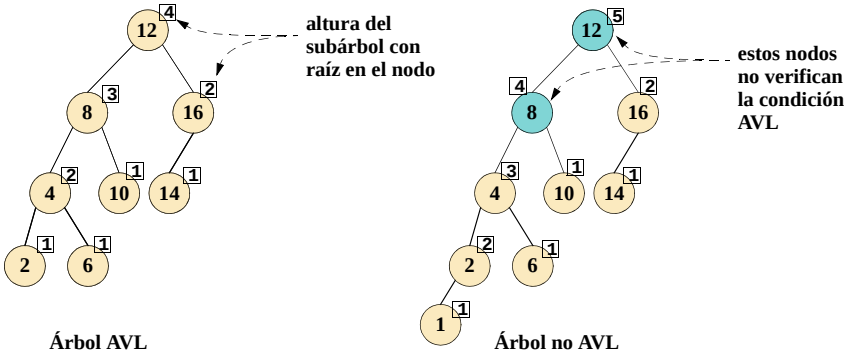
4.5 Árboles AVL (Adelson-Velskii y Landis)

Árbol binario de búsqueda que verifica que

- **las alturas de los subárboles derecho e izquierdo de cada nodo sólo pueden diferir, a lo sumo, en 1**
- **lo que *garantiza que la altura sea siempre O(log n)***

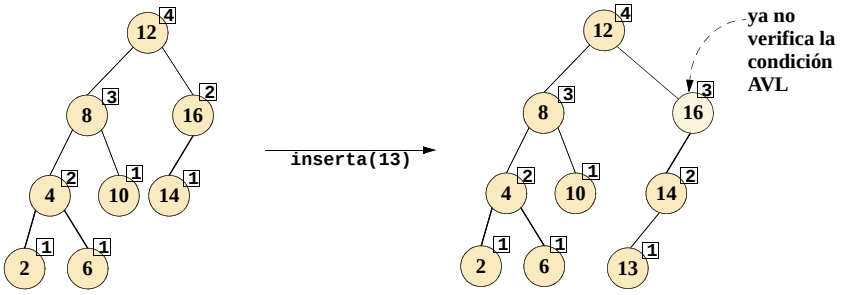


Inserción y extracción en árboles AVL

Se realizan igual que lo explicado para los árboles binarios de búsqueda "normales" (sección 4.4)

Pero una vez finalizada la inserción o extracción *puede ser necesario restablecer el equilibrio*

- **ya que uno o más de los nodos en el camino entre la raíz y el insertado o extraído podría dejar de verificar la condición AVL**

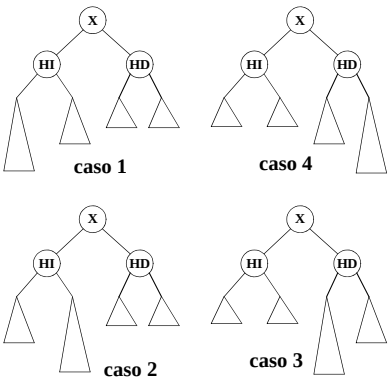


Tras una modificación se recorre el árbol desde el punto de inserción o extracción hacia la raíz

Sea X el nodo *más profundo* que incumple la propiedad AVL

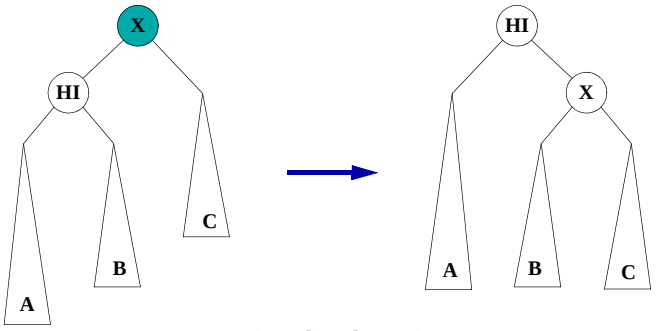
- **este nodo tiene a lo sumo dos hijos**
- **la diferencia entre las profundidades de los dos subárboles es 2**

Se dan 4 casos posibles



Operaciones de rotación: rotación simple izquierda de X

Se trata de ajustar la profundidad de dos ramas para hacer el árbol más equilibrado, manteniendo la relación de orden



(Se aplica al caso 1)

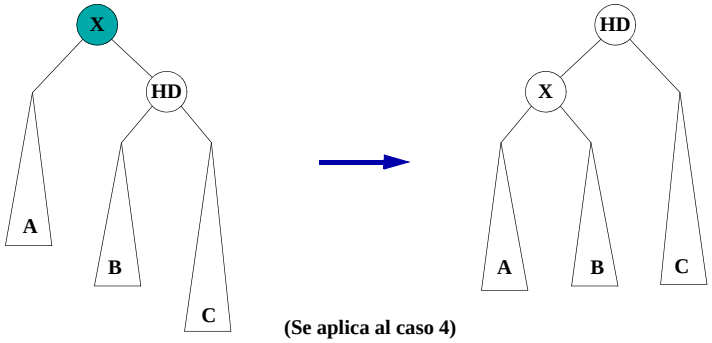
Pseudocódigo: rotación simple izquierda de X

```
método rotacionSimpleIzquierda (Nodo x) retorna Nodo
   Nodo hi:=x.hijoIzquierdo;
   x.hijoIzquierdo:=hi.hijoDerecho;
   x.hijoIzquierdo.padre=x;
   hi.hijoDerecho:=x;
   x.padre=hi;
   retorna hi;
fmétodo
```

Luego es preciso reinsertar la rama retornada en el lugar donde estaba en el árbol

Operaciones de rotación: rotación simple derecha de X

Es como la anterior, pero elevando la rama derecha



(Se aplica al caso 4)

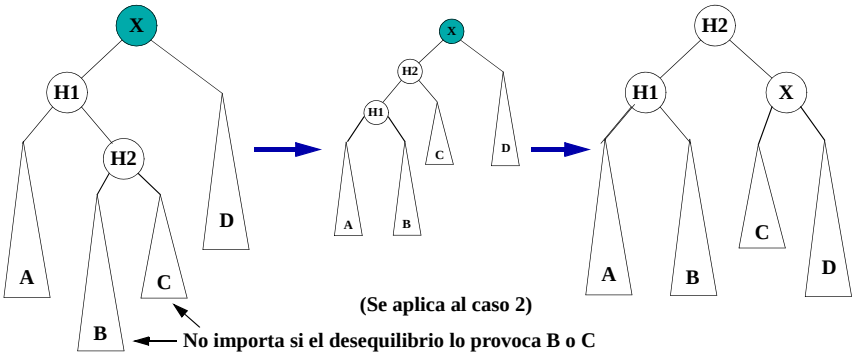
Pseudocódigo: rotación simple derecha de X

```
método rotacionSimpleDerecha (Nodo x) retorna Nodo
   Nodo hd:=x.hijoDerecho;
   x.hijoDerecho:=hd.hijoIzquierdo;
   x.hijoDerecho.padre=x;
   hd.hijoIzquierdo:=x;
   x.padre=hd;
   retorna hd;
fmétodo
```

Luego es preciso reinsertar la rama retornada en el lugar donde estaba en el árbol

Operaciones de rotación: rotación doble derecha-izquierda de X

Cuando la parte desequilibrada es una rama central, es preciso realizar dos rotaciones: entre H1 y H2 y entre X y H2



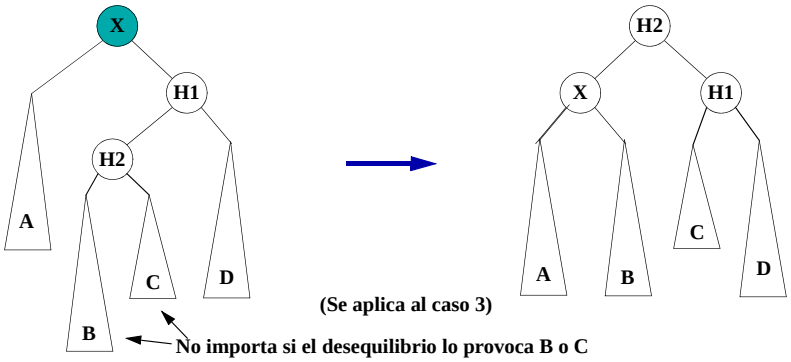
Pseudocódigo: rotación doble derecha-izquierda

```
método rotacionDobleDI (Nodo x) retorna Nodo
   x.hijoIzquierdo=
      rotacionSimpleDerecha(x.hijoIzquierdo);
   x.hijoIzquierdo.padre=x;
   retorna rotacionSimpleIzquierda(x);
fmétodo
```

Luego es preciso reinsertar la rama retornada en el lugar donde estaba en el árbol

Operaciones de rotación: rotación doble izquierda-derecha de X

Cuando la parte desequilibrada es una rama central, es preciso realizar dos rotaciones: entre H1 y H2 y entre X y H2



(Se aplica al caso 3)

No importa si el desequilibrio lo provoca B o C

Pseudocódigo: rotación doble izquierda-derecha

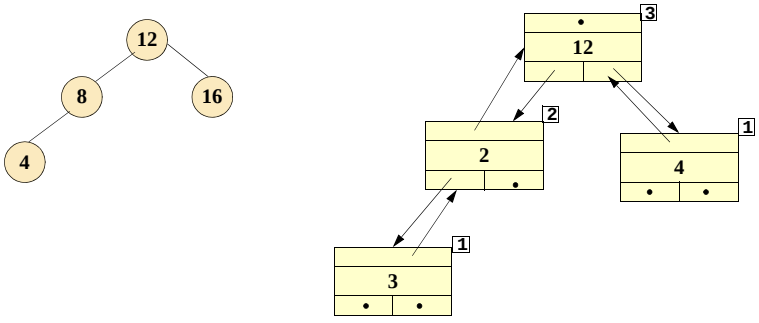
```
método rotacionDobleID (Nodo x) retorna Nodo
   x.hijoDerecho=
      rotacionSimpleIzquierda(x.hijoDerecho);
   x.hijoderecho.padre=x;
   retorna rotacionSimpleDerecha(x);
fmétodo
```

Luego es preciso reinsertar la rama retornada en el lugar donde estaba en el árbol

Árboles AVL: implementación

Se implementan como cualquier otro árbol binario

- **punteros a padre e hijos (o sólo a los hijos)**
- **añadiendo, además, a cada nodo un campo que indica su altura**



Árboles AVL: eficiencia de las operaciones

Comparación de la eficiencia de las operaciones de los árboles AVL con las de un árbol binario de búsqueda no equilibrado:

Operación	AVL (caso peor y promedio)	Árbol no equilibrado (caso promedio)	Árbol no equilibrado (caso peor: altura=n)
inserta	O(log n)	O(log n)	O(n)
busca	O(log n)	O(log n)	O(n)
elimina	O(log n)	O(log n)	O(n)

- **Donde n es el número de nodos**
- **La altura de un árbol AVL es siempre *O*(log n)**
- **En el peor caso (para entradas ordenadas) la altura de un árbol no equilibrado puede ser *O*(n)**

4.6 Árboles Rojinegros

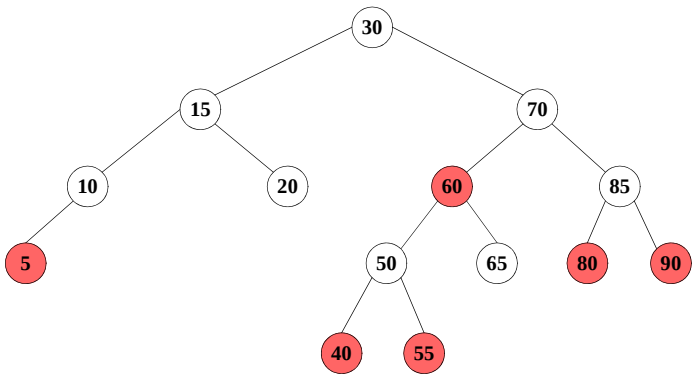
Un árbol rojinegro es un árbol binario de búsqueda que verifica las siguientes propiedades

- **Cada nodo está coloreado de rojo o negro**
- **La raíz es negra**
- **Si un nodo es rojo, sus hijos deben ser negros**
- **Todos los caminos desde un nodo cualquiera a una referencia nula deben tener el mismo número de nodos negros**

Evita el doble recorrido del árbol AVL

Estudiaremos ahora las operaciones de inserción y eliminación

Ejemplo de árbol rojinegro



Inserción en un árbol rojinegro

Insertamos un nuevo elemento como una hoja

Para saber dónde insertar hacemos un recorrido descendente como en la inserción no equilibrada

Durante este recorrido, si encontramos un nodo X con dos hijos rojos

- **convertimos X en rojo y sus dos hijos en negro**
  - **el número de nodos negros en un camino no varía**
  - **pero tendremos dos nodos rojos consecutivos, si el padre de X es rojo**
- **si el padre de X es rojo, hacemos una rotación simple o doble del padre de X, cambiando los colores de forma apropiada**
- **Este proceso se repite hasta alcanzar una hoja**

Inserción en un árbol rojinegro (cont.)

Pondremos el nuevo nodo rojo

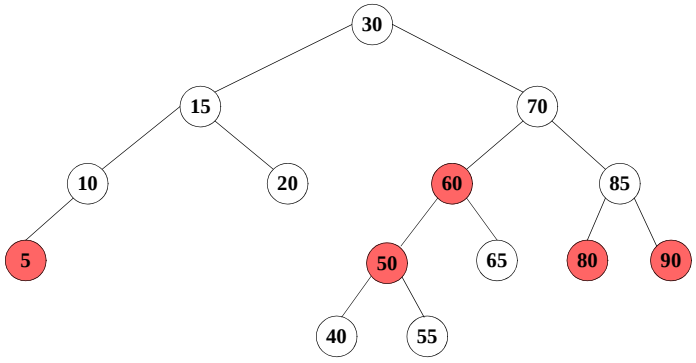
- **si le ponemos negro, incumplimos la última regla**

Por el procedimiento anterior el padre es negro y no hay que hacer nada más

- **p.e., insertar el número 45 en el árbol de la figura anterior**

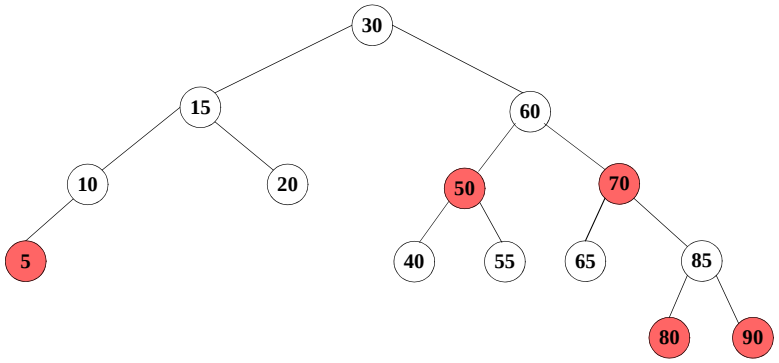
Ejemplo de inserción en un árbol rojinegro (paso 1)

En el recorrido encontramos el nodo 50 con dos hijos rojos, que cambiamos a negros, cambiando luego el 50 a rojo



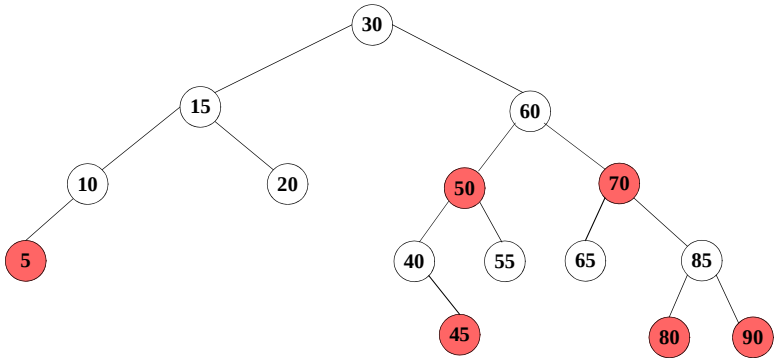
Ejemplo de inserción en un árbol rojinegro (paso 2)

Hacemos una rotación simple de 60 y 70, y reajustamos los colores para recuperar las propiedades



Ejemplo de inserción en un árbol rojinegro (paso 3)

No es preciso hacer más cambios de color ni rotaciones, por lo que finalizamos insertando el nodo 45, de color rojo



Eliminación en árboles rojinegros

Al igual que en el algoritmo de eliminación en árboles no equilibrados

- **sólo borramos nodos que son hojas o sólo tienen un hijo**
- **si tiene dos hijos, se reemplaza su contenido, y se borra otro nodo que es una hoja o sólo tiene un hijo**

Si el nodo que vamos a borrar es rojo no hay problema

Si el nodo a borrar es negro, la eliminación hará que se incumpla la 4ª propiedad

- **la solución es transformar el árbol para asegurarnos de que siempre borramos un nodo rojo**

Eliminación en árboles rojinegros (cont.)

Haremos un recorrido descendente del árbol, comenzando por la raíz; llamaremos

- X al nodo actual
- T a su hermano
- P a su padre

Intentaremos que X sea siempre rojo, y mantener las propiedades del árbol a cada paso

- por tanto, al descender, el nuevo P será siempre rojo, y el nuevo X y el nuevo T serán negros

Nodos "centinela"

Para reducir los casos especiales que complicarían el algoritmo, supondremos que existen unos nodos extra o "centinelas", en lugar de los punteros nulos que haya en el árbol

- uno está por encima de la raíz
- si a un nodo le falta un hijo, supondremos un nodo centinela en su lugar
- si un nodo es una hoja, tendrá dos centinelas en lugar de sus hijos

Supondremos los centinelas negros inicialmente

El algoritmo comienza haciendo que X sea el centinela por encima de la raíz, y coloreándolo de rojo

Eliminación en árboles rojinegros (cont.)

Existen dos casos principales, además de sus variantes simétricas

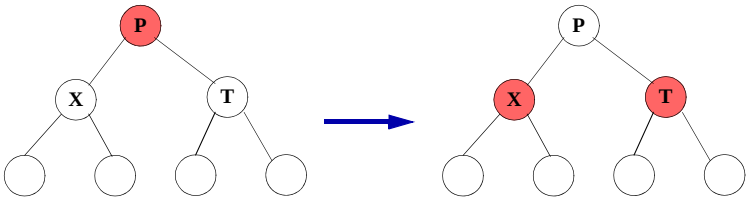
- *caso a*: X tiene dos hijos negros
  - *subcaso a1*: T tiene dos hijos negros
  - *subcaso a2*: T tiene un hijo exterior rojo
  - *subcaso a3*: T tiene un hijo interior rojo
  - *subcaso a4*: T tiene dos hijos rojos: lo resolvemos como *a2*
- *caso b*: alguno de los hijos de X es rojo

Nota

- un hijo es exterior si es un hijo derecho de un hijo derecho o un hijo izquierdo de un hijo izquierdo
- es interior en los otros dos casos

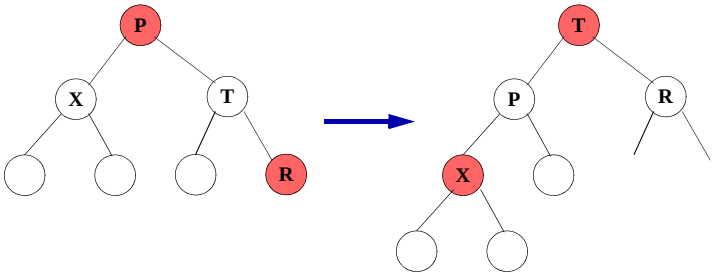
Eliminación en árboles rojinegros: subcaso a1

Hacemos un cambio de color



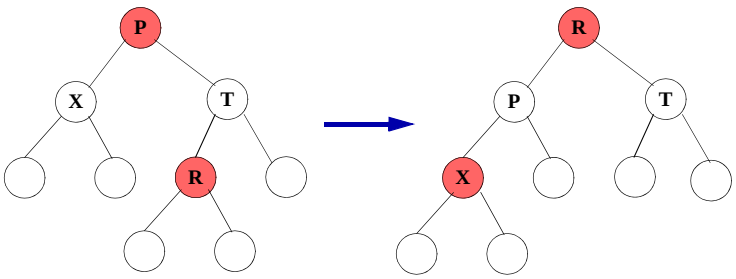
Eliminación en árboles rojinegros: subcasos a2 y a4

Hacemos una rotación simple entre P y T, y los cambios de color que se indican



Eliminación en árboles rojinegros: subcaso a3

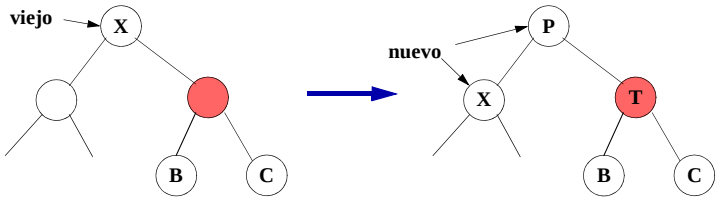
Hacemos una rotación doble entre T y R y luego entre P y R, y los cambios de color que se indican



Eliminación en árboles rojinegros: caso b

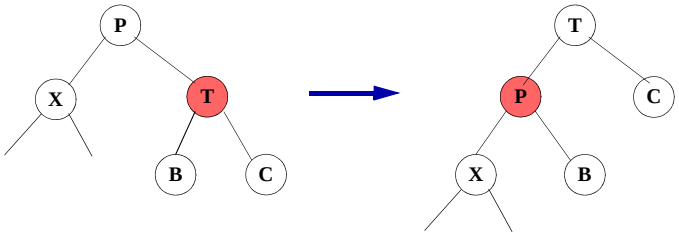
Descendemos al siguiente nivel del árbol obteniendo nuevo X, T, P

- **Si el siguiente nodo en el descenso del árbol es rojo, continuamos por él sin necesidad de más cambios**
- **En caso contrario estamos en esta situación**



Eliminación en árboles rojinegros: caso b (cont.)

Hacemos una rotación entre T y P



Y ahora repetimos el algoritmo para tratar de hacer X rojo

Eliminación en árboles rojinegros: caso b (cont.)

El algoritmo siempre finaliza, ya que está garantizado que al llegar al nodo a eliminar habremos alcanzado uno de estos dos casos

- **X es una hoja, que se considera que tiene dos hijos negros (caso *a*)**
- **X tiene un solo hijo**
  - **si es negro, su "hermano" es un nodo centinela negro, y se aplica el caso *a***
  - **si es rojo, eliminamos X y coloreamos ese hijo de negro**

4.7 Montículo binario

Estructura de datos que nos permite gestionar un conjunto de elementos entre los que existe una relación de orden total

- **con operaciones de inserción y extracción eficientes (*O*(log *n*))**
- **permite conocer el menor elemento en tiempo constante (*O*(1))**

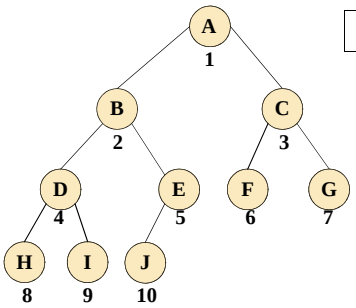
Es un árbol binario:

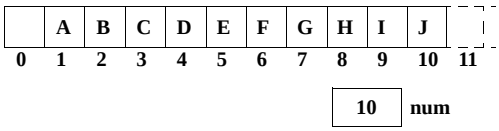
- **implementable muy eficientemente mediante un array**
- **más simple que un árbol binario de búsqueda equilibrado (AVL o rojinegro)**

Definición: árbol binario completo

Árbol binario en el que *todos sus niveles están completos* salvo quizás el nivel inferior que se rellena de izquierda a derecha

- **su profundidad es *O*(log *n*)**
- **Se puede implementar mediante un array y un contador con el número de nodos**





10 num

La altura máxima del árbol depende del tamaño del array:

$$\text{altura}_{max} = \lceil \log_2(\text{tamArray}) \rceil$$

En esta estructura, para el elemento `i`:

- **El hijo izquierdo está en `2*i`**
- **El hijo derecho en `2*i+1`**
  - **si el valor sobrepasa el número de nodos (`num`), ello indica que ese hijo no existe**
- **El padre está en `i div 2`**

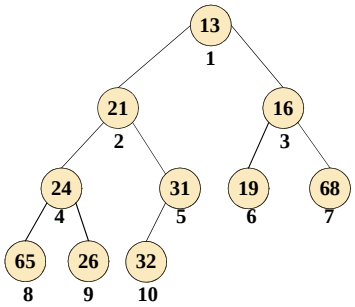
En la posición `0` situaremos un nodo "centinela"

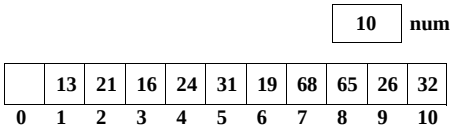
- **un padre falso para la raíz**
- **facilita la implementación**

Montículo binario

Un montículo binario es un árbol binario completo en que:

- **el padre siempre es menor que sus hijos**
- **el menor elemento siempre es la raíz**
  - **buscar el mínimo es siempre *O*(1)**





Inserción en montículos binarios

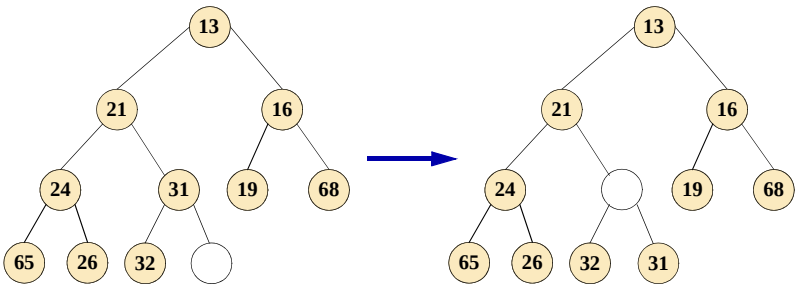
Al añadir un nodo al árbol debemos hacerlo en el siguiente hueco disponible

En caso de que el elemento no quedase bien ordenado, debemos desplazar el hueco

- **lo hacemos intercambiándolo con el padre del hueco**
- **repetimos este paso sucesivas veces hasta alcanzar la relación de orden**
- **a este proceso lo llamamos *reflotamiento***

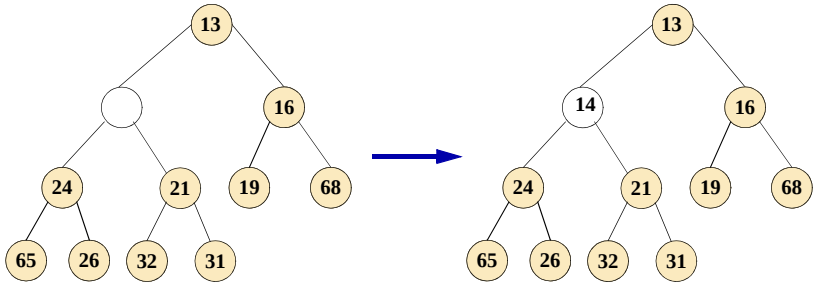
Ejemplo de inserción (paso 1)

Queremos meter el elemento 14; reflotamos el hueco



Ejemplo de inserción (pasos 2 y 3)

Queremos meter el elemento 14; reflotamos el hueco



En el peor caso se requieren $\texttt{altura}_{\texttt{max}}$ comparaciones para insertar ($O(\texttt{altura}_{\texttt{max}})$)

Eliminación en montículos binarios

Queremos eliminar el elemento mínimo, es decir, la raíz

- **allí queda un hueco**

Hay que recolocar el elemento `X` que ocupa la última posición

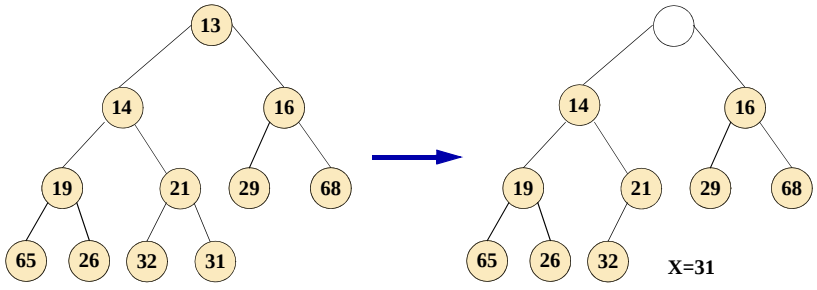
- **se obtiene en $O(1)$ utilizando el contador `num`**

Hay que trasladar el hueco hasta una posición adecuada para `X`

- **elegimos el hijo más pequeño del hueco y lo movemos hacia arriba**
- **repetimos este proceso hasta encontrar una posición apropiada para `X`**
- **llamamos a este proceso *hundimiento***

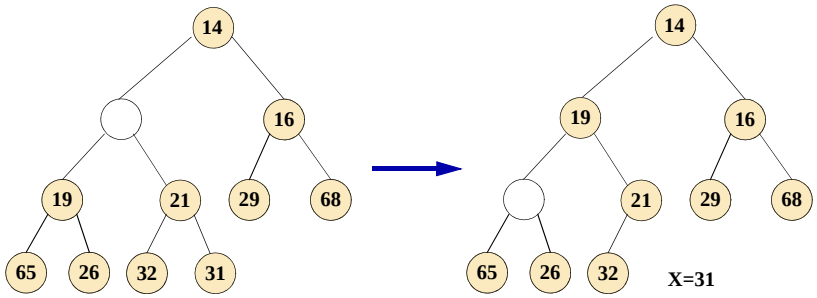
Ejemplo de eliminación (paso 1)

Queremos eliminar la raíz, y recolocar `X=31`



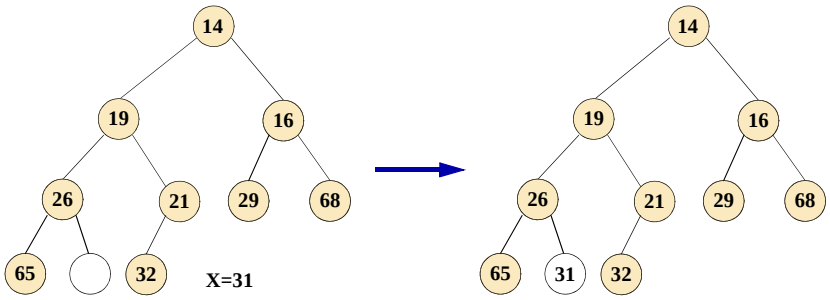
Ejemplo de eliminación (pasos 2 y 3)

Hundimos el hueco



Ejemplo de eliminación (pasos 4 y 5)

Seguimos hundiendo el hueco y colocamos X



En el peor caso se requiere hundir el hueco $\texttt{altura}_{\texttt{max}}$ niveles ($O(\texttt{altura}_{\texttt{max}})$)

Montículo binario: eficiencia de las operaciones

Operación	Ritmo de crecimiento
obtienePrimero	O(1)
inserta	O(log n)
eliminaPrimero	O(log n)
busca	O(n)
elimina	O(n)[1]

- **Donde n es el número de nodos**

[1] **Hay que encontrar el elemento ($O(n)$) antes de eliminarle**