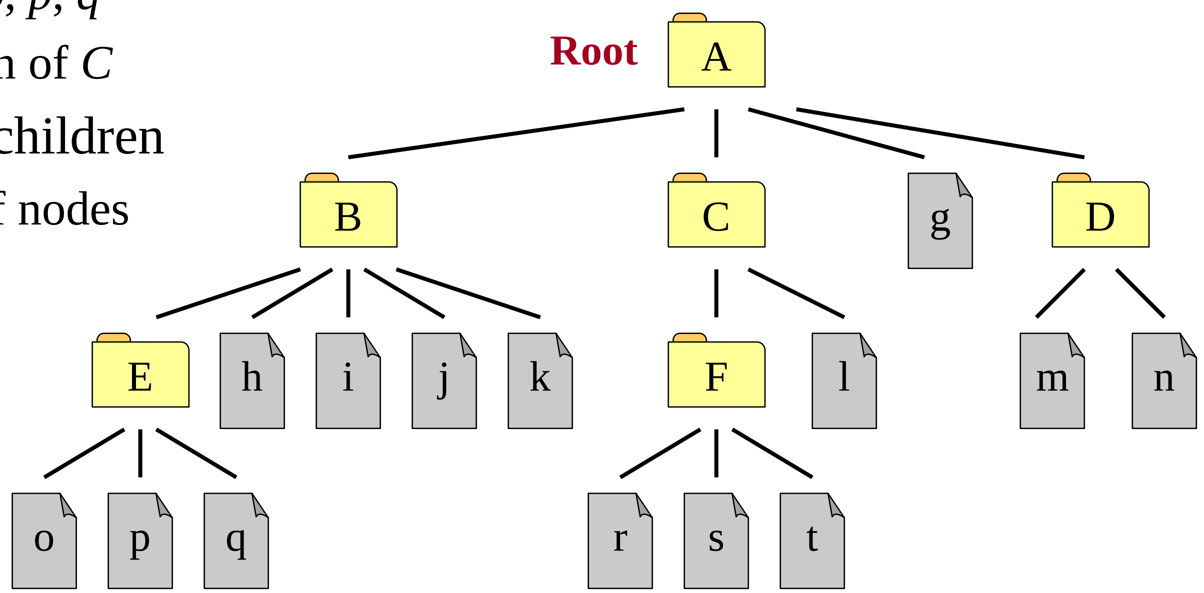
# Tree Concepts: Root, Parent, Child, and Leaf

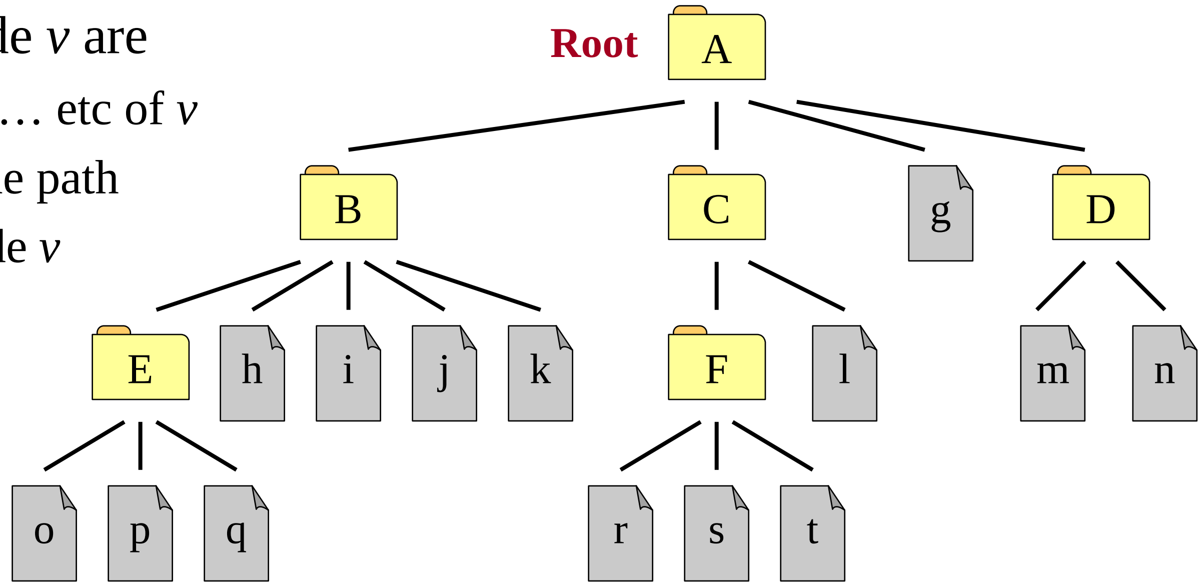
- A tree provides a way of storing **hierarchical data**
  - A **node** or a **vertex** is a container of data
  - The top node of the tree is identified as the **root node**
  - An **edge** is a link from a **parent node** to a successor node, called **child node**
    - *B* is the parent of *E*, *h*, *i*, *j*, and *k*
    - *E* is the parent of *o*, *p*, *q*
    - *F* and *l* are children of *C*
  - A **leaf node** has no children
    - *g* through *t* are leaf nodes




An example of a tree is a **hierarchical file system** consisting of **directories** and **files**

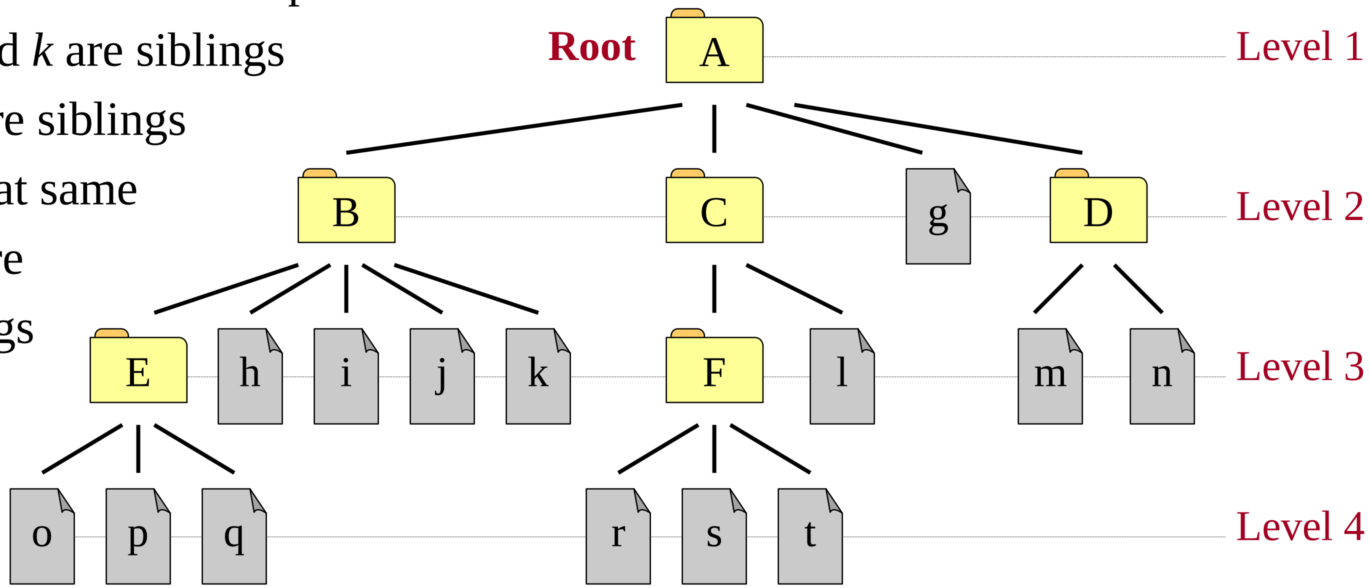
# Tree Concepts: Path, Descendants, and Ancestors

- A **path** from node $v_0$ to $v_n$ is a sequence of nodes
  - $v_0, v_1, v_2, \ldots, v_{n-1}, v_n$ , where there is an edge from one node to the next
  - Path from *A* to *r* is: *A*, *C*, *F*, *r*
- The **descendants** of a node *v* are
  - Children, grand children, grand grand children, … etc of *v*
  - All nodes reached by a path from node *v* to the leaf nodes
  - The descendants of *C* are: *F*, *l*, *r*, *s*, and *t*
- The **ancestors** of a node *v* are
  - Parent, grand parent, … etc of *v*
  - All nodes found on the path from root node to node *v*
  - Ancestors of *p* are *E*, *B*, and *A*



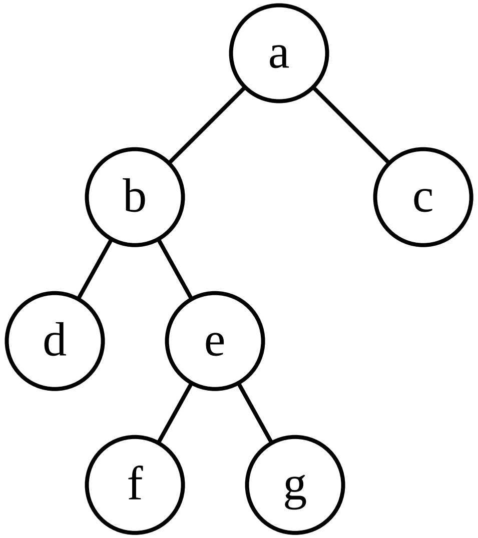
# Tree Concepts: Level, Height, and Siblings

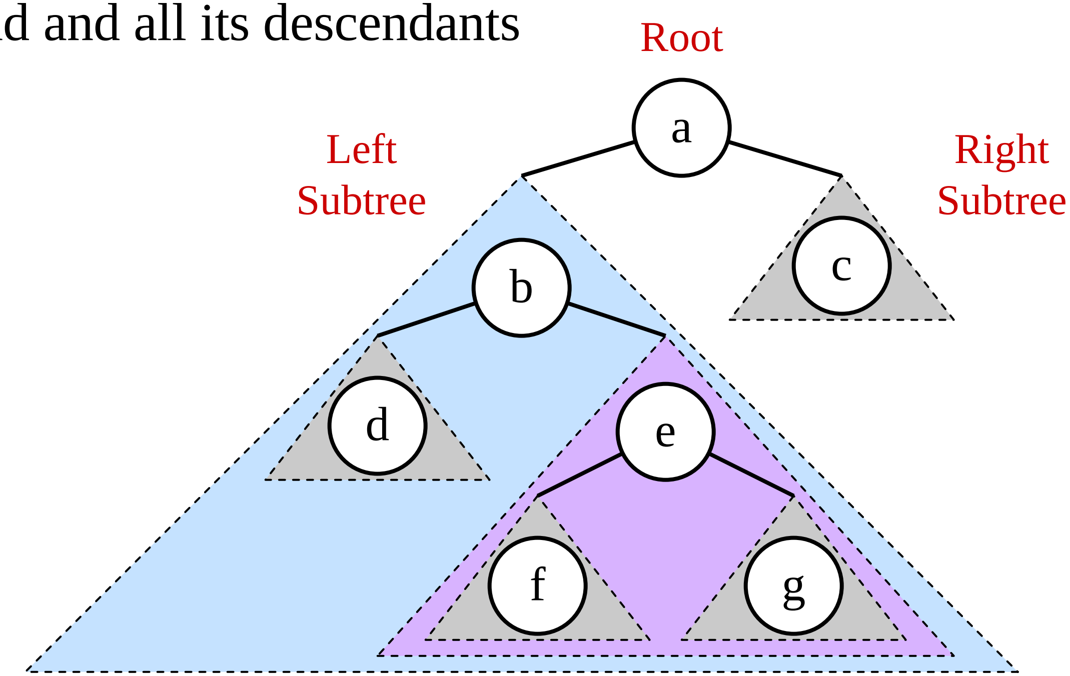
- The **level** of a node *v* is the number of vertices in the path from root to *v*
  - Root node *A* is at level 1
  - Leaf nodes *o*, *p*, *q*, *r*, *s*, and *t* are at level 4
- The **height** of a tree is the maximum level
  - An empty tree has height 0
- Nodes are called **siblings** iff
  - They are children of same parent
  - *E*, *h*, *i*, *j*, and *k* are siblings
  - *r*, *s*, and *t* are siblings
  - *q* and *r* are at same level, but are NOT siblings



# Binary Trees

- A **binary tree** is a tree in which
  - Every node has at most 2 children, called the **left child** and **right child**
- A binary tree can be defined recursively as
  - **Root node**
  - **Left subtree:** left child and all its descendants
  - **Right subtree:** right child and all its descendants



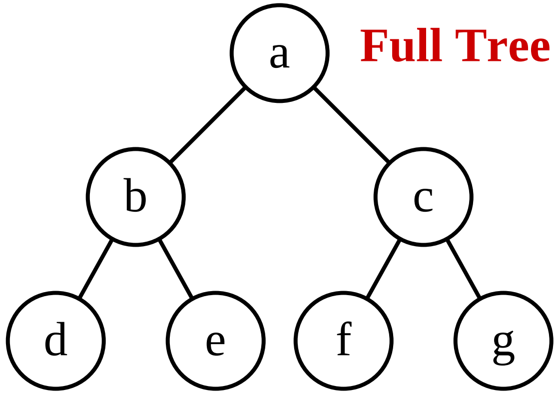



# Full and Complete Binary Trees

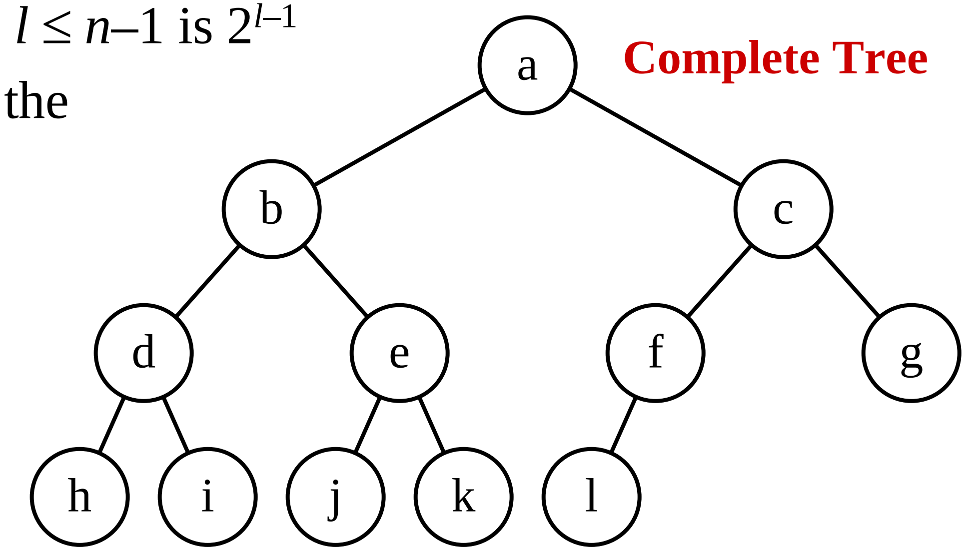
❖ A **full tree** is a binary tree in which
  - Number of nodes at level $l$ is $2^{l-1}$

❖ Total nodes in a full tree of height $n$ is

$$\sum_{l=1}^{n} 2^{l-1} = \sum_{l=0}^{n-1} 2^{l} = 2^{n} - 1$$

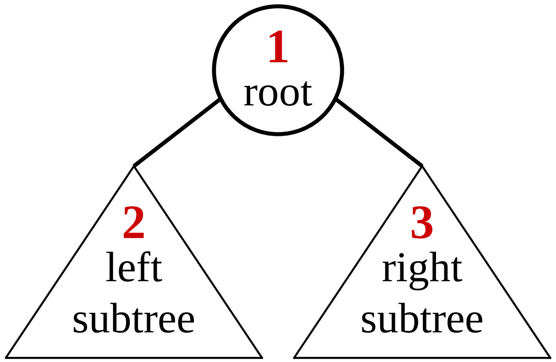



❖ A **complete tree** of height $n$ is a binary tree
  - Number of nodes at level $1 \le l \le n-1$ is $2^{l-1}$
  - Leaf nodes at level $n$ occupy the leftmost positions in the tree



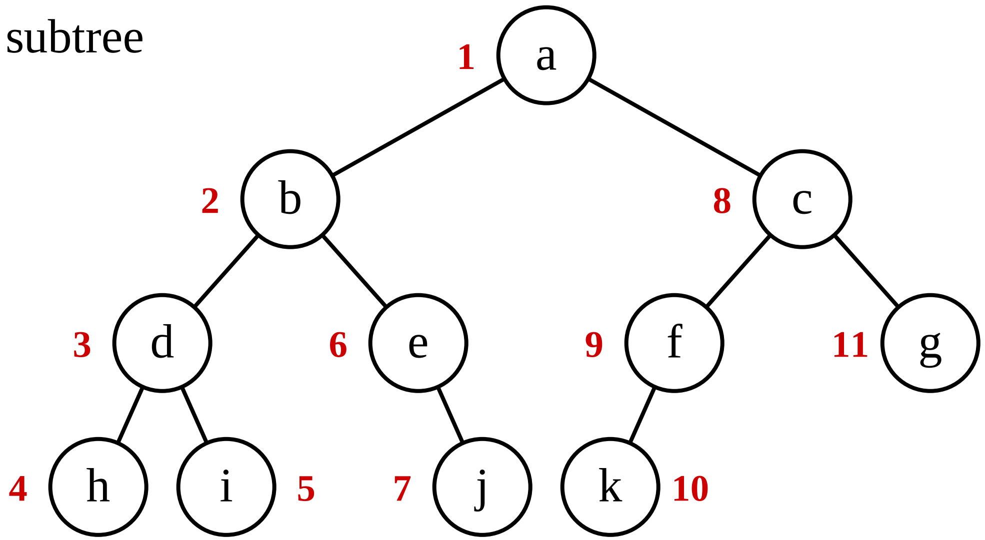
# Binary Tree Traversal: Preorder Traversal

- To **traverse a tree** means to …
  - Visit all the tree nodes and perform a task at each node
  - The order in which we visit nodes depends on the traversal algorithm
- In a **preorder traversal** …
  - If the tree is not empty
    1. Visit the root node
    2. Recursively, traverse the left subtree
    3. Recursively, traverse the right subtree



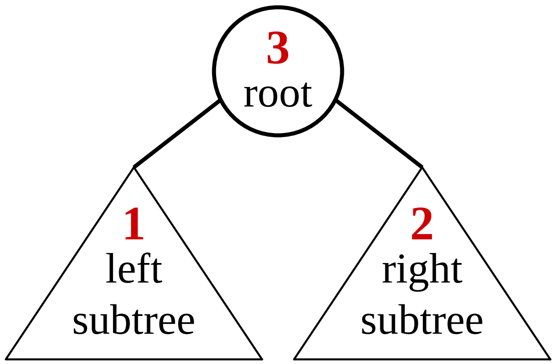

**Example of a preorder traversal**

Nodes are visited in this order:
*a*, *b*, *d*, *h*, *i*, *e*, *j*, *c*, *f*, *k*, *g*



# Postorder Traversal

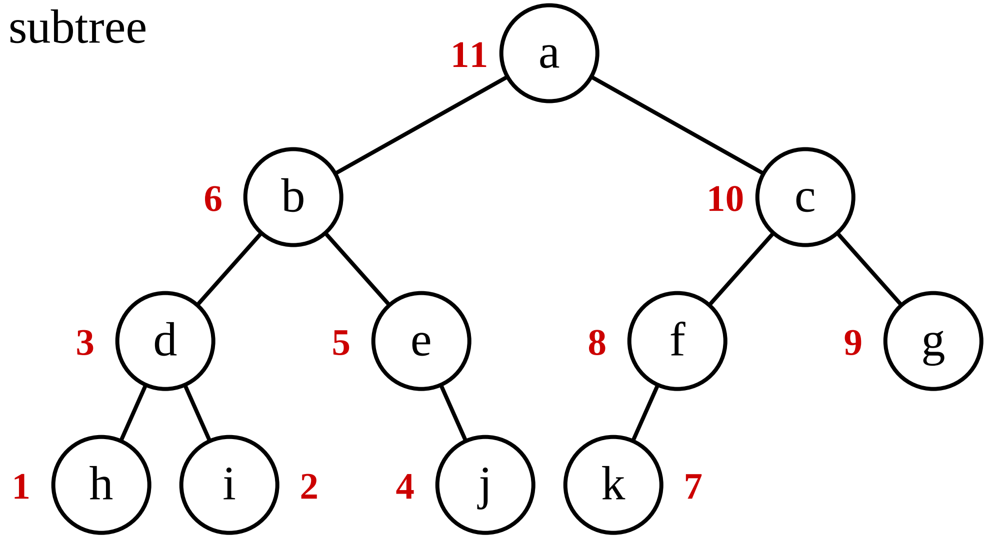
- If a **postorder traversal**, root is visited after visiting the subtrees
- **Postorder traversal** algorithm:
  - If the tree is not empty
    1. Recursively, traverse the left subtree
    2. Recursively, traverse the right subtree
    3. Visit the root node




**Example of a postorder traversal**

Nodes are visited in this order:

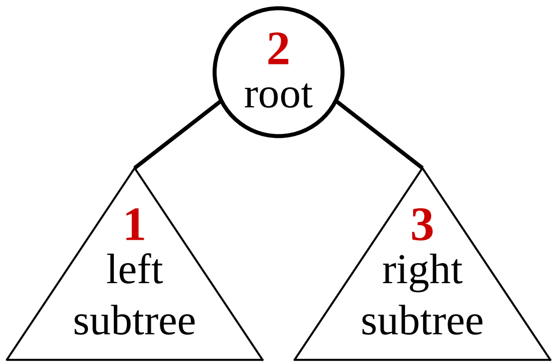
*h, i, d, j, e, b, k, f, g, c, a*




- **Preorder** is a **top-down depth-first** traversal of the tree
- **Postorder** is a **bottom-up** traversal of the tree

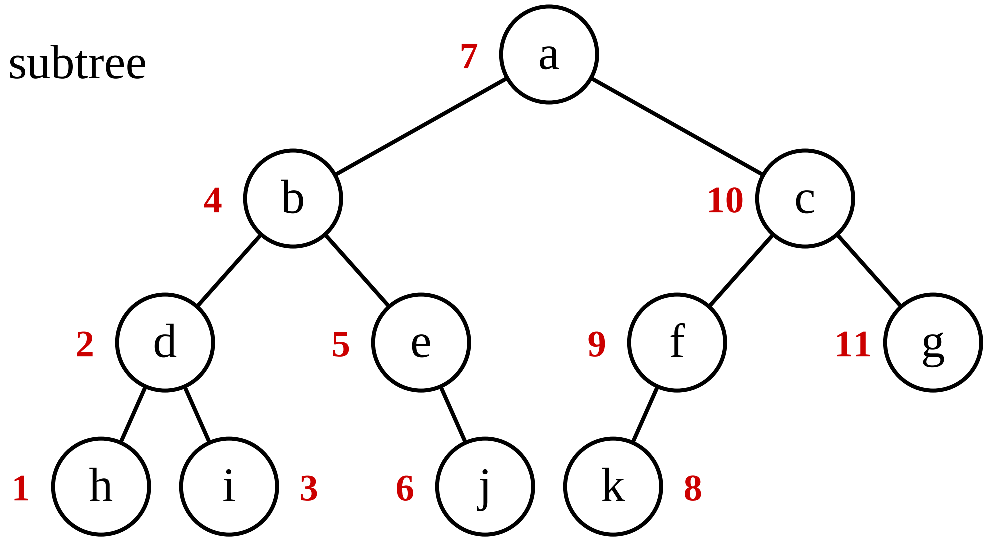
# Inorder Traversal

- **Inorder traversal** algorithm:
  - If the tree is not empty
    1. Recursively, traverse the left subtree
    2. Visit the root node
    3. Recursively, traverse the right subtree




**Example of an inorder traversal**

Nodes are visited in this order:
*h, d, i, b, e, j, a, k, f, c, g*

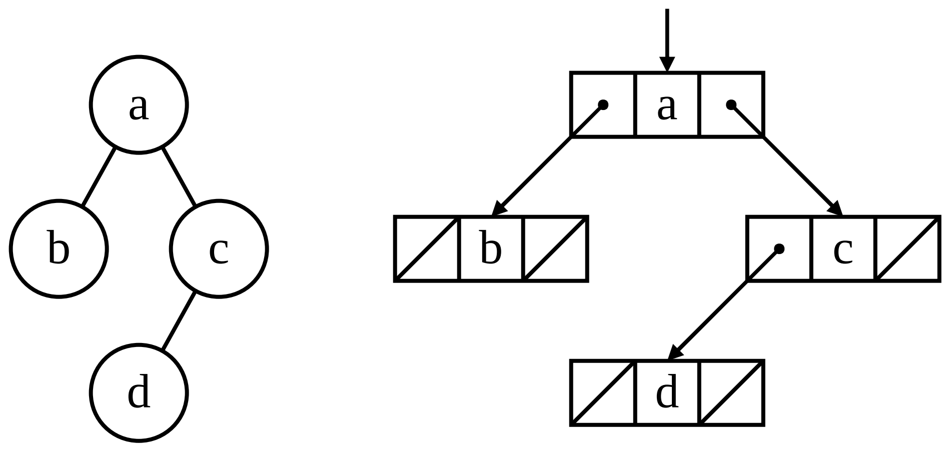



- The tree traversal algorithms are the basis for many applications
  - Provide an orderly access to the nodes and their data values

# Tree Node Structure

- A binary tree is built with nodes, much like a linked list
- A binary tree node contains a **data field** and **two pointers**
  - The **left pointer** points to the left subtree
  - The **right pointer** points to the right subtree
- The root node is the entry point into the binary tree
- A leaf node has a NULL left and right pointers

```
class TreeNode {
private:
  TreeNode* leftptr;
  TreeNode* rightptr;
public:
  DataType data;
  // Public Operations
};
```



# Tree Node Abstraction: Construction and Destruction

## Structure:

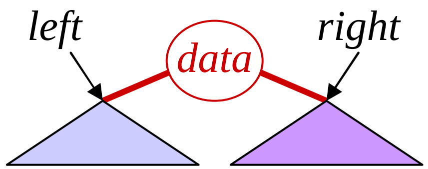
A tree node consists of data and two links to left and right subtrees

## Operations:

*Constructor* (*data*)

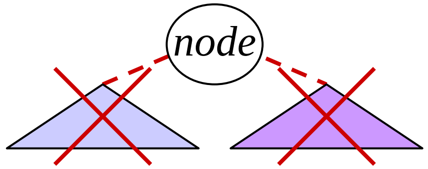

Purpose: Construct a node to contain *data* with NULL left and right pointers
Input: *data* value to be stored

*Constructor* (*data*, *left*, *right*)




Purpose: Construct a node to contain *data*, and link *left* and *right* subtrees
Input: *data* value, and pointers to *left* and *right* subtrees
Post: Binary tree is constructed from *left* and *right* subtrees

*Destructor* ( )




Purpose: Delete left and right subtrees of this tree node
Post: Tree node links to NULL left and right subtrees

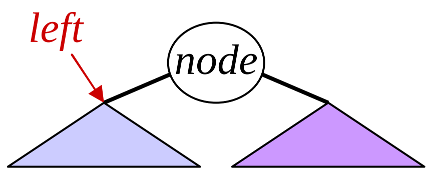
*isLeaf* ( ) → *bool*

Purpose: Checks whether tree node is a leaf node
Result: True if left and right pointers are NULL and false otherwise

# Tree Node Abstraction: Attach and Detach

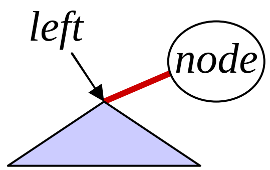
## Operations:

*Left* ( ) → *TreeNodePtr*



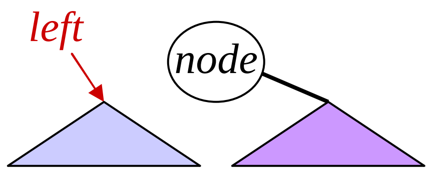

| | |
|---|---|
| Purpose: | Access the left subtree of this tree node |
| Result: | Pointer to left subtree |
| Post: | Tree node is NOT modified |

*AttachLeft* (*left*) → *bool*




| | |
|---|---|
| Purpose: | Attach a left subtree to this tree node |
| Pre: | Left subtree is initially Empty |
| Input: | Pointer to left subtree to be attached |
| Result: | True if operation is successful and false otherwise |
| Post: | *left* subtree is attached to this tree node |

*DetachLeft* ( ) → *TreeNodePtr*



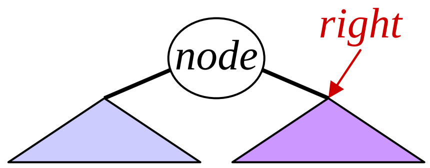

| | |
|---|---|
| Purpose: | Detach left subtree of this tree node |
| Result: | Pointer to detached left subtree |
| Post: | Tree node has an Empty left subtree |

# Tree Node Abstraction: Attach and Detach - cont'd

## Operations:

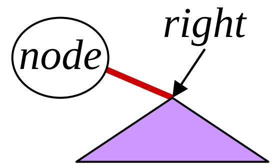
*Right* () → *TreeNodePtr*

| | |
|---|---|
| Purpose: | Access the right subtree of this tree node |
| Result: | Pointer to right subtree |
| Post: | Tree node is NOT modified |



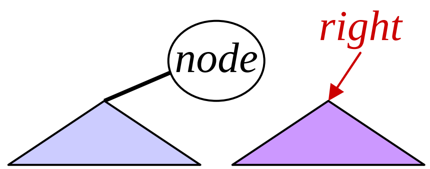

*AttachRight* (*right*) → *bool*

| | |
|---|---|
| Purpose: | Attach a right subtree to this tree node |
| Pre: | Right subtree is initially Empty |
| Input: | Pointer to right subtree to be attached |
| Result: | True if operation is successful and false otherwise |
| Post: | *right* subtree is attached to this tree node |




*DetachRight* ( ) → *TreeNodePtr*

| | |
|---|---|
| Purpose: | Detach right subtree of this tree node |
| Result: | Pointer to detached right subtree |
| Post: | Tree node has an Empty right subtree |

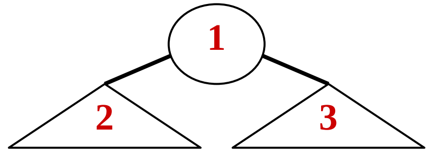


# Tree Node Abstraction: Traversals and Copy

## Operations:

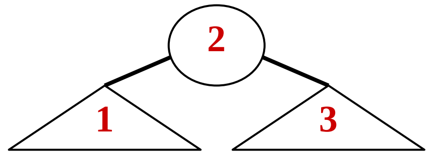
*PreOrder* ( )

Purpose: Pre-order traversal of a tree
Post: Nodes are visited and processed according to pre-order



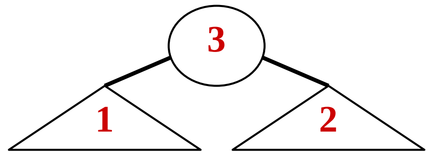

*InOrder* ( )

Purpose: Post-order traversal of a tree
Post: Nodes are visited and processed according to in-order



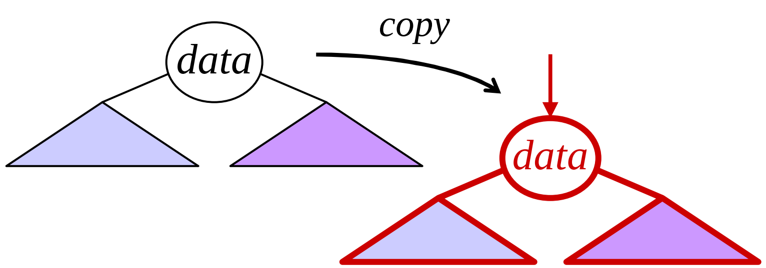

*PostOrder* ( )

Purpose: Post-order traversal of a tree
Post: Nodes are visited and processed according to post-order




*Copy* ( ) → *TreeNodePtr*

Purpose: Duplicate a tree node and its subtrees
Result: Pointer to the duplicated binary tree
Post: Original Binary tree is NOT modified



# Tree Node Class Specification

```
class TreeNode {
private:
  TreeNode* leftptr;                    // Pointer to left subtree
  TreeNode* rightptr;                   // Pointer to right subtree

public:
  DataType data;                        // Stored data

  TreeNode(const DataType& d,           // Construct a tree node with data
           TreeNode* l = 0,             // and optional pointers to
           TreeNode* r = 0);            // left and right subtrees

  ~TreeNode();                          // Delete left and right subtrees

  TreeNode* left() const;               // Return pointer to left subtree
  TreeNode* right() const;              // Return pointer to right subtree
  TreeNode* detachLeft();               // Detach and return left subtree
  TreeNode* detachRight();              // Detach and return right subtree
  bool attachLeft (TreeNode* l);        // Attach l as left subtree
  bool attachRight(TreeNode* r);        // Attach r as right subtree
  void preorder();                      // Preorder Traversal of tree nodes
  void inorder();                       // Inorder Traversal of tree nodes
  void postorder();                     // Postorder Traversal of tree nodes
  bool isLeaf() const;                  // Check whether node is a leaf node
  TreeNode* copy();                     // Duplicate nodes of this tree
};
```

# Tree Node Implementation

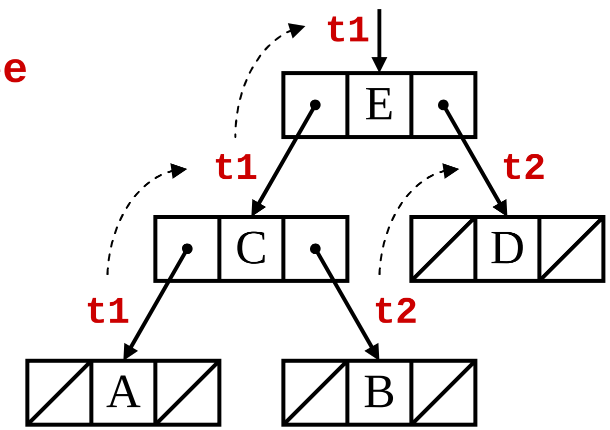
```
TreeNode::TreeNode(const DataType& d, TreeNode* l, TreeNode* r)
{
  data = d;
  leftptr = l;
  rightptr = r;
}

TreeNode::~TreeNode() {
  if (leftptr) {
    delete leftptr;
    leftptr = 0;
  }
  if (rightptr) {
    delete rightptr;
    rightptr = 0;
  }
}

bool TreeNode::isLeaf() const {
  return (leftptr == 0 && rightptr == 0);
}
```

*Example: Bottom up construction of a tree*

```
typedef char DataType;
. . .
TreeNode* t1 = new TreeNode(’A’);
TreeNode* t2 = new TreeNode(’B’);
t1 = new TreeNode(’C’, t1, t2);
t2 = new TreeNode(’D’);
t1 = new TreeNode(’E’, t1, t2);
. . .
delete t1;
// Entire tree
// is deleted
```



# Tree Node Implementation: Attach and Detach

```
TreeNode* TreeNode::left()  const { return leftptr; }
TreeNode* TreeNode::right() const { return rightptr; }

bool TreeNode::attachLeft(TreeNode* l) {
  if (leftptr) return false;  // Cannot attach if left subtree exists
  leftptr = l; return true;
}
bool TreeNode::attachRight(TreeNode* r) {
  if (rightptr) return false; // Cannot attach if right subtree exists
  rightptr = r; return true;
}

TreeNode* TreeNode::detachLeft() {
  TreeNode* l = leftptr;
  leftptr = 0;
  return l;
}
TreeNode* TreeNode::detachRight() {
  TreeNode* r = rightptr;
  rightptr = 0;
  return r;
}
```

*Example: Top down construction of a tree*

```
TreeNode* t1 = new TreeNode('A');
TreeNode* t2 = new TreeNode('B');
TreeNode* t3 = new TreeNode('C');
t1->attachLeft(t2);
t1->attachRight(t3);
t3 = new TreeNode('D');
t2->attachLeft(t3);
t3 = new TreeNode('E');
t2->attachRight(t3);
```

# Tree Node Implementation: Tree Traversals

```
void TreeNode::preorder() {
  process(data);                          // Access or modify data
  if (leftptr)  leftptr->preorder();      // Recursive call
  if (rightptr) rightptr->preorder();     // Recursive call
}

void TreeNode::inorder() {
  if (leftptr)  leftptr->inorder();       // Recursive call
  process(data);                          // Access or modify data
  if (rightptr) rightptr->inorder();      // Recursive call
}

void TreeNode::postorder() {
  if (leftptr)  leftptr->postorder();     // Recursive call
  if (rightptr) rightptr->postorder();    // Recursive call
  process(data);                          // Access or modify data
}
```

- Processing *data* can call any function that we may choose
- *data* can be read only or can be modified

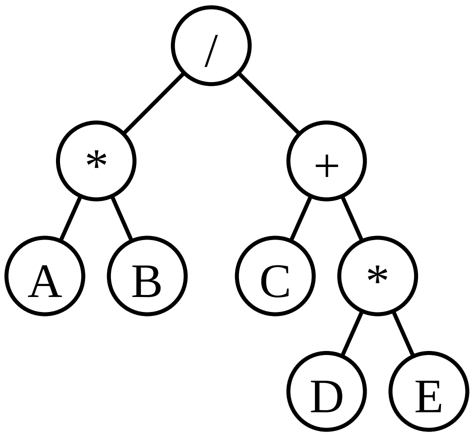
# Tree Node Implementation: Copy Tree

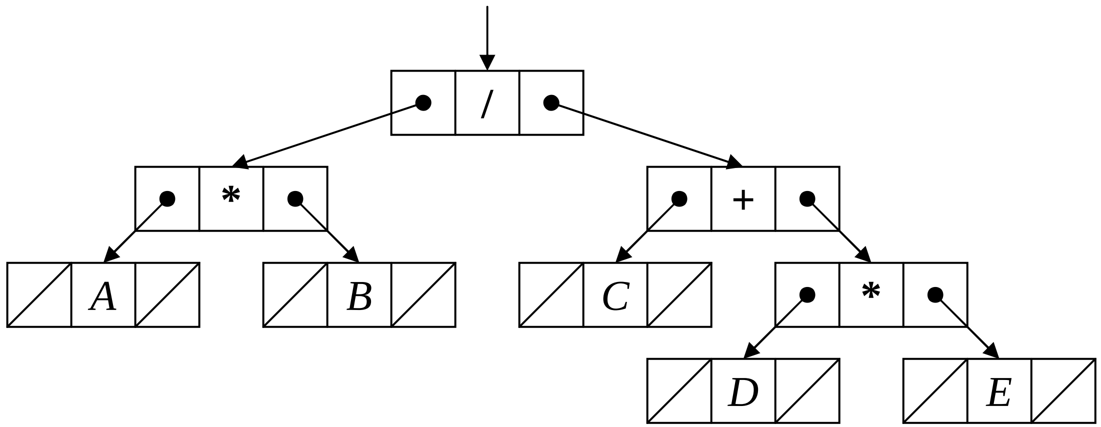
```
TreeNode* TreeNode::copy() {                // Pre-Order Recursive copying
  TreeNode* tree;
  tree = new TreeNode(data);                // Allocate node and copy data
  if (leftptr)                              // If left subtree exists
    tree->leftptr = leftptr->copy();        // Recursively, copy left tree
  if (rightptr)                             // If right subtree exists
    tree->rightptr = rightptr->copy();// Recursively, Copy right tree
  return tree;
}

TreeNode* TreeNode::copy() {                // Post-Order Recursive copying
  TreeNode *tree, *l=0, *r=0;
  if (leftptr)                              // If left subtree exists
    l = leftptr->copy();                    // Recursively, copy left tree
  if (rightptr)                             // If right subtree exists
    r = rightptr->copy();                   // Recursively, copy right tree
  tree = new TreeNode(data,l,r);            // Allocate new node, copy data
  return tree;                              // and link copies of subtrees
}
```

# Example on Binary Trees : Expression Trees

- An expression can be represented as a binary tree
  - Such a binary tree is called an **expression tree**
  - The operands are stored in the leaves of the tree
  - The operators are stored in the root and internal nodes
  - Each binary operator operates on two operands
    - The first operand is the left subtree
    - The second operand is the right subtree
- For example, $A * B / (C + D * E)$ is represented as such:



# Traversing an Expression Tree

- There are three major traversals of an expression tree
  - **Preorder Traversal**: generates **prefix notation** of expression
    - For example: / * *A B* + *C* * *D E* (no parentheses)
  - **Postorder Traversal**: generates **postfix notation** of expression
    - For example: *A B* * *C D E* * + / (no parentheses)
  - **Inorder Traversal**: generates **infix notation** of expression
    - For example: ((*A* * *B*) / (*C* + (*D* * *E*))) (fully parenthesized)