ÁRBOLES BINARIOS DE BÚSQUEDA

Joaquín Fernández-Valdivia
Javier Abad
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

DECSAI

Definición de ABB

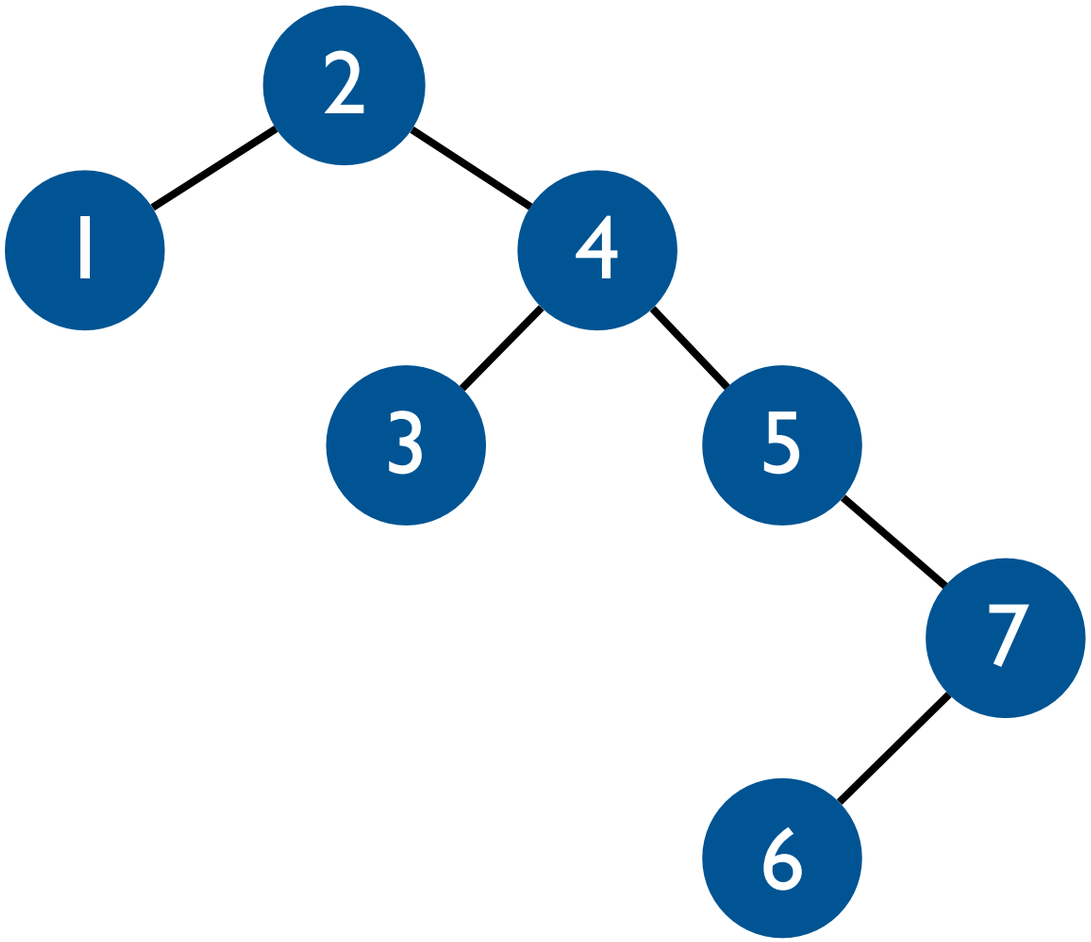
- Un ABB es un árbol binario en el que, dado un nodo, x:
  - ❖ todos los elementos almacenados en el subárbol izquierdo de x son menores (o iguales*) que el elemento almacenado en x,

  y
  - ❖ todos los elementos almacenados en el subárbol derecho de x son mayores que el elemento almacenado en x

- Recursivamente: todas las etiquetas del subárbol izquierdo de un nodo x son menores que el elemento almacenado en x, y las etiquetas del subárbol derecho de un nodo x son mayores que él

*Habitualmente tendremos claves no repetidas

Ejemplos



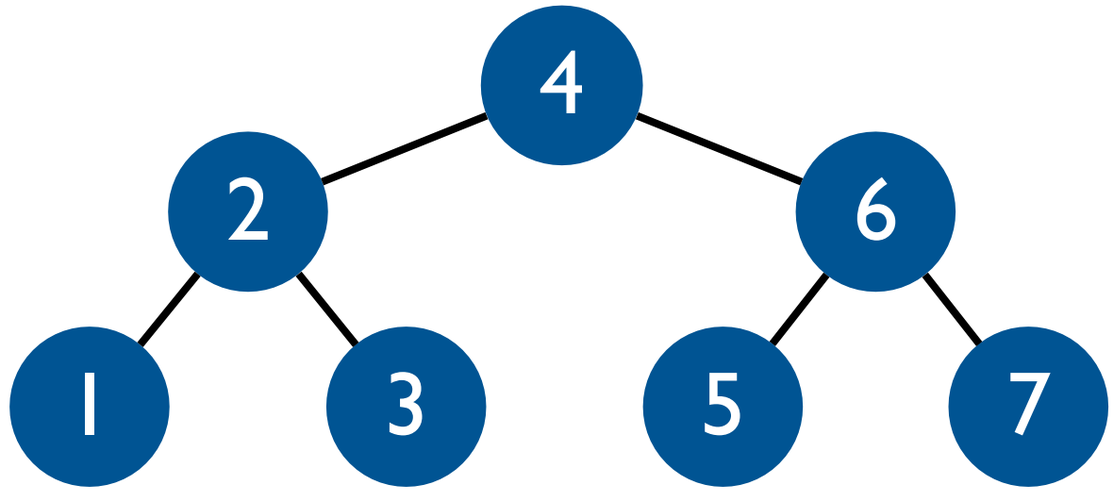
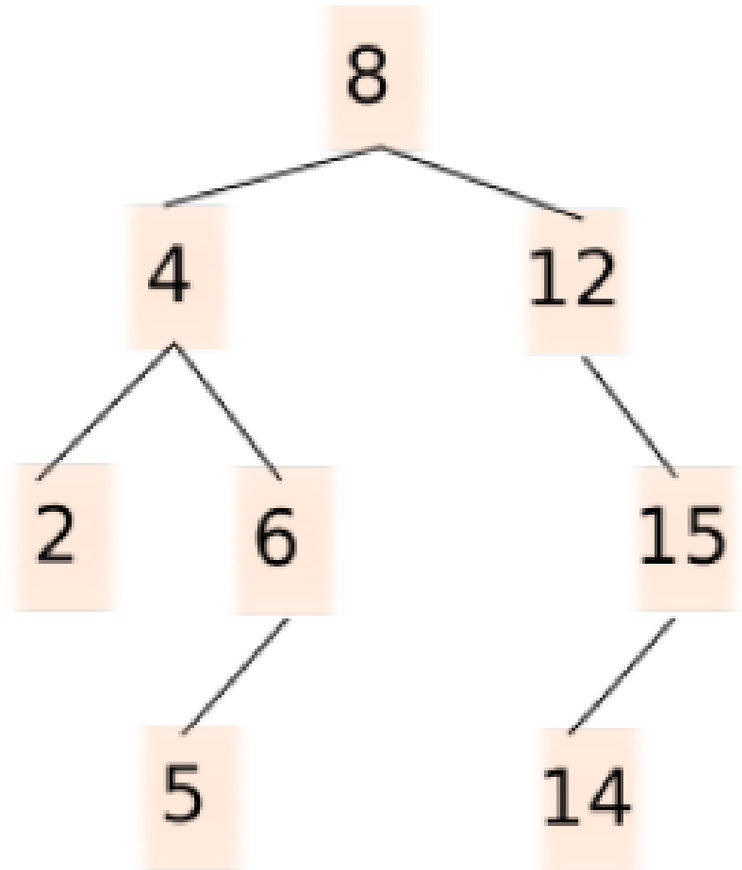


ABB vs APO

- A diferencia del APO, un ABB es un árbol binario que no cumple ninguna condición geométrica y solo cumple una analítica:

  - La etiquetas están ordenadas de forma que el elemento situado en un nodo es mayor que todos los elementos que se encuentran en el subárbol izquierdo y menor que los que se encuentran en el subárbol derecho

Ejemplos



ABB

- ¿Dónde se localizará el elemento menor en un ABB?

ABB

- ¿Dónde se localizará el elemento menor en un ABB?
- ¿Y el mayor?

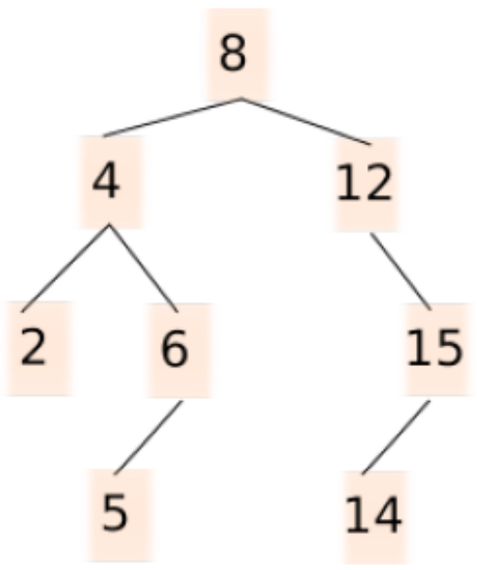
ABB

- ¿Dónde se localizará el elemento menor en un ABB?
- ¿Y el mayor?
- ¿El nodo con el elemento mayor podría tener algún hijo?

ABB

- En un ABB el menor de los elementos será o bien el nodo mas a la izquierda que sea hoja o que como mucho tenga un hijo a la derecha
- De la misma forma el elemento mayor almacenado en un ABB será aquel que se situé más a la derecha teniendo como mucho un hijo a la izquierda o siendo hoja

- En el ABB el mínimo es 2 (el elemento que se encuentra más a la izquierda y que en este caso es hoja)
- Y el mayor elemento es 15 que aunque no es una hoja solamente tiene un hijo a la izquierda



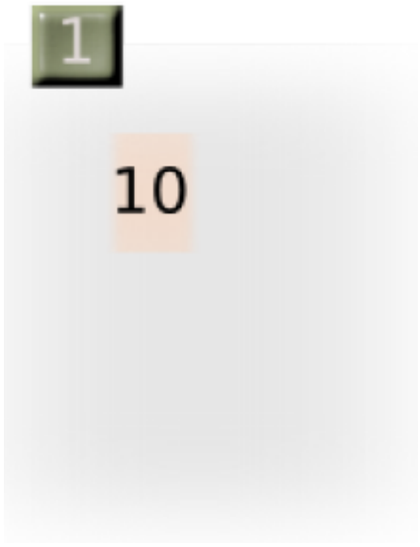
Construcción del ABB

- Ejemplo: construcción de un ABB con las claves

$$\{10, 5, 14, 7, 12, 3, 19, 8, 6\}$$

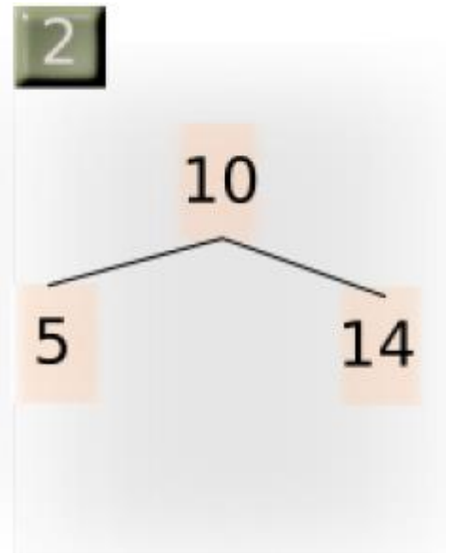
Construcción del ABB

1. El primer elemento del conjunto de etiquetas, 10, es la raíz. Se construye un árbol con un solo nodo



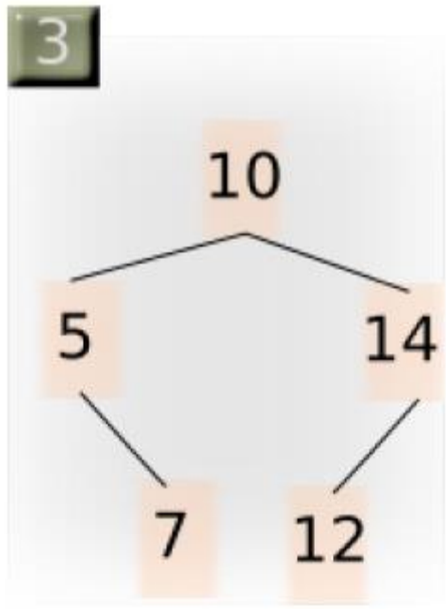
Construcción del ABB

2. A continuación nos dan la etiqueta 5, ya que es menor que 10 esta se coloca como hijo izquierdo de 10, el hijo derecho será 14



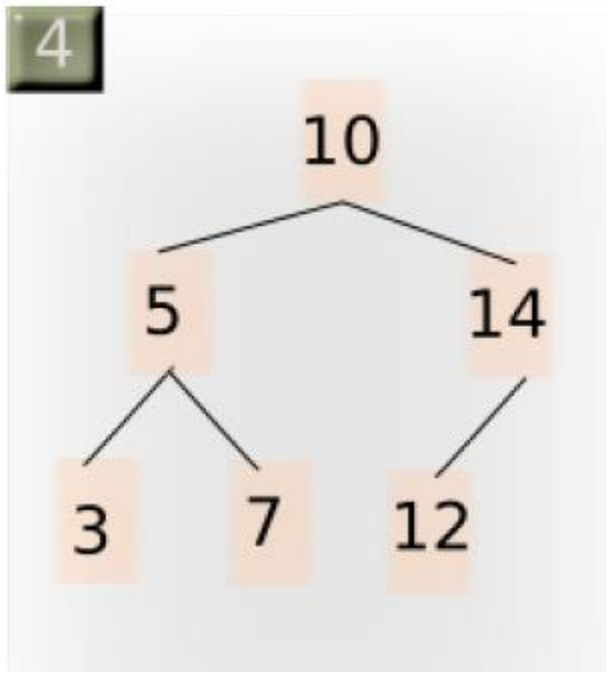
Construcción del ABB

3. Para insertar 7 en primer lugar se compara con 10 que es menor, por lo tanto redirigimos nuestro proceso de inserción por el subárbol izquierdo



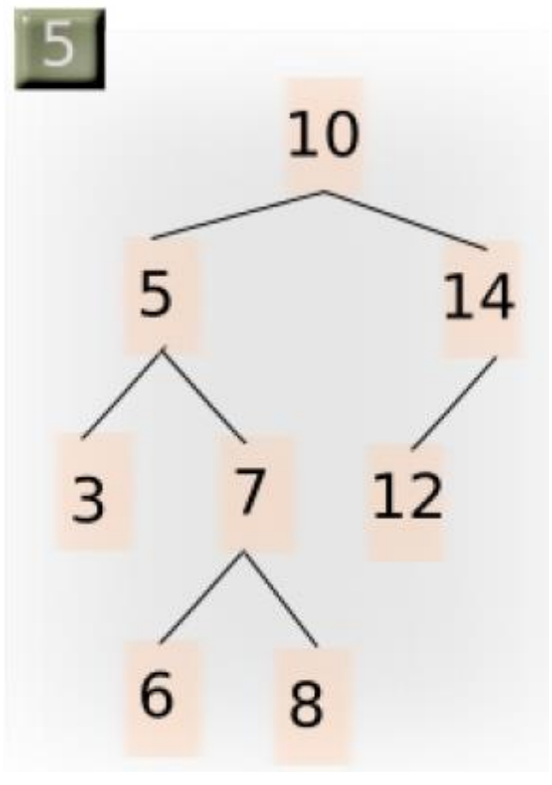
Construcción del ABB

4. Ahora se compara con 5 al ser mayor y 5 no tener hijo a la derecha, el 7 pasa a ser el hijo a la derecha de 5



Construcción del ABB

5. Así, para insertar el 6 en el árbol debemos ir nodo por nodo viendo si tirar para la derecha o la izquierda. Los pasos serían:
   I. 6 < 10 → tiramos a la izquierda
   II. 6 > 5 → tiramos al subárbol derecho
   III. 6 < 7 → como no tiene hijo izquierdo, ponemos a 6 como hijo izquierdo



Eficiencia

- En promedio, es decir en un conjunto de búsquedas, la mayoría va a tener una eficiencia de $\log_2(n)$

- Esto es así ya que cada vez que realizamos una comparación, en promedio, no tendremos que comparar con la mitad de los restantes valores

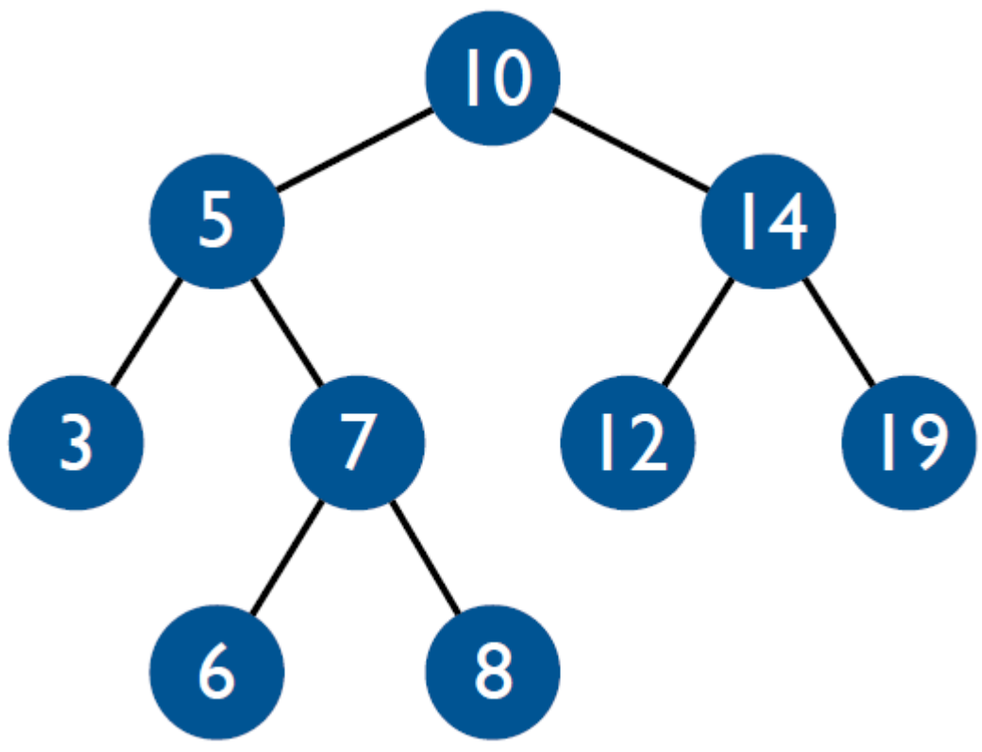
- Pero existe un caso donde la búsqueda de un elemento tiene eficiencia O(n)

- Esta es la situación que ocurre cuando las claves se disponen en una sola rama

- ¿Cuándo se da este caso?

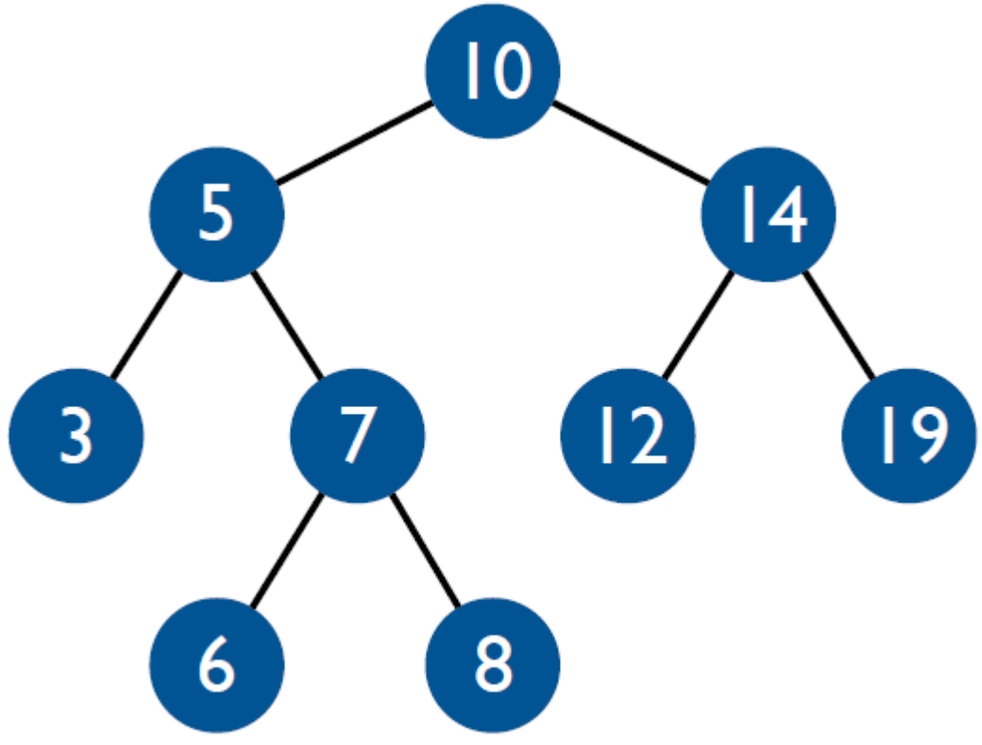
Propiedad

- Preorden:



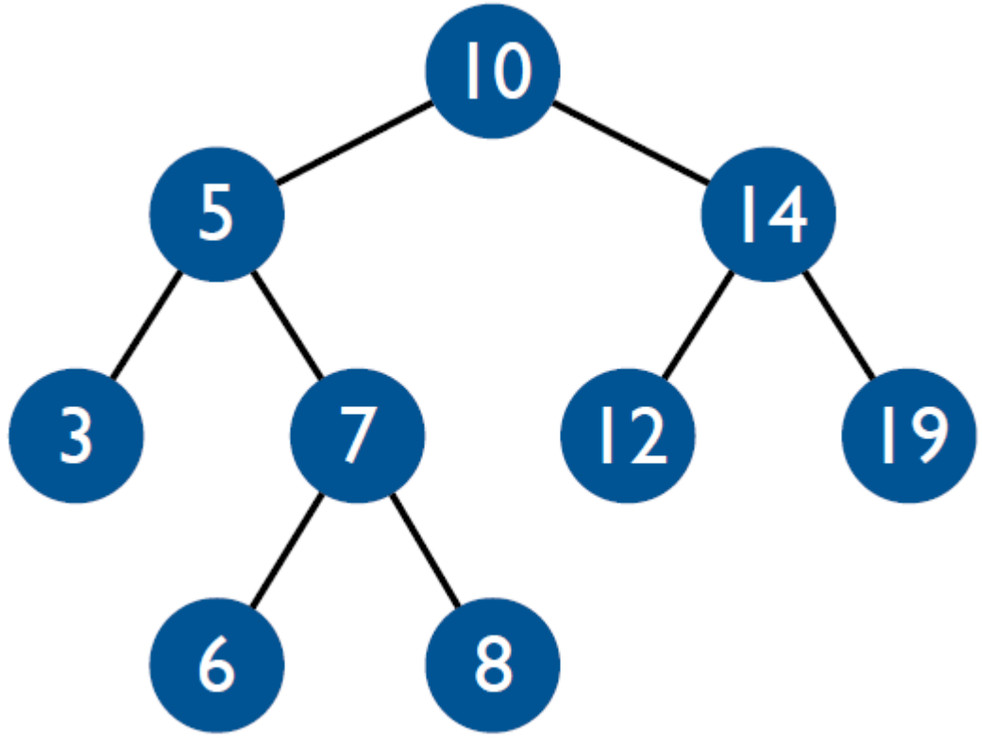
Propiedad

- Preorden: 10, 5, 3, 7, 6, 8, 14, 12, 19



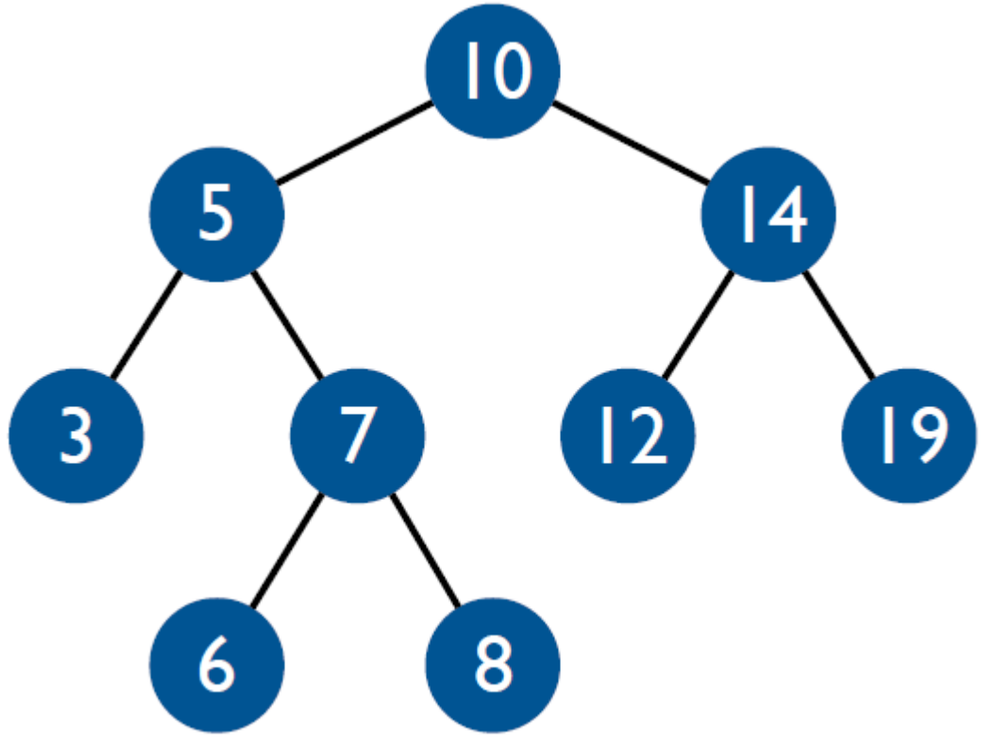
Propiedad

- Preorden: 10, 5, 3, 7, 6, 8, 14, 12, 19
- Postorden:



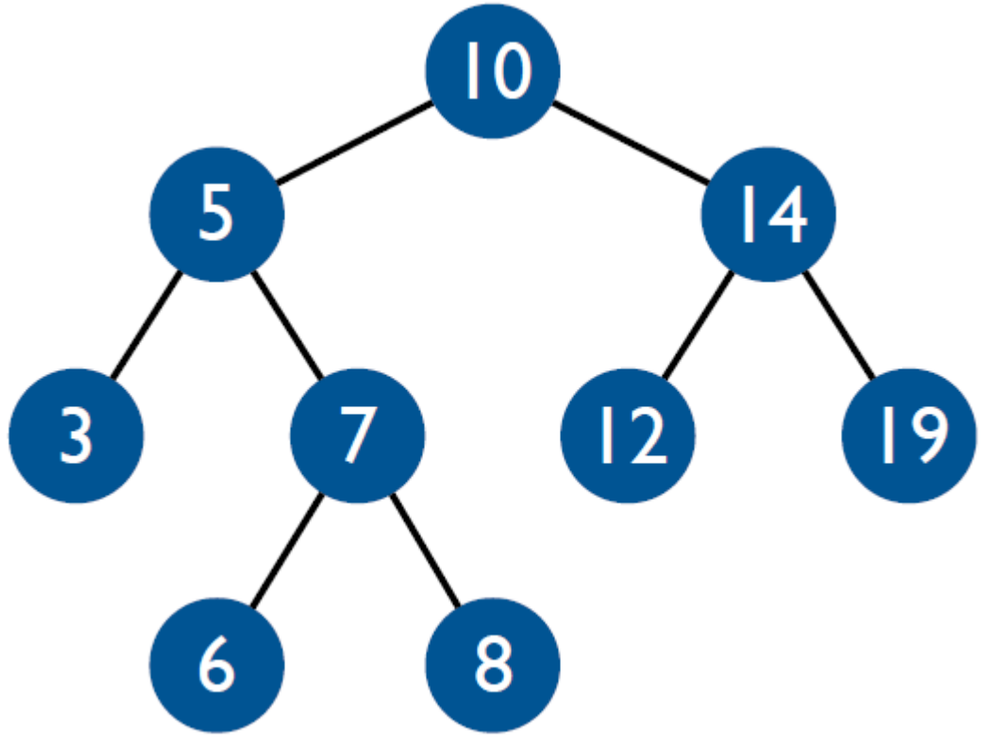
Propiedad

- Preorden: 10, 5, 3, 7, 6, 8, 14, 12, 19
- Postorden: 3, 6, 8, 7, 5, 12, 19, 14, 10



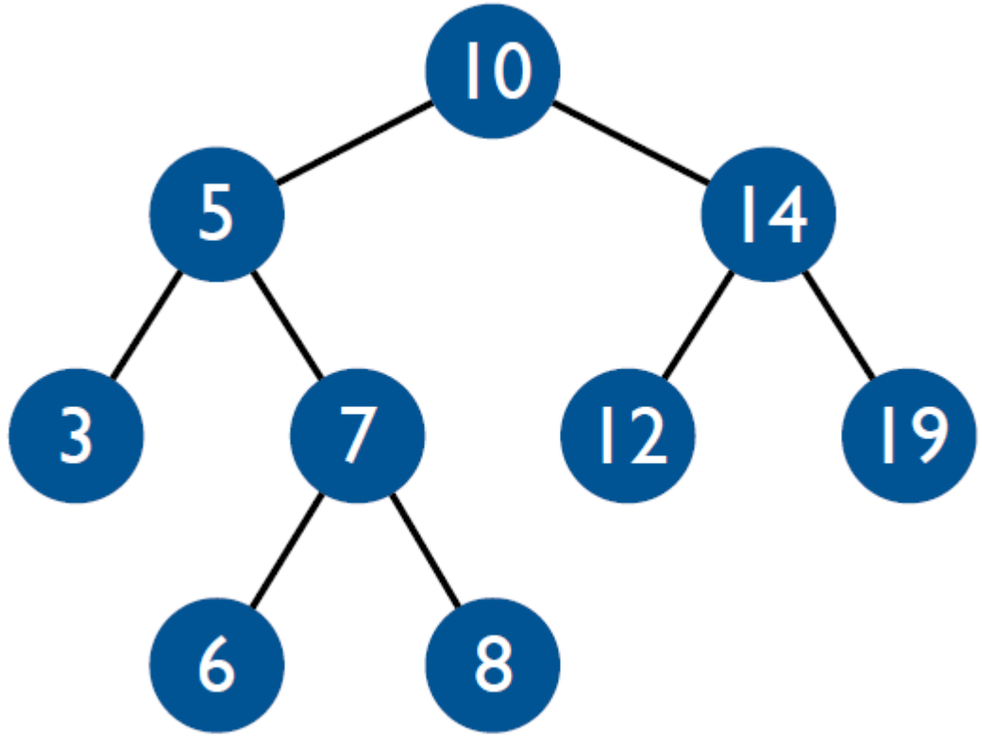
Propiedad

- Preorden: 10, 5, 3, 7, 6, 8, 14, 12, 19
- Postorden: 3, 6, 8, 7, 5, 12, 19, 14, 10
- Anchura:



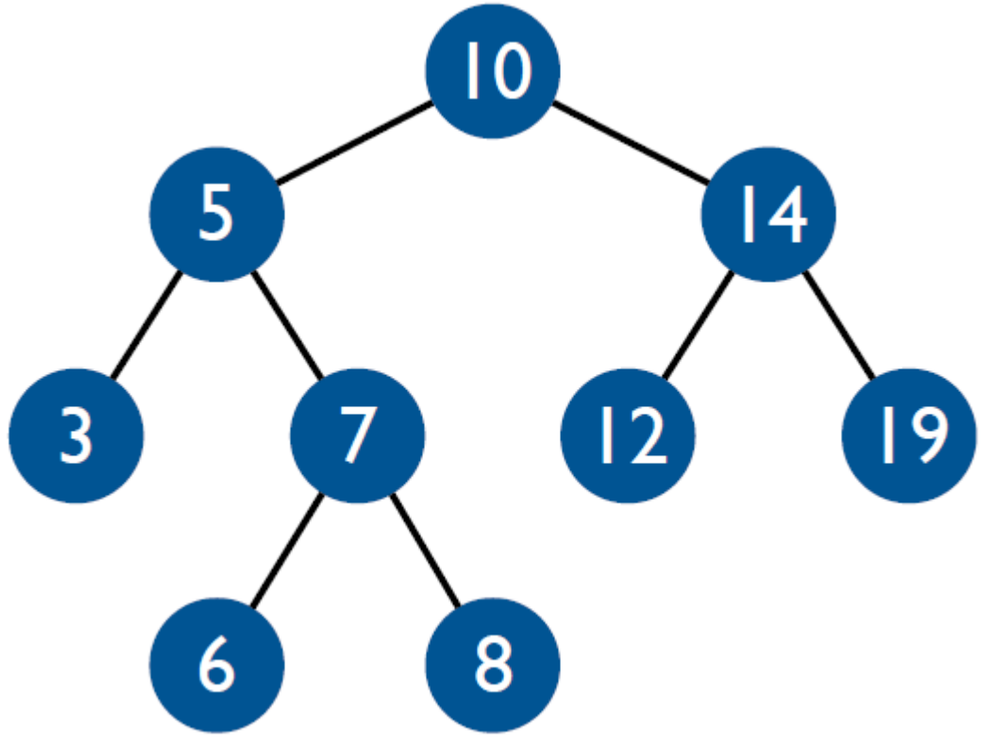
Propiedad

- Preorden: 10, 5, 3, 7, 6, 8, 14, 12, 19
- Postorden: 3, 6, 8, 7, 5, 12, 19, 14, 10
- Anchura: 10, 5, 14, 3, 7, 12, 19, 6, 8



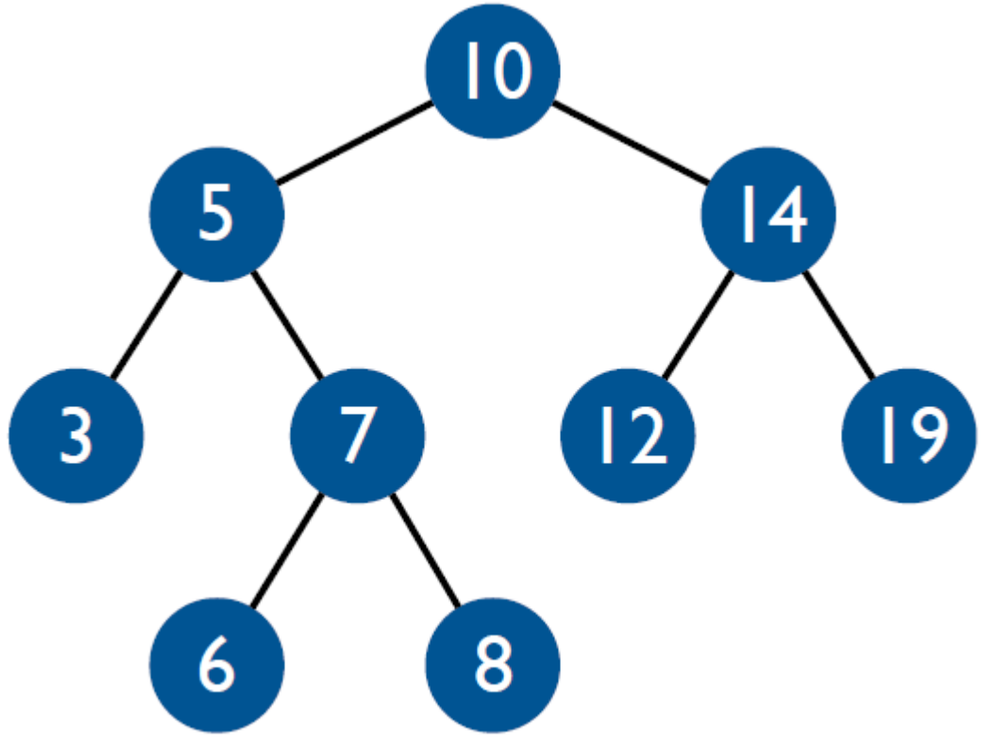
Propiedad

- Preorden: 10, 5, 3, 7, 6, 8, 14, 12, 19
- Postorden: 3, 6, 8, 7, 5, 12, 19, 14, 10
- Anchura: 10, 5, 14, 3, 7, 12, 19, 6, 8
- Inorden:



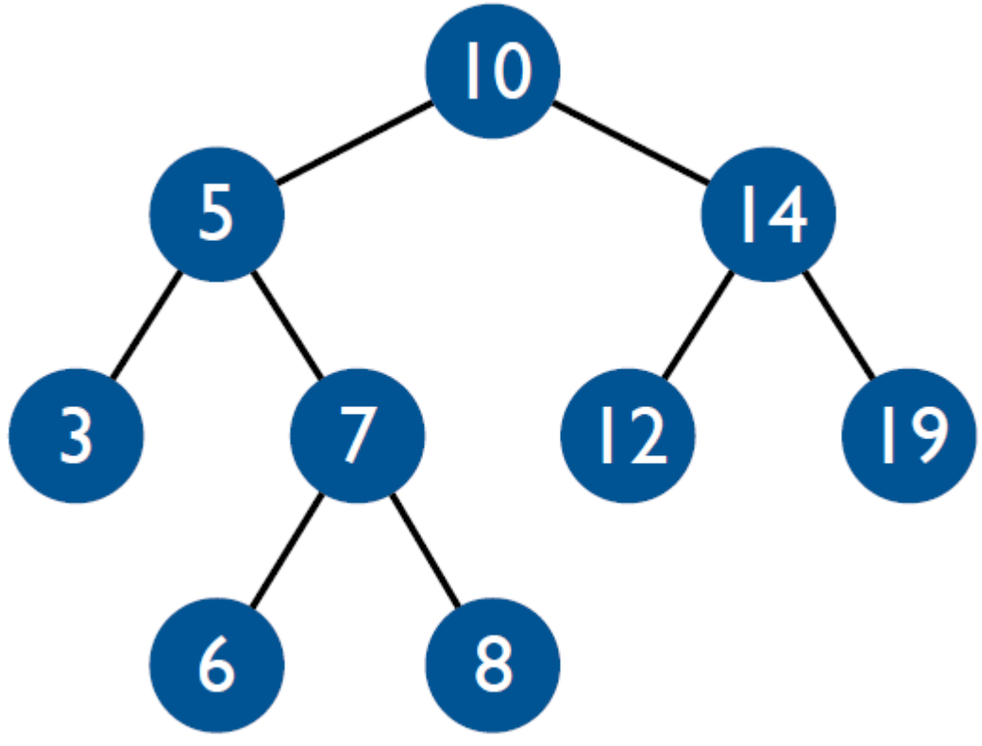
Propiedad

- Preorden: 10, 5, 3, 7, 6, 8, 14, 12, 19
- Postorden: 3, 6, 8, 7, 5, 12, 19, 14, 10
- Anchura: 10, 5, 14, 3, 7, 12, 19, 6, 8
- Inorden: 3, 5, 6, 7, 8, 10, 12, 14, 19



Propiedad

- Preorden: 10, 5, 3, 7, 6, 8, 14, 12, 19
- Postorden: 3, 6, 8, 7, 5, 12, 19, 14, 10
- Anchura: 10, 5, 14, 3, 7, 12, 19, 6, 8
- Inorden: 3, 5, 6, 7, 8, 10, 12, 14, 19



Propiedad

- El recorrido en inorden de un ABB nos da la lista de nodos ordenada de menor a mayor

- Esta propiedad define un método de ordenación similar al Quicksort, con el nodo raíz jugando un papel similar al del elemento de partición del Quicksort

- En los ABB hay un gasto extra de memoria por los punteros que mantienen la estructura del árbol

Propiedad

- Usar un ABB tiene la ventaja de que el número de datos que se pueden almacenar no está limitado

- Se podría construir un algoritmo de ordenación que consistiera en insertar todos los datos en un ABB y después listar el árbol en inorden

- ¿Qué estructura/s de datos se podría/n implementar como un ABB?

Propiedad

- Usar un ABB tiene la ventaja de que el número de datos que se pueden almacenar no está limitado

- Se podría construir un algoritmo de ordenación que consistiera en insertar todos los datos en un ABB y después listar el árbol en inorden

- ¿Qué estructura/s de datos se podría/n implementar como un ABB?

- Los tipos *set* y *map* de la STL están implementados con una variante equilibrada del ABB

https://www.cplusplus.com/reference/set/set/

https://www.cplusplus.com/reference/map/map/

Implementación

- La implementación de un ABB en C++ puede tomar como implementación base el *ArbolBinario*

- A diferencia de éste tenemos que añadir una función para buscar, insertar y borrar un elemento en el ABB

- Por otro lado el único iterador que nos interesa es el inorden para, dado el ABB, obtener una ordenación de las claves que almacena

- Por lo tanto se puede mantener simplemente la clase nodo del árbol binario y sobrecargar en este el operador ++ para pasar al siguiente nodo en inorden y respecto al operador -- igual

TDA ABB

```
/**
   TDA ABB::ABB,Insertar,Existe,Borrar,begin,end,~ABB.

   El TDA ABB modela un Arbol Binario de Búsqueda.
   Es un árbol binario etiquetado con datos del tipo
   Tbase, entre los que existe un orden lineal
   (modelado mediante operator<). Para todo nodo se
   cumple que las etiquetas de los nodos a su izqda
   son menores estrictos que la suya, y que las
   etiquetas de los nodos a su drcha son mayores o
   iguales que la suya.

   Requisitos para el tipo instanciador Tbase:
   Tbase debe tener definidas las siguientes
    operaciones:

   - Tbase & operator=(const Tbase & e);
   - bool operator!=(const Tbase & e);
   - bool operator==(const Tbase & e);
   - bool operator<(const Tbase & e);

   Son objetos mutables.
   Residen en memoria dinámica.
*/
```

TDA ABB

```
template <class Tbase>
class ABB {
public:

  ABB();
  /**
     Constructor por defecto.

     @doc
     Crea un Arbol Binario de Búsqueda vacío.
  */


  ABB(const ABB<Tbase> & a);
  /**
     Constructor de copia.

     @param a: Arbol que se copia.

     @doc
     Crea un Arbol Binario de Búsqueda duplicado de a.
  */
```

TDA ABB

```
ABB(const Tbase & e);
```

```
/**
   Constructor primitivo.

   @param e: Elemento a insertar.

   @doc
   Crea un Arbol Binario de Búsqueda con un sólo
   nodo, que se etiqueta con el valor "e".
*/
```

```
bool Existe(const Tbase & e) const;
```

```
/**
   Informa de la existencia de un elemento en el ABB.

   @param e: elemento que se busca.

   @return true, si el elemento e está en el árbol.
           false, en otro caso.
 */
```

TDA ABB

```
void Insertar(const Tbase & e);
/**
   Inserta un elemento en el árbol.

   @param e: Elemento que se inserta.

   @doc
   Añade al ABB un nuevo nodo etiquetado con e.
*/
```

```
void Borrar(const Tbase & e);
/**
   Elimina un elemento.

   @param e: Elemento a eliminar.

   @doc
   Si existen uno o más nodos en el receptor con la
   etiqueta e, elimina uno de ellos.
*/
```

TDA ABB

```
/**
   TDA ABB<Tbase>::iterator permite realizar un
   recorrido por orden ascendente (según operator<)
   de los elementos de un ABB<Tbase>.
*/
class iterator {
public:
   iterator();
   iterator(ArbolBinario<Tbase>::Nodo n);
   iterator(ArbolBinario<Tbase>::iterator [redacted] it);
   bool operator!=(const ABB<Tbase>::iterator & it);
   bool operator==(const ABB<Tbase>::iterator & it);
   Tbase operator*();
   iterator operator++();
};
```

→ **iterator_inorder**

TDA ABB

```
iterator begin();
/**
   Posición de inicio del recorrido.

   @return Posición de inicio del recorrido.
*/

iterator end();
/**
   Posición final del recorrido.

   @return Posición final del recorrido.
*/

iterator begin() const;
/**
   Posición de inicio del recorrido.

   @return Posición de inicio del recorrido.
*/
```

TDA ABB

```
iterator end() const;
/**
   Posición final del recorrido.

   @return Posición final del recorrido.
*/

~ABB();
/**
   Destructor.
*/
```

Ejemplo de uso del TDA ABB

```
/**
   Programa ejemplo de prueba del TDA ABB
*/

#include <iostream>
#include "abb.h"

template <class Tbase>
ostream & operator<<(ostream & s,
                     const ABB<Tbase> & abb)
{
  ABB<int>::iterator i = abb.begin();
  while (i != abb.end())
    {
      s << *i << ", ";
      ++i;
    }
  s << endl;
  return s;
}
```

Ejemplo de uso del TDA ABB

```
int main()
{
  ABB<int> abb;

  cout << "Introduce un entero (<0 para terminar): ";
  int e;
  cin >> e;
  while (e > 0)
    {
      abb.Insertar(e);
      cout << "Introduce un entero (<0 para terminar): ";
      cin >> e;
    }

  ABB<int>::iterator i = abb.begin();
  while (i != abb.end())
    {
      cout << *i << ", ";
      ++i;
    }
  cout << endl;
```

Ejemplo de uso del TDA ABB

```
  cout << "Buscando datos" << endl;
  cout << "Introduce un entero (<0 para terminar): ";
  cin >> e;
  while (e > 0)
    {
      if (abb.Buscar(e))
        cout << e << " está en el ABB" << endl;
      else
        cout << e << " NO está en el ABB" << endl;
      cout << "Introduce un entero (<0 para terminar): ";
      cin >> e;
    }

  cout << "Borrando elementos del ABB:" << endl;
  cout << "Introduce un entero (<0 para terminar): ";
  cin >> e;
  while (e > 0)
    {
      abb.Borrar(e);
      cout << abb;
      cout << "Introduce un entero (<0 para terminar): ";
      cin >> e;
    }

  return 0;
}
```

TDA ABB: Representación

```
template <class Tbase>
class ABB{
    ...

  class iterator {
    ...
  private:
    ArbolBinario<Tbase>::iterator [illegible] eliterador;
  };

private:

  ArbolBinario<Tbase> arbolb;

  void borrar_nodo(ArbolBinario<Tbase>::Nodo n);
  /**
     Elimina un nodo del árbol.
     @param n: Nodo a eliminar. n != NODO_NULO.

     @doc
     Elimina n del árbol receptor.
  */
};
```

TDA ABB: Representación

```
/*
  Función de abstracción:
  -----------------------
  Cada objeto del tipo rep r = {arbolb} representa al
  objeto abstracto arbolb.

  Invariante de representación:
  -----------------------------
  Para cada nodo n de r.arbolb se cumple:
  - arbolb.Etiqueta(n) > arbolb.Etiqueta(m), con m un
    nodo a la izqda de n.
  - arbolb.Etiqueta(n) <= arbolb.Etiqueta(m), con m un
    nodo a la drcha de n.
*/
```

TDA ABB: Constructores

```
template <class Tbase>
inline ABB<Tbase>::ABB()
{
}
```

```
template <class Tbase>
inline ABB<Tbase>::ABB(const ABB<Tbase> & a)
    : arbolb(a.arbolb)
{
}
```

```
template <class Tbase>
inline ABB<Tbase>::ABB(const Tbase & e)
    : arbolb(e)
{
}
```

TDA ABB: Existe

```
template <class Tbase>
bool ABB<Tbase>::Existe(const Tbase & e) const
{
  if (arbolb.Nulo())
    return false;

  ArbolBinario<Tbase>::Nodo n = arbolb.Raiz();
  bool encontrado = false;
  while (!encontrado &&
              (n != ArbolBinario<Tbase>::NODO_NULO))
    {
      if (e == arbolb.Etiqueta(n))
        encontrado = true;
      else if (e < arbolb.Etiqueta(n))
        n = arbolb.HijoIzqda(n);
      else
        n = arbolb.HijoDrcha(n);
    }
  return encontrado;
}
```

TDA ABB: Insertar

```
template <class Tbase>
void ABB<Tbase>::Insertar(const Tbase & e)
{
  if (arbolb.Nulo())
    {
      arbolb = ArbolBinario<Tbase>(e);
      return;
    }

  // Buscar la posición en la que insertar:
  // será un hijo de n
  ArbolBinario<Tbase>::Nodo n = arbolb.Raiz();
  bool posicionEncontrada = false;
  while (!posicionEncontrada)
    {
      if (e < arbolb.Etiqueta(n))
        {
          if (arbolb.HijoIzqda(n) !=
                    ArbolBinario<Tbase>::NODO_NULO)
            n = arbolb.HijoIzqda(n);
          else
            posicionEncontrada = true;
        }
      else
        {
```

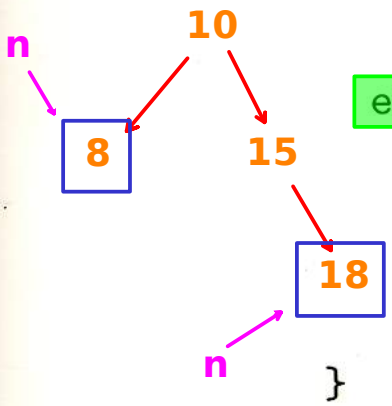
```
            if (arbolb.HijoDrcha(n) !=
                        ArbolBinario<Tbase>::NODO_NULO)
              n = arbolb.HijoDrcha(n);
            else
              posicionEncontrada = true;
          }
      }

    ArbolBinario<Tbase> a(e);
    if (e < arbolb.Etiqueta(n))
      arbolb.InsertarHijoIzqda(n, a);
    else
      arbolb.InsertarHijoDrcha(n, a);
}
```

TDA ABB: Borrar

```
template <class Tbase>
void ABB<Tbase>::Borrar(const Tbase & e)
{
  if (arbolb.Nulo())
    return;

  // Comprobar que la etiqueta "e" está en el árbol
  const ArbolBinario<Tbase>::Nodo NODO_NULO =
      ArbolBinario<Tbase>::NODO_NULO;
  ArbolBinario<Tbase>::Nodo n = arbolb.Raiz();
  bool encontrado = false;
  while (!encontrado && (n != NODO_NULO))
    {
      if (e == arbolb.Etiqueta(n))
        encontrado = true;
      else if (e < arbolb.Etiqueta(n))
        n = arbolb.HijoIzqda(n);

      else
        n = arbolb.HijoDrcha(n);
    }
  if (!encontrado)
    return;
  else
    borrar_nodo(n);
}
```

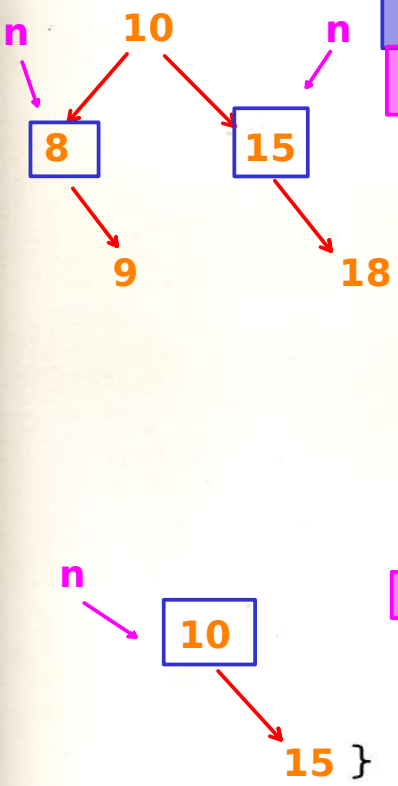
TDA ABB: borrar_nodo

```
template <class Tbase>
void ABB<Tbase>::borrar_nodo(ArbolBinario<Tbase>::Nodo n)
{
  const ArbolBinario<Tbase>::Nodo NODO_NULO =
    ArbolBinario<Tbase>::NODO_NULO;

  if (arbolb.HijoIzqda(n) == NODO_NULO)
    if (arbolb.HijoDrcha(n) == NODO_NULO)
      {   // Primer caso: el nodo es una hoja
        ArbolBinario<Tbase>::Nodo padre =
                                  arbolb.Padre(n);
        // Subcaso: el árbol sólo tiene un nodo
        if (padre == NODO_NULO)
          arbolb = ArbolBinario<Tbase>();
        else if (n == arbolb.HijoIzqda(padre))
          {
            ArbolBinario<Tbase> a;
            arbolb.PodarHijoIzqda(padre, a);
          }
        else
          {
            ArbolBinario<Tbase> a;
            arbolb.PodarHijoDrcha(padre, a);
          }
      }
```



TDA ABB: borrar_nodo

```
else // Segundo caso: El nodo sólo tiene
     //  un hijo a la drcha
{
  ArbolBinario<Tbase>::Nodo padre =
                              arbolb.Padre(n);
  if (padre != NODO_NULO)
    {
      ArbolBinario<Tbase> a;
      arbolb.PodarHijoDrcha(n, a);
      if (n == arbolb.HijoIzqda(padre))
        arbolb.InsertarHijoIzqda(padre, a);
      else
        arbolb.InsertarHijoDrcha(padre, a);
    }
  else
    arbolb.AsignarSubarbol(arbolb,
                              arbolb.HijoDrcha(n));
}
```

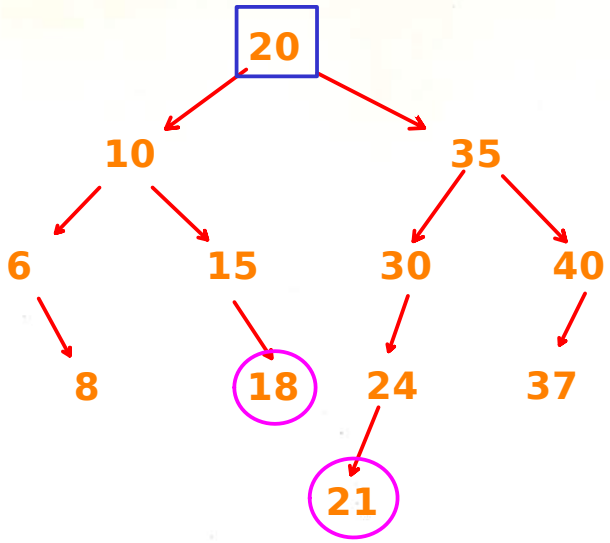


TDA ABB: borrar_nodo

```
else // (arbolb.HijoIzqda(n) != NODO_NULO)
  if (arbolb.HijoDrcha(n) == NODO_NULO)
    { // Tercer caso: El nodo sólo tiene un
      //  hijo a la izqda
      ArbolBinario<Tbase>::Nodo padre =
                                  arbolb.Padre(n);
      if (padre != NODO_NULO)
        {
          ArbolBinario<Tbase> a;
          arbolb.PodarHijoIzqda(n, a);
          if (n == arbolb.HijoIzqda(padre))
            arbolb.InsertarHijoIzqda(padre, a);
          else
            arbolb.InsertarHijoDrcha(padre, a);
        }
      else
        arbolb.AsignarSubarbol(arbolb,
                               arbolb.HijoIzqda(n));
    }
```

TDA ABB: borrar_nodo

```
    else // Cuarto caso: el nodo tiene dos hijos
      {
        ArbolBinario<Tbase>::Nodo mhi;
        // Buscar el mayor hijo a la izqda
        mhi = arbolb.HijoIzqda(n);
        while (arbolb.HijoDrcha(mhi) != NODO_NULO)
          mhi = arbolb.HijoDrcha(mhi);
        arbolb.Etiqueta(n) = arbolb.Etiqueta(mhi);
        borrar_nodo(mhi);
      }
}
```



TDA ABB: *Iterador*

```
template <class Tbase>
inline ABB<Tbase>::iterator::iterator()
{
}
```

```
template <class Tbase>
inline ABB<Tbase>::iterator::iterator(
                        ArbolBinario<Tbase>::Nodo n)
  : eliterador(n)
{
}
```

```
template <class Tbase>
inline ABB<Tbase>::iterator::iterator(
              ArbolBinario<Tbase>::iterator [redacted] it)
  : eliterador(it)
{
}
```

```
template <class Tbase>
inline bool ABB<Tbase>::iterator::operator!=(
                        const ABB<Tbase>::iterator & it)
{
  return eliterador != it.eliterador;
}
```

TDA ABB: *Iterador*

```
template <class Tbase>
inline bool ABB<Tbase>::iterator::operator==(
                    const ABB<Tbase>::iterator & it)
{
  return eliterador == it.eliterador;
}
```

```
template <class Tbase>
inline Tbase ABB<Tbase>::iterator::operator*()
{
  return *eliterador;
}
```

```
template <class Tbase>
inline ABB<Tbase>::iterator
                ABB<Tbase>::iterator::operator++()
{
  return ++eliterador;
}
```

```
template <class Tbase>
ABB<Tbase>::iterator ABB<Tbase>::begin()
{
  return iterator(arbolb.beginInOrden());
}
```

TDA ABB: *Iterador*

```
template <class Tbase>
ABB<Tbase>::iterator ABB<Tbase>::begin() const
{
   return iterator(arbolb.beginInOrden());
}
```

```
template <class Tbase>
ABB<Tbase>::iterator ABB<Tbase>::end()
{
   return iterator(arbolb.endInOrden());
}
```

```
template <class Tbase>
ABB<Tbase>::iterator ABB<Tbase>::end() const
{
   return iterator(arbolb.endInOrden());
}
```

```
template <class Tbase>
ABB<Tbase>::~ABB()
{
}
```