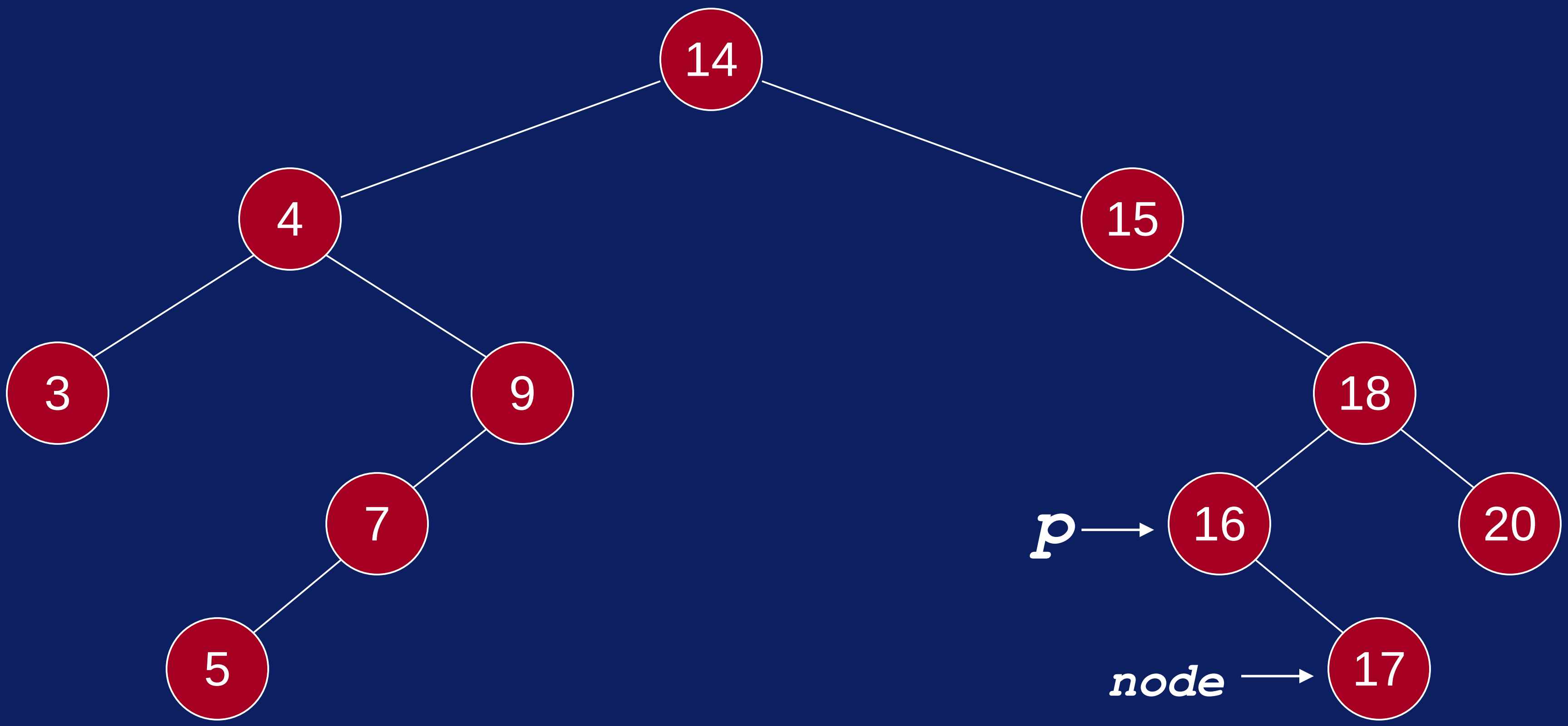
Lecture No.13

Data Structures

Trace of insert



17, 9, 14, 5

```
p->setRight( node );
```

Cost of Search

- Given that a binary tree is level *d* deep. How long does it take to find out whether a number is already present?
- Consider the insert(17) in the example tree.
- Each time around the while loop, we did one comparison.
- After the comparison, we moved a level *down*.

Cost of Search

- With the binary tree in place, we can write a routine *find(x)* that returns true if the number *x* is present in the tree, false otherwise.
- How many comparison are needed to find out if *x* is present in the tree?
- We do one comparison at each level of the tree until either *x* is found or q becomes NULL.

Cost of Search

- If the binary tree is built out of *n* numbers, how many comparisons are needed to find out if a number *x* is in the tree?
- Recall that the depth of the complete binary tree built using ‘*n*’ nodes will be $log_2(n+1) - 1$.
- For example, for n=100,000, $\log_2(100001)$ is less than 20; the tree would be 20 levels deep.

Cost of Search

- If the tree is complete binary or nearly complete, searching through 100,000 numbers will require a maximum of 20 comparisons.
- Or in general, approximately $log_2(n)$.
- Compare this with a linked list of 100,000 numbers. The comparisons required could be a maximum of n.

Binary Search Tree

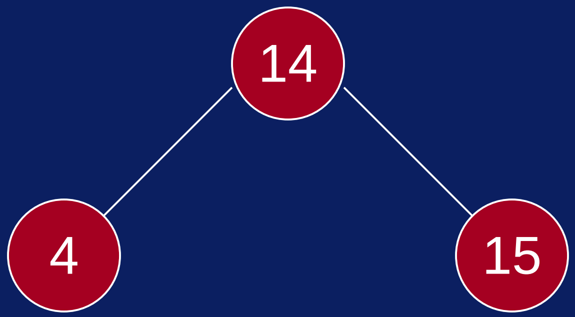
- A binary tree with the property that items in the left subtree are smaller than the root and items are larger or equal in the right subtree is called a *binary search tree* (BST).
- The tree we built for searching for duplicate numbers was a binary search tree.
- BST and its variations play an important role in searching algorithms.

Traversing a Binary Tree

- Suppose we have a binary tree, ordered (BST) or unordered.
- We want to print all the values stored in the nodes of the tree.
- In what order should we print them?

Traversing a Binary Tree

- Ways to print a 3 node tree:



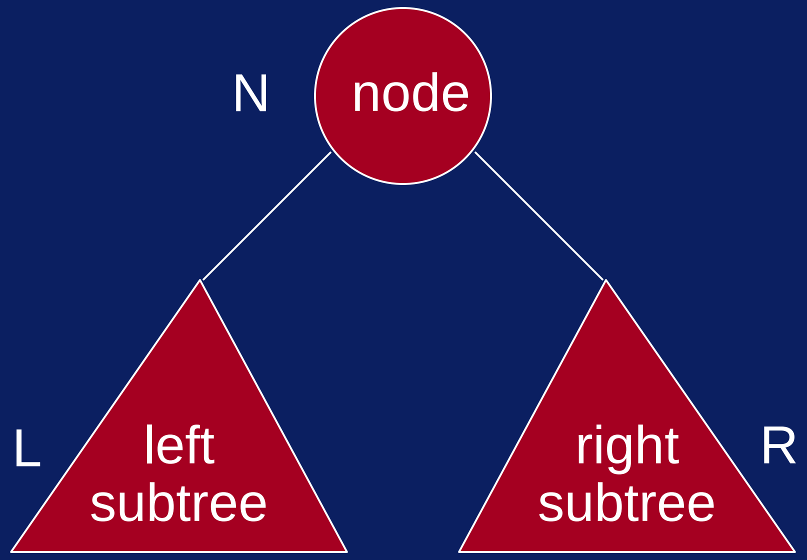
(4, 14, 15), (4,15,14)

(14,4,15), (14,15,4)

(15,4,14), (15,14,4)

Traversing a Binary Tree

- In case of the general binary tree:



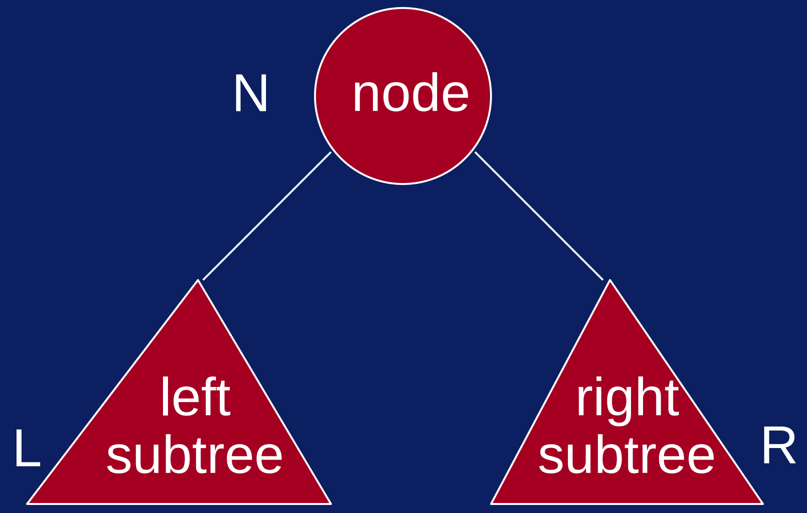
(L,N,R), (L,R,N)

(N,L,R), (N,R,L)

(R,L,N), (R,N,L)

Traversing a Binary Tree

- Three common ways



Preorder:	(N,L,R)
Inorder:	(L,N,R)
Postorder:	(L,R,N)

Traversing a Binary Tree

```
void preorder(TreeNode<int>* treeNode)
{
    if( treeNode != NULL )
    {
        cout << *(treeNode->getInfo())<<" ";
        preorder(treeNode->getLeft());
        preorder(treeNode->getRight());
    }
}
```

Traversing a Binary Tree

```
void inorder(TreeNode<int>* treeNode)
{
    if( treeNode != NULL )
    {
        inorder(treeNode->getLeft());
        cout << *(treeNode->getInfo())<<" ";
        inorder(treeNode->getRight());
    }
}
```

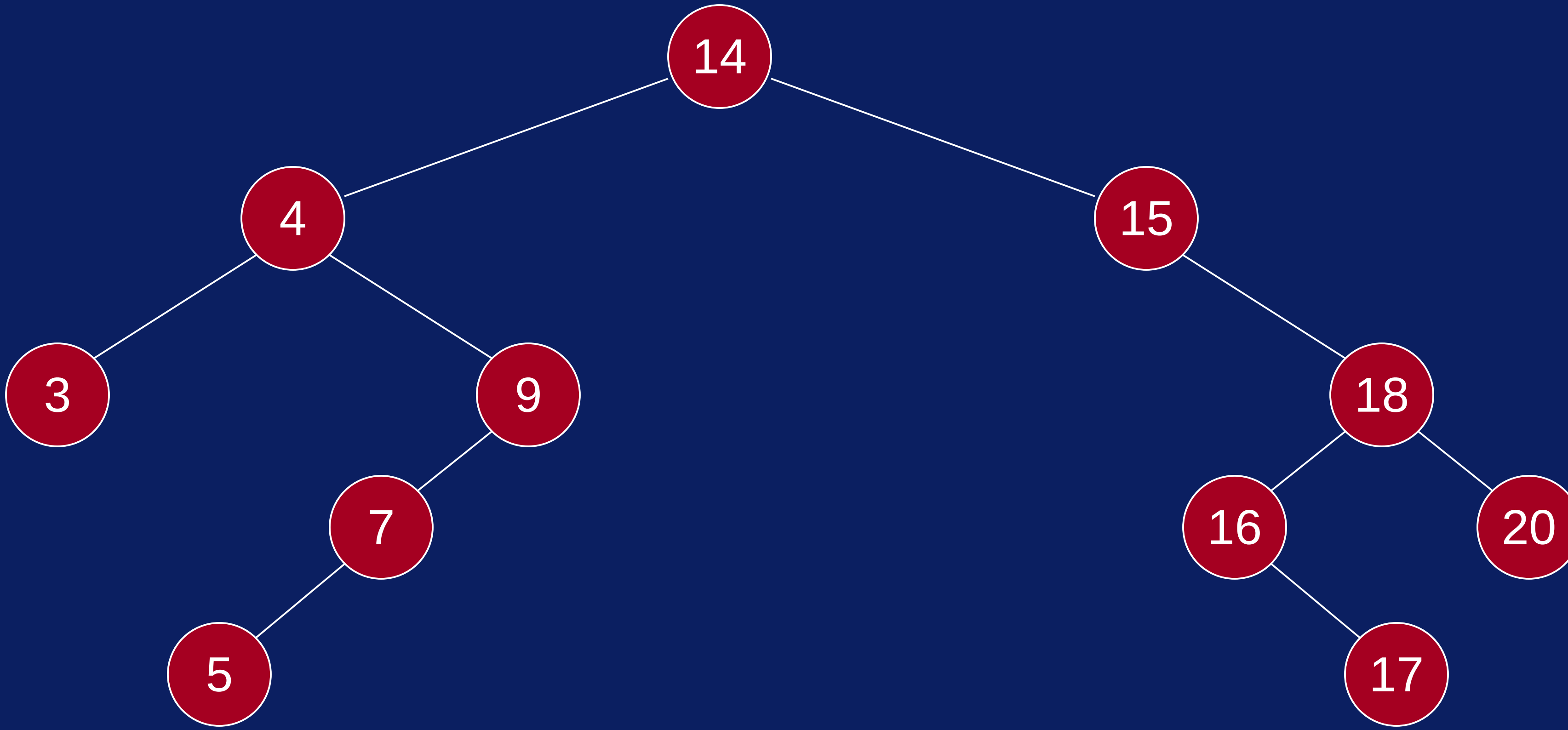
Traversing a Binary Tree

```
void postorder(TreeNode<int>* treeNode)
{
    if( treeNode != NULL )
    {
        postorder(treeNode->getLeft());
        postorder(treeNode->getRight());
        cout << *(treeNode->getInfo())<<" ";
    }
}
```

Traversing a Binary Tree

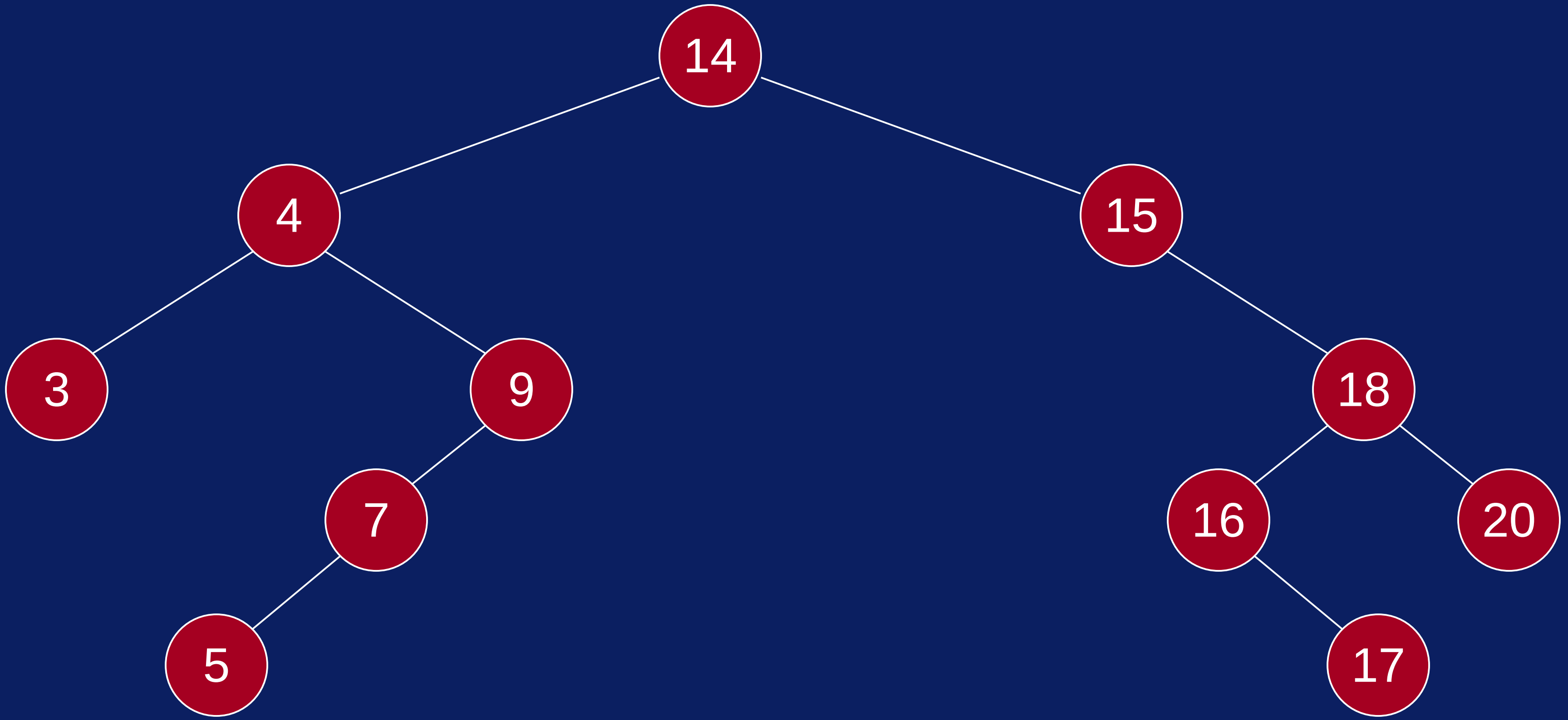
```
cout << "inorder: ";   preorder( root);
cout << "inorder: ";   inorder( root );
cout << "postorder: "; postorder( root );
```

Traversing a Binary Tree



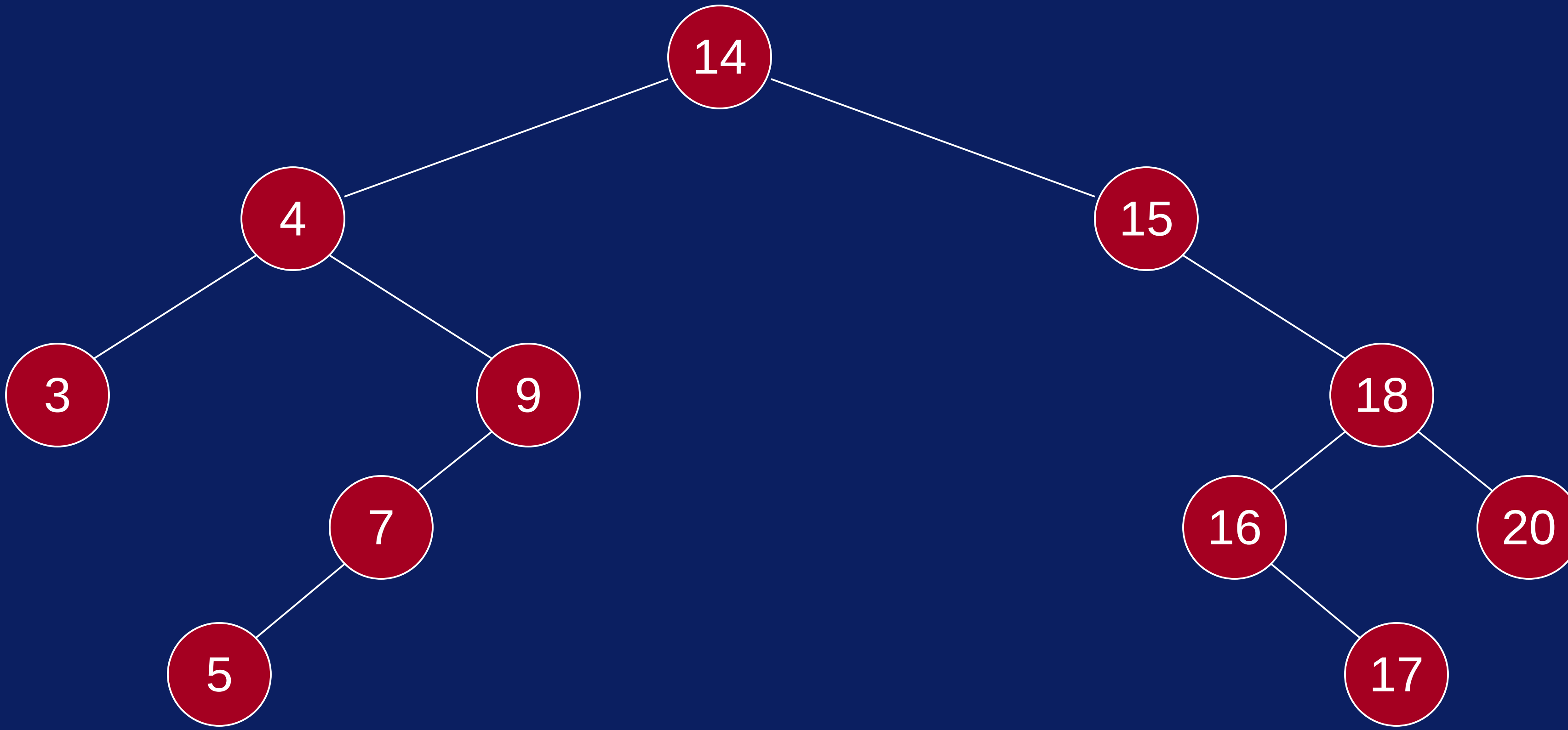
Preorder: 14 4 3 9 7 5 15 18 16 17 20

Traversing a Binary Tree



Inorder: 3 4 5 7 9 14 15 16 17 18 20

Traversing a Binary Tree



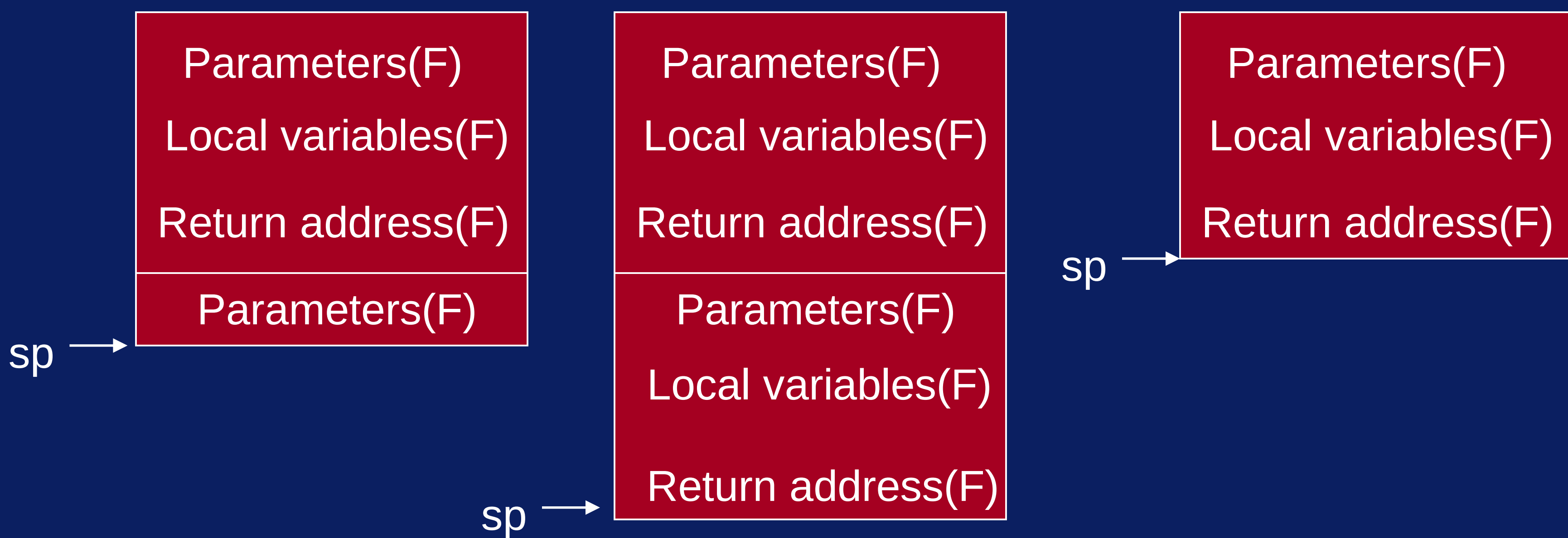
Postorder: 3 5 7 9 4 17 16 20 18 15 14

Recursive Call

- Recall that a stack is used during function calls.
- The caller function places the arguments on the stack and passes control to the called function.
- Local variables are allocated storage on the call stack.
- Calling a function itself makes no difference as far as the call stack is concerned.

Stack Layout during a call

- Here is stack layout when function F calls function F (recursively):



At point of call | During execution of F | After call