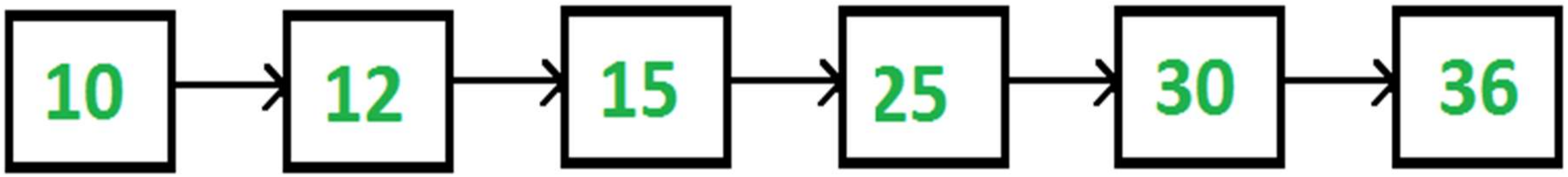
The Tree Data Structure

Dr. Amit Praseed

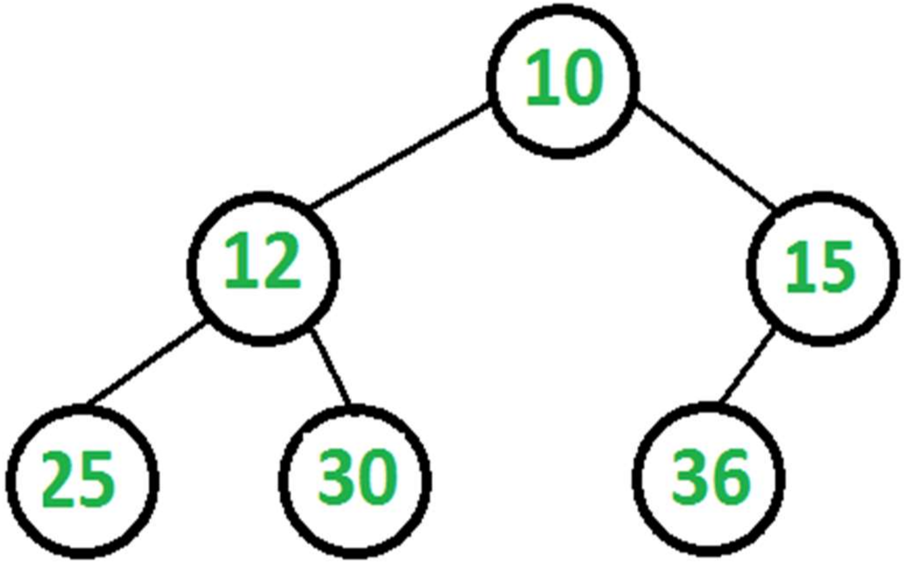
Trees vs Linked Lists

- In a linked list, every node (except the last) is connected to exactly two nodes – a predecessor node and a successor node
- A linked list fails to efficiently represent many of the relationships that are commonly encountered in real life, for example organizational hierarchies or family trees
- In such situations, a node many need to have multiple successors, which gives rise to the concept of a tree

Trees vs Linked Lists



Linked List

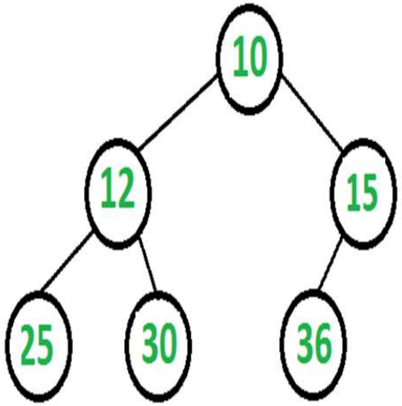


Tree

A Formal Definition of a Tree

- A tree is a collection of nodes such that either
  - the collection is empty, or
  - the collection contains the following
    - A distinguished node r called the root
    - Zero or more non-empty subtrees T_1, T_2, … T_k, each of whose roots are connected by an edge from r
- A tree is a collection of N nodes, one of which is a designated node called the root, and contains N-1 edges

Tree Terminology



- Node 10 is called the root of the tree
- Nodes 12 and 15 are the children of node 10
- Node 10 is the parent of nodes 12 and 15
- *Subtree* of a node: A tree whose root is a child of that node
- **Depth of a node**: Number of edges from the node to the tree's root node.
- **Height of a node:** Number of edges on the ***longest path*** from the node to a leaf.
- **Height of a Tree = Height of Root**

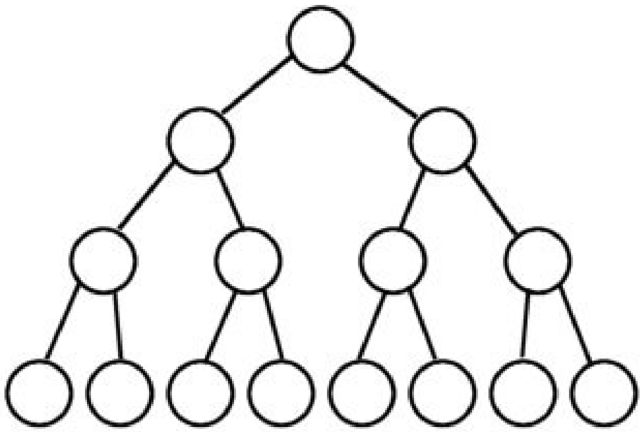
Binary Trees

- *Binary* tree: a node has *atmost 2* non-empty subtrees
- Set of nodes T is a binary tree if either of these is true:
  - T is empty
  - Root of T has two subtrees, both of which are binary trees

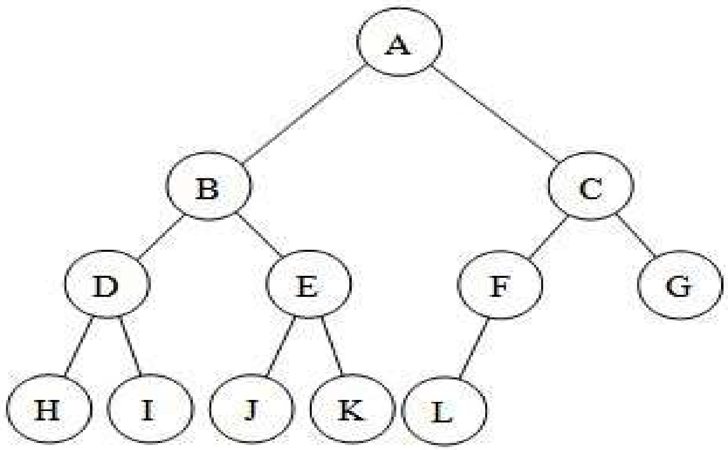
Complete and Full Binary Trees

- A full binary tree is a tree in which every node other than the leaves has two children.
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- Perfect Binary Tree??

Full Binary Tree

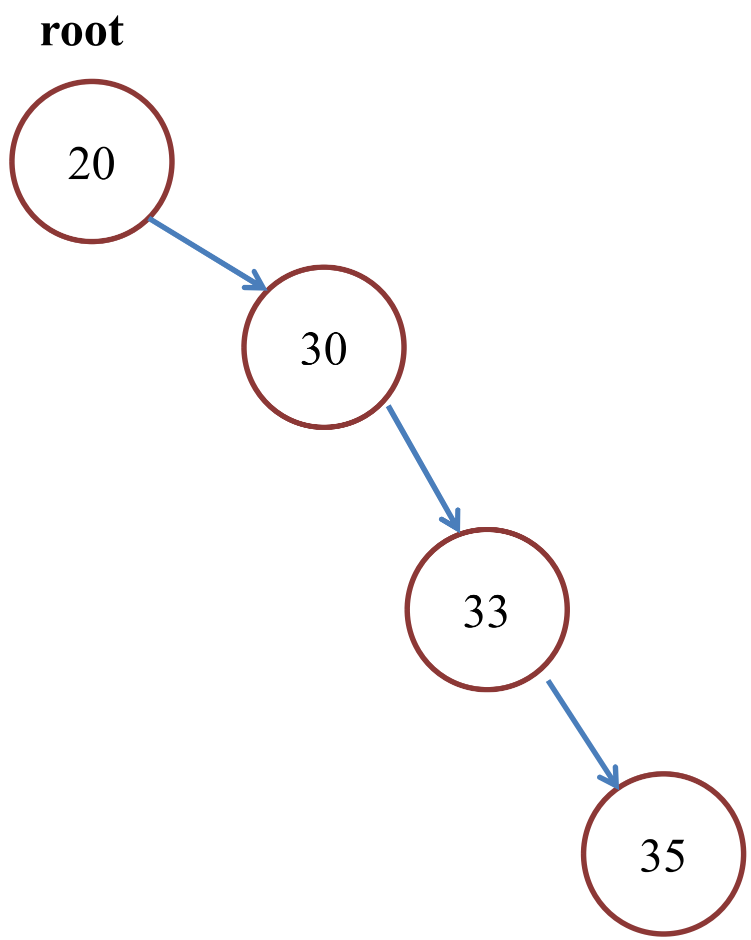


Complete Binary Tree



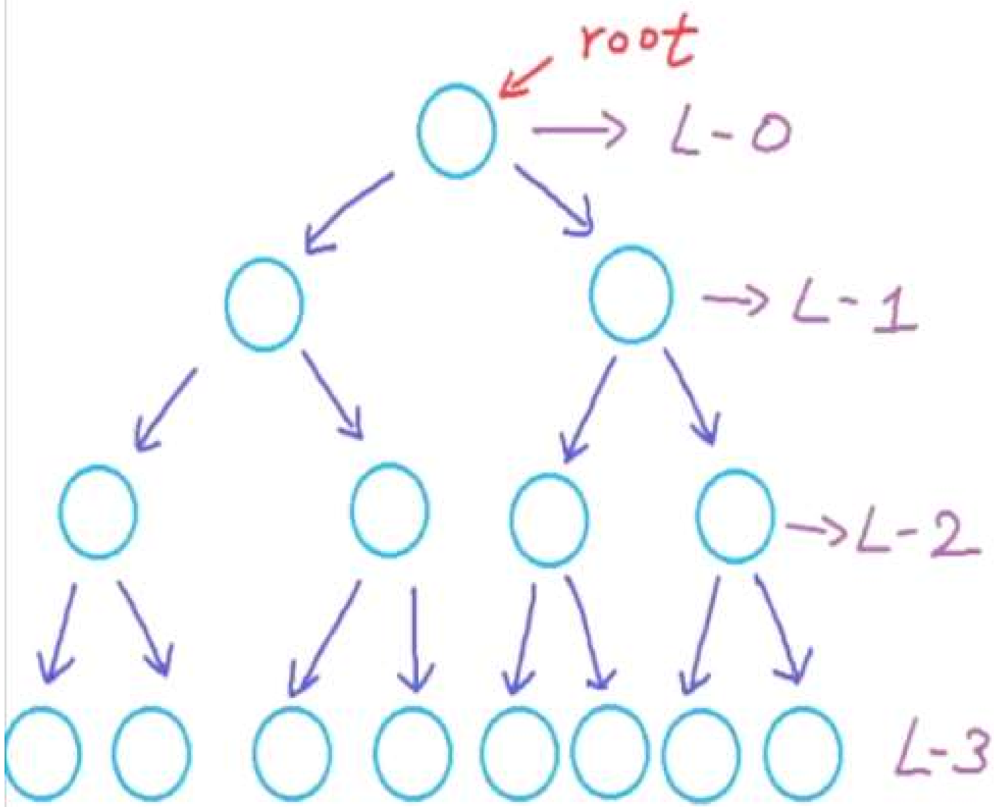
Height of a Binary Tree

- The height of a full or complete binary tree with n nodes will be log n [Why?]
- However, in the worst case, a binary tree may degenerate into a linked list
  - Such a tree is called a skewed tree
  - Height of a completely skewed tree will be n-1
- Thus the height of a binary tree is *[n-1, log n]*



A right skewed (binary) tree

Height of Perfect Binary Tree



Perfect Binary tree

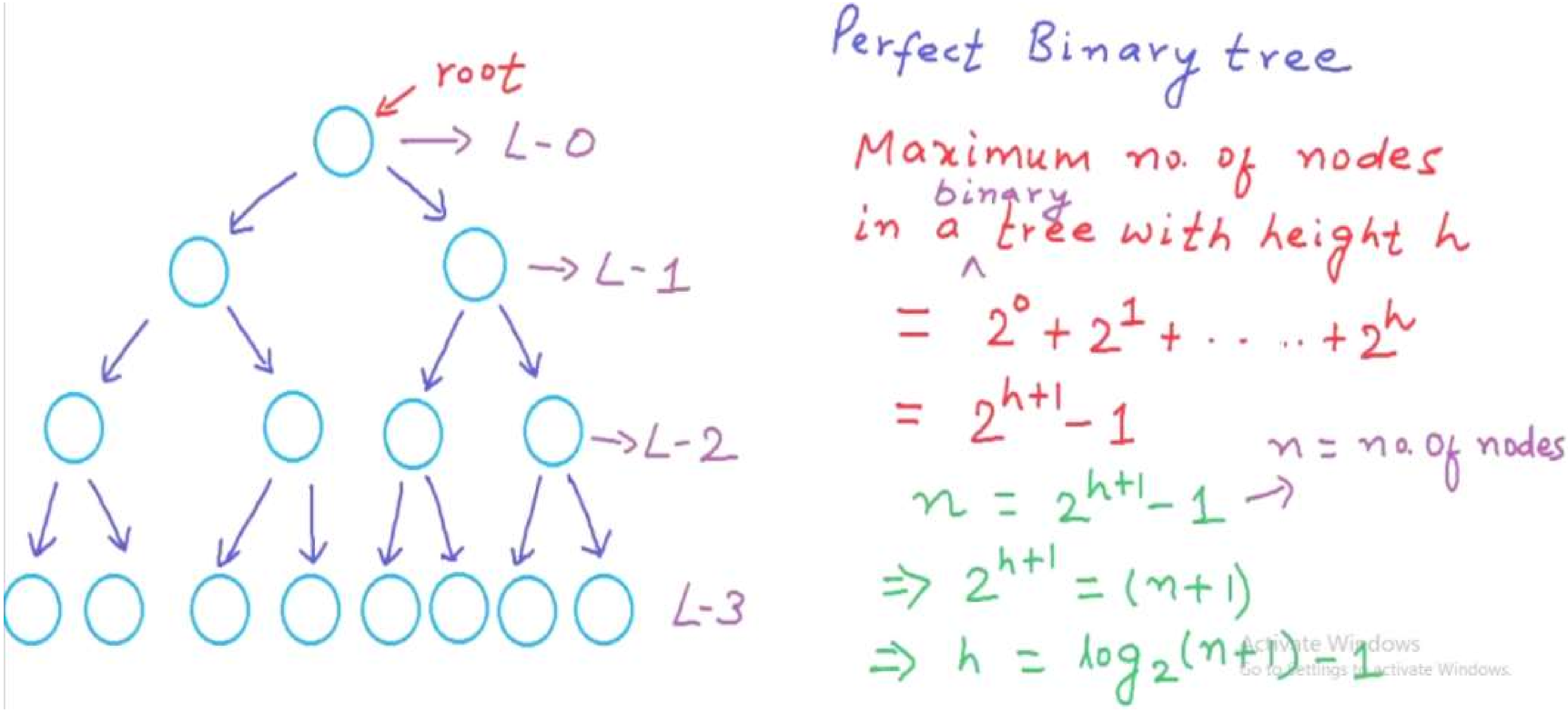
Maximum no. of nodes in a binary tree with height h

$= 2^0 + 2^1 + \dots + 2^h$

$= 2^{h+1} - 1$

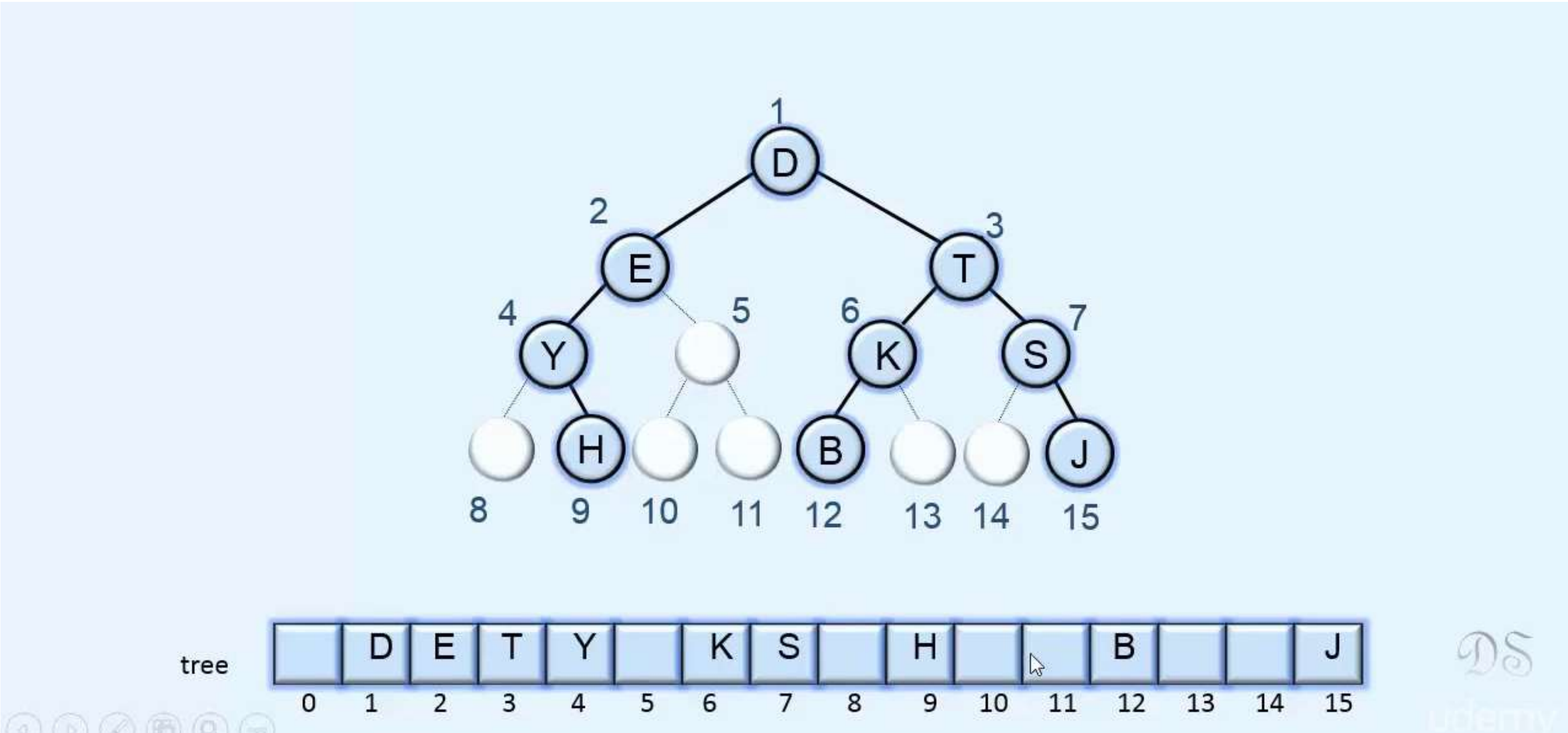
$= 2^{(\text{no. of levels})} - 1$

Contd…



Height of Perfect Binary Tree = $\log_2(n)$

Array Representation of Binary Trees



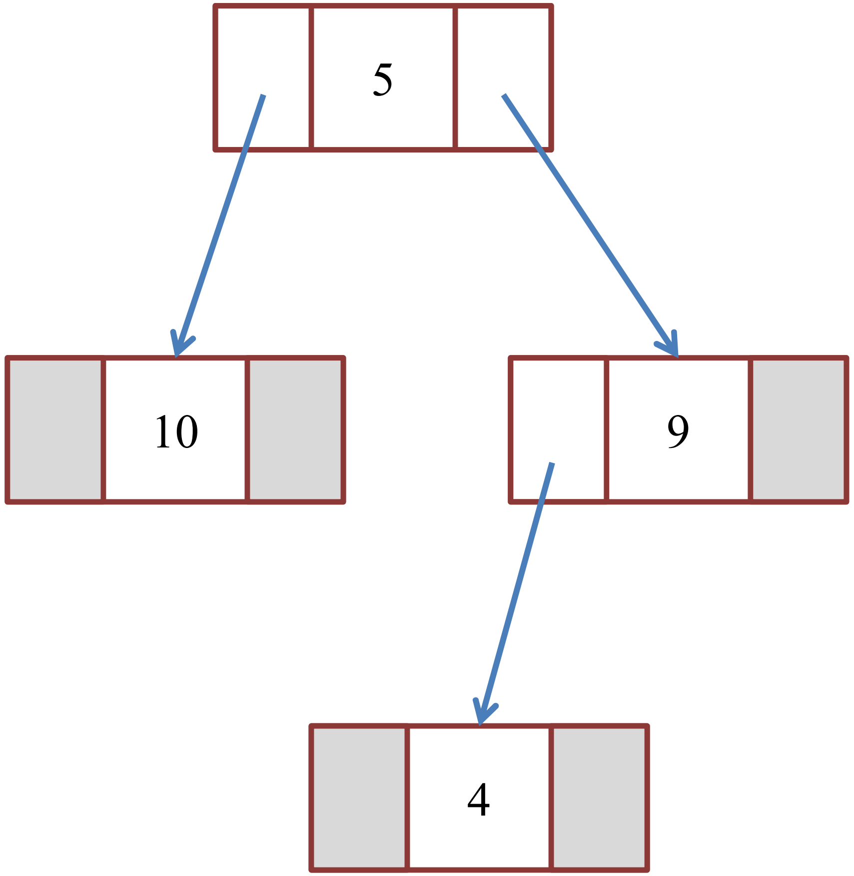
Parent of node at location i is at location i/2

Children of a node at location i are at locations 2*i and 2*i+1

Binary Trees using Pointers

- Every node within a binary tree can be represented as a structure

```
struct node
{
        int data;
        struct node * lchild;
        struct node * rchild;
}
```



Traversing a Binary Tree

- A binary tree can be traversed by simply following its lchild and rchild pointers
- This is not as trivial as a linked list, which has only one possible order of traversal
- A binary tree has multiple ways of traversal, depending upon whether the node, left child or right child are explored first
  - Inorder Traversal
  - Preorder Traversal
  - Postorder Traversal

In-order Traversal

- During the in-order traversal algorithm, the left subtree is explored first, followed by root, and finally nodes on the right subtree.

INORDER (root)

INORDER(root → lchild)

PROCESS(root)

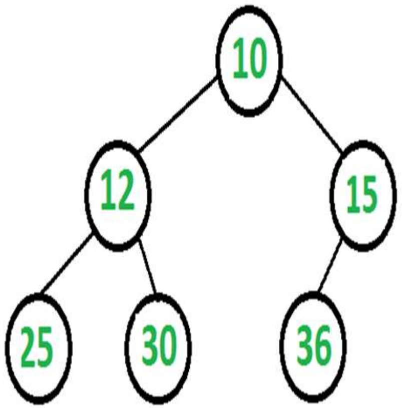
INORDER(root → rchild)

Post-order and Pre-order Traversal

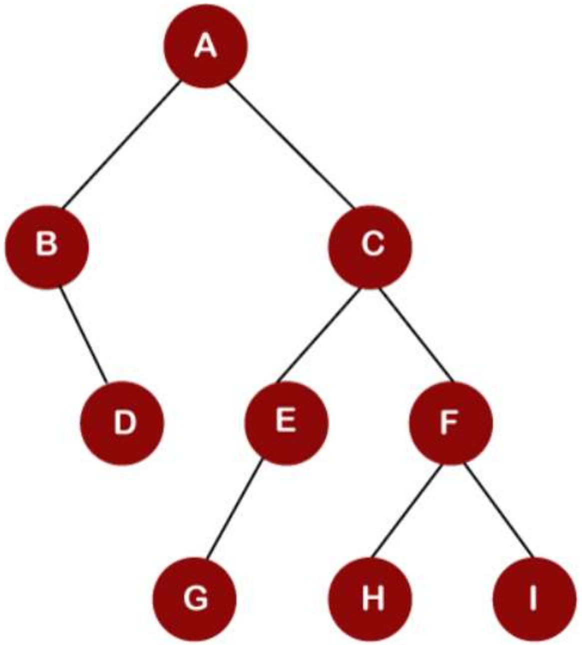
- During the post-order traversal algorithm, the left subtree is explored first, followed by nodes on the right subtree, and finally the root.
- During the pre-order traversal algorithm, the root is explored first, followed by nodes on the left subtree, and finally the nodes on the right subtree.

EXERCISE: Write the recursive algorithm for post-order and pre-order traversal algorithms

Examples



In-order – 25 12 30 10 36 15
Pre-order – 10 12 25 30 15 36
Post-order – 25 30 12 36 15 10



In-order – B D A G E C H F I
Pre-order – A B D C E G F H I
Post-order – D B G E H I F C A

Binary Search Tree (BST)

- A Binary Search Tree (BST) is a binary tree which has the following special properties:
  - The left subtree of a node contains only nodes with keys lesser than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - The left and right subtree each must also be a binary search tree.

Inserting Elements into a BST

```
INSERT_BST(root, node)
    if root == NULL
        root = node
    else
        if node→data < root→data
            INSERT_BST(root→lchild, node)
        if node→data > root→data
            INSERT_BST(root→rchild, node)
```

Inserting Elements into a BST

Current BST

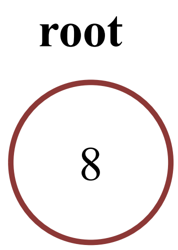
root= NULL

Element to be added

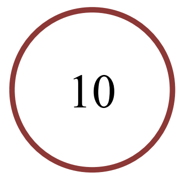
8

Inserting Elements into a BST

Current BST

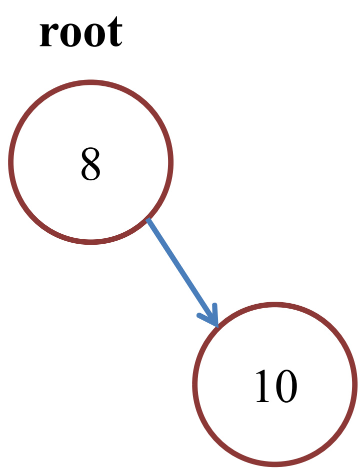


Element to be added

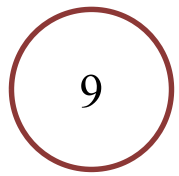


Inserting Elements into a BST

Current BST

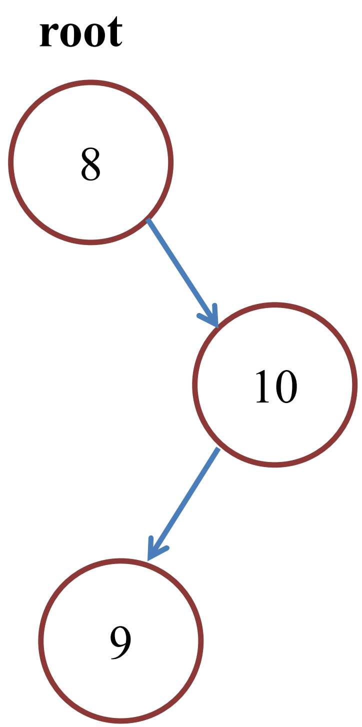


Element to be added

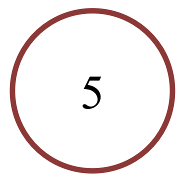


Inserting Elements into a BST

Current BST

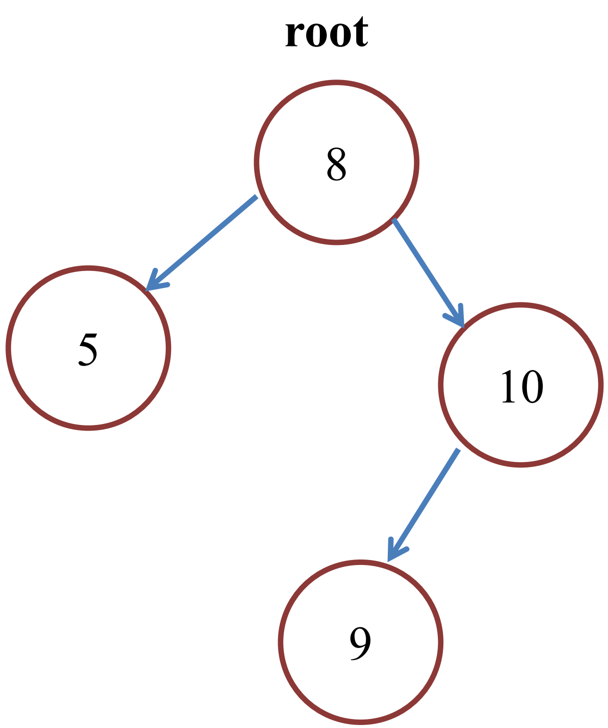


Element to be added

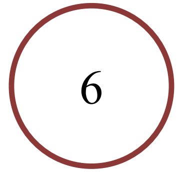


Inserting Elements into a BST

Current BST

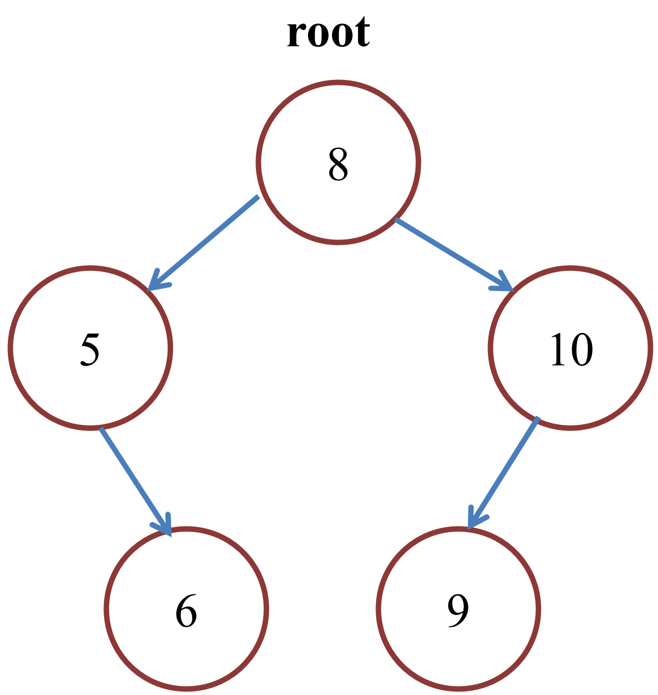


Element to be added



Inserting Elements into a BST

Current BST



Complexity of *INSERT_BST*

- The *INSERT_BST* function explores exactly one path within the tree
  - There is no backtracking or going back within the algorithm
- So the complexity of *INSERT_BST* depends upon the maximum number of nodes in any such path within the tree
  - Which is nothing but the height of the tree
  - Complexity of *INSERT_BST* = $O(h)$, where h is the height of the tree

Searching a BST

- While searching for an element within a BST, we can employ the same strategy as in binary search
  - All elements greater than the root node are in the right subtree of the root
  - All elements greater than the root node are in the right subtree of the root
  - This definition follows recursively
  - If we reach a leaf node and still were unable to find the element, the element is not present in the BST

Searching a BST

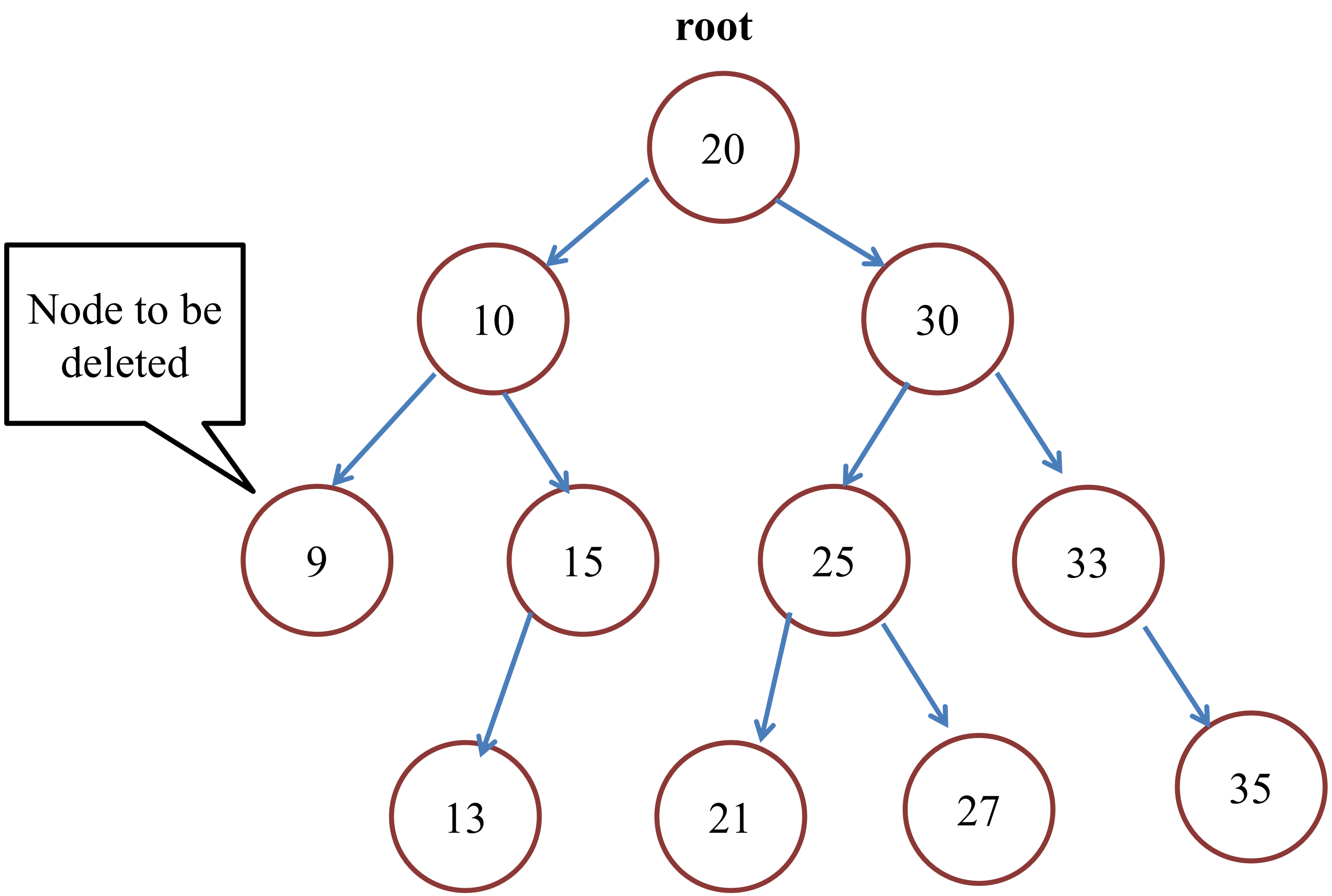
```
SEARCH_BST (root, key)
        ptr = root
        while ptr !=NULL, do
                if ptr→data==key
                        return ptr
                else if ptr→data > key
                        ptr = ptr→lchild
                else
                        ptr = ptr→rchild
        return NULL
```

EXERCISE: What is the complexity of this algorithm?

Deleting an Element from a BST

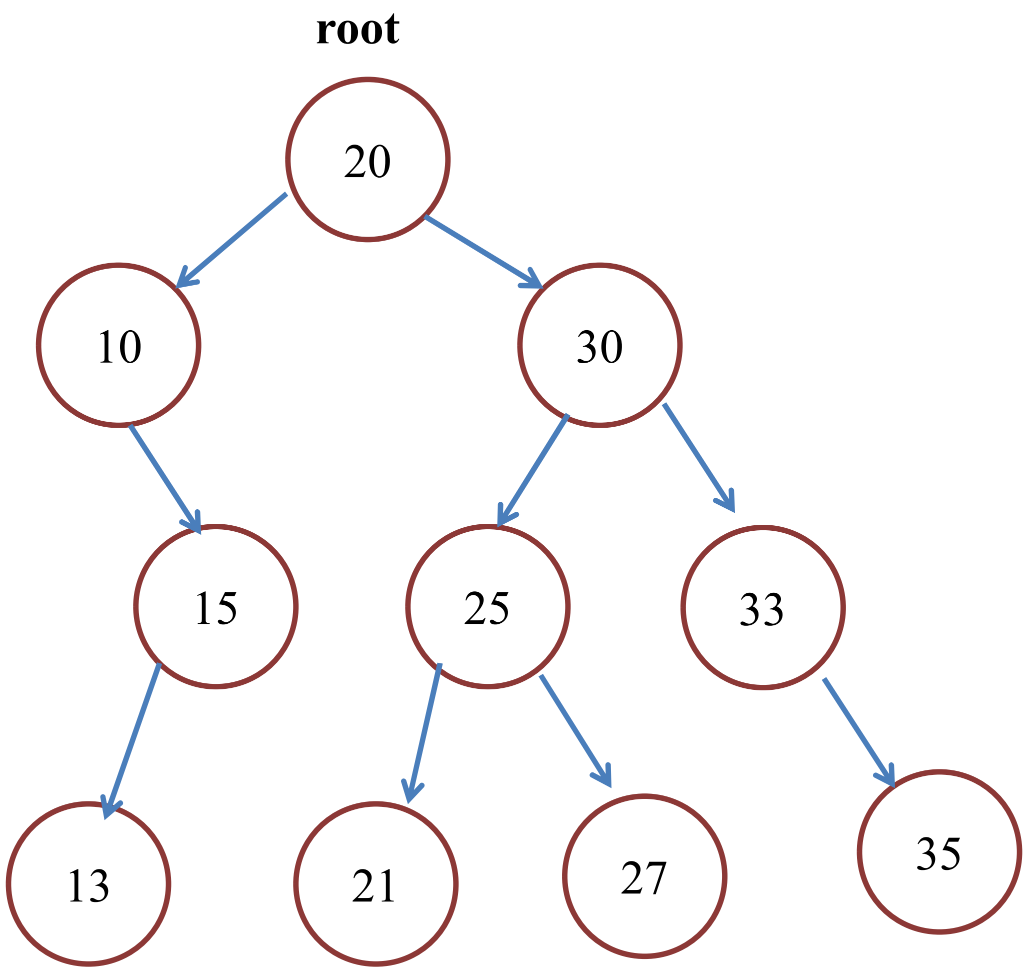
- Deletion from a BST is much more complicated than insertion or searching
  - We may need to delete a node which has children/subtrees
  - How do we delete the node and still maintain the BST property?
- Three cases
  - Deleting a leaf node – Simple, since there are no children
  - Deleting a node with one child
  - Deleting a node with two children

Deleting an Element from a BST

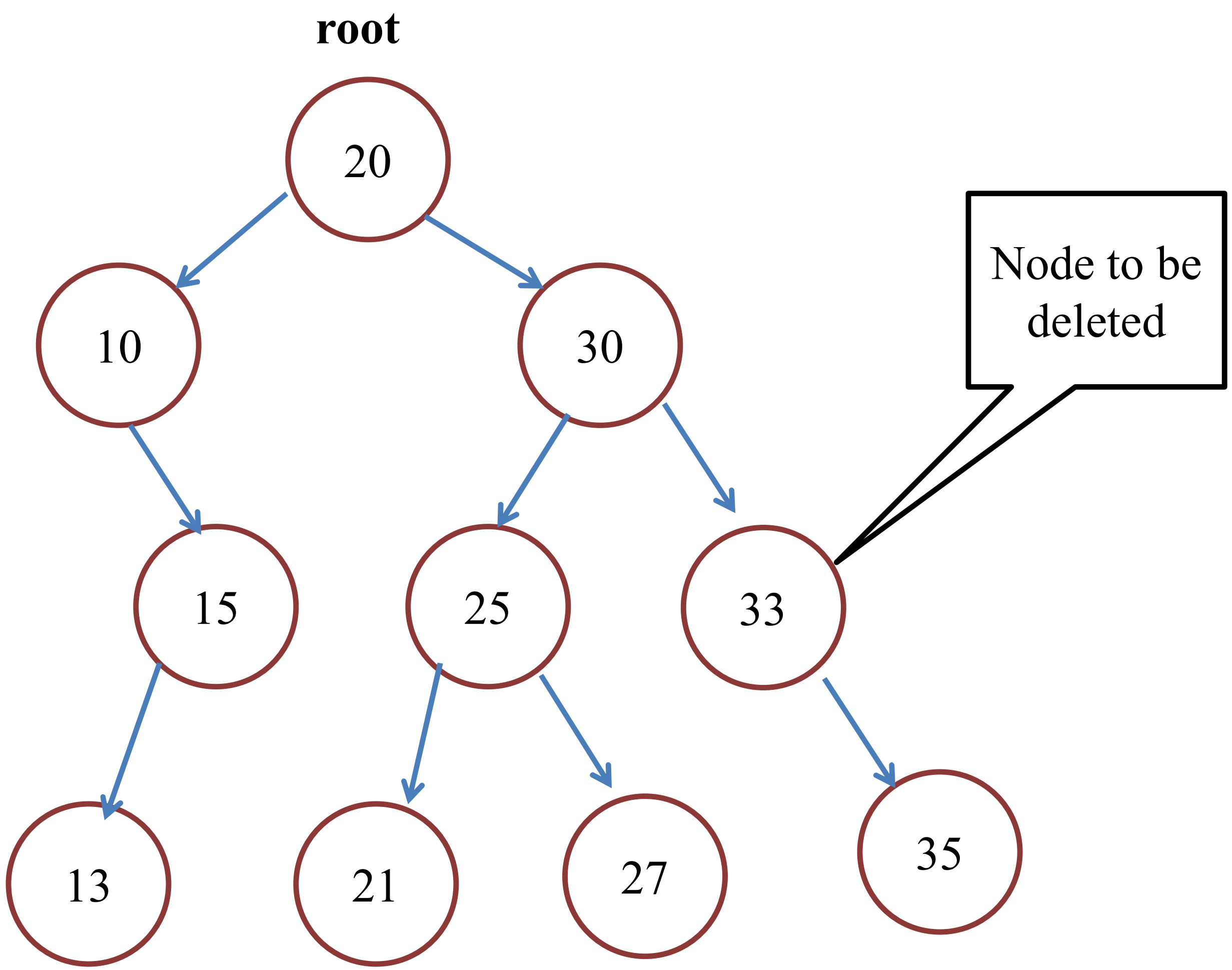


Deleting an Element from a BST

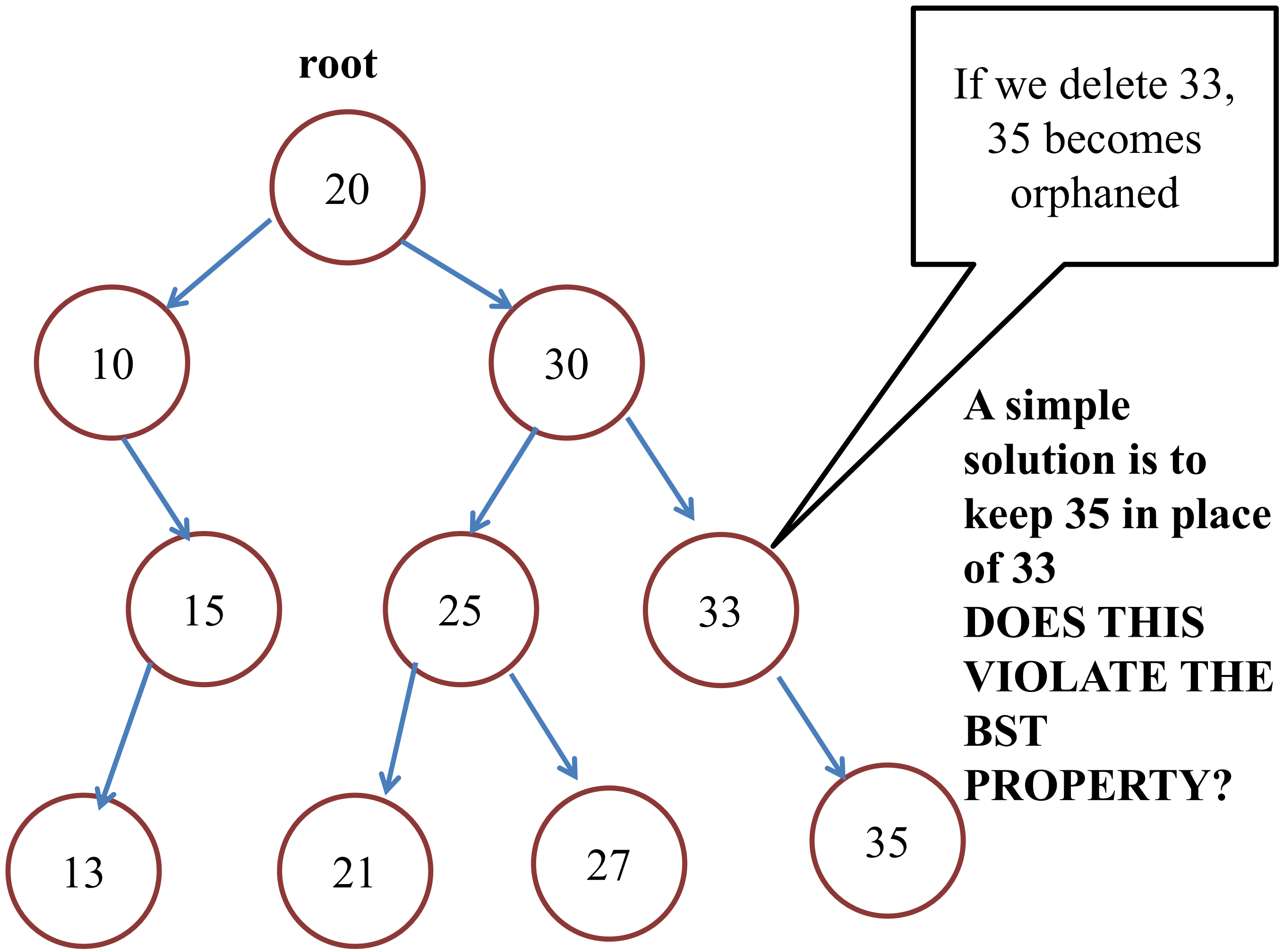
Simple!
Just set the parent's child pointer as NULL and delete the node



Deleting an Element from a BST

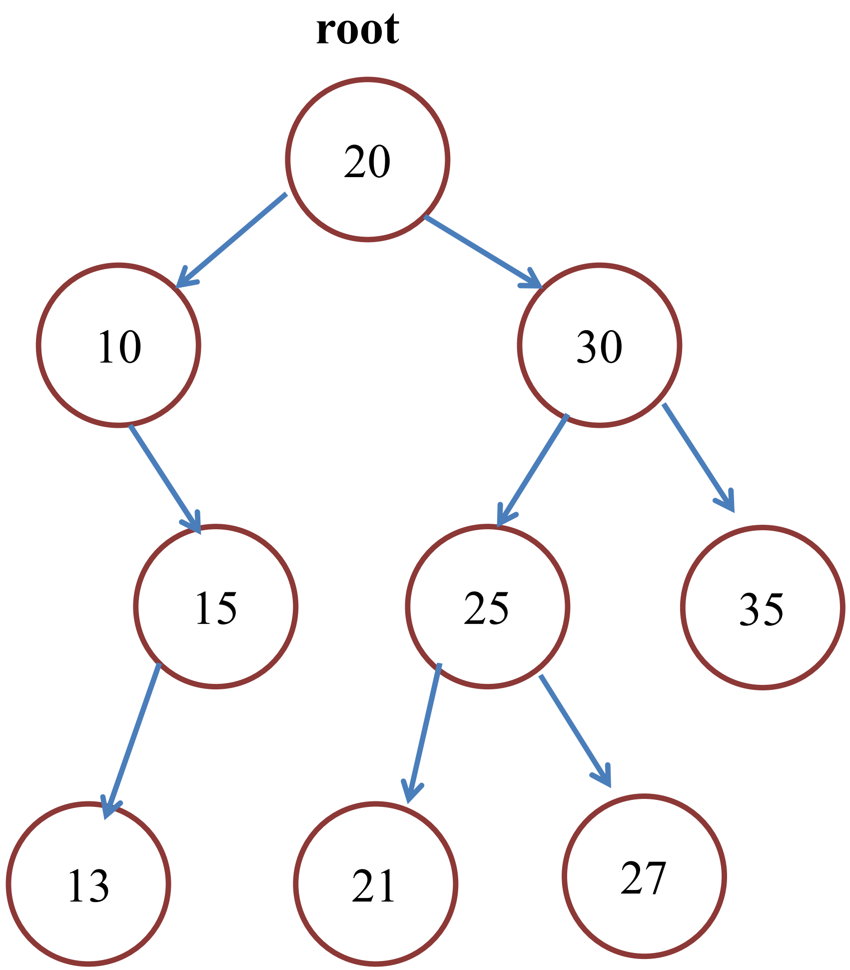


Deleting an Element from a BST



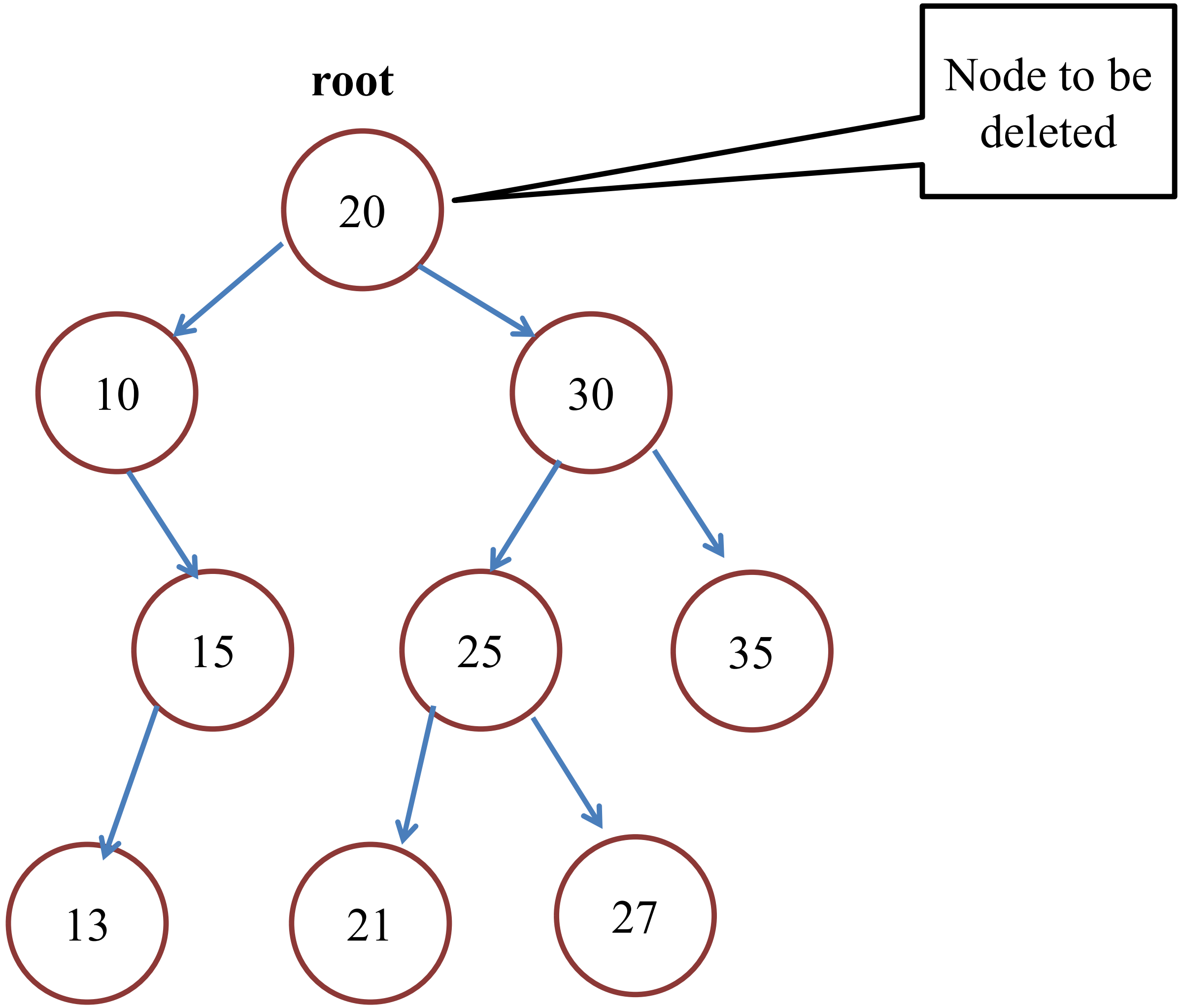
A simple solution is to keep 35 in place of 33
DOES THIS VIOLATE THE BST PROPERTY?

Deleting an Element from a BST

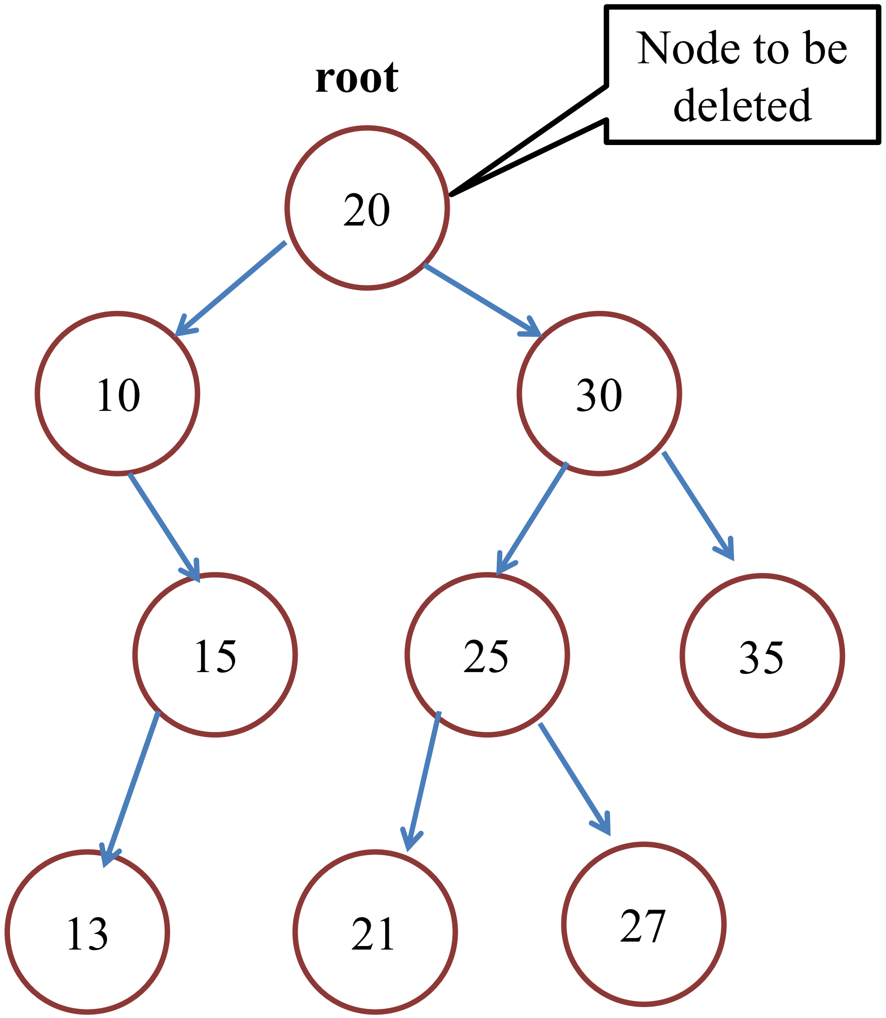


If a node only has one child, keep the child in place of the node and delete the node

Deleting an Element from a BST



Deleting an Element from a BST

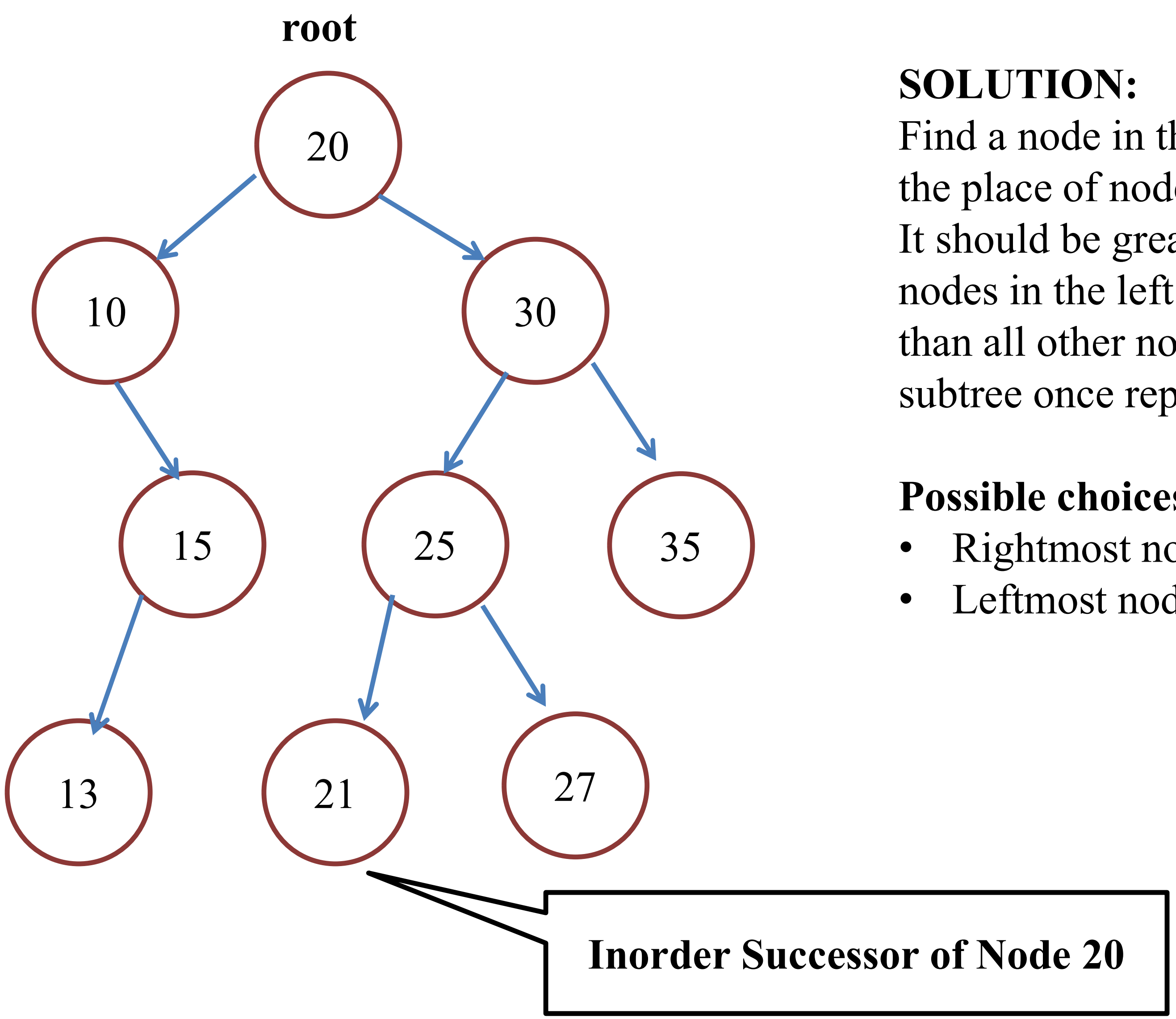


SOLUTION:
Find a node in the BST that can take the place of node 20
It should be greater than all other nodes in the left subtree and less than all other nodes in the right subtree once replaced

Possible choices:

- Rightmost node in the left subtree
- Leftmost node in the right subtree

Deleting an Element from a BST

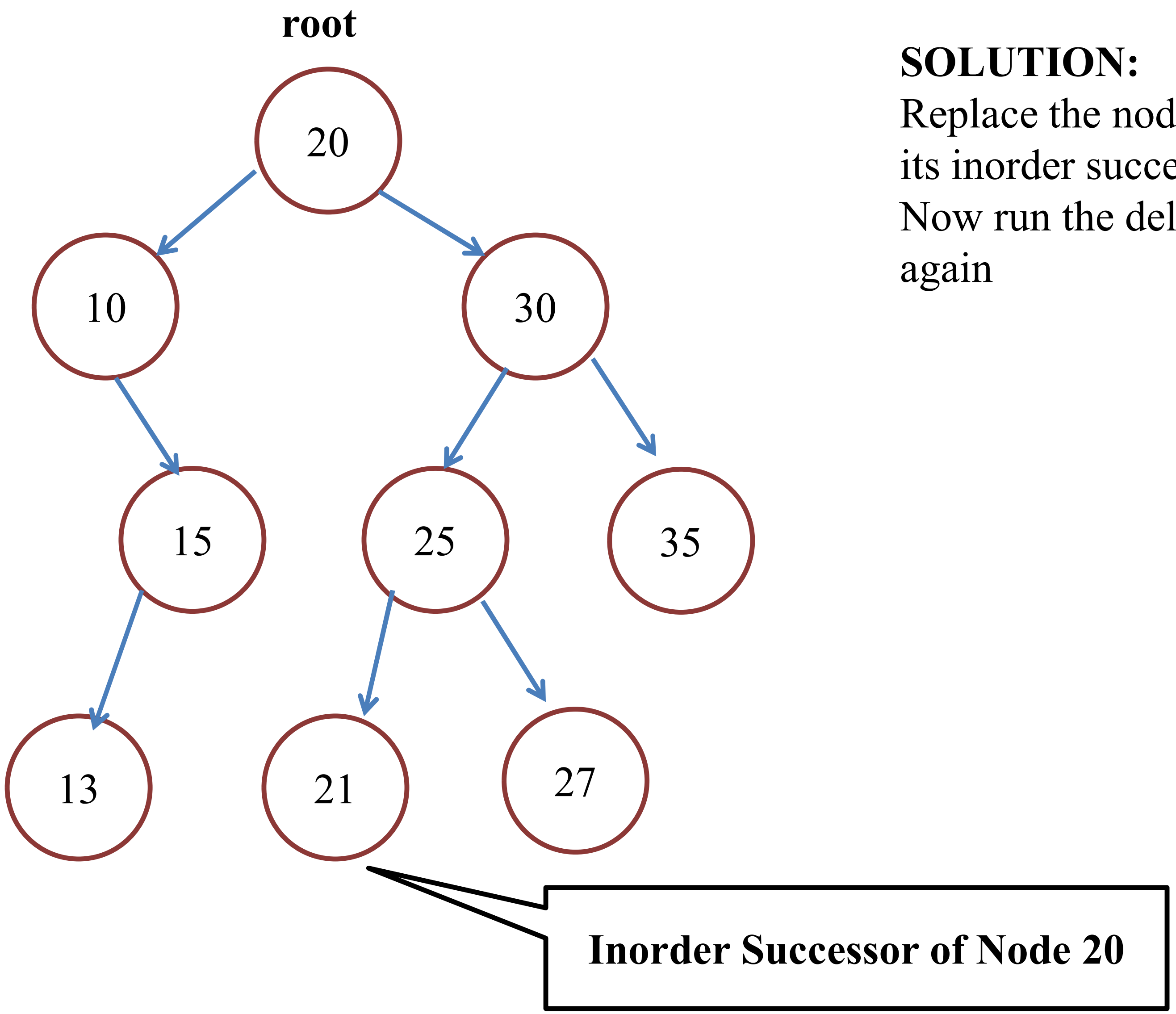


SOLUTION:
Find a node in the BST that can take the place of node 20
It should be greater than all other nodes in the left subtree and less than all other nodes in the right subtree once replaced

Possible choices:

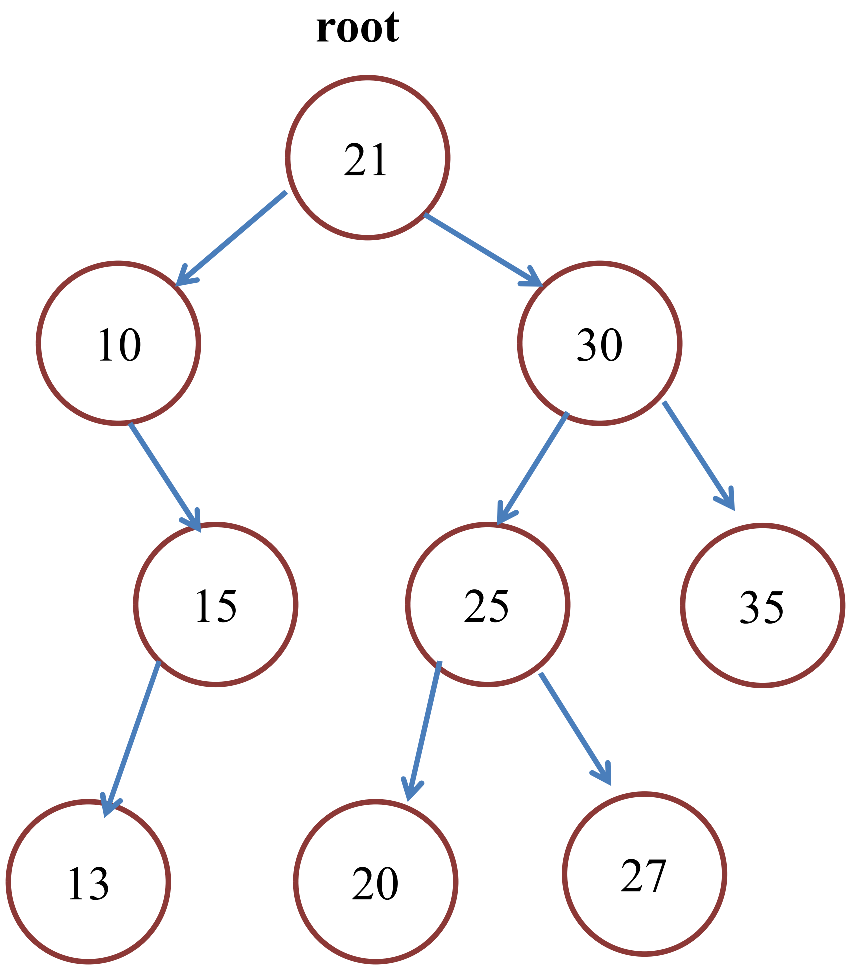
- Rightmost node in the left subtree
- Leftmost node in the right subtree

Deleting an Element from a BST



SOLUTION:
Replace the node to be deleted with its inorder successor
Now run the delete operation once again

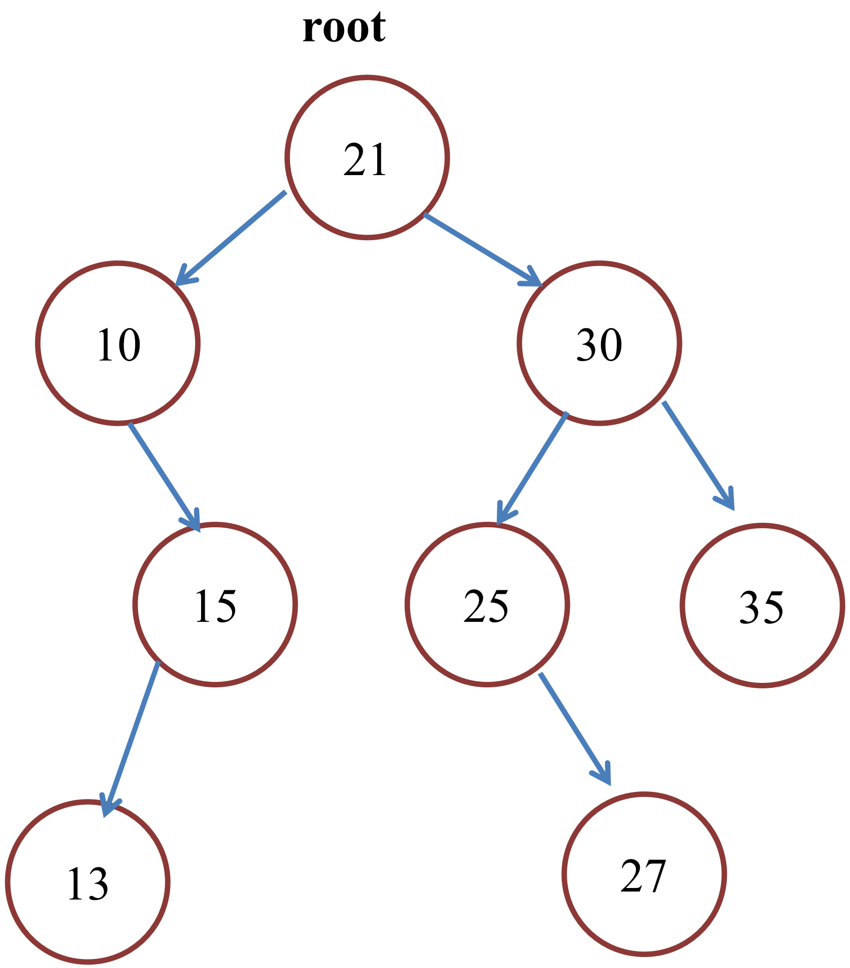
Deleting an Element from a BST



SOLUTION:
Replace the node to be deleted with its inorder successor
Now run the delete operation once again

Now 20 is a leaf node and can be deleted easily

Deleting an Element from a BST



Deleting an Element from a BST

```
DELETE_BST(root, key)
        prev = curr = root
        while curr !=NULL, do
                if curr→data==key
                        break
                else
                        prev=curr
                        if curr→data>key
                                curr=curr→lchild
                        else
                                curr=curr→rchild
        if curr==NULL
                print "Element to be deleted does not exist"
                exit
```

Deleting an Element from a BST

DELETE_BST(root, key) *//Continued*

```
if curr→lchild==NULL && curr→rchild==NULL
    if curr==prev→lchild
        prev→lchild=NULL
    else
        prev→rchild=NULL
    free(curr)
```

Deleting an Element from a BST

```
DELETE_BST(root, key)          //Continued
    else if curr→lchild==NULL || curr→rchild==NULL
        if curr→lchild==NULL
            if curr==prev→lchild
                prev→lchild=curr→rchild
            else
                prev→rchild=curr→rchild
            free(curr)
        if curr→rchild==NULL
            if curr==prev→lchild
                prev→lchild=curr→lchild
            else
                prev→rchild=curr→lchild
            free(curr)
```

Deleting an Element from a BST

DELETE_BST(root, key) *//Continued*

    else

        ptr=INORDER_SUCCESSOR(curr)

        SWAP*(curr→data, ptr→data)*

        DELETE_BST*(root, key)*

EXERCISE: What is the complexity of DELETE_BST?

Threaded Binary Trees

- A threaded binary tree is a modification of a binary tree to allow in-order traversal without using recursion or stack
- Two types of threaded binary trees:
  - ***Single Threaded:*** Where a NULL right pointers is made to point to the inorder successor (if successor exists)
  - ***Double Threaded:*** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

Threaded Binary Trees