Grado en Ingeniería Información

Estructura de Datos y Algoritmos

Sesión 11

Curso 2022-2023

Marta N. Gómez

UNIVERSIDAD NEBRIJA

T3. Tipos Abstractos de Datos (TAD)

- **Árboles.**
  - Conceptos generales
  - Realización del TAD Árbol Binario
  - Recorridos de Árboles Binarios
  - Árboles Binarios de Búsqueda (ABB)
  - Árboles Equilibrados (AVL)
  - **Montículos**

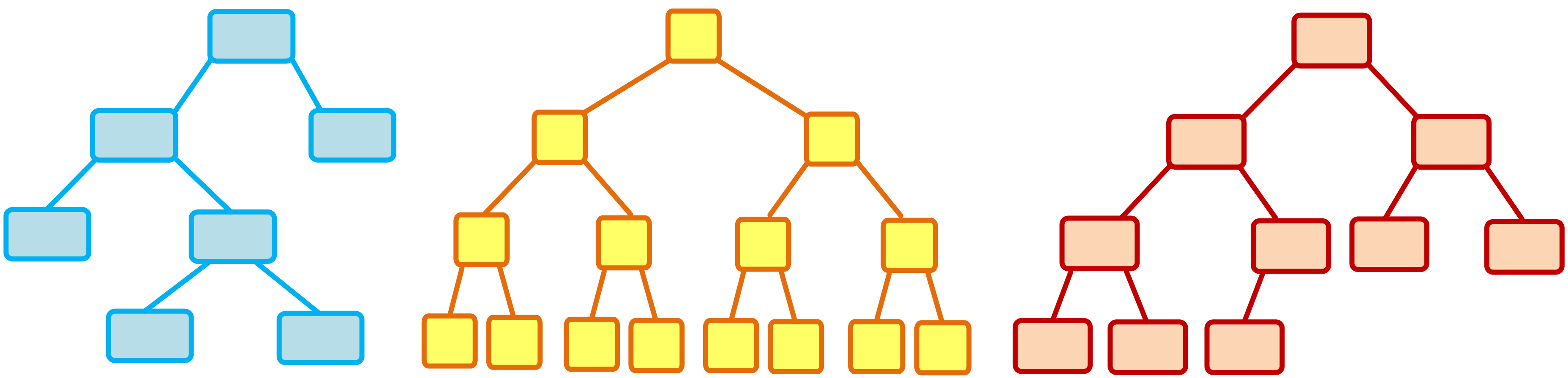
ÍNDICE

TIPOS DE ÁRBOLES BINARIOS

Árbol Estricto es un árbol binario donde cada nodo puede tener **0** o **2 hijos**.

Árbol Lleno es un árbol binario estricto donde la **altura del subárbol izquierdo** es **igual** que la del **subárbol derecho**, y además, **ambos subárboles son llenos**.

Árbol Completo es un árbol binario lleno hasta el penúltimo nivel. En el **último nivel** los nodos están agrupados a la **izquierda**.



Árbol Estricto

Árbol Lleno

Árbol Completo

TIPOS DE ÁRBOLES BINARIOS

Los **árboles llenos** son:

- Los árboles con **máximo número de nodos** (***n***) **para una altura** (***h***) **dada**. Se cumple que: $n = 2^h - 1$
- El **número de nodos de un árbol lleno** sólo puede ser una **potencia de dos menos uno**: 1, 3, 7, 15, 31, …

Los **árboles completos** pueden almacenar cualquier número de nodos y se cumple que:

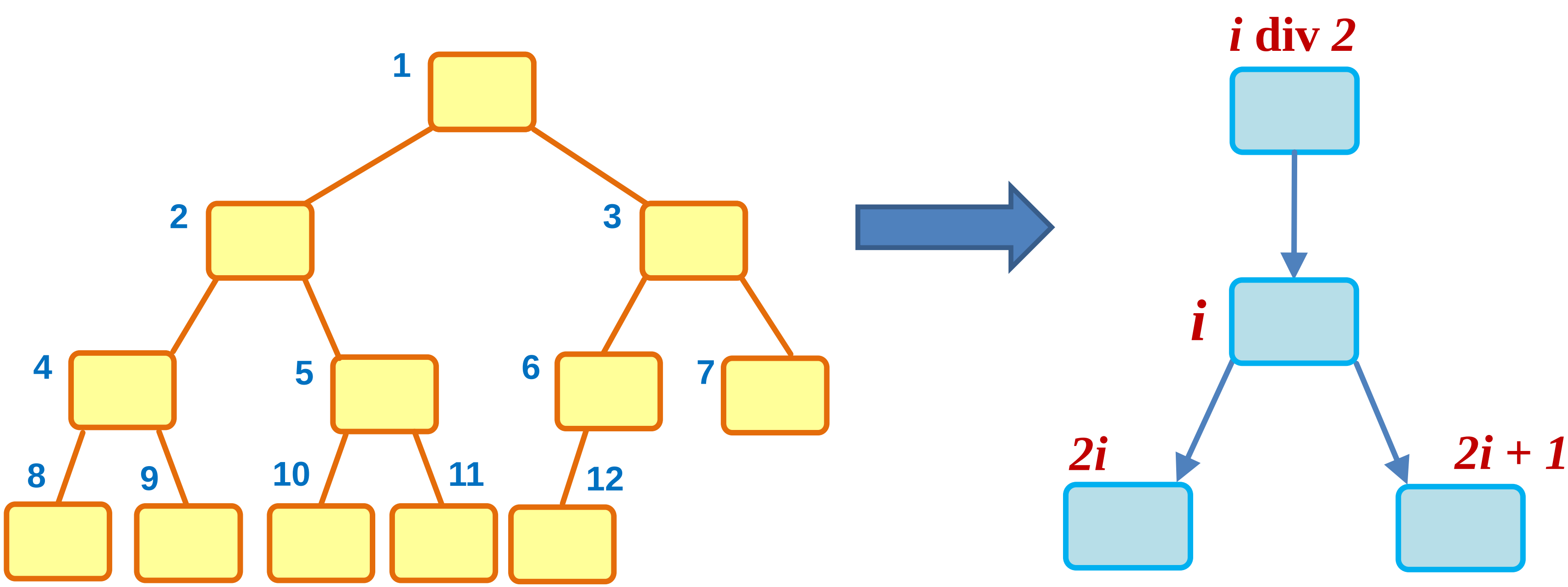
- **Su altura es proporcional al logaritmo del número de nodos**:

$$h \in O(\log n)$$

- Cuando se conoce el **recorrido en anchura/niveles** del árbol es posible **reconstruirlo**.

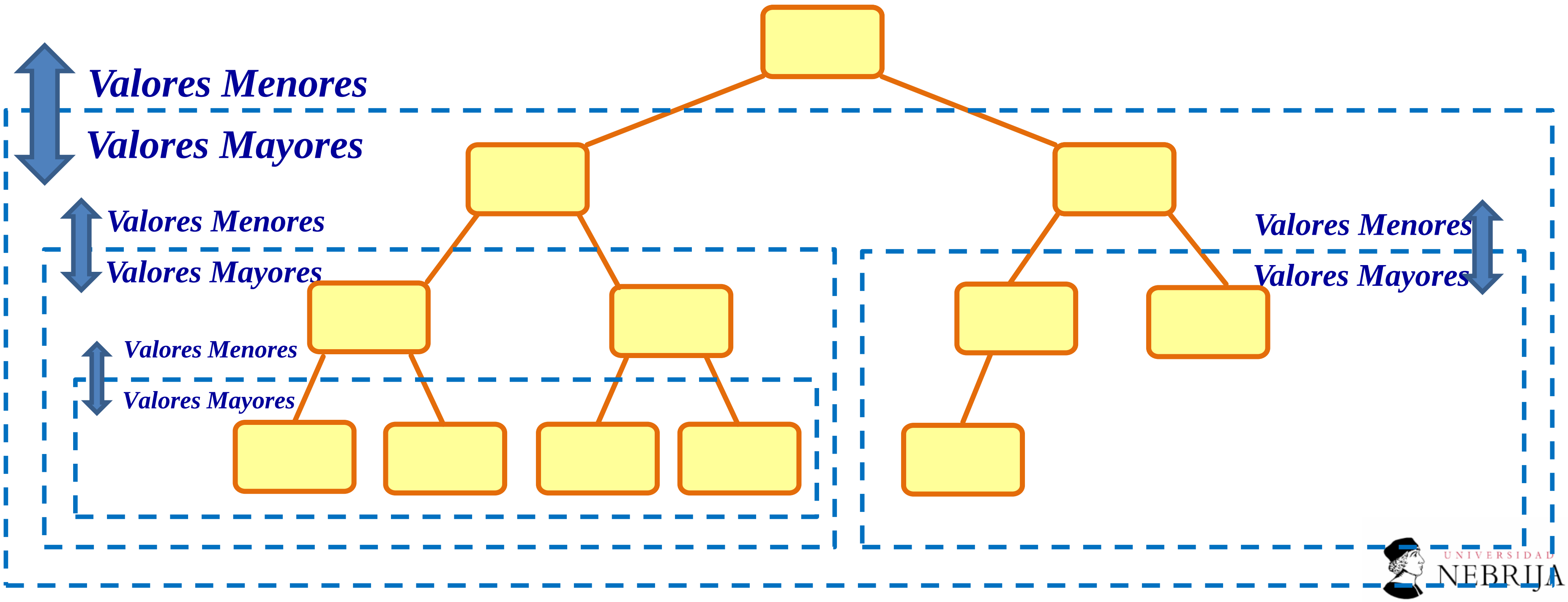
Si se almacena el contenido de un **árbol completo** en un **vector en el orden dado por su recorrido por anchura/niveles**, conocido el **índice/posición** de un elemento ($i > 0$) en el vector se puede determinar el **índice** de su **nodo padre** y los de sus **nodos hijos**:

- Nodo padre: ***i div 2***
- Nodo hijo-izq: ***2i*** Nodo hijo-dcho: ***2i +1***

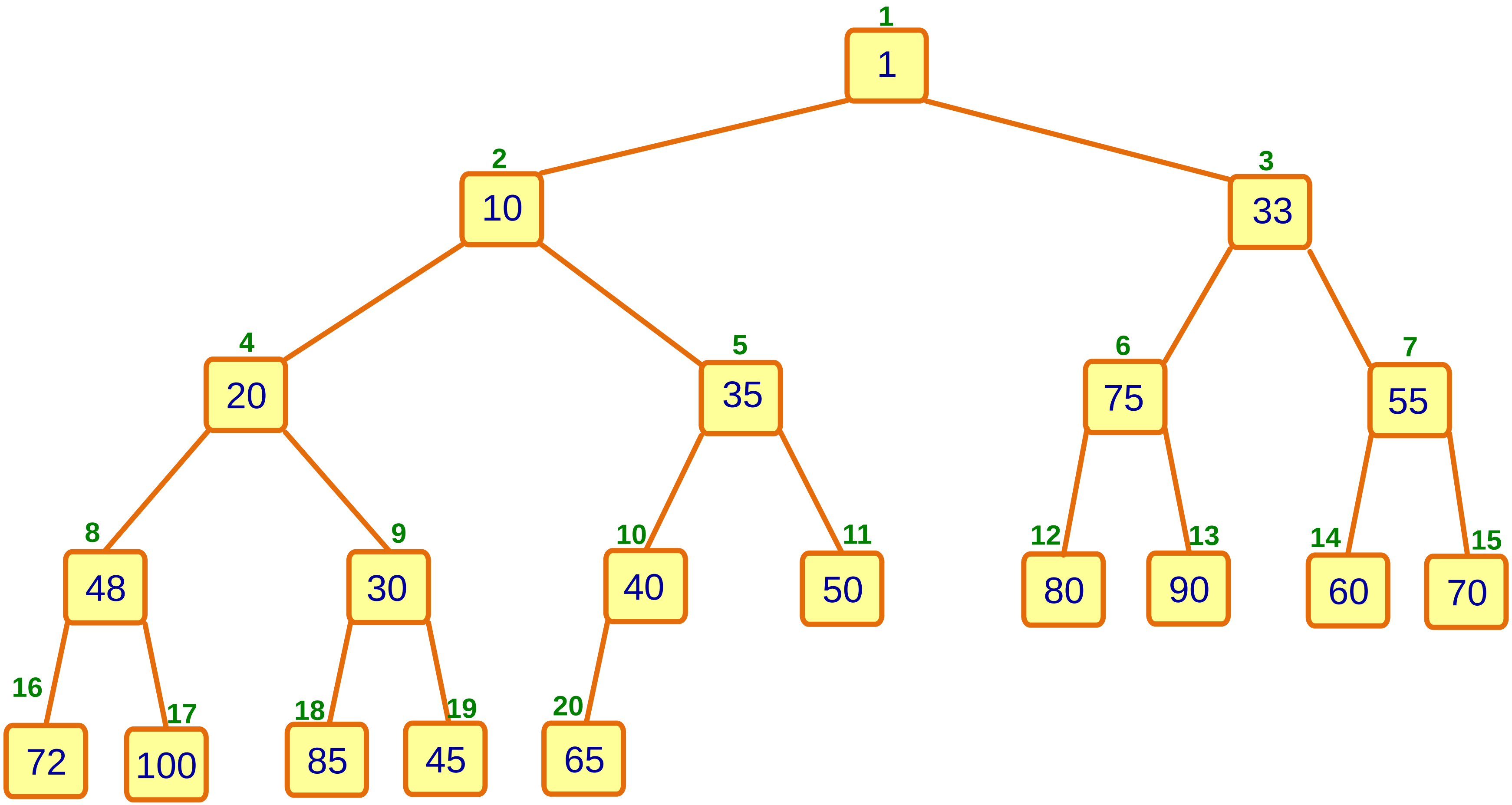


Un **montículo (heap)** es un **árbol completo** cuyos nodos almacenan elementos **comparables** mediante la relación $\leq$ y donde todo nodo cumple la ***propiedad de montículo***:

- **Propiedad de montículo:** Todo nodo es menor que sus descendientes (montículo de **mínimos**).



MONTÍCULOS BINARIOS



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	**10**	**33**	**20**	**35**	**75**	**55**	**48**	**30**	**40**	**50**	**80**	**90**	**60**	**70**	**72**	**100**	**85**	**40**	**65**

MONTÍCULOS BINARIOS - PROPIEDADES

- El **nodo raíz** (en *primera posición del vector*) es el **mínimo**.
- La **altura de un montículo** es ***logarítmica*** respecto al número de elementos almacenados (por ser ***árbol completo***).
- Si un sólo elemento **no cumple** la propiedad de montículo, se puede restablecer la propiedad mediante **ascensos sucesivos** en el árbol (intercambios con su padre) o mediante **descensos** en el árbol (intercambios con el mayor de sus hijos). El número de operaciones es **proporcional** a la **altura**.
- **Insertar un nuevo elemento:** siempre se **inserta al final** del vector (última hoja del árbol) y se **asciende hasta que cumpla la propiedad**.
- **Eliminar la raíz:** se **intercambia con el último elemento** (se elimina en O(1)) y se **desciende la nueva raíz hasta que cumpla la propiedad.**

Un **montículo** es una representación extremadamente útil para **el TAD Cola de Prioridad**:

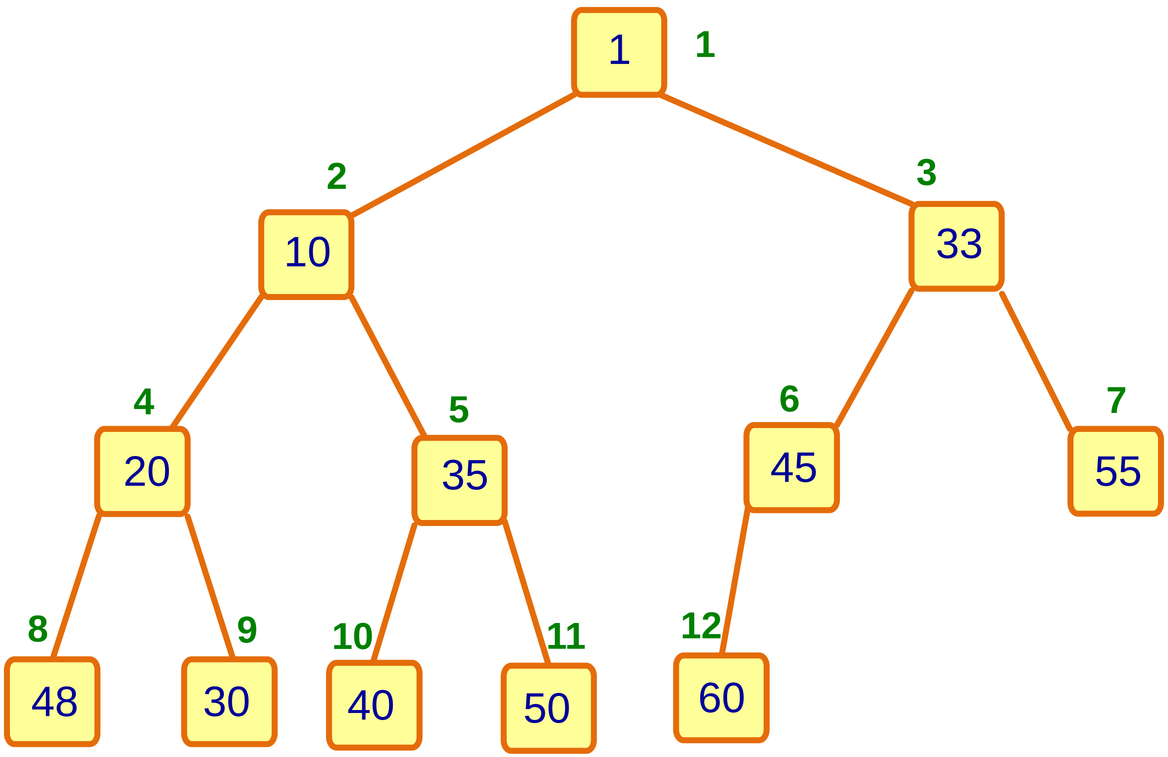
- El acceso al mínimo es ***O(1)***.
- La inserción por valor es ***O(log n)***.
- El borrado del mínimo es ***O(log n)***.
- Utiliza un **vector** en lugar de una representación **con punteros**.
- La creación partiendo de un vector es ***O(n)*** y no necesita espacio adicional.
- El borrado o modificación de un elemento, desde su posición en el montículo, es ***O(log n)***.

Operaciones para las que no tiene un buen comportamiento:

- Para la búsqueda y acceso al i-ésimo menor se comporta igual que un vector desordenado.
- La fusión de montículos (binarios) es ***O(n)***.

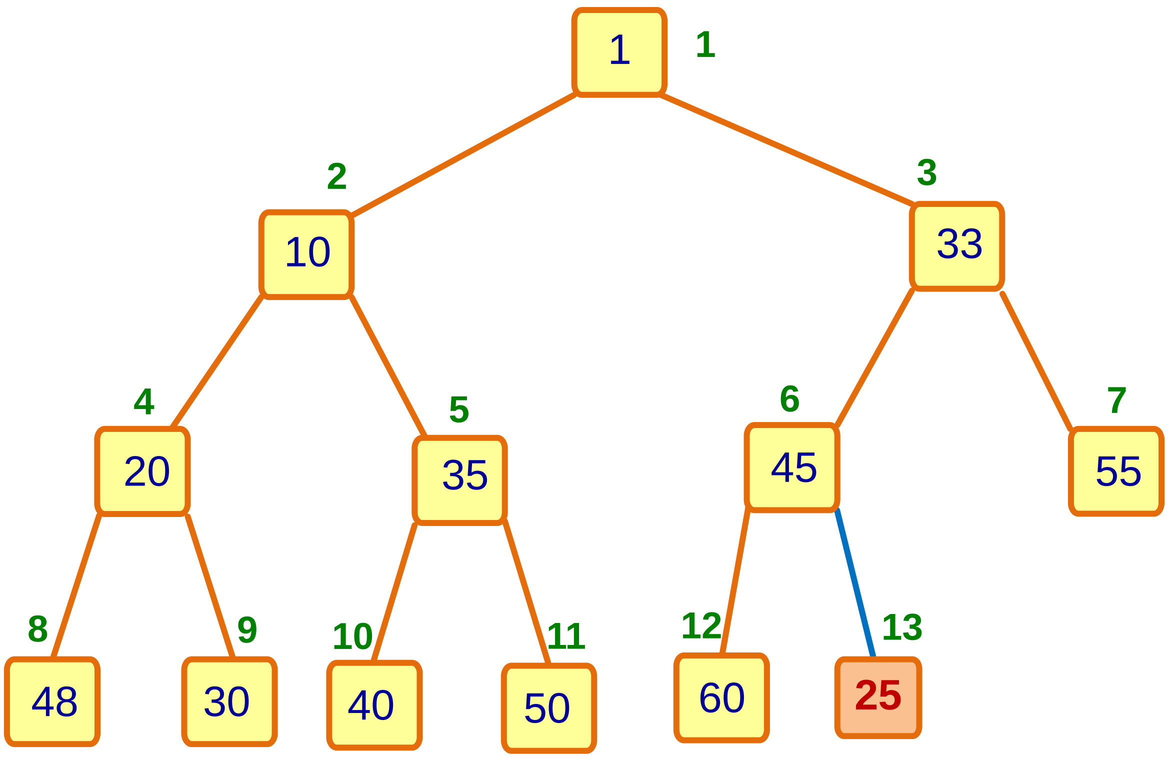
UNIVERSIDAD NEBRIJA

MONTÍCULOS BINARIOS - Inserción



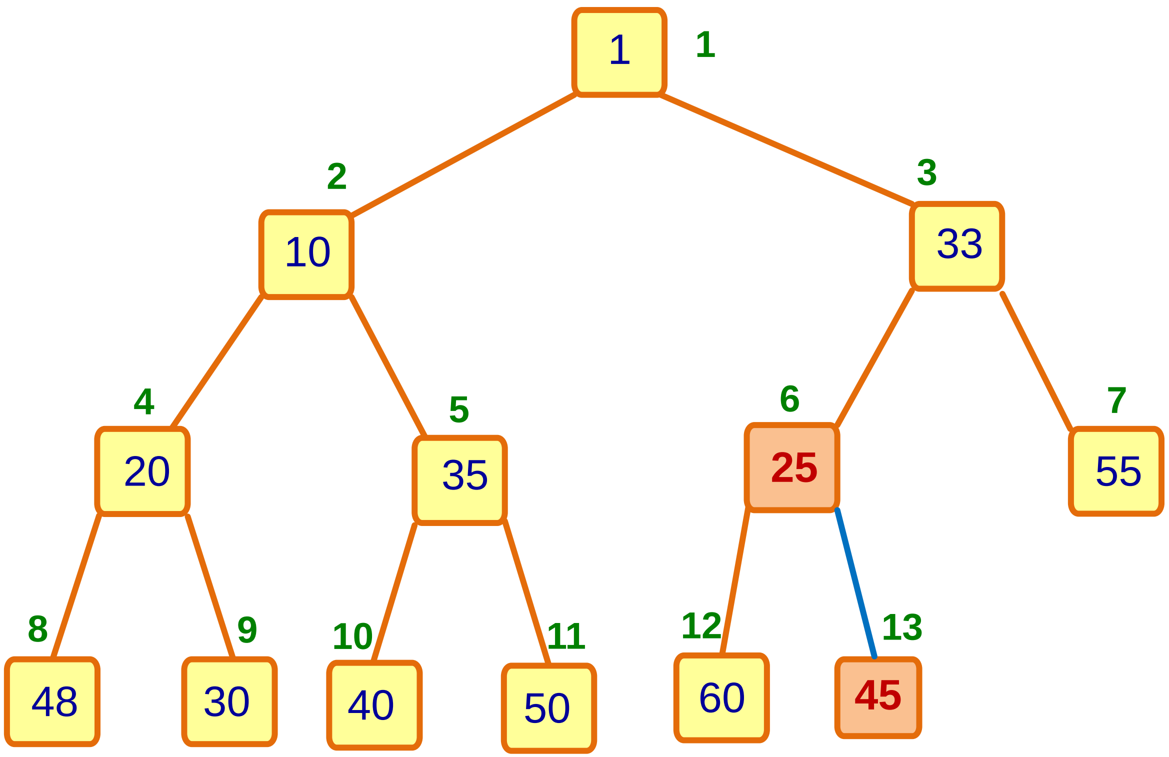
1	2	3	4	5	6	7	8	9	10	11	12	13
1	**10**	**33**	**20**	**35**	**45**	**55**	**48**	**30**	**40**	**50**	**60**	

MONTÍCULOS BINARIOS - Inserción



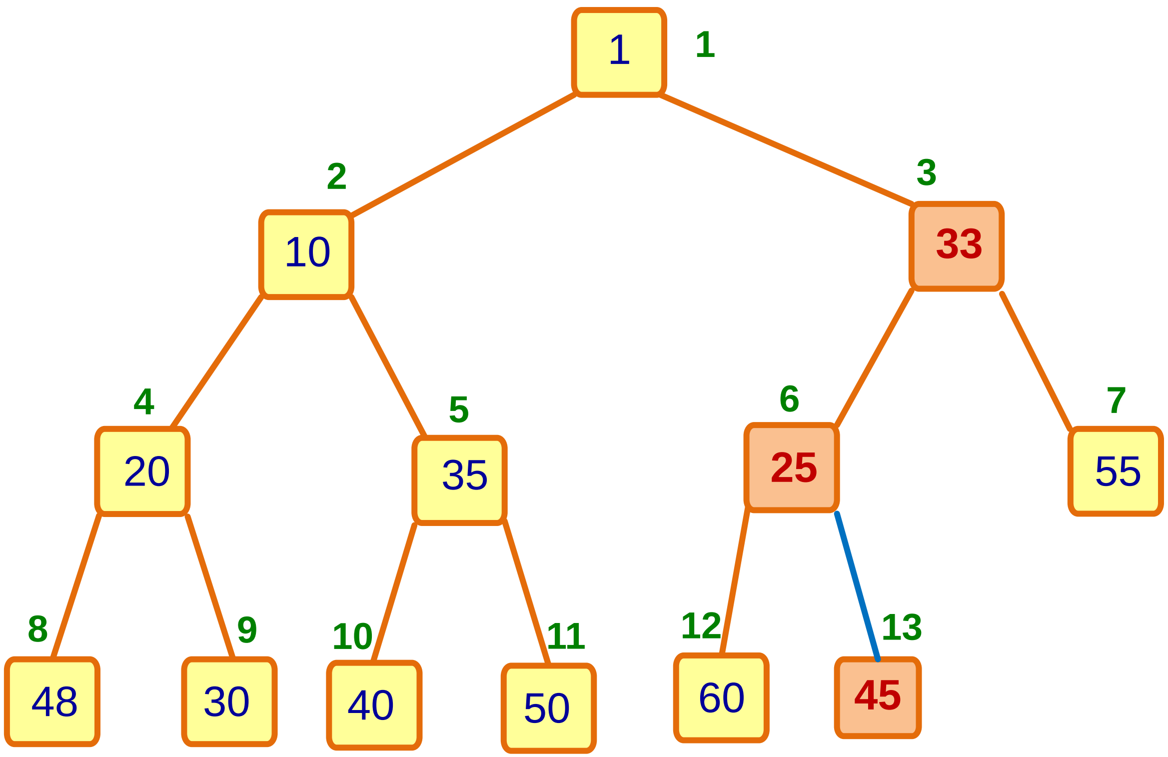
1	2	3	4	5	6	7	8	9	10	11	12	13
1	**10**	**33**	**20**	**35**	**45**	**55**	**48**	**30**	**40**	**50**	**60**	**25**

MONTÍCULOS BINARIOS - Inserción



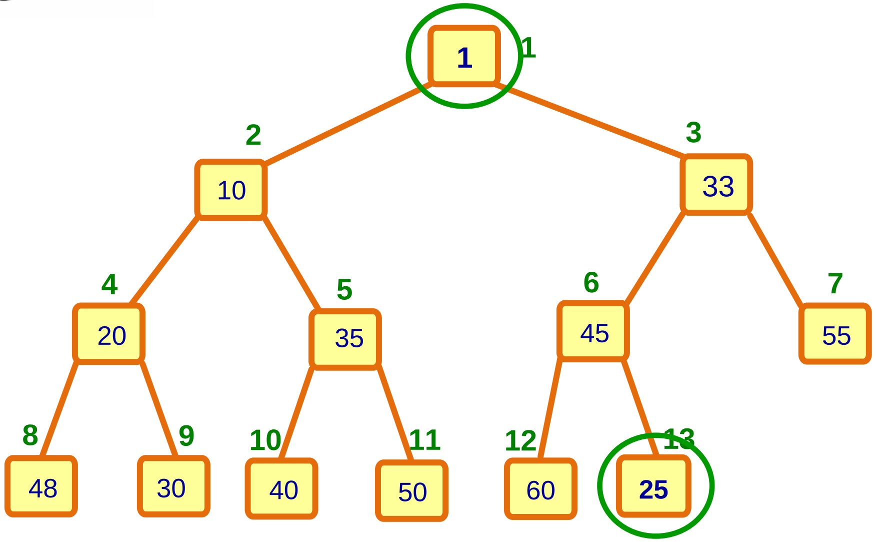
1	2	3	4	5	6	7	8	9	10	11	12	13
1	**10**	**33**	**20**	**35**	**25**	**55**	**48**	**30**	**40**	**50**	**60**	**45**

MONTÍCULOS BINARIOS - Inserción

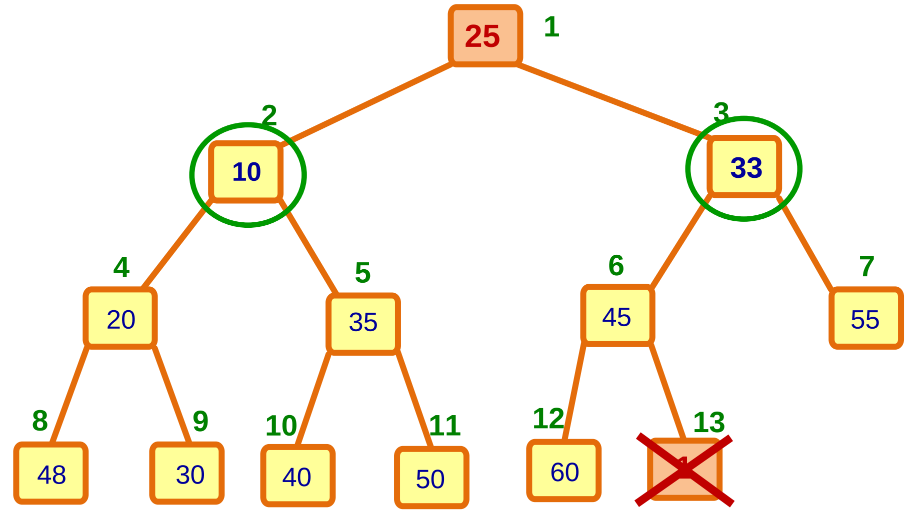


1	2	3	4	5	6	7	8	9	10	11	12	13
1	**10**	**25**	**20**	**35**	**33**	**55**	**48**	**30**	**40**	**50**	**60**	**45**

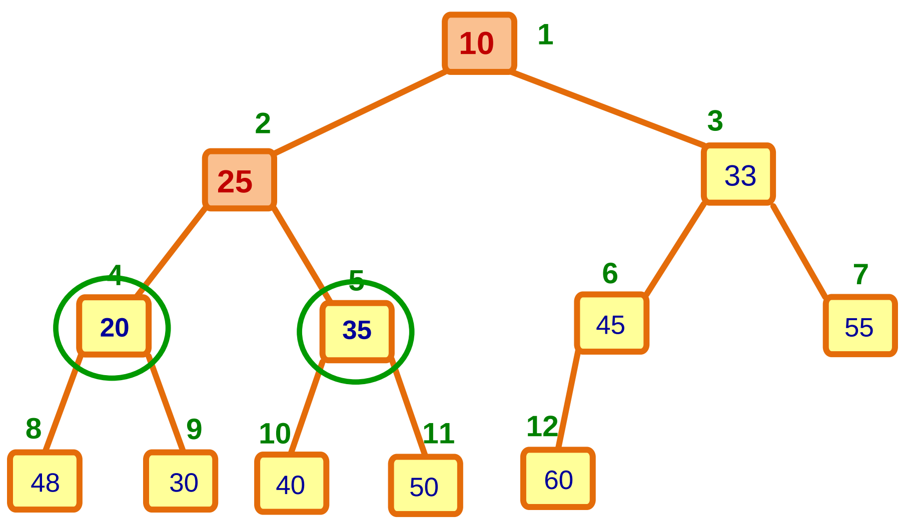
MONTÍCULOS BINARIOS – Borrado del mínimo



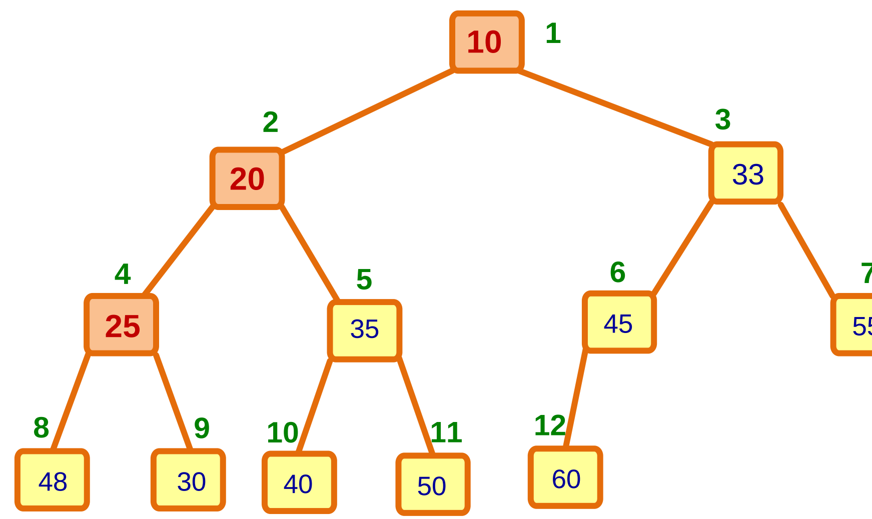
1	2	3	4	5	6	7	8	9	10	11	12	13
1	10	33	20	35	45	55	48	30	40	50	60	25



1	2	3	4	5	6	7	8	9	10	11	12	13
25	10	33	20	35	75	55	45	30	40	50	60	1



1	2	3	4	5	6	7	8	9	10	11	12	13
10	25	33	20	35	75	55	45	30	40	50	60	



1	2	3	4	5	6	7	8	9	10	11	12	13
10	20	33	25	35	75	55	45	30	40	50	60	

```
class colaPrioridad {
public:
    colaPrioridad();
    int size() const;
    bool empty() const;
    const TipoDato &top() const;
    void push(const TipoDato &dato);
    void pop();
private:
    void hundir(int i);
    void flotar(int i);
    int getPadre(int indice) const;
    int getLeft (int indice) const;
    int getRight(int indice) const;
private:
    vector<TipoDato> cp;
};
```

```
colaPrioridad::colaPrioridad() {}

int colaPrioridad::size() const {
        return cp.size();
}

bool colaPrioridad::empty() const {
        return cp.empty();
}
```

El primer elemento es el de **mayor prioridad**, coincide con la **raíz del montículo.**

```
const TipoDato &colaPrioridad::top() const {
        return cp.at(0);
}
```

La primera posición del vector está en el índice 0.

```
int colaPrioridad::getPadre(int indice) const {
        return (indice-1) / 2;
}
int colaPrioridad::getLeft(int indice) const {
        return (indice*2 + 1);
}
int colaPrioridad::getRight(int indice) const {
        return (indice*2 + 2);
}
```

Inserción de un elemento


```
void colaPrioridad::push(const TipoDato &dato)  {
    // Insertamos el nuevo elemento al final del vector
    cp.push_back(dato);

    // Se obtiene el índice del elemento y se llama a flotar
    int indice = this->size()  – 1;
    flotar(indice);
}
```

Inserción de un elemento


```
void colaPrioridad::flotar (int i) {

    if (i != 0) {
        // se comprueba la propiedad del montículo
        int pos = getPadre(i);
        if (cp.at(pos)  > cp.at(i))  {
            swap(cp[i], cp[pos]);

            // llamada recursiva con el padre del nuevo nodo
            flotar(pos);
        }
    }
}
```

Eliminar la raíz del montículo (elemento menor)

```
void colaPrioridad::pop()  {
    if (!this->empty())  {
        // se intercambia la raíz con el último elemento del vector
        // y dicho valor se elimina
        cp[0] = cp.back();
        cp.pop_back();


        // llamada al método hundir con la raíz
        hundir(0);
    }
}
```

Eliminar la raíz del montículo (elemento menor)

```
void colaPrioridad::hundir (int i)  {
    int indice_left = getLeft(i);        // índice del hijo izquierdo
    int indice_right = getRight(i);   // índice del hijo derecho

    int indice_low = i;                    // índice del elemento menor
    if (indice_left < this->size() && cp.at(indice_low) > cp.at(indice_left))  {
        indice_low = indice_left; }
    if (indice_right < this->size() && cp.at(indice_low) > cp.at(indice_right))  {
        indice_low = indice_right; }

    // se intercambia el valor del elemento con el hijo con menor valor
    // y se llama a la función hundir con el índice del hijo
    if (indice_low != i) {
        swap(cp[i], cp[indice_low]);
        hundir(indice_low); }
}
```

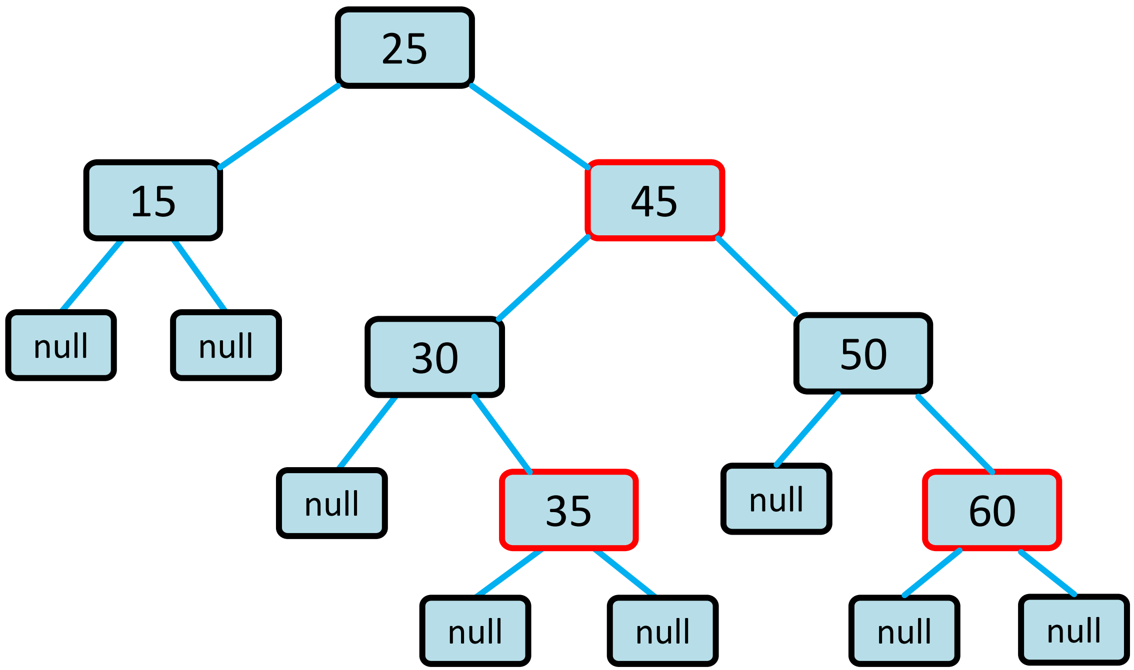
T3. Tipos Abstractos de Datos (TAD)

- **Árboles.**
  - Conceptos generales
  - Realización del TAD Árbol Binario
  - Recorridos de Árboles Binarios
  - Árboles Binarios de Búsqueda (ABB)
  - Árboles Equilibrados (AVL)
  - Montículos
  - **Árboles Rojo-Negro**

Los **árboles rojo-negro** son árboles de búsqueda binarios "equilibrados".

La complejidad o tiempo de ejecución de sus operaciones es ***O(log n)***.

Un **árbol rojo-negro** es una estructura de datos donde en cada nodo se incluye un atributo para indicar el color: **rojo** o **negro**.



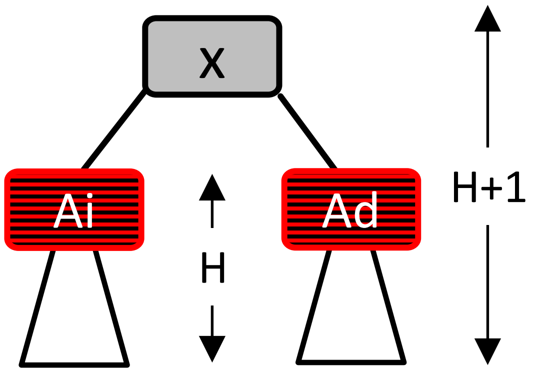
Es un **árbol binario de búsqueda** donde se cumple que:

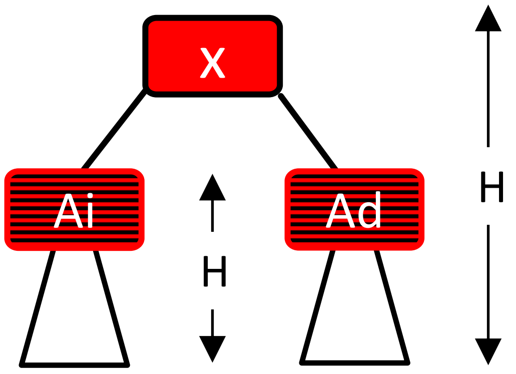
1. Cada nodo tiene estado **rojo** o **negro**.
2. La **raíz** siempre es **negra** (esta condición permite simplificar algunas operaciones).
3. Los **nodos hoja** son los **enlaces nulos** (*nullptr*) y son **nodos negros.**
4. Un **nodo** **rojo** tiene **dos hijos** **negros**.
5. Cualquier **camino desde la raíz hasta una hoja** incluye el **mismo número** de **nodos** **negros**.

- **Altura negra** de un **nodo x (H)**: es el **número** de nodos **negros** desde el nodo **x** hasta cualquier **hoja descendiente** de **x**.

$$
H(x) = \begin{cases} \max(H(x.izq), H(x.dch)) + 1 & \text{si x es } \textbf{negro} \\ \max(H(x.izq), H(x.dch)) & \text{si x es } \textbf{rojo} \end{cases}
$$

- **Altura negra de un árbol** es la **altura negra** del nodo **raíz**.





Un árbol **rojinegro** tiene las siguientes propiedades:

- Si se cambia un nodo de **rojo** a **negro**, la **altura negra** de los **nodos ascendientes** se **incrementa**.

- Cambiar un nodo de **negro** a **rojo** puede afectar a la **condición 4** (un nodo **rojo** tiene **dos hijos negros**), si el padre o alguno de los hijos es **rojo.** Además, se **decrementa** la **altura negra** en todos los **nodos ascendientes**.

- Si después de realizar alguna operación sobre el árbol la **raíz** pasa a ser **roja**, se puede cambiar a **negro** directamente sin afectar al resto de características del árbol.
- Si se borrar un nodo **rojo** no afecta a las condiciones.
- Si se borrar un nodo **negro**, **altura negra decrece en los ascendientes**.

La **inserción** de un nodo se realiza igual que en un ABB.

El nuevo nodo siempre se inserta con color **rojo**.

Los casos que se pueden dar son los siguientes:

- Si el nodo padre es **negro**, el árbol obtenido es **correcto**.
- Si el nodo padre es **rojo**, afecta a una de las condiciones de los árboles rojinegros: “Un nodo **rojo** tiene **dos hijos negros**“.

  Esto implica, **iterar realizando el análisis y la reestructuración** del árbol hasta que se cumplan las condiciones de los árboles **rojo-negro**.

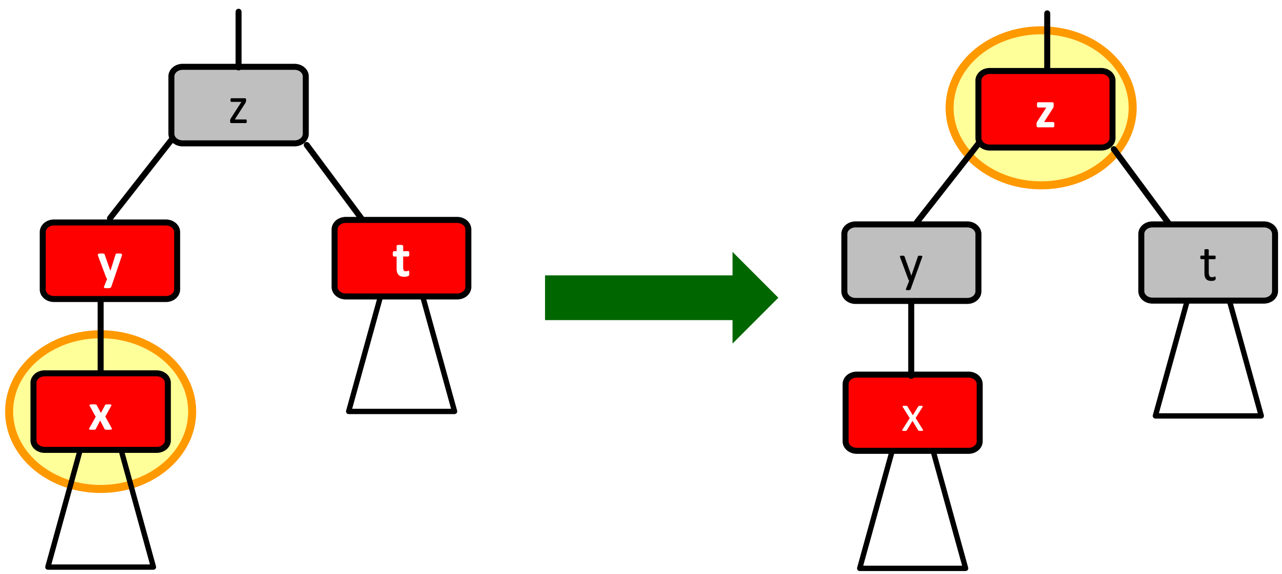
  En cada caso habrá que comprobar un **nodo (x) rojo**, donde su **padre (y) existe** y es **rojo**, su **abuelo (z) existe** y **puede existir** el nodo hermano del padre, **nodo tío (t)**.

  A continuación, se explican los casos considerando que el **nodo padre del nodo x es un hijo izquierdo**. Existen los mismos casos siendo el **nodo padre un hijo derecho**.

Caso 1: Nodo x, hijo izquierdo o derecho, padre (y) y tío (t) rojos

Si tanto el nodo "**analizado**" (**x**) como los nodos **padre** (**y**) y **tío** (**t**) son **rojos**, repintamos a estos últimos de **negro** y al padre de ambos (nodo **abuelo**, **z**) de **rojo** (nodo que pasa a ser "analizado").

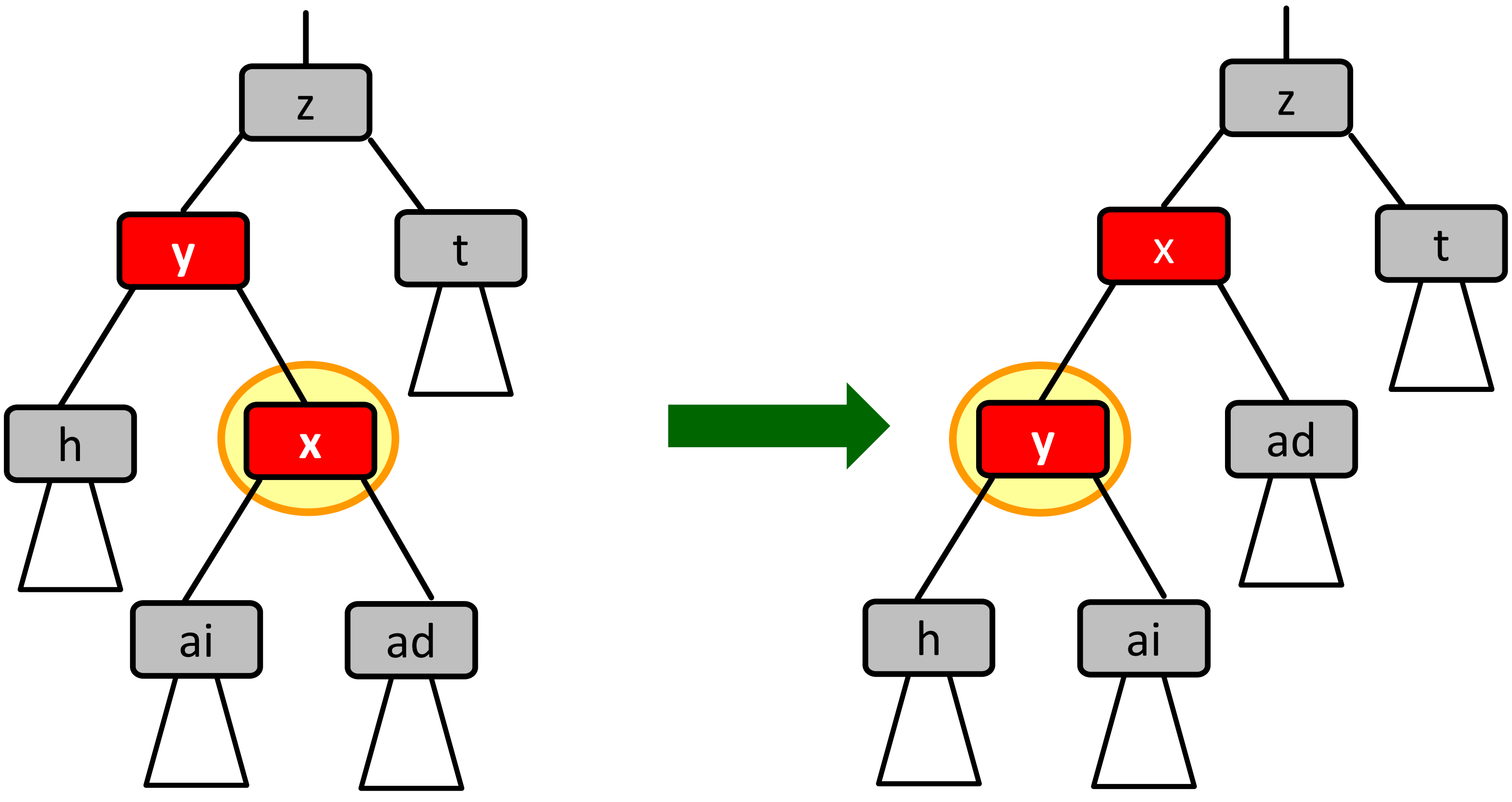
Si el árbol obtenido no es correcto, se realiza **otra iteración** para comprobar el nodo **z** (nuevo nodo "analizado").



Caso 2: Nodo x (hijo derecho) el nodo padre (y) rojos y tío (t) negro

Hay que hacer una Rotación simple dcha-dcha entre los nodos x-y.

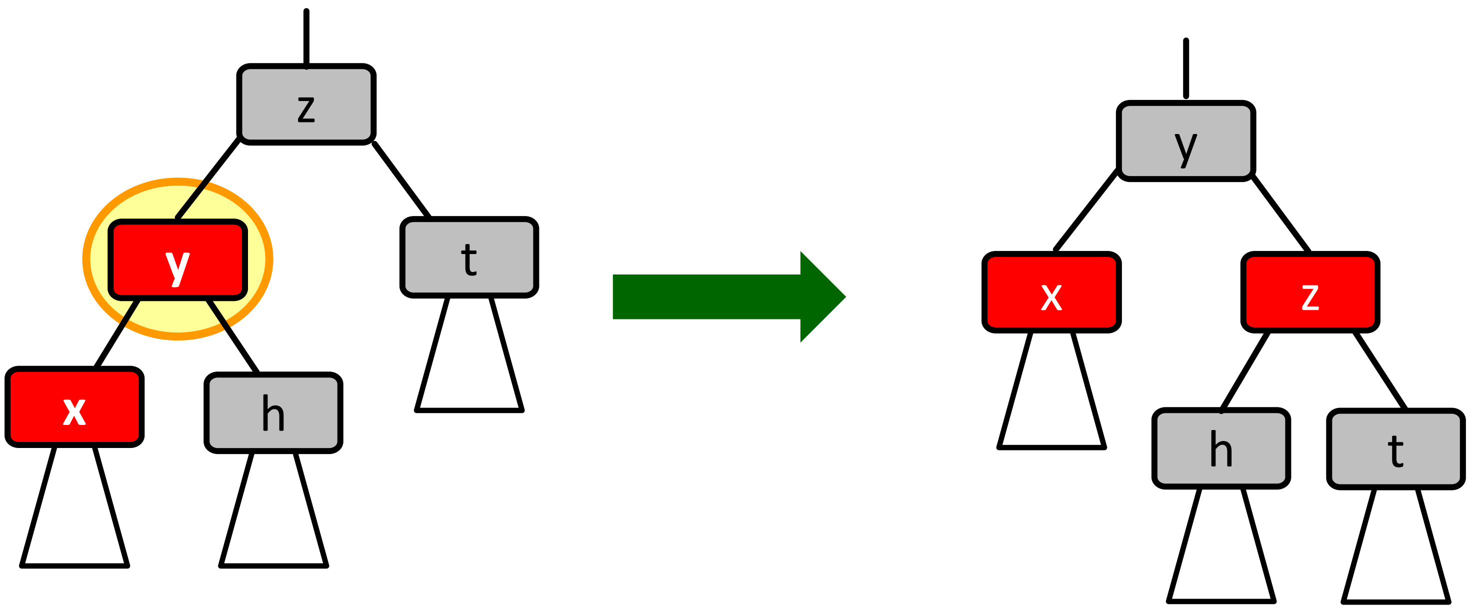
El árbol no es correcto y hay que realizar otra iteración donde el nodo "analizado" es el nodo padre (nodo y).



Caso 3: Nodo x (hijo izquierdo) el nodo padre (y) rojos y tío (t) negro

Hay que hacer una Rotación simple izq-izq entre los nodos z-y que cambian de color.

El árbol es correcto y no hay que realizar más iteraciones.



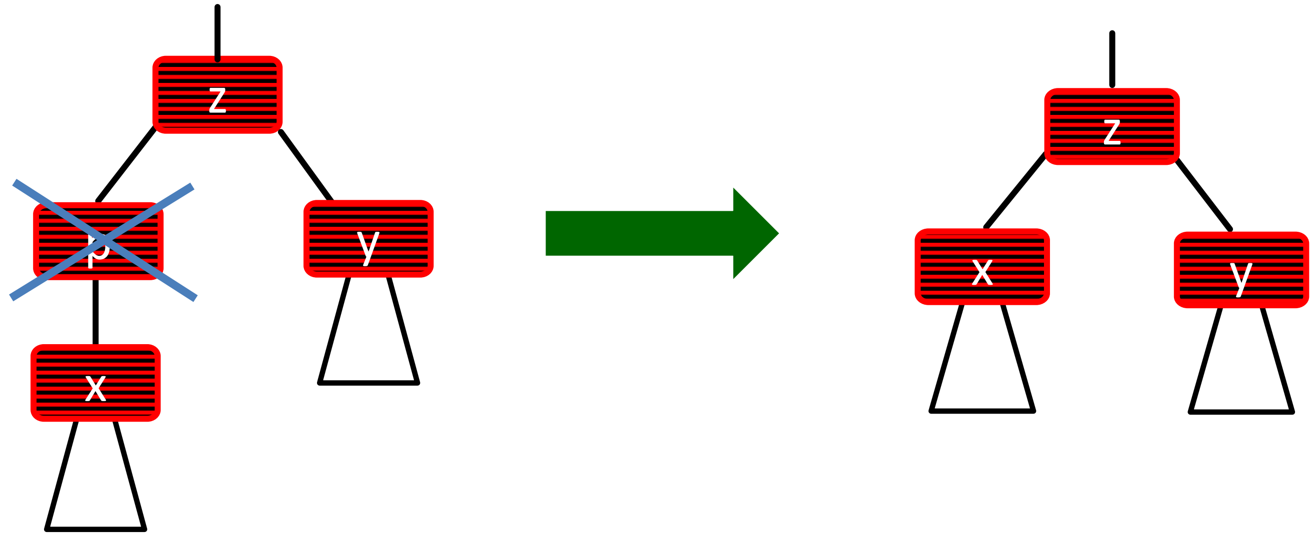
https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Árboles Rojo-Negro – Borrar

El **borrado** de un nodo se realiza igual que en los árboles **ABB**:

1. Se busca el nodo a **borrar** (nodo **p**), que **siempre tendrá dos hijos** porque los nodos **hojas** son los nodos **nulos**.
2. Si el nodo a **borrar** (nodo **p**) es **un nodo con dos hijos no nulos**, se **busca el mayor de los menores** (se identifica como "**x**") y se **intercambia su valor** con el nodo a borrar para proceder a **borrar el nodo "x"**.

   El nodo "**x**", al ser el nodo más a la derecha, tendrá su **hijo derecho nulo**.



Árboles Rojo-Negro – Borrar

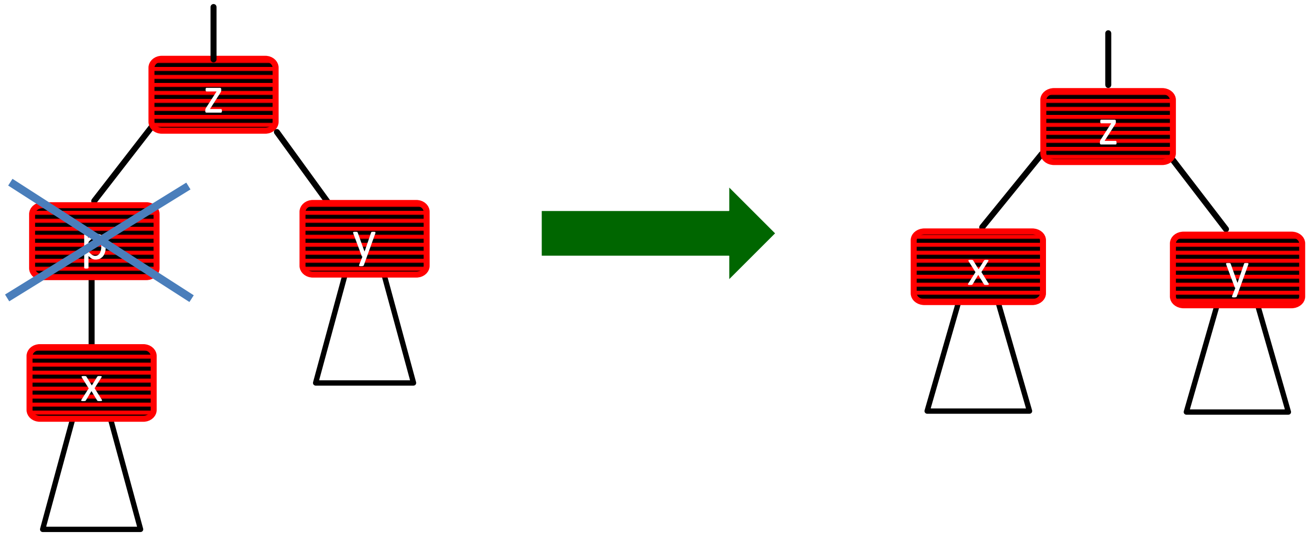
Después de **borrar** el nodo **p** puede ser necesario reestructurar el árbol.

Para ello es necesario conocer los nodos **x** e **y**.

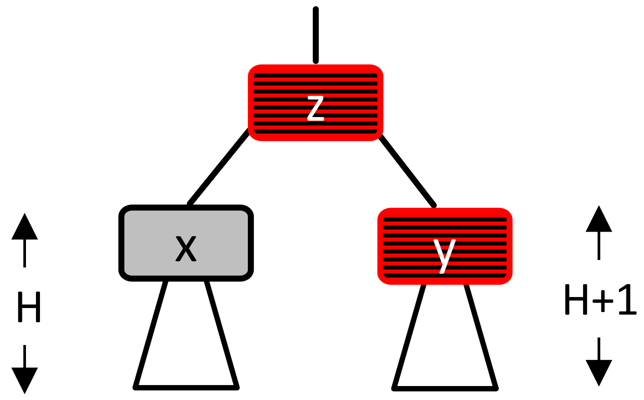
El nodo **x** puede ser nulo, igual que **y**, pero el nodo **z** debe existir.

Si se tiene que **borrar el nodo raíz** se trata como un caso especial, es decir, se **elimina la raíz** y si el **nuevo nodo raíz** es de color **rojo** se cambia su color a **negro**.

La operación de reestructuración consistirá en una comprobación de los casos triviales que se indican a continuación.



Si **no es un caso trivial**, se entra en un **bucle**. En cada iteración se puede tener una estructura donde los nodos **z** e **y no son nodos nulos**, mientras que el nodo **x** puede serlo:

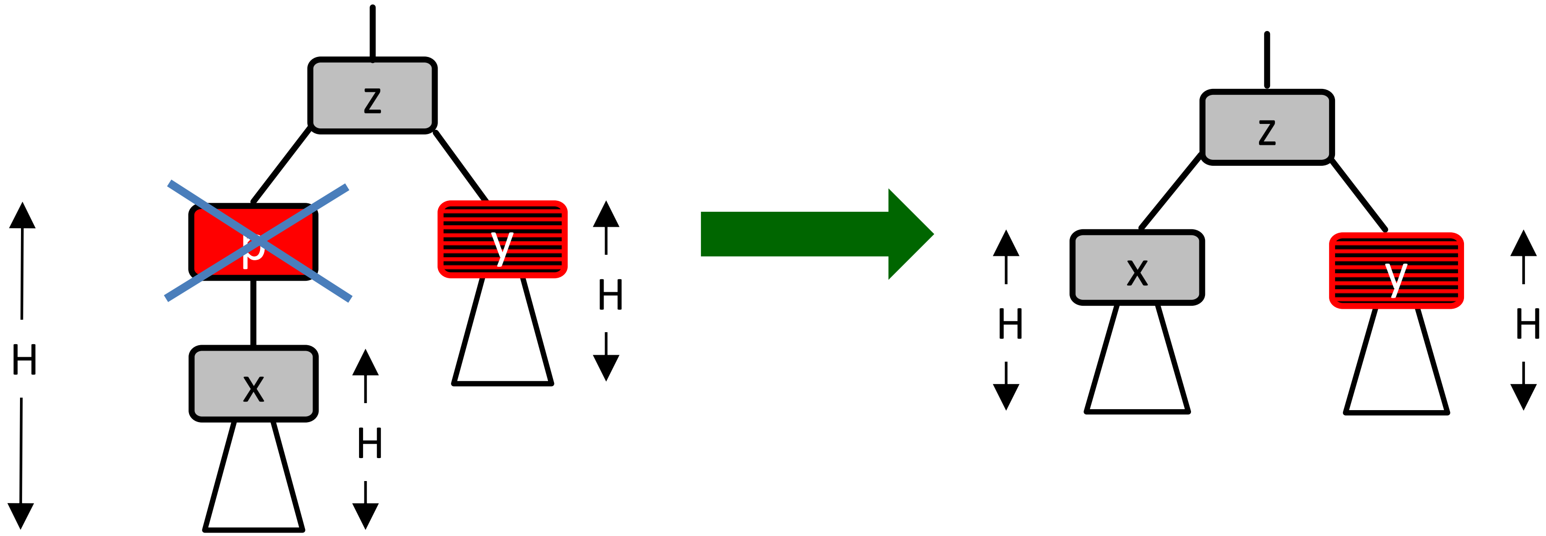


Se comprueba en que caso no trivial se está para realizar la operación de ajuste correcta para, posteriormente, comprobar si es necesario volver a realizar otra iteración.

A continuación, se exponen cinco casos triviales considerando que **x** es un **hijo izquierdo**. De igual forma, existen otros cinco casos cuando **x** es un **hijo derecho**.

Caso trivial: Nodo borrado **rojo**.

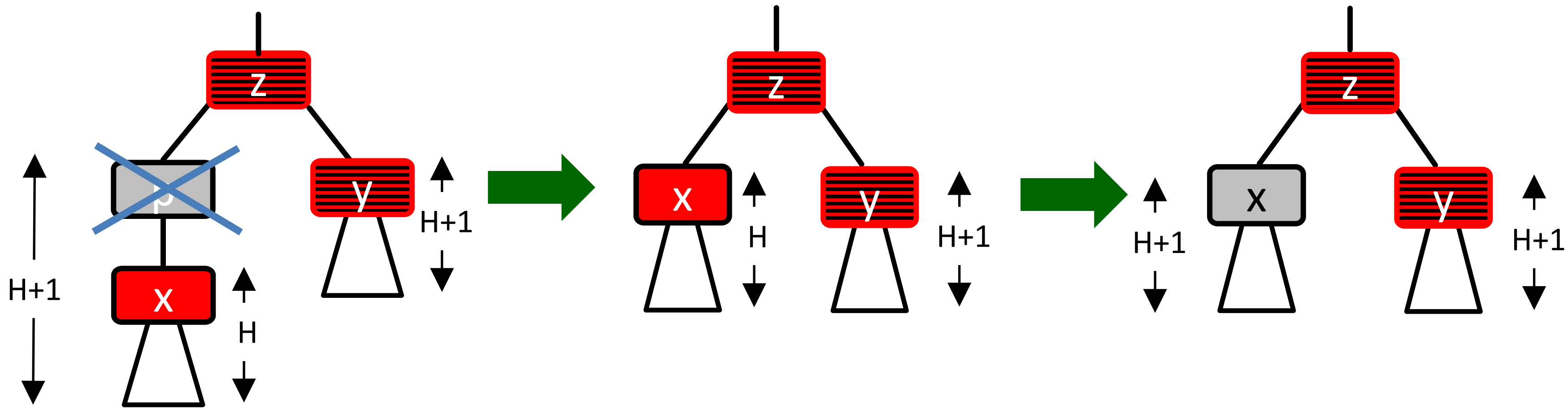
El **árbol es correcto**, sigue siendo rojo-negro. No hay que realizar ningún ajuste.



Árboles Rojo-Negro – Borrar

Caso trivial: El **nodo borrado** tiene un nodo hijo **x** que es **rojo**.

El árbol incumple la condición de que los hijos de **z** tengan la **misma altura negra**. La solución es cambiar el color de **x** a **negro**.



Para determinar en qué caso se está es necesario fijarse en el **color del nuevo padre** (**z**) y, sobre todo, en el del **hermano** (**y**).

Hay un caso en el que hay que tener en cuenta el **color de los sobrinos** (los hijos de **y**).

Además, hay que considerar que:

- Tanto **x** como **y** pueden ser **nulos** (serían nodos **negros**).
- Si un nodo es **rojo** entonces obligatoriamente **no es nulo** y tiene **hijos**.

Caso imposible: Nodo hermano **negro nulo**.

Si el hermano del nodo **x** es **negro** y **nulo**, tiene **altura negra** 1.

Eso significa que, después de borrar el nodo **p,** su altura será 0 porque será una **unidad menos**.

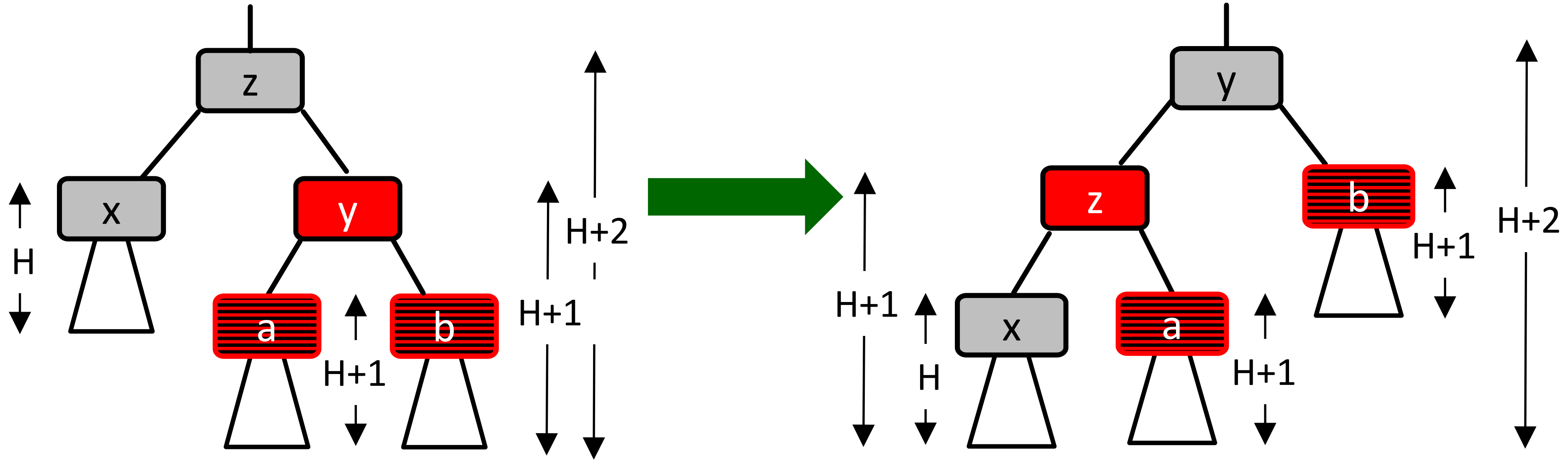
Sin embargo, **x** es un nodo **negro**, y aunque sea nulo tendrá altura 1.

Por lo tanto, no se puede dar que **x** sea **negro** y esté desequilibrado respecto a un **hermano nulo**. Luego, el **hermano debe existir**.

Caso 1: Nodo hermano (**y**) **rojo** y padre (**z**) **negro** .

Se hace una **rotación simple dcha-dcha** entre el padre(**z**)-hermano(**y**) y se cambian los colores.

El nodo **x** pasa tener una altura negra menor que la de su hermano (nodo **a**). Ahora su padre es **rojo**, por tanto habrá que volver a iterar con los mismos nodos (**x** y **z**) y dependiendo del color del nodo **a** se tendrá que aplicar el caso 3, 4 o 5.



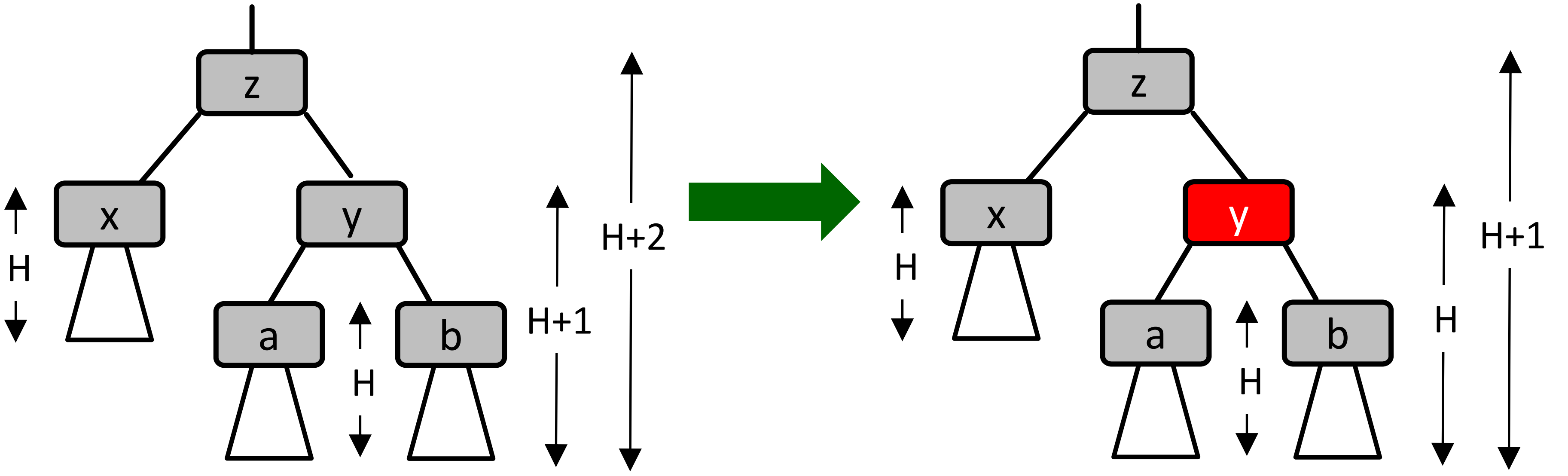
Árboles Rojo-Negro – Borrar

Caso 2: Nodo hermano (**y**) **negro no nulo,** sobrinos (**a** y **b**) y padre (**z**) **negro**.

Se cambia el color del hermano (**y**) a **rojo**. Así los nodos **x** e **y** pasan a tener la **misma altura negra**. Sin embargo, el problema es que la altura de **z** es menor porque ha **disminuido**.

En la siguiente iteración se vuelve a comprobar las condiciones del árbol, pero ahora el nodo llamado **x** es el nodo **z** y el nodo llamado **z** es el padre de **z** (no aparece en el dibujo).

Nota: si **z** es la **raíz del árbol**, se cumplen las condiciones y no hay que iterar.

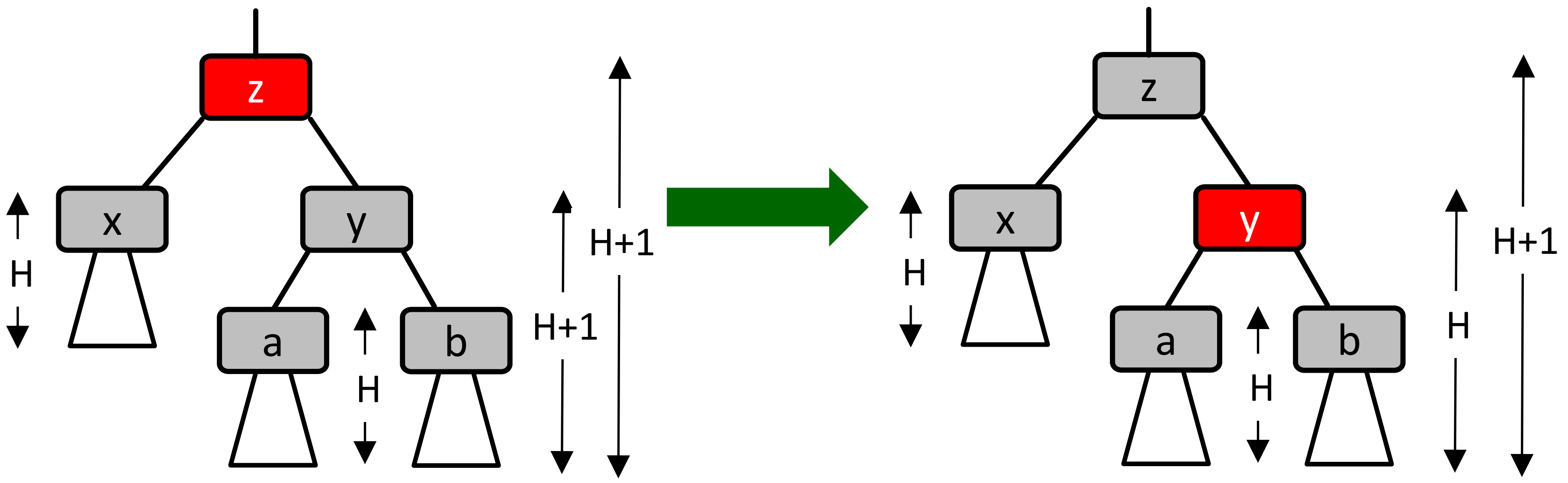


Caso 3: Nodo hermano **negro no nulo,** sobrinos **negros** y padre **rojo**.

Se cambia el color del hermano (**y**) a **rojo** y el del padre (**z**) a **negro**.

Así, los nodos **x** e **y** pasan a tener la **misma altura negra**. La altura de **z** no cambia, es la misma.

El árbol cumple todas las condiciones y se termina de iterar.



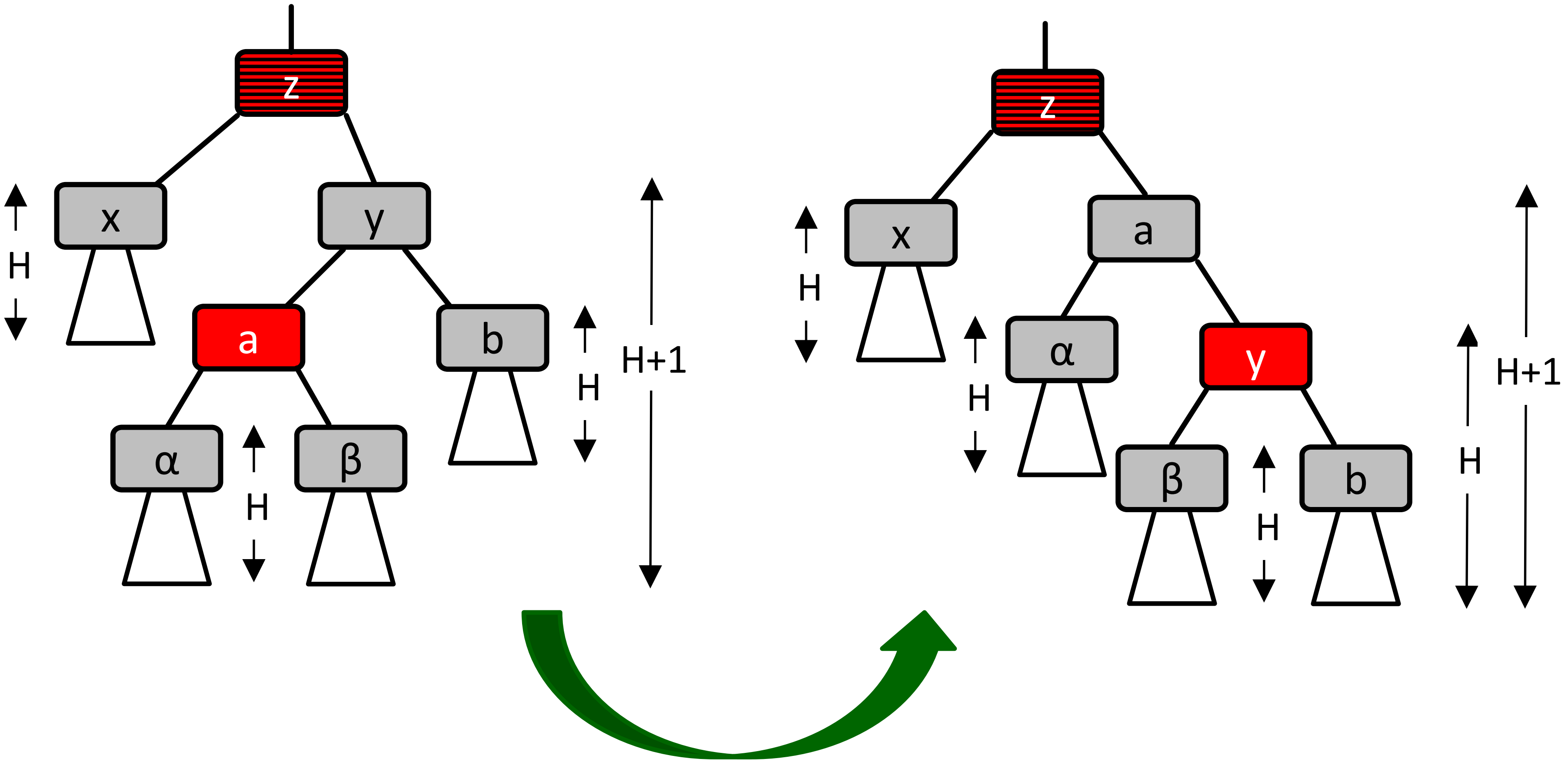
Caso 4: Nodo hermano **negro no nulo,** sobrinos **rojo/negro** y padre **cualquier color**.

Se realiza una rotación simple izq-izq entre hermano-sobrino izquierdo y se cambian sus colores. Los hijos del sobrino izquierdo existen (aunque pueden ser nulos) y son **negros**, ya que el sobrino izquierdo es **rojo**. El nodo **x** pasa a tener como hermano al nodo **a** y sigue teniendo una **altura negra menor** en uno que la de su hermano.

Se tiene que hacer una nueva iteración con los mismo nodos **x** y **z**. En este caso se tendrá que aplicar el caso 5 ya que el hermano sigue siendo **negro** y los sobrinos son **negro** y **rojo**.

Árboles Rojo-Negro – Borrar

Caso 4: Nodo hermano **negro no nulo,** sobrinos **rojo**/**negro** y padre **cualquier color**.



Árboles Rojo-Negro – Borrar

Caso 5: Nodo hermano **negro no nulo,** sobrinos **cualquier color/rojo** y padre **cualquier color**.

Se realiza una rotación simple dch-dch entre padre-hermano y se cambian los colores como sigue:

- El padre (**z**) pasa a ser **negro**.
- El hermano (**y**) toma el color que originalmente tenía el nodo **z**.
- El sobrino derecho pasa de **rojo** a **negro**. Este sobrino debía existir con el color **rojo**.

El árbol cumple todas las condiciones y se termina de iterar.

El nodo **x** nunca cambia de color y puede ser un nodo nulo.

Caso 5: Nodo hermano **negro no nulo,** sobrinos **cualquier color/rojo** y padre **cualquier color**.