Trees and hierarchical orders, ordered trees, Search trees

Sudipta Roy

Jio Institute, Ulwe, India

Tree Definition

- Here is the recursive definition of an (unordered) tree:
  - A *tree* is a pair (*r*, *S*), where *r* is a node and *S* is a set of disjoint trees, none of which contains *r*.
    - The node *r* is called the *root* of the tree *T*, and the elements of the set *S* are called its *subtrees*.
    - The set *S*, of course, may be empty.
    - The elements of a tree are called its *nodes.*

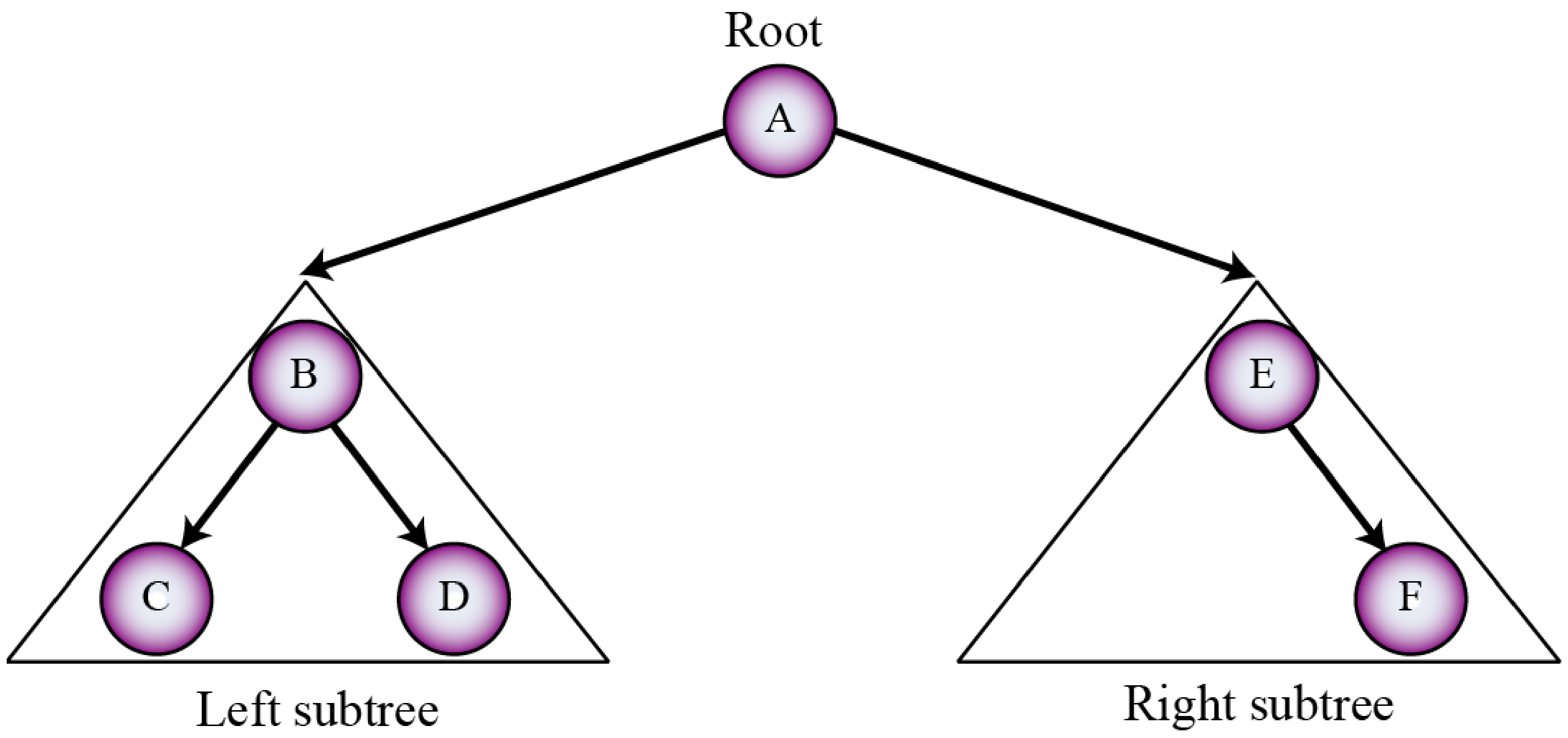
Root, Parent and Children

- If $T = (x, S)$ is a tree, then
  - x is the root of T and
  - S is its set of subtrees $S = \{T_1, T_2, T_3, \ . \ . \ ., T_n\}$.
- Each subtree T_j is itself a tree with its own root r_j .
- In this case, we call the node r the *parent* of each node r_j, and we call the r_j the *children* of r. In general.

Binary Trees

Binary Trees

- A binary tree is either the empty set or a triple T = (x, L, R), where x is a node and L and R are disjoint binary trees, neither of which contains x.
- The node x is called the root of the tree T, and the subtrees L and R are called the left subtree and the right subtree of T rooted at x.



Binary Trees (Another way)

- A ***binary tree*** is a a collection of nodes that consists of the ***root*** and other subsets to the root points, which are called the ***left*** and ***right subtrees***.

- Every ***parent*** node on a binary tree can have up to two ***child*** nodes (roots of the two subtrees); any more children and it becomes a general tree.

- A node that has no children is called a ***leaf*** node.

Abstract Data Type Binary_Tree

structure *Binary_Tree*(abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:

for all *bt*, *bt1*, *bt2* ∈ *BinTree*, *item* ∈ *element*

Bintree **Create**()::= creates an empty binary tree

Boolean **IsEmpty**(*bt*)::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE*

BinTree **MakeBT**(*bt1*, *item*, *bt2*)::= return a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item*

Bintree Lchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the left subtree of *bt*

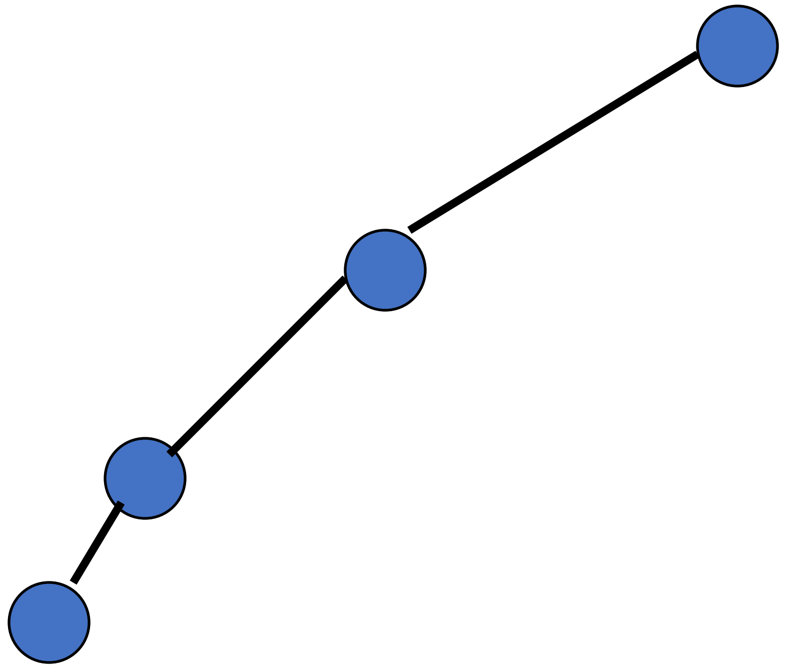
element Data(*bt*)::= if (IsEmpty(*bt*)) return error
else return the data in the root node of *bt*

Bintree Rchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the right subtree of *bt*

Binary Trees – Properties, Representation, Implementation

Minimum Number Of Nodes

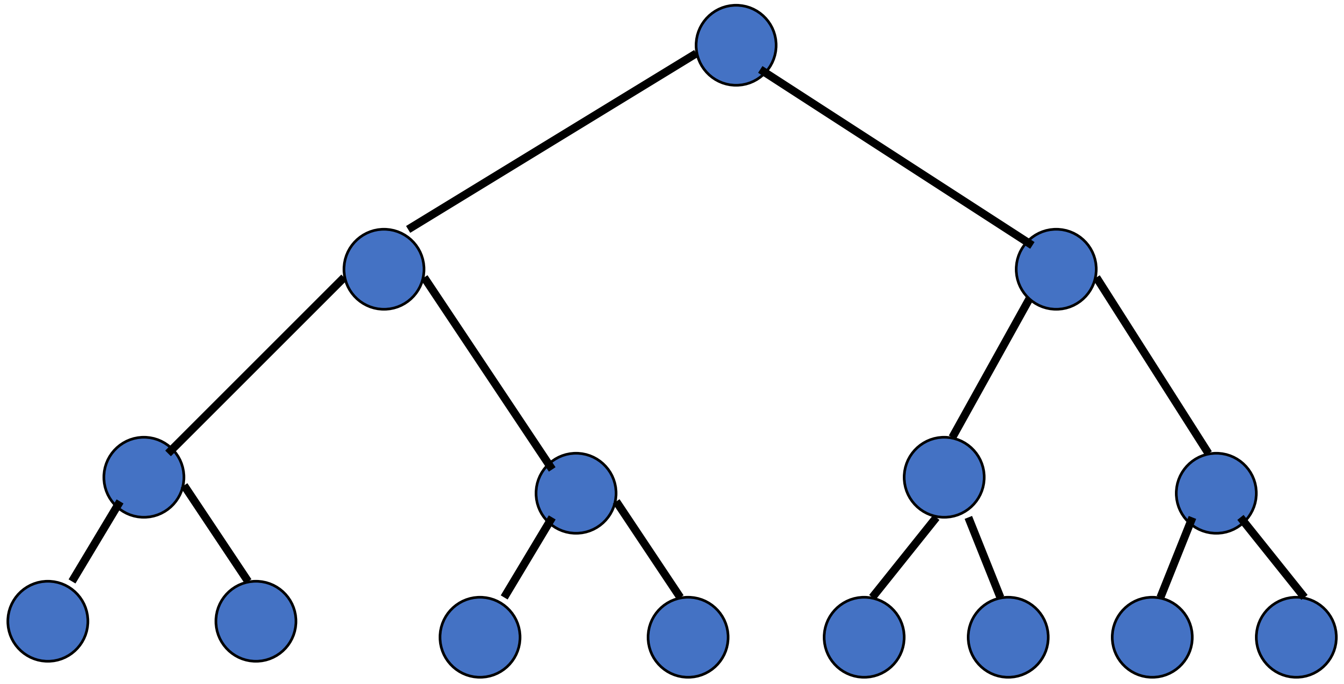
- Minimum number of nodes in a binary tree whose height is h.
- At least one node at each of first h levels.



minimum number of nodes is h

Maximum Number Of Nodes

- All possible nodes at first h levels are present.



Maximum number of nodes

$= 1 + 2 + 4 + 8 + \ldots + 2^{h-1} = 2^h - 1$

Number Of Nodes & Height

- Let n be the number of nodes in a binary tree whose height is h.
- $h <= n <= 2^h – 1$
- $\log_2(n+1) <= h <= n$

Relations between Number of Leaf Nodes and Nodes of Degree 2

For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0=n_2+1$

proof:

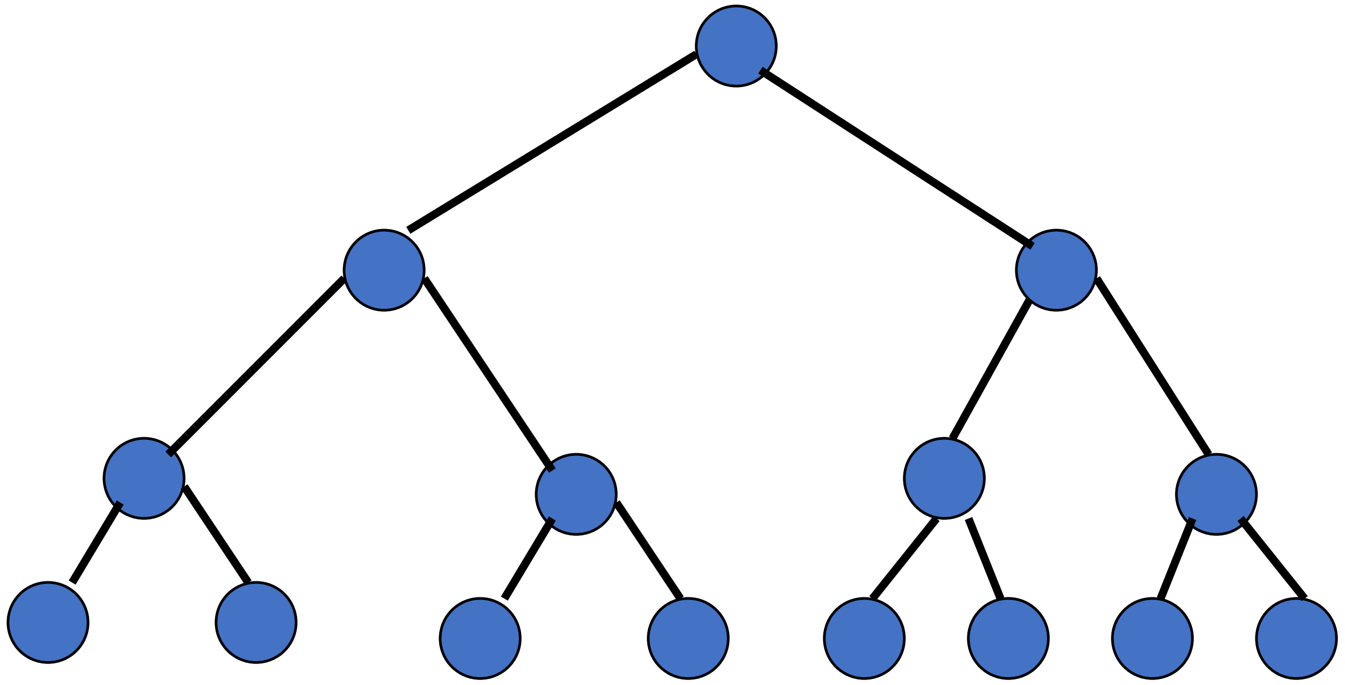
Let n and B denote the total number of nodes & branches in T.

Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.

$n= n_0+n_1+n_2$, $B+1=n$, $B=n_1+2n_2$ ==> $n_1+2n_2+1= n$,
$n_1+2n_2+1= n_0+n_1+n_2$ ==> $n_0=n_2+1$

Full Binary Tree

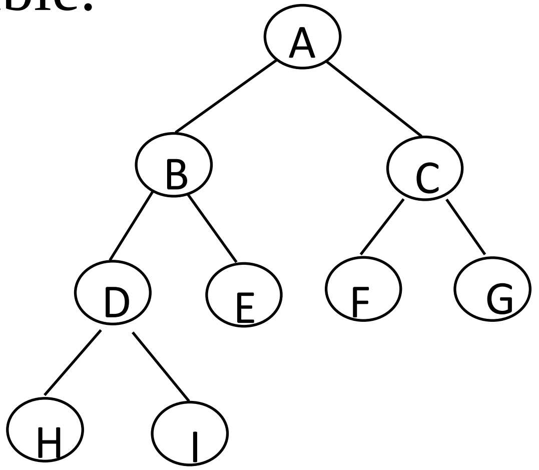
- A full binary tree of a given height h has $2^h – 1$ nodes.



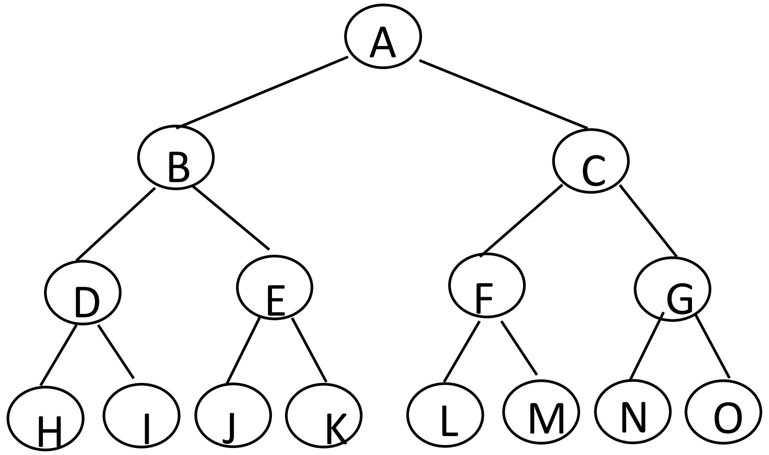
Height 4 full binary tree.

Full BT VS Complete BT

- A full binary tree of depth *k* is a binary tree of depth ***k* having 2^k -1** nodes, *k*>=0.
- A binary tree with *n* nodes and depth *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.
  - Except last level the BT will be full and in the last level all possible nodes are left as possible.



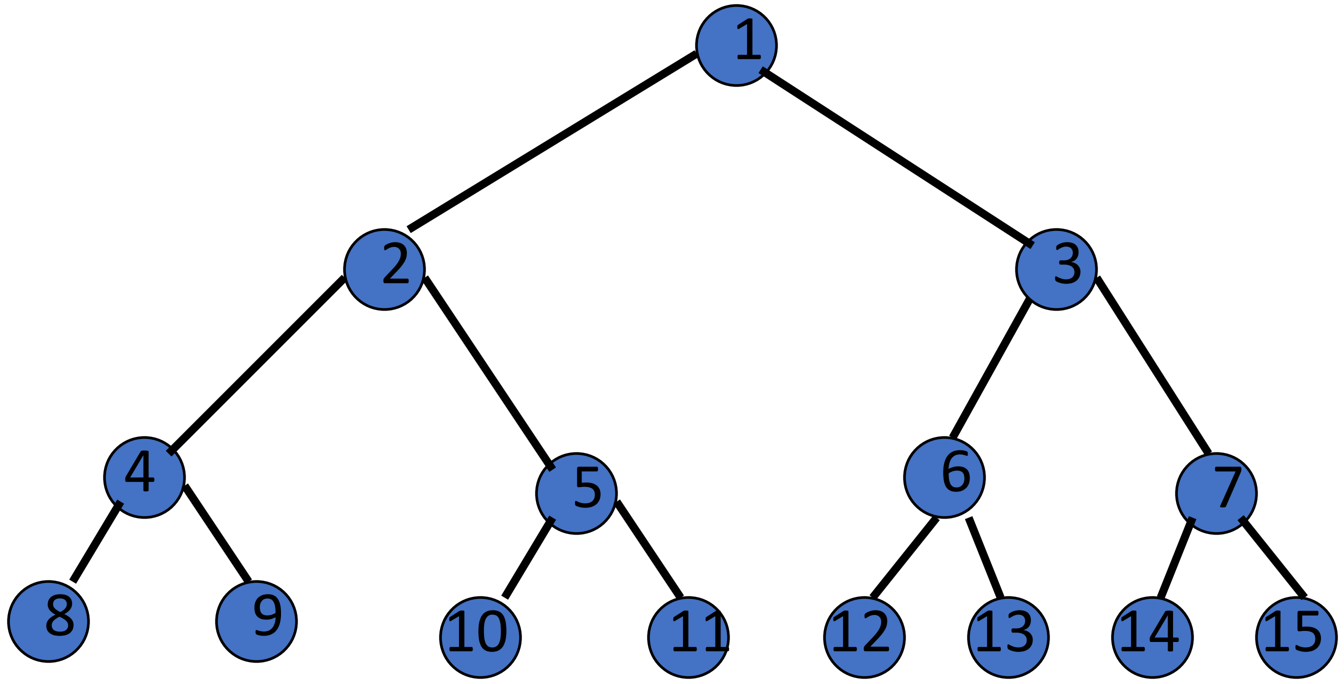
Complete binary tree



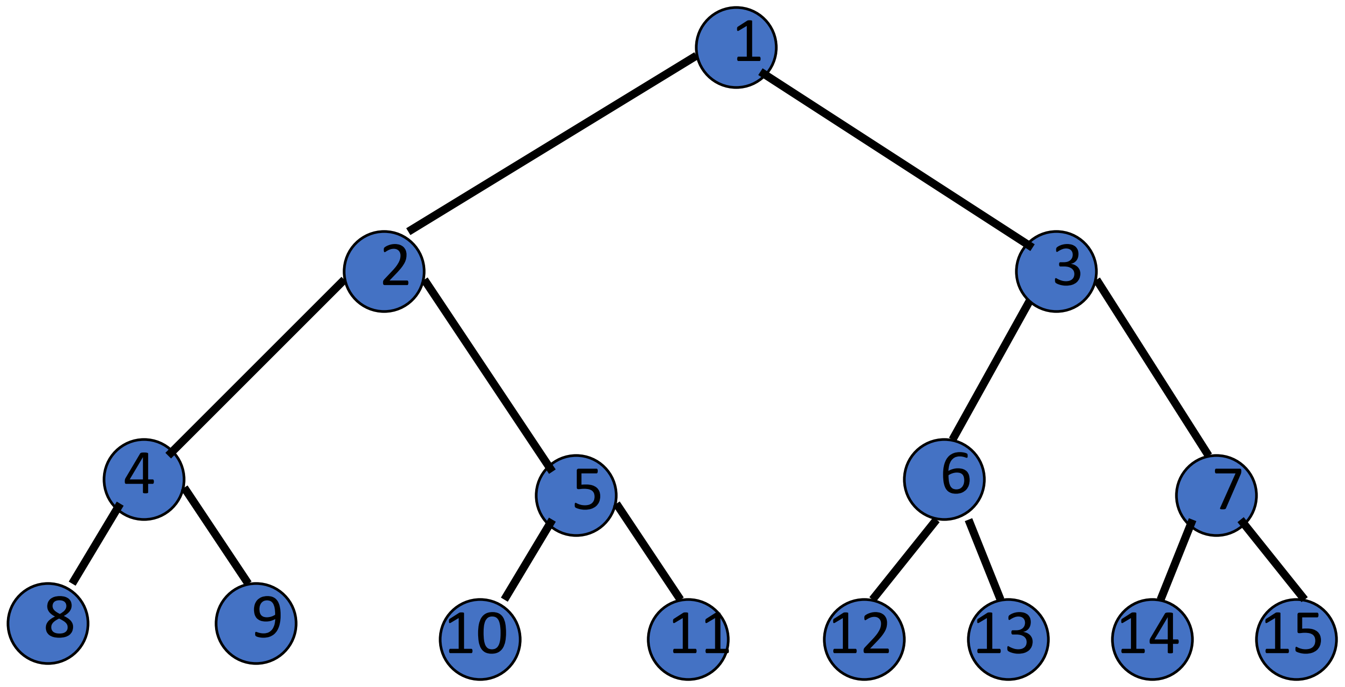
Full binary tree of depth 4

Numbering Nodes In A Full Binary Tree

- Number the nodes 1 through $2^h - 1$.
- Number by levels from top to bottom.
- Within a level number from left to right.

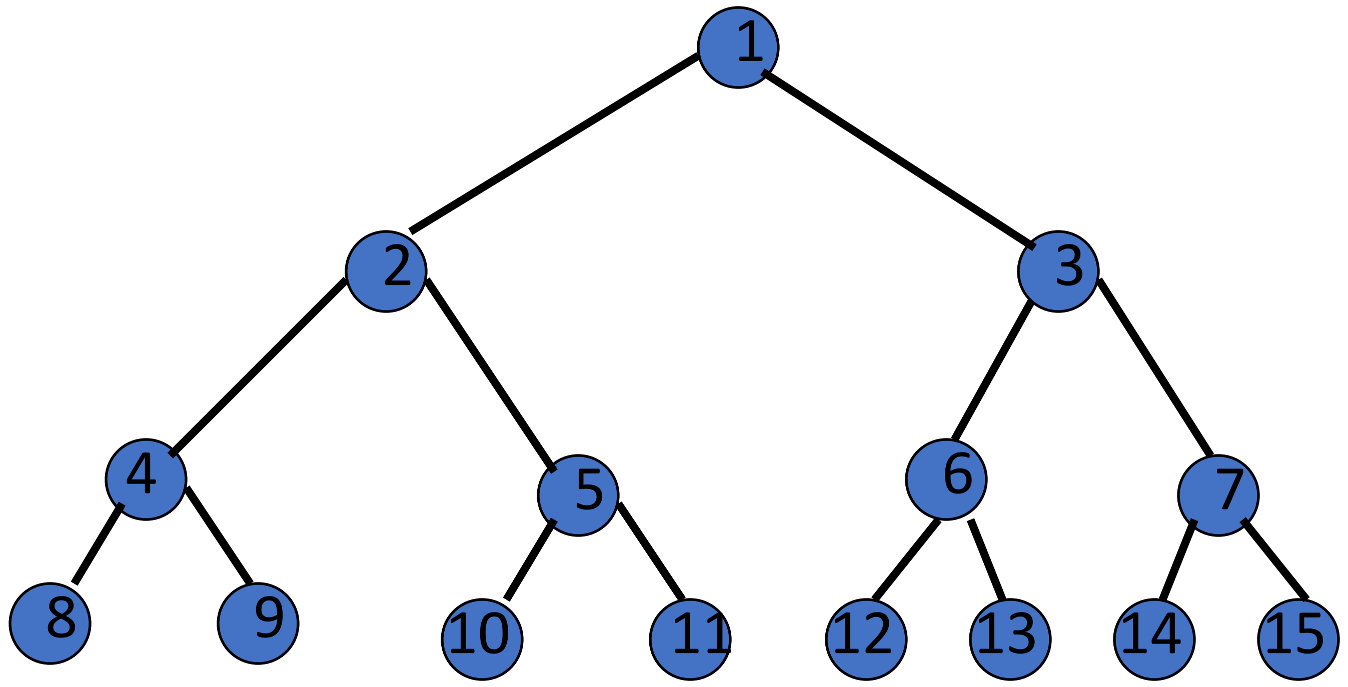


Node Number Properties



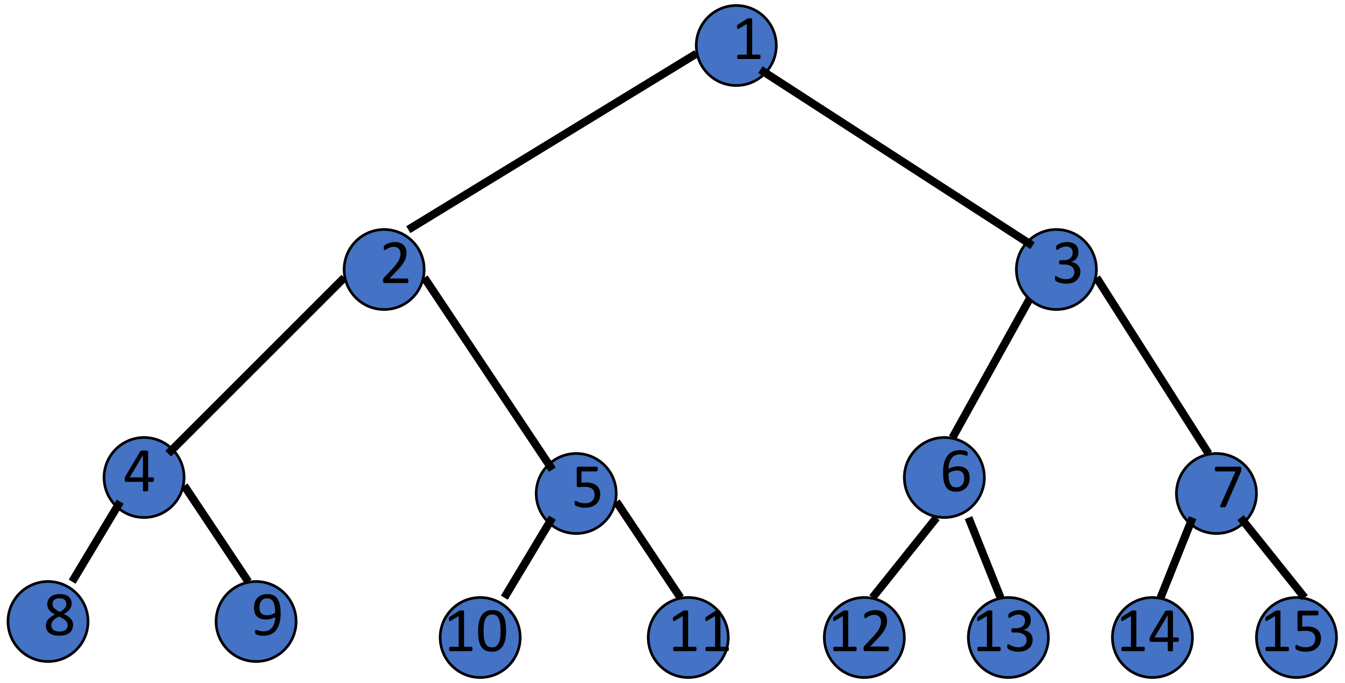
- Parent of node i is node i / 2, unless i = 1.
- Node 1 is the root and has no parent.

Node Number Properties



- Left child of node i is node 2i, unless 2i > n, where n is the number of nodes.
- If 2i > n, node i has no left child.

Node Number Properties



- Right child of node i is node 2i+1, unless 2i+1 > n, where n is the number of nodes.
- If 2i+1 > n, node i has no right child.

Binary Tree Representation

Array representation
Linked representation

Binary Tree Representations

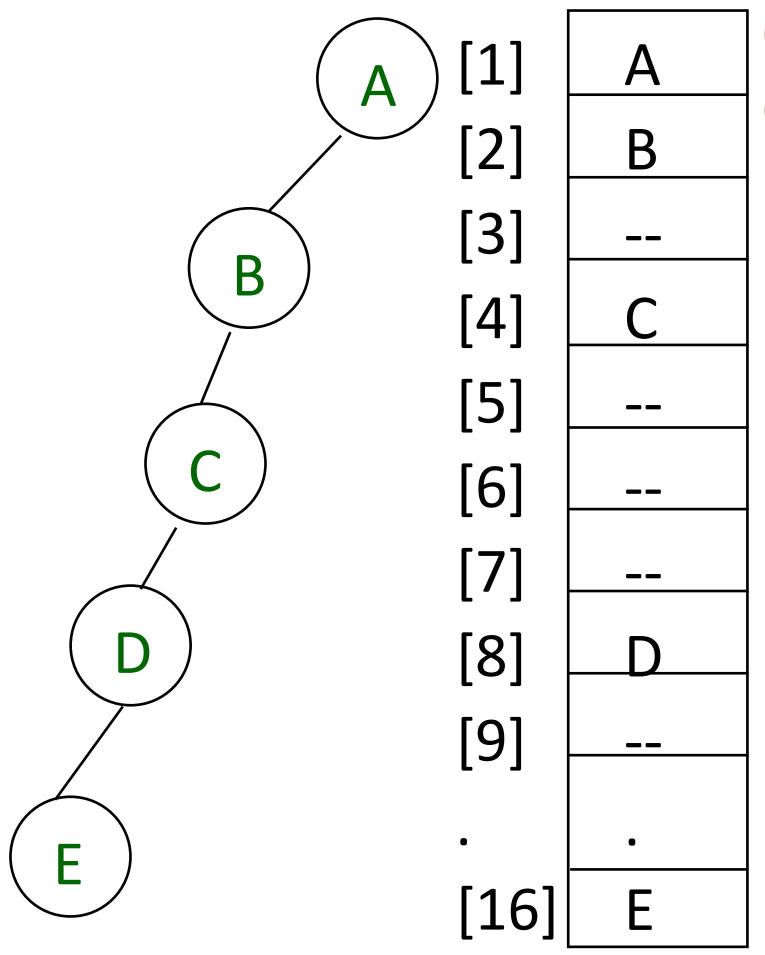
If a complete binary tree with *n* nodes (depth = log *n* + 1) is represented sequentially, then for any node with index *i*, 1<=*i*<=*n*, we have:

- *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent.
- *left_child*(*i*) ia at **2*i*** if 2*i*<=*n*. If 2*i*>n, then *i* has no left child.
- *right_child*(*i*) ia at **2*i*+1** if 2*i* +1 <=*n*. If 2*i* +1 >n, then *i* has no right child.

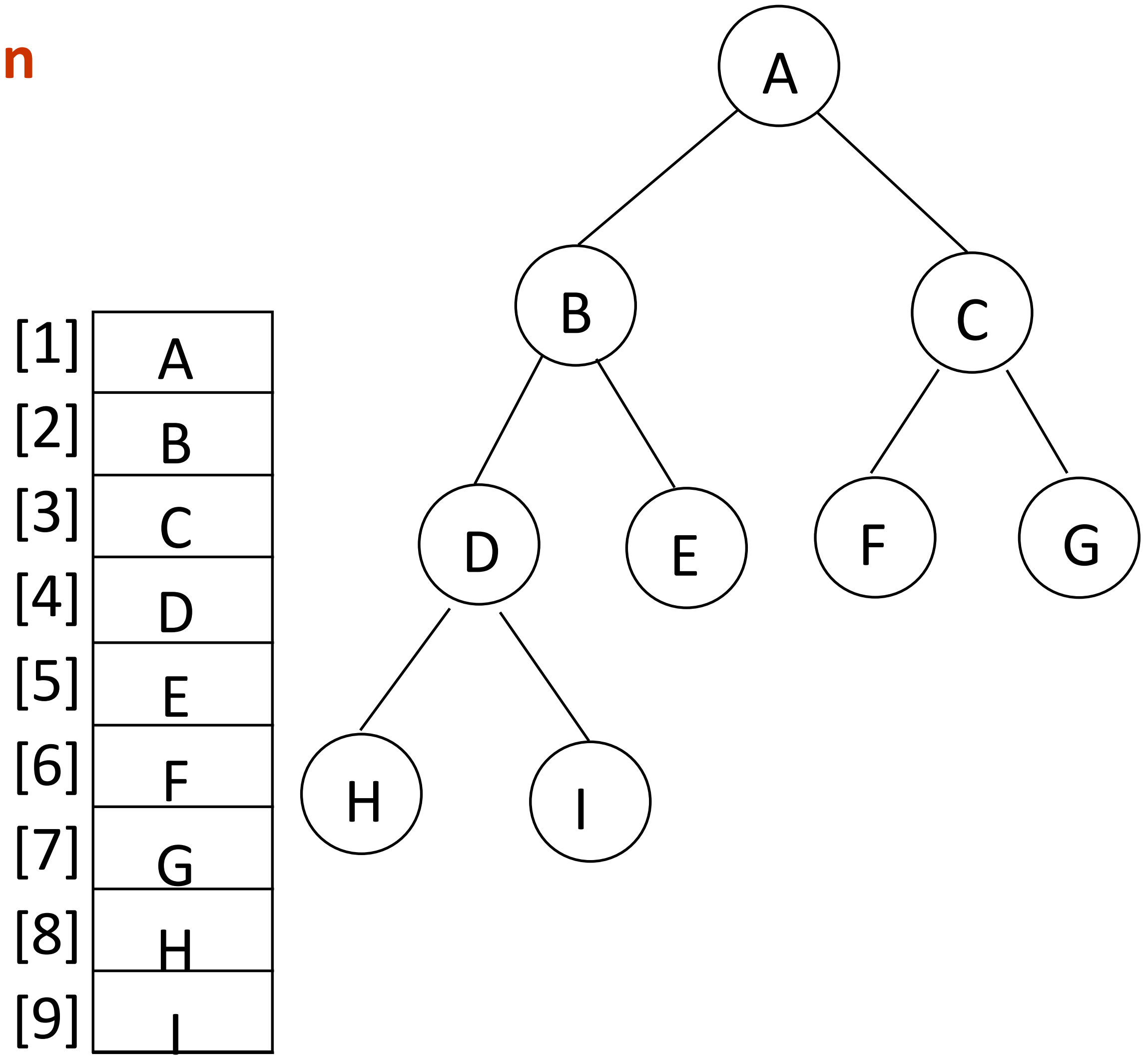
Case 2: (0—n-1)

```
if (say)father=p;  then
left_son=(2*p)+1;  and
right_son=(2*p)+2;
```

Sequential Representation

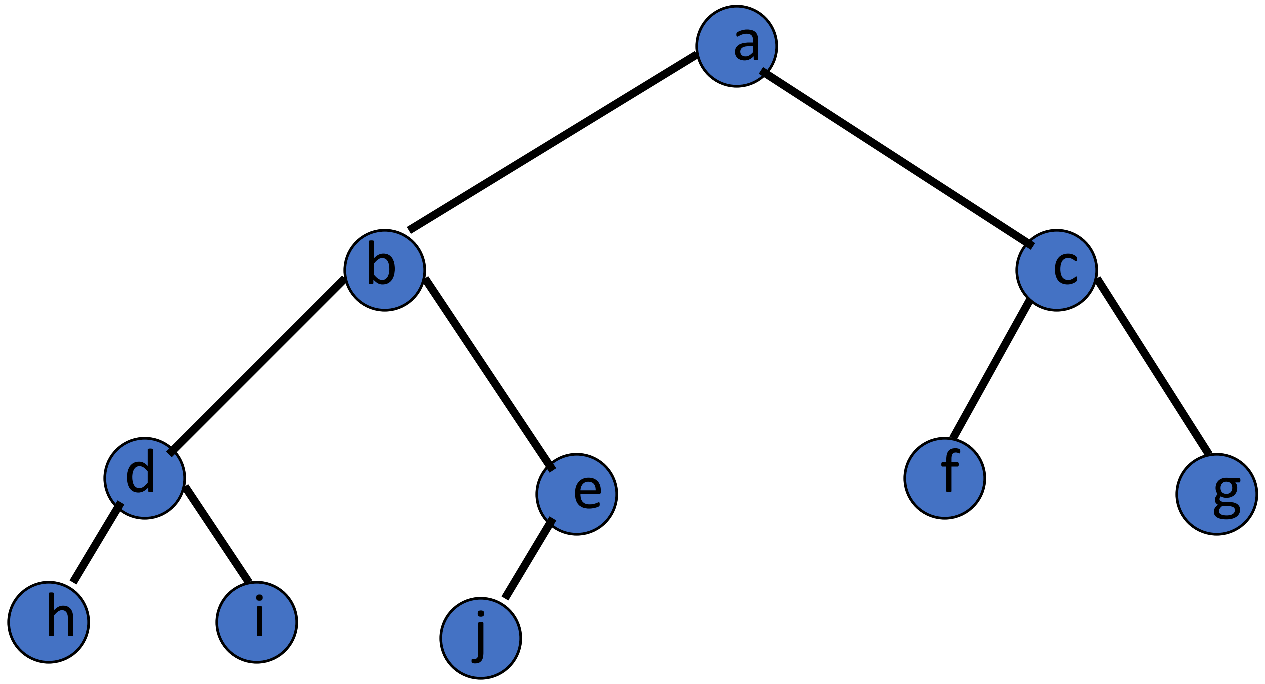


(1) waste space
(2) insertion/deletion problem



Array Representation

- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in tree[i].



tree[]	a	b	c	d	e	f	g	h	i	j

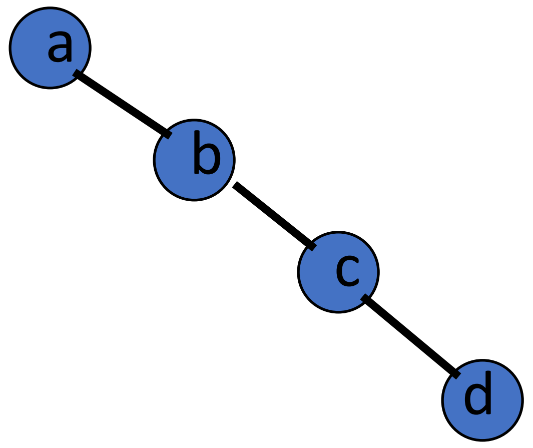
Array Representation

```
int complete_node = 15;

// array to store the tree
char tree[] = {'\0', 'D', 'A', 'F', 'E', 'B', 'R', 'T', 'G', 'Q', '\0', '\0', 'V', '\0', 'J', 'L'};
```

```
int get_right_child(int index)
{
    // node is not null
    // and the result must lie within the number of nodes for a
complete binary tree
    if(tree[index]!='\0' && ((2*index)+1)<=complete_node)
        return (2*index)+1;
    // right child doesn't exist
    return -1;
}
```

```
int get_left_child(int index)
{
    // node is not null
    // and the result must lie within the number of nodes
for a complete binary tree
    if(tree[index]!='\0' && (2*index)<=complete_node)
        return 2*index;
    // left child doesn't exist
    return -1;
}
```

Right-Skewed Binary Tree



tree[]	a	-	b	-	-	-	c	-	-	-	-	-	-	-	d

- An n node binary tree needs an array whose length is between $n+1$ and 2^n.

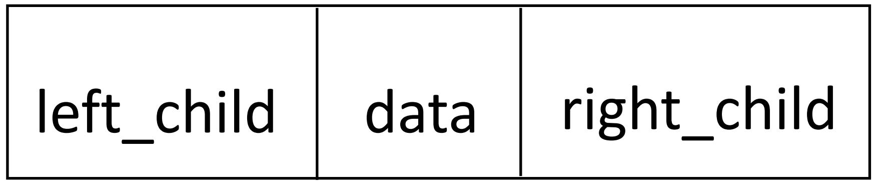
Linked Representation

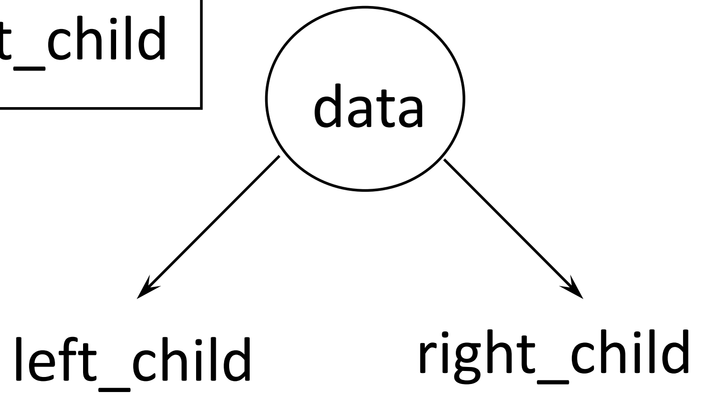
- Each binary tree node is represented as an object whose data type is BinaryTreeNode.
- The space required by an n node binary tree is n * (space required by one node).

Linked Representation

```
// Create a new Node
struct node* createNode(value) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;
```

```
struct node {
  int item;
  struct node* left;
  struct node* right;
};
```

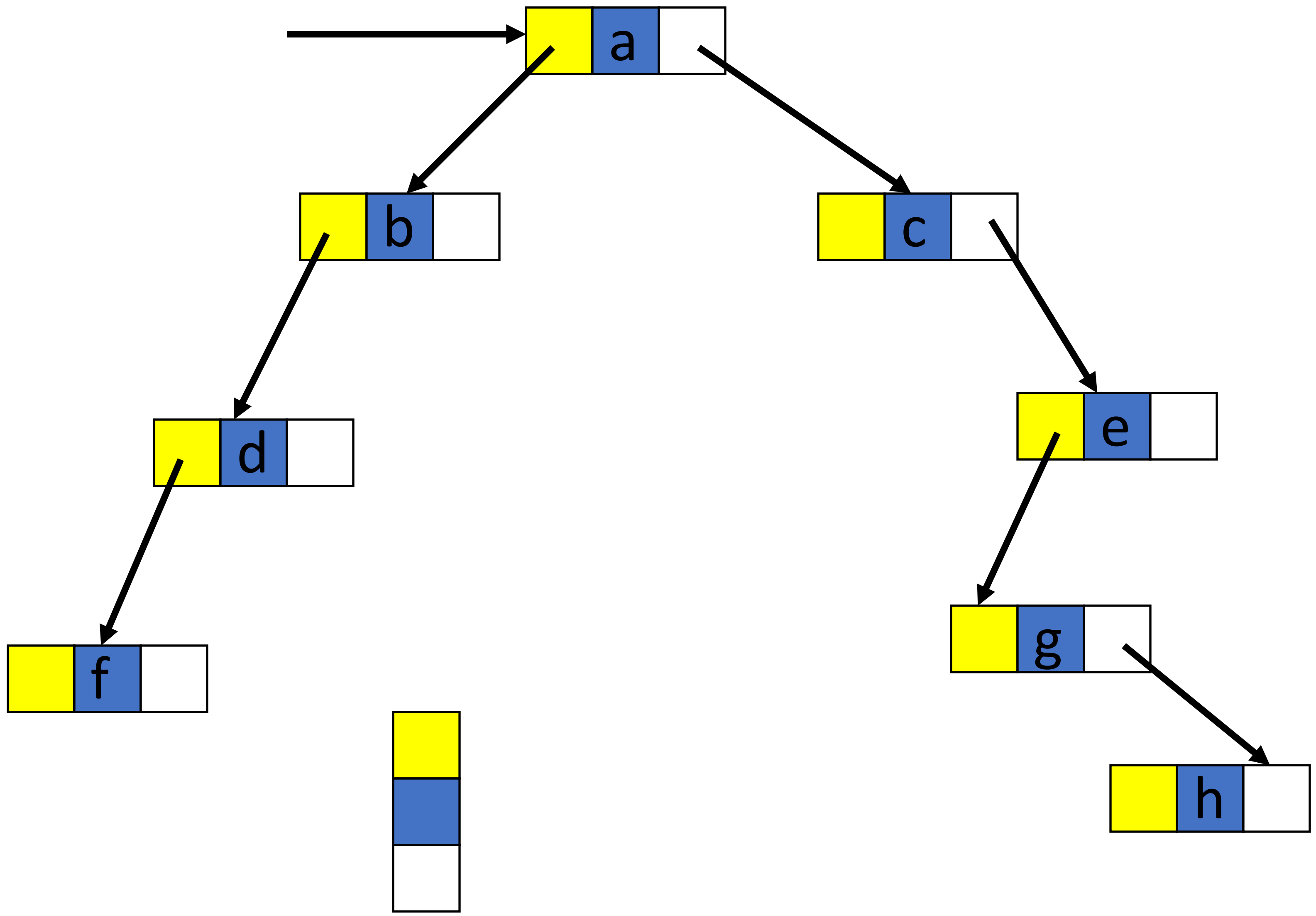




Python →

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

Linked Representation Example



Binary Tree implementation

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node* left;
    struct node* right;
};

/* newNode() allocates a new node
with the given data and NULL left
and right pointers. */

struct node* newNode(int data)
{
    // Allocate memory for new node
    struct node* node
        = (struct node*)malloc(sizeof(struct node));

    // Assign data to this node
    node->data = data;

    // Initialize left and
    // right children as NULL
    node->left = NULL;
    node->right = NULL;
    return (node);
}
```

```c
int main()
{
    /*create root*/
    struct node* root = newNode(1);

    /* following is the tree after above statement
        1
       /  \
     NULL NULL
    */
    root->left = newNode(2);
    root->right = newNode(3);

    /* 2 and 3 become left and right children of 1
            1
           / \
          2   3
         / \ / \
    NULL NULL NULL NULL
    */

    root->left->left = newNode(4);

    /* 4 becomes left child of 2
      1
     / \
    2   3
  /   \ / \
4 NULL NULL NULL
/   \
NULL NULL
*/
    getchar();
    return 0;
}
```

Binary Tree implementation

```
# Python program to introduce Binary Tree A class that represents an
individual node in a Binary Tree
class Node:
    def __init__(self, key):
        self.left = None
          self.right = None
          self.val = key

if __name__ == '__main__':
    # Create root
    root = Node(1)
    ''' following is the tree after above statement
    1
    / \
    None None'''
root.left = Node(2)
root.right = Node(3)

''' 2 and 3 become left and right children of 1
1
/ \
2 3
/ \ / \
None None None None'''

root.left.left = Node(4)
'''4 becomes left child of 2
1
/ \
2 3
/ \ / \
4 None None None
/ \
None None'''
```

Binary Trees Traversal

Binary Tree Traversal

- Many binary tree operations are done by performing a traversal of the binary tree.
- In a traversal, each element of the binary tree is visited exactly once.
- During the visit of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

Binary Tree Traversals

- Let L, V, and R stand for moving **left**, **visiting** the node, and moving **right**.
- There are six possible combinations of traversal
  - LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain
  - LVR, LRV, VLR
  - **inorder**, **postorder**, **preorder**

V→Root

Tree Traversal

- There are three common ways to traverse a tree:
  - **Preorder**: Visit the root, traverse the left subtree (preorder) and then traverse the right subtree (preorder)
  - **Postorder**: Traverse the left subtree (postorder), traverse the right subtree (postorder) and then visit the root.
  - **Inorder**: Traverse the left subtree (in order), visit the root and the traverse the right subtree (in order).

Using Linked List

```
// Preorder traversal
void preorderTraversal(struct node* root)
{
  if (root == NULL)
              return;
  printf("%d ", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}
```

```
void preorder(int index)
{
   // checking for valid index and null node
   if(index>0 && tree[index]!='\0')
   {
      printf(" %c ",tree[index]); // visiting root
      preorder(get_left_child(index)); //visiting left subtree
      preorder(get_right_child(index)); //visiting right subtree
   }
}
```

Using ARRAY

Tree Traversal

- There are three common ways to traverse a tree:
  - **Preorder**: Visit the root, traverse the left subtree (preorder) and then traverse the right subtree (preorder)
  - **Postorder**: Traverse the left subtree (postorder), traverse the right subtree (postorder) and then visit the root.
  - **Inorder**: Traverse the left subtree (in order), visit the root and the traverse the right subtree (in order).

Using Linked List

```
// Postorder traversal
void postorderTraversal(struct node* root)
{
  if (root == NULL)
            return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ", root->item);
}
```

```
void postorder(int index)
{
   // checking for valid index and null node
   if(index>0 && tree[index]!='\0')
   {
      postorder(get_left_child(index)); //visiting left subtree
      postorder(get_right_child(index)); //visiting right
subtree
      printf(" %c ",tree[index]); //visiting root
   }
}
```

Using ARRAY

Tree Traversal

- There are three common ways to traverse a tree:
  - **Preorder**: Visit the root, traverse the left subtree (preorder) and then traverse the right subtree (preorder)
  - **Postorder**: Traverse the left subtree (postorder), traverse the right subtree (postorder) and then visit the root.
  - **Inorder**: Traverse the left subtree (in order), visit the root and the traverse the right subtree (in order).

Using Linked List

```
// Inorder traversal
void inorderTraversal(struct node* root)
{
        if (root == NULL)
            return;
        inorderTraversal(root->left);
        printf("%d ->", root->item);
        inorderTraversal(root->right);
}
```

```
void inorder(int index)
{
  // checking for valid index and null node
  if(index>0 && tree[index]!='\0')
  {
    inorder(get_left_child(index)); //visiting left subtree
    printf(" %c ",tree[index]); //visiting root
    inorder(get_right_child(index)); // visiting right subtree
  }
}
```

Using ARRAY

Binary Tree Traversal Methods

- **Preorder**
- **Inorder**
- **Postorder**
- Level order

Algorithm for Inorder Traversal of Binary Tree

The algorithm for inorder traversal is shown as follows:

Input ?

```
Inorder(root):
1.Follow step 2 to 4 until root != NULL
2.Inorder (root -> left)
3.Write root -> data
4.Inorder (root -> right)
5.End loop
```

Output?

Tree Traversal

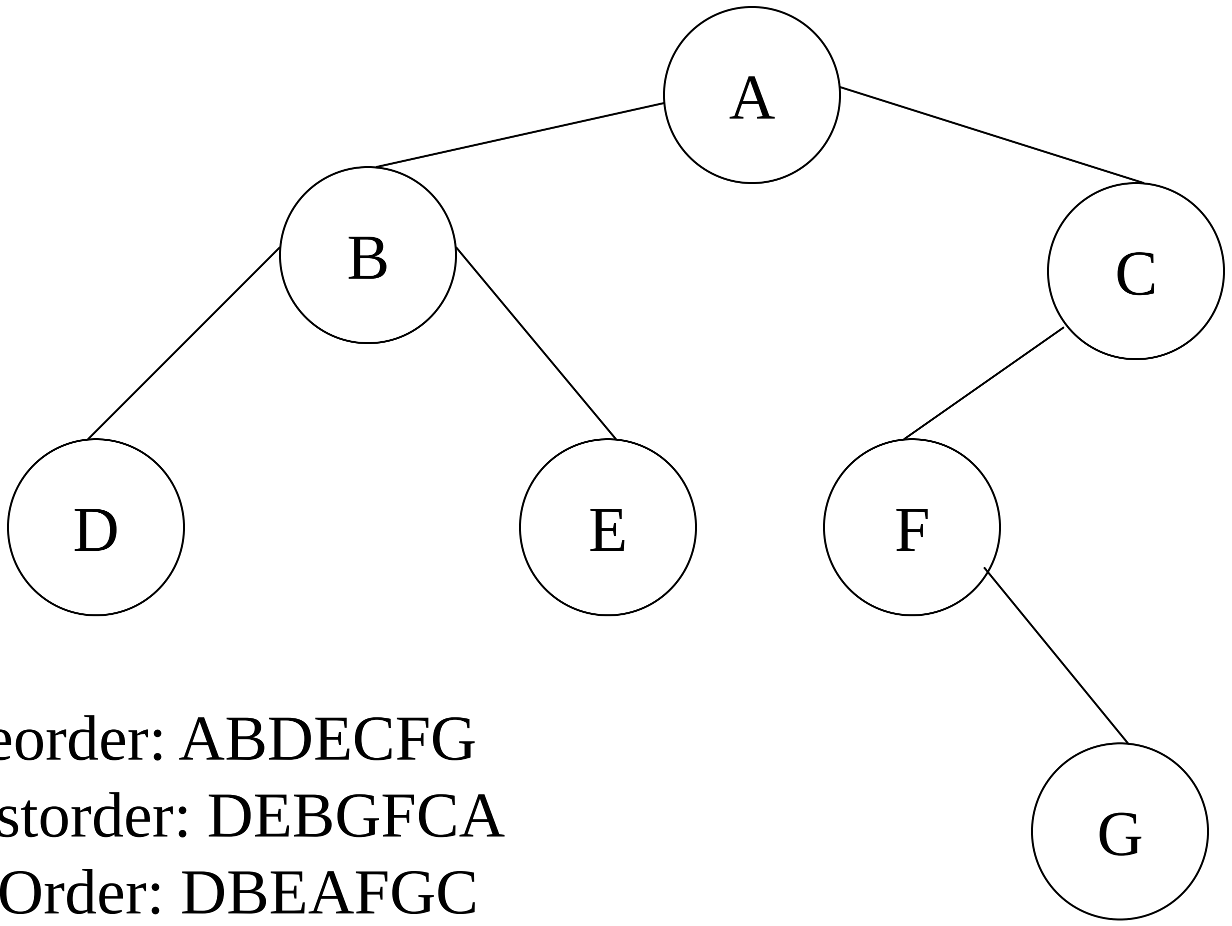
```
# Python3 program to for tree traversals A class that represents an individual
node in a Binary Tree


class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


# A function to do postorder tree traversal
def printPostorder(root):

    if root:

        # First recur on left child
        printPostorder(root.left)

        # the recur on right child
        printPostorder(root.right)

        # now print the data of node
        print(root.val),


# Driver code
if __name__ == "__main__":
  root = Node(1)
  root.left = Node(2)
  root.right = Node(3)
  root.left.left = Node(4)
  root.left.right = Node(5)


  # Function call
  print "\nPostorder traversal of binary tree is"
  printPostorder(root)
```

```
int main() {
  struct node* root = create(1);
  insertLeft(root, 4);
  insertRight(root, 6);
  insertLeft(root->left, 42);
  insertRight(root->left, 3);
  insertLeft(root->right, 2);
  insertRight(root->right, 33);

  printf("Traversal of the inserted binary tree \n");
  printf("Inorder traversal \n");
  inorderTraversal(root);

  printf("\nPreorder traversal \n");
  preorderTraversal(root);


  printf("\nPostorder traversal \n");
  postorderTraversal(root);

}
```

```
/ Create a new Node
struct node* create(int value) {
  struct node* newNode = malloc(sizeof(struct
node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}

// Insert on the left of the node
struct node* insertLeft(struct node* root, int value)
{
  root->left = create(value);
  return root->left;
}

// Insert on the right of the node
struct node* insertRight(struct node* root, int
value) {
  root->right = create(value);
  return root->right;
}
```

```
struct node {
  int item;
  struct node* left;
  struct node* right;
};
```

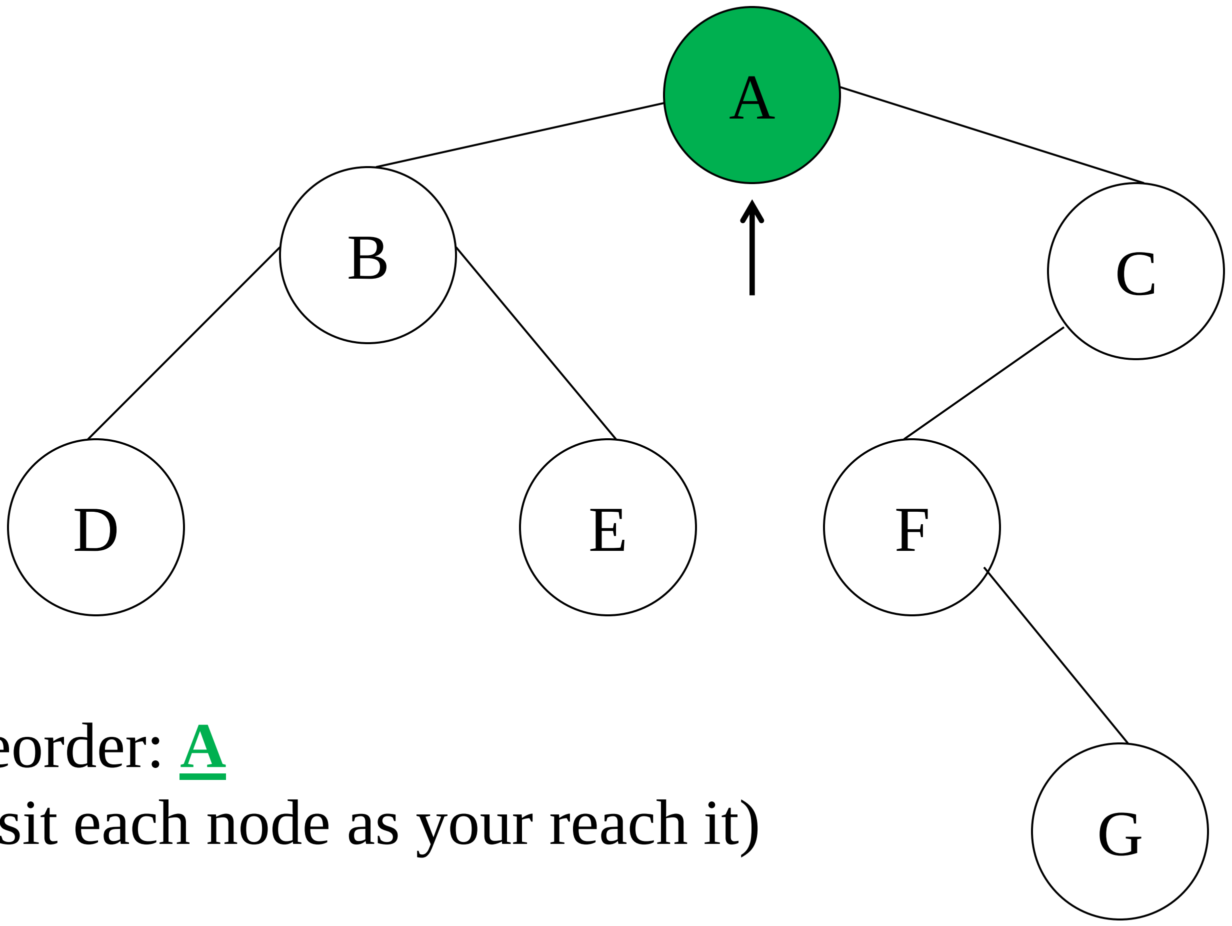
Tree Traversals: An Example



Preorder: ABDECFG
Postorder: DEBGFCA
In Order: DBEAFGC

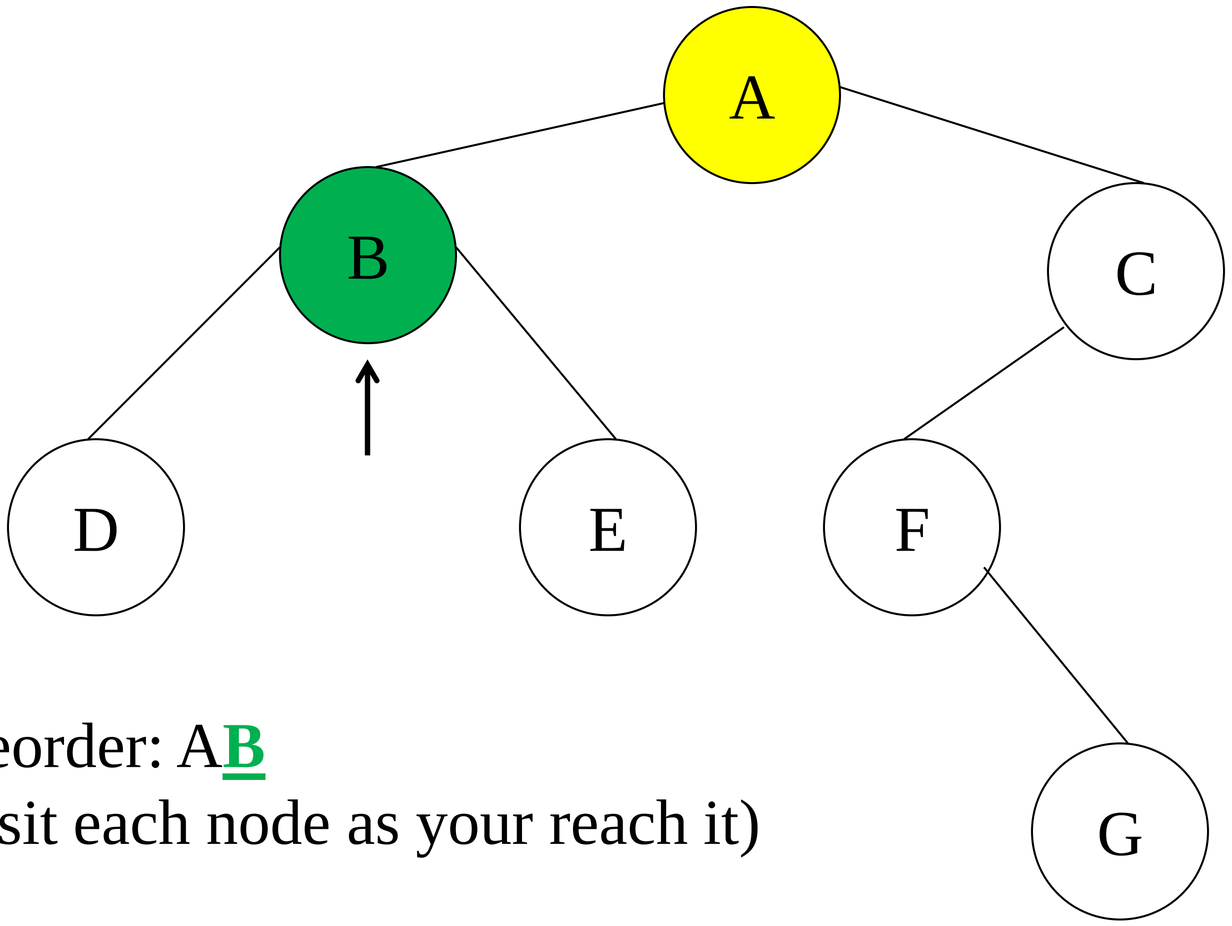
Tree Traversals: An Example

Preorder



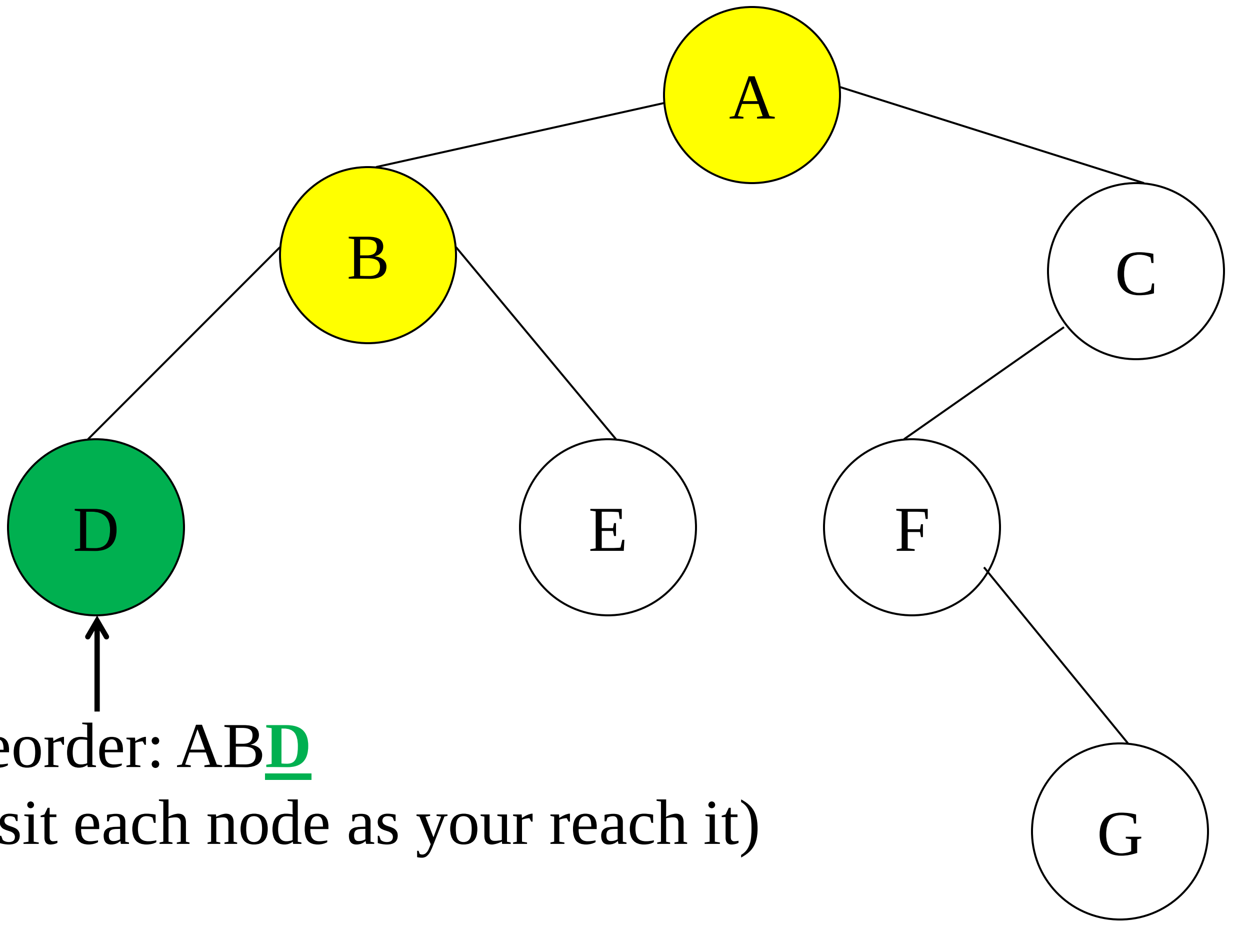
Preorder: **A**
(visit each node as your reach it)

Tree Traversals: An Example



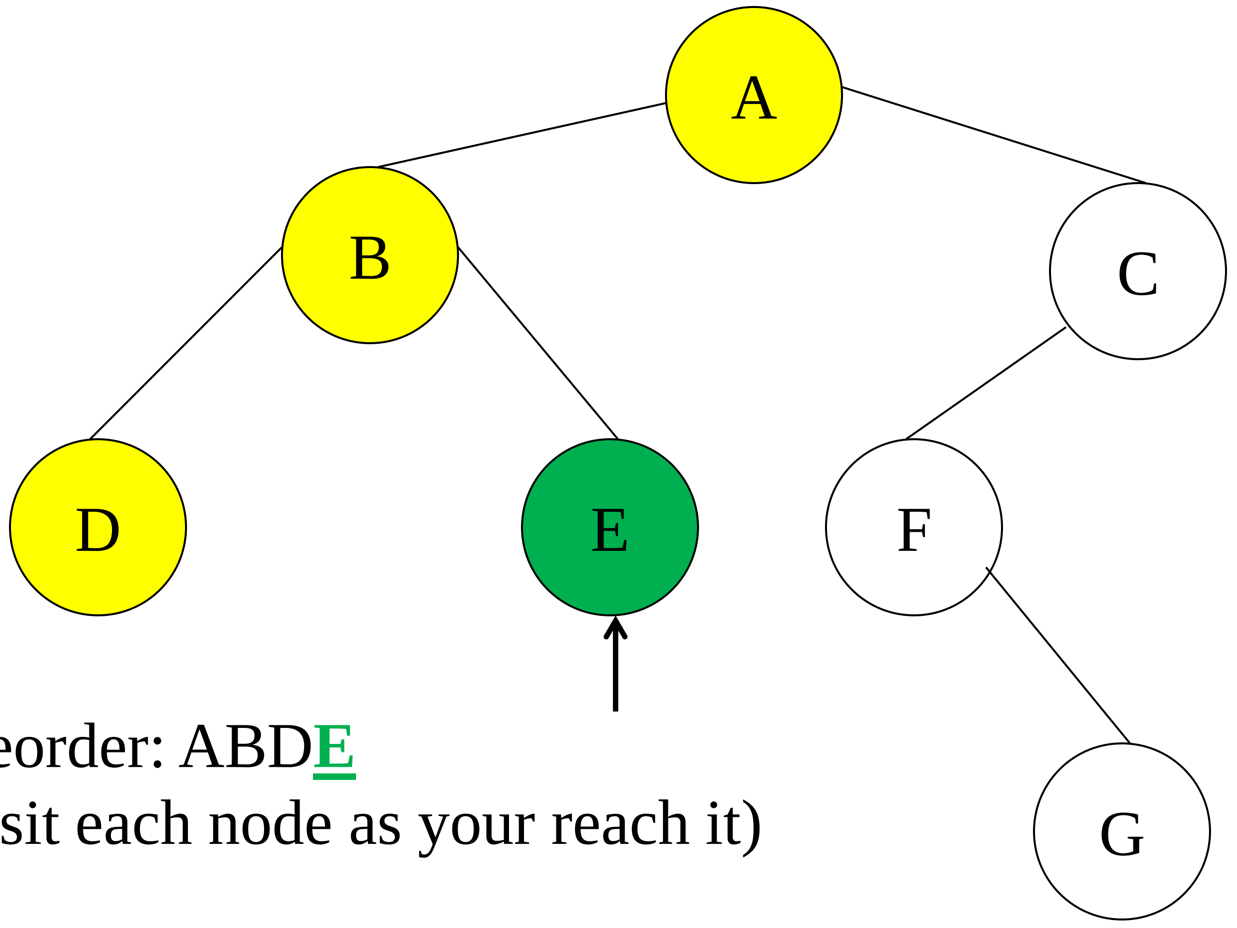
Preorder: A**B**
(visit each node as your reach it)

Tree Traversals: An Example



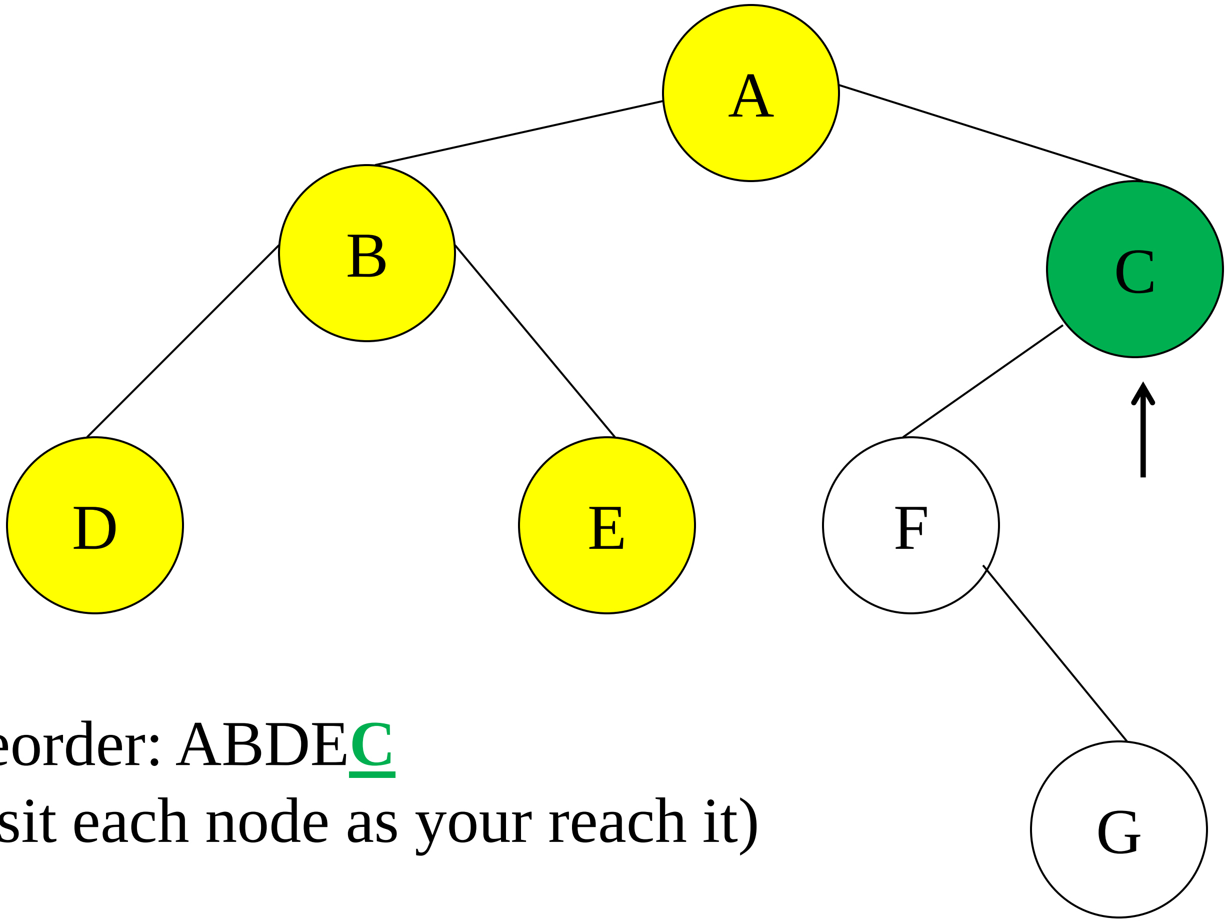
Preorder: AB**D**
(visit each node as your reach it)

Tree Traversals: An Example



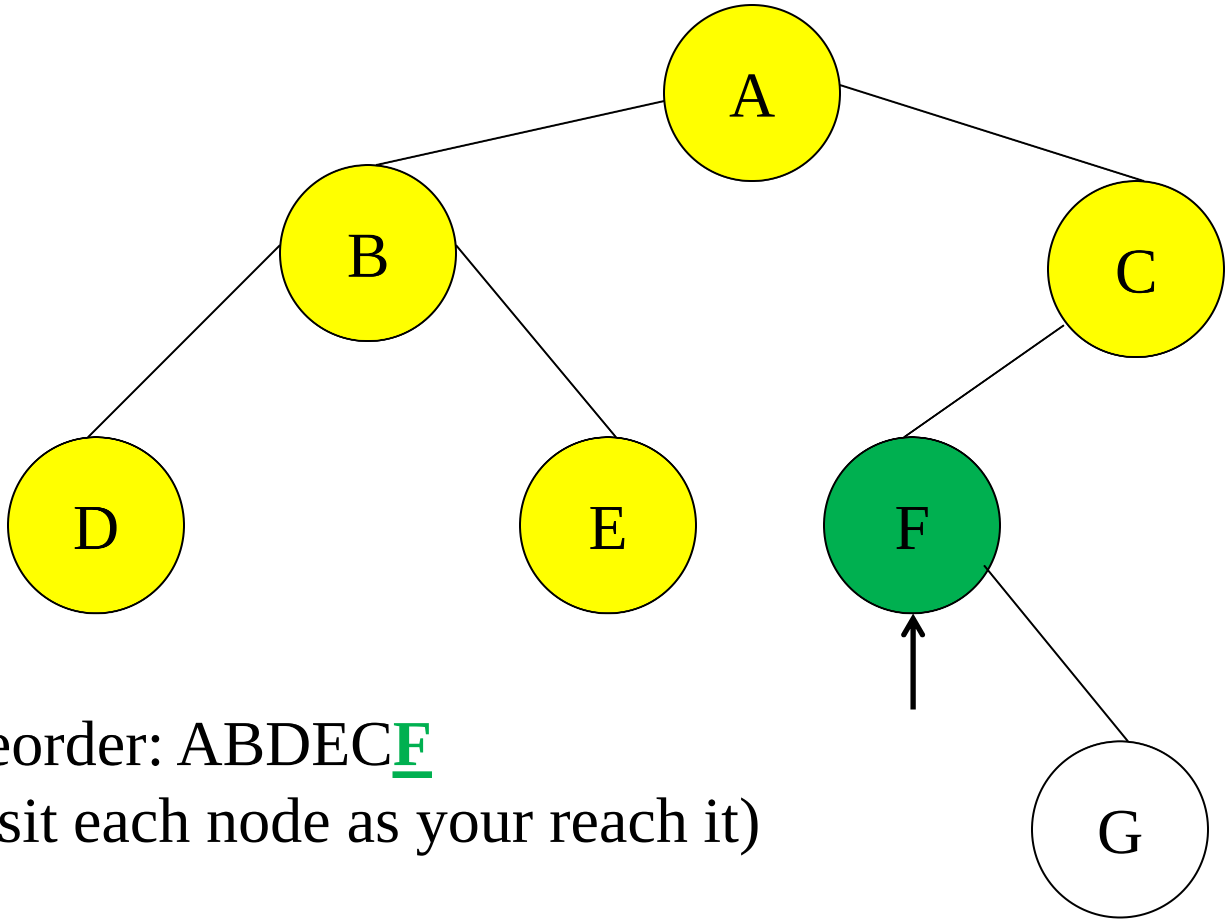
Preorder: ABD**E**
(visit each node as your reach it)

Tree Traversals: An Example



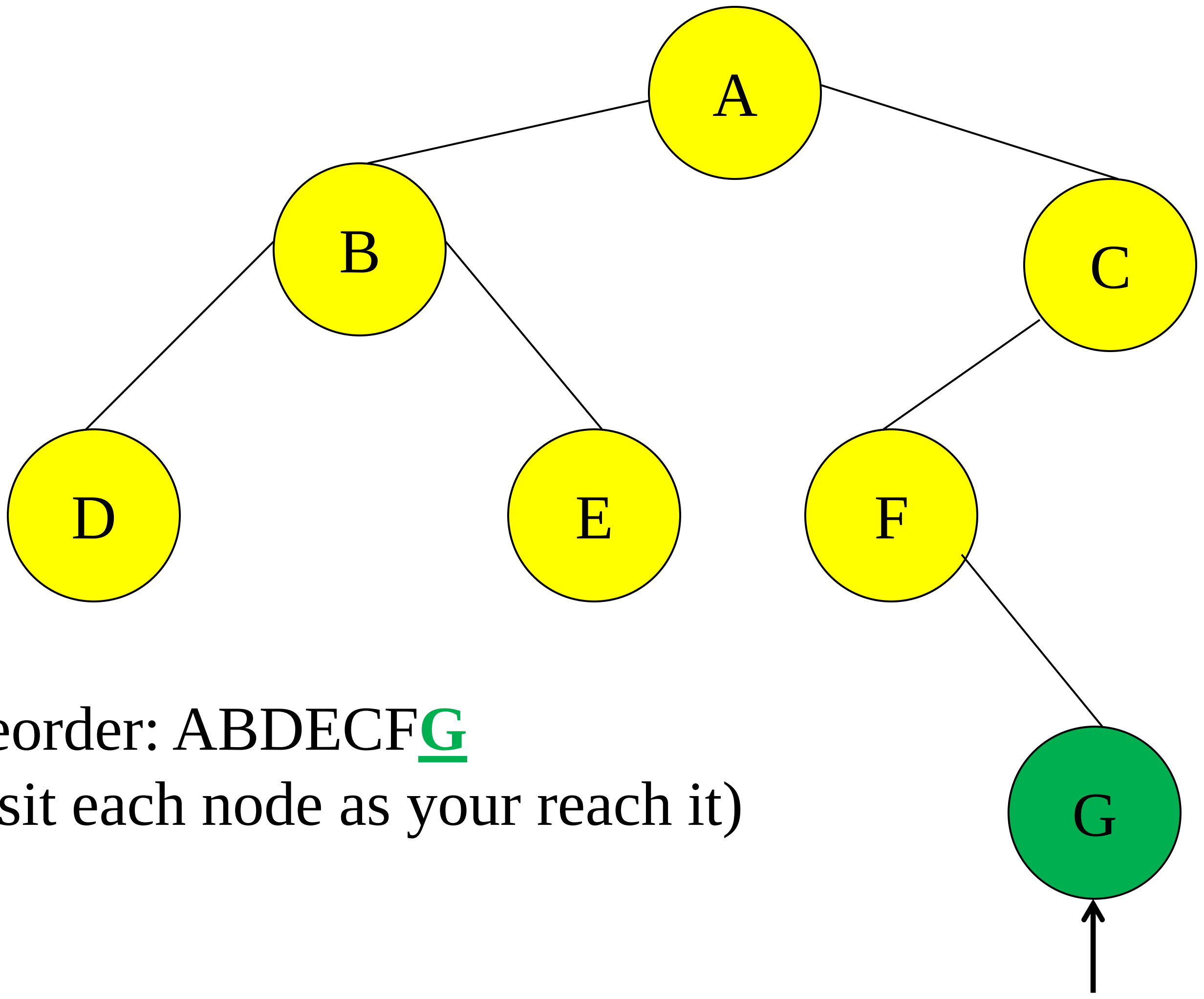
Preorder: ABDE**C**
(visit each node as your reach it)

Tree Traversals: An Example



Preorder: ABDEC**F**
(visit each node as your reach it)

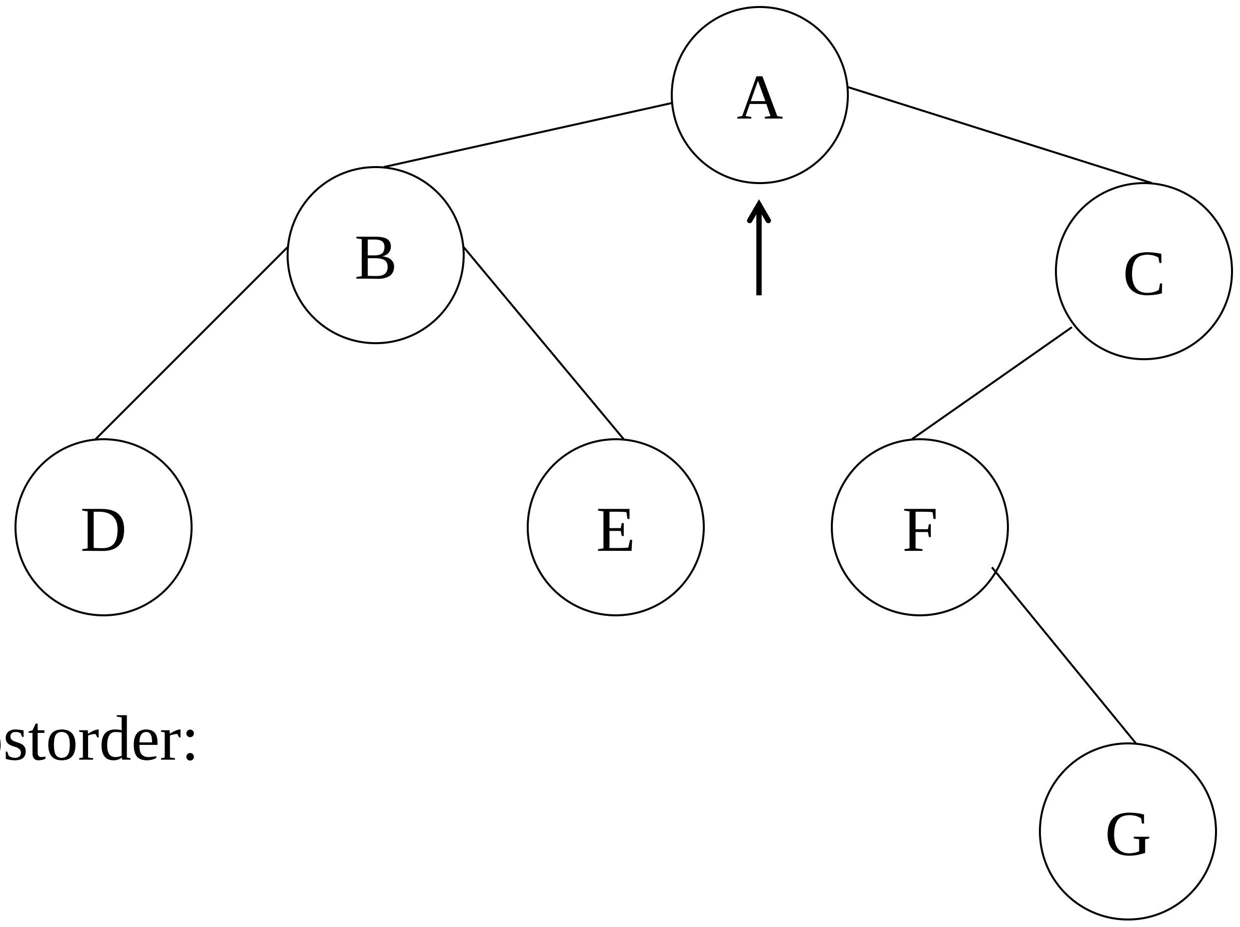
Tree Traversals: An Example



Preorder: ABDECF**G**
(visit each node as your reach it)

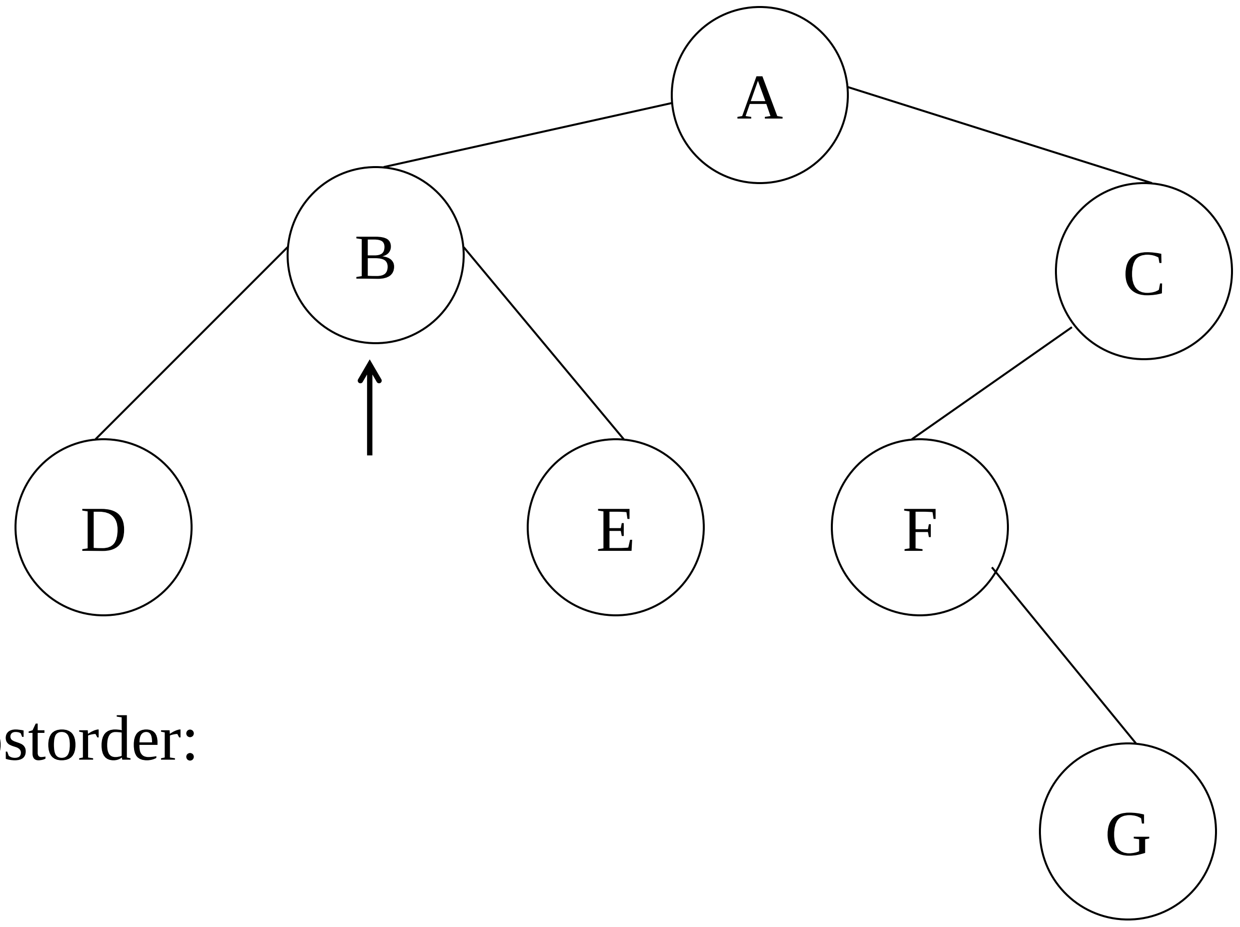
Tree Traversals: An Example

Postorder:



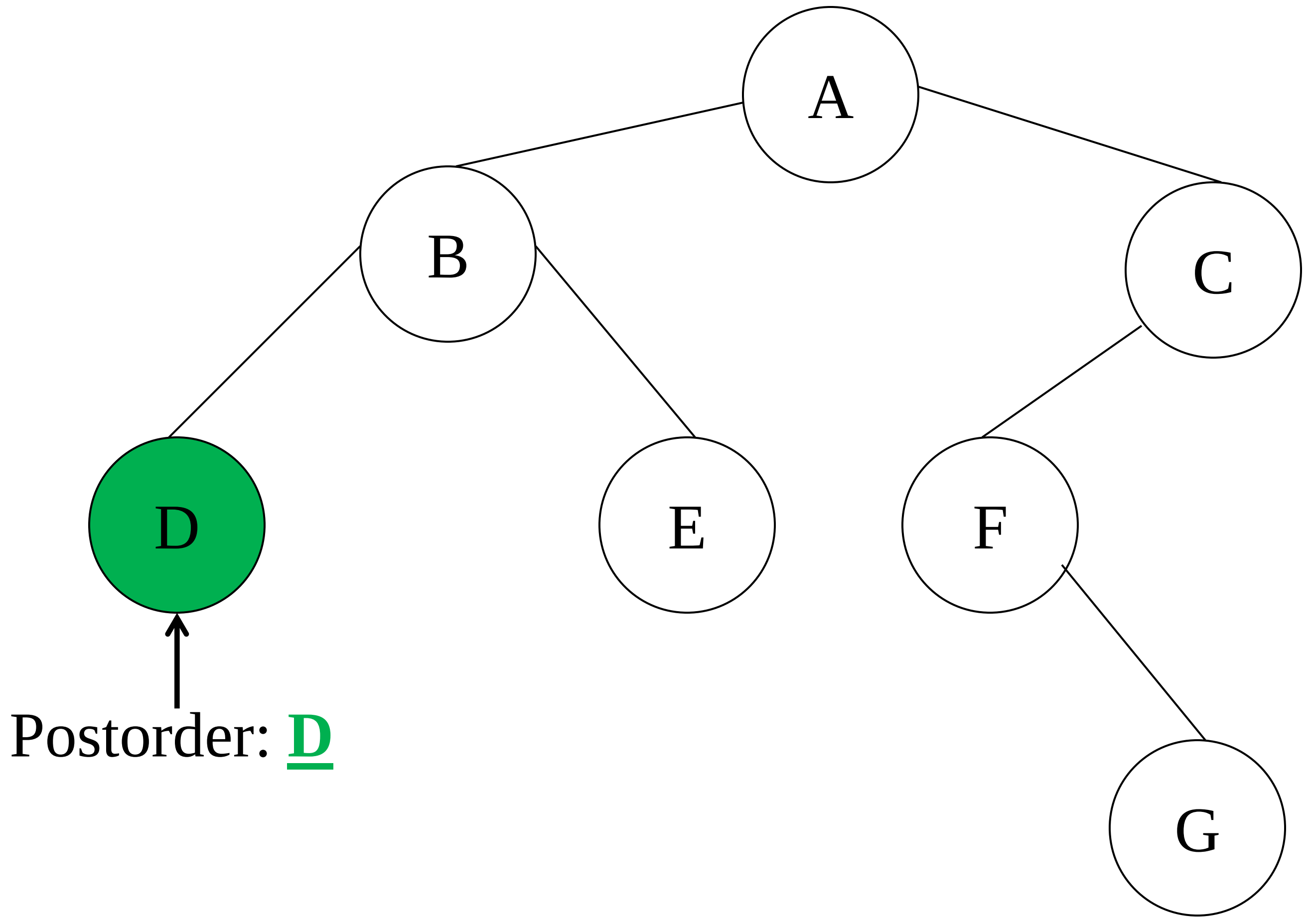
Postorder:

Tree Traversals: An Example



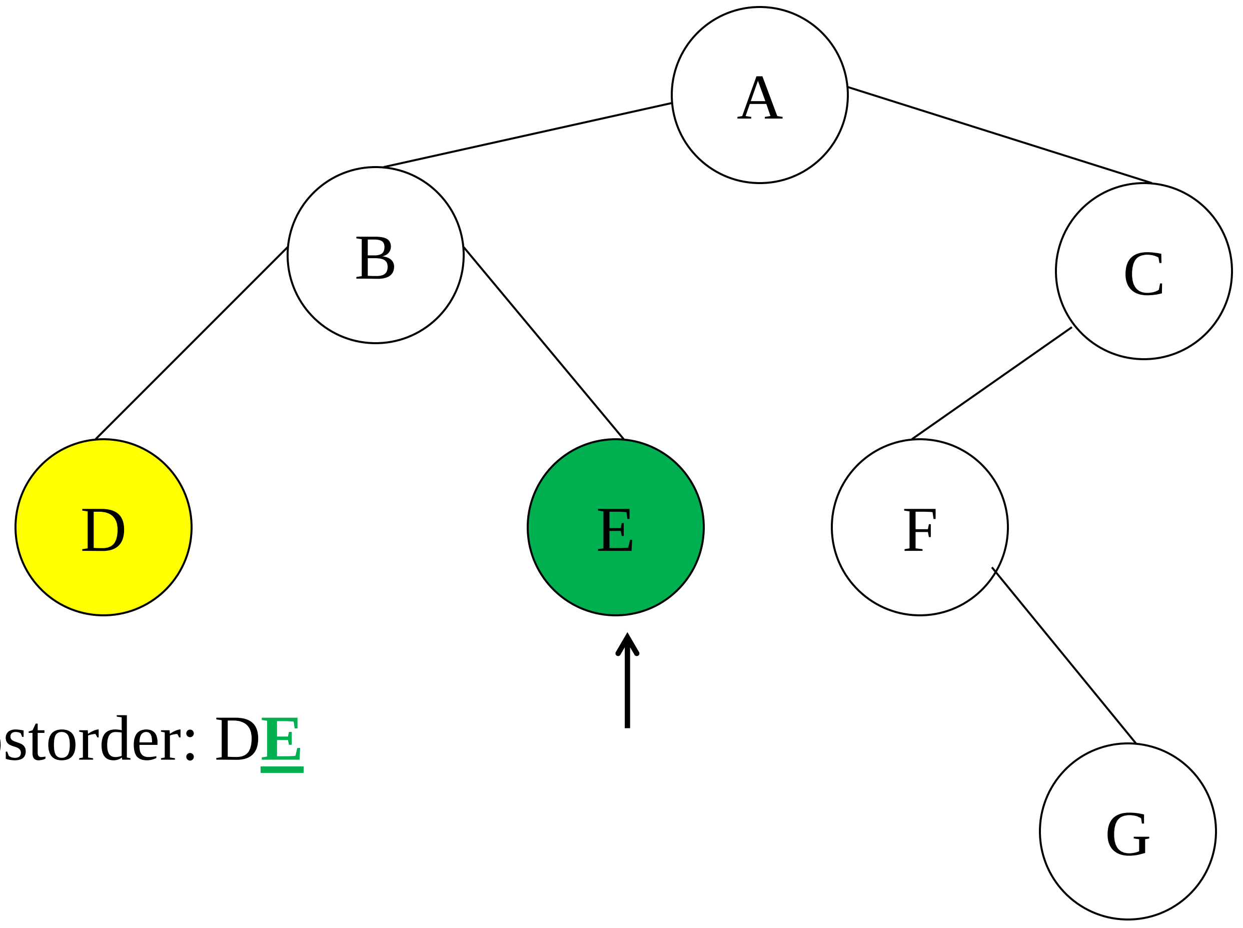
Postorder:

Tree Traversals: An Example



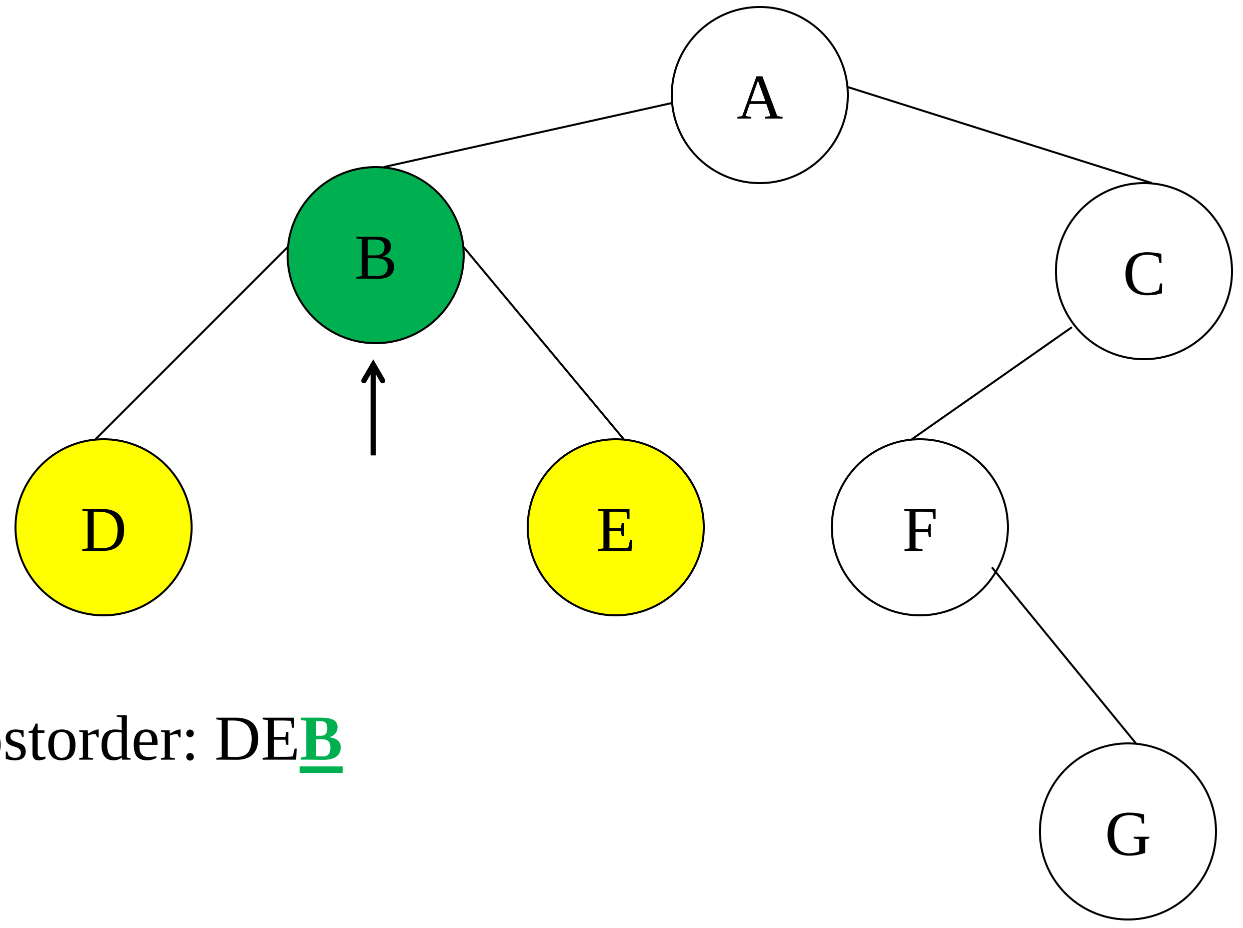
Postorder: **D**

Tree Traversals: An Example



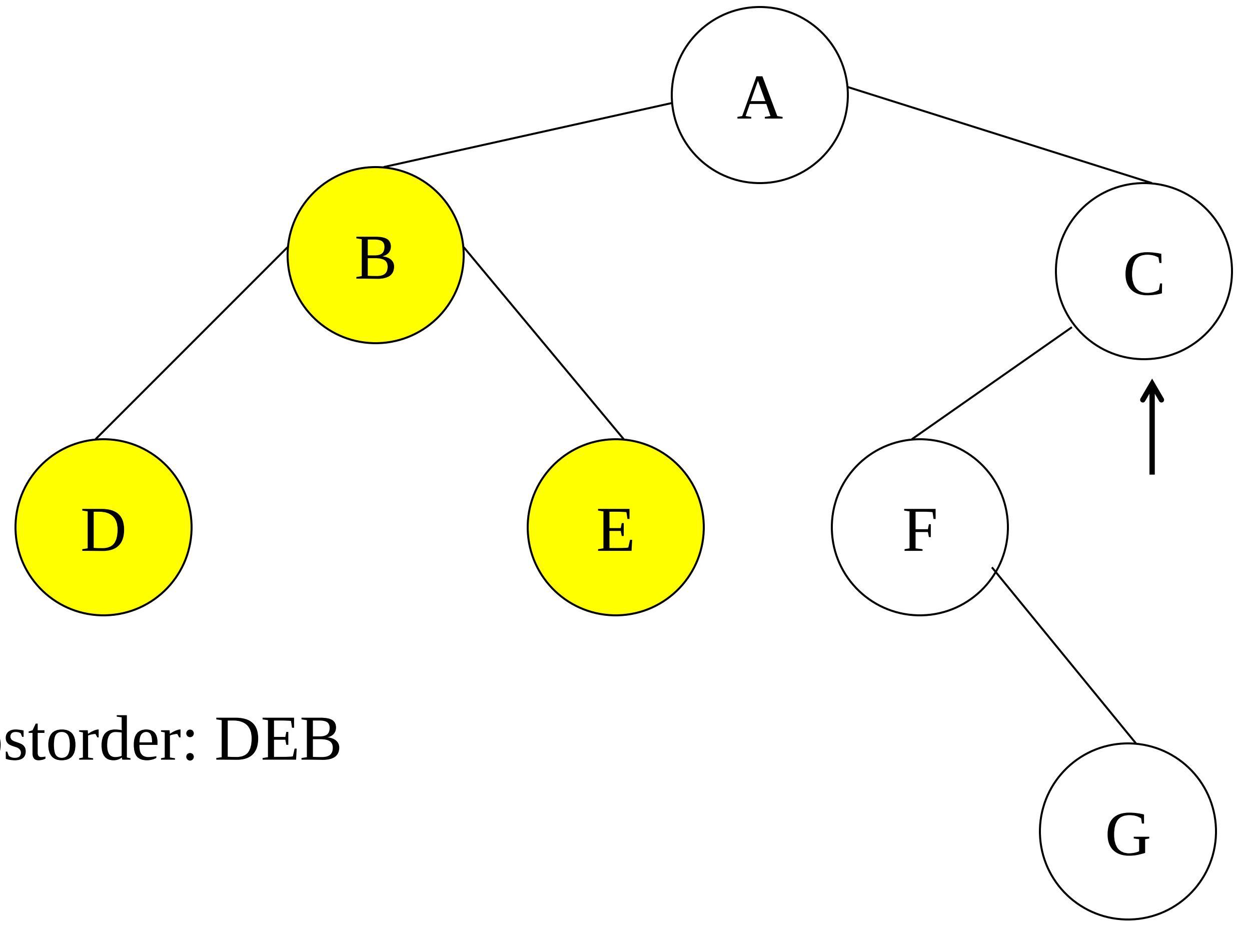
Postorder: D**E**

Tree Traversals: An Example



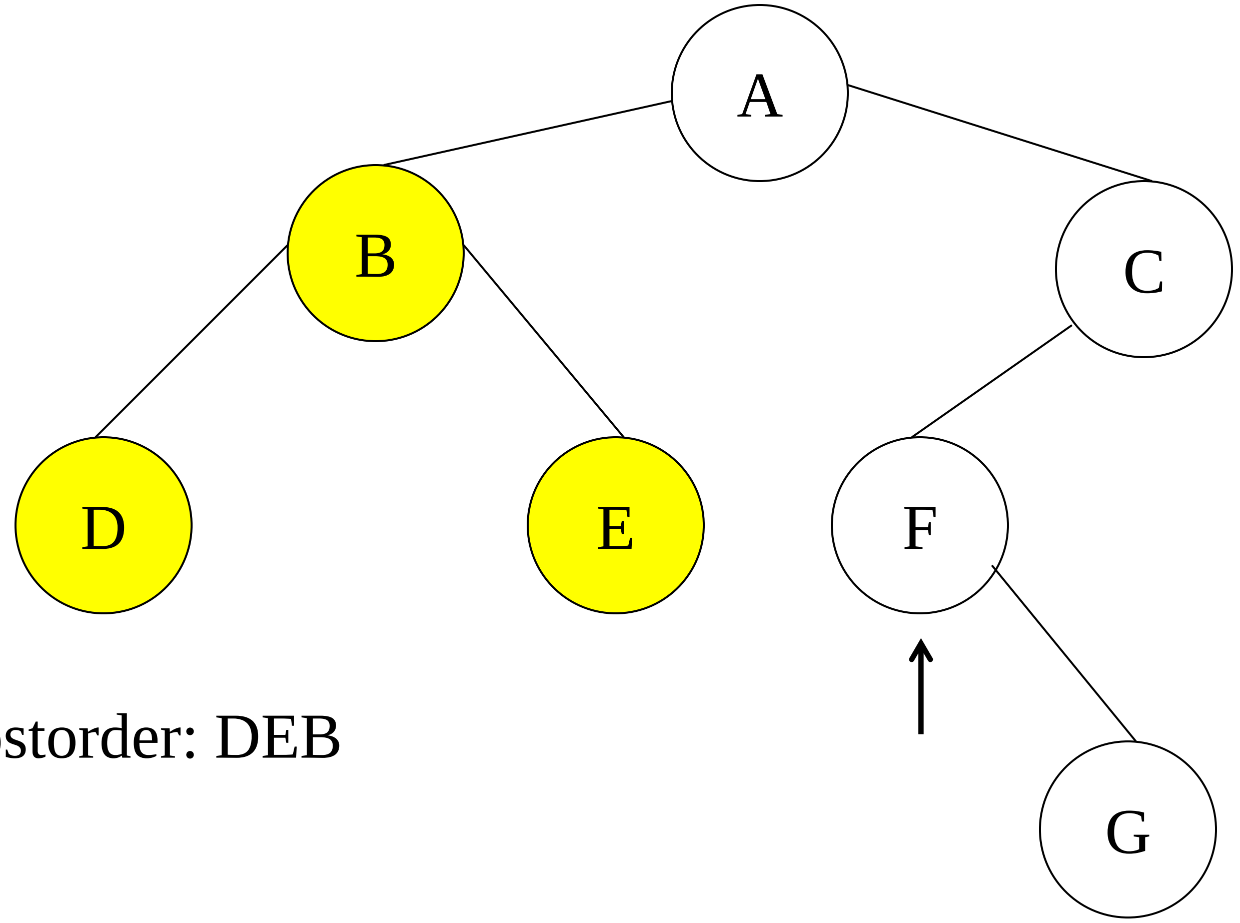
Postorder: DE**B**

Tree Traversals: An Example



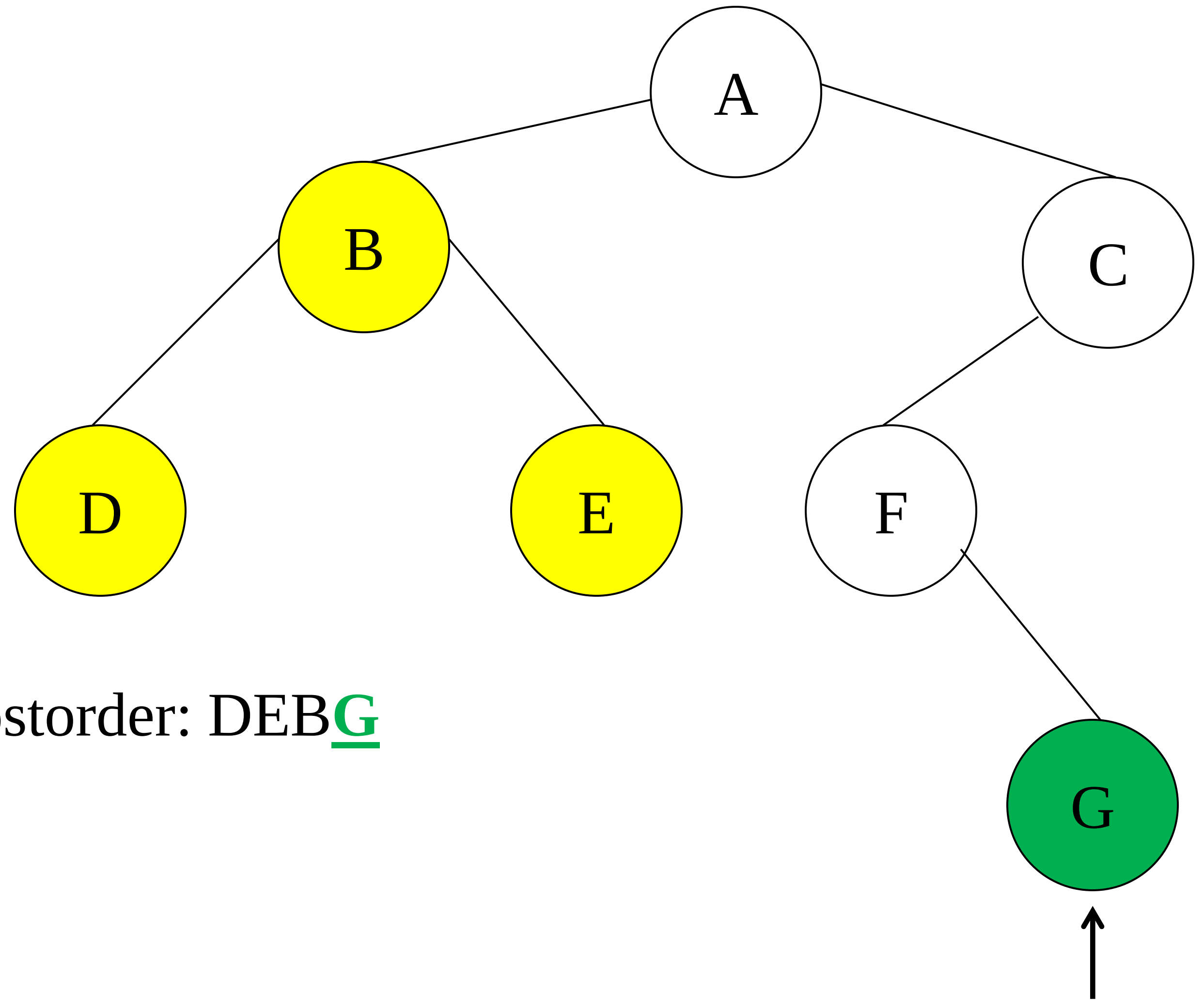
Postorder: DEB

Tree Traversals: An Example



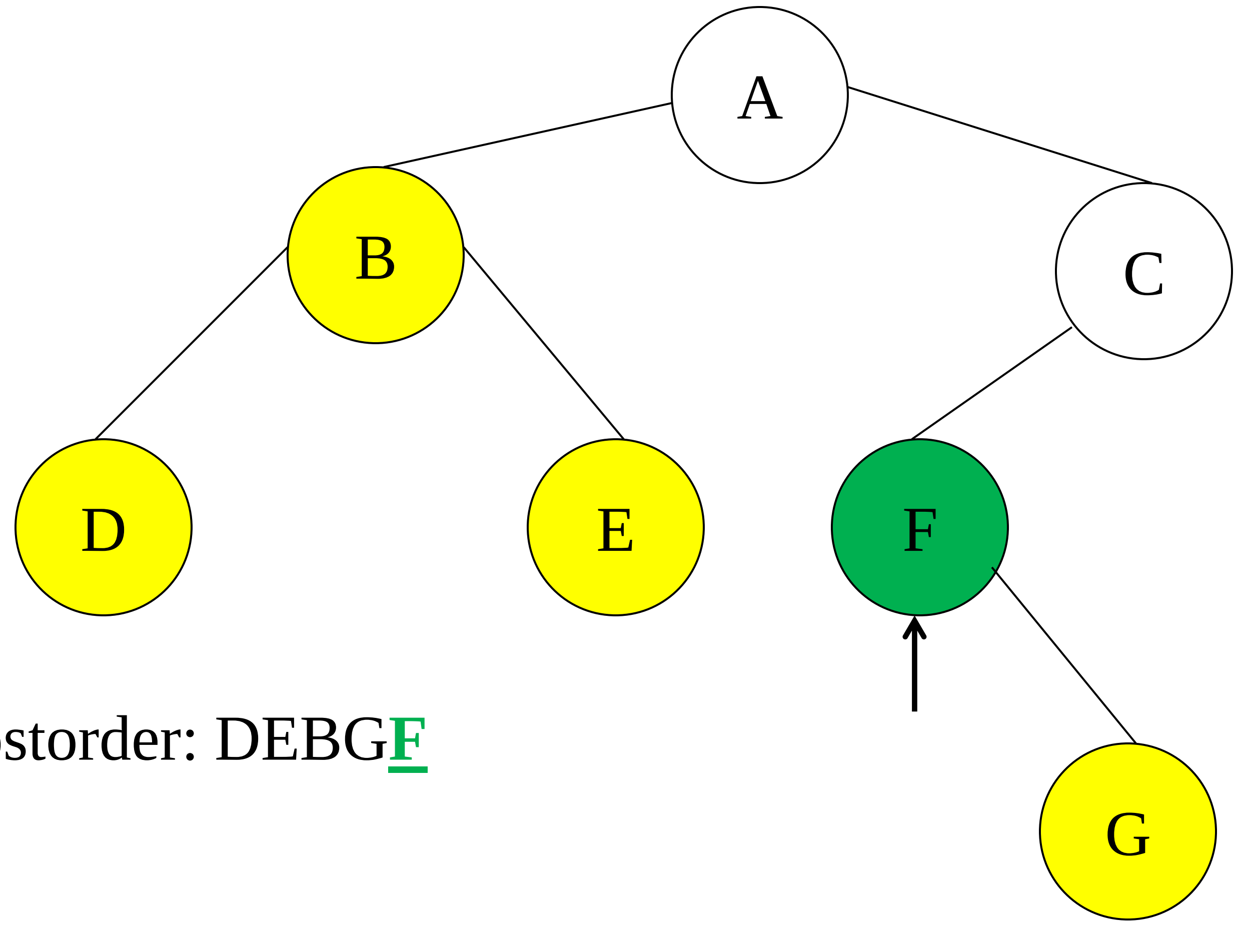
Postorder: DEB

Tree Traversals: An Example



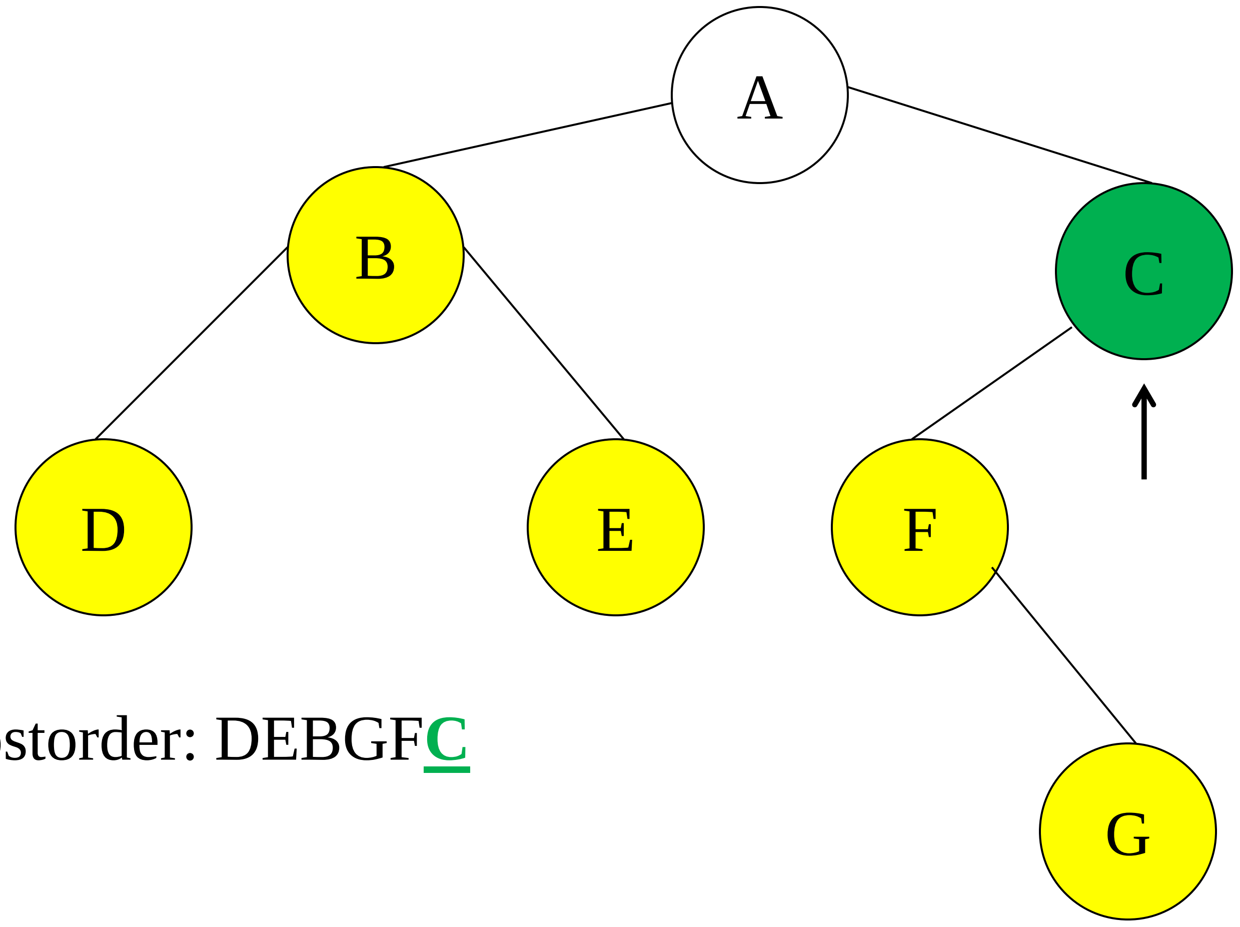
Postorder: DEB**G**

Tree Traversals: An Example



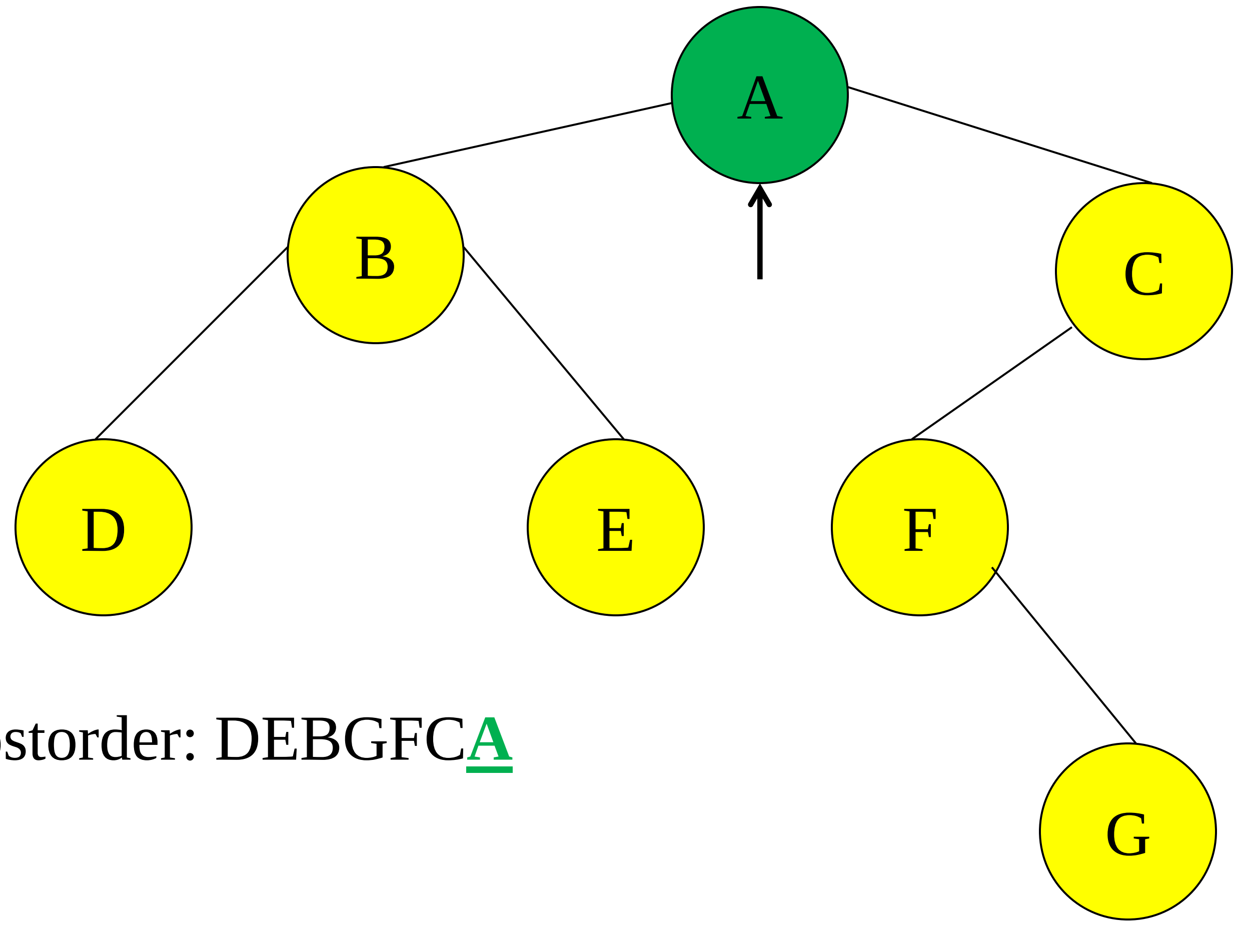
Postorder: DEBG**F**

Tree Traversals: An Example



Postorder: DEBGF**C**

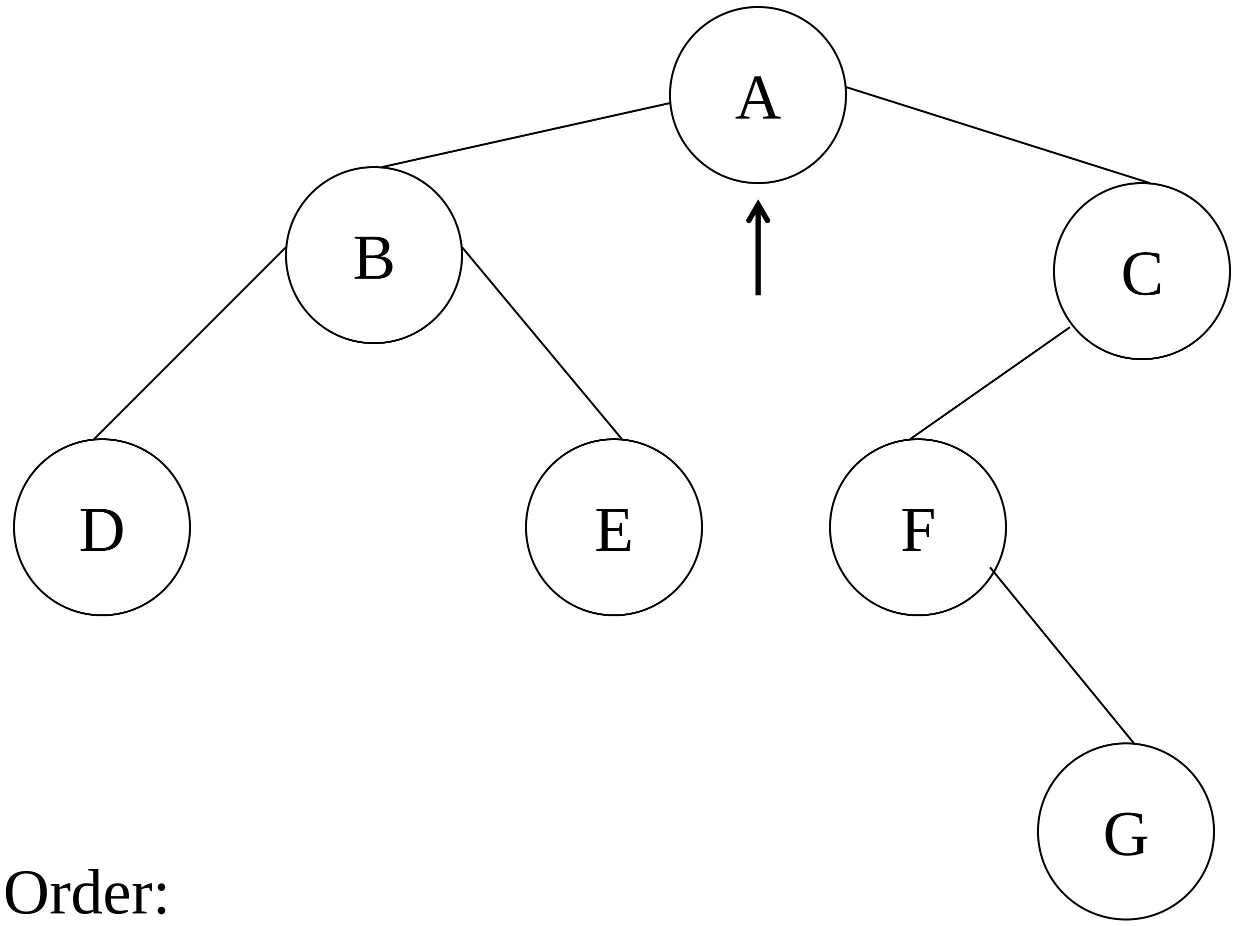
Tree Traversals: An Example



Postorder: DEBGFC**A**

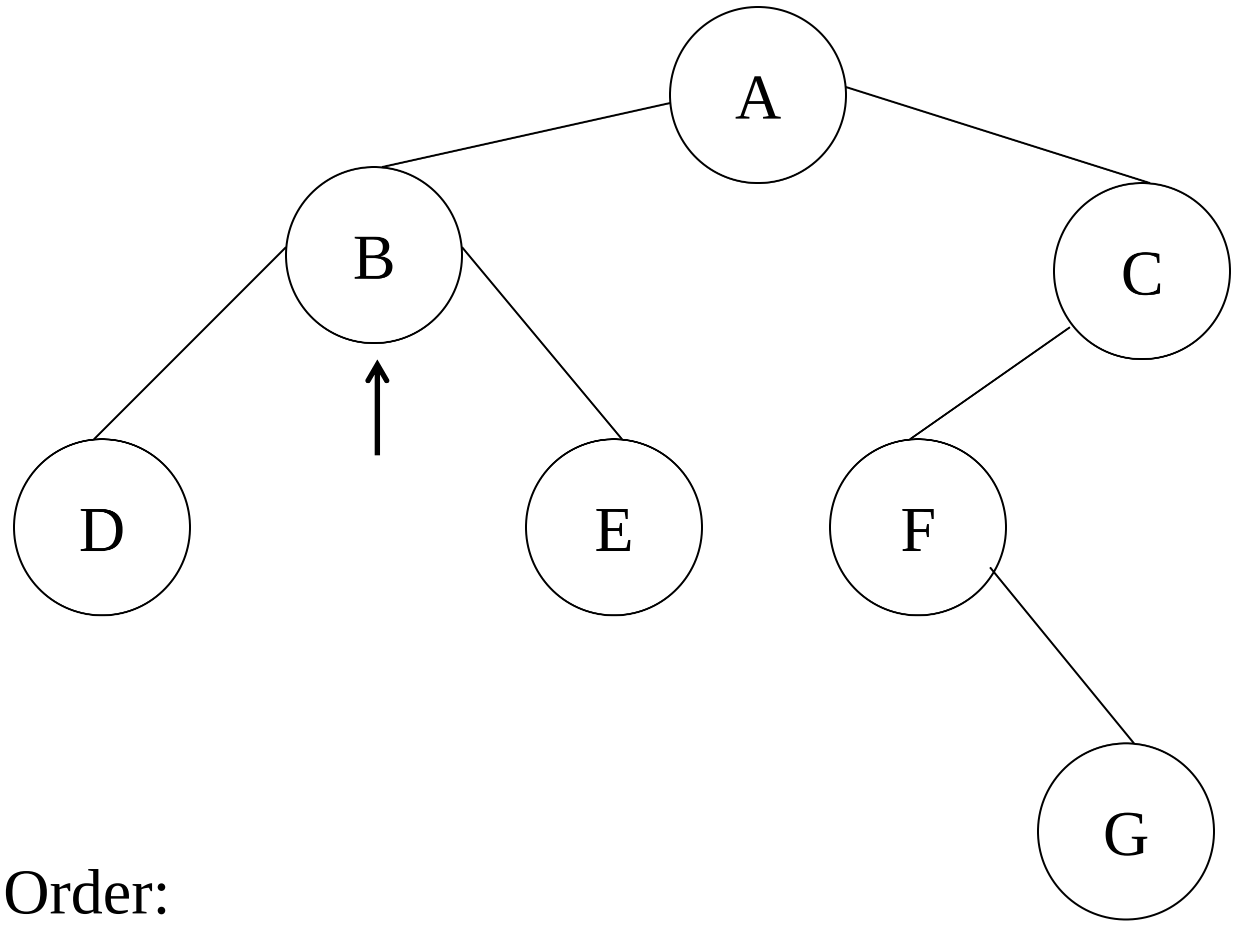
Tree Traversals: An Example

In Order:



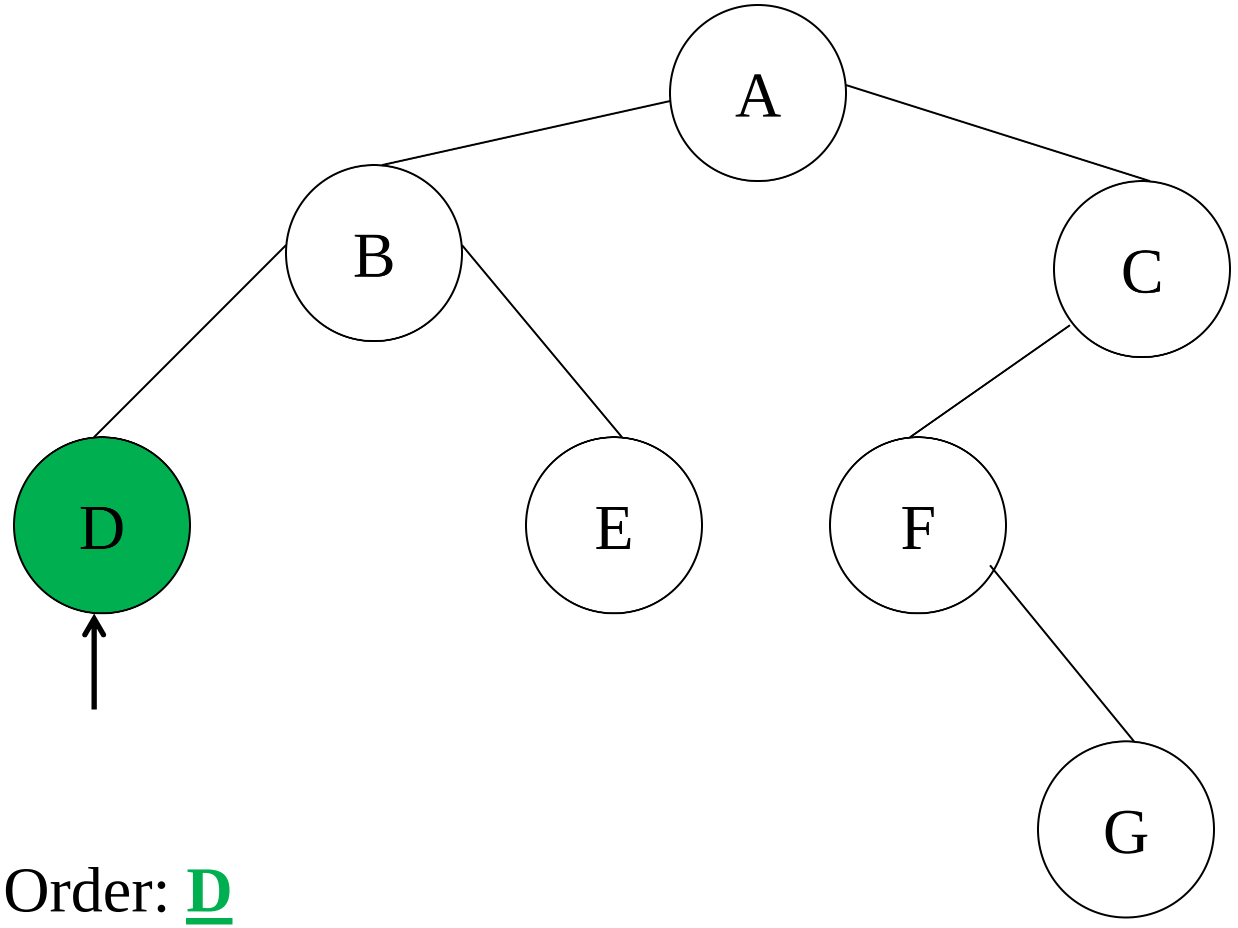
In Order:

Tree Traversals: An Example



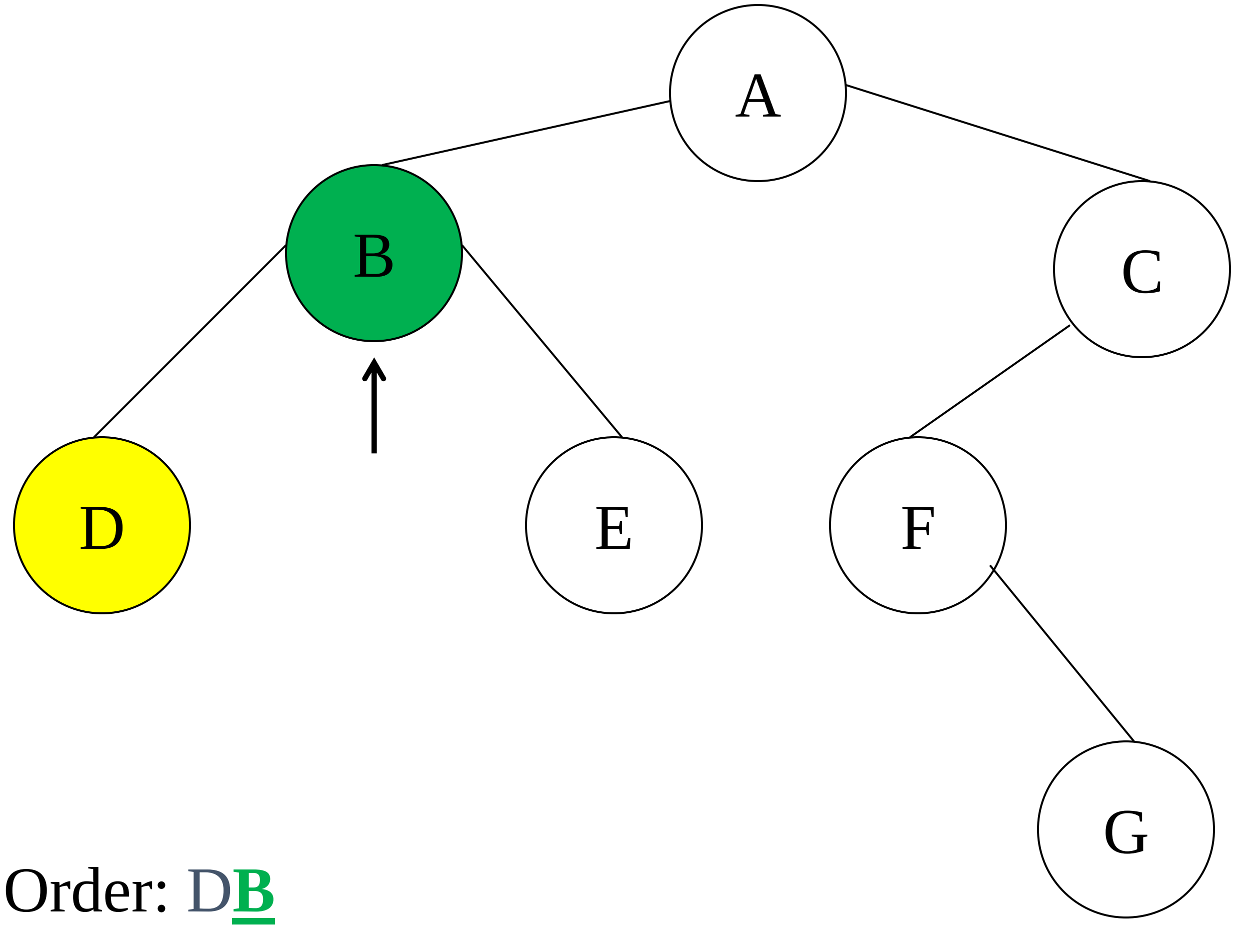
In Order:

Tree Traversals: An Example



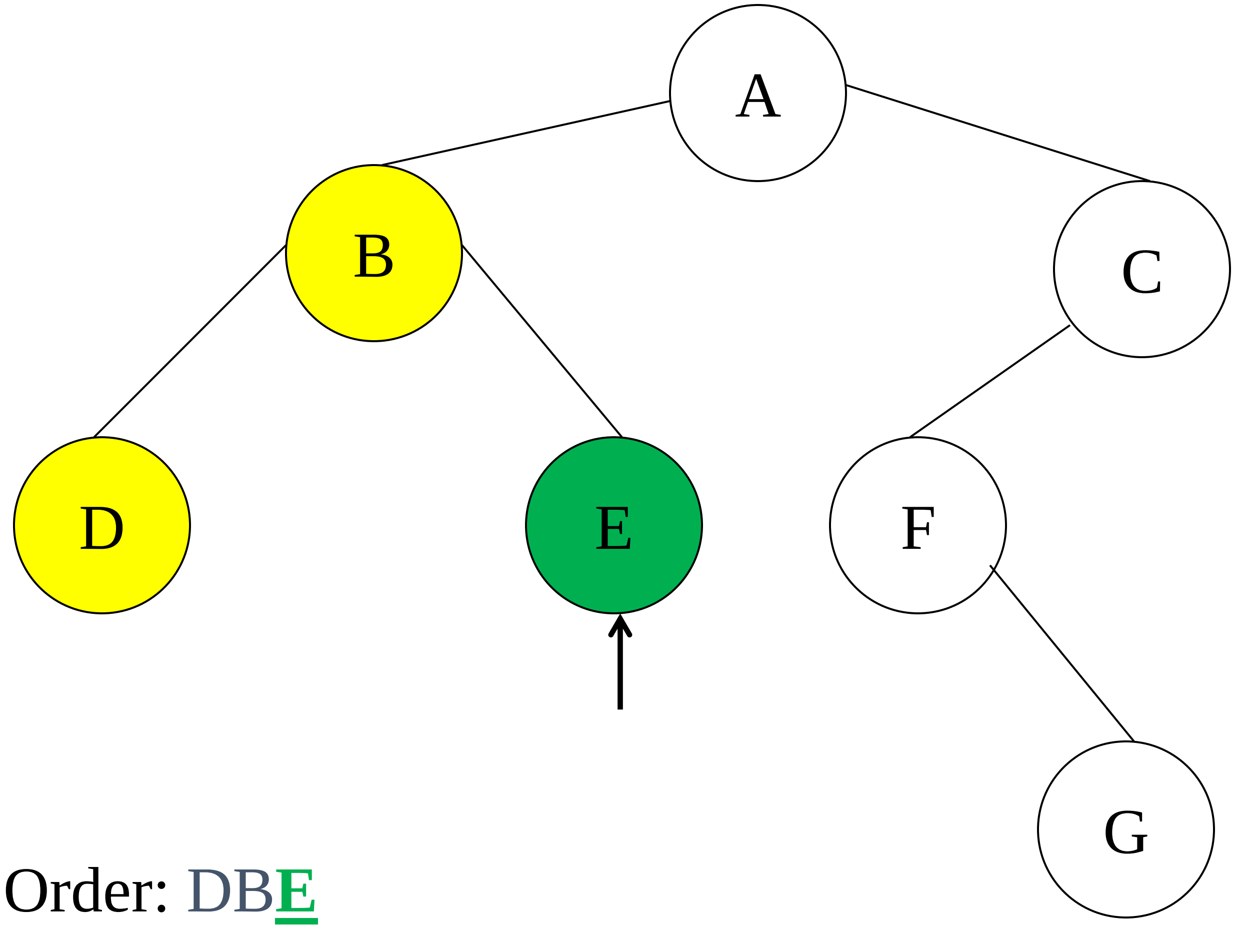
In Order: **D**

Tree Traversals: An Example



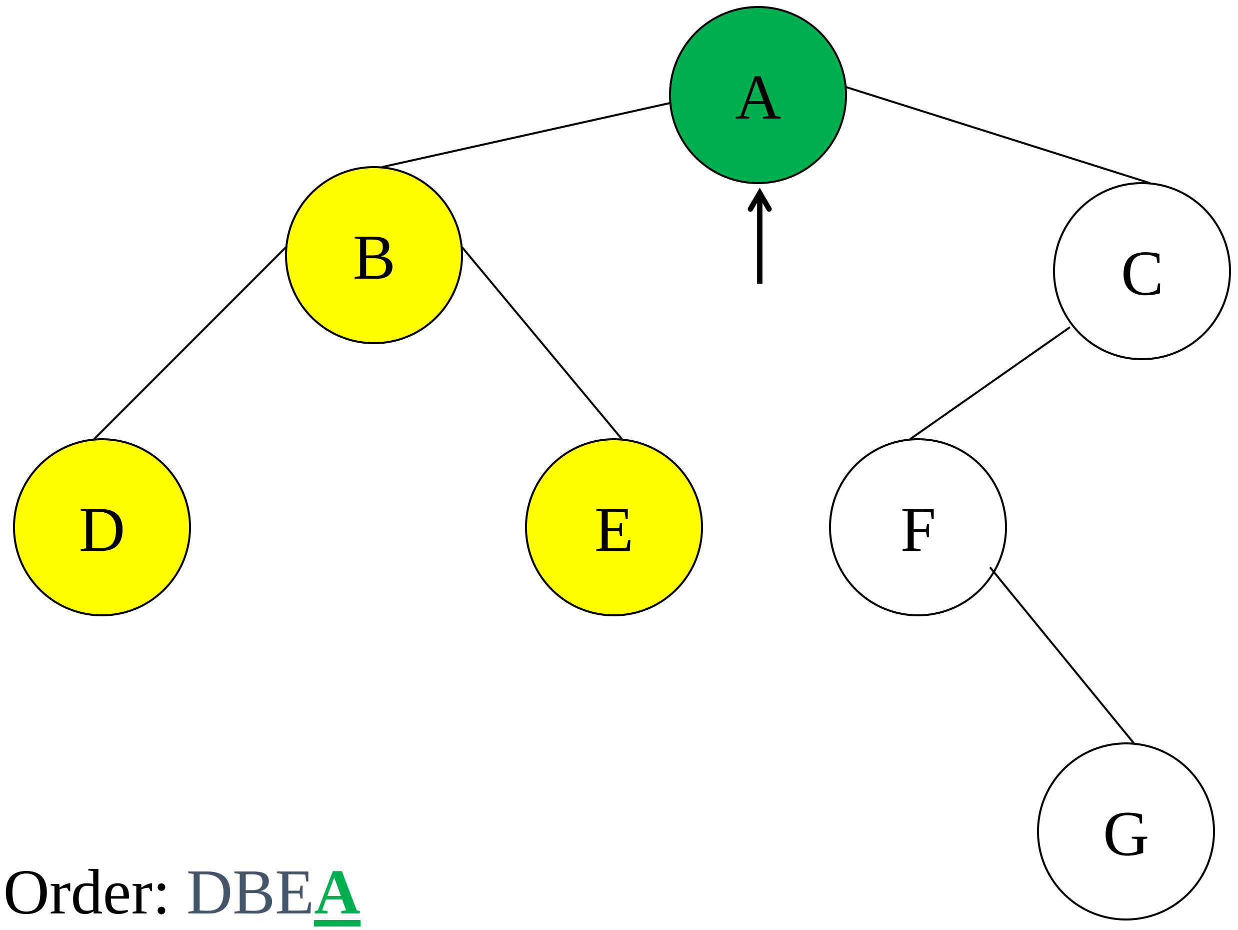
In Order: D**B**

Tree Traversals: An Example



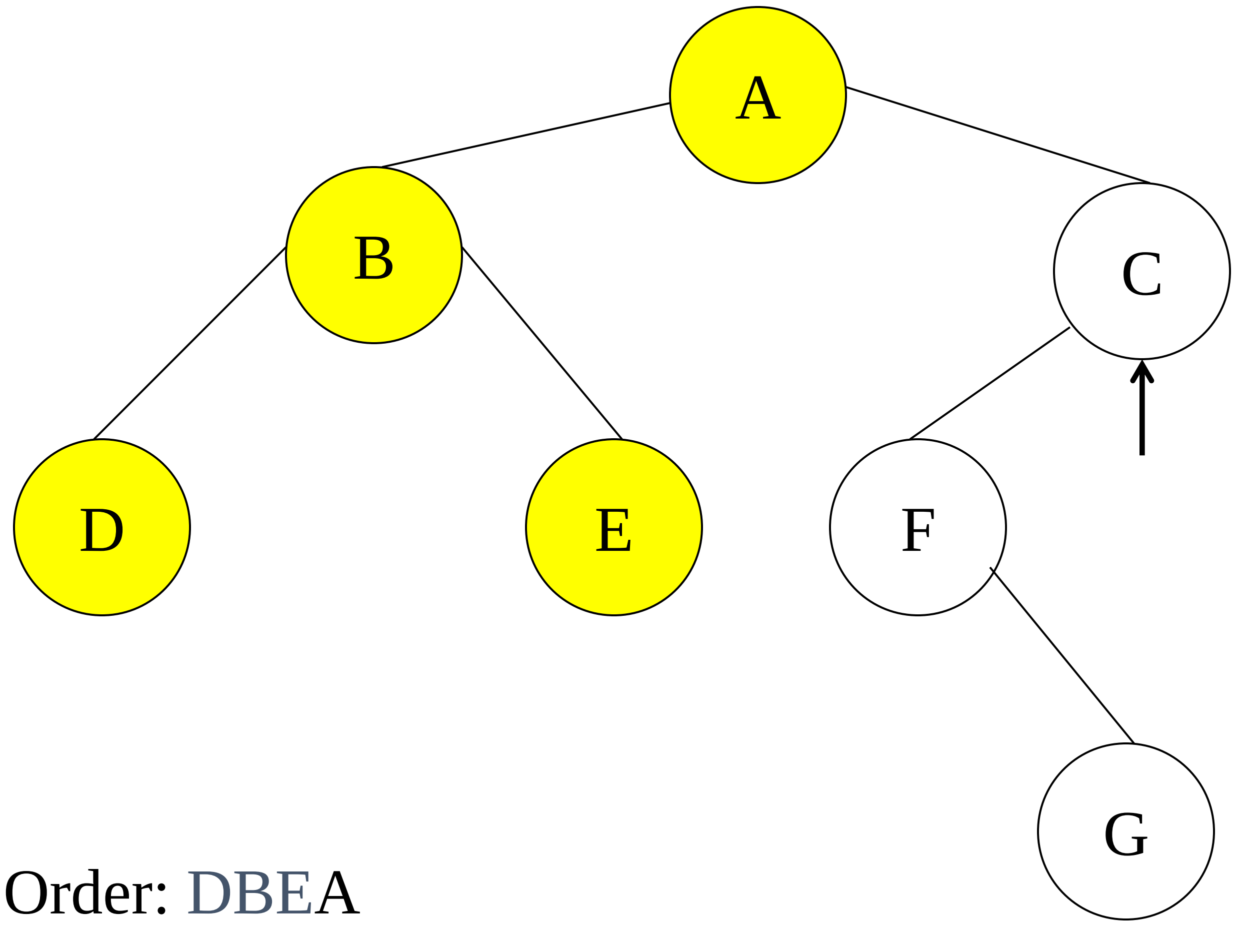
In Order: DB**E**

Tree Traversals: An Example



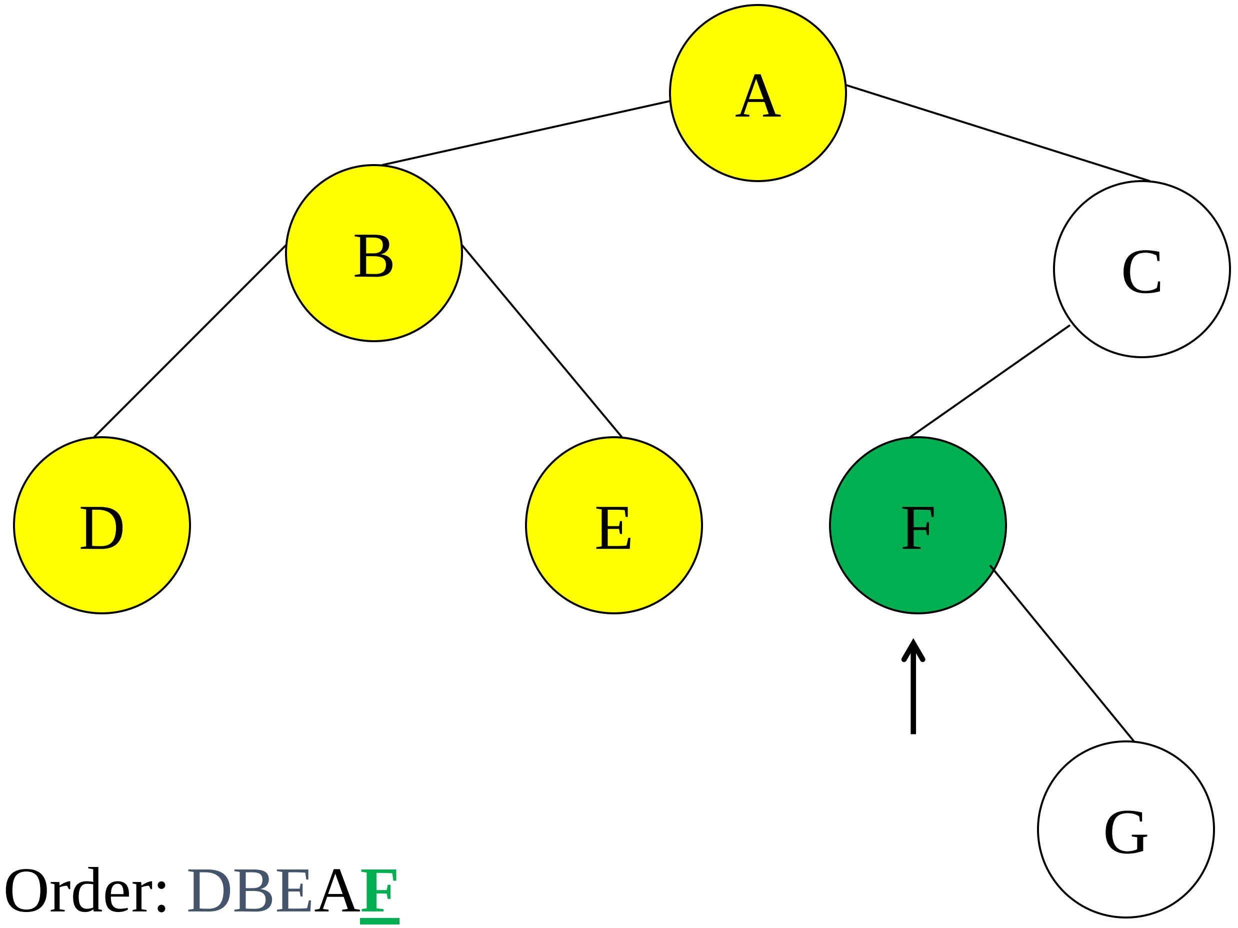
In Order: DBE**A**

Tree Traversals: An Example



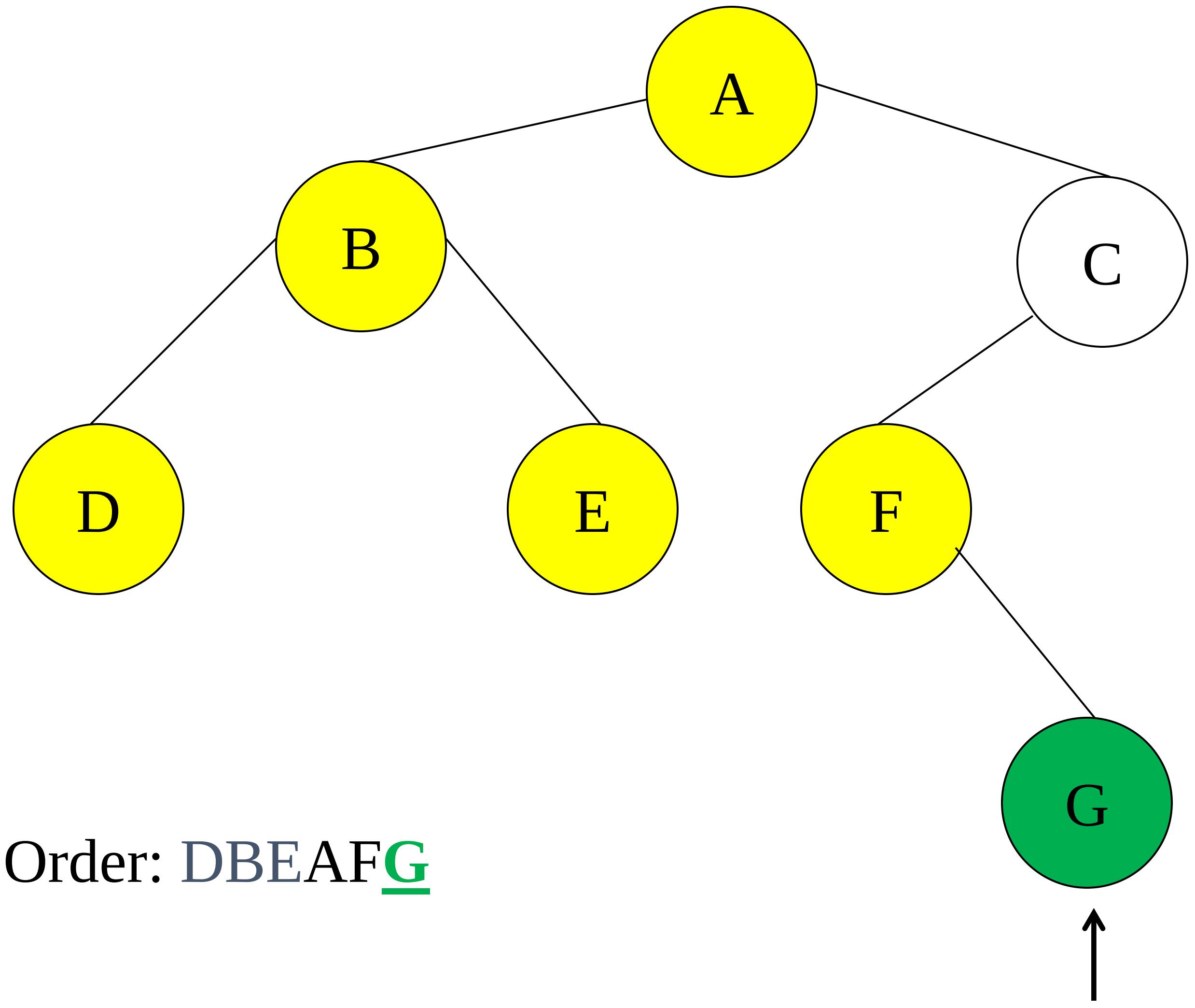
In Order: DBEA

Tree Traversals: An Example



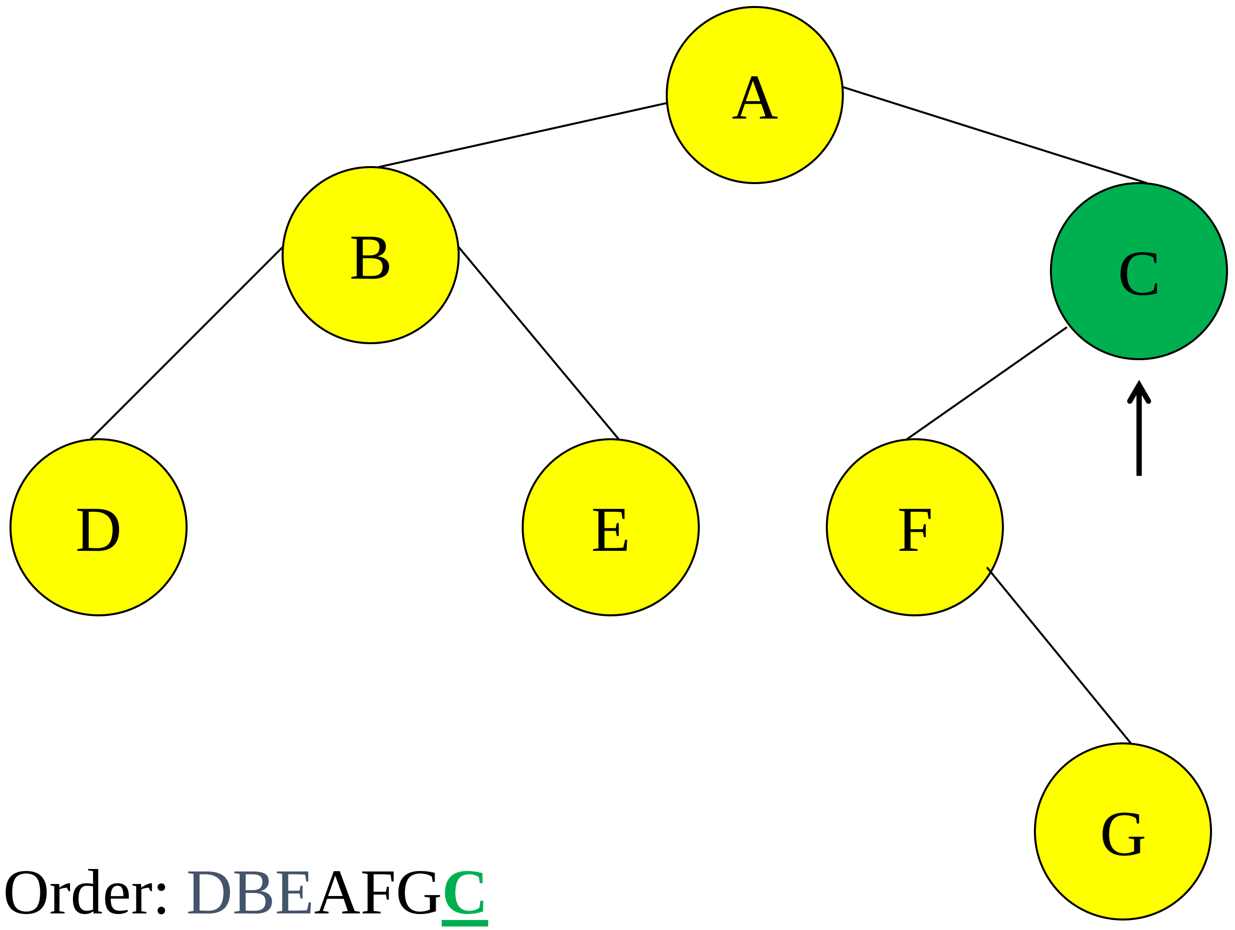
In Order: DBEA**F**

Tree Traversals: An Example



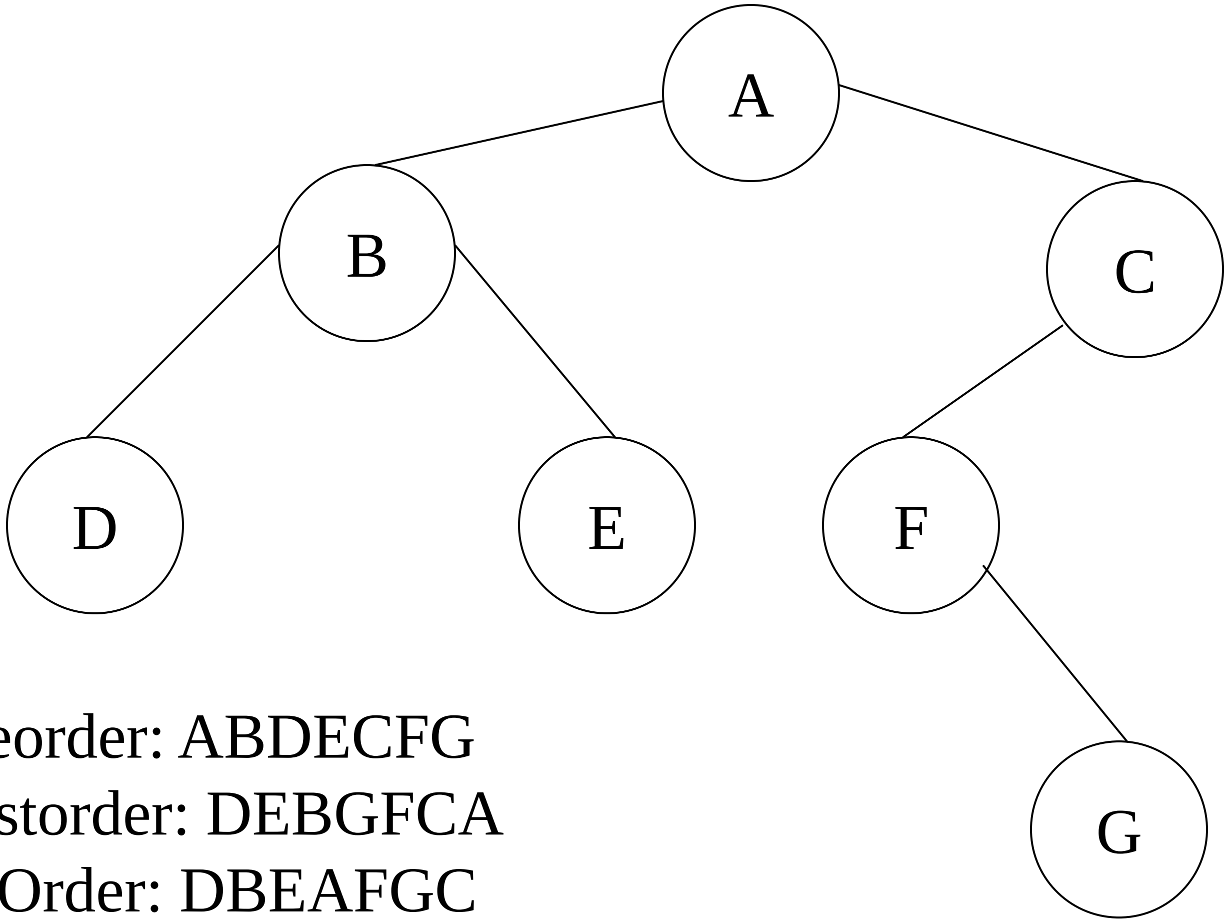
In Order: DBEAF**G**

Tree Traversals: An Example



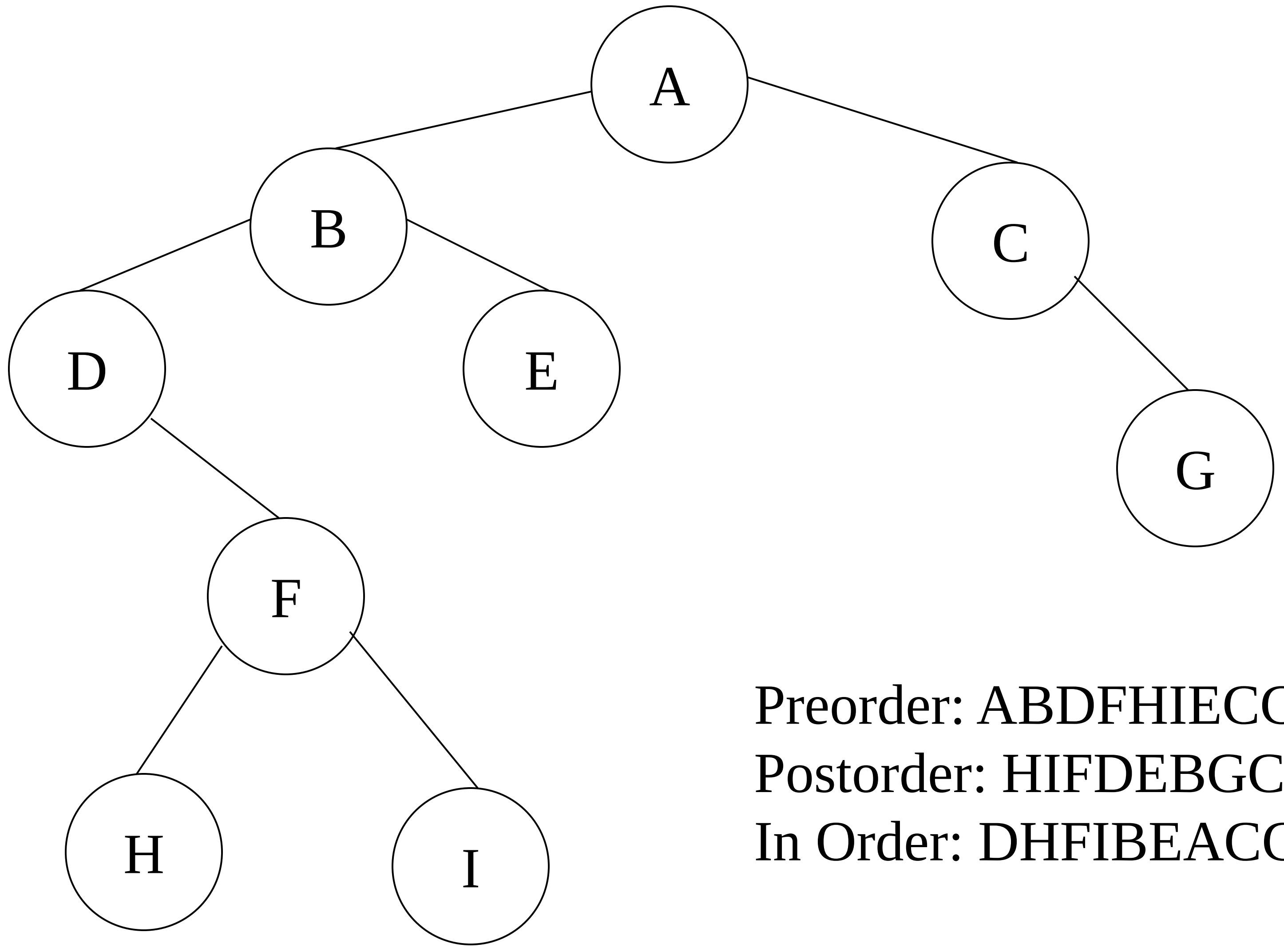
In Order: DBEAFGC

Tree Traversals: An Example



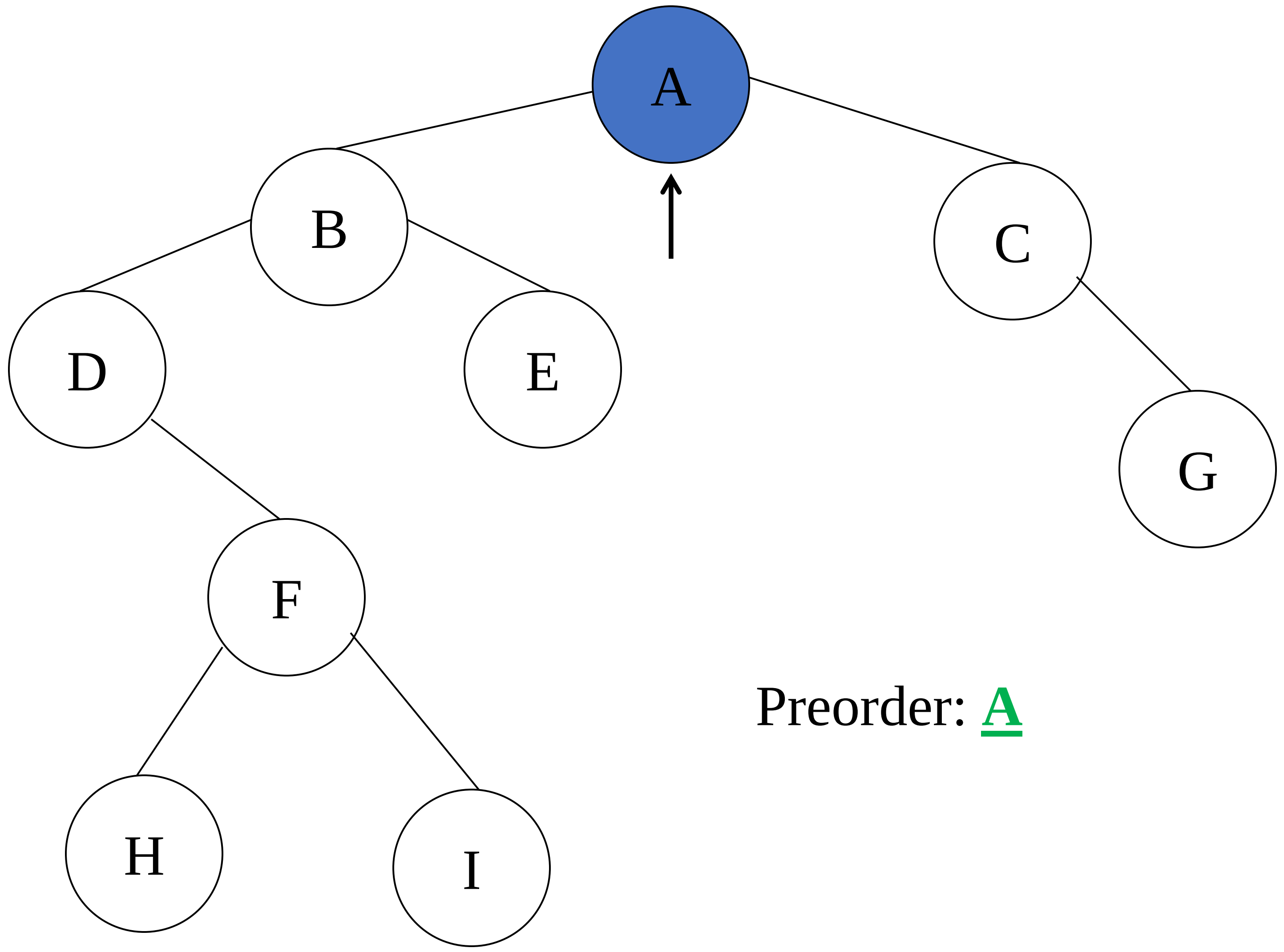
Preorder: ABDECFG
Postorder: DEBGFCA
In Order: DBEAFGC

Tree Traversals: Another Example



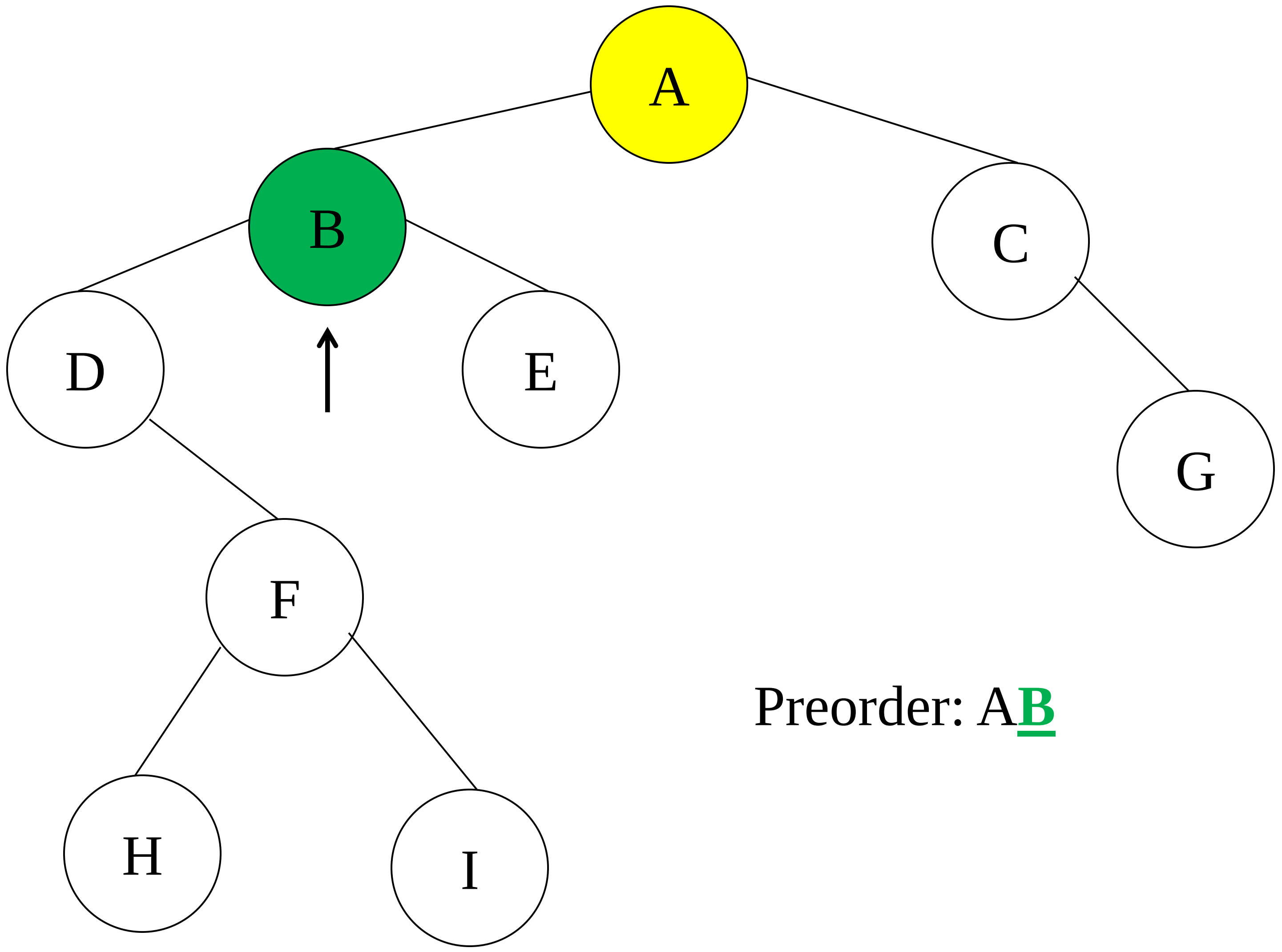
Preorder: ABDFHIECG
Postorder: HIFDEBGCA
In Order: DHFIBEACG

Tree Traversals: Another Example



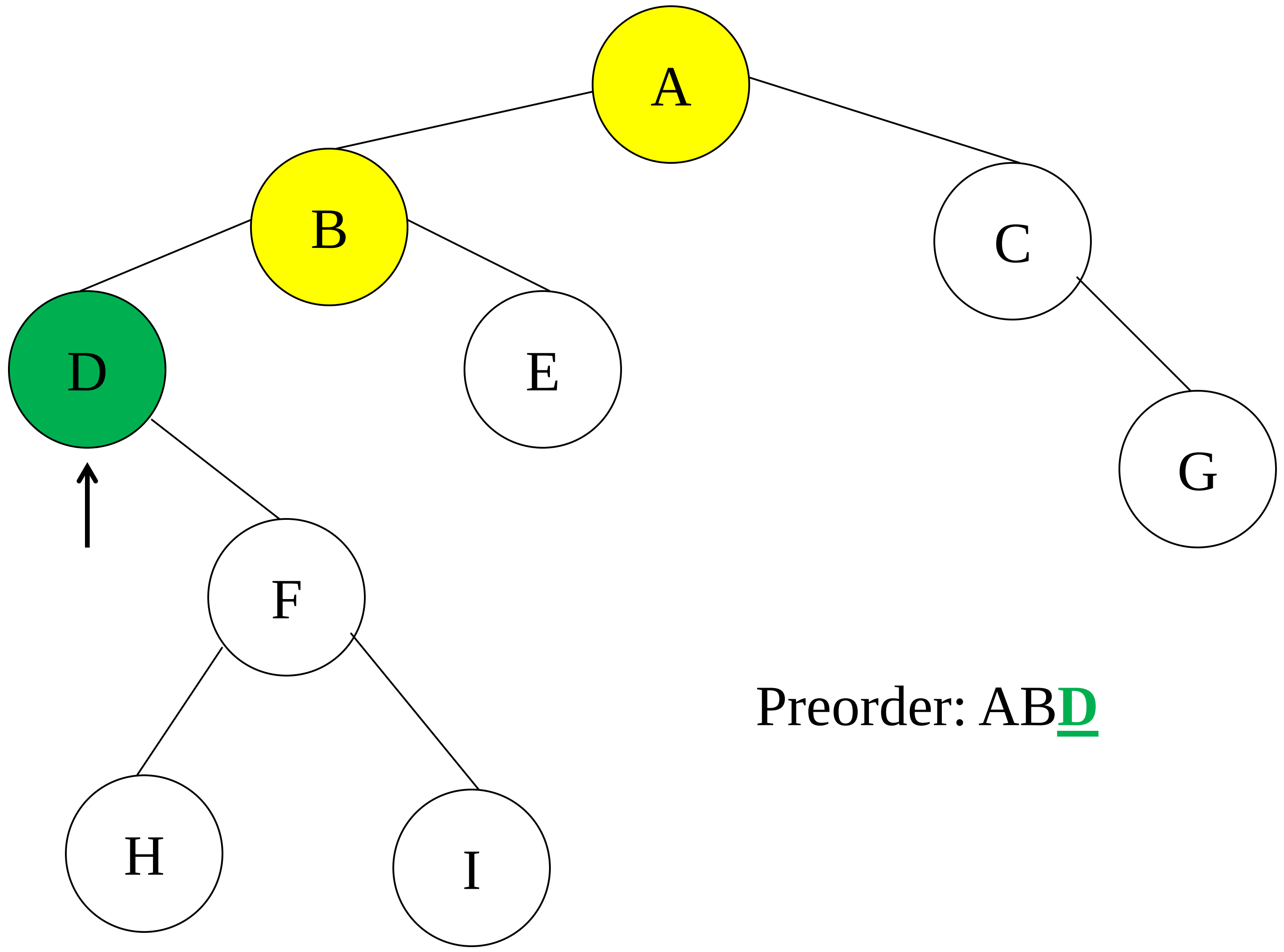
Preorder: **A**

Tree Traversals: Another Example



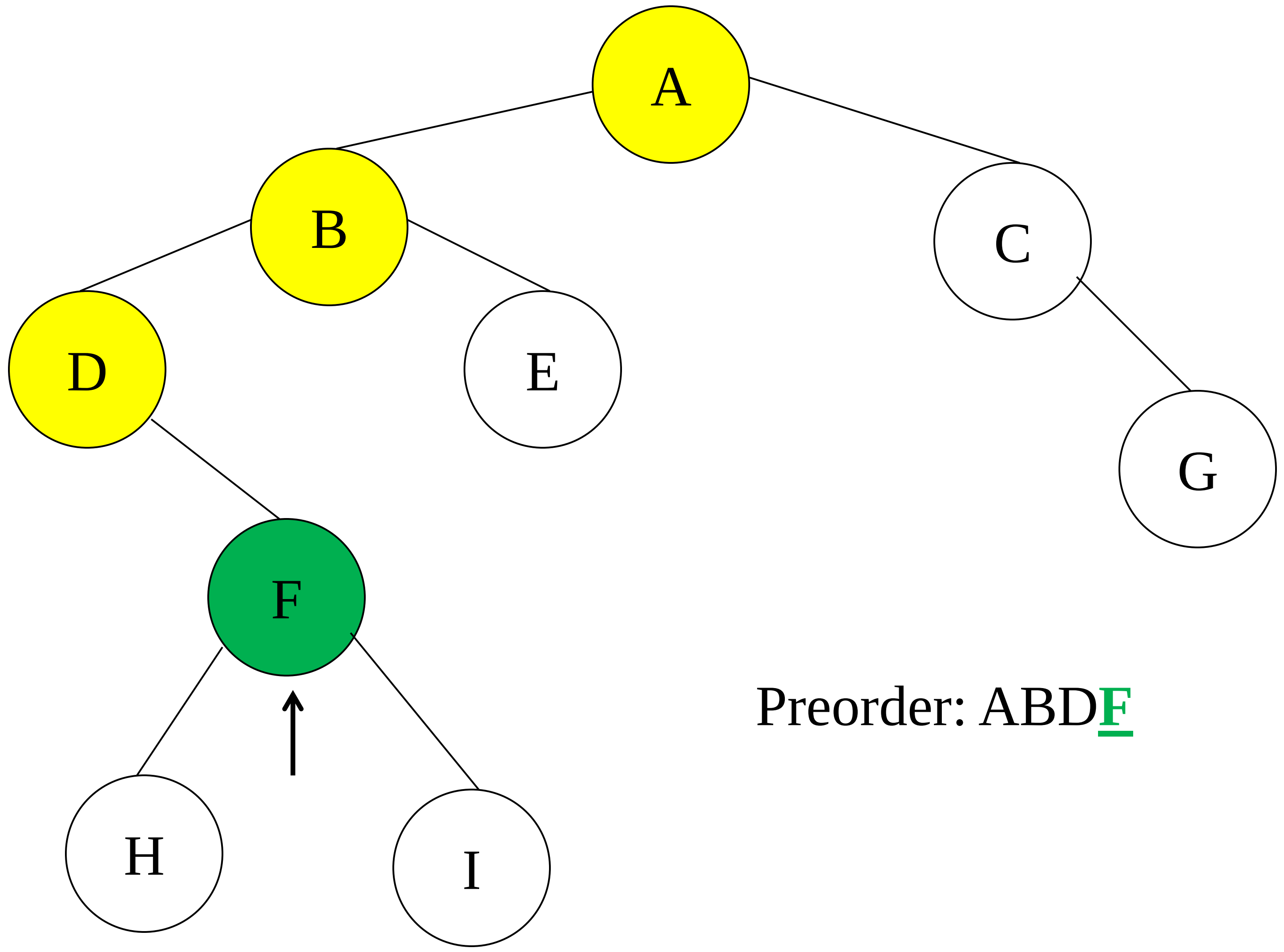
Preorder: A**<u>B</u>**

Tree Traversals: Another Example



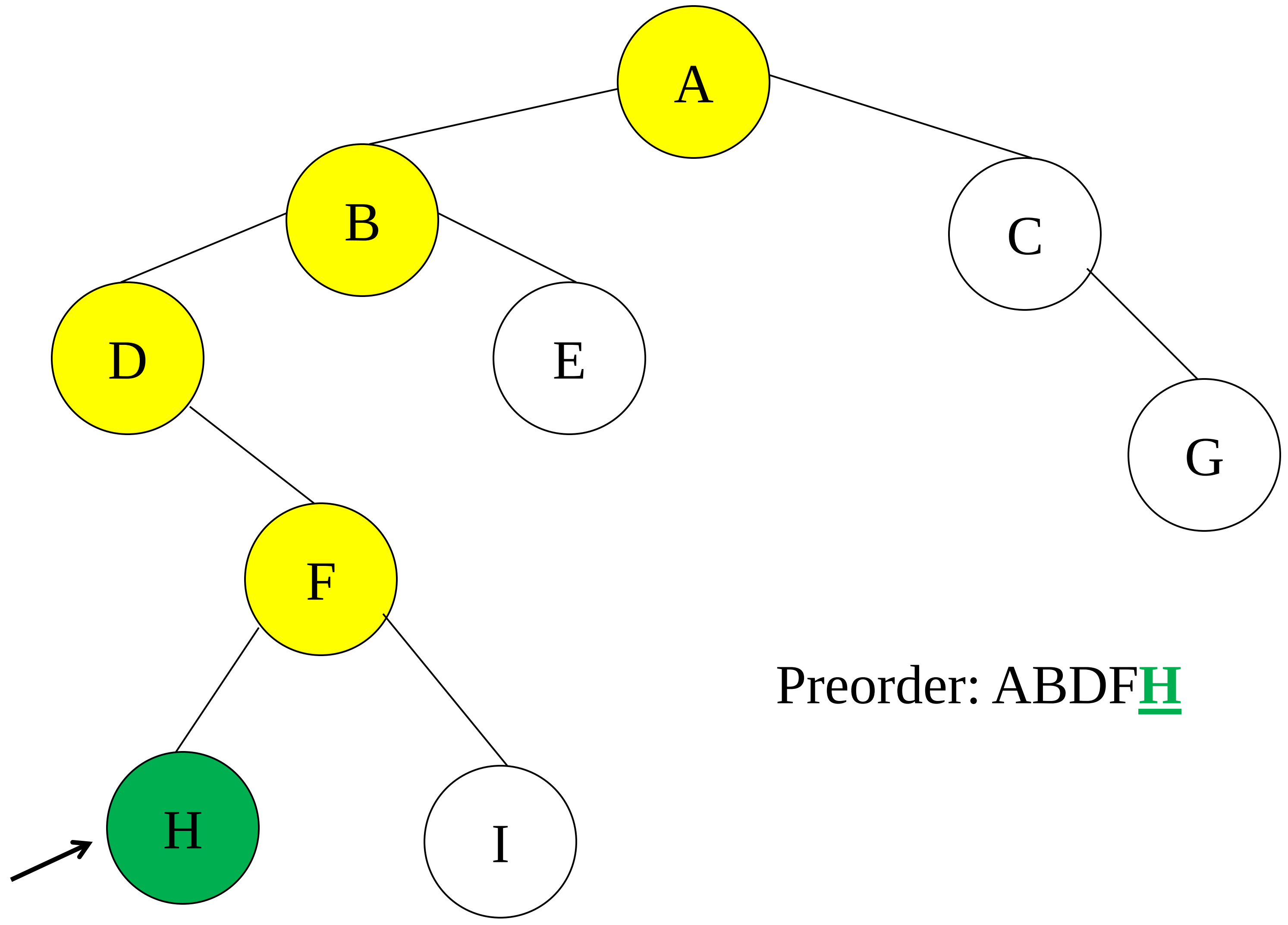
Preorder: AB**D**

Tree Traversals: Another Example



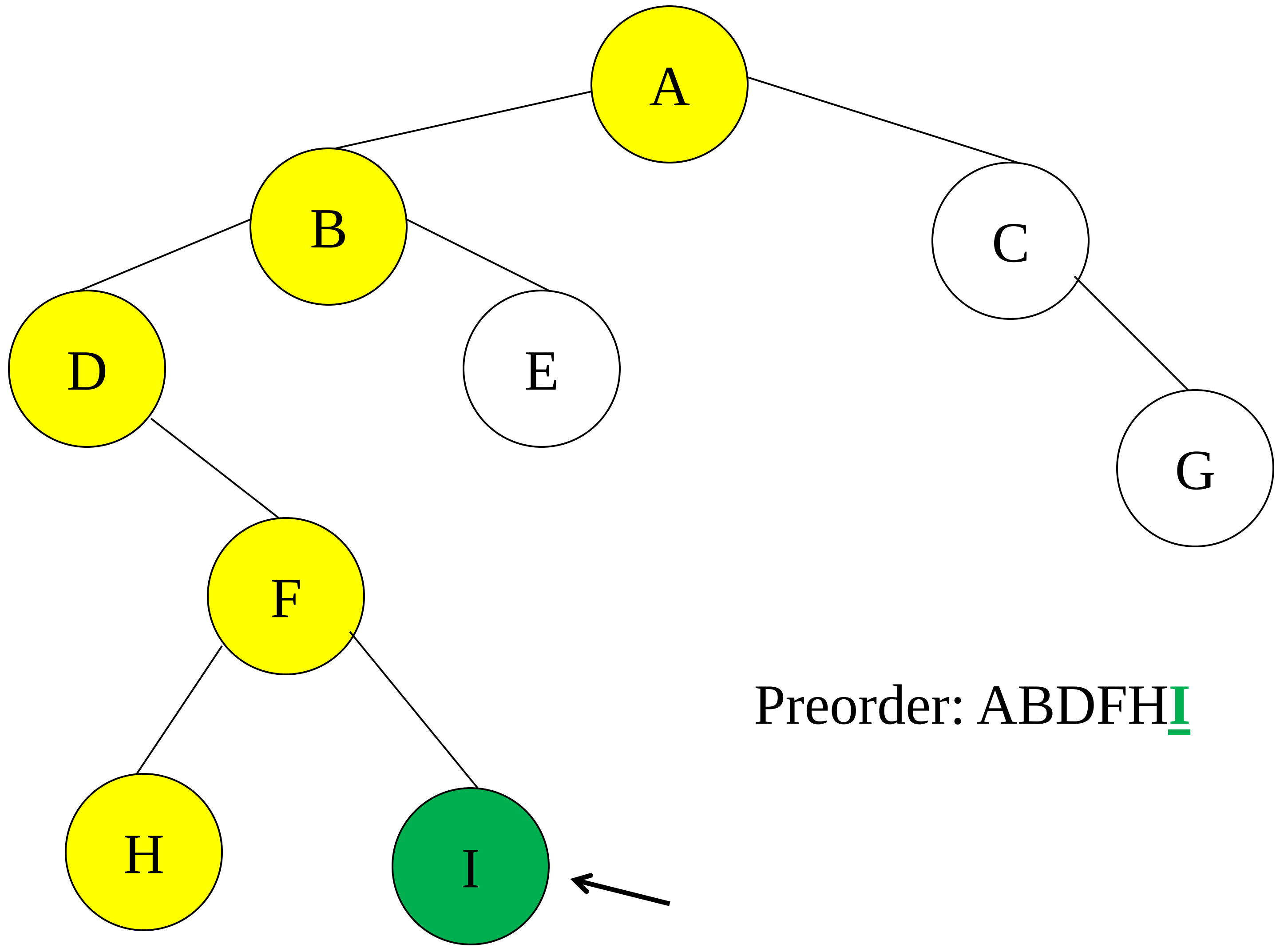
Preorder: ABD**F**

Tree Traversals: Another Example



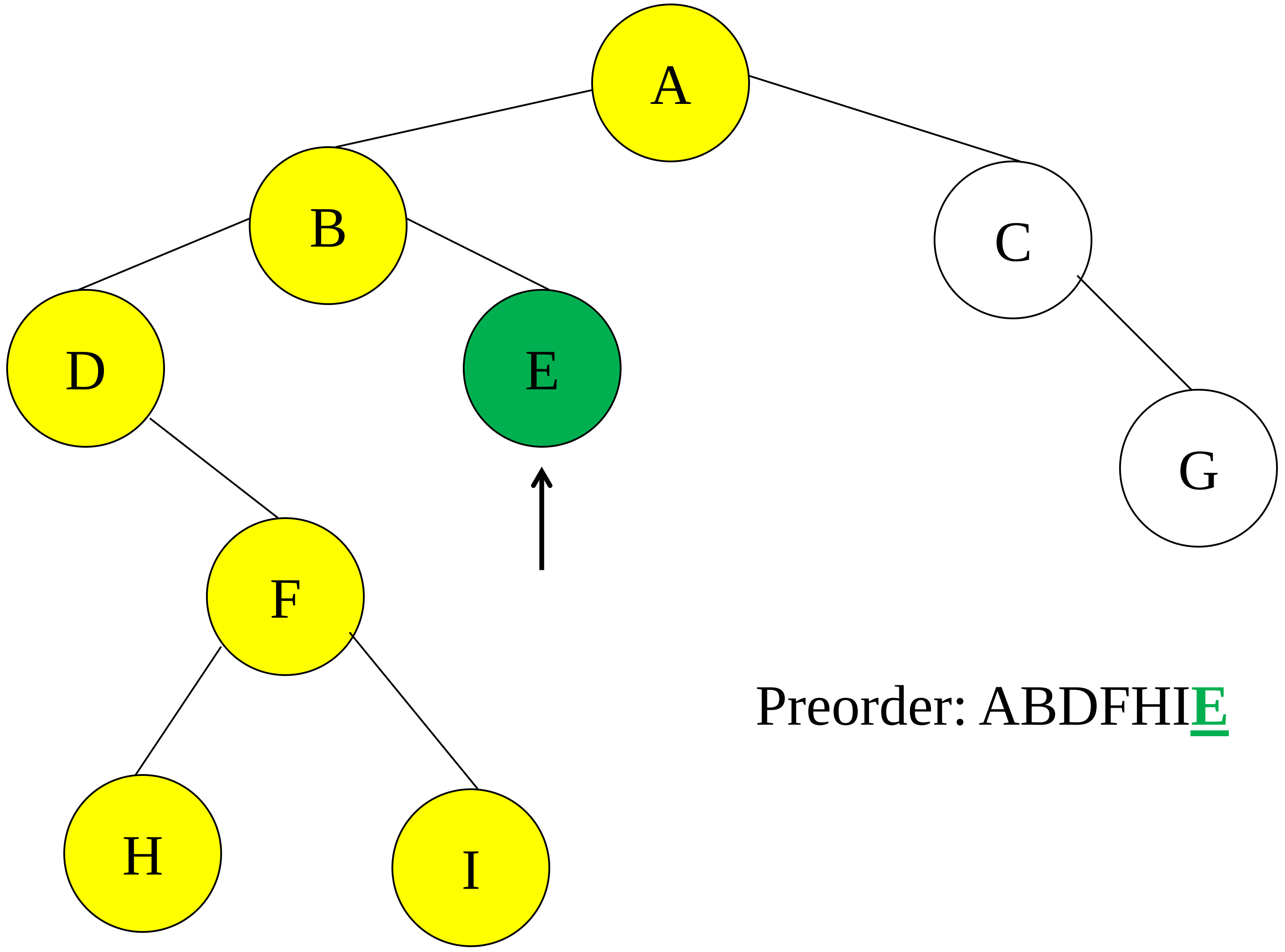
Preorder: ABDF**H**

Tree Traversals: Another Example



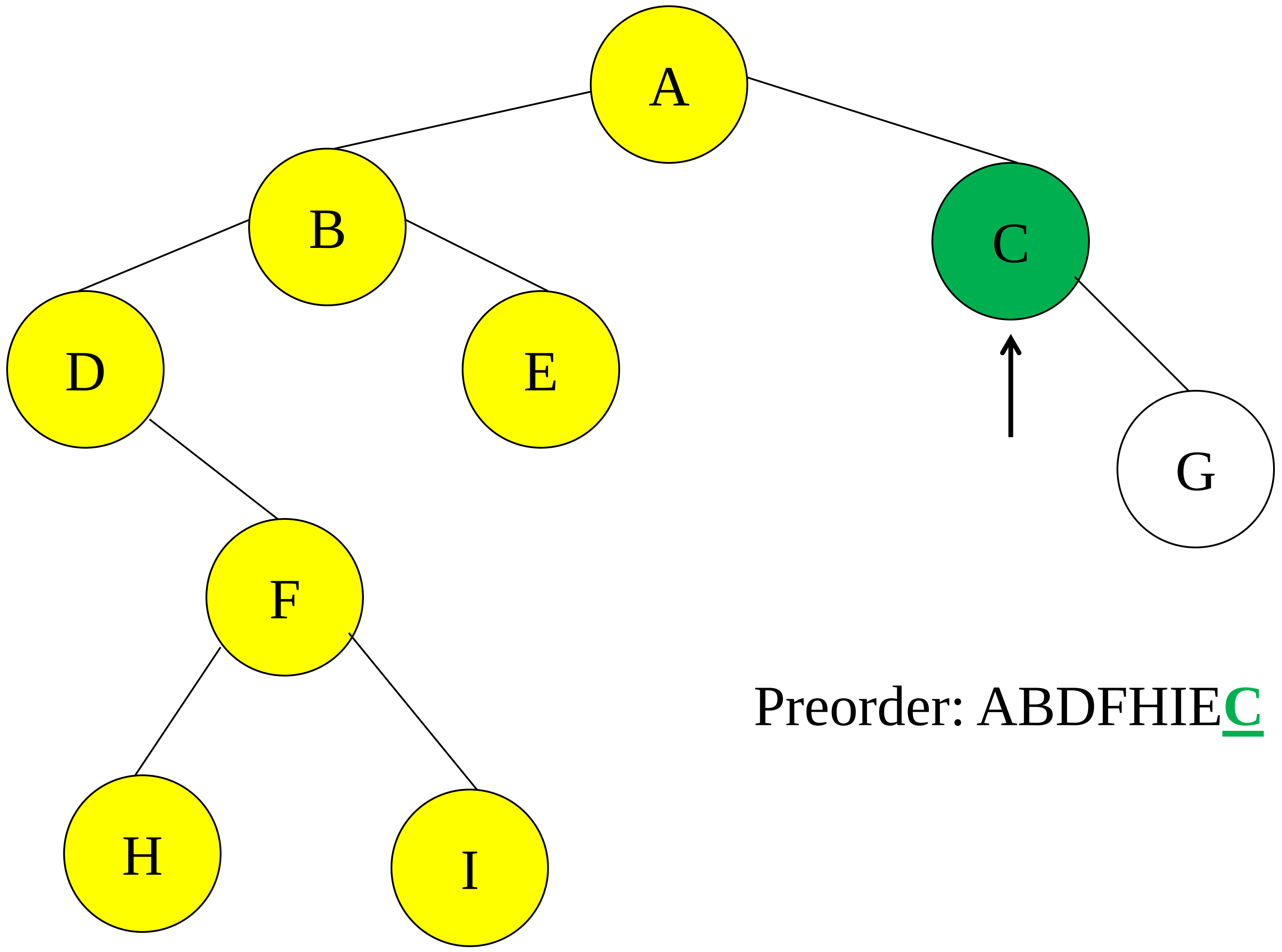
Preorder: ABDFH**I**

Tree Traversals: Another Example



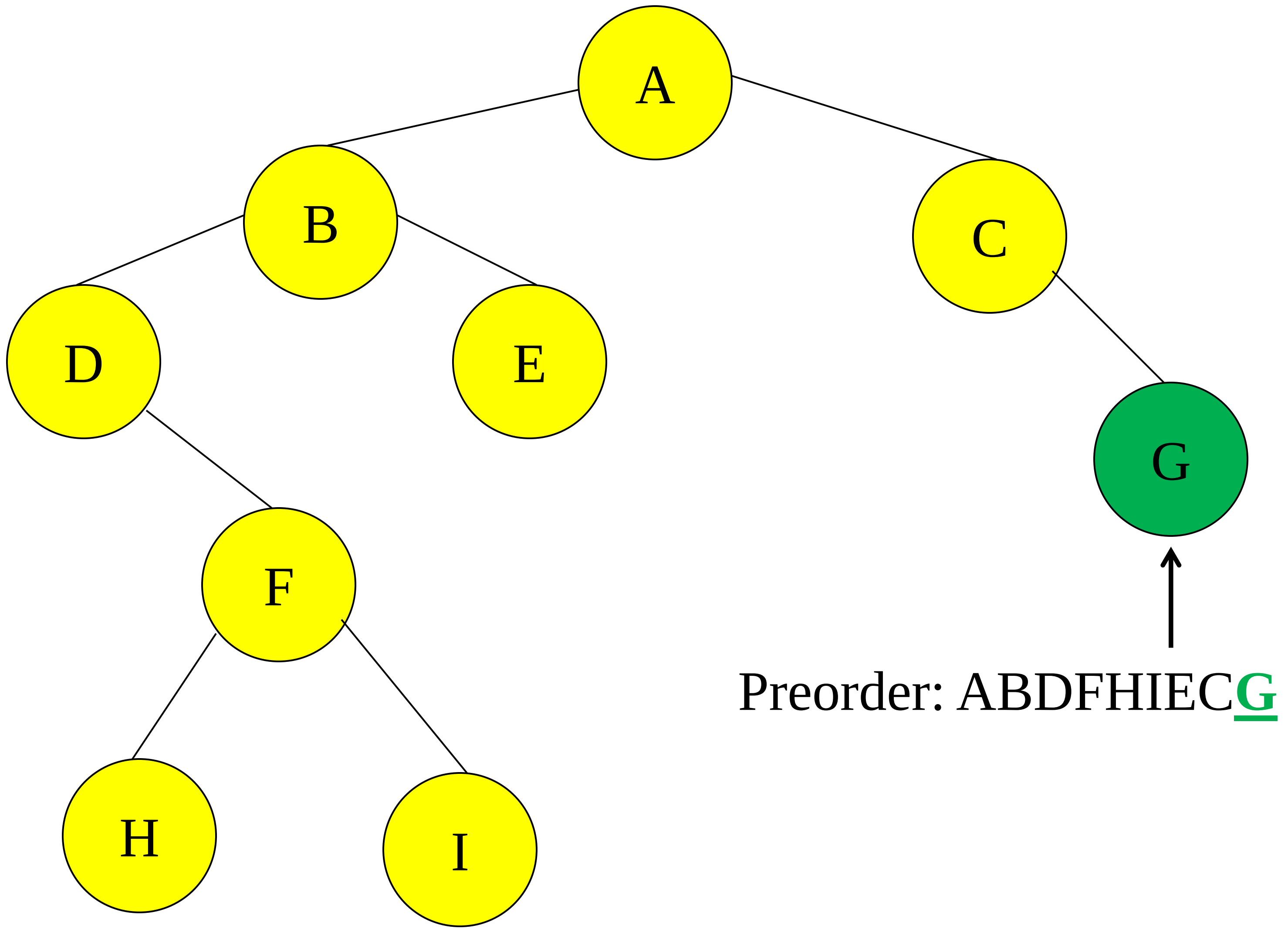
Preorder: ABDFHI**E**

Tree Traversals: Another Example



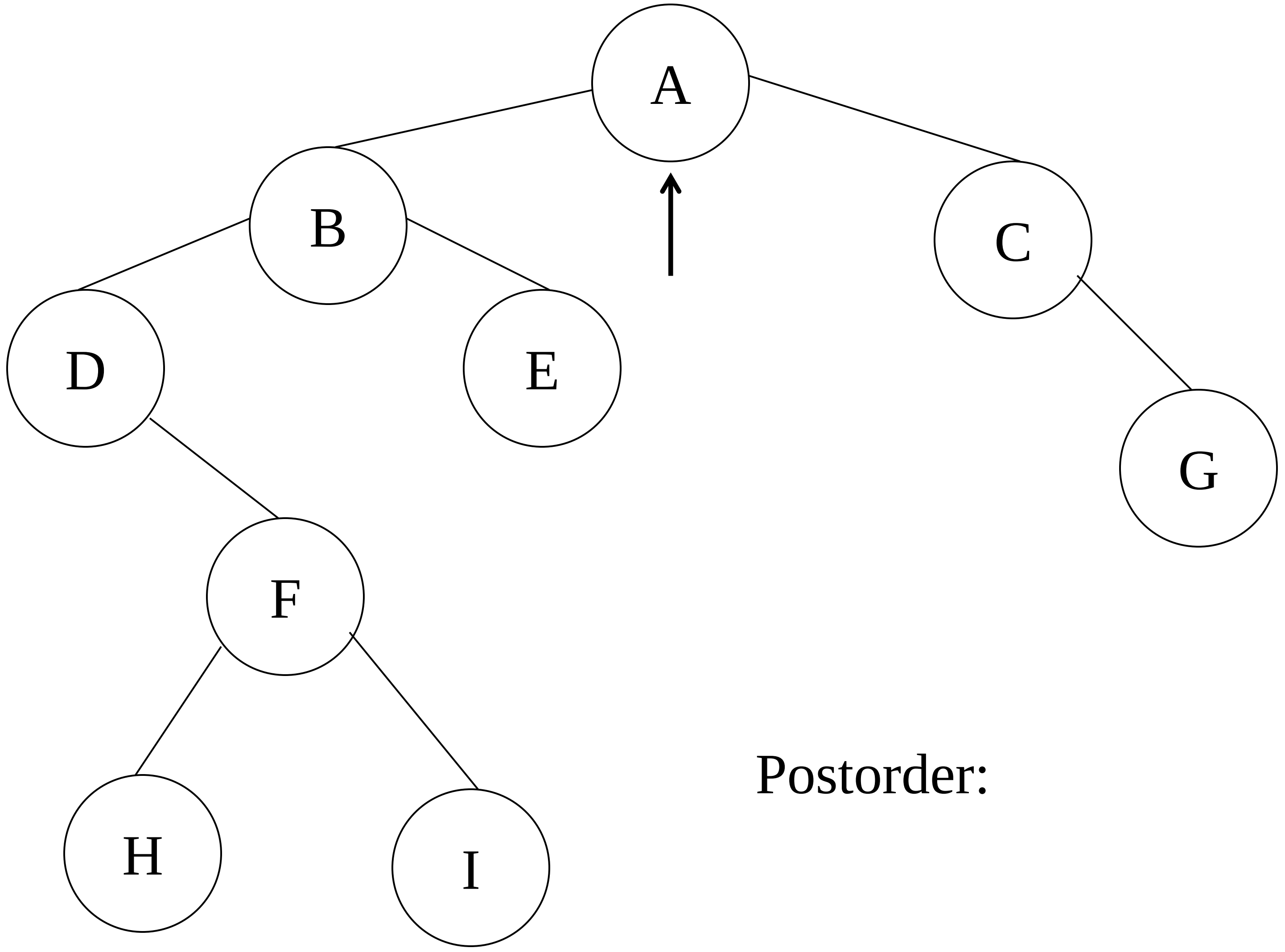
Preorder: ABDFHIE**C**

Tree Traversals: Another Example



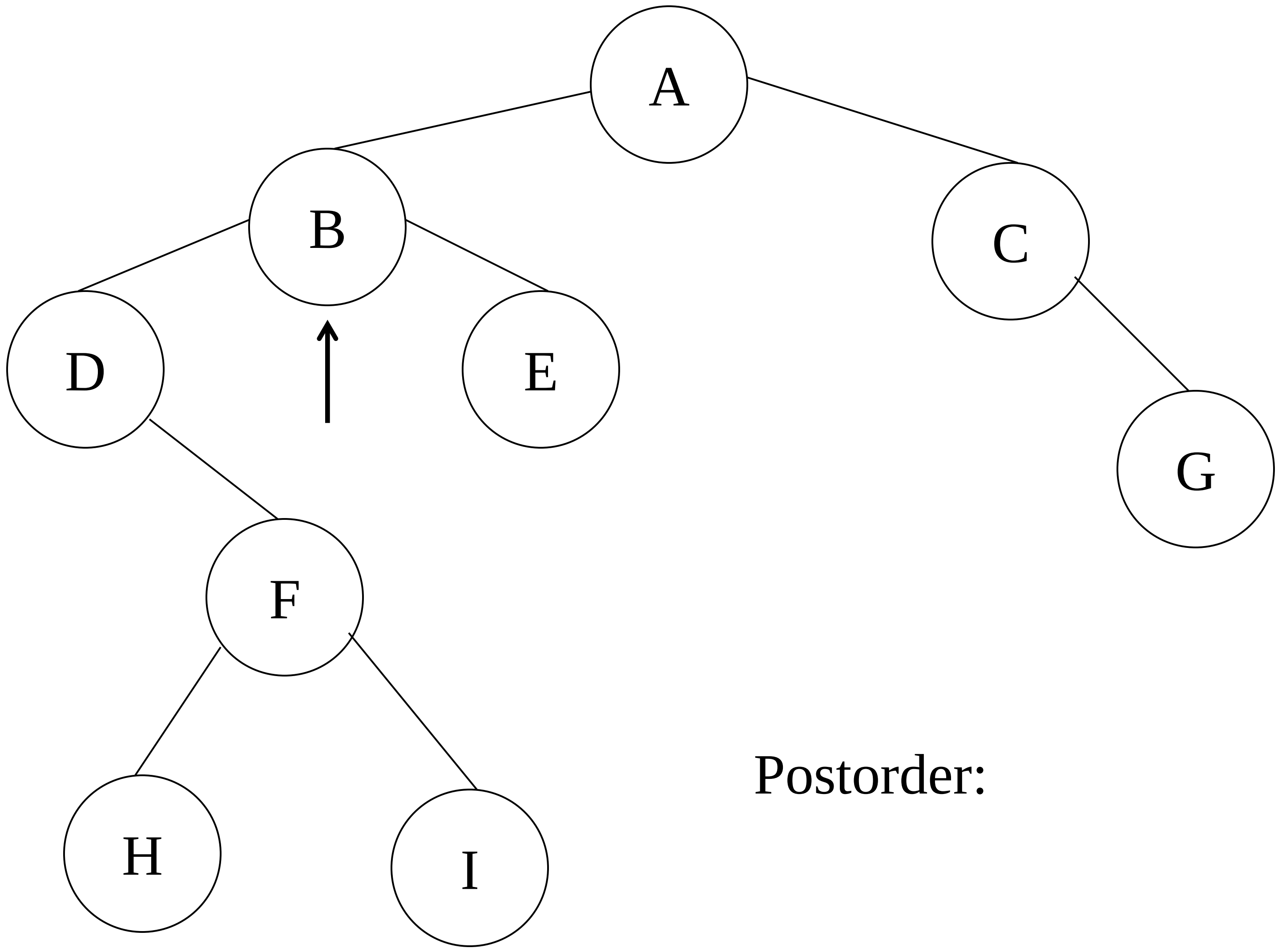
Preorder: ABDFHIEC**G**

Tree Traversals: Another Example



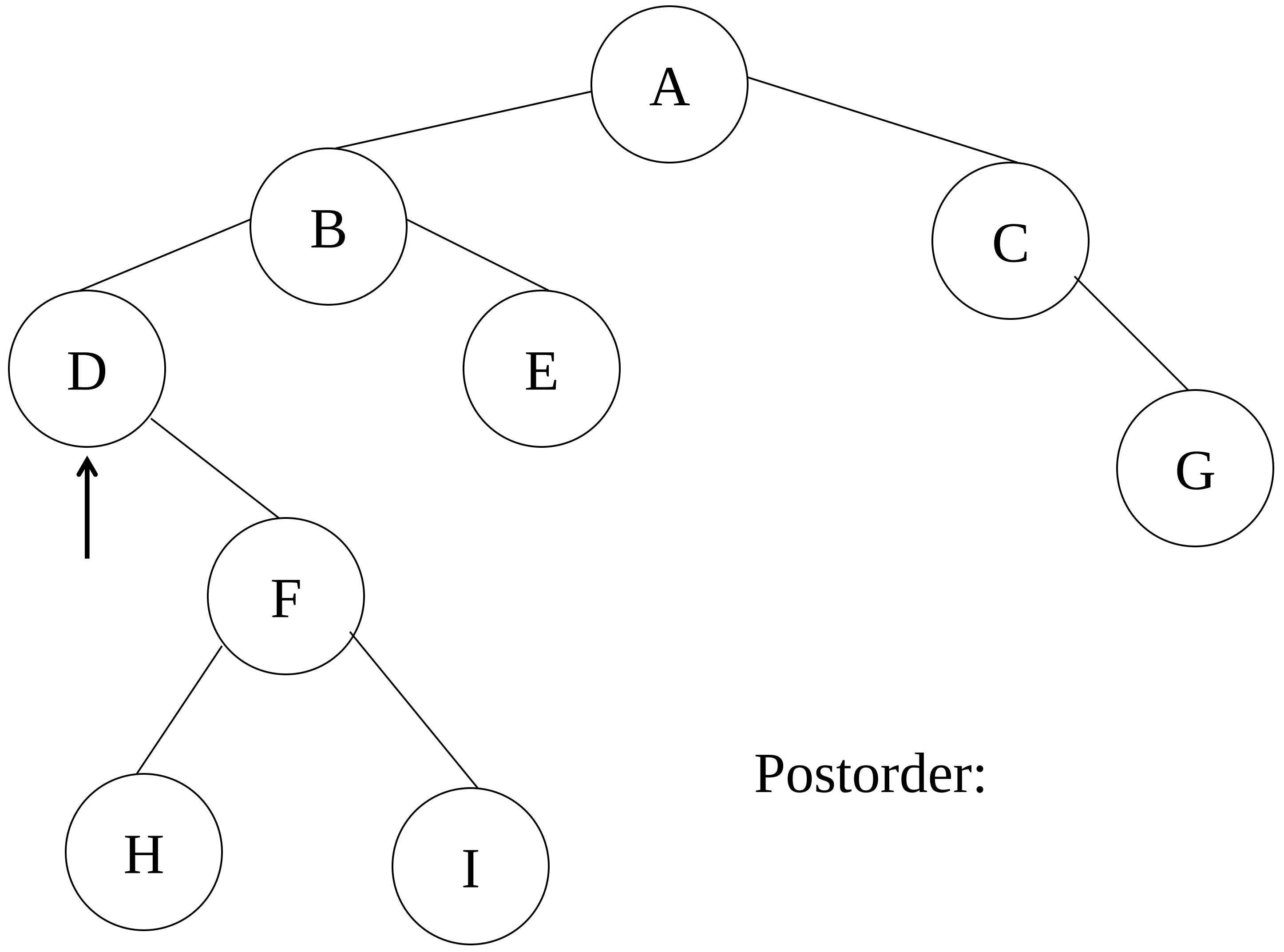
Postorder:

Tree Traversals: Another Example



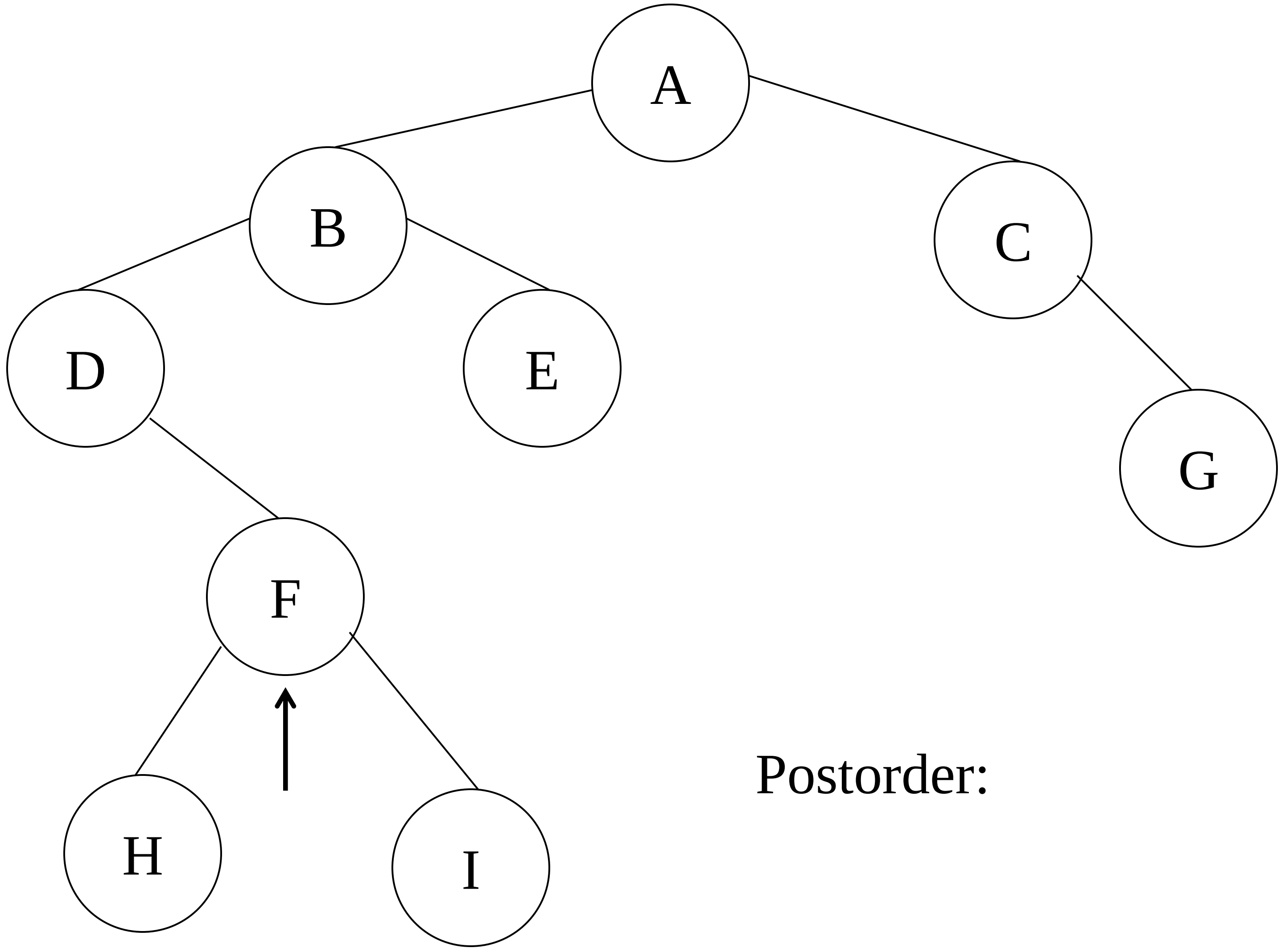
Postorder:

Tree Traversals: Another Example



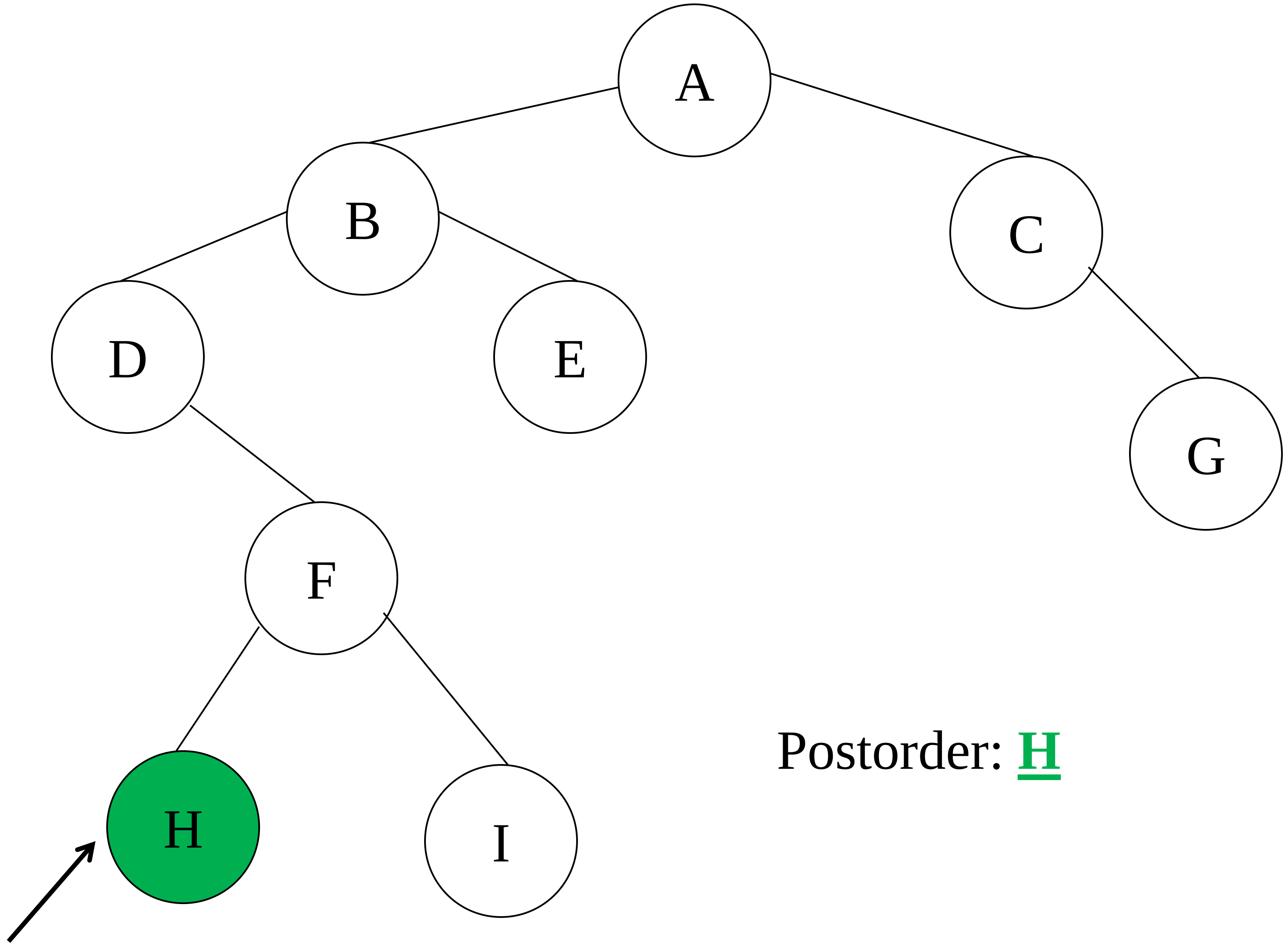
Postorder:

Tree Traversals: Another Example



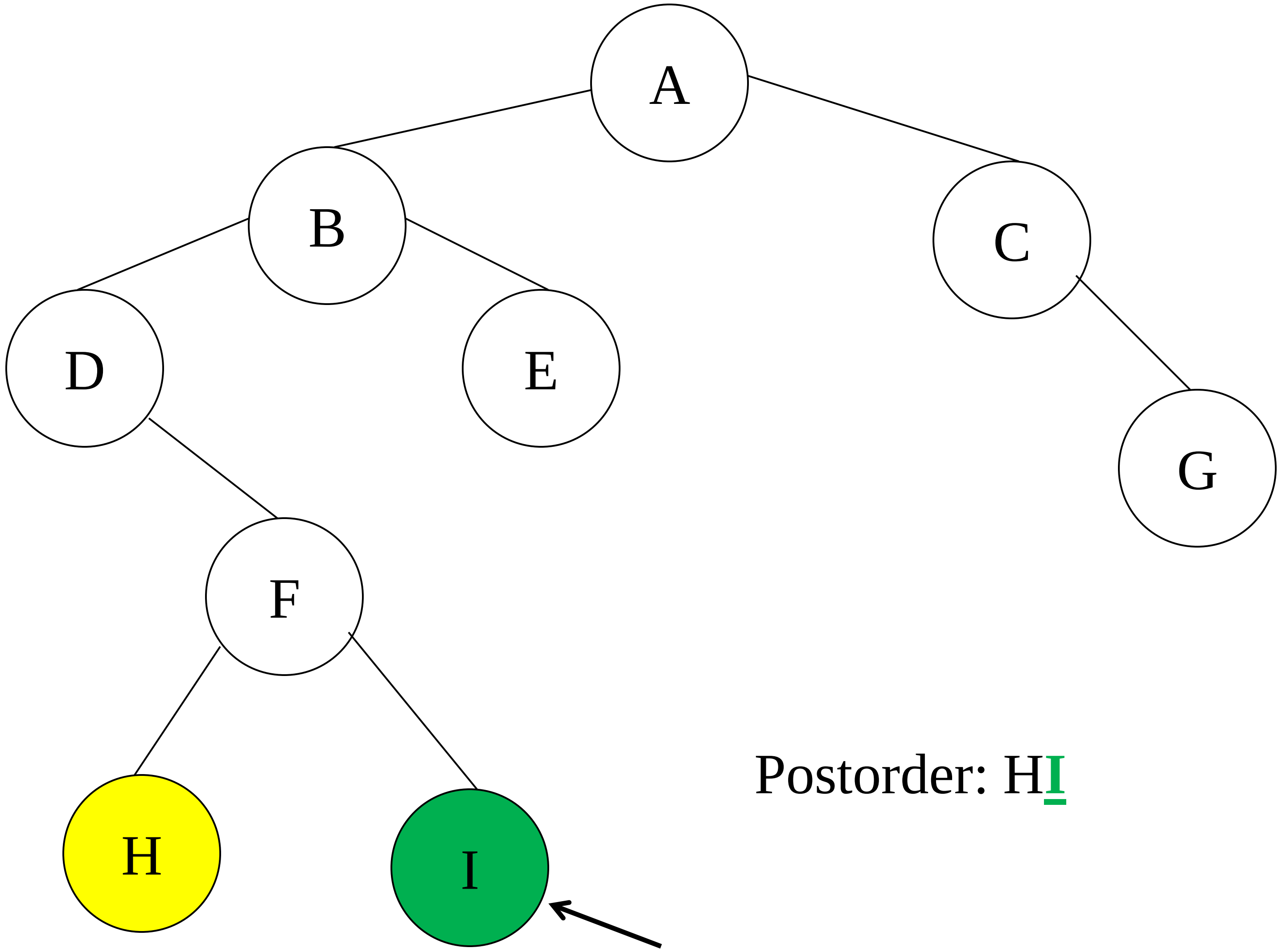
Postorder:

Tree Traversals: Another Example



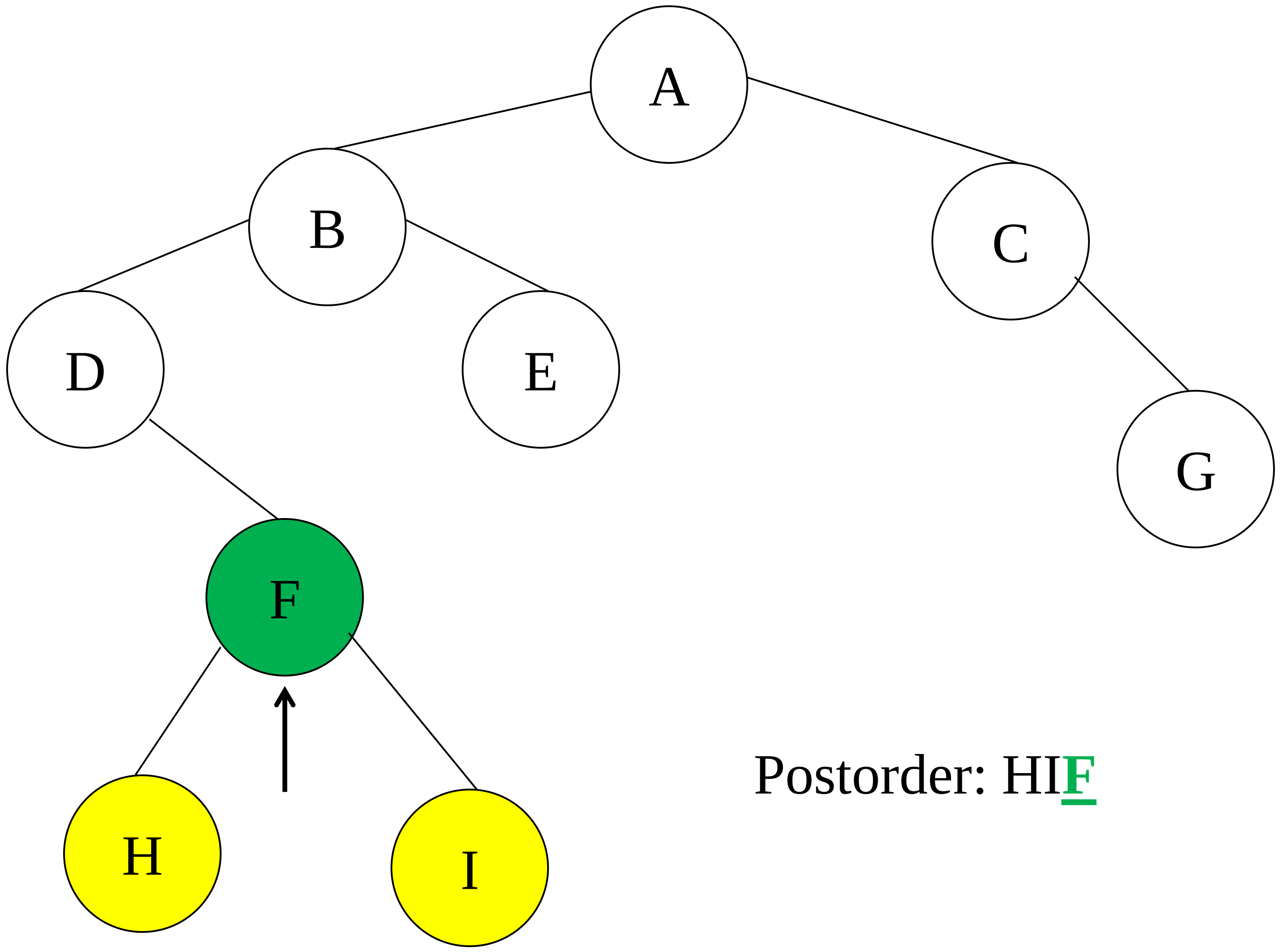
Postorder: **H**

Tree Traversals: Another Example



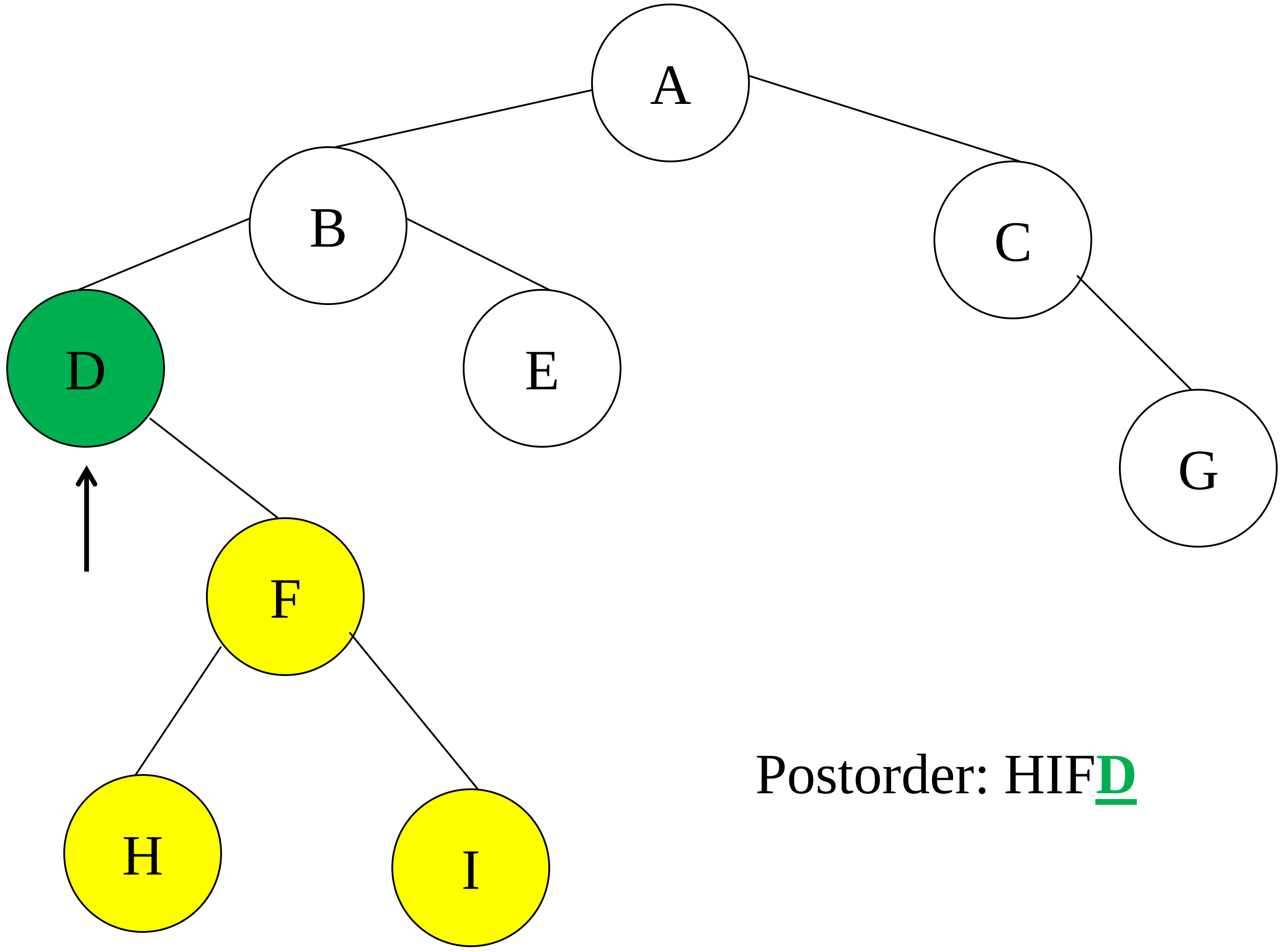
Postorder: H**I**

Tree Traversals: Another Example



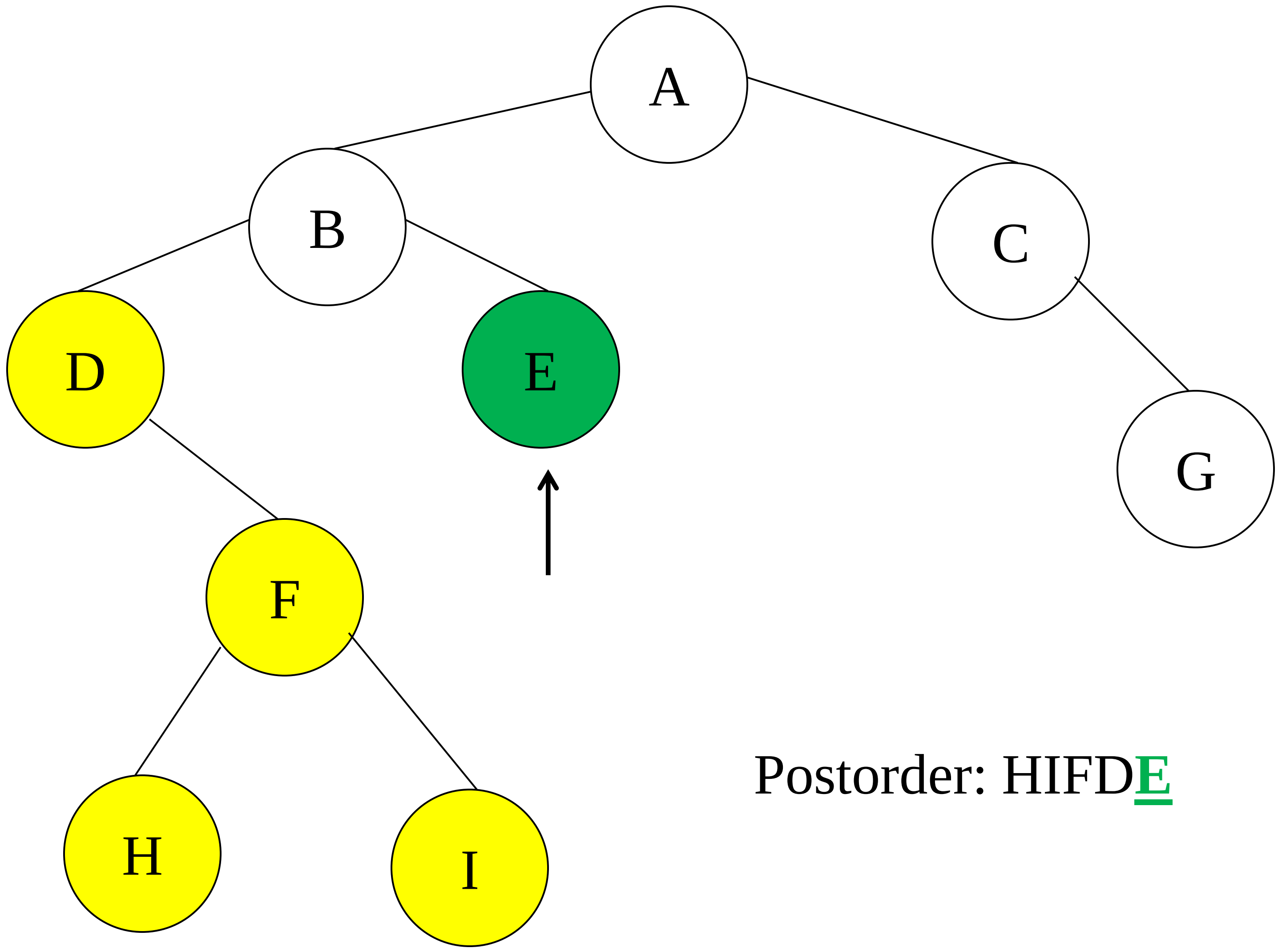
Postorder: HI**F**

Tree Traversals: Another Example



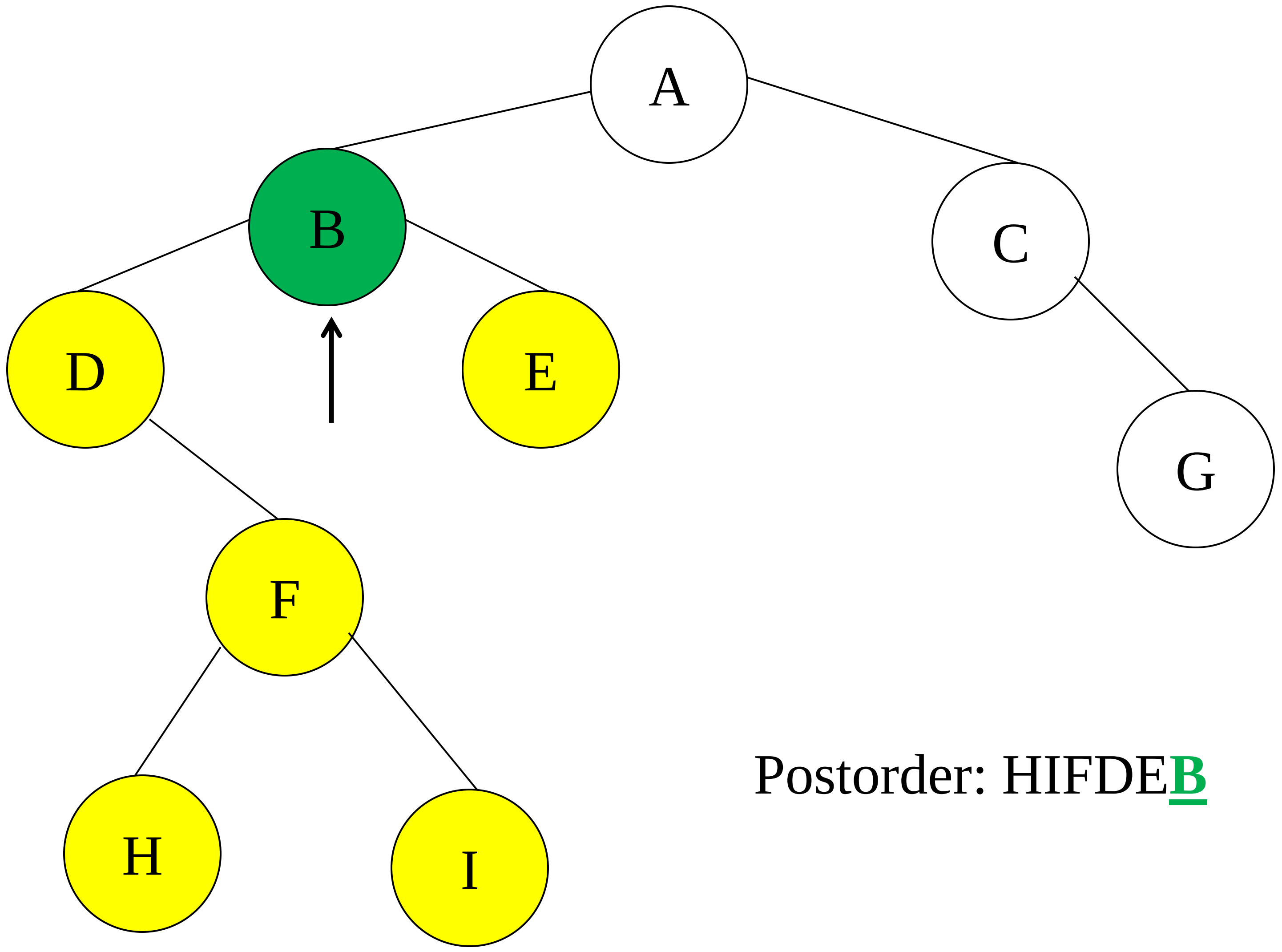
Postorder: HIF**D**

Tree Traversals: Another Example



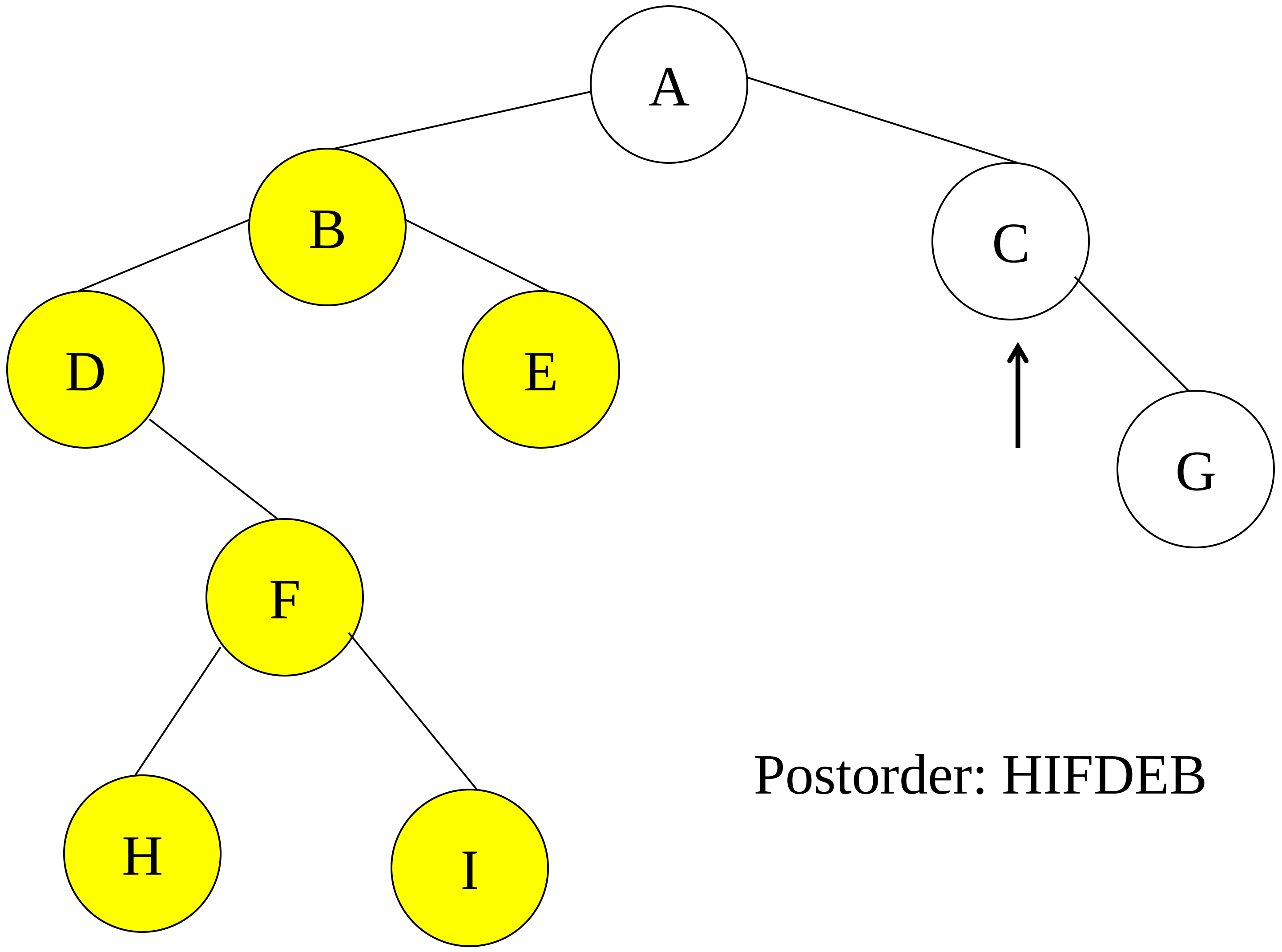
Postorder: HIFD**E**

Tree Traversals: Another Example



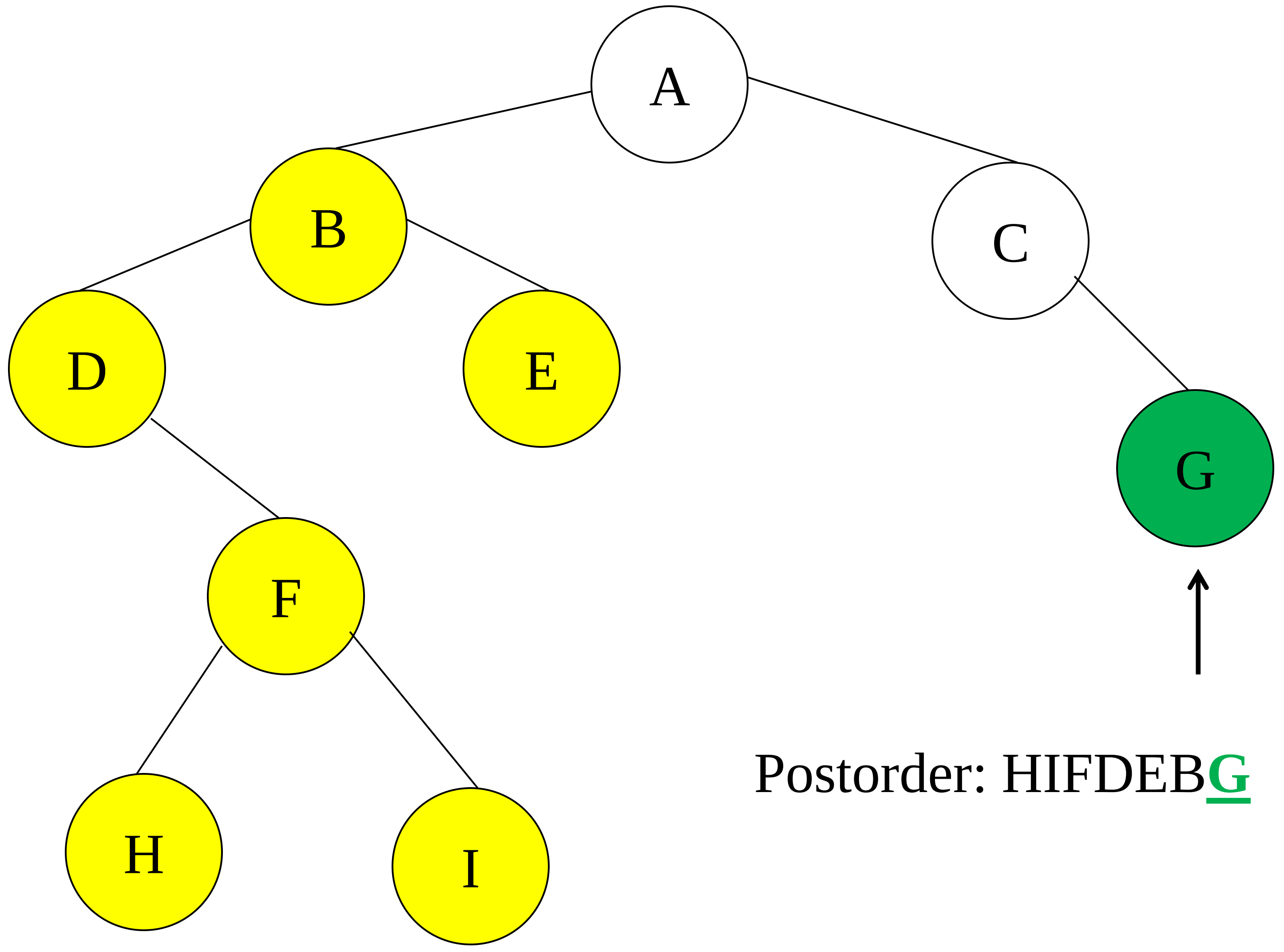
Postorder: HIFDE**B**

Tree Traversals: Another Example



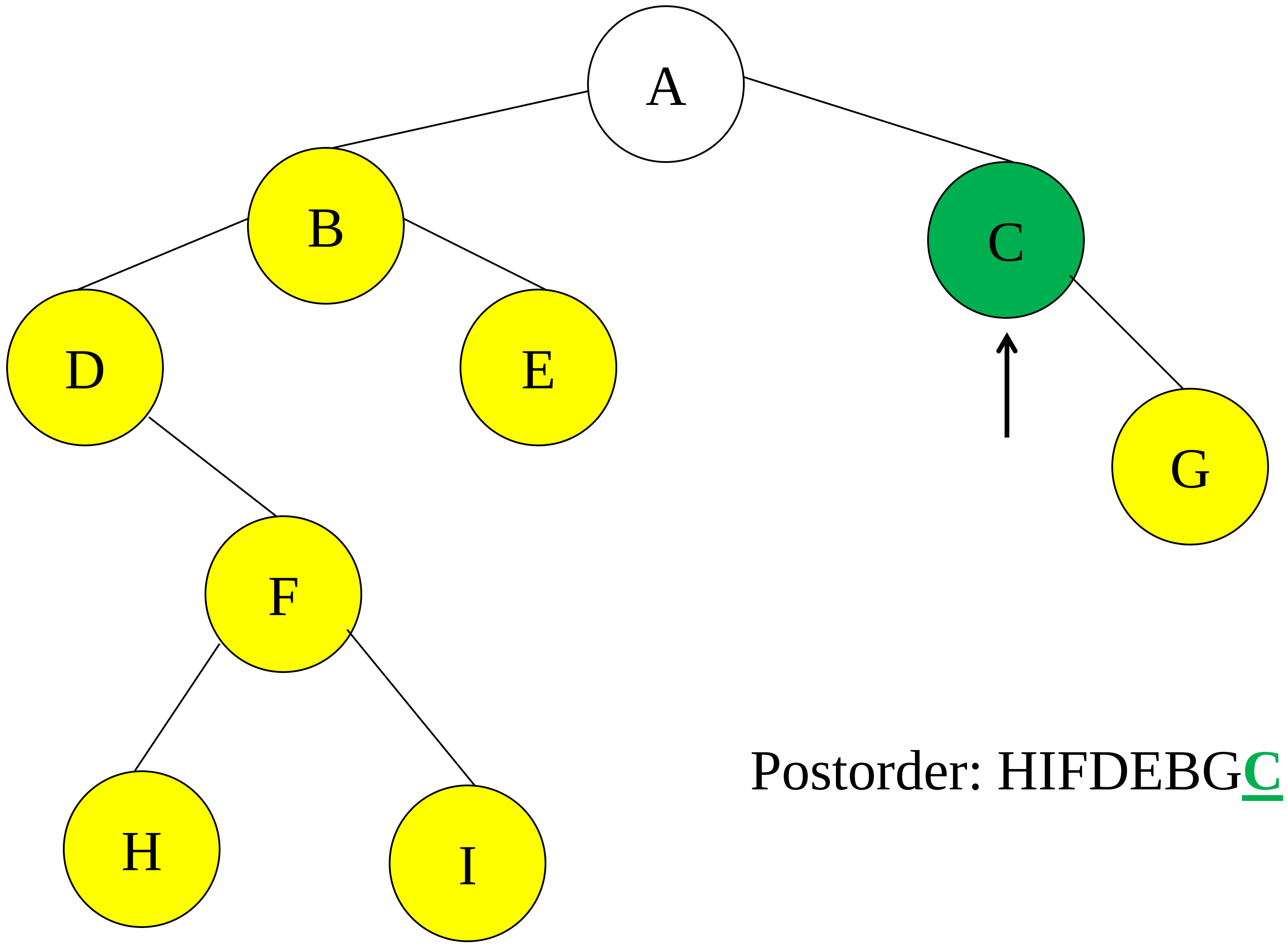
Postorder: HIFDEB

Tree Traversals: Another Example



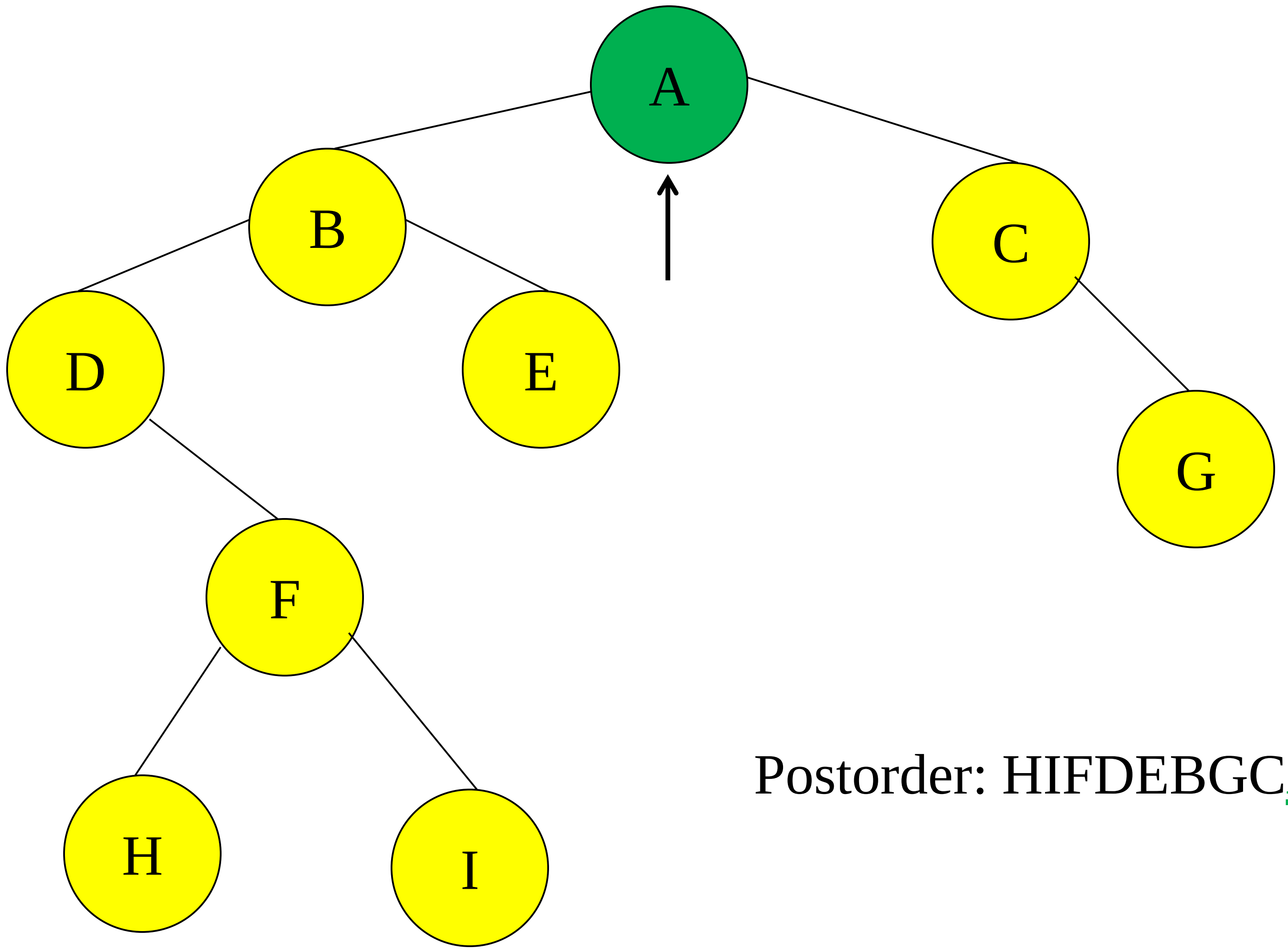
Postorder: HIFDEB**G**

Tree Traversals: Another Example



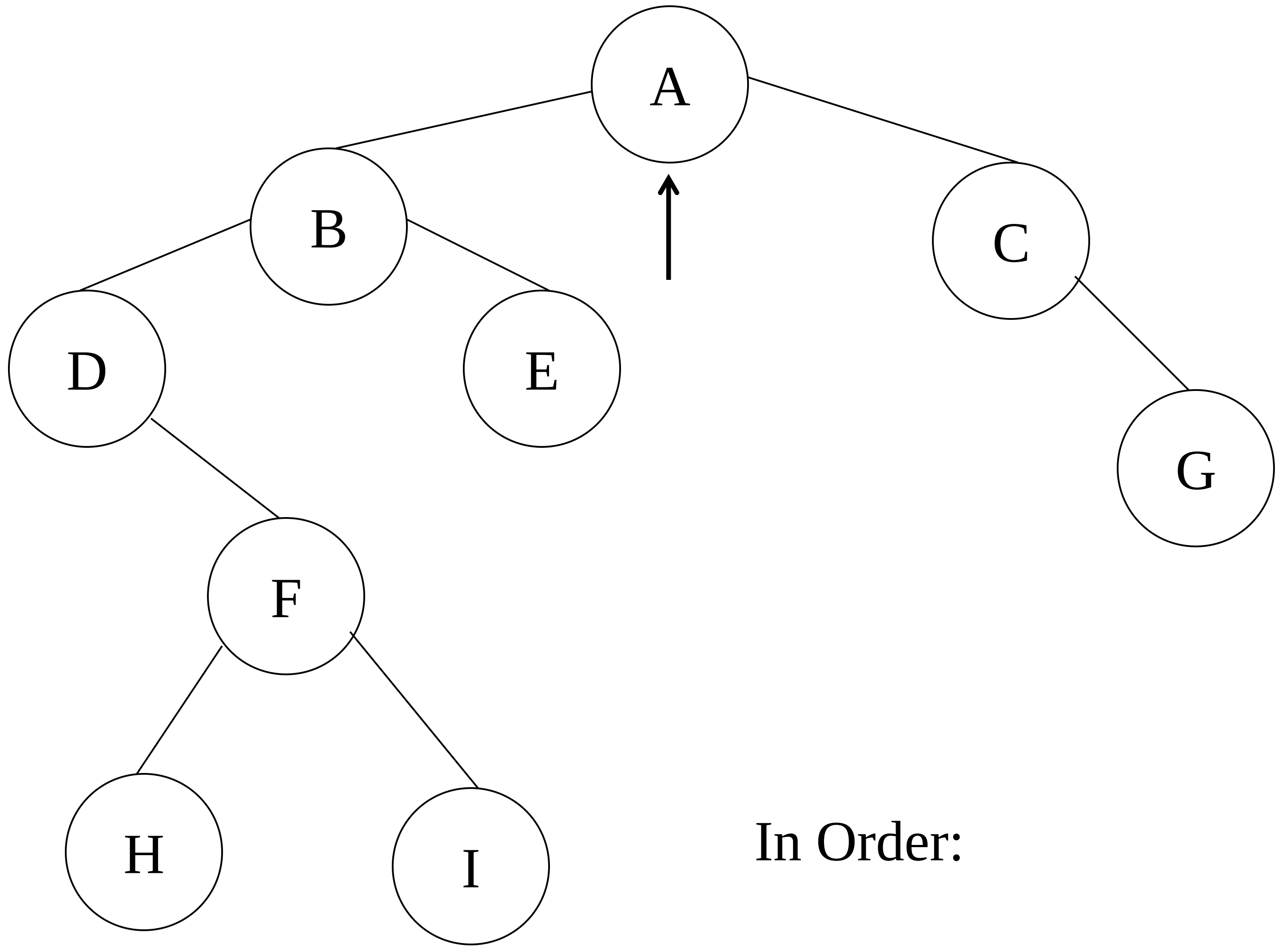
Postorder: HIFDEBG**C**

Tree Traversals: Another Example



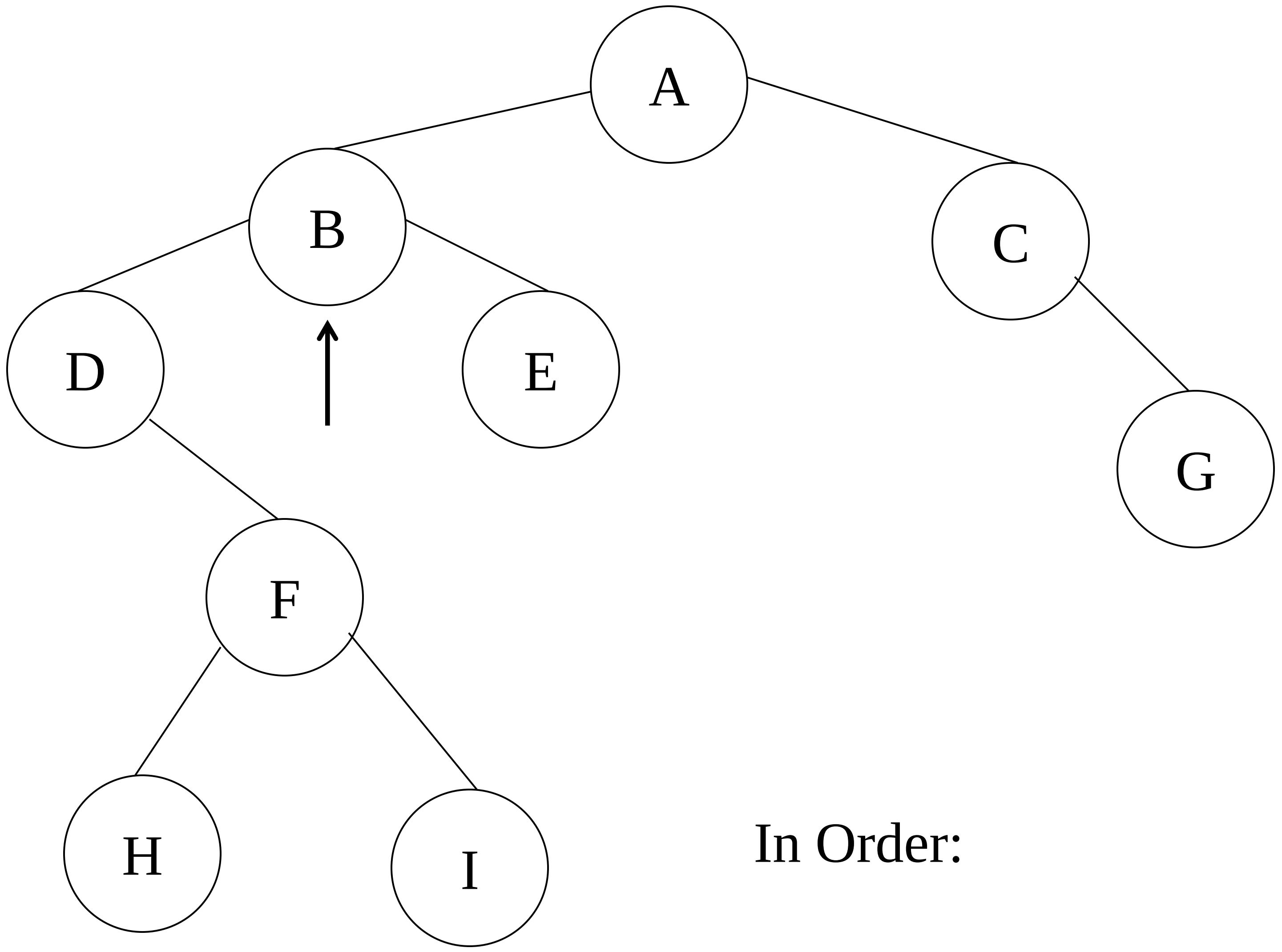
Postorder: HIFDEBGC**A**

Tree Traversals: Another Example



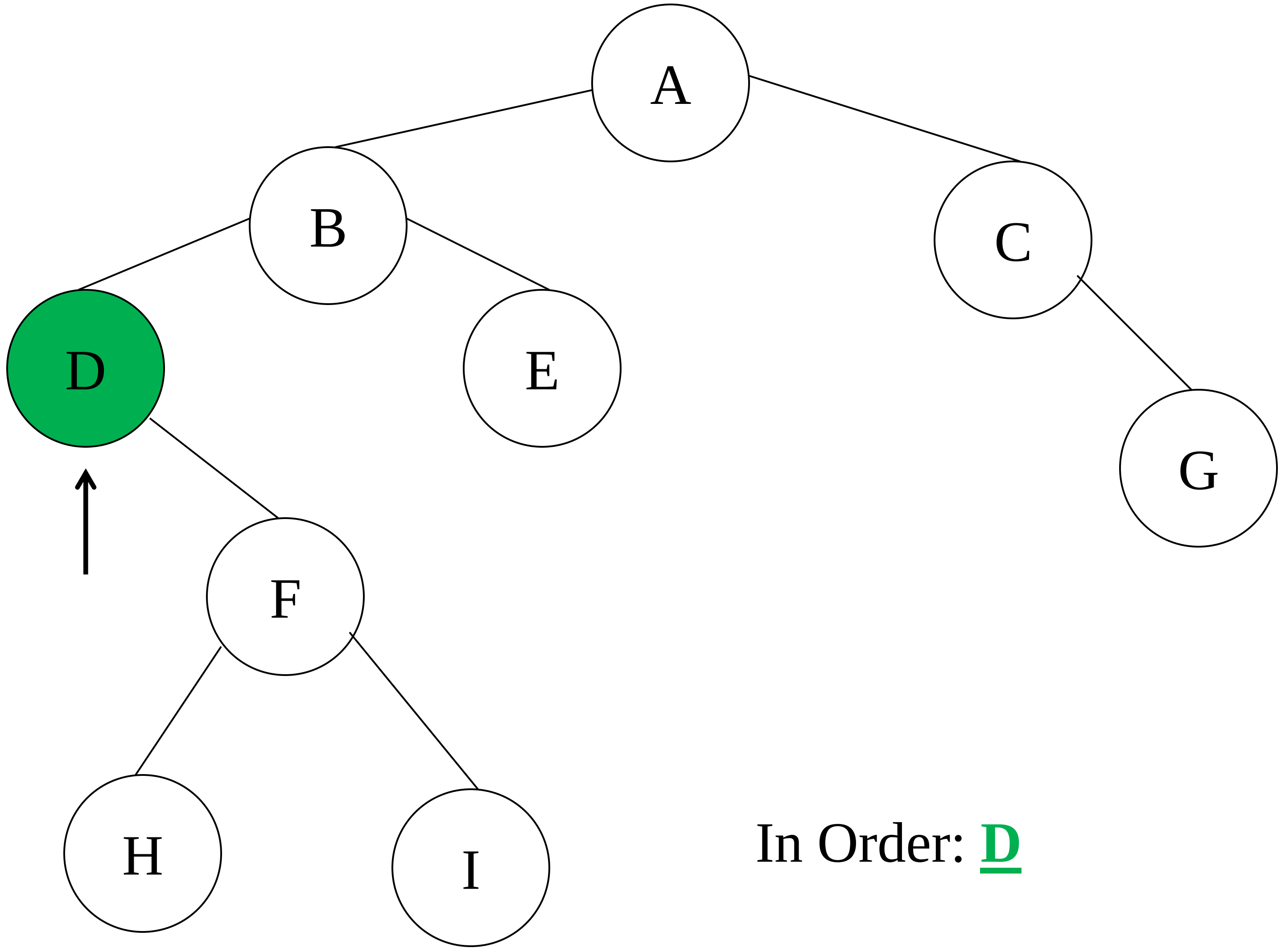
In Order:

Tree Traversals: Another Example



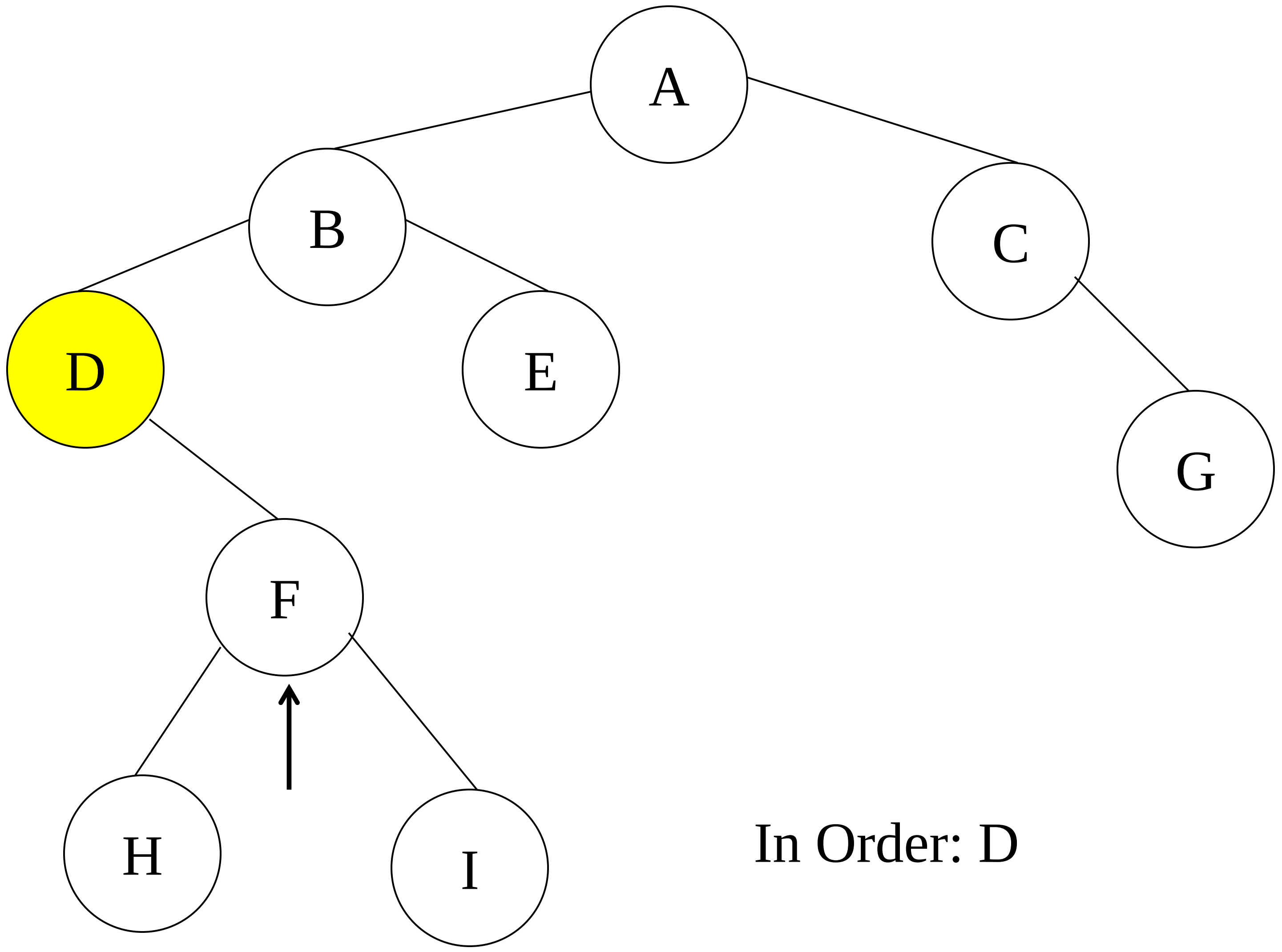
In Order:

Tree Traversals: Another Example



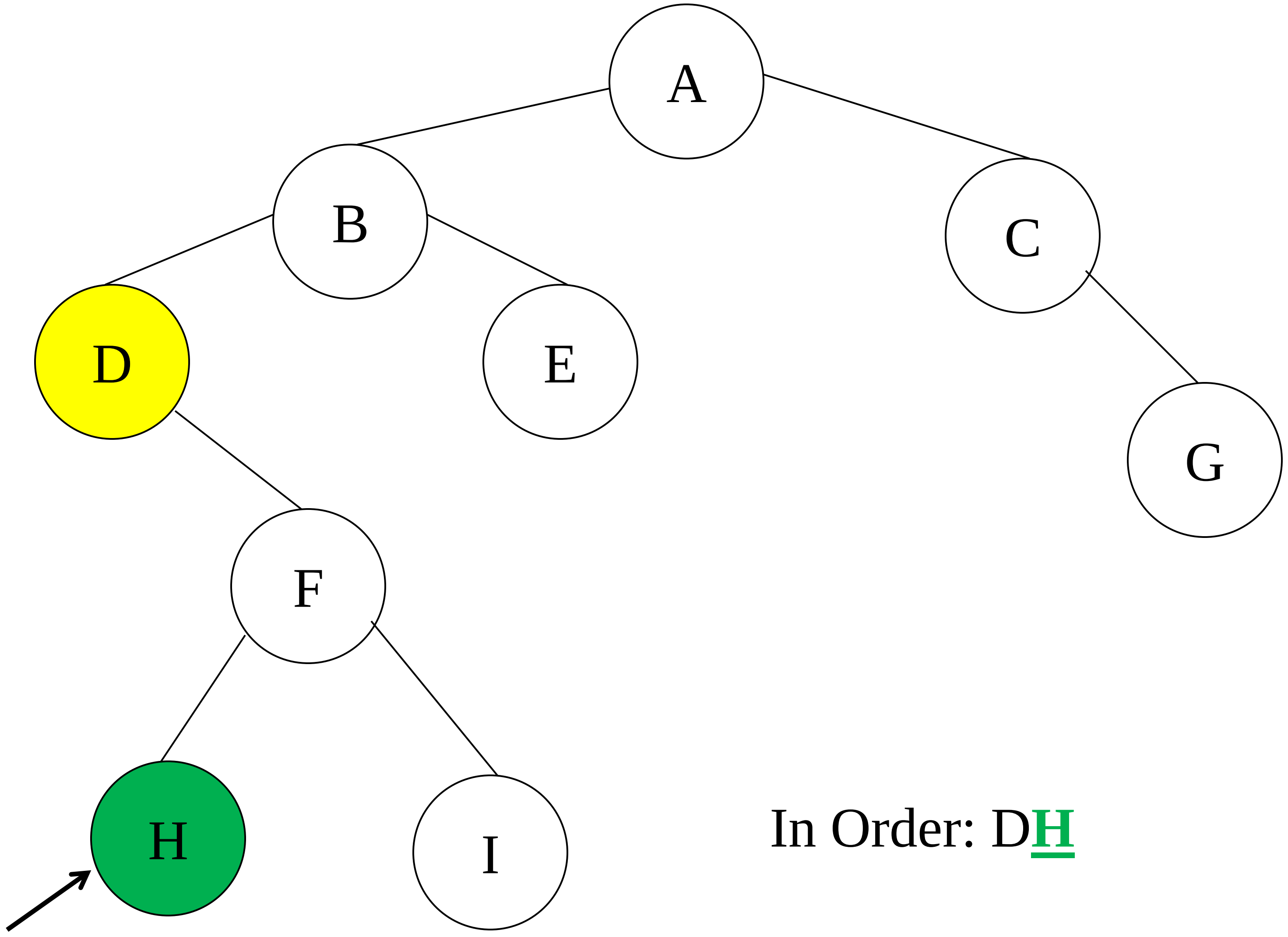
In Order: **D**

Tree Traversals: Another Example



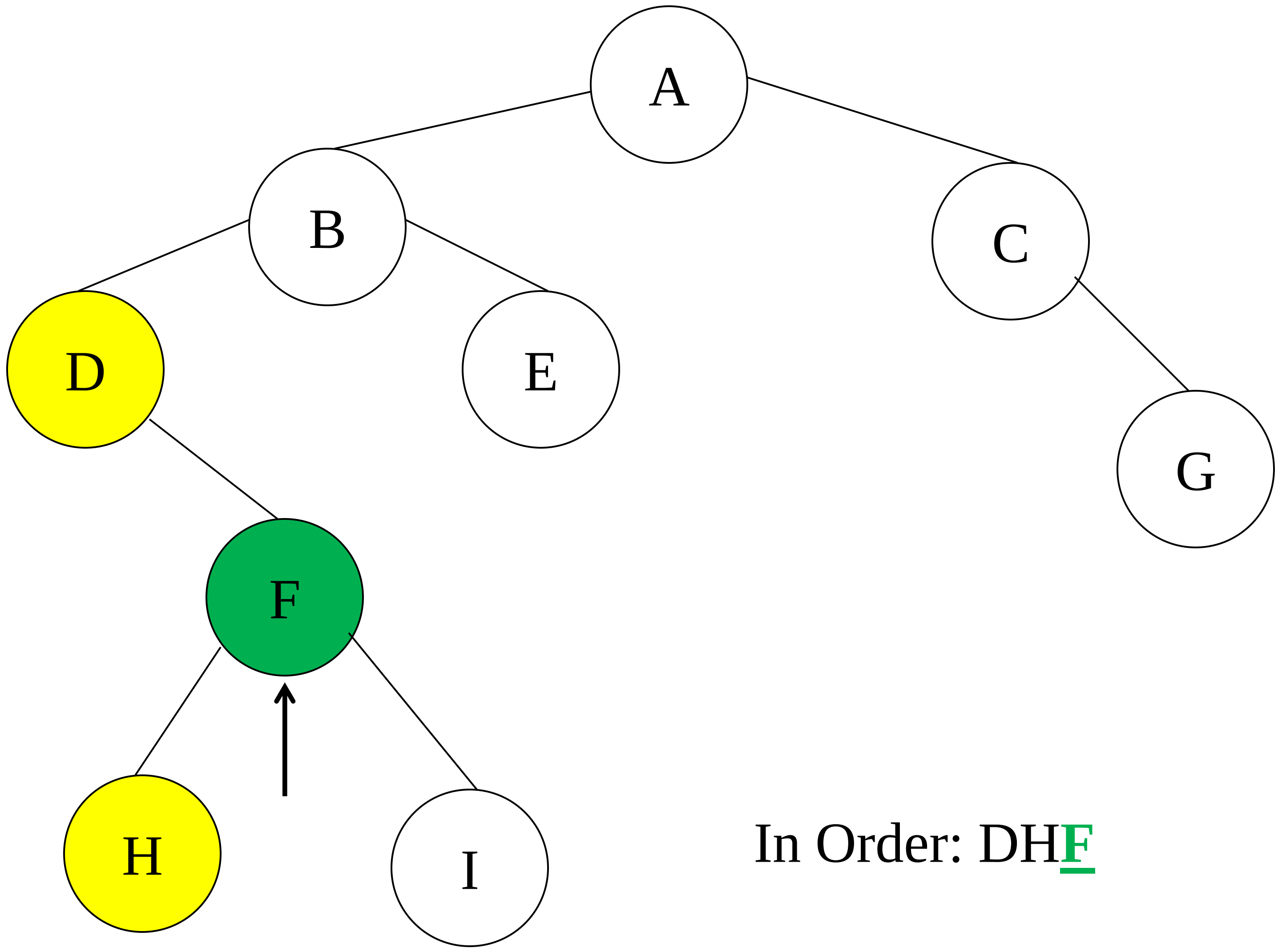
In Order: D

Tree Traversals: Another Example



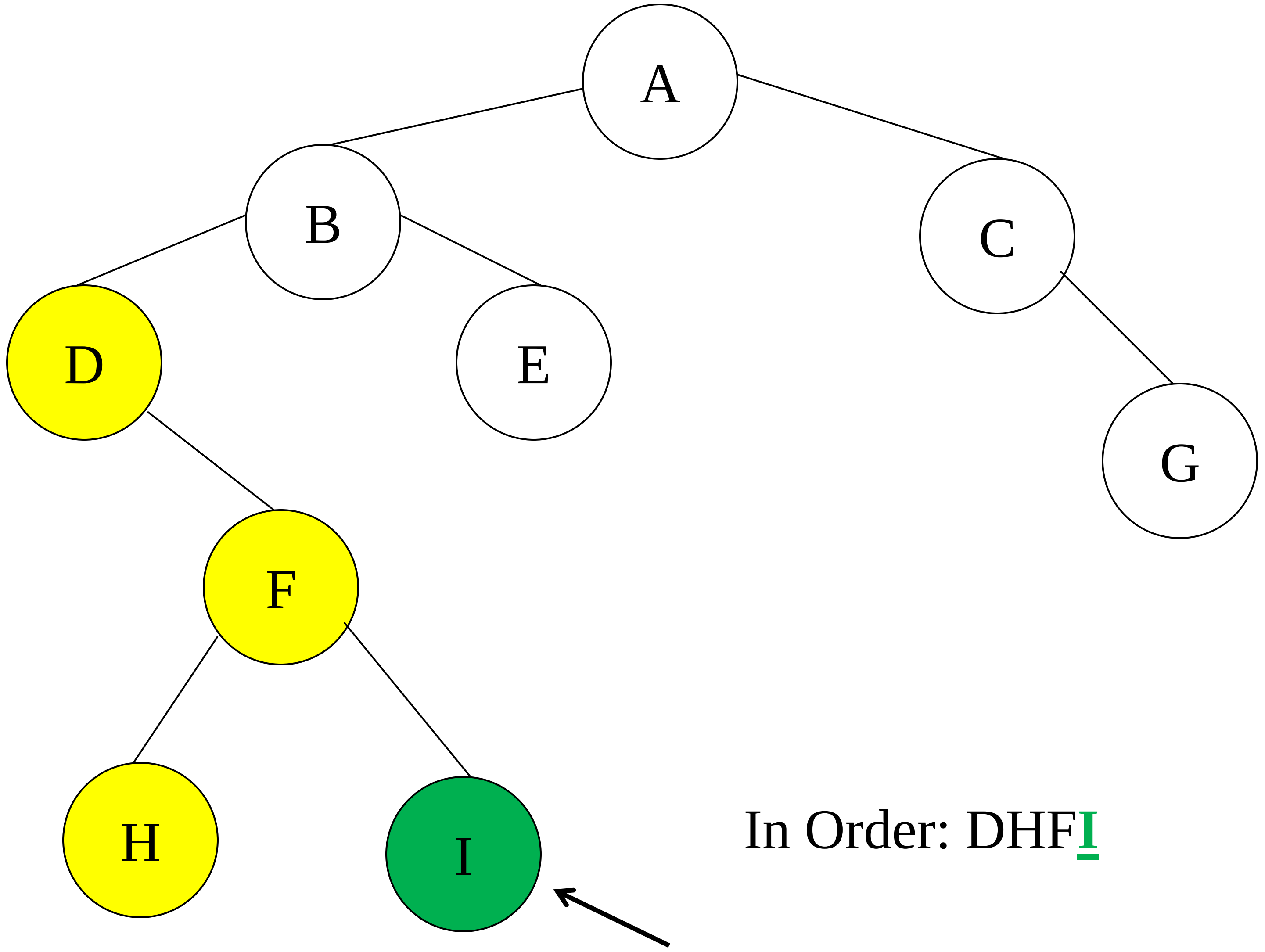
In Order: D**H**

Tree Traversals: Another Example



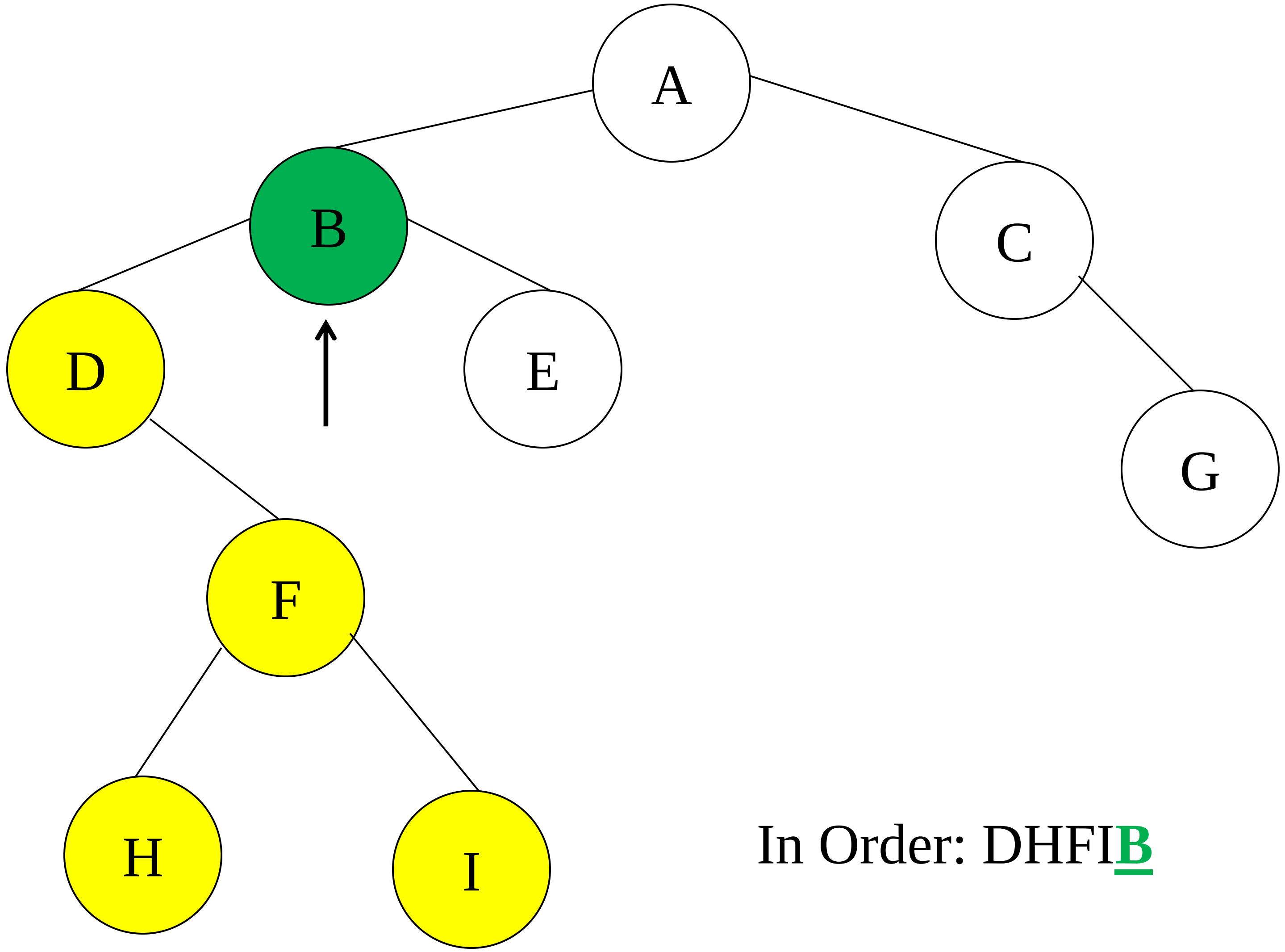
In Order: DH**F**

Tree Traversals: Another Example



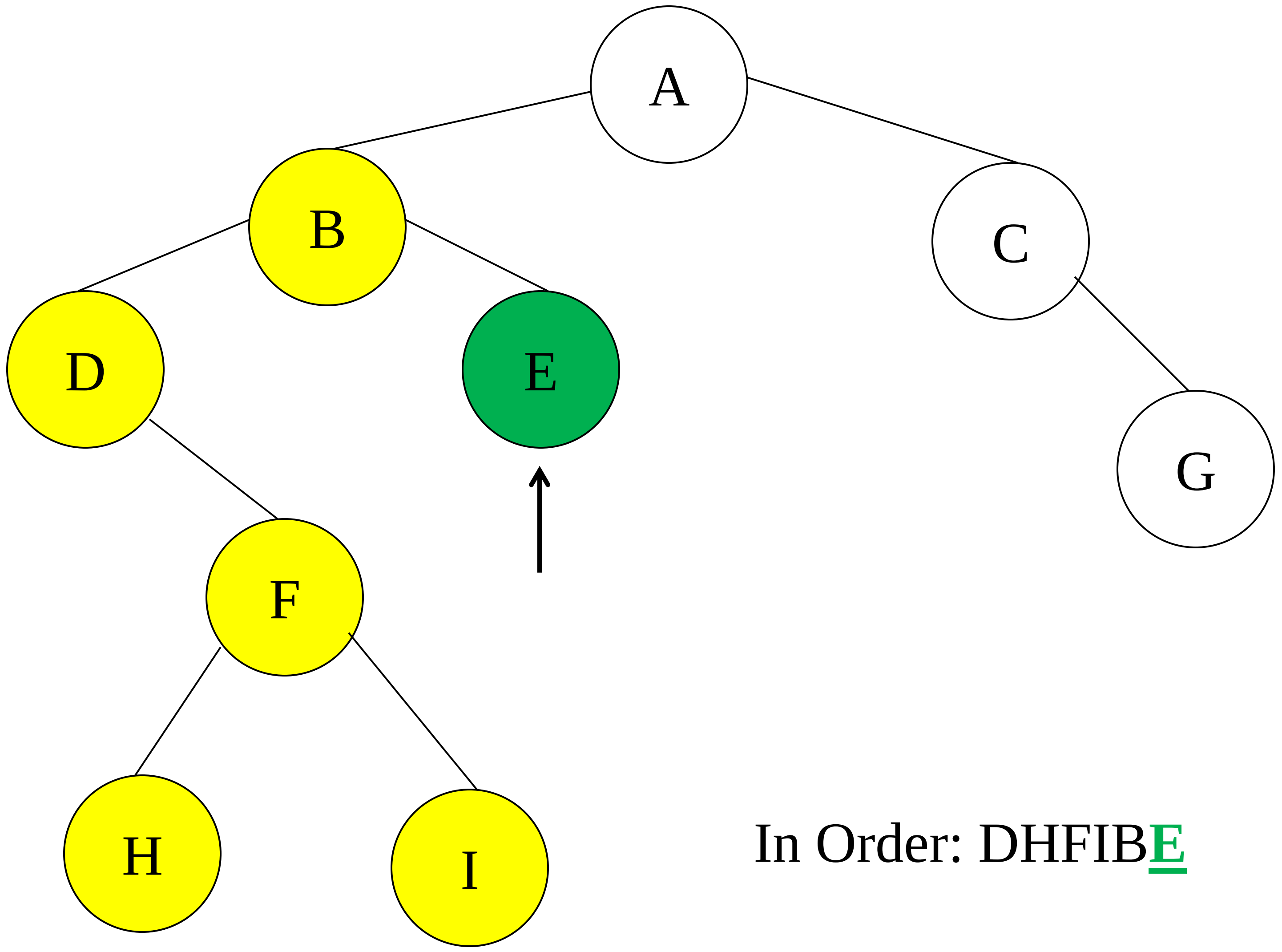
In Order: DHF**I**

Tree Traversals: Another Example



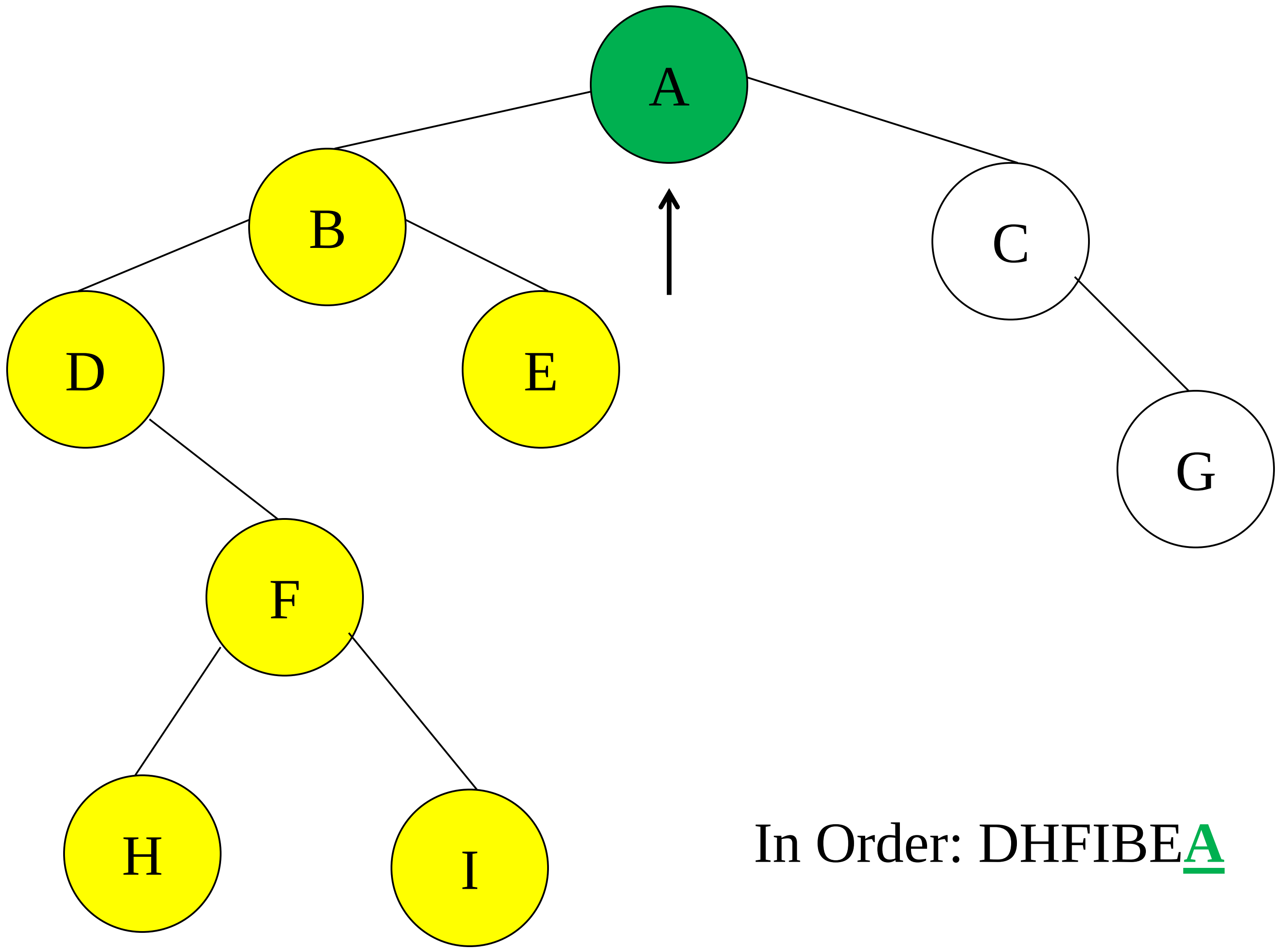
In Order: DHFI**B**

Tree Traversals: Another Example



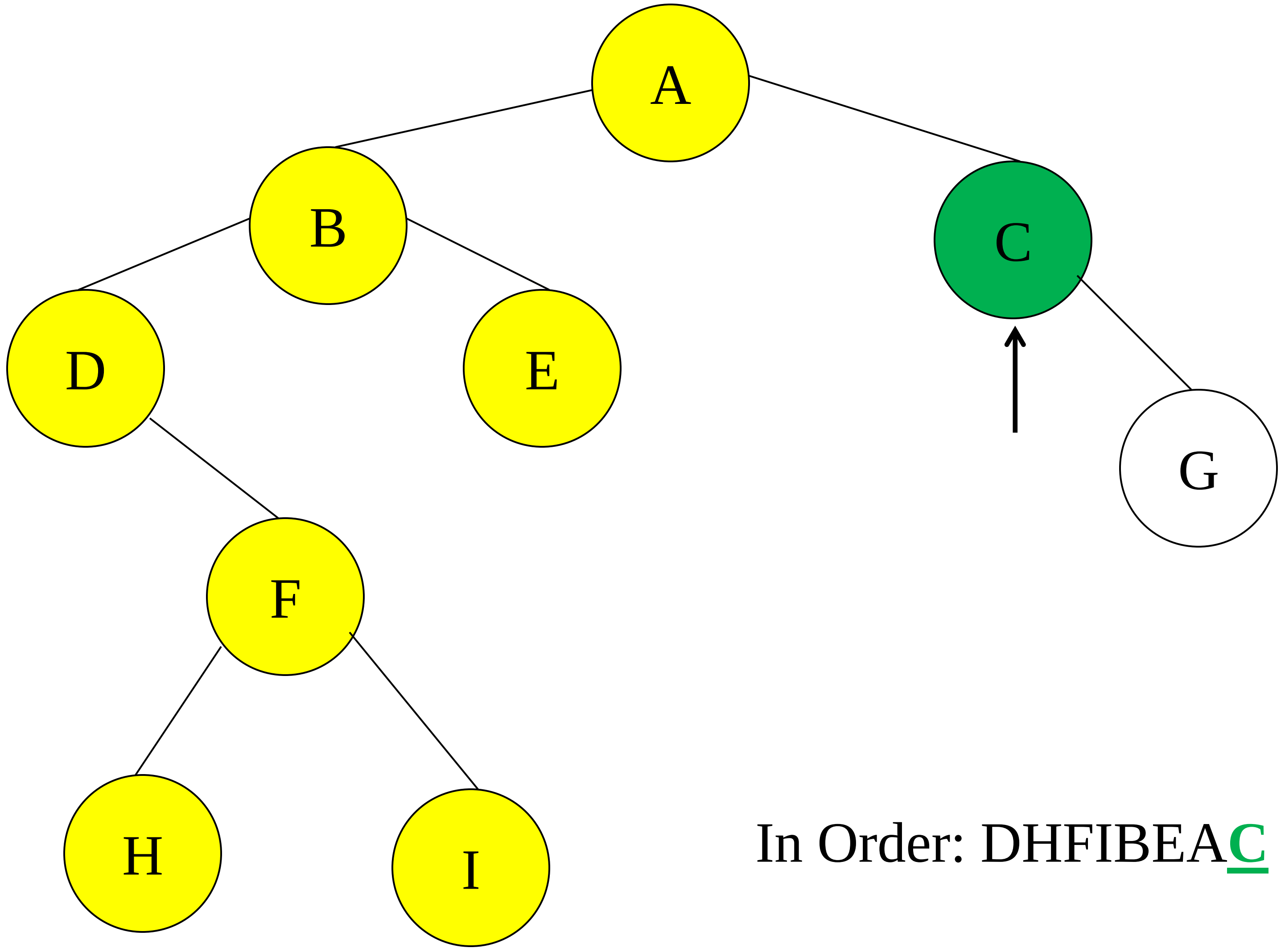
In Order: DHFIB**E**

Tree Traversals: Another Example



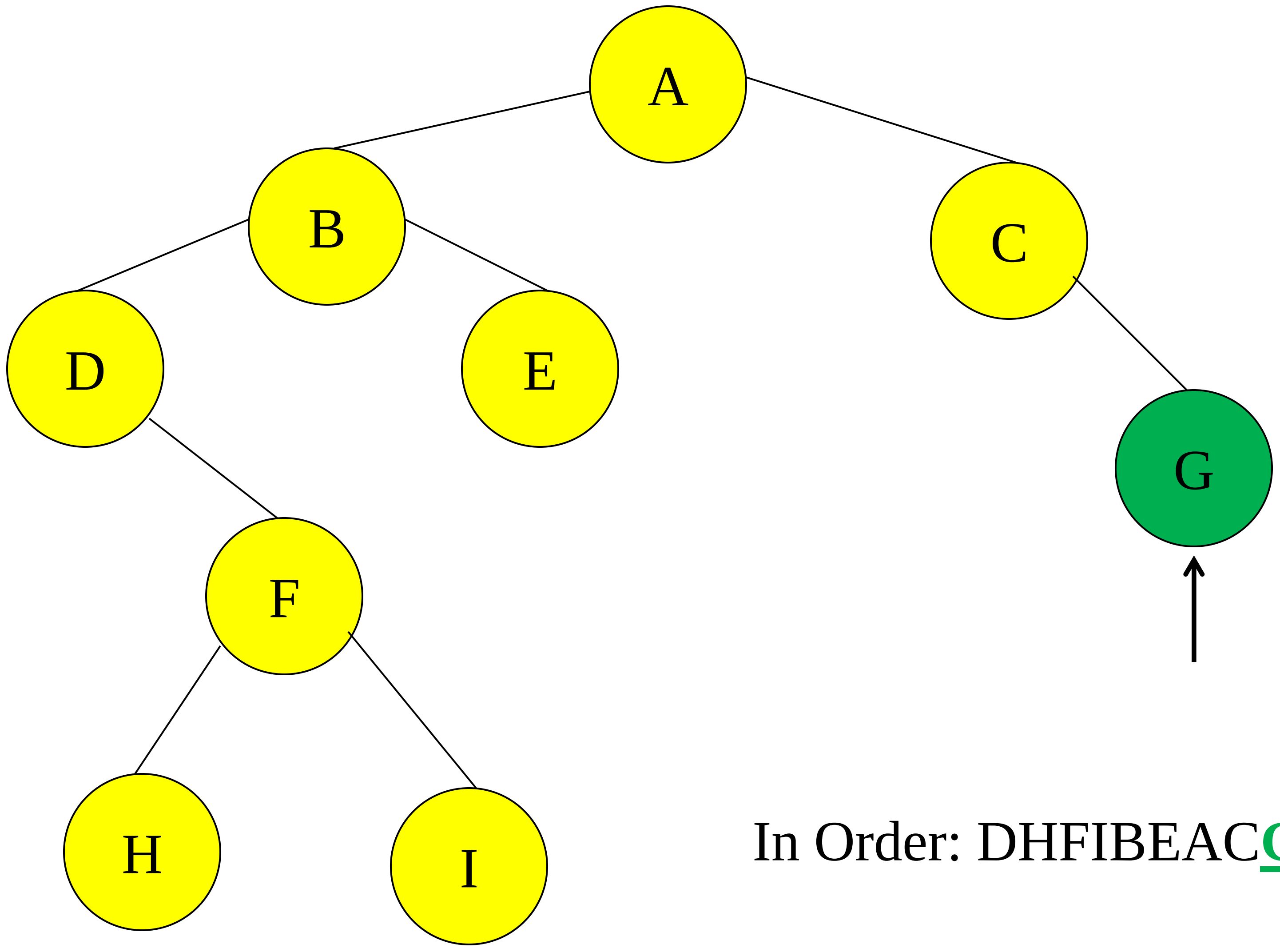
In Order: DHFIBE**A**

Tree Traversals: Another Example



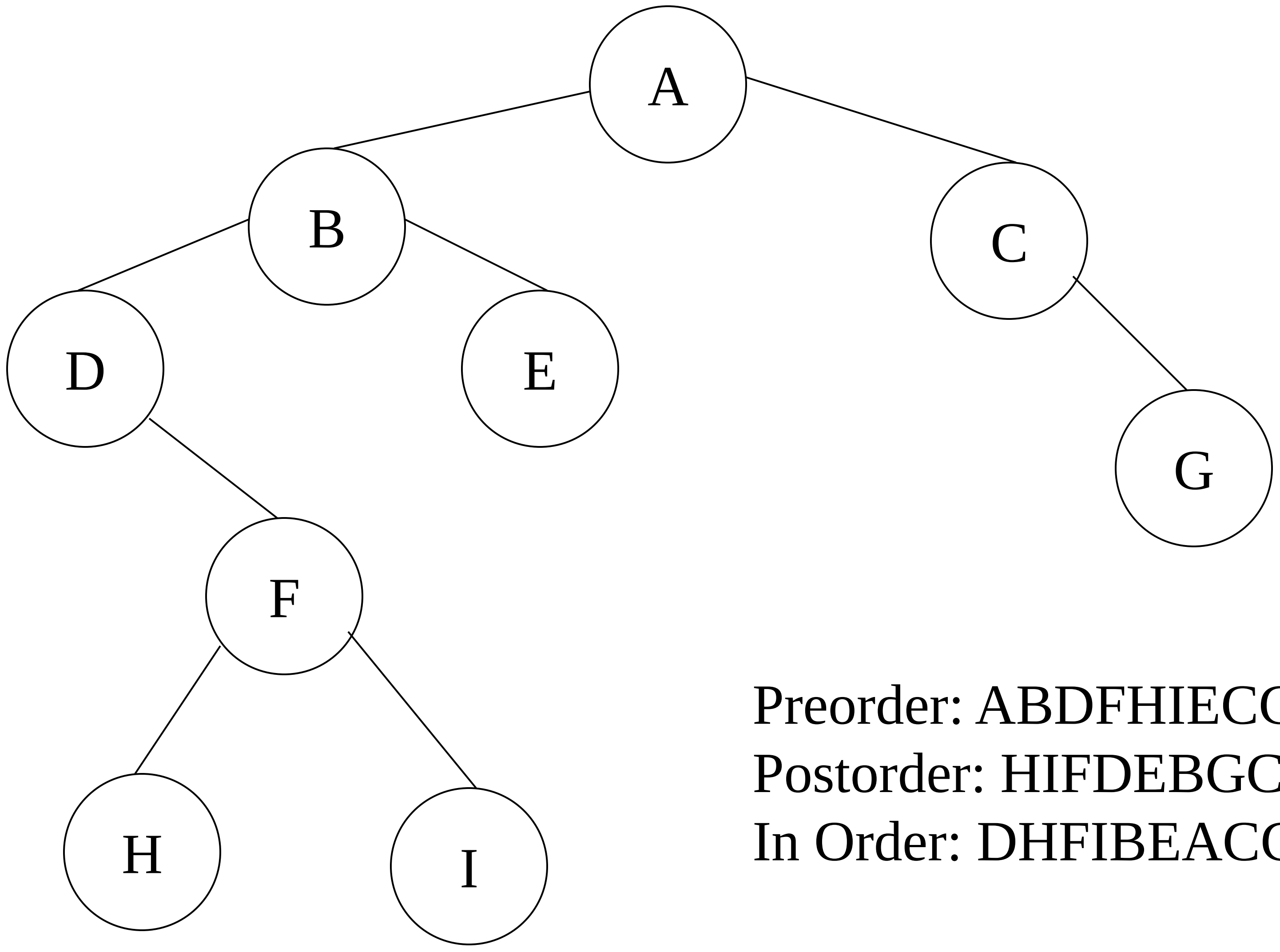
In Order: DHFIBEA**C**

Tree Traversals: Another Example



In Order: DHFIBEAC**G**

Tree Traversals: Another Example



Preorder: ABDFHIECG
Postorder: HIFDEBGCA
In Order: DHFIBEACG

Without recursion ?

HW

Without recursion ?

Inorder Tree Traversal without Recursion

1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
    a) Pop the top item from stack.
    b) Print the popped item, set current = popped_item->right
    c) Go to step 3.
5) If current is NULL and stack is empty then we are done.

```
# A binary tree node
class Node:
       # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


# Iterative function for inorder tree traversal
def inOrder(root):

    # Set current to root of binary tree
    current = root
    stack = [] # initialize stack

    while True:

        # Reach the left most Node of the current Node
        if current is not None:

            # Place pointer to a tree node on the stack
            # before traversing the node's left subtree
            stack.append(current)

            current = current.left

         # BackTrack from the empty subtree and visit the Node
        # at the top of the stack; however, if the stack is
        # empty you are done
        elif(stack):
            current = stack.pop()
            print(current.data, end=" ") # Python 3 printing

            # We have visited the node and its left
            # subtree. Now, it's right subtree's turn
            current = current.right

        else:
            break
    print()
```

Without recursion ?

Preorder Tree Traversal without Recursion

Following is a simple stack based iterative process to print Preorder traversal.
1.Create an empty stack nodeStack and push root node to stack.
2.Do the following while nodeStack is not empty.
   1. *Pop an item from the stack and print it.*
   2. *Push right child of a popped item to stack*
   3. *Push left child of a popped item to stack*

The right child is pushed before the left child to make sure that the left subtree is processed first.

```
# An iterative process to print preorder traversal of BT
def iterativePreorder(root):

    # Base CAse
    if root is None:
        return


    # create an empty stack and push root to it
    nodeStack = []
    nodeStack.append(root)


    # Pop all items one by one. Do following for every popped
item
    # a) print it
    # b) push its right child
    # c) push its left child
    # Note that right child is pushed first so that left
    # is processed first */
    while(len(nodeStack) > 0):

        # Pop the top item from stack and print it
        node = nodeStack.pop()
        print (node.data, end=" ")

        # Push right and left children of the popped node
        # to stack
        if node.right is not None:
            nodeStack.append(node.right)
        if node.left is not None:
            nodeStack.append(node.left)
```

Without recursion ?

Postorder Tree Traversal without Recursion

```
1.1 Create an empty stack
2.1 Do following while root is not NULL
    a) Push root's right child and then root to stack.
    b) Set root as root's left child.
2.2 Pop an item from stack and set it as root.
    a) If the popped item has a right child and the right child
       is at top of stack, then remove the right child from stack,
       push the root back and set root as root's right child.
    b) Else print root's data and set root as NULL.
2.3 Repeat steps 2.1 and 2.2 while stack is not empty.
```

```python
def peek(stack):
    if len(stack) > 0:
        return stack[-1]
    return None
# A iterative function to do postorder traversal of
# a given binary tree
def postOrderIterative(root):
      # Check for empty tree
    if root is None:
        return

     stack = []
     while(True):
         while (root):
            # Push root's right child and then root to stack
            if root.right is not None:
                stack.append(root.right)
            stack.append(root)

             # Set root as root's left child
            root = root.left
          # Pop an item from stack and set it as root
        root = stack.pop()

          # If the popped item has a right child and the
        # right child is not processed yet, then make sure
        # right child is processed before root
        if (root.right is not None and
            peek(stack) == root.right):
            stack.pop() # Remove right child from stack
            stack.append(root) # Push root back to stack
            root = root.right # change root so that the
                            # right childis processed next

          # Else print root's data and set root as None
        else:
            ans.append(root.data)
            root = None


        if (len(stack) <= 0):
                break
```

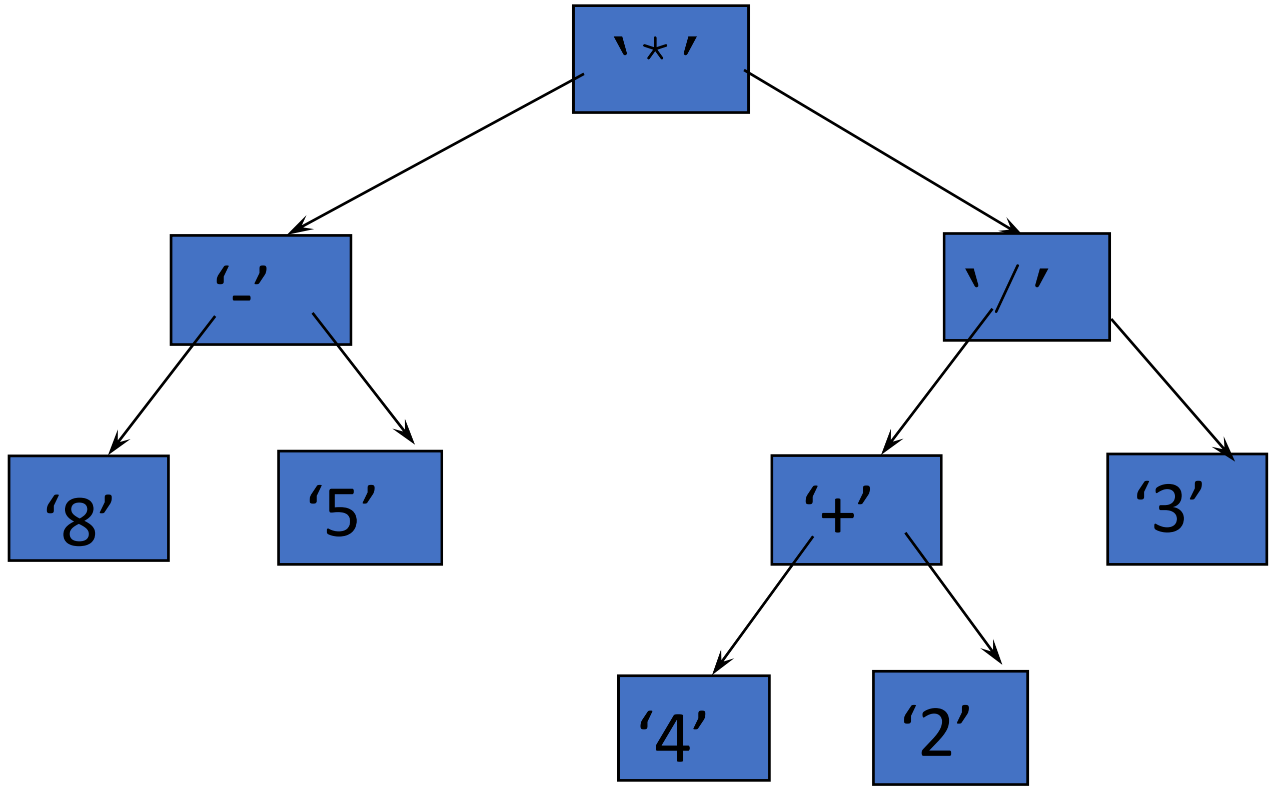
Expression Tress

A Binary Expression Tree is . . .

A special kind of binary tree in which:

1. **Each leaf node contains a single operand**
2. **Each nonleaf node contains a single binary operator**
3. **The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.**

A Four-Level Binary Expression

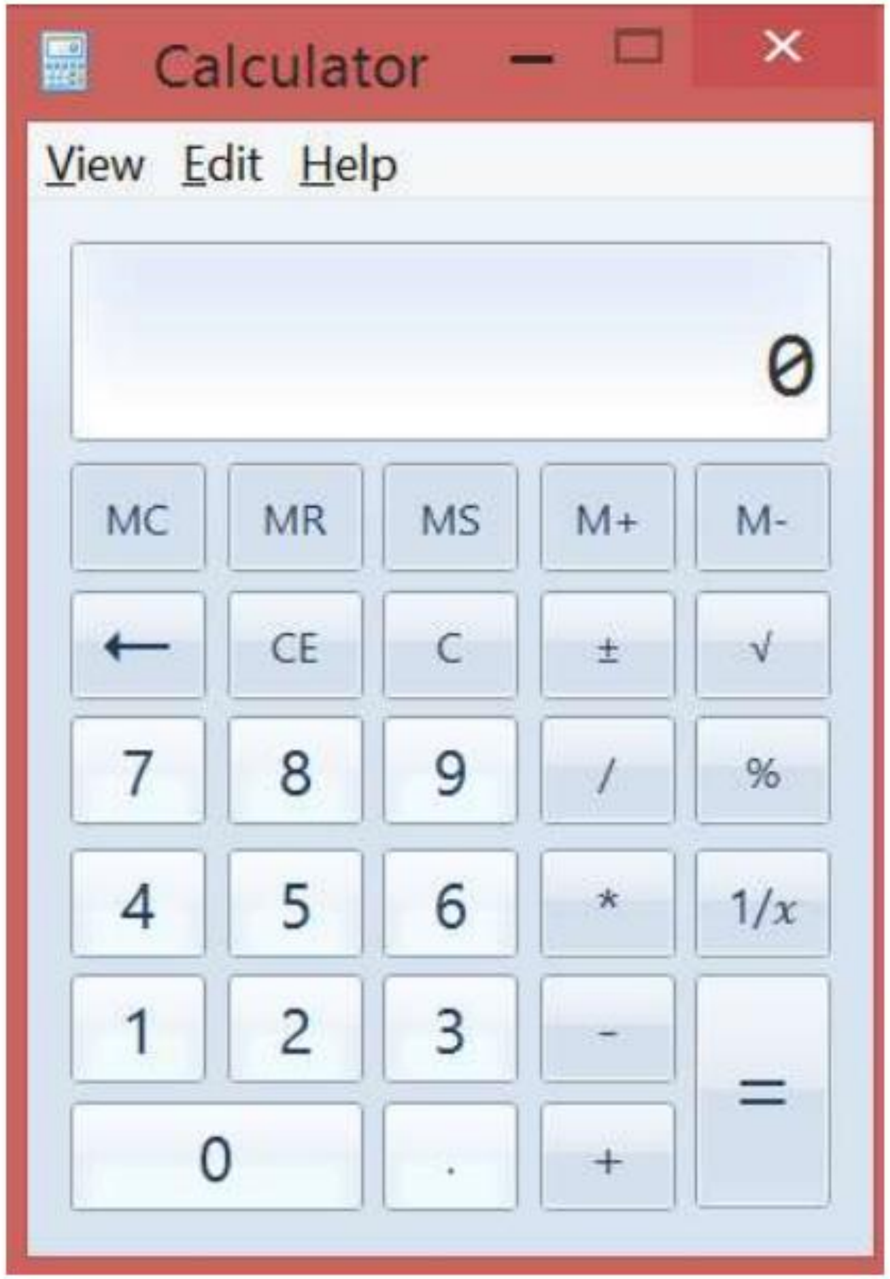


Levels Indicate Precedence

The levels of the nodes in the tree indicate their relative precedence of evaluation (we do not need parentheses to indicate precedence).

Operations at higher levels of the tree are evaluated later than those below them.

The operation at the root is always the last operation performed.



My Windows calculator says
3 + 4 * 2 = 14.

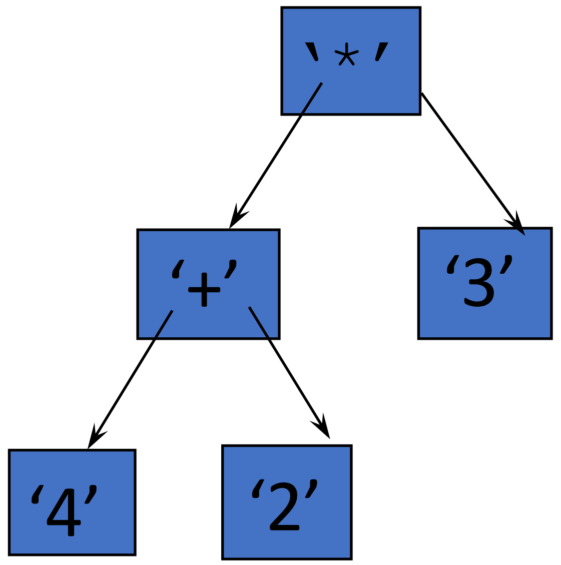
Old calculator

(3 + 4) * 2 = 14.

Whereas….
if I google “3+4*2”, I get 11.

3 + (4*2) = 11.

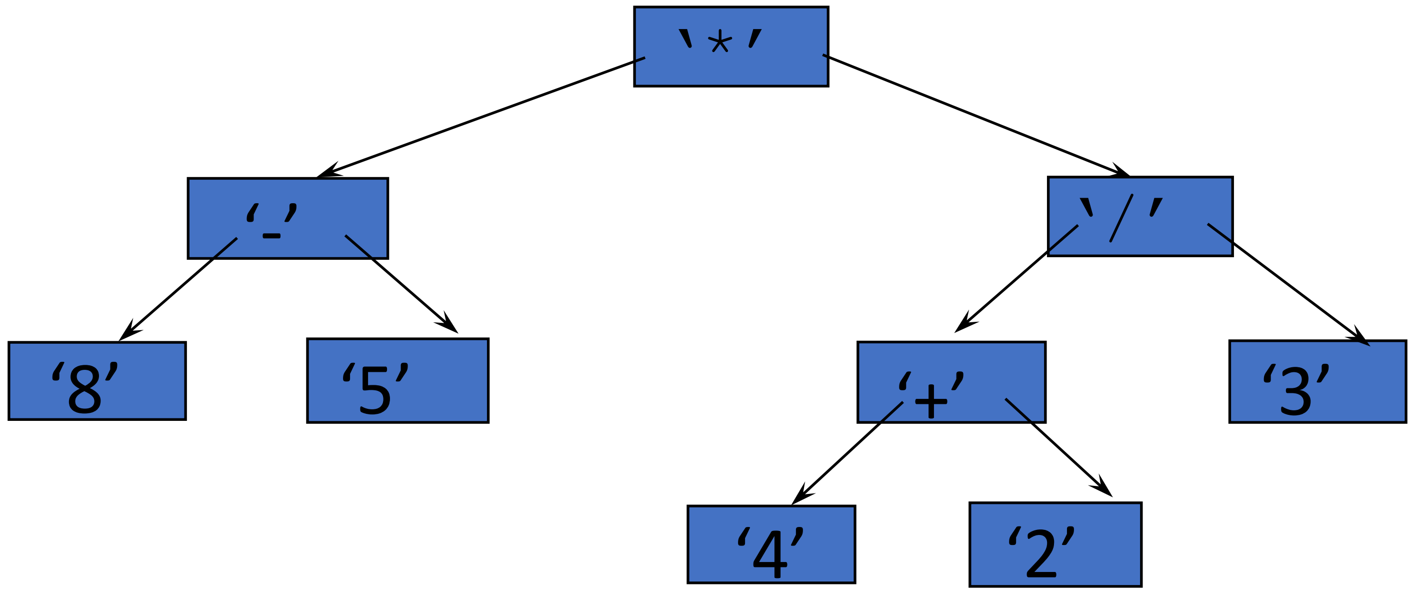
A Binary Expression Tree



What value does it have?

(4 + 2) * 3 = 18

Easy to generate the infix, prefix, postfix expressions (how?)

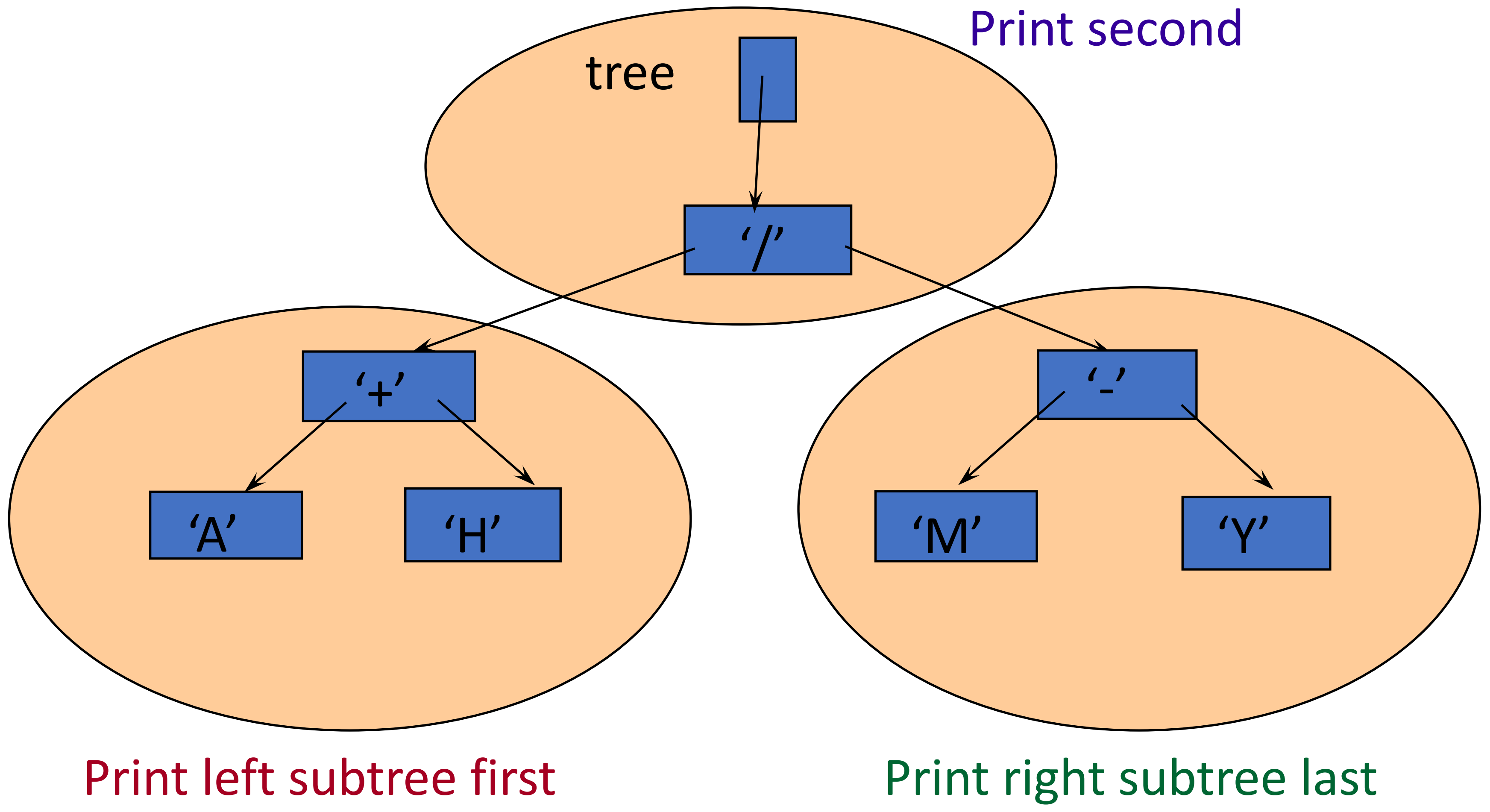


Infix: ((8 - 5) * ((4 + 2) / 3))

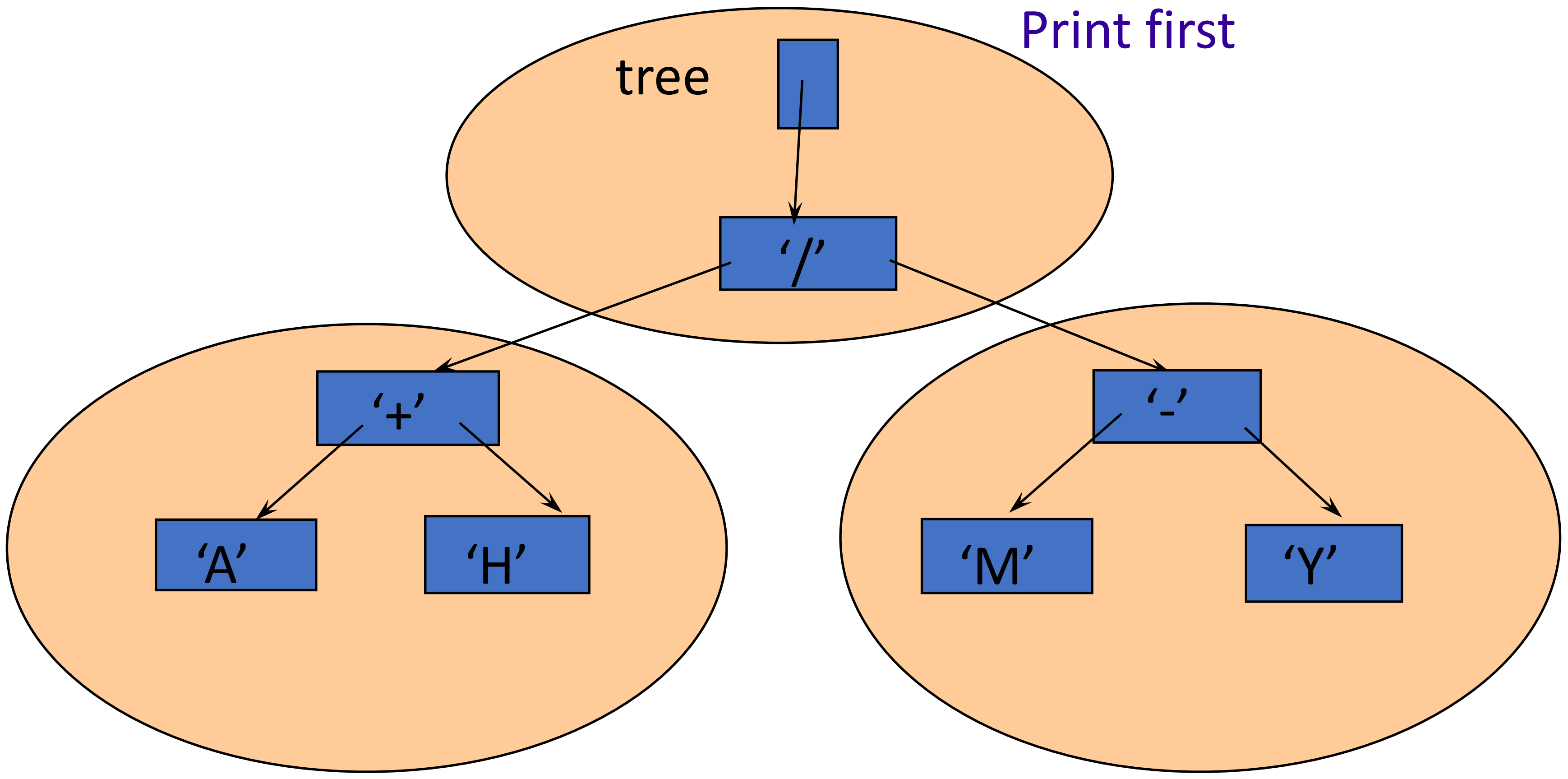
Prefix: * - 8 5 / + 4 2 3

Postfix: 8 5 - 4 2 + 3 / *

Inorder Traversal: (A + H) / (M - Y)



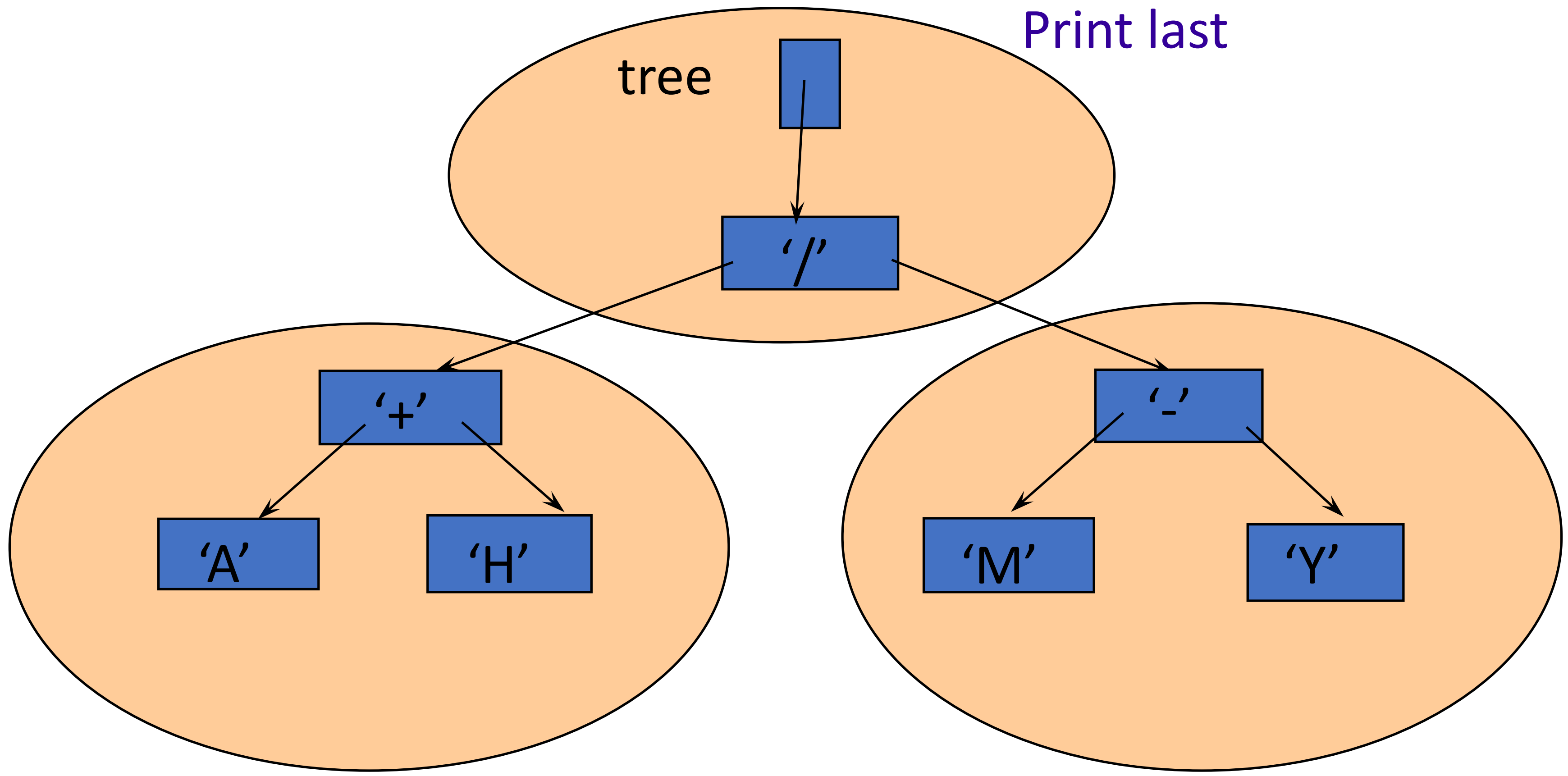
Preorder Traversal: / + A H - M Y



Print left subtree second

Print right subtree last

Postorder Traversal: A H + M Y - /



Print left subtree first

Print right subtree second

Construction of Expression Tree

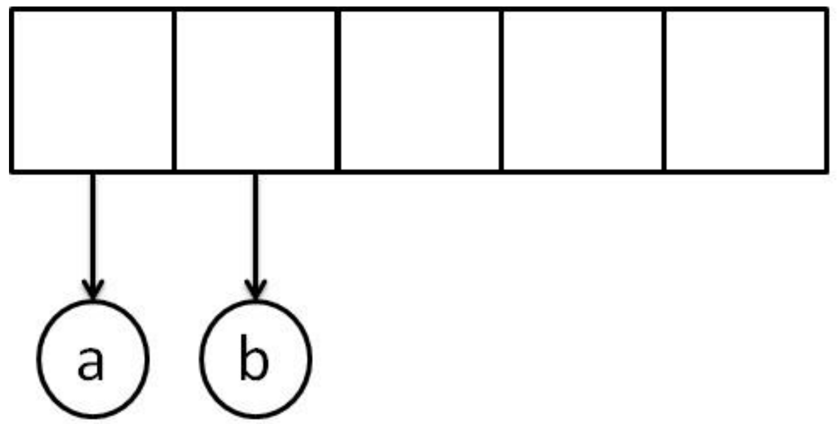
Let us consider a **postfix expression** is given as an input for constructing an expression tree. Following are the step to construct an expression tree:

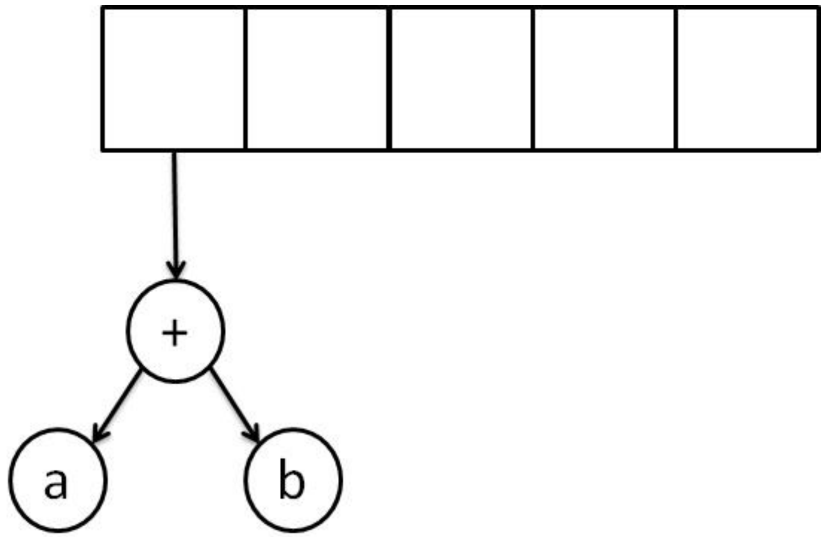
1) **Read one symbol at a time from the postfix expression**.
2) Check if the symbol is an **operand or operator**.
    1) **If the symbol is an operand, create a one node tree and push a pointer onto a stack**
    2) **If the symbol is an operator, pop two pointers from the stack namely T_1 & T_2 and form a new tree with root as the operator, T_1 & T_2 as a left and right child**
3) A pointer to this new tree is pushed onto the stack

- Thus, An expression is created or constructed by reading the symbols or numbers from the left. If operand, create a node. If operator, create a tree with operator as root and two pointers to left and right subtree

Example - Postfix Expression Construction

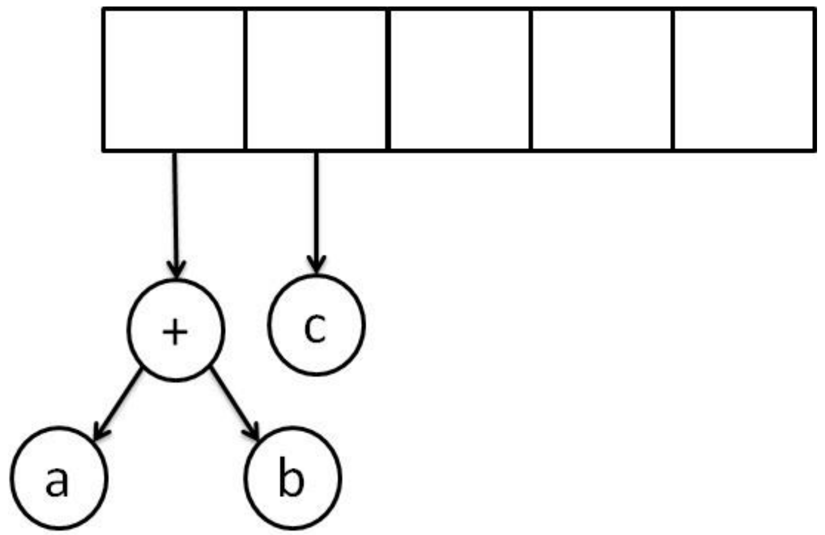
- a b + c *
- The first two symbols are operands, we create one-node tree and push a pointer to them onto the stack.
- Next, read a'+' symbol, so two pointers to tree are popped,a new tree is formed and push a pointer to it onto the stack.

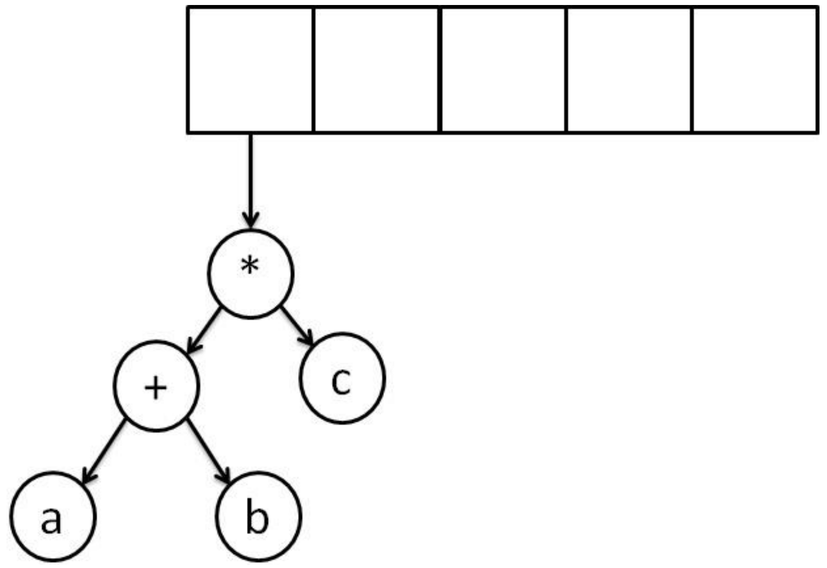




Example - Postfix Expression Construction/ evaluate a postfix expression

- Next, 'c' is read, we create one node tree and push a pointer to it onto the stack.
- Finally, the last symbol is read '*', we pop two tree pointers and form a new tree with a, '*' as root, and a pointer to the final tree remains on the stack.

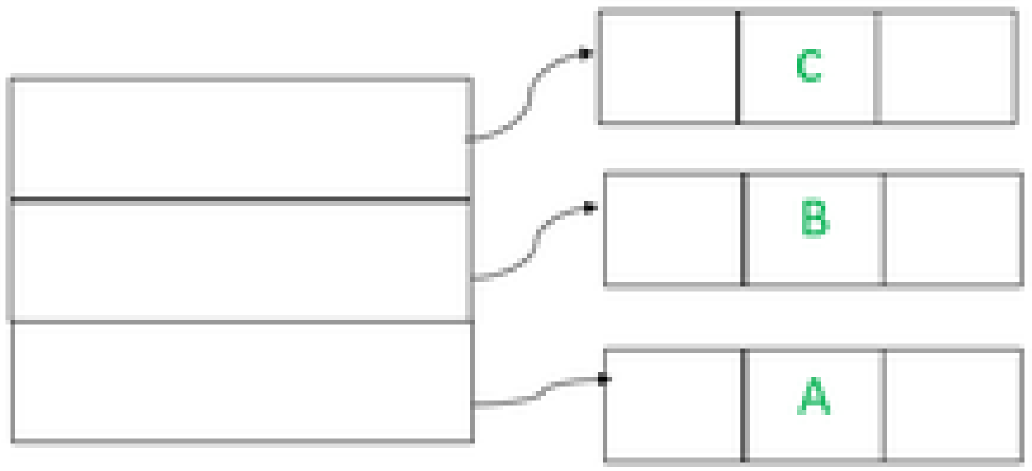




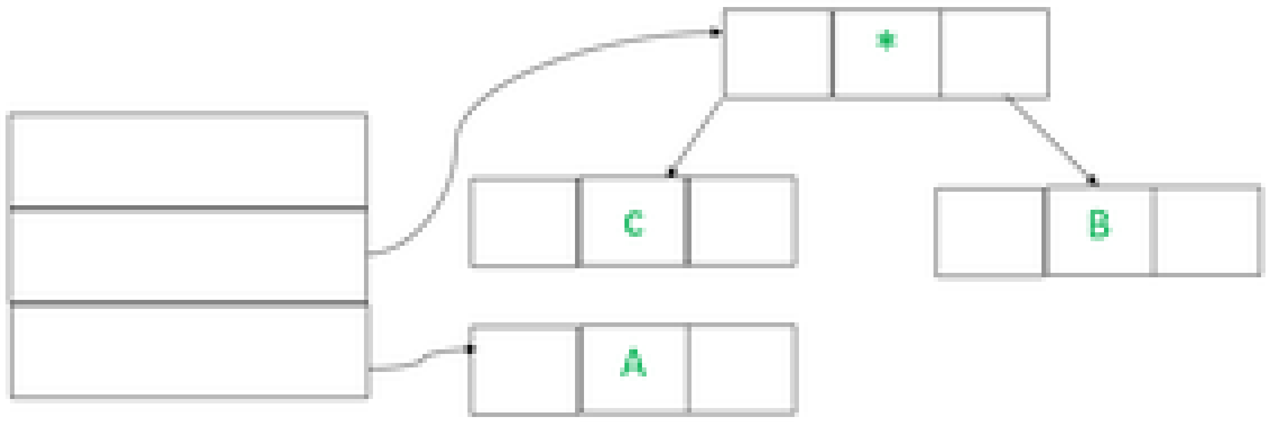
Examples:

- Input : A + B * C / D
- **Postfix : A B C*+ D/**

The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.



In the Next step, an operator '*' will going read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack



In the Next step, an operator '+' will read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.

Threaded Binary Trees

Threaded Binary Trees

- Two many null pointers in current representation of binary trees
  - n: number of nodes
  - number of non-null links: n-1
  - total links: 2n
  - null links: 2n-(n-1)=n+1
- Replace these null pointers with some useful “threads”.

Threaded Binary Trees (Continued)

A Threaded Binary Tree is one of the variants of a normal Binary Tree that assists faster tree traversal and doesn't need a Stack or Recursion. It reduces memory wastage by placing the null pointers of a leaf node to the in-order successor or in-order predecessor.

It decreases the memory wastage by setting the null pointers of a leaf node to the in-order predecessor or in-order successor.

Threaded Binary Trees (Continued)

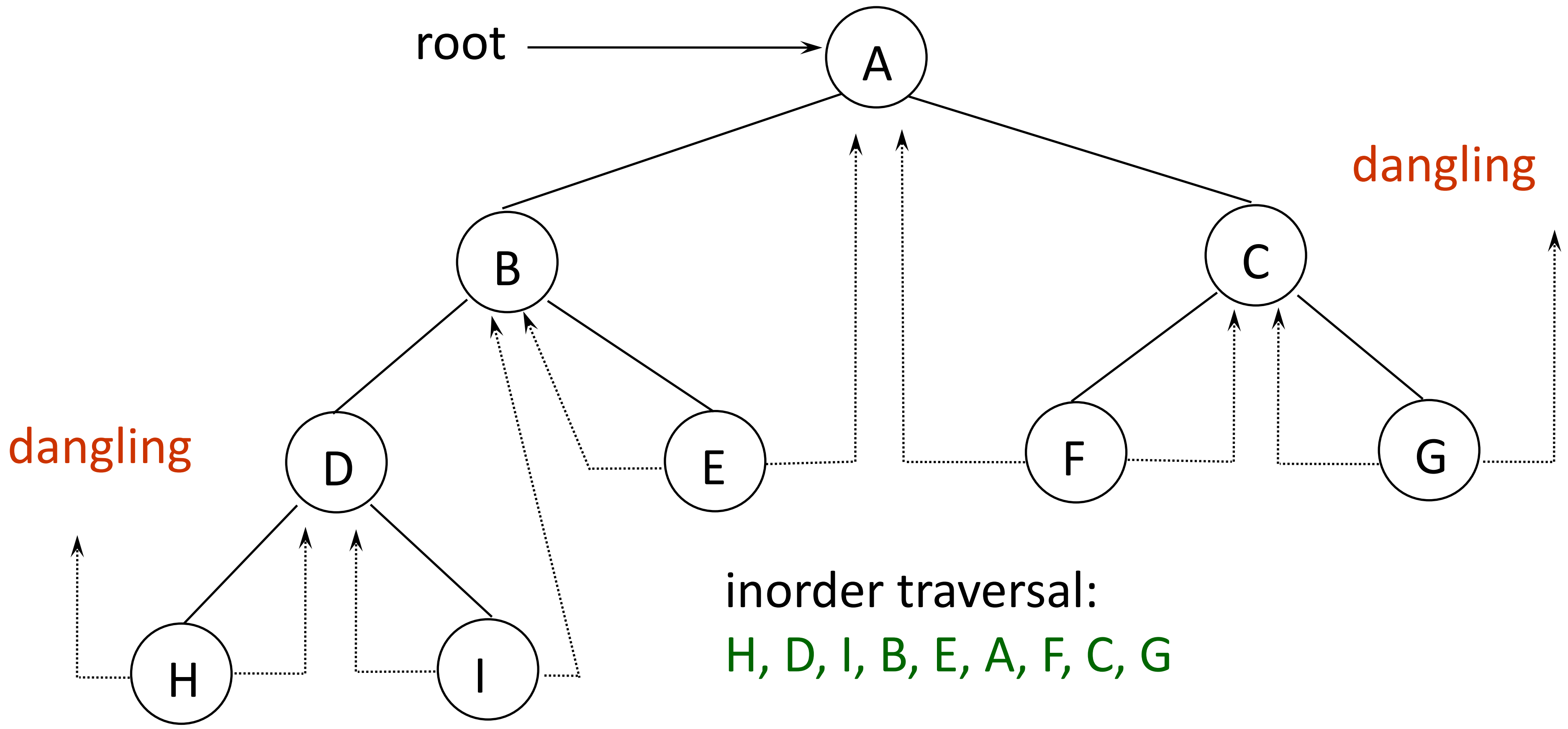
If `ptr->left_child` is null,
replace it with a pointer to the node that would be visited ***before*** **`ptr`** **in an *inorder traversal***

inorder predecessor

If `ptr->right_child` is null,
replace it with a pointer to the node that would be visited ***after*** **`ptr`** **in an *inorder traversal***

inorder successor

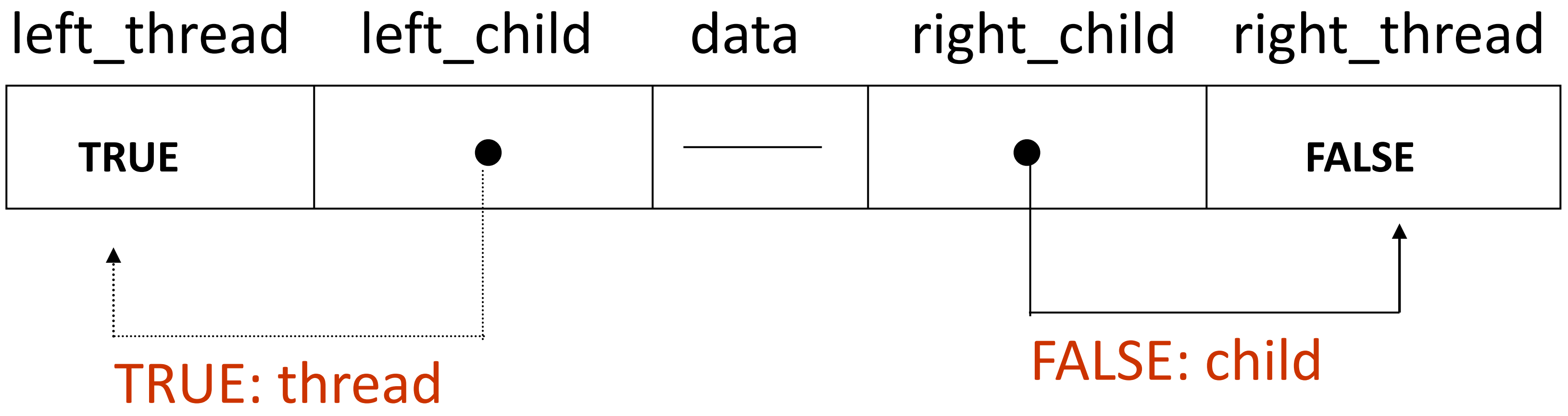
A Threaded Binary Tree



inorder traversal:

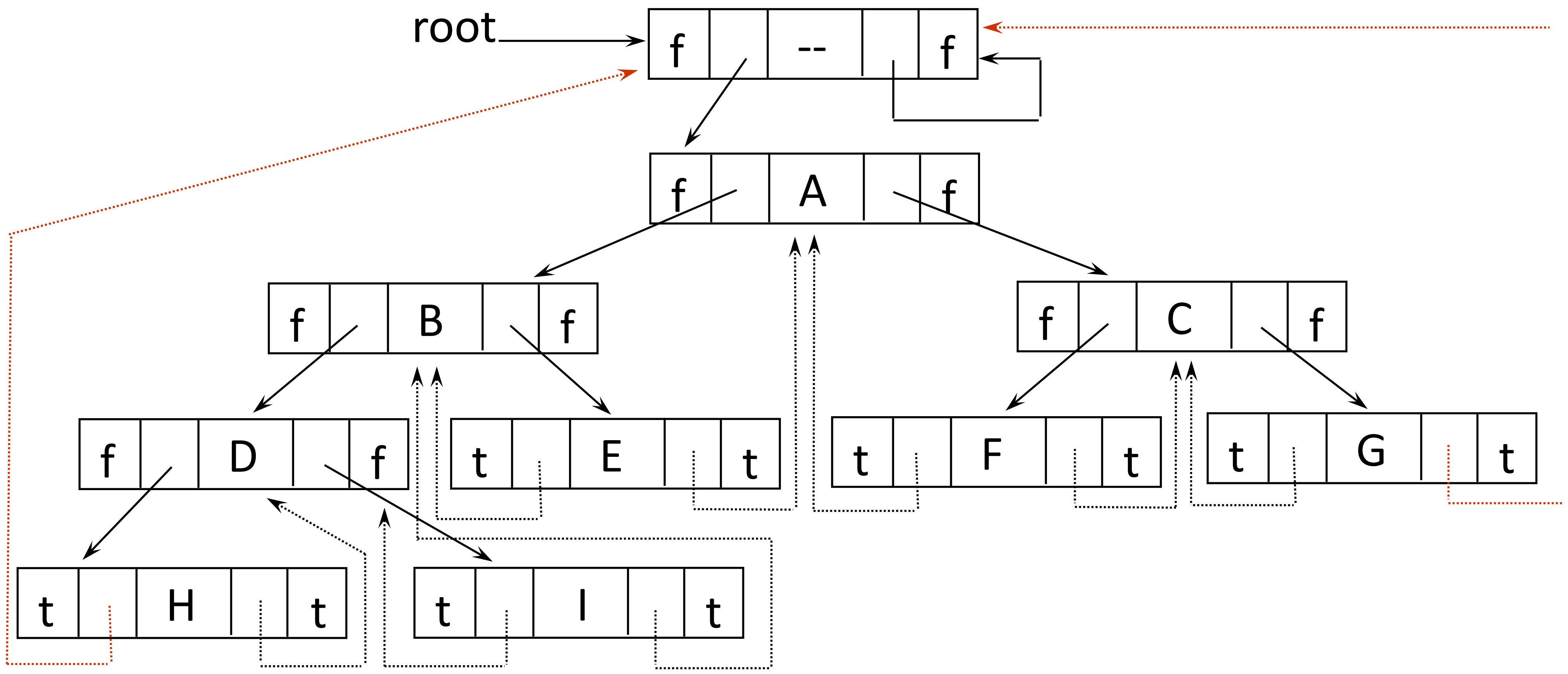
H, D, I, B, E, A, F, C, G

Data Structures for Threaded BT



```
typedef struct threaded_tree *threaded_pointer;
typedef struct threaded_tree {
    short int left_thread;
    threaded_pointer left_child;
    char data;
    threaded_pointer right_child;
    short int right_thread;  };
```

Memory Representation of A Threaded BT



Threaded Binary Trees

Advantages of Threaded Binary Tree

•In this Tree it enables linear traversal of elements.
•It eliminates the use of stack as it perform linear traversal, so save memory.
•Enables to find parent node without explicit use of parent pointer
•Threaded tree give forward and backward traversal of nodes by in-order fashion
•Nodes contain pointers to in-order predecessor and successor
•For a given node, we can easily find inorder predecessor and successor. So, searching is much more easier.
• In threaded binary tree there is no NULL pointer present. Hence memory wastage in occupying NULL links is avoided.
•The threads are pointing to successor and predecessor nodes. This makes us to obtain predecessor and successor node of any node quickly.
•There is no need of stack while traversing the tree, because using thread links we can reach to previously visited nodes.

Threaded Binary Trees

Disadvantages of Threaded Binary Tree

•Every node in threaded binary tree need extra information(extra memory) to indicate whether its left or right node indicated its child nodes or its inorder predecessor or successor.

•Insertion and deletion are way more complex and time consuming than the normal one since both threads and ordinary links need to be maintained.

•Implementing threads for every possible node is complicated.

•Increased complexity: Implementing a threaded binary tree requires more complex algorithms and data structures than a regular binary tree. This can make the code harder to read and debug.

•Extra memory usage: In some cases, the additional pointers used to thread the tree can use up more memory than a regular binary tree. This is especially true if the tree is not fully balanced, as threading a skewed tree can result in a large number of additional pointers.

•Limited flexibility: Threaded binary trees are specialized data structures that are optimized for specific types of traversal.

•Difficulty in parallelizing: It can be challenging to parallelize operations on a threaded binary tree, as the threading can introduce data dependencies that make it difficult to process nodes independently. This can limit the performance gains that can be achieved through parallelism.

Threaded Binary Trees

Applications of threaded binary tree –

•Expression evaluation: Threaded binary trees can be used to evaluate arithmetic expressions in a way that avoids recursion or a stack. The tree can be constructed from the input expression, and then traversed in-order or pre-order to perform the evaluation.

•Database indexing: In a database, threaded binary trees can be used to index data based on a specific field (e.g. last name). The tree can be constructed with the indexed values as keys, and then traversed in-order to retrieve the data in sorted order.

•Disk-based data structures: Threaded binary trees can be used in disk-based data structures (e.g. B-trees) to improve performance. By threading the tree, it can be traversed in a way that minimizes disk seeks and improves locality of reference.

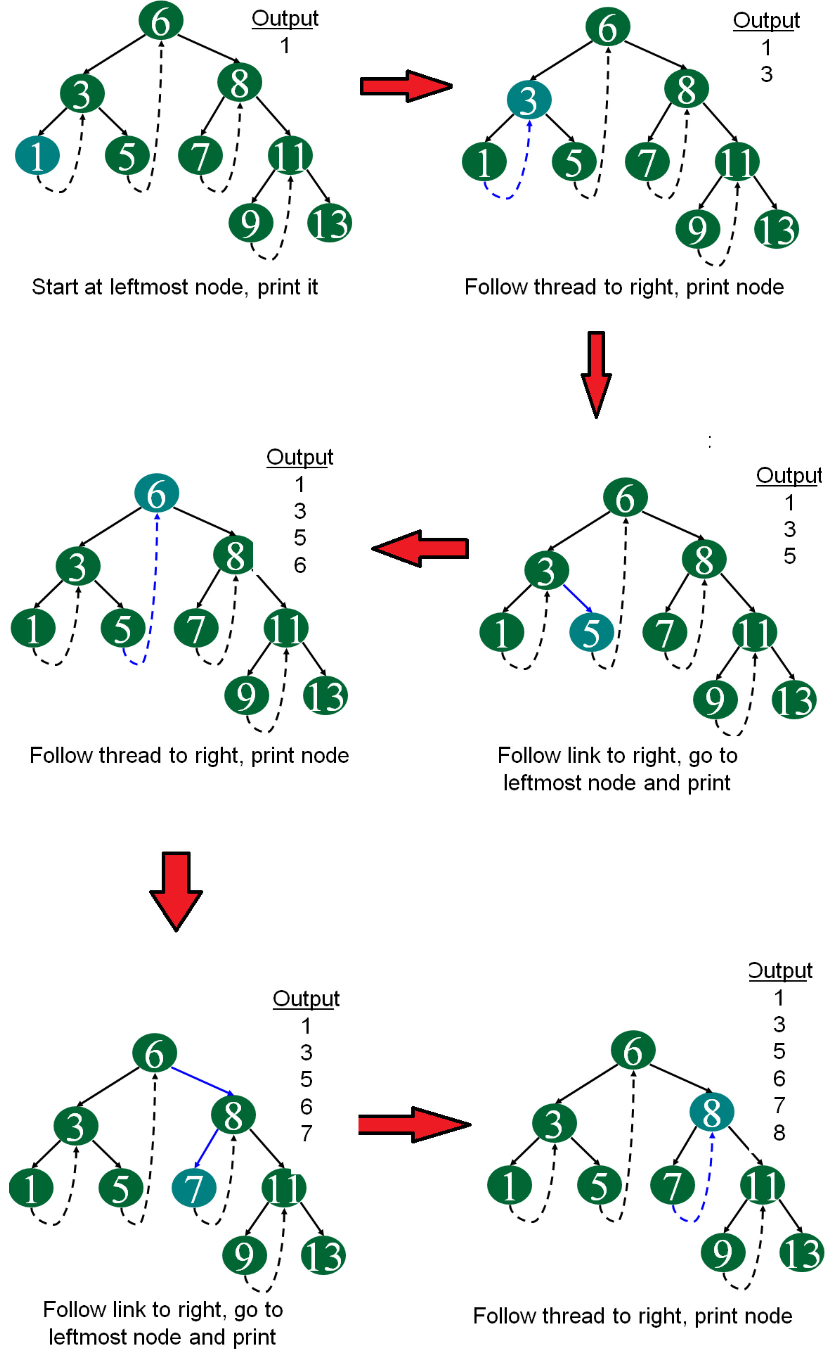
•Navigation of hierarchical data: In certain applications, threaded binary trees can be used to navigate hierarchical data structures, such as file systems or web site directories. The tree can be constructed from the hierarchical data, and then traversed in-order or pre-order to efficiently access the data in a specific order.

Inorder Traversal using Threads

```c
// Utility function to find leftmost node in a tree rooted
// with n
struct Node* leftMost(struct Node* n)
{
    if (n == NULL)
        return NULL;

    while (n->left != NULL)
        n = n->left;

    return n;
}

// C code to do inorder traversal in a threaded binary tree
void inOrder(struct Node* root)
{
    struct Node* cur = leftMost(root);
    while (cur != NULL) {
        printf("%d ", cur->data);

        // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
            cur = cur->right;
        else // Else go to the leftmost child in right
             // subtree
            cur = leftmost(cur->right);
    }
}
```

```python
# Utility function to find leftmost Node in a tree rooted
# with n
def leftMost(n):

    if (n == None):
        return None;

    while (n.left != None):
        n = n.left;

    return n;


# C code to do inorder traversal in a threaded binary tree
def inOrder(root):

    cur = leftMost(root);
    while (cur != None):
        print(cur.data," ");

        # If this Node is a thread Node, then go to
        # inorder successor
        if (cur.rightThread):
            cur = cur.right;
        else: # Else go to the leftmost child in right
             # subtree
            cur = leftmost(cur.right);
```

Following diagram demonstrates inorder order traversal using threads.


Output
1
Start at leftmost node, print it
Output
1
3
Follow thread to right, print node
Output
1
3
5
Follow link to right, go to leftmost node and print
Output
1
3
5
6
Follow thread to right, print node
Output
1
3
5
6
7
Follow link to right, go to leftmost node and print
Output
1
3
5
6
7
8
Follow thread to right, print node

continue same way for remaining node.....

Any questions ?