Tema 5

Árboles Binarios de Búsqueda (BST)

Árboles Equilibrados (AVL)

Árboles Parcialmente Ordenados (POT)

Árboles binarios de búsqueda: Motivación

La búsqueda binaria es un proceso rápido de búsqueda de elementos en un vector ordenado O(log(n)).

Sin embargo, las inserciones y borrados son muy ineficientes (O(n)).

Los ABB (BST en inglés) son de utilidad para representar TDAs donde:

- búsqueda, inserción y borrado sean eficientes, idealmente en O(log(n))
- Permiten acceder a los elementos en orden.

BST: Definición.

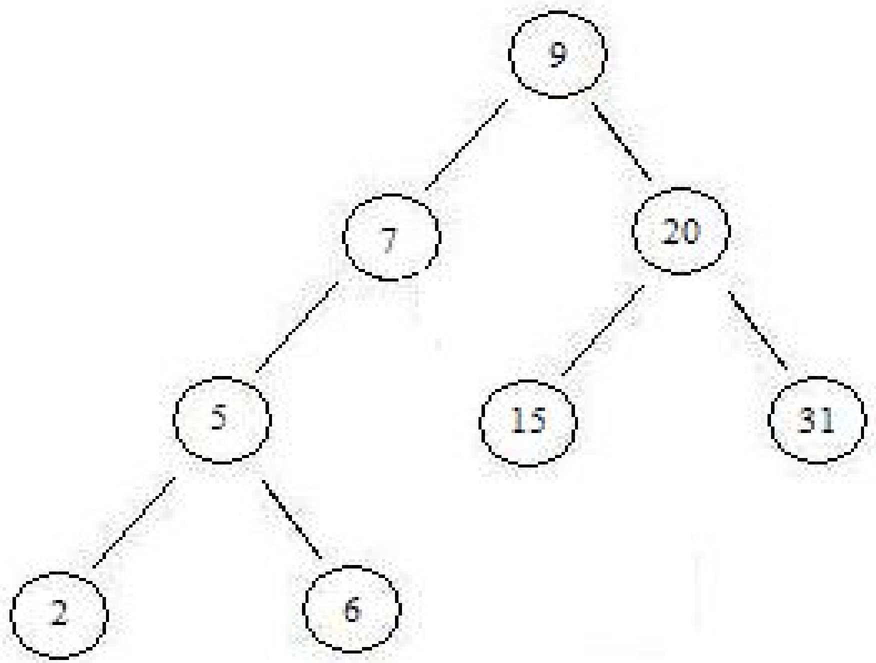
Árbol Binario de Búsqueda (BST):

- árbol binario verificando que
  - todos los elementos almacenados en el subárbol izquierdo de un nodo *n* son menores que el elemento almacenado en *n*,
  - todos los elementos del subárbol derecho de *n* son mayores (o iguales) que el elemento almacenado en *n*.

BST: Propiedades

- Un BST Es un tipo de dato orientado a búsqueda, es decir, que tan sólo tiene operaciones para crear, destruir, insertar etiquetas, borrar etiquetas y buscar etiquetas (aparte de un iterador)
- Si el árbol está equilibrado, las operaciones de inserción, búsqueda y borrado de un elemento se pueden realizar en orden O(log(n)) .
- El recorrido en inorden da el conjunto de etiquetas ordenado

BST: Ejemplo



BST y STL

- Métodos del BST:
  - `insert, find, erase, size, clear, empty, begin, end,`
- Como tal no existe un TDA BST en la STL. Sin embargo, un BST puede ser la base para implementar los contenedores asociativos
  - Set<T> o Multiset<T>
  - Map<Key,T> o Multimap<Key,T>

BST: Especificación

```
/**   BST::BST, insert, find, erase, size, clear,
  empty, begin, end, ~BST
 El TDA BST representa objetos que abstraen el
  concepto de Árbol Binario de Búsqueda. Es un tipo
  contenedor, cuyos componentes son Entradas
  formadas por dos partes {clave, valor}, de tipos
  Key y T, resp.
 Los objetos Key deben tener las operaciones:
 - bool operator<(const Key & c, const T & v);
 - bool operator==(const Key & c, const T & v);
 Los objetos T deben tener las operaciones:
 - bool operator==(const T & c, const T & v);
*/
```

MUY SIMILAR AL MAP

BST: Especificación (2)

```
/** @brief Constructor primitivo por defecto.
   Crea un BST vacío.
*/
BST();

/**  @brief Inserta o actualiza una entrada en el
  BST.
  @param entrada:  Entrada a insertar en el receptor.
  Si no existe una entrada con clave igual a
  entrada.first,  inserta entrada en el receptor.  Si
  ya  existe, su valor asociado es reemplazado por
  entrada.second.
*/
void insert(const pair<Key, T> & entrada);
```

BST: Especificación (3)

```
/**
   @brief Buscar una clave en el BST.
   @param clave:  clave que se busca.

   @return un iterador a la posición del
  BST con la clave `clave`, si está en
  el BST. end(), en otro caso.
*/
iterator find(const Key & clave);
```

BST: Especificación (4)

```
/**
   @brief Devuelve el número de entradas del
  receptor.
  @return  Número de entradas del receptor.
*/
size_type size() const;

/**
    @brief Vacía el receptor.
    Elimina todas las entradas del receptor.
*/
void clear();
```

BST: Especificación (5)

```
/**
    @brief Indica si el receptor está vacío.
     @return true: Si el receptor no tiene
  entradas; false, en otro caso
*/
bool empty() const;


/**  @brief Destructor.
  Liberar todos los recursos asociados al receptor.
*/
 ~BST();
```

BST: Especificación (6)

```
/**  @brief Posición de la primera entrada.
   @return Posición inicial del recorrido del
   receptor (posición del primer elemento).
*/
 iterator begin();
 const_iterator begin() cosnt;

/**   @brief Posición final del BST.
   @return Posición final del recorrido del receptor
   (posición posterior al último elemento).
*/
iterator end();
const_iterator end() const;
```

Iteradores sobre BST

Además de los clásicos, destacamos

```
/**
  @brief Obtener el elemento al que apunta el
  iterador.
  @pre El iterador receptor NO está al final del
  recorrido: (receptor) != end()
  @return  La entrada {clave, valor}
  correspondiente al dato de la posición actual del
  recorrido.
  */
  pair<Key, T> operator*() const;
```

Ejemplo uso BST

```
BST<string, string> arb;
  string palabras[NUM] = {"piedra", "tiza",... };
  string definiciones[NUM] = {"objeto muy duro",
  "objeto muy blando", ...};
  pair<string,strig> aux;
for (int i= 0; i<NUM; i++) {
    aux.first = palabras[i];
    aux.second = definiciones[i];
    arb.insert(aux);
   }
cout << "Num.datos: " << arb.size();
 BST<string, string>::iterator i;
 for (i = arb.begin(); i != arb.end(); ++i)
     cout << i->first << ": " << i->second << endl;
```

Ejemplo Uso BST: Ordenación

```
template <class T>
void ordenar(T v[], int num_datos)
{
  int i;
  BST<T,char> abb;

  for (i = 0; i < num_datos; i++)
    abb.insert(pair<T,char>(v[i], ''));

  BST<T,char>::iterator ite;
  for (ite = abb.begin(), i = 0;
         ite != abb.end(); ++ite, i++)
    v[i] = ite->first;
}
```

BST: Representación

- Vamos a utilizar como representación el árbol binario (bintree).

- Función de abstracción:

rep={arbolb, tama}

un árbol binario que identifica directamente al árbol binario de búsqueda.

Representación

```
template <typename Key, typename T>
class BST {

private:
   bintree<pair<Key,T> > arbolb;
   int tama;
}
```

Invariante de la representación

Un objeto válido de rep del TDA BST debe cumplir:

- Dados dos nodos n, m del árbol binario
  - n->first != m->first
- Para todo nodo n se verifica que:
    - m->first < n->first: para todo nodo m descendiente a la izquierda de n
  - n->first < u->first: para todo nodo u es descendiente a la derecha de n

Implementación

`iterator find (const tClave & k)`

- Descender por el árbol (hasta encontrar la clave o llegar al nodo nulo) comparando en cada nodo con la clave k.
- Si la clave almacenada en el nodo es igual a k, ¡¡la hemos encontrado !!
- Si la clave almacenada en el nodo es menor que k, se avanza al hijo dcha.
- Si la clave almacenada es mayor que k se avanza al hijo izda.

Implementación (2)

Usaremos una función auxiliar *encontrar*

```
/* Busca la clave e en el árbol; n referenciará al
  nodo donde se encontraría la clave y nodo_padre
  referenciará al padre de n.
   @return true si la clave pertenece al árbol,
  false en caso contrario.
*/
bool BST<Key,T>::encontrar(
  const Key & e,
  bintree<BST<Key,T>::entry>::node & n,
  bintree<BST<Key,T>::entry>::node & n_padre)
  const
```

Implementación (3)

```
bool BST<tClave,tDef>::encontrar(
       const tClave & clave,
       bintree<pair<tClave,tDef> >::node &n,
       bintree<pair<tClave,tDef>>::node &n_padre) const{
    bool encontrado = false;
    n=arbolb.root();
    n_padre= n.parent();
    while (!n.null()  && !encontrado) {
      if ( (*n).first==clave) // La clave ya aparece
           encontrado = true;
       else {
           ...
```

Implementación (4)

```
            ...
            n_padre = n.parent();
            if ( clave < (*n).first )
                n = n.left(); // Debe estar a la
                                      izq de n
             else // Debe estar a la dch    de n
                n = n.right();
        }
    }
    return encontrado;
}
```

Implementación (5)

```
iterator BST<tClave, tDef>::find(const
 tClave  &clave){
  bool  enc ;
  bintree<pair<tClave,tDef> >::node nodo,
 nodo_padre;
  enc = encontrar(clave, nodo, nodo_padre);
  if (enc == true) {
     BST<tClave,tDef>::iterator it(nodo);
        //debemos implementar este constructor
     return it;
  } else return  end();
}
```

Implementación (7)

```
void BST<tClave, tDef>::insert(const tClave &clave,
  const tDef &def)
{
 bintree<pair<tClave,tDef> >::node nodo, nodo_padre;
 pair<tClave,tDef> Entrada (clave,def);
 bool encontrado = encontrar(clave, nodo, nodo_padre);
  if (!encontrado)
    if (nodo_padre.null())  {   // Arbol vacio
          arbolb.setroot(Entrada);
    }  else {
          ...
```

Implementación (7)

```
...
if (clave < (*nodo_padre).first)
      arbolb.insert_left(nodo_padre, Entrada);
else
      arbolb.insert_right(nodo_padre, Entrada);
} else // Encontrada la clave, sustituimos la def.
      (*n).second = def;   //Esto es válido pues el
  operator*() del bintree<.>::node devuelve una
  referencia al par (T&) y no una copia (T)
}
```

Borrar valor del Árbol Binario (esquema)

Se busca el nodo,n, que contiene el valor a borrar.

- Existen 3 casos:
  - N es un nodo hoja: Se borra el nodo hoja (o raíz)
  - N tiene un unico hijo, h: Entonces h se situa como hijo del padre de n en la misma rama que ocupaba n.   (Subcaso: n es la raíz)
  - N tiene dos hijos:
    - Sea pred el nodo que contiene la etiqueta que precede a la etiqueta de n en el orden.
    - Se cambia la etiqueta de n por la de pred.
    - Se borra pred (es un caso simple).

Implementación

```
bool BST<tClave, tDef>::erase(const tClave & e)
{
bintree<pair<tClave,tDef> >::node nodo, nodo_padre;
bintree<pair<tClave,tDef> > Taux;
bool encontrado = encontrar(e, nodo, nodo_padre);
if (encontrado) {
 if (n.esHoja() { /
      // Primer caso, es un nodo hoja ....
} else if ( nodo.left().null()|| nodo.right().null() )
{
     // 1 hijo
}
else {
      // 2 hijos
}
```

Implementación (2)

```
if (n.esHoja() { // Primer caso, es un nodo hoja
   if (nodo_padre.null()// Es la raíz
        arbolb.clear();
   else {      // NO es la raíz
     if (nodo_padre.left()==nodo)
         arbolb.prune_left(nodo_padre,Taux);
     else arbolb.prune_right(nodo_padre, Taux);
     Taux.clear(); //Liberamos el arbol Taux
   }

 } // Fin de caso es hoja
```

Implementación (3)

```
else if ( nodo.left().null() || nodo.right().null() )
{ // 1 hijo
    if (!nodo.left().null() ) //El hijo esta en izq
        arbolb.prune_left(nodo, Taux);
    else  arbolb.prune_right(nodo, Taux);//en Dch

    if (nodo_padre.null() ) // Nodo es raiz
        arbolb = Taux; // Es copia, antes destr.
    else  if (nodo_padre.left()==nodo)
        arbolb.insert_left(nodo_padre, Taux);
    else  arbolb.insert_right(nodo_padre, Taux);

// Recordad que insert libera el hijo si existiese
  } //Fin caso 1 hijo
```

Implementación (5)

```
else { //Caso nodo tiene  2 hijos
  bintree<pair<tClave,tDef> >::node pred;
  pred =  nodo.Izq(); // Predecesor
  while (!pred.right().null() )//Busqueda pred
        pred = pred.right();

  pair<tClave,tDef> valor_pred(*pred);
  bintree<pair<tClave,tDef> >::node
  nodo_a_borrar = nodo;
  erase( (*pred).first); // Llamada recursiva
                           para borrar el predecesor,
  *nodo_a_borrar = valor_pred; // Sustituimos
                                  el valor del nodo
 } // Fin caso 2 hijos

}  // Fin de borrar
```

Iterando en Arboles Binarios….

```
Template <typename tClave, typename tDef> class BST
  {public: ...

   private:

      bintree<pair<tClave,tDef> > arbolb;

   public:

  class iterator {

   public: ....

   private:

   typename

   bintree<pair<tClave,tDef> >::inorder_iterator el_it;

};

};
```

Iterando en Arboles Binarios….

```
BST<tClave, tDef>::iterador BST<tClave,
  tDef>::begin()
{
  BST<tClave, tDef>::iterator it_ret;
  it_ret.el_it = arbolb.begin_inorder();
  return it_ret;
}
```

Iterando en Arboles Binarios... (2)

```
BST<tClave, tDef>::iterador  BST<tClave,
  tDef>::end()
{
  BST<tClave, tDef>::iterator  it_ret;
  it_ret.eliterador = arbolb.end_inorder();
  return it_ret;
}
```

Iterando en Arboles Binarios… (3)

```
BST<tClave, tDef>::iterador&
    BST<tClave,tDef>::iterator::operator++()
{   el_ite++;
    return *this;
}


pair<tClave,tDef>  BST<tClave,
  tDef>::iterator::operator*()
//OJO !!! operator* no cambia la clave
{
   return (*el_it);
}
```

Operator++ del inorder .....

OJO: Estamos considerando bintree !!!

template <typename T>

typename bintree<T>::inorder_iterator &

bintree<T>::inorder_iterator::operator++()

{ Casos:

Si n es nulo: Nada

Si n tiene hijo dcha:

El ss es el descendiente derecho mas a la izq

Si no tiene hijo dcha:

El ss es el primer ancestro no visitado

==> paso a la derecha

Operator++ del inorder .....

```
template <typename T>
typename bintree<T>::inorder_iterator &
bintree<T>::inorder_iterator::operator++()
{   //elnodo hace referencia al nodo actual
  if (elnodo.null())   return *this;


   if (!elnodo.right().null()) { //Tiene hijo derecho
      elnodo = elnodo.right();
      while (!elnodo.left().null())
        elnodo = elnodo.left();
  }
```

Operator++ del inorder .....

```
else {  //No tiene hijo derecho
      while (!elnodo.parent().null() &&
         elnodo.parent().right() == elnodo)
        elnodo = elnodo.parent();
      // Si (padre de elnodo es nodo_nulo), hemos
   terminado
      // En caso contrario, el siguiente node es el padre
      elnodo = elnodo.parent();
  }
  return *this;
}
```

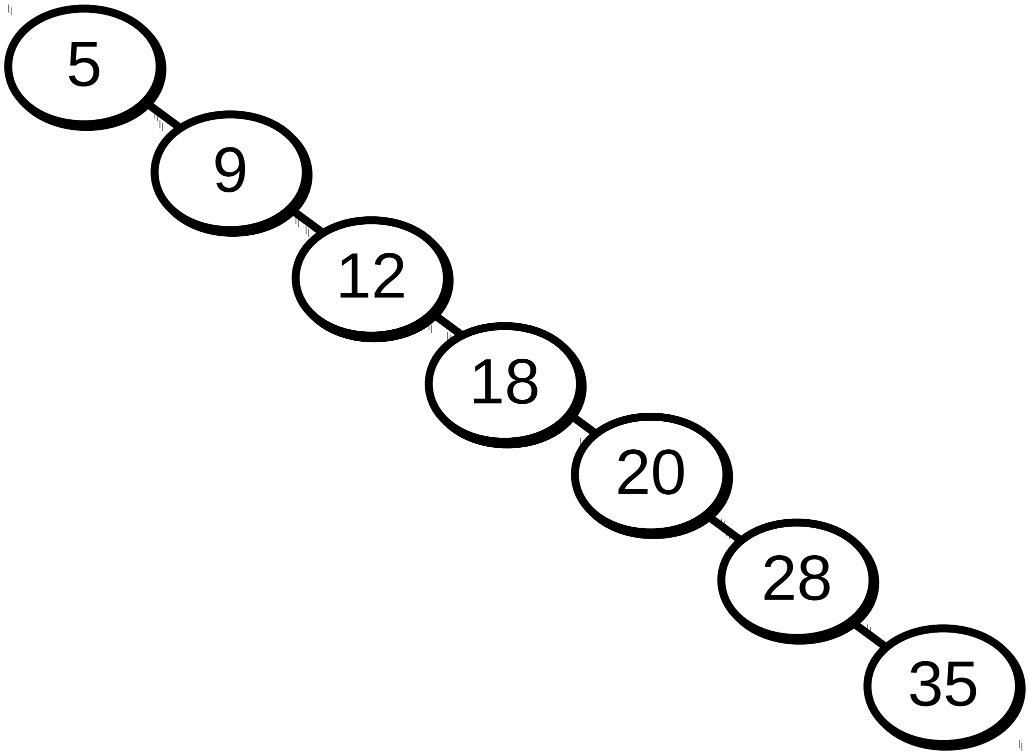
Arbol Equilibrados: AVL

Motivación:

- Las operaciones de Búsqueda, Inserción y Borrado en un BST genérico son de orden O(altura(arbol)) !!!

Arboles Equilibrados

Altura (arbol) es de O(log(n))



- **Conclusión:** Es necesario garantizar que el árbol está balanceado o equilibrado.
- **Condición de balanceo:** basada en número de nodos o en altura de subárboles.

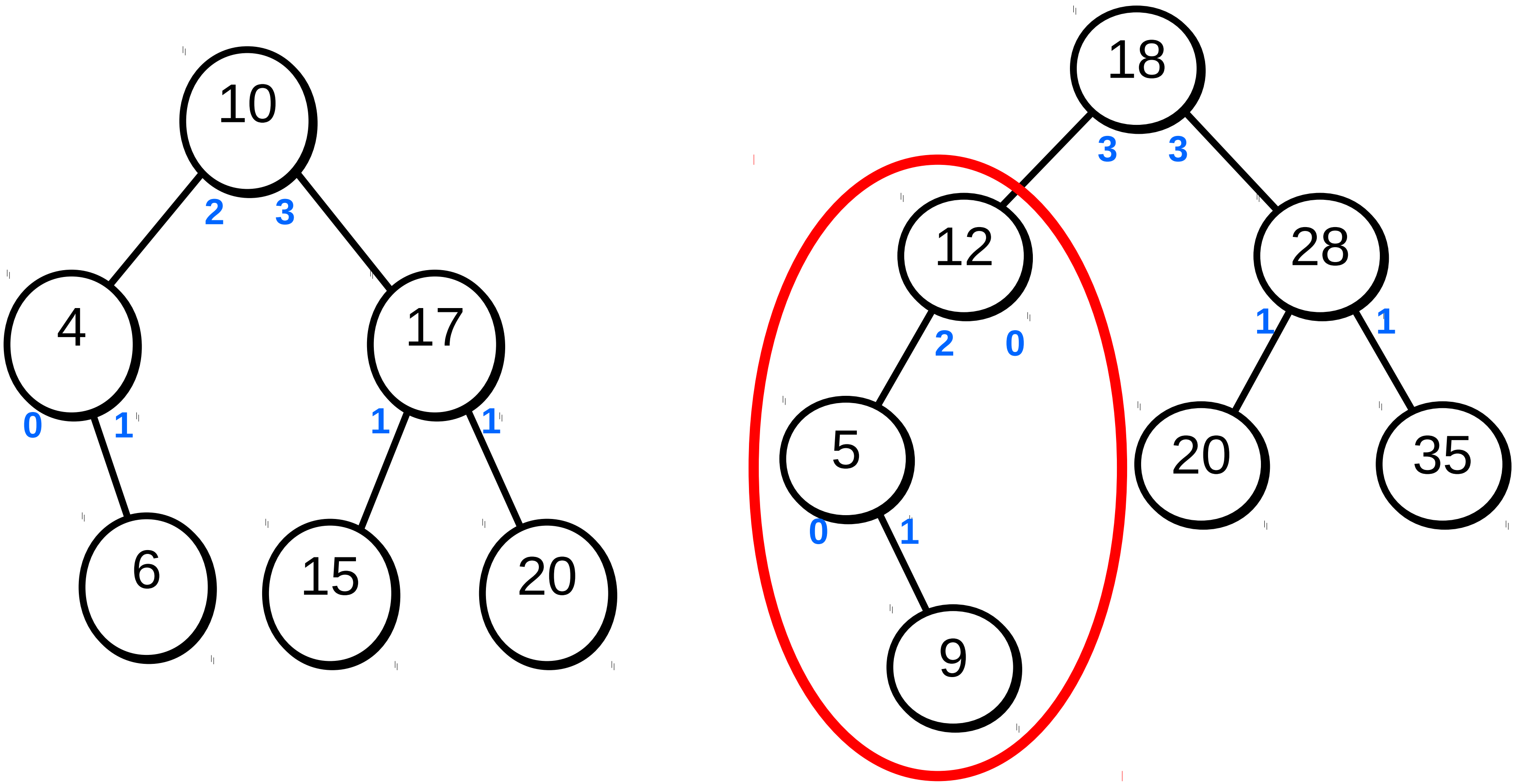
Arboles AVL

BST Equilibrado donde las operaciones de búsqueda e inserción y borrado tienen un orden de eficiencia logarítmico.

para cada nodo se cumple la condición de **AVL (Adelson-Velskii y Landis)**: :

la diferencia de la altura de sus dos hijos es como mucho de una unidad.

Ejemplos de AVL

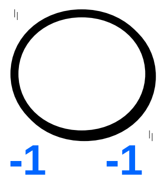


Análisis de Eficiencia

- ¿Cuánto será el tiempo de ejecución de la búsqueda en un AVL en el peor caso, para **n** nodos?
- El tiempo será proporcional a la altura del árbol.
- **Cuestión:** ¿Cuál es la máxima altura del árbol para **n** nodos?
- Le **damos la vuelta** a la pregunta: ¿Cuál es el mínimo número de nodos para una altura **h**?

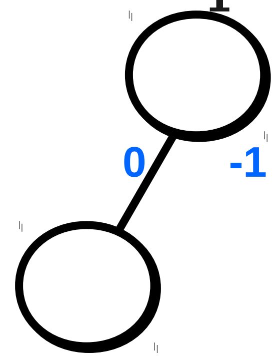
Análisis de Eficiencia:Peor caso de AVL

- **N(h)**: Menor número de nodos para altura **h**.



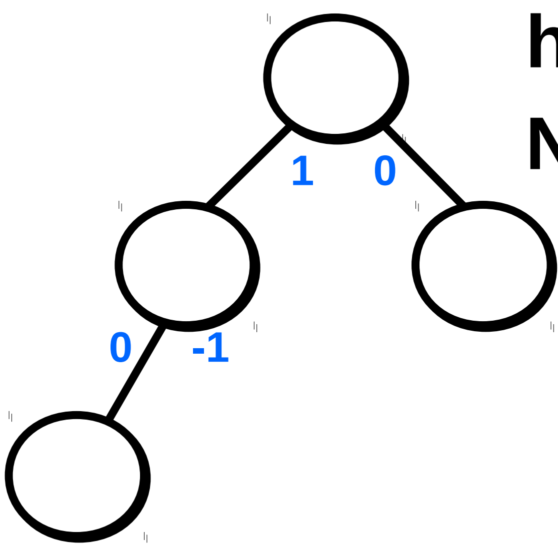
h= 0

N(0)= 1



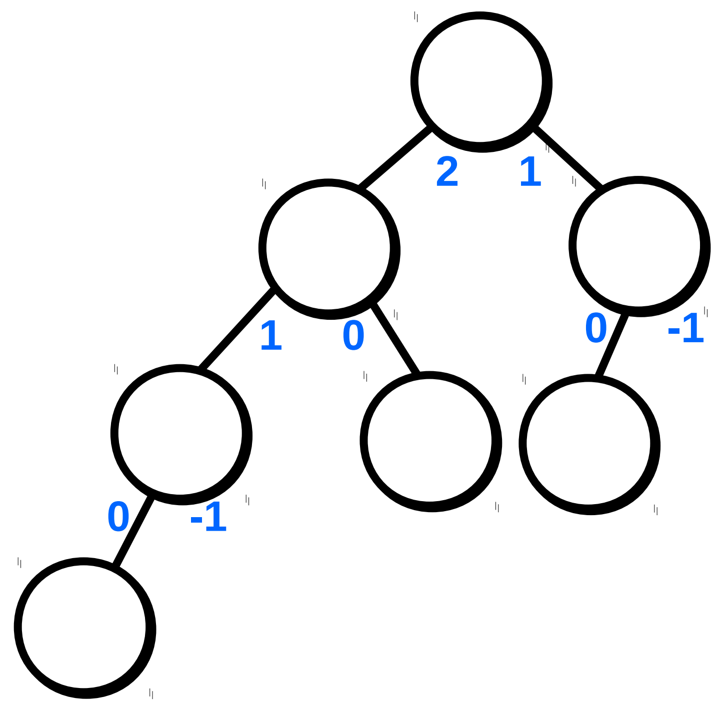
h= 1

N(1)= 2



h= 2

N(2)= 4

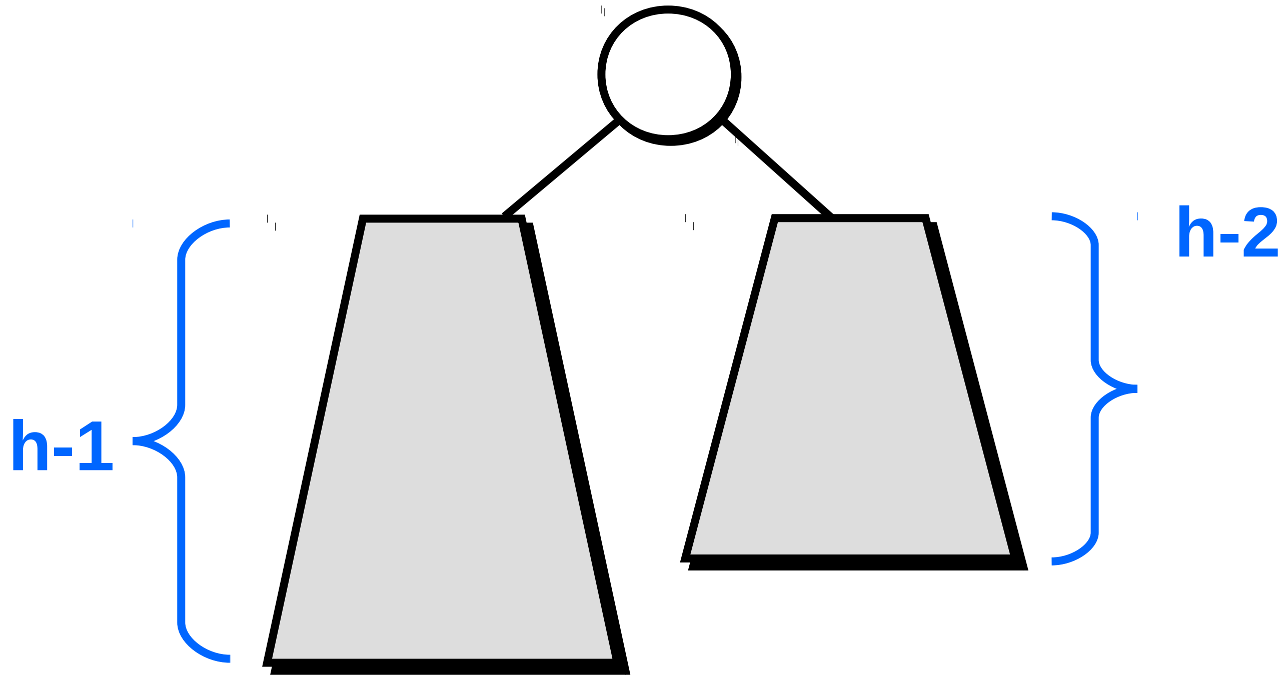


h= 3

N(3)= 7

Análisis de Eficiencia:Peor caso de AVL

- **Caso general.**



- N(h) = N(h-1) + N(h-2) + 1
- Sucesión parecida a la de **Fibonacci**.
- **Solución:** $N(h) = C \cdot 1{,}62^{h} + \ldots$

Análisis de Eficiencia:Peor caso de AVL

- **Mínimo número de nodos para altura h:**
  $N(h) = C \cdot 1{,}62^h + \ldots$

- **Máxima altura para n nodos:**
  $h(N) = D \cdot \log_{1,62} n + \ldots$

- **Conclusión:**
  – En el peor caso, la altura del árbol es **O(log n)**.
  – Por lo tanto, la búsqueda es **O(log n)**.
  – Inserción y eliminación serán de **O(log n)** si el **rebalanceo** se puede hacer en **O(1)**.

Operaciones sobre un AVL

- La especificación coincide con la del BST
- La **búsqueda** **find(const T & x)** en un AVL es exactamente igual que sobre un ABB.
- La **inserción** insert**(const T & x)**y **eliminación** erase(const T & x) son también como en un ABB, pero después de insertar o eliminar hay que comprobar la condición de balanceo.

Representación nodo AVL

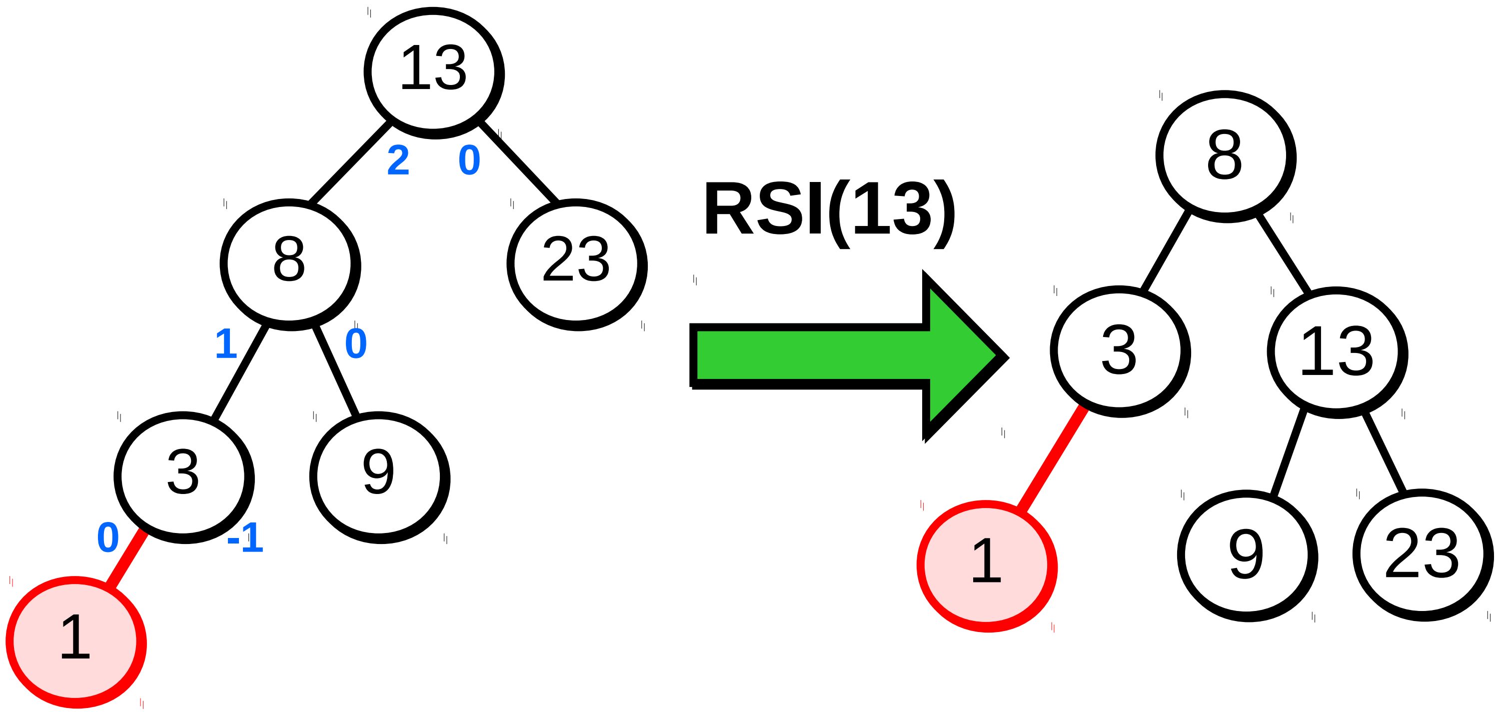
En cada nodo hay que almacenar la altura del subárbol (en concreto la diferencia entre alturas).

altura	izq	T	der
h		x	

- Inserción o eliminación normal
- Subir hacia a la raíz comprobando por los nodos que pasala condición de balanceo.
- Si no se cumple, **rebalancear** el árbol.

Operación de inserción en AVL

- Inserción normal como en un ABB.
- En cada nodo **A** en el camino a la raíz,
  - Si |Altura(A→izq) – Altura(A→der)|>1 entonces rebalancear.

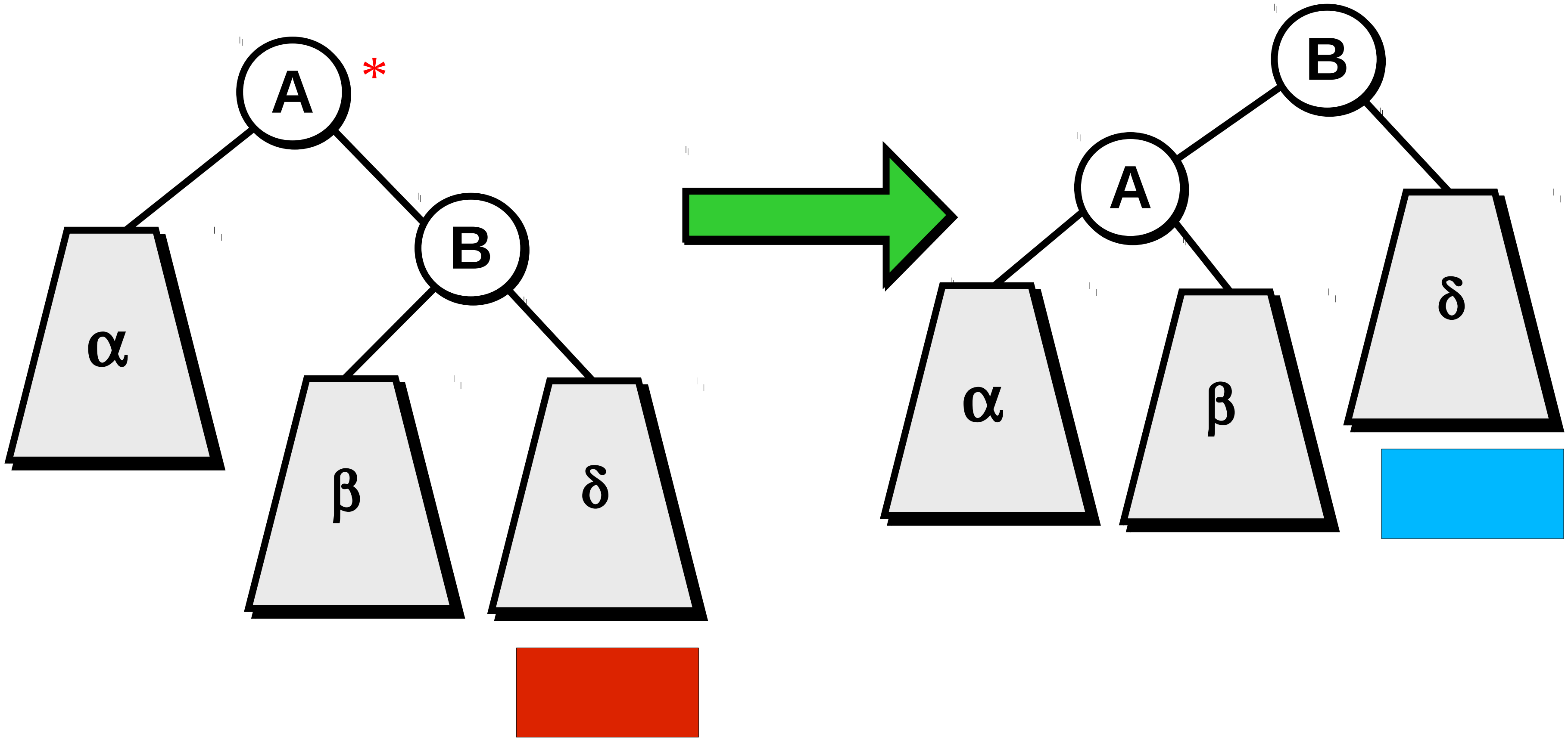


Rotaciones en un AVL

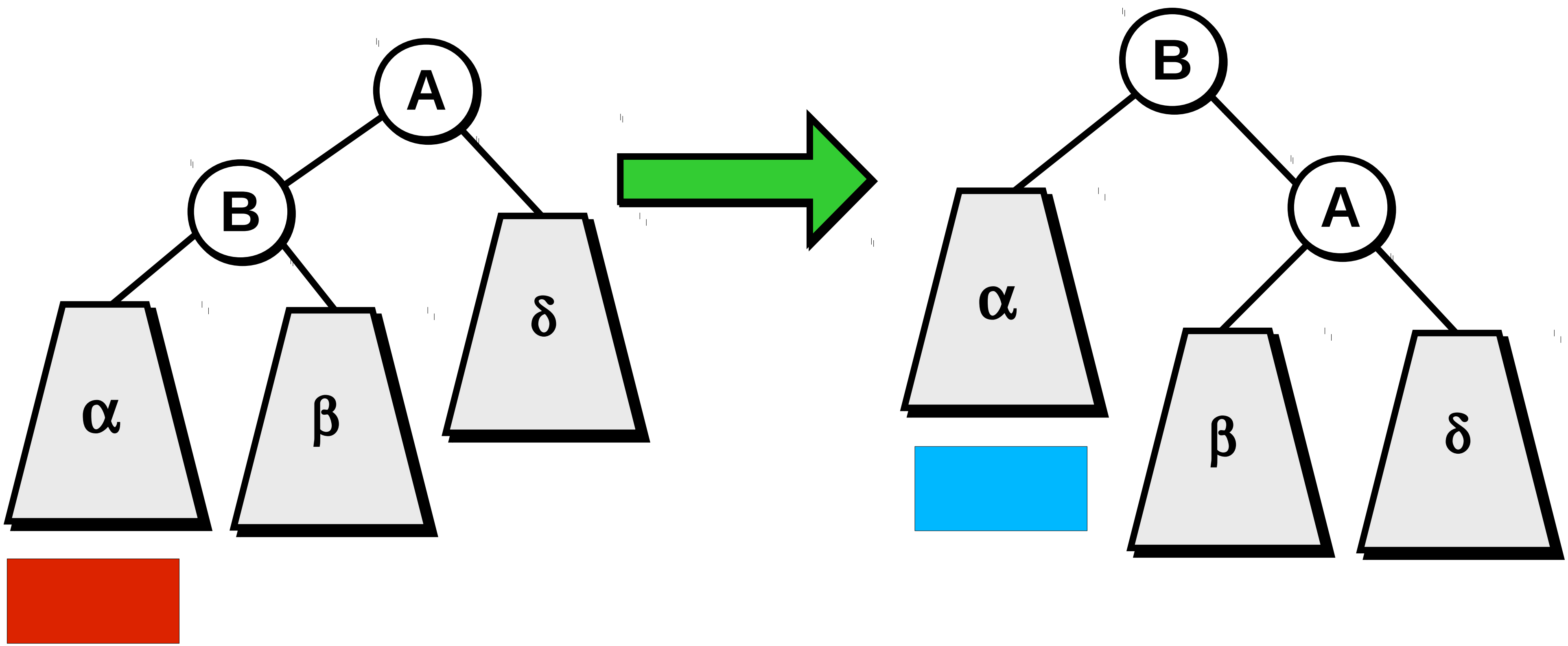
Los rebalanceos en un AVL hacen uso de operaciones conocidas como **rotaciones en ABB**.

- **Rotación:** cambiando algunos punteros, obtener otro árbol que siga siendo un ABB.
- 4 tipos de rotaciones
  - **RSD(n). Rotación simple derecha sobre n**
  - **RSI(n). Rotación simple izquierda sobre n**
  - **RDDI(n).Rotación doble derecha+izquierda**
  - **RDID(n). Rotación doble Izquierda+derecha**

Rotaciones en un AVL: RSI(A)



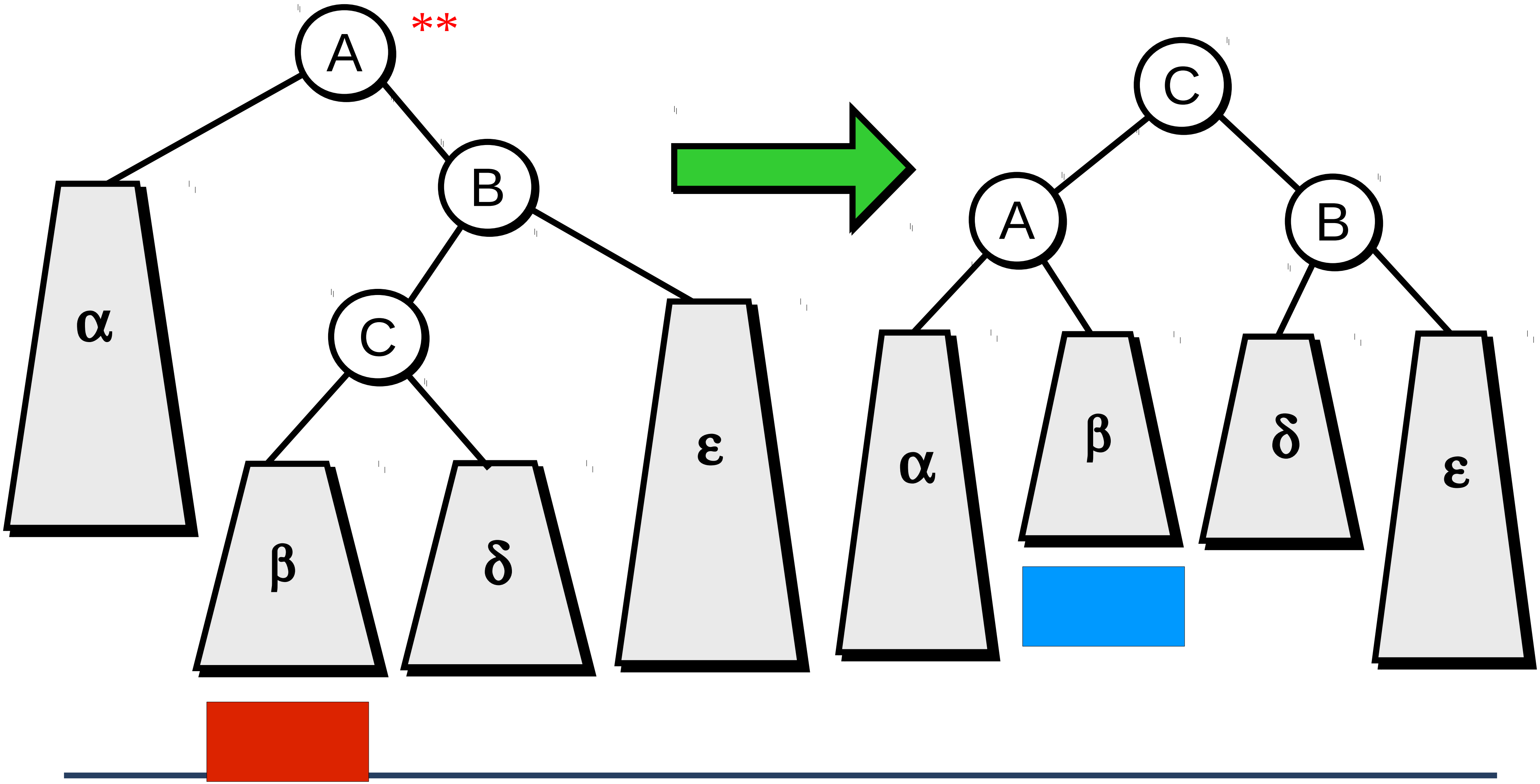
Rotaciones en un AVL: RSD(A)



- Ejercicio: Implementar rotaciones simples en O(1)

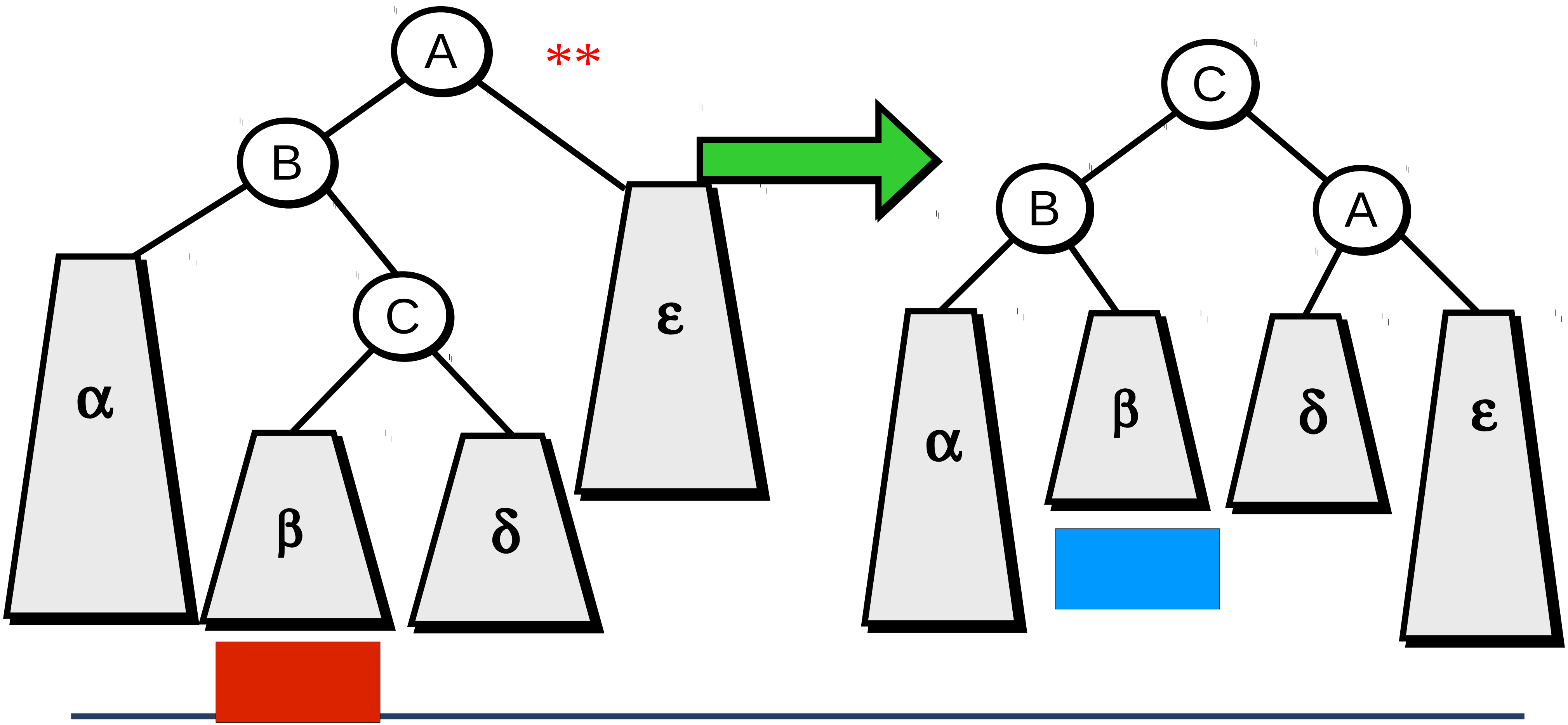
Rotaciones en un AVL: RDDI(A)

- **RDDI(A). Rotación doble Derecha+Izquierda**
  Es equivalente a: RSD(A_der) + RSI(A)



Rotaciones en un AVL: RDID(A)

- **RDID(A). Rotación doble izquierda + derecha**
- **Es equivalente a: RSI(A→izq) + RSI(A)**



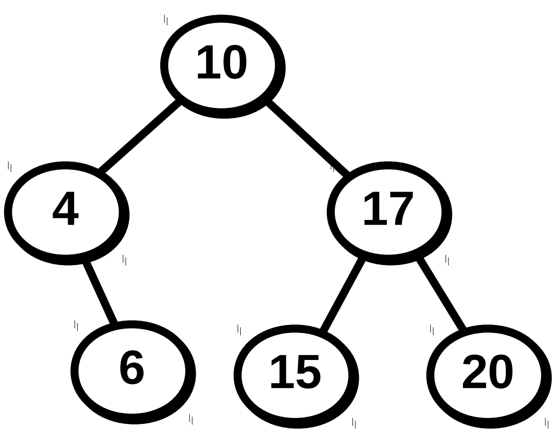
Inserción en un AVL (Resume)

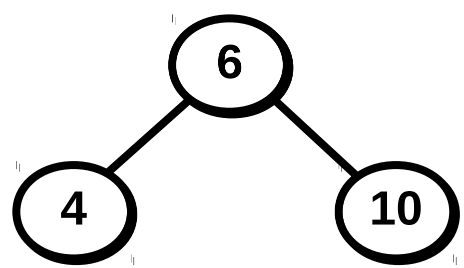
- Igual que en ABB, pero en el camino hacia la raíz estudiar la condición de balanceo por los nodos que se pasa.
- Importante: Cuando se haga el primer balanceo no será necesario hacer otros balanceos. ¿Por qué?

- **Ejemplo**: Dado un árbol nuevo insertar:
  - 4, 5, 7, 2, 1, 3, 6
  - 1, 2 , 3 , 4 , 5 , 6 , 7 , 8.

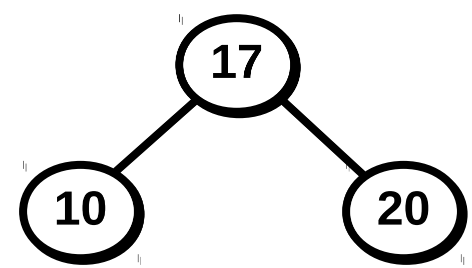
Borrado en AVL

- Es una operación algo más compleja. Hay más casos y puede ser necesario balancear a varios niveles.
- **Algoritmo de eliminación:** Eliminación normal en ABB + comprobación de la condición.
  - Ejercicio 1: Borrar 20, 17,15,
  - Ejercicio 2: Borrar 15,4 , 6





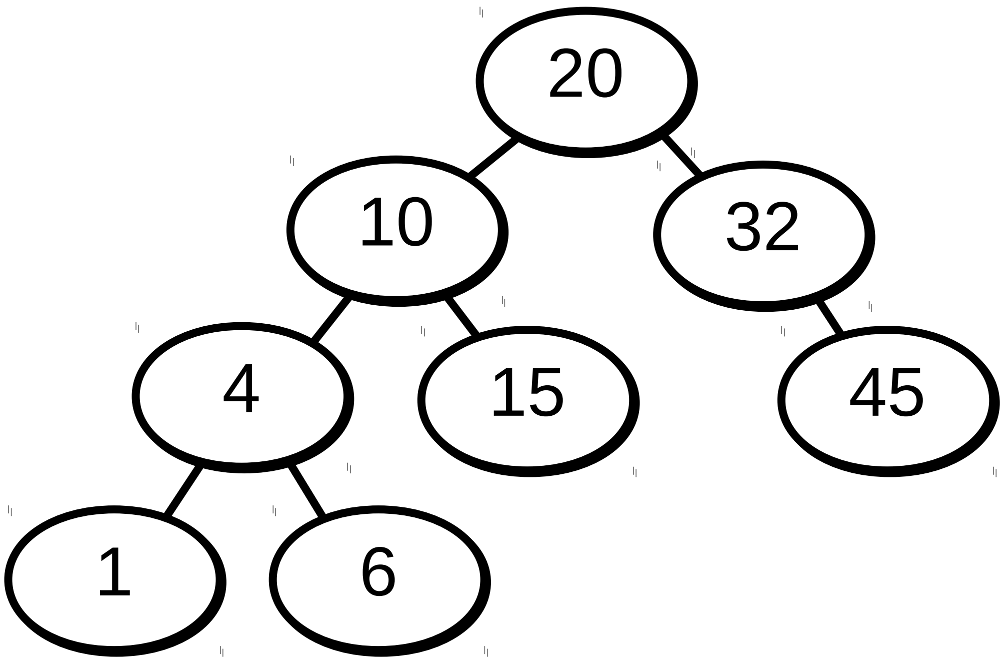
Ej 1



Ej 2

Borrado en un AVL

- **Ejemplo**: Dado el siguiente AVL, eliminar las claves: 4, 15, 32, 45.



Arbol parcialmente ordenado

- Son árboles equilibrados que permiten obtener el mínimo/máximo de un conjunto de datos de forma eficiente O(1)
- La inserción y el borrado de elementos se hace en O(log n)
- Por tanto, son las estructuras utilizadas para representar una cola con prioridad

Arbol parcialmente ordenado

- Conceptualmente un APO, (o POT en inglés) es un árbol binario que debe cumplir:

  1) Para cada nodo, n, el resultado de evaluar

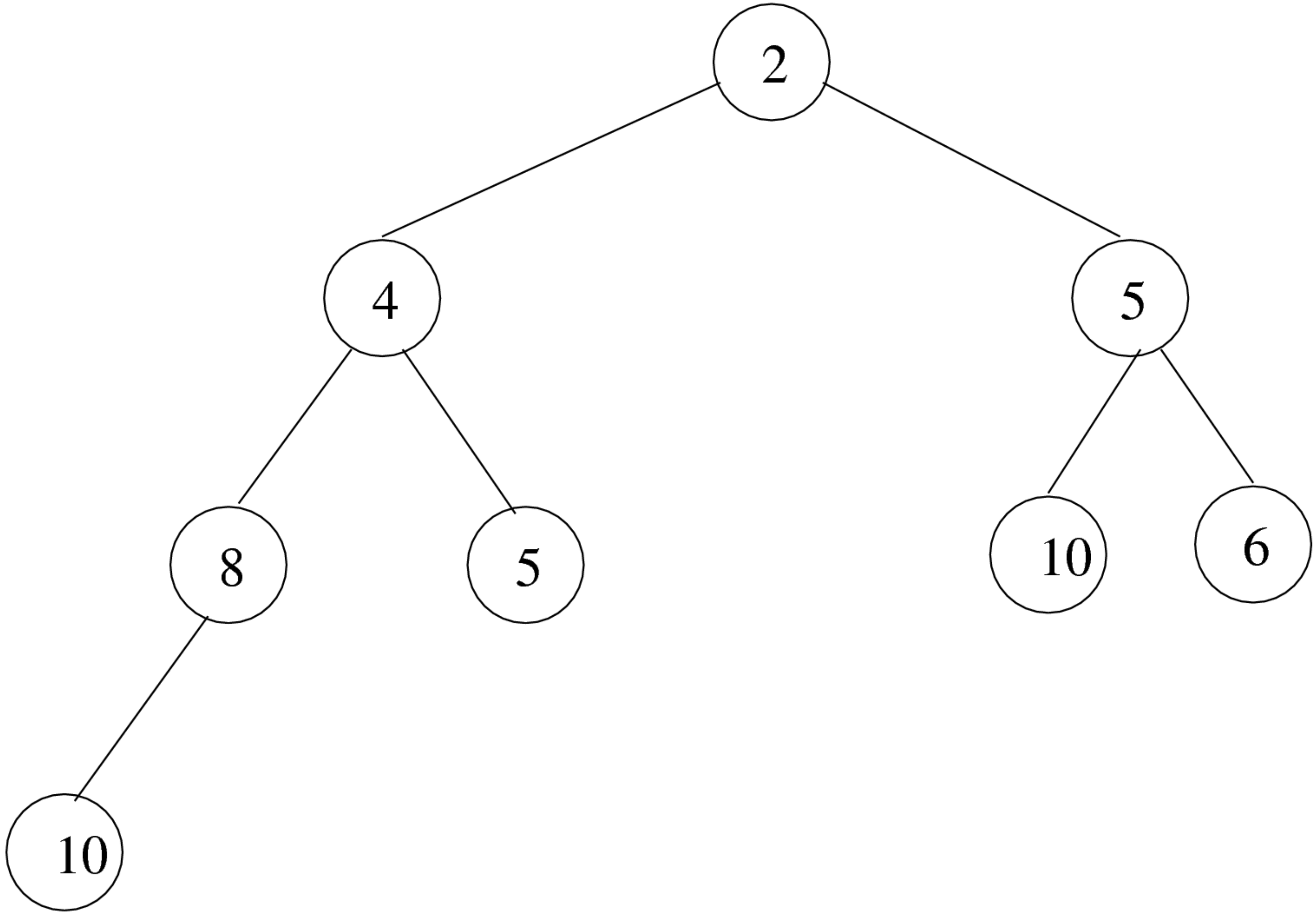
  comp(*n,*(n.left()) && comp(*n, *(n.right())==true

  Si comp es mayor entonces el nodo con valor máximo del conjunto se encuentra en la raíz.

  Si comp es menor en la raíz se encuentra el menor.

  2) Está lo más equilibrado posible: Todos los niveles están completos, excepto el último donde los elementos se encuentran lo más a la izq. posible.

Ejemplo POT - MIN



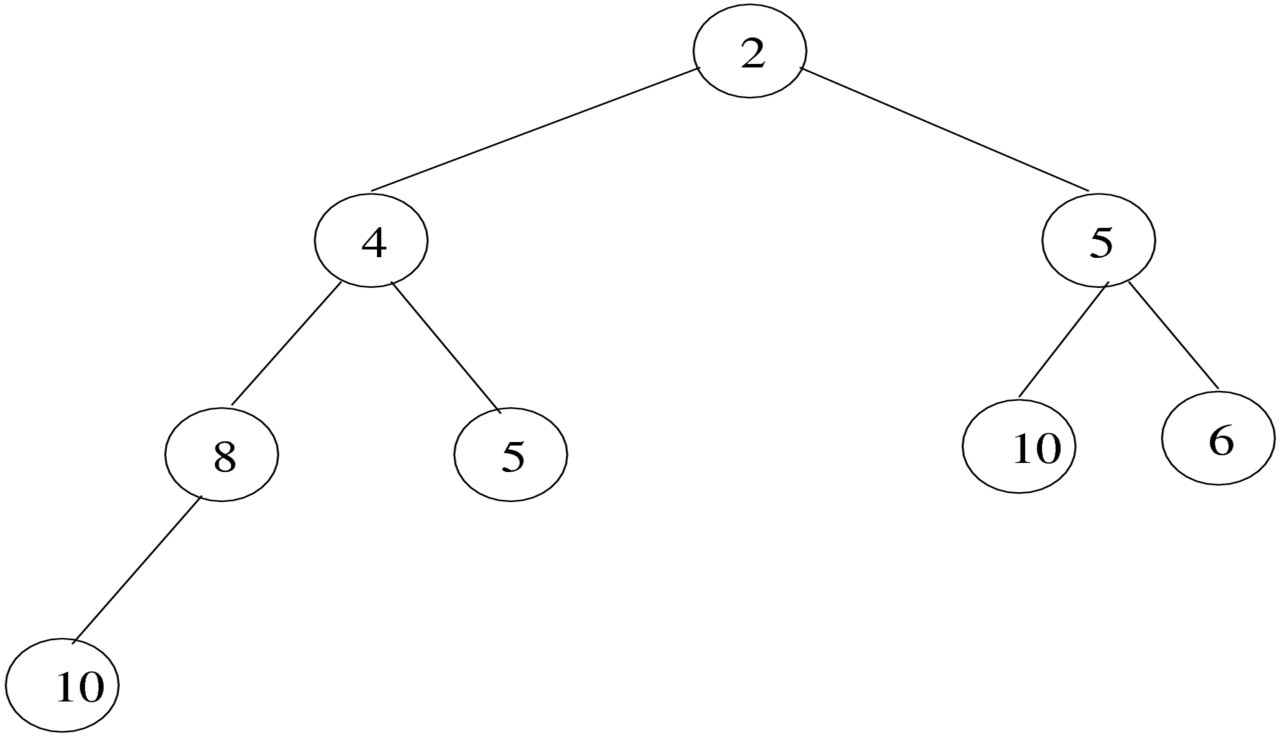
POT: Especificación Métodos

- el constructor por defecto: **`POT(int tam);`**
- el constructor de copia:
  `POT (const POT<T> &copiado);`
- método que devuelve el número de entradas del POT:
  `size_type size() const;`
- método que devuelve el elemento en el tope del receptor (que debe tener al menos un elemento):
  `T top() const;`
- método que inserta el elemento **`elem`** en el receptor:
  `void insert (const T& elem);`
- método que elimina el elemento en el tope del receptor
  `void pop ( );`

Representación del TDA POT (2)

```
template <typename T,
            class comparar=less<T> >
class APO {
...
private:
  vector<T> elementos;
  comparar comp; // Functor
};
```

Representación de un pot<T,less<T> >



	0	1	2	3	4	5	6	7
rep	2	4	5	8	5	10	6	10

Comp = less<int>

Representación del TDA POT (3)

```
/*Función de abstracción:

Cada objeto del tipo rep r = {elementos}
  representa al objeto abstracto árbol
  parcialmente ordenado como:

 - elementos[i] es el elemento almacenado
  en el nodo i, donde

    - 0 es el nodo raíz

    - para cada nodo i su hijo izq está en
     la casilla 2i+1 y el hijo dcha en 2i+2
```

Representación del TDA POT (4)

```
Invariante de representación:
   Para cada nodo i,0<= i <elementos.size():
- comp(elementos[i],elementos[2*i+1]) &&
  comp(elementos[i],elementos[2*i+2]) ==
  true
- Si 2*i+1>num_datos-1 entonces el nodo i
  no tiene hijos
- Si 2*i+1=num_datos-1 entonces el nodo i
  tiene sólo hijo  izquierda
*/
```

Implementación del TDA POT

```
int POT<Tbase,comparar>::parent(int n)
  const
{   if (n==0)     return -1;
   return (n-1)/2; }

int POT<Tbase,comparar>::left(int n) const
{   int elhijo = 2*n+1;
    if (elhijo>=elementos.size())  elhijo =
  -1;
   return elhijo;
}
```

Implementación del TDA POT (2)

```
int POT<Tbase,comparar>::right(int n) const
{  int elhijo = 2*n+2;
   if (elhijo>=elementos.size())
      elhijo = -1;
   return elhijo; }

POT<Tbase,comparar>::POT(int tama)
{  elementos.reserve(tama); }
```

Implementación del TDA POT (3)

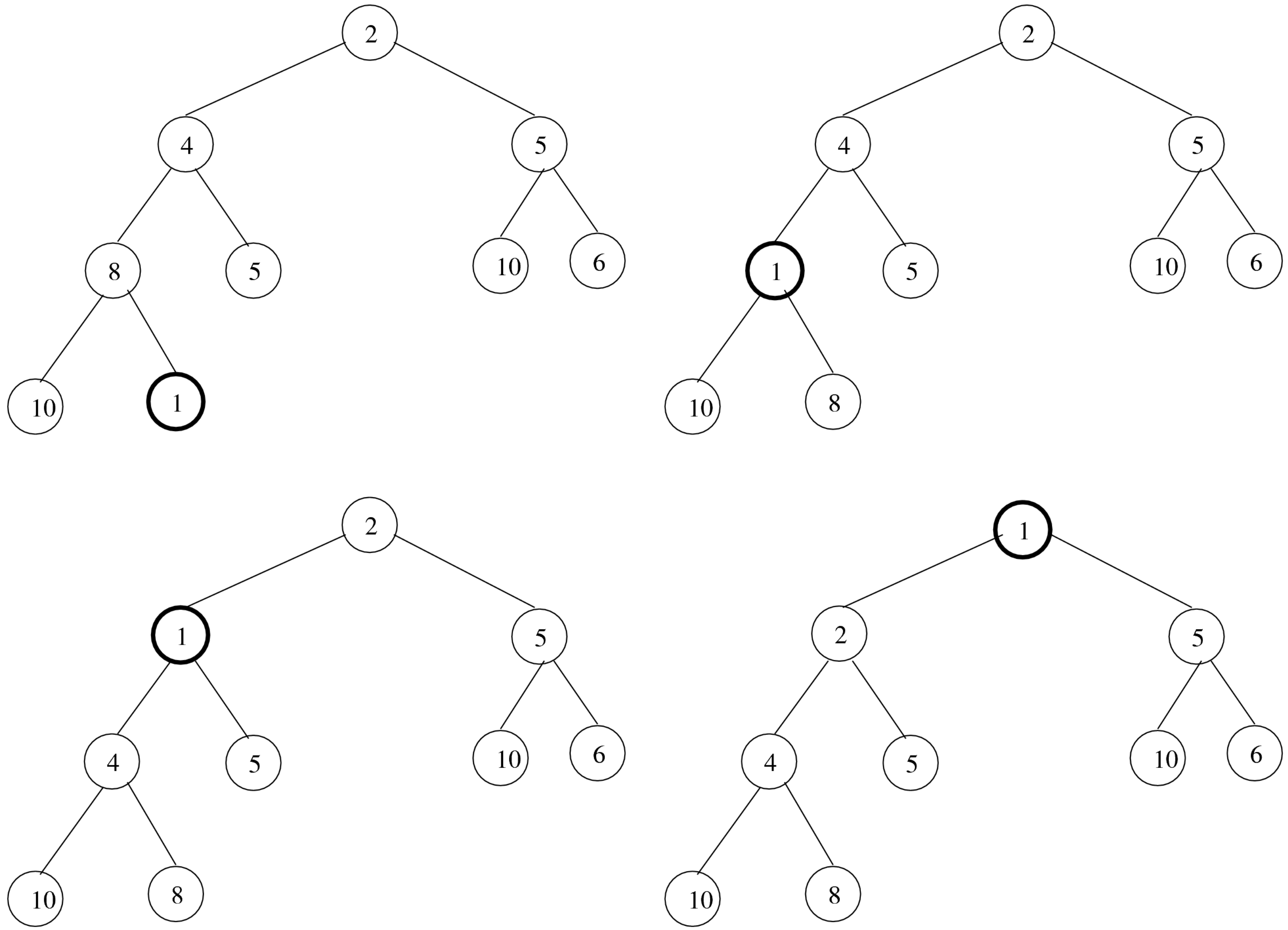
```
POT<Tbase,comparar>::POT(const
  POT<Tbase,comparar> & pot)
{ elementos= pot.elementos);
  comp = pot.comp;
}

Tbase POT<Tbase,comparar>::top() const
{
  assert (elementos.size()>0);
  return elementos[0];
}
```

POT: Inserción

- Cuando se añade un elemento siempre se hace lo más a la izquierda posible en el nivel que no esté completo, y si el nivel está completo como el elemento más a la izquierda en el siguiente nivel.

- Una vez insertado, la estructura se reorganiza para asegurar que se satisface el invariante de la representación, esto es, que tenemos un POT válido.

POT: inserción



Implementación del TDA POT (4)

```
void POT<Tbase,comparar>::insert(const Tbase & e)
{
  if (elementos.capacity()==elementos.size())
    elementos.reserve(2*elementos.capacity());

  elementos.push_back(e);   // Se inserta al final
  reajusta_hacia_arriba(elementos.size()-1);
}
```

`Reajustar` sube el elemento, e, hacia la raíz hasta garantizar que sus hijos sean mayores que él.

Implementación del TDA POT (5)

```
template <typename Tbase,class comparar>
void
  POT<Tbase,comparar>::reajusta_hacia_arriba(int
  n)
{ // n es la posición donde se ha insertado
  el // elemento
   bool terminar = false;
   int pos_actual = n;
   ...
```

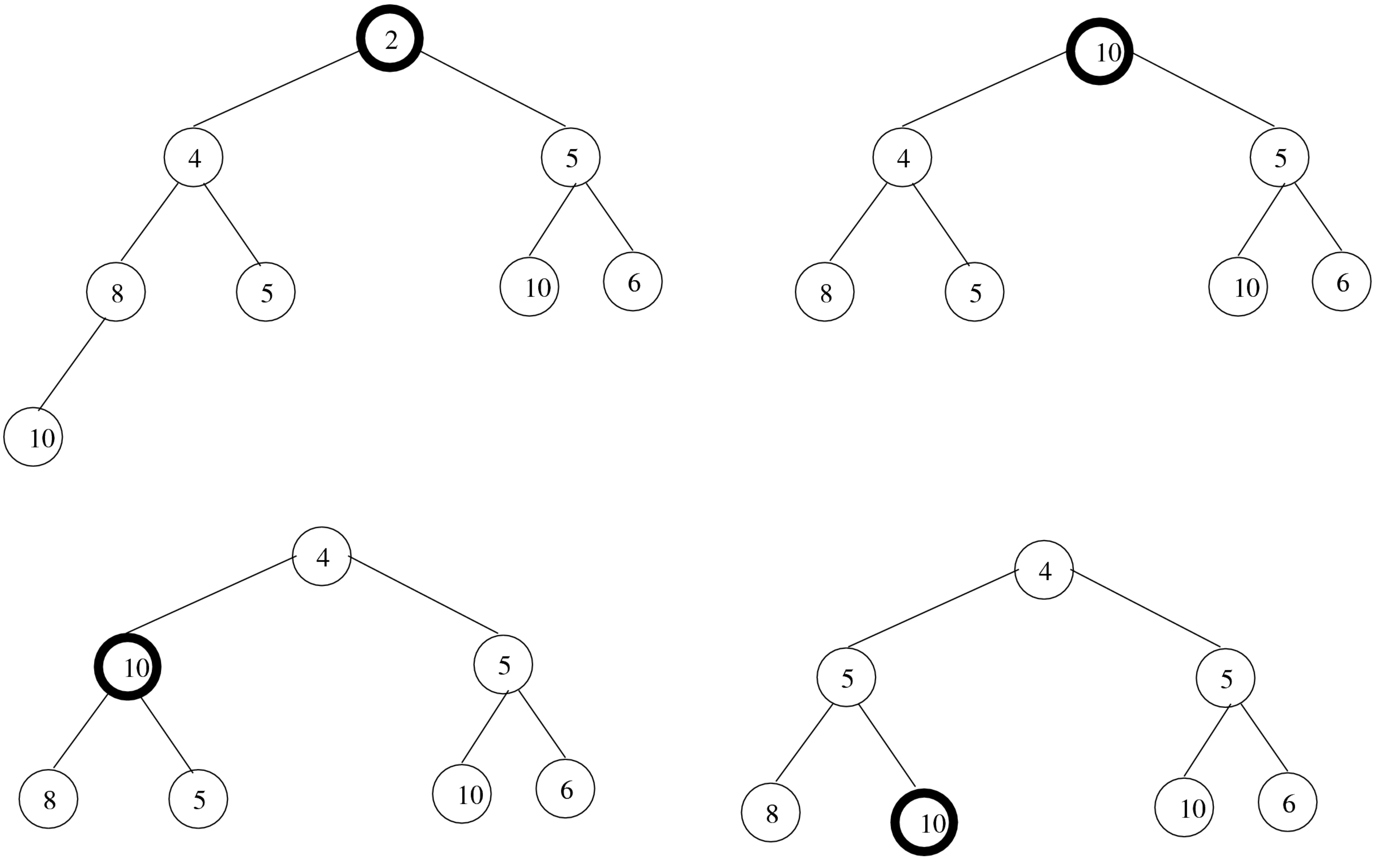
Implementación del TDA POT (6)

```
...
while (padre(pos_actual)>=0 && !terminar)
  if (!comp( elementos[padre(pos_actual)],
             elementos[pos_actual])) {
     intercambia
        (elementos[padre(pos_actual)],
         elementos[pos_actual]);
     pos_actual = padre(pos_actual);
  }
  else
     terminar = true;
}
```

Borrado de elementos.

- Siempre se borra el elemento situado en la raíz del POT (el mínimo o máximo)
- Esquema:
  - En la raíz del árbol se copia el elemento en la última posición, ult, del árbol,
  - Se elimina el elemento en dicha posición, ult.
  - Se deja caer el elemento hasta que se garantice la condición de APO

POT: Borrado



Implementación del TDA POT (7)

```
template <typename Tbase, class comparar>void
  POT<Tbase,comparar>::top()

{
  assert(elementos.size()>0);
  elementos[0] = elementos[elementos.size()-1];
  elementos.pop_back();
  reajusta_hacia_abajo();
}
```

Implementación del TDA APO (8)

```
void APO<Tbase,comparar>::reajusta_hacia_abajo() {
  int pos_actual = 0;
  bool terminar = false;

  while (pos_actual<elementos.size()-1 &&
        !terminar) {
        if (hijoIzda(pos_actual)==-1)
            // No tiene hijos
            terminar = true;
        else if (hijoIzda(pos_actual)==
            elementos.size()-1) {
            // Tiene solo hijo izquerda (1 solo hijo)
            if (comp(elementos[hijoIzda(pos_actual)],
                 elementos[pos_actual])) {
```

Implementación del TDA APO (9)

```
    ...
         intercambia(
             elementos[hijoIzda(pos_actual)],
             elementos[pos_actual]);
         pos_actual = hijoIzda(pos_actual);
    } else  terminar = true;
}  else { // Tiene dos hijos
   .....
```

Implementación del TDA APO (10)

```
else { // Tiene dos hijos
        int pos_menor;
        if (comp(elementos[hijoIzda(pos_actual)],
                 elementos[hijoDcha(pos_actual)]))
            pos_menor = hijoIzda(pos_actual);
        else pos_menor = hijoDcha(pos_actual);
        if (comp(elementos[pos_menor],
                 elementos[pos_actual])) {
            intercambia(elementos[pos_menor],
            elementos[pos_actual]);
            pos_actual = pos_menor;
        }  else terminar = true;
      }
    }
  }
```