Binary trees

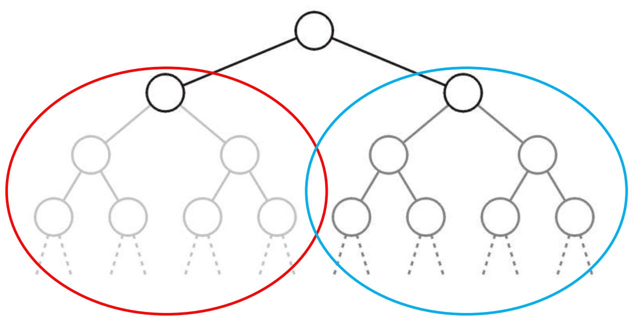
Definition

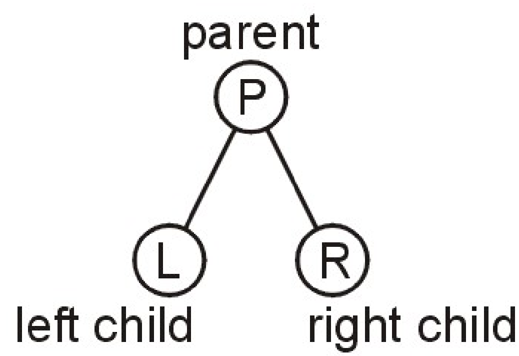
This is not a binary tree:

Definition

A binary tree has a restriction where each node has exactly two children:

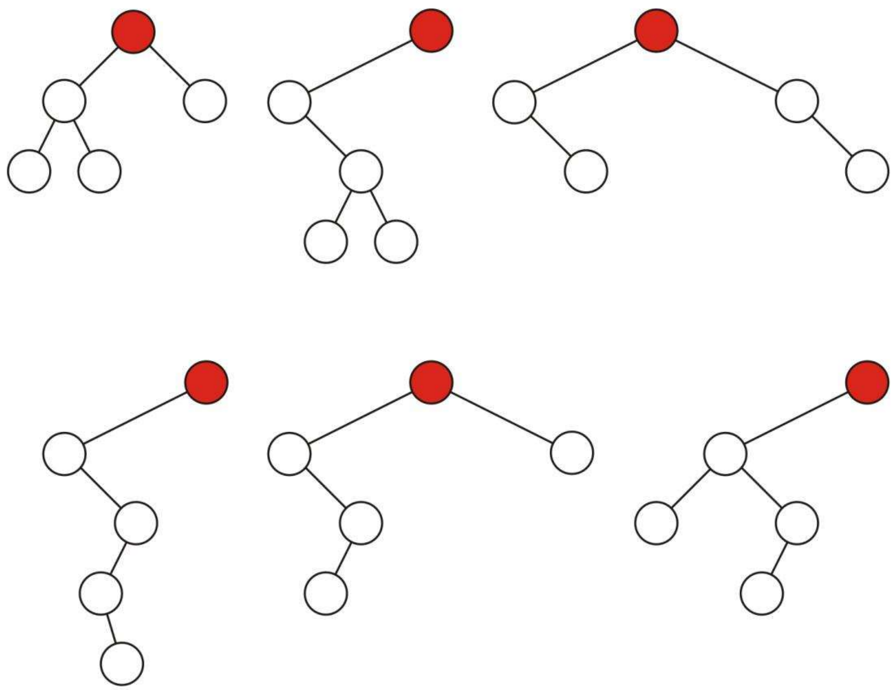
- Each child is either empty or another binary tree
- This restriction allows us to label the children as *left* and *right* subtree





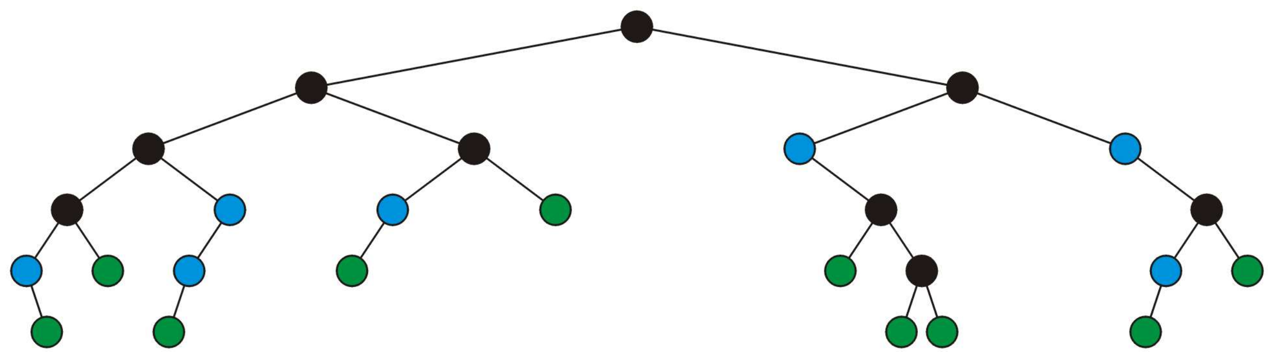
Definition

Sample variations on binary trees with five nodes:



Definition

A ***full*** node is a node where both the left and right sub-trees are non-empty trees

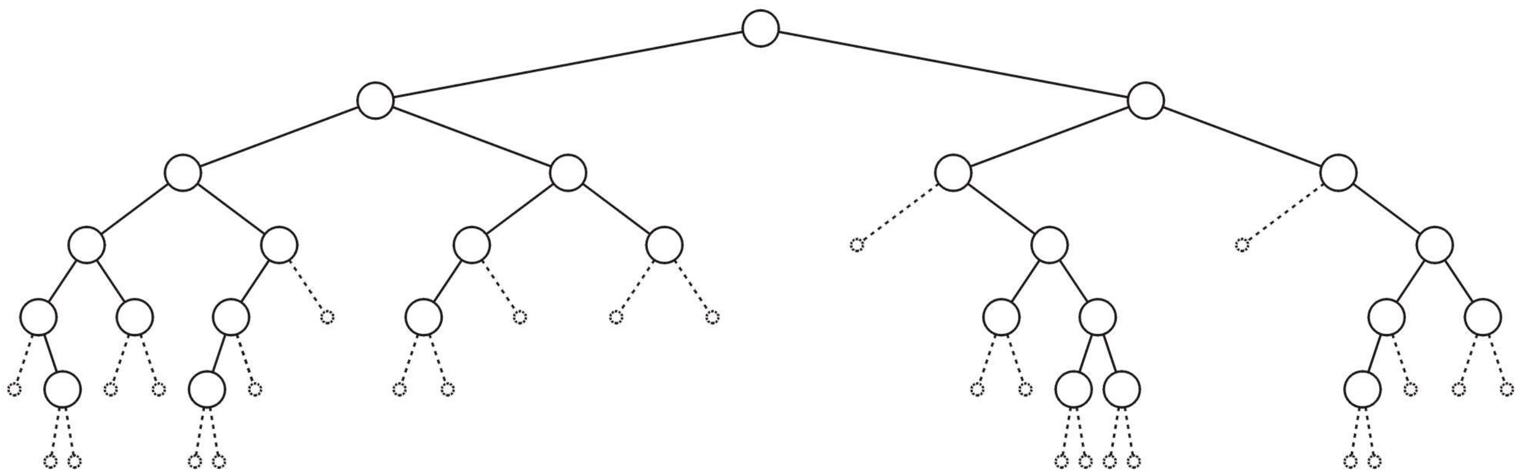


Legend:

full nodes neither leaf nodes

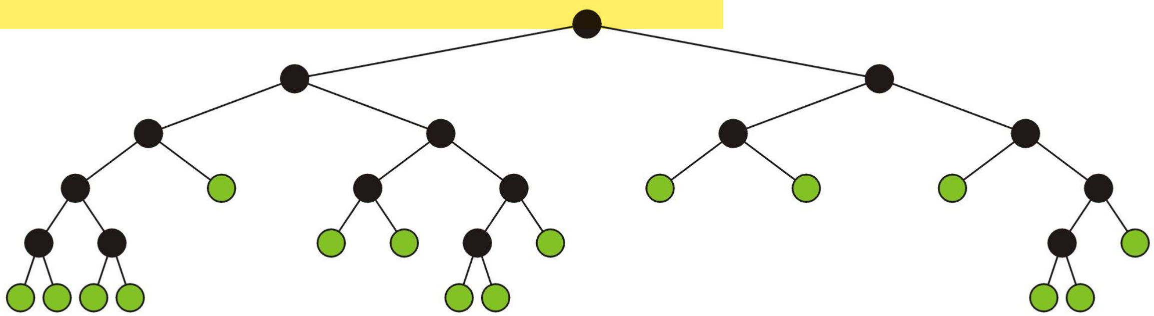
Definition

An ***empty** node or a null sub-tree* is any location where a new leaf node could be appended



Definition

A *full binary tree* is where each node is:
- A full node, or
- A leaf node

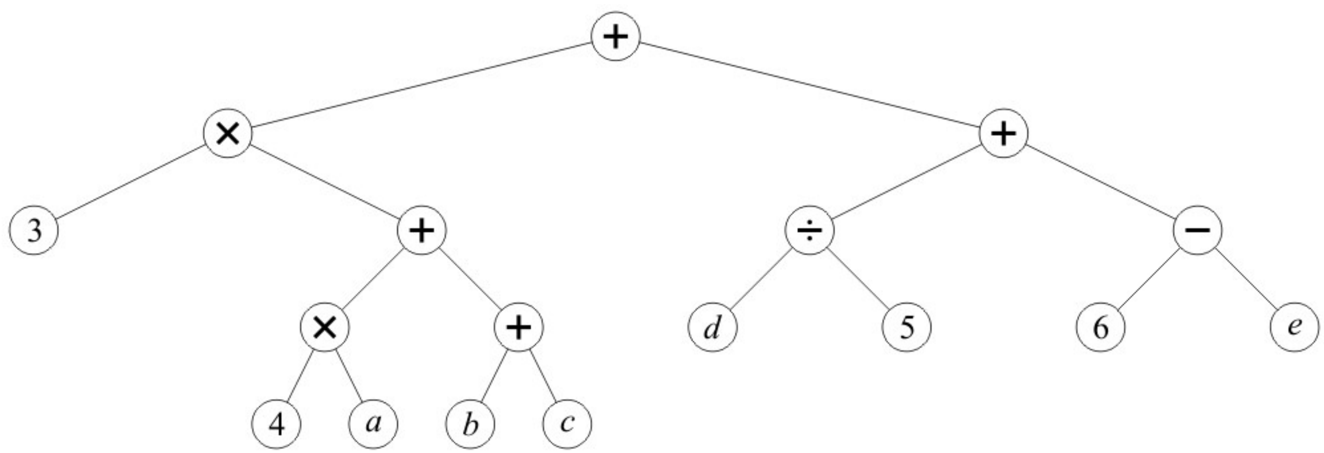


These have applications in
- Expression trees
- Huffman encoding

Application: Expression Trees

Any basic mathematical expression containing binary operators may be represented using a binary tree

For example, $3(4a + b + c) + d/5 + (6 - e)$



Application: Expression Trees

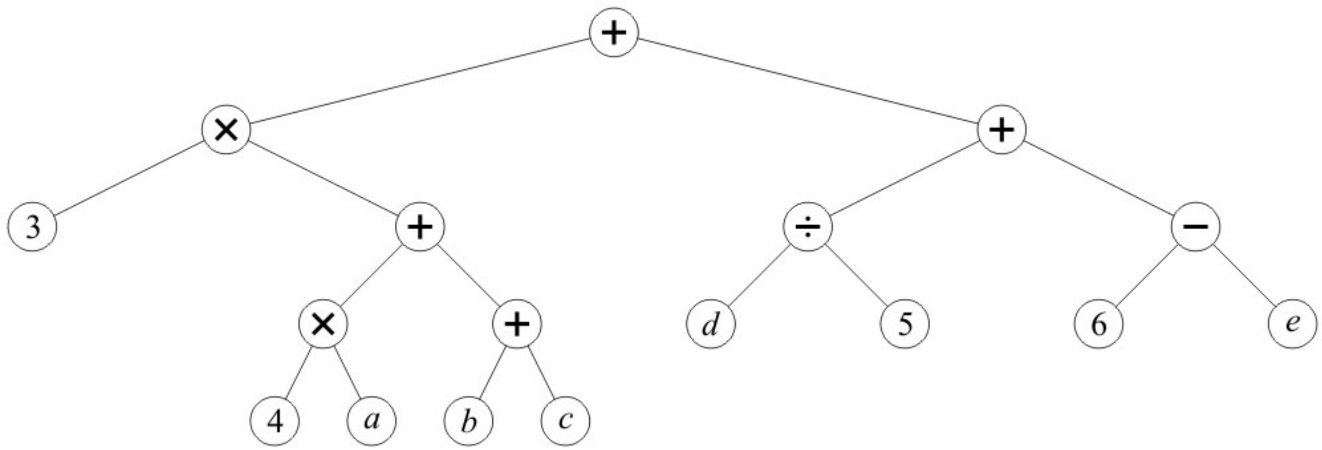
Observations:

- Internal nodes store operators
- Leaf nodes store literals or variables
- No nodes have just one sub tree
- The order is not relevant for
  - Addition and multiplication (commutative)
- Order is relevant for
  - Subtraction and division (non-commutative)
- It is possible to replace non-commutative operators using the unary negation and inversion:

$$(a/b) = a\,b^{-1} \qquad (a - b) = a + (-b)$$

Application: Expression Trees

A post-order depth-first traversal converts such a tree to the reverse-Polish format



$$3\ 4\ a\ \times\ b\ c\ +\ +\ \times\ d\ 5\ \div\ 6\ e\ -\ +\ +$$

Binary Tree Traversal

Binary Tree Traversal

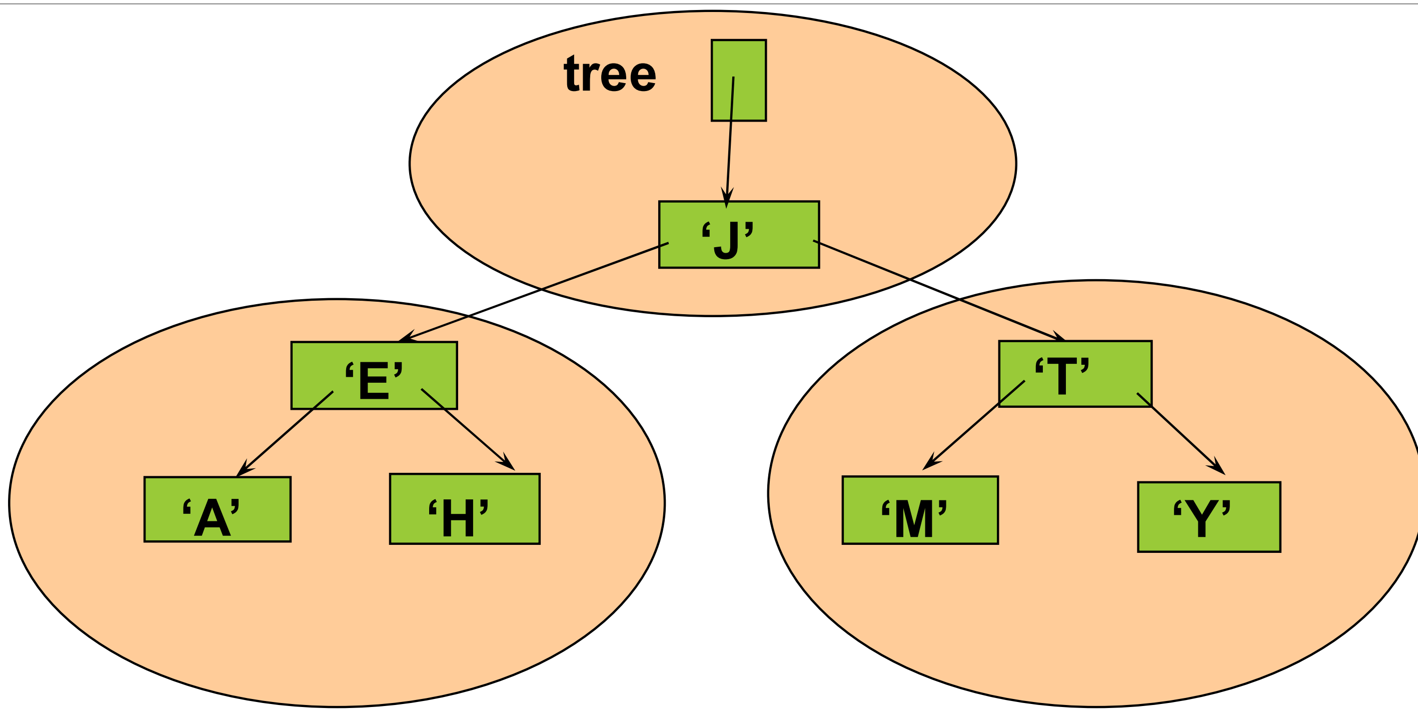
Many binary tree operations are done by performing a traversal of the binary tree

In a traversal, each element of the binary tree is visited exactly once

In binary trees there are three basic ways to traverse a tree using the a depth-first search idea (in fact there may be others but the ones below are the most common for binary trees)

- **Preorder**: We visit a node, then visit the left and the right subtrees NLR
- **Inorder**: We visit the left subtree then we visit the node, then we visit the right subtree LNR
- **Postorder**: We visit the left and right subtree and then we visit the node. This is what normally authors mean if they mention just depth -first traversal LRN

Preorder Traversal:

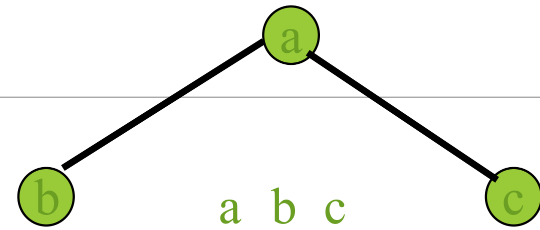


Visit left subtree second

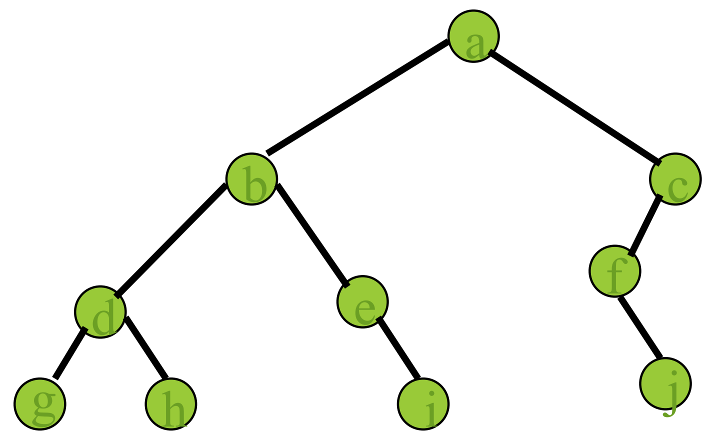
Visit right subtree last

Preorder Traversal

```
Void preOrder(BinaryTreeNode t)
{
      if (t != null)
      {
          visit(t);
          preOrder(t.leftChild);
          preOrder(t.rightChild);
      }
}
```

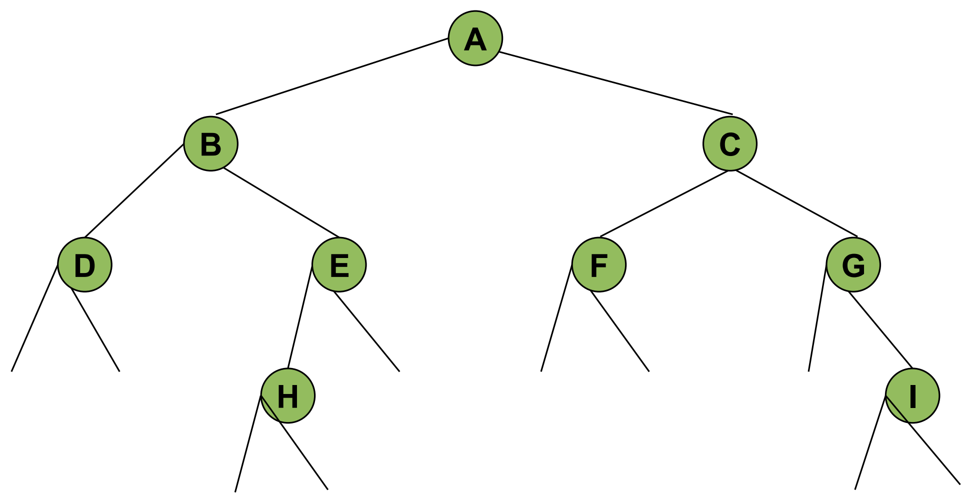


a b c

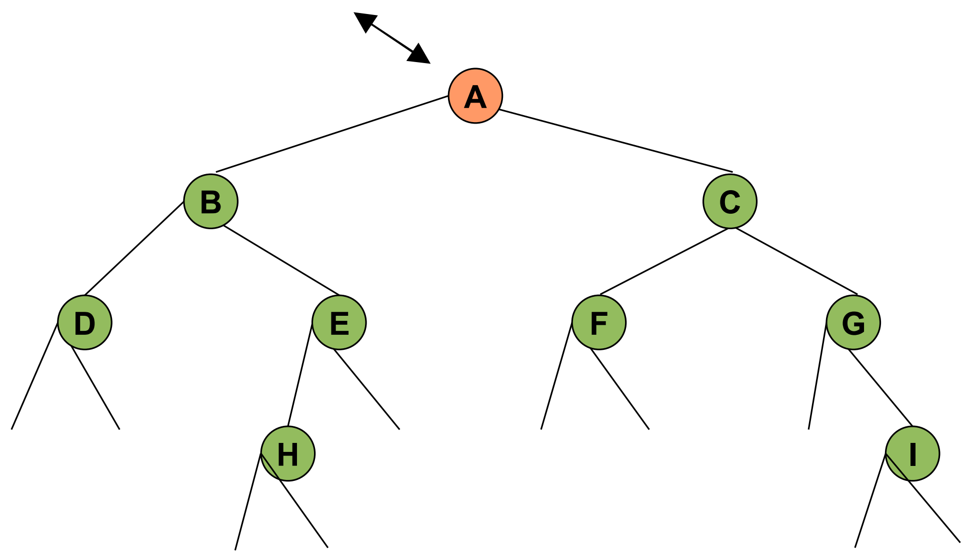


a b d g h e i c f j

Traversing a Tree Preorder

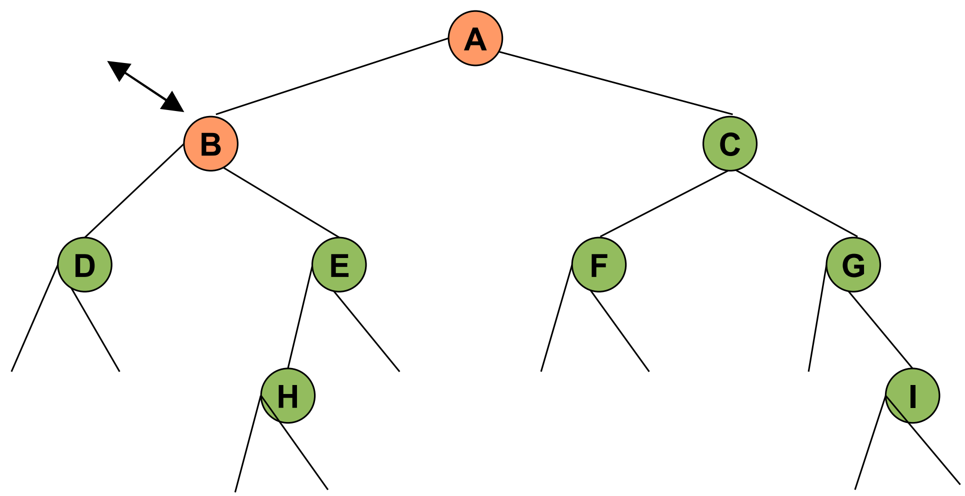


Traversing a Tree Preorder



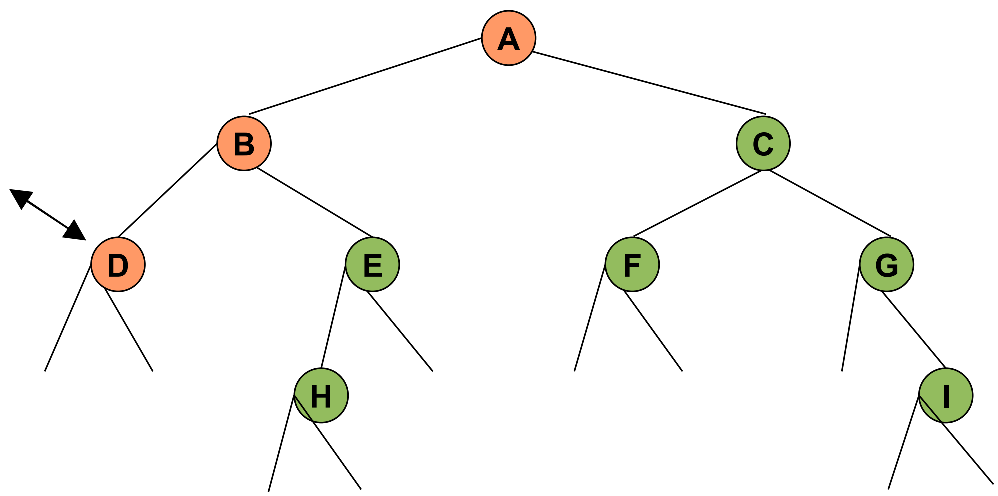
Result: A

Traversing a Tree Preorder



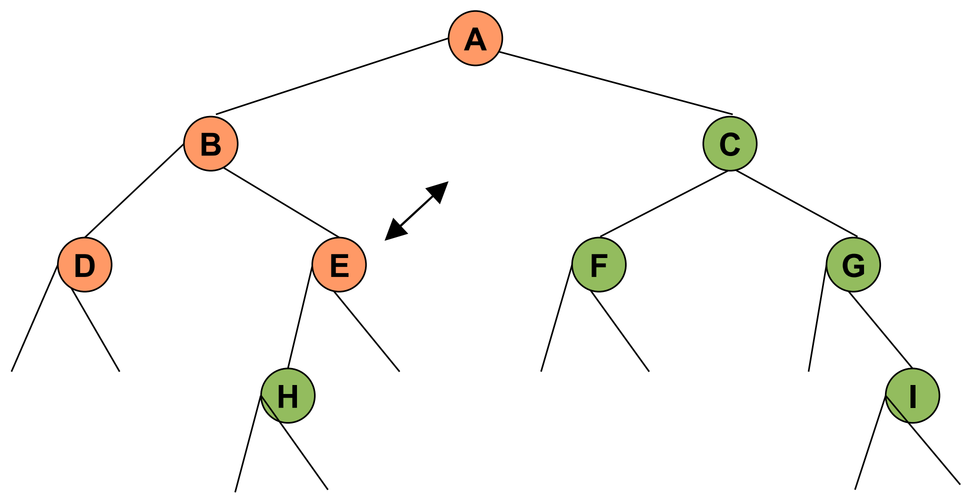
Result: AB

Traversing a Tree Preorder



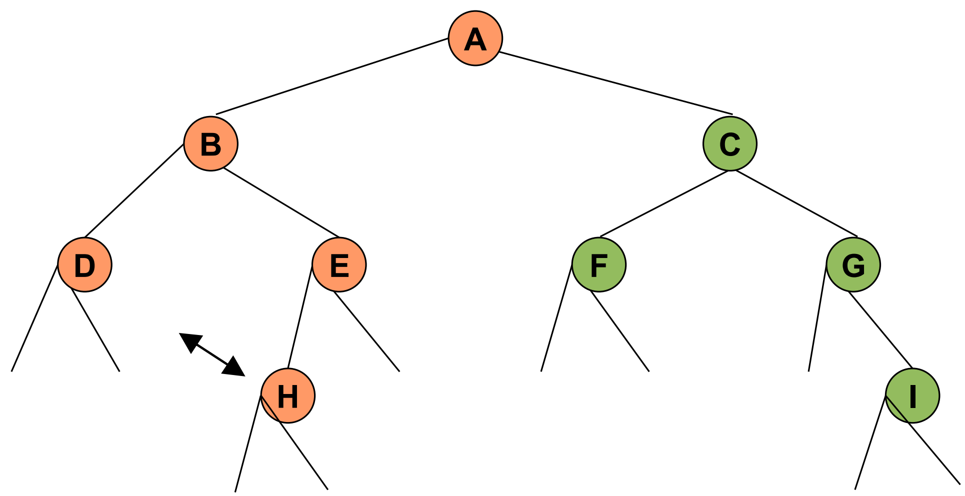
Result: ABD

Traversing a Tree Preorder



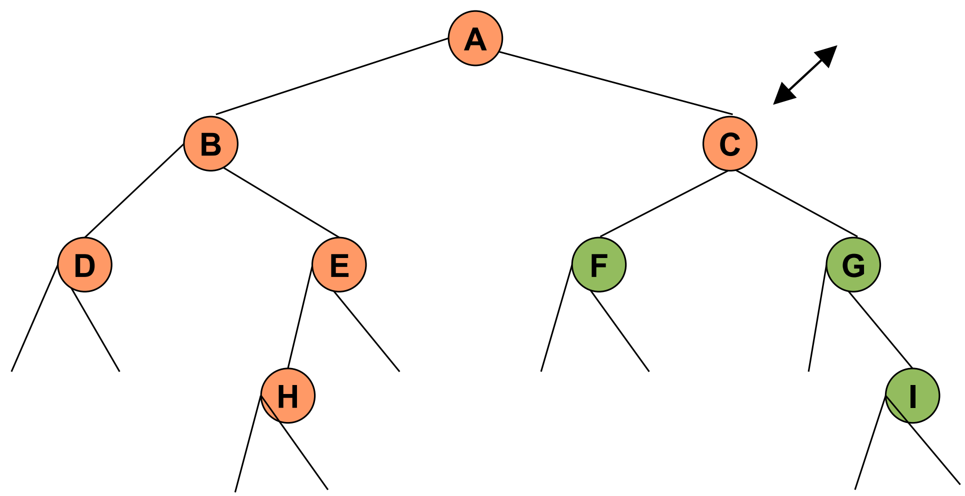
Result: ABDE

Traversing a Tree Preorder



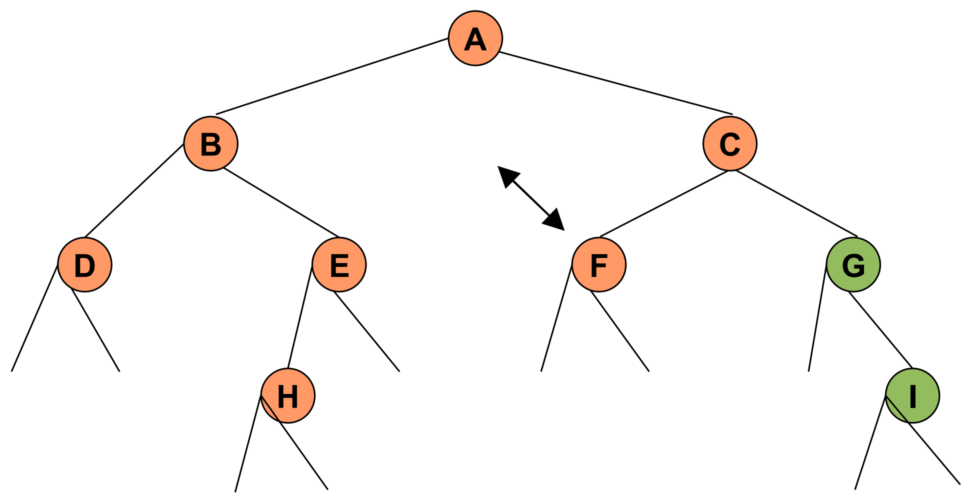
Result: ABDEH

Traversing a Tree Preorder



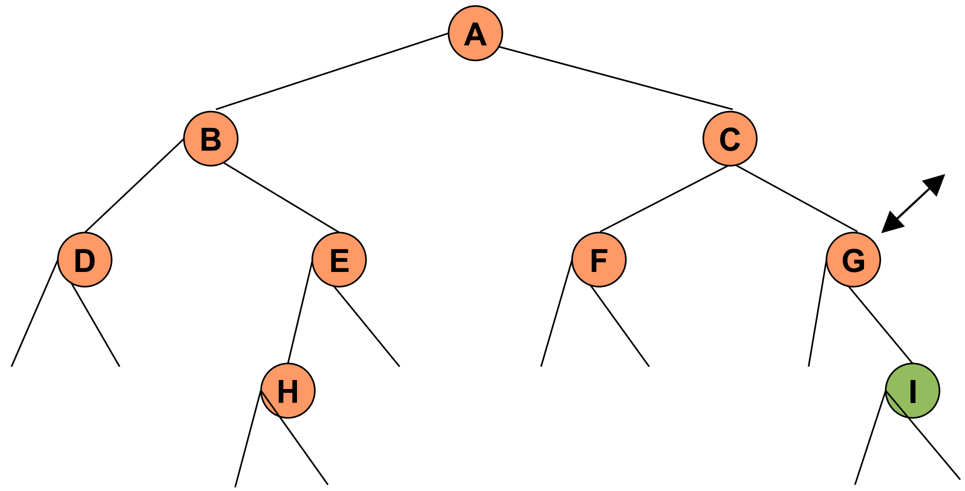
Result: ABDEHC

Traversing a Tree Preorder



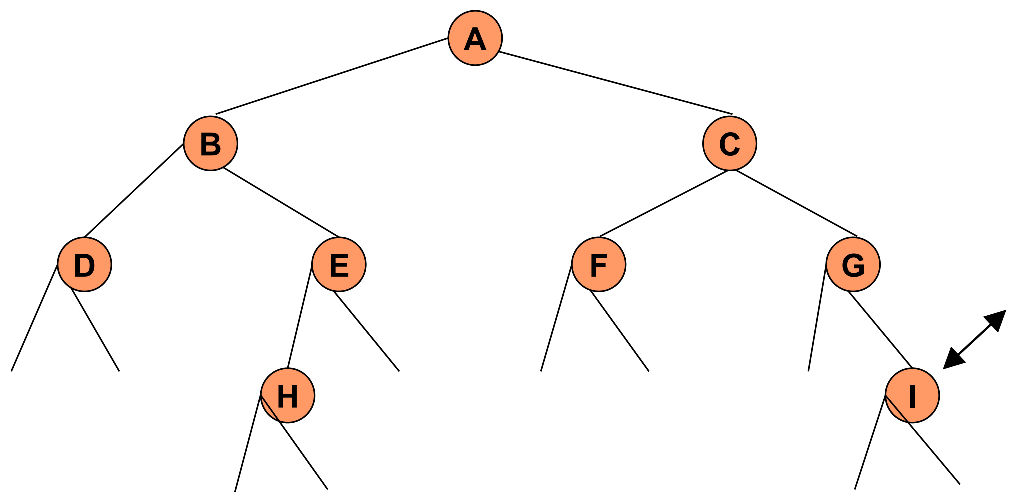
Result: ABDEHCF

Traversing a Tree Preorder



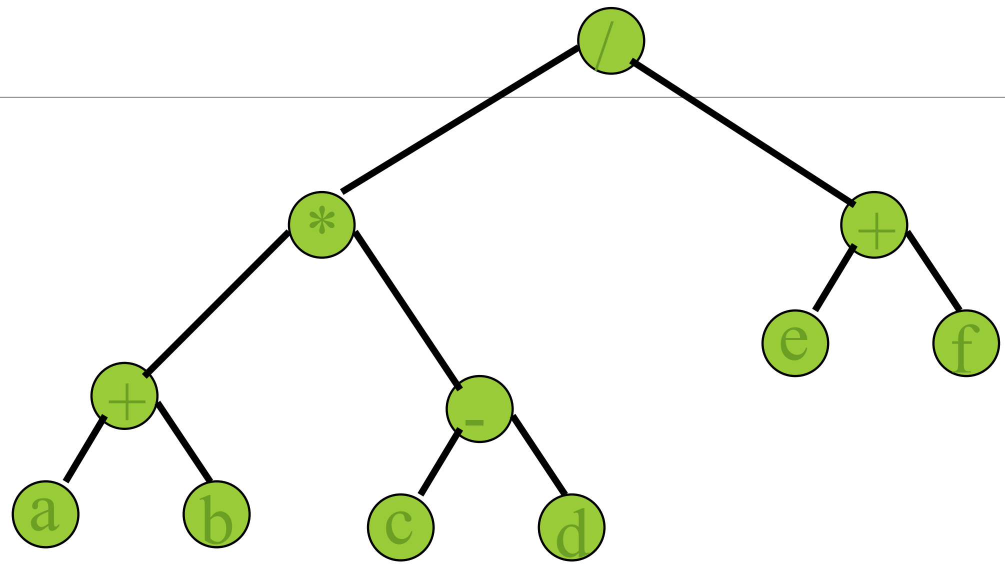
Result: ABDEHCFG

Traversing a Tree Preorder



Result: ABDEHCFGI

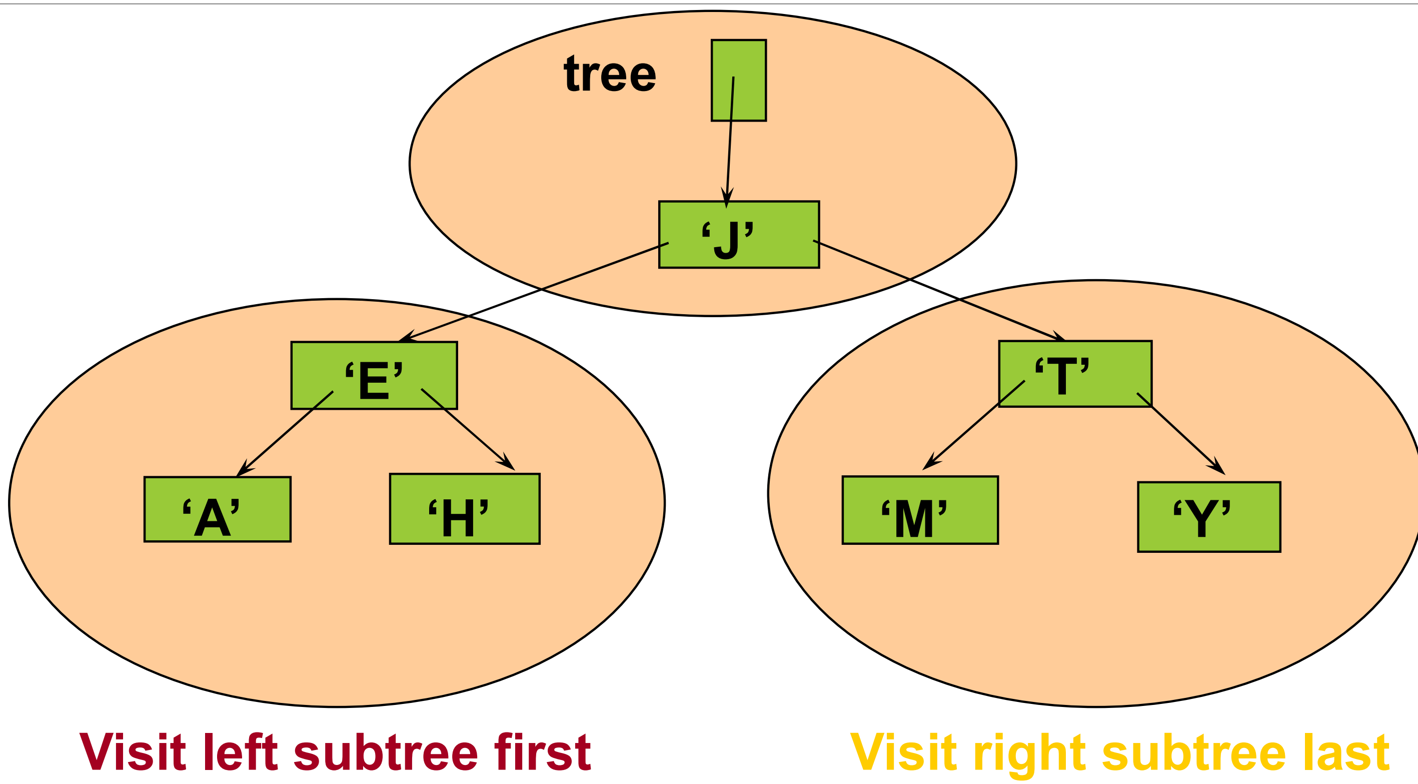
Preorder Of Expression Tree



/ * + a b - c d + e f

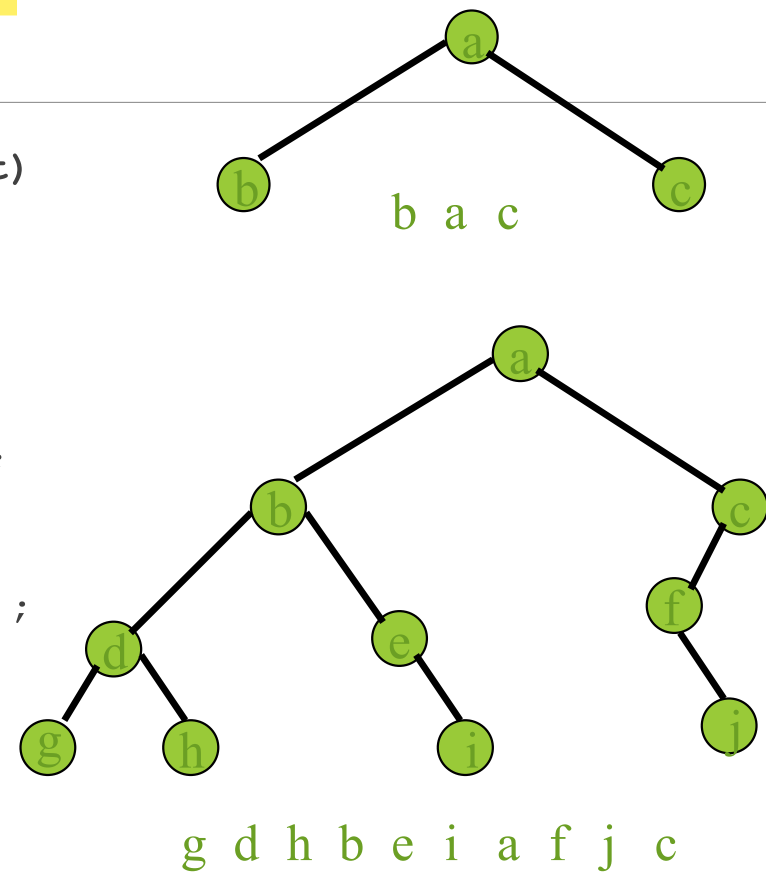
Gives prefix form of expression!

Inorder Traversal:

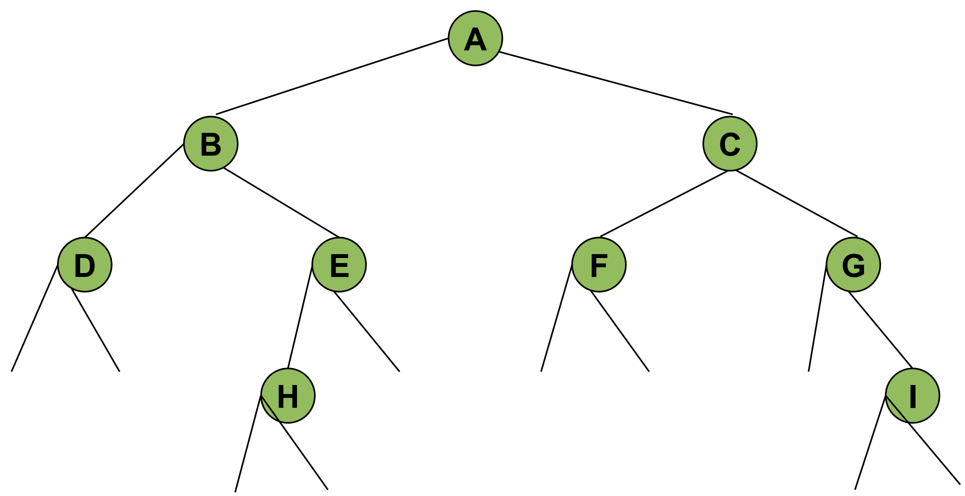


Inorder Traversal

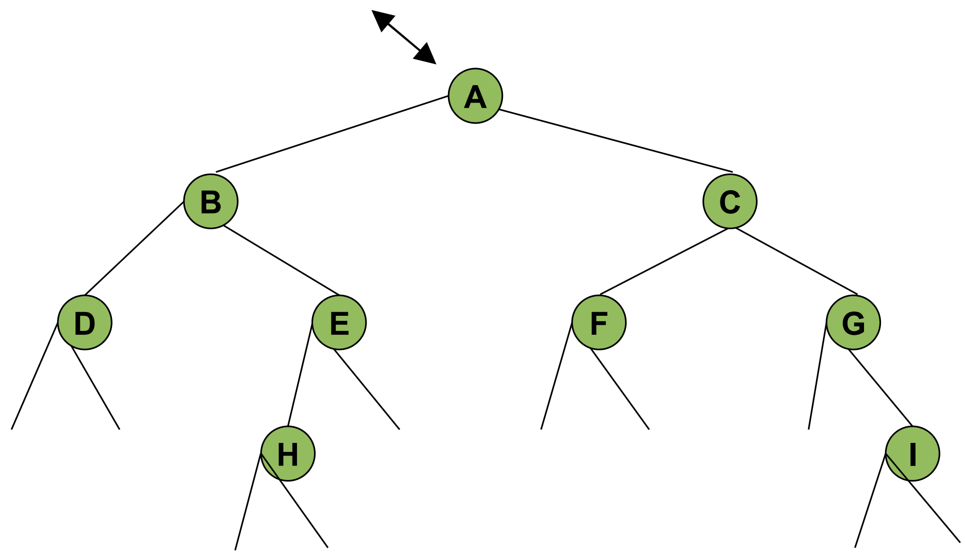
```
Void inOrder(BinaryTreeNode t)
{
      if (t != null)
      {
          inOrder(t.leftChild);
          visit(t);
          inOrder(t.rightChild);
      }
}
```



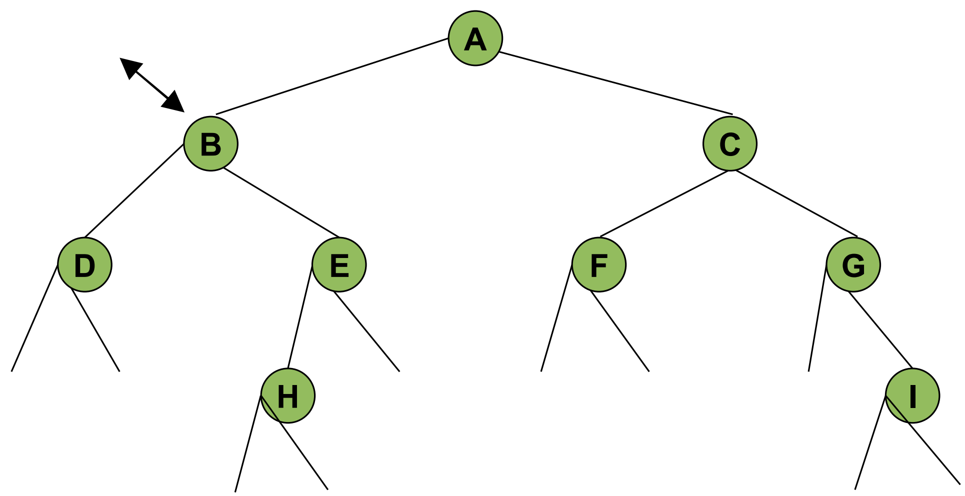
Traversing a Tree Inorder



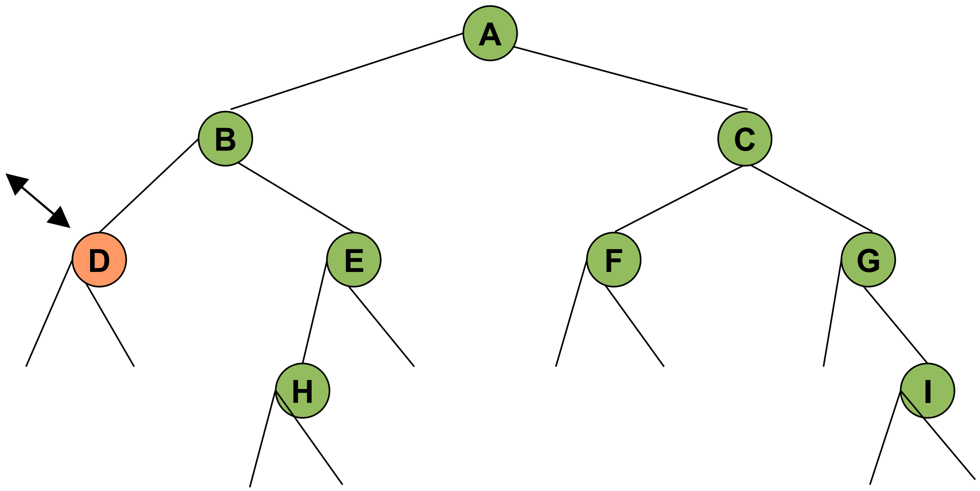
Traversing a Tree Inorder



Traversing a Tree Inorder

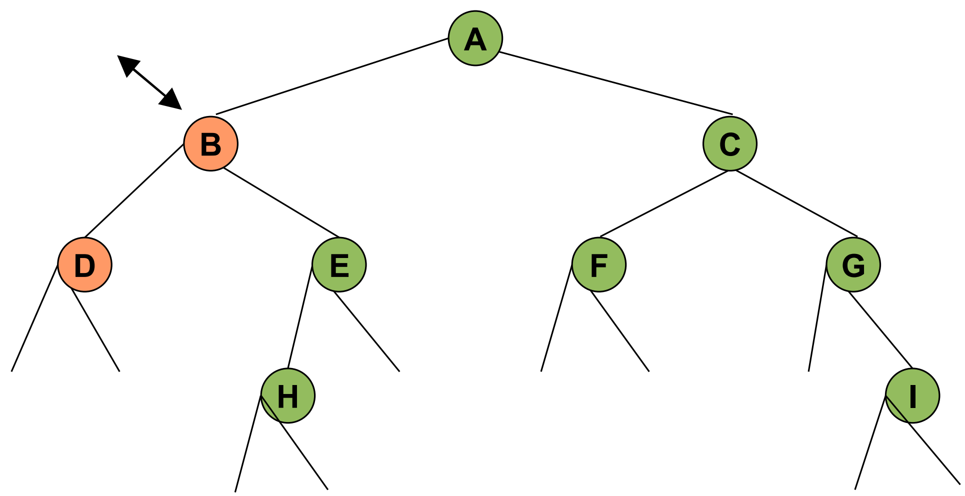


Traversing a Tree Inorder



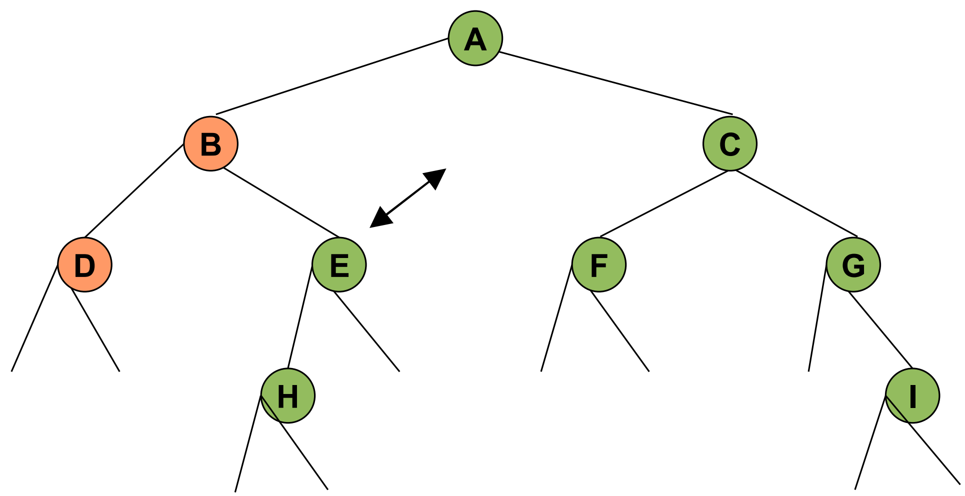
Result: D

Traversing a Tree Inorder



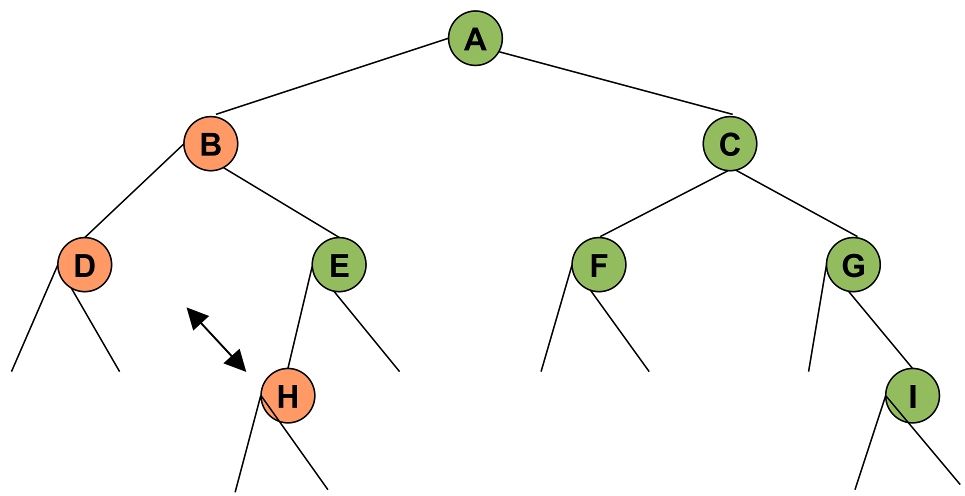
Result: DB

Traversing a Tree Inorder



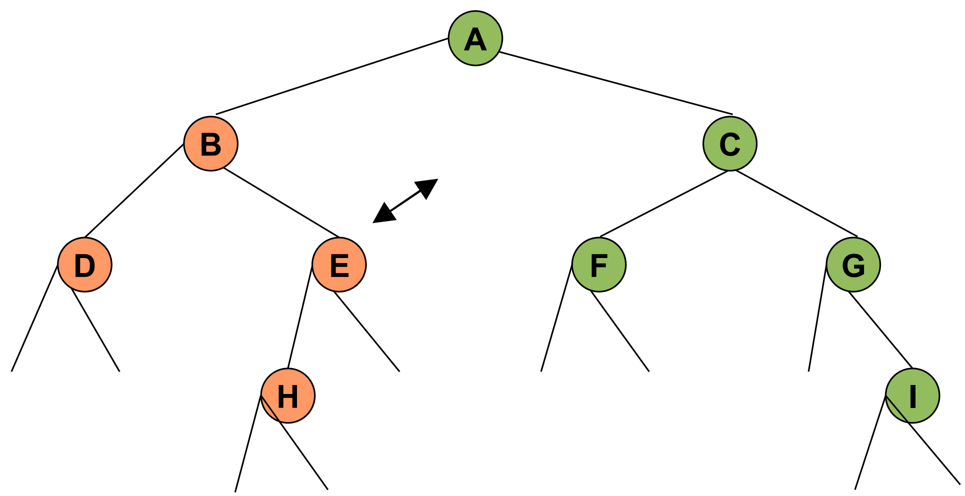
Result: DB

Traversing a Tree Inorder



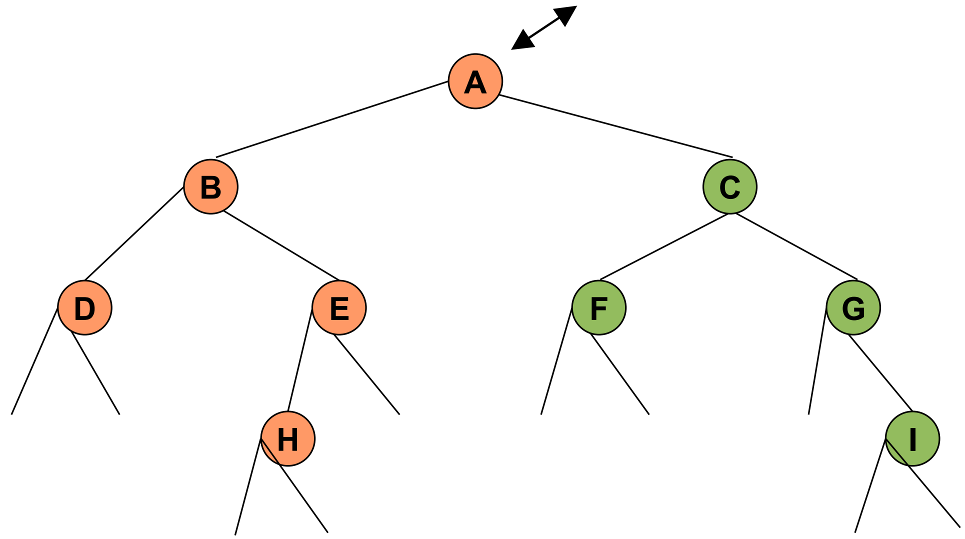
Result: DBH

Traversing a Tree Inorder



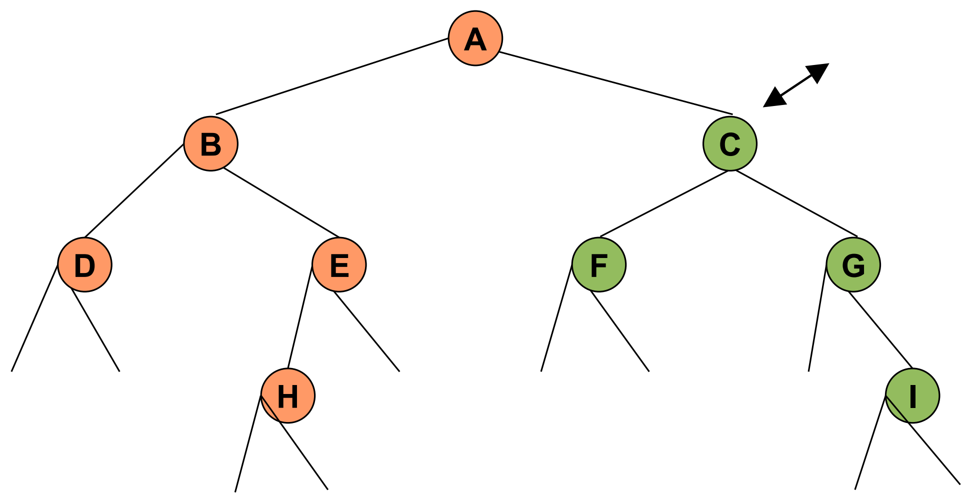
Result: DBHE

Traversing a Tree Inorder



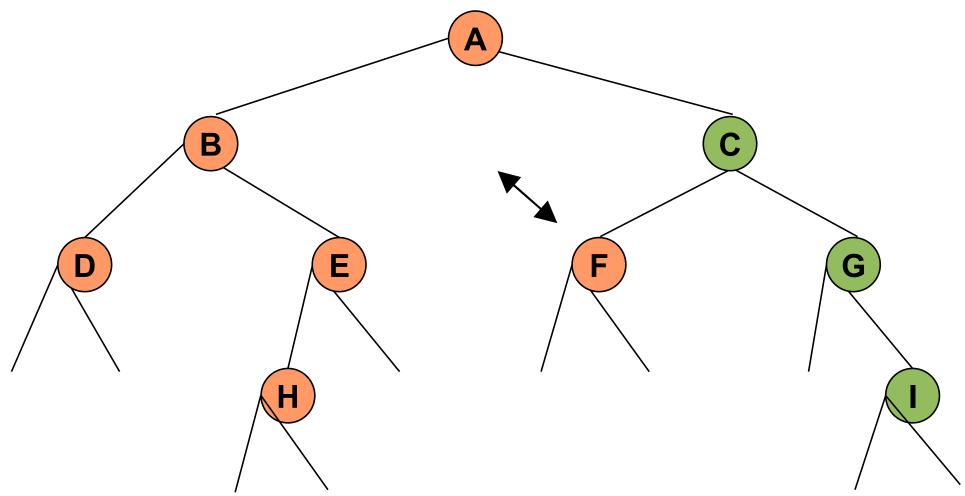
Result: DBHEA

Traversing a Tree Inorder



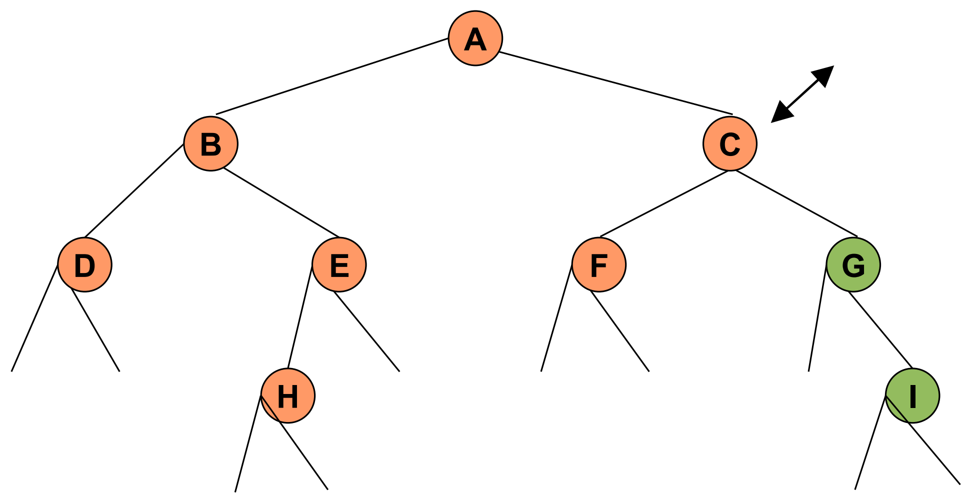
Result: DBHEA

Traversing a Tree Inorder



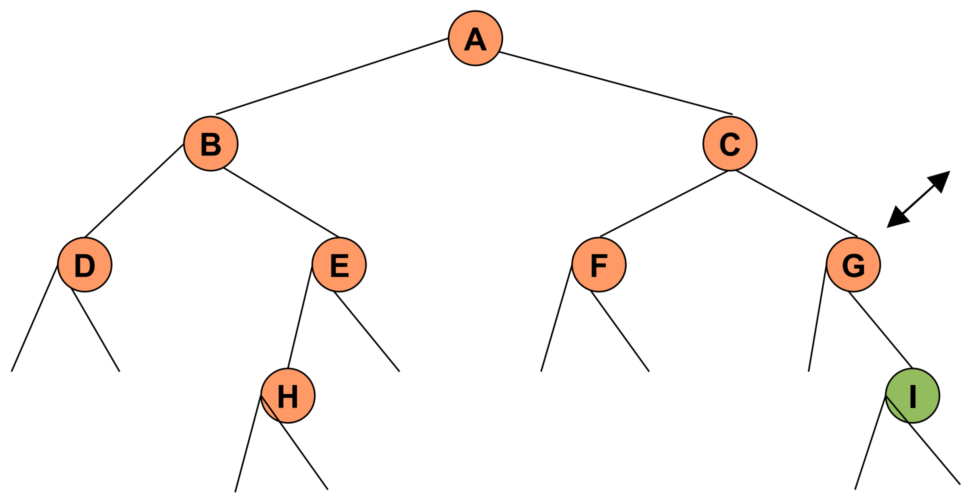
Result: DBHEAF

Traversing a Tree Inorder



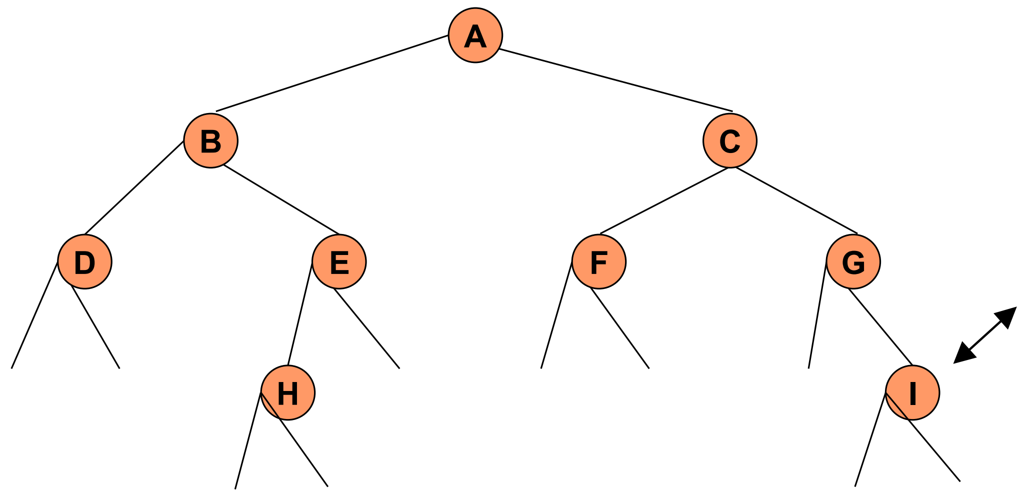
Result: DBHEAFC

Traversing a Tree Inorder



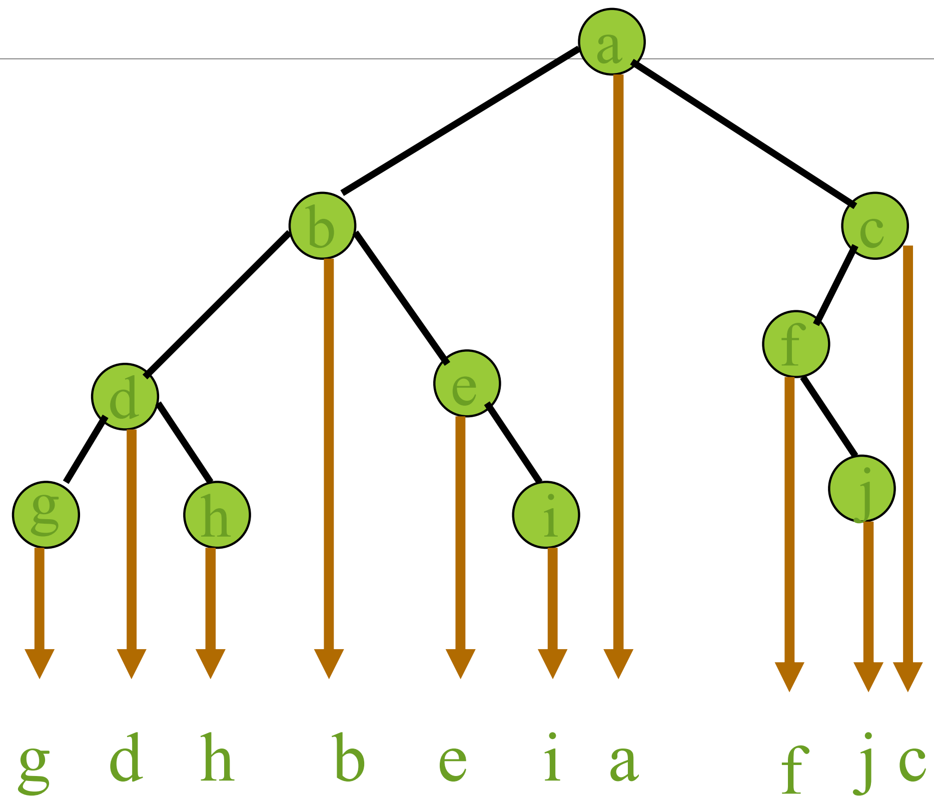
Result: DBHEAFCG

Traversing a Tree Inorder

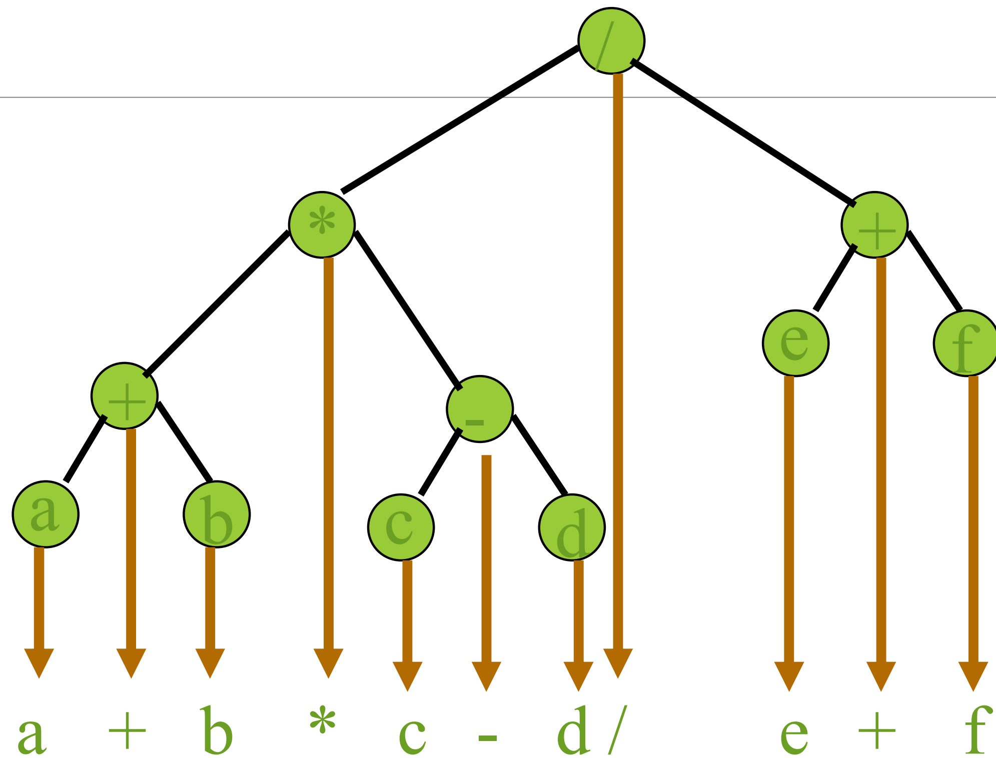


Result: DBHEAFCGI

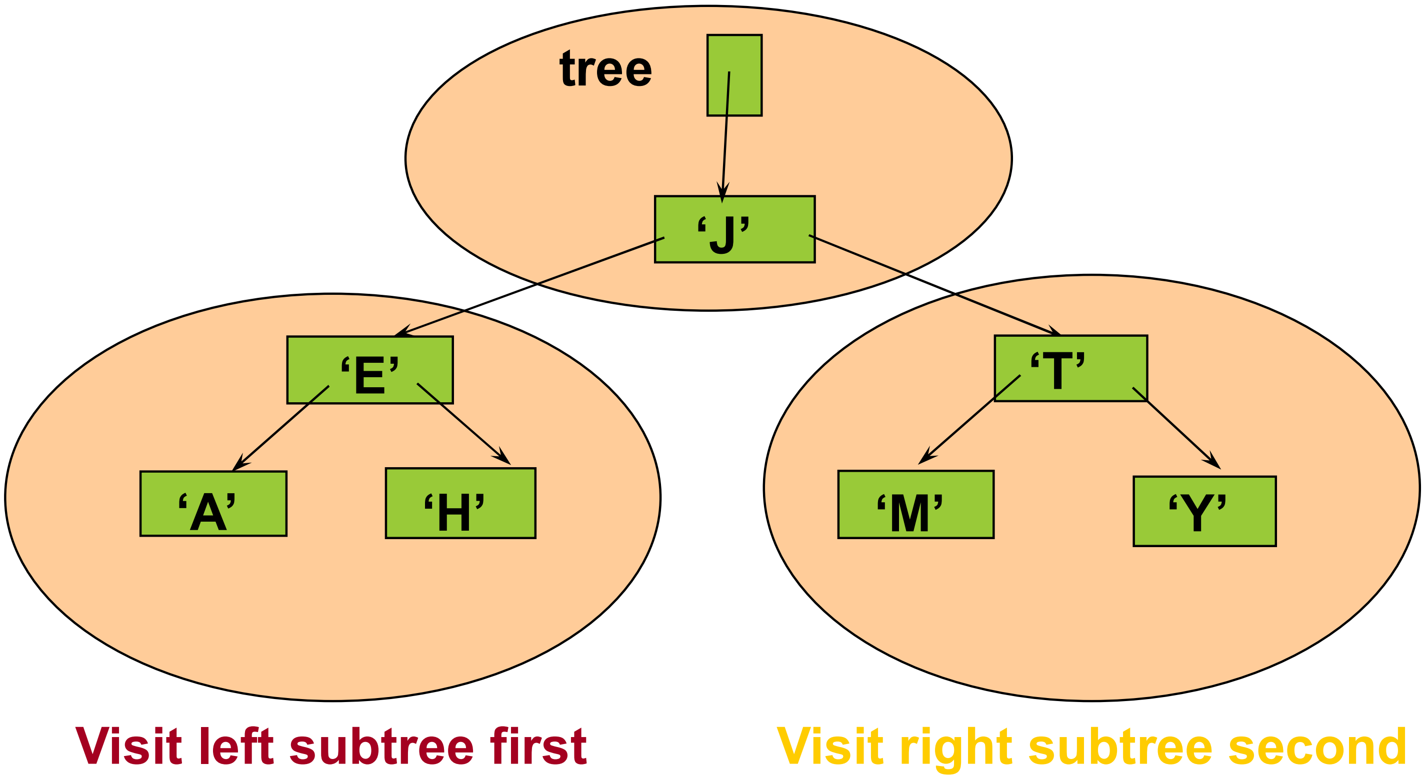
Inorder By Projection



Inorder Of Expression Tree

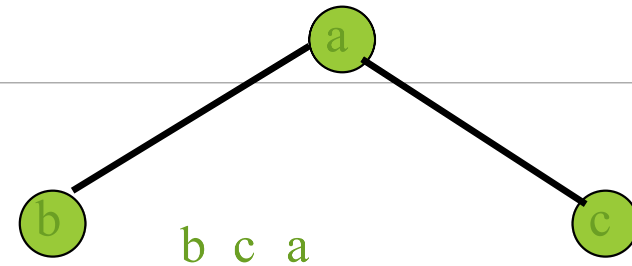


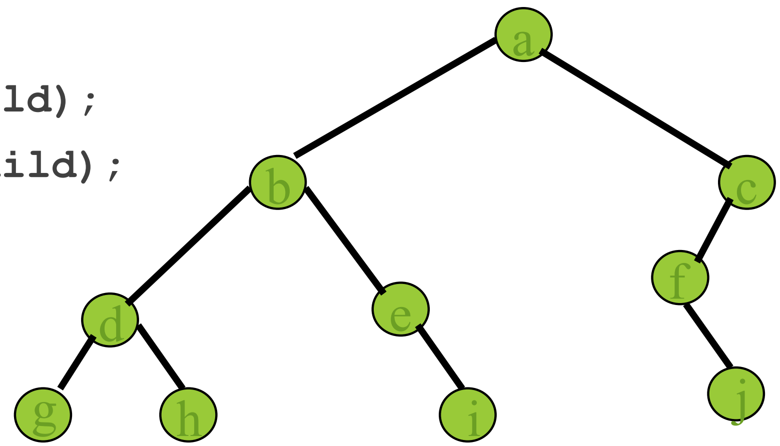
Postorder Traversal



Postorder Traversal

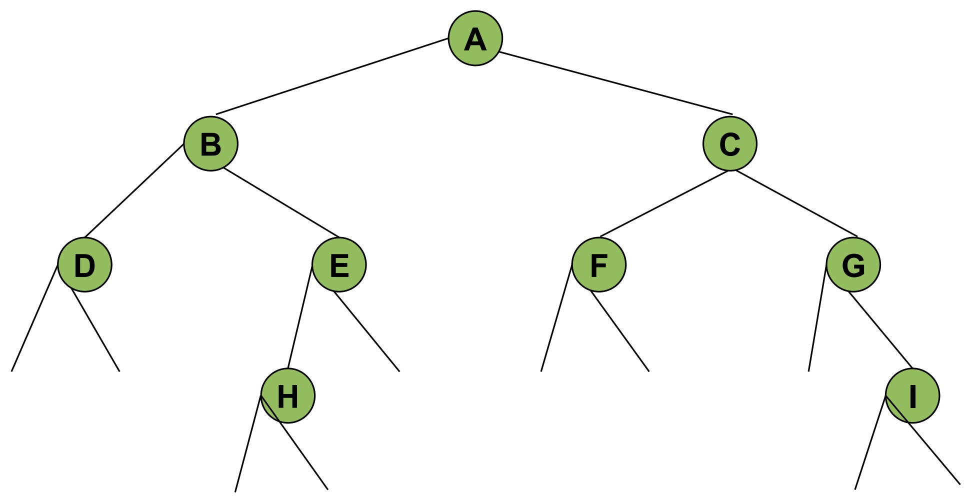
```
void postOrder(BinaryTreeNode t)
{
      if (t != null)
      {
         postOrder(t.leftChild);
         postOrder(t.rightChild);
         visit(t);
      }
}
```



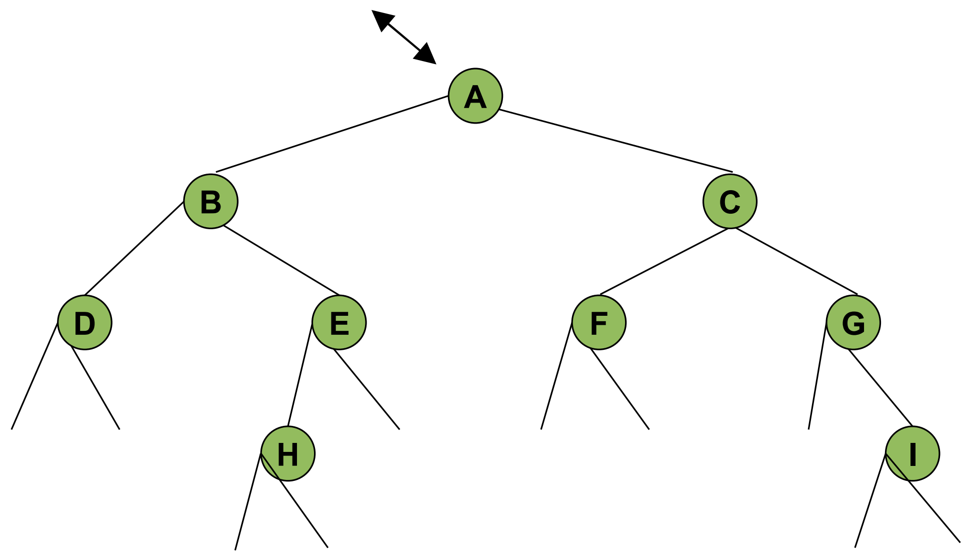


g h d i e b j f c a

Traversing a Tree Postorder

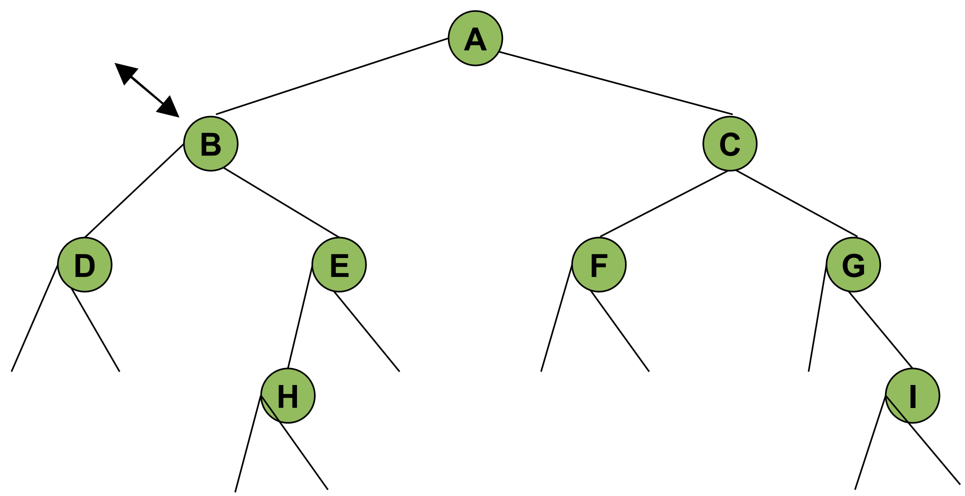


Traversing a Tree Postorder



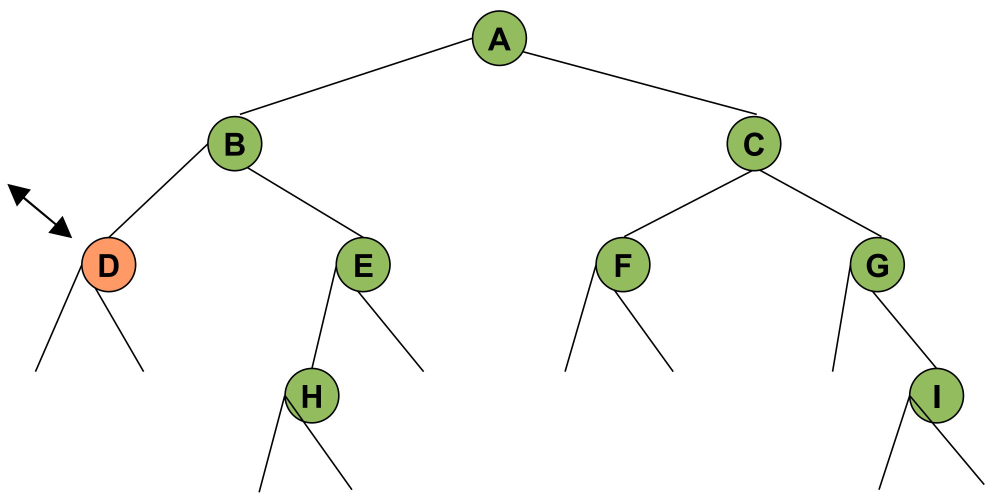
Result:

Traversing a Tree Postorder



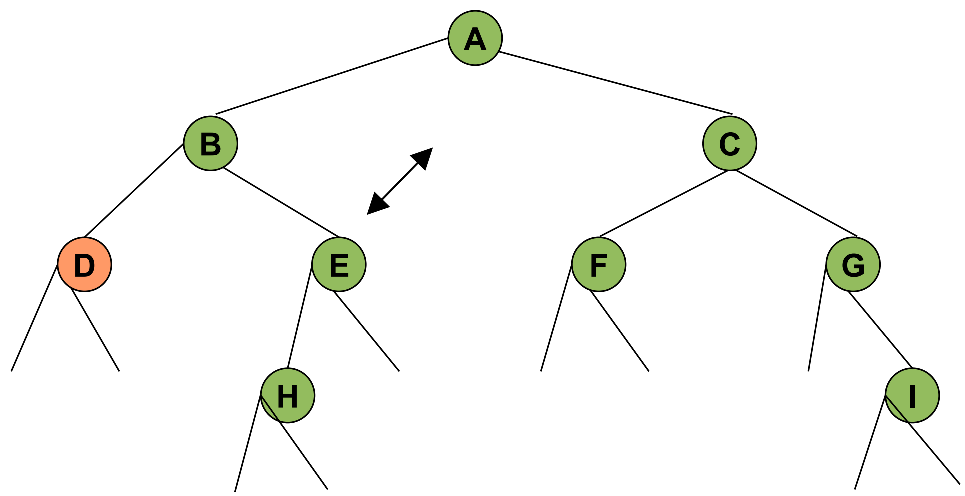
Result:

Traversing a Tree Postorder



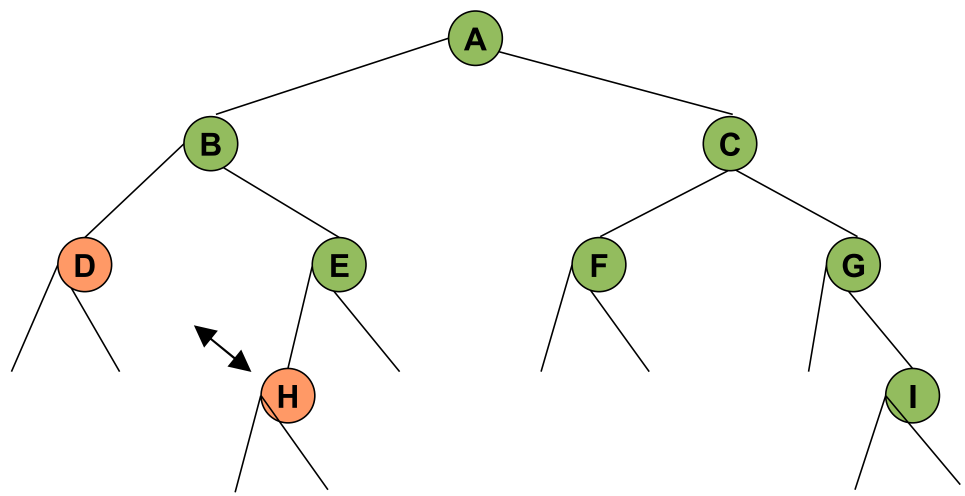
Result: D

Traversing a Tree Postorder



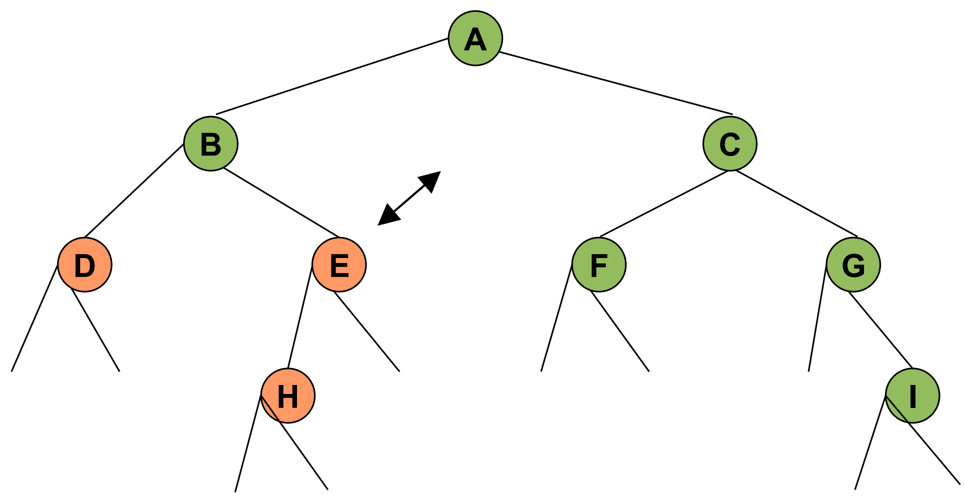
Result: D

Traversing a Tree Postorder



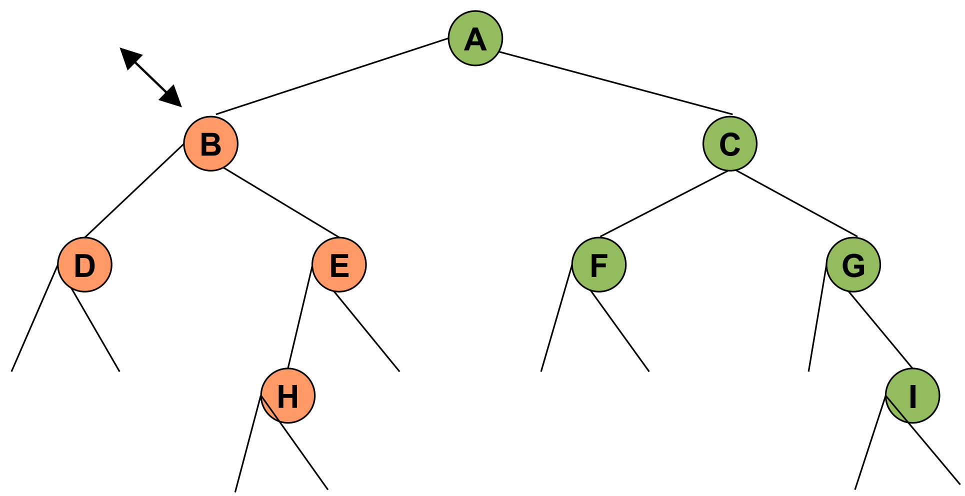
Result: DH

Traversing a Tree Postorder



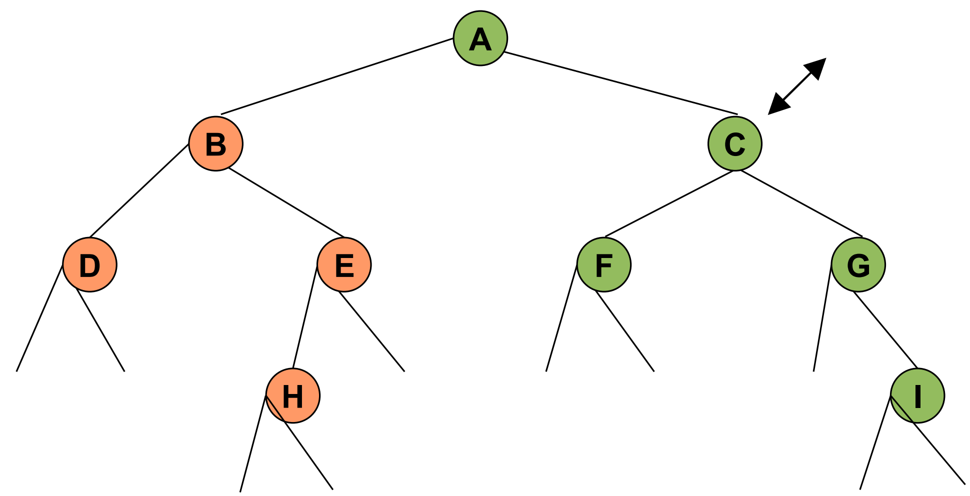
Result: DHE

Traversing a Tree Postorder



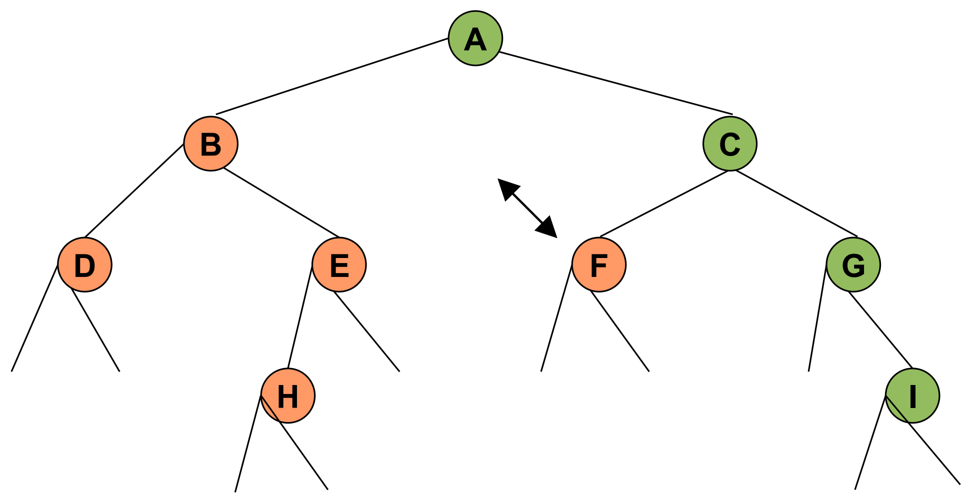
Result: DHEB

Traversing a Tree Postorder



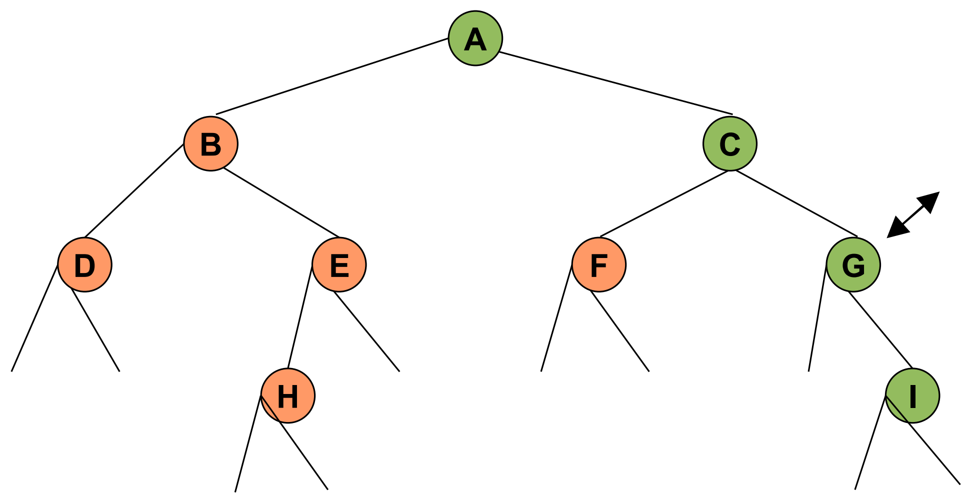
Result: DHEB

Traversing a Tree Postorder



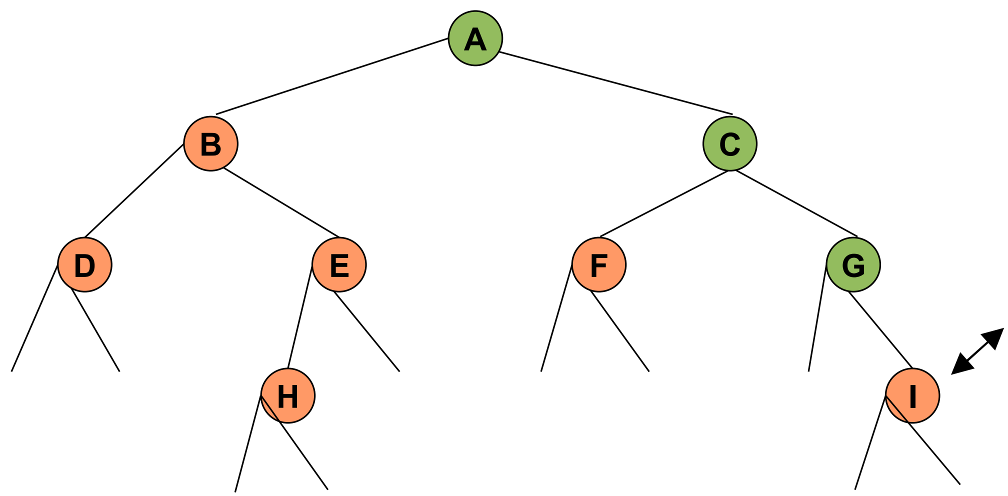
Result: DHEBF

Traversing a Tree Postorder



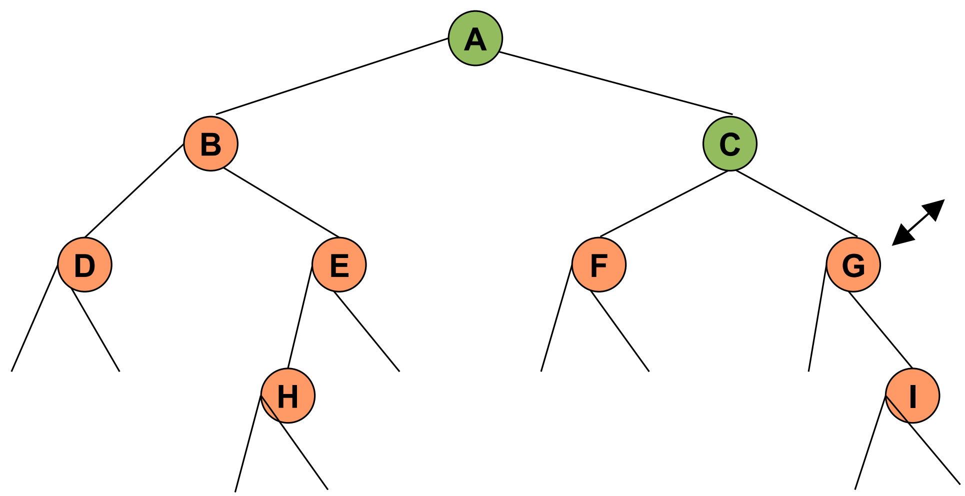
Result: DHEBF

Traversing a Tree Postorder



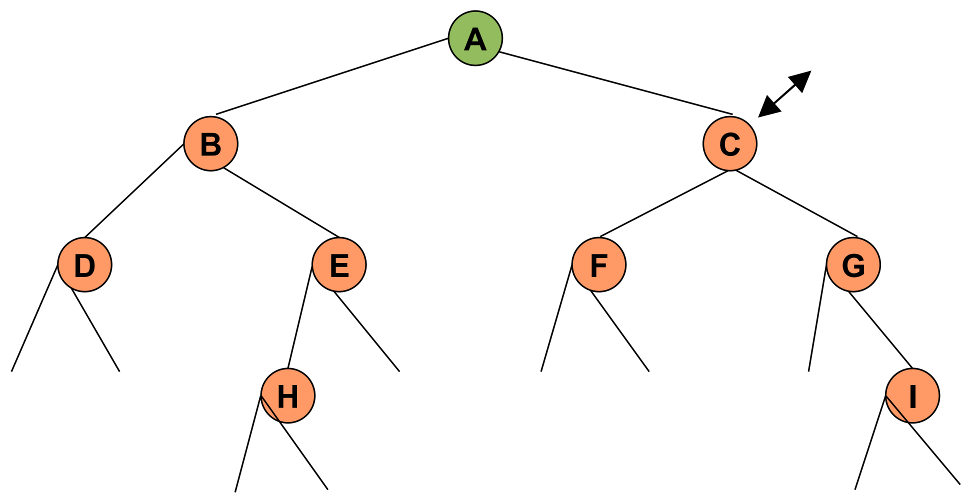
Result: DHEBFI

Traversing a Tree Postorder



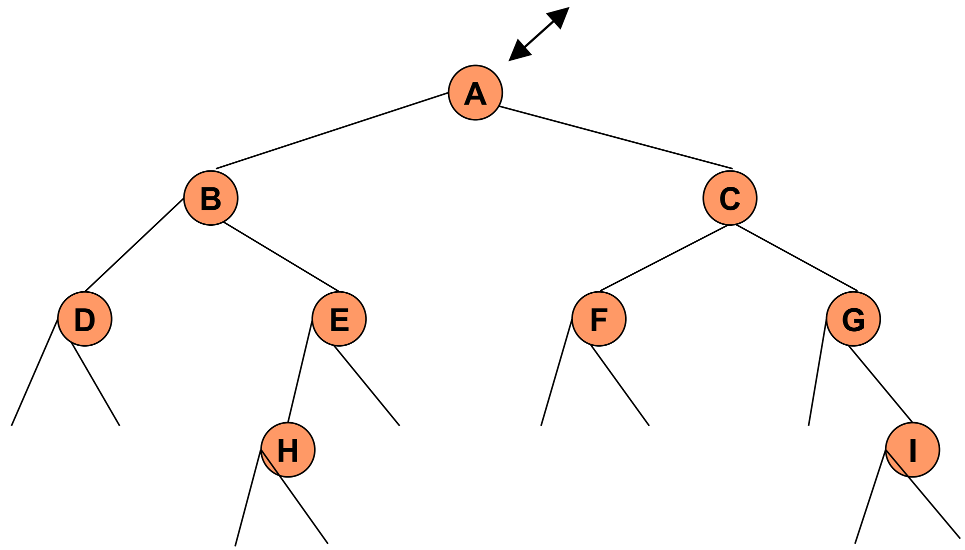
Result: DHEBFIG

Traversing a Tree Postorder



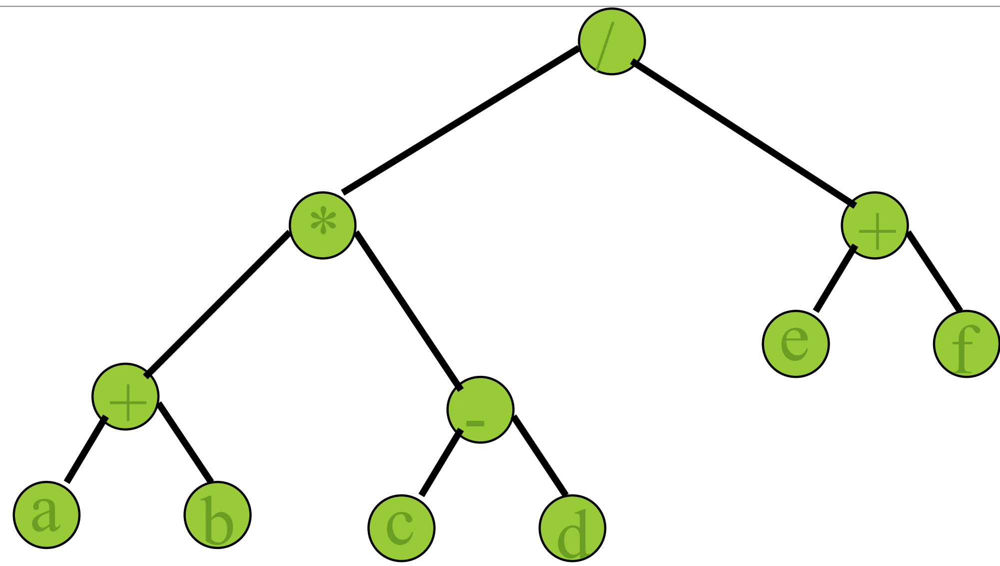
Result: DHEBFIGC

Traversing a Tree Postorder



Result: DHEBFIGCA

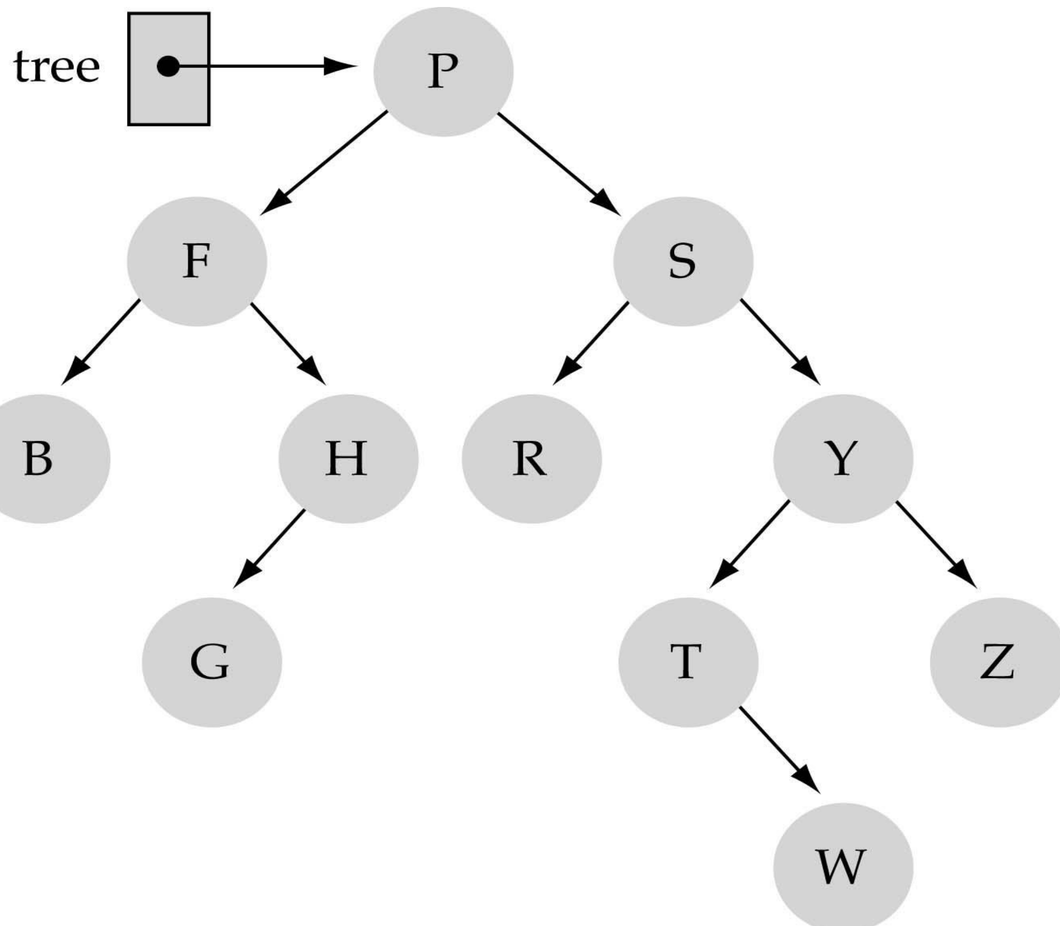
Postorder Of Expression Tree



a b + c d - * e f + /

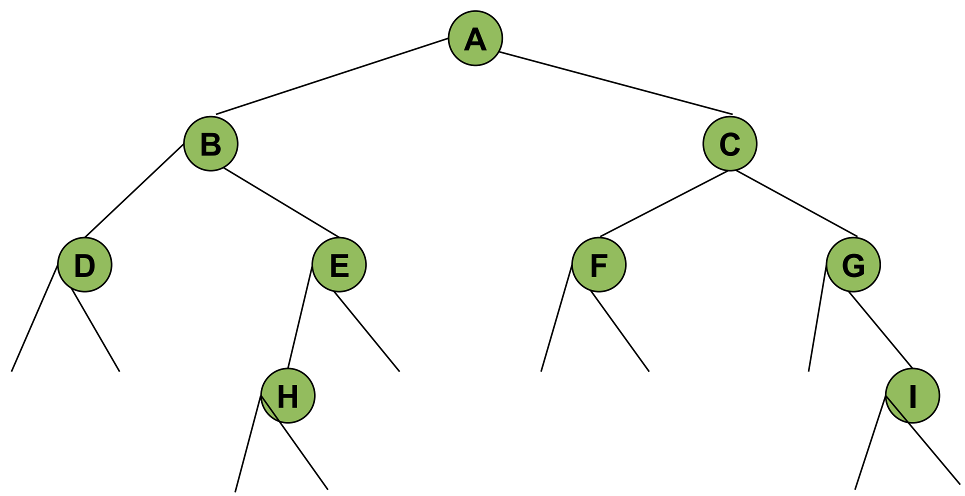
Gives postfix form of expression!

Tree Traversals

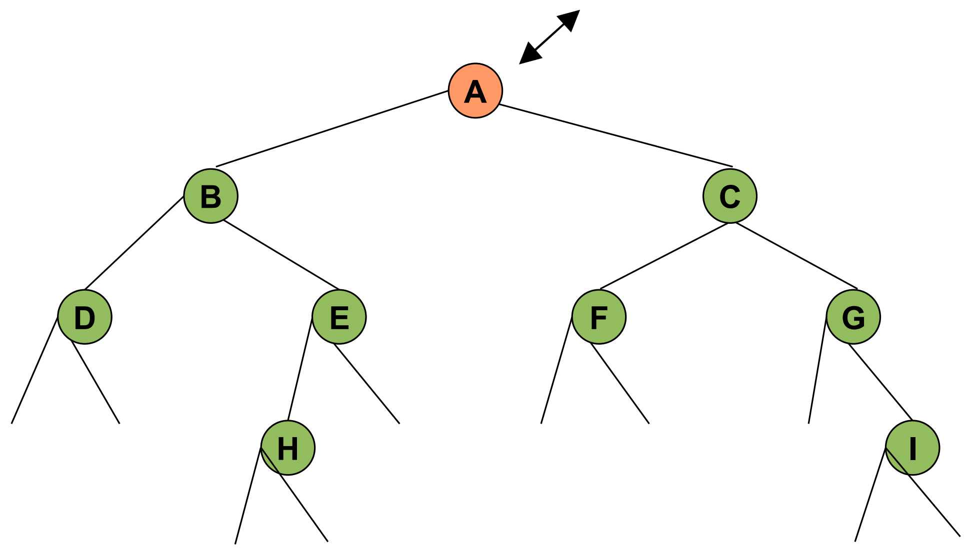


Inorder: B F G H P R S T W Y Z
Preorder: P F B H G S R Y T W Z
Postorder: B G H F R W T Z Y S P

Breadth-first traversal of a tree

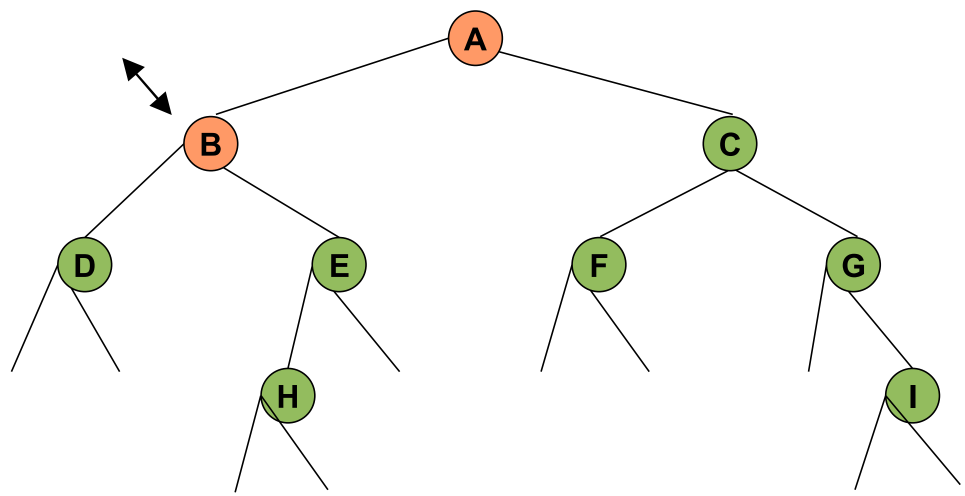


Breadth-first traversal of a tree



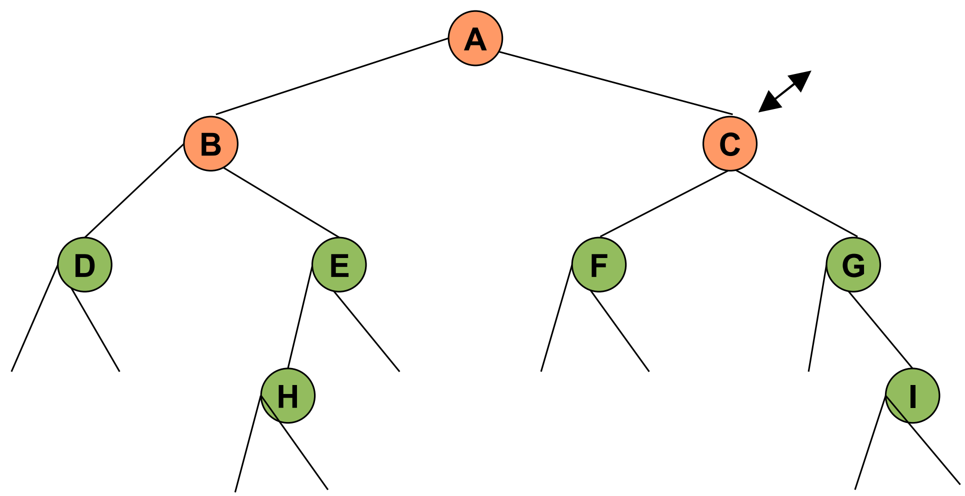
Result: A

Breadth-first traversal of a tree



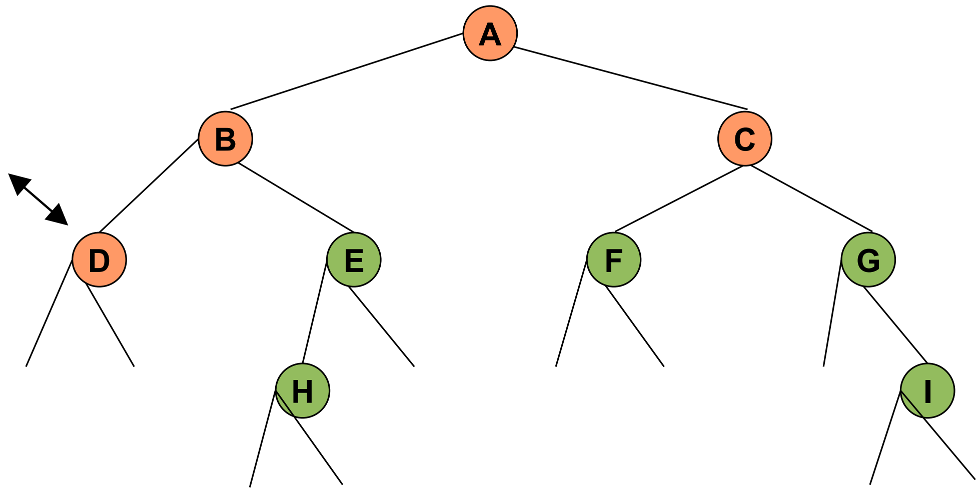
Result: AB

Breadth-first traversal of a tree



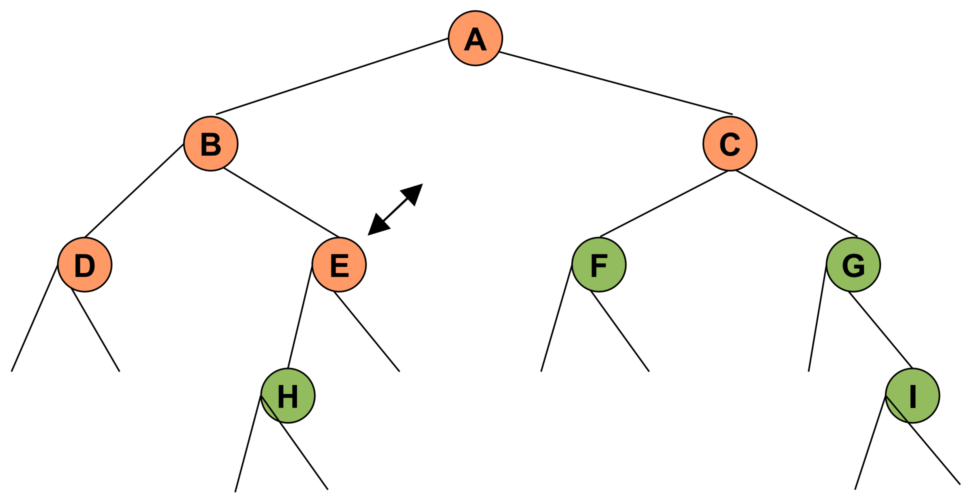
Result: ABC

Breadth-first traversal of a tree



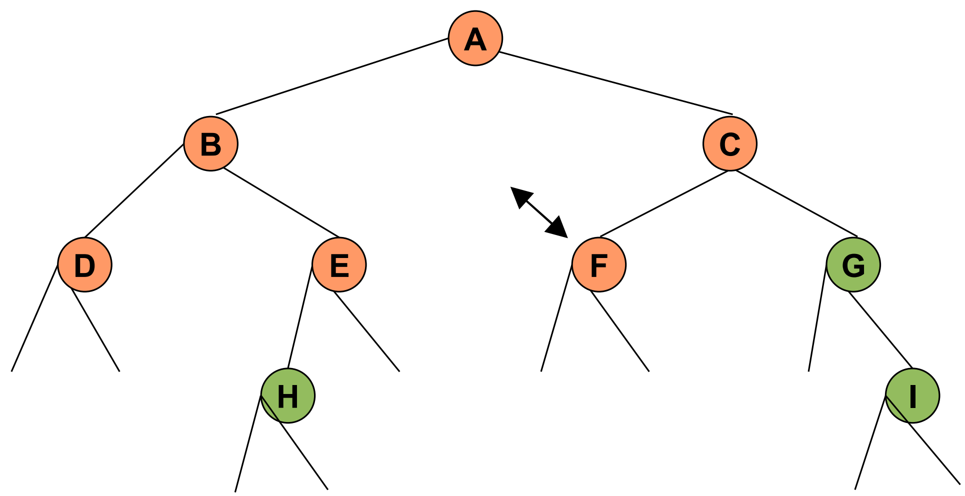
Result: ABCD

Breadth-first traversal of a tree



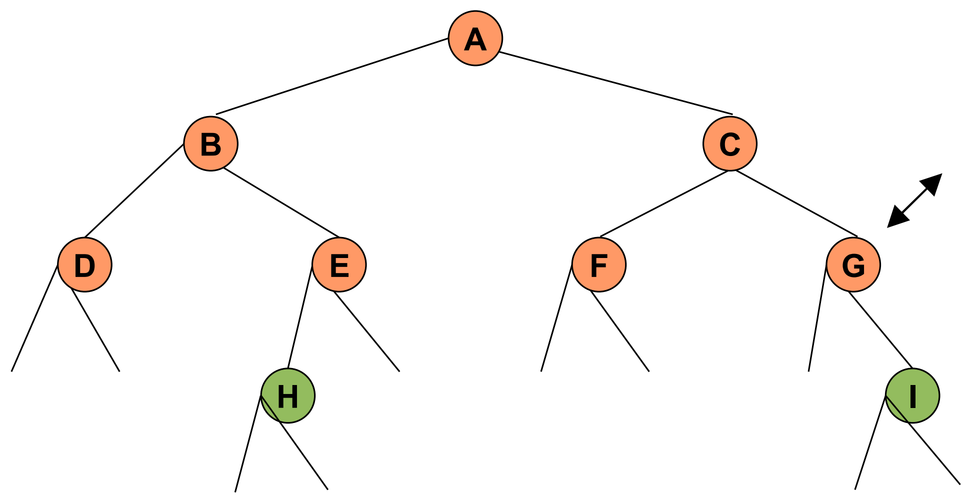
Result: ABCDE

Breadth-first traversal of a tree



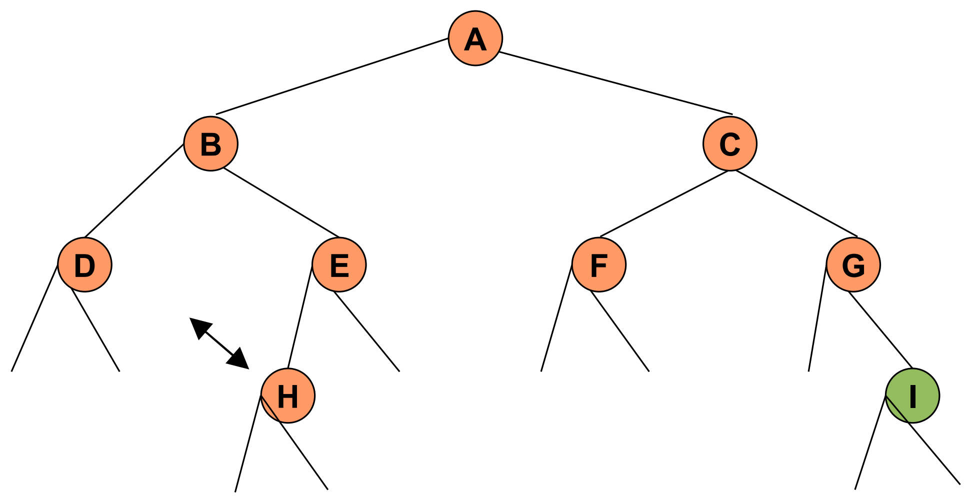
Result: ABCDEF

Breadth-first traversal of a tree



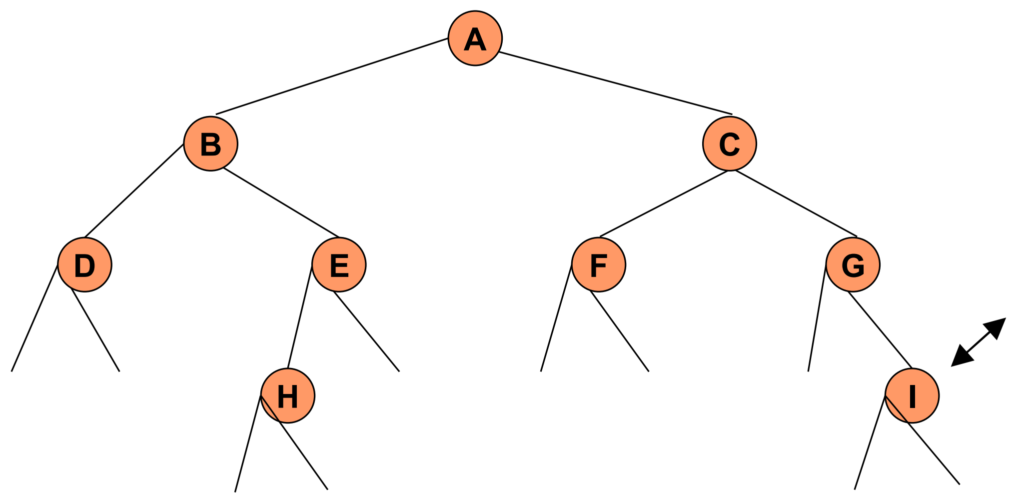
Result: ABCDEFG

Breadth-first traversal of a tree



Result: ABCDEFGH

Breadth-first traversal of a tree



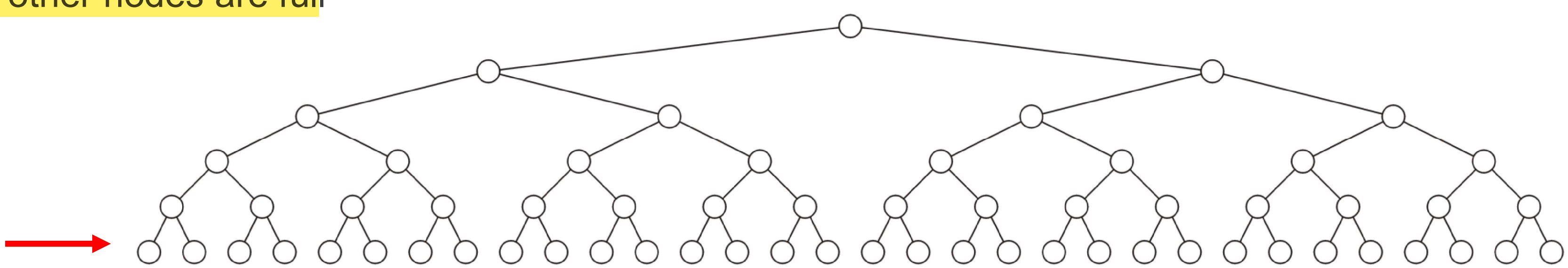
Result: ABCDEFGHI

Perfect binary trees

Definition

Standard definition:

- A perfect binary tree of height h is a binary tree where
  - All leaf nodes have the same depth h
  - All other nodes are full

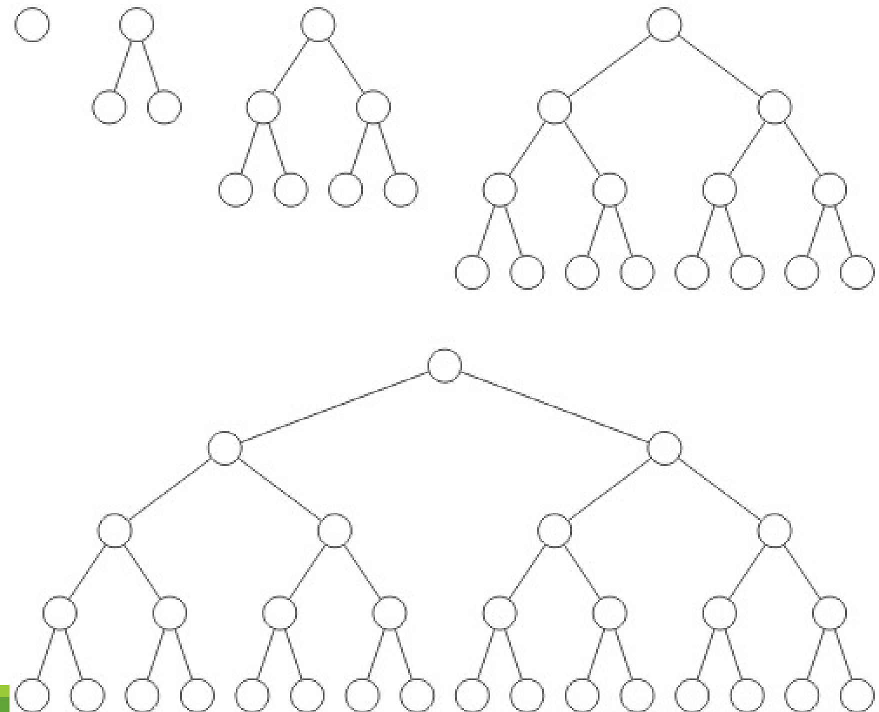


Recursive definition:

- A binary tree of height $h = 0$ is perfect
- A binary tree with height $h > 0$ is a perfect if both sub-trees are prefect binary trees of height $h - 1$

Examples

Perfect binary trees of height $h = 0, 1, 2, 3$ and 4



Theorems

Four theorems that describe the properties of perfect binary trees:

- A perfect tree has $2^{h+1} - 1$ nodes
- The height is $\Theta(\ln(n))$
- There are 2^h leaf nodes
- The average depth of a node is $\Theta(\ln(n))$

The results of these theorems will allow us to determine the optimal run-time properties of operations on binary trees

$2^{h+1} - 1$ Nodes

Theorem

A perfect binary tree of height h has $2^{h+1} - 1$ nodes

Proof:

We will use mathematical induction:

1. Show that it is true for $h = 0$
2. Assume it is true for an arbitrary h
3. Show that the truth for h implies the truth for $h + 1$

$2^{h+1} - 1$ Nodes

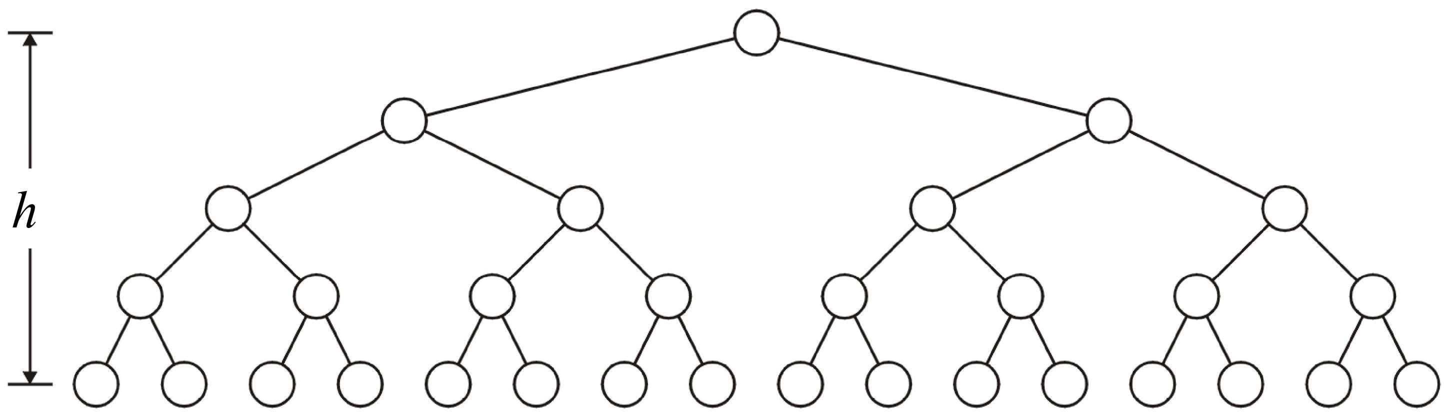
The base case:

- When $h = 0$ we have a single node $n = 1$
- The formula is correct: $2^{0+1} - 1 = 1$

$2^{h+1} - 1$ Nodes

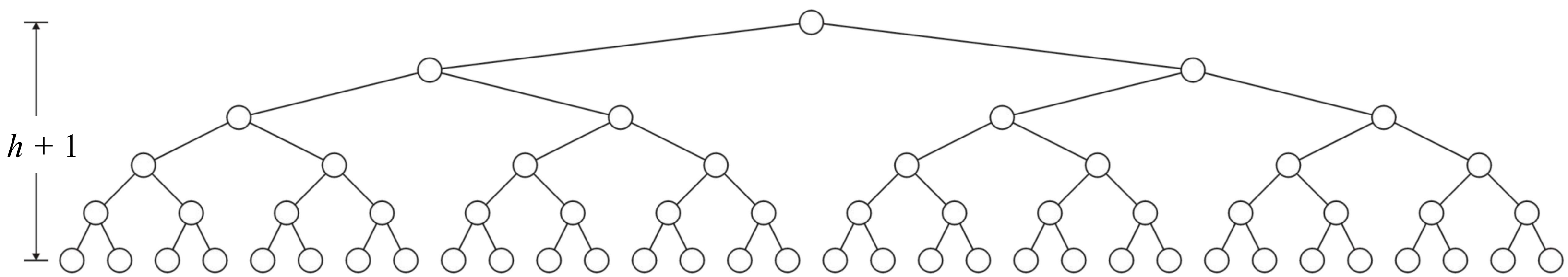
The inductive step:

- If the height of the tree is h, then assume that the number of nodes is $n = 2^{h+1} - 1$



$2^{h+1} - 1$ Nodes

We must show that a tree of height $h + 1$ has
$n = 2^{(h+1)+1} - 1 = 2^{h+2} - 1$ nodes

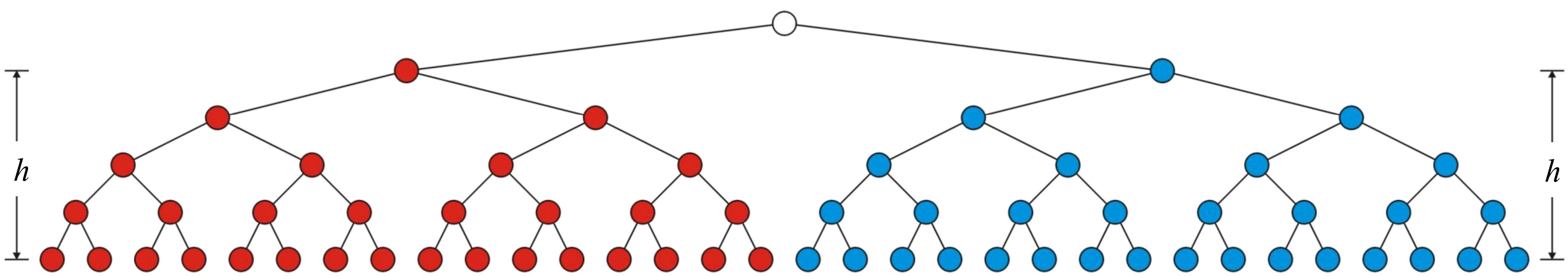


$2^{h+1} - 1$ Nodes

Using the recursive definition, both sub-trees are perfect trees of height h

- By assumption, each sub-tree has $2^{h+1} - 1$ nodes
- Therefore the total number of nodes is

$$(2^{h+1} - 1) + 1 + (2^{h+1} - 1) = 2^{h+2} - 1$$



$2^{h+1} - 1$ Nodes

Consequently

The statement is true for $h = 0$ and the truth of the statement for an arbitrary h implies the truth of the statement for $h + 1$.

Therefore, by the process of mathematical induction, the statement is true for all $h \geq 0$

Logarithmic Height

Theorem

A perfect binary tree with n nodes has height $\lg(n + 1) - 1$

Proof

Solving $n = 2^{h+1} - 1$ for h:

$$n + 1 = 2^{h+1}$$
$$\lg(n + 1) = h + 1$$
$$h = \lg(n + 1) - 1$$

Lemma

$$\lg(n + 1) - 1 = \Theta(\ln(n))$$

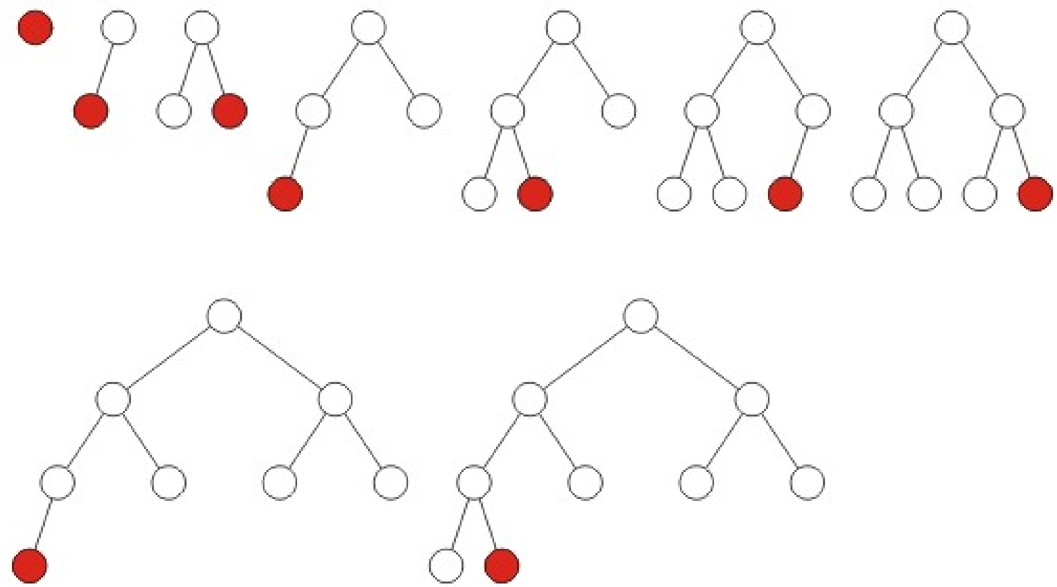
So, the height is $\Theta(\ln(n))$

Complete binary trees

Definition

A complete binary tree filled at each depth from left to right:

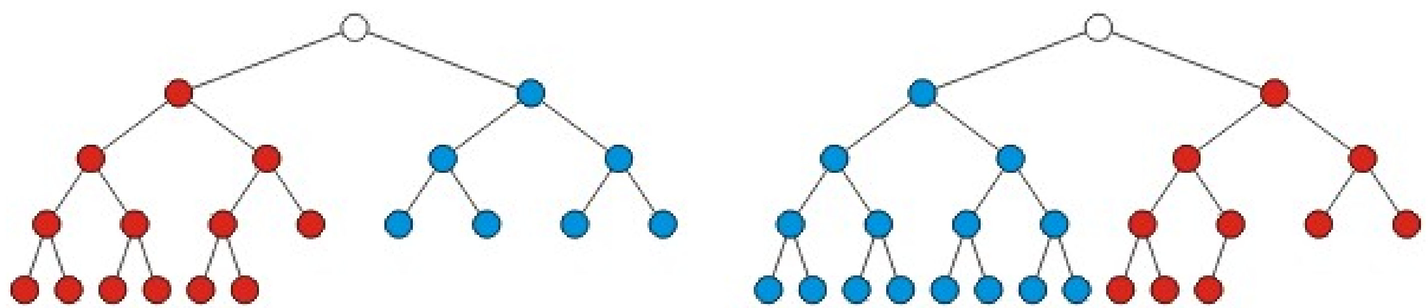
The order is identical to that of a breadth-first traversal



Recursive Definition

Recursive definition: a binary tree with a single node is a complete binary tree of height $h = 0$ and a complete binary tree of height h is a tree where either:

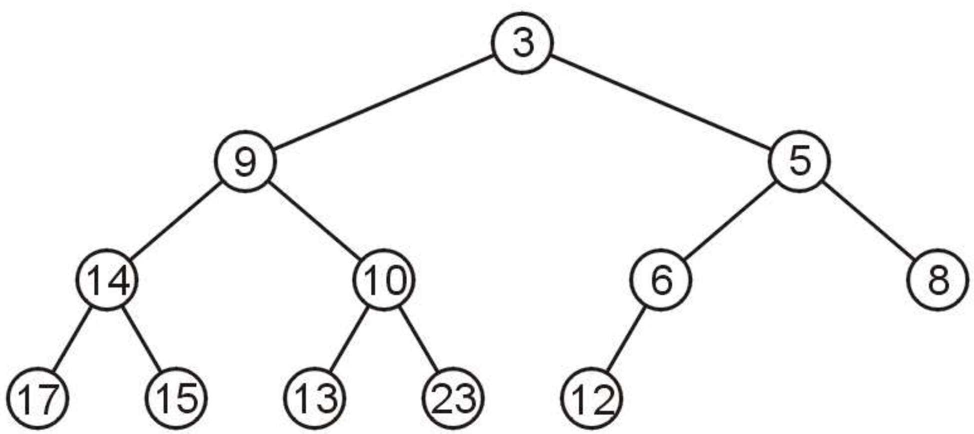
- The left sub-tree is a **complete tree** of height $h - 1$ and the right sub-tree is a **perfect tree** of height $h - 2$, or
- The left sub-tree is **perfect tree** with height $h - 1$ and the right sub-tree is **complete tree** with height $h - 1$



Array storage

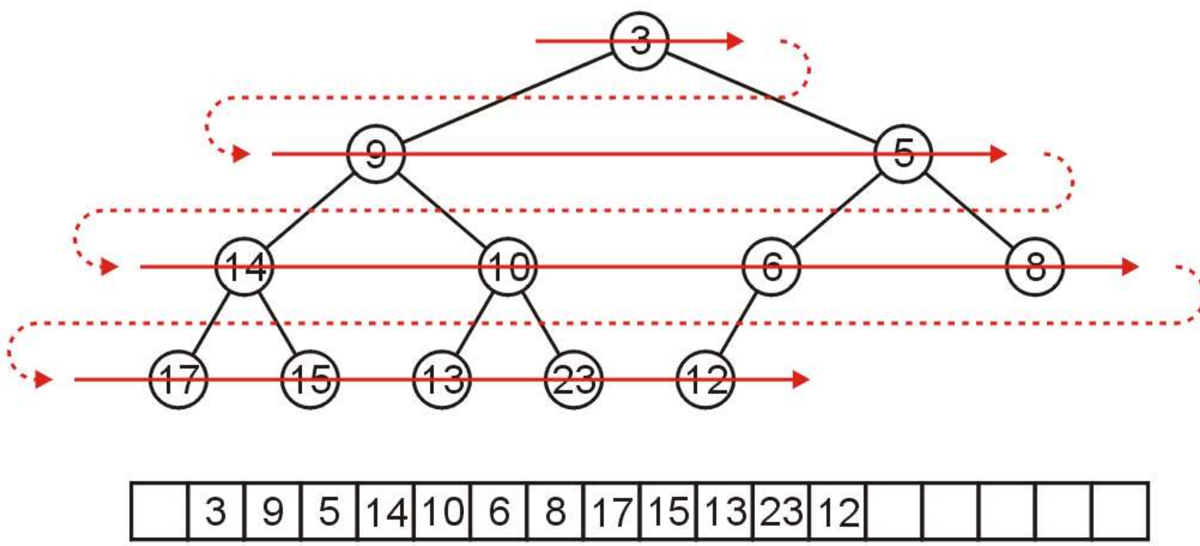
We are able to store a complete tree as an array

- Traverse the tree in breadth-first order, placing the entries into the array



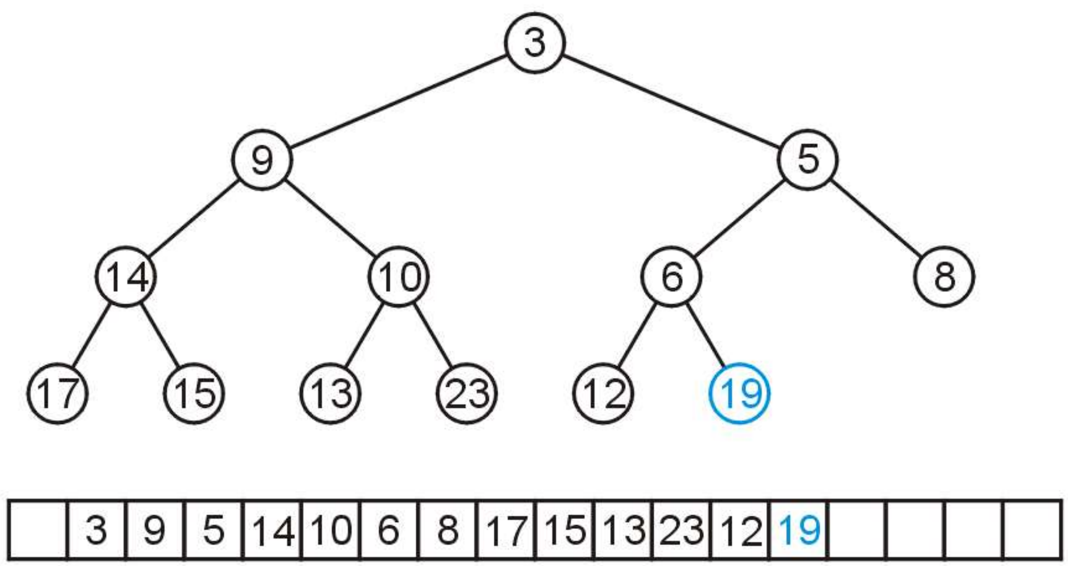
Array storage

We can store this in an array after a quick traversal:



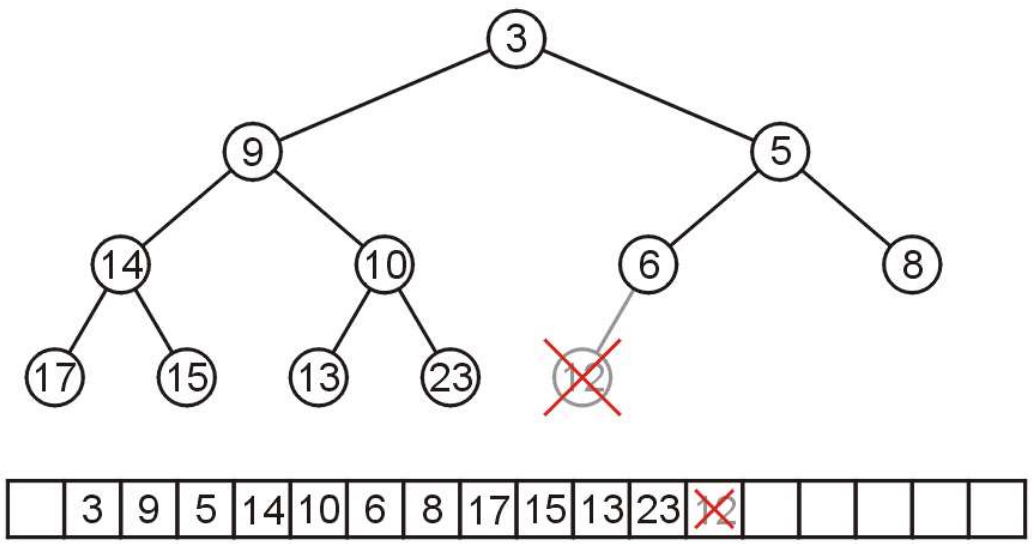
Array storage

To insert another node while maintaining the complete-binary-tree structure, we must insert into the next array location



Array storage

To remove a node while keeping the complete-tree structure, we must remove the last element in the array

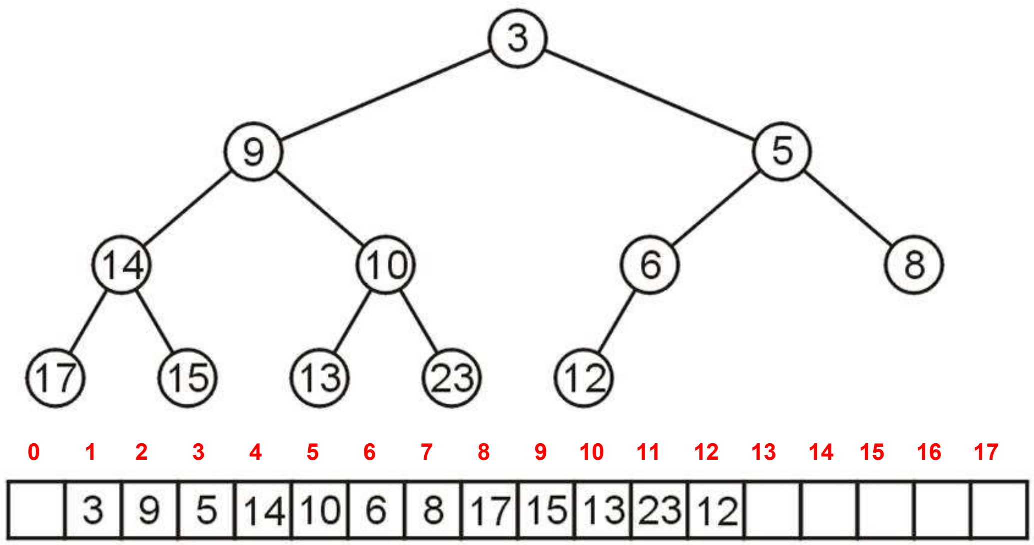


Array storage

Leaving the first entry blank yields a bonus:

– The children of the node with index k are in $2k$ and $2k + 1$

– The parent of node with index k is in $k \div 2$



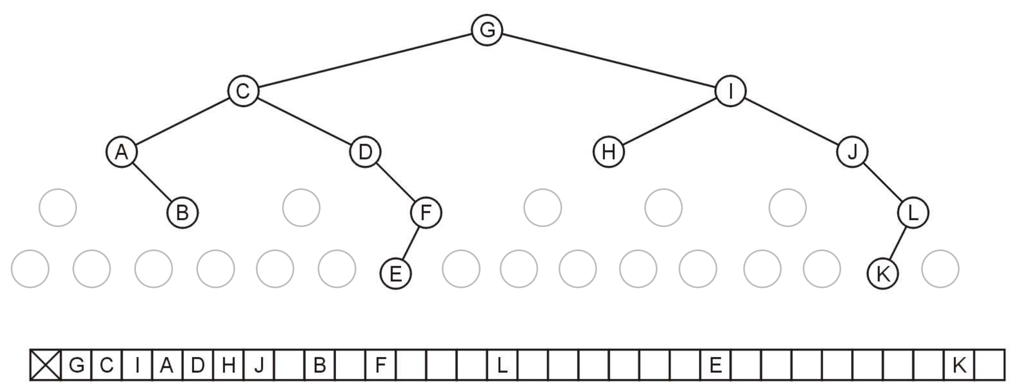
Array storage

Question: why not store any tree as an array using breadth-first traversals?

- There is a significant potential for a lot of wasted memory

Consider this tree with 12 nodes would require an array of size 32

- Adding a child to node K doubles the required memory



Array storage

In the worst case, an exponential amount of memory is required

These nodes would be stored in entries 1, 3, 6, 13, 26, 52, 105