Algoritmos y Estructuras de Datos II

Árboles binarios de búsqueda

6 de mayo de 2013

Clase de hoy

1. Parcial del 29 de abril
2. Repaso
   - Árboles binarios
   - Especificación
   - Terminología habitual
   - Implementación con punteros
   - Posiciones
3. Árbol binario de búsqueda
   - Ejemplos y definiciones
   - TAD diccionario
4. Anuncio de charla

Ejercicio 1
proc A

```
proc A(in/out a: array[1..n] of T)
    var x: nat
    for i:= 1 to n-1 do
        x:= i
        for j:= i+1 to n do
            if a[j] < a[x] then x:= j fi
        od
        swap(a,i,x)
    od
end proc
```

Ejercicio 1

proc A = ordenación por selección

```
proc selection_sort(in/out a: array[1..n] of T)
    var minp: nat
    for i:= 1 to n-1 do
        minp:= i
        for j:= i+1 to n do
            if a[j] < a[minp] then minp:= j fi
        od
        swap(a,i,minp)
    od
end proc
```

Ejercicio 1
proc B

```
proc B (in/out a: array[1..n] of T)
    var i: nat
    var q: bool
    q:= true
    i:= 1
    do i ≤ n-1 ∧ q →
        q:= false
        for j:= n-1 downto i do
            if a[j] > a[j+1] then swap(a,j,j+1)
                                  q:= true
            fi
        od
        i:= i+1
    od
end proc
```

Ejercicio 1

proc B = bubble con interrupción

```
proc bubble_sort (in/out a: array[1..n] of T)
    var i: nat
    var swapped: bool
    swapped:= true
    i:= 1
    do i ≤ n-1 ∧ swapped →
        swapped:= false
        for j:= n-1 downto i do
            if a[j] > a[j+1] then swap(a,j,j+1)
                                  swapped:= true
            fi
        od
        i:= i+1
    od
end proc
```

Ejercicio 1
proc C

```
proc C (in/out a: array[1..n] of T)
        var j: nat
        for i:= 2 to n do
            j:= i
            do j > 1 ∧ a[j] < a[j − 1] → swap(a,j,j-1)
                                         j:= j-1
            od
        od
end proc
```

Ejercicio 1

proc C = ordenación por inserción

```
proc insertion_sort (in/out a: array[1..n] of T)
        var j: nat
        for i:= 2 to n do
            j:= i
            do j > 1 ∧ a[j] < a[j − 1] → swap(a,j,j-1)
                                          j:= j-1
            od
        od
end proc
```

Ejercicio 1
proc D

```
proc D (in/out a: array[1..n] of T)
        for i:= 1 to n-1 do
            for j:= n-1 downto i do
                if a[j] > a[j+1] then swap(a,j,j+1) fi
            od
        od
end proc
```

Ejercicio 1

proc D = bubble sin interrupción

```
proc bubble_sort2 (in/out a: array[1..n] of T)
        for i:= 1 to n-1 do
            for j:= n-1 downto i do
                if a[j] > a[j+1] then swap(a,j,j+1) fi
            od
        od
end proc
```

Ejercicio 1
proc E

```
proc E (in/out a: array[1..n] of T)
    F(a,1,n)
end proc

proc F (in/out a: array[1..n] of T, in u,v: nat)
    var t: nat
    if v > u → G(a,u,v,t)
               F(a,u,t-1)
               F(a,t+1,v)
    fi
end proc

proc G (in/out a: array[1..n] of elem, in u,v: nat, out t: nat)
    var i,j: nat
    t:= u
    i:= u+1
    j:= v
    do i ≤ j → if a[i] ≤ a[t] → i:= i+1
                  a[j] > a[t] → j:= j-1
                  a[i] > a[t] ∧ a[j] ≤ a[t] → swap(a,i,j)
                                              i:= i+1
                                              j:= j-1
               fi
    od
    swap(a,t,j)
    t:= j
end proc
```

Ejercicio 1

proc E = ordenación rápida

```
proc quick_sort (in/out a: array[1..n] of T)
    quick_sort_rec(a,1,n)
end proc

proc quick_sort_rec (in/out a: array[1..n] of T, in izq,der: nat)
    var piv: nat
    if der > izq → pivot(a,izq,der,piv)
                   quick_sort_rec(a,izq,piv-1)
                   quick_sort_rec(a,piv+1,der)
    fi
end proc

proc pivot (in/out a: array[1..n] of elem, in izq,der: nat, out piv: nat)
    var i,j: nat
    piv:= izq
    i:= izq+1
    j:= der
    do i ≤ j → if a[i] ≤ a[piv] → i:= i+1
               a[j] > a[piv] → j:= j-1
               a[i] > a[piv] ∧ a[j] ≤ a[t] → swap(a,i,j)
                                             i:= i+1
                                             j:= j-1
               fi
    od
    swap(a,piv,j)
    piv:= j
end proc
```

Ejercicio 1
Tabla

algoritmos	afirmaciones
A	(1) en el mejor caso es lineal
B	(2) en el peor caso es cuadrático
C	(3) en la práctica es del orden de $n \log n$
D	(4) siempre es del orden de $n \log n$
E	(5) siempre es del orden de n^2

Ejercicio 1
Tabla

algoritmos	afirmaciones
selection_sort	(1) en el mejor caso es lineal
bubble_sort	(2) en el peor caso es cuadrático
insertion_sort	(3) en la práctica es del orden de $n \log n$
bubble_sort2	(4) siempre es del orden de $n \log n$
quick_sort	(5) siempre es del orden de n^2

Ejercicio 1
Tabla

algoritmos	afirmaciones
selection_sort	(2) y (5)
bubble_sort	(1) y (2)
insertion_sort	(1) y (2)
bubble_sort2	(2) y (5)
quick_sort	(2) y (3)

afirmaciones
(1) en el mejor caso es lineal
(2) en el peor caso es cuadrático
(3) en la práctica es del orden de $n \log n$
(4) siempre es del orden de $n \log n$
(5) siempre es del orden de n^2

Ejercicio 2
Inciso a

Dado el siguiente algoritmo, plantear la recurrencia que indica la cantidad de asignaciones realizadas a la variable *m* en función de la entrada *n*:

```
fun f (n: nat) ret m: nat
    if n ≤ 2 then
        m := n
    else
        m := f(n-1) + f(n-2) - f(n-3)
    fi
end
```

Ejercicio 2
Inciso a

$t(n) =$ número de asignaciones a m que realiza una llamada a f(n)

$$t(n) = \begin{cases} 1 & \text{si } n \leq 2 \\ t(n-1) + t(n-2) + t(n-3) + 1 & \text{si } n > 2 \end{cases}$$

Ejercicio 2
Inciso b

Resolver la siguiente recurrencia

$$t(n) = \begin{cases} n & \text{si } n \leq 2 \\ t(n-1) + t(n-2) - t(n-3) & \text{si } n > 2 \end{cases}$$

Ejercicio 2 b

Paso 1: ecuación característica

- Llevar la recurrencia a una **ecuación característica** de la forma

$$a_k t_n + \ldots + a_0 t_{n-k} = 0$$

,

- En el ejemplo, $t(n) = t(n-1) + t(n-2) - t(n-3)$ puede llevarse a $t_n - t_{n-1} - t_{n-2} + t_{n-3} = 0$.
- Entonces, $k = 3$, $a_k = a_3 = 1$, $a_2 = -1$, $a_1 = -1$ y $a_0 = 1$.

Ejercicio 2 b

Paso 2: polinomio característico

- Considerar el **polinomio característico asociado** $a_k x^k + \ldots + a_0$,
- En el ejemplo el polinomio es $x^3 - x^2 - x + 1$.

Ejercicio 2 b

Paso 3: raíces y multiplicidades

- determinar las raíces $r_1, \ldots, r_j$ del polinomio característico, de multiplicidad $m_1, \ldots, m_j$ respectivamente (se tiene $m_i \geq 1$ y $m_1 + \ldots + m_j = k$),
- En el ejemplo, una raíz del polinomio es $r_1 = 1$. Dividendo el polinomio característico por $x - 1$, da el cociente $x^2 - 1$ cuyas raíces son 1 y -1.
- Entonces, $j = 2$, $r_1 = 1$, $r_2 = -1$, $m_1 = 2$ y $m_2 = 1$.

Ejercicio 2 b

Paso 4: forma general de la solución

- considerar la forma general de las soluciones de la ecuación característica:

$$\begin{aligned} t(n) &= c_1 r_1^n + c_2 n r_1^n + \ldots + c_{m_1} n^{m_1-1} r_1^n + \\ &+ c_{m_1+1} r_2^n + c_{m_1+2} n r_2^n + \ldots + c_{m_1+m_2} n^{m_2-1} r_2^n + \\ &\vdots \quad \vdots \quad \vdots \\ &+ c_{m_1+\ldots+m_{j-1}+1} r_j^n + c_{m_1+\ldots+m_{j-1}+2} n r_j^n + \ldots + c_k n^{m_j-1} r_j^n \end{aligned}$$

  como $m_1 + \ldots + m_j = k$, tenemos k incógnitas: $c_1, \ldots, c_k$,
- En el ejemplo, la forma general es

$$t(n) = c_1 r_1 + c_2 n r_1^n + c_3 r_2^n = c_1 + c_2 n + c_3 (-1)^n$$

Ejercicio 2 b

Paso 5: sistema de ecuaciones

- con las k **condiciones iniciales** $t_{n_0}, \ldots, t_{n_0+k-1}$ (n_0 es usualmente 0 ó 1) plantear un sistema de k ecuaciones con k incógnitas:

$$\begin{aligned} t(n_0) &= t_{n_0} \\ t(n_0+1) &= t_{n_0+1} \\ &\vdots \\ t(n_0+k-1) &= t_{n_0+k-1} \end{aligned}$$

- En el ejemplo, $n_0 = 0$ y el sistema es

$$\begin{aligned} c_1 + c_3 &= c_1 + c_2 * 0 + c_3(-1)^0 = t(0) = t_0 = 0 \\ c_1 + c_2 - c_3 &= c_1 + c_2 + c_3(-1) = t(1) = t_1 = 1 \\ c_1 + 2c_2 + c_3 &= c_1 + 2c_2 + c_3(-1)^2 = t(2) = t_2 = 2 \end{aligned}$$

Ejercicio 2 b

Paso 6: cálculo de incógnitas

- obtener de este sistema los valores de las k incógnitas $c_1, \ldots, c_k$,
- En el ejemplo, de la primera ecuación, se obtiene $c_1 = -c_3$, reemplazando en la segunda:

$$1 = -c_3 + c_2 - c_3 = c_2 - 2c_3$$

Entonces $c_2 = 1 + 2c_3$.

- Reemplazando en la tercera:

$$2 = -c_3 + 2(1 + 2c_3) + c_3 = 2 + 4c_3$$

Entonces $c_3 = 0$, $c_2 = 1$ y $c_1 = 0$.

Ejercicio 2 b

Paso 7: solución final

- escribir la **solución final** de la forma $t_n = t'(n)$, donde $t'(n)$ se obtiene a partir de $t(n)$ reemplazando c_i y r_i por sus valores y simplificando la expresión final.
- En el ejemplo,

$$\begin{aligned} t_n &= t(n) \\ &= c_1 + c_2 * n + c_3(-1)^n \\ &= n \end{aligned}$$

Ejercicio 2 b

Paso 8: comprobación

- La **solución final** obtenida puede demostrarse por inducción. Más sencillo que eso es corroborar que $t_{n_o+k} = t'(n_0 + k)$, donde $n_0 + k$ es un **valor nuevo, no utilizado en el sistema de ecuaciones** anterior
- En el ejemplo,

$$\begin{aligned} t_3 &= t_2 + t_1 - t_0 = 2 + 1 - 0 = 3 \\ t'(3) &= 3 \end{aligned}$$

Ejercicio 2 b

Paso 9: orden

- Se concluye que $t'(n)$ es la solución de la recurrencia. Si el objetivo era calcular el **orden** ya se pueden utilizar las propiedades conocidas.
- En el ejemplo, $t'(n) = n \in \Theta(n)$.

Ejercicio 3

Ordenar las siguientes funciones según el orden creciente de sus $\mathcal{O}$.

- $\log_2(n^n)$
- $n^{\log_n n}$
- $n^{1.001}$
- $n^{0.001}$
- 1.001^n

Ejercicio 3

Ordenar las siguientes funciones según el orden creciente de sus $\mathcal{O}$.

- $\log_2(n^n) = n \log_2 n \in \Theta(n \log n)$
- $n^{\log_n n} = n^1 = n$, polinomial con exponente 1.
- $n^{1.001}$ polinomial con exponente apenas mayor que 1.
- $n^{0.001}$ polinomial con exponente apenas mayor que 0.
- 1.001^n exponencial.

$$\mathcal{O}(n^{0.001}) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log n) \subset \mathcal{O}(n^{1.001}) \subset \mathcal{O}(1.001^n)$$

$$\mathcal{O}(n^{0.001}) \subset \mathcal{O}(n^{\log_n n}) \subset \mathcal{O}(\log_2(n^n)) \subset \mathcal{O}(n^{1.001}) \subset \mathcal{O}(1.001^n)$$

Ejercicio 4

TAD listado

constructores

vacío : listado

agrega-char : listado $\times$ char $\rightarrow$ listado

operaciones

cuenta : listado $\times$ char $\rightarrow$ nat

reducir : listado $\rightarrow$ listado

cuenta-palabras : listado $\rightarrow$ nat

quitar-última : listado $\rightarrow$ listado

Ejercicio 4

ecuaciones

cuenta(vacío,c) = 0

c = c' $\Longrightarrow$ cuenta(agrega-char(l,c), c') = cuenta(l,c') + 1

c $\neq$ c' $\Longrightarrow$ cuenta(agrega-char(l,c), c') = cuenta(l,c')

reducir(vacío) = vacío

c = ' ' $\Longrightarrow$ reducir(agrega-char(l,c)) = reducir(l)

c $\neq$ ' ' $\Longrightarrow$ reducir(agrega-char(vacío,c)) = agrega-char(vacío,c)

c $\neq$ ' ' $\wedge$ c' $\neq$ ' ' $\Longrightarrow$ reducir(agrega-char(agrega-char(l,c),c')) = agrega-char(reducir(agrega-char(l,c)),c')

c = ' ' $\wedge$ c' $\neq$ ' ' $\wedge$ reducir(l) = vacío $\Longrightarrow$ reducir(agrega-char(agrega-char(l,c),c')) = agrega-char(vacío,c')

c = ' ' $\wedge$ c' $\neq$ ' ' $\wedge$ reducir(l) $\neq$ vacío $\Longrightarrow$
reducir(agrega-char(agrega-char(l,c),c')) = agrega-char(agrega-char(reducir(l),c),c')

reducir(l) = vacío $\Longrightarrow$ cuenta-palabras(l) = 0

reducir(l) $\neq$ vacío $\Longrightarrow$ cuenta-palabras(l) = cuenta(reducir(l),' ') + 1

c $\neq$ ' ' $\Longrightarrow$ quitar-última(agrega-char(vacío,c)) = vacío

c = ' ' $\Longrightarrow$ quitar-última(agrega-char(l,c)) = quitar-última(l)

c = ' ' $\wedge$ c' $\neq$ ' ' $\Longrightarrow$ quitar-última(agrega-char(agrega-char(l,c),c')) = agrega-char(l,c)

c $\neq$ ' ' $\wedge$ c' $\neq$ ' ' $\Longrightarrow$ quitar-última(agregar-char(agrega-char(l,c),c')) = quitar-última(agrega-char(l,c))

Ejercicio 5

Implementar el siguiente TAD usando un **nat**:

TAD bin
constructores
uno : bin
$\_ \lhd_0 : \text{bin} \to \text{bin}$
$\_ \lhd_1 : \text{bin} \to \text{bin}$
operaciones
es_uno : bin $\to$ booleano
es_par : bin $\to$ booleano
mover : bin $\to$ bin $\quad \{$ pre: argumento $\neq$ uno $\}$
suc : bin $\to$ bin
sum : bin $\times$ bin $\to$ bin

Ejercicio 5

ecuaciones

es_uno(uno) = verdadero
es_uno(b $\lhd_0$) = falso
es_uno(b $\lhd_1$) = falso
es_par(uno) = falso
es_par(b $\lhd_0$) = verdadero
es_par(b $\lhd_1$) = falso
mover(b $\lhd_0$) = b
mover(b $\lhd_1$) = b
suc(uno) = uno $\lhd_0$
suc(b $\lhd_0$) = b $\lhd_1$
suc(b $\lhd_1$) = suc(b) $\lhd_0$

Ejercicio 5

sum(uno,c) = suc(c)
sum(b,uno) = suc(b)
sum(b $\lhd_0$,c $\lhd_0$) = sum(b,c) $\lhd_0$
sum(b $\lhd_0$,c $\lhd_1$) = sum(b,c) $\lhd_1$
sum(b $\lhd_1$,c $\lhd_0$) = sum(b,c) $\lhd_1$
sum(b $\lhd_1$,c $\lhd_1$) = suc(sum(b,c)) $\lhd_0$

Ejercicio 5
Implementación

type bin = **nat**

fun one() **ret** b:bin
    b:= 1
end fun
{Post: b $\sim$ one}

Ejercicio 5
Implementación

$\{$Pre: b $\sim$ B$\}$
fun right0(b:bin) **ret** c:bin
    c:= 2*b
end fun
$\{$Post: c $\sim$ B $\lhd_0\}$

$\{$Pre: b $\sim$ B$\}$
fun right1(b:bin) **ret** c:bin
    c:= 2*b+1
end fun
$\{$Post: c $\sim$ B $\lhd_1\}$

Ejercicio 5
Implementación

{Pre: b $\sim$ B}
fun is_one(b:bin) **ret** c:**bool**
    c:= (b = 1)
end fun
{Post: c $\sim$ (B = uno) }

{Pre: b $\sim$ B}
fun is_even(b:bin) **ret** c:**bool**
    c:= (b mod 2 = 0)
end fun
{Post: c $\sim$ es_par(B) }

Ejercicio 5
Implementación

$\{$Pre: b $\sim$ B$\}$
fun shift(b:bin) **ret** c:bin
    c:= b $\div$ 2
end fun
$\{$Post: c $\sim$ mover(B) $\}$

$\{$Pre: b $\sim$ B $\wedge$ c $\sim$ C$\}$
fun add(b,c:bin) **ret** d:bin
    d:= b + c
end fun
$\{$Post: d $\sim$ sum(B,C) $\}$

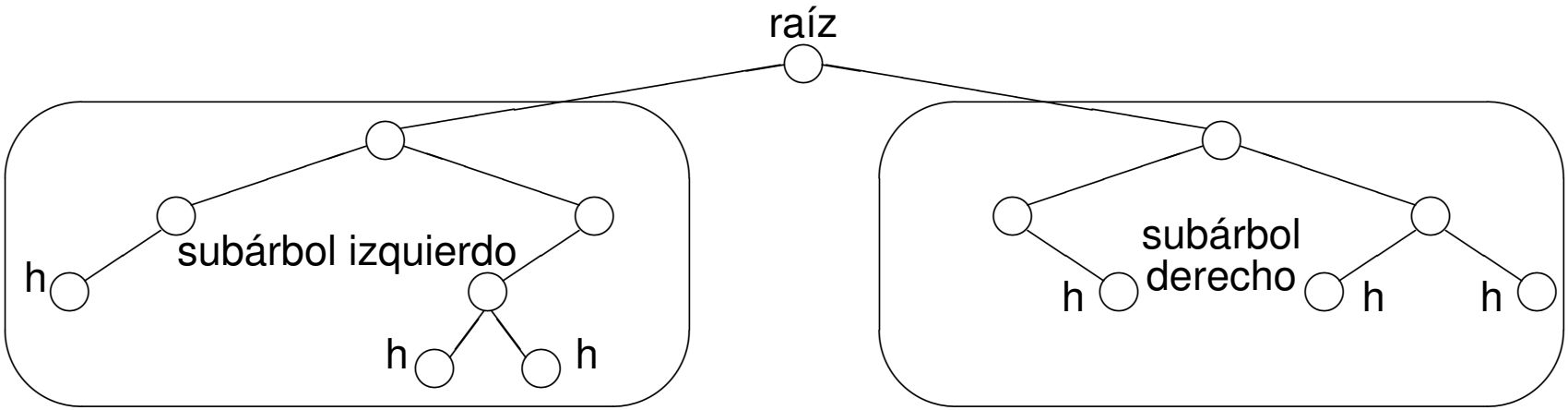
Ejercicio 5
Implementación

{Pre: b $\sim$ B}
fun succ(b:bin) **ret** c:bin
    c:= b + 1
end fun
{Post: c $\sim$ suc(B) }

Repaso

- cómo vs. qué
- 3 partes
  1. análisis de algoritmos
     - algoritmos de ordenación
     - notación $\mathcal{O}$, Ω y Θ.
     - propiedades y jerarquía
     - recurrencias (D. y V., homogéneas y no homogéneas)
  2. tipos de datos
     - tipos concretos (arreglos, listas, tuplas, punteros)
     - tipos abstractos (TAD contador, TAD pila, TAD cola, TAD pcola)
     - implementaciones elementales
     - implementaciones utilizando listas enlazadas
     - árboles binarios
  3. técnicas de resolución de problemas

Intuición



Todos los árboles pueden construirse con los constructores

- $<>$, que construye un árbol vacío
- $< \_, \_, \_ >$, que construye un árbol no vacío a partir de un elemento y dos subárboles

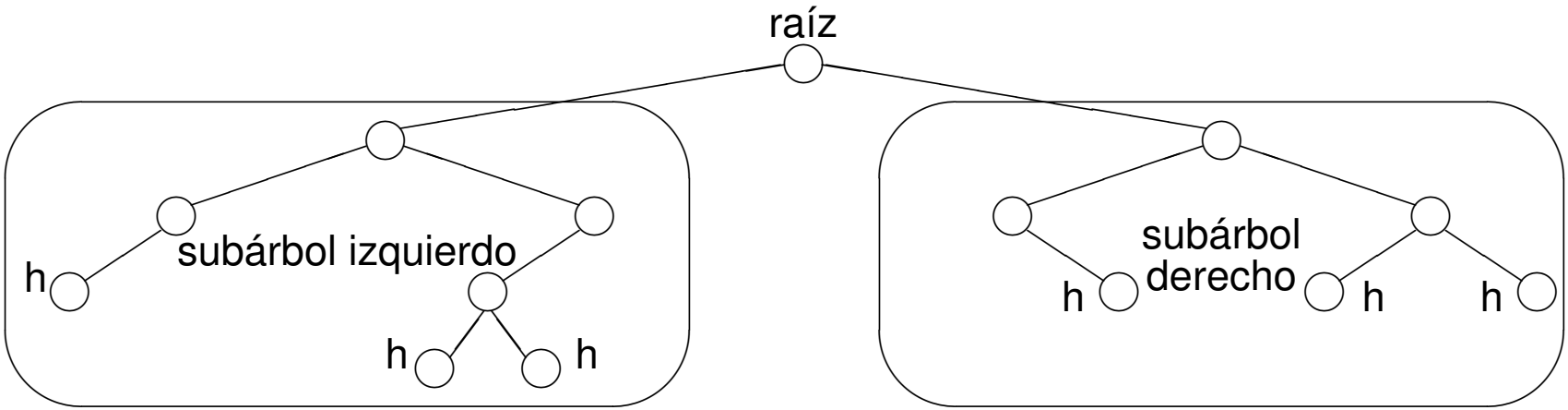
Especificación

TAD árbol_binario[elem]
constructores
$<>$: árbol_binario
$< \_, \_, \_ >$: árbol_binario $\times$ elem $\times$ árbol_binario $\rightarrow$ árbol_binario
operaciones
raíz : árbol_binario $\rightarrow$ elem {se aplica sólo a un árbol no vacío}
izquierdo : árbol_binario $\rightarrow$ árbol_binario {sólo a un árbol no vacío}
derecho : árbol_binario $\rightarrow$ árbol_binario {sólo a un árbol no vacío}
es_vacío : árbol_binario $\rightarrow$ booleano
ecuaciones
raíz(<i,r,d>) = r
izquierdo(<i,r,d>) = i
derecho(<i,r,d>) = d
es_vacío(<>) = verdadero
es_vacío(<i,r,d>) = falso

Notación $<>$

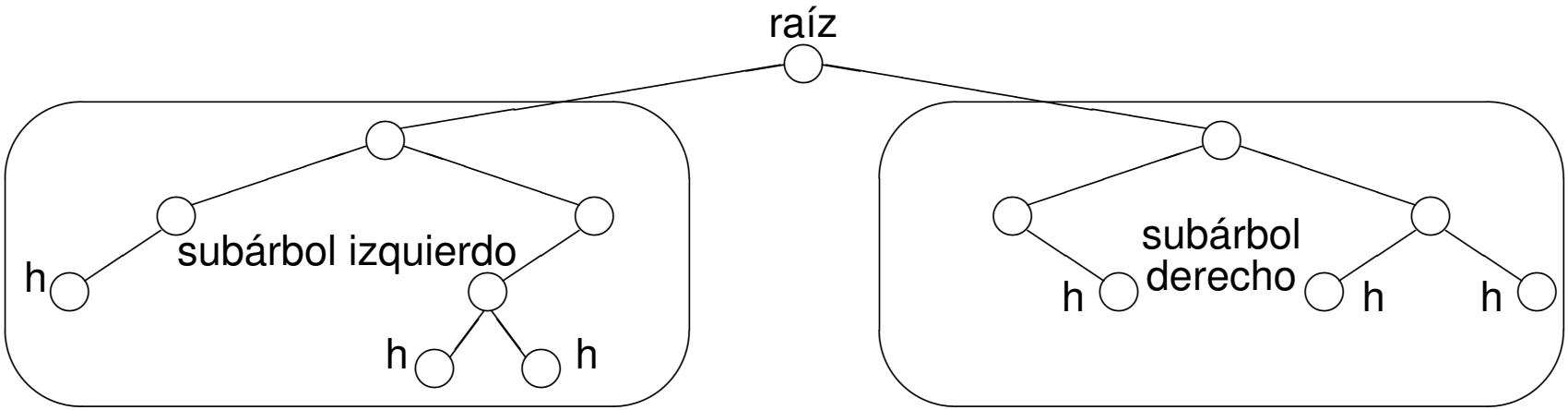
- Notar la sobrecarga de la notación $<>$:
  - $<>$ es el árbol vacío,
  - $< i, r, d >$ es el árbol no vacío cuya raíz es r, subárbol izquierdo es i y subárbol derecho es d.
  - $< r >$ es la hoja $<<>, r, <>>$
- Conclusión: la notación $<>$ puede tener 0, 1 ó 3 argumentos.

Botánica y genealogía



- Un **nodo** es un árbol no vacío.
- Tiene **raíz**, **subárbol izquierdo** y **subárbol derecho**.
- A los subárboles se los llama también **hijos** (izquierdo y derecho).
- Y al nodo se le dice **padre** de sus hijos.
- Una **hoja** es un nodo con los dos hijos vacíos.

Más terminología



Terminología:

- Se usa terminología genealógica como **hijo, padre, nieto, abuelo, hermanos, ancestro, descendiente**.
- También de la botánica: **raíz, hoja**.
- Se define **camino, altura, profunidad, nivel**.

Sobre los niveles

- En el nivel 0 hay a lo sumo 1 nodo.
- En el nivel 1 hay a lo sumo 2 nodos.
- En el nivel 2 hay a lo sumo 4 nodos.
- En el nivel 3 hay a lo sumo 8 nodos.
- En el nivel i hay a lo sumo 2^i nodos.
- En un árbol de altura n hay a lo sumo $2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$ nodos.
- En un árbol "balanceado" la altura es del orden del $\log_2 k$ donde k es el número de nodos.

Implementación con punteros

```
type node = tuple
                lft: pointer to node
                value: elem
                rgt: pointer to node
            end
type bintree = pointer to node

fun empty() ret t:bintree
    t:= null
end
{Post: t ∼<> }
```

Implementación con punteros

{Pre: l $\sim$ L $\wedge$ e $\sim$ E $\wedge$ r $\sim$ R}
fun node(l:bintree,e:elem,r:bintree) **ret** t:bintree
    alloc(t)
    t$\rightarrow$lft:= l
    t$\rightarrow$value:= e
    t$\rightarrow$rgt:= r
{Post: t $\sim$ <L,E,R>} **end**

{Pre: t $\sim$ T $\wedge\neg$ is_empty(t)}
fun root(t:bintree) **ret** e:elem
    e:= t$\rightarrow$value
end
{Post: e $\sim$ raíz(T)}

Implementación con punteros

{Pre: t $\sim$ T $\wedge\neg$ is_empty(t)}
fun left(t:bintree) **ret** l:bintree
    l:= t$\rightarrow$lft
end
{Post: l $\sim$ izquierdo(T)}

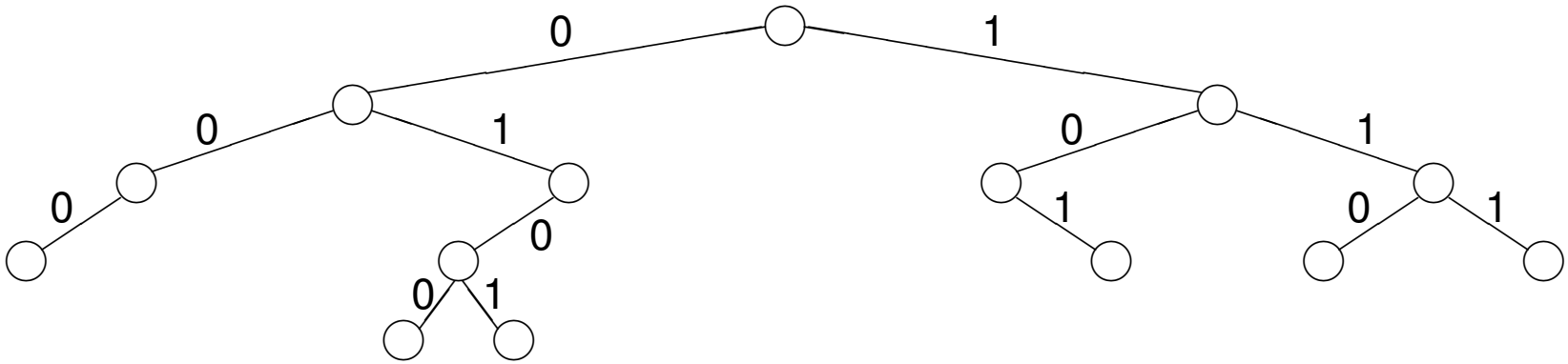
{Pre: t $\sim$ T $\wedge\neg$ is_empty(t)}
fun right(t:bintree) **ret** r:bintree
    r:= t$\rightarrow$rgt
end
{Post: r $\sim$ derecho(T)}

Implementación con punteros

{Pre: t $\sim$ T}
fun is_empty(t:bintree) **ret** b:**bool**
    b:= (t = **null**)
end
{Post: b $\sim$ es_vacío(T)}

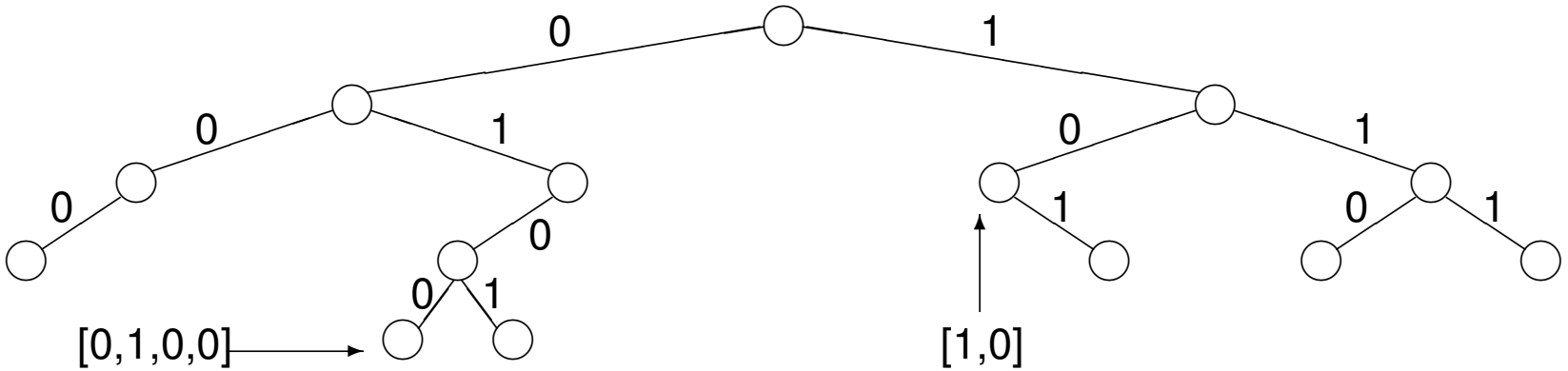
proc destroy(**in**/**out** t:bintree)
    if $\neg$ is_empty(t) **then** destroy(left(t))
                destroy(right(t))
                free(t)
                t:= **null**
    fi
end

Indicaciones



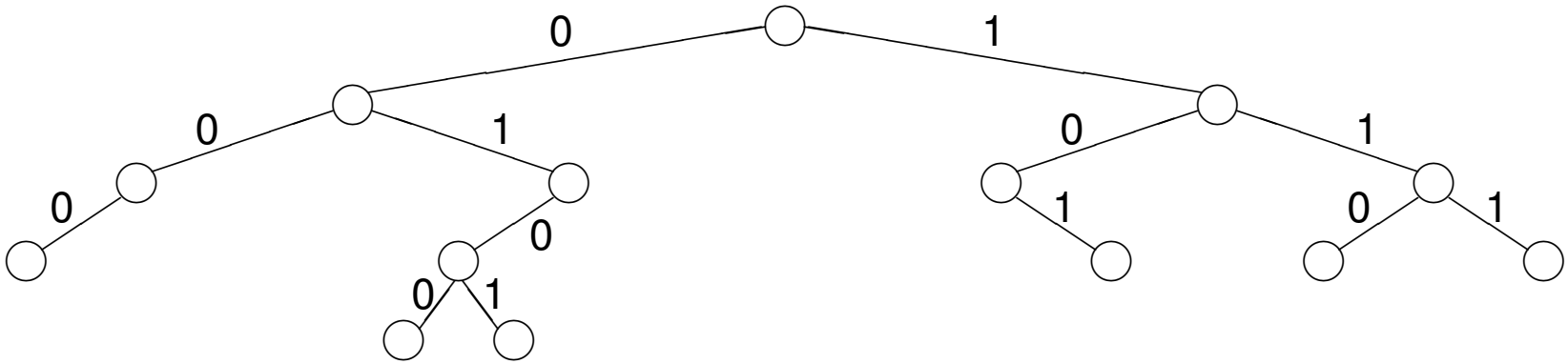
- A cada arista que conecta un padre con su hijo se la rotula 0 si es con el hijo izquierdo y 1 si es el derecho,
- Este 0 ó 1 puede entenderse como dando **indicaciones**
- 0 es ir a la izquierda
- 1 es ir a la derecha

Posiciones



- Una lista de 0's y 1's sirve para desplazarse desde la raíz hacia las hojas.
- Cada subárbol queda señalado por una lista de 0's y 1's.
- Estas listas de 0's y 1's marcan **posiciones** dentro del árbol.
- Definimos $pos = [\{0, 1\}]$.
- Es el conjunto de todas las posiciones.

Selección de subárbol

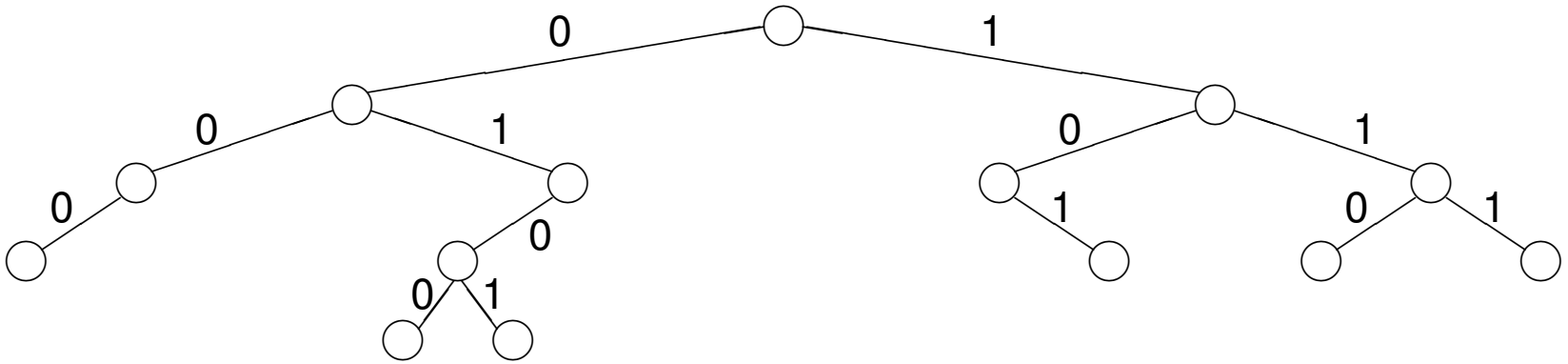


Dado un árbol t y una posición $p \in pos$, $t \downarrow p$ es el subárbol de t que se encuentra en la posición p:

$<>\downarrow p = <>$
$< i, e, d >\downarrow [] = < i, e, d >$
$< i, e, d >\downarrow (0 \triangleright p) = i \downarrow p$
$< i, e, d >\downarrow (1 \triangleright p) = d \downarrow p$

Se define $pos(t) = \{p \in pos \mid t \downarrow p \neq <>\}$. Es el conjunto de las posiciones del árbol binario t.

Selección de elemento



Dado un árbol t y una posición $p \in pos(t)$, $t.p$ es el elemento de t que se encuentra en la posición p:

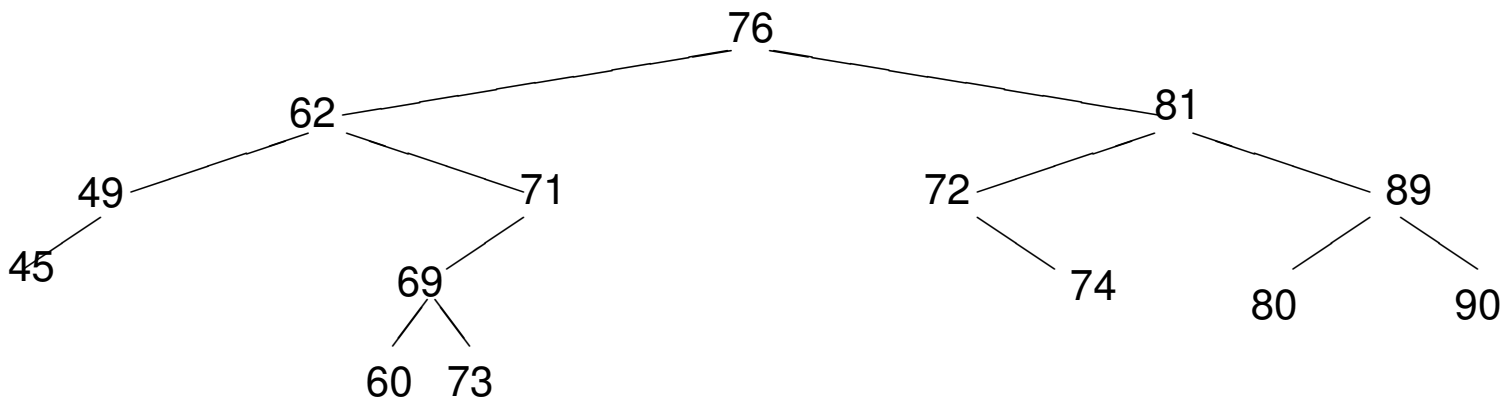
$< i, e, d > .[] = e$
$< i, e, d > .(0 \triangleright p) = i.p$
$< i, e, d > .(1 \triangleright p) = d.p$

o equivalentemente $t.p = raiz(t \downarrow p)$.

Árboles binarios de búsqueda

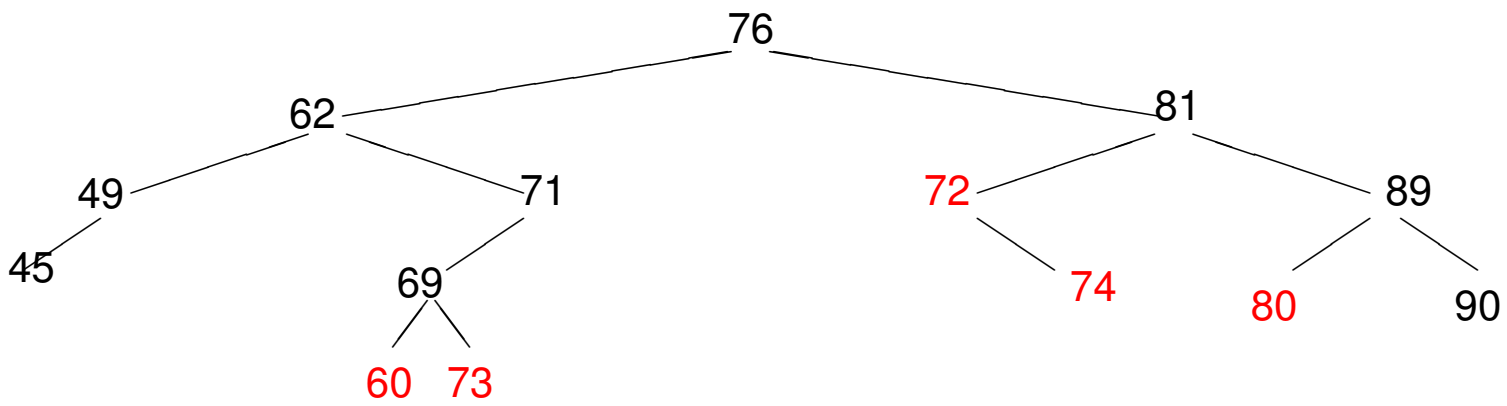
- Son casos particulares de árboles binarios,
- son árboles binarios t en donde la información está organizada de tal forma de que un algoritmo sencillo permite buscar eficientemente un elemento:
- el elemento buscado se compara con la raíz de t
  - si es el mismo, la búsqueda finaliza
  - si es menor que la raíz, la búsqueda se restringe al subárbol izquierdo de t con el mismo algoritmo
  - si es mayor que la raíz, la búsqueda se restringe al subárbol derecho de t con el mismo argumento.
- Si el árbol está “balanceado”, es un algoritmo logarítmico.

Ejemplo



¿Es un árbol binario de búsqueda?

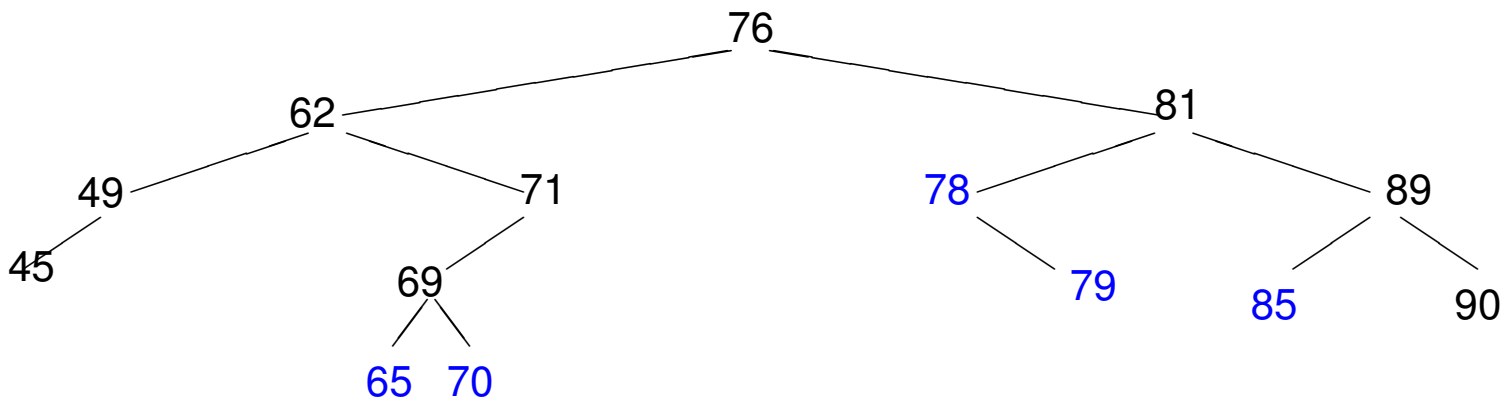
Ejemplo



No es un árbol binario de búsqueda.

- 60 debe cambiar por uno entre 63 y 68
- 72 debe cambiar por uno entre 77 y 80
- 73 debe cambiar por 70
- 74 debe cambiar por uno entre 77 y 80.
- 80 debe cambiar por uno entre 82 y 88.

Ejemplo



Ahora sí es un árbol binario de búsqueda.

Definición intuitiva

Para que este algoritmo funcione, t debe cumplir lo siguiente:

- los valores alojados en el subárbol izquierdo de t deben ser menores que el alojado en la raíz de t,
- los valores alojados en el subárbol derecho de t deben ser mayores que el algojado en la raíz de t,
- estas dos condiciones deben darse para todos los subárboles de t.

Si se cumplen estas condiciones, decimos que t es un **árbol binario de búsqueda** o **ABB**.

Entendiendo la definición

Otra forma de decirlo:

- los valores alojados en el subárbol izquierdo de t deben ser menores que el alojado en la raíz de t,
- los valores alojados en el subárbol derecho de t deben ser mayores que el algojado en la raíz de t,
- estas dos condiciones deben darse para el subárbol $t \downarrow p$ para todo $p \in pos(t)$.

Formalizando la definición

- Para todo $p \in pos(t)$,
  - los valores alojados en el subárbol izquierdo de $t \downarrow p$ deben ser menores que $t.p$
  - los valores alojados en el subárbol derecho de $t \downarrow p$ deben ser mayores que $t.p$

Formalizando la definición

- Para todo $p \in pos(t)$,
  - los valores del árbol de la forma $t.(p \lhd 0 +\!\!+ q)$ deben ser menores que $t.p$
  - los valores del árbol de la forma $t.(p \lhd 1 +\!\!+ q)$ deben ser mayores que $t.p$
- habría que aclarar que siempre y cuando $p \lhd 0 +\!\!+ q$ y $p \lhd 1 +\!\!+ q$ no se vayan fuera del árbol.

Definición formal

- Para todo $p \in pos(t)$ y para todo $q \in pos$
  - si $p \lhd 0 +\!\!+ q \in pos(t)$ entonces $t.(p \lhd 0 +\!\!+ q) < t.p$
  - si $p \lhd 1 +\!\!+ q \in pos(t)$ entonces $t.(p \lhd 1 +\!\!+ q) > t.p$
- O como lo escribimos en los apuntes: $ABB(t)$ sii $\forall p \in \text{pos}(t).\forall q \in \text{pos}$

$$\begin{cases} (p \lhd 0) +\!\!+ q \in \text{pos}(t) \Rightarrow t.((p \lhd 0) +\!\!+ q) < t.p \\ (p \lhd 1) +\!\!+ q \in \text{pos}(t) \Rightarrow t.p < t.((p \lhd 1) +\!\!+ q) \end{cases}$$

TAD diccionario
Especificación

TAD diccionario[elem]
constructores
vacío : diccionario
agregar : elem $\times$ diccionario $\rightarrow$ diccionario
operaciones
es_vacío : diccionario $\rightarrow$ booleano
está : elem $\times$ diccionario $\rightarrow$ booleano
borrar : elem $\times$ diccionario $\rightarrow$ diccionario

TAD diccionario
Especificación

ecuaciones

es_vacío(vacío) = verdadero
es_vacío(agregar(e,d)) = falso
está(e,vacío) = falso
está(e,agregar(e,d)) = verdadero
e $\neq$ e' $\Rightarrow$ está(e,agregar(e',d)) = está(e,d)
borrar(e,vacío) = vacío
borrar(e,agregar(e,d)) = borrar(e,d)
e $\neq$ e' $\Rightarrow$ borrar(e,agregar(e',d)) = agregar(e',borrar(e,d))

TAD diccionario
Implementación usando ABBs

type dictionary = <T>

empty : dictionary
empty = <>

is_empty : dictionary $\rightarrow$ Bool
is_empty <> = true
is_empty <l,e,r> = false

TAD diccionario

Implementación usando ABBs

search : T $\times$ dictionary $\rightarrow$Bool {se aplica a un ABB}
search(e,<>) = false
search(e,<l,e',r>) = **if** e<e' $\rightarrow$ search(e,l)
e=e' $\rightarrow$ true
e>e' $\rightarrow$ search(e,r)
fi

TAD diccionario

Implementación usando ABBs

insert : T $\times$ dictionary $\rightarrow$ dictionary {se aplica a un ABB}

insert(e,<>) = <e>

insert(e,<l,e',r>) = **if** e<e' $\rightarrow$ <insert(e,l),e',r>
e=e' $\rightarrow$ <l,e',r>
e>e' $\rightarrow$ <l,e',insert(e,r)>
fi

TAD diccionario
Implementación usando ABBs

max : dictionary $\rightarrow$ T {se aplica a un ABB no vacío}
max(<l,e,r>) = **if** r=<> $\rightarrow$ e
r$\neq$<> $\rightarrow$ max(r)
fi

delete_max : dictionary $\rightarrow$ dictionary {se aplica a un ABB no vacío}
delete_max(<l,e,r>) = **if** r=<> $\rightarrow$ l
r$\neq$<> $\rightarrow$ <l,e,delete_max(r)>
fi

TAD diccionario

Implementación usando ABBs

delete : T $\times$ dictionary $\rightarrow$ dictionary {se aplica a un ABB}

delete(e,<>) = <>

delete(e,<l,e',r>) = **if** e<e' $\rightarrow$ <delete(e,l),e',r>

e=e' $\wedge$ l =<> $\rightarrow$ r

e=e' $\wedge$ l $\neq$<> $\rightarrow$ <delete_max(l),max(l),r>

e>e' $\rightarrow$ <l,e',delete(e,r)>

fi

Charla sobre software libre

Hoy a las 16 horas en el auditorio.