Non-Linear Data Structures

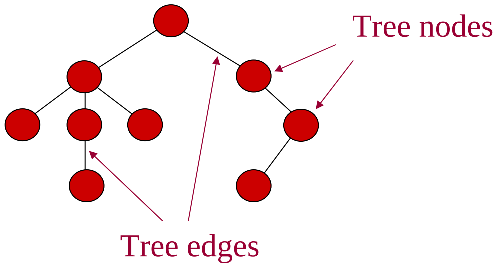
General Trees

Introduction

- A **tree** is a data structure used to represent different kinds of data and help solve a number of algorithmic problems
- Game trees (e.g. chess), UNIX directory trees, sorting trees … etc.
- We will study extensively two useful kind of trees: **Binary Search Trees** and **Heaps**

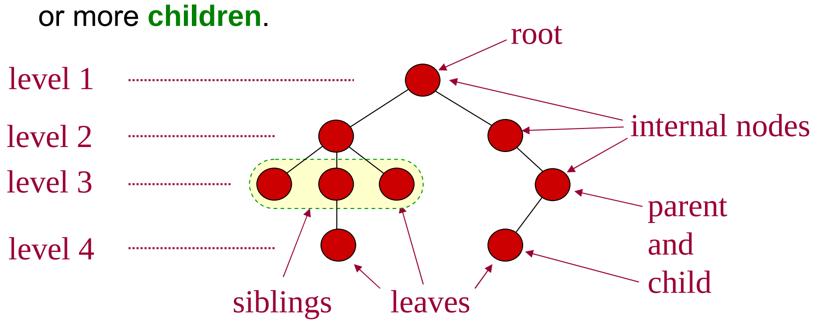
Definitions

- Trees have **nodes**.
- They also have **edges** that connect the nodes.
- Each node contains information about the element it represents
- Between any two nodes there is *always only one* **path**.



Relationships Between Tree Nodes

- Trees are rooted. Once the **root** is defined (by the user) all nodes have a specific **level**.
- Trees have **internal** nodes and **leaves**.
- Every node (except the root) has a **parent** and it also has zero or more **children**.

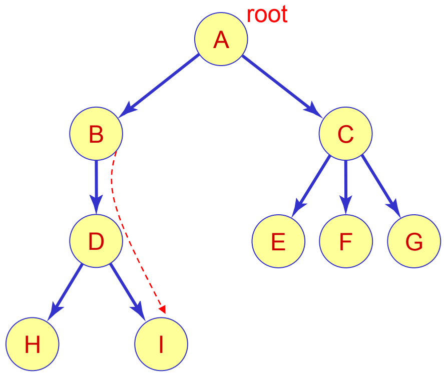


Relationships Between Tree Nodes

- The **root** node is unique, it has no predecessors, but may have many successors.
- A **leaf** node has a unique predecessor, and no successors.
- An **internal** node has a unique predecessor, and at least one successor.

Tree Definitions:

- A **tree** is a finite set of one or more nodes. One of these nodes is called the **root** node. The remaining nodes, if any, are partitioned into $n >= 0$ disjoint sets, each of which is a tree. (Note: this is a recursive definition)
- A **simple path** is a sequence of nodes, $n_1, n_2, \ldots, n_k$, such that they are all distinct and there exists an edge between each pair of them.
- A **path length** is the number of nodes in the simple path.
- A **path length** is also one plus the number of edges in the simple path.

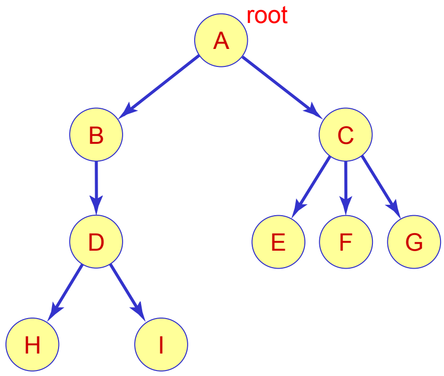


Example:

path: B, D, I
length: 3

Tree Definitions:

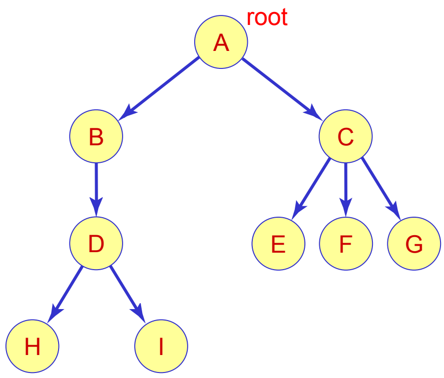
- A **parent** is the predecessor of a node.
- A **child** is the successor of a node.
- **Siblings** are children nodes of the same parent.
- A **leaf** node is a node that has no children.
- **Ancestors** of a node are all the nodes along the unique simple path from the root to that node.
- **Descendents** of a node are all its children and all descendents of each one of its children.
- A **subtree** starting at node n, is that node and all of its descendents.



Example: Ancestors of D: A, B
Descendents of B: D, H, I

Tree Definitions:

- **Degree of a node** is the number of its children.
- **Degree of a tree** is the maximum degree of the nodes in the tree.
- **Level of a node**: The root is at level 1, If a node is at level L, then its children are at level L+1.
- **Height of a tree**: is the maximum level of any node in the tree. Also it is the length of the longest path.
- A **forest** is a set of n >= 0 disjoint trees.



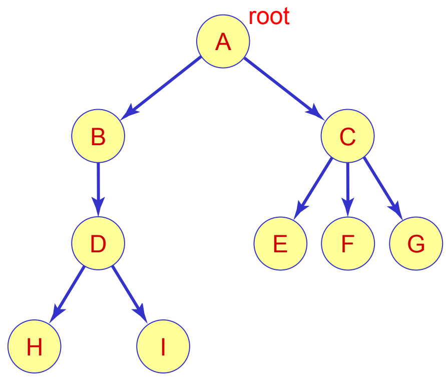
Example: Degree of B: 1, of tree: 3
Height of tree: 4

Tree Representation: As a Linked Structure

- To represent the structure of a tree as a linked structure in memory, a tree node representation must have a variable number of fields depending on the number of branches.

A tree node:

Information fields		
Link 1	…	Link n



- This is not practical

Tree Representation: As a Linear Linked Lists Structure

- A structure of linear linked lists of fixed size nodes can be used to represent a tree structure in memory.
- Use two kinds of nodes:
  - Information node, representing the actual node of the tree.
  - Link node, representing its links.
- Most of the list operations can be used with this structure

Information node structure:

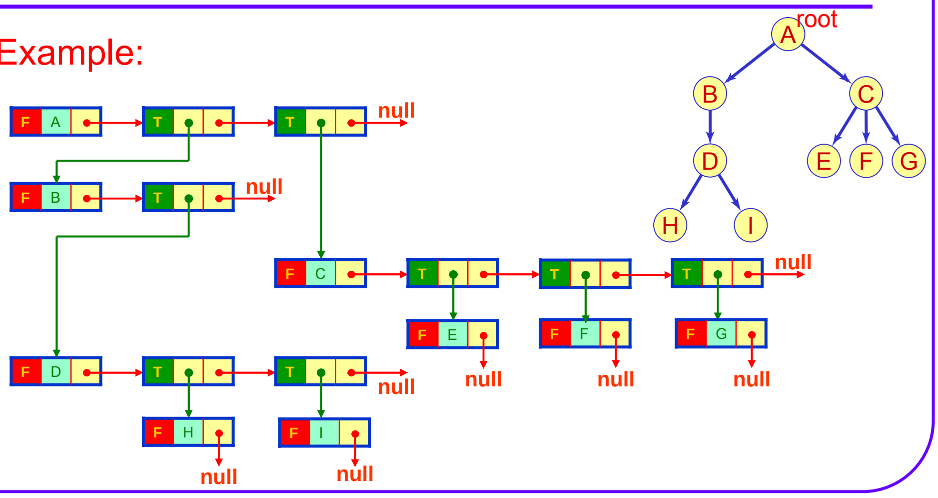
false: Data Node	Information fields	Link to its first child link node

Link node structure:

true: Link Node	Link to the child's info node	Link to next sibling's link node

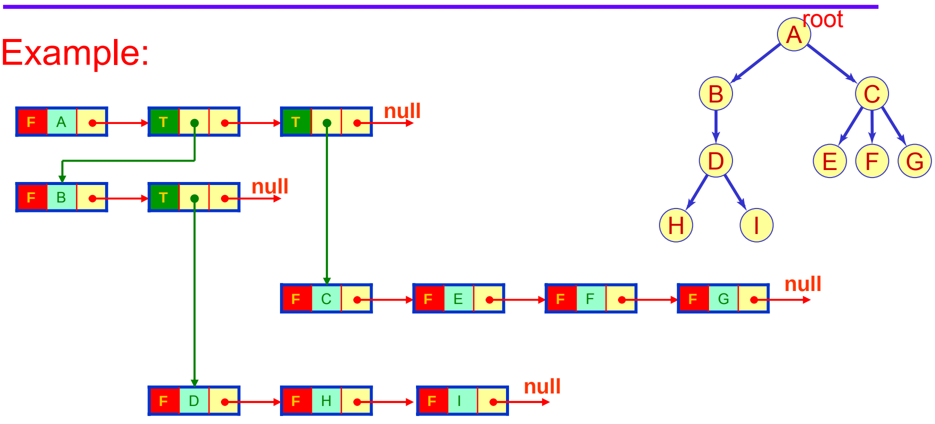
Tree Representation: As a Linear Linked List Structure

Example:



Tree Representation: As a Linear Linked List Structure

Example:



THIS IS NOT CORRECT

Tree Representation Using Other Structures

- Another data object called The Binary Tree can be used to represent a general tree
- The Binary Tree can also be used to represent a forest as we shall see later

Non-Linear Data Structures

Binary Trees

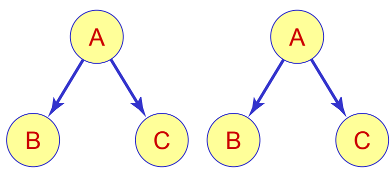
Binary Trees

- A binary tree is an important data object

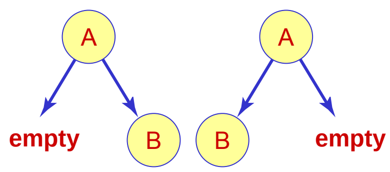
- A binary tree differs from an ordinary tree as follows:
  - Each node of a binary tree can have at most two children
  - Each child is identified as being either left or right child of its parent node.
  - A binary tree may be empty

Definition

- A **Binary Tree** is a finite set of nodes, which is either empty or consists of a root and two disjoint binary trees called the left subtree or the right subtree. (Note: this is a recursive definition)



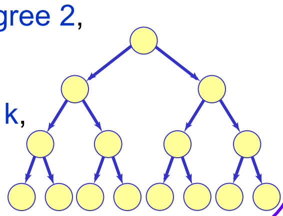
Identical Binary Trees



Not Identical Binary Trees

Properties of Binary Trees

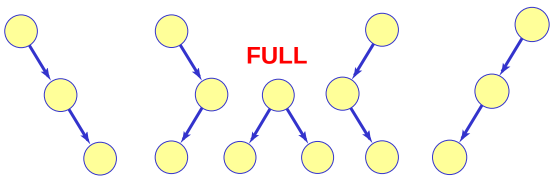
1. The maximum number of nodes on level i of a binary tree is: 2^{i-1}, i >= 1
2. The maximum number of nodes in a binary tree of depth k is: $2^k - 1$, k >= 1
3. For any non-empty binary tree, T, if n_0 is the number of leaf nodes, and n_2 is the number of nodes of degree 2, then: $n_0 = n_2 + 1$
4. A **Full Binary Tree** is a binary tree of depth k, that has $2^k - 1$ nodes.



Properties of Binary Trees

5. The height of a full binary tree with n nodes is: $h = \log_2 (n+1)$
6. Many distinct binary trees with the same number of nodes exist.

   Example: n = 3,



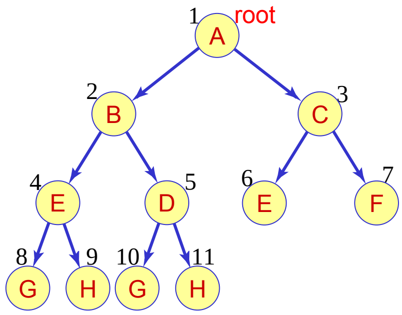
Five distinct trees are possible

7. A **Perfectly Balanced tree** is a binary tree where, for each node, the number of nodes in its left and right sub-trees differ by at most one. Such a tree has minimum height.

Properties of Binary Trees

8. A **Complete tree** is a binary tree with n nodes and depth k, where, its nodes correspond to the nodes that are numbered from 1 to n in the full binary tree of depth k

Notice: leaves are only in the last two bottom levels, with bottom one placed as far left as possible

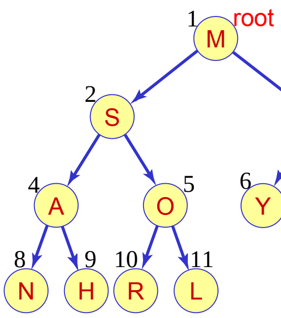


Binary Tree Representation: Using Arrays

- A Complete Binary Tree can be represented in memory with the use of an array, A, so that all nodes can be accessed in O(1) time:
  - Label nodes sequentially top-to-bottom and left-to-right
  - The root node is always at A[1]
  - The left child of A[i] is at position A[2i]
  - The right child of A[i] is at position A[2i + 1]
  - The parent of A[i] is at A[i/2]

Binary Tree Representation: Using Arrays

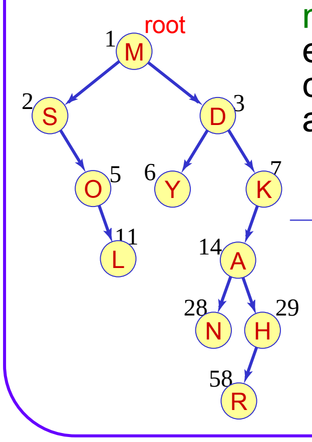
Example 1:



	1	2	3	4	5	6	7	8	9	10	11
Array A:	M	S	D	A	O	Y	K	N	H	R	L

Binary Tree Representation: Using Arrays

Example 2:



For a non-complete tree, there can be much wasted space in the array. In this example, the array contains the same characters as before but requires 58 array positions for storage

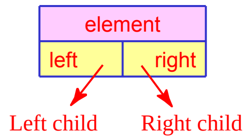
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A[i]	M	S	D		O	Y	K				L			A					

20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
								N	H										

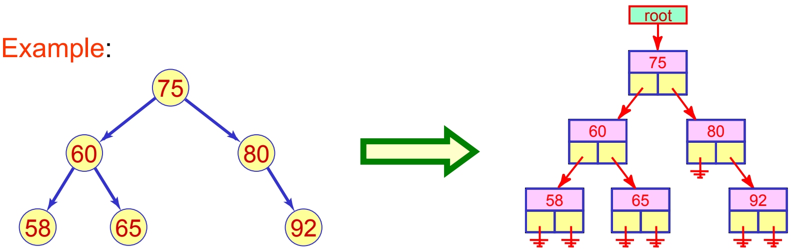
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	…
																		R	…

Binary Tree Representation: Using a Linked Structure

- Each node of the tree can be represented by three fields in a structure of the form shown here:
- This structure will not allow easy access to the parent of a node, but it is adequate for most applications.
- A pointer to the root node is also needed.



- Example:



The Binary Tree Node: A Java Implementation

```
public  class BTNnode<E>  {
   protected E element;
   protected BTNode<E> leftLink;
   protected BTNode<E> rightLink;

   // CONSTRUCTOR
   public BTNode(E el, BTNode<E> left, BTNode<E> right)  {
      element = el;
      leftLink = left;
      rightLink = right;
   }
```

The Binary Tree Node: A Java Implementation

```
// MUTATOR METHODS:
public void setElement (E newElement)   { element = newElement; }
public void setLeft (BTNode<E> newLeft)   { leftLink = newLeft; }
public void setRight (BTNode<E> newRight)   { rightLink = newRight; }

// OBSERVER METHODS:
public E getElement( )   { return element; }
public BTNode<E> getLeft ( )   { return leftLink; }
public BTNode<E> getRight ( )   { return rightLink; }
public boolean isLeaf ( )   {
    return (leftLink == null) && (rightLink == null);
}
}
```

Binary Tree Operations: Tree Construction -- Recursive

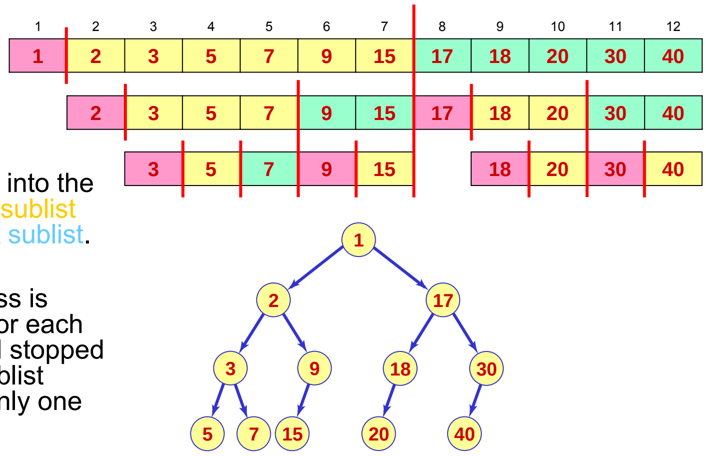
- Given the number of nodes n, a minimal height binary tree can be constructed from a sequence of n nodes as follows:
  1. Use the first node for the root
  2. Construct the left subtree with nl = n / 2 nodes
  3. Construct the right subtree with nr = n – nl – 1 nodes

  Example:

  Given n=12 and the following sequence, construct a minimal height binary tree:

1	2	3	4	5	6	7	8	9	10	11	12
1	**2**	**3**	**5**	**7**	**9**	**15**	**17**	**18**	**20**	**30**	**40**

Binary Tree Operations:
Tree Construction – Example (cont.)



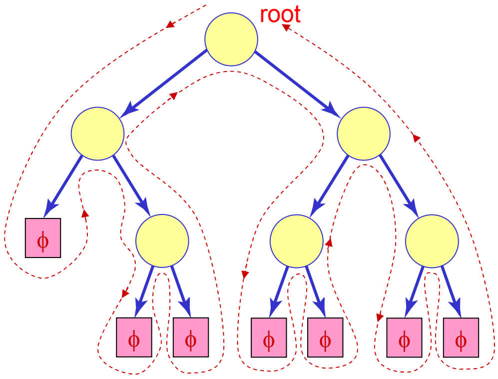
- The list is partitioned into the root, a left sublist and a right sublist.
- The process is repeated for each sublist and stopped when a sublist contains only one item

Binary Tree Operations:
Tree Traversal

- A full traversal of a binary tree produces a linear order for the information contained in the tree, which may be helpful.
- To ***Process*** a node is to perform some simple operation on it, such as print, update, …, etc.
- To ***Traverse*** a finite collection of objects is to process each object in the collection exactly once.

Binary Tree Operations: Tree Traversal

- Trace the path from the root node, through the tree and back to the root
- Each node is visited three times
- Processing a node could be done during any of the three visits.
- This will give three possibilities for traversing the tree.



Binary Tree Operations: Tree Traversal

- A tree can be traversed in three different orders:
  - In-order traversal:

    Each node is processed after all nodes in its left subtree but before any node in its right subtree
  - Pre-order traversal:

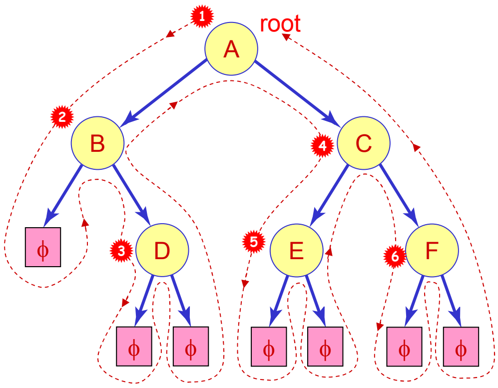
    Each node is processed before any node in either of its subtrees
  - Post-order traversal:

    Each node is processed after all nodes in both of its subtrees

Binary Tree Operations: Tree Traversal

Example:

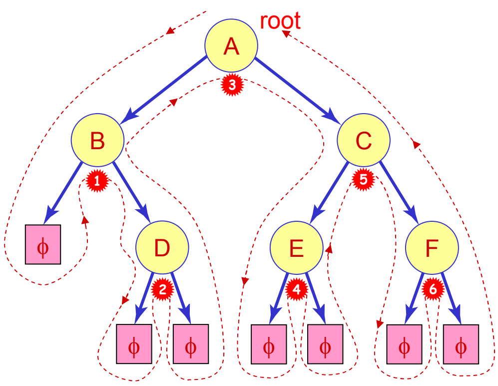
- Traverse the tree in:
  - *Pre-order*:
    A, B, D, C, E, F.



Binary Tree Operations: Tree Traversal

Example:

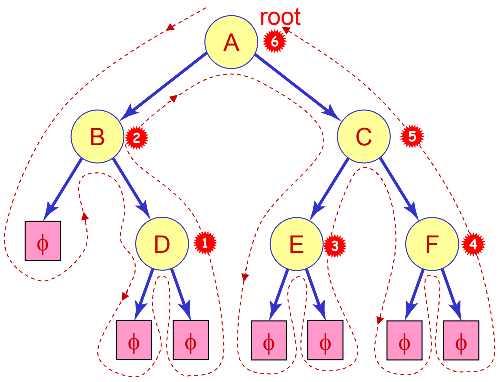
- Traverse the tree in:
  - Pre-order:
    A, B, D, C, E, F.
  - *In-order*:
    B, D, A, E, C, F.



Binary Tree Operations: Tree Traversal

Example:

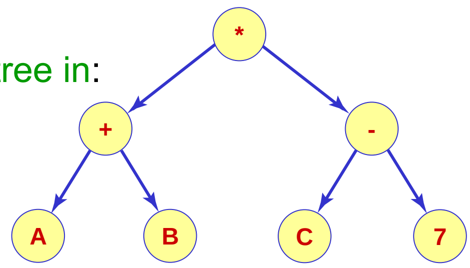
- Traverse the tree in:
  - Pre-order:
    A, B, D, C, E, F.
  - In-order:
    B, D, A, E, C, F.
  - *Post-order*:
    D, B, E, F, C, A.



Binary Tree Operations: Tree Traversal

Example 2:

- Traverse the expression tree in:
  - Pre-order ➔ Prefix expression
    * + A B – C 7
  - In-order ➔ Infix expression
    (A + B) * (C – 7)
  - Post-order ➔ Postfix expression
    A B + C 7 - *



Expression Tree for: (A+B)*(C-7)

Mapping the Binary Tree Traversal Operation: preorder(f, p) – $O(n)$

- Precondition:
  - p is a pointer to a node in a binary tree (or null to indicate the empty tree).
- Postcondition:
  - If p is not null, then the function f has been applied to the element of p and all of its descendants, using a pre-order traversal.
- Note:
  - A Node may be a BTNode<E> or any binary tree node type.
  - A Process may be a class implementing some function f that can be called to process the element in the node.

Mapping the Binary Tree Traversal Operation: preorder(f, p) – $O(n)$

```
public static <E>
   void preorder(Process<E> proc, BTNode<E> p)  {
   if (p != null)  {
      proc.f (p.getElement( ));          // Process the node
      preorder(proc, p.getLeft( ));      // Traverse the left subtree
      preorder(proc, p.getRight( ));     // Traverse the right subtree
    }
 }
```

Mapping the Binary Tree Traversal Operation: inorder(f, p) – *O*(*n*)

```
public static <E>
   void inorder(Process<E> proc, BTNode<E> p)  {
   if (p != null)  {
       inorder(proc, p.getLeft( ));        // Traverse the left subtree
       proc.f (p.getElement( ));           // Process the node
       inorder(proc, p.getRight( ));       // Traverse the right subtree
    }
 }
```

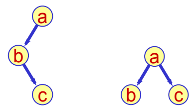
Mapping the Binary Tree Traversal Operation: postorder(f, p) – *O*(*n*)

```
public static <E>
   void postorder(Process<E> proc, BTNode<E> p)  {
   if (p != null)  {
       postorder(proc, p.getLeft( ));     // Traverse the left subtree
       postorder(proc, p.getRight( ));  // Traverse the right subtree
       proc.f (p.getElement( ));           // Process the node
    }
 }
```

Notes on the Traversal of Binary Trees

- Notes:
  - The above functions are recursive.
  - A non-recursive function must use a stack to keep the pointers
  - Given any one traversal output sequence, it is not possible to reconstruct the tree

    Example:

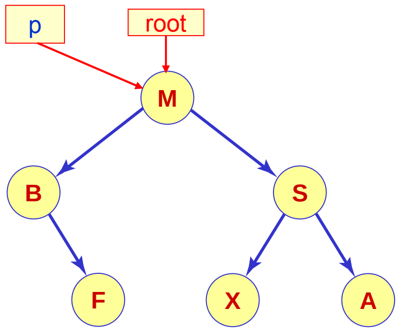


The preorder of both trees is: a b c

A Non-Recursive Preorder Traversal Algorithm Using a Stack

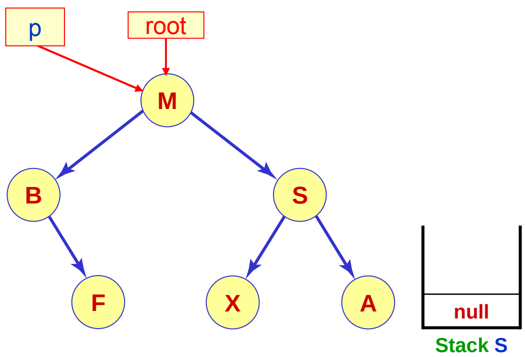
```
public static <E>
void preorder (Process<E> proc, BTNode<E> p)  {
// This function uses a Stack generic class assumed to be defined

     Stack<BTNode<E>> S = new Stack<BTNode<E>>();

     S.push(null);
     while (p != null)  {
          proc.f (p.getElement());                    // Process the node

          if (p.getRight( ) != null)
               S.push(p.getRight( ));                 // Save pointer of the right branch
          if (p.getLeft( ) != null)
               p = p.getLeft( );                      // Go to the left branch
          else p = S.pop();
     }
}
```

Non-Recursive Preorder Traversal Example: preorder (print, root);



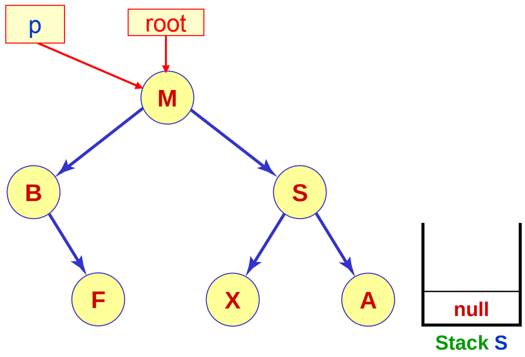
Non-Recursive Preorder Traversal Example: preorder (print, root);

```
Stack<BTNode<E>> S = new Stack<BTNode<E>>();
S.push(null);
```



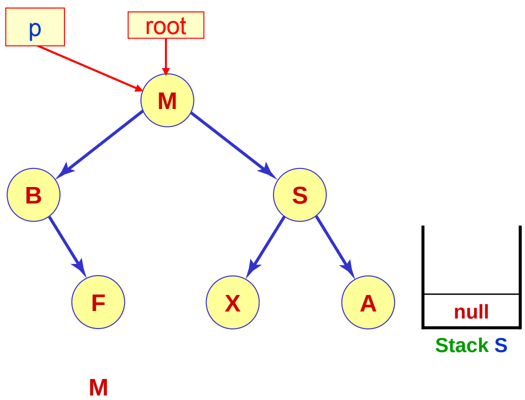
Non-Recursive Preorder Traversal Example: preorder (print, root);

```
while (p != null) {

   proc.f (p.getElement());

   if (p.getRight( ) != null)
      S.push(p.getRight( ));

   if (p.getLeft( ) != null)
      p = p.getLleft( );
   else p = S.pop();
}
```



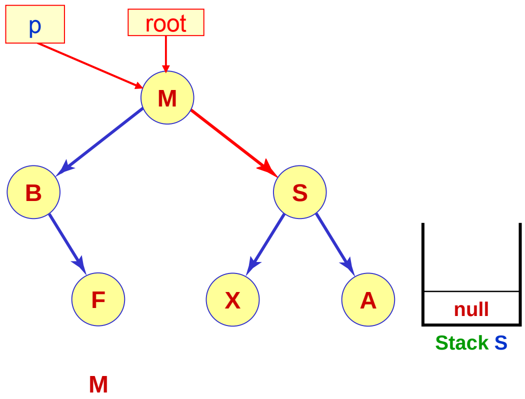
Non-Recursive Preorder Traversal Example: preorder (print, root);

```
while (p != null) {

   proc.f (p.getElement());

   if (p.getRight( ) != null)
      S.push(p.getRight( ));

   if (p.getLeft( ) != null)
      p = p.getLleft( );
   else p = S.pop();
}
```



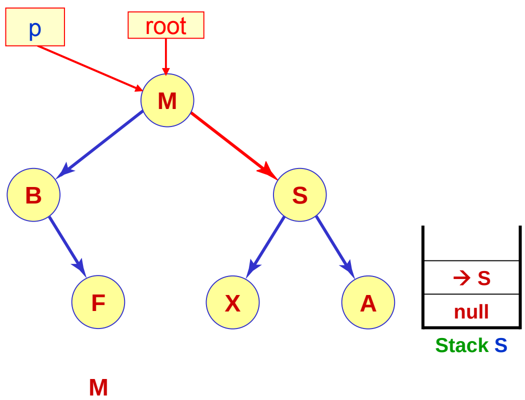
Non-Recursive Preorder Traversal Example: preorder (print, root);

```
while (p != null)  {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
        S.push(p.getRight( ));

    if (p.getLeft( ) != null)
        p = p.getLleft( );
    else p = S.pop();
}
```



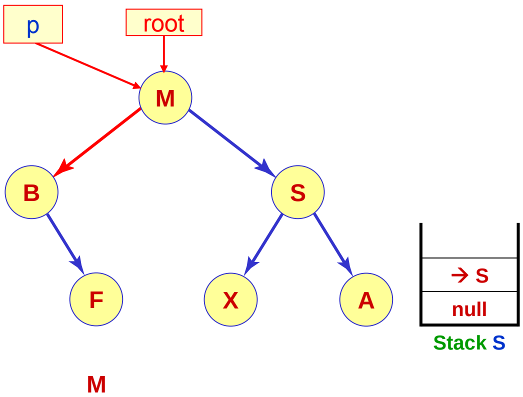
Non-Recursive Preorder Traversal Example: preorder (print, root);


```
while (p != null)  {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
        S.push(p.getRight( ));

    if (p.getLeft( ) != null)
        p = p.getLleft( );
    else p = S.pop();
}
```



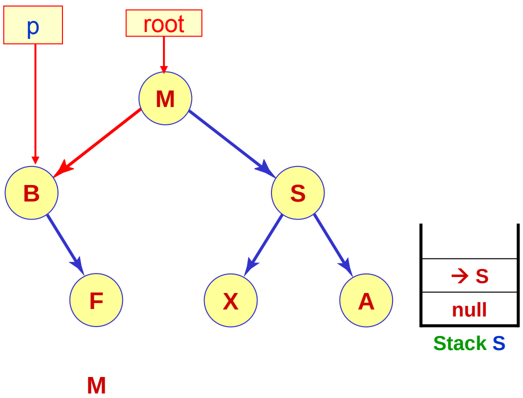
Non-Recursive Preorder Traversal Example: preorder (print, root);

```
while (p != null) {

   proc.f (p.getElement());

   if (p.getRight( ) != null)
      S.push(p.getRight( ));

   if (p.getLeft( ) != null)
      p = p.getLleft( );
   else p = S.pop();
}
```



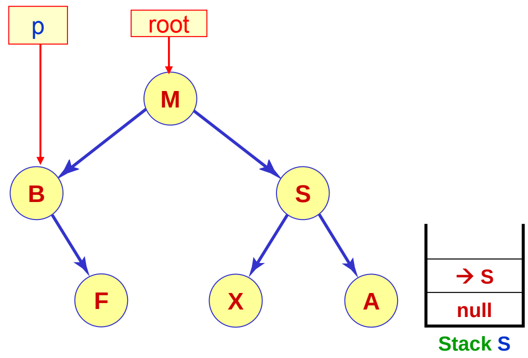
Non-Recursive Preorder Traversal Example: preorder (print, root);

```
while (p != null) {

   proc.f (p.getElement());

   if (p.getRight( ) != null)
      S.push(p.getRight( ));

   if (p.getLeft( ) != null)
      p = p.getLleft( );
   else p = S.pop();
}
```



Non-Recursive Preorder Traversal Example: preorder (print, root);

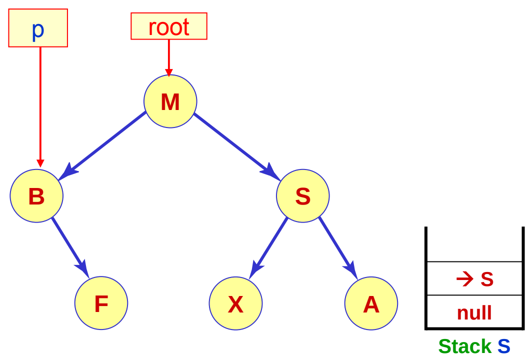
```
while (p != null) {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
        S.push(p.getRight( ));

    if (p.getLeft( ) != null)
        p = p.getLleft( );
    else p = S.pop();
}
```



M

Non-Recursive Preorder Traversal Example: preorder (print, root);

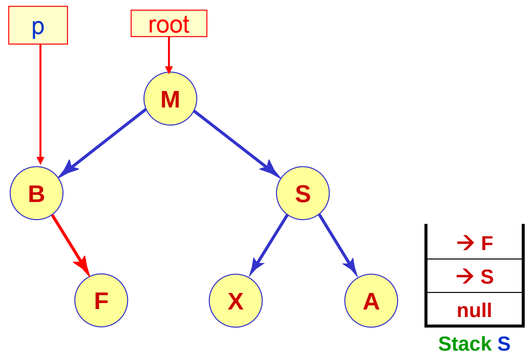
```
while (p != null) {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
        S.push(p.getRight( ));

    if (p.getLeft( ) != null)
        p = p.getLleft( );
    else p = S.pop();
}
```



M B

Non-Recursive Preorder Traversal Example: preorder (print, root);

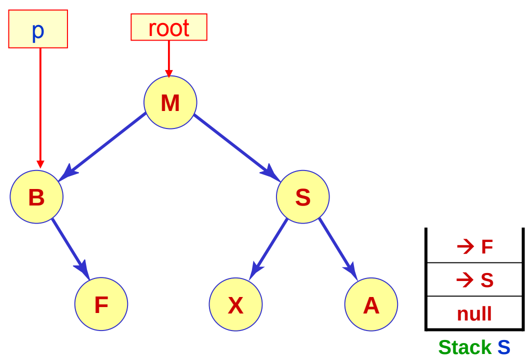
```
while (p != null) {

   proc.f (p.getElement());

   if (p.getRight( ) != null)
      S.push(p.getRight( ));

   if (p.getLeft( ) != null)
      p = p.getLleft( );
   else p = S.pop();
}
```



M B

Non-Recursive Preorder Traversal Example: preorder (print, root);

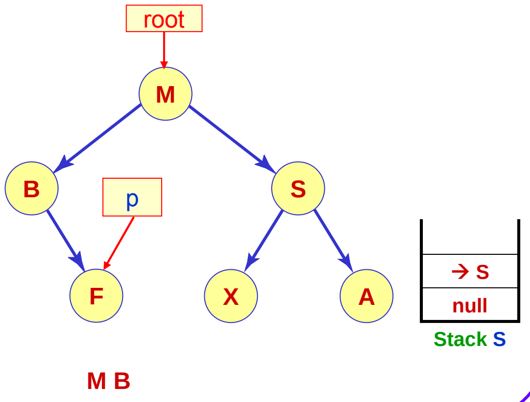
```
while (p != null) {

   proc.f (p.getElement());

   if (p.getRight( ) != null)
      S.push(p.getRight( ));

   if (p.getLeft( ) != null)
      p = p.getLleft( );
   else p = S.pop();
}
```



M B

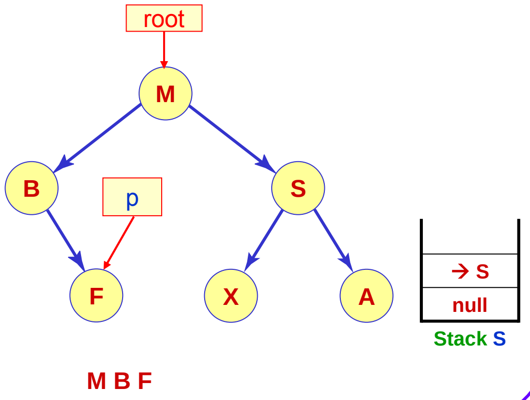
Non-Recursive Preorder Traversal Example: preorder (print, root);

```
while (p != null) {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
        S.push(p.getRight( ));

    if (p.getLeft( ) != null)
        p = p.getLleft( );
    else p = S.pop();
}
```



Non-Recursive Preorder Traversal Example: preorder (print, root);

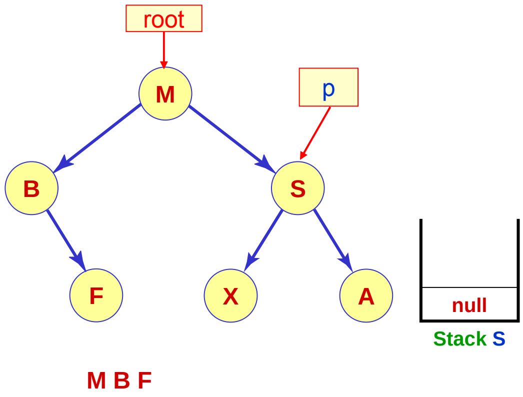
```
while (p != null) {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
        S.push(p.getRight( ));

    if (p.getLeft( ) != null)
        p = p.getLleft( );
    else p = S.pop();
}
```



Non-Recursive Preorder Traversal

Example: preorder (print, root);

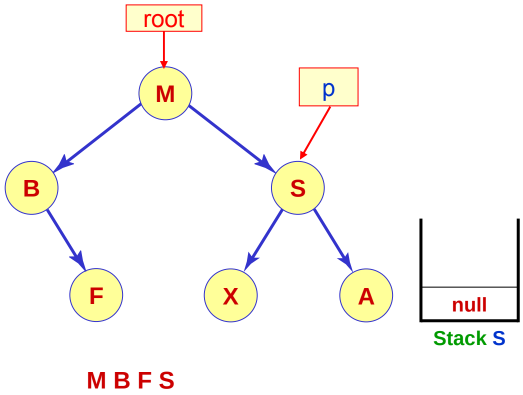
```
while (p != null)  {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
        S.push(p.getRight( ));

    if (p.getLeft( ) != null)
        p = p.getLleft( );
    else p = S.pop();
}
```



Non-Recursive Preorder Traversal

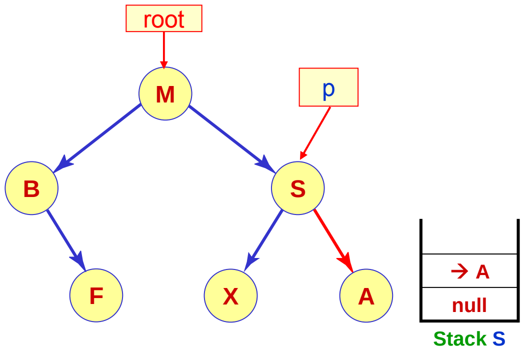
Example: preorder (print, root);

```
while (p != null)  {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
        S.push(p.getRight( ));

    if (p.getLeft( ) != null)
        p = p.getLleft( );
    else p = S.pop();
}
```



Non-Recursive Preorder Traversal
Example: preorder (print, root);

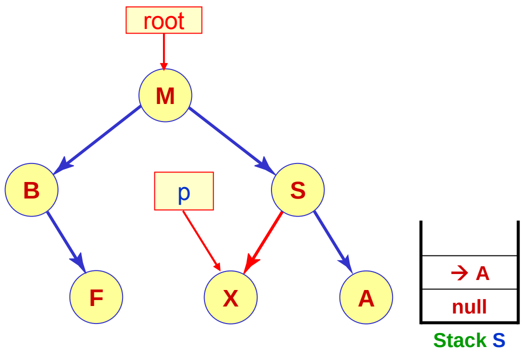
```
while (p != null) {

   proc.f (p.getElement());

   if (p.getRight( ) != null)
      S.push(p.getRight( ));

   if (p.getLeft( ) != null)
      p = p.getLleft( );
   else p = S.pop();
}
```



M B F S

Non-Recursive Preorder Traversal
Example: preorder (print, root);

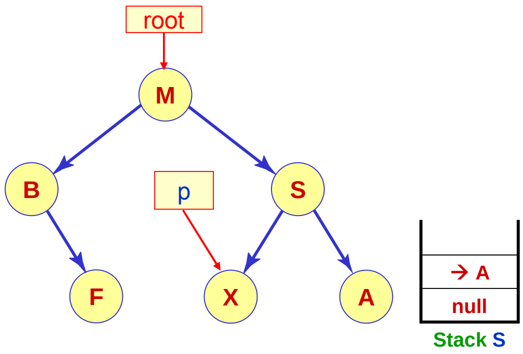
```
while (p != null) {

   proc.f (p.getElement());

   if (p.getRight( ) != null)
      S.push(p.getRight( ));

   if (p.getLeft( ) != null)
      p = p.getLleft( );
   else p = S.pop();
}
```



M B F S

Non-Recursive Preorder Traversal
Example: preorder (print, root);

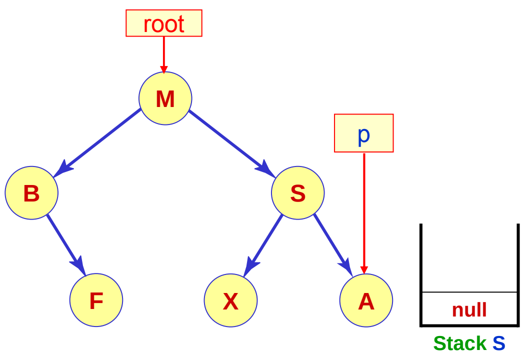
```
while (p != null) {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
       S.push(p.getRight( ));

    if (p.getLeft( ) != null)
       p = p.getLleft( );
    else p = S.pop();
}
```



M B F S X

Non-Recursive Preorder Traversal
Example: preorder (print, root);

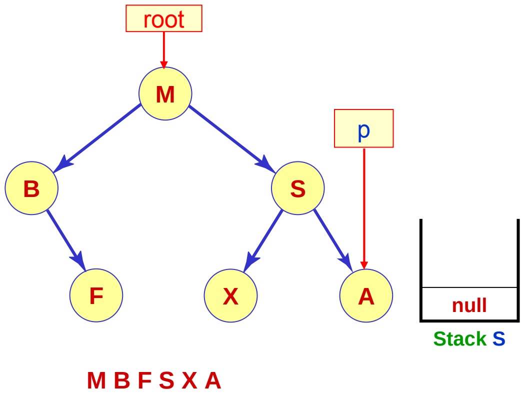
```
while (p != null) {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
       S.push(p.getRight( ));

    if (p.getLeft( ) != null)
       p = p.getLleft( );
    else p = S.pop();
}
```



M B F S X

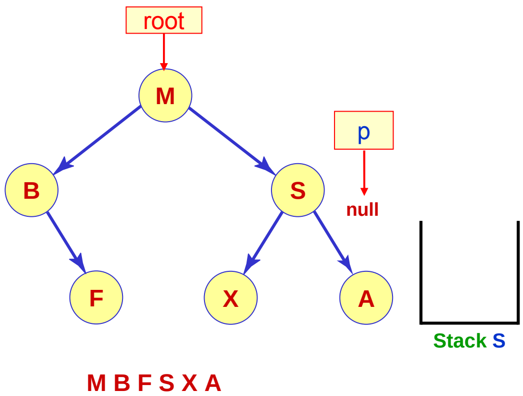
Non-Recursive Preorder Traversal Example: preorder (print, root);

```
while (p != null) {

   proc.f (p.getElement());

   if (p.getRight( ) != null)
      S.push(p.getRight( ));

   if (p.getLeft( ) != null)
      p = p.getLleft( );
   else p = S.pop();
}
```



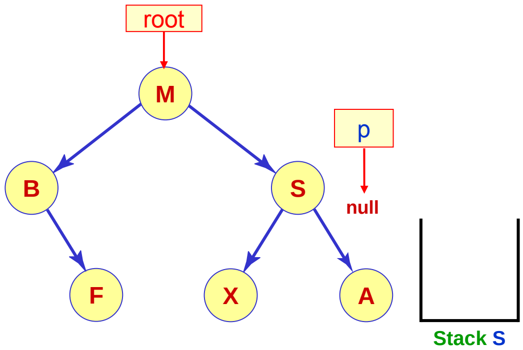
Non-Recursive Preorder Traversal Example: preorder (print, root);

```
while (p != null) {

   proc.f (p.getElement());

   if (p.getRight( ) != null)
      S.push(p.getRight( ));

   if (p.getLeft( ) != null)
      p = p.getLleft( );
   else p = S.pop();
}
```



Non-Recursive Preorder Traversal
Example: preorder (print, root);

```
while (p != null) {

    proc.f (p.getElement());

    if (p.getRight( ) != null)
        S.push(p.getRight( ));

    if (p.getLeft( ) != null)
        p = p.getLleft( );
    else p = S.pop();
}
```



M B F S X A

Non-Linear Data Structures

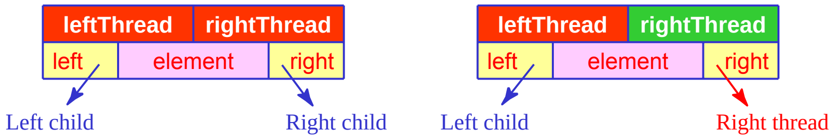
Threaded Binary Trees

What are Threaded Binary Trees?

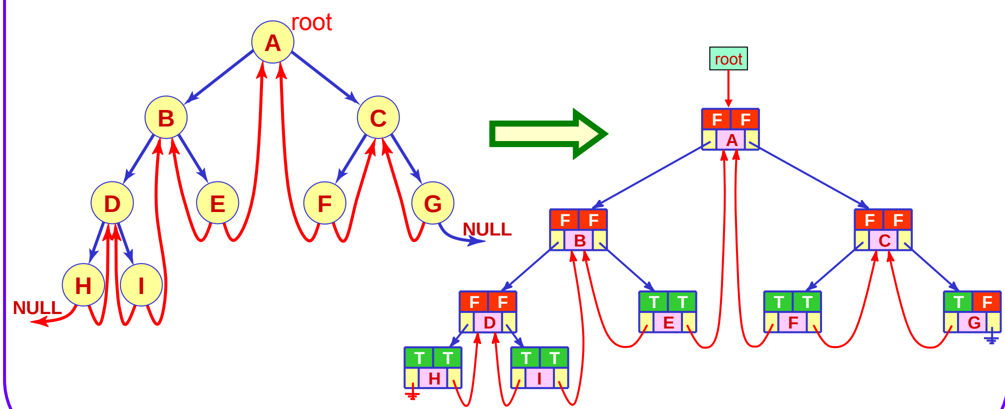
- A Binary Tree with n nodes, has n+1 empty subtrees.
- In a linked representation, this means n+1 null pointers.
- The non-recursive inorder traversal algorithm is complicated and uses a stack.
- It can be written more efficiently if the null pointers are replaced with pointers to the inorder successors and/or predecessors of the nodes
- These new pointers are called ***threads***
  - Always replace a left null pointer with predecessor threads
  - Always replace a right null pointer with successor threads
- Such a tree is called a ***threaded binary tree***

Threaded Binary Trees Node Representation

- Use an extra boolean field for each link field in the node to indicate whether the link is a normal tree pointer or a thread pointer.
- If the pointer is a thread, its boolean field is true, otherwise it is false.



Threaded Binary Trees Representation Example



A Non-Recursive Inorder Traversal Algorithm Using a Threaded Binary Tree

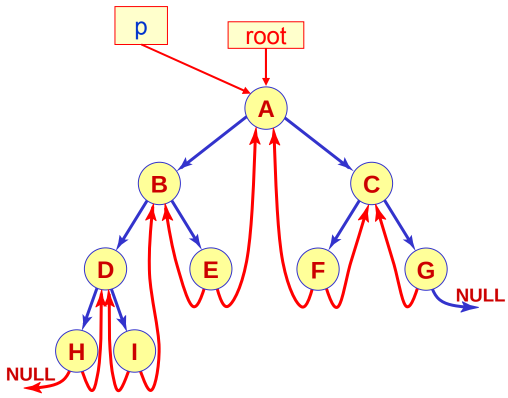
```
public static <E>
    void tinorder (Process<E> proc, BTNode<E> p)  {
    // This function uses a threaded binary tree structure

    while (p != null)  {
        while (!p.leftThread())
            p = p.getLeft( );                // Go all the way to the leftmost node

        proc.f(p.getElement());              // Process the leftmost node

        while (p.rightThread())  {           // Now follow the right threads, if any
            p = p.getRight( );
            proc.f(p.getElement());          // Process the nodes
        }
        p = p.getRight( );                   // Go to the right branch
    }
}
```

Threaded Binary Trees
Non-Recursive Inorder Traversal

Example:

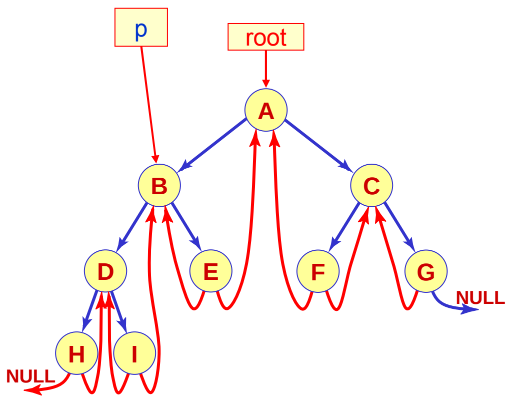
Traverse the threaded binary tree shown, inorder.

tinorder (print, root);



Non-Recursive Inorder Traversal
Example: tinorder (print, root);

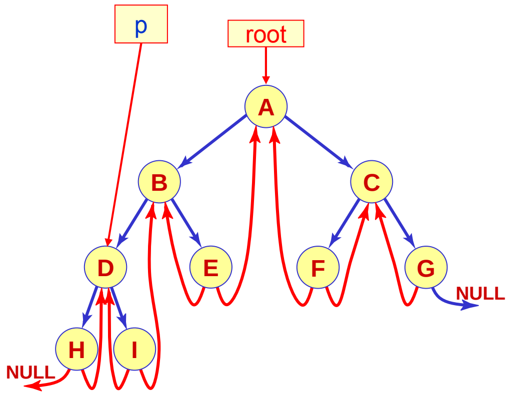
```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



Non-Recursive Inorder Traversal

Example: tinorder (print, root);

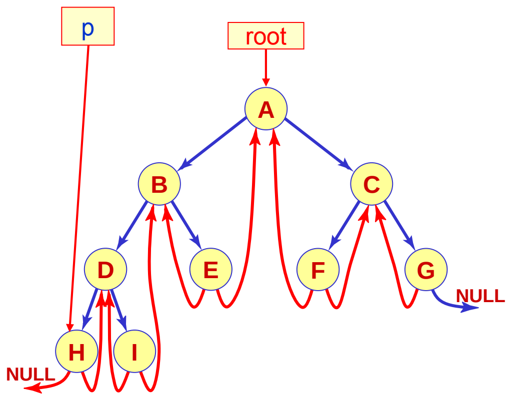
```
while (p != null)  {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread())  {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



Non-Recursive Inorder Traversal

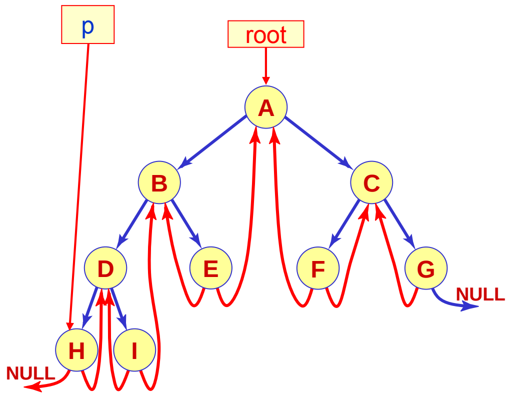
Example: tinorder (print, root);

```
while (p != null)  {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread())  {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



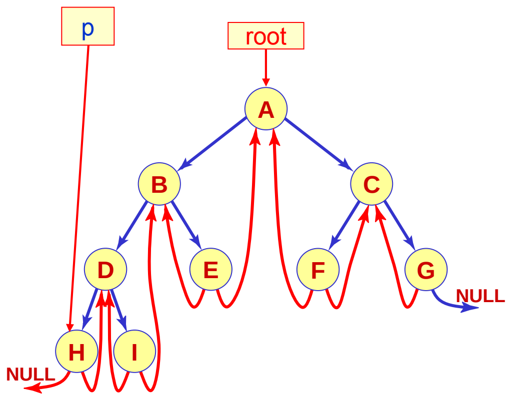
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```

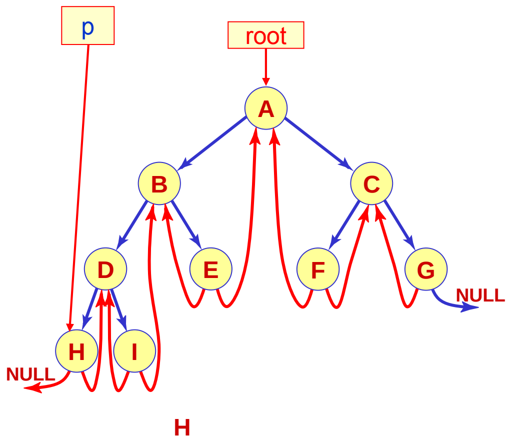


H

Non-Recursive Inorder Traversal

Example: tinorder (print, root);

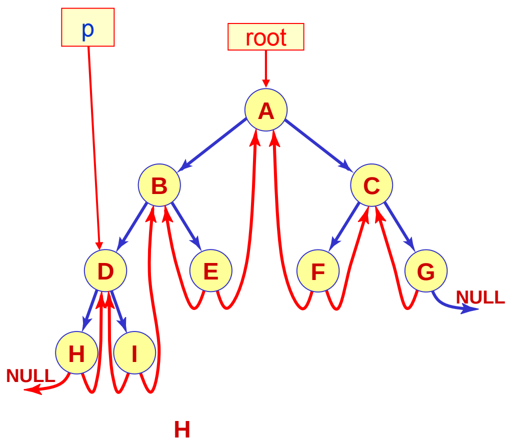
```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



Non-Recursive Inorder Traversal

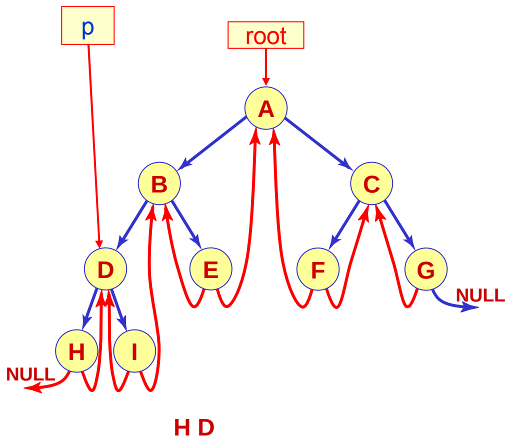
Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



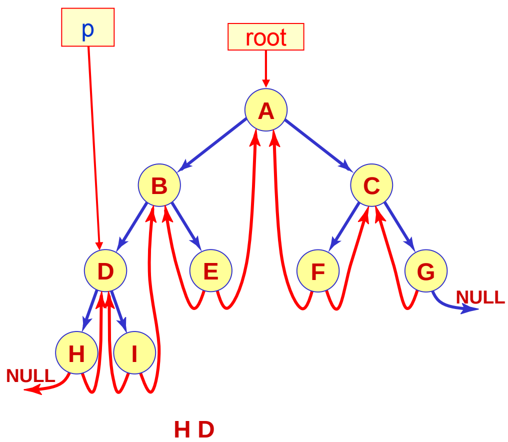
Non-Recursive Inorder Traversal
Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



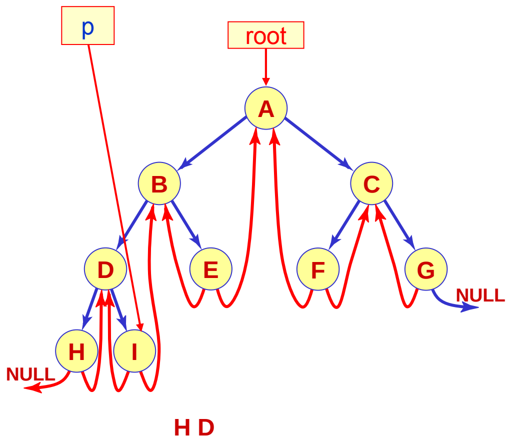
Non-Recursive Inorder Traversal
Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```

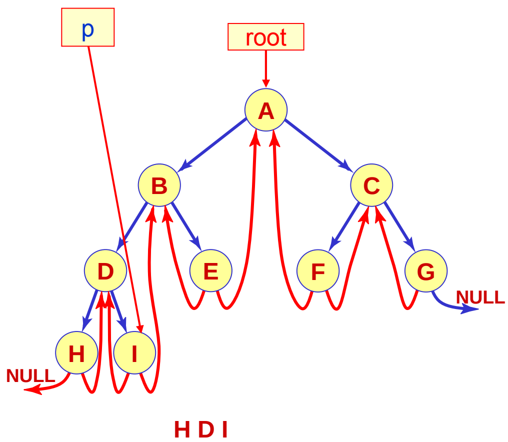
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



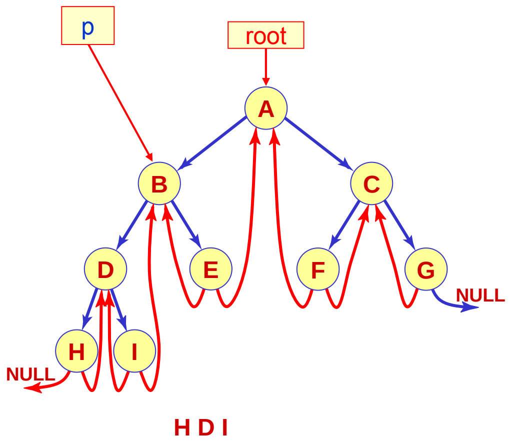
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



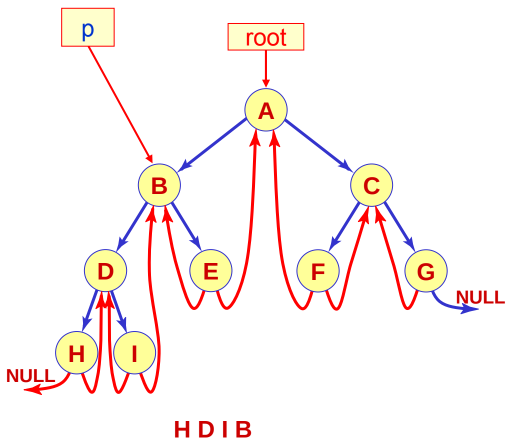
Non-Recursive Inorder Traversal
Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



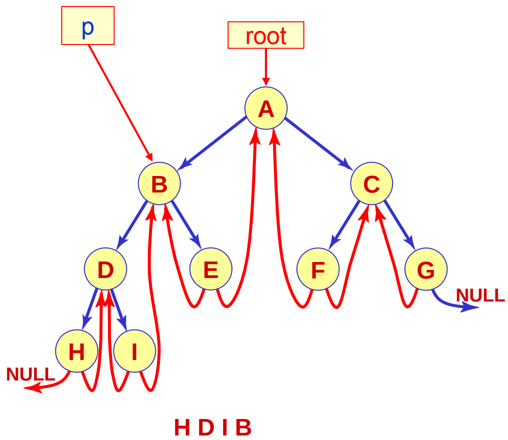
Non-Recursive Inorder Traversal
Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



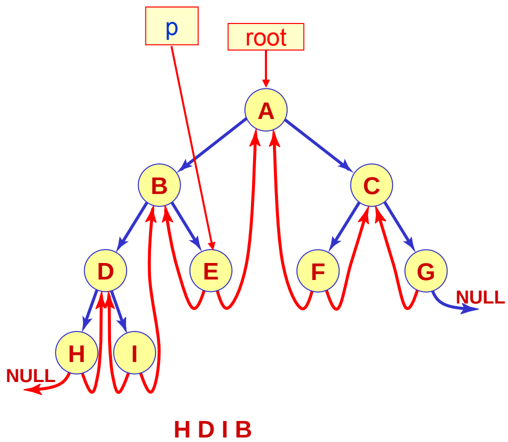
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



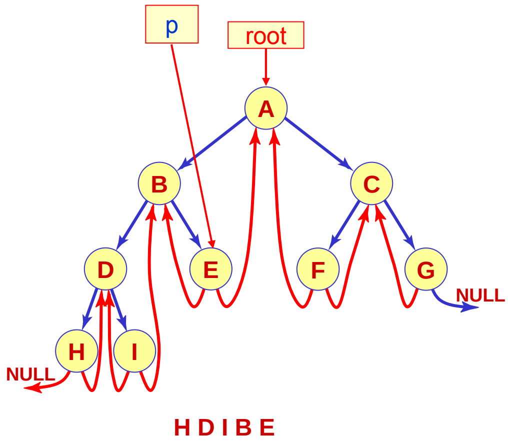
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



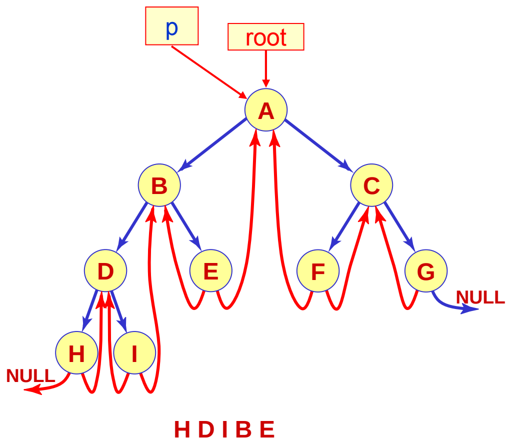
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



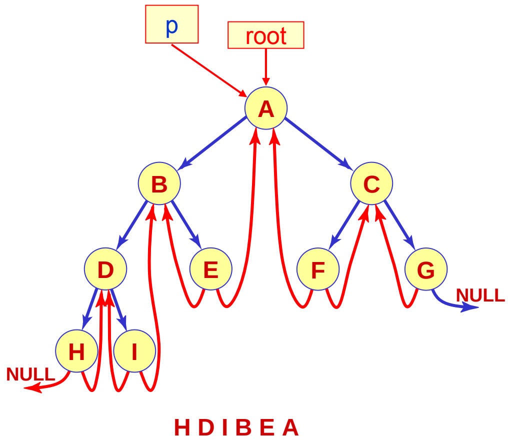
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



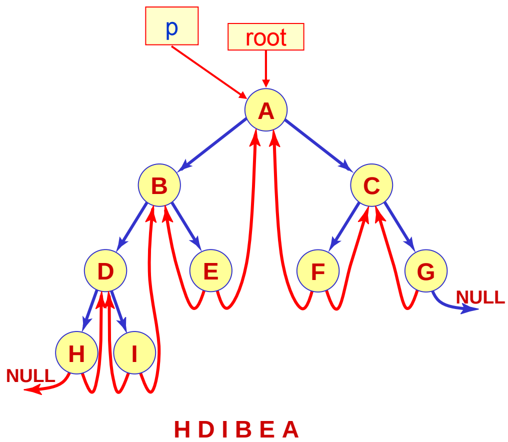
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null)  {
    while (!p.leftThread())
        p = p.getLeft( );

    proc.f (p.getElement());

    while (p.rightThread())  {
        p = p.getRight( );
        proc.f (p.getElement());
    }
    p = p.getRight( );
}
```



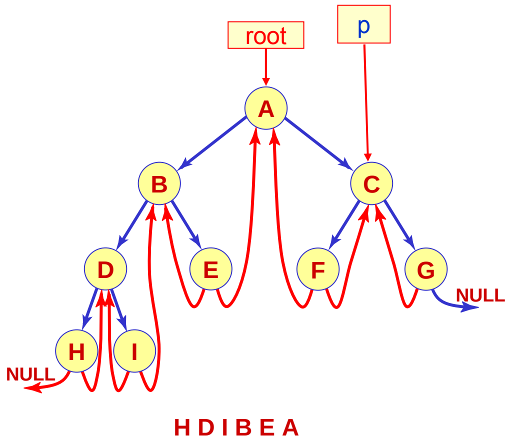
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null)  {
    while (!p.leftThread())
        p = p.getLeft( );

    proc.f (p.getElement());

    while (p.rightThread())  {
        p = p.getRight( );
        proc.f (p.getElement());
    }
    p = p.getRight( );
}
```



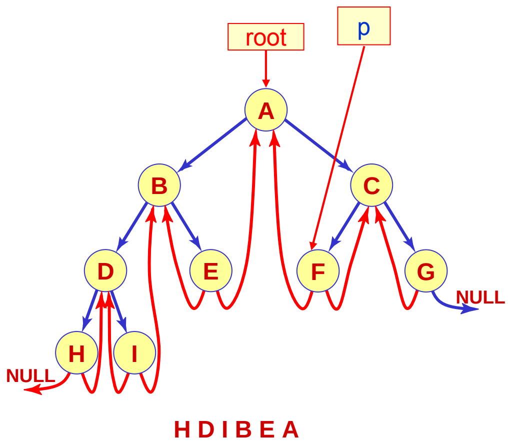
Non-Recursive Inorder Traversal
Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



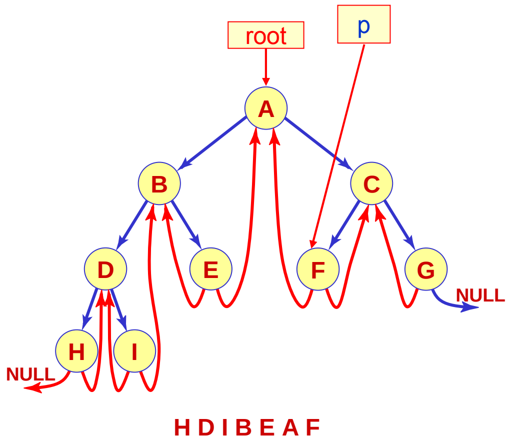
Non-Recursive Inorder Traversal
Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



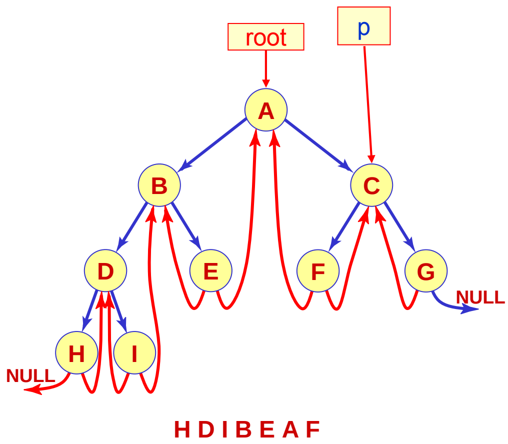
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



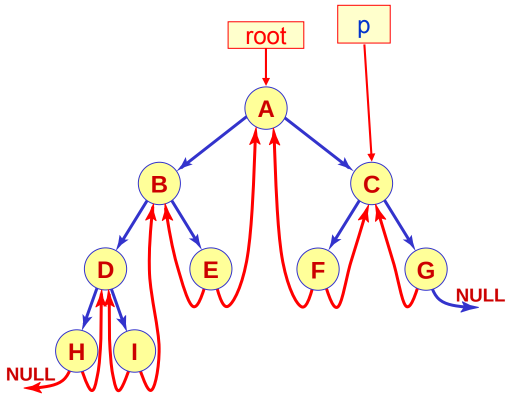
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



Non-Recursive Inorder Traversal Example: tinorder (print, root);

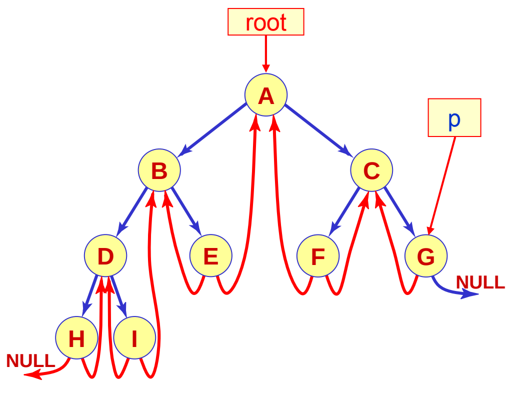
```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



H D I B E A F C

Non-Recursive Inorder Traversal Example: tinorder (print, root);

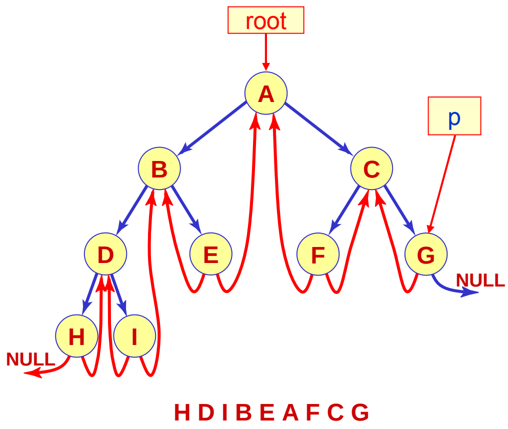
```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



H D I B E A F C

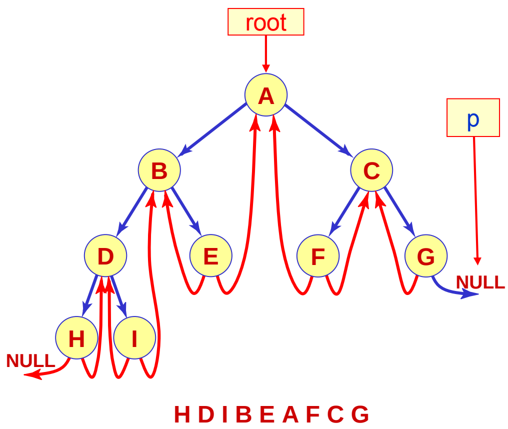
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



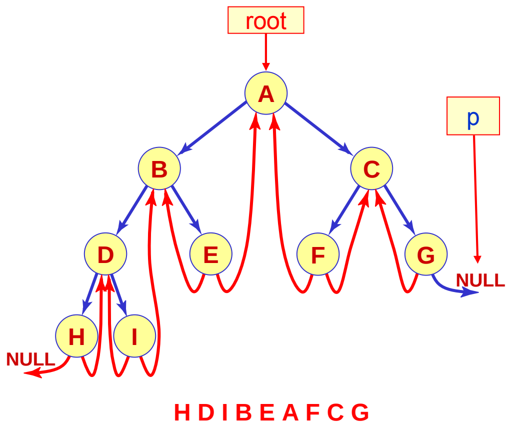
Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



Non-Recursive Inorder Traversal Example: tinorder (print, root);

```
while (p != null) {
   while (!p.leftThread())
      p = p.getLeft( );

   proc.f (p.getElement());

   while (p.rightThread()) {
      p = p.getRight( );
      proc.f (p.getElement());
   }
   p = p.getRight( );
}
```



Notes on Threaded Binary Trees

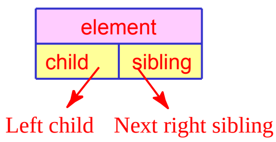
- Traversal is possible, non-recursively, without using a stack
- Traversal can begin at any node in the tree
- Traversal of a tree still takes O(n) time complexity
- Slightly more storage is needed per node for the thread flags
- Slightly more work is needed for constructing the tree, and when inserting or deleting a node.

Non-Linear Data Structures

Back to General Trees

General Tree Representation Using Binary Trees

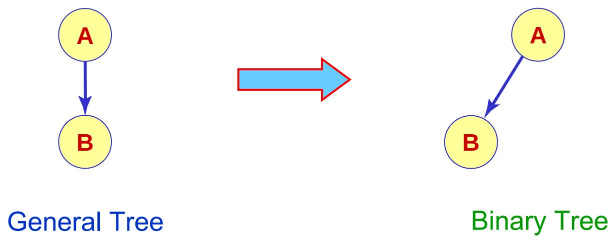
- Using the important property of a general tree, that the order of the children of a node is not relevant, any child could be a left child.
- A general tree node can be represented as shown:
  - One link to its left child
  - One link to its next right sibling
- Since we have two link fields in each node, that means a binary tree!



- The sibling pointer of the root node is not used, and so it can be used to point at the root of another tree
- Thus a binary tree can be used to represent a general ***forest***

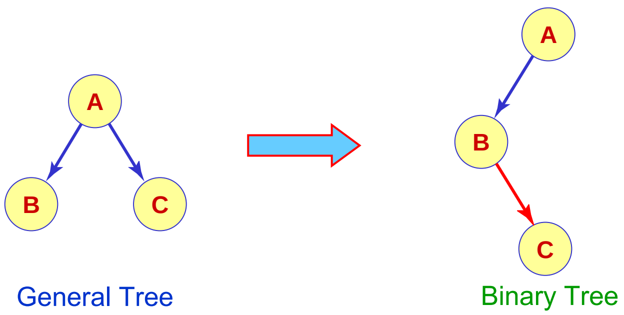
General Tree Representation Using Binary Trees

- Example1:



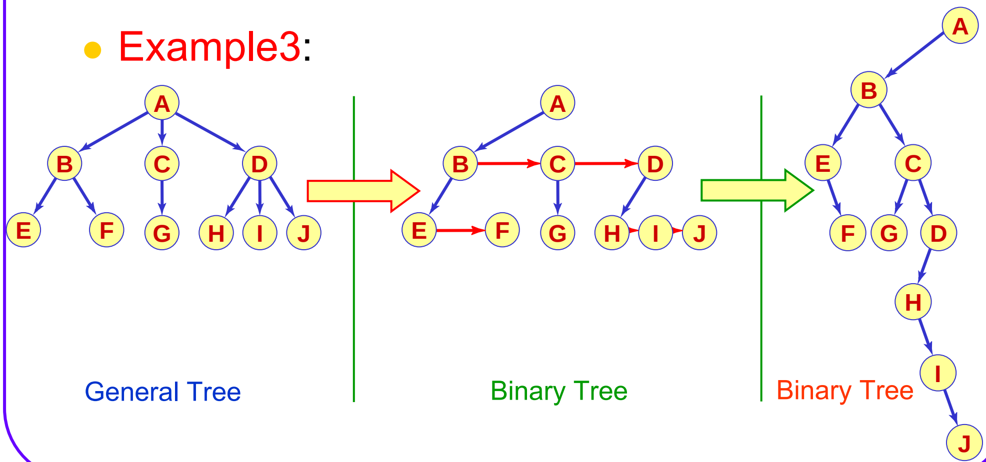
General Tree Representation Using Binary Trees

- Example2:



General Tree Representation Using Binary Trees

- Example3:



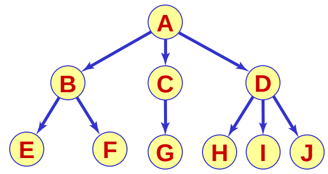
Traversal of General Trees

- A forest can be traversed similar to a binary tree:
  - Pre-order traversal:
    1. ***Process*** the root of the first tree in the forest
    2. Traverse the subtrees of the first tree in tree pre-order
    3. Traverse the remaining trees of the forest in tree pre-order
  - In-order traversal:
    1. Traverse the subtrees of the first tree in tree in-order
    2. ***Process*** the root of the first tree in the forest
    3. Traverse the remaining trees of the forest in tree in-order
  - Post-order traversal:
    1. Traverse the subtrees of the first tree in tree post-order
    2. Traverse the remaining trees of the forest in tree post-order
    3. ***Process*** the root of the first tree in the forest

Traversal of General Trees Example

Traverse the shown tree:

1. Pre-order:
   A B E F C G D H I J
2. In-order:
   E B F A G C H D I J
3. Post-order:
   E F B G C H I J D A

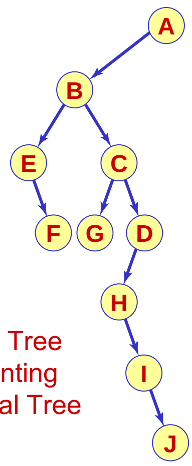


General Tree

Traversal of General Trees Example

Traverse the shown tree and compare with above:

1. Pre-order:
   A B E F C G D H I J
2. In-order:
   E F B G C H I J D A
3. Post-order:
   F E G J I H D C B A



A Binary Tree Representing A General Tree

Applications of General Trees

- General Expression Trees
- Decision Trees
- Game Trees
- Set Representation