Estructuras de Datos y Algoritmos Tema 4: Árboles

Departamento de Informática Universidad de Valladolid

Curso 2021-22



Grado en Ingeniería Informática Grado en Estadística



1. DEFINICIONES Y PROPIEDADES

Definiciones (I)

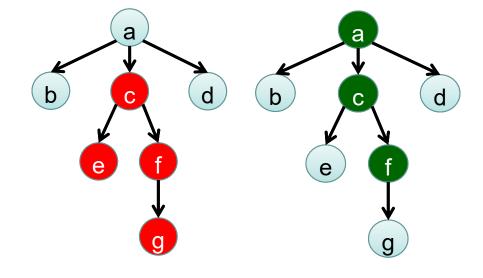


- Un Árbol consiste en un nodo (r, denominado nodo raiz) y una lista o conjunto de subárboles (A₁, A₂, ... A_k).
- Si el orden de los subárboles importa, entonces forman una lista, y se denomina árbol ordenado (por defecto un árbol se supone que es ordenado). En caso contrario los subárboles forman un conjunto, y se denomina árbol no ordenado.
- Se definen como nodos hijos de r a los nodos raices de los subárboles $A_1,\,A_2,\,..\,A_k$
- Si b es un nodo hijo de a entonces a es el nodo padre de b
- Un nodo puede tener cero o más hijos, y uno o níngun padre. El único nodo que no tiene padre es el nodo raíz del árbol.
- Un nodo sin hijos se denomina nodo hoja o externo. En caso contrario se denomina nodo interno.

Definiciones (II)



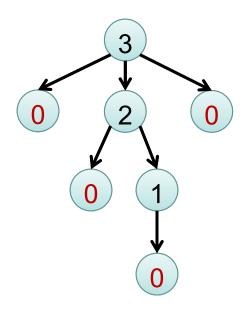
- Se define un camino en un arbol como cualquier secuencia de nodos del arbol, n₁ ... n_p, que cumpla que cada nodo es padre del siguiente en la secuencia (es decir, que n_i es el padre de n_{i+1}). La longitud del camino se define como el número de nodos de la secuencia menos uno (p-1).
- Los descendientes de un nodo (c en el diagrama) son aquellos nodos accesibles por un camino que comience en el nodo.
- Los ascendientes de un nodo (f en el diagrama) son los nodos del camino que va desde la raiz a él.



Altura



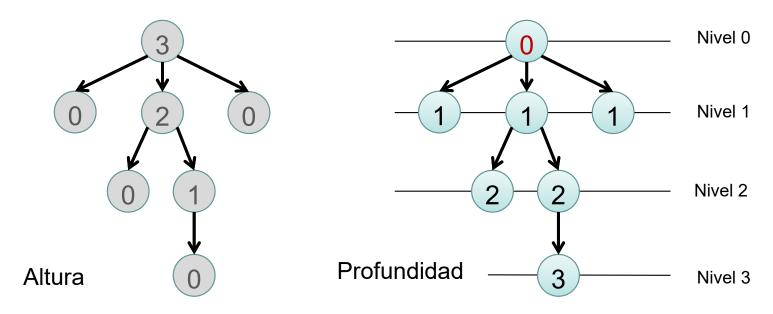
- Se define la altura de un nodo en un arbol como la longitud del camino más largo que comienza en el nodo y termina en una hoja.
 - La altura de un nodo hoja es 0
 - La altura de un nodo es igual a la mayor altura de sus hijos + 1
- La altura de un árbol se define como la altura de la raiz.
- La altura de un arbol determina la eficiencia de la mayoría de operaciones definidas sobre árboles.



Profundidad



- Se define la profundidad de un nodo en un arbol como la longitud del camino (único) que comienza en la raíz y termina en el nodo. También se denomina nivel.
 - La profundidad de la raiz es 0
 - La profundidad de un nodo es igual a la profundidad de su padre + 1



Recorrido de árboles



- Preorden: Se pasa por la raiz y luego se recorre en preorden cada uno de los subárboles. Recursivo.
- Postorden: Se recorre en postorden cada uno de los subárboles y luego se pasa por la raiz. Recursivo.
- Inorden: Se recorre en inorden el primer subárbol (si existe). Se pasa por la raíz y por último se recorre en inorden cada uno de los subárboles restantes. Tiene sentido fundamentalmente en árboles binarios. Recursivo.
- Por Niveles: Se etiquetan los nodos según su profundidad (nivel).
 Se recorren ordenados de menor a mayor nivel, a igualdad de nivel se recorren de izquierda a derecha.
 - No recursivo: Se introduce el raiz en una cola y se entra en un bucle en el que se extrae de la cola un nodo, se recorre su elemento y se insertan sus hijos en la cola.

Recorrido de árboles (II)



• **Preorden**: a,b,c,e,f,g,d

• **Postorden**: b,e,g,f,c,d,a

• **Inorden**: b,a,e,c,g,f,d

• Por Niveles: a,b,c,d,e,f,g

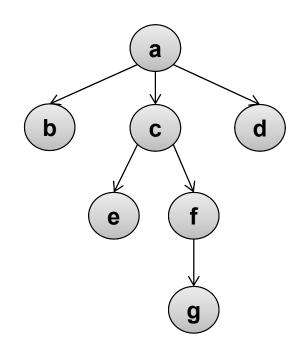
Parentizado sobre subárboles:

• **Preorden**: a (b) (c (e) (f (g))) (d)

Postorden: (b) ((e) ((g) f) c) (d) a

Inorden: (b) a ((e) c ((g) f)) (d)

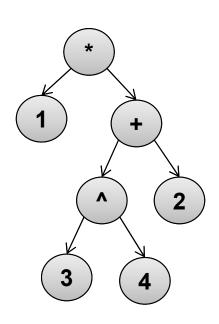
• **Por Niveles**: (a) (b c d) (e f) (g)



Expresiones matemáticas



- Preorden → Notación prefija : * 1 + ^ 3 4 2
- Postorden → Notación postfija: 1 3 4 ^ 2 + *
- Inorden → Notación habitual: 1 * ((3 ^ 4) + 2)



Evaluación de expresiones

Se recorre el arbol en **postorden**: Si es un operando, se inserta en **pila** Si es un operador:

- Se extraen dos operandos
- Se aplica el operador
- Se inserta en pila el resultado Al final, la pila debe contener un único valor, el resultado.





```
-- Un arbol es un nodo que contiene un elemento,
-- de tipo genérico a, y una lista de árboles
data Arbol a = Nodo a [Arbol a]
-- Arhol de test
test :: Arbol Char
test = Nodo 'a' [(Nodo 'b' []), (Nodo 'c' [(Nodo 'e' []),
(Nodo 'f' [(Nodo 'g' [])]), (Nodo 'd' [])]
-- Comprobación de si un nodo es una hoja
esHoja :: Arbol a -> Bool
esHoja (Nodo _ []) = True
esHoja _ = False
-- Valor máximo de una lista
maxlis :: (Ord a) => [a] -> a
maxlis lis = foldl1 max lis
```





```
-- Altura de un árbol
altura :: Arbol a → Int
altura (Nodo _ []) = 0
altura (Nodo _ lis) = 1 + maxlis (map altura lis)
-- Convierte una lista de listas en una lista, concatenándolas
aplanar :: [[a]] -> [a]
aplanar [] = []
aplanar lis = foldl1 (++) lis
-- Recorrido en preorden
preorden :: Arbol a -> [a]
preorden (Nodo x []) = [x]
preorden (Nodo x lis) = x : aplanar (map preorden lis)
```





```
-- Recorrido en postorden
postorden :: Arbol a -> [a]
postorden (Nodo x []) = [x]
postorden (Nodo x lis) = (aplanar (map postorden lis)) ++ [x]
-- Recorrido en inorden
inorden :: Arbol a -> [a]
inorden (Nodo x []) = [x]
inorden (Nodo x (a1:res)) = (inorden a1) ++ [x] ++
                            (aplanar (map inorden res))
-- Recorrido por niveles
niveles :: Arbol a -> [a]
niveles a = nivcol [a] where -- auxiliar, procesa cola
              nivcol [] = []
              nivcol ((Nodo x lis):res) = x : nivcol (res ++ lis)
```



2. REPRESENTACIONES DEL TAD DIRECTORIO

Representaciones

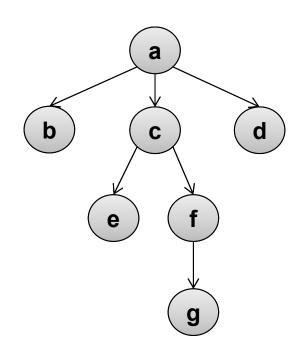


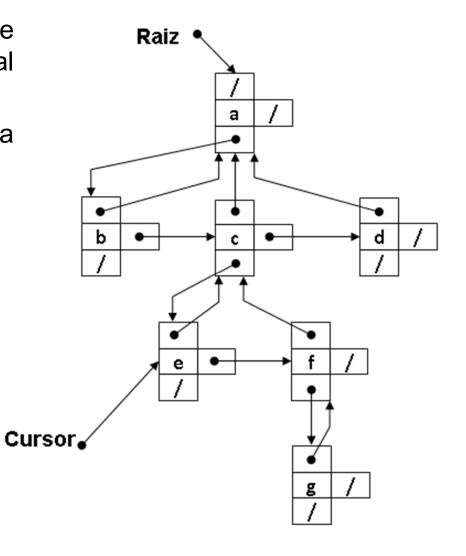
- Las representaciones del TAD Directorio (elementos con relación de jerarquía) suelen ser representaciones enlazadas, donde cada nodo almacena enlaces al nodo padre y/o a los nodos hijos.
- El único nodo distinguido es el nodo raíz.
- El método más habitual de realizar las operaciones es mediante un iterador (cursor) que marca un nodo concreto que sirve de referencia.
- Otra posibilidad es indicar un nodo concreto mediante el camino de la raiz a ese nodo.

Padre - primer hijo - hermano



- Los nodos tienen un número fijo de enlaces: al padre, al primer hijo y al siguiente hermano.
- La lista de hijos esta representada como una lista enlazada.







3. ÁRBOLES BINARIOS

Árboles binarios



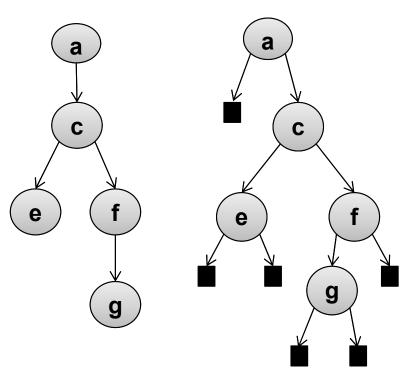
 Árbol binario: Es un árbol que o bien esta vacío (sin contenido) o bien consta de un nodo raiz con dos subárboles binarios, denominados izquierdo y derecho.

 La existencia de árboles vacíos es una convención para que no exista ambigüedad al identificar el subarbol izquierdo y derecho. Se

representa por un cuadrado.

La altura de un árbol vacío es -1

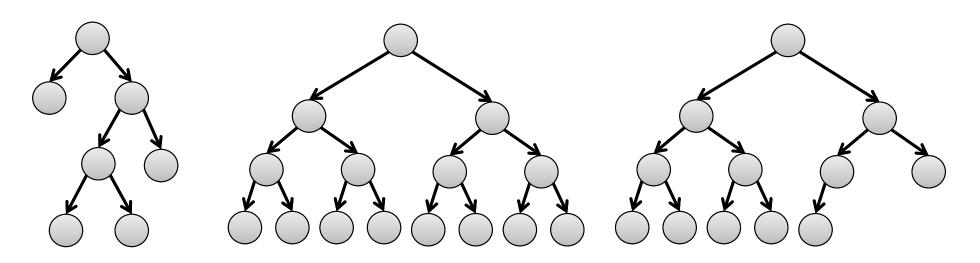
 Cada nodo puede tener 0 hijos (subárbol izquierdo y derecho vacíos), 1 hijo (algún subárbol vacío) o 2 hijos.



Variantes de árboles binarios



- Árbol estricto: Si un subárbol está vacío, el otro también. Cada nodo puede tener 0 ó 2 hijos.
- Árbol Ileno: Árbol estricto donde en cada nodo la altura del subárbol izquierdo es igual a la del derecho, y ambos subárboles son árboles llenos.
- Árbol completo: Arbol lleno hasta el penúltimo nivel. En el último nivel los nodos están agrupados a la izquierda.

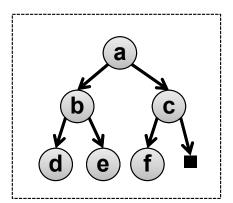


Árboles completos (I)

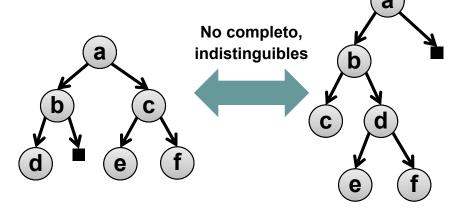


- Los árboles llenos son los árboles con máximo número de nodos (n) para una altura (h) dada. Se cumple que $n=2^{h+1}$ -1
 - El número de nodos de un árbol lleno sólo puede ser una potencia de dos menos uno: 1, 3, 7, 15, 31, ...
- Los árboles completos pueden almacenar cualquier número de nodos y se sigue cumpliendo que su altura es proporcional al logaritmo del número de nodos: $h \in O(\log n)$

 Además tienen la propiedad de que conocido el recorrido por niveles del árbol es posible reconstruirle:



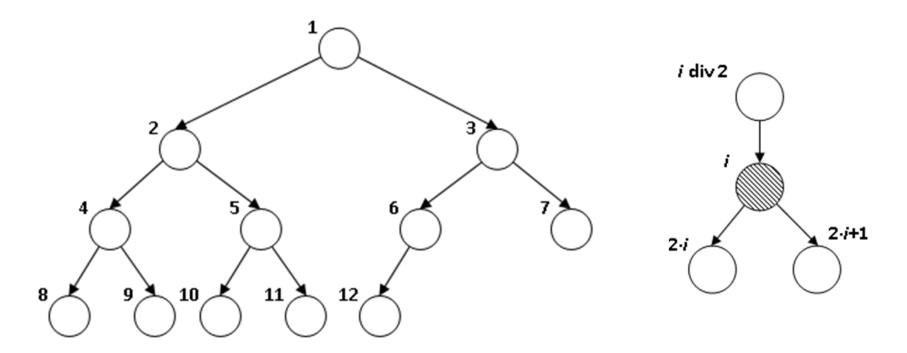
Completo, único.



Árboles completos (II)



 Es posible almacenar un árbol completo en un vector en el orden dado por su recorrido por niveles, y a partir del índice de un elemento en el vector conocer el índice de su nodo padre y los de sus nodos hijos:



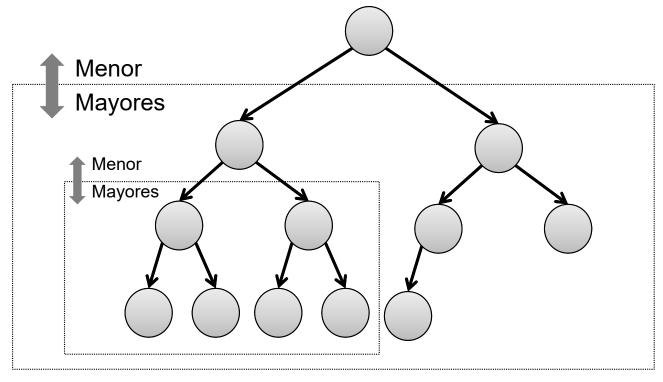


4. MONTÍCULOS (BINARIOS)

Montículo

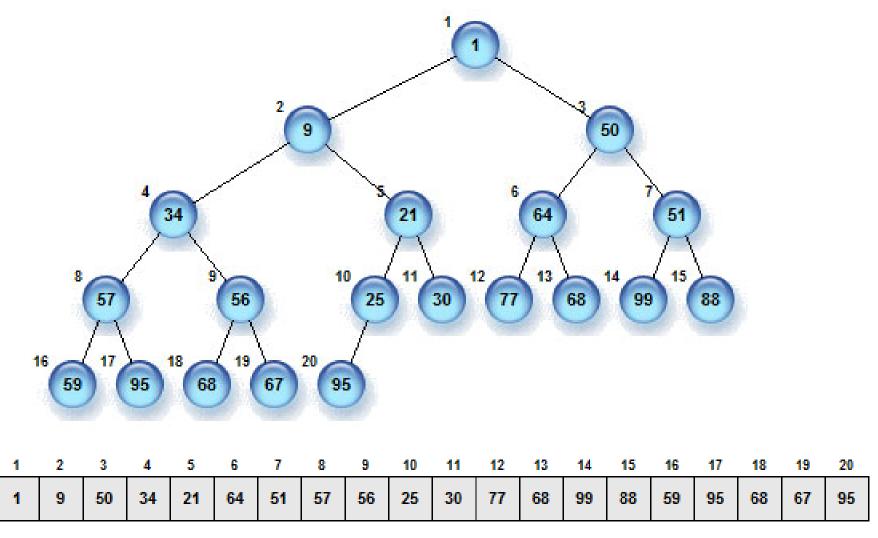


- Un montículo (binario) es un arbol completo cuyos nodos almacenan elementos comparables mediante ≤ y donde todo nodo cumple la propiedad de montículo:
- Propiedad de montículo: Todo nodo es menor que sus descendientes. (montículo de mínimos).



Ejemplo





Propiedades del Montículo



- 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 arbol completo).
- Si un sólo elemento no cumple la propiedad de montículo, es posible restablecer la propiedad mediante ascensos sucesivos en el árbol (intercambiándole con su padre) o mediante descensos en el árbol (intercambiándole con el mayor de sus hijos). El número de operaciones es proporcional a la altura.
- Para insertar un nuevo elemento se situa al final del vector (última hoja del árbol) y se asciende hasta que cumpla la propiedad.
- Para eliminar la raiz se intercambia con el último elemento (que se elimina en O(1)) y se desciende la nueva raiz hasta que cumpla la propiedad.

Utilidad



- 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) (tiempo amortizado).
 - El borrado del mínimo es O(log n).
 - No usa una representación enlazada, sino un vector.
 - La creación a partir de un vector es O(n) y no requiere espacio adicional.
 - El borrado o modificación de un elemento, conocida su posición en el montículo, es O(log n).
- Existen otras operaciones para las que no se comporta bien:
 - Para la búsqueda y acceso al i-ésimo menor se comporta igual que un vector desordenado.
 - La **fusión** de montículos (binarios) es O(n)





```
public class Monticulo<E> implements ColaPrioridad<E> {
 // Vector que almacena los elementos, los hijos de vec[n]
 // son vec[2*n+1] y vec[2*n+2]. El padre es vec[(n-1)/2].
 Object[] vec;
 // Número de elementos
 int num;
 // Ampliar la capacidad del vector
  protected void ampliar() {
   vec = Arrays.copyOf(vec, 2*vec.length);
// Resto de operaciones ...
```





```
void elevar(int i) {
  int k = i;  // Posición del elemento
  E x = (E) vec[i]; // Elemento
  while(k > 0) {
    int p = (k-1)/2; // Posición del padre
    // Si el elemento es >= padre, terminar
    if(vec[k] >= vec[p]) break;
    // En caso contrario, intercambiarlo con el padre
   vec[k] = vec[p];
    k = p;
 // Colocar elemento en posición final
 vec[k] = x;
```





```
void descender(int i) {
  if(num < 2) return;</pre>
  int k = i;  // Posición del elemento
  E x = (E) vec[i]; // Elemento
  int lim = (num-2)/2; // Posición del ultimo nodo con hijos
 while(k <= lim) {</pre>
    int h = 2*k+1; // Posición del primer hijo
   // Escoger el hijo más pequeño
    if(h+1 < num && vec[h] > vec[h+1]) { h++; }
    // Si el elemento es menor que el menor hijo, terminar
    if(x <= vec[h]) break;</pre>
   // En caso contrario, intercambiar con hijo menor
   vec[k] = vec[h];
    k = h;
 vec[k] = x; // Colocar elemento en posición final
```

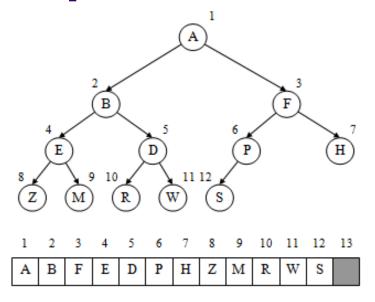


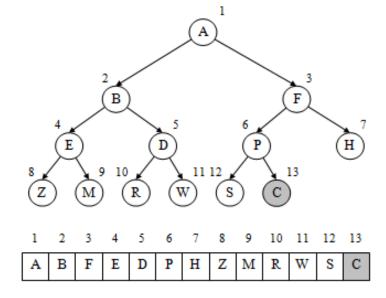


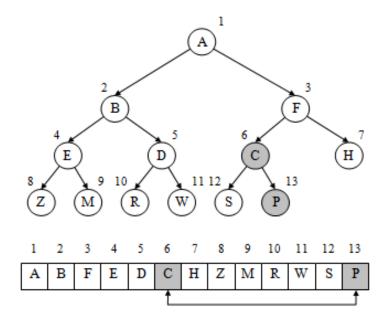
```
public E min() { return (E) vec[0]; }
public void add(E elem) {
 // Ampliar array si lleno
  if(num >= vec.length) ampliar();
 // Poner el elemento al final
  num++;
 vec[num-1] = elem;
 // Elevar el elemento
  elevar(num-1);
```

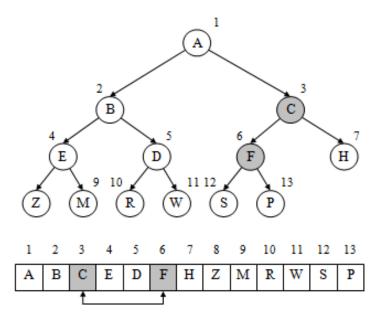
Ejemplo de inserción











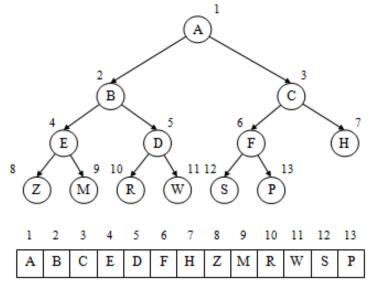


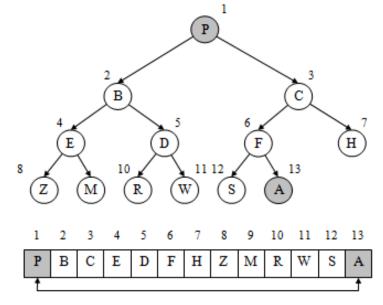


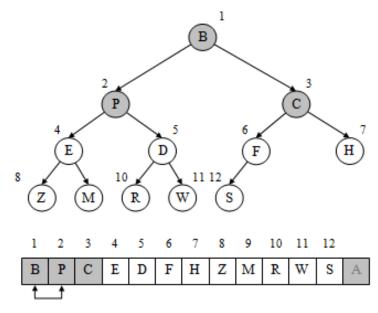
```
public E delMin() {
   E x = (E) vec[0];
   // Mover último a raiz (elemento a borrar)
   vec[0] = vec[num-1];
   vec[num-1] = x;
   num--;
   // Descender el nuevo elemento raiz
   descender(0);
   return x;
}
```

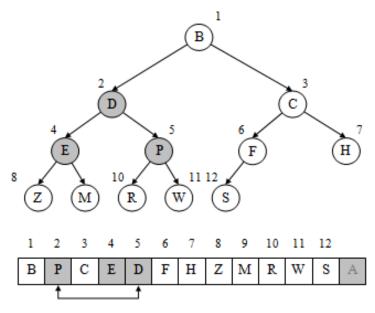
Ejemplo de borrado del mínimo















 Es posible crear un montículo directamente de un array, sin necesidad de realizar n inserciones: Se hace un recorrido por niveles, del penúltimo hacia arriba, descendiendo la raiz de esos subárboles. El orden es O(n) y no se requiere espacio extra.

```
public void crear(Object[] vec) {
    // Se trabaja sobre el vector proporcionado
    this.vec = vec;
    this.num = vec.length;
    // Recorrer nodos por niveles (del último al
    // primero) descendiendo su raiz
    for(int i = (num-2)/2; i >= 0; i--) {
        descender(i);
    }
}
```

Ordenación por montículos



- La ordenación por montículos se basa en la posibilidad de crear un montículo directamente sobre el propio array, y del efecto colateral del borrado del mínimo, que (al intercambiarle con el último) lo coloca en la última posición.
 - Primero se reorganiza un vector desordenado como montículo: Esta operación tarda O(n).
 - A continuación se realizan n extracciones del mínimo: O(n log n).
- El resultado es un montículo vacío (num = 0), pero en el vector que lo sostenía se han depositado los elementos borrados en las posiciones inversas: Se obtiene un vector ordenado de mayor a menor.
 - Con un montículo de máximos se obtendría un vector ordenado de menor a mayor.
- El tiempo es O(n log n) y el espacio O(1).

Otros montículos



- Los montículos que hemos visto son los montículos binarios.
 Existen otros tipos de montículos, generalmente basados en representación enlazada (se sigue manteniendo la propiedad de montículo)
 - Montículo binomial: La operación de fusión de montículos es O(log n), en vez de O(n) como en los binarios. Sin embargo, el acceso al mínimo es O(log n) en vez de O(1).
 - Montículo de Fibonacci: Las operaciones de acceso al mínimo, inserción y fusión son O(1) en tiempo amortizado. La operación de borrado del mínimo es O(log n), también en tiempo amortizado.
 - Montículo Min-Max: Cada nodo en nivel par es menor que sus descendientes, y cada nodo en nivel impar es mayor que sus descendientes.

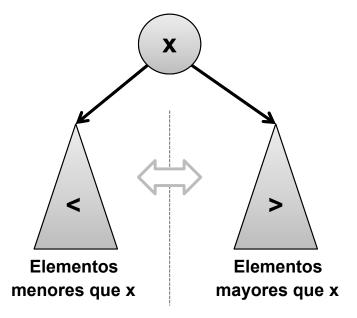


5. ÁRBOLES BINARIOS DE BÚSQUEDA

Árbol Binario de Búsqueda

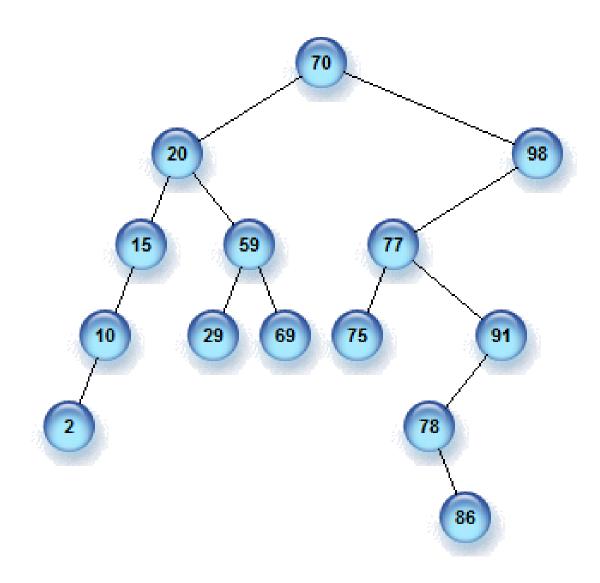


- Un árbol binario de búsqueda (árbol BB) es un árbol binario cuyos nodos almacenan elementos comparables mediante ≤ y donde todo nodo cumple la propiedad de ordenación:
- Propiedad de ordenación: Todo nodo es mayor que los nodos de su subárbol izquierdo, y menor que los nodos de su subárbol derecho.









Propiedades y operaciones



- Un recorrido inorden por el árbol recorre los elementos en orden de menor a mayor.
- El elemento mínimo es el primer nodo sin hijo izquierdo en un descenso por hijos izquierdos desde la raiz.
- El elemento máximo es el primer nodo sin hijo derecho en un descenso por hijos derechos desde la raiz.
- Para buscar un elemento se parte de la raiz y se desciende escogiendo el subárbol izquierdo si el valor buscado es menor que el del nodo o el subárbol derecho si es mayor.
- Para insertar un elemento se busca en el árbol y se inserta como nodo hoja en el punto donde debería encontrarse.
- Para borrar un elemento, se adaptan los enlaces si tiene 0 o 1 hijo. Si tiene dos hijos se intercambia con el máximo de su subárbol izquierdo y se borra ese máximo.





```
public class ArbolBB<E> {
 // Clase interna que representa un nodo BB
 private class Nodo<E> {
    E elem;
            // Elemento
    Nodo<E> izdo, dcho; // Enlaces
    // Constructor (nodo sin enlaces)
    Nodo(E elem) { this.elem = elem; izdo = dcho = null; }
 // Nodo raiz
 Nodo<E> raiz = null;
 // Resto de operaciones ...
```





```
public E get(E elem) {
   if(raiz == null) return null;
   Nodo<E> p = raiz;
   do {
      if(elem == p.elem) break;
      p = (elem < p.elem) ? p.izdo : p.dcho;
   } while(p != null);
   return (p == null) ? null : p.elem;
}</pre>
```





```
public void add(E elem) {
  if(raiz == null) { raiz = new Nodo(elem); return; }
  Nodo<E> ant = null;
  Nodo<E> act = raiz;
  do {
     ant = act;
     act = (elem < act.elem) ? act.izdo : act.dcho;</pre>
  } while(act != null);
  // Insertar nuevo nodo
  act = new Nodo(elem);
  if(elem < ant.elem) {</pre>
    ant.izdo = act;
  } else {
    ant.dcho = act;
```





```
public void del(E elem) {
 // Si el elemento no existe, no hacer nada
  if (get(elem) == null) return;
 // Búsqueda del nodo a borrar (existe)
  if(raiz == null) { raiz = new Nodo(elem); return; }
  Nodo<E> ant = null;
  Nodo<E> act = raiz;
  while(elem != act.elem) {
     ant = act;
     act = (elem < act.elem) ? act.izdo : act.dcho;</pre>
  } while(act != null);
 // act apunta al elemento a borrar y ant a su padre
```

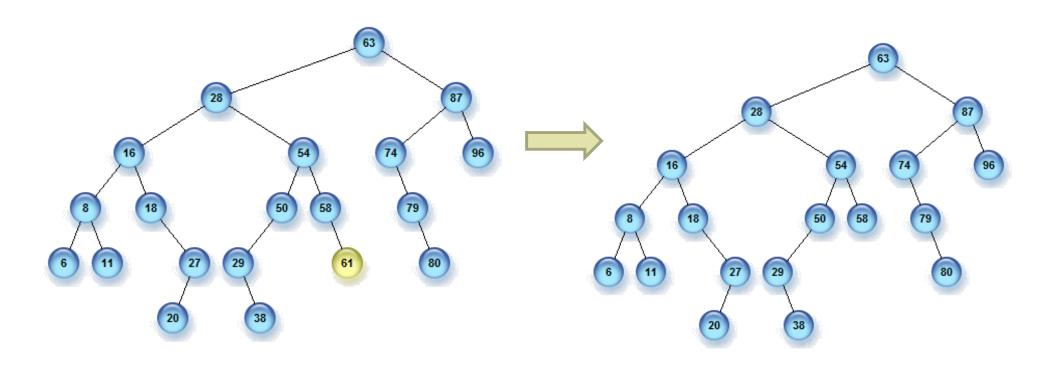




```
// Si tiene dos hijos, lo intercambiamos con
// el máximo de su subarbol izquierdo
if(act.izdo != null && act.dcho != null) {
  Nodo<E> tmp = act;
  ant = act; act = act.izdo;
  while(act.dcho != null) { ant = act; act = act.dcho; }
  tmp.elem = act.elem;
// El nodo a borrar solo tiene 0 o 1 hijos
Nodo<E> h = (act.izdo != null) ? act.izdo : act.dcho;
if(ant == null) {
 raiz = h;
} else {
  if(ant.izdo = act) { ant.izdo = h; } else { ant.dcho = h; }
```

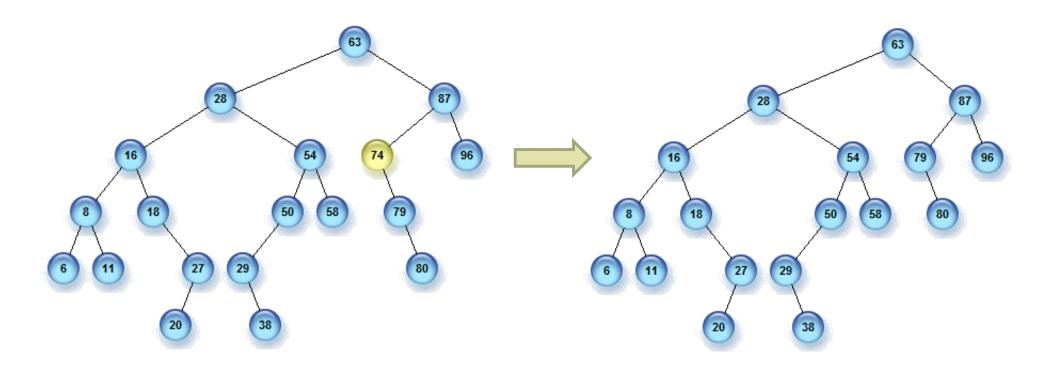
Ejemplo de borrado – 0 hijos





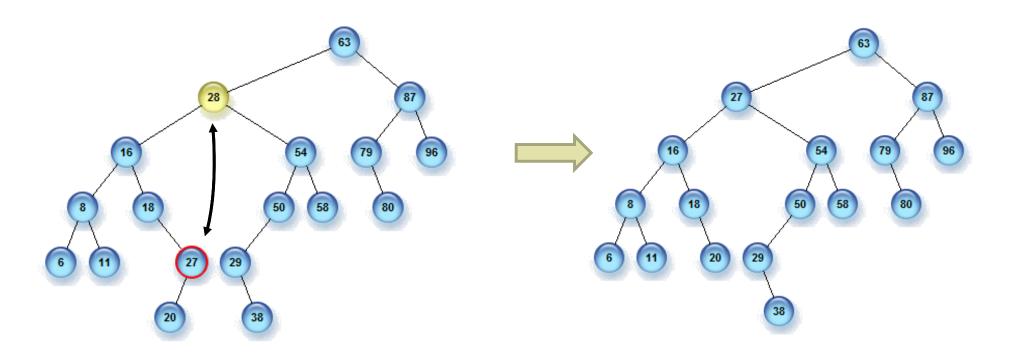
Ejemplo de borrado – 1 hijos





Ejemplo de borrado – 2 hijos

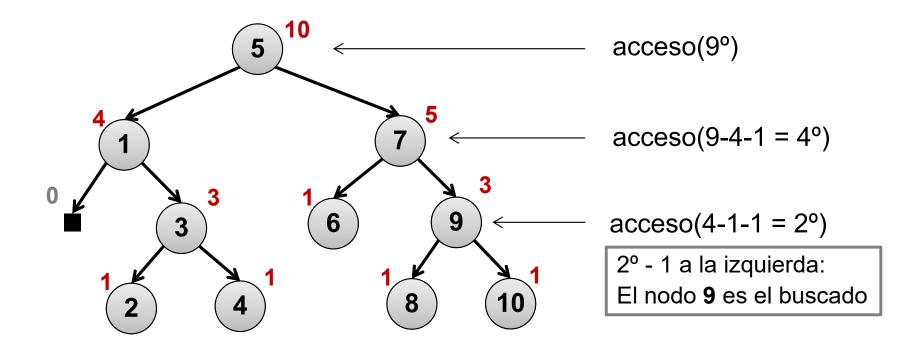




Extensión: Acceso por índice



 Es posible extender un ABB para que la operación de acceso al i-ésimo menor sea eficiente añadiendo un campo a cada nodo que indique el número de elementos del subárbol:



Utilidad

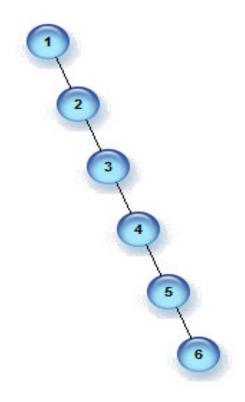


- Un árbol BB podría ser adecuado para representar los TADs Conjunto, Mapa, Diccionario y Lista ordenada:
 - El acceso por valor (búsqueda) es O(h)
 - La inserción por valor es O(h)
 - El borrado por valor es O(h).
 - El acceso al í-ésimo menor (con la extensión anterior) es O(h).
 - El borrado del i-ésimo menor es O(h).
 - La fusión es O(n).
- En las medidas de eficiencia h es la altura del árbol.
 - Se define arbol equilibrado como aquél que garantiza que su altura es logarítmica h ∈ O(log n)
 - Desafortunadamente, los árboles BB no son equilibrados (no tiene porqué cumplirse que la altura sea logarítmica).

Equilibrado en árboles BB



- El que un árbol BB esté equilibrado o no depende de la secuencia de inserciones. Desafortunadamente, el insertar elementos en orden provoca caer en el peor caso: Un árbol lineal (altura O(n), proporcional al número de elementos)
- En un árbol lineal todas las operaciones relevantes serían O(n), arruinando la eficiencia.
- Si los elementos se insertan al azar, se puede demostrar que la altura del árbol BB es, en promedio, logarítmica.





6. ÁRBOLES AVL

Árboles equilibrados



- Los árboles equilibrados son árboles BB que imponen restricciones estructurales para garantizar (o tender a) que su altura sea logarítmica.
- Para ello añaden etapas extra a las operaciones de inserción y borrado (y a veces al acceso)
- Árboles AVL: Imponen que para todo nodo la diferencia de altura entre los subárboles izquierdo y derecho no sea mayor que uno.
- Árboles Rojo-Negro: Los nodos se clasifican como rojos o negros, y se cumple:
 - Los hijos de un nodo rojo son negros
 - Todo camino de la raiz a una hoja pasa por el mismo número de nodos negros.
- Splay Trees: Cada vez que se accede a un nodo se "eleva" en el árbol pasando a ser la raiz (equilibrado "promedio")

Árboles AVL



 Los árboles AVL son árboles BB donde todo nodo cumple la propiedad de equilibrado AVL:

La altura del subárbol izquierdo y del derecho no se diferencian en más de uno.

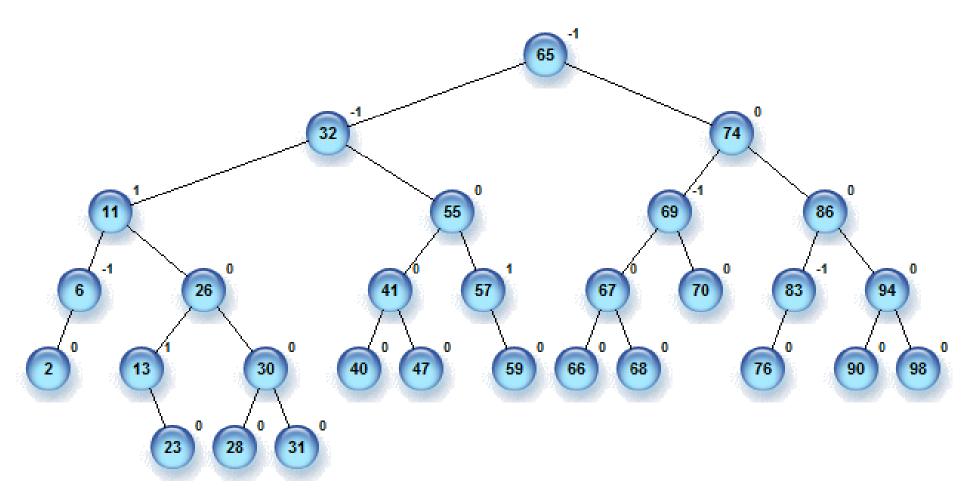
Se define factor de equilibrio de un nodo como:

Fe(nodo) = altura(derecho) – altura(izquierdo)

- En un árbol AVL el factor de equilibrio de todo nodo es -1, 0 ó +1.
- Tras la inserción o borrado de un elemento, sólo los ascendientes del nodo pueden sufrir un cambio en su factor de equilibrio, y en todo caso sólo en una unidad.
- Se añade una etapa donde se recorren los ascendientes. Si alguno está desequilibrado (+2 o -2) se vuelve a equilibrar mediante operaciones denominadas rotaciones.

Ejemplo de árbol AVL

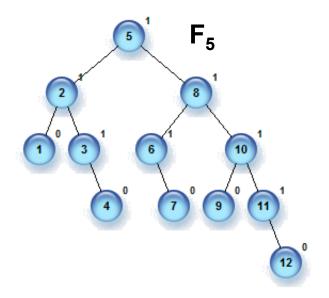




Áltura logarítmica



- Todo árbol binario con equilibrado AVL tiene altura logarítmica
- Se define árbol de Fibonacci (F_h) como:
 - F₋₁ es el árbol vacío.
 - F₀ es el árbol con un único nodo.
 - F_h es el árbol con subárbol izquierdo F_{h-2} y derecho F_{h-1}
- El árbol F_L tiene altura h y número de elementos:



$$N(h) = N(h-1) + N(h-2) + 1$$
$$N(h) \in O(\phi^h) \Rightarrow h \in O(\log n)$$

Un árbol de fibonacci es el árbol AVL con mayor desequilibrio

Operaciones en Árbol AVL

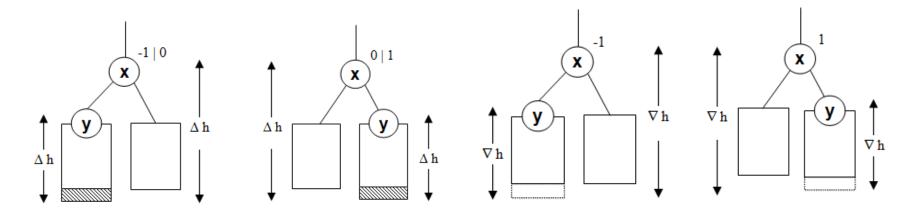


- Un árbol AVL es un árbol binario de búsqueda (ABB), ampliado con un campo que indica el factor de equilibrio de cada nodo.
- Las operaciones de acceso son idénticas a las de un ABB.
- Las operaciones de inserción y borrado se realizan igual que en un ABB, salvo que se añade una etapa posterior de reequilibrado.
- El reequilibrado recorre los ascendientes del nodo que ha sufrido modificación, recalculando sus factores de equilibrio y aplicando las rotaciones adecuadas cuando es necesario.
- El recorrido se detiene al llegar al nodo raiz o cuando el subárbol del nodo actual no haya sufrido cambios en altura respecto a la situación anterior a la operación.
- Es necesario controlar el cambio de altura de los subárboles, dH, a lo largo del recorrido.

Cambios en altura



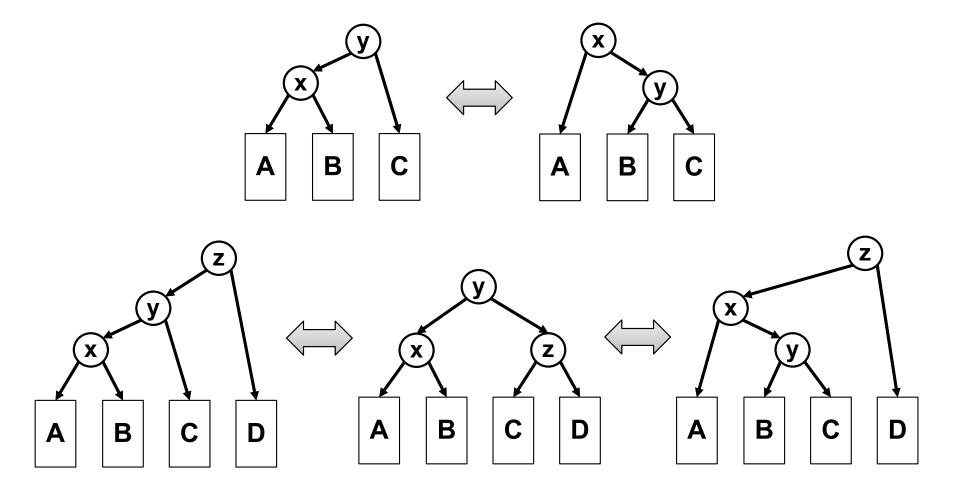
- En inserción (dH > 0), si un hijo (y) incrementa su altura, el padre (x) también la incrementa si su factor de equilibrio era -1 o 0 (hijo izquierdo) o bien 0 o +1 (hijo derecho)
- En borrado (dH < 0), si un hijo (y) decrementa su altura, el padre (x) también la decrementa si su factor de equilibrio era -1 (hijo izquierdo) o +1 (hijo derecho)



Rotaciones



 Una rotación es una reestructuración local de un subárbol BB que mantiene la propiedad de ordenación.



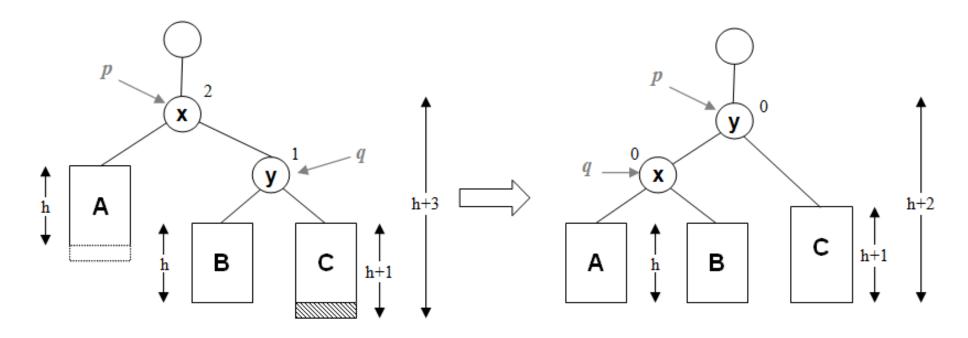
Rotaciones en AVL



- Tras una operación de inserción o borrado, se recorren los ascendientes, recalculando sus factores de equilibrio y teniendo en cuenta el cambio en altura del subárbol.
- Es posible que en el recorrido el factor de equilibrio de algún nodo pasa a valer +2 ó -2 (desequilibrado).
- En ese caso se aplica una determinada rotación que restablece el equilbrio del nodo (aunque es posible que cambie la altura del nodo).
- En un árbol AVL se necesitan 2 tipos de rotaciones (simples y dobles), en un sentido u otro (izquierdas y derechas).
- Teniendo en cuenta los distintos ajustes de factores de equilibrio y posibles resultados respecto al cambio de altura, existen seis casos a considerar.

Rotación 2|1 (Simple derecha)

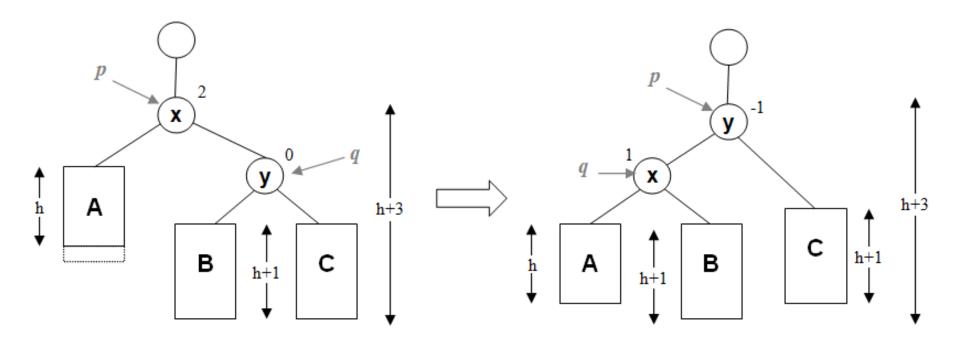




- Posibles causas: Borrado en A que decrementa su altura (sin cambiar la del subárbol x) o inserción en C que incrementa su altura (incrementando la de los subarboles y, x).
- Tras la rotación el subarbol decrementa en uno su altura.

Rotación 2|0 (Simple derecha)

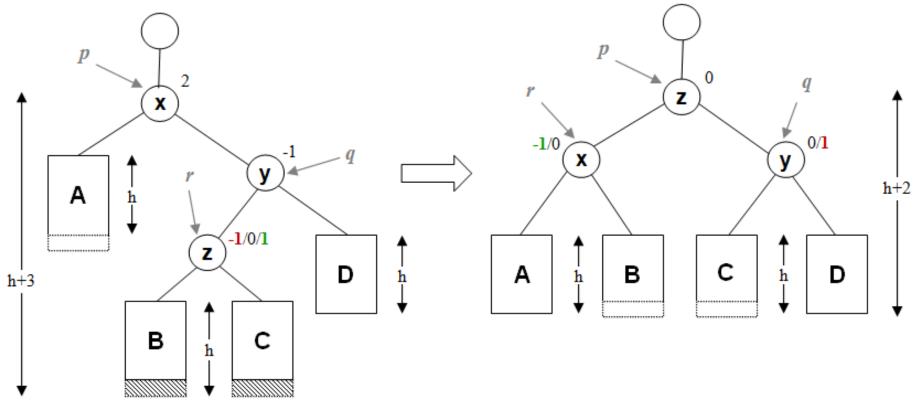




- Posibles causas: Borrado en A que decrementa su altura (sin cambiar la del subárbol x)
- Tras la rotación el subarbol mantiene su altura.
- Las modificaciones son las mismas que el caso anterior

Rotación 2|-1 (Doble derecha)

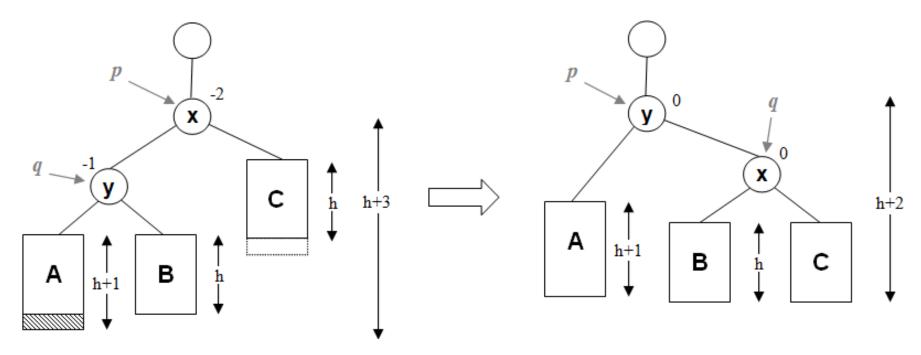




- Posibles causas: Borrado en A que decrementa su altura (sin cambiar la del subárbol x) ó inserción en B ó C que incrementa su altura y la de los subarboles z, y, x
- Tras la rotación el subarbol decrementa en uno su altura.

Rotación -2|-1 (Simple izquierda)

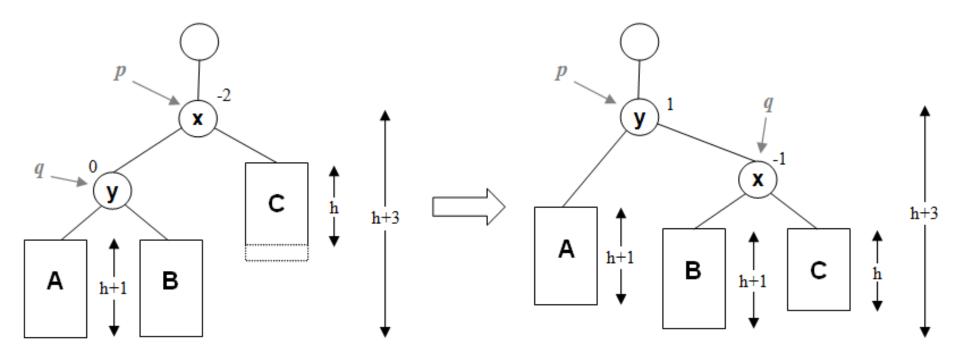




- Posibles causas: Borrado en C que decrementa su altura (sin cambiar la del subárbol x) o inserción en A que incrementa su altura (incrementando la de los subarboles y, x).
- Tras la rotación el subarbol decrementa en uno su altura.

Rotación -2|0 (Simple izquierda)

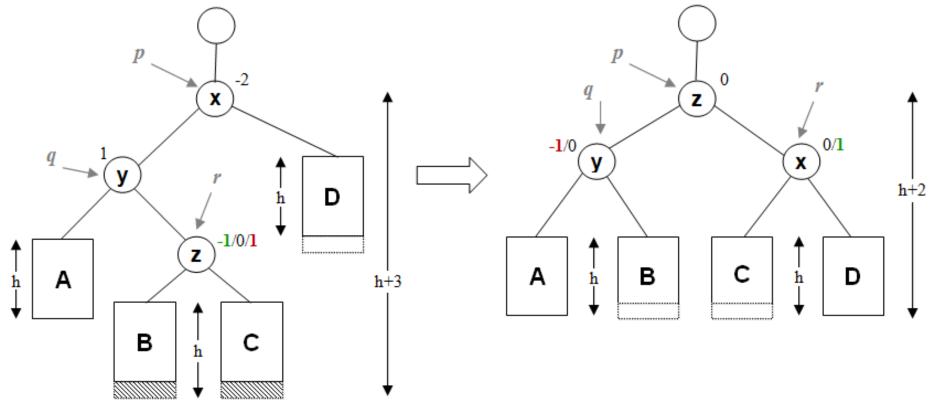




- Posibles causas: Borrado en C que decrementa su altura (sin cambiar la del subárbol x)
- Tras la rotación el subarbol mantiene su altura.
- Las modificaciones son las mismas que el caso anterior.

Rotación -2|1 (Doble izquierda)





- Posibles causas: Borrado en D que decrementa su altura (sin cambiar la del subárbol x) ó inserción en B ó C que incrementa su altura y la de los subarboles z, y, x
- Tras la rotación el subarbol decrementa en uno su altura.





```
-- Nodo e i x d
-- i,d : Subarboles izquierdo y derecho
-- x : elemento almacenado
-- e : Factor de equilibrio = altura(d)-altura(i)
data AVL a = Nulo | Nodo Int (AVL a) a (AVL a)
-- Elemento mínimo
minimo :: AVL a -> a
minimo (Nodo _ Nulo x _) = x
minimo (Nodo _ i _ _) = minimo i
-- Búsqueda
busqueda :: AVL a -> a -> Bool
busqueda Nulo _ = False
busqueda (Nodo _ i y d) x
  | x == y = True
  | x < y = busqueda i x
   x > y = busqueda d x
```

Implementación en Haskell (II)



```
-- Equilibra un subarbol en el cuál el nodo raiz esté deseguilibrado
-- (+2 o -2) aplicando la rotación adecuada. Devuelve un par con el
-- subarbol equilibrado y la modificación de la altura respecto al subarbol
-- anterior (se acumula al parámetro p que toma en cuenta modificaciones
-- de altura de operaciones anteriores).
equil :: AVL a -> Int -> (AVL a, Int)
equil (Nodo 2 a x (Nodo 1 b y c)) p =
      (Nodo 0 (Nodo 0 a x b) y c, p-1)
equil (Nodo 2 a x (Nodo 0 b y c)) p =
      (Nodo (-1) (Nodo 1 a x b) y c, p)
equil (Nodo 2 a x (Nodo (-1) (Nodo e b z c) y d)) p =
      (Nodo 0 (Nodo (min (-e) 0) a x b) z (Nodo (max (-e) 0) c y d), p-1)
equil (Nodo (-2) (Nodo (-1) a y b) x c) p =
      (Nodo 0 a y (Nodo 0 b x c), p-1)
equil (Nodo (-2) (Nodo 0 a y b) x c) p =
      (Nodo 1 a y (Nodo (-1) b x c), p)
equil (Nodo (-2) (Nodo 1 a y (Nodo e b z c)) x d) p =
      (Nodo 0 (Nodo (min (-e) 0) a y b) z (Nodo (max (-e) 0) c x d), p-1)
equil a p = (a,p) - Caso en que no está desequilibrado
```

Implementación en Haskell (III)



```
-- insercion
insertar :: (Ord a) => AVL a -> a -> AVL a
insertar a x = fst (insaux a x)
-- Inserta en un subarbol y devuelve la modificacion en altura
-- (+1,0) del subarbol resultante
insaux :: (Ord a) => AVL a -> a -> (AVL a, Int)
insaux Nulo x = (Nodo 0 Nulo x Nulo, 1)
insaux a@(Nodo e i y d) x
  x < y = let(i',k) =
     insaux i x in equil (Nodo (e-k) i' y d) (min (k*(1-e)) 1)
  x > y = let(d',k) =
     insaux d x in equil (Nodo (e+k) i y d') (min (k*(1+e)) 1)
  x == y = (a,0)
-- Nota: En violeta aparecen fórmulas derivadas de los
-- diagramas de la transparencia 57
```

Implementación en Haskell (IV)



```
-- borrado
borrar :: (Ord a) => AVL a -> a -> AVL a
borrar a x = fst (boraux a x) where
 boraux Nulo _ = (Nulo, 0)
 boraux (Nodo e i y d) x
    x < y = let(i',k) =
      boraux i x in equil (Nodo (e-k) i' y d) (min (-k*e) 0)
    | x \rangle y = let (d',k) =
      boraux d x in equil (Nodo (e+k) i y d') (min (k*e) 0)
    x == y = case(i,d) of
               (Nulo, Nulo) -> (Nulo, -1)
               (Nulo,_) -> (d, -1)
               (_,Nulo) -> (i, -1)
               (_,_)
                           -> equil (Nodo (e-k) i' z d) k where
                                z = maximo i
                                (i',k) = boraux i z
```



7. ANÁLISIS DE EFICIENCIA

Eficiencia TADs Conjunto/Mapa



	Contigua ordenada	Árbol bin. búsq. (peor caso)	Árbol bin. búsq. (caso promedio)	Árbol AVL
Pertenencia (conjunto) Acceso por clave (mapa)	$O(\log n)$	O(n)	O(log n)	O(log n)
Borrado (por valor/clave)	O(n)	O(n)	$O(\log n)$	$O(\log n)$
Inserción (por valor)	O(n)	O(n)	$O(\log n)$	$O(\log n)$
Iterar todos los elementos	O(n)	O(n)	O(n)	O(n)
Unión (ambos tamaño <i>n</i>)	O(n)	O(n)	O(n)	O(n)





	Contigua ordenada	Árbol AVL
Acceso i-ésimo menor	O(1)	$O(\log n)$
Borrado <i>i</i> -ésimo menor	O(n)	$O(\log n)$
Inserción por valor	O(n)	$O(\log n)$
Búsqueda	$O(\log n)$	$O(\log n)$
Fusión	O(n)	O(n)

Nota:

Se supone que los nodos del árbol AVL disponen de un campo extra que almacena el número de elementos del subárbol.

Eficiencia TAD Cola de Prioridad



	Contigua ordenada	Contigua	Arbol AVL	Montículo
Acceso mínimo	O (1)	O (1)	$O(\log n)$	O(1)
Borrado mínimo	O (1)	O(n)	$O(\log n)$	$O(\log n)$
Borrado elemento dada su referencia	O(n)	O(n)	O(log n)	O(log n)
Inserción por valor	O(n)	O (1)	$O(\log n)$	$O(\log n)$
Creación a partir de un array desordenado	$O(n \log n)$		$O(n \log n)$	O(n)
Fusión	$O(n \log n)$	O(n)	O(n)	O(n)





	Contigua ordenada	Enlazada ordenada	Arbol AVL
Acceso por clave	$O(\log n)$	O(n)	$O(\log n)$
Acceso clave <i>i</i> -ésima menor	O (1)	O(n)	O(log n)
Acceso por iterador	O (1)	O (1)	O (1)
Borrado por clave	O(n)	O(n)	$O(\log n)$
Borrado clave <i>i</i> -ésima menor	O(n)	O(n)	O(log n)
Borrado por iterador	O(n)	O (1)	$O(\log n)$
Inserción por valor	O(n)	O(n)	$O(\log n)$



8. ÁRBOLES B

Motivación



 Los sistemas de almacenamiento masivo suelen tener un tiempo de acceso mucho mayor que el tiempo de transferencia: La localización de un elemento es mucho más costosa que la lectura secuencial de datos, una vez localizados.



- Esto se aplica sobre todo a discos duros, pero también, aunque en menor medida, a memorias de estado sólido (flash) e incluso a memorias volátiles.
- Esto supone un problema para estructuras enlazadas, como los árboles AVL, donde las operaciones acceden a bastantes nodos de pequeño tamaño.
- Para grandes volúmenes de datos, sería conveniente reducir el número de accesos, a cambio de que esos accesos contuvieran elementos de mayor tamaño.





- El SACYL trabaja con una base de datos de unas 2.500.000 tarjetas sanitarias, ocupando cada una aprox. 1 Kb de datos.
- Si se almacenan en un árbol AVL, su altura (árbol de Fibonacci) sería:

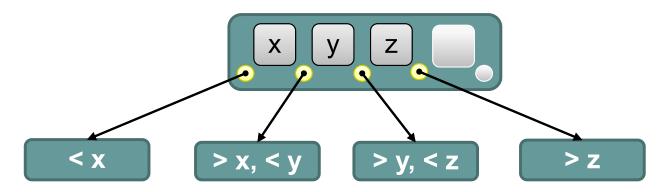
$$h = \log_{\phi} 2.500.000 = 31.9$$

- Lo que supone entre 25-31 accesos a disco para cualquier búsqueda de un elemento.
- En cambio, si se almacenan en un Árbol B de orden 1.000 (aproximadamente 1 Mb por nodo) tendría altura 3, o 2 con una ocupación media del 80%.
- Sólo se necesitarían 1 ó 2 accesos a disco (la raiz reside en memoria) para cada búsqueda.
- El orden para ambos casos es logarítmico, pero si el tiempo de acceso es dominante, la segunda solución sería 10-30 veces más rapida.

Árboles (a,b)



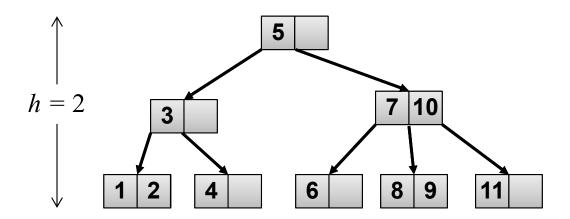
- Los árboles (a,b) son árboles generales (no binarios) donde cada nodo interno puede tener un número de hijos, m+1, en el rango [a,b].
- Cada nodo almacena m claves (elementos comparables por ≤), ordenadas de menor a mayor, que sirven para que se pueda usar como un árbol de búsqueda.
- El contenido típico de un nodo consiste en:
 - Un entero, m ∈ [a-1,b-1], que indica el número de claves almacenadas.
 - Un vector, **c**, de capacidad **b-1**, que almacena las *m* claves.
 - Un vector, e, de capacidad b, que almacena los m+1 enlaces a hijos.
- Propiedad de ordenación: Nodo e[i] almacena claves menores que c[i]



Árboles B

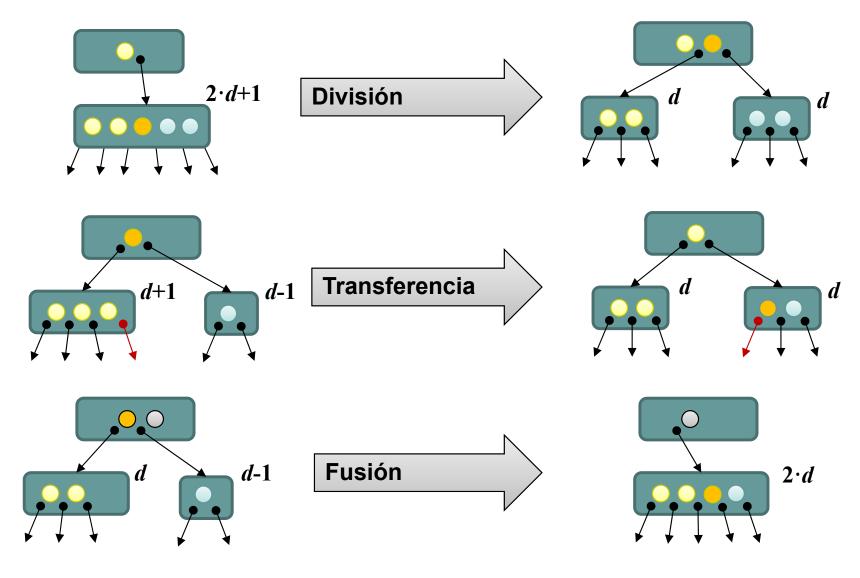


- Un árbol B (Bayer-McCreight 1972) de orden d es un árbol (d+1,2d+1) con las propiedades adicionales siguientes:
 - La raiz puede tener cualquier número de claves.
 - Todas las hojas se encuentran a la misma profundidad, h.
- La segunda propiedad garantiza que un árbol B es un árbol equilibrado:
 Su altura es logarítmica respecto al número de claves almacenadas.
- **Ejemplo**: Un árbol B de orden 1 es un árbol (2,3): Cada nodo puede contener 1 o 2 claves y tener 2 o 3 hijos.



Reestructuraciones





Búsqueda e Inserción



Búsqueda:

- Se desciende desde la raiz hasta el nodo que contenga el elemento (o bien llegar a una hoja que no lo contenga).
- En cada nodo se busca en el array de claves (búsqueda secuencial o binaria). Si no se encuentra, se pasa al hijo asociado a la primera clave mayor que el valor buscado (o el último hijo si el valor buscado es mayor que todas las claves).

Inserción:

- Se desciende (igual que en la búsqueda) hasta el nodo hoja que debería contener el elemento.
- Se inserta en la posición adecuada del array de claves.
- Si con ello se supera el número máximo de claves (2d), el nodo se divide, transfiriendo su clave en posición media al padre.
- Es posible que el padre deba dividirse a su vez, y así con todos los ascendientes.

Borrado



Borrado en nodo interno:

- Se desciende desde la raiz hasta el nodo que contenga el elemento a borrar.
- Se intercambia con el máximo del hijo izquierdo o con el mínimo del hijo derecho (se elige el hijo con más claves).
- Se pasa a borrar el elemento en el hijo (al final el borrado se produce en un nodo hoja)

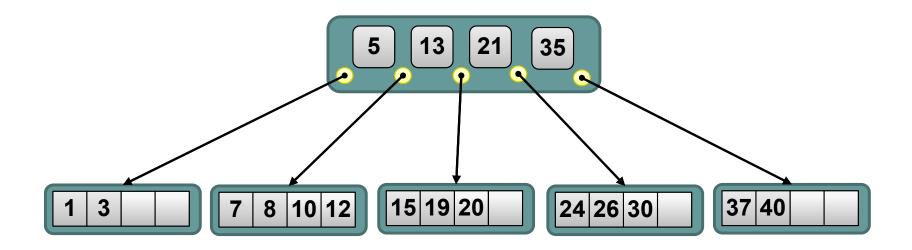
Borrado en nodo hoja:

- Se elimina del array de claves (desplazamiento).
- Si con ello el número de claves es *d*-1:
 - Se intenta una **transferencia** con el hermano izquierdo o derecho, el que contenga más claves.
 - Si no es posible (ambos tienen d hijos o no existen), se produce una fusión con el hermano izquierdo (o el derecho, si no existe).
 - La fusión toma un elemento del padre, por lo que éste a su vez puede necesitar transferencias o fusiones (y así con los ascendientes)





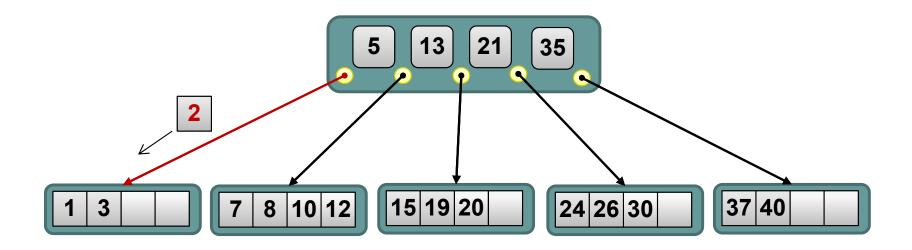
Inserción del valor 2 en árbol B de orden 2 (árbol (3,5))







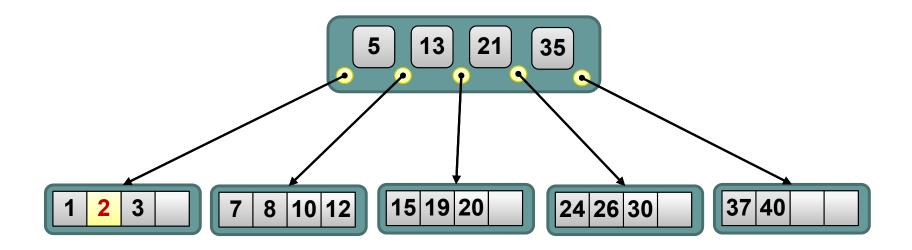
Se busca el nodo hoja donde debe encontrarse el elemento







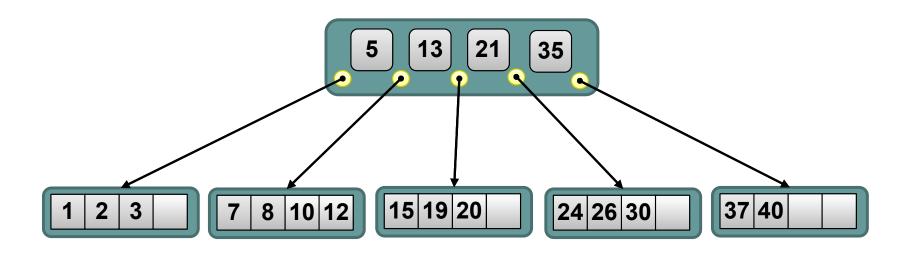
Se inserta en orden en la hoja (desplazamiento)



Inserción – División de nodos



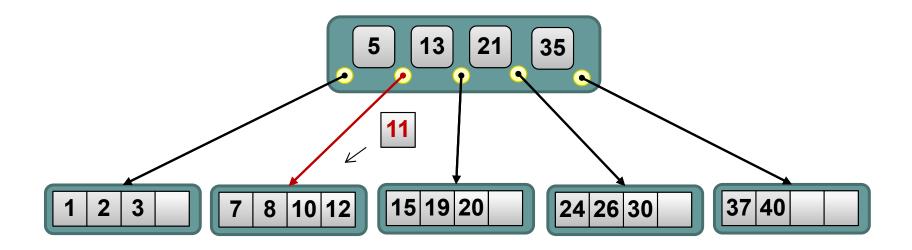
Inserción del valor 11







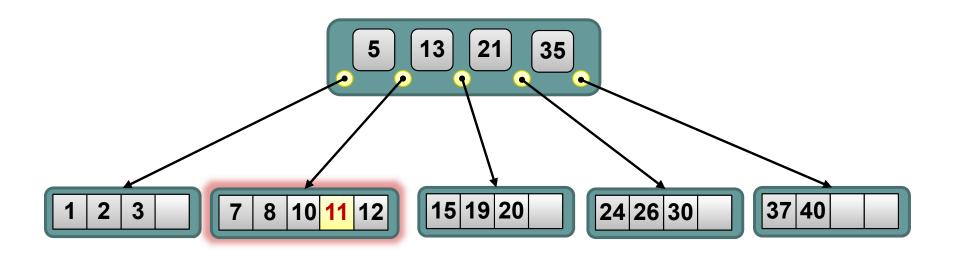
Se busca el nodo hoja donde debe encontrarse el elemento







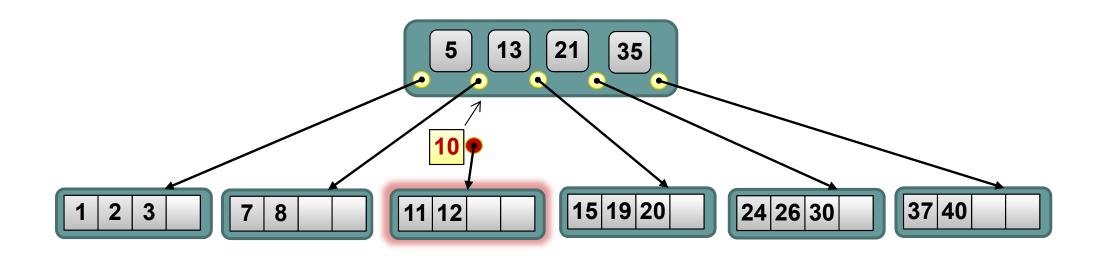
• Se inserta en el nodo. En este caso el nodo sobrepasa el límite de claves (4).







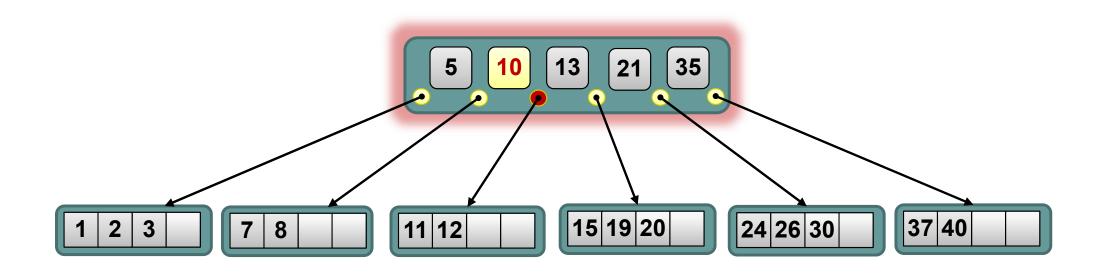
 Se crea un nuevo nodo y se traslada la mitad derecha de los elementos a él. El elemento en posición media (10), junto con el enlace al nuevo nodo, se envía al padre para su inserción







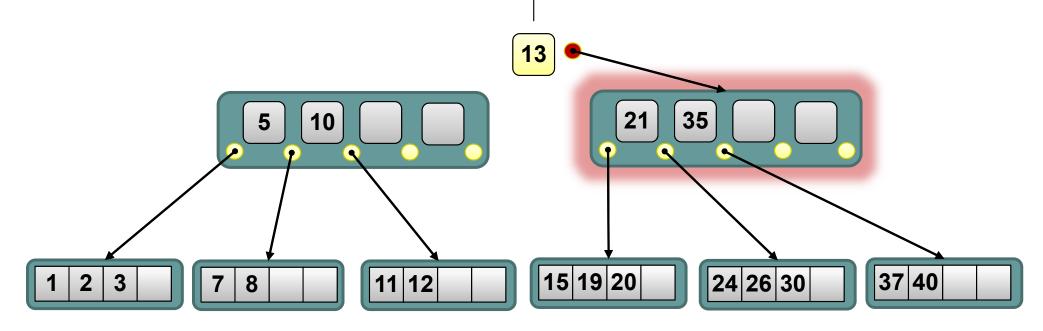
 Se inserta en el nodo padre. Se sobrepasa el límite de claves permitidas (4)





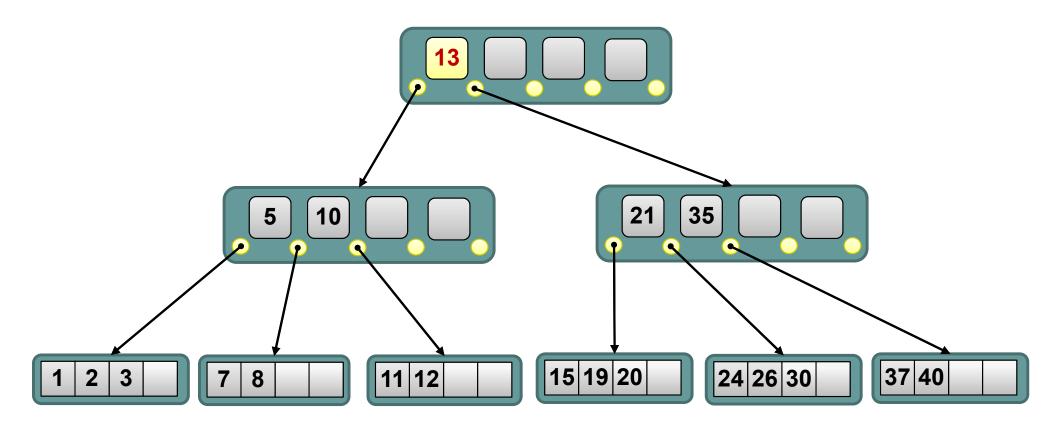


 Se crea un nuevo nodo y se traslada la mitad derecha de los elementos a él. El elemento en posición media (13), junto con el enlace al nuevo nodo, se envía al padre para su inserción



Inserción – División de nodos

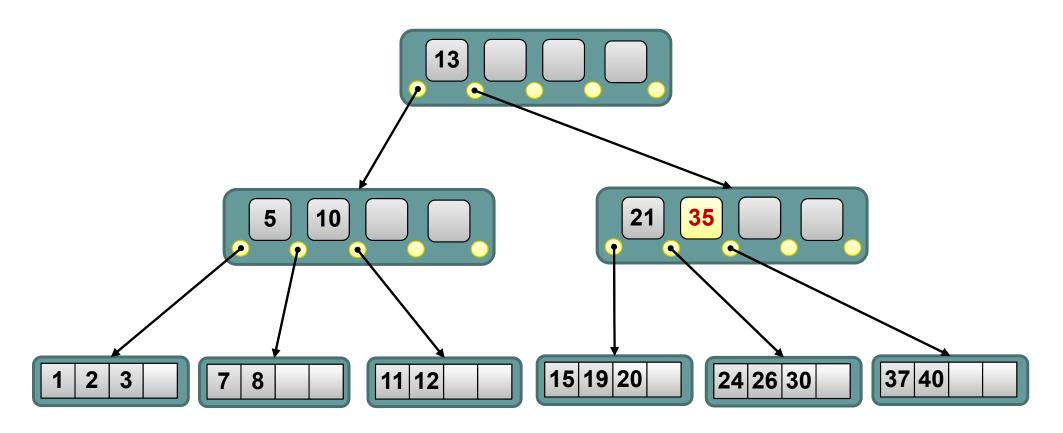




 Como no existe padre, se crea un nuevo nodo raiz que contiene únicamente esa clave

Borrado – Sin reestructuración

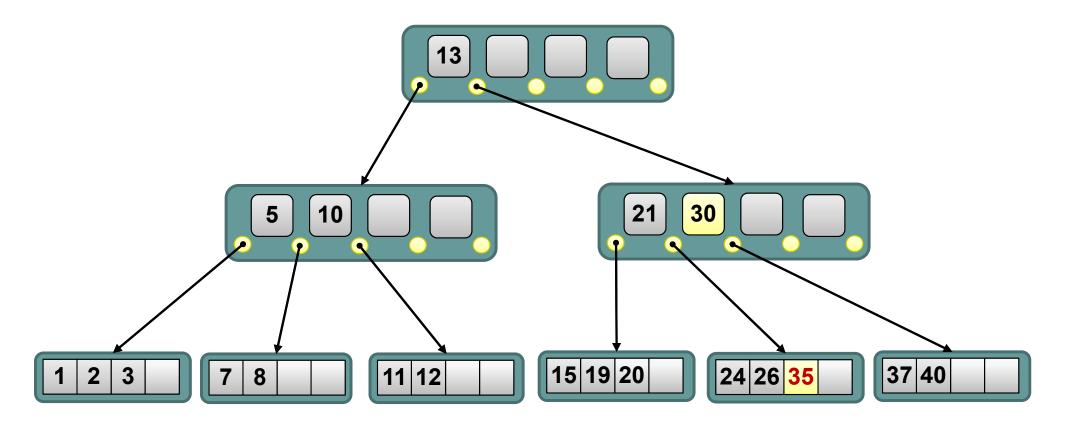




Borrado de la clave 35. Se busca el nodo donde está el elemento.

Borrado – Sin reestructuración

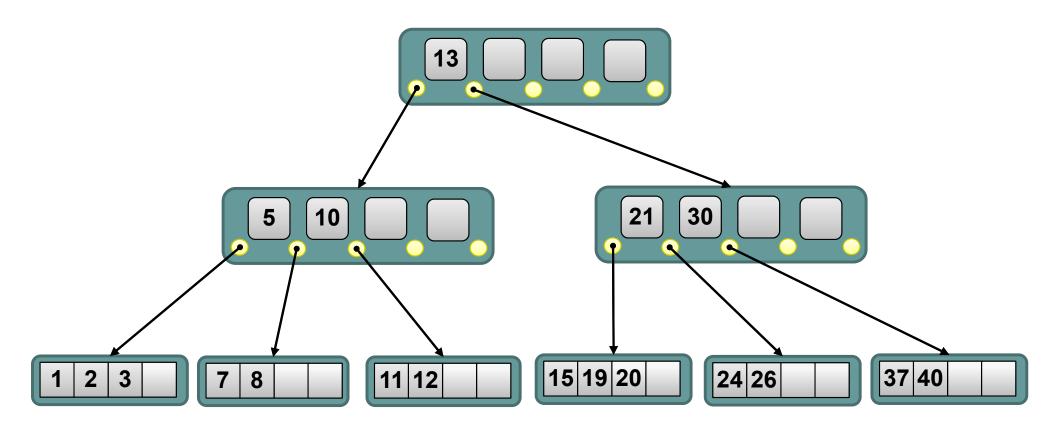




• Es un nodo interno. Se intercambia con el máximo de su hijo izquierdo y se pasa a borrar esa clave.

Borrado - Sin reestructuración

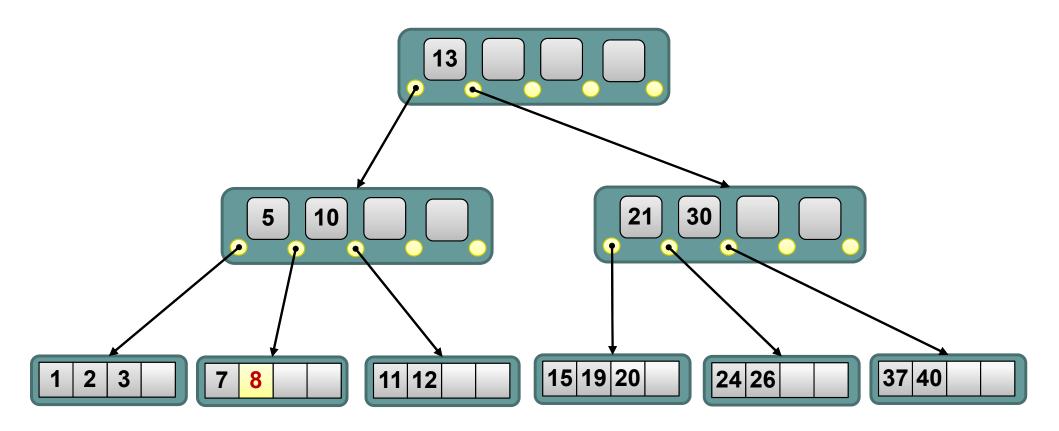




Se borra el elemento.

Borrado – Transferencia

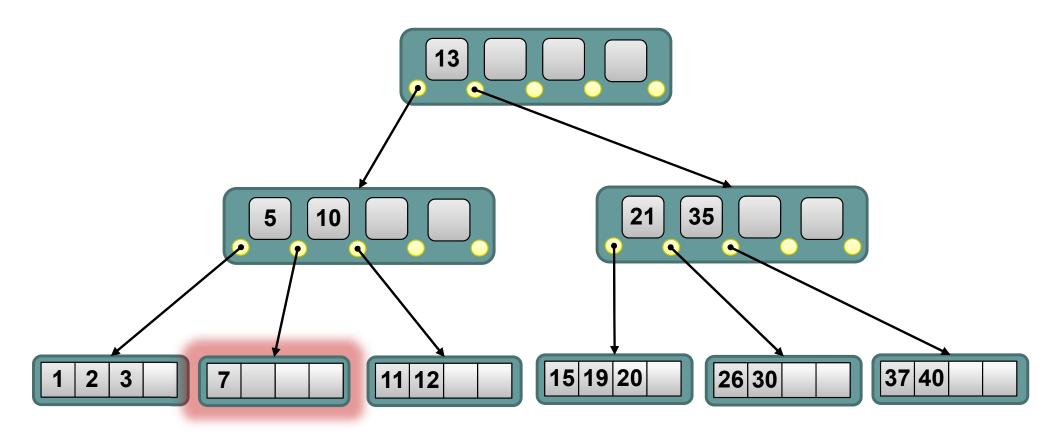




• Borrado de la clave 8. Se busca el nodo.

Borrado – Transferencia

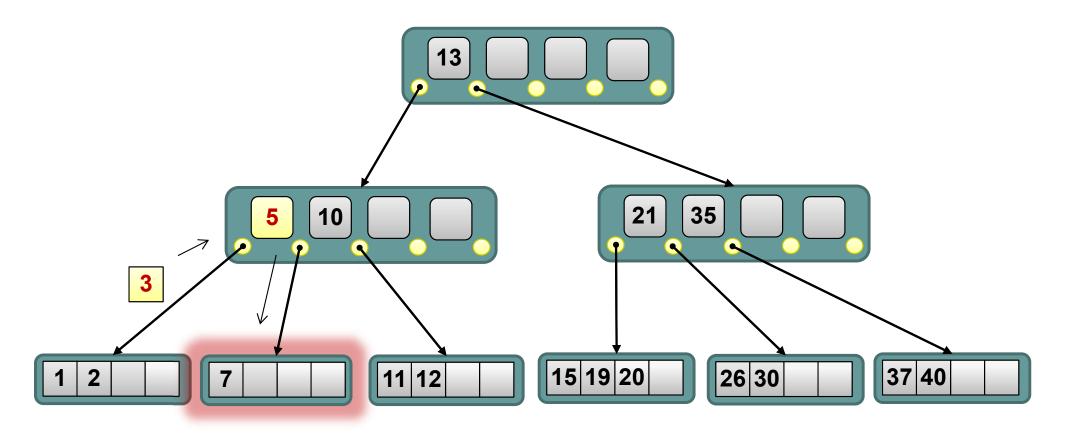




• Se borra la clave. El nodo pasa a tener menos claves que las permitidas (2).

Borrado – Transferencia

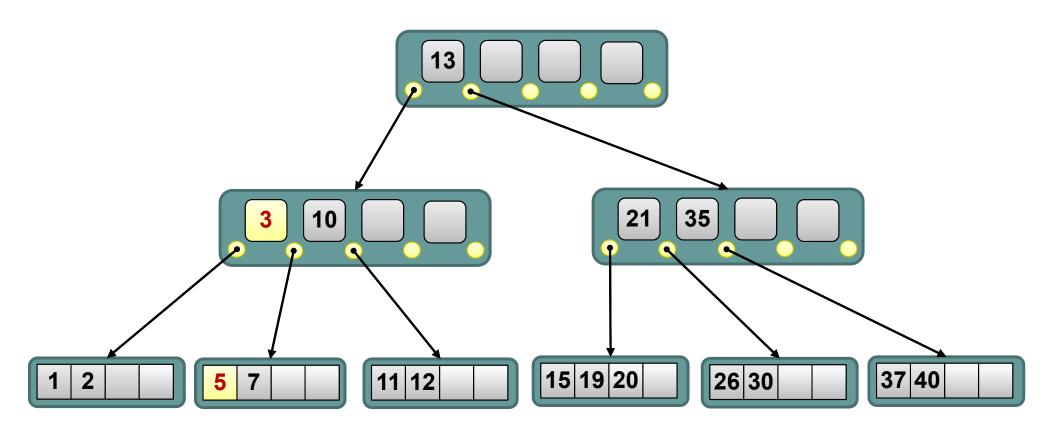




• Se comprueba el hermano con más claves (el izquierdo). Se transfiere su última clave al padre y la del padre al nodo.

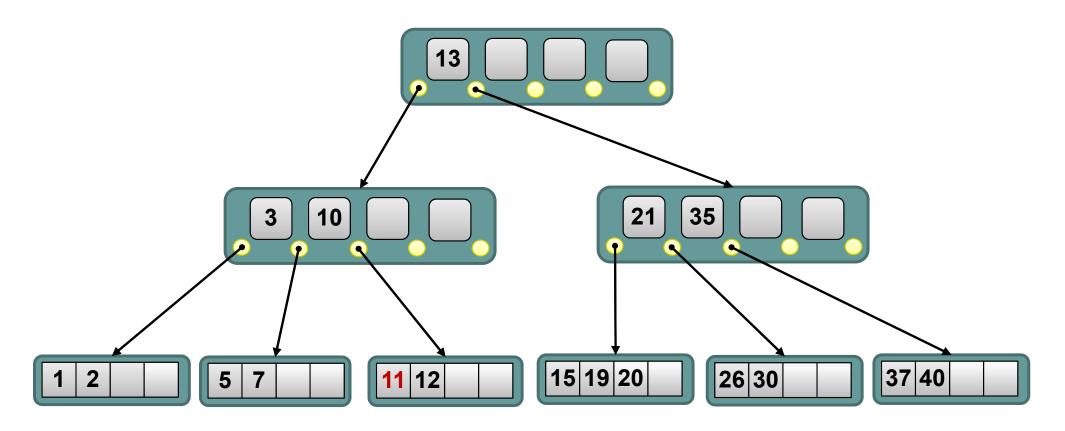






Borrado - Fusión

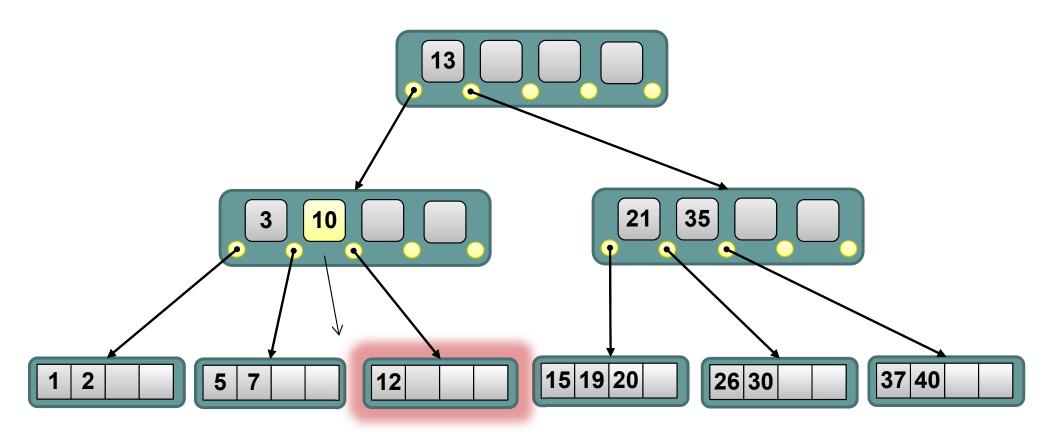




• Borrado del elemento con clave 11. Se busca el nodo.

Borrado – Fusión

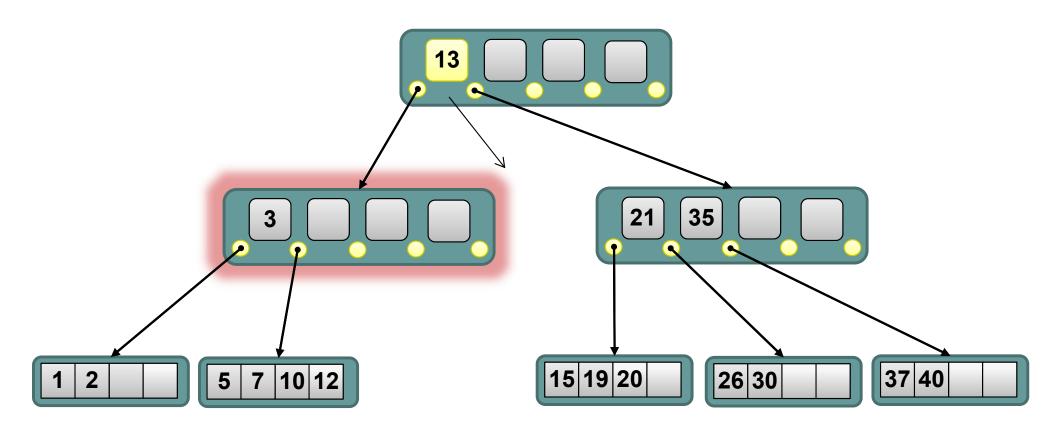




 Al borrar la clave pasa a tener menos claves que las permitidas. Su único hermano (izquierdo) no puede transferir claves.

Borrado – Fusión

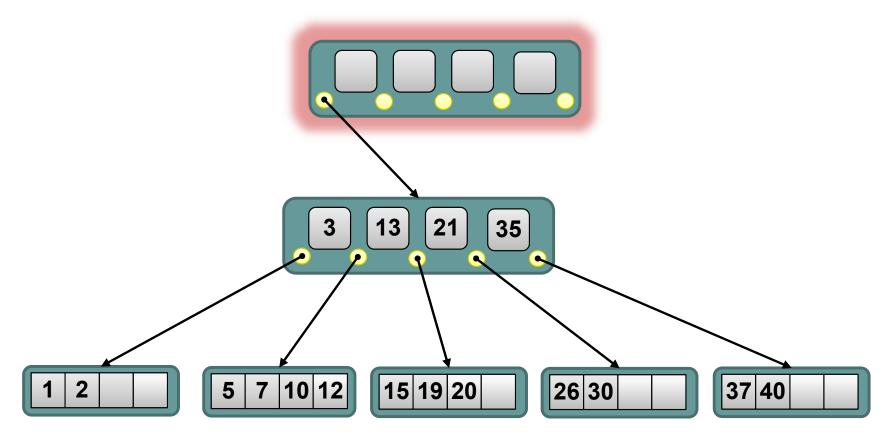




 Se fusionan el nodo con su hermano izquierdo, tomando una clave extra del padre. El padre pasa a tener una sola clave, y su hermano derecho no puede transferir.

Borrado - Fusión



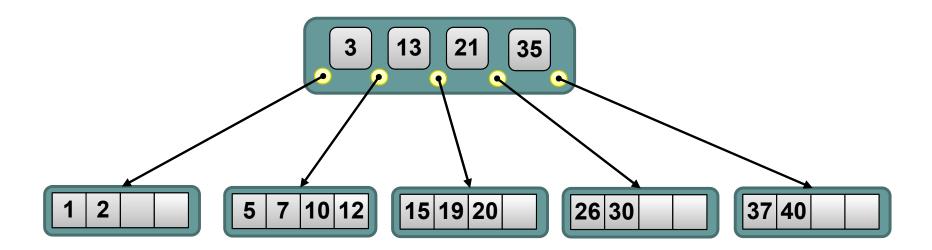


 Se fusionan los nodos, tomando la única clave del raiz, que queda vacío.

Borrado - Fusión



Se elimina el nodo raiz.



Usos y Variantes



- Los árboles B y sus variantes se usan en:
 - Gestores de Bases de Datos.
 - Sistemas de Ficheros: NTFS (Windows), HFS+ (Apple), btrfs, Ext4 (Linux)
- Variantes principales:
 - Árboles con prerecorrido: Antes de insertar se realiza una búsqueda que divide todos los nodos llenos. El número máximo de claves es 2d+1.
 - Árboles B+: Sólo las hojas contienen elementos, los nodos internos contienen claves para dirigir la búsqueda (esas claves se encuentran también en los nodos hoja). Los nodos hoja forman una lista doblemente enlazada.
 - Árboles B*: El número mínimo de claves es 2/3 de la capacidad. Se fusionan 3 nodos en 2, y se dividen 2 nodos en 3.