Estructuras de Datos y Algoritmos

Árboles Binarios de Búsqueda (ABB)

Lección 13

Universidad Zaragoza

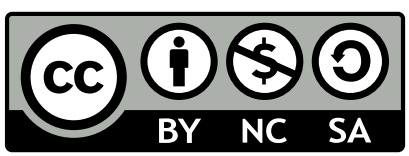
Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2023/2024

Grado en Ingeniería Informática

Universidad de Zaragoza

Aula 0.04, Edificio Agustín de Betancourt



Adaptadas de diapositivas de Javier Campos

Índice

Universidad Zaragoza

Índice

Universidad Zaragoza

Árboles binarios de búsqueda

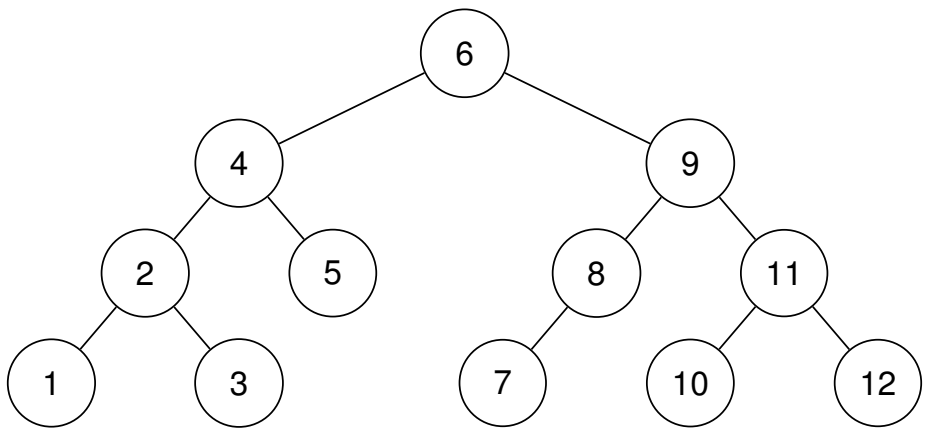
- Existe una relación de orden en el tipo de los elementos del árbol
  - Relación de tipo $\leq$ (o $<$, si evitamos elementos repetidos)
- Todo árbol vacío es un árbol binario de búsqueda
- Propiedades:
  - El elemento raíz del árbol es $\geq$ que todos los elementos de su subárbol izquierdo
    - Si no se permiten elementos repetidos, será estrictamente mayor
  - El elemento raíz del árbol es menor que todos los elementos de su subárbol derecho
  - Los subárboles izquierdo y derecho son a su vez árboles binarios de búsqueda

Universidad Zaragoza

Árboles binarios de búsqueda

Ventajas

- Mantienen los elementos ordenados (de alguna forma)
- Operaciones habituales: inserción, búsqueda y borrado
- Búsqueda eficiente de elementos
- Recorrido en orden de los elementos: *recorrido in-orden*
  1. Se recorre (en *in*-orden) el hijo izquierdo
  2. Se visita la raíz
  3. Se recorre (en *in*-orden) el hijo derecho



Universidad Zaragoza

Índice

Universidad Zaragoza

Aplicaciones: implementación de multiconjuntos

```
espec abbs
    usa booleanos
    parámetro formal
        género elemento
        operación _<_: elemento e1, elemento e2 -> booleano
        { ídem con el resto de operaciones relacionales ≤, =, >, ≥ }
    fpf
género abb
{ Multiconjunto de elementos, almacenados en árbol binario, con operaciones de
    inserción, búsqueda y borrado. Es decir, puede haber elementos repetidos }
operaciones
    vacío: -> abb
    {Devuelve el árbol vacío}
    esVacío?: abb a -> booleano
    {Devuelve verdad si y sólo si a es el árbol vacío}
    insertar: abb a, elemento e -> abb
    {Devuelve el abb resultante de añadir e al árbol a, manteniendo la
        propiedad de abb}
    está?: abb a, elemento e -> booleano
    {Devuelve verdad si y sólo si e está en a}
    borrar: abb a, elemento e -> abb
    {Si e está en a, devuelve un árbol igual al resultante de borrar una de las
        apariciones de e en a. Si e no está en a, devuelve un árbol igual que a}
fespec
```

Aplicaciones: implementación de multiconjuntos

```
módulo árbolesBinariosBúsqueda { implementa la especificación abbs}
parámetros tipo elemento
     función "<"(e1, e2: elemento) devuelve booleano
    { ... ídem resto funciones: <=, =, >, >= }
exporta
    tipo abb
    procedimiento vacío(sal a: abb)
    función esVacío?(a: abb) devuelve booleano
    procedimiento insertar(e/s a: abb; ent e: elemento)
    función está?(a: abb; e: elemento) devuelve booleano
    procedimiento borrar(e/s a: abb; ent e: elemento)
implementación
tipos abb = ↑nodo
     nodo = registro
                dato: elemento;
                izq, der: abb
            freg
...
```

```
...
    procedimiento vacío(sal a: abb)
    principio
        a := nil
    fin

    función esVacío(a: abb) devuelve booleano
    principio
        devuelve a = nil
    fin

    procedimiento insertar(e/s a: abb; ent e: elemento)
    principio
        si a = nil entonces
            nuevoDato(a);
            a↑.dato := e;
            a↑.izq := nil;
            a↑.der := nil
        sino
            si e <= a↑.dato entonces
                insertar(a↑.izq, e)
            sino
                insertar(a↑.der, e)
            fsi
        fsi
    fin
...
```

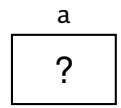
*¿Qué cambiaría si quisiéramos implementar ABBs **SIN** elementos repetidos?*
(es decir, conjuntos de elementos)

```
...
    función está?(a: abb; e: elemento) devuelve booleano
    principio
        si a = nil entonces
            devuelve falso
        sino
            selección
                e < a↑.dato: devuelve está?(a↑.izq, e);
                e = a↑.dato: devuelve verdad;
                e > a↑.dato: devuelve está?(a↑.der, e)
            fselección
        fsi
    fin

    { Este procedimiento es auxiliar para el procedimiento borrar }
    procedimiento borrarMáx(e/s a: abb; sal e: elemento)
    {Precondición: a es no vacío.
    Devuelve en e un elemento máximo de a y lo borra de a}
    variable aux: abb
    principio
        si a↑.der = nil entonces {el máximo del árbol está en la raíz}
            e := a↑.dato;
            aux := a;
            a := a↑.izq;
            disponer(aux)
        sino {el máximo del árbol está en el subárbol derecho}
            borrarMáx(a↑.der, e)
        fsi
    fin
```

```
...
    procedimiento borrar(e/s a: abb; ent e: elemento)
    variable aux: abb
    principio
        si a ≠ nil entonces
            selección
                e < a↑.dato: borrar(a↑.izq, e);
                e > a↑.dato: borrar(a↑.der, e);
                e = a↑.dato: si a↑.izq = nil entonces
                                 aux := a;
                                 a := a↑.der;
                                 disponer(aux)
                             sino
                                 borrarMáx(a↑.izq, a↑.dato)
                             fsi
            fselección
        fsi
    fin
fin { del módulo }
```

Aplicaciones: implementación de multiconjuntos
Ejemplo

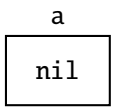
```
a: abbDeEnteros;
```



Aplicaciones: implementación de multiconjuntos

Ejemplo

```
a: abbDeEnteros;

vacío(a);
```

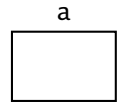



```
procedimiento vacío(sal a: abb)
principio
    a := nil
fin
```

Aplicaciones: implementación de multiconjuntos

Ejemplo

```
a: abbDeEnteros;

vacío(a);
insertar(a, 15);
```

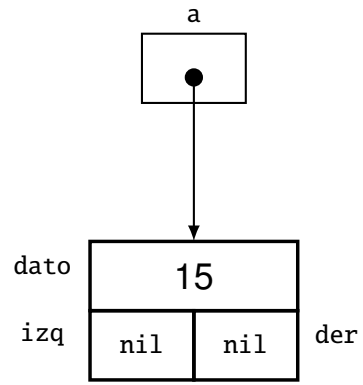



```
procedimiento insertar(e/s a: abb; ent e: elemento)
principio
    si a = nil entonces
        nuevoDato(a);
        a↑.dato:=e;
        a↑.izq:=nil;
        a↑.der:=nil
    sino
        si e <= a↑.dato entonces
            insertar(a↑.izq,e)
        sino
            insertar(a↑.der,e)
        fsi
    fsi
fin
```

Aplicaciones: implementación de multiconjuntos

Ejemplo

```
a: abbDeEnteros;

vacío(a);
insertar(a, 15);
insertar(a, 8);
```

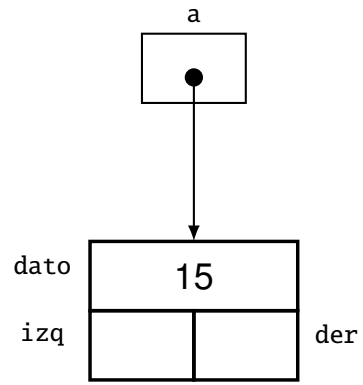



```
procedimiento insertar(e/s a: abb; ent e: elemento)
principio
    si a = nil entonces
        nuevoDato(a);
        a↑.dato:=e;
        a↑.izq:=nil;
        a↑.der:=nil
    sino
        si e <= a↑.dato entonces
            insertar(a↑.izq,e)
        sino
            insertar(a↑.der,e)
        fsi
    fsi
fin
```

Aplicaciones: implementación de multiconjuntos

Ejemplo

```
a: abbDeEnteros;

vacío(a);
insertar(a, 15);
insertar(a, 8);
insertar(a, 3);
```

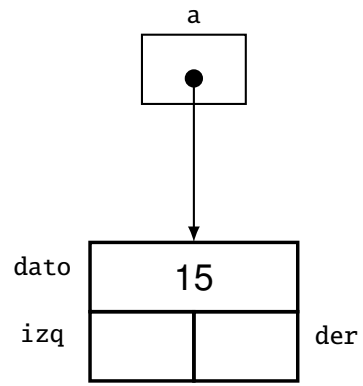



```
procedimiento insertar(e/s a: abb; ent e: elemento)
principio
    si a = nil entonces
        nuevoDato(a);
        a↑.dato:=e;
        a↑.izq:=nil;
        a↑.der:=nil
    sino
        si e <= a↑.dato entonces
            insertar(a↑.izq,e)
        sino
            insertar(a↑.der,e)
        fsi
    fsi
fin
```

Aplicaciones: implementación de multiconjuntos

Ejemplo

```
a: abbDeEnteros;

vacío(a);
insertar(a, 15);
insertar(a, 8);
insertar(a, 3);
insertar(a, 20);
```

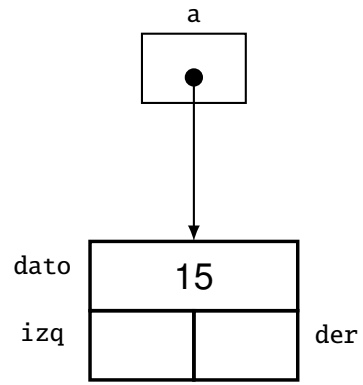



```
procedimiento insertar(e/s a: abb; ent e: elemento)
principio
    si a = nil entonces
        nuevoDato(a);
        a↑.dato:=e;
        a↑.izq:=nil;
        a↑.der:=nil
    sino
        si e <= a↑.dato entonces
            insertar(a↑.izq,e)
        sino
            insertar(a↑.der,e)
        fsi
    fsi
fin
```

Aplicaciones: implementación de multiconjuntos

Ejemplo

```
a: abbDeEnteros;

vacío(a);
insertar(a, 15);
insertar(a, 8);
insertar(a, 3);
insertar(a, 20);
insertar(a, 12);
```

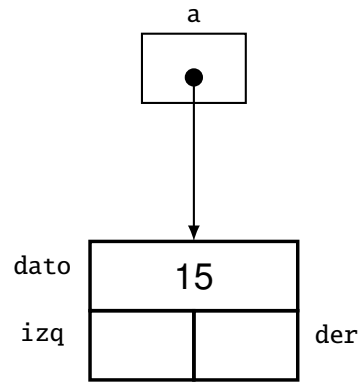



```
procedimiento insertar(e/s a: abb; ent e: elemento)
principio
    si a = nil entonces
        nuevoDato(a);
        a↑.dato:=e;
        a↑.izq:=nil;
        a↑.der:=nil
    sino
        si e <= a↑.dato entonces
            insertar(a↑.izq,e)
        sino
            insertar(a↑.der,e)
        fsi
    fsi
fin
```

Aplicaciones: implementación de multiconjuntos

Ejemplo

```
a: abbDeEnteros;

vacío(a);
insertar(a, 15);
insertar(a, 8);
insertar(a, 3);
insertar(a, 20);
insertar(a, 12);
insertar(a, 1);
```

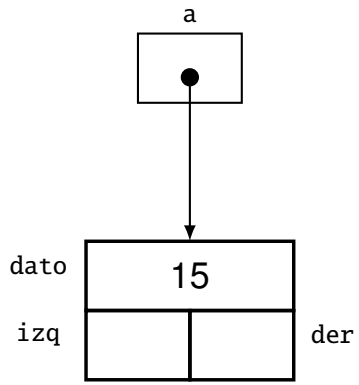



```
procedimiento insertar(e/s a: abb; ent e: elemento)
principio
    si a = nil entonces
        nuevoDato(a);
        a↑.dato:=e;
        a↑.izq:=nil;
        a↑.der:=nil
    sino
        si e <= a↑.dato entonces
            insertar(a↑.izq,e)
        sino
            insertar(a↑.der,e)
        fsi
    fsi
fin
```

Aplicaciones: implementación de multiconjuntos

Ejemplo

```
a: abbDeEnteros;

vacío(a);
insertar(a, 15);
insertar(a, 8);
insertar(a, 3);
insertar(a, 20);
insertar(a, 12);
insertar(a, 1);
borrar(a, 3);
```

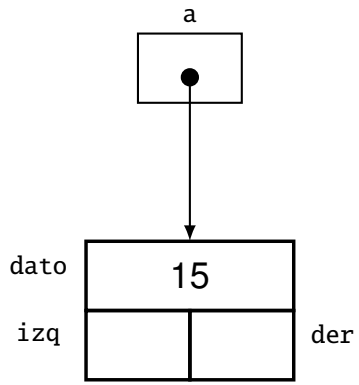



```
procedimiento borrarMáx(e/s a: abb; sal e: elemento)
{Precondición: a es no vacío.
Devuelve en e un elemento máximo de a y lo borra de a}
variable aux: abb
principio
    si a↑.der = nil entonces {el máximo del árbol está en la raíz}
        e := a↑.dato;
        aux := a;
        a := a↑.izq;
        disponer(aux)
    sino {el máximo del árbol está en el subárbol derecho}
        borrarMáx(a↑.der, e)
    fsi
fin

procedimiento borrar(e/s a: abb; ent e: elemento)
variable aux: abb
principio
    si a ≠ nil entonces
        selección
            e < a↑.dato: borrar(a↑.izq, e);
            e > a↑.dato: borrar(a↑.der, e);
            e = a↑.dato: si a↑.izq = nil entonces
                            aux := a;
                            a := a↑.der;
                            disponer(aux)
                         sino
                             borrarMáx(a↑.izq, a↑.dato)
                         fsi
        fselección
    fsi
fin
```

Aplicaciones: implementación de multiconjuntos

Ejemplo

```
a: abbDeEnteros;

vacío(a);
insertar(a, 15);
insertar(a, 8);
insertar(a, 3);
insertar(a, 20);
insertar(a, 12);
insertar(a, 1);
borrar(a, 8);
```




```
procedimiento borrarMáx(e/s a: abb; sal e: elemento)
{Precondición: a es no vacío.
Devuelve en e un elemento máximo de a y lo borra de a}
variable aux: abb
principio
    si a↑.der = nil entonces {el máximo del árbol está en la raíz}
        e := a↑.dato;
        aux := a;
        a := a↑.izq;
        disponer(aux)
    sino {el máximo del árbol está en el subárbol derecho}
        borrarMáx(a↑.der, e)
    fsi
fin

procedimiento borrar(e/s a: abb; ent e: elemento)
variable aux: abb
principio
    si a ≠ nil entonces
        selección
            e < a↑.dato: borrar(a↑.izq, e);
            e > a↑.dato: borrar(a↑.der, e);
            e = a↑.dato: si a↑.izq = nil entonces
                            aux := a;
                            a := a↑.der;
                            disponer(aux)
                         sino
                             borrarMáx(a↑.izq, a↑.dato)
                         fsi
        fselección
    fsi
fin
```

Aplicación: implementación de diccionarios

```
tipos
    ptNodo = ↑nodo {almacenamos el diccionario en un ABB ordenado por claves}
    nodo   = registro
                laClave: clave; {no podrá haber 2 nodos con igual clave}
                elValor: valor;
                izq, der: ptNodo
             freg
    diccionario = registro
                    raíz: ptNodo; {puntero a la raíz del árbol}
                    tamaño: natural; {nº de pares (clave, valor)}
                    iter: pila {pila de datos de tipo ptNodo,
                                para implementar el iterador}
                  freg
```

Aplicación: implementación de diccionarios

```
procedimiento crear(sal d: diccionario)
{Devuelve en d un diccionario vacío, sin elementos}
principio
    d.raíz := nil;
    d.tamaño := 0
fin
```

```
procedimiento añadir(e/s d: diccionario; ent c: clave; ent v: valor)
{Si en d no hay ningún par con clave c, añade a d el par (c, v);
si en d hay un par (c, v'), entonces lo sustituye por (c, v)}
variable
    nuevo: booleano
principio
    añadirRec(d.raíz, c, v, nuevo);
    si nuevo entonces
        d.tamaño := d.tamaño + 1
    fsi
fin

procedimiento añadirRec(e/s p: ptNodo; ent c: clave; ent v: valor; sal nuevo: booleano}
{Si en el árbol de raíz apuntada por p no hay ningún par con clave c, añade a ese árbol
el par (c, v) y nuevo = verdad.
Si en el árbol de raíz apuntada por p hay un par (c, v'), entonces lo sustituye por el par (c,v)
y nuevo = falso}
variables
    p: ptNodo
principio
    si p = nil entonces
        nuevoDato(p);
        p↑.laClave := c;
        p↑.elValor := v;
        p↑.izq := nil;
        p↑.der := nil;
        nuevo := verdad
    sino {p ≠ nil}
        si c < p↑.laClave entonces
            añadirRec(p↑.izq, c, v, nuevo)
        sino_si c > p↑.laClave entonces
            añadirRec(p↑.der, c, v, nuevo)
        sino {c = p↑.laClave, es decir, ya existía un par con esa clave}
            p↑.elValor := v;
            nuevo := falso
        fsi
    fsi
fin
```

```
procedimiento buscar(ent d: diccionario; ent c: clave; sal éxito: booleano; sal v: valor)
{Devuelve éxito = verdad si en d hay algún par con clave c, falso en caso contrario.
En caso de éxito, además, devuelve en v el valor asociado a la clave c en d}
principio
    buscarRec(d.raíz, c, éxito, v)
fin

procedimiento buscarRec(ent p: ptNodo; ent c: clave; sal éxito: booleano; sal v: valor)
{Devuelve éxito = verdad si en el árbol de raíz apuntada por p hay algún par con clave c,
falso en caso contrario. En caso de éxito, además, devuelve en v el valor asociado a la clave c}
principio
    si p = nil entonces
        éxito := falso
    sino
        selección
            c < p↑.laClave: buscarRec(p↑.izq, c, éxito, v);
            c = p↑.laClave: éxito := verdad;
                            v := p↑.elValor;
            c > p↑.laClave: buscarRec(p↑.der, c, éxito, v)
        fselección
    fsi
fin
```

```
procedimiento borrar(ent c: clave; e/s d: diccionario)
{Si en d hay un par con clave c, lo borra. En caso contrario, d no se modifica}
variable
    borrado: booleano
principio
    borrarRec(c, d.raíz, borrado)
    si borrado entonces
        d.tamaño := d.tamaño - 1
    fsi
fin

procedimiento borrarRec(ent c: clave; e/s p: ptNodo; sal borrado: booleano)
{Si en el árbol de raíz apuntada por p hay un par con clave c, lo borra y borrado = verdad.
En caso contrario, p no se modifica y borrado = falso}
variable
    aux: ptNodo
principio
    si p = nil entonces
        borrado := falso
    sino
        selección
            c < p↑.laClave: borrarRec(c, p↑.izq, borrado);
            c > p↑.laClave: borrarRec(c, p↑.der, borrado);
            c = p↑.laClave:
                si p↑.izq = nil entonces
                    aux := p;
                    p := p↑.der;
                    disponer(aux)
                sino
                    borrarMáximo(p↑.izq, p↑.laClave, p↑.elValor)
                fsi;
                borrado := verdad
        fselección
    fsi
fin
```

```
procedimiento borrarMáximo(e/s p: ptNodo; sal c: clave; sal v: valor)
{Precondición: p es no vacío. Devuelve el par (c,v) cuya clave c es la máxima del
subárbol de raíz p y borra ese par del subárbol de raíz p}
variable
    aux: ptNodo
principio
    si p↑.der = nil entonces
        c := p↑.laClave;
        v := p↑.elValor;
        aux := p;
        p := p↑.izq;
        disponer(aux)
    sino
        borrarMáximo(p↑.der, c, v)
    fsi
fin

función cardinal(d: diccionario) devuelve natural
{Devuelve el nº de pares del diccionario d}
principio
    devuelve d.tamaño
fin

función esVacío(d: diccionario) devuelve booleano
{Devuelve verdad si y sólo si d no tiene pares}
principio
    devuelve d.raíz = nil
fin
```

```
procedimiento duplicar(sal dSal: diccionario; ent dEnt: diccionario)
{Duplica la representación del diccionario dEnt en dSal}
principio
    duplicarRec(dSal.raíz, dEnt.raíz);
    dSal.tamaño := dEnt.tamaño
fin

procedimiento duplicarRec(sal pSal: ptNodo; ent pEnt: ptNodo)
{Duplica el árbol apuntado por pEnt en el árbol apuntado por pSal}
principio
    si pEnt = nil entonces
        pSal := nil
    sino
        nuevoDato(pSal);
        pSal↑.laClave := pEnt↑.laClave;
        pSal↑.elValor := pEnt↑.elValor;
        duplicarRec(pSal↑.izq, pEnt↑.izq);
        duplicarRec(pSal↑.der, pEnt↑.der)
    fsi
fin
```

```
función iguales(d1, d2: diccionario) devuelve booleano
{Devuelve verdad si y sólo si los diccionarios d1 y d2 tienen los mismos pares
de claves y valores}
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 pares}
        devuelve igualesRec(d1.raíz, d2.raíz)
    fsi
fin

función igualesRec(p1, p2: ptNodo) devuelve booleano
{Devuelve verdad si y sólo si todas las parejas <clave,valor> de p1 están en p2 }
variable
    éxito: booleano; v: valor
principio
    si p1 = nil entonces
        devuelve verdad {trivialmente}
    sino
        buscarRec(p2, p1↑.laClave, éxito, v); {busca en p2 la raíz de p1}
        si éxito andThen v = p1↑.elValor entonces {la raíz de p1 está en p2}
            devuelve igualesRec(p1↑.izq, p2) andThen igualesRec(p1↑.der, p2)
        sino
            devuelve falso
        fsi
    fsi
fin
```

Alternativa: *usar los iteradores*

```
procedimiento liberar(e/s d: diccionario)
{Devuelve en d el diccionario vacío y además libera la memoria
utilizada previamente por d}
principio
    liberarRec(d.raíz);
    d.tamaño := 0;
    d.raíz := nil
fin

procedimiento liberarRec(e/s p: ptNodo)
{Libera toda la memoria utilizada por el árbol apuntado por p}
principio
    si p ≠ nil entonces
        liberarRec(p↑.izq);
        liberarRec(p↑.der);
        disponer(p)
    fsi
fin
```

```
procedimiento iniciarIterador(e/s d: diccionario)
{Inicializa el iterador para recorrer los pares del diccionario d,
de forma que el siguiente par a visitar sea el primero que
visitamos (situación de no haber visitado ningún par).}
variable
    aux: ptNodo
principio
    crearVacía(d.iter); {crea pila vacía de punteros a nodos del árbol}
    aux := d.raíz; {raíz del árbol}
    mientrasQue aux ≠ nil hacer
        apilar(d.iter, aux); {apila el puntero aux (a un nodo del árbol)
                              en la pila del iterador}
        aux := aux↑.izq
    fmq
fin

función existeSiguiente(d: diccionario) devuelve booleano
{Devuelve falso si ya se han visitado todos los pares de d; verdad en otro caso}
principio
    devuelve not esVacía(d.iter) {existe siguiente si la pila del iterador
                                  es no vacía}
fin
```

```
procedimiento siguiente(e/s d: diccionario; sal c:clave; sal v: valor; sal error: booleano)
{Si existe algún par de d pendiente de visitar, devuelve en c y v la clave y el valor,
respectivamente, del siguiente par a visitar y error=falso, y además avanza el iterador
para que a continuación se pueda visitar otro par de d. Si no quedan pares pendientes
de visitar devuelve error=verdad, c y v quedan indefinidos y d queda como estaba}
variable
    aux: ptNodo
principio
    si existeSiguiente(d) entonces
        error := falso;
        aux := cima(d.iter);
        desapilar(d.iter);
        c := aux↑.laClave; {es la clave del siguiente elemento visitado}
        v := aux↑.elValor; {y el valor asociado a la clave}
        aux := aux↑.der;
        mientrasQue aux ≠ nil hacer
            apilar(d.iter, aux);
            aux := aux↑.izq
        fmq
    sino
        error := verdad
    fsi
fin
```

Trabajo encargado

Leer lección 13. Ejercicios hoja 1, 2, 3 y 4 de la hoja de problemas 3

Estructuras de Datos y Algoritmos

Árboles Binarios de Búsqueda (ABB)

Lección 13

Universidad Zaragoza

1542

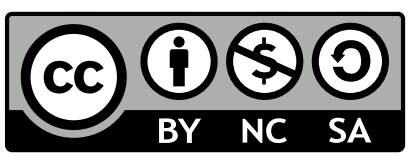
Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2023/2024

Grado en Ingeniería Informática

Universidad de Zaragoza

Aula 0.04, Edificio Agustín de Betancourt



Adaptadas de diapositivas de Javier Campos