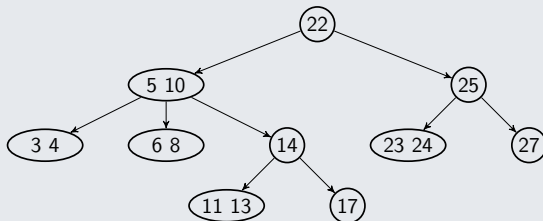
Árboles de Búsqueda *Multi-Way*



Árboles de Búsqueda *Multi-Way*

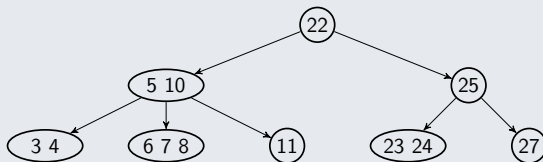
- 1 Contienen d -nodos ($d \geq 2$)
- 2 Cada d -nodo tiene $d - 1$ claves ordenadas (k_1, \dots, k_{d-1}) y d hijos v_1, \dots, v_d
- 3 Dado un d -nodo, si definimos $k_0 = -\infty$ y $k_d = \infty$, toda clave k contenida en el subárbol con raíz v_i es tal que

$$k_{i-1} \leq k \leq k_i$$



Árboles (2,4)

- Son árboles *multi-way* perfectamente balanceados.
- Cumplen las siguientes propiedades:
 - 1 **Tamaño:** Cada nodo interno tiene a lo más 4 hijos
 $d \in \{2, 3, 4\}$
 - 2 **Profundidad:** Cada hoja tiene la misma profundidad.



Propiedad: La altura de un árbol (2,4) con n entradas es $O(\log n)$.



Inserción (ejemplo)

Mostramos en pizarra cómo, al insertar 6,12,15,3,5,10,8, en

④

generamos:

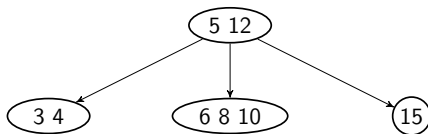


Inserción (ejemplo)

Mostramos en pizarra cómo, al insertar 6,12,15,3,5,10,8, en



generamos:



Algoritmo de Inserción

Suponemos que queremos insertar una nueva clave k .

- 1 Recorrer el árbol de búsqueda hasta encontrar una hoja n en donde insertar k .
- 2 Insertar k en n .
- 3 Si n hace *overflow*, es decir, contiene las claves (k_1, k_2, k_3, k_4) , entonces hacemos un *split* de n , reemplazando n por dos nodos n' y n'' , donde:
 - 1 n' es un 3-nodo con hijos v_1, v_2, v_3 , y almacenando las claves k_1 y k_2 .
 - 2 n'' es un 2-nodo con hijos v_4, v_5 , almacenando a la clave k_4 .
 - 3 Sea u el padre de n' o un nuevo nodo vacío si n' es la raíz.
 - 4 Insertar k_3 dentro de u y hacer que n' y n'' queden colgados de las posiciones correctas.
- 4 Si u hace *overflow*, hacemos $n \leftarrow u$ y volvemos al paso anterior.
- 5 Sea np el padre de n o un nuevo nodo vacío



Algoritmo de Eliminación

Suponemos que queremos eliminar un objeto de clave k .

- 1 Si k no está en una hoja intercambiamos al objeto de clave k con su predecesor (que es una hoja) y luego lo eliminamos desde la hoja.
- 2 Sea v la hoja desde la cual se elimina k .
- 3 Eliminar a k desde v . Si ello causa un *underflow* (viola la propiedad de tamaño):
 - 1 Si v tiene un hermano inmediato que es un 3-nodo o 4-nodo, se ejecuta una operación de transferencia:
una clave del hermano, digamos k , la más cercana a v , pasa hacia el padre, reemplazando a una clave k' del padre que ahora es ubicada en v . El hijo correspondiente a k es ahora ubicado como hijo de v
 - 2 En caso contrario, se hace una *fusión* entre v y un hermano y movemos una de las claves del padre hacia el nodo que resulta de la fusión. Si el padre hace *underflow* se repite el proceso.

