Lecture # 10

Operations on Binary Tree

- There are a number of operations that can be defined for a binary tree.
  - If *p* is pointing to a node in an existing tree then
  - left(*p*) returns pointer to the left subtree
  - right(*p*) returns pointer to right subtree
  - parent(*p*) returns the father of *p*
  - brother(*p*) returns brother of *p*.
  - info(p) returns content of the node.

Operations on Binary Tree

- In order to construct a binary tree, the following can be useful:
- setLeft(p,x) creates the left child node of p. The child node contains the info 'x'.
- setRight(p,x) creates the right child node of p. The child node contains the info 'x'.

Applications of Binary Trees

- A binary tree is a useful data structure when two-way decisions must be made at each point in a process.

- For example, suppose we wanted to find all duplicates in a list of numbers:

  14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

Applications of Binary Trees

- One way of finding duplicates is to compare each number with all those that precede it.

14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

14, 15, $\boxed{4}$, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

Searching for Duplicates

- If the list of numbers is large and is growing, this procedure involves a large number of comparisons.
- A linked list could handle the growth but the comparisons would still be large.
- The number of comparisons can be drastically reduced by using a binary tree.
- The tree grows dynamically like the linked list.

Searching for Duplicates

- The binary tree is built in a special way.
- The first number in the list is placed in a node that is designated as the root of a binary tree.

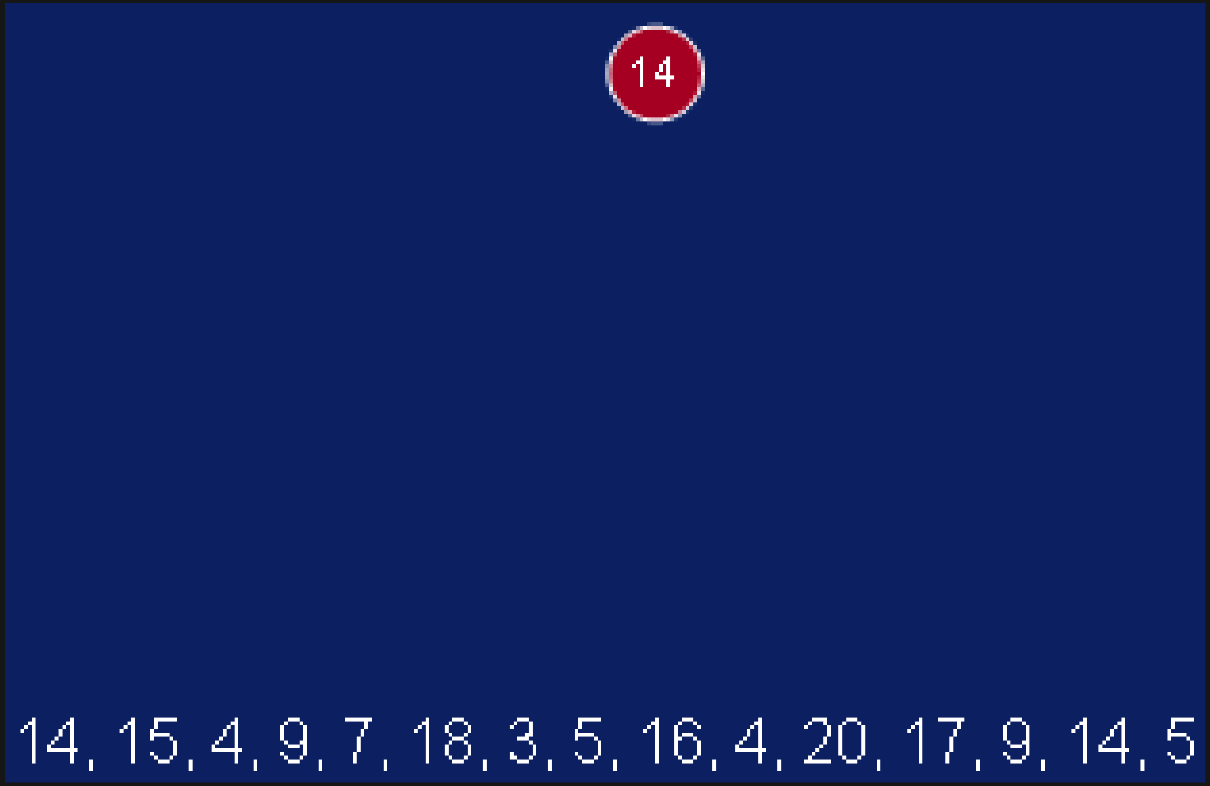
- Initially, both left and right subtrees of the root are empty.

- We take the next number and compare it with the number placed in the root.
- If it is the same then we have a duplicate.

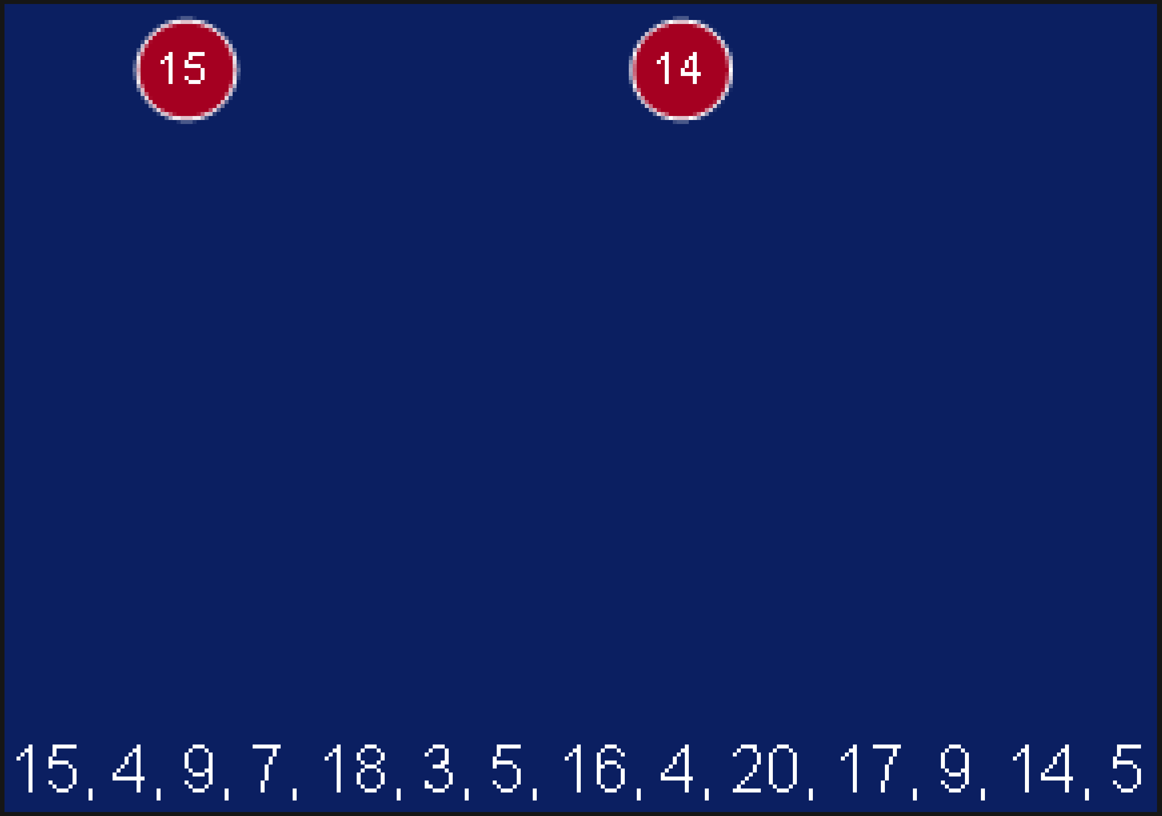
Searching for Duplicates

- Otherwise, we create a new tree node and put the new number in it.
- The new node is made the left child of the root node if the second number is less than the one in the root.
- The new node is made the right child if the number is greater than the one in the root.

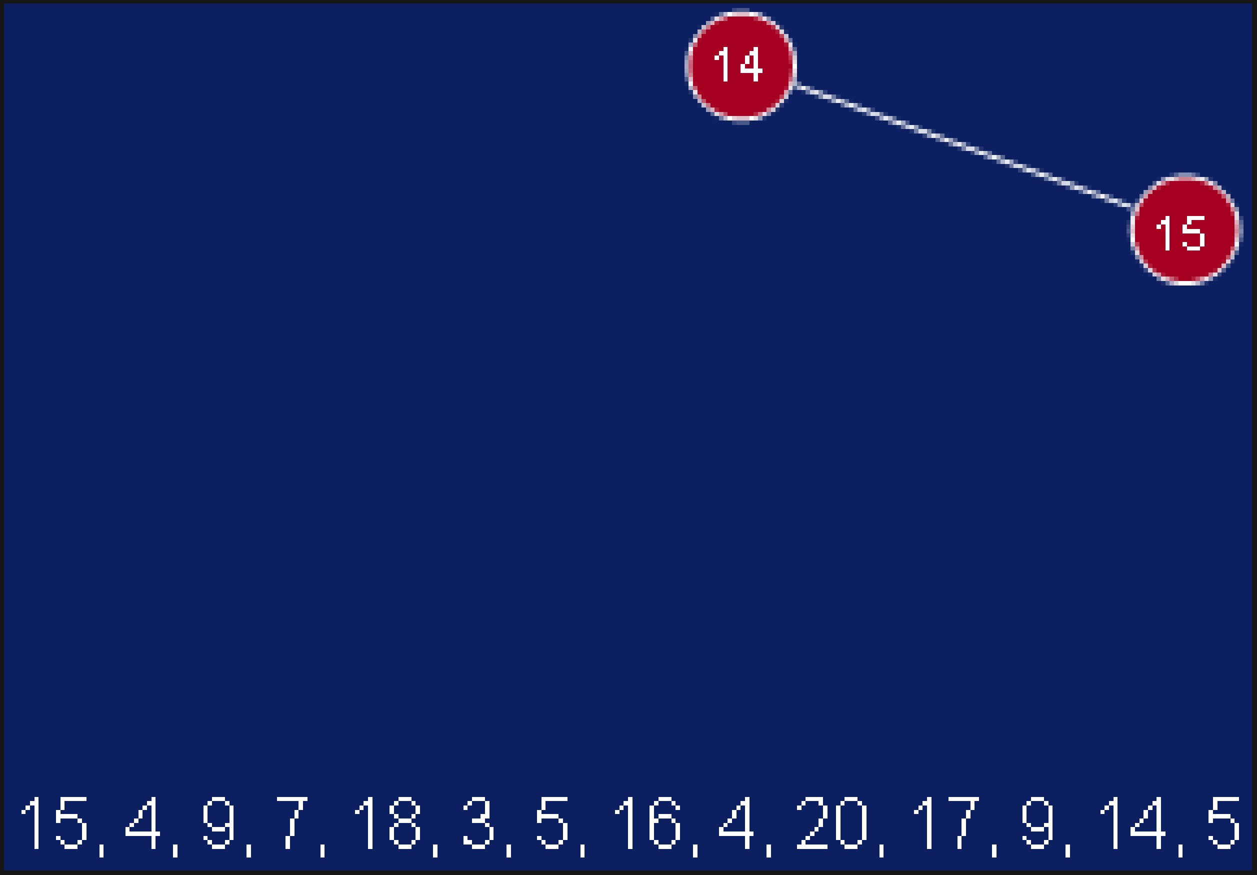
Searching for Duplicates



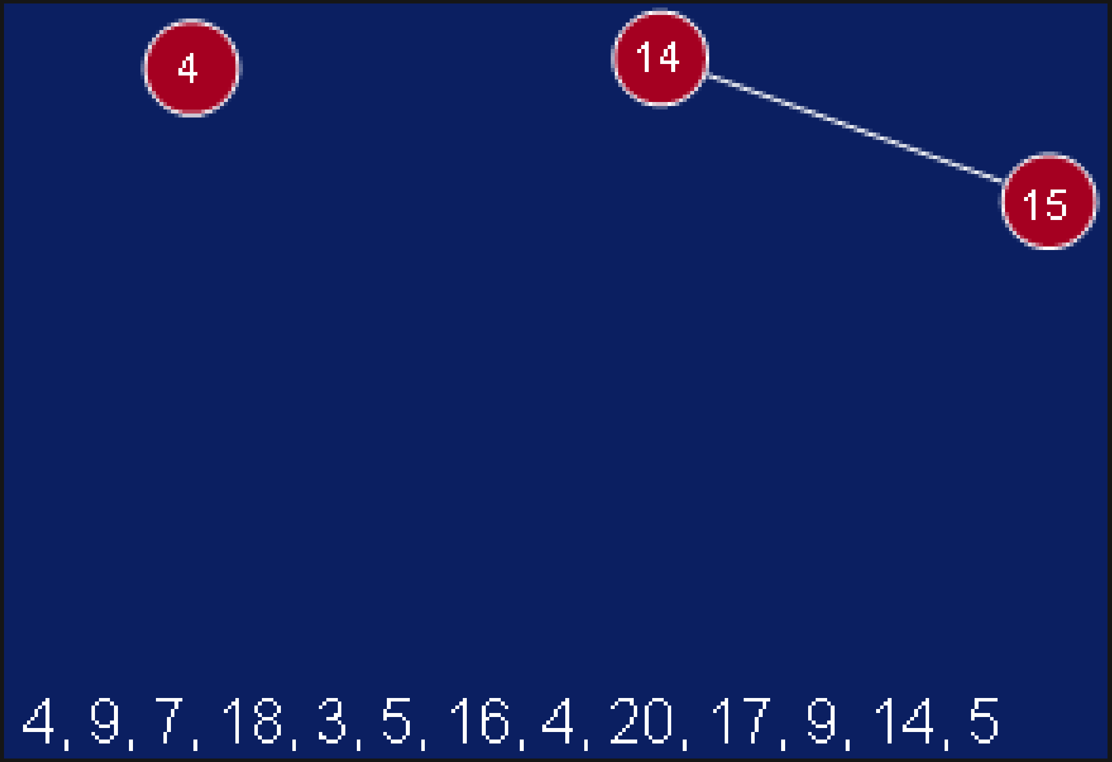
Searching for Duplicates



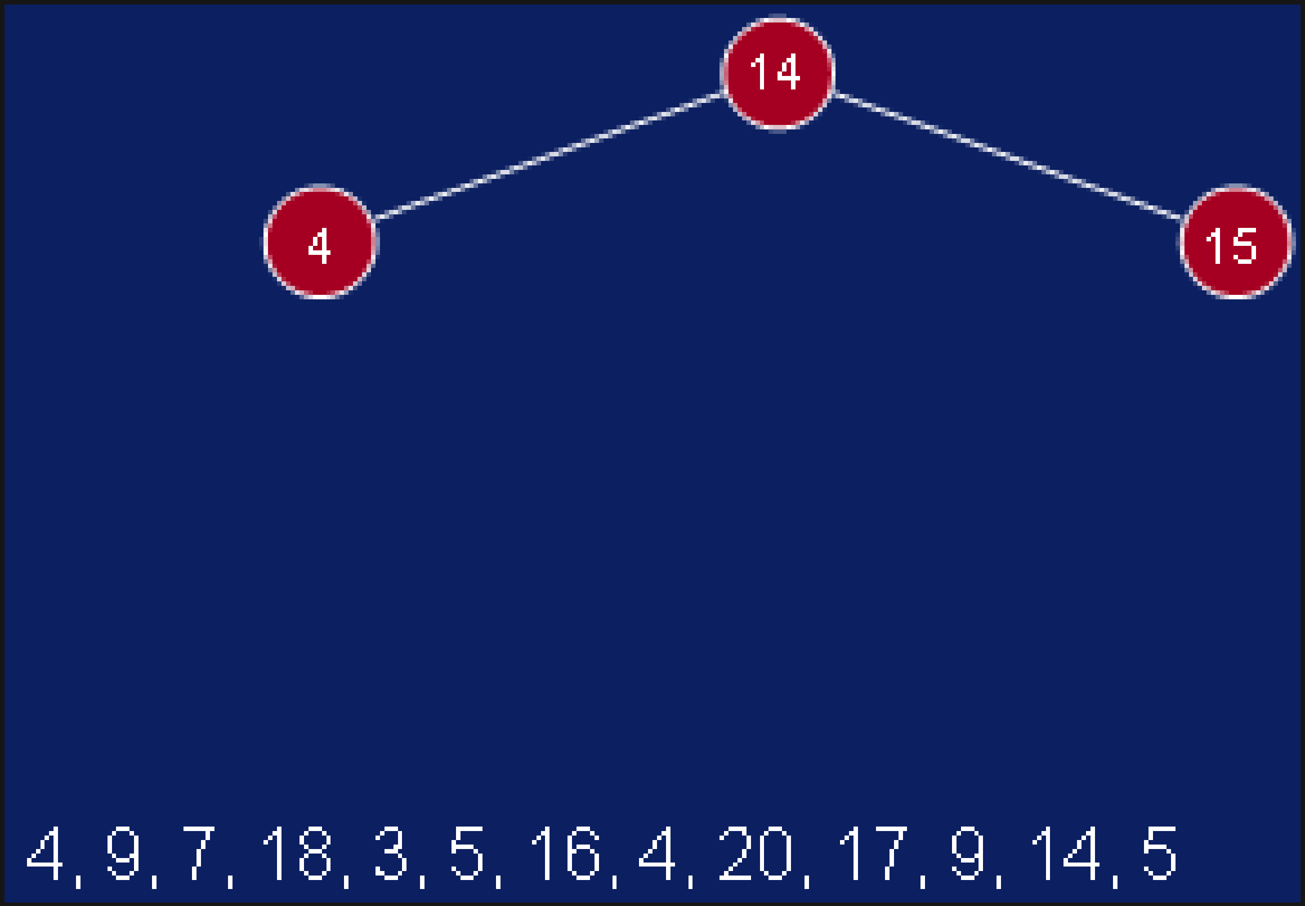
Searching for Duplicates



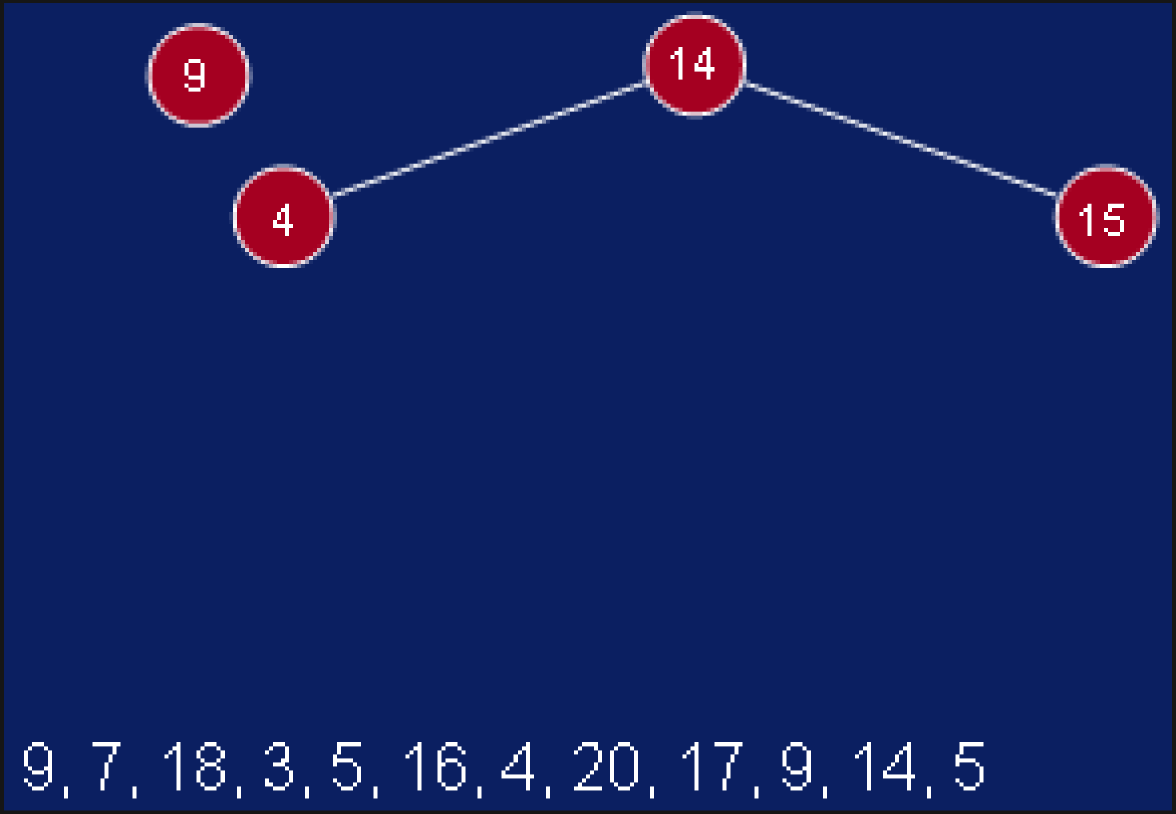
Searching for Duplicates



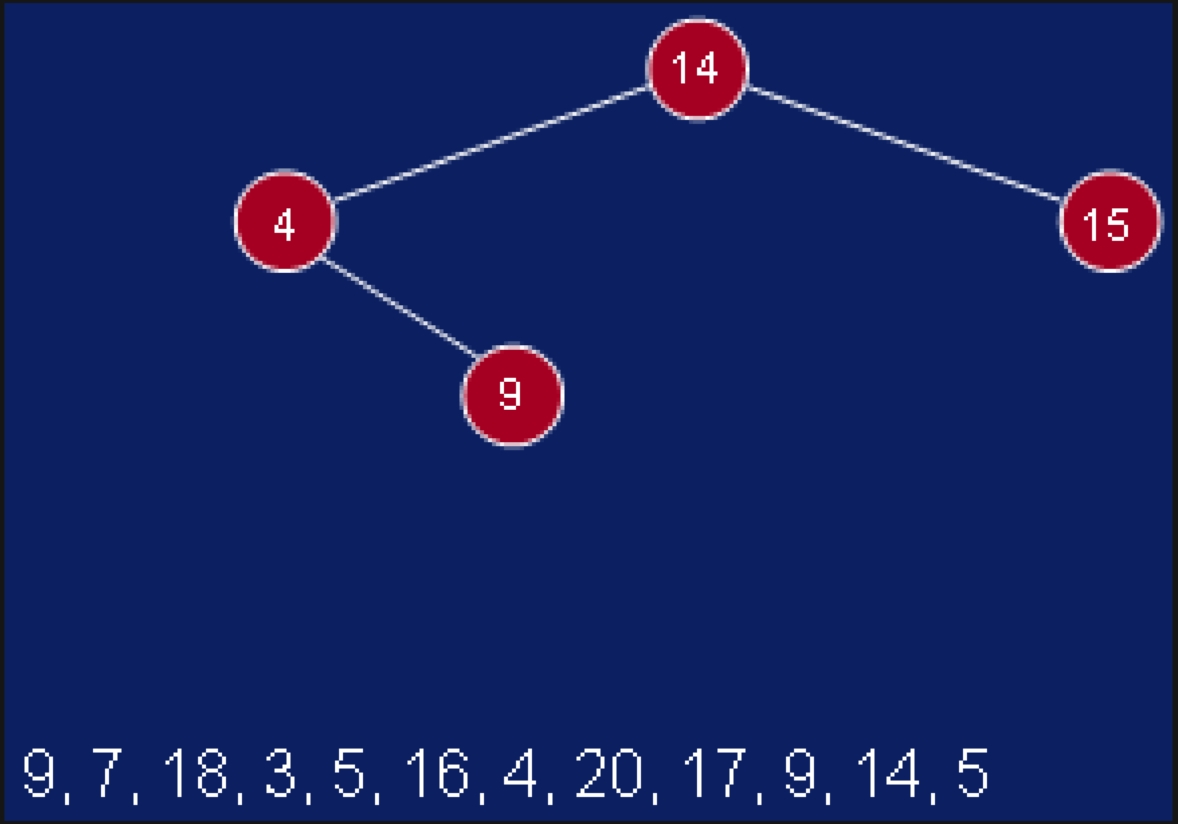
Searching for Duplicates



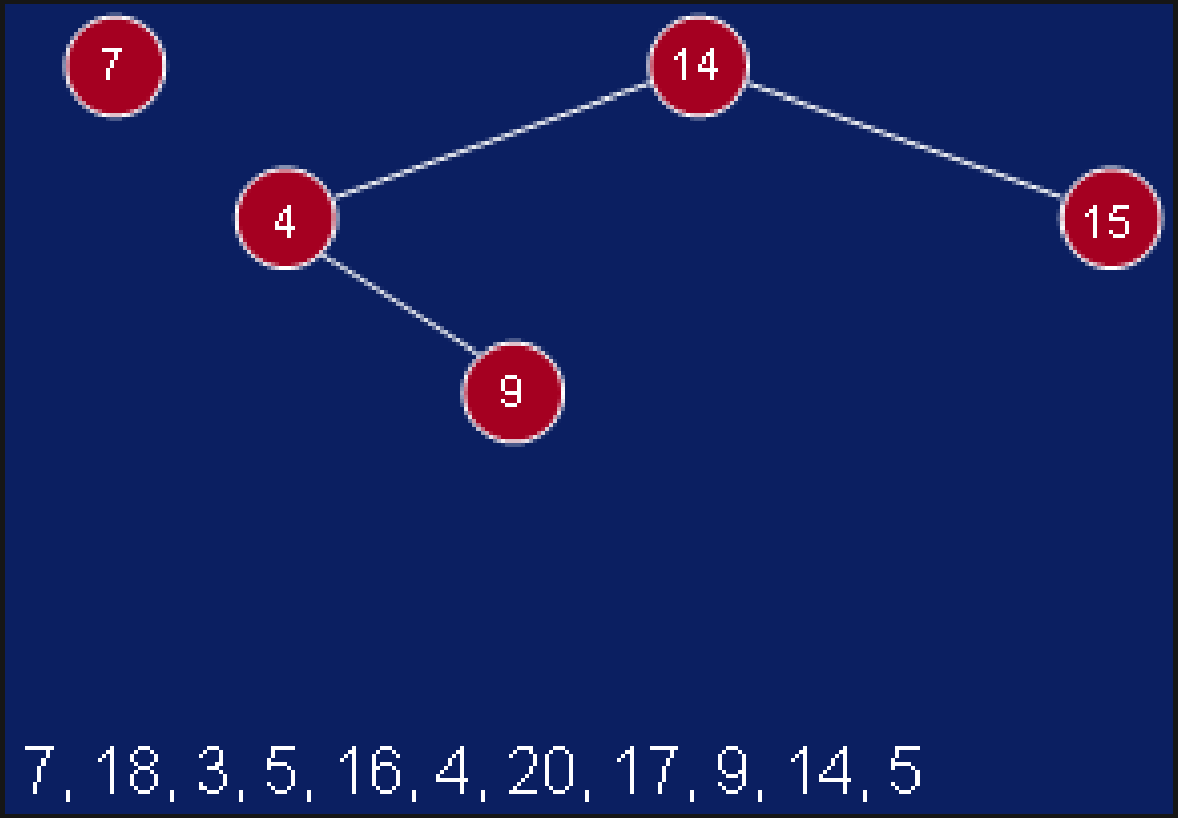
Searching for Duplicates



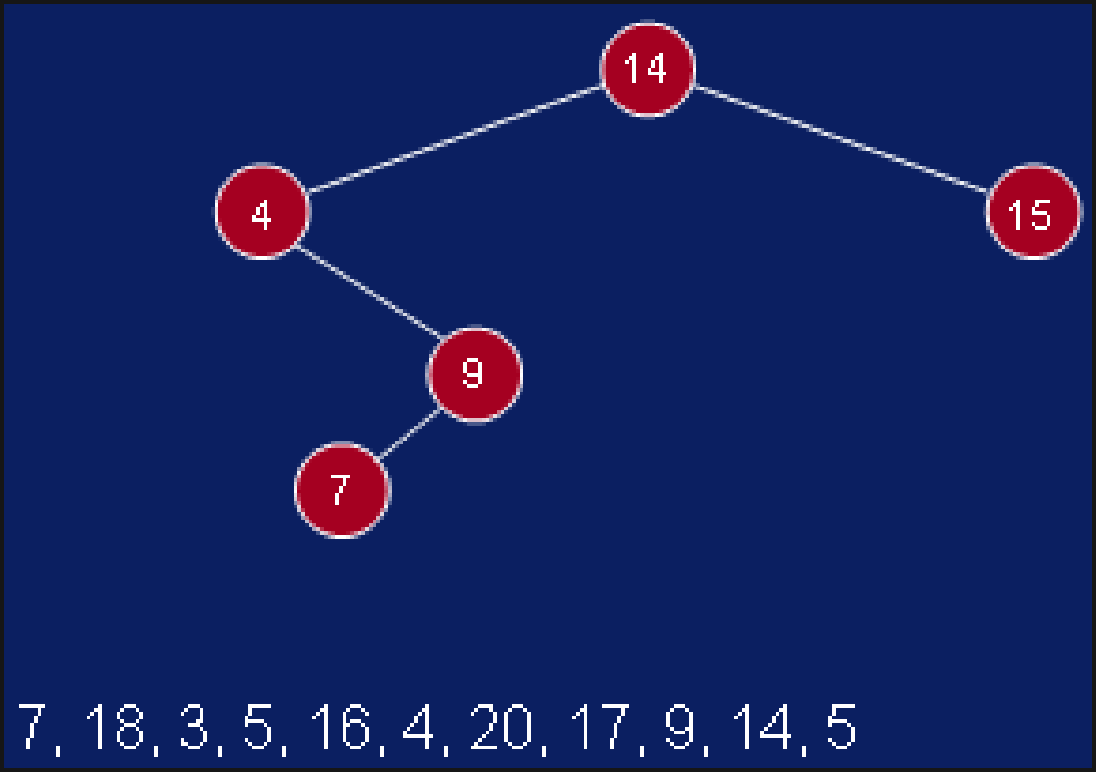
Searching for Duplicates

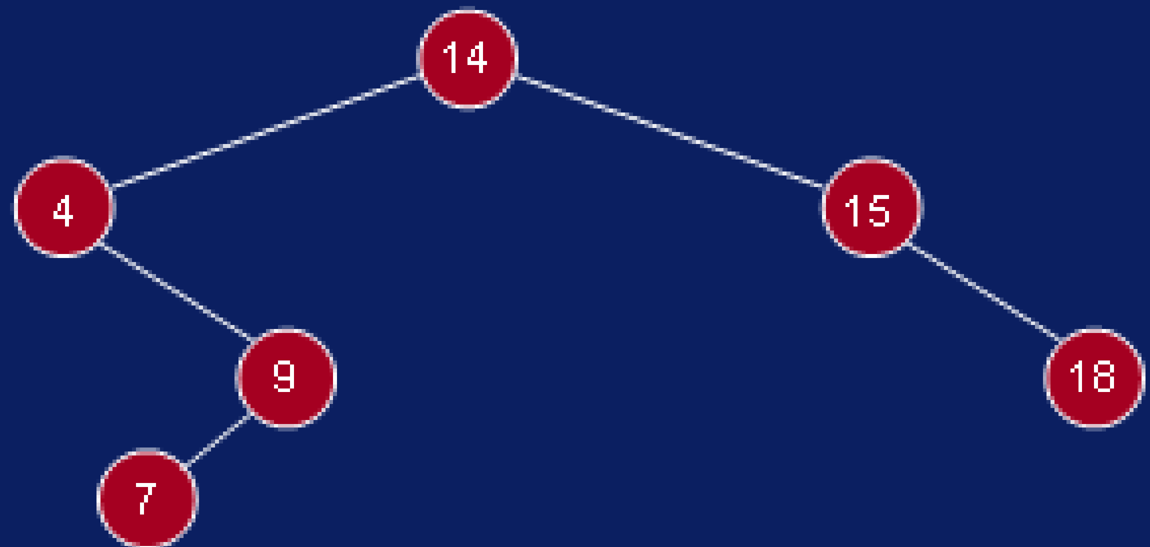


Searching for Duplicates

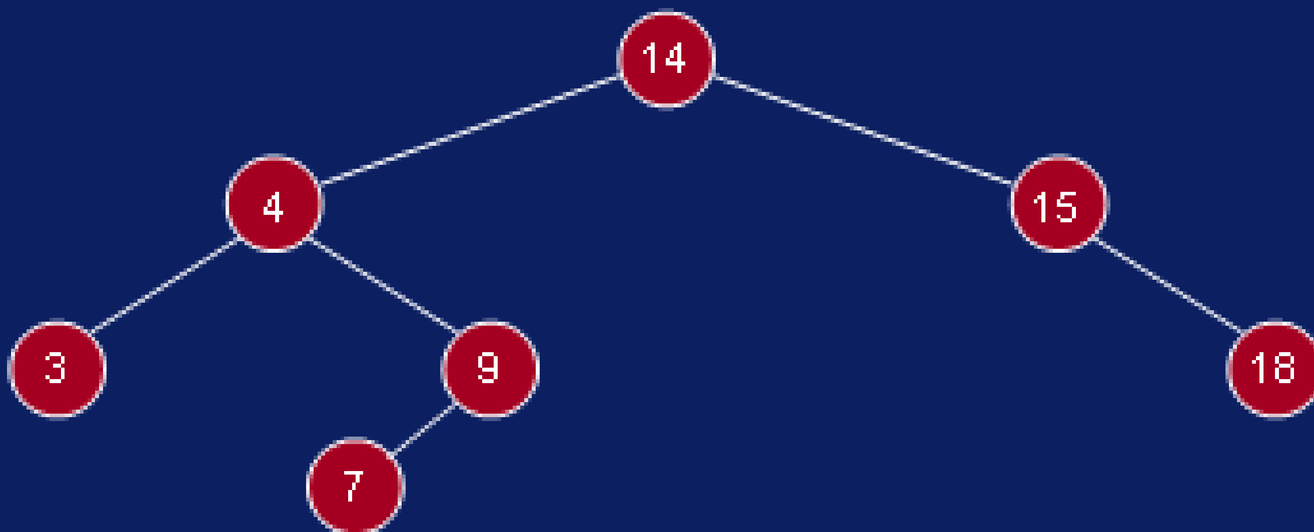


Searching for Duplicates

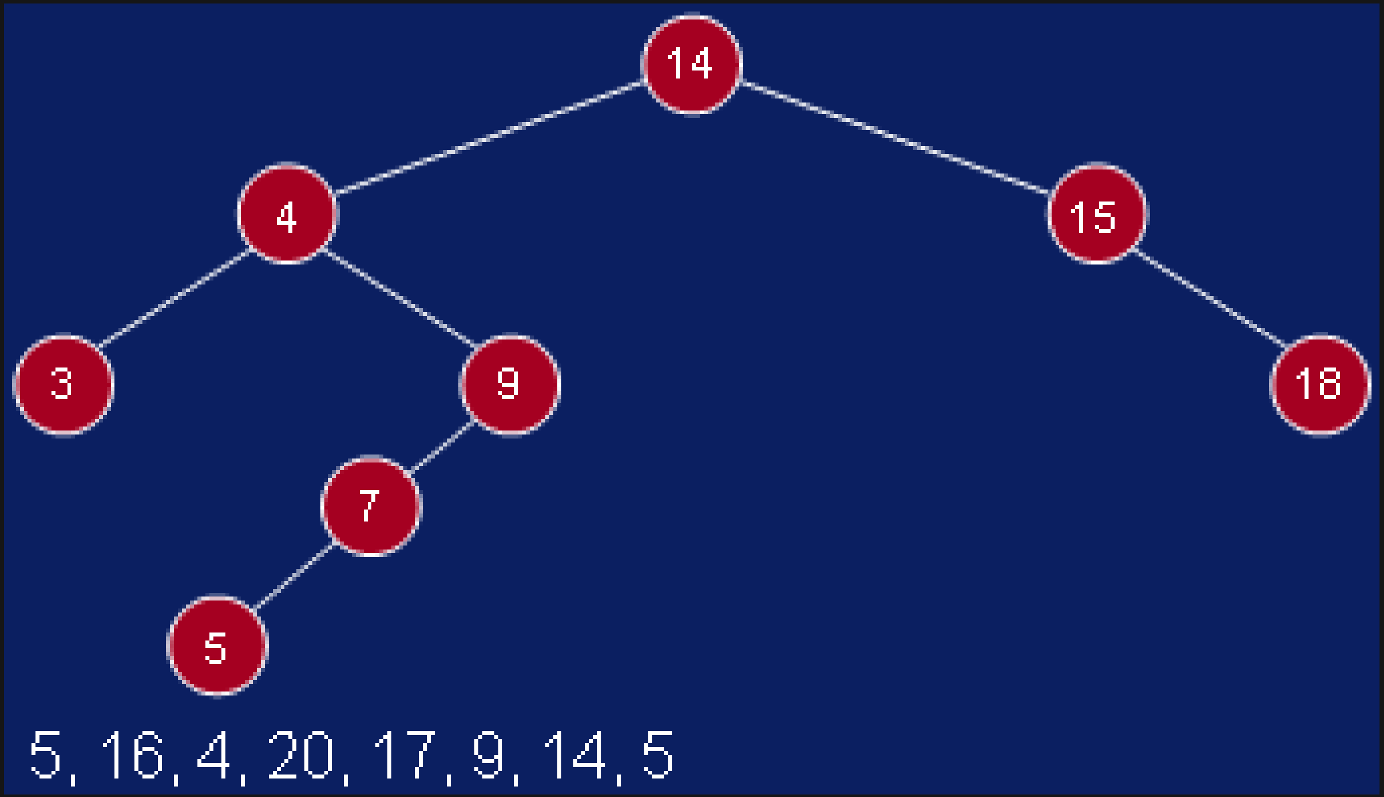


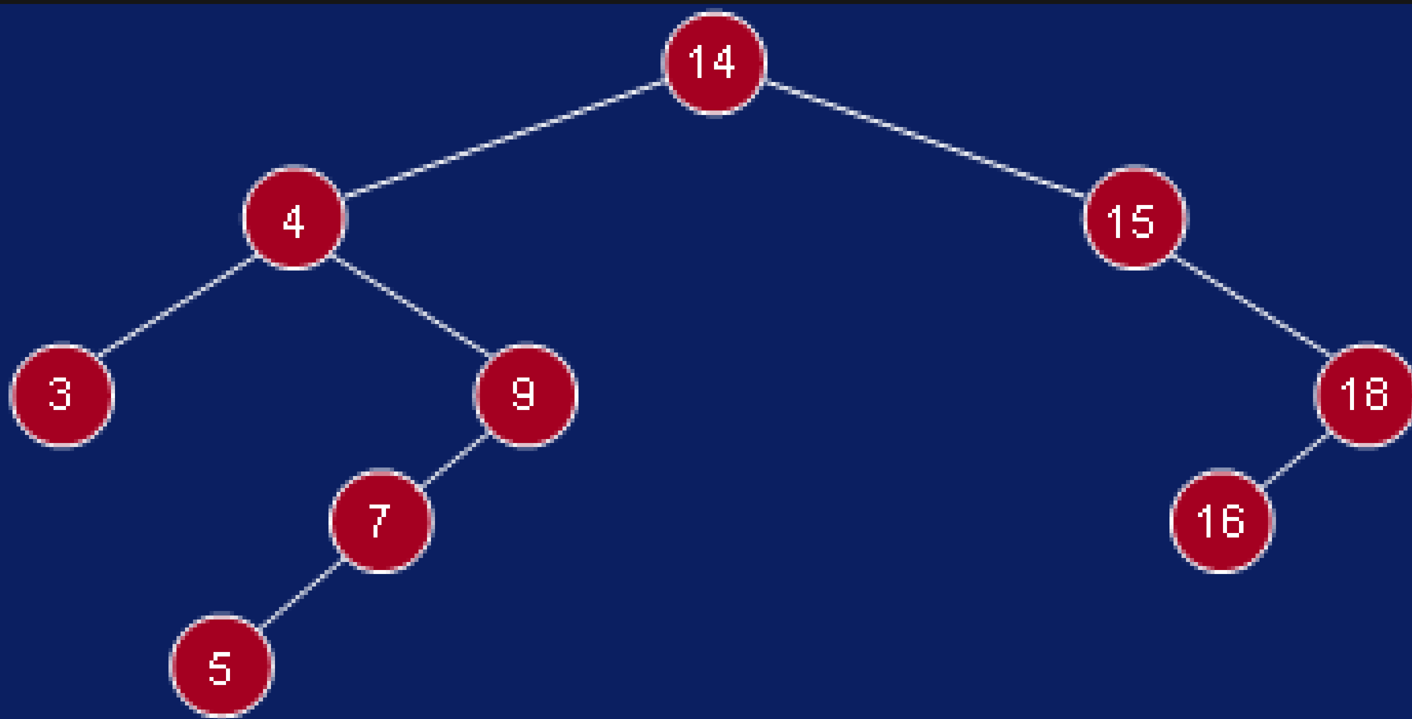


18, 3, 5, 16, 4, 20, 17, 9, 14, 5

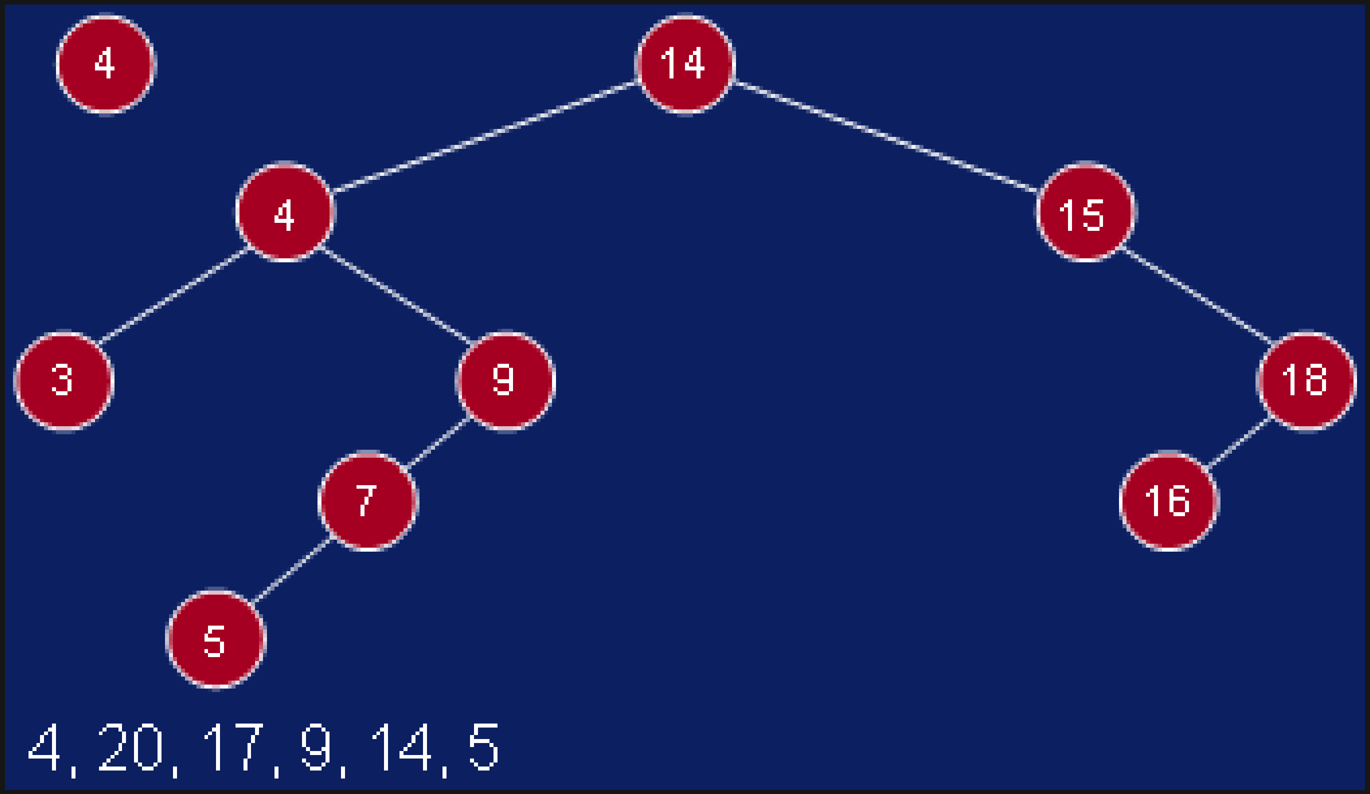


3, 5, 16, 4, 20, 17, 9, 14, 5

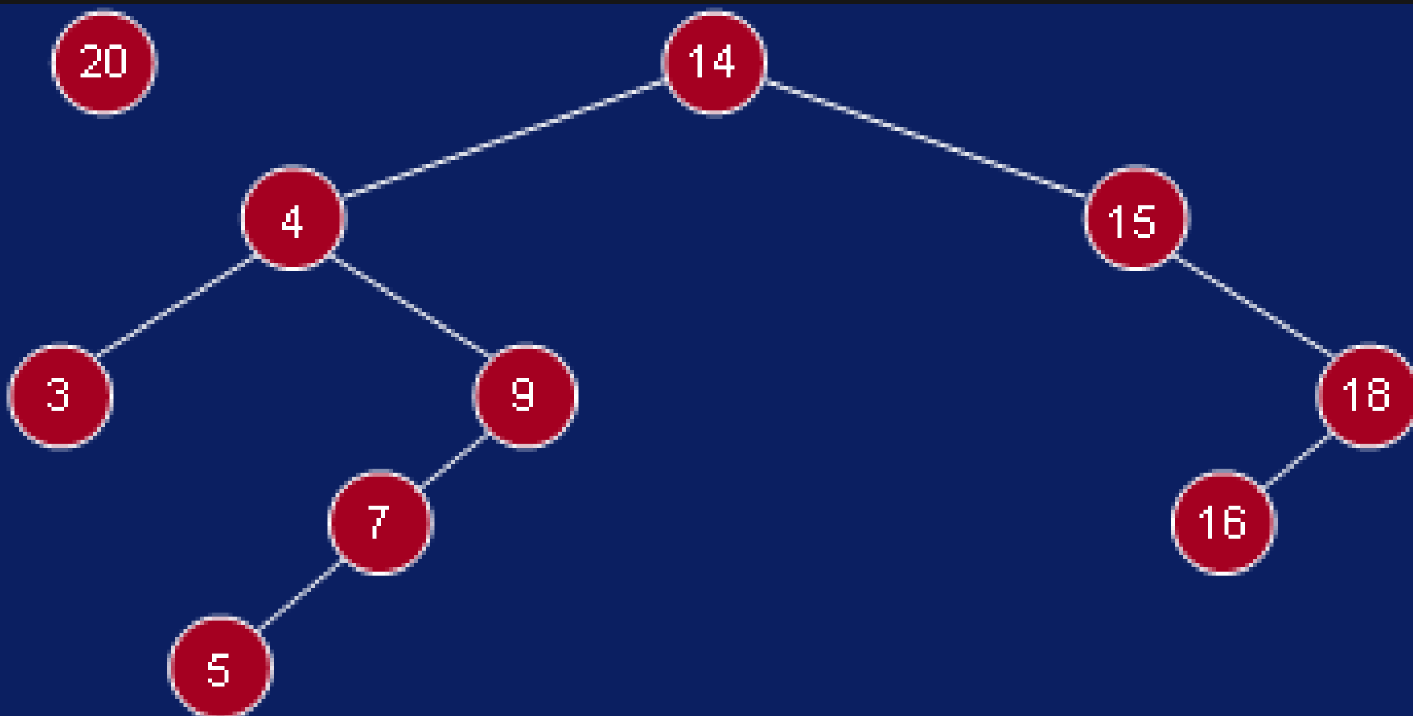

14
4
15
3
9
18
7
5
5, 16, 4, 20, 17, 9, 14, 5



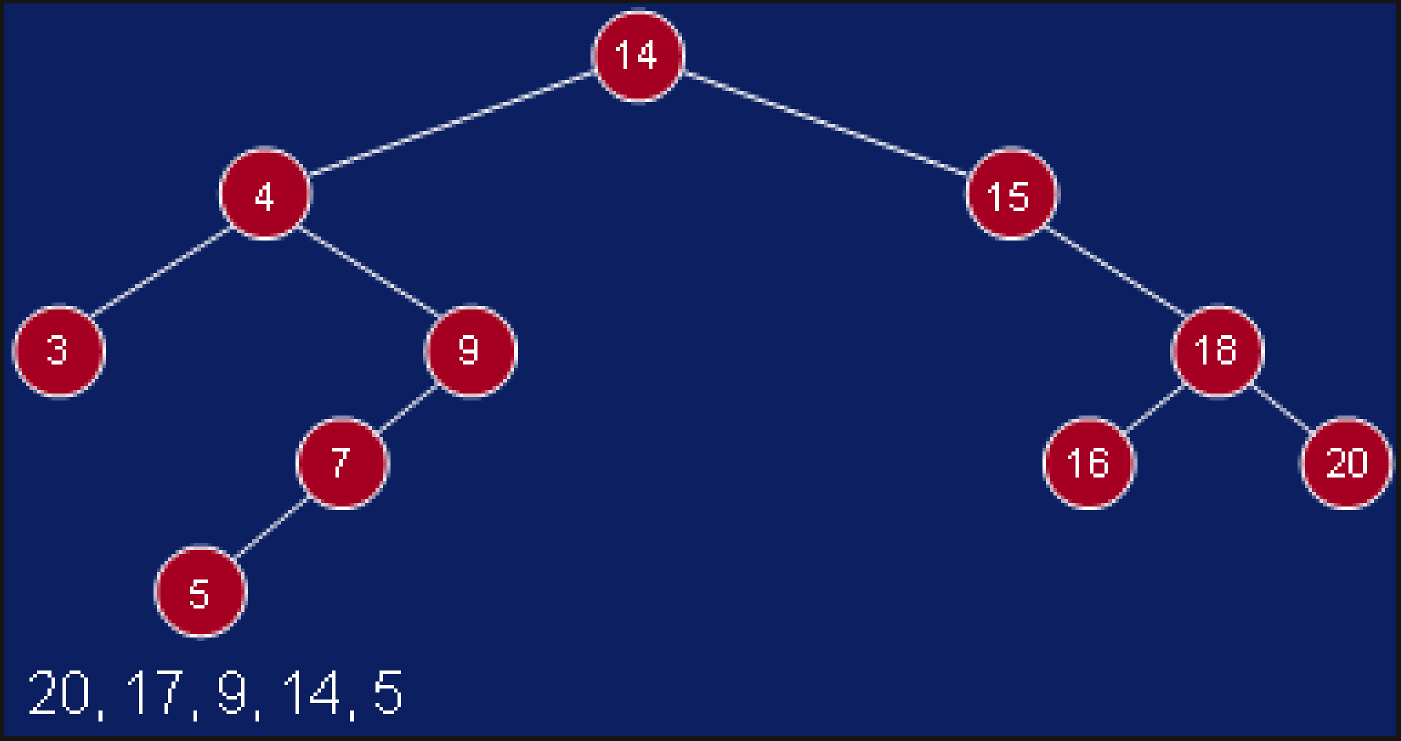
16, 4, 20, 17, 9, 14, 5

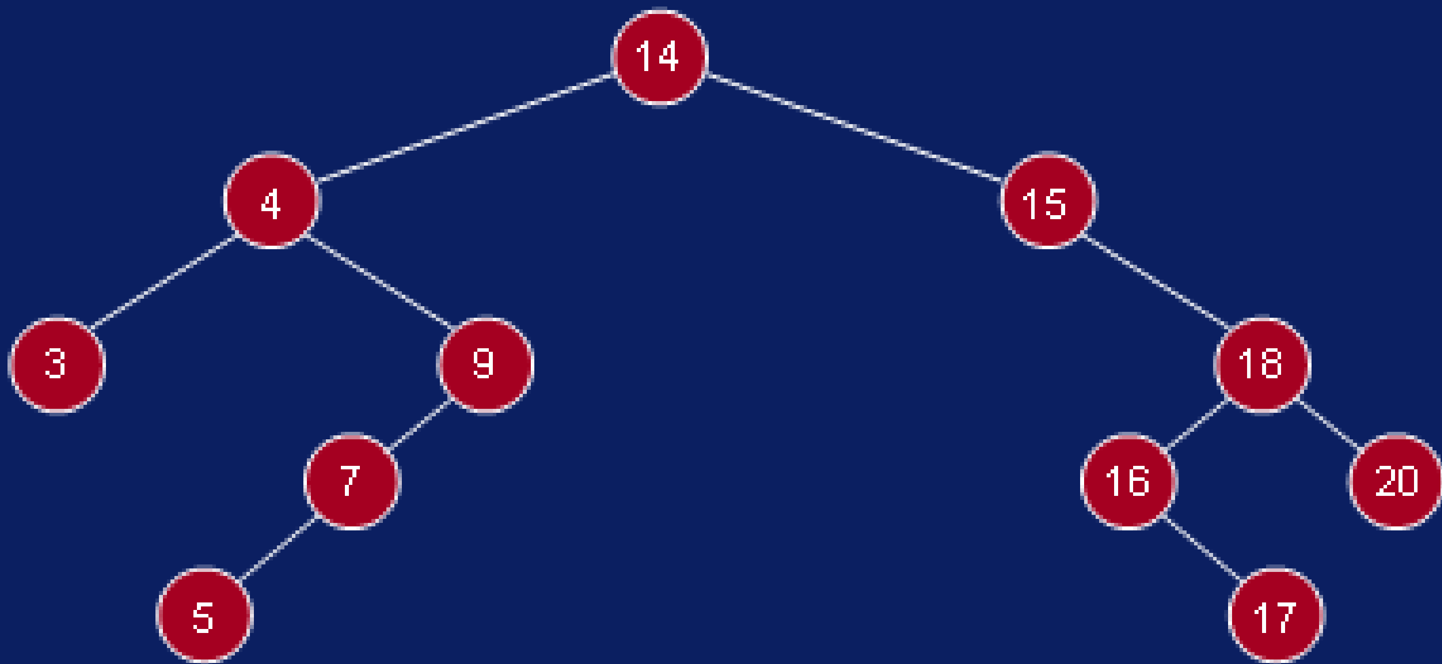


- Duplicate (4) Appeared .....

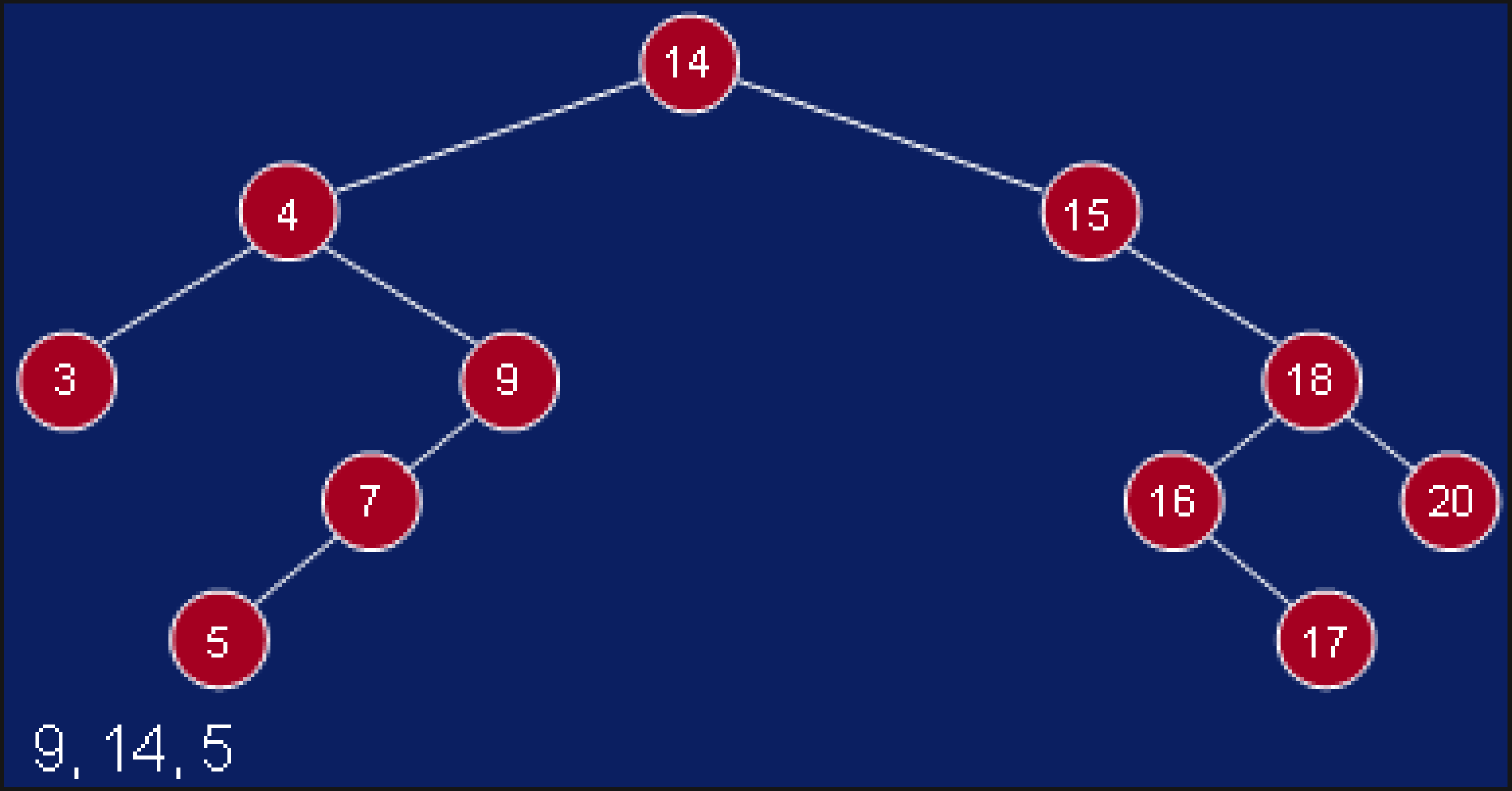


20, 17, 9, 14, 5


14
4
15
3
9
18
7
16
20
5
20, 17, 9, 14, 5



17, 9, 14, 5



- Duplicate (9), (14), (5), Appeared .....

Tree Traversals

1. PreOrder
2. InOrder
3. PostOrder

Preorder

- To traverse non empty binary tree in Preorder, we perform the following operations.

1. Visit the Root
2. Traverse the left sub tree in preorder
3. Traverse the right sub tree in preorder

Inorder

- To traverse a non empty binary tree in Inorder (or Symmetric Order), we perform the following operations.

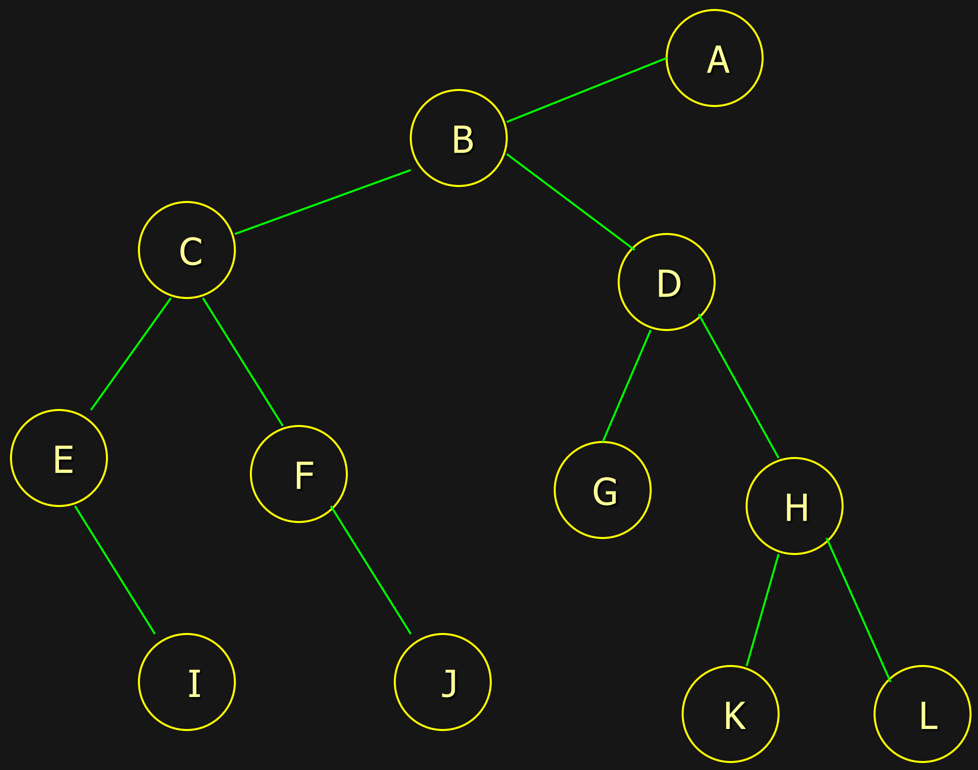
1. Traverse the left sub tree in Inorder
2. Visit the Root
3. Traverse the right sub tree in Inorder

Postorder

- To traverse non empty binary tree in Postorder, we perform the following operations.

1. Traverse the left sub tree in postorder
2. Traverse the right sub tree in postorder
3. Visit the Root

Example of Tree Traversals



- PreOrder : A B C E I F J D G H K L
- InOrder : E I C F J B G D K H L A
- PostOrder : I E J F C G K L H D B A