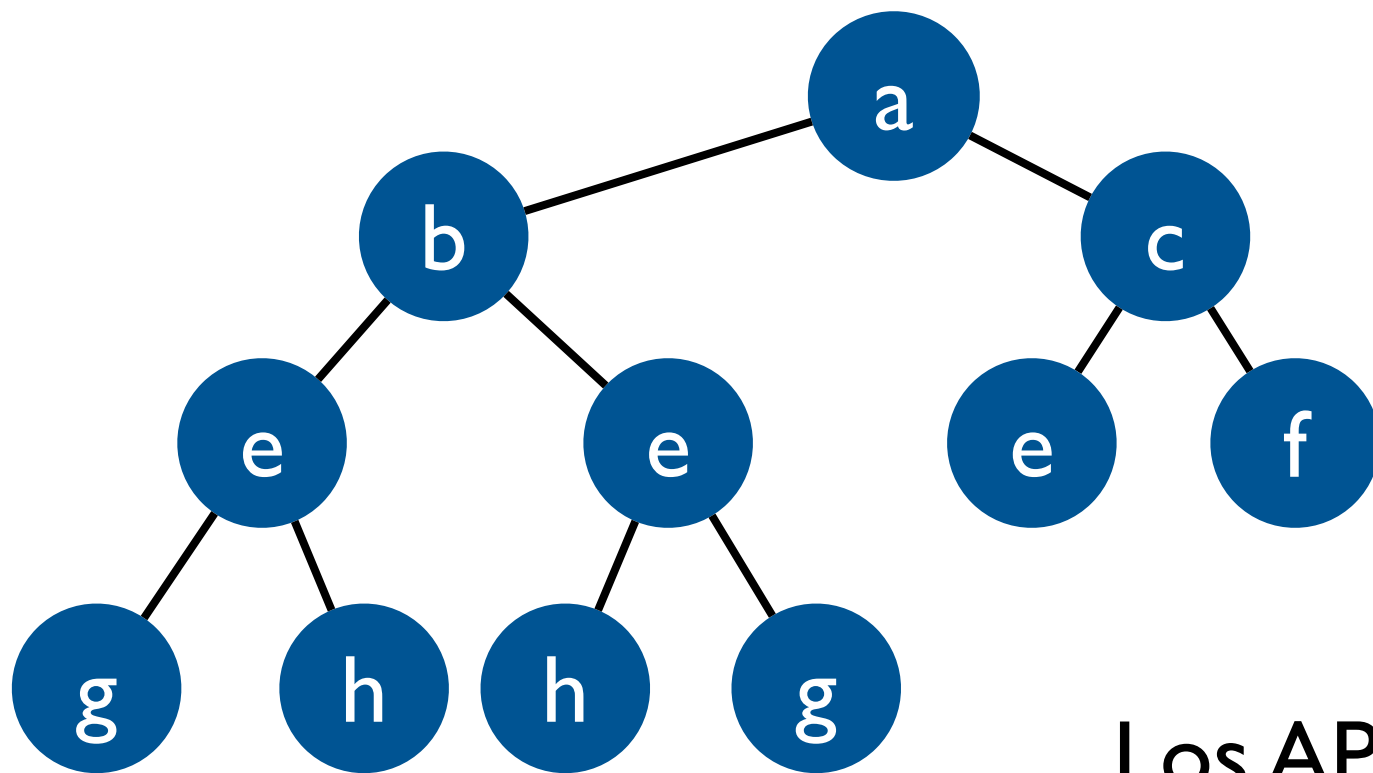


# **ÁRBOLES BINARIOS PARCIALMENTE ORDENADOS**

# Definición de APO

- Se dice que un árbol binario es un APO si cumple la condición de que la etiqueta de cada nodo es menor o igual que las etiquetas de los hijos, manteniéndose tan equilibrado (balanceado) como sea posible (hojas empujadas a la izquierda)



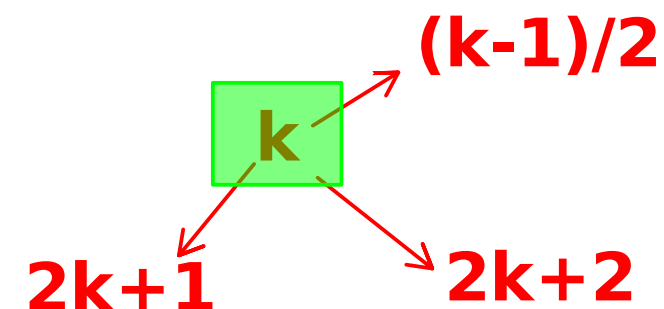
Sólo nos interesan funciones para:

- insertar elementos
- borrar la raíz
- consultar la raíz

Los APO son útiles para ordenación  
**HEAPSORT**

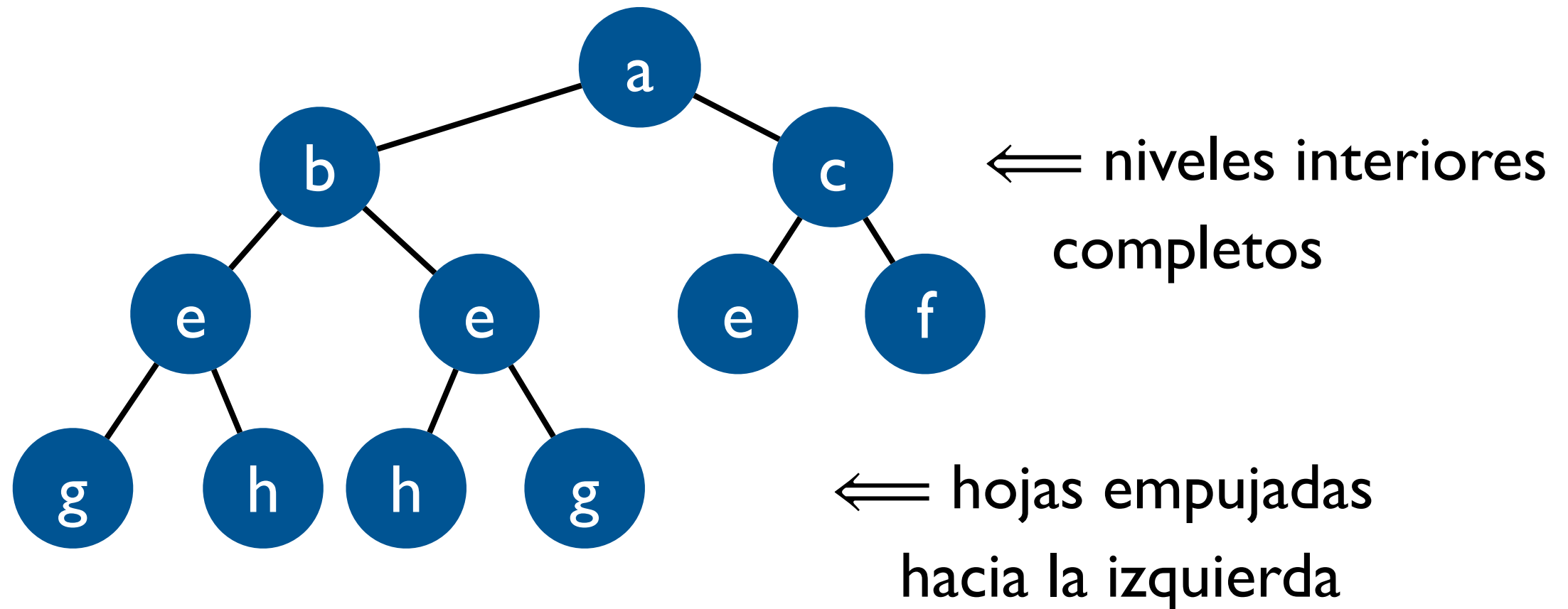
# Representación: el montón

- La representación que usaremos para los APO es la del montón (Heap)
- Un montón,  $M$ , en este contexto, será un vector en el que guardaremos el APO por niveles, de forma que si existen  $n$  nodos:
  - $M[0]$  alojará a la raíz
  - Los hijos izquierdo y derecho (si existen) del nodo  $M[k]$  estarán en  $M[2k+1]$  y  $M[2k+2]$ , lo que equivale a decir que el padre de  $M[k]$  es  $M[(k-1)/2]$ ,  $\forall k > 0$



# Representación: el montón

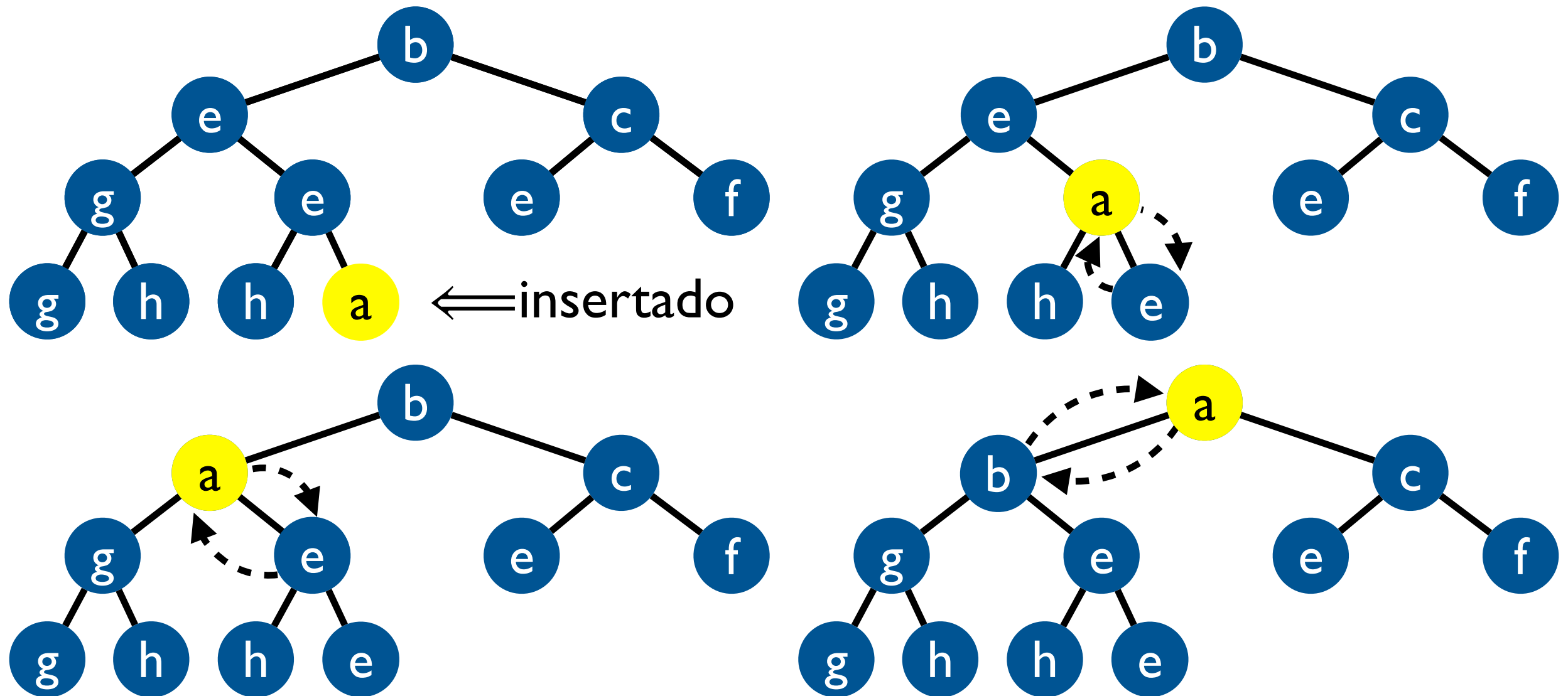
- El APO



se representa en el vector

0	1	2	3	4	5	6	7	8	9	10	...
a	b	c	e	e	e	f	g	h	h	g	...

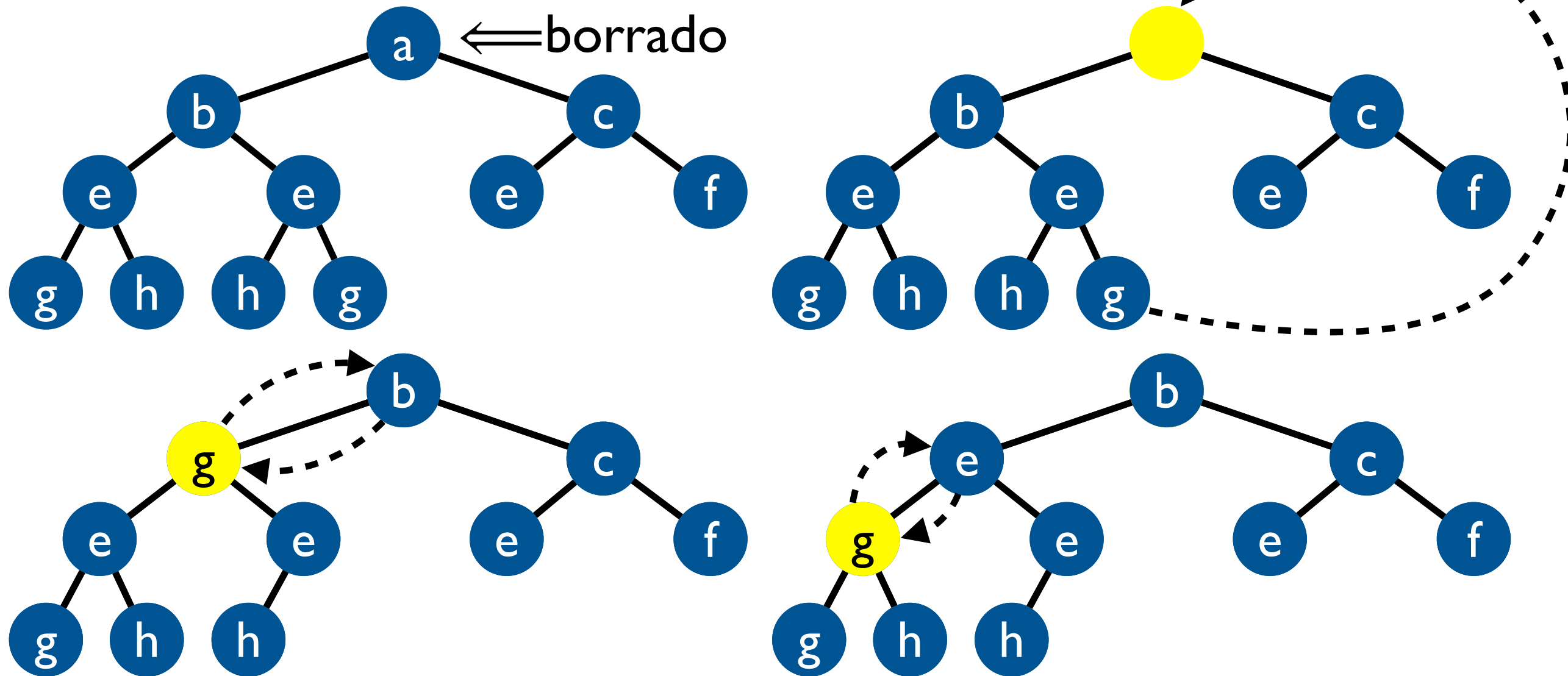
# Inserción en el APO



$O(\log_2 n)$

Estas operaciones pueden realizarse gracias a la idea de mantener las hojas del APO empujadas a la izquierda

# Borrado en el APO



$O(\log_2 n)$

Estas operaciones pueden realizarse gracias a la idea de mantener las hojas del APO empujadas a la izquierda

# Consideraciones

- Si inserciones y borrados tienen  $O(\log_2 n)$ , ¿utilizar un APO para ordenar es mejor o peor que el algoritmo Quicksort? Con ambos tendríamos  $O(n \log_2 n)$ ...
  - Con Quicksort (y con cualquier método habitual de ordenación), tenemos que reordenar si añadimos nuevos datos (o insertar ordenado).
  - Con Heapsort (usando un APO), cada inserción tiene sólo un coste de  $O(\log_2 n)$  y los datos estarán "ordenados" (mejor dicho, preparados para obtenerse de forma ordenada).
- Heapsort es particularmente útil si el conjunto de datos es muy dinámico (inserciones y/o borrados)
- ¿Qué otra estructura de datos podemos implementar con un APO? (Pista: la habéis estudiado/usado este curso)

# Ejercicio propuesto

- Sí, lo habéis adivinado (o no): podemos usar un APO para implementar una cola con prioridad.
- Recuerda, las operaciones que debe ofrecer una cola con prioridad son:
  - Frente: devuelve el elemento del frente
  - Poner: añade un elemento con la prioridad asociada
  - Quitar: elimina el elemento del frente
  - Vacía: indica si la cola está vacíaademás de los métodos básicos (constructores por defecto y de copia, destructor y operador de asignación).
- Podemos obviar la operación Prioridad\_Frente si seguimos la filosofía de la clase `priority_queue` de la STL: dejar en manos del usuario de la clase la definición del tipo base, al que sólo se le exige que tenga definido el operador `<`



```
/* Fichero: APO.h */
```

```
#ifndef __APO_h__  
#define __APO_h__
```

```
/**
```

```
@memo T.D.A. APO
```

```
@doc Definición:
```

Una instancia  $a$  del tipo de dato abstracto  $Apo$  sobre un dominio  $Tbase$  es un árbol binario con etiquetas en  $Tbase$  y un orden parcial que consiste en que la etiqueta de un nodo es menor o igual que la de sus descendientes. Para poder gestionarlo, debe existir la operacin  $menor(<)$  para el tipo  $Tbase$ .

```
template <class Tbase>
```

```
class Apo {  
private:
```

```
/**
```

```
@name Implementación de T.D.A. ABB
```

```
@memo Parte privada. */
```

```
Tbase *vec;
```

```
/**
```

```
@memo Matriz de elementos
```

```
@doc En la matriz vec se almacenan los ele-  
mentos del Apo. */
```

```
int nelementos;
```

```
/**
```

```
@memo Número de elementos
```

```
@doc Este entero almacena el número de el-  
ementos del Apo, es decir, el Apo se alma-  
cena en las posiciones desde la 0 a nelementos-  
1. */
```

```
int Maxelementos;
```

```
/**
```

*@memo* Capacidad de almacenamiento

*@doc* En este entero se indica la cantidad de posiciones que hay reservadas en la matriz *vec*, que obviamente tiene que ser mayor o igual a *nelementos* \*/

```
void expandir(int nelem);
```

→ **resize**

```
/**
```

*@memo* Aumenta la capacidad de almacenamiento

*@param* nelem: Nuevo número de casillas reservadas para la matriz *vec*  $nelementos \leq nelem$

*@doc* Asigna un bloque de memoria para *nelem* elementos para almacenar la matriz *vec*. Al terminar, *vec* apunta a un bloque con esa capacidad, conservando el contenido anterior y el miembro *Maxelementos* vale *nelem*. Sin embargo, si *nelem* es menor o igual que *Maxelementos* no hace nada. \*/



**/\*\* @name Invariante de la representación**

**@memo** Inv. de Apo

**@doc** El invariante para un apo  $a$  es

$a.nelementos \leq a.Maxelementos$

$a.vec$  apunta a un bloque de memoria reservada de  $a.Maxelementos$ .

$\forall i, j$  tal que  $0 \leq i < j < a.nelementos$  y ( $j=2*i+1$  o  $j=2*i+2$ )  $a.vec[i] \leq a.vec[j]$

**\*/**

**/\*\* @name Función de abstracción**

**@memo** F.A. de Apo.

**@doc**

Sea  $T$  un árbol parcialmente ordenado sobre el tipo  $Tbase$ . Diremos que el subárbol a partir de la posición  $i$  es:

Si  $0 \leq i < nelementos$  el que tiene como elemento raíz el valor  $T.vec[i]$  y como subárboles izquierda y derecha, los subárboles a partir de  $i*2+1$  y  $i*2+2$ . En caso contrario ( $i \geq nelementos$ ), el árbol vacío.

El árbol  $T$ , del conjunto de valores en la representación se aplica al árbol a partir de la posición 0.

**\*/**

**public:**

*@name* Operaciones de T.D.A. Apo

*@memo* Operaciones sobre Apo

*\*/*

**Apo();**

*/\*\**

*@memo* Constructor por defecto

*@doc* Reserva los recursos e inicializa el árbol a vacío {}. La operación se realiza en tiempo  $O(1)$ . *\*/*

**Apo(int tam);**

*/\*\**

*@memo* Constructor con tamaño

*@param* tam: número de elementos que se espera pueda llegar a tener el árbol.

*@doc* Reserva los recursos e inicializa el Arbol a vacío. La operación se realiza en tiempo  $O(tam)$ . *\*/*



```
Apo (const Apo<Tbase>& a);
```

```
/**
```

```
@memo Constructor de copia
```

```
@param a: Apo a copiar
```

```
@doc Construye el árbol duplicando el contenido de a en el árbol receptor. La operación se realiza en tiempo  $O(n)$ , donde n es el número de elementos de a. */
```

```
~Apo();
```

```
/**
```

```
@memo Destructor
```

```
@doc Libera los recursos ocupados por el árbol receptor. La operación se realiza en tiempo  $O(n)$  donde n es el número de elementos del árbol receptor. */
```

```
Apo<Tbase>& operator=(const Apo<Tbase>  
&a);
```

```
/**
```

```
@memo Asignación
```

```
@param a: Apo a copiar
```

```
@return Referencia al árbol receptor.
```

```
@doc Asigna el valor del árbol duplicando el  
contenido de a en el árbol receptor. La op-  
eración se realiza en tiempo  $O(n)$ , donde  $n$  es  
el número de elementos de a. */
```

```
const Tbase& minimo() const;
```

```
/**
```

```
@memo Mínimo elemento almacenado
```

```
@return referencia constante al elemento que  
es mínimo en el árbol receptor.
```

```
@precondition El árbol no está vacío
```

```
@doc Devuelve una referencia al elemento más  
pequeño de los almacenados en el árbol recep-  
tor. La operación se realiza en tiempo  $O(1)$ . */
```



```
void borrar_minimo();
```

```
/**
```

```
@memo Elimina el mínimo
```

```
@precondition El árbol no está vacío
```

```
@doc Elimina del árbol receptor el elemento  
más pequeño. La operación se realiza en tiempo  
 $O(\log n)$ . */
```

```
void insertar (const Tbase& el);
```

```
/**
```

```
@memo Insertar un elemento
```

```
@param el: nuevo elemento a insertar
```

```
@doc Inserta el elemento el en el árbol recep-  
tor. La operación se realiza en tiempo  $O(\log n)$ . */
```

```
void clear();
```

```
/**
```

```
@memo Borra todos los elementos
```

```
@doc Borra todos los elementos del árbol receptor. Cuando termina, el árbol está vacío. La operación se realiza en tiempo  $O(1)$ . */
```

```
int size() const;
```

```
/**
```

```
@memo Número de elementos
```

```
@return El número de elementos del árbol receptor.
```

```
@doc La operación se realiza en tiempo  $O(1)$ . */
```

```
bool empty() const;
```

```
/**
```

```
@memo Vacío
```

```
@return Devuelve true si el número de elementos del árbol receptor es cero, false en otro caso.
```

```
@doc La operación se realiza en tiempo  $O(1)$ . */
```

```
};
```

```
/* IMPLEMENTACION DE LAS FUNCIONES *
```

```
#include <cassert>
```

```
/*-----*/
```

```
// FUNCIONES PRIVADAS
```

```
/*-----*/
```

```
template <class Tbase>
```

```
void Apo<Tbase>::expandir (int nelem)
```

```
{
```

```
    Tbase *aux;
```

```
    int i;
```

```
    if (nelem>Maxelementos) {  
        aux= new Tbase[nelem];  
        for (i=0;i<nelementos;i++)  
            aux[i]=vec[i];  
        delete[] vec;  
        vec= aux;  
        Maxelementos=nelem;
```

```
    }
```

```
}
```

```
/*-----*/
```

```
/*-----*/  
// FUNCIONES PUBLICAS  
/*-----*/
```

```
template <class Tbase>  
Apo<Tbase>::Apo()
```

```
{
```

```
    vec= new Tbase;  
    nelementos= 0;  
    Maxelementos= 1;
```

```
}
```

```
/*-----*/
```

```
template <class Tbase>  
Apo<Tbase>::Apo(int tam)
```

```
{
```

```
    vec= new Tbase[tam];  
    nelementos= 0;  
    Maxelementos= tam;
```

```
}
```

```
/*-----*/
```



```
template <class Tbase>
Apo<Tbase>::Apo (const Apo<Tbase>& a)
{
    int i,aux;
```

```
    aux=a.nelementos;
    if (aux==0) aux=1;
    vec= new Tbase[aux];
    nelementos= a.nelementos;
    Maxelementos= aux;
    for (i=0;i<nelementos;i++)
        vec[i]= a.vec[i];
```

```
}
```

```
/*-----*/
```

```
template <class Tbase>
inline Apo<Tbase>::~~Apo()
```

```
{
```

```
    delete[] vec;
```

```
}
```

```
/*-----*/
```

```
template <class Tbase>
```

```
Apo<Tbase>& Apo<Tbase>::operator=(const
                                Apo<Tbase> &a)
```

```
{
```

```
    int i;
```

```
    if (this!=&a) {
```

```
        delete[] vec;
```

```
        vec= new Tbase[a.nelementos];
```

```
        nelementos= a.nelementos;
```

```
        Maxelementos= a.nelementos;
```

```
        for (i=0;i<nelementos;i++)
```

```
            vec[i]= a.vec[i];
```

```
    }
```

```
    return *this;
```

```
}
```

```
/*-----*/
```

```
template <class Tbase>
inline const Tbase& Apo<Tbase>::minimo() const
{
    assert(nelementos>0);
    return vec[0];
}
/*-----*/
```

```
template <class Tbase>
void Apo<Tbase>::insertar (const Tbase& el)
{
```

```
    int pos;
    Tbase aux;
```

```
    if (nelementos==Maxelementos)
        expandir(2*Maxelementos);
```

```
    nelementos++;
    pos=nelementos-1;
    vec[pos]=el;
```

```
    while((pos>0) && (vec[pos]<vec[(pos-1)/2])) {
        aux= vec[pos];
        vec[pos]=vec[(pos-1)/2];
        vec[(pos-1)/2]= aux;
        pos= (pos-1)/2;
    }
```

```
}
```

```
/*-----*/
```



```
template <class Tbase>
void Apo<Tbase>::borrar_minimo()
{
```

```
    int pos,pos_min, ultimo;
    bool acabar;
    Tbase aux;
```

```
    assert(nelementos>0);
    vec[0]=vec[nelementos-1];
    nelementos--;
```

```
    if (nelementos>1) {
```

```
        ultimo= nelementos-1;
        pos=0;
        acabar= false;
```

```
        while(pos<=(ultimo-1)/2 && !acabar) {
            if (2*pos+1==ultimo)
                pos_min=2*pos+1;
            else if (vec[2*pos+1]<vec[2*pos+2])
                pos_min= 2*pos+1;
            else pos_min= 2*pos+2;
```

```
        if (vec[pos_min]<vec[pos]){  
aux= vec[pos]; // swap..se puede mejorar  
vec[pos]=vec[pos_min];  
vec[pos_min]= aux;  
pos=pos_min;  
        }  
        else acabar= true;
```

```
    }
```

```
}
```

```
}
```

```
/*-----*/
```

```
template <class Tbase>
inline void Apo<Tbase>::clear()
```

```
{
```

```
    nelementos=0;
```

```
}
```

```
/*-----*/
```

```
template <class Tbase>
inline int Apo<Tbase>::size() const
```

```
{
```

```
    return nelementos;
```

```
}
```

```
/*-----*/
```

```
template <class Tbase>
inline bool Apo<Tbase>::empty() const
```

```
{
```

```
    return nelementos==0;
```

```
}
```

```
/*-----*/
```

```
#endif
```

```
/*  Fin Fichero: Apo.h  */
```

# Algoritmo de ordenação Heapsort

```
#include <ctime>
#include <iostream>
#include <ap0.h>
```

```
int main()
```

```
{
```

```
    Ap0<int> a(400.000);
```

```
    int i;
```

```
    srand(time(NULL));
```

```
    for (i=0; i<400.000; i+=2)
```

```
        a.inserir(i, (1.000.0 * rand() / RAND_MAX));
```

```
    while (!a.empty()) {
```

```
        cout << a.minimo() << ' ';
```

```
        a.borrar_minimo();
```

```
    }
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```