Data Structures and Algorithms

Lecture 8 – Binary Trees Traversals

Miao Zhang

ADT of Binary Tree

```
template <class T>
class BinaryTree  {
private:
    BinaryTreeNode<T>* root;    // root node of the binary tree

public:
    BinaryTree() {root = NULL;};                    // constructor
  ~BinaryTree() {DeleteBinaryTree(root);};      // destructor
    bool isEmpty() const;         // determine if it is empty
    BinaryTreeNode<T>* Root(){return root;};
                  // return the root
    ……
};
```

Common Binary Tree Operations

- Determine the height
- Determine the number of nodes
- Make a copy
- Determine if two binary trees are identical
- Display the binary tree
- Delete a tree
- If it is an expression tree, evaluate the expression
- If it is an expression tree, obtain the parenthesized form of the expression

Binary Tree Traversal

- Many binary tree operations are done by performing a traversal of the binary tree
- In a traversal, each element of the binary tree is **visited exactly once**
- During the visit of an element, all actions (make a copy, display, evaluate the operator, etc.) with respect to this element are taken

Traversals

- For a tree with n nodes there are n! traversals
- Of course most of those are hard to program
- There are two very common traversals
  - Breadth First
  - Depth First

Binary Tree Traversal Methods

- **Level order (Breadth first traversal)**
  - The tree is processed by levels. So first all nodes on level i are processed from left to right before the first node of level i+1 is visited
- **Preorder (Depth first traversal)**
  - The root of the subtree is processed first before going into the left then right subtree (root, left, right).
- **Inorder (Depth first traversal)**
  - After the complete processing of the left subtree the root is processed followed by the processing of the complete right subtree (left, root, right).
- **Postorder (Depth first traversal)**
  - The root is processed only after the complete processing of the left and right subtree (left, right, root).

Breadth First

- In a breadth first traversal all of the nodes on a given level are visited and then all of the nodes on the next level are visited.
- Usually in a left to right fashion
- The Recursive Approach of BFS Traversal

The Recursive Approach of BFS Traversal

1. First calculate the height of the tree h using the function **height(root)**.
2. Run a loop from **l = 0 to h - 1** and access each level in the tree.

Inside the loop, the function **processCurrentLevel(root, l)** is used to visit and process nodes at the current level l.

Pseudo-code:

```
class TreeNode
{
    int data
    TreeNode left
    TreeNode right
}

void recursiveLevelOrder(TreeNode root)
{
    int h = height(root)
    for (int l = 0; l < h; l = l + 1)
        processCurrentLevel(root, l)
}
```

```
void processCurrentLevel(TreeNode root, int l)
{
    if (root == NULL)
        return
    if (l == 0)
        process(root->data)
    else if (l > 0)
    {
        processCurrentLevel(root->left, l - 1)
        processCurrentLevel(root->right, l - 1)
    }
}
```

Time and space complexity analysis

The worst-case: the skewed tree

- Process CurrentLevel() takes O(n) time for the last level, O(n-1) time for the second last level, and so on. Here n is the number of nodes in the tree.

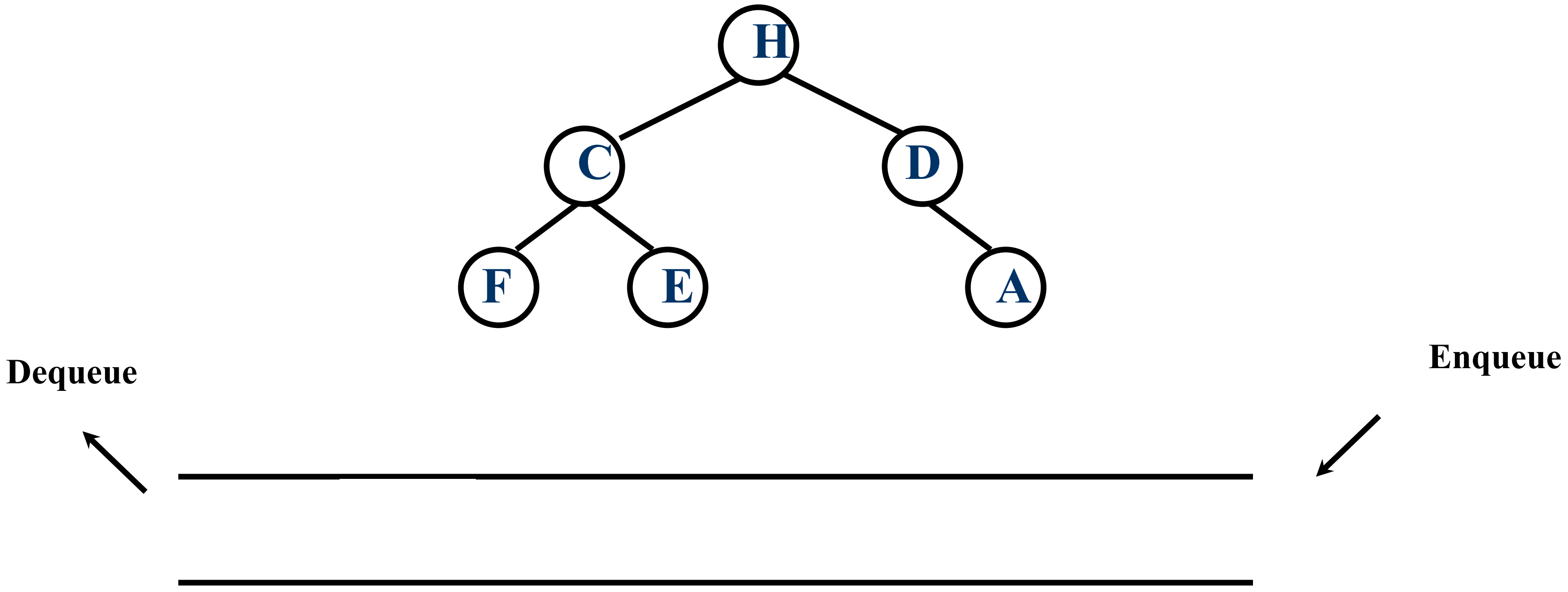
Time complexity = O(n) + O(n-1) + .. + O(1) = O(n + (n-1) + ... + 1) = O(n^2).

Space complexity = O(n) in the worst case.

An efficient approach: BFS traversal using queue

Can we **optimize** the time complexity of the BFS traversal?

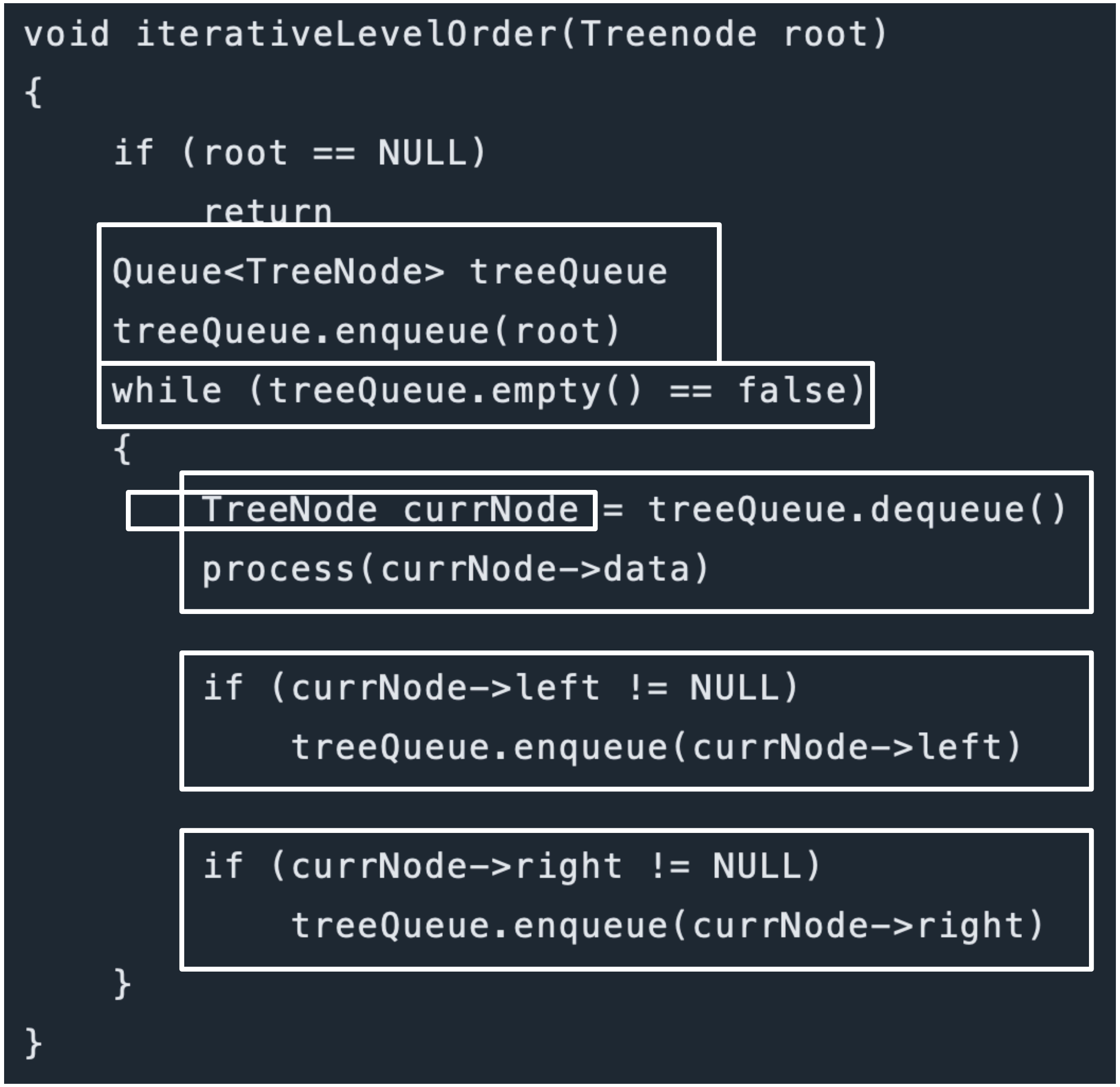
At any given level, the node which will be processed first, the children of that node will be processed first at the next level. This is the First In First Out Order (FIFO Order) of processing nodes!



H C D F E A

Level Order Traversal

Pseudocode of BFS traversal using queue



```
void iterativeLevelOrder(Treenode root)
{
    if (root == NULL)
        return
    Queue<TreeNode> treeQueue
    treeQueue.enqueue(root)
    while (treeQueue.empty() == false)
    {
        TreeNode currNode = treeQueue.dequeue()
        process(currNode->data)

        if (currNode->left != NULL)
            treeQueue.enqueue(currNode->left)

        if (currNode->right != NULL)
            treeQueue.enqueue(currNode->right)
    }
}
```

Time complexity analysis of the BFS traversal

Time complexity analysis of the BFS traversal

•Suppose n number of nodes are given in the input.

•The time complexity of enqueue and dequeue operations = O(1)

•Two queue operations for each node inside the loop: inserting once into the queue and deleting once from the queue. So total queue operations = 2n.

•**Overall time time complexity** = total queue operations * time complexity of each queue operation = 2n * O(1) = O(n)

The space complexity analysis of the BFS traversal

Worst case: when the tree is balanced

space complexity = O(n)

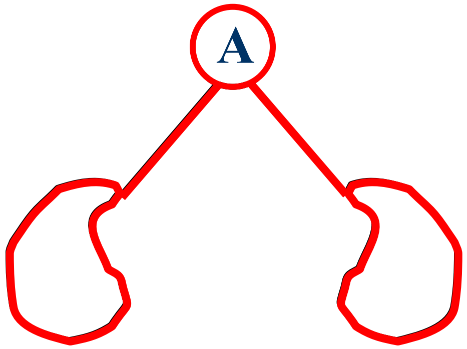
Best case: when the tree is a skewed tree

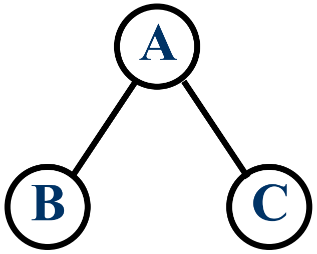
space complexity = O(1)

Depth First

- In a depth first traversal all the nodes on a branch are visited before any others are visited
- There are three common depth first traversals
  - Preorder
  - Postorder
  - Inorder
- Each type has its applications

Preorder traversal



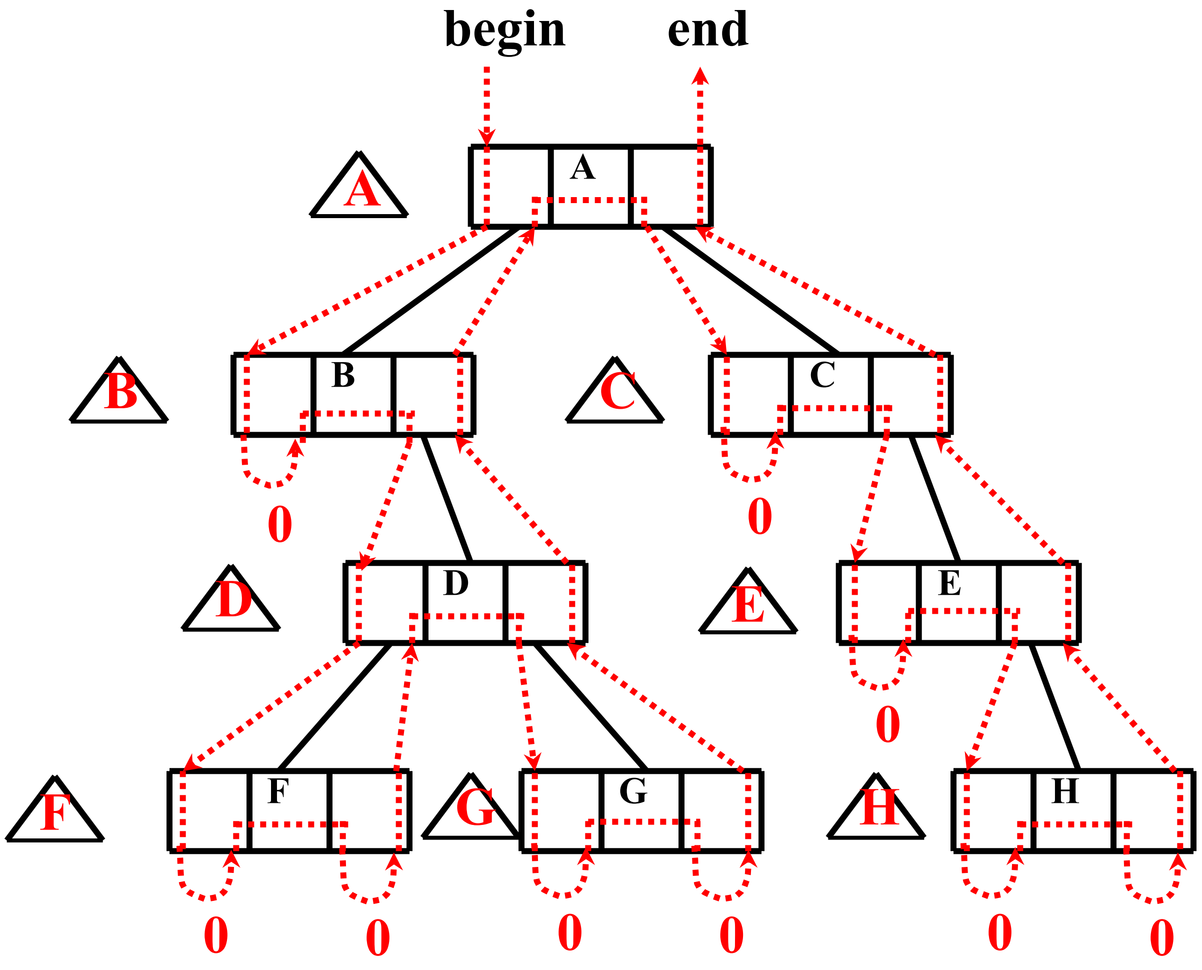


A B C

Preorder Traversals

```
template <class Elem>
void preorder(BinNode<Elem>* root) {
  if (root == NULL) return;  // Empty
  visit(root);  // Perform some action
  preorder(root->left());
  preorder(root->right());
}
```

Preorder Traversals



Pre-order: A B D F G C E H

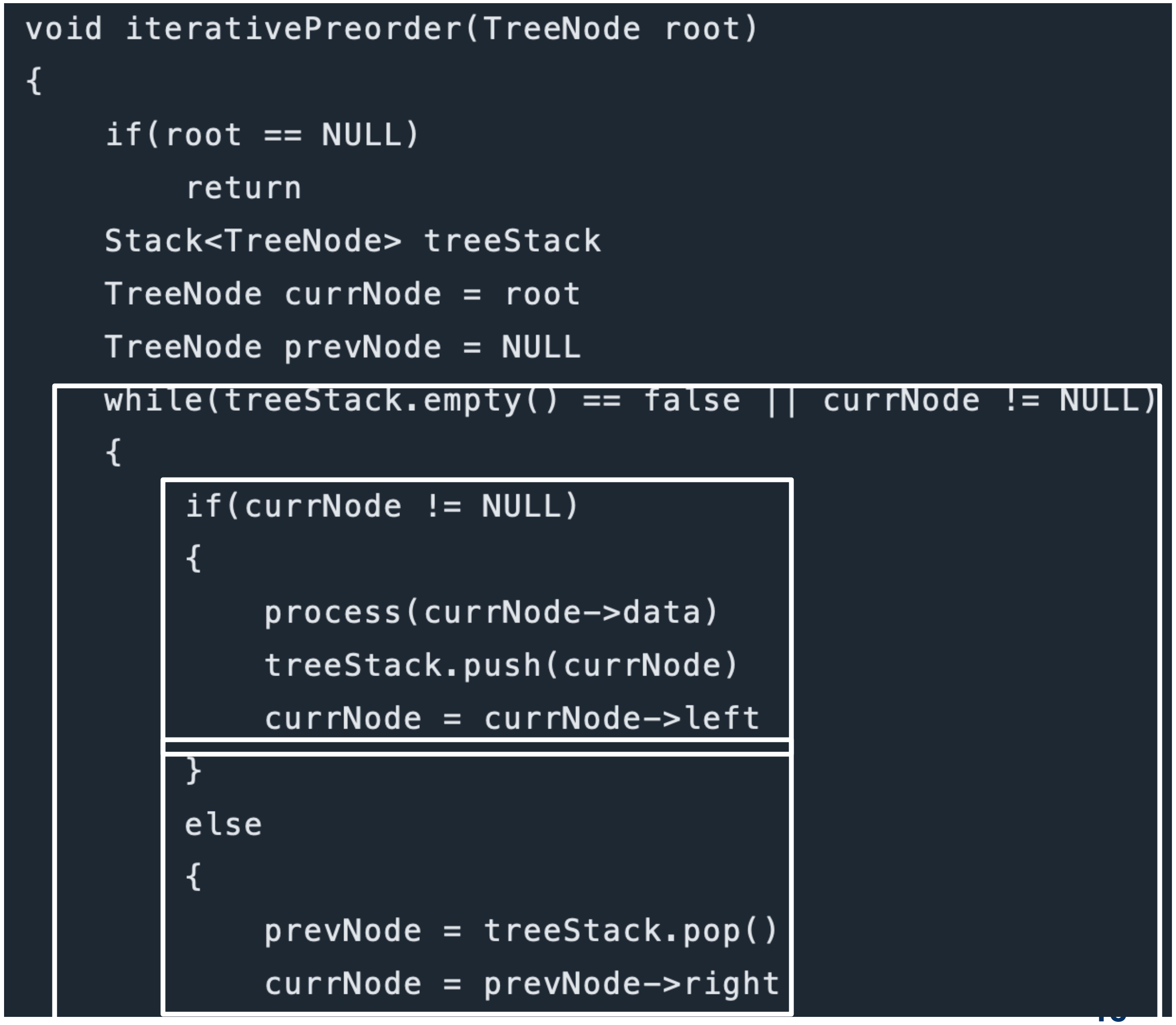
Non-recursive preorder traversal (1)

1. When we visit any node the first time, we process it push the node into the stack (to process the right subtree of that node later), go to the left child.

2. If there's no left child, we grab one node from the top of the stack and go to the right child of that node.

3. Continue the same process for the subtree rooted at the right child of the popped node

Non-recursive preorder traversal (1)

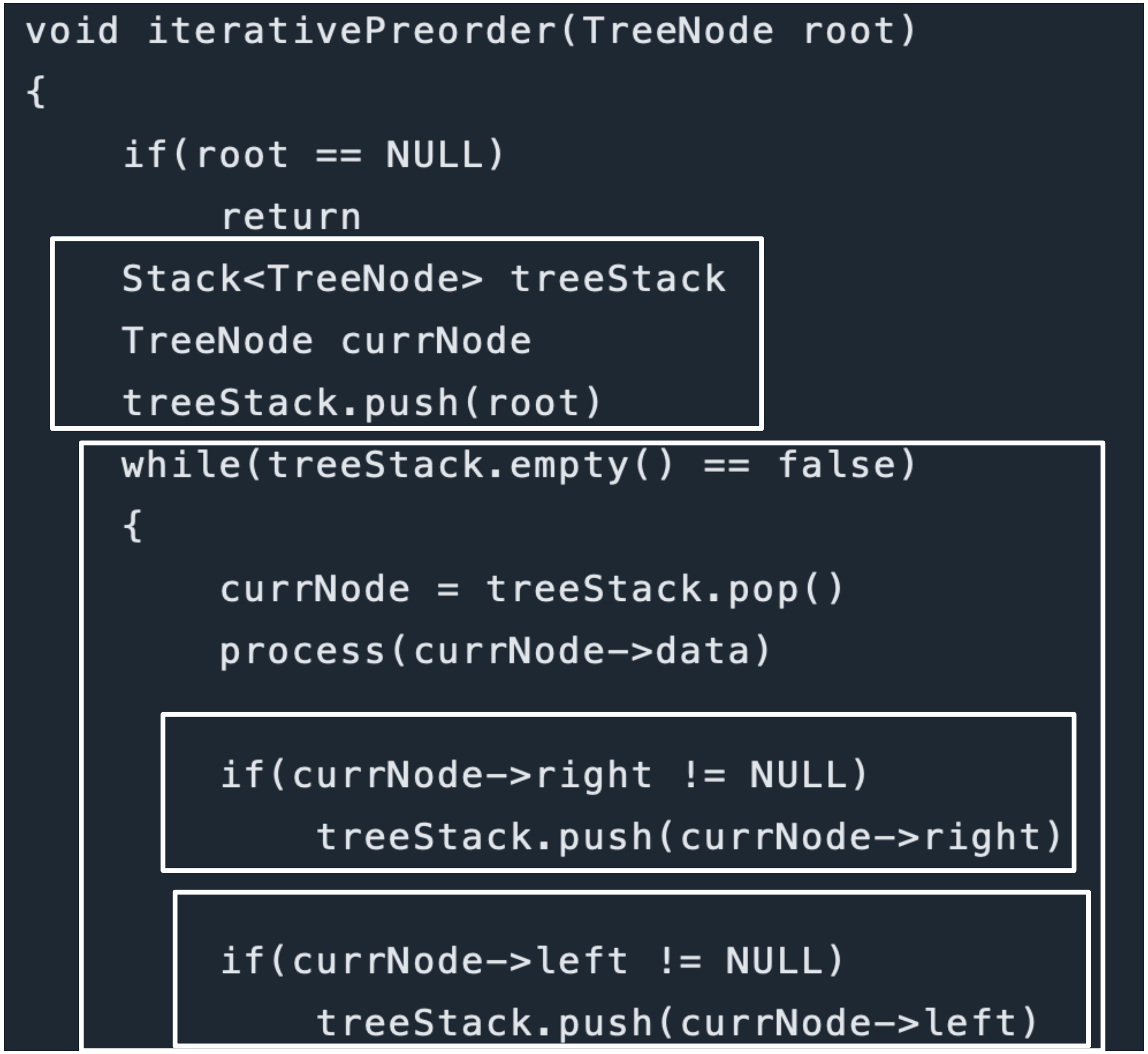


```
void iterativePreorder(TreeNode root)
{
    if(root == NULL)
        return
    Stack<TreeNode> treeStack
    TreeNode currNode = root
    TreeNode prevNode = NULL
    while(treeStack.empty() == false || currNode != NULL)
    {
        if(currNode != NULL)
        {
            process(currNode->data)
            treeStack.push(currNode)
            currNode = currNode->left
        }
        else
        {
            prevNode = treeStack.pop()
            currNode = prevNode->right
```

Non-recursive preorder traversal (2)

1. We create an empty stack **treeStack** and push the root node.
2. We also use a pointer **currNode** to track the current node.
3. Now we run a loop till **treeStack** is not empty.
4. We pop the top node of the stack and assign it with **currNode**. Now we move forward to process the currNode.
5. **if(currNode->right != NULL)**, then we push the right child of the currNode.
6. **if(currNode->left != NULL)**, then we push the left child of the currNode.
7. Now we move to the next iteration of the loop.

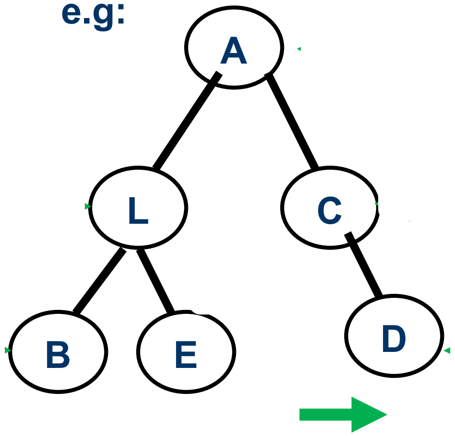
Non-recursive preorder traversal (2)



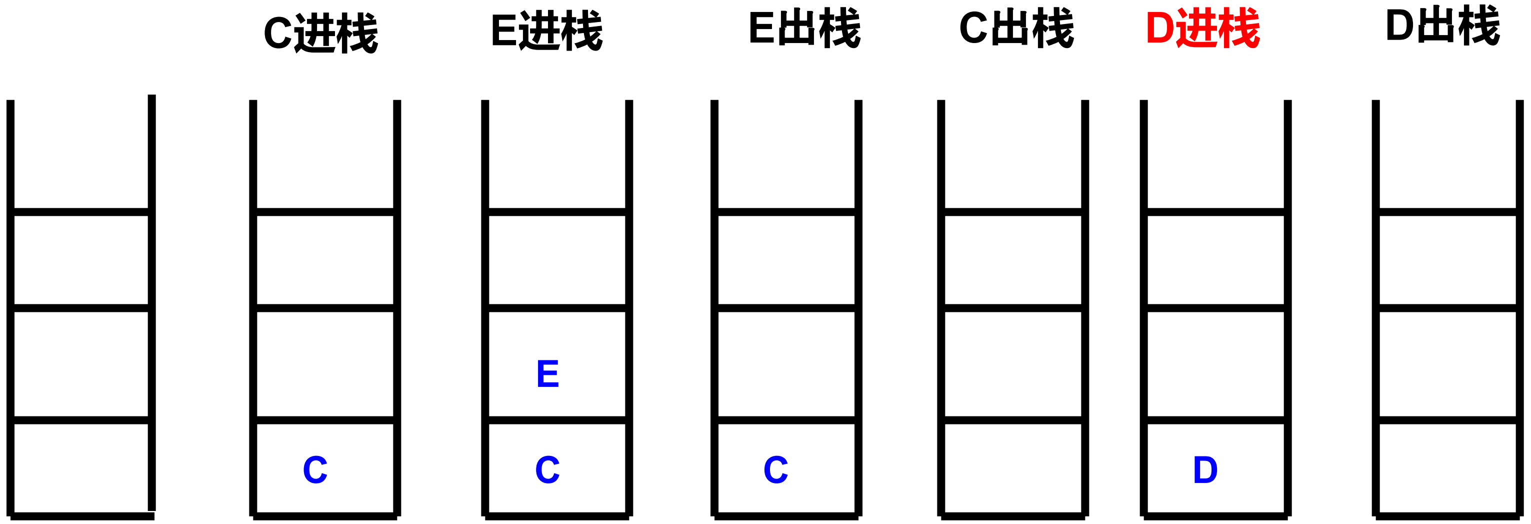
```
void iterativePreorder(TreeNode root)
{
     if(root == NULL)
          return
     Stack<TreeNode> treeStack
     TreeNode currNode
     treeStack.push(root)
     while(treeStack.empty() == false)
     {
          currNode = treeStack.pop()
          process(currNode->data)

          if(currNode->right != NULL)
               treeStack.push(currNode->right)

          if(currNode->left != NULL)
               treeStack.push(currNode->left)
```

Non-recursive preorder traversal (2)

e.g:

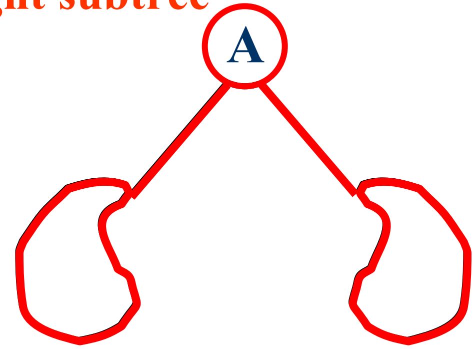


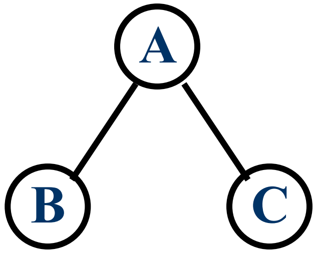
Preorder: A L B E C D



Inorder Traversal

```
template<class T>
void BinaryTree<T>:: InOrder (BinaryTreeNode<T> *root)  {
//  indorder traverse the binary tree and its subtrees

      if (root != NULL) {
              InOrder (root->leftchild()); // indorder traverse the left subtree
           Visit(root->value());          // visit the current node
              InOrder(root->rightchild()); // indorder traverse the right subtree
       }
}
```

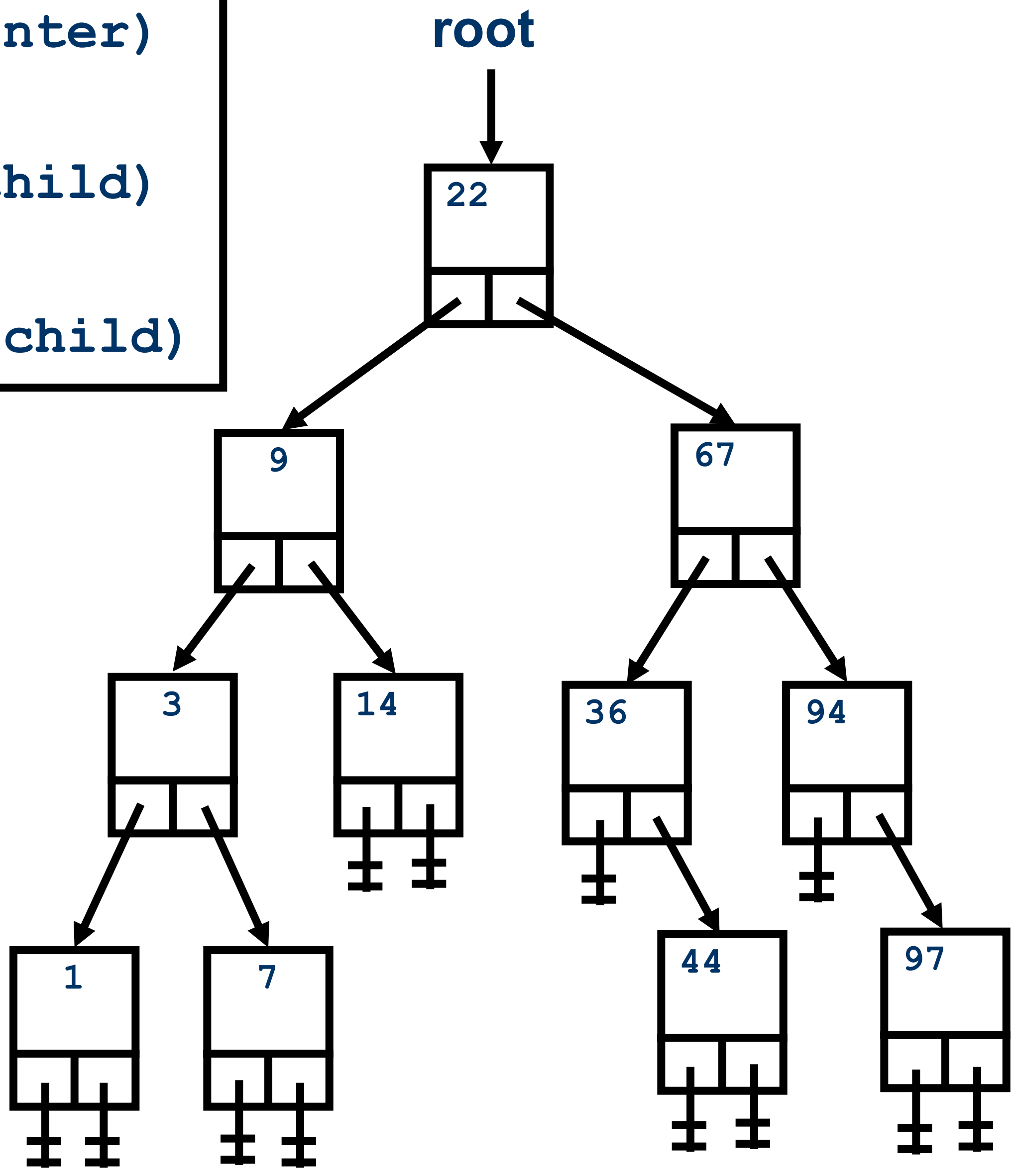




B A C

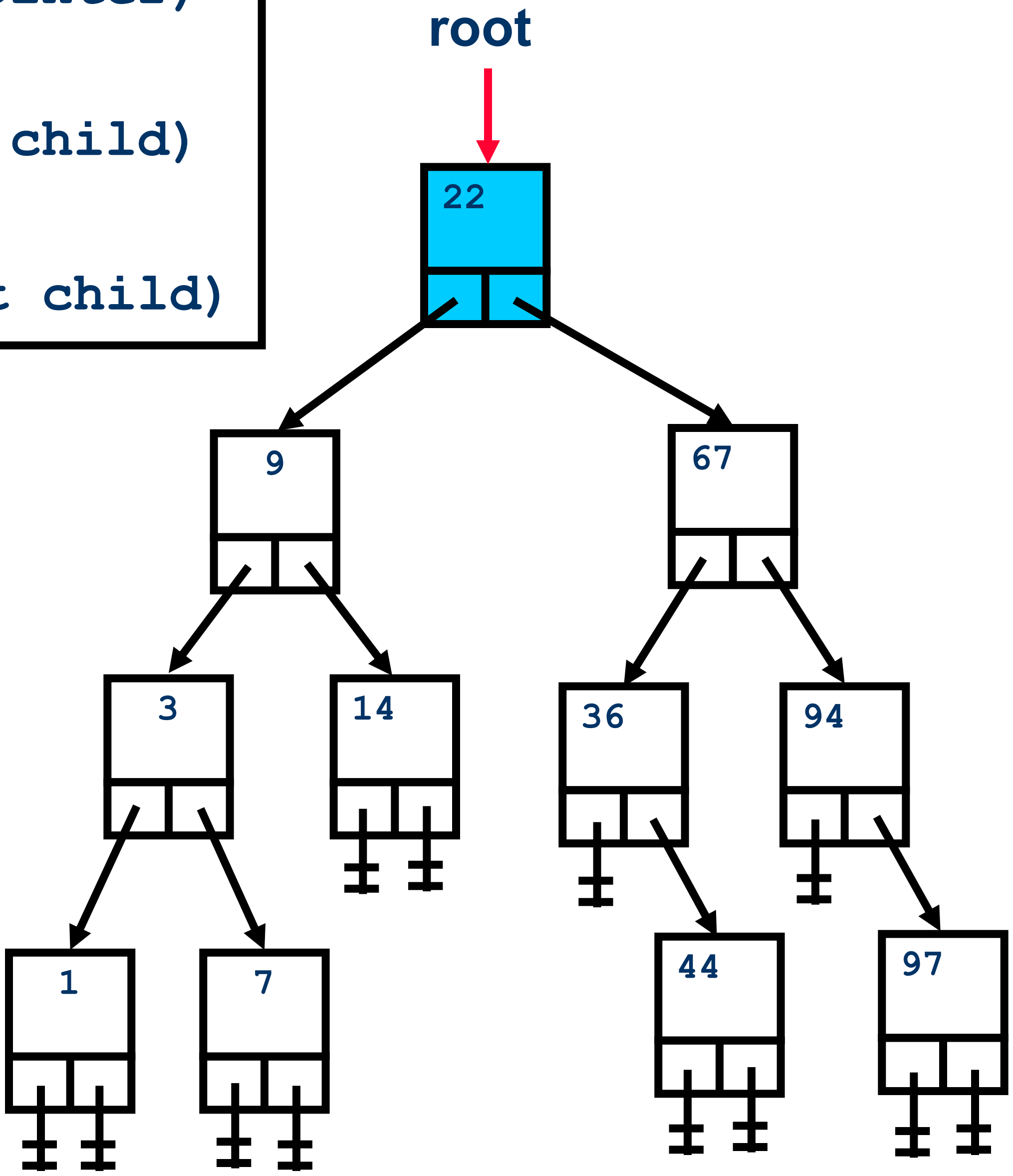
Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
  L  InOrderPrint(left child)
  P  print(data)
  R  InOrderPrint(right child)
```



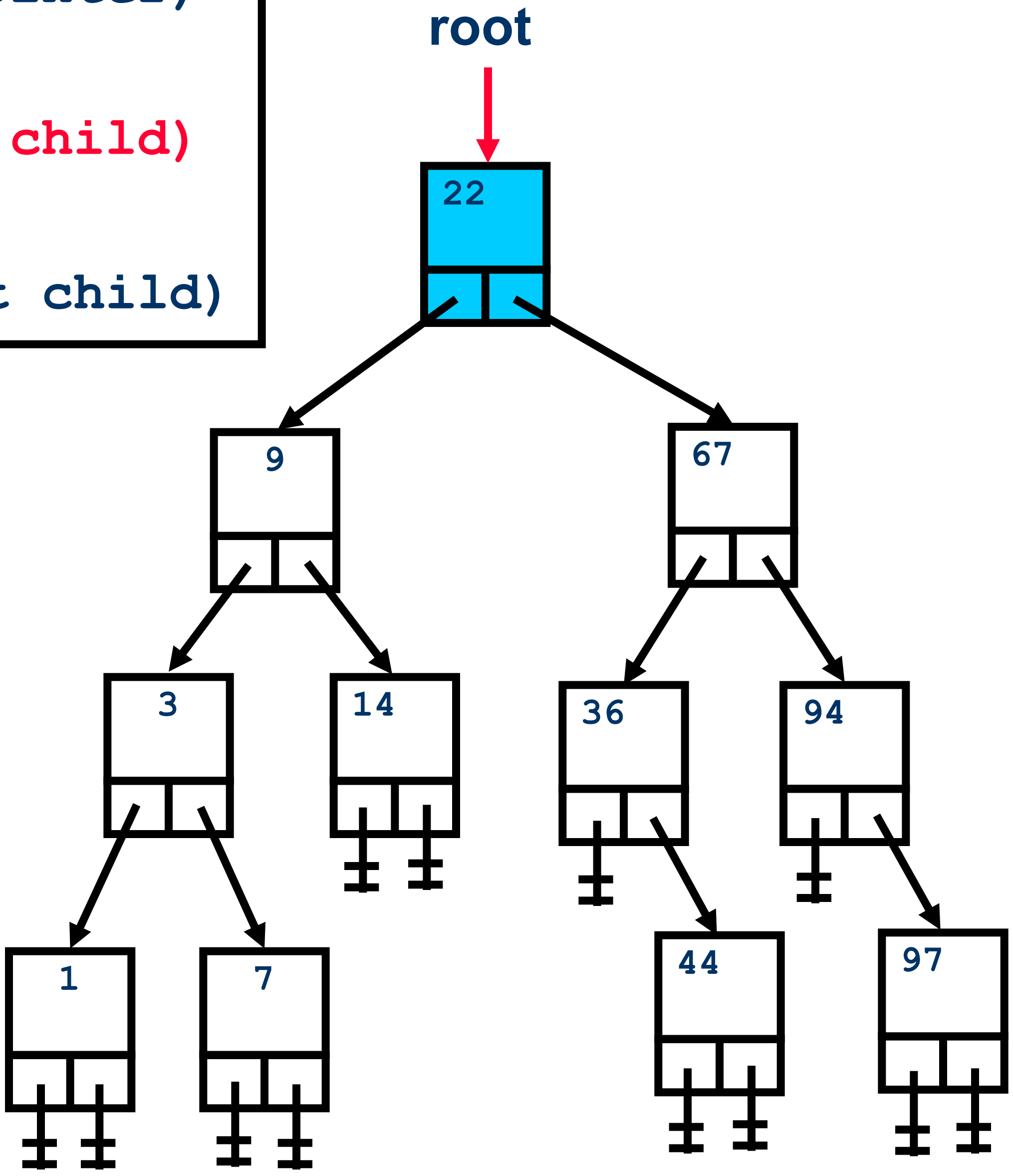
Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```



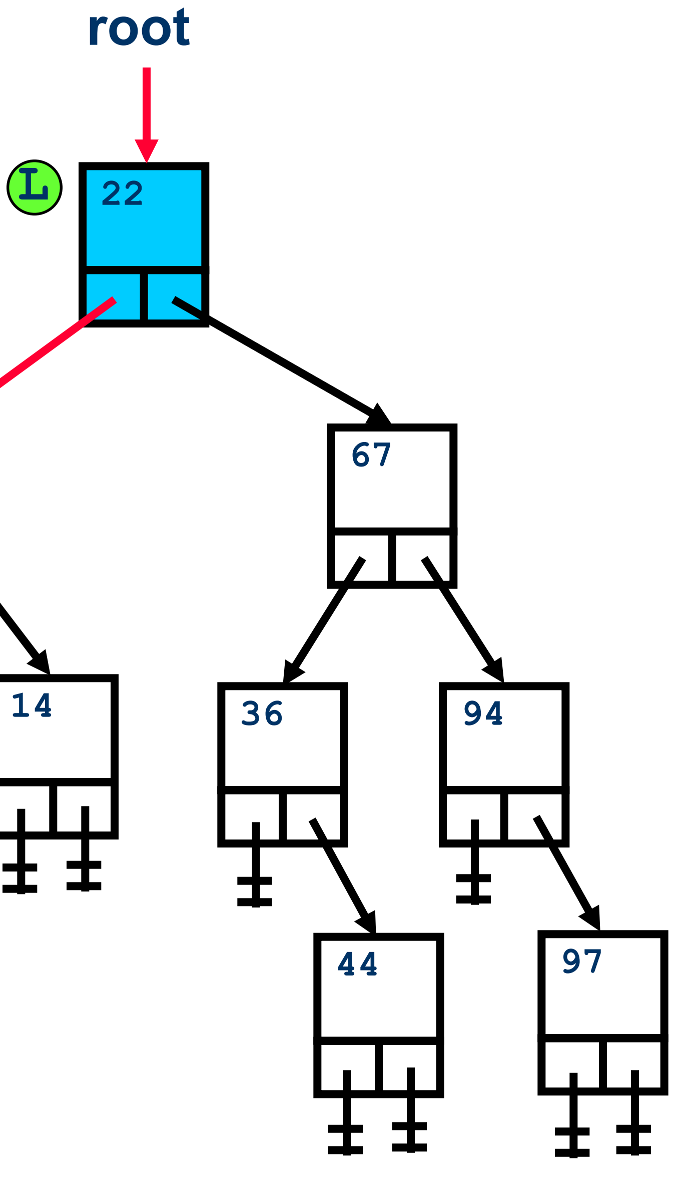
Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```



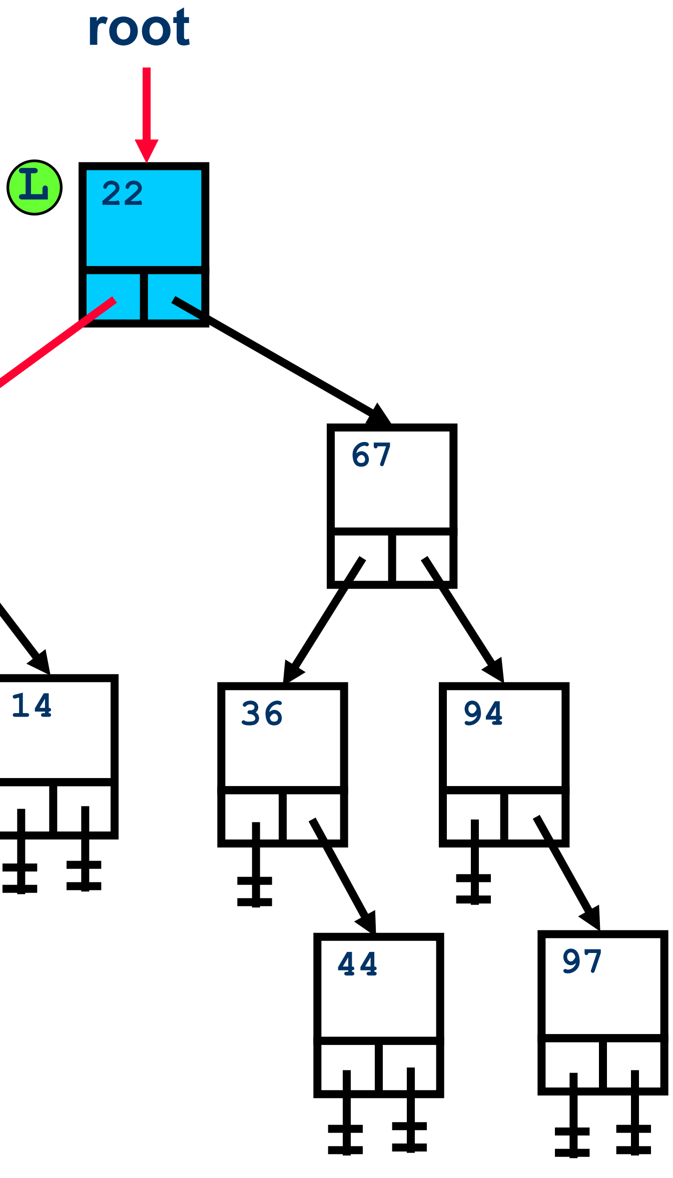
Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```



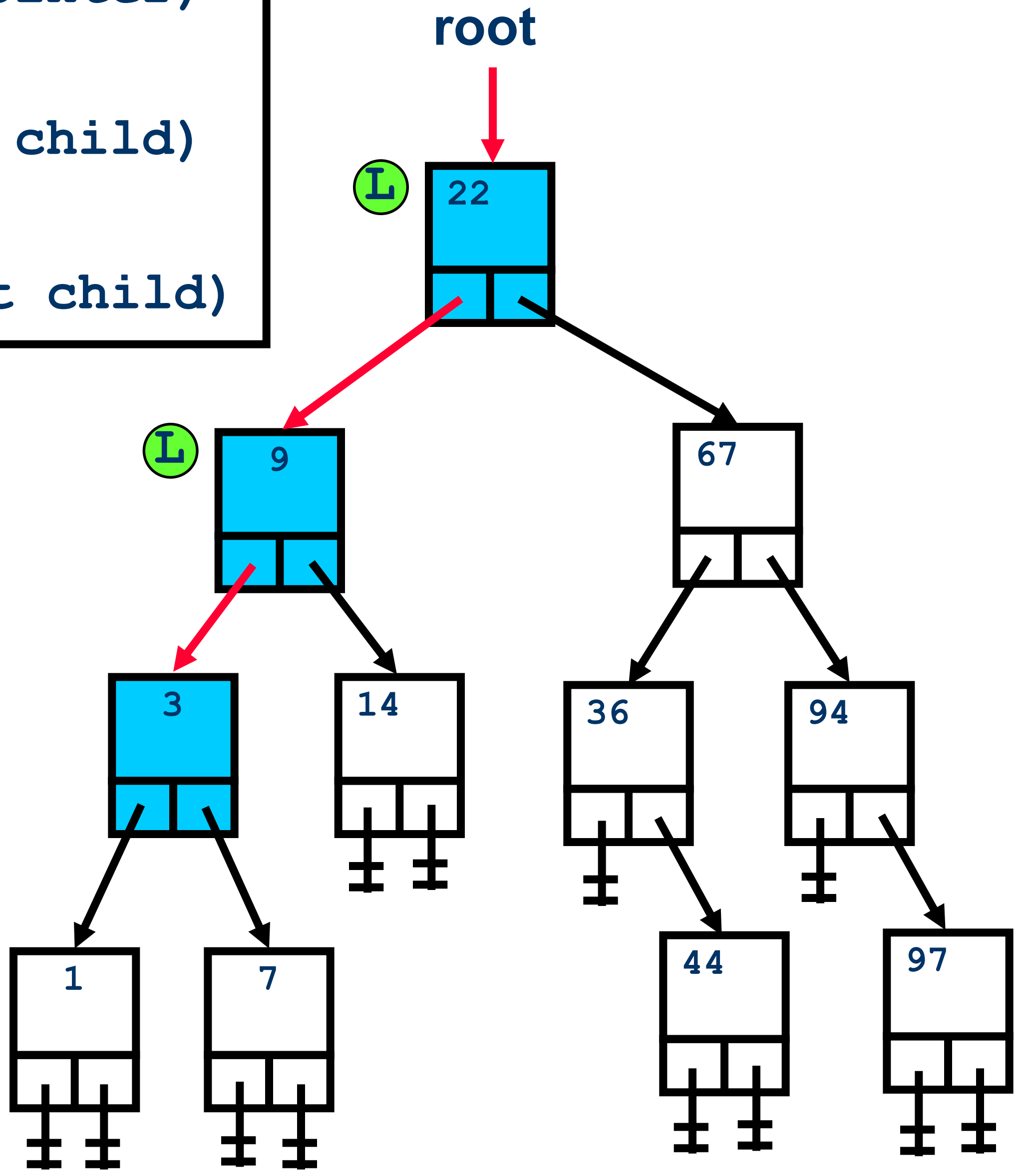
Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```



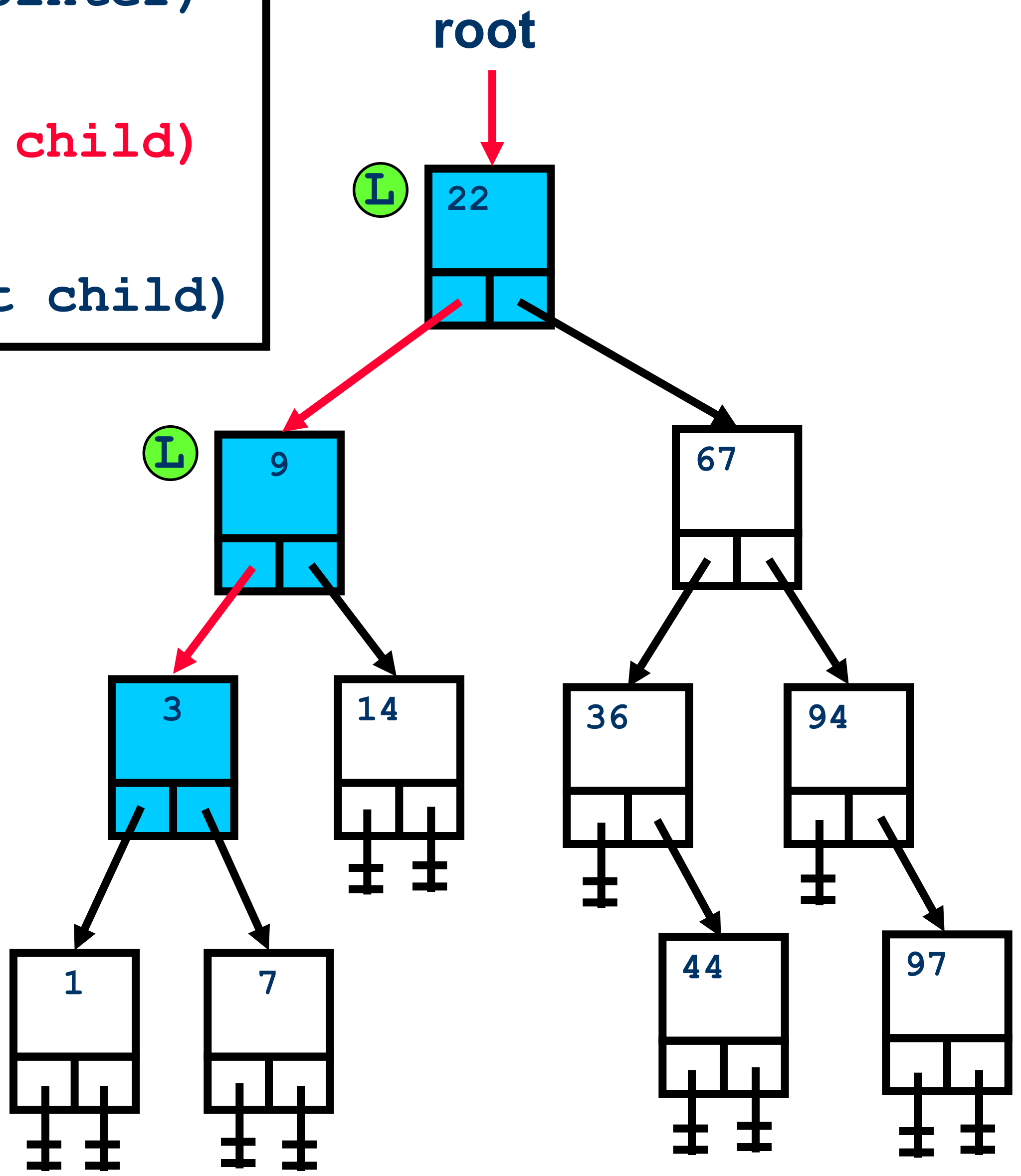
Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```



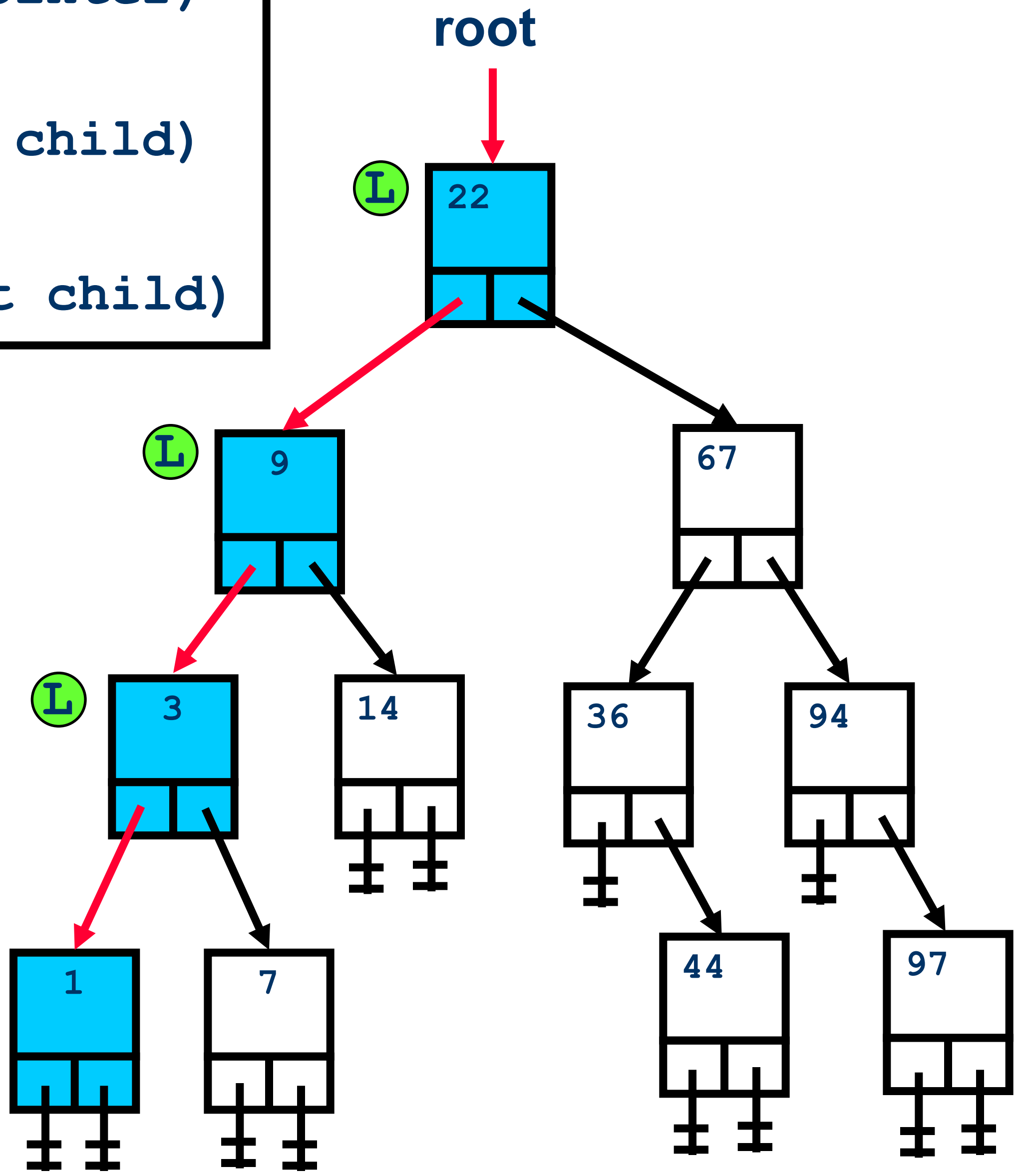
Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```



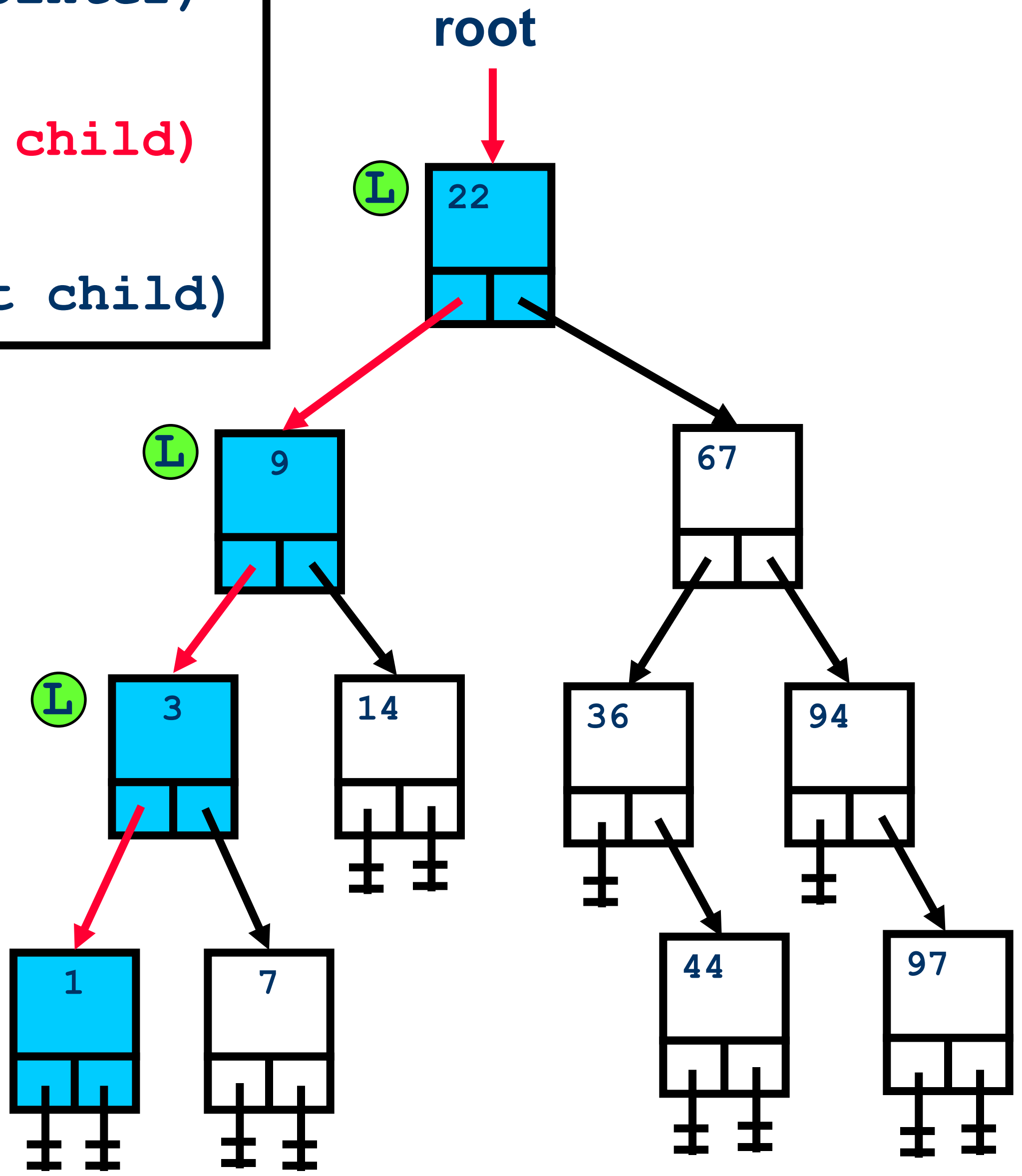
Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```



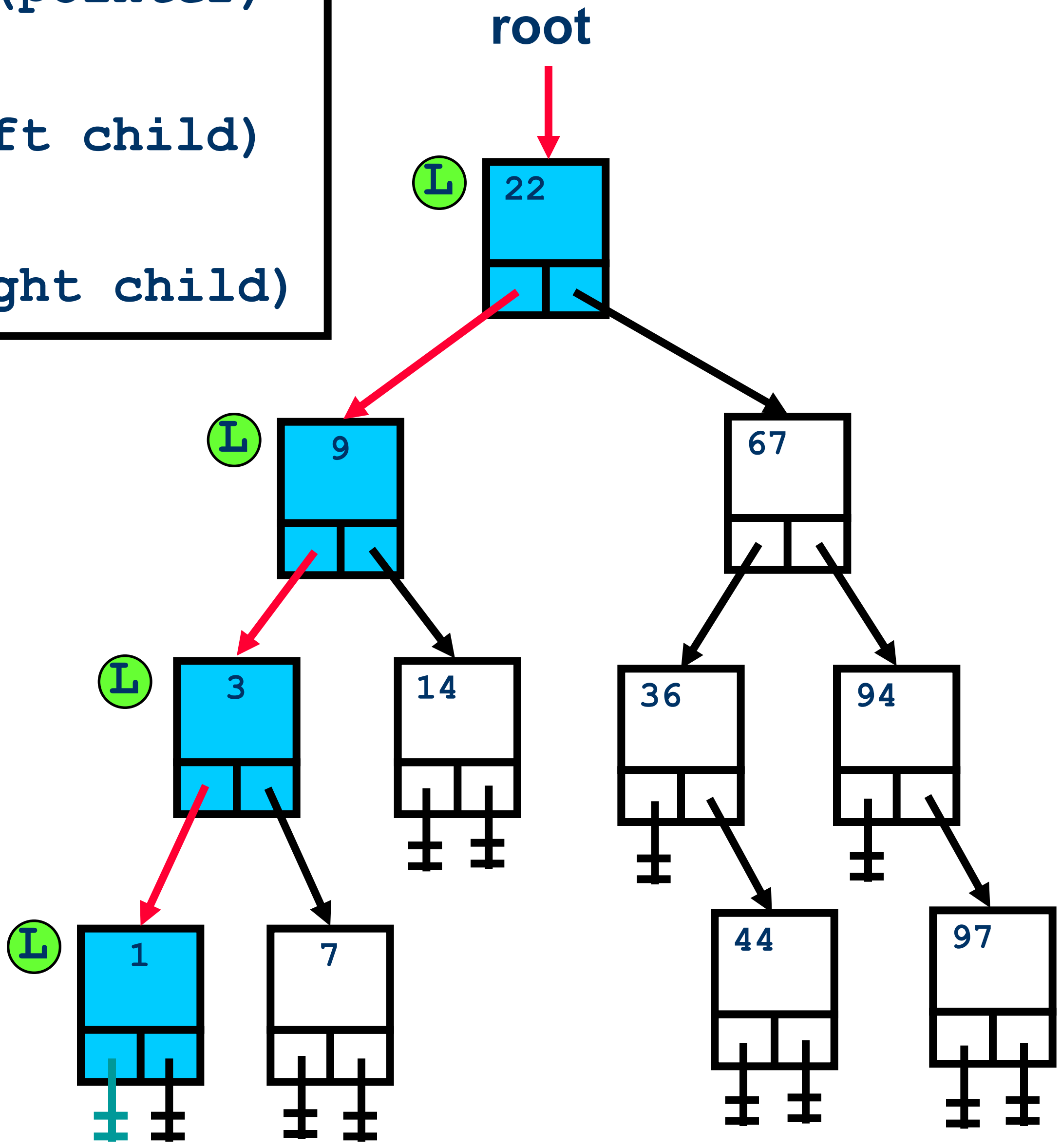
Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```



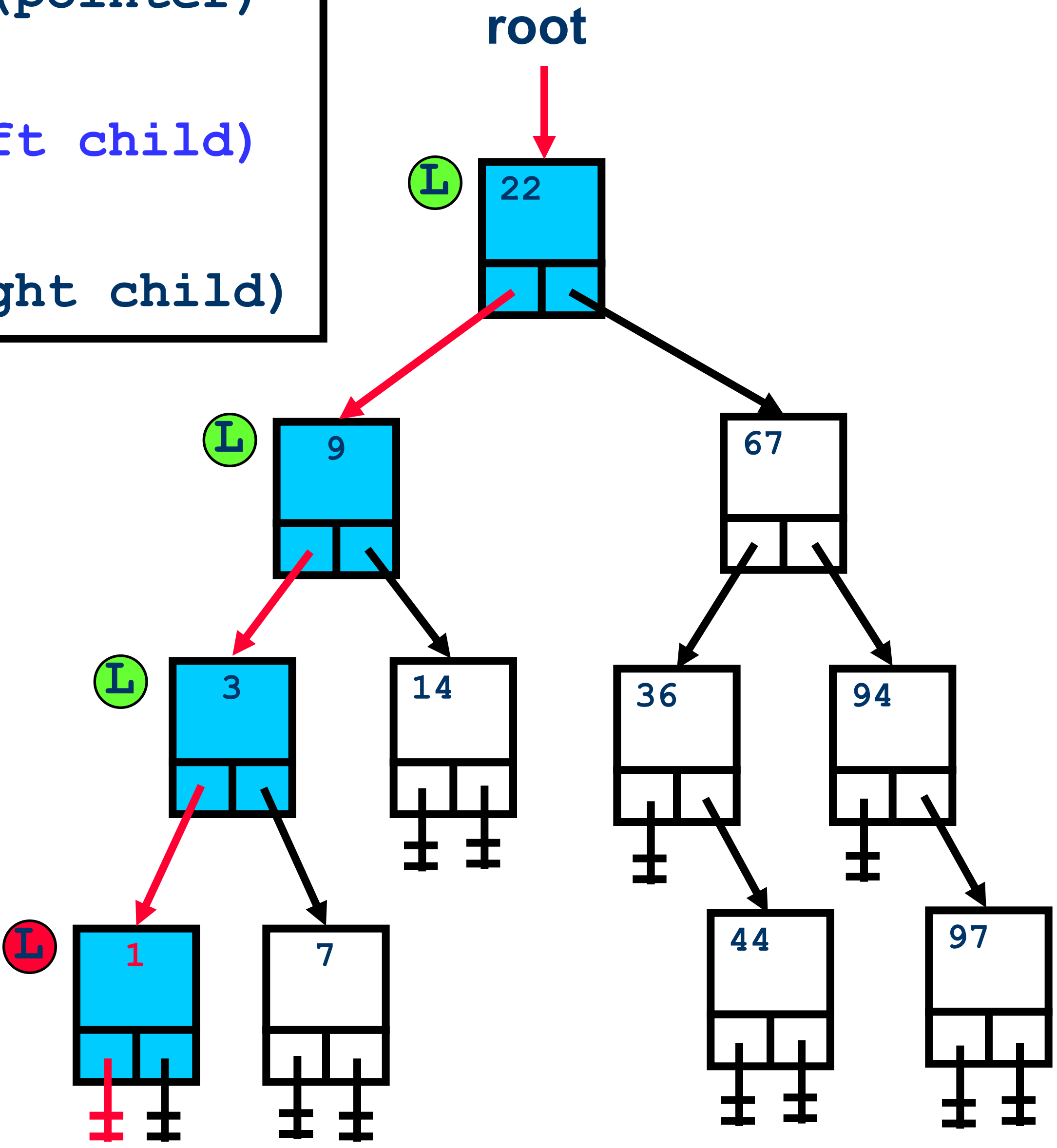
Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```



Inorder Traversal

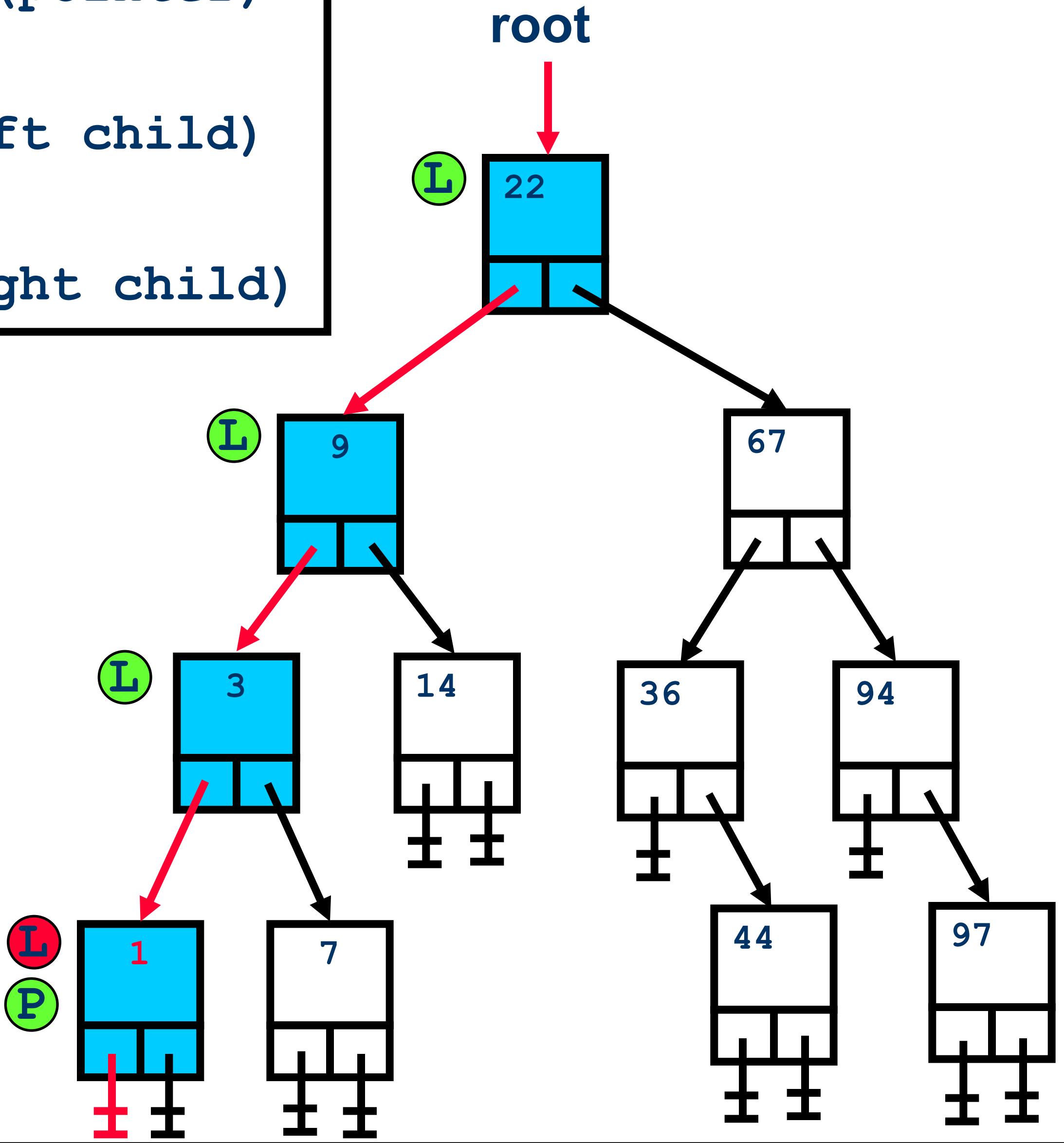
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

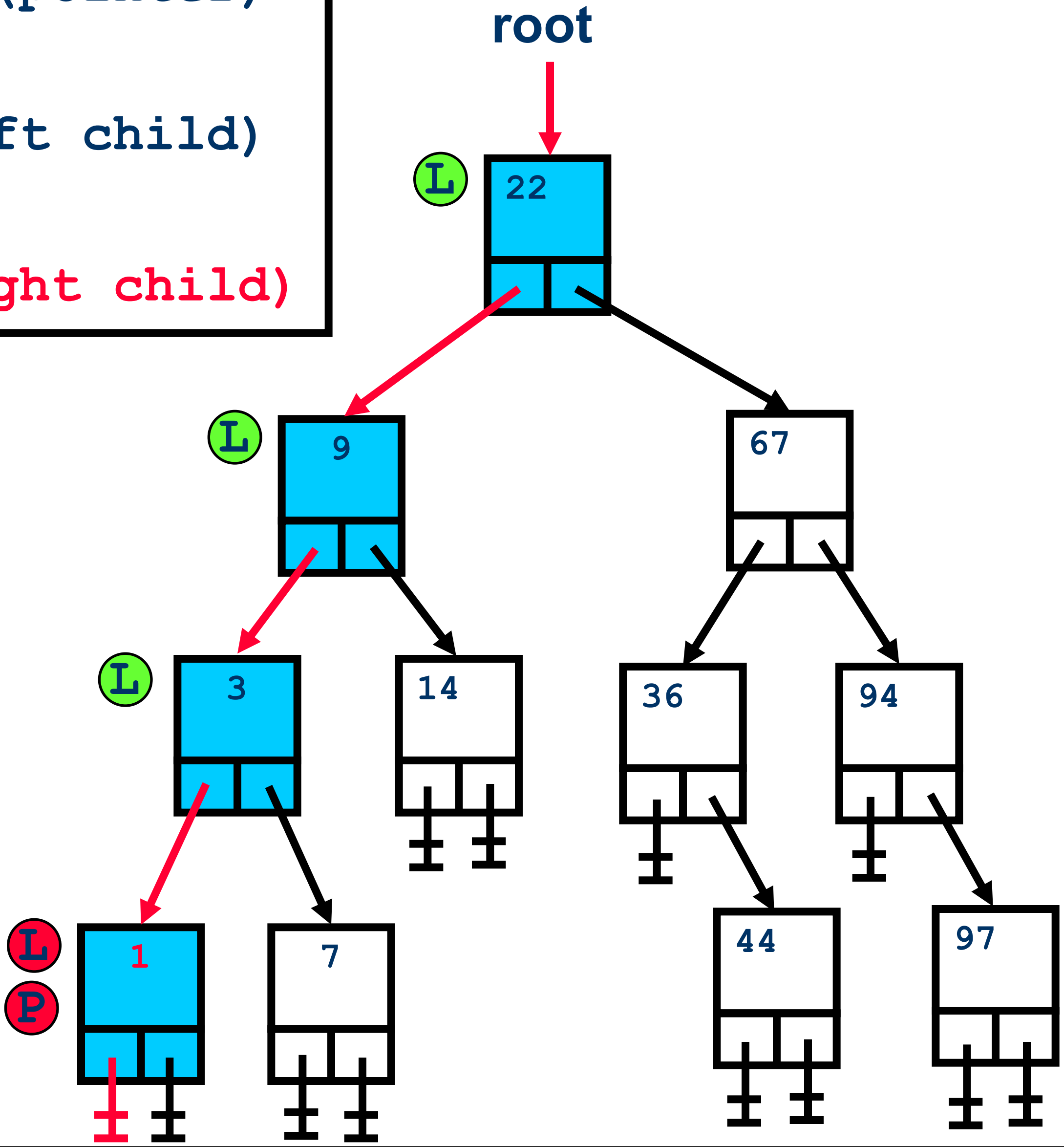
```
Output: 1
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

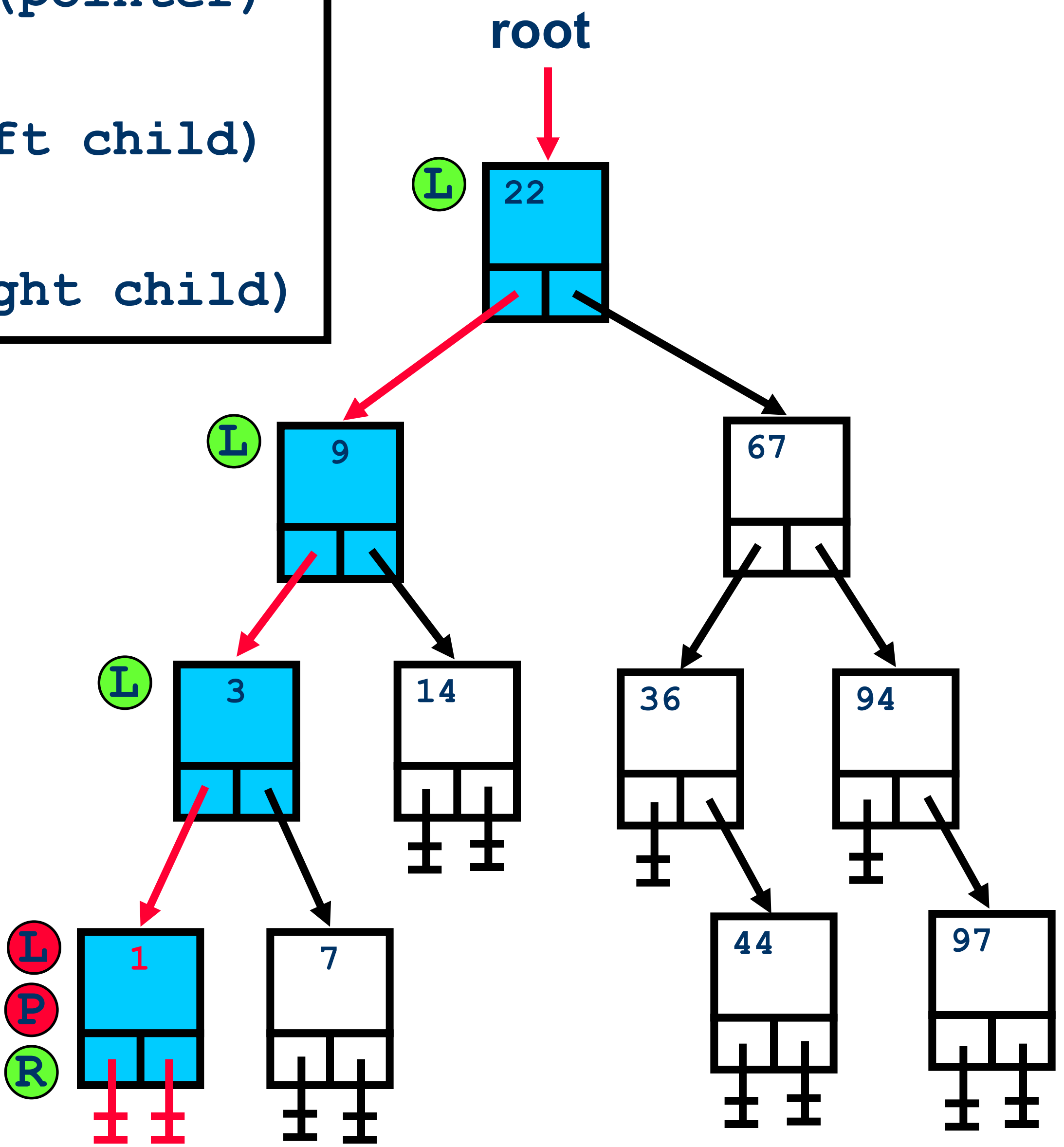
```
Output: 1
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

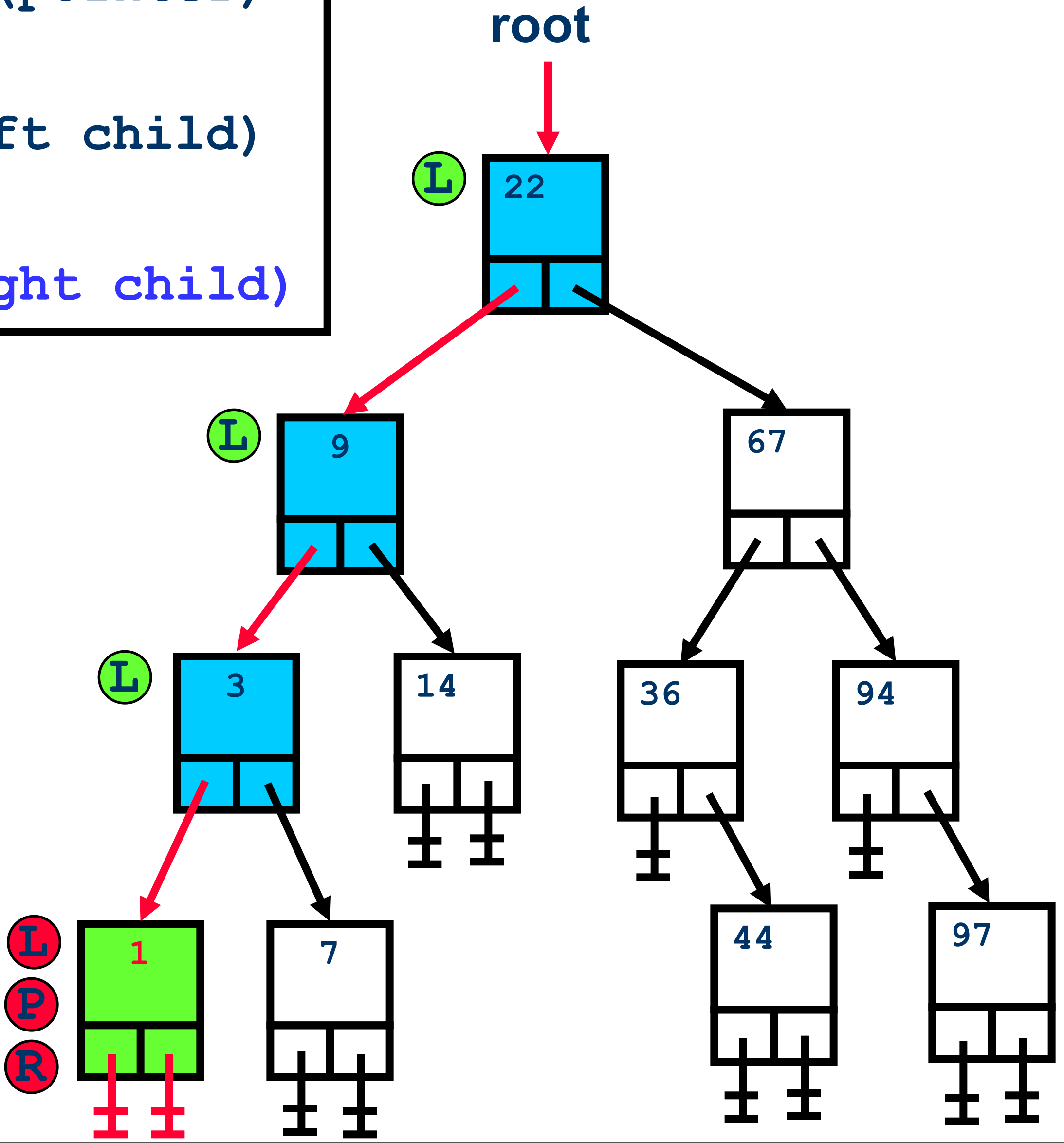
```
Output: 1
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

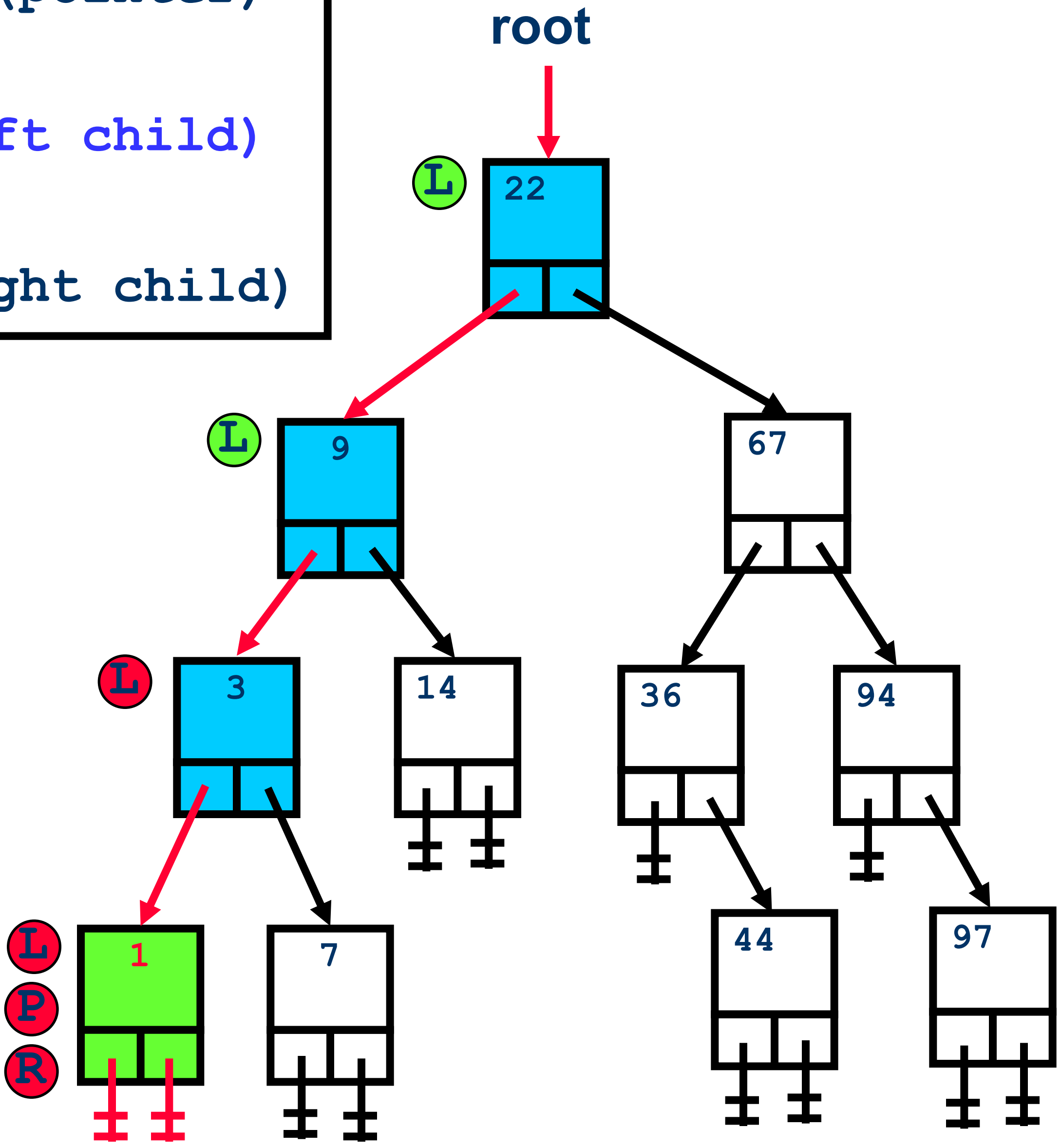
```
Output: 1
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

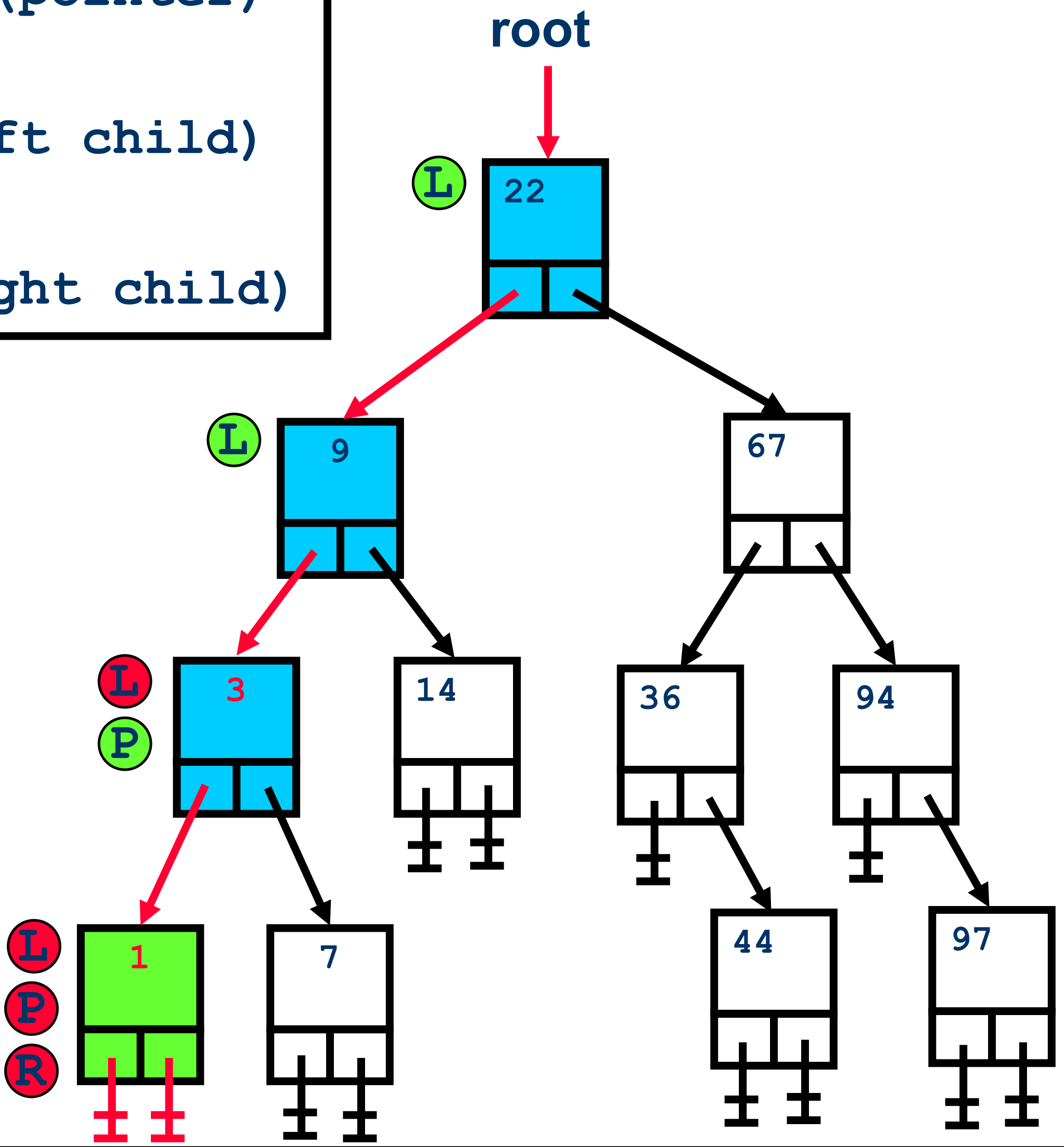
```
Output: 1
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right child)
```

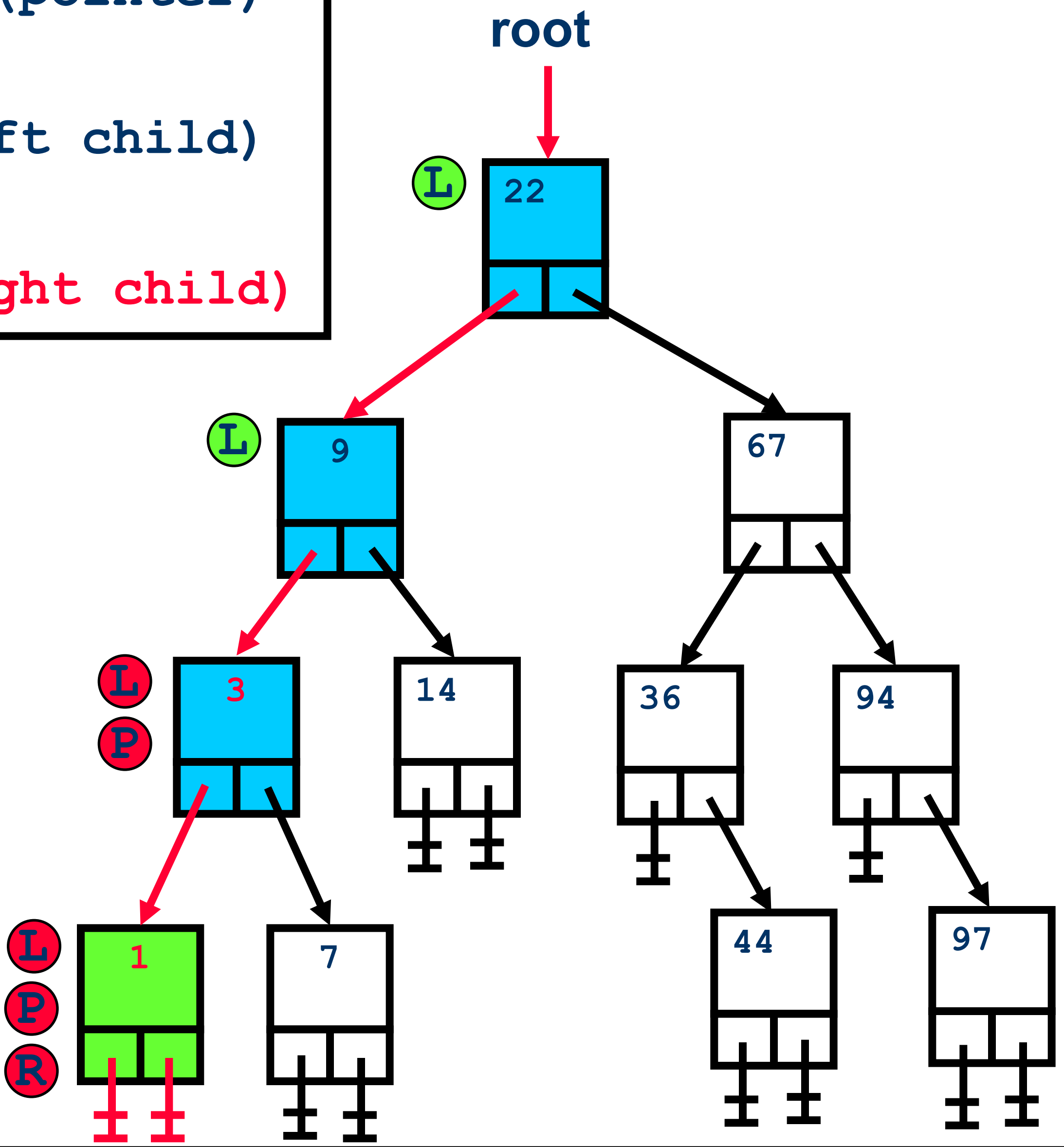
```
Output: 1 3
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

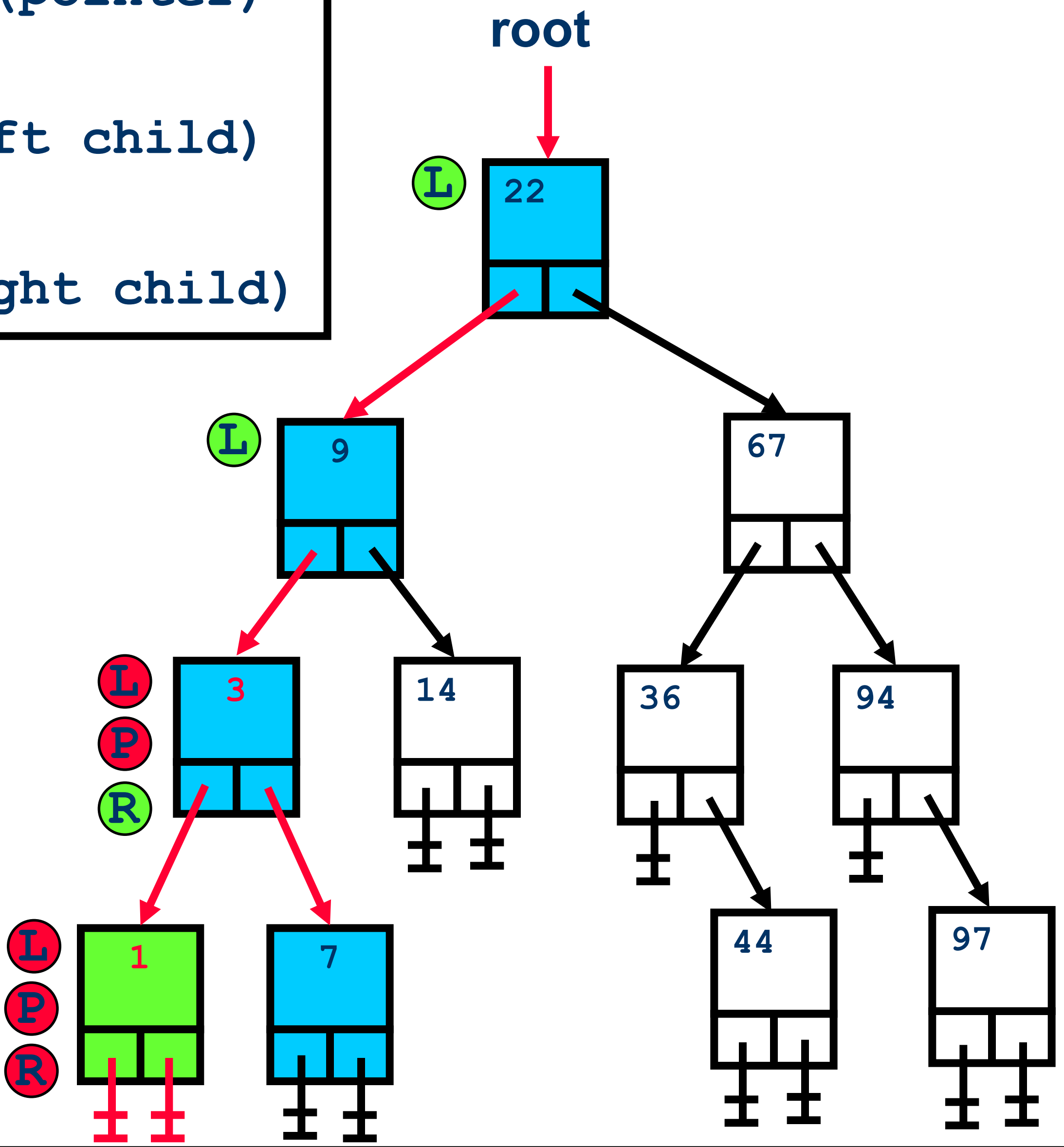
```
Output: 1 3
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

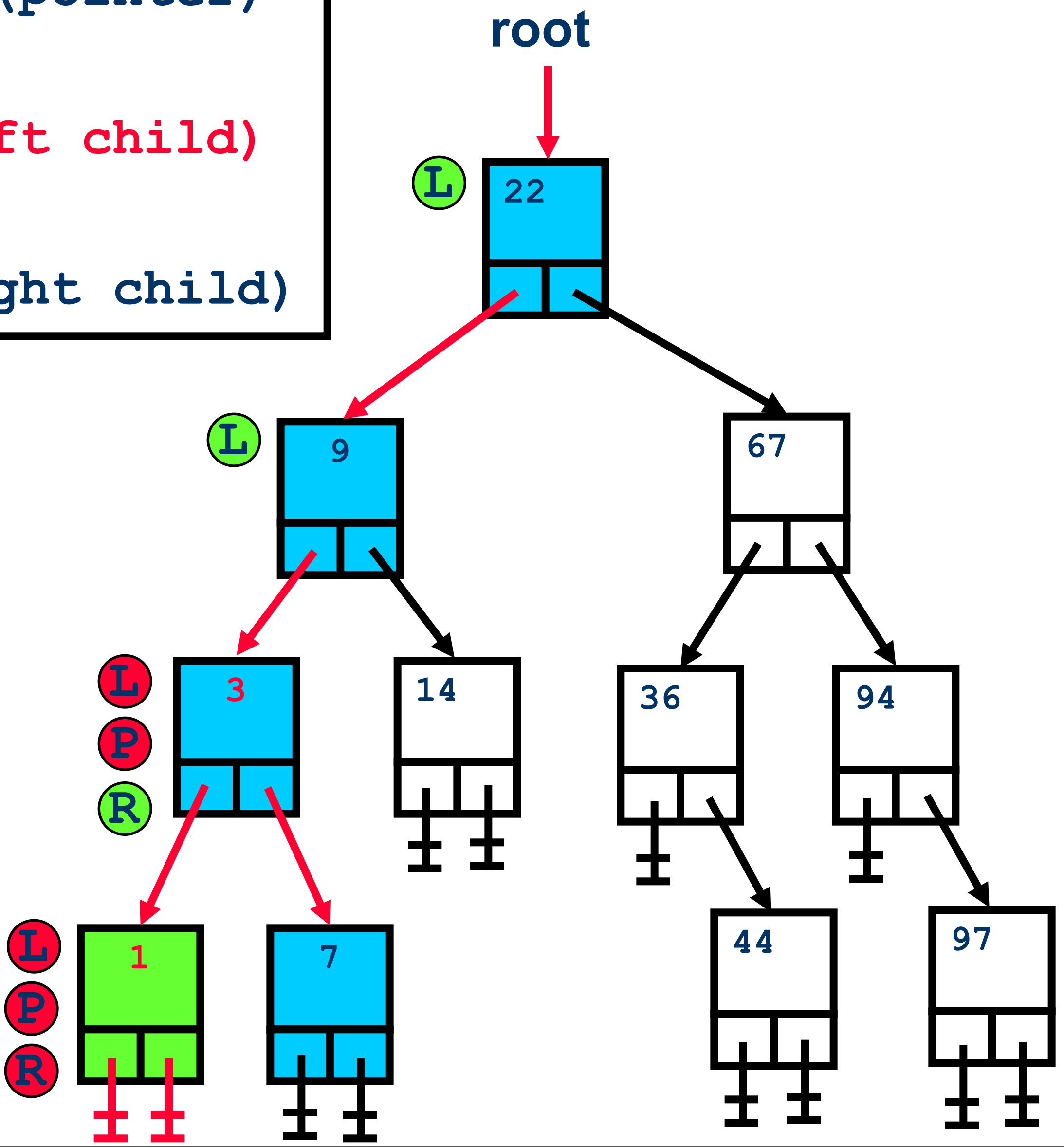
```
Output: 1 3
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

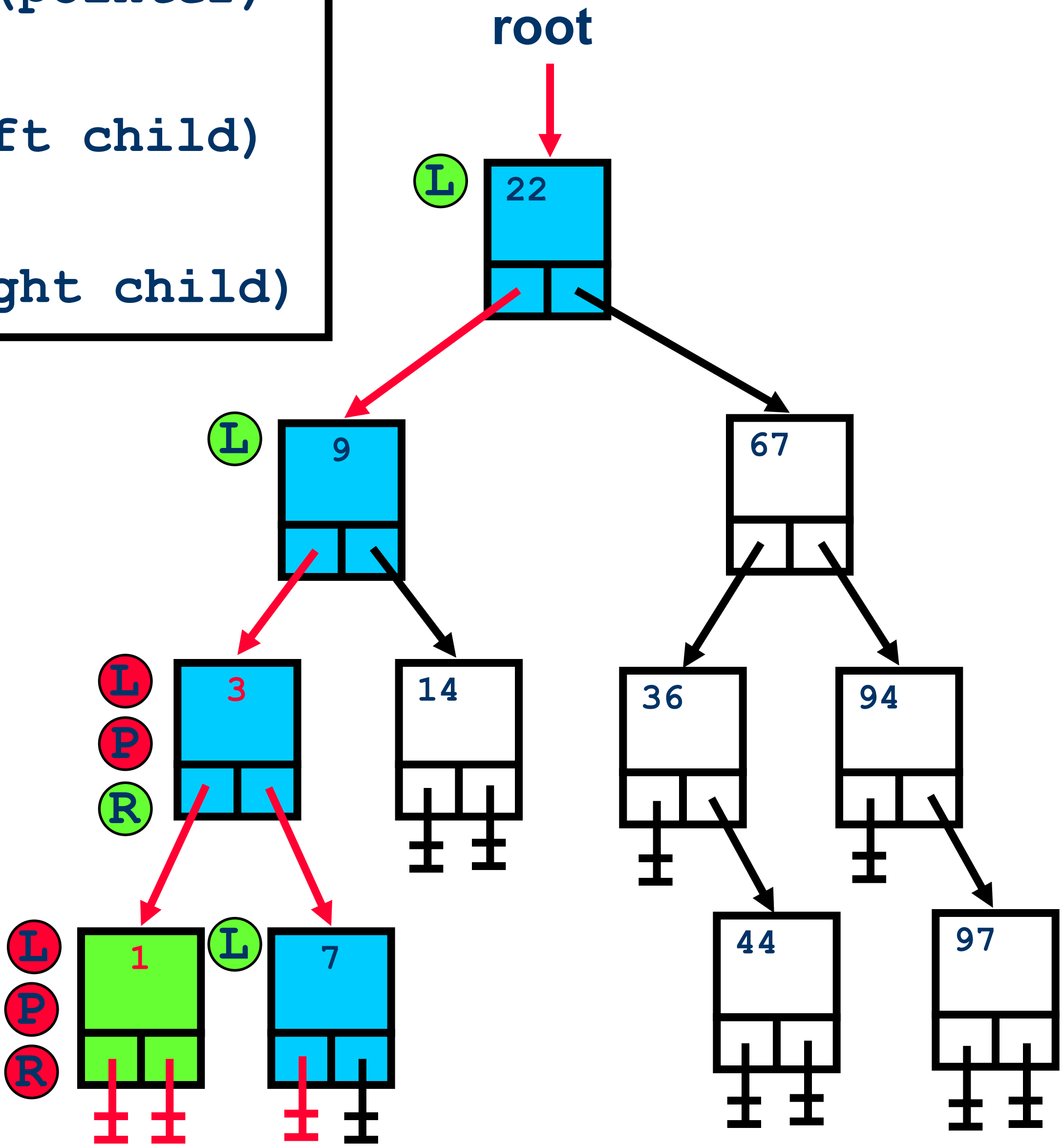
```
Output: 1 3
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

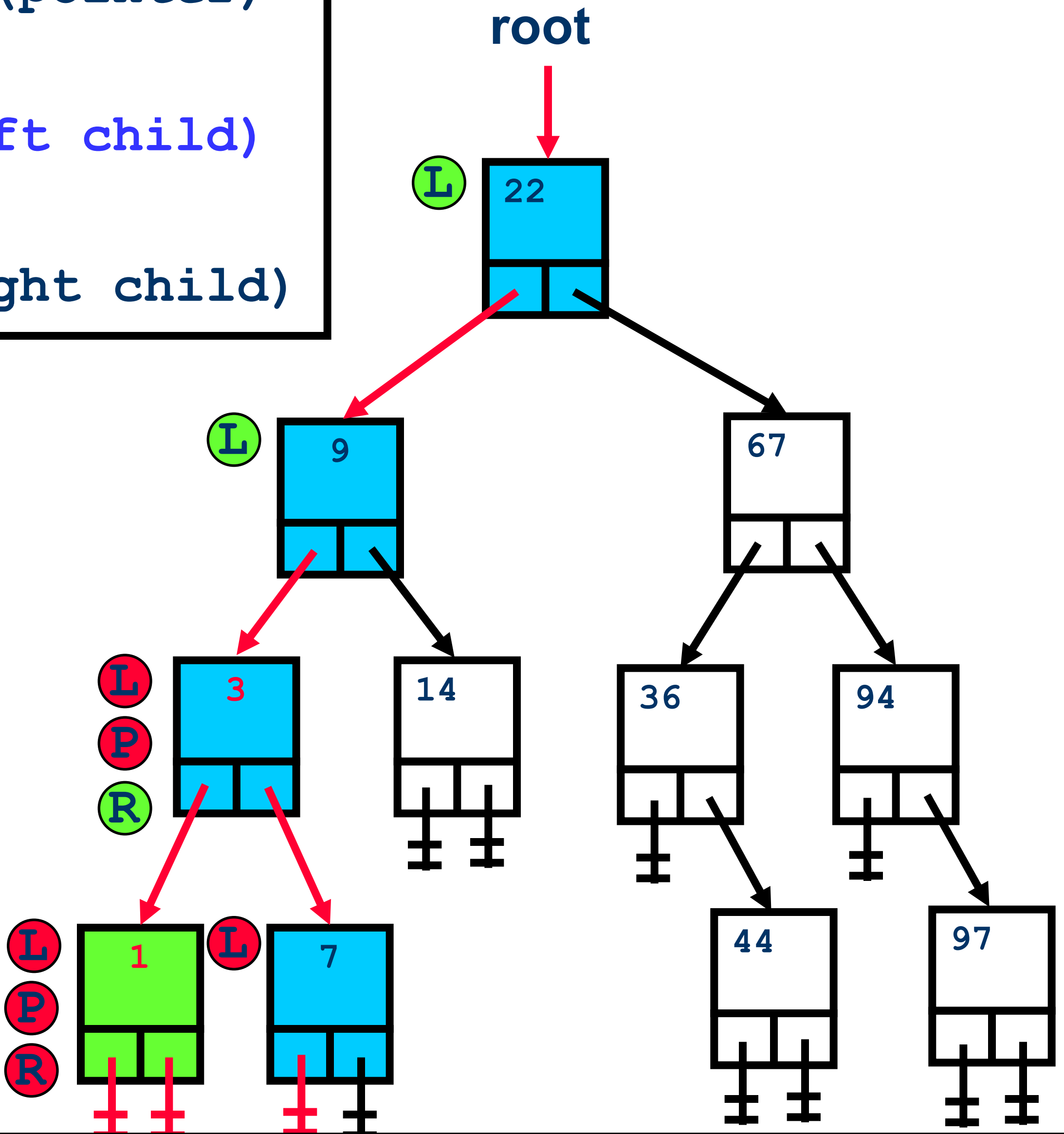
```
Output: 1 3
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

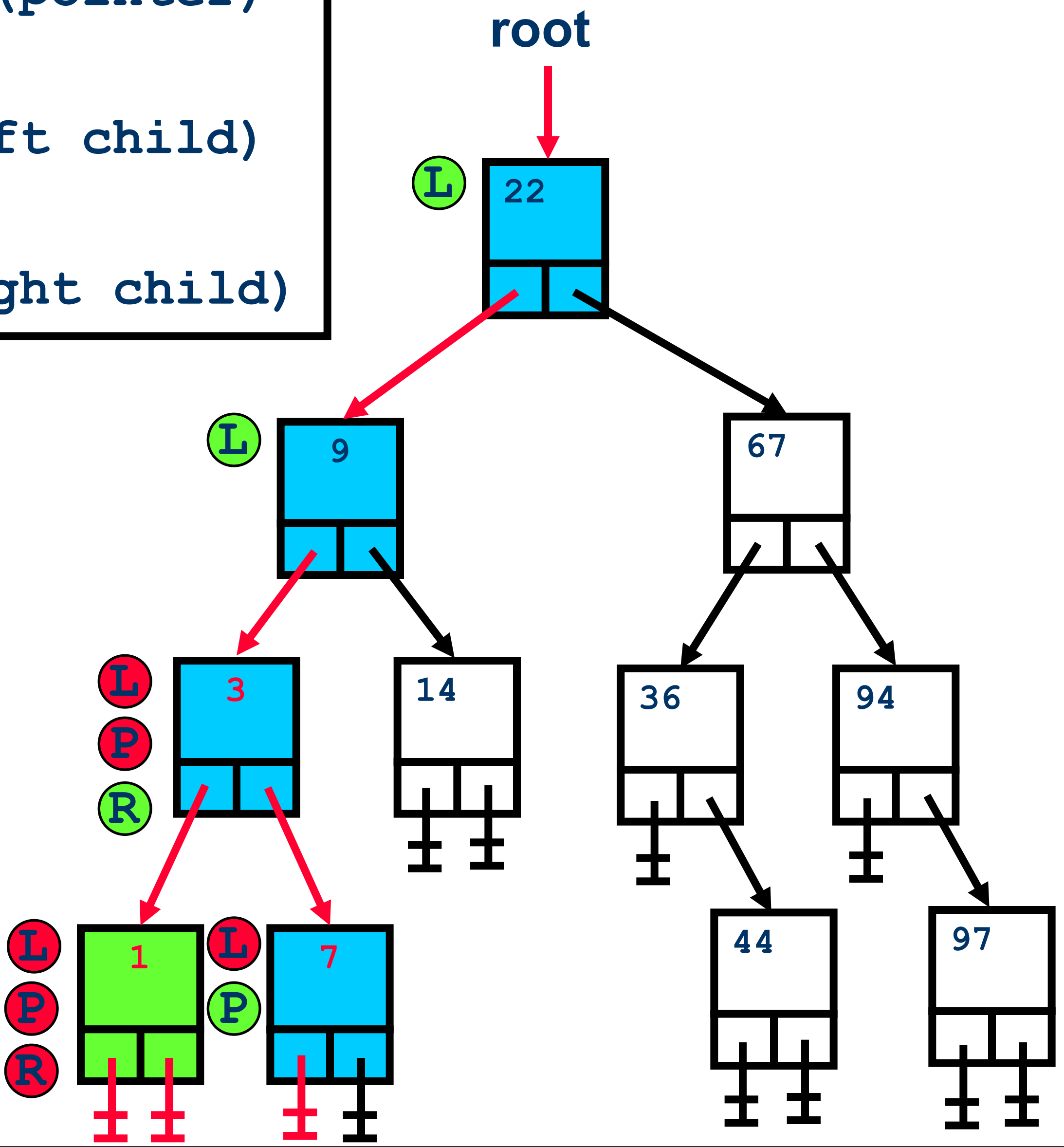
```
Output: 1 3
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

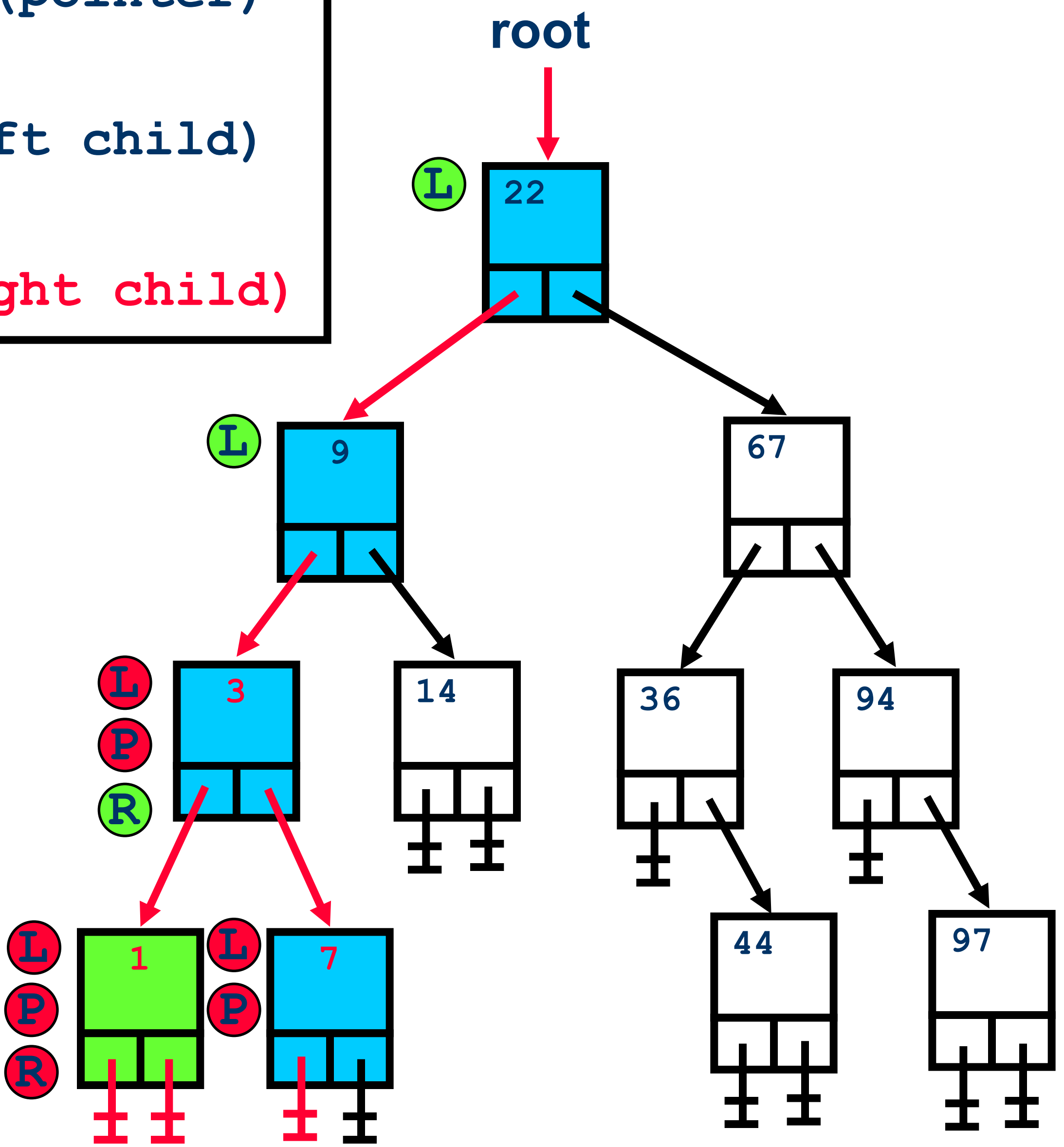
```
Output: 1 3 7
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

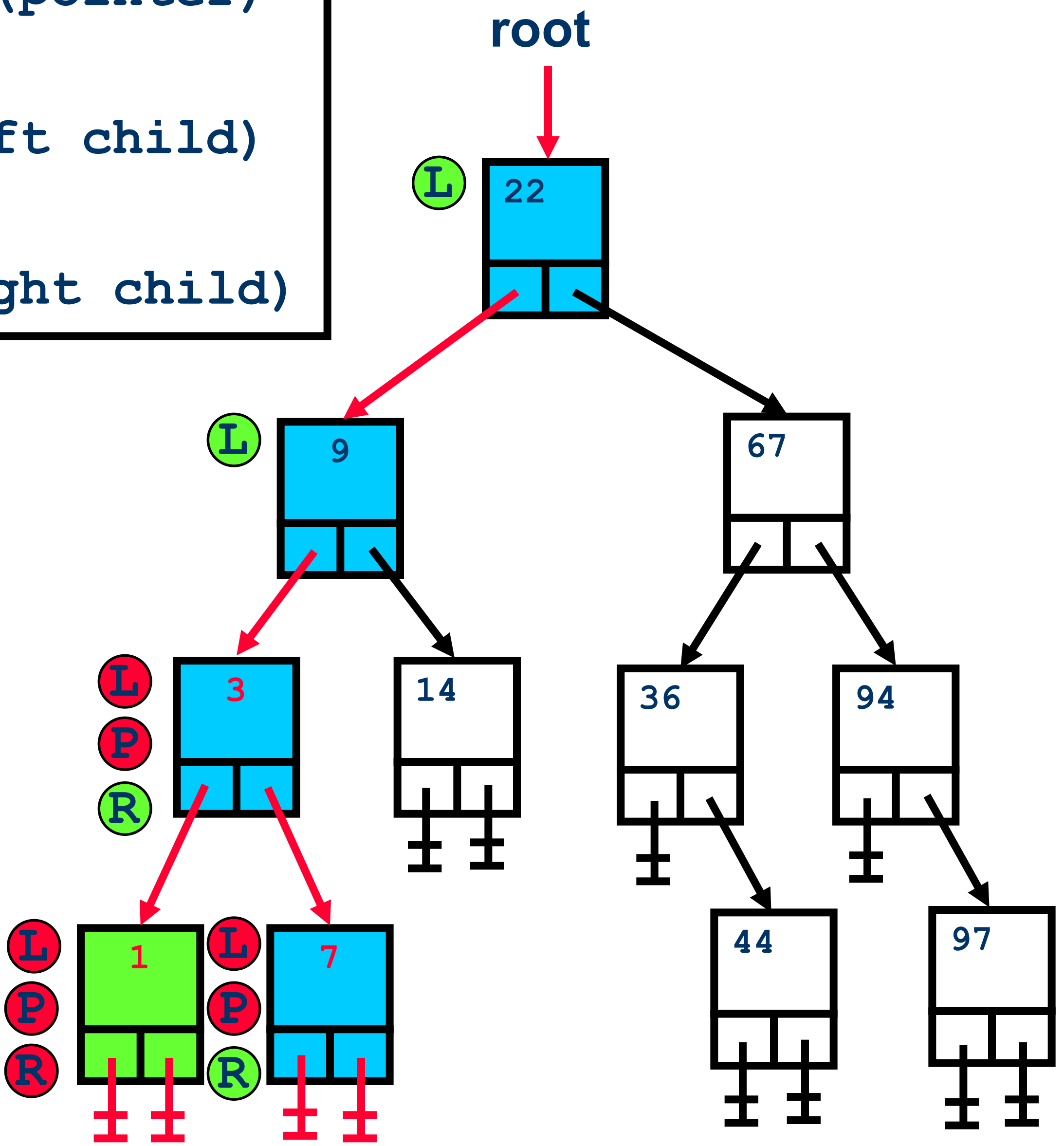
```
Output: 1 3 7
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

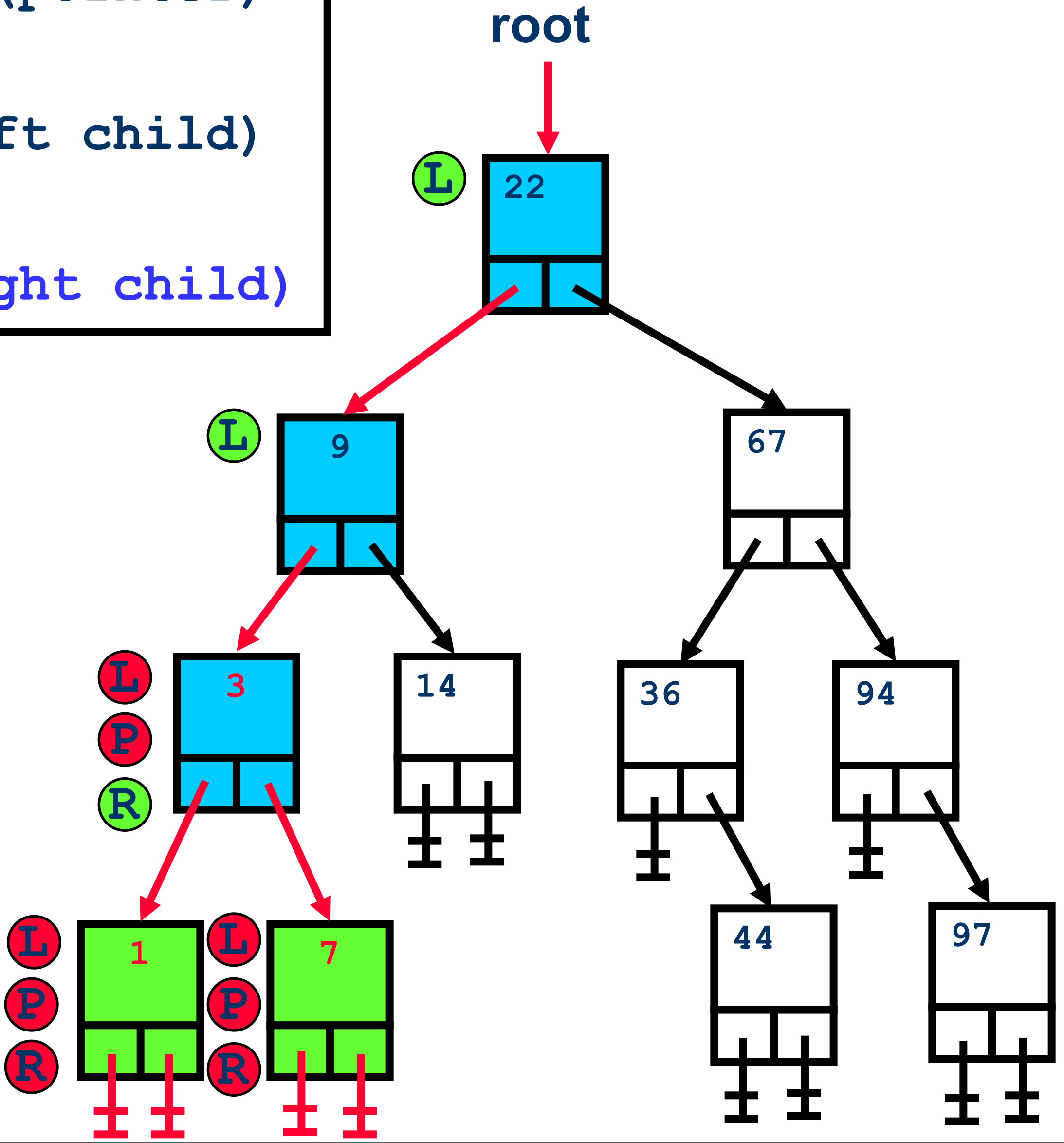
```
Output: 1 3 7
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

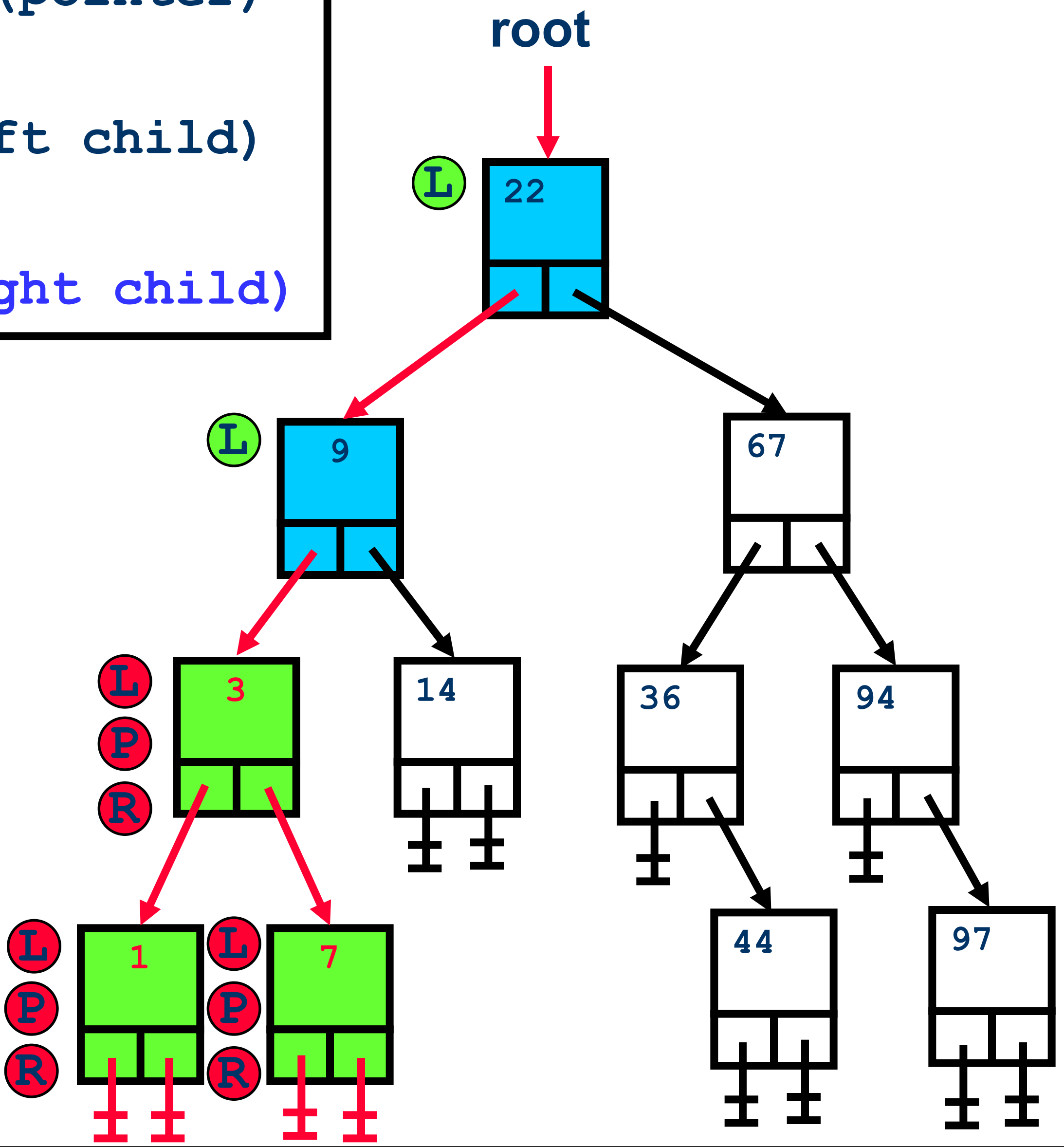
```
Output: 1 3 7
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

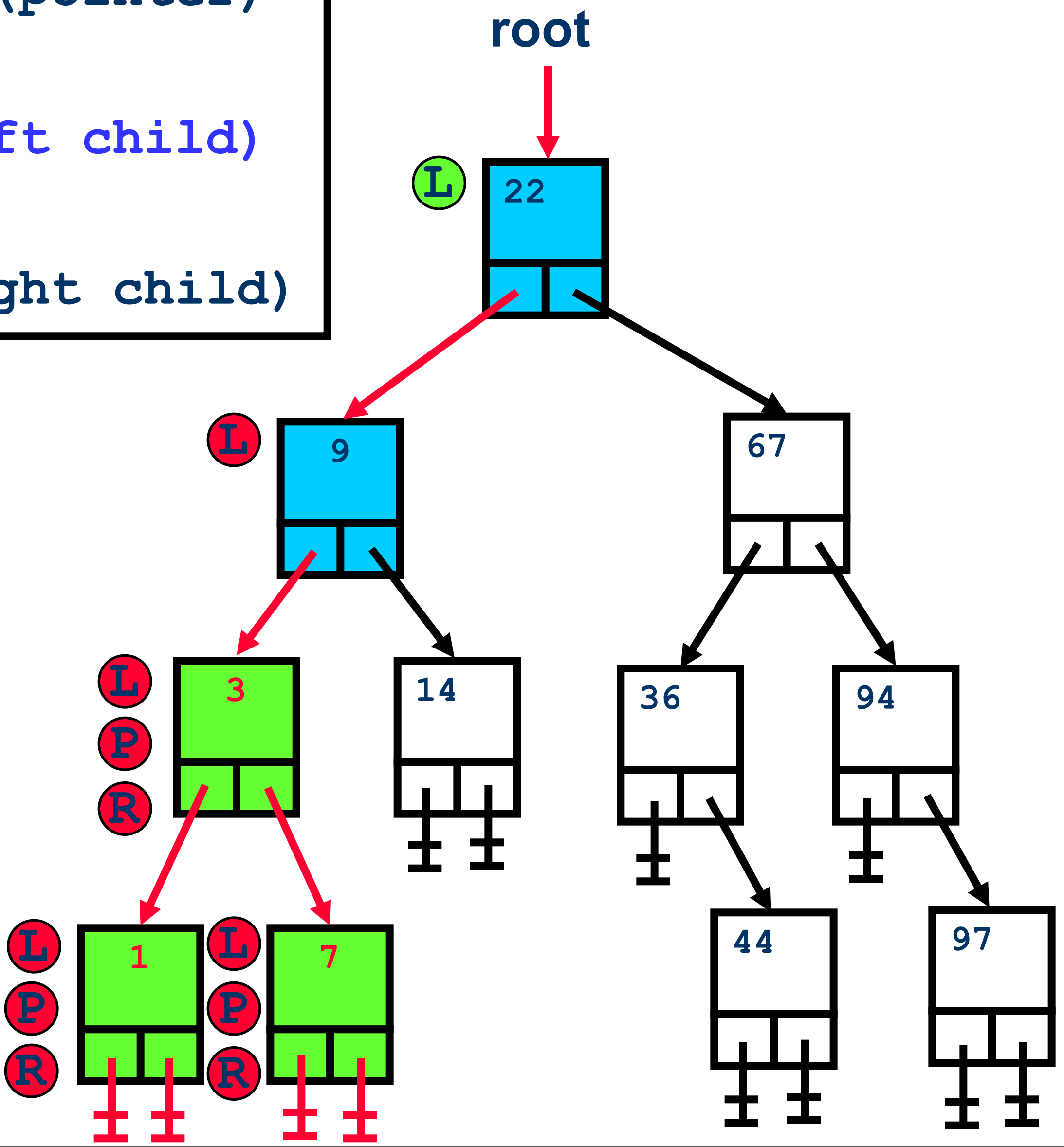
```
Output: 1 3 7
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right child)
```

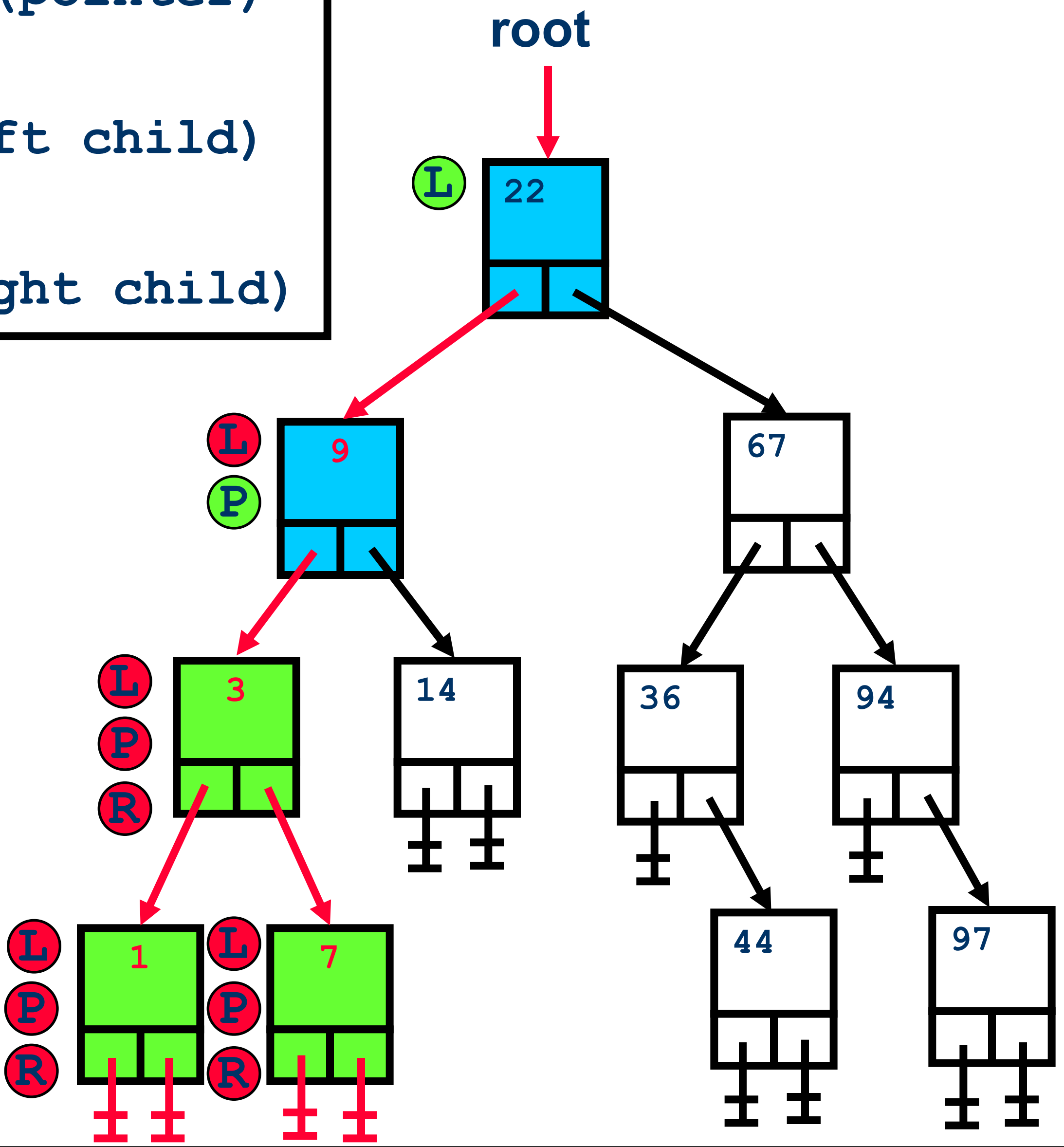
```
Output: 1 3 7
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

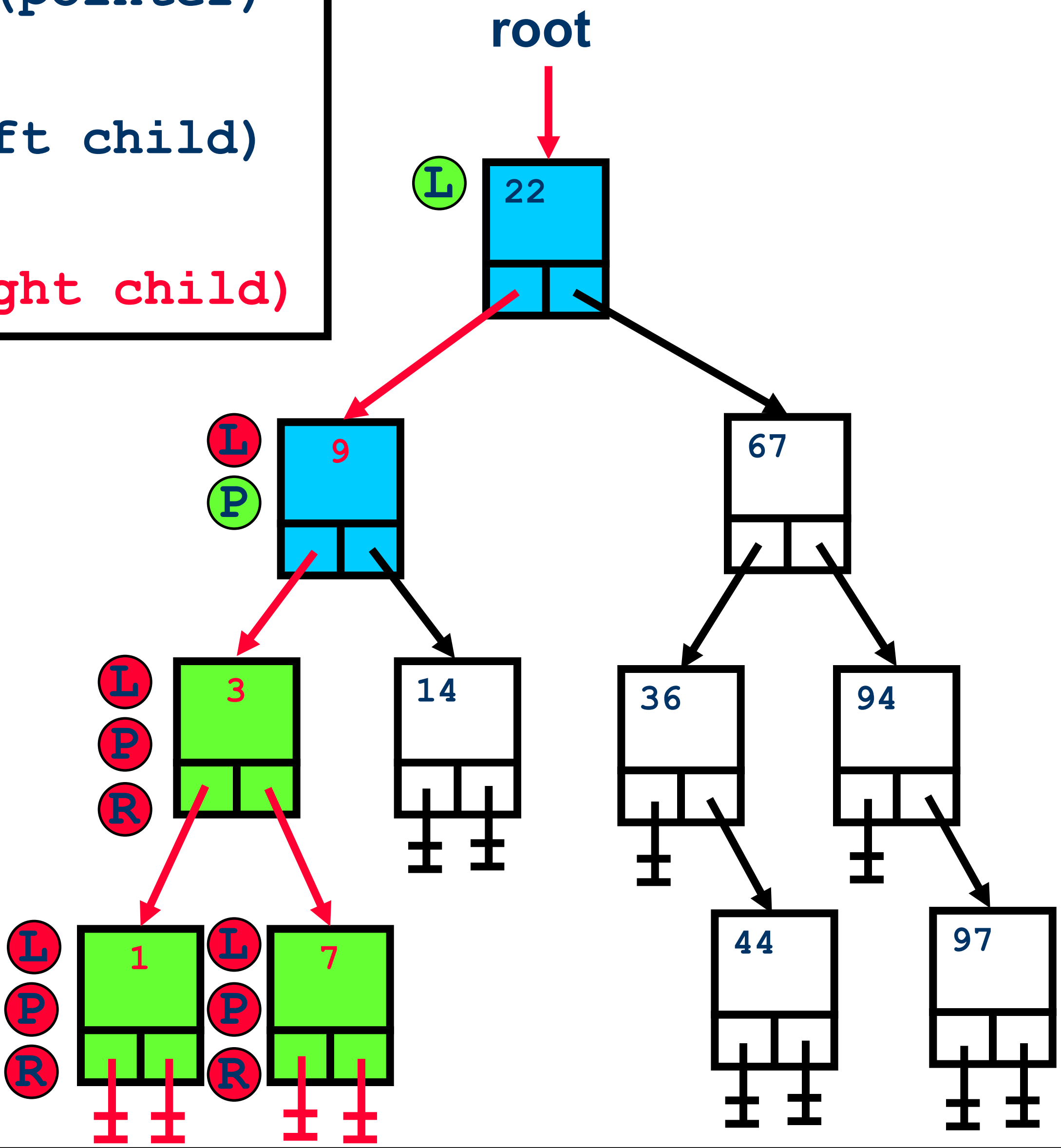
```
Output: 1 3 7 9
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

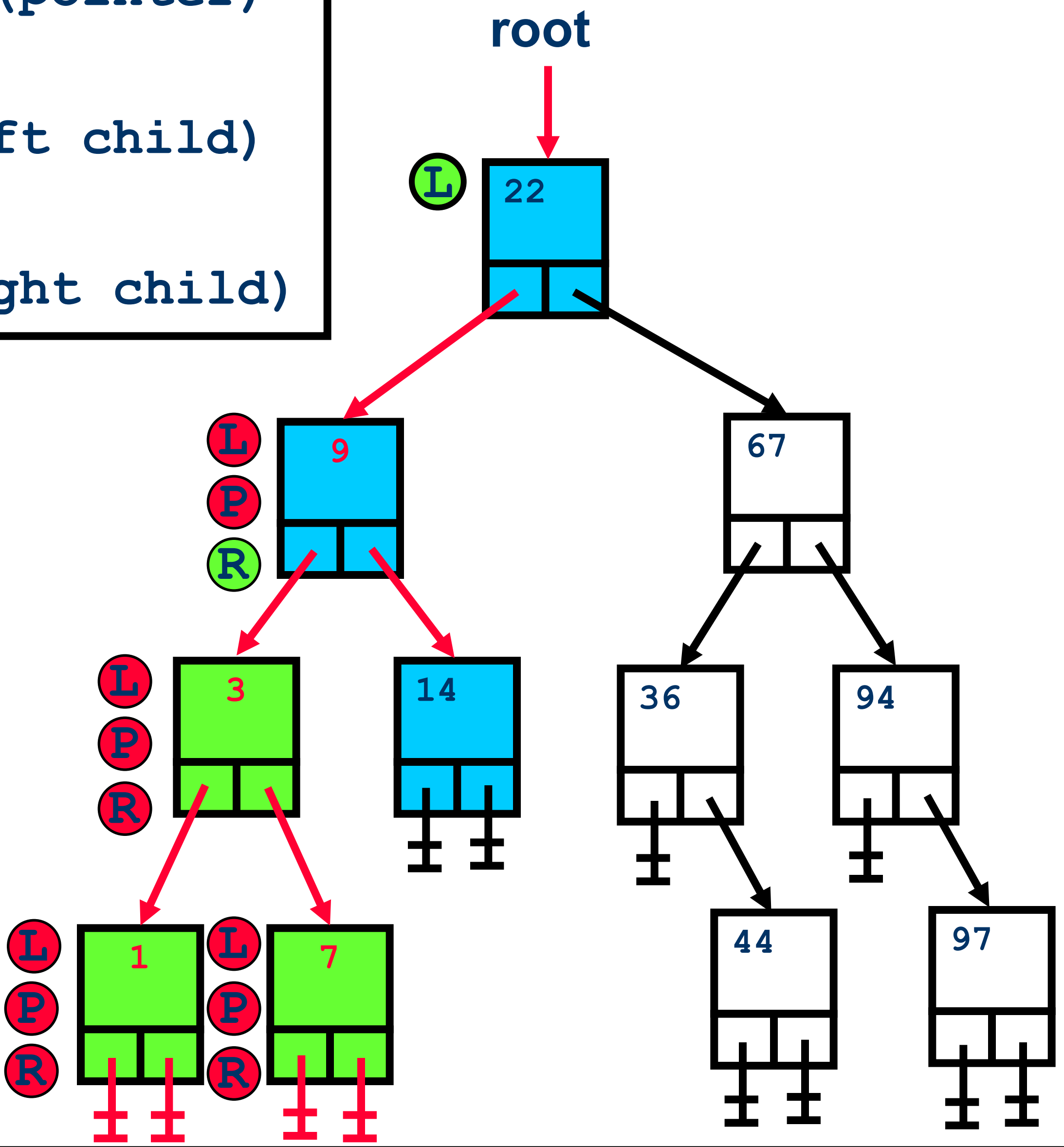
```
Output: 1 3 7 9
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

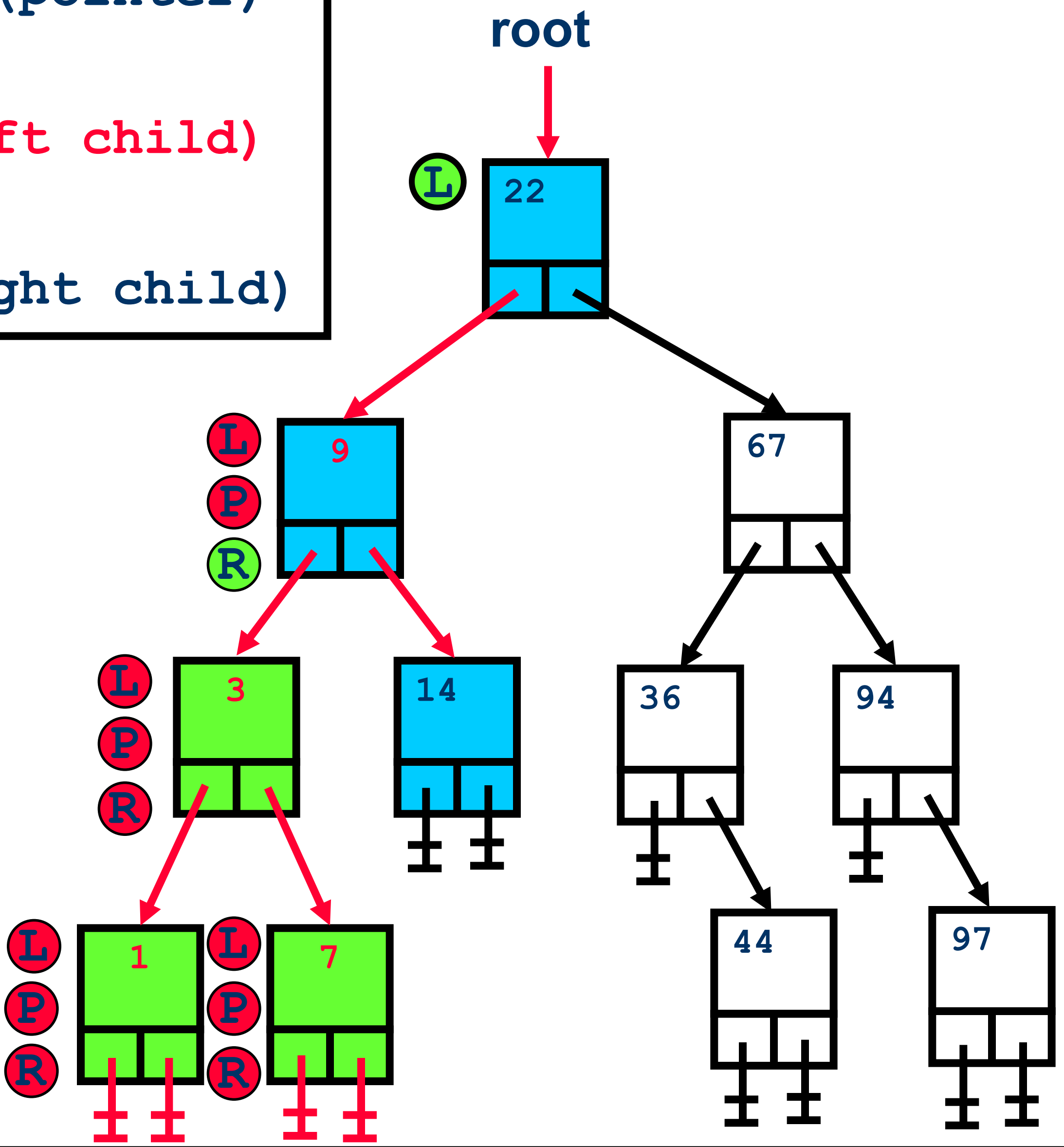
```
Output: 1 3 7 9
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

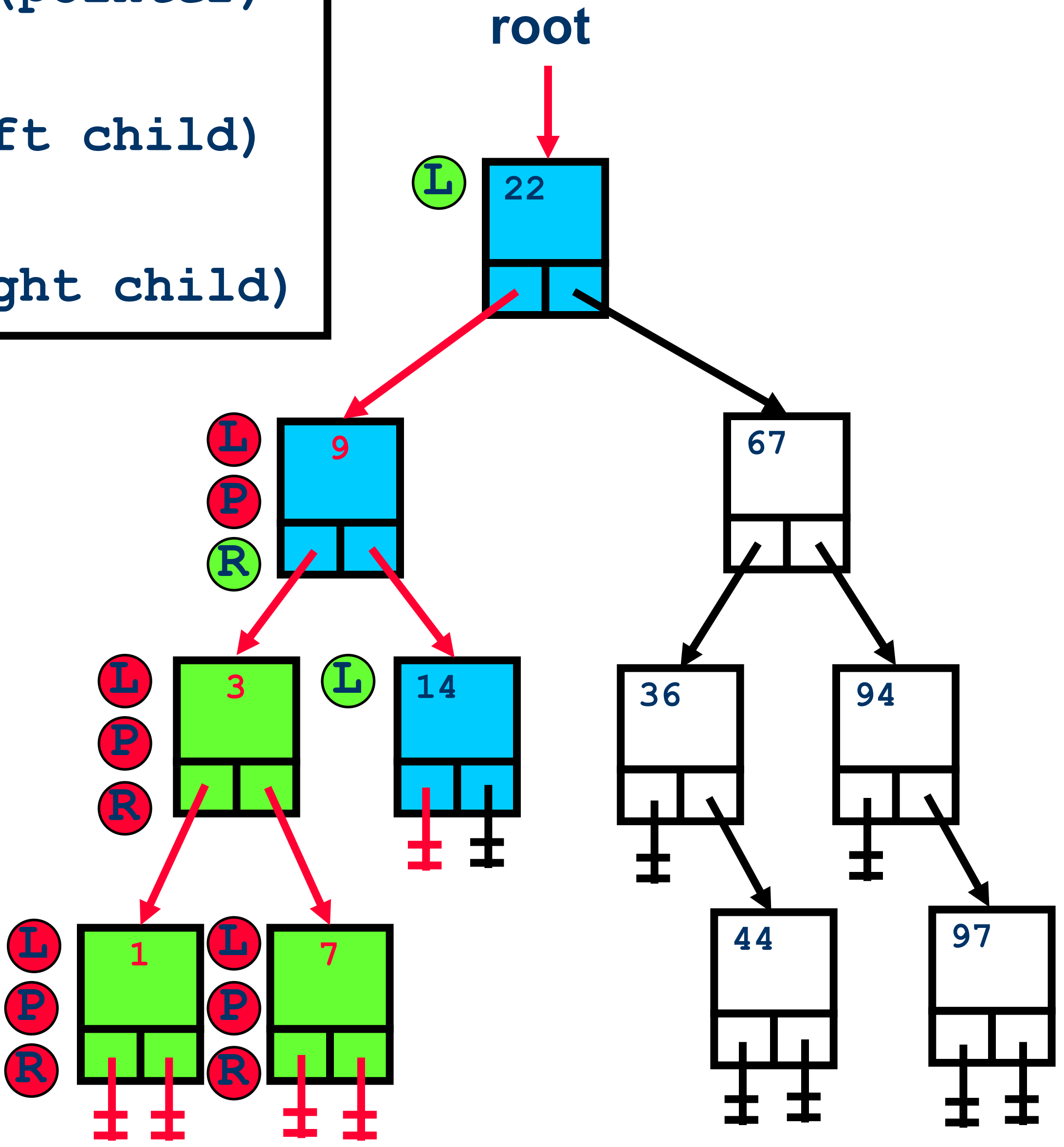
```
Output: 1 3 7 9
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

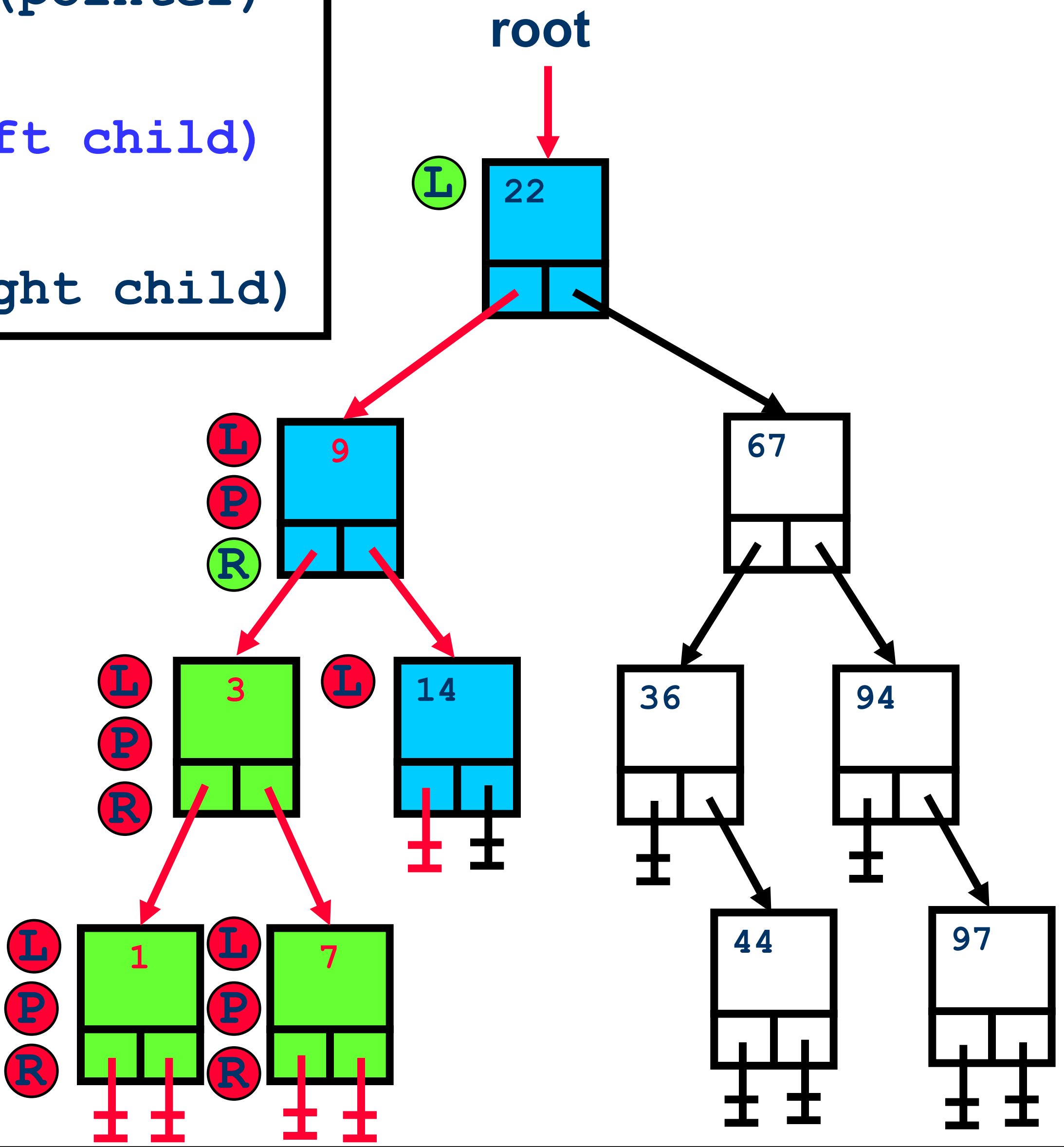
```
Output: 1 3 7 9
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

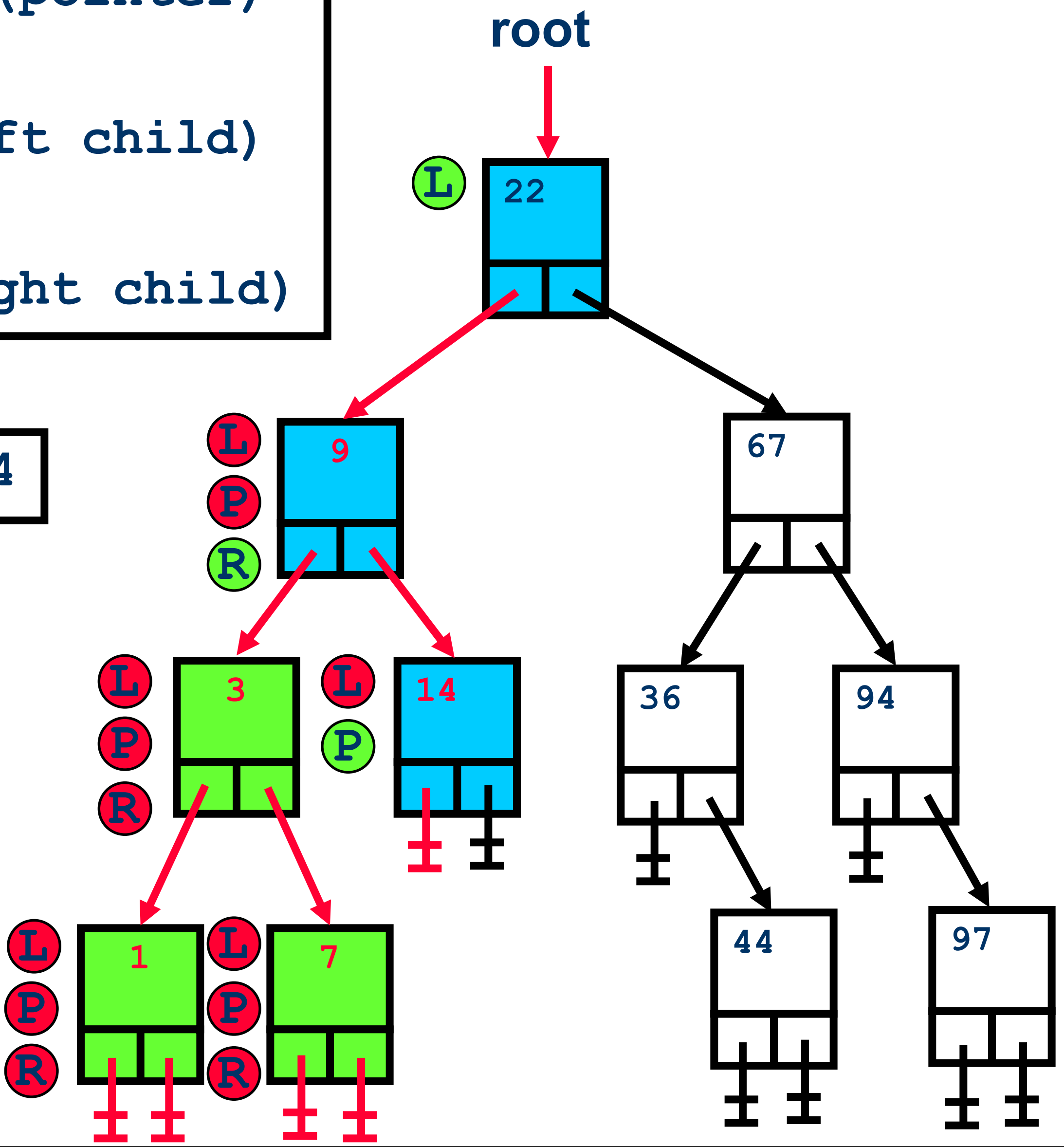
```
Output: 1 3 7 9
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

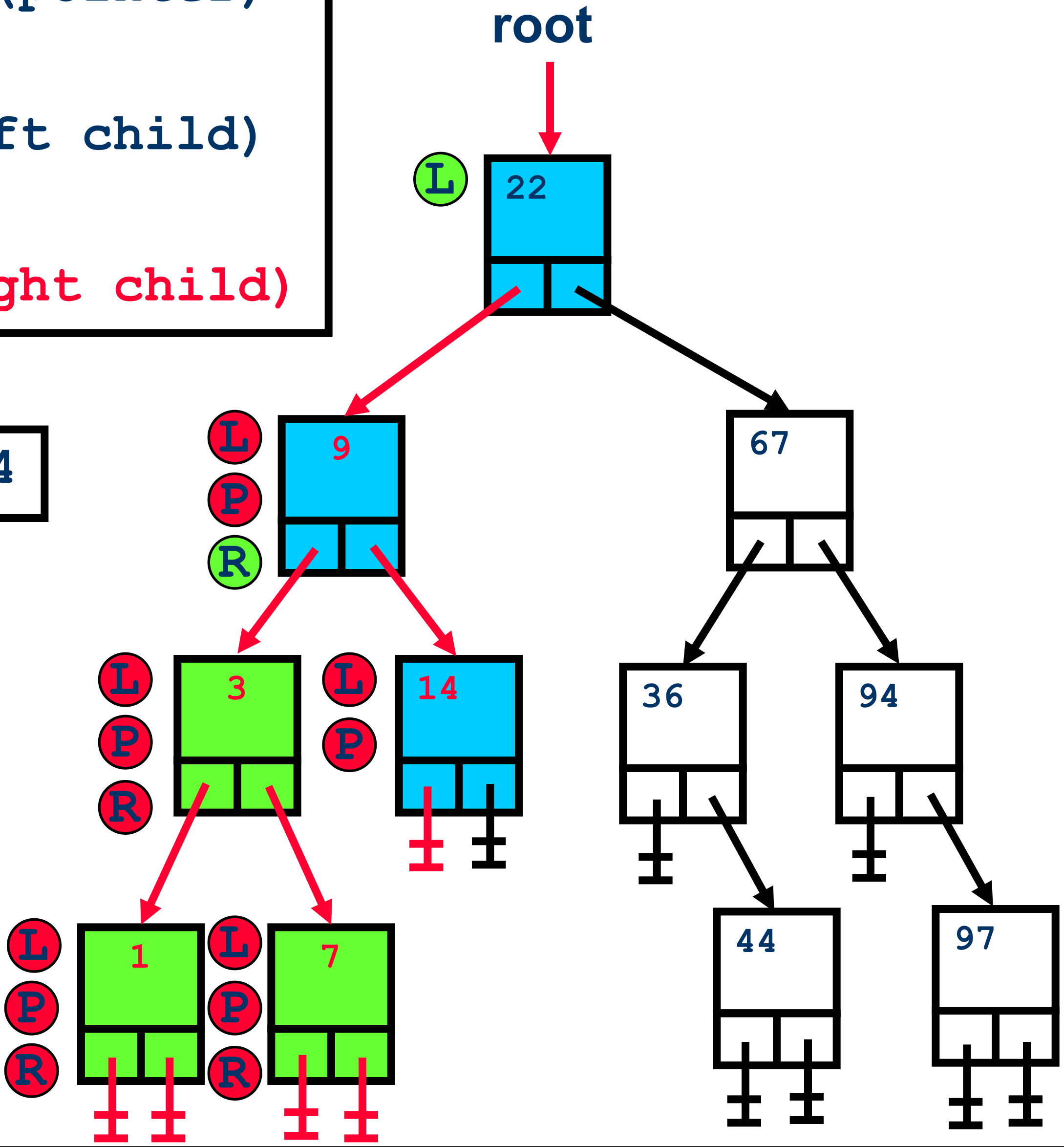
```
Output: 1 3 7 9 14
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

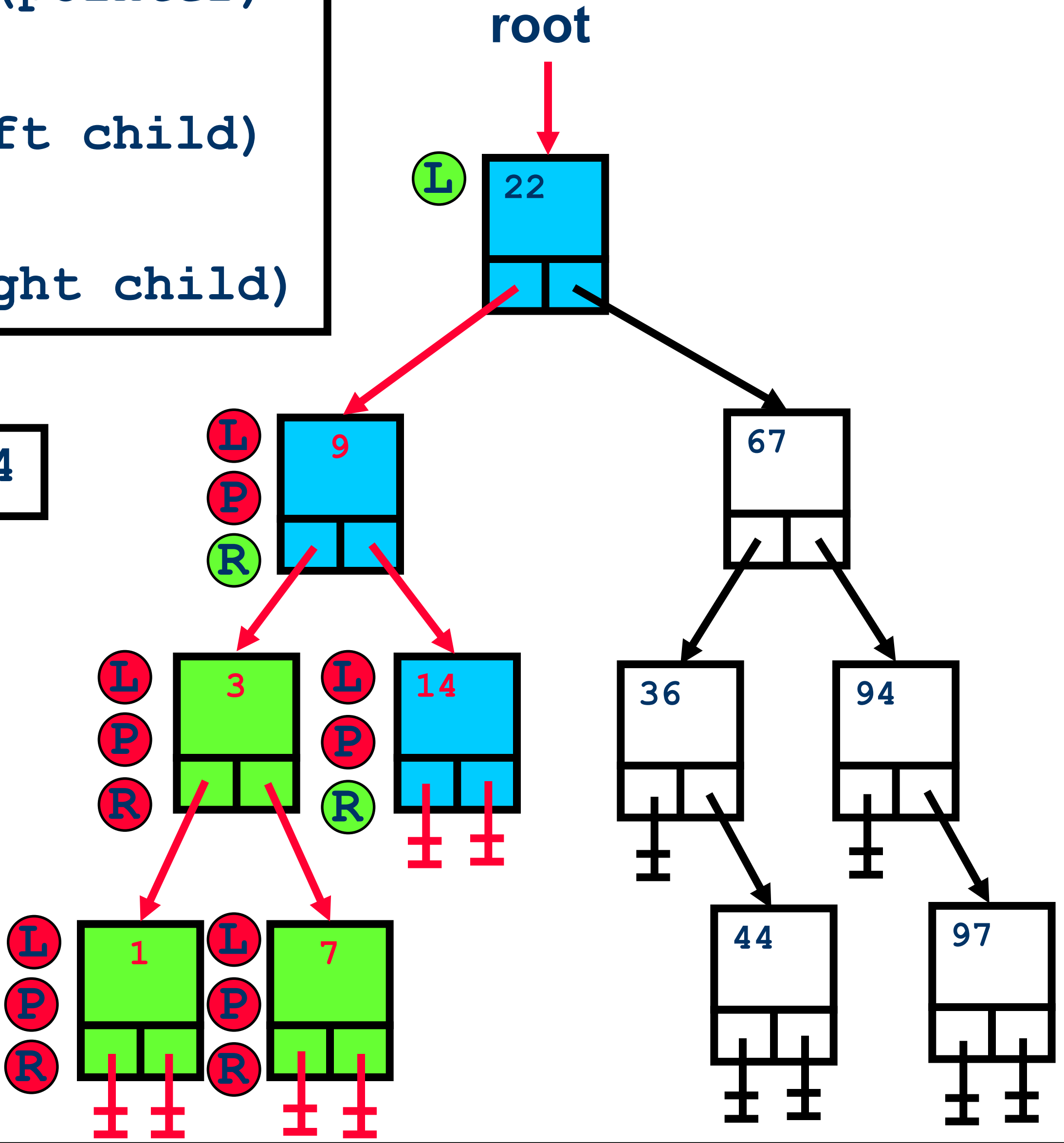
```
Output: 1 3 7 9 14
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

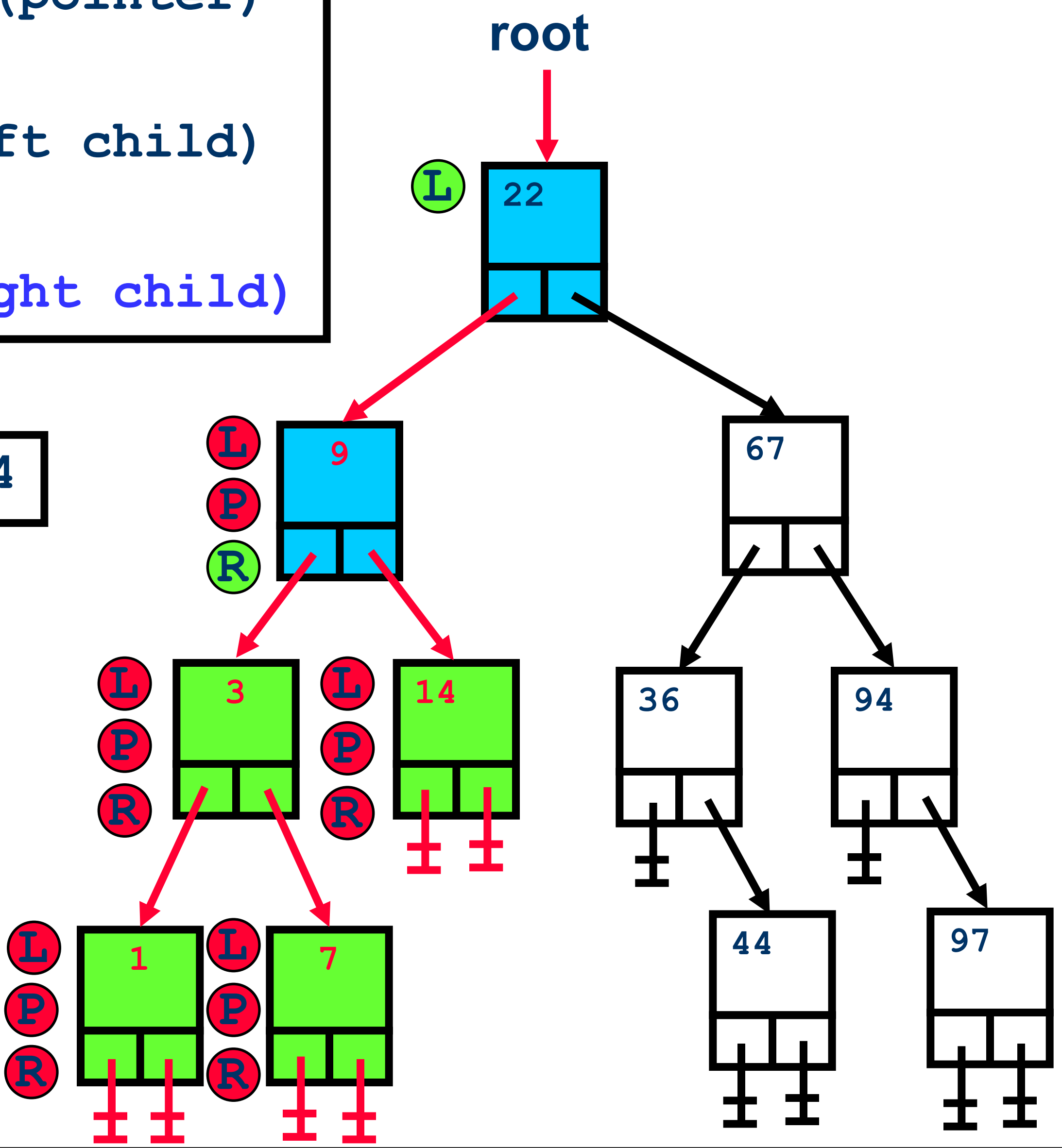
```
Output: 1 3 7 9 14
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

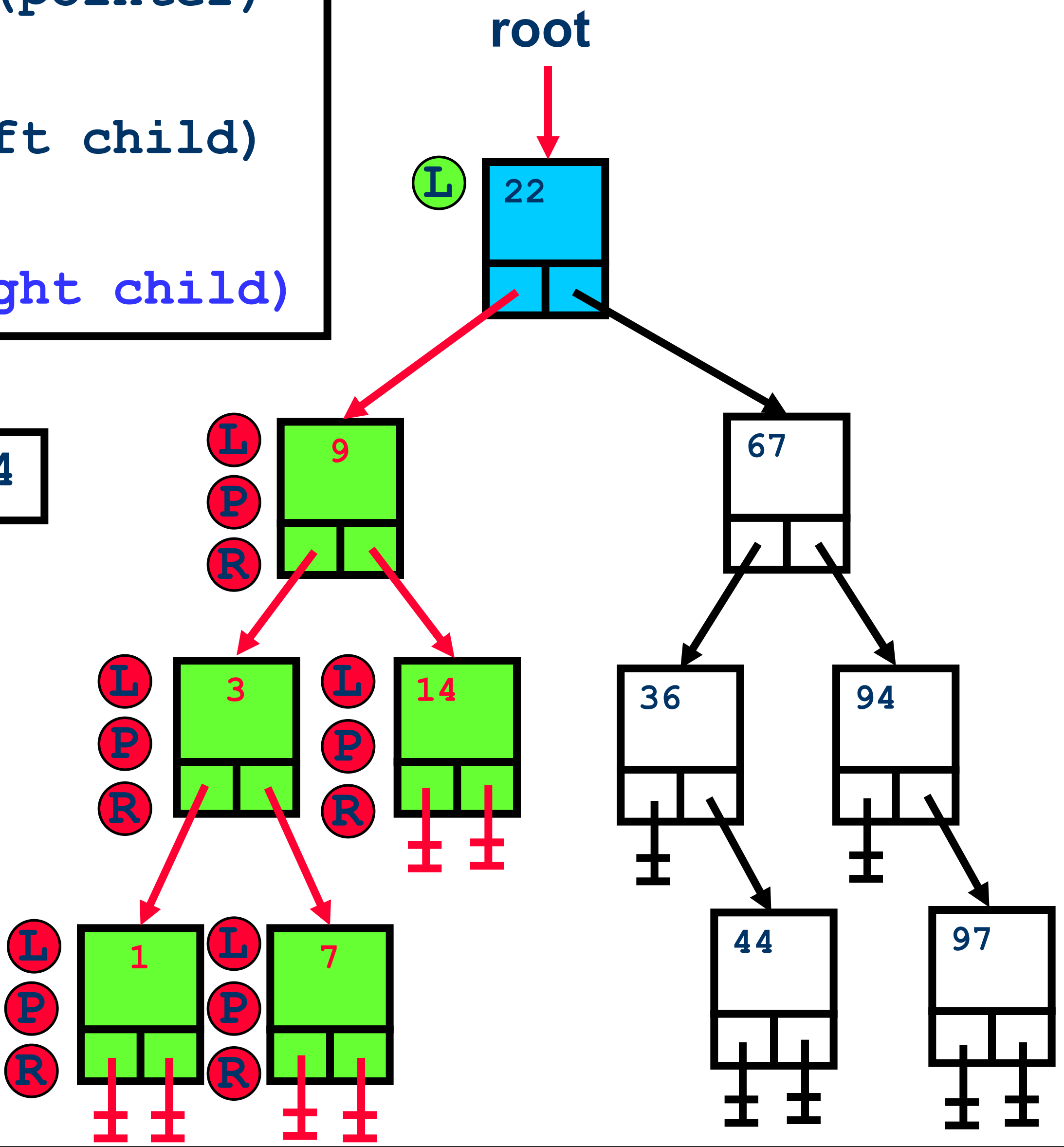
```
Output: 1 3 7 9 14
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

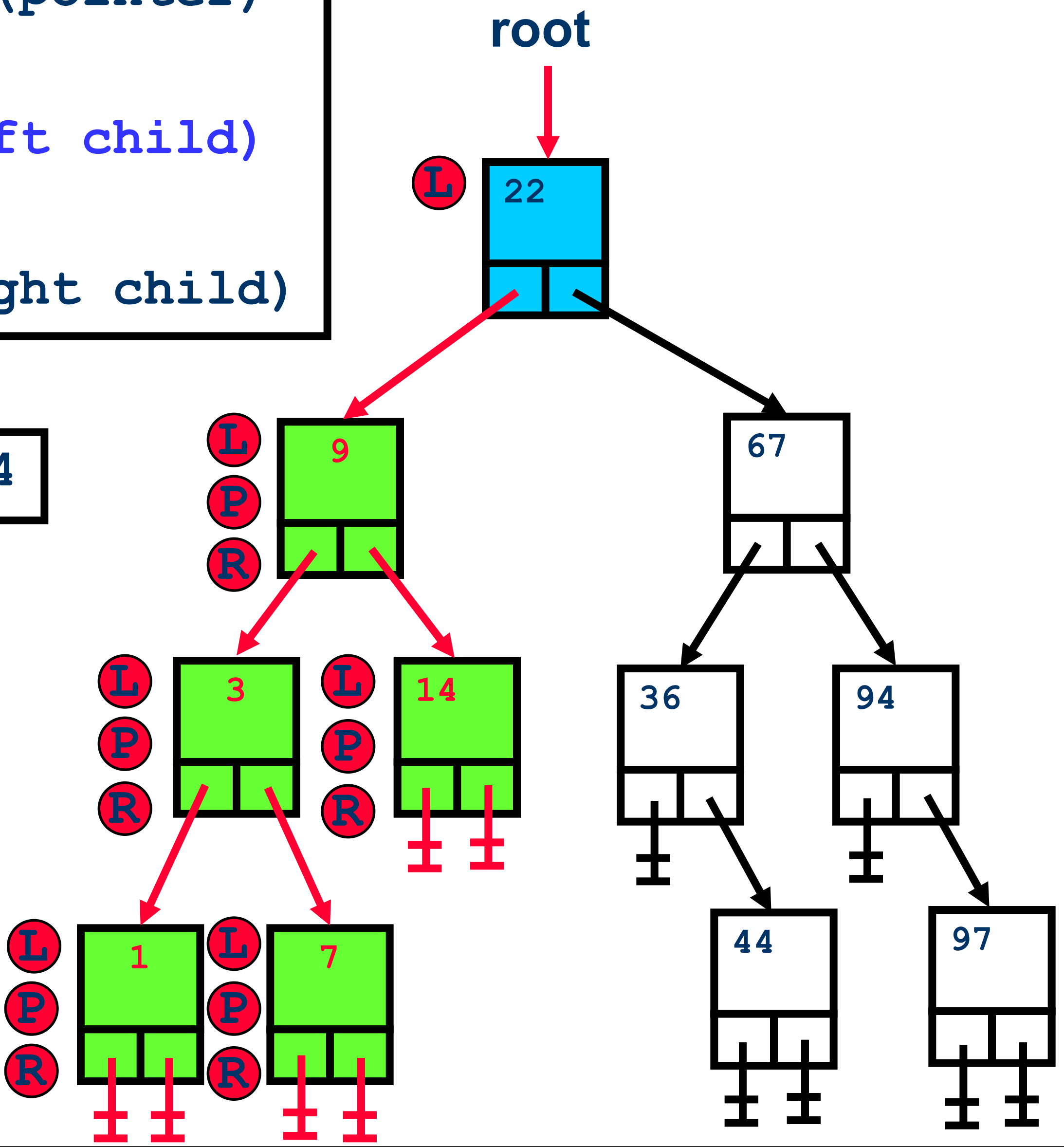
```
Output: 1 3 7 9 14
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

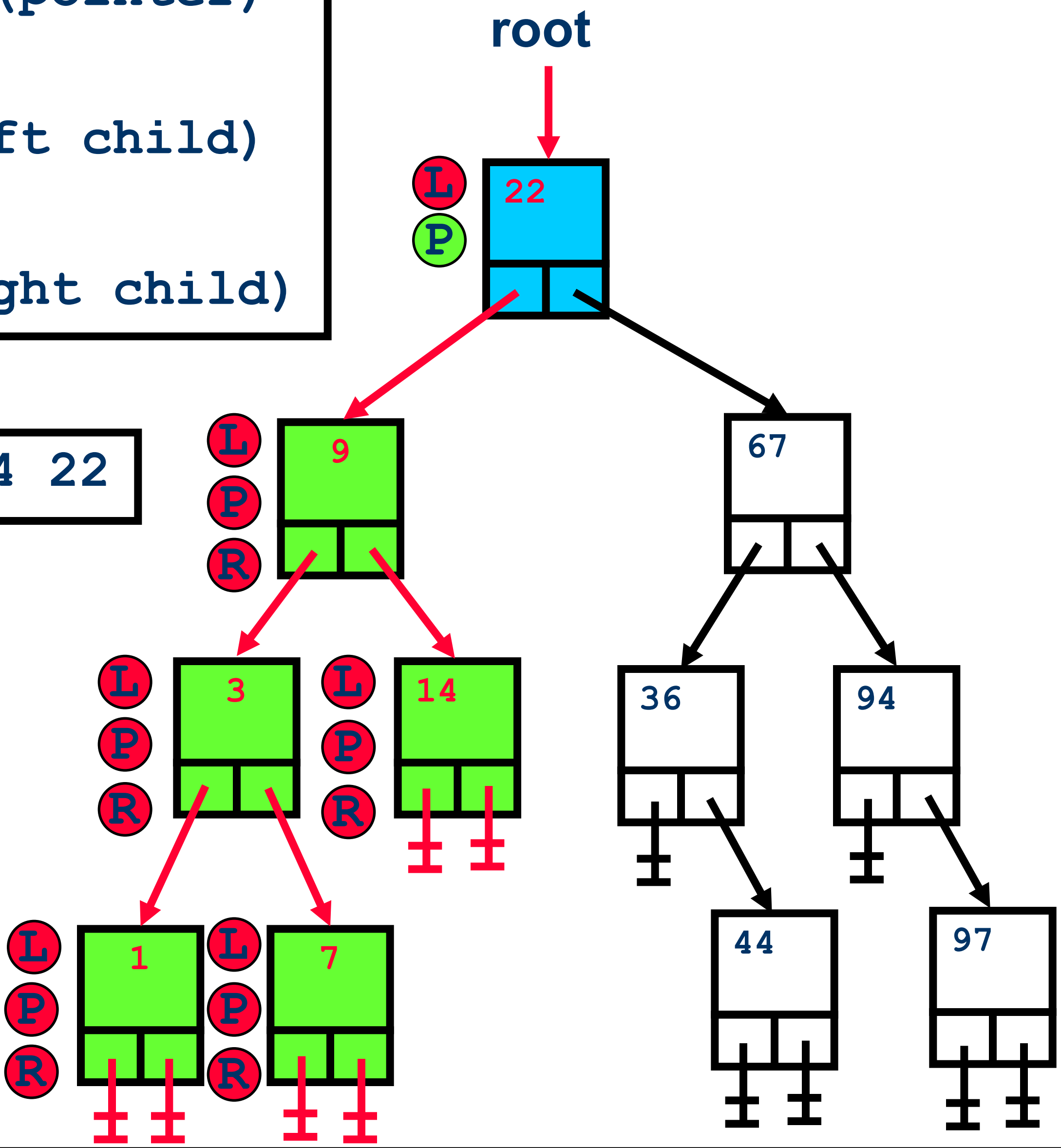
```
Output: 1 3 7 9 14
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

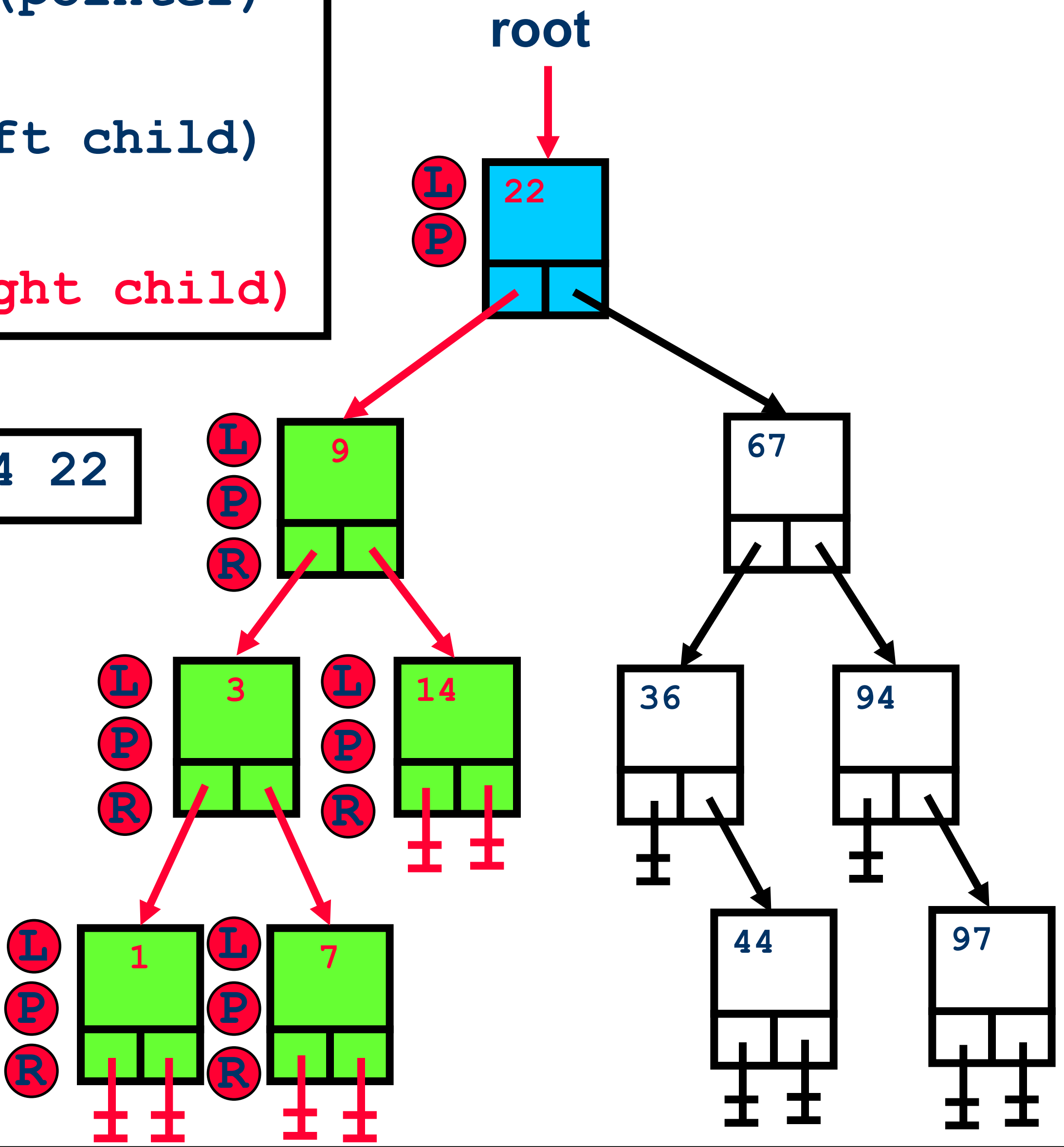
```
Output: 1 3 7 9 14 22
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

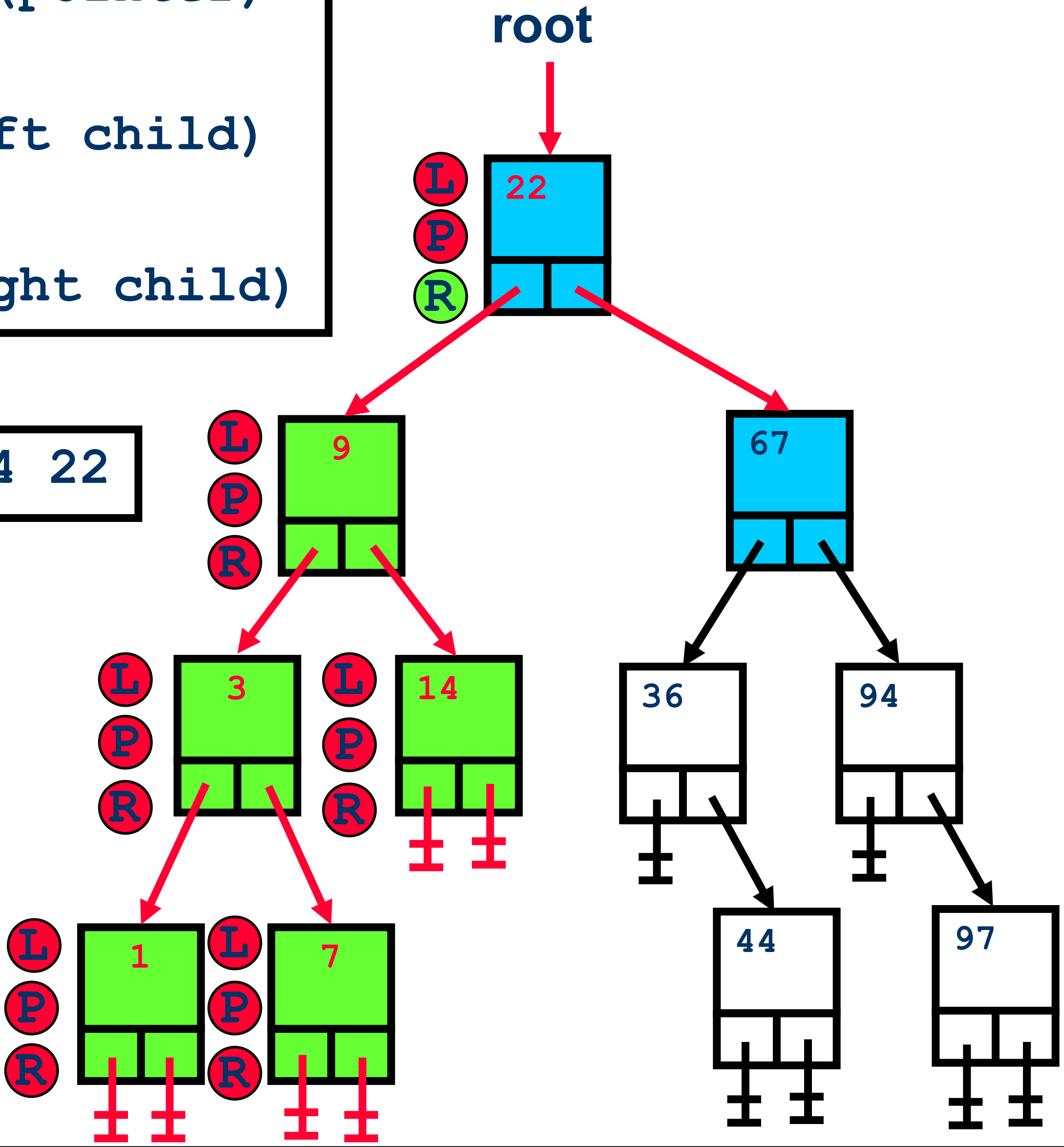
```
Output: 1 3 7 9 14 22
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

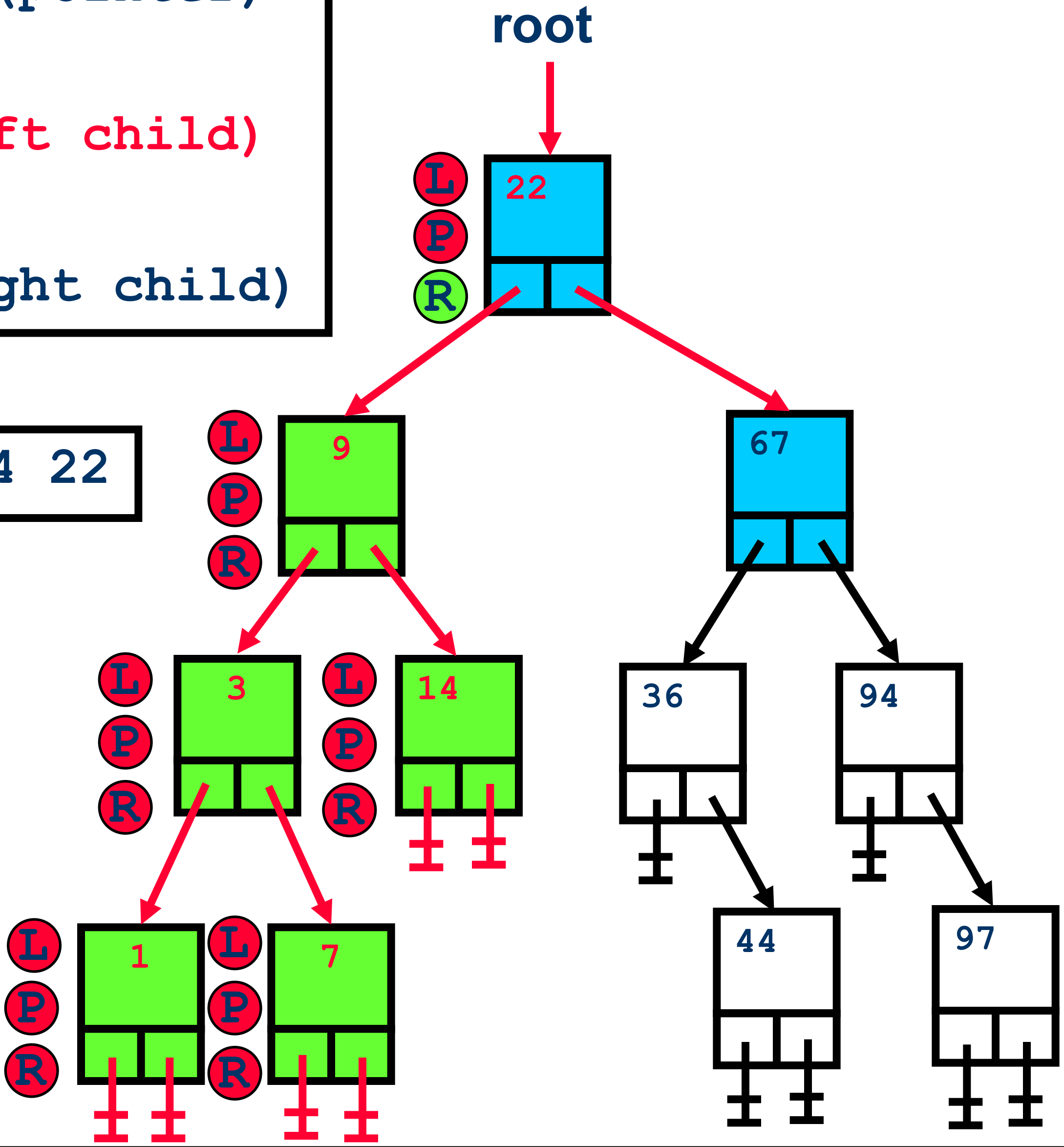
```
Output: 1 3 7 9 14 22
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

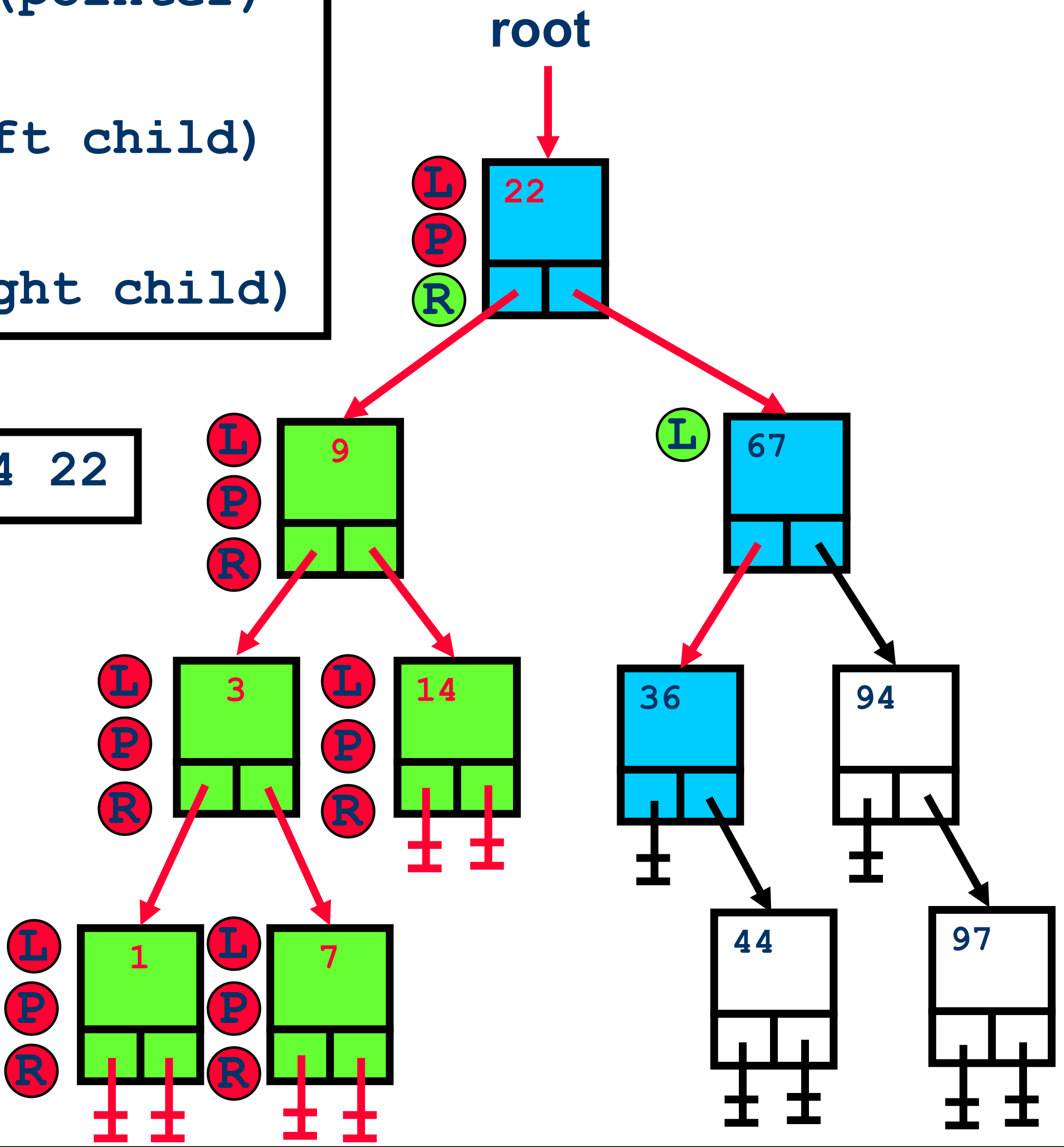
```
Output: 1 3 7 9 14 22
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

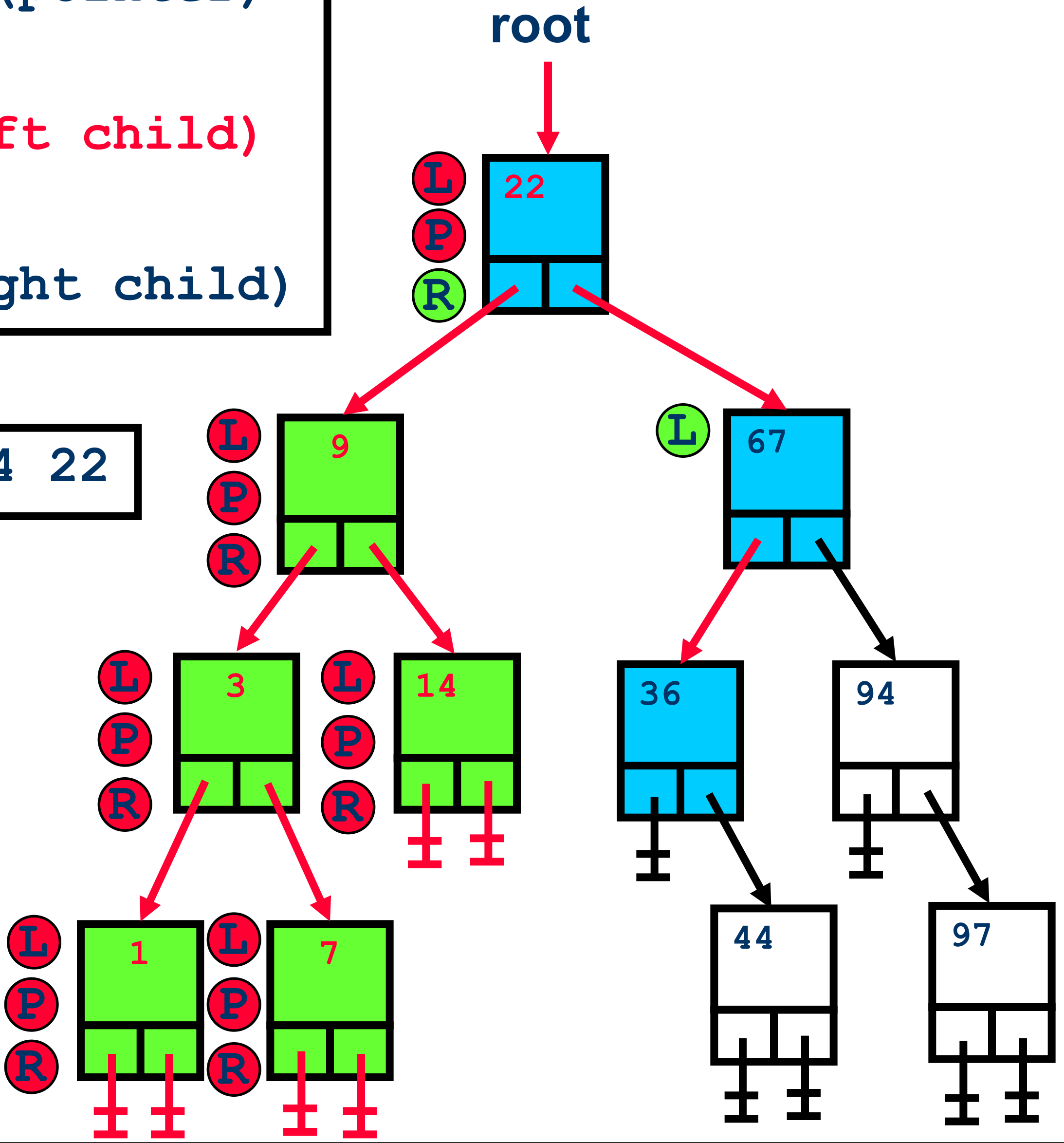
```
Output: 1 3 7 9 14 22
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

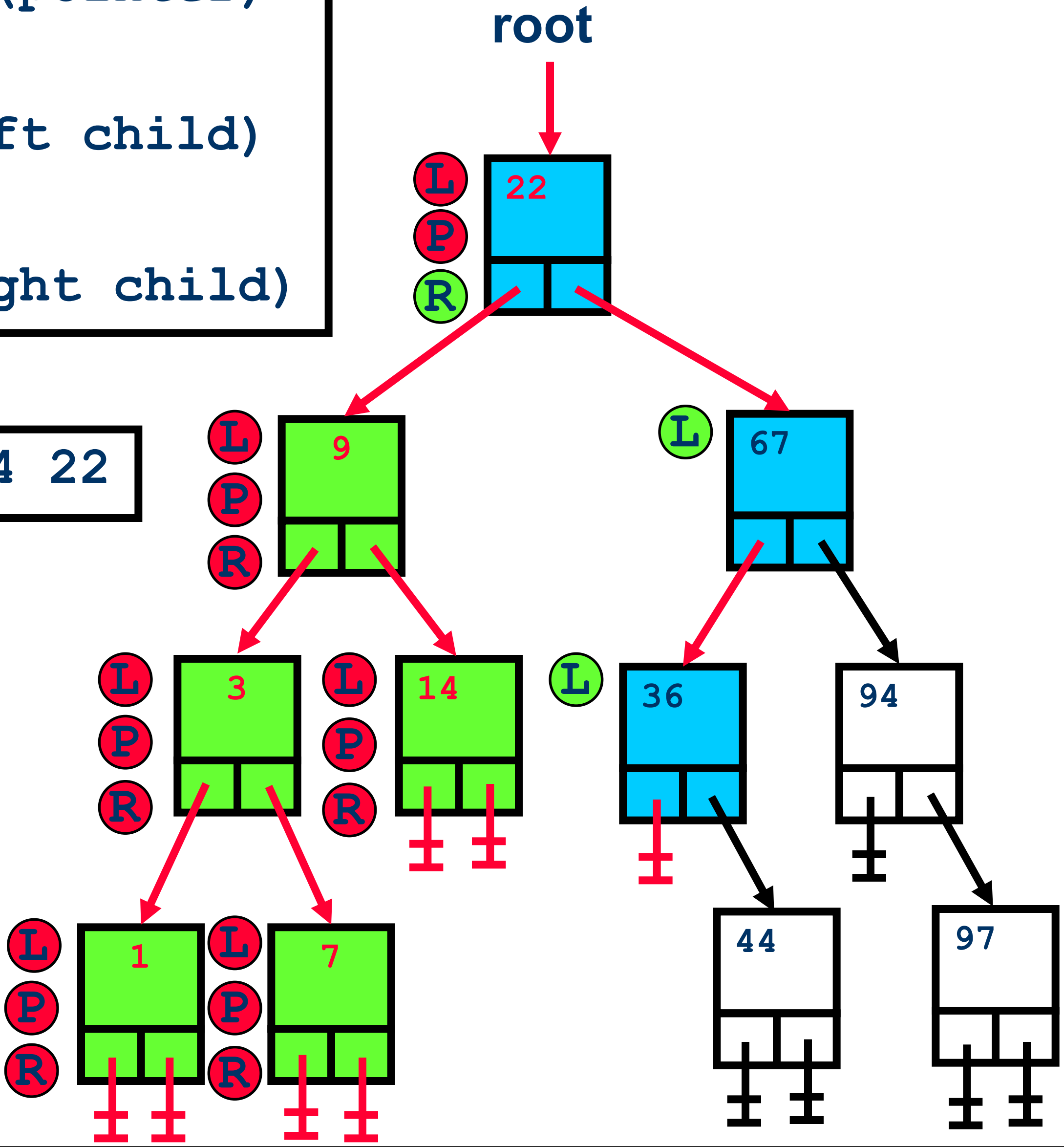
```
Output: 1 3 7 9 14 22
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

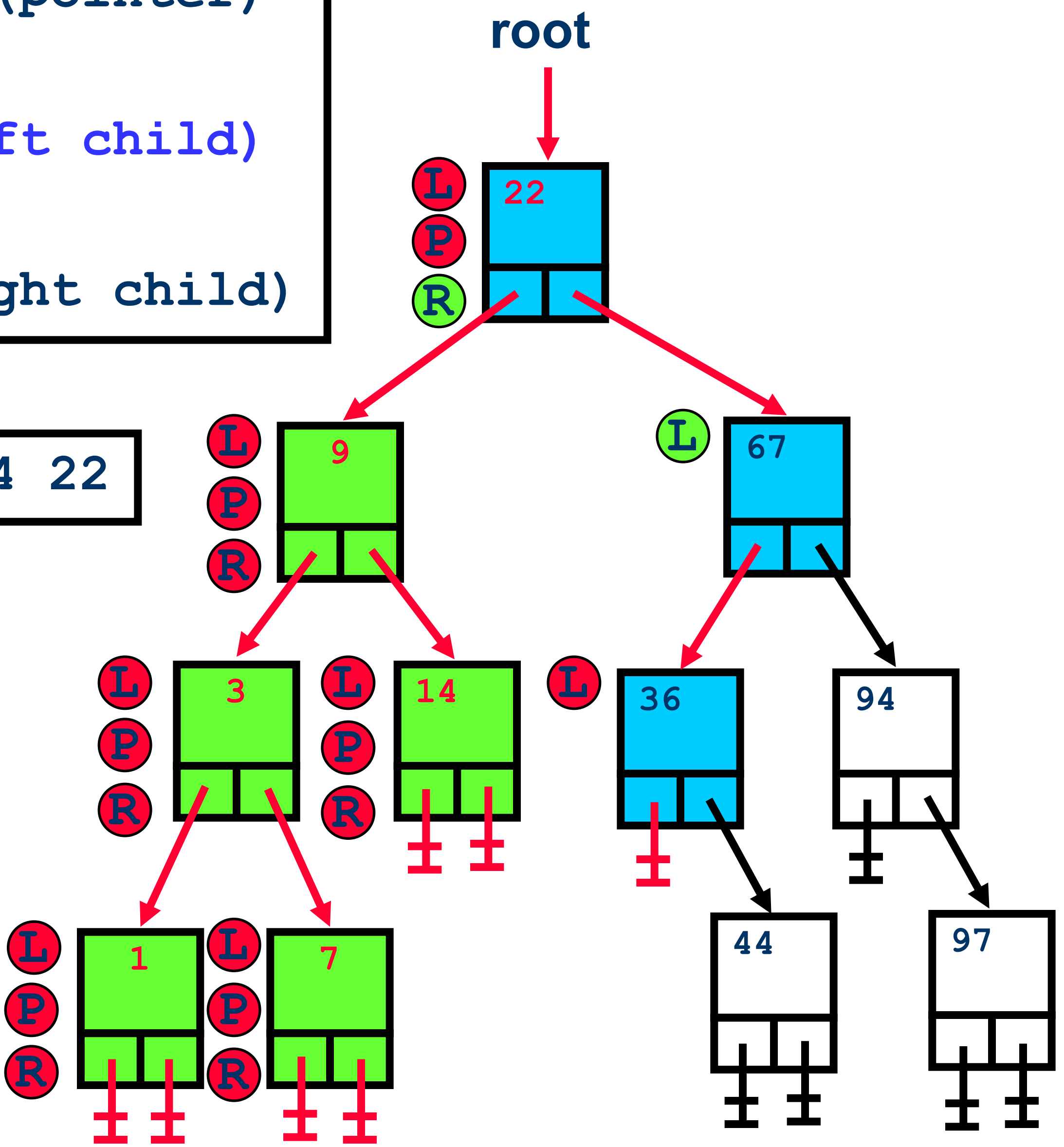
```
Output: 1 3 7 9 14 22
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

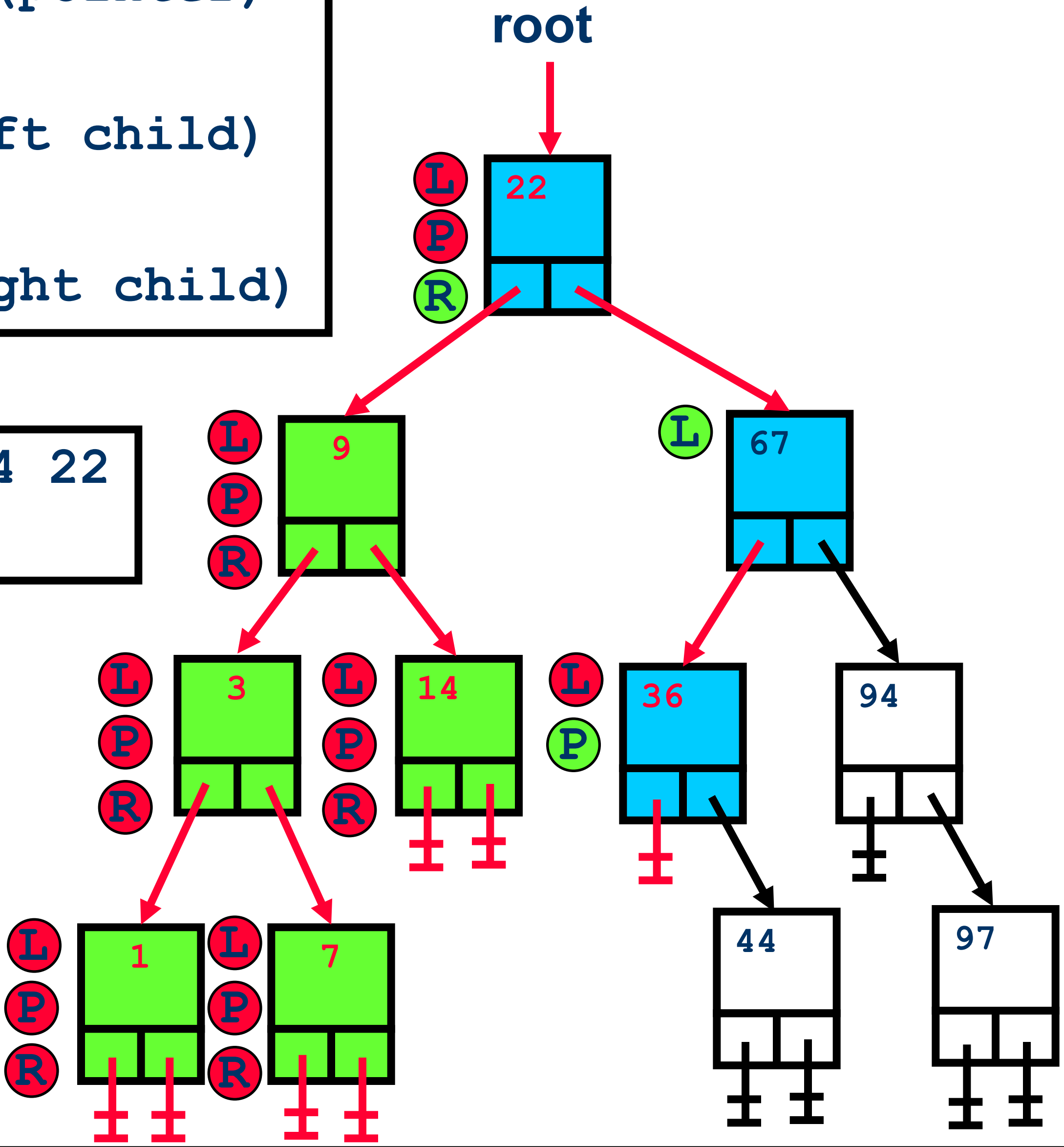
```
Output: 1 3 7 9 14 22
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

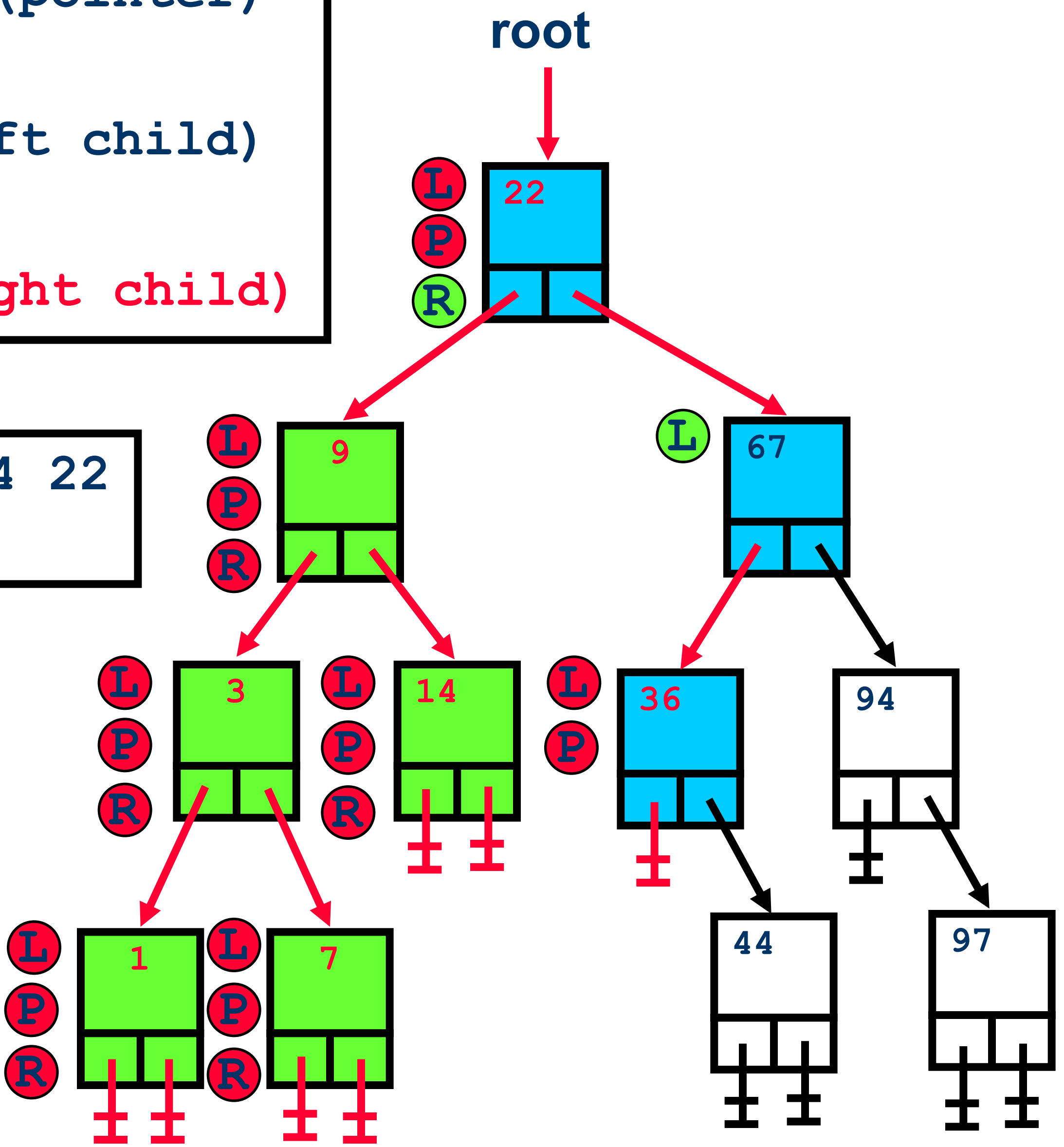
```
Output: 1 3 7 9 14 22
        36
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

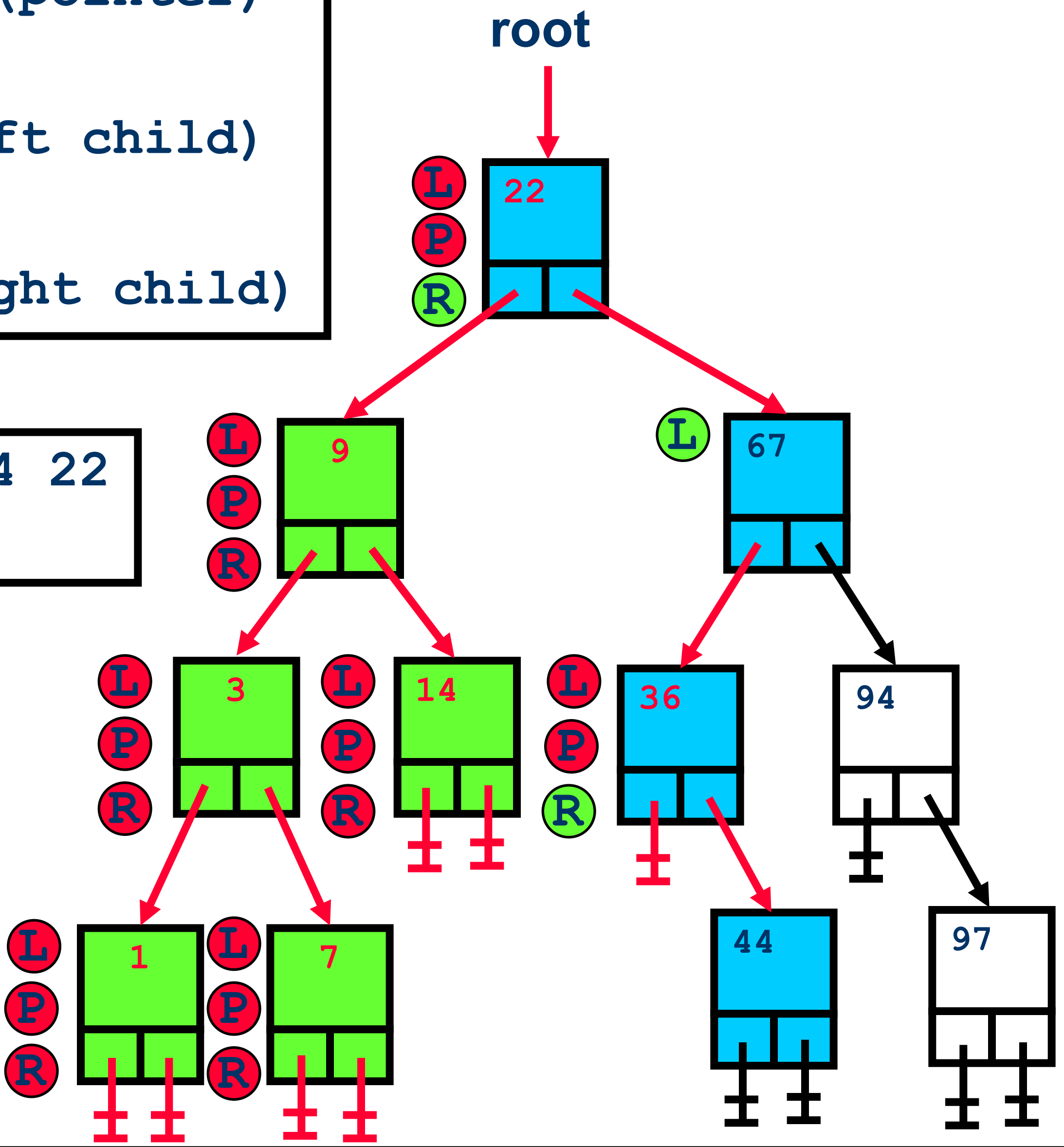
```
Output: 1 3 7 9 14 22
        36
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

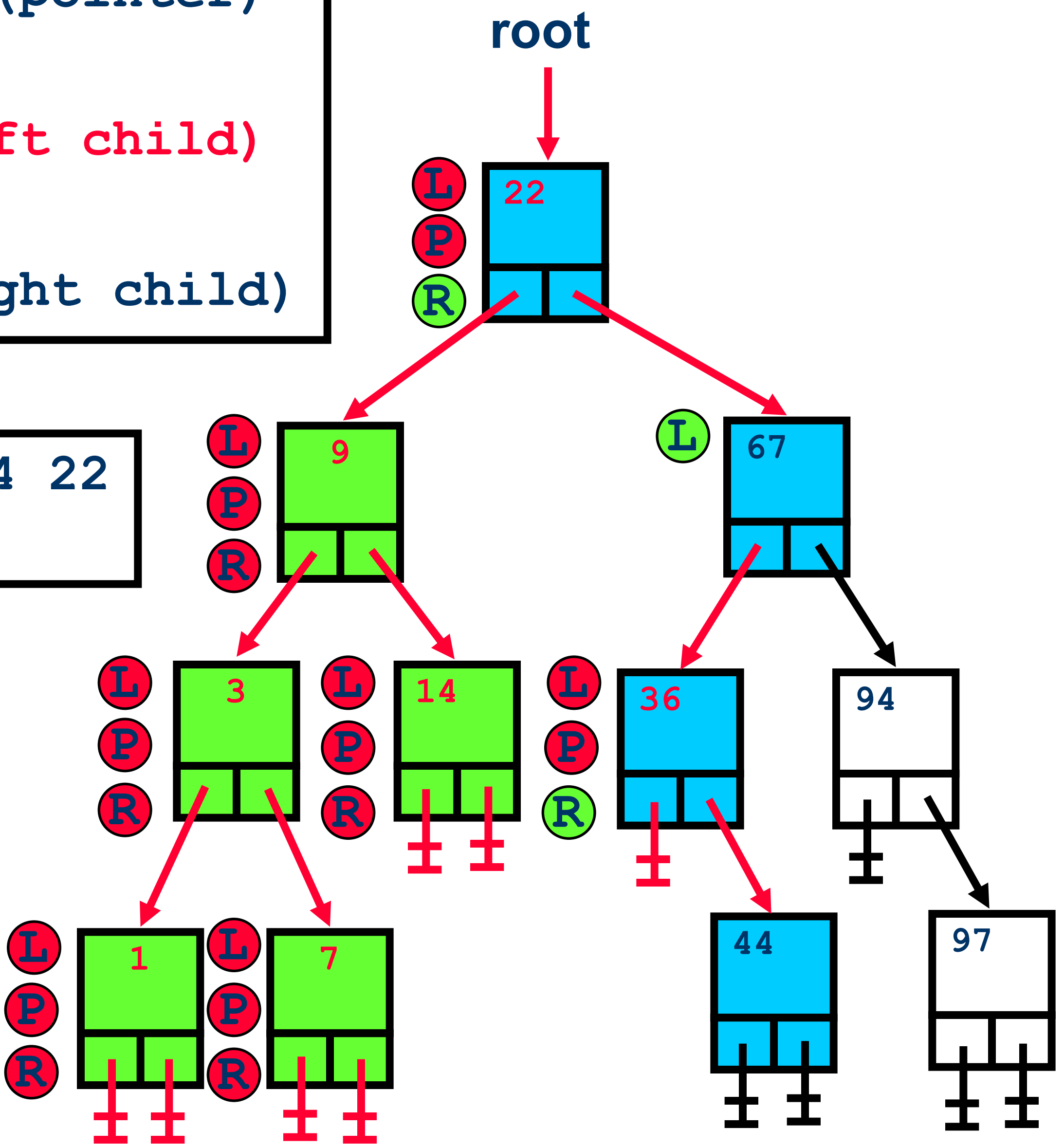
```
Output: 1 3 7 9 14 22
        36
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

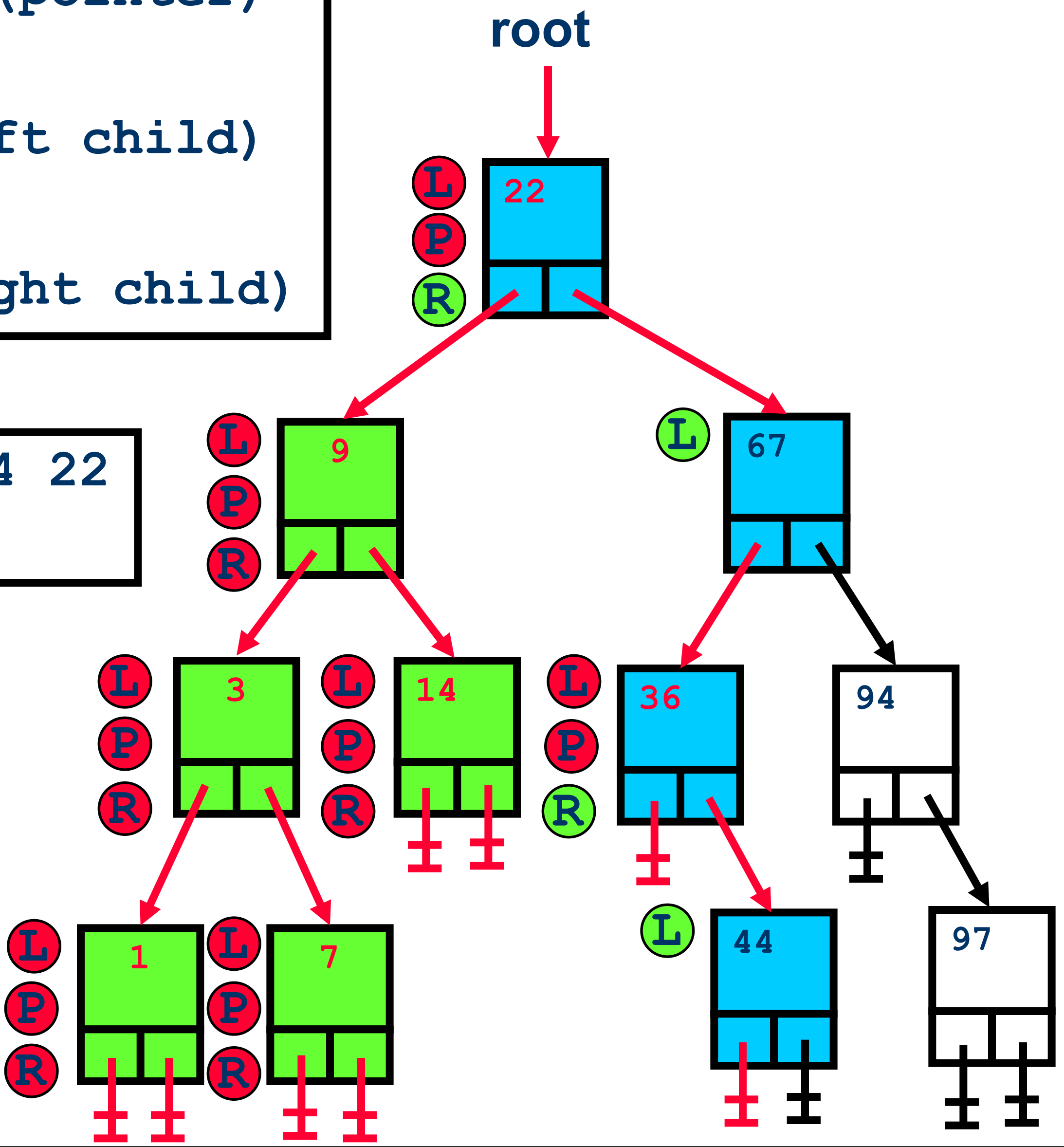
```
Output: 1 3 7 9 14 22
        36
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

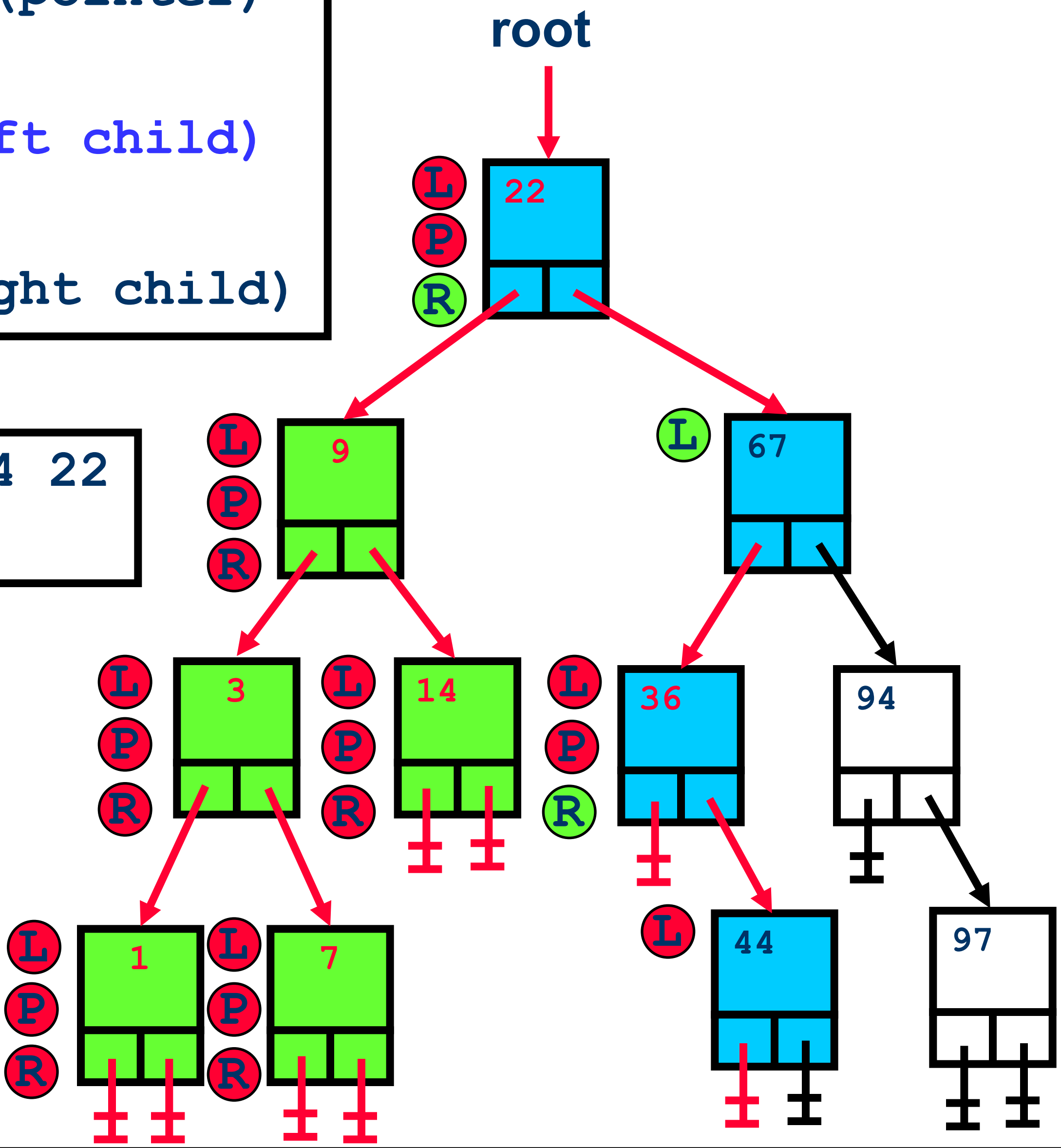
```
Output: 1 3 7 9 14 22
        36
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

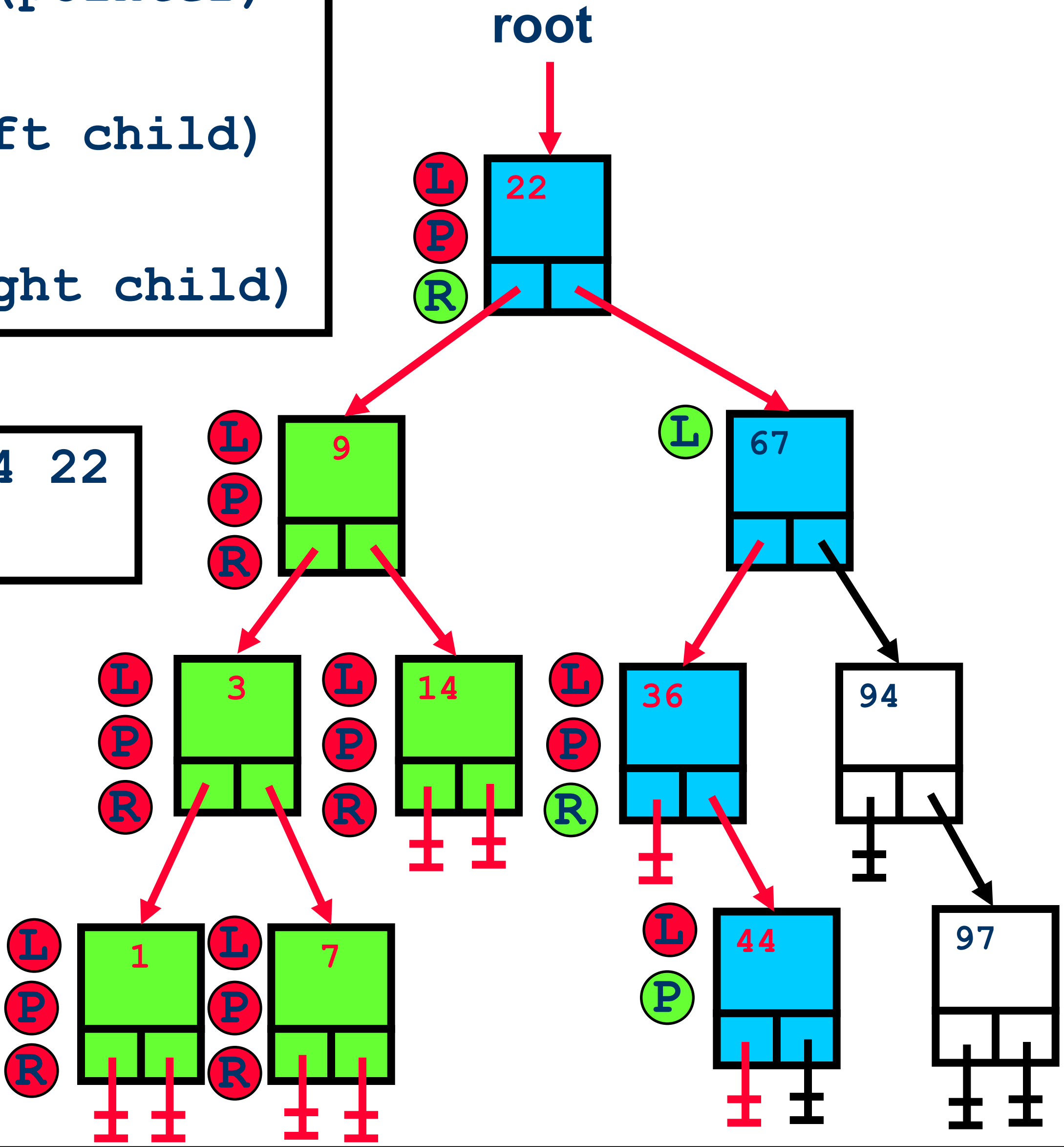
```
Output: 1 3 7 9 14 22
        36
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

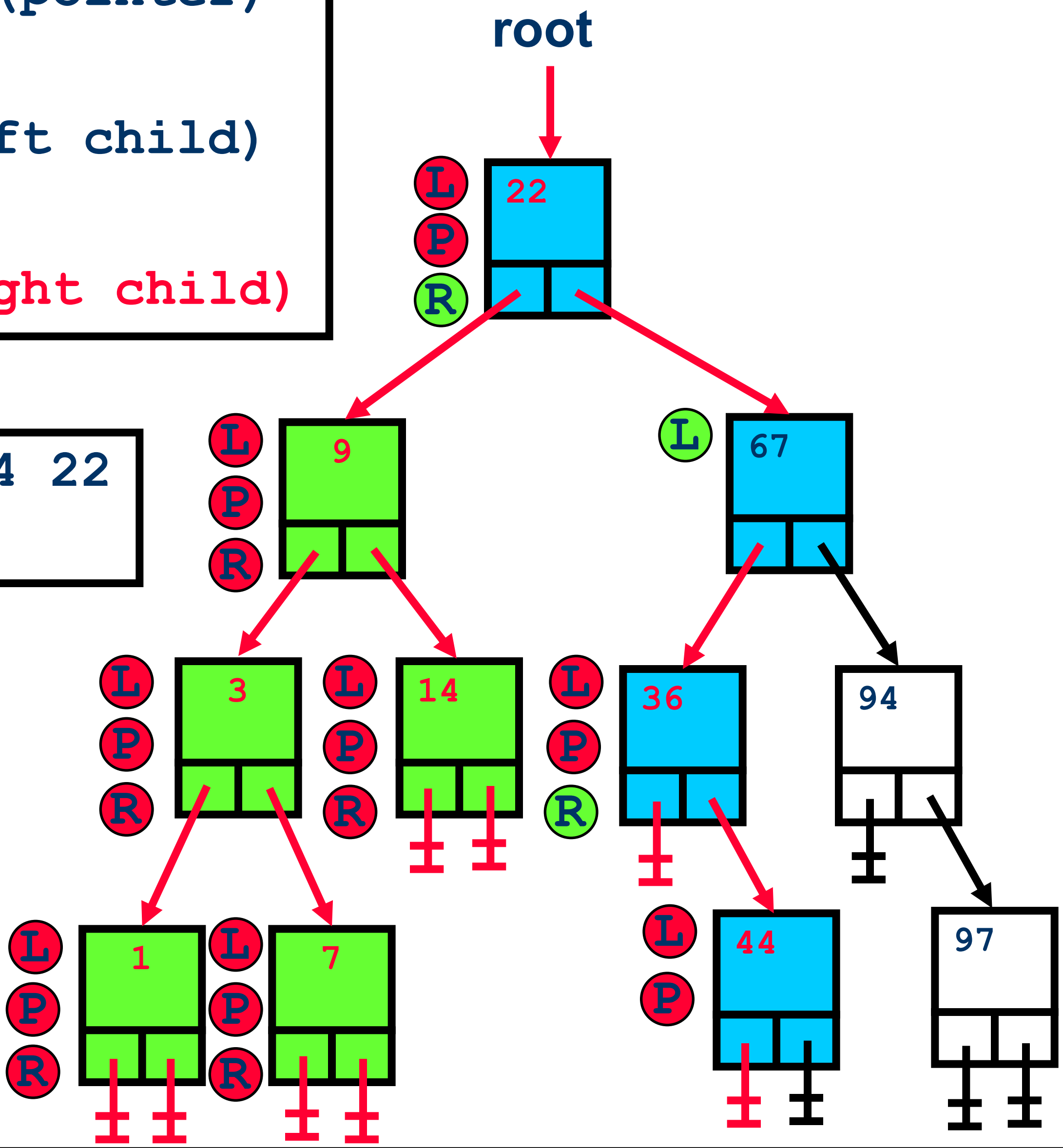
```
Output: 1 3 7 9 14 22
        36 44
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

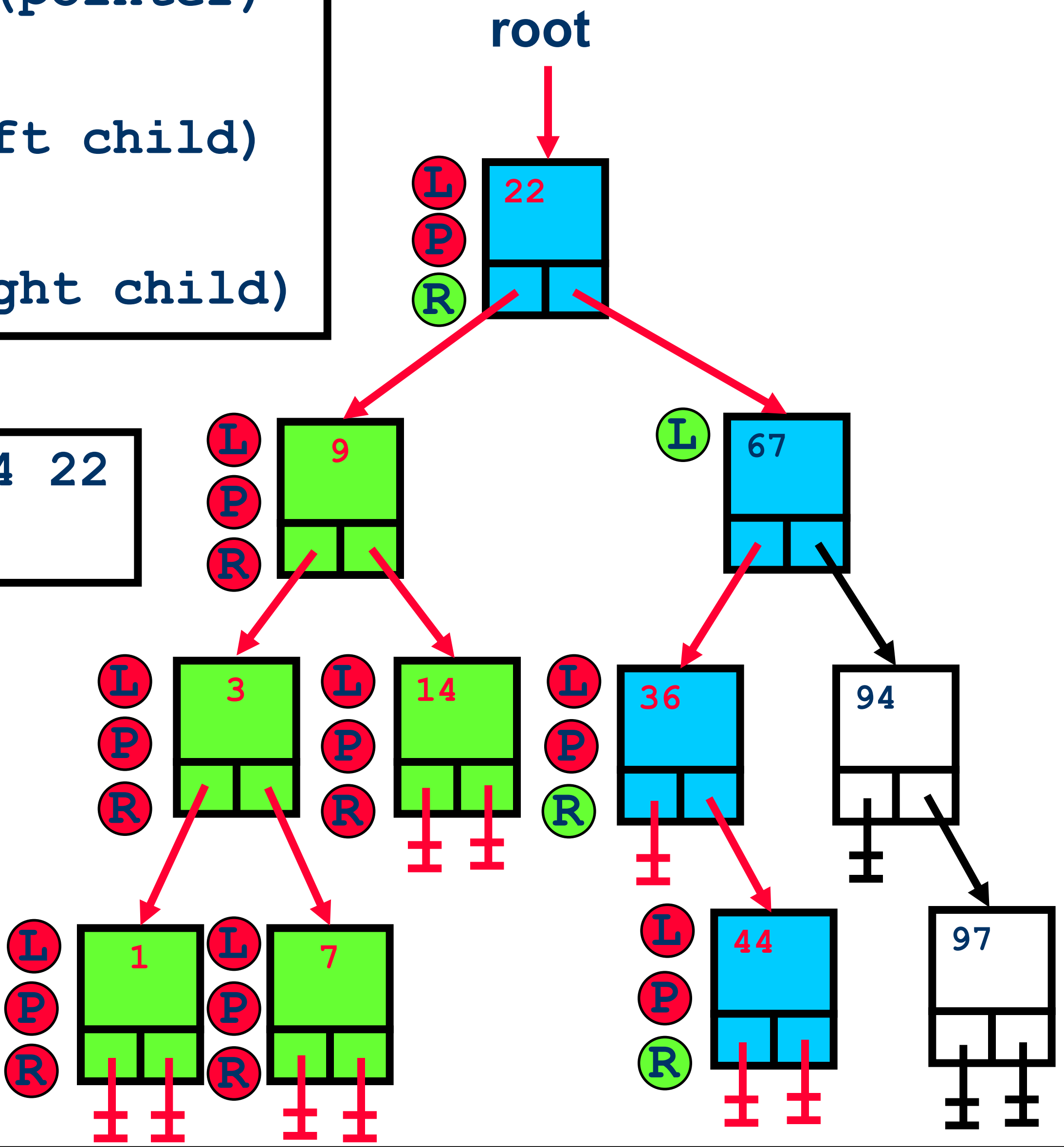
```
Output: 1 3 7 9 14 22
        36 44
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

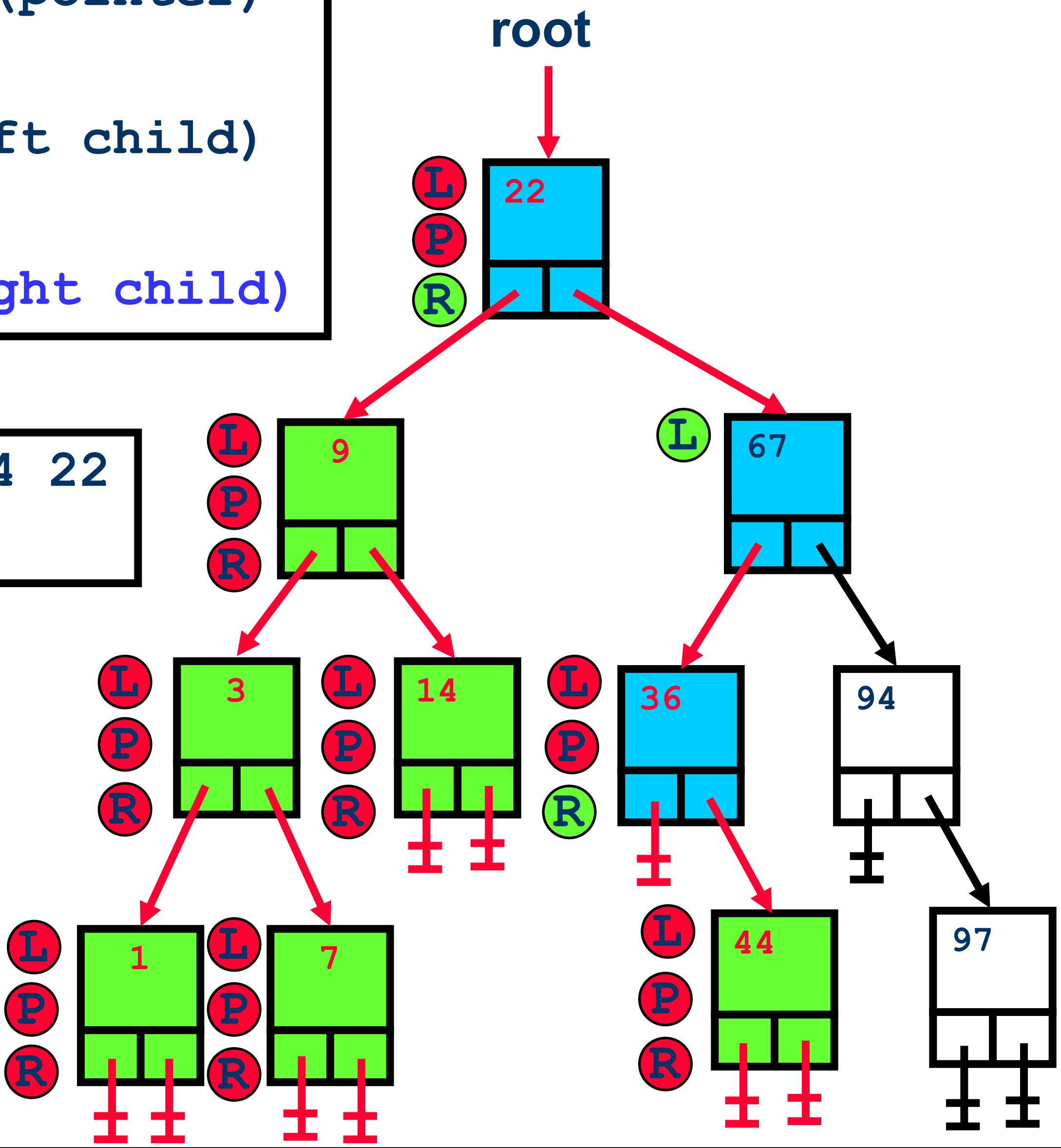
```
Output: 1 3 7 9 14 22
        36 44
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

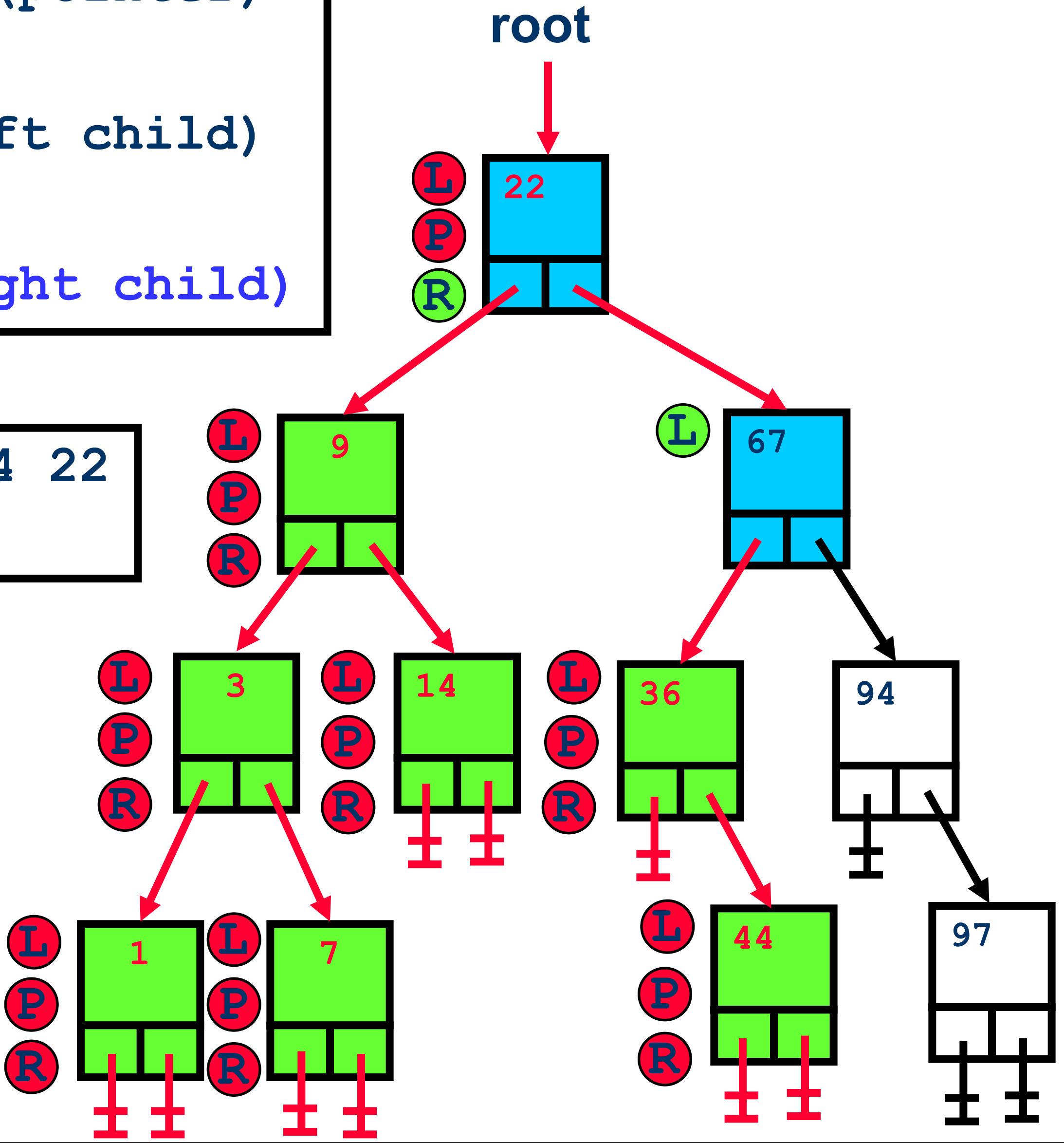
```
Output: 1 3 7 9 14 22
        36 44
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

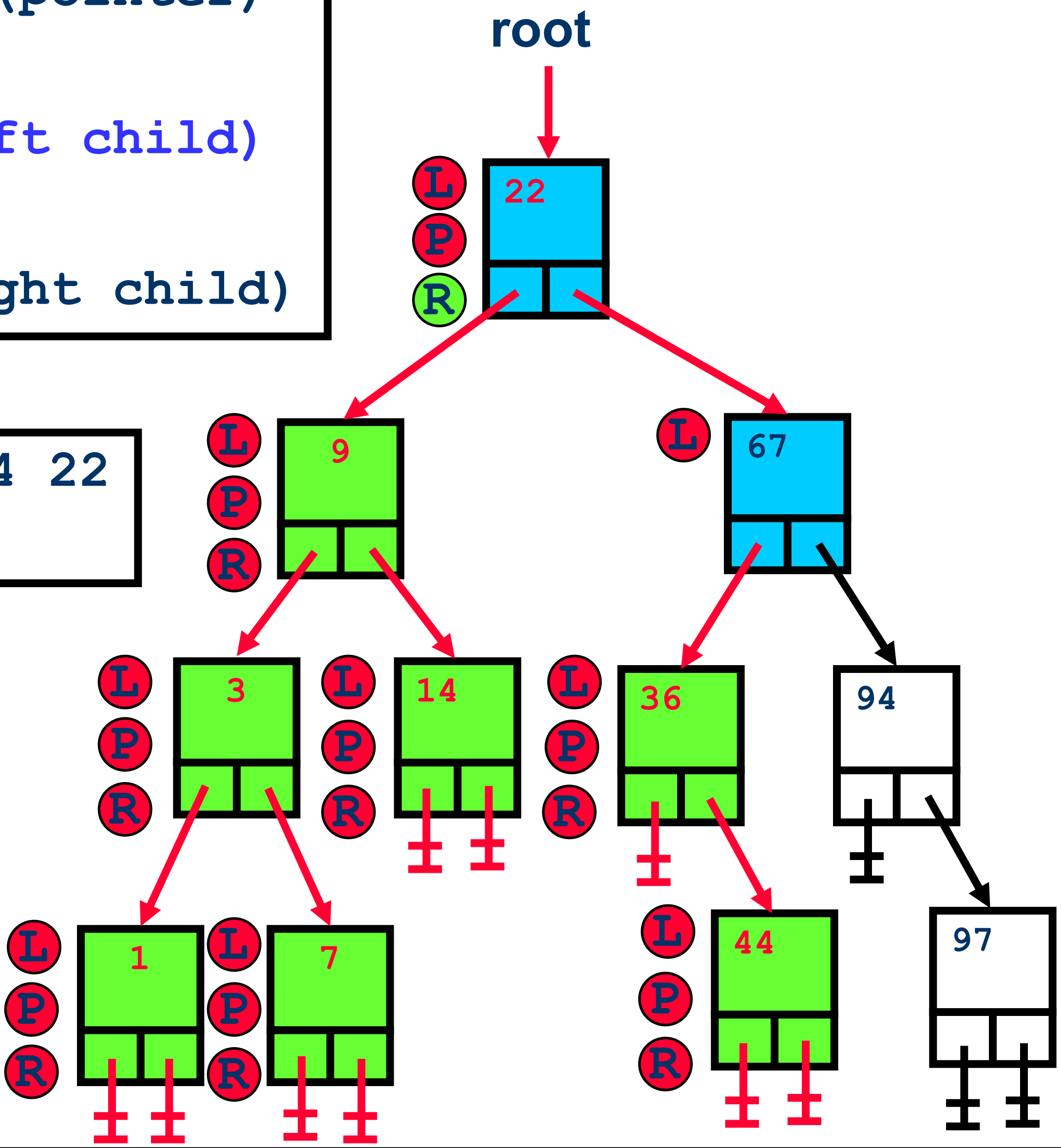
```
Output: 1 3 7 9 14 22
        36 44
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

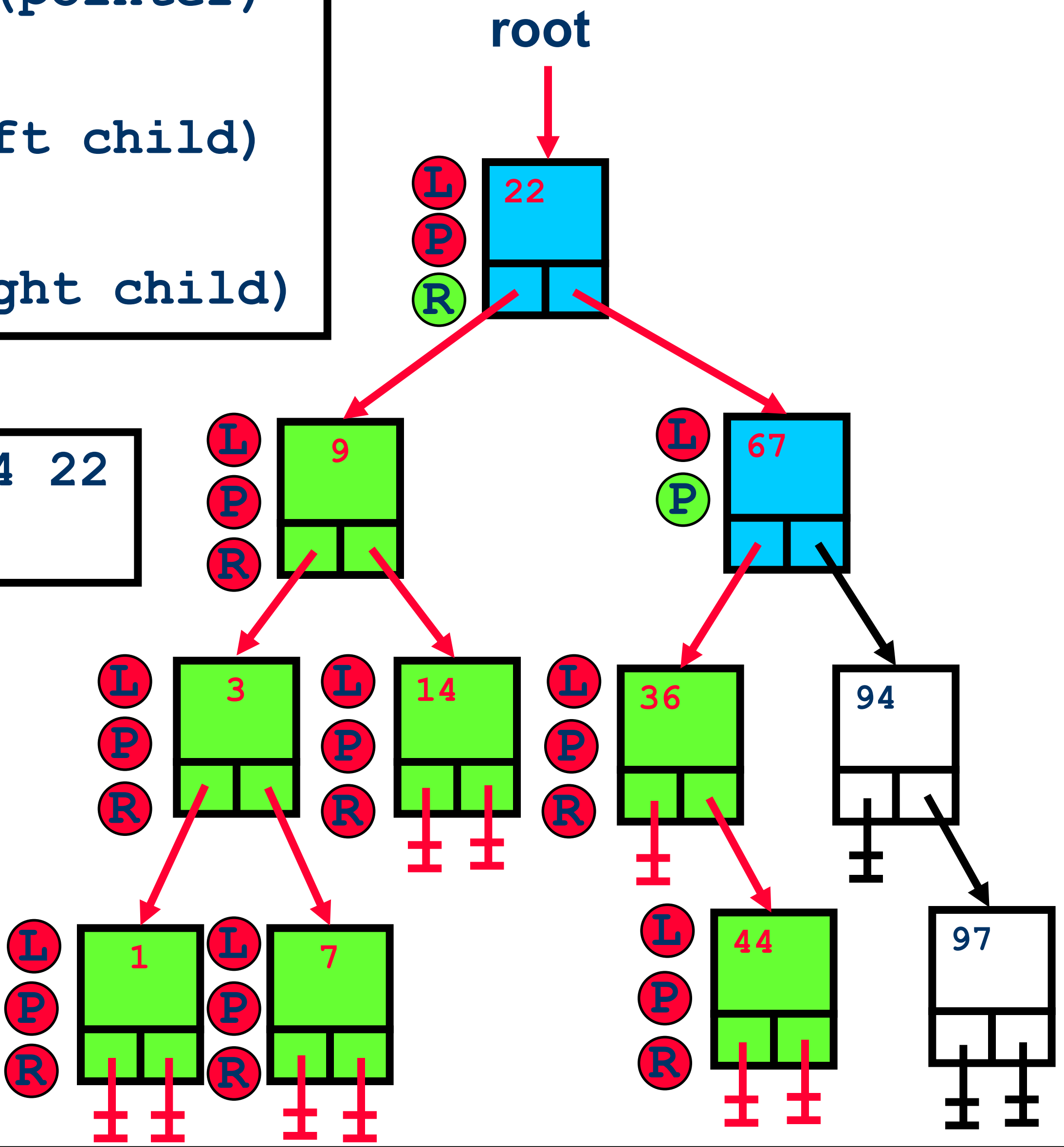
```
Output: 1 3 7 9 14 22
        36 44
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

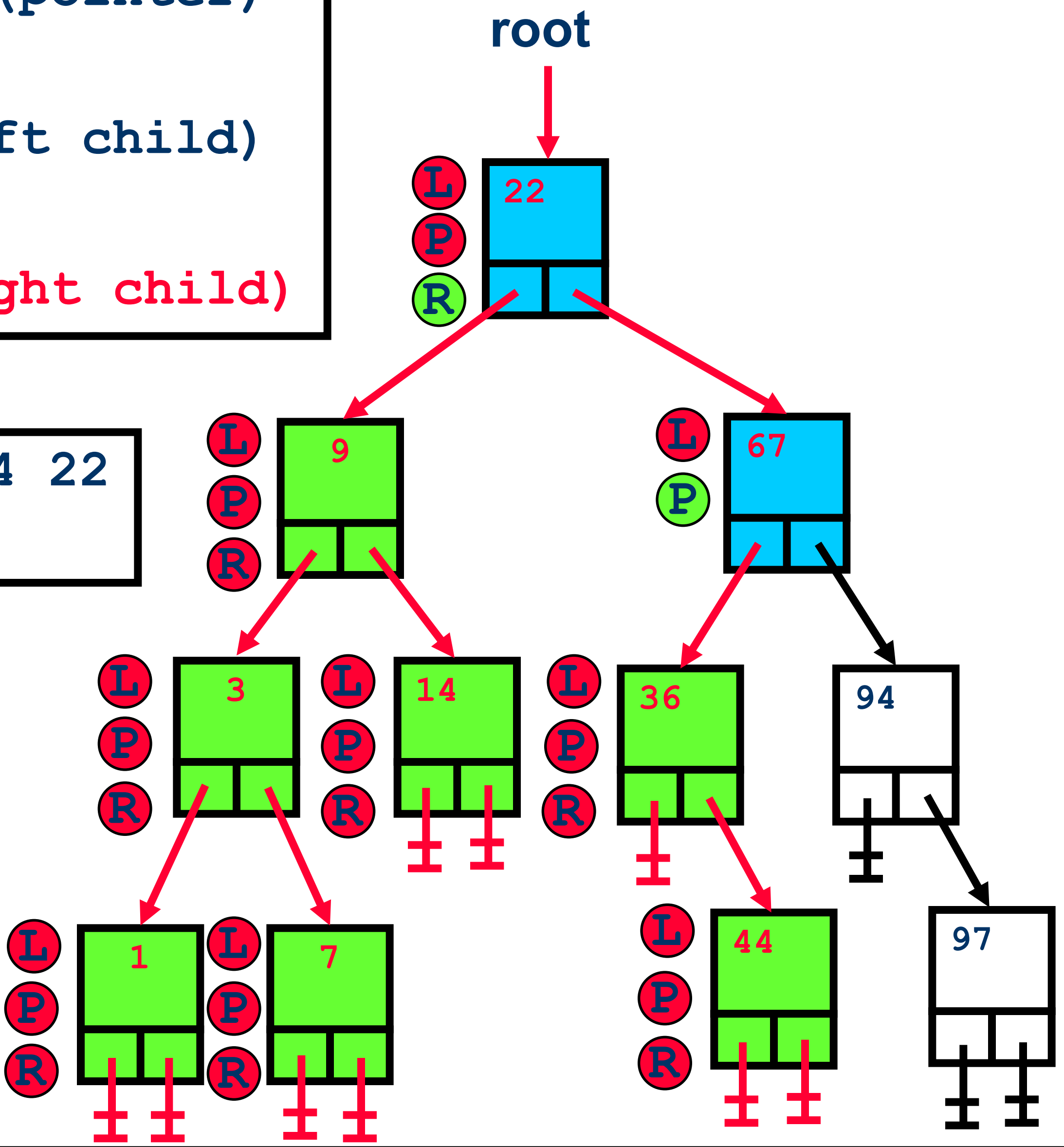
```
Output: 1 3 7 9 14 22
        36 44 67
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

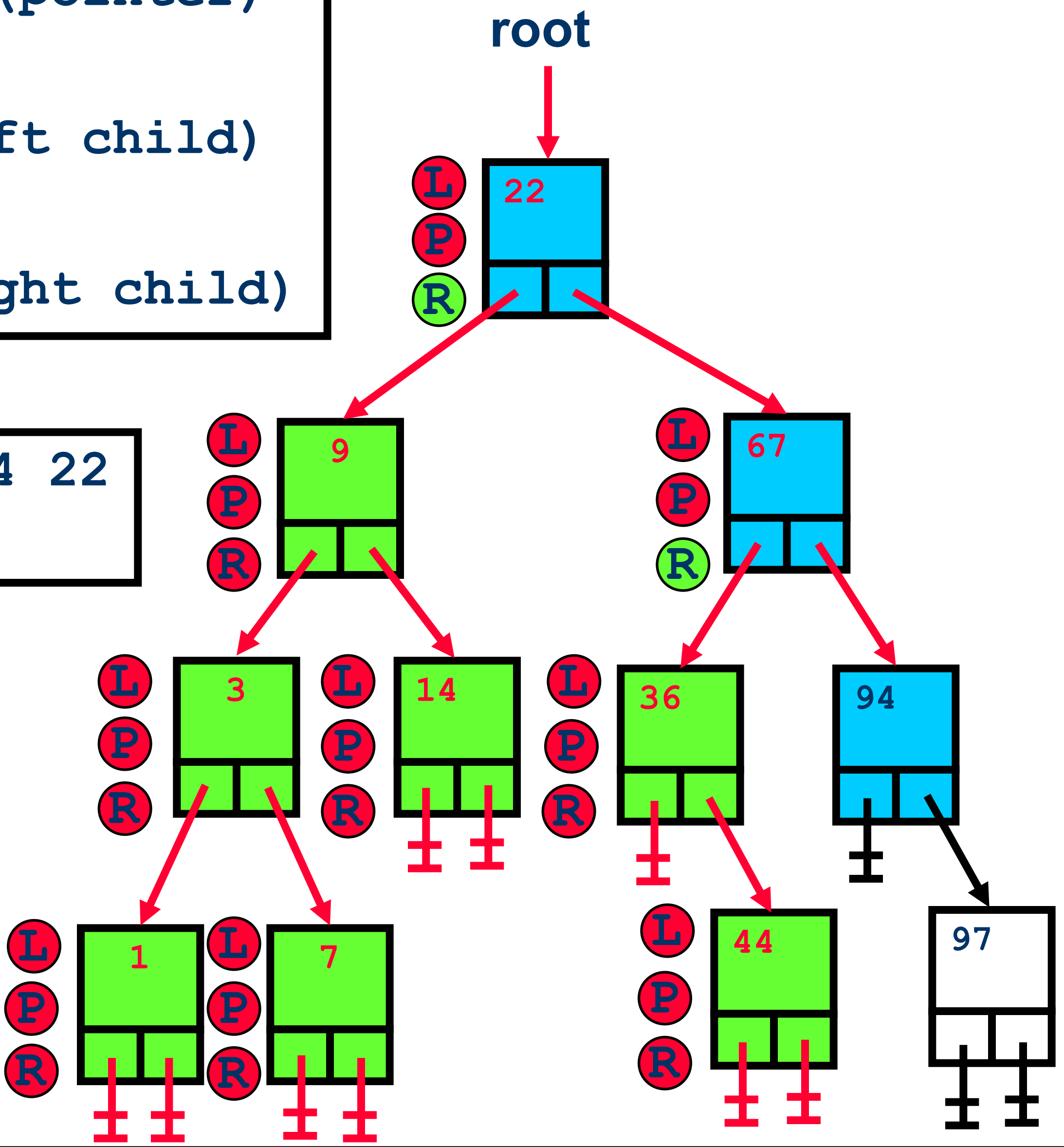
```
Output: 1 3 7 9 14 22
        36 44 67
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

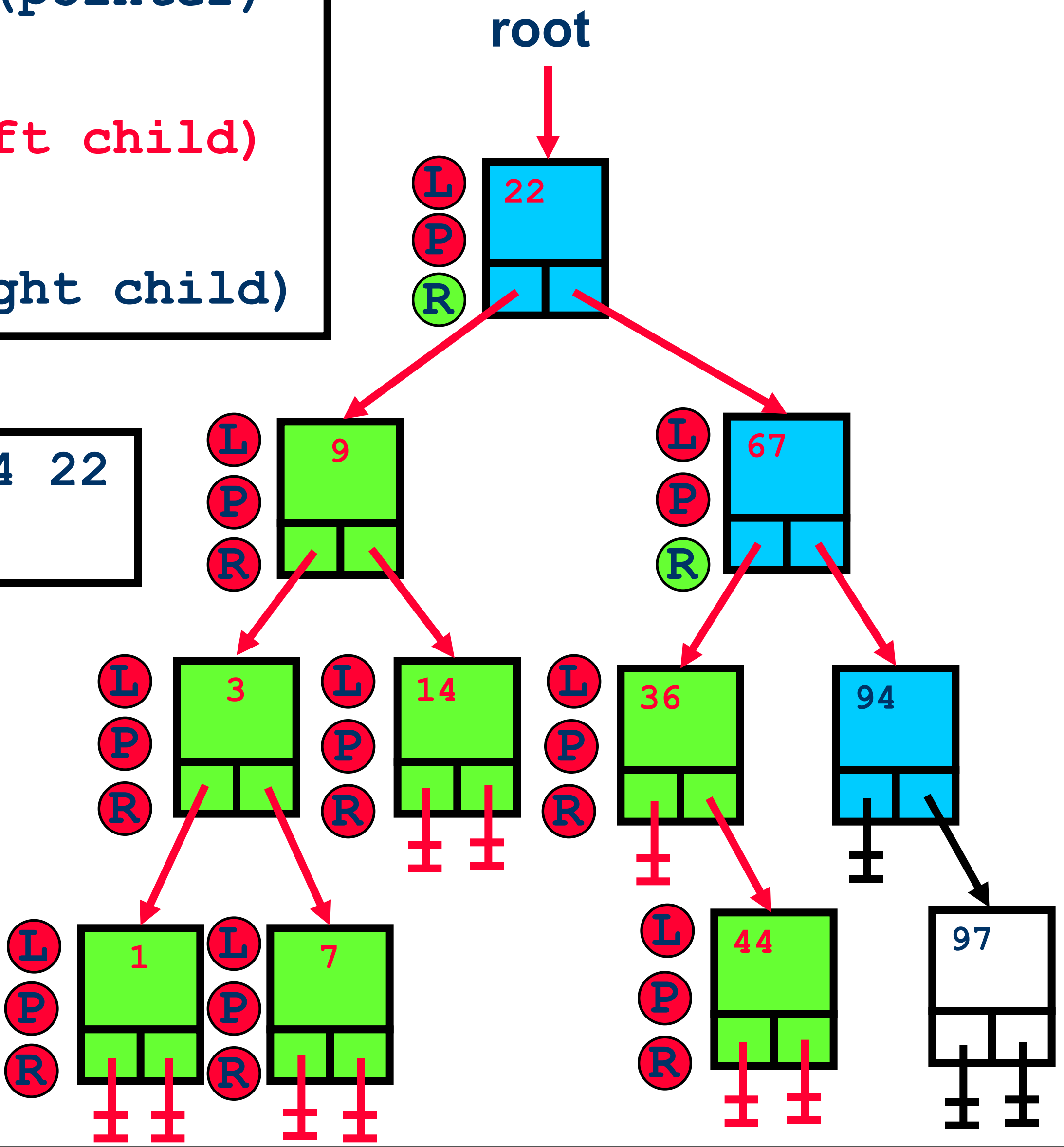
```
Output: 1 3 7 9 14 22
        36 44 67
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

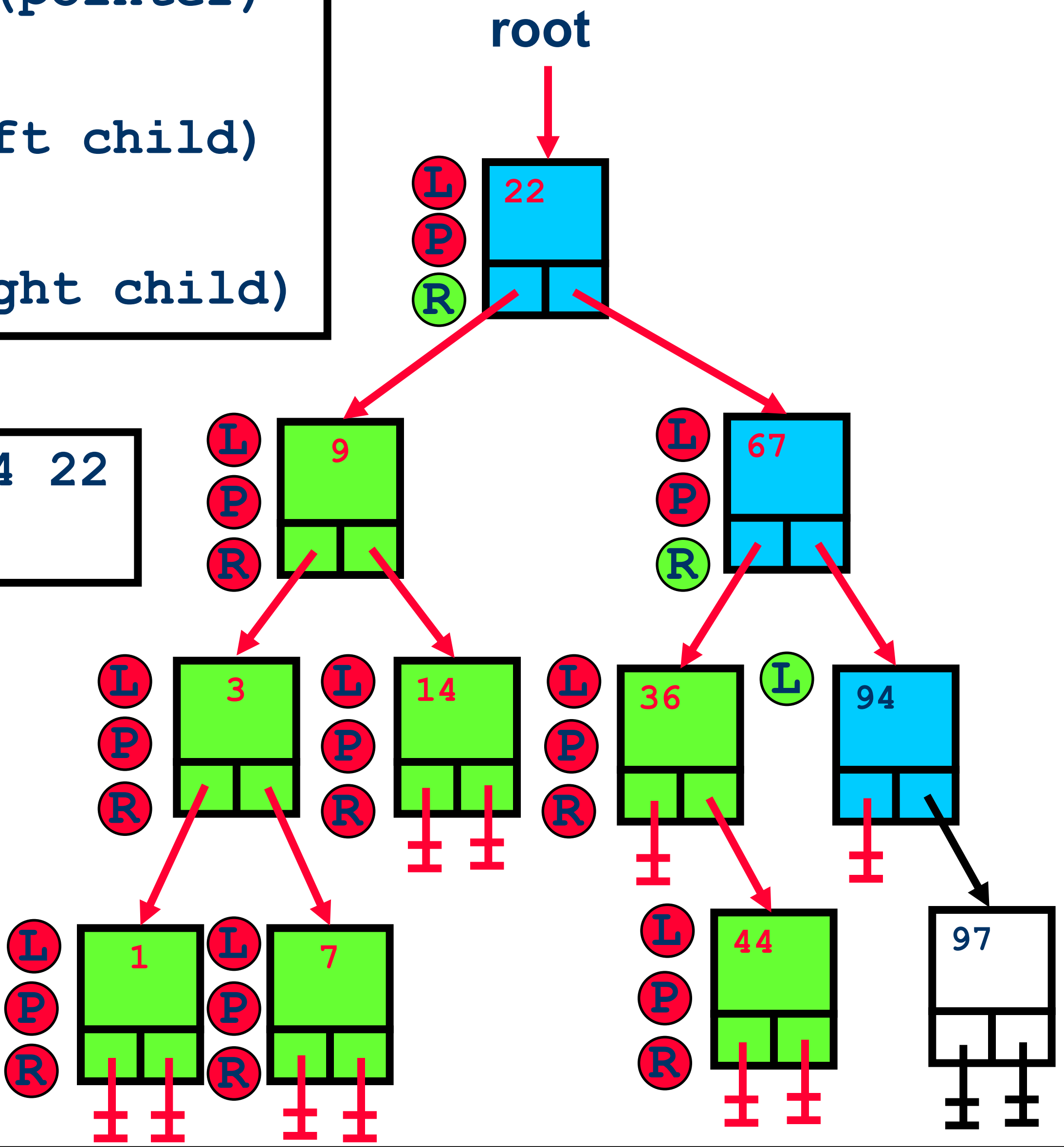
```
Output: 1 3 7 9 14 22
        36 44 67
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

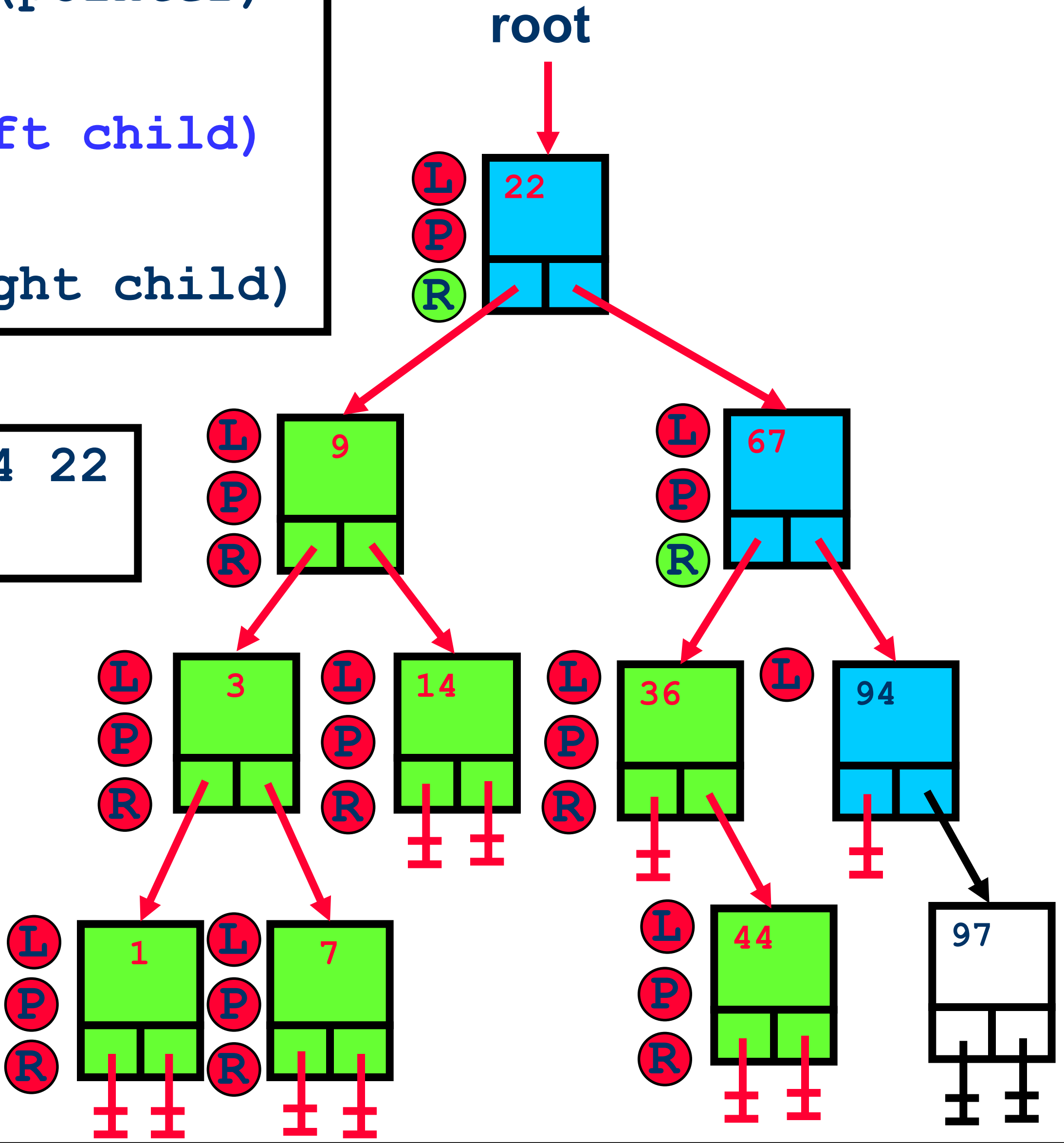
```
Output: 1 3 7 9 14 22
        36 44 67
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

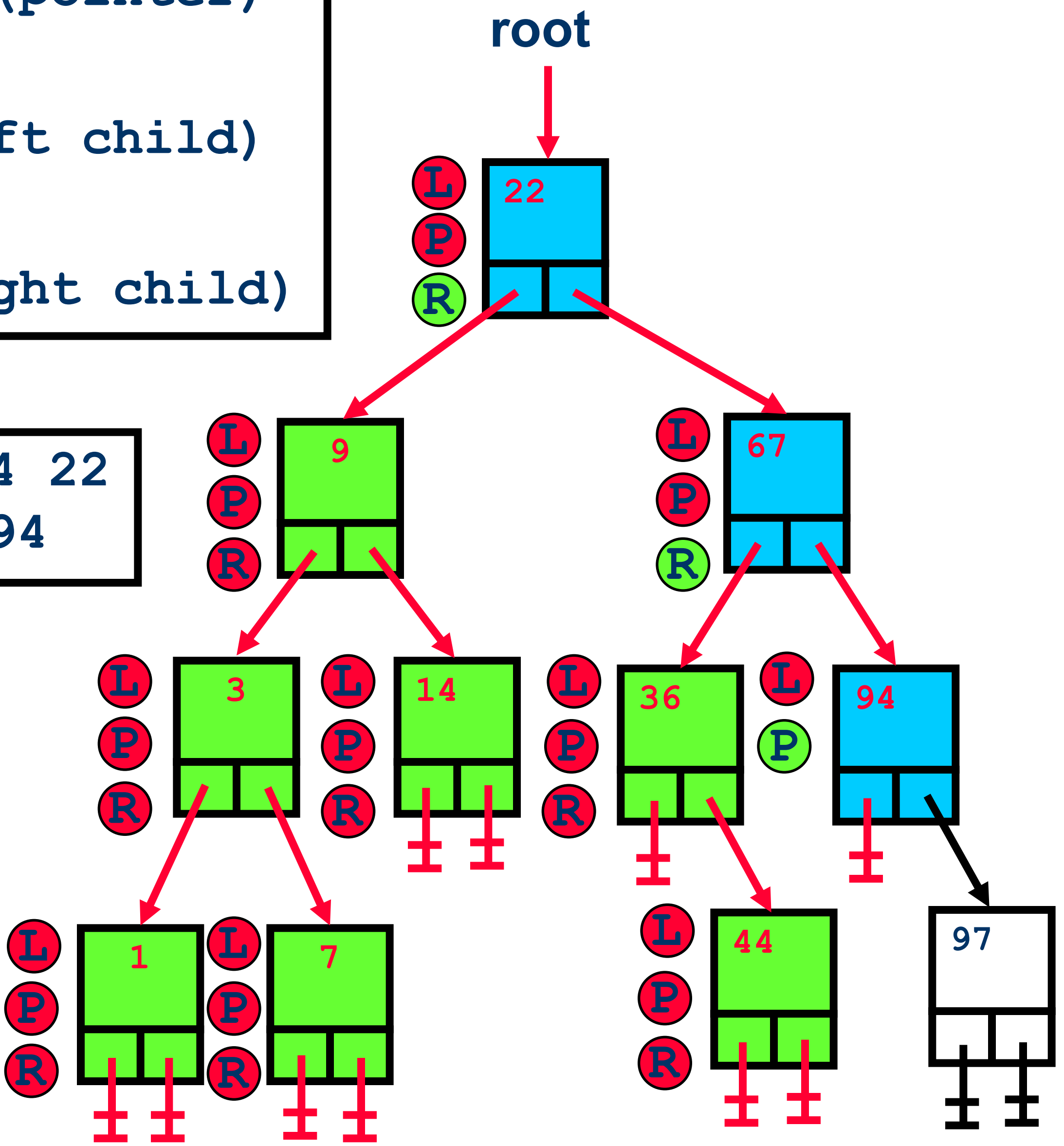
```
Output: 1 3 7 9 14 22
        36 44 67
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

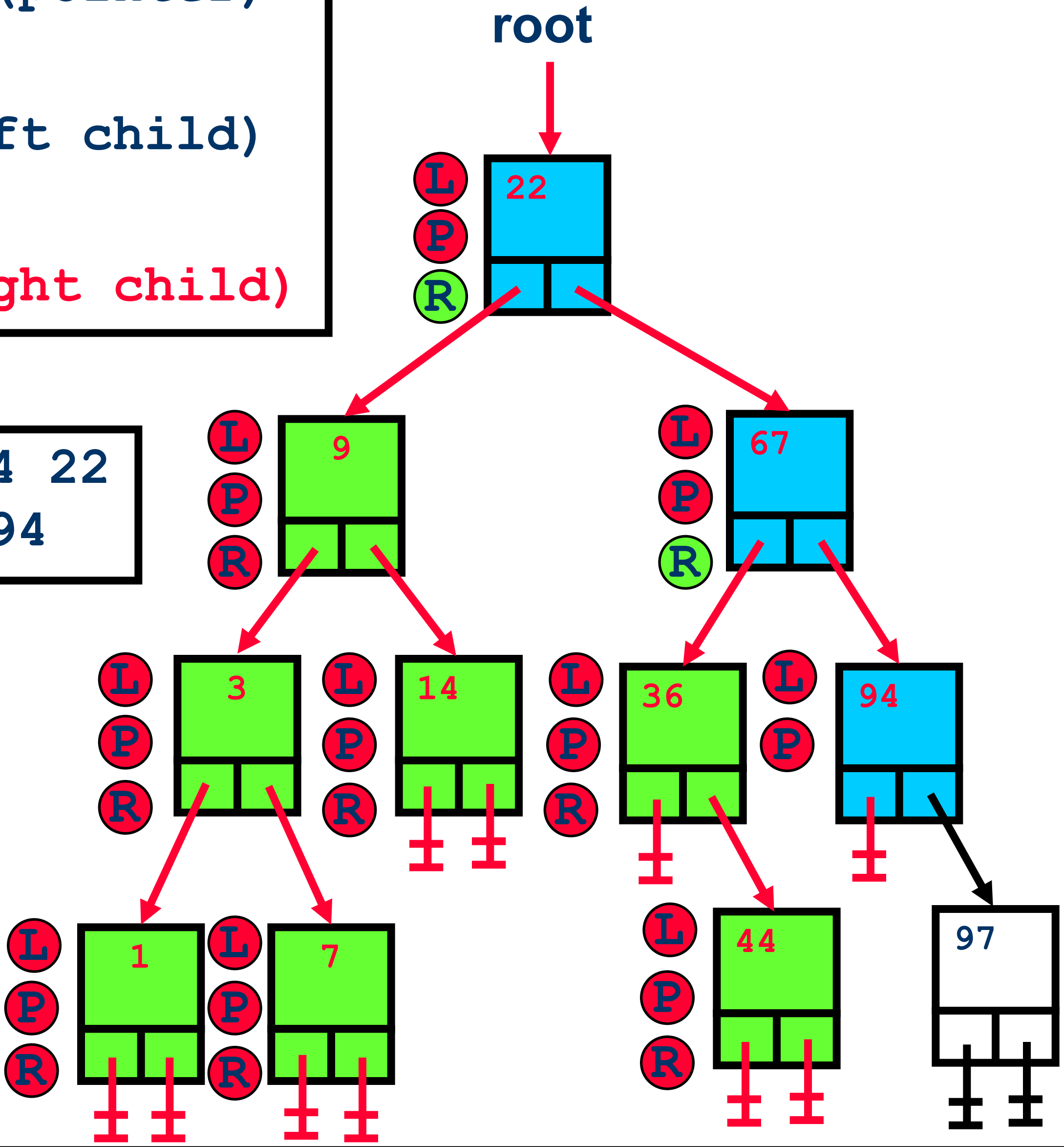
```
Output: 1 3 7 9 14 22
        36 44 67 94
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

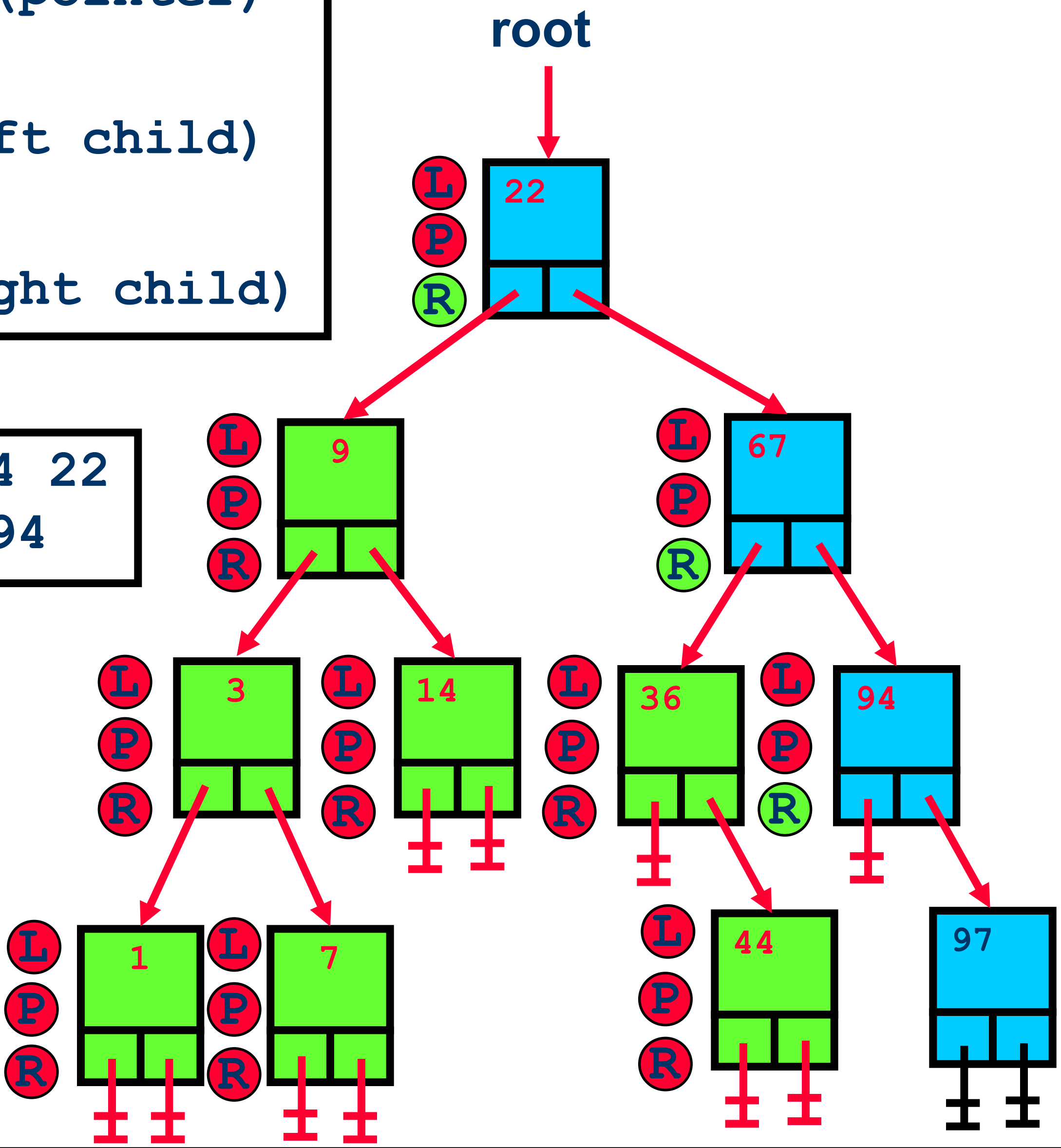
```
Output: 1 3 7 9 14 22
        36 44 67 94
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

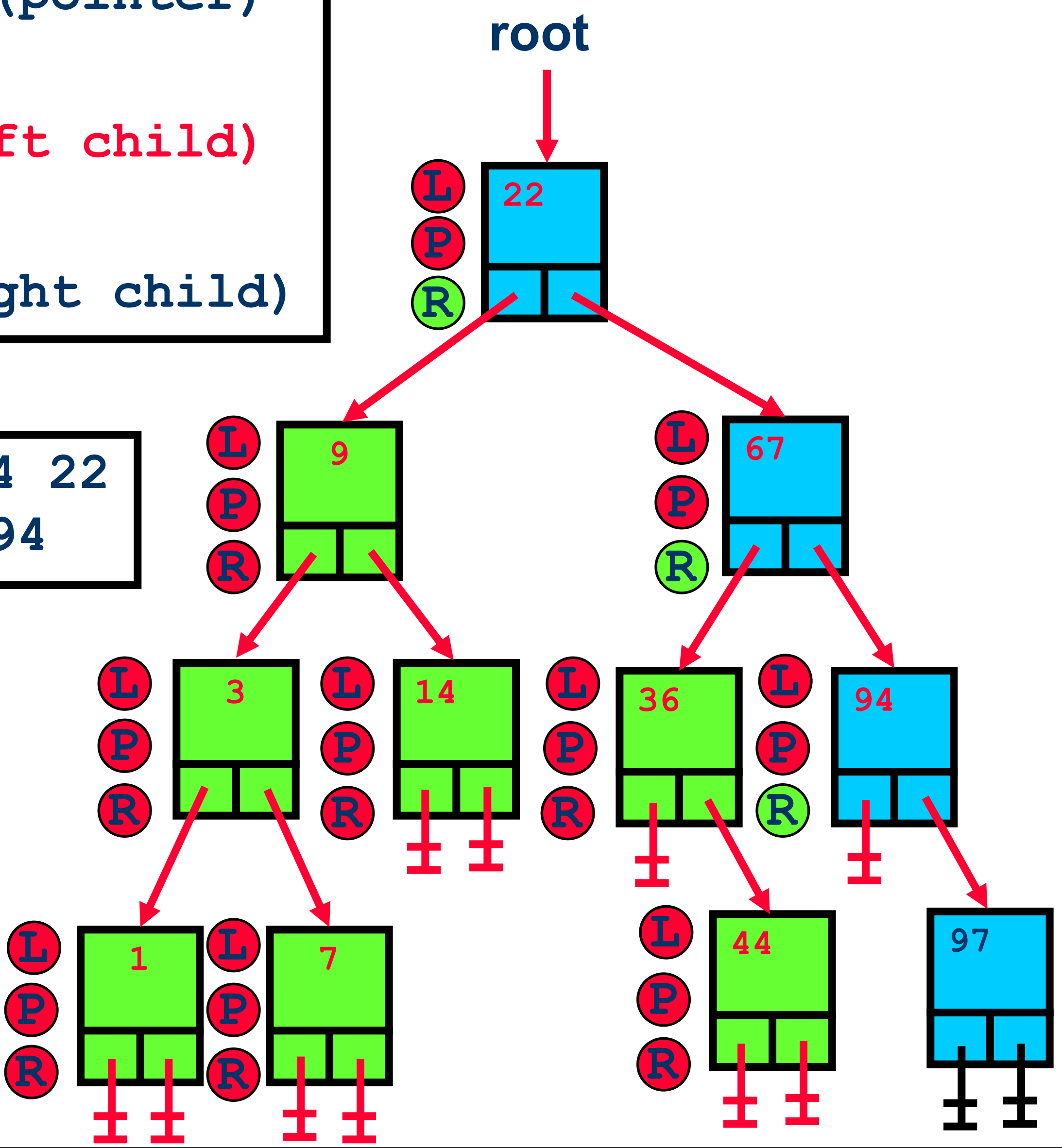
```
Output: 1 3 7 9 14 22
        36 44 67 94
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

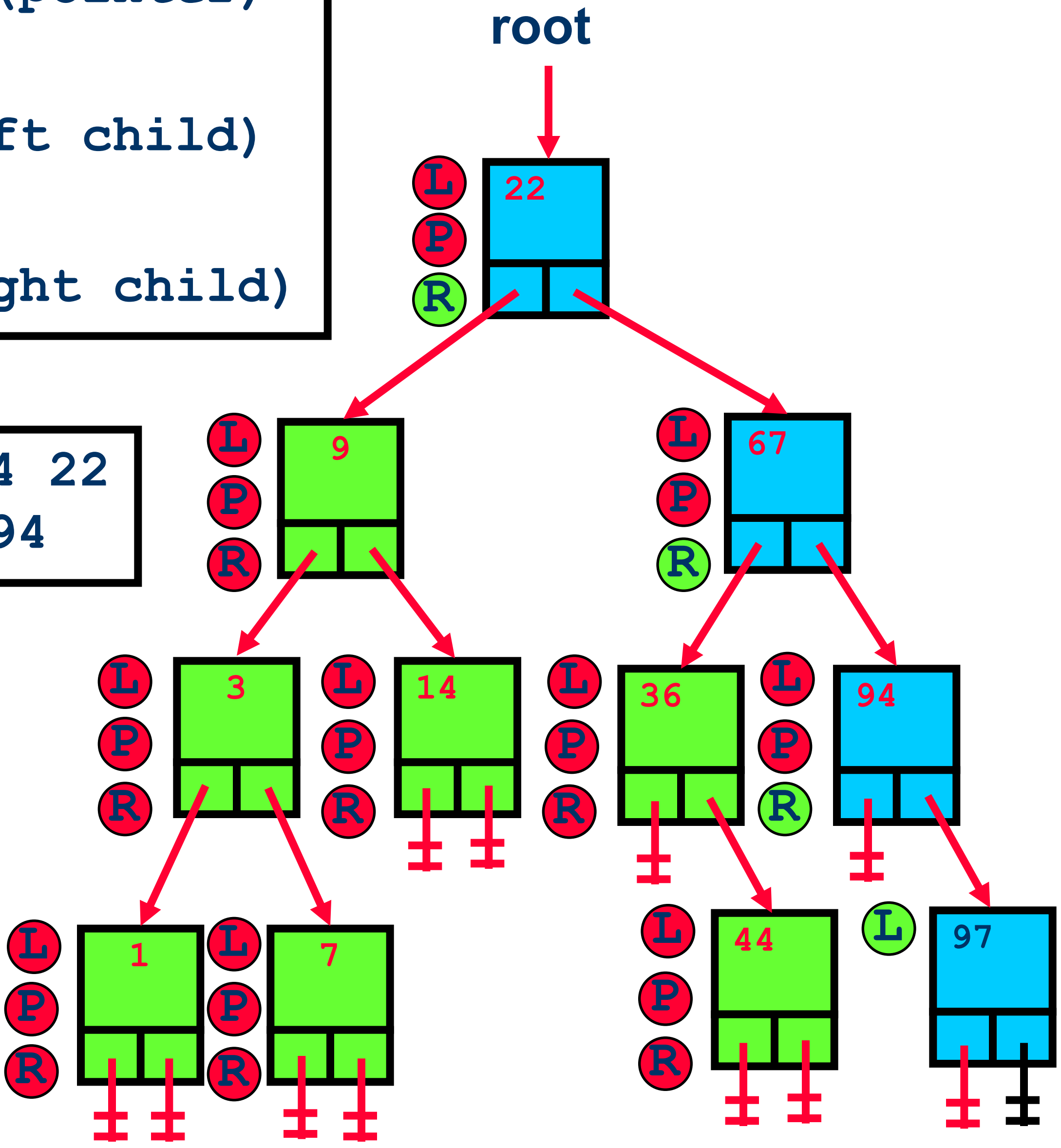
```
Output: 1 3 7 9 14 22
        36 44 67 94
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

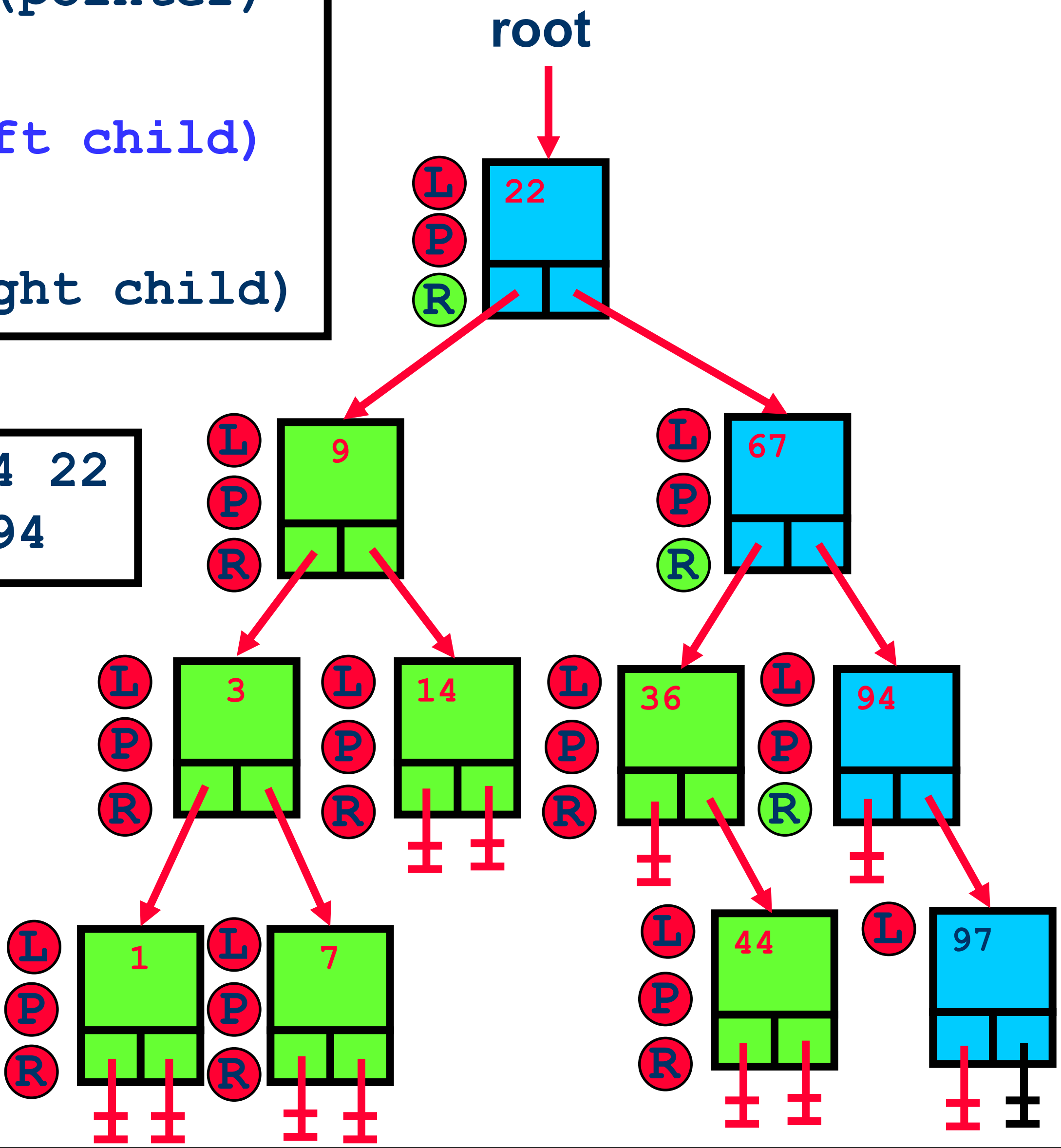
```
Output: 1 3 7 9 14 22
        36 44 67 94
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

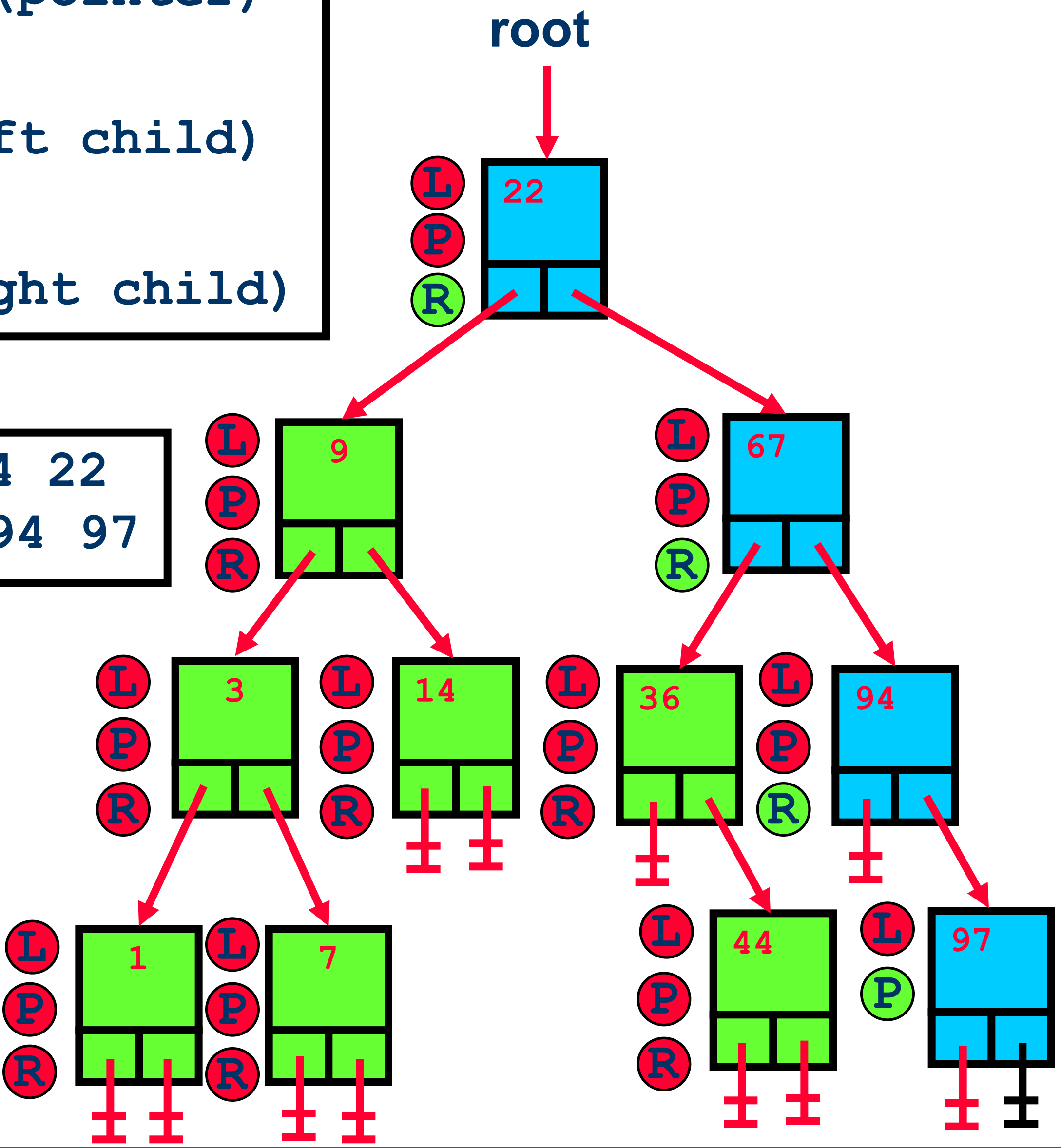
```
Output: 1 3 7 9 14 22
        36 44 67 94
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

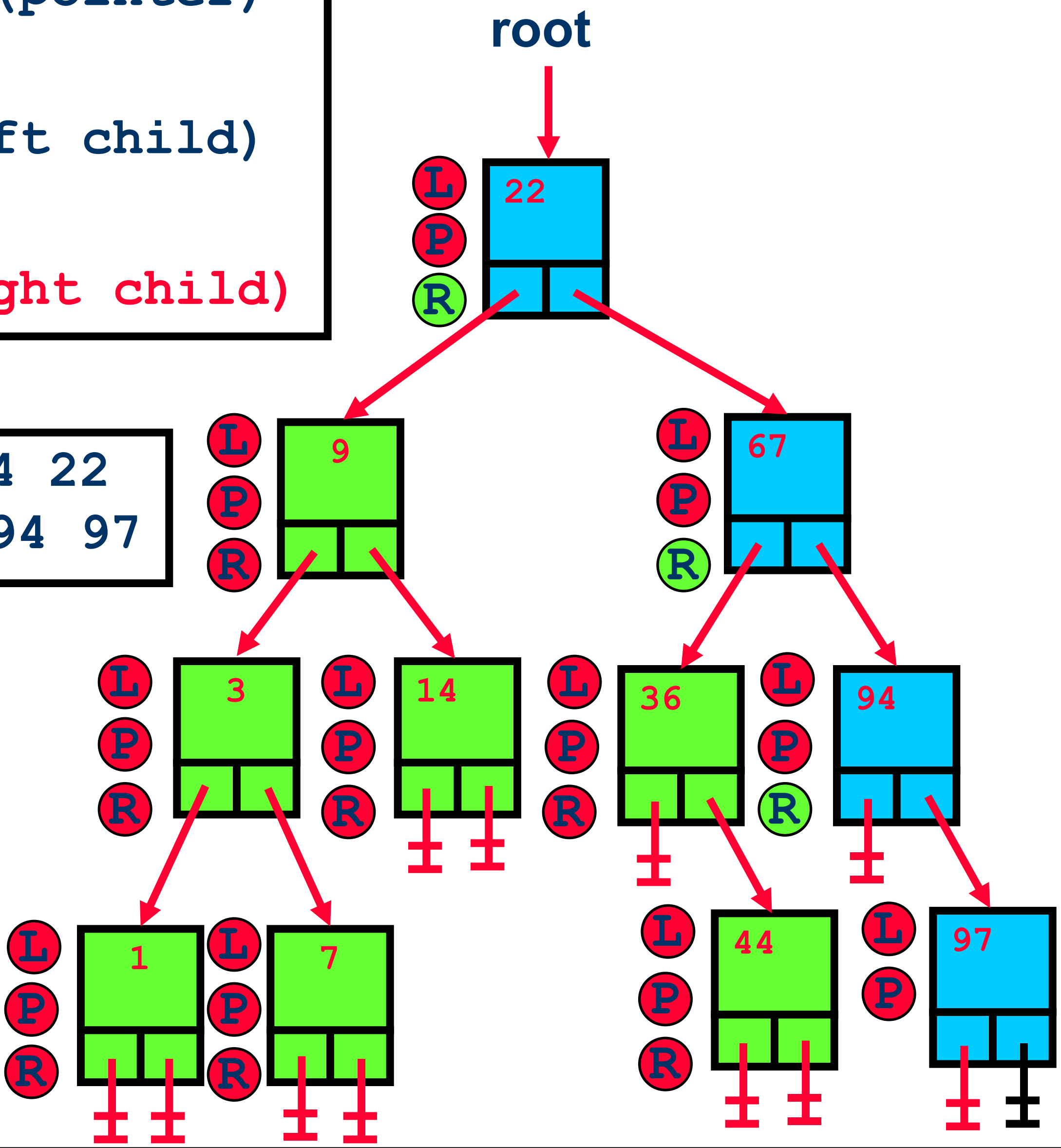
```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

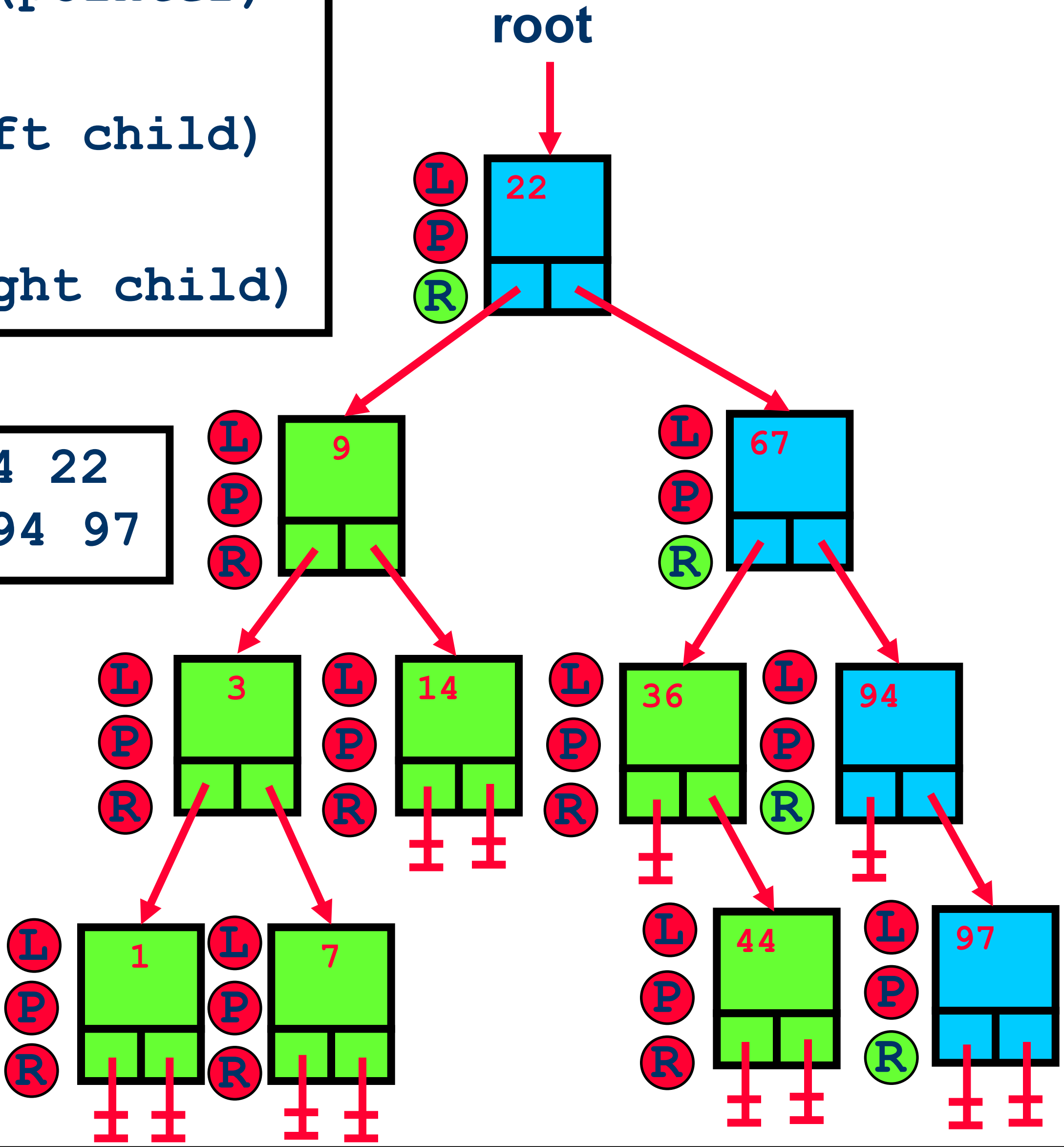
```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

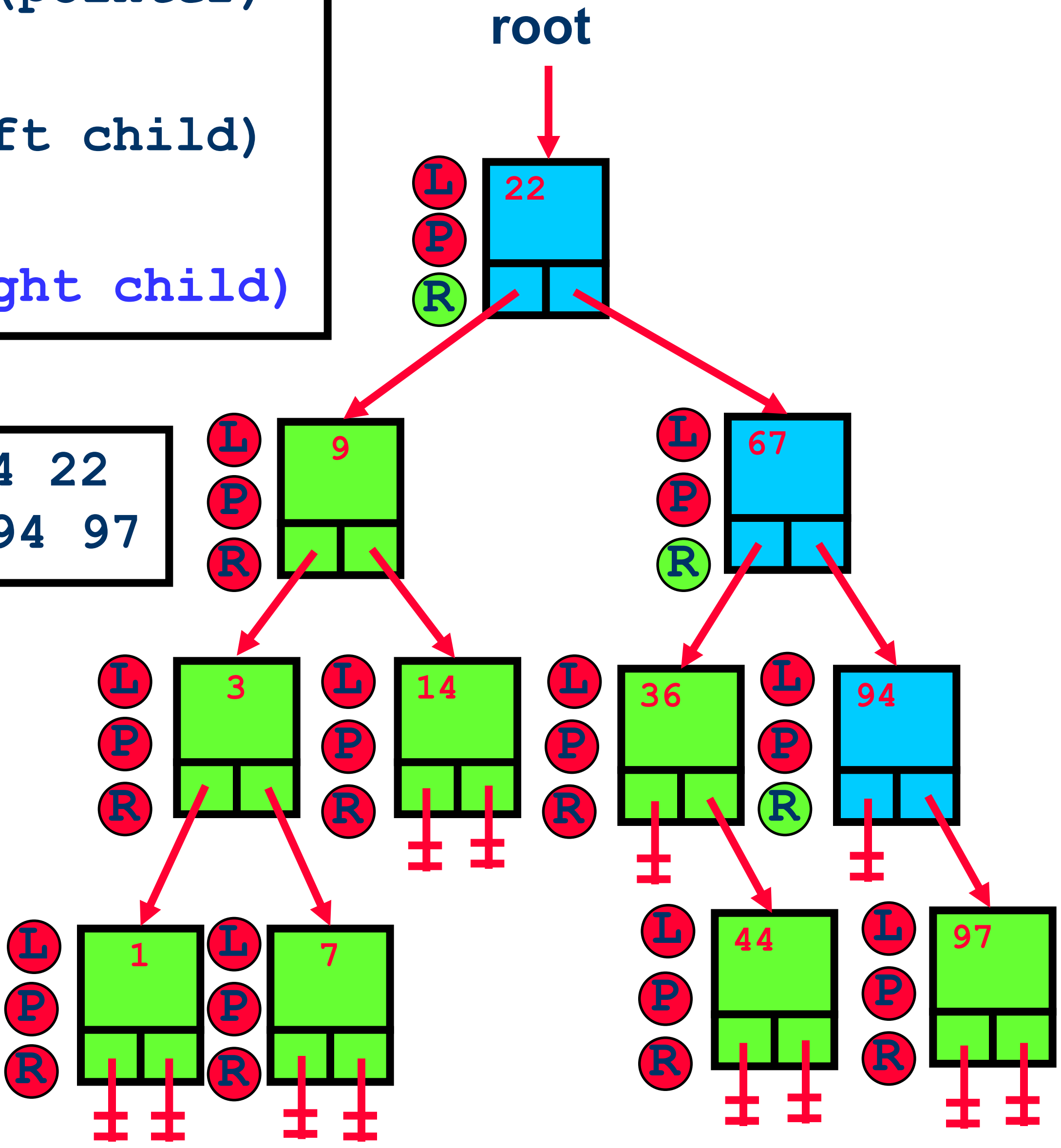
```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

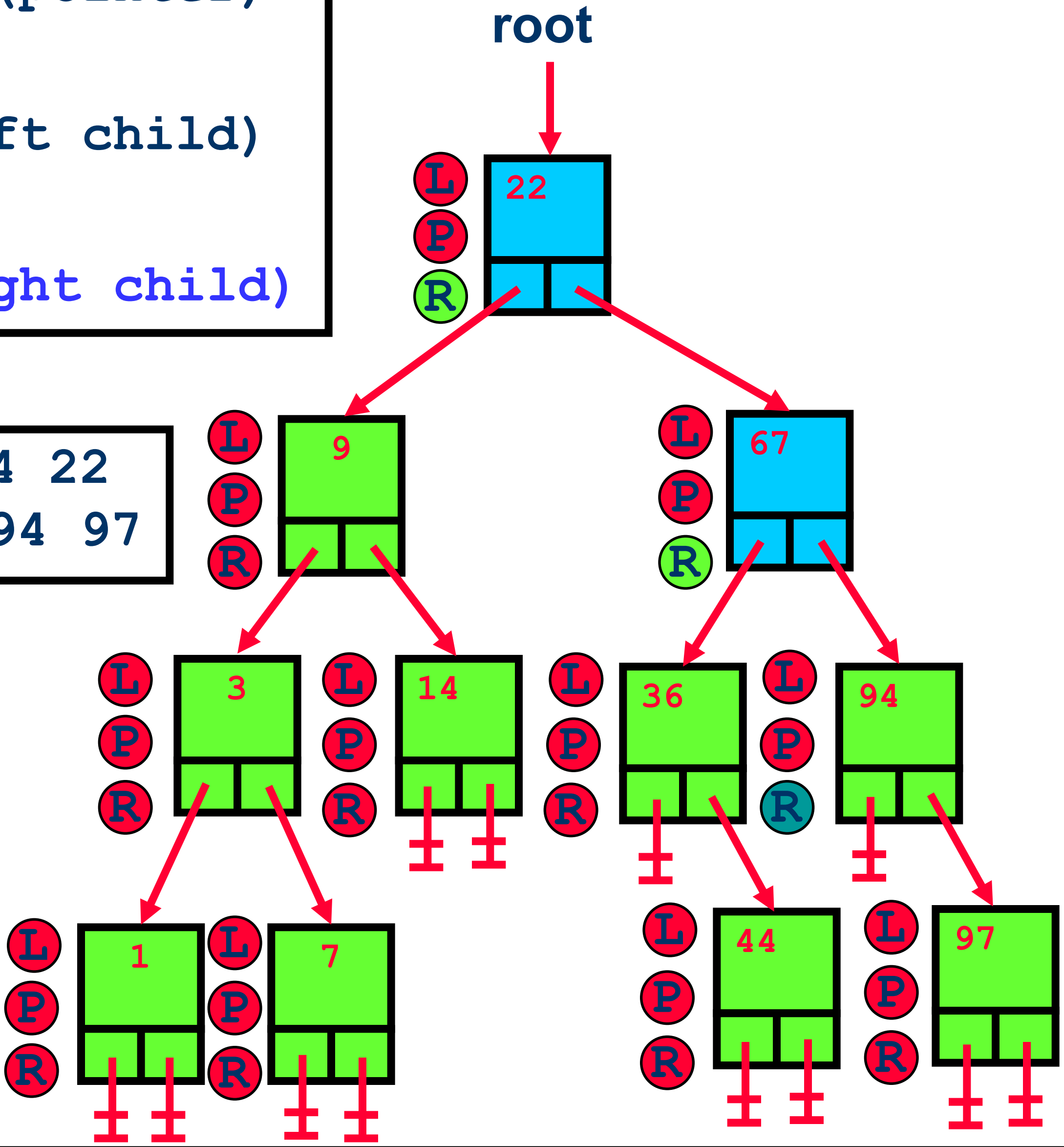
```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

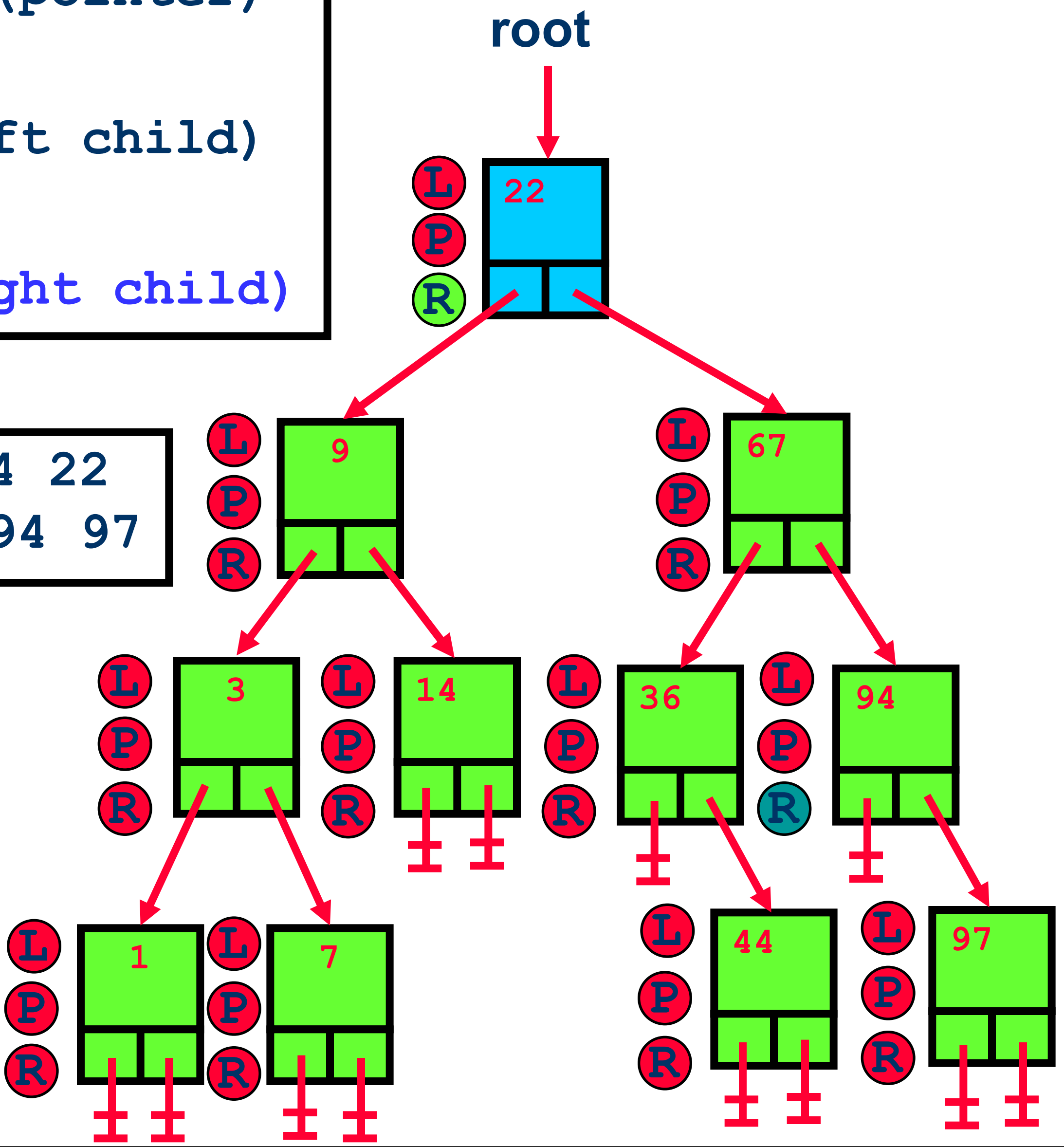
```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
Ⓛ InOrderPrint(left child)
Ⓟ print(data)
Ⓡ InOrderPrint(right child)
```

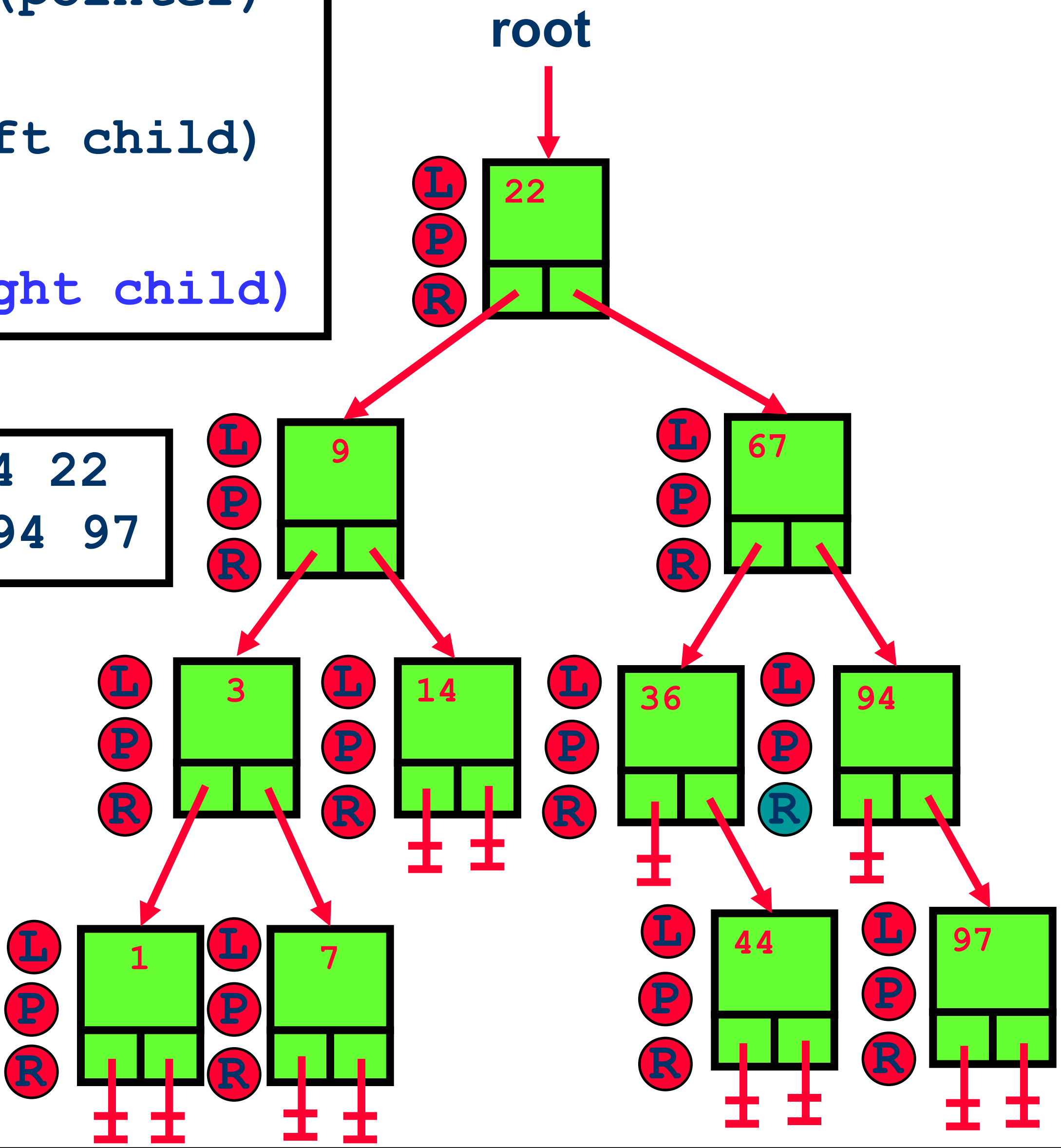
```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



Inorder Traversal

```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

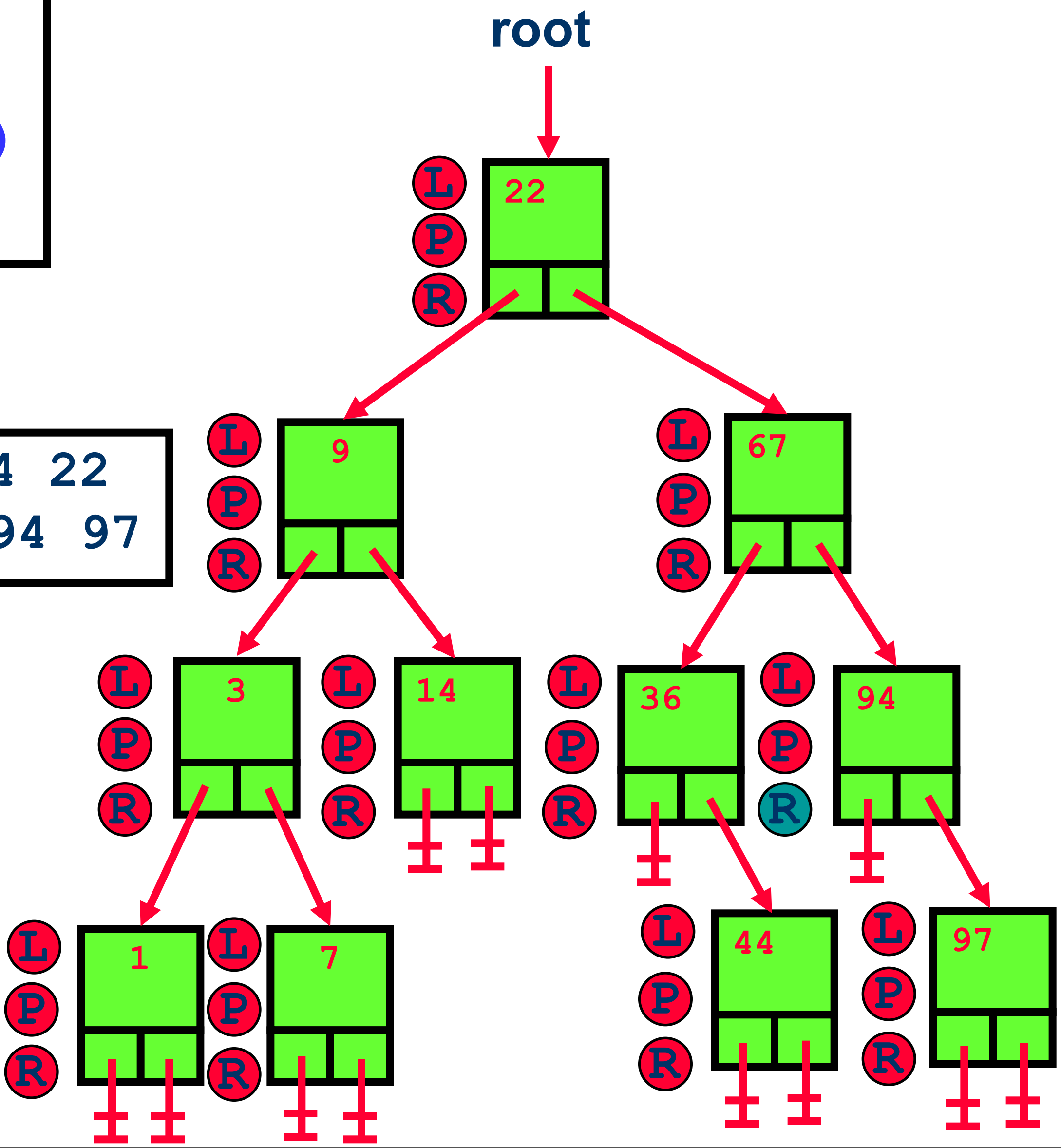
```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



Inorder Traversal

```
Algorithm Example
. . .
InOrderPrint(root)
. . .
```

```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```

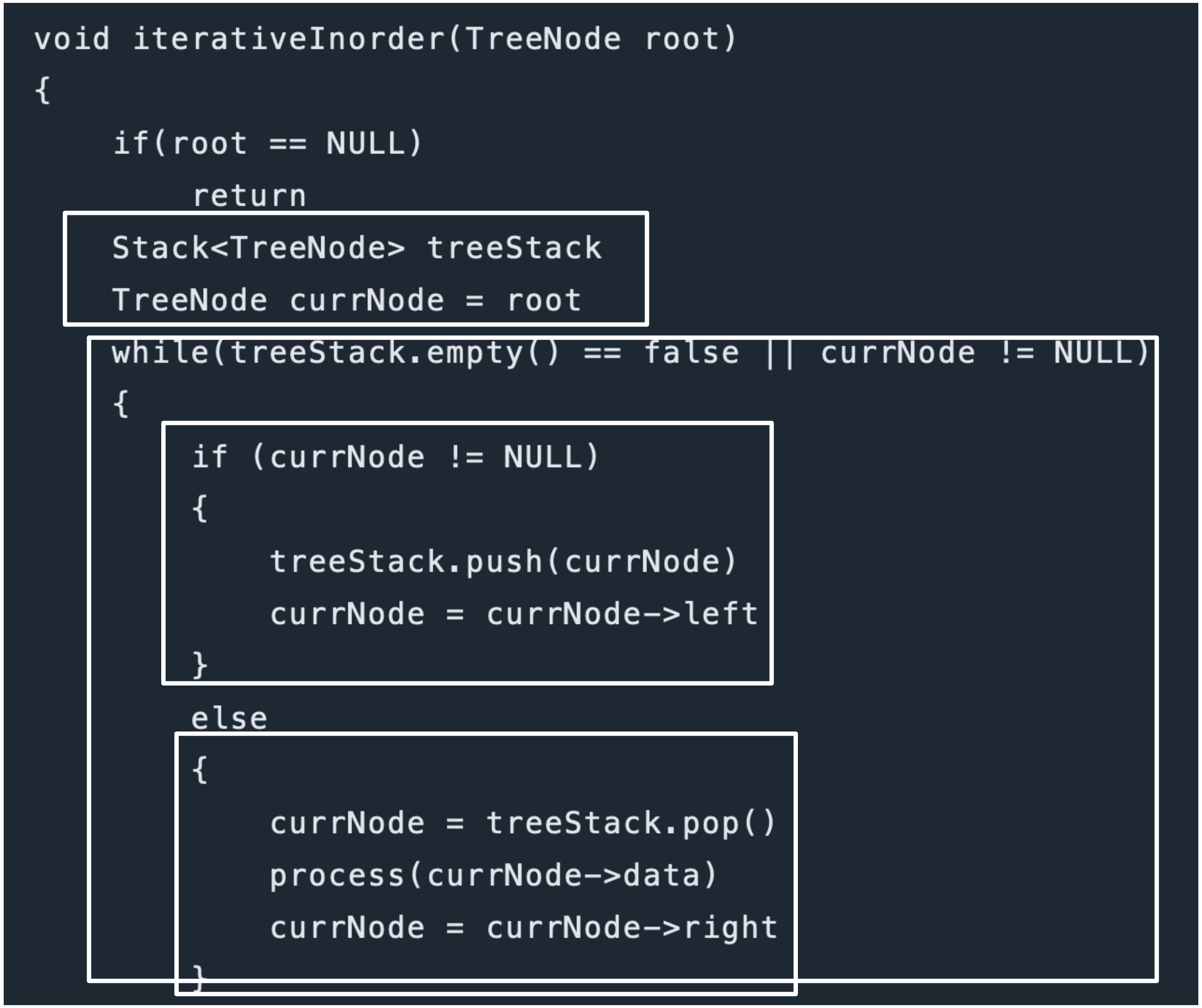


Non-recursive Inorder traversal

Pseudo Code:

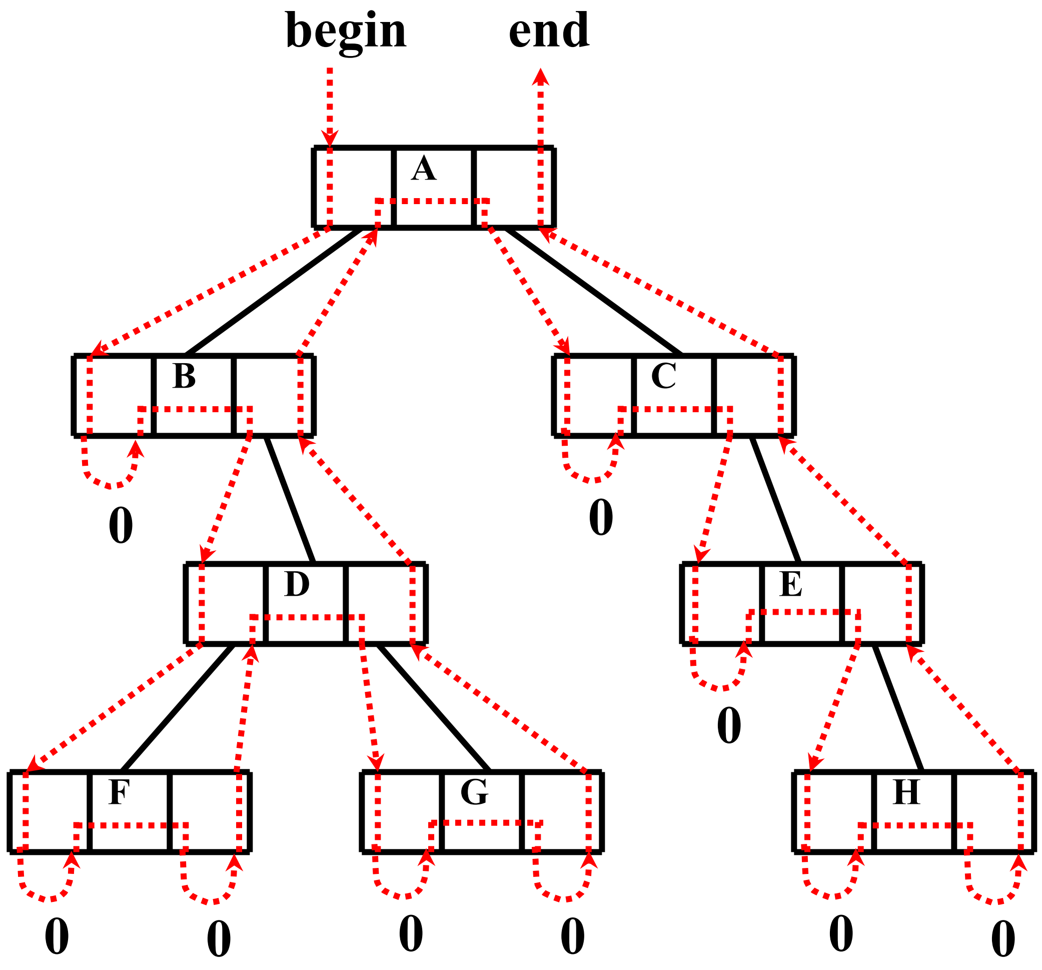
```
1. Create a Stack.
2. Push the root into the stack and set the root = root.left
continue till it hits the NULL.
3. If root is null and Stack is empty Then
        1. return, we are done
4. Else
        1. Pop the top Node from the Stack and set it as,
root = popped_Node.
        2. print the root and go right, root = root.right.
        3. Go to step 2.
5. End If
```

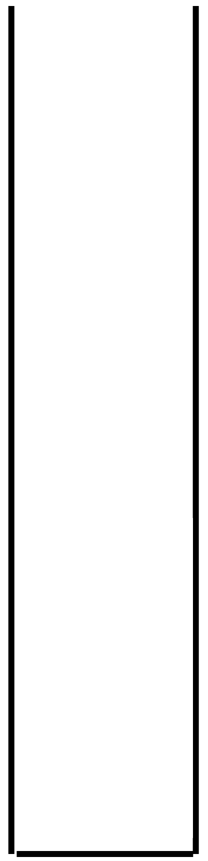
Non-recursive Inorder traversal



```
void iterativeInorder(TreeNode root)
{
    if(root == NULL)
        return
    Stack<TreeNode> treeStack
    TreeNode currNode = root
    while(treeStack.empty() == false || currNode != NULL)
    {
        if (currNode != NULL)
        {
            treeStack.push(currNode)
            currNode = currNode->left
        }
        else
        {
            currNode = treeStack.pop()
            process(currNode->data)
            currNode = currNode->right
        }
```

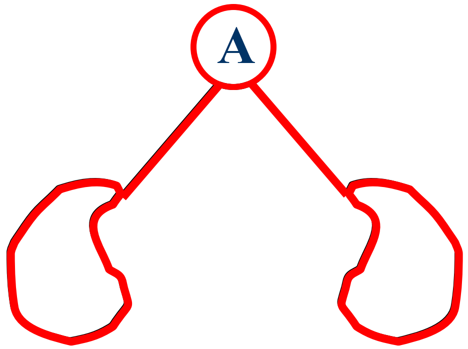
Non-recursive Inorder Traversal

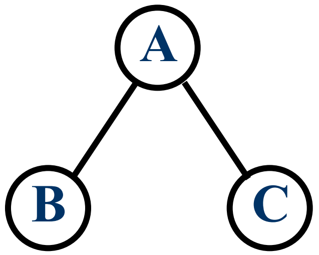




Inorder: B F D G A C E H

Postorder traversal



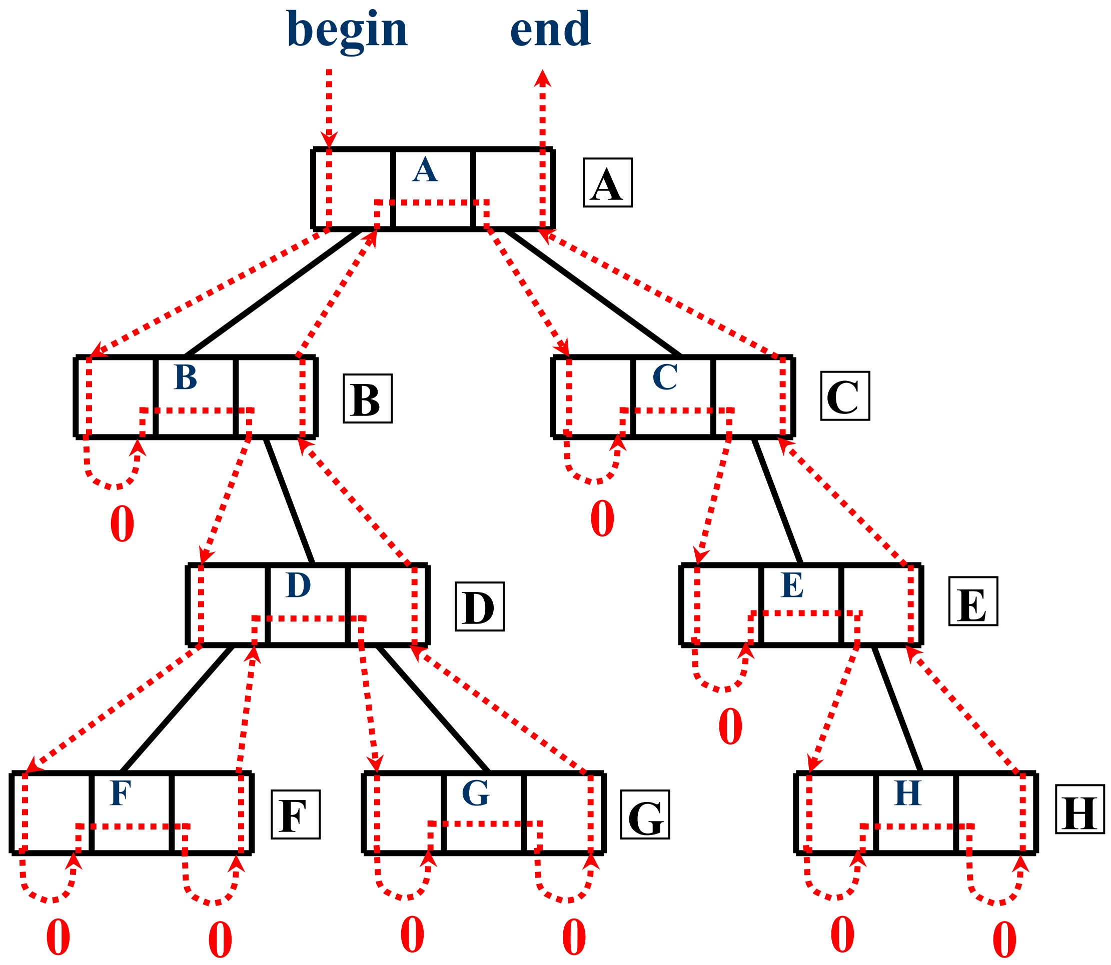


B C A

Recursive Postorder Traversal

```
template<class T>

void BinaryTree<T>:: PostOrder (BinaryTreeNode<T> *root)  {

        // postordor traverse the binary tree and its subtrees

          if (root != NULL) {
               PostOrder(root->leftchild());   // postorder traverse the left subtree

               PostOrder (root->rightchild());//postorder traverse the right subtree

               Visit(root->value());             // visit the current node
         }
}
```

Postorder Traversal



Postorder: F G D B H E C A

Non-recursive postorder traversal (1)

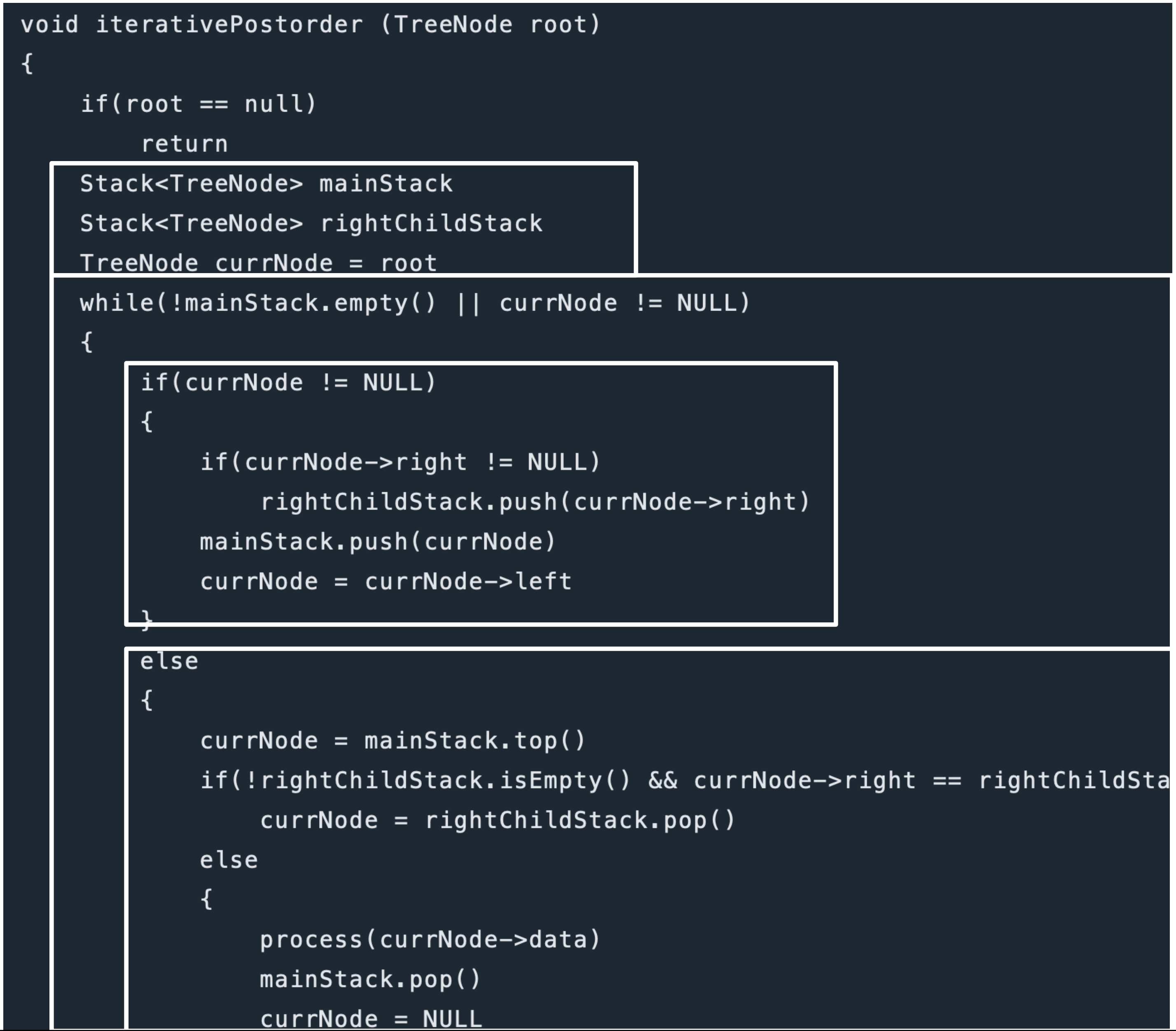
Before processing the left subtree of any node, we need to save two things on the stack:

1) Right child of the current node to process right subtree after the traversal of left subtree

2) Current node, so that we can process it after the traversal of right subtree.

Key: two separate stacks to store these two nodes

Non-recursive postorder traversal (2)



```
void iterativePostorder (TreeNode root)
{
    if(root == null)
        return
    Stack<TreeNode> mainStack
    Stack<TreeNode> rightChildStack
    TreeNode currNode = root
    while(!mainStack.empty() || currNode != NULL)
    {
        if(currNode != NULL)
        {
            if(currNode->right != NULL)
                rightChildStack.push(currNode->right)
            mainStack.push(currNode)
            currNode = currNode->left
        }
        else
        {
            currNode = mainStack.top()
            if(!rightChildStack.isEmpty() && currNode->right == rightChildSta
                currNode = rightChildStack.pop()
            else
            {
                process(currNode->data)
                mainStack.pop()
                currNode = NULL
```

Non-recursive postorder traversal (3)

1.1 Create an empty stack

2.1 Do following while root is not NULL

    a) Push root's right child and then root to stack.

    b) Set root as root's left child.

2.2 Pop an item from stack and set it as root.

    a) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.

    b) Else print root's data and set root as NULL.

2.3 Repeat steps 2.1 and 2.2 while stack is not empty.

BFT vs. DFT

- **Is there any difference in terms of Time Complexity?**
  - All four traversals require **O(n)** time as they visit every node exactly once.

- **Is there any difference in terms of Extra Space?**

  There is difference in terms of extra space required.
  - Extra Space required for **Level Order Traversal is O(w)** where w is maximum width of Binary Tree. In level order traversal, **queue** one by one stores nodes of different level.
  - Extra Space required for **Depth First Traversals is O(h)** where h is maximum height of Binary Tree. In Depth First Traversals, **stack** (or function call stack) stores all ancestors of a node.

Questions 1

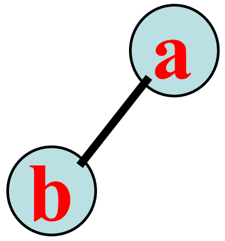
- Describe all binary trees that has the following properties
  - The sequences of pre-order and inorder are same
    - ➢Empty binary trees or binary trees without left subtrees
  - The sequences of pre-order and postorder are same
    - ➢Empty binary trees or binary trees without left and right subtrees
  - The sequences of inorder and postorder are same
    - ➢Empty binary trees or binary trees without right subtrees

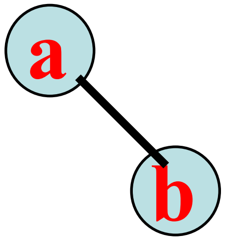
Binary Tree Construction

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

Some Examples

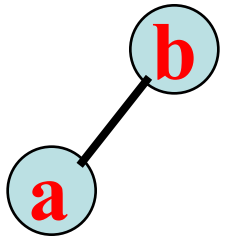
preorder =
ab

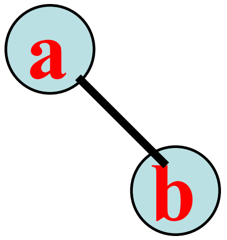




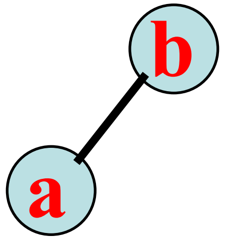
inorder

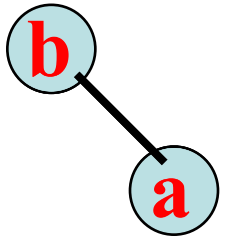
= ab



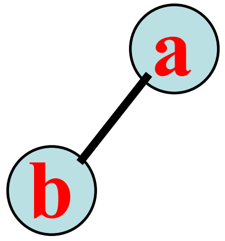


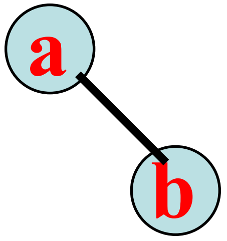
postorder
= ab





level order
= ab





Binary Tree Construction

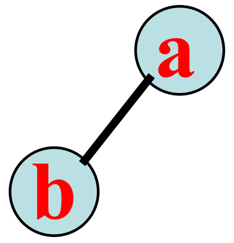
- Can you construct the binary tree, given two traversal sequences?

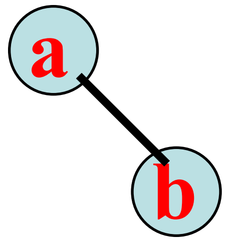
- Depends on which two sequences are given

Preorder And Postorder

preorder = ab

postorder = ba





Preorder and postorder do not uniquely define a binary tree.

Nor do preorder and level order (same example)

Nor do postorder and level order (same example)

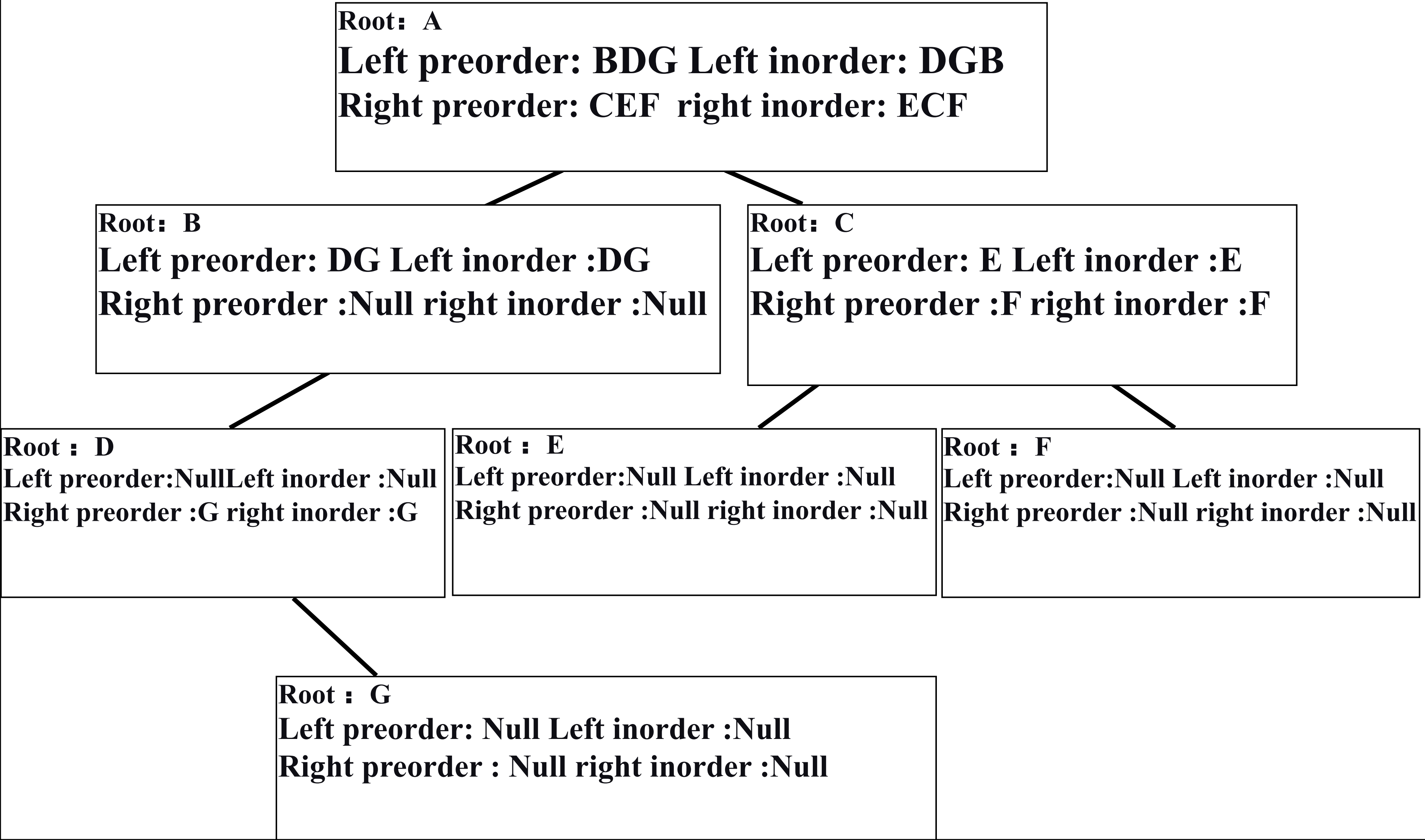
Preorder and Inorder

Preorder : $a_1a_2\ldots a_n$
Inorder : $b_1b_2\ldots b_n$
Root： a_1
In the order sequences, b_j is equal to a_1.

$$\{ b_1\ldots b_{j-1}\}b_j\{b_{j+1}\ldots b_n\} \longleftrightarrow a_1\{a_2\ldots a_k\}\{a_{k+1}\ldots a_n\}$$

Preorder and Inorder

Ex: Preorder sequence A B D G C E F, Inorder sequence D G B A E C F



Root: A
Left preorder: BDG Left inorder: DGB
Right preorder: CEF right inorder: ECF

Root: B
Left preorder: DG Left inorder :DG
Right preorder :Null right inorder :Null

Root: C
Left preorder: E Left inorder :E
Right preorder :F right inorder :F

Root : D
Left preorder:NullLeft inorder :Null
Right preorder :G right inorder :G

Root : E
Left preorder:Null Left inorder :Null
Right preorder :Null right inorder :Null

Root : F
Left preorder:Null Left inorder :Null
Right preorder :Null right inorder :Null

Root : G
Left preorder: Null Left inorder :Null
Right preorder : Null right inorder :Null

Postorder and Inorder

Postorder: $a_1a_2\ldots a_n$

Inorder: $b_1b_2\ldots b_n$

Root： a_n

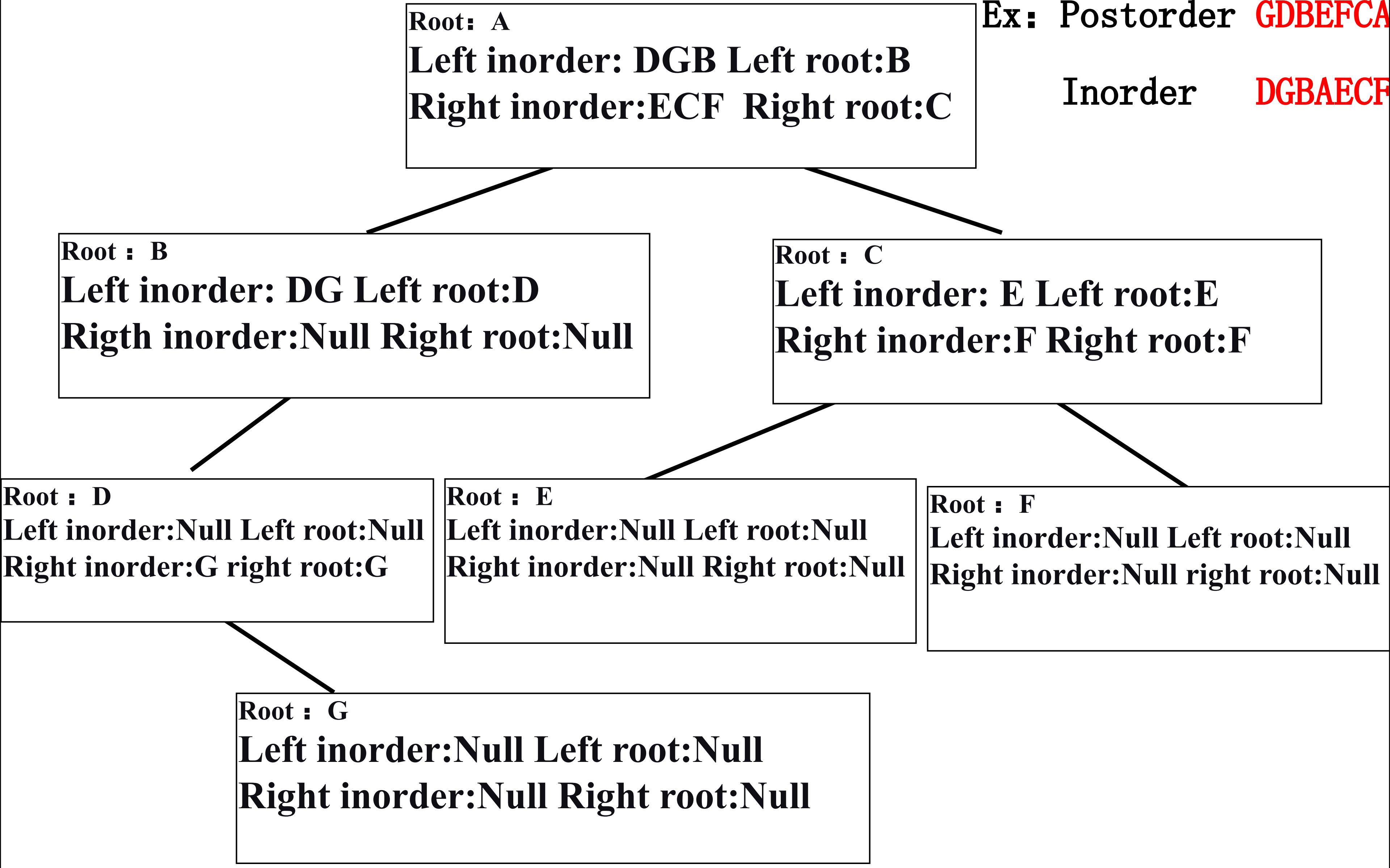
In the Inorder squence, b_j is equal to $a_{n.}$

$\{ b_1\ldots b_{j-1}\}b_j\{b_{j+1}\ldots b_n\} \longleftrightarrow \{a_1a_2\ldots a_k\}\{a_{k+1}\ldots a_{n-1}\}a_n$

Postorder and Inorder

Ex： Postorder GDBEFCA

Inorder DGBAECF



Root： A
Left inorder: DGB Left root:B
Right inorder:ECF Right root:C

Root： B
Left inorder: DG Left root:D
Rigth inorder:Null Right root:Null

Root： C
Left inorder: E Left root:E
Right inorder:F Right root:F

Root： D
Left inorder:Null Left root:Null
Right inorder:G right root:G

Root： E
Left inorder:Null Left root:Null
Right inorder:Null Right root:Null

Root： F
Left inorder:Null Left root:Null
Right inorder:Null right root:Null

Root： G
Left inorder:Null Left root:Null
Right inorder:Null Right root:Null

Homework

- Please refer to Icourse， Huawei Cloud.

- Due date for quiz: 23:30 2022/4/19
- Due date for homework: 23:30 2022/4/24
- Due data for online lab assignment: 2022/4/24 23：30
- Due data for offline lab assignment: 2022/4/24 18:00