



Estructuras de Datos

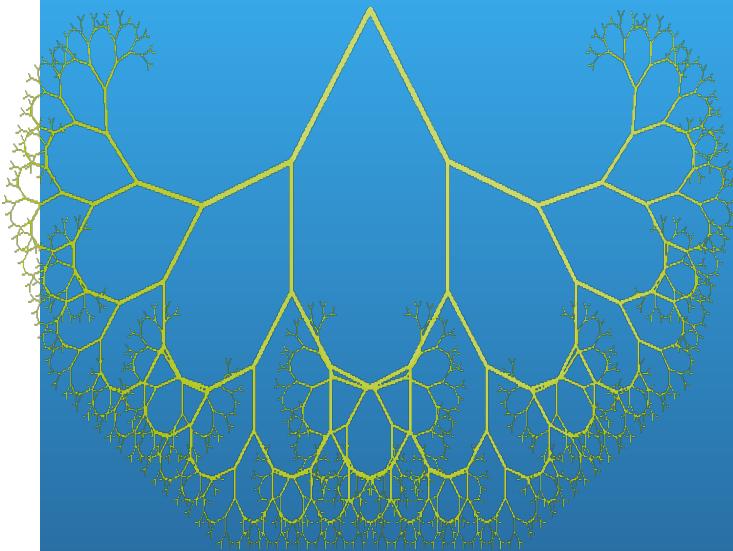
IIS262

Profesor: Patricio Galeas

CAPÍTULO 7 : Árboles Binarios



Introducción

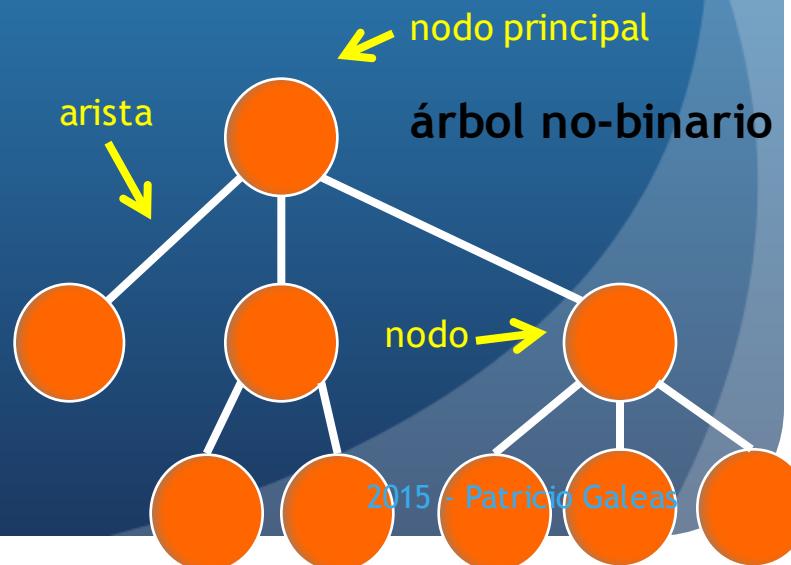


- Los árboles binarios son estructuras de almacenamiento **fundamentales** para la programación
- Combina las **ventajas** de un **arreglo ordenado** y de una **lista enlazada**:
 - Búsqueda rápida
 - Inserción y eliminación rápida

	Búsqueda	Inserción	Eliminación
Arreglo Ordenado	↑ O(LogN)	↓ O(N)	↓ O(N)
Lista Enlazada	↓ O(N)	↑ O(1)	↑ O(1)
Árbol Binario	↑	↑	2015 - Patricio Galeas

¿Qué es un Árbol ?

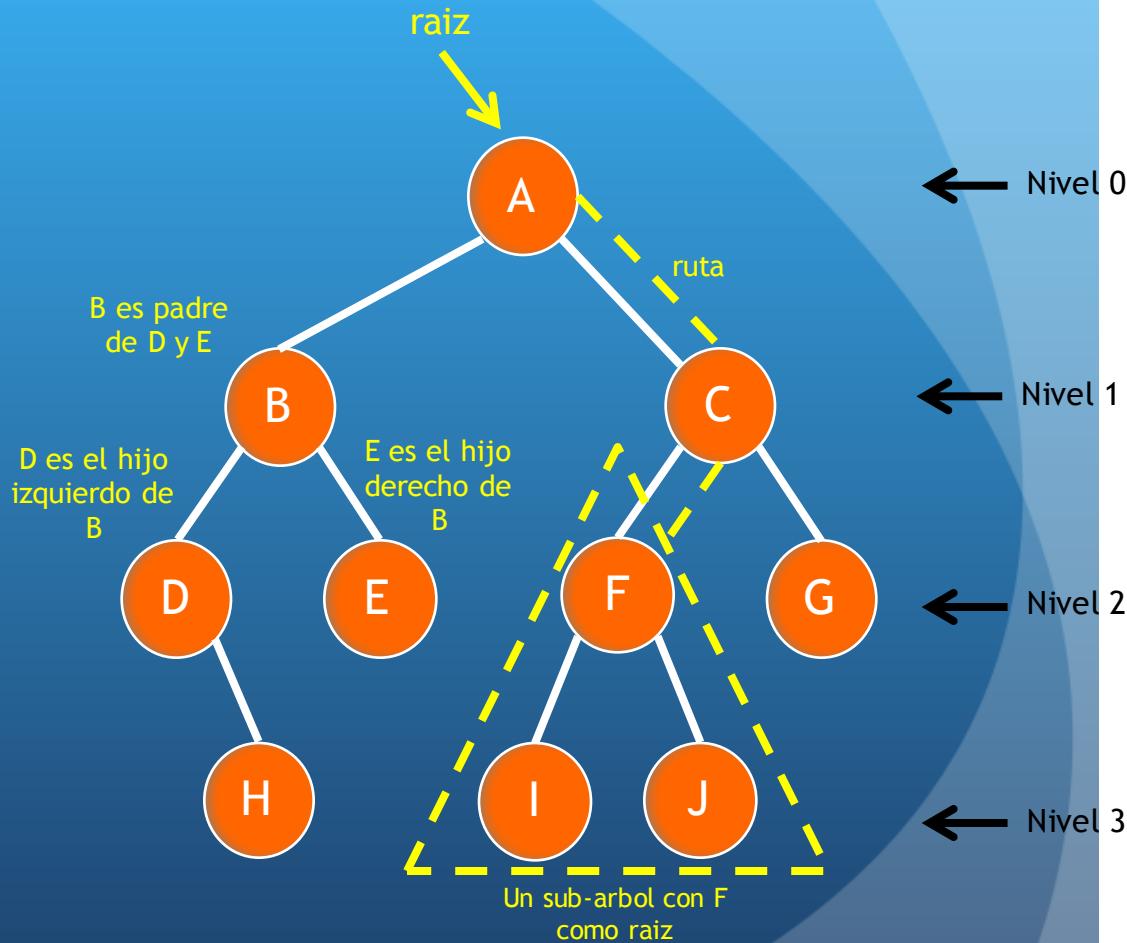
- Un **árbol** consiste en la conexión de **nodos** (nodes) a través de **aristas** (edges).
- Los árboles son **casos especiales** de una estructura más general llamada “**grafo**”, que revisaremos más adelante.
- Los **nodos** representan normalmente **entidades** (personas, autos, reservaciones, etc.)
- Las **aristas** representan la forma en que los **nodos** están **relacionados**.
- Las aristas son **referencias** y sirven para llegar de uno nodo a otro.
- Hay normalmente un **nodo principal** en la parte superior del árbol, el cual se conecta a otros **nodos** del nivel siguiente.
- Hay diferentes tipos de árboles: **binarios** y **múltiples**.



Árboles Binarios

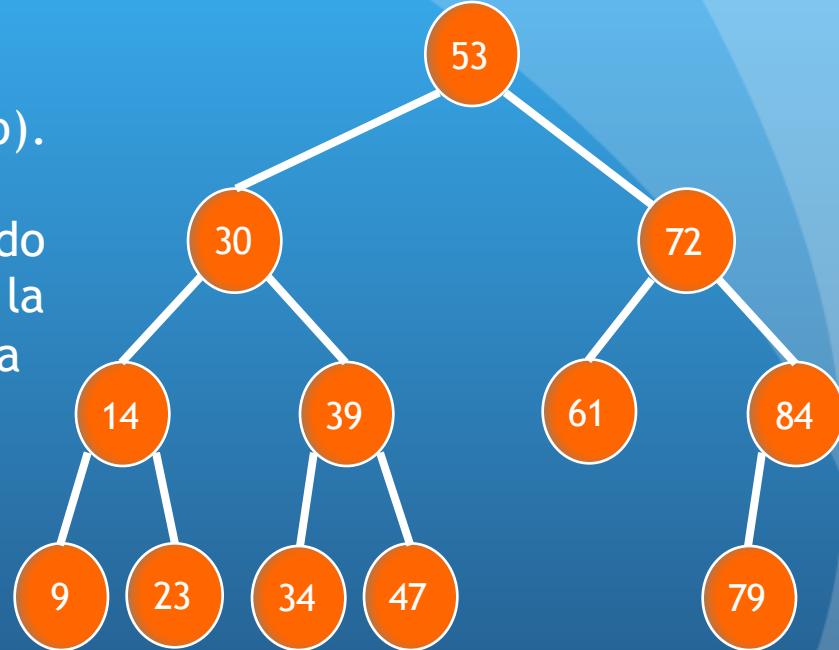
- **Ruta**: es el camino de un nodo a otro.
- **Raíz** : nodo (único) ubicado en la parte superior del árbol. Existe sólo un camino entre la raíz y cualquier otro nodo.
- **Padre**: es un nodo superior de otro nodo conectado por una arista.
- **Hijo** : es un nodo inferior a otro nodo conectado por una arista.
- **Hoja** : Es un nodo que no tiene hijos.
- **Sub-Árbol** : cualquier nodo puede considerarse como la raíz de un sub-árbol.
- **Visitar** : Un nodo es visitado cuando el control del programa llega al nodo, para ejecutar una operación. (sólo pasar por el nodo, no es visitar).
- **Atravesar** : Atravesar significa visitar todos los nodos en un cierto orden.
- **Niveles** : Es el número de generaciones desde la raíz.
- **Clave (Key)** : Es el valor de un nodo que permite buscar y ejecutar operaciones.

Terminología

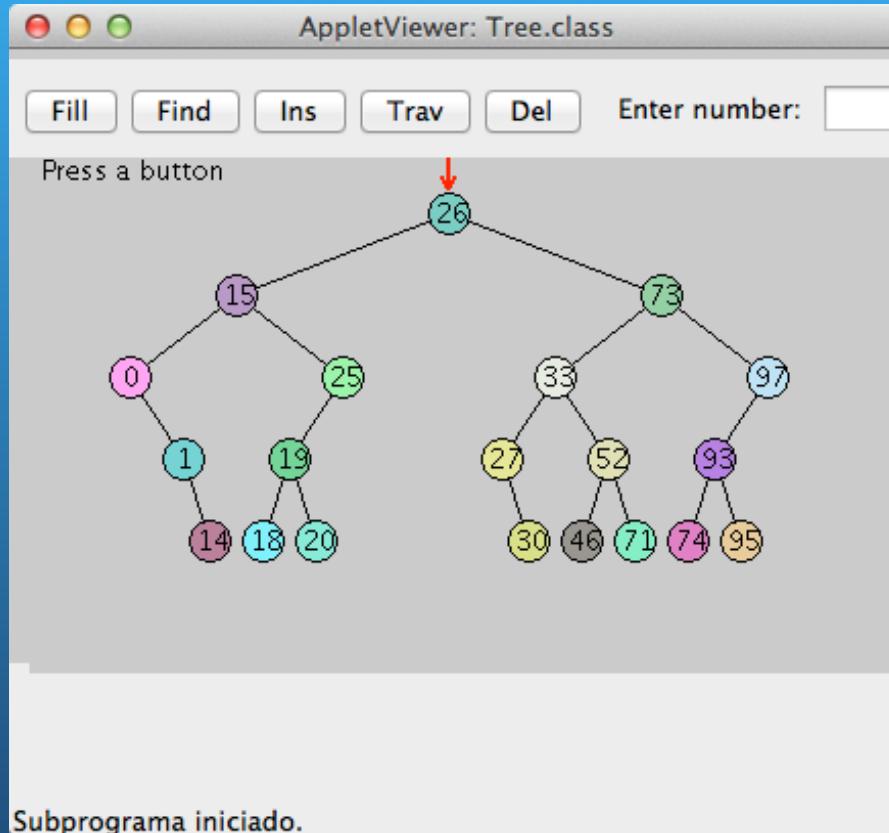


Árboles Binarios

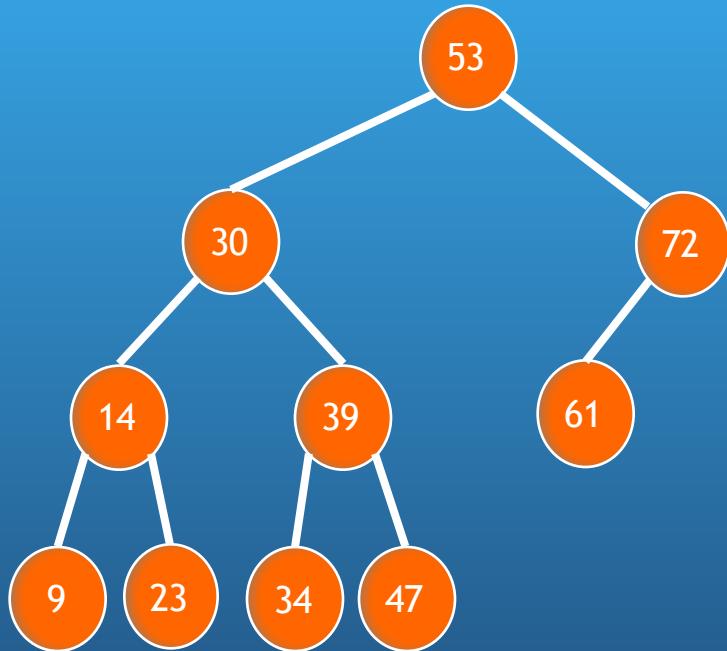
- Cada nodo puede tener **máximo dos nodos hijo** (hijo izquierdo, hijo derecho).
- En un **Árbol de Búsqueda Binario**: el nodo padre es siempre **mayor que su hijo de la izquierda y menor o igual a su hijo de la derecha**.



Applet - Árboles Binarios



Árboles Desbalanceados



- Algunos árboles son generados en forma desbalanceada: tienen la mayor parte de sus nodos en un de los lados de la raíz.
- El desbalanceo se origina a través del orden en que son insertados los elementos.
 - Aleatorio : más o menos balanceado.
 - Ascendente : hacia la derecha.
 - Descendente: hacia la izquierda.
- Los árboles desbalanceados presentan algunos problemas de eficiencia

Representando un árbol en Java

- Existen varias alternativas para representar un árbol en la memoria del computador.
- La más común es almacenar los nodos como posiciones independientes en memoria y luego conectarlas usando referencias.

Representando un árbol en Java

```
class Node
{
    int iData;
    double fData;
    node leftChild;
    node rightChild;
}
```

```
class Tree
{
    private Node root;

    public void find(int key)
    {
    }

    public void insert(int id, double dd)
    {
    }

    public void delete(int id)
    {
    }

    // various other methods
} // end class Tree
```

- Primero necesitamos una clase para los Nodos.
- Node contiene los datos que representan los objetos que estamos almacenando (empleados, repuestos, etc.)
- También incluye las referencias a sus dos nodos hijos.
- También necesitamos una clase para el árbol.
- Tree contiene todos los nodos.
- Tree contiene una sola variable, el nodo raíz (root).
- No necesita campos para otros nodos, ya que estos son accedidos a través de la raíz.

Representando un árbol en Java

```
class TreeApp
{
    public static void main(String[] args)
    {
        Tree theTree = new Tree;

        theTree.insert(50, 1.5);
        theTree.insert(25, 1.7);
        theTree.insert(75, 1.9);

        node found = theTree.find(25);
        if(found != null)

            System.out.println("Found the node with key 25");
        else
            System.out.println("Could not find node with key 25");
    }
}
```

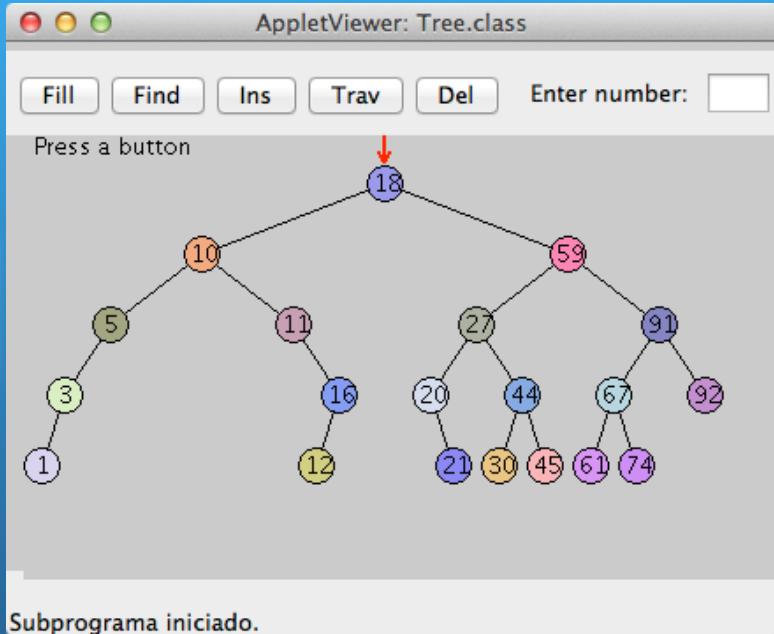
- Finalmente necesitamos una clase para realizar operaciones sobre el árbol.
- Esta rutina **Main()**
 - Crea un árbol
 - Inserta tres nodos
 - Busca uno de ellos

Revisemos estas operaciones básicas con el Applet ...

Simulación de un árbol



applet



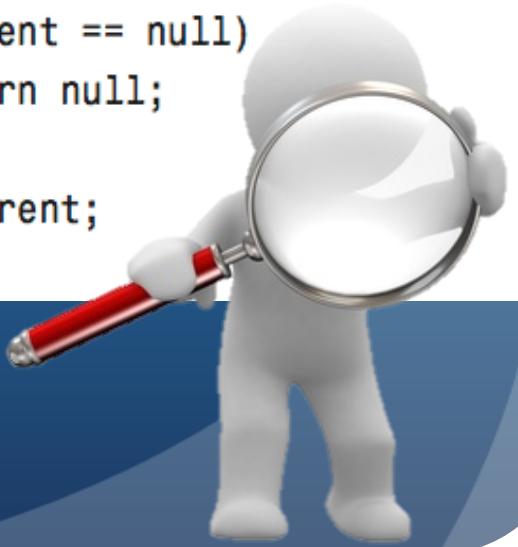
- Busquemos un nodo ubicado en las hojas.
- ¿Cómo opera el algoritmo?

```
public Node find(int key)
{
    Node current = root;
    while(current.iData != key)
    {
        if(key < current.iData)
            current = current.leftChild;
        else
            current = current.rightChild;
        if(current == null)
            return null;
    }
    return current;
}
```

Encontrando un Nodo

- La variable “**current**” contiene el nodo actualmente examinado.
- “**key**” es el valor buscado.
- La rutina **parte** en el nodo “**root**”.
- Compara “**key**” con “**current.iData**”
 - **key < current.iData** : va por la izquierda
 - **key > current.iData** : va por la derecha.
 - Si no encuentra nada, retorna **Null**.

```
public Node find(int key)
{
    Node current = root;
    while(current.iData != key)
    {
        if(key < current.iData)
            current = current.leftChild;
        else
            current = current.rightChild;
        if(current == null)
            return null;
    }
    return current;
}
```



Insertando un Nodo

- La idea es **buscar el padre del nodo** que queremos insertar, y luego conectarlo, según su valor a su izquierda o derecha.
- Parte creando un **nuevo nodo** con los datos de los **argumentos**.
- Luego utiliza código **similar** al de **find()**, para encontrar la posición del **nodo nuevo** e insertarlo.
- Se usa “**parent**” como referencia para hacer la conexión, ya que “**current**” queda en **Null** cuando se encuentra la posición final del nuevo nodo.

```
public void insert(int id, double dd)
{
    Node newNode = new Node();      // make new node
    newNode.iData = id;           // insert data
    newNode.dData = dd;
    if(root==null)                // no node in root
        root = newNode;
    else                           // root occupied
    {
        Node current = root;     // start at root
        Node parent;
        while(true)              // (exits internally)
        {
            parent = current;
            if(id < current.iData) // go left?
            {
                current = current.leftChild;
                if(current == null) // if end of the line,
                {
                    parent.leftChild = newNode; // insert on left
                    return;
                }
            } // end if go left
            else                  // or go right?
            {
                current = current.rightChild;
                if(current == null) // if end of the line
                {
                    parent.rightChild = newNode; // insert on right
                    return;
                }
            } // end else go right
        } // end while
    } // end else not root
} // end insert()
```

Recorriendo el Árbol

- El recorrer (**traversing**) árboles significa visitar los nodos de un árbol en un orden determinado.
- Esta operación **no es tan común** como: `find()`, `insert()` o `delete()` y además es más **lenta**.
- Recorrer un árbol **es teóricamente interesante** y muy útil en algunas circunstancias.
- Hay 3 formas de recorrer un árbol:
 - Pre-Orden
 - In-Orden
 - Post-Orden
- La forma más común en árboles binarios es **In-Orden**



Recorriendo el Árbol In-Orden

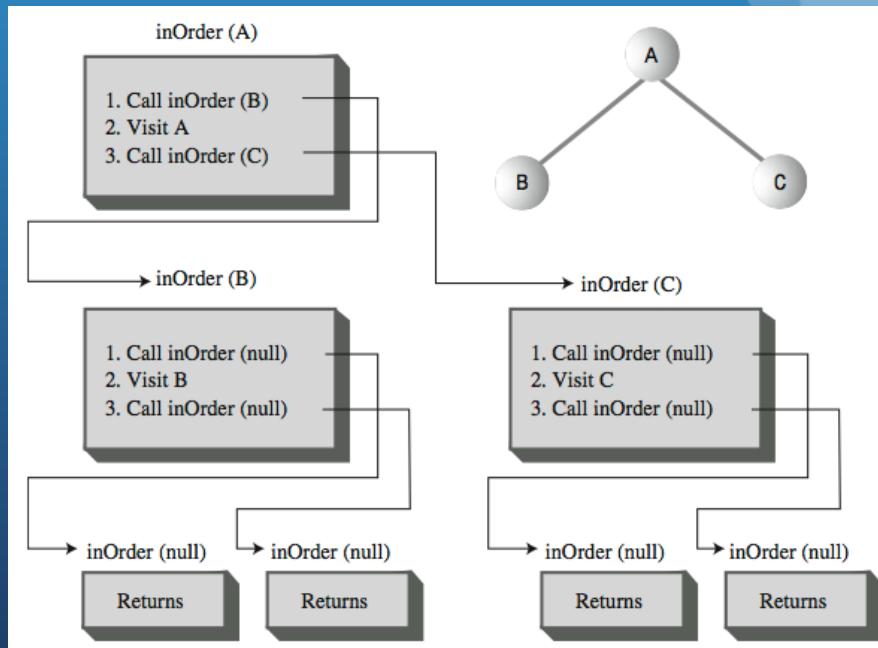
- Si es un árbol de búsqueda binario. los nodos son visitados en orden ascendente.
- La manera más fácil es recursivamente. La cual parte en el nodo raíz.
- El método sólo hace tres cosas:
 - Se llama a si mismo para recorrer el sub-árbol izquierdo.
 - Visita el nodo.
 - Se llama a si mismo para recorrer el sub-árbol derecho.
- Parte con el nodo raíz, y se llama recursivamente hasta que no queden más nodos por recorrer.
- Revisemos “Trav” en el Applet



```
private void inOrder(node localRoot)
{
    if(localRoot != null)
    {
        inOrder(localRoot.leftChild);

        System.out.print(localRoot.iData + " ");

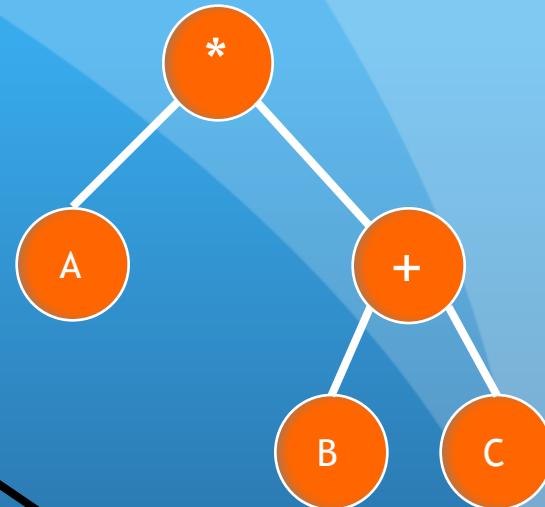
        inOrder(localRoot.rightChild);
    }
}
```



Recorriendo el Árbol

Recorridos Pre- y Post-Orden

- Este tipo de recorrido es útil para analizar expresiones algebraicas.
- Un árbol binario (no necesariamente de búsqueda) sirve para representar expresiones algebraicas.
- Recorriendo el árbol en In-Orden generará la expresión en Infix correcta. Sólo hay que agregar los paréntesis.
- Para Pre- y Post-orden se usan los mismo pasos que en In-Orden, sólo la secuencia es diferente.
- Para Pre-Orden:
 - Visitar el nodo.
 - Se llama a si mismo para recorrer el sub-árbol izquierdo.
 - Se llama a si mismo para recorrer el sub-árbol derecho.
- Para Post-Orden:
 - Se llama a si mismo para recorrer el sub-árbol izquierdo.
 - Se llama a si mismo para recorrer el sub-árbol derecho.
 - Visitar el nodo.



Infix : $A^*(B+C)$

Prefix : $*A+B+C$

Postfix : $ABC+*$

Valores máximos y mínimos

- Para encontrar el mínimo, se parte de la raíz y se elige siempre el hijo izquierdo hasta el nodo que no tenga hijo izquierdo. Este último nodo es el mínimo.
- Para encontrar el máximo, el procedimiento es equivalente: current = current.rightChild;
- El cálculo del valor mínimo es parte del procedimiento necesario para eliminar un nodo.

```
public Node minimum()
{
    Node current, last;
    current = root;
    while(current != null)
    {
        last = current;
        current = current.leftChild;
    }
    return last;
}
```



Eliminando un nodo

- Eliminar un nodo **es la operación más compleja** en árboles de búsqueda binarios.
- Se parte buscando el nodo que se desea eliminar: `find()`.
- Una vez encontrado el nodo **hay tres casos** a considerar:
 - El nodo a eliminar **es una hoja**.
 - El nodo a eliminar **tiene un hijo**.
 - El nodo al eliminar **tiene dos hijos**.
- El primer caso es **fácil**, el segundo es **relativamente fácil**, pero el tercero es **complejo**.



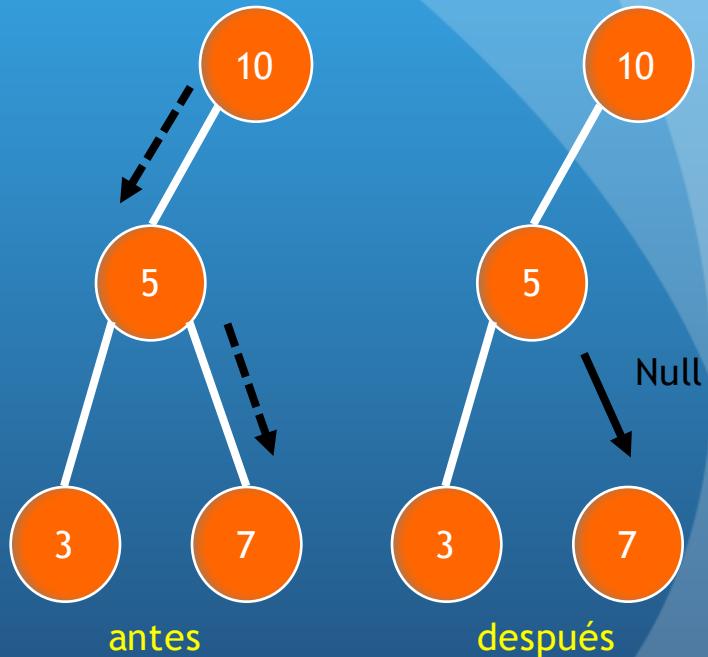
Eliminando un nodo

CASO 1 : El nodo a eliminar es una hoja

- Una vez encontrado el nodo a eliminar, simplemente se cambia en el padre el valor del hijo a correspondiente a Null.
- El nodo seguirá existiendo, pero no será más parte del árbol. Y será removido automáticamente de la memoria (garbage collection).
- Revisemos como funciona en el applet.



applet



Eliminando un nodo

CASO 1 : El nodo a eliminar es una hoja

- La primera parte del código para eliminar es similar a `find()` e `insert()`.

```
public boolean delete(int key) // delete node with given key
{
    Node current = root;                                // (assumes non-empty list)
    Node parent = root;
    boolean isLeftChild = true;

    while(current.iData != key)                         // search for node
    {
        parent = current;
        if(key < current.iData)                         // go left?
        {
            isLeftChild = true;
            current = current.leftChild;
        }
        else                                         // or go right?
        {
            isLeftChild = false;
            current = current.rightChild;
        }
        if(current == null)                            // end of the line,
            return false;                             // didn't find it
    } // end while
    // found node to delete
    // continues...
}
```

Eliminando un nodo

CASO 1 : El nodo a eliminar es una hoja

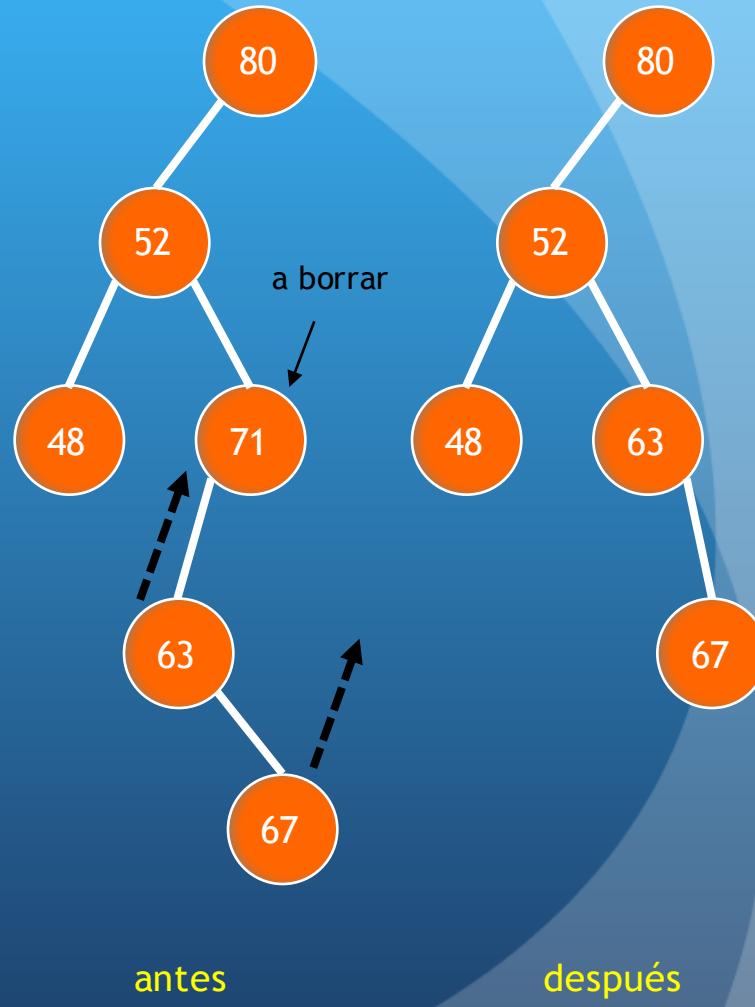
- Despues de encontrar al nodo ...
- Hay que verificar el caso de que no tenga hijos.
- Si no tiene hijos, se verifica el caso especial de la raíz.

```
// delete() continued...
// if no children, simply delete it
if(current.leftChild==null &&
   current.rightChild==null)
{
    if(current == root)                  // if root,
        root = null;                   // tree is empty
    else if(isLeftChild)
        parent.leftChild = null;       // disconnect
    else                                // from parent
        parent.rightChild = null;
}
// continues...
```

Eliminando un nodo

CASO 2 : El nodo a eliminar tiene un hijo

- Este caso tampoco es muy complicado
- El nodo tiene sólo dos conexiones:
 - Una con su padre
 - Una con su único hijo
- Luego se “extrae” al nodo conectando a “su padre” directo con “su hijo”.
- Revisemos como funciona en el applet.



applet

Eliminando un nodo

CASO 2 : El nodo a eliminar tiene un hijo

- Esta situación se maneja con 4 variaciones:
 - El hijo del nodo a ser borrado puede ser izquierdo o derecho
 - El nodo a ser borrado, puede ser hijo izquierdo o derecho de su padre.
- También incluye el caso especial en que el nodo a eliminar es el raíz.
- Gracias a las referencias, el mover un sub-árbol a otra posición es relativamente simple.

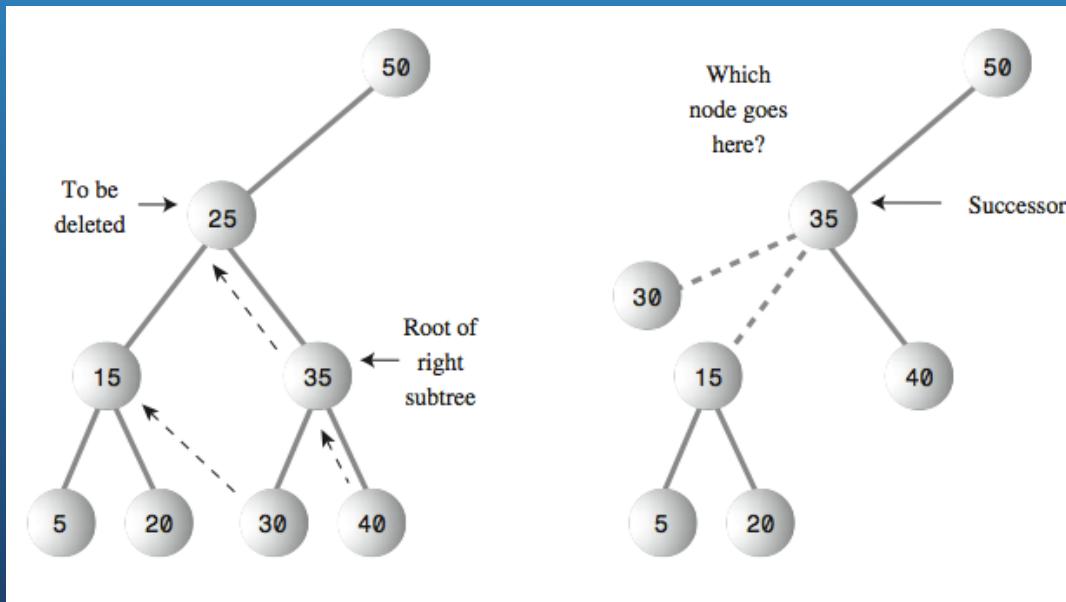
```
// delete() continued...
// if no right child, replace with left subtree
else if(current.rightChild==null)
    if(current == root)
        root = current.leftChild;
    else if(isLeftChild)          // left child of parent
        parent.leftChild = current.leftChild;
    else                      // right child of parent
        parent.rightChild = current.leftChild;

// if no left child, replace with right subtree
else if(current.leftChild==null)
    if(current == root)
        root = current.rightChild;
    else if(isLeftChild)          // left child of parent
        parent.leftChild = current.rightChild;
    else                      // right child of parent
        parent.rightChild = current.rightChild;
// continued...
```

Eliminando un nodo

CASO 3 : El nodo a eliminar tiene dos hijos

- ¿Por que no simplemente reemplazar por uno de sus hijos?
- Por ejemplo: reemplazar por el hijo derecho.
- ¿Que hacemos entonces con el nodo 30 ? ... no nos sirve el reemplazo !



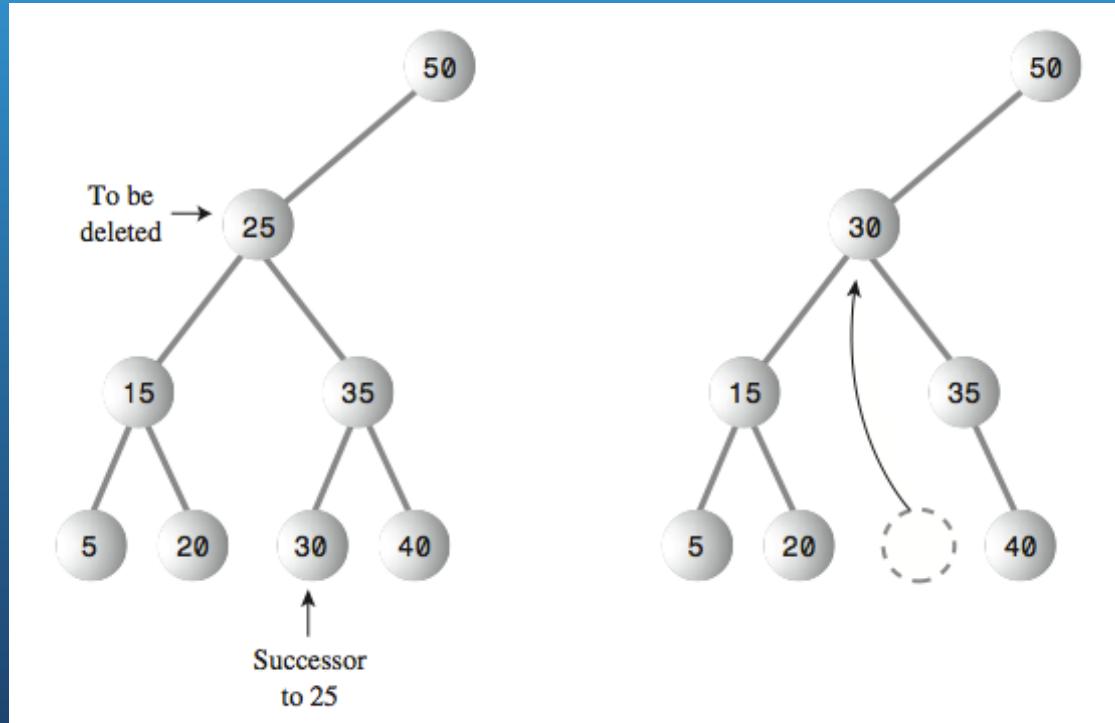
- **La buena noticia:**
 - Existe un truco para resolver el problema.
- **La mala noticia:**
 - Este truco involucra una serie de **casos especiales**.

Eliminando un nodo

CASO 3 : El nodo a eliminar tiene dos hijos

- **TRUCO:**

Para eliminar a un nodo con dos hijos, reemplazar el nodo con su sucesor (in-orden).



- Los nodos permanecen en orden.
- El reemplazo es simple en el caso de que el sucesor no tenga hijos.
- ¿Cómo encontrar al sucesor algorítmicamente?

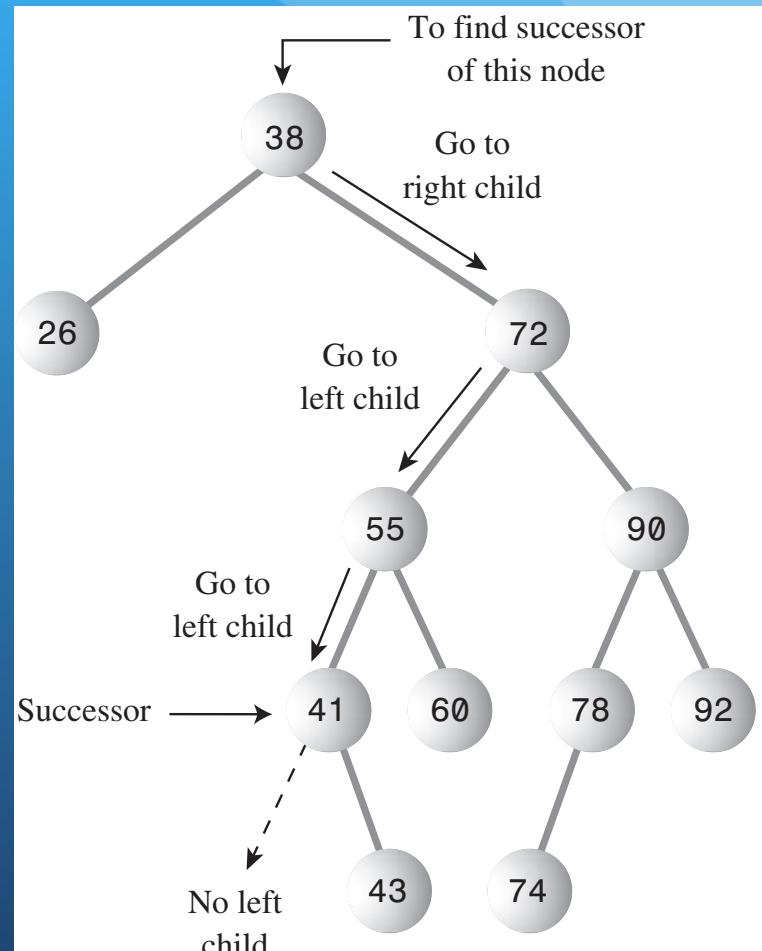
Eliminando un nodo

CASO 3 : El nodo a eliminar tiene dos hijos

Para encontrar el sucesor algorítmicamente:

Ir al nodo hijo derecho, y luego elegir siempre al nodo de la izquierda, hasta llegar al último nodo de la izquierda, el cual es su sucesor.

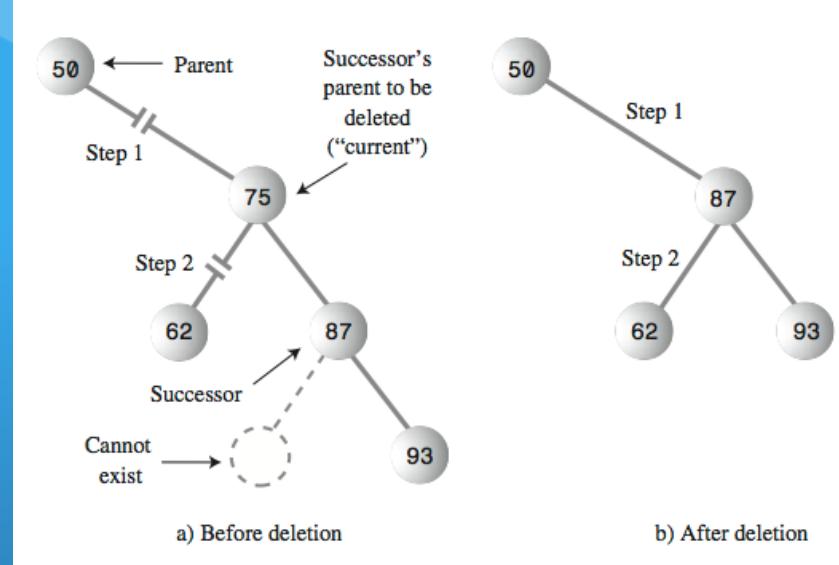
En realidad, estamos buscando al menor de los nodos mayores que el nodo a eliminar.



Eliminando un nodo

CASO 3 : El nodo a eliminar tiene dos hijos

- Si el “sucesor” es el hijo derecho del nodo a borrar. Es decir, el hijo derecho del nodo a borrar no tiene hijos izquierdos.
- Entonces, habría que mover el subárbol donde el sucesor es raíz, a la posición del nodo eliminado.
- Esta operación requiere 2 pasos:
 1. Desconectar de su padre el nodo a borrar (current), y apuntar esta conexión del padre directamente con el sucesor.
 2. Desconectar el hijo izquierdo del nodo a borrar y conectar este hijo como hijo izquierdo del sucesor.



```

// delete() continued
else // two children, so replace with inorder successor
{
    // get successor of node to delete (current)
    Node successor = getSuccessor(current);

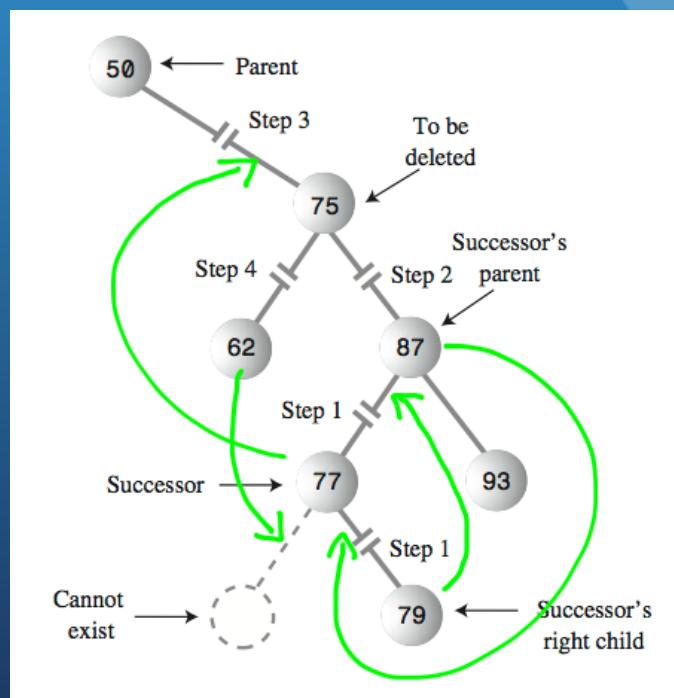
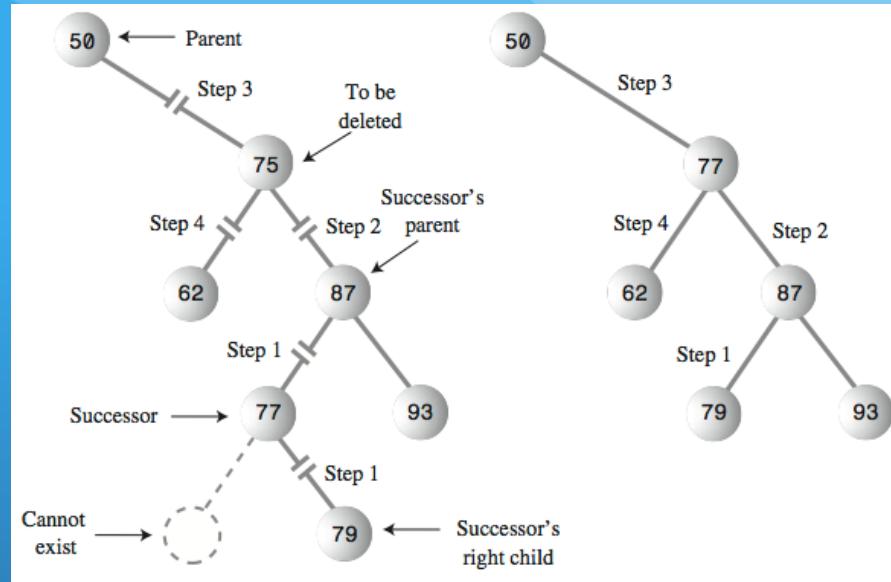
    // connect parent of current to successor instead
    if(current == root)
        root = successor;
    else if(isLeftChild)
        parent.leftChild = successor;
    else
        parent.rightChild = successor;
    // connect successor to current's left child
    successor.leftChild = current.leftChild;
} // end else two children
// (successor cannot have a left child)

return true;
} // end delete()
  
```

Eliminando un nodo

CASO 3 : El nodo a eliminar tiene dos hijos

- Si el “sucesor” es un descendiente izquierdo del hijo derecho del nodo a eliminar. Es decir, el hijo derecho del nodo a eliminar tiene hijos izquierdos.
- Esta operación requiere 4 pasos:
 1. Conectar el hijo derecho del sucesor como hijo izquierdo del padre del sucesor.
 2. Conectar el hijo derecho del nodo a eliminar como el hijo derecho del sucesor.
 3. Desconectar el nodo a eliminar de la posición de hijo derecho de su padre y colocar al sucesor en su posición.
 4. Desconectar el hijo izquierdo del nodo a eliminar y conectarlo como hijo izquierdo de sucesor.



Eliminando un nodo

- No es trivial !
- Algunos programadores simplemente agregan una variable booleana a la clase Nodo para definir si el no fue eliminado.
- ¿Cuál podría ser la desventaja?



Eficiencia en Árboles Binarios

- La mayoría de las operaciones en árboles se relacionan con descender el árbol de nivel en nivel hasta un nodo en particular.
- En un árbol lleno la mitad de los nodos están en el último nivel. Es decir la mitad de las operaciones (búsquedas, inserciones, eliminaciones, etc.) se llevan a cabo en este nivel inferior.
- Durante la búsqueda (por ejemplo) es necesario visitar un nodo por cada nivel. Entonces podemos asociar el tiempo de ejecución al número de niveles del árbol.



Número de Nodos	Número de Niveles
1	1
3	2
7	3
15	4
31	5
1.023	10
32.767	15
1.048.575	20
33.554.432	25

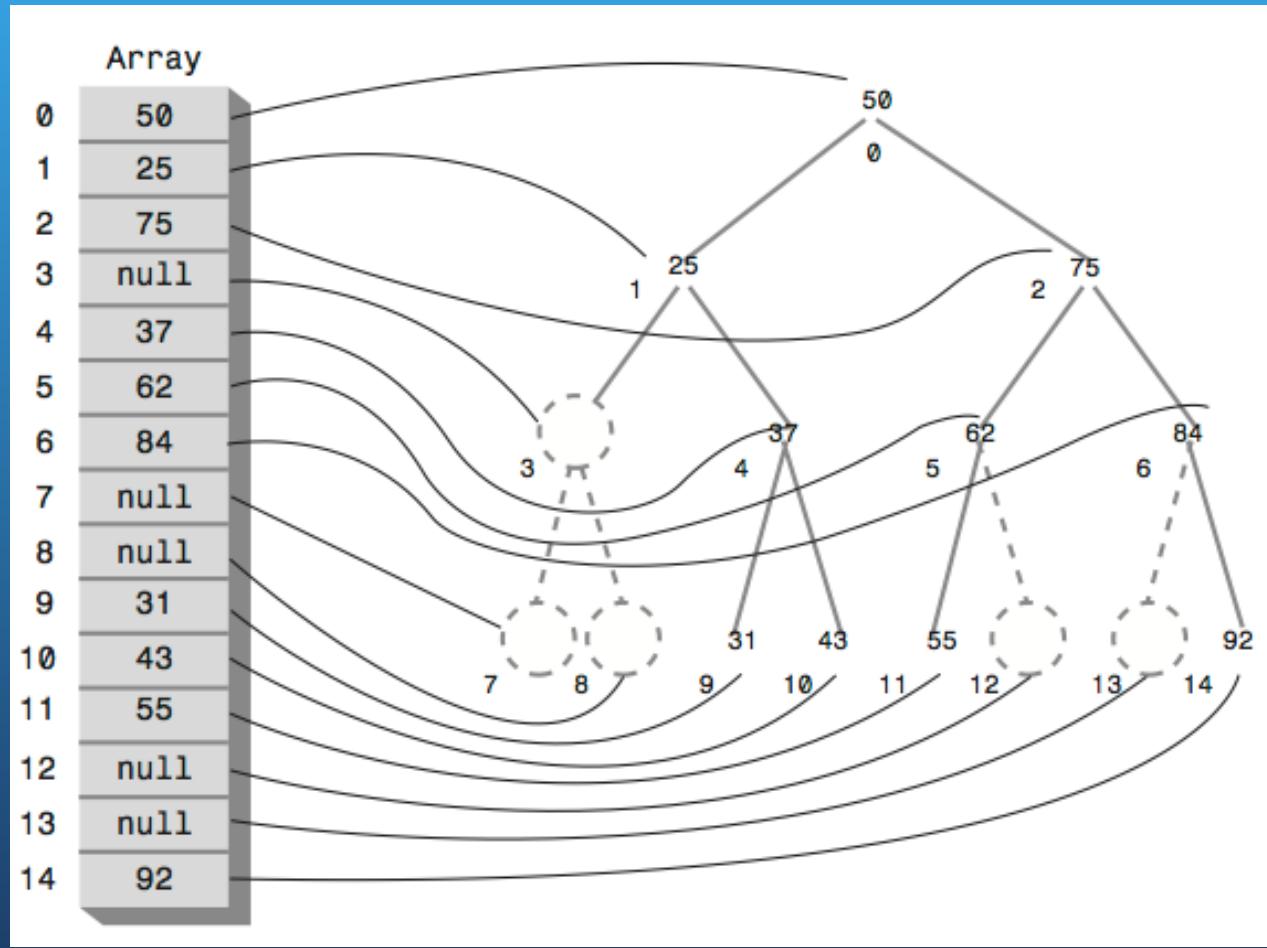
$$N = 2^L - 1$$

$$N + 1 = 2^L$$

$$L = \log_2(N + 1)$$

Es decir, el orden para ejecutar las operaciones básicas en un árbol binario es : $O(\log N)$

Árboles representados como Arreglos



- No es muy eficiente. Los nodos vacíos o eliminados gastan memoria.
- Pero si las eliminaciones no están permitidas, puede ser una estructura a considerar.

Valores duplicados

- Al igual que en otras estructuras, el **manejo de valores duplicados** debe ser considerada.
- Por ejemplo en el Applet, el ingreso de un nodo con un valor duplicado es insertado a la **derecha** de su gemelo.



- **find()**, debe ser modificado para chequear ítems adicionales.
- Si quiero **prohibir** valores duplicados, debo modificar el método **insert()**.

Resumen



- Los árboles son nodos conectados a través de aristas.
- El raíz, es el nodo superior del árbol y no tiene padres.
- En un árbol de búsqueda binario, todos los nodos que son descendientes izquierdos de un nodo A, tienen valor menores que A; y todos los nodos que son descendientes derechos de A tienen valores mayores que A.
- Los árboles ejecutan búsquedas, inserciones y eliminaciones en tiempo $O(\log N)$.
- Los nodos representan los objetos de información que están almacenados en el árbol.
- Las aristas son representadas comúnmente en un programa como referencias a los hijos de un nodo (y a veces a su padre).
- Recorrer un árbol significa visitar todos los nodos en un orden determinado.
- Los recorridos más simples son: pre-orden, in-orden, y post-orden.
- Un árbol desbalanceado es aquel donde la raíz tiene muchos mas descendientes al lado izquierdo que el derecho, o viceversa.
- Buscar un nodo involucra comparar el valor buscado con los valores de cada nodo, yendo por su hijo izquierdo si el valor es menor y por el lado derecho si el valor es mayor.

Resumen



- La inserción involucra primero encontrar el lugar donde insertar el nuevo nodo, y luego cambiar sus referencias a los hijos.
- En el recorrido in-orden los valores se visitan en orden ascendente.
- Pre-Orden y Post-Orden son útiles para evaluar expresiones algebraicas.
- Cuando un nodo no tiene hijos, este puede ser eliminado simplemente definiendo la conexión con su padre como Null.
- Cuando un nodo tiene 1 hijo, puede ser eliminado conectando la referencia del hijo de su padre con su hijo.
- Cuando un nodo tiene 2 hijos, reemplazando a este con su sucesor.
- El sucesor de un nodo A puede ser encontrado buscando el mínimo nodo en el sub-árbol del hijo derecho de A.
- En la eliminación de un nodo con 2 hijos, afloran varias situaciones distintas, dependiendo si el sucesor es el hijo derecho del nodo a ser eliminado o uno de los descendientes de su hijo izquierdo.
- Los árboles también pueden ser representados en la memoria del computador como arreglos.
- Los nodos con valores duplicados pueden causar algunos problemas en arreglos, por que sólo el primer elemento puede ser encontrado

Experimentos

- Use el applet del árbol binario y cree 20 árboles
 - . ¿qué porcentaje de estos árboles es desbalanceado?
- Cree un diagrama de actividades UML de las distintas posibilidades cuando se elimina un nodo en un árbol binario.
Esto debe considerar los tres casos, con ambos hijos (derecho e izquierdo) y el caso especial de la eliminación de la raíz.
- Utilice el applet para verificar el borrado de nodos en todas las casos posibles descritos.

