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 $\textsc{Nil}$)
  2. Se entiende que hay un arco entre r y caad 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

Subárboles

- El nodo y es hijo de x si $y = \mathit{left}[x]$ o $y = \mathit{right}[x]$
- 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 $\mathit{left}[x]$.
- El subárbol derecho de x es el subárbol con raíz en $\mathit{right}[x]$.

In-Order, Pre-Order, Post-Order

```
procedure Inorder-Tree-Walk(x)
    if x=NIL then return
    Inorder-Tree-Walk(left[x])
    print x
    Inorder-Tree-Walk(right[x])
```

```
procedure Preorder-Tree-Walk(x)
    if x=NIL then return
    print x
    Preorder-Tree-Walk(left[x])
    Preorder-Tree-Walk(right[x])
```

```
procedure Postorder-Tree-Walk(x)
    if x=NIL then return
    Postorder-Tree-Walk(left[x])
    Postorder-Tree-Walk(right[x])
    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).

```
procedure Iterative-Tree-Search(x, k)
    while x ≠ NIL and k ≠ key[x] do
        if k < key[x] then
            x ← left[x]
        else
            x ← 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).

procedure *Iterative-Tree-Search*(x, k)
    while $x \neq \textsc{Nil}$ *and* $k \neq key[x]$ **do**
        if $k < key[x]$ **then**
            $x \leftarrow left[x]$
        else
            $x \leftarrow right[x]$

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

Inserción

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

```
procedure Tree-Insert(T, z)
    y ← NIL
    x ← root[T]
    while x ≠ NIL do
        y ← x
        if key[z] < key[x] then  x ← left[x]
        else  x ← right[x]
    p[z] ← y
    if y = NIL then  root[T] ← z
    else
        if key[z] < key[y] then  left[y] ← z
        else  right[y] ← z
```

Propiedad: Tiempo de ejecución es

Inserción

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

```
procedure Tree-Insert(T, z)
    y ← NIL
    x ← root[T]
    while x ≠ NIL do
        y ← x
        if key[z] < key[x] then  x ← left[x]
        else  x ← right[x]
    p[z] ← y
    if y = NIL then  root[T] ← z
    else
        if key[z] < key[y] then  left[y] ← z
        else  right[y] ← z
```

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

Mínimo y Máximo

```
function Tree-Min(x)
    while left[x] ≠ NIL do
        x ← left[x]
    return x
```

```
function Tree-Max(x)
    while right[x] ≠ NIL do
        x ← right[x]
    return x
```

Sucesor

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.

Sucesor

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.

function *Successor*(x)
  if $right[x] \neq \textsc{Nil}$ **then**
    return *Tree-Min*(x)
  $y \leftarrow p[x]$
  while $y \neq \textsc{Nil}$ *and* $x = right[y]$ **do**
    $x \leftarrow y$
    $y \leftarrow p[y]$
  return y

Propiedad: Tiempo de ejecución $O(h)$
Propiedad: Retorna una hoja ssi x tiene hijo derecho

Predecesor

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:

procedure *Splice-Out*(x, y)
    `//` y `es el nodo a eliminar;` x `su hijo`
    if $y = \mathit{left}[p[y]]$ **then**
        $\mathit{left}[p[y]] \leftarrow x$
    else
        $\mathit{right}[p[y]] \leftarrow x$
    if $x \neq \textsc{Nil}$ **then**
        $p[x] \leftarrow p[y]$

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

Pseudo Código de Eliminación

```
procedure Tree-Delete(T, z)
    // y es el nodo que será spliced out
    if left[z] = NIL or right[z] = NIL then
        y ← z
    else
        y ← Tree-Successor(z)
    if left[y] ≠ NIL then  x ← left[y]
    else x ← right[y]
    Splice-Out(x, y)
    if x ≠ NIL and p[x] = NIL then root[T] ← x
    if y ≠ z then
        Copiar datos del nodo y al nodo z
        Eliminar nodo y
```

Árboles Balanceados

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

Árboles AVL

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.

Inserción y Eliminación

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).

Operación restructure

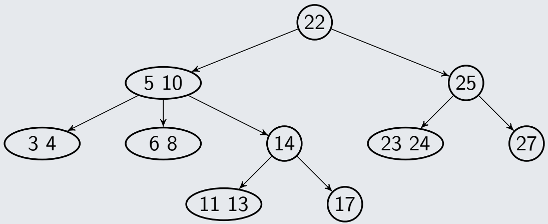
restructure(x,y,z)

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 $O(h)$

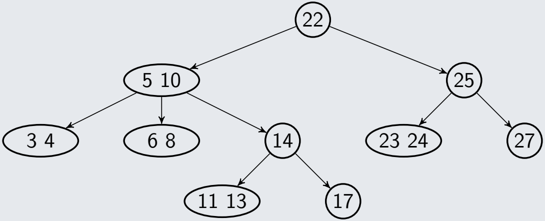
Árboles de Búsqueda *Multi-Way*



Árboles de Búsqueda *Multi-Way*

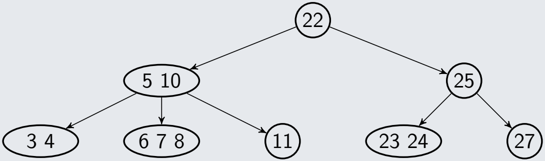
1. Contienen d-nodos ($d \geq 2$)
2. Cada d-nodo tiene $d-1$ claves ordenadas $(k_1, \ldots, k_{d-1})$ y d hijos $v_1, \ldots, v_d$
3. Dado un d-nodo, si definimos $k_0 = -\infty$ y $k_d = \infty$, toda clave k contenida en el subárbol con raíz v_i es tal que

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



Árboles (2,4)

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

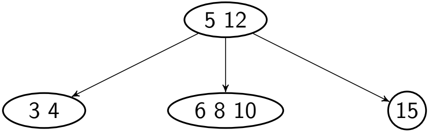
Mostramos en pizarra cómo, al insertar 6,12,15,3,5,10,8, en

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

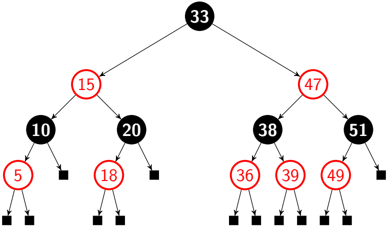
Algoritmo de Eliminación

Suponemos que queremos eliminar un objeto de clave k.

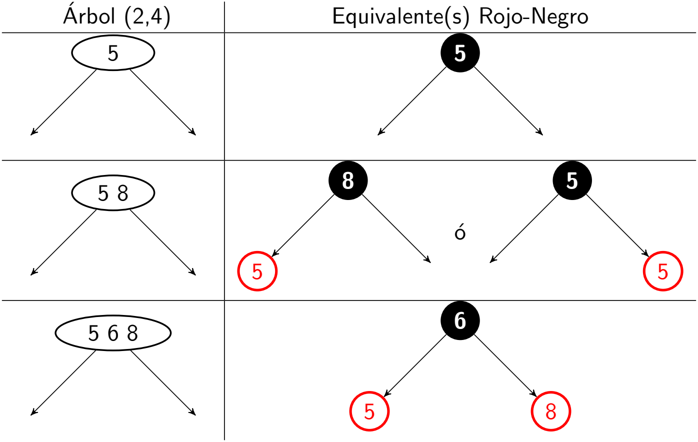
1. Si k no está en una hoja intercambiamos al objeto de clave k con su predecesor (que es una hoja) y luego lo eliminamos desde la hoja.
2. Sea v la hoja desde la cual se elimina k.
3. Eliminar a k desde v. Si ello causa un *underflow* (viola la propiedad de tamaño):
    1. Si v tiene un hermano inmediato 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 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.

Árboles Rojo-Negro

- Son árboles binarios de búsqueda balanceados con una correspondencia directa con los árboles (2,4).
- Satisfacen las siguientes propiedades:
  1. **Color**: Cada nodo es rojo o negro.
  2. **Raíz**: la raíz es negra.
  3. **Hojas**: Cada hoja (nula) es negra.
  4. **Nodos internos**: Los hijos de nodos rojos son negros.
  5. **Profundidad**: Cada rama contiene el mismo número de nodos negros.



Correspondencia con Árboles (2,4)



Propiedad

Theorem

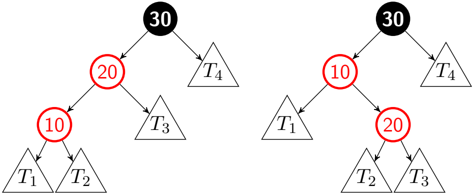
La altura de un árbol rojo-negro con n entradas es $O(\log n)$.

Inserción

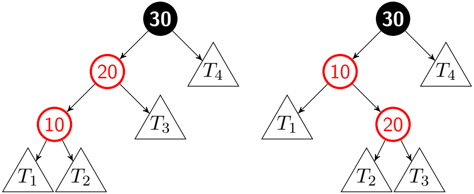
1. Se inserta el nuevo nodo en forma estándar.
2. Si el nuevo nodo quedó en la raíz se pinta rojo. En caso contrario se pinta negro.
3. Se pintan sus hojas nil de color negro.
4. Si se viola una propiedad (dos casos), arreglar el problema.

Inserción: Caso 1 (tío del nodo conflictivo es negro)

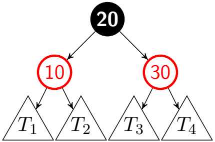
En este ejemplo, acabamos de insertar 10 (y T_4 tiene raíz negra)



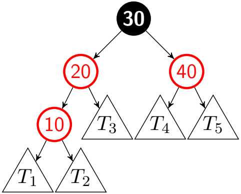
En este ejemplo, acabamos de insertar 10 (y T_4 tiene raíz negra)



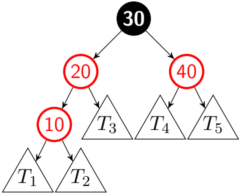
Se aplica la operación *restructure* para obtener:



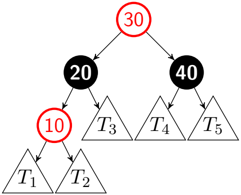
Inserción: Caso 2 (tío del nodo conflictivo es rojo)



Inserción: Caso 2 (tío del nodo conflictivo es rojo)



Simplemente se re-colorean los nodos, obteniéndose:



... y si esto causa otra violación se llama el procedimiento recursivamente. Si la raíz es pintada roja, se vuelve a pintar negra.