

## El Tipo de Dato Abstracto Conjunto

Es un TDA contenedor que almacena elementos del mismo tipo. No está permitido que haya repeticiones entre sus elementos. No es necesario que haya orden entre los mismos.

Las operaciones básicas son las más habituales entre conjuntos *matemáticos*: agregar un elemento, sacar un elemento, determinar si un elemento pertenece o no a un conjunto. A veces se agregan otras operaciones propias de conjuntos como unión, intersección, diferencia, etc, pero no es imprescindible que estén puesto que con las básicas puede obtenerse aquellos resultados. Los conjuntos suelen tener también operaciones que permiten hacer recorridos según diversas condiciones.

### Primitivas del TDA Conjunto:

**Crear Conjunto:** crea una instancia de Conjunto.

Precondiciones: ---

Postcondiciones: conjunto en condiciones de ser utilizado

**Destruir Conjunto:** elimina la instancia de Conjunto.

Precondiciones: la instancia de conjunto debe existir

Postcondiciones: ---

**Alta de un elemento:** recibe un elemento, que agrega al conjunto

Precondiciones: la instancia de conjunto debe existir y el elemento no debe estar en ella.

Postcondiciones: el conjunto se modifica, aumentando la cantidad de elementos en 1.

**Baja de un elemento:** recibe un elemento que elimina del conjunto

Precondiciones: la instancia de conjunto debe existir y el elemento debe estar en ella..

Postcondiciones: el conjunto se modifica, disminuyendo la cantidad de elementos en 1.

**Pertenece, o Está:** recibe un elemento y retorna un valor lógico indicando si ese elemento está en el Conjunto.

Precondiciones: la instancia de conjunto debe existir.

Postcondiciones: ---

**Recorrer** (puede haber distintas variantes): visitan todos los elementos del conjunto.

Precondiciones: la instancia de conjunto debe existir.

Postcondiciones: ---

### **Clase Conjunto:**

Si consideramos implementar el Conjunto con una clase, el siguiente podría ser el aspecto de la misma. Los recorridos podrían plantearse como métodos de la clase (como señalamos, puede haber distintos tipos de recorrido) o bien utilizar iteradores para realizarlos.

```
class Conjunto
{
public:
    Conjunto();
    void Alta(Tipodato);
    bool Buscar(Tipodato) const;
    void Borrar(Tipodato);
    void Recorrer() const; //una posibilidad...

private:
    .....
    .....
};
```

El TDA Conjunto es ampliamente utilizado en informática, y tiene muchas implementaciones distintas. Vamos a considerar, primero, cómo podríamos implementarlo con las estructuras que ya conocemos.

- En array (estático o dinámico): en este caso los elementos se almacenarían en un array (generalmente se utiliza uno dinámico) de forma ordenada o no. Se necesitaría considerar el tamaño del array (podríamos llamarlo tam) y la posición del último elemento escrito (que podría llamarse pos\_ult) para poder efectuar las operaciones de alta, baja, búsqueda y recorrido. Las operaciones básicas que mencionamos podrían plantearse del siguiente modo .

Crear Conjunto: genera el array e inicializa tam y pos\_ult.

**Destruir Conjunto:** libera la memoria usada por el array

**Alta:** recibe un elemento, chequea si está y si no está lo agrega. El chequeo tiene coste distinto si el array está ordenado ( $O(\log n)$ ) si está ordenado y  $O(n)$  si no). Si el orden se mantiene siempre, baja el coste de la búsqueda pero sube el de la inserción (si hay corrimiento de los elementos para ubicar al nuevo, en el peor caso el coste es  $O(n)$ ). Si el orden no se mantiene es bajo el coste de la inserción ( $O(1)$ ), pero sube el de la búsqueda. También hay que considerar que si el elemento no está puede no haber espacio suficiente para el nuevo elemento; en este caso podría no llevarse a cabo el alta, o bien generar un nuevo array de mayor tamaño, copiar los datos anteriores, liberar la memoria del array “viejo” y agregar el nuevo elemento. Obviamente, si el alta modifica el valor de `pos_ult`. Podemos decir entonces que, esté o no el array ordenado, el alta es  $O(n)$  para el peor caso.

**Baja:** se recibe el elemento, se chequea si está y si es así se lo elimina del array, actualizando `pos_ult`. Como en el caso del alta, en el cálculo del coste influye si el array está ordenado o no, pero lo que se gana en tiempo de búsqueda si está ordenado se pierde si hay que realizar un corrimiento para mantener el orden. La baja será pues,  $O(n)$  para el peor caso.

**Búsqueda de un dato:** dependerá de si el array está ordenado (coste temporal peor caso  $O(\log n)$ ) o no (coste  $O(n)$ )

**Recorrido:** se recorre el array eventualmente emitiendo sus elementos a un coste  $O(n)$ .

- En estructura de lista ligada (simple, doble, circular, etc.): las posibilidades de estructura de lista son diversas, pero los análisis son análogos para todas ellas. Los elementos del conjunto quedarían almacenados en los nodos de una lista ligada de algún tipo. Las operaciones básicas pueden pensarse así:

**Crear el Conjunto:** se pone el puntero de entrada en nulo. Coste  $O(1)$

**Destruir el conjunto:** se recorre la lista, liberando cada nodo. Esto tiene un costo  $O(n)$ .

**Alta de un elemento:** se recibe un dato a agregar y, para chequear que el elemento no está, puede tener que recorrerse la lista completa, con un coste  $O(n)$ , luego de lo cual se generará un nodo con el nuevo elemento. Entonces, para el peor caso, el coste es  $O(n)$ .

**Baja de un elemento:** se recibe un dato que debe eliminarse del conjunto. Al igual que en el alta, se debe revisar la lista en busca del dato que hay que eliminar antes de proceder a la eliminación en sí, por lo cual el coste para el peor caso es  $O(n)$ .

**Búsqueda de un dato:** se recorrerá la lista rastreando un elemento, a un coste de  $O(n)$  para el peor caso.

**Recorrido:** como se trata de recorrer una estructura de lista ligada, su coste es  $O(n)$ .

Hay muchas otras implementaciones posibles; algunas las desarrollaremos en este curso, con mayor o menor profundidad según los objetivos planteados. Entre ellas las siguientes:

- En árbol binario de búsqueda (ABB): esta estructura de datos, que se tratará más adelante, permite, bajo ciertas condiciones, bajar el coste de las operaciones básicas en Conjuntos.
- En ABB balanceado (por su peso o por su altura): de las diversas variantes de ABB balanceados desarrollaremos los árboles AVL.
- En árboles multivías (en particular, en árbol B): otra variante de árboles, muy usada para implementar diccionarios.
- En array de bits: una alternativa de muy bajo coste, muy útil en ciertas condiciones.
- En árboles digitales (“trie”): otra variante de estructura arborescente también muy usada para implementar diccionarios.
- En tablas de dispersión (tablas “hash”): el hashing puede usarse para manipular tablas donde se implementen conjuntos o diccionarios. Las tablas hash constituyen un tema muy extenso en informática.
- Y hay más...

### **El Tipo de Dato Abstracto Diccionario:**

Es un TDA relacionado con el TDA Conjunto. Básicamente, es un Conjunto de pares (clave, valor), en el que las claves no pueden tener repeticiones.

En este TDA, a las operaciones que se ha mencionado de los conjuntos se agrega una primitiva Valor, que recibe una clave y retorna el valor asociado a la misma.

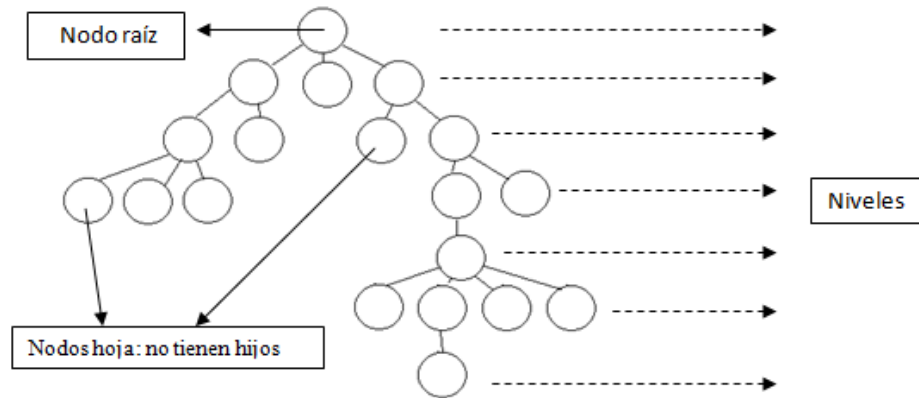
Las estructuras utilizadas para implementar el TDA Diccionario son las mismas que se utilizan para implementar el TDA Conjunto.

Los Diccionarios también tienen una extensa aplicación en informática, en particular como Índices de archivos.

### **Estructura arborescente (árbol):**

Los árboles son estructuras de datos que constan de un conjunto de nodos organizados jerárquicamente. Cada nodo tiene un padre y sólo uno, excepto un nodo distinguido llamado raíz.

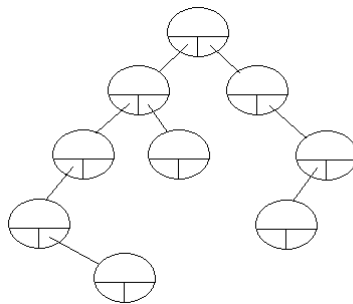
Cada nodo puede tener cero o más hijos. Según las características del árbol, la cantidad de hijos puede estar limitada o no.



Nota: es fundamental diferenciar el TDA Árbol de la estructura de datos árbol. En el TDA Árbol las operaciones propias nos remiten a la manipulación de nodos, árboles, relación con el padre, etc. ( ejemplo: tendremos primitivas para determinar quién es el padre de un nodo, insertar un elemento como hijo de otro especificado, eliminar un subárbol, etc.) . El TDA árbol obviamente puede implementarse a través de una estructura que no sea arborescente. Como en todo TAD, lo que hace que sea árbol y no otro tipo, es aquello que ofrece a través de su interfaz, considerando estas primitivas junto con sus precondiciones y postcondiciones.

En el caso del TDA Conjunto, queremos implementarlo usando diversas estructuras arborescentes para poder comparar los pro y contra de cada una de ellas, pero en todos los casos, la interfaz tendrá las mismas operaciones crear, destruir, alta, baja, está (o pertenece) y recorrer.

### Árbol Binario:



Es una estructura de datos compuesta por nodos que verifica que cada nodo, excepto la raíz tiene uno y solo un padre. Cada nodo tiene 0, 1 ó 2 hijos.

También puede definirse así: un árbol binario es nulo, o bien consta de un nodo raíz y dos subárboles hijos, que son árboles binarios.

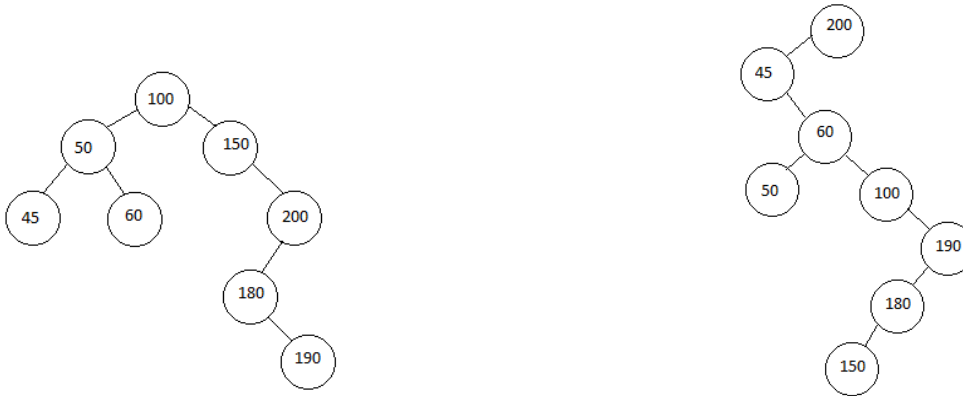
Ejercicio propuesto:

¿Cuántos árboles binarios distintos se pueden formar con  $n$  nodos? (no importa el orden).

### Árbol binario de búsqueda:

Es un árbol binario que verifica que para todo nodo con clave  $x$ , toda clave del subárbol derecho de ese nodo es mayor que  $x$  y toda clave del subárbol izquierdo de ese nodo es menor que  $x$ .

Ejemplos:



Como se observa en el ejemplo, con los mismos datos las configuraciones pueden ser distintas. El primer ejemplo tiene un balanceo mayor, el árbol tiene un aspecto más triangular que el caso de la derecha.

Las estructuras ABB tienen asociados algoritmos para realizar operaciones básicas, a través de las cuales implementaremos las primitivas del TDA conjunto.

Estas operaciones son: la búsqueda de una clave, el alta de una clave, el borrado de una clave, y distintas variantes de recorridos. A través de ellas implementaremos respectivamente la búsqueda, el alta, la baja y el recorrido en un conjunto.

#### Búsqueda de una clave en un ABB:

El algoritmo sigue el siguiente esquema

Si (p es nulo)            // p es puntero a nodo, inicialmente apuntando a la raíz.

    {retornar false;} // la clave k no está en el ABB

Si no

    si ( $k ==$  clave del nodo apuntado por p)

        {retornar true;} // clave encontrada

    si no

        Si ( $k >$  clave del nodo apuntado por p)

{Continuar búsqueda en subarbol derecho}

si no

{Continuar búsqueda en subárbol izquierdo}

Lo que se menciona en el esquema anterior como “continuar la búsqueda” implica un descenso en el árbol hasta localizar la clave o determinar que no está en la estructura.

Observar que de cada nivel del ABB se analiza un solo nodo. Esto implica que haremos, en el peor caso tantas comparaciones como cantidad de niveles tenga el ABB (si el árbol binario tiene  $n$  nodos y está completo en todos sus niveles, la altura es aproximadamente  $\log n$ ). Por eso es más conveniente que la estructura tenga aspecto triangular y no similar a una lista ligada.

Algoritmo iterativo de búsqueda en ABB (pseudocódigo):

// p es el puntero de entrada al árbol binario de búsqueda; aux es puntero a nodo de ABB

// x es la clave buscada

esta= falso;

Si (p no es nulo)

{

aux=p;

mientras ((aux no es nulo) y (esta = falso))

{

si (x==clave de nodo apuntado por aux)

{esta=verdadero;}

si no

si (x> clave de nodo apuntado por aux)

{aux= puntero\_derecho\_de\_aux ;}

si no

{aux= puntero\_izquierdo\_de\_aux ;}

```
}
```

```
}
```

retornar esta;

Análisis del coste temporal: La primera parte del algoritmo, hasta que comience el ciclo es  $O(1)$ . Consideremos el ciclo: en el peor caso se recorrerán todos los niveles del árbol. El número de niveles del árbol depende de la configuración del mismo. Las situaciones “extremas” son: la mejor, si el aspecto es triangular, y la peor si hay sólo un nodo por nivel. En este último caso decimos que el ABB ha “degenerado en lista” y habrá  $n$  niveles, lo que hace que el ciclo sea  $O(n)$ . El bloque de sentencias controlado por el ciclo “mientras” tiene un coste  $O(1)$ . Entonces, por regla del producto, para el peor caso la búsqueda iterativa planteada pertenece a  $O(n)$ .

Algoritmo recursivo de búsqueda en ABB (pseudocódigo):

bool Esta (p: puntero de entrada al árbol binario de búsqueda; x: clave buscada)

```
{
```

Si (p es nulo)

```
{retornar falso}
```

Si no

```
{
```

si ( $x ==$  clave del nodo apuntado por p)

```
{retornar verdadero}
```

si no

si ( $x >$  clave del nodo apuntado por p)

```
{retornar Esta (puntero_derecho_de_p, x);}
```

si no

```
{retornar Esta (puntero_izquierdo_de_p, x);}
```

```
}
```

```
}
```



Análisis del coste temporal: analizando lo que presenta el algoritmo nos encontramos con que, si el puntero no es nulo, se analiza la situación del nodo apuntado (coste  $O(1)$ ); si ahí no está el dato buscado se invoca con el puntero izquierdo o derecho de p. Solo se realiza una de esas invocaciones para cada nivel del ABB, y como el número de niveles del ABB descontando el nivel del nodo analizado es  $n-1$ , para el peor caso resulta esta ecuación de recurrencia:

$$T(n) = T(n-1) + 1, \quad T(0) = 1$$

Lo cual indica que  $T(n)$  pertenece a  $O(n)$

Observar que, además, el algoritmo recursivo tiene recursividad “de cola”, lo cual lo vuelve más eficiente en cuanto al coste espacial.

#### Alta en un ABB:

Los ABB crecen “de arriba hacia abajo”, es decir que se genera un nodo cuando se encuentra nulo el puntero apropiado. La búsqueda de este puntero se hace rastreando desde la raíz y hacia las hojas del ABB, como se muestra en los siguientes esquemas:

// p es puntero a la raíz del ABB

// x es la clave a almacenar

Si (p es nulo)

```
{generar nuevo nodo;  
  almacenar x en nodo;  
  poner punteros izquierdo y derecho en nulo;  
}
```

Si no

si (la clave es igual a x)

{detener el proceso; la clave ya está en el ABB}

si no

si ( $x >$  clave del nodo apuntado)

{continuar proceso en el subárbol derecho del nodo considerado;}

si no

{continuar proceso en el subárbol izquierdo del nodo considerado;}

La “continuación de la búsqueda” puede plantearse de forma iterativa o recursiva, lo cual nos lleva a los siguientes algoritmos:

Algoritmo iterativo de alta en ABB:

```
AltaABB (x: clave a insertar)    //p es el puntero al nodo raíz
{
si (p es nulo)
{
    generar nodo nuevo almacenando dirección en p;
    almacenar x en nodo nuevo;
    poner a nulos punteros derecho e izquierdo de nodo nuevo;
}
si no
{
    //aux: puntero a nodo, usado como auxiliar en el descenso
    aux = p;

    //listo: variable booleana que indica si terminó el proceso de alta
    listo = falso;
    mientras (listo == falso)
    {
        Si (x < clave almacenada en nodo apuntado por aux)
        {
            Si (el puntero izquierdo de aux no es nulo)    //si hay árbol izquierdo no nulo
                {aux = puntero izquierdo de aux;}    //descender por izquierda
        }
        Si no
        {
```

```
generar nodo nuevo desde puntero izquierdo de aux;  
almacenar x en nodo nuevo;  
poner en nulo punteros izquierdo y derecho de nodo nuevo;  
listo = verdadero;  
}  
}
```

si no

Si (  $x >$  clave almacenada en nodo apuntado por aux)

```
{  
  Si (el puntero derecho de aux no es nulo)  //si hay árbol derecho no nulo  
    {aux = puntero derecho de aux;}  //descender por derecha  
  Si no  
  {  
    generar nodo nuevo desde puntero derecho de aux;  
    almacenar x en nodo nuevo;  
    poner en nulo punteros izquierdo y derecho de nodo nuevo;  
    listo = verdadero;  
  }  
}
```

si no // x coincide con clave almacenada en nodo apuntado desde aux

```
{listo = verdadero;}
```

```
}
```

Análisis del coste: el algoritmo de alta busca determinar el puntero nulo adecuado para generar desde él el nodo nuevo. Esto implica analizar si el puntero de entrada es nulo para generar el nuevo desde ahí, o bien descender hasta un nodo que tenga nulo el puntero que se requiere, lo cual puede llevarnos, en el peor caso, al nodo más

alejado de la raíz. El ciclo que permite el “descenso” puede tener, por tanto  $n-1$  iteraciones, efectuando en cada una un bloque  $O(1)$ . El coste será, entonces, para el peor caso,  $O(n)$ .

#### Algoritmo recursivo de alta en ABB:

Alta Recursiva (var p: puntero a nodo, x: clave)

```
{
  Si (p es nulo)
  {
    generar nodo nuevo almacenando dirección en p;
    almacenar x en nodo nuevo;
    poner a nulos punteros derecho e izquierdo de nodo nuevo;
  }
  Si no
  {
    Si (x > clave almacenada en nodo apuntado por p)
    {AltaRecursiva (puntero_ derecho_ d e_p, x);}
    Si no
    {
      Si (x < clave almacenada en nodo apuntado por p)
      {AltaRecursiva (puntero_ izquierdo_ d e_p, x);}
    }
  }
}
```

Análisis del coste: dado que para poder generar el nodo nuevo necesitamos rastrear un puntero nulo en el lugar adecuado y esto nos puede llevar a la hoja más alejada de la raíz, el análisis es análogo al de la búsqueda recursiva.

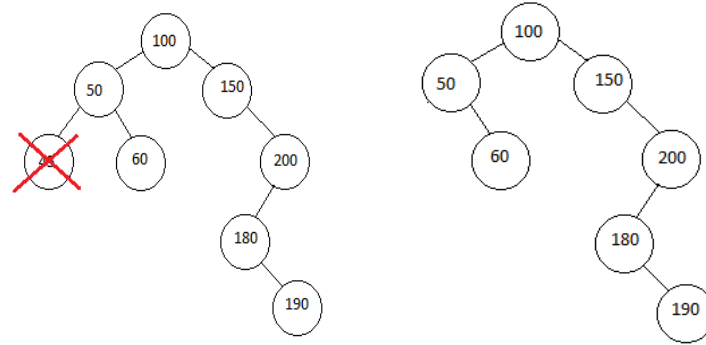
#### Borrado de un dato en un ABB:

Existen varios algoritmos de borrado en ABB. En el que desarrollaremos, se consideran diferentes casos.

**Caso 1:** el nodo que contiene el valor a borrar está en una hoja:

En este caso se elimina el nodo y, si el nodo eliminado tenía padre, el puntero que desde el padre le apuntaba se pone nulo. Si no tenía padre (era hoja y raíz) se pone nulo el puntero de entrada.

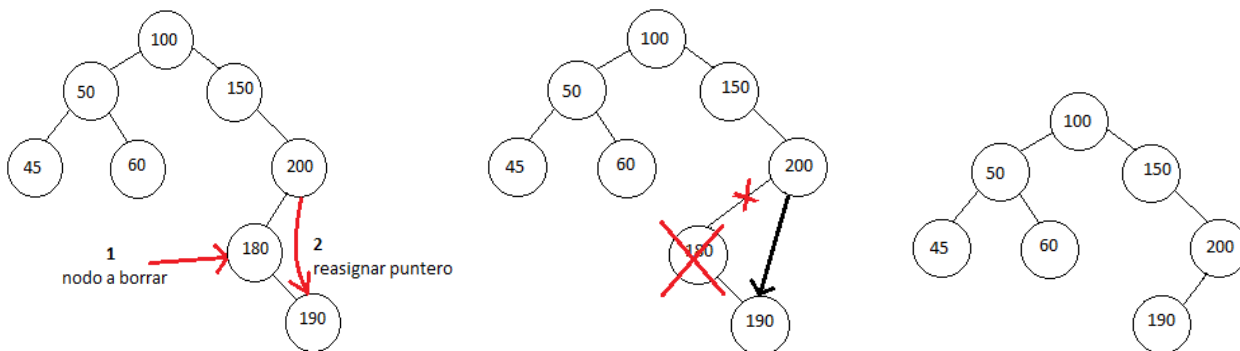
En el siguiente gráfico se muestra el borrado de la clave 40. A la derecha, el estado final del ABB.



**Caso 2:** el nodo que contiene x tiene 1 hijo:

Si el nodo a eliminar tiene padre, se apunta con un auxiliar al que debe eliminarse y el puntero que desde el padre le apuntaba pasará a apuntar al nodo hijo del que se eliminará. Si no tiene padre, el puntero de entrada apuntará al hijo del que se elimina.

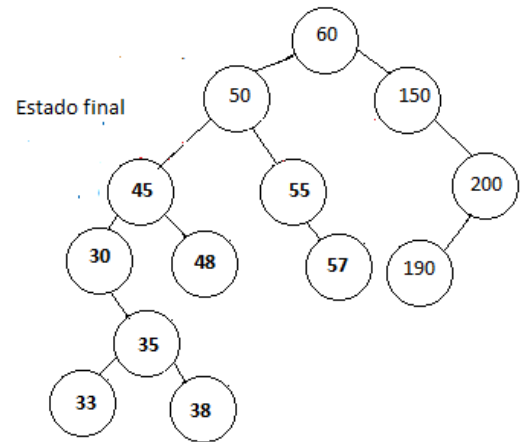
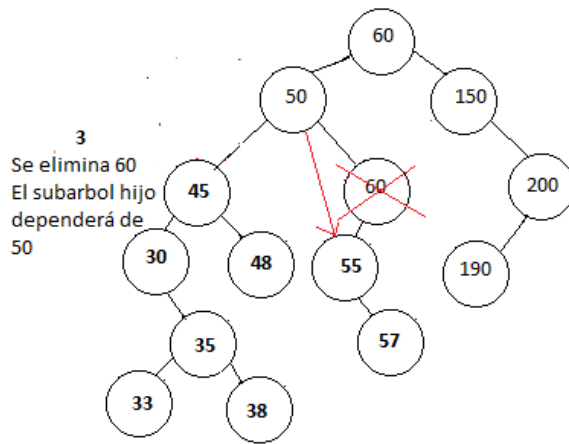
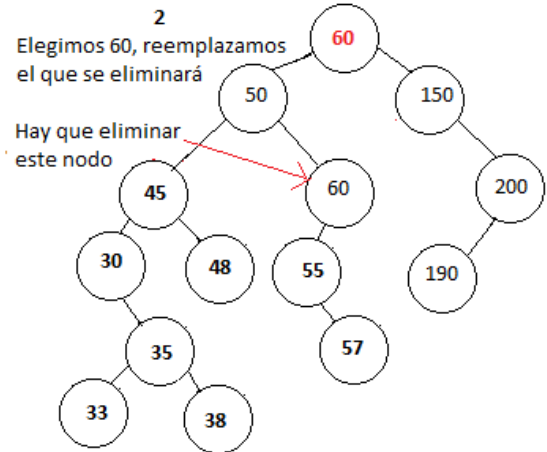
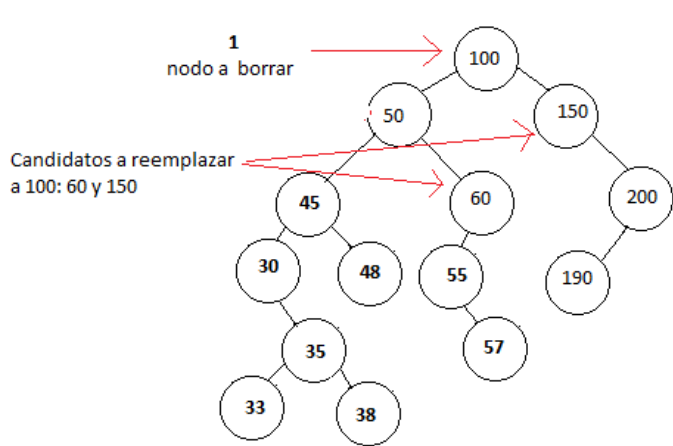
Ejemplo: se elimina la clave 180.



**Caso 3:** el nodo que contiene la clave a borrar tiene dos hijos:

(Llamaremos x a la clave a borrar). En este caso se reemplaza el valor de x por el menor de los mayores a x o por el mayor de los menores que x. (Para encontrar el mayor de los menores: bajar al subárbol izquierdo y moverse por los punteros derechos todo lo posible. Se llega a un nodo que, o no tiene hijos, o sólo tiene uno a izquierda. Para encontrar el menor de los mayores: bajar al subárbol derecho y moverse por los punteros izquierdos todo lo posible. Se llega a un nodo que, o no tiene hijos, o sólo tiene uno a derecha). Luego se elimina el valor duplicado; este nodo tiene 0 o 1 hijo, por lo cual se aplica el Caso 1 o el Caso 2 para eliminarlo.

Ejemplo: eliminación de la clave 100



### Algoritmo de baja en ABB:

//x: clave a eliminar

//pa: puntero que apunta al padre del nodo que contiene x

//p: puntero de entrada

si (p apunta a nodo con x)

{si (nodo con x no tiene hijos)

{ElimCaso1 (p);}

Si no

```
Si (nodo con x tiene un solo hijo)
    {ElimCaso2(p);}
Si no
    {ElimCaso3 (p);}
}
si no
{
    pa= EncontrarPadre (); // esta función retorna el puntero al padre del nodo que contiene x
    si (pa no es nulo) //si pa es nulo, x no está en el ABB
    {
        Si (nodo con x no tiene hijos)
            {ElimCaso1(pa);}
        Si no
            Si (nodo con x tiene un solo hijo)
                {ElimCaso2(pa);}
            Si no
                {ElimCaso3(pa);}
    }
}
```

ElimCaso1 (var pa: puntero a nodo)

```
{
    //aux: puntero a nodo
    Asignar a aux la dirección del nodo que contiene x;
    Poner a nulo el puntero del nodo apuntado por pa que apuntaba a nodo que contiene x;
    Eliminar nodo que contiene x utilizando aux;
}
```

ElimCaso2 (var pa: puntero a nodo)

```
{  
    Asignar a aux la dirección del nodo que contiene x;  
  
    Asignar al puntero del nodo apuntado por pa que apuntaba a nodo que contiene x la dirección del hijo no nulo  
    del nodo que contiene x;  
  
    Eliminar nodo que contiene x utilizando aux;  
}  
  
ElimCaso3 (var pa: puntero a nodo)  
{  
    //aux2: puntero a nodo  
  
    Asignar a aux la dirección del nodo que contiene x;  
  
    aux2= pPadreMayorMenores(aux); // pPadreMayorMenores retorna el puntero al padre del nodo que contiene el  
    mayor de los menores a x  
  
    Asignar al atributo clave del nodo que contiene x el mayor de los menores, apuntado por un hijo de aux2  
  
    Si (nodo apuntado por aux2 tiene un hijo)  
        {elimCaso1(aux2);}   
  
    Si no  
        {ElimCaso2(aux2);}   
  
    Eliminar nodo que contiene x utilizando aux;  
}
```

Análisis del coste: si el nodo no tiene hijos, la búsqueda del padre del nodo a borrar se extenderá, a lo sumo hasta el padre de alguna hoja, con un coste  $O(n)$  si el ABB ha degenerado en lista. Si tiene un solo hijo, nos detendremos en el peor caso, un nivel antes del último. Si el nodo a borrar tiene dos hijos, el rastreo de la ubicación del mayor de los menores nos obliga a continuar el descenso (estamos, en este caso, situados en el padre del que hay que eliminar)), el cual, a lo sumo nos llevará desde el nodo a borrar hasta alguna de las hojas. Como se aprecia en el pseudocódigo, en cada nivel del árbol las sentencias a ejecutar tienen un coste temporal constante. En consecuencia, en cualquiera de los casos, el coste temporal depende del número de niveles del ABB, y en el peor de los casos será  $O(n)$ .

Recorridos en un ABB:

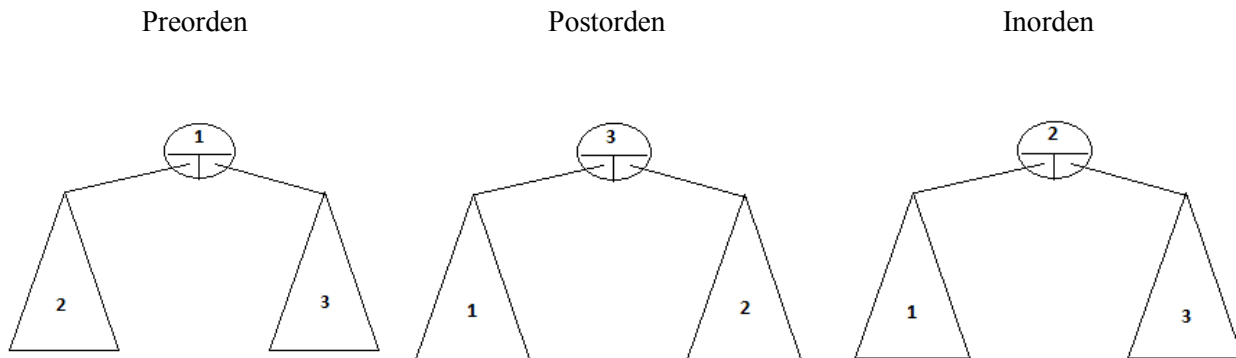


Los recorridos se clasifican en dos categorías: en profundidad y en anchura. En los recorridos en profundidad se procesa cada nodo, su subárbol izquierdo y el derecho. El orden en que se realicen esos pasos diferencia distintas posibilidades del recorrido en profundidad: si el nodo se procesa antes que el subárbol se trata de preorden; si el nodo se procesa después que los subárboles, es post orden, y si se procesa primero el subárbol izquierdo, luego el nodo y luego el subárbol derecho, es inorden. Este último recorrido permite procesar las claves en orden creciente. Estos recorridos pueden implementarse recursivamente o iterativamente con ayuda de una pila.

El recorrido en anchura procesa nivel a nivel los nodos del ABB y generalmente se implementa con una pila.

Cualquiera de estos recorridos permite implementar el recorrido de un conjunto.

### Recorridos en Profundidad:



### Algoritmos recursivos de recorridos en profundidad en ABB:

Preorden:

//p: puntero a nodo,

// inicialmente el de entrada al árbol

Preorden (p)

```
{
    si (p no es nulo)
    {
        procesar clave nodo apuntado por p;
```

```
Preorden (puntero izquierdo de p);  
Preorden (puntero derecho de p);  
}  
}
```

Postorden:

```
//p: puntero a nodo,  
// inicialmente el de entrada al árbol  
Postorden (p)  
{  
    si (p no es nulo)  
    {  
        Postorden (puntero izquierdo de p);  
        Postorden (puntero derecho de p);  
        procesar clave nodo apuntado por p;  
    }  
}
```

Inorden:

```
//p: puntero a nodo,  
// inicialmente el de entrada al árbol  
Inorden (p)  
{  
    si (p no es nulo)  
    {  
        Inorden (puntero izquierdo de p);
```

```
    procesar clave nodo apuntado por p;  
    Inorden (puntero derecho de p);  
}  
}
```

Análisis del coste: considerando las situaciones extremas de un ABB, a saber , que esté equilibrado o que haya degenerado en lista, se tiene:

Si el árbol está equilibrado, al plasmar en una ecuación de recurrencia cualquiera de los algoritmos recursivos de recorrido en profundidad, es:

$$T(n) = 2 T(n/2) + 1 \quad \text{y} \quad T(1) = 1$$

Lo que nos da  $T(n) \in O(n)$

Si el árbol ha degenerado en lista, en cada nivel recursivo una de las dos invocaciones corresponde a  $O(0)$ , ya que el subárbol izquierdo o el derecho estará nulo, y es:

$$T(n) = T(n-1) + 1 \quad \text{y} \quad T(0) = 1$$

Lo que nos da  $T(n) \in O(n)$

Entonces, vemos que independientemente de la configuración del ABB, el recorrido tiene un coste temporal lineal, lo cual es evidente, dado que el coste de visitar un nodo es constante, y debe visitarse cada uno de los  $n$  nodos del ABB.

#### Algoritmos iterativos de recorridos en profundidad en ABB:

Preorden:

// pi: pila

//p: puntero a la raíz del árbol

// x: variable puntero a nodo

Hacer

```
{  
si (p no es nulo)  
{
```

```
pi.apilar (p);  
mientras (pi no vacia)  
{  
  x = pi.desapilar();  
  procesar clave nodo apuntado por x  
  si (puntero derecho de x no es nulo)  
    { pi. Apilar ( puntero derecho de p);}  
  si (puntero izquierdo de x no es nulo)  
    { pi.apilar ( puntero izquierdo de p)}  
}  
}  
}
```

Inorden

//pi: una pila

//p: puntero a la raíz del árbol

// x: : variable puntero a nodo

Hacer

```
{  
  x=p  
  mientras (x no nulo)  
  {  
    pi.apilar (x);  
    x=puntero izquierdo de x;  
  }  
  X= pi.desapilar();
```

Procesar clave nodo apuntado por x;

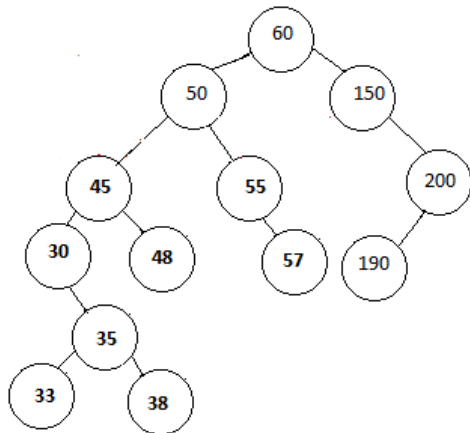
x=puntero derecho de x;

}

mientras (pila no vacia)

Análisis del coste: se observa que para cada uno de los nodos del ABB, se apilará y desapilará posteriormente el puntero al mismo, para procesarlo. El proceso termina cuando la pila se vacía. El coste temporal de apilar y desapilar, considerando una implementación de la pila de las estudiadas antes (por ejemplo, en array o en lista ligada) es  $O(1)$ . Por lo cual, el coste temporal del recorrido iterativo es  $O(n)$ .

Ejemplo: Para el siguiente ABB, se indican los nodos según los tres recorridos en profundidad:



Recorrido Preorden:

60, 50, 45, 30, 35, 33, 38, 48, 55, 57, 150, 200, 190

Recorrido Postorden:

33, 38, 35, 30, 48, 45, 57, 55, 50, 190, 200, 150, 60

Recorrido en anchura en un ABB: Este recorrido visita los nodos nivel a nivel, de izquierda a derecha. Habitualmente se lo plantea iterativo y se implementa con una cola.

Algoritmo iterativo de recorrido en anchura:

//co: cola

// p: puntero a nodo, de entrada al ABB

// x: puntero a nodo

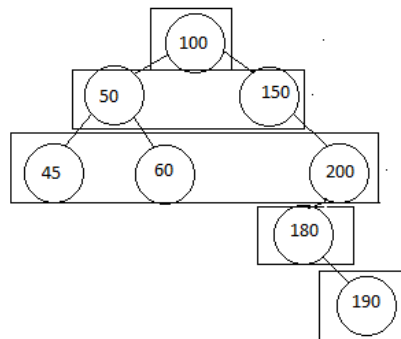
Co.acolar(p)

Mientras (co no vacía)

```
{  
  x= co. desacolar();  
  procesar clave de nodo apuntado por p;  
  si (puntero izquierdo de x no es nulo)  
    {co.acolar(puntero izquierdo de x);}  
  si (puntero derecho de x no es nulo)  
    {co.acolar(puntero derecho de x);}  
}
```

Análisis del coste: se observa que, para los recorridos desarrollados, para cada uno de los nodos del ABB, se acolará y desacolará el puntero que le apunta. El proceso termina cuando la cola queda vacía. Dado que el coste de acolar y desacolar puede ser  $O(1)$ , lo que sucede por ejemplo para las implementaciones de cola en array circular, o bien en lista ligada simple con punteros al principio y al final de la misma, el coste temporal del recorrido en anchura pertenece a  $O(n)$ .

Ejemplo:



El recorrido en anchura de este árbol visita los nodos en este orden: 100, 50, 150, 45, 60, 200, 180, 190

Altura de un ABB:

Para este concepto se pueden encontrar definiciones ligeramente diferentes según el autor que se considere.

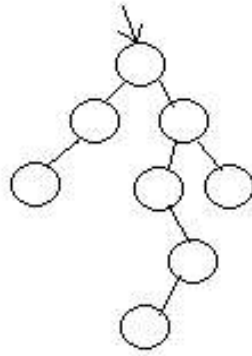
“Si el ABB es nulo su altura es -1. De otro modo, es la longitud del camino más largo que una la raíz con una hoja.”

Altura 0                      Altura 1                      Altura 5

### ABB balanceado por su altura

$$| \text{Altura (subárbol derecho de x)} - \text{Altura (subárbol izquierdo de x)} | \leq d$$

pág. 23



Tomando  $d = 3$  está balanceado

Tomando  $d = 2$  está balanceado

Tomando  $d=1$  no está balanceado

ABB balanceado por el peso

También para este concepto encontramos distintas definiciones:

Una de ellas enuncia: Un ABB está  $\alpha$ -balanceado sii se verifica para todo nodo  $x$ :

(Peso (subárbol izquierdo de  $x$ )  $\leq \alpha \cdot$  Peso(árbol con raíz en  $x$ )) y (Peso (subárbol derecho de  $x$ )  $\leq \alpha \cdot$  Peso(árbol con raíz en  $x$ ))

Siendo el peso de un árbol con raíz en  $r$  el número de nodos de dicho árbol y  $\alpha \in [1/2, 1)$

Otra definición que suele usarse para árbol balanceado por el peso enuncia: Un árbol binario está balanceado por su peso si para todo nodo, la razón entre su propio peso y el peso de su hijo derecho (o izquierdo) está en el intervalo  $[\alpha, 1 - \alpha]$  para un valor arbitrario  $\alpha > 0$ . El peso de un nodo es el número de subárboles que posee.

En este curso no estudiaremos los árboles binarios balanceados por el peso.

Ejercicios propuestos:

1. Escribir algoritmos para resolver los siguientes problemas, y luego codificarlos y evaluar su coste temporal para el peor caso.
  - a. Dado un ABB, escribir un algoritmo para determinar el valor de la mayor y menor clave.
  - b. Dado un ABB y una clave, escribir un algoritmo para retornar el nivel en que se encuentra la misma, o -1 si no está en el ABB
  - c. Dado un ABB, escribir un algoritmo para establecer cuál es el nivel en el que hay más nodos, o el primero de ellos si el máximo se alcanza en más de uno.
  - d. Dado un ABB, escribir un algoritmo para determinar la altura del mismo
  - e. Dado un ABB, escribir un algoritmo para establecer si está balanceado por su altura con diferencia 1, o no
  - f. Dados dos ABB, escribir un algoritmo para determinar si son simétricos (en forma) o no



2. Para un ABB completo con  $m$  niveles ¿cuál es la cantidad de nodos del mismo? ¿Cuántos son hojas?
3. Si un ABB es casi completo y tiene  $n$  nodos ¿cuántos nodos hoja tiene?
4. Implementar las operaciones unión e intersección de conjuntos mediante implementaciones con ABB
5. Considere que un árbol general de orden  $m$  ( $m$  es el máximo número de hijos por nodo) y altura  $h$  se representa mediante un árbol binario, en donde el enlace izquierdo sea al primer hijo y el enlace derecho sea al hermano siguiente. Escriba un algoritmo para determinar si un nodo es hijo de otro según la configuración inicial.

### Árboles AVL

Los arboles AVL (diseñados por Adelson-Velskii y Landis en 1962) son ABB balanceados por su altura con diferencia permitida 1 que usan ciertas rotaciones para rebalancear el ABB cuando es necesario. (ver traducción al inglés del artículo en <http://professor.ufabc.edu.br/~jesus.mena/courses/mc3305-2q-2015/AED2-10-avl-paper.pdf>)

Para ello se agrega un atributo en los nodos del ABB, el factor de balanceo, o FB, el cual almacena la diferencia entre la altura del subárbol derecho y la altura del subárbol izquierdo de ese nodo. Los valores que indican que en

ese nodo no hay problemas son 1, 0 y -1. Si algún nodo del árbol binario tiene en algún momento un valor de FB mayor que 1 o menor que -1, el árbol se ha desbalanceado y deben ejecutarse rutinas de rebalanceo.

El alta, o la baja siguen los siguientes pasos:

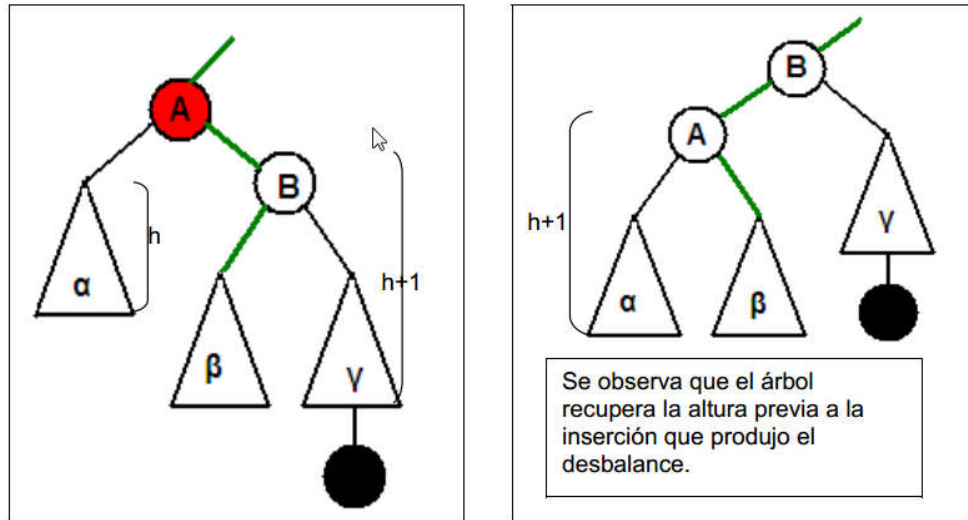
- 1) Ejecutar algoritmo habitual de alta o baja en un ABB.
- 2) Recalcular los valores de los FB desde el punto de la inserción o borrado y hacia la raíz; a lo sumo se analizarán y eventualmente modificarán tantos nodos como número de niveles tiene el árbol luego de la operación realizada.
- 3) Si hay desbalanceo, establecer cuál es el nodo pivote, es decir, el nodo más próximo al punto de inserción o de borrado de la clave que provocó el desbalanceo, ya que las rutinas de rebalanceo indican las reasignaciones a realizar en función de la variación de FB del pivote y alguno de sus hijos. Realizar entonces las rotaciones adecuadas. Las rotaciones son reasignaciones en función de la configuración del árbol desbalanceado.

### Rotaciones:

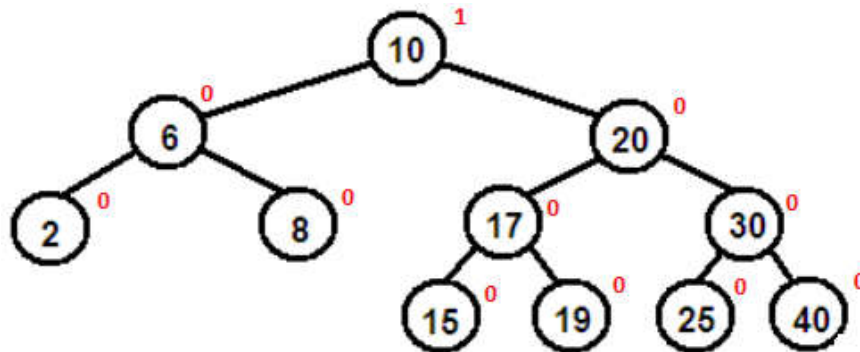
Rotaciones simples (involucran 3 punteros): hay dos casos, antihoraria y horaria imágenes especulares entre sí.

Rotaciones dobles (involucran 5 punteros): hay dos casos, antihoraria-horaria y horaria-antihoraria, cada uno de ellos se subdivide en forma trivial y forma no trivial. También entre si los triviales y los no triviales son imágenes especulares.

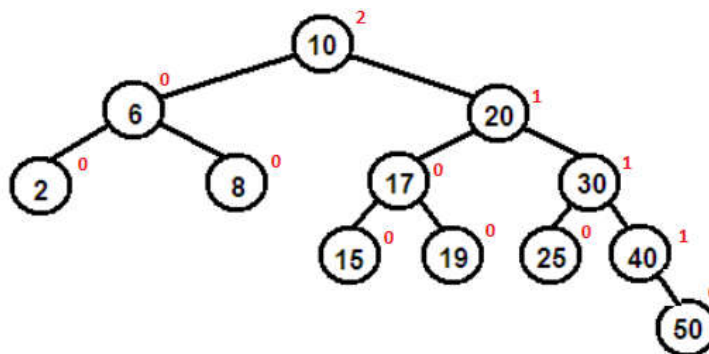
Rotación simple antihoraria:

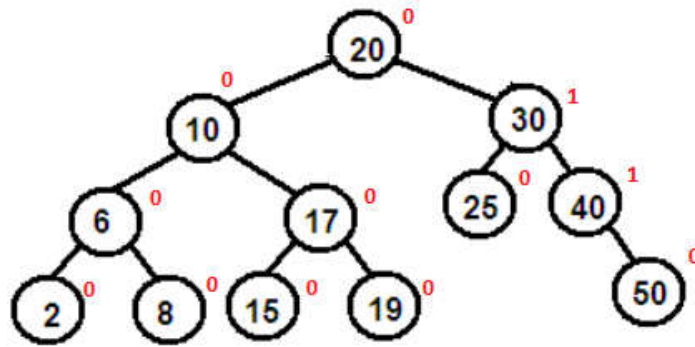


Ejemplo: Se tiene el siguiente ABB, del cual se indica junto a cada nodo el valor de su FB:

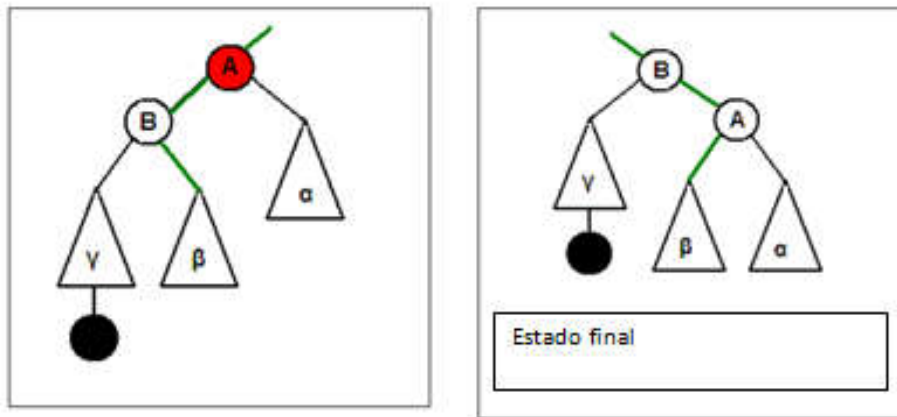


Se inserta 50, con las modificaciones indicadas para los FB:



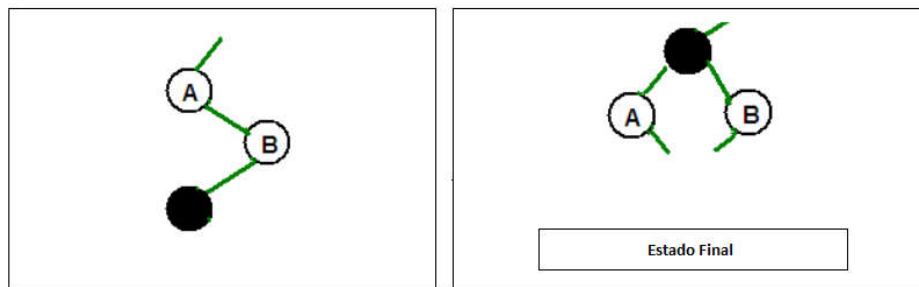


Rotación simple horaria:

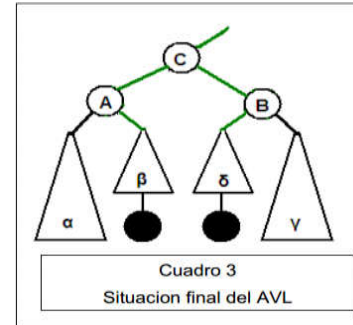
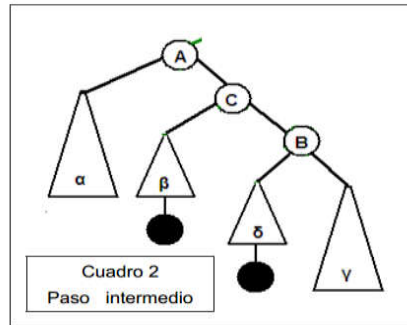
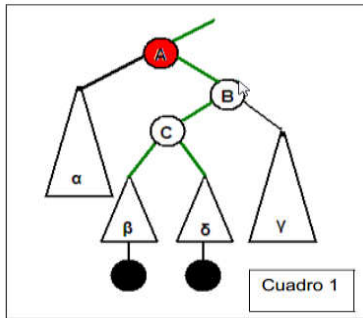


Rotaciones dobles:

Caso 1 horaria-antihoraria trivial:



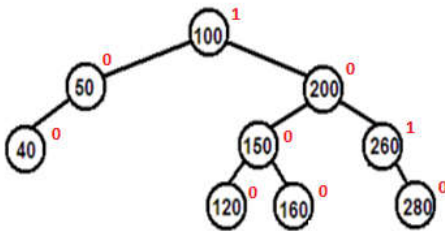
Caso 1 horaria-antihoraria no trivial:



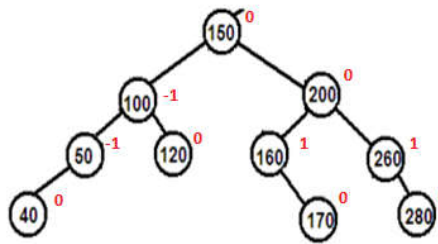
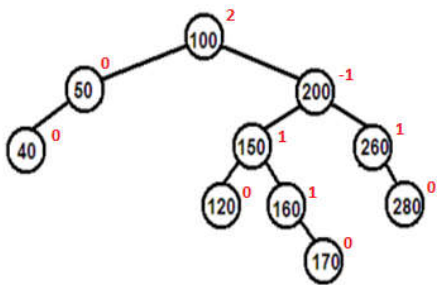
Nota1: en el gráfico anterior, el nodo nuevo, se inserta en una de las posiciones indicadas en negro (no en ambas, obviamente) . Es decir que cualquiera de los nodos negros puede ser el nuevo.

Nota2: se deja como ejercicio plantear el otro caso de rotación (antihoraria-horaria) doble en sus formas trivial y no trivial y analizar qué sucede con el balance del pivote y del resto de los nodos involucrados.

Ejemplo:



En este árbol se da de alta 170, lo cual provoca las modificaciones en los FB indicadas en el cuadro de abajo a la izquierda, y tras la rotación, la resolución de la derecha.



Nuevamente el AVL recupera la altura que tenía previamente a la inserción que produjo el desbalanceo.

Observar que el AVL recupera la altura previa a la inserción que produjo el desbalanceo.

Rotaciones dobles Caso 2 antihoraria-horaria: como se ha indicado antes, estas rotaciones (la trivial y la no trivial) son imágenes especulares de las homólogas del caso 1.

Borrado en un AVL:

Cuando se lleva a cabo un borrado, el desequilibrio se produce cuando la eliminación de un nodo trae como consecuencia el no cumplimiento de la propiedad del AVL.

En principio, el borrado en un AVL se lleva a cabo como en cualquier ABB;

y luego se considera lo siguiente:

Si la supresión es en el subárbol izquierdo del pivote, entonces debe analizarse la situación del subárbol derecho, ya que los algoritmos de rotación que deban realizarse dependen de la relación entre las alturas de los subárboles del subárbol derecho( las cuales no pueden ser 0).

Análogo razonamiento se hace para un borrado en el subárbol derecho del pivote, pero considerando el subárbol izquierdo.

Este es el análisis y las rotaciones a realizar en cada caso:

*derecho*

Desequilibrio por supresión en el subárbol izquierdo del pivote:

*izquierdo*

Los algoritmos dependen de la relación entre las alturas del subárbol derecho

*izquierdo*

del pivote (el subárbol derecho no puede ser 0).

*izquierdo*

Relación posible entre las alturas del subárbol derecho:

*izquierdo*

*antihoraria*

Alturas iguales entre los subárboles del subárbol derecho: realizar rotación horaria.

En este caso, el árbol resultante la misma altura que antes de la rotación.

Alturas distintas:

*izquierdo*

Analizar relación entre las alturas de los subárboles del subárbol derecho.

*derecho*

*izquierdo*

*antihoraria*

Altura del subárbol izquierdo menor que altura del subárbol derecho: realizar rotación horaria.

El árbol resultante tiene una altura inferior en 1 a la altura previa al borrado; se vuelve necesario examinar los subárboles englobados para detectar y corregir eventuales desbalanceos.

*derecho*

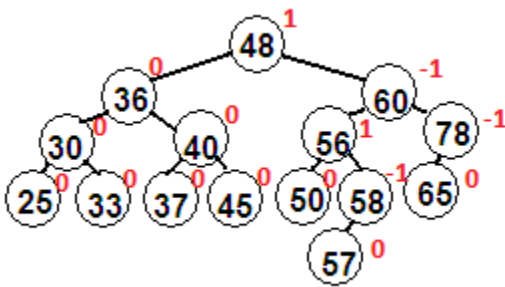
*izquierdo*

*antihoraria-horaria I D*

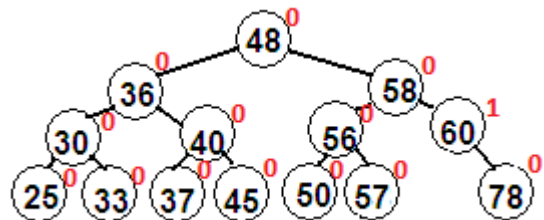
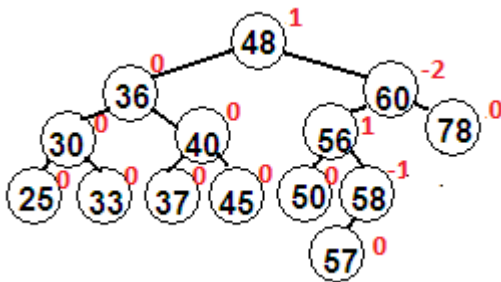
Altura del subárbol izquierdo mayor que altura del subárbol derecho: realizar rotación horaria-antihoraria D I.

El árbol resultante tiene una altura inferior en 1 a la altura previa al borrado; se vuelve necesario examinar los subárboles englobados para detectar y corregir eventuales desbalanceos.

Ejemplo:



En el este árbol se realiza el borrado del 65, lo cual provoca las modificaciones en los FB indicadas en el cuadro de abajo a la izquierda, y tras la rotación, la resolución de la derecha.



Los estudios empíricos indican que, en promedio se realiza una rotación cada 2 altas y una rotación cada 5 supresiones, y que ambas rotaciones (simples y dobles) son equiprobables.

Eficiencia de un AVL:

La intención de las estructuras de árbol balanceadas por la altura, como los árboles AVL, es disminuir la complejidad temporal de las operaciones de alta, baja y búsqueda en un ABB.

El algoritmo de búsqueda es idéntico al de un ABB común, por lo cual el coste temporal dependerá del número de niveles del árbol.







El proceso de recalculer los FB luego de realizada un alta o una baja involucran a los nodos que están en el camino desde el punto en el que se hizo la alteración, y hasta la raíz, es decir tantos nodos como niveles tiene el árbol (Es frecuente que para mejorar los tiempos de estos cálculos, cada nodo tenga un enlace con su nodo padre, y almacenada la altura del subárbol del cual él mismo es raíz). En el caso del alta, dado que los punteros que pueden ser reasignados para desbalancear son a lo sumo 5, el coste temporal del algoritmo depende de la altura del AVL. Se puede indicar que es  $O(m)$ , siendo  $m$  el número de niveles del AVL. En el caso de la baja, en algunos casos puede llegar a ser necesario analizar subárboles englobados, pero el número de estos análisis no hace crecer el coste temporal del algoritmo más allá de  $O(m)$ , siendo  $m$  el número de niveles del AVL.

La complejidad de esas tres operaciones depende, pues, de la altura del árbol.

#### Relación entre la altura de un AVL y la menor altura de un ABB:

Consideremos árboles AVL lo más malos que sea posible, es decir, que, para una altura determinada tengan la menor cantidad de nodos que sea posible. Esta es la peor situación para un árbol, dado que se “desaprovecha” la capacidad de cada nivel del mismo para contener nodos; se ha visto que de cada nivel de un ABB se analiza, en los procesos de alta, baja o búsqueda, un nodo por nivel, por tanto, cuántos más nodos haya por nivel, más serán “descartados” en el rastreo, y el proceso será tanto más eficiente. Veremos pues, cual es la peor situación que puede darse para un AVL, para sacar algunas conclusiones.

En la siguiente tabla se muestra que ocurre (los dibujos son orientativos, la forma del árbol AVL puede variar, pero el número de nodos, no):

1	2	3	4	5	6
1	2	4	7	12	20
					

Se observa una relación entre la cantidad mínima de nodos para un árbol AVL de determinada altura y las cantidades correspondientes a los precedentes en altura.

Llamando  $F(h)$  a la menor cantidad de nodos para un árbol AVL de altura  $h$ , se puede indicar:

$$F(0)=0$$

$$F(1)=1$$

$$F(h) = F(h-1) + F(h-2)+1$$

Esta relación tiene similitud con la sucesión de Fibonacci, por lo cual a estos árboles se los llama árboles de Fibonacci. Un árbol de Fibonacci es un árbol AVL con la cantidad de nodos mínima.

Estos árboles representan la mayor ‘deformación posible’ de un AVL, en el sentido de que ‘desperdician’ la mayor cantidad de espacio disponible en los niveles.

Se demuestra que, si  $F(h)=n$ , entonces  $h$  es aproximadamente  $1.44 * (\log n)$ .

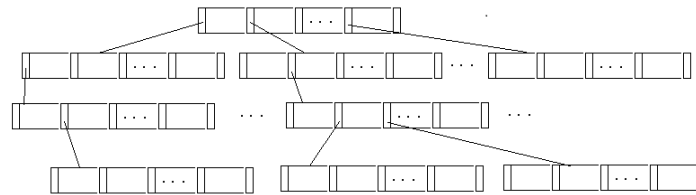
Entonces, un árbol AVL tiene una altura que, a lo sumo (si es árbol de Fibonacci) es de  $1.44 * \log n$ . Esto indica que el coste del alta, la baja y la búsqueda en un AVL pertenecen a  $O(\log n)$ .

Ejercicios propuestos:

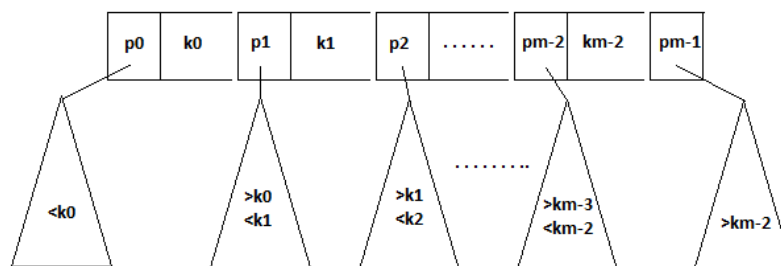
1. En el peor caso ¿cuántas rotaciones hay que hacer para un alta en un AVL de altura  $h$ ?
2. En el peor caso ¿cuántas rotaciones hay que hacer para un borrado en un AVL de altura  $h$ ?

### Árboles multivía:

Estos árboles están formados por nodos que tienen capacidad para  $m$  enlaces y  $m-1$  claves.



Formato del nodo:



Un árbol de  $m$  vías está formado por nodos que pueden contener hasta  $m-1$  claves ( $k_0, k_1, k_2, k_3, \dots, k_{m-2}$ ).



Las claves están ordenadas de forma creciente.

No hay “posiciones vacías” entre dos claves. Si hay una clave en la posición  $i$  y hay otra clave en la posición  $i+2$ , entonces hay clave en la posición  $i+1$ . Si hay clave en la posición 1, entonces hay clave en posición 0 (la primera según nuestra convención).

Los subárboles hijos verifican que las claves de los mismos Las claves del primer subárbol son menores que  $K_0$ , las claves del segundo subárbol son mayores que  $K_0$  y menores que  $K_1$ , etc. Las claves del último subárbol son mayores que la última clave,  $k_{m-2}$ .

Los árboles de  $m$  vías pueden crecer desbalanceados, lo cual resta mucha eficiencia. Para garantizar el balanceo y mejorar su funcionamiento se desarrollaron a partir de ellos los árboles B.

### **Árbol B**

Son estructuras diseñadas por R.Bayer y E.McCreight en 1972.( <http://www.inf.fu-berlin.de/lehre/SS10/DBS-Intro/Reader/BayerBTree-72.pdf>)

Definición: un árbol B es un árbol de  $m$  vías que verifica

- Todos los nodos excepto la raíz están completos con claves al menos hasta la mitad.
- La raíz, o bien es hoja, o bien tiene al menos dos hijos.
- Si un nodo no hoja tiene  $h$  claves almacenadas, entonces tiene  $h+1$  hijos.
- Todas las hojas están en el mismo nivel.

Por lo anterior, es un árbol de  $m$  vías balanceado.

### **Alta en B**

El esquema del algoritmo se muestra a continuación:

Alta(clave  $x$ )

Pos <- Ubicar\_posición\_hoja\_adeuada

//si el árbol esta nulo, pos será nula; de otro modo, se “desciende en el árbol hasta la hoja correcta”

Clave\_ubicada= falso

Mientras ( clave\_ubicada == falso)

{

Si (Pos es nula)

{

```
Generar nodo

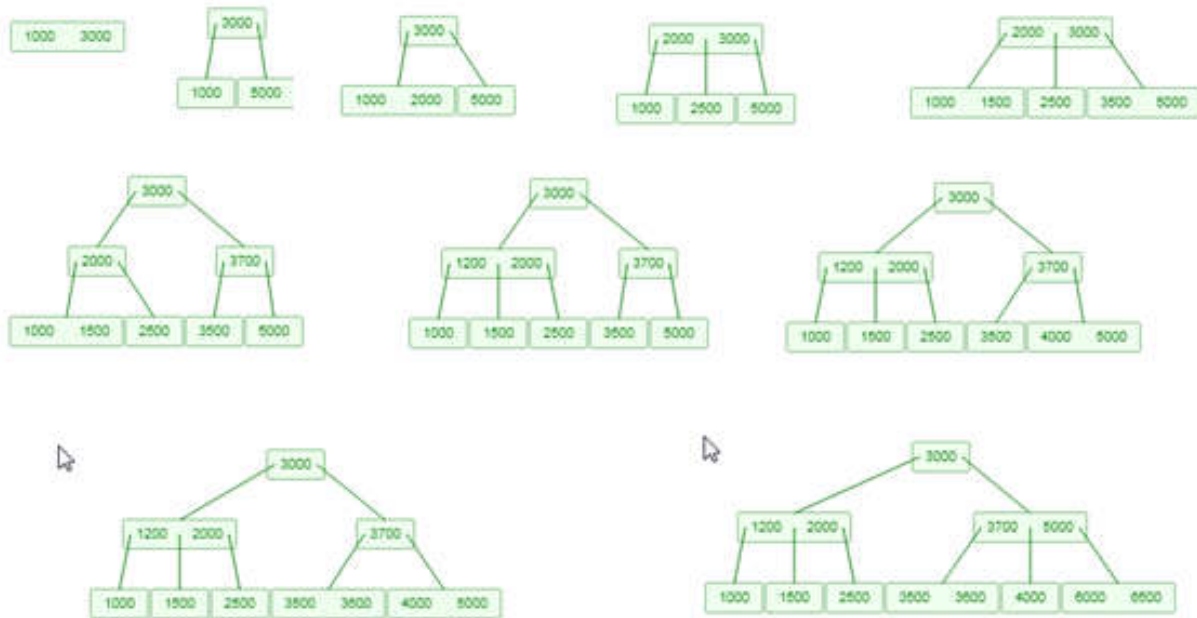
Almacenar clave x

Clave_ubicada <- verdadero

Asignar los valores a los enlaces de este nodo
}
Si no
{
    Si (hay lugar en nodo señalado por Pos)
    {
        Ubicar clave x en nodo de forma ordenada
        Asignar los valores a los enlaces de este nodo
        Clave_ubicada =verdadero
    }
    Si no
    {
        Val=Determinar valor central de secuencia asociada al nodo apuntado desde Pos
        Generar nodo nuevo
        Claves menores que el valor central se acomodan en el nodo izquierdo
        Claves mayores que el valor central se acomodan en el nodo derecho
        //Se continúa con el nodo padre de los dos involucrados; ahí se intenta ubicar el valor central
        Pos =Ubicar_posición_del _nodo _padre
    }
}
}
```

Ejemplo: si se da de alta la siguiente secuencia en un árbol B de 3 vías (también llamado árbol 2-3) inicialmente vacío, éste evolucionará como se muestra a continuación:

Secuencia de claves: 3000, 1000, 5000, 2000, 2500, 1500, 3500, 1200, 4000, 3600, 6000, 6500.



### Baja en árbol B

El esquema del algoritmo se indica a continuación:

Eliminar (clave x)

Pos <- Ubicar\_nodo\_clave\_x

Si (nodo señalado por Pos es hoja)

{

Eliminar clave sin dejar “huecos” entre las mismas

Finalizar <- falso

Mientras (Finalizar == falso)

{

Si (quedaron menos claves que la cantidad mínima en nodo señalado por Pos)

{

Si existe hermano izquierdo o derecho de ese nodo con más claves que la cantidad mínima

{

Realizar una rotación adecuada para balancear

```
Finalizar<-verdadero
```

```
}
```

```
Si no //hay menos claves que el mínimo y no se puede “tomar” de un hermano para balancear
```

```
{
```

```
Fusionar nodos hermanos, bajando la clave del nodo padre que separaba los nodos involucrados
```

```
Pos <- Ubicar_nodo_padre_de los fusionados
```

```
}
```

```
}
```

```
Si no //el nodo tiene suficientes claves
```

```
{
```

```
Finalizar<-verdadero
```

```
}
```

```
} //fin mientras
```

```
Si no //nodo no es hoja
```

```
{
```

```
Reemplazar clave x por menor de los mayores o mayor de los menores
```

```
// ese reemplazante está en un nodo hoja
```

```
Pos<-Ubicar_nodo_hoja_del_reemplazante
```

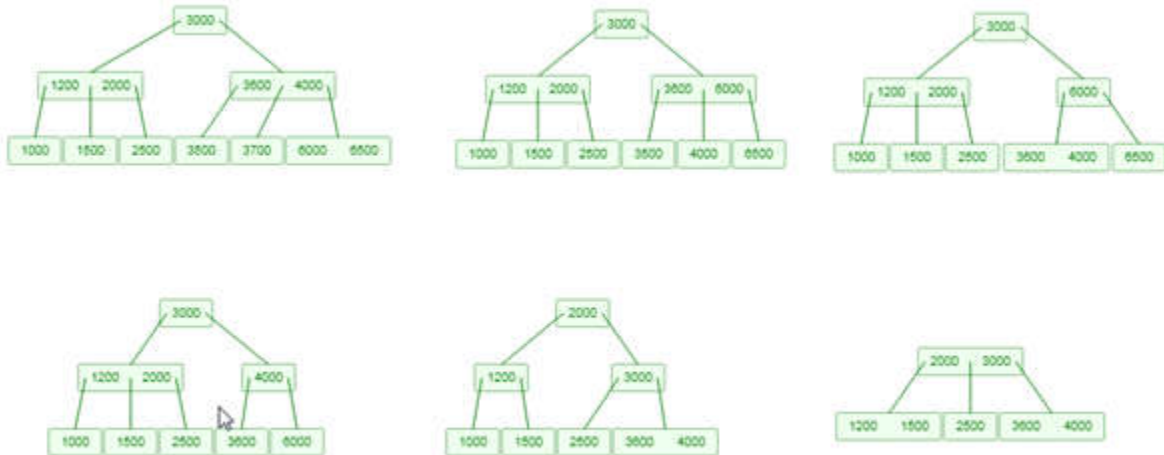
```
Eliminar (reemplazante de x)
```

```
}
```

Ejemplo:

Si en el árbol B del ejemplo anterior se dan de baja sucesivamente las claves que se indica, la evolución de la estructura será como se muestra en los siguientes gráficos.

Secuencia de claves a eliminar: 5000, 3700, 3500, 6500, 6000, 1000



### Mínima cantidad de claves de un árbol B con h niveles

Tomando  $q = \lceil m/2 \rceil$ , consideramos el peor árbol B de m vías, en el sentido de tener la menor cantidad de claves posibles por nivel, y es:

Número de nivel	Cantidad mínima de claves en el nivel
1 (raíz)	1
2	$2(q-1)$
3	$2q(q-1)$
4	$2q^2(q-1)$
.....	.....
t	$2q^{t-2}(q-1)$

Entonces la mínima cantidad de claves en los t niveles es  $1 + (q-1) \sum_{i=1}^{t-2} 2q^i$

Operando, mínima cantidad de claves en los t niveles =  $-1 + 2q^{(t-1)}$

Si hay  $n$  claves almacenadas,  $n \geq -1 + 2q^{(t-1)}$

Y se deduce  $h \leq (\log_q((n+1)/2)) + 1$

### Array de bits:

En esta implementación se utilizan los bits de un array para indicar si un elemento (dado por el subíndice) está o no en el conjunto.

Ejemplo: consideremos un byte, es decir, una secuencia de 8 bits  $b$  en el cual se ha implementado un conjunto  $A$ . Los elementos que pueden almacenarse en  $A$  serán por tanto los enteros de 0 a 7

Entonces, esta situación del byte:

0	0	1	0	1	1	0	1
7	6	5	4	3	2	1	0

Indica que en el conjunto  $A$  están los elementos 2, 0, 3 y 5.

La convención es que un 1 en la posición  $i$  indica que  $i$  está en el conjunto, y un 0 indica que no está.

Para implementar las operaciones de alta, baja y búsqueda de un elemento hay que tener en cuenta que un byte se manipula completo, es una unidad en sí; es imposible manipular cada bit por separado. Eso nos obliga a utilizar “máscaras” implementadas también a través de bytes para modificar el valor de un bit en particular, o bien para testear su contenido y establecer si está en 0 o en 1.

Para, por ejemplo, dar de alta al elemento 4 en el conjunto tenemos que poner a 1 el valor de esa posición dejando inalterados los otros. Eso se puede lograr usando una máscara como la siguiente:

0	0	0	1	0	0	0	0
7	6	5	4	3	2	1	0

Con esta máscara, si se realiza la operación or entre ella y el byte  $b$ , modifica la situación del bit de posición 4, pasándolo a 1.

Observar que la máscara tiene 0 en todas las posiciones excepto en aquella que queremos “encender” a través de un or.

Así, el conjunto  $A$  quedará como:

0	0	1	1	1	1	0	1
7	6	5	4	3	2	1	0

Ahora consideremos que queremos eliminar el elemento 2 de conjunto.

Se puede usar para ello esta máscara:

1	1	1	1	1	0	1	1
7	6	5	4	3	2	1	0

y operar con un and entre ella el byte b donde está implementado A.

La máscara tiene 1 en todas las posiciones, excepto en aquella cuyo valor queremos “apagar” a través del and.

De este modo, A quedará así:

0	0	1	1	1	0	0	1
7	6	5	4	3	2	1	0

Para determinar si un elemento, por ejemplo el 5, está en el conjunto A, se puede operar con una máscara de esta forma:

0	0	1	0	0	0	0	0
7	6	5	4	3	2	1	0

Y operar con un and. Si luego de la operación el byte resultado se identifica con un 0, el elemento no está.

Ahora bien: si pensamos en una máscara que se prepara de diferente forma según la posición del elemento a manipular, el algoritmo puede ser muy ineficiente. En lugar de ello se puede plantear siempre que, para manipular el elemento x,

Utilizamos un byte con 00000001 (siete 0 y un 1 al final)

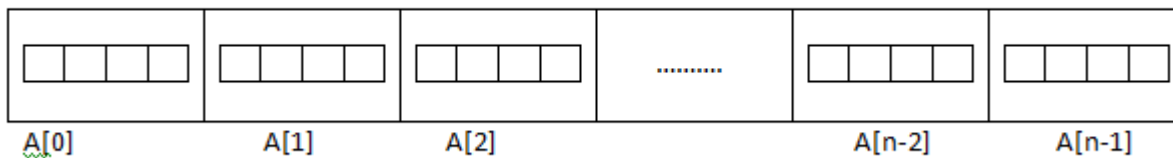
Desplazamos los bits del byte x posiciones a izquierda (con lo cual el 1 quedará ubicado en la posición x).

Luego, si queremos dar de alta a x, la máscara ya está lista para operar con el or.

Si, en cambio, queremos dar de baja un elemento, hay que invertir los unos y los ceros de la máscara, lo cual se puede efectuar negando la misma, luego de lo cual operamos con un and.

Ahora vamos a generalizar la situación, ampliando las posibilidades de almacenamiento: en lugar de trabajar con un byte, trabajaremos con un array de enteros. Cada posición del array tiene un tamaño  $s$ , correspondiente al tamaño (sizeof) de un entero, con lo cual la capacidad se amplía considerablemente.

**int A[n];**



Ahora, lo que antes planteamos para un byte aislado se llevará a cabo en alguno de los bytes que componen alguno de los enteros del array.

Considerando  $n$  enteros en el array, para trabajar almacenando, borrando o determinando la pertenencia al conjunto del elemento  $k$ , lo importante es establecer en qué int del array y en qué bit de ese int se ubica  $k$ .

Puede hacerse lo siguiente:

$I = k / (s * 8);$  //  $s$  es sizeof(int)

$Pos = k \% (s * 8);$  // % corresponde al operador módulo

y, para preparar la máscara para operar con  $k$  (tener en cuenta que estamos usando los operadores de C/C++ ; en otros lenguajes el operador puede ser distinto)

`unsignedint f=1; // f= 00000...00001`

`f= f << pos; // desplazar el 1 k posiciones a izq`

Alta de  $k$  en el array de bits (implementación de la primitiva alta en el conjunto):

$A[i] = A[i] | f;$

Borrado de  $k$  del array de bits (implementación de la primitiva baja en el conjunto):

$f = \sim f;$  // Modificar la máscara, invirtiendo los bits

$A[i] = A[i] \& f;$



Determinación de la pertenencia o no de k al array de bits: (implementación de la primitiva esta o pertenece en el conjunto):

```
if (A[i] & f)
    { // k pertenece al conjunto}
else
    {k no pertenece al conjunto}
```

Coste: estas operaciones en array de bits son de tiempo constante. El alta, la baja y la búsqueda tienen un coste temporal que pertenece a  $O(1)$ . Si embargo hay que tener en cuenta que solo permiten trabajar con claves numéricas enteras. Si las claves fueran de otra naturaleza, habría que considerar el costo de mapear las mismas sobre el conjunto de enteros correspondiente.

Ejercicio propuesto:

Indicar cómo se puede realizar la unión, intersección, diferencia y diferencia simétrica entre conjuntos con esta implementación.

## **Trie**

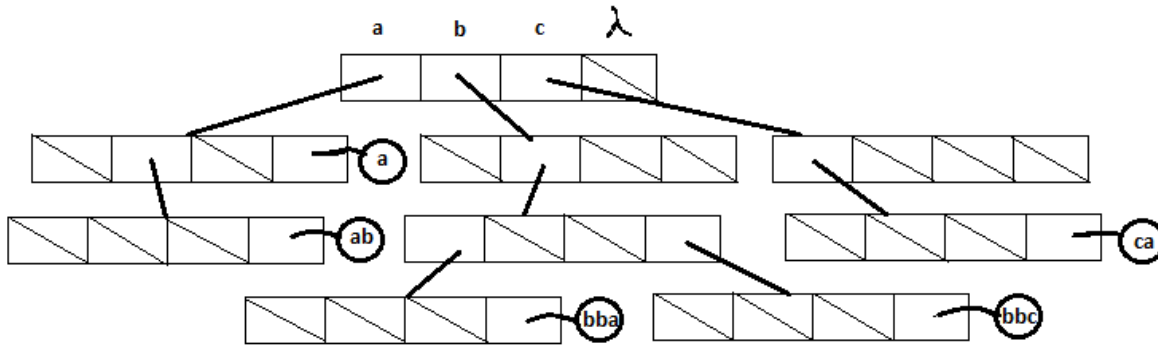
Estructura que permite el almacenamiento de cadenas de símbolos de un alfabeto finito compuesto por  $s$  símbolos. Los caracteres que forman cada clave no se encuentran almacenados de forma contigua. (de la Briandais, 1959) (ver <https://www.computer.org/csdl/proceedings/afips/1959/5054/00/50540295.pdf>, <https://doc.lagout.org/Others/Data%20Structures/Advanced%20Data%20Structures%20%5BBrass%202008-09-08%5D.pdf>)

El concepto de trie puede plasmarse de distintas formas; hay diversas implementaciones. Aquí mostraremos algunas de ellas.

### Implementación de un trie en árbol:

En una implementación como árbol, un trie de orden  $m$  o bien es nulo, o bien es una secuencia ordenada de exactamente  $m$  tries de orden  $m$ . Cada nodo es un array de punteros.

Si el conjunto de caracteres o símbolos es  $\{a,b,c,\lambda\}$ , en el siguiente trie se han almacenado las claves: a,ab,bba,ca,bbc.



Esquema del alta de una clave x. Toda clave es una sucesión de caracteres o símbolos

Alta de un elemento

```
{
//aux: puntero a nodo
//p es puntero a la raíz del árbol
//pos: posición del carácter a considerar
aux = p;
pos=0; //consideramos el primer símbolo de la clave
Mientras (secuencia de caracteres de la clave no haya terminado)
{
    Si ( aux[pos] no nulo)
        {aux=aux[pos];}
    Si no
        {
            Generar nodo nuevo desde aux[pos];
            Poner todos los punteros del nodo nuevo en nulo;
            aux=aux[pos];
        }
    pos = pos + 1 ; //Avanzar al siguiente carácter de la clave;
```

}

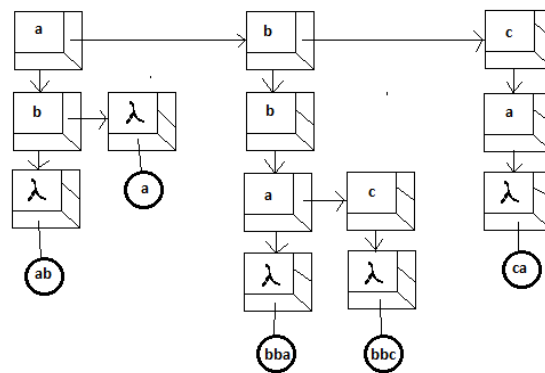
Almacenar en posición correspondiente al terminador la dirección del valor asociado con la clave

}

La baja se lleva a cabo mediante un movimiento que parte del enlace en la hoja y retrocede poniendo a nulo los punteros correspondientes; si el array de punteros quedara nulo, se libera la memoria que ocupaba y se pone a nulo el puntero correspondiente del nodo padre (se deja como propuesta desarrollar el algoritmo).

#### Implementación de un trie con listas de listas :

Como alternativa para reducir el costo espacial de la implementación anterior, se puede considerar una implementación con listas. Sólo se almacenan los nodos de símbolos que correspondan a caracteres de claves almacenadas. En el ejemplo que se presenta a continuación, se ha implementado el mismo conjunto del ejemplo anterior:



Ejercicios propuestos:

1. Desarrollar el algoritmo de búsqueda de una clave en un trie implementado con un árbol.
2. Escribir el algoritmo de baja para un trie implementado con un árbol.
3. Analizar el coste temporal del alta, baja y búsqueda de claves en un trie implementado con un árbol.
4. Describir una estructura adecuada para implementar el trie con listas, definir los algoritmos para el alta, la baja y la búsqueda, y analizar el coste de dichas operaciones.
5. Para las dos implementaciones vistas, diseñar algoritmos para obtener la unión y la intersección de conjuntos
6. Para las estructuras vistas, realizar un análisis comparativo del coste temporal y espacial para las implementaciones de cada una de las operaciones básicas de un Conjunto. Discutir en qué situaciones elegiría una u otra implementación para el TDA Conjunto y Diccionario.