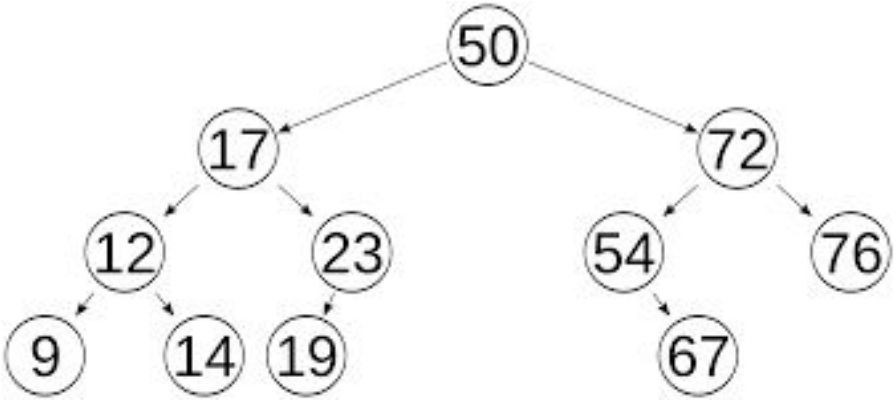
ITSC - Programación III

Árboles binarios de búsqueda

Temas

- definición
- insertar un elemento
- complejidad de búsqueda
- El TAD Conjunto finito

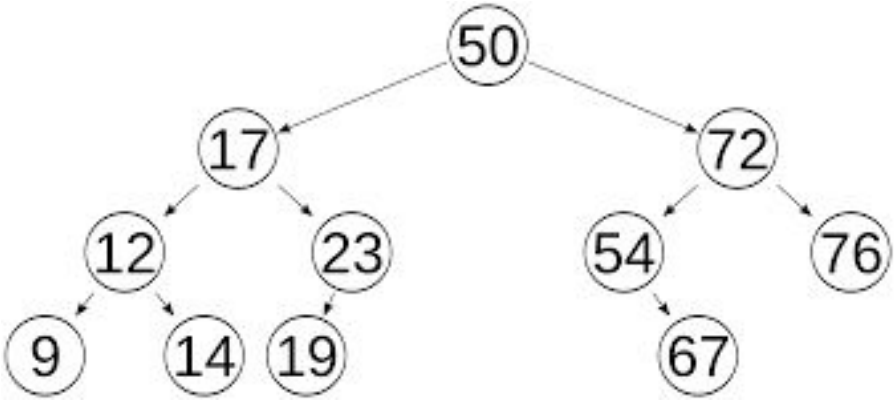
Definición



Un árbol binario de búsqueda es un tipo particular de árbol binario. Se cumple que para buscar un elementos podemos hacer lo siguiente:

1) el elemento buscado se compara con la raíz del árbol. Si es igual, la búsqueda finaliza.
2) si es menor, seguimos buscando por la izquierda (con el mismo algoritmo)
3) si es mayor, seguimos buscando por la derecha (con el mismo algoritmo)
4) ¿qué significa llegar a un árbol vacío?

Definición



Un árbol binario de búsqueda es un tipo particular de árbol binario. Se cumple que para buscar un elementos podemos hacer lo siguiente:

1) el elemento buscado se compara con la raíz del árbol. Si es igual, la búsqueda finaliza.
2) si es menor, seguimos buscando por la izquierda (con el mismo algoritmo)
3) si es mayor, seguimos buscando por la derecha (con el mismo algoritmo)
4) si llego a un árbol vacío, significa que el elemento buscado no está en el árbol.

Inserción

La inserción debe garantizar que en todo momento el árbol binario continúa siendo un árbol binario **de búsqueda**. Para esto, cada subárbol debe cumplir que:

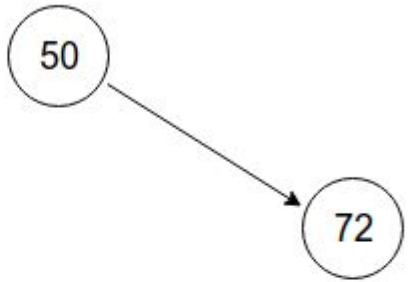
- o bien es vacío
- o bien guarda la siguiente relación con sus subárboles:
  - el elemento en la raíz es **mayor** que el elemento en la raíz del subárbol izquierdo (si hay subárbol izquierdo)
  - elemento en la raíz es **menor** que el elemento en la raíz del subárbol derecho (si hay subárbol derecho)

Inserción

- Comencemos con un árbol con raíz y dos subárboles vacíos.

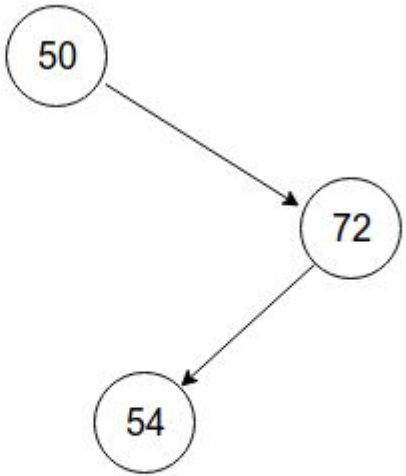
Inserción

- Comencemos con un árbol con raíz y dos subárboles vacíos.
- Insertamos el elemento 72 (a la derecha, porque es mayor)



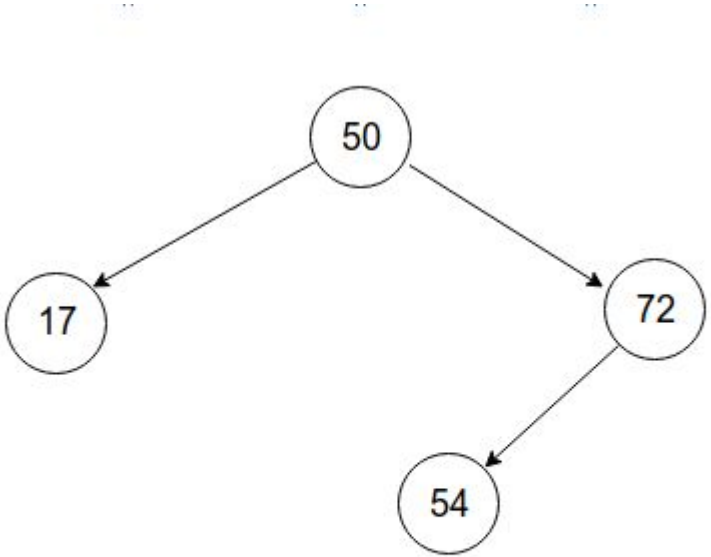
Inserción

- Comencemos con un árbol con raíz y dos subárboles vacíos.
- Insertamos el elemento 72
- Insertamos el elemento 54 (a la derecha del 50, pero a la izquierda del 72)



Inserción

- Comencemos con un árbol con raíz y dos subárboles vacíos.
- Insertamos el elemento 72
- Insertamos el elemento 54
- Insertamos el elemento 17 (a la izquierda del 50).



Inserción

Es un caso perfecto para las funciones recursivas:

- Cada árbol decide si se agrega el elemento en la subrama derecha o la subrama izquierda.
- ¿Cuál es el caso base? ¿Dónde termina la recursión?
- La respuesta no está aquí porque es parte del práctico.

Usos

El TAD Árbol binario de búsqueda organizar los elementos de forma que para encontrar algo (¡o descubrir que no está!) no necesitamos buscar en TODO el árbol (aunque veremos excepciones).

Nos interesa utilizar un ABB para almacenar cierta información en contextos donde la **búsqueda** es una operación común. Queremos garantizar que se pueda realizar rápidamente.

Usos

En cada paso de la búsqueda, o bien encontramos el elemento, o bien averiguamos en cuál de los dos subárboles puede estar.

Usos

Un caso de uso típico es poner en la raíz del árbol un par (Clave, Valor): usamos la clave para buscar, pero en realidad estamos interesados en el valor. Por ejemplo:

(palabra, significado)

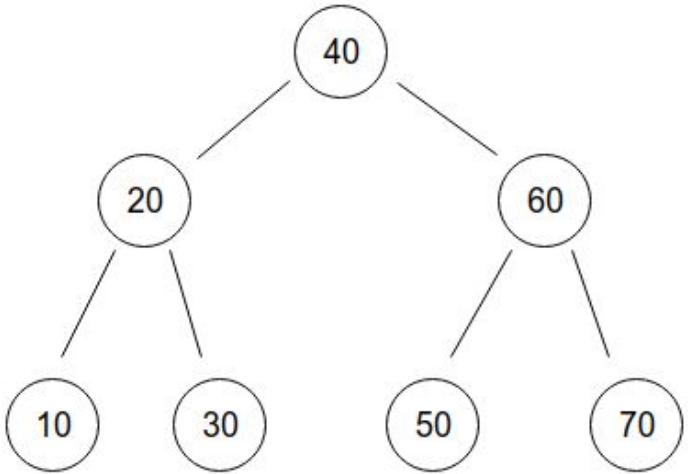
(user_id, Información de perfil)

(dominio, dirección IP)

Complejidad de la búsqueda

¿Cuántas comparaciones debo realizar en este ABB para encontrar…

- el 10?
- el 50?
- el 60?
- el 54?



Complejidad de la búsqueda

Para encontrar un elemento dentro de un ABB se realiza una cantidad de operaciones que es proporcional a la altura del árbol. Pequeño repaso:

- La **profundidad** de un nodo es la longitud del camino que va desde la raíz hasta él. (Notar que la profundidad de la raíz es 0).
- La **altura** del árbol es la máxima profundidad de cualquier nodo en el árbol.

Esto es intuitivamente claro: mientras más “alto” es el árbol, más veces podemos seguir “bajando” para ver si encontramos (o no) el elemento que buscamos.

Complejidad de la búsqueda

Entonces en un árbol de altura *h*, tendremos que hacer a lo sumo (peor caso) *h + 1* comparaciones.

¿Pero cuántos nodos hay en un árbol de altura *h*?

O sea, ¿hicimos *h* + 1 operaciones para buscar entre **cuántos** nodos?

La altura del árbol y la cantidad de nodos guardan ciertas relaciones, veamos:

Complejidad de la búsqueda

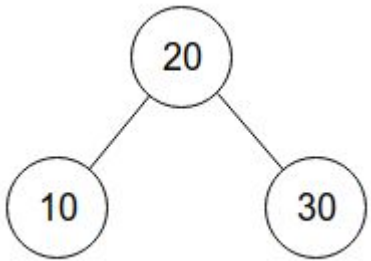
Notemos que:

- un árbol binario de altura 0 puede tener un único elemento

Complejidad de la búsqueda

Notemos que:

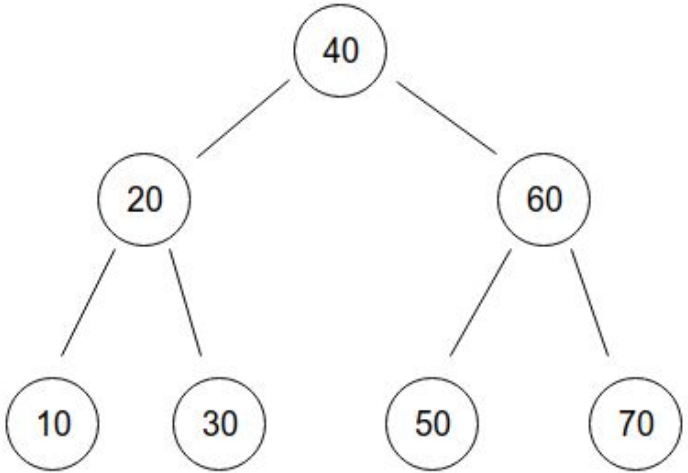
- un árbol binario de altura 0 puede tener un único elemento
- un árbol binario de altura 1 puede tener hasta 3 elementos



Complejidad de la búsqueda

Notemos que:

- un árbol binario de altura 0 puede tener un único elemento
- un árbol binario de altura 1 puede tener hasta 3 elementos
- un árbol binario de altura 2 puede tener hasta 7 elementos



Complejidad de la búsqueda

Notemos que cada nivel que agregamos puede tener exactamente el doble de nodos que el nivel anterior:

- Nivel 0: capacidad para 1 nodo
- Nivel 1: capacidad para 2 nodos
- Nivel 2: capacidad para 4 nodos
- …
- Nivel n: capacidad para 2^n

Complejidad de la búsqueda

Si sumamos las capacidades de los niveles

- un árbol de altura 0 (sólo nivel 0) puede tener sólo 1 nodo
- un árbol de altura 1 (niveles 0, 1) puede tener hasta 1 + 2 = 3 nodos
- un árbol de altura 2 (niveles 0, 1, 2) puede tener hasta 1 + 2 + 4 = 7 nodos
- un árbol de altura 3 (niveles 0, 1, 2, 3) puede tener hasta 1 + 2 + 4 + 8 = 15 nodos
- ...

¿Notás algún patrón en esta secuencia?

1, 2, 4, 8, ...

Complejidad de la búsqueda

Si sumamos las capacidades de los niveles

- un árbol de altura 0 (sólo nivel 0) puede tener sólo 1 nodo
- un árbol de altura 1 (niveles 0, 1) puede tener hasta 1 + 2 = 3 nodos
- un árbol de altura 2 (niveles 0, 1, 2) puede tener hasta 1 + 2 + 4 = 7 nodos
- un árbol de altura 3 (niveles 0, 1, 2, 3) puede tener hasta 1 + 2 + 4 + 8 = 15 nodos
- ...

Son las potencias de 2:

$2^0, 2^1, 2^2, 2^3, ...$

Complejidad de la búsqueda

Generalizando, un árbol binario de altura *h* (niveles 0, 1, 2, … h) puede tener hasta $1 + 2 + ... + 2^h = 2^{h+1} - 1$ nodos

Complejidad de la búsqueda

Un árbol de altura *h* **puede** tener a lo sumo $2^{h+1} - 1$ nodos.

Pero cuidado, lo contrario no es cierto: si un árbol tiene $2^{h+1} - 1$ nodos, no necesariamente tiene altura *h*. ¿Por qué no?

Complejidad de la búsqueda

Un árbol de altura *h* **puede** tener a lo sumo $2^{h+1} - 1$ nodos.

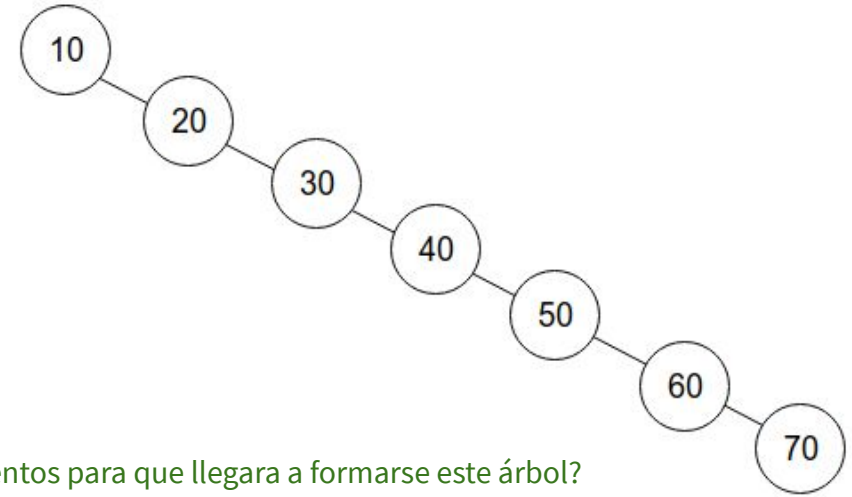
Pero cuidado, lo contrario no es cierto: si un árbol tiene $2^{h+1} - 1$ nodos, no necesariamente tiene altura *h*. ¿Por qué no?

Ese caso sólo se da si todos los niveles (salvo quizás el último) **tienen todos los nodos que pueden tener**. En ese caso, se dice que el árbol está **balanceado**.

Complejidad de la búsqueda

¿Cuántas comparaciones debo realizar en este ABB para encontrar…

- el 10?
- el 50?
- el 60?
- el 54?

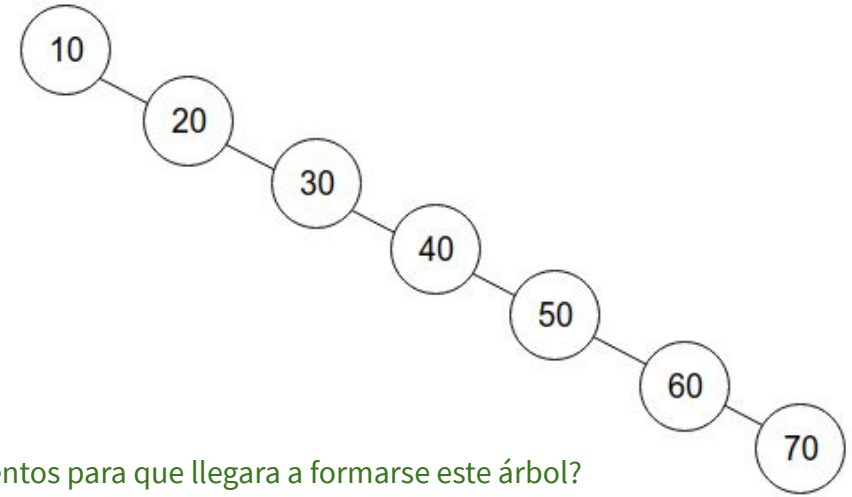


Para pensar: ¿en qué orden se insertaron lo elementos para que llegara a formarse este árbol?

Complejidad de la búsqueda

¿Cuántas comparaciones debo realizar en este ABB para encontrar…

- el 10?
- el 50?
- el 60?
- el 54?



Para pensar: ¿en qué orden se insertaron lo elementos para que llegara a formarse este árbol?

EXACTAMENTE EN ORDEN (10, 20, 30, ...)

Complejidad de la búsqueda

Este es un ABB, ya que cumple la propiedad que le pedimos:

Cada subárbol cumple que

- es vacío o
- el elemento en la raíz es mayor que el elemento en la raíz del subárbol izquierdo (si no está vacío)
- el elemento en la raíz es menor que el elemento en la raíz del subárbol derecho (si no está vacío)

Sin embargo, buscar en este árbol no es muy distinto que buscar en una lista...

Complejidad de la búsqueda

Para garantizarnos que no vamos a caer en ese caso fatal (independientemente del orden de inserción de los elementos), es necesario que el árbol se mantenga lo más balanceado posible (dicho de otra forma, que su altura sea siempre la mínima indispensable).

En esta materia **no** vamos a profundizar en algoritmos para lograr esto. Si te interesa, acá hay más info (wikipedia).

Complejidad de la búsqueda

Resumiendo, en un árbol binario de búsqueda de altura *h*, en el peor caso, es necesario realizar *h + 1* operaciones para encontrar el elemento buscado (o asegurarse de que no está presente).

Si el árbol está balanceado, quiere decir que con *h* operaciones buscamos entre aproximadamente 2^{h+1} elementos.

Complejidad de la búsqueda

Para darse cuenta del poder del árbol binario de búsqueda:

Si el árbol está bastante balanceado y tiene altura 39, con sólo 40 comparaciones buscamos un elemento entre cerca de 2^{40} = 1.099.511.627.776.

Esto es aproximadamente 3.7 veces el número de estrellas en nuestra galaxia.

TAD Conjunto

Otro uso importante que le podemos dar a un ABB es como base para implementar el TAD Conjunto.

Un conjunto es un contenedor donde los elementos no se repiten.

Estamos interesados en hacer **preguntas** como: `conjunto.tiene(elemento)`

Estamos interesados en hacer **operaciones** como `conjunto.agregar(elemento)` y que si el mismo ya está presente, el conjunto no se modifique.