Chapter Six

Tree

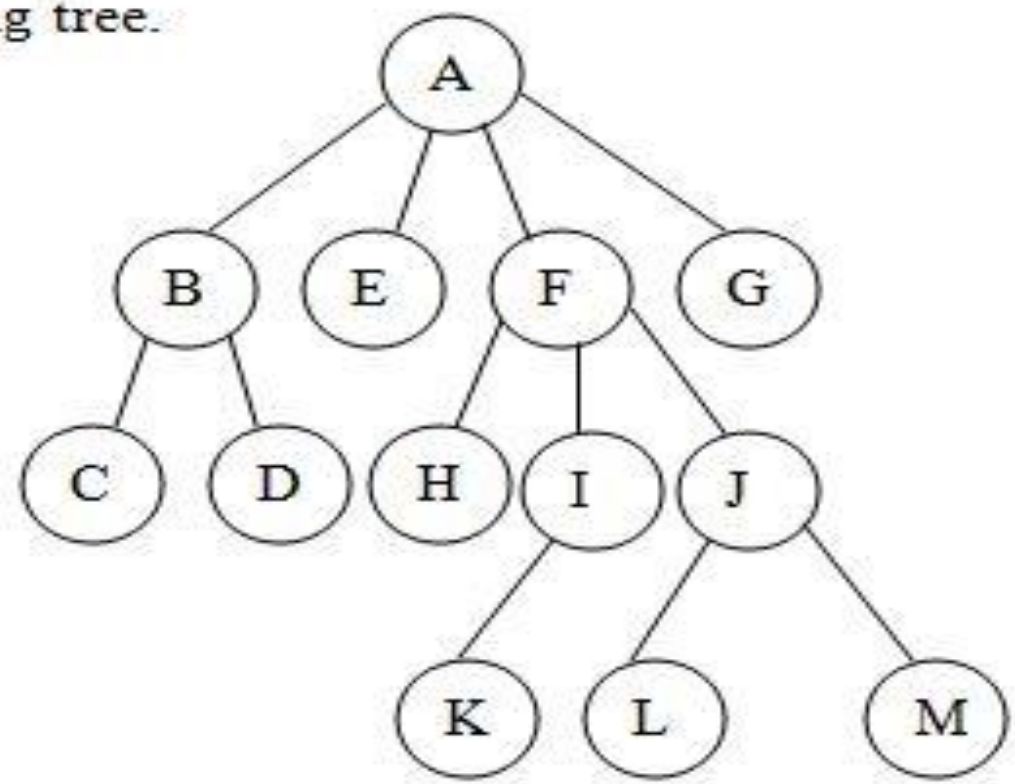
Compiled by: Nigusu Y.(nigusu02@gmail.com

Introduction

- In the case linked list, stack, Queue it is impossible to show **hierarchical** structure on a collection of data items
- Tree is a set of nodes and edges that connect pairs of nodes
- Its an abstract model of a **hierarchical** structure
- Rooted tree has the following structure:
  - One node distinguished as **root**.
  - Every node C except the root is connected from exactly other node P
    - P is C's parent, and C is one of P's children.
  - There is a unique path from the root to each child node.
  - The number of edges in a path is the length of the path.

Tree Terminologies

Consider the following tree.



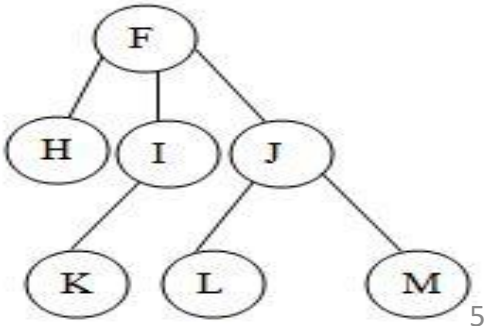
- **Root**
  - A node with out a parent. → A
- **Siblings**
  - Nodes with the same parent. B→C,D

Tree Terminologies…

- **Internal node**
  - a node with at least one child →A, B, F, I, J
- **External (leaf) node**
  - a node without a child. → C, D, E, H, K, L, M, G
- **Ancestors of a node**
  - parent, grandparent, grand-grandparent… of a node.
  - Ancestors of K→ A, F, I
- **Descendants of a node**
  - children, grandchildren, grand-grandchildren… of a node.
  - Descendants of F→ H, I, J, K, L, M

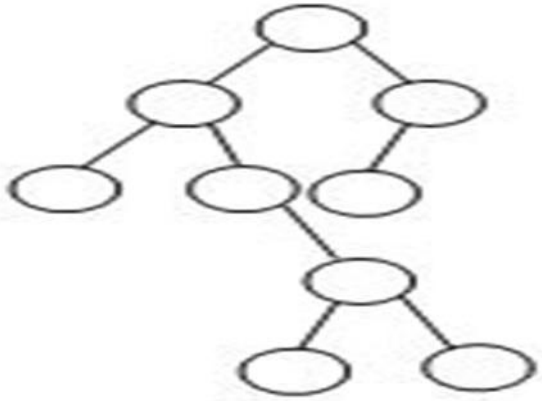
Tree Terminologies…

- **The level of a node**
  - Its the length of the path from the root to that node **plus** one.
  - If root node is at level 0, then its next child node is at **level** 1 and so on
- **Depth of a node**
  - Number of ancestors/length of the path from the root to the node. Depth of H→ 2
- **Height of a tree**
  - Depth of the deepest node. → 3
- **Subtree**
  - ✓ A tree consisting of a node and its descendants



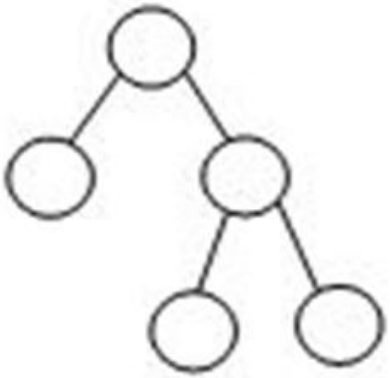
Binary tree

- **Binary Tree** is a special data structure used for data storage purposes.
- A binary tree has a special condition that each node can have **two children** at maximum.
- Each child is designated as either a left or right child
- A binary tree have benefits of both an ordered array and a linked list as search is as quick as in sorted array and insertion or deletion operation are as fast as in linked list



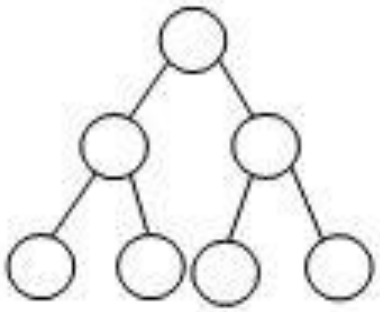
Variations of Binary Tree

- **Full binary tree**
  - A binary tree in which every non-leaf node has exactly two children If all leaves of full binary tree are on the same level, this binary tree is said to be **complete.**
  - A binary tree where each node has either 0 or 2 children

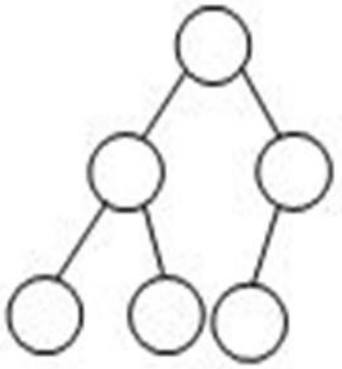


Variations of Binary Tree…

- **Balanced binary tree**
  - A binary tree where each node except the leaf nodes has left and right children's
  - All the leaves are at the same level



- **Complete binary tree**:
  - A binary tree in which the length from the root to any leaf node is either h or h-1 where h is the height of the tree
  - The deepest level should also be filled **from left to right**

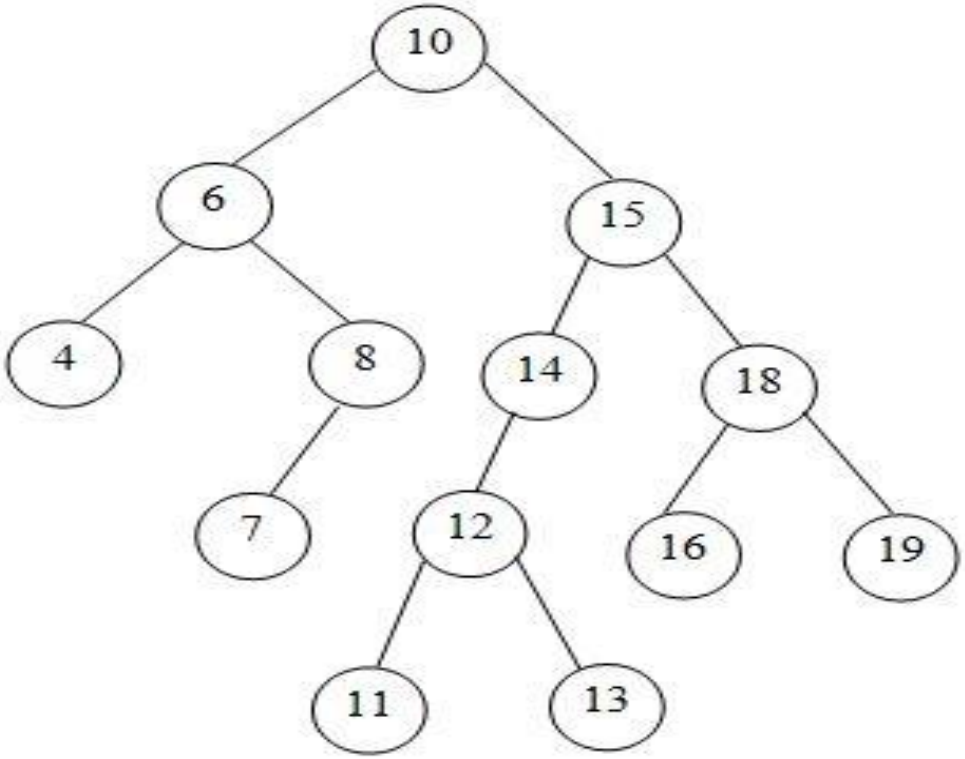


Binary search tree

- Also called **ordered binary tree**
- Its a binary tree that may be empty, but if it is not empty it satisfies the following.
  - Every node has a key and no two elements have the same key.
  - The keys in the right subtree are larger than the keys in the root ***i.e.*** ***R > P***
  - The keys in the left subtree are **smaller** than the keys in the root ***i.e.*** ***L < P***
  - The left and the right subtrees are also binary search trees.

Example of Binary Search tree

- **Keys -** Key represents a value of a node based on which a search operation is to be carried out for a node.



Binary Search Tree …

- Binary search tree is a structure for holding a set of ordered data elements in such a way that it is easy to find any specified element and easy to insert and delete elements

```
struct Node
 {
     //Declaration of data fields
     Node * Left;
     Node *Right;
 };
 Node *headPtr = NULL;
```

Operations on Binary Search Tree

- **Traversing**
  - 3-types
    - Pre-order traversal
    - In-order traversal
    - Post-order traversal
- Application of binary tree traversal
    - Searching
    - Inserting
    - Deleting
      - ✓Deleting by merging
      - ✓Deleting by copying

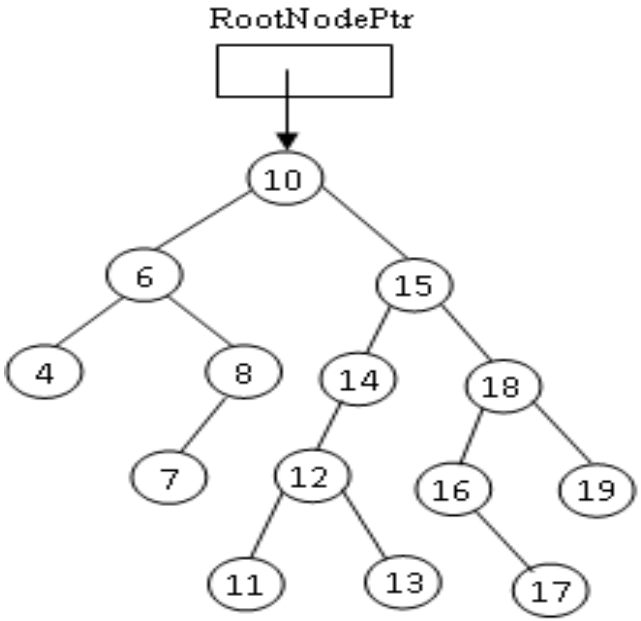
Traversing Binary tree search

- **Three ways**
  - **Pre-order traversal** : **PLR**
    - Traversing binary tree in the order of ***parent***, left and right.
  - **In-order traversal** : **LPR**
    - Traversing binary tree in the order of left, ***parent*** and right.
  - **Post-order traversal** : **LRP**
    - Traversing binary tree in the order of *left*, *right* and ***parent***
- **Example**
  - **Pre-order**

    10, 6, 4, 8, 7, 15, 14, 12, 11, 13, 18, 16, 17, 19
  - **In-order :** displays nodes in ascending order

    4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
  - **Post-order**

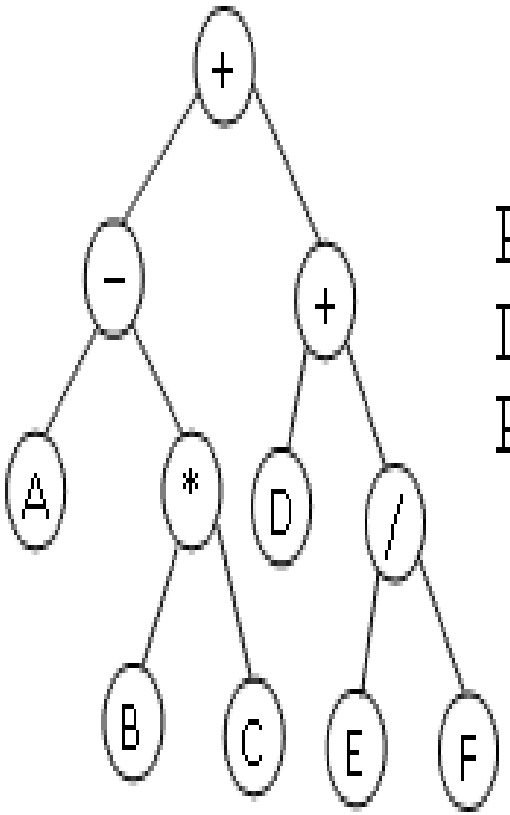
    4, 7, 8, 6, 11, 13, 12, 14, 17, 16, 19, 18, 15, 10



Application of binary tree traversal

- Used for mathematical expression evaluation
  - Store
    - ✓**Values/operands** on leaf nodes and
    - ✓**Operators** on internal nodes
  - **Pre-order traversal**
    - ✓Used to generate mathematical expression in prefix notation.
  - **In-order traversal**
    - ✓Used to generate mathematical expression in infix notation
  - **Post-order traversal**
    - ✓Used to generate mathematical expression in postfix notation

Application of binary tree traversal …



Preorder traversal - + – A * B C + D / E F → Prefix notation

Inorder traversal - A – B * C + D + E / F → Infix notation

Postorder traversal - A B C * – D E F / + + → Postfix notation

Implementation of BST traversals

- Consider the following definition of binary search tree

```
struct Node
{
   int Num;
   Node * Left, *Right;
};
Node *RootNodePtr=NULL;
```

- **Pre-order traversal:-**

```
void Preorder (Node *CurrNodePtr)
{
     if(CurrNodePtr ! = NULL)
     {
        cout<< CurrNodePtr->Num;  // or any operation on the node
        Preorder(CurrNodePtr->Left);
        Preorder(CurrNodePtr->Right);
      }
}
```

- ***It is called at first as:*** Preorder(RootNodePtr);

Implementation BST traversals…

- **In-order traversal:-**

```
void Inorder (Node *CurrNodePtr)
{
    if(CurrNodePtr ! = NULL)
    {
      Inorder(CurrNodePtr->Left);
      cout<< CurrNodePtr->Num;  // or any operation on the node
      Inorder(CurrNodePtr->Right);
     }
}
```

- ***It is called at first as:*** *Inorder(RootNodePtr);*

Implementation BST traversals…

- **Post-order traversal:-**

```
void Postorder (Node *CurrNodePtr)
{
    if(CurrNodePtr ! = NULL)
    {
      Postoder(CurrNodePtr->Left);
      Postorder(CurrNodePtr->Right);
      cout<< CurrNodePtr->Num;  // or any operation on the node
     }
}
```

- ***It is called at first as:*** *Postorder(RootNodePtr);*

Implementation BST Search

- One of the three traversal methods can be used.
- Return type of the SearchBST function can be
  - Pointer that points to the node containing or
  - Simple Boolean type True/False

```
Node * SearchBST (Node *RNP, int x)   //'x' is the number to be searched
{
  if((RNP = = NULL) || (RNP->Num = = x))
      return(RNP);
  else if(RNP->Num > x)
      return(SearchBST(RNP->Left, x));
  else
      return(SearchBST (RNP->Right, x));
}
```

- **It is called at first as:-**

```
Node *SearchedNodePtr=NULL;
SearchedNodePtr = SearchBST (RootNodePtr, Number);
```

Inserting node in BST

- When a node is inserted, the definition of binary search tree should be preserved
- **Case-1**:- There is no data in the tree (i.e. RootNodePtr is NULL)
  - The new node is made the root node
- **Case-2**:- There is data in the tree (i.e. RootNodePtr is not NULL)
  - Search the appropriate position
  - Insert the new node in that position
- Let the node to be inserted is pointed by ***InsNodePtr***
- **Function call:-**

```
if(RootNodePtr = = NULL)
        RootNodePtr=InsNodePtr;
else
        InsertBST(RootNodePtr, InsNodePtr);
```

Inserting node in BST…

```
void InsertBST(Node *RNP, Node *INP)
{
  if(RNP->Num>INP->Num)
    {
      if(RNP->Left==NULL)
          RNP->Left = INP;
       else
      InsertBST(RNP->Left, INP);
    }
  else
    {
      if(RNP->Right==NULL)
      RNP->Right = INP;
       else
      InsertBST(RNP->Right, INP);
    }
}
```

Approaches of deleting node from BST

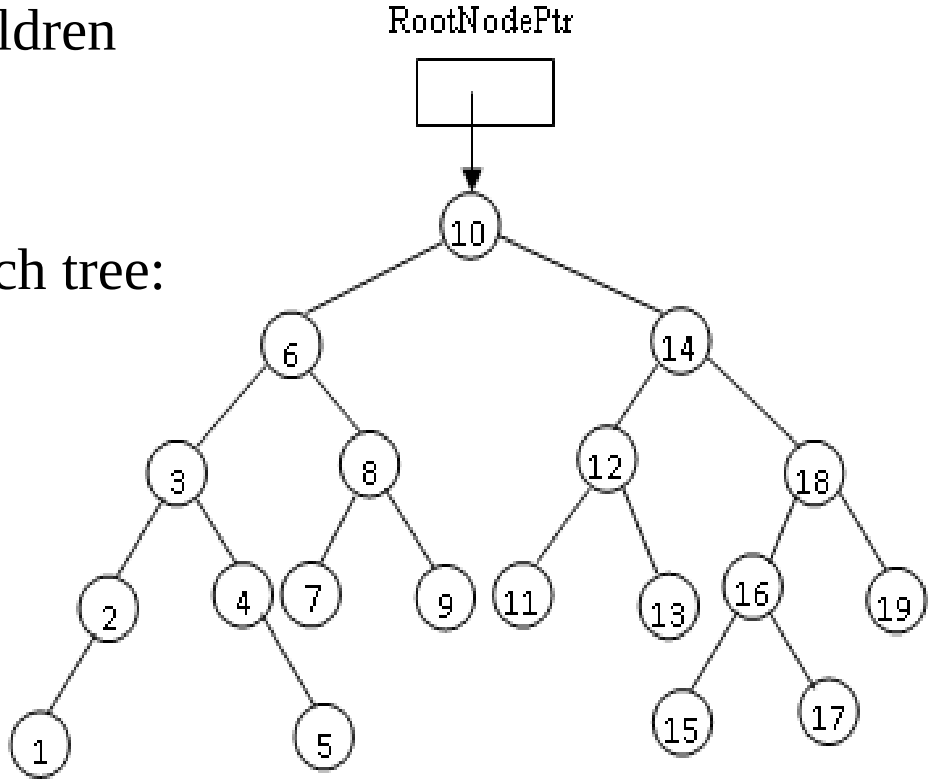
- Two approaches are used in replacing position of the deleted node
  - If the deleted node is ***replaced from its direct children***, then the deletion is called **Deleting by Merging**
  - If the deleted node is ***replaced from its leaf children***, this way of deleting a node is called **Deleting by Copying**
- **Approach-1**: **Deleting by copying**
  - Deleted node is ***replaced from its leave children***
  - So, it is replaced
    - By the largest element from the ***left*** *sub-tree* OR
    - By the smallest element from the **right** sub-tree

Approaches of deleting node from BST…

- **<u>Approach-2</u>: Deleting by merging**
  - Deleted node is ***replaced from its direct children***
  - So, if it is replaced by its direct **left child,** the right child (including its descendants) is **made right child of the node** containing the **largest** element in the left of the deleted node.
  - if it is replaced by its direct **right child,** the left child (including its descendants) is made **left child of the node** containing the **smallest** element in the right of the deleted nodes

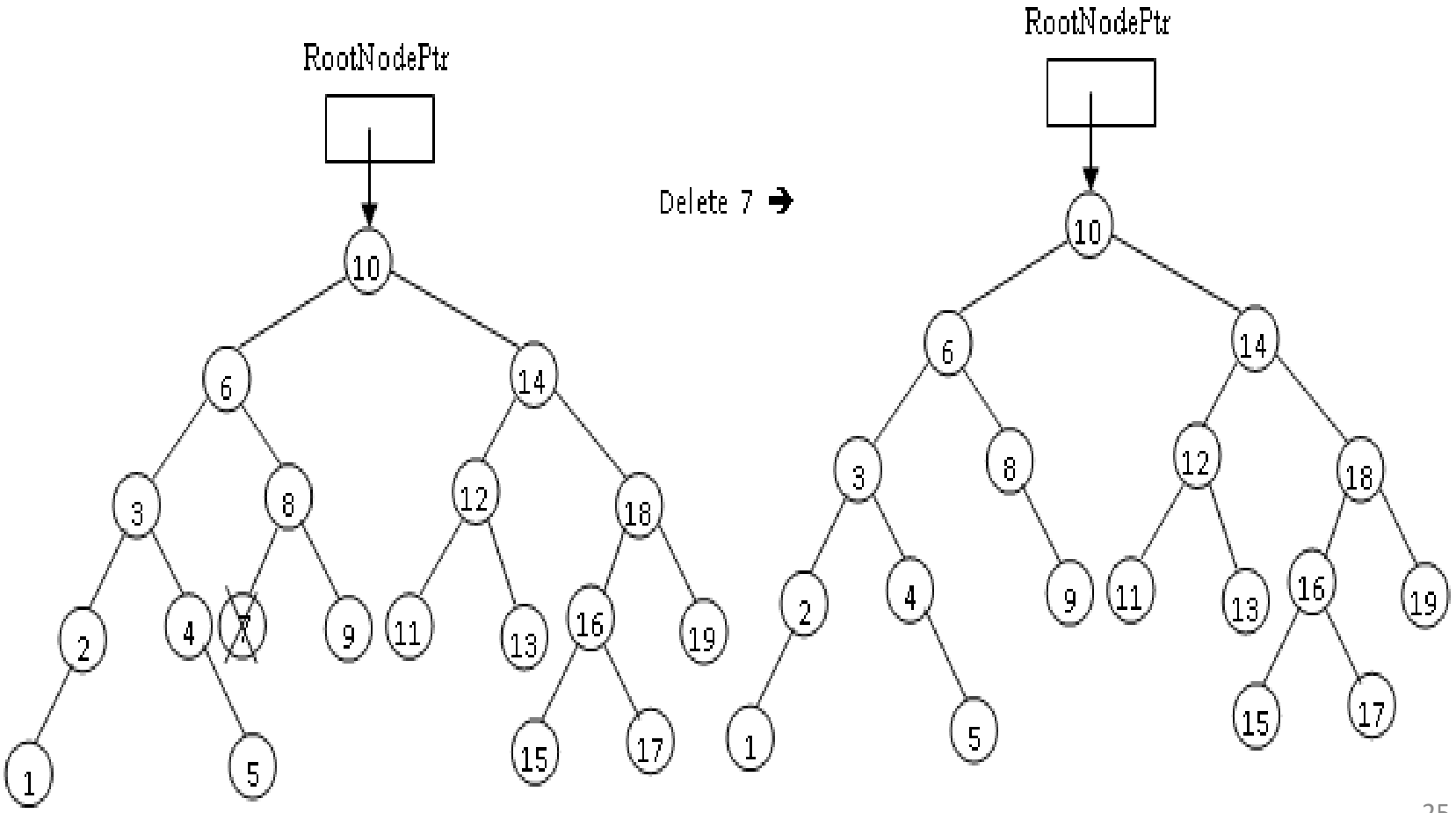
Deleting node from BST

- **4-Cases should be considered**
  - Deleting a leaf node (a node having no child)
  - Deleting a node having only one child
  - Deleting a node having two children
  - Deleting the root node
- Consider the following binary search tree:



Deleting node from BST…

- **Case-1**: Deleting a leaf node (a node having no child)
  - Find the node and delete it directly Example → 7



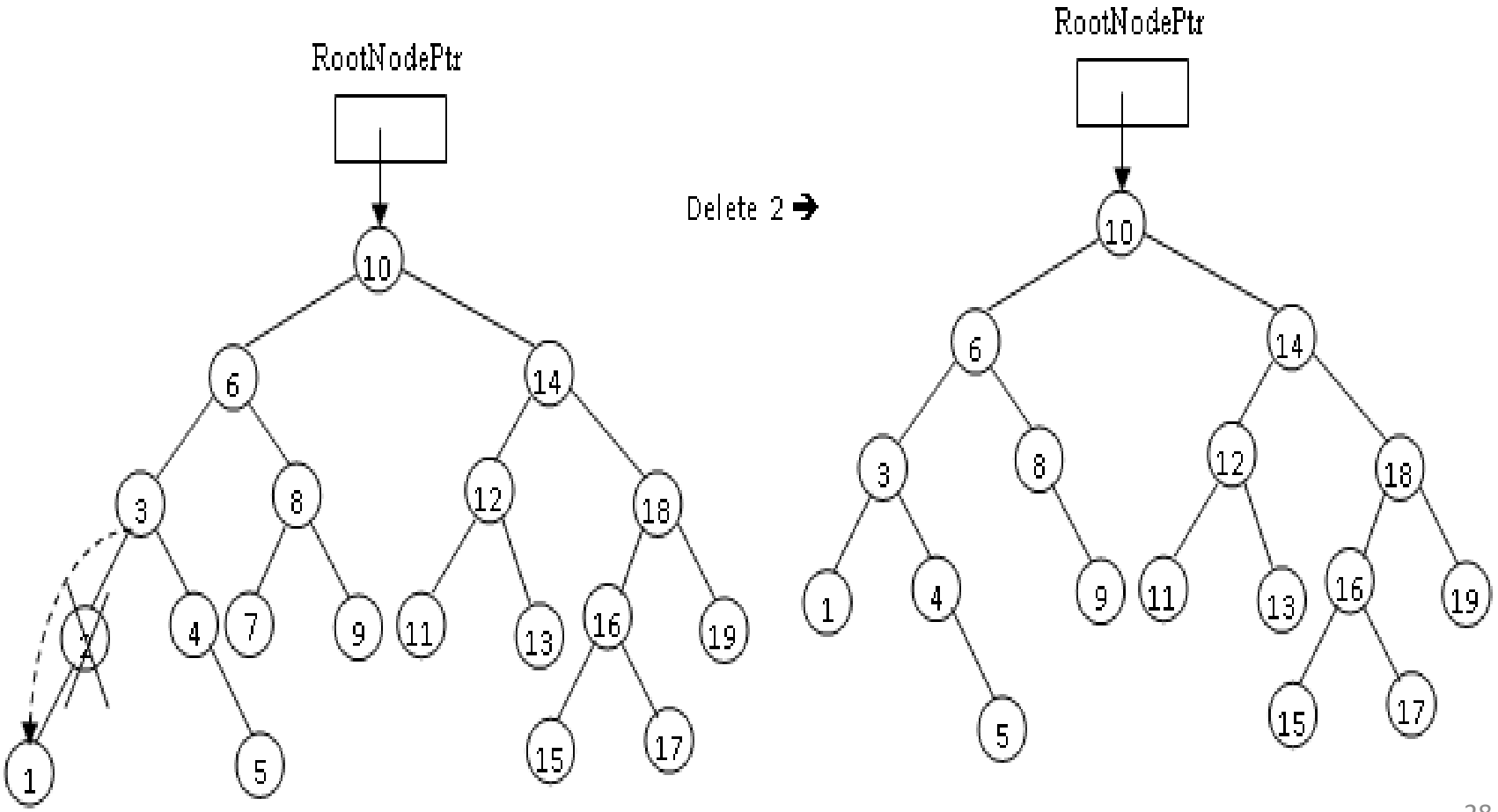
Deleting node from BST…

- **Case 2: Deleting a node having only one child, e.g. 2**
  - If the deleted node is the **left child** of its parent and the deleted node has only the left child, the left child of the deleted node is made the left child of the parent of the deleted node.
  - If the deleted node is the **left child** of its parent and the deleted node has only the right child, the right child of the deleted node is made the left child of the parent of the deleted node.

- If the deleted node is the **right child** of its parent and the node to be deleted has only the left child, the left child of the deleted node is made the right child of the parent of the deleted node.
- If the deleted node is the **right child** of its parent and the deleted node has only the right child, the right child of the deleted node is made the **right child** of the parent of the deleted node.

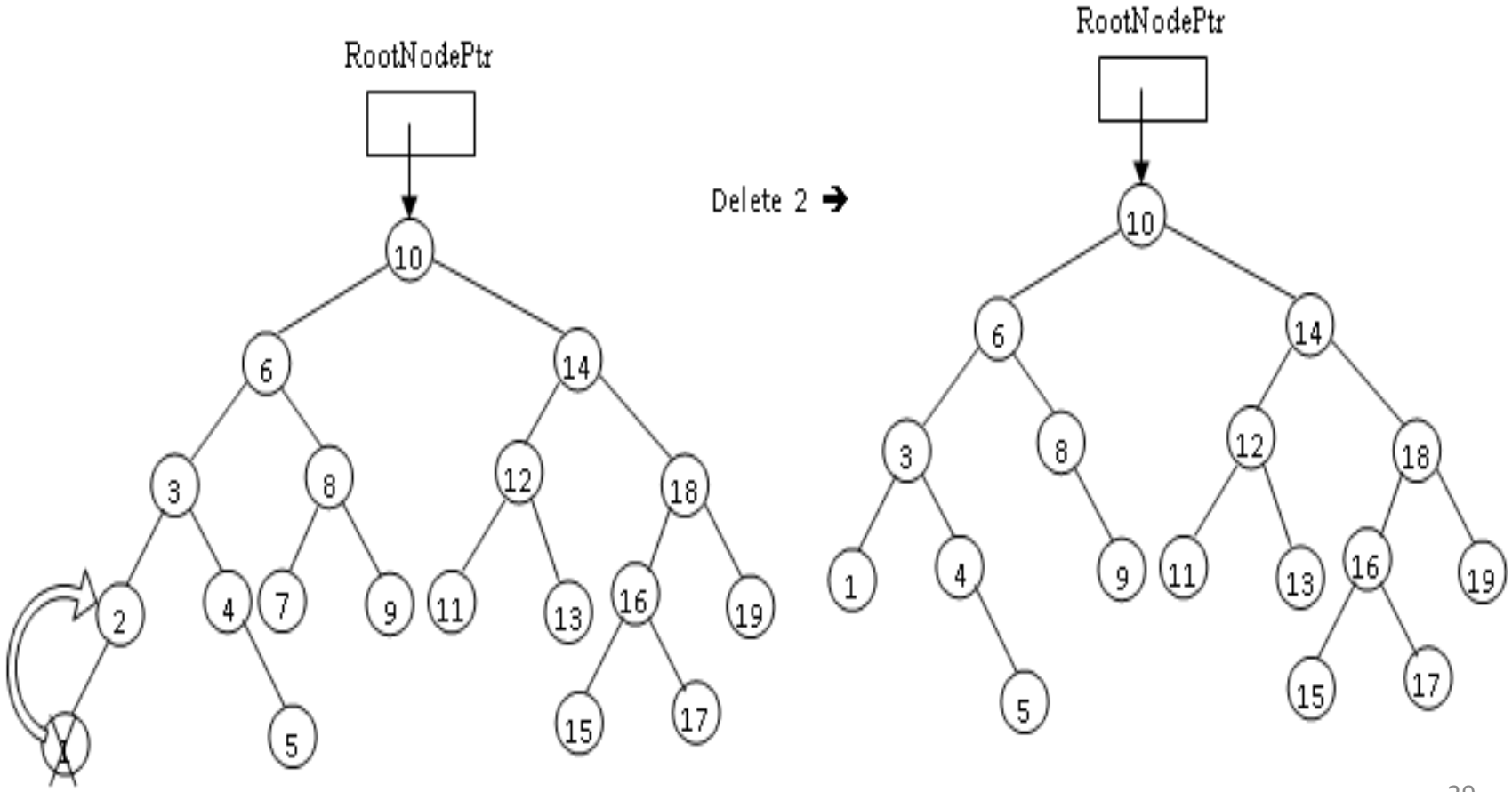
Deleting node from BST…

- **Case-2: Deleting a node having only one child**
  - **<u>Approach 1</u>:Deletion** by **merging** → replace it from its direct child



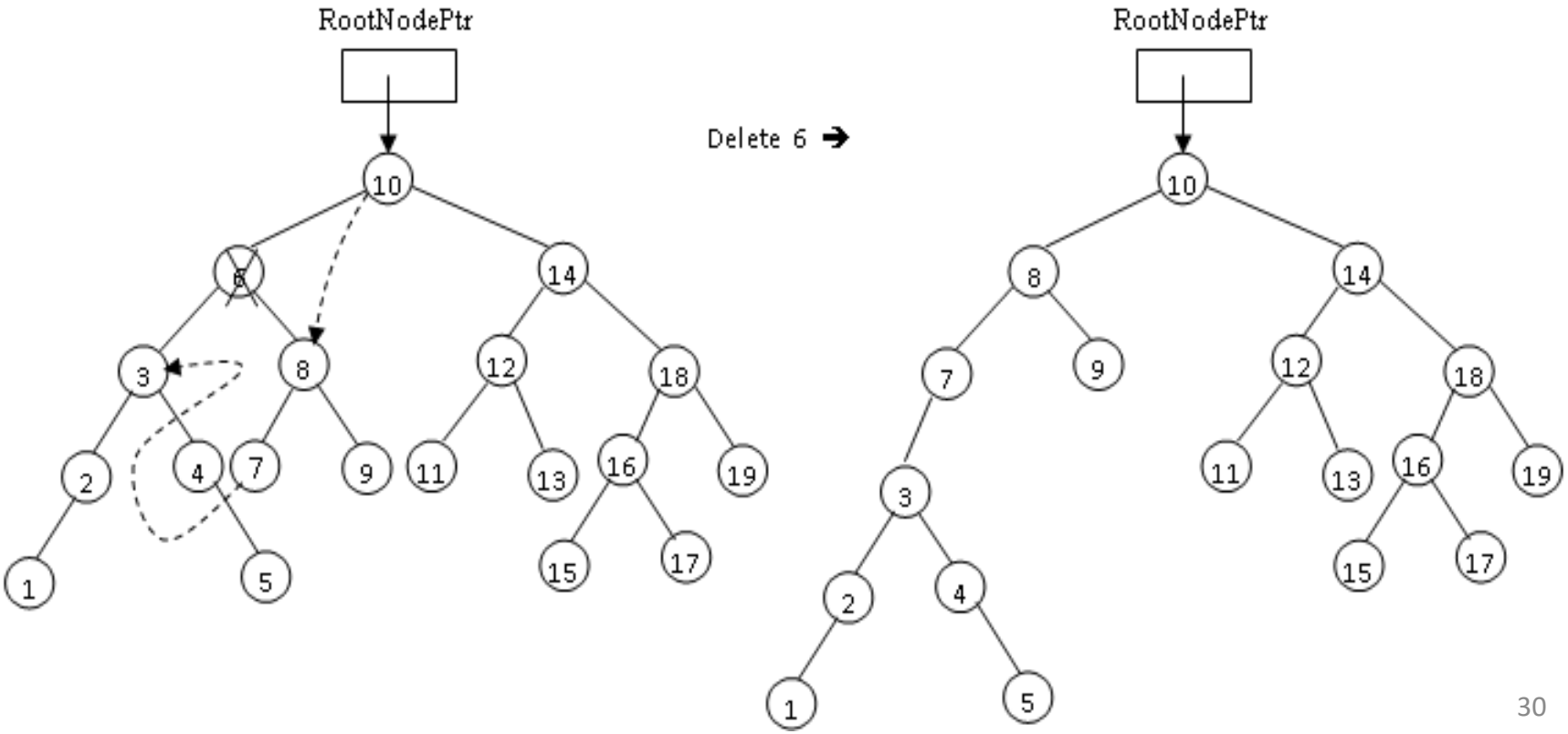
Deleting node from BST…

- **Case-2: Deleting a node having only one child**
  - **<u>Approach 2</u>**: Deletion by **copying** – replace it from its **leave** children



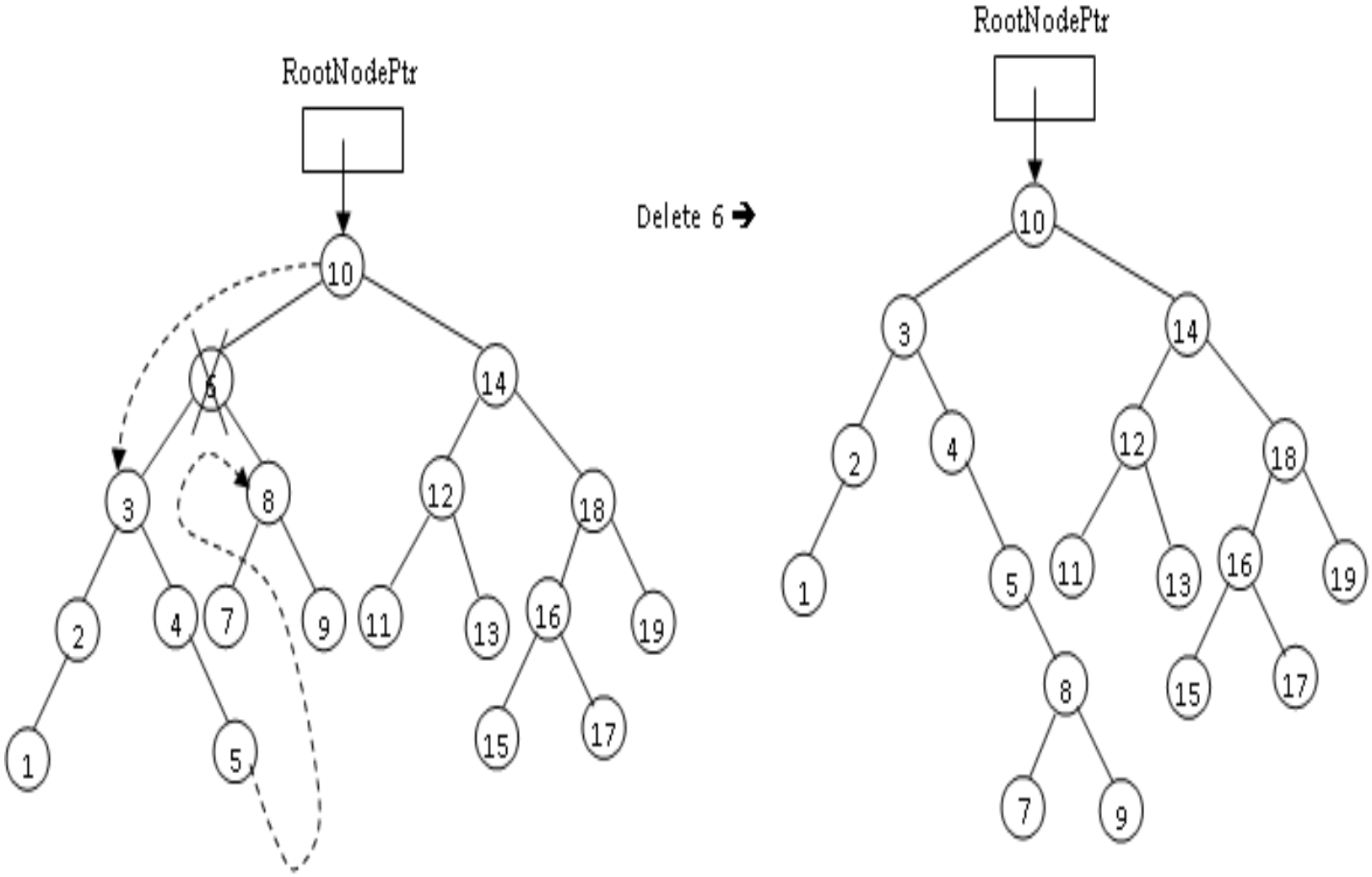
Deleting node from BST…

- **Case-3: Deleting a node having two children**
  - **<u>Approach 1</u>**:Deletion by **merging** – replace it from its direct child
  - ***Two possible*** answers: ***Option A…using right child of right subtree***



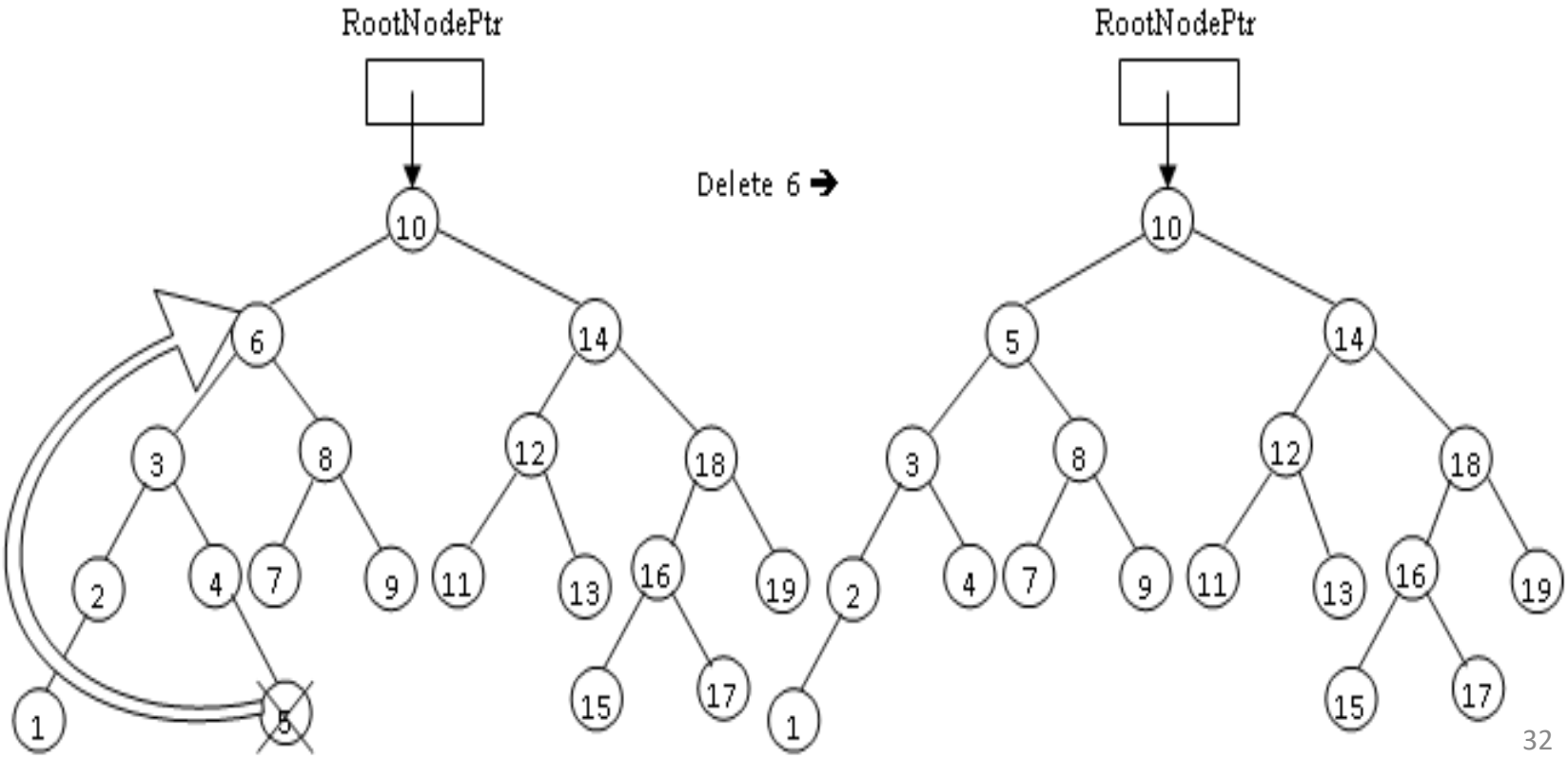
Deleting node from BST…

- ***Option B…Using left child of left sub tree***



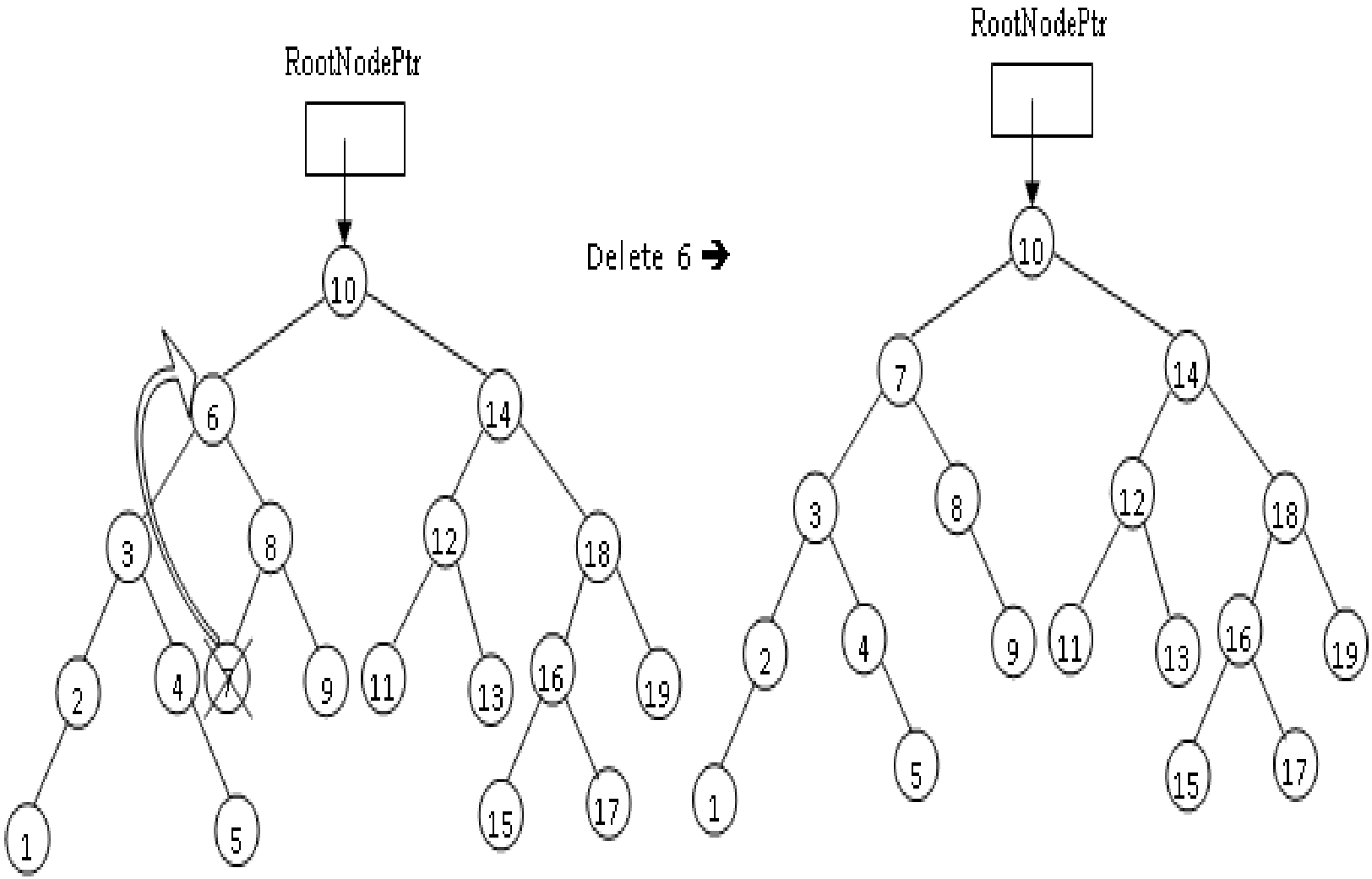
Deleting node from BST…

- **Case-3: Deleting a node having two children**
  - **<u>Approach 2</u>**: Deletion by **copying** – replace it from its **leave** children
  - ***Two possible*** answers: ***Option A…using right leaf node of left subtree***



Deleting node from BST…

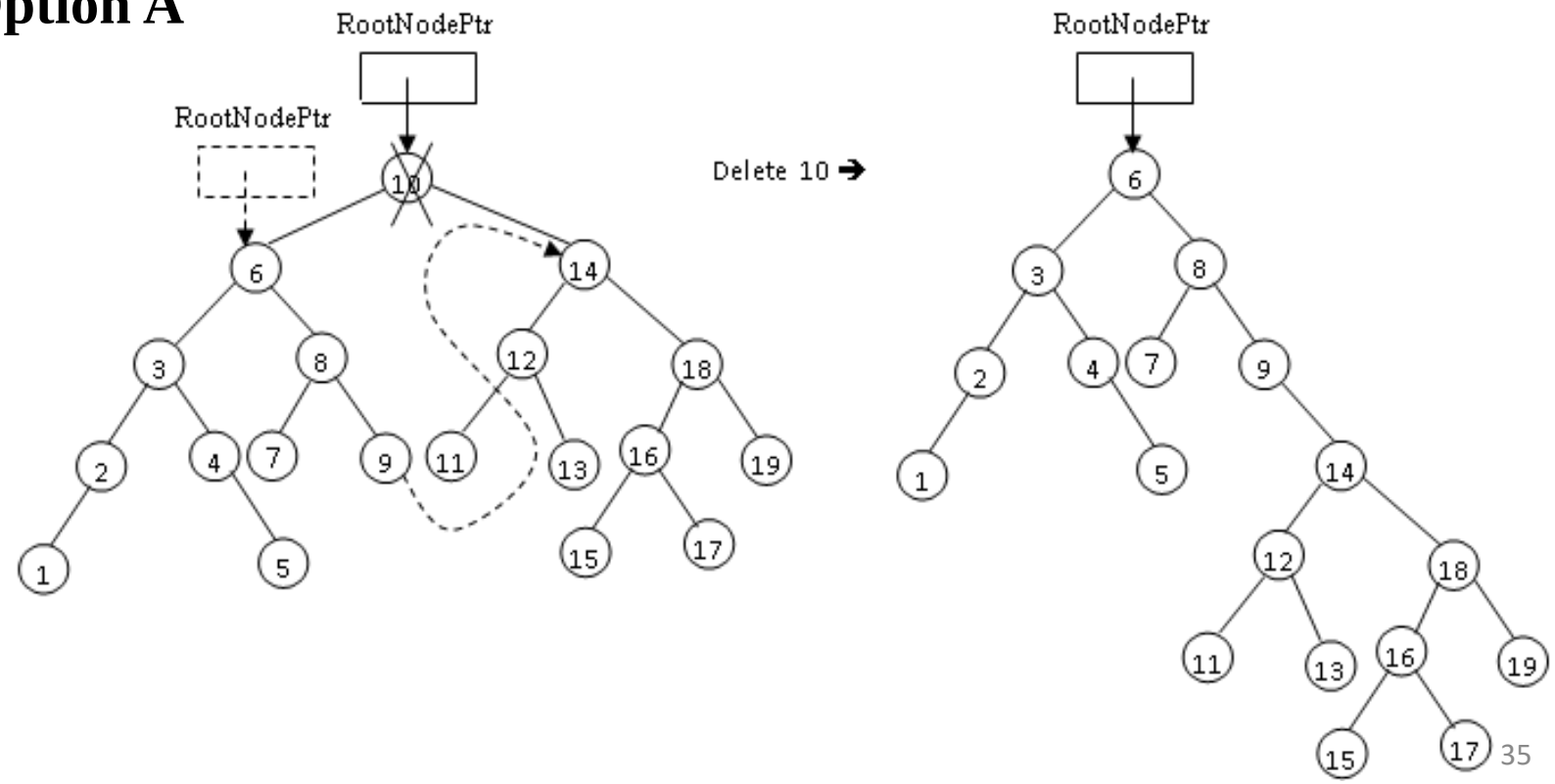
- **Option B…Using left child of right subtree**



Deleting node from BST…

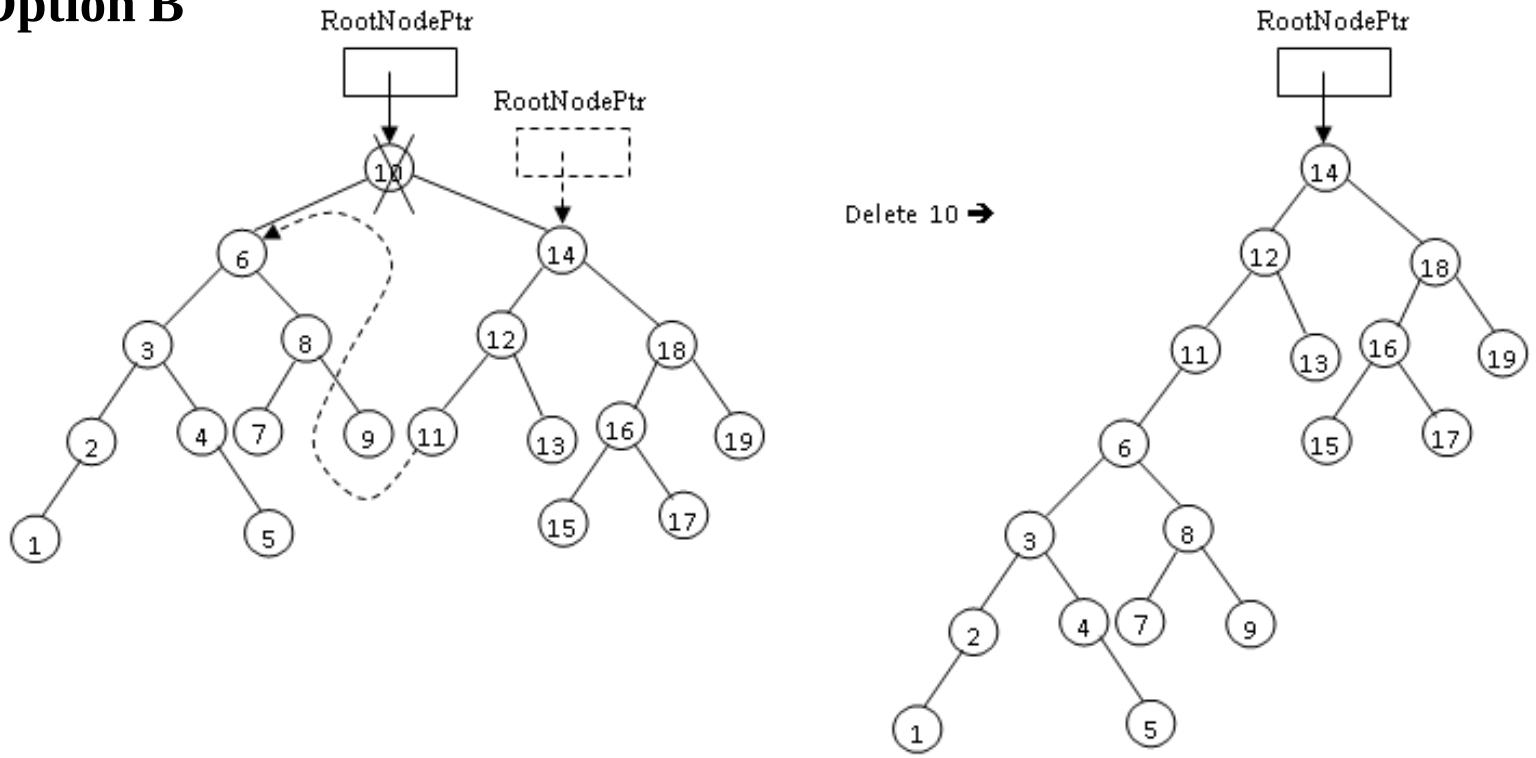
- **Case-4: Deleting the root node**
  - If the tree has only one node, make root node pointer NULL
  - **<u>Approach 1</u>:** Deletion by **merging** – replace it from its direct children
- If the tree has only one node the root node pointer is made to point to nothing (NULL)

- **Option A:** If the root node has **left** child
  - The root node pointer is made to point to the left child
  - The right child of the root node is made the right child of the node containing the largest element in the left of the root node
  - **Option A**



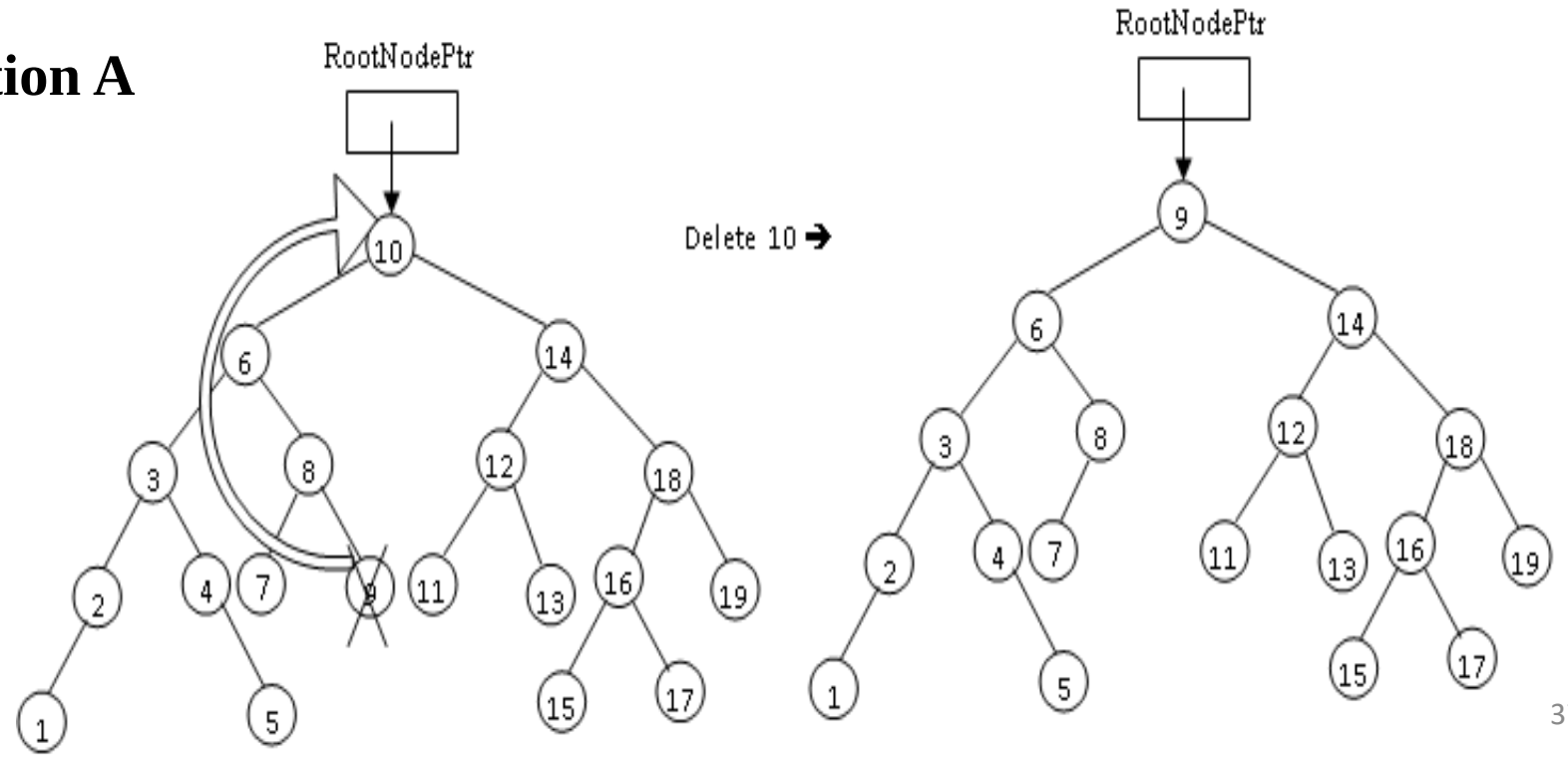
Deleting node from BST…

- **Option B:** If root node has **right** child
  - The root node pointer is made to point to the right child
  - The left child of the root node is made the left child of the node containing the smallest element in the right of the root node
  - **Option B**



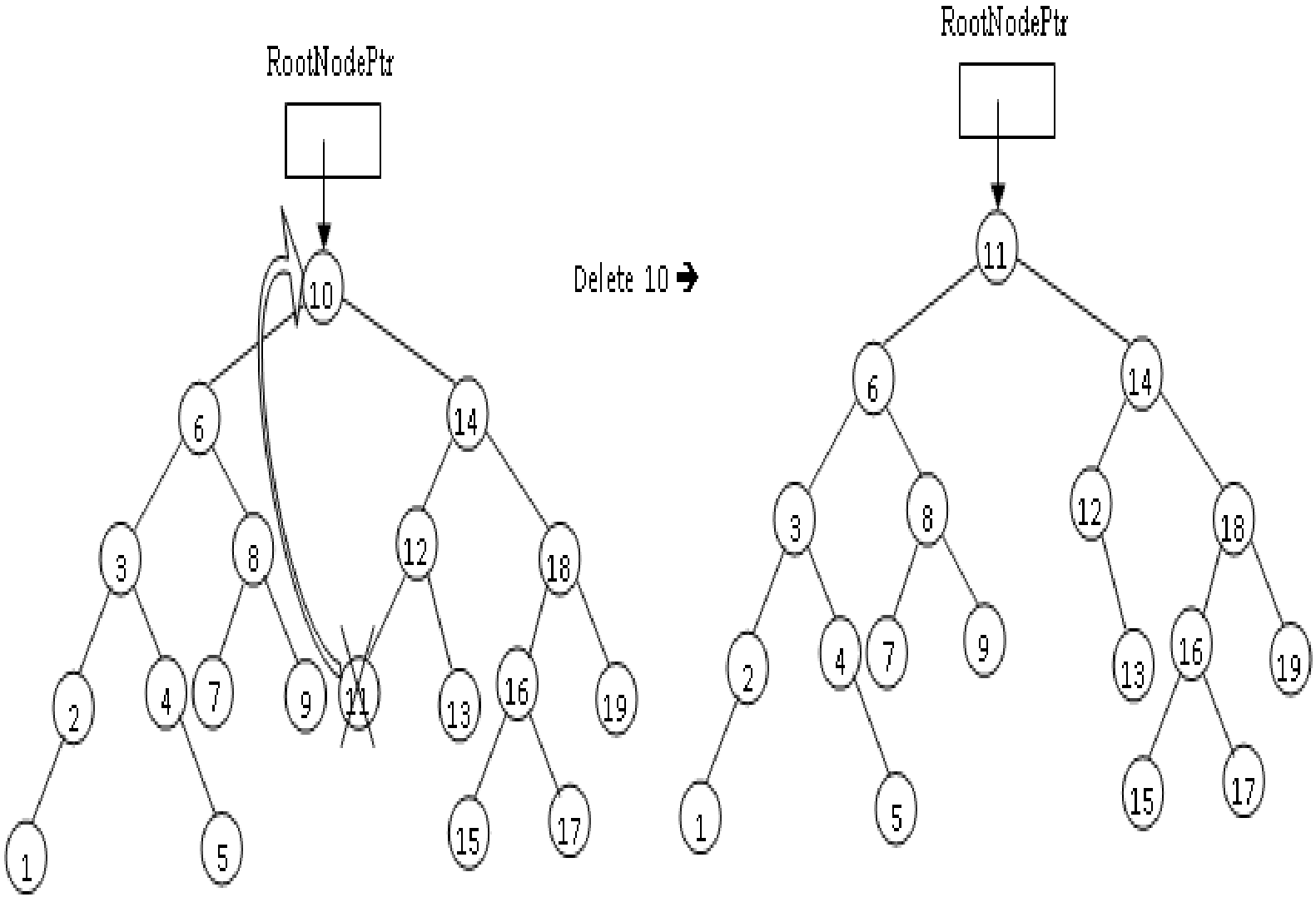
Deleting node from BST…

- **Case-4: Deleting the root node**
  - **Approach 2**: Deletion by **copying** – replace it from its **leave** children
- Copy the node containing the **largest element** in the left (or the **smallest** element in the right) to the node containing the element to be deleted
- Delete the copied node : has ***two possible*** answers:
- **Option A**



Deleting node from BST…

- **Option B**



Chapter End!!!