TEMA 0: Retomando desde FunPro

Introducción

Algoritmo: conjunto de instrucciones claras, no ambiguas, y finitas que son ejecutadas en un orden determinado y permiten resolver un conjunto de problemas.

Estructuras de control:

- Secuencial (;).
- Selección: elegir el camino de ejecución según determinadas condiciones. (`if`, `if...else`, `switch`).
- Iterativas (`while`, `do...while`, `for`).

Utilizando estos tipos estas pueden combinarse de dos maneras:

- Apilando: uno después del otro.
- Anidando: uno dentro del otro.

Relación entre algoritmo y programa: un algoritmo puede ser un programa, pero no necesariamente. Un programa debe ser un algoritmo escrito en un lenguaje entendible por la PC.

Abstracción: obtener datos o propiedades relevantes para el dominio que se va a trabajar.

Struct

Estructura útil para tener una colección de valores de diferentes tipos y tratar la colección como una sola cosa.

```
struct CuentaCD //El nombre es la ETIQUETA de la estructura
{
//miembros:
double saldo;
double tasa_interes;
int plazo; //meses hasta el vencimiento
};
```

Variables de estructura, variables miembro y operador punto

Una **variable de estructura** es aquella que lleva como "tipo de dato" el nombre del struct:

```
CuentaCD mi_cuenta, tu_cuenta; //Definición de variables de la estructura
```

Una **variable miembro** es aquella que existe y se declara *dentro* del struct, ej.: *saldo, tasa_interes, plazo.*

Al igualar dos variables miembro, se asigna a la segunda todos los datos contenidos en los miembros de la primera. Ej.: *mi_cuenta = tu_cuenta* equivale a *mi_cuenta.saldo = tu_cuenta.saldo*, etc.

El **operador punto** se utiliza para especificar una variable miembro de una variable de estructura.
mi_cuenta.saldo = 0;

Estructuras como argumentos de función

Una función puede tener **parámetros de llamada** por valor de un **tipo** de **estructura** y/o **parámetros de** llamada por **referencia** de un tipo de estructura.
Ej.: `void imprimir (CuentaCD &mi_cuenta) {...}`

Una función también puede **devolver** un struct.
Ej.: `CuentaCD empaquetar(double el_saldo, double la_tasa, int el_plazo) {...}` retorna un struct del tipo `CuentaCD`.

Inicialización de estructuras

Para inicializar una estructura al momento de su declaración deben colocarse los datos entre llaves y separados por coma. Puede o no utilizarse como constante.
Ej.:

```
struct Fecha {
int dia, mes, año;
}
Se puede inicializar como:
const Fecha vencimiento = {15, 03, 2025};
```

Los valores se asignan en orden, es decir, dia = 15, mes = 03 y año = 2025.

TEMA 1: Asignación dinámica de memoria

Punteros

Un **puntero** es una **dirección de memoria** y puede ser almacenado en una variable **de tipo puntero**. Estas pueden contener la dirección de memoria en la que se almacena otra variable, en cuyo caso el puntero se define con el tipo de dato que contiene la dirección de memoria a la que se apunta, utilizando el operador ***&*** para acceder a la dirección de memoria de la variable. Además, al momento de querer *ver* que es lo que tiene esta variable a la que se está apuntando o modificarla, debe utilizarse el operador ***** para ***desreferenciar*** el puntero.
Ej.:

```
int numero = 5;
int* puntero_a_numero = &numero; //Definición de puntero (tipo int*) y asignación a esta de la memoria en la que se almacena la variable numero.
cout<<*puntero_a_numero; //Desreferenciar el puntero e imprimir el valor de la variable numero (5).
```

Operador *new*

Genera una nueva variable *sin nombre* y hace que el puntero interviniente apunte a ella. Ej.:

```
int* puntero = new int; //Almacena la dirección de memoria de esta nueva variable sin nombre de tipo
*puntero = 4; //Almacena en esa dirección de memoria el valor 4.
```

Las variables creadas mediante este operador se denominan ***variables dinámicas***, ya que se crean y destruyen mientras el programa se está ejecutando.

Administrador de memoria básica

Se reserva un área especial de la memoria (***heap***) para las variables dinámicas.
Al dejar en desuso una variable dinámica es buena práctica utilizar el operador ***delete*** para liberar la memoria de la forma: *delete puntero;*. Al eliminar una variable dinámica el valor del puntero queda *indefinido* y se convierte en un ***puntero colgante*** y al tratar de desreferenciarla el resultado es impredecible y por lo regular desastroso.

Variables estáticas, dinámicas y automáticas

Las **variables dinámicas** son aquellas que se crean mediante el operador *new* porque se crean y se destruyen mientras el programa está en ejecución.

Las **variables automáticas** son aquellas en las que sus propiedades dinámicas se controlan de forma automática; se crean y destruyen automáticamente.

Definiciones de tipos

Para asignar un nombre a una definición de tipo y luego usar el nombre de ese tipo para declarar variables se usa ***typedef*** de forma: `typedef in* ApuntInt;`

Arreglos dinámicos

Es un tipo de array donde **no se especifica el tamaño** al momento de escribir el programa, si no durante la **ejecución**.

Variables de arreglo y variables de puntero

Un *array* es un puntero que apunta a la primera variable indicada del mismo. Dado a que estos se manejan con memoria estática, puede utilizarse el operador *new* para crear variables dinámicas que sean arrays y tratarlas como si fueran arreglos normales.

Ej.:

```
int t = 5;
double* p = new double[t];
delete [] p;
```

Los corchetes indican que se quiere eliminar una variable de arreglo dinámico, así que el sistema verifica el tamaño del arreglo y elimina ese número de variables. Sin ellos, solo eliminaría el espacio de una variable de tipo *int*.

Aritmética de punteros

Suponiendo un puntero dinámico *p*, la suma *p+1* devuelve la dirección de memoria de *p[1]* ya que avanza a la siguiente posición de memoria. Por tanto, **(p+1)* imprime el valor almacenado en *p[1]*. También pueden compararse dos punteros para saber si apuntan a la misma variable dinámica o compararse con NULL.

Arreglos dinámicos multidimensionales

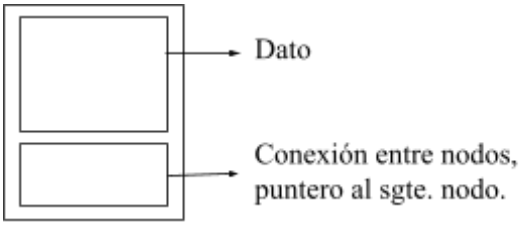
Para crear un array de arrays se debe, primeramente, definir un array bidimensional tipo *int** y luego crear un array dinámico de *int*s para cada variable del arreglo inicial.
Ej. para crear un arreglo dinámico de 3x4:

```
int** m = new int* [3];
for (int i = 0; i < 3; i++) m[i] = new int[4];
```

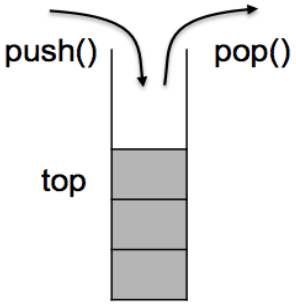
TEMA 2: Estructuras de Datos Lineales

Nodos

Estas estructuras están compuestas por un conjunto de ***nodos*** anidados entre sí. Estos son elementos que contienen un dato y un puntero al siguiente nodo (cada uno tiene exactamente un predecesor y un antecesor, excepto el primero que solo tiene sucesor y el último que solo tiene predecesor y apunta a NULL). Los ***nodos*** pueden implementarse como *struct* o clases y se generan dinámicamente, es decir, en tiempo de ejecución.

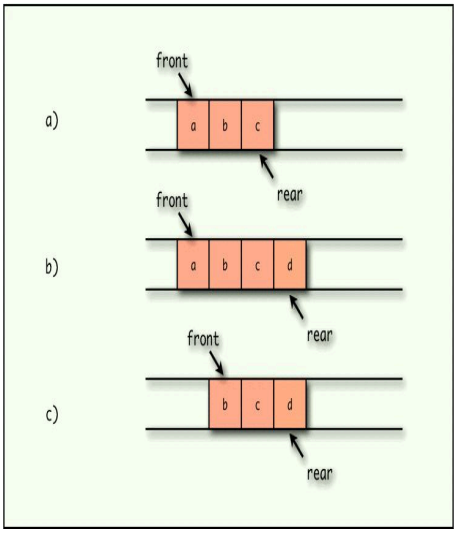


Pilas (Stack)



Estructura de datos vertical que recupera datos en orden inverso al que están almacenados (LIFO). La inserción y borrado de elementos se realiza sólo por *uno de los extremos*, es *restrictiva*, por lo que solo se permiten tres operaciones: ***push***, ***pop***, ***preguntar si está vacía***, y de *acceso destructivo*, por lo que al sacar un elemento este se destruye. El último elemento apuntará a NULL.
Si se intentara obtener un elemento de una pila vacía se generaría un ***underflow*** o desbordamiento negativo y si se intentara agregar un elemento a una pila llena se produciría un ***overflow*** o un desbordamiento.
Ejemplos de pilas: cadena de invocación de llamadas recursivas, pasaje de decimal a binario, etc.

Colas (Queue)



Es una estructura de datos vertical que recupera los datos en el mismo orden que se agregan, es decir, es de tipo FIFO. Al igual que la pila es *restrictiva*, por lo que solo se permiten tres operaciones: ***push***, ***pop***, ***preguntar si está vacía***, y de *acceso destructivo*, por lo que al sacar un elemento este se destruye.

Implementación de las colas

Existen dos formas de implementar una cola:

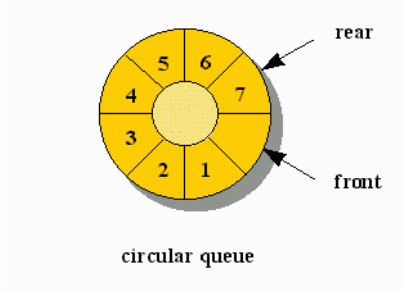
- De manera ***estática*** con el uso de vectores donde el dimensionado y alojamiento de memoria se define en tiempo de compilación.

- De manera ***dinámica*** donde no tienen tamaño fijo y el alojamiento se realiza cuando se necesita en la ejecución del proceso de encolado.

Implementación dinámica

La implementación se basa en el uso de estructuras autoreferenciadas y la definición de un ***TDA*** con su estructura y las tres funciones básicas permitidas (pop, push, isEmpty)
Una ***cola dinámica*** se implementa con dos punteros: uno apuntando al primer elemento (*frente*), desde donde se obtienen los elementos, y otro al último elemento (*fondo*), desde donde se agregan elementos. El último elemento, ubicado en el *fondo,* apuntará a NULL.
Ejemplos de colas: cola de impresión, cola del supermercado.

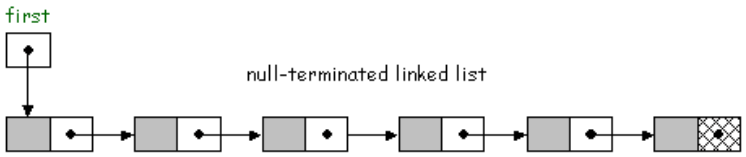
Cola con un array circular



Dado que la operación quitar se realiza desde el frente y lo hace avanzar al siguiente nodo sucede que las posiciones anteriores quedan desocupadas, pero no accesibles, lo que hace posible que se pueda percibir la cola como llena cuando hay posiciones del array sin ocupar. Es por esto que se presenta la ***cola circular***, donde el último elemento referencia al primero. Conserva los punteros *frente* y *cola*, donde el primero apunta al primer elemento y el segundo contiene la posición del último elemento. Ambas avanzan en sentido de las agujas del reloj. Esta representación se realiza con arrays (*memoria estática*) y es útil para el control de tamaño de memoria ya que lo define.

Lista simplemente enlazada

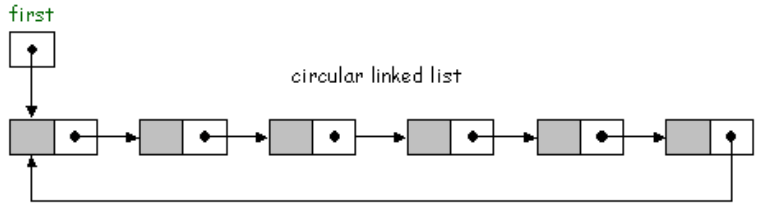
Es una estructura de datos que, a diferencia de las pilas y las colas, puede recuperar los datos en el cualquier orden, no tiene restricciones ni es de acceso destructivo. Estas estructuras necesitan de un un puntero al inicio para su acceso.



Una lista puede ser una ***multilista*** (lista de listas), la cual utiliza dos tipos de lista (principal y auxiliar) para definir una estructura multinivel.

Lista simplemente enlazada circular

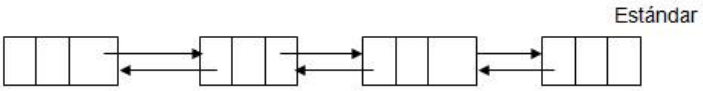
Esta estructura cuenta con las mismas características que la LSE, la diferencia es que en este caso el último elemento apunta al primero. Al agregar un nuevo nodo este debe apuntar a sí mismo (y no a ^).
Ejemplo de LSE circular: cuando el procesador hace “varias tareas” es que hace un poco de cada tarea y pasa a la siguiente, por lo que el último nodo debe conectarse al primero.



Diferencia entre Lista Circular y Cola Circular: Son estructuras de datos diferentes. Si bien la estructura de los nodos es equivalente, las listas permiten el acceso y las operaciones en cualquiera de sus nodos mientras que las colas solo permiten inserción al final y eliminación al frente (FIFO), además son de acceso destructivo y por esto es que no permiten recorrido. Además, las listas utilizan memoria dinámica con nodos enlazados y las colas suelen utilizar arrays.

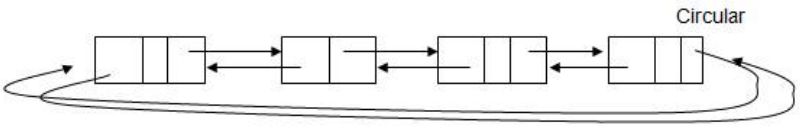
Lista doblemente enlazada

Tiene las mismas características que la LSE y LSE circular, con la diferencia de que cada nodo referencia al siguiente y al anterior (excepto el primer nodo que apunta al siguiente y a ^ y el último que apunta al anterior y a ^).
Ejemplo de lista doblemente enlazada:



Lista doblemente enlazada circular

Consta de la misma estructura que una LDE con la particularidad de que el último nodo referencia al primero y viceversa.



TEMA 3: Estructuras de Datos No Lineales

Árbol

Es una estructura de datos que consta de un conjunto finito de elementos (***nodos***), un nodo especial (***raíz***), por el cual se accede a la estructura, y un conjunto finito de líneas dirigidas (***ramas***) que conectan los nodos.

Un nodo puede considerarse ***padre*** si tiene sucesores. Si dos o más nodos comparten el mismo ***padre*** se consideran ***hermanos***. Un nodo que no tiene ***hijos*** (*nodo terminal*) se denomina nodo ***hoja*** y un nodo que no es hoja se denomina ***nodo interno***. Un ***ascendente*** de un nodo es aquel es el padre o ascendiente del padre, un ***descendiente*** es un hijo o descendiente del hijo.

Este tipo de estructura es *recursiva*, ya que cada nodo interno forma un árbol y un nodo de ese subárbol forma otro y así sucesivamente.

Definiciones generales:

- ***Camino:*** secuencia de nodos en los que cada uno de estos es adyacente al siguiente. Cada nodo es accesible siguiendo un único camino (comenzando desde la raíz). No hay ciclos en un árbol, si se siguen las ramas se llega a un "final".
- ***Longitud:*** es la cantidad de nodos en el camino menos uno. Siempre existe un camino de longitud 0 de un nodo a sí mismo.
- ***Altura:*** es la máxima longitud de un camino que va desde el nodo a una hoja más uno.
- ***Profundidad:*** longitud del único camino que va desde la raíz al nodo.
- ***Nivel***: distancia de un nodo con la raíz. También puede definirse como todos los nodos que están a una misma *profundidad*. Considerando la definición de hermano se sabe entonces que los hermanos están en el mismo nivel, pero dos nodos en el mismo nivel no son necesariamente hermanos.
- ***Órden***: cantidad de sucesores que puede tener un mismo nodo.
- ***Grado***: cantidad de sucesores que posee un nodo.

En cada nodo del árbol puede avanzarse en dos direcciones:

- Por el hermano derecho: recorre la lista de hermanos de izquierda a derecha.
- Por el hijo más izquierdo: desciende lo más posible en profundidad.

Ejemplo de árbol: almacenamiento de archivos del OS.

Particionamiento del conjunto de nodos

Sea un nodo *n* y el conjunto *N* de todos los nodos en el árbol se puede dividir en 5 conjuntos *disjuntos*:
$N = \{n\} \cup \{descendientes(n)\} \cup \{antecesores(n)\} \cup \{derecha(n)\} \cup \{izquierda(n)\}$
Y el conjunto de los descendientes de los nodos hoja es el vacío (∅).

Estructura

Cada nodo contiene el campo *dato* y dos campos de *punteros*, uno al subárbol izquierdo y otro al subárbol derecho.

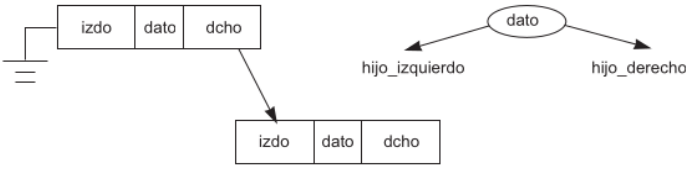


Figura 16.17. Representación gráfica de los campos de un nodo.

Equilibrio

El *factor de equilibrio* determina si un árbol está equilibrado y se calcula con la diferencia en altura entre los subárboles derecho e izquierdo: $B = h_D - h_I$. Si este resultado es *cero* el árbol está ***perfectamente equilibrado***, y está ***equilibrado*** si la altura de sus subárboles difiere en no más de uno.

Tipos

- **Árboles llenos**: todos los nodos hojas están al mismo nivel y cada nodo (excepto las hojas) posee la máxima cantidad de hijos.
- **Árboles completos**: todos los nodos (excepto las hojas) poseen la máxima cantidad de hijos.
- **Árbol degenerado**: todos los nodos (incluyendo la raíz) poseen un único sucesor. Su figura se asemeja a la de una LSE.

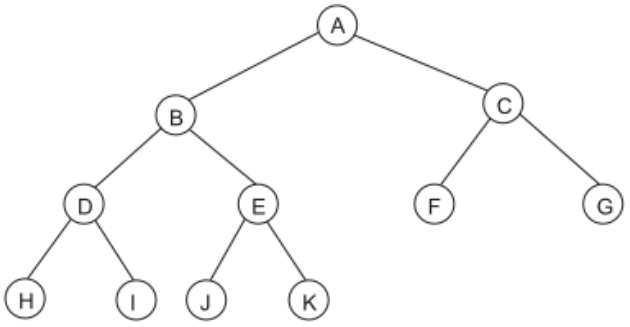
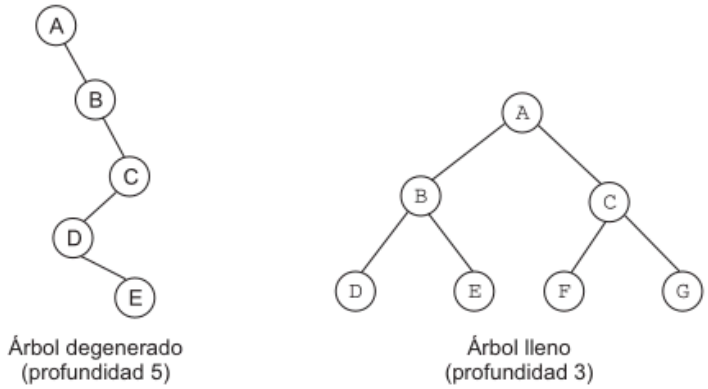


Figura 16.13. Árbol completo (profundidad 4).



Recorrido de un árbol

El **recorrido de un árbol** supone visitar cada nodo sólo una vez.

- En el **recorrido en profundidad** todos los descendientes de un hijo se procesan *antes* del siguiente hijo.
- En el **recorrido en anchura** *cada nivel* se procesa totalmente antes de que comience el siguiente nivel.

Preorden (RID)

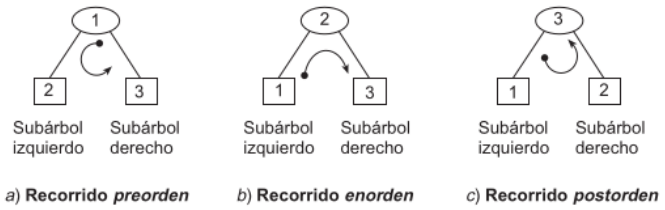
1. Visitar el nodo raíz (**R**).
2. Recorrer el subárbol izquierdo (**I**) en preorden.
3. Recorrer el subárbol derecho (**D**) en preorden.

Inorden (IRD)

1. Recorrer el subárbol izquierdo (**I**) en inorden.
2. Visitar el nodo raíz (**R**).
3. Recorrer el subárbol derecho (**D**) en inorden.

Postorden (IDR)

1. Recorrer el subárbol izquierdo (**I**) en postorden.
2. Recorrer el subárbol derecho (**D**) en postorden.
3. Visitar la raíz. (**R**).



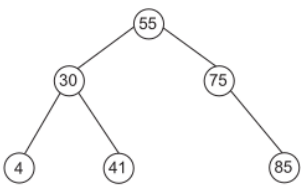
Árbol binario

Es aquel donde cada nodo puede tener como máximo dos subárboles y cada nodo puede tener uno, dos o ningún hijo. En cualquier nivel *n* puede contener de 1 a 2^n nodos.
Usos: almacenar y recuperar datos de forma eficiente.

Árbol binario de búsqueda

Un ***ABB*** es aquel que dado un nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que sus propios datos.

"Para cualquier nodo del árbol debe cumplirse que los valores del subárbol izquierdo son menores que su raíz y del subárbol derecho mayores que su raíz."



Operaciones en ABB

Búsqueda

En los ABB la búsqueda de una clave da lugar a un *camino de búsqueda*, de tal forma que *baja* por la rama izquierda si la clave buscada es menor que la clave de la raíz o *baja* por la rama derecha si es mayor.

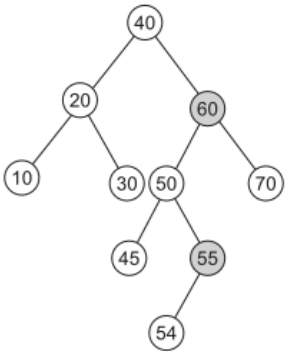
Inserción

La inserción de un nuevo nodo en un árbol de búsqueda siempre se hace como nodo hoja. Para ello se *baja* por el árbol según el camino de búsqueda.

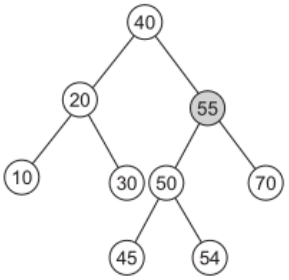
Eliminar

1. Buscar en el árbol la posición de "nodo a eliminar".
2. Si el nodo a suprimir tiene menos de dos hijos, reajustar los enlaces de su antecesor.
3. Si el nodo tiene dos hijos, es necesario subir a la posiciǫn que este ocupa el dato más próximo de sus subárboles (el inmediatamente superior o el inmediatamente inferior) con el fin de mantener la estructura de ABB.

EJEMPLO 16.15. Borrar el elemento de clave 60 del siguiente árbol:



Se reemplaza `60` por el elemento mayor (`55`) en su subárbol izquierdo, o por el elemento más pequeño (`70`) en su subárbol derecho. Si se opta por reemplazar por el mayor del subárbol izquierdo, se mueve el `55` al raíz del subárbol y se reajusta el árbol.



Eficiencia de un ABB

La eficiencia de un ABB varía entre $O(n)$ y $log(n)$.
Sin embargo, si la mitad de los elementos insertados después de otro con clave *k* tiene claves menores y la otra mitad mayores que *k*, se obtiene un árbol equilibrado, en el cual las comparaciones para obtener un elemento son como máximo $log_2(n)$, para un árbol de n nodos.

Árboles AVL

Un árbol equilibrado o ***árbol AVL*** es un ABB en el que las alturas de los subárboles izquierdo y derecho de cualquier nodo *difieren como máximo en 1.*
La condición de equilibrio de cada nodo implica una restricción en las alturas de los subárboles. SI v_t es la raíz de cualquier subárbol de un árbol equilibrado y h la altura de la rama izquierda entonces la altura de la rama derecha puede tomar los valores: $h - 1,\ h,\ h + 1$.

Altura de un AVL

La altura de un árbol binario perfectamente equilibrado de *n* nodos es $log(n)$. Las operaciones que se aplican a los árboles AVL no requieren más del 44% del tiempo (en el peor caso) que si se aplican a un árbol perfectamente equilibrado.

Inserción en AVL

Una inserción de una nueva clave, o un borrado, puede destruir el criterio de equilibrio de varios nodos del árbol. Se debe recuperar la condición de equilibrio del árbol antes de dar por finalizada la operación para que el árbol siga siendo equilibrado.
En el proceso de inserción una reestructuración de los nodos implicados en la violación del *criterio de equilibrio* hace que se recupere el equilibrio en todo el árbol, no siendo necesario seguir analizando los nodos del *camino de búsqueda*.

Rotación simple

Esta resuelve la *violación de quilibrio* de un nodo *izquierda-izquierda*, simétrica a la *derecha-derecha*. Se cambia de dirección para la rotación (en doble no) y luego se bajan dos niveles desde el nodo en que se halla el "error".

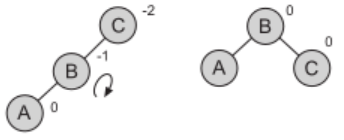


Figura 17.9. Árbol binario después de rotación simple `II`.

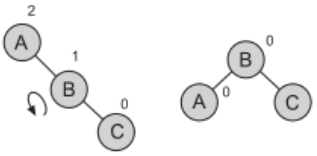


Figura 17.11. Árbol binario después de rotación simple DD.

Para la rotación II, los ajustes necesarios de los enlaces suponiendo *n* la referencia al nodo problema y *n1* la referencia al nodo de su rama izquierda (o derecha en ID):

$n \text{ -> } izq = n1 \text{ -> } der$
$n1 \text{ -> } der = n$
$n = n1$

$n \text{ -> } der = n1 \text{ -> } izq$
$n1 \text{ -> } izq = n$
$n = n1$

Una vez finalizada la rotación, los factores de equilibrio de los nodos intervinientes son siempre ***0***.

Rotación doble

La rotación doble resuelve los dos casos simétricos, se pueden denominar rotación *ID* y rotación *DI*. Equivale a dos rotaciones simples (una a la izquierda y otra a la derecha o al revés) pero sin cambiar la dirección.

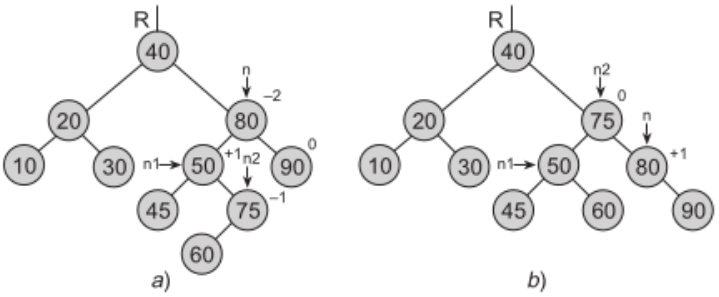


Figura 17.14. *a*) Árbol después de insertar clave 60. *b*) Rotación doble, izquierda derecha.

En rotación doble hay que mover los enlaces de tres nodos: el padre, el descendiente y el descendiente del descendiente por la rama contraria.
Los movimientos de los enlaces para las rotaciones *ID* y *DI*, respectivamente:

$n1 -> der = n2 -> izq$
$n2 -> izq = n1$
$n -> izq = n2 -> der$
$n2 -> der = n$
$n = n2$

$n1 -> izq = n2 -> der$
$n2 -> der = n1$
$n -> der = n2 -> izq$
$n2 -> izq = n$
$n = n2$