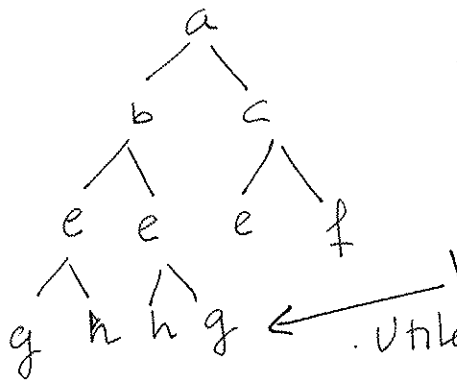
LECCION 21: (APO$)

Arboles Parcialmente Ordenados (APO)

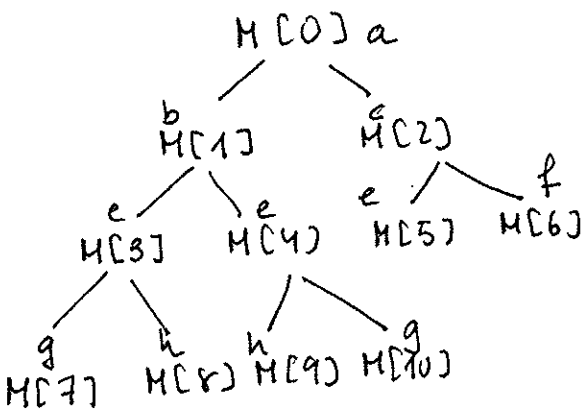
Definición → Un arbol binario se dice que es un APO si cumple la condición de que la etiqueta de cada nodo es menor o igual que la etiqueta de sus hijos y tiene completo todos los niveles excepto el último que tiene los vacios en la derecha.



Funciones
- Inserción de un elemento
- Borrado del mínimo

hojas empujadas a la izq.
. Utiles para ordenación.

Representación optima ⟹ HEAP (MONTON) es un vector en el que el APO se guarda por niveles. M[0] está la raiz M[1] y M[2] hijq e hder resp. Para el nodo K-esimo sus M[K] sus descendientes si existe estan M[2K+1] y M[2K+2]



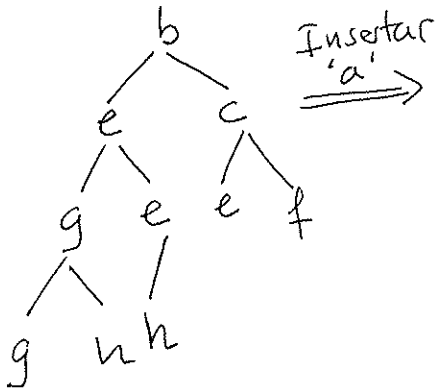
0	1	2	3	4	5	6	7	8	9	10
a	b	c	e	e	e	f	g	h	h	g

Dado un nodo M[K] su padre estará en $M\left(\left\lfloor \frac{K-1}{2} \right\rfloor\right)$ $\forall$ $K>0$

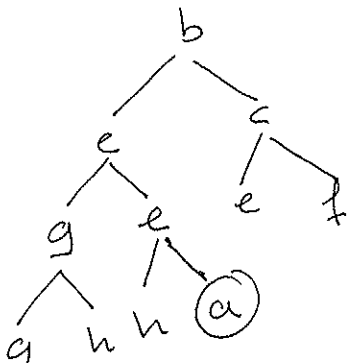
2APO

LECCIÓN 21: APO

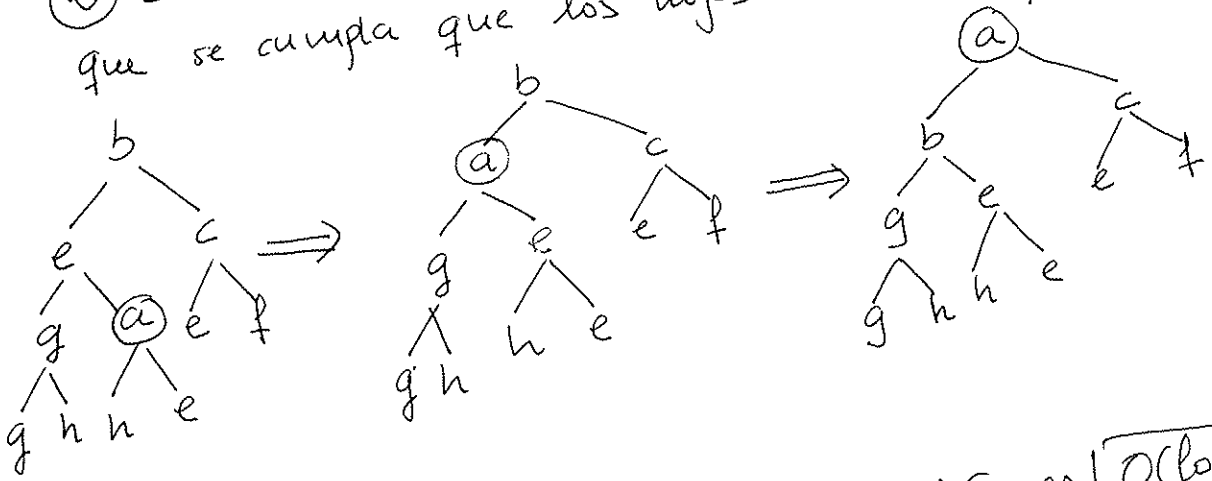
Proceso de Inserción



(1) Buscar en el último nivel donde se inserta sin romper la condición que las hojas están empujadas hacia la izquierda



(2) Ir intercambiando la etiqueta 'a' con el padre hasta que se cumpla que los hijos tienen etiquetas mayores.



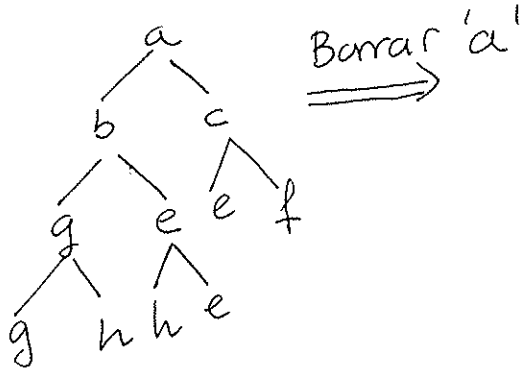
Eficiencia del proceso de inserción es $O(\log_2(n))$

LECCION 21 : APO

Proceso de Insercción

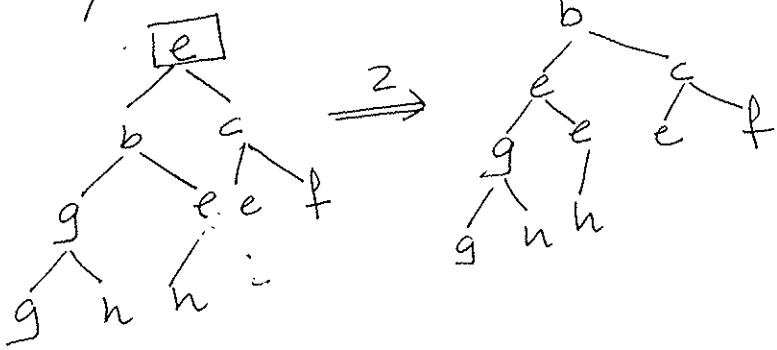
```
int * datos; // representación del APO
int nelementos; // numero de elementos del APO
  :
// insertamos la nueva etiqueta x al final
datos[nelementos] = x;
nelementos++;

// Empieza el proceso de intercambio
int pos = nelementos - 1;
while (pos > 0 && datos[pos] < datos[((pos-1)/2)]) {
                                  (el padre)
        swap(datos[pos], datos[((pos-1)/2)]);
        pos = (pos-1)/2;
}
```

Borrado (siempre se borra la raíz)



Borrar 'a' ⟹

(1) Buscamos el siguiente que está en el último nivel más a la dcha. y lo sustituimos por a.



(2) Haced bajar a [e] al nivel de los hijos. Parad cuando los hijos tengan etiquetas mayores. Intercambiamos [e] por el menor de sus hijos.

Eficiencia $O(\log_2(n))$

4APO

LECCIÓN 21: APO

Proceso de Borrado

```
datos[0] = datos[nelementos-1]; nelementos--;
ultimo = nelemento-1;
bool acabar = false
pos = 0;
while (pos <= (ultimo-1)/2) && !acabar){   // el padre del penúltimo

    int pos_min;
    if (pos*2+1 == ultimo) //si es el hijo a la izquierda
        pos_min = ultimo; //solo un hijo
    else //existen dos hijos quién es menor?
     if (datos[2*pos+1] < datos[2*pos+2])
                pos_min = 2*pos+1;

    else
            pos_min = 2*pos+2;

    //comparamos con la etiqueta que la tenemos en pos.
     if (datos[pos] > datos[pos_min]){
                swap(datos[pos], datos[pos_min]);
                pos = pos_min;

     }
     else
        acabar = true;

}
```

5APO

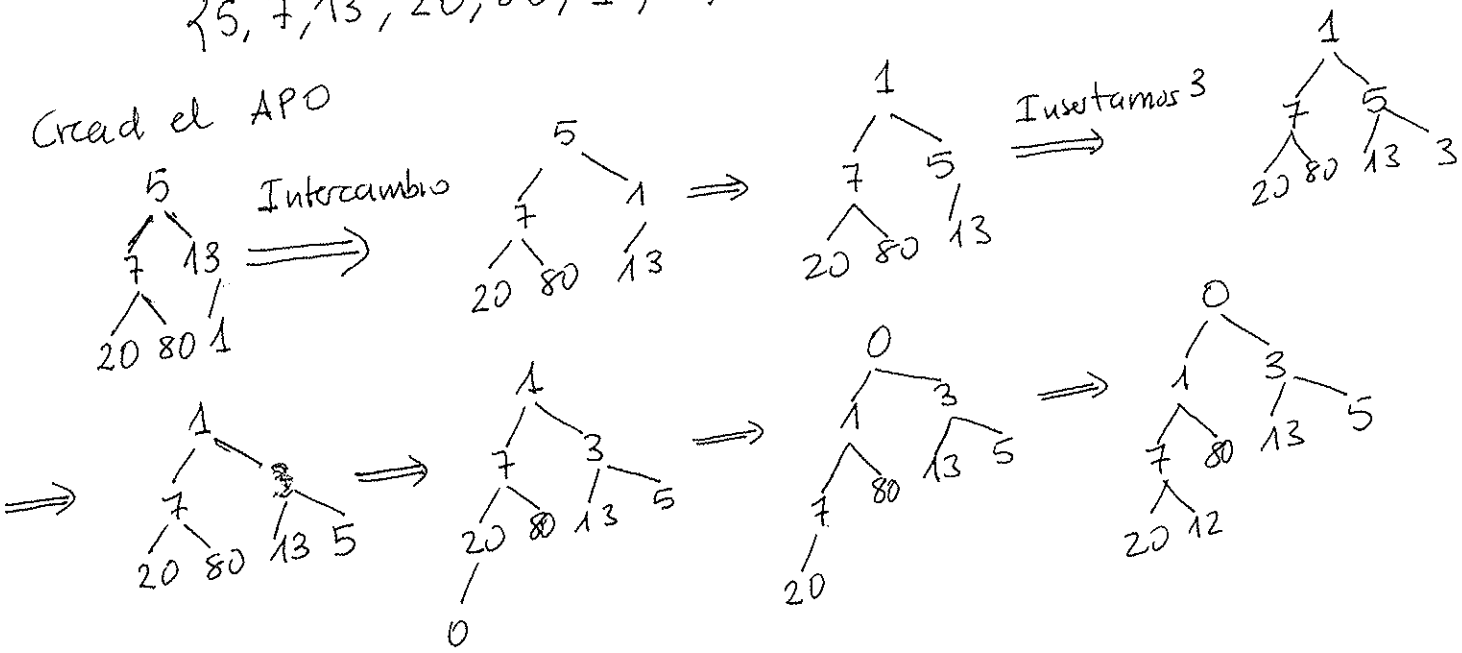
LECCIÓN 21: APO

Ejercicio

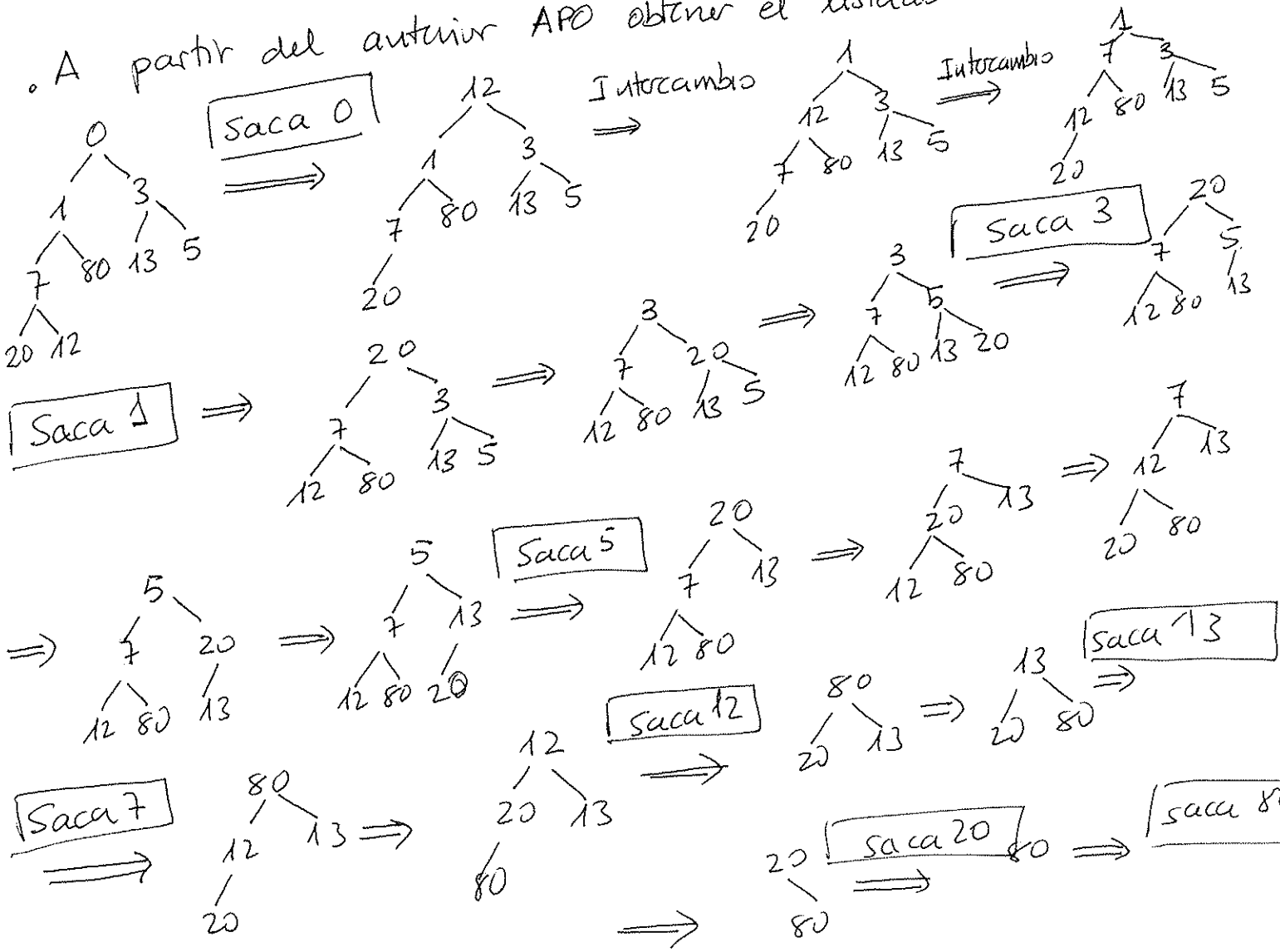
Supone que tenemos las siguientes claves:

{5, 7, 13, 20, 80, 1, 3, 0, 12}

Crear el APO

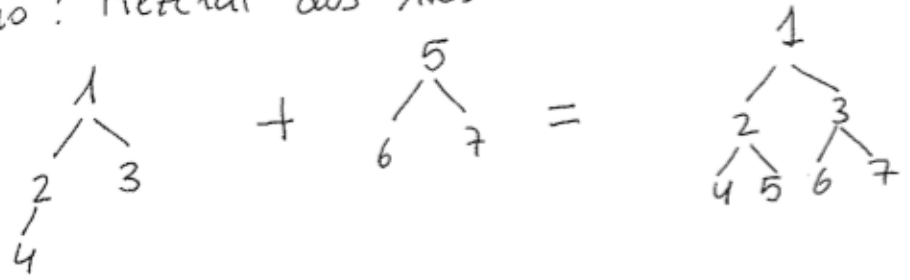


- A partir del anterior APO obtener el listado ordenado



6 APO

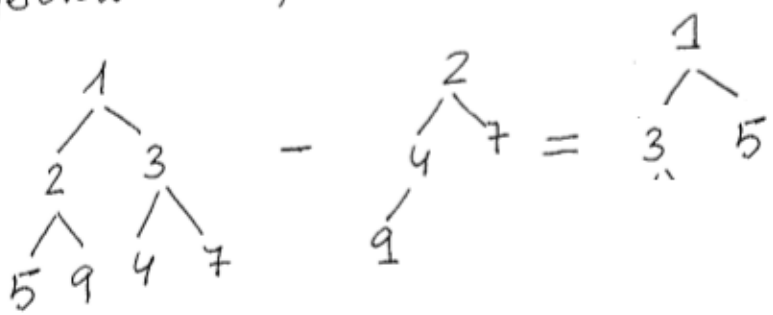
Ejercicio: Mezclar dos APOs



- Suponiendo que tenemos la clase APO con las funciones { vacío, mínimo, borrar_mínimo, insertar.

```
template <class T>
APO<T> operator+ (const APO<T> &apo1, const APO<T> &apo2){
    APO<T> nuevo(apo1);
    APO<T> aux(apo2);
    while(!aux.vacio()){
        T v = aux.minimo();
        aux.borrar_minimo();
        nuevo.insertar(v);
    }
    return nuevo;
}
```

Ejercicio: Implementar la diferencia

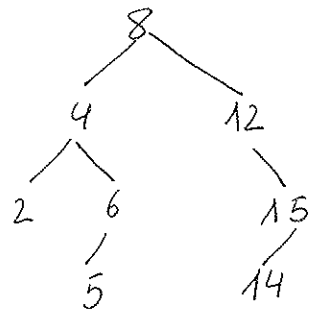


(ABB)

LECCIÓN 21: Arboles Binarios Búsqueda

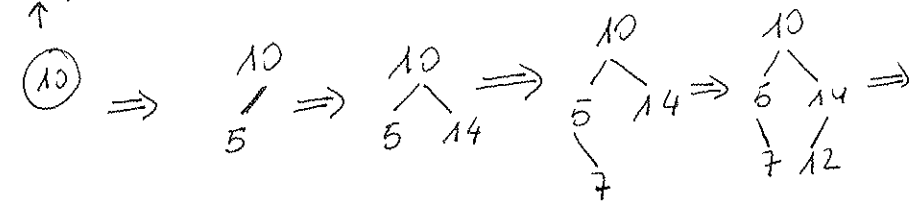
Arboles Binarios de Búsqueda (ABB)

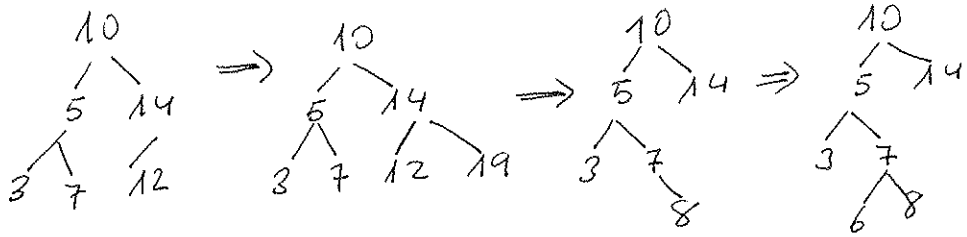
Es un árbol binario, con las etiquetas de los nodos ordenados de forma que el elemento situado en un nodo es mayor que todos los que se encuentran en el subarbol izquierdo y menor que los que se situan en el subarbol derecho.



Cómo se construye

10, 5, 14, 7, 12, 3, 19, 8, 6 ¿





—

- Se suponen que no hay elementos repetidos. Las búsquedas se realizan en $O(\log_2(n))$ donde n es el nº de nodos o etiquetas
- Los procesos de insercion y borrado son más complicados.
- El recorrido en inorden de un ABB da la ordenacion de todos los elementos.

```
template <class T>
struct info-nodo{
   T et;
   infonodo*padre*hizq*hder;
}
```

operacion ==, <, > sobre T tienen que estar definidas.

```
template <class T>
info-nodo*T* Buscar(infonodo<T> * n, T x)
{
  if (n != 0){
    if (n->et == x)
       return n;
    else
       if (n->et < x)
          return Buscar(n->hizq, x)
       else return Buscar(n->hder, x)
  }
  else return n;
}
```

2ABB

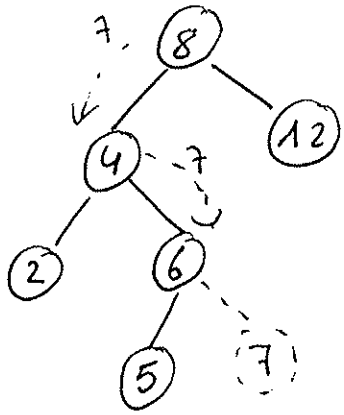
LECCION 21: ABB

Búsqueda sin usar recursividad

```
template <class T>
info_nodo <T> * Buscar(info_nodo <T> * n, T x){
        if (n==0)
            return n;
        else{
            info_nodo <T> *p=n;
            while (p!=0){
                if (p->et==x)
                    return p;
                else
                    if (p->et <x)
                        p= p->hder;
                    else
                        p=p->hizq;
            }
            return p;
        }
}
```

LECCION 21: A BB

- Buscar donde insertar el elemento x=7 en el siguiente árbol:



El proceso va comparando con las etiquetas del arbol empezando por la raiz hasta alcanzar un nodo que no tiene hder y x es mayor que el nodo o alcanzar un nodo que no tiene hizq y x es menor que el nodo.

```
template <class T>
bool Insertar (info_nodo <T> *&n, T x) {
    bool res=false;
    if (n==0){
        n= new info_nodo (x);
        return true;
    }else{
        if (n->et <x){
            res= Insertar (n->hder, x);
            if (res){
                n->hder->padre = n;
            }
            return res;
        }else
            if (n->et >x){
                res= Insertar (n->hizq, x);
                if (res){
                    n->hizq->padre=n;
                }
                return res;
            }else return false;
    }
}
```