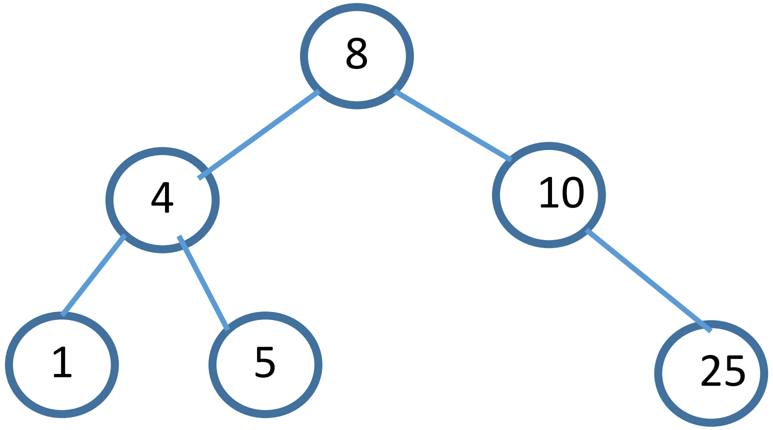
# COMP 2611, Data Structures

## LECTURE 9: BINARY SEARCH TREES

# Binary Search Trees

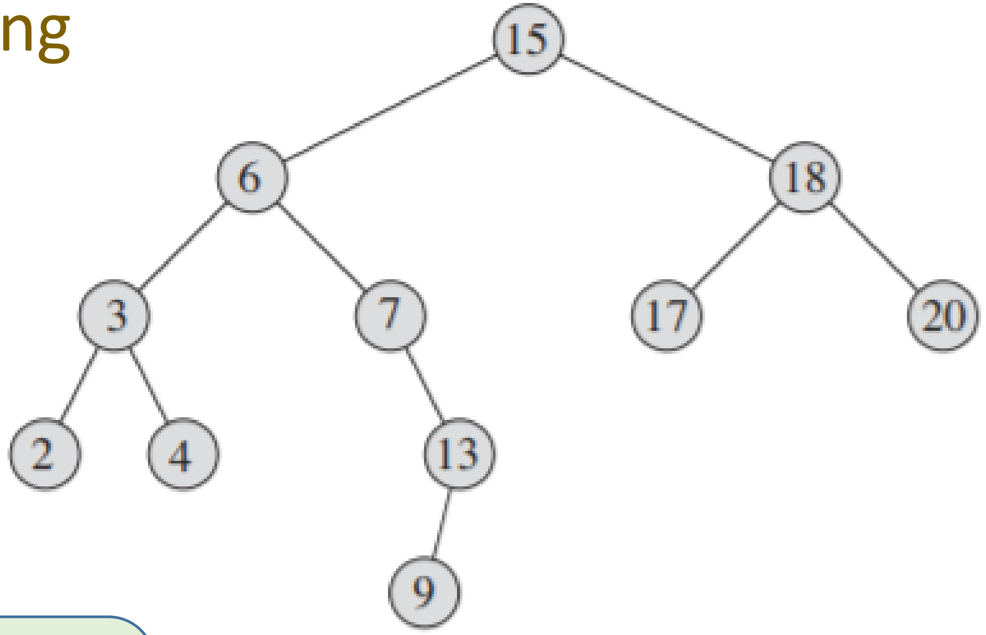
- A binary search tree (BST) is a binary tree where the keys stored at each node satisfy the *binary-search-tree property*:

  - Let *x* be a node in a BST.
  - If *y* is a node in the left subtree of *x*, then y.key ≤ x.key.
  - If *y* is a node in the right subtree of *x*, then y.key ≥ x.key.



# Binary Search Trees: Inorder Traversal

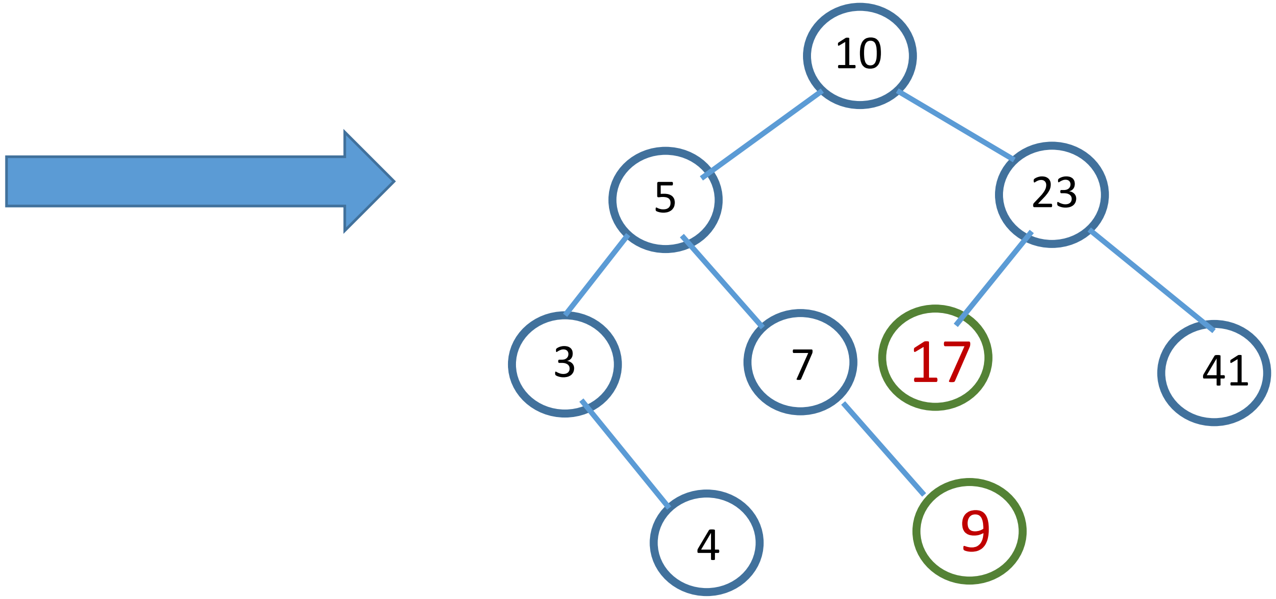
- An inorder traversal of a binary search tree always results in the nodes being visited in ascending order:



**2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20**

# Binary Search Trees: Insertion

- Where to insert 9 and 17?



- Insert 9 as the right child of 7.

- Insert 17 as the left child of 23.

# Binary Search Trees: Insertion

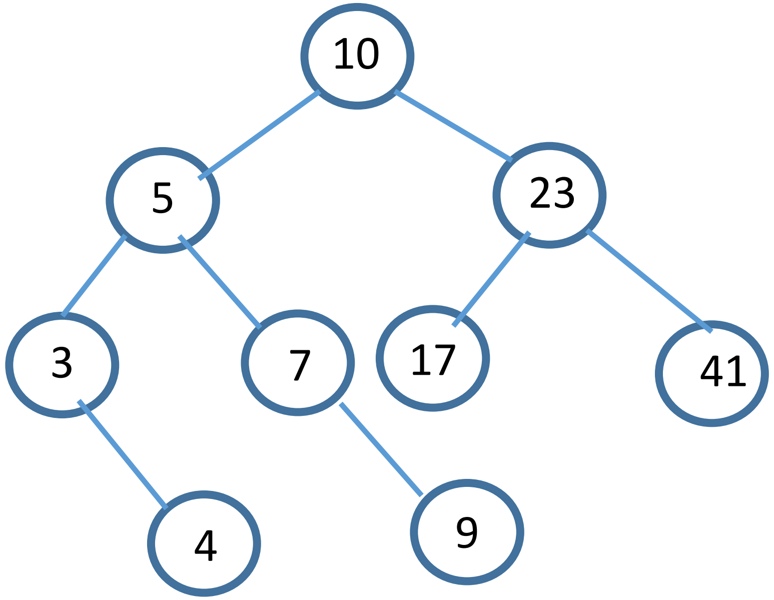
- Case 0: Tree is empty – insert node in empty tree (root is the address of this node)

- Case 1: If data < root->data – go left

- Case 2: If data > root->data – go right

- Repeat (1) and (2) until position is found (i.e., parent with empty left or right subtree

- Create BTNode and connect to parent.



# Binary Search Trees: Search (Return Node Where Found)

Cases for contains (BTNode * root, int key):

- The root of binary tree is empty:

  ⏚ (Empty tree)

  return NULL (or root)

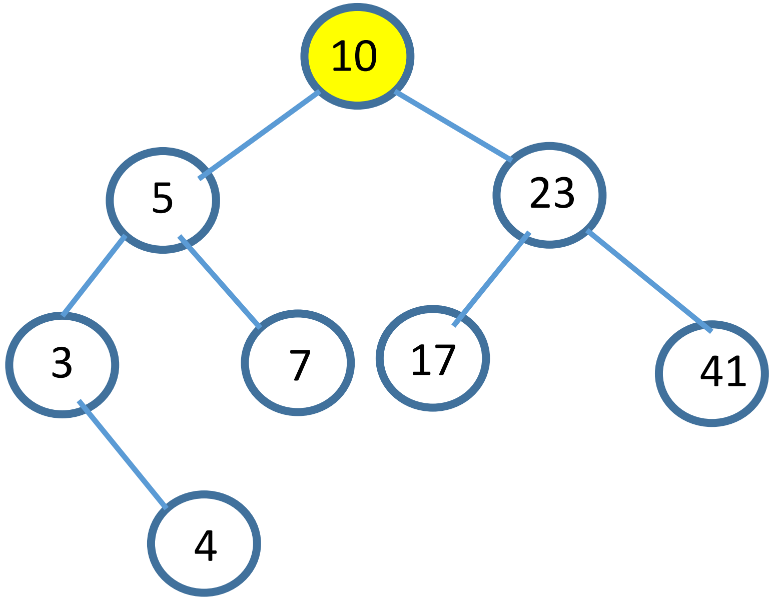
# Binary Search Trees: Search (Return Node Where Found)

Cases for contains (BTNode * root, int key):

- The root of binary tree is empty:
  return NULL (or root)

- The root of the binary tree contains *key*:

  return root



# Binary Search Trees: Search (Return Node Where Found)

Cases for contains (BTNode * root, int key):

- The root of binary tree is empty:

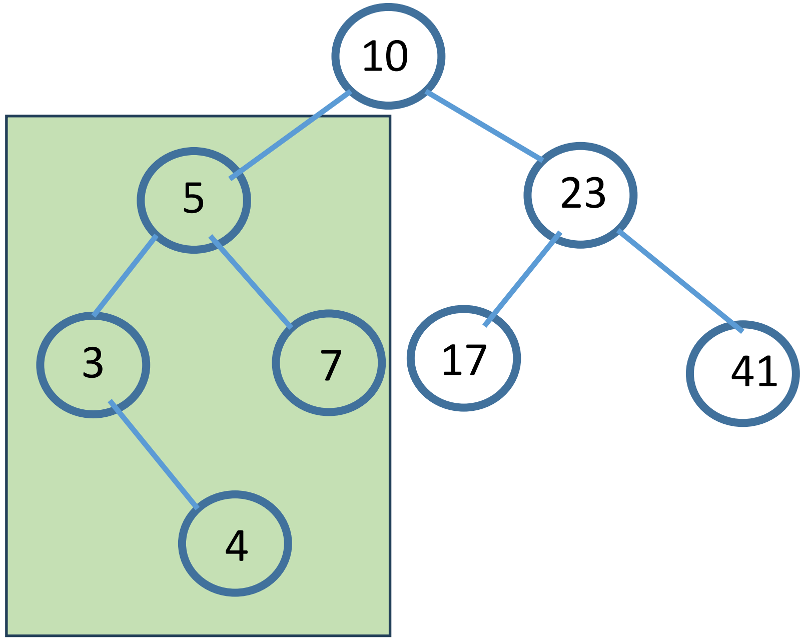
  return NULL (or root)

- The root of the binary tree contains *key*:

  return root

- If key < root->data:

  return contains(root->left, key)



# Binary Search Trees: Search (Return Node Where Found)

Cases for contains (BTNode * root, int key):

- The root of binary tree is empty:

  return NULL (or root)

- The root of the binary tree contains *key*:

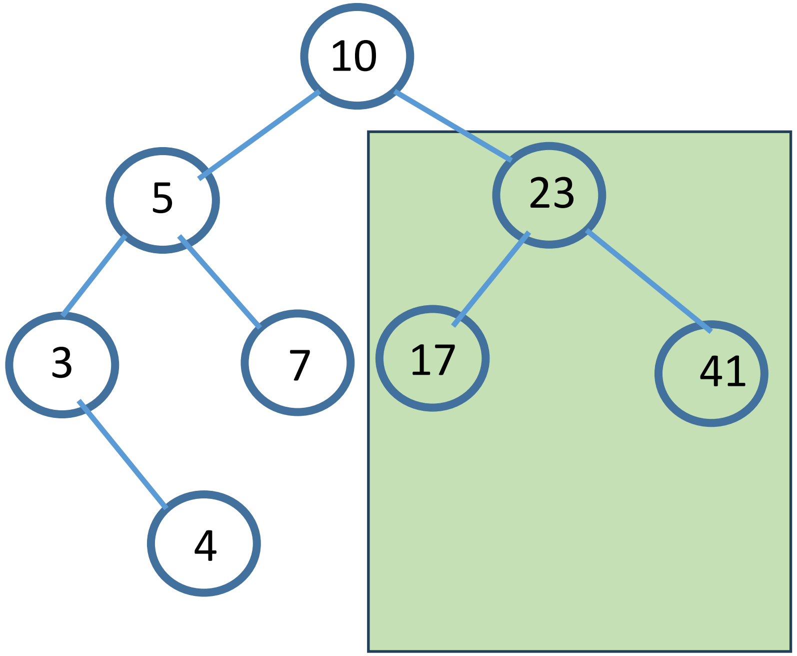
  return root

- If key < root->data:

  return contains(root->left, key)

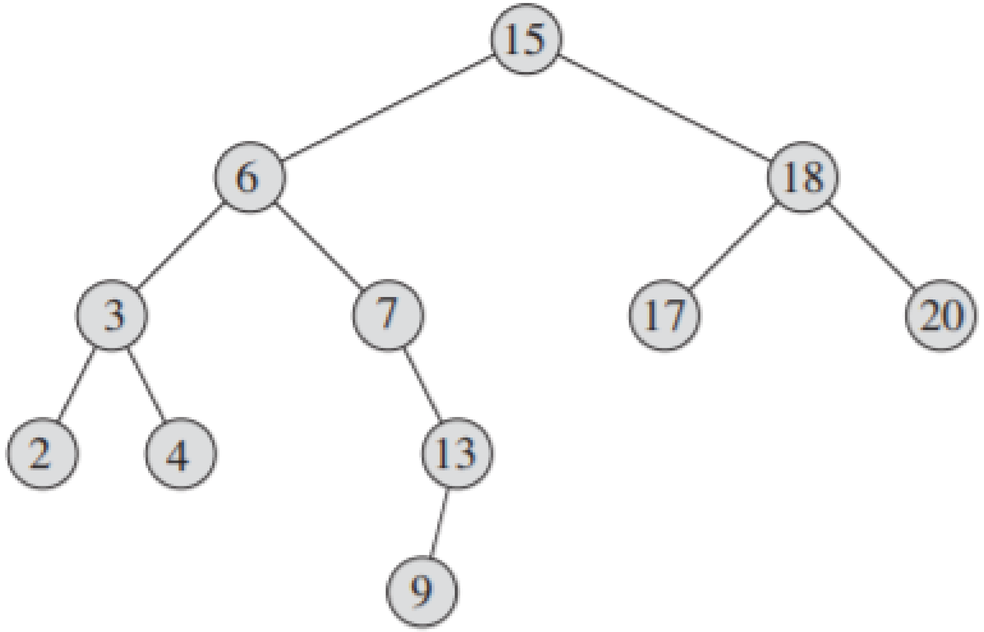
- If key > root->data:

  return contains(root->right, key)



# Binary Search Trees: Finding the Nodes with the Minimum and Maximum Keys

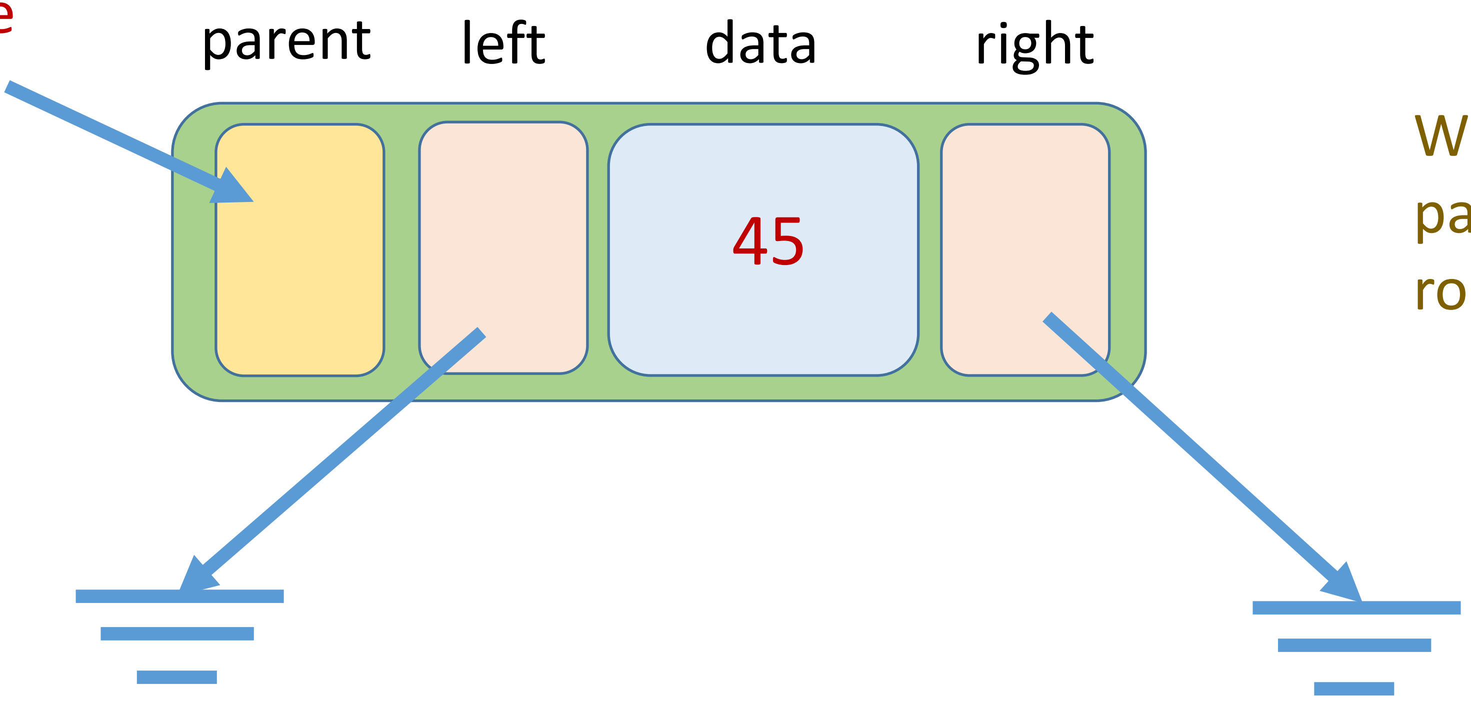
- `BTNode * minimum (BTNode * root)`
- `BTNode * maximum (BTNode * root)`



Question 2 of Lab #4

# A Node in a Binary Tree: Using Parent Pointers

Add a parent field in the struct:



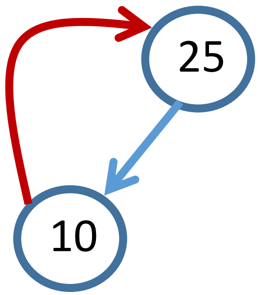
*parent* field contains the address of the node’s parent

What is the parent of the root of the BST?

# Changes To Code

In BinaryTree.h

```
struct BTNode {
    int data;
    BTNode * left;
    BTNode * right;
    BTNode * parent;
};
```



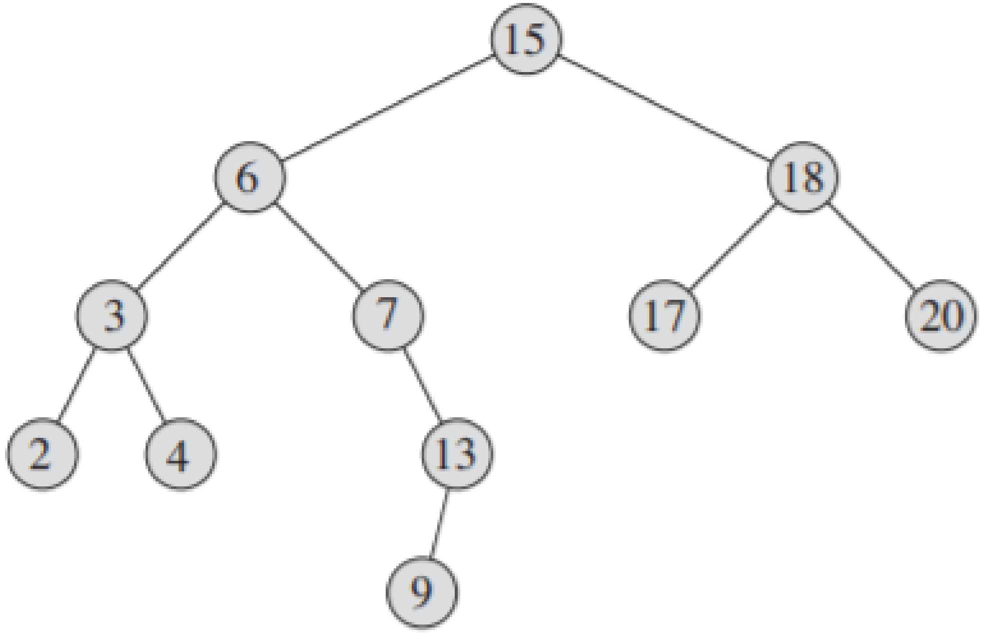
```
BTNode * node1;
BTNode * node2;

node1 = createBTNode (25);
node2 = createBTNode (10);

node1->left = node2;
node2->parent = node1;
```

In BinaryTree.cpp

```
BTNode * createBTNode (int data) {
    BTNode * newNode;

    newNode = new BTNode;

    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->parent = NULL;

    return newNode;
```

# Binary Search Trees: Find Depth of a Node

- The *depth* of a node is the number of branches that must be traversed on the path to the node from the root.
- The depth of 13 is 3; the depth of 17 is 2.

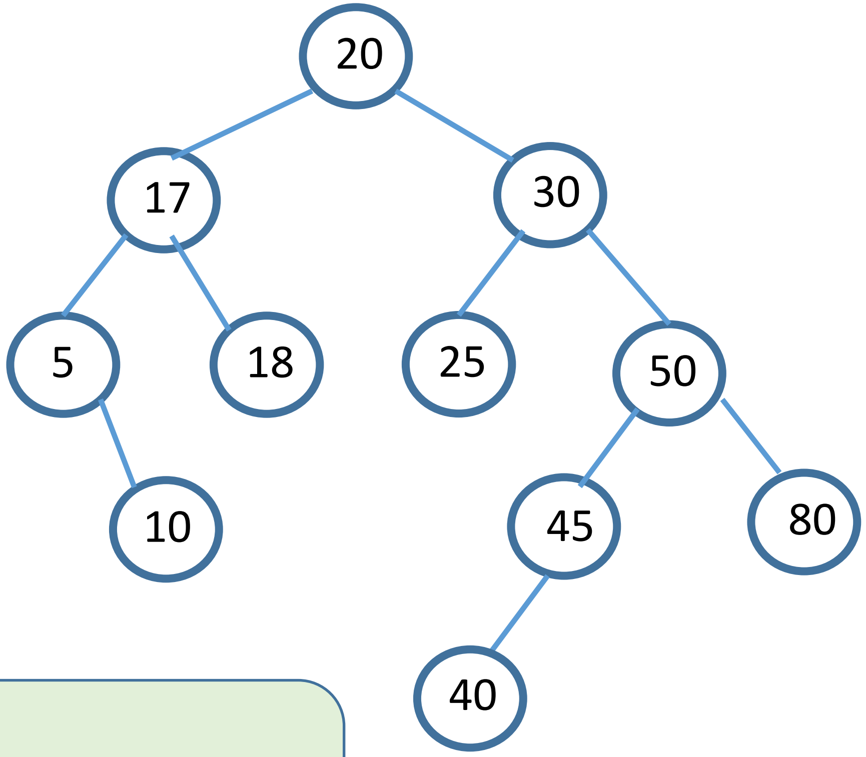


Function to find depth:

```
int nodeDepth (BTNode * node) {
    int depth = 0;

    while (node->parent != NULL) {
        node = node->parent;
        depth++;
    }
    return depth;
}
```

# Inorder Successor

- Give the inorder traversal of the following BST:

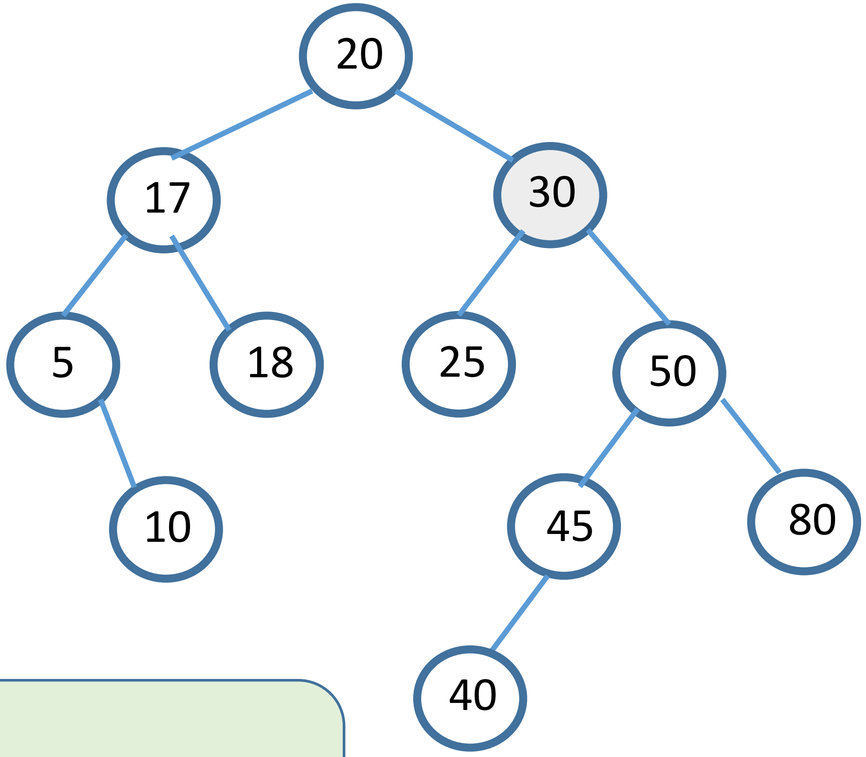


What's the inorder successor of 30?

**5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80**

# Inorder Successor

- Give the inorder traversal of the following BST:

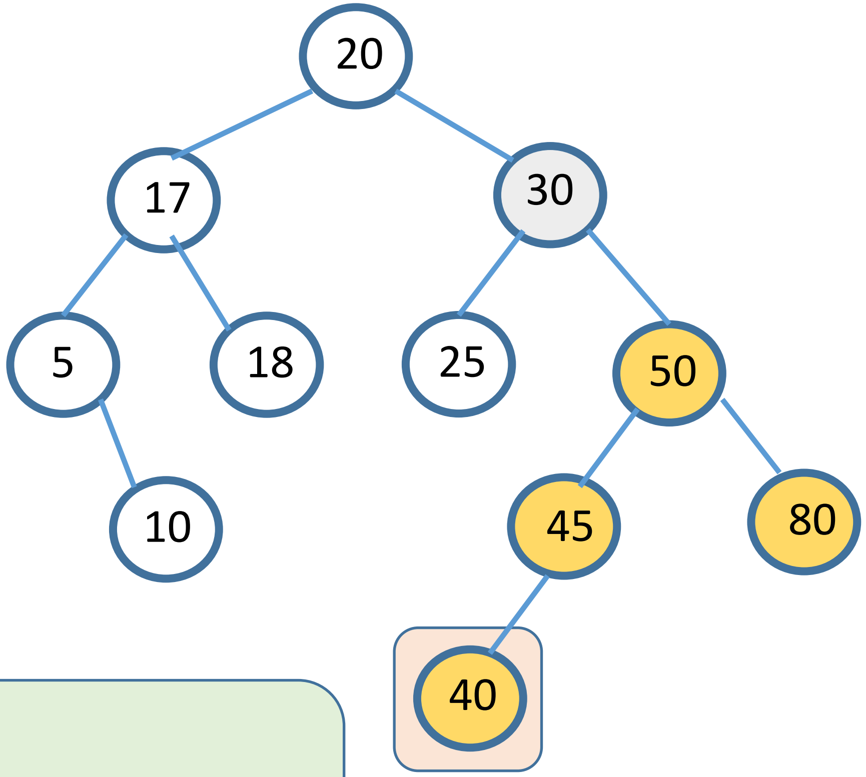


What's the inorder successor of 30?

**5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80**

# Inorder Successor

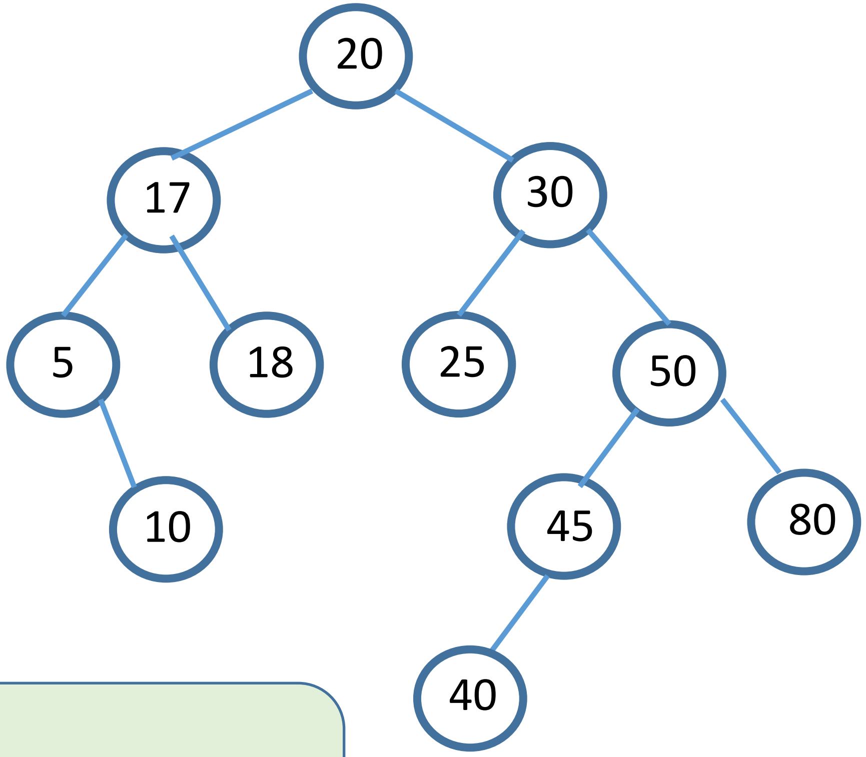
- Give the inorder traversal of the following BST:



How to find the inorder successor of 30?

**5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80**

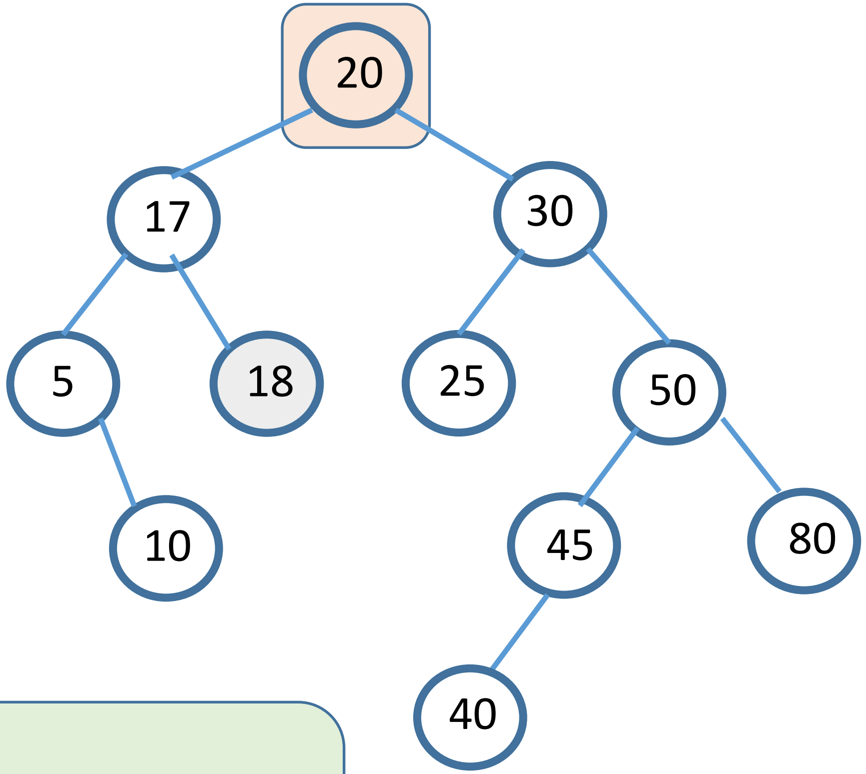
# Inorder Successor



What's the inorder successor of 18?

**5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80**

# Inorder Successor



What's the inorder successor of 18?

**5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80**

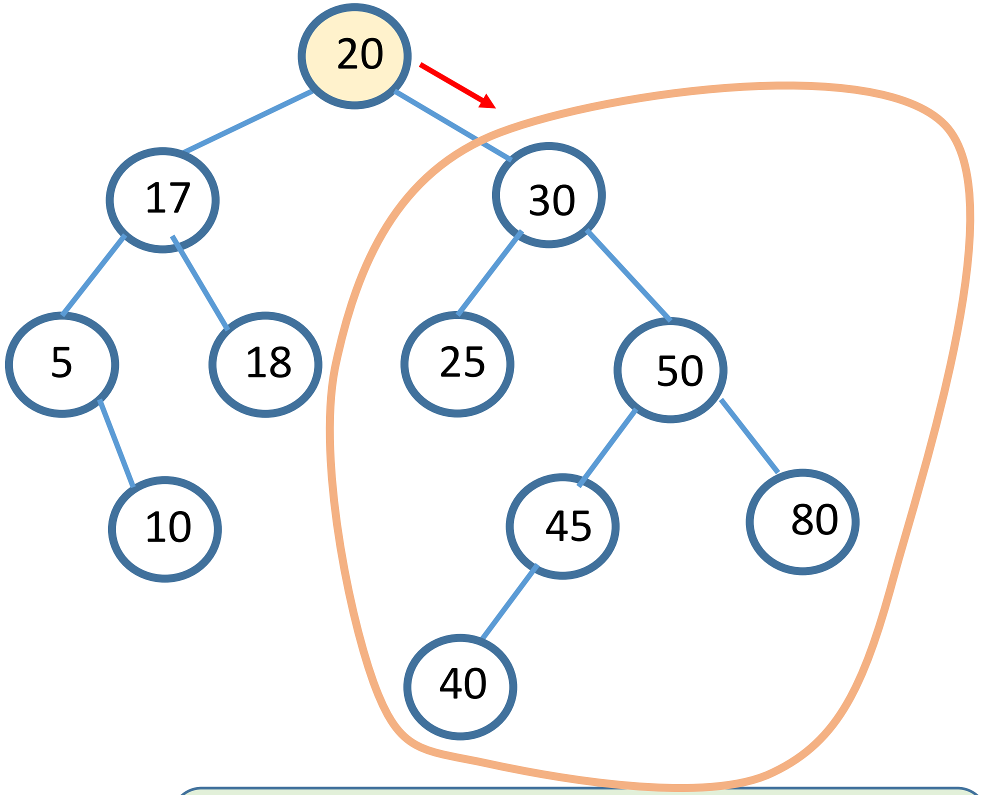
# Finding the Inorder Successor

## Case 1:

- Node has a non-empty right subtree (e.g., 5, 17, 20, 30, 50)

- The inorder successor is the first node in an inorder traversal of the right subtree. How to find this node?

```
int inOrderSuccessor (BTNode * node) {

       if (node->right != NULL) {
             return minimum (node->right);
       }
       …
}
```



**5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80**

# Finding the Inorder Successor

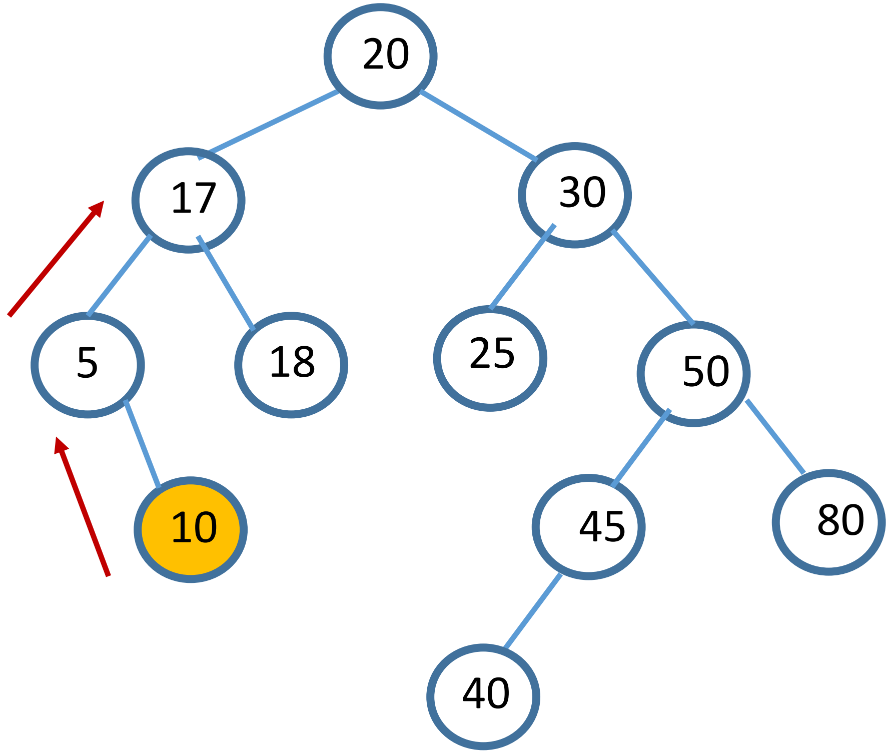
Case 2:

- Node has an empty right subtree e.g., 10, 18, 25, 45, 80)

- The inorder successor is one of its ancestors. Which one?

- Suppose that *x* has a successor *y*. Then, *y* is the lowest ancestor of *x* whose left child is also an ancestor of *x*.

- To find *y*, we go up the tree from *x* until we encounter a node that is the left child of its parent. If no such node is encountered, there is no successor.



Inorder successor of:

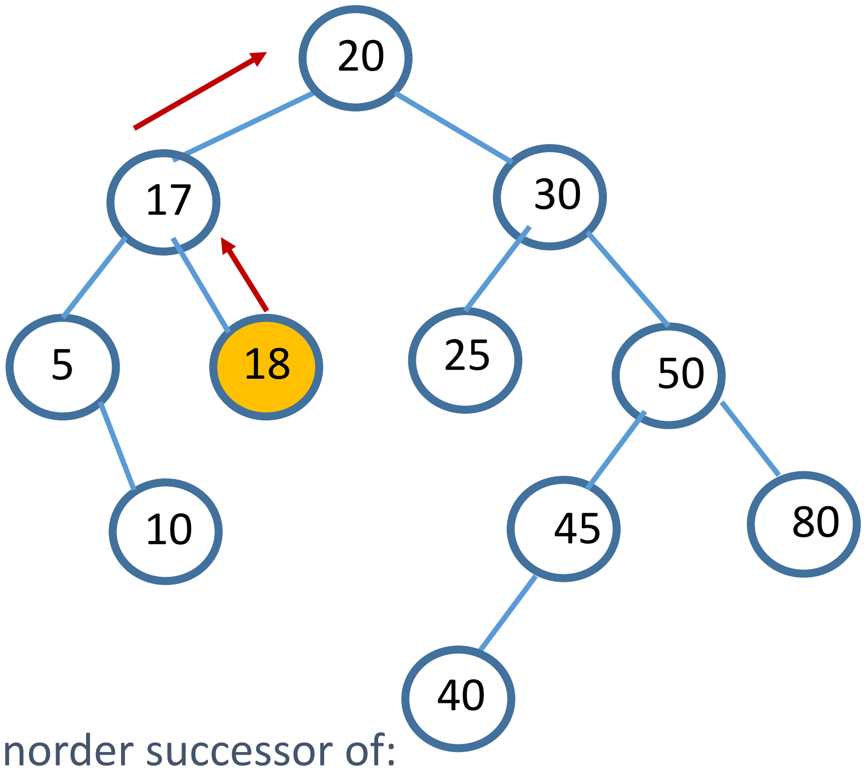
- 10 is 17

**5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80**

# Finding the Inorder Successor

Case 2:

- Node has an empty right subtree e.g., 10, 18, 25, 45, 80)
- The inorder successor is one of its ancestors. Which one?
- Suppose that *x* has a successor *y*. Then, *y* is the lowest ancestor of *x* whose left child is also an ancestor of *x*.
- To find *y*, we go up the tree from *x* until we encounter a node that is the left child of its parent. If no such node is encountered, there is no successor.



Inorder successor of:

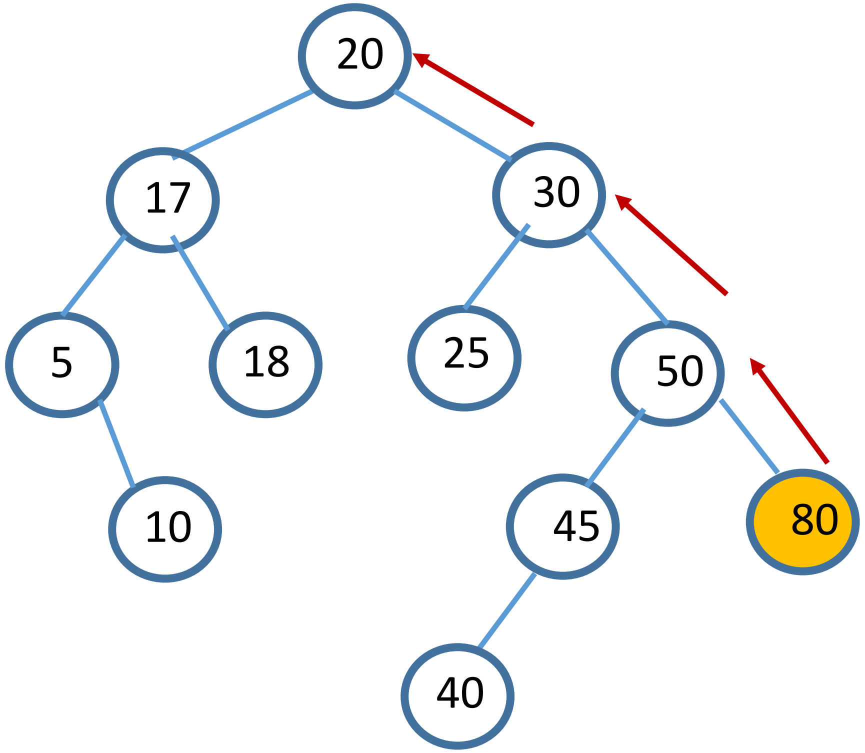
- 10 is 17
- 18 is 20

**5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80**

# Finding the Inorder Successor

Case 2:

- Node has an empty right subtree e.g., 10, 18, 25, 45, 80)
  - The inorder successor is one of its ancestors. Which one?
  - Suppose that *x* has a successor *y*. Then, *y* is the lowest ancestor of *x* whose left child is also an ancestor of *x*.
  - To find *y*, we go up the tree from *x* until we encounter a node that is the left child of its parent. If no such node is encountered, there is no successor.



Inorder successor of:

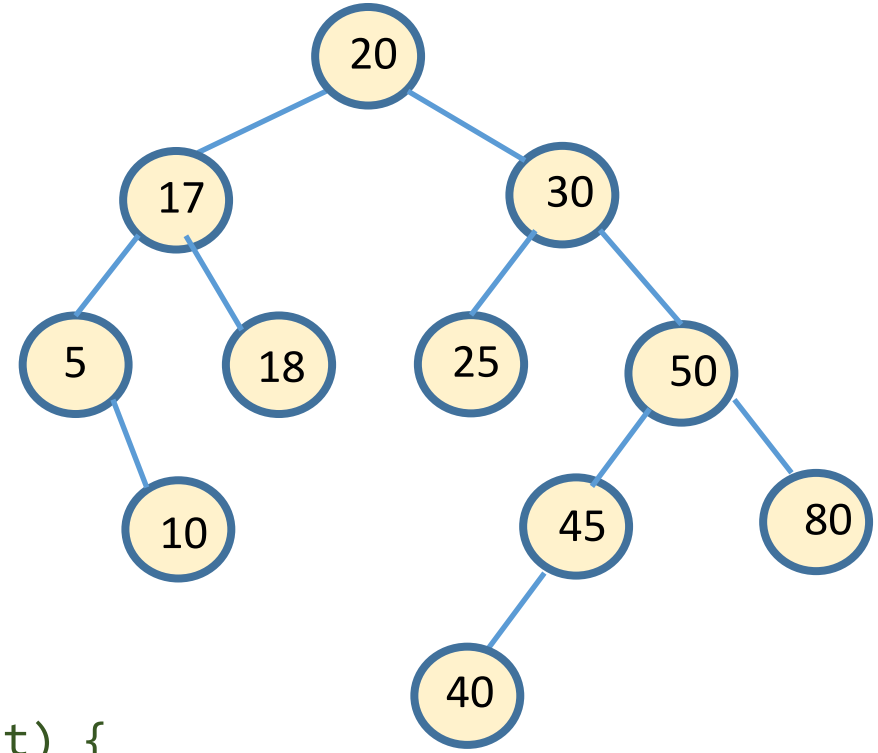
- 10 is 17
- 18 is 20
- 80 is **???**

**5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80**

# Code to Find the Inorder Successor

## Case 1 and Case 2:

```
BTNode * inOrderSuccessor (BTNode * node) {
       if (node == NULL)
              return NULL;

       if (node->right != NULL)
              return minimum (node->right);

       BTNode * parent;

       parent = node->parent;
       while (parent != NULL && node == parent->right) {
              node = parent;
              parent = parent->parent;
       }
       return parent;
}
```



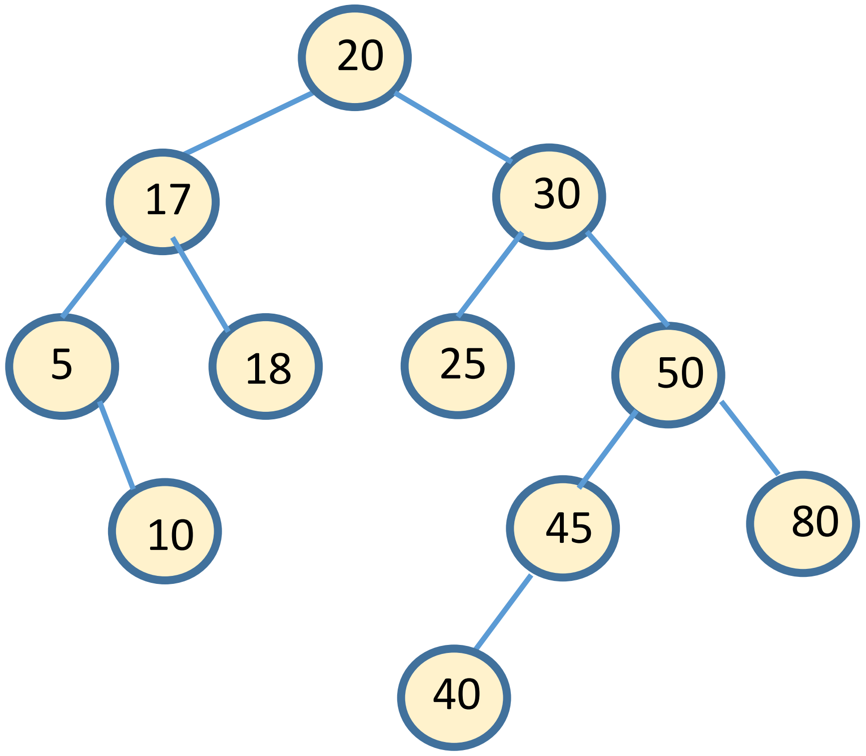
5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80

# Another Version of Inorder Traversal

- You are given two functions:

```
BTNode * minimum (BTNode * root)

BTNode * inorderSuccessor (BTNode * node)
```

- Perform an inorder traversal using only these two functions.



5, 10, 17, 18, 20, 25, 30, 40, 45, 50, 80