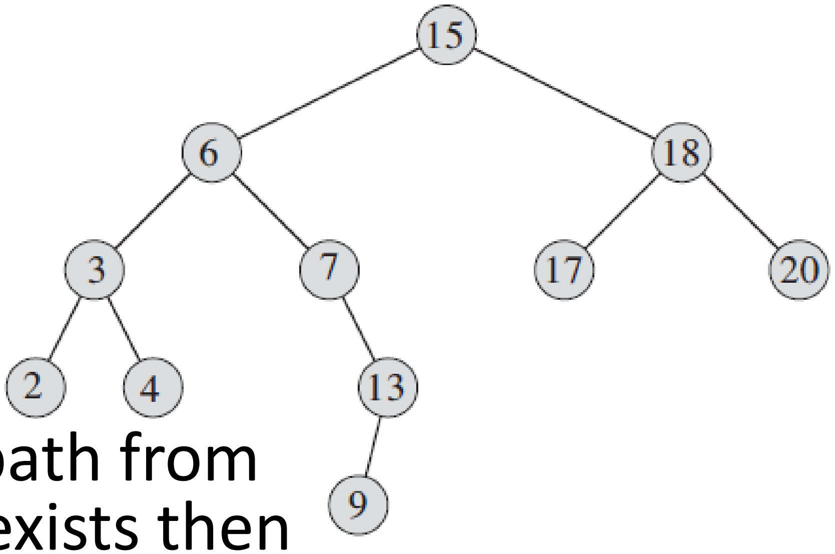
Binary Search Tree

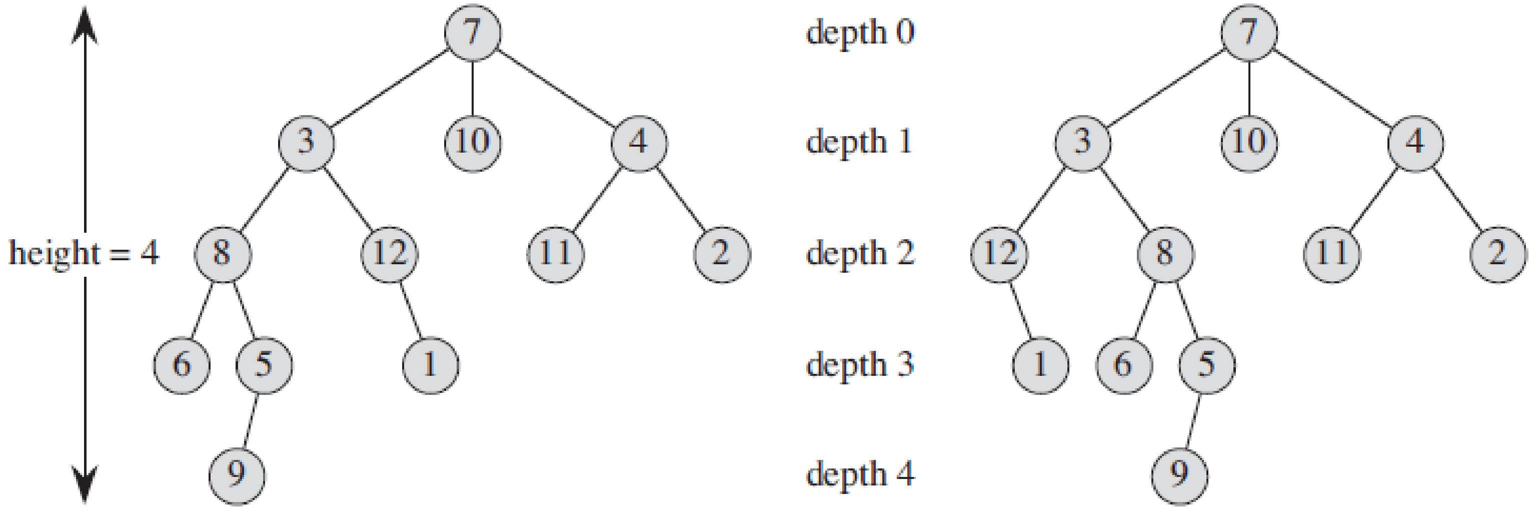
Terminology



- **Root**: Topmost node in a tree
- **Child** and **Parent**: On a simple path from root to a node, if an edge (x,y) exists then x is the parent of y, and y is a child of x.
- **Siblings**: Nodes with the same parent
- **Descendant**: Node reachable by repeated proceeding from parent to child
- **Ancestor**: Node reachable by repeated proceeding from child to parent.
- **Leaf (External node)**: Node with no children
- **Internal node**: Node with at least one child
- **Degree**: Number of children of a node

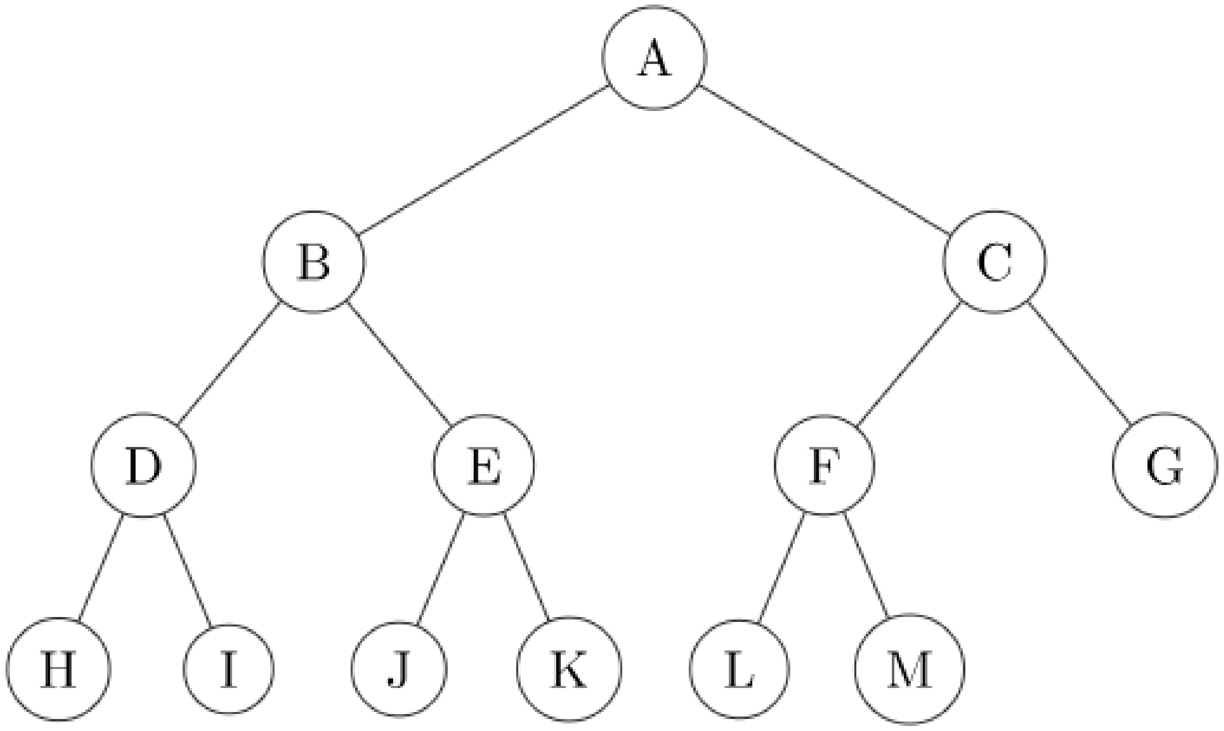
Contd…

- **Depth (or Level) of a node**: Length of a simple path from root to a node.
- **Height of a node**: Number of edges on the longest simple downward path from a node to a leaf.
- **Height of a tree**: Height of its root. It is also equal to the largest depth of any node in the tree.
- **Ordered tree**: Rooted tree with ordered children of each node.



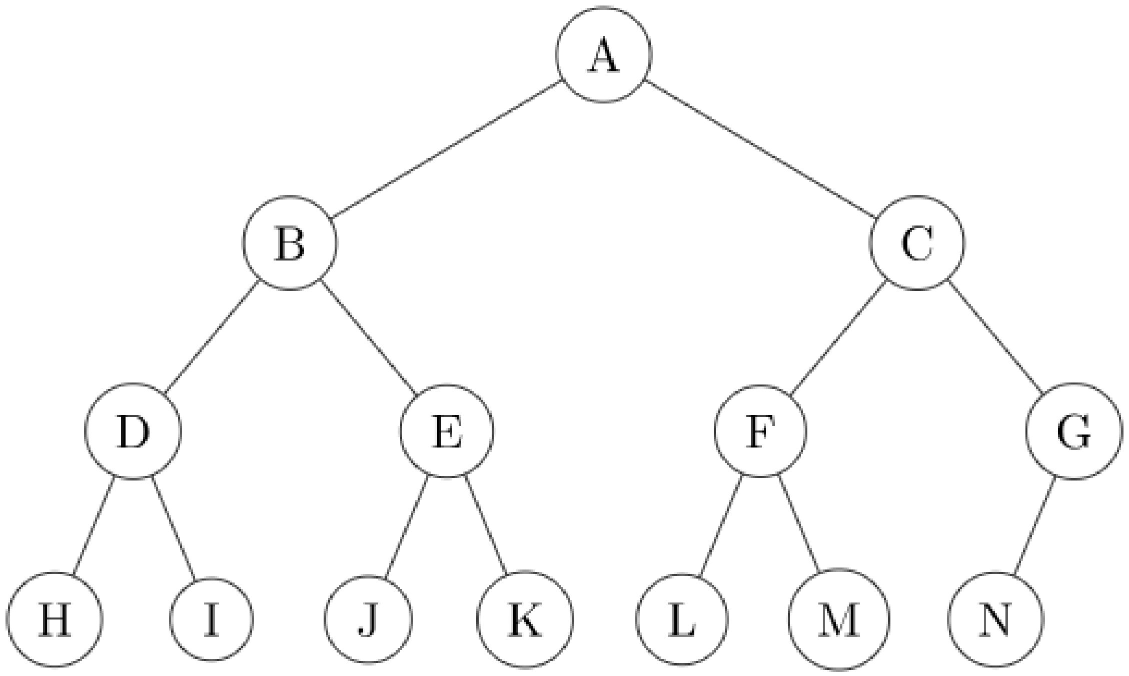
Full Binary Tree

- A full binary tree is one where every node has either 0 or 2 children.



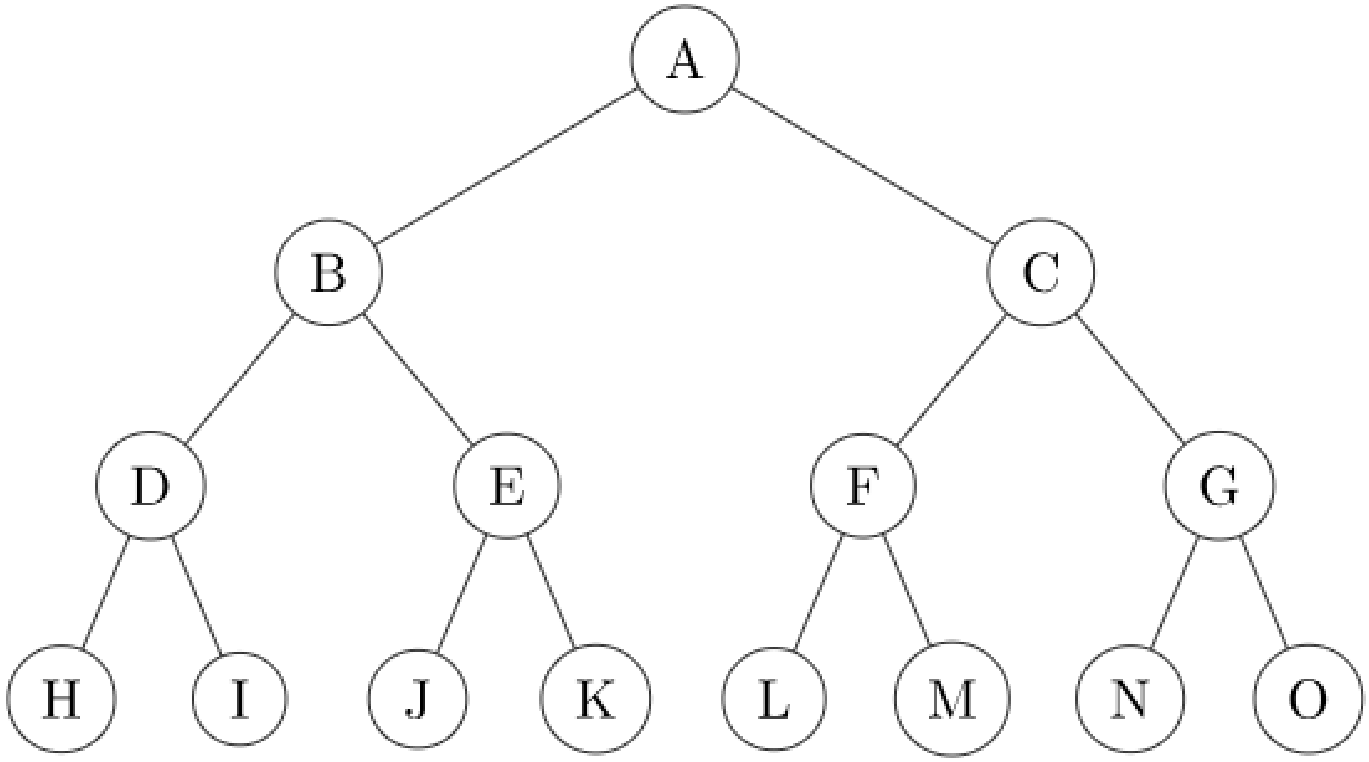
Complete Binary Tree

- In a complete binary tree, all levels are filled except the lowest level nodes, which are filled from the left.

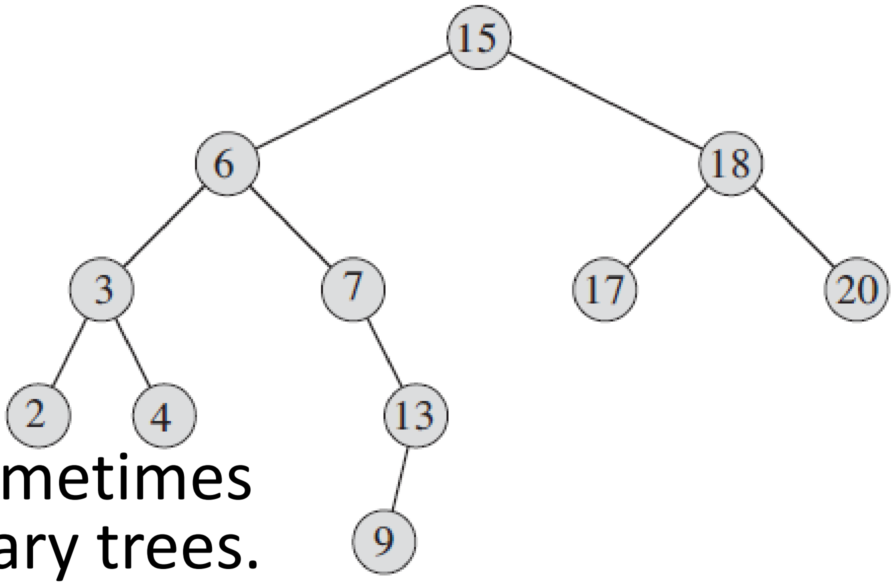


Perfect Binary Tree

- A perfect binary tree, as the name suggests, is the most perfect kind of binary tree with all its nodes with exactly two children, and all leaf nodes at the same level.



Introduction

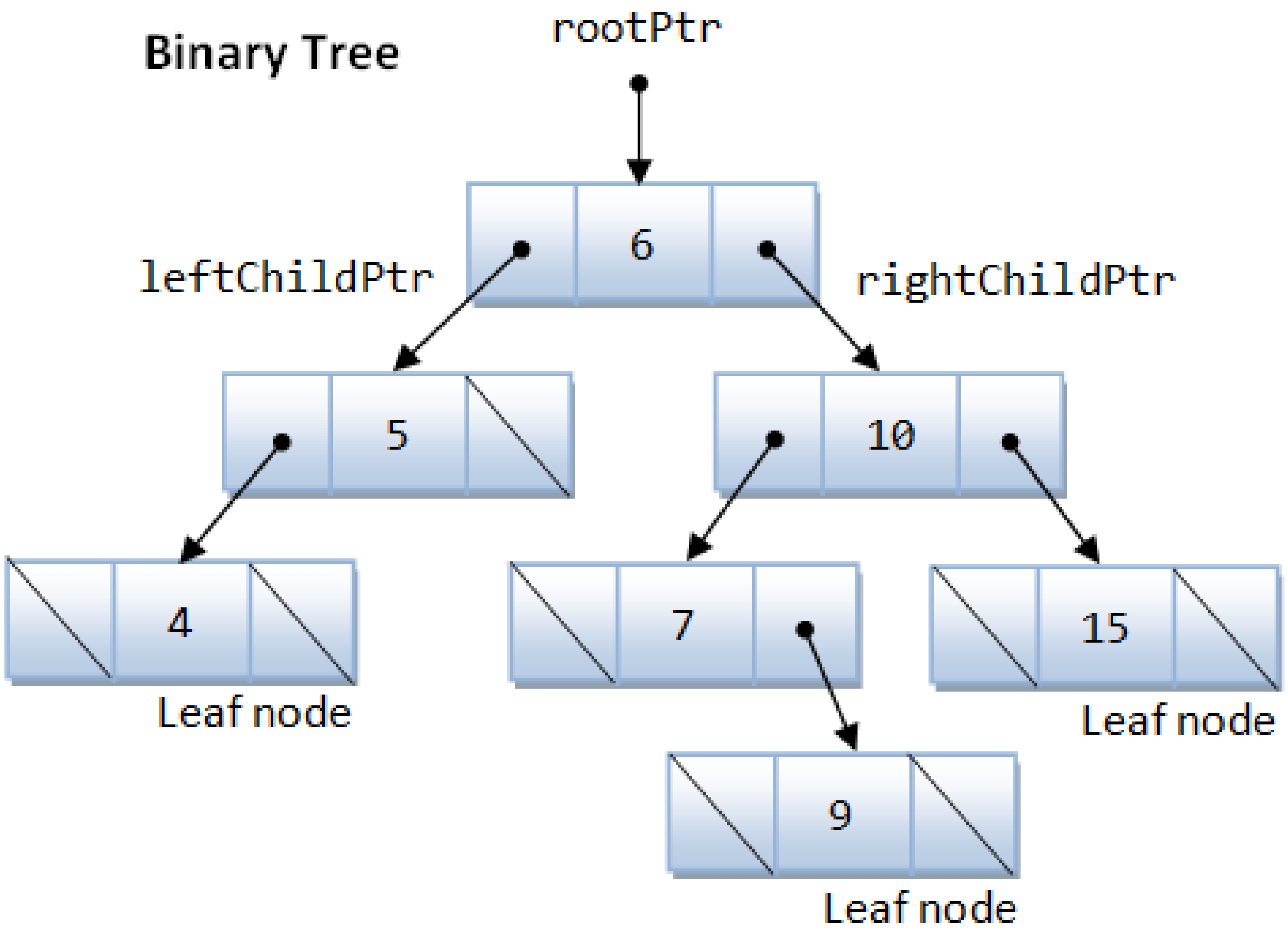


- Binary search trees (BST), sometimes called ordered or sorted binary trees.
- Properties:
  - The left subtree of a node contains only nodes with keys less than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.
- BSTs keep keys in sorted order, so that lookup, addition, and deletion operations can be executed efficiently.

Contd…

- All the operations traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees.
- On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree.
- On average, BST with n nodes has $O(\log_2 n)$ height.
- In the worst case, BST can have $O(n)$ height.

Example


Binary Tree
rootPtr
leftChildPtr
rightChildPtr
6
5
10
4
7
15
9
Leaf node
Leaf node
Leaf node

Structure code of a tree node

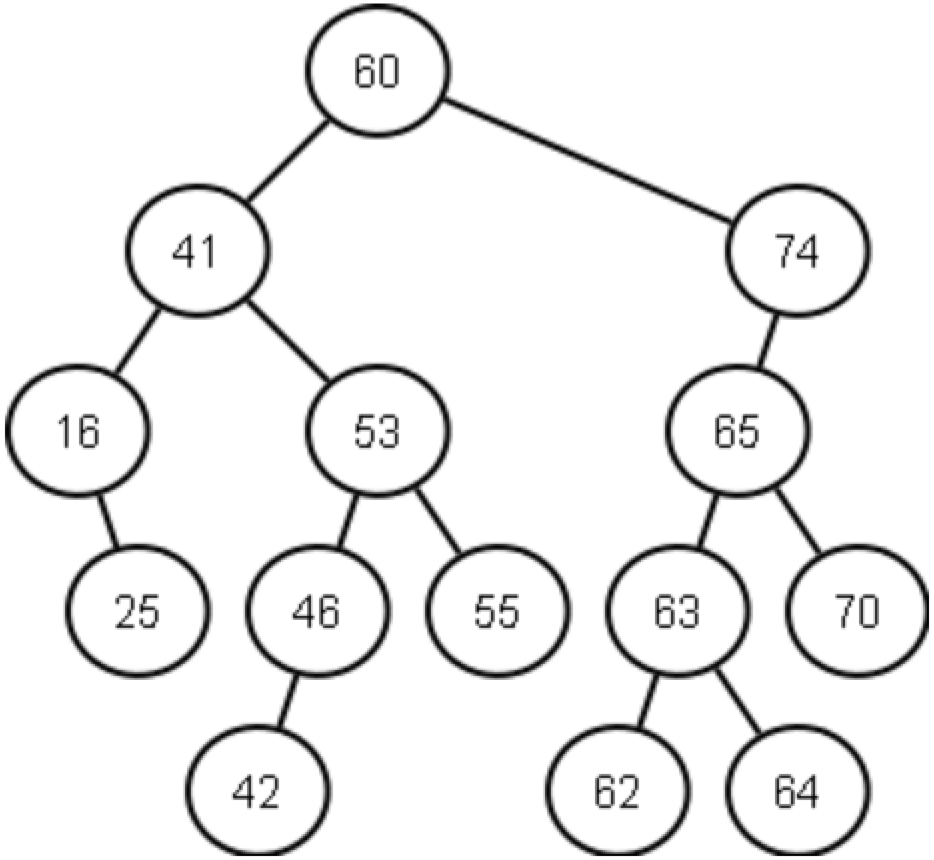
```
struct node
{ int data; //Data element
  struct node * left; //Pointer to left node
  struct node * right; //Pointer to right node
};

struct node * root = NULL;
```

Contd…

```
struct node * newnode(int element)
{  struct node * temp =
                    (node * )malloc(sizeof(node));
   temp->data=element;
   temp->left = temp->right = NULL;
   return temp; }
```

Traversals

- Preorder traversal: (parent, left, right)
- Inorder traversal: (left, parent, right)
- Postorder traversal: (left, right, parent)



- Preorder traversal
  60, 41, 16, 25, 53, 46, 42, 55, 74, 65, 63, 62, 64, 70
- Inorder traversal
  16, 25, 41, 42, 46, 53, 55, 60, 62, 63, 64, 65, 70, 74
- Postorder traversal
  25, 16, 42, 46, 55, 53, 41, 62, 64, 63, 70, 65, 74, 60

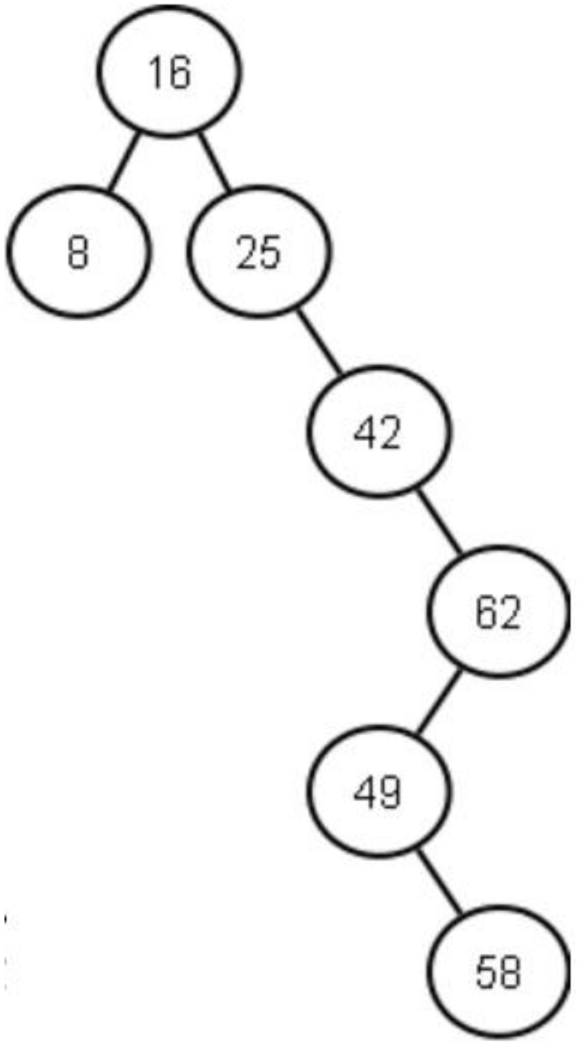
Traversals

- Preorder traversal

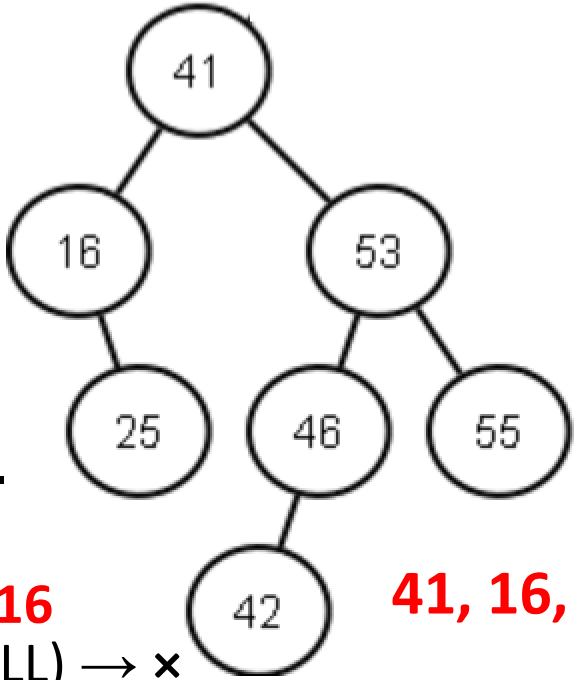
  16, 8, 25, 42, 62, 49, 58
- Inorder traversal

  8, 16, 25, 42, 49, 58, 62
- Postorder traversal

  8, 58, 49, 62, 42, 25, 16

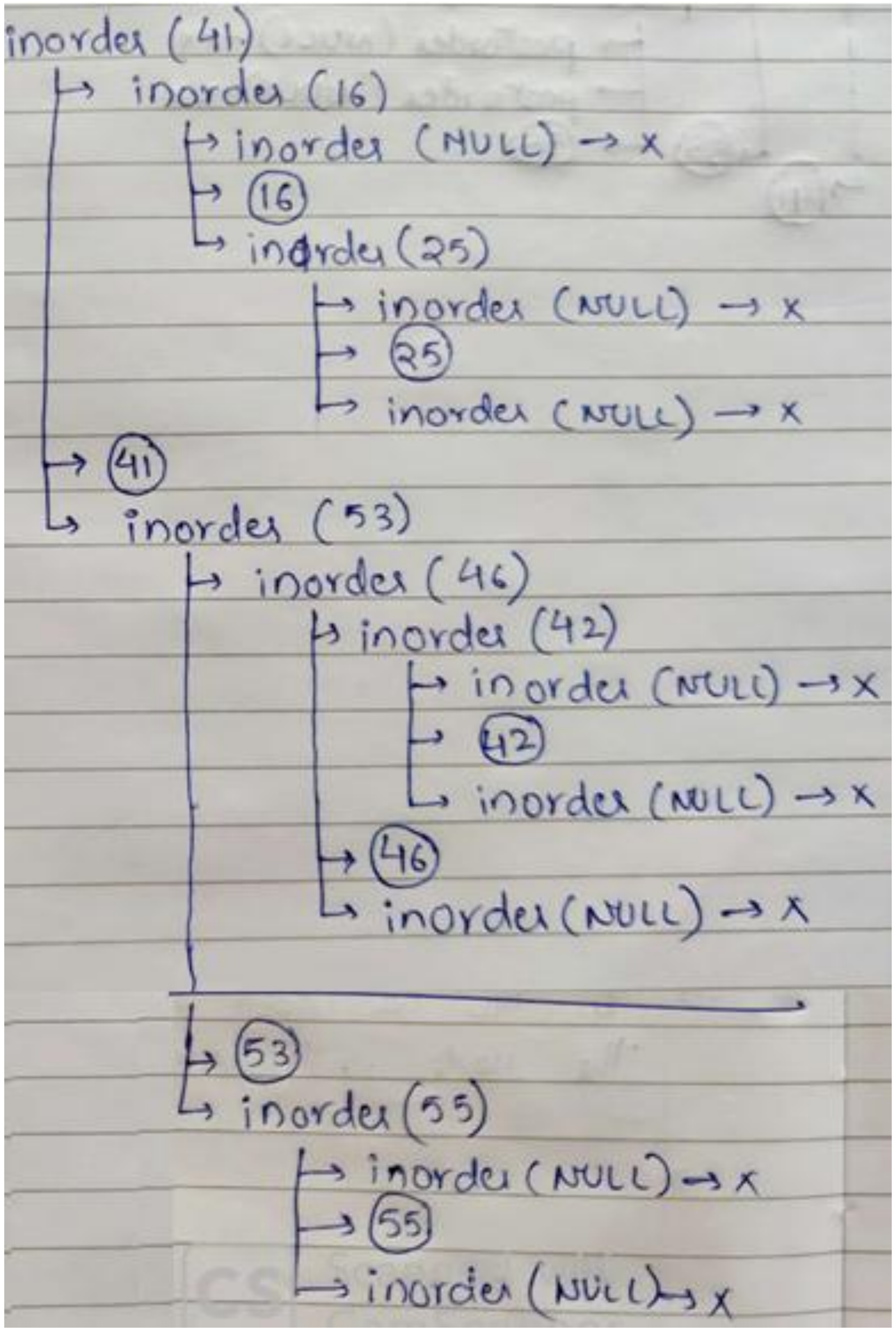


```
void preorder(node *temp)
{  if (temp != NULL)
   {  printf("%d", temp->data);
      preorder(temp->lchild);
      preorder(temp->rchild);
   } }
```

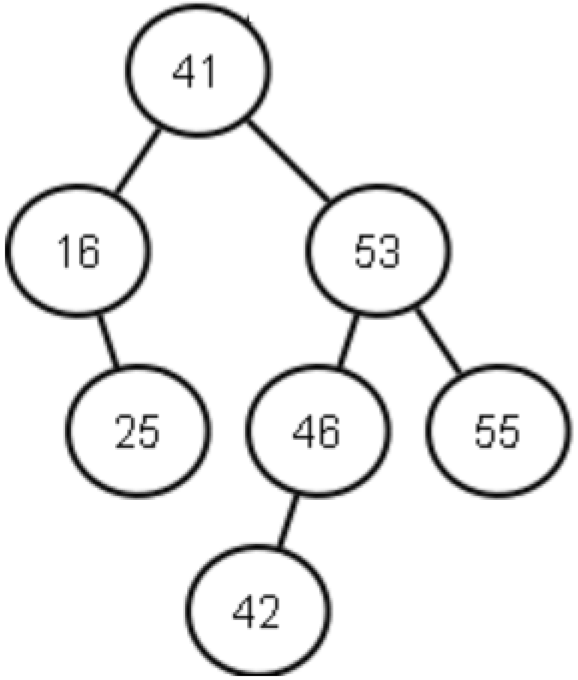


41, 16, 25, 53, 46, 42, 55

```
preorder(41) → 41
  → preorder(16) → 16
      → preorder(NULL) → ×
      → preorder(25) → 25
          → preorder(NULL) → ×
          → preorder(NULL) → ×
  → preorder(53) → 53
      → preorder(46) → 46
          → preorder(42) → 42
              → preorder(NULL) → ×
              → preorder(NULL) → ×
          → preorder(NULL) → ×
      → preorder(55) → 55
          → preorder(NULL) → ×
          → preorder(NULL) → ×
```

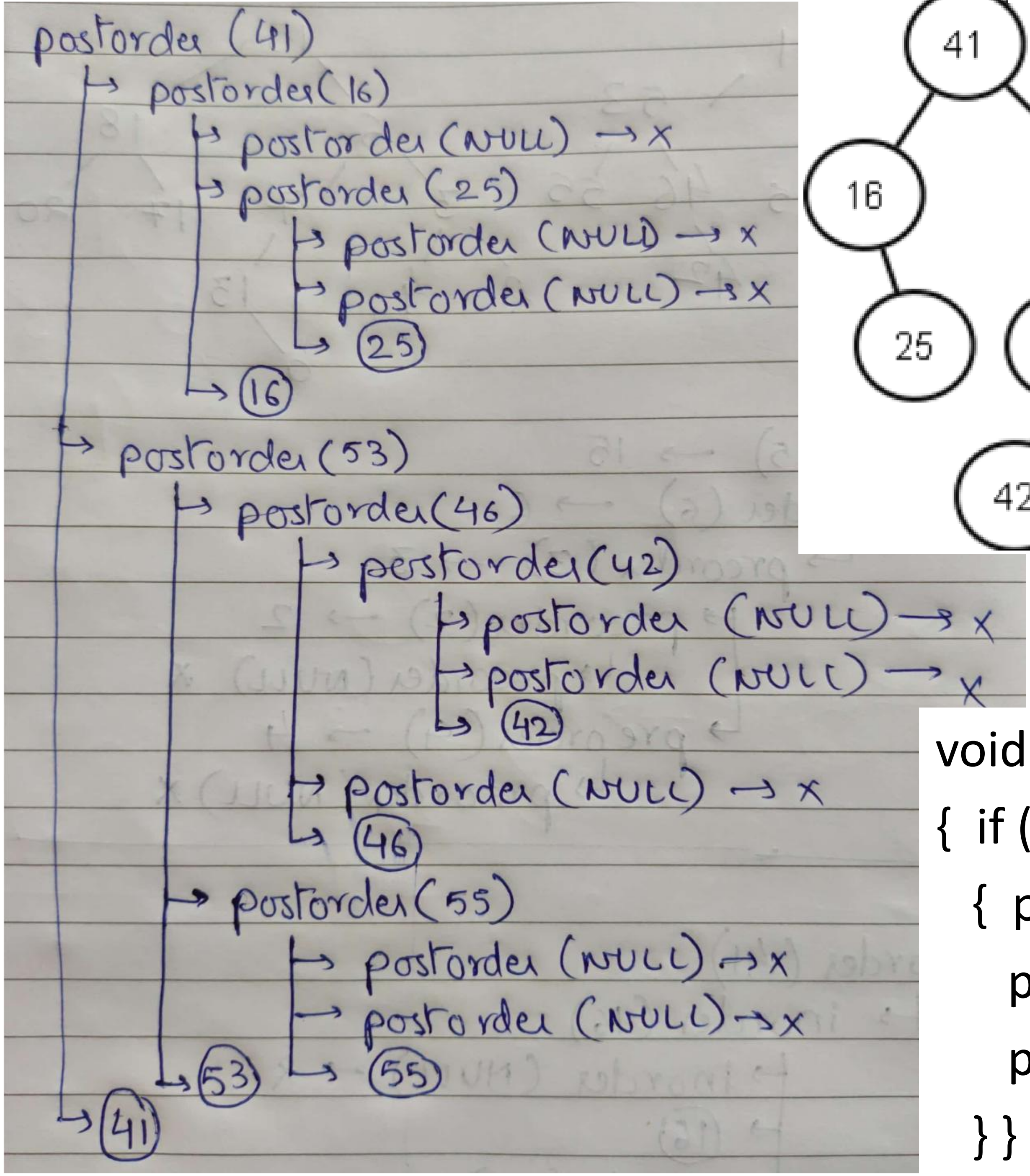


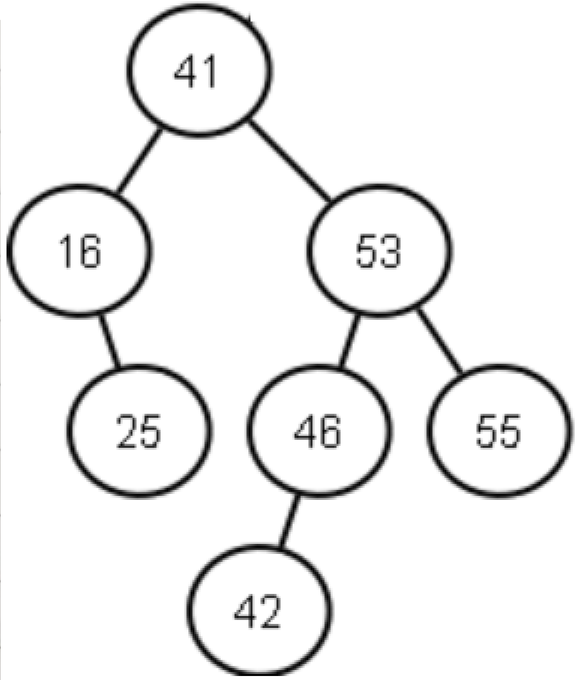
```
inorder (41)
 ├→ inorder (16)
 │    ├→ inorder (NULL) → x
 │    ├→ 16
 │    └→ inorder (25)
 │         ├→ inorder (NULL) → x
 │         ├→ 25
 │         └→ inorder (NULL) → x
 ├→ 41
 └→ inorder (53)
      ├→ inorder (46)
      │    ├→ inorder (42)
      │    │    ├→ inorder (NULL) → x
      │    │    ├→ 42
      │    │    └→ inorder (NULL) → x
      │    ├→ 46
      │    └→ inorder (NULL) → x
      ├→ 53
      └→ inorder (55)
           ├→ inorder (NULL) → x
           ├→ 55
           └→ inorder (NULL) → x
```



16, 25, 41, 42, 46, 53, 55

```
void inorder(node *temp)
{  if (temp != NULL)
   {  inorder(temp->lchild);
      printf("%d", temp->data);
      inorder(temp->rchild);
   }}
```

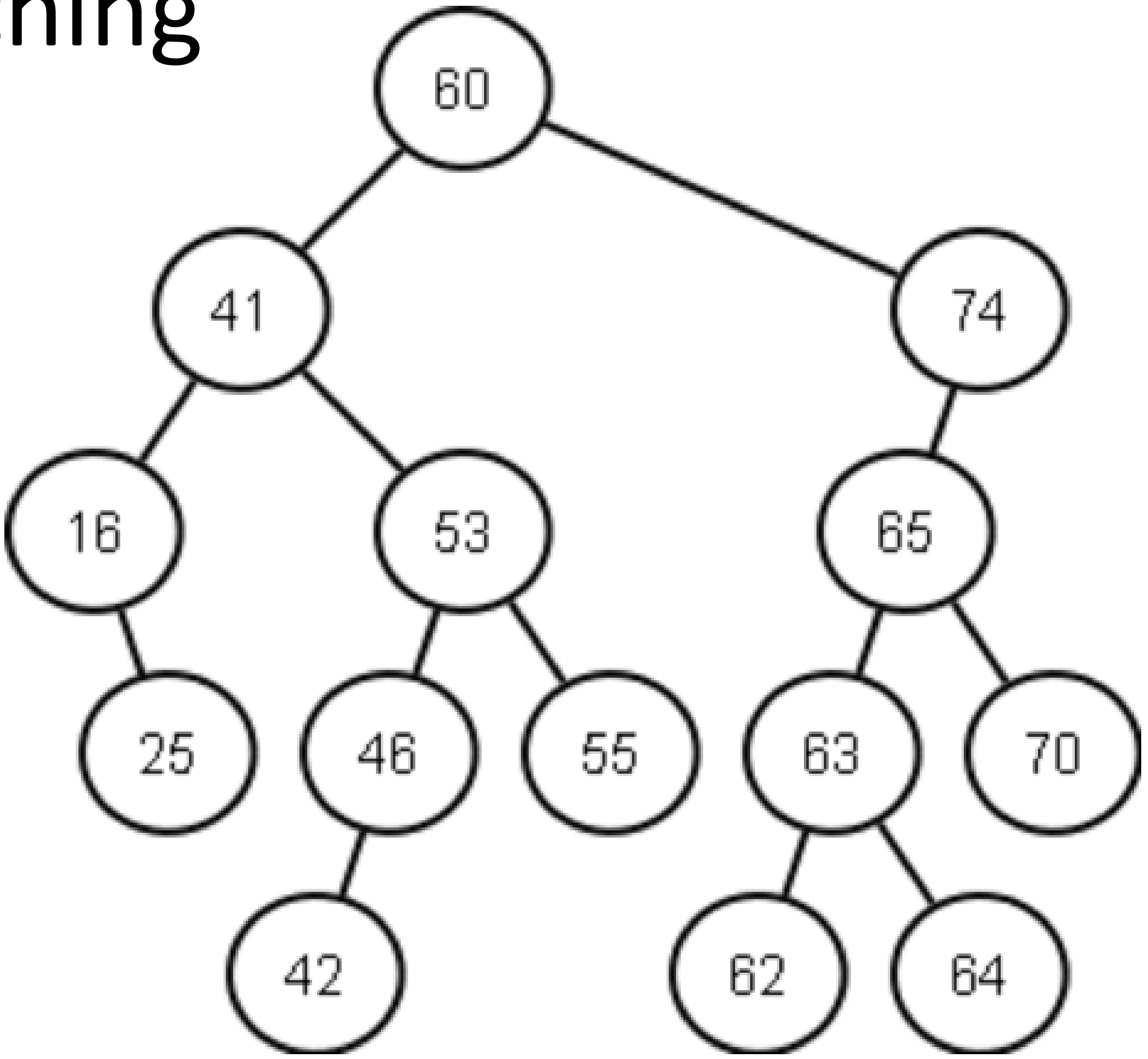




25, 16, 42, 46, 55, 53, 41

```
void postorder(node *temp)
{  if (temp != NULL)
   {  postorder(temp->lchild);
      postorder(temp->rchild);
      printf("%d", temp->data);
   }}
```

Searching



Contd…

```
TREE-SEARCH($x, k$)
if $x == \text{NIL}$ or $k == x.key$
    return $x$
if $k < x.key$
    return TREE-SEARCH($x.left, k$)
else return TREE-SEARCH($x.right, k$)
```

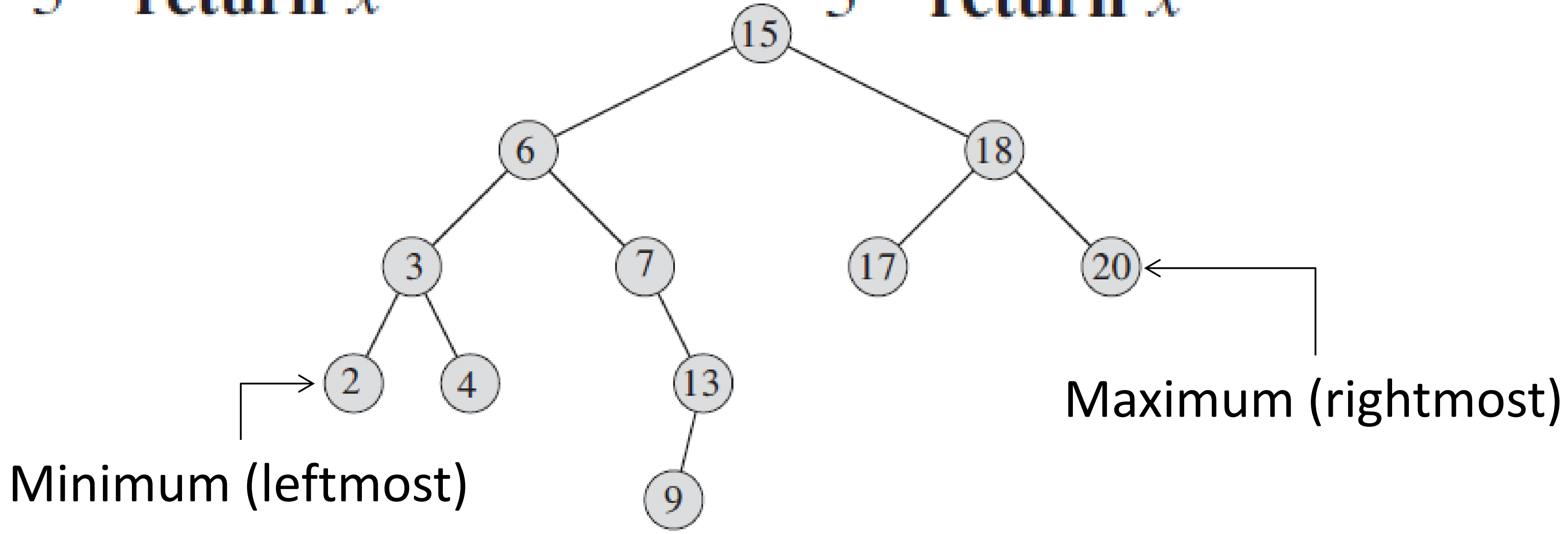
Running time is O(h), where h is the height of the tree.

```
ITERATIVE-TREE-SEARCH($x, k$)
while $x \neq \text{NIL}$ and $k \neq x.key$
    if $k < x.key$
        $x = x.left$
    else $x = x.right$
return $x$
```

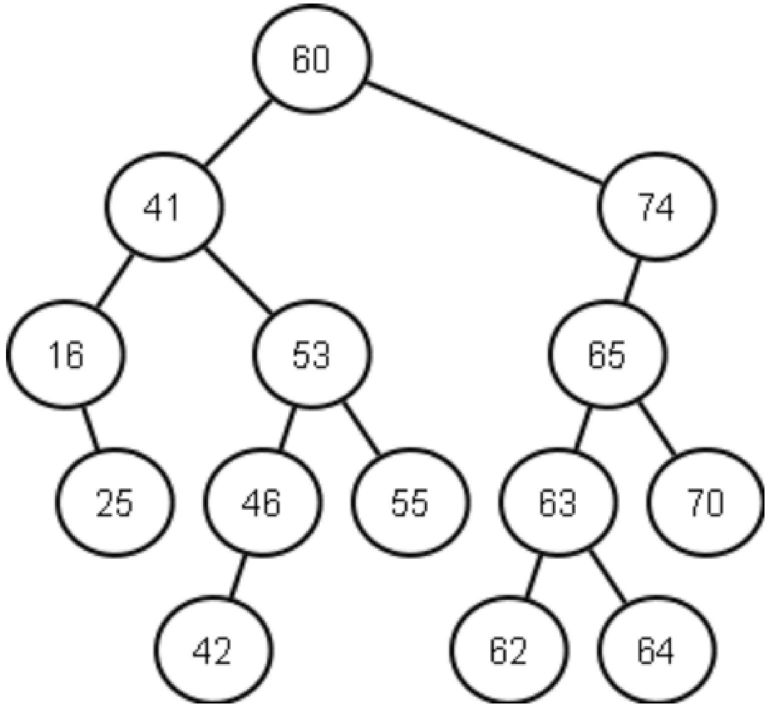
Minimum and Maximum

```
TREE-MINIMUM(x)
1  while $x.left \neq$ NIL
2      $x = x.left$
3  return $x$
```

```
TREE-MAXIMUM(x)
1  while $x.right \neq$ NIL
2      $x = x.right$
3  return $x$
```

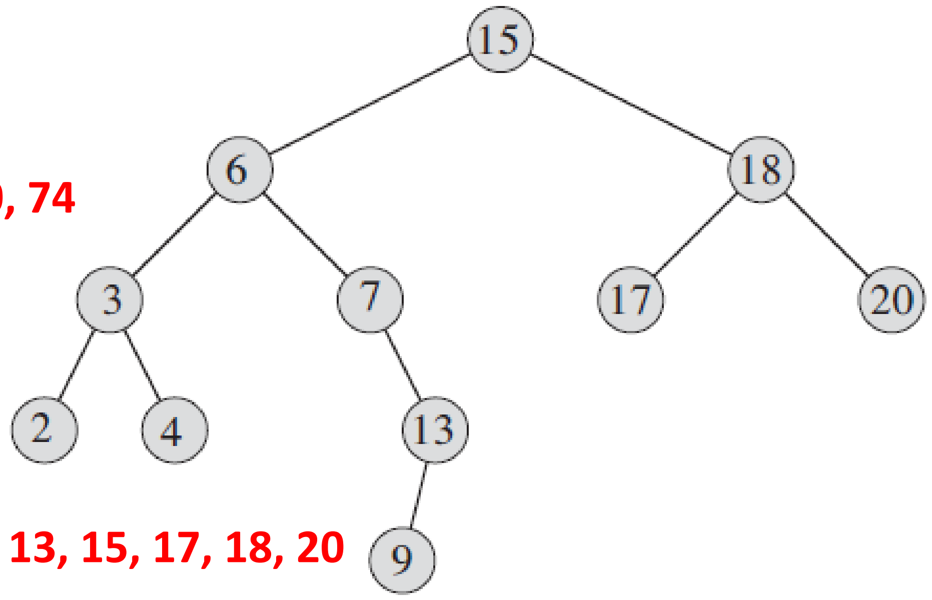


Successor and Predecessor



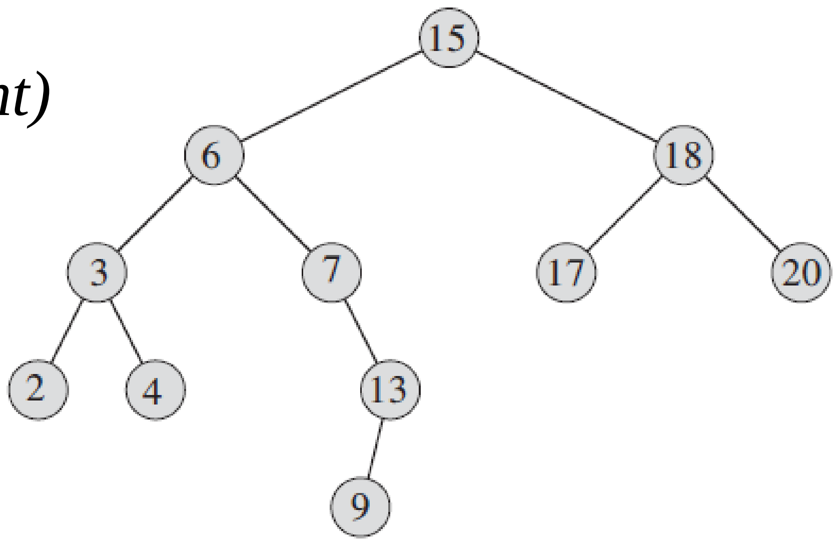
- Successor is next node in inorder traversal.
- Predecessor is previous node in inorder traversal.

16, 25, 41, 42, 46, 53, 55, 60, 62, 63, 64, 65, 70, 74



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

```
TREE-SUCCESSOR-WITHOUT-PARENT($y$,$x$)
if $x.right \neq$ NIL
    return TREE-MINIMUM($x.right$)
$succ$ = NIL
while TRUE
    if $x.key < y.key$
        $succ = y$
        $y = y.left$
    else if $x.key > y.key$
        $y = y.right$
    else break
return $succ$
```

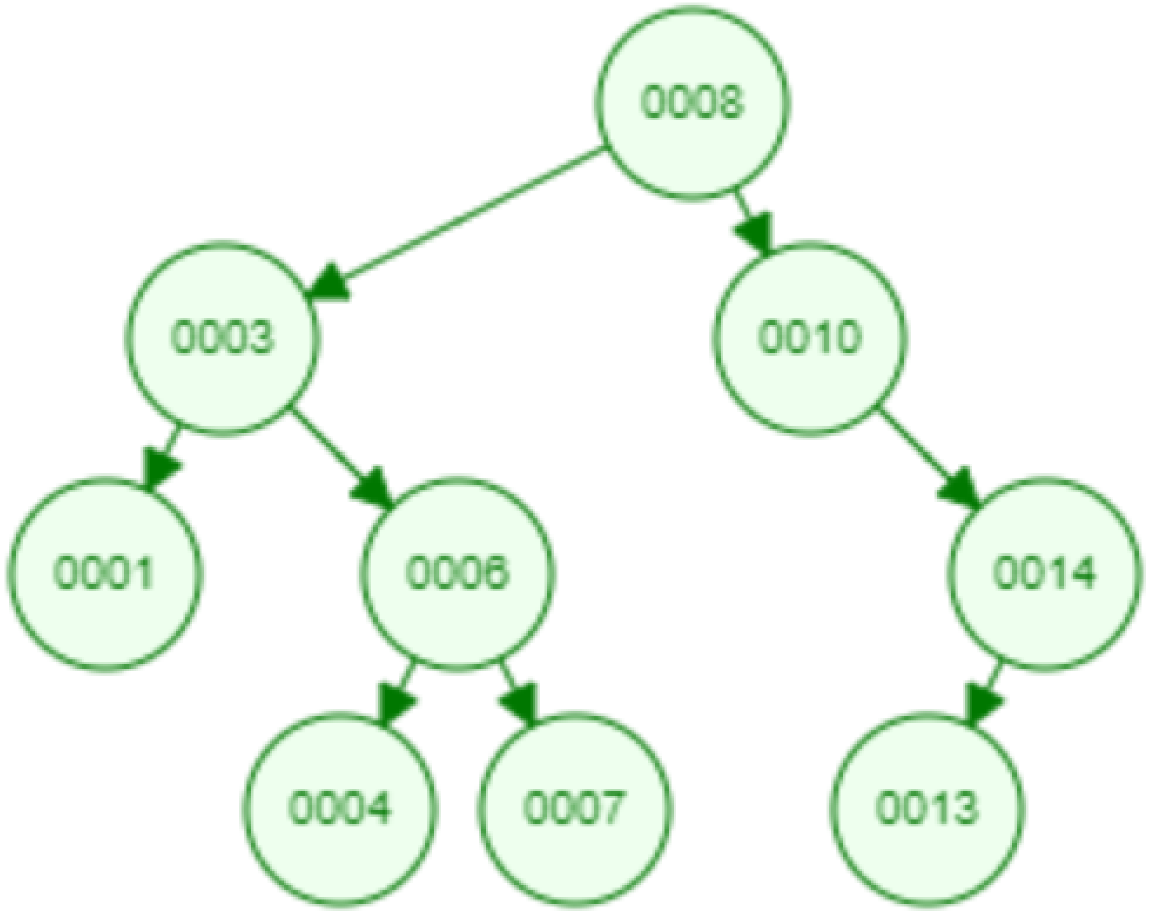


```
TREE-SUCCESSOR($x$)
if $x.right \neq$ NIL
    return TREE-MINIMUM($x.right$)
$y = x.p$
while $y \neq$ NIL and $x == y.right$
    $x = y$
    $y = y.p$
return $y$
```

Insertion

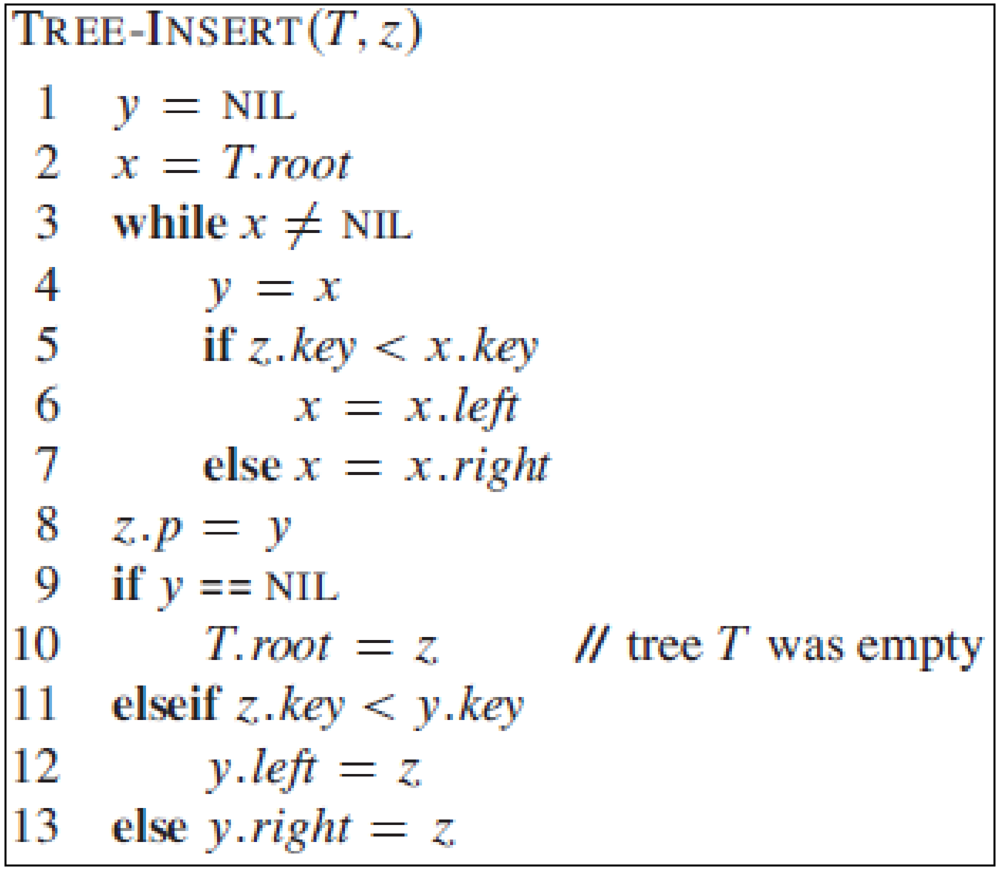
- A new key is always inserted at leaf.
- Start searching a key from root till a leaf node is found.
  - Add the new node as a child of the leaf node.
- Duplicate key values are not allowed.
- If key to be inserted is already present, then return that node.

Create BST using
8, 3, 1, 10, 6, 14, 4, 7, 13



Contd...

```
TREE-INSERT-WITHOUT-PARENT($T.root$,$z$)
    $x = T.root$
    if $x$ == NIL
        return $z$
    if $z.key < x.key$
        $x.left$ = TREE-INSERT-WITHOUT-PARENT($x.left$,$z$)
    else if $z.key > x.key$
        $x.right$ = TREE-INSERT-WITHOUT-PARENT($x.right$,$z$)
    return $x$
```



```
TREE-INSERT($T, z$)
    $y$ = NIL
    $x = T.root$
    while $x \neq$ NIL
        $y = x$
        if $z.key < x.key$
            $x = x.left$
        else $x = x.right$
    $z.p = y$
    if $y$ == NIL
        $T.root = z$          // tree $T$ was empty
    elseif $z.key < y.key$
        $y.left = z$
    else $y.right = z$
```

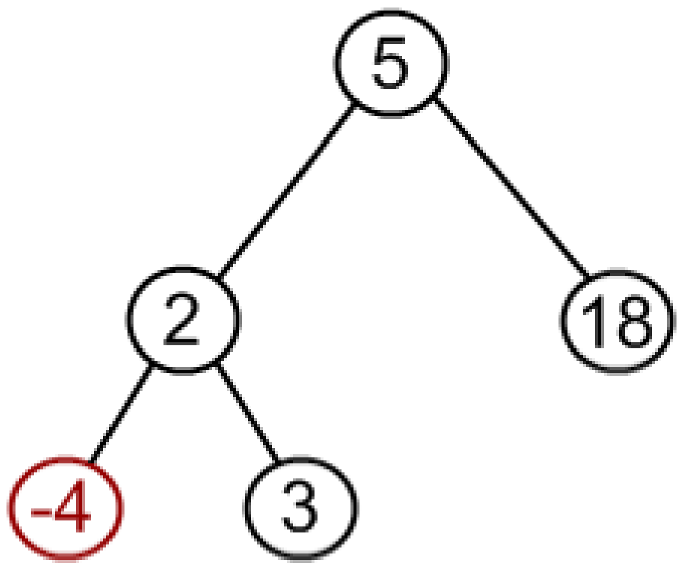
Deletion

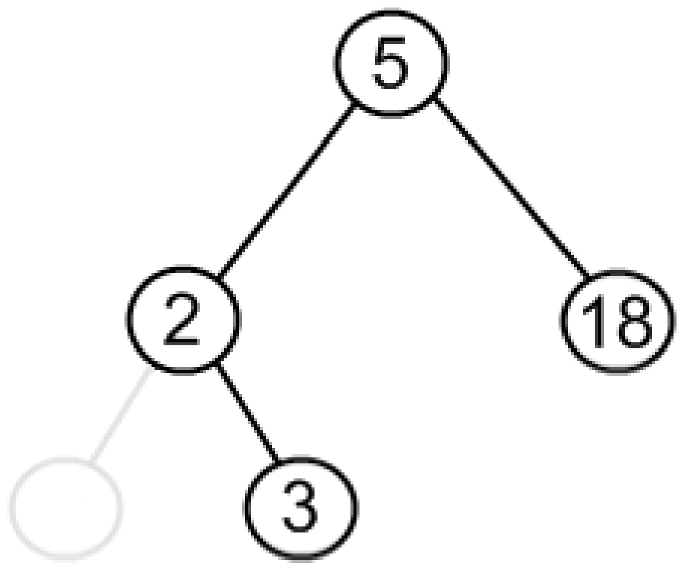
1. Search for a node to remove;
2. If the node is found, execute the remove algorithm.

- Three cases,
  1. Node to be removed has no children.
  2. Node to be removed has one child.
  3. Node to be removed has two children.

Node to be removed has no children

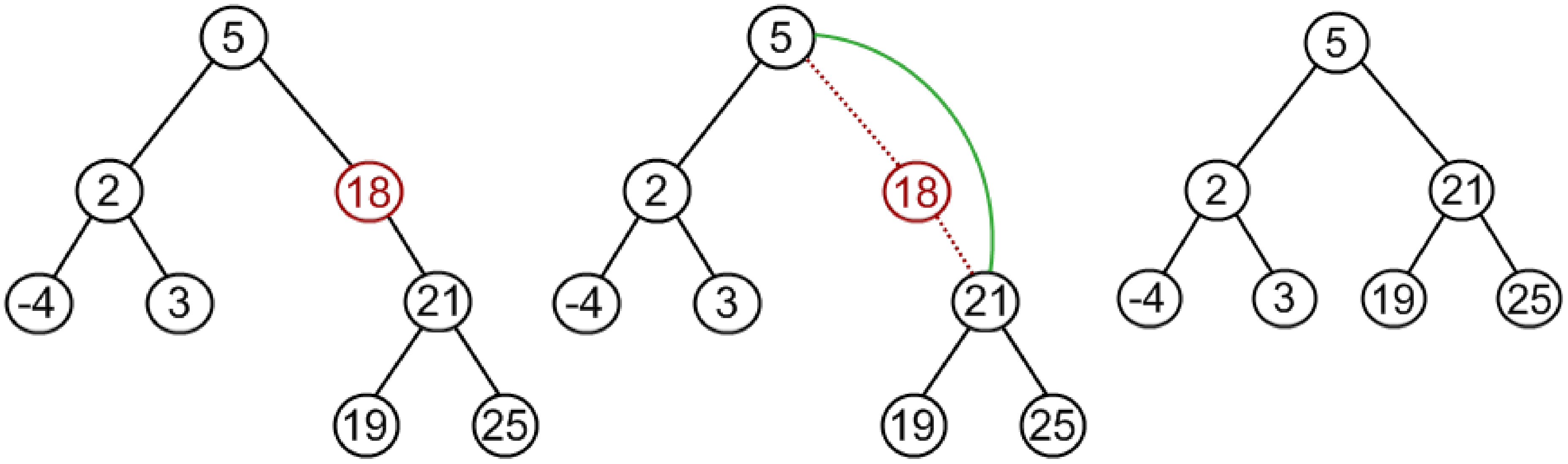
- Set corresponding link of the parent to NULL and disposes the node.
- Example. Remove -4 from a BST.





Node to be removed has one child

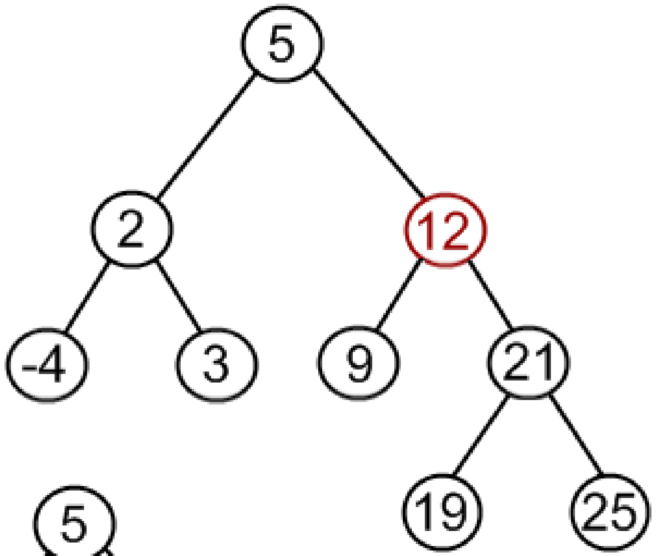
- Link single child (with it's subtree) directly to the parent of the removed node.
- Example. Remove 18 from a BST.



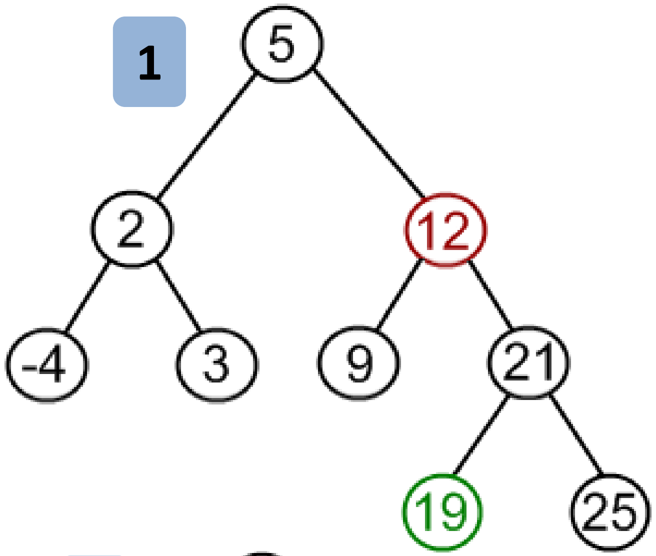
Node to be removed has two children

- Find inorder successor, i.e. find the minimum value in right child of the node.
- Copy contents of the inorder successor (or found minimum) to the node being removed with.
- Delete the inorder successor (or found minimum) from the right subtree.
- Note:
  - The node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.
  - Inorder predecessor can also be used.

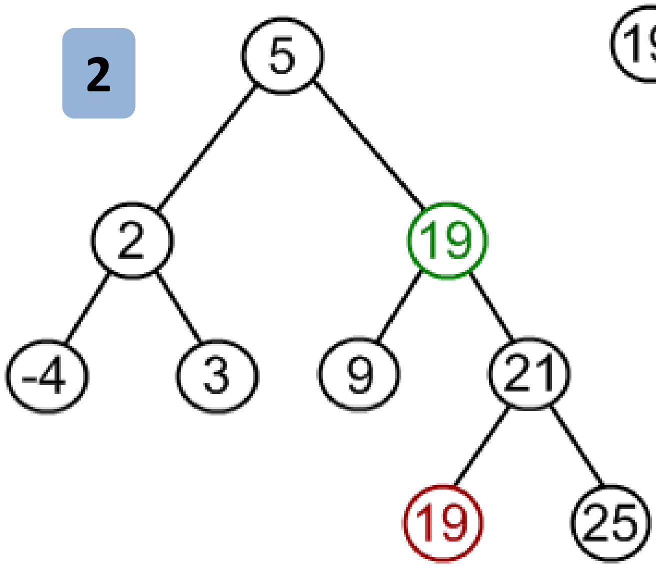
Contd… Remove 12 from a BST.



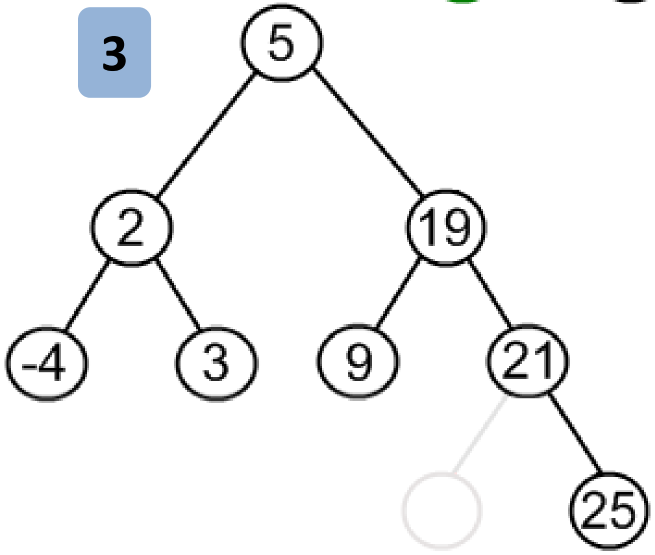
1



2



3



```
TRANSPLANT($T, u, v$)
if $u.p == \text{NIL}$
    $T.root = v$
elseif $u == u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq \text{NIL}$
    $v.p = u.p$
```

```
TREE-DELETE($T, z$)
if $z.left == \text{NIL}$
    TRANSPLANT($T, z, z.right$)
elseif $z.right == \text{NIL}$
    TRANSPLANT($T, z, z.left$)
else $y = $ TREE-MINIMUM($z.right$)
    if $y.p \neq z$
        TRANSPLANT($T, y, y.right$)
        $y.right = z.right$
        $y.right.p = y$
    TRANSPLANT($T, z, y$)
    $y.left = z.left$
    $y.left.p = y$
```

```
TREE-DELETE-WITHOUT-PARENT($T.root$, $k$)
    $x = T.root$
    if $x$ == NIL
        return $x$
    if $k < x.key$
        $x.left$ = TREE-DELETE-WITHOUT-PARENT($x.left$, $k$)
    else if $k > x.key$
        $x.right$ = TREE-DELETE-WITHOUT-PARENT($x.right$, $k$)
    else
        if $x.left$ == NIL
            $temp = x.right$
            delete $x$
            return $temp$
        else if $x.right$ == NIL
            $temp = x.left$
            delete $x$
            return $temp$
        $temp$ = TREE-MINIMUM($x.right$)
        $x.key = temp.key$
        $x.right$ = TREE-DELETE-WITHOUT-PARENT($x.right$, $temp.key$)
    return $x$
```

Evaluating an Expression Tree

The expression tree is a binary tree with each internal node representing an operand, and each leaf node representing an operator.

Properties of an expression tree

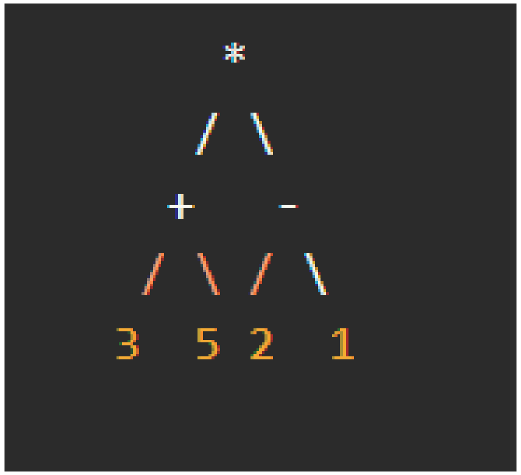
- Leaf nodes represent operands (numbers or variables).
- Internal nodes represent operators (such as +, -, *, /). Each operator in the tree corresponds to a sub-expression of the original expression.
- The structure of the tree preserves operator precedence, ensuring that the order of operations is maintained.
- An expression tree can be used to evaluate an arithmetic expression. By recursively evaluating the left and right subtrees and applying the operator at the internal node, the result can be computed.

Evaluating an Expression Tree

Example:

The expression (3 + 5) * (2 - 1) can be represented as an expression tree where:

- The root is the multiplication operator ‘*’.
- The left child is the addition operator ‘+’, with operands ‘3’ and ‘5’ as its children.
- The right child is the subtraction operator’ -’, with operands ‘2’ and ‘1’ as its children.



```
       *
      / \
     +   -
    / \ / \
   3  5 2  1
```

Infix to Expression Tree

Step 1: Start with the lowest precedence operator (if multiple operators are present).

Step 2: Make this operator the root of the tree.

Step 3: The left subtree is created from the part of the expression to the left of the operator, and the right subtree is created from the part of the expression to the right of the operator.\

Step 4: Recursively apply this process to the left and right subexpressions.

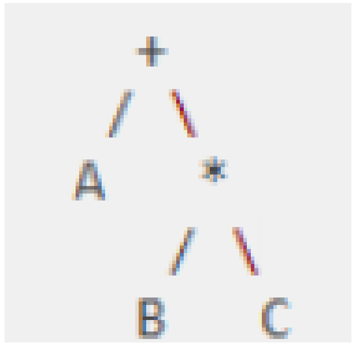
Infix to Expression Tree

Example: A + B * C

The + has a lower precedence than *, so + becomes the root.

Operand A becomes the left child of +.

The expression B*C becomes the right subtree. Since * has the highest precedence, it is the root of this subtree, and B and C are its children.



Question…

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

- Generate BST for the given sequence.
- Visit the generated BST using inorder, preorder, and postorder traversals.
- Delete 4, 7, and 15 in sequence showing BST after every deletion.