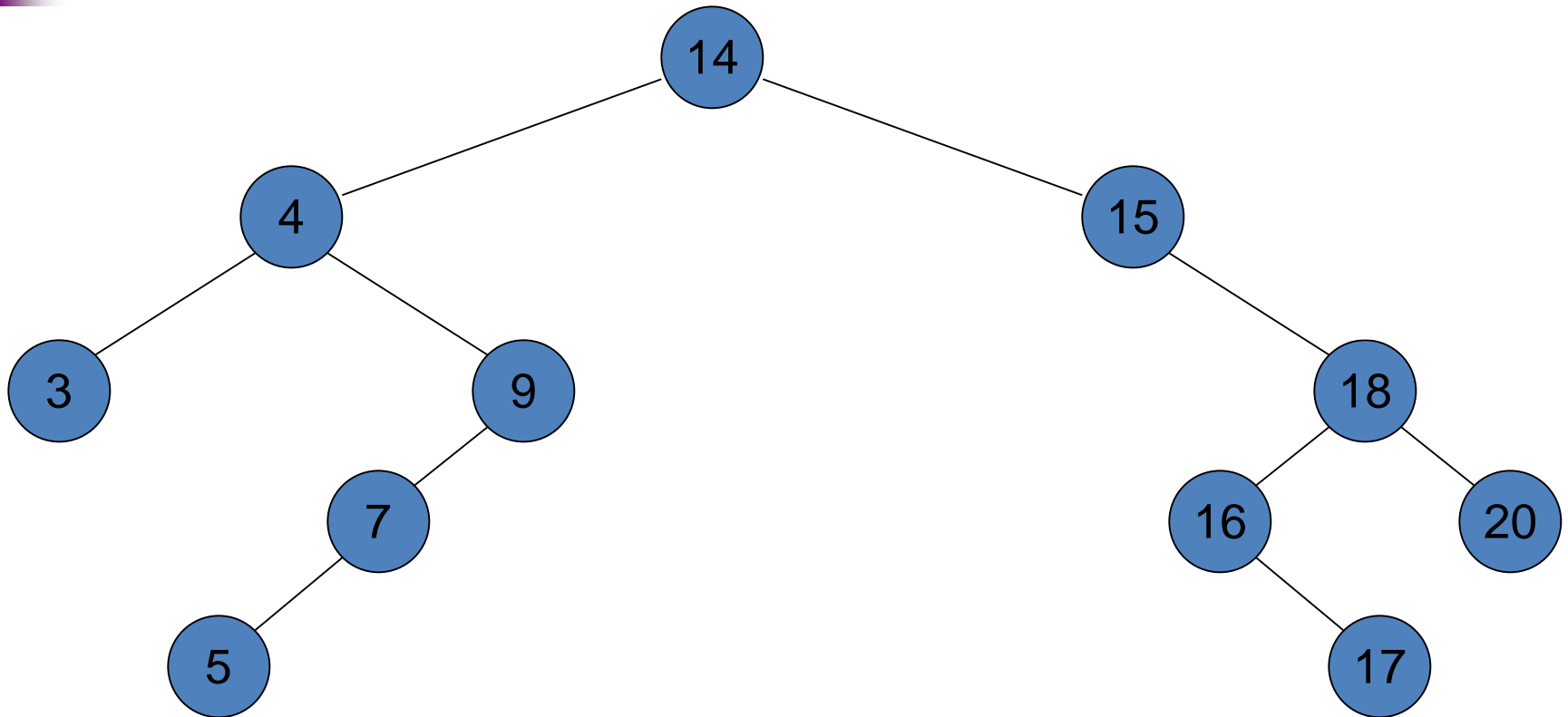




Tree Traversal

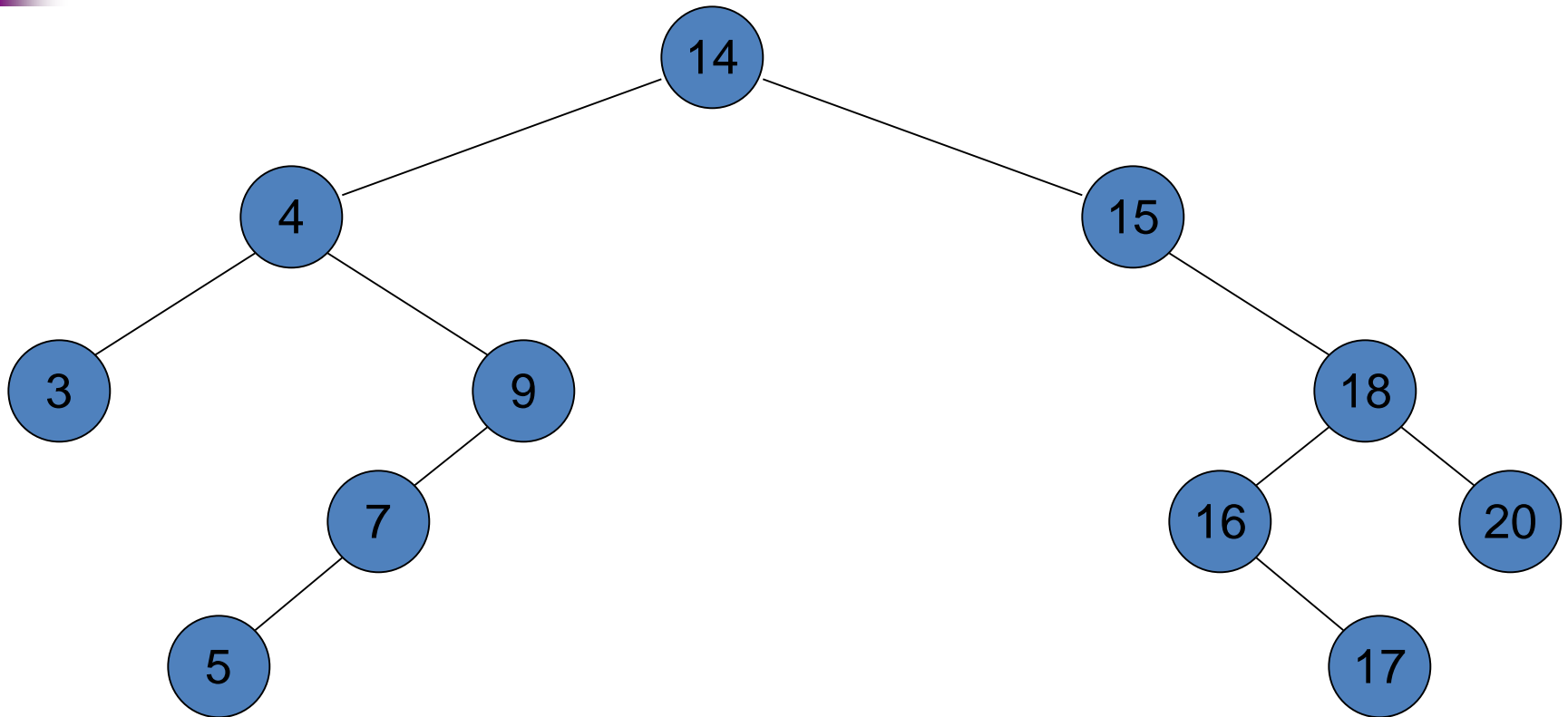
Home task solution and non
recursive traversal implemenation

Traversing a Binary Tree



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

Traversing a Binary Tree



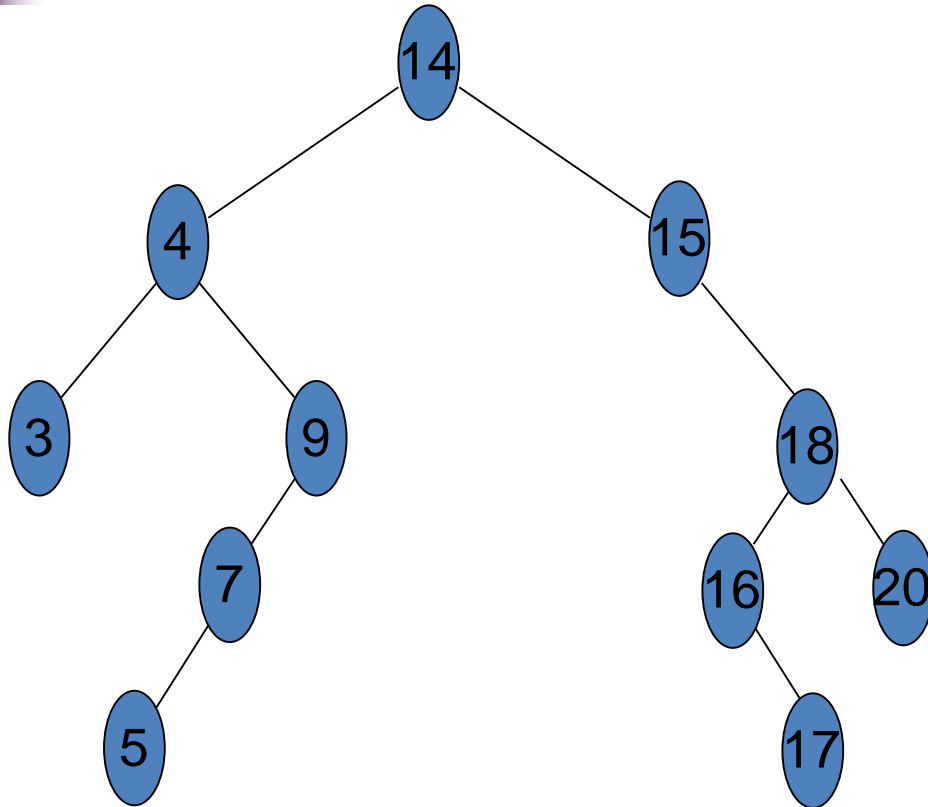
Inorder: 3 4 5 7 9 14 15 16 17 18 20



Recursive Call

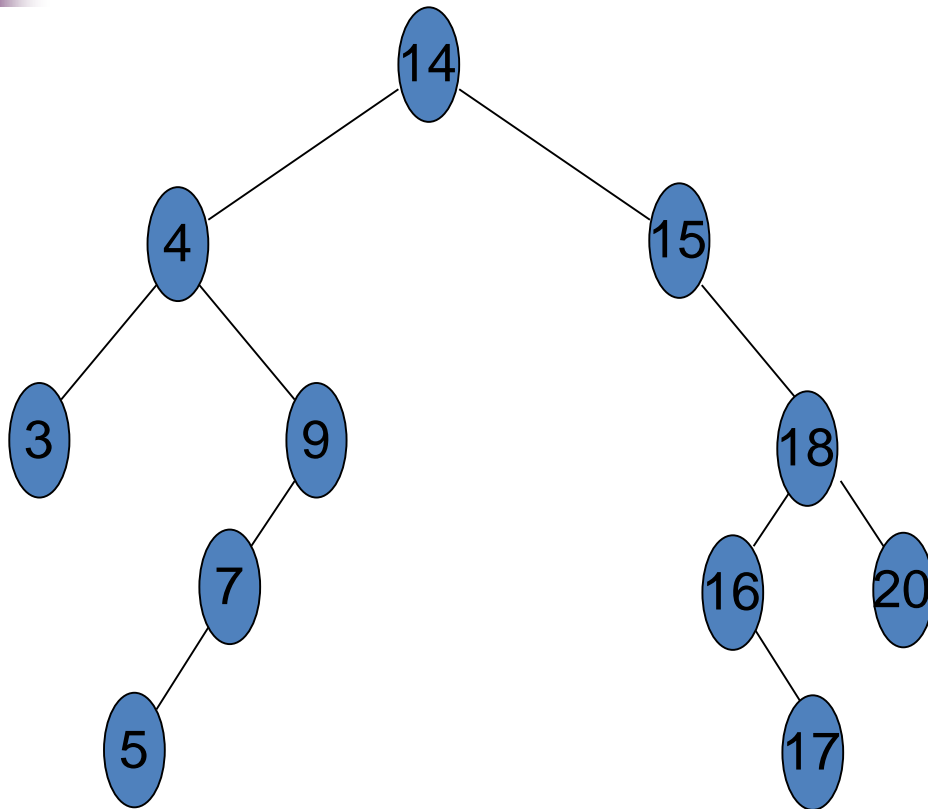
- Recall that a stack is used during function calls.
- The caller function places the arguments on the stack and passes control to the called function.
- Local variables are allocated storage on the call stack.
- Calling a function itself makes no difference as far as the call stack is concerned.

Recursion: preorder



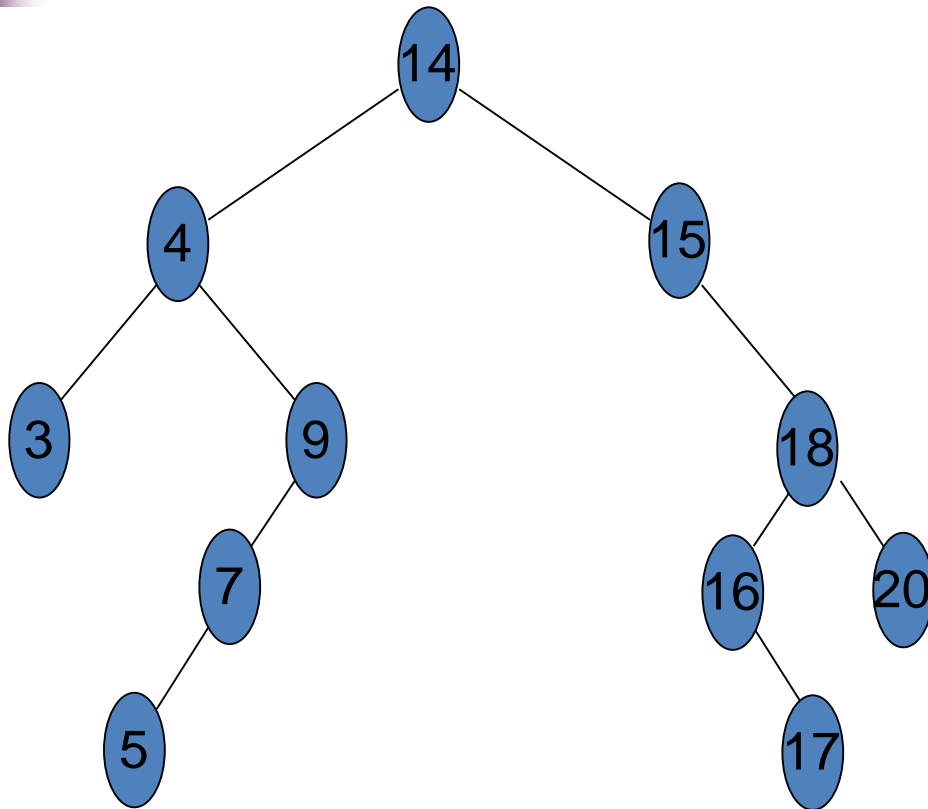
```
preorder(14)
14
  ..preorder(4)
  4
    ....preorder(3)
    3
      .....preorder(null)
      .....preorder(null)
      ....preorder(9)
      9
        .....preorder(7)
        7
          .....preorder(5)
          5
            .....preorder(null)
            .....preorder(null)
            .....preorder(null)
            .....preorder(null)
```

Recursion: preorder



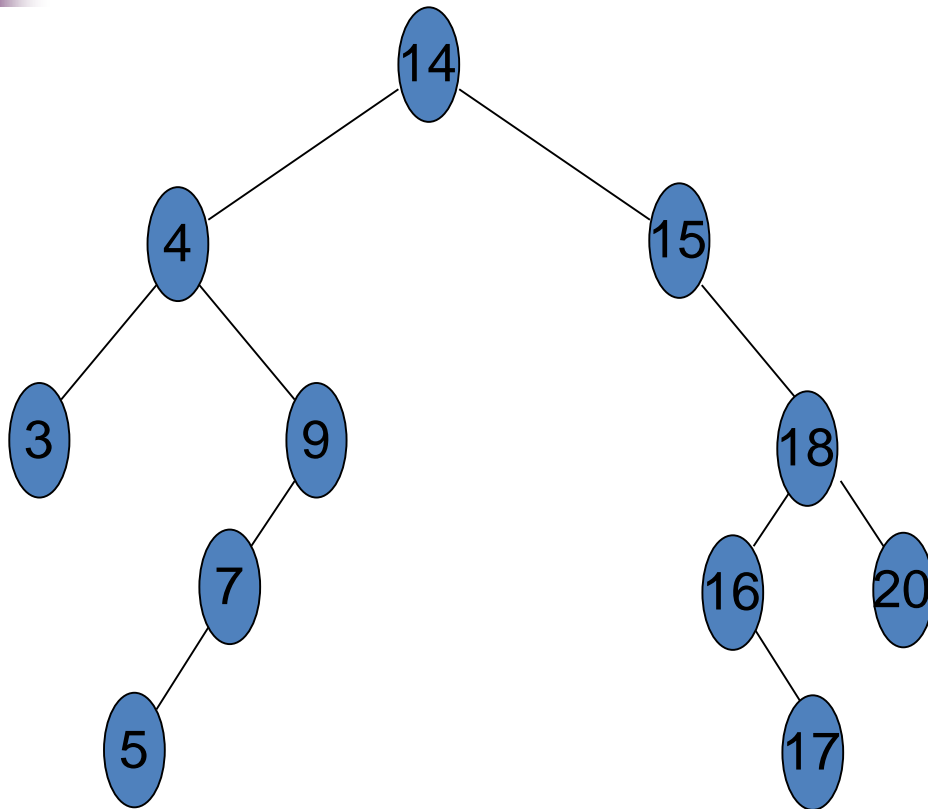
```
..preorder(15)
15
....preorder(null)
....preorder(18)
18
.....preorder(16)
16
.....preorder(null)
.....preorder(17)
17
.....preorder(null)
.....preorder(null)
.....preorder(20)
20
.....preorder(null)
.....preorder(null)
```

Recursion: inorder



```
inorder(14)
..inorder(4)
....inorder(3)
.....inorder(null)
3
.....inorder(null)
4
....inorder(9)
.....inorder(7)
.....inorder(5)
.....inorder(null)
5
.....inorder(null)
7
.....inorder(null)
9
.....inorder(null)
14
```

Recursion: inorder



```
..inorder(15)
....inorder(null)
15
....inorder(18)
.....inorder(16)
.....inorder(null)
16
.....inorder(17)
.....inorder(null)
17
.....inorder(null)
18
.....inorder(20)
.....inorder(null)
20
.....inorder(null)
```




Non Recursive Traversal

- We can implement non-recursive versions of the preorder, inorder and postorder traversal by using an explicit stack.
- The stack will be used to store the tree nodes in the appropriate order.
- Here, for example, is the routine for inorder traversal that uses a stack.

Non Recursive Traversal

```
void inorder(TreeNode<int>* root)
```

```
{
```

```
    Stack<TreeNode<int>* > stack;
```

```
    TreeNode<int>* p;
```

```
    p = root;
```

```
    do
```

```
    {
```

```
        while( p != NULL )
```

```
        {
```

```
            stack.push( p );
```

```
            p = p->getLeft();
```

```
        }
```

```
        // at this point, left tree is empty
```

Non Recursive Traversal

```
void inorder(TreeNode<int>* root)
{
    Stack<TreeNode<int>* > stack;
    TreeNode<int>* p;
    ➡ p = root;
    do
    {
        while( p != NULL )
        {
            stack.push( p );
            p = p->getLeft();
        }
        // at this point, left tree is empty
    }
```

Non Recursive Traversal

```
void inorder(TreeNode<int>* root)
{
    Stack<TreeNode<int>* > stack;
    TreeNode<int>* p;
    p = root;
    do
    {
```

```
        while( p != NULL )
        {
            stack.push( p );
            p = p->getLeft();
        }
```

```
    // at this point, left tree is empty
```

Non Recursive Traversal

```
➔ if( !stack.empty() )
```

```
{
```

```
    p = stack.pop();
```

```
    cout << *(p->getInfo()) << " ";
```

```
    // go back & traverse right subtree
```

```
    p = p->getRight();
```

```
}
```

```
} while ( !stack.empty() || p != NULL );
```

```
}
```

Non Recursive Traversal

```
if( !stack.empty() )  
{
```

```
    p = stack.pop();  
    cout << *(p->getInfo()) << " ";  
    // go back & traverse right subtree  
    p = p->getRight();
```

```
}
```

```
} while ( !stack.empty() || p != NULL );
```

```
}
```

Non Recursive Traversal

```
if( !stack.empty() )  
{
```

```
    p = stack.pop();
```

```
    cout << *(p->getInfo()) << " ";
```

```
    // go back & traverse right subtree
```

```
    p = p->getRight();
```

```
}
```

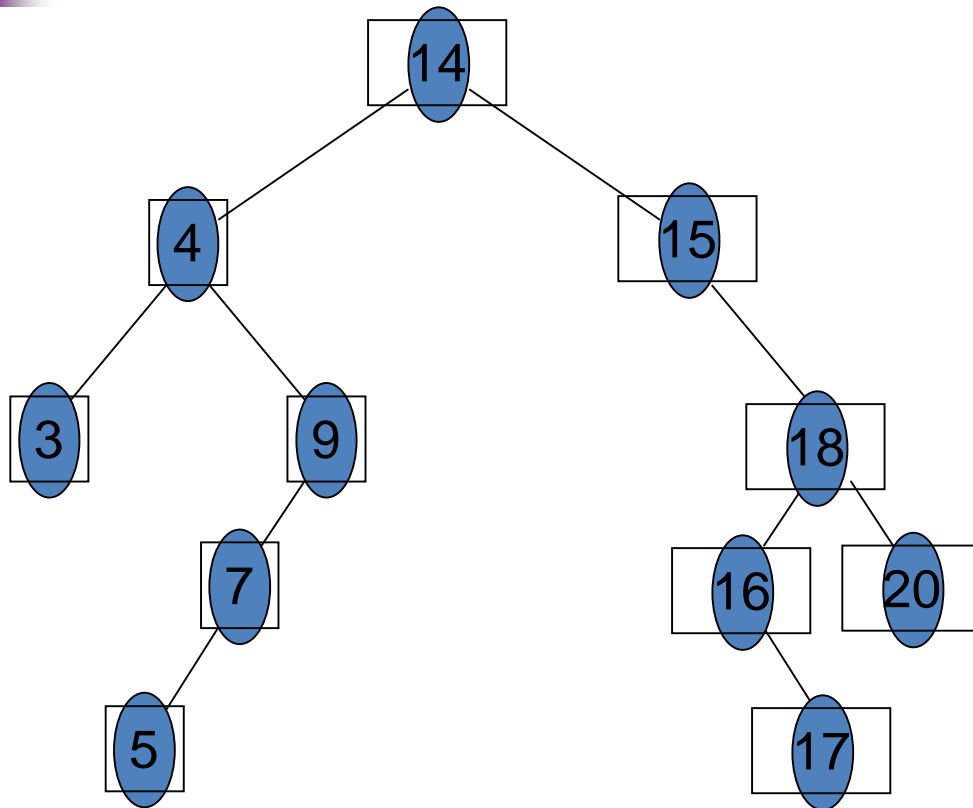
```
} while ( !stack.empty() || p != NULL );
```

```
}
```

Non Recursive Traversal

```
if( !stack.empty() )
{
    p = stack.pop();
    cout << *(p->getInfo()) << " ";
    // go back & traverse right subtree
    p = p->getRight();
}
} while ( !stack.empty() || p != NULL );
}
```


Nonrecursive Inorder



```
push(14)
..push(4)
....push(3)
3
4
..push(9)
....push(7)
.....push(5)
5
7
9
14
push(15)
15
push(18)
..push(16)
16
..push(17)
17
18
push(20)
20
```

Traversal Trace

recursive inorder

nonrecursive inorder

```
inorder(14)
..inorder(4)
....inorder(3)
3
4
..inorder(9)
....inorder(7)
.....inorder(5)
5
7
9
14
inorder(15)
15
inorder(18)
..inorder(16)
16
..inorder(17)
17
18
inorder(20)
20
```

```
push(14)
..push(4)
....push(3)
3
4
..push(9)
....push(7)
.....push(5)
5
7
9
14
push(15)
15
push(18)
..push(16)
16
..push(17)
17
18
push(20)
20
```

Traversal Trace

recursive inorder

```
inorder(14)
..inorder(4)
....inorder(3)
3
4
..inorder(9)
....inorder(7)
.....inorder(5)
5
7
9
14
inorder(15)
15
inorder(18)
..inorder(16)
16
..inorder(17)
17
18
inorder(20)
20
```

nonrecursive inorder

```
push(14)
..push(4)
....push(3)
3
4
..push(9)
....push(7)
.....push(5)
5
7
9
14
push(15)
15
push(18)
..push(16)
16
..push(17)
17
18
push(20)
20
```

Traversal Trace

recursive inorder

```
inorder(14)
..inorder(4)
....inorder(3)
3
4
➤ ..inorder(9)
....inorder(7)
.....inorder(5)
5
7
9
14
inorder(15)
15
inorder(18)
..inorder(16)
16
..inorder(17)
17
18
inorder(20)
20
```

nonrecursive inorder

```
push(14)
..push(4)
....push(3)
3
4
..push(9)
....push(7)
.....push(5)
5
7
9
14
push(15)
15
push(18)
..push(16)
16
..push(17)
17
18
push(20)
20
```

Traversal Trace

recursive inorder

```
inorder(14)
..inorder(4)
....inorder(3)
3
4
..inorder(9)
....inorder(7)
.....inorder(5)
5
7
9
14
inorder(15)
15
inorder(18)
..inorder(16)
16
..inorder(17)
17
18
inorder(20)
20
```

nonrecursive inorder

```
push(14)
..push(4)
....push(3)
3
4
..push(9)
....push(7)
.....push(5)
5
7
9
14
push(15)
15
push(18)
..push(16)
16
..push(17)
17
18
push(20)
20
```