Tema 3
Árboles II: Estructuras para búsqueda

1. Árboles binarios de búsqueda
2. Árboles parcialmente ordenados (montículos)

Introducción

- **Motivación:** Compatibilidad que presenta el diseño de árboles con algoritmos eficientes para determinadas tareas, p.e. búsqueda en una estructura ordenada.
- **Importancia de la ordenación:**
  - Uno de los problemas más y mejor estudiados en computación, con una amplia variedad de algoritmos.
  - Si la lista está ordenada de forma ascendente, podremos reconocer una ausencia en cuanto encontremos una clave mayor que la que buscamos.
- **Ejemplo: Búsqueda de un elemento en una lista de N elementos.**
  - Búsqueda lineal, COSTE **O(N)**: **cs+cc*k**
  - Búsqueda binaria , COSTE **O(log N): cs+cc*log(k)**

    cs = coste constante de definir una búsqueda (analizar argumentos, etc.)

    cc = coste de comprobar un elemento de la lista

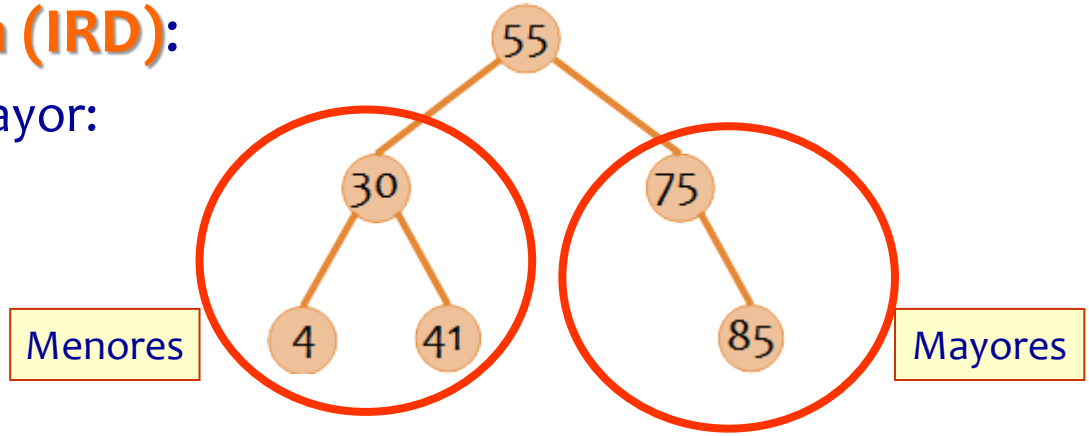
    k = 1, en el mejor de los casos, y **N** en el peor de los casos.

Definición:

- Dado un nodo:
  - todos los datos del subárbol izquierdo son menores
  - todos los datos del subárbol derecho son mayores.
- **Clave de ordenación**: criterio de ordenación menor-mayor
- **Ventaja**: Tiempo de ejecución promedio: **O (log N)**
  - Imponer condiciones de balanceado (árboles AVL, árboles B, etc.) sobre los ABB para que se mantenga *O*(log N).
    - Si no balanceamos, tiempo de ejecución ***O*(*n*)** en el peor caso (prueba a insertar los elementos de mayor a menor y obtendrás un árbol que es como una lista).
- **Recorrido inorden (IRD)**:

  Datos de menor a mayor:

  4 30 41 55 75 85



1.1. Especificación del TAD Árbol Binario de Búsqueda

ESPECIFICACIÓN= *definición del **tipo de dato** y las **operaciones** que sirven para manipularlo o hacer preguntas sobre él, pues trabajaremos con **TIPOS DE DATOS OPACOS** (la implementación está oculta al usuario).*

abb=TAD con operaciones **crear**, **destruir**, **es_vacio**, **es_miembro**, **leer**, **izq**, **der**, **buscar_nodo**, **insertar**, **suprimir**, **modificar**.

DESCRIPCIÓN: Los valores del TAD **abb** son árboles binarios de búsqueda donde cada nodo contiene un dato del tipo **tipoelem**. Los árboles son **mutables**: **insertar, suprimir** y **modificar** añaden, eliminan y modifican el árbol.

OPERACIONES de creación/destrucción* del árbol:

crear() devuelve **(abb)**

- efecto: Devuelve un árbol binario vacío.

1.1. Especificación del TAD Árbol Binario de Búsqueda

OPERACIONES de información, ya que no puedo acceder al contenido del tipo de datos por ser OPACO:

`es_vacio(A:abb)` devuelve **`(booleano)`**

- efecto: Devuelve **cierto** si **A** es el árbol vacío, y **falso** en caso contrario.

`izq/der(A:abb)` devuelve **`(abin)`**

- efecto: Devuelve la posición del nodo hijo izquierdo/derecho del subárbol **A**. Si el nodo **A** no tiene hijo izquierdo/derecho, devuelve **nulo**.

`leer(A:abb)` devuelve **`(tipoelem)`**

- efecto: Devuelve el contenido del nodo **A** en el árbol.

`es_miembro(A:abb;E:tipoelem)` devuelve **`(booleano)`**

- efecto: devuelve cierto si E es un elemento del árbol binario de búsqueda A, y falso en caso contrario.

`buscar_nodo(A:abb;C:tipoclave)` devuelve **`(tipoelem)`**

- efecto: devuelve el elemento del árbol binario de búsqueda A cuya clave de ordenación es C.

1.1. Especificación del TAD Árbol Binario de Búsqueda

OPERACIONES de modificación, ya que no puedo acceder al contenido del tipo de datos por ser OPACO:

`insertar(A:abb; E:tipoelem)`

- modifica: **A**.
- efecto: Añade un nodo, con contenido **E**, al árbol binario de búsqueda **A**.

`suprimir(A:abb; E:tipoelem)`

- modifica: **A**.
- efecto: Suprime el elemento E, si existe, en el árbol binario de búsqueda **A**.

`modificar(A:abb; C:tipoclave; E:tipoelem)`

- modifica: **A**.
- efecto: Modifica el elemento **E** cuya clave de ordenación es **C**, en el árbol binario de búsqueda **A**.

1.2. Implementación del TAD Árbol Binario de Búsqueda

- **Implementación del TAD:**
  - **Selección** de un lenguaje de programación para pasar de la especificación a la implementación: **lenguaje C**
  - Implementación de un TAD en lenguaje C:
    - **Interfaz de usuario: `abb.h`**
    - **Implementación: `abb.c`** (oculta al usuario)

OPCIÓN A: menos repetición, menos opacidad

abb.h

- Se define abb como un puntero a `struct` (se pierde opacidad):

```
typedef struct celda *abb;
```

- Se definen `tipoelem` y `tipoclave`, el tipo de dato de cada nodo y su clave:

```
typedef int tipoelem;
typedef int tipoclave;
```

abb.c

- Se incluye `abb.h`, lo que evita la repetición de la definición de `tipoelem` y `tipoclave`:

```
#include "abb.h"
```

OPCIÓN B: más repetición, más opacidad

abb.h

- Se define abb como un puntero a void (se gana opacidad):

```
typedef void *abb;
```

- Se definen tipoelem y tipoclave, el tipo de dato de cada nodo y su clave:

```
typedef int tipoelem;
typedef int tipoclave;
```

abb.c

- Se repite la definición de tipoelem y tipoclave:

```
typedef int tipoelem;
typedef int tipoclave;
```

1.2. Implementación del TAD Árbol Binario de Búsqueda

- **abb.h:**

abb.h

```
1   #ifndef ABB_H
2   #define ABB_H
3   /* Tipo de dato abstracto para arbol binario con clave de
4    * ordenacion y elemento de celda. */
5   typedef int tipoclave;
6   typedef int tipoelem;
7   //TIPO OPACO, no sabemos como esta construido celda
8   typedef struct celda *abb;
9   ////////////////////////////// FUNCIONES
10  void crear(abb *A);
11  void destruir(abb *A);
12  unsigned es_vacio(abb A);
13  abb izq(abb A);
14  abb der(abb A);
15  void leer(abb A, tipoelem *E);
16  unsigned es_miembro(abb A, tipoelem E);
17  void buscar_nodo(abb A, tipoclave cl, tipoelem *nodo);
18  void insertar(abb *A, tipoelem E);
19  void suprimir(abb *A, tipoelem E);
20  void modificar(abb A, tipoelem nodo);
21  #endif  /* ABB_H */
```

GUARDAS DEL COMPILADOR

Evitan la redefinición de tipos y funciones

1.2. Implementación del TAD Árbol Binario de Búsqueda

- **abb.c:**

abb.c

```c
#include <stdlib.h>

// Importar aqui abb.h permite no repetir la definición de tipoelem, tipoclave y abb.
#include "abb.h"

////////////////////////// SE DEFINE EL CONTENIDO DE struct celda
struct celda {
    tipoelem info;
    struct celda *izq, *der;
};
```

Detalles de la implementación (ocultos al usuario)

Creación/Destrucción de abb

```
void crear(abb *A){
    *A=NULL;
}
```

```
void destruir(abb *A) {
    if (!es_vacio(*A)) {
        destruir(&((*A)->izq));
        destruir(&((*A)->der));
        _destruir_elem(&((*A)->info));
        free(*A);
        *A = NULL;
    }
}
```

Función privada que libera la memoria ocupada por info, si contiene estructuras dinámicas

Funciones de información

```
unsigned es_vacio(abb A){
    return A==NULL;
}
```

```
abb izq(abb A){
    return A->izq;
}
```

```
abb der(abb A){
    return A->der;
}
```

```
void leer(abb A, tipoelem *E){
    *E=A->info;
}
```

Funciones de información

```
unsigned es_miembro(abb A, tipoelem E) {
    return _es_miembro_clave(A, _clave_elem(&E));
}
```

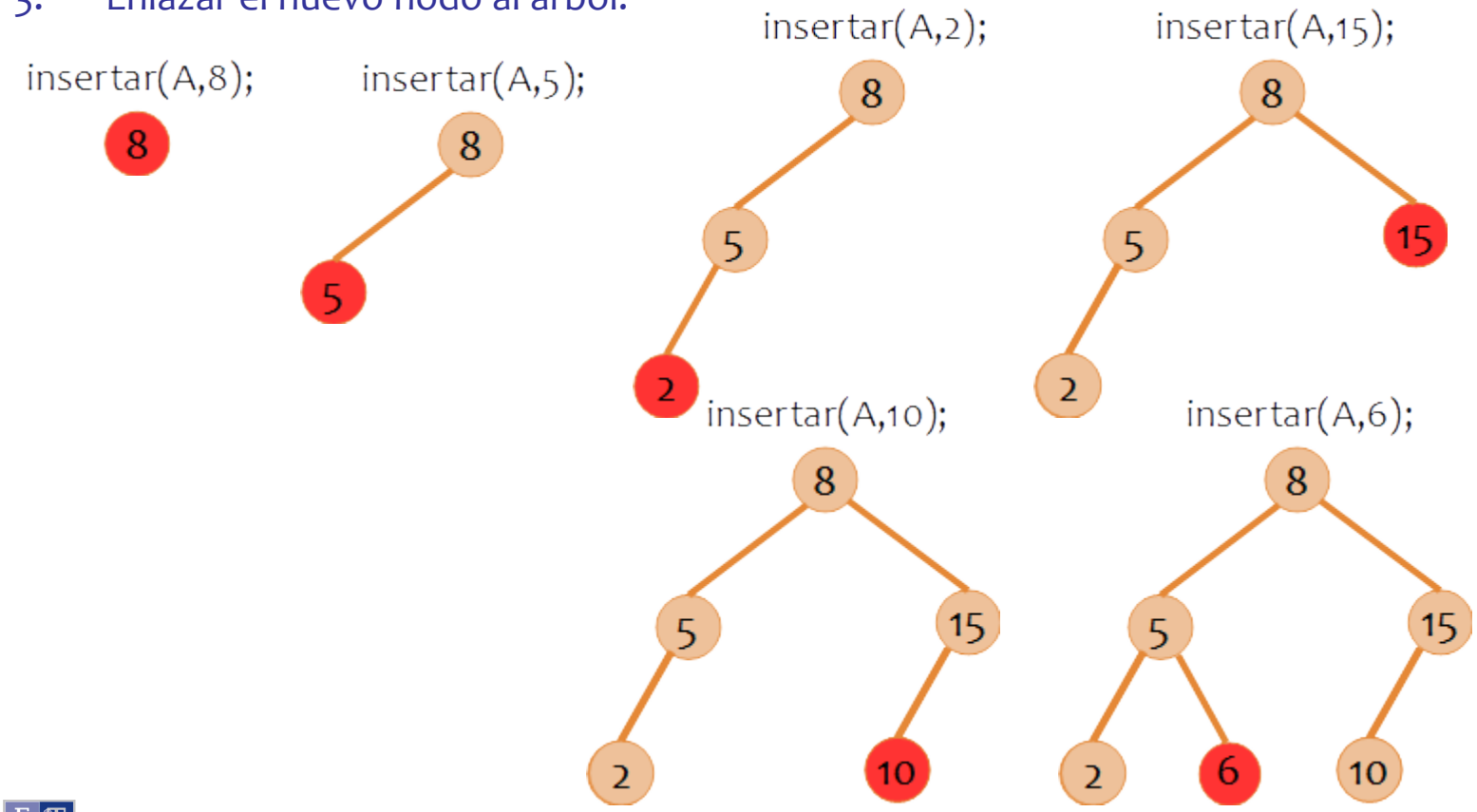
Función privada

```
void buscar_nodo(abb A, tipoclave cl, tipoelem *nodo) {
    if (es_vacio(A)) {
        return;
    }
    int comp = _comparar_clave_elem(cl, A->info);

    if (comp == 0) {                    // cl == A->info
        *nodo = A->info;
    } else if (comp < 0) {              // cl < A->info
        buscar_nodo(A->izq, cl, nodo);
    } else {                            // cl > A->info
        buscar_nodo(A->der, cl, nodo);
    }
}
```

Función privada

Funciones de modificación: Inserción recursiva

1. Asignar memoria para un nuevo nodo
2. Buscar en el árbol para encontrar la posición de inserción del nuevo nodo, que se colocará como nodo hoja:
   1. *Si dato<raíz, insertamos por subárbol izquierdo*
   2. *Si dato>raíz, insertamos por subárbol derecho*
3. Enlazar el nuevo nodo al árbol.



Funciones de modificación

```
void modificar(abb A, tipoelem nodo) {
    tipoclave cl = _clave_elem(&nodo);
    _modificar(A,cl,nodo);
}
```

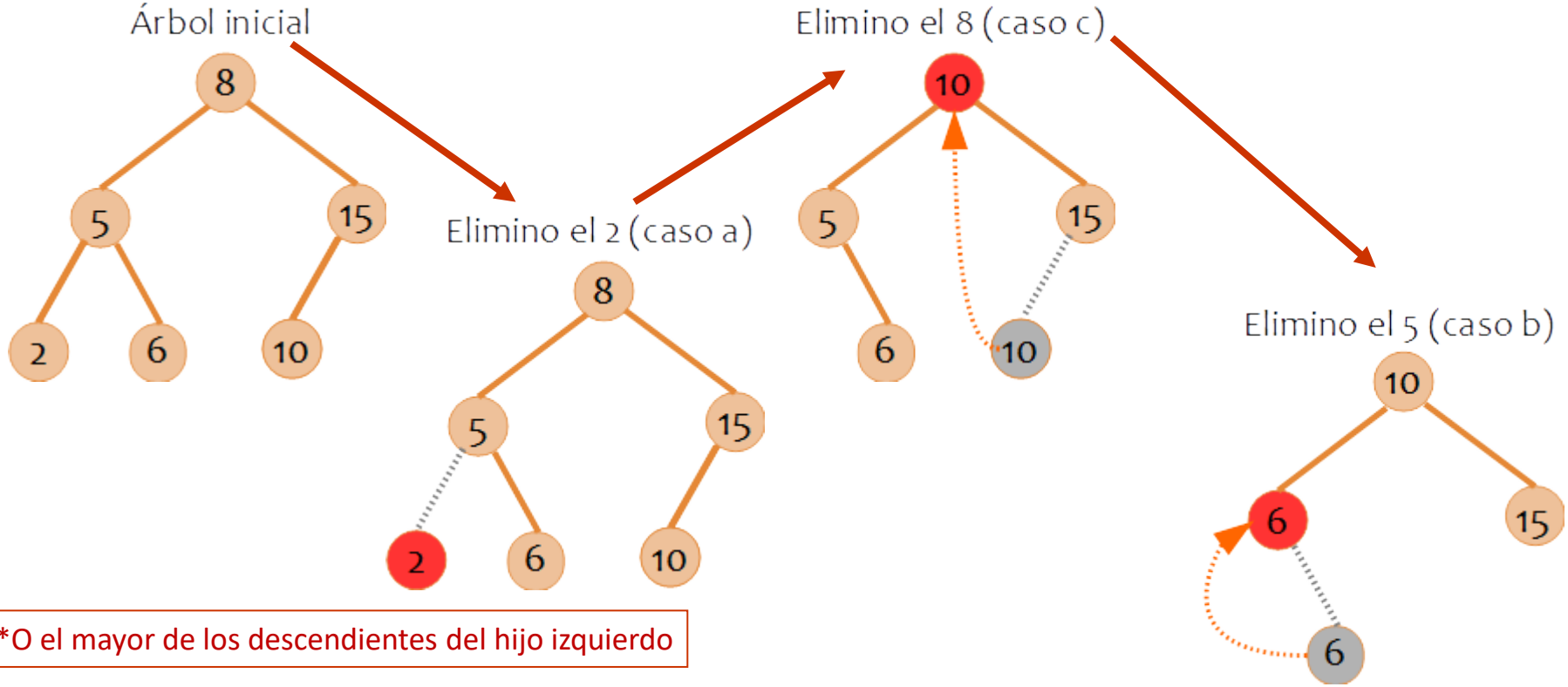
Función privada

```
void insertar(abb *A, tipoelem E) {
    if (es_vacio(*A)) {
        *A = (abb) malloc(sizeof (struct celda));
        (*A)->info = E;
        (*A)->izq = NULL;
        (*A)->der = NULL;
        return;
    }
    tipoclave cl = _clave_elem(&E);
    int comp = _comparar_clave_elem(cl, (*A)->info );
    if (comp > 0 ) {
        insertar(&(*A)->der, E);
    } else {
        insertar(&(*A)->izq, E);
    }
}
```

Función privada

Funciones de modificación: Eliminación:

1. Buscamos posición nodo a eliminar
2. Reajustamos los punteros:
   a) Si el nodo es hoja, se suprime del árbol
   b) Si el nodo tiene un único hijo, sustituimos el nodo por el hijo
   c) Si el nodo tiene dos hijos, buscamos el menor de los descendientes del hijo derecho*, y sustituimos el nodo por este descendiente.



*O el mayor de los descendientes del hijo izquierdo

1.2. Implementación del TAD Árbol Binario de Búsqueda

Funciones de modificación: Eliminación:

```
void suprimir(abb *A, tipoelem E) {
    abb aux;
    if(!es_vacio(*A)){ return; }
    tipoclave cl = _clave_elem(&E);
    int comp = _comparar_clave_elem(cl, (*A)->info);
    if(comp < 0){              //E < (*A)->info
        suprimir(&(*A)->izq, E);
    } else if (comp > 0){      //E > (*A)->info
        suprimir(&(*A)->der, E);
    } else if (es_vacio((*A)->izq) && es_vacio((*A)->der)){
        _destruir_elem(&((*A)->info));
        free(*A);
        *A = NULL;
    } else if (es_vacio((*A)->izq)) {
        aux = *A;
        *A = (*A)->der;
        _destruir_elem(&aux->info);
        free(aux);
    } else if (es_vacio((*A)->der)) {
        aux = *A;
        *A = (*A)->izq;
        _destruir_elem(&aux->info);
        free(aux);
    } else {          //comp==0 y tiene dos hijos
        _destruir_elem(&((*A)->info));
        (*A)->info = _suprimir_min(&(*A)->der);
    }
}
```

1. Buscamos posición nodo a eliminar

2. Reajustamos los punteros

a) Nodo hoja: lo suprimo del árbol

b) Sólo tiene hijo derecho: se sustituye por él

b) Sólo tiene hijo izquierdo: se sustituye por él

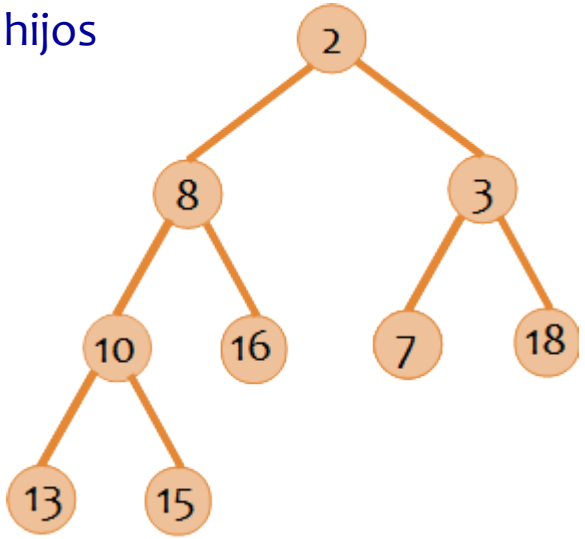
c) Tiene los dos hijos:
suprimo el mínimo del subárbol derecho*

*O el máximo del subárbol izquierdo

- **Definición de Montículo binario:**

  Es un árbol binario completo parcialmente ordenado:

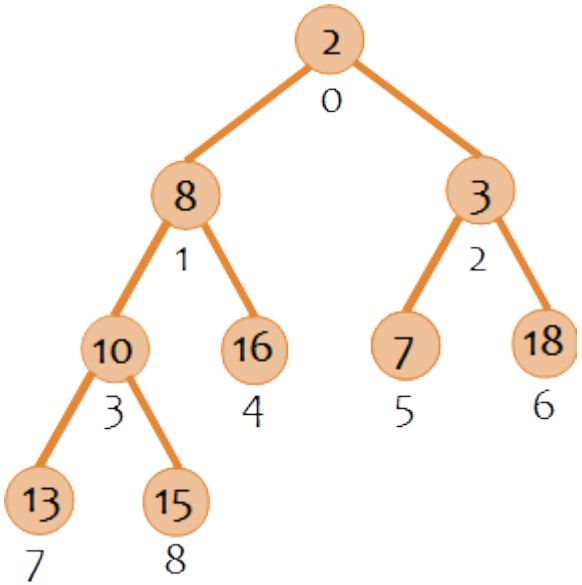
  - Completo: en el nivel más bajo pueden faltar algunos nodos, pero éstos han de ser los de más a la derecha posible.
  - Parcialmente ordenado:
    - **Montículo de mínimos**: clave nodo <= clave hijos
    - **Montículo de** máximos: clave de nodo >= clave hijos



2. Árboles parcialmente ordenados o Montículos

- **Propiedades de Montículo binario o *heap*:**
  - Da soporte eficiente a las operaciones del TAD cola de prioridad, que proporciona una flexibilidad extra a la hora de ordenar:
    - Los trabajos, usuarios, etc. a menudo llegan al sistema a intervalos arbitrarios.
    - Es más eficiente insertar un trabajo nuevo en una cola de prioridad que reordenar todo cada vez que llega un nuevo dato.
  - El orden parcial es más débil que el orden total (sin embargo más eficiente de mantener) pero más fuerte que el orden aleatorio (de manera que el elemento con mayor prioridad se puede identificar de forma rápida, ya que **SIEMPRE es la raíz**).
- **Aplicaciones**: todas las de cola de prioridad
  - Simulación de eventos discretos: Aeropuertos, aparcamientos, planificación de trabajos en sistema multiusuario, colas de impresión, etc. las colas de prioridad se utilizan para gestionar quién va a continuación.
  - Algoritmos voraces: Siempre elegimos el siguiente elemento, valor, etc. que maximiza localmente nuestra puntuación/función objetivo.

- **Implementación mediante arrays**
  - Representación más natural: árbol binario. Sin embargo, podemos almacenar el árbol como un array de claves, usando la **posición** de las claves para representar *implícitamente* el papel de los punteros:
    - Raíz: `A[0]`
    - Hijo izquierdo de A[k]: `A[2k+1]`
    - Hijo derecho de A[k]: `A[2k+2]`
    - Padre de A[k]: `A[(k-1)/2], k>1`



0	1	2	3	4	5	6	7	8
2	8	3	10	16	7	18	13	15

2.1. Implementación de montículos

- **Representación mediante arrays**
- **¿Se puede representar de manera implícita cualquier árbol binario?**
  - **Es más eficiente cuanto más lleno esté el árbol.**
  - **Todos los nodos internos no existentes ocupan espacio en la estructura**
    - Por eso es importante que los montículos estén tan balanceados / llenos en cada nivel como sea posible y que se llenen siempre desde la izquierda (árbol semicompleto).
  - **Y esta representación no es tan flexible para modificaciones arbitrarias como un árbol basado en punteros.**

- **Representación mediante arrays**

- **Ventajas**
  - Se aprovecha la posibilidad de operar aritméticamente con los índices del vector para obtener, en tiempos de ejecución constantes, las posiciones de los padres e hijos izquierdo y derecho.
  - Es más eficiente en espacio que la representación enlazada pues no hay necesidad de almacenar punteros y, como el árbol es completo, no hay espacio desaprovechado de nodos internos.
- **Desventajas**
  - Se requiere conocer a priori el número máximo de elementos del árbol.

Operaciones en las colas de prioridad

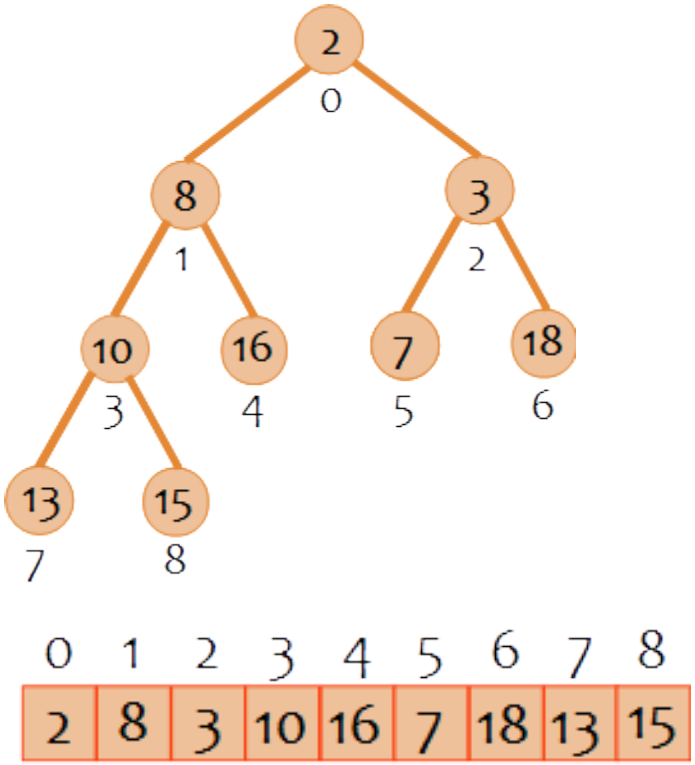
La cola de prioridad básica soporta tres operaciones primarias:

- `insertar(C,E,p)`: Dado un elemento E con clave k, lo inserta en la cola de prioridad C con una prioridad p.
- `suprimir(C)` – Elimina el elemento de la cola de prioridad cuya clave tiene la mayor prioridad (suele ser la raíz).

Cada una de estas operaciones se puede ejecutar fácilmente usando montículos o árboles binarios balanceados en O(log N), ya que la altura de un árbol binario completo de N nodos es log(N+1).

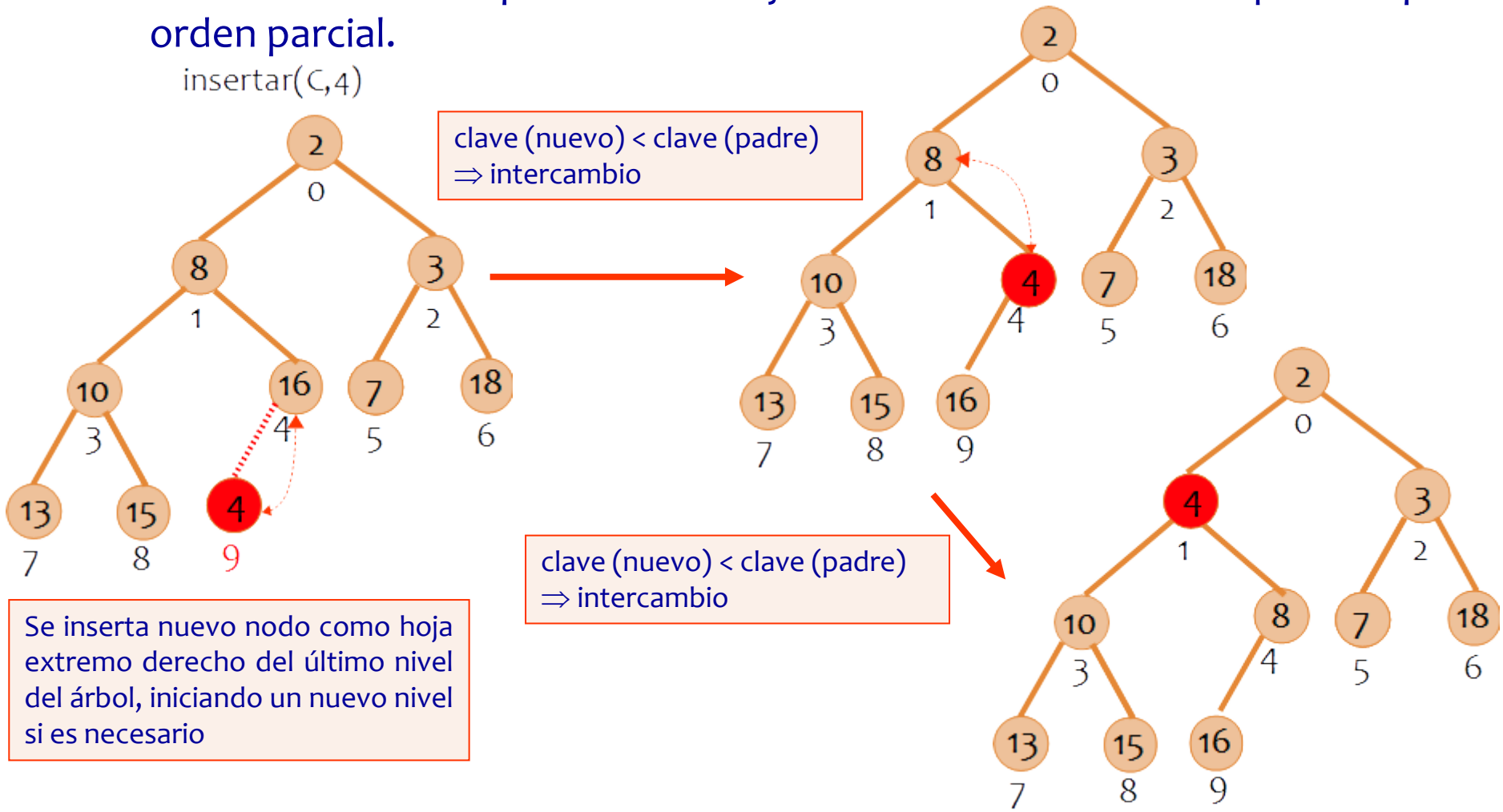
- Implementación de montículo=cola de prioridad en C
  - Definición de los tipos de datos

```
#define maxLong 100

typedef struct{
     tipoelem info;
     int prioridad;
}celda;

typedef struct monticulo{
     celda datos[maxLong];
     int ult;
}
typedef struct monticulo * cola;
```



Mantenimiento de montículos: insertar

- Asegurar que el montículo mantiene su forma y el orden parcial. Se inserta el elemento en el primer hueco y se hace "flotar" hasta que cumple el orden parcial.



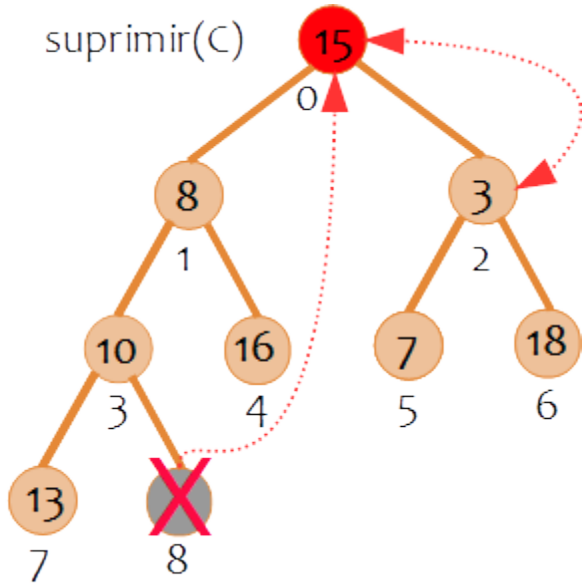
Mantenimiento de montículos: insertar

```
void insertar(cola *C, tipoelem E, int prioridad){
     int I, padre;
     unsigned encontrado;
     (*C)->ult=(*C)->ult+1;
     I=(*C)->ult;
     encontrado=0;
     while(I>0 && !encontrado){
        padre=(I-1)/2;
        if (prioridad > (*C)->datos[padre].prioridad)
            encontrado=1;
        else{ //pongo padre en posición I
            (*C)->datos[I].elemento=(*C)->datos[padre].elemento;
            (*C)->datos[I].prioridad=(*C)->datos[padre].prioridad;
            I=padre;
        }
     }
     //Encontré su sitio, inserto el dato
     (*C)->datos[I].elemento=E;
     (*C)->datos[I].prioridad=prioridad;
}
```

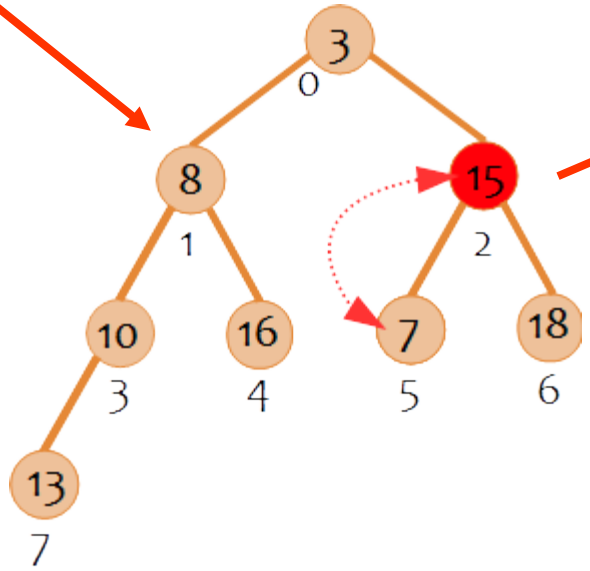
Mantenimiento de montículos: suprimir

- Asegurar que el montículo mantiene su forma y el orden parcial. Siempre se suprime la raíz y se "sube" el último elemento. Después, se "hunde" ese elemento hasta que se cumple la regla del orden parcial.

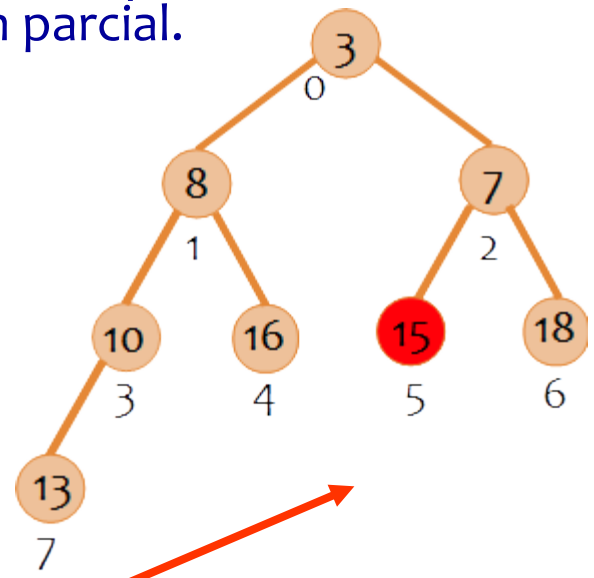
1. Devolvemos elemento raíz (valor 2)
2. Muevo elemento en hoja extremo derecha (valor 15) a raíz y elimino posición extremo derecha.



clave (nuevo) > clave (hijos)
⇒ Intercambio con hijo menor



clave (nuevo) > clave (hijos)
⇒ Intercambio con hijo menor



- **Mantenimiento de montículos: suprimir**

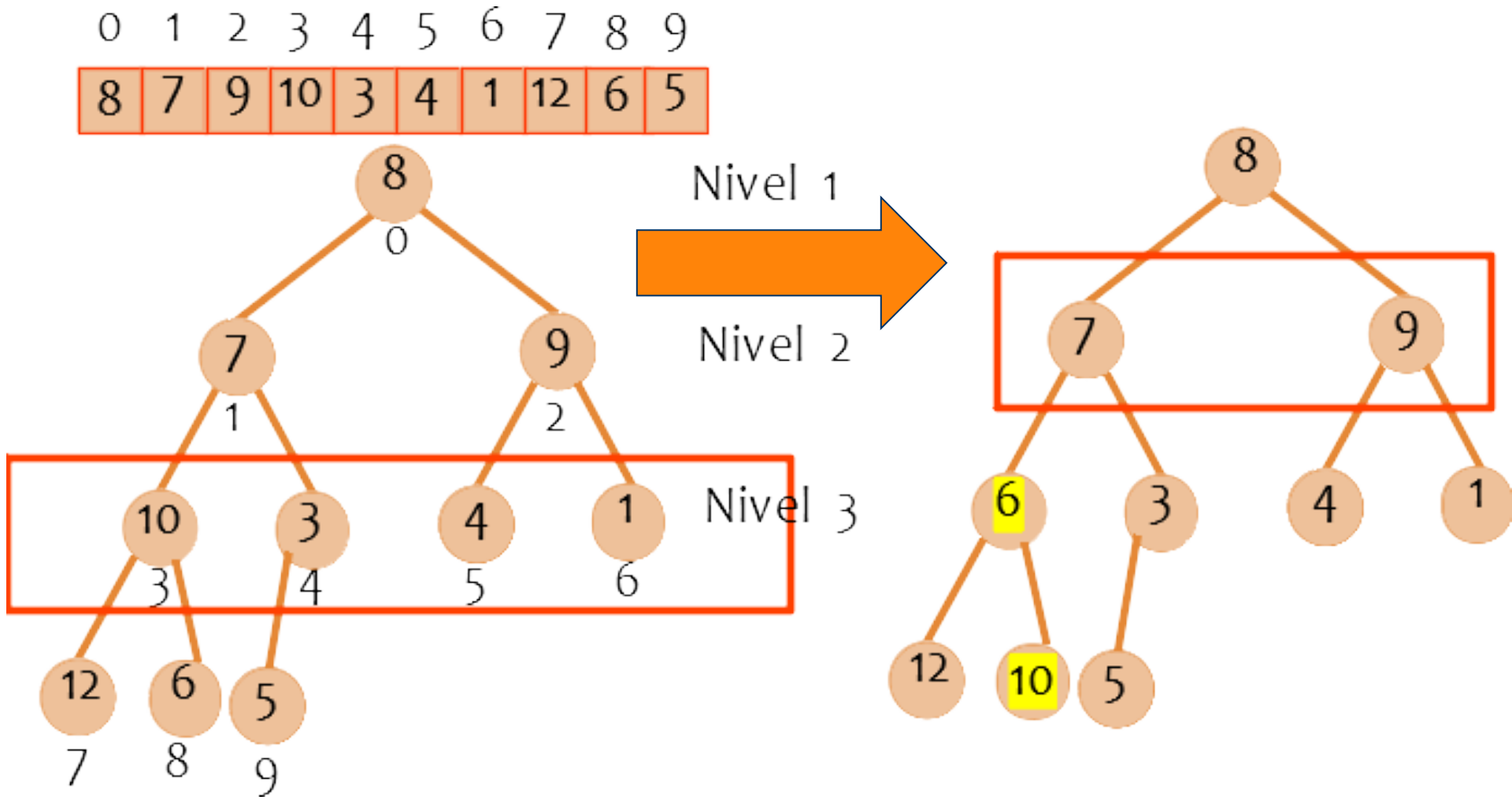
```
void suprimir(cola *C){
     tipoelem elemento;
     int I,J, prioridad;
     unsigned encontrado;

     elemento=(*C)->datos[(*C)->ult].elemento;
     prioridad=(*C)->datos[(*C)->ult].prioridad;
     (*C)->ult=(*C)->ult-1;
     I=0; //raíz
     J=2*I+1; //hijo izquierdo de raíz
     encontrado=0;
     while(J<=(*C)->ult && !encontrado){
         if(J<(*C)->ult)
             if ((*C)->datos[J].prioridad > (*C)->datos[J+1].prioridad)
                  J++; //hijo derecho
         if(prioridad <= (*C)->datos[J].prioridad )
             encontrado=1;
         else{
             (*C)->datos[I].elemento=(*C)->datos[J].elemento;
             (*C)->datos[I].prioridad=(*C)->datos[J].prioridad;
             I=J;
             J=2*I+1;
         }
     }
     (*C)->datos[I].elemento=elemento;
     (*C)->datos[I].prioridad=prioridad;
}
```

2.2. Transformación en un montículo

- **Monticulización (o *heapify*)**
  - A veces, dado un conjunto cerrado de elementos, nos pedirán representarlos por medio de un montículo (de máximos o de mínimos), proceso que se denomina *monticulizar* o *heapify*.
  - El procedimiento es el siguiente:
    - Dibujar el árbol que resulta a partir del array.
    - Si el árbol tiene L niveles, desde i=L-1 hasta 1 hacer lo siguiente:
      - Comprobar si cada nodo del nivel i cumple la regla de orden parcial, es decir, es menor (min-heap) o mayor (max-heap) que sus hijos. Si no la cumple, intercambiar con el menor/mayor de sus hijos.
      - Propagar esta comparación hacia los niveles desde i hasta L-1

2.2. Transformación en un montículo

- **Monticulización (*heapify*)**
  - Ejemplo:



2.2. Transformación en un montículo

- **Monticulización (*heapify*)**
  - Ejemplo: