Algoritmos y Estructuras de Datos II

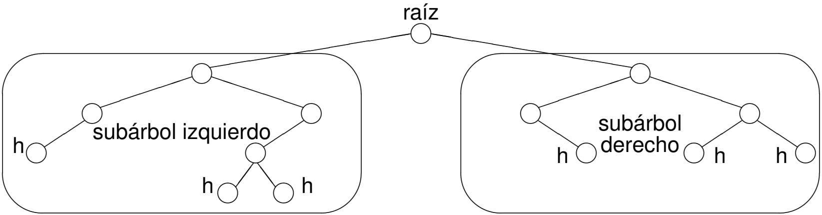
Árboles binarios de búsqueda

Clase de hoy

1. Árboles binarios
   - Especificación
   - Terminología habitual
   - Posiciones

2. Árbol binario de búsqueda
   - Ejemplos y definiciones

Intuición

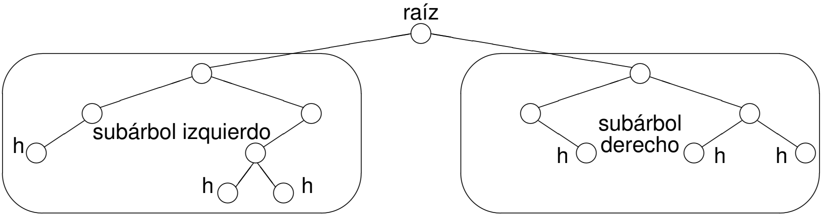


Estructura de datos que consiste en *nodos* que almacenan un elemento y dos subestructuras, que a la vez puede ser otro nodo o el vacío.

Constructores:

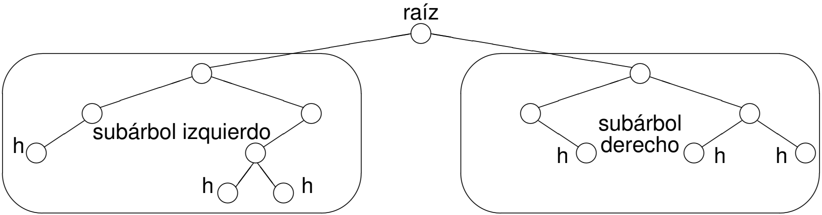
- Árbol vacío.
- Nodo consistente de un elemento de algún tipo T y dos árboles.

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:

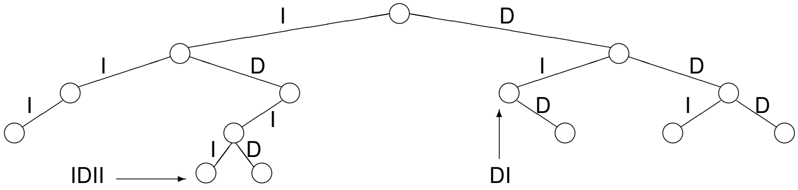
- Se usa terminología genealógica como **hijo, padre, nieto, abuelo, hermanos, ancestro, descendiente**.
- También de la botánica: **raíz, hoja**.
- Se define **camino, altura, profundidad, nivel**.

Sobre los niveles

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

Indicaciones/posiciones

Podemos recorrer un árbol indicando el camino desde la raíz.
Un camino lo podemos representar como una secuencia donde cada elemento nos indica si debo bajar a la *izquierda* o a la *derecha*.



El camino LRLL nos lleva hasta la hoja **H**.
El camino LRLLL no es válido.

Especificación del TAD Árbol Binario

```
type Direction = enumerate
                     Left
                     Right
                 end enumerate

type Path = List of Direction
```

Especificación del TAD Árbol Binario

spec Tree **of** T **where**

constructors

fun empty_tree() **ret** t : Tree **of** T
{- *crea una árbol vacío.* -}

fun node (tl : Tree **of** T, e : T, tr : Tree **of** T)
ret t : Tree **of** T
{- *crea el nodo con elemento* e *y subárboes* tl y tr. -}

Especificación del TAD Árbol Binario

operations

fun is_empty_tree(t : Tree **of** T) **ret** b : Bool
{- *Devuelve* True *si el árbol es vacío* -}

fun root(t : Tree **of** T) **ret** e : T
{- *Devuelve el elemento que se encuentra en la raíz de* t. -}
{- **PRE:** not is_empty_tree(t) -}

fun left(t : Tree **of** T) **ret** tl : Tree **of** T
{- *Devuelve el subárbol izquierdo de* t. -}
{- **PRE:** not is_empty_tree(t) -}

fun right(t : Tree **of** T) **ret** tl : Tree **of** T
{- *Devuelve el subárbol derecho de* t. -}
{- **PRE:** not is_empty_tree(t) -}

fun height(t : Tree **of** T) **ret** n : Nat
{- *Devuelve la distancia que hay entre la raíz de* t *y la hoja más profunda*. -}

fun is_path(t : Tree **of** T, p : Path) **ret** b : Bool
{- *Devuelve* True *si* p *es un camino válido en* t -}

fun subtree_at(t : Tree **of** T, p : Path) **ret** t0 : Tree **of** T
{- *Devuelve el subárbol que se encuentra al recorrer el camino* p *en* t . -}

fun elem_at(t : Tree **of** T, p : Path) **ret** e : T
{- *Devuelve el elemento que se encuentra al recorrer el camino* p *en* t . -}
{- **PRE:** is_path(t,p) -}

Implementación de árboles binarios

La manera usual de implementar árboles binarios en lenguajes imperativos con punteros es similar a la que usamos cuando implementamos listas enlazadas. Definimos un tipo *node* como una tupla con un elemento y dos punteros a sí mismo:

```
implement Tree of T where

type Node of T = tuple
                   left: pointer to (Node of T)
                   value: T
                   right: pointer to (Node of T)
               end tuple

type Tree of T= pointer to (Node of T)
```

Implementación de árboles binarios

```
fun empty_tree() ret t : Tree of T
    t := null
end fun

fun node (tl : Tree of T, e : T, tr : Tree of T) ret t : Tree of T
    alloc(t)
    t->value := e
    t->left := tl
    t->right := tr
end fun
```

Árboles binarios de búsqueda

- Son casos particulares de árboles binarios,
- son árboles binarios t en donde la información está organizada de forma tal que el siguiente 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 algoritmo.
- Si el árbol está “balanceado”, es un algoritmo logarítmico.

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.

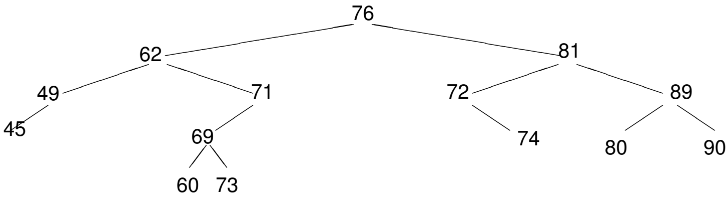
Si se cumplen estas condiciones, decimos que t es un **árbol binario de búsqueda** o **ABB**.

Agregar un elemento en un ABB

Si quiero agregar un elemento e a un Árbol Binario de Búsqueda t de manera de mantener la propiedad debo realizar el siguiente procedimiento recursivo:

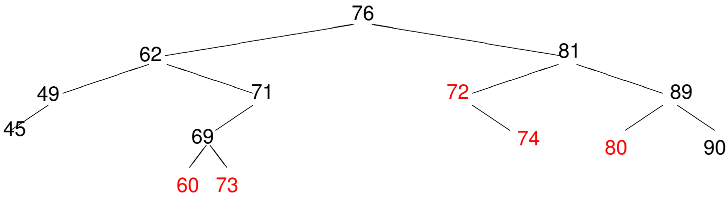
- Si t es vacío, formo el nodo que consta del elemento e y los dos subárboles vacíos.
- En caso contrario, comparo el elemento e con la raíz del árbol t,
- Si e es menor que la raíz, lo agrego al subárbol izquierdo.
- Si e es mayor o igual a la raíz, lo agrego al subárbol derecho.

Ejemplo



¿Es un árbol binario de búsqueda?

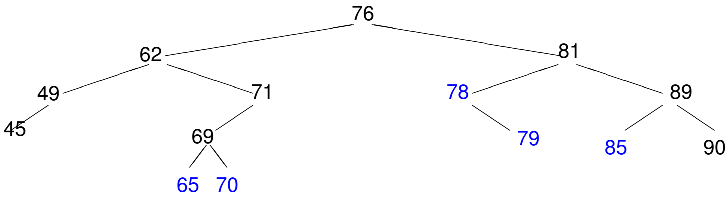
Ejemplo



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.