## Árboles

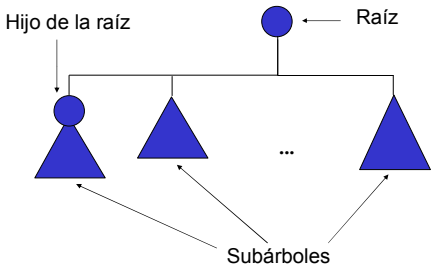
Una estructura árbol (general) con tipo base T es,

1. O bien la estructura vacía
2. O bien un nodo de tipo T, llamado raíz del árbol, junto con un número finito de estructuras de árbol, de tipo base T, disjuntas, llamadas *subárboles*

Si el número de estructuras es siempre dos, el árbol se llama binario. Según la siguiente representación, uno de los subárboles será el izquierdo y el otro el derecho.

1

## Una posible representación




Los elementos se llaman habitualmente nodos del árbol.

2

## Árbol binario de búsqueda

Un árbol binario de búsqueda es un árbol binario que cumple la siguiente propiedad: para cada nodo $n_i$, todos los elementos en el subárbol izquierdo de $n_i$ son menores que el elemento de $n_i$, y todos los elementos en el subárbol derecho de $n_i$ son mayores que los elementos de $n_i$.

Los elementos en los nodos de un ABB pueden ser números, caracteres, o estructuras complejas.

Para ello se requiere un orden total definido sobre el conjunto de los elementos.

3

## Orden total

Un orden total en el conjunto de elementos (S) contenidos en el árbol es una relación (=<) tal que:

(refl.) a =< a para todo a en S

(antis.) si a =< b y b =< a, entonces a = b para todos a, b en S

(trans.) si a =< b y b =< c, entonces a =< c para todos a, b, c en S

(lineal) para todos a, b en S, o bien a =< b o b =< a

4

## Orden total finito

Dado que, en nuestro caso, el árbol es finito, existe un "primer" elemento (mínimo). Análogamente, existe un último elemento.

El orden total permite definir las siguientes operaciones:

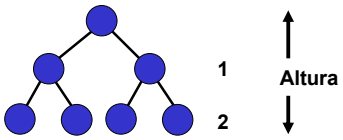
```
Minimo();
Maximo();
Predecesor(x);
Sucesor(x);
```

5

## Ventaja ABB

Permite el ordenamiento y búsqueda de datos en forma simple y eficiente.

Consideremos un AB (o ABB) completo:




Cantidad de nodos en un árbol binario perfecto = n
$= 1 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$

6

## Ventaja ABB

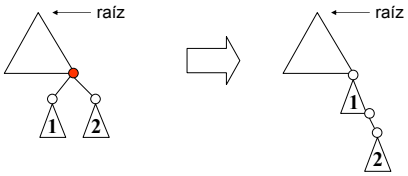
Luego, la altura de dicho árbol es $O(\log_2 n)$. Pero, en un árbol arbitrario la altura es O(n). (cuando el árbol degenera en una lista).

Insertar, Buscar y Borrar tienen tiempo de ejecución proporcional a $\log_2 n$ (en cada paso, se toma uno de los dos caminos, con lo cual el espacio de búsqueda se biparticiona recursivamente), si el árbol es completo. Pero: la eficiencia puede caer a orden n si el árbol degenera (a una lista).



## Eliminar elemento

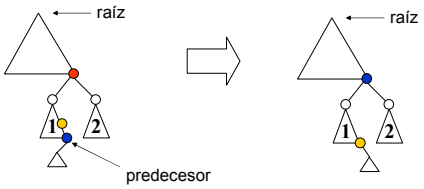
Eliminar "por mezcla"






## Eliminar elemento (2)

Eliminar "por copia"






## Implementación

En un ABB los elementos poseen, en general, dos atributos distinguidos: un **campo clave** (por el cual se ordenan los elementos) y un **campo de información** (de datos). Nosotros asumiremos que ambos campos son indistinguibles, ya que en C++ se pueden sobrecargar los operadores <, > e ==



```
#ifndef ABB_H
#define ABB_H

template <class T>
class ABB
{
public:
    /* CONSTRUCTORAS */
    virtual void Vacio() = 0;
    virtual void Insertar(const T &x) = 0;

    /* PREDICADOS */
    virtual bool EsVacio() = 0;
    virtual bool Esta(const T&) = 0;

    /* DESTRUCTORAS */
    // pre :
    // post: retorna el ABB del que se ha borrado el elemento con clave x (si
    está).
    virtual void Eliminar(const T &x) = 0;

    // pre : árbol no vacio
    // post: retorna el máximo elemento del ABB
    virtual T Maximo() = 0;

    // pre : árbol no vacio
    // post: retorna el minimo elemento del ABB
    virtual T Minimo() = 0;
};

#endif
```

**x es de sólo lectura**

**x es una referencia, por lo tanto, podría modificarse (en este caso no porque es const)**
**Se puede ver como un parámetro de salida.**
**Al ser una referencia, no hay copia.**



```
#ifndef ABBIMP_H
#define ABBIMP_H

#include "ABB.h"
#include <assert.h>

template<class T>
class ABBImp : public ABB<T>
{
private:
   //...

public:
   ABBImp();
   ~ABBImp();

   //... Interfaz
};

#include "ABBImp.cpp"

#endif
```

## private:

```
struct Nodo
{
    T dato;
    Nodo* izq;
    Nodo* der;
};

Nodo* raiz;

void Eliminar(Nodo*);
void Insertar(const T&, Nodo*&);
void Eliminar(const T&, Nodo*&);
void Eliminar1(const T&, Nodo*&);
```



```
Nodo* Maximo(Nodo* abb)
{
      if (!EsVacio(abb->der))
            return abb->der;
      else
            return abb;
}

Nodo* Minimo(Nodo* abb)
{
      if (!EsVacio(abb->izq))
            return abb->izq;
      else
            return abb;
}

bool EsVacio(Nodo* nodo)
{
      return nodo == 0;
}

Nodo* Buscar(const T &x, Nodo* abb)
{
      if (EsVacio(abb))
            return 0;
      else if (abb->dato < x)
            return Buscar(x, abb->der);
      else if (abb->dato > x)
            return Buscar(x, abb->izq);
      else return abb;
}
```



## public: (interfaz)

```
/* CONSTRUCTORAS */
void Vacio() { Eliminar(raiz); raiz = 0; }
void Insertar(const T &x);

/* PREDICADOS */
bool EsVacio() { return raiz == 0; }
bool Esta(const T&);

/* DESTRUCTORAS */
void Eliminar(const T &x);

/* SELECTORAS */
T Maximo();
T Minimo();
```



## .cpp

```
template<class T>
void
ABBImp<T>::Eliminar(Nodo* abb)
{
  if (!EsVacio(abb))
  {
      Eliminar(abb->izq);
      Eliminar(abb->der);
      delete abb;
      abb = 0;
  }
}
```



## .cpp (2)

```
template<class T>
void
ABBImp<T>::Insertar(const T &x, Nodo* &abb)
{
   if (!EsVacio(abb))
   {
        if (abb->dato > x)
              Insertar(x, abb->izq);
        else if (abb->dato < x)
              Insertar(x, abb->der);
   }
   else
   {
        abb = new Nodo;
        abb->izq = 0;
        abb->der = 0;
        abb->dato = x;
   }
}
```



## Por “copia”

```
template<class T>
void
ABBImp<T>::Eliminar(const T& x, Nodo* &abb)
{
   if (!EsVacio(abb))
   {
        if (abb->dato < x)
              Eliminar(x, abb->der);
        else if (abb->dato > x)
              Eliminar(x, abb->izq);
        else
        {
              if (!EsVacio(abb->der) && !EsVacio(abb->izq))
              {
                    Nodo* pred = Maximo(abb->izq);
                    abb->dato = pred->dato;
                    Eliminar(pred->dato, abb->izq);
              }
              else
              {
                    Nodo* temp = abb;
                    if (EsVacio(abb->der))
                          abb = abb->izq;
                    else
                          abb = abb->der;
                    delete temp;
              }
        }
   }
}
```

## Por “mezcla”

```
Nodo* pred = Maximo(abb->izq);
pred->der = abb->der;
Nodo* temp = abb;
abb = abb->izq;
delete temp;
```

19

## Recorridas

2 tipos:
- BFS (amplitud)
- DFS (profundidad)

A su vez hay 3 formas de DFS:
- Preorden
- Enorden
- Postorden

20

## Enorden recursiva (.cpp)

```
template<class T>
void
ABBImp<T>::EnOrden(Nodo* abb)
{
   if (!EsVacio(abb))
   {
        EnOrden(abb->izq);
        cout << abb->dato << endl;
        EnOrden(abb->der);
   }
}

template<class T>
void
ABBImp<T>::EnOrden()
{
   EnOrden(raiz);
}
```

21

## Enorden recursiva (.h)

```
public:
  void EnOrden();

private:
  void EnOrden(Nodo* abb);
```

22

## Árboles binarios balanceados

Siempre es deseable que el árbol esté balanceado, es decir, que se utilicen de forma “similar” ambos subárboles.

En particular, es deseable incorporar alguna restricción en cuanto a la forma de crecimiento.

Un árbol es perfectamente balanceado si para cada nodo, el número de nodos de sus subárboles difieren como máximo en uno.

Dichos árboles tienen altura mínima.

23

## Balanceo perfecto (binario)

```
template<class T>
void
ABBImp<T>::Balanceado(T datos[], int ini, int
  fin)
{
  if (ini <= fin)
  {
      int medio = (ini+fin) / 2;
      Insertar(datos[medio-1], raiz);
      Balanceado(datos, ini, medio-1);
      Balanceado(datos, medio+1, fin);
  }
}
```

24

## Balanceo perfecto (binario) (2)

Implica la reorganización del árbol en cada inserción.

Necesitamos un compromiso entre la eficiencia de búsqueda y el costo de rebalanceo.

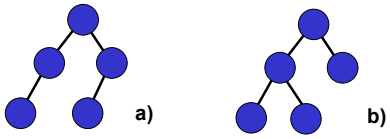
25

## Árboles AVL

Criterio propuesto por Adelson-Velskii y Landis:

Un árbol binario es AVL si y sólo si en cada nodo las alturas de sus dos subárboles difieren como máximo en 1.

Todos los APB son AVL, pero el recíproco no es cierto, luego esta condición es más débil que la anterior.




26

## Inserción en AVL

Además del criterio de búsqueda, se debe cumplir el criterio de equilibrio.

Una vez realizada la inserción será necesario analizar si se satisface el criterio de equilibrio, y en caso de ser necesario se deben realizar las manipulaciones adecuadas.

Lo mejor es analizar todas las situaciones posibles.

27

## Inserción en AVL (2)

Supongamos que tenemos un AVL, y que se va a insertar un nuevo elemento (Nn) en el nodo N (padre) y con subárboles I y D, de alturas h(I) y h(D), respectivamente.

Antes de la inserción, puede suceder que (por ser AVL):

h(I) = h(D)

h(I) < h(D)

h(I) > h(D)

Supongamos que Nn se inserta en I,

28

## Inserción en AVL (3)

entonces,

Si h(I) = h(D), entonces h(I) > h(D)

Si h(I) < h(D), entonces h(I) = h(D)

Si h(I) > h(D), entonces no será AVL, es necesario rebalanceo

En general analizamos dos cuestiones fundamentales:

- En qué casos es necesario rebalancear?
- Cómo rebalanceo?

29

## Inserción en AVL (4)

Definimos bal(N) = h(D) – h(I)

Para rebalancear necesitamos guardar información sobre el balance de cada nodo:

```
struct Nodo
{
   T dato;
   int bal;
   Nodo* izq;
   Nodo* der;
};
```

30

## Inserción por la izquierda de N

La inserción por la derecha puede determinarse por analogía (el problema es simétrico)

- Cuando es necesario rebalancear?

Cuando bal(N) = -2, para algún nodo N

(consideramos N al nodo más profundo que antes de la inserción tuviese un bal = -1 , ya que si fuese AVL, también lo serían sus ascendientes)

31

## Cuando es necesario rebalancear?

Sea Nl el hijo izquierdo de N.

Cuando se inserta Nn por la izquierda de N, se pueden dar dos casos:

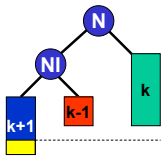
1) Se inserta a la izquierda de Nl

2) a la derecha

32

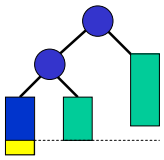
## 1) a la izquierda de Nl

a)



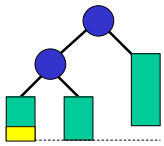

**Antes:**
bal(N) = -1
bal(Nl) = -1
**Después:**
bal(N) = -2
bal(Nl) = -2

b)



**Antes:**
bal(N) = -1
bal(Nl) = 0
**Después:**
bal(N) = -2
bal(Nl) = -1

c)



**Antes:**
bal(N) = -1
bal(Nl) = 1
**Después:**
bal(N) = -1
bal(Nl) = 0

33

## 1) a la izquierda de Nl

a) Si antes de la inserción bal(Nl) = - 1, el nodo a rebalancear sería Nl y no N (se debe considerar el nodo más profundo)

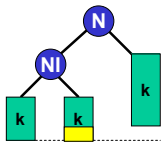
b) Si antes de la inserción bal(Nl) = 0, hay necesidad de rebalancear a N

c) Si antes de la inserción bal(Nl) = 1, no hay necesidad de rebalancear a N

34

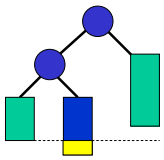
## 2) a la derecha de Nl

a')



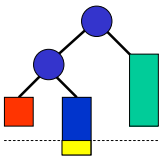

**Antes:**
bal(N) = -1
bal(Nl) = -1
**Después:**
bal(N) = -1
bal(Nl) = 0

b')



**Antes:**
bal(N) = -1
bal(Nl) = 0
**Después:**
bal(N) = -2
bal(Nl) = 1

c')



**Antes:**
bal(N) = -1
bal(Nl) = 1
**Después:**
bal(N) = -2
bal(Nl) = 2

35

## 2) a la derecha de Nl

a) Si antes de la inserción bal(Nl) = -1, no hay necesidad de rebalancear a N

b) Si antes de la inserción bal(Nl) = 0, hay necesidad de rebalancear a N

c) Si antes de la inserción bal(Nl) = 1, el nodo a rebalancear sería Nl y no N

36

## Resumen

En resumen, la situación a tener en cuenta es cuando

bal(N) = -1 y bal(NI) = 0

Independientemente de si la inserción se hace por izquierda o derecha de NI

37

## Cómo rebalanceo?

Segunda cuestión fundamental:

- Cómo rebalanceo?

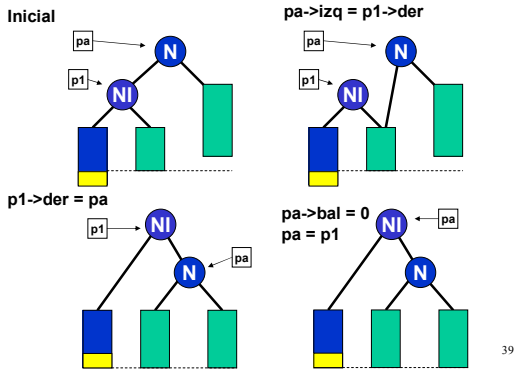
En este caso, los rebalanceos son I, porque el árbol está sobrecargado hacia la izquierda.

Se pueden (otra vez) dar dos casos:

1) Se inserta a la izquierda de NI (rebalanceo II)
2) a la derecha (rebalanceo ID)

38

## a la izquierda de NI (rebalanceo II)

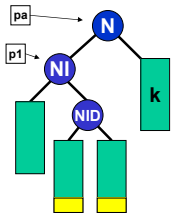



39

## a la derecha de NI (rebalanceo ID)

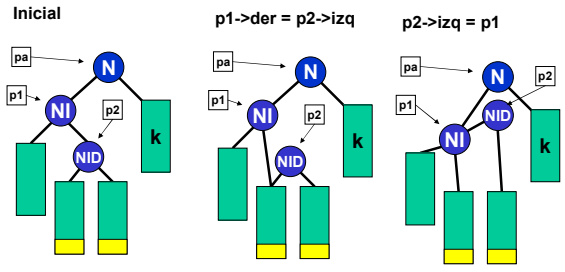
No es tan sencillo. Si rebalanceo igual que antes, empeoro la situación. Por ello, tengo que discriminar el hijo derecho de NI. La idea es disminuir la altura de ambos hijos de NI.

Para ello, debemos colocar a NI como hijo izquierdo de NID y a N como hijo derecho.



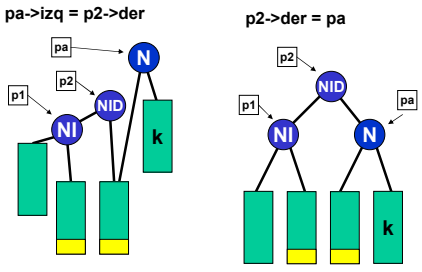

40

## a la derecha de NI (rebalanceo ID)




41

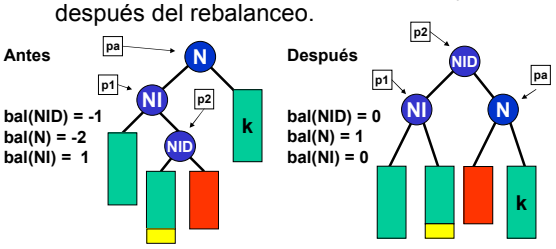
## a la derecha de NI (rebalanceo ID) (2)




42

## a la derecha de NI (rebalanceo ID) (3)

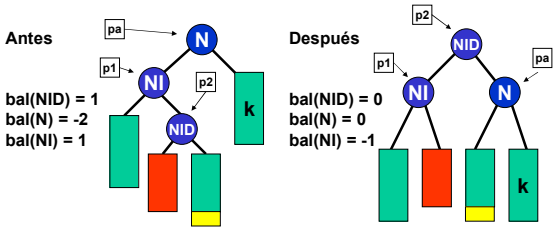
Resta ajustar el balance. Debemos tener en cuenta las posibles situaciones antes y después del rebalanceo.




```
pa->bal = (p2->bal == -1) ?  1 : 0;
p1->bal = (p2->bal ==  1) ? -1 : 0;
```

43

## a la derecha de NI (rebalanceo ID) (4)




```
pa->bal = (p2->bal == -1) ?  1 : 0;
p1->bal = (p2->bal ==  1) ? -1 : 0;
```

44

## a la derecha de NI (rebalanceo ID) (5)

Luego resta setear el padre, y poner en 0 su balance:

```
pa = p2;

pa->bal = 0;
```

45

## Inserción completa

El proceso de inserción consta de 3 partes:

1) Buscar siguiendo la trayectoria normal de búsqueda binaria.

2) Insertar el nodo y determinar su balance.

3) Retroceder (backtrack) y determinar el factor de balance en cada caso. Realizar rebalanceo en caso de ser necesario.

46

## Inserción completa (retroceso)

Para 3) necesitamos evaluar, luego de la inserción, si la altura del árbol varió. Para ello usamos una variable booleana (`vario_h`) que es "propagada" hacia los ascendientes, en caso de ser necesario. (Se declara en la clase implementación, tiene los efectos de una variable global).

47

## Inserción completa (retroceso) (2)

Cuando en el nivel de recursión en el que se creó un nodo, apuntado por `p` se recibe la información de que la altura varió, pueden darse 3 casos si el nodo se insertó por la izquierda (y otros 3 simétricos si se insertó por la derecha).

Por la izquierda:

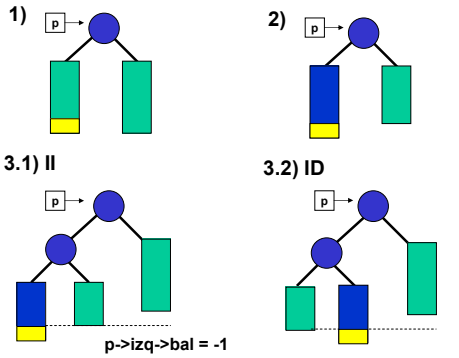
1) bal(`p`) = 1 , ahora debe ser 0. La altura no cambia.
2) bal(`p`) = 0 , ahora es -1. La altura cambia.
3) bal(`p`) = -1 se debe realizar un rebalanceo, que puede ser II o ID.

Por la derecha:

1) bal(`p`) = -1 , ahora debe ser 0. La altura no cambia.
2) bal(`p`) = 0 , ahora es 1. La altura cambia.
3) bal(`p`) = 1 se debe realizar un rebalanceo (DD ó DI)

48

## Inserción completa (retroceso) (3) izquierda




49

## Inserción completa (implementación)

```
template <class T>
void AVLImp<T>::Insertar(avl &pa, const T& x)
{
   avl p1, p2;

   if (pa == NULL)
   {    /* se da el alta */
        pa = new Nodo;
        pa->dato = x;
        pa->bal = 0;
        pa->izq = pa->der = NULL;
        vario_h = true;
   }
   else if (pa->dato > x)
        /* insertar en subarbol izq: menores */

   else if (pa->dato < x)
        /* insertar en subarbol der: mayores */

   else
        /* el nombre ya esta en el arbol */
        vario_h = false;
}
```

50

```
Insertar(pa->izq, e);
if (vario_h)  /* crecio altura de subarbol izquierdo */
    switch(pa->bal)
    {
          case 1:
                pa->bal = 0;
                vario_h = false;
                break;
          case 0:
                pa->bal= -1;
                break;
          case -1:
                p1 = pa->izq;
                if (p1->bal == -1)
                {          /* II */
                      pa->izq = p1->der;
                      p1->der = pa;
                      pa->bal = 0;
                      pa = p1;
                }
                else
                {          /* ID */
                      p2 = p1 ->der;
                      p1->der = p2->izq;
                      p2->izq = p1;
                      pa->izq = p2->der;
                      p2->der = pa;
                      pa->bal = p2->bal == -1 ?  1 : 0;
                      p1->bal = p2->bal ==  1 ? -1 : 0;
                      p = p2;
                }
                pa->bal = 0;
                vario_h = false;
    }
```

51

```
Insertar(pa->der,e);
if(vario_h)          /* crecio la altura subarbol derecho */
    switch (pa->bal)
    {
          case -1:
                pa->bal = 0;
                vario_h = false;
                break;
          case 0:
                pa->bal = 1;
                break;
          case 1:
                p1 = p->der;
                if (p1->bal == 1)
                {          /* DD */
                      p->der = p1->izq;
                      p1->izq = pa;
                      pa->bal = 0;
                      pa = p1;
                }
                else
                {          /* DI */
                      p2 = p1 ->izq;
                      p1->izq = p2->der;
                      p2->der = p1;
                      pa->der = p2->izq;
                      p2->izq = pa;
                      p->bal = p2->bal ==  1 ? -1 : 0;
                      p1->bal = p2->bal == -1 ?  1 : 0;
                      pa = p2;
                }
                pa->bal = 0;
                vario_h = false;
    }
```

52

```
template<class T>
void
ABBImp<T>::Eliminar(const T& x, Nodo* &abb)
{
    if (!EsVacio(abb))
    {
          if (abb->dato < x)
                Eliminar(x, abb->der);
          else if (abb->dato > x)
                Eliminar(x, abb->izq);
          else
          {
                if (!EsVacio(abb->der) && !EsVacio(abb->izq))
                {
                      Nodo* pred = Maximo(abb->izq);
                      abb->dato = pred->dato;
                      Eliminar(pred->dato, abb->izq);
                }
                else
                {
                      Nodo* temp = abb;
                      if (EsVacio(abb->der))
                            abb = abb->izq;
                      else
                            abb = abb->der;
                      delete temp;
                }
          }
    }
}
```

**Eliminación por "copia" en ABB**

53

## Eliminación en AVL

Deben tenerse en cuenta las mismas consideraciones que en la eliminación de ABB más los rebalanceos.

Las inserciones de AVL y ABB son idénticas, a menos de los rebalanceos.

Con las eliminaciones pasa lo mismo. De allí, el código es el mismo. Veremos en cada caso (luego de la eliminación) qué rebalanceos corresponden.

54

## Eliminación en AVL (2)

Los rebalanceos son similares a la inserción, pero no son iguales.

Cuando se elimina a la derecha de N,

1) Si bal(N) = -1 , al eliminar bal(N) = -2 . Se requiere rebalanceo, que puede ser II o ID. Misma situación que la inserción por la izquierda.

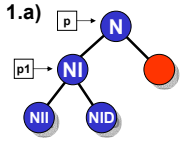
2) bal(N) = 0 , ahora es -1, la altura derecha no varía

3) bal(N) = 1 , ahora debe ser 0, la altura derecha varía



## Eliminación por la derecha de N

1.a)



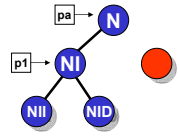

Recordar que la eliminación, en última instancia, se da en una hoja, y por ser AVL, el balance de su padre como mucho puede ser -1

Dependiendo del balance de NI, será un rebalanceo II o ID.
a) Si bal(NI) = 0, la rotación es II pero los balances son diferentes que antes
b) Si bal(NI) = -1, el caso es equivalente a una inserción por la izquierda de NI y la rotación es II
c) Si bal(NI) = 1, el caso es equivalente a una inserción por la derecha de NI y la rotación es ID

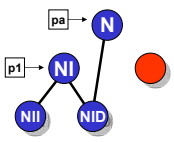


## Eliminación por la derecha de N (2)

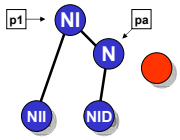
1.a) Luego de eliminar

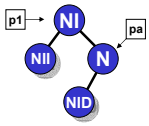



pa->izq = p1->der




p1->der = pa



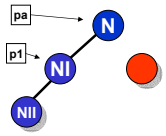




p1->bal = 1
pa->bal = -1

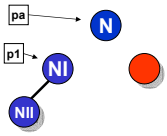


## Eliminación por la derecha de N (3)

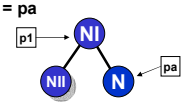
1.b) Luego de eliminar




pa->izq = p1->der




p1->der = pa




pa->bal = 0
p1->bal = 0



```
p1 = pa->izq;
b1 = p1->bal;

if(b1 <= 0){/* casos 1.a y 1.b */
     pa->izq = p1->der;
     p1->der = pa;
     if (b1 == 0){/* 1.a */
            pa->bal =-1;
            p1->bal = 1;
            vario_h = false;
     }
     else /* 1.b */
            pa->bal = p1->bal = 0;
     pa = p1;
}
else {      /* ID */
     p2 = p1->der;
     d2 = p2->bal;
     p1->der = p2->izq;
     p2->izq = p1;
     pa->izq = p2->der;
     p2->der = pa;
     pa->bal = b2 == -1 ?  1 : 0;
     p1->bal = b2 ==  1 ? -1 : 0;
     pa = p2;
     p2->bal = 0;
}
```

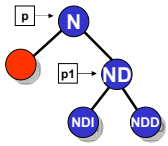


```
template <class T>
void AVLImp<T>::BalanceoD(avl &pa){
  avl p1, p2;
  int b1, b2;

  switch(pa->bal){
     case 1:
          pa->bal = 0;
          break;
     case 0:
          pa->bal = -1;
          vario_h = false;
          break;
     case -1:

  }
}
```

## Eliminación por la izquierda de N

Cuando se elimina a la izquierda de N,
1) Si bal(N) = 1 , al eliminar bal(N) = 2 . Se requiere rebalanceo, que puede ser DD o DI. Misma situación que la inserción por la derecha.
2) bal(N) = 0 , ahora es 1. La altura no cambia.
3) bal(N) = -1 , ahora debe ser 0. La altura cambia.

**1.a)**




Dependiendo del balance de ND, será un rebalanceo DD o DI.
a) Si bal(ND) = 0, la rotación es DD pero los balances son diferentes que antes
b) Si bal(ND) = 1, el caso es equivalente a una inserción por la derecha de ND y la rotación es DD
c) Si bal(ND) = -1, el caso es equivalente a una inserción por la izquierda de ND y la rotación es DI



```
template <class T>
void AVLImp<T>::BalanceoI(avl &pa){
  avl p1, p2;
  int d1, d2;

  switch(pa->bal){
     case -1:
          pa->bal = 0;
          break;
     case 0:
          pa->bal = 1;
          vario_h = false;
          break;
     case 1:

  }
}
```



```
p1 = pa->der;
d1 = p1->bal;

if(d1 >= 0) {       /* DD */
     pa->der = p1->izq;
     p1->izq = pa;
     if(d1==0){
             pa->bal = 1;
             p1->bal = -1;
             vario_h = false;
     }
     else
             pa->bal =p1->bal = 0;
     pa = p1;
}
else{       /* DI */
     p2 = p1->izq;
     d2 = p2->bal;
     p1->izq = p2->der;
     p2->der = p1;
     pa->der = p2->izq;
     p2->izq = pa;
     pa->bal = d2 ==  1 ? -1 : 0;
     p1->bal = d2 == -1 ?  1 : 0;
     pa = p2;
     p2->bal = 0;
}
```



## Árbol de búsqueda general

Una estructura árbol (general) con tipo base T es,

1. O bien la estructura vacía
2. O bien un nodo de tipo T, llamado raíz del árbol, junto con un número finito de estructuras de árbol, de tipo base T, disjuntas, llamadas *subárboles*

Un árbol de búsqueda general es una estructura árbol general que cumple:

1) Cada nodo posee m subárboles y m-1 elementos
2) Los elementos en cada nodo están ordenados (asc)
3) El conjunto de elementos contenidos en los i primeros subárboles son todos menores que el elemento i
4) El conjunto de elementos contenidos en los m-i últimos subárboles son todos mayores que el elemento i



## Árbol B

Un árbol B es un árbol de búsqueda general que cumple:

1) La raíz tiene al menos dos subárboles salvo que sea hoja.
2) Cada nodo posee k-1 elementos (llaves) y k subárboles (hijos), donde techo(m/2) =< k =<m (las hojas tienen 0 hijos).
3) Todas las hojas están en el mismo nivel.



## Árbol B (2)

Es decir, un árbol B siempre está al menos medio lleno, tiene pocos niveles y es perfectamente balanceado.

La ventaja es que, la altura (h) del árbol B está acotada por
h =< log(q, (n+1)/2 + 1)
Siendo q = techo(m/2)
(más eficiente que un AVL)

La desventaja es la relativa complejidad de la inserción y la eliminación.

En este caso, conviene definir el nodo como una clase aparte, ya que conviene distribuir la complejidad de la implementación entre el árbol y el nodo.

## Implementación (nodo)

```
template<class K, int M>
class NodoArbolB
{
public:
   NodoArbolB()
   {
         esHoja = true;
         nroLlaves = 0;
         padre = 0;

         for (int i = 0; i < M; i++)
                  hijos[i] = 0;
   }

   /* Métodos del nodo */

   bool esHoja;
   int nroLlaves;

   K llaves[M-1];

   NodoArbolB* padre;
   NodoArbolB* hijos[M];
};
```



## Métodos del nodo

```
bool EsLleno() { return nroLlaves == M-1; }
bool EsVacio() { return nroLlaves == 0; }
bool EsRaiz() { return padre == 0; }

NodoArbolB* HermanoIzquierdo();
NodoArbolB* HermanoDerecho();

K SacarUltima();
void Insertar(K llave, NodoArbolB*
hijoDerecho);
void Eliminar(K llave);

/* 0 =< retorno =< nroLlaves < M */
int BuscarPosicion(K llave);
```



## Métodos del nodo (2)

```
/* separar this teniendo en cuenta la llave,
retorno:
     para toda tupla de llaves (ll1, ll2) de (retorno X
this),
     ll1 > ll2 */
NodoArbolB* Separar(K llave, NodoArbolB* hijo);

/* combina el nodo parámetro con this,
las llaves de nodo deben ser menores a las de this */
void Copiar(NodoArbolB* nodo);

/* llaves sobrantes := nroLlaves > M/2 */
NodoArbolB*
SeleccionarHermanoInmediatoConLlavesSobrantes();

/* llaves balanceadas := nroLlaves == M/2 */
NodoArbolB*
SeleccionarHermanoInmediatoConLlavesBalanceadas();

/* retorno = posicion llave pivot entre los dos nodos */
int Pivot(NodoArbolB* hermano);
```



## Búsqueda en ArbB

```
template<class K, int M>
NodoArbolB<K,M>*
ArbolBImp<K,M>::BuscarArbolB(K llave, NodoArbolB<K,M>* nodo)
{
   if (nodo != 0)
   {
         int pos = nodo->BuscarPosicion(llave);
         if (pos < nodo->nroLlaves)
         {
                  if (llave < nodo->llaves[pos])
                           return BuscarArbolB(llave, nodo->hijos[pos]);
                  else if (llave > nodo->llaves[pos])
                           return BuscarArbolB(llave, nodo->hijos[pos+1]);
         }
         else
         {
                  return 0;
         }
   }
   return nodo;
}
```



## Búsqueda en ArbB (2)

```
/* 0 =< retorno =< nroLlaves < M */
int BuscarPosicion(K llave)
{
   for (int i = 0; i < nroLlaves && llaves[i] < llave;
         i++);
   return i;
}
```



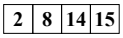
## Inserción en ArbB

La clave está en construir el árbol bottom-up.

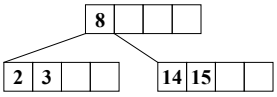
Cuando se quiere insertar una nueva llave, se busca la hoja correspondiente para insertarlo, si hay lugar, se inserta y termina la inserción, si no hay lugar, se crea otra hoja, se distribuyen las llaves y la llave “mediana” se inserta en el padre.

## Ejemplos (Árbol B orden 5)

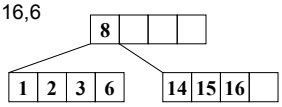
Insertar 2,8,14,15




Insertar 3



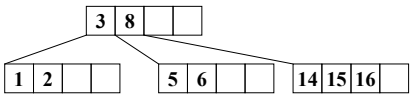

Insertar 1,16,6



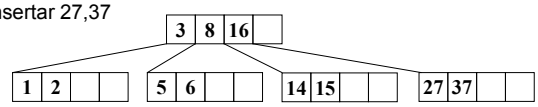



## Ejemplos (Árbol B orden 5) (2)

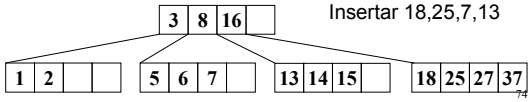
Insertar 5




Insertar 27,37



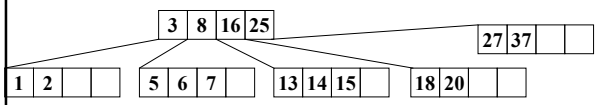

Insertar 18,25,7,13






## Ejemplos (Árbol B orden 5) (3)

Insertar 20




Insertar 22,23,24 ?



## Implementación inserción

```
template<class K, int M>
void
ArbolBImp<K,M>::InsertarArbolB(K llave,
  NodoArbolB<K,M>* arbol)
{
  NodoArbolB<K,M>* nodo =
  BuscarHojaConSucesor(llave, arbol);
  NodoArbolB<K,M>* nodoAReubicar = 0;

  InsertarArbolB(llave, nodo,
  nodoAReubicar);
}
```



```
template<class K, int M>
void
ArbolBImp<K,M>::InsertarArbolB(K llave, NodoArbolB<K,M>* nodo,
    NodoArbolB<K,M>* nodoAReubicar)
{
    if (!nodo->EsLleno())
    {
        nodo->Insertar(llave, nodoAReubicar);
    }
    else
    {
        nodoAReubicar = nodo->Separar(llave, nodoAReubicar);
        llave = nodo->SacarUltima();

        if (nodo->EsRaiz())                // Se debe crear nueva raiz
        {
            raiz = new NodoArbolB<K,M>();
            nodo->padre = raiz;
            raiz->hijos[0] = nodo;
            raiz->Insertar(llave, nodoAReubicar);
            raiz->esHoja = false;
        }
        else
        {
            InsertarArbolB(llave, nodo->padre, nodoAReubicar);
        }
    }
}
```



```
template<class K, int M>
NodoArbolB<K,M>*
ArbolBImp<K,M>::BuscarHojaConSucesor
                        (K llave, NodoArbolB<K,M>* nodo)
{
  assert(nodo != 0);

  NodoArbolB<K,M>* ret = 0;

  int i = nodo->BuscarPosicion(llave);
  if ((i == nodo->nroLlaves || llave < nodo->llaves[i])
      && nodo->hijos[i] != 0)
      ret = BuscarHojaConSucesor(llave, nodo->hijos[i]);
  else if (i < nodo->nroLlaves && nodo->hijos[i+1] != 0)
      ret = BuscarHojaConSucesor(llave, nodo->hijos[i+1]);
  else
      ret = nodo;

  assert(ret != 0 && ret->esHoja);
  return ret;
}
```

```
/* inserta la llave en this */
void Insertar(K llave, NodoArbolB* hijoDerecho)
{
    assert(!EsLleno());

    int pos = BuscarPosicion(llave);

    for (int i = nroLlaves-1; i >= pos; i--)
    {
          llaves[i+1] = llaves[i];
          hijos[i+2] = hijos[i+1];
    }

    llaves[pos] = llave;
    if (hijoDerecho != 0)
    {
          hijos[pos+1] = hijoDerecho;
          hijoDerecho->padre = this;
    }
    nroLlaves++;
}
```

79

## Bibliografía


*Introduction to Algorithms.* Thomas H. Cormen, Ronald L. Rivest, Charles E. Leiserson. The MIT Press. 1990

*Algoritmos y Estructuras de Datos.* Niklaus Wirth. Prentice Hall Hispanoamericana. 1986


80