Segunda parte de árboles

mat-151

Argumentos y constructores por copia

Argumentos y constructores por copia

```
#include <iostream>
using namespace std;

class A{
public:
    int * ptr;    float b;

    A(void){
        cout << "\nSe dispara constuctor\n";
        b = 0;
    }
    A(A & original){
        cout << "\nSe dispara constuctor por copia\n";
        b = original.b;
    }
    void operator=(A &right){
        cout << "\nSe dispara = 1 \n";
        ptr = right.ptr;
        b = right.b;
    }
};
```

```
void unaFunc(A unA){

    cout << "\n Entra a funcion\n";
    unA.b = 5;
}

int main(int argc, char**argv) {
    A uno, dos;

    uno = dos;
    unaFunc(uno);
    cout << endl << uno.b << endl ;
    return 0;
}
```

Recorrido de árboles

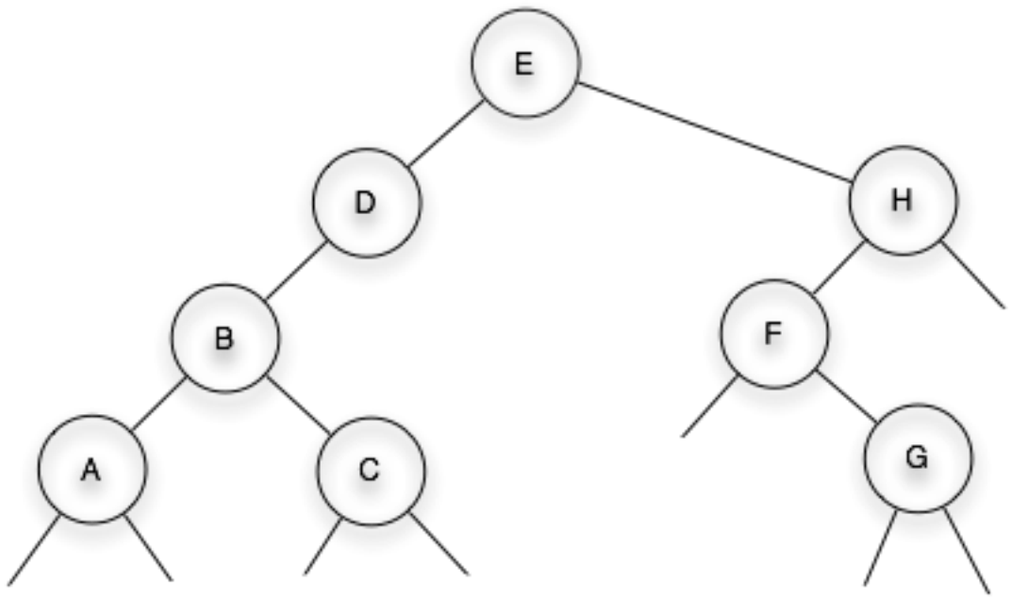
- Dado un apuntador a un árbol, queremos procesar cada nodo en el árbol de forma sistemática.
- En una lista ligada, nos movemos de un nodo a otro siguiendo el *link único*
- en un árbol hay decisiones que hacer ya que se pueden seguir links múltiples.

Recorrido de árboles

- Para una lista ligada tenemos dos operaciones básicas:
  - procesar un nodo y
  - seguir un link
- Si primero procesamos el nodo y luego seguimos el link se dice que la lista es visitada en **pre-orden.**
- Si primero seguimos el link y luego procesamos el nodo, se dice que la lista se recorre en **post-orden.**

Recorrido de árboles

- Hay esencialmente dos métodos para visitar sistemáticamente todos los nodos de un árbol: ***depth-first*** y ***breadth-first***.
- Algunos métodos ***depth-first*** ocurren frecuentemente y tienen nombres propios: ***recorrido en pre-orden***, ***recorrido en orden***, ***recorrido en post-orden***.



$$T = \{\ E\ ,\ \underline{\{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}}\ ,\ \underline{\{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \}}\ \}$$

Recorrido de árboles

- En un árbol tenemos tres ordenes básicos para recorrerlo:
  - ***Pre-orden***
  - ***Orden (Solo para árbol binario)***
  - ***Post-orden***

Recorrido de árboles: pre-orden

- Para un árbol general:

1. Visitar la raíz,

2. hacer un recorrido en pre-orden de todos los sub-árboles de la raíz uno por uno en el orden dado.

- Para un árbol binario:

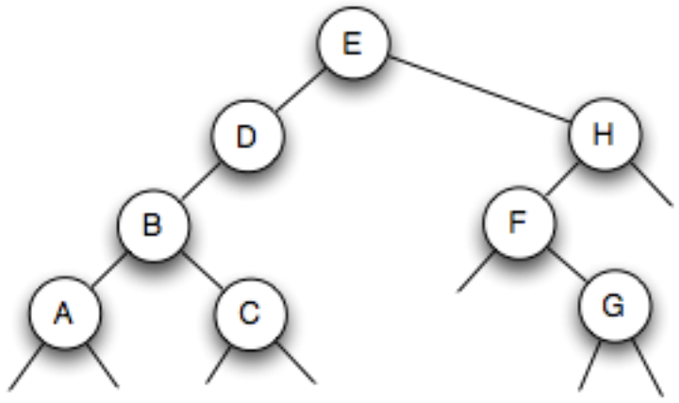
1. Visitar la raíz,

2. recorrer el sub-árbol izquierdo

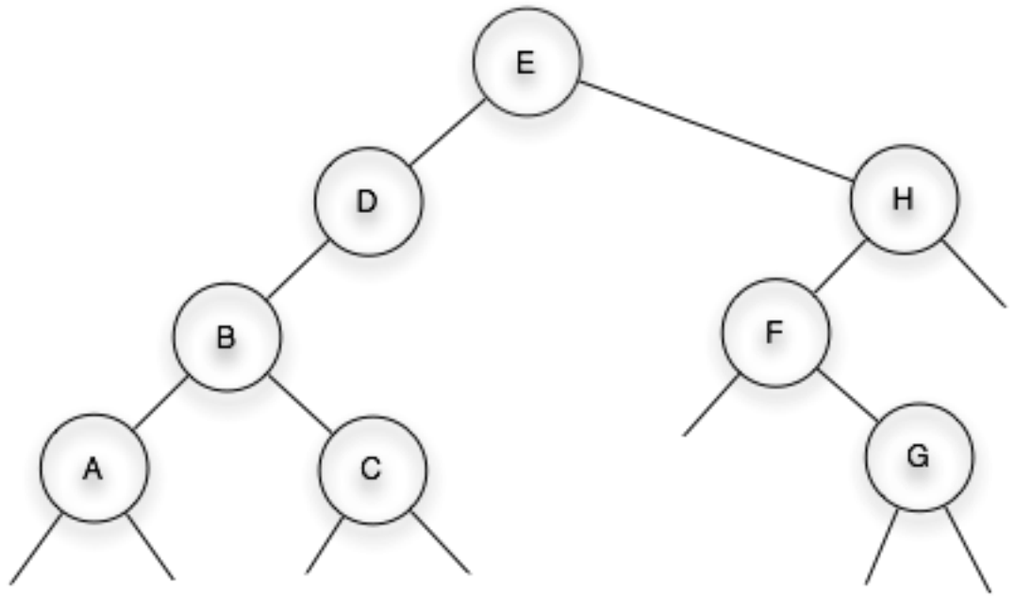
3. recorrer el sub-árbol derecho

Recorrido de árboles: pre-orden

```
void Tree::preorder( void(*process)(Item& item))
{
    process(m_data);
    start();
    while( is_Valid())
    {
        currentChild->preorder(process);
        forth();
    }
}
```

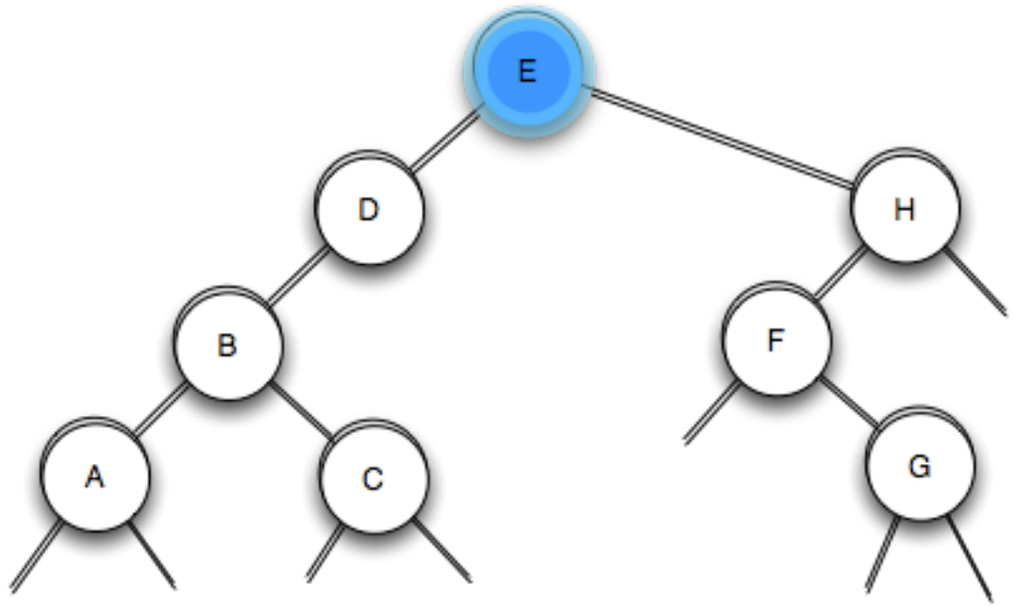


Recorrido de árboles binarios: pre-orden



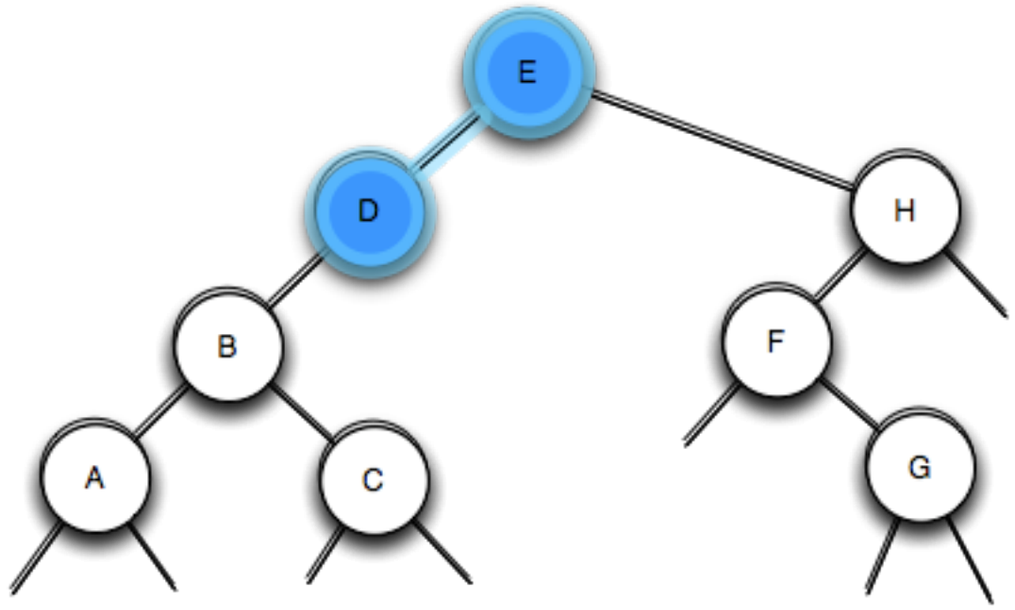
$$T = \{\ E\ ,\ \{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}\ ,\ \{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \ \}\ \ \}$$

Recorrido de árboles binarios: pre-orden



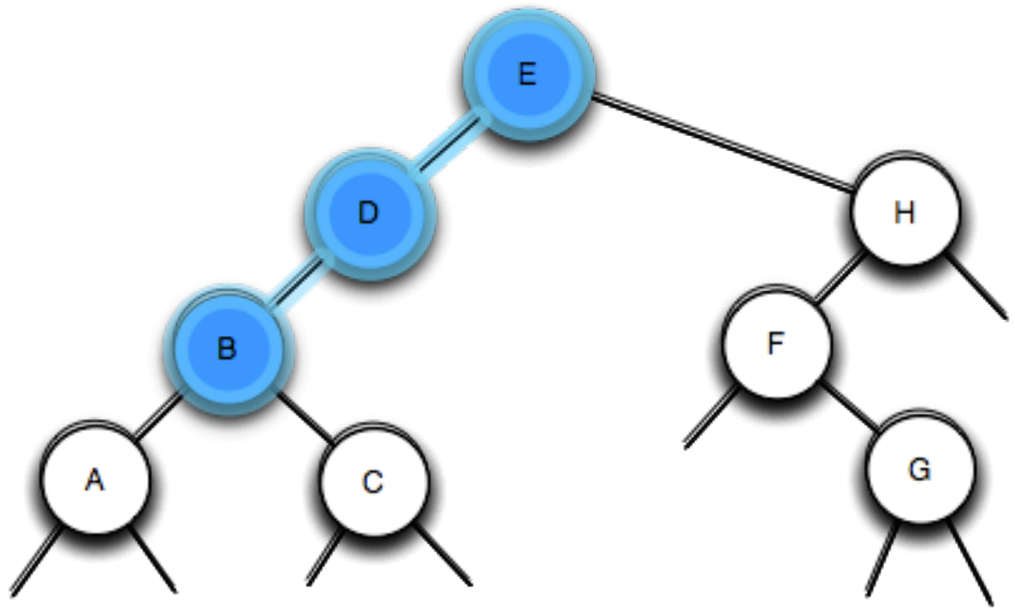
$$T = \{\ E\ ,\ \{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}\ ,\ \{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \ \}\ \ \}$$

Recorrido de árboles binarios: pre-orden



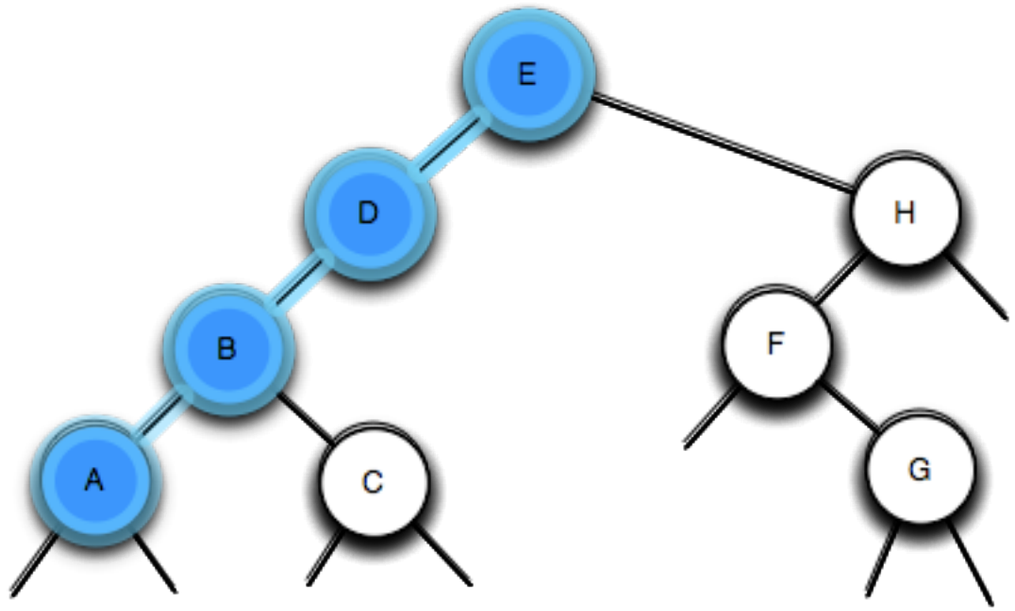
$$T = \{\ E\ ,\ \{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}\ ,\ \{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \ \}\ \ \}$$

Recorrido de árboles binarios: pre-orden



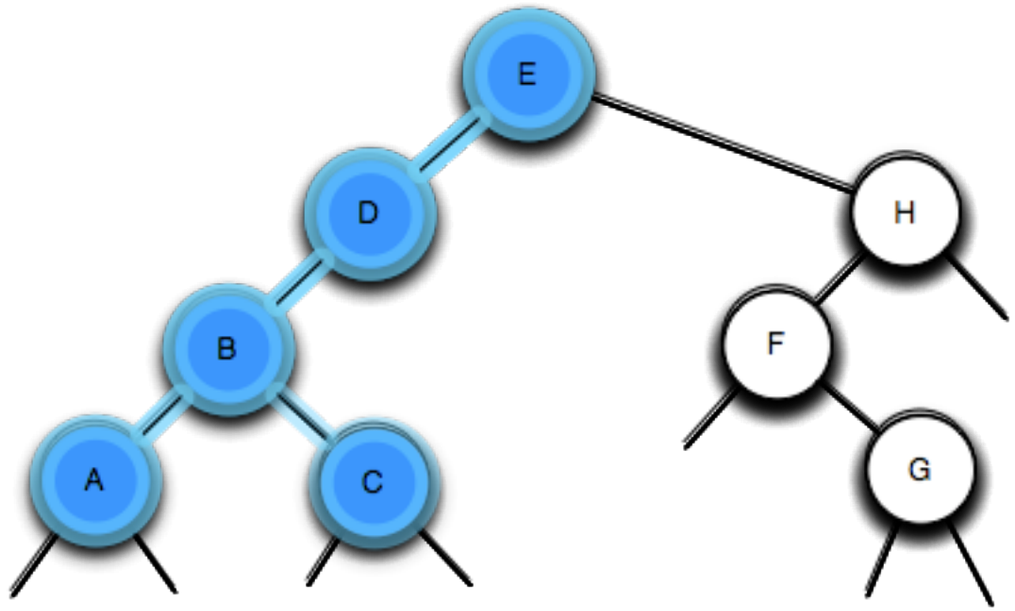
$$T = \{\ E\ ,\ \{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}\ ,\ \{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \}\ \}$$

Recorrido de árboles binarios: pre-orden



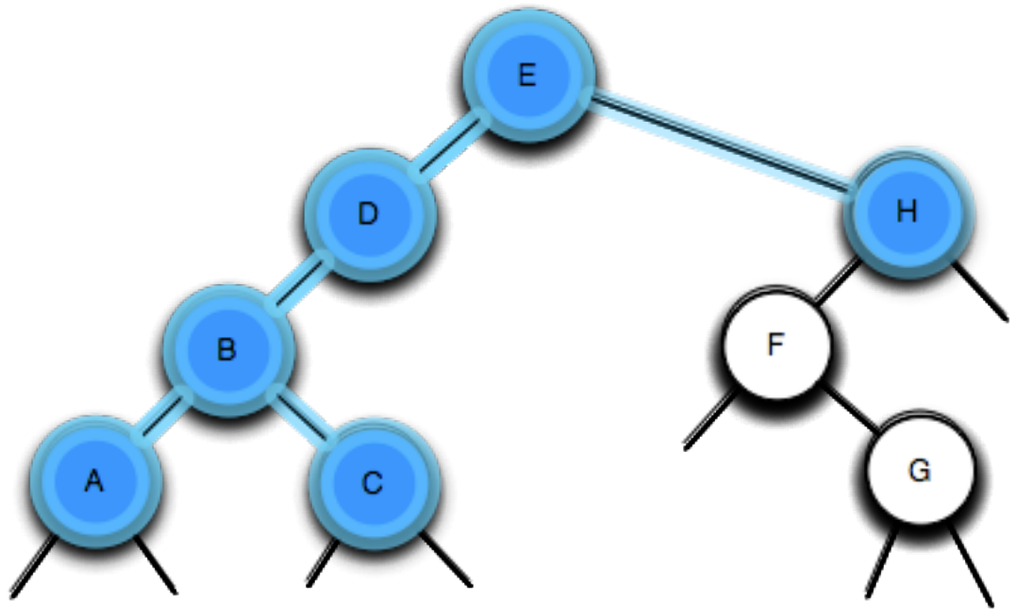
$$T = \{\ E\ ,\ \{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}\ ,\ \{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \}\ \}$$

Recorrido de árboles binarios: pre-orden



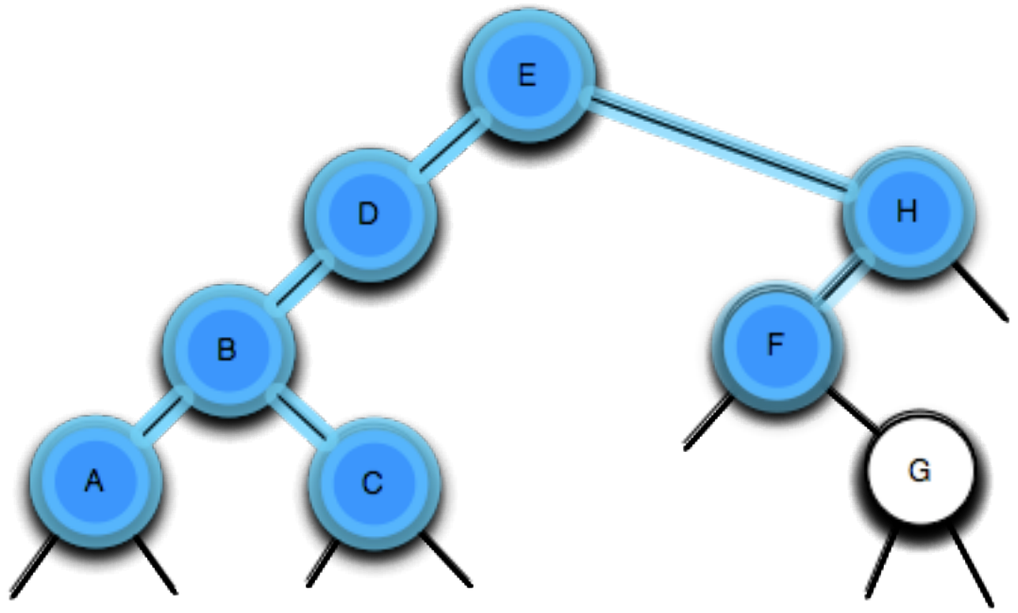
$$T = \{\ E\ ,\ \{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}\ ,\ \{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \}\ \}$$

Recorrido de árboles binarios: pre-orden



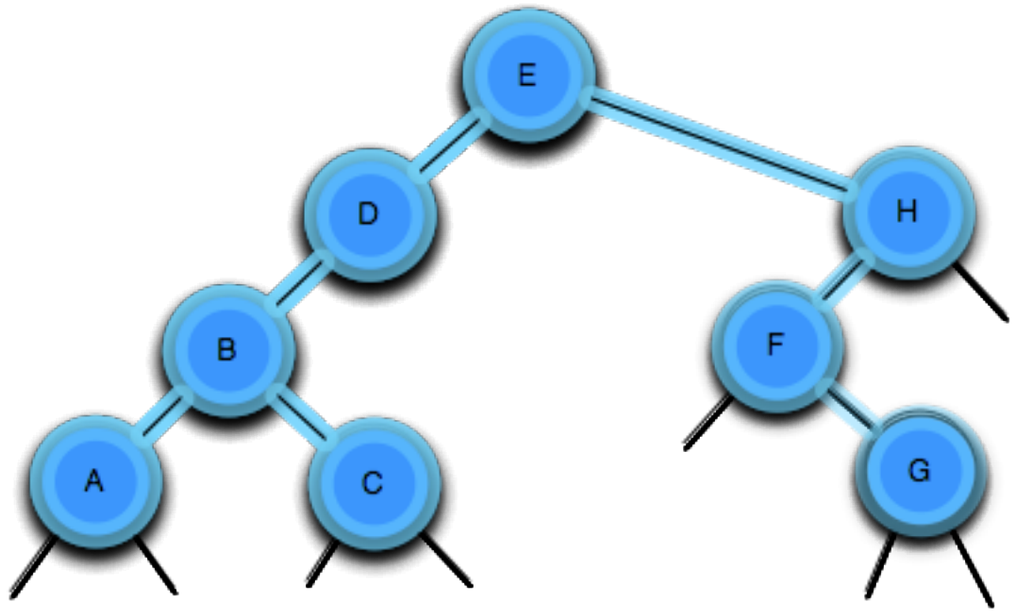
$$T = \{\ E\ ,\ \{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}\ ,\ \{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \}\ \}$$

Recorrido de árboles binarios: pre-orden



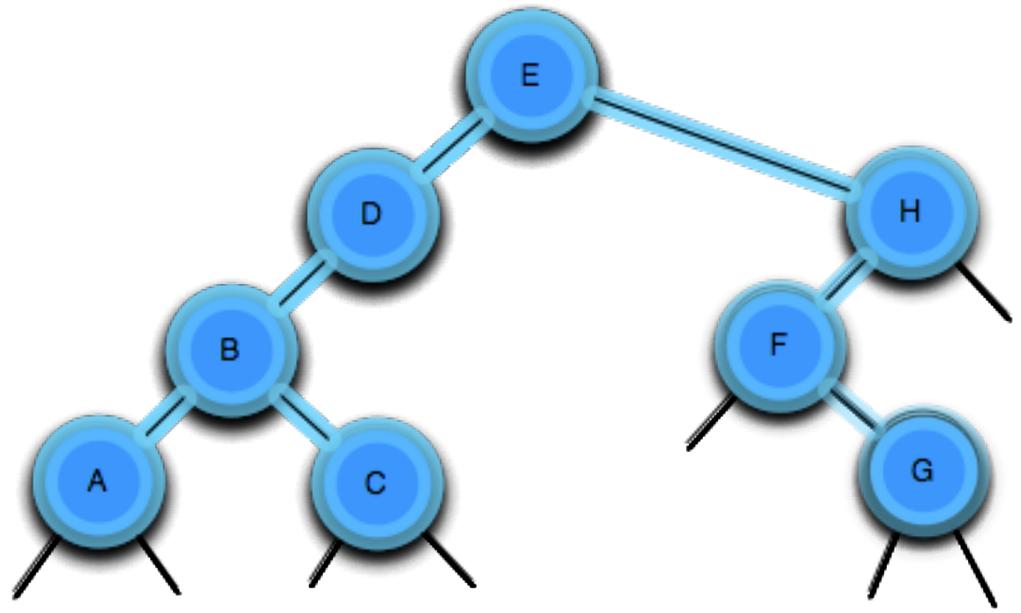
$$T = \{\ E\ ,\ \{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}\ ,\ \{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \ \}\ \ \}$$

Recorrido de árboles binarios: pre-orden



$$T = \{\ E\ ,\ \{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}\ ,\ \{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \ \}\ \ \}$$

Recorrido de árboles binarios: pre-orden



Camino: E-D-B-A-C-H-F-G

$$T = \{\ E\ ,\ \{D,\ \{B,\{A,0,0\}\{C,0,0\}\},\ 0\ \}\ ,\ \{H,\ \{F\ ,\ 0,\{G,0,0\}\}\ ,\ 0\ \ \}\ \ \}$$

Recorrido de árboles: post-orden

- Para un árbol general:

1. Hacer un recorrido en post-orden para todos los sub-árboles de la raíz, uno a uno en el orden dado,

2. visitar la raíz.

- Para un árbol binario:

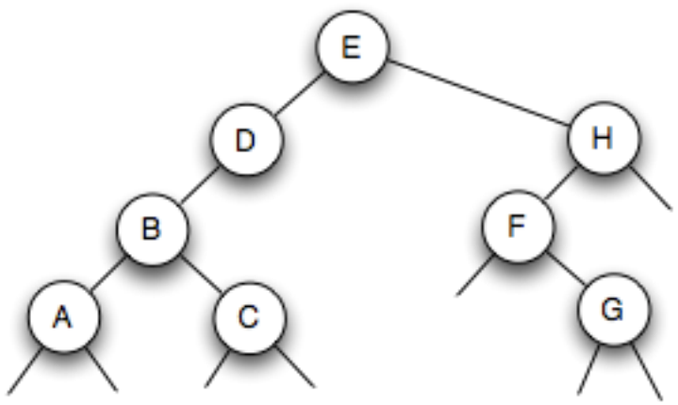
1. Recorrer el sub-árbol izquierdo,

2. recorrer el sub-árbol derecho,

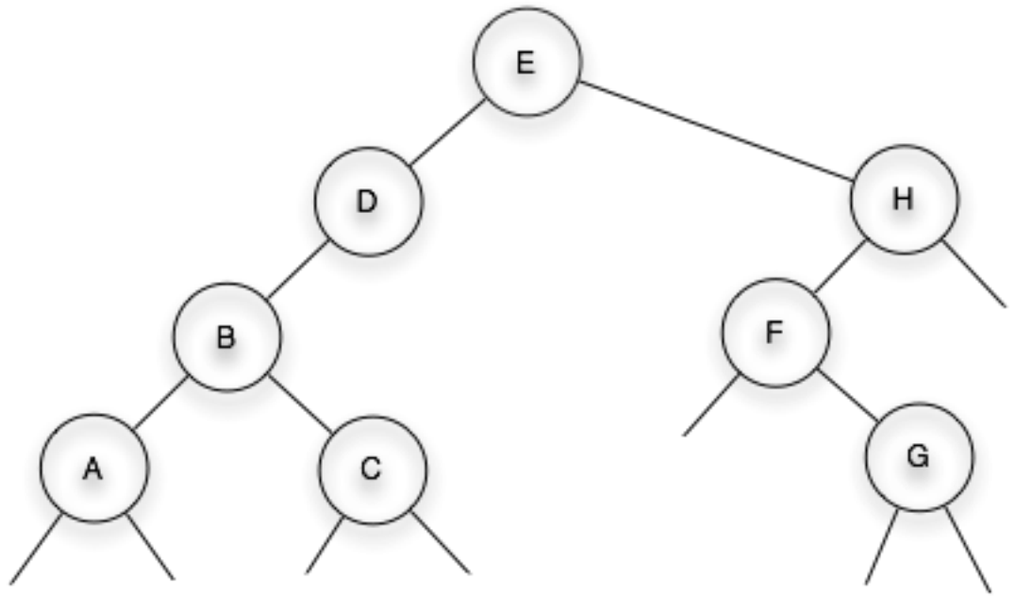
3. visitar la raíz.

Recorrido de árboles: post-orden

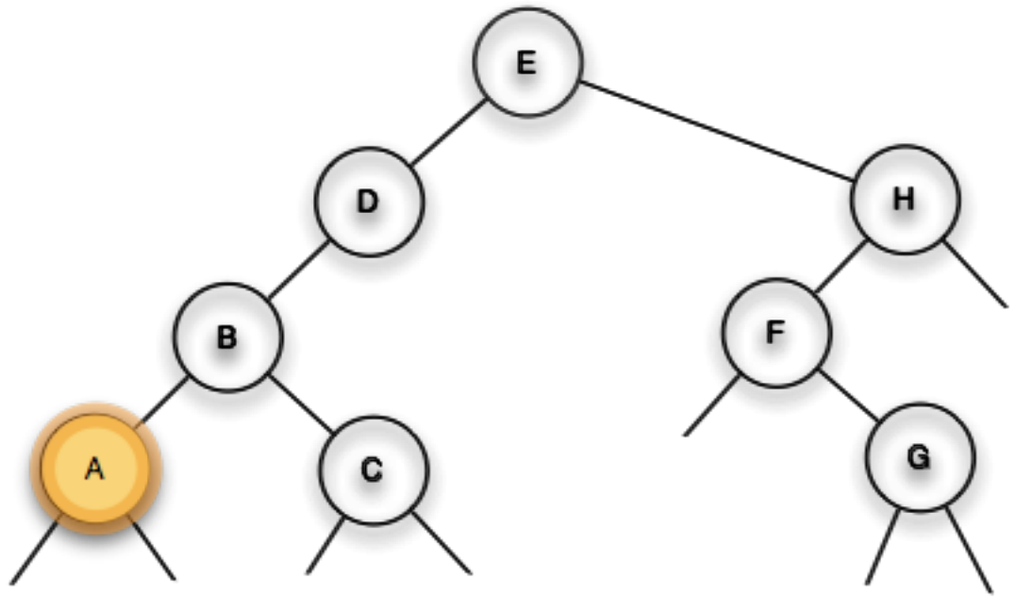
```
void Tree:: postorder ( void(*process)(Item& item))
{
    start();
    while( is_Valid())
    {
        currentChild->postorder(process);
        forth();
    }
    process(m_data);
}
```



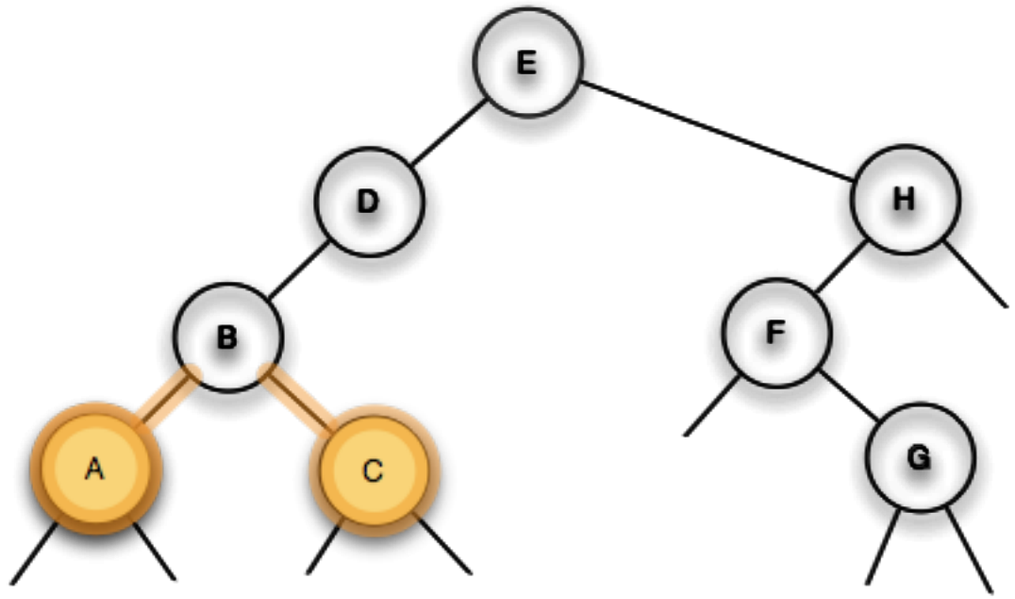
Recorrido de árboles binarios: post-orden



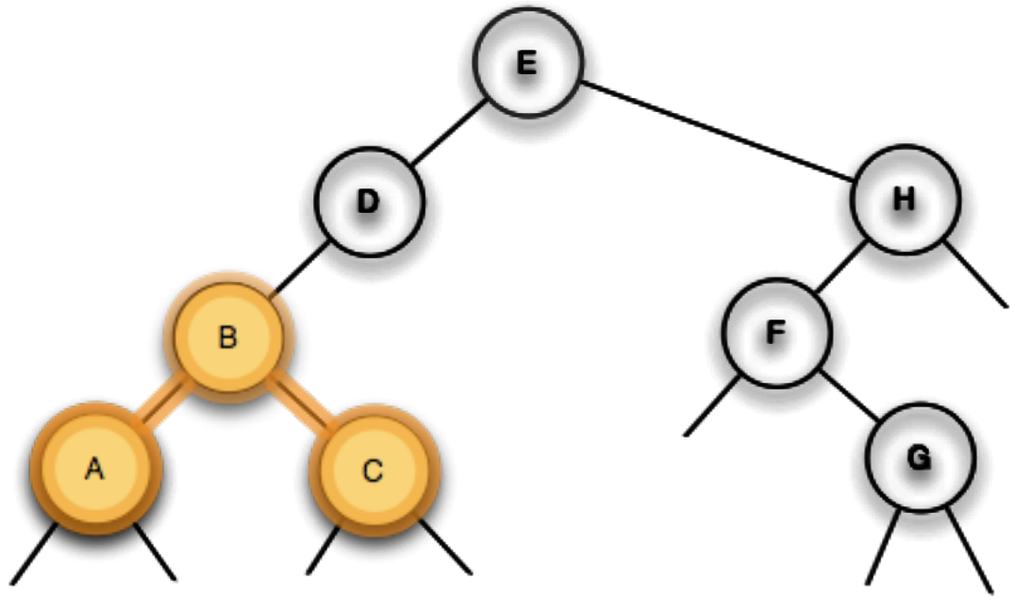
Recorrido de árboles binarios: post-orden



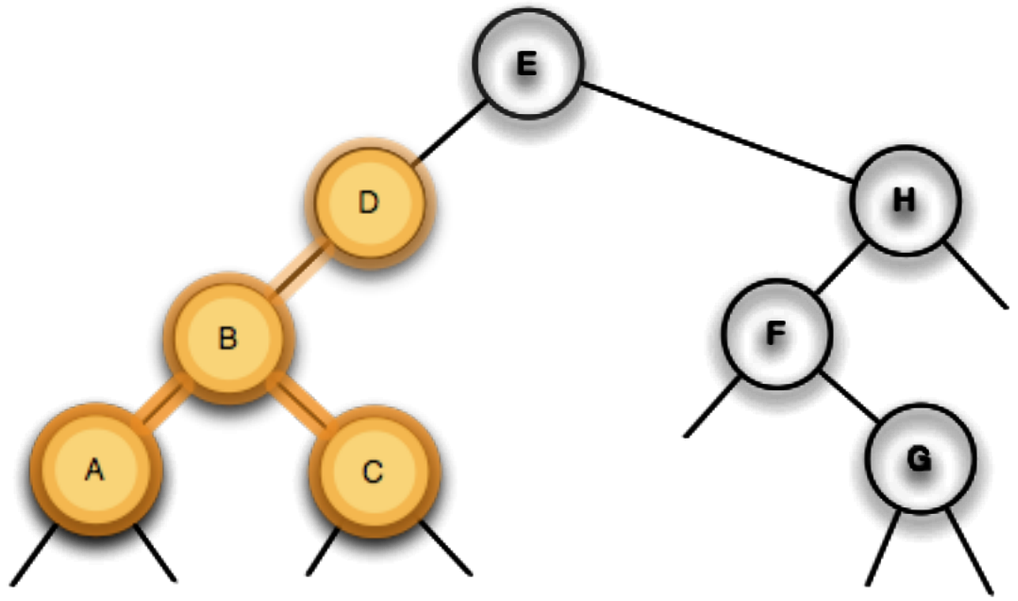
Recorrido de árboles binarios: post-orden



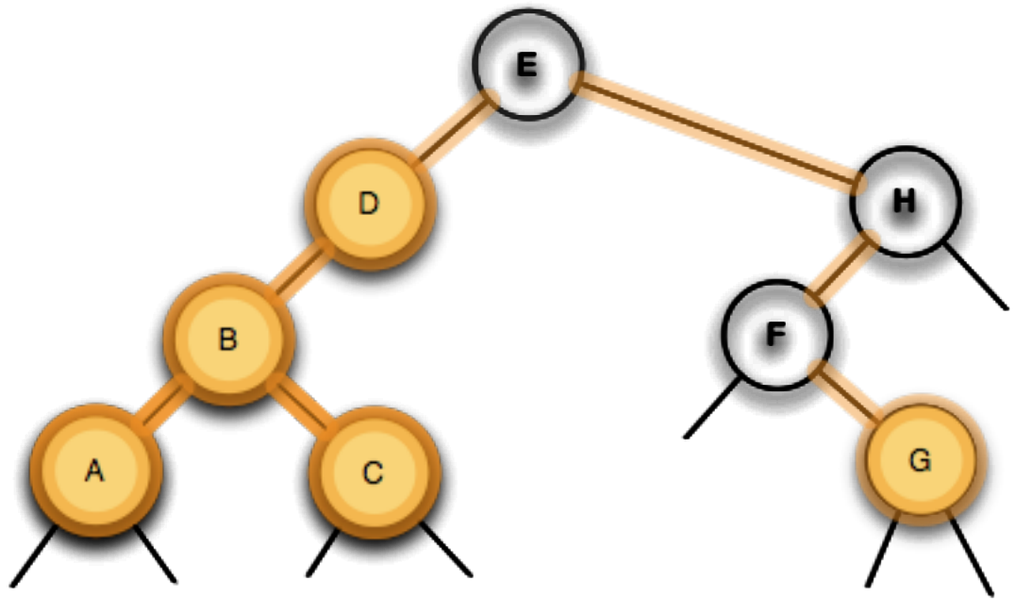
Recorrido de árboles binarios: post-orden



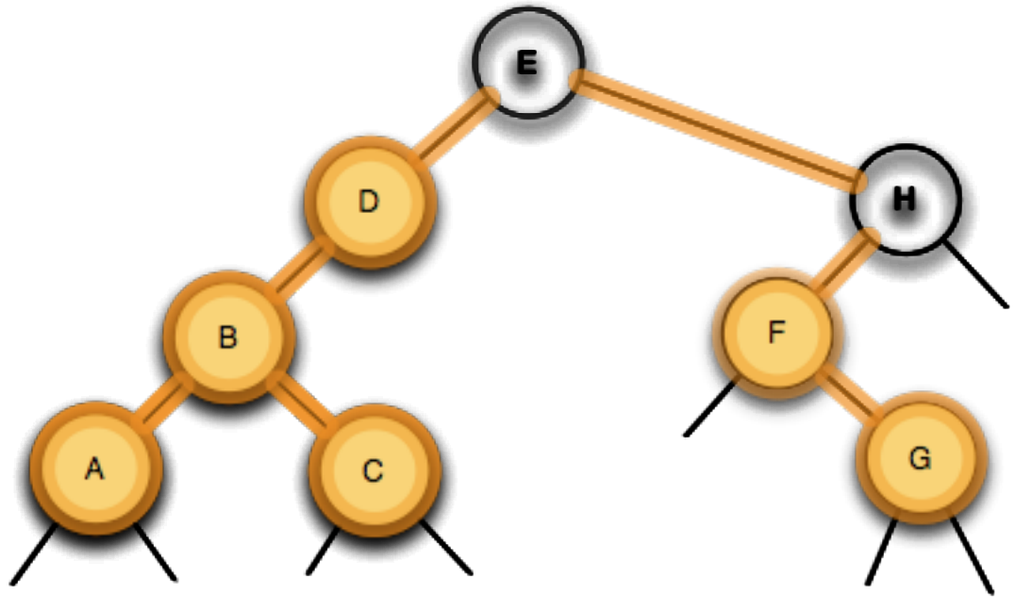
Recorrido de árboles binarios: post-orden



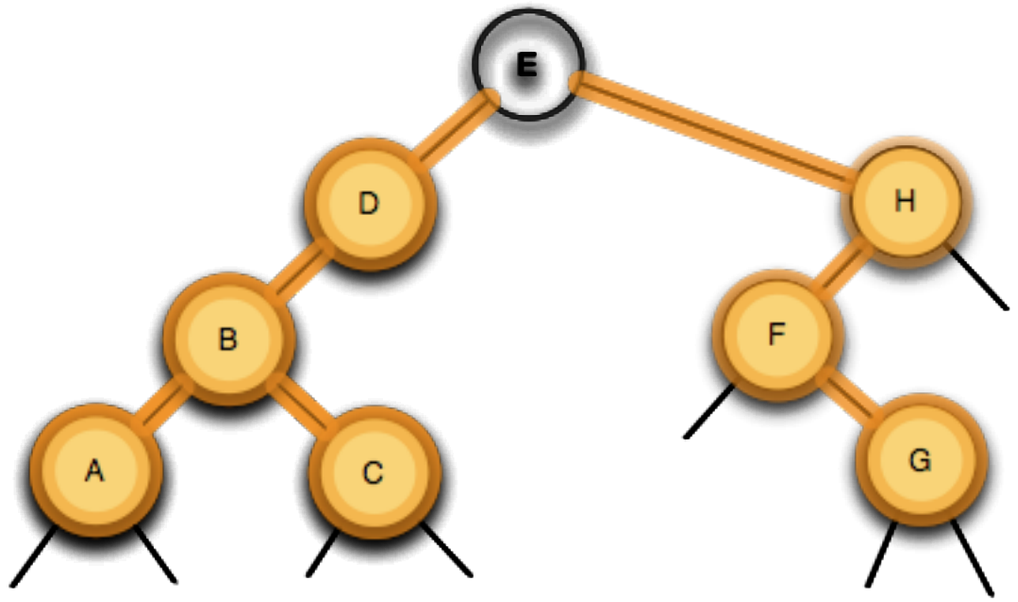
Recorrido de árboles binarios: post-orden



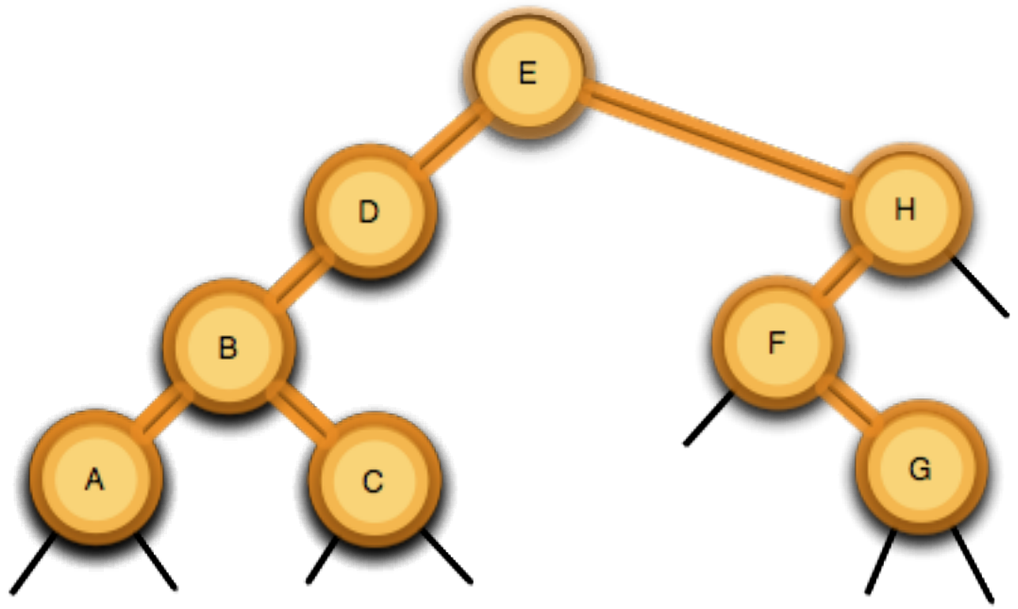
Recorrido de árboles binarios: post-orden



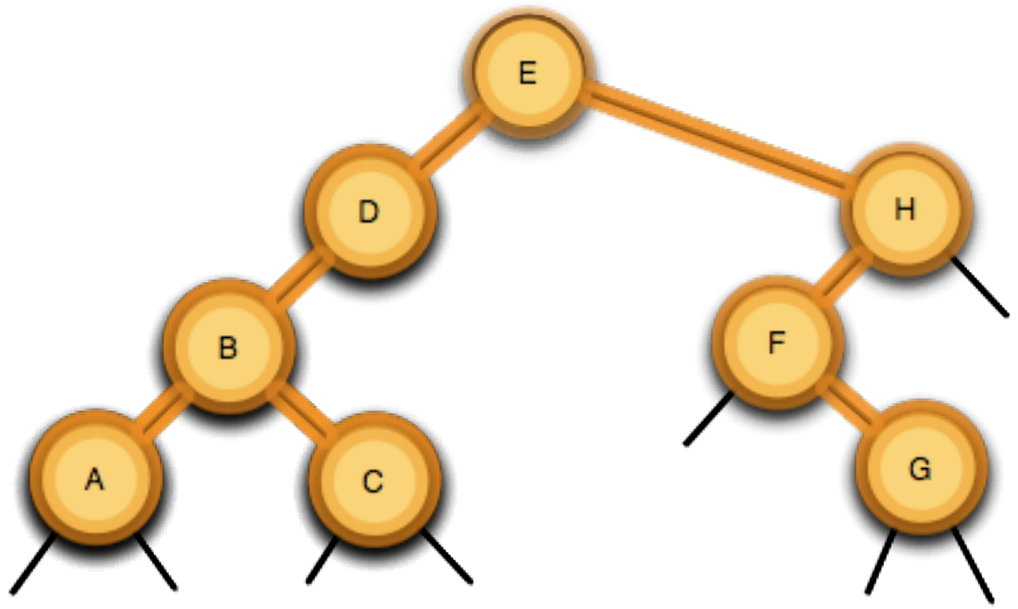
Recorrido de árboles binarios: post-orden



Recorrido de árboles binarios: post-orden



Recorrido de árboles binarios: post-orden



Camino: A-C-B-D-G-F-H-E

Recorrido de árboles binarios: orden

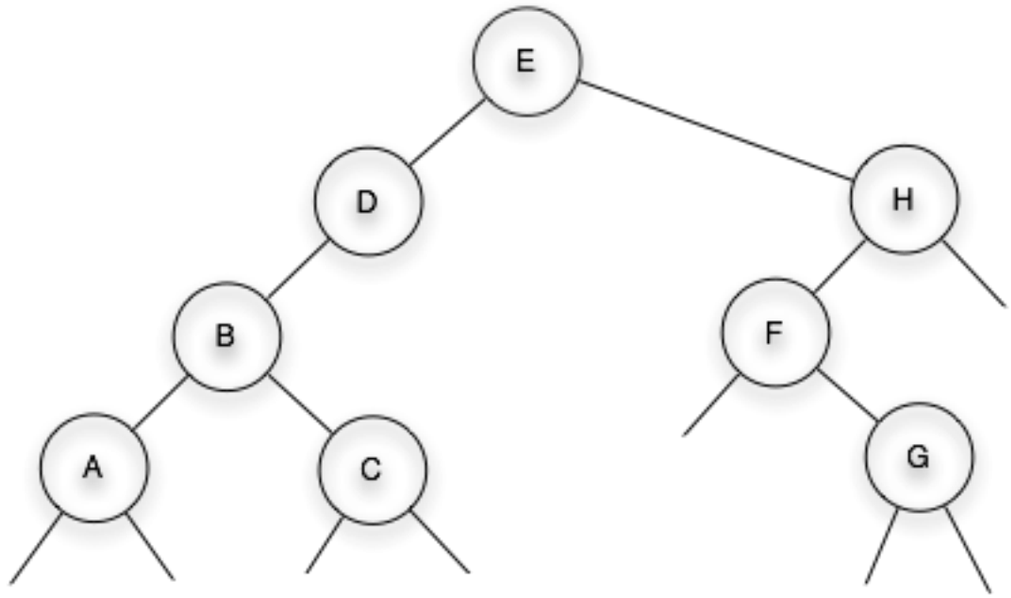
- Solo tiene sentido para árboles binarios.

1. Recorrer el sub-árbol izquierdo,

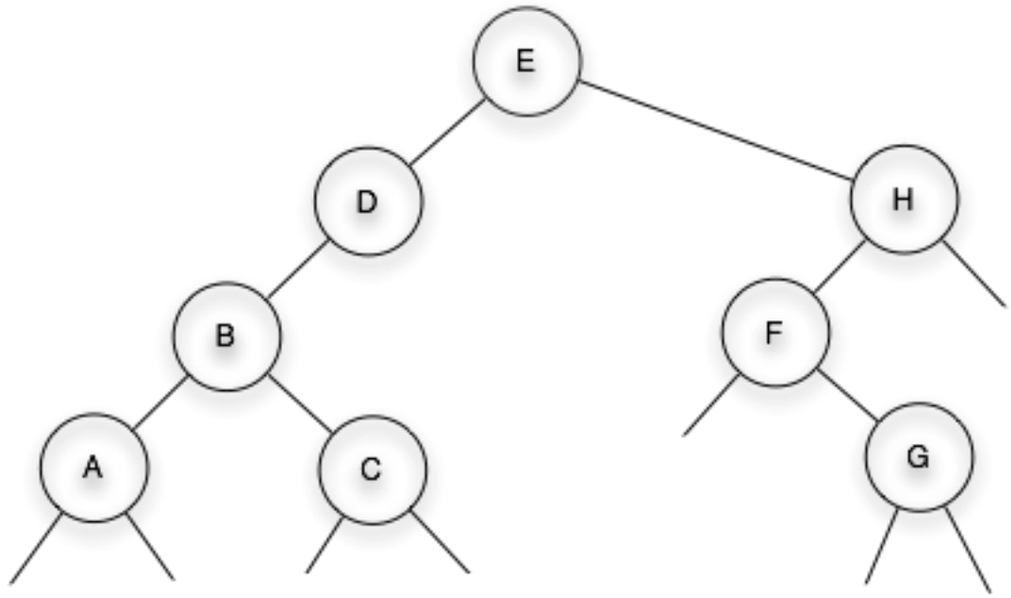
2. visitar la raíz,

3. recorrer el sub-árbol derecho.

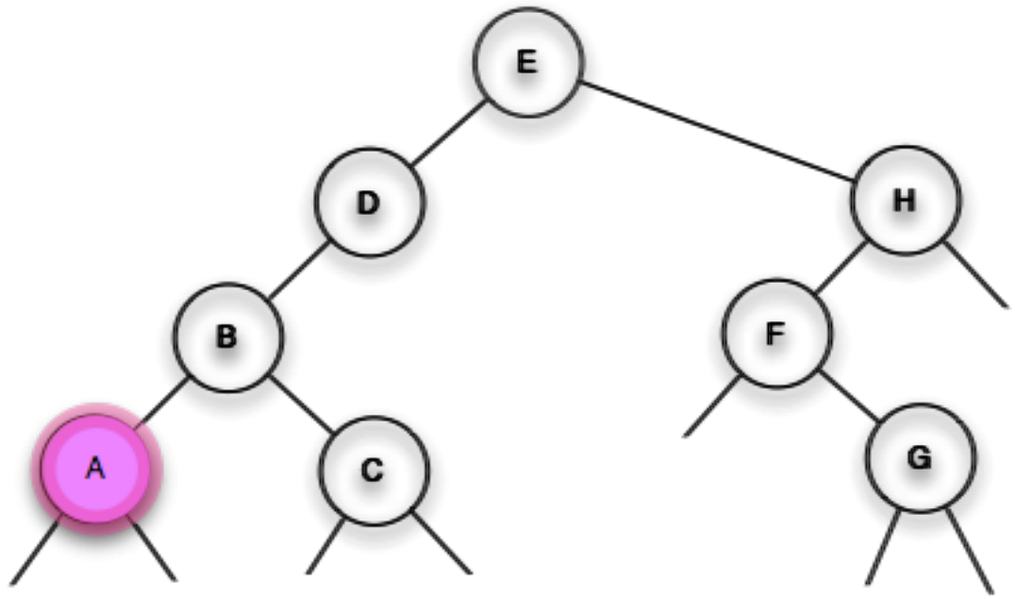
Recorrido de árboles binarios: orden



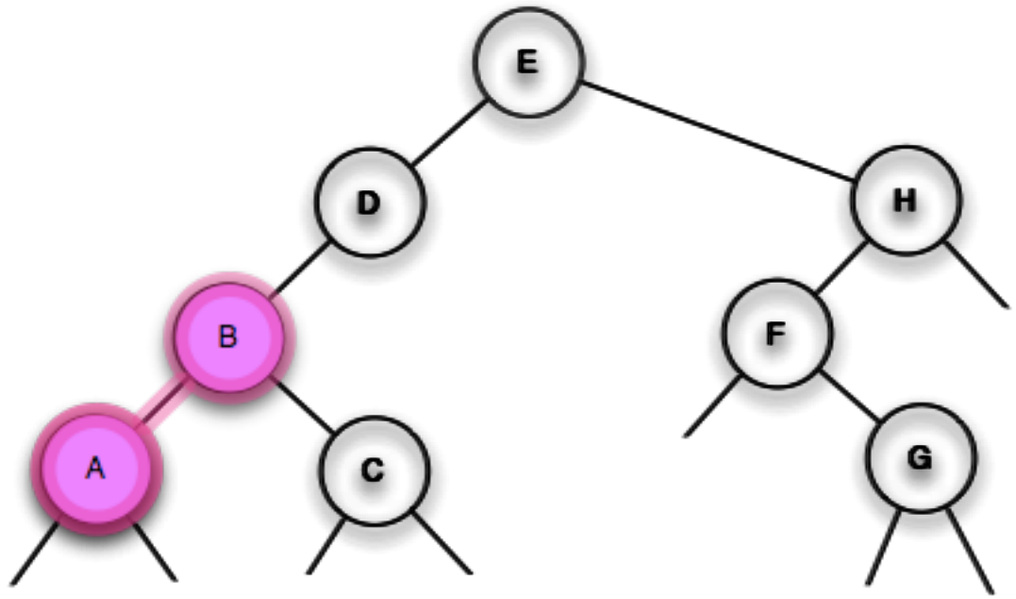
Recorrido de árboles binarios: orden



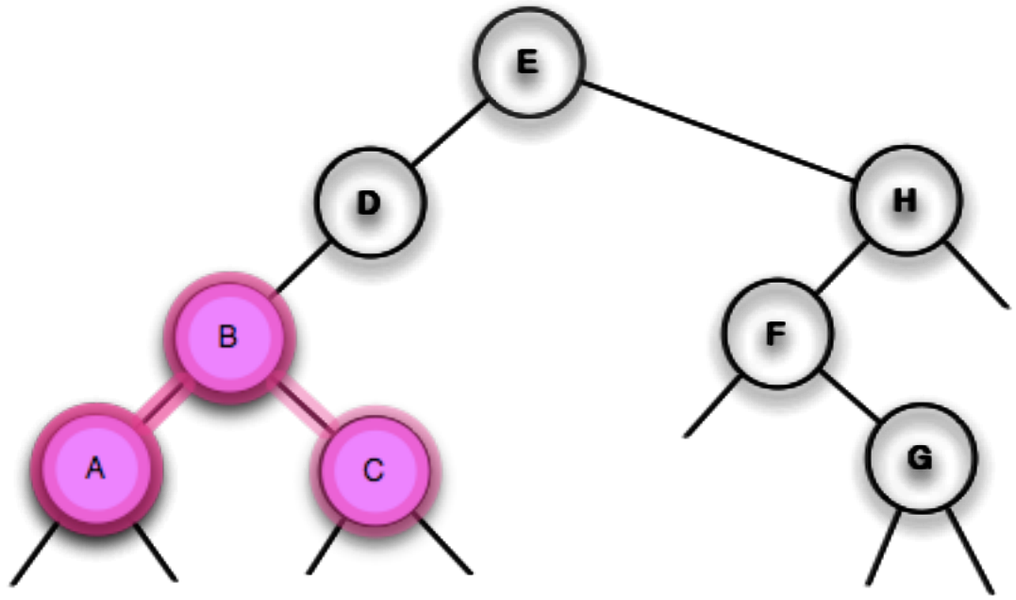
Recorrido de árboles binarios: orden



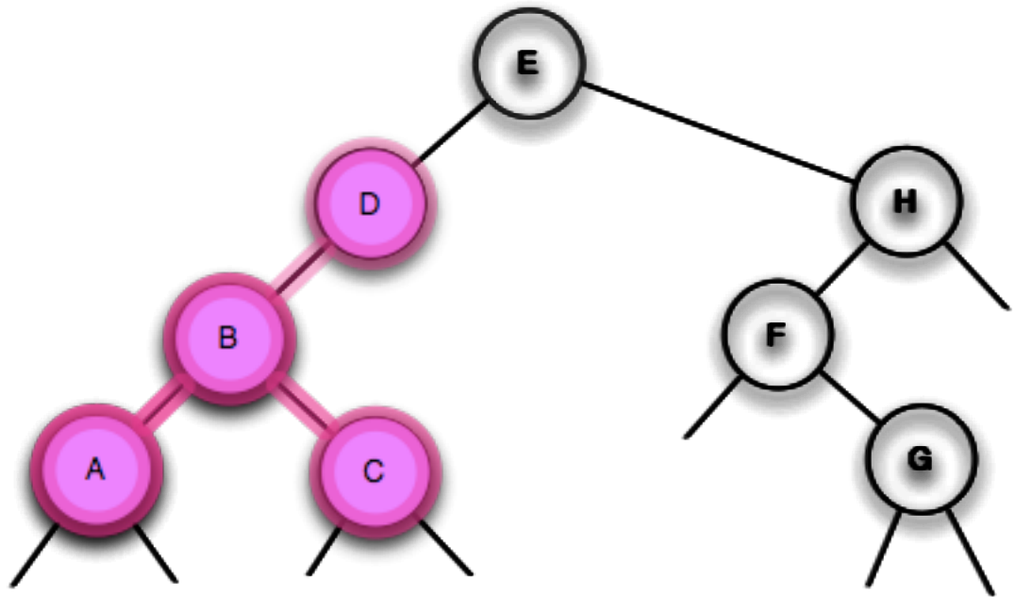
Recorrido de árboles binarios: orden



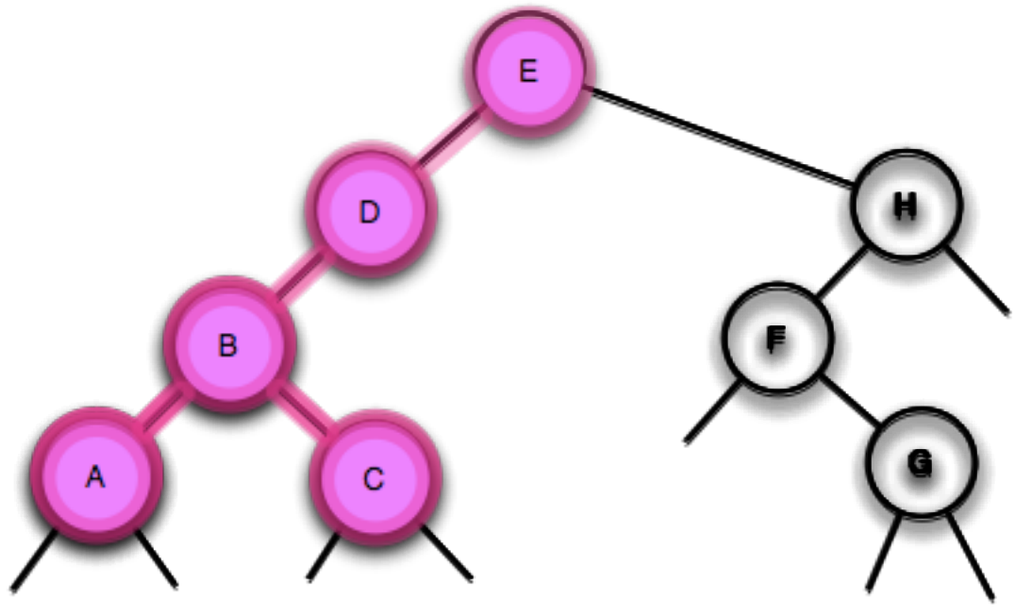
Recorrido de árboles binarios: orden



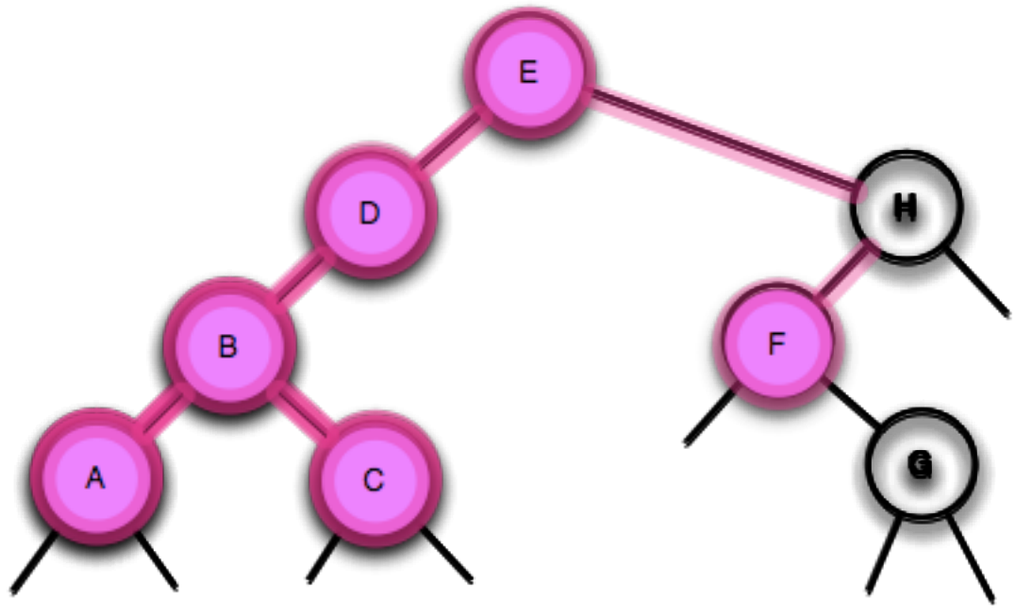
Recorrido de árboles binarios: orden



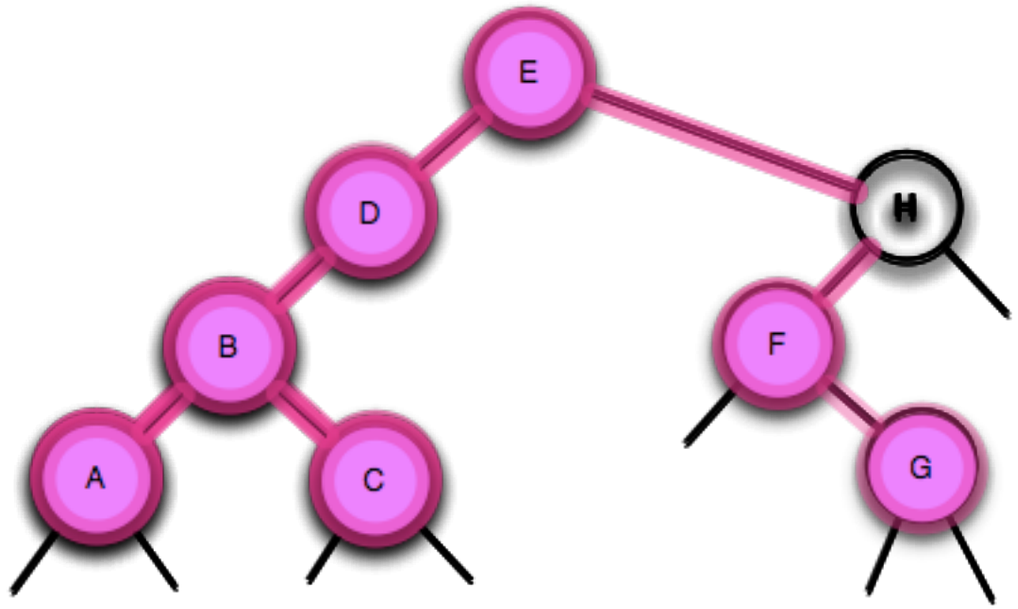
Recorrido de árboles binarios: orden



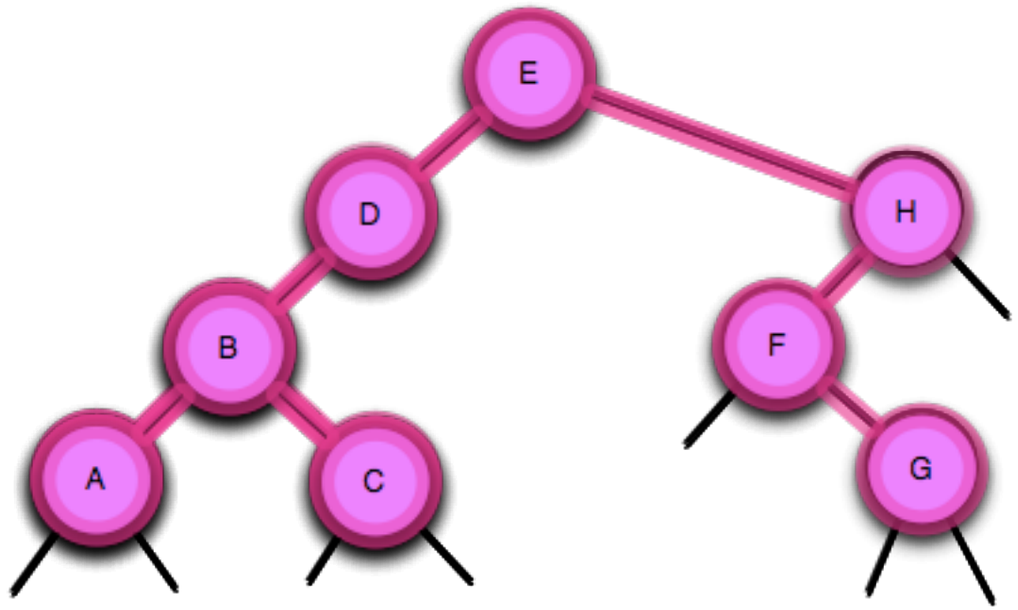
Recorrido de árboles binarios: orden



Recorrido de árboles binarios: orden

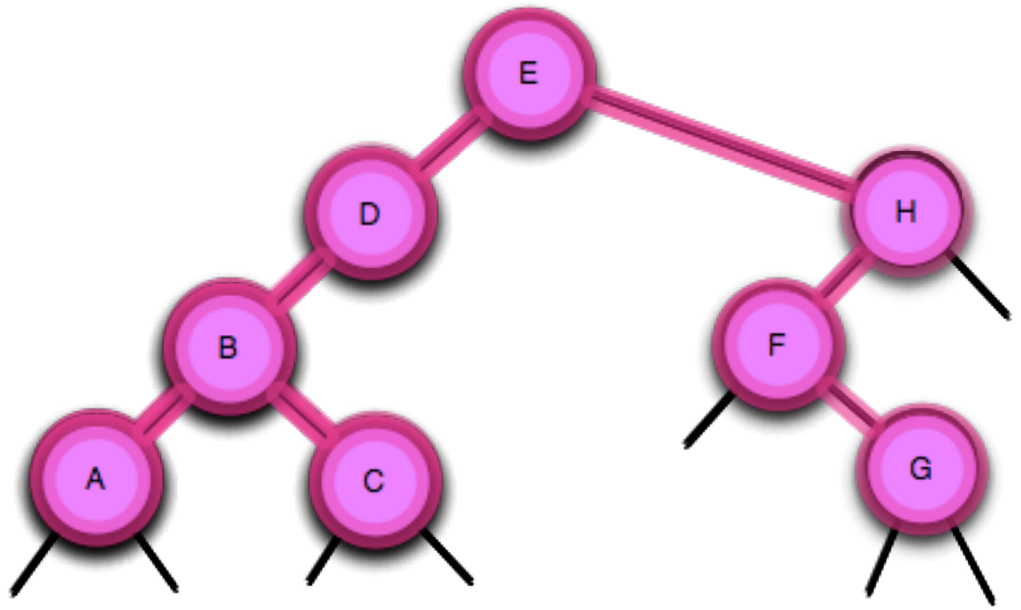


Recorrido de árboles binarios: orden



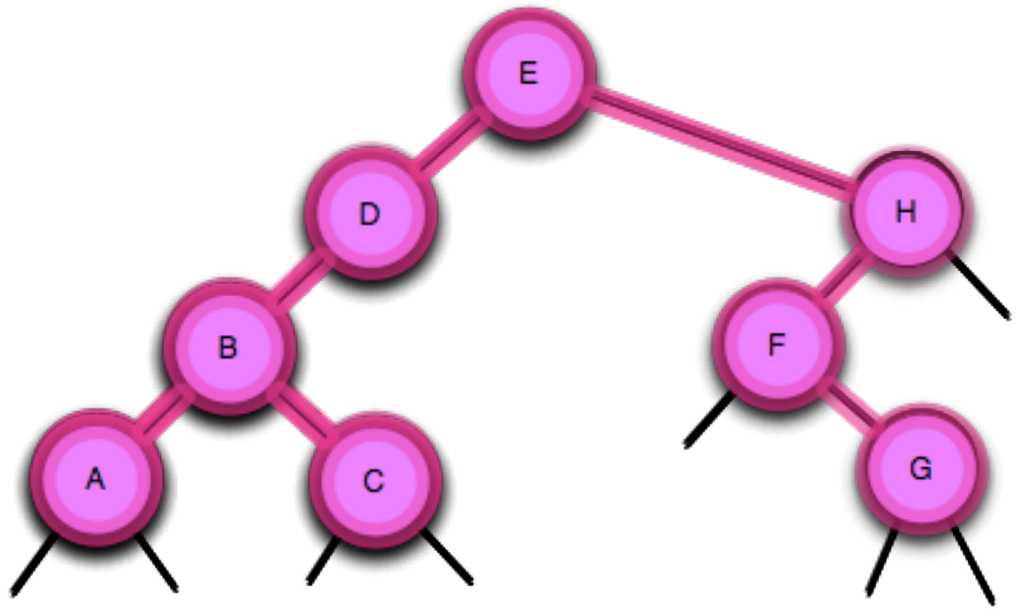
Recorrido de árboles binarios: orden

http://nova.umuc.edu/~jarc/idsv/lesson1.html



Recorrido de árboles binarios: orden

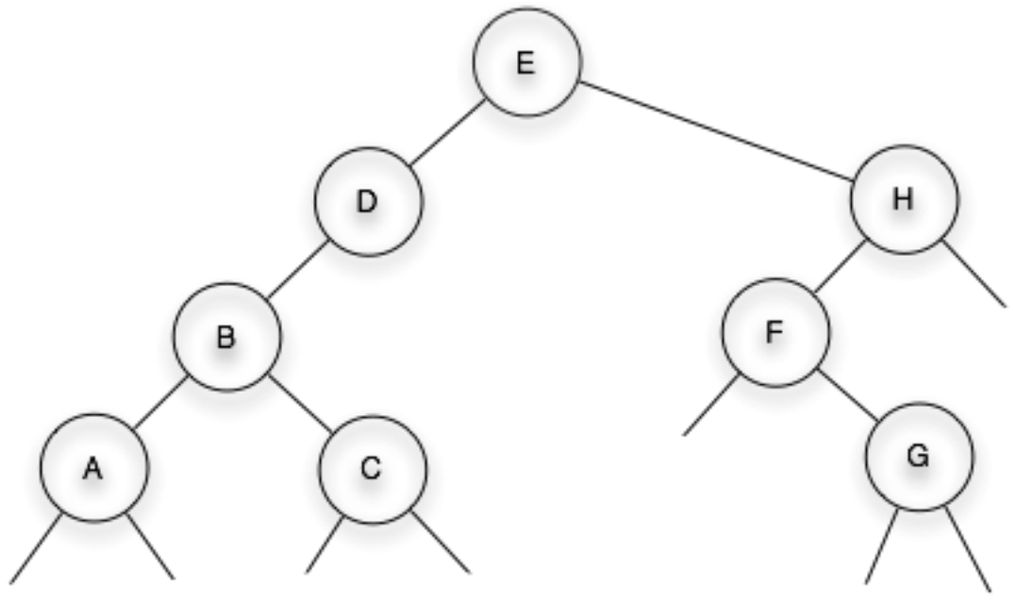
http://nova.umuc.edu/~jarc/idsv/lesson1.html



Camino: A-B-C-D-E-F-G-H

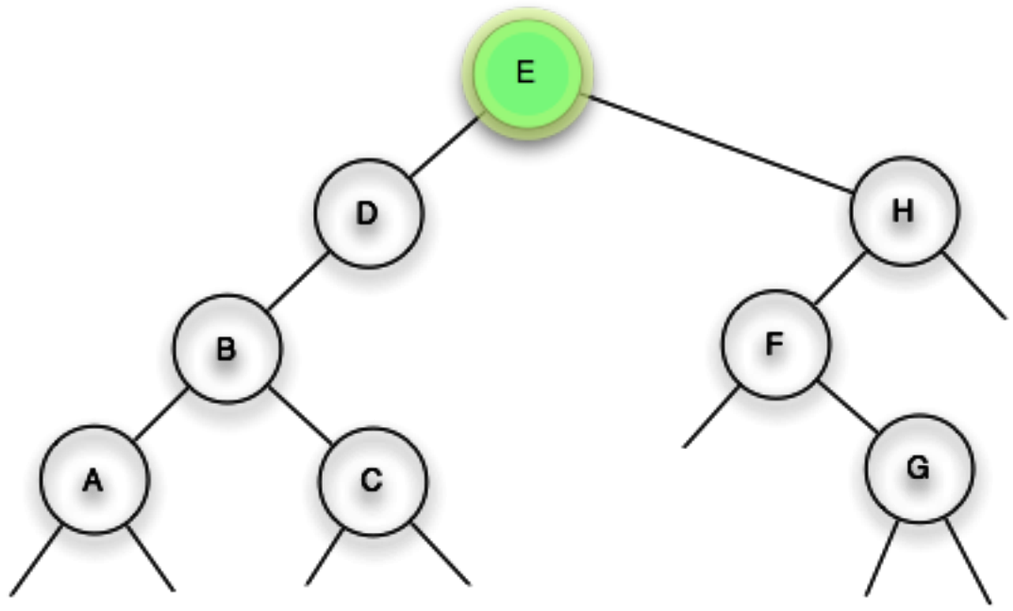
Recorrido de árboles: *breadth-first*

- Los recorridos ***depth-first*** están definidos de manera recursiva,
- Los recorridos ***breadth-first*** se definen como recorridos no-recursivos.
- Recorre todos los nodos del nivel cero (la raíz), luego todos los nodos del nivel uno, etc.
- En cada nivel los nodos se recorren de izquierda a derecha.



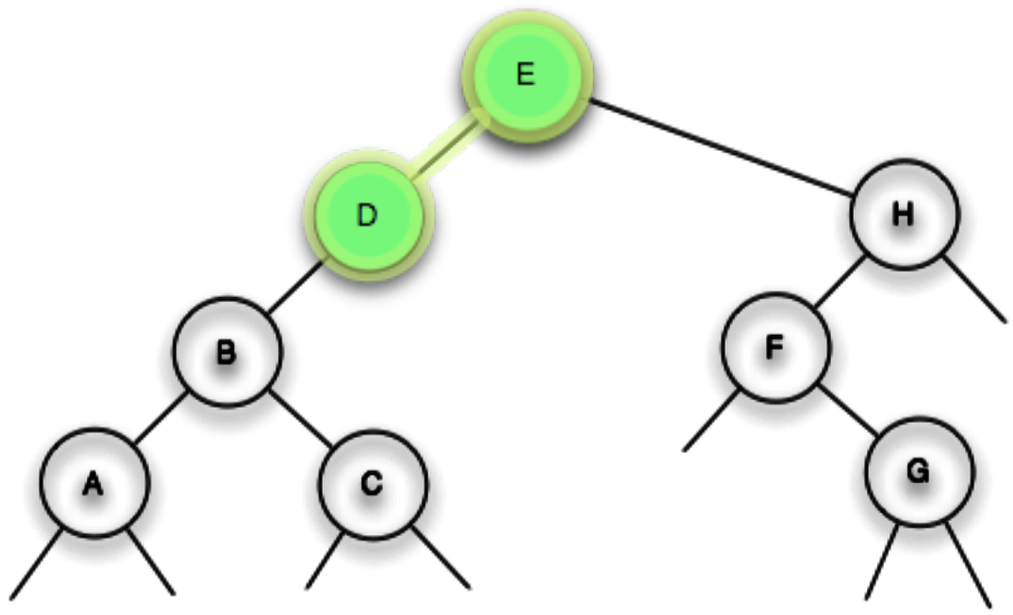
Recorrido de árboles: *breadth-first*

- Los recorridos ***depth-first*** están definidos de manera recursiva,
- Los recorridos ***breadth-first*** se definen como recorridos no-recursivos.
- Recorre todos los nodos del nivel cero (la raíz), luego todos los nodos del nivel uno, etc.
- En cada nivel los nodos se recorren de izquierda a derecha.



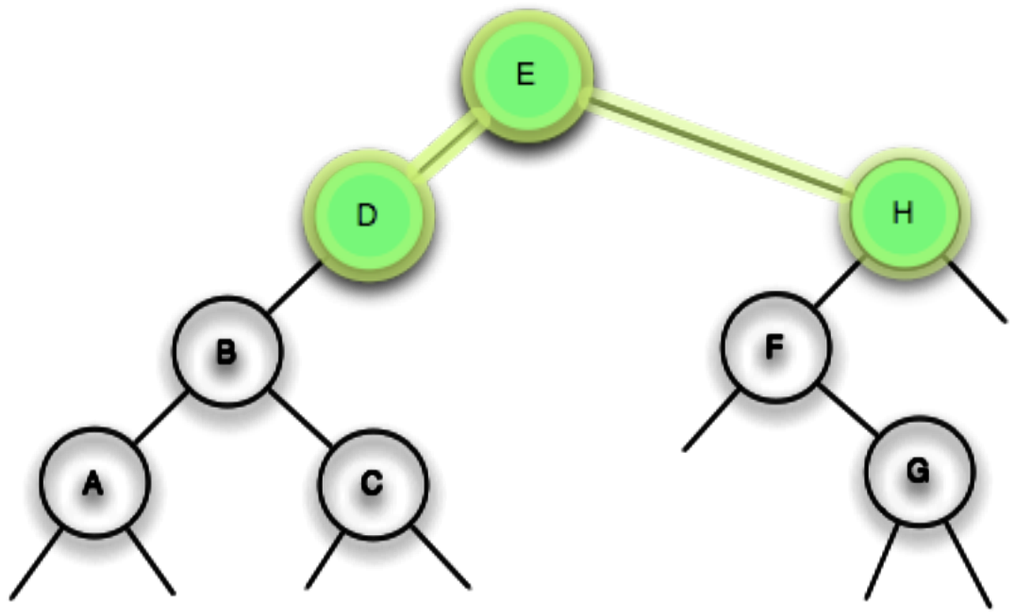
Recorrido de árboles: *breadth-first*

- Los recorridos ***depth-first*** están definidos de manera recursiva,
- Los recorridos ***breadth-first*** se definen como recorridos no-recursivos.
- Recorre todos los nodos del nivel cero (la raíz), luego todos los nodos del nivel uno, etc.
- En cada nivel los nodos se recorren de izquierda a derecha.



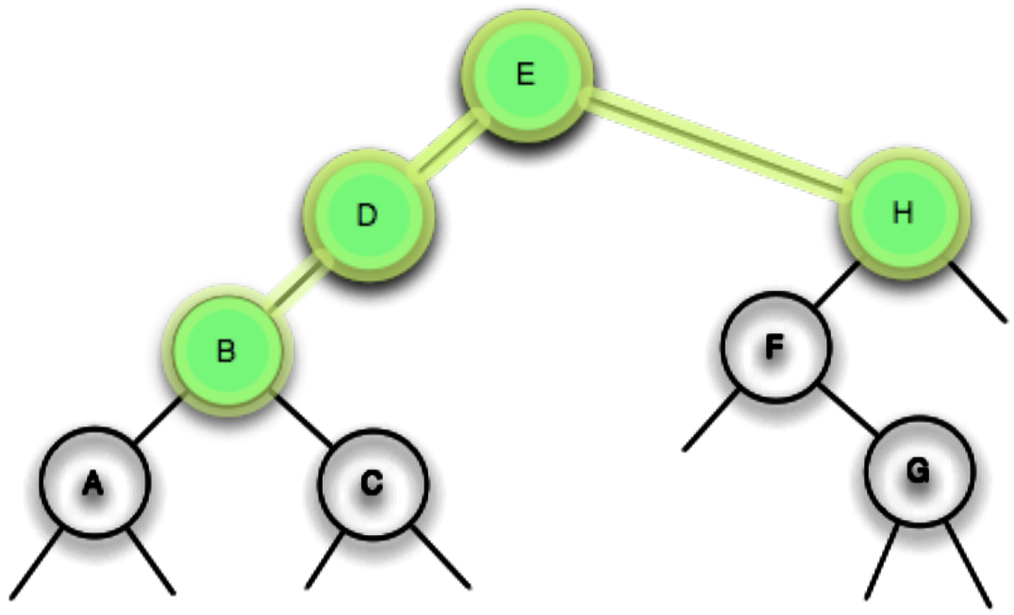
Recorrido de árboles: *breadth-first*

- Los recorridos ***depth-first*** están definidos de manera recursiva,
- Los recorridos ***breadth-first*** se definen como recorridos no-recursivos.
- Recorre todos los nodos del nivel cero (la raíz), luego todos los nodos del nivel uno, etc.
- En cada nivel los nodos se recorren de izquierda a derecha.



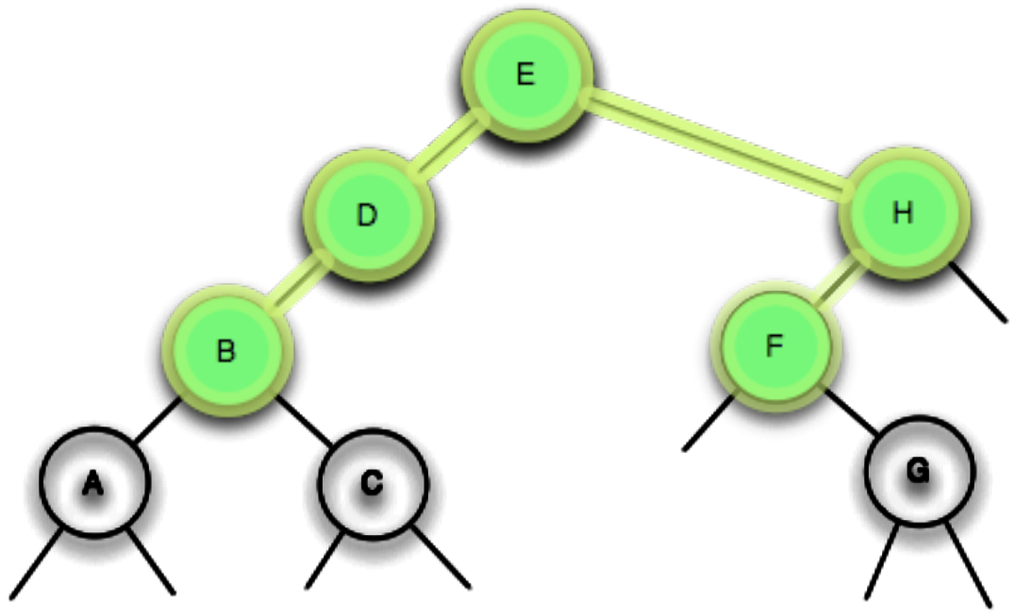
Recorrido de árboles: *breadth-first*

- Los recorridos ***depth-first*** están definidos de manera recursiva,
- Los recorridos ***breadth-first*** se definen como recorridos no-recursivos.
- Recorre todos los nodos del nivel cero (la raíz), luego todos los nodos del nivel uno, etc.
- En cada nivel los nodos se recorren de izquierda a derecha.



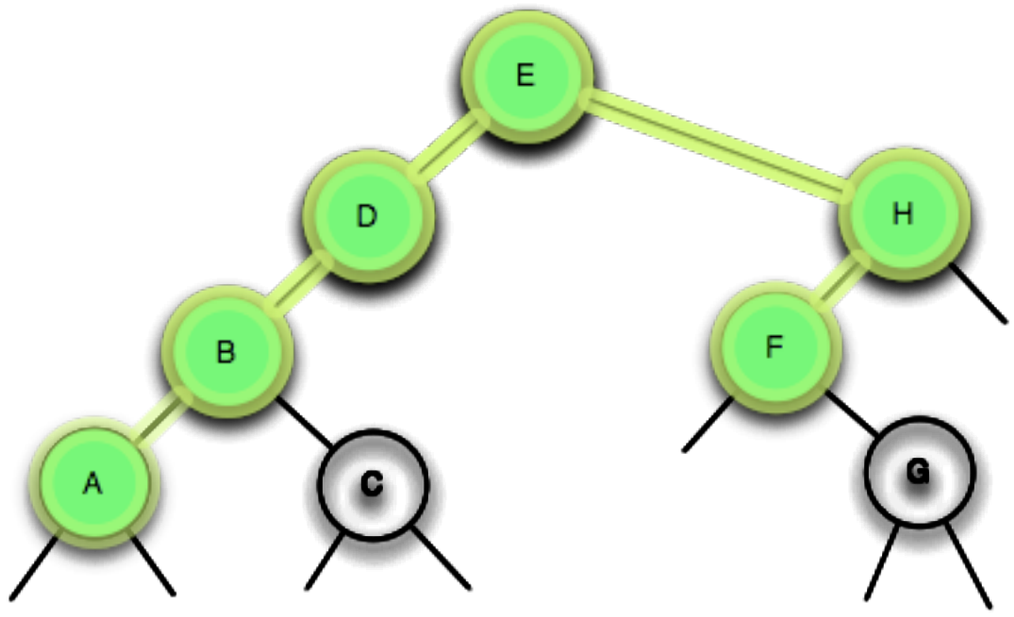
Recorrido de árboles: *breadth-first*

- Los recorridos ***depth-first*** están definidos de manera recursiva,
- Los recorridos ***breadth-first*** se definen como recorridos no-recursivos.
- Recorre todos los nodos del nivel cero (la raíz), luego todos los nodos del nivel uno, etc.
- En cada nivel los nodos se recorren de izquierda a derecha.



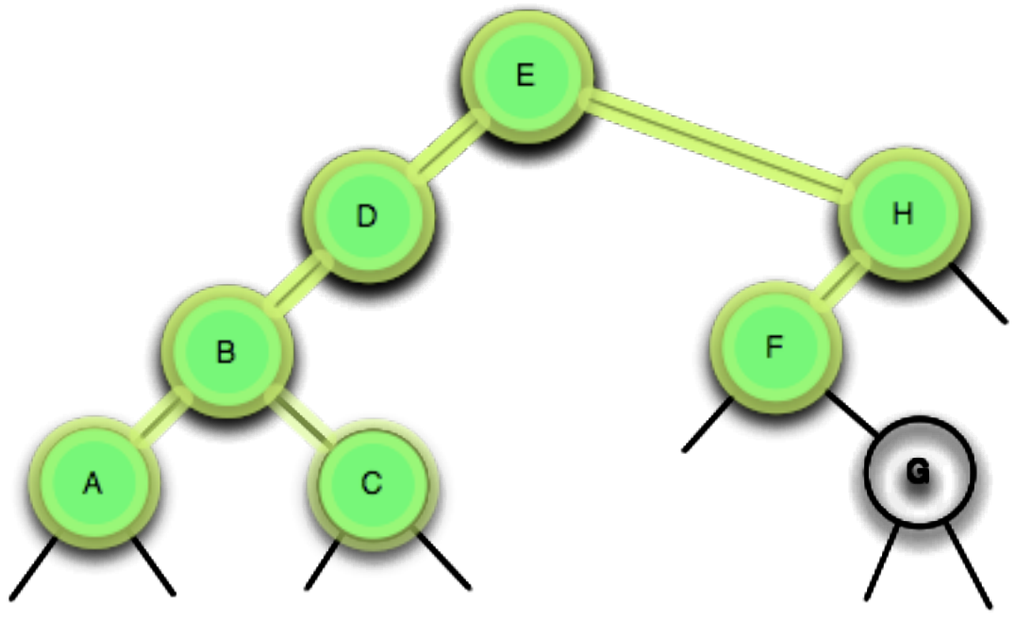
Recorrido de árboles: *breadth-first*

- Los recorridos ***depth-first*** están definidos de manera recursiva,
- Los recorridos ***breadth-first*** se definen como recorridos no-recursivos.
- Recorre todos los nodos del nivel cero (la raíz), luego todos los nodos del nivel uno, etc.
- En cada nivel los nodos se recorren de izquierda a derecha.



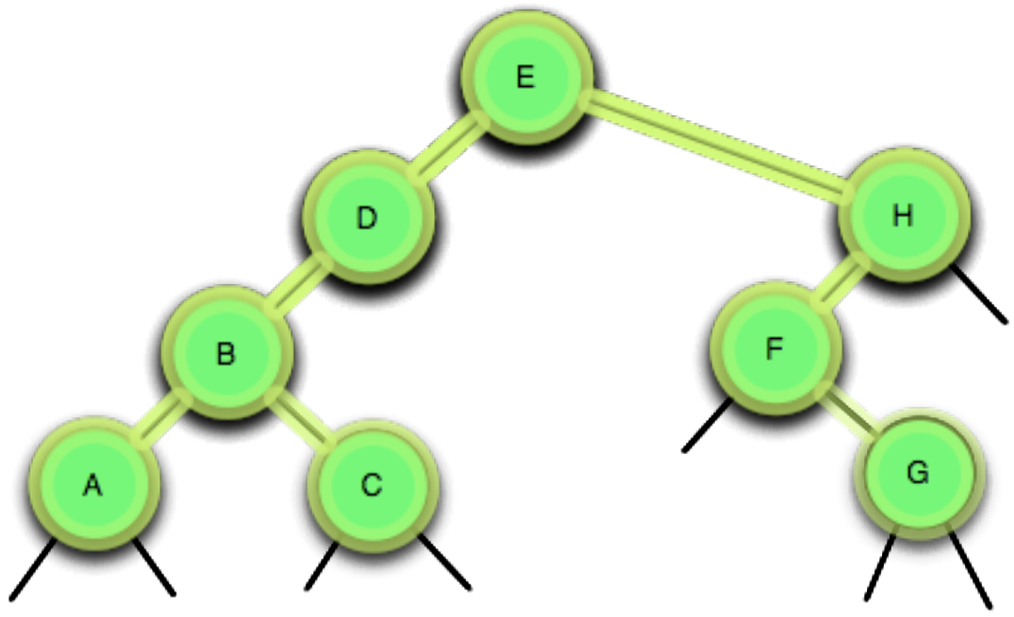
Recorrido de árboles: *breadth-first*

- Los recorridos ***depth-first*** están definidos de manera recursiva,
- Los recorridos ***breadth-first*** se definen como recorridos no-recursivos.
- Recorre todos los nodos del nivel cero (la raíz), luego todos los nodos del nivel uno, etc.
- En cada nivel los nodos se recorren de izquierda a derecha.



Recorrido de árboles: *breadth-first*

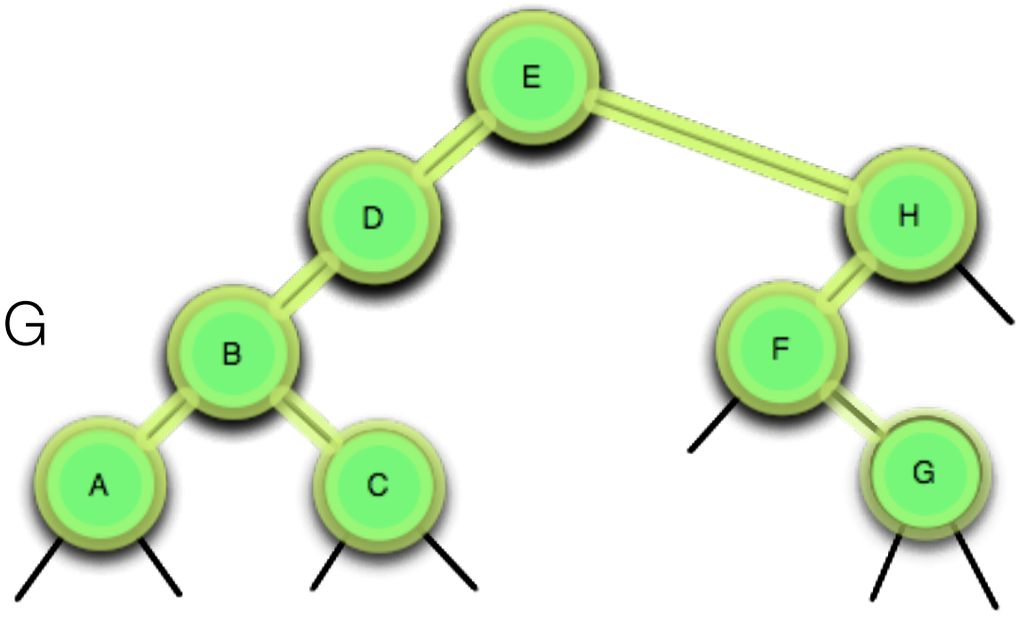
- Los recorridos ***depth-first*** están definidos de manera recursiva,
- Los recorridos ***breadth-first*** se definen como recorridos no-recursivos.
- Recorre todos los nodos del nivel cero (la raíz), luego todos los nodos del nivel uno, etc.
- En cada nivel los nodos se recorren de izquierda a derecha.



Recorrido de árboles: *breadth-first*

- Los recorridos ***depth-first*** están definidos de manera recursiva,
- Los recorridos ***breadth-first*** se definen como recorridos no-recursivos.
- Recorre todos los nodos del nivel cero (la raíz), luego todos los nodos del nivel uno, etc.
- En cada nivel los nodos se recorren de izquierda a derecha.

Camino: E,D,H,B,F,A,C,G

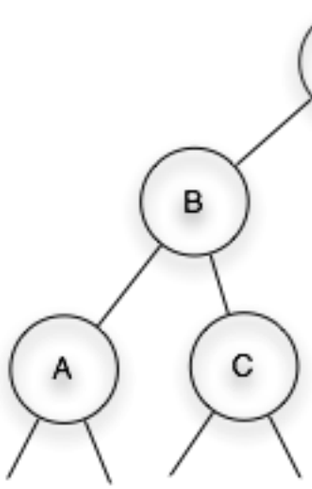


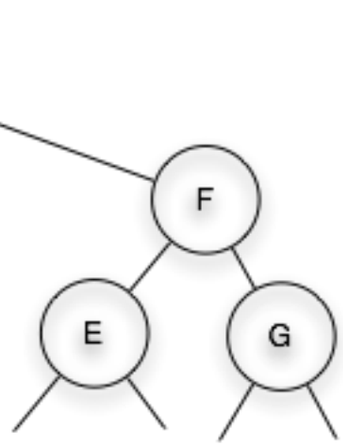
Recorrido de árboles: *breadth-first*

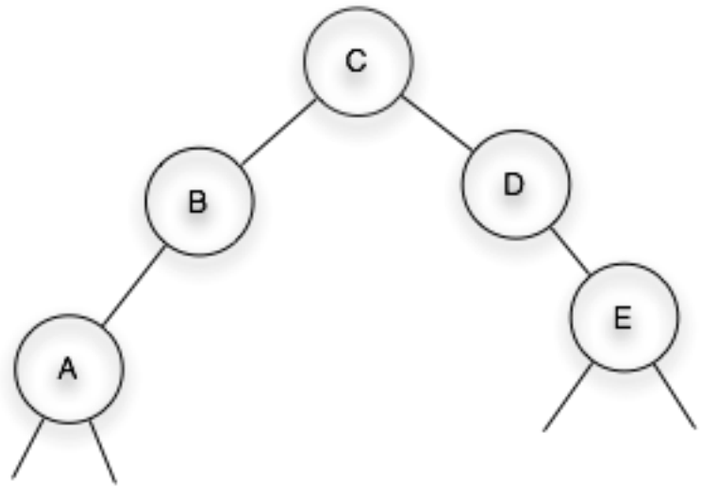
```
void traverse(link h, void visit(link))
  { QUEUE<link> q(max);
    q.put(h);
    while (!q.empty())
      {
        visit(h = q.get());
        if (h->l != 0) q.put(h->l);
        if (h->r != 0) q.put(h->r);
      }
  }
```

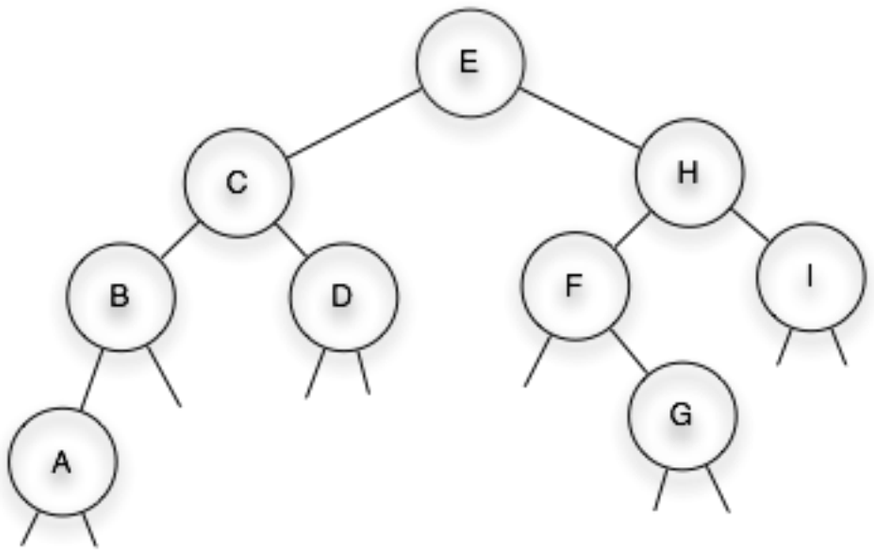
Ejercicio

- Tarea: Dar los caminos para pre-orden, orden, post-orden y orden por nivel para los siguientes arboles:







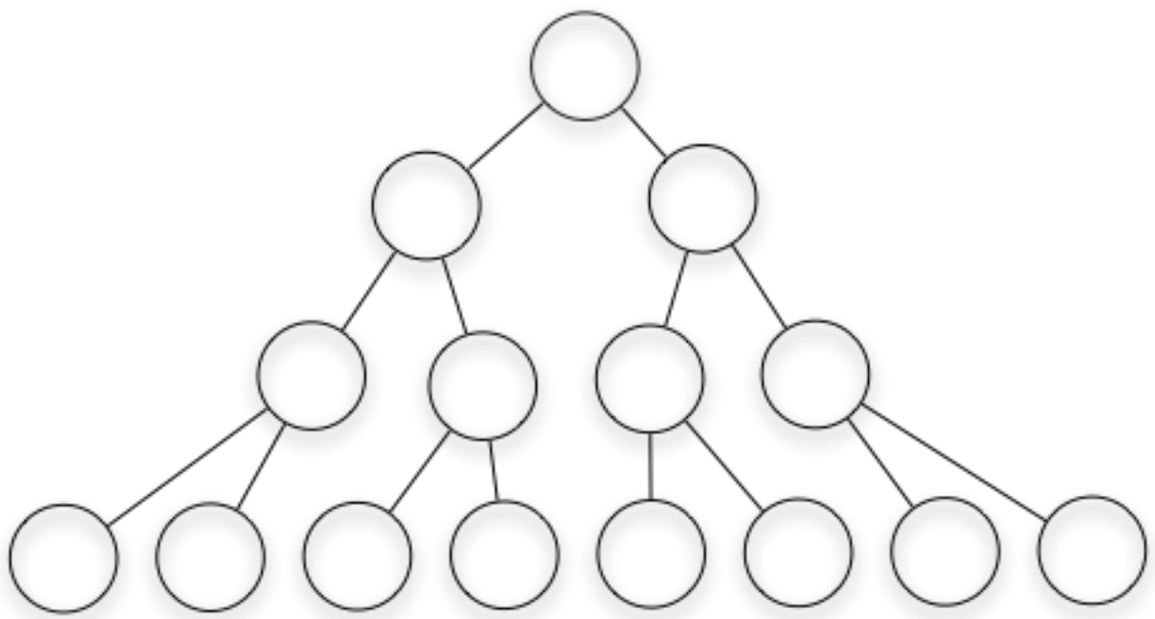


Otras propiedades de árboles binarios

- **Fullness**

  •*Un árbol binario se dice **perfecto** ssi cada nodo en el árbol tiene **dos** o **cero** hijos, y cada nodo que tiene cero hijos está en el nivel más bajo del árbol.*

  •Esta propiedad tiene sentido solamente para árboles con un número definido de hijos.



Otras propiedades de árboles binarios

- **Balance**

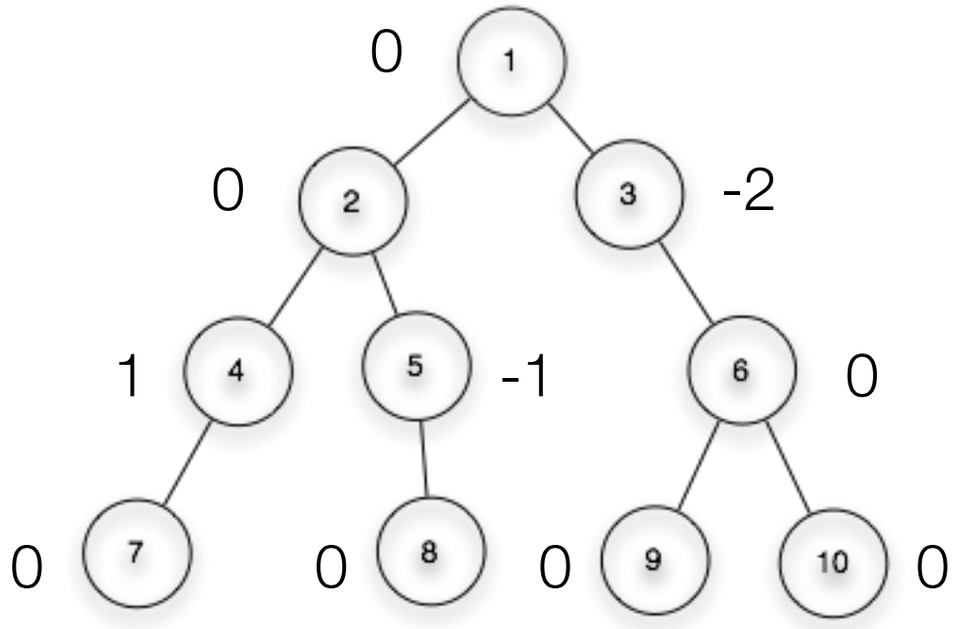
  •*Un árbol binario se dice **balanceado** si cada una de sus ramas es **casi** del mismo tamaño y profundidad que las demás.*

  • Para determinar si un nodo del árbol es balanceado, hay que encontrar el nivel máximo de altura de ambos hijos y si difieren en **más de un nivel**, el árbol **no** está balanceado.

  •Una función recursiva **depth()** nos puede dar la profundidad máxima del hijo izquierdo y del hijo derecho.

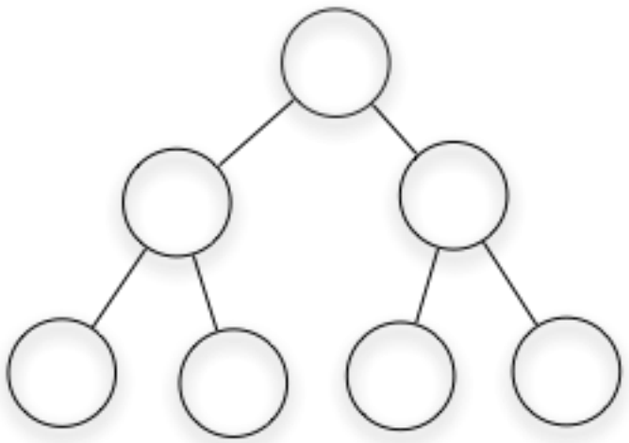
  •La diferencia de las profundidades de **todos los nodos del árbol** debe ser igual a ***-1, 0, 1*** para que el árbol sea balanceado.

Otras propiedades de árboles binarios



Implementaciones de árboles binarios: dinamico

- árbol dinamico



```
class BinaryTree
{
    public:

        typedef BinaryTree node;

        // ------------------------------------------------
        // Name:             BinaryTree::m_data
        // Description:      data stored in current tree node
        // ------------------------------------------------
        Item m_data;

        // ------------------------------------------------
        // Name:             BinaryTree::m_parent
        // Description:      pointer to the parent of the node
        // ------------------------------------------------
        node* m_parent;

        // ------------------------------------------------
        // Name:             BinaryTree::m_left
        // Description:      pointer to the left child of the node
        // ------------------------------------------------
        node* m_left;

        // ------------------------------------------------
        // Name:             BinaryTree::m_left
        // Description:      pointer to the left child of the node
        // ------------------------------------------------
        node* m_right;
};
```

métodos para un árbol binario

```
// ---------------------------------------------------------------
// Name:              BinaryTree::BinaryTree
// Description:       default constructor. creates a tree node.
// Arguments:         data to initialize the node with
// Return value:      none
// ---------------------------------------------------------------
BinaryTree( Item p_item );

// ---------------------------------------------------------------
// Name:              BinaryTree::~BinaryTree
// Description:       destructor. deletes all child nodes.
// Arguments:         none
// Return value:      none
// ---------------------------------------------------------------
~BinaryTree();
```

métodos para un árbol binario

Ejemplo:

- Forma simple de añadir hijos al árbol.

- si ya existe un hijo en el lugar de inserción, pone el apuntador al padre del hijo existente a cero, haciendo un nuevo árbol.

- agrega el nuevo nodo.

- hacer el nuevo nodo padre del nodo existente.

```
// ---------------------------------------------------------------
// Name:              BinaryTree::setLeft
// Description:       sets the left child pointer, rearranging parents
//                    as necessary. Existing left child is promoted to
//                    the root of its own tree, but beware, it is up to
//                    the client to remember where it is, or else there
//                    is a memory leak.
// Arguments:         pointer to a binary tree node.
// Return value:      none.
// ---------------------------------------------------------------
void setLeft( node* p_left );

// ---------------------------------------------------------------
// Name:              BinaryTree::setRight
// Description:       sets the right child pointer, rearranging parents
//                    as necessary. Existing right child is promoted to
//                    the root of its own tree, but beware, it is up to
//                    the client to remember where it is, or else there
//                    is a memory leak.
// Arguments:         pointer to a binary tree node.
// Return value:      none.
// ---------------------------------------------------------------
void setRight( node* p_right );
```

métodos para un árbol binario

```
// ---------------------------------------------------------------
// Name:              BinaryTree::preorder
// Description:       processes the tree using a pre-order traversal
// Arguments:         process: A function which takes a reference to
//                    a data type and performs an operation on it.
// Return value:      none
// ---------------------------------------------------------------
void preorder( void(*process)(Item& item))
```

```
// ---------------------------------------------------------------
// Name:              BinaryTree::postorder
// Description:       processes the tree using a post-order traversal
// Arguments:         process: A function which takes a reference to
//                    a data type and performs an operation on it.
// Return value:      none
// ---------------------------------------------------------------
void postorder( void(*process)(Item& item))
```

```
// ---------------------------------------------------------------
// Name:              BinaryTree::inorder
// Description:       processes the tree using a in-order traversal
// Arguments:         process: A function which takes a reference to
//                    a data type and performs an operation on it.
// Return value:      none
// ---------------------------------------------------------------
void inorder( void(*process)(Item& item))
```

métodos para un árbol binario

```
// ---------------------------------------------------------------
// Name:              BinaryTree::isLeft
// Description:       determines if node is a left subtree. Note that
//                    a result of false does NOT mean that it is a
//                    right child; it may be a root instead
// Arguments:         none
// Return value:      TRUE or FALSE
// ---------------------------------------------------------------
bool isLeft();

// ---------------------------------------------------------------
// Name:              BinaryTree::isRight
// Description:       determines if node is a right subtree. Note that
//                    a result of false does NOT mean that it is a
//                    left child; it may be a root instead
// Arguments:         none
// Return value:      TRUE or FALSE
// ---------------------------------------------------------------
bool isRight();

// ---------------------------------------------------------------
// Name:              BinaryTree::isRoot
// Description:       determines if a node is a root node
// Arguments:         none
// Return value:      TRUE or FALSE
// ---------------------------------------------------------------
bool isRoot();
```

métodos para un árbol binario

- Función recursiva
- Recursivamente a cada hijo hace regresar la cuenta de hijos.

```
// ---------------------------------------------------------------
// Name:              BinaryTree::count
// Description:       recursively counts all children
// Arguments:         none
// Return value:      the number of children.
// ---------------------------------------------------------------
int count();
```

métodos para un árbol binario

```
// ---------------------------------------------------------------
// Name:              BinaryTree::depth
// Description:       recursively determines the lowest relative depth
//                    of all its children
// Arguments:         none
// Return value:      the lowest depth of the current node
// ---------------------------------------------------------------
int depth();
```

```
int depth()
{
    int left = -1;                  // clear left;
    int right = -1;                 // clear right;
    if ( m_left != 0 )              // if the left child exists
        left = m_left->depth();     // update the left depth
    if (m_right != 0 )              // if the right child exists
        right = m_right->depth();   // update the right depth
    if ( left > right )
        return left + 1;
    return right + 1;
}
```

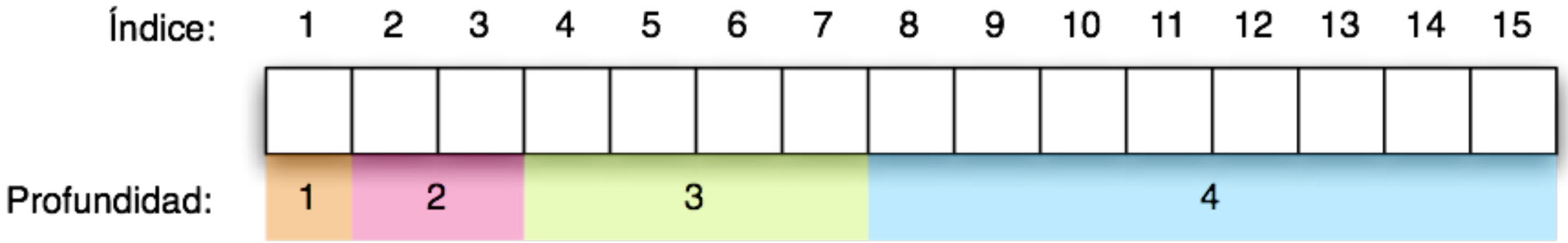
Implementaciones de árboles binarios: dinámico

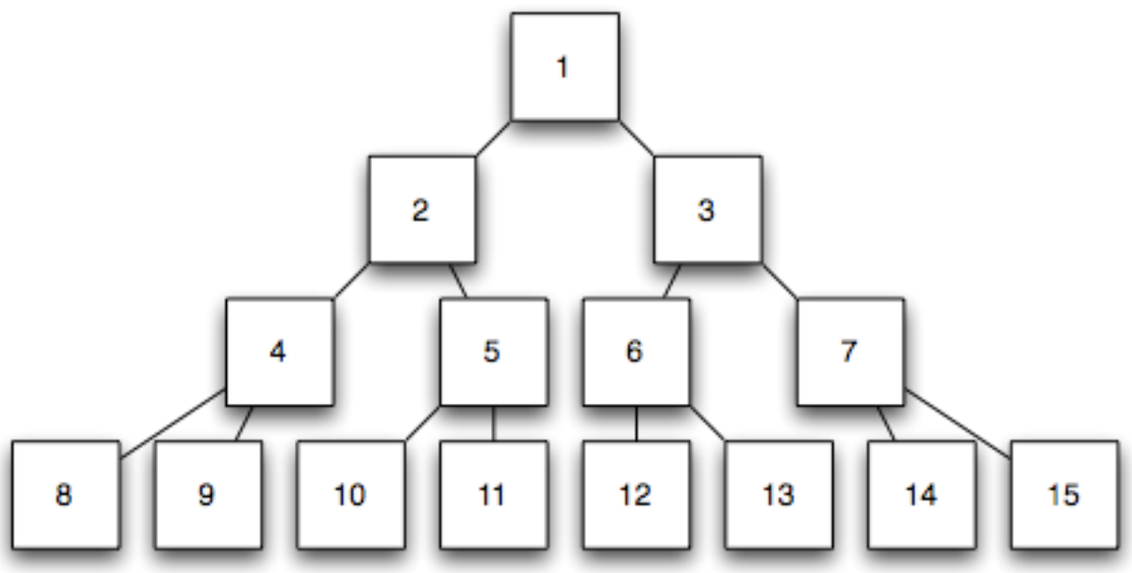
- **Ventajas:**
  - el tamaño máximo del árbol está limitado al total de la memoria del sistema.

- **Desventajas:**
  - cada nodo requiere dos apuntadores a sus hijos y un tercero para su padre. Si el árbol es de tipos de datos simples (enteros, flotantes o caracteres), cada nodo toma entre 3 y 12 veces más que el tamaño del tipo de datos.

Implementaciones de árboles binarios: arreglos

- Arreglos





Implementaciones de árboles binarios: arreglos

- Los índices se numeran a partir de uno.
- El hijo ***izquierdo*** de cualquier índice ***i*** está en ***i*2**.
- El hijo ***derecho*** de cualquier índice ***i*** está en ***(i*2)+1***.
- El ***padre*** de cada índice ***i*** está en ***i/2 (división entera)***.

Implementaciones de árboles binarios: arreglos

- **Ventajas:**
  - Relativamente rápidos de recorrer
  - Fáciles de transmitir por interfases seriales (internet, disco externo, etc.)
  - Toma menos espacio que la implementación con listas...
- **Desventajas:**
  - El arreglo tendrá tamaño fijo de $(2^{d+1})-1$, donde ***d*** representa el máximo de profundidad de un nodo dado del árbol.
  - Solo si el árbol es lleno y balanceado no desperdicia memoria.
  - Agregar nodos que van más allá de la memoria reservada requiere re-dimensionar el arreglo.

Ejemplos para un árbol binario

```
void printnode(Item x, int h)
  { for (int i = 0; i < h; i++) cout << "   ";
    cout << x << endl;
  }
void show(link t, int h)
  {
    if (t == 0) { printnode('*', h); return; }
    show(t->r, h+1);
    printnode(t->item, h);
    show(t->l, h+1);
  }
```

Debugger

Instruccione importantes

- Start debugger, F8
- Next Line F7
- Run to Cursor, F4
- Togle breakpoint, F5 o con el mouse en el marco.
- Step into, Shift-F7
- Step out, Shift-Ctrl-F7

Watches

- Menu: **Debug->Edit Watches-> add** (podemos agregar tambien expresiones del tipo *"i+j+k/2"*)

Probrar llamado de funciones

- Menu: **Debug->Send user command to debugger** (para ejecutar la funcion *f* con parametros *a,5* tecleamos *"call f(a,5)"*)