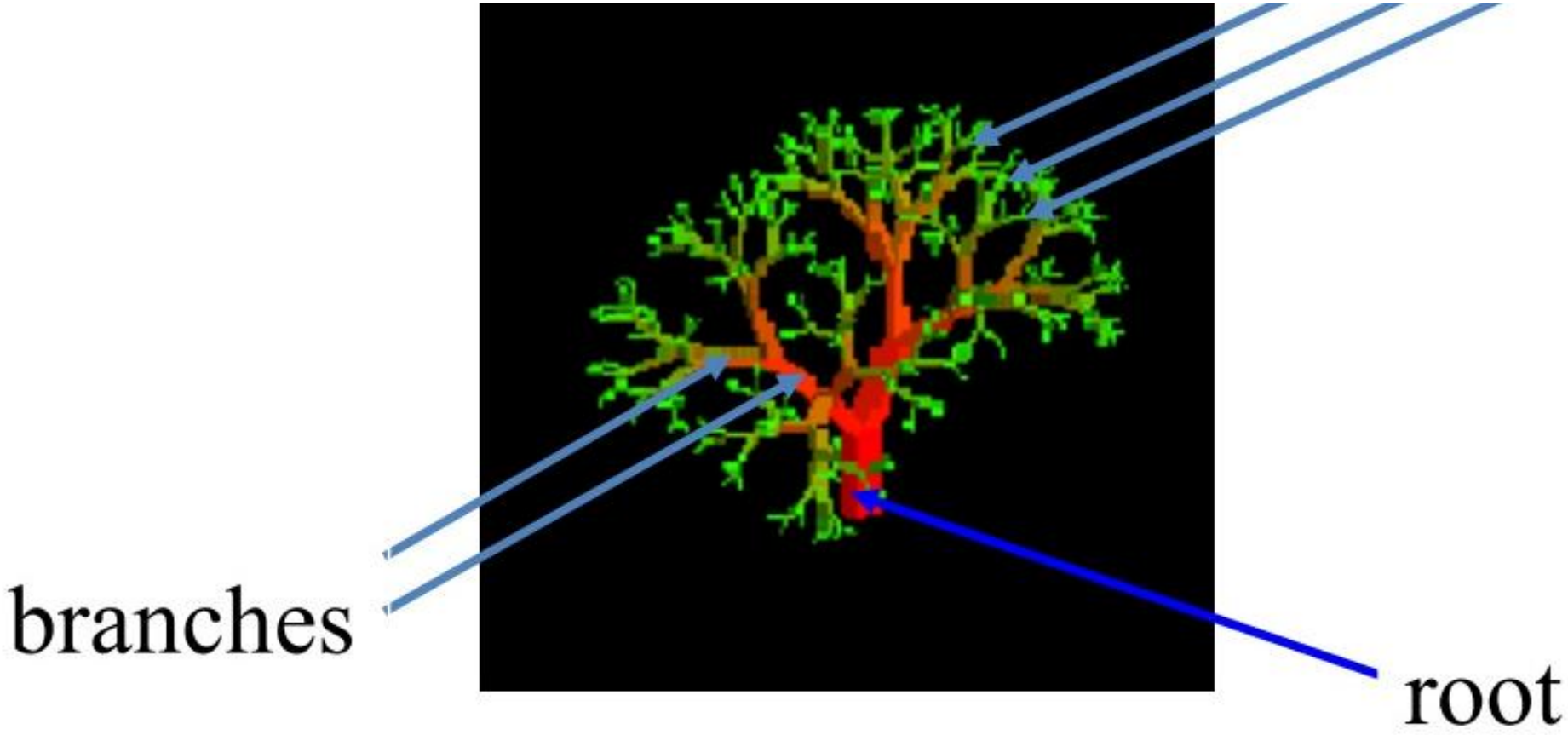
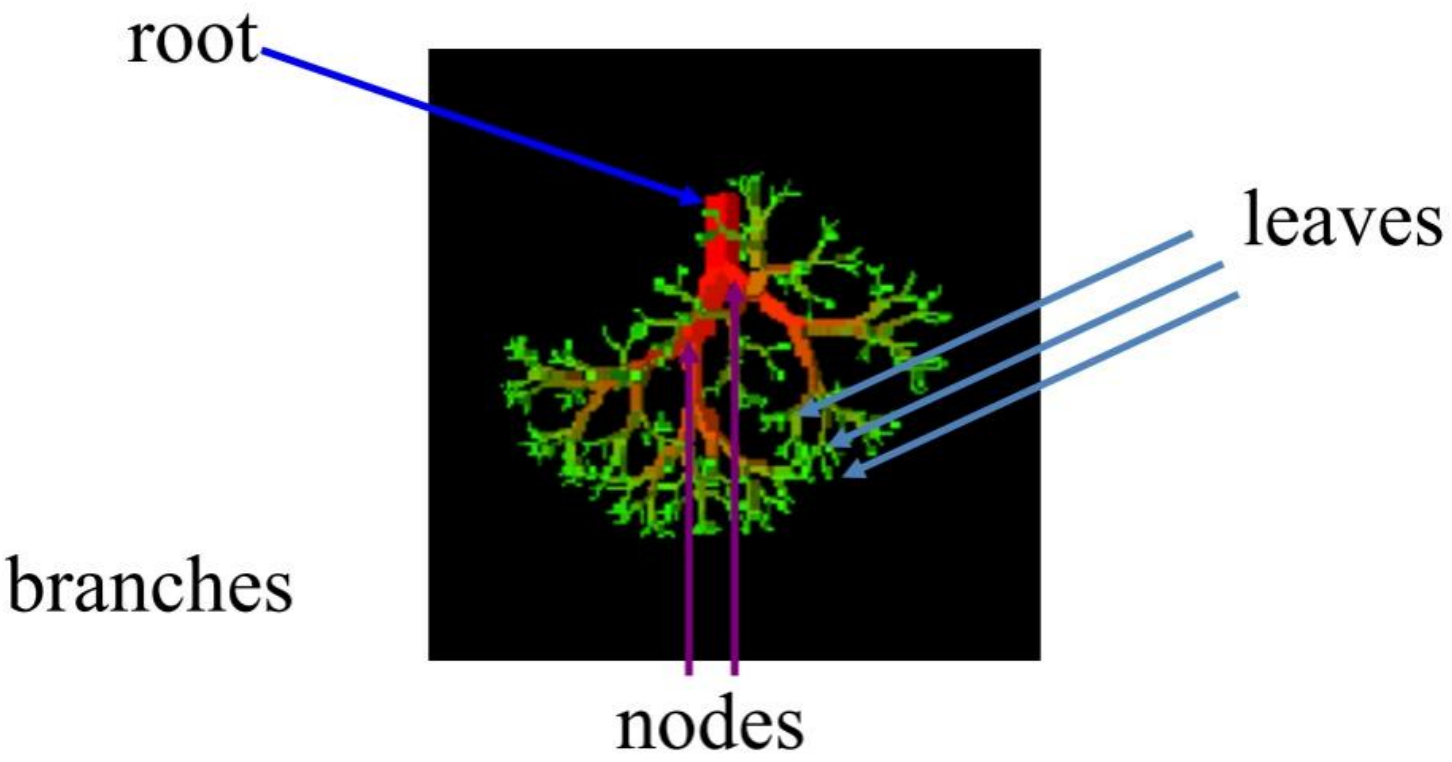
Module 4 TREES

General View



Data Structures View



TREES

- A **tree** is a **finite set of one or more nodes** such that:
- There is a **specially designated node** called the **root**.
- The remaining nodes (if any) are partitioned into **n >= 0 disjoint sets** $T_1, T_2, \ldots, Tn$,
  - each of these sets is itself a **tree**.
  - These trees $T_1, T_2, \ldots, Tn$ are called the **subtrees of the root**.

- Applications:
- –Organization charts
- –File systems
- –Programming environments

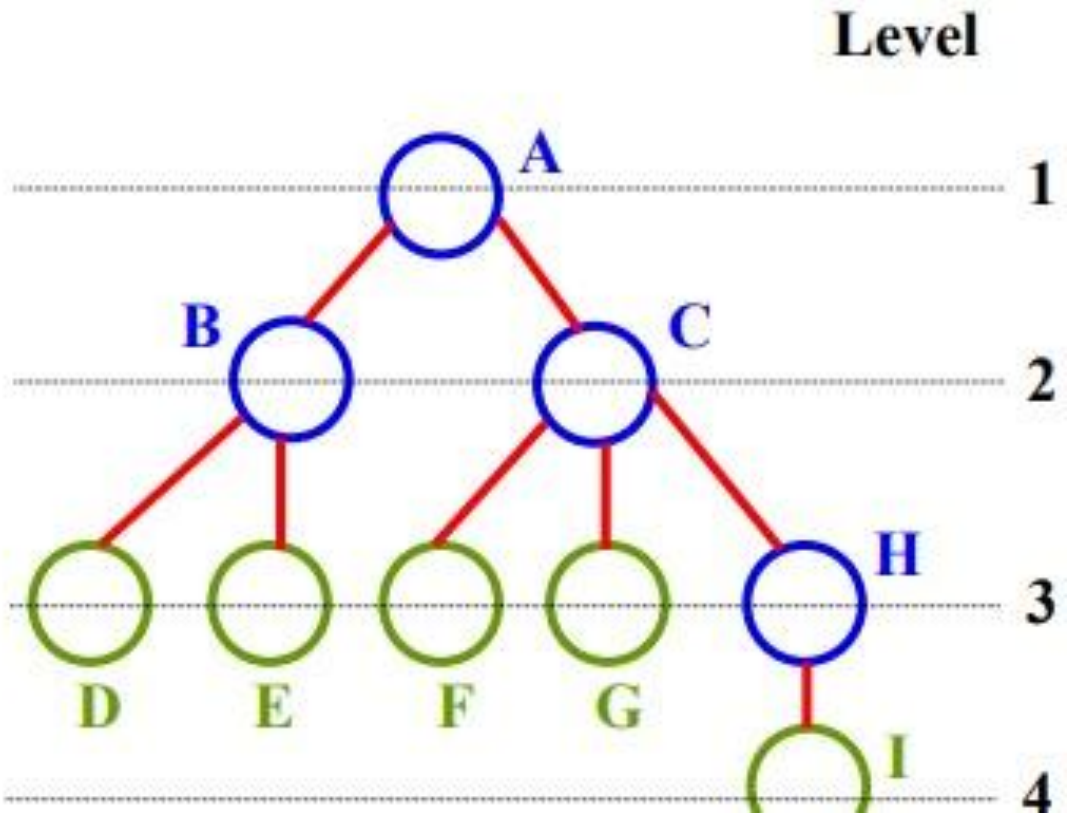
Linear DS Vs Non-Linear DS

Feature	Linear DS	Non-Linear DS
Arrangement	Sequential (one after another)	Hierarchical (tree) or Network (graph)
Relationship	One-to-one	One-to-many / many-to-many
Traversal	Simple (for loop, while loop)	Complex (DFS, BFS)
Memory	Usually continuous (array) or linked sequentially	Random nodes linked via pointers
Examples	Array, Stack, Queue, Linked List	Tree, Graph

Tree Terminology

- **Root:** Node without a parent. The starting point of the tree.
- **Siblings:** Nodes that share the same parent.
- **Internal node:** Node with at least one child.
- **External node (leaf):** Node without children.
- **Ancestors of a node:** All nodes along the path from the root to the parent of the node (parent, grandparent, etc.).
- **Descendants of a node:** All nodes in the subtree rooted at the node, excluding the node itself (children, grandchildren, etc.).
- **Depth of a node:** The number of nodes on the path from the root to the node.
- *Example:* If the root is included, depth(root) = 1.
- **Height of a node:** The number of nodes in the longest path from that node down to a leaf.
- *Example:* If the node is a leaf, its height = 1.
- **Height of the tree:** The height of the root node; i.e., the number of nodes on the longest path from root to a leaf.
- **Degree of a node:** The number of children of the node.
- **Degree of a tree:** The maximum degree among all nodes in the tree.
- **Subtree:** A tree consisting of a node and all its descendants.

EXAMPLE

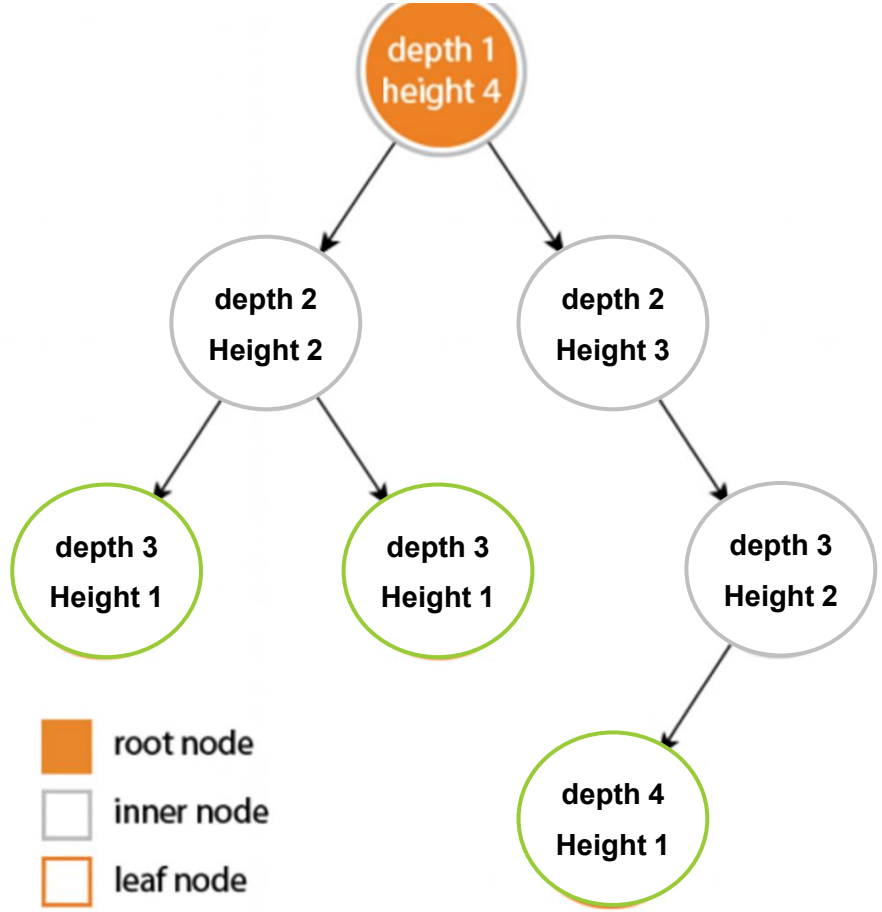


- *A* is the *root* node
- *B* is the *parent* of D and E
- *C* is the *sibling* of B
- *D* and *E* are the *children* of B
- *D, E, F, G, I* are *external nodes*, or *leaves*
- *A, B, C, H* are *internal nodes*
- The *height* of the node A is 4
- The *depth* of the node A is 1
- The *degree* of node *B* is *2*
- The *degree* of the tree is *3*
- The *ancestors* of node *I* is *A, C, H*
- The *descendants* of node *C* is *F, G, H, I*

What will be the height of the node B & C ?

Height and Depth Difference

- Height and depth of a tree is equal...
- but height and depth of a node is not equal because...
- the height is calculated by traversing from the given node to the deepest possible leaf.
- depth is calculated from traversal from root to the given node.....

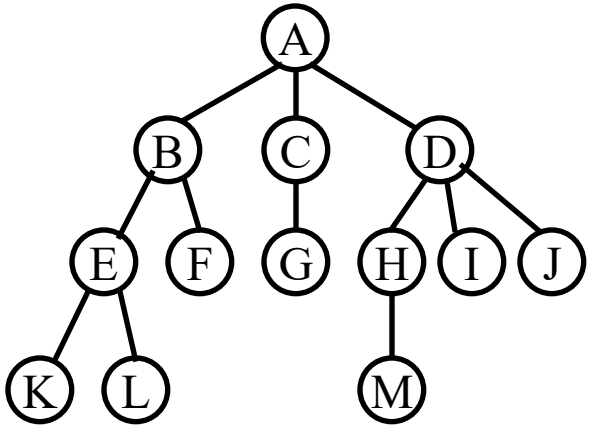


Representation of Tree

List Representation

Left Child-Right Sibling Representation

1. List Representation

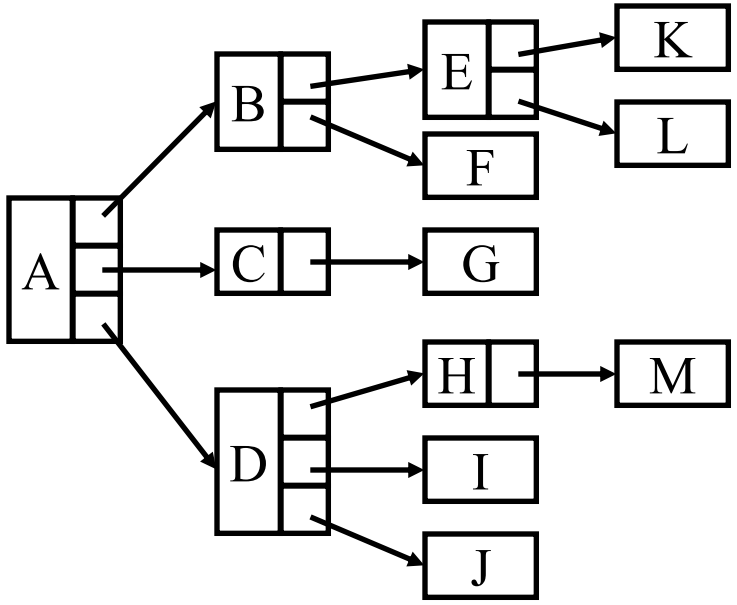


(A)

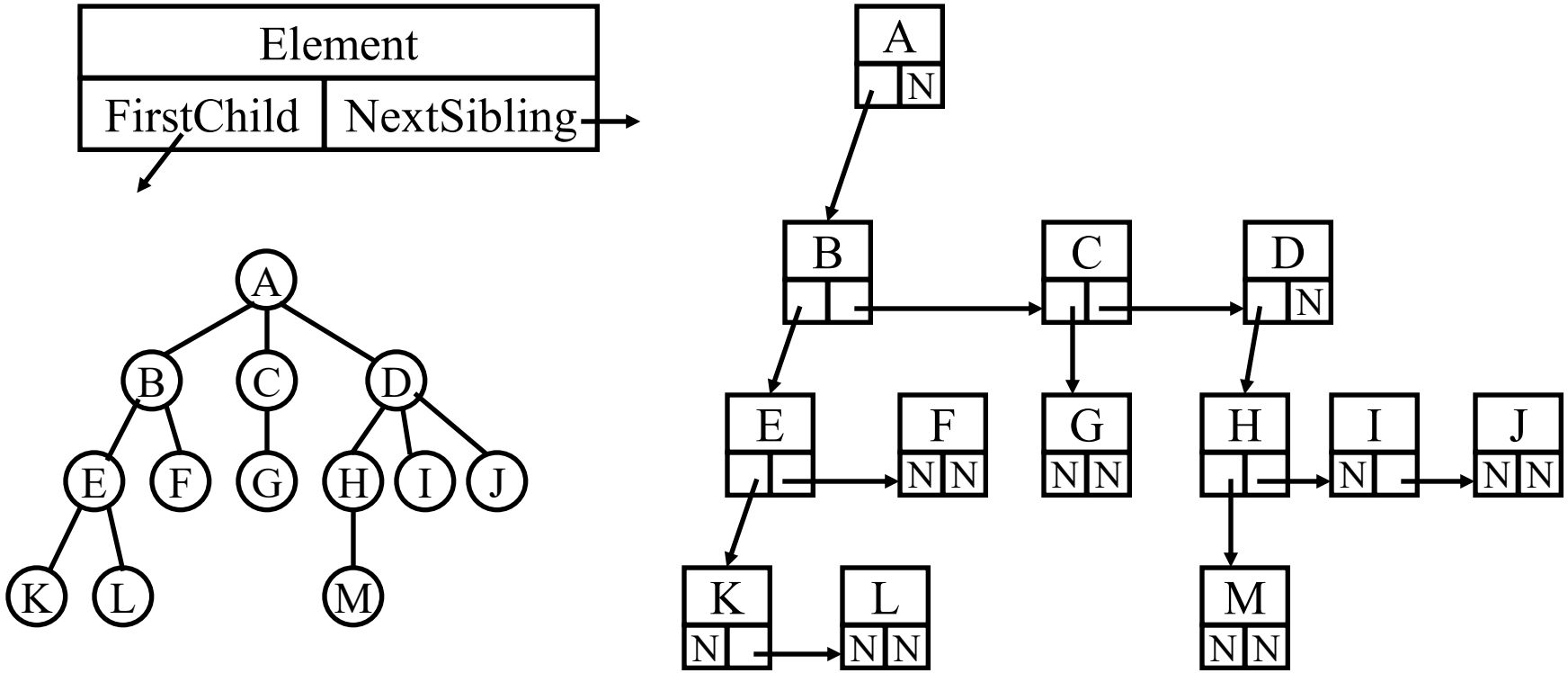
(A (B, C, D))

(A (B (E, F), C (G), D (H, I, J)))

(A (B (E (K, L), F), C (G), D (H (M), I, J)))



2. FirstChild-NextSibling Representation (Left Child Right Sibling)



Note: The representation is not unique since the children in a tree can be of any order.

Binary Tree

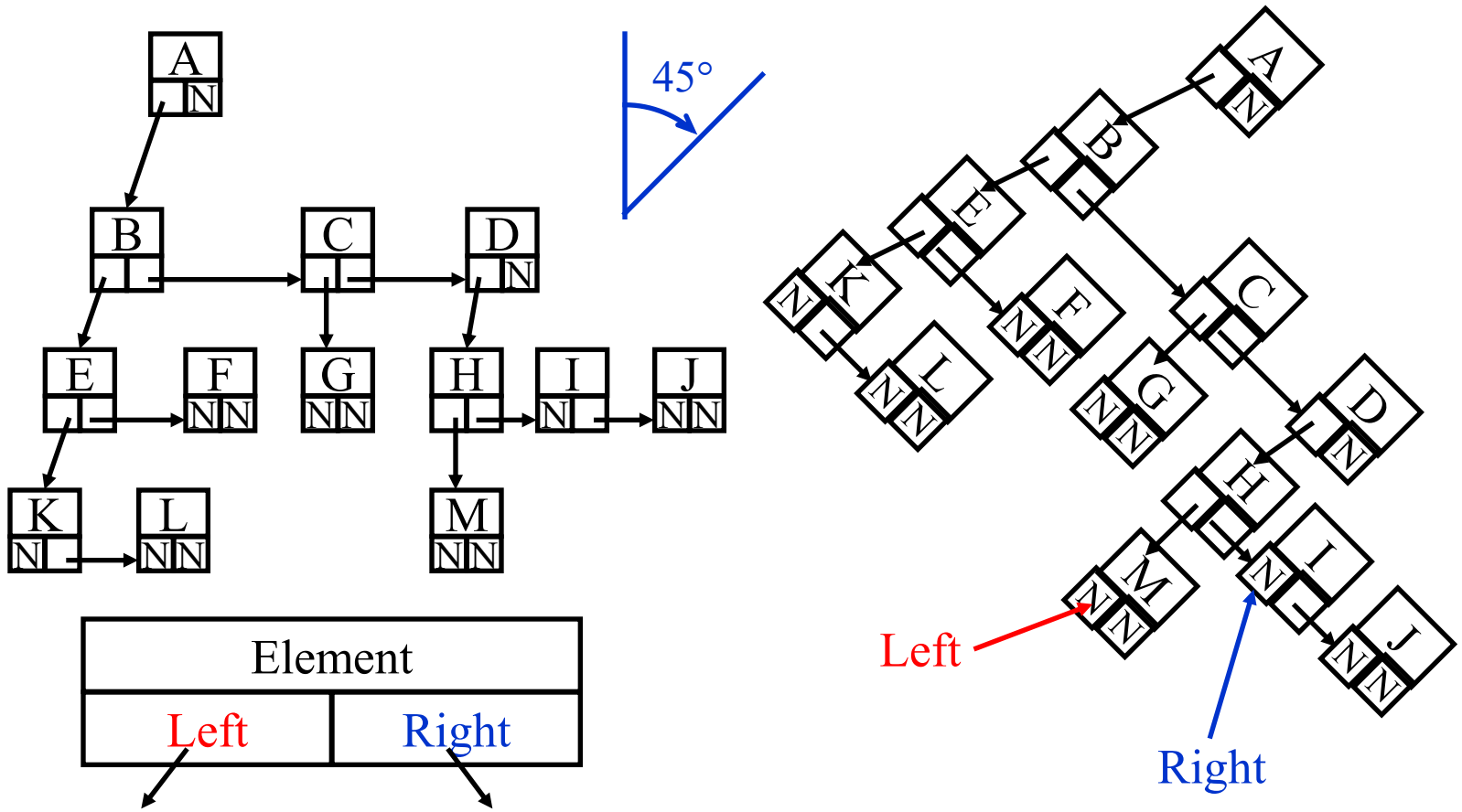
binary trees are an important type of tree structure that occurs very often.

The chief characteristic of a binary tree is the stipulation that the degree of any given node must not exceed two.

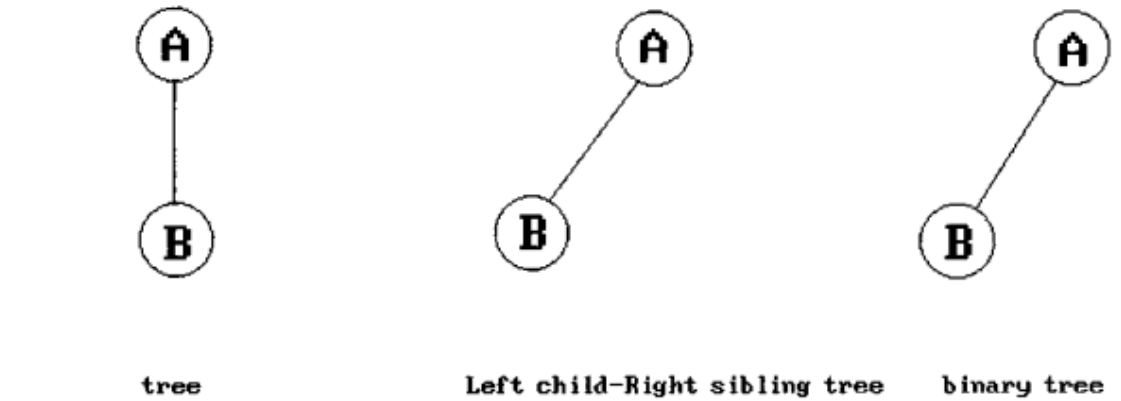
Binary Trees

A binary tree is a tree in which no node can have more than two children.

Rotate the FirstChild-NextSibling tree clockwise by 45°.



Tree Representation



Binary Tree ADT

The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT

Additional methods:

-position leftChild(p)

-position rightChild(p)

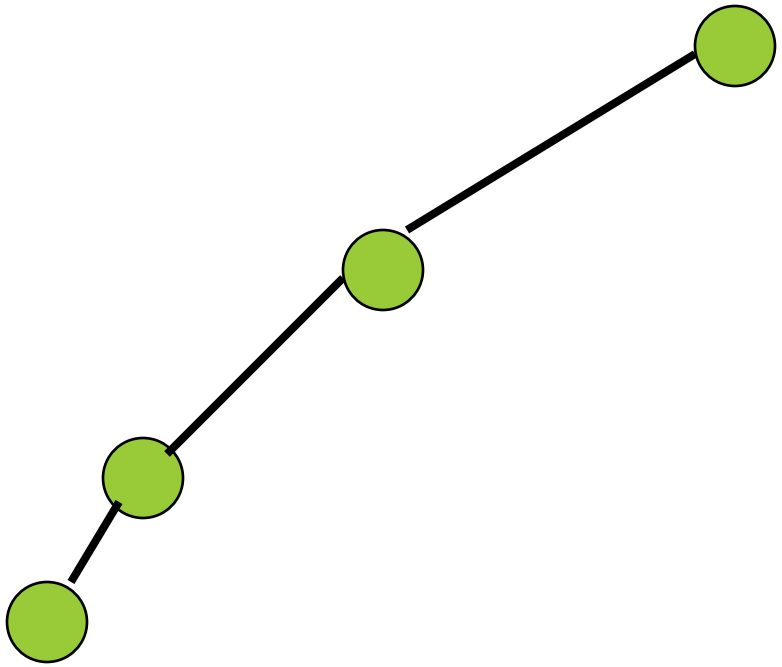
-position sibling(p)

Update methods may be defined by data structures implementing the BinaryTree ADT

Binary Tree Properties

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 in BT

- The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.
- The maximum number of nodes in a binary tree of depth k is 2^k-1, k>=1.

Prove by induction.

$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

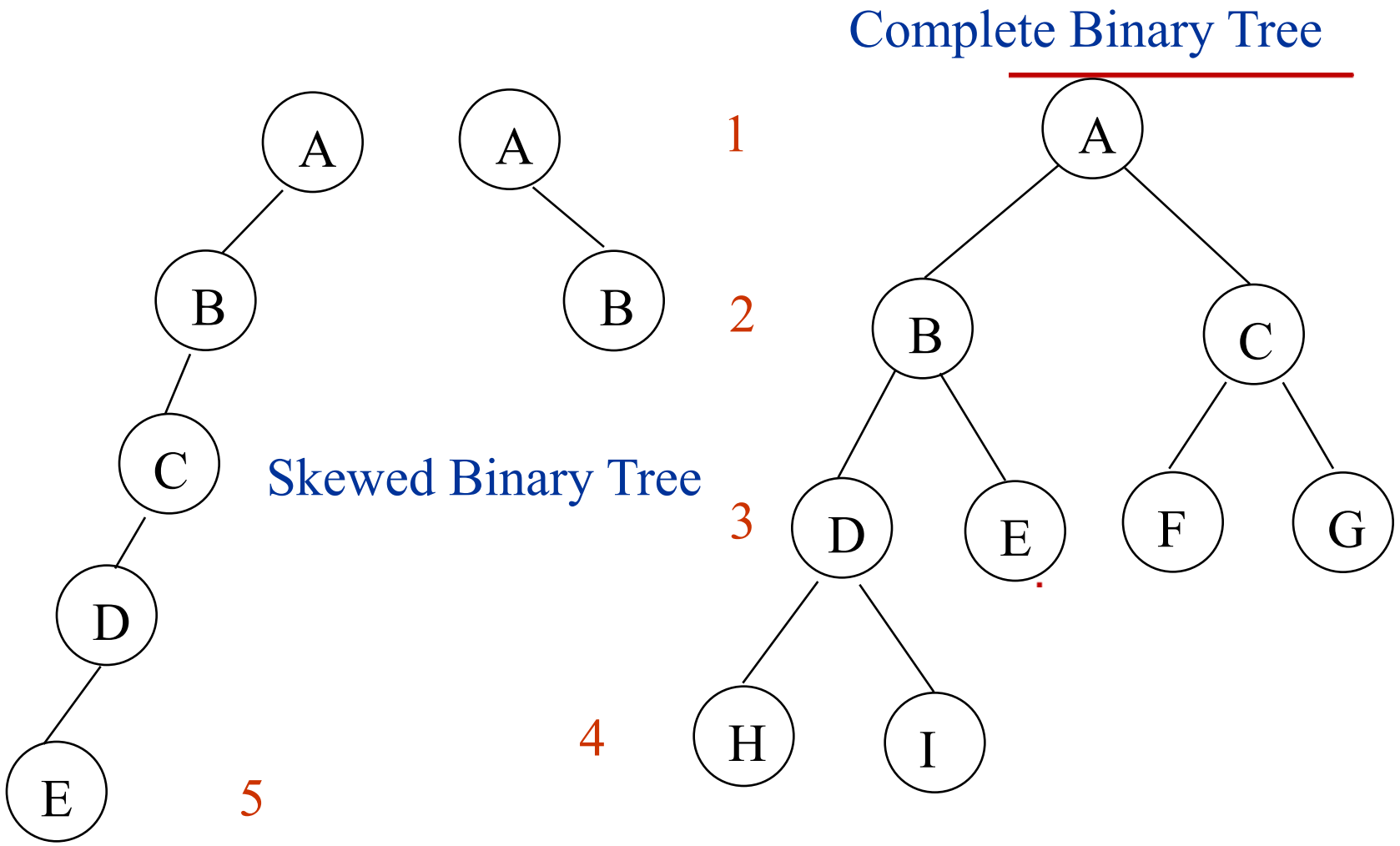
Full Binary Tree

•Definition: A binary tree in which every node has either 0 or 2 children (no node has only one child).
•Key Point: Each node is "full" — either it's a leaf or it has two children.
•Shape Constraint: No restriction on how "balanced" the tree is; it can be skewed in height as long as each node has 0 or 2 children.

Complete Binary Tree

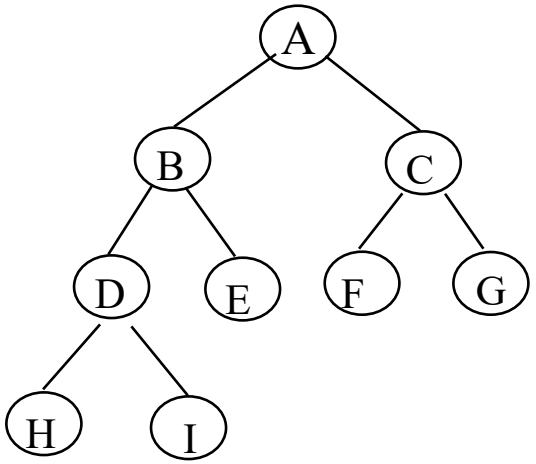
•Definition: A binary tree in which all levels are completely filled except possibly the last level, and in the last level, all nodes are as far left as possible.
•Key Point: Structure is almost like a "filled array representation" of a heap.
•Shape Constraint: Must be filled level by level, left to right.

Samples of Trees

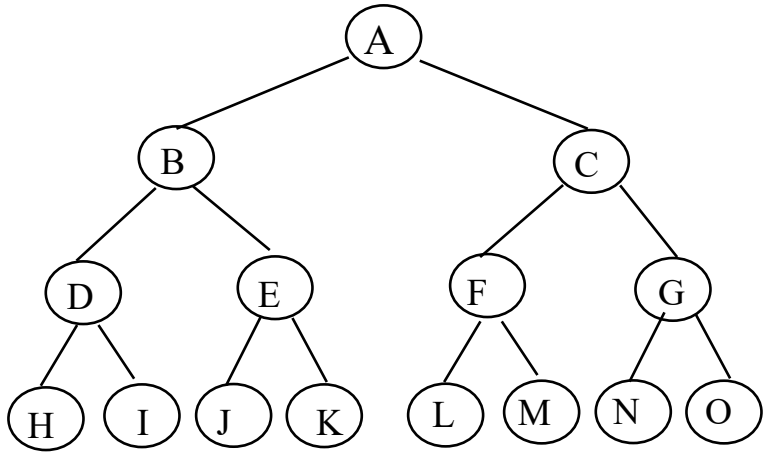


Full BT VS Complete BT

- A full binary tree of depth *k* is a binary tree of depth *k* having $2^k - 1$ nodes, *k*>=1.
- A binary tree with *n* nodes and depth *k* is complete *iff* all of its nodes are filled except possibly the last level, where in the last level the nodes are filled from left to right .



Complete binary tree



Full binary tree of depth 4

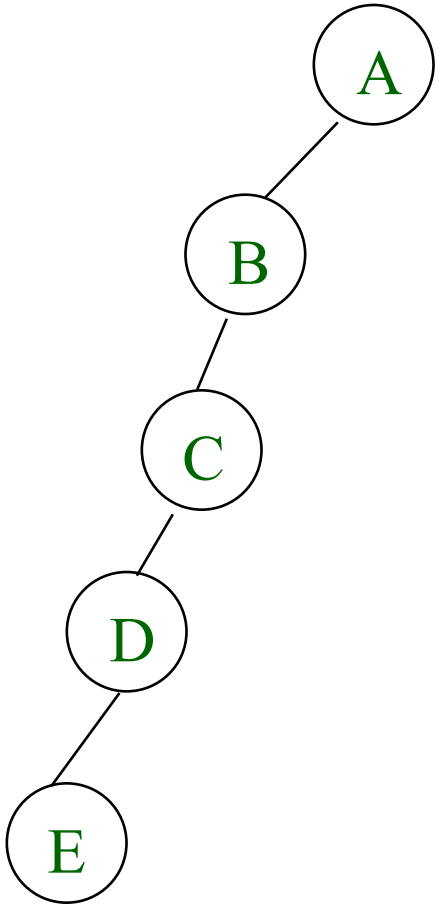
Key Difference

Feature	Full Binary Tree	Complete Binary Tree
Node Children	Each node has 0 or 2 children	Nodes can have 0, 1, or 2 children
Leaf Node Placement	Can be at different levels	Must be as far left as possible
Last Level Requirement	No specific requirement	Filled left to right
Structural Symmetry	Focused on node completeness	Focused on level-wise filling

Binary Tree Representation

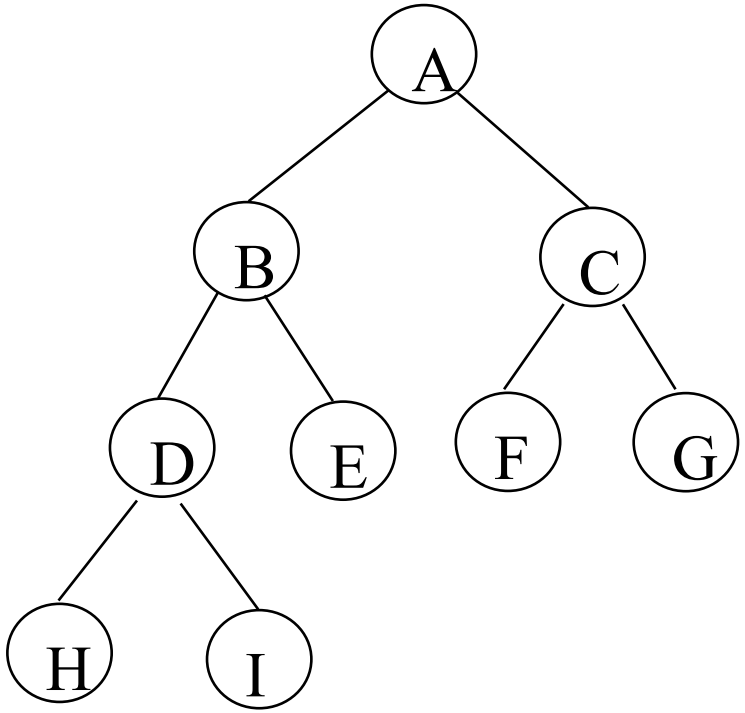
- Array representation.
- Linked representation.

Sequential Representation



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

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



[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Binary Tree Representations

- If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1<=i<=n$, we have:
  - *parent*(i) is at $i/2$ if i!=1. If i=1, i is at the root and has no parent.
  - *left_child*(i) is at $2i$ if $2i<=n$. If $2i>$n, then i has no left child.
  - *right_child*(i) is at $2i+1$ if $2i+1<=n$. If $2i+1>$n, then i has no right child.

Advantages and disadvantages of Array representation

Advantages:

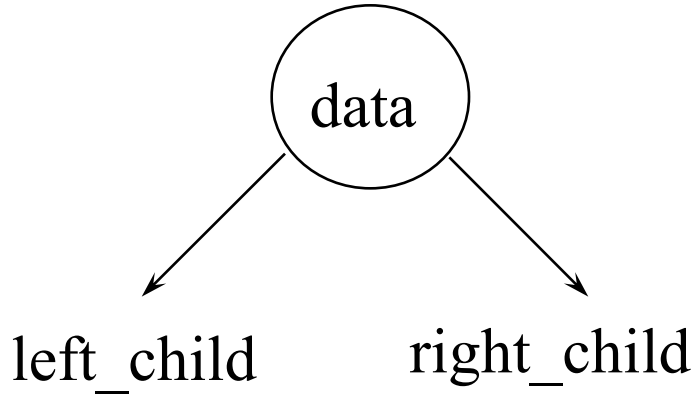
1. This representation is very easy to understand.
2. This is the best representation for full and complete binary tree representation.
3. Programming is very easy.
4. It is very easy to move from a child to its parents and vice versa.

Disadvantages:

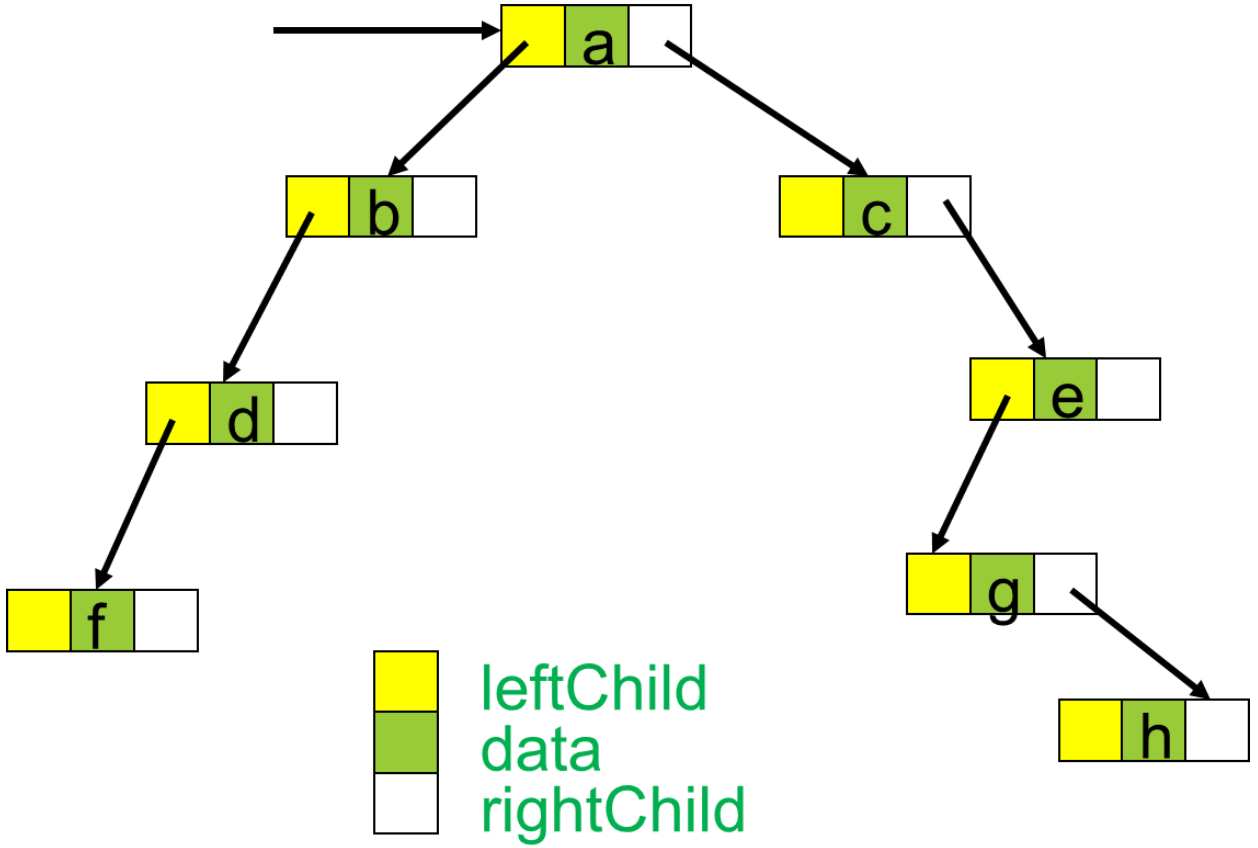
1. Lot of memory area wasted.
2. Insertion and deletion of nodes needs lot of data movement.
3. This is not suited for trees other than full and complete tree.

Linked Representation

```
struct node {
 int data;
 node *left_child, *right_child;
};
```



Linked Representation Example



Advantages and disadvantages of linked representation

Advantages

1. A particular node can be placed at any location in the memory.
2. Insertions and deletions can be made directly without data movements.
3. It is best for any type of trees.
4. It is flexible because the system take care of allocating and freeing of nodes.

Disadvantage

1. It is difficult to understand.
2. Additional memory is needed for storing pointers
3. Accessing a particular node is not easy.

Implementation

```
// Node structure
typedef struct Node {
    int data;
    struct Node *lchild;
    struct Node *rchild;
} *Nodeptr;
```

```
// Function to allocate a new node
Nodeptr getnode() {
    Nodeptr temp = (Nodeptr)malloc(sizeof(struct Node));
    if (!temp) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    temp->lchild = temp->rchild = NULL;
    return temp;
}
```

Implementation

```
// Recursive function to create a binary tree
Nodeptr CreateBinaryTree(int item) {
    int x;
    if (item != -1) {   // -1 indicates no node
        Nodeptr temp = getnode();
        temp->data = item;
        printf("Enter the lchild of %d (-1 for no child): ", item);
        scanf("%d", &x);
        temp->lchild = CreateBinaryTree(x);
        printf("Enter the rchild of %d (-1 for no child): ", item);
        scanf("%d", &x);
        temp->rchild = CreateBinaryTree(x);
        return temp;
    }
    return NULL;
}
```

Binary Tree Traversals

- Binary tree operations often require **traversing the tree**.
- In a **traversal**, each node is visited **exactly once**.
- During the visit, any operation on the node can be performed, such as:
  - Displaying the node
  - Cloning the node
  - Evaluating operators in an expression tree

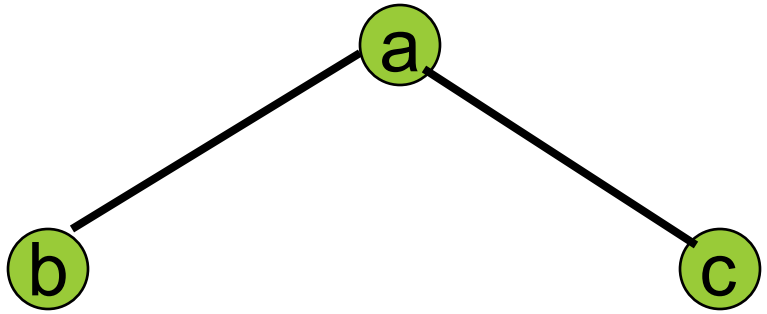
Types of Binary Tree Traversals

- Preorder Traversal (Root → Left → Right)
- Inorder Traversal (Left → Root → Right)
- Postorder Traversal (Left → Right → Root)
- Level-order Traversal (Breadth-first)

Pre order Traversal-Recursive

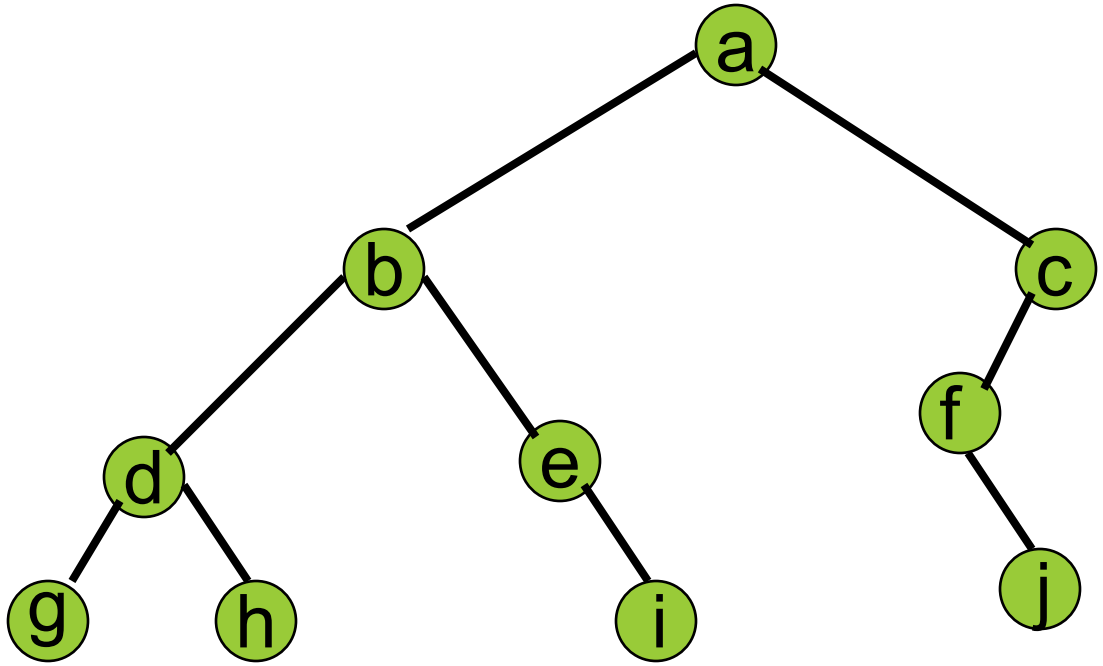
```
void preorderTraversal(struct Node* node)
{
    if (node == NULL) {
        return;    }
    printf("%d ", node->data);            // Print the data of the current node
    preorderTraversal(node->left);        // Recursively traverse the left subtree
    preorderTraversal(node->right);       // Recursively traverse the right subtree
}
```

Preorder Example (Visit = print)



a b c

Preorder Example (Visit = print)

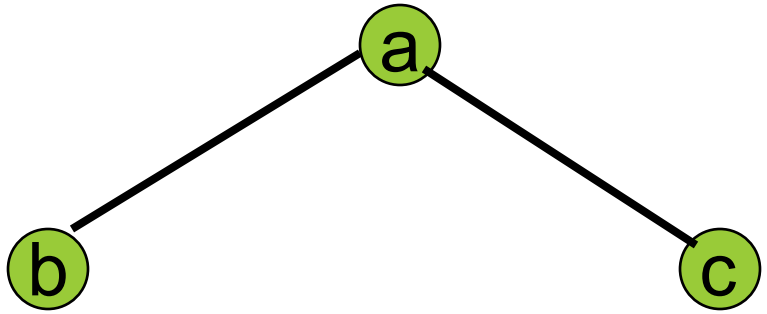


a b d g h e i c f j

In order Traversal- Recursive

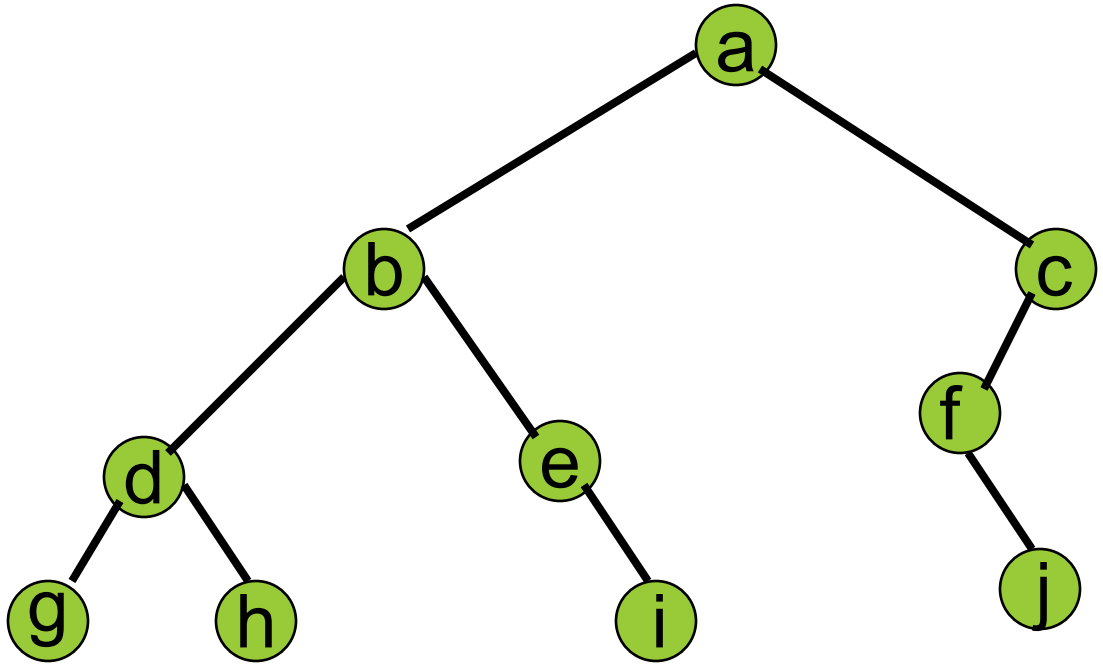
```
// Function to perform inorder traversal on a binary tree
void inorderTraversal(struct Node* node) {
    if (node == NULL) {
        return;
    }
    inorderTraversal(node->left);          // Recursively traverse the left subtree
    printf("%d ", node->data);             // Print the data of the current node
    inorderTraversal(node->right);         // Recursively traverse the right subtree
}
```

Inorder Example (Visit = print)



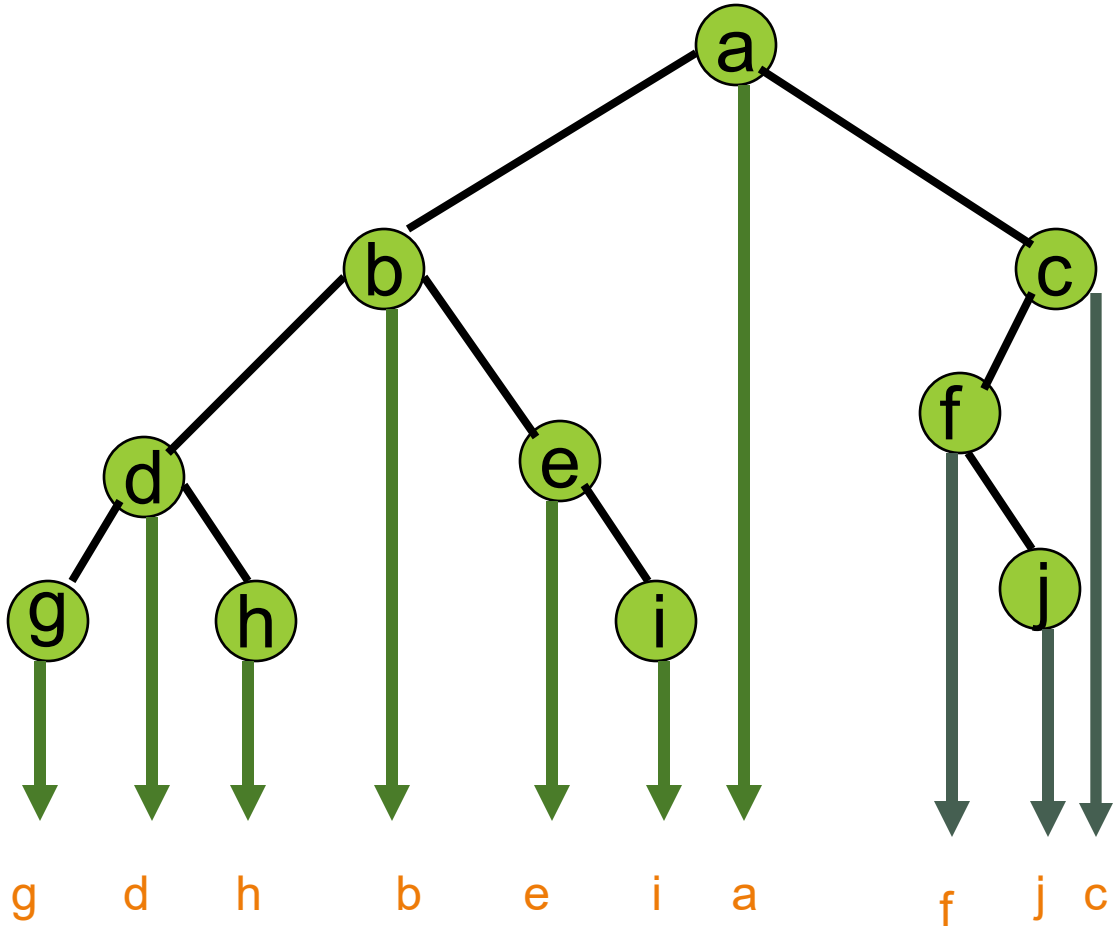
b a c

Inorder Example (Visit = print)



g d h b e i a f j c

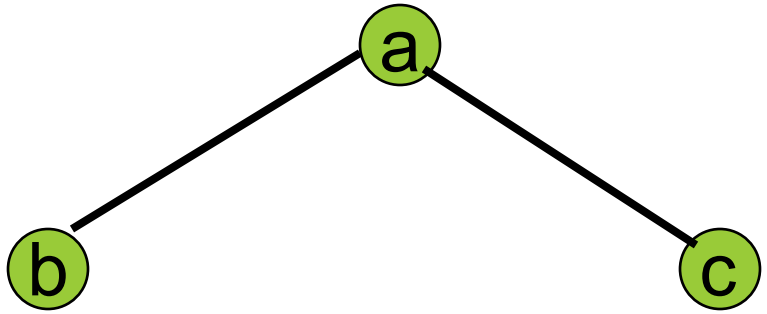
Inorder By Projection (Squishing)



Post order Traversal- Recursive

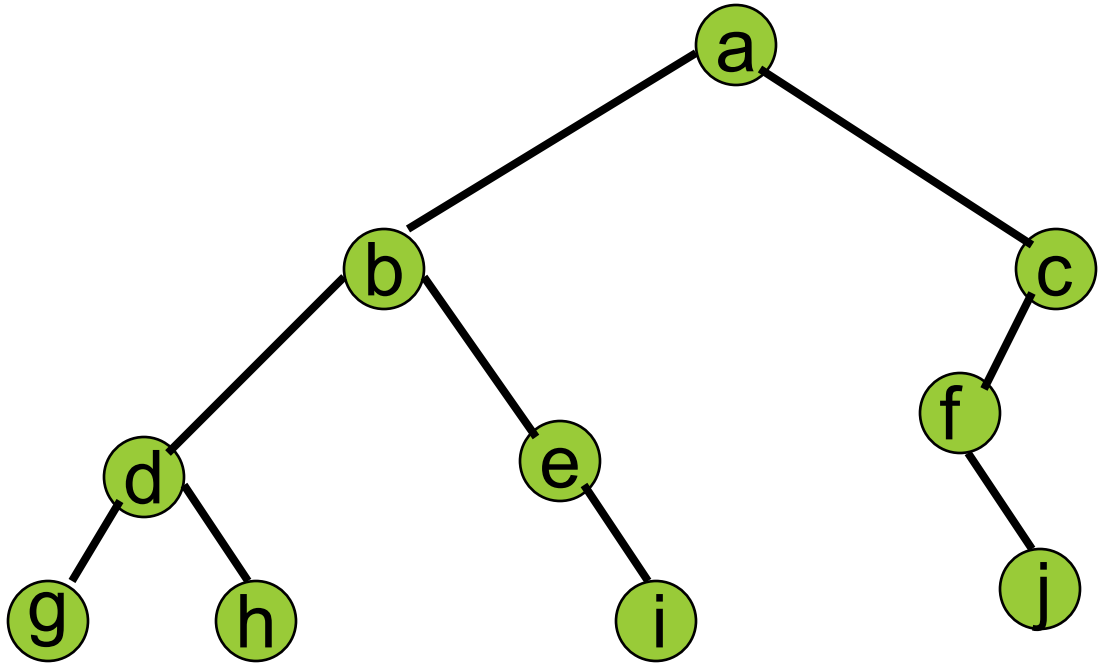
```
// Function to perform postorder traversal on a binary tree
void postorderTraversal(struct Node* node) {
    if (node == NULL) {
        return;
    }
    postorderTraversal(node->left);        // Recursively traverse the left subtree
    postorderTraversal(node->right);       // Recursively traverse the right subtree
    printf("%d ", node->data);             // Print the data of the current node
}
```

Postorder Example (Visit = print)



b c a

Postorder Example (Visit = print)



g h d i e b j f c a

Pre order Traversal- Iterative

```
// Function for pre-order traversal (Iterative)
void preOrderTraversal(struct Node* root) {
    if (root == NULL)
        return;
    struct Node* stack[100];
    int top = -1;
    stack[++top] = root;
    while (top != -1) {
        struct Node* current = stack[top--];
        printf("%d ", current->data);
        if (current->right != NULL)
            stack[++top] = current->right;
        if (current->left != NULL)
            stack[++top] = current->left;   }}
```

In order Traversal- Iterative

```
// Function for in-order traversal (Iterative)
void inOrderTraversal(Node* root) {
    Node* stack[MAX_SIZE];
    int top = -1;
  Node* current = root;
 while (current != NULL || top > -1) // pop
 {
 while (current != NULL) // travel leftmost of current subtree
            {stack[++top] = current;
            current = current->left;    }
        current = stack[top--];
        printf("%d ", current->data);
        current = current->right;
    } }
```

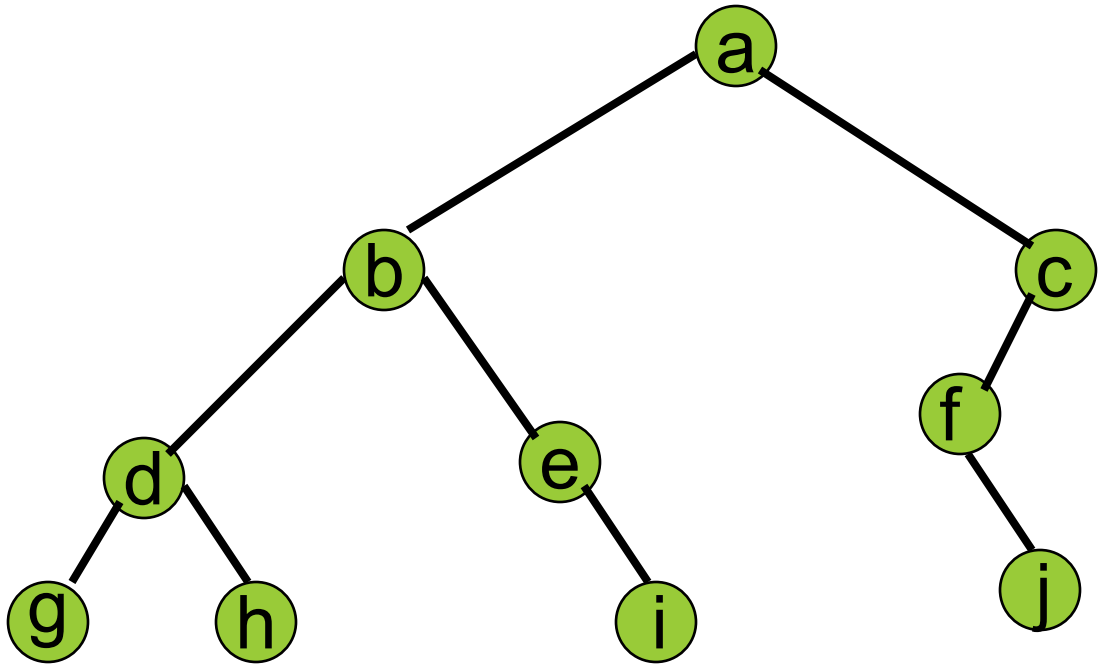
Post order Traversal- Iterative

```
// Function for post-order traversal (Iterative)

void postorder(struct Node* root) {
    if (root == NULL)
        return;
    struct Node* current = root;
    struct Node* prev = NULL;
    struct Node* stack[100];
    int top = -1;
    do {        while (current != NULL) {
            stack[++top] = current;
            current = current->left;
 }
```

Post order Traversal-Iterative

```
while (current == NULL && top != -1) {
    current = stack[top];
    // If the right child is either not present or has been visited
    if (current->right == NULL || current->right == prev)
{
        printf("%d ", current->data);      top--;
        prev = current;
        current = NULL;            }
    // Else traverse the right child
    else {
        current = current->right;
    }
}    } while (top != -1);}
```

Traversal Applications



- Make a clone.
- Determine height.
- Determine number of nodes.

Level Order Traversal

- Unlike preorder, inorder, and postorder (which are DFS traversals), Level Order Traversal is a BFS traversal that visits nodes level by level from left to right. It uses a queue for implementation.

- It is a breadth-first search (BFS) technique where we visit all nodes at each level from left to right before moving to the next level

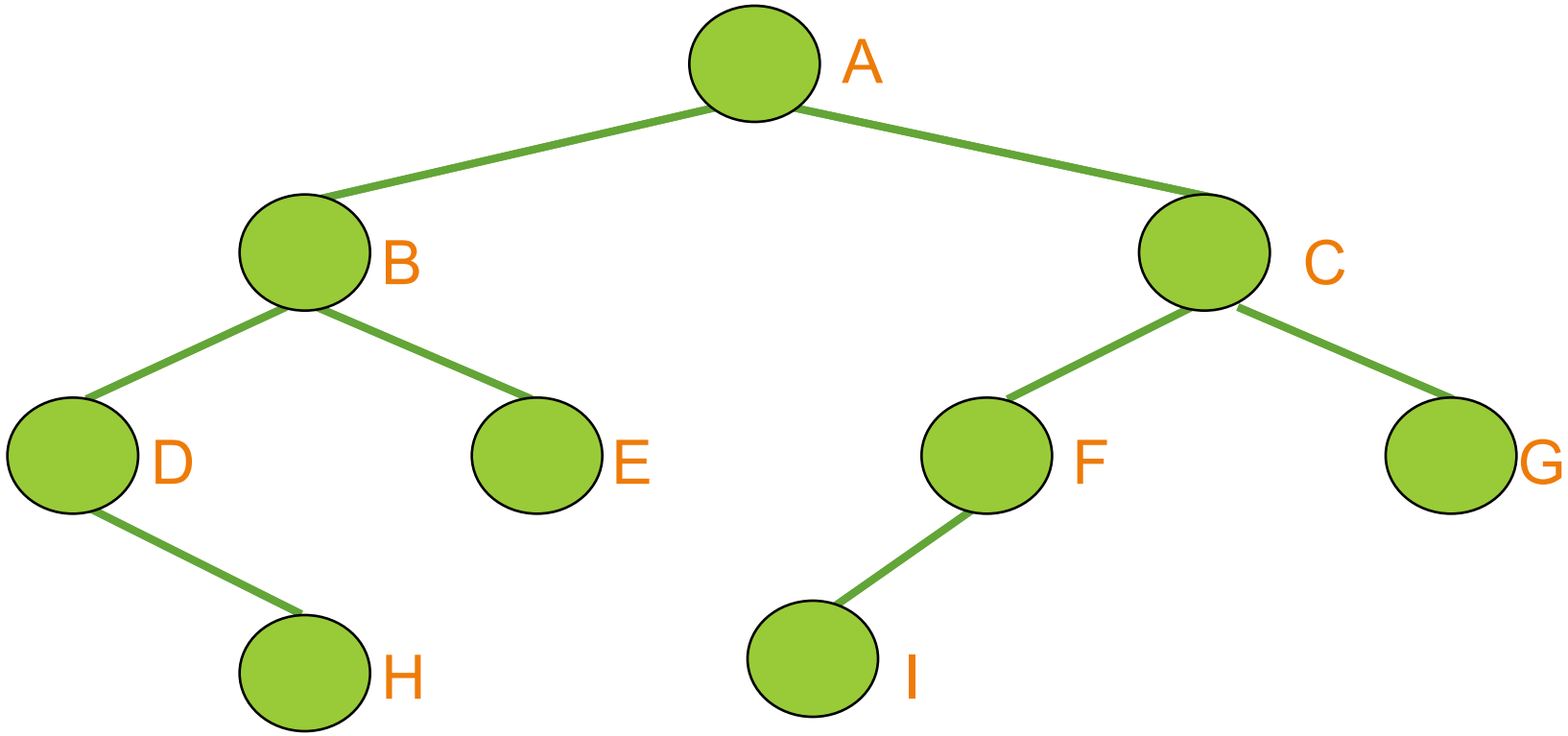
- Start at the root
- Visit the nodes at each level, from left to right

Algorithm for Level Order Traversal

The standard way to implement level order traversal is by using a queue.

1. Start from the root node and initialize a queue.
2. Push the root node into the queue.
3. While the queue is not empty:
   - Dequeue a node from the front of the queue and print its value.
   - If the dequeued node has a left child, enqueue the left child.
   - If the dequeued node has a right child, enqueue the right child.
4. Continue this process until the queue becomes empty, meaning all nodes have been processed.

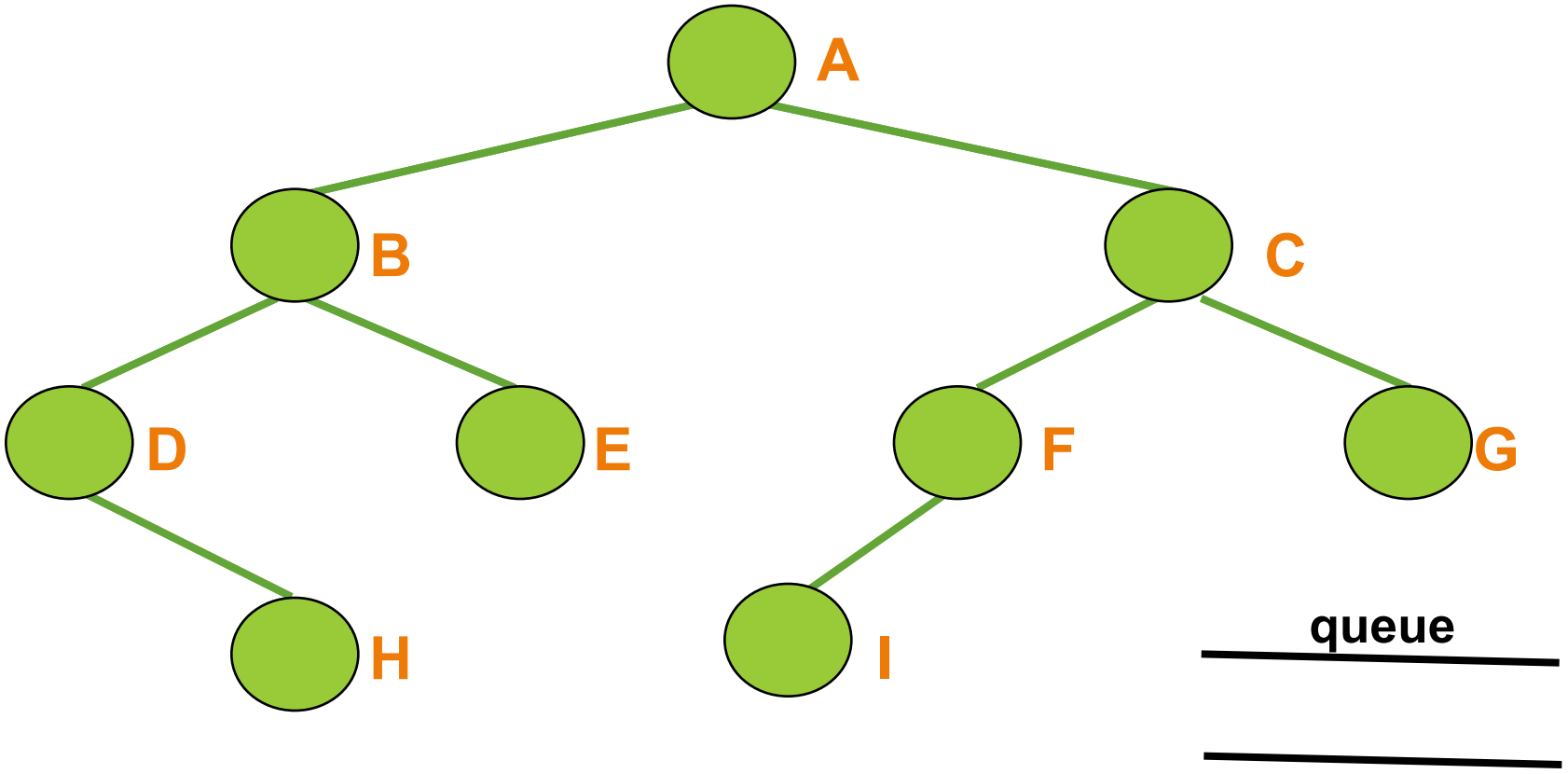
Level Order Traversal



Nodes will be visited in the order ABCDEFGHI

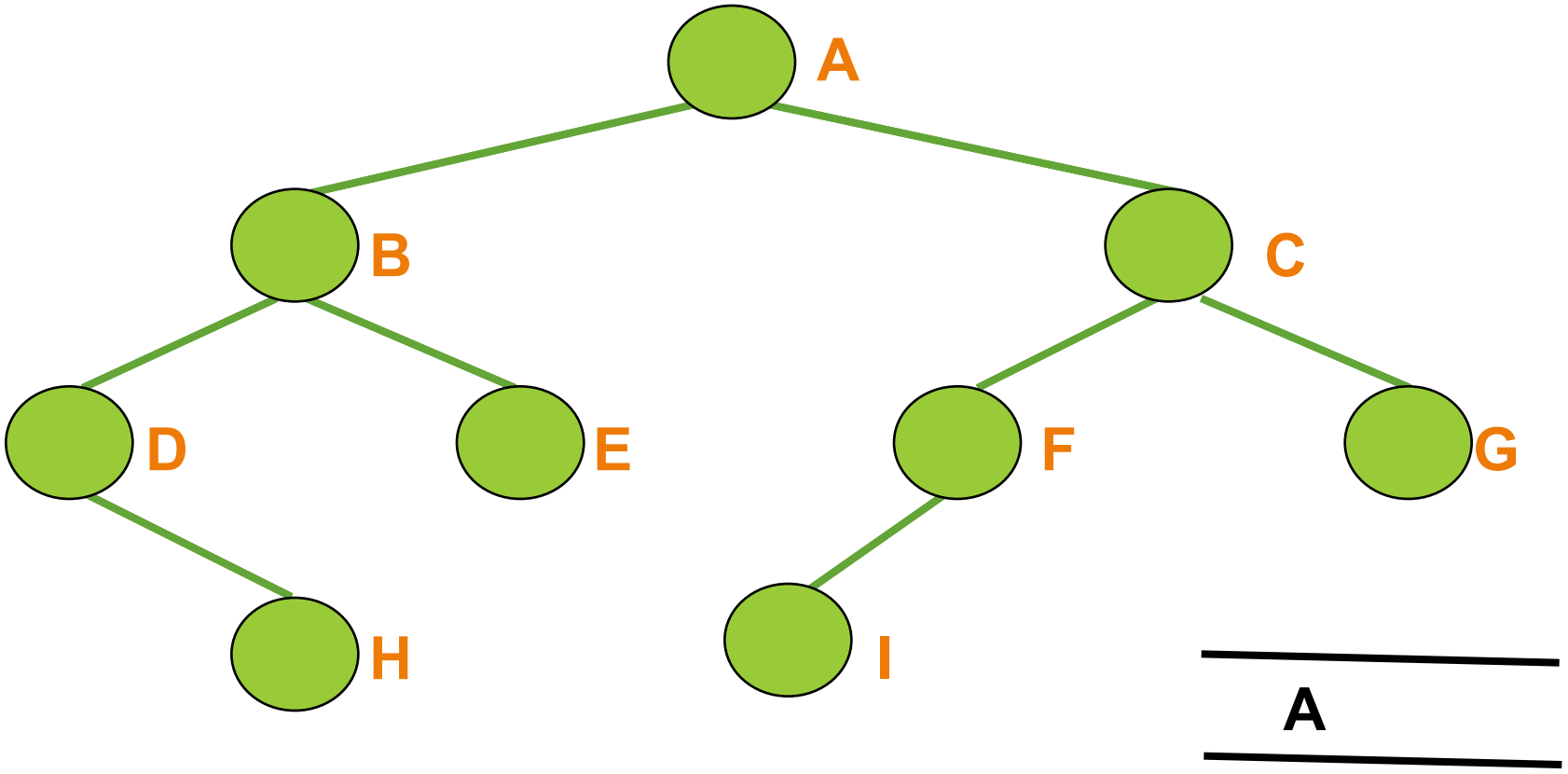
Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



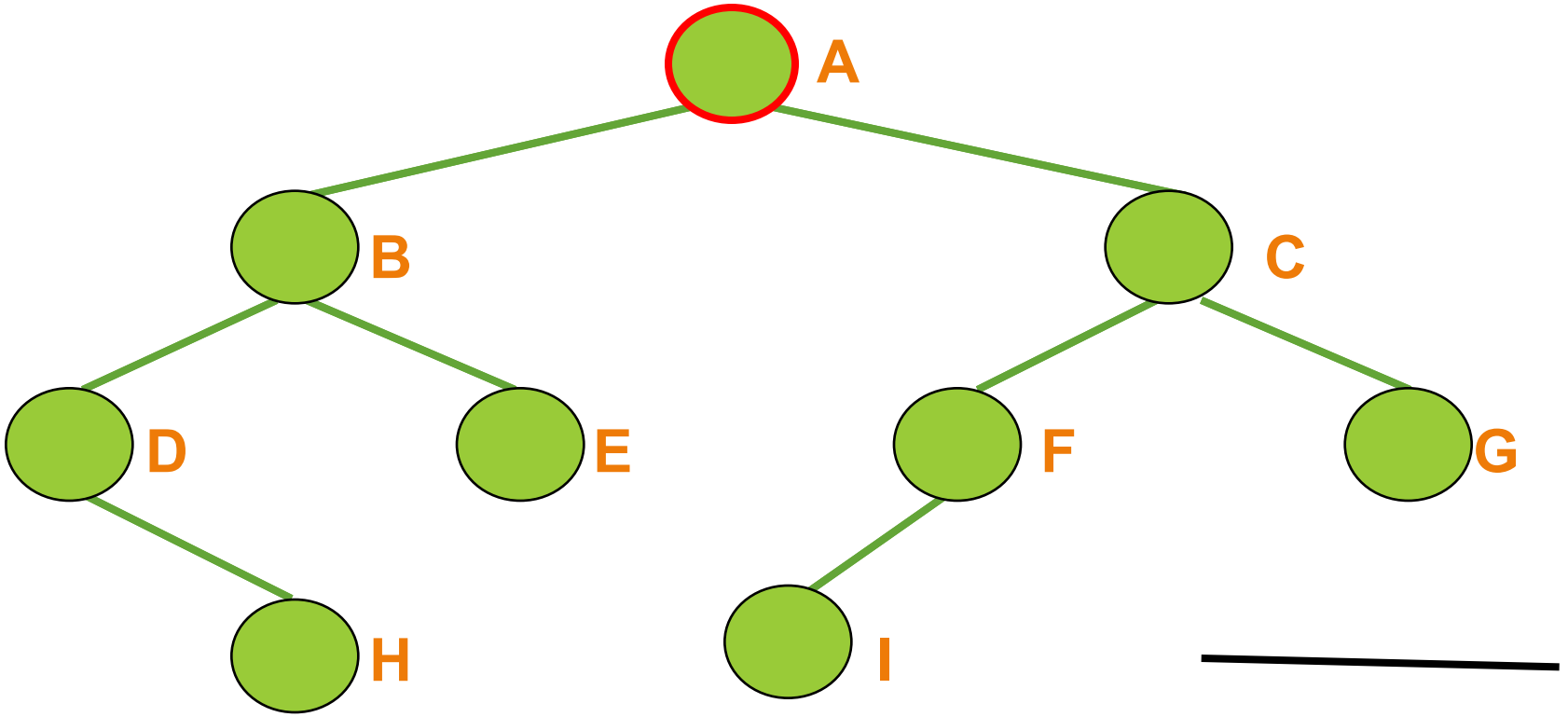
Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



Level Order Traversal

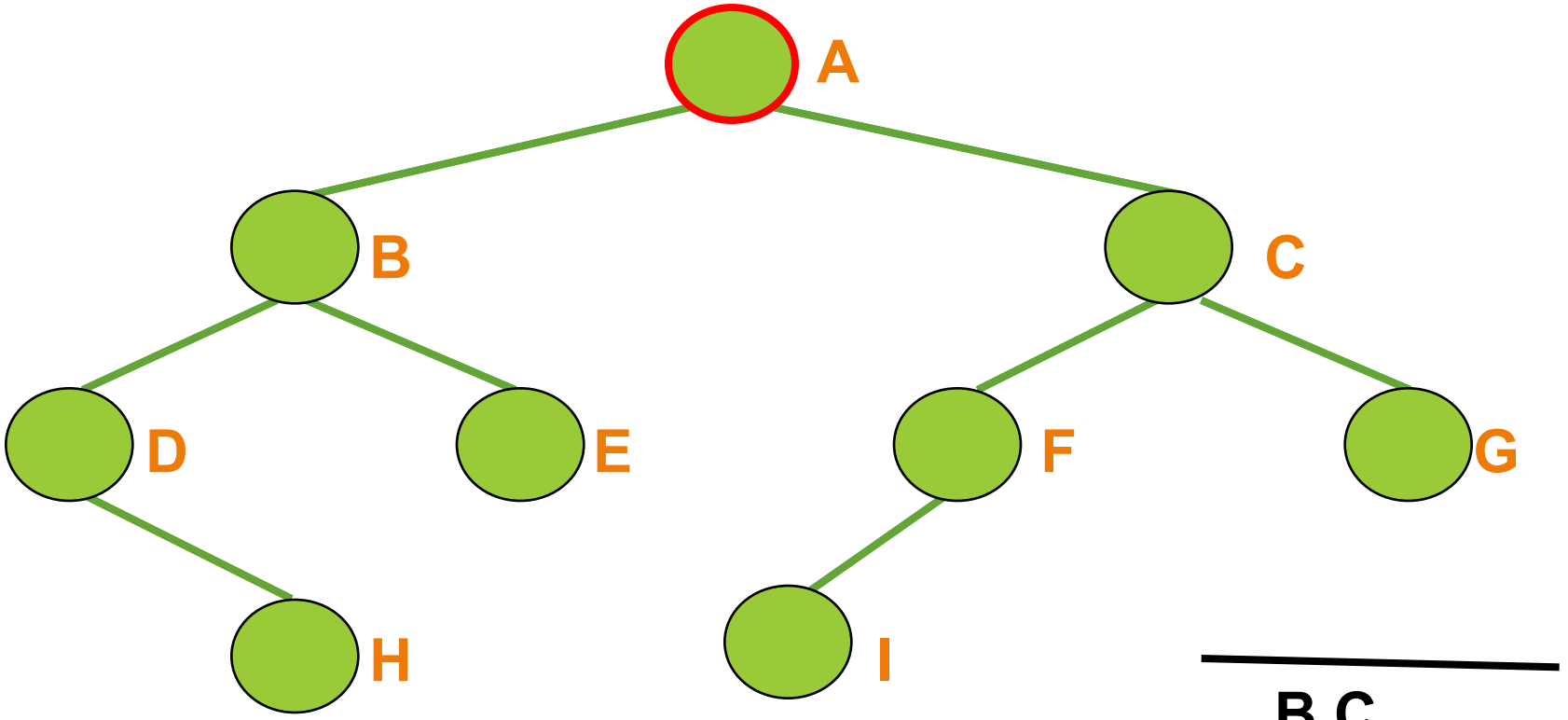
- Start at the root
- Visit the nodes at each level, from left to right



A

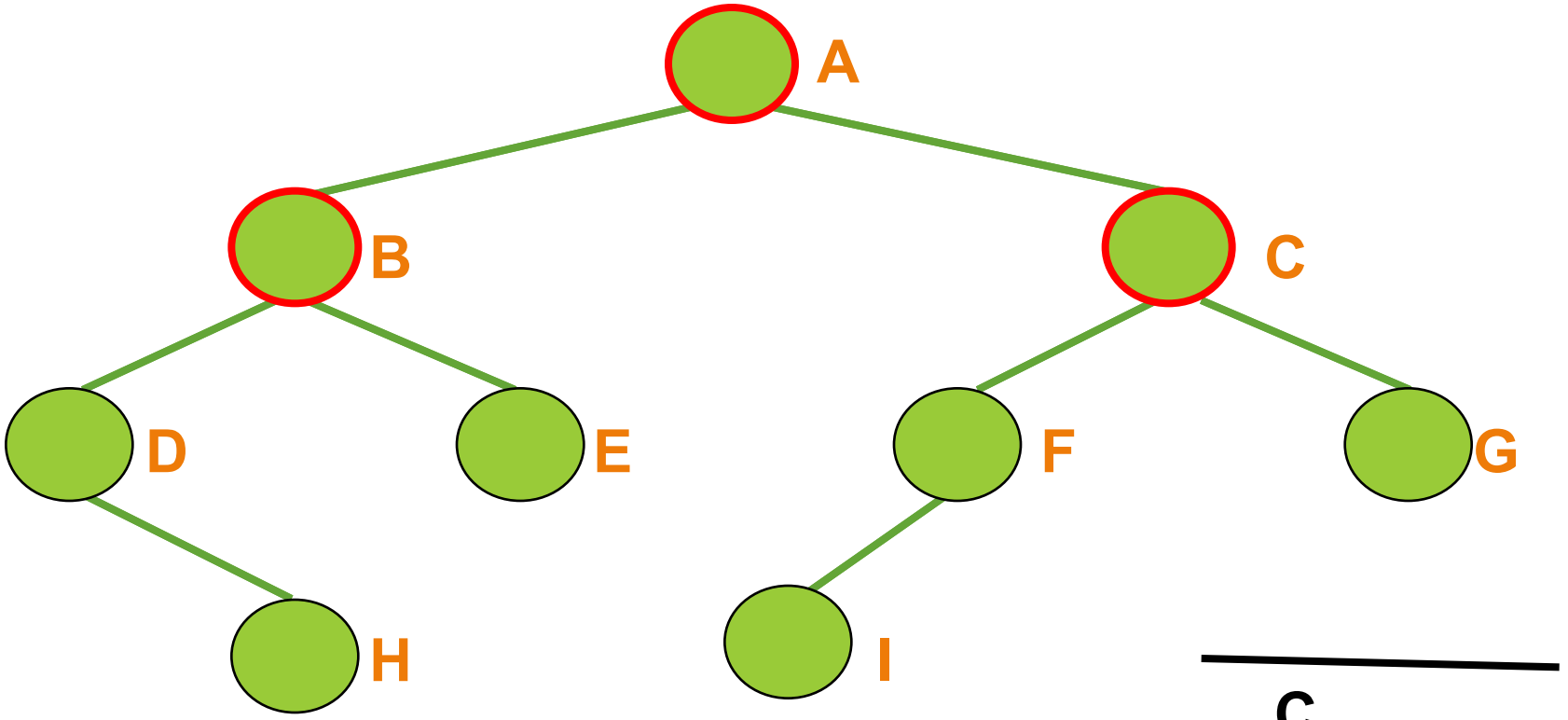
Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



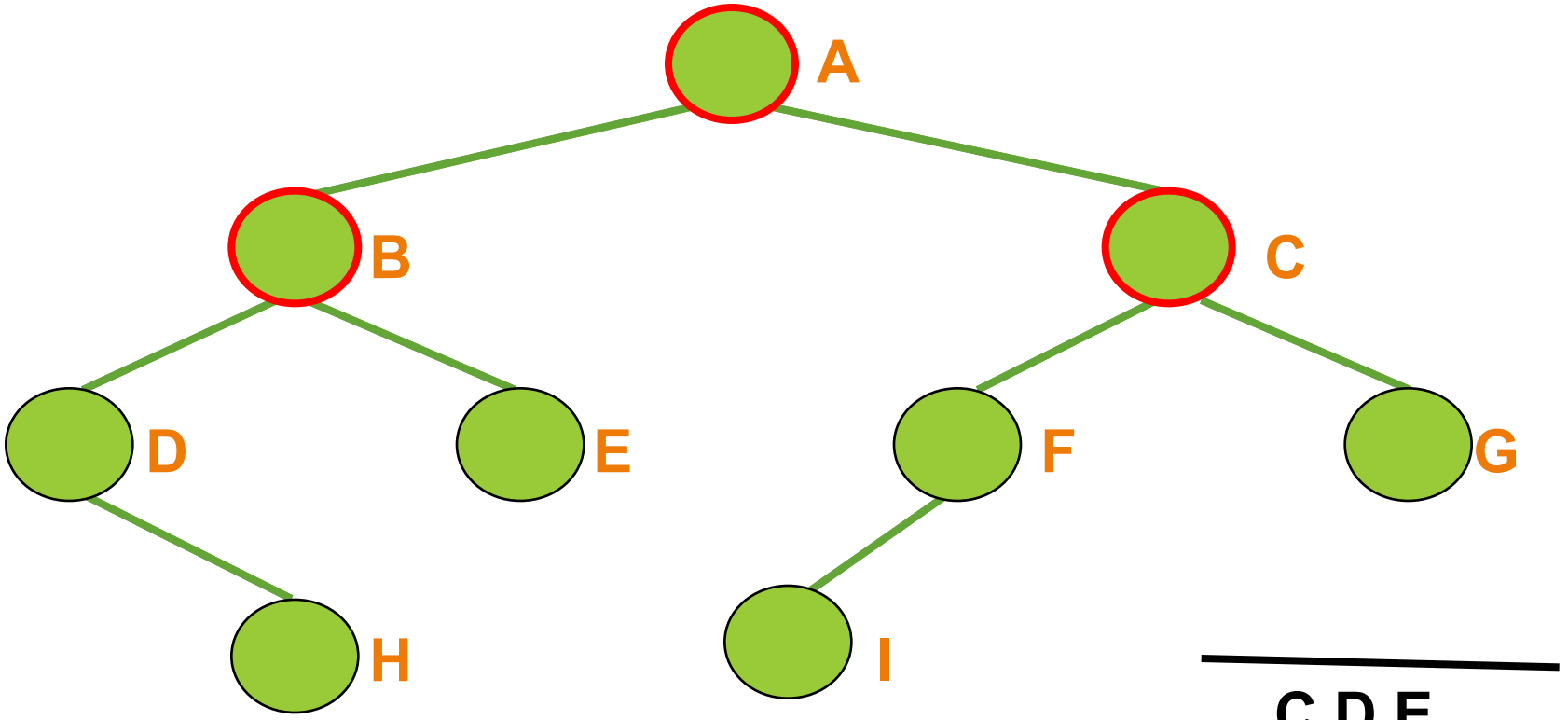
Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



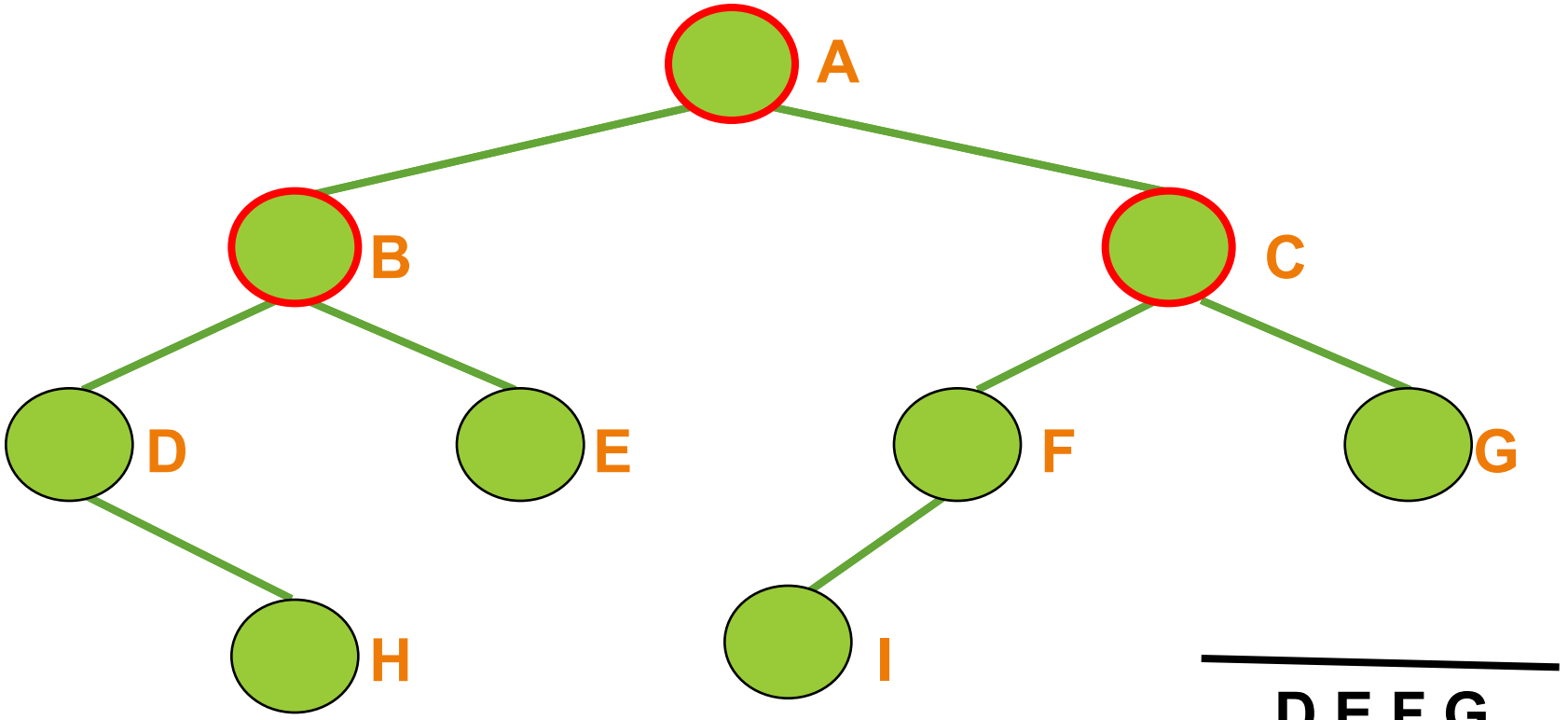
Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



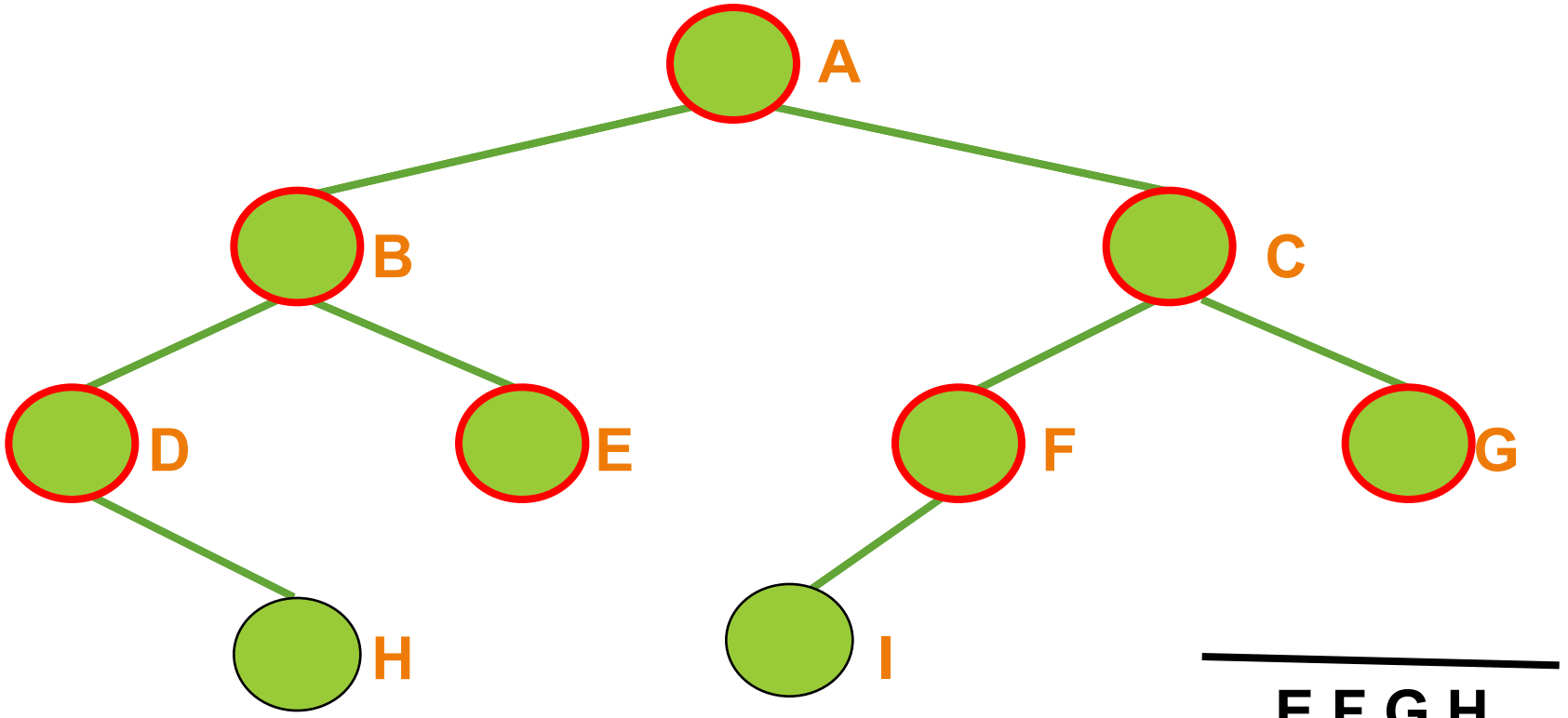
Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



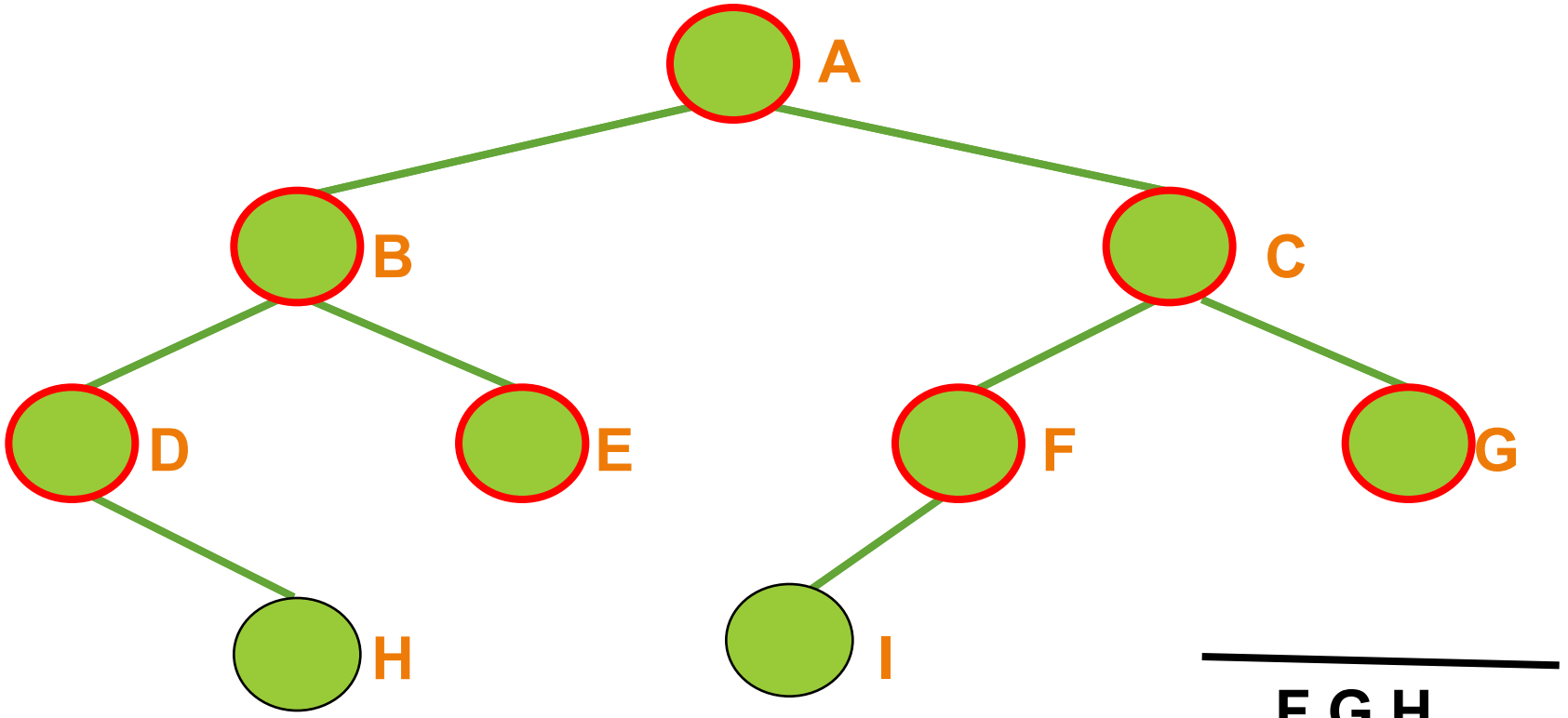
Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



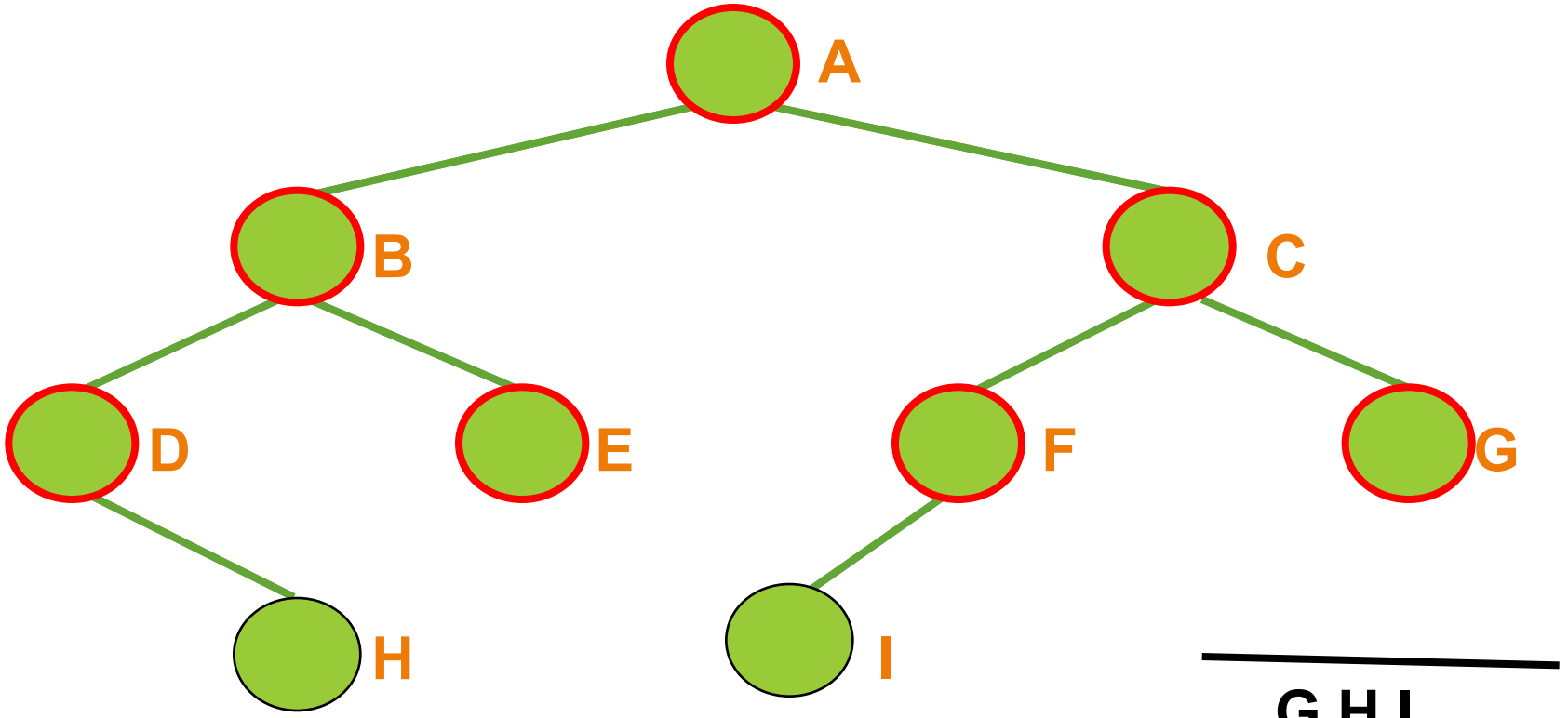
Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



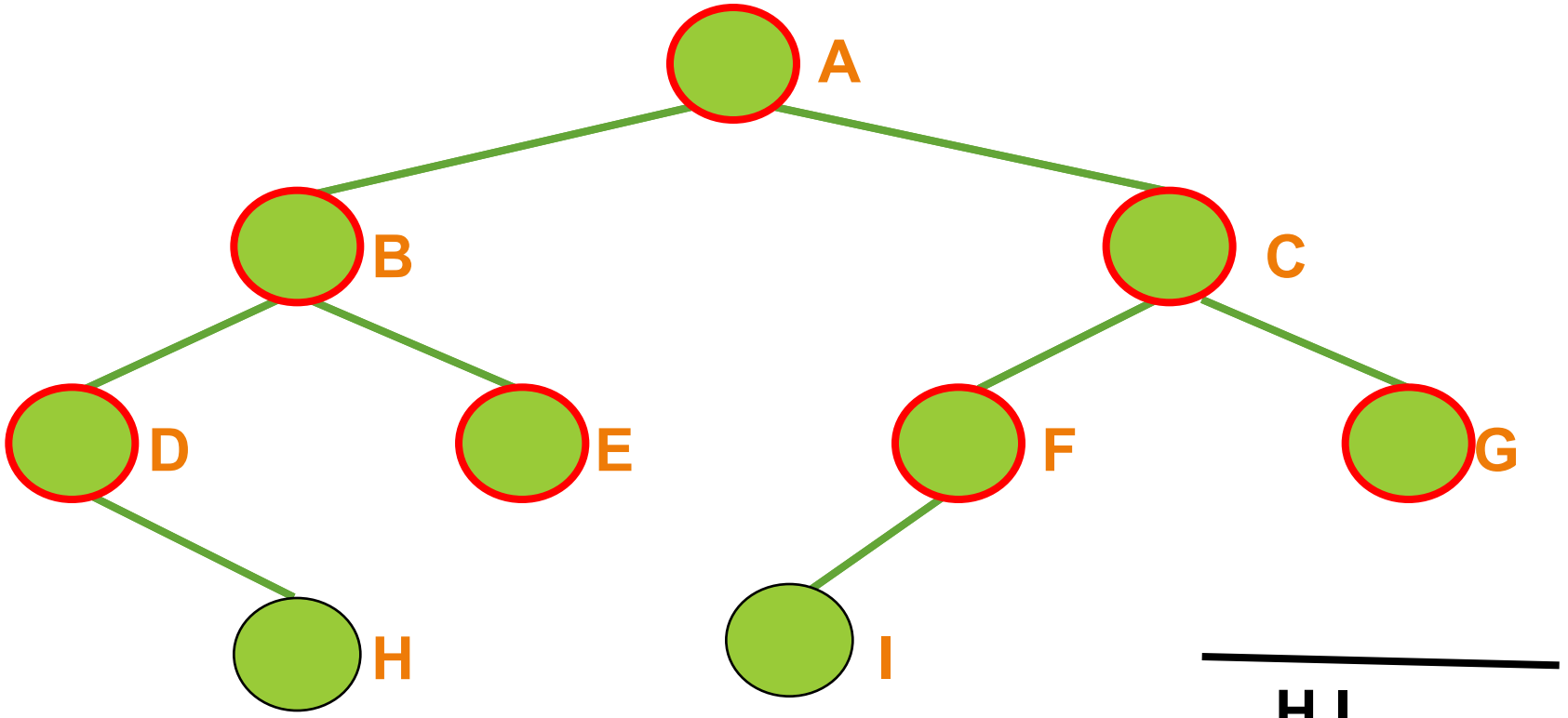
Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



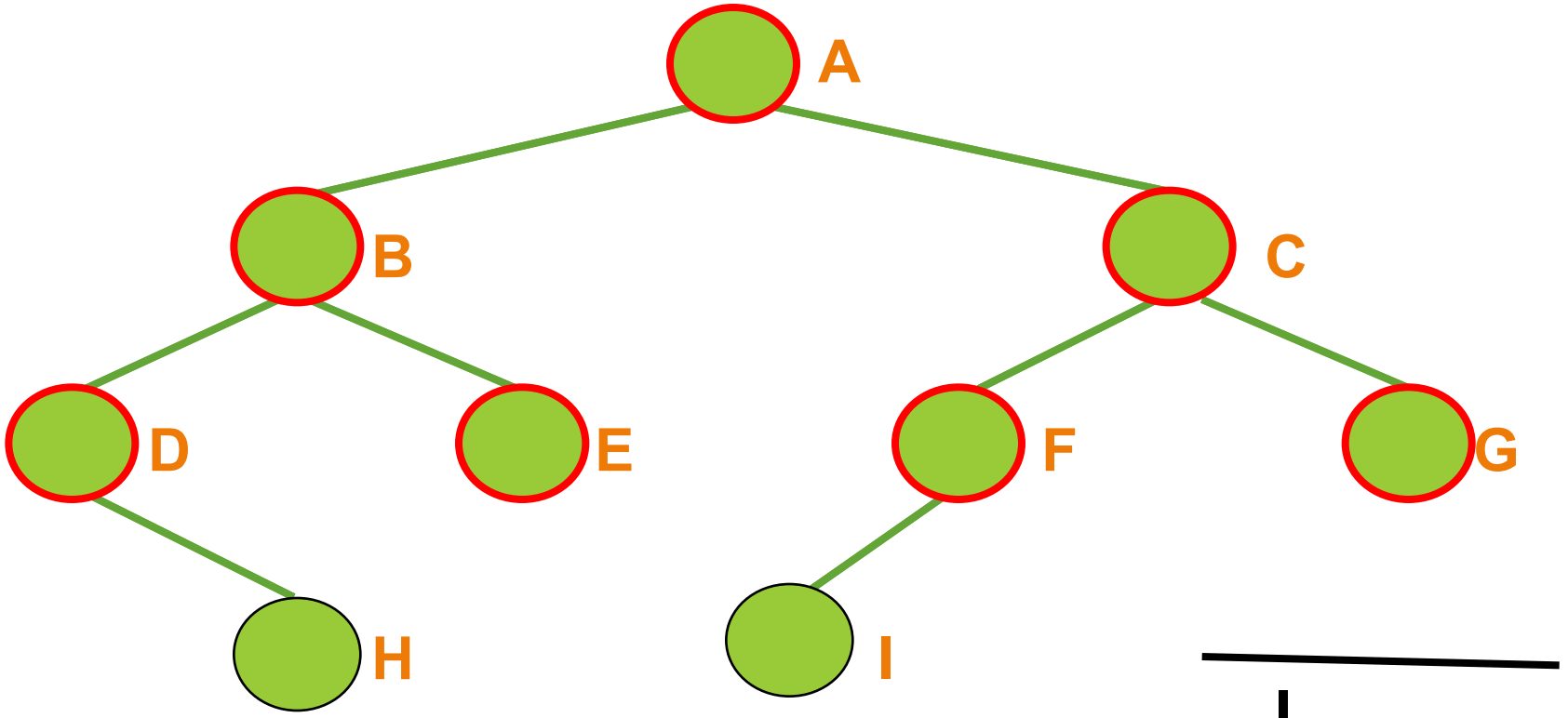
Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



Level Order Traversal

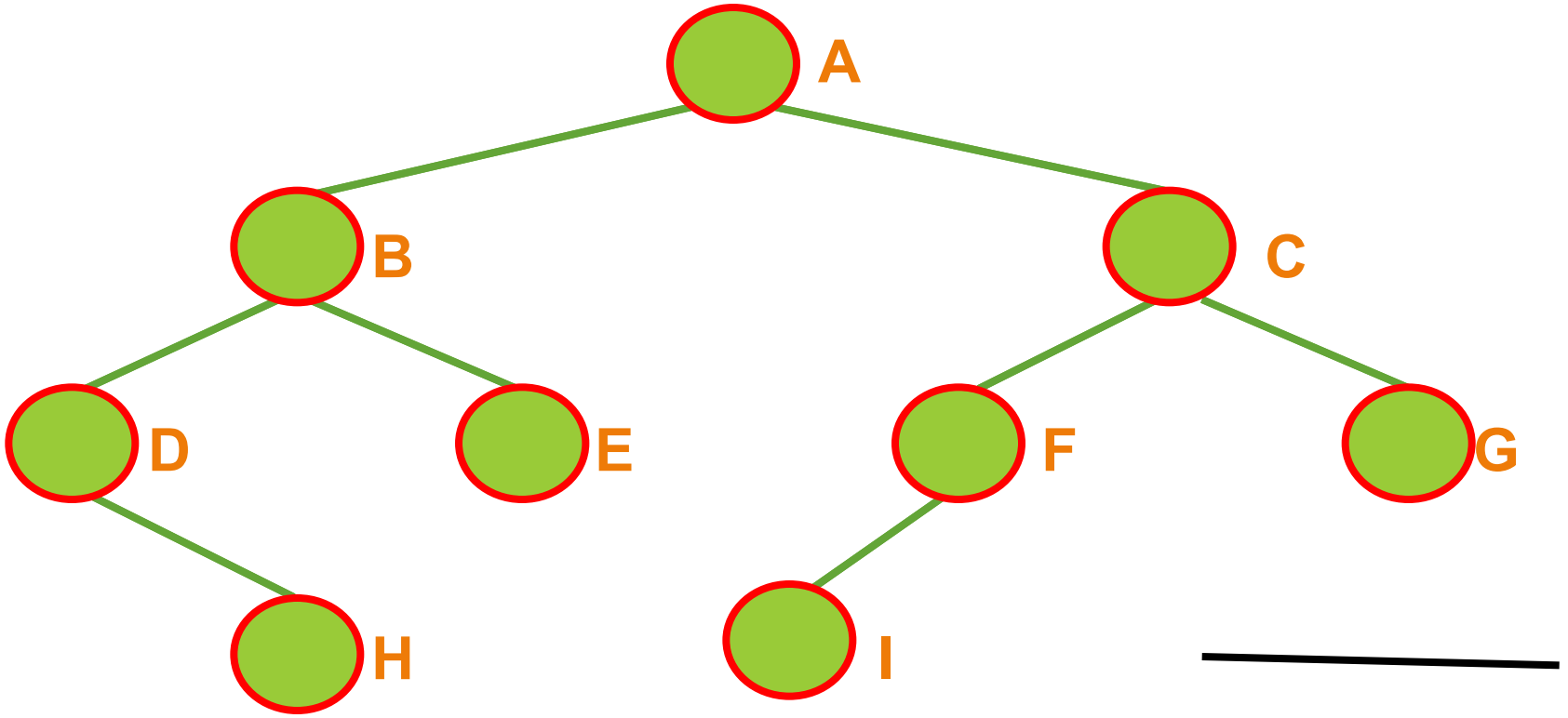
- Start at the root
- Visit the nodes at each level, from left to right



A B C D E F G H

Level Order Traversal

- Start at the root
- Visit the nodes at each level, from left to right



A B C D E F G H I

Level order Traversal

```
void levelOrderTraversal(struct Node* root) {
    if (root == NULL) return;
// Create a queue for level order traversal
    struct Queue* queue = createQueue(100);
    enqueue(queue, root);
    while (!isEmpty(queue)) {
        struct Node* current = dequeue(queue);
        printf("%d ", current->data);
        // Enqueue left child
        if (current->left != NULL)
            enqueue(queue, current->left);
      // Enqueue right child
        if (current->right != NULL)
            enqueue(queue, current->right);
    }}
```

Additional Binary Tree Operations

Searching

- Searching an item in the tree can be done while traversing the tree in inorder, preorder or postorder traversals.
- While visiting each node during traversal, instead of printing the node info, it is checked with the item to be searched.
- If item is found, search is successful.

Searching a binary tree

```
int Search(Nodeptr root,int ele) //uses preorder traversal
{
   static int t=0;
   if(root)
   {
      if(root->data==ele){
         t++;
         return t;
      }
      if (t==0) Search(root->lchild,ele);
      if (t==0) Search(root->rchild,ele);
   }
}
```

Creating a copy of a binary tree

Getting the exact copy of the given tree.

```
/*recursive function to copy a tree*/
Nodeptr copy (Nodeptr root){
    Nodeptr temp;
    if(root == NULL)
        return NULL;
    temp=getnode();
    temp→data=root→data;
    temp→lchild=copy(root→lchild);
    temp→rchild=copy(root→rchild);
    return temp;
}
```

Counting the number of nodes in a tree

Traverse the tree in any of the 3 techniques and every time a node is visited, count is incremented.

```
void count_nodes( Nodeptr root)
{
   static int count = 0;
  if(root!=NULL)
  {
        count_nodes(root→llink);
        count++;
        count_nodes(root→rlink);
}     return count;}
```

Counting the number of leaf nodes in a binary tree

Every time a node is visited, check whether the right and left link of that node is NULL. If yes, count is incremented.

```
/*counting number of leaf nodes using inorder technique*/

int count_leafnodes( Nodeptr root){
   static int count = 0;
   if(root!=NULL){
         if(root->lchild==NULL && root->rchild==NULL)
                        count++;
         count_leafnodes(root->lchild);
         count_leafnodes(root->rchild);
   }     return count; }
```

Equality of 2 binary trees

Returns FALSE if the binary trees root1 and root2 are not equal otherwise returns TRUE

```
int Equal( Nodeptr root1, Nodeptr root2)
{
return ((!root1 && !root2) || (root1 && root2 && (root1→data
   == root2→data) && Equal( root1->lchild,root2->lchild) &&
   Equal ( root1->rchild,root2->rchild));
}
```

Expression Tree

An expression tree is a binary tree used to represent expressions

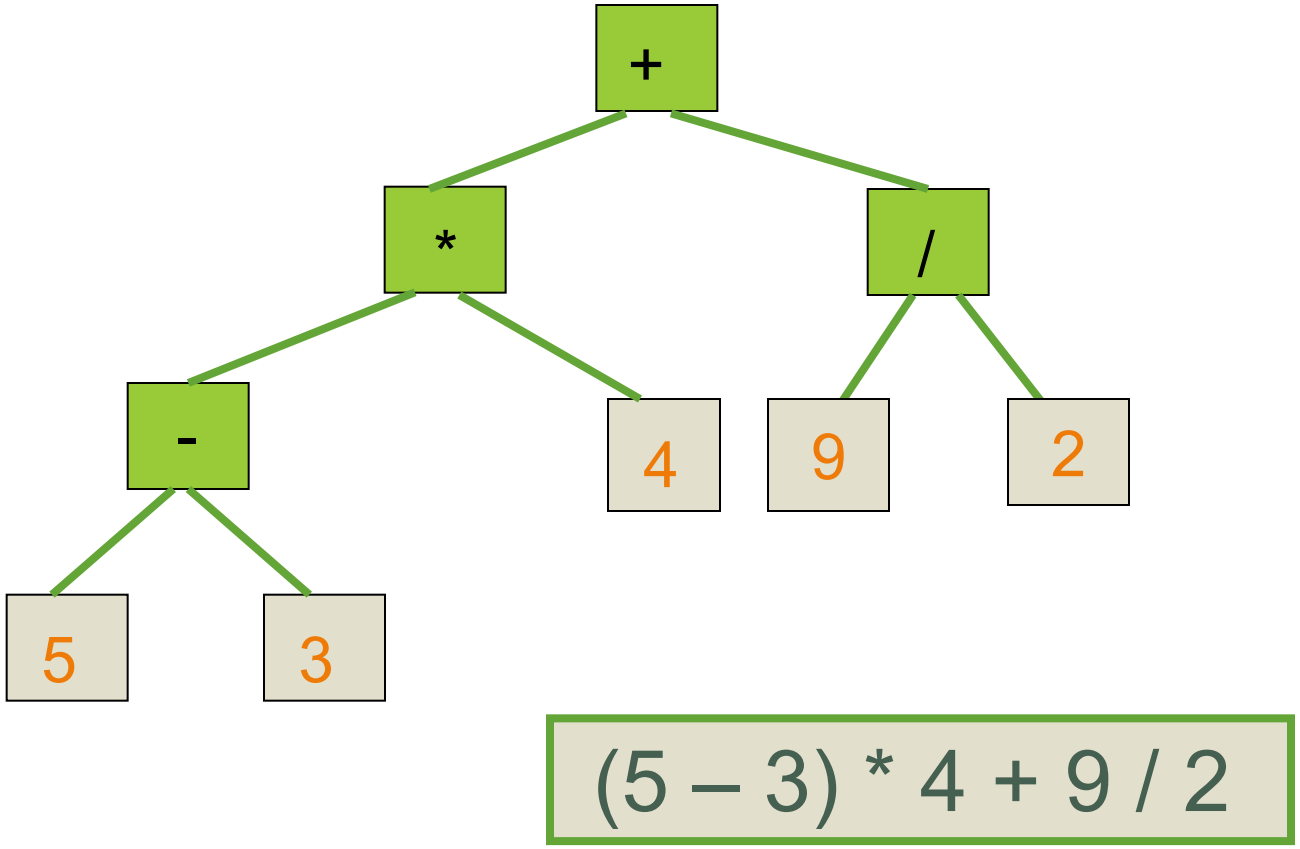
Using Binary Trees: Expression Trees

- Programs that manipulate or evaluate arithmetic expressions can use binary trees to hold the expressions

- An ***expression tree*** represents an arithmetic expression such as (5 – 3) * 4 + 9 / 2

  - Root node and interior nodes contain ***operations (***operator***)***
  - Leaf nodes contain ***operands (***variable or a constant***)***

Example: An Expression Tree

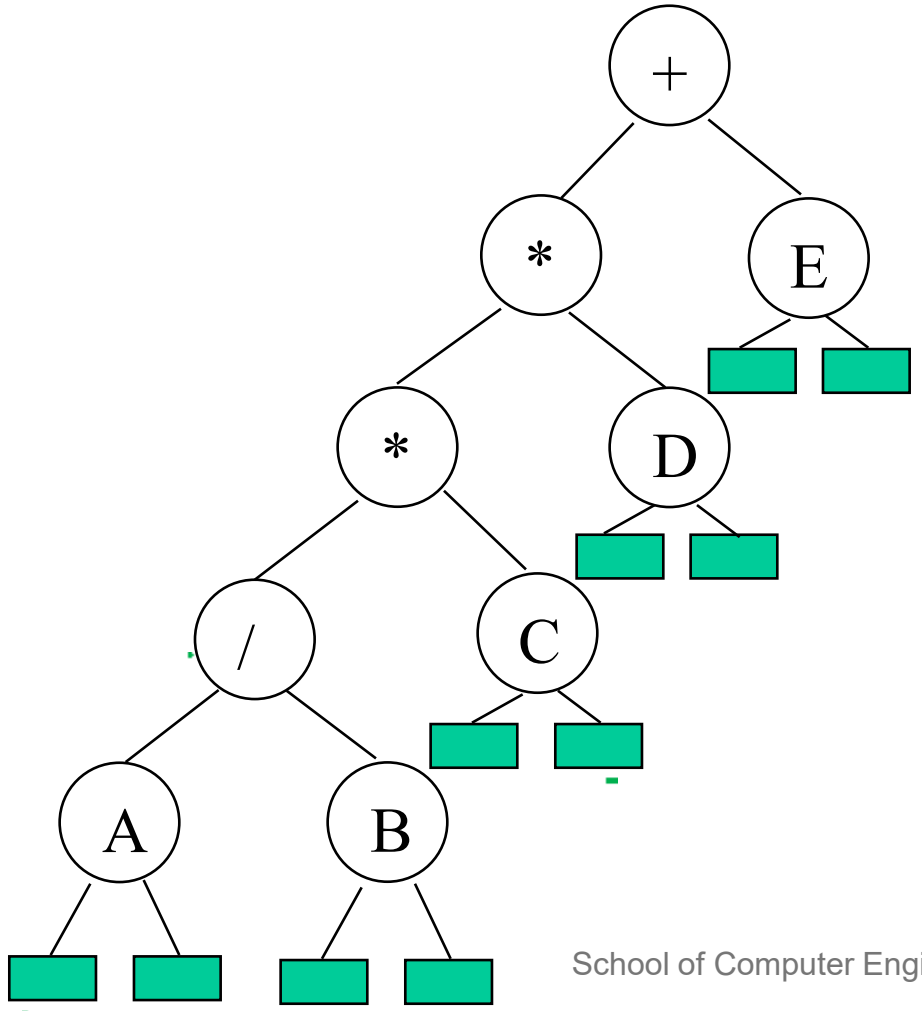


Why use Expression Trees?

- Evaluation: You can easily evaluate the expression by traversing the tree.
- Transformation: It helps transform infix, postfix, or prefix expressions.

Expression Tree Traversals

- **Inorder Traversal**: Yields the infix expression.
- **Preorder Traversal**: Yields the prefix expression.
- **Postorder Traversal**: Yields the postfix expression.

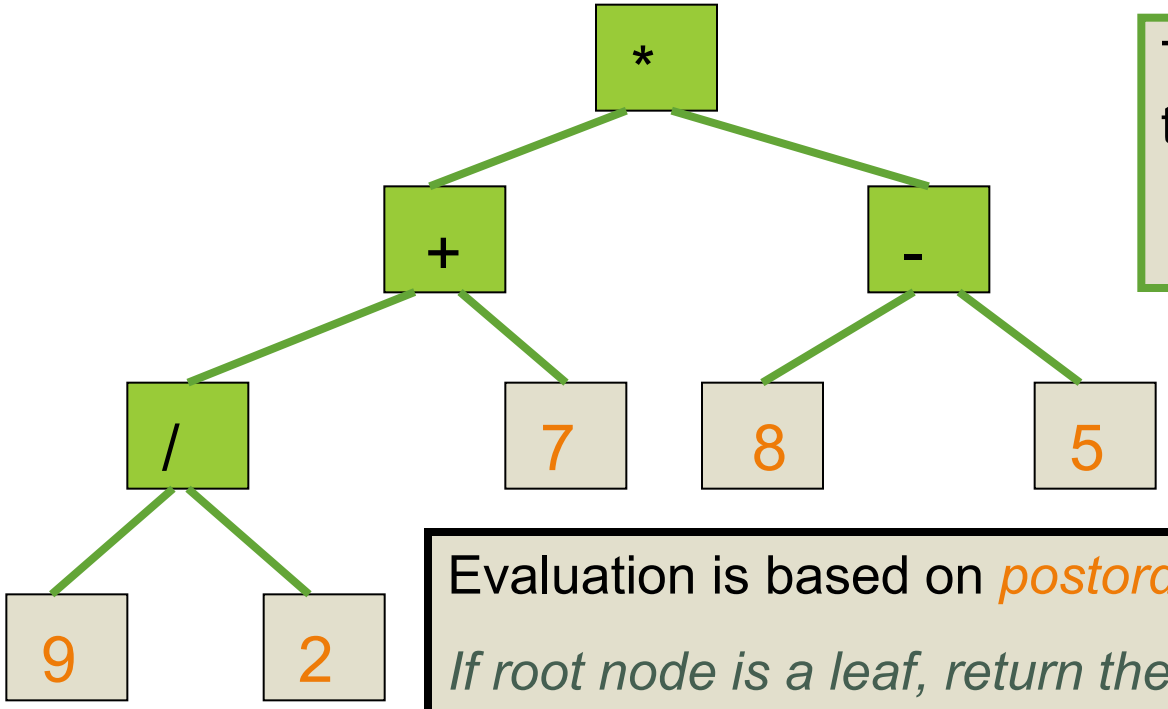


inorder traversal
A / B * C * D + E
infix expression
preorder traversal
+ * * / A B C D E
prefix expression
postorder traversal
A B / C * D * E +
postfix expression
level order traversal
+ * E * D / C A B

Evaluating Expression Trees

- We can use an expression tree to ***evaluate an expression***
  - We start the evaluation at the *bottom left*
  - What kind of traversal is this?

Evaluating an Expression Tree



This tree represents the expression

(9 / 2 + 7) * (8 – 5)

Evaluation is based on *postorder* traversal:

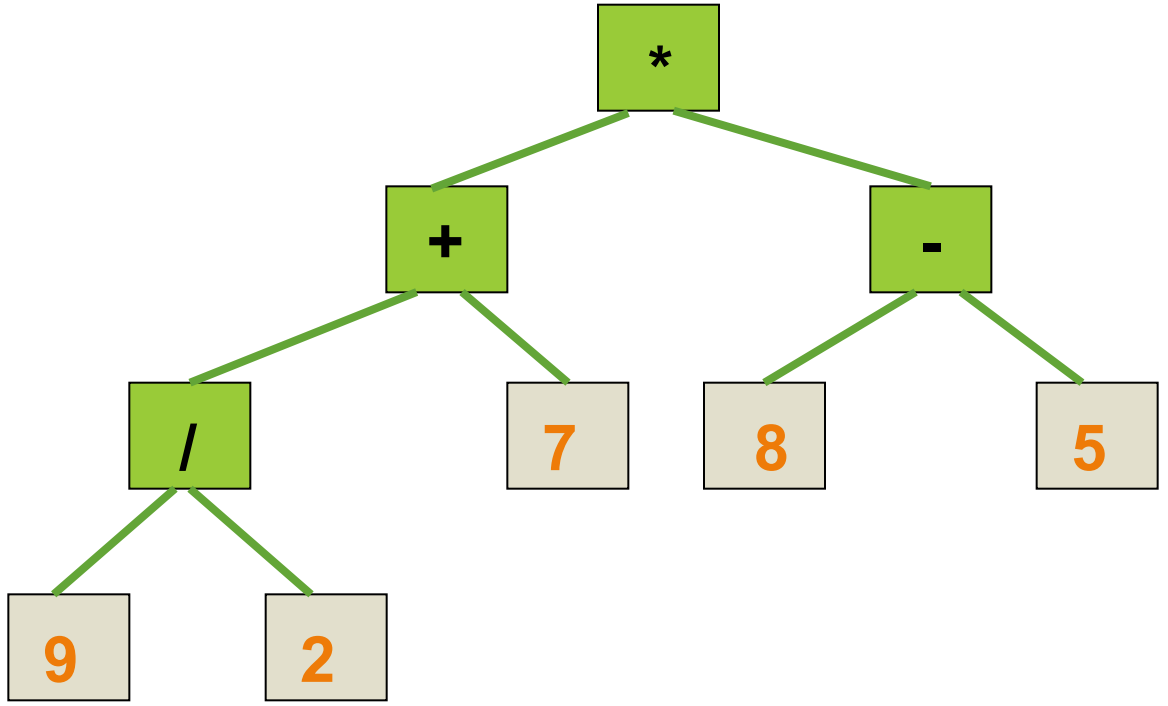
If root node is a leaf, return the associated value.

Recursively evaluate expression in left subtree.

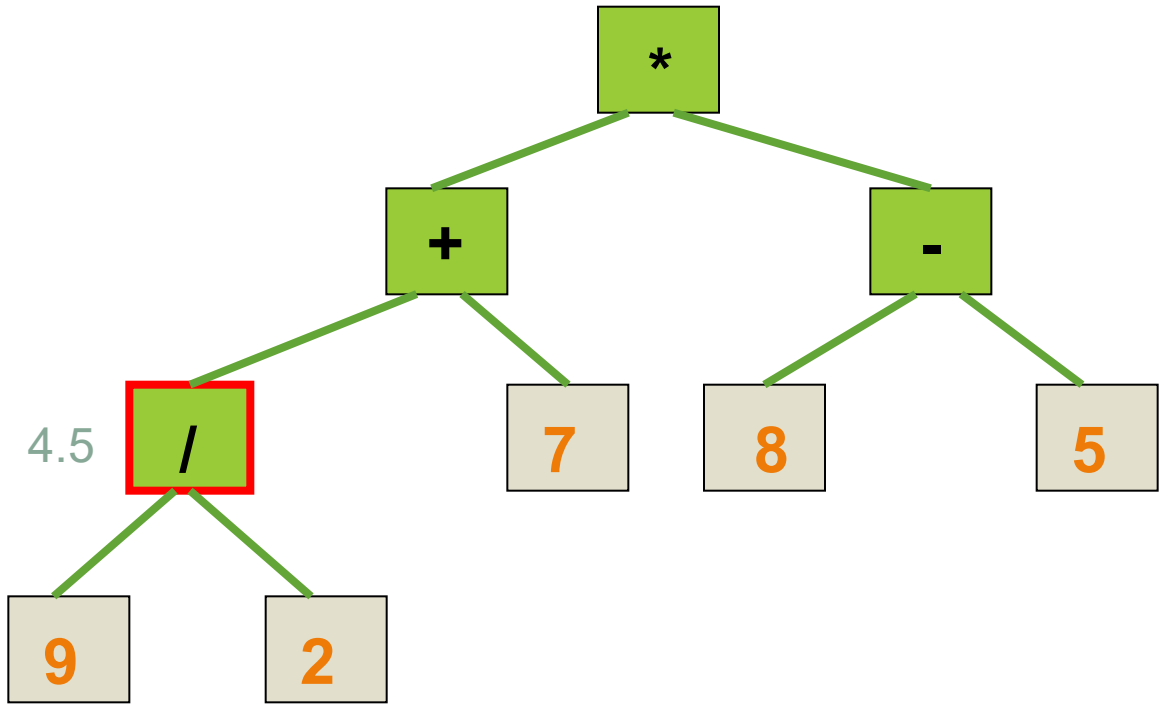
Recursively evaluate expression in right subtree.

Perform operation in root node on these two values, and return result.

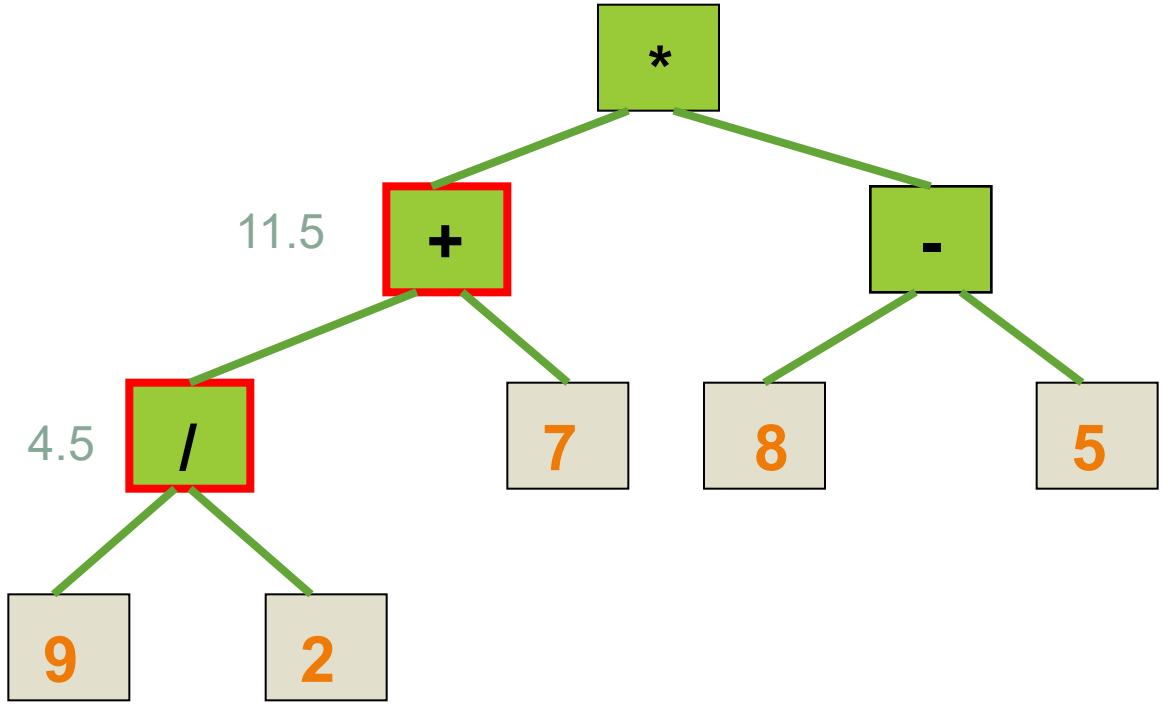
Evaluating an Expression Tree



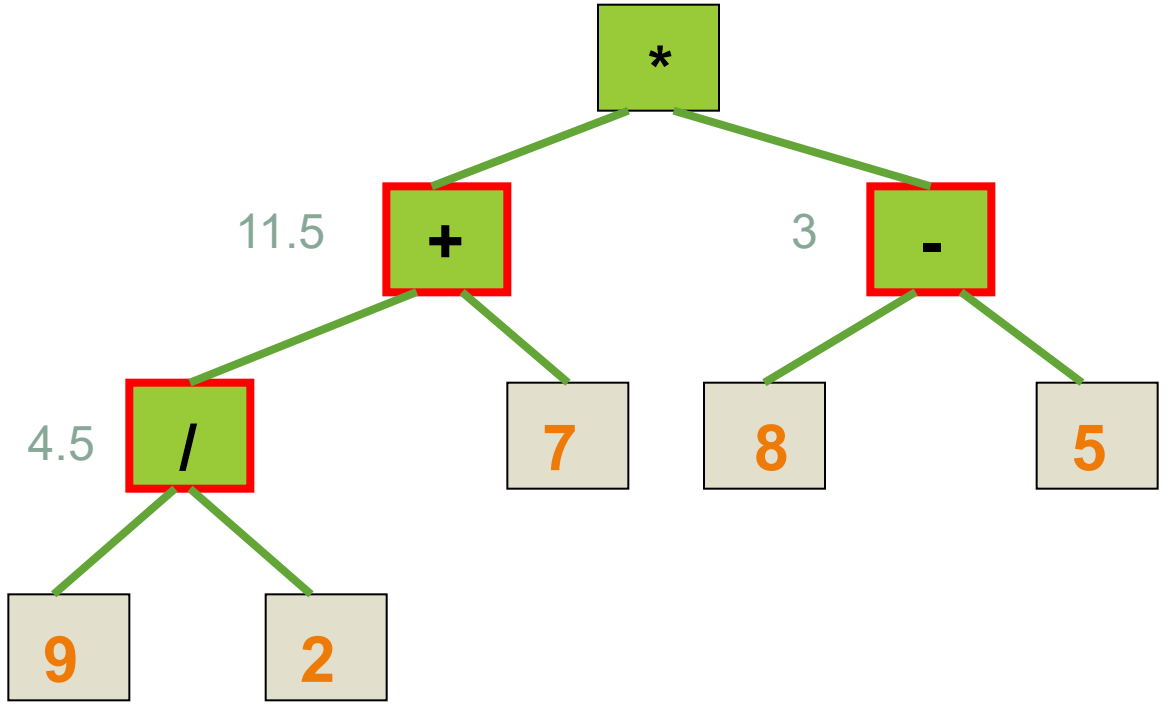
Evaluating an Expression Tree



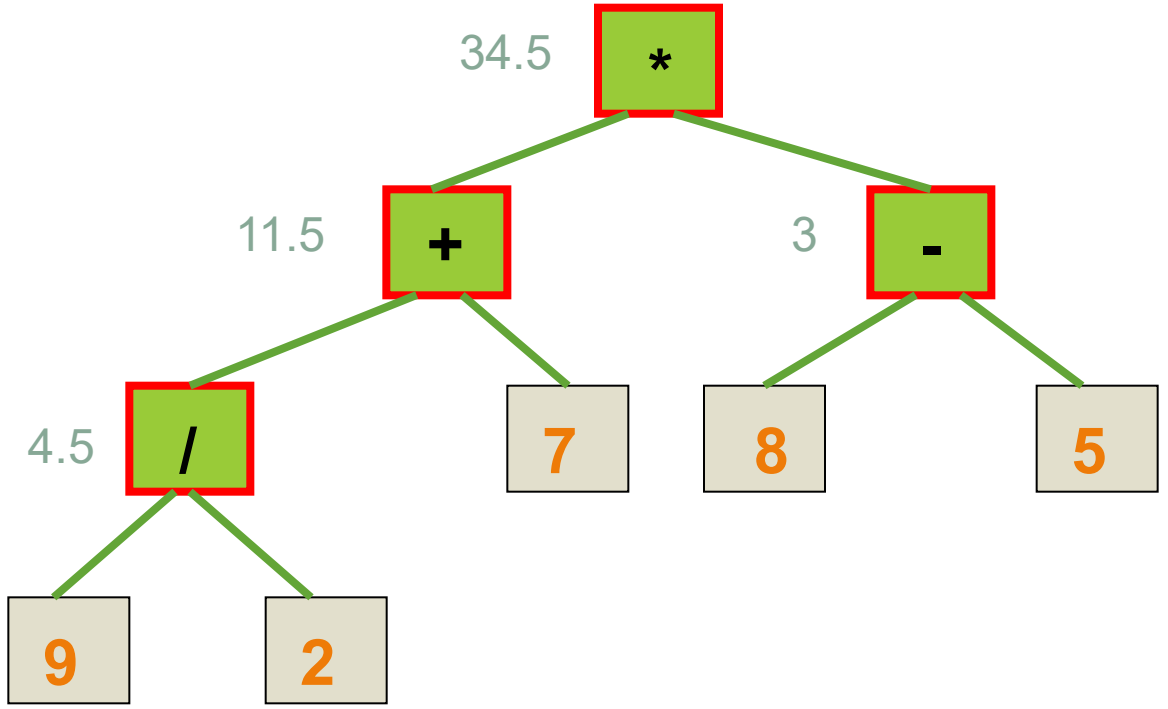
Evaluating an Expression Tree



Evaluating an Expression Tree



Evaluating an Expression Tree



Building an Expression Tree

- **From Postfix Expression:**

1. Initialize an empty stack.
2. Traverse the postfix expression from left to right.
3. For each symbol:
    1. If it’s an operand, create a tree node and push it onto the stack.
    2. If it’s an operator, pop two nodes from the stack, make them children of a new operator node, and push the new node back onto the stack.
4. The final node in the stack is the root of the expression tree.

Build an expression tree from the postfix expression `5 3 - 4 * 9 +`

Symbol

Processing Step(s)

```
t = newNode (5,null, null); push(stack, t);
```

Expression Tree Stack (top at right)

Symbol

Processing Step(s)

```
t = newNode (3,null, null); push(stack, t);
```

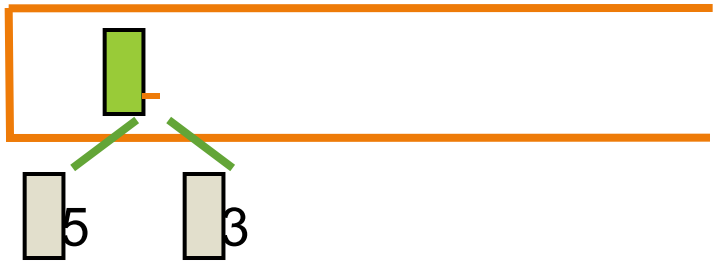
Expression Tree Stack (top at right)

Symbol

Processing Step(s)

```
op2 = pop
op1 = pop
t = newNode (-,op1, op2);
push(stack, t);
```

Expression Tree Stack

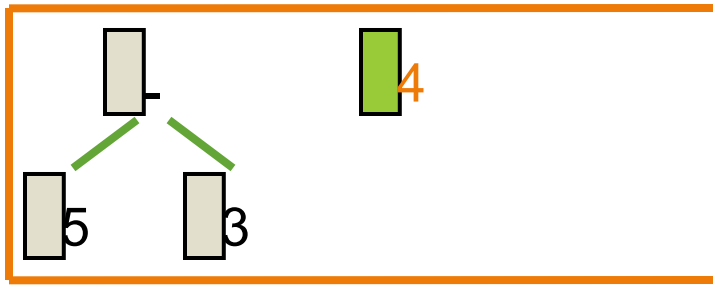


Symbol

Processing Step(s)

```
t = newNode (4,null, null);
push(stack, t);
```

Expression Tree Stack (top at right)

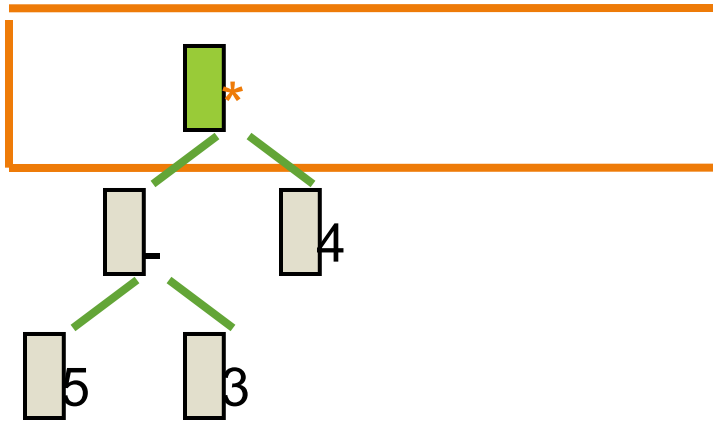


Symbol

Processing Step(s)

```
op2 = pop
op1 = pop
t = newNode (*,op1, op2);
push(stack, t);
```

Expression Tree Stack (top at right)



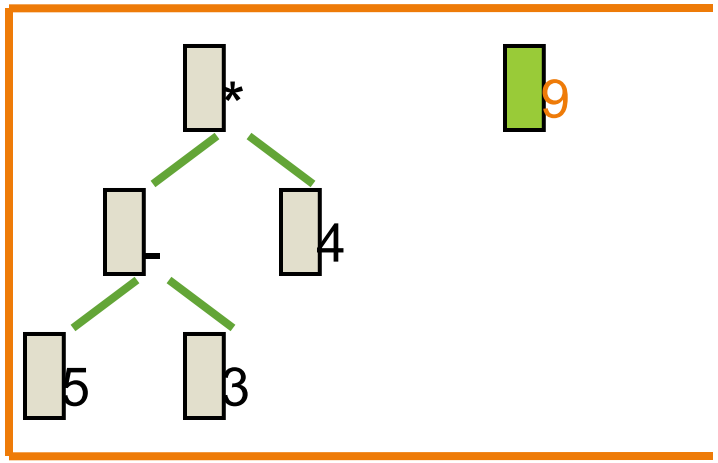
Symbol

9

Processing Step(s)

```
t = newNode (9,null, null);
push(stack, t);
```

Expression Tree Stack (top at right)

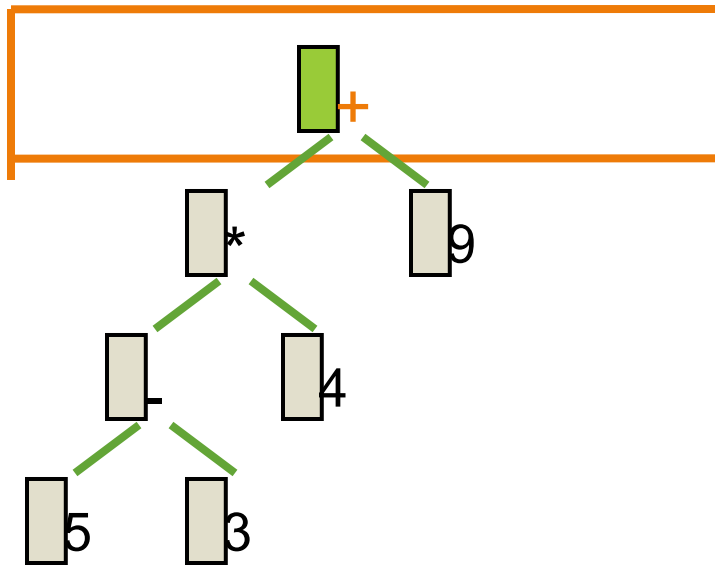


Symbol

Processing Step(s)

```
op2 = pop
op1 = pop
t = newNode (+,op1, op2);  push(stack, t);
```

Expression Tree Stack (top at right)

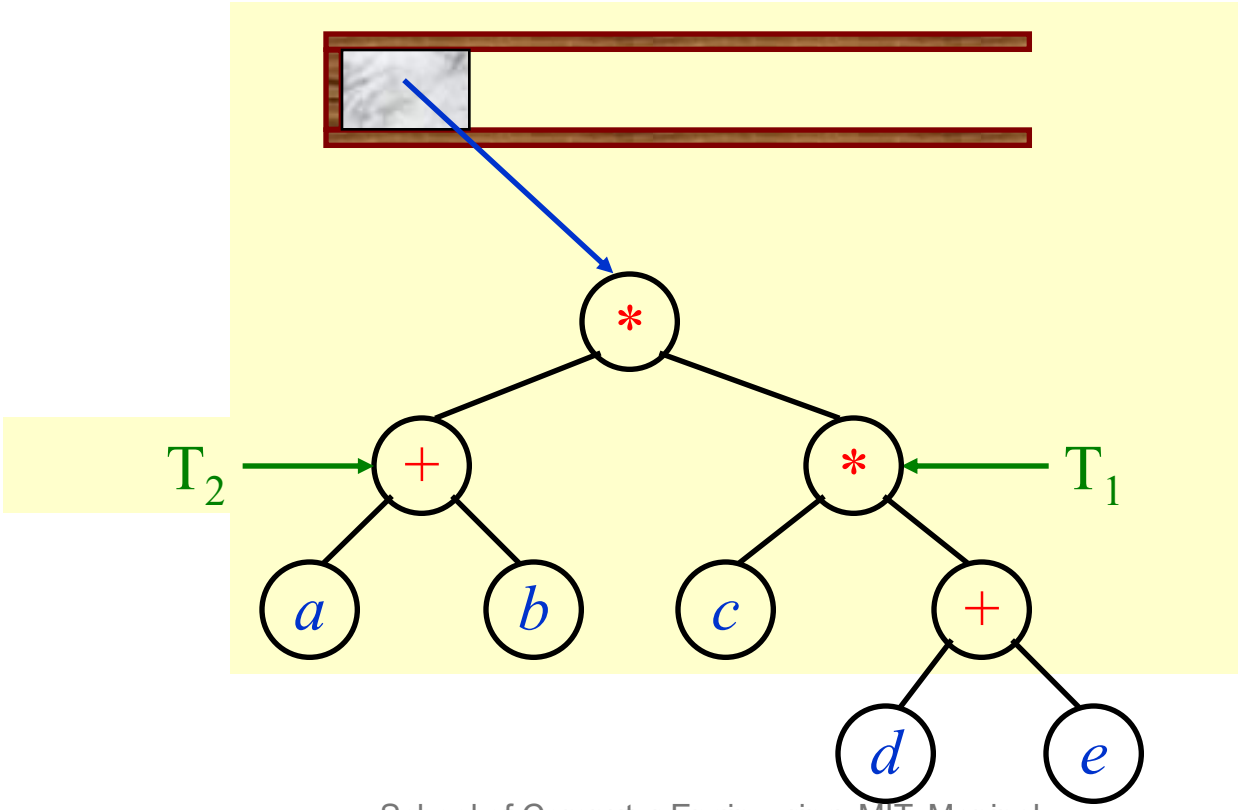


End of the expression has been reached, and the full expression tree is the only tree left on the stack

❖ Expression Trees

☞ Constructing an Expression Tree
(from postfix expression)

〖Example〗 $(a + b) * (c * (d + e)) =$ $a\ b + c\ d\ e + * *$



Threaded Binary Tree

Threads

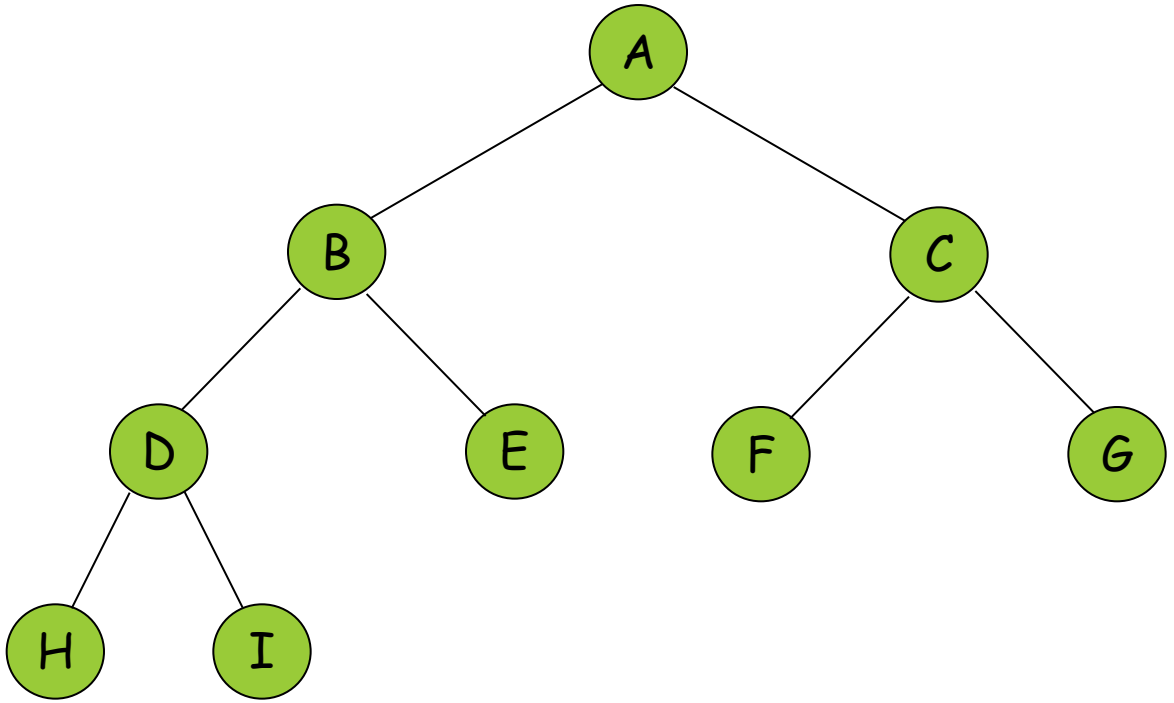
- In a linked representation of a binary tree, there are more NULL links than actual pointers.
- In a binary tree with n nodes containing 2n links, there are n+1 NULL links.
- Perlis and Thornton devised a way to make use of NULL links.
- Here the NULL links are replaced by pointers, called *threads*, to other nodes in the tree.

Threaded Binary Tree

Threading Rules

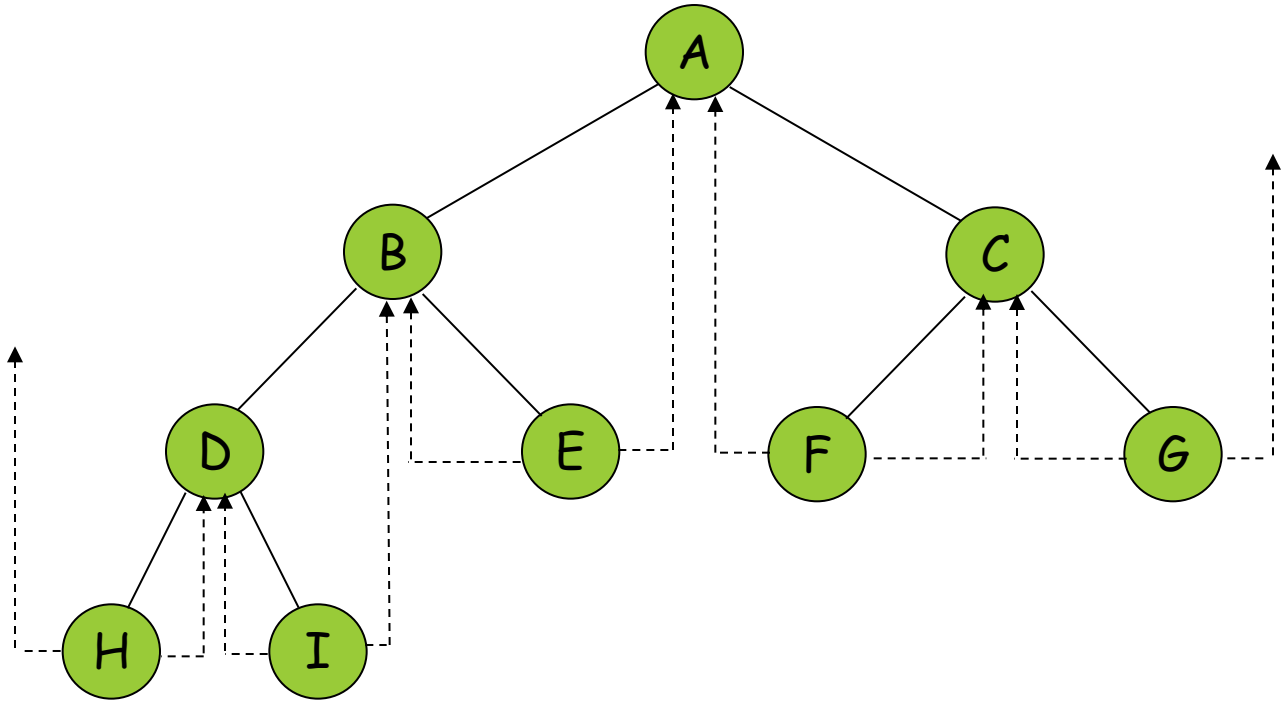
- A NULL RightChild field at node p is replaced by a pointer to the node that would be visited after p when traversing the tree in inorder. That is, it is replaced by the inorder successor of p.
- A NULL LeftChild link at node p is replaced by a pointer to the node that immediately precedes node p in inorder (i.e., it is replaced by the inorder predecessor of p).

A Binary Tree



Inorder sequence: H, D, I, B, E, A, F, C, G

Threaded Tree Corresponding to Given Binary Tree



Inorder sequence: H, D, I, B, E, A, F, C, G

Threads

- To distinguish between normal pointers and threads, two boolean fields, LeftThread and RightThread, are added to the record in memory representation.
  - t->leftThread= TRUE

    => t->lchild is a **thread**
  - t->leftThread= FALSE

    ⇒t->lchild is a **pointer** to the left child.
  - t->rightThread= TRUE

    => t->rchild is a **thread**
  - t->rightThread= FALSE

    => t->rchild is a **pointer** to the right child.

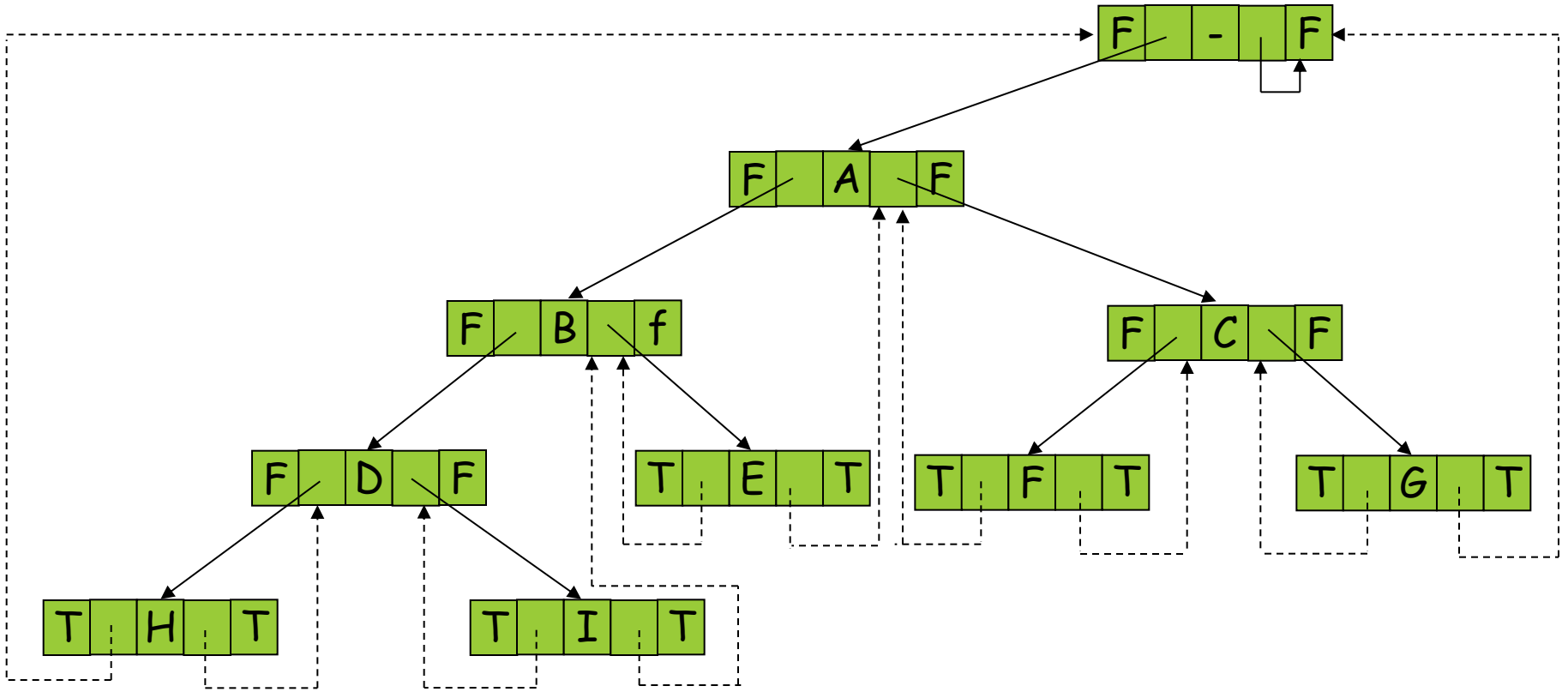
Threaded Binary Tree Node Structure Declaration

```
typedef struct threadedTree *threadedPointer;

struct threadedTree{
        short int leftThread;
        threadedPointer lchild;
        char data;
        threadedPointer rchild;
        short int rightThread;
};
```

Threads (Cont.)

- To avoid dangling threads, a head node is used in representing a binary tree.
- The original tree becomes the left subtree of the head node.
- Empty Binary Tree

Memory Representation of Threaded Tree



Finding the inorder successor without stack

By using the threads, we can perform an inorder traversal without making use of a stack.

```
threadedPointer insucc(threadedPointer node)
{       //Return the inorder successor of node
   threadedPointer temp = node-> rchild;
  if (node->rightThread==FALSE)
          while (temp->leftThread==FALSE)
                  temp = temp -> lchild;
  return temp;
  }
```

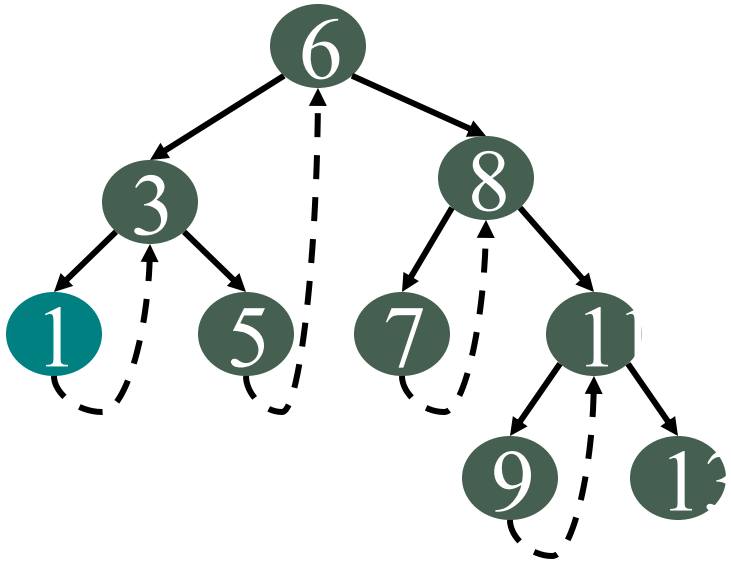
Inorder Traversal of a threaded Binary Tree

```
void tinorder(threadedPointer treehead)
{
  threadedPointer temp = treehead;
  while(1){
    temp = insucc(temp);
    if (temp == treehead) break;
    printf("%c", temp->data);
  }
}
```

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

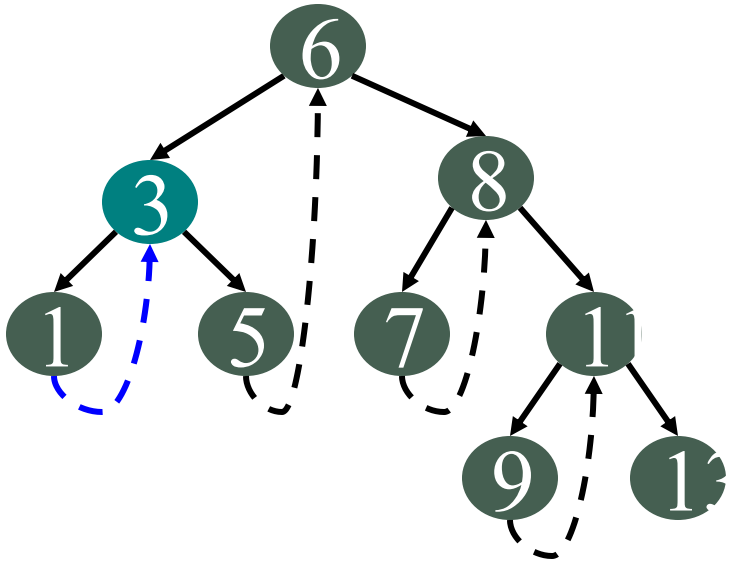
Threaded Tree Traversal



Output
1

Start at leftmost node, print it

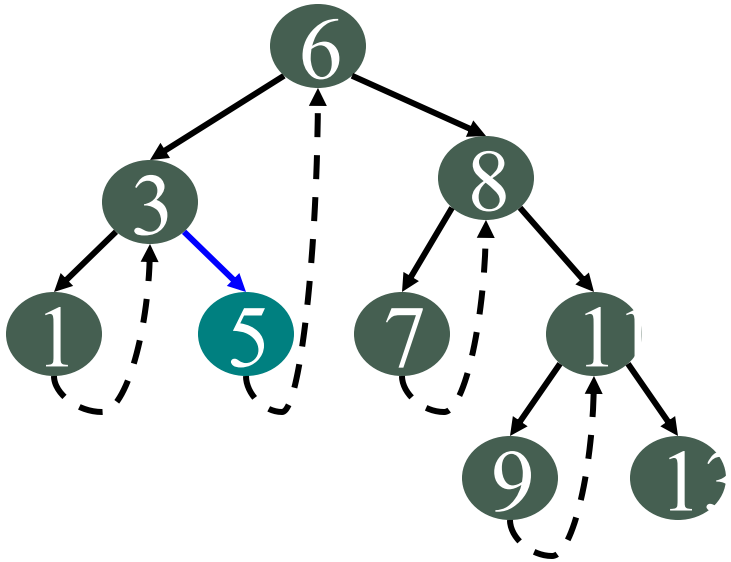
Threaded Tree Traversal



Output
1
3

Follow thread to right, print node

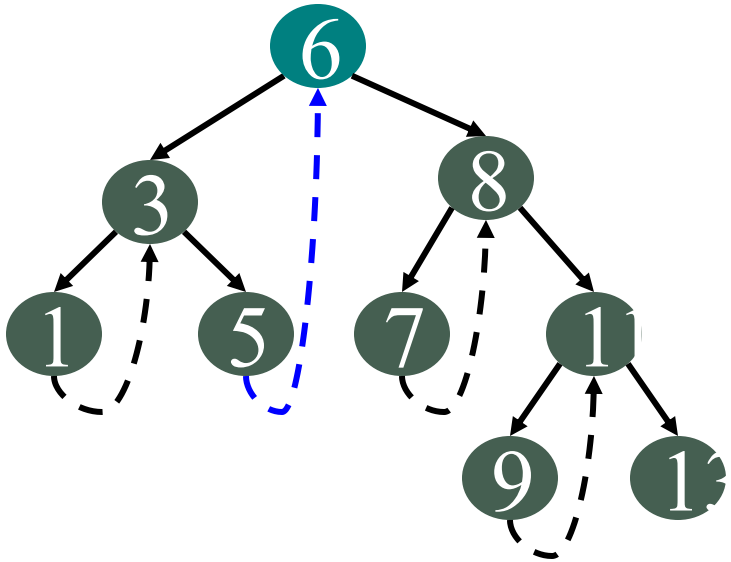
Threaded Tree Traversal



Output
1
3
5

Follow link to right, go to leftmost node and print

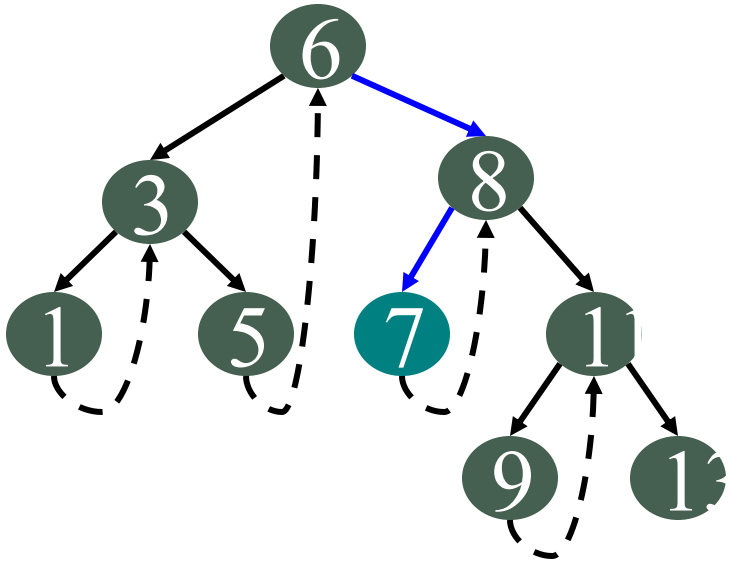
Threaded Tree Traversal



Output
1
3
5
6

Follow thread to right, print node

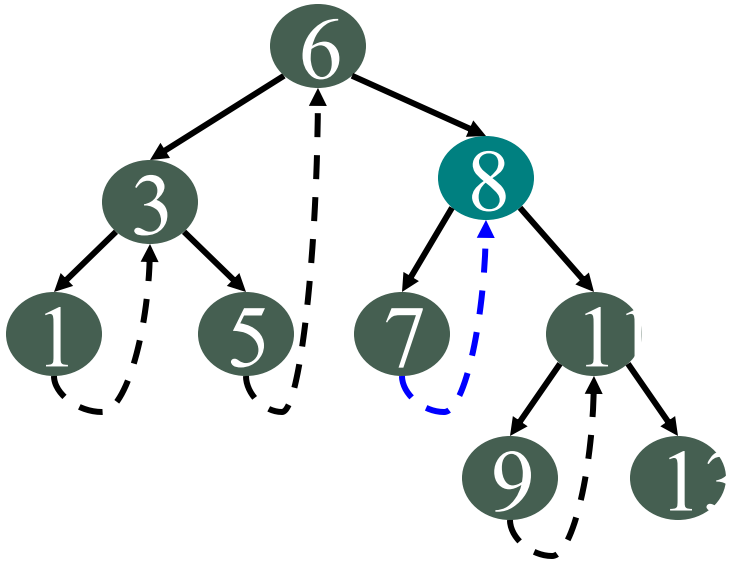
Threaded Tree Traversal



Output
1
3
5
6
7

Follow link to right, go to leftmost node and print

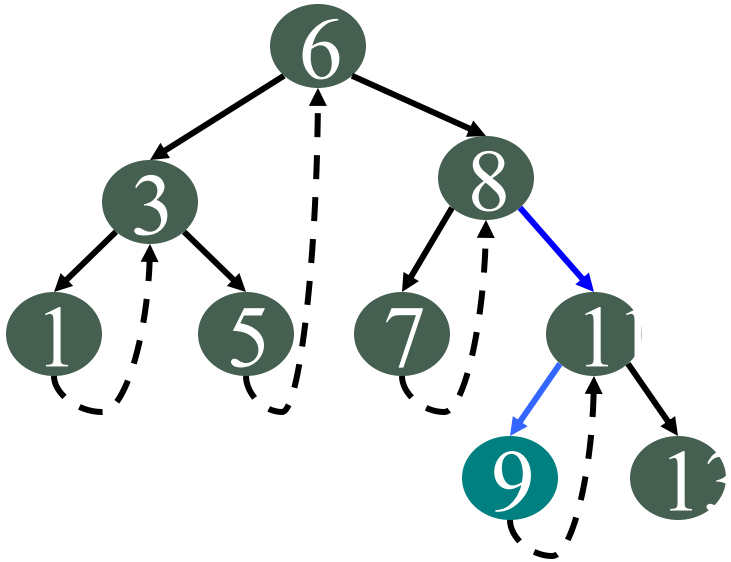
Threaded Tree Traversal



Output
1
3
5
6
7
8

Follow thread to right, print node

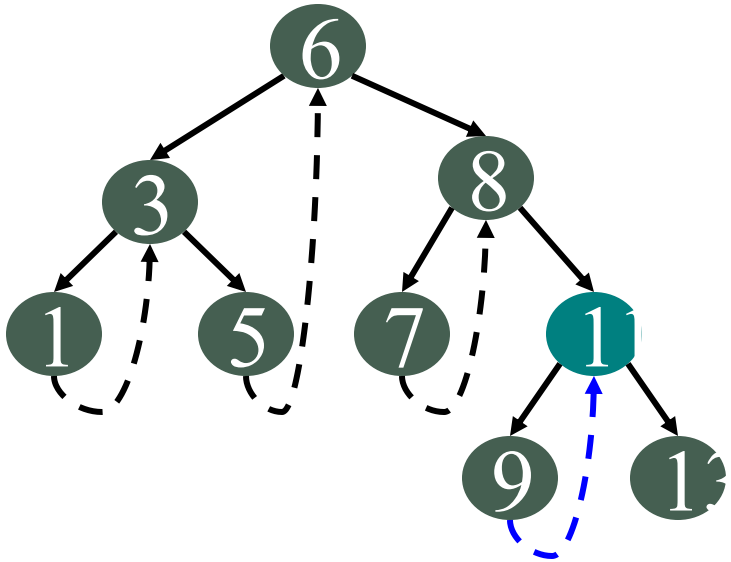
Threaded Tree Traversal



Output
1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

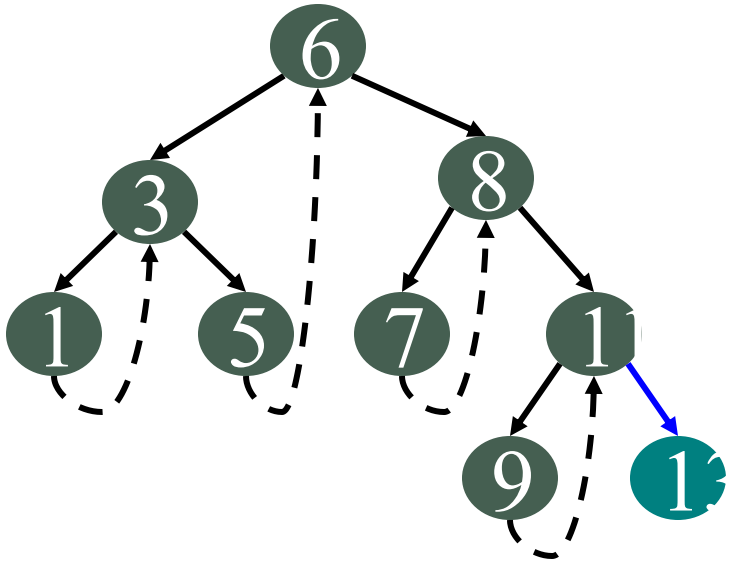
Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11

Follow thread to right, print node

Threaded Tree Traversal



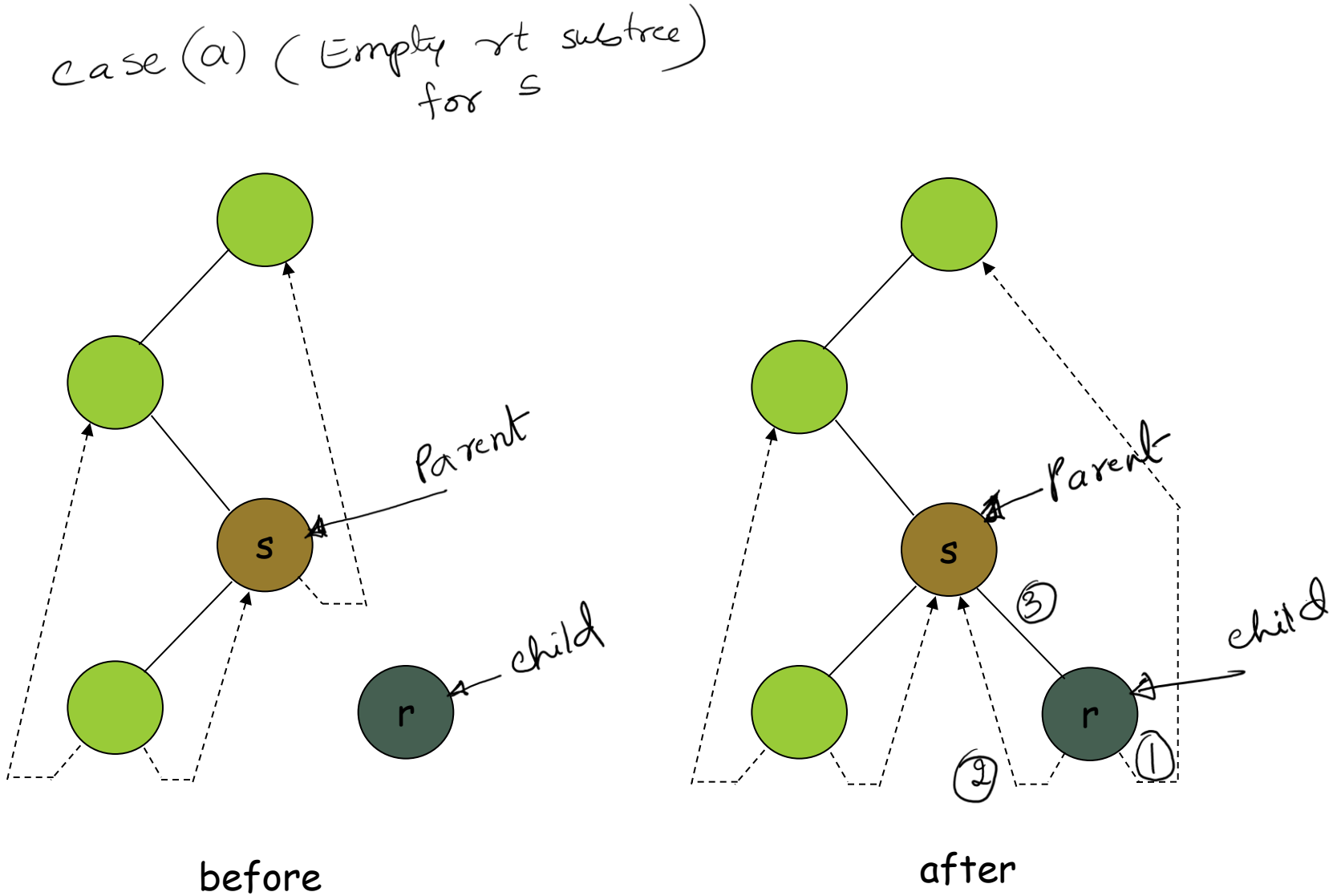
Output
1
3
5
6
7
8
9
11
13

Follow link to right, go to leftmost node and print

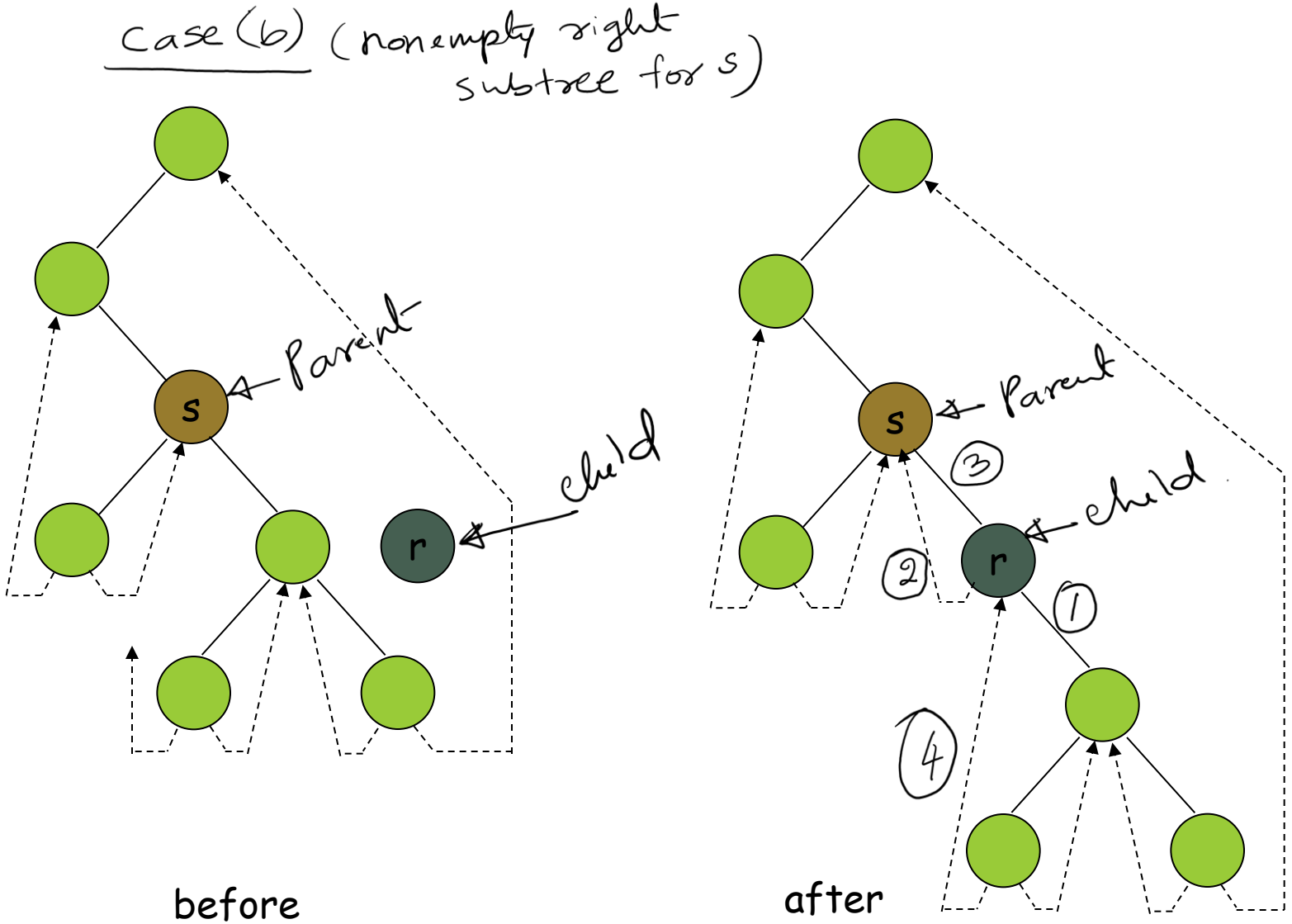
Inserting A Node to AThreaded Binary Tree

- Inserting a node r as the right child of a node s.
  - If s has an empty right subtree, then the insertion is simple (as shown in diagram next slide)
  - If the right subtree of s is not empty, then, this right subtree is made the right subtree of r after insertion. When this is done, r becomes the inorder predecessor of a node that has a leftThread==TRUE field, and consequently there is an thread which has to be updated to point to r. The node containing this thread was previously the inorder successor of s. Figure illustrates the insertion for this case.

Insertion of r As A Right Child of s in A Threaded Binary Tree



Insertion of r As A Right Child of s in A Threaded Binary Tree (Cont.)



Advantages

- **Efficient Traversal:** Threaded binary trees allow for linear and fast traversal of nodes in the tree without the need for a stack. This eliminates the memory and time overhead associated with stack-based traversal methods.

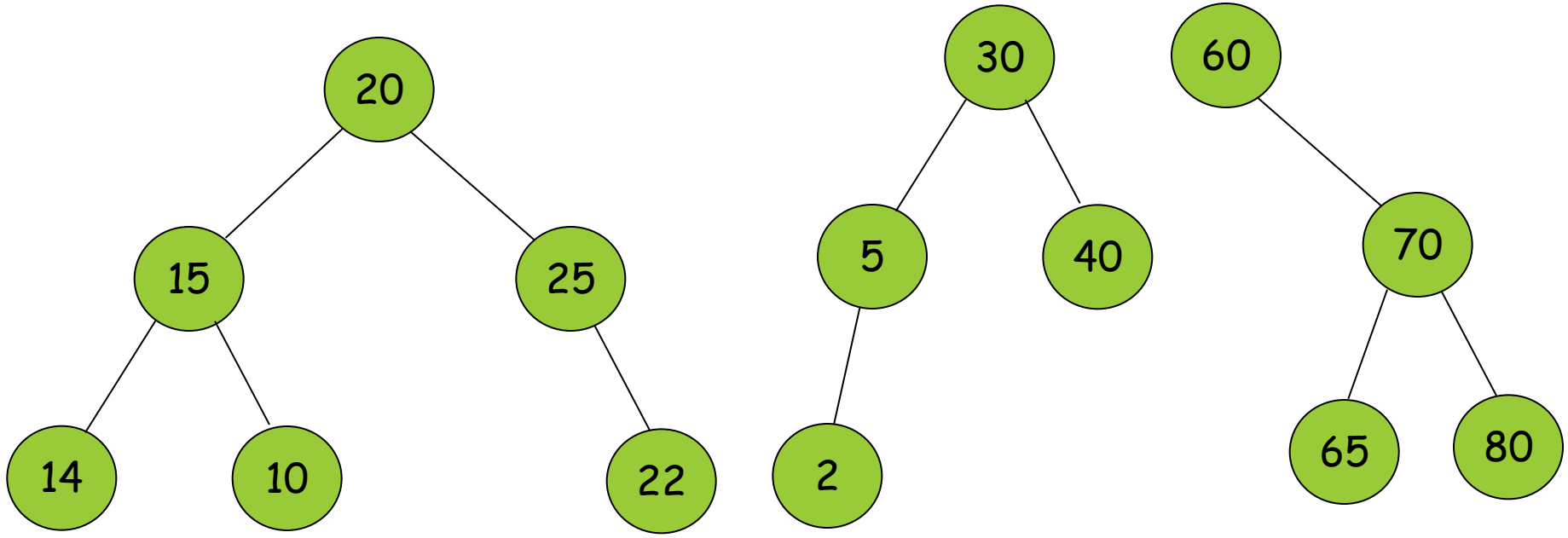
- **Reduced Memory Consumption:** Since threaded binary trees do not require a stack for traversal, they consume less memory compared to other traversal methods that rely on recursion or iterative approaches with stacks.

- **Quick Successor and Predecessor Determination:** In threaded binary trees, it is easy to determine the successor and predecessor of any node by following the thread and links. This allows for efficient retrieval and manipulation of neighboring nodes, similar to a circular linked list.

Binary Search Tree (BST)

- Definition: A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:
  - Every node has exactly one key and no two nodes have the same key (i.e., the keys in the tree are distinct)
  - The keys (if any) in the left subtree are smaller than the key in the root.
  - The keys (if any) in the right subtree are larger than the key in the root.
  - The left and right subtrees are also binary search trees.
- Inorder traversal of BST results in ascending order of elements

Binary Tree vs BST



Not binary search tree

Binary search trees

Difference between BT and BST

- ♠ A binary tree is simply a tree in which each node can have at most two children.
- ♠ A binary search tree is a binary tree in which the nodes are assigned values, with the following restrictions :

1. No duplicate values.
2. The left subtree of a node can only have values less than the node and recursively defined
3. The right subtree of a node can only have values greater than the node and recursively defined
4. The left subtree of a node is a binary search tree.
5. The right subtree of a node is a binary search tree.

Recursive function to create a BST

```
Nodeptr CreateBST(Nodeptr root, int item){
   if (root==NULL){
      root = getnode();
      root->data= item;
      root->lchild=root->rchild = NULL;
      return root;
   }
   else
   if (item<root->data)
      root->lchild = CreateBST(root->lchild, item);
   else
      if (item>root->data)
         root->rchild = CreateBST(root->rchild, item);
    else
         printf("Duplicates are not allowed\n");
   return root;
}
```

Searching in A Binary Search Tree

- If the root is NULL, then this is an empty tree. No search is needed.
- If the root is not NULL, compare the x with the key of root.
  - If x equals to the key of the root, then it's done.
  - If x is less than the key of the root, then no elements in the right subtree can have key value x. We only need to search the left tree.
  - If x larger than the key of the root, only the right subtree is to be searched.

Searching in A Binary Search Tree

if the tree is empty
    return NULL

else if the key value in the node(root) equals the target
    return the node value

else if the key value in the node is greater than the target
    return the result of searching the left subtree

else if the key value in the node is smaller than the target
    return the result of searching the right subtree

Searching a BST

```
typedef struct node *Nodeptr;
struct node{
    int data;
    Nodeptr rchild;
    Nodeptr lchild;
};
Nodeptr search(Nodeptr root,int key)
{
/* return a pointer to the node that contains key. If there is no such node, return NULL */

  if (root==NULL) return NULL;
  if (key == root->data) return root;
  if (key < root->data)
      return search(root->lchild, key);
  return search(root->rchild,key);
}
```

Iterative Searching Algorithm

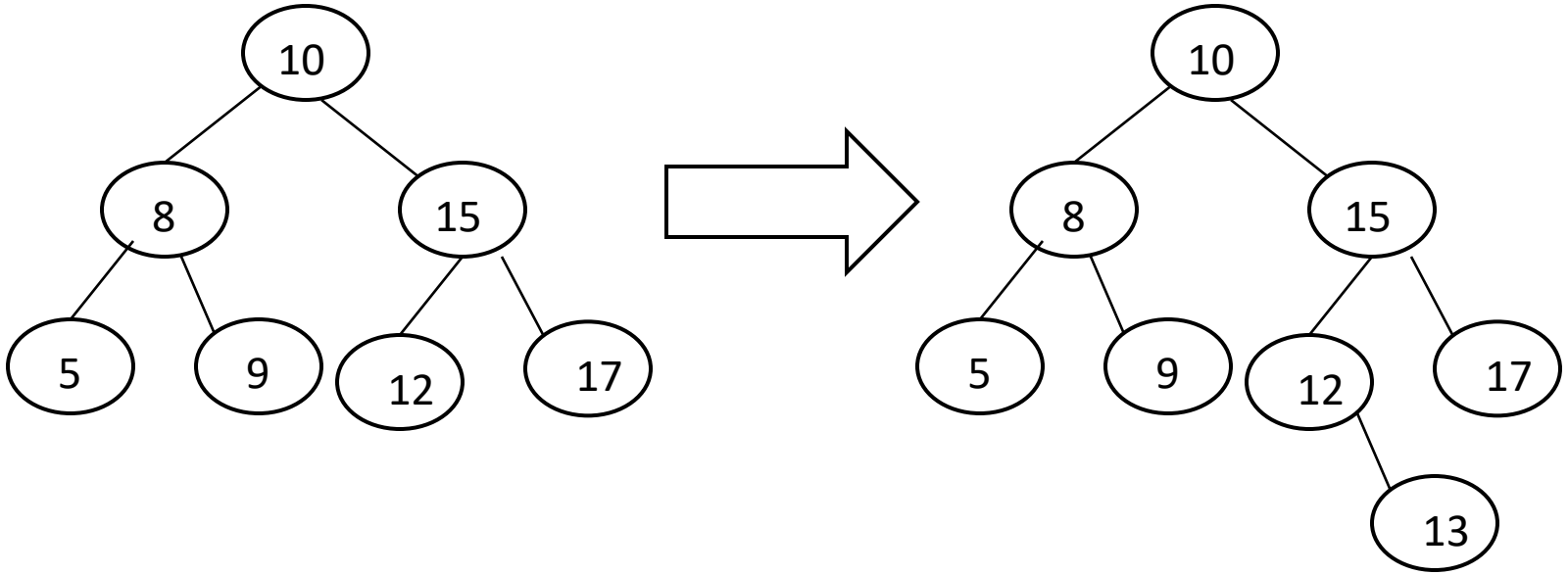
```
Nodeptr itersearch(Nodeptr root, int key)
{
  while (root) {
    if (key == root->data) return root;
    if (key < root->data)
        root = root->lchild;
    else root = root->rchild;
  }
  return NULL;
}
```

Other operations:

1. Finding the maximum element in BST: maximum element will always be the last right child in a BST. Move to the rightmost node in a BST and you will end up in the maximum element.
2. Finding the minimum element: Move to the leftmost child and you will reach the least element in BST.
3. Finding the height of a tree: height is nothing but maximum level in the tree plus one. It can be easily found using recursion.

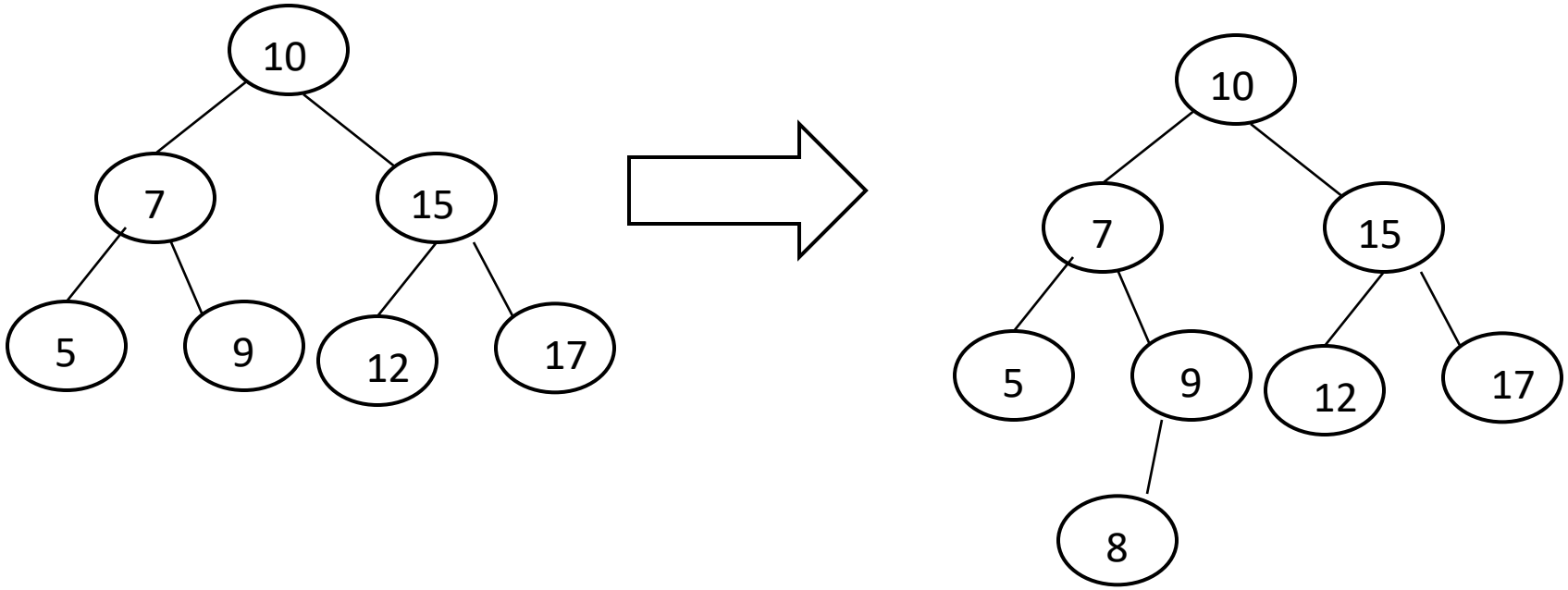
Insertion To A Binary Search Tree

- Before insertion is performed, a search must be done to make sure that the value to be inserted is not already in the tree.
- If the search fails, then we know the value is not in the tree. So, it can be inserted into the tree.
- If item is lesser than the root item, move to left or else move to the right of root node.
- This process is repeated until the correct position is found.



To insert item 13 to above BST

- Compare with the root item. 13> 10, hence move to right and reach 15.
- Now 13<15, So go to left and reach 12.
- 13>12, hence move right.
- Now the correct position is found and hence insert the new node to the right of 12.



To insert 8 into the above tree

- Compare with root item. 8<10, hence move left and reach 7.
- Now 8>7. So move right and reach 9.
- 8<9. Move left and the correct position is obtained.
- Insert 8 to the left of 9.

Insertion Into A Binary Search Tree

```
typedef struct node *Nodeptr;
struct node{
    int data;
    Nodeptr rchild;
    Nodeptr lchild;
};
void Insert(Nodeptr *root, int item){
    Nodeptr temp= getnode();
    temp->data = item;
    temp->lchild = NULL;
    temp->rchild = NULL;
    if (*root==NULL) {
                *root = temp; return;
    }
    Nodeptr parent, cur;
```

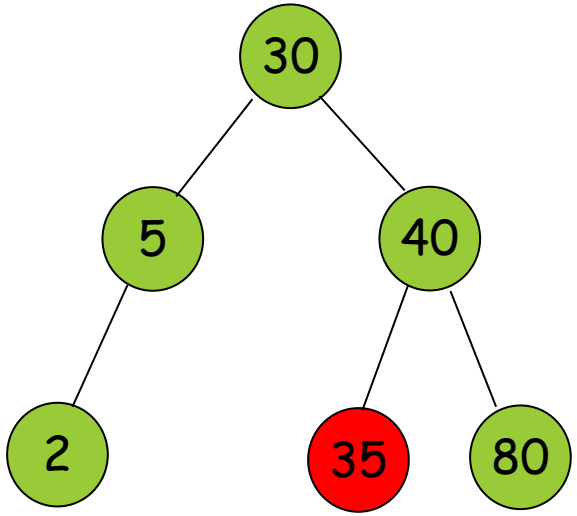
```
/*traverse until correct position is found*/

 parent=NULL;
 cur=*root;
 while(cur){
        parent=cur;
        if (item==cur->data ){
            printf("Duplicates Not allowed");
            free(temp);
            return;
        }
        else if (item<cur->data)
                cur=cur->lchild;
             else
                cur=cur->rchild;
    }
    if (item<parent->data)
        parent->lchild = temp;
    else
        parent->rchild = temp;
    return;
}
```

Deletion From A Binary Search Tree

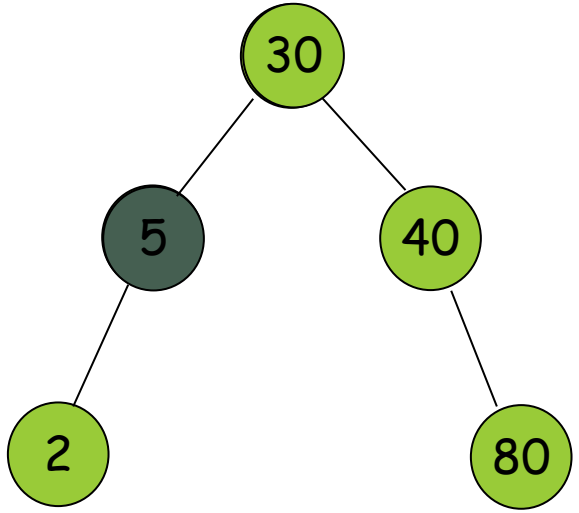
- Delete a leaf node
  - A leaf node which is a right child of its parent
  - A leaf node which is a left child of its parent
- Delete a non-leaf node
  - A node that has one child
  - A node that has two children
    - Replaced by the largest element in its left subtree, or
    - Replaced by the smallest element in its right subtree

Deleting From A Binary Search Tree

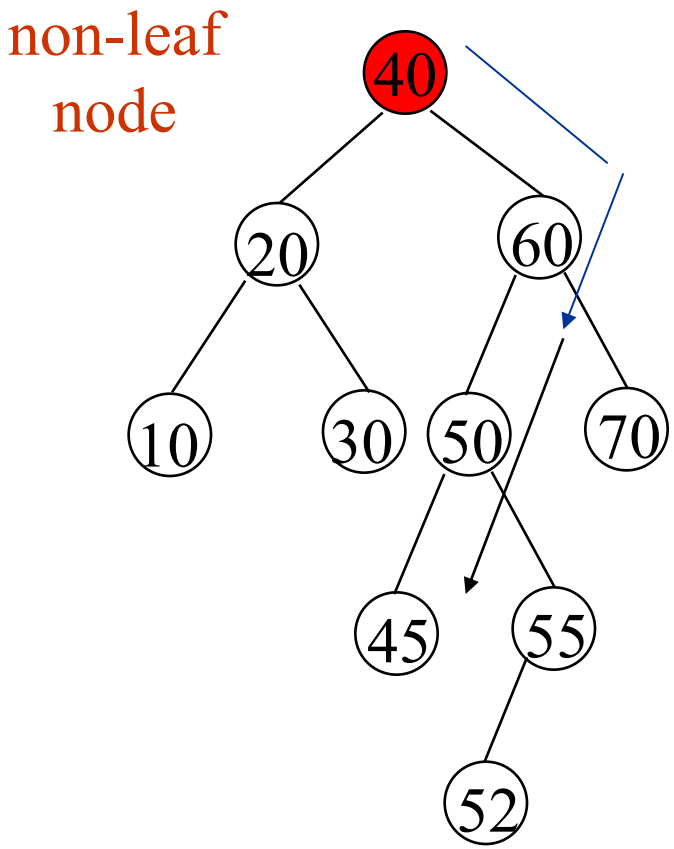
leaf node



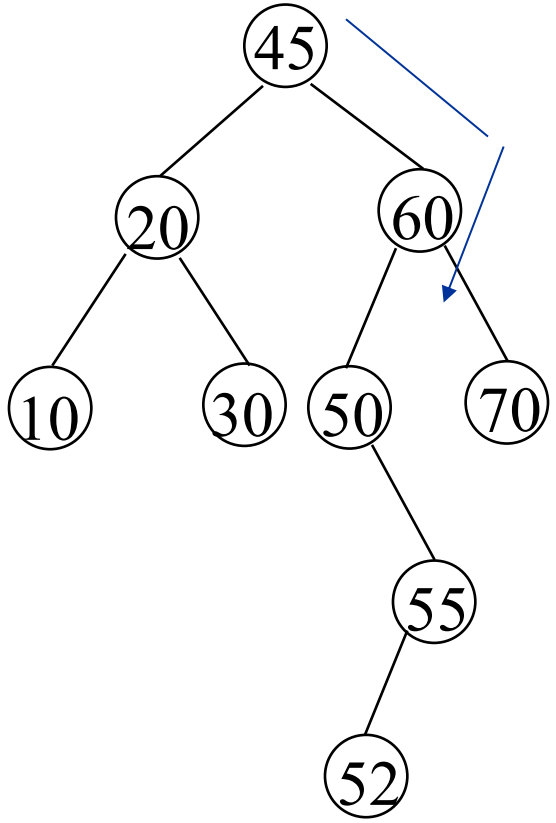
non-leaf node



Deletion from a Binary Search Tree



Before deleting 40



After deleting 40

Function for BST Delete (Replaced by the smallest element in its right subtree)

```
void Delete(Nodeptr *root, int item){
        Nodeptr parent, cur;
         Nodeptr q, succ;

        if (*root== NULL){
        printf("Empty Tree\n");  return;
        }
        //traverse the tree until the item is found or entire tree is traversed
        parent = NULL;
        cur = *root;
        while(cur && (cur->data!= item)){
            parent = cur;
            if (item<cur->data)
                cur = cur->lchild;
            else
                cur = cur->rchild;
        }
        if (cur==NULL) {
        printf("Item Not Found\n");
        return;
        }
```

Function for BST Delete (Replaced by the smallest element in its right subtree)

```
//item found and check for case 1
if (cur->lchild == NULL) //node to be deleted has empty left subtree
   q= cur->rchild;     //get the address of right subtree
else if (cur->rchild == NULL) //node to be deleted has empty right subtree
   q = cur->lchild;          //get the address of left subtree
else //interior node
{
//find inorder successor->smallest element in the right subtree
   parent = cur;
   succ = cur->rchild; //get address of rightchild of node to be deleted*/

   while (succ->lchild){    //move to the leftmost node of succ
      parent = succ;
      succ= succ->lchild;
   }
   cur->data = succ->data;//exchange the data of current and succ;
   cur = succ;
   q = cur->rchild;
}
```

Function for BST Delete (Replaced by the smallest element in its right subtree)

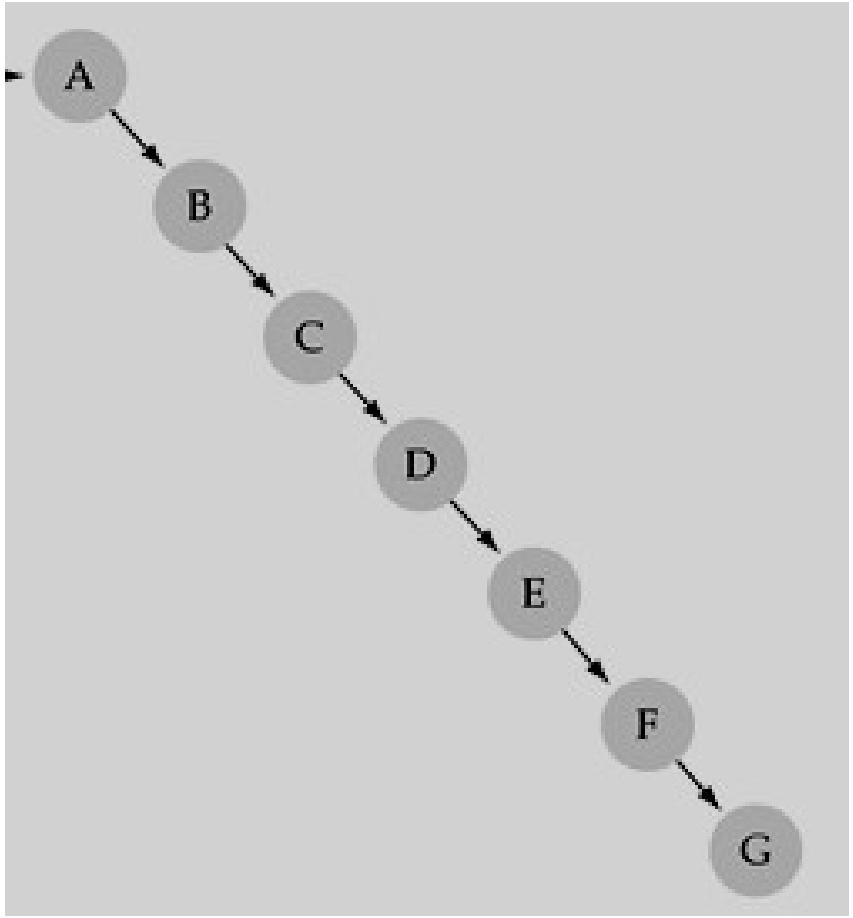
```
    if (parent == NULL){
          free(cur);
          *root = q;
          return;
        }
if (cur== parent->lchild)
          parent->lchild = q;
   else
          parent->rchild = q;
       free(cur);
       return;
}
```

Applications for BST

- Binary Search Tree: Used in *many* search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
- Binary Space Partition: Used in almost every 3D video game to determine what objects need to be rendered.
- Binary Tries: Used in almost every high-bandwidth router for storing router-tables.
- Heaps: Used in implementing efficient priority-queues. Also used in heap-sort.
- Huffman Coding Tree:- used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
- GGM Trees - Used in cryptographic applications to generate a tree of pseudo-random numbers.
- Syntax Tree: Constructed by compilers and (implicitly) calculators to parse expressions.

What is a Degenerate BST?

- A degenerate binarysearch tree is one where most or all of the nodes contain only one sub node.
- It is unbalanced and, in the worst case, performance degrades to that of a linked list.
- If your add node function does not handle rebalancing, then you can easily construct a degenerate tree by feeding it data that is already sorted.

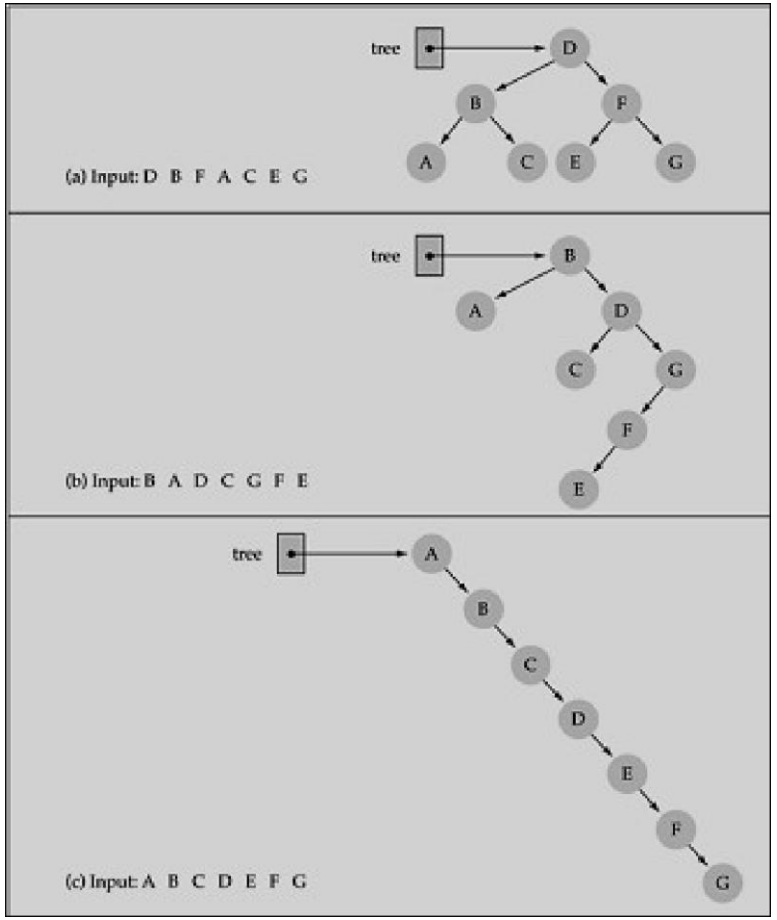


Does the order of inserting elements into a tree matter?

- Yes, certain orders might produce very unbalanced trees or degenerated trees!

- Advanced tree structures, such as **red-black trees, AVL trees**, guarantee balanced trees.

Does the order of inserting elements into a tree matter?



Unbalanced trees are not desirable because search time increases!

Advanced tree structures, such as **red-black trees, AVL trees**, guarantee balanced trees.

Better Search Trees

- Prevent the degeneration of the BST :
- A BST can be set up to maintain balance during updating operations (insertions and removals)
- Types of ST which maintain the optimal performance in other words balanced trees:
- splay trees
- AVL trees
- 2-4 Trees
- Red-Black trees
- B-trees

AVL TREE

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balance condition.

A tree is said to be balanced if for each node, the number of nodes in the left subtree and the number of nodes in the right subtree differ by at most one.

A tree is said to be height-balanced or AVL if for each node, the height of the left subtree and the height of the right subtree differ by at most one.

Height of AVL tree is 1.44log(N+2) -0.328 » logN and the searching complexity » O(log N)

AVL Tree Definition

- Binary tree.
- If T is a nonempty binary tree with T_L and T_R as its left and right sub trees, then T is an AVL tree iff
  - T_L and T_R are AVL trees, and
  - $\mathbf{|h_L - h_R| <= 1}$ where h_L and h_R are the heights of T_L and T_R, respectively

Balance Factor

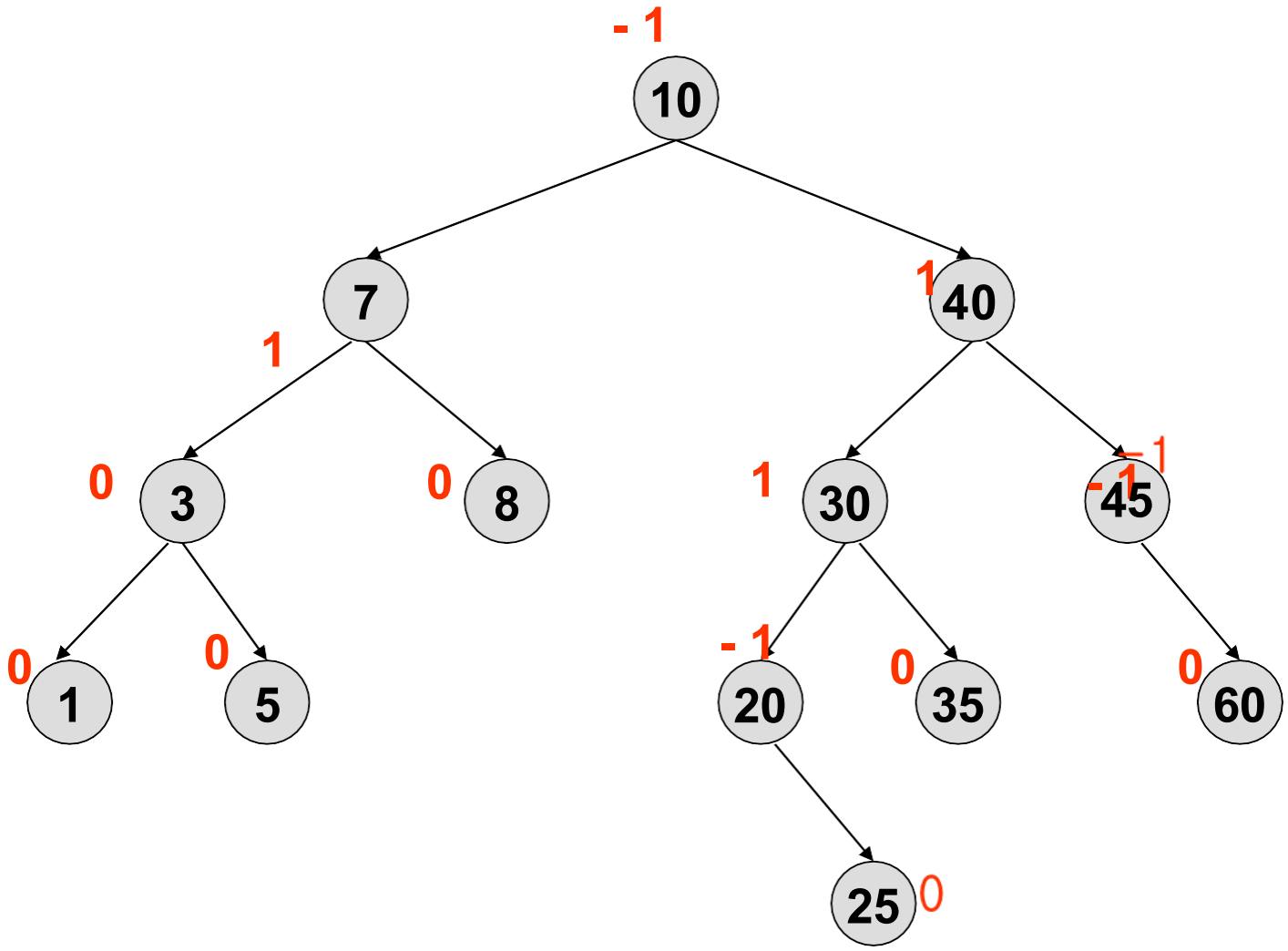
- AVL trees are normally represented using the linked representation
- To facilitate insertion and deletion, a Balance Factor (BF) is associated with each node
- **Balance Factor = Height(Left-sub tree) - Height(Right-sub tree)) (i.e.) h_L-h_R**
- Balance factor of each node in an AVL tree must be –1, 0, or 1

  -1: height of the right sub tree is one greater than the left sub-tree.

  0: height of the left and right sub-trees is equal.

  +1: height of the left sub-tree is one greater than the right sub-tree.

- Example



Properties of AVL Tree

- The height of an AVL tree with n nodes is O(log n)
- For every value of n, $n \geq 0$, there exists an AVL tree
- An n-node AVL search tree can be searched in O(height) = O(log n) time
- A new node can be inserted into an n-node AVL search tree so that the result is an $n+1$ node AVL tree and insertion can be done in O(log n) time

- A node can be deleted from an *n*-node AVL search tree, *n>0*, so that the result is an *n-1* node AVL tree and deletion can be done in O(log *n*) time
- After insertion and deletion of any node in an AVL tree if the balance factor of any node becomes other than -1, 0, +1 then it is said that AVL property is violated. Then we have to restore the destroyed balance condition.
- The balancing should be such that the entire tree should satisfy AVL property.

Insertion into an AVL Tree

- To insert an element into an AVL search tree, the result may not be an AVL tree. (i.e.) the tree may become unbalanced
- If the tree becomes unbalanced, we must adjust the tree to restore balance - this adjustment is called rotation
- There are four different cases when rebalancing is required after insertion of new node.
- left sub tree of left child (LL)
- right sub tree of left child (RL)
- left sub tree of right child (LR)
- right sub tree of right child (RR)

Definition of Rotation

- Any modification done on an AVL tree in order to rebalance it is called a rotation of an AVL tree.
- Types of rotations are

1. Single Rotation
    - LL rotation(Left-Left)
    - RR rotation(Right Right)

2. Double Rotation

- LR rotation(Left-Right) - The transformation to correct LR imbalance can be achieved by an RR rotation followed by an LL rotation.
- RL rotation(Right-Left) - The transformation to correct RL imbalance can be achieved by an LL rotation followed by an RR rotation.

Insertion Algorithm

- Create new node and insert a new node as leaf node as in ordinary binary search tree.
- Now track the path from insertion point towards root.
- Checks each node of heights of left (n) and right (n) differ by at most 1.

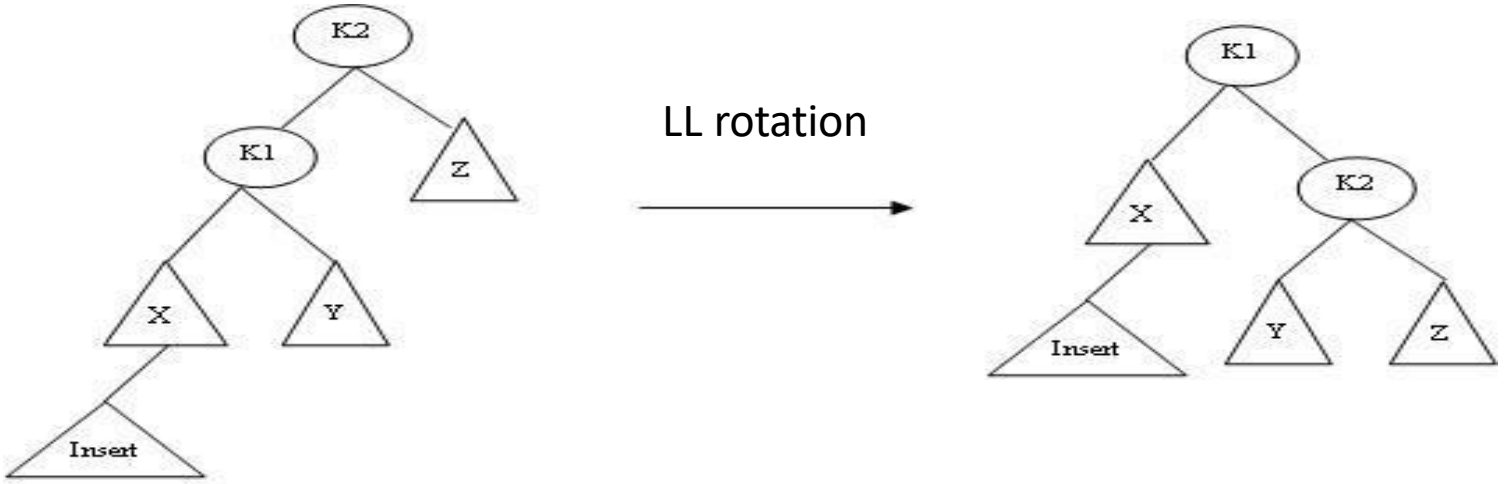
  If yes, move towards parent(n)

  Otherwise restructure by doing either a single or double rotation.

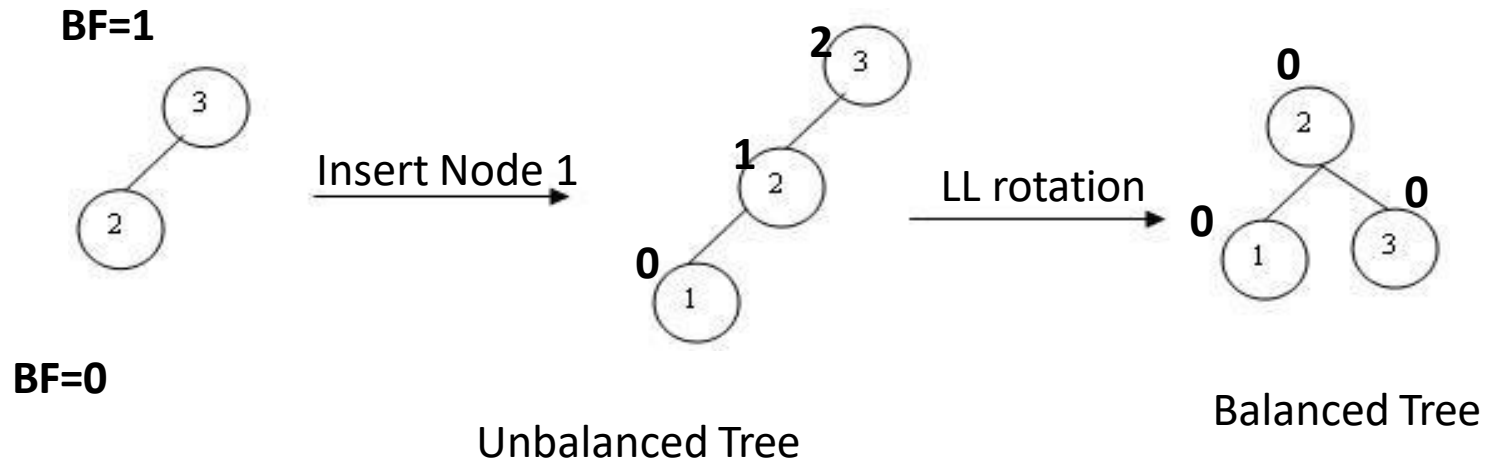
Single Rotation

- (a)LL rotation [Insert Left , Rotate Right]

General

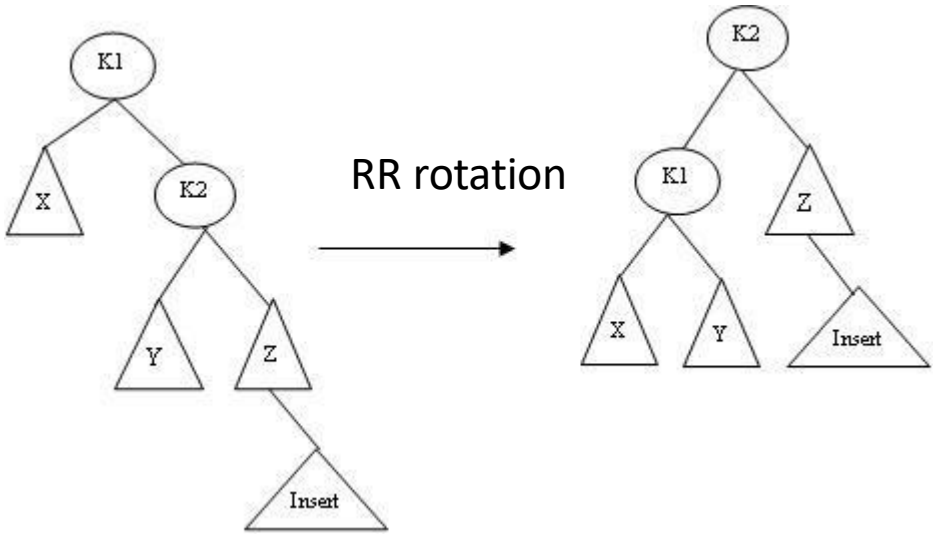


Example:

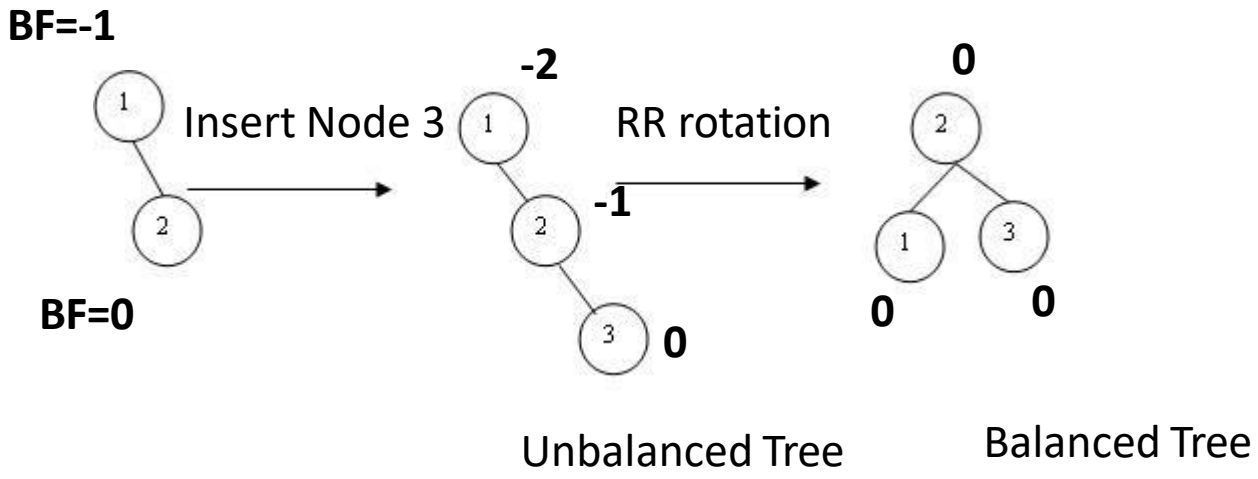


When node 1 is inserted to the left of 2, the tree becomes unbalanced hence rotate right (LL rotation) to make the balanced tree

RR rotation [Insert Right, Rotate Left]



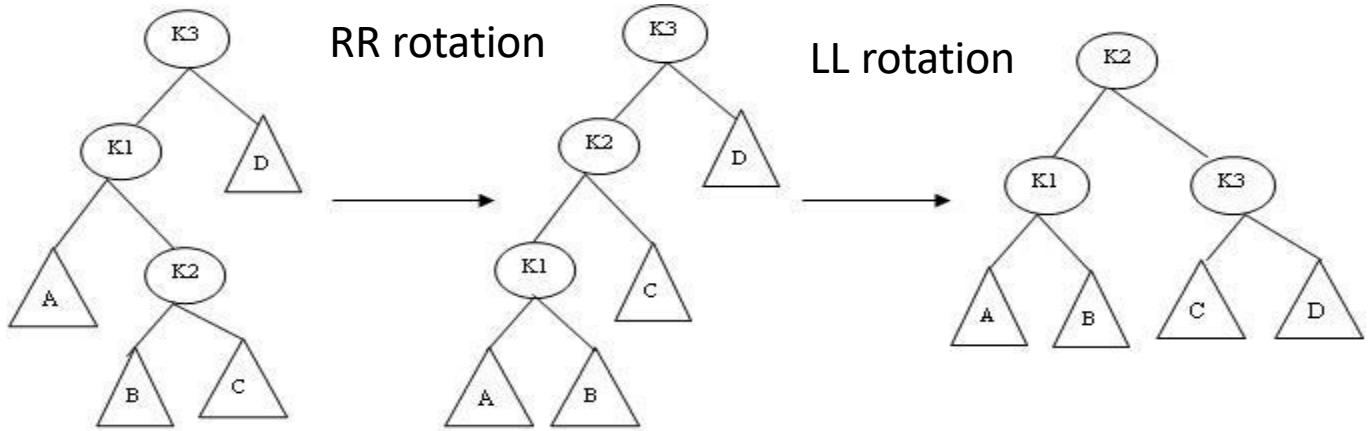
Example:



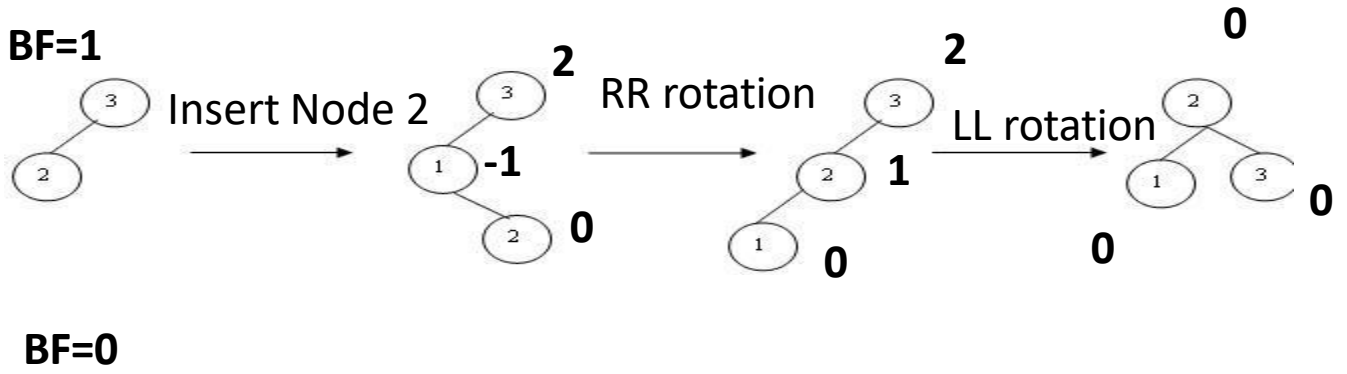
•When node 3 is inserted to the right of 2, the tree becomes unbalanced hence rotate left(RR rotation) to make the balanced tree

Double Rotation

- LR rotation [RR rotation followed by an LL rotation]



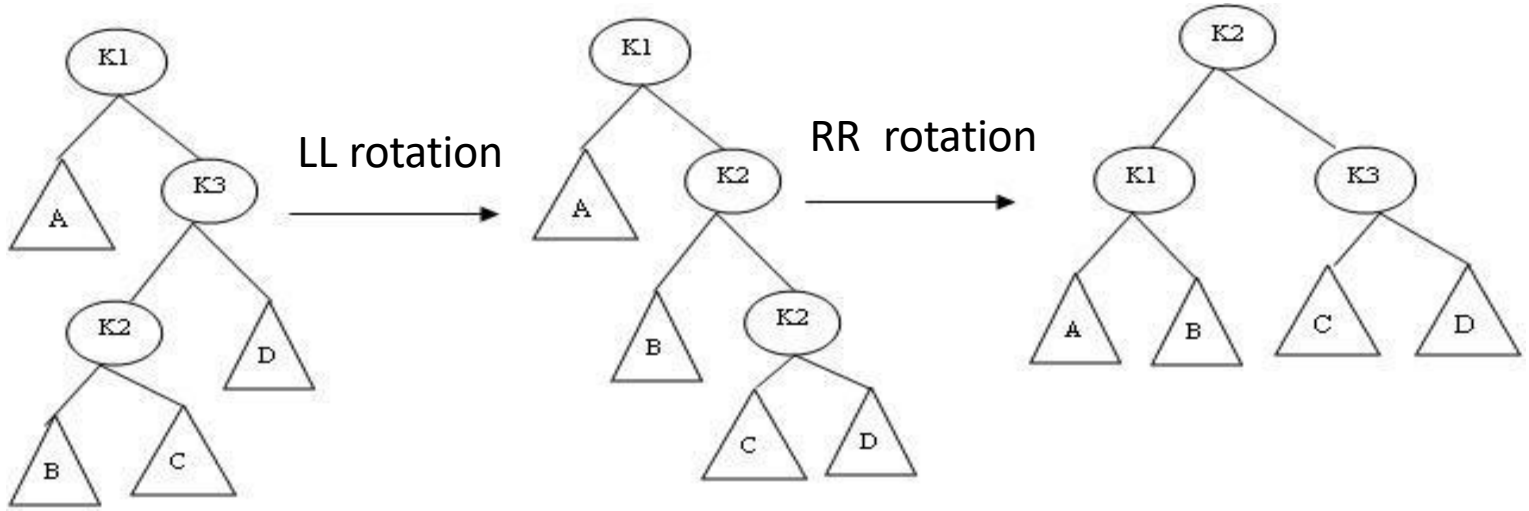
Example



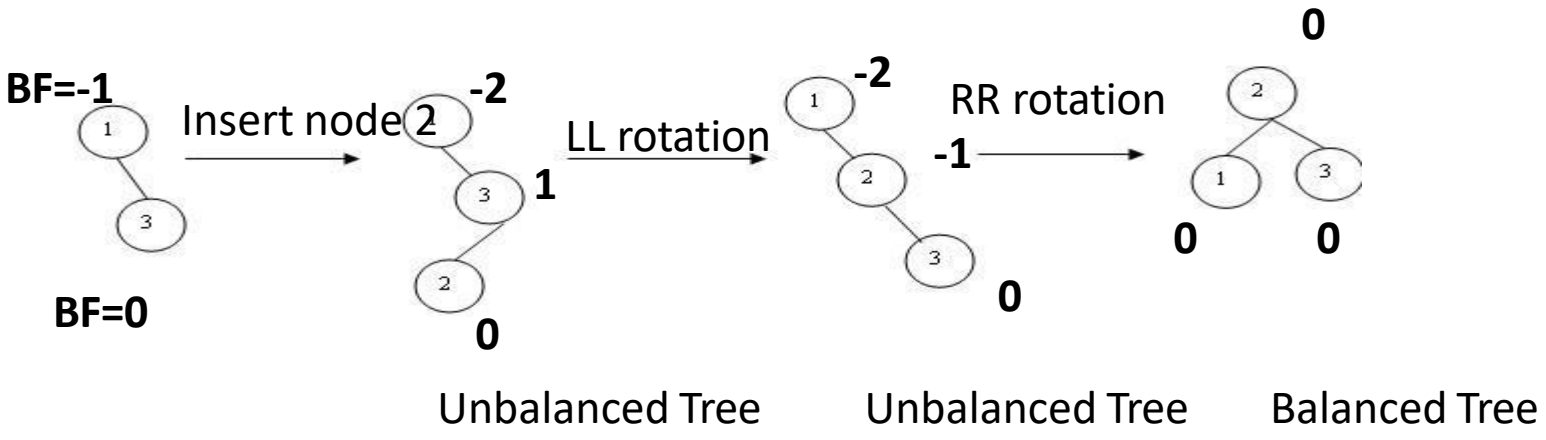
When node 2 is inserted to the right of 1, the tree becomes unbalanced hence rotate left (RR rotation), still it is unbalanced hence rotate right (LL rotation) to make a balanced tree

RL rotation [LL rotation followed by an RR rotation]

- General

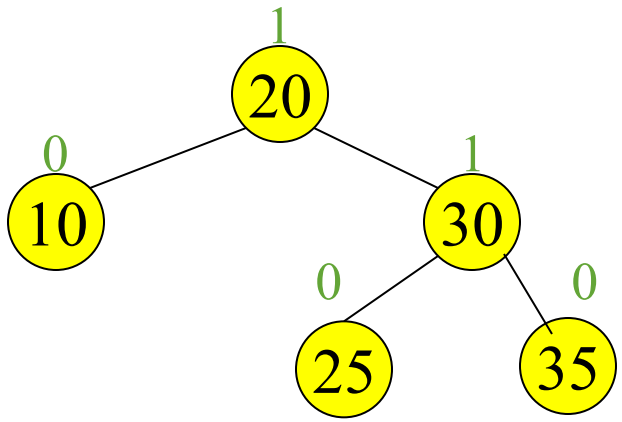


Example



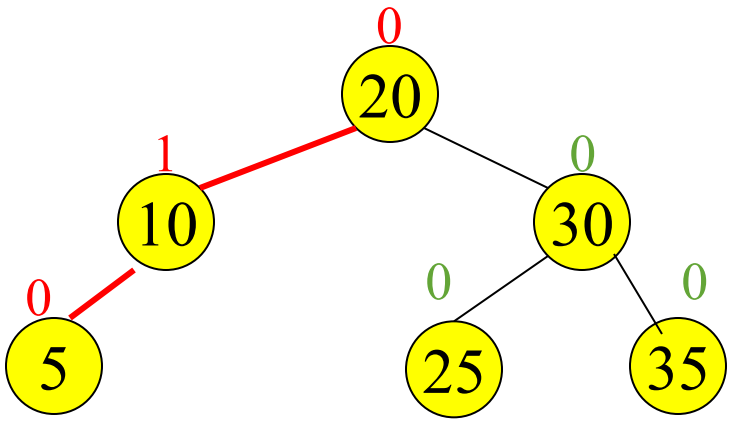
- When node 2 is inserted to the left of 3, the tree becomes unbalanced hence rotate right (LL rotation), still it is unbalanced hence rotate left (RR rotation) to make a balanced tree.

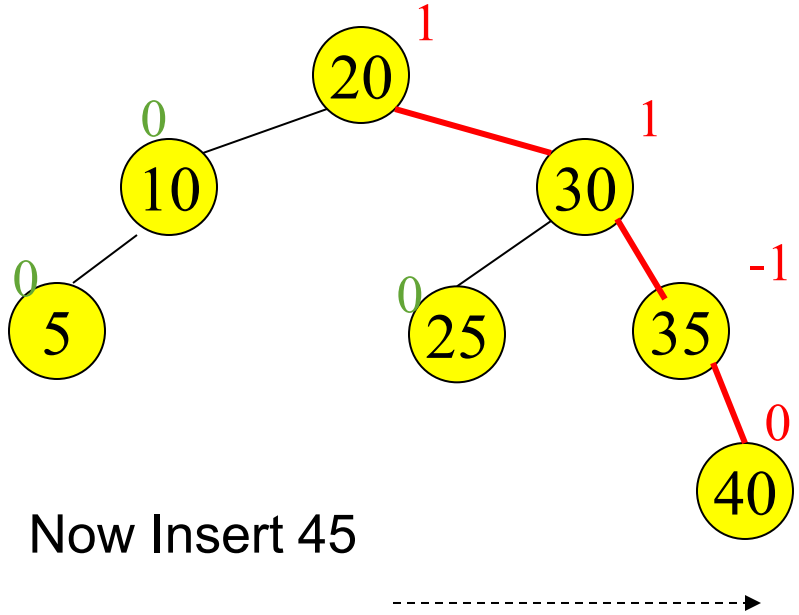
Example of Insertions in an AVL Tree



Insert 5, 40

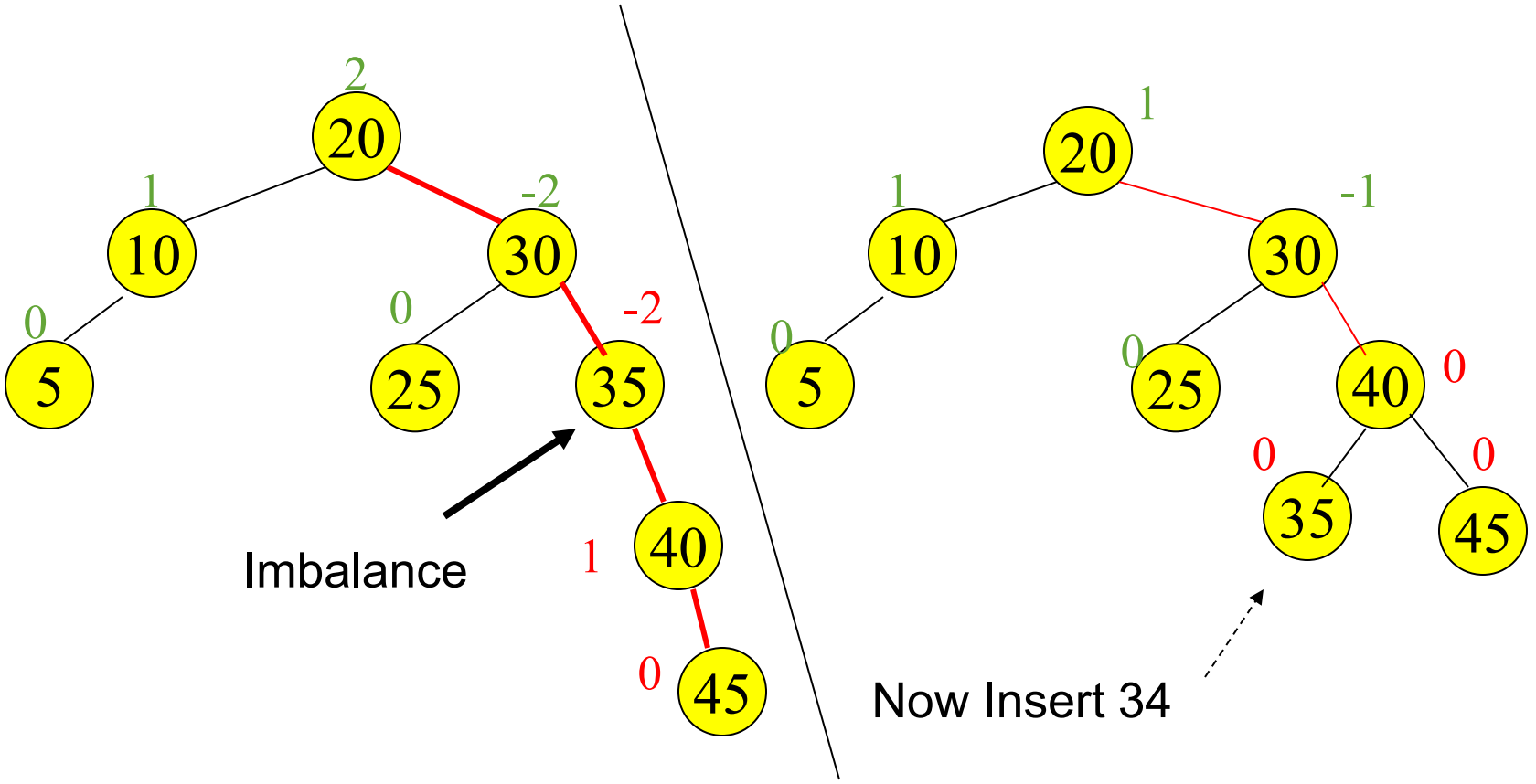
Example of Insertions in an AVL Tree



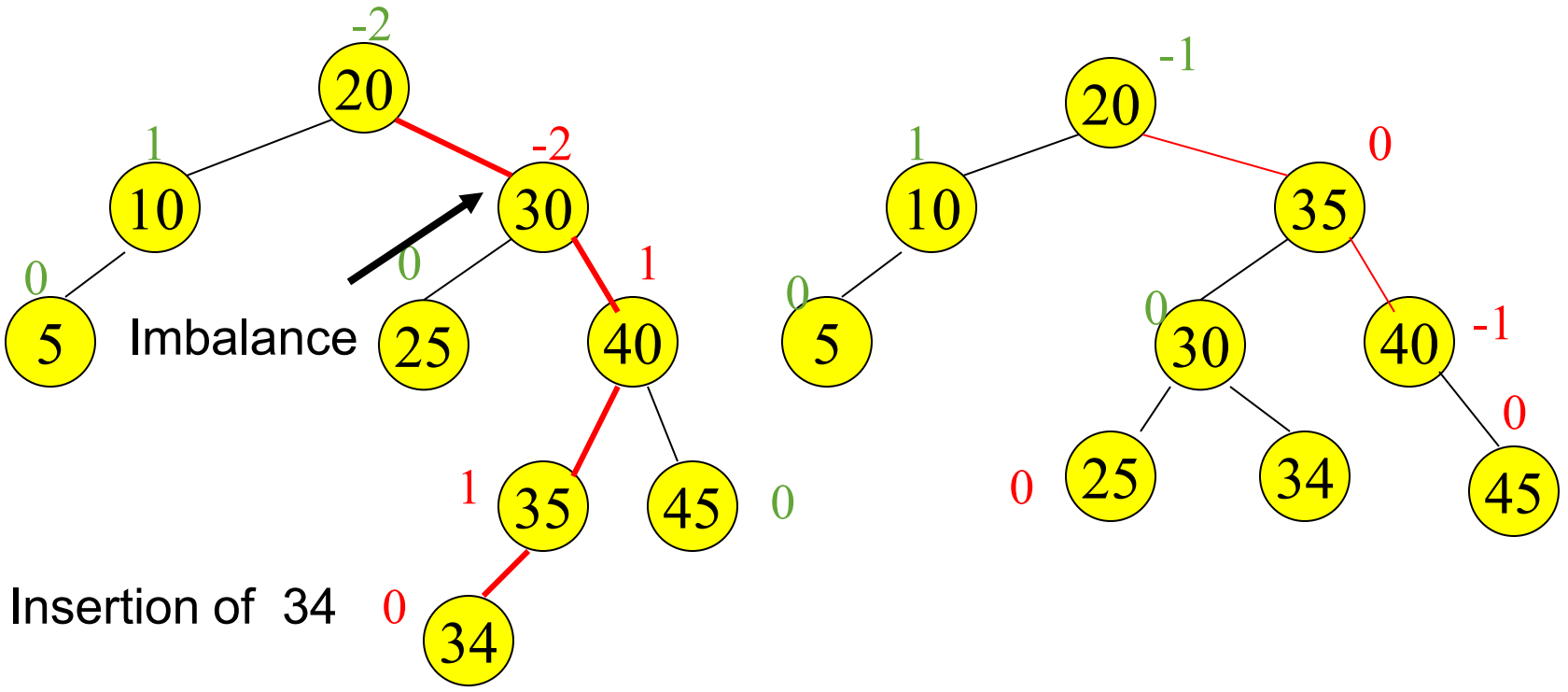


Now Insert 45

Single rotation

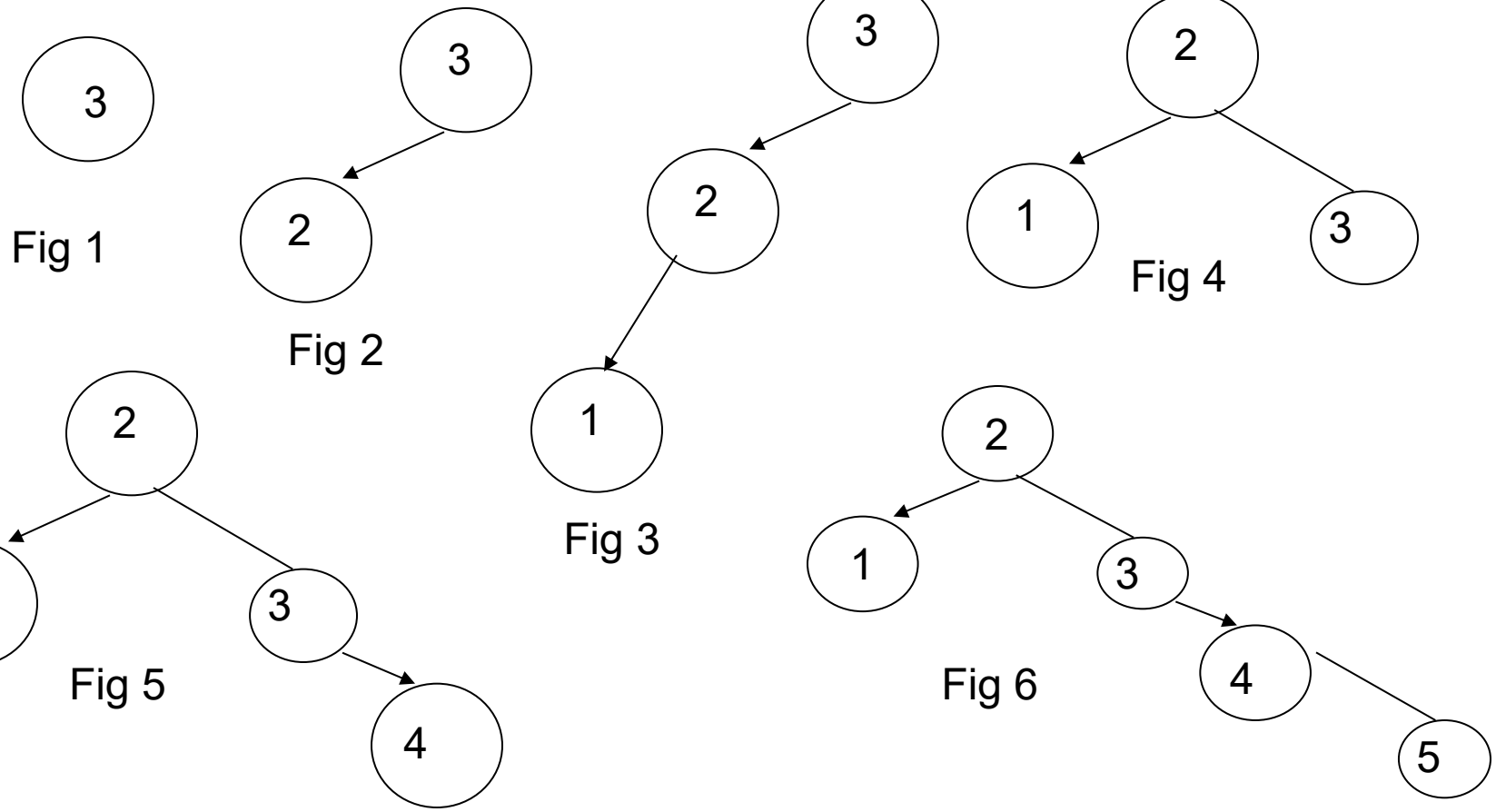


Double rotation



Example 2: AVL Tree

Insert 3,2,1,4,5,6,7, 16,15,14



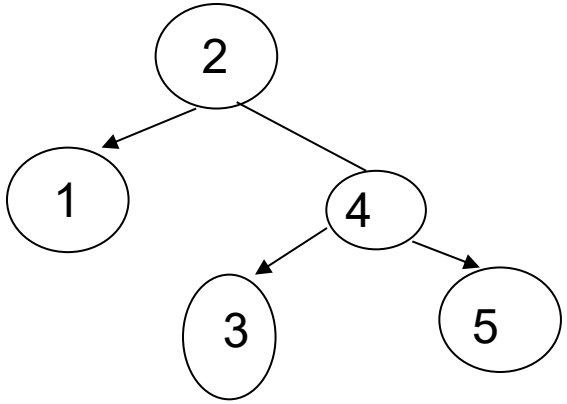


Fig 7

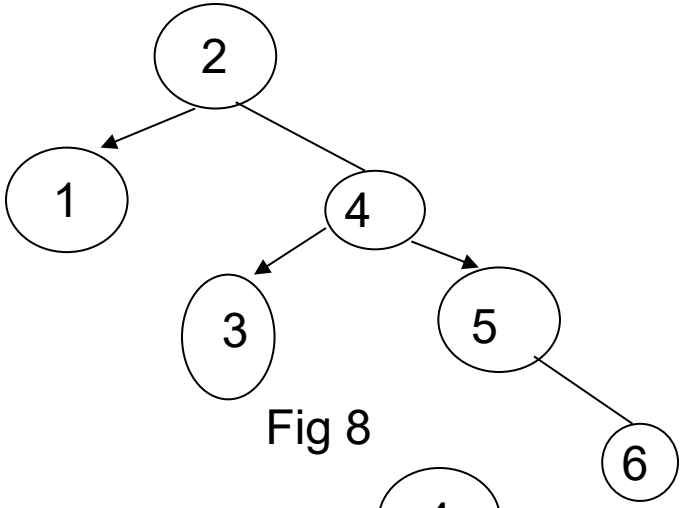


Fig 8

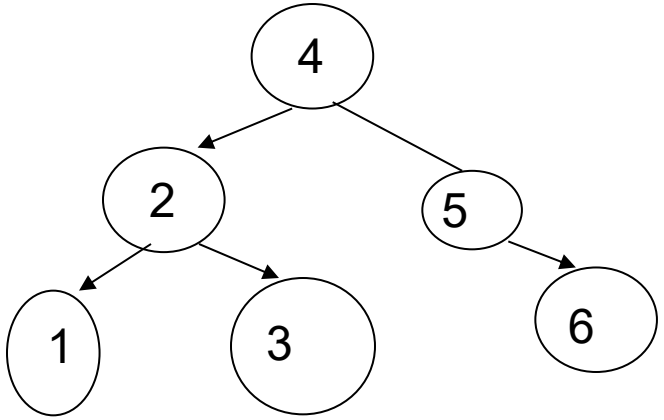


Fig 9

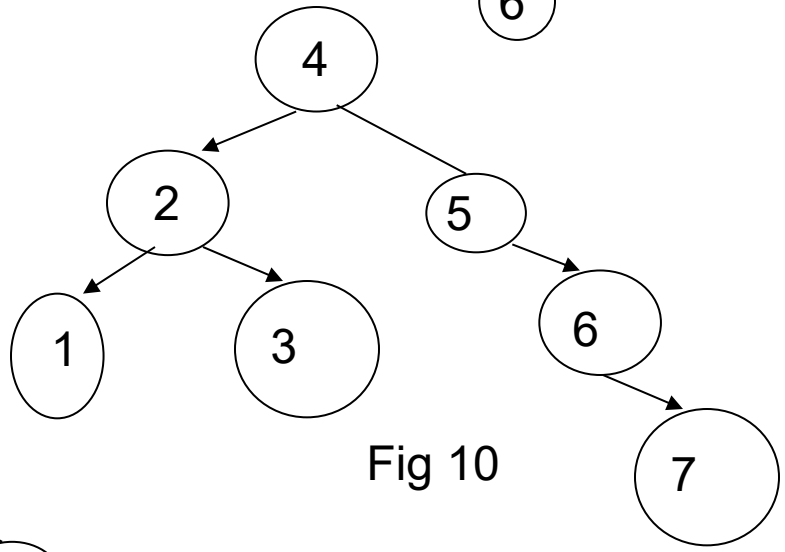


Fig 10

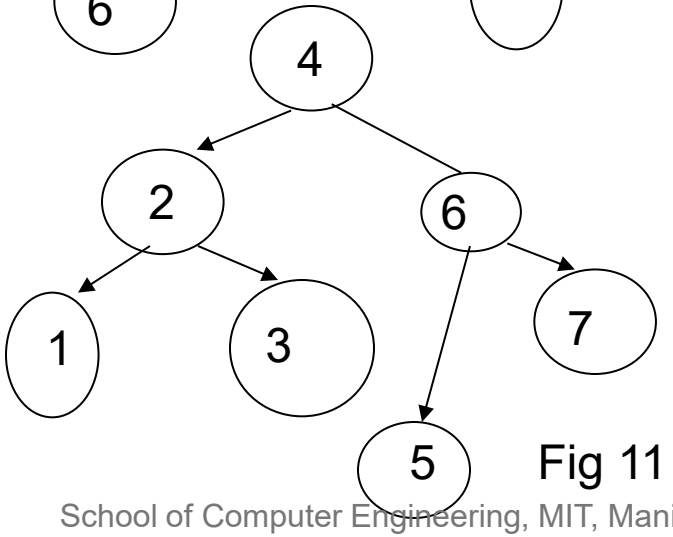


Fig 11

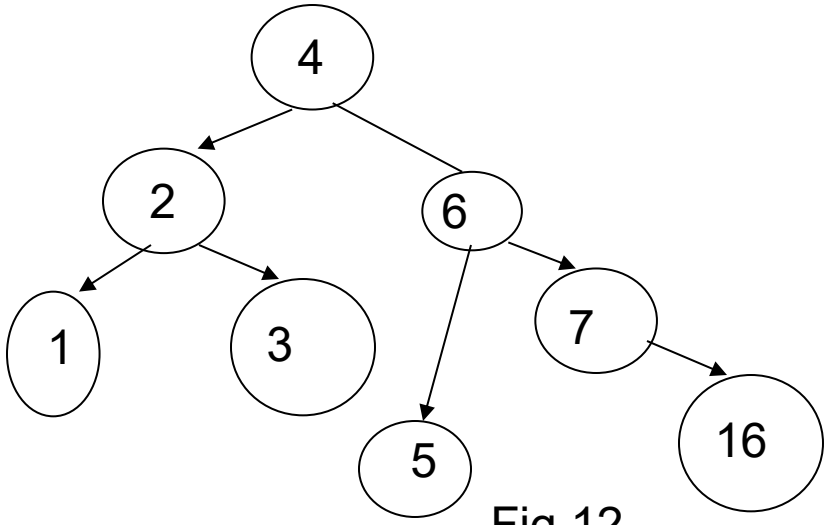


Fig 12

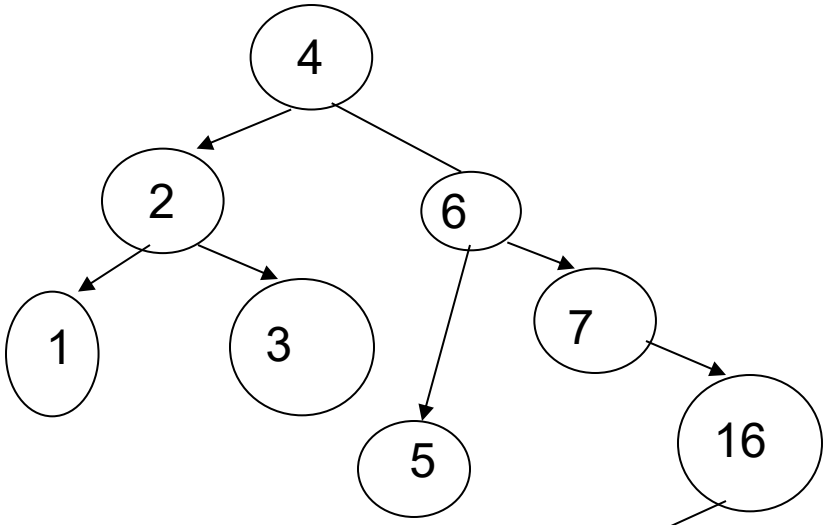


Fig 13

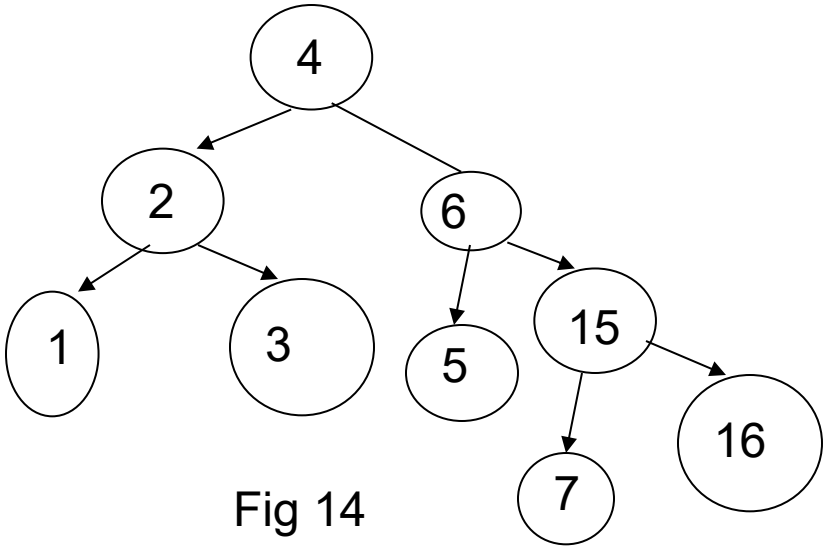


Fig 14

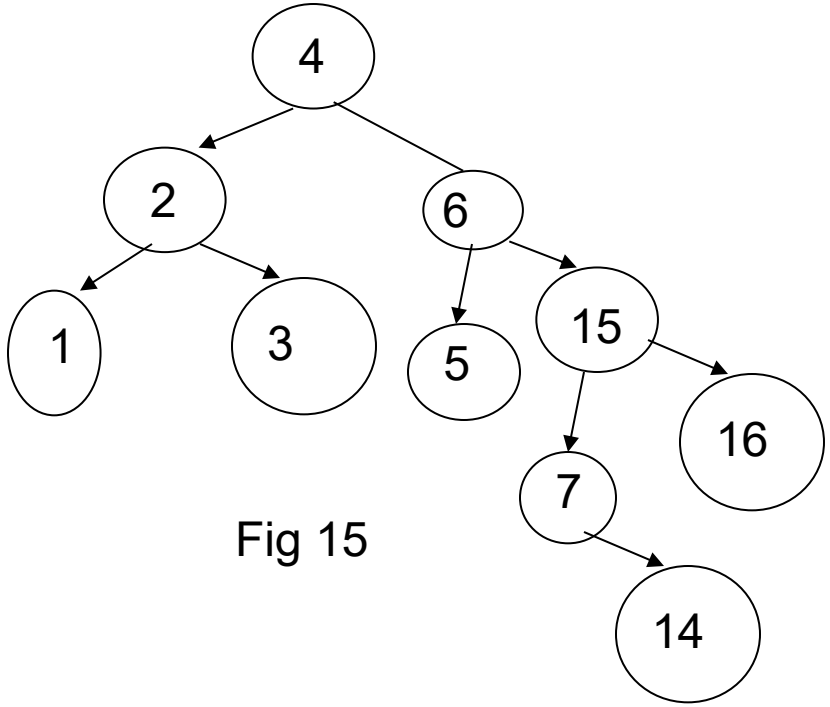


Fig 15

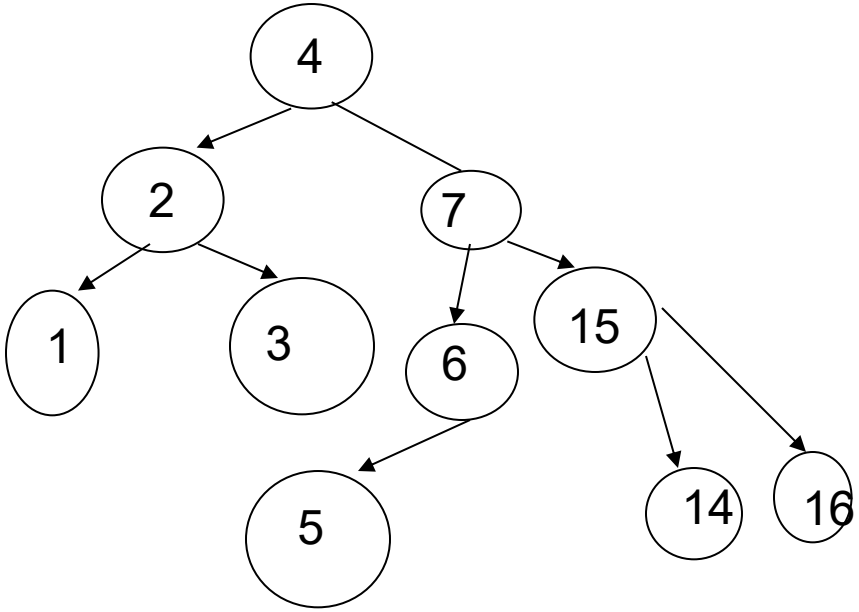


Fig 16

Deletions can be done with similar rotations

Implementation

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))

// Node Declaration for AVL trees
typedef struct AvlNode {
    int element;
    struct AvlNode *left;
    struct AvlNode *right;
    int height;
} AvlNode;
```

Implementation

```
// Function to get height
int height(AvlNode *t) {
    return t == NULL ? -1 : t->height;
}
// Single rotation with left child
void rotateWithLeftChild(AvlNode **k2) {
    AvlNode *k1 = (*k2)->left;
    (*k2)->left = k1->right;
    k1->right = *k2;
    (*k2)->height = MAX(height((*k2)->left), height((*k2)->right)) + 1;
    k1->height = MAX(height(k1->left), (*k2)->height) + 1;
    *k2 = k1;
}
```

Implementation

```
// Single rotation with right child
void rotateWithRightChild(AvlNode **k1) {
    AvlNode *k2 = (*k1)->right;
    (*k1)->right = k2->left;
    k2->left = *k1;
    (*k1)->height = MAX(height((*k1)->left), height((*k1)->right)) + 1;
    k2->height = MAX(height(k2->right), (*k1)->height) + 1;
    *k1 = k2;
}
```

```
// Double rotation: left-right
void doubleWithLeftChild(AvlNode **k3) {
    rotateWithRightChild(&((*k3)->left));
    rotateWithLeftChild(k3);
}
```

```
// Double rotation: right-left
void doubleWithRightChild(AvlNode **k3) {
    rotateWithLeftChild(&((*k3)->right));
    rotateWithRightChild(k3);
}
```

Implementation

```
// Insert element into AVL tree
void insert(int x, AvlNode **t) {
    if (*t == NULL) {
        *t = (AvlNode*)malloc(sizeof(AvlNode));
        (*t)->element = x;
        (*t)->left = (*t)->right = NULL;
        (*t)->height = 0;
    }
    else if (x < (*t)->element) {
        insert(x, &((*t)->left));
        if (height((*t)->left) - height((*t)->right) == 2) {
            if (x < (*t)->left->element)
                rotateWithLeftChild(t);
            else
                doubleWithLeftChild(t);
        }
    }
    else if (x > (*t)->element) {
        insert(x, &((*t)->right));
        if (height((*t)->right) - height((*t)->left) == 2) {
            if (x > (*t)->right->element)
                rotateWithRightChild(t);
            else
                doubleWithRightChild(t);
        }
    }
    // Duplicate: do nothing

    (*t)->height = MAX(height((*t)->left), height((*t)->right)) + 1;
}
```

Implementation

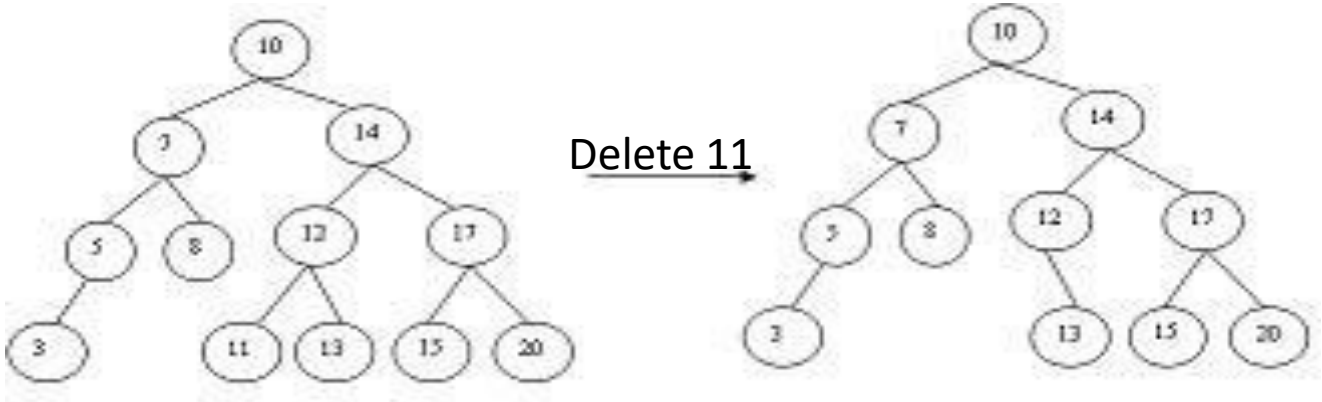
```
// In-order traversal
void inorder(AvlNode *t) {
    if (t != NULL) {
        inorder(t->left);
        printf("%d ", t->element);
        inorder(t->right);
    }
}
```

```
// Test the AVL tree
int main() {
    AvlNode *root = NULL;

    int arr[] = {30, 20, 40, 10, 25, 35, 50, 5};
    int n = sizeof(arr)/sizeof(arr[0]);

    for (int i = 0; i < n; i++)
        insert(arr[i], &root);

    printf("Inorder traversal of AVL tree:\n");
    inorder(root);
    printf("\n");

    return 0;
}
```

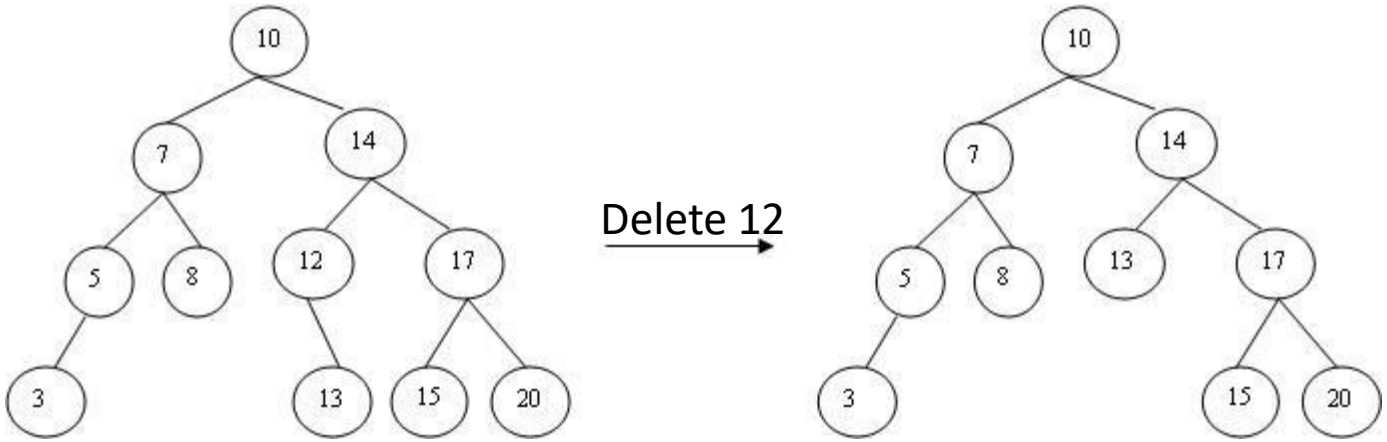
Deletion in AVL Trees

- The deletion algorithm is more complex than insertion algorithm.
- Search the node which is to be deleted
- If the node to be deleted is a leaf node then simply make it NULL to remove.
- If the node to be deleted is not a leaf node then the node must be swapped with its inorder successor. After swap we can remove this node.

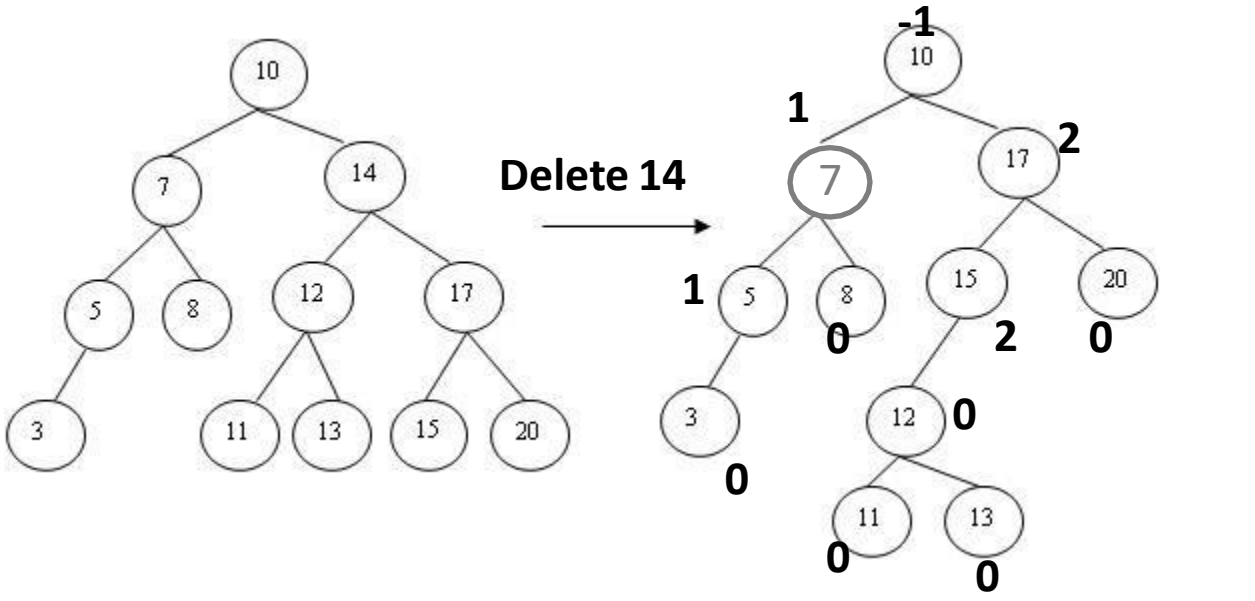
- **Example:**
- Case 1: Leaf Node



- Case 2: One Child

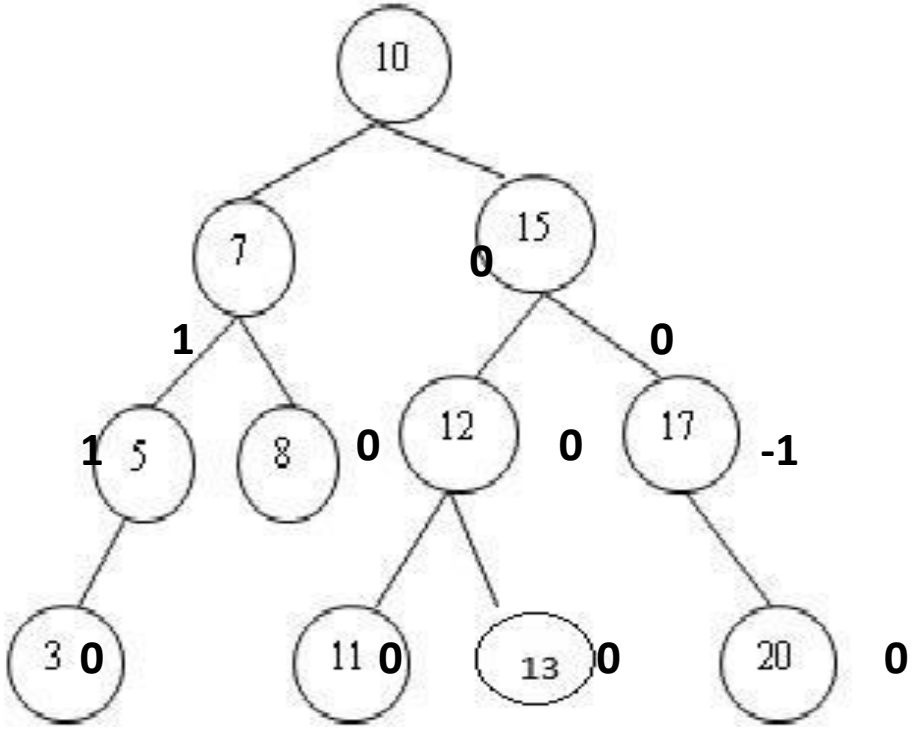


- Case 3: Two Child



Unbalanced (violates AVL property)

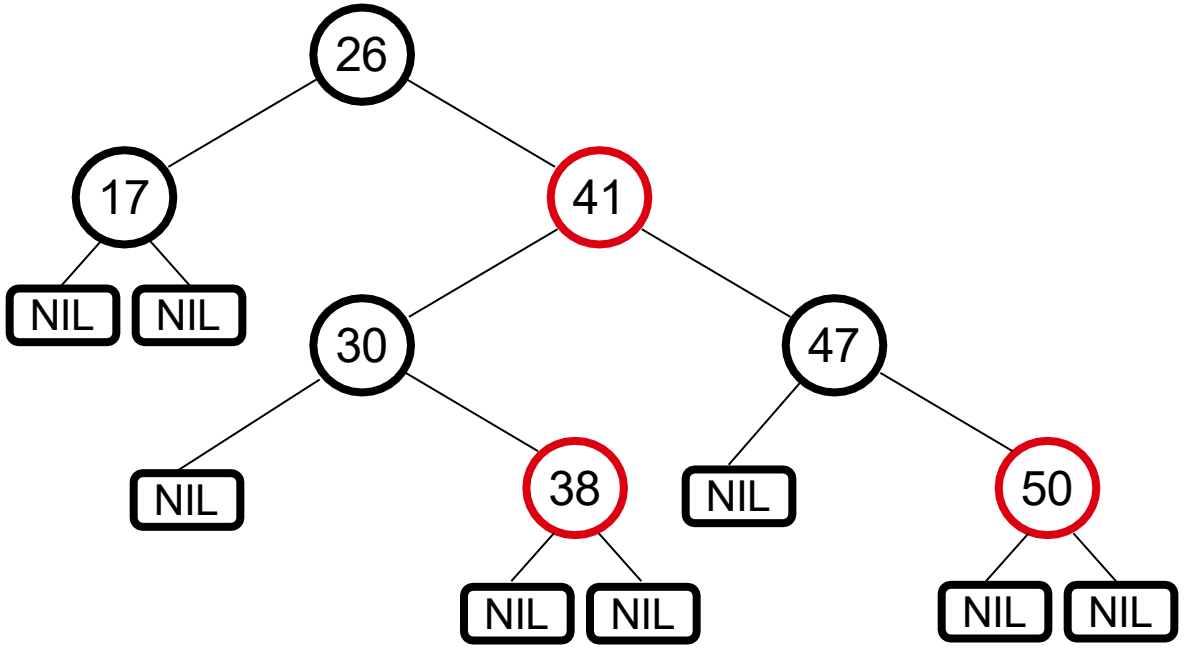
LL rotation


10
7
15
0
1
0
1
5
8
0
12
0
17
-1
3
0
11
0
13
0
20
0

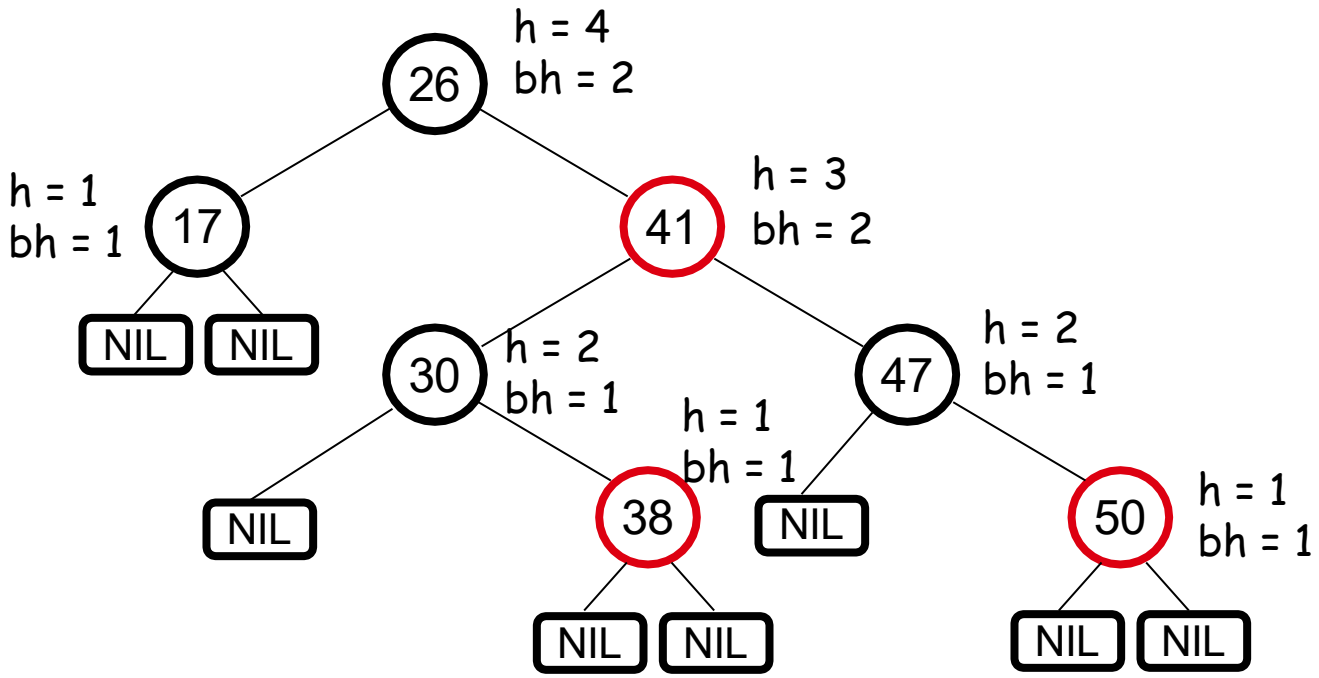
RED-BLACK TREES

- A red-black tree is a balanced binary search tree with the following properties.
- Every node is colored red or black.
- The root is black.
- Every leaf is a NIL node, and is colored black.
- If a node is red, then both its children are black.
- Every simple path from a node to a descendant leaf contains the same number of black nodes=black-height(x).

Example:



- Black-Height of a node:

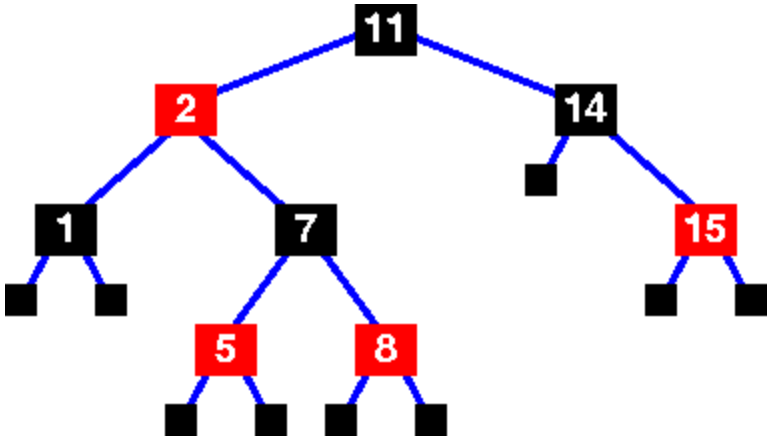


- **Height of a node : ** the number of edges in the longest path to a leaf.
- **Black-height of a node x:** bh(x) is the number of black nodes (including NIL) on the path from x to a leaf, not counting x
- A red-black tree with n internal nodes has height at most 2lg(n + 1).

Insertion in red-black tree

1) If empty tree create black root node
2) Insert new leaf node as red
   a) If its parent is black then done
   b) If parent is red
      i) If black sibling rotate, recolor and done.
      ii) If red sibling then recolor and check again.

Example



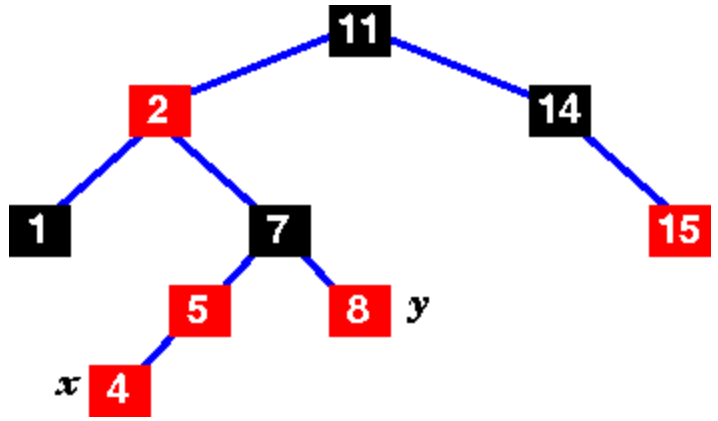
The tree insert routine has just been called to insert node "4" into the tree.

This is no longer a red-black tree - there are two successive red nodes on the path

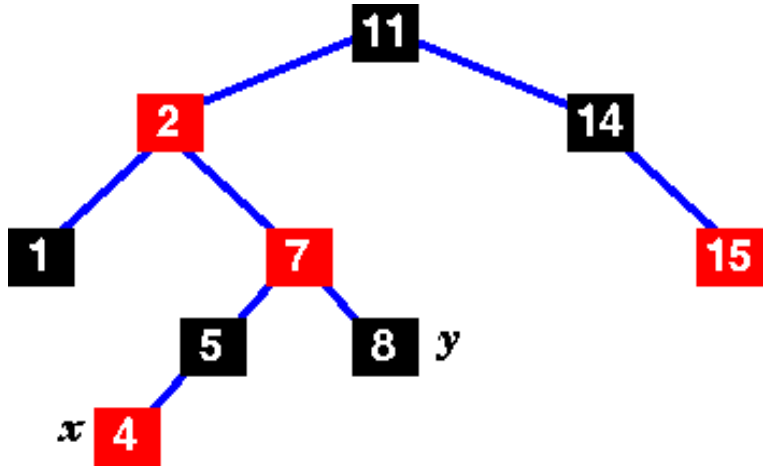
11 - 2 - 7 - 5 - 4

Mark the new node, x, and it's uncle, y.

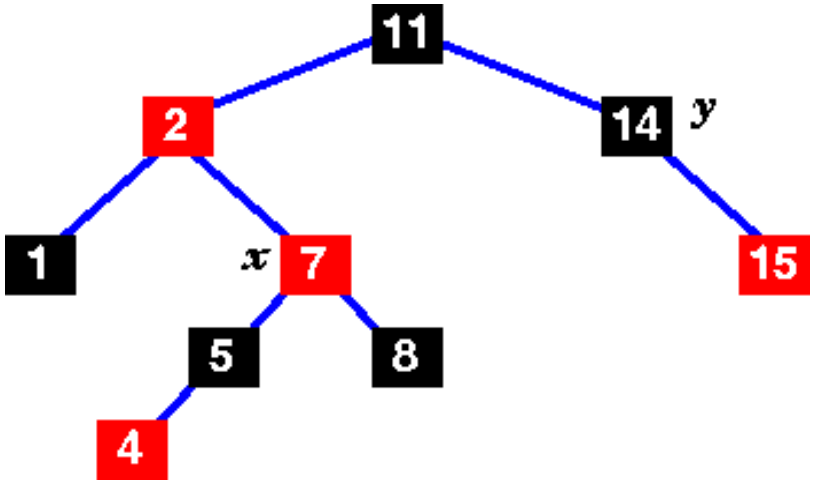
y is red, so we have case 1 ...



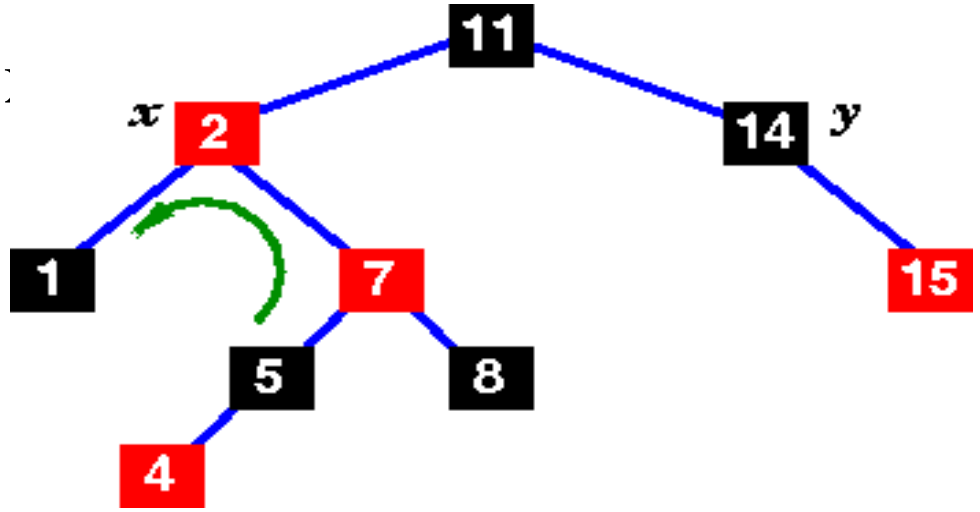
Change the colors of nodes 5, 7 and 8.



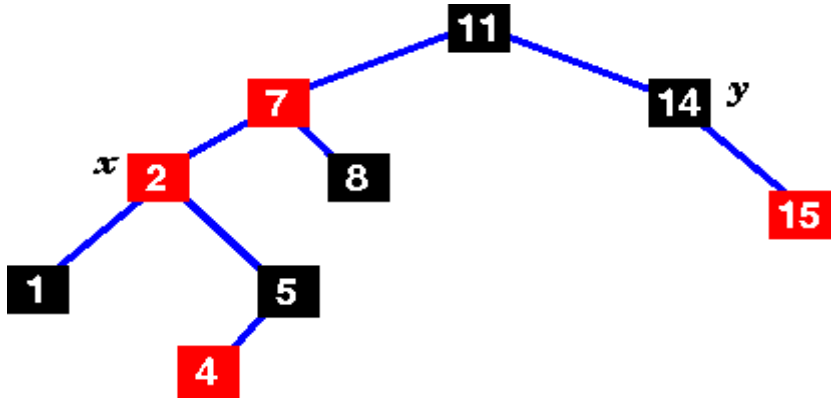
- Move x up to its grandparent, 7.
- x's parent (2) is still red, so this isn't a red-black tree yet.
- Mark the uncle, y.
- In this case, the uncle is black, so we have case 2 .



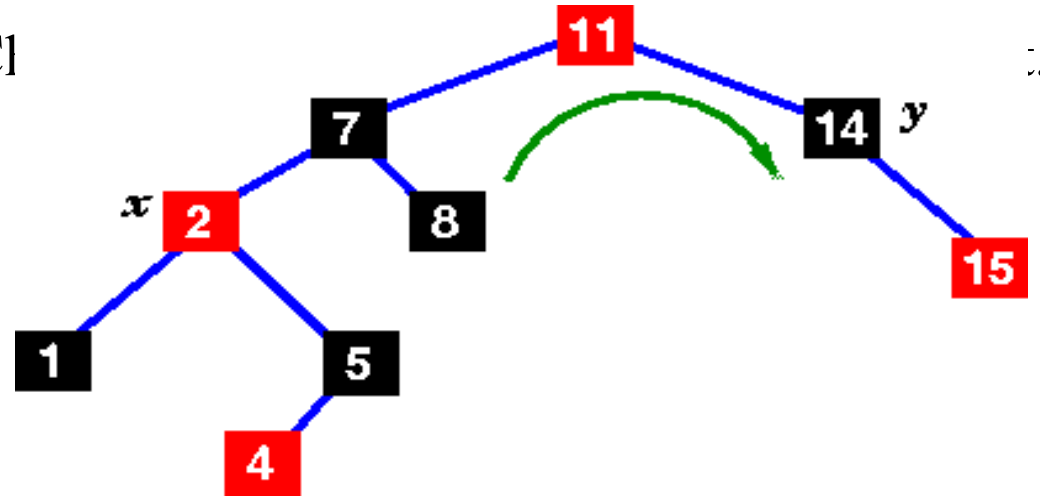
Move :



Still not a red-black tree... the uncle is black, but x's parent is to the left .



Cl



This is now a red-black tree,