Tema 10. Árboles

`http://aulavirtual.uji.es`

José M. Badía, Begoña Martínez, Antonio Morales y José M. Badía

{badia, bmartine, morales, sanchiz}@icc.uji.es

Estructuras de datos y de la información

Universitat Jaume I

Índice

Bibliografía

- (Nyhoff '06), Capítulos 12 y 15.
- (Main, Savitch '01), Capítulos 10 y 11.
- *Fundamentals of Data Structures in C++*. E. Horowitz, S. Sahni, D. Mehta. Computer Science Press 1995. Capítulos 5, 7.6, 9, 10.1 y 10.2.
- *Estructuras de Datos, Algoritmos y Programación Orientada a Objetos*. G.L. Heileman, Mc Graw Hill 1998. Capítulos 7, 9, 11.1 y 11.2.

Objetivos

- Asimilar el concepto de árbol en general y en particular el concepto de árbol binario y la nomenclatura asociada.
- Ser capaz de implementar árboles binarios tanto con memoria estática como dinámica.
- Conocer e implementar los diferentes recorridos de árboles binarios. Ser capaz de aplicar los algoritmos de recorrido para resolver problemas.
- Asimilar el concepto de montículo y de sus aplicaciones y ser capaz de implementar los algoritmos que trabajan con montículos.
- Comprender e implementar el algoritmo de ordenación Heapsort.
- Conocer la aplicación de los árboles de Huffman y asimilar el algoritmo para construir dichos árboles.
- Asimilar el concepto y la utilidad de los árboles binarios de búsqueda.

Objetivos

- Ser capaz de implementar y utilizar un árbol binario de búsqueda.
- Trabajar con árboles binarios de búsqueda equilibrados AVL.
- Asimilar el concepto de árbol B y su aplicación. Conocer su implementación.

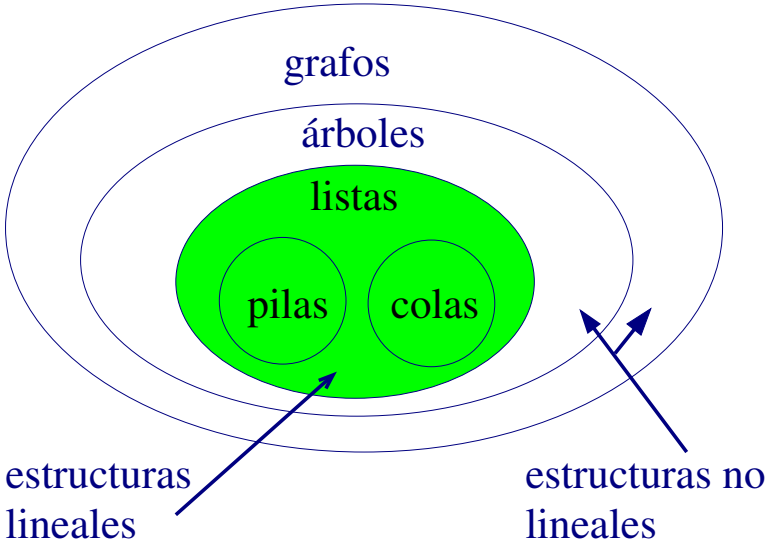
1 Introducción

Motivación:

- La gestión de grandes cantidades de datos mediante acceso lineal es ineficiente.

Los árboles son una estructura de datos **no lineal** donde los datos están organizados de forma jerárquica.

La nomenclatura utilizada en informática para los árboles está tomada de los árboles reales y de las relaciones familiares.



2 Definiciones

Definición: Un **árbol** es un conjunto finito de *nodos* y un conjunto finito de arcos dirigidos, llamados *ramas*, de modo que,

1. Existe un nodo especial llamado **raíz**.
2. Los nodos restantes se dividen en $n >= 0$ conjuntos disjuntos, $T_1 \ldots T_n$ donde cada T_i es a su vez un árbol (subárboles de la raíz).
3. Las ramas conectan pares de nodos.

- Un nodo está formado por la información que contiene y las ramas que lo unen con otros nodos.
- Cada nodo es la raíz de un árbol.
- Cualquier árbol de *n* nodos tiene *n-1* ramas.

2 Definiciones (II)

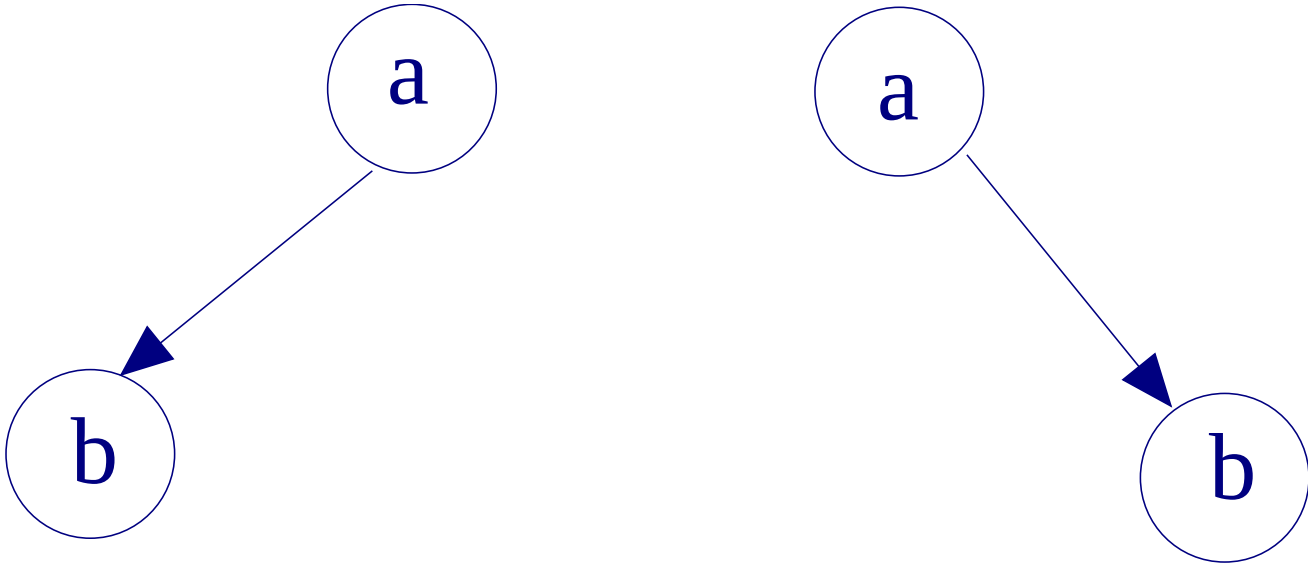
- Si de un nodo *n* hay una rama hacia otro nodo *h*, entonces se dice que *h es* **hijo** *de n* y por tanto, *n es el* **padre** *de h*.
- Cada nodo tiene un único padre, excepto la raíz que no tiene. Dos nodos son **hermanos** si tienen el mismo padre.
- Un **nodo hoja** es aquel que no tiene hijos.
- **Camino**: Un camino del nodo n_1 al nodo n_k es una secuencia de nodos $n_1, n_2, \ldots, n_{k-1}, n_k$, distintos entre sí, de modo que existe una rama que conecta cada par de nodos consecutivos (siempre en sentido descendente). Su **longitud** es el número de ramas que contiene.
- Si hay un camino del nodo n_i hasta el nodo n_k, entonces n_i es **antecesor** de n_k y, por tanto, n_k es **descendiente** de n_i. Sólo hay un camino entre la raíz y cada nodo del árbol.

2 Definiciones (III)

- **Subárbol**: cualquier nodo del árbol junto con todos sus descendientes.
- **Bosque**: colección de dos o más árboles. Al eliminar la raíz de un árbol, se tiene un bosque formado por los subárboles de la raíz.
- **Grado de un nodo**: número de hijos (subárboles) que tiene.
  - ¿Qué grado tienen los nodos hoja?
- **Grado de un árbol**: máximo de los grados de sus nodos.
- **Profundidad o nivel de un nodo**, definición recursiva:
  1. La raíz tiene profundidad 1.
  2. Si un nodo tiene profundidad n, sus hijos tienen profundidad n+1.
- **Profundidad o altura de un árbol**: máxima profundidad de sus nodos. También viene indicada por la longitud+1 del camino más largo entre la raíz y una hoja.

3 Árboles binarios

Un **árbol binario** es un conjunto finito de nodos que puede estar vacío o está compuesto por un nodo raíz y dos subárboles binarios disjuntos llamados subárbol izquierdo y subárbol derecho.



Árboles binarios distintos

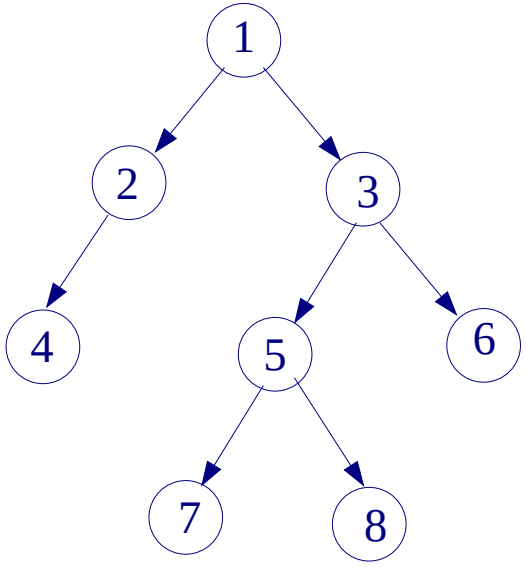
- La altura de un árbol binario vacío es 0.

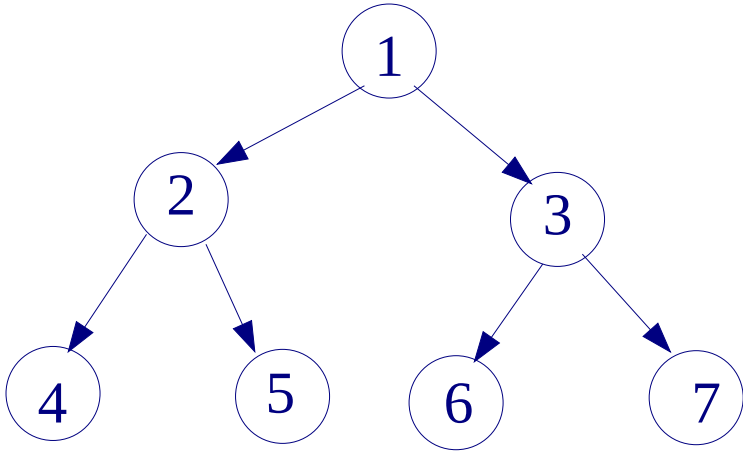
3 Árboles binarios (II)

- **Árbol binario equilibrado**: para todos sus nodos, se cumple:

$$|altura(subarbolizquierdo) - altura(subarbolderecho)| \leq 1$$

- **Árbol binario extendido**: todos sus nodos tienen 0 ó 2 hijos no vacíos.
- **Árbol binario completo de profundidad** n: todos los nodos hojas están en el nivel n, y además el número de nodos es $2^n - 1$ (número máximo de nodos para un árbol de profundidad n).





4 TAD Árbol binario

TAD Arbin

Usa Bool, tipobase

Operaciones:

//Crea un árbol binario vacío

CrearAb: → **arbin**

//Construye un nuevo árbol binario

ConstruirAb:tb x arbin x arbin → **arbin**

//Modifica un árbol binario

Modificar: tb x arbin x arbin → **arbin**

//Devuelve cierto si el árbol esta vacío, falso en caso contrario

EsVacio: arbin → **bool**

//Devuelve el árbol binario que es el hijo izquierdo

Izquierdo: arbin → **arbin**

//Devuelve el árbol binario que es el hijo derecho

Derecho: arbin → **arbin**

4 TAD Árbol binario (II)

//Devuelve el dato en la raíz del árbol

DatoRaiz: arbin → tb

Axiomas: ∀ ai, ad, aiz, ade ∈ arbin, ∀ e, f ∈ tb

1) Modificar(CrearAb)= Error
2) Modificar(ConstruirAb(e,ai,ad),f,aiz,ade)=ConstruirAb(f,aiz,ade)
3) EsVacio(CrearAb) = Verdadero
4) EsVacio(ConstruirAb(e, ai, ad)) = Falso
5) Izquierdo(CrearAb) = error
6) Izquierdo(ConstruirAb(e, ai, ad)) = ai
7) Derecho(CrearAb) = error
8) Derecho(ConstruirAb(e, ai, ad)) = ad
9) DatoRaiz(CrearAb) = error
10) DatoRaiz(ConstruirAb(e, ai, ad)) = e

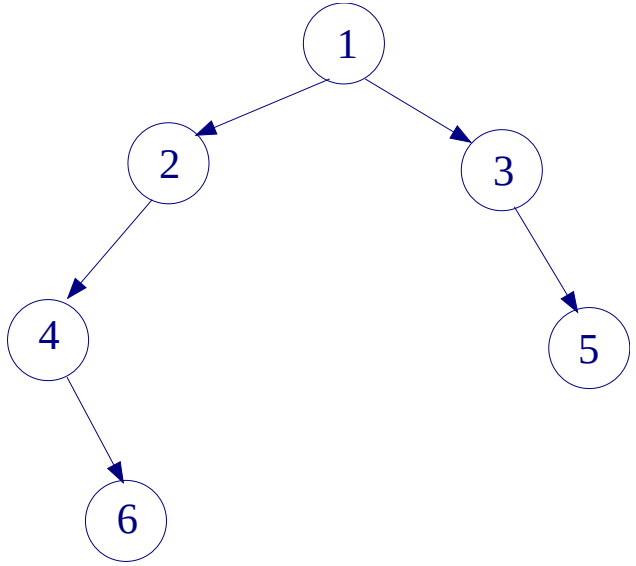
5 Representaciones estáticas

Utilizando un enlace explícito para los nodos hijos

```
template <class T>
class arbin {
private:
    struct nodo {
        T info;
        int izq, der;
    };
    nodo *elementos;
    int capacidad;
    int raiz;
public:
    //operaciones del TAD arbin
};
```

5 Representaciones estáticas (II)

	0	1	2	3	4	5	6	
info	1	2	3	4	5	6		
izq	1	3	−1	−1	−1	−1		
der	2	−1	4	5	−1	−1		



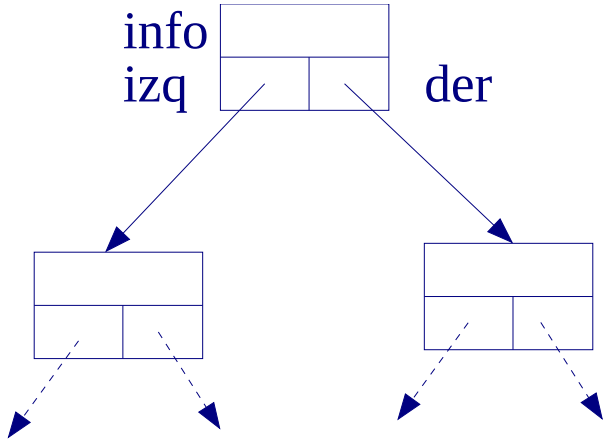
- El valor -1 se utiliza para indicar arbol vacío.
- Mantener una lista enlazada con las posiciones libres del vector.

6 Representación dinámica enlazada

```
template <class T>
class arbin {
public:
  arbin();
  arbin(const T &e, const arbin<T> &ai=arbin(),
                    const arbin<T> &ad=arbin());
  void Modificar(const T &e, const arbin<T> &ai, const arbin<T> &ad);
  void Vaciar();
  void Copiar(const arbin<T> &origen);
  arbin<T> Izquierdo() const;
  arbin<T> Derecho() const;
  const T & DatoRaiz() const;
  bool EsVacio() const;
```

6 Representación dinámica enlazada (II)

```
private:
  //nodo del arbol binario
  class nodo {
  public:
    T info;
    arbin<T> izq, der;
    nodo (const T &e=T(),
          const arbin<T> &ni=arbin(),
          const arbin<T> &nd=arbin()):
          info(e), izq(ni), der(nd) {}
  };
  typedef nodo *nodoptr;
  nodoptr raiz; };
```



7 Recorridos de los árboles binarios

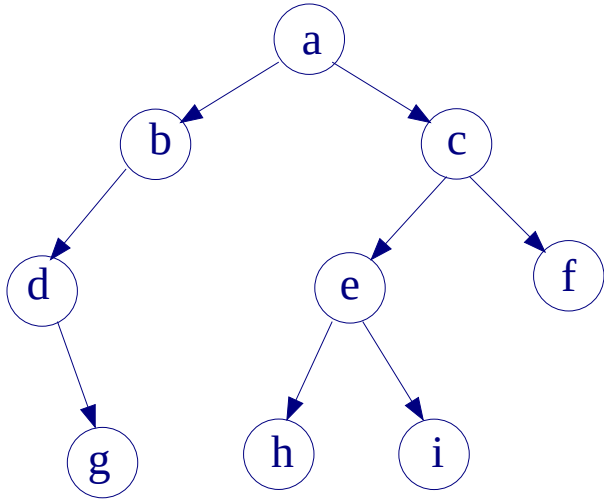
- Preorden
  1. Visitar la raíz
  2. Visitar en preorden el subárbol izquierdo
  3. Visitar en preorden el subárbol derecho
- Inorden
  1. Visitar en inorden el subárbol izquierdo
  2. Visitar la raíz
  3. Visitar en inorden el subárbol derecho

7 Recorridos de los árboles binarios (II)

- Postorden
  1. Visitar en postorden el subárbol izquierdo
  2. Visitar en postorden el subárbol derecho
  3. Visitar la raíz
- Por niveles
  - Los nodos del nivel *i* se visitan antes que los nodos del nivel *i+1*.
  - No es recursivo: se usa una cola auxiliar.

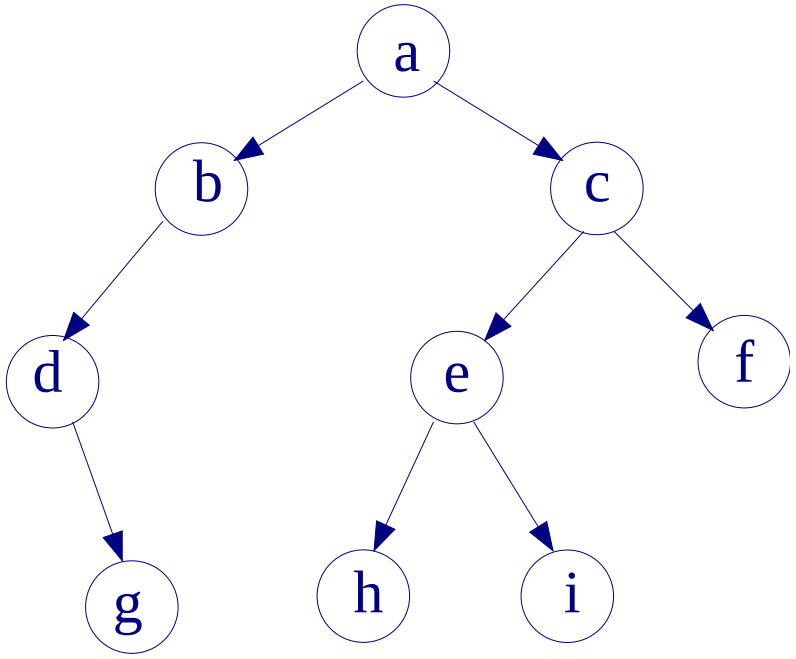
7 Recorridos de los árboles binarios (III)

```
template <class T>
void arbin<T>::Preorden() {
   if (!EsVacio()) {
      cout << DatoRaiz();
      Izquierdo().Preorden();
      Derecho().Preorden();
   }
}
```



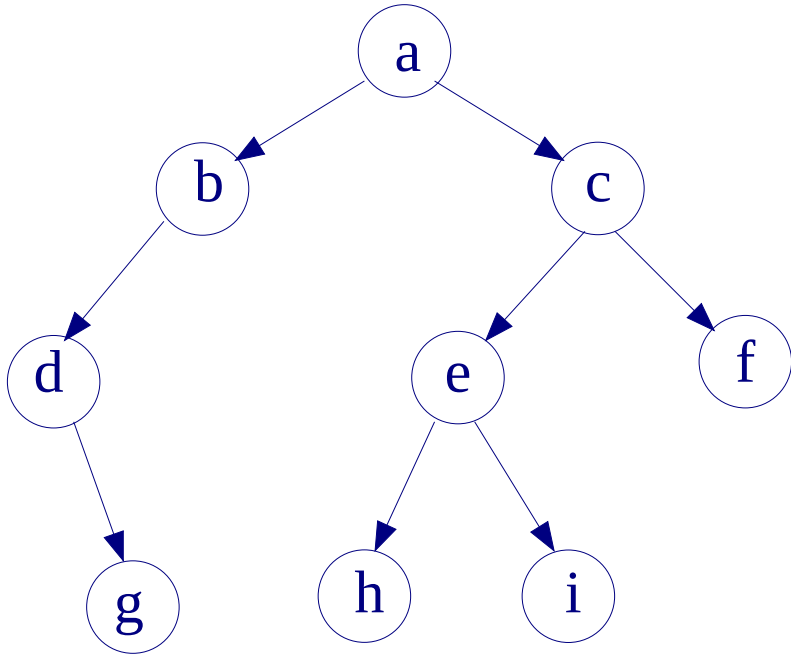
7 Recorridos de los árboles binarios (IV)

Inorden



d g b a h e i c f

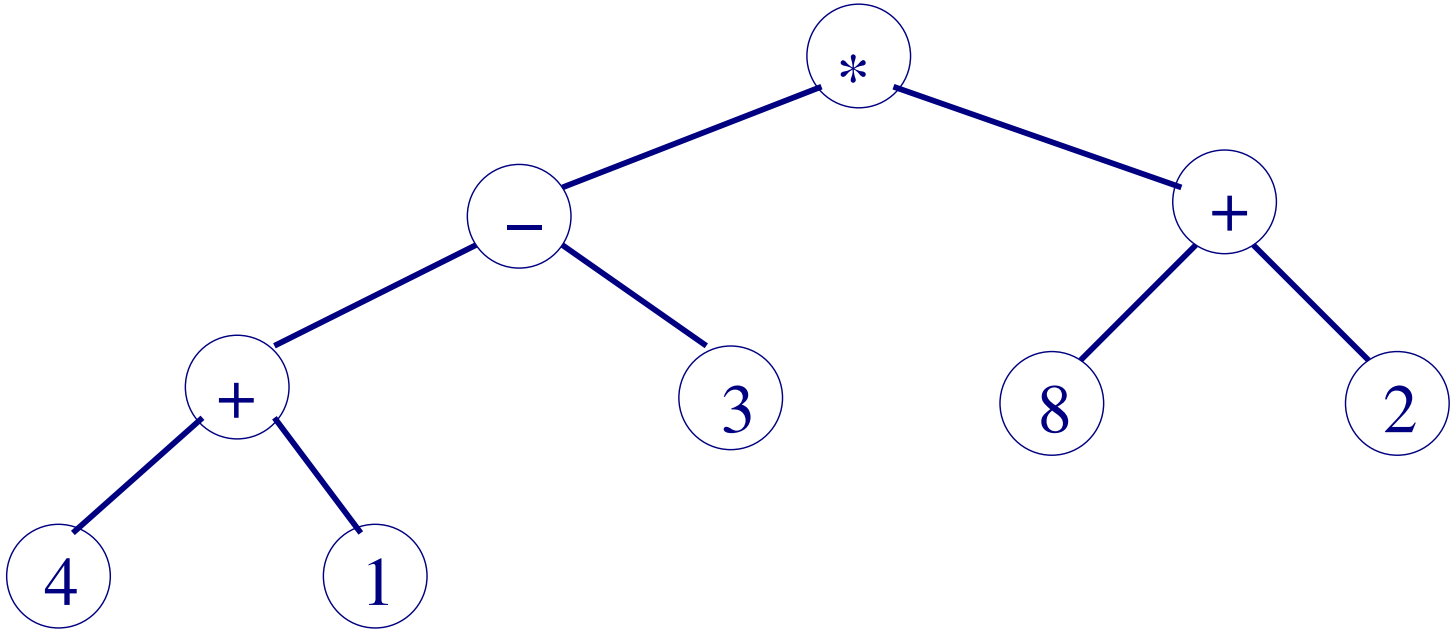
Postorden



g d b h i e f c a

7.1 Árboles de expresiones

Aplicación: Los **árboles de expresiones** representan las expresiones escritas en el programa. Los compiladores emplean estos árboles como representación intermedia entre el código fuente y el objeto.



((4+1) – 3) * (8+2)

7.1 Árboles de expresiones (II)

En un árbol de expresiones:

- Los nodos hojas contienen operandos: constantes o variables.
- Los nodos internos contienen operadores.

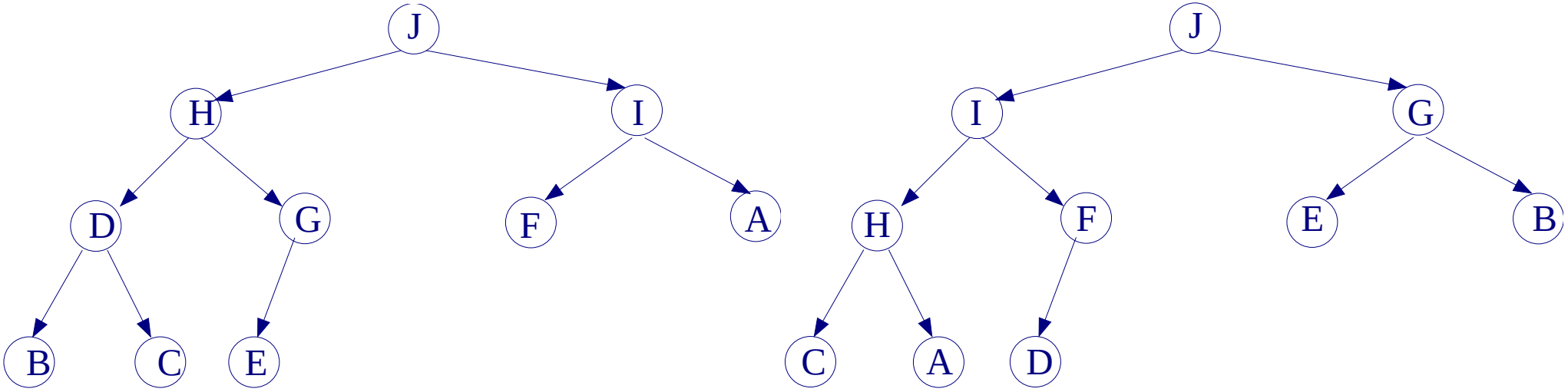
Una expresión aritmética representada mediante un árbol binario puede evaluarse mediante un recorrido en postorden. En cada nodo:

- Se evalúan su subárbol izquierdo y su subárbol derecho.
- Se aplica el operador de ese nodo a los resultados obtenidos en el paso anterior.

8 Montículos

Un **montículo** (*heap*) es un árbol binario que cumple las siguientes propiedades:

1. Es un árbol binario casi-completo
2. Para cada nodo de montículo, el valor que almacena es mayor o igual que el valor almacenado en sus nodos hijos.



8 Montículos (II)

Propiedades de los montículos:

- Para un mismo conjunto de valores, podemos construir diferentes montículos, aunque la *forma* de todos ellos siempre será la misma.
- El nodo raíz contendrá siempre el elemento mayor del conjunto.
- Todo subárbol de un montículo es también un montículo.
- Puede haber valores repetidos.

Un montículo donde el elemento mayor está en la raíz es un *max-heap*.
Es posible implementar un *min-heap*, donde el elemento menor estará en la raíz y cada nodo tendrá un valor menor o igual que el de sus hijos.

8.1 Aplicaciones y operaciones

Aplicaciones:

- Implementación de colas de prioridad.
- Ordenación de elementos.

Operaciones:

- Insertar un elemento.
- Leer el elemento mayor (o menor): elemento situado en la raíz.
- Borrar el elemento mayor (o menor).

8.1 Aplicaciones y operaciones (II)

```
template <class T>
class monticulo {
public:
  monticulo(int n=256);  //montículo vacío
  //construye un montículo a partir de un vector de n elementos
  monticulo(T vector[], int n);
  ~monticulo();
  monticulo(const monticulo<T> & origen);
  monticulo<T> & operator=(const monticulo<T> & origen);
  bool empty() const;
  void insert(const T & elem);
  const T & top() const;
  void pop();
```

8.1 Aplicaciones y operaciones (III)

```
private:
    int capacidad;
    int tam;
    T *elementos;

    //intercambia los elementos de dos posiciones
    void intercambiar (int pos1, int pos2);
    //reconstruye el montículo hacia abajo a partir de una posición
    void hundir (int pos);  //necesaria para pop
    //reconstruye el montículo hacia arriba a partir de una posición
    void subir (int pos);   //necesaria para insert
};
```

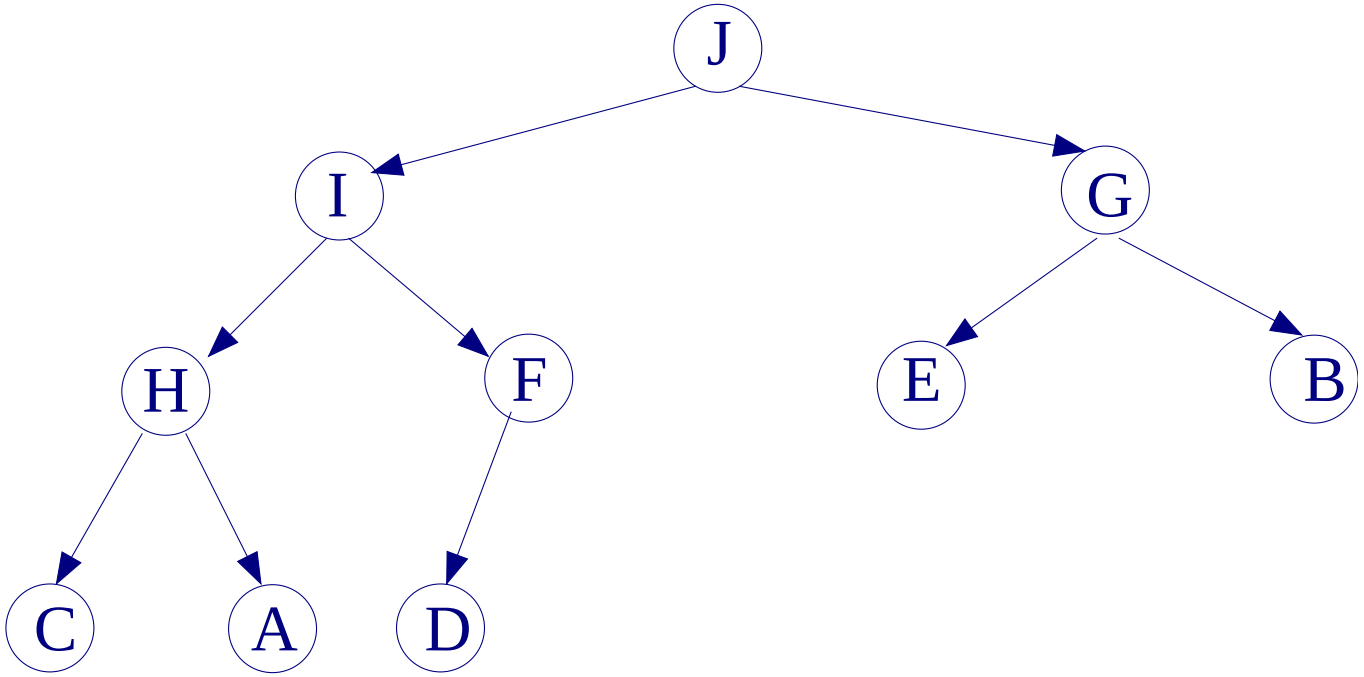
8.2 Implementación

Los montículos se implementan sobre un vector, donde los elementos están guardados de forma secuencial del siguiente modo:

1. La raíz se guarda siempre en la posición 0 del vector.
2. Para cualquier nodo interno almacenado en la posición i del vector se cumple que su hijo izquierdo está en la posición $2*i+1$ y su hijo derecho en la posición $2*i+2$ (siempre y cuando dicho nodo tenga hijos). Por tanto,
   - $elementos[2*i+1] \leq elementos[i]$ y
   - $elementos[2*i+2] \leq elementos[i]$.

Esta implementación permite conocer cuál es el nodo padre de cualquier otro nodo (excepto la raíz).

8.2 Implementación (II)



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
J	I	G	H	F	E	B	C	A	D						

capacidad = 16
tam = 10

8.3 Inserción en un montículo

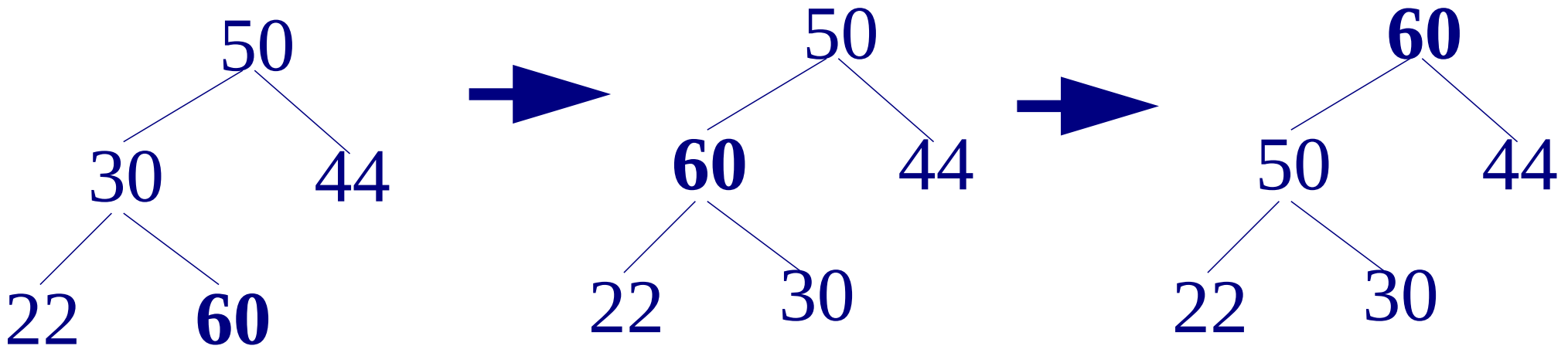
Si M es un montículo con n nodos, la inserción de un elemento e se realiza del siguiente modo:

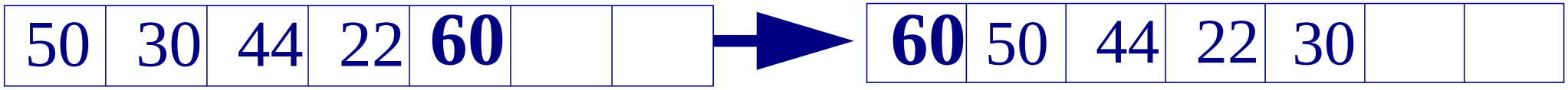
1. Se añade e en un nuevo nodo al final de $elementos$, de forma que M siga siendo un árbol binario casi-completo, aunque no necesariamente un montículo.
2. Se hace *subir* e hasta el lugar apropiado en M, para que M sea definitivamente un montículo: e *sube* mientras el valor contenido en su nodo padre sea menor que él o hasta que alcance la raíz.

Coste de insertar: $O(log_2 n)$

8.3 Inserción en un montículo (II)

Ejemplo: Insertar 60 en el siguiente montículo:





Representación: vector.

8.4 Borrado en un montículo

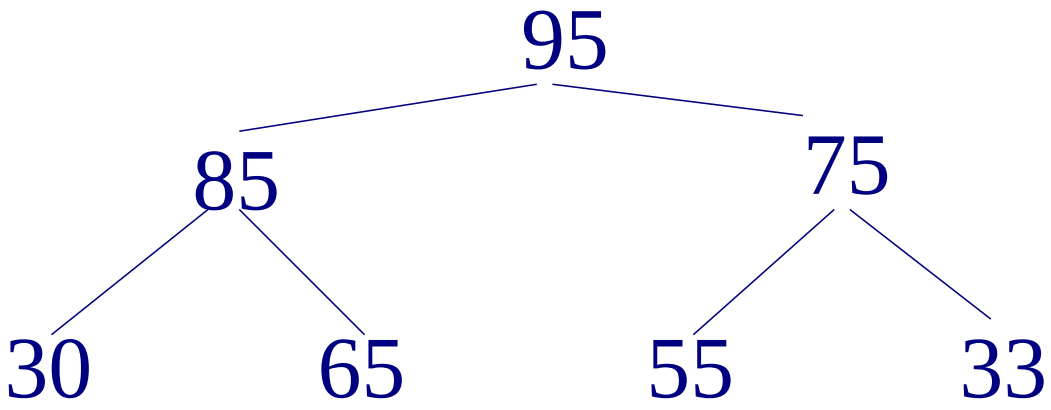
Sea M un montículo con n nodos en el que se quiere eliminar la raíz r. Los pasos son:

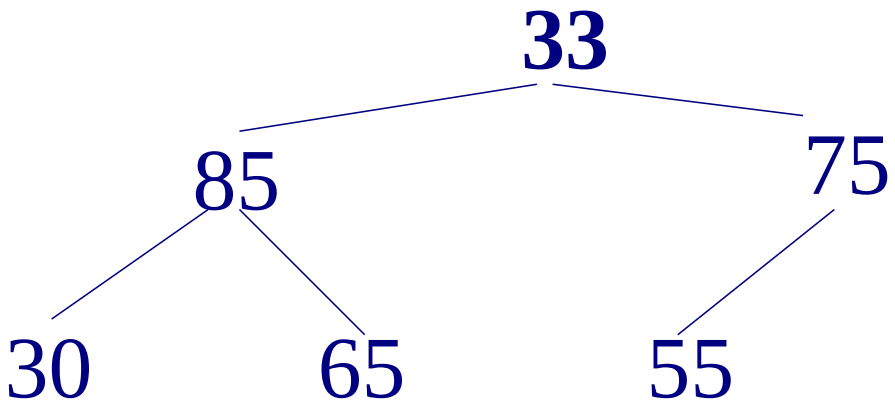
1. Reemplazar el nodo r con el último nodo u de M de forma que M siga siendo casi-completo, aunque no necesariamente un montículo.
2. Hacer que el nuevo valor de la raíz *descienda* por el árbol hasta alcanzar su sitio correcto: mientras dicho valor sea menor que sus hijos (si los tuviera) se reemplaza por el mayor de sus hijos.

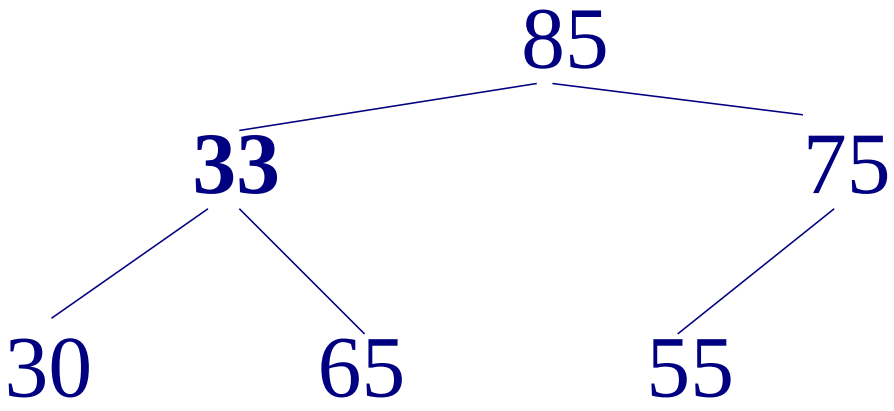
Coste de borrar la raíz: $O(log_2 n)$

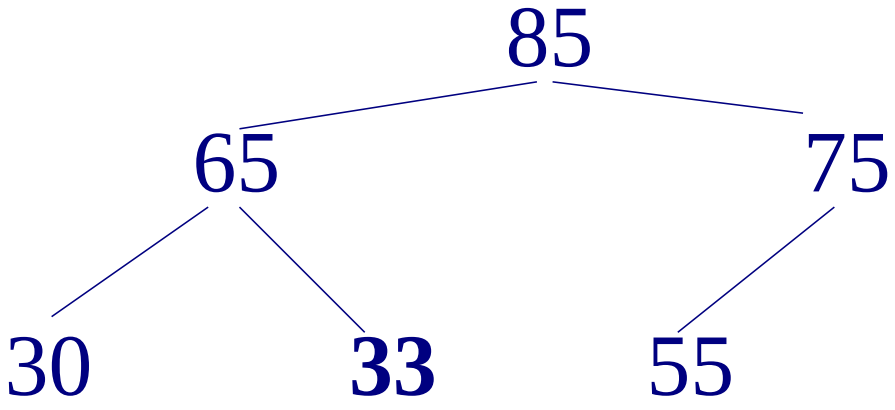
¿Cuál sería el coste de borrar cualquier elemento del montículo?

8.4 Borrado en un montículo (II)









9 Heapsort

- Heapsort es un algoritmo de ordenación de coste $O(nlog_2n)$ en el peor caso.
- Heapsort combina la eficiencia temporal de Mergesort con la eficiencia espacial de Quicksort.
- No requiere un vector adicional al vector que está ordenando.
- Se basa en un montículo.

9 Heapsort (II)

Heapsort:

1. Construir un montículo con el vector de elementos a ordenar.
2. Mientras queden elementos por ordenar:
   a) intercambiar el primer elemento del montículo (elemento mayor) con el último.
   b) reconstruir el montículo con todos los elementos menos el último, que ya está colocado en su lugar definitivo.

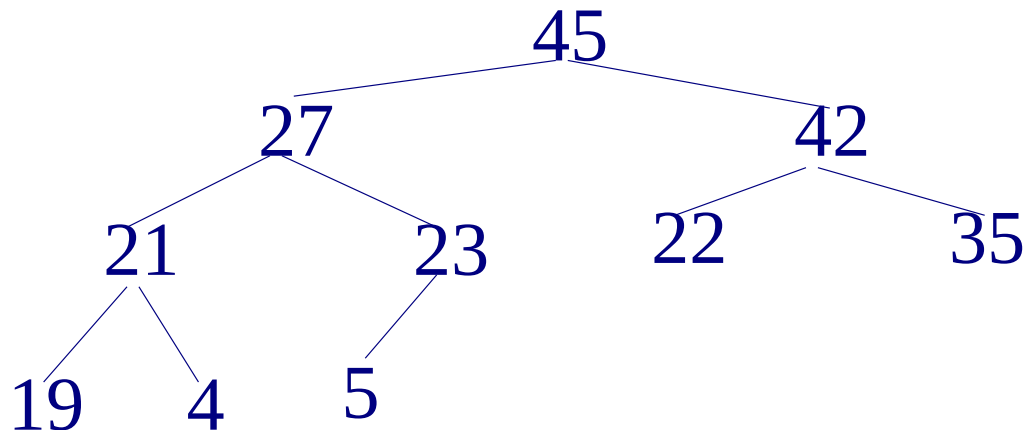
9 Heapsort (III)

Vector a ordenar:

21	23	22	4	45	42	35	19	27	5

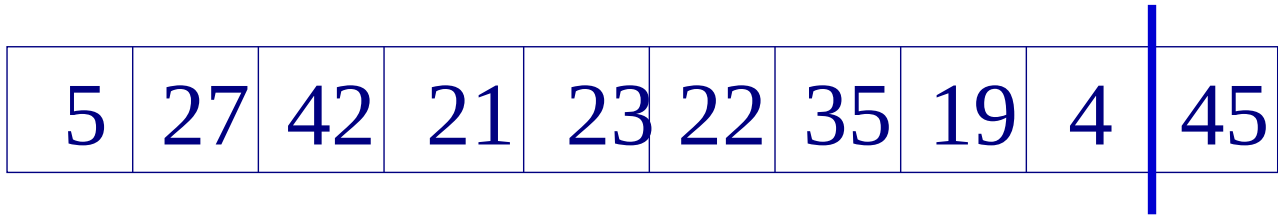
1. Construir un montículo

45	27	42	21	23	22	35	19	4	5

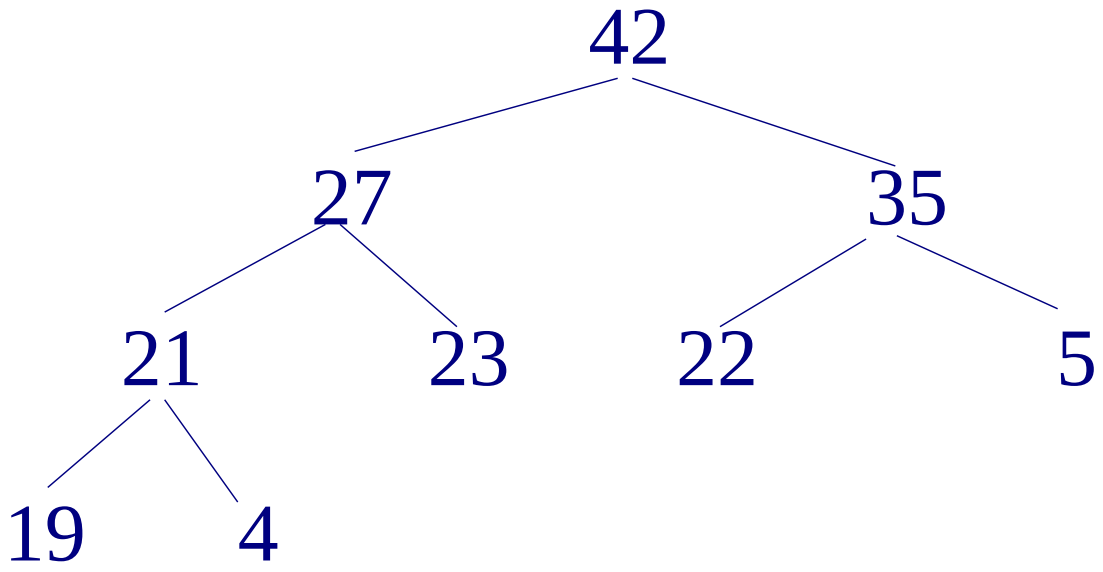


9 Heapsort (IV)

Iteración 1: a) Intercambiar primer elemento por el último

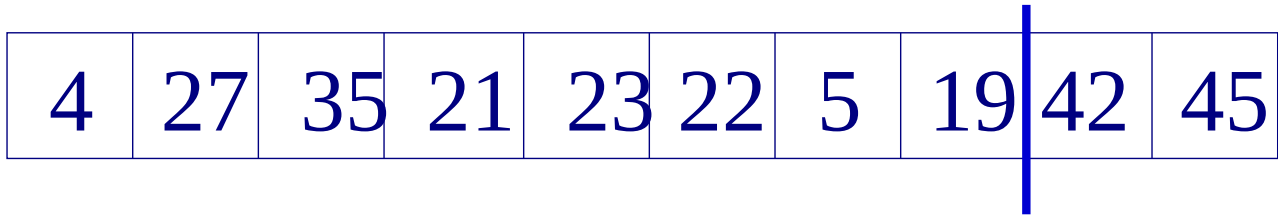


b) Reconstruir el montículo con los elementos restantes

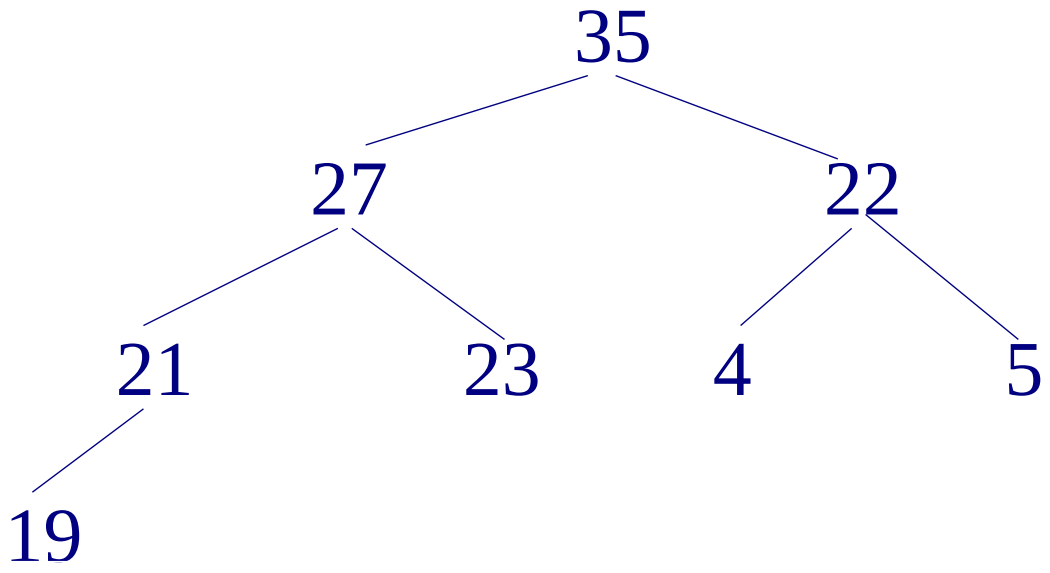


9 Heapsort (V)

Iteración 2: a) Intercambiar primer elemento por último del montículo



b) Reconstruir el montículo



9 Heapsort (VI)

```
template <class T>
void heapsort (T vect[], int n) {
    int faltan;
    FormarMonticulo(vect, n);
    faltan = n;
    while (faltan > 1) {
        --faltan;
        //intercambia los elementos de esas posiciones
        intercambia(vect[0], vect[faltan]);
        //reconstruye el montículo desde 0 hasta faltan
        hundirElem(vect,0,faltan);
    }
}
```

9 Heapsort (VI)

```
template <class T>
void FormarMonticulo (T vect[], int n) {
    //desde el primer nodo no hoja
    for (int i=(n/2)-1; i>=0; i--)
        hundirElem(vect, i, n);
 }
```

10 Árboles de Huffman

Para representar n caracteres en binario se necesitarían r bits tal que $2^{r-1} < n \leq 2^r$.
Por ejemplo, el código ASCII utiliza 7 bits para codificar 128 caracteres.

En general, utilizando códigos de bits de longitud variable, de manera que los caracteres más frecuentes tengan códigos más cortos mientras que los menos frecuentes tengan códigos más largos, se puede reducir el tamaño de los ficheros.

El algoritmo de Huffman construye un código binario para un conjunto de caracteres:

- La longitud de los códigos es la mínima.
- Cada carácter puede ser unívocamente decodificado.

10 Árboles de Huffman (II)

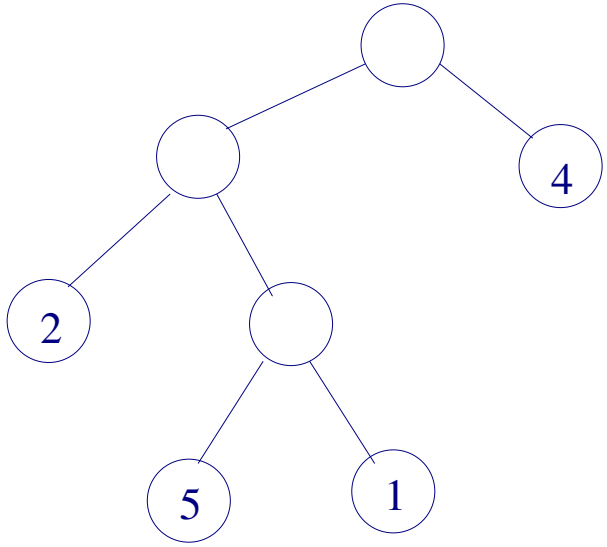
Sea un árbol extendido con n nodos hoja donde a cada uno se le ha asignado un peso w_i.

La **longitud externa de caminos** con peso se define como la suma de la longitud de cada camino desde la raíz a un nodo hoja por su peso:

$p = w_1 L_1 + w_2 L_2 + \ldots + w_n L_n$

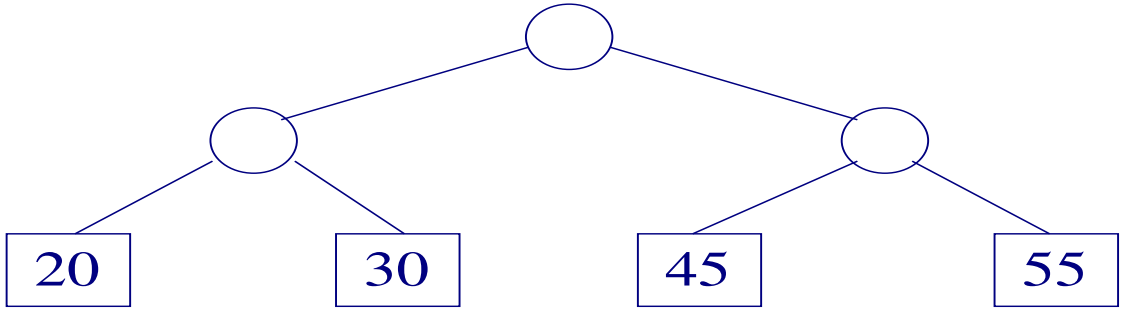
donde w_i y L_i denotan respectivamente, el peso del nodo i y la longitud del camino hasta el nodo hoja i.

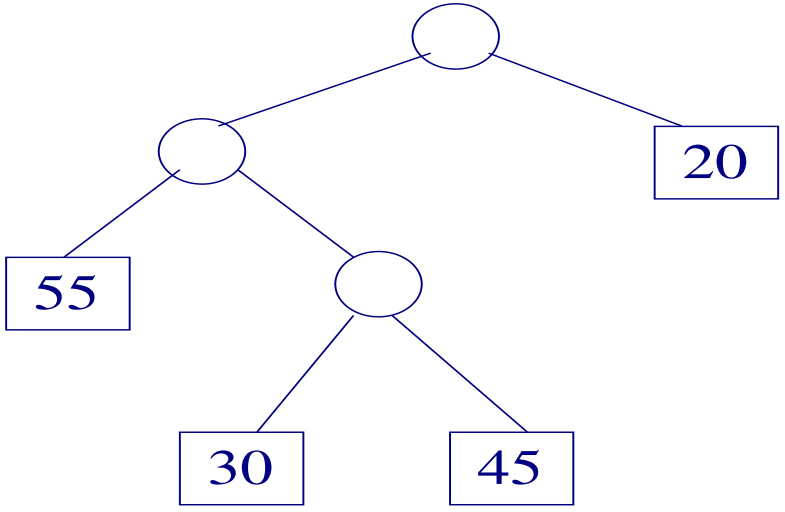
Ejemplo: $p = 2 * 2 + 5 * 3 + 1 * 3 + 4 * 1 = 26$

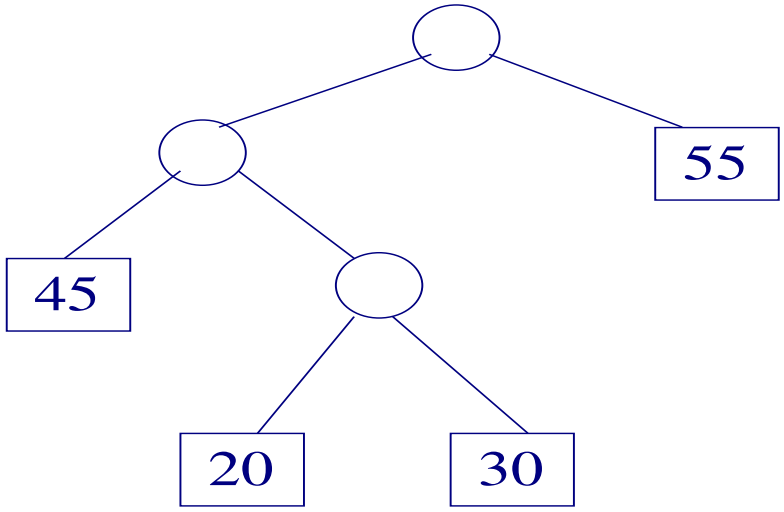


10 Árboles de Huffman (III)

Si se consideran todos los árboles binarios extendidos con n nodos hoja con iguales pesos asignados, el algoritmo de Huffman construye el árbol cuya longitud externa de caminos es mínima.







10 Árboles de Huffman (IV)

Algoritmo de Huffman aplicado a la codificación de caracteres:

- **Entrada**: Conjunto de n caracteres $C_1, C_2, \ldots, C_n$ y conjunto de frecuencias de ocurrencia de cada carácter $w_1, w_2, \ldots, w_n$, donde w_i es la frecuencia del carácter C_i.
- **Salida**: n cadenas de unos y ceros representando los códigos para cada uno de los caracteres de entrada.
- **Codificación**: las ramas a la izquierda se codifican con 0 y las ramas a la derecha con 1. El camino de la raíz al nodo externo da lugar a la cadena que lo codifica.

10 Árboles de Huffman (V)

Algoritmo:

Entrada: conjunto de n caracteres con una frecuencia asociada cada una.

1. Inicialmente cada carácter es un árbol binario con un sólo nodo (bosque).
2. Mientras número de árboles sea mayor que 1
   a) Buscar los dos árboles, T' y T'', cuyas raíces tengan la menor frecuencia asociada, w' y w''.
   b) Reemplazar estos dos árboles por un nuevo árbol cuya raíz será $w' + w''$ y cuyos subárboles serán T' y T''.

Se obtiene un árbol binario extendido.

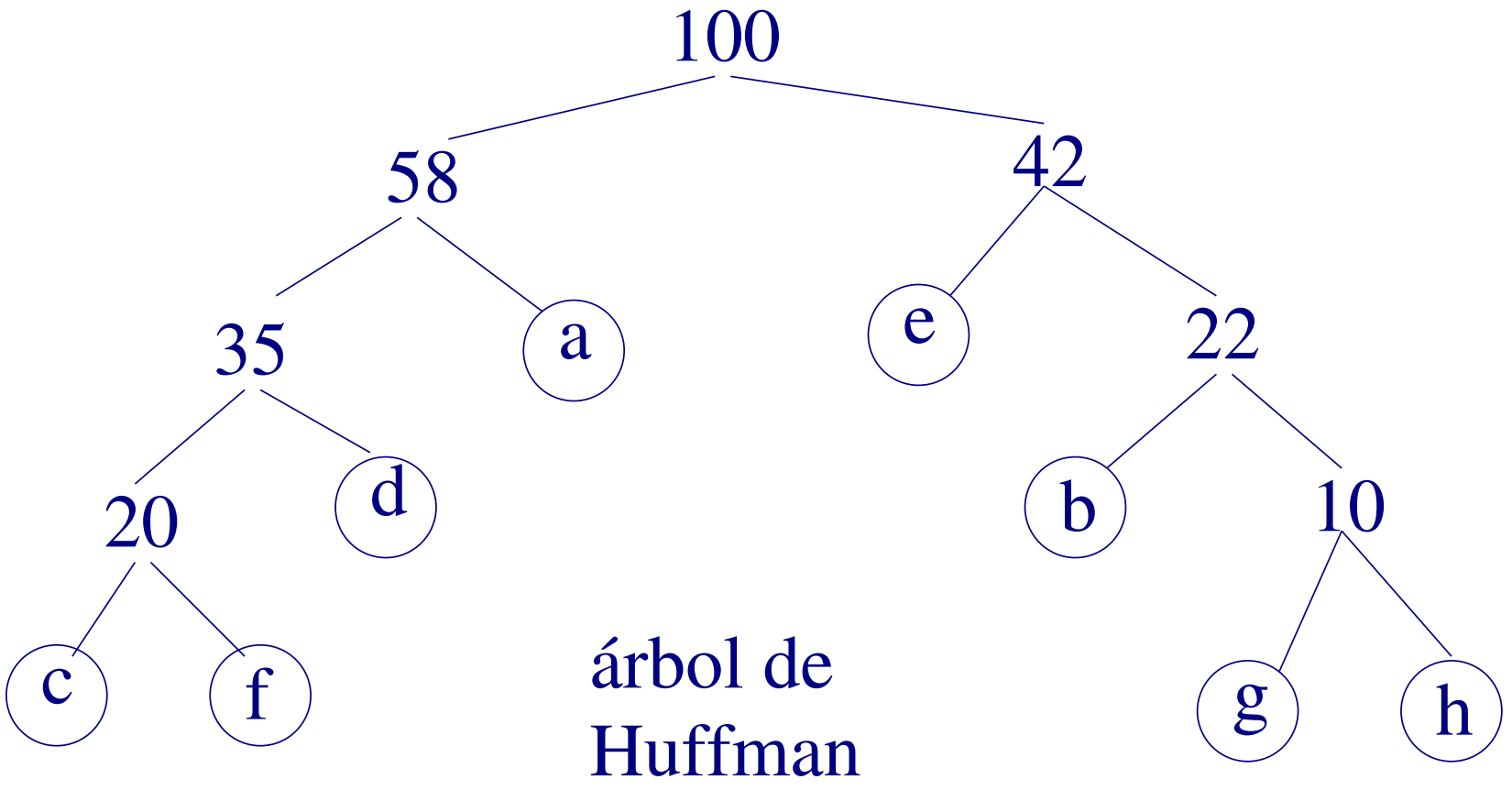
Implementación del bosque: montículo ordenado por la frecuencia, de menor a mayor.

Coste del algoritmo: $O(nlog_2 n)$

10 Árboles de Huffman (VI)

entrada:

a	b	c	d	e	f	g	h
23	12	10	15	20	10	5	5



árbol de Huffman

10 Árboles de Huffman (VII)

- Cada carácter tiene un código único
- Dada una cadena binaria se puede decodificar unívocamente

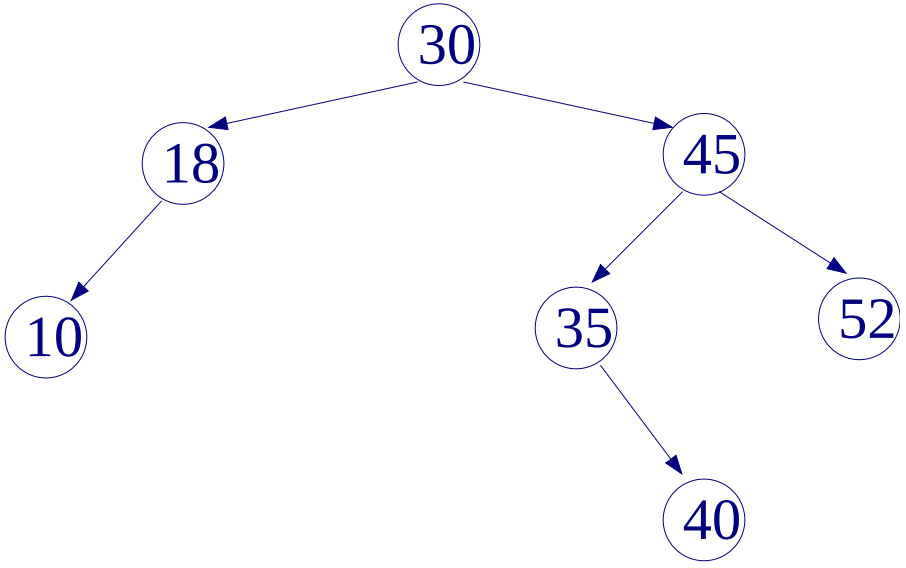
Ejercicios:

- Dado un árbol de Huffman y un carácter, devolver la cadena binaria que codifica dicho carácter
- Dada una cadena binaria y el correspondiente árbol de Huffman, devolver la cadena de caracteres que han sido codificados.

11 Árboles binarios de búsqueda

Un **árbol binario de búsqueda** (*ABB*) es un árbol binario tal que para cada nodo N se cumple:

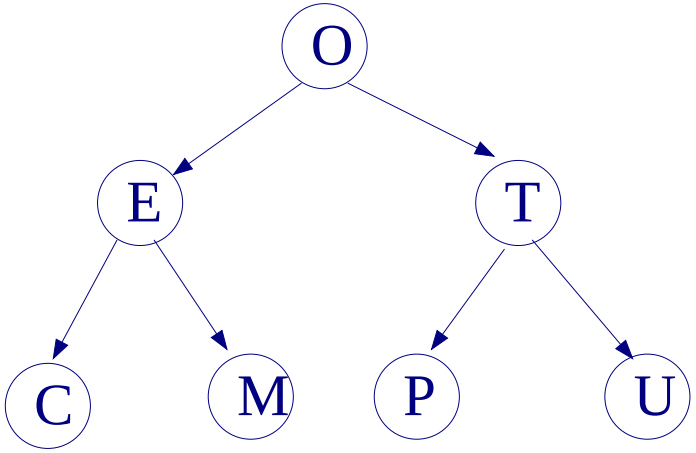
1. Si L es un nodo en el subárbol izquierdo de N, entonces el valor contenido en L es menor que el de N.
2. Si R es un nodo en el subárbol derecho de N, entonces el valor contenido en R es mayor que el de N.



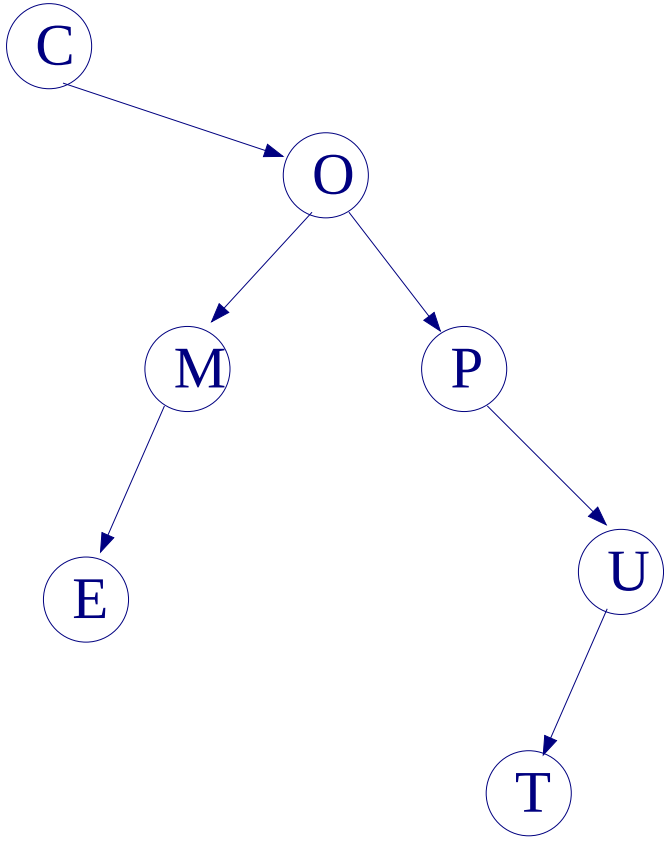
11 Árboles binarios de búsqueda (II)

- La forma de un ABB depende del orden de inserción de los elementos.

O, E, T, C, U, M, P



C, O, M, P, U, T, E

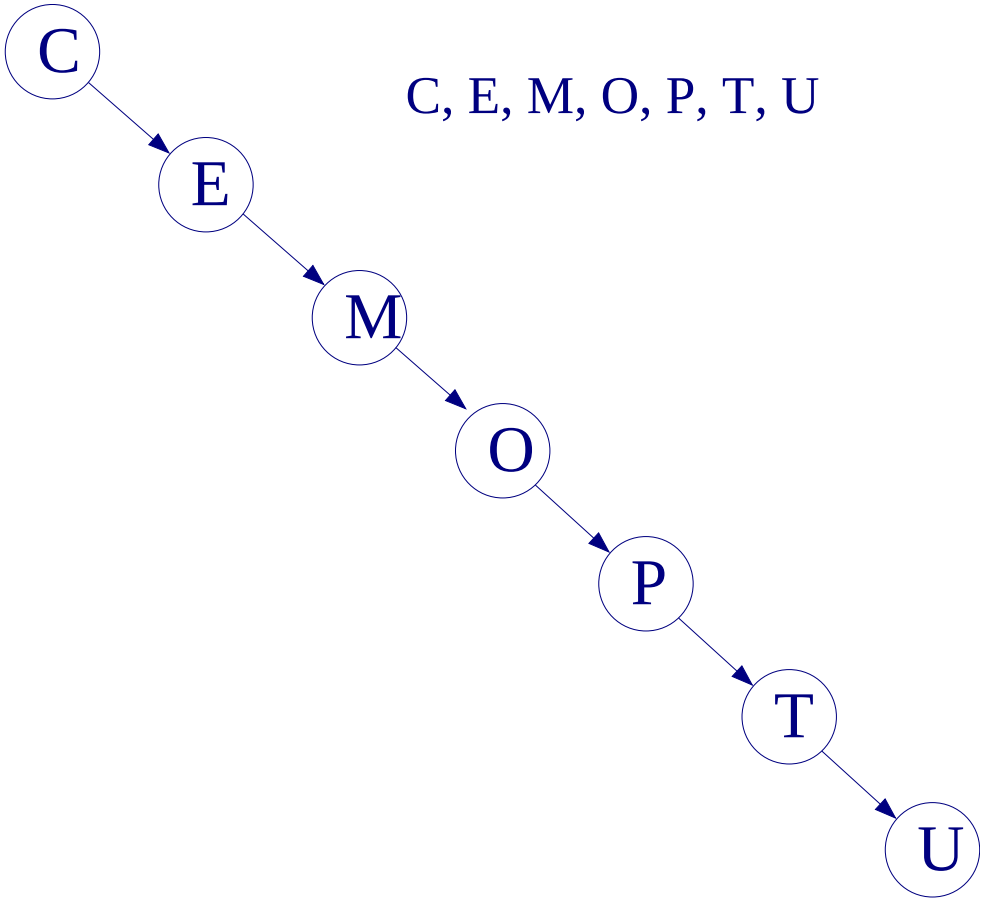


- El recorrido en inorden de un ABB devuelve la secuencia ordenada de valores en el árbol.

11 Árboles binarios de búsqueda (III)

El coste de las operaciones es $O(log_2 n)$ si el árbol es equilibrado.

Los elementos insertados en orden hacen que el árbol degenere en una lista.



11.1 TAD ABB

TAD abb

Usa bool, tb, arbin *//operaciones: ArbolVacio, CrearAb, ConstruirAb*

Operaciones:

//Inserta un nuevo elemento en el abb

insertar: abb x tb → abb

//Cierto si el abb contiene el elemento y falso en caso contrario

buscar: abb x tb → bool

//Borra un elemento dado del abb

borrar: abb x tb → abb

Axiomas: ∀ ai, ad ∈ abb, ∀ e, f ∈ tb

1) insertar(CrearAb, e) = ConstruirAb(e, CrearAb, CrearAb)
2) insertar(ConstruirAb(f, ai, ad), e) = Si e=f entonces error
   sino Si e<f entonces ConstruirAb(f, insertar(ai, e), ad)
   sino ConstruirAb(f, ai, insertar(ad, e))

11.1 TAD ABB (II)

```
3) buscar(CrearAb, e) = false
4) buscar(ConstruirAb(f, ai, ad), e) =
      Si e=f entonces true
      sino Si e<f entonces buscar(ai, e)
      sino buscar(ad, e)
5) borrar(CrearAb, e) = error
6) borrar(ConstruirAb(f, ai, ad), e) =
      Si e<f entonces ConstruirAb(f, borrar(ai, e), ad)
      sino Si e>f entonces ConstruirAb(f, ai, borrar(ad, e))
      sino Si ai=CrearAb y ad=CrearAb entonces CrearAb
      sino Si ai=CrearAb entonces ad
      sino Si ad=CrearAb entonces ai
      sino ConstruirAb(minimo(ad), ai, borrar(ad,minimo(ad)))
```

¿Cómo se definirá la operación **minimo** y cuáles serán sus axiomas?

11.1 TAD ABB (III)

```
template <class T>
class ABB{
private:
  class nodo{
  public:
     T info;
     nodo *izq, *der;
     nodo(const T& item=T(), nodo* iz=NULL, nodo * de=NULL);
  };
  typedef nodo * nodoptr;
  nodoptr raiz;
  void vaciar (nodoptr r);
  nodoptr copiar (nodoptr r);
```

11.1 TAD ABB (IV)

```
    bool search(nodoptr r, const T & item) const;
    void insert(nodoptr & r, const T & item);
    void erase(nodoptr & r, const T & item);
    void erasesuc(nodoptr & r, T & elem);
  public:
    ABB();
    ABB(const ABB<T> & origen);
    ~ABB();
    ABB<T> & operator=(const ABB<T> & origen);
    bool empty() const;
    bool search(const T & item) const;
    void insert(const T & item);
    void erase(const T & item);
};
```

11.2 Operaciones básicas

Operación **insertar** (la operación **buscar** se deja como ejercicio)

```
template <class T>
void ABB<T>::insert(const T & item) {
    insert(raiz, item);
}
template <class T>
void ABB<T>::insert (nodoptr & r, const T & item) {
  if (r == NULL) r=new nodo(item);
  else if (item != r->info)
          if (item < r->info)
             insert (r->izq, item);
          else
             insert (r->der, item);
}
```

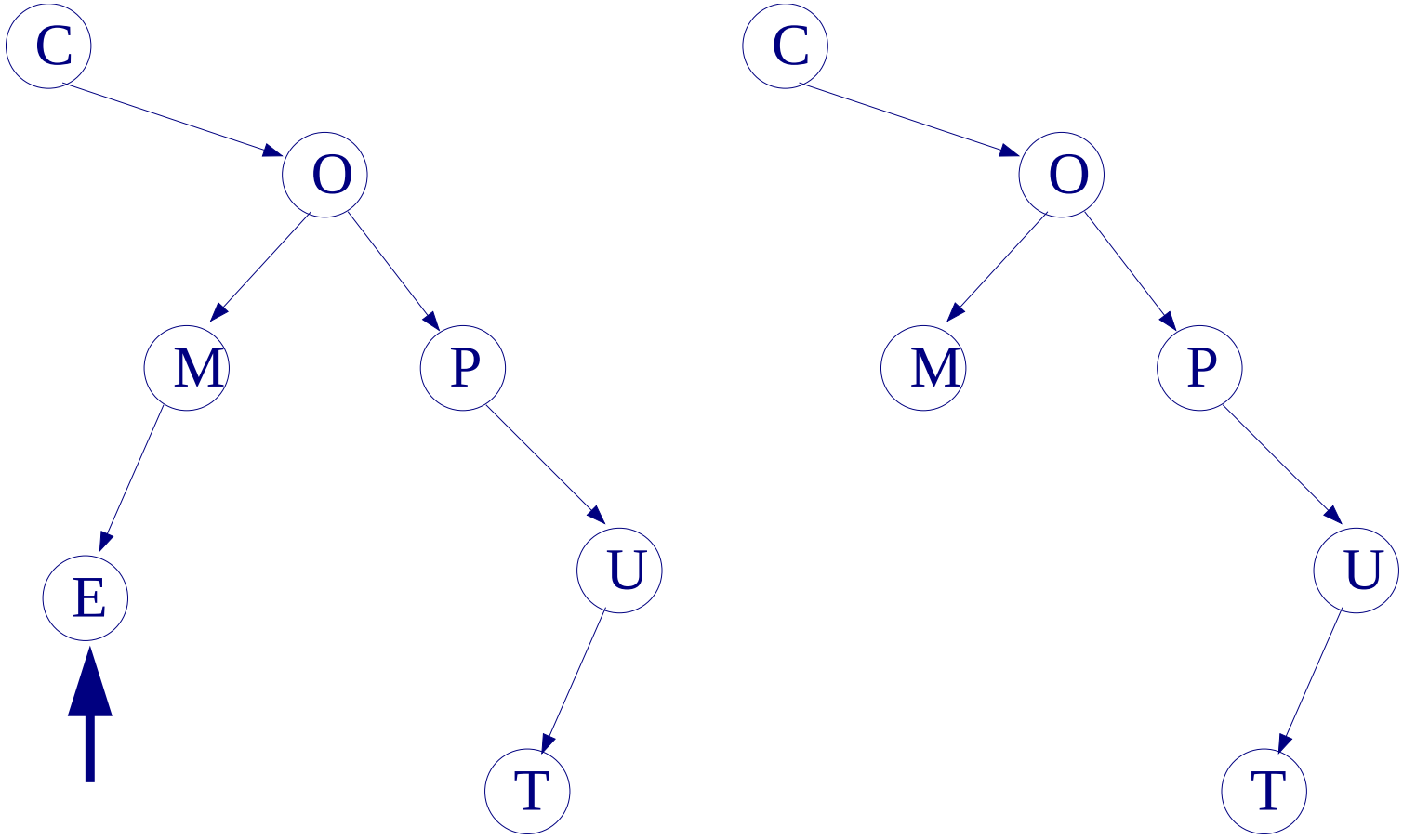
11.2 Operaciones básicas (II)

Borrar un elemento de un ABB. Tres casos según el nodo donde está el elemento a borrar:

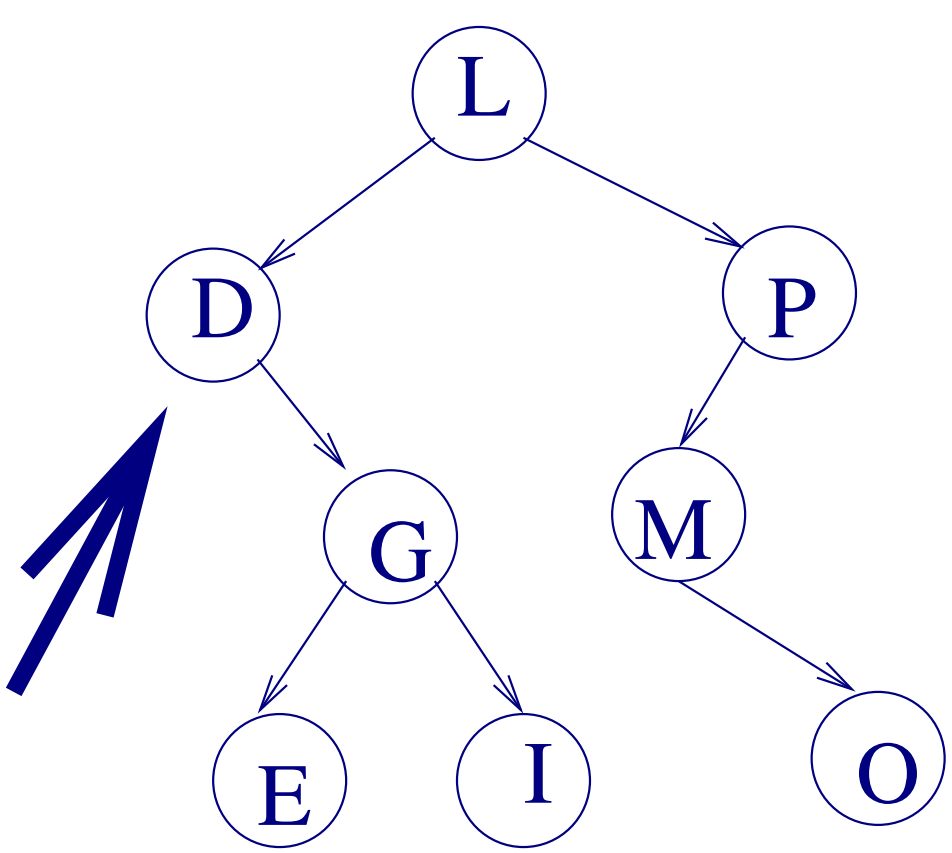
1. El nodo es un nodo hoja.
2. Es un nodo que sólo tiene un hijo.
3. Es un nodo con dos hijos.

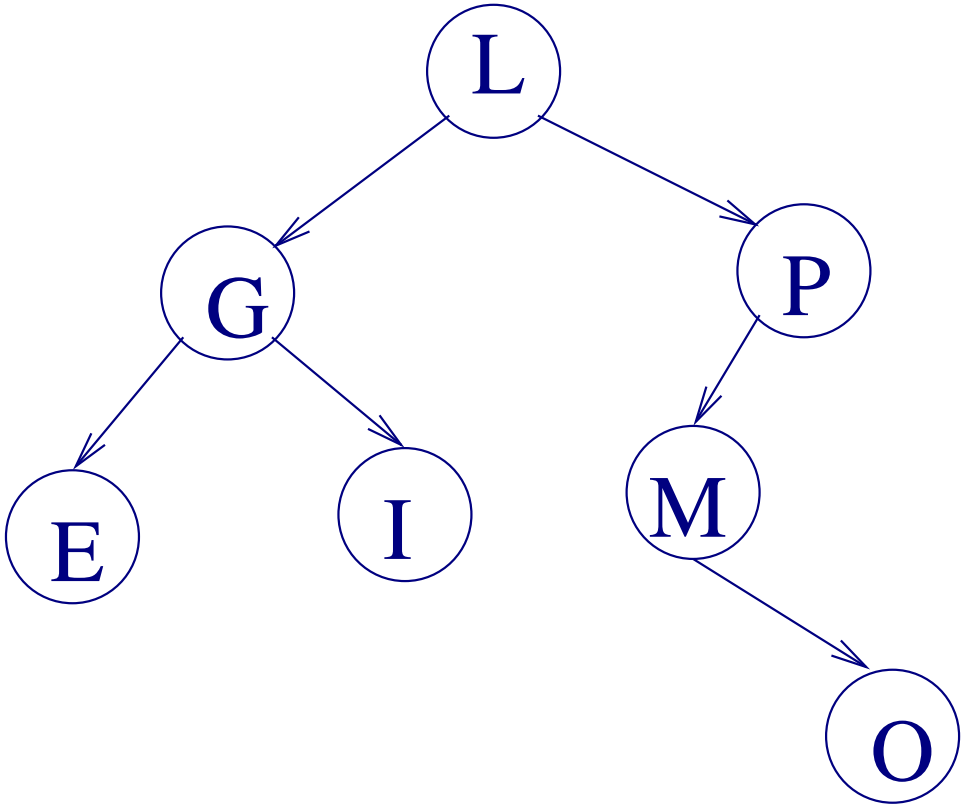
11.2 Operaciones básicas (III)

1. **Es un nodo hoja**: se elimina el nodo sin afectar al resto del árbol.

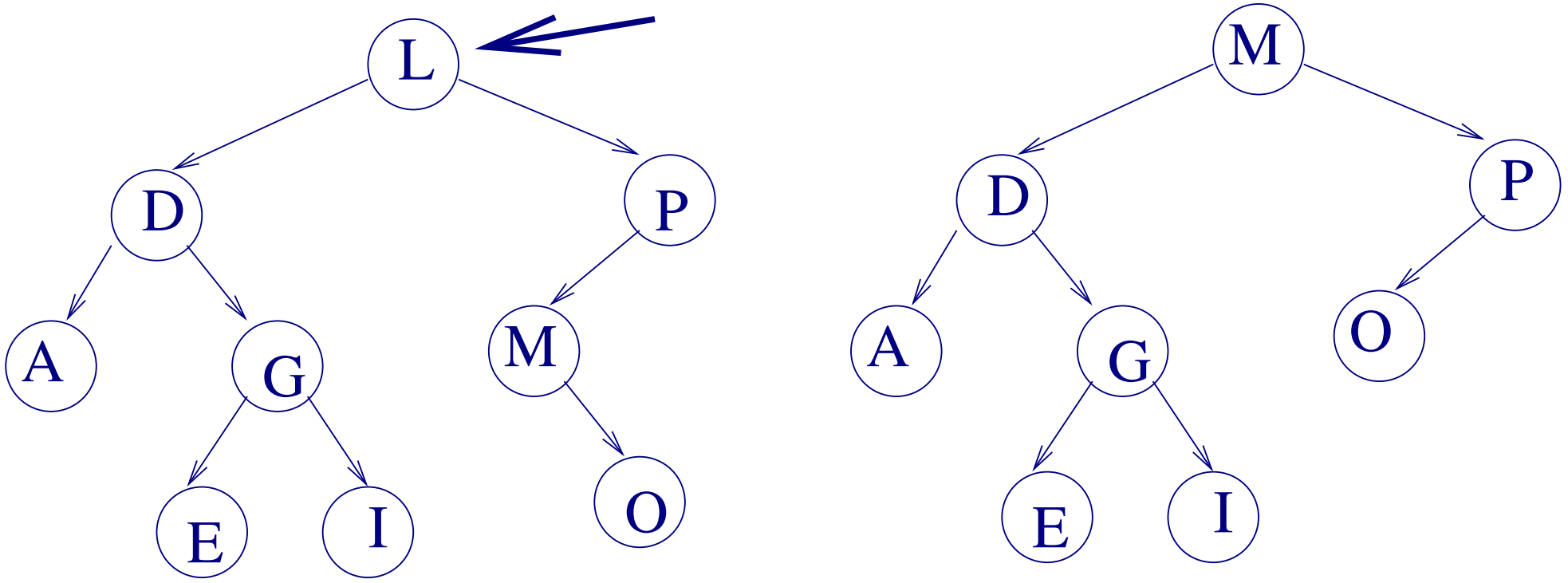


2. Es un nodo con un solo hijo: se sustituye el nodo a borrar por su subárbol hijo.





3. **Es un nodo con dos hijos**: se sustituye el elemento a borrar por la menor clave del subárbol derecho del nodo (alternativamente por la mayor clave de su subárbol izquierdo). Esto origina, a su vez, el borrado de un nodo hoja o un nodo con sólo hijo derecho.



11.2 Operaciones básicas (VI)

```
template <class T>
void ABB<T>::erase (const T & item) {
    erase(raiz, item);
}
template <class T>
void ABB<T>::erase (nodoptr & r, const T & item) {
  nodoptr temp;  T suc;
  if (r != NULL)
     if (item < r->info)
        erase (r->izq, item);
     else if (item > r->info)
        erase (r->der, item);
     else {   // item==r->info
```

11.2 Operaciones básicas (VII)

```
        if ((r->izq!=NULL) && (r->der!=NULL)) {   //2 hijos
            erasesuc(r->der, suc);
            r->info=suc;
        } else {
              temp=r;
              if ((r->izq==NULL) && (r->der==NULL)) //nodo hoja
                  r=NULL;
              else {  //nodo con solo un hijo
                  if (r->izq==NULL)  r=r->der;
                  else  r=r->izq;
              }
              delete temp;
        }
    }   }
```

12 Árboles AVL

Los árboles AVL son árboles binarios equilibrados en los que las alturas de los subárboles izquierdo y derecho de cualquier nodo difieren a lo sumo en 1.

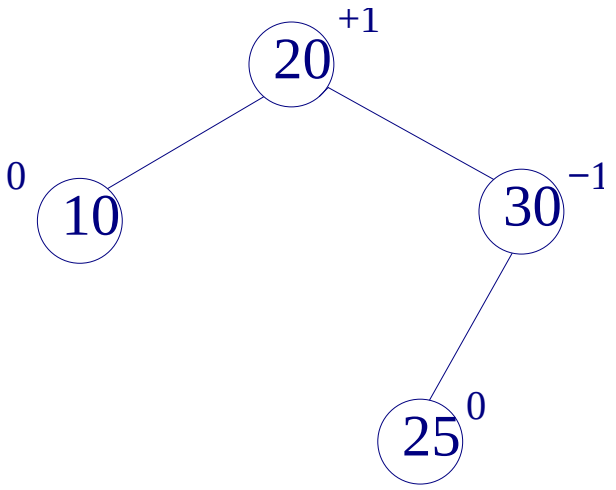
- La inserción y borrado en estos árboles es más compleja.
- Cada nodo llevará asociado un factor de equilibrio.
- Cuando la condición de equilibrio se rompe, se realizan una serie de rotaciones para equilibrar el árbol.
- La búsqueda, inserción y borrado tienen coste $O(log_2 n)$.

La idea de equilibrar ABB mediante rotaciones fue propuesta por **A**del'son-**V**el'skii y **L**andis.

12 Árboles AVL (II)

Factor de equilibrio:

- Implementación: añadir un campo *equilibrio* a cada nodo.
- Valores del campo *equilibrio*:
  - 0: si sus dos subárboles tienen igual altura.
  - +1: si el subárbol derecho tiene mayor altura que el subárbol izquierdo.
  - -1: si la altura del subárbol izquierdo es mayor que la del subárbol derecho.



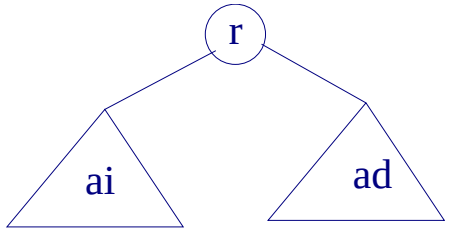
Inserción

- Igual que en un ABB: seguir el camino por el árbol hasta encontrar el lugar de inserción.
- Después de la inserción, hay que comprobar si sigue siendo un ABB equilibrado. Si es así, bastará con actualizar el campo *equilibrio* de los nodos en el camino de inserción. Si ya no es equilibrado, habrá que reequilibrarlo.

Pivote: Nodo en el camino de búsqueda cuyo *equilibrio* es +1 o -1 y es el nodo más próximo al nuevo nodo.

Durante la búsqueda del lugar de inserción, se localiza el nodo pivote (si existe).

12.1 Operaciones con árboles AVL (II)



El árbol de la figura está equilibrado.

r representa el nodo pivote si lo hay, y si no existe el pivote, cualquier nodo en el camino de inserción. Supóngase que el nuevo nodo n es menor que r.

Se distinguen 3 casos, según la condición que sea cierta antes de insertar:

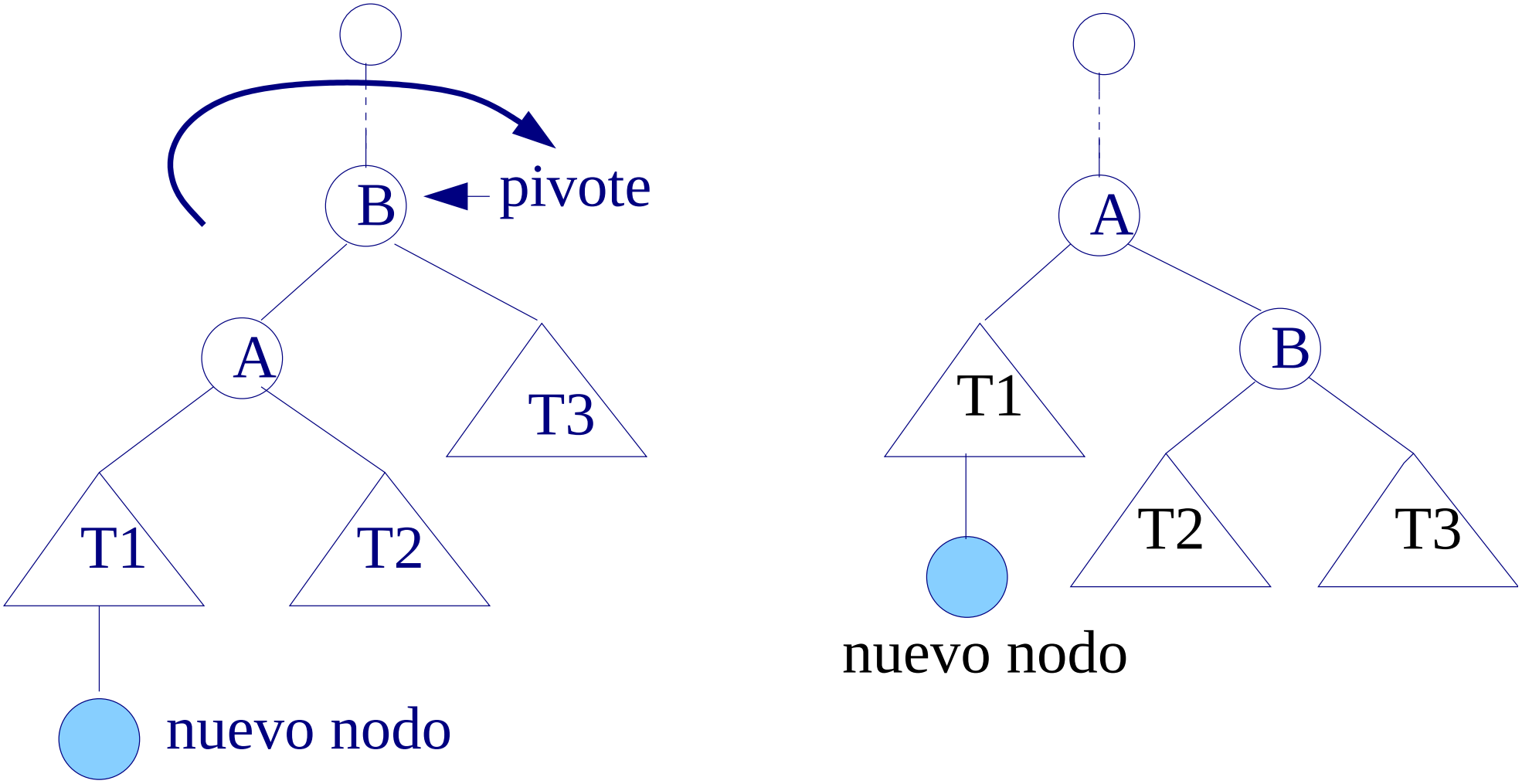
1. Si no hay pivote, Altura(ai)=Altura(ad): después de insertar, *ai* y *ad* tendrán distinta altura pero seguirá siendo un árbol equilibrado.
2. Altura(ai)$<$Altura(ad): tras la inserción, *ai* y *ad* tendrán la misma altura.
3. Altura(ai)$>$Altura(ad): se vulnerará el criterio de equilibrio. Habrá que reequilibrar el árbol mediante rotaciones.

12.1 Operaciones con árboles AVL (III)

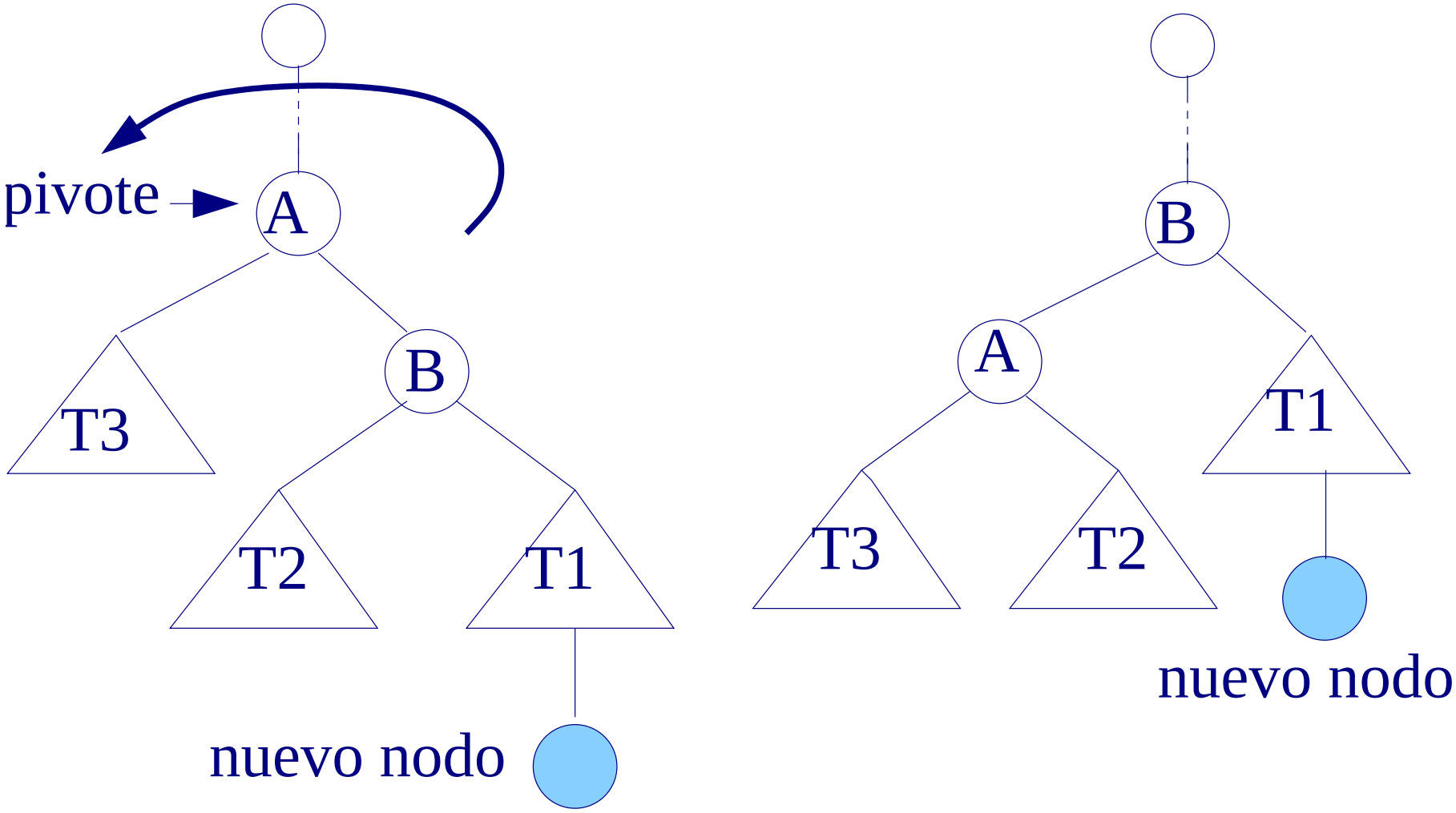
Casos:

1. No hay nodo pivote: Actualizar todos los campos *equilibrio* en los nodos del camino de búsqueda desde la raíz al nuevo nodo.
2. Existe nodo pivote pero añadimos el nodo en el subárbol de menor altura: actualizar los campos *equilibrio* en todos los nodos del camino de búsqueda desde el pivote hasta el nuevo nodo.
3. Existe nodo pivote y el nuevo nodo se añade en el subárbol de mayor altura: rotaciones para reequilibrar el árbol.

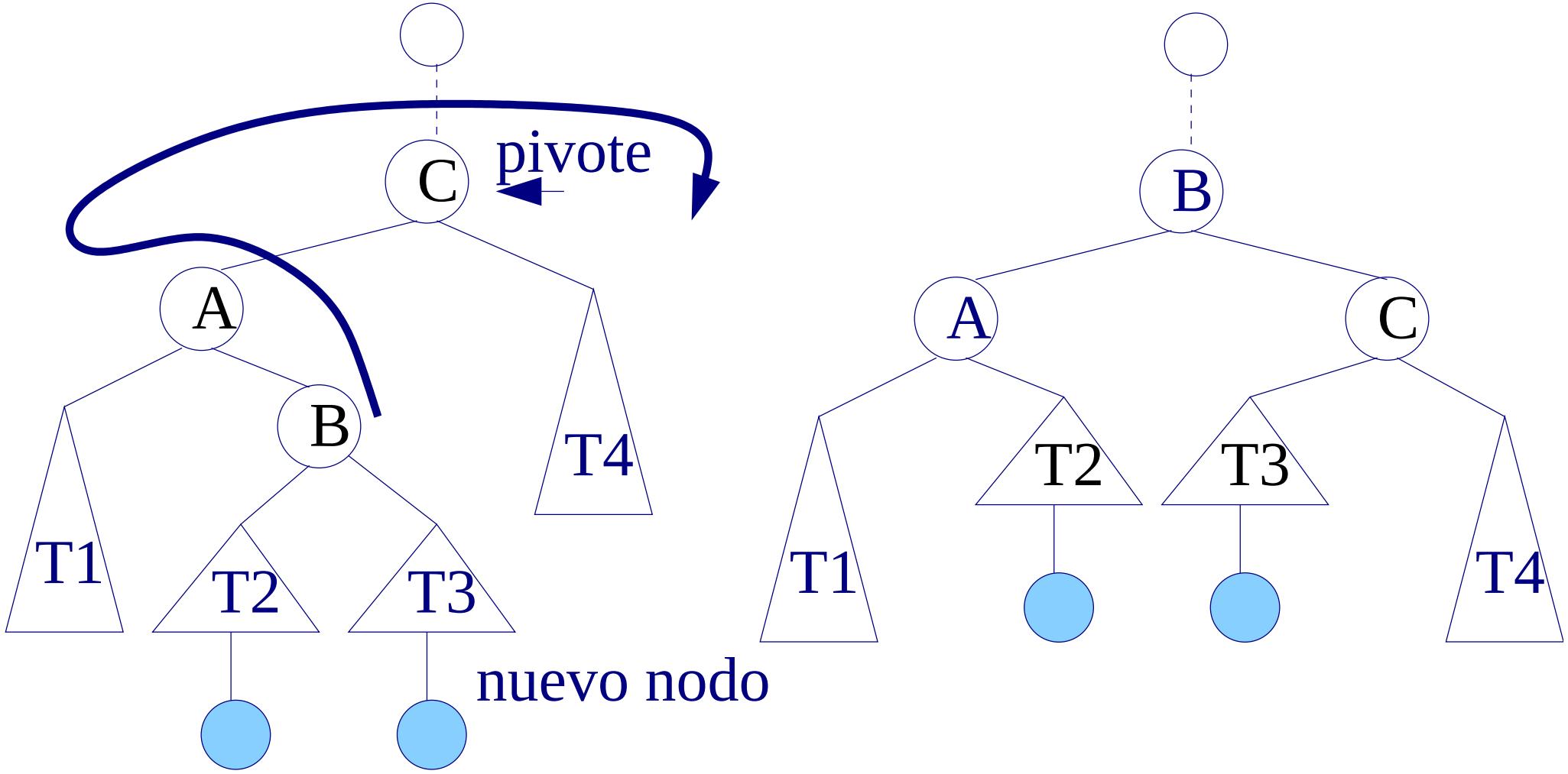
1. (a) Rotación única a la derecha



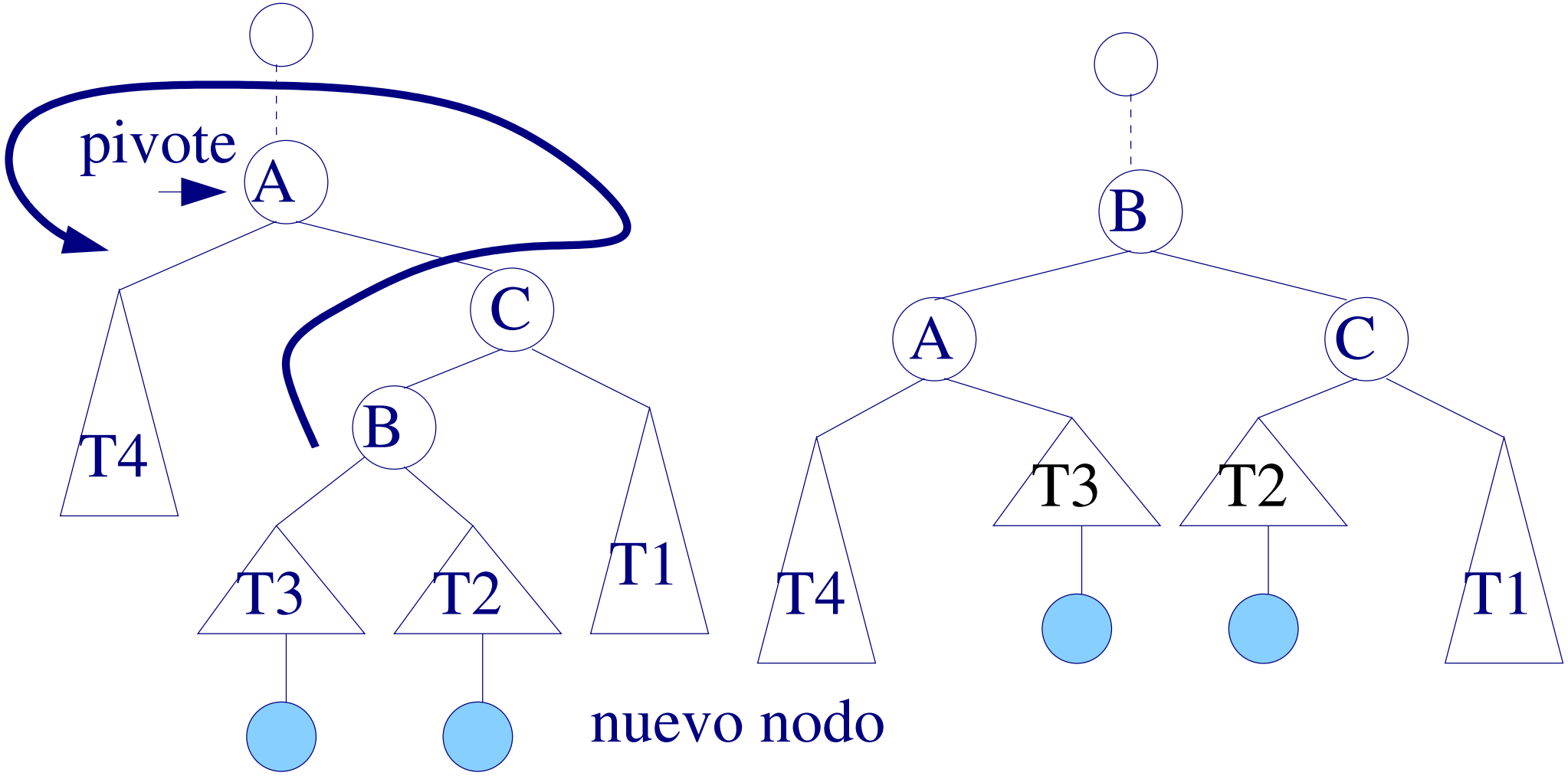
1. (b) Rotación única a la izquierda



2. (a) Rotación doble izquierda - derecha



2. (b) Rotación doble derecha - izquierda



13 Árboles B

Los **árboles B** son árboles de búsqueda **no binarios**.

- Cada nodo puede contener más de un elemento.
- Son árboles equilibrados.
- Los caminos de acceso son cortos en relación con el gran número de elementos que pueden almacenar.

13 Árboles B (II)

Aplicaciones:

- Implementar índices en bases de datos: acceder a registros almacenados en dispositivos externos a partir de un valor clave. Las claves se encuentran en el árbol B junto con la dirección del bloque en el fichero donde se encuentra el registro asociado.
- Gestión del sistema de archivos en el sistema operativo OS/2 para aumentar la eficiencia de la búsqueda de archivos en el árbol de subdirectorios.
- Sistemas de compresión de datos: se utilizan árboles B para la búsqueda por clave de datos comprimidos.

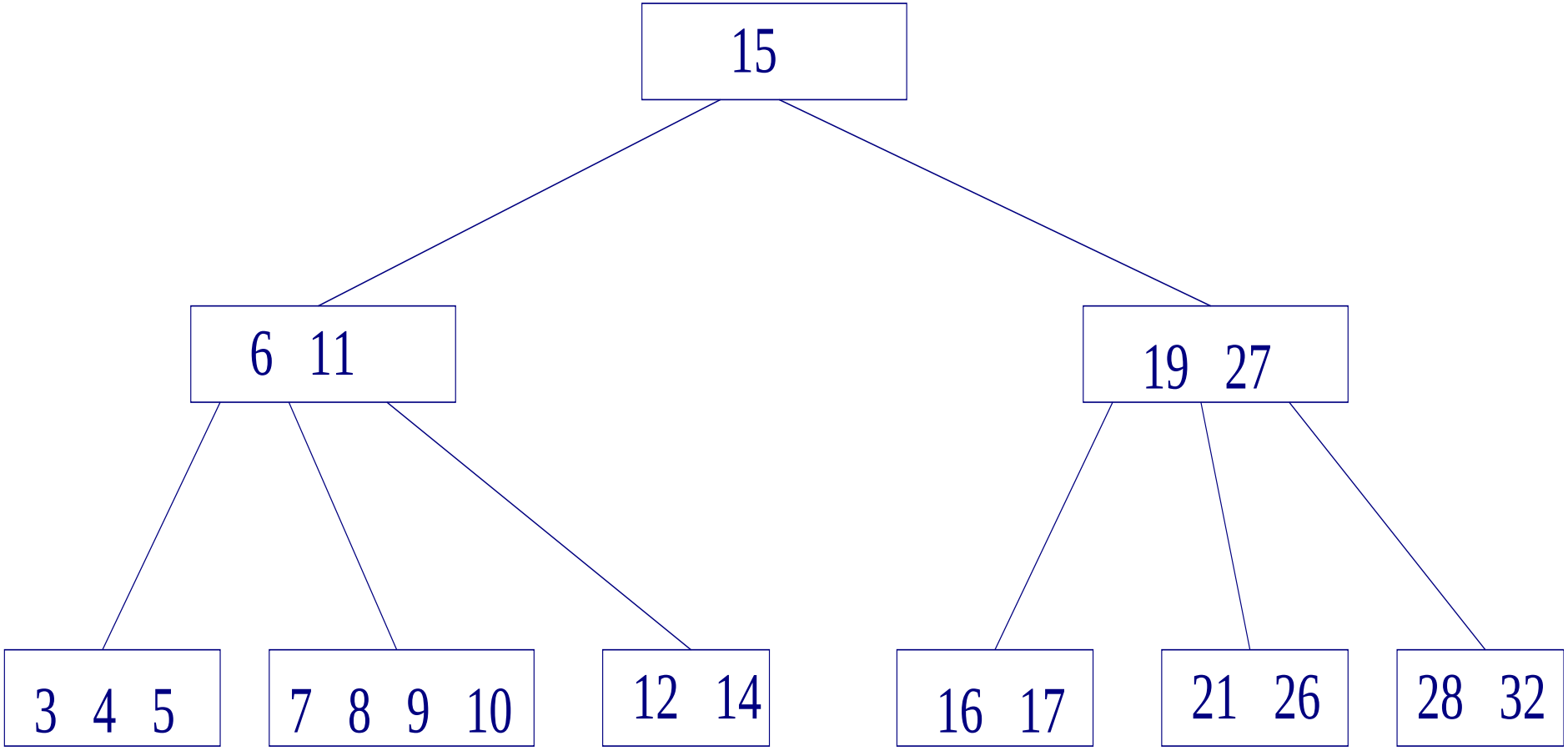
13 Árboles B (III)

Cuando el árbol B se utiliza para implementar un índice, a un nodo se le conoce como página, siendo una unidad que se accede en bloque.

- El acceso a memoria secundaria es costoso.
- Minimizar el número de accesos a disco: minimizar la altura del árbol.
- Cada vez que se accede a disco, se lee un página completa.
- En cada página (nodo) se almacenará tantas claves como sea posible.

13 Árboles B (IV)

Ejemplo: árbol B de orden 2



13 Árboles B (V)

Características de un arbol B de orden m:

- Todas las hojas están al mismo nivel. No hay subárboles vacíos.
- Cada nodo (excepto la raíz) puede contener entre m y $2 * m$ claves. El orden del árbol, m, es por tanto el número mínimo de elementos que puede tener un nodo.
- El nodo raíz puede contener entre 1 y $2 * m$ claves.
- El número de ramas (hijos) de los nodos internos es uno más del número de claves que contiene el nodo (como mínimo 0 y como máximo $2 * m + 1$).
- Las claves en cada nodo siguen una ordenación de izquierda a derecha.
- Las claves de un nodo dividen a los nodos descendientes como en un árbol de búsqueda.

13.1 TAD árbol B

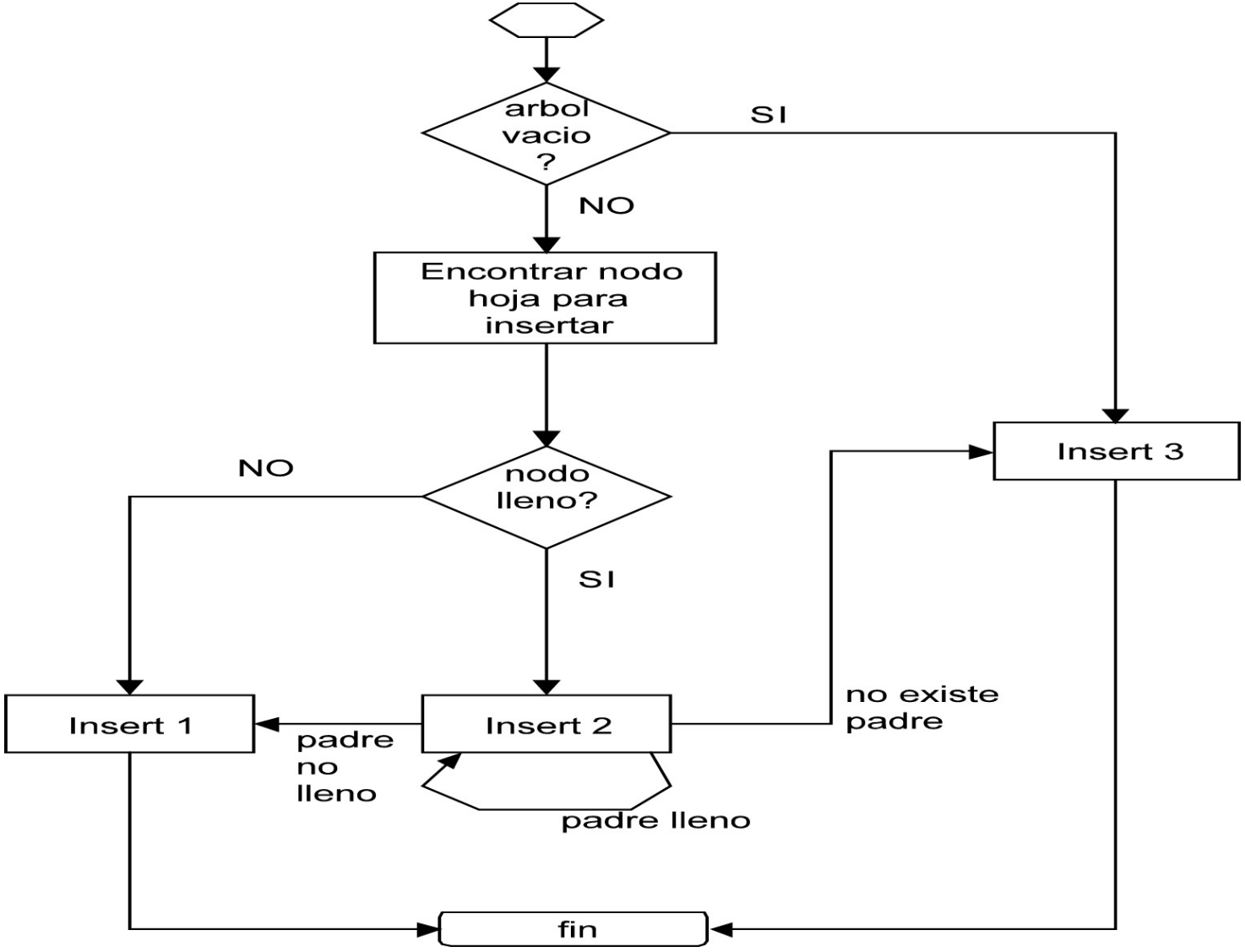
```
template <class T>
class arbolB {
public:
    arbolB(int nClaves);
    ~arbolB();
    long Buscar (int clave);      // Buscar un valor de clave
    bool Insertar(T clave);       // Insertar una clave
    void Borrar(T clave);         // Borrar una clave

private:
    class nodob {
    public:
        nodob(int nClaves);
        ~nodob();
```

```
        int clavesUsadas;       // Claves usadas en el nodo
        T *claves;              // Vector de claves del nodo
        nodob **hijos;          // Vector de punteros a nodob
        nodob *padre;           // Puntero a nodo padre
    };

    int nClaves;            // Número de claves por nodo
    int nodosMinimos;       // Número de punteros mínimos por nodo
    nodob * raiz;           // Puntero a nodo de entrada en el árbol-B
};
```

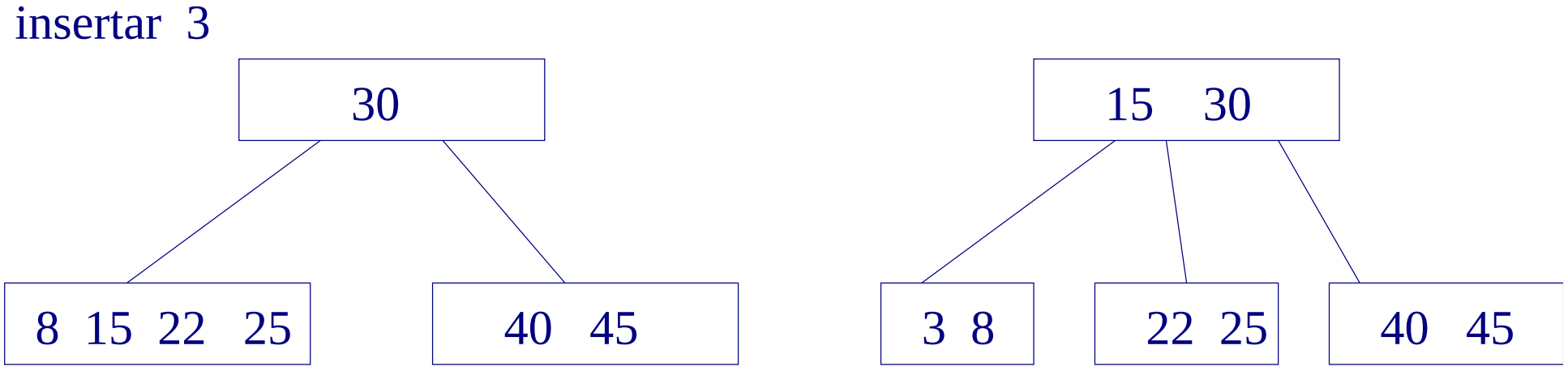
13.2 Inserción en un árbol B



13.2 Inserción en un árbol B (II)

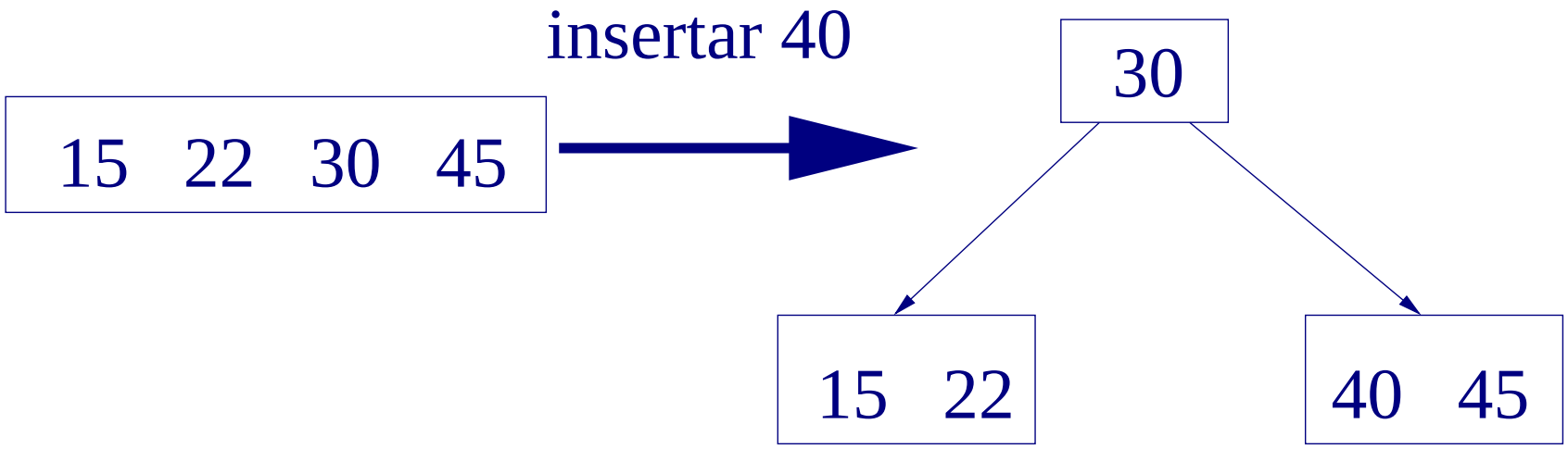
1. Localizar el nodo hoja donde realizar la inserción de modo similar a un ABB.

- *Insert 1*: El nodo contiene menos de $2 * m$ elementos. Se inserta en orden y la inserción finaliza.
- *Insert 2*: El nodo está lleno y tiene que dividirse. Un nuevo nodo contedrá los m elementos menores y el otro nuevo nodo los m elementos mayores. El elemento medio pasa a insertarse en el nodo padre del nodo dividido. La inserción continua en el nodo padre hasta llegar a la raíz.



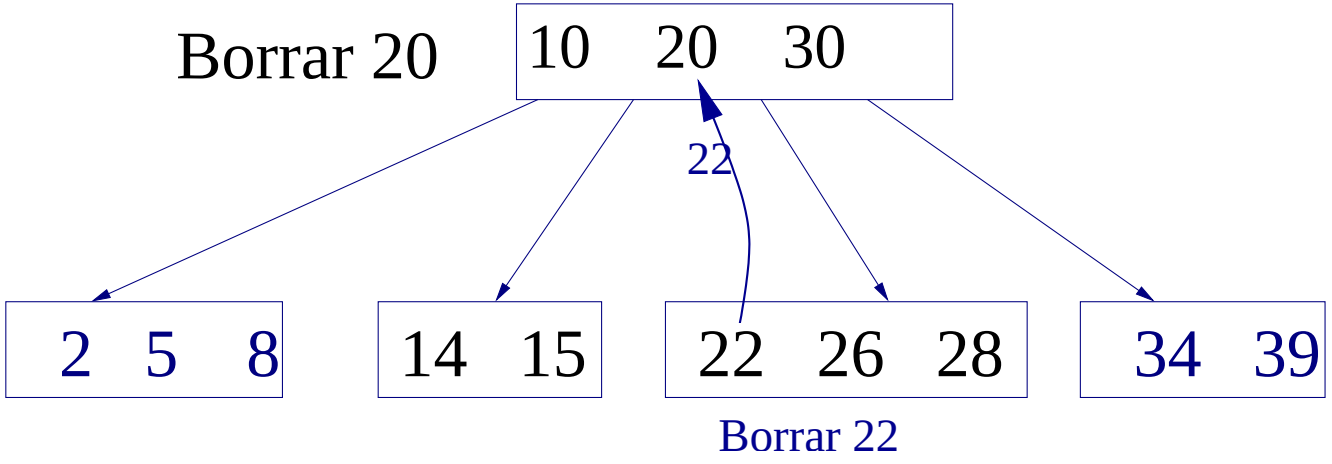
13.2 Inserción en un árbol B (III)

- *Insert 3*: Se crea una nueva raíz con un único elemento. Aumenta, por tanto, el nivel del árbol en 1.



13.3 Borrado en un árbol B

1. Buscar el elemento a borrar (similar a la búsqueda en un ABB).
2. Si el elemento a borrar NO está en un nodo hoja: sustituirlo por el menor elemento inmediatamente posterior (opcionalmente por el mayor elemento inmediatamente anterior). Este elemento está en un nodo hoja.
   - Esta operación dará lugar a un borrado en un nodo hoja.

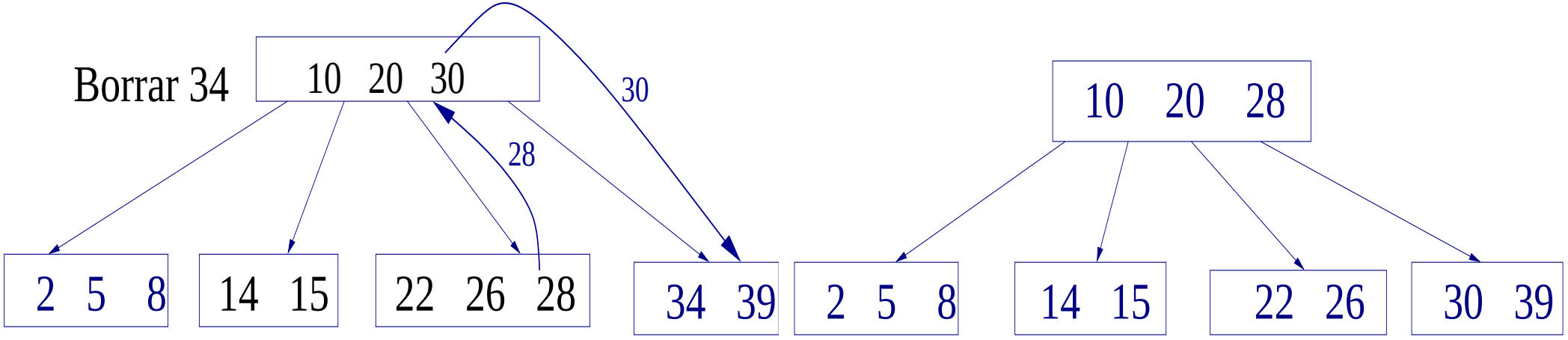


13.3 Borrado en un árbol B (II)

Sea el nodo hoja donde se borra el *nodo objetivo*.

Casos:

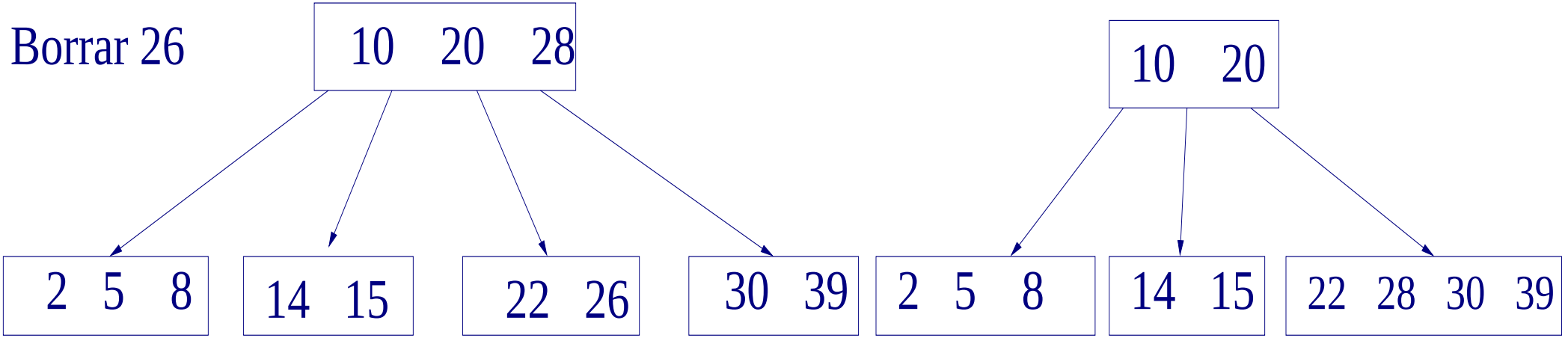
- *Borrar 1*: El nodo objetivo tiene más de m elementos. El elemento se borra y el proceso finaliza.
- *Borrar 2*: El nodo objetivo contiene exactamente m elementos (el mínimo).
  - *Caso (a)*: El hermano izquierdo o derecho del nodo objetivo tiene más de m elementos, entonces se pasa el elemento correspondiente al nodo objetivo (ver dibujo). Esto implica un intercambio con un elemento del nodo padre. El proceso termina.



13.3 Borrado en un árbol B (III)

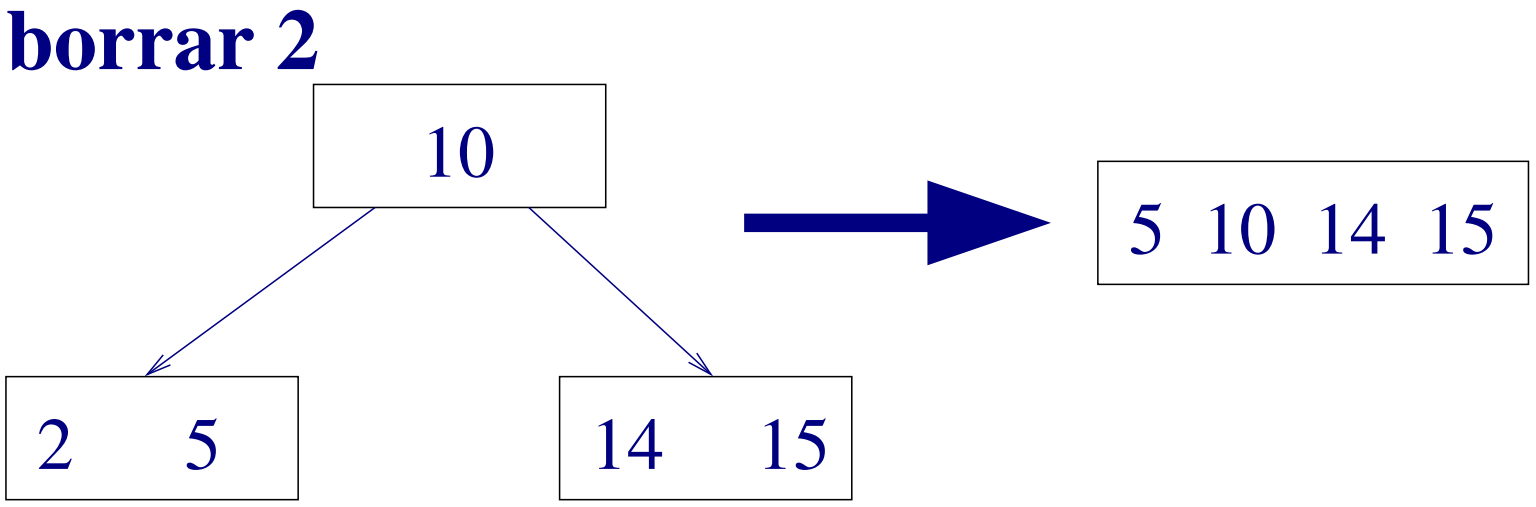
- *Borrar 2 (cont.)*
  - *Caso (b)*: Si los dos nodos hermanos contienen exáctamente m elementos, entonces el nodo objetivo se combina con uno de ellos para formar un sólo nodo con $2 * m$ elementos: los elementos de ambos nodos más el elemento del nodo padre que los separaba. Por tanto, se produce un borrado en el nodo padre.

1. Si el nodo padre tiene más del mínimo de elementos el proceso finaliza.
2. Si no, el nodo padre pasa a ser el nodo objetivo:
   a) Si el padre es la raíz y tiene 1 elemento: ver *Borrar3*.
   b) Si no, repetir *Borrar 2* con el nodo padre.



13.3 Borrado en un árbol B (IV)

- *Borrar 3*: Se borra el nodo raíz con un sólo elemento. La altura del árbol B decrece en una unidad y la nueva raíz será el resultado de unir los dos nodos hijos de la raíz original.



13.4 Variantes de los árboles B

- **Árboles B***: Si el nodo donde se inserta está lleno, mueve las claves a uno de sus hermanos. El proceso de división se postpone hasta que los dos nodos están completamente llenos. Entonces, se dividen en 3 nuevos nodos, de modo que dichos nodos están llenos en dos terceras partes.
- **Árboles B+**: Todas las claves están en las hojas y se duplican en la raíz y en los nodos internos aquellas necesarias para definir el camino de búsqueda. Cada hoja está enlazada con la siguiente.
  - Acceso aleatorio rápido y acceso secuencial rápido.

14 Árboles generales

- Para representar un árbol de grado g debemos disponer de espacio para guardar cada posible subárbol. Esto puede resultar en un desperdicio de memoria.
- Es posible representar cualquier árbol como un árbol binario, utilizando sólo 2 subárboles por nodo.
  - El primer subárbol guardado será aquel cuya raíz sea el primer hijo (en orden de aparición de los hijos de izquierda a derecha).
  - El segundo subárbol será aquel cuya raíz sea el primer hermano derecho (iniciando una lista de hermanos).

14 Árboles generales (II)

```
class arbol {
public:
  //constructor y operaciones
private:
  class nodo {
  public:
    T info;
    arbol<T> izq, her;
    nodo (const T &e=T(),
          const arbol<T> &ni=arbol(),
          const arbol<T> &nh=arbol()):
          info(e), izq(ni), her(nh) {}
  };
  nodo  *raiz; };
```

14 Árboles generales (III)