

Estructuras de Datos y Algoritmos

Árboles n -arios de búsqueda

LECCIÓN 16

© All wrongs reversed – bajo licencia [CC-BY-NC-SA 4.0](#)



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA



Adaptadas de diapositivas de Javier Campos

Índice

1 Definiciones

2 Ejemplo: árboles 2-3

3 Implementación de árboles 2-3

Índice

1 Definiciones

2 Ejemplo: árboles 2-3

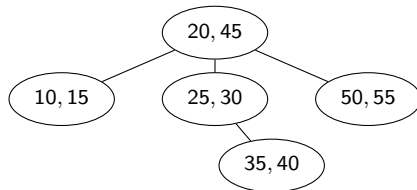
3 Implementación de árboles 2-3

Árboles n -arios de búsqueda

Definiciones

Generalización de definición de ABBs

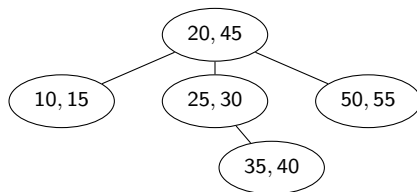
- Los árboles n -arios de búsqueda (árboles de búsqueda múltiples o multicamino) son árboles de grado n definidos de la forma:
 - Si el árbol A es vacío, entonces es un árbol n -ario de búsqueda;
 - Si el árbol A es no vacío, entonces es un árbol n -ario de búsqueda si verifica que:
 - a) La raíz de A tiene m subárboles n -arios de búsqueda $A_1, \dots, A_m$, con $1 \leq m \leq n$;
 - b) La raíz de A contiene la siguiente información: $(m, e_1, e_2, \dots, e_{m-1})$ de forma que $e_{i-1} < e_i$, para $2 \leq i \leq m-1$
 - c) Todo elemento e perteneciente al subárbol A_i es tal que $e < e_i$, para $1 \leq i \leq m-1$;
 - d) Todo elemento e perteneciente al subárbol A_m es tal que $e > e_{m-1}$



Árbol 3-ario de búsqueda

Árboles n -arios de búsqueda

Definiciones – ejemplo



Ejemplo de búsqueda de elemento e

- Mirar en primer lugar en el interior de la raíz
- Si $e = e_i$, para algún $i \in [1, m - 1]$, la búsqueda termina con éxito. Si no, determinar si:
 - $e < e_1$: buscar en el subárbol A_1
 - $e_i < e < e_{i+1}$, $i \in [1, m - 2]$: buscar en el subárbol A_{i+1}
 - $e > e_{m-1}$: buscar en el subárbol A_m
- Si m es grande, la búsqueda entre $e_1, e_2, \dots, e_{m-1}$ ha de realizarse mediante una búsqueda dicotómica. Para valores pequeños de m , una búsqueda secuencial puede ser más apropiada

Árboles n -arios de búsqueda

Definiciones

Dicho de otra forma:

- Los árboles n -arios de búsqueda se llaman también *árboles multicamino* (aquellos en los que cada nodo puede tener más de dos descendientes directos) cuyas ramas están ordenadas “a modo de” un ABB
- En un árbol multicamino, de grado n , cada nodo interno puede tener como máximo n nodos descendientes, y puede almacenar como máximo $n - 1$ claves ordenadas

Árboles n -arios de búsqueda

Definiciones

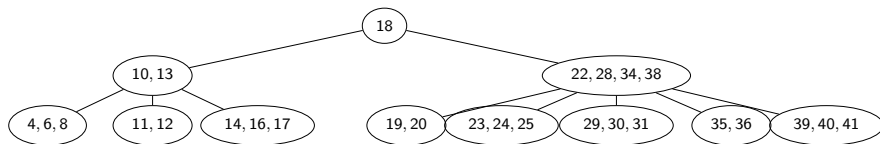
- Los árboles n -arios de búsqueda son especialmente útiles cuando se utilizan en búsquedas externas (los datos almacenados en disco, etcétera, no en memoria), permitiendo reducir así el número de accesos a disco (1 acceso por nodo consultado o nivel del árbol)
- El **coste de la operación de búsqueda es del orden de la altura del árbol**: nos interesará que estos árboles sean lo más equilibrados posible

Árboles n -arios de búsqueda

Definiciones

Árboles B

- Tipo particular de árboles n -arios de búsqueda equilibrados
- La raíz es una hoja o tiene al menos 2 hijos
- Cada nodo, excepto la raíz y las hojas, tiene al menos $\left\lceil \frac{n}{2} \right\rceil$ hijos
- Todas las hojas están en un mismo nivel

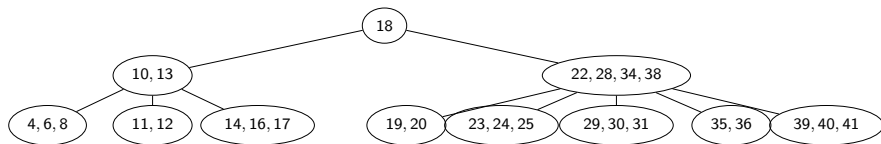


Árbol B de orden 5

Árboles n -arios de búsqueda

Definiciones

- Si hay M hojas y las hojas están en el nivel L :
 - El número de nodos en los niveles 1, 2, 3, ... es por lo menos $2, 2 \left\lceil \frac{n}{2} \right\rceil, 2 \left\lceil \frac{n}{2} \right\rceil^2, \dots$
 - Por tanto: $M \geq 2 \left\lceil \frac{n}{2} \right\rceil^{L-1}$
 - Es decir: $L \leq 1 + \log \left\lceil \frac{n}{2} \right\rceil \frac{M}{2}$
- Es decir, **la altura está acotada por el logaritmo del número de claves**



Árboles n -arios de búsqueda

Definiciones

- Búsqueda, inserción y borrado en árboles B:
 - Algoritmos descritos (por ejemplo) en el libro de la bibliografía de la asignatura de L. Joyanes et al. (capítulo 10)
- Los árboles B de orden 3 se denominan también **árboles 2-3**:
 - Cada nodo, excepto las hojas, tiene 2 o 3 hijos

Árboles n -arios de búsqueda

Definiciones

- Applet que permite visualizar paso a paso las operaciones sobre árboles B (y otros tipos de árboles)
 - <http://people.ksp.sk/~kuko/bak/>

The screenshot shows a web-based applet for visualizing B-trees. At the top, there is a navigation bar with tabs for different tree types: BST, AVL tree, B tree (selected), Red-black tree, AA tree, Skiplist, Heap, Treap, Scapegoat tree, and Splay tree.

The main area is divided into three sections:

- Display:** Shows a B-tree structure. The root node contains keys [23, 33] and has pointers to three leaf nodes: [1, 12], [32, 45, 56], and [80, 99]. The root node also has a pointer to a fourth leaf node [85].
- Text:** Displays the message "Insertion" and "We insert the key into this node.".
- Control:** Contains a text input field with the value "1", buttons for "Insert", "Find", "Delete", and "Next", a "Pause" checkbox (checked), a "Small" checkbox (unchecked), a spinner box with the value "3", and buttons for "Clear" and "Random". Below these controls, it shows the statistics: "#Nodes: 8; #Keys: 11 = 68% full; Height: 3".

Árboles n -arios de búsqueda

Definiciones

- Otra clase de árboles n -arios de búsqueda es la de los **árboles B^***
- Un **árbol B^* de orden n** (n -ario), se define como:
 - Todo nodo, excepto la raíz, tiene como mucho n hijos
 - Cada nodo de un árbol B^* de orden n , excepto la raíz y las hojas, tiene como mínimo $\left\lceil \frac{2n-1}{3} \right\rceil$ hijos
 - La raíz de un árbol B^* de orden n tiene como mínimo 2 y como máximo $2 \left\lceil \frac{2n-2}{3} \right\rceil + 1$ hijos
 - *Según la definición dada en el libro de Knuth (vol. 3, pág. 488), la raíz puede llegar a tener $4/3$ del grado*
 - Todas las hojas de un árbol B^* de orden n están en un mismo nivel
 - Un nodo interno que tenga k hijos, contendrá $k - 1$ claves

Árboles n -arios de búsqueda

Definiciones

Árboles B^*

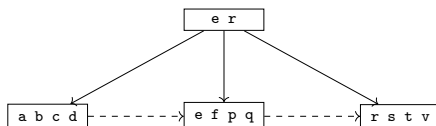
- Mejora de árboles B para aumentar el promedio de utilización de los nodos:
 - La inserción de claves en el árbol B^* supone que si el nodo que le corresponde está lleno, se mueven las claves a uno de sus hermanos
 - Así se pospone la división del nodo hasta que ambos hermanos están completamente llenos
 - Cuando finalmente se dividan, esos dos hermanos pasarán a ser 3, y cada uno estará lleno en $2/3$ partes de su capacidad

Árboles n -arios de búsqueda

Definiciones

Árboles B+

- Otra mejora de los árboles B
 - Mantienen la propiedad de acceso rápido en búsquedas para claves cualquiera,
 - Permiten un recorrido secuencial rápido
- Todas las claves se encuentran en las hojas, duplicándose en la raíz y en los nodos interiores aquellas que definen los caminos de búsqueda
 - Las claves en el nodo raíz e interiores se utilizan únicamente como índices
 - Para facilitar el recorrido secuencial rápido, las hojas están encadenadas (*enhebradas*)

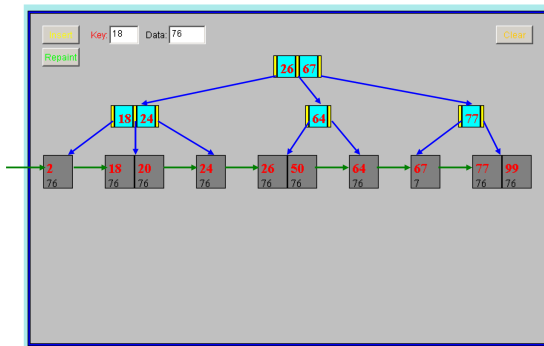


Árboles n -arios de búsqueda

Definiciones

Árboles B+

- Applet en <https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>
- Ejemplo de árbol B+ construido con dicho applet (se han añadido las flechas  )



Índice

1 Definiciones

2 Ejemplo: árboles 2-3

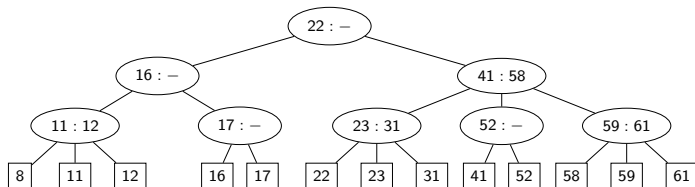
3 Implementación de árboles 2-3

Árboles n -arios de búsqueda

Ejemplo: árboles 2-3

- Árboles B de grado 3, pero “al estilo de” los B+:
 - Las claves se guardan únicamente en las hojas
 - Los nodos internos son las copias de las claves necesarias para hacer la búsqueda

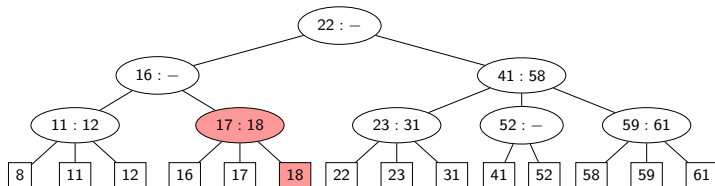
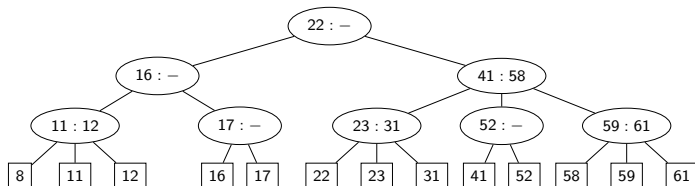
Ejemplo de inserción en árboles 2-3



Árbol 2-3 de partida

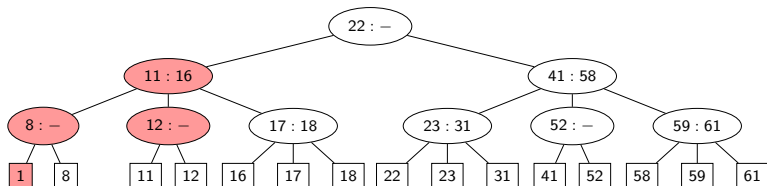
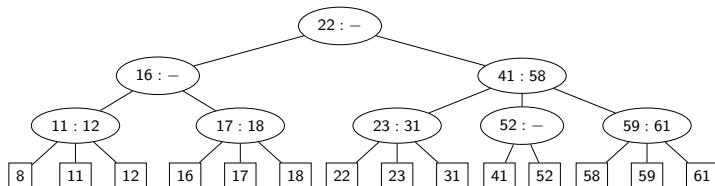
Árboles n -arios de búsqueda

Ejemplo: árboles 2-3 – Primera inserción: elemento 18



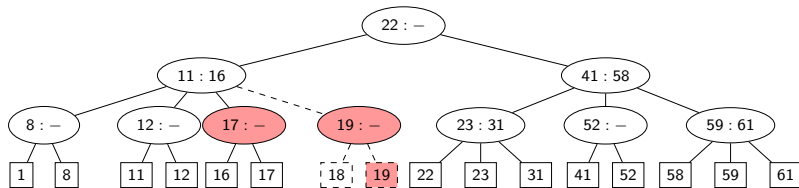
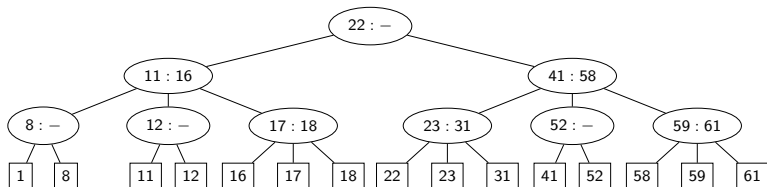
Árboles n -arios de búsqueda

Ejemplo: árboles 2-3 – Segunda inserción: elemento 1



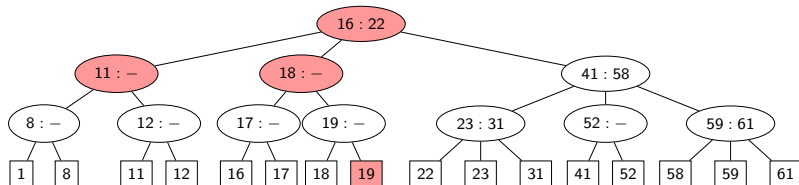
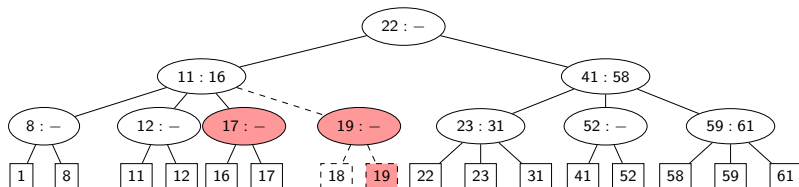
Árboles n -arios de búsqueda

Ejemplo: árboles 2-3 – Tercera inserción: elemento 19



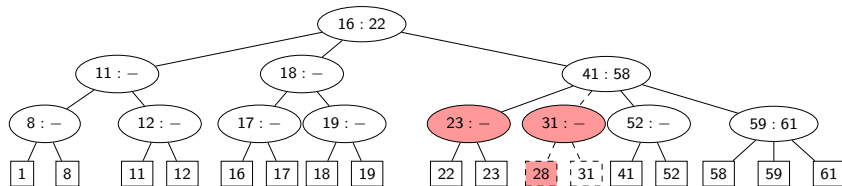
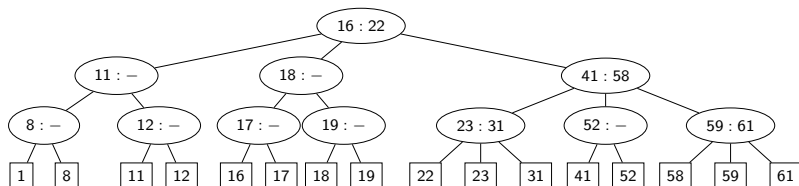
Árboles n -arios de búsqueda

Ejemplo: árboles 2-3 – Tercera inserción: elemento 19 (*continuación*)



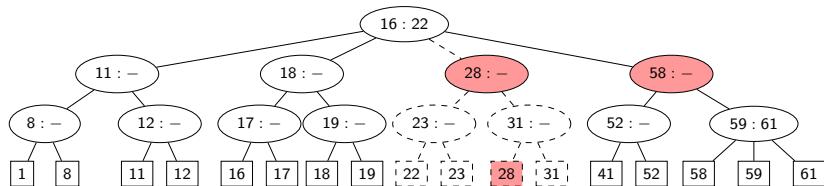
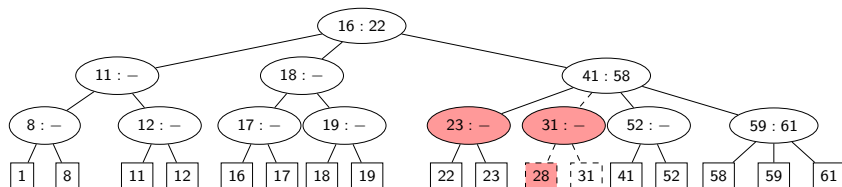
Árboles n -arios de búsqueda

Ejemplo: árboles 2-3 – Cuarta inserción: elemento 28



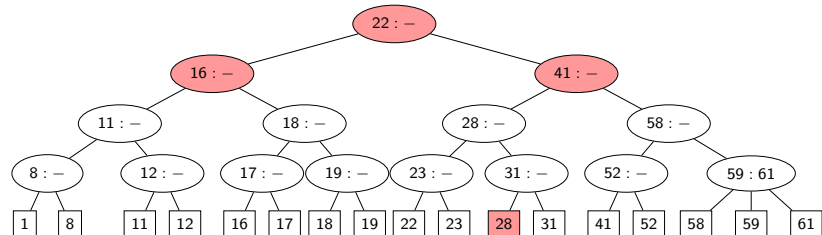
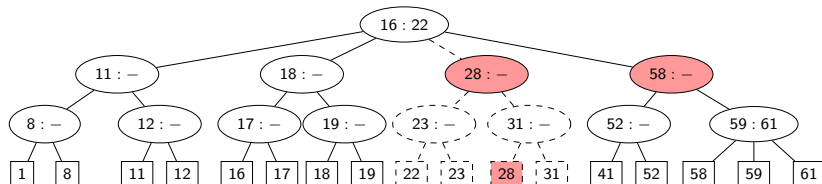
Árboles n -arios de búsqueda

Ejemplo: árboles 2-3 – Cuarta inserción: elemento 28 (*continuación*)



Árboles n -arios de búsqueda

Ejemplo: árboles 2-3 – Cuarta inserción: elemento 28 (*final*)



Índice

1 Definiciones

2 Ejemplo: árboles 2-3

3 Implementación de árboles 2-3

Implementación de árboles 2-3

Estructura de datos

```
tipos
    tipo_elmto      = registro
                    clave: tipo_clave;
                    ... {el resto de campos necesarios}
                    freg;
    tipos_nodo      = (hoja, interior);
    diccionario     = ↑nodo_dos_tres;
    nodo_dos_tres   = registro
                    clase: tipos_nodo;
                    {el siguiente campo sólo se usa si clase = hoja}
                    elmto: tipo_elmto;
                    {los siguientes campos sólo se usan si clase = interi
                    primer_hijo, segundo_hijo, tercer_hijo: diccionario;
                    menor_de_segundo, menor_de_tercero: tipo_clave
                    freg
```

Implementación de árboles 2-3

Operación inserta

procedimiento inserta(**ent** x: tipo_elmto; **e/s** S: diccionario)

variables

pt_atrás: diccionario; *{puntero al nuevo nodo devuelto por inserta1}*
menor_atrás: tipo_clave; *{valor mínimo en el subárbol de pt_atrás}*
guardaS: diccionario *{para almacenar una copia temporal de S}*

principio

{prueba si S está vacío o si hay un solo nodo, y debe incluirse un procedimiento de inserción apropiado}

...

insertaRec(S, x, pt_atrás, menor_atrás);

si pt_atrás $\neq$ nil **entonces**

{crea la raíz nueva; sus hijos están ahora apuntados por S y pt_atrás}
guardaS := S;
nuevoDato(S);
S↑.primer_hijo := guardaS;
S↑.segundo_hijo := pt_atrás;
S↑.menor_de_segundo := menor_atrás;
S↑.tercer_hijo := nil

fsi

fin

Implementación de árboles 2-3

Operación insertaRec – primera aproximación (esquema)

```
procedimiento insertaRec(e/s nodo: diccionario;  
    ent x: tipo_elmto; {x se insertará en el subárbol de nodo}  
    sal pt_nuevo: diccionario; {puntero al nodo recién creado a la derecha de nodo}  
    sal menor: tipo_clave) {elemento más pequeño del subárbol al que apunta pt_nuevo}  
principio  
    pt_nuevo := nil;  
    si nodo es una hoja entonces  
        si x no es el elemento que está en nodo entonces  
            Crea un nodo nuevo apuntado por pt_nuevo;  
            Pone x en el nodo nuevo;  
            menor := x.clave  
        fsi  
    sino {nodo es un nodo interno}  
        Sea w el hijo de nodo a cuyo subárbol pertenece x;  
        insertaRec(w, x, pt_atrás, menor_atrás);  
        si pt_atrás ≠ nil entonces  
            Inserta el puntero pt_atrás entre los hijos de nodo justo a la derecha de w;  
            si nodo tiene cuatro hijos entonces  
                Crea un nodo nuevo apuntado por pt_nuevo;  
                Da al nuevo nodo los hijos 3º y 4º de nodo;  
                Ajusta menor_de_segundo y menor_de_tercero en nodo y el nodo nuevo;  
                Coloca menor como la menor clave entre los hijos del nodo nuevo  
            fsi  
        fsi  
    fin
```

Implementación de árboles 2-3

Operación insertaRec – primera aproximación (detalles)

```
procedimiento insertaRec(e/s nodo: diccionario;  
    ent x: tipo_elmto; { x se insertará en el subárbol de nodo }  
    sal pt_nuevo: diccionario; { puntero al nodo recién creado a la derecha de nodo }  
    sal menor: tipo_clave) { elemento más pequeño del subárbol al que apunta pt_nue
```

variables

```
pt_atrás: diccionario;  
menor_atrás: tipo_clave;  
hijo: 1..3; { indica qué hijo de nodo se sigue en la llamada recursiva  
             { (relacionado con la variable w del esquema anterior) }  
w: diccionario { puntero al hijo }
```

principio

```
pt_nuevo := nil;  
si nodo↑.clase = hoja entonces  
    si nodo↑.elmto.clave↑ ≠ x.clave entonces  
        { crea una hoja nueva que contiene x.clave y "devuelve" este nodo }  
        nuevoDato(pt_nuevo);  
        pt_nuevo↑.clase := hoja;  
    si nodo↑.elmto.clave < x.clave entonces  
        { pone x en el nuevo nodo a la dcha del nodo actual }  
        pt_nuevo↑.elmto := x;  
        menor := x.clave  
    sino { x está a la izquierda del elemento en el nodo actual }  
        pt_nuevo↑.elmto := nodo↑.elmto;  
        nodo↑.elmto := x;  
        menor := pt_nuevo↑.elmto.clave  
    fsi  
    fsi  
    si { nodo es un nodo interno }
```

...

...

```
sino { nodo es un nodo interno }
  { selecciona el hijo de nodo que se debe seguir }
si x.clave < nodo↑.menor_de_segundo entonces
  hijo := 1;
  w := nodo↑.primer_hijo
sino
  si (nodo↑.tercer_hijo = nil) or (x.clave < nodo↑.menor_de_tercero) entonces
    { x está en el segundo subárbol }
    hijo := 2;
    w := nodo↑.segundo_hijo
  sino { x está en el tercer subárbol }
    hijo := 3;
    w := nodo↑.tercer_hijo
  fsi
fsi;
insertaRec(w, x, pt_atrás, menor_atrás);
si pt_atrás ≠ nil entonces
  { debe insertarse un nuevo hijo de nodo }
  si nodo↑.tercer_hijo = nil entonces { nodo tiene 2 hijos, así que
    se inserta el nuevo en el lugar adecuado }
    si hijo = 2 entonces
      nodo↑.tercer_hijo := pt_atrás;
      nodo↑.menor_de_tercero := menor_atrás
    sino {hijo=1}
      nodo↑.tercer_hijo := nodo↑.segundo_hijo;
      nodo↑.menor_de_tercero := nodo↑.menor_de_segundo;
      nodo↑.segundo_hijo := pt_atrás;
      nodo↑.menor_de_segundo := menor_atrás
    fsi
  sino { nodo ya tiene tres hijos }
```

...

...

```
sino {nodo ya tiene tres hijos}
  nuevoDato(pt_nuevo);
  pt_nuevo↑.clase := interior;
  si hijo = 3 entonces
    { pt_atrás y el tercer hijo se convierten en hijos del nuevo nodo }
    pt_nuevo↑.primer_hijo := nodo↑.tercer_hijo;
    pt_nuevo↑.segundo_hijo := pt_atrás;
    pt_nuevo↑.tercer_hijo := nil;
    pt_nuevo↑.menor_de_segundo := menor_atrás;
    {menor_de_tercero está indefinido para pt_nuevo}
    menor := nodo↑.menor_de_tercero;
    nodo↑.tercer_hijo:=nil
  sino { hijo ≤ 2; pasa el 3º hijo de nodo a pt_nuevo }
    pt_nuevo↑.segundo_hijo := nodo↑.tercer_hijo;
    pt_nuevo↑.menor_de_segundo := nodo↑.menor_de_tercer;
    pt_nuevo↑.tercer_hijo := nil;
    nodo↑.tercer_hijo := nil
  fsi;
  si hijo = 2 entonces
    {pt_atrás se convierte en el 1º hijo de pt_nuevo}
    pt_nuevo↑.primer_hijo := pt_atrás;
    menor := menor_atrás
  sino_si hijo =1 entonces
    { el segundo hijo de nodo pasa a pt_nuevo;
      pt_atrás se convierte en el 2º hijo de nodo }
    pt_nuevo↑.primer_hijo := nodo↑.segundo_hijo;
    menor := nodo↑.menor_de_segundo;
    nodo↑.segundo_hijo := pt_atrás;
    nodo↑.menor_de_segundo := menor_atrás
  fsi
fsi
fsi
fin
```

Implementación de árboles 2-3

Esquema de borrado

```
procedimiento borra(e/s d: diccionario; ent clave: tipo_clave)
variables
    unHijo: booleano;
    pAux: ptNodo;
    menor: tipo_clave
principio
    si el diccionario tiene 0 o 1 elemento entonces
        hacer el borrado de esos casos especiales
    sino { el diccionario tiene más de un elemento }
        borraRec(d.nodos, clave, unHijo, menor);
        si unHijo entonces { la raíz sólo tiene un hijo }
            eliminar la raíz y hacer que d sea el único hijo de d
        fsi
    fsi
fin
```

Implementación de árboles 2-3

Esquema de borrado

```
procedimiento borraRec(e/s p: ptNodo; ent clave: tipo_clave;
                        sal unHijo: booleano; sal menor: tipo_clave)
{ borra 'clave' de 'p'; si 'p' se queda con un solo hijo devuelve verdad en 'unHijo';
  si 'clave' es la menor de p entonces 'menor' devuelve la siguiente clave mas pequeña de p }
variables
  soloUno: booleano;
  w: ptNodo;
  hijo: natural
principio
  unHijo := falso;
  si los hijos de p son hojas entonces
    si la clave a borrar es el primer hijo de p entonces
      Se borra y se desplaza el segundo hacia la izquierda;
      Se actualiza el valor de 'menor' con el nuevo primer hijo;
      si ahora p tiene un solo hijo entonces
        unHijo := verdad
      sino
        Se desplaza el tercero hacia su izquierda
      fsi
    sino_si la clave a borrar es el segundo hijo de p entonces
      Se borra;
      si ahora p tiene un solo hijo entonces
        unHijo := verdad
      sino
        Se desplaza el tercero hacia su izquierda
      fsi
    sino_si la clave a borrar es el tercer hijo de p entonces
      ...
```

```

...
sino_si la clave a borrar es el tercer hijo de p entonces
    Se borra
fsi
sino { los hijos de p no son hojas }
    { se selecciona el hijo de p en que hay que borrar }
    si puede estar en el primer subárbol entonces
        hijo := 1;
        w := primer hijo de p
    sino_si puede estar en el segundo subárbol entonces
        hijo := 2;
        w := segundo hijo de p
    sino { puede estar en el tercer subárbol }
        hijo := 3;
        w := tercer hijo de p
    fsi;
    borraRec(w, clave, soloUno, menor);
    si soloUno entonces
        { arreglar los hijos de p para que ninguno tenga menos de dos hijos }
        si hijo = 1 entonces {el primer hijo de p sólo tiene un hijo}
            si el segundo hijo de p tiene tres hijos entonces
                Se pasa el 1º hijo del 2º de p a 2º hijo del 1º de p
            sino {el segundo hijo de p tiene dos hijos}
                Se pasa el hijo del 1º de p como 1º hijo del 2º de p
                Se elimina el primer hijo de p
                si ahora p sólo tiene un hijo entonces
                    unHijo := verdad
            fsi
        fsi
    sino_si hijo = 2 entonces {el 2º hijo de p sólo tiene un hijo}

```

```

...
sino_si hijo = 2 entonces { el 2º hijo de p sólo tiene un hijo }
    si el primer hijo de p tiene tres hijos entonces
        Se pasa el 3º hijo del 1º de p como 1º hijo del 2º de p
        si se ha borrado la clave menor del 2º hijo de p entonces
            Se actualiza con el valor 'menor' el campo adecuado
            sino {no ha cambiado la clave menor de 2º hijo de p}
                No se usa el valor 'menor'
            fsi
        sino_si el 3º hijo de p existe y tiene tres hijos entonces
            Se pasa el 1º hijo del 3º de p como 2º del 2º de p
            si se ha borrado la menor clave del 2º hijo de p entonces
                Se actualiza con el valor 'menor' el campo adecuado
            fsi
        sino { ningún otro hijo de p tiene tres hijos }
            El único hijo del 2º de p pasa a ser el 3º del 1º de p
            si se ha borrado la menor clave del 2º hijo de p entonces
                Se actualiza con el valor 'menor' el campo adecuado
            sino
                No se usa el valor 'menor'
            fsi;
            si p se ha quedado con un solo hijo entonces
                unHijo := verdad
            fsi
        fsi
    sino { hijo=3; el tercer hijo de p solo tiene un hijo }

```

...

...

```
sino { hijo = 3; el tercer hijo de p solo tiene un hijo }
  si el segundo hijo de p tiene tres hijos entonces
    Se pasa el 3º hijo del 2º de p como 1º del 3º de p
    si se ha borrado la menor clave del 3º hijo de p entonces
      Se actualiza con el valor 'menor' el campo adecuado
    sino
      no se usa el valor 'menor'
    fsi
  sino { el segundo hijo de p tiene dos hijos }
    El único hijo del 3º de p pasa como 3º del 2º de p
    si se ha borrado la menor clave del 3º hijo de p entonces
      Se actualiza con el valor 'menor' el campo adecuado
    sino
      No se usa el valor 'menor'
    fsi
  fsi
fisi
sino { soloUno = falso; todos los hijos de p tienen 2 o 3 hijos }
  Cambiar, si hace falta, los campos que guardan la menor clave del segun-
  y del tercer subárbol de p
fisi
fisi
fin
```

Árboles n -arios de búsqueda

El método en acción: buscar en un árbol 2-3

- 1 **Operación y tamaño:** buscar(a, c); sea n el número de claves y h la altura del árbol
- 2 **Caso:** peor caso (la clave está en una hoja o no está)
- 3 **Descomposición:** en cada nivel: comparar c con las claves del nodo (una o dos) y bajar al hijo que corresponda
- 4 **Coste local:** a lo sumo dos comparaciones y un descenso por nivel: $\mathcal{O}(1)$
- 5 **Composición:** un camino de la raíz a una hoja: h niveles $\Rightarrow \mathcal{O}(h)$
- 6 **Resultado:** todas las hojas están a la misma profundidad y cada nodo interno tiene ≥ 2 hijos $\Rightarrow h \leq \log_2(n+1) \Rightarrow \mathcal{O}(\log n)$ en caso peor, **garantizado** como en el AVL

Método de análisis de coste en 6 pasos (lección 8): **rutina del curso** al cerrar cada implementación

Estructuras de Datos y Algoritmos

Árboles n -arios de búsqueda

LECCIÓN 16

☹ All wrongs reversed – bajo licencia [CC-BY-NC-SA 4.0](#)



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA

