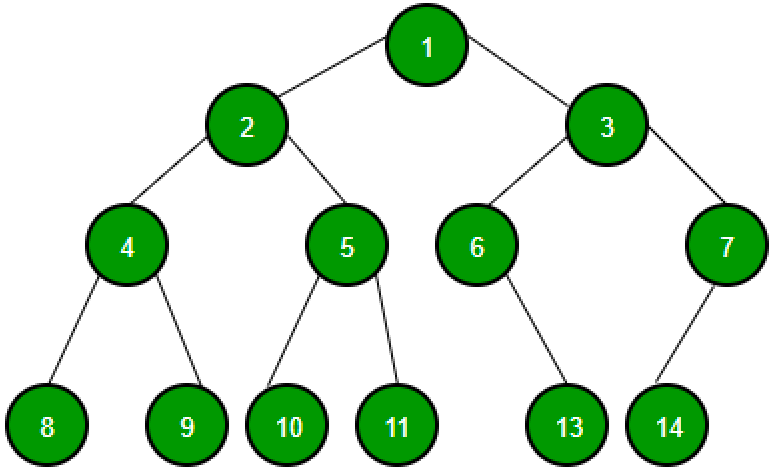
Data Structure & Algorithms

BST (Binary Search Tree)

Binary Tree

A binary tree is a linked data structure in which each node is an object that contains following attributes:

- a key and satellite data
- left, pointing to its left child
- right, pointing to its right child
- p, pointing to its parent



Binary Tree

- Each node in the binary tree is termed as either a parent node or as a child node.
- The topmost node of the Binary Tree is referred to as the root node. Each parent node can have at most 2 child nodes which are the left child node and the right child node.
- A binary tree is a recursive structure
- Particular kinds of nodes:
  - Root
  - Leaves

Binary Tree - Definition

```
struct node {
    int data;
    struct node* left;
    struct node* right;
};
```

Maximum Depth or Height of a Tree

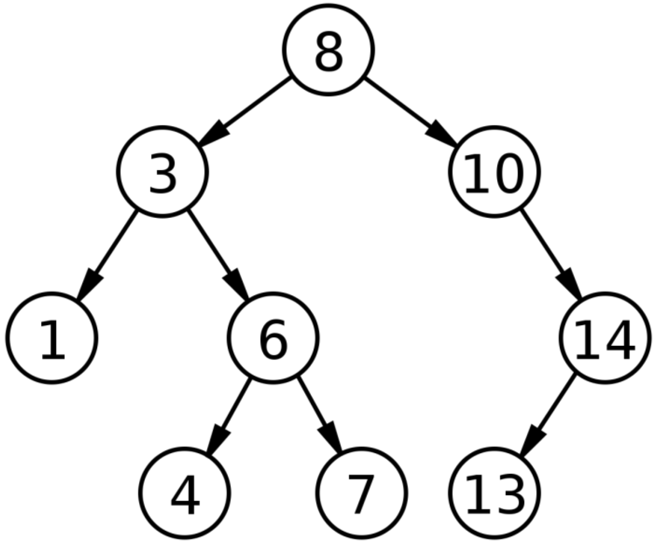
- Height of a tree: longest path from the root to one of the leaves; max(heights of subtrees) + 1

```
int maxDepth(struct node* node)
{
    if (node == NULL)
        return 0;
    else {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        /* use the larger one */
        if (lDepth > rDepth)
            return (lDepth + 1);
        else
            return (rDepth + 1);
    }
}
```

Binary Search Tree

A binary tree is a **linked** data structure in which **each node** is an object that contains following **attributes**:

- a key and satellite data
- left, pointing to its left child
- right, pointing to its right child
- p, pointing to its parent



A binary tree is a recursive structure

Particular kinds of nodes:

- Root
- Leaves

Height of a tree: longest path from the root to one of the leaves; max(heights of sub-trees) + 1

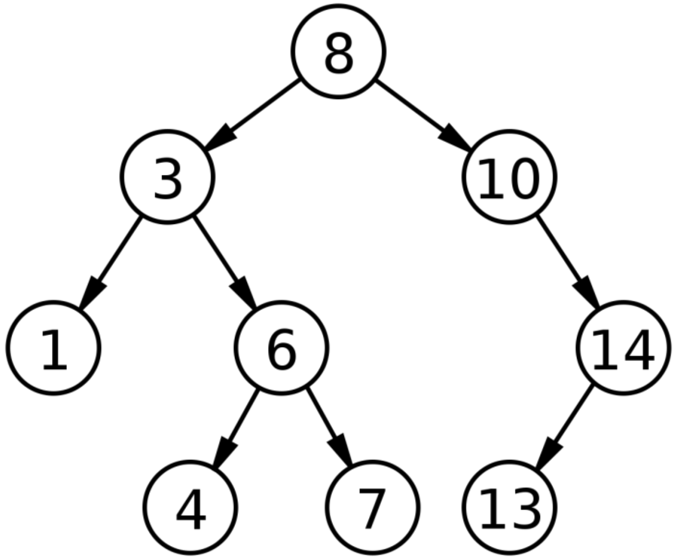
Binary Search Tree cont.

Binary-search-tree property

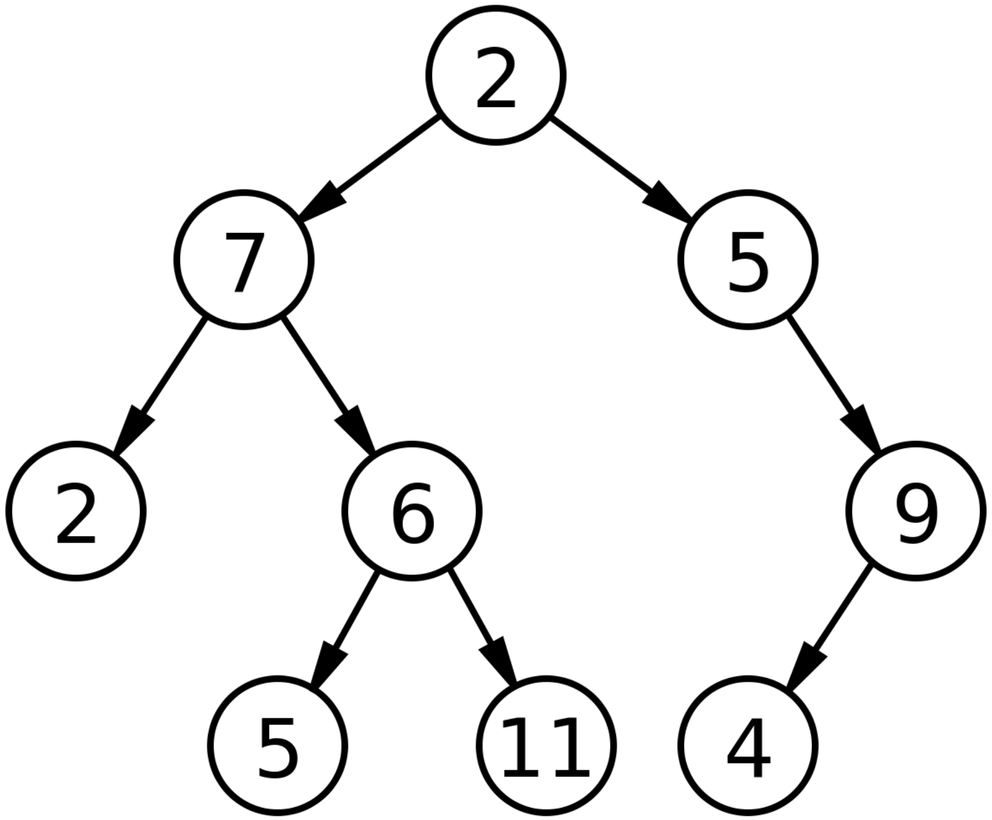
For any node x,

$Key[\,y\,] \;\leq\; Key[x]$ if y in left sub-tree of x

$Key[\,x\,] \;\leq\; Key[\,y]$ if y in right sub-tree of x



Counterexample – NOT a BST !

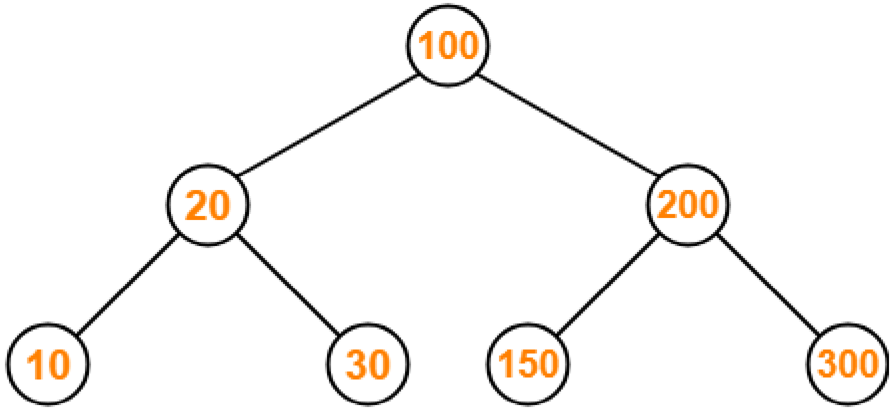


Tree Walks (Traversals)

- The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an in-order tree walk.
- This algorithm is so named because it prints the key of the root in between printing the values in its left sub-tree and printing those in its right subtree.

- Post-order: print sub-trees first, then print root
- Pre-order: print root first, then print sub-trees

Example – Tree walks



<u>Preorder Traversal-</u>

100 , 20 , 10 , 30 , 200 , 150 , 300

<u>Inorder Traversal-</u>

10 , 20 , 30 , 100 , 150 , 200 , 300

<u>Postorder Traversal-</u>

10 , 30 , 20 , 150 , 300 , 200 , 100

In-order traversal

```
void inorder(struct node* root){
    if(root == NULL) return;
    //recursively traverse left subtree first
    inorder(root->leftChild);
    //traverse current node
    printf("%d ", root-data);
    // recursively traverse right subtree lastly
    inorder(root->rightChild);
}
```

In-order – Time Complexity

$$T(n) = T(n_1) + T(n_2) + \theta(1)$$

$n_1 = number\ of\ left\ subtree's\ nodes$

$n_2 = number\ of\ left\ subtree's\ nodes$

$For\ the\ root$

$T(n_1) = T(n_{11}) + T(n_{12}) + \theta(1)$

.
.
.

$T(n_2) = T(n_{21}) + T(n_{22}) + \theta(1)$

.
.
.

In-order – Time Complexity cont.

$$T(n) = T(n_1) + T(n_2) + \theta(1)$$

$n_1 = number\ of\ left\ subtree's\ nodes$

$n_2 = number\ of\ left\ subtree's\ nodes$

$For\ the\ root$

$T(n_1) = T(n_{11}) + T(n_{12}) + \theta(1)$

.
.
.

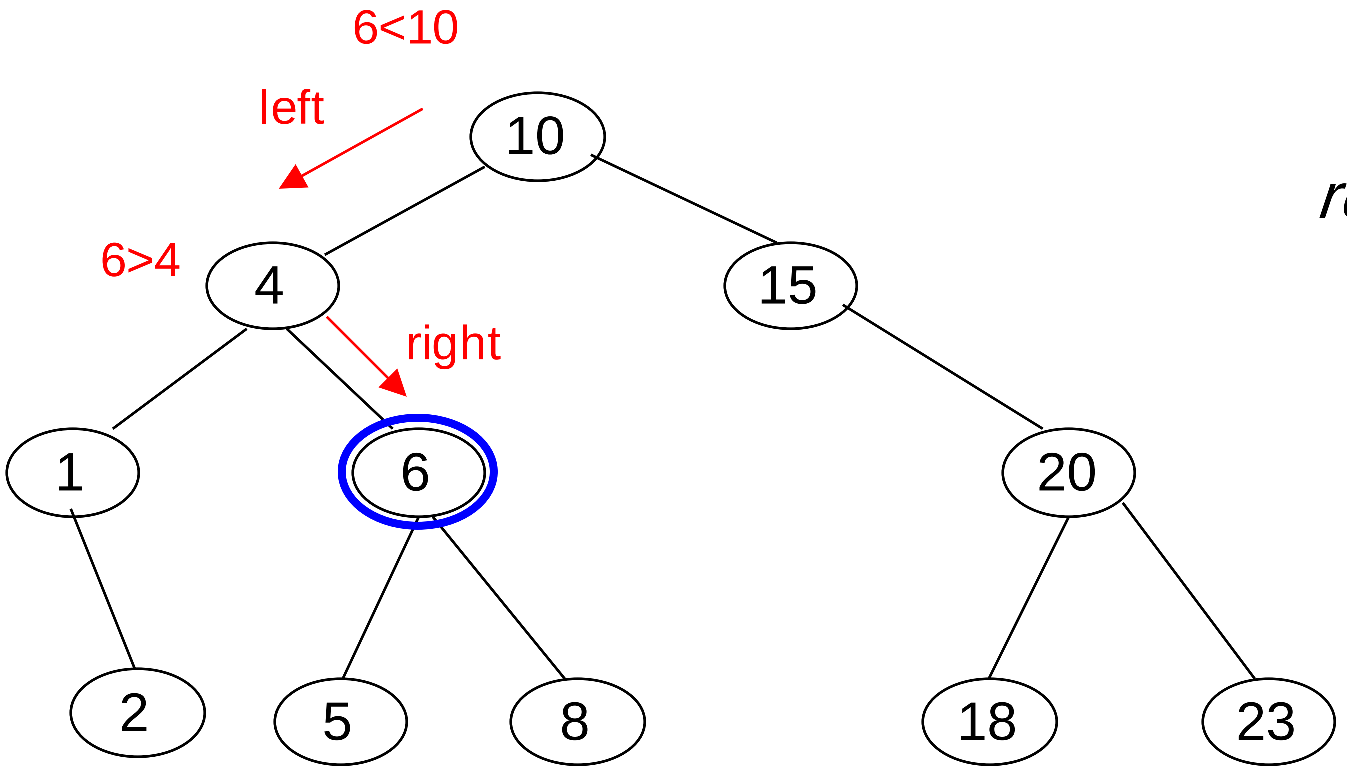
$T(n_2) = T(n_{21}) + T(n_{22}) + \theta(1)$

.
.
.

Overall using induction $\Longrightarrow \boxed{T(n)\ \epsilon\ \ \theta(n)}$

We have n nodes, and for each node it takes $\theta(1)$ time, thus the total time will be $n\ *\ \theta(1)$ $= \theta(n)$

Search on a BST - Example



Search for k=6,
return pointer x to node containing k

Search – Recursive Version - Algorithm

```
Tree-Search(x, k)
if x == NIL or K == x.key
    return x
if k < x.key
    return Tree_Search(x.left, k)
else return Tree_Search(x.right, k)
```

Search – Recursive Version - Code

```
// C function to search a given key in a given BST
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is smaller than root's key
    if (key < root->key)
        return search(root->left, key);

    // Key is greater than root's key
    return search(root->right, key);
}
```

Search – Recursive Version – Time Complexity

For each level of tree it takes constant time (according to code) → Total time will be:

- Height of tree * Constant Time
- Overall: $T(n) \in O(h)$ → $T(n) \in O(logn)$

Search – Iterative Version - Algorithm

```
Iterative-Tree-Search(x, k)
while x != NIL and k != x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

Search – Iterative Version - Code

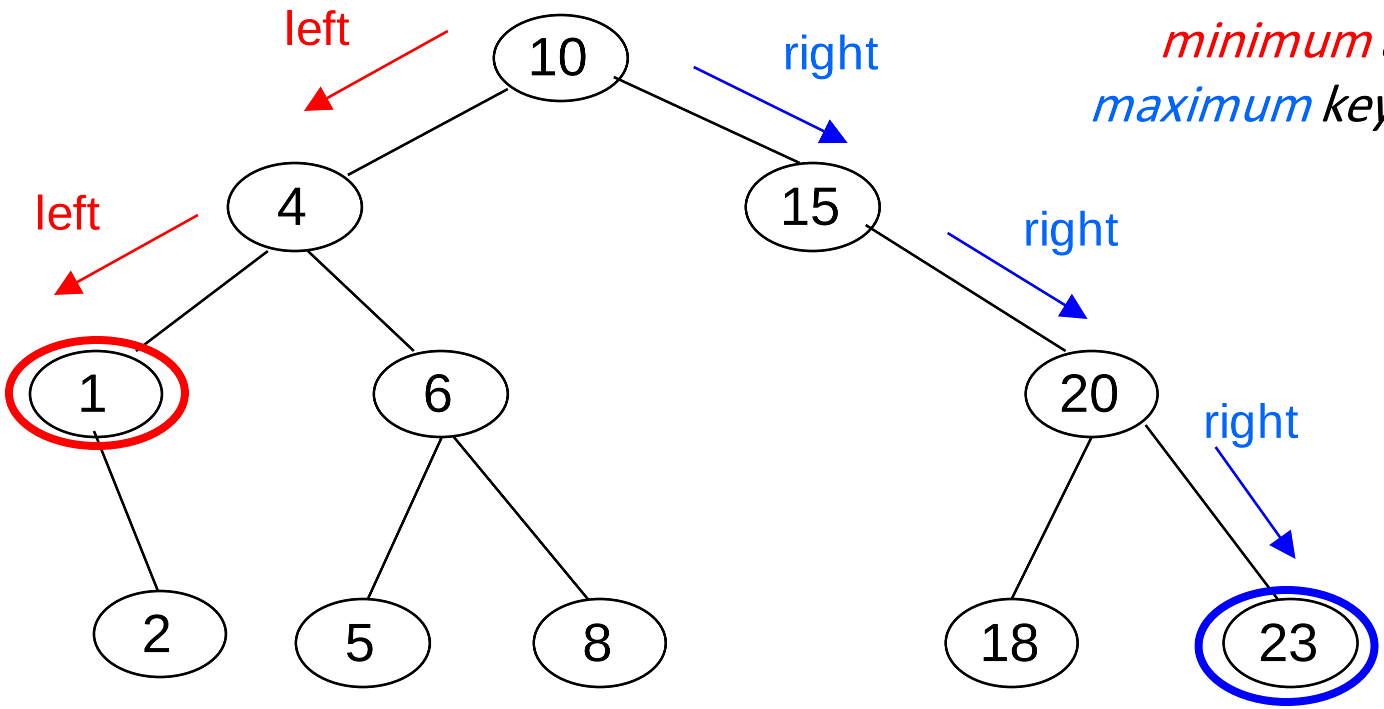
```
// Function to check the given key exist or not
bool iterativeSearch(struct Node* root, int key)
{
    // Traverse until root reaches to dead end
    while (root != NULL) {
        // pass right subtree as new tree
        if (key > root->data)
            root = root->right;

        // pass left subtree as new tree
        else if (key < root->data)
            root = root->left;
        else
            return true; // if the key is found return 1
    }
    return false;
}
```

Search – Iterative Version – Time Complexity

For each level of tree it takes constant time (according to code) → Total time will be:

- Height of tree * Constant Time
- Overall: $T(n) \in O(h)$ → $T(n) \in O(logn)$

Finding Minimum and Maximum - Example



Search nodes with minimum and maximum key values

Finding Minimum and Maximum - Algorithm

```
Tree-Minimum(x)
while x.left != NIL
    x = x.left
return x
```

Finding Minimum and Maximum - Code

```
int minValue(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
    {
        current = current->left;
    }
    return(current->key);
}
```

Time complexity: $T(n) \in O(h) \rightarrow T(n) \in O(logn)$

Finding Minimum and Maximum - Algorithm

```
Tree-Maximum(x)
while x.right != NIL
    x = x.right
return x
```

Finding Minimum and Maximum - Code

```c
int maxValue(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->right != NULL)
    {
        current = current->right;
    }
    return(current->key);
}
```

Time complexity: $T(n) \in O(h) \rightarrow T(n) \in O(logn)$

Finding Successor

Successor(x)

- The node y with the smallest key greater than or equal to $x.key$
- Ambiguous when multiple nodes have same key
- Successor in the in-order tree walk
- Return Nil if none exists -- has largest key

Finding Successor cont.

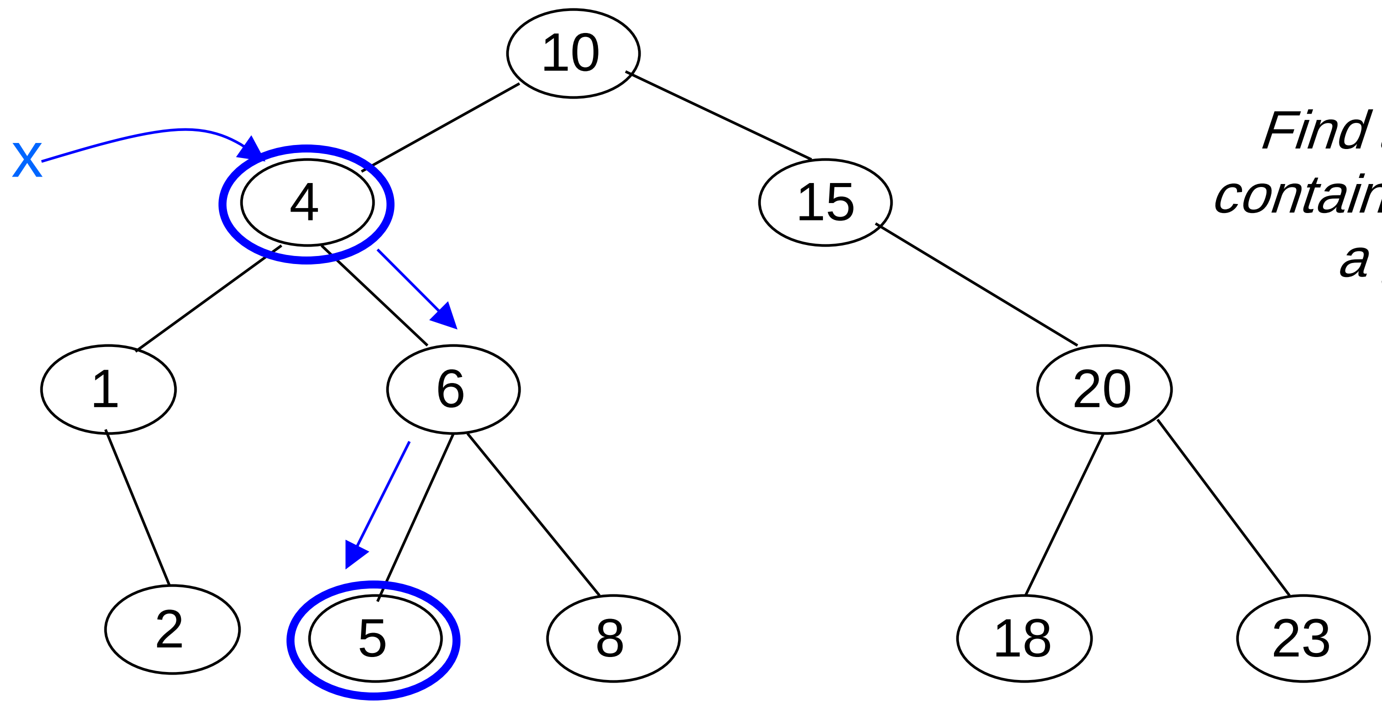
We have two cases in the process of finding successor:

1. If right child of x exists
   - Leftmost node in right sub-tree
2. Otherwise
   - Lowest ancestor y where x is in the left-sub-tree of y

Finding Successor cont.

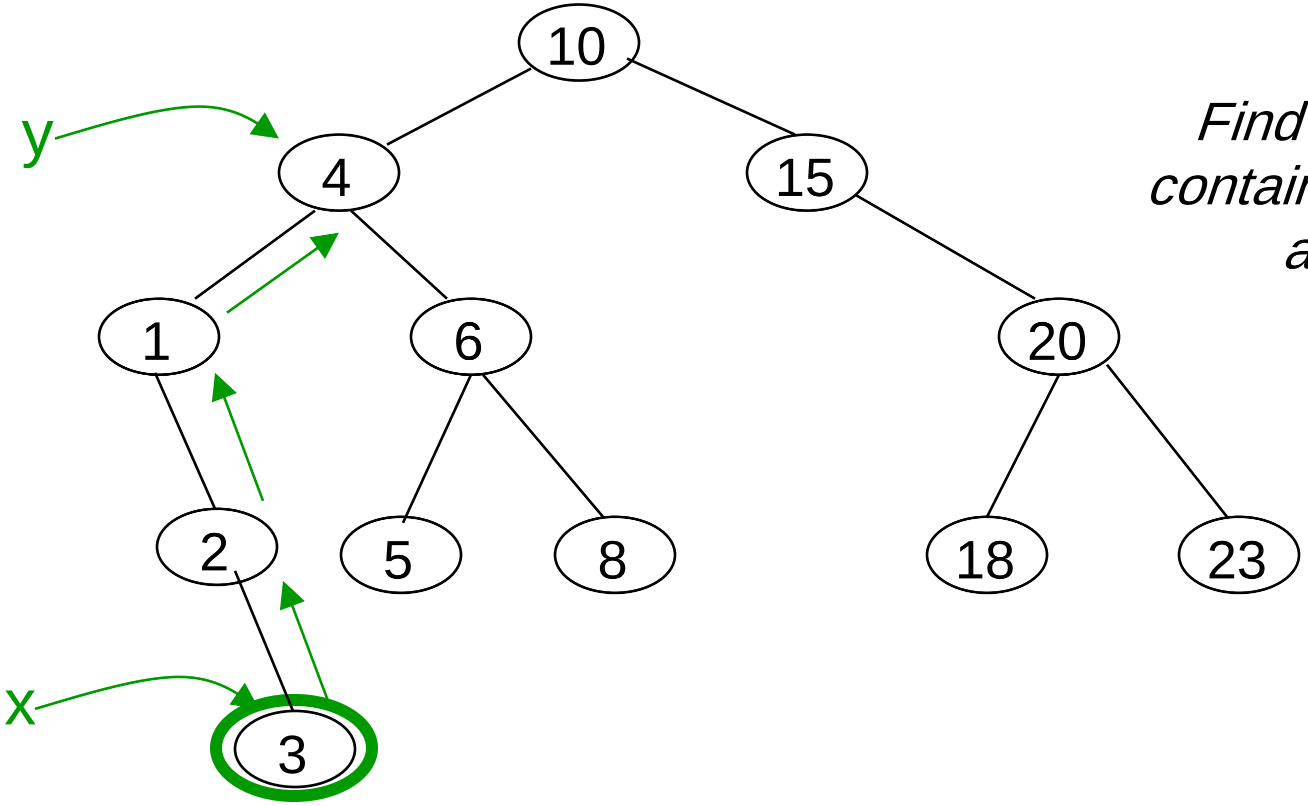
```
Tree-successor(x)
if (x.right ≠ NULL)
    return Tree-minimum(x.right)

    y = x.parent
    while (y ≠ NULL and x = y.right)
        x = y
        y = y.parent
    return  y
```

Finding successor - Example



Find the node y which contains the successor of a given node x

Finding successor – Example cont.



Find the node y which contains the successor of a given node x

Finding successor - Algorithm

```
Tree-successor(x)
if (x.right ≠ NULL)
    // Case 1
    return Tree-minimum(x.right)

    // Case 2
    y = x.parent
    while (y ≠ NULL and x = y.right)
        x = y
        y = y.parent
    return  y
```

Finding Successor – Time Complexity

For each level of tree it takes constant time (according to code) → Total time will be:

- Height of tree * Constant Time
- Overall: $T(n) \in O(h)$ → $T(n) \in O(logn)$

Finding Predecessor

We have two cases in the process of finding predecessor :

1. The left child of x exists
    - Rightmost child in the left sub-tree of x
2. Otherwise
    - Lowest ancestor y where x is in the right sub-tree of y.

- Completely symmetric