Algoritmos y Estructuras de Datos II

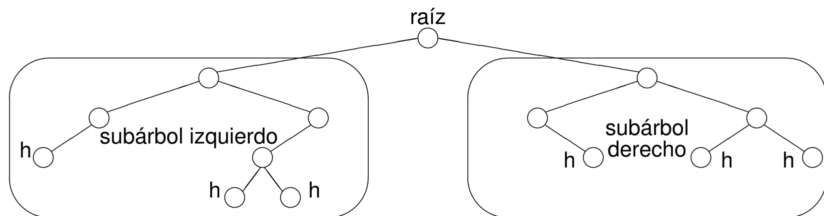
Árboles binarios de búsqueda

24 de abril de 2017

Clase de hoy

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

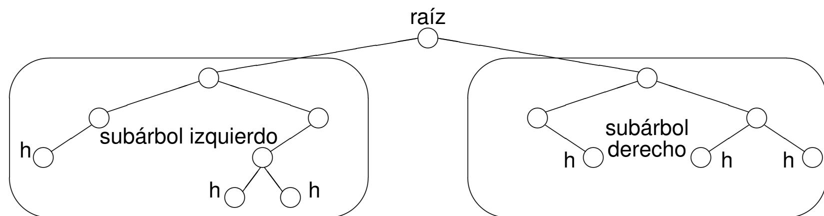
Intuición



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

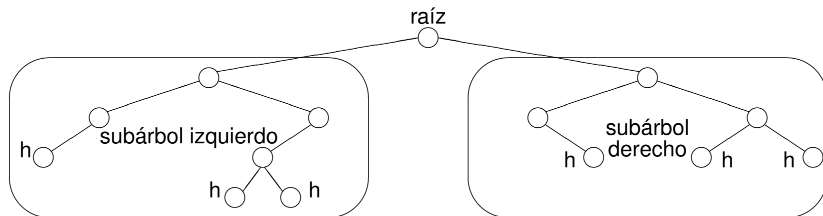
Intuición



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

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

Clase de hoy

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

Clase de hoy

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

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.

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.

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.

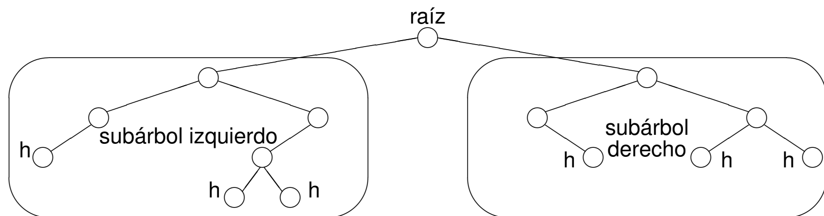
Notación $<>$

- 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.

Notación $<>$

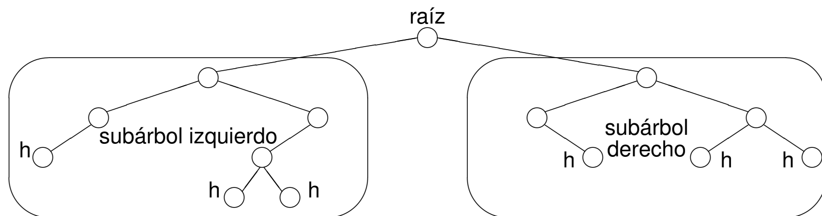
- Notar la sobrecarga de la notación $<>$:
  - $<>$ es el árbol vacío,
  - $<i,r,d>$ es el árbol no vacío cuya raíz es r, subárbol izquierdo es i y subárbol derecho es d.
  - $<r>$ es la hoja $<<>,r,<>>$
- Conclusión: la notación $<>$ puede tener 0, 1 ó 3 argumentos.

Botánica y genealogía



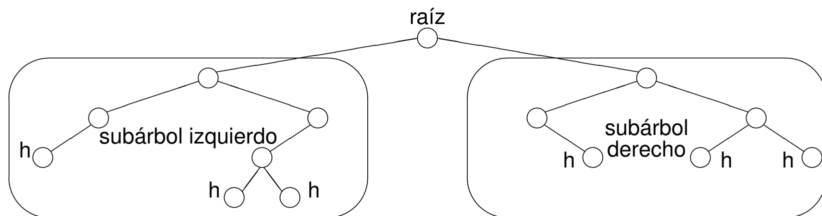
- Un **nodo** es un árbol no vacío.
- Tiene **raíz**, **subárbol izquierdo** y **subárbol derecho**.
- A los subárboles se los llama también **hijos** (izquierdo y derecho).
- Y al nodo se le dice **padre** de sus hijos.
- Una **hoja** es un nodo con los dos hijos vacíos.

Botánica y genealogía



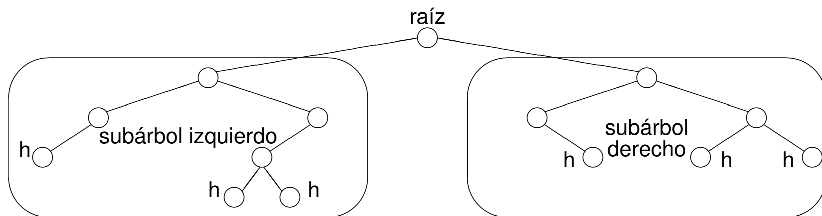
- Un **nodo** es un árbol no vacío.
- Tiene **raíz**, **subárbol izquierdo** y **subárbol derecho**.
- A los subárboles se los llama también **hijos** (izquierdo y derecho).
- Y al nodo se le dice **padre** de sus hijos.
- Una **hoja** es un nodo con los dos hijos vacíos.

Botánica y genealogía



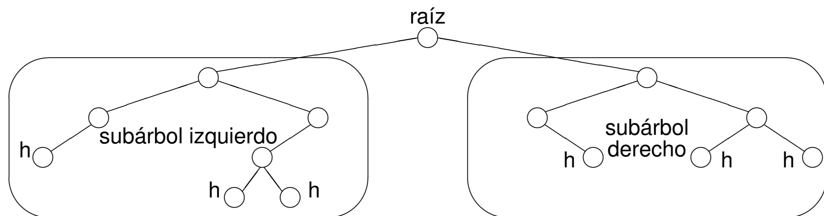
- Un **nodo** es un árbol no vacío.
- Tiene **raíz**, **subárbol izquierdo** y **subárbol derecho**.
- A los subárboles se los llama también **hijos** (izquierdo y derecho).
- Y al nodo se le dice **padre** de sus hijos.
- Una **hoja** es un nodo con los dos hijos vacíos.

Botánica y genealogía



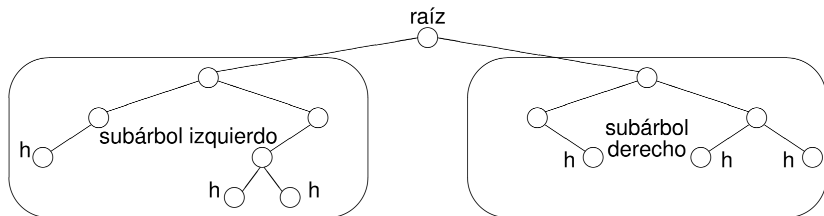
- Un **nodo** es un árbol no vacío.
- Tiene **raíz**, **subárbol izquierdo** y **subárbol derecho**.
- A los subárboles se los llama también **hijos** (izquierdo y derecho).
- Y al nodo se le dice **padre** de sus hijos.
- Una **hoja** es un nodo con los dos hijos vacíos.

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.

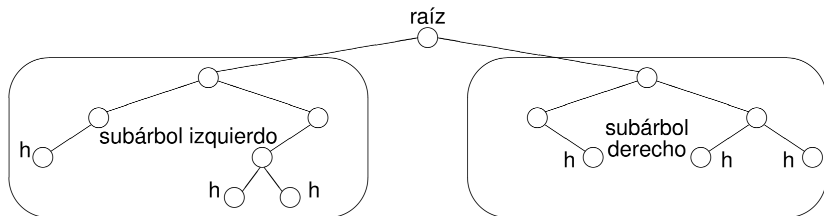
Más terminología



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**.

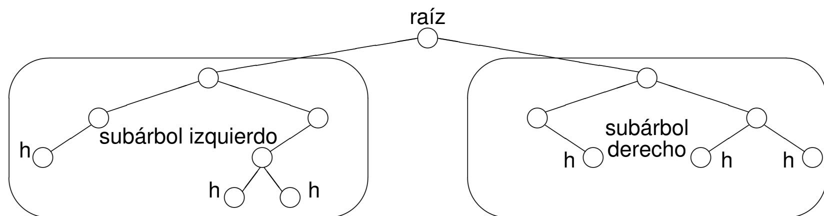
Más terminología



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**.

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 no hay ningún nodo.
- 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 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.

Sobre los niveles

- En el nivel 0 no hay ningún nodo.
- 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 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.

Sobre los niveles

- En el nivel 0 no hay ningún nodo.
- 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 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.

Sobre los niveles

- En el nivel 0 no hay ningún nodo.
- 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 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.

Sobre los niveles

- En el nivel 0 no hay ningún nodo.
- 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 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.

Sobre los niveles

- En el nivel 0 no hay ningún nodo.
- 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 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.

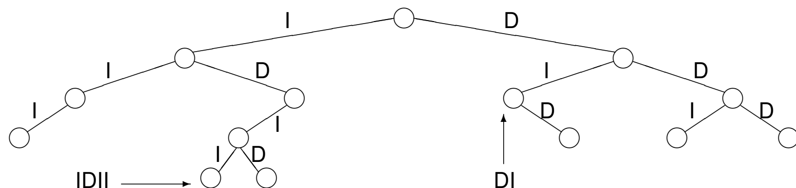
Sobre los niveles

- En el nivel 0 no hay ningún nodo.
- 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 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.

Clase de hoy

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

Indicaciones/posiciones



$<> \downarrow p = <>$
$<i,e,d> \downarrow R = <i,e,d>$
$<i,e,d> \downarrow (I\ p) = i \downarrow p$
$<i,e,d> \downarrow (D\ p) = d \downarrow p$

$<i,e,d>.R = e$
$<i,e,d>.(I\ p) = i.p$
$<i,e,d>.(D\ p) = d.p$

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

Clase de hoy

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

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 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.

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 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.

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 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.

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 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.

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 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.

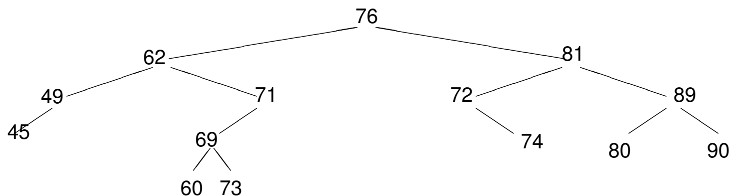
Árboles binarios de búsqueda

- Son casos particulares de árboles binarios,
- son árboles binarios t en donde la información está organizada de tal forma de que 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.

Árboles binarios de búsqueda

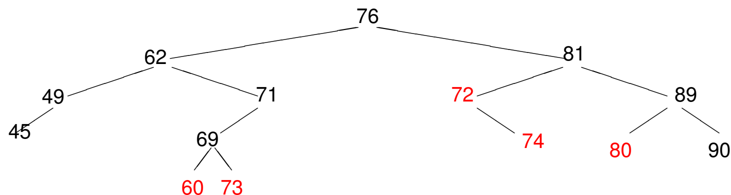
- Son casos particulares de árboles binarios,
- son árboles binarios t en donde la información está organizada de tal forma de que el siguiente algoritmo sencillo permite buscar eficientemente un elemento:
- el elemento buscado se compara con la raíz de t
  - si es el mismo, la búsqueda finaliza
  -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

Ejemplo



¿Es un árbol binario de búsqueda?

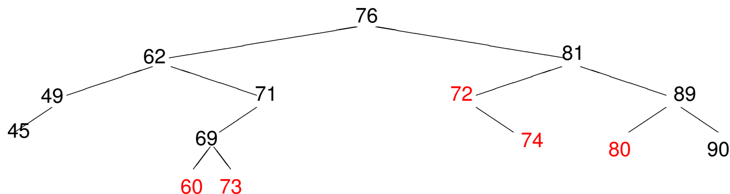
Ejemplo



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.

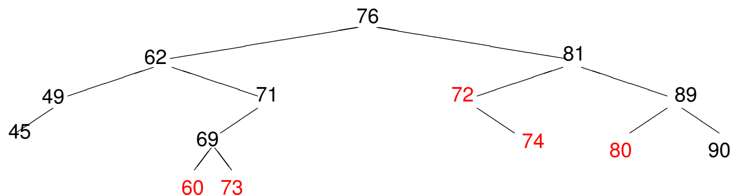
Ejemplo



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.

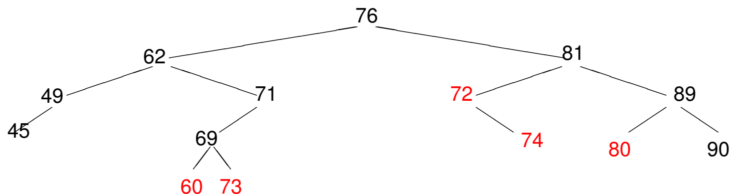
Ejemplo



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.

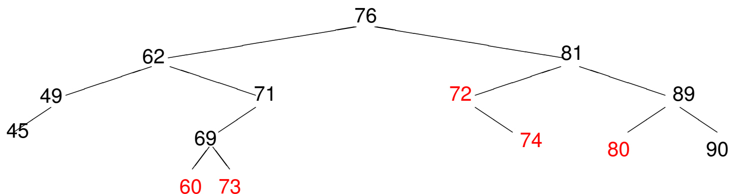
Ejemplo



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.

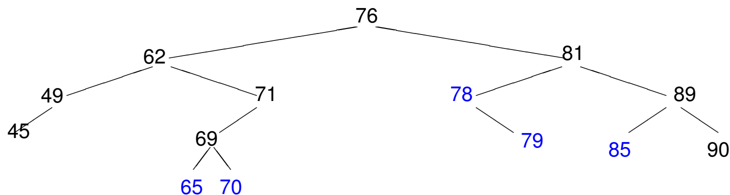
Ejemplo



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 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 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**.

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 $p \in pos(t)$.

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 $p \in pos(t)$.

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 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 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.

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.

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.

Formalizando la definición

- Para todo $p \in pos(t)$,
  - los valores del árbol de la forma $t.(p \triangleleft 0 +\!\!+ q)$ deben ser menores que $t.p$
  - los valores del árbol de la forma $t.(p \triangleleft 1 +\!\!+ q)$ deben ser mayores que $t.p$
- habría que aclarar que siempre y cuando $p \triangleleft 0 +\!\!+ q$ y $p \triangleleft 1 +\!\!+ q$ 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 $\forall p \in \mathrm{pos}(t).\forall q \in \mathrm{pos}$

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

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 $\forall p \in \mathrm{pos}(t). \forall q \in \mathrm{pos}$

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

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 $\forall p \in \mathrm{pos}(t). \forall q \in \mathrm{pos}$

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

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 $\forall p \in \mathrm{pos}(t).\forall q \in \mathrm{pos}$

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

Clase de hoy

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

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

Agregar e (Agregar e’ c) = Agregar e’ (Agregar e c)
Agregar e (Agregar e c) = Agregar e c

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 ~ Vacío}

fun is_empty(s:set) ret b:Bool
    b:= (s = < >)
end fun
{Post: b = (s ~ Vacío)}
```

TAD conjunto finito

Implementación usando ABBs

{Pre: e ∼ E ∧ s ∼ C ∧ abb s}
fun search(e:T,s:set) **ret** b:**Bool**
  if is_empty(s) → b:= False
   ¬ es_empty(s) ∧ e < root(s) → b:= search(e,left(s))
   ¬ es_empty(s) ∧ e = root(s) → b:= True
   ¬ es_empty(s) ∧ e > root(s) → b:= search(e,right(s))
  fi
end fun
{Post: b ∼ está E C}

TAD conjunto finito

Implementación usando ABBs

```
{Pre: e ∼ E ∧ s ∼ C ∧ abb s}
proc insert(in e:T, in/out s:set)
      if is_empty(s) → s:= <e>
         ¬ es_empty(s) ∧ e < root(s) → s:= <insert(e,left(s)),root(s),right(s)>
         ¬ es_empty(s) ∧ e = root(s) → skip
         ¬ es_empty(s) ∧ e > root(s) → s:= <left(s),root(s),insert(e,right(s))>
      fi
end proc
{Post: s ∼ Agregar E C ∧ abb s}
```

TAD conjunto finito
Implementación usando ABBs

```
{Pre: e ∼ E ∧ s ∼ C ∧ abb s}
proc delete(in e:T, in/out s:set)
      if ¬ is_empty(s) then
        if e < root(s) → s:= <delete(e,left(s)),root(s),right(s)>
          e = root(s) ∧ is_empty(left(s)) → s:= right(s)
          e = root(s) ∧ ¬ is_empty(left(s)) →
                           s:= <delete_max(left(s)),max(left(s)),right(s)>
          e > root(s) → s:= <left(s),root(s),delete(e,right(s))>
        fi
      fi
end proc
{Post: s ∼ borrar E C ∧ abb s}
```

Conclusión

- Usando árboles binarios de búsqueda, hemos implementado el TAD conjunto con operaciones:
  - search (implementación de la operación está) de orden $\log n$,
  - insert (implementación de la operación agregar) de orden $\log n$,
  - delete (implementación de la operación borrar) de orden $\log n$,
  - las otras dos operaciones (empty, is_empty) son constantes.
- (asumiendo que el árbol binario de búsqueda está balanceado)

Conclusión

- Usando árboles binarios de búsqueda, hemos implementado el TAD conjunto con operaciones:
  - search (implementación de la operación está) de orden $\log n$,
  - insert (implementación de la operación agregar) de orden $\log n$,
  - delete (implementación de la operación borrar) de orden $\log n$,
  - las otras dos operaciones (empty, is_empty) son constantes.
- (asumiendo que el árbol binario de búsqueda está balanceado)

Conclusión

- Usando árboles binarios de búsqueda, hemos implementado el TAD conjunto con operaciones:
  - search (implementación de la operación está) de orden $\log n$,
  - insert (implementación de la operación agregar) de orden $\log n$,
  - delete (implementación de la operación borrar) de orden $\log n$,
  - las otras dos operaciones (empty, is_empty) son constantes.
- (asumiendo que el árbol binario de búsqueda está balanceado)

Conclusión

- Usando árboles binarios de búsqueda, hemos implementado el TAD conjunto con operaciones:
  - search (implementación de la operación está) de orden $\log n$,
  - insert (implementación de la operación agregar) de orden $\log n$,
  - delete (implementación de la operación borrar) de orden $\log n$,
  - las otras dos operaciones (empty, is_empty) son constantes.
- (asumiendo que el árbol binario de búsqueda está balanceado)

Conclusión

- Usando árboles binarios de búsqueda, hemos implementado el TAD conjunto con operaciones:
  - search (implementación de la operación está) de orden $\log n$,
  - insert (implementación de la operación agregar) de orden $\log n$,
  - delete (implementación de la operación borrar) de orden $\log n$,
  - las otras dos operaciones (empty, is_empty) son constantes.
- (asumiendo que el árbol binario de búsqueda está balanceado)

Conclusión

- Usando árboles binarios de búsqueda, hemos implementado el TAD conjunto con operaciones:
    - search (implementación de la operación está) de orden $\log n$,
    - insert (implementación de la operación agregar) de orden $\log n$,
    - delete (implementación de la operación borrar) de orden $\log n$,
    - las otras dos operaciones (empty, is_empty) son constantes.
- (asumiendo que el árbol binario de búsqueda está balanceado)

Implementación de ABB con punteros

Recordemos la implementación de árboles binarios con punteros:

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

Implementación de ABB con punteros

Con sus operaciones:

fun empty() **ret** t:bintree
fun node(l:bintree,e:elem,r:bintree) **ret** t:bintree
fun root(t:bintree) **ret** e:elem
fun left(t:bintree) **ret** l:bintree
fun right(t:bintree) **ret** r:bintree
fun is_empty(t:bintree) **ret** b:**bool**
proc destroy(**in**/**out** t:bintree)

vistas en las filminas de árboles binarios.

emptyABB

```
proc emptyABB(out s:set)
        s:= empty()
end proc

fun is_emptyABB(s:set) ret b:Bool
      b:= is_empty(s)
end fun
```

searchABB

```
fun searchABB(e:T,s:set) ret b:Bool
    if is_empty(s) → b:= False
       ¬ es_empty(s) ∧ e < root(s) → b:= searchABB(e,left(s))
       ¬ es_empty(s) ∧ e = root(s) → b:= True
       ¬ es_empty(s) ∧ e > root(s) → b:= searchABB(e,right(s))
    fi
end fun
```

insertABB

proc insertABB(**in** e:T, **in/out** s:set)
  if is_empty(s) $\rightarrow$ s:= node(emptyABB(),e,emptyABB())
   $\neg$ es_empty(s) $\wedge$ e < root(s) $\rightarrow$ insertABB(e,left(s))
   $\neg$ es_empty(s) $\wedge$ e = root(s) $\rightarrow$ **skip**
   $\neg$ es_empty(s) $\wedge$ e > root(s) $\rightarrow$ insertABB(e,right(s))
  fi
end proc

deleteABB

```
proc deleteABB(in e:T, in/out s:set)
    var q : pointer to node
    var m : T
    var lft,rgt : bintree
    if ¬ is_empty(s) then
      if e < root(s) → deleteABB(e,left(s))
         e = root(s) ∧ is_empty(left(s)) → q:= s
                                           s:= right(s)
                                           free(q)
         e = root(s) ∧ ¬ is_empty(left(s)) → rgt:= right(s)
                                             m:= max(left(s))
                                             lft:= delete_max(left(s))
                                             q:= s
                                             s:= node(lft,m,rgt)
                                             free(q)
         e > root(s) → deleteABB(e,right(s))
      fi
    fi
end proc
```