Arboles Binarios

Estructuras de Datos

1

Arboles Binarios

- Es una estructura de datos fundamental
- Combina las ventajas de arreglos y listas enlazadas
  - Tiempo de búsqueda rápido
  - inserción rápida
  - eliminación rápida
  - Tiempo de acceso moderadamente rápido
- Por supuesto, son un poco más complejos de implementar

2

Recordemos los arreglos ordenados ...

- **Su tiempo de búsqueda es más rápido, porque hay un cierto 'ordenamiento' de los elementos.**
  - **Podemos usar búsqueda binaria, O (log n)**
  - **En lugar de búsqueda lineal, O (n)**

En Árboles

- **Su tiempo de inserción es más lento, porque tiene que encontrar la posición correcta primero, para luego insertar**
  - **Para eso se necesita O (log n)**
  - **En lugar de simplemente dejar caer el elemento al final, O (1)**
- **Los árboles proporcionan 'algo' de ordenamiento**
  - **Cada uno de estos algoritmos será O (log n)**

3

Recordemos las Listas Enlazadas...

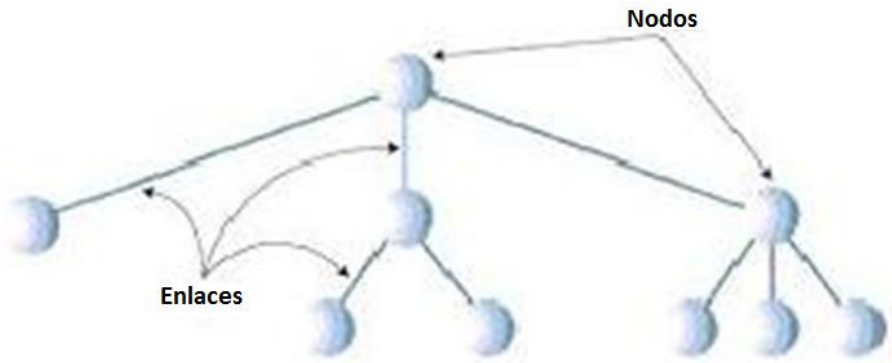
- Inserción y eliminación son rápidos
  - O (1) en el extremo
  - En el centro, O (n) para encontrar la posición, pero O (1) para insertar / eliminar
  - Mejor que el desplazamiento caro de los arreglos
- Búsqueda es más lenta, O (n)

- Los arboles realizan inserción / eliminación de manera similar, cambiando las referencias
- Pero proporcionan caminos más cortos para buscar, que son de largo log(n), en oposición a una lista enlazada que podría ser de longitud n

4

Arboles: General

- Un árbol consta de nodos, conectados por enlaces
- Los árboles no pueden tener ciclos
  - De lo contrario, es un grafo
- Aquí hay un árbol típico:



5

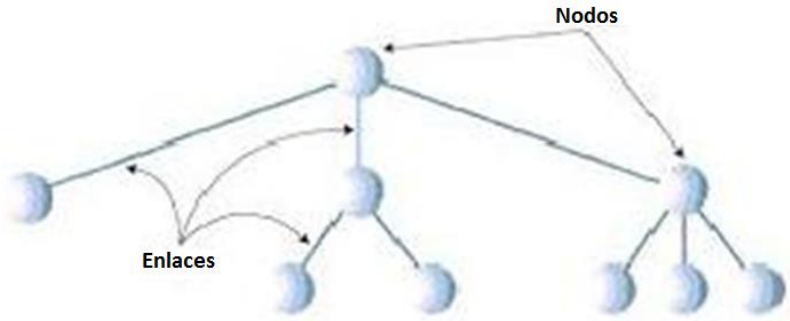
Atravesando un Árbol

- **Comience en la raíz y muévase hacia abajo a lo largo de sus enlaces**
- **Normalmente, los enlaces representan algún tipo de relación**
- **Nosotros representamos estos por referencias**
- **Al igual que en las listas enlazadas:**

```
class Link {
    int data;
    Link next;
}
```

- **En un árbol:**

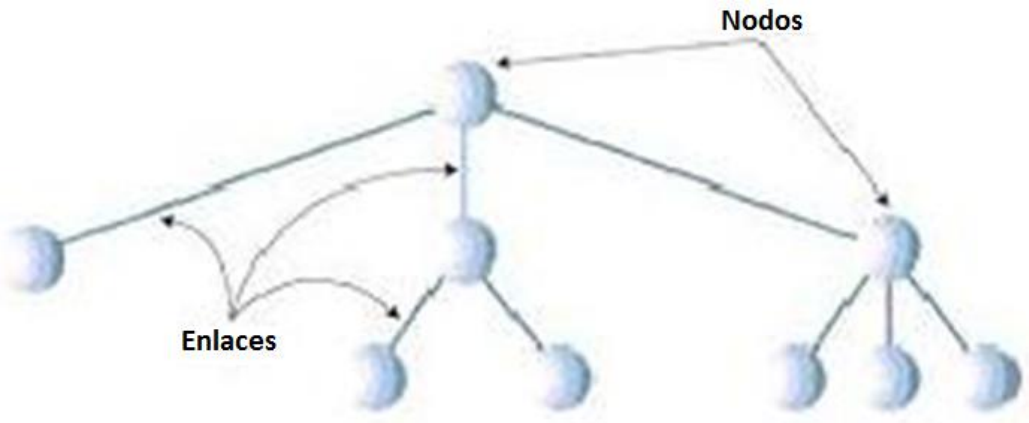
```
class Node {
    int data;
    Node child1;
    Node child2;
    …
}
```



6

Tamaño de un árbol

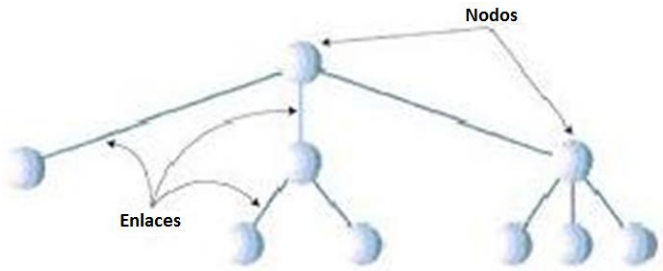
- **Aumenta a medida que avanzas hacia abajo**
- **Opuesto a la naturaleza.**



7

Arboles Binarios

- **Un tipo especial de árbol**
- **Con este árbol, los nodos tienen diferentes cantidades de hijos:**

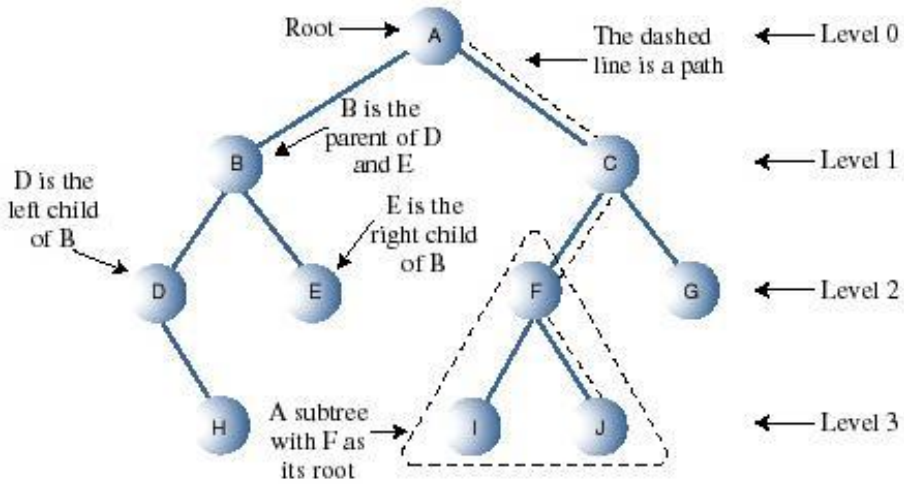


- **En los árboles binarios, los nodos pueden tener un máximo de dos hijos.**
  - **El árbol de arriba se llama un árbol multipunto**

8

Un Árbol Binario

- **Por ahora, tenga en cuenta que cada nodo tiene a lo más dos hijos**

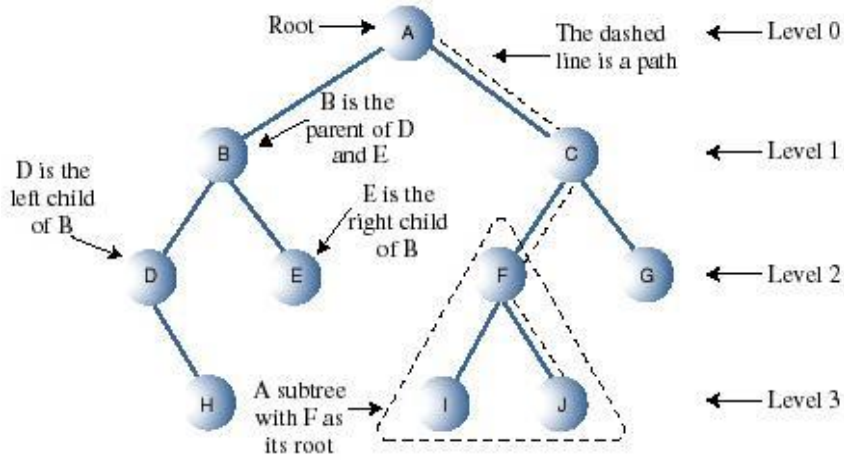


9

H, E, I, J, and G are leaf nodes

Un Árbol Binario

- **Cada nodo tiene un hijo izquierdo y un hijo derecho**
  - **Como debería lucir la clase Java?**

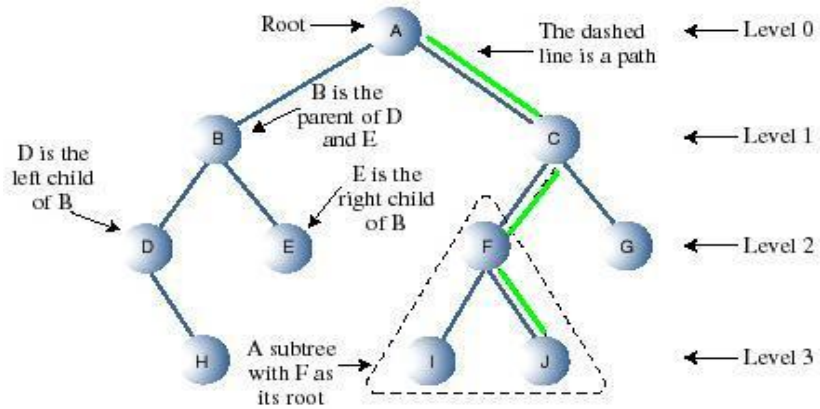


H, E, I, J, and G are leaf nodes

10

Árbol Binario: Términos

- **Camino: Secuencia de nodos conectados por enlaces**
  - **La línea verde es un camino desde A a J**

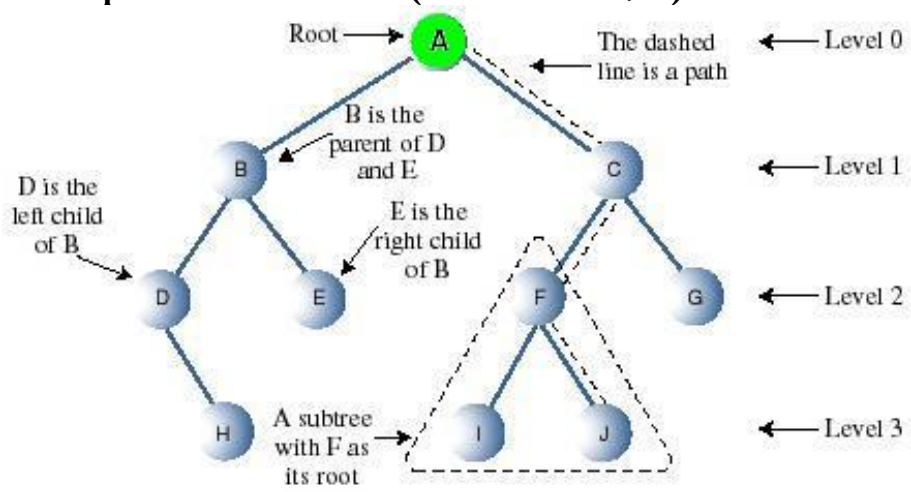


H, E, I, J, and G are leaf nodes

11

Árbol Binario: Términos

- **Raíz: el nodo en la cima del árbol**
  - **Sólo puede haber uno (en este caso, A)**

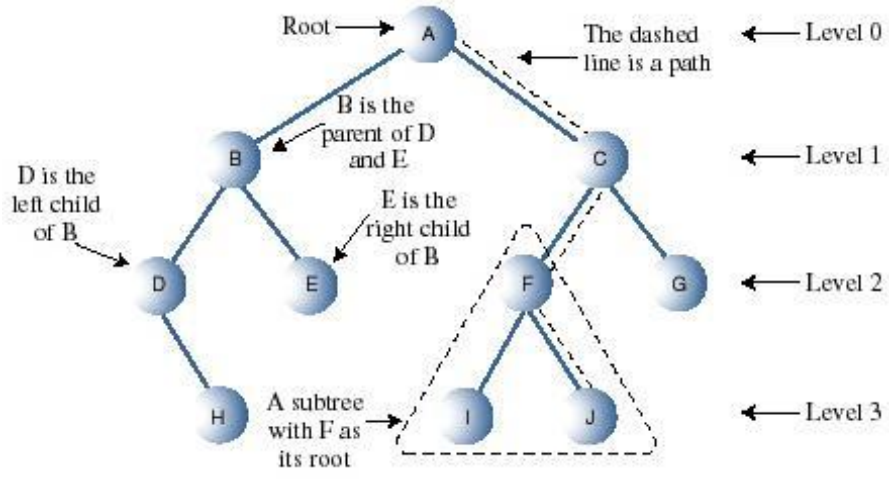


12

H, E, I, J, and G are leaf nodes

Árbol Binario: Términos

- **Padre: El nodo en la cima. (B es el padre de D, A es el padre de B, A es el abuelo de D)**

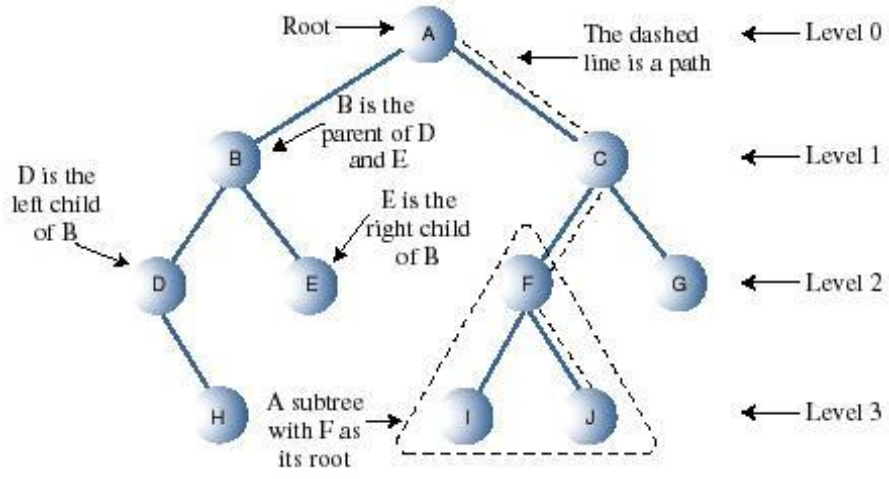


13

H, E, I, J, and G are leaf nodes

Árbol Binario: Términos

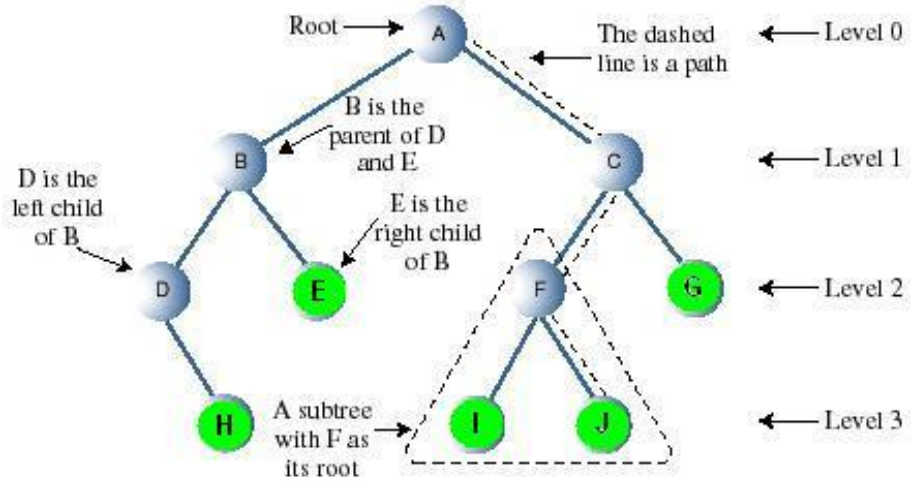
- **Hijo: Un nodo descendiente. (B es un hijo de A, C es un hijo de A, D es un hijo de B y A es un abuelo**



H, E, I, J, and G are leaf nodes

Árbol Binario: Términos

- **Hoja: Un nodo sin hijos**
  - **En este grafo: H, E, I, J, y G**

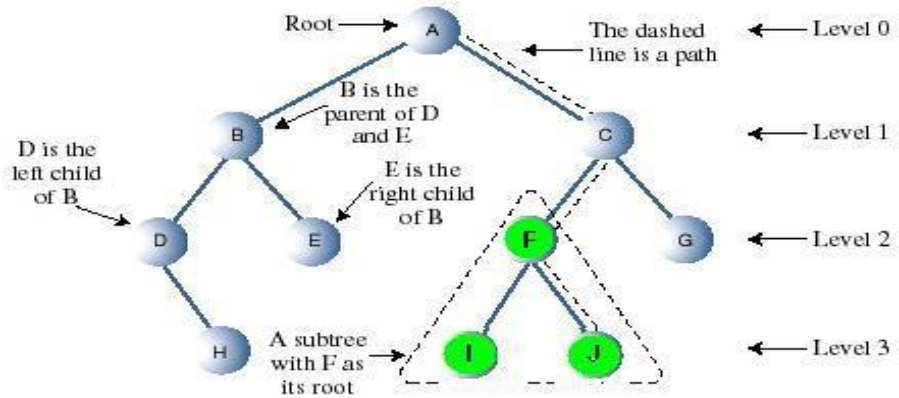


15

H, E, I, J, and G are leaf nodes

Árbol Binario: Términos

- **Subárbol: Los hijos de un nodo, los hijos de sus hijos, etc.**
  - **Un ejemplo es aquel que esta destacado pero, hay varios en este árbol**

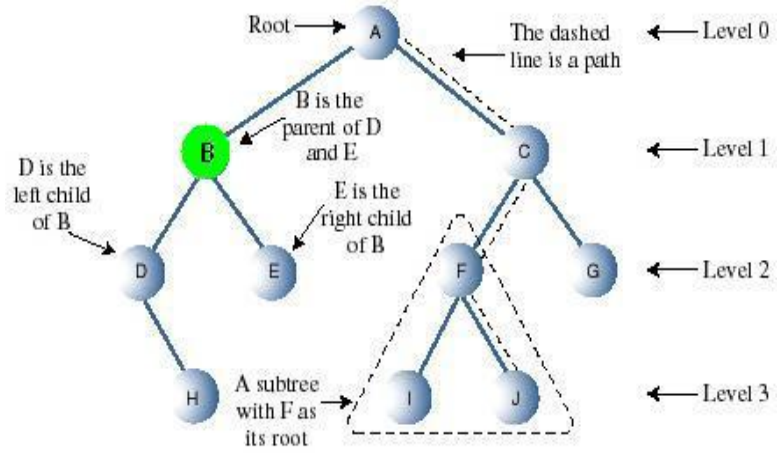


16

H, E, I, J, and G are leaf nodes

Árbol Binario: Términos

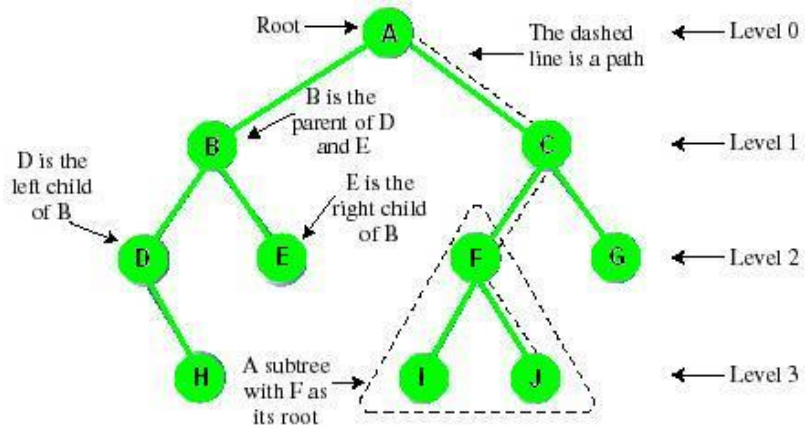
- **Visitar: Acceder un nodo, y hacer algo con sus datos.**
  - **Por ejemplo podemos visitar nodo B y chequea su valor**



17

Árbol Binario: Términos

- **Travesía: Visita todos los nodos en un orden especificado.**
  - **Un ejemplo: A, B, D, H, E, C, F, I, J, G**

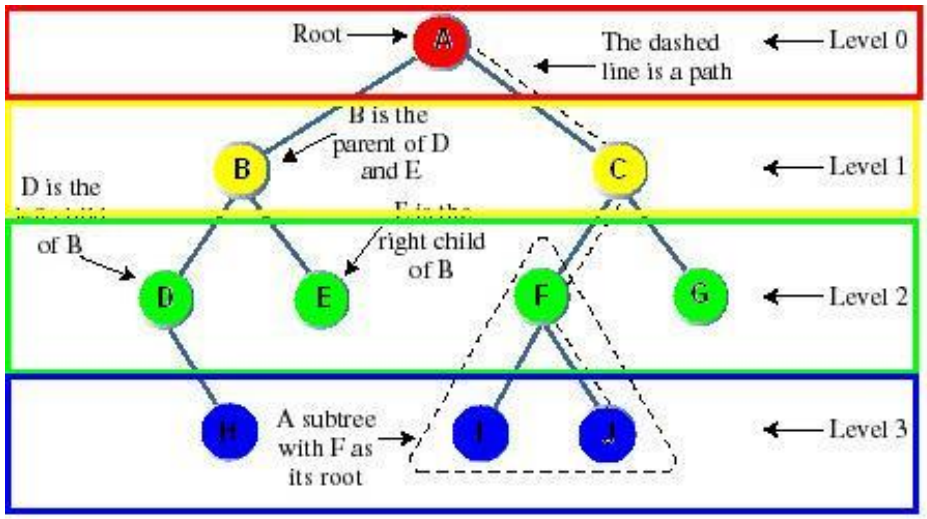


H, E, I, J, and G are leaf nodes

18

Árbol Binario: Términos

- **Niveles: Número de generaciones de un nodo desde la raíz**
  - **A esta en nivel 0, B y C están al nivel 1, D, E, F, G están al nivel 2, etc.**

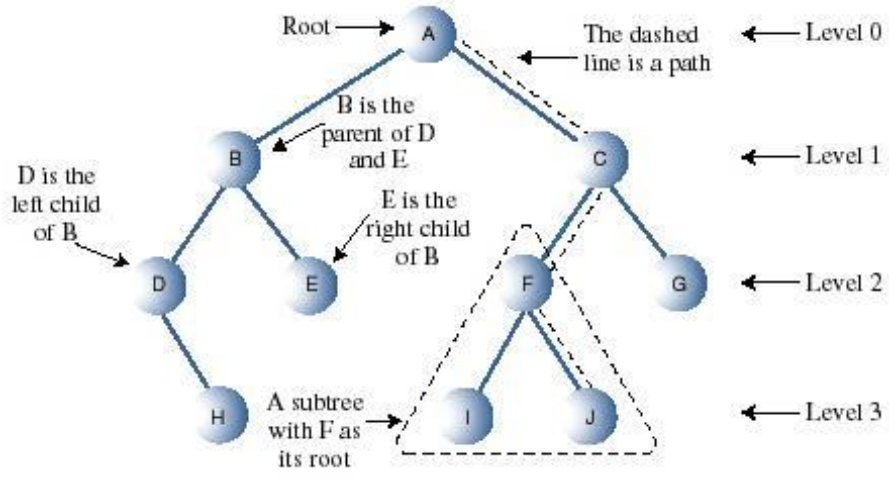


H, E, I, J, and G are leaf nodes

19

Árbol Binario: Términos

- **llave(Key): El contenido de un nodo**

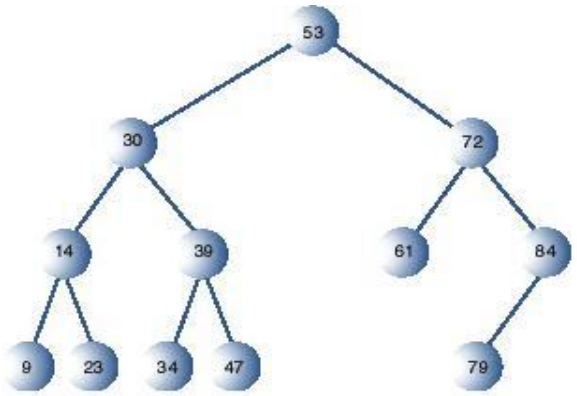


H, E, I, J, and G are leaf nodes

20

Un Árbol de Búsqueda Binaria

- **Es un árbol binario, con las siguientes características:**
  - **El hijo de la izquierda es siempre más pequeño que su padre**
  - **El hijo derecho es siempre más grande que su padre**
  - **Todos los nodos a la derecha son más grandes que todos los nodos a la izquierda**



21

Árbol de números enteros

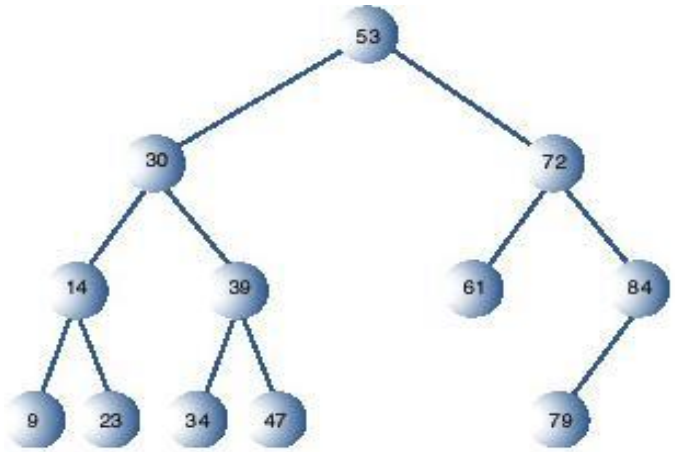
- **Usaremos esta clase para individualizar a los nodos**

```
class Node {
  public int data;
  public Node left;
  public Node right;
}
```

22

Encontrar un nodo

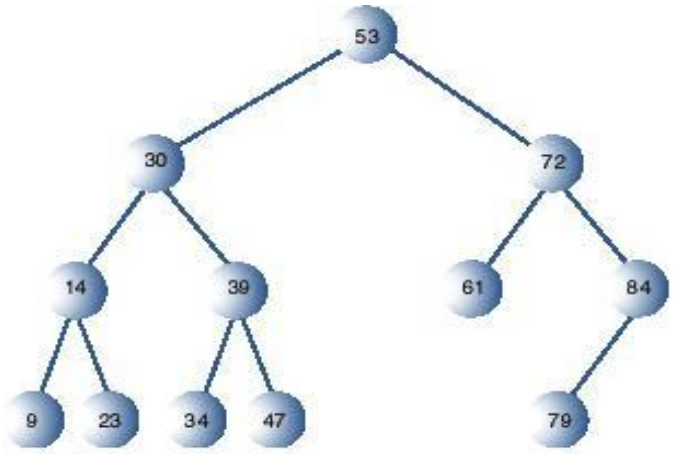
- ¿Cómo lo hacemos?
- Para todos los nodos:
  - Todos los elementos en el subárbol izquierdo son menores
  - Todos los elementos en el subárbol derecho son más grandes



23

Búsqueda por una LLAVE (Key)

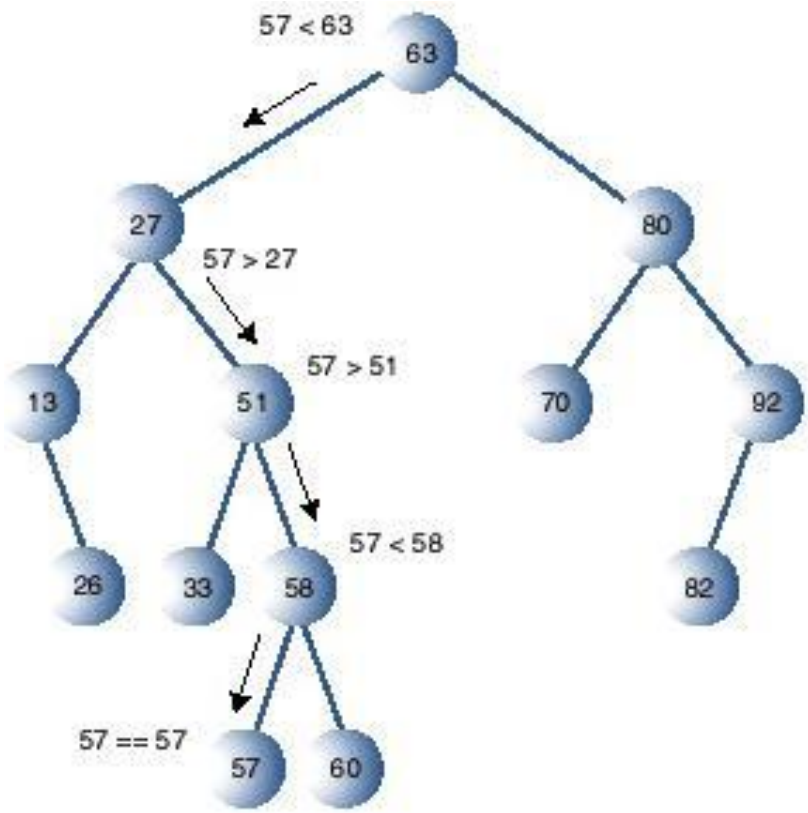
- **Comenzaremos en la raíz, y comprobaremos su valor**
  - **Si el valor = clave, estamos listos.**
- **Si el valor es mayor que la clave, vea su hijo izquierdo**
- **Si el valor es inferior a la clave, vea su hijo derecho**
- **Repita.**



24

Ejemplo

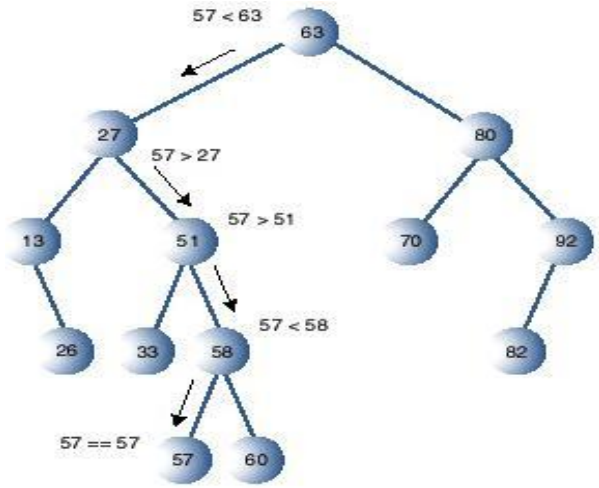
Buscar (Find) elemento 57



25

Número de operaciones: Find

- Típicamente es O(log n). ¿Por que?

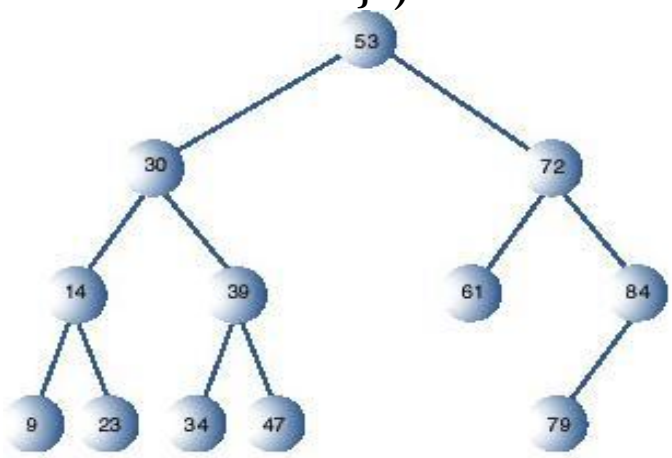


- ¿En que caso no lo será?
- ¿Cómo garantizamos O(log n)?

26

Insertando un nodo

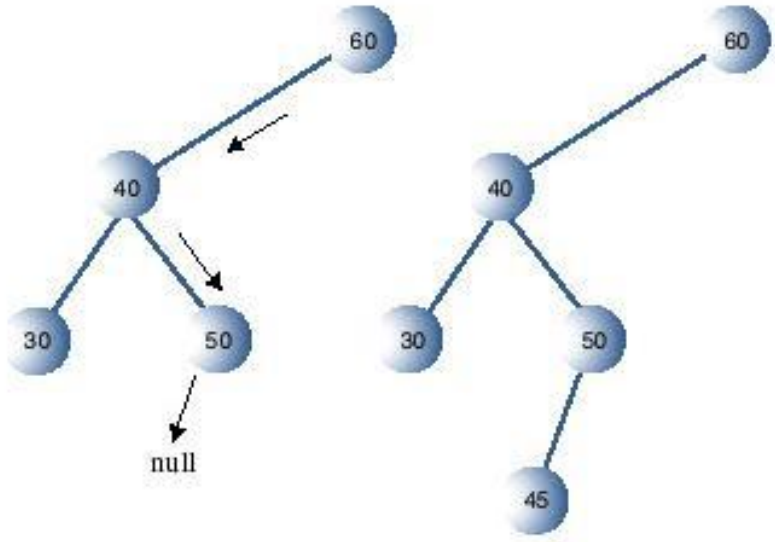
- ¿Qué debemos hacer?
- Encontrar un lugar para insertar un nodo (similar a buscar (find), excepto que recorremos todo el camino hasta que no encontremos más hijo)
- Ponlo allí



27

Ejemplo

Insertar elemento 45



a) Before insertion

b) After insertion

28

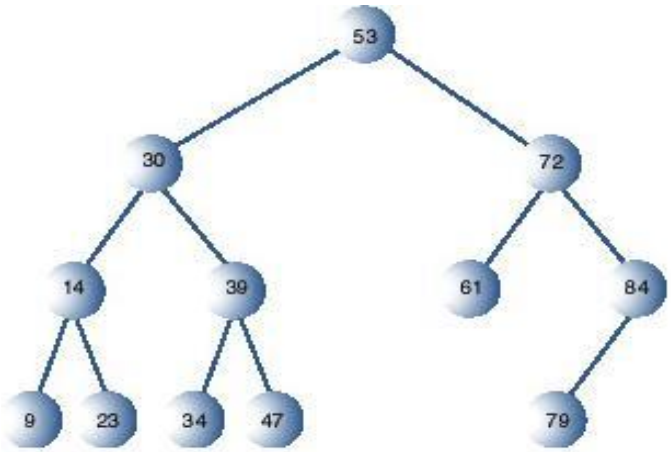
Atravesando un árbol

- **Hay tres formas:**
  - **In orden (más común)**
  - **Pre orden**
  - **Post orden**

29

Travesía In orden

- **Visite cada nodo del árbol en orden ascendente:**

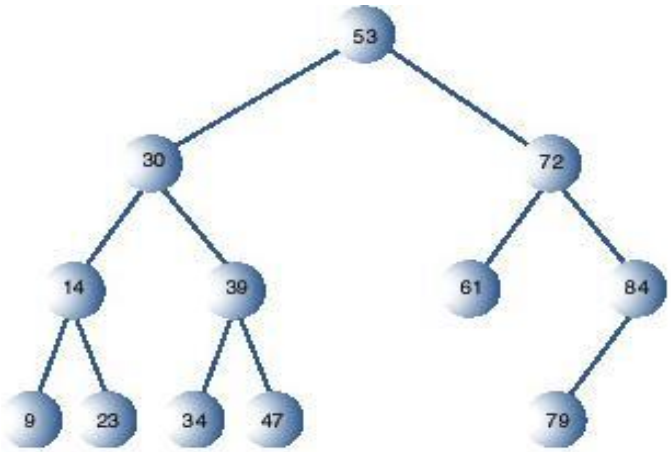


- **En este árbol, un recorrido en orden produce:**
  - 9 14 23 30 34 39 47 53 61 72 79 84

30

Travesía In orden

- **Orden ascendente**

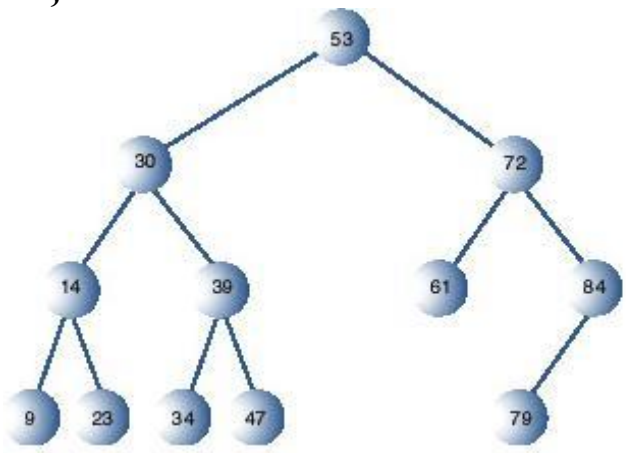


- **Implicación:Tenemos que imprimir hijo izquierdo del nodo antes que el propio nodo, luego imprimir el propio nodo antes que su hijo derecho**

31

Travesía In orden

- **Orden: Izquierdo, Raíz, Derecha**

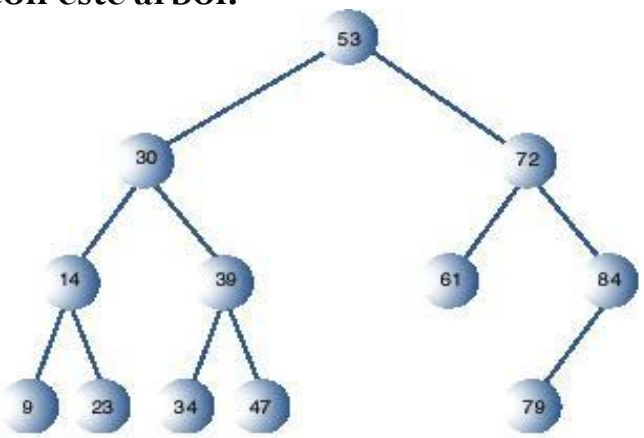


- **Podemos pensar esto de manera recursiva: empezar en la raíz, a recorrer en in orden el subárbol izquierdo, imprimir la raíz, recorrer en in orden el subárbol derecho**

32

Travesía en Pre orden

- **Imprime todos los padres antes de que sus hijos Imprime todos los hijos izquierdos antes que los hijos derechos. Así que con este árbol:**

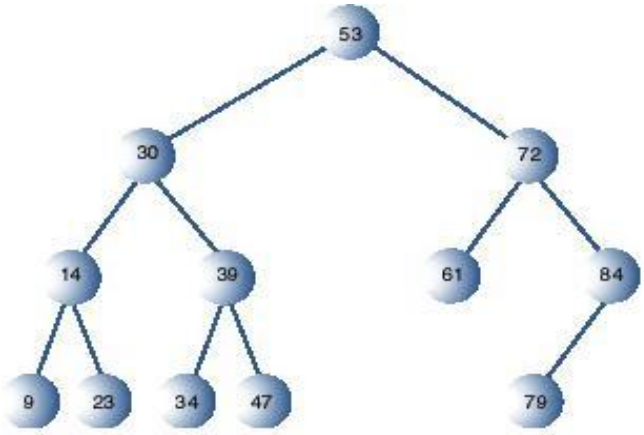


- **Una travesía en pre orden produce:**
  - **53 30 14 9 23 39 34 47 72 61 84 79**

33

Travesía en pre orden

- **Orden: Raíz, izquierdo, derecho**

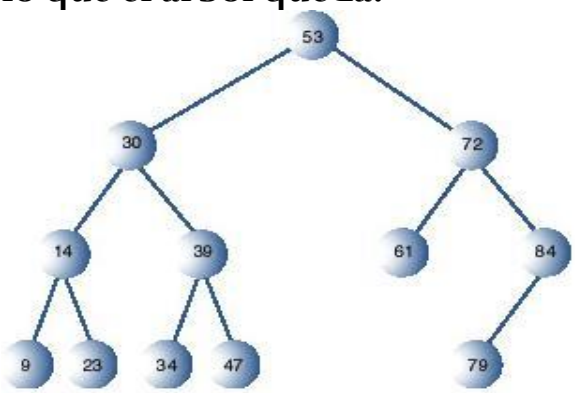


- **De nuevo podemos hacer esto de forma recursiva: imprimir la raíz, travesía en pre-orden del subárbol izquierdo, travesía en pre orden del subárbol derecho**

34

Travesía en post orden

- Imprime todos los hijos antes que los padres
- Imprime todos los hijos izquierdos antes que los hijos derechos. Por lo que el árbol queda:

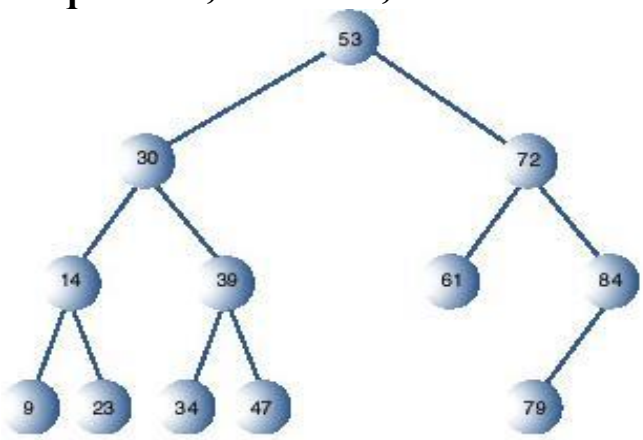


- Una travesía en post orden produce:
  - 9 23 14 34 47 39 30 61 79 84 72 53

35

Travesía en post orden

- **Orden: Izquierdo, derecho, raíz**

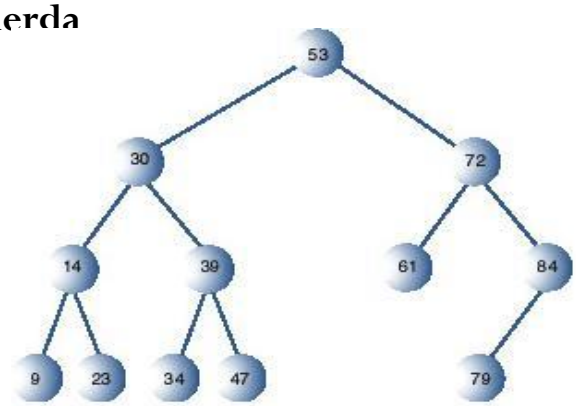


- **De nuevo podemos hacer esto de forma recursiva: post orden recorrido en post orden del subárbol izquierdo, recorrido en post orden del subárbol derecho, imprima la raíz**

36

Encontrar el mínimo

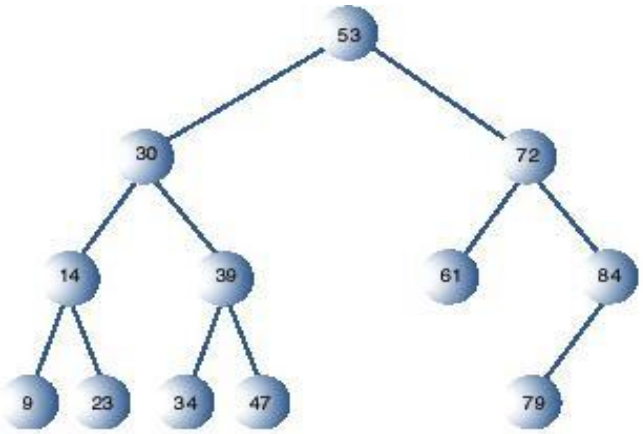
- **En un árbol de búsqueda binario, ¡este es siempre el hijo más a la izquierda del árbol! Fácil. ¿en Java?**
  - **Comience en la raíz, y recorra hasta que no encuentre un hijo a la izquierda**



37

Encontrar el Máximo

- **En un árbol de búsqueda binaria, esto también es fácil - es el hijo más a la derecha en el árbol**
  - **Comience en la raíz, recorra hasta que no haya más hijos derechos**
  - **¿Java?**



38

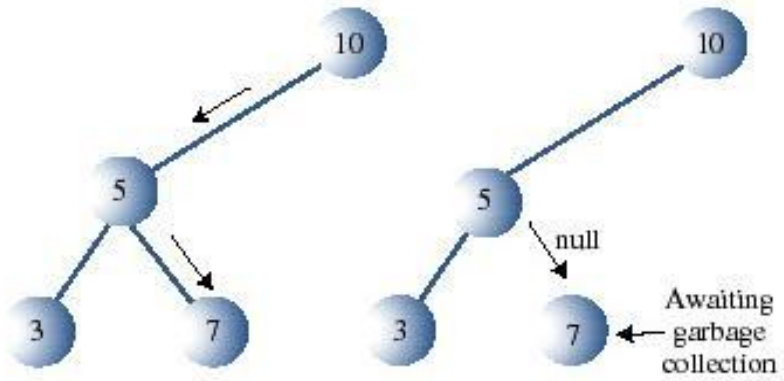
Eliminación

- Este es el reto
- En primer lugar, encontrar el elemento que deseamos eliminar
- Una vez que lo hayas encontrado, chequear uno de los tres casos:
  - 1. El nodo no tiene hijos (Fácil)
  - 2. El nodo tiene un hijo (decentemente Fácil)
  - 3. El nodo tiene dos hijos (Difícil)

39

Caso 1: Sin hijos

- **Para eliminar un nodo sin hijos:**
  - **Encontrar el nodo**
  - **Asigne a la referencia hijo del correspondiente padre con null**
  - **Ejemplo: Eliminar 7 del siguiente árbol:**



a) Before deletion b) After deletion

40

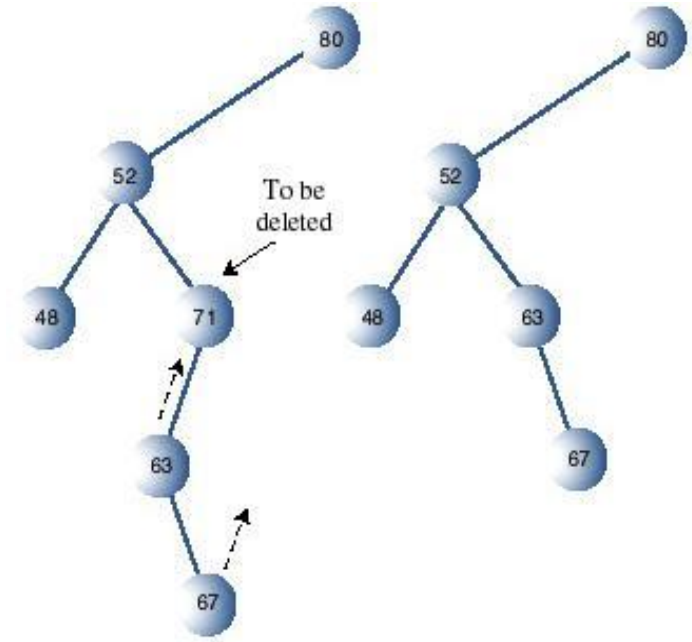
Implementación en Java

- **Primero, encuentre el nodo**
  - **A medida que avanzamos, no perdamos vista de:**
    - **El padre**
    - **Si el nodo es un hijo de la izquierda o la derecha de su padre**
- **Luego, verifique el caso cuando ambos hijo son null**
  - **Ajuste ya sea el hijo izquierdo o derecho del padre con null**
  - **A menos que sea la raíz, en cuyo caso el árbol está vacío**

41

Caso 2: Un sólo Hijo

- **Asigne el hijo del nodo eliminado como hijo de su padre**
  - **En esencia, un 'tijeretazo' al nodo eliminado de la secuencia**
- **Ejemplo, elimine 71 del siguiente árbol:**



42

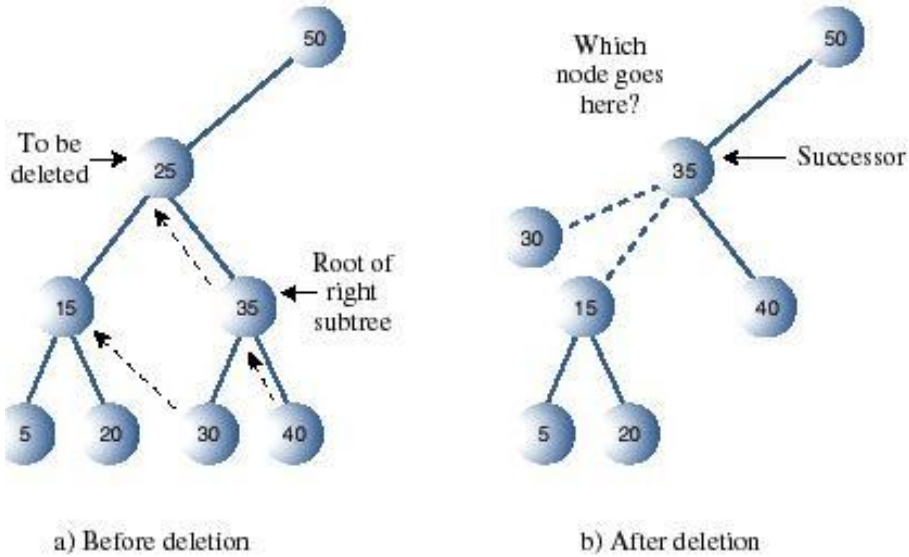
Implementación Java

- Dos casos a manejar. O bien:
  - El hijo derecho es null
    - Si el nodo es un hijo izquierdo, ajuste hijo izquierdo del padre como hijo izquierdo del nodo
    - Si el nodo es un hijo derecho, ajuste el hijo derecho del padre como hijo izquierdo del nodo
  - El hijo izquierdo es null
    - Si el nodo es un hijo izquierdo, ajuste hijo izquierdo de su padre como hijo derecho del nodo
    - Si el nodo es un hijo derecho, establezca el hijo derecho del padre como hijo derecho del nodo

43

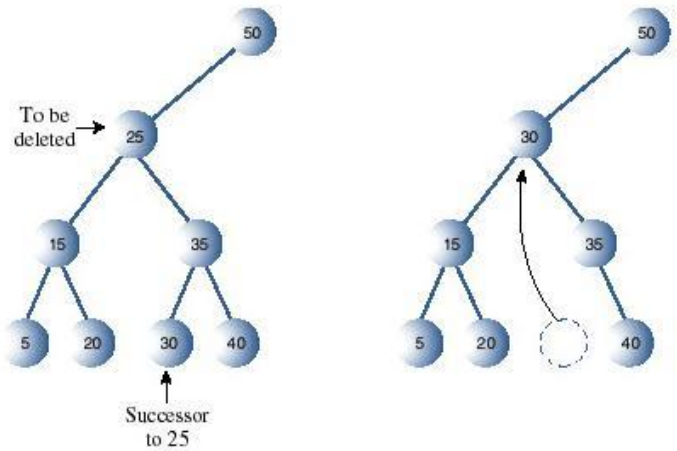
Caso 3: Dos hijos

- **Finalmente el caso difícil.**
- **Veamos un ejemplo del porque es complicado…**



Case 3: Dos hijos

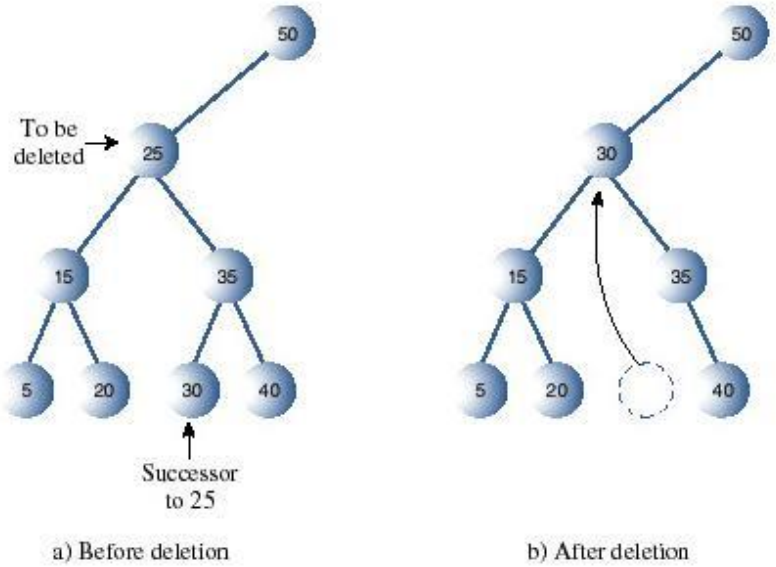
- **Lo que necesitamos es conocer el próximo nodo más alto que reemplace a 25.**
  - **Por ejemplo, si reemplazamos 25 con 30, estamos listos.**



45

Case 3: Dos hijos

- **Lo llamamos el sucesor in orden del nodo eliminado**
  - **Es decir, 30 es el sucesor in orden de 25. Este reemplaza a 25.**



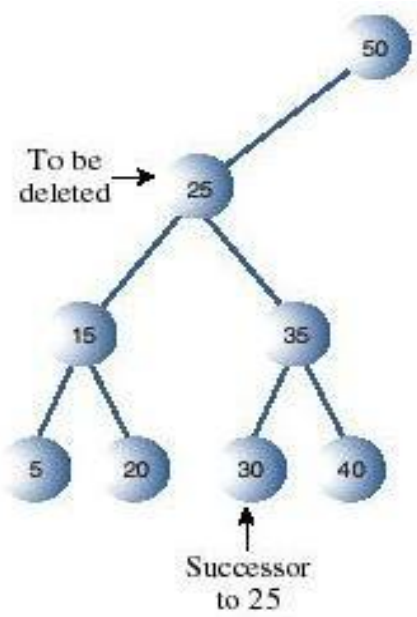
a) Before deletion

b) After deletion

46

Sucesor in orden

- **El sucesor in orden siempre va a ser el menor elemento del sub árbol derecho**
  - **En otras palabras, es el menor elemento que es más grande que el nodo eliminado.**

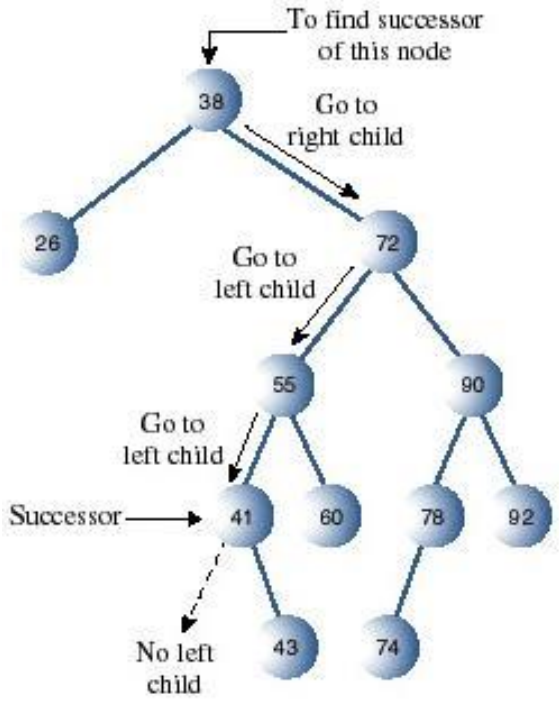


a) Before deletion

47

Encontrar el sucesor in orden

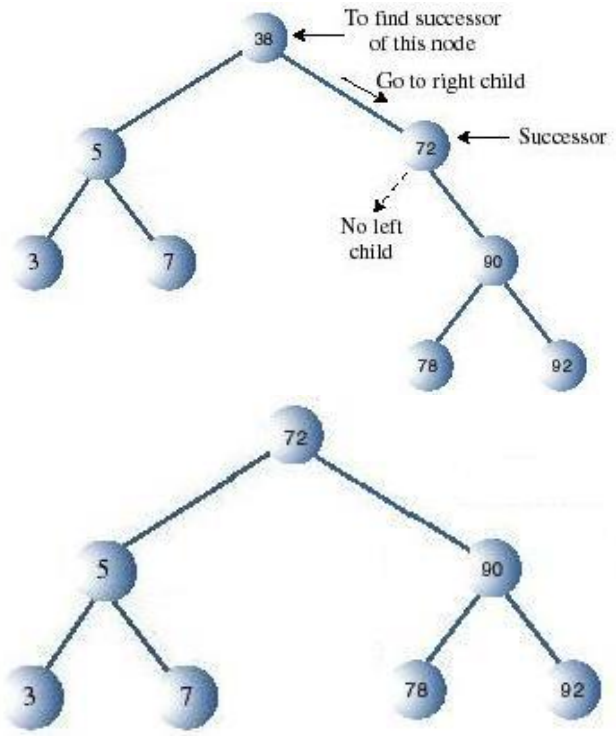
- El algoritmo para encontrar el sucesor in orden de algún nodo X:
  - Primero vaya al hijo derecho de X
  - Luego siga moviéndose hacia los hijos izquierdos
  - Hasta que no haya más
  - Entonces hemos encontrado el sucesor in orden
- Este es el que reemplazará a X



48

Remover el sucesor

- **Debemos eliminar el sucesor de su lugar actual, y colocarlo en el lugar del nodo eliminado**
- **Si el sucesor es hijo derecho del nodo eliminado:**
  - **Asigne al hijo izquierdo del sucesor con el hijo izquierdo del nodo eliminado**
  - **Reemplace el nodo eliminado por el sucesor**



49

Implementación en Java (Ayudantía)

- Función getSuccessor()
- Acepta un nodo
- Primero va el hijo derecho
- Luego va hacia el hijo izquierdo
  - Hace esto hasta que no encuentre más hijos izquierdos
- También remueve sucesor

51

Eficiencia: Arboles de Búsqueda Binaria

- **Note que:**
  - **Inserción, eliminación y búsqueda, todas requieren visitar los nodos del árbol hasta que encontremos:**
    - **La posición de inserción**
    - **El valor a eliminar**
    - **El valor que buscábamos**
- **Para cualquiera de estas, visitaremos no más del número de niveles en el árbol**
  - **Porque cada nodo que visitamos, revisamos su valor, y si no estamos listos, nos vamos a uno de sus hijos**

52

Eficiencia: Arboles de Búsqueda Binaria

- Hasta aquí un árbol de n nodos, cuantos niveles tenemos:
- Nodos Niveles
  - 1 1
  - 3 2
  - 7 3
  - 15 4
  - 31 5
  - ….
  - 1.073.741.824 30
- Es realmente log(n) + 1!

53

Entonces...

- Los tres algoritmos: inserción, eliminación, y búsqueda toma tiempo orden O(log n)
- Vamos a través de los log n + 1 niveles, cada vez con una comparación
- En el punto de inserción o eliminación, manipulamos un número constante de referencias (digamos, c)
  - Esto es independiente de n
- Por lo que el número de operaciones es log n + 1 + c, o O(log n)

54

Comparado con los Arreglos

- Considere 1 millón de elementos y borre un elemento en el medio
  - Arreglos -> caso promedio, 500 mil desplazamientos
  - Árbol de Búsqueda Binaria -> 20 o menos comparaciones
- Un caso similar es cuando se compara con inserción en un arreglo ordenado

- Lo que es lento en un árbol de búsqueda binaria es la travesía
  - Ir a través de los elementos del árbol
  - Pero en bases de datos grandes, esto probablemente nunca será necesario

55