CS-218
DATA STRUCTURE

Dr. Hashim Yasin

National University of Computer and Emerging Sciences,

Faisalabad, Pakistan.

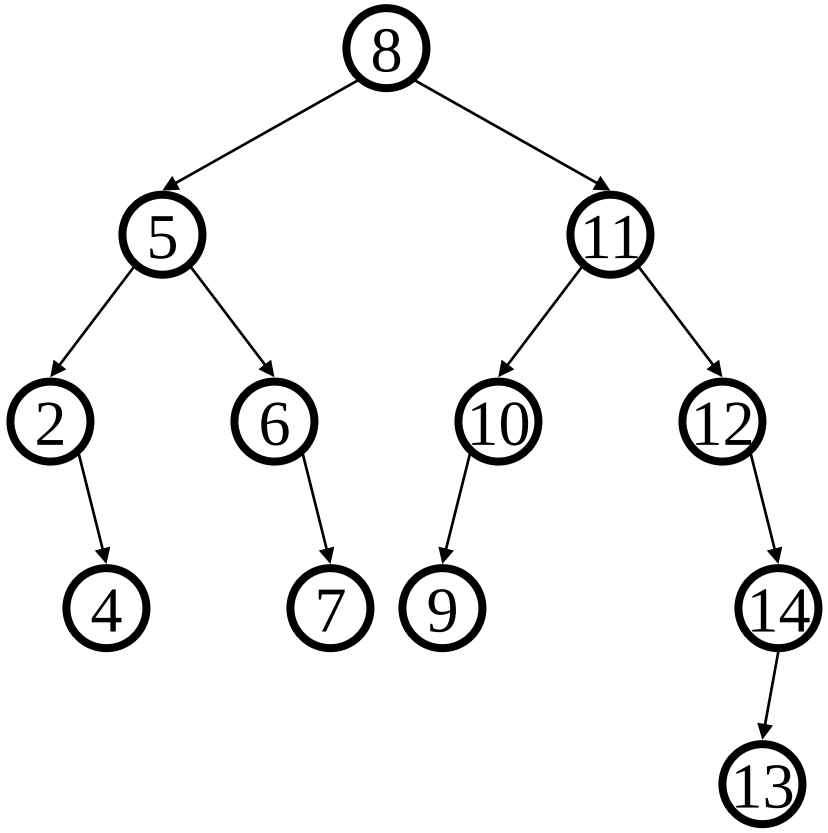
BINARY SEARCH TREE

Binary Search Trees (BSTs)

- **Binary Search Tree**

  Property:

  The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child



Binary Search Trees (BSTs)

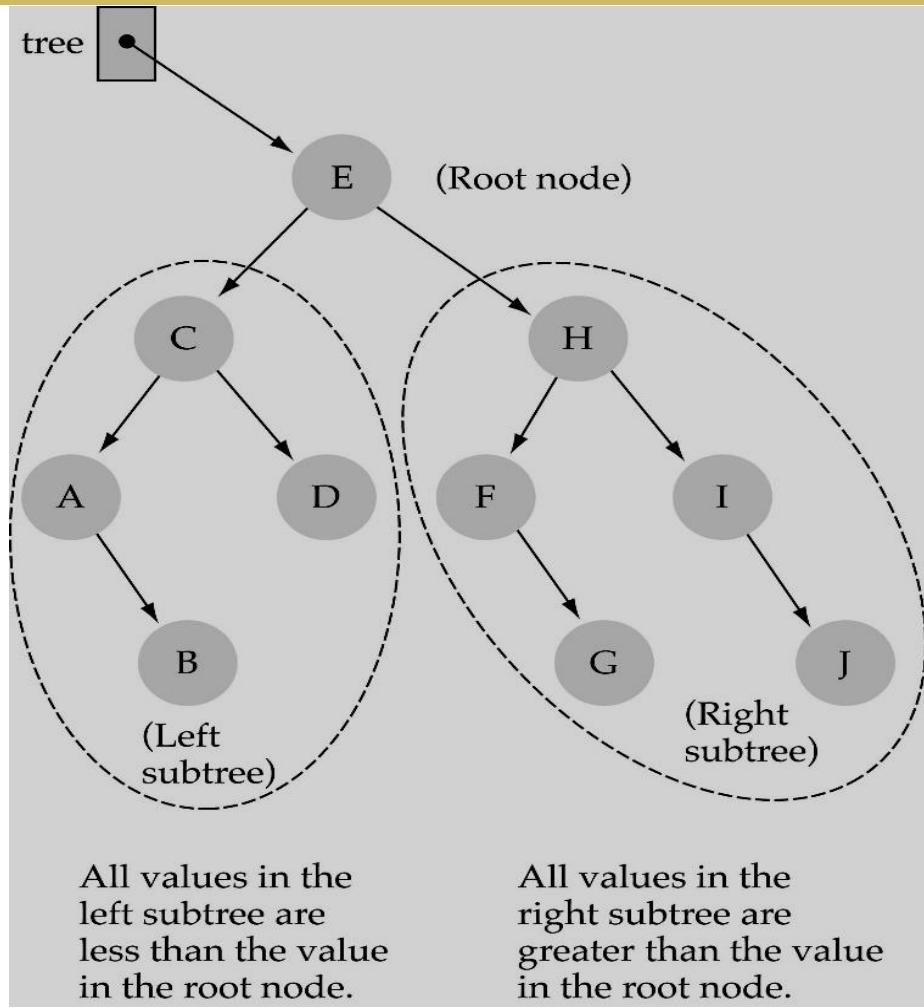
- **Binary Search Tree Property:**

Where is the smallest element?

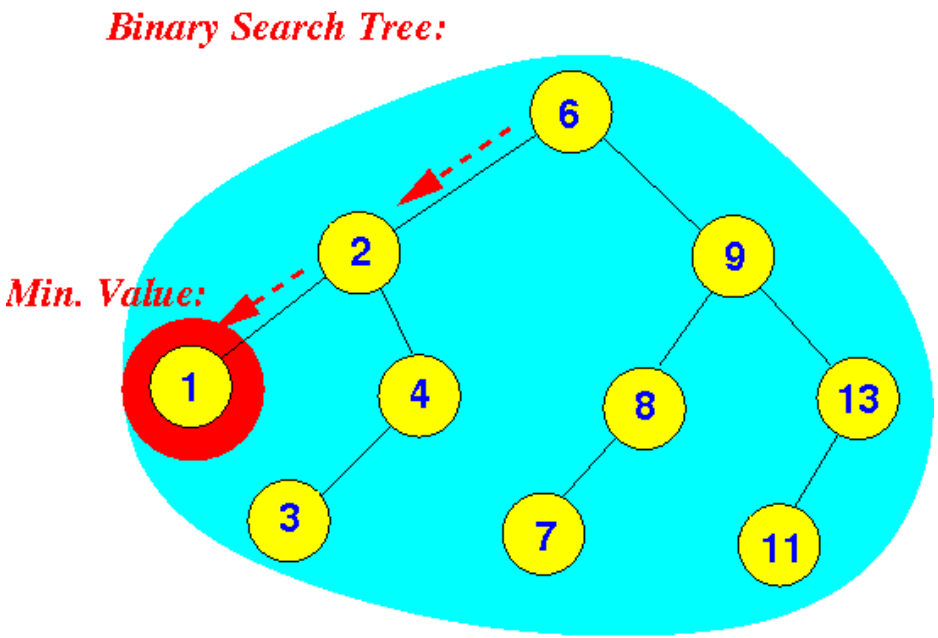
Ans: leftmost element

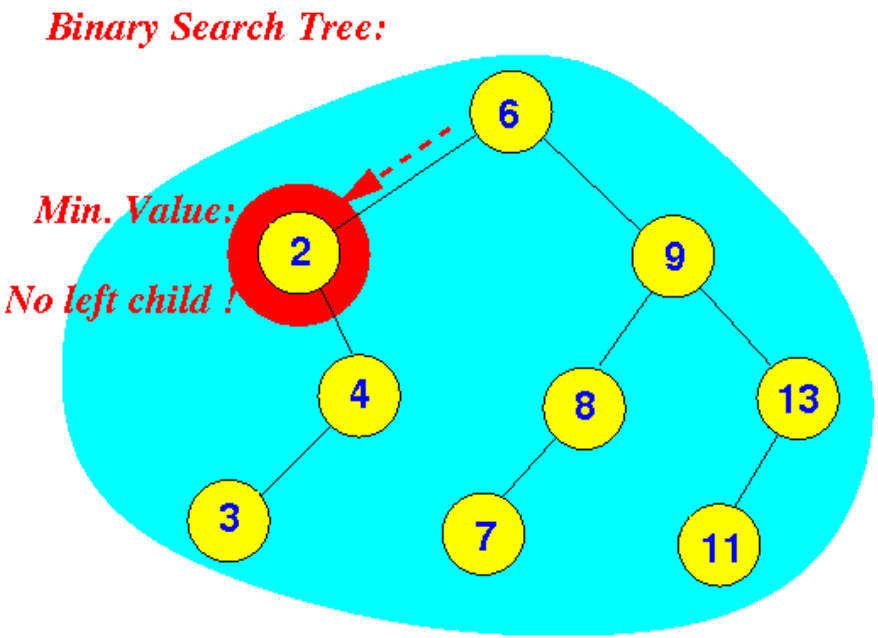
Where is the largest element?

Ans: rightmost element

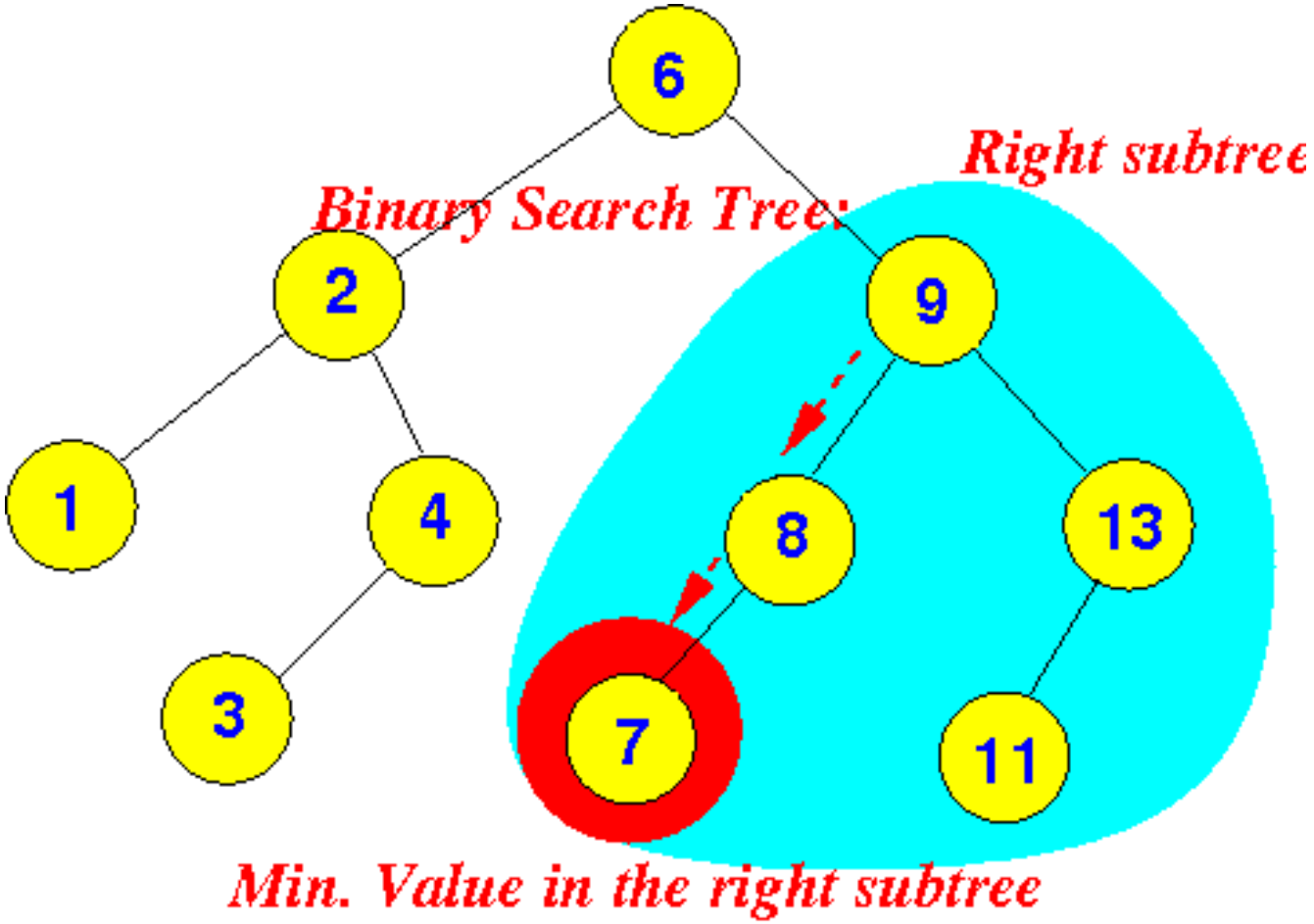


Minimum Value in BST





Min. Value in Right Subtree BST



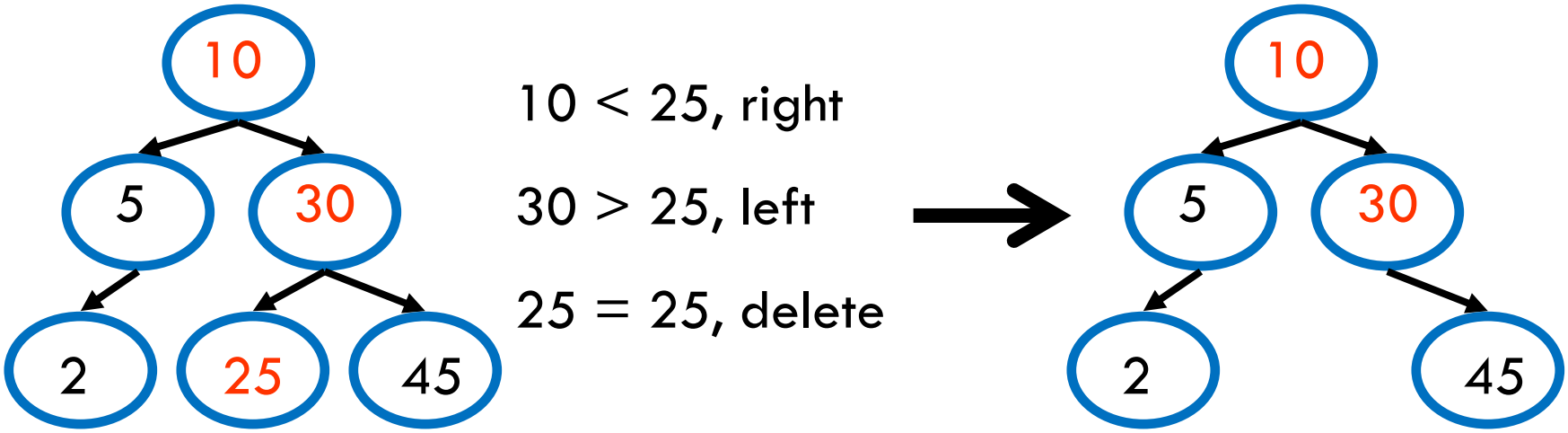
DELETION

Deletion

- A node being erased is not always going to be a leaf node
- There are **three possible scenarios**:
  - The node is a leaf node,
  - It has exactly one child, or
  - It has two children (it is a full node).

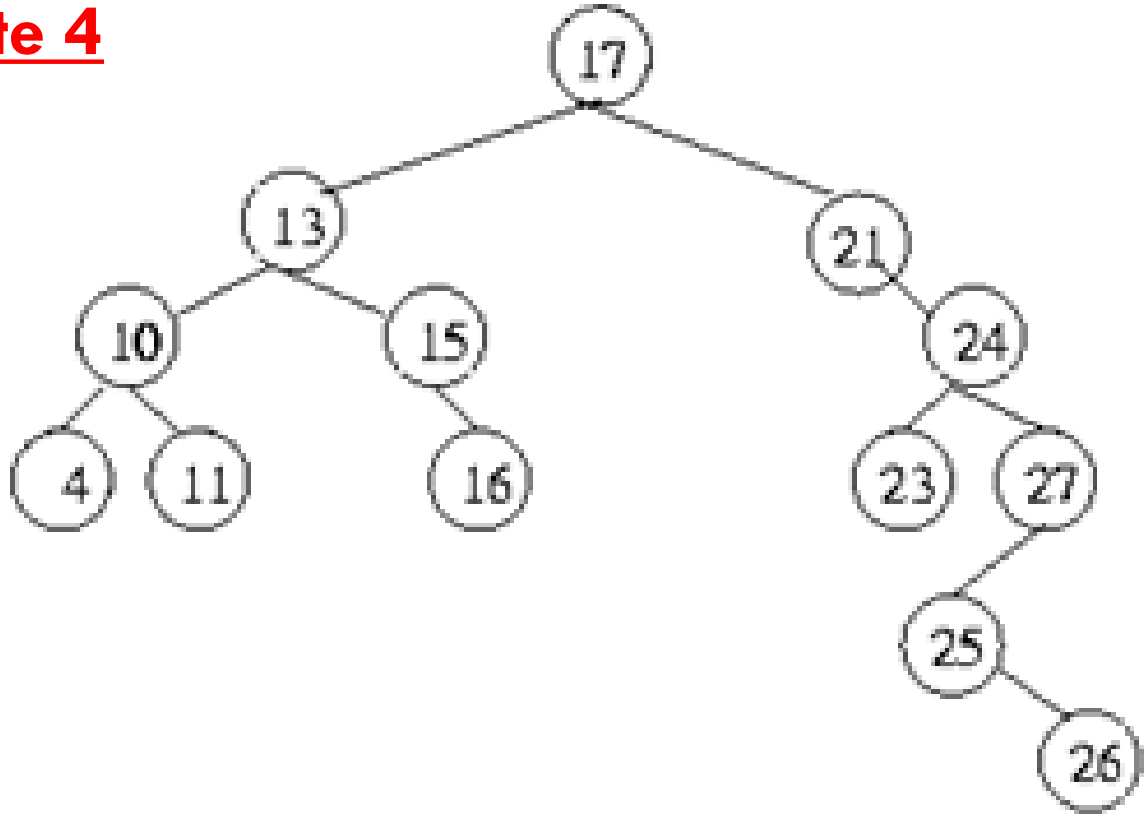
Deletion of a Leaf Node

- **Deleting a leaf node** is quite easy
  - Find its parent
  - Set the child pointer that links it with parent to NULL
  - Free the node's memory
- Consider deleting node containing 25



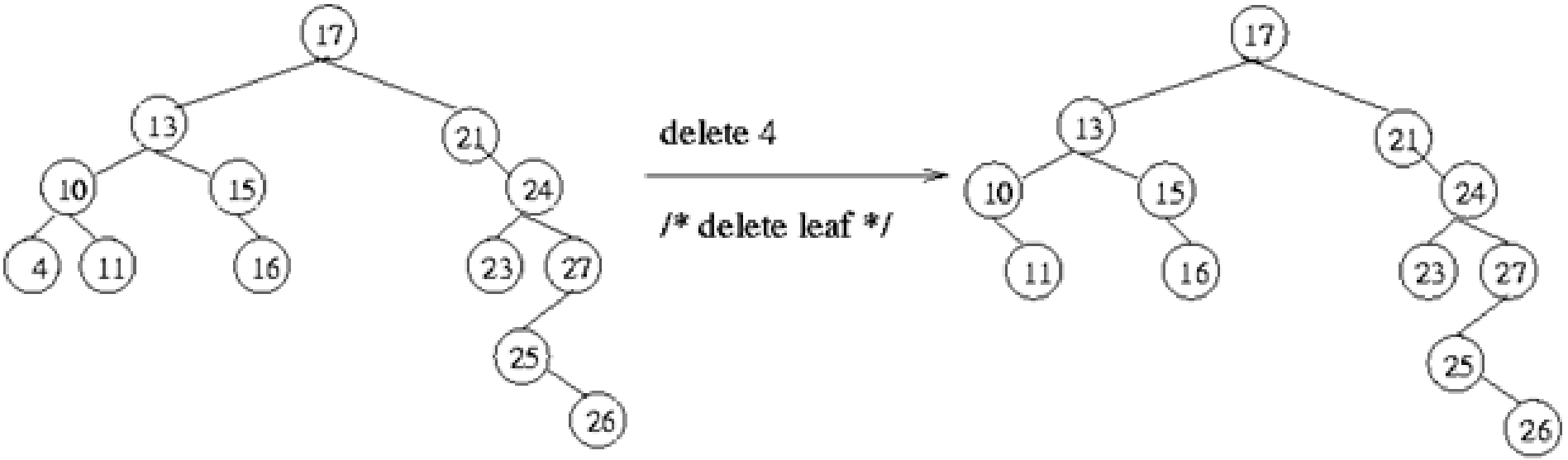
Deletion of a Leaf Node

Delete 4



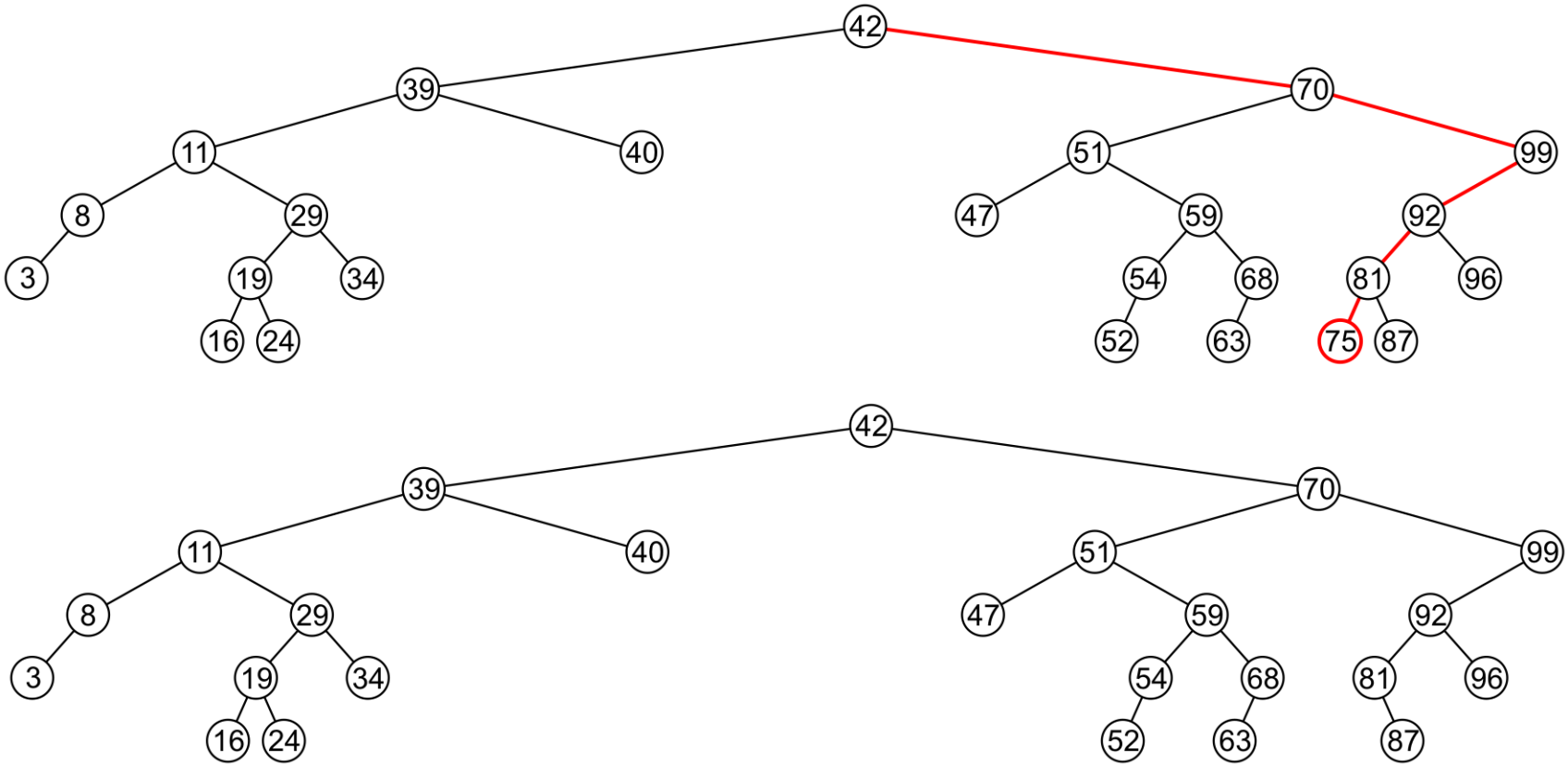
Deletion of a Leaf Node

Delete 4 The node is deleted and left child of 10 is set to NULL



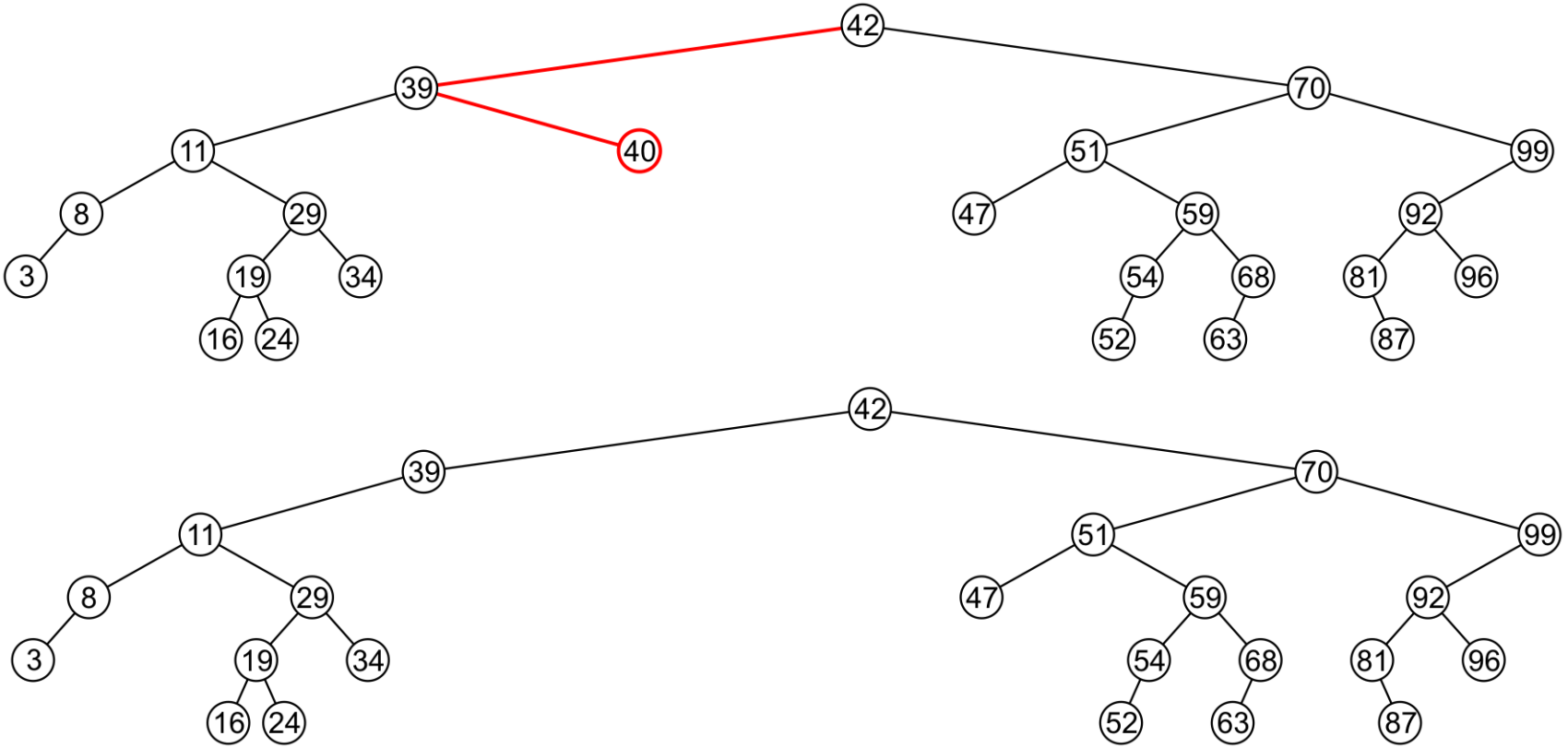
Deletion of a Leaf Node

Delete 75 The node is deleted and left child of 81 is set to NULL



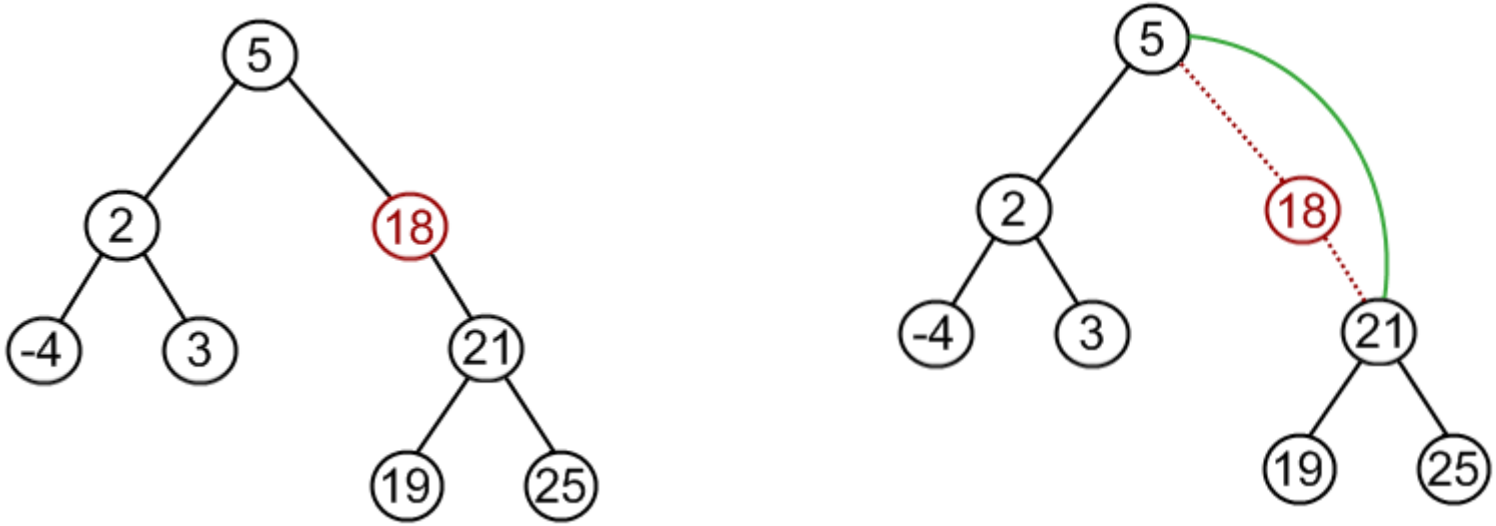
Deletion of a Leaf Node

Delete 40 The node is deleted and right child of 39 is set to NULL



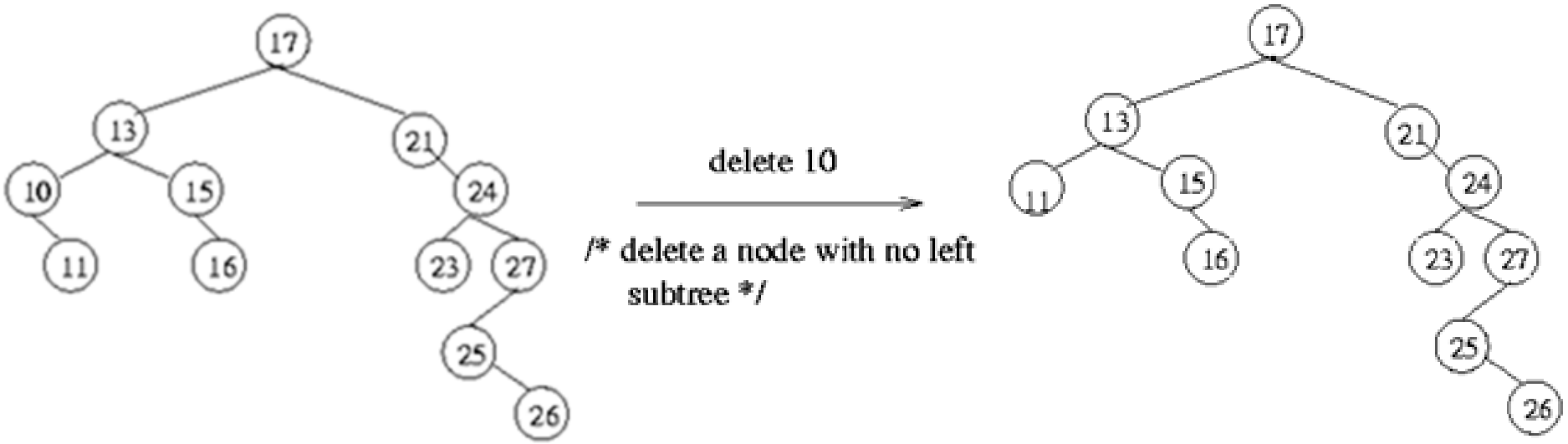
Deletion of a Node with Child

- If a node has only one child (left or right)
  - Simply promote the subtree associated with the child
- Consider deleting 18 which has one right child
  - Node 18 is deleted and the right tree of node 5 is updated to point to 21



Deletion of a Node with Child

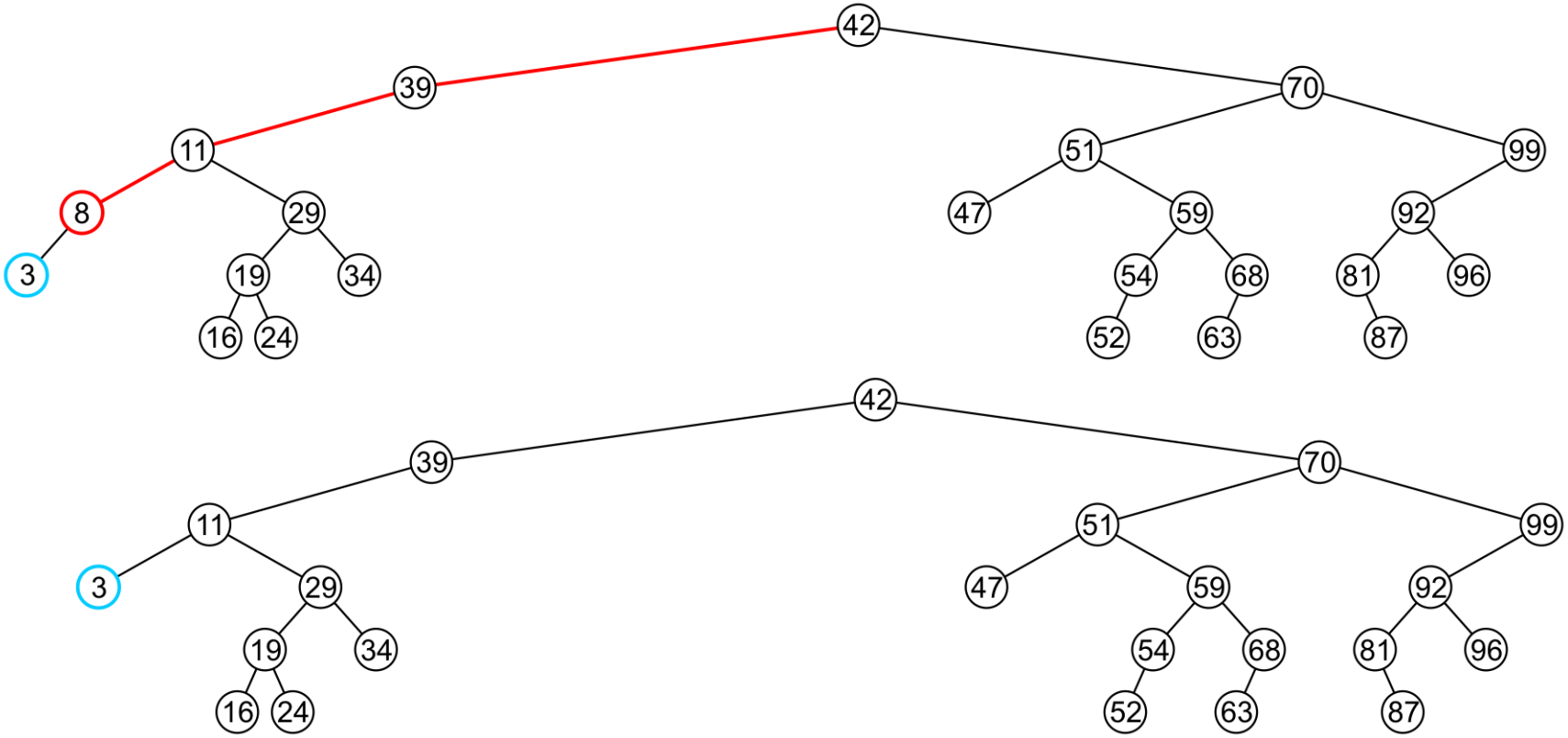
Delete 10 (left subtree)



Deletion of a Node with Child

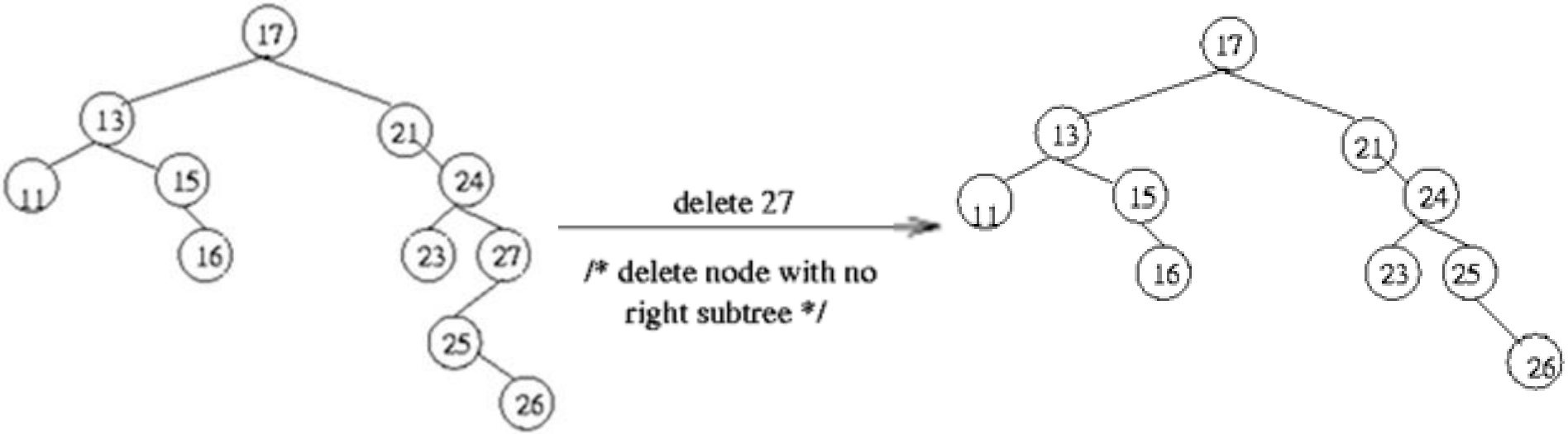
Delete 8 (left subtree)

Node 8 is deleted and the left tree of 11 is updated to point to 3.



Deletion of a Node with Child

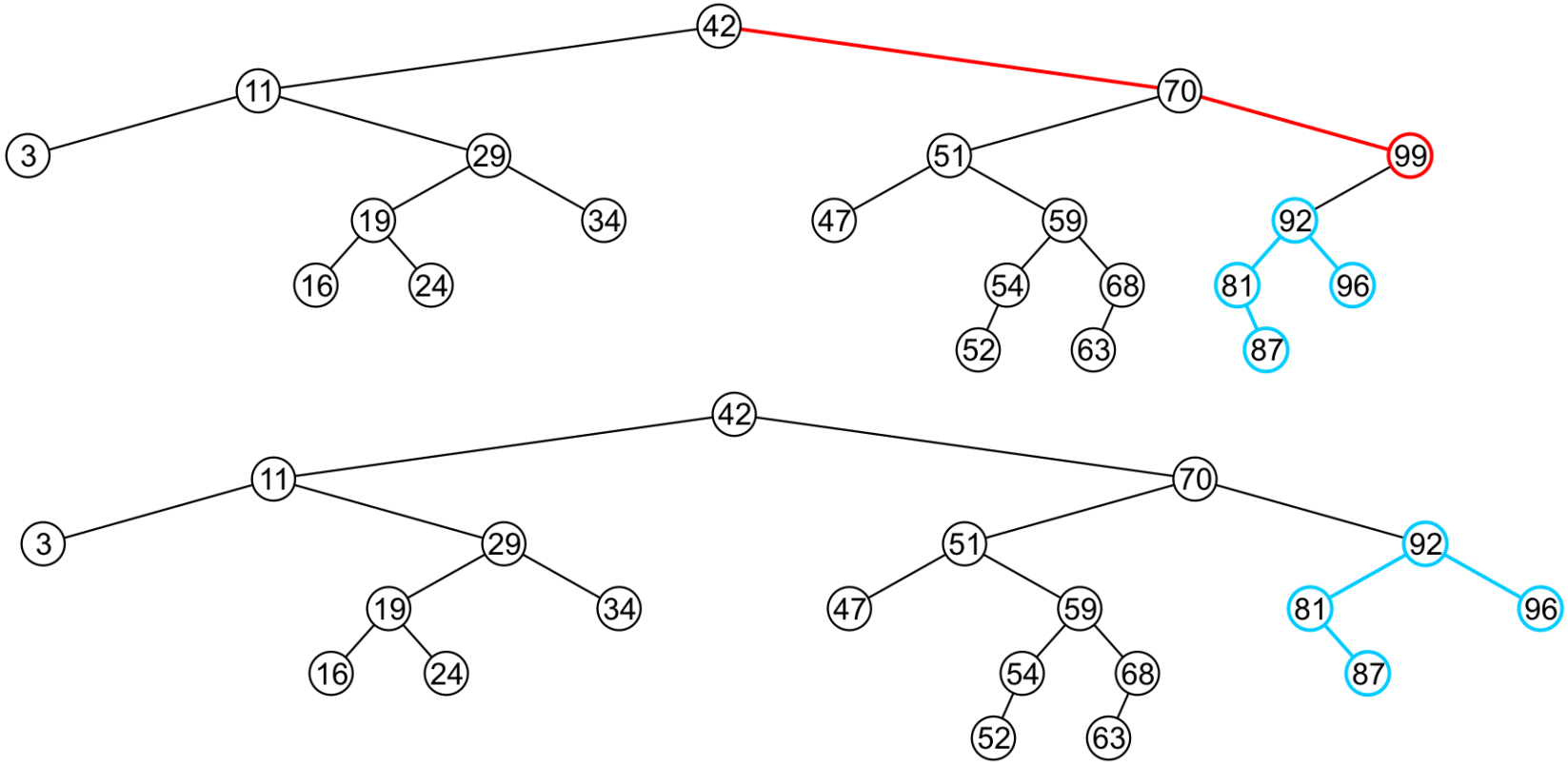
Delete 27 (right subtree)



Deletion … Node with

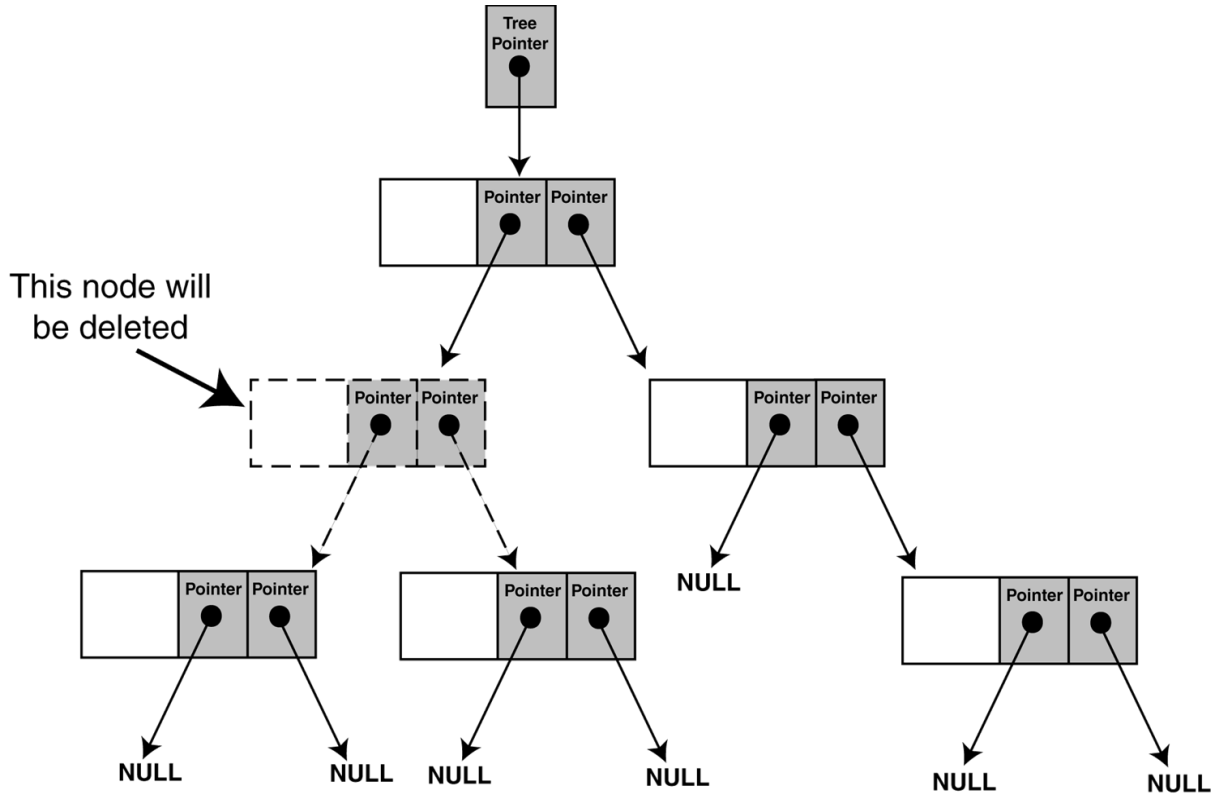
Delete 99 (right subtree)

The right tree of 70 is set to point to node 92



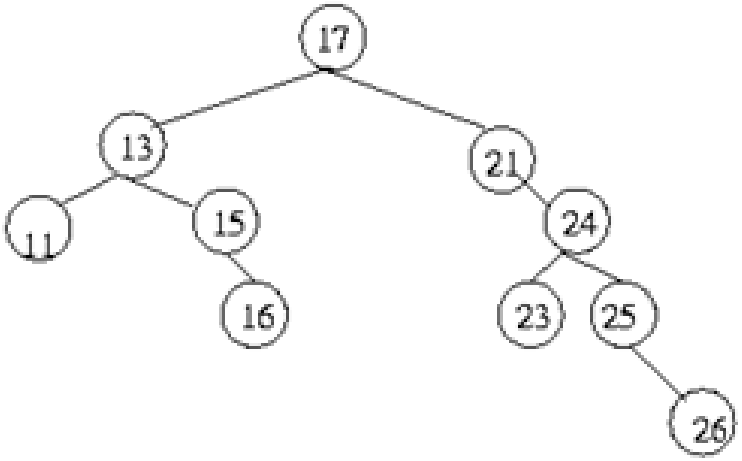
Deletion: Node with two Subtrees

Delete node that has both left and right subtrees:



Deletion: Node with two Subtrees

Delete 13 (both left and right subtrees)



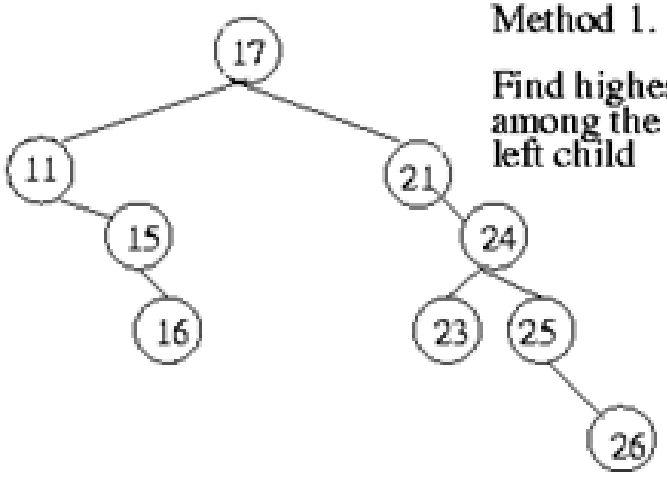
delete 13

/* delete node with both

left and right subtrees */

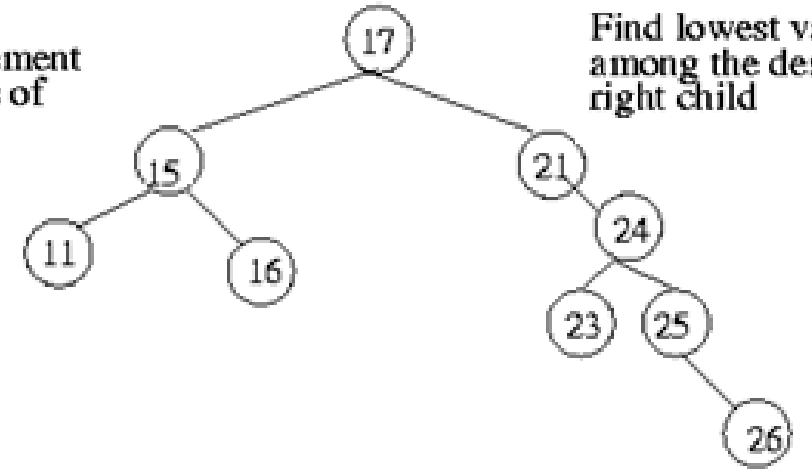
Method 1.

Find highest valued element among the descendants of left child

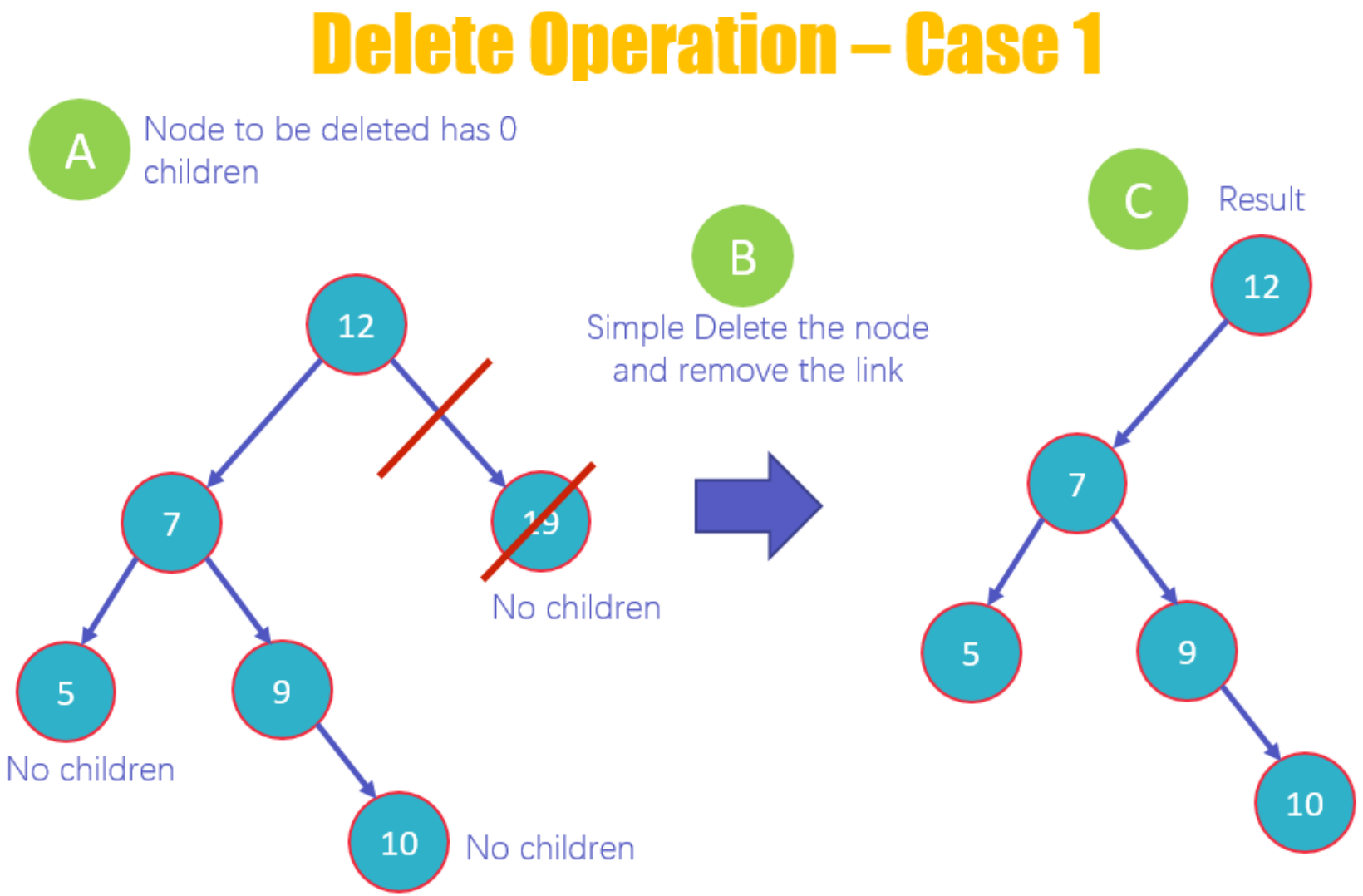


Method 2

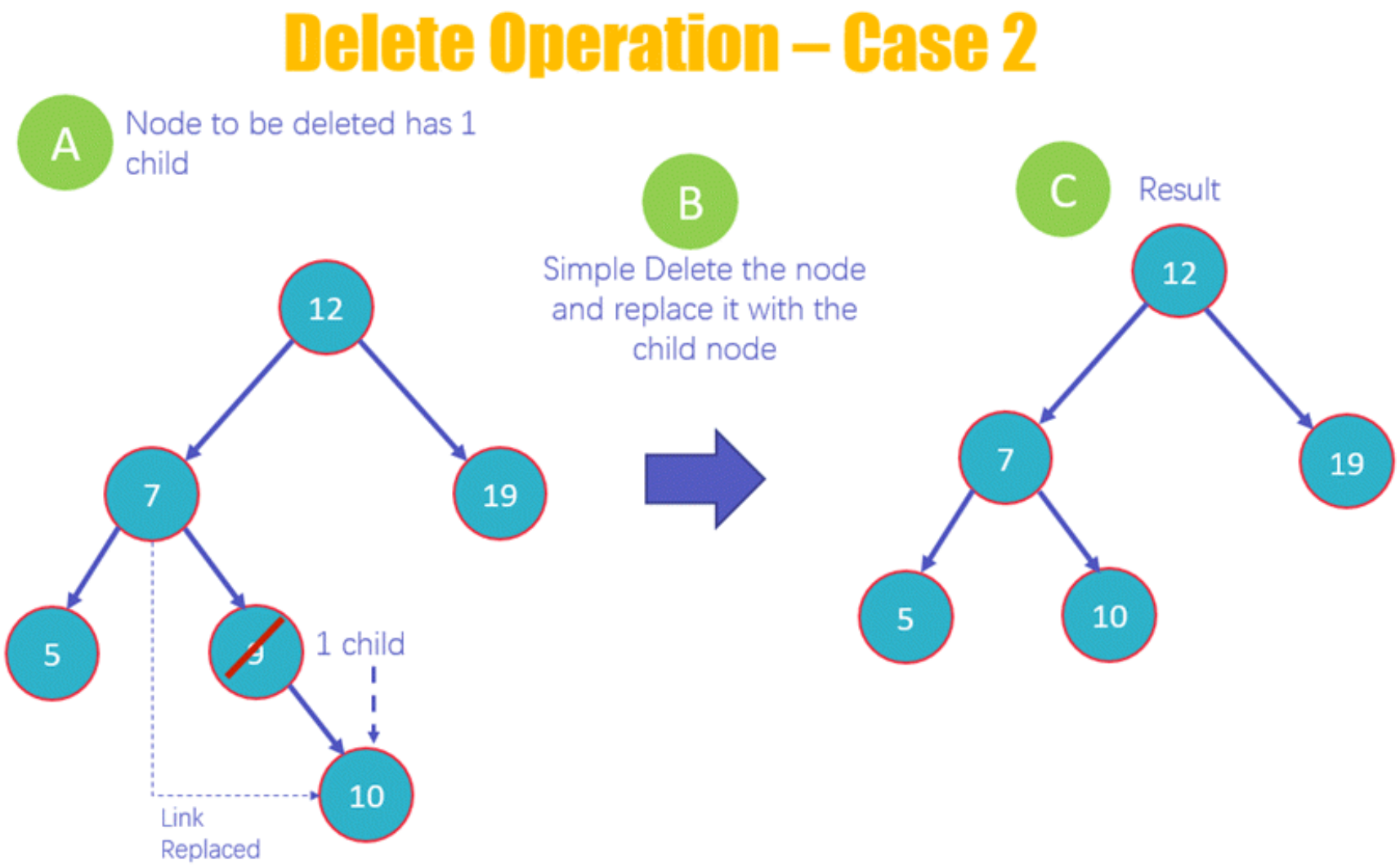
Find lowest valued element among the descendants of right child



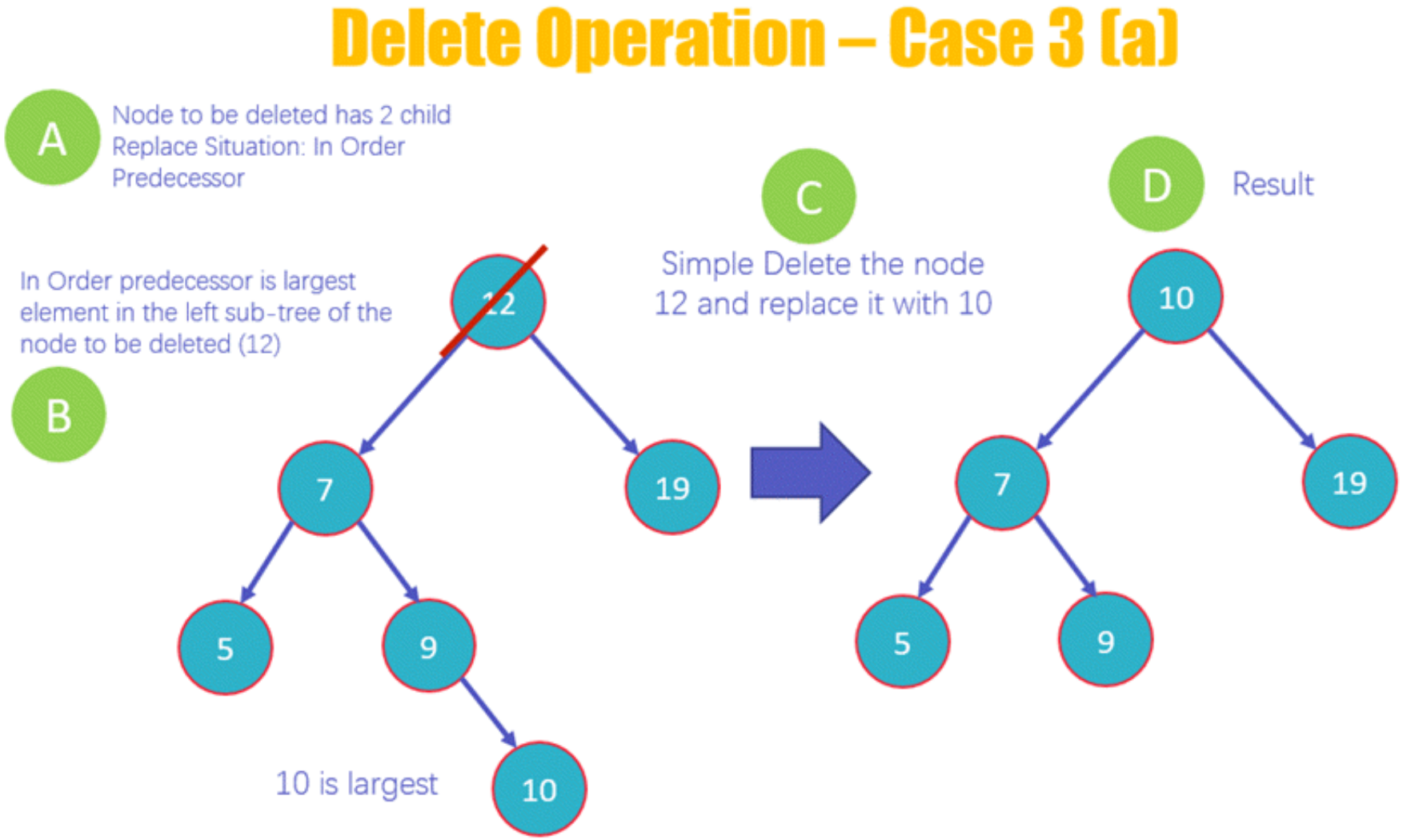
Deletion: Node with two Subtrees



Deletion: Node with two Subtrees



Deletion: Node with two Subtrees



Deletion: Node with two Subtrees

Delete Operation – Case 3 (b)

A

Node to be deleted has 2 child
Replace Situation: In Order Successor

In Order successor is largest element in the right sub-tree of the node to be deleted (12)

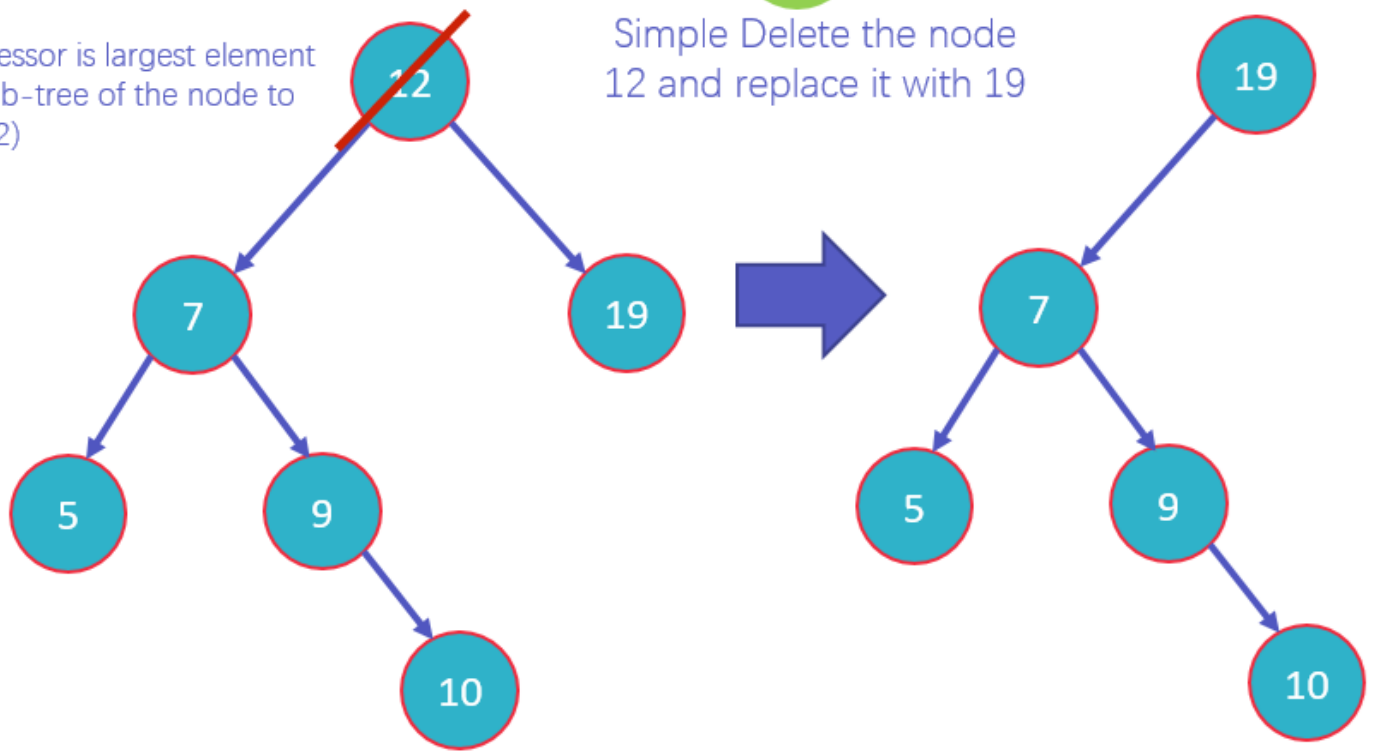
B

C

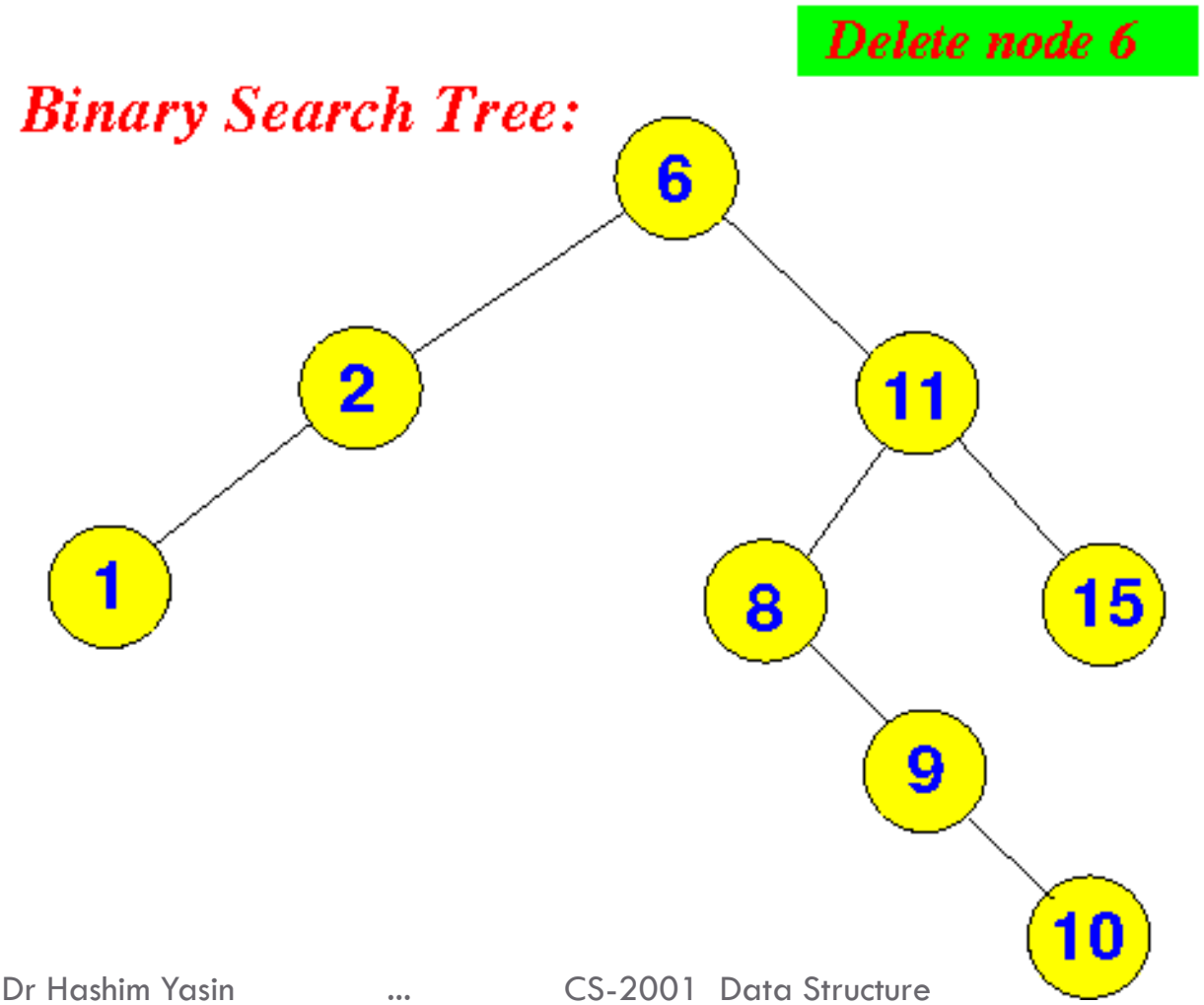
Simple Delete the node 12 and replace it with 19

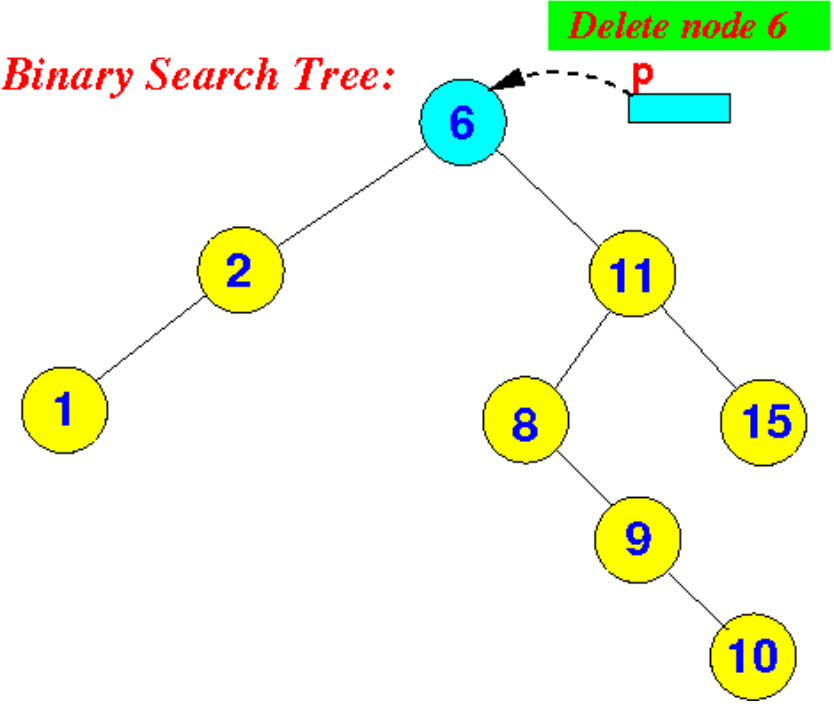
D

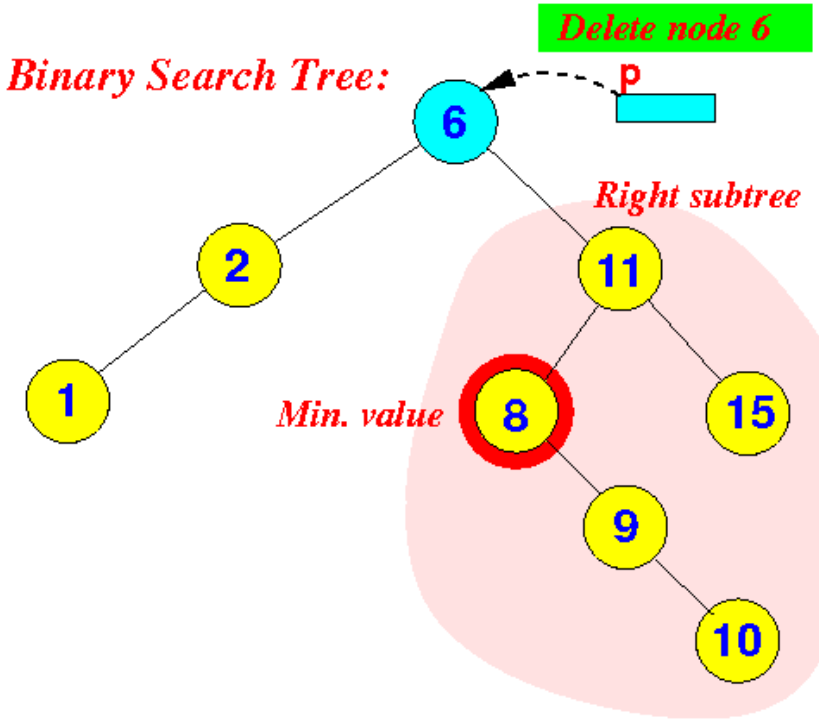
Result

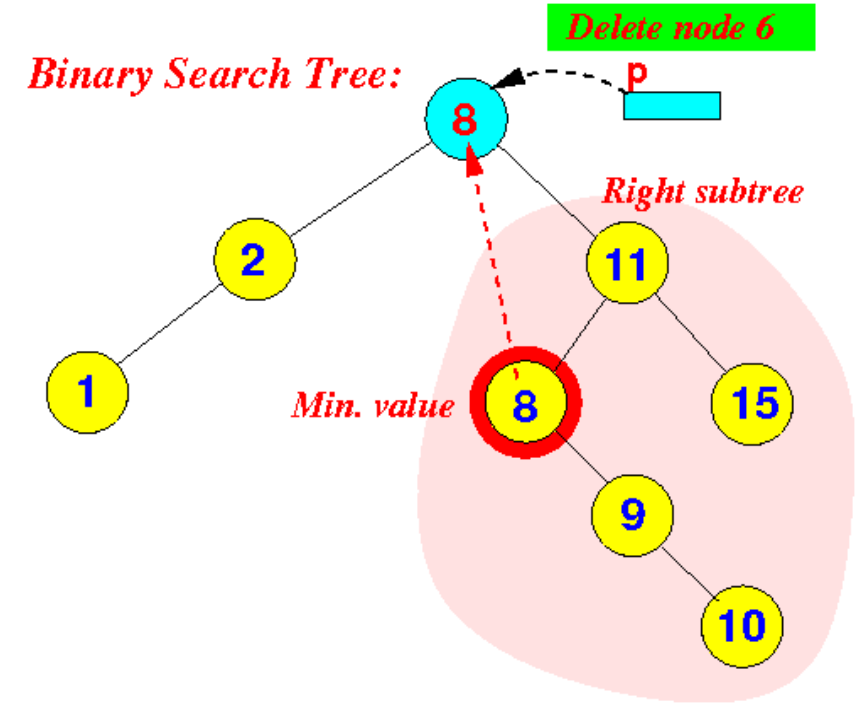


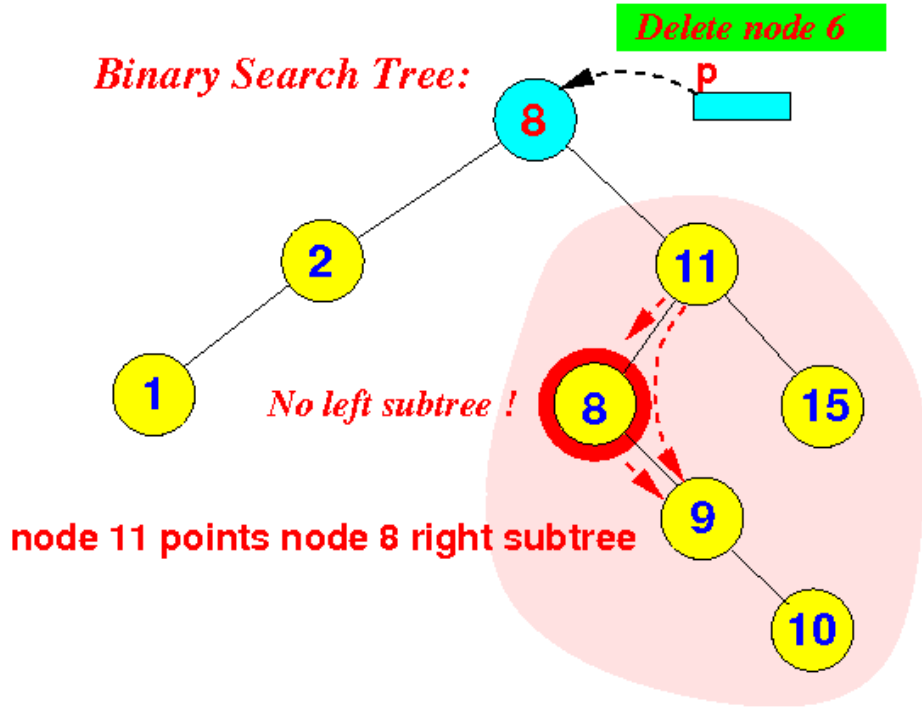
Deletion: Node with two Subtrees




Delete node 6
Binary Search Tree:
p
6
2
11
1
8
15
9
10

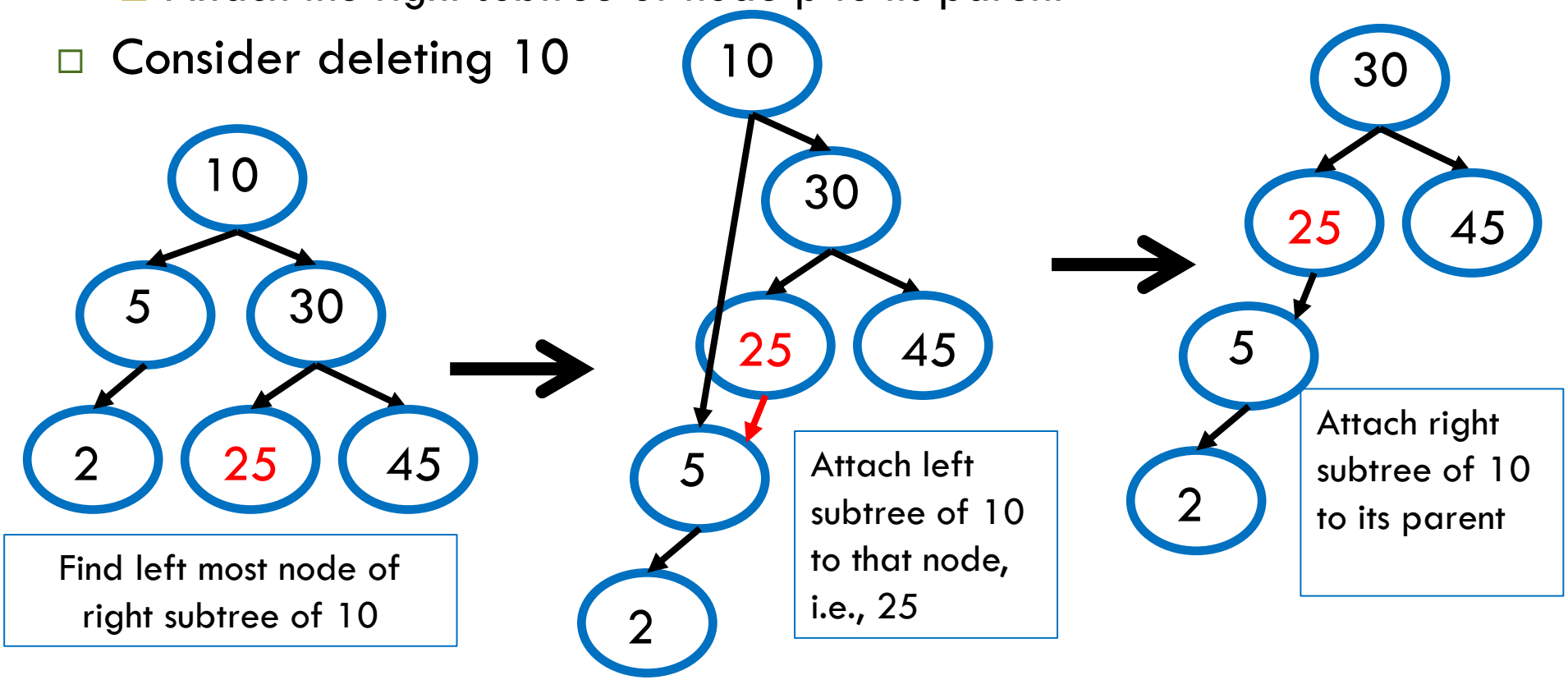

Delete node 6
Binary Search Tree:
p
6
Right subtree
2
11
1
Min. value
8
15
9
10


Delete node 6
Binary Search Tree:
p
8
Right subtree
2
11
1
Min. value
8
15
9
10


Delete node 6
Binary Search Tree:
p
8
2
11
1
No left subtree !
8
15
9
node 11 points node 8 right subtree
10

Deletion: Node with two Subtrees

- Suppose node p with two children has to be deleted
  - Find a position in the right subtree of p to attach its left subtree
    - Left most node in the right subtree of node p (successor of p)
  - Attach the right subtree of node p to its parent
- Consider deleting 10



Implementation

- The basis of our binary tree node is the following `struct` declaration:

```
struct TreeNode
   {
        int value;
        TreeNode *left;
        TreeNode *right;
   };
```

```
class IntBinaryTree{
private:
        struct TreeNode{
                int value;
                TreeNode *left;
                TreeNode *right;
        };

        TreeNode *root;
        void destroySubTree(TreeNode *);
        void deleteNode(int, TreeNode *&);
        void makeDeletion(TreeNode *&);
        void displayInOrder(TreeNode *);
        void displayPreOrder(TreeNode *);
        void displayPostOrder(TreeNode *);

 public:

        IntBinaryTree() { root = NULL; }              // Constructor
        ~IntBinaryTree() { destroySubTree(root); }// Destructor
        void insertNode(int);
        bool searchNode(int);
        void remove(int);
        void showNodesInOrder(){    displayInOrder(root); }
        void showNodesPreOrder(){   displayPreOrder(root); }
        void showNodesPostOrder(){  displayPostOrder(root); }
};
```

Implementation


```
void IntBinaryTree::deleteNode(int num, TreeNode *&nodePtr) {
    if (nodePtr == NULL) // node does not exist in the tree
        cout << num <<“ not found.\n";
    else if (num < nodePtr->value)
        deleteNode(num, nodePtr->left);  // find in left subtree
    else if (num > nodePtr->value)
        deleteNode(num, nodePtr->right); // find in right subtree
    else // num == nodePtr->value i.e., node is found
        makeDeletion(nodePtr); // actually deletes node from BST
}
```

Note:

- The declaration of the `nodePtr` parameter: `TreeNode *&nodePtr;`
- `nodePtr` is a reference to a pointer to a `TreeNode` structure
  - Any action performed on `nodePtr` is actually performed on the argument passed into `nodePtr`.

```
void IntBinaryTree::makeDeletion(TreeNode *&nodePtr) {
   TreeNode *tempNodePtr; // Temperary pointer

   if (nodePtr->right == NULL) { // case for leaf and one (left) child
      tempNodePtr = nodePtr;
      nodePtr = nodePtr->left; // Reattach the left child
      delete tempNodePtr;
   }

   else if (nodePtr->left == NULL) { // case for one (right) child
      tempNodePtr = nodePtr;
      nodePtr = nodePtr->right; // Reattach the right child
      delete tempNodePtr;
   }

   else {  // case for two children.
      tempNodePtr = nodePtr->right; // Move one node to the right
      while (tempNodePtr->left) { // Go to the extreme left node
         tempNodePtr = tempNodePtr->left;

      }

      tempNodePtr->left = nodePtr->left; // Reattach the left subtree
      tempNodePtr = nodePtr; //save nodePtr to delete later
      nodePtr = nodePtr->right; // Reattach the right subtree
      delete tempNodePtr;

   }
}
```

```
// This program builds a binary tree with 5 nodes.
// The DeleteNode function is used to remove two of them.
#include <iostream.h>
#include "IntBinaryTree.h“

void main(void) {
   IntBinaryTree tree;

   cout << "Inserting nodes.\n";
   tree.insertNode(5);
   tree.insertNode(8);
   tree.insertNode(3);
   tree.insertNode(12);
   tree.insertNode(9);

   cout << "Here are the values in the tree:\n";
   tree.showNodesInOrder();

   cout << "Deleting 8...\n";
   tree.remove(8);
   cout << "Deleting 12...\n";
   tree.remove(12);

   cout << "Now, here are the nodes:\n";
   tree.showNodesInOrder();
}
```

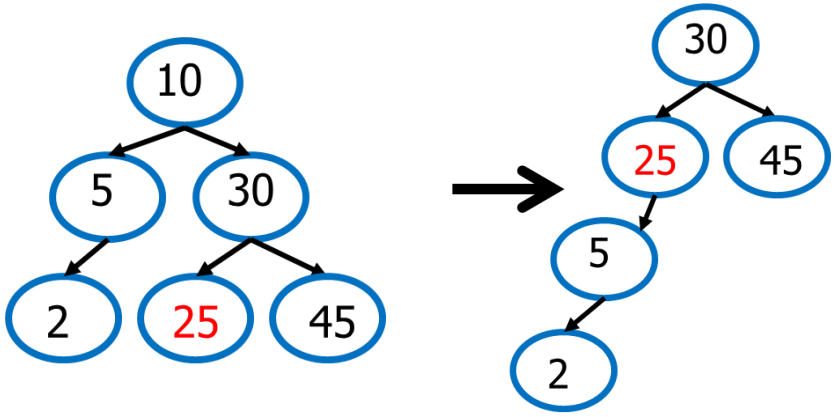
```
// This program builds a binary tree with 5 nodes.
// The DeleteNode function is used to remove two of them.
#include <iostream.h>
#include "IntBinaryTree.h“

void main(void) {
   IntBinaryTree tree;

   cout << "Inserting nodes.\n";
   tree.insertNode(5);
   tree.insertNode(8);
   tree.insertNode(3);
   tree.insertNode(12);
   tree.insertNode(9);

   cout << "Here are the values in the tree:\n";
   tree.showNodesInOrder();

   cout << "Deleting 8...\n";
   tree.remove(8);
   cout << "Deleting 12...\n";
   tree.remove(12);

   cout << "Now, here are the nodes:\n";
   tree.showNodesInOrder();
}
```

Program Output:

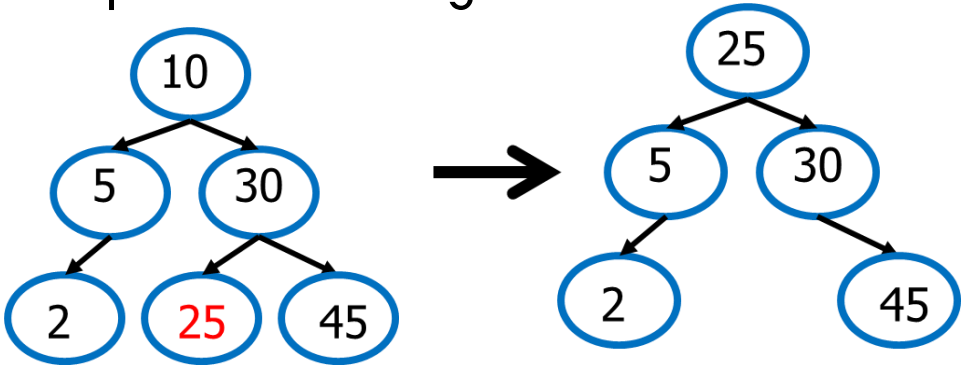
```
Inserting nodes.
Here are the values in the tree:
3
5
8
9
12
Deleting 8...
Deleting 12...
Now, here are the nodes:
3
5
9
```

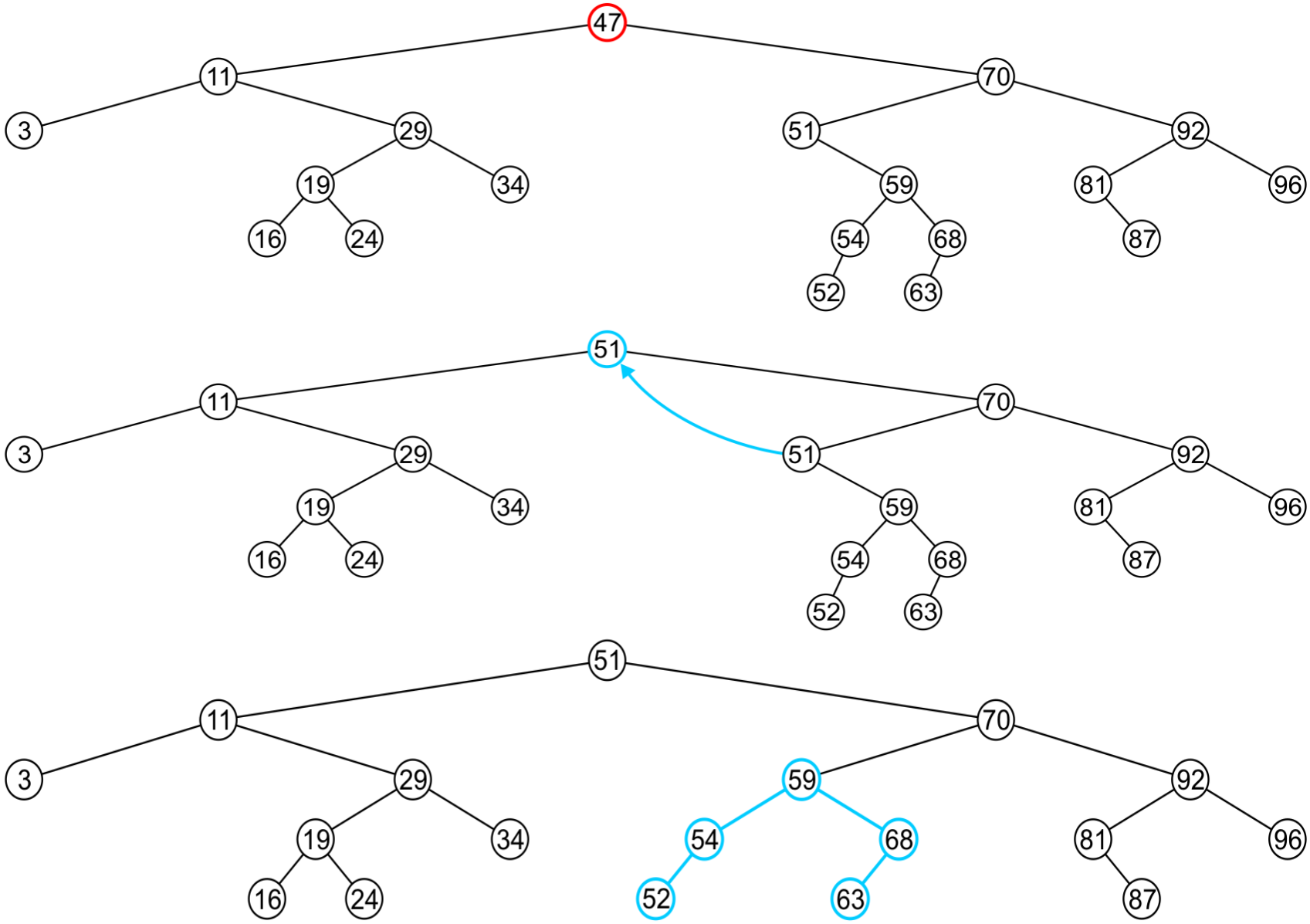
Deletion: Node with two Subtrees

- **Problem:** Height of the BST increases



- A better Solution to delete node p with two children
  - Replace node p with the minimum object in the right subtree
  - Delete that object from the right subtree

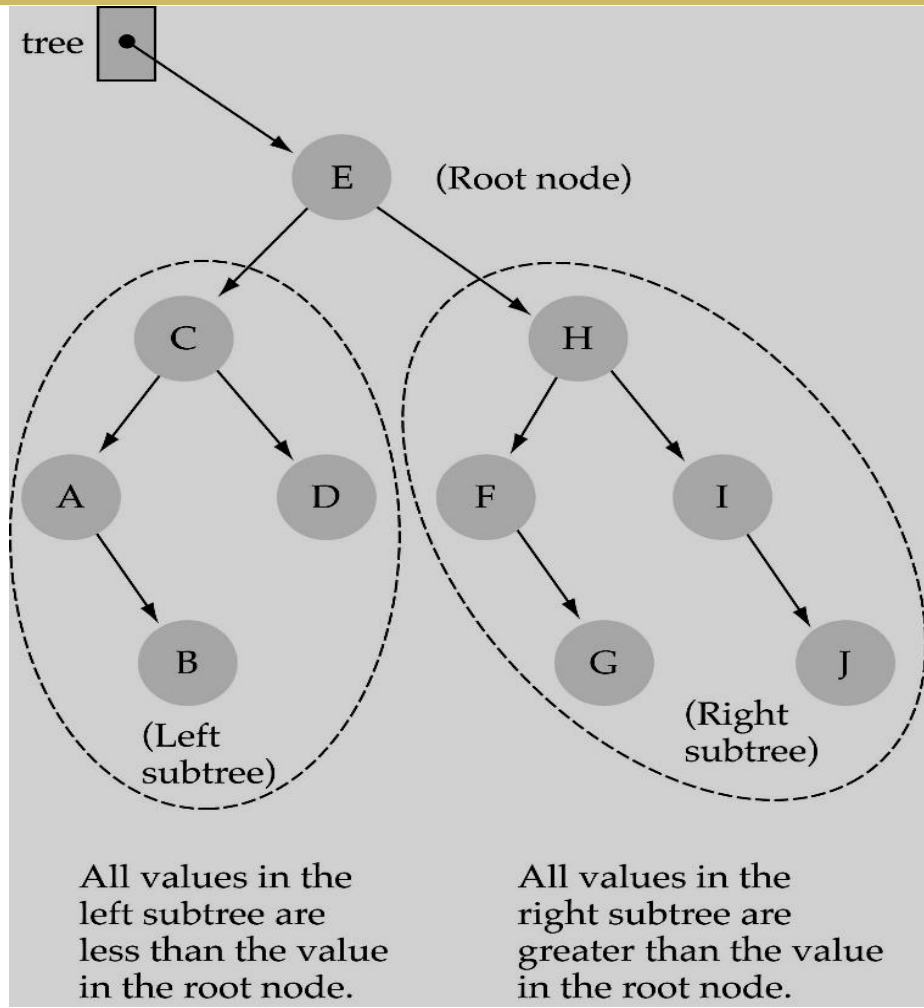



47
11
70
3
29
51
92
19
34
59
81
96
16
24
54
68
87
52
63
51
11
70
3
29
51
92
19
34
59
81
96
16
24
54
68
87
52
63
51
11
70
3
29
59
92
19
34
54
68
81
96
16
24
52
63
87

Searching in BST

Is this better than searching a linked list?

Yes !! ---> O(logN)



Why BST

Array

- Searching in the Array O(n)
- Insertion O(1)
- Remove O(n)

Linked List

- Searching in the Linked List O(n)
- Insertion O(1)
- Remove O(n)

BST

- Searching in the BST O(log n)
- Insertion O(log n)
- Remove O(log n)

Reading Materials

- Schaum's Outlines: Chapter # 7
- D. S. Malik: Chapter # 11
- Nell Dale: Chapter # 8
- Allen Weiss: Chapter # 4
- Tenebaum: Chapter # 5