Algoritmos y Estructuras de Datos II

Árboles binarios de búsqueda

29 de abril de 2015

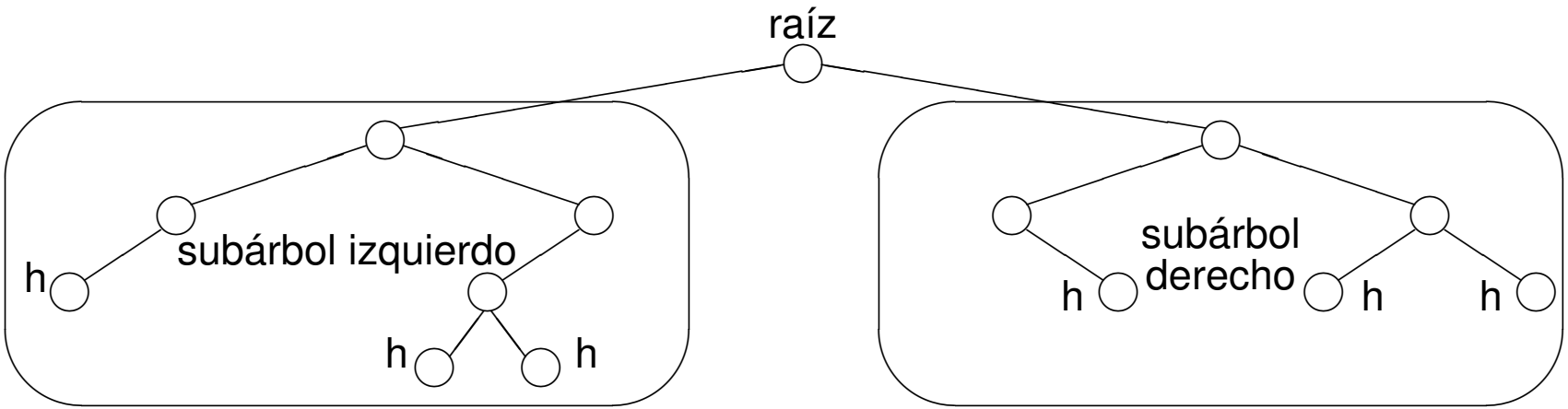
Clase de hoy

1. Repaso
   - Árboles binarios
   - Especificación
   - Terminología habitual
   - Implementación con punteros
   - Posiciones
2. Árbol binario de búsqueda
   - Ejemplos y definiciones
   - TAD conjunto finito

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

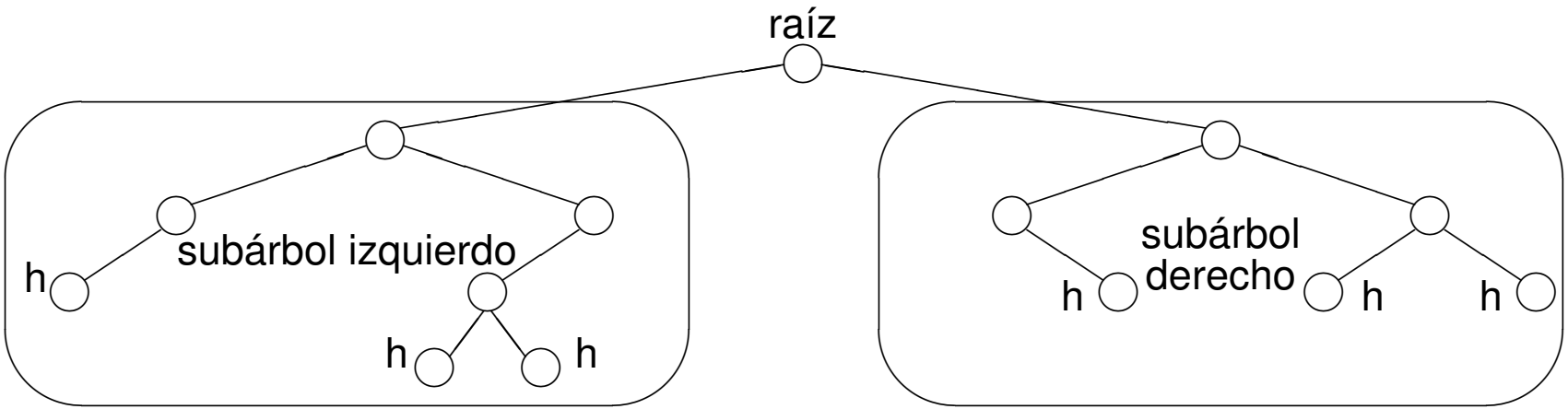
Especificación del TAD árbol binario

```
module TADÁrbolBinario where

data ÁrbolBinario e = <>
                    | < _, _, _ > (ÁrbolBinario e) e (ÁrbolBinario e)

es_vacío :: ÁrbolBinario e → Bool
raíz :: ÁrbolBinario e → e
izquierdo :: ÁrbolBinario e → ÁrbolBinario e
derecho :: ÁrbolBinario e → ÁrbolBinario e
- - las tres últimas se aplican sólo a árbol no vacío

es_vacía <> = True
es_vacía <i,r,d> = False

raíz <i,r,d> = r
izquierdo <i,r,d> = i
derecho <i,r,d> = d
```

Notación <>

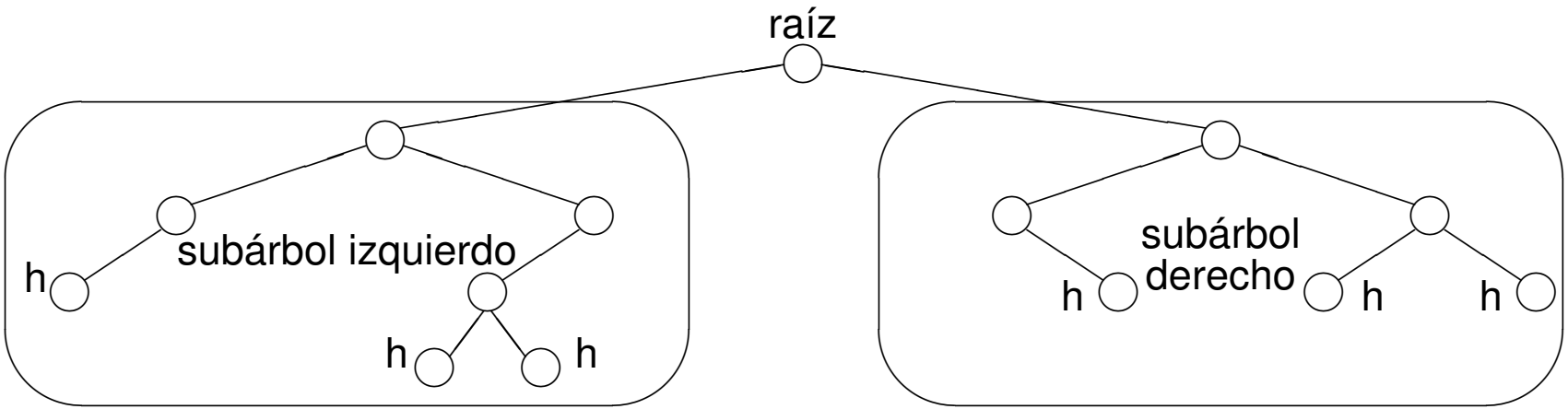
- Notar la sobrecarga de la notación <>:
  - <> es el árbol vacío,
  - <i,r,d> es el árbol no vacío cuya raíz es r, subárbol izquierdo es i y subárbol derecho es d.
  - <r> es la hoja <<>,r,<>>
- Conclusión: la notación <> puede tener 0, 1 ó 3 argumentos.

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, profundidad, 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}

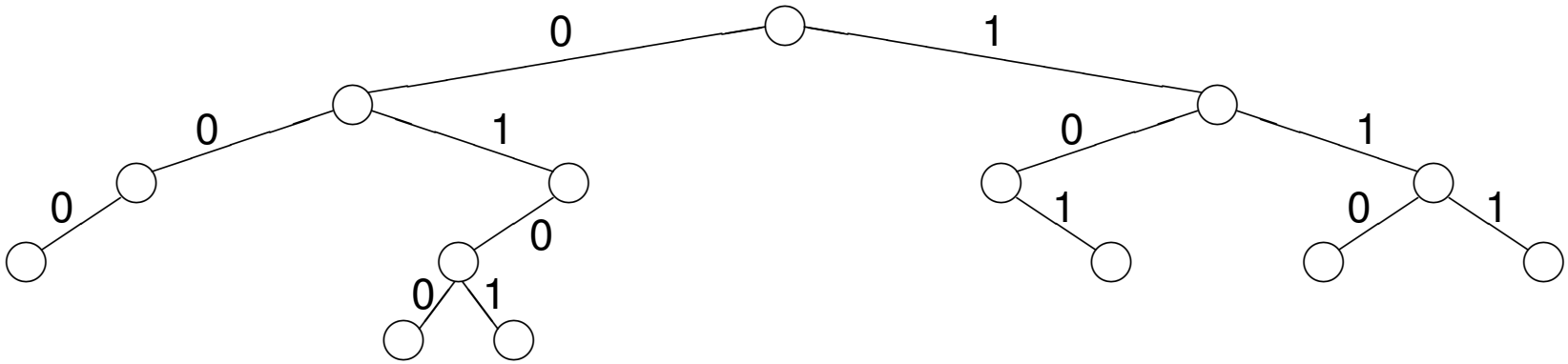
{Pre: t $\sim$ T $\wedge \neg$ is_empty(t)}
fun right(t:bintree) **ret** r:bintree
    r:= t$\rightarrow$rgt
end
{Post: r $\sim$ derecho T}

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}

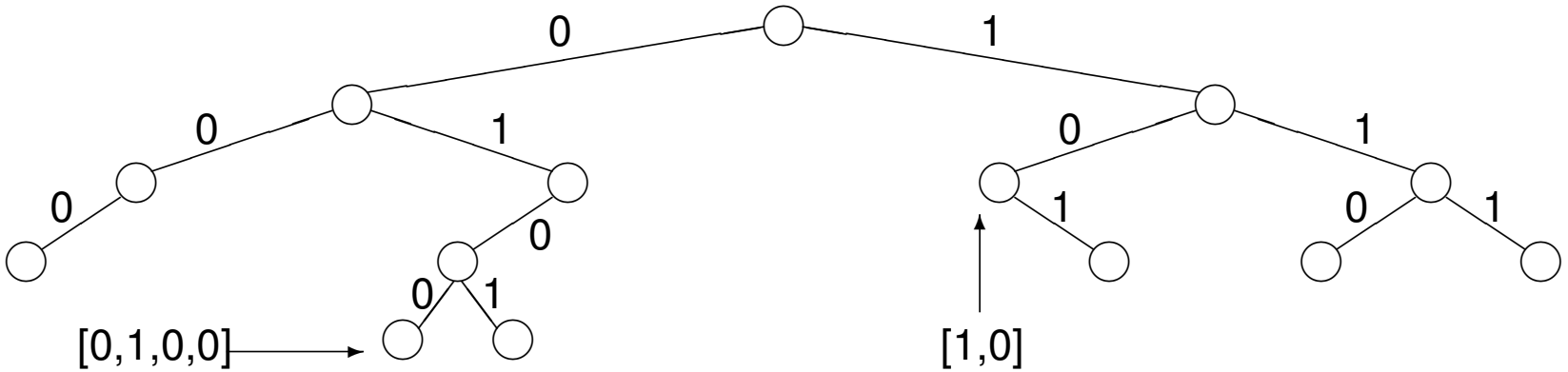
proc destroy(**in**/**out** t:bintree)
    if $\neg$ is_empty(t) **then** destroy(left(t))
                destroy(right(t))
                free(t)
                t:= **null**
    fi
end

Indicaciones



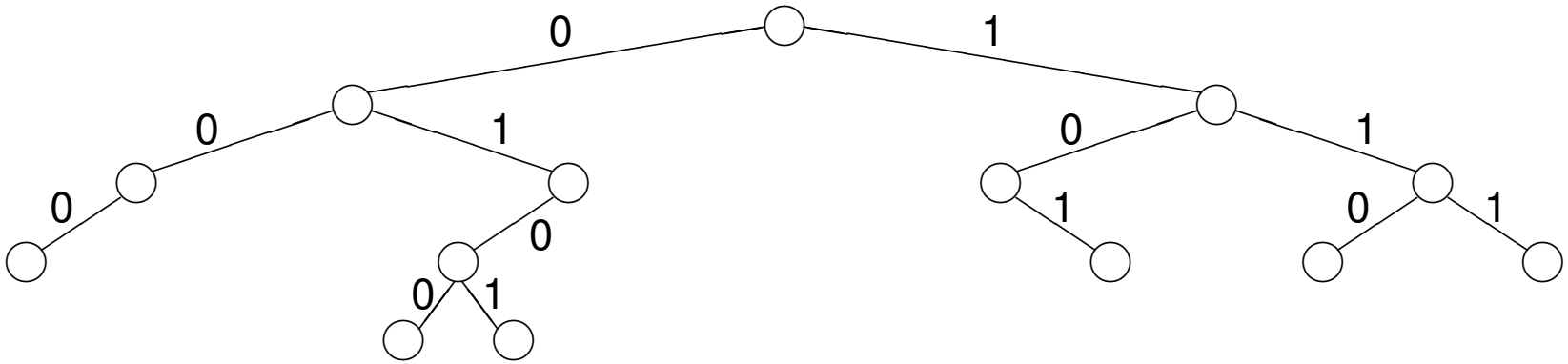
- A cada arista que conecta un padre con su hijo se la rotula 0 si es con el hijo izquierdo y 1 si es el derecho,
- Este 0 ó 1 puede entenderse como dando **indicaciones**
- 0 es ir a la izquierda
- 1 es ir a la derecha

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

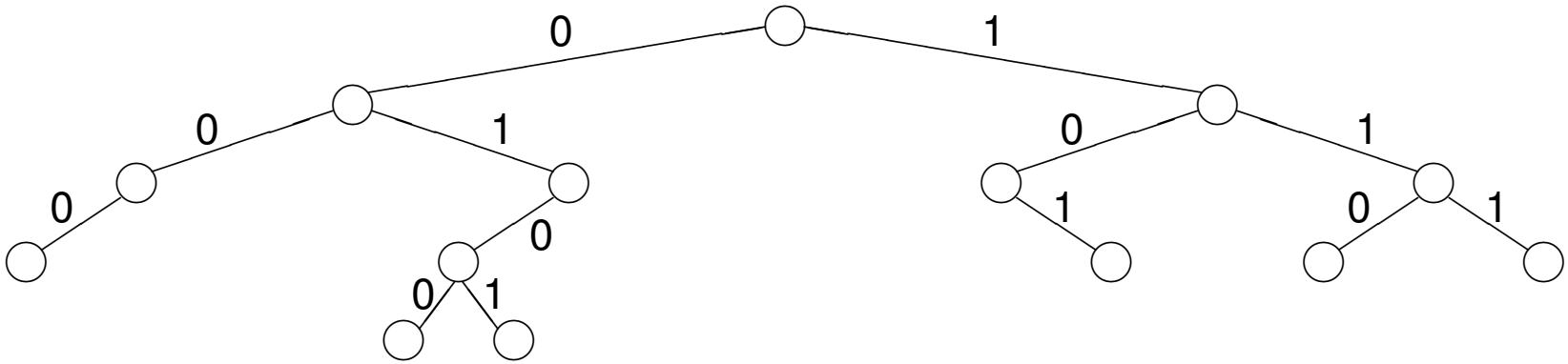


Dado un árbol t y una posición $p \in pos$, $t \downarrow p$ es el subárbol de t que se encuentra en la posición p:

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

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:

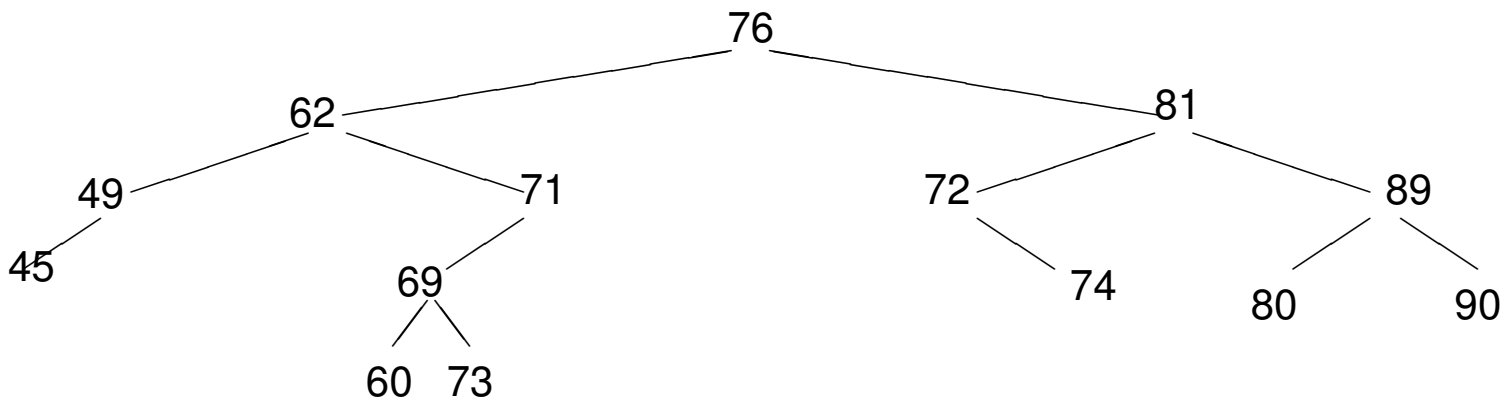
$< i, e, d > .[] = e$
$< i, e, d > .(0 \triangleright p) = i.p$
$< i, e, d > .(1 \triangleright p) = d.p$

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

Árboles binarios de búsqueda

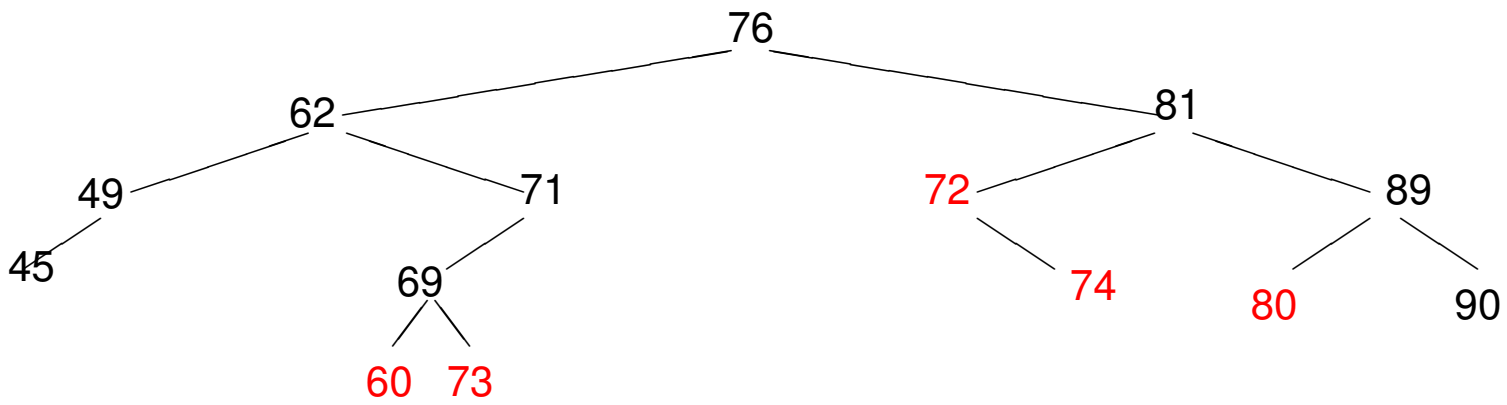
- Son casos particulares de árboles binarios,
- son árboles binarios t en donde la información está organizada de tal forma de que un algoritmo sencillo permite buscar eficientemente un elemento:
- el elemento buscado se compara con la raíz de t
  - si es el mismo, la búsqueda finaliza
  - si es menor 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?

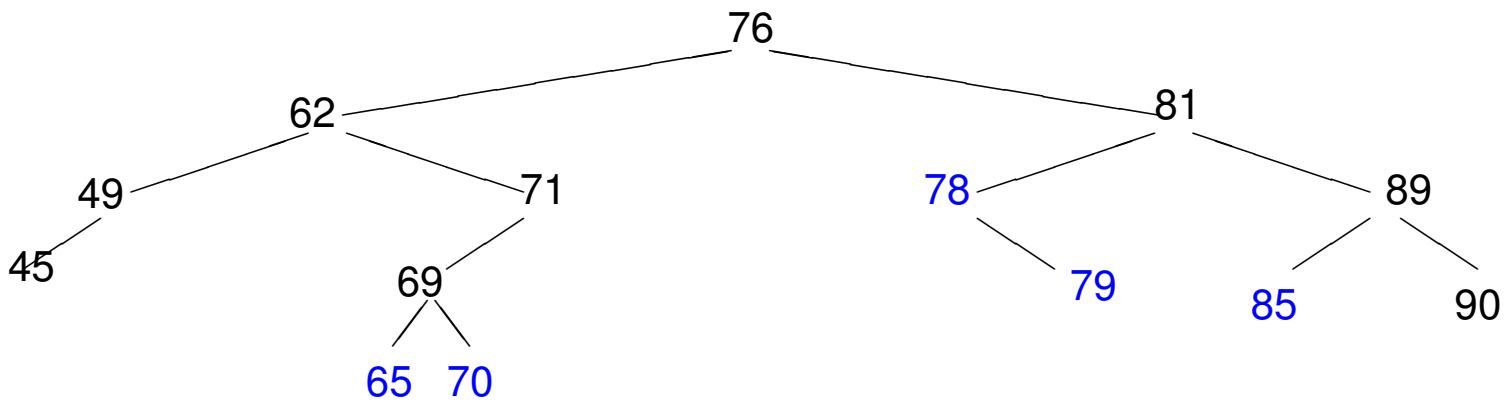
Ejemplo



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 alojado 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**.

Definición en Haskell

Ver en Haskell.

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 alojado 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 conjunto finito
Especificación

```
module TADCjtoFinito where

data Eq e ⇒ CjtoFinito e = Vacío
                         | Agregar e (CjtoFinito e)

es_vacío :: Eq e ⇒ CjtoFinito e → Bool
está :: Eq e ⇒ e → CjtoFinito e → Bool
borrar :: Eq e ⇒ e → CjtoFinito e → CjtoFinito e

es_vacío Vacío = True
es_vacío (Agregar e c) = False

está e c = ?
borrar e c = ?
```

TAD conjunto finito
Especificación

```
module TADCjtoFinito where

...

está e Vacío = False
está e (Agregar e' c) | e == e' = True
                      | otherwise = está e c

borrar e c = ?
```

TAD conjunto finito
Especificación

```
module TADCjtoFinito where

...

está e Vacío = False
está e (Agregar e' c) | e == e' = True
                      | otherwise = está e c

borrar e Vacío = Vacío
borrar e (Agregar e' c) | e == e' = borrar e c
                        | otherwise = Agregar e' (borrar e c)
```

TAD conjunto finito
Implementación usando ABBs

type set = <T>

proc empty(**out** s:set)
    s:= < >
end proc
{Post: s $\sim$ Vacío}

fun is_empty(s:set) **ret** b:**Bool**
    b:= (s = < >)
end fun
{Post: b = (s $\sim$ Vacío)}

TAD conjunto finito
Implementación usando ABBs

{Pre: e $\sim$ E $\wedge$ s $\sim$ C $\wedge$ abb s}
fun search(e:T,s:set) **ret** b:**Bool**
    if is_empty(s) $\rightarrow$ b:= False
        $\neg$ es_empty(s) $\wedge$ e < root(s) $\rightarrow$ b:= search(e,left(s))
        $\neg$ es_empty(s) $\wedge$ e = root(s) $\rightarrow$ b:= True
        $\neg$ es_empty(s) $\wedge$ e > root(s) $\rightarrow$ b:= search(e,right(s))
    fi
end fun
{Post: b $\sim$ está E C}

TAD conjunto finito
Implementación usando ABBs

{Pre: e $\sim$ E $\wedge$ s $\sim$ C $\wedge$ abb s}
proc insert(**in** e:T, **in**/**out** s:set)
    if is_empty(s) $\rightarrow$ s:= <e>
      $\neg$ es_empty(s) $\wedge$ e < root(s) $\rightarrow$ s:= <insert(e,left(s)),root(s),right(s)>
      $\neg$ es_empty(s) $\wedge$ e = root(s) $\rightarrow$ **skip**
      $\neg$ es_empty(s) $\wedge$ e > root(s) $\rightarrow$ s:= <left(s),root(s),insert(e,right(s))>
    fi
end fun
{Post: s $\sim$ Agregar E C $\wedge$ abb s}

TAD conjunto finito
Implementación usando ABBs

{Pre: e $\sim$ E $\wedge$ s $\sim$ C $\wedge$ abb s}
proc delete(**in** e:T, **in/out** s:set)
  if $\neg$ is_empty(s) **then**
    if e < root(s) $\rightarrow$ s:= <delete(e,left(s)),root(s),right(s)>
      e = root(s) $\wedge$ is_empty(left(s)) $\rightarrow$ s:= right(s)
      e = root(s) $\wedge$ $\neg$ is_empty(left(s)) $\rightarrow$
        s:= <delete_max(left(s)),max(left(s)),right(s)>
      e > root(s) $\rightarrow$ s:= <left(s),root(s),delete(e,right(s))>
    fi
  fi
end fun
{Post: s $\sim$ borrar E C $\wedge$ abb s}