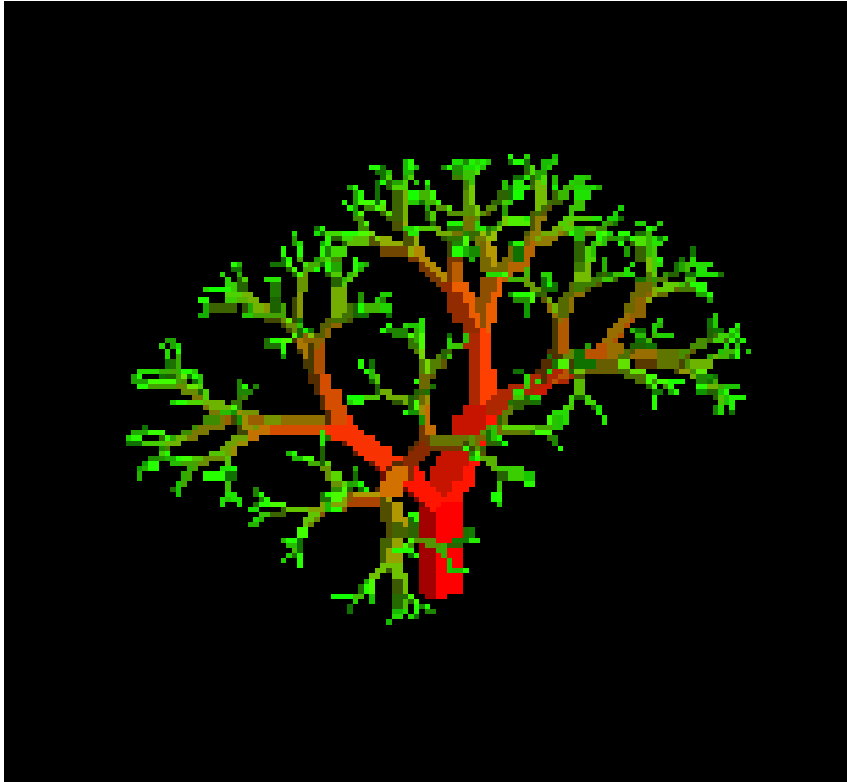


Unit II : Trees

Tree- basic terminology, General tree and its representation, representation using sequential and linked organization, Binary tree-properties, converting tree to binary tree, binary tree traversals(recursive and non-recursive)- inorder, preorder, post order, depth first and breadth first, Operations on binary tree. Huffman Tree (Concept and Use), Binary Search Tree (BST), BST operations, Threaded binary search tree-concepts, threading, insertion and deletion of nodes in inorder threaded binary search tree, in order traversal of in-order threaded binary search tree.

Case Study : Use of binary tree in expression tree-evaluation and Huffman's coding

How We View a Tree



Nature Lovers View



Computer Scientists View

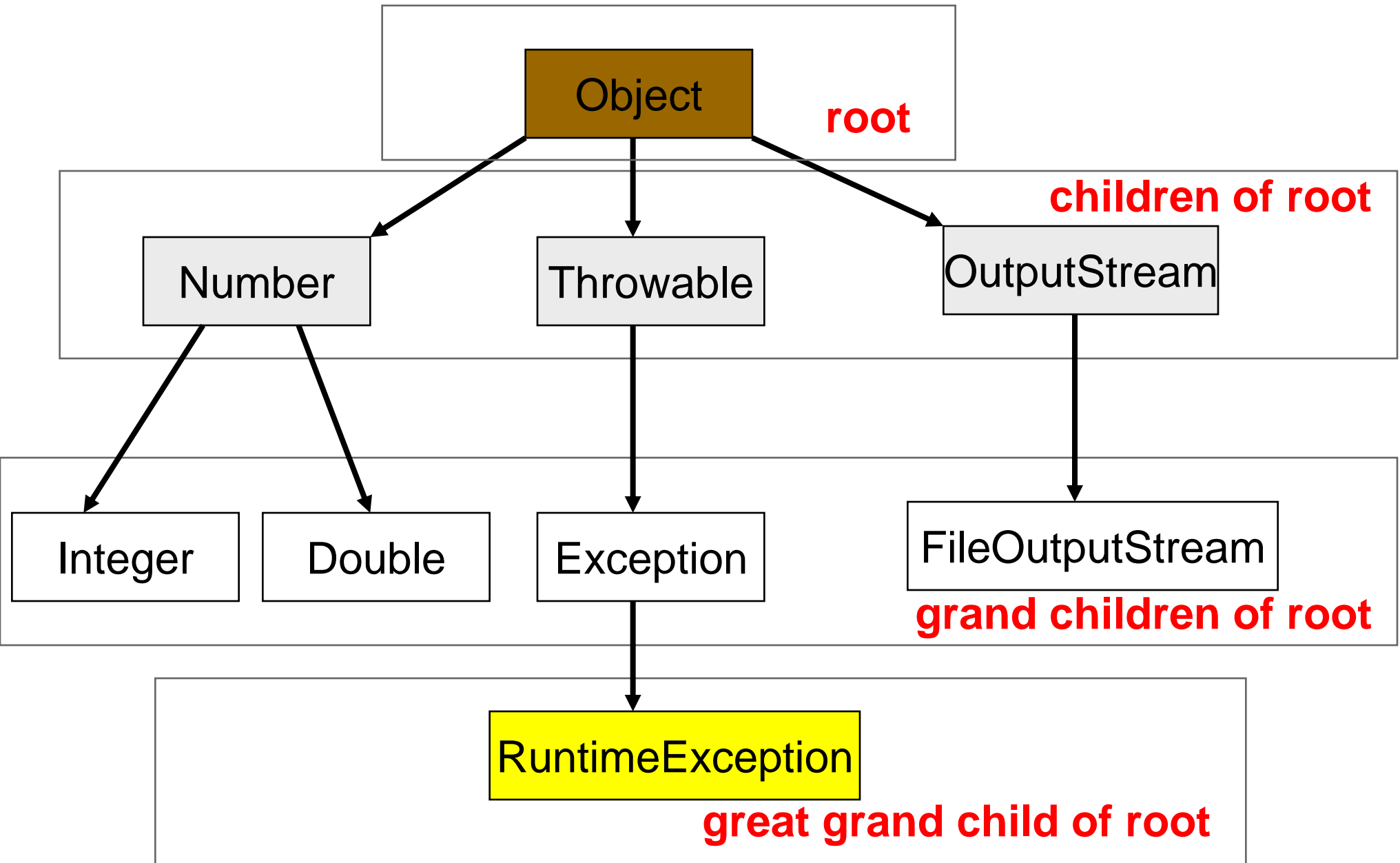
Linear Lists And Trees

- **Linear lists are useful for serially ordered data**
 - $(e_0, e_1, e_2, \dots, e_{n-1})$
 - Days of week, months in a year, students in this class
- **Trees are useful for hierarchically ordered data**
 - **Employees of a corporation**
 - President, vice presidents, managers, etc.
 - **Java's classes.**
 - Object is at the top of the hierarchy.
 - Subclasses of Object are next, etc.

Hierarchical Data And Trees

- The element at the top of the hierarchy is the **root**
- Elements next in the hierarchy are the **children** of the root
- Elements next in the hierarchy are the **grandchildren** of the root, and so on
- Elements at the lowest level of the hierarchy are the **leaves**

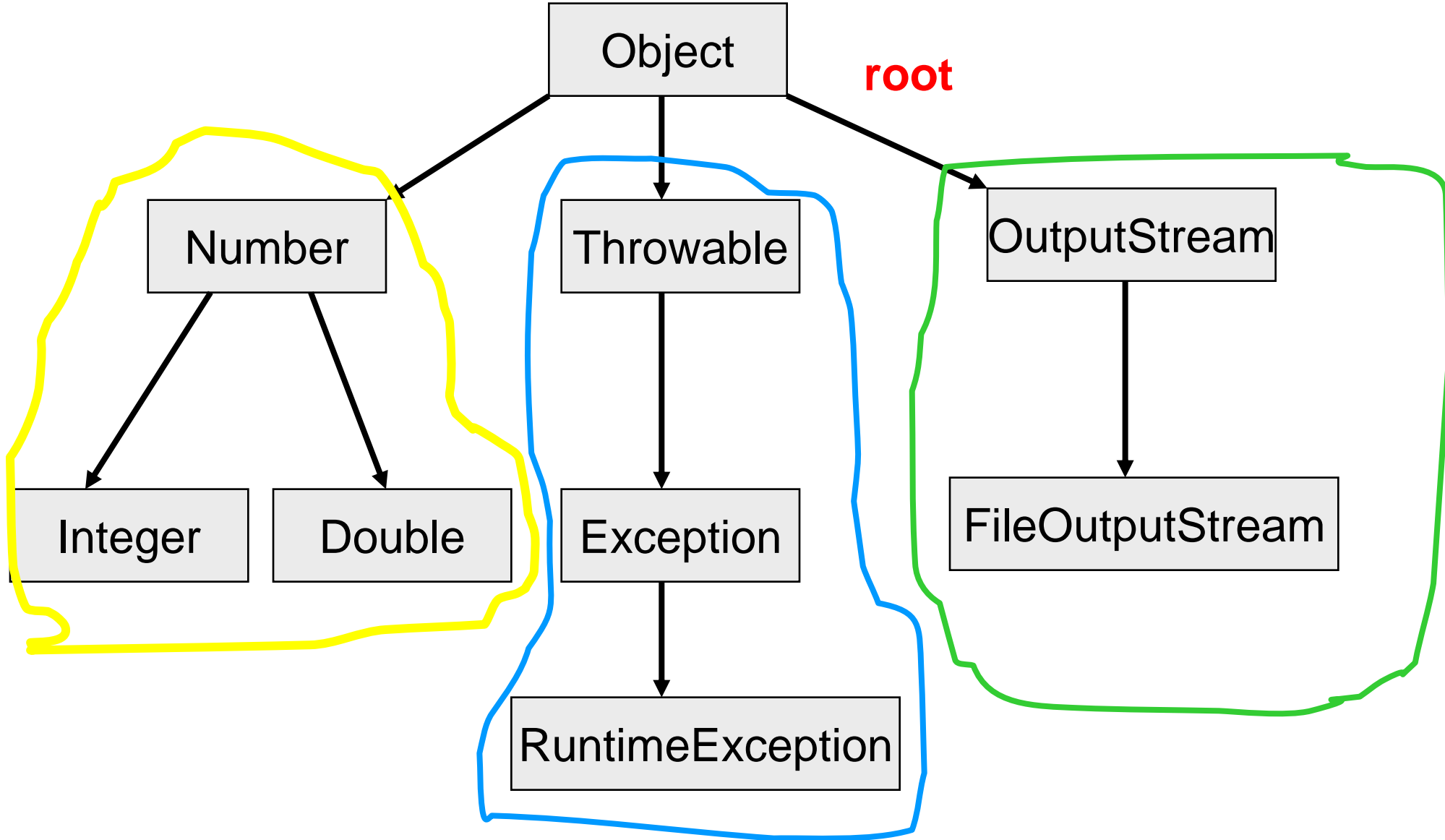
Java's Classes



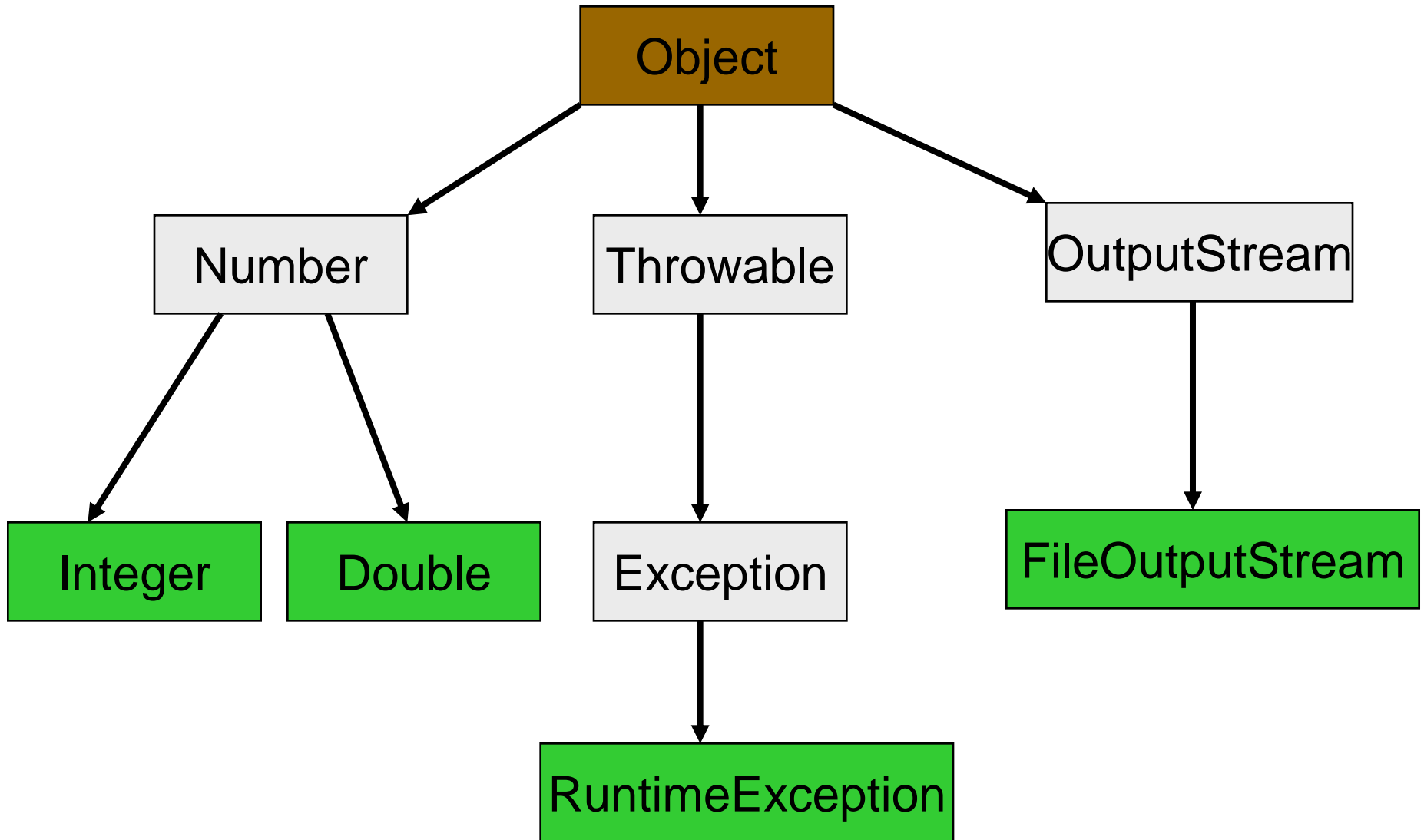
Tree Definition

- A tree t is a finite nonempty set of elements
- One of these elements is called the **root**
- The remaining elements, if any, are partitioned into trees, which are called the **subtrees** of t .
- A Tree is defined as a connected graph without a circuit.

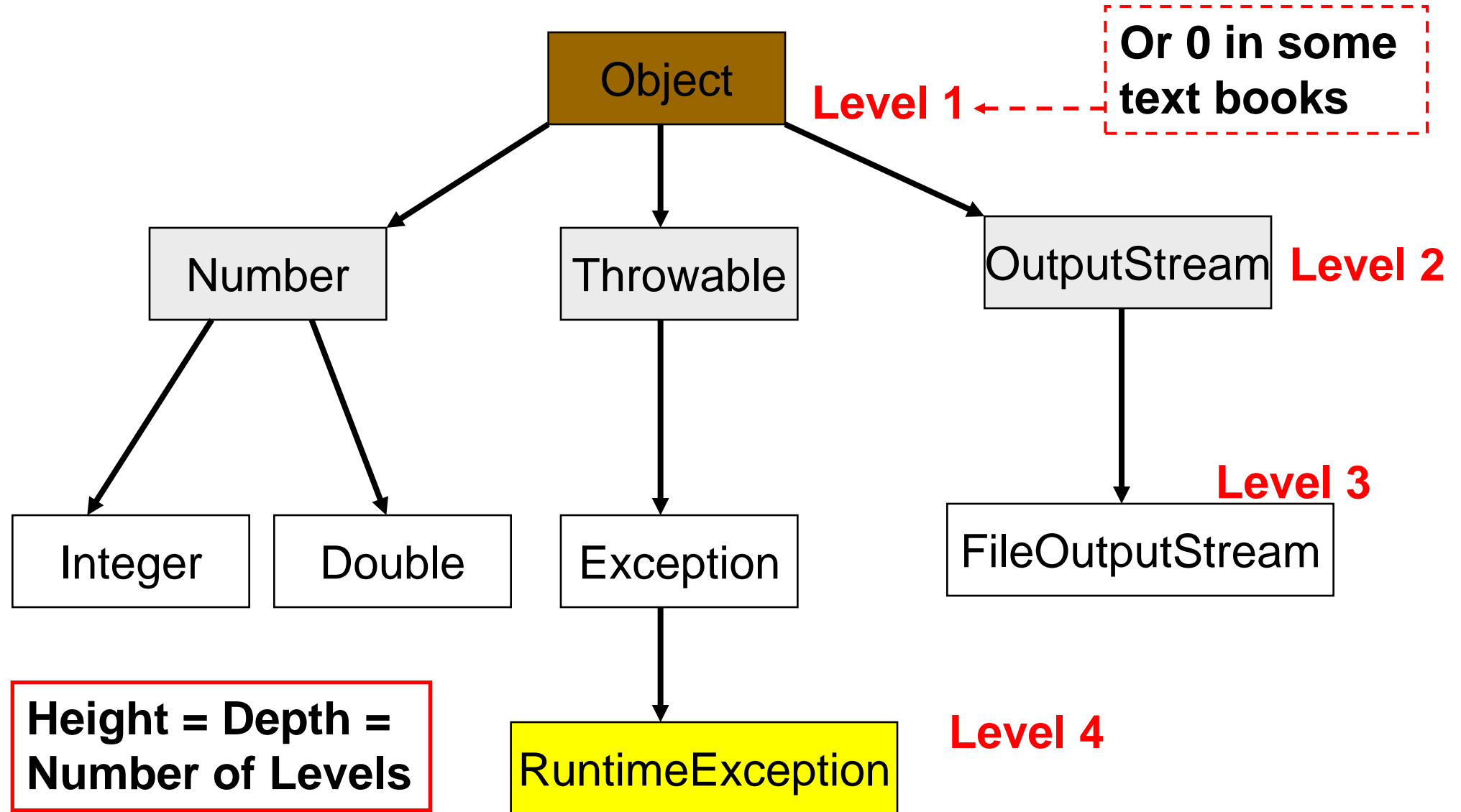
Subtrees



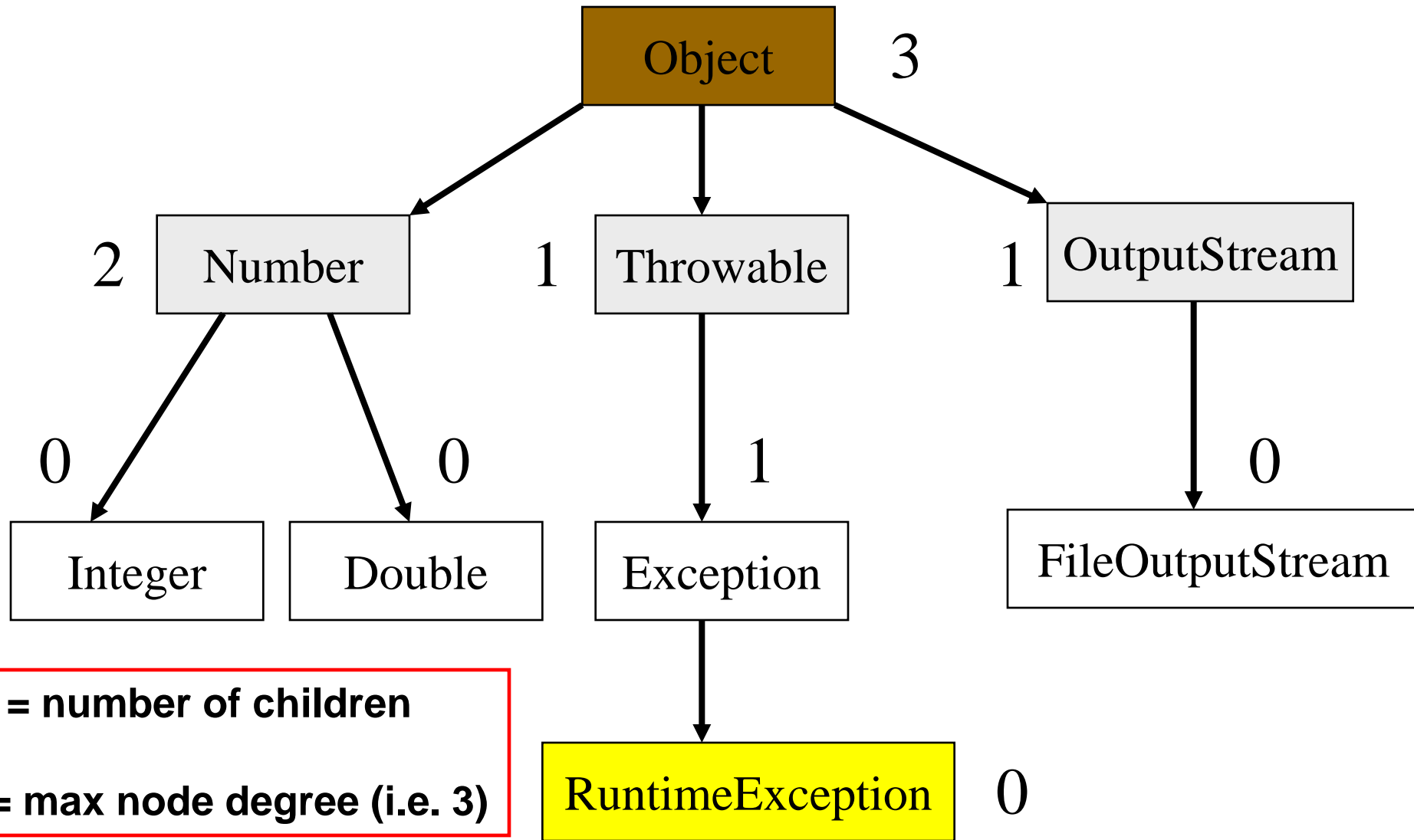
Leaves



Levels



Degree



○ Siblings

- Childrens of the same parent

○ Descendents

- All nodes reachable from that node

○ Ancestors

- All the nodes along the path from the root to that node

Types of Trees

- **Binary Tree**
- **Binary Search Tree**
- **Threaded binary Tree**
- **Expression Tree**
- **AVL Trees**
- **Optimal Binary Search Tree**
- **Red black Trees**
- **B Trees / B+ Trees**
- **Splay Trees**
- **Trie Trees**

Binary Tree

- **Binary Tree is a rooted tree in which root has maximum two children such that each of them again is binary tree.**
- **A tree in which one vertex is distinguished from others and called as ROOT is known as a Rooted Tree.**

Tree vs Binary Tree

- No node in a binary tree may have a degree more than **2**, whereas there is no limit on the degree of a node in a tree
- A binary tree may be empty; a tree cannot be empty

Tree vs Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered.



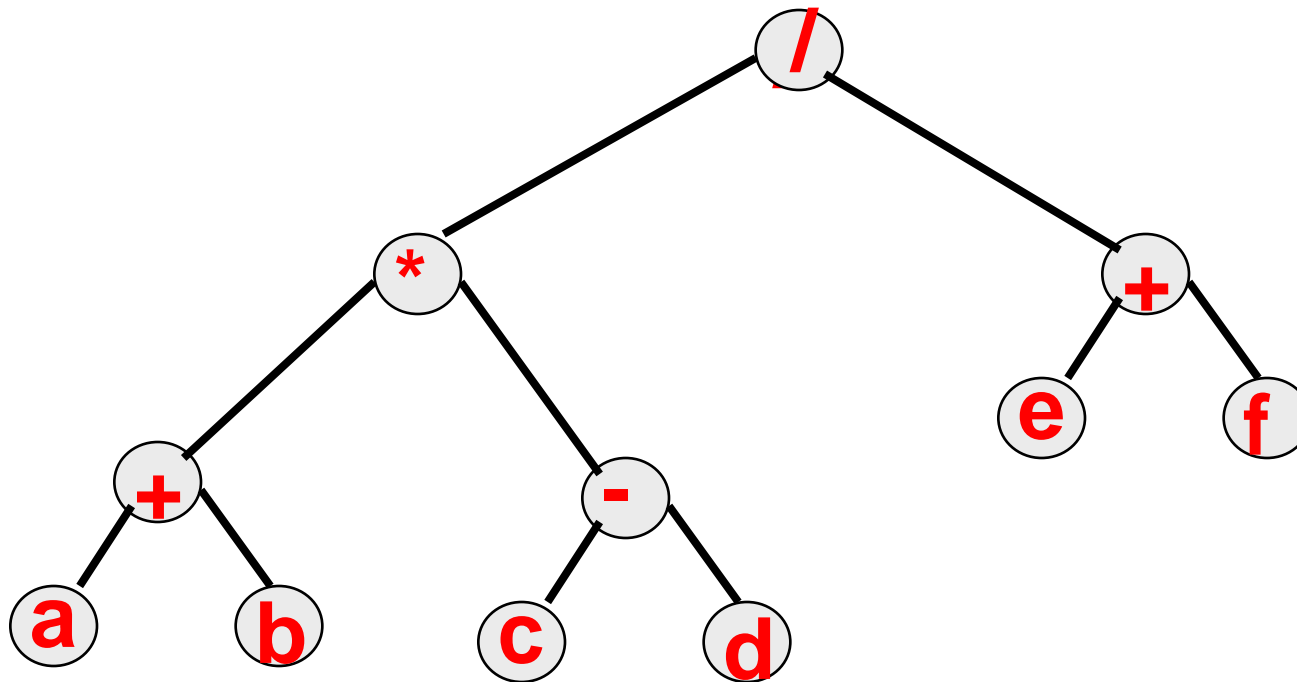
- Are different when viewed as binary trees
- Are the same when viewed as trees

Expression Tree

- Application of Binary Tree
- A Binary Expression tree can be used to represent an expression containing operands and binary operators. The tree will be a strictly binary tree
- The root contains the operators to be applied to the result obtained by evaluating the left and right subtrees.
- The operands are in the leaf nodes while the operators are in the internal nodes.
- Traversing the tree in preorder and postorder gives the prefix and postfix forms respectively of the expressions.
- $(a + b) * (c + d) + e - f/g * h + 3.25$
- Expressions comprise three kinds of entities.
 - Operators (+, -, /, *)
 - Operands (a, b, c, d, e, f, g, h, 3.25, (a + b), (c + d), etc.)
 - Delimiters ((,))

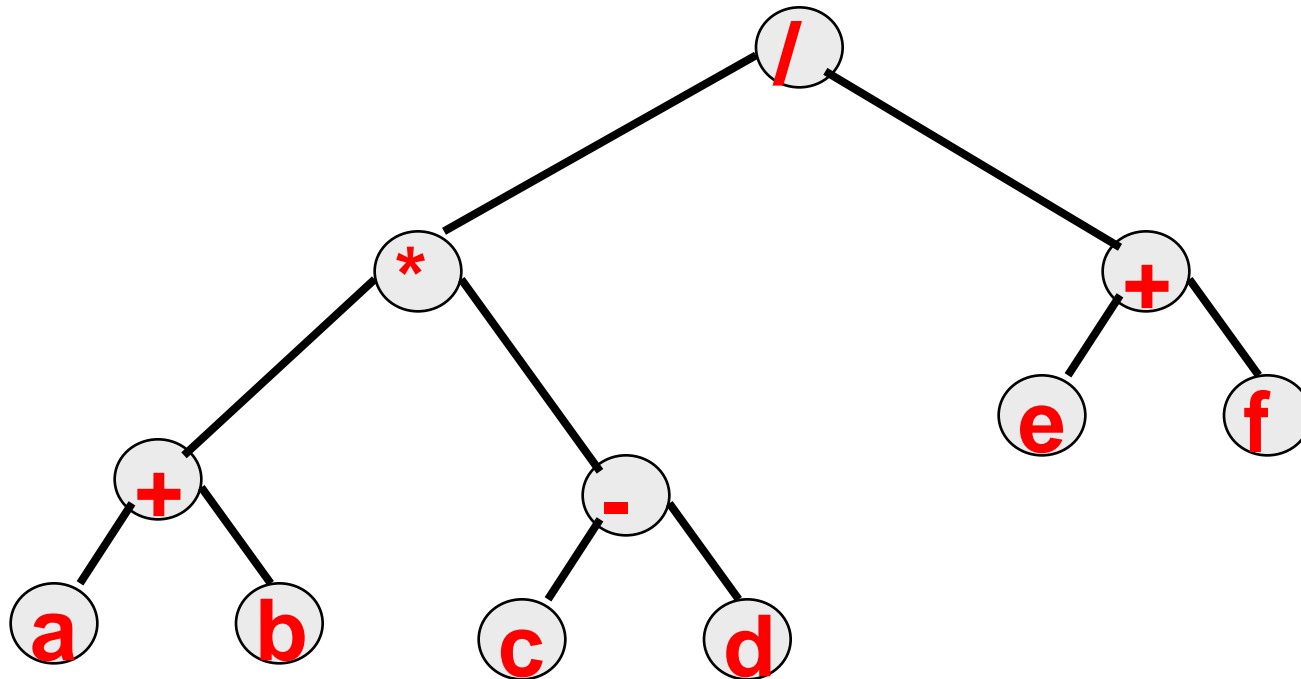
Binary Tree Form

○ $(a + b) * (c - d) / (e + f)$



Merits Of Binary Tree Form

- Left and right operands are easy to visualize
- Code optimization algorithms work with the binary tree form of an expression
- Simple recursive evaluation of expression

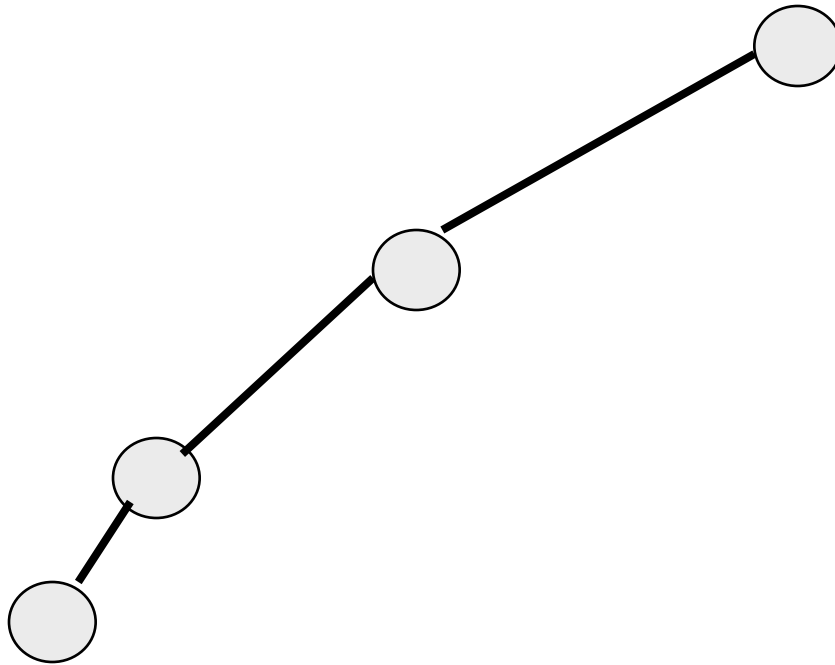


Properties and Terminology

- **Minimum Number of Nodes**
- **Maximum Number of Nodes**
- **Number of Nodes and Height**
- **Full Binary Tree**
- **Complete Binary Tree**

Minimum Number Of Nodes

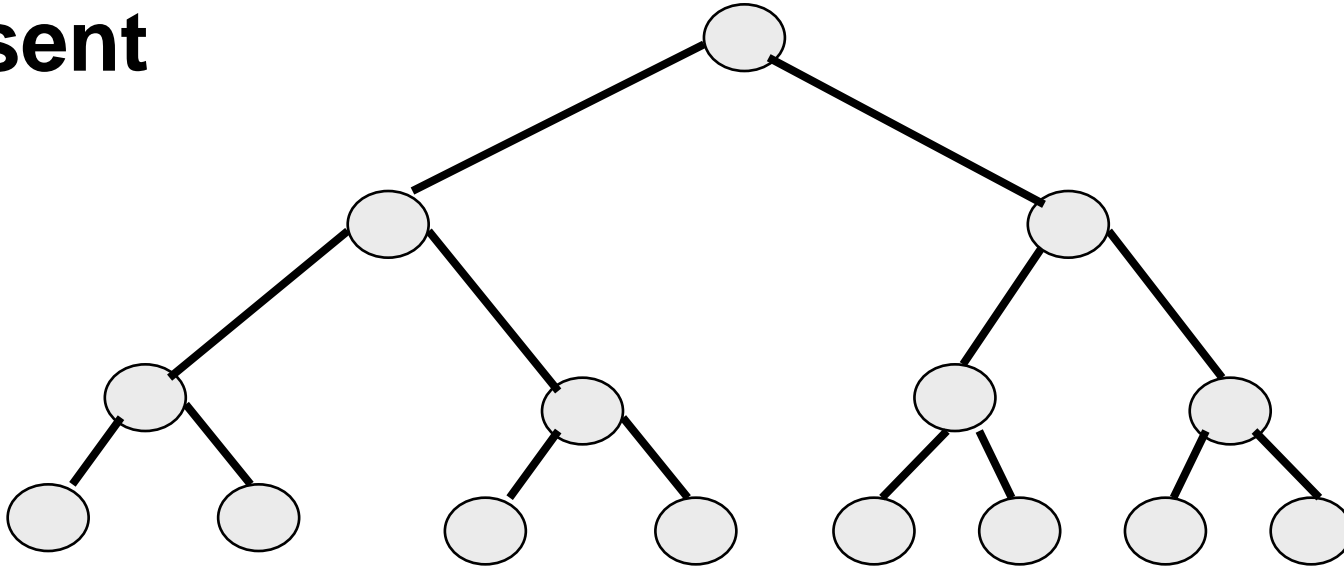
- Minimum number of nodes in a binary tree whose height is **h**
- At least one node at each of first **h** levels



minimum number of
nodes is h

Maximum Number Of Nodes

- All possible nodes at first **h** levels are present



Maximum number of nodes

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

$$= 2^h - 1$$

Number Of Nodes & Height

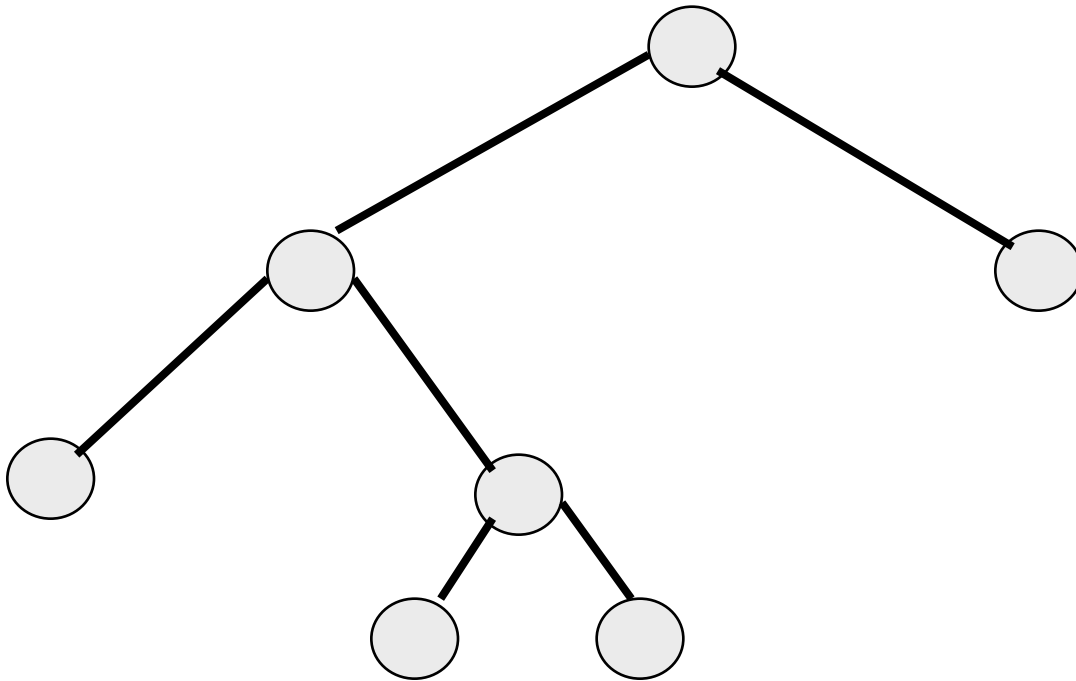
○ Let **n** be the number of nodes in a binary tree whose height is **h**

□ $h \leq n \leq 2^h - 1$

□ $\log_2(n+1) \leq h \leq n$

Strictly Binary Tree

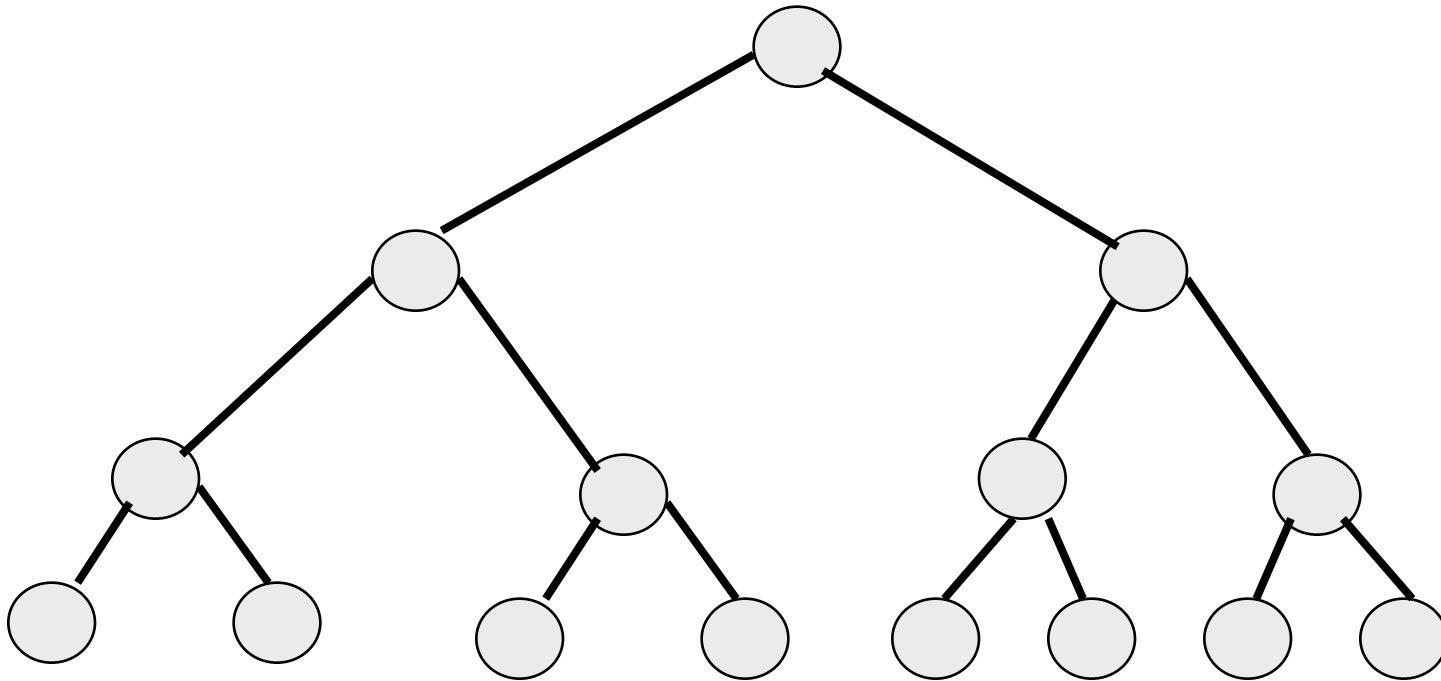
- Is a binary tree where all non leaf nodes have two branches.



Height **4** full binary tree - **15 nodes**

Full Binary Tree

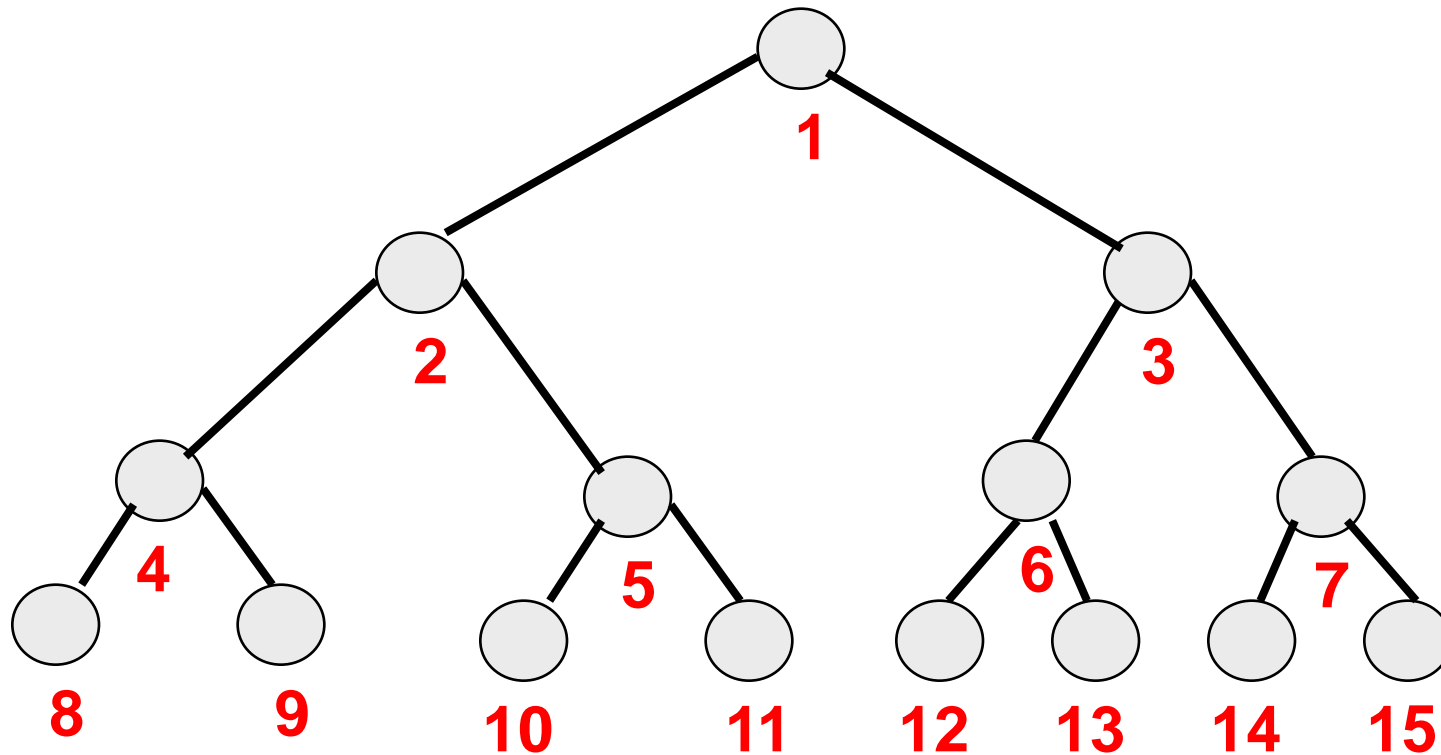
- A full binary tree of a given height **h** has **$2^h - 1$** nodes



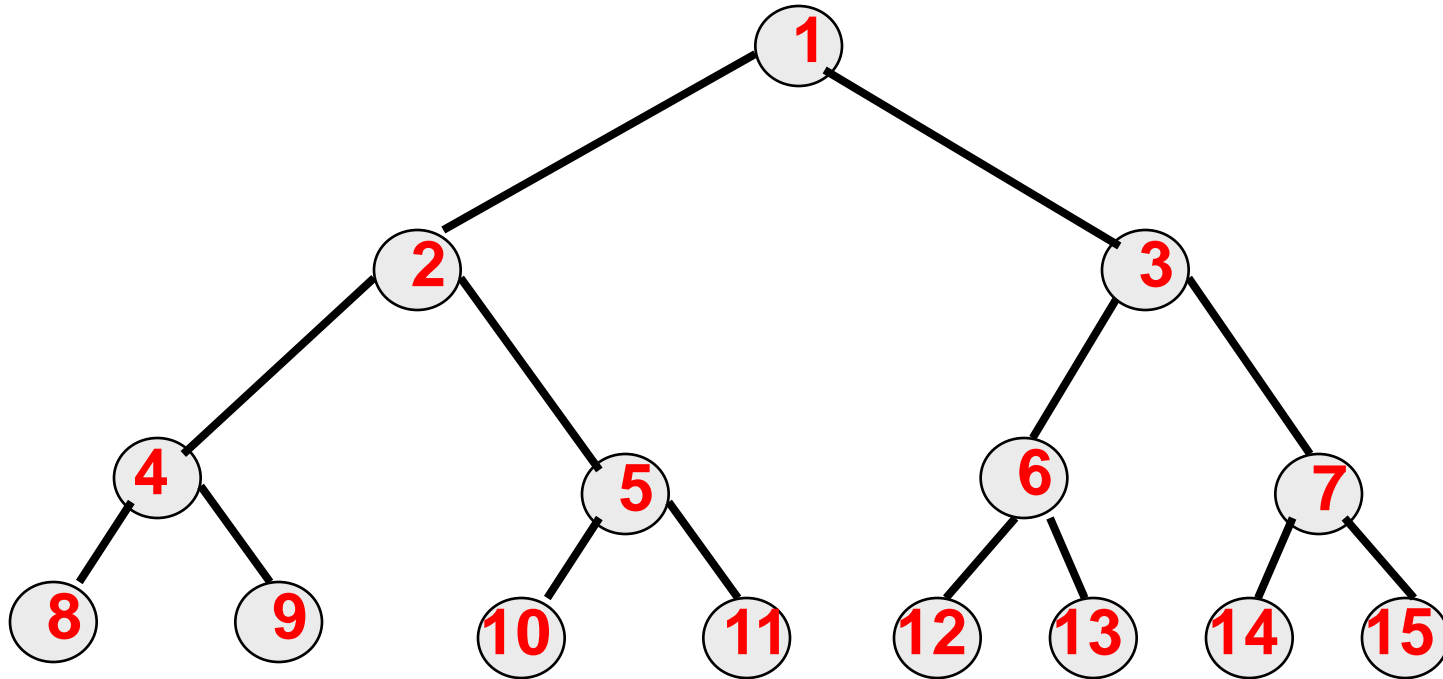
Height **4** full binary tree - **15 nodes**

Numbering Nodes In 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

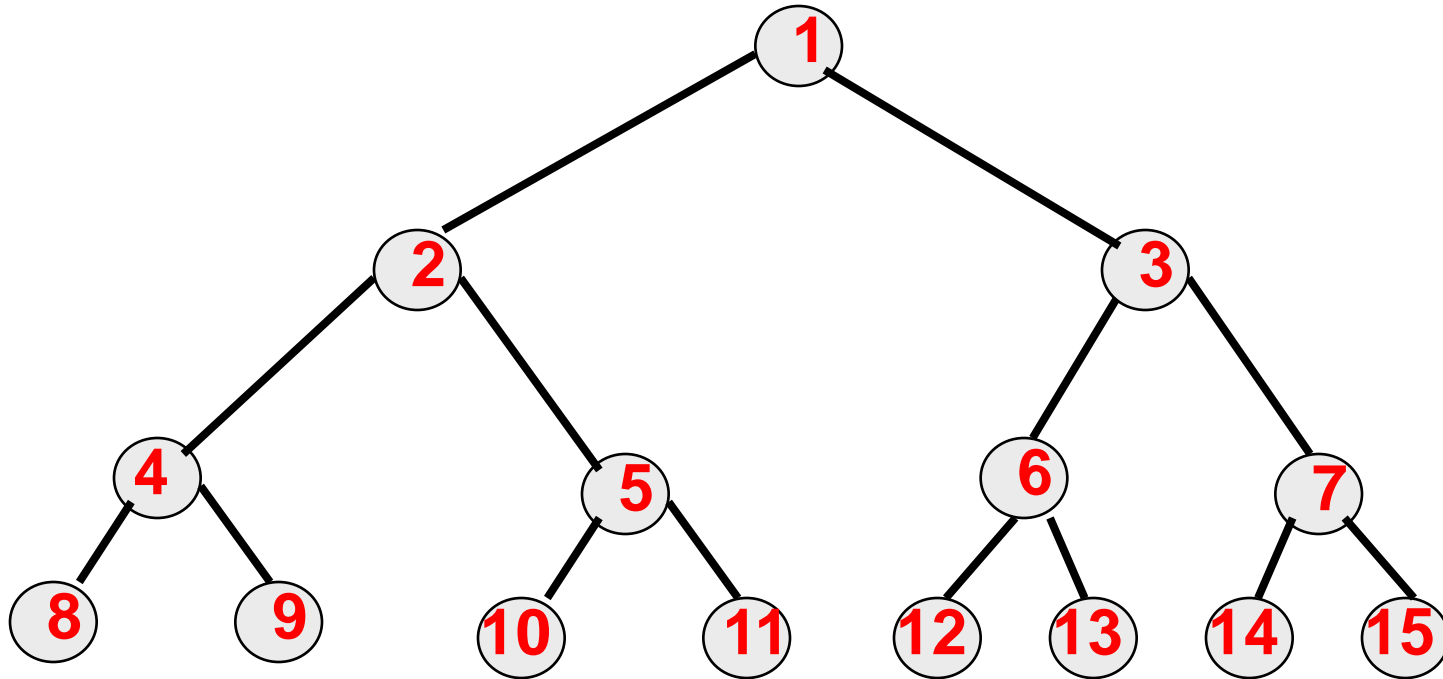


Node Number Properties



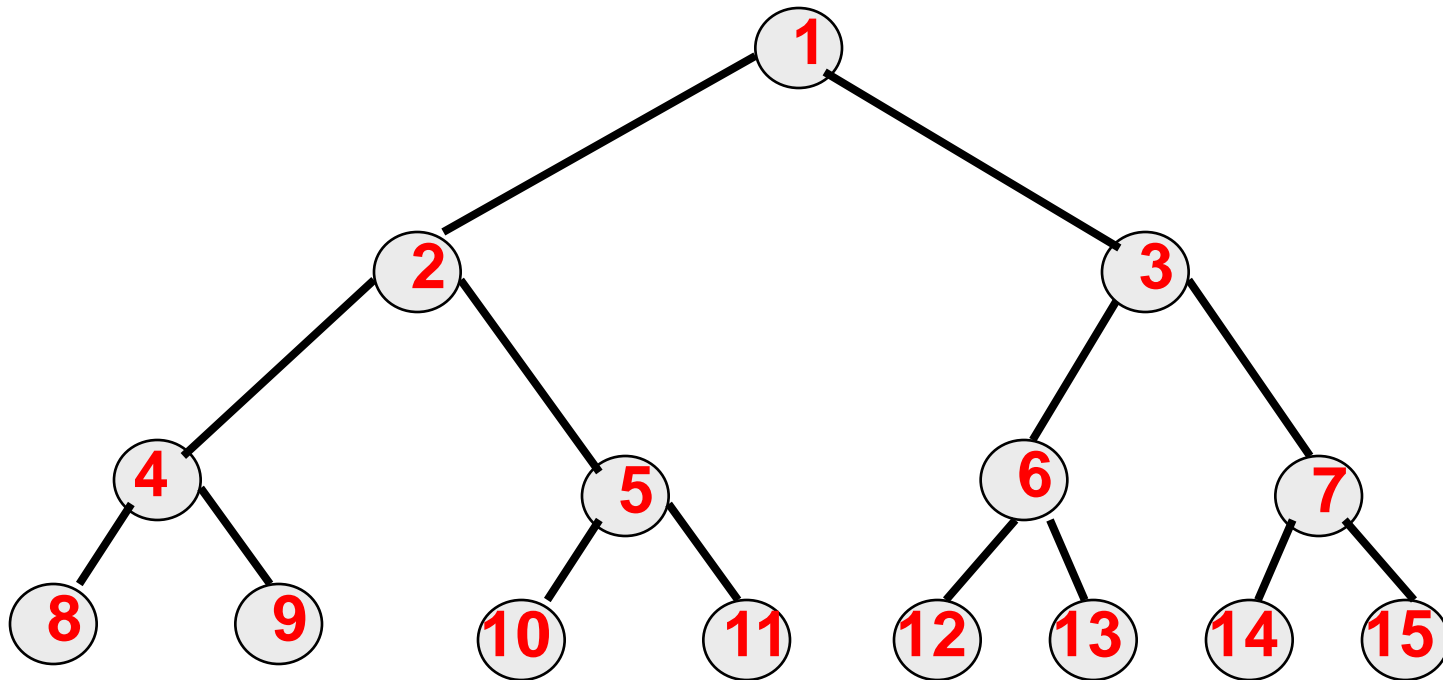
- Parent of node i is node $i / 2$, unless $i = 1$
- Node 1 is the root and has no parent

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

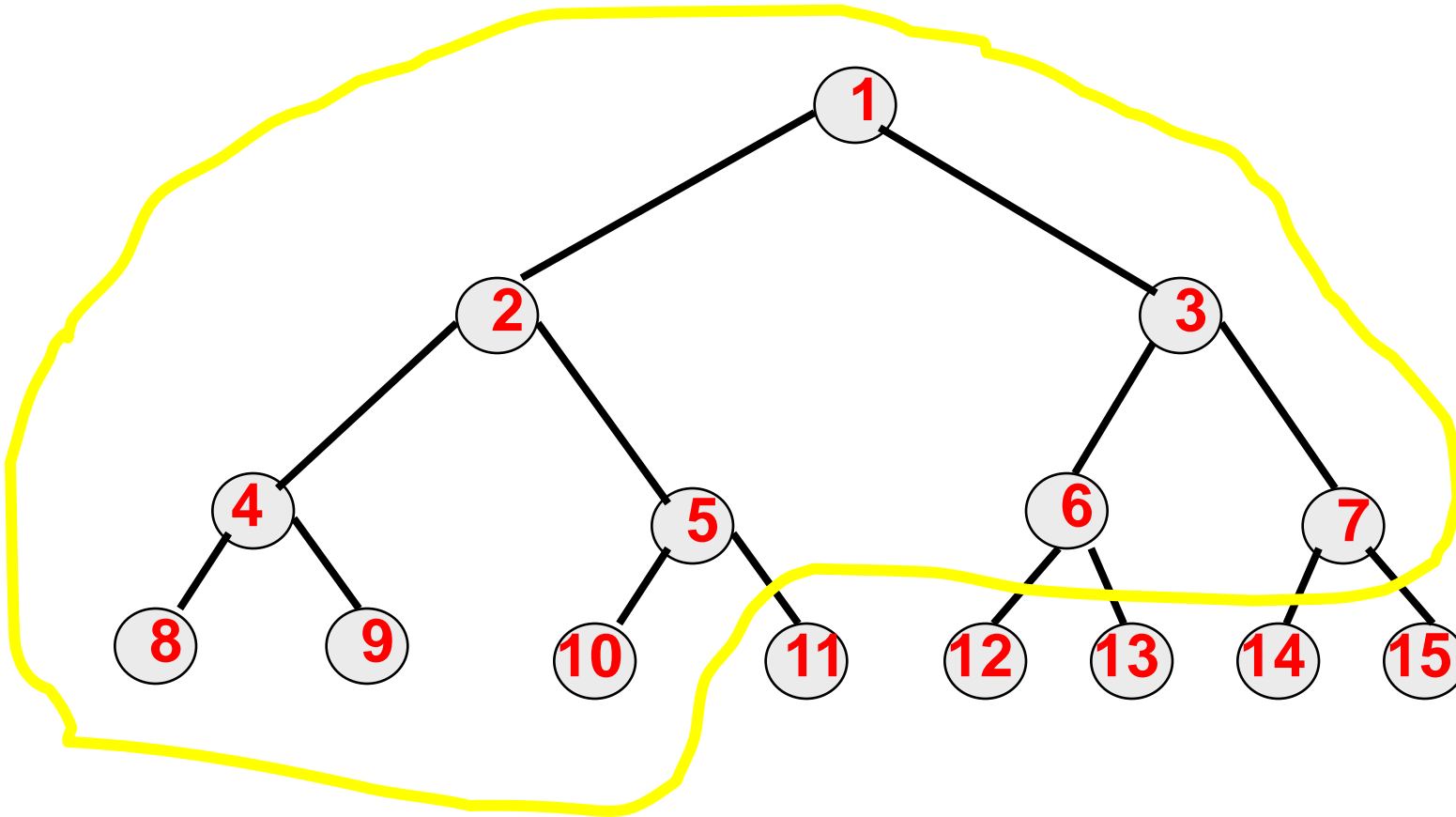


- Right child of node i is node $2i+1$, unless $2i+1 > n$, where n is the number of nodes
- If $2i+1 > n$, node i has no right child

Complete Binary Tree With n Nodes

- **Start with a full binary tree that has at least n nodes**
- **Number the nodes as described earlier**
- **The binary tree defined by the nodes numbered 1 through n is the unique n node complete binary tree**

Example



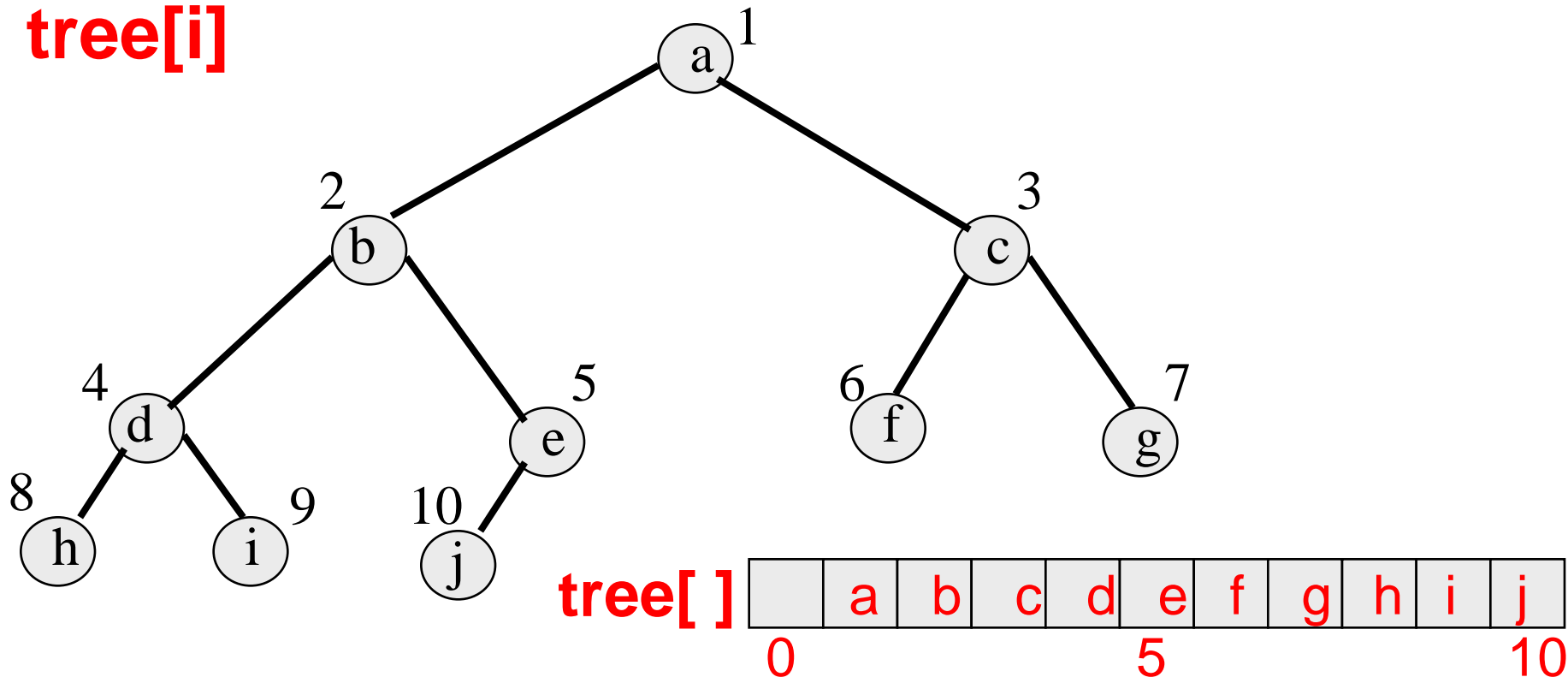
- Complete binary tree with **10** nodes

Binary Tree Representation

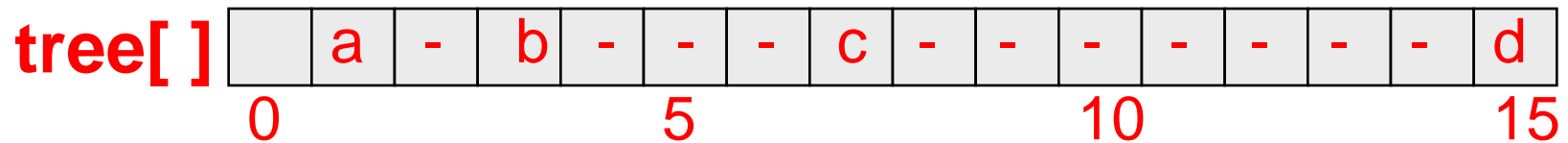
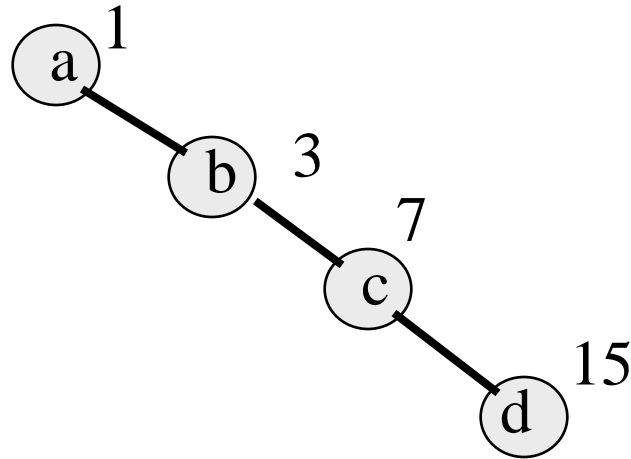
- Again, there are two ways to implement a tree data structure:
 - Array representation
 - Linked representation

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]**



Right-Skewed Binary Tree

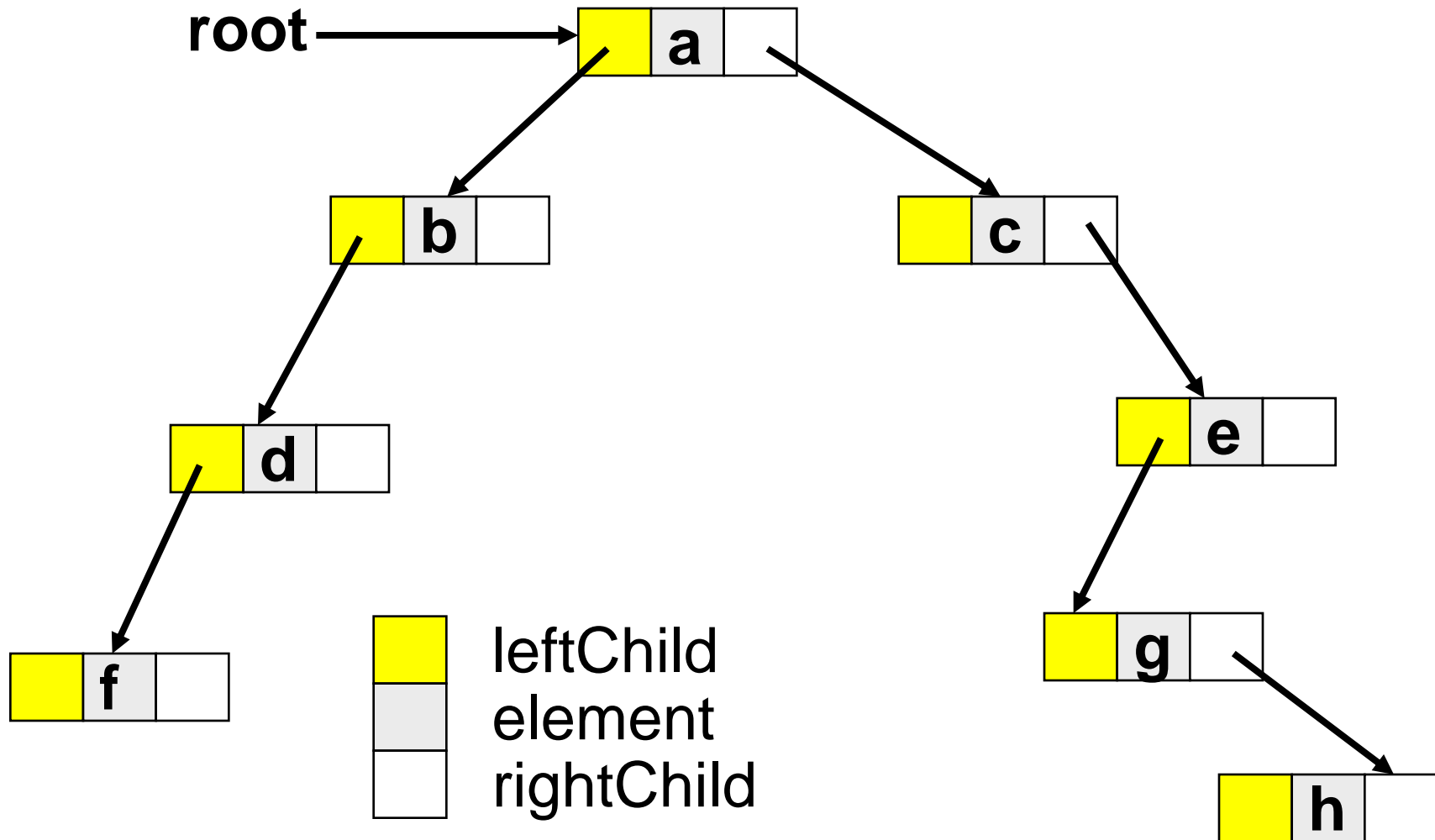


- An **n** node binary tree needs an array whose length is between **n+1** and **2ⁿ**

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 **$n * (\text{space required by one node})$**

Linked Representation Example



Binary Tree ADT

Sahni p473

AbstractDataType *BinaryTree*{
 instances

collection of elements; if not empty, the collection is partitioned into a root,
left subtree and right subtree; each subtree is also a binary tree

operations

isEmpty(): return true if the tree is empty, false otherwise

root(): return the root element, return null if tree is empty

makeTree(root, left, right) create a binary tree

removeLeftSubtree(): remove the left subtree and return it

removeRightSubtree(): remove the right subtree and return it

preOrder(): preorder traversal of binary tree

postOrder(): preorder traversal of binary tree

inOrder(): preorder traversal of binary tree

levelOrder(): preorder traversal of binary tree

}

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

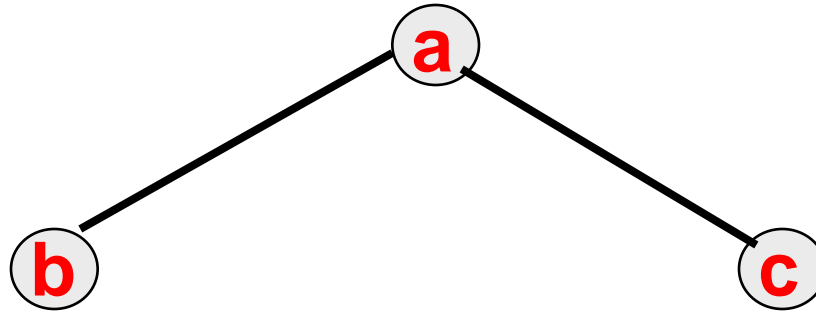
Binary Tree Traversals

- 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

Binary Tree Traversal Methods

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

Preorder Example (visit = print)

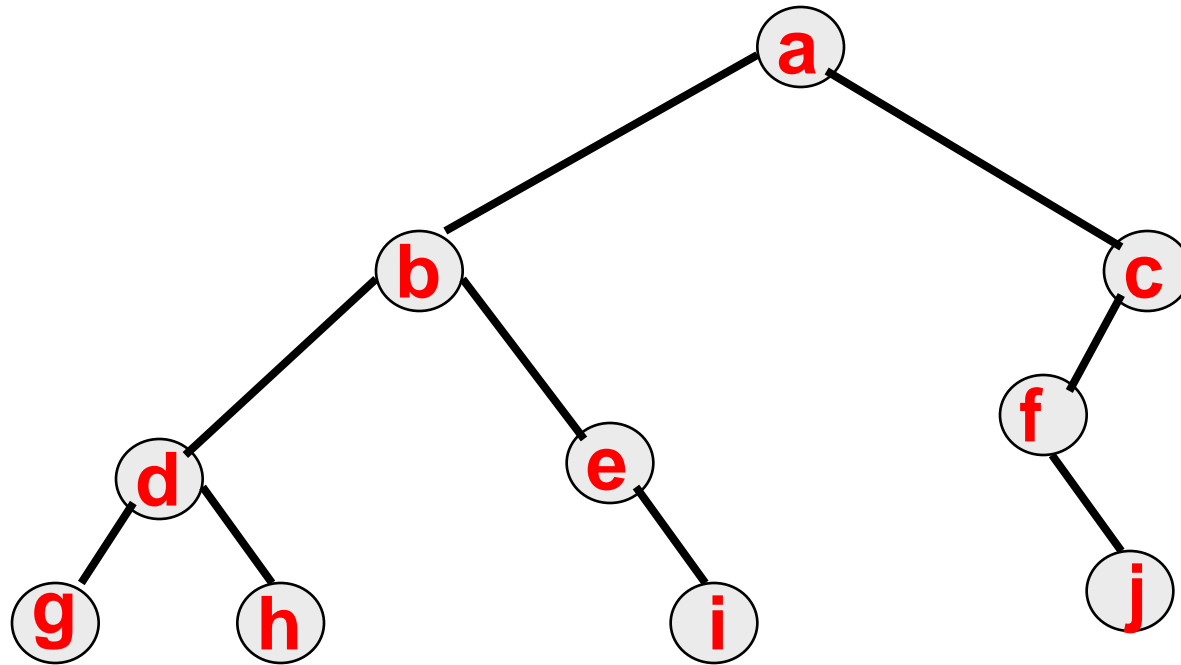


a b c

Preorder Traversal (DLR)

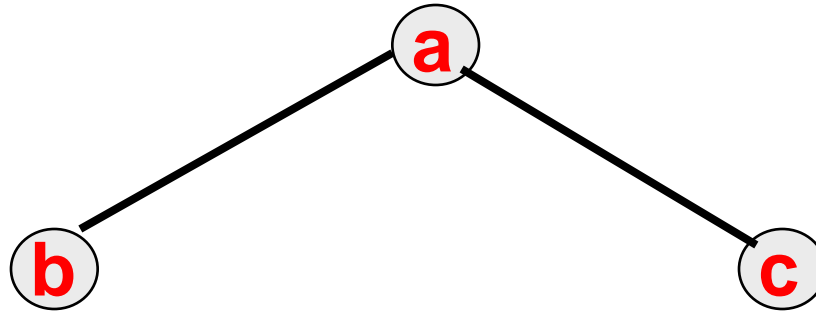
- 1. Visit the root**
- 2. Traverse the left subtree in Preorder**
- 3. Traverse the right subtree in Preorder**

Preorder Example (visit = print)



a b d g h e i c f j

Inorder Example (visit = print)

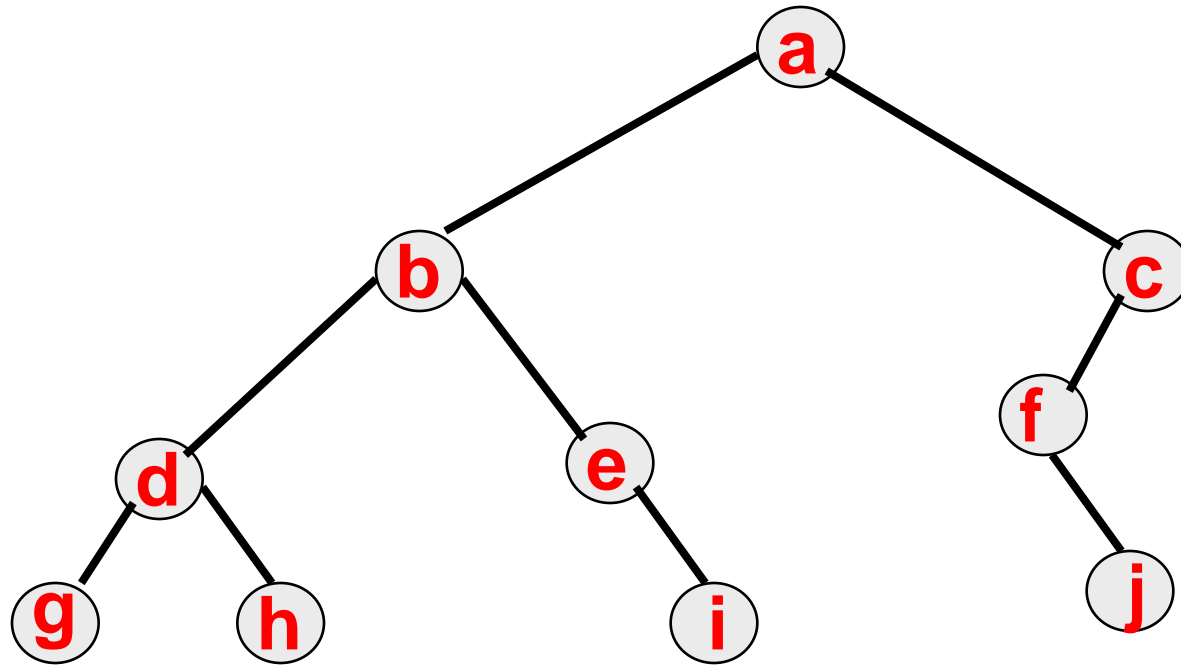


b a c

Inorder Traversal (LDR)

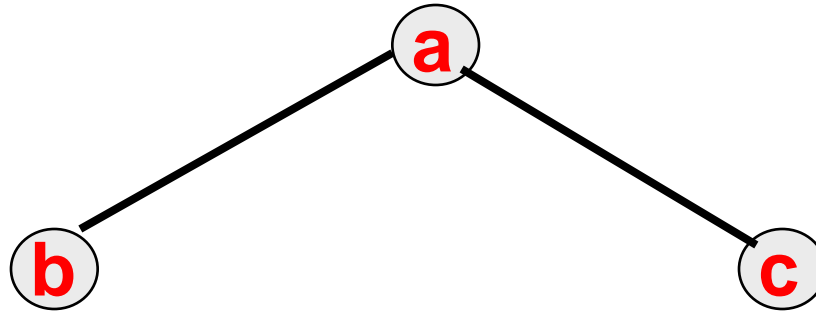
1. **Traverse the left subtree in Inorder.**
2. **Visit the root**
3. **Traverse the right subtree in Inorder.**

Inorder Example (visit = print)



g d h b e i a f j c

Postorder Example (visit = print)

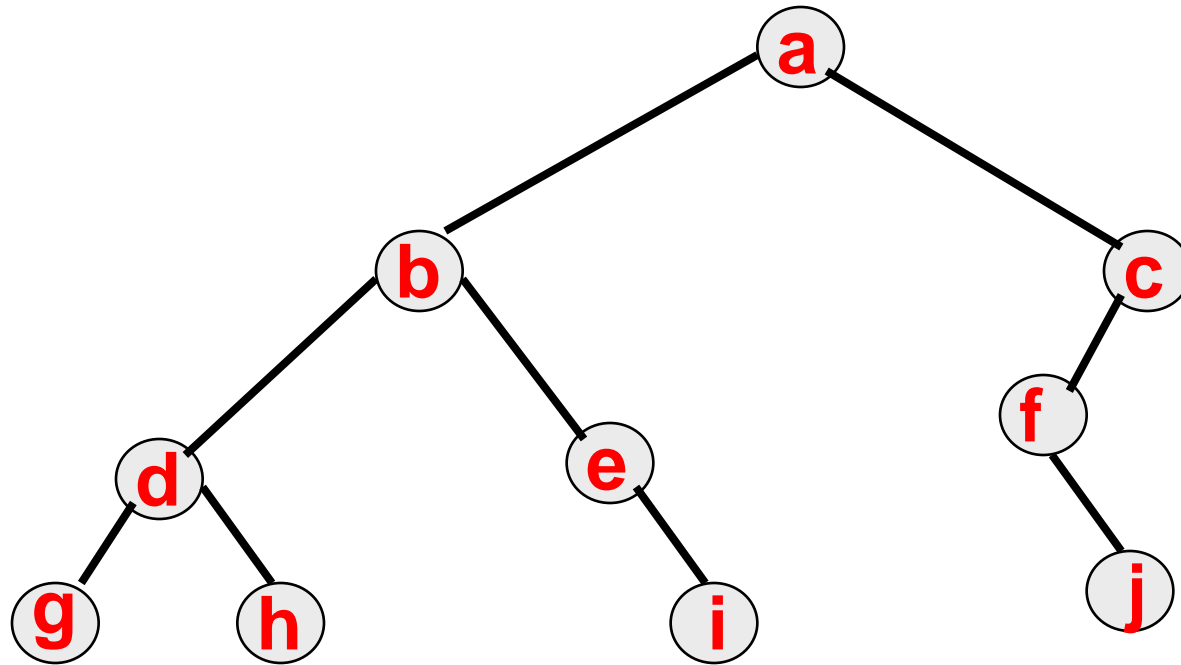


b c a

Postorder Traversal (LRD)

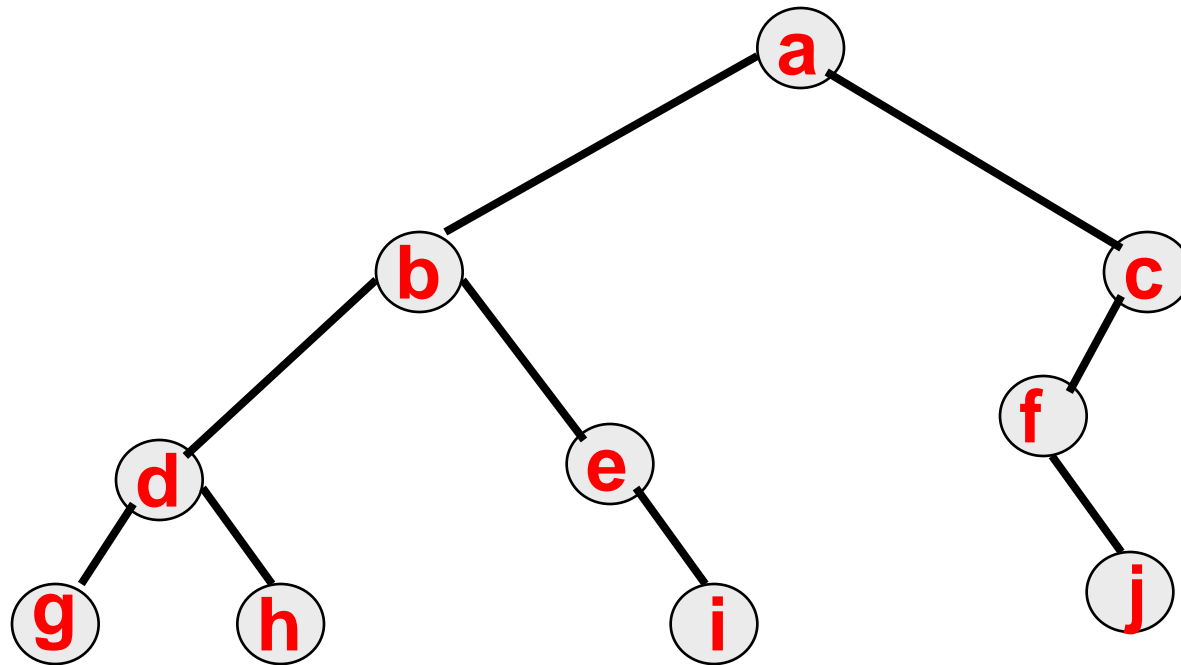
- 1. Traverse the left subtree in Postorder.**
- 2. Traverse the right subtree in Postorder**
- 3. Visit the root.**

Postorder Example (visit = print)



g h d i e b j f c a

Traversal Applications



- Determine height
- Determine number of nodes

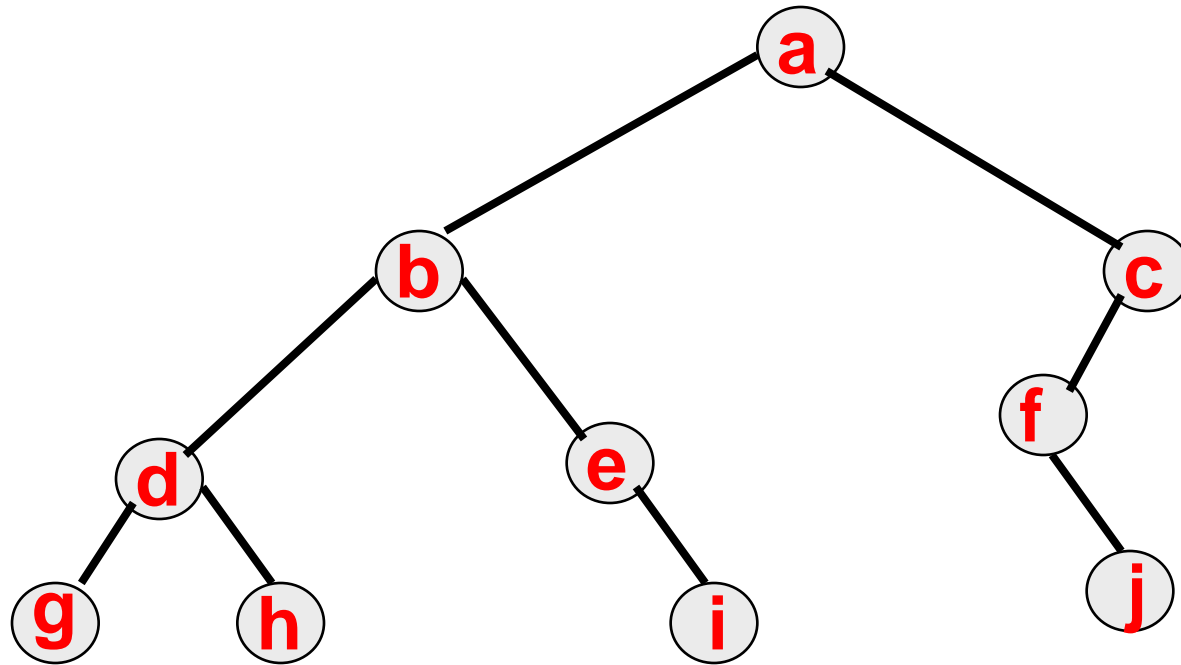
Level Order

Traverse the tree level wise from left to right starting from the root. It is implemented using a queue.

Let t be the tree root

```
while (t != null){  
  
    visit t and put its children on a FIFO queue;  
  
    remove a node from the FIFO queue and call it t;  
  
    // remove returns null when queue is empty  
  
}
```

Level-Order Example (visit = print)



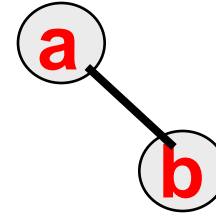
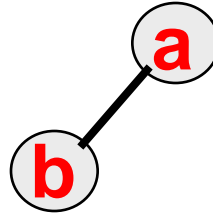
a b c d e f g h i j

Binary Tree Construction

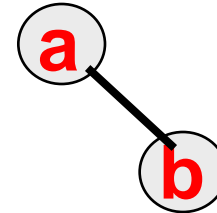
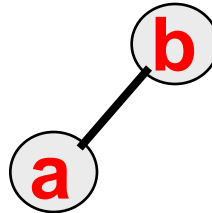
- **Suppose that the elements in a binary tree are distinct ...**
 - **Can you construct the binary tree from which a given traversal sequence came?**
- **When a traversal sequence has more than one element, the binary tree is not uniquely defined ...**
 - **Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely**

Some Examples

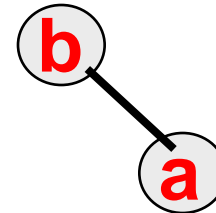
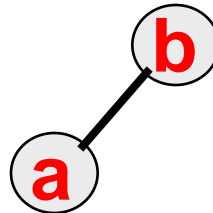
preorder = ab



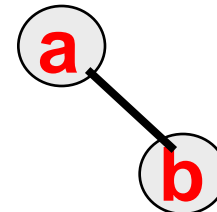
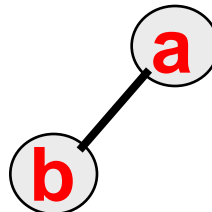
inorder = ab



postorder = ab



level order = ab



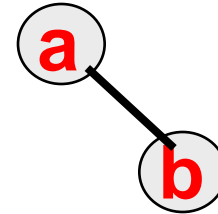
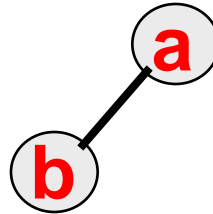
Binary Tree Construction

- Can you construct the binary tree, given two traversal sequences?
- Depends on which two sequences are given ...

Preorder And Postorder

preorder = ab

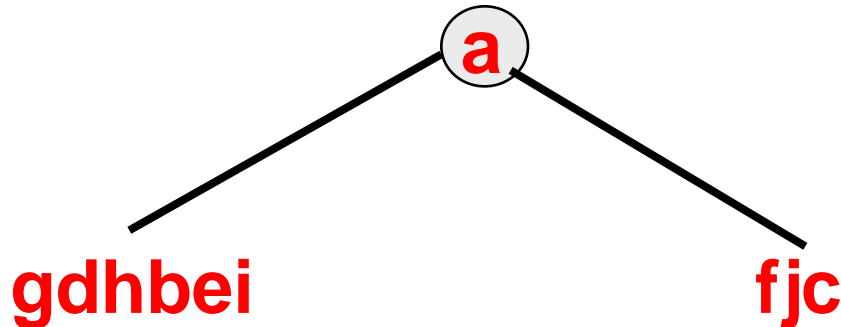
postorder = ba



- **Preorder and postorder do not uniquely define a binary tree**
- **Nor do preorder and level order (same example)**
- **Nor do postorder and level order (same example)**

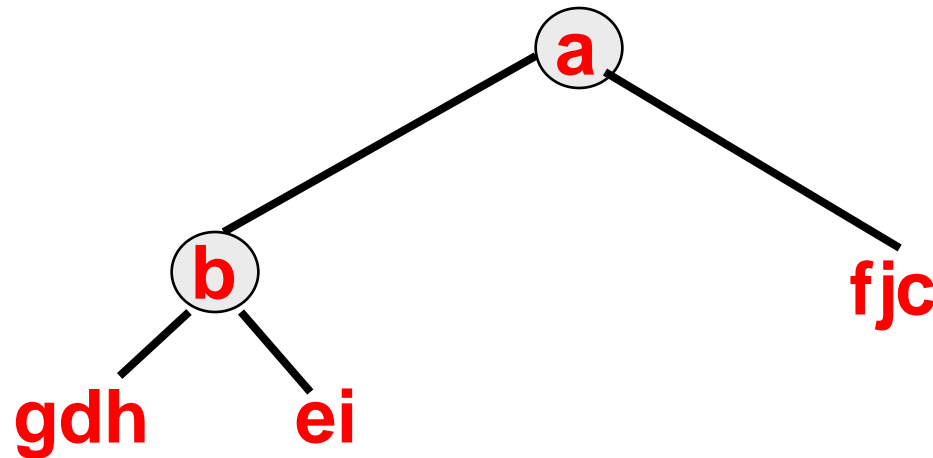
Inorder And Preorder

- **inorder = g d h b e i a f j c**
- **preorder = a b d g h e i c f j**
- **Scan the preorder left to right using the inorder to separate left and right subtrees.**
- **a** is the root of the tree; **gdhbei** are in the left subtree; **fjc** are in the right subtree.



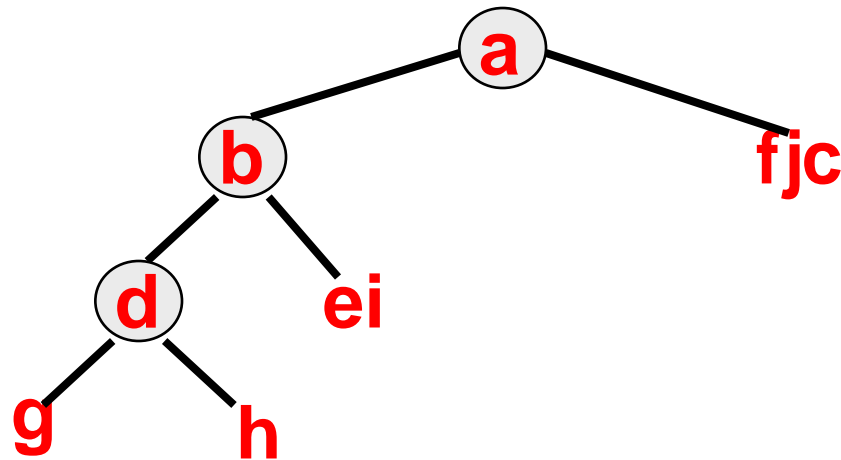
Inorder And Preorder

- preorder = **a b d g h e i c f j**
- **b** is the next root; **gdh** are in the left subtree; **ei** are in the right subtree



Inorder And Preorder

- preorder = **a b d g h e i c f j**
- **d** is the next root; **g** is in the left subtree; **h** is in the right subtree



Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees
- Example:
 - inorder = **g d h b e i a f j c**
 - postorder = **g h d i e b j f c a**
 - Tree root is **a**; **gdhbei** are in left subtree; **fjc** are in right subtree.

Inorder And Level Order

- Scan level order from left to right using inorder to separate left and right subtrees.
- Example:
 - inorder = **g d h b e i a f j c**
 - level order = **a b c d e f g h i j**
 - Tree root is **a**; **gdhbei** are in left subtree; **fjc** are in right subtree.

Trees Traversals

Expression Trees

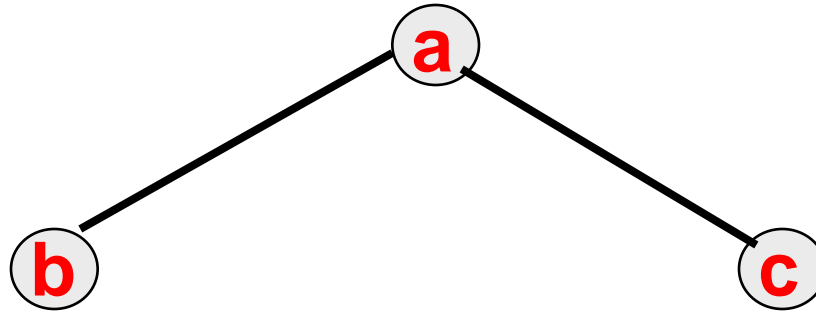
Binary tree creation

```
btree * insert_btree(btree *node)
{
    if(root == NULL)
        root = node;
    else
    {
        flag = 0;
        Initialize temp = root;
        while(flag == 0)
        {
            Display "\nWhere to add (l/r) of temp->data ? ";
            ch = getche();
            if(ch == 'l')
            {
                if(temp->lc == NULL)
                {
                    temp->lc = node;
                    flag = 1;
                }
                else
                    temp = temp->lc;
            }
            else
            {
                if(temp->rc == NULL)
                {
                    temp->rc = node;
                    flag = 1;
                }
                else
                    temp = temp->rc;
            }
        }
    }
}
```

Binary Tree Traversal Methods

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

Preorder Example (visit = print)



a b c

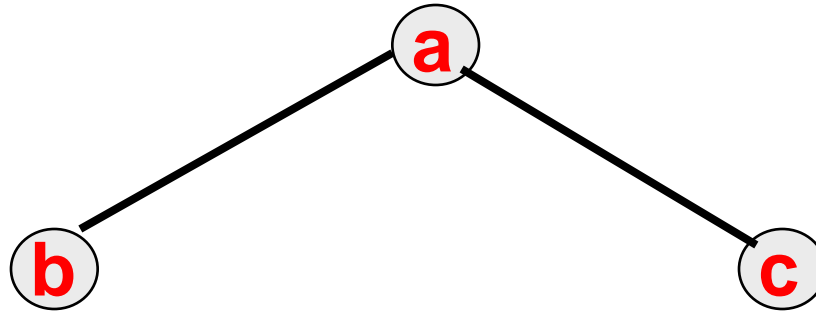
Preorder Traversal (VLR)

- 1. Visit the root**
- 2. Traverse the left subtree in Preorder**
- 3. Traverse the right subtree in Preorder**

Preorder Recursive Traversal

```
void Preorder(btree *node)
{
    if ( node != NULL )
    {
        print node->data
        Preorder( node->lchild );
        Preorder( node->rchild );
    }
}
```

Inorder Example (visit = print)



b a c

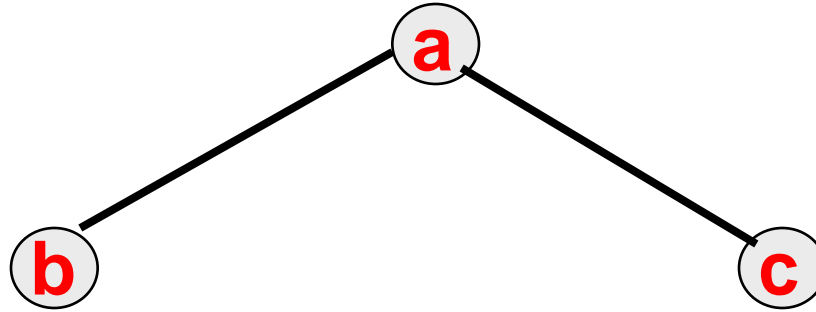
Inorder Traversal (LVR)

1. **Traverse the left subtree in Inorder.**
2. **Visit the root**
3. **Traverse the right subtree in Inorder.**

Inorder Recursive Traversal

```
void Inorder(btree *node)
{
    if ( node != NULL )
    {
        Inorder( node->lchild );
        print node->data
        Inorder( node->rchild );
    }
}
```

Postorder Example (visit = print)



b c a

Postorder Traversal (LRV)

- 1. Traverse the left subtree in Postorder.**
- 2. Traverse the right subtree in Postorder**
- 3. Visit the root.**

Postorder Recursive Traversal

```
void Postorder(btree *node)
{
    if ( node != NULL )
    {
        Postorder( node->lchild );
        Postorder( node->rchild );
        print  node->data
    }
}
```


Non Recursive Traversals?

- Can we implement traversal *non-recursively*?
 - Recursion is implemented by using a stack.
 - We can traverse non-recursively by maintaining the stack ourselves (*emulate stack of activation records*).
- Is a non-recursive approach faster than recursive approach?
 - Possibly: we can place only the essentials, while the compiler places an entire activation record.

NonRecursive Inorder Traversal

```
void Inorder_nonrecursive(btree *root)
{
    Initialize temp to root
    while ( temp != NULL OR  stack is not empty)
    {
        while(temp != NULL)
        {
            Push temp onto stack;
            temp = temp->lchild;
        }
        temp = pop from the stack
        Print temp->data
        temp = temp->rchild;
    }
}
```

NonRecursive Preorder Traversal

```
Void Preorder_nonrecursive(btree *root)
{
    Initialize temp to root
    while ( temp != NULL OR  stack is not empty)
    {
        while(temp != NULL)
        {
            Print temp->data
            Push temp onto stack;
            temp = temp->lchild;
        }
        temp = pop from the stack
        temp = temp->rchild;
    }
}
```

Non Recursive Preorder Traversal

Also called as Depth First Search (DFS)

```
void preorder_nonrecursive(btree *root)
```

```
{
    Initialize temp to root
    while(temp != NULL )
    {
        while(temp != NULL)
        {
            print temp->data
            if temp's rchild exists then push that rchild onto the stack;
            temp = temp->lc;
        }
        temp = pop from the stack
    }
}
```

Postorder Traversal using Stack

- **Use stack to store the current state (nodes we have traversed but not yet completed)**
 - **Similar to PC (program counter) in the activation record**
- **We “pop” each node three times, when:**
 - 0. about to make a recursive call to left subtree**
 - 1. about to make a recursive call to right subtree**
 - 2. about to process the current node itself**

Postorder Non Recursive Algorithm

```
void Postorder_nonrecursive(btree *root)
```

```
{
```

```
    push root onto stack with state 0
```

```
    while (stack is not empty)
```

```
    {
```

```
        node X = pop from the stack
```

```
        switch (state X)
```

```
        {
```

```
            case 0 : push node X with state 1
```

```
                    push left child of X (if it exists) with state 0  
                    break;
```

```
            case 1 : push node X with state 2
```

```
                    push right child of X (if it exists) with state 0  
                    break;
```

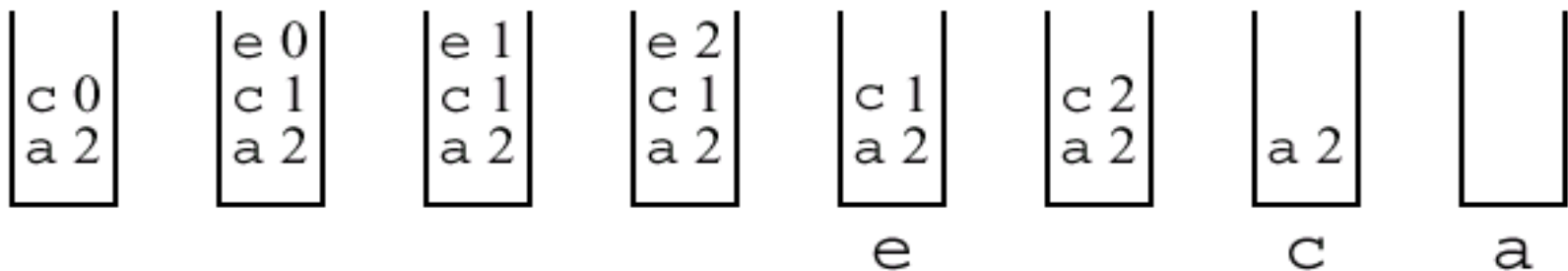
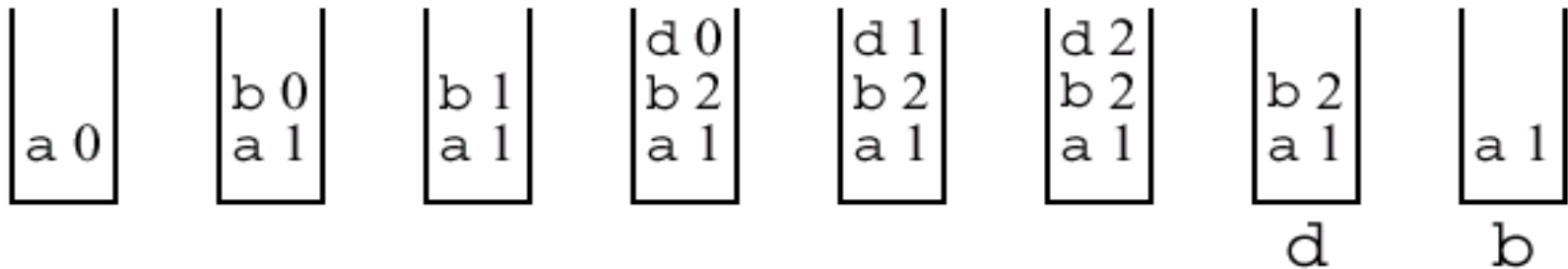
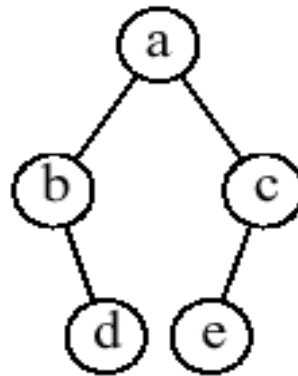
```
            case 2 : visit / process node X
```

```
                    break;
```

```
        }
```

```
    }
```

Postorder Traversal: Stack States



Non recursive postorder traversal

○ Short cut method

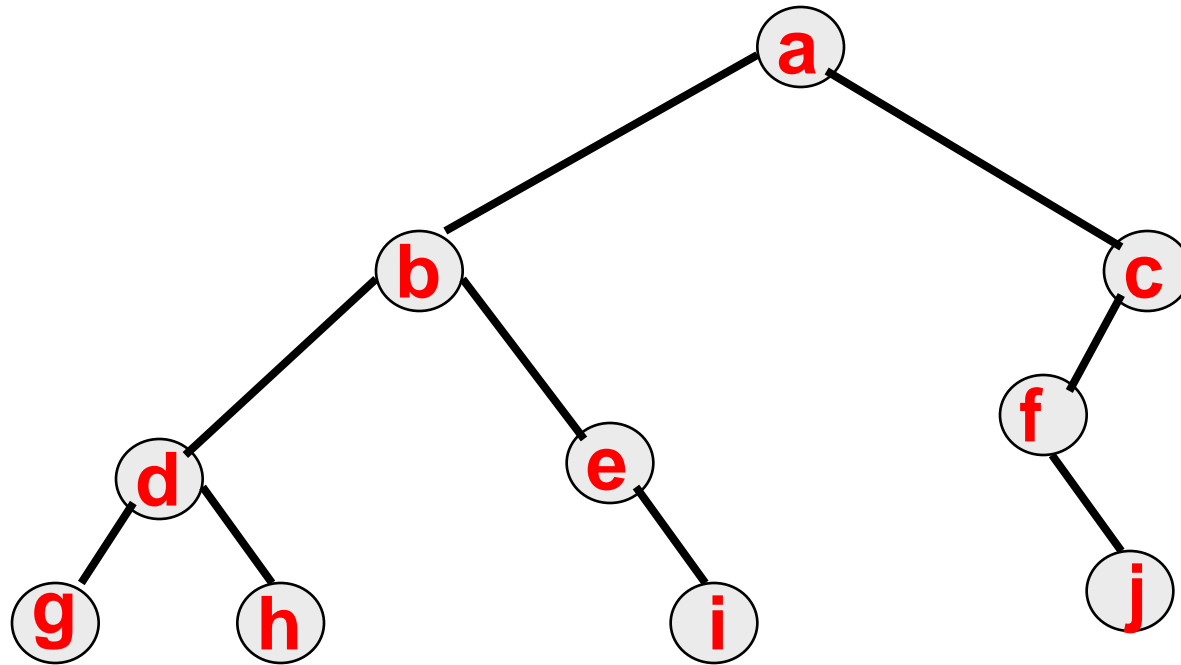
```
void BTREE :: postorder_nonrec()
{
    Initialize temp to root and k to 0
    while(temp != NULL )
    {
        while(temp != NULL)
        {
            A[k++] = temp->data;
            if(temp's lchild exists then push that lchild onto the stack
            temp = temp->rc;
        }
        temp = pop from the stack
    }
    for(int i=k-1;i>= 0 ;i--)
        print A[i]
}
```


Level Order

It is implemented using a queue. Also called as Breadth First Search (BFS) .

```
void levelorder(btree *root)
{
    enqueue ( root)
    while ( Queue is not empty )
    {
        temp = dequeue();
        print temp->data;
        if (temp->lchild != NULL)
            enqueue ( temp->lchild);
        if (temp->rchild != NULL)
            enqueue ( temp->rchild);
    }
}
```

Level-Order Example (visit = print)



a b c d e f g h i j

5 / 6 Marks University Questions

- Write an non-recursive algorithm for **inorder** traversal .
- Write an non-recursive algorithm for **preorder** traversal.
- Write an non-recursive algorithm for **postorder** traversal.
- Write an algorithm for traversing the tree levelorder or BFS for Binary tree
- Write an algorithm for traversing the tree by DFS method.

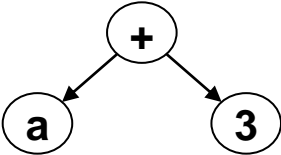
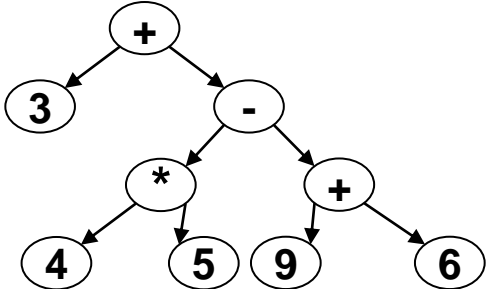
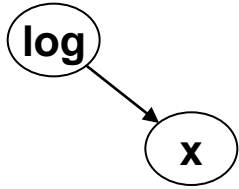
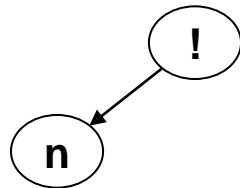
Expression Trees

- **What is an Expression tree?**
- **Expression tree implementation**
- **Why expression trees?**
- **Evaluating an expression tree (pseudo code)**
- **Prefix, Infix, and Postfix forms**
- **Infix to Postfix conversion**
- **Constructing an expression tree from a postfix expression or prefix expression or infix expression**

What is an Expression tree?

- **An expression tree for an arithmetic, relational, or logical expression is a binary tree in which:**
 - **The parentheses in the expression do not appear.**
 - **The leaves are the variables or constants in the expression.**
 - **The non-leaf nodes are the operators in the expression:**
 - **A node for a binary operator has two non-empty subtrees.**
 - **A node for a unary operator has one non-empty subtree.**
- **The operators, constants, and variables are arranged in such a way that an inorder traversal of the tree produces the original expression without parentheses.**

Expression Tree Examples

Expression	Expression Tree	Inorder Traversal Result
$(a+3)$	 <pre>graph TD; A((+)) --> B((a)); A --> C((3))</pre>	$a + 3$
$3+(4*5-(9+6))$	 <pre>graph TD; A((+)) --> B((3)); A --> C((-)); C --> D((*)); C --> E((+)); D --> F((4)); D --> G((5)); E --> H((9)); E --> I((6))</pre>	$3+4*5-9+6$
$\log(x)$	 <pre>graph TD; A((log)) --> B((x))</pre>	$\log x$
$n!$	 <pre>graph TD; A((!)) --> B((n))</pre>	$n !$

Expression tree implementation

```
struct ETREE
{
    struct ETREE *lc;
    char data;
    struct ETREE *rc;
};
typedef struct ETREE etree;
```

Why Expression trees?

- Expression trees are used to remove ambiguity in expressions.
- Consider the algebraic expression $2 - 3 * 4 + 5$.
- Without the use of precedence rules or parentheses, different orders of evaluation are possible:
 - $((2-3)*(4+5)) = -9$
 - $((2-(3*4))+5) = -5$
 - $(2-((3*4)+5)) = -15$
 - $((2-3)*4)+5 = 1$
 - $(2-(3*(4+5))) = -25$
- The expression is ambiguous because it uses infix notation: each operator is placed between its operands.

Why Expression trees? (contd.)

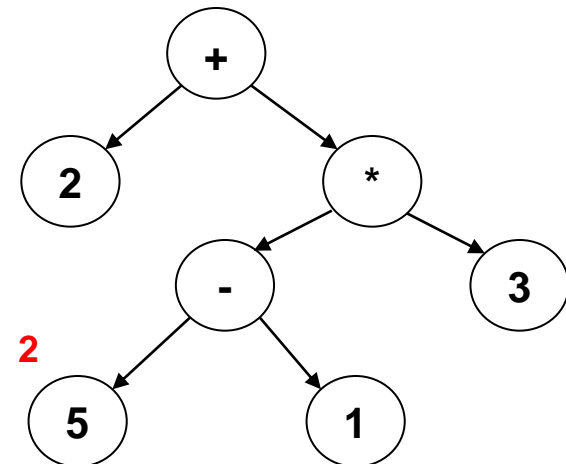
- Storing a fully parenthesized expression, such as $((x+2)-(y*(4-z)))$, is wasteful, since the parentheses in the expression need to be stored to properly evaluate the expression.
- A compiler will read an expression in a language like Java, and transform it into an expression tree.
- Expression trees impose a hierarchy on the operations in the expression. Terms deeper in the tree get evaluated first. This allows the establishment of the correct precedence of operations without using parentheses.
- Expression trees can be very useful for:
 - Evaluation of the expression.
 - Generating correct compiler code to actually compute the expression's value at execution time.
 - Performing symbolic mathematical operations (such as differentiation) on the expression.

- **A pseudo code algorithm for evaluating the expression tree is:**

```

Int  evaluate( Etree *temp)
{
    if(temp is a leaf node )
        return value of temp operand ;
    else
    {
        operator =  temp->data ;
        operand1 = evaluate(temp->lchild) ;
        operand2 = evaluate(temp->rchild) ;
        return(applyOperator(operand1, operator, operand2)) ;
    }
}

```



Order of evaluation: **3** **1**

$(2 + ((5 - 1) * 3))$

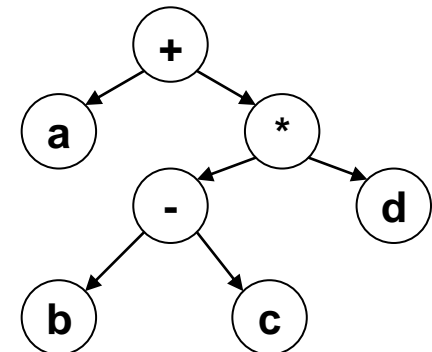
Prefix, Infix, and Postfix Forms

- A preorder traversal of an expression tree yields the prefix (or polish) form of the expression.
 - In this form, every operator appears before its operand(s).
- An inorder traversal of an expression tree yields the infix form of the expression.
 - In this form, every operator appears between its operand(s).
- A postorder traversal of an expression tree yields the postfix (or reverse polish) form of the expression.
 - In this form, every operator appears after its operand(s).

Prefix form: + a * - b c d

Infix form: a + b - c * d

Postfix form: a b c - d * +

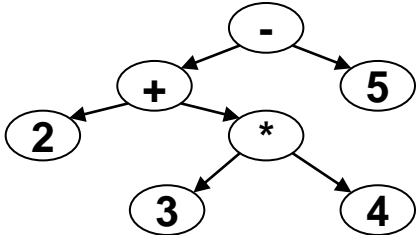
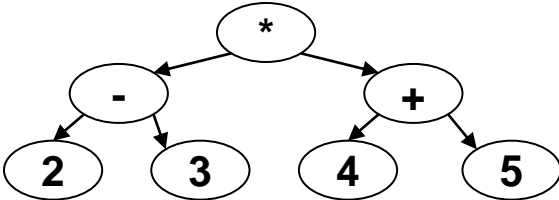
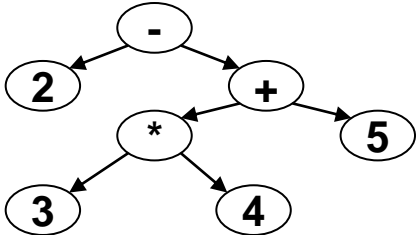


Prefix, Infix, and Postfix Forms (contd.)

Expression	Prefix forms	Infix forms	Postfix forms
$(a + b)$	$+ a b$	$a + b$	$a b +$
$a - (b * c)$	$- a * b c$	$a - b * c$	$a b c * -$
$\log (x)$	$\log x$	$\log x$	$x \log$
$n !$	$! n$	$n !$	$n !$

Prefix, Infix, and Postfix Forms (contd.)

- A prefix or postfix form corresponds to exactly one expression tree.
- An infix form may correspond to more than one expression tree. It is therefore not suitable for expression evaluation.

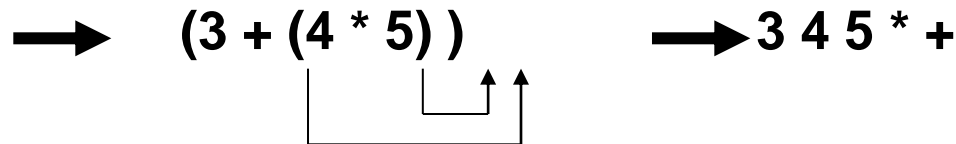
Expression	Expression Tree	Infix form
$2 - 3 * 4 + 5$		$2 - 3 * 4 + 5$
$(2 - 3) * (4 + 5)$		$2 - 3 * 4 + 5$
$2 - (3 * 4 + 5)$		$2 - 3 * 4 + 5$

Infix to Postfix conversion (manual)

- An Infix to Postfix manual conversion algorithm is:
 1. Completely parenthesize the infix expression according to order of priority you want.
 2. Move each operator to its corresponding right parenthesis.
 3. Remove all parentheses.
- Examples:

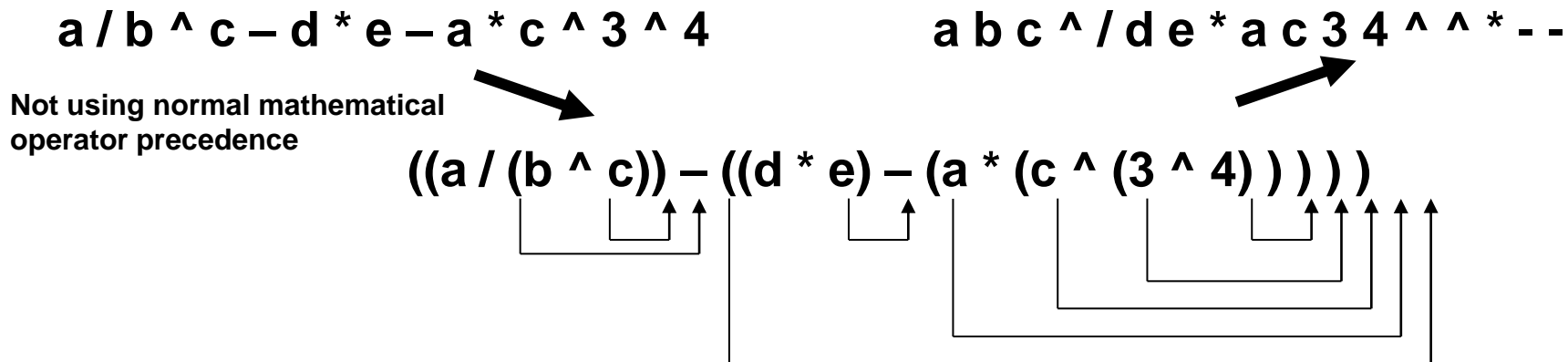
$3 + 4 * 5 \longrightarrow (3 + (4 * 5)) \longrightarrow 3\ 4\ 5\ *\ +$

Using normal mathematical operator precedence



$a / b ^ c - d * e - a * c ^ 3 ^ 4 \longrightarrow a\ b\ c\ ^\ /\ d\ e\ *\ a\ c\ 3\ 4\ ^\ ^\ *\ -\ -$

Not using normal mathematical operator precedence



Infix to Prefix conversion (manual)

- An Infix to Postfix manual conversion algorithm is:
 - 1 Completely parenthesize the infix expression according to order of priority you want.
 - 2 Move each operator to its corresponding left parenthesis.
 - 3 Remove all parentheses.
- Examples:

$3 + 4 * 5$

$(3 + (4 * 5))$

$3\ 4\ 5\ *\ +$



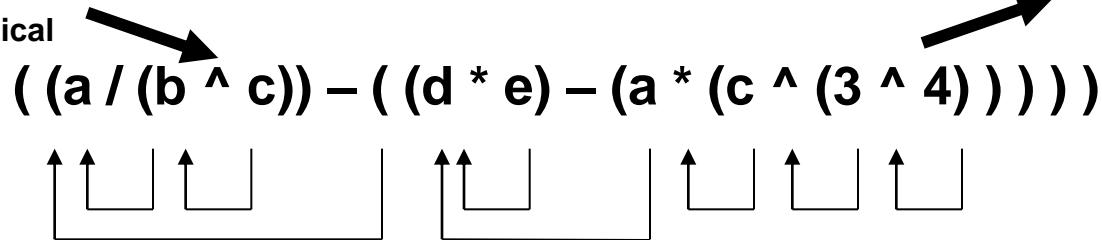
Using normal mathematical
operator precedence



$a / b ^ c - d * e - a * c ^ 3 ^ 4$

$a\ b\ c\ ^\ /\ d\ e\ *\ a\ c\ 3\ 4\ ^\ ^\ *\ -\ -$

Not using normal mathematical
operator precedence



Constructing an expression tree from a postfix expression

- The pseudo code algorithm to convert a valid postfix expression, containing binary operators, to an expression tree:

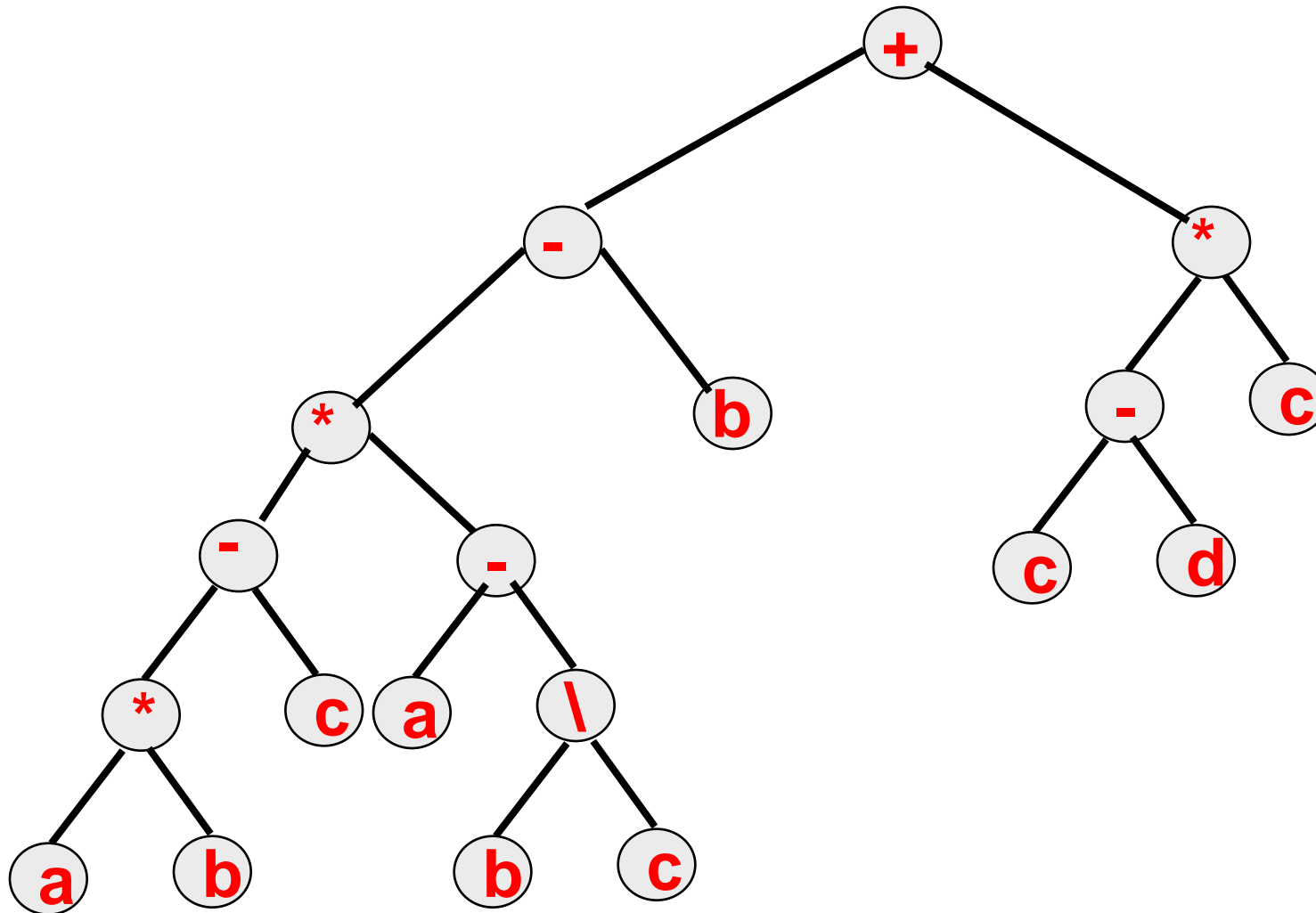
```
Etree * Construct_expression_tree( char pos [ ] )
{
    while(not the end of the expression)
    {
        if(the next symbol in the expression is an operand)
        {
            create a node for the operand ;
            push the node onto the stack ;
        }
        else // next symbol in the expression is a binary operator
        {
            create a node for the operator ;
            node->rchild = pop from the stack
            node->lchild = pop from the stack
            push the operator node onto the stack ;
        }
    }
    return(top of the stack);
}
```


Constructing tree from the prefix expression

1. First operator will form the root of the tree
2. Continue attaching the operators as the left child of the previous operator till we get the operand.
3. This operand will also be attached as left child and the next will be attached as the right child, irrespective of whether it is operand or operator.
4. If it is operator we repeat the procedure and if it operand, we go back to the previous operator whose right child is empty.

+ - * - * a b c - a \ b c b * - c d c

$+ - * - * a b c - a \backslash b c b * - c d c$



Pseudo code for constructing expression tree from prefix expression

```
Etree * Construct_expression_tree(char pre_expr[])
{
    root = createnode(pre_expr[0]);  temp = root;
    while(pre_expr[i] != '\0' )
    {
        while(pre_expr[i] is an operator)
        {
            temp->lchild = createnode(pre_expr[i++]);
            push temp onto stack;
            temp = temp->lchild;
        }
        temp->lchild = createnode(pre_expr[i++]);
        flag = 0;
        while(flag == 0)
        {
            temp->rchild = createnode(pre_expr[i++]);
            if(pre_expr[i-1] is an operand)
                temp = pop from stack;
            else
            {
                temp = temp->rchild;
                flag = 1;
            }
        }
    }
    return root;
}
```

Printing the tree graphically

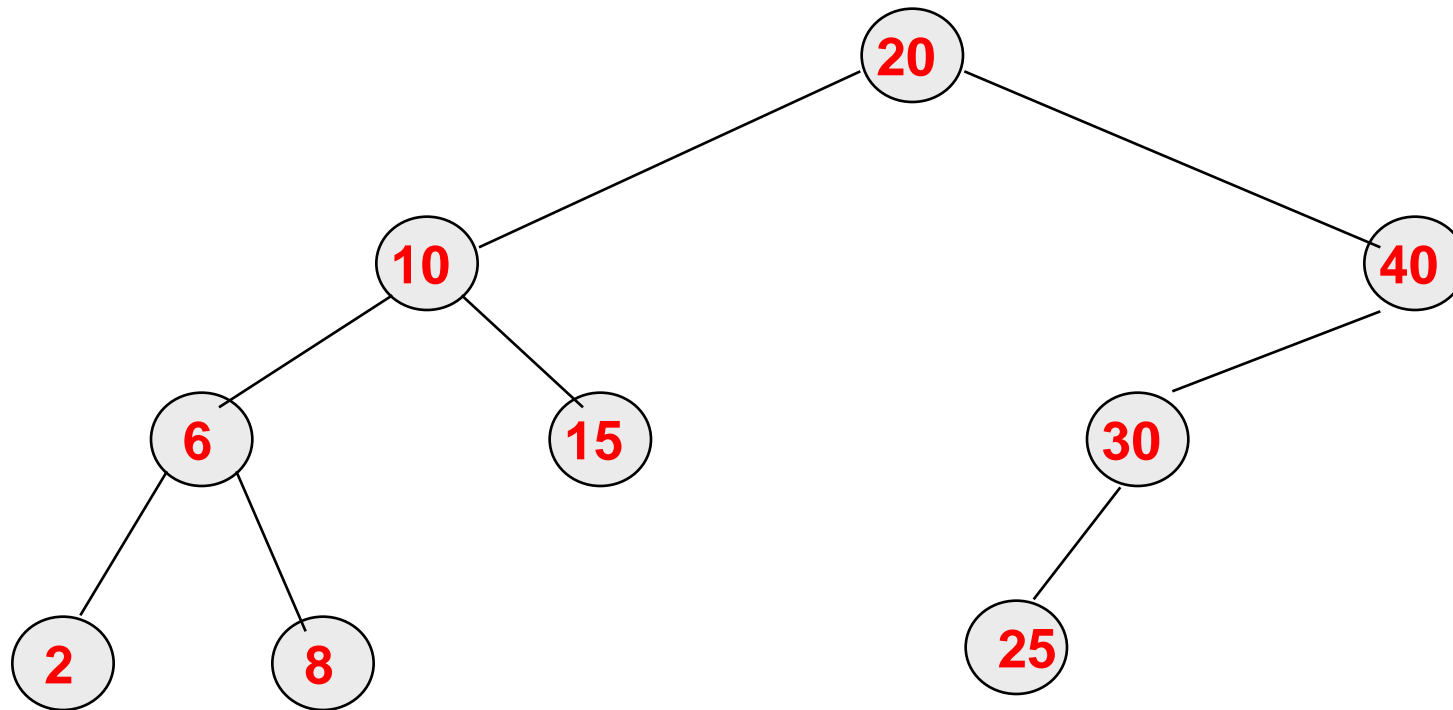
```
void display(bst *temp,int x, int y, int d,int px,int py)
{
    char ch[4];
    if(temp != NULL)
    {
        circle(x,y,12);
        itoa(temp->data,ch,10); // integer to string conversion
        outtextxy(x-7,y-2,ch);
        if(px!=0)
            line(px,py+12,x,y-12);
        display(temp->lc,x-d,y+50,d/2,x,y);
        display(temp->rc,x+d,y+50,d/2,x,y);
    }
}
```

Definition Of Binary Search Tree

- Is a binary tree, which may be empty. If it is not empty then it satisfies the following properties:
 - Every element (node) has a (key, value) pair. No two elements have the same key - keys are therefore distinct
 - The *keys in the left subtree* are smaller than those in the right subtree
 - The *keys in the right subtree* are larger than those in the left subtree
 - The left and right subtrees are also BSTs

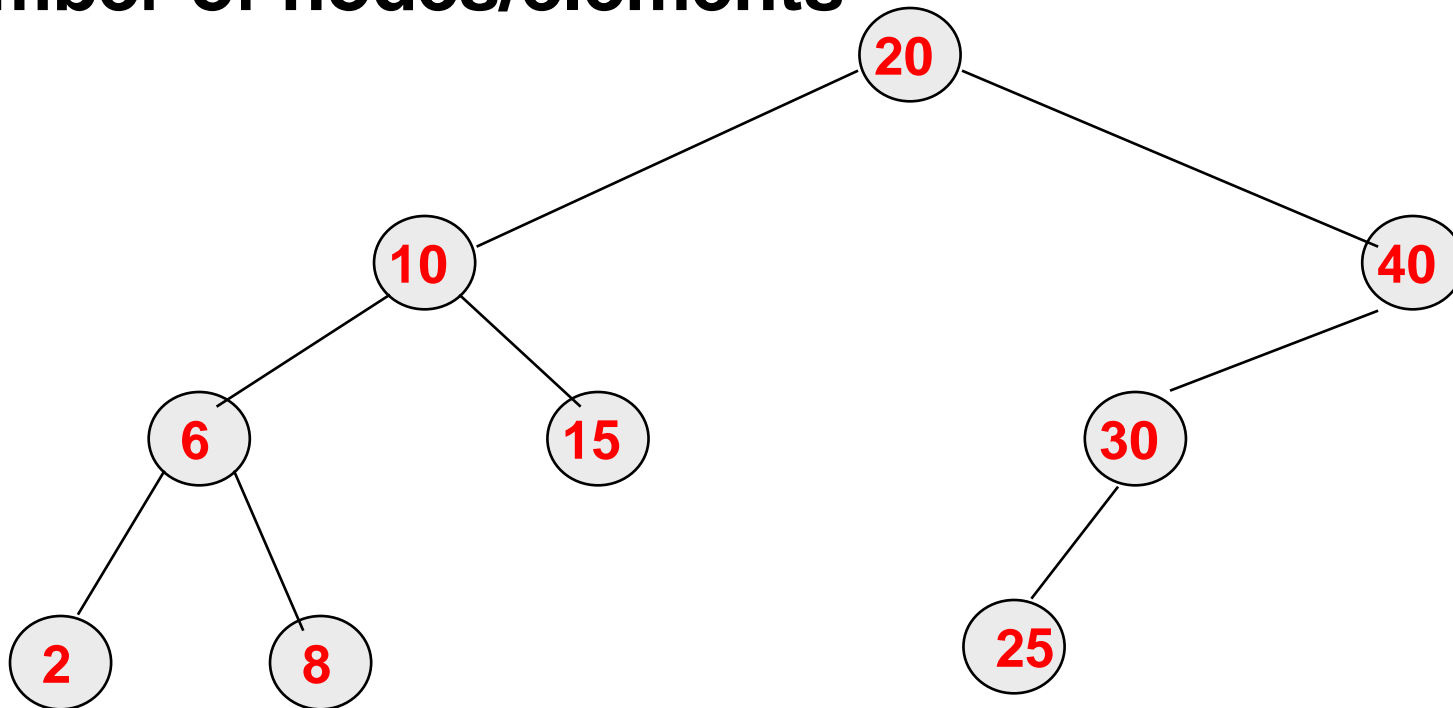
Example Binary Search Tree

- Only keys are shown

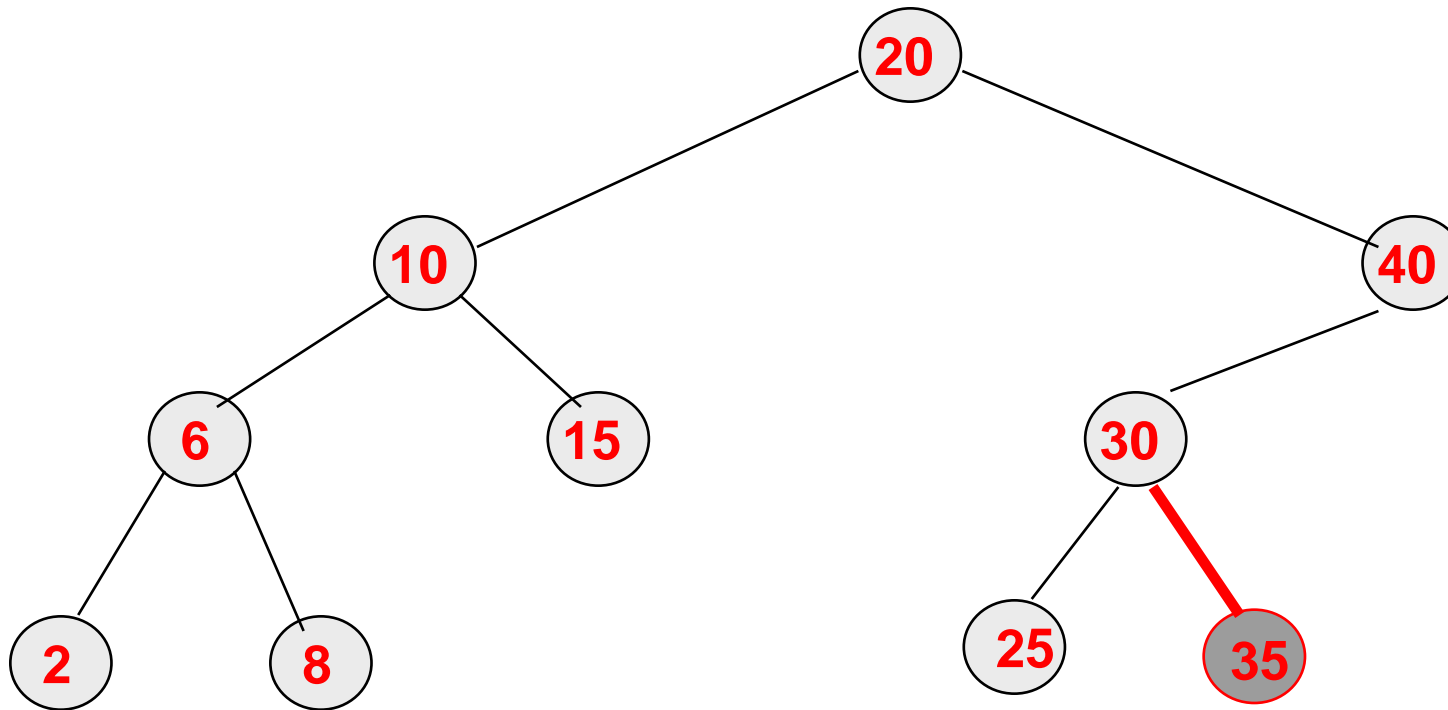


BST Operations

- `ascend()` : Do an inorder traversal. $O(n)$ time
- `get()` : Complexity is $O(\text{height}) = O(n)$, where n is number of nodes/elements

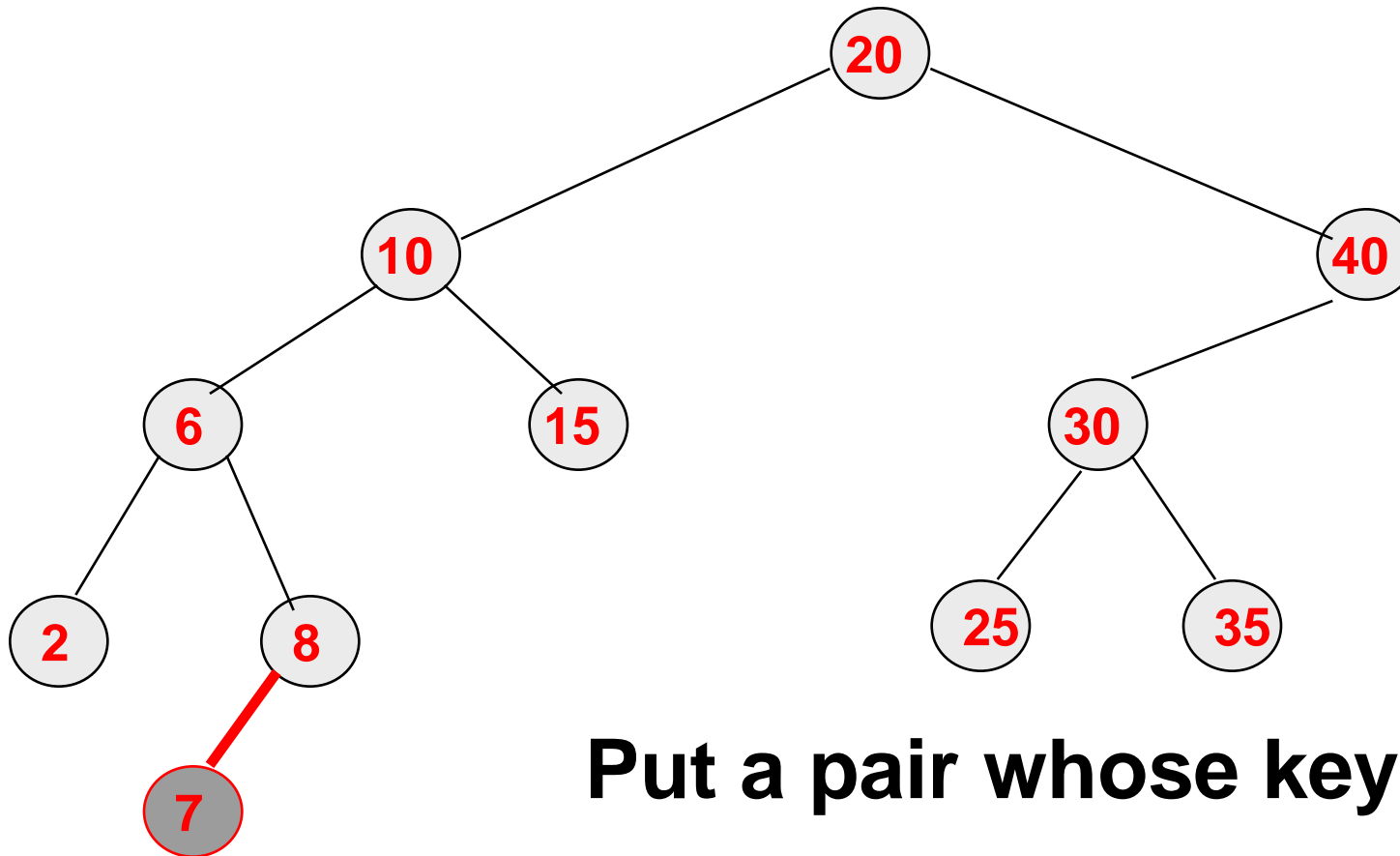


BST Operation put()



Put a pair whose key is 35

BST Operation put()



Put a pair whose key is 7

Complexity of put() is $O(\text{height})$

Operations on Trees

- **Inserting an element in the tree**
- **Deleting an element from the tree**
- **Computing the height of the tree**
- **Traversing the tree**
- **Levelwise printing of tree**
- **Copying a tree**
- **Comparing two trees**
- **Mirroring a given tree**
- **Counting the total nodes in a tree**
- **Counting the leaf nodes of a tree.**

Level wise printing

```
Void levelwiseprinting_bst(bst *root)
```

```
{ initialize front to 0 & rear to -1
```

```
  Initialize level to 1
```

```
  Display "\nLEVEL 1 : ";
```

```
  insertQ(root);
```

```
  insertQ(NULL);
```

```
  while(front != rear)
```

```
  {
```

```
    temp = deleteQ();
```

```
    if(temp != NULL)
```

```
    {
```

```
      Display temp->data;
```

```
      if(temp->lc != NULL)
```

```
        insertQ(temp->lc);
```

```
      if(temp->rc != NULL)
```

```
        insertQ(temp->rc);
```

```
    }
```

```
  else
```

```
  {
```

```
    level++;
```

```
    Display "\nLEVEL %d",level;
```

```
    insertQ(NULL);
```

```
  }
```

```
}
```

```
}
```

Mirror of Tree

```
void mirror(bst *temp)
{
    if(temp!= NULL)
    {
        bst *node;
        node = temp->lc;
        temp->lc = temp->rc;
        temp->rc = node;
        mirror(temp->lc);
        mirror(temp->rc);
    }
}
```

Copy

```
bst * copy_tree(bst *temp)
{
    if(temp != NULL)
    {
        bst *node = getnode(temp->data);
        node->lc = copy_tree(temp->lc);
        node->rc = copy_tree(temp->rc);
        return(node);
    }
    return(NULL);
}
```

height

```
int height(bst *temp)
{
    if(temp != NULL)
    {
        ht_lc = height(temp->lc);
        ht_rc = height(temp->rc);
        return (max(ht_lc,ht_rc) + 1);
    }
    return 0;
}
```

Height non recursive

```
Void levelwiseprinting_bst(bst *root)
```

```
{ initialize front to 0 & rear to -1
```

```
  Initialize level to 1
```

```
  Display "\nLEVEL 1 : ";
```

```
  insertQ(root);
```

```
  insertQ(NULL);
```

```
  while(front != rear)
```

```
  {
```

```
    temp = deleteQ();
```

```
    if(temp != NULL)
```

```
    {
```

```
      Display temp->data;
```

```
      if(temp->lc != NULL)
```

```
        insertQ(temp->lc);
```

```
      if(temp->rc != NULL)
```

```
        insertQ(temp->rc);
```

```
    }
```

```
  else
```

```
  {
```

```
    level++;
```

```
    Display "\nLEVEL %d",level;
```

```
    insertQ(NULL);
```

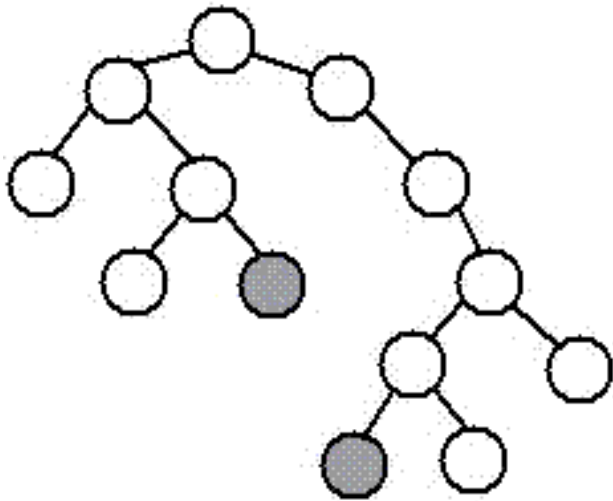
```
  }
```

```
}
```

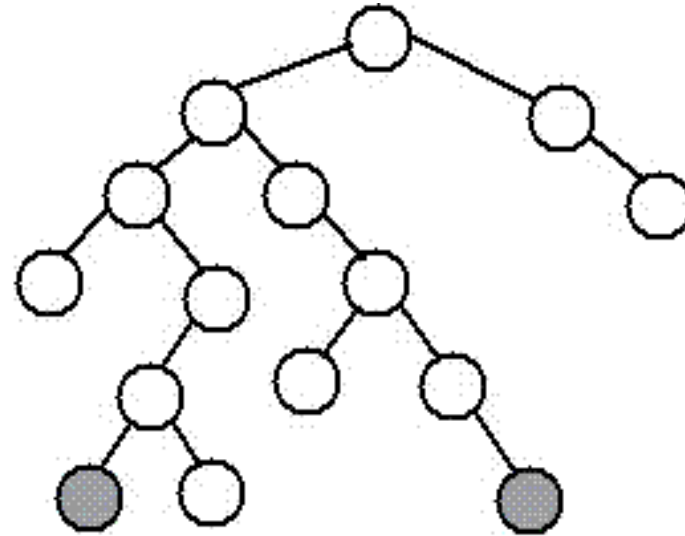
```
}
```

Diameter of a Tree

- The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two leaves in the tree.
- The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



diameter, 9 nodes, through root



diameter, 9 nodes, NOT through root

Diameter of a Tree

- The diameter of a tree T is the largest of the following quantities:
 - the diameter of T 's left subtree
 - the diameter of T 's right subtree
 - the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

Diameter of a Tree

```
int diameter(btree * node)
{

    if (node != NULL)
    {
        lht = height(node->lc);
        rht = height(node->rc);
        ldm = diameter(node->lc);
        rdm = diameter(node->rc);
        return max(lht + rht + 1, max(ldm, rdm));
    }
    else
        return 0;
}
```

Delete operation

- **4 Cases**

- **The node to be deleted is :**

- ☐ **Leaf node**
- ☐ **Has only left subtree**
- ☐ **Has only right subtree**
- ☐ **Has both left and right subtrees**

Leaf node

// case 1 leaf node

if(temp->lc == NULL && temp->rc == NULL) // leaf node

{

if(temp == root)

root = NULL;

else

{

if(par->lc == temp)

par->lc = NULL;

else

par->rc = NULL;

}

}

Only left subtree

```
if(temp->lc != NULL && temp->rc == NULL) // only left subtree
{
    if(temp == root)
        root = temp->lc;
    else
    {
        if(par->lc == temp)
            par->lc = temp->lc;
        else
            par->rc = temp->lc;
    }
}
```

Only right subtree

```
if(temp->lc == NULL && temp->rc != NULL)
{
    if(temp == root)
        root = temp->rc;
    else
    {
        if(par->lc == temp)
            par->lc = temp->rc;
        else
            par->rc = temp->rc;
    }
}
```

Both subtrees

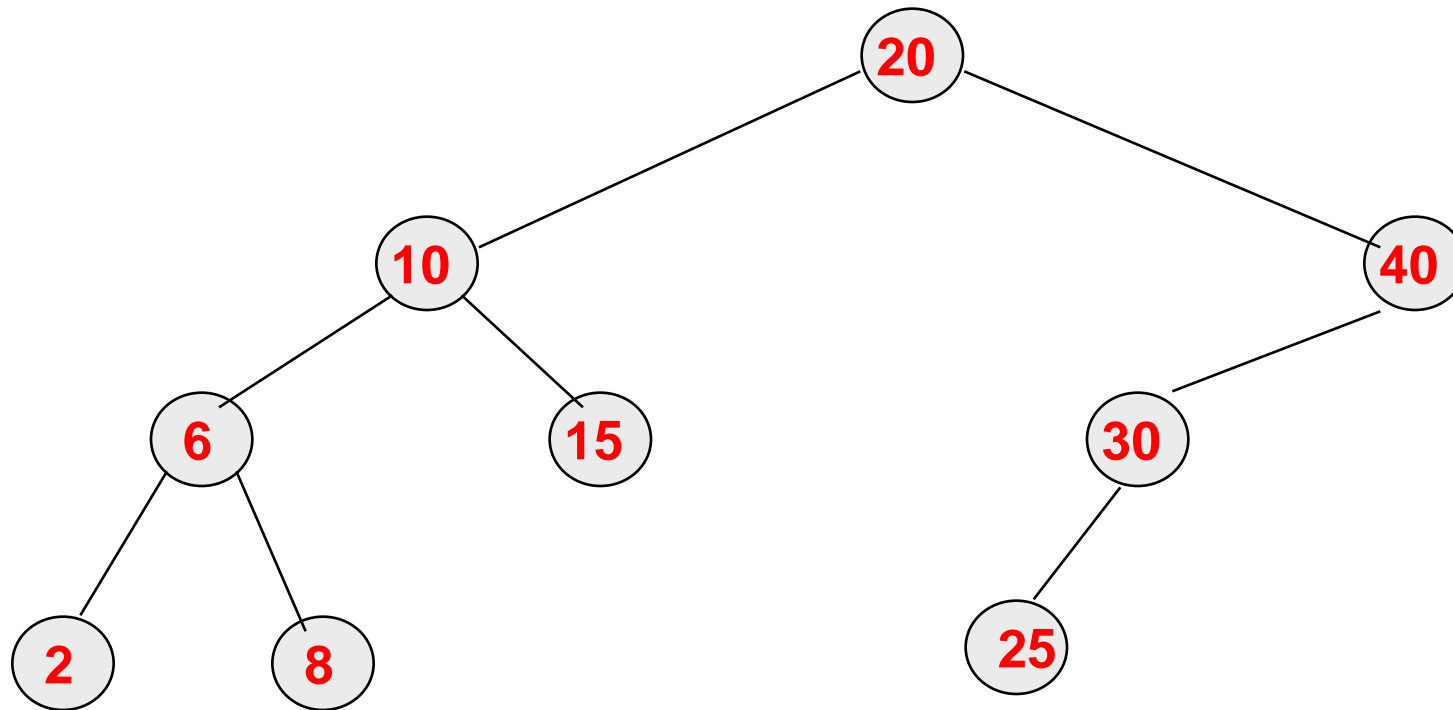
```
if(temp->lc != NULL && temp->rc != NULL) //both subtrees
{
    bst *node = temp->rc;
    while(node->lc != NULL)
        node = node->lc;
    node->lc = temp->lc;
    if(temp == root)
        root = temp->rc;
    else
    {
        if(par->lc == temp)
            par->lc = temp->rc;
        else
            par->rc = temp->rc;
    }
}
```

Definition Of Binary Search Tree

- Is a binary tree, which may be empty. If it is not empty then it satisfies the following properties:
 - Every element (node) has a (key, value) pair. No two elements have the same key - keys are therefore distinct
 - The *keys in the left subtree* are smaller than those in the right subtree
 - The *keys in the right subtree* are larger than those in the left subtree
 - The left and right subtrees are also BSTs

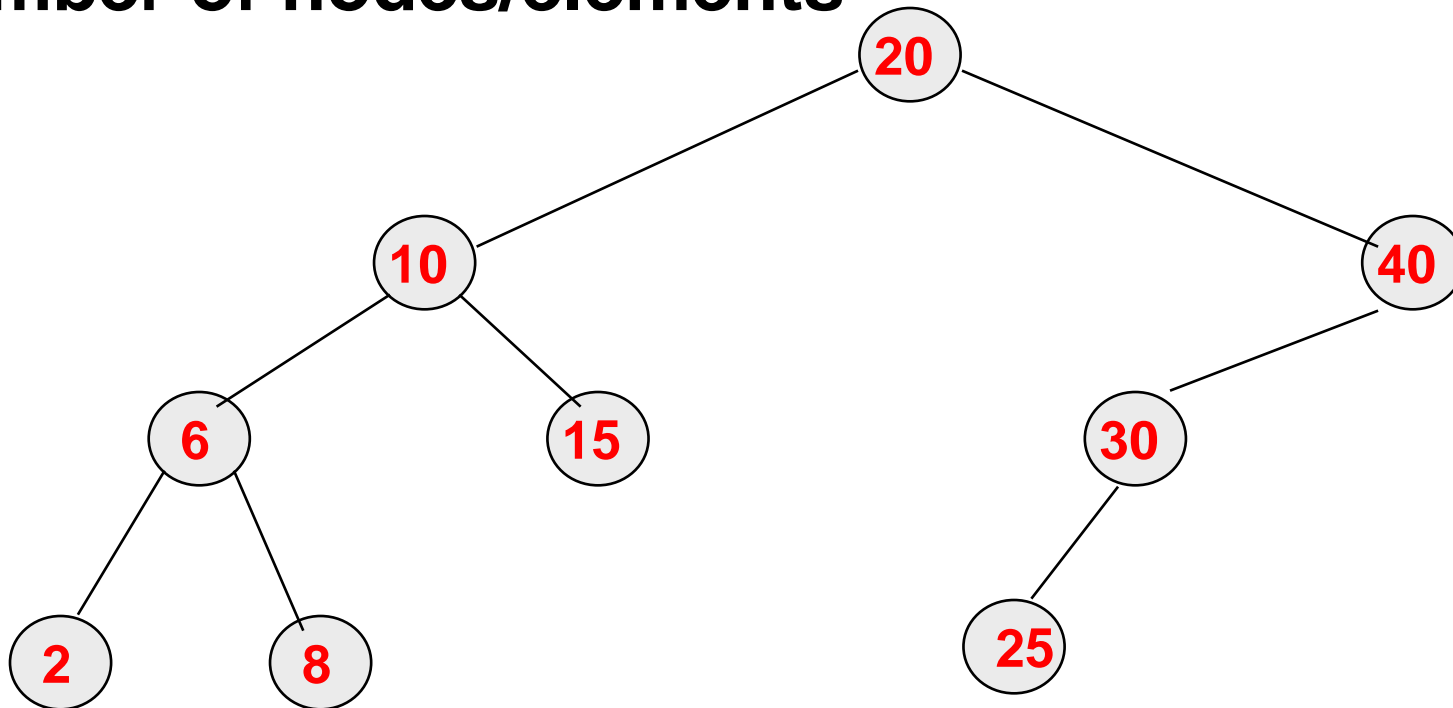
Example Binary Search Tree

- Only keys are shown

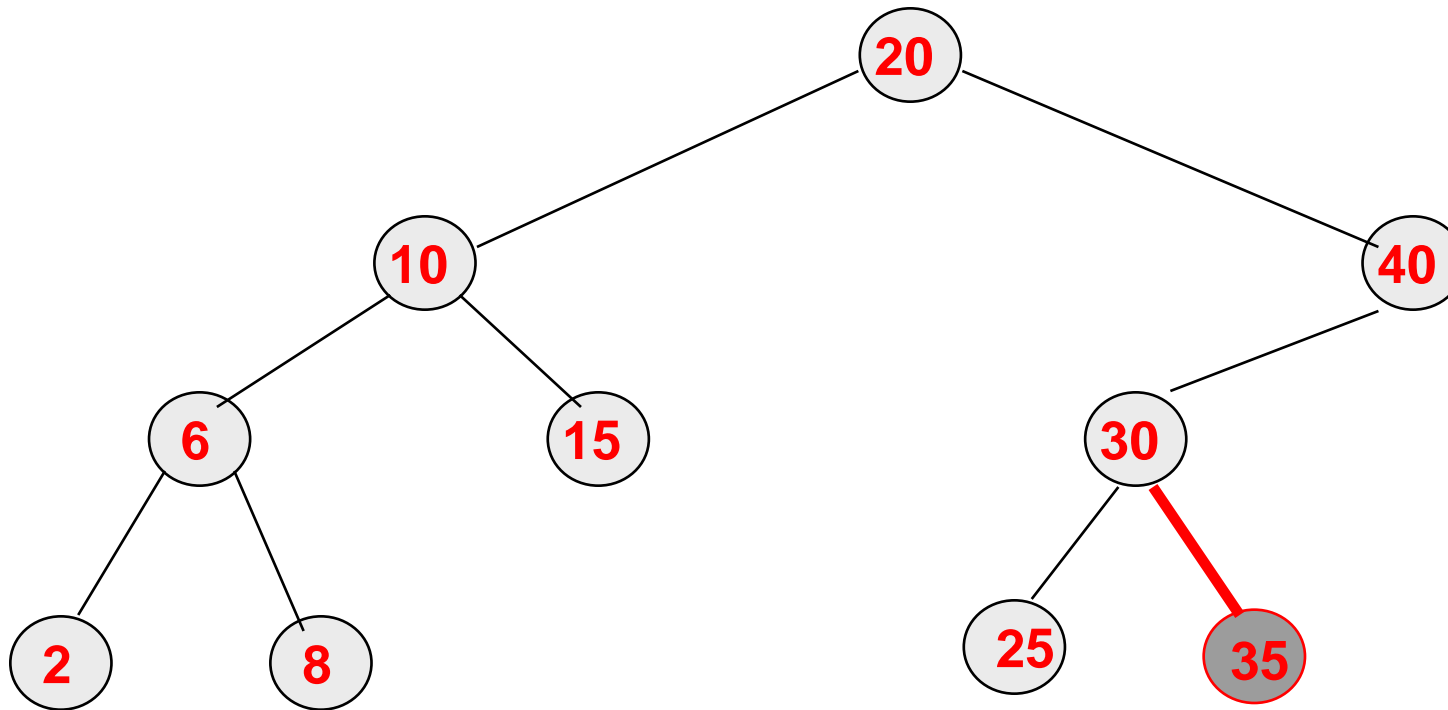


BST Operations

- `ascend()` : Do an inorder traversal. $O(n)$ time
- `get()` : Complexity is $O(\text{height}) = O(n)$, where n is number of nodes/elements

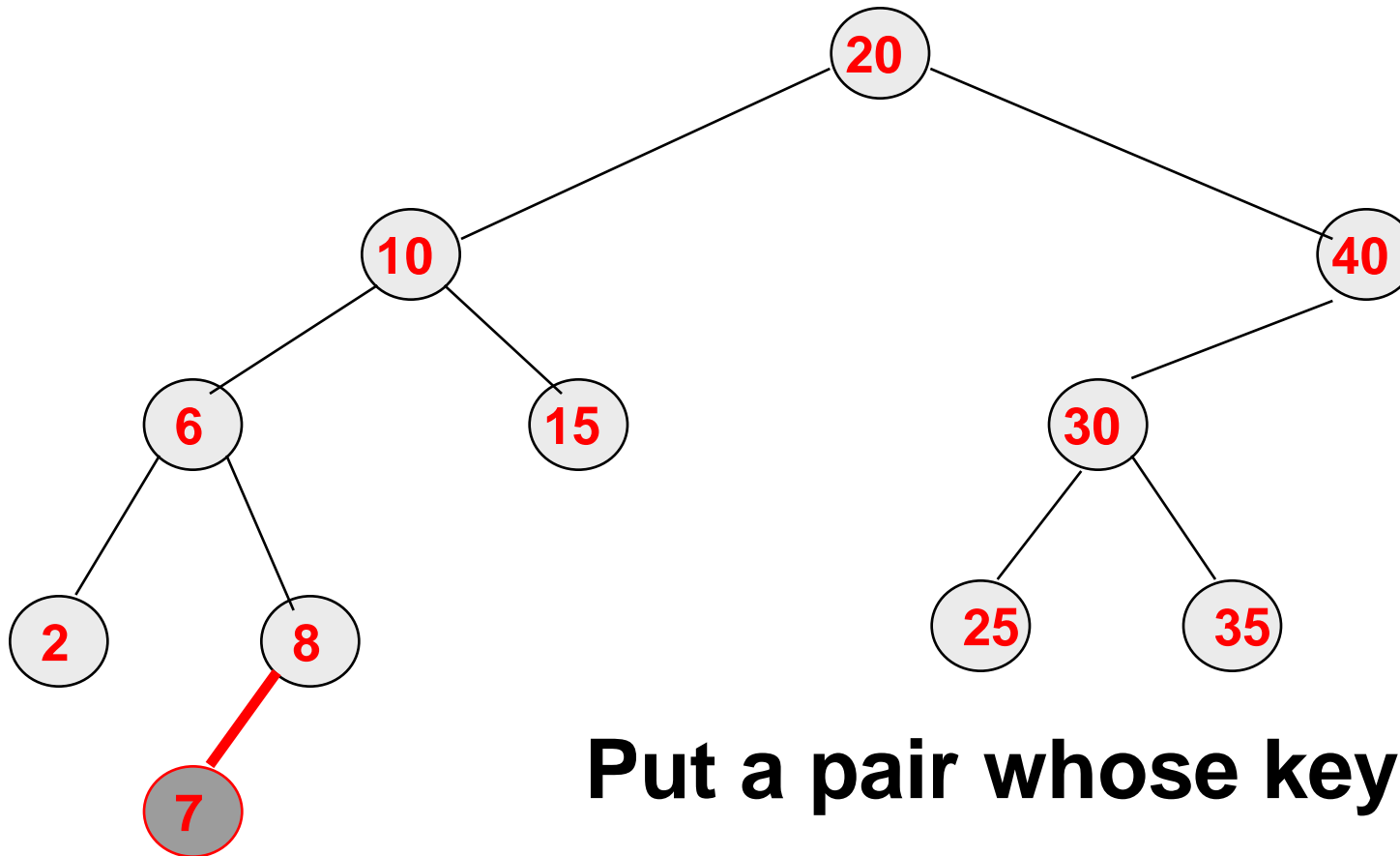


BST Operation put()



Put a pair whose key is 35

BST Operation put()



Put a pair whose key is 7

Complexity of put() is $O(\text{height})$

Operations on Trees

- **Inserting an element in the tree**
- **Deleting an element from the tree**
- **Computing the height of the tree**
- **Traversing the tree**
- **Levelwise printing of tree**
- **Copying a tree**
- **Comparing two trees**
- **Mirroring a given tree**
- **Counting the total nodes in a tree**
- **Counting the leaf nodes of a tree.**

General Trees

❑ **Trees in which the number of subtrees of any node need not be required to be 0, 1, or 2.**

❑ **The number of subtrees for any node may be variable.**

Some nodes may have 1 or no subtrees, others may have 3, some 4, or any other combination.

❑ **Ternary tree – 3 subtrees per node**

Problems with General Trees and Solutions

❑ Problems

- ✓ The number of references for each node must be equal to the maximum that will be used in the tree.
- ✓ Most of the algorithms for searching, traversing, adding and deleting nodes become much more complex in that they must now cope with situations where there are not just two possibilities for any node but multiple possibilities.

❑ Solution

- ✓ General trees can be converted to binary trees
- ✓ The algorithms that are used for binary tree processing can be used with minor modifications.

Creating a Binary Tree from a General Tree

☐ Conversion process

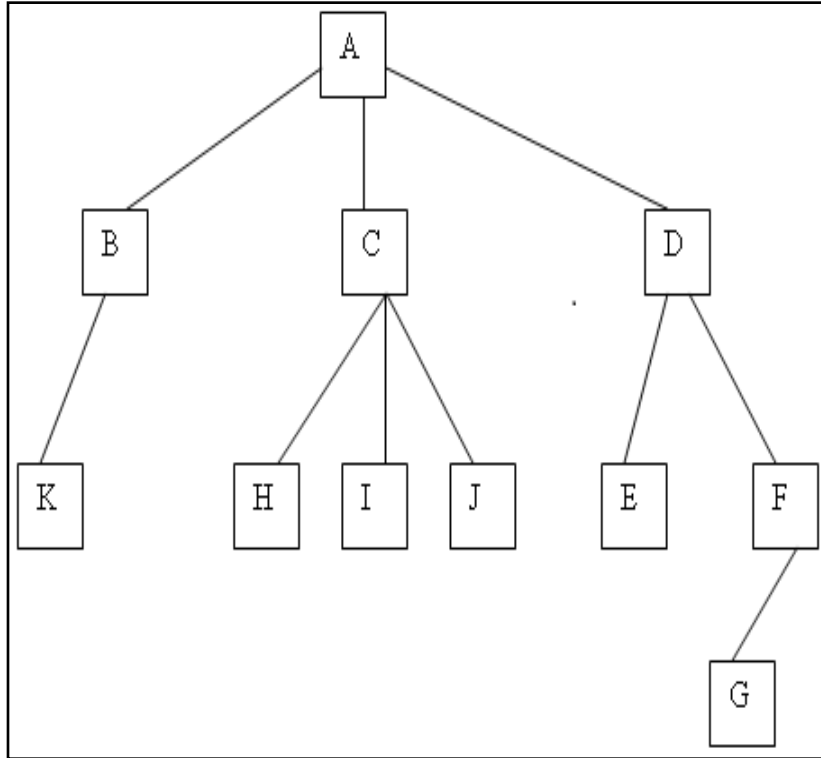
- ✓ Use the root of the general tree as the root of the binary tree.
- ✓ Determine the first child of the root. This is the leftmost node in the general tree at the next level.
- ✓ Insert this node. The child reference of the parent node refers to this node.
- ✓ Continue finding the first child of each parent node and insert it below the parent node with the child reference of the parent to this node.

Creating a Binary Tree from a General Tree

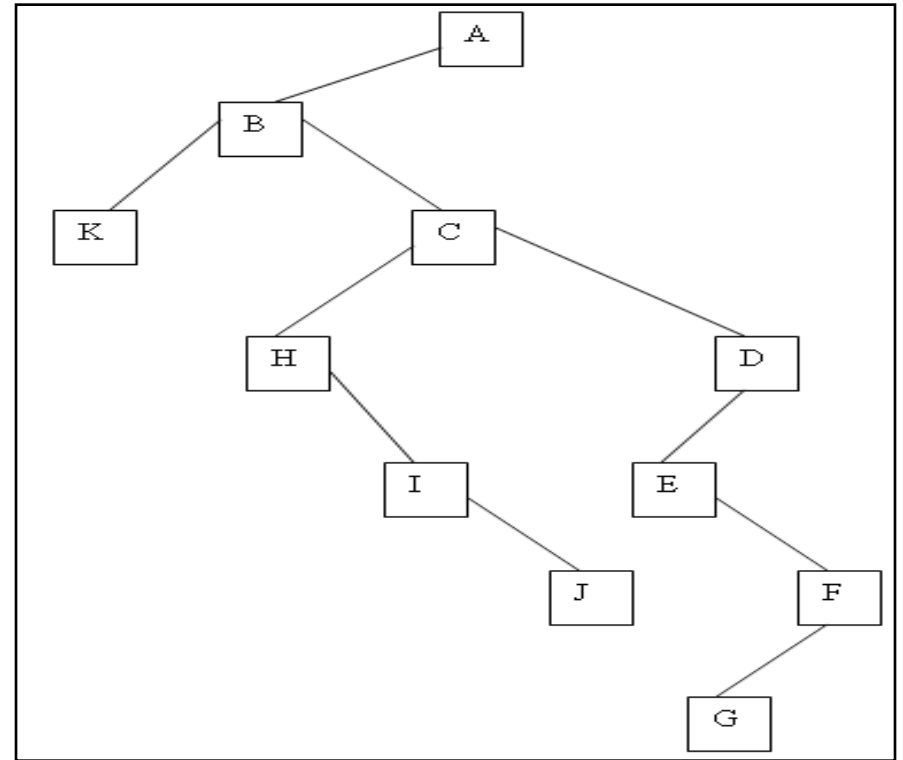
❑ Conversion process (continued)

- ✓ When no more first children exist in the path just used, move back to the parent of the last node entered and repeat the above process. In other words, determine the first sibling of the last node entered.
- ✓ Complete the tree for all nodes. In order to locate where the node fits you must search for the first child at that level and then follow the sibling references to a nil where the next sibling can be inserted. The children of any sibling node can be inserted by locating the parent and then inserting the first child. Then the above process is repeated.

Creating a Binary Tree from a General Tree



General Tree



Binary Tree

Traversing a Tree

- ✓ When the general tree has been represented as a binary tree, the algorithms which were used for the binary tree can now be used for the general tree.
- ✓ In-order traversals make no sense when a general tree is converted to a binary tree. In the general tree each node can have more than two children so trying to insert the parent node in between the children is rather difficult, especially if there is an odd number of children.

Pre – order

- ✓ This is a process where the root is accessed and processed and then each of the subtrees is preorder processed. It is also called a **depth-first traversal**.

Traversing a Tree

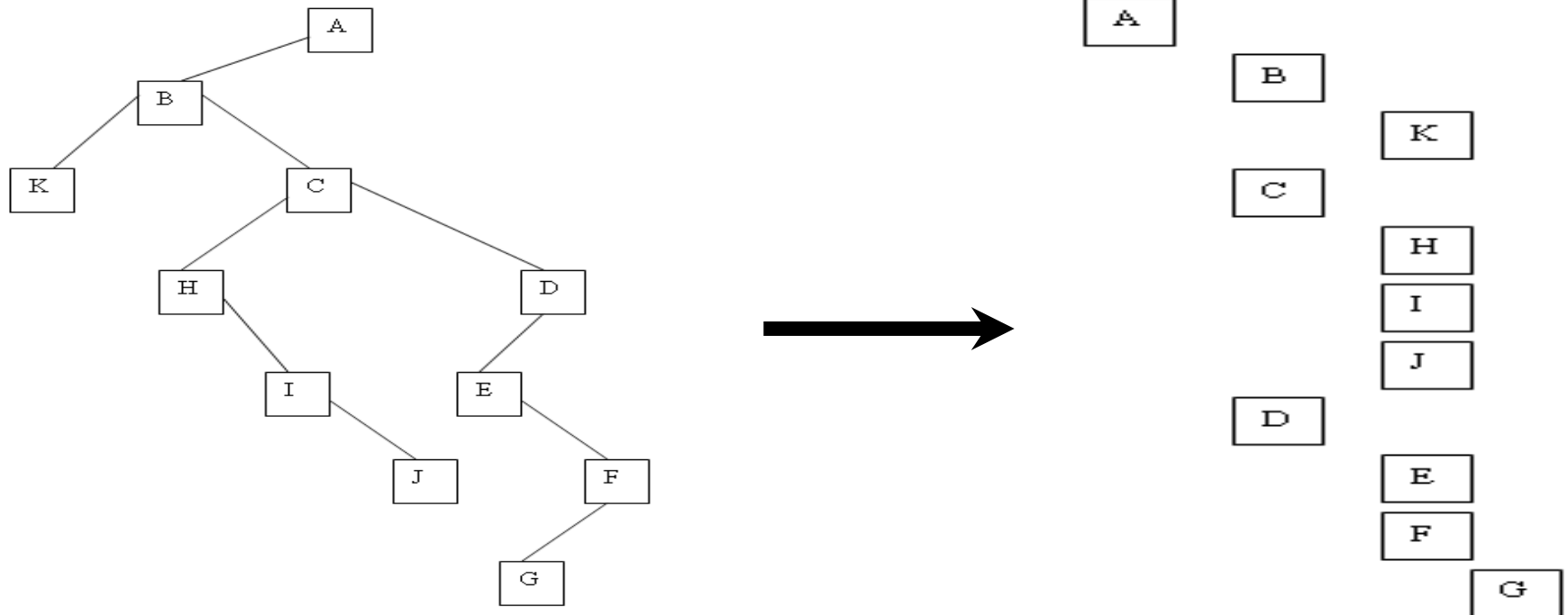
❑ Pre-order Traversal Steps

- ✓ Process the root node and move left to the first child.
- ✓ Each time the reference moves to a new child node, the print should be indented one tab stop and then the node processed.
- ✓ When no more first children remain then the processing involves the right sub-tree of the parent node. This is indicated by the nil reference to another first child but a usable reference to siblings. Therefore the first sibling is accessed and processed.
- ✓ If this node has any children they must be processed before moving on to other siblings. Therefore the number of tabs is increased by one and the siblings processed. If there are no children the processing continues through the siblings.
- ✓ Each time a sibling list is exhausted and a new node is accessed the number of tab stops is decreased by one.

Traversing a Tree

❑ Pre-order Traversal

- ✓ In this way the resulting printout has all nodes at any given level starting in the same tab column.
- ✓ It is relatively easy to draw lines to produce the original general tree except that the tree is on its side with it's root at the left rather than with the root at the top.



Result of pre-order traversal

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 Trees

- **Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers**
- **We can use these pointers to help us in inorder traversals**
- **We have the pointers reference the next node in an inorder traversal; called *threads***
- **We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer**

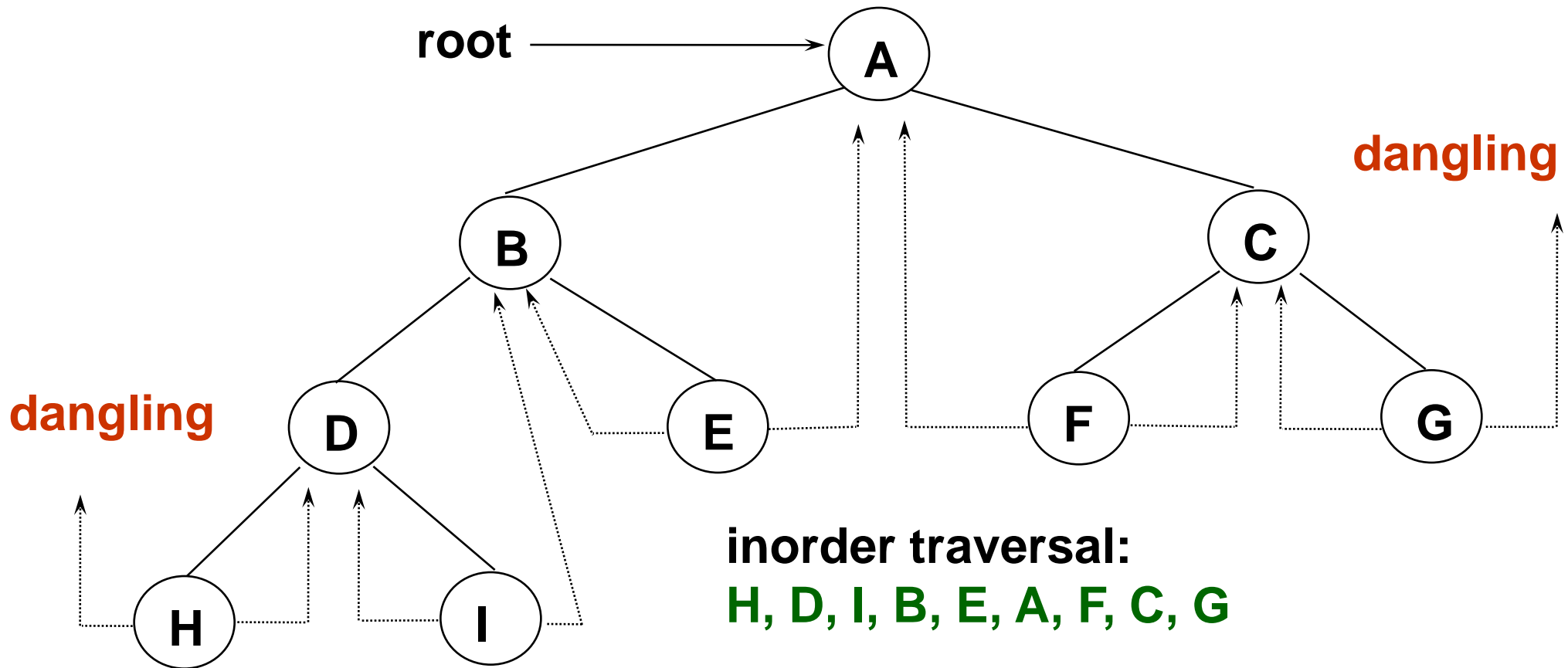
Types of threaded tree

- **Inorder Threaded Tree**
- **Preorder Threaded Tree**
- **Postorder Threaded Tree**

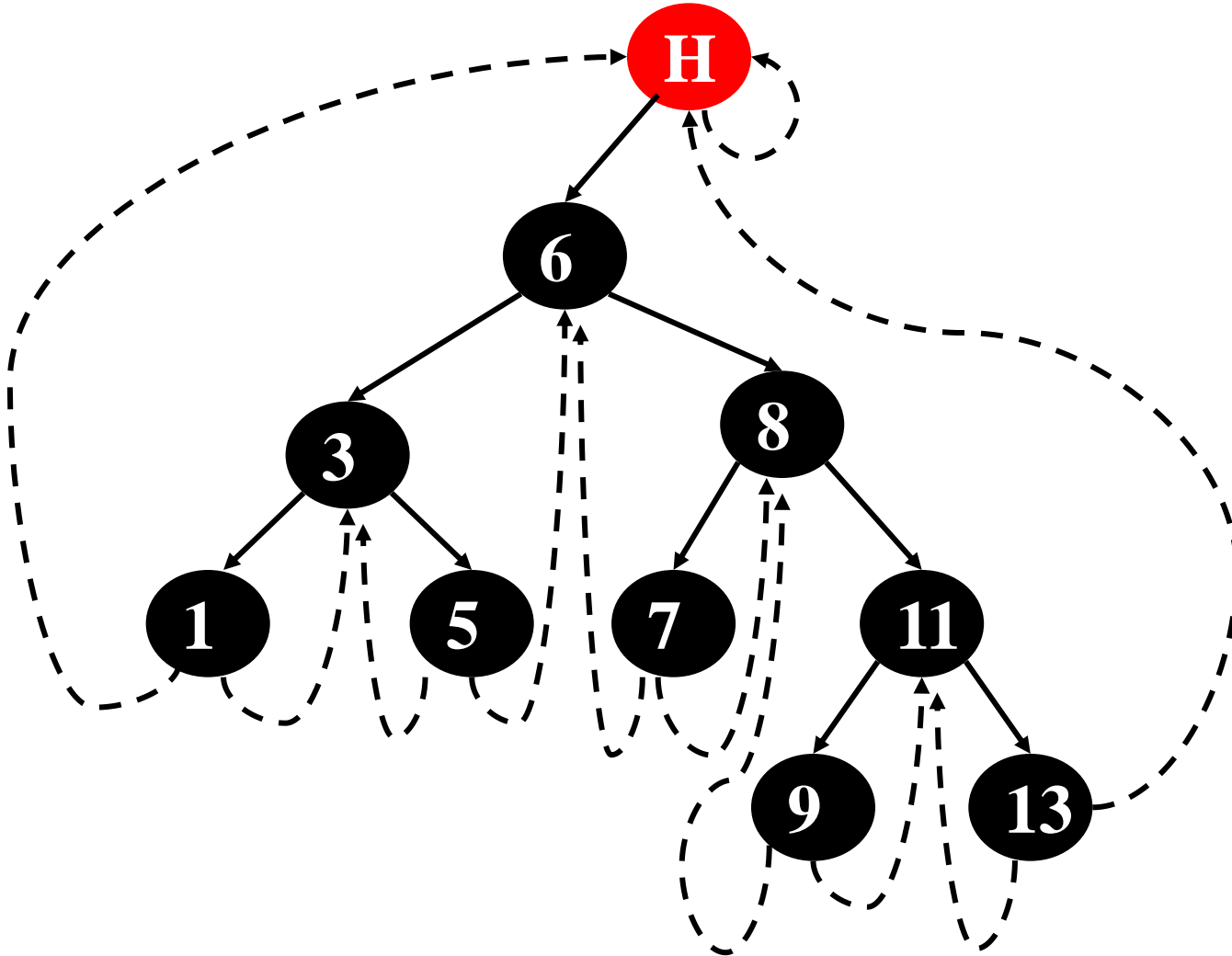
Inorder Threaded Binary Tree

- **When the left thread points to the inorder predecessor and right thread to inorder successor, the threading is known as “Inorder Threading”.**

Inorder Threaded Binary Tree



Inorder threaded tree example



Inorder Traversal : 1 3 5 6 7 8 9 11 13

Data Structures for Threaded BT

left_thread left_child data right_child right_thread

TRUE	●	—	●	FALSE
------	---	---	---	-------



TRUE: thread



FALSE: child

```
struct threaded_tree
```

```
{
```

```
    char left_thread;
```

```
    struct Threaded_tree *left_child;
```

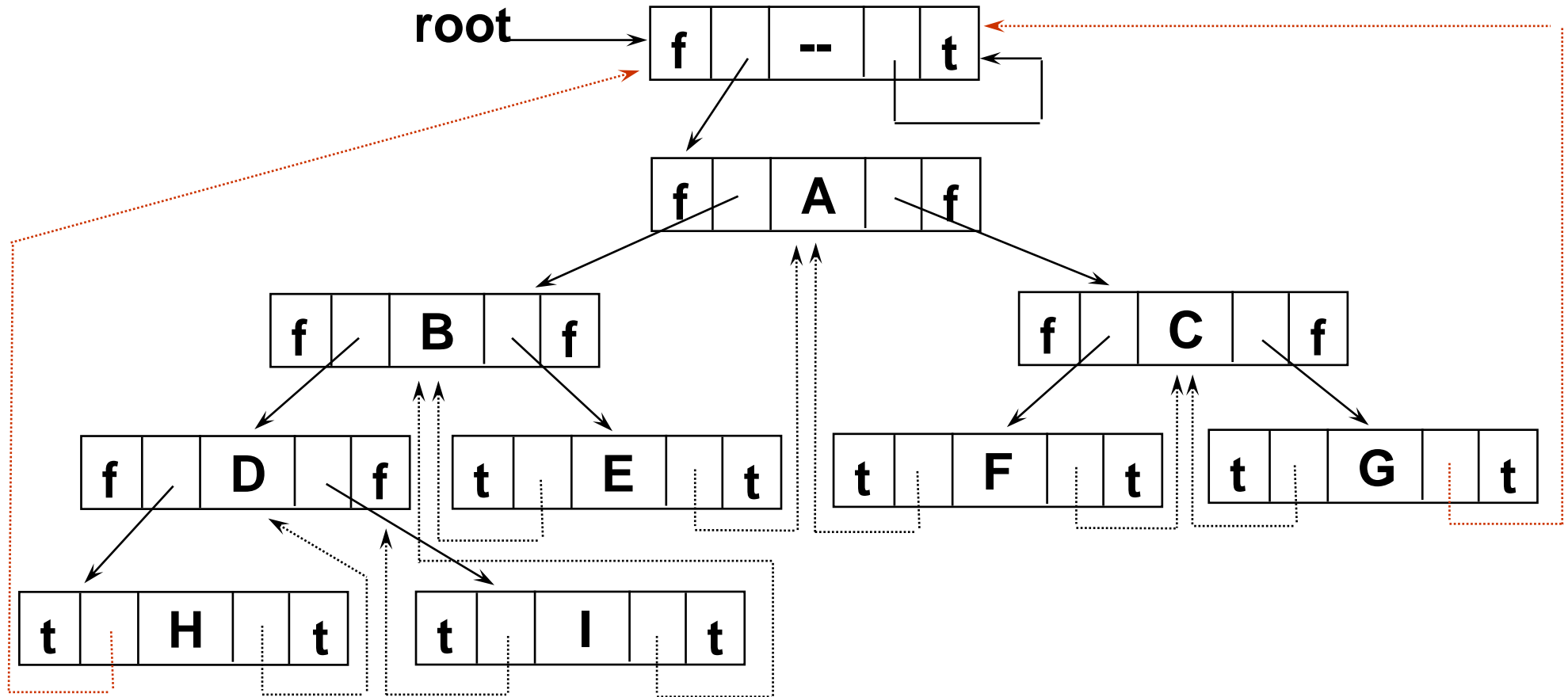
```
    char data;
```

```
    struct Threaded_tree *right_child;
```

```
    char right_thread;
```

```
};
```

Memory Representation of Threaded Binary Tree



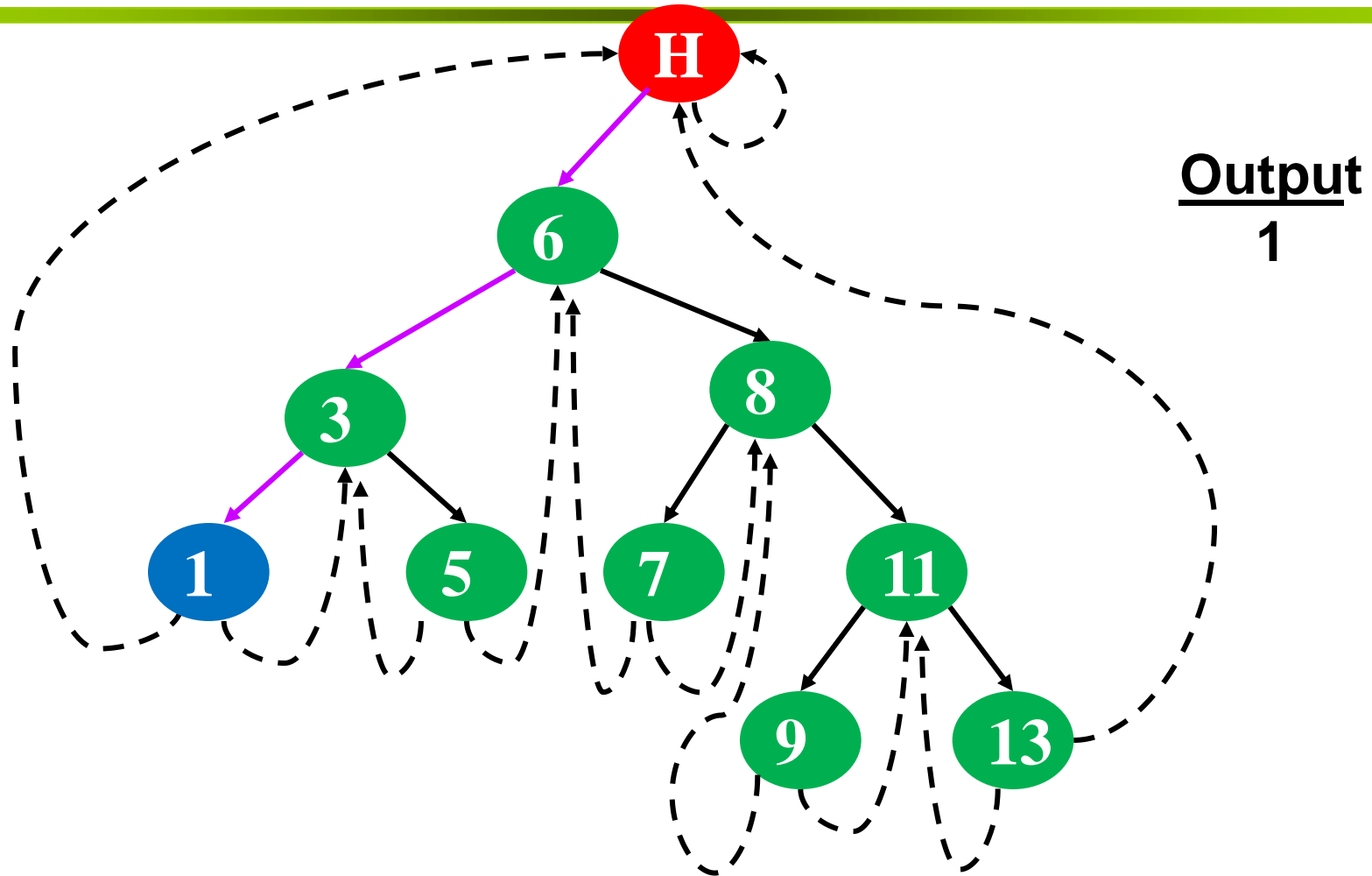
Ways of Creation

- **Create a simple binary tree and then thread it.**
- **During creation only, form these threads.**

Threaded Tree Traversal

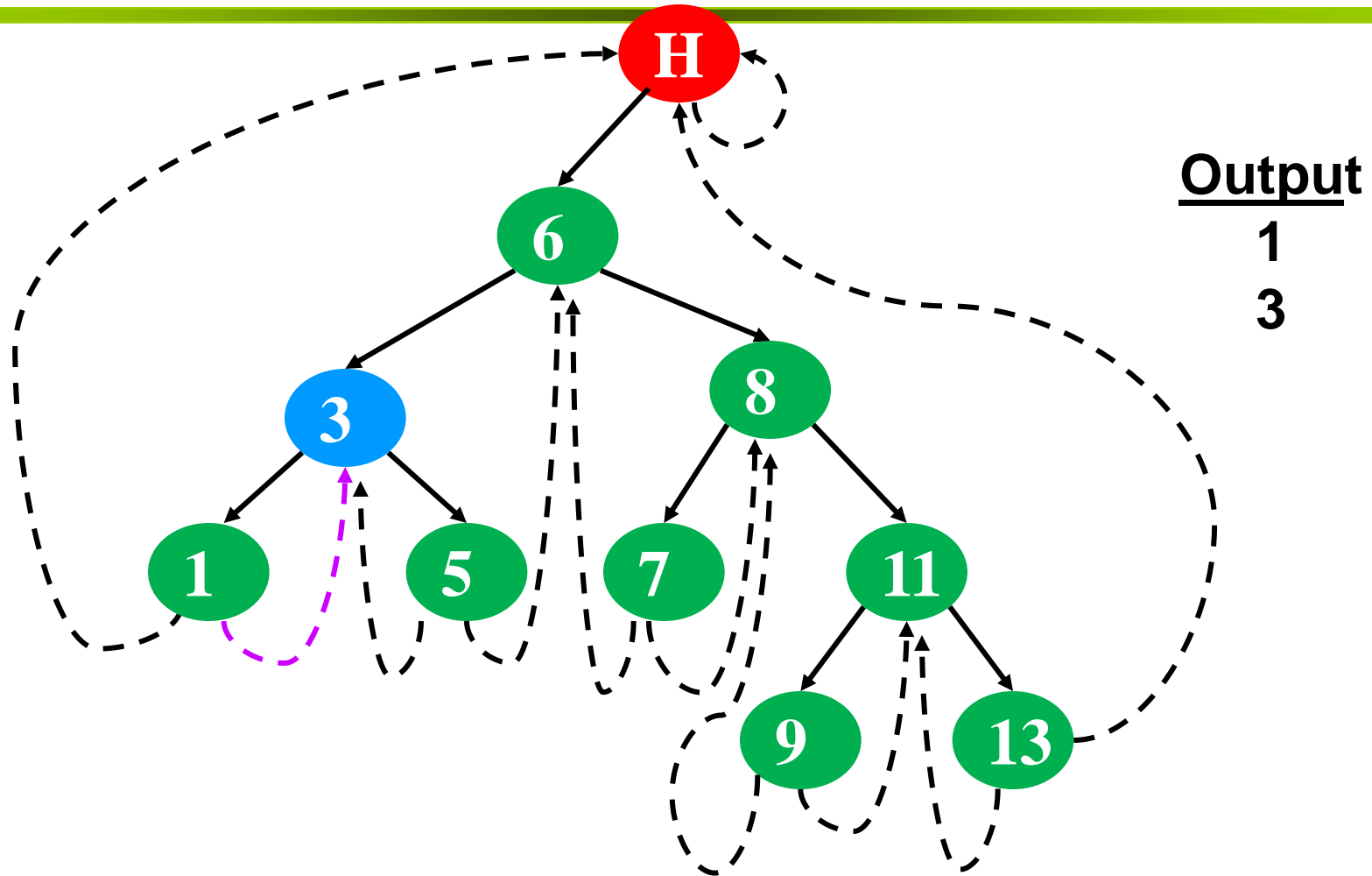
- **We start at the leftmost node in the tree, print it, and follow its right thread**
- **If we follow a thread to the right, we output the node and continue to its right**
- **If we follow a link to the right, we go to the leftmost node, print it, and continue**

Threaded Tree Inorder Traversal



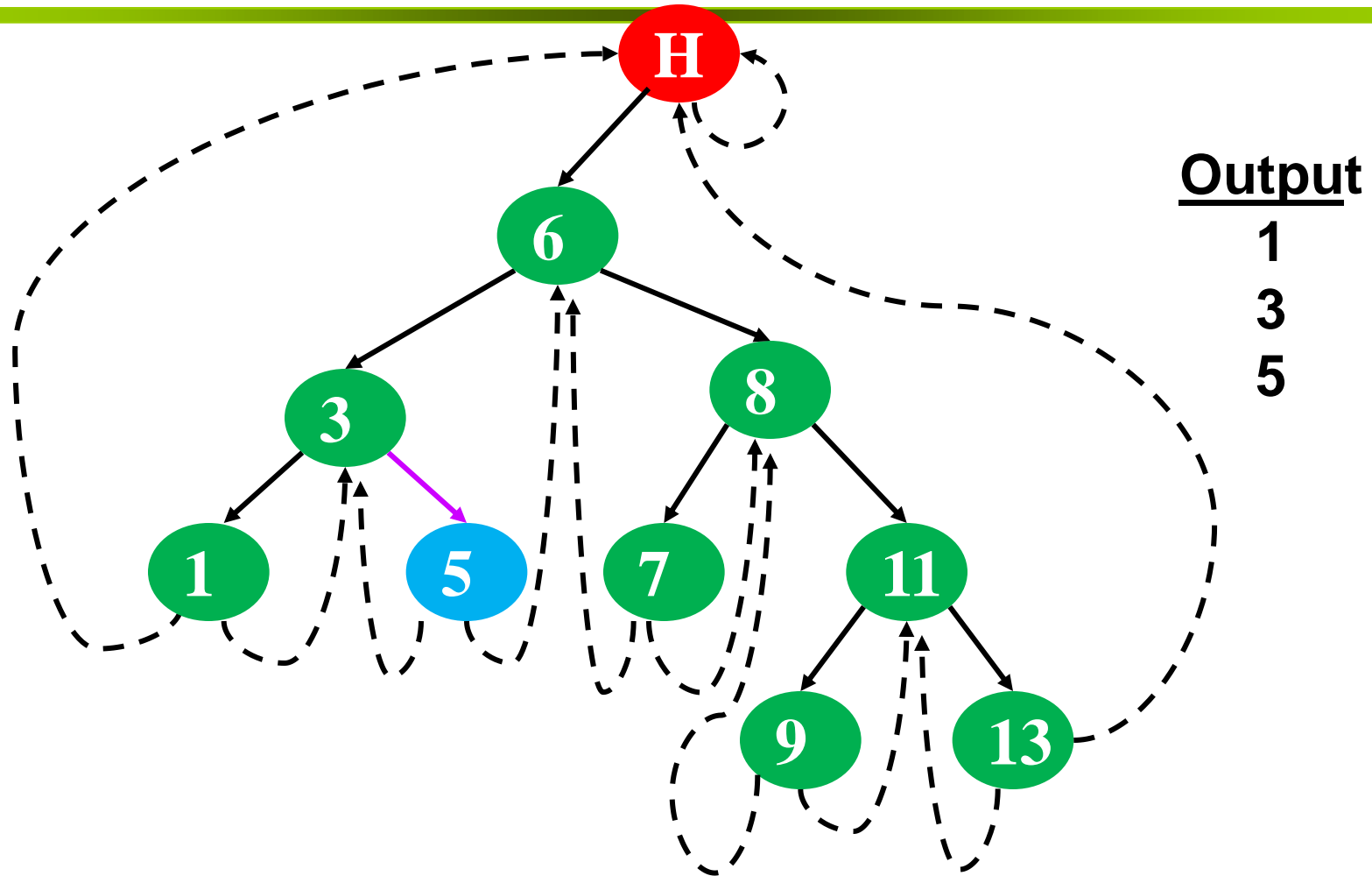
Start at leftmost node, print it

Threaded Tree Inorder Traversal



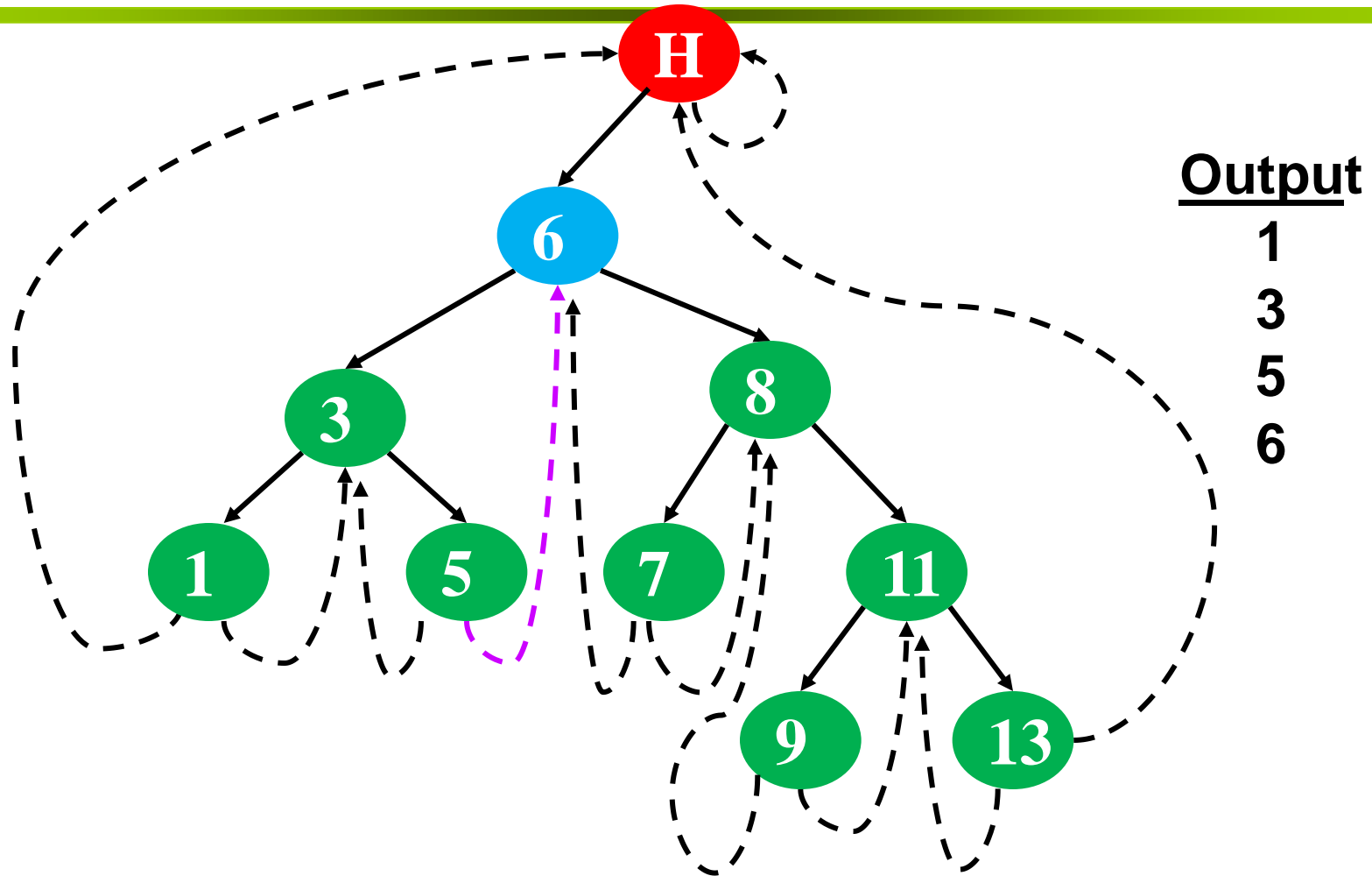
Follow thread to right, print node

Threaded Tree Inorder Traversal



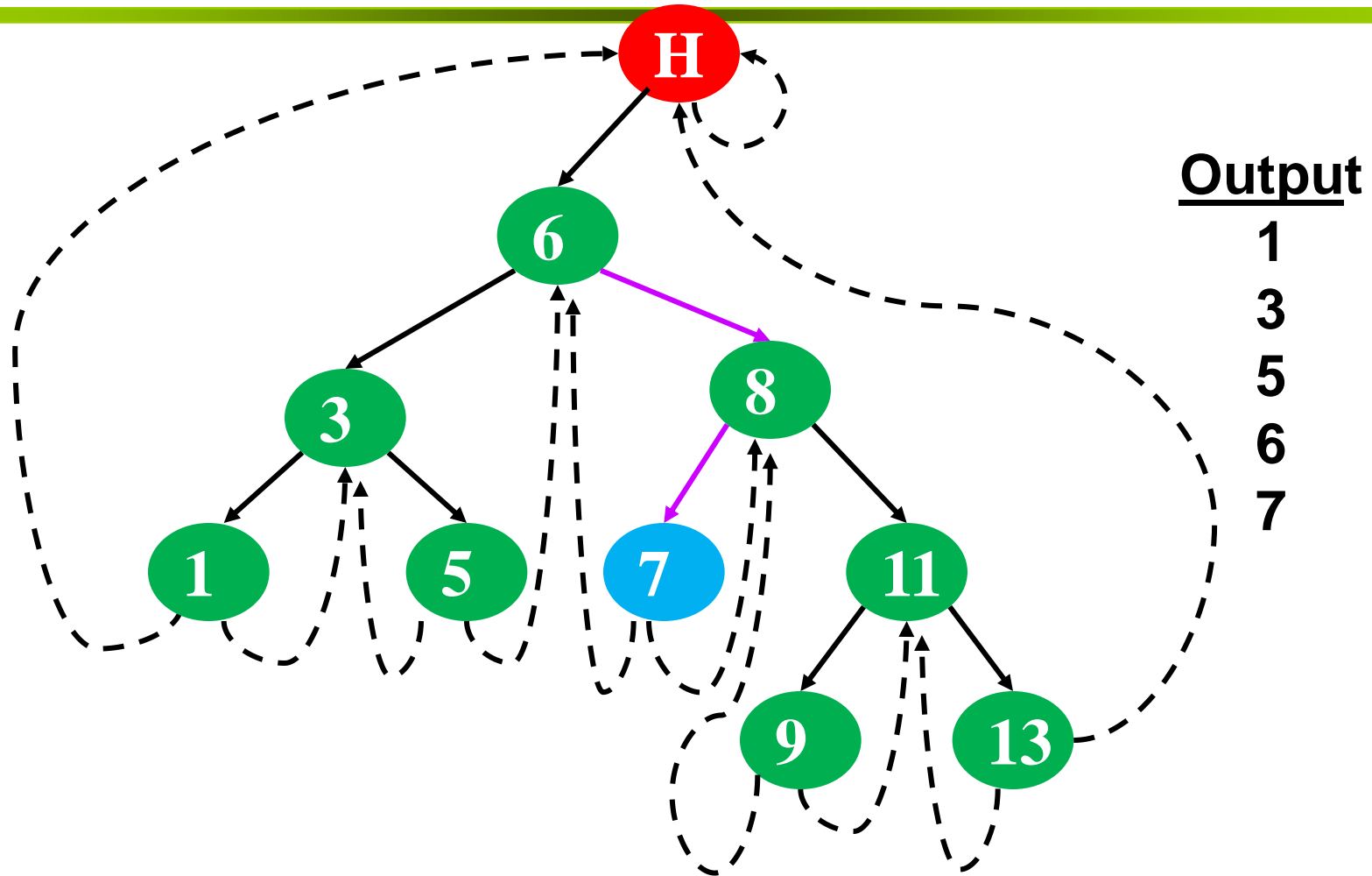
Follow link to right, go to
leftmost node and print

Threaded Tree Inorder Traversal



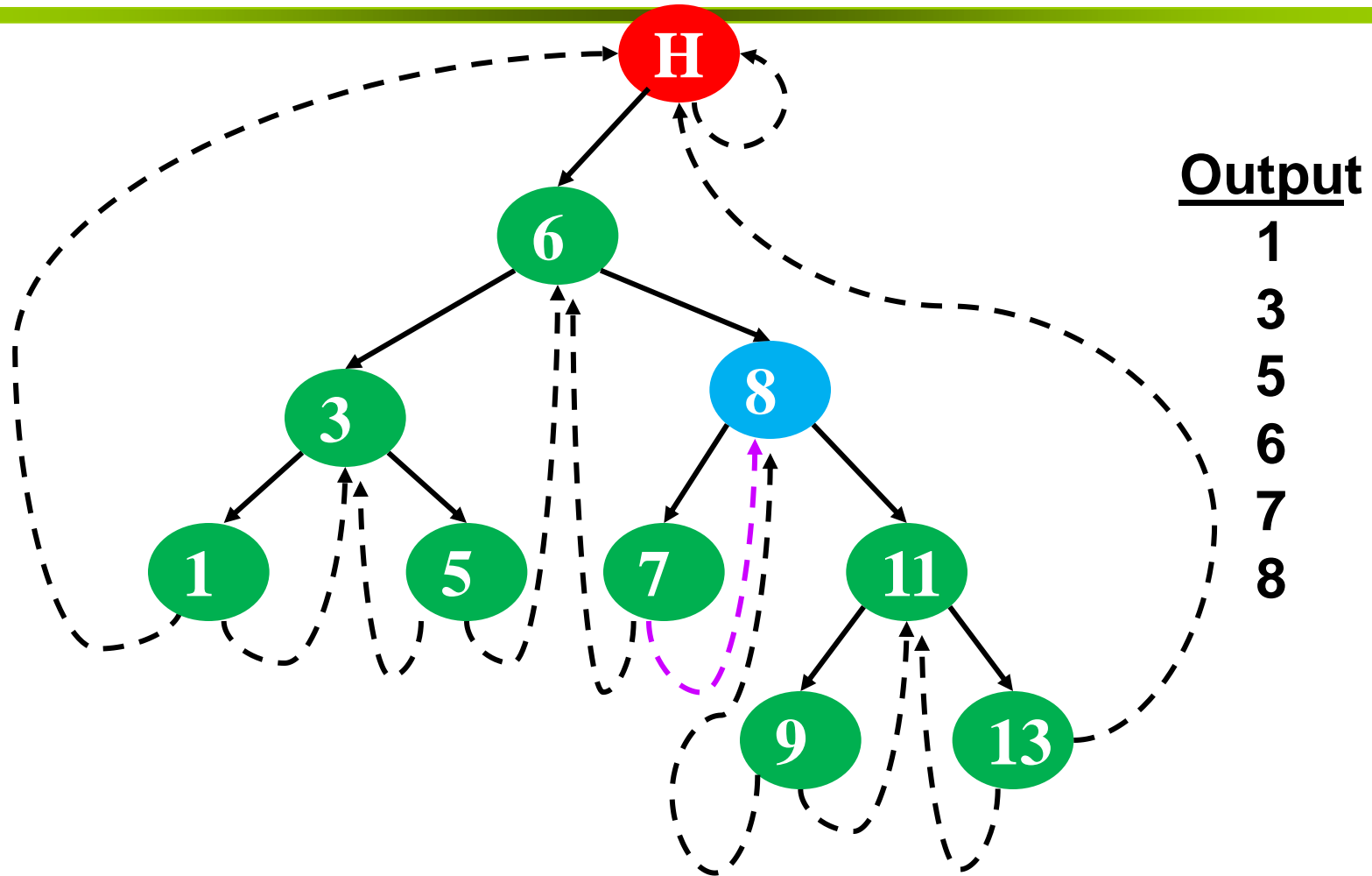
Follow thread to right, print node

Threaded Tree Inorder Traversal



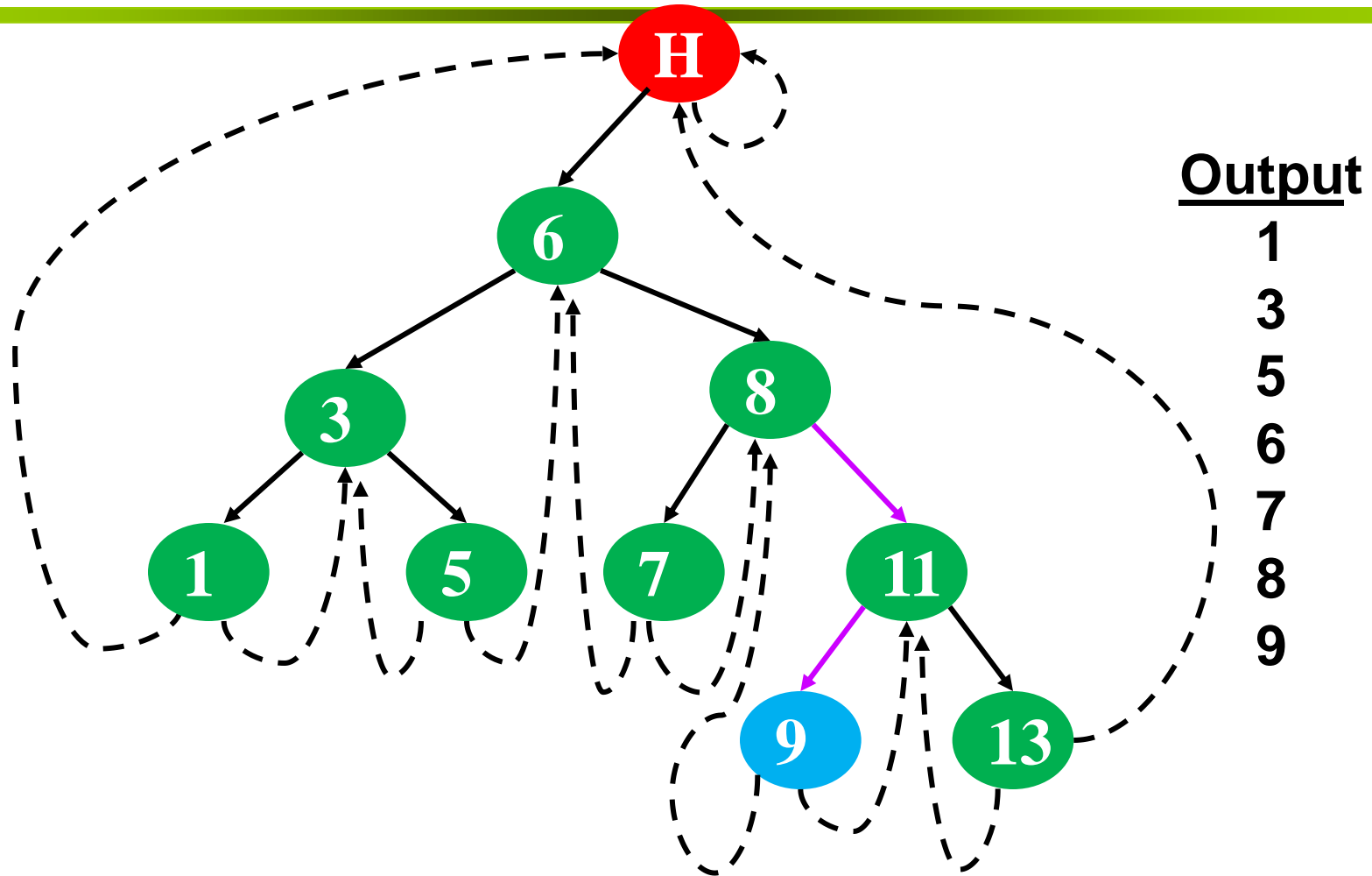
Follow link to right, go to
leftmost node and print

Threaded Tree Inorder Traversal



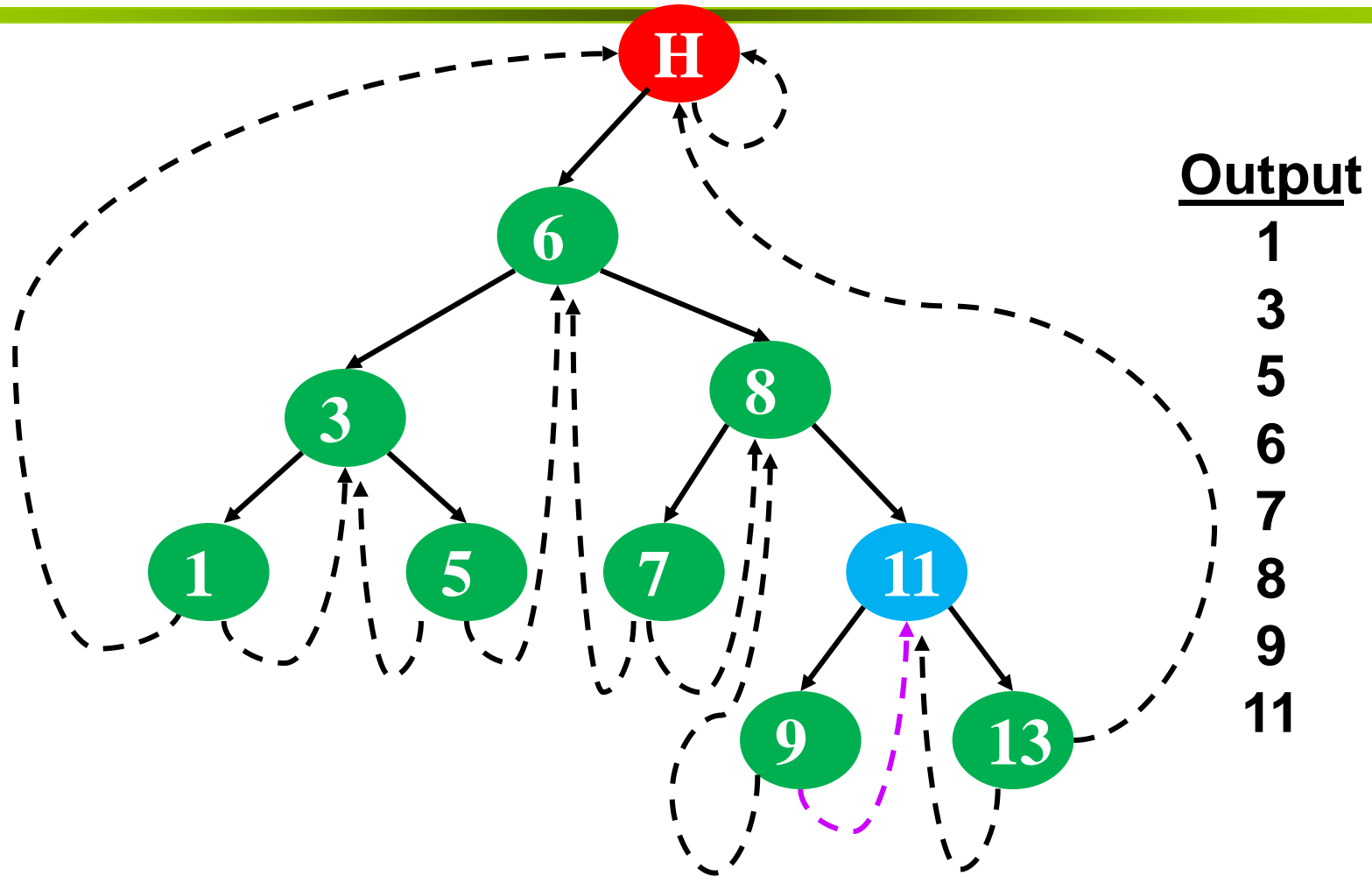
Follow thread to right, print node

Threaded Tree Inorder Traversal



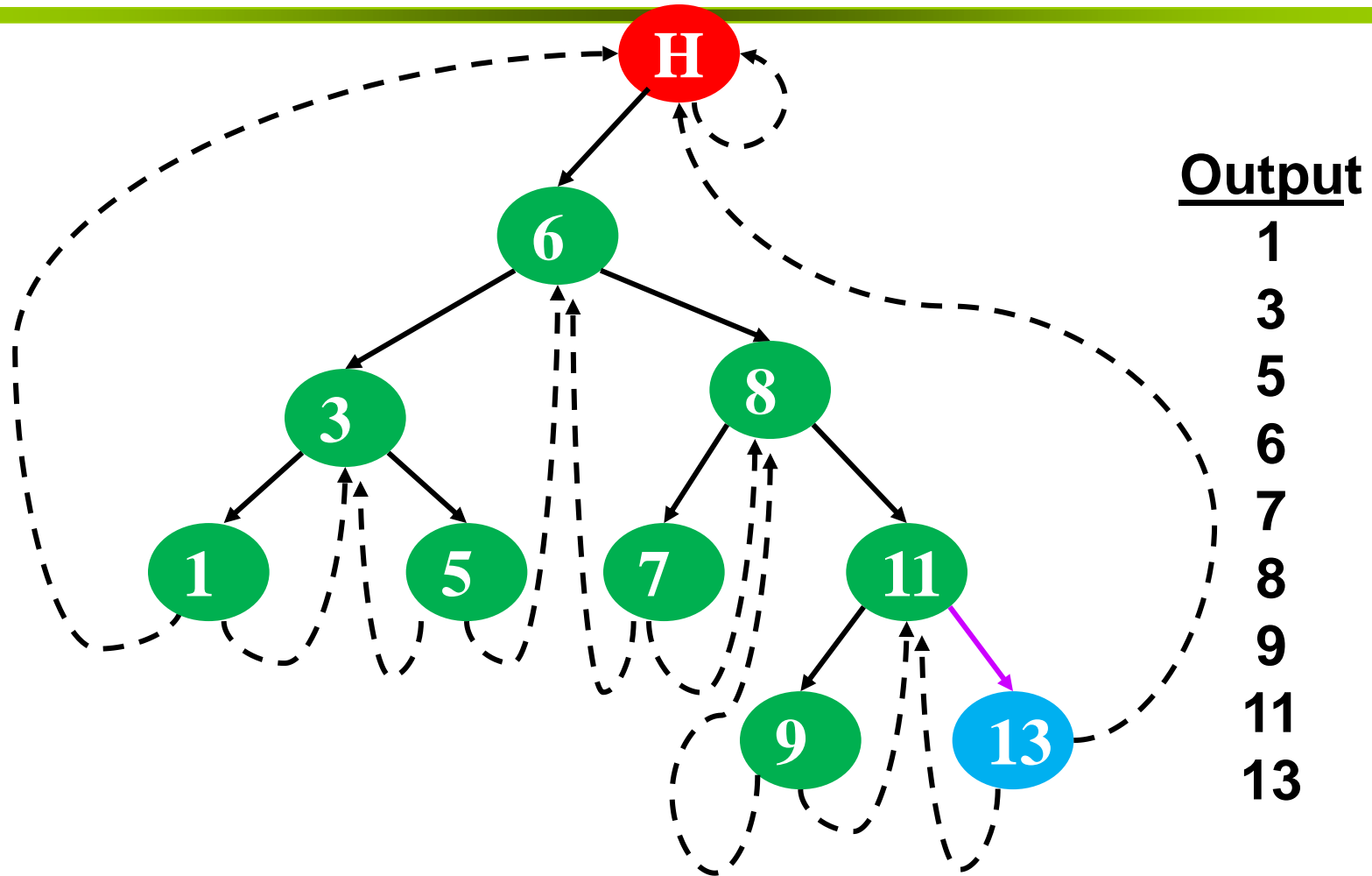
Follow link to right, go to
leftmost node and print

Threaded Tree Inorder Traversal



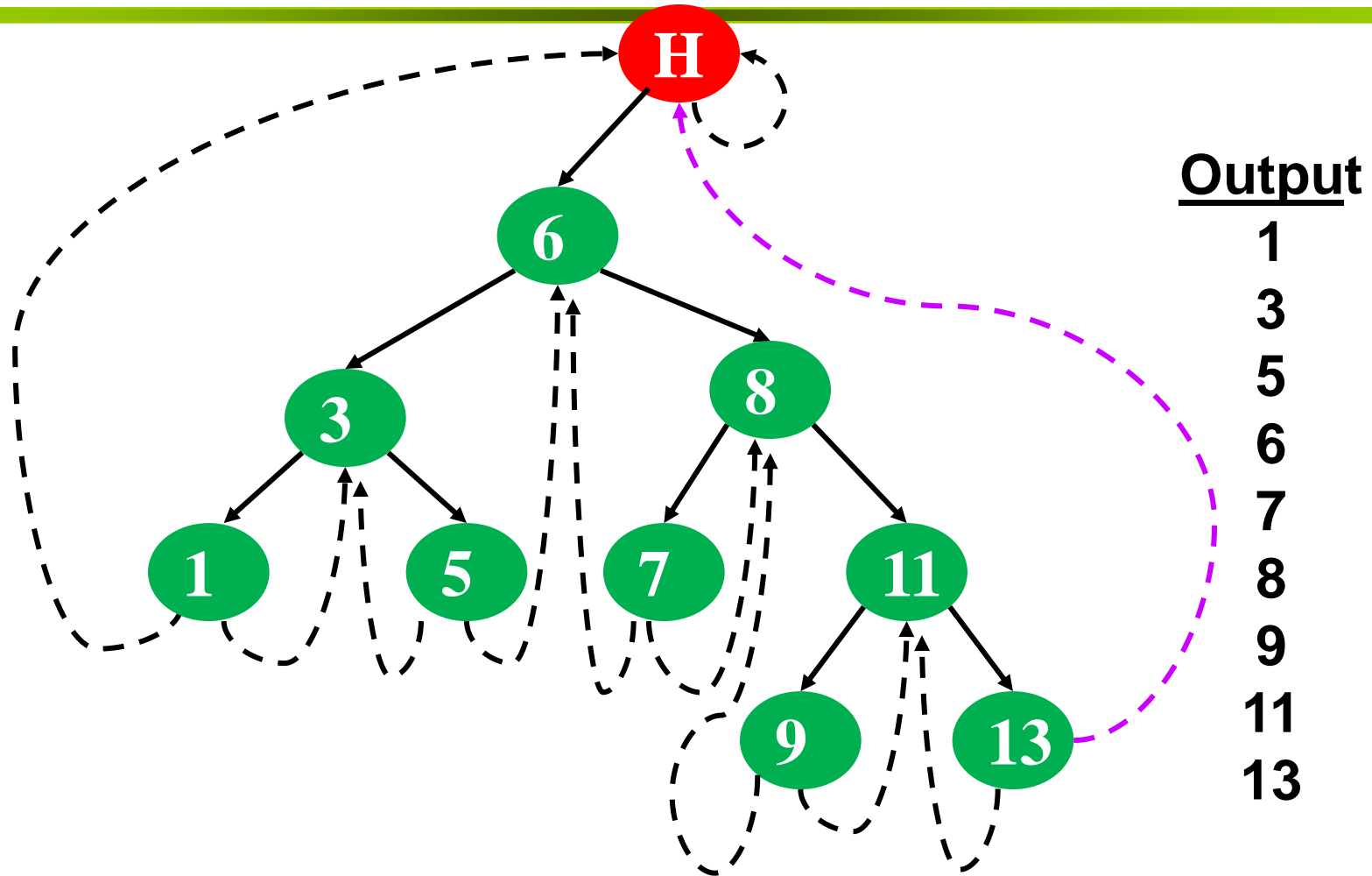
Follow thread to right, print node

Threaded Tree Inorder Traversal



Follow link to right, go to
leftmost node and print

Threaded Tree Inorder Traversal



Follow thread to right, and stop

Inorder Traversal for Inorder threaded tree

```
void inorder(ttree *header)
```

```
{
```

```
    Set temp = header->lc;
```

```
    while(temp != header)
```

```
    {
```

```
        while(temp->lt == 'f')
```

```
        {
```

```
            temp = temp->lc;
```

```
        }
```

```
        print temp->data
```

```
        while(temp->rt == 't' && temp->rc != header)
```

```
        {
```

```
            temp = temp->rc;
```

```
            print temp->data
```

```
        }
```

```
        temp=temp->rc;
```

```
    }
```

```
}
```

Preorder Traversal for Inorder threaded tree

Void preorder(ttree *header)

{

Set temp = header->lc;

while(temp != header)

{

while(temp->lt == 'f')

{

print temp->data

temp = temp->lc;

}

print temp->data

while(temp->rt == 't' && temp->rc != header)

{

temp = temp->rc;

}

temp=temp->rc;

}

}

Postorder Traversal for Inorder threaded tree

```
Void postorder(ttree *header)
```

```
{   Set temp = header->lc;
    while(temp != header)
    {
        while(temp->rt == 'f')
        {
            A[k++] = temp->data;
            temp = temp->rc;
        }
        A[k++] = temp->data;
        while(temp->lt == 't' && temp->lc != header)
        {
            temp = temp->lc;
        }
        temp=temp->lc;
    }
    for(i=k-1;i>=0;i--)
        printf(" %d",A[i]);
}
```

Deletion of a node in inorder threaded BT

- **4 Cases**

- **The node to be deleted is :**

- ☐ **Leaf node (both threads)**
- ☐ **Has only left subtree (right thread)**
- ☐ **Has only right subtree (left thread)**
- ☐ **Has both left and right subtrees (No Thread)**

Case Study

Huffman's Coding

Introduction

- **A key problem with multimedia is the huge quantities of data that result from raw digitized data of audio, image or video source.**
- **The main goal for coding and compression is to alleviate the storage, processing and transmission costs for these data.**
- **There are a variety of compression techniques commonly used in the Internet and other system.**

Compression

○ Definition

- Reduce size of data

(number of bits needed to represent data)

○ Benefits

- Reduce storage needed
- Reduce transmission cost / latency / bandwidth

Why Compress?

- **To reduce the volume of data to be transmitted (text, fax, images)**
- **To reduce the bandwidth required for transmission and to reduce storage requirements (speech, audio, video)**

Compression Examples

○ Tools

- winzip, pkzip, compress, gzip

○ Formats

□ Images

- .jpg, .gif

□ Audio

- .mp3, .wav

□ Video

- mpeg1 (VCD), mpeg2 (DVD), mpeg4 (Divx)

□ General

- .zip, .gz

Sources of Compressibility

- **Redundancy**

- **Recognize repeating patterns**
 - **Exploit using**
 - **Dictionary**
 - **Variable length encoding**

- **Human perception**

- **Less sensitive to some information**
 - **Can discard less important data**

Types of Compression

○ Lossless

- ❑ Preserves all information
- ❑ Exploits **redundancy** in data
- ❑ Applied to general data

○ Lossy

- ❑ May lose some information
- ❑ Exploits **redundancy** & **human perception**
- ❑ Applied to audio, image, video

Lossless Data Compression

- **Lossless means the reconstructed image doesn't lose any information according to the original one.**
- **There is a huge range of lossless data compression techniques.**
- **The common techniques used are:**
 - **runlength encoding**
 - **Huffman coding**
 - **dictionary techniques**

Lossless Data Compression

○ Runlength compression

- ❑ Removing repetitions of values and replacing them with a counter and single value.
- ❑ Fairly simple to implement.
- ❑ Its performance depends heavily on the input data statistics. The more successive value it has, the more space we can compress.

Run-length Coding

- Repeated occurrence of the same character is called a run
- Number of repetition is called the length of the run
- Run of any length is represented by three characters
 - eeeeeeeetnnnnnnnnnn
 - @e7t@n8

Lossless Data Compression

○ Huffman compression

- Use more less bits to represent the most frequently occurring characters/codeword values, and more bits for the less commonly occurring once.**
- It is the most widespread way of replacing a set of fixed size code words with an optimal set of different sized code words, based on the statistics of the input data.**
- Sender and receiver must share the same codebook which lists the codes and their compressed representation.**

Huffman Coding

- **Assigns fewer bits to symbols that appear more often and more bits to the symbols that appear less often**
- **Efficient when occurrence probabilities vary widely**
- **Huffman codebook from the set of symbols and their occurring probabilities**
- **Two properties:**
 - **generate compact codes**
 - **prefix property**

Lossless Data Compression

○ Dictionary compression

- ❑ Look at the data as it arrives and form a dictionary. when new input comes, it look up the dictionary. If the new input existed, the dictionary position can be transmitted; if not found, it is added to the dictionary in a new position, and the new position and string is sent out.
- ❑ Meanwhile, the dictionary is constructed at the receiver dynamically, so that there is no need to carry out statistics or share a table separately.

Lempel-Ziv-Welch (LZW) Coding

- Works by building a dictionary of phrases from the input stream**
- A token or an index is used to identify each distinct phrase**
- Number of entries in the dictionary determines the number of bits required for the index -- a dictionary with 25,000 words requires 15 bits to encode the index**

Effectiveness of Compression

- **Metrics**

- **Bits per byte (8 bits)**

- 2 bits / byte \Rightarrow $\frac{1}{4}$ original size
 - 8 bits / byte \Rightarrow no compression

- **Percentage**

- 75% compression \Rightarrow $\frac{1}{4}$ original size

Effectiveness of Compression

- **Depends on data**

- **Random data \Rightarrow hard**

- **Example: 1001110100 \Rightarrow ?**

- **Organized data \Rightarrow easy**

- **Example: 1111111111 $\Rightarrow 1 \times 10$**

- **Corollary**

- **No universally best compression algorithm**

Effectiveness of Compression

- **Lossless Compression is not guaranteed**
 - **Pigeonhole principle**
 - Reduce size 1 bit \Rightarrow can only store $\frac{1}{2}$ of data
 - Example
 - 000, 001, 010, 011, 100, 101, 110, 111 \Rightarrow 00, 01, 10, 11
 - **If compression is always possible (alternative view)**
 - Compress file (reduce size by 1 bit)
 - Recompress output
 - Repeat (until we can store data with 0 bits)

Lossless Compression Techniques

- **LZW (Lempel-Ziv-Welch) compression**
 - **Build pattern dictionary**
 - **Replace patterns with index into dictionary**
- **Run length encoding**
 - **Find & compress repetitive sequences**
- **Huffman code**
 - **Use variable length codes based on frequency**

Huffman Code

○ Approach

- ❑ Variable length encoding of symbols
- ❑ Exploit statistical frequency of symbols
- ❑ Efficient when symbol probabilities vary widely

○ Principle

- ❑ Use fewer bits to represent **frequent** symbols
- ❑ Use more bits to represent **infrequent** symbols



Huffman Code Example

Symbol	Dog	Cat	Bird	Fish
Frequency	1/8	1/4	1/2	1/8
Original Encoding	00	01	10	11
	2 bits	2 bits	2 bits	2 bits
Huffman Encoding	110	10	0	111
	3 bits	2 bits	1 bit	3 bits

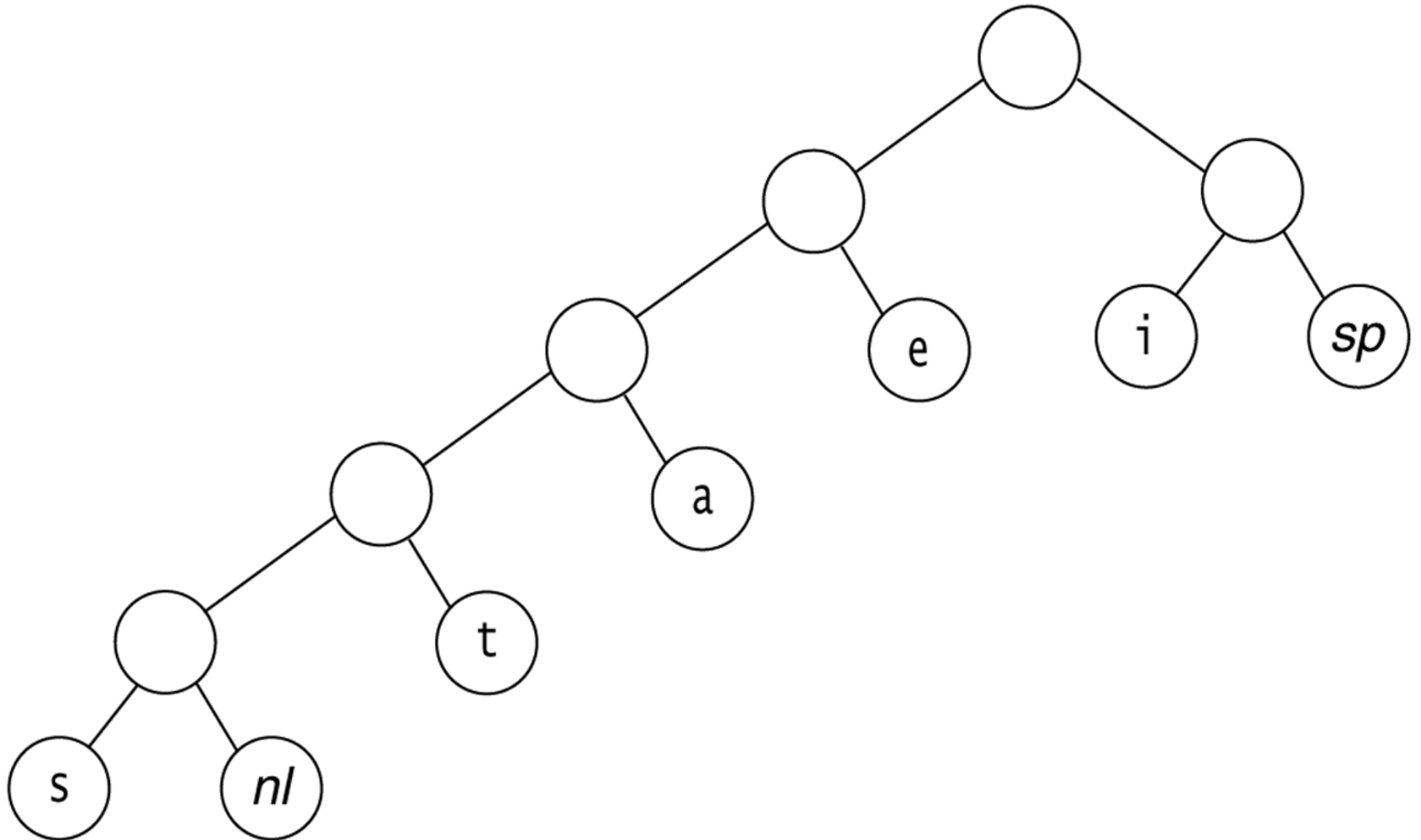
○ Expected size

- ❑ Original $\Rightarrow 1/8 \times 2 + 1/4 \times 2 + 1/2 \times 2 + 1/8 \times 2 = 2$ bits / symbol
- ❑ Huffman $\Rightarrow 1/8 \times 3 + 1/4 \times 2 + 1/2 \times 1 + 1/8 \times 3 = 1.75$ bits / symbol

Standard Coding Scheme

Character	Code	Frequency	Total Bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
<i>sp</i>	101	13	39
<i>nl</i>	110	1	3
Total			174

Optimal Prefix Code Tree



Optimal Prefix Code Cost

Character	Code	Frenquency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
<i>sp</i>	11	13	26
<i>nl</i>	00001	1	5
Total			146

Huffman Code Data Structures

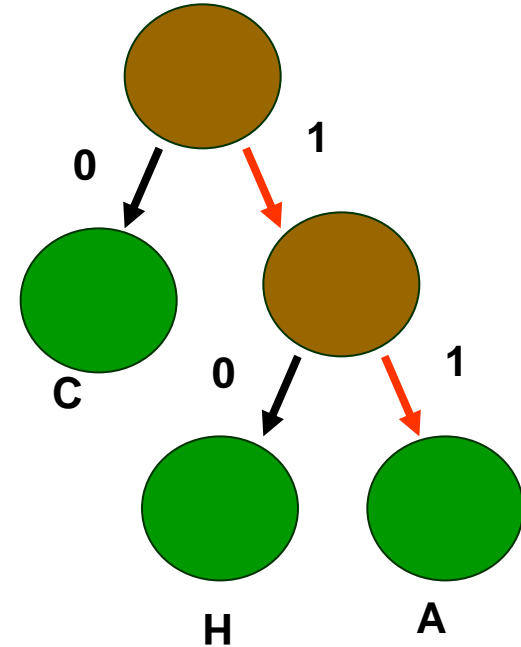
- **Binary (Huffman) tree**

- Represents Huffman code
- Edge \Rightarrow code (0 or 1)
- Leaf \Rightarrow symbol
- Path to leaf \Rightarrow encoding
- Example

○ A = "11", H = "10", C = "0"

- **Priority queue**

- To efficiently build binary tree



Huffman Code Algorithm Overview

○ Encoding

- Calculate frequency of symbols in file
- Create binary tree representing “best” encoding
- Use binary tree to encode compressed file
 - For each symbol, output path from root to leaf
 - Size of encoding = length of path
- Save binary tree

Huffman Code – Creating Tree

○ Algorithm

□ Place each symbol in leaf

- Weight of leaf = symbol frequency

□ Select two trees L and R (initially leafs)

- Such that L, R have lowest frequencies in tree

□ Create new (internal) node

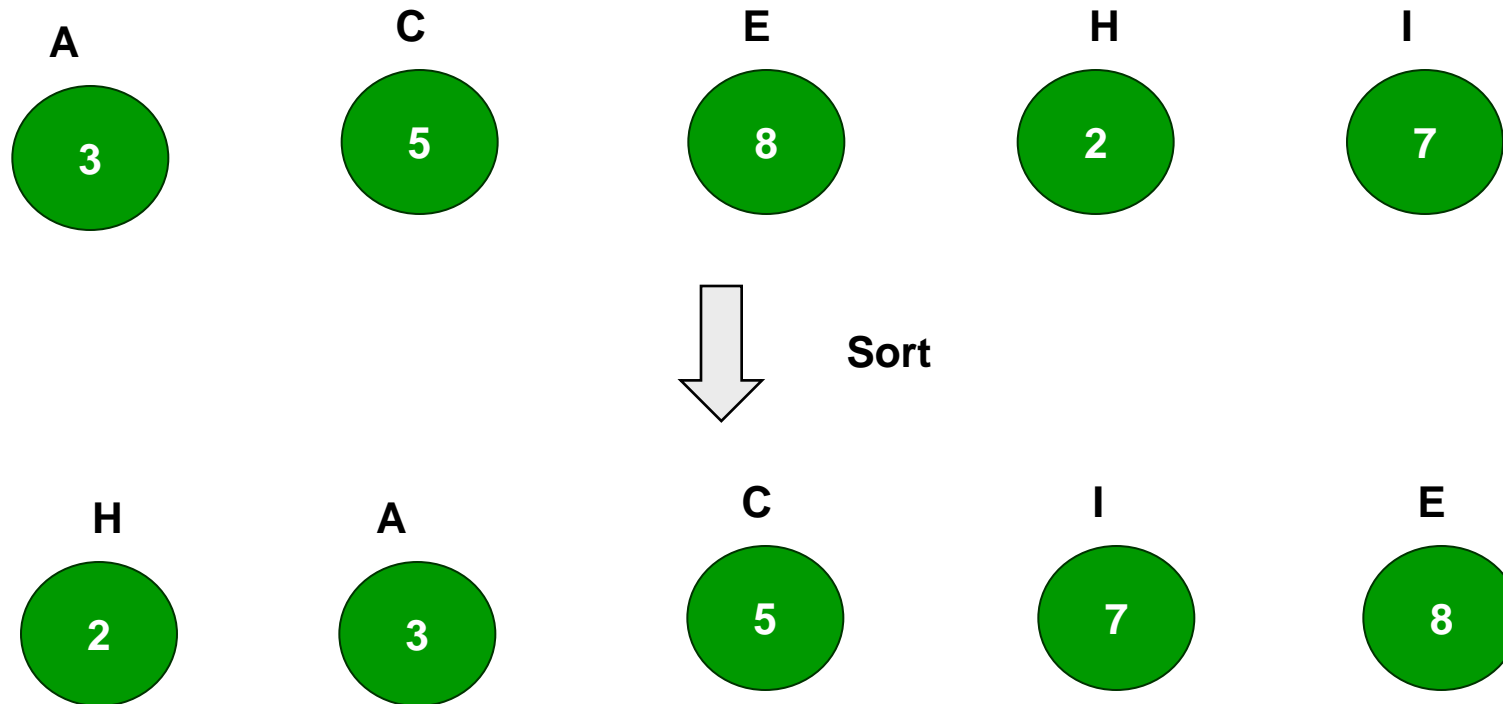
- Left child \Rightarrow L

- Right child \Rightarrow R

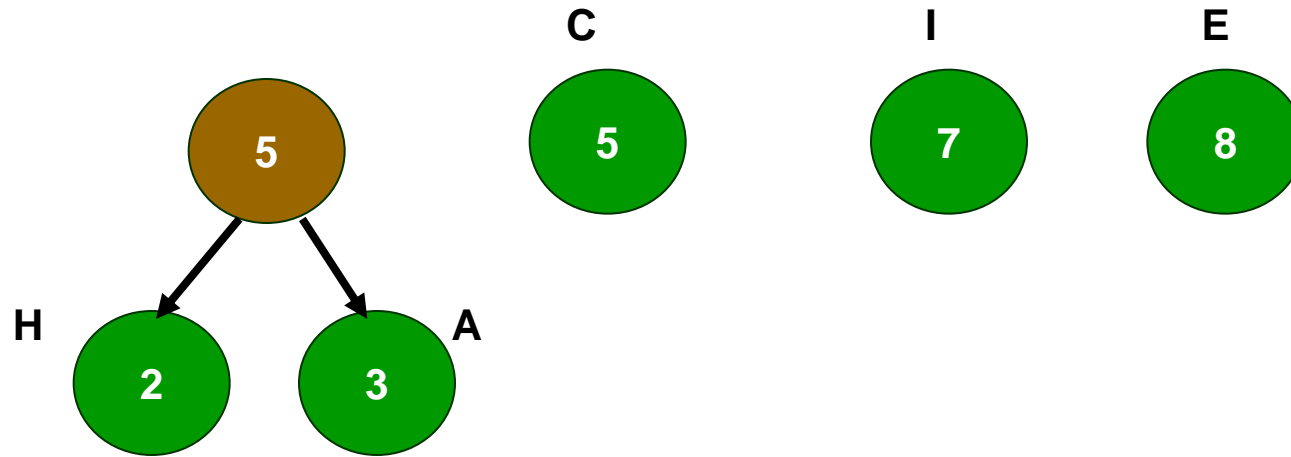
- New frequency \Rightarrow frequency(L) + frequency(R)

□ Repeat until all nodes merged into one tree

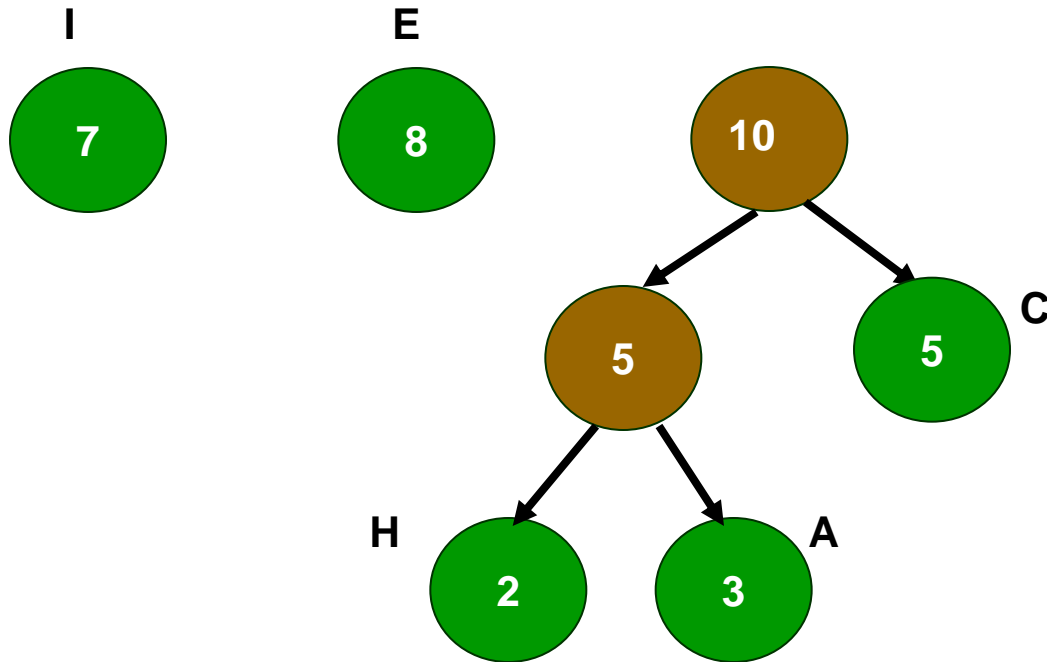
Huffman Tree Construction 1



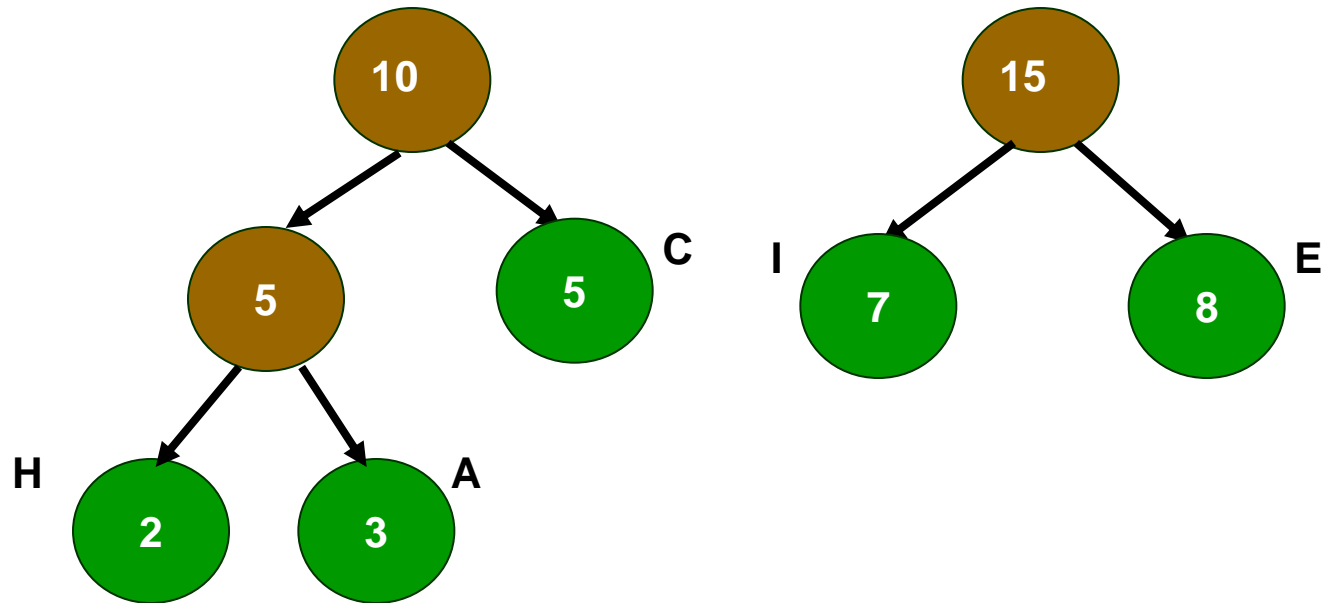
Huffman Tree Construction 2



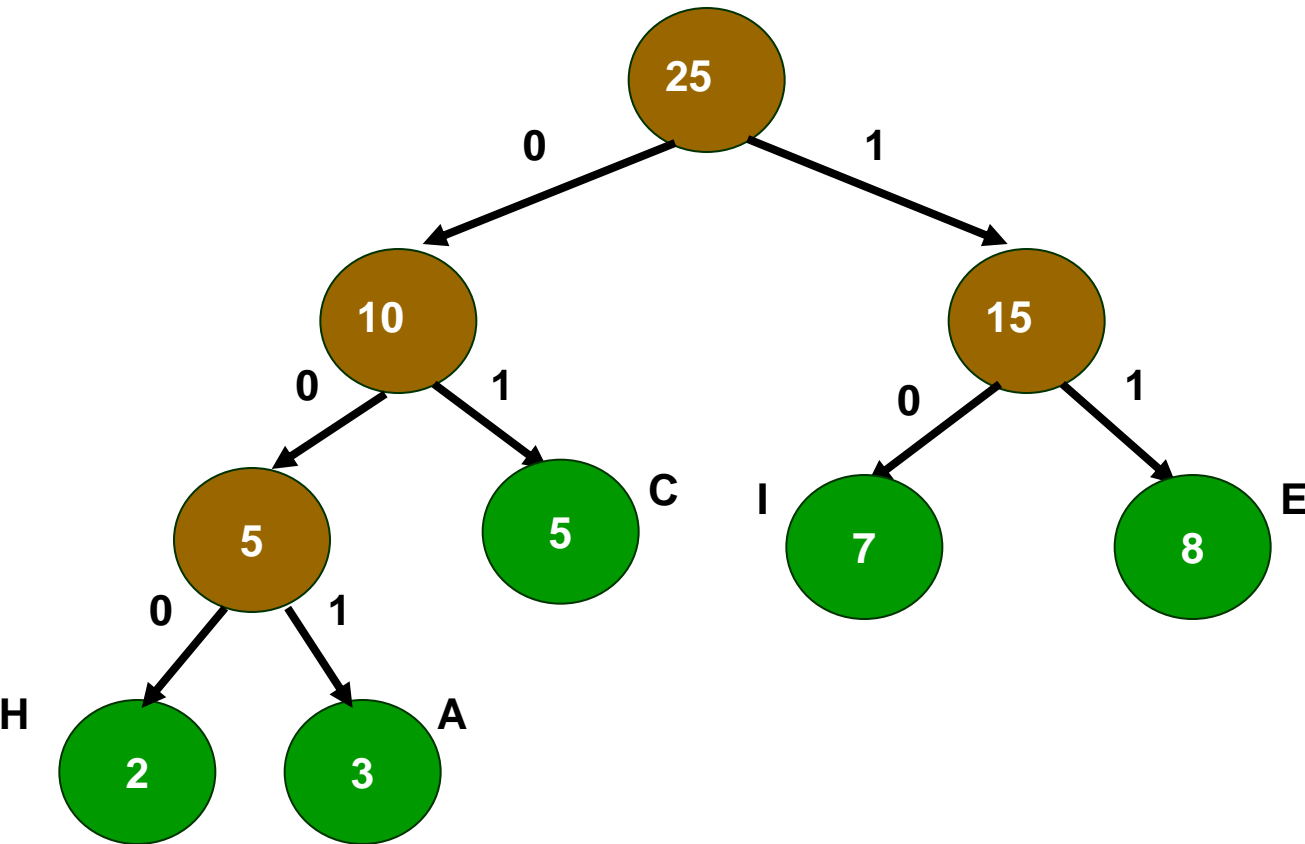
Huffman Tree Construction 3



Huffman Tree Construction 4



Huffman Tree Construction 5



E	=	11
I	=	10
C	=	01
A	=	001
H	=	000

Huffman Coding Example

- Huffman code

E	=	11
I	=	10
C	=	01
A	=	001
H	=	000

- Input

 - ACE

- Output

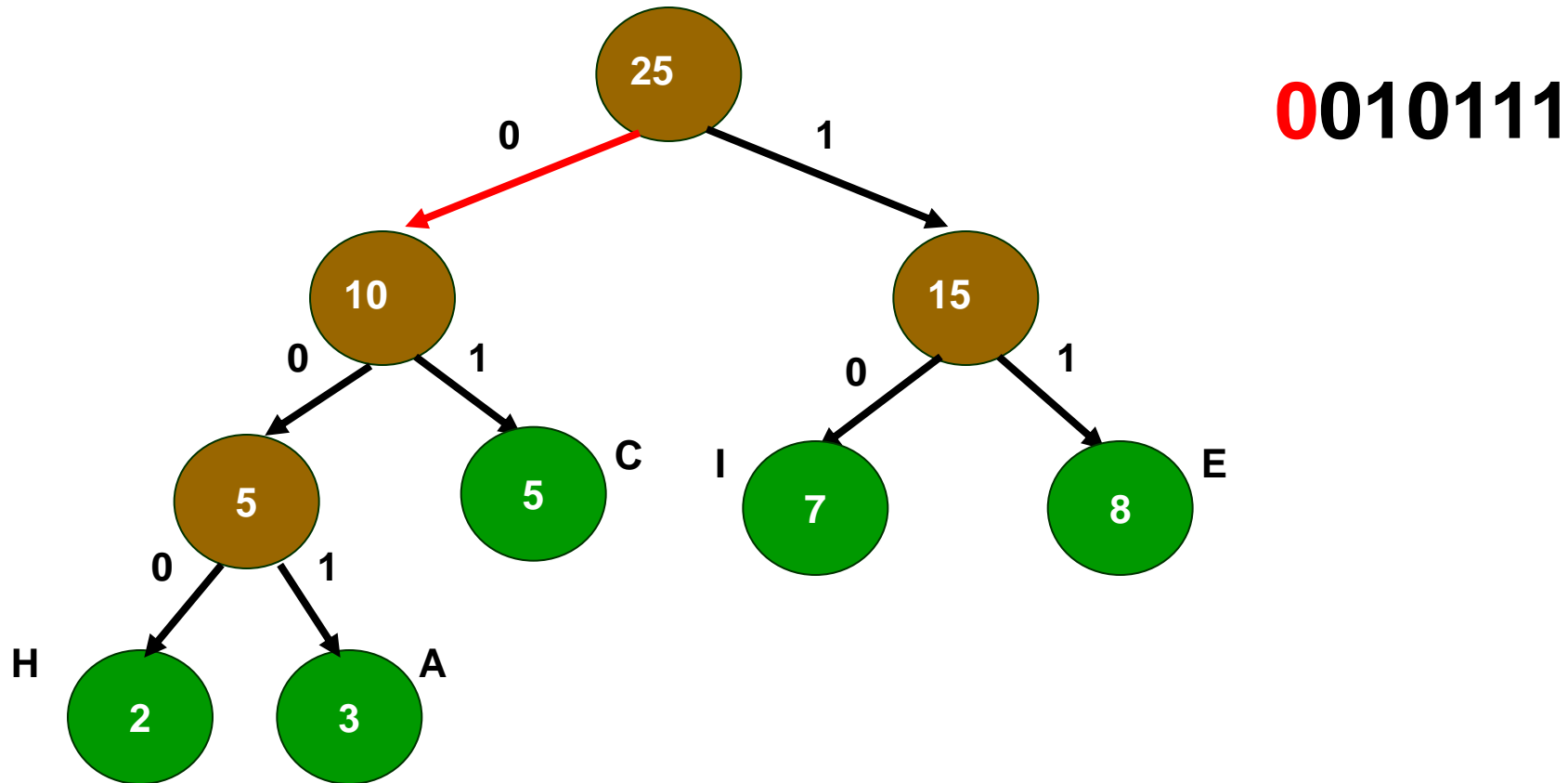
 - (001)(01)(11) = 0010111

Huffman Code Algorithm Overview

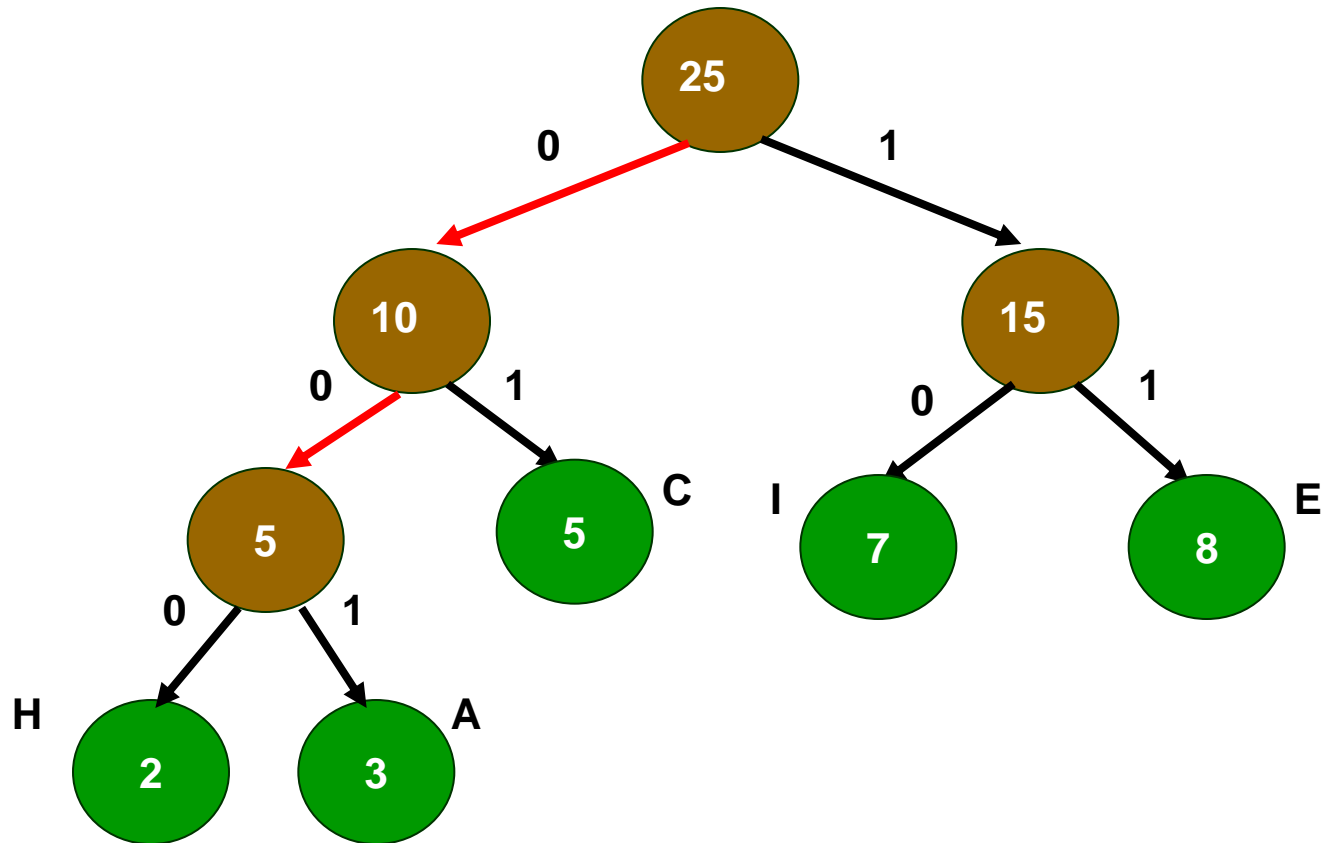
- **Decoding**

- **Read compressed file & binary tree**
 - **Use binary tree to decode file**
 - **Follow path from root to leaf**

Huffman Decoding 1

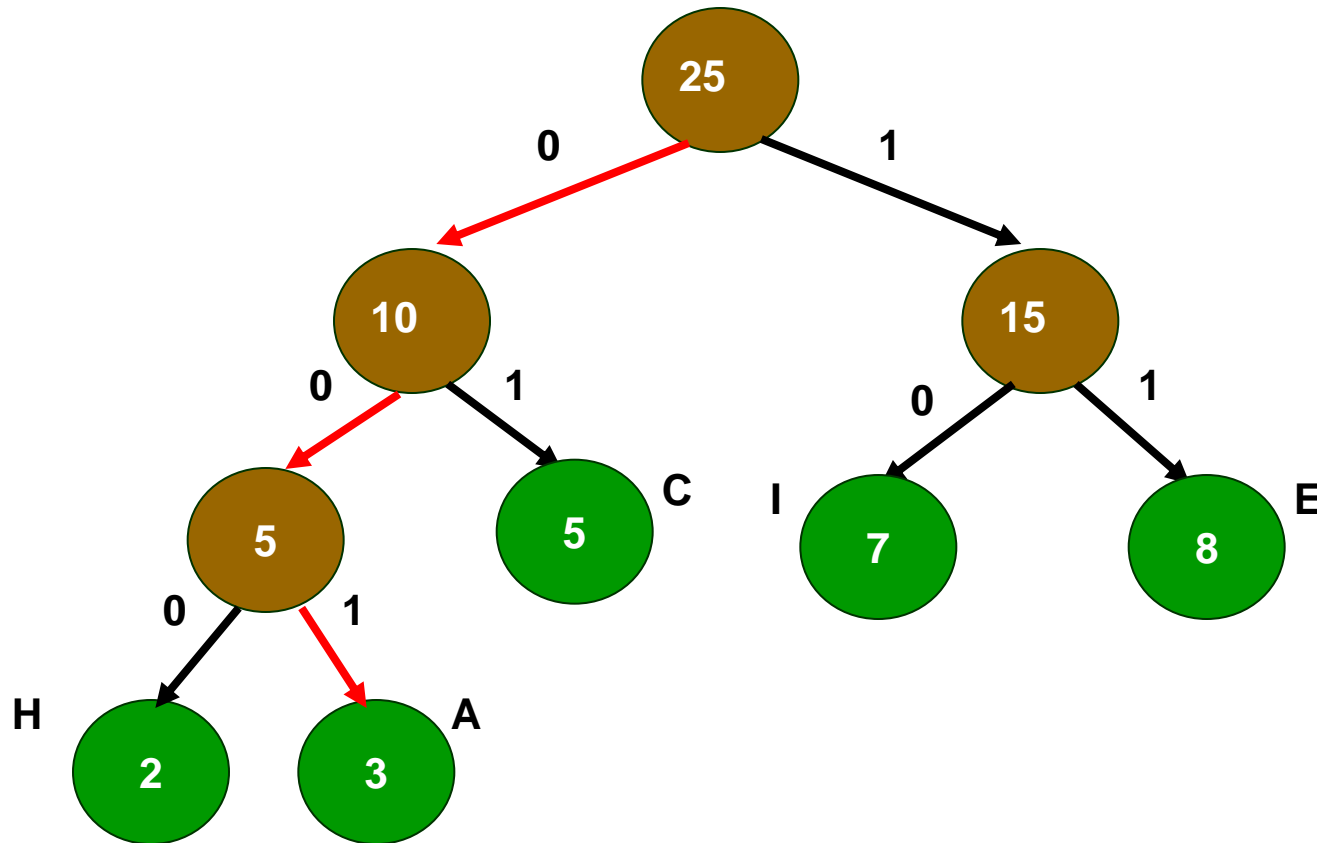


Huffman Decoding 2



0010111

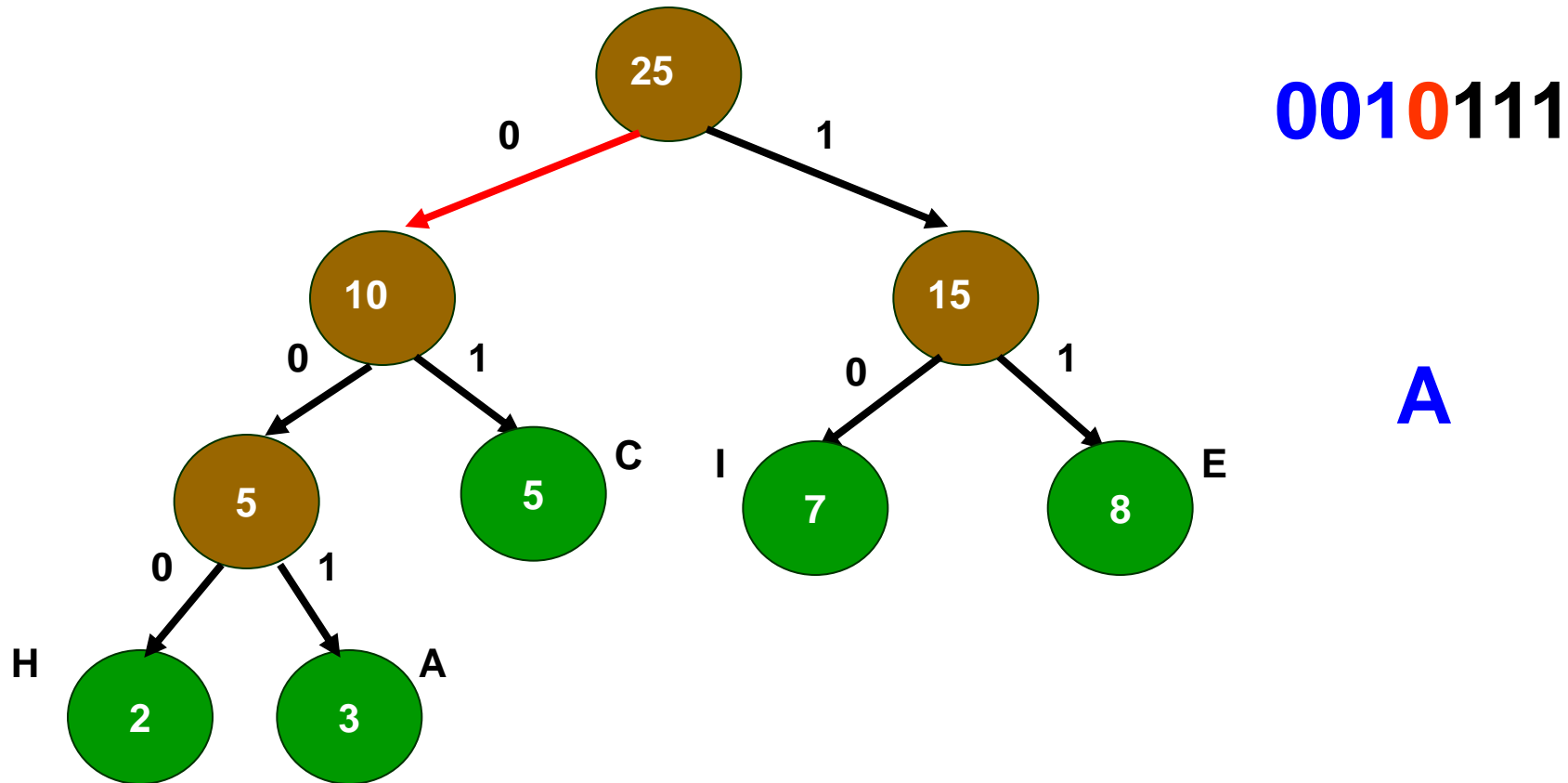
Huffman Decoding 3



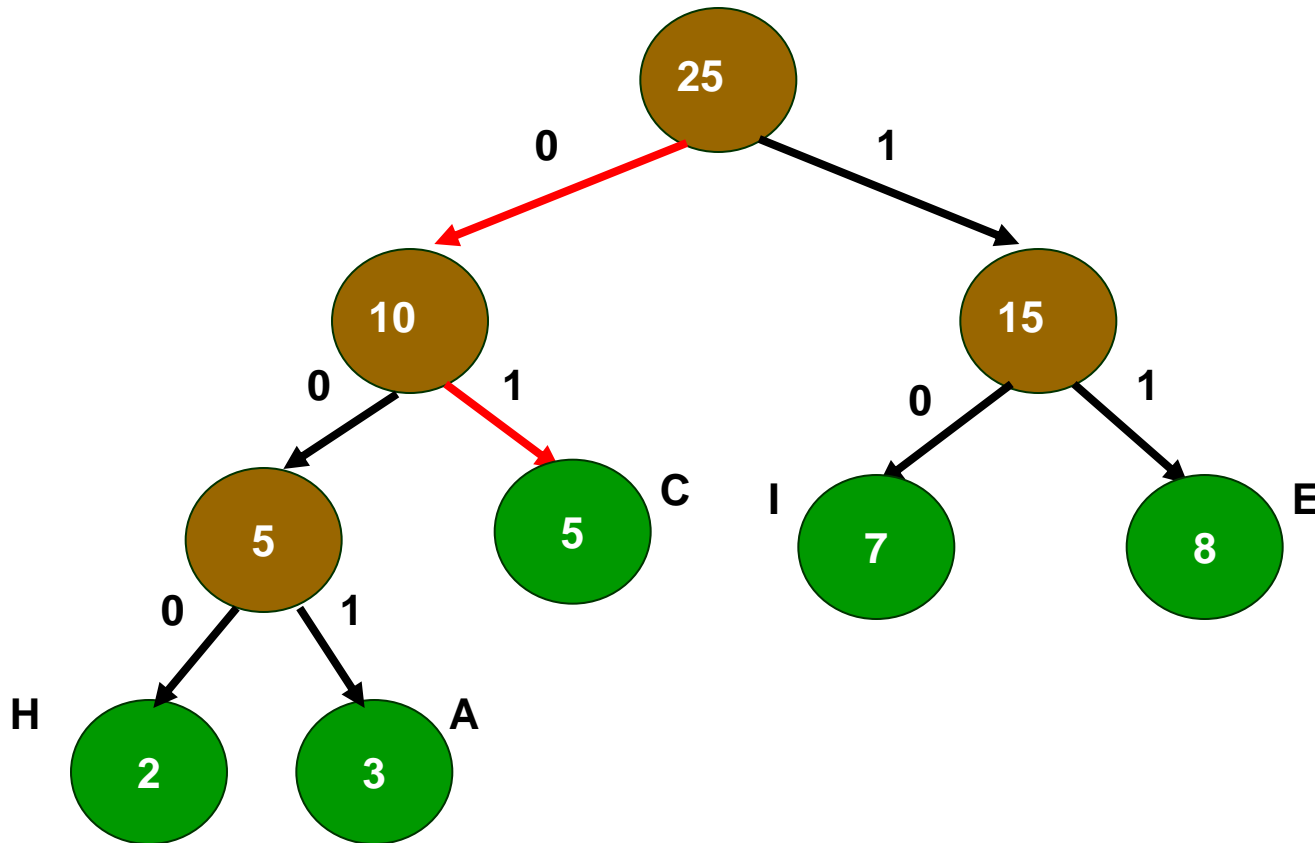
0010111

A

Huffman Decoding 4



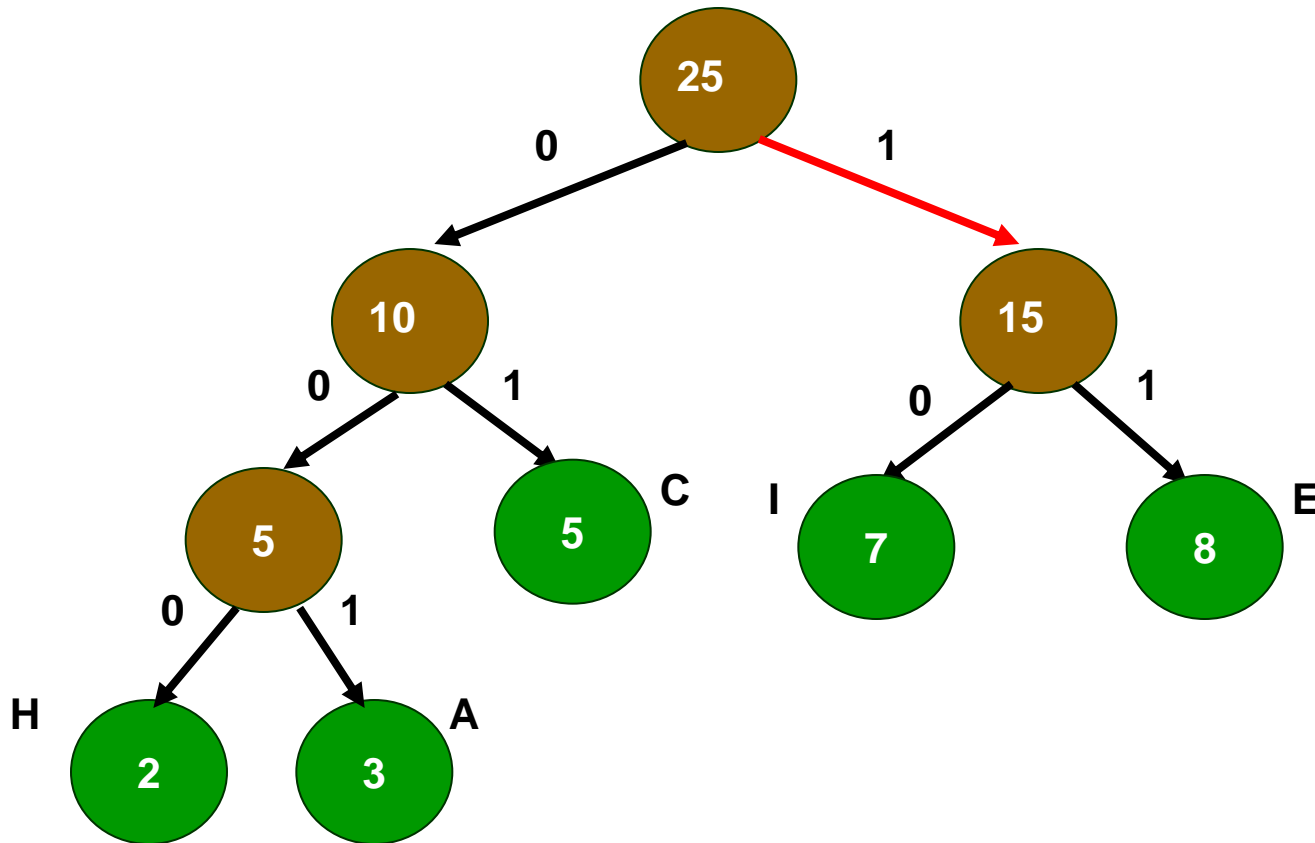
Huffman Decoding 5



0010111

AC

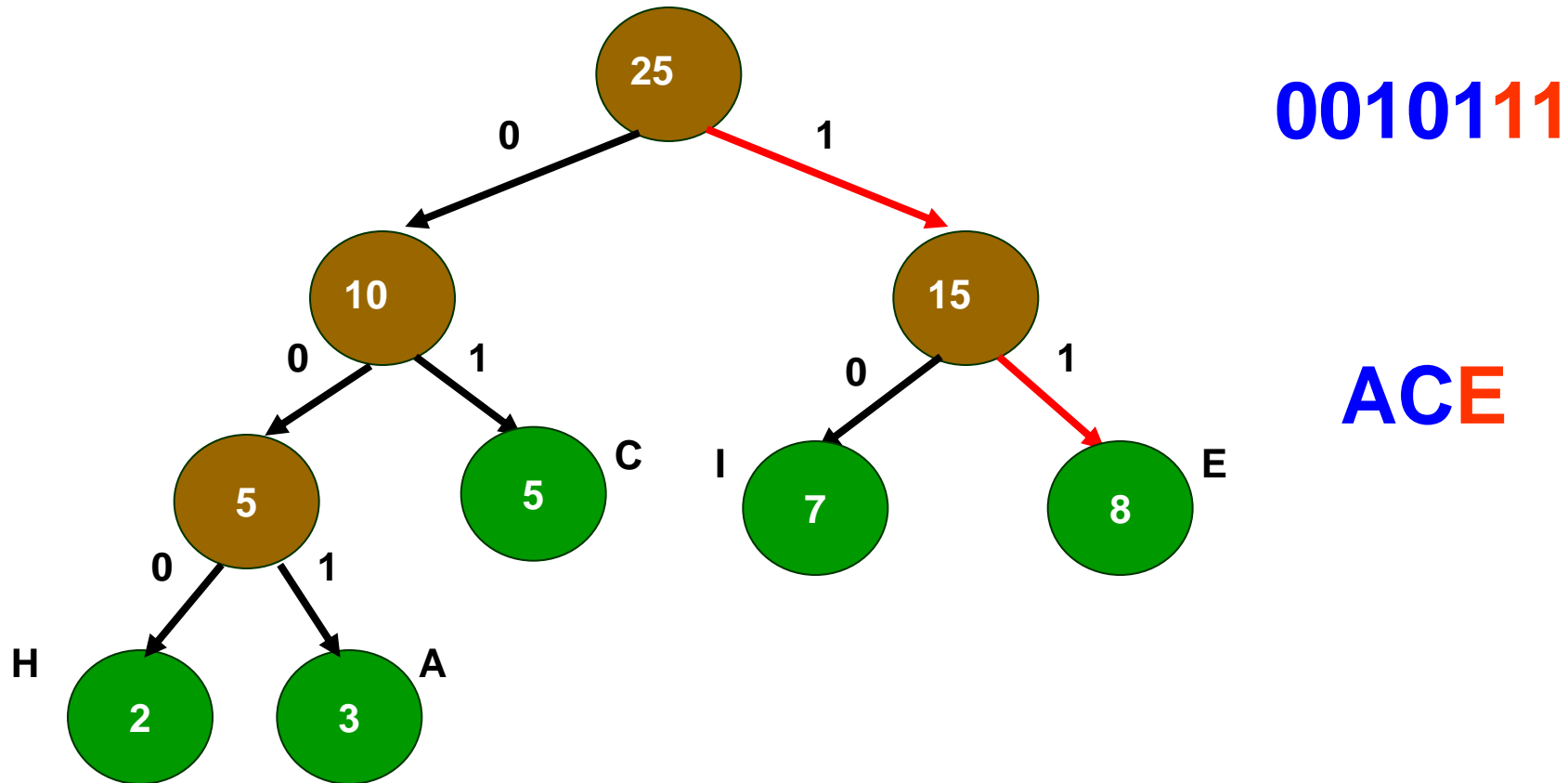
Huffman Decoding 6



0010111

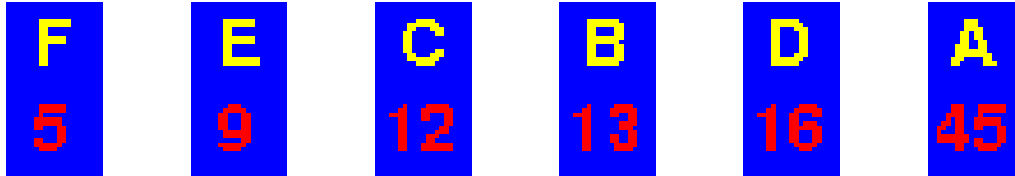
AC

Huffman Decoding 7

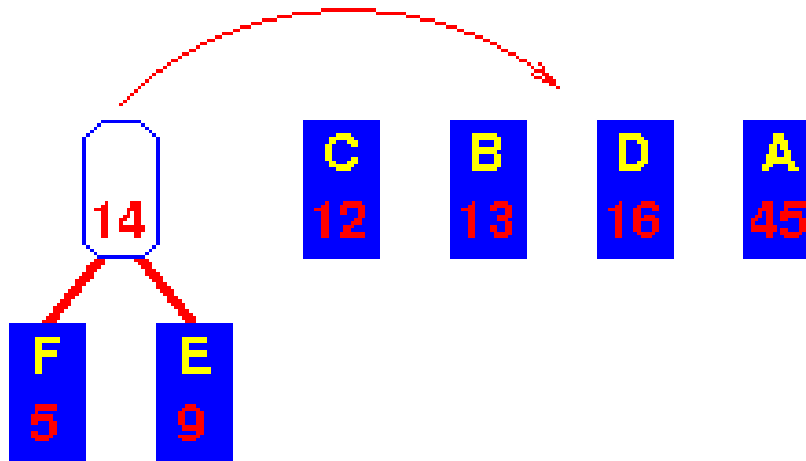


Huffman Encoding - Operation

Initial sequence
Sorted by frequency



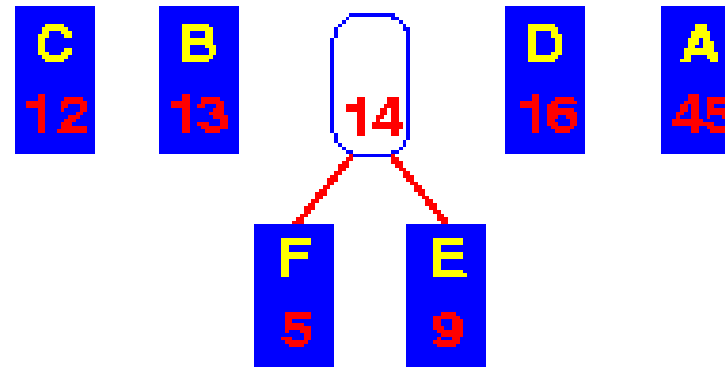
Combine lowest two
into sub-tree



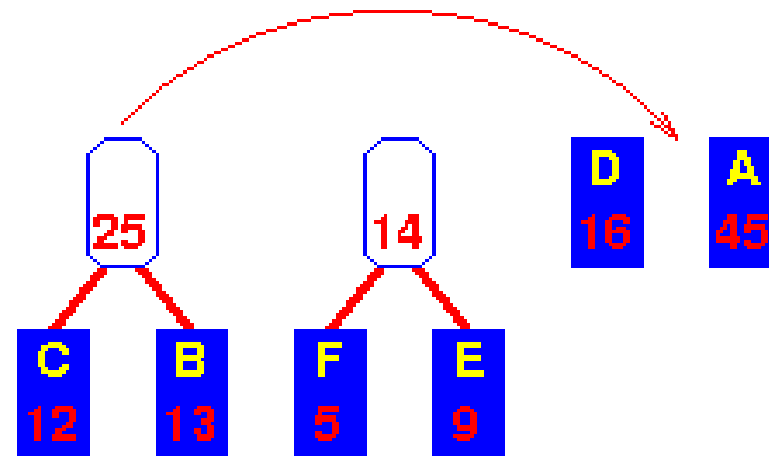
Move it to correct
place

Huffman Encoding - Operation

After shifting sub-tree
to its correct place ...



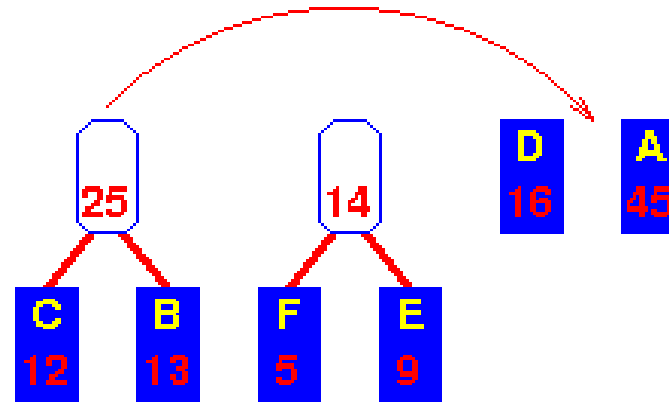
Combine next lowest
pair



Move sub-tree to
correct place

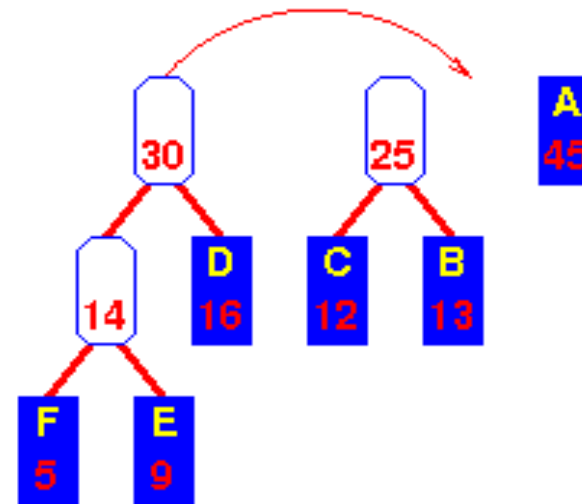
Huffman Encoding - Operation

Move the new tree
to the correct place ...

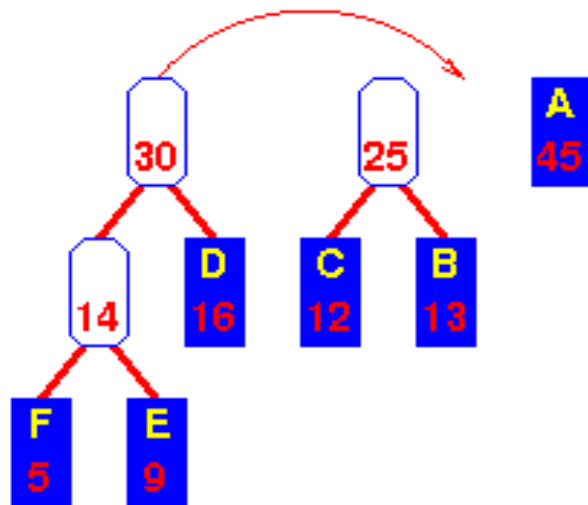


Now the lowest two are the
“14” sub-tree and D

Combine and move to
correct place



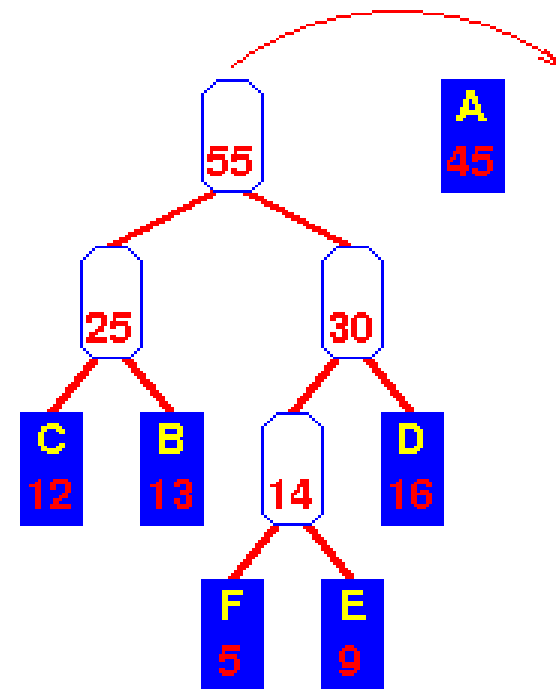
Huffman Encoding - Operation



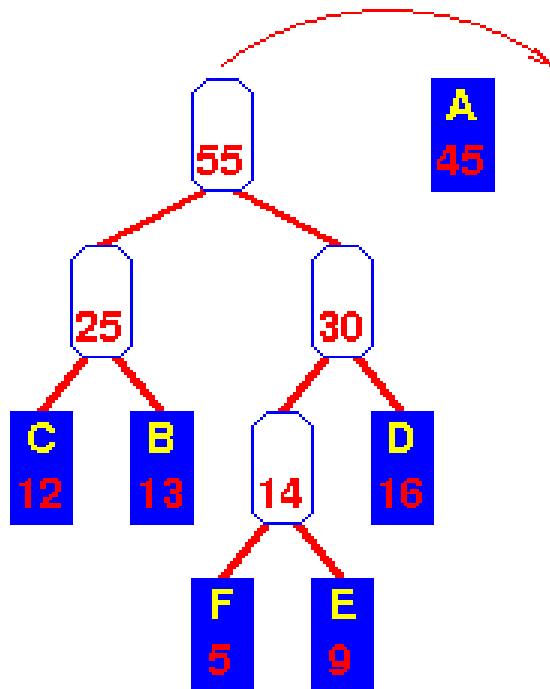
Move the new tree
to the correct place ...

Now the lowest two are the
the “25” and “30” trees

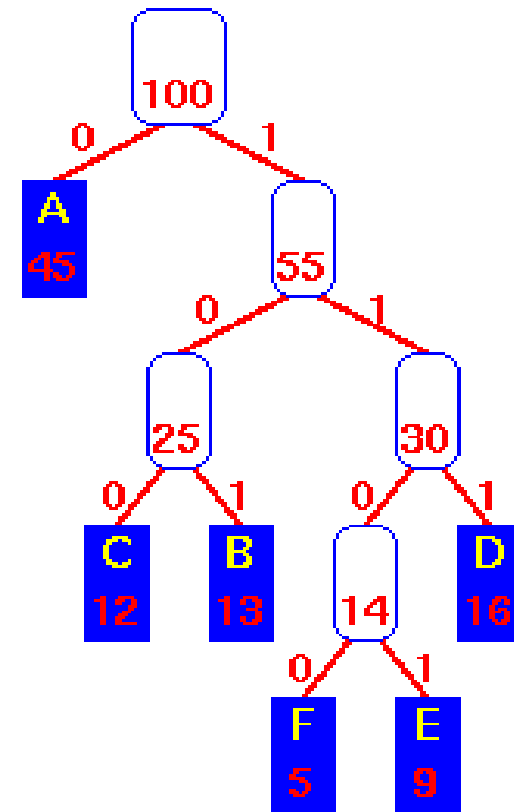
Combine and move to
correct place



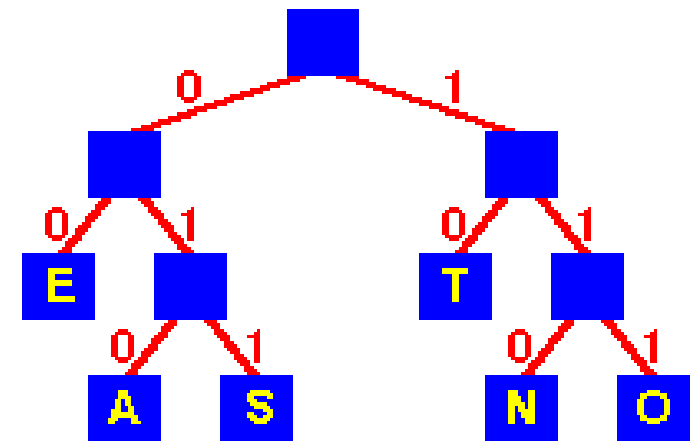
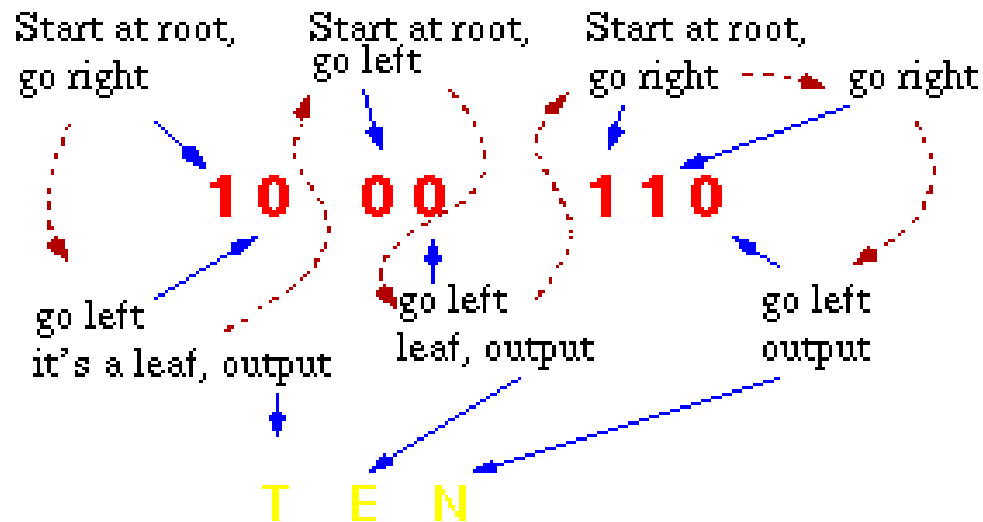
Huffman Encoding - Operation



Combine
last two trees



Huffman Encoding - Decoding



Huffman Encoding - Time Complexity

- **Sort keys** $O(n \log n)$
- **Repeat n times**
 - **Form new sub-tree** $O(1)$
 - **Move sub-tree** $O(\log n)$
(binary search)
 - **Total** $O(n \log n)$
- **Overall** $O(n \log n)$

Huffman Code Properties

○ Prefix code

- No code is a **prefix** of another code

- Example

 - Huffman("dog") \Rightarrow 01

 - Huffman("cat") \Rightarrow **01**1 // not legal prefix code

- Can stop as soon as complete code found

- No need for end-of-code marker

○ Nondeterministic

- Multiple Huffman coding possible for same input

- If more than two trees with same minimal weight

Huffman Code Properties

- **Greedy algorithm**
 - Chooses best local solution at each step
 - Combines 2 trees with lowest frequency
- **Still yields overall best solution**
 - Optimal prefix code
 - Based on statistical frequency
- **Better compression possible (depends on data)**
 - Using other approaches (e.g., pattern dictionary)