Implementación ABB

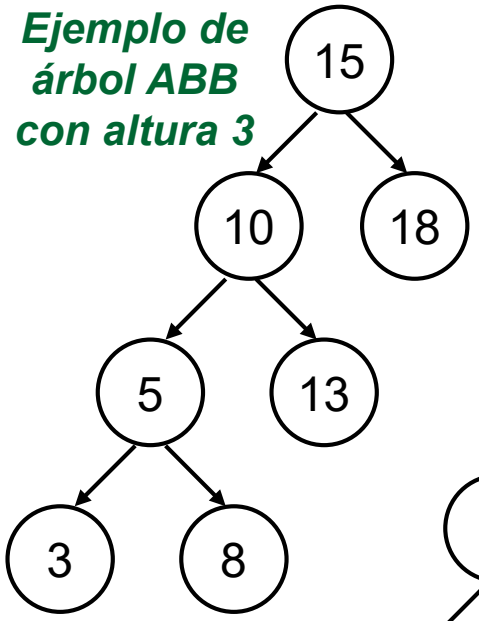
- Implementación de operaciones duplicar, liberar y recorridos?

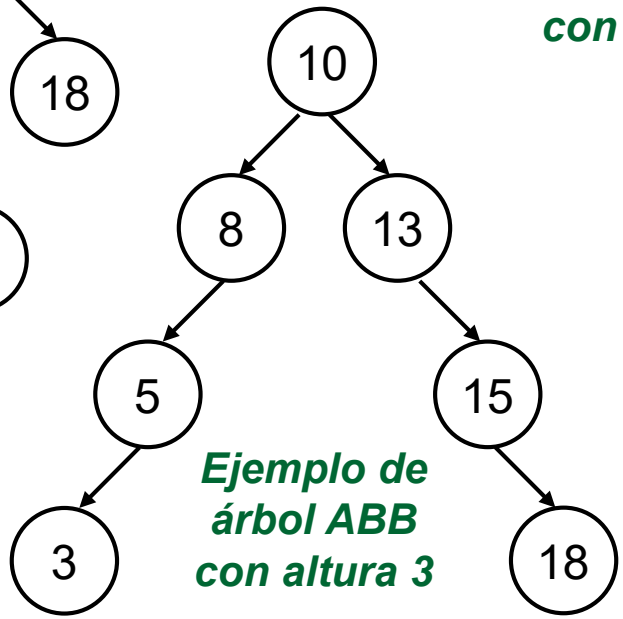
```
procedimiento duplicar(sal abb; ent viejo:abb)
{Duplica la representación del árbol viejo guardándolo en nuevo.}
procedimiento liberar(e/s a:abb)
{Libera la memoria dinámica accesible desde a, quedando a vacío.}
...
```

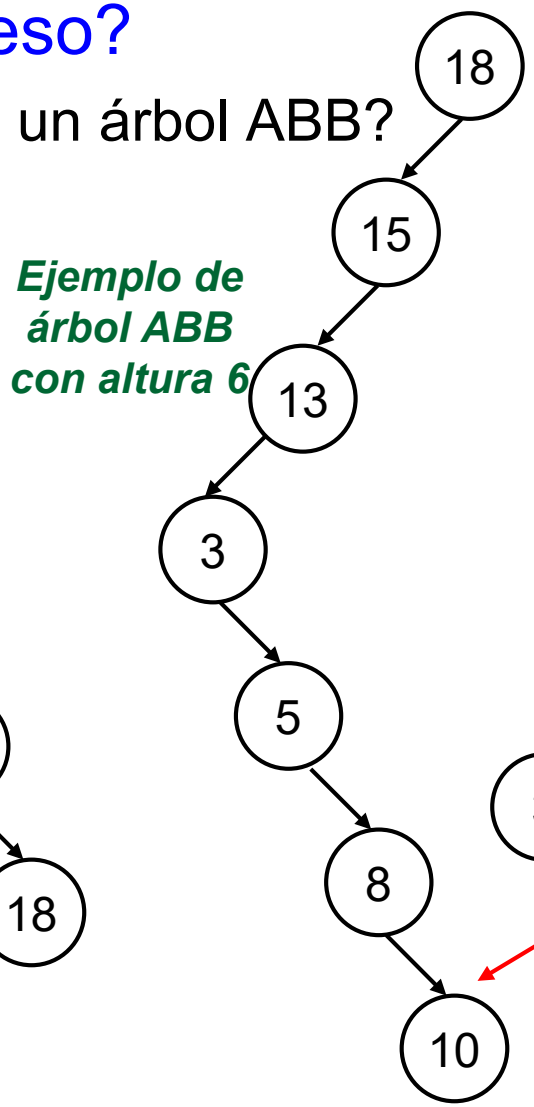
→ *Como las vistas para árboles binarios en la lección 12*

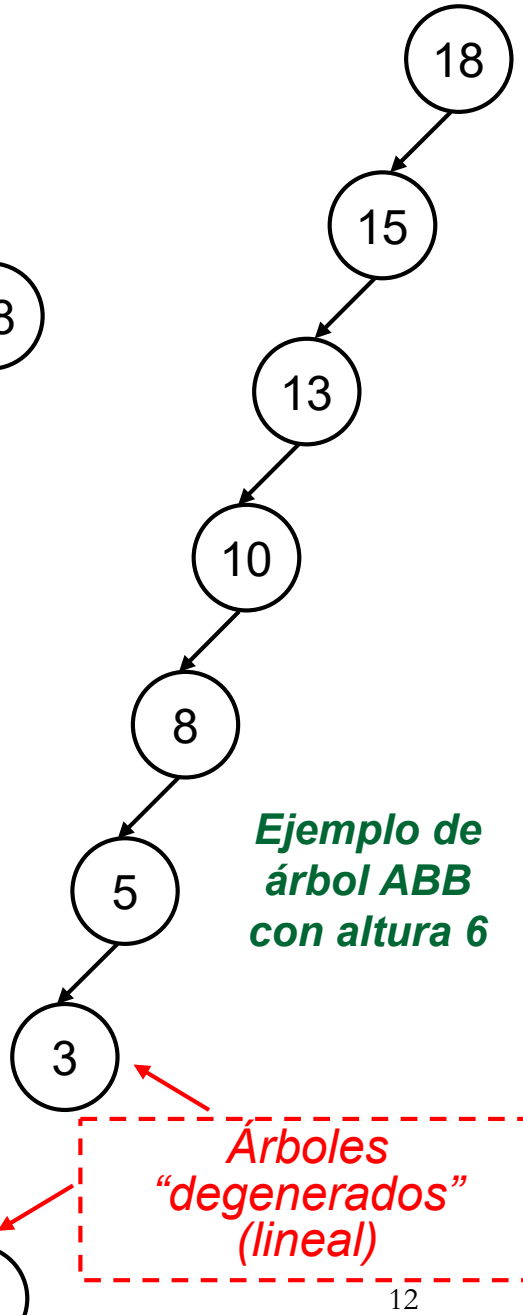
Implementación ABB

- Costes lineales en la altura del árbol….
  - ¿Cómo de bueno es eso?
    - ¿Cuál será la altura de un árbol ABB?





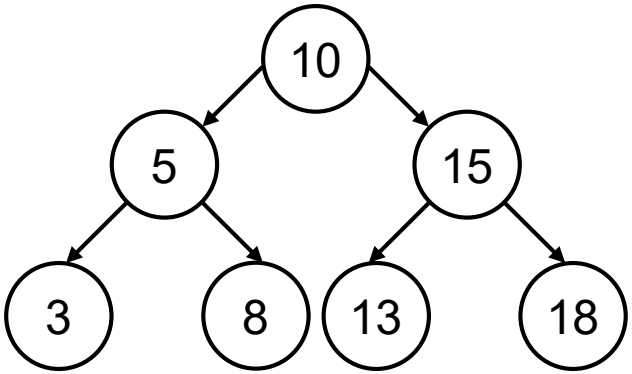




Implementación ABB

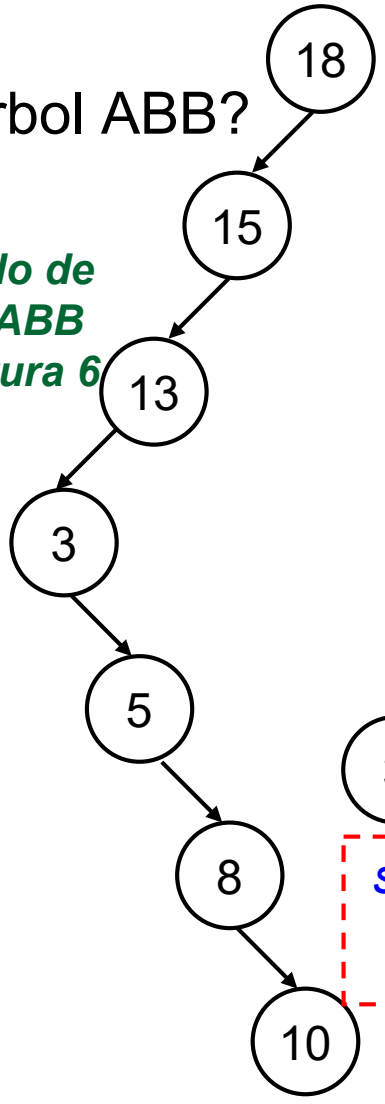
- Costes lineales en la altura del árbol….
  - ¿Cómo de bueno es eso?
    - ¿Cuál será la altura de un árbol ABB?

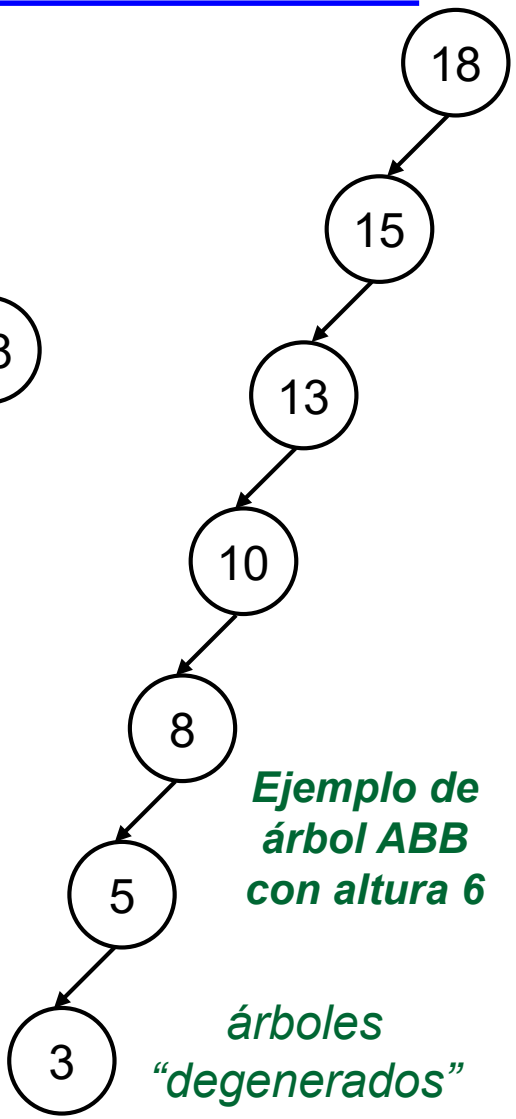
el "CASO MEJOR": Mínima altura=$log_2 N$



Si último nivel lleno:
árbol completo
(Equilibrio perfecto)

Ejemplo de árbol ABB con altura 6





Ejemplo de árbol ABB con altura 6

árboles "degenerados"

son el "CASO PEOR":

Máxima altura=(N-1)

Implementación ABB

- ¿Cómo se implementaría el TAD diccionario utilizando un *ABB para almacenar sus datos*?
  - Recordatorio:
    - Especificación de *diccionario vista en una lección anterior*
    - Los elementos serán parejas <clave,valor> y el diccionario no contendrá claves repetidas.
    - Operaciones relativas a la clave

Especificación Diccionarios

- **Es imprescindible que las claves se puedan comparar por igualdad**
- **No es imprescindible, pero si es habitual, que las claves se puedan comparar y ordenar**

  →Si se pueden ordenar las claves, el diccionario se puede organizar como una secuencia de elementos (clave, valor) ordenada por clave (secuencia ordenada sin repetidos) → *lineal*

espec diccionarios
  usa booleanos, naturales
  parámetros formales
    géneros clave, valor
    operación *{suponemos que en el género de las claves hay definida una función de comparación "<" y otra de igualdad "=" }*
      _=_: clave c1 , clave c2 -> booleano
      _<_: clave c1 , clave c2 -> booleano
  fpf
  género diccionario *{Los valores del TAD representan conjuntos de pares (clave,valor) en los que no se permiten claves repetidas}*
operaciones
    crear: → diccionario
    {Devuelve un diccionario vacío, sin elementos (pares)}

    añadir: diccionario d , clave c , valor v → diccionario
    {Si en d no hay ningún par con clave c, devuelve un diccionario igual al resultante de añadir el par (c,v) a d; si en d hay un par (c,v'), entonces devuelve un diccionario igual al resultante de sustituir (c,v') por el par (c,v) en d}
  …

Especificación Diccionarios

...

pertenece?: clave c , diccionario d → booleano
{Devuelve verdad si y sólo si en d hay algún par (c,v)}

parcial obtenerValor: clave c , diccionario d → valor
{Devuelve el valor asociado a la clave c en d.
Parcial: la operación no está definida si c no está en d }

quitar: clave c , diccionario d → diccionario
{Si c está en d, devuelve un diccionario igual al resultante de borrar en d el par con clave c; si c no está en d, devuelve un diccionario igual a d}

cardinal: diccionario d → natural
{Devuelve el nº de elementos (de pares) en el diccionario d}
esVacío?: diccionario d → booleano
{Devuelve verdad si y sólo si d no tiene elementos}

. . . ***{...operaciones de iterador,... }***

Especificación Diccionarios

... *{operaciones de iterador interno, que permitirá visitar los elementos del diccionario siguiendo el orden por clave (en este caso, de menor a mayor): }*

iniciarIterador: diccionario d → diccionario

{ Prepara el iterador y su cursor para que el siguiente elemento (par) a visitar sea el primero del diccionario d (situación de no haber visitado ningún elemento)}

existeSiguiente?: diccionario d → booleano

{ Devuelve falso si ya se ha visitado el último elemento, devuelve verdad en caso contrario}

parcial siguienteClave: diccionario d→ clave

{ Devuelve la clave del siguiente elemento (par) de d.
Parcial: la operación no está definida si no existeSiguiente?(d) }

parcial siguienteValor: diccionario d→ valor

{ Devuelve el valor del siguiente elemento (par) de d.
Parcial: la operación no está definida si no existeSiguiente?(d) }

parcial avanza: diccionario d → diccionario

{ Devuelve el diccionario resultante de avanzar el cursor en d.
Parcial: la operación no está definida si no existeSiguiente?(d) }

fespec

Implementación Diccionario. Interfaz.

```
módulo genérico diccionarioABB
usa pilasGenéricas {Implementación de la lección 9: pilas genéricas dinámicas)}
parámetros
   tipos clave,valor
   con
      función "<"(c1,c2:clave) devuelve booleano
     función "="(c1,c2:clave) devuelve booleano
exporta
   tipo diccionario {Conjunto de pares (clave,valor) en los que no se
  permiten claves repetidas, y cuenta con operaciones guiadas por la clave}
   procedimiento crear(sal d:diccionario)
   procedimiento añadir(e/s d:diccionario;
                        ent c:clave; ent v:valor)

   procedimiento buscar(ent d:diccionario; ent c:clave;
                        sal éxito:booleano; sal v:valor)
   {pertenece? y obtenerValor se implementan en una única operación}

   procedimiento quitar(ent c:clave; e/s d:diccionario)
   función cardinal(d:diccionario) devuelve natural
   función esVacío?(d:diccionario) devuelve booleano
   ...
```

Implementación Diccionario. Interfaz.

```
{Operaciones para duplicar y liberar}
procedimiento duplicar(sal dSal:diccionario;
                       ent dEnt:diccionario)

{Si hacemos una implementación en memoria dinámica,
 tendremos:}
procedimiento liberar(e/s d:diccionario)

{Operaciones de Iterador interno (en este caso, para recorrerlo en
 orden por clave, de menor a mayor): }
procedimiento iniciarIterador(e/s d:diccionario)
función existeSiguiente?(d:diccionario) devuelve booleano
procedimiento siguienteYavanza(e/s d:diccionario;
                               sal c:clave; sal v:valor;
                               sal error:booleano)
implementación
  ... {¿REPRESENTACION INTERNA?}
```

Implementación Diccionario: representación interna

```
implementación {¿REPRESENTACION INTERNA?}
tipos
    ptNodo= ↑nodo;
    nodo = registro
              laClave: clave;
              elValor: valor;
              izq,der: ptNodo
           freg
    modulo pila_de_ptNodo concreta pilasGenéricas(ptNodo);

    diccionario=registro
        raíz:ptNodo; {puntero a la raíz de un árbol ABB que
                      almacena los datos del diccionario}
        tamaño:natural;
        iter: pila_de_ptNodo.pila {pila de punteros a nodos del
                                   árbol, para el iterador}
    freg  ...
```

podría ser **dato: elemento;**

{Ver completa en la página de material de clase}

{Tipos de datos para implementar un ABB y un diccionario que almacene sus datos en el ABB}

Implementación Diccionario: implementación interna

```
...
procedimiento crear(sal d:diccionario)
principio
  d.raíz:=nil;
  d.tamaño:=0;
  crear(d.iter) {por seguridad: inicializar la pila del iterador}
fin


procedimiento añadir(e/s d:diccionario;
                          ent c:clave; ent v:valor)
variable añadido:booleano
principio
 añadirRec(d.raíz,c,v,añadido);
 si añadido entonces
   d.tamaño:=d.tamaño+1;
 fsi
fin

...
```

Implementación Diccionario: implementación interna

```
{Operación interna (oculta) NO DEBE aparecer en la Interfaz:
añadirRec: si en el árbol a no estaba almacenada ninguna clave c, la
  inserta adecuadamente en el árbol ABB a (junto con su valor). Si ya
  estaba almacenada la clave c, actualiza su valor con v}
procedimiento añadirRec (e/s a:ptNodo; ent c:clave; ent v:valor;
                                    sal nuevo:booleano)
principio
  si a=nil entonces
     nuevoDato(a);
     a↑.laClave:=c; a↑.elValor:=v;
     a↑.izq:=nil; a↑.der:=nil;
     nuevo:=verdad;
  sino
     si c<a↑.laClave entonces
          añadirRec(a↑.izq,c,v,nuevo)
     sino si a↑.laClave<c entonces
          añadirRec(a↑.der,c,v,nuevo)
     sino {c=a↑.laClave, y en un diccionario no se pueden repetir}
          a↑.elValor:=v;
          nuevo:=falso;
     fsi
  fsi
fin                    ...
```

{De forma similar se implementarán las otras operaciones que requieren el código recursivo sobre el árbol (que quedará encapsulado en una operación interna y oculta). Ver la implementación completa en la página del material de clase}

Implementación Diccionario: implementación interna

```
...
procedimiento iniciarIterador(e/s d: diccionario)
  variable aux:ptNodo
principio
   {eliminar restos de recorridos anteriores (no exhaustivos) y
    empezar con pila vacía:}
   liberar(d.iter);
   aux:=d.raíz; {raíz del árbol}
   mq aux≠nil hacer
      {apila puntero a nodo del árbol, en la pila del iterador:}
      apilar(d.iter,aux);
      aux:=aux↑.izq
   fmq
fin

función haySiguiente(d:diccionario) devuelve booleano
principio
  devuelve not esVacía(d.iter) {hay siguiente si d.iter es no vacía}
fin
```

Implementación Diccionario: implementación interna

```
procedimiento siguienteYavanza(e/s d:diccionario;
                    sal c:clave; sal v:valor; sal error:booleano)
     variable aux:ptNodo
principio
    si haySiguiente(d) entonces
        error:=falso; aux:=cima(d.iter);
        c:=aux↑.laClave; {clave del siguiente elemento visitado}
        v:=aux↑.elValor; {valor del siguiente elemento visitado}
       {avanzar: }
        desapilar(d.iter);
        aux:=aux↑.der;
        mq aux≠nil hacer
            apilar(d.iter,aux); aux:=aux↑.izq
        fmq
     sino
        error:=verdad
   fsi
fin
```

{Ver la implementación completa del módulo en la página del material de clase}

```
{Si se añadiese la operación de Iguales: verdad sii d1 y d2 tienen exactamente la misma colección de pares (clave,valor)}
función iguales?(d1,d2:diccionario) devuelve booleano
  variables igual:booleano; {y para recorrerlos simultáneamente en inorden:}
            pAux1,pAux2:ptNodo; pilaD1,pilaD2:pila_de_ptNodo.pila;
principio
    si esVacío?(d1) and esVacío?(d2) entonces  devuelve verdad
    sino_si cardinal(d1)≠cardinal(d2) entonces devuelve falso
    sino  {ambos tienen el mismo número (no nulo) de claves:
           recorrerlos simultáneamente en orden por clave (inorden)}
      igual:= verdad;
      {inicializar recorrido en diccionario d1:}
      {Llegar hasta el primero y prepararlo para que sea el siguiente a tratar:}
        crear(pilaD1);
        pAux1:=d1.raíz;
        mq pAux1≠nil hacer
           apilar(pilaD1,pAux1);   pAux1:=pAux1↑.izq
        fmq;
      {Ídem en diccionario d2:}
        crear(pilaD2);
        pAux2:=d2.raíz;
        mq pAux2≠nil hacer
           apilar(pilaD2,pAux2);   pAux2:=pAux2↑.izq
        fmq;
      . . .
```

```
    . . . {Mientras sean iguales y no se hayan tratado todos:}
    mientrasQue igual and not esVacía(pilaD1) and not esVacía(pilaD2)
    hacer
       {Obtener el siguiente que toca tratar de cada diccionario: }
       pAux1:=cima(pilaD1);  pAux2:=cima(pilaD2);
       {Compararlos: }
       igual:=(pAux1↑.laClave=pAux2↑.laClave) and (pAux1↑.elValor=pAux2↑.elValor);
       {Avanzar, para preparar el siguiente dato a tratar en diccionario d1:}
          desapilar(pilaD1);
          pAux1:=pAux1↑.der;
          mq pAux1≠nil hacer
            apilar(pilaD1,pAux1);   pAux1:=pAux1↑.izq
          fmq
       {Idem en diccionario d2:}
          desapilar(pilaD2);
          pAux2:=pAux2↑.der;
          mq pAux2≠nil hacer
            apilar(pilaD2,pAux2);   pAux2:=pAux2↑.izq
          fmq
    fmq;
    si not igual entonces liberar(pilaD1); liberar(pilaD2) fsi;
    devuelve igual
  fsi
fin
```

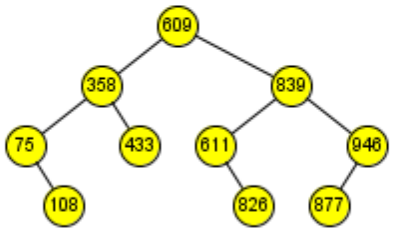
http://people.ksp.sk/~kuko/gnarley-trees/ | Download Java for Windows | Gnarley trees | visualization ...

Gnarley Trees

Data structures Language

Data structures > Dictionaries > binary search Tree

Binary search tree



Insert 826

1. We start at the root.
2. Since 826 > 609, we insert it in the right subtree.
3. Since 826 < 839, we insert it in the left subtree.
4. Since 826 > 611, we insert it in the right subtree.

Done.

Una buena aplicación para visualizar árboles ABBs y su funcionamiento:

nueva dirección: https://kubokovac.eu/gnarley-trees/BST.html

(se puede descargar como applet y ejecutar localmente)

Insert | Find | Delete | Previous | Next

Pause | Clear | Random | Show order

Size: 10; Height: 4 = 1.00·opt; Ave. depth: 2.90

couldn't parse an integer; using the default value 10