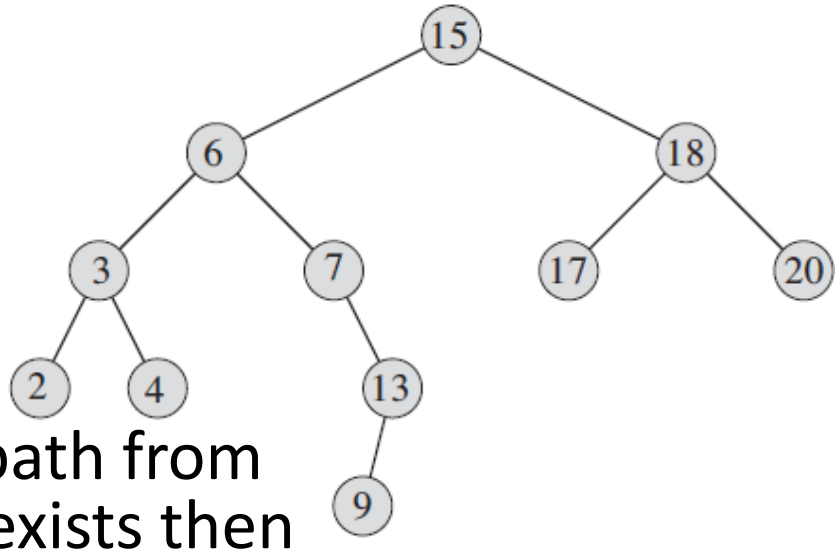
# Binary Search Tree

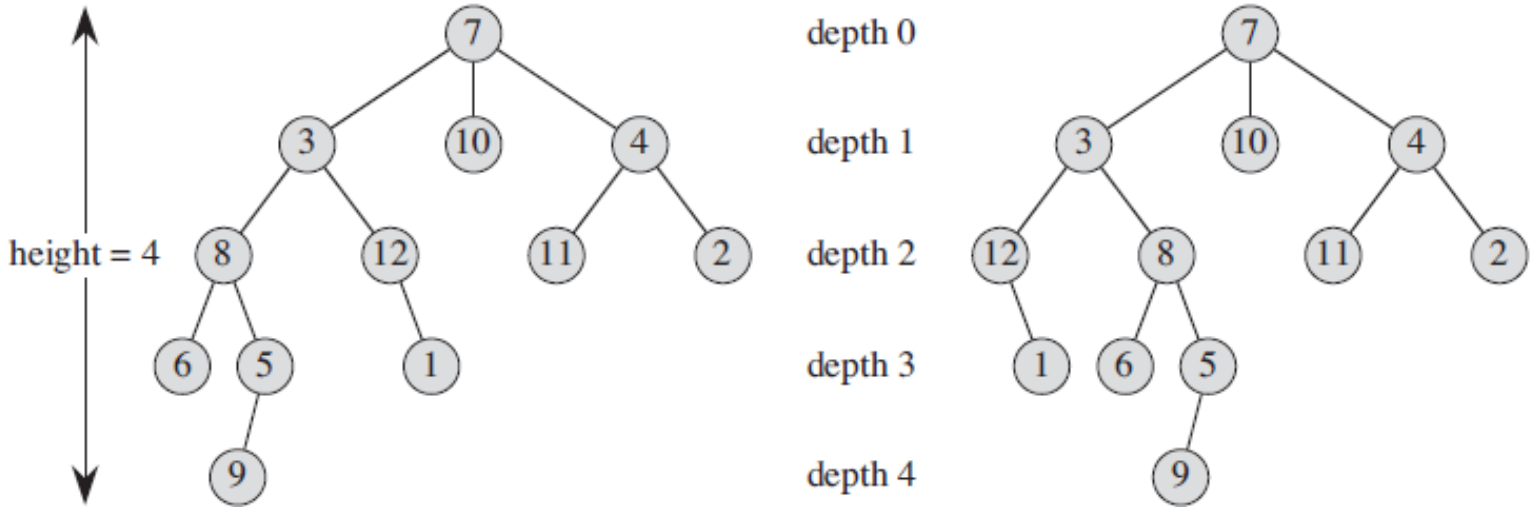
# Terminology



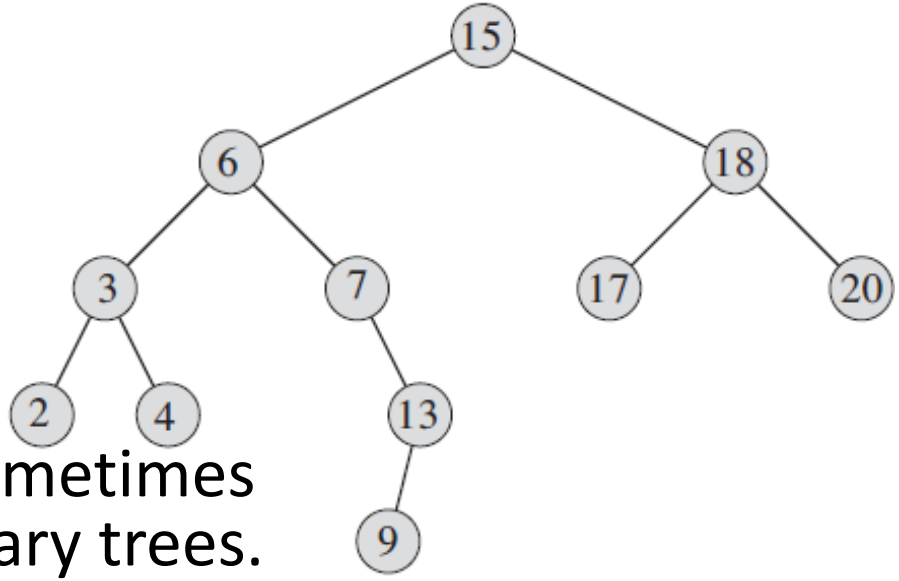

- **Root**: Topmost node in a tree
- **Child** and **Parent**: On a simple path from root to a node, if an edge (x,y) exists then x is the parent of y, and y is a child of x.
- **Siblings**: Nodes with the same parent
- **Descendant**: Node reachable by repeated proceeding from parent to child
- **Ancestor**: 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.



# Introduction

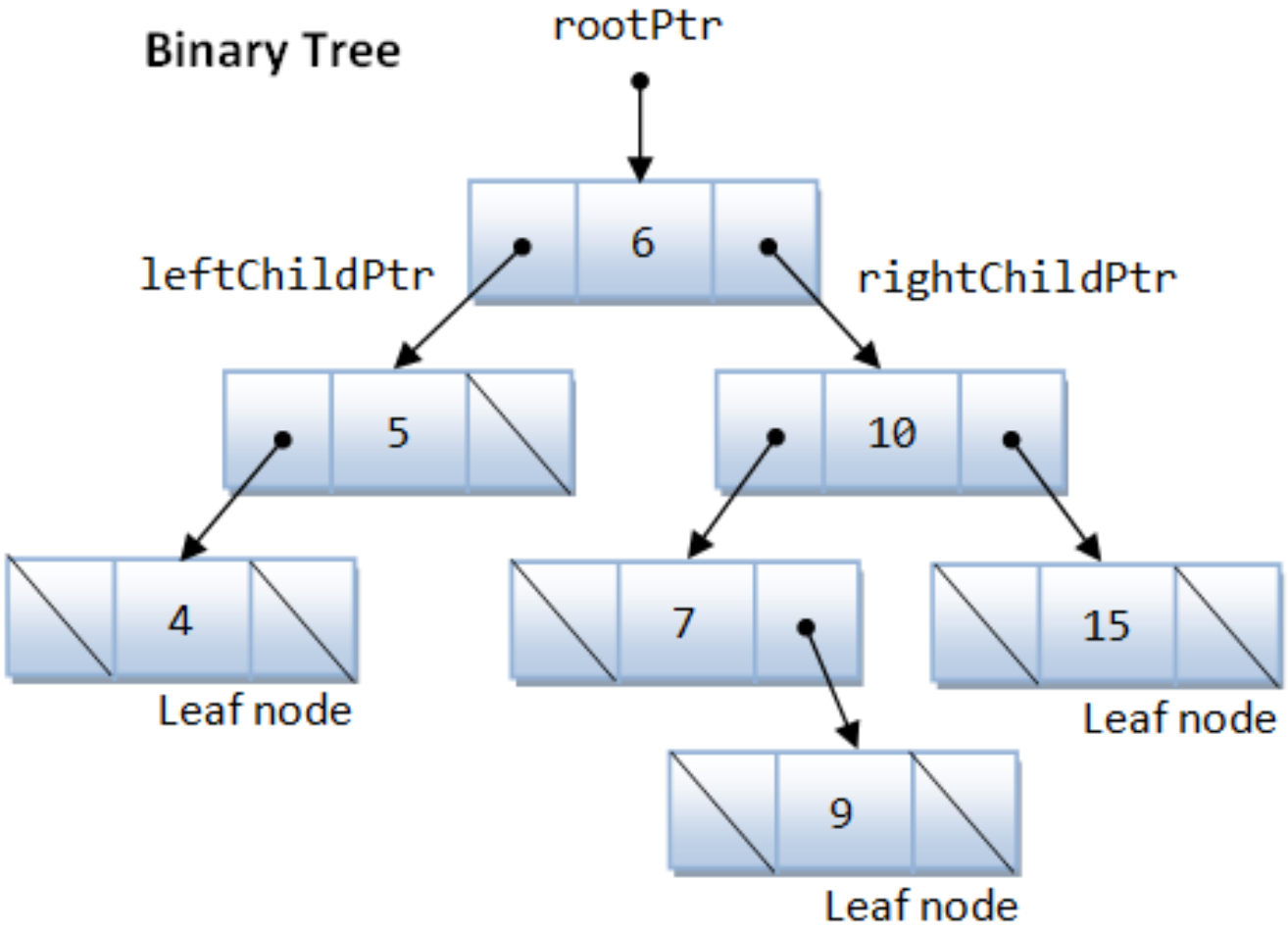



- Binary search trees (BST), sometimes called ordered or sorted binary trees.
- Properties:
  - The left subtree of a node contains only nodes with keys less than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - The left and right subtree 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
6
leftChildPtr
rightChildPtr
5
10
4
7
15
Leaf node
9
Leaf node
Leaf node

# Structure code of a tree node

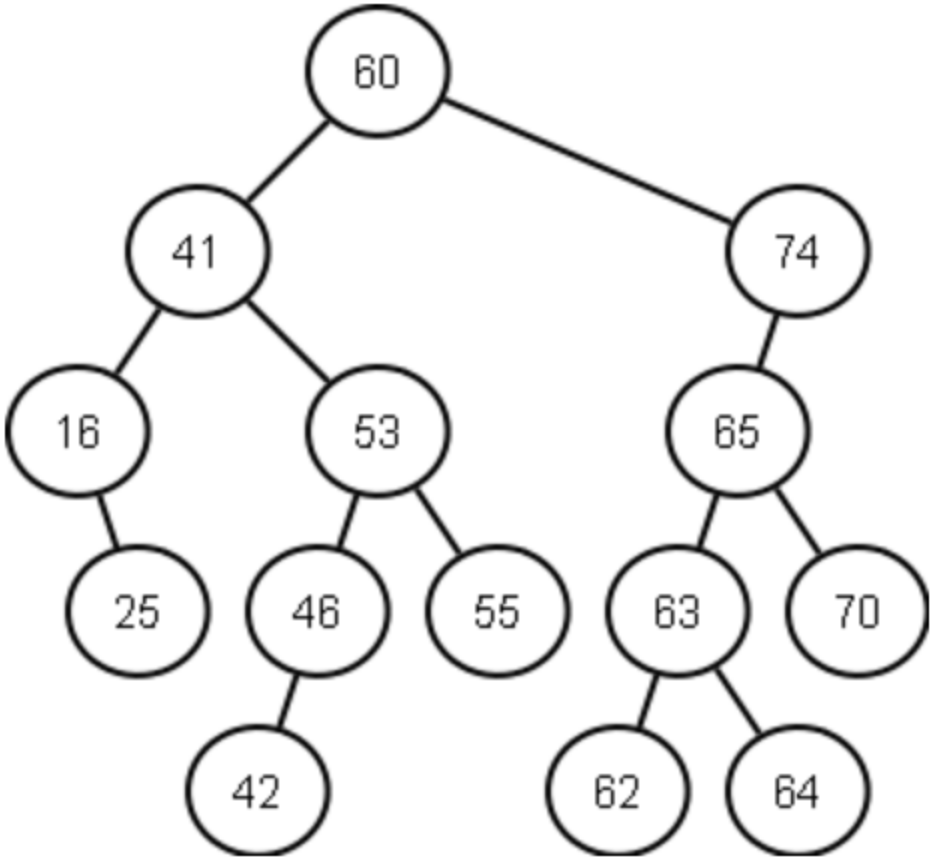
```
struct node
{ int data; //Data element
   struct node * left; //Pointer to left node
   struct node * right; //Pointer to right node
};

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**

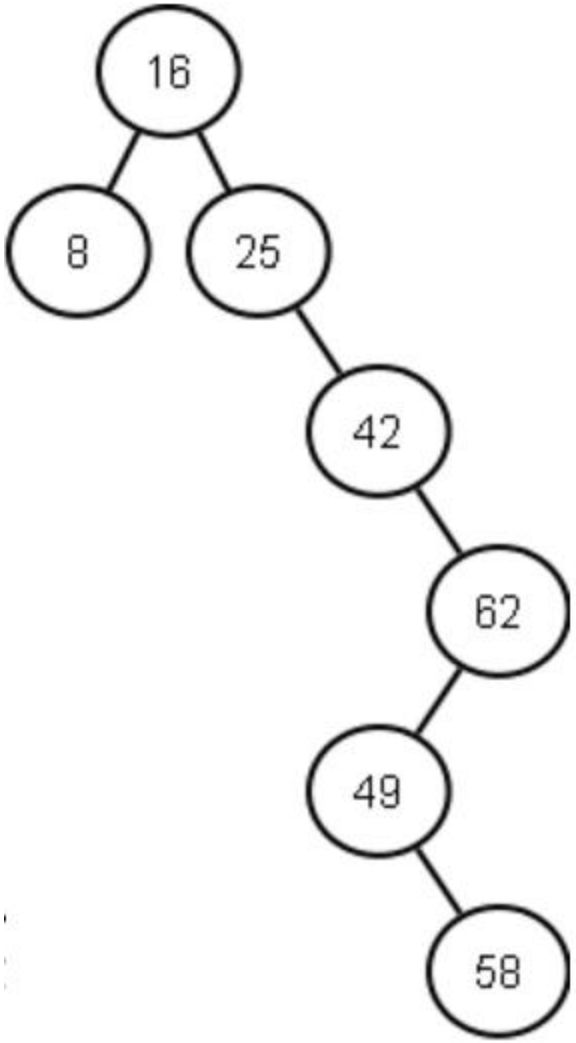
# Traversals

- Preorder traversal

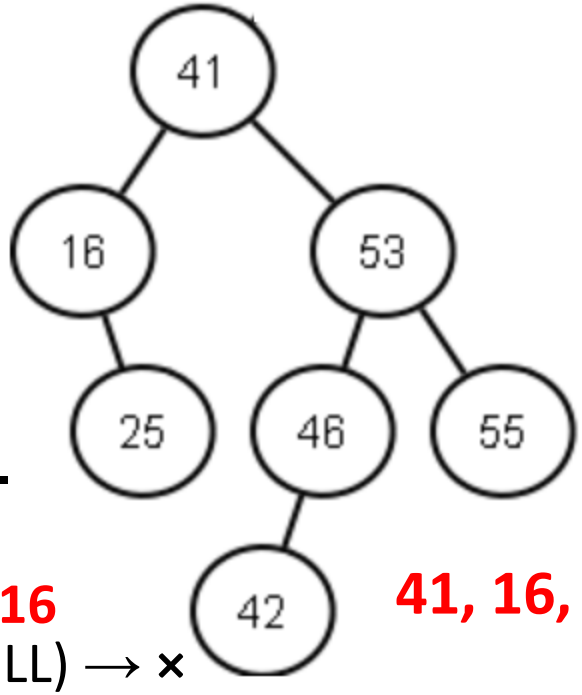
  **16, 8, 25, 42, 62, 49, 58**
- Inorder traversal

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

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

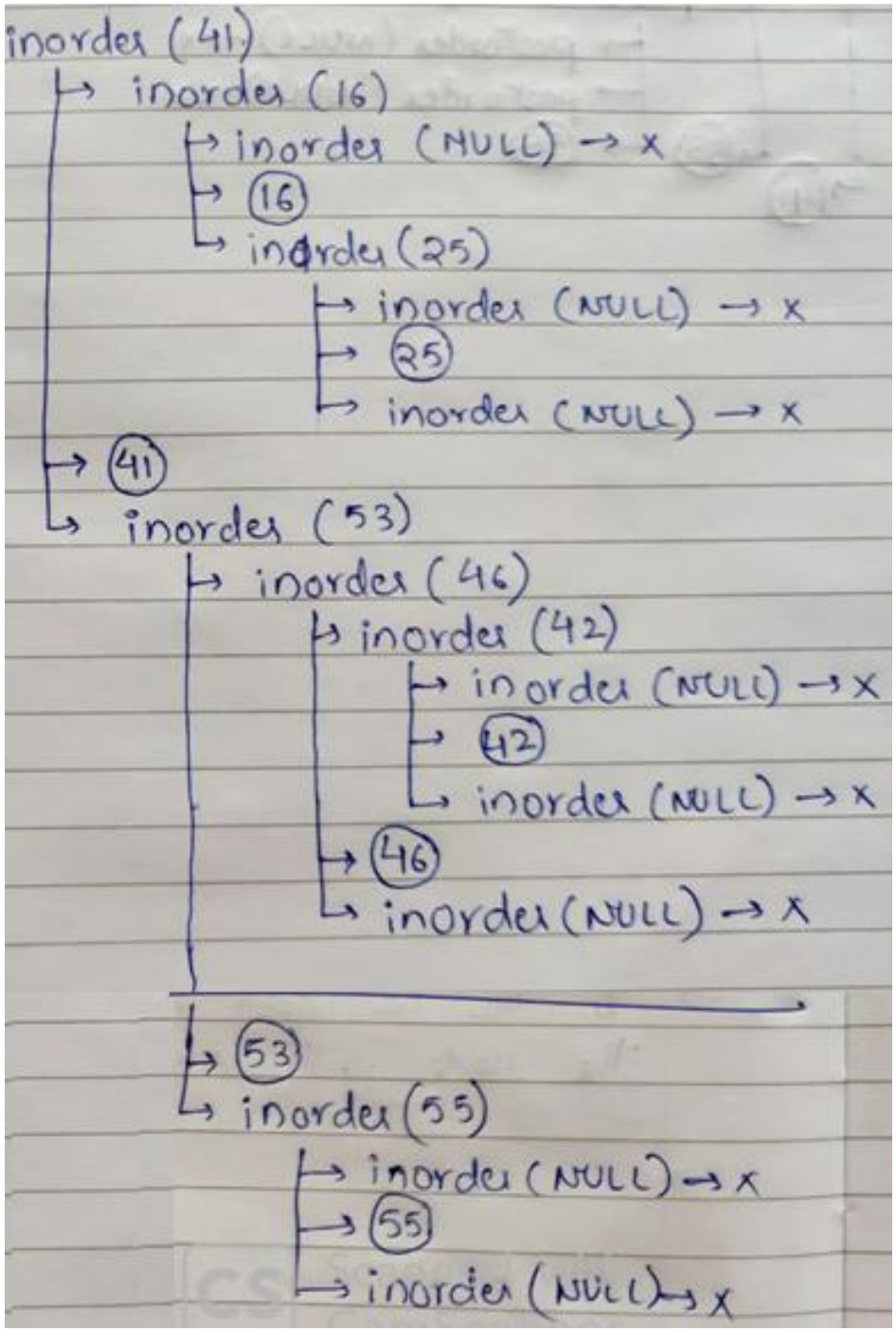


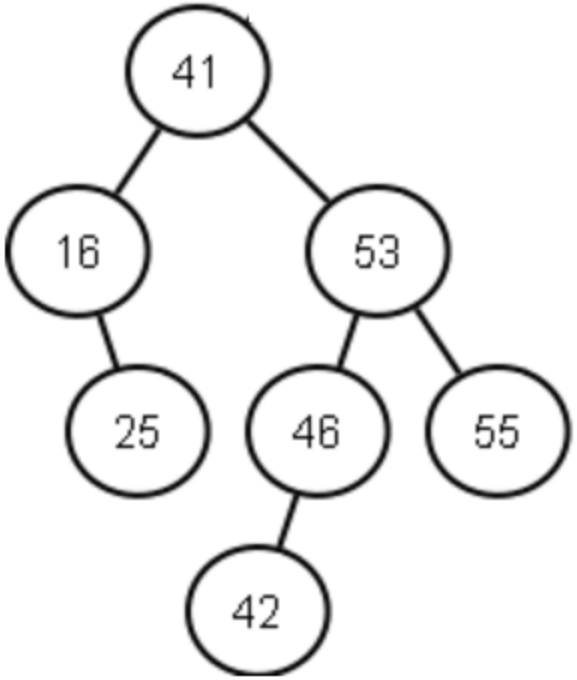
```
void preorder(node *temp)
{  if (temp != NULL)
   {  printf("%d", temp->data);
      preorder(temp->lchild);
      preorder(temp->rchild);
   } }
```




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

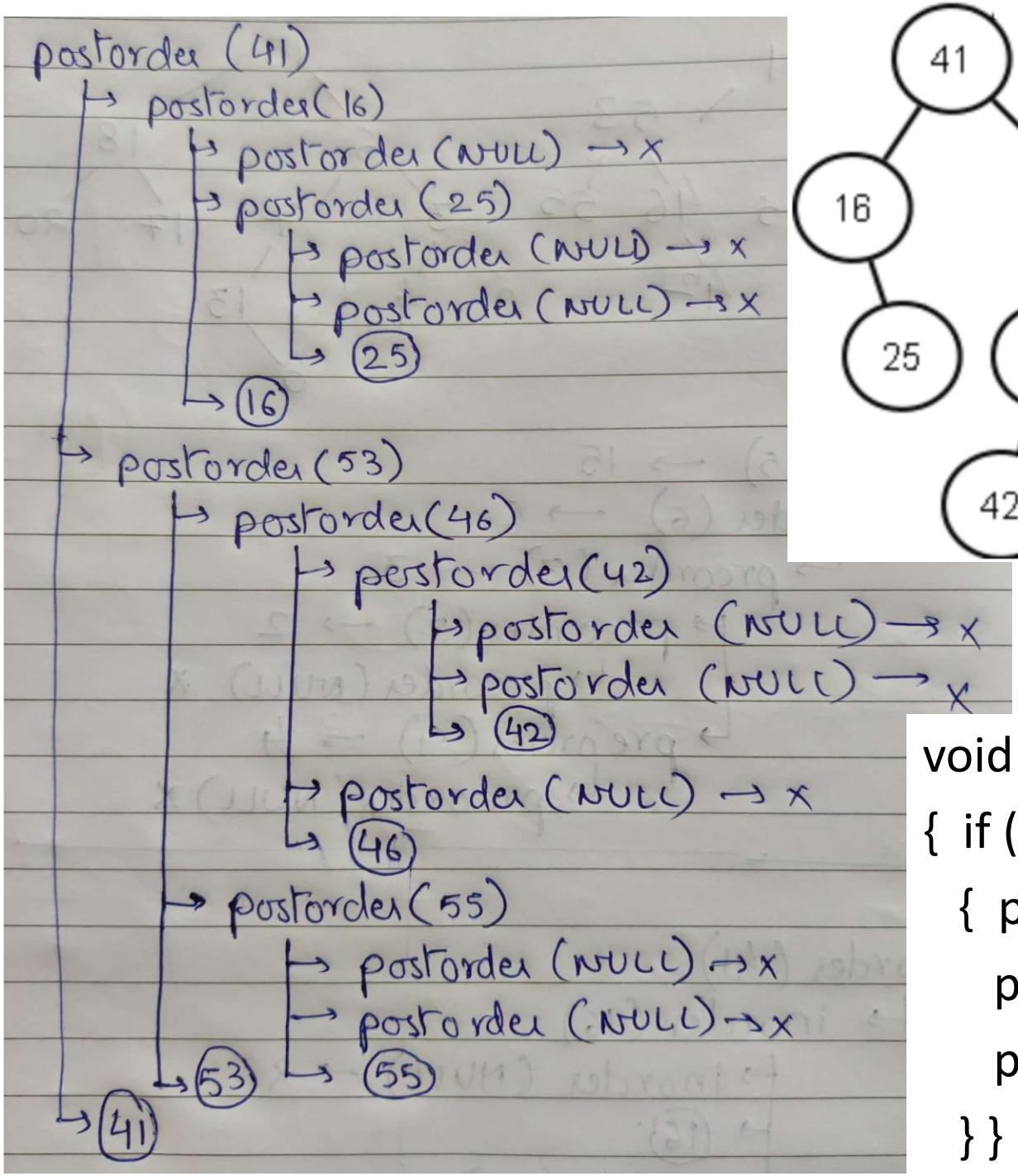
preorder(41) → **41**
- preorder(16) → **16**
  - preorder(NULL) → ×
  - preorder(25) → **25**
    - preorder(NULL) → ×
    - preorder(NULL) → ×
- preorder(53) → **53**
  - preorder(46) → **46**
    - preorder(42) → **42**
      - preorder(NULL) → ×
      - preorder(NULL) → ×
    - preorder(NULL) → ×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

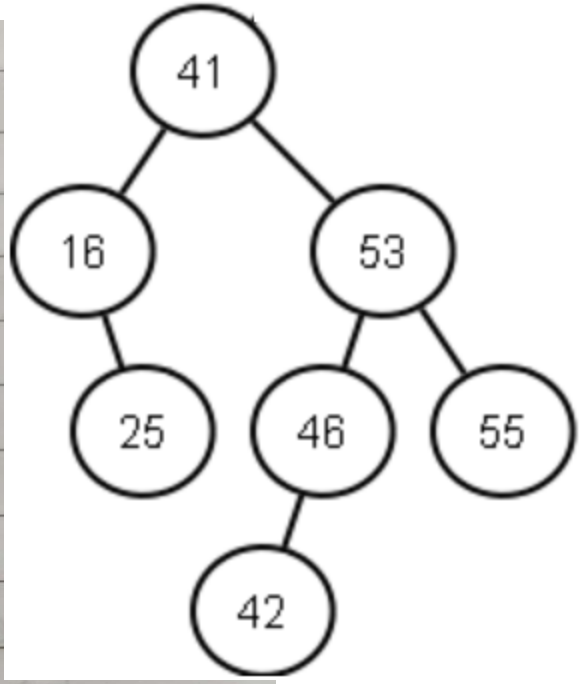



**16, 25, 41, 42, 46, 53, 55**

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

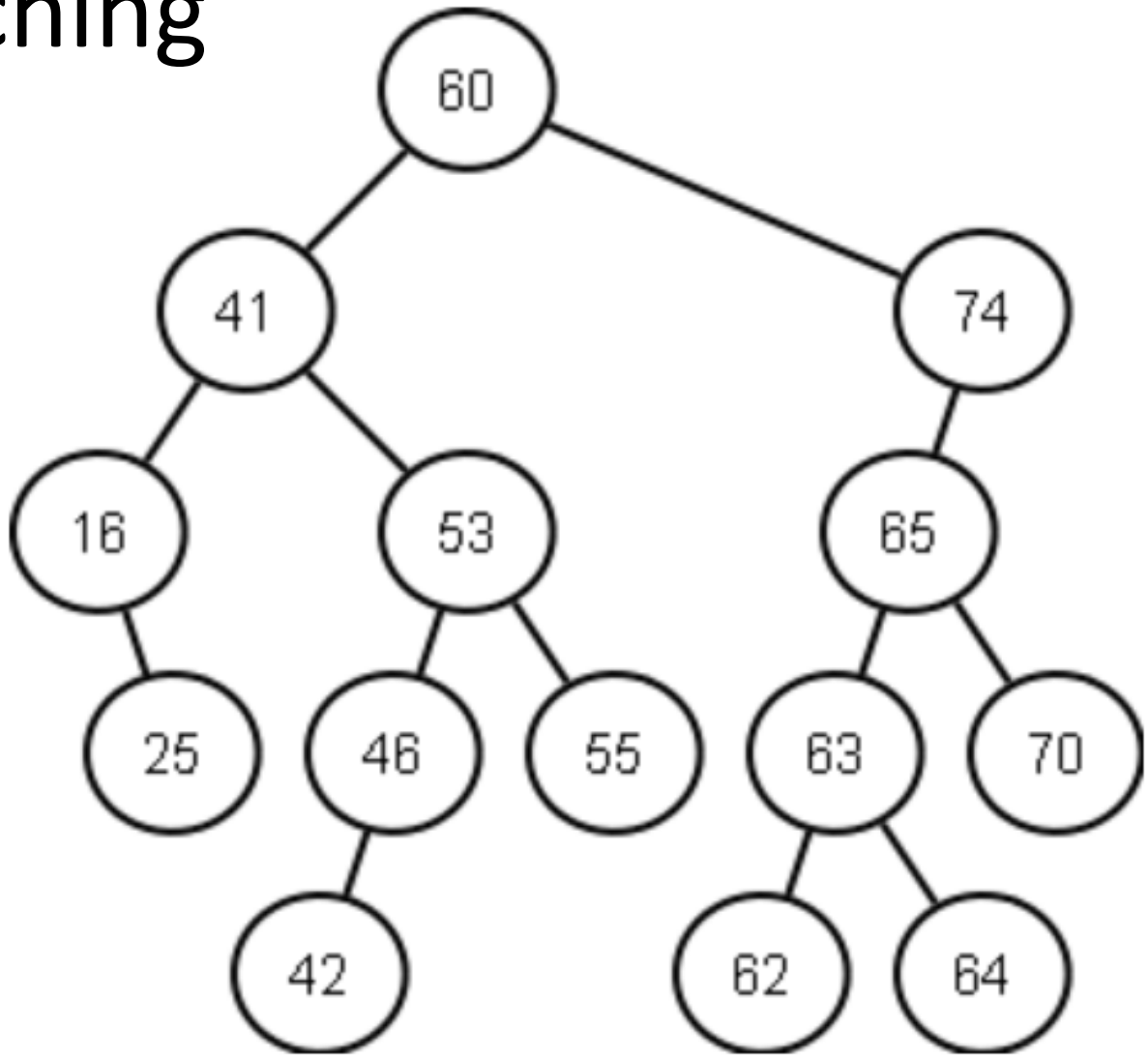



**25, 16, 42, 46, 55, 53, 41**

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

# Searching



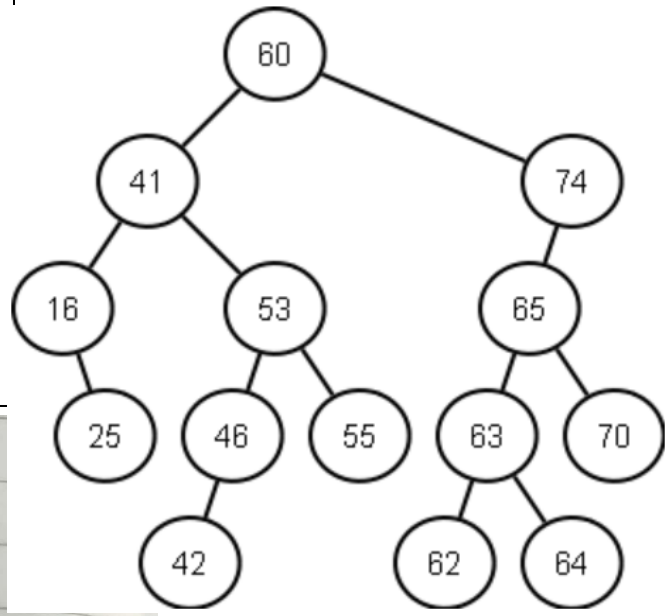
# Contd…

```
TREE-SEARCH(x, k)
1   if x == NIL or k == x.key
2         return x
3   if k < x.key
4         return TREE-SEARCH(x.left, k)
5   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)
1   while x ≠ NIL and k ≠ x.key
2         if k < x.key
3               x = x.left
4         else x = x.right
5   return x
```

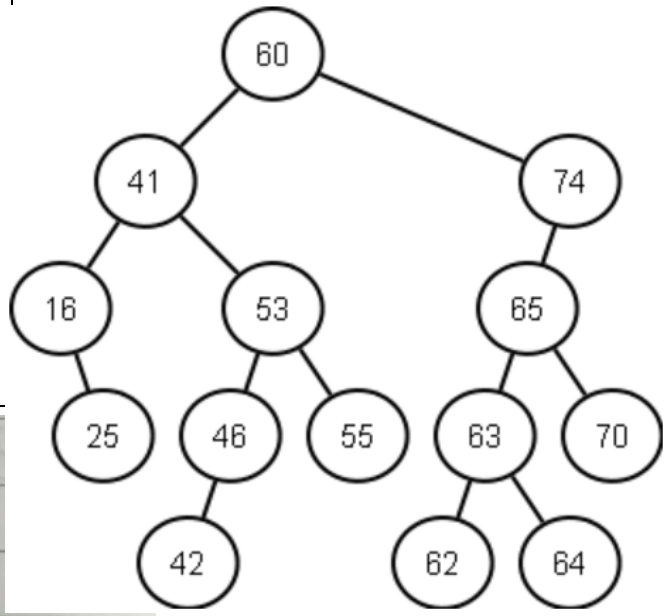
```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2        return x
3  if k < x.key
4        return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```




Tree-Search (root, 100)

| x | K | Execution |
|---|---|---|
| Node 60 | 100 | ① x ≠ NULL or 100 ≠ 60 |
| | | ③ 100 < 60 X |
| | | ⑤ return Tree-Search(Node 74, 100) |
| Node 74 | 100 | ① x ≠ NULL or 74 ≠ 100 |
| | | ③ 100 < 74 X |
| | | ⑤ return Tree-Search(NULL, 100) |
| NULL | 100 | ① x == NULL ✓ |
| | | return (NULL) |

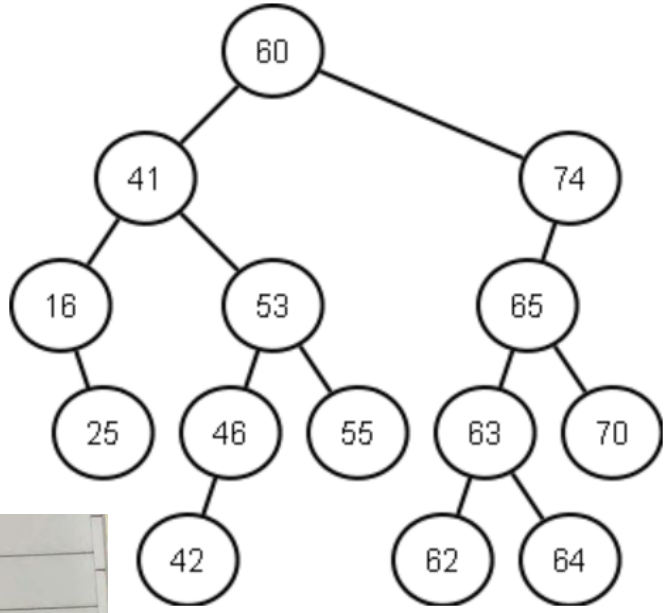
```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2        return x
3  if k < x.key
4        return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```




Tree-Search (root, 53)

| x | K | Execution |
|---|---|---|
| Node 60 | 53 | ① x ≠ NULL or 53 ≠ 60 |
| | | ③ 53 < 60 ✓ |
| | | ④ return Tree-Search (Node 41, 53) |
| Node 41 | 53 | ① x ≠ NULL or 53 ≠ 41 |
| | | ③ 53 < 41 x |
| | | ⑤ return Tree-Search (Node 53, 53) |
| Node 53 | 53 | ① x ≠ NULL or 53 == 53 ✓ |
| | | return (Node 53) |

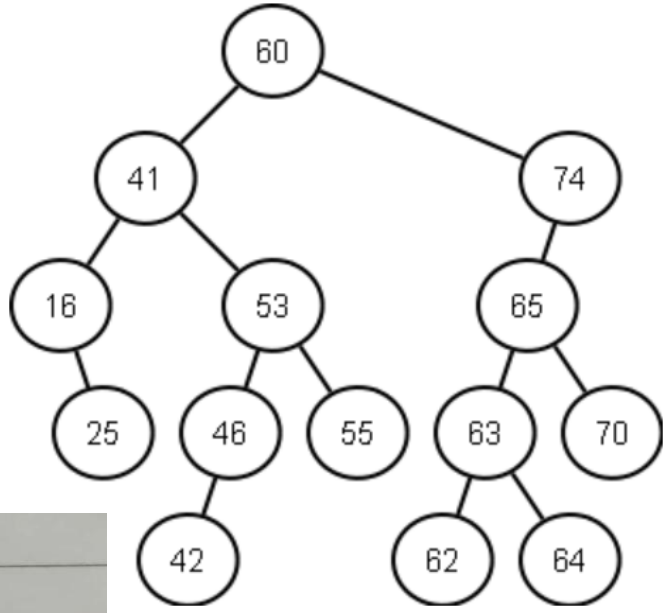
```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```




Iterative - Tree - Search (root, 100)

| x | k | Execution |
|---|---|---|
| Node 60 | 100 | $x \neq$ NULL ✓ $\quad 100 \neq 60$ ✓<br>$100 < 60$ ×<br>$x = x.right$ = Node 74 |
| Node 74 | 100 | $x \neq$ NULL ✓ $\quad 100 \neq 74$ ✓<br>$100 < 74$ ×<br>$x = x.right$ = Node (NULL) |
| NULL | 100 | $x \neq$ NULL ×<br>return (NULL) |

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```



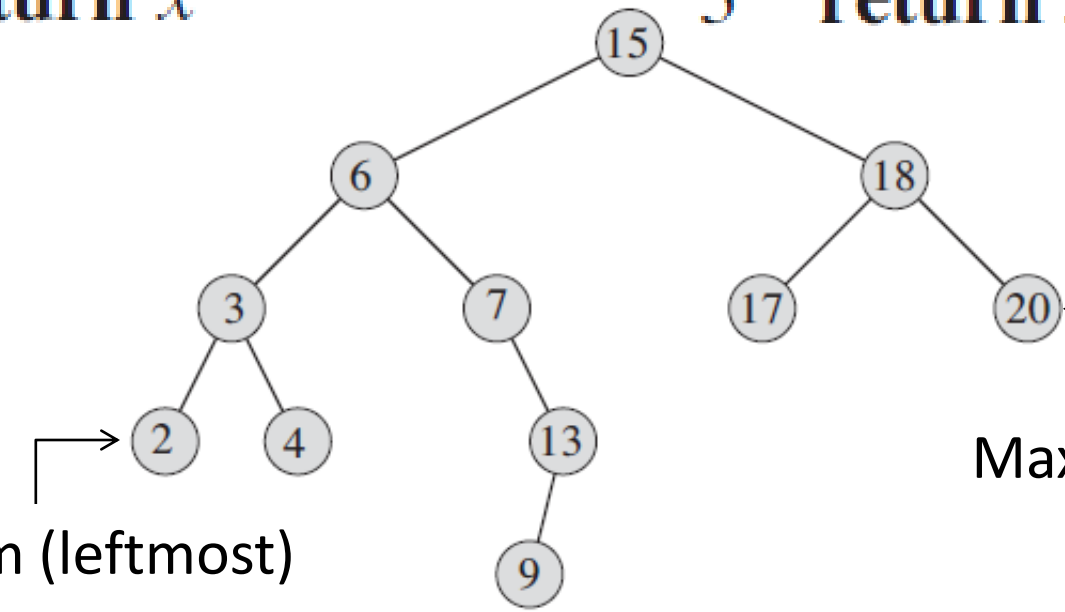

Iterative - Tree - Search (root, 53)

| x | K | Execution |
|---|---|---|
| Node 60 | 53 | x ≠ NULL ✓ 60 ≠ 53 ✓<br>53 < 60 ✓<br>x = x.left = Node 41 |
| Node 41 | 53 | x ≠ NULL ✓ 41 ≠ 53 ✓<br>53 < 41 ×<br>x = x.right = Node 53 |
| Node 53 | 53 | x ≠ NULL ✓ 53 ≠ 53 ×<br>return (Node 53) |

# Minimum and Maximum

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

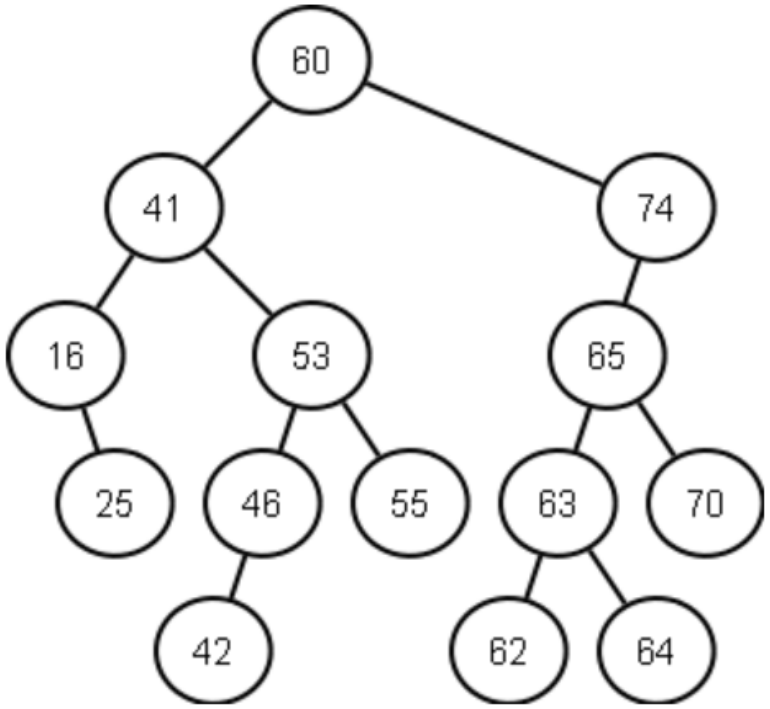
```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```




Minimum (leftmost)

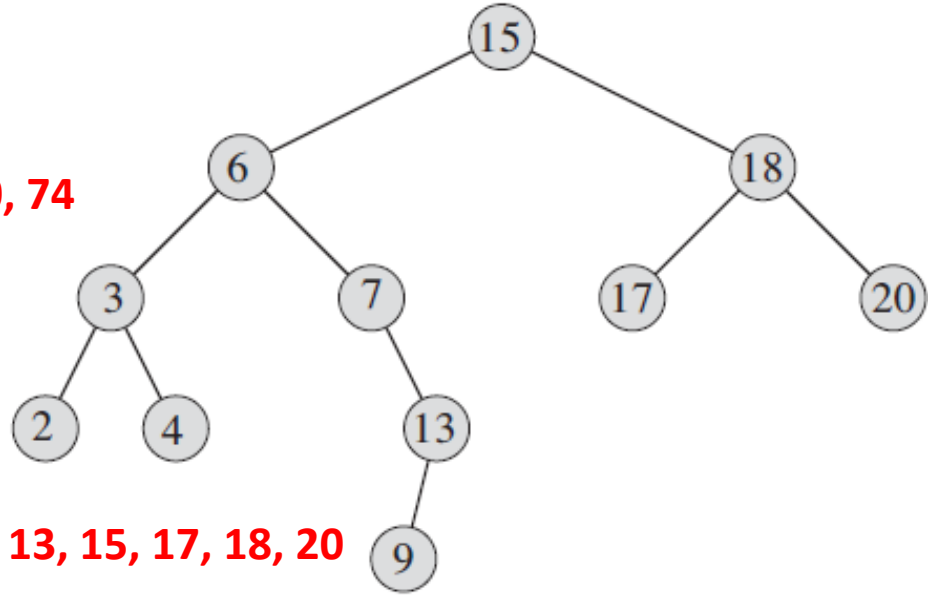
Maximum (rightmost)

# Successor and Predecessor



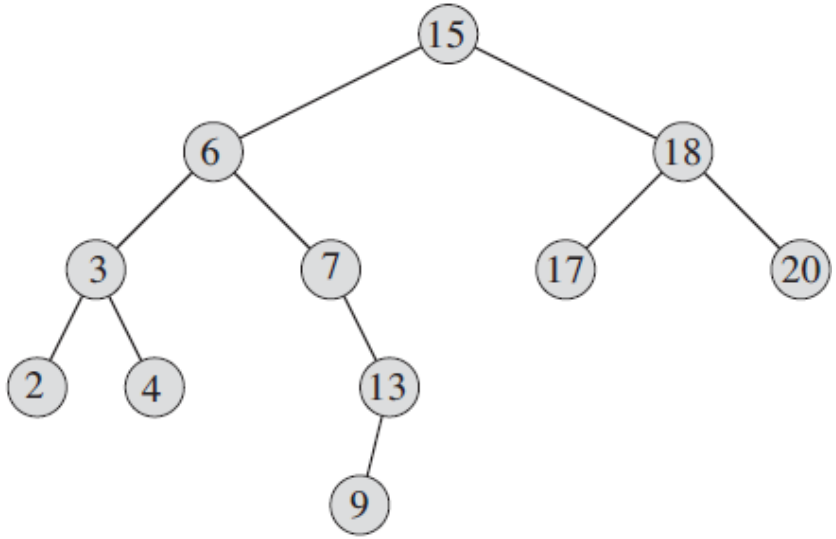

16, 25, 41, 42, 46, 53, 55, 60, 62, 63, 64, 65, 70, 74

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




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

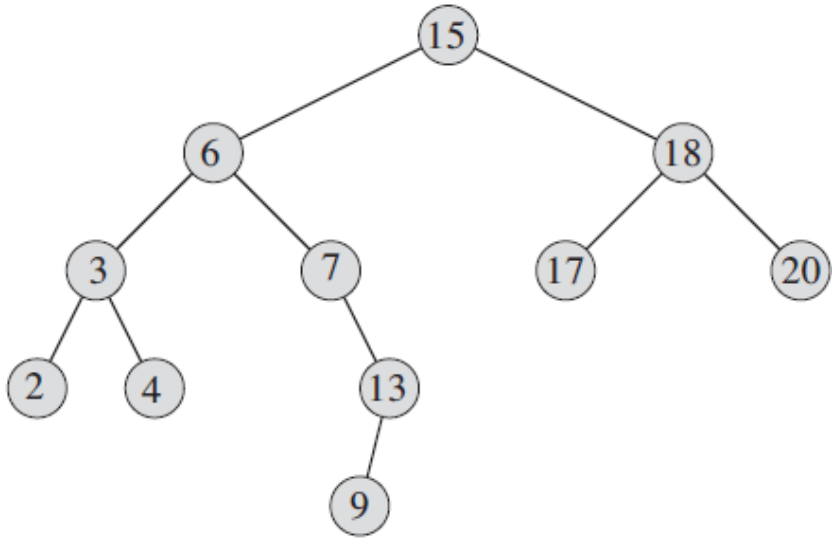
```
TREE-SUCCESSOR(x)
1   if x.right ≠ NIL
2         return TREE-MINIMUM(x.right)
3   y = x.p
4   while y ≠ NIL and x == y.right
5         x = y
6         y = y.p
7   return y
```




Tree-successor(6)

| x | x.right | y | Execution |
|---|---|---|---|
| Node 6 | Node 7 | | ① Node 7 ≠ NULL<br>↳ Tree-Minimum(Node 7)<br>↳ (7) |

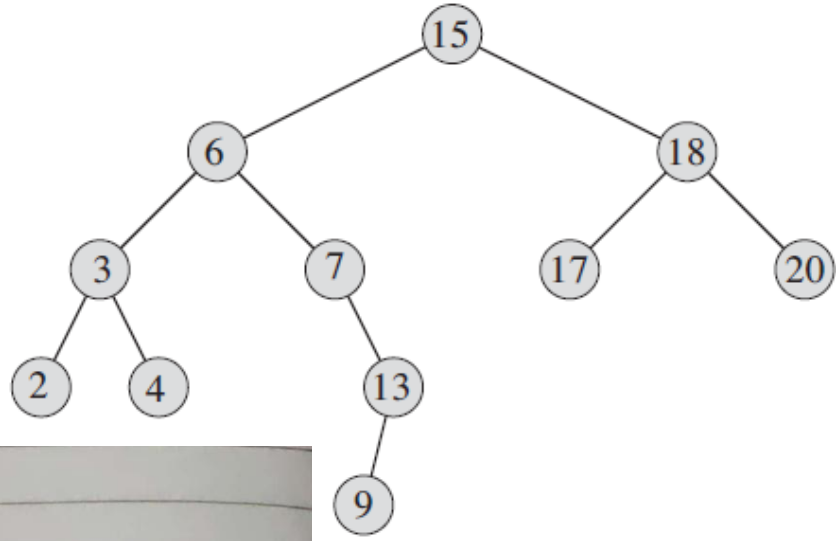
```
TREE-SUCCESSOR(x)
1   if x.right ≠ NIL
2         return TREE-MINIMUM(x.right)
3   y = x.p
4   while y ≠ NIL and x == y.right
5         x = y
6         y = y.p
7   return y
```




Tree-Successor (13)

| x | x.right | y | Execution |
|---|---|---|---|
| Node 13 | NULL | Node 7 | ① NULL ≠ NULL ✗<br>④ Node 7 ≠ NULL and Node 13 == Node 13 |
| Node 7 | | Node 6 | ④ Node 6 ≠ NULL and Node 7 == Node 7 |
| Node 6 | | Node 15 | ④ Node 15 ≠ NULL and Node 6 == Node 18 ✗<br>↳ (15) |

```
TREE-SUCCESSOR(x)
1   if x.right ≠ NIL
2         return TREE-MINIMUM(x.right)
3   y = x.p
4   while y ≠ NIL and x == y.right
5         x = y
6         y = y.p
7   return y
```

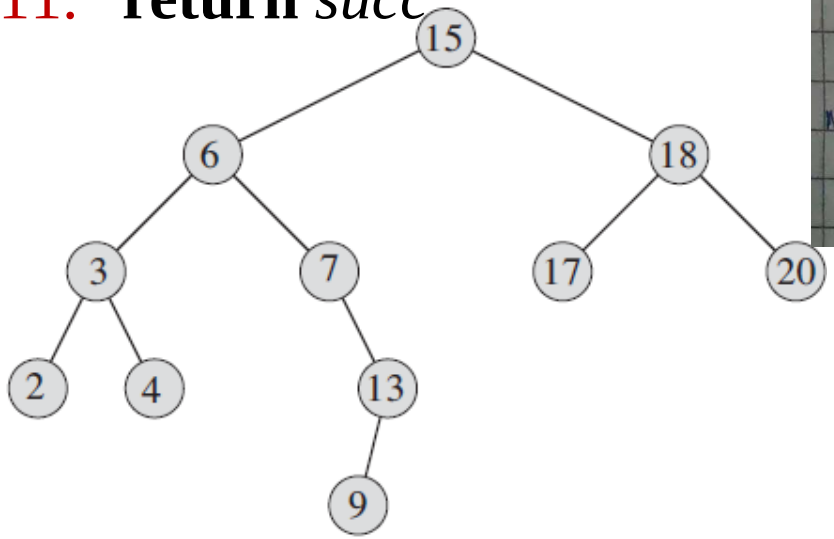



Tree - successor (20)

| x | x.right | y | Execution |
|---|---|---|---|
| Node 20 | NULL | Node 18 | (1) NULL ≠ NULL ✗<br>(4) Node 18 ≠ NULL and Node 20 == Node 20 |
| Node 18 | | Node 15 | (4) Node 15 ≠ NULL and Node 18 == Node 18 |
| node 15 | | NULL | (4) NULL ≠ NULL ✗<br>↳ NULL |

TREE-SUCCESSOR-WITHOUT-PARENT($y$,$x$)

1. **if** $x.right \neq$ NIL
2. **return** TREE-MINIMUM($x.right$)
3. $succ$ = NIL
4. **while** TRUE
5. **if** $x.key < y.key$
6. $succ = y$
7. $y = y.left$
8. **else if** $x.key > y.key$
9. $y = y.right$
10. **else break**
11. **return** $succ$

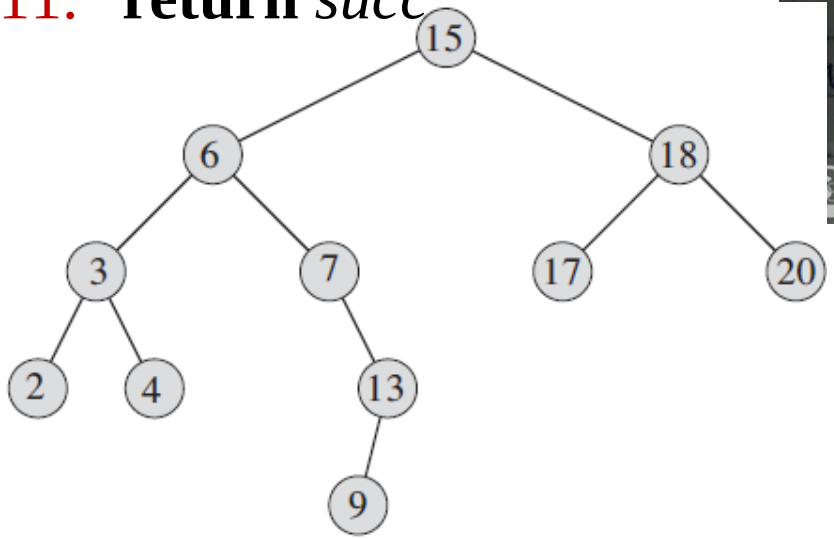



Tree-Successor-Without-Parent (root, 13)

| Succ | y | x | Execution |
|---|---|---|---|
| NULL | Node 15 | Node 13 | 13 < 15 ✓ |
| | | | succ = Node 15, y = Node 6 |
| Node 15 | Node 6 | Node 13 | 13 < 6 x |
| | | | 13 > 6 ✓ |
| | | | y = Node 7 |
| Node 15 | Node 7 | Node 13 | 13 < 7 x |
| | | | 13 > 7 ✓ |
| | | | y = Node 13 |
| Node 15 | Node 13 | Node 13 | 13 < 13 x |
| | | | 13 > 13 x |
| | | | break ; return (15) |

TREE-SUCCESSOR-WITHOUT-PARENT($y$,$x$)

1. **if** $x.right \neq$ NIL
2. **return** TREE-MINIMUM($x.right$)
3. $succ$ = NIL
4. **while** TRUE
5. **if** $x.key < y.key$
6. $succ = y$
7. $y = y.left$
8. **else if** $x.key > y.key$
9. $y = y.right$
10. **else break**
11. **return** $succ$



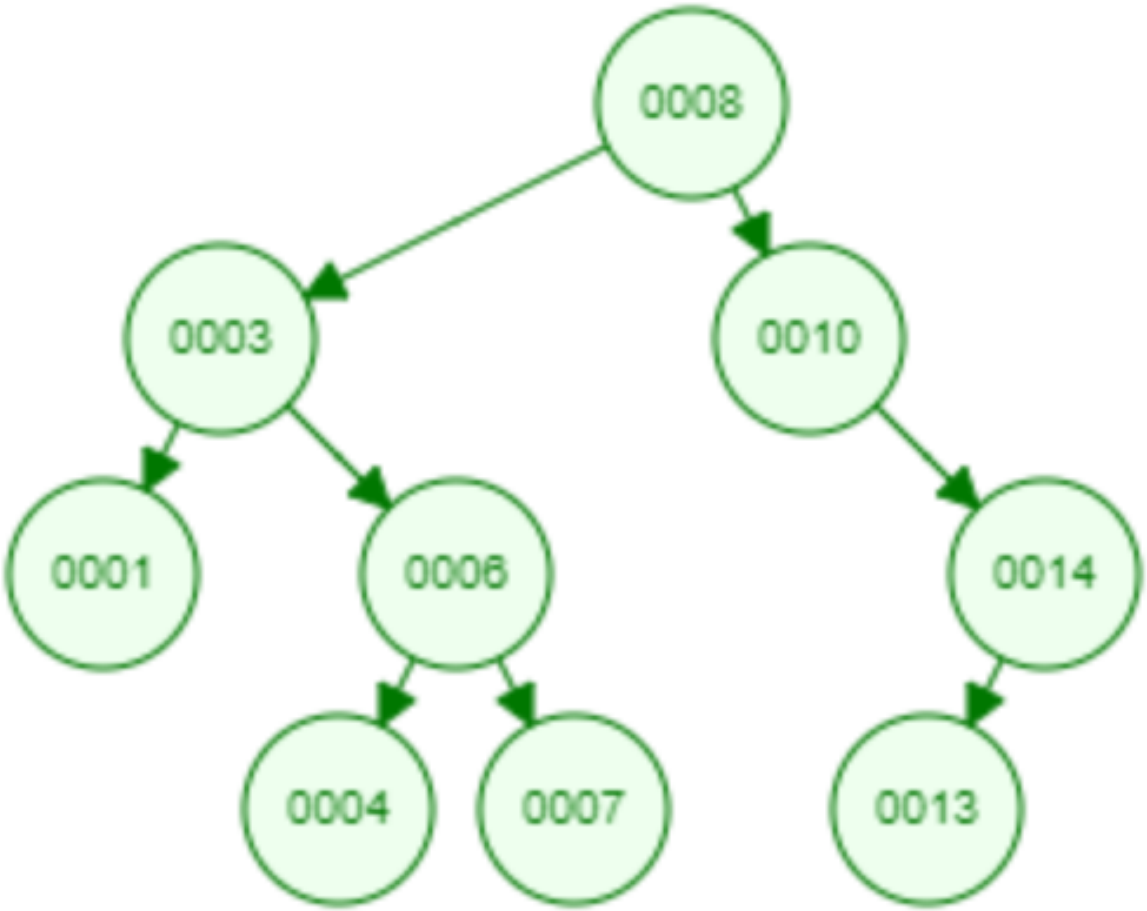

Tree - Successor - without - Parent (root, 20)

| succ | y | x | Execution |
|---|---|---|---|
| NULL | Node 15 | Node 20 | 20 < 15 ✗<br>20 > 15 ✓<br>y = Node 18 |
| NULL | Node 18 | Node 20 | 20 < 18 ✗<br>20 > 18 ✓<br>y = Node 20 |
| NULL | Node 20 | Node 20 | 20 < 20 ✗<br>20 > 20 ✗<br>break; return NULL |

# 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



TREE-INSERT-WITHOUT-PARENT(*T.root*,*z*)

1. *x* = *T.root*
2. **if** *x* == NIL
3. **return** *z*
4. **if** *z.key* < *x.key*
5. *x.left* = TREE-INSERT-WITHOUT-PARENT(*x.left*,*z*)
6. **else if** *z.key* > *x.key*
7. *x.right* = TREE-INSERT-WITHOUT-PARENT(*x.right*,*z*)
8. **return** *x*

# Contd…

```
TREE-INSERT(T, z)
 1   y = NIL
 2   x = T.root
 3   while x ≠ NIL
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == NIL
10         T.root = z          // tree T was empty
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```

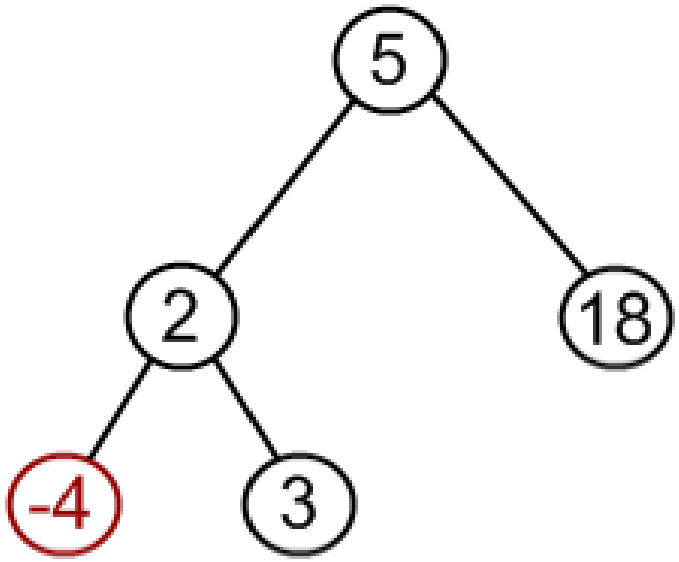
# Deletion

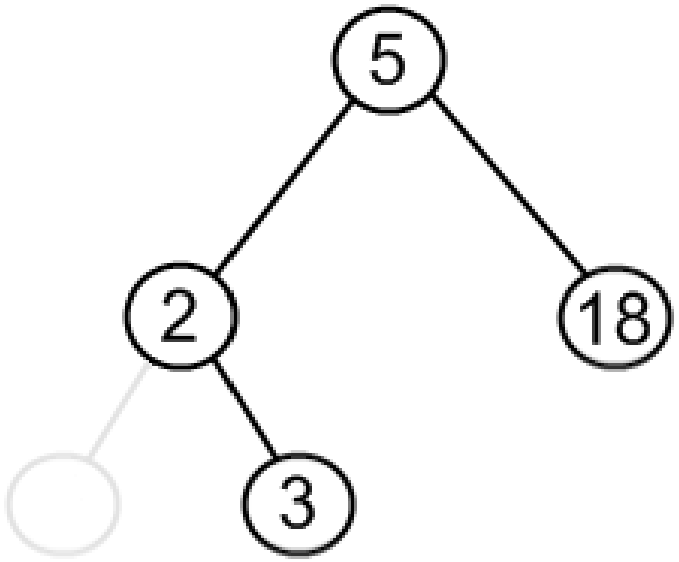
1. Search for a node to remove;
2. If the node is found, execute the remove algorithm.

- Three cases,
  i. Node to be removed has no children.
  ii. Node to be removed has one child.
  iii. Node to be removed has two children.

# Node to be removed has no children

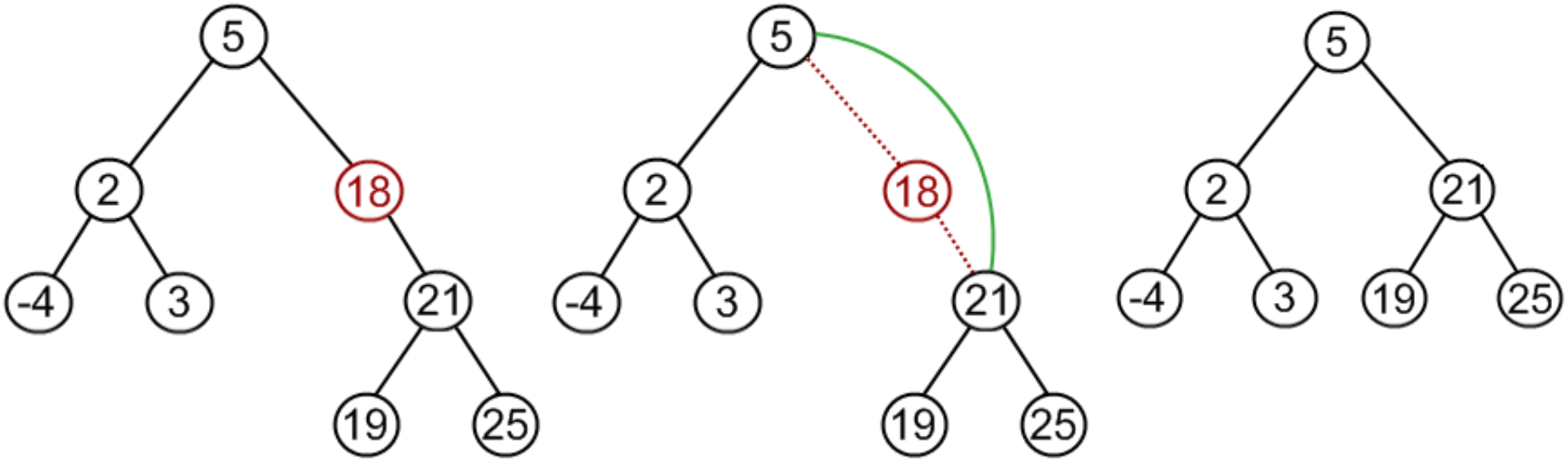
- Set corresponding link of the parent to NULL and disposes the node.
- Example. Remove -4 from a BST.



# Node to be removed has one child

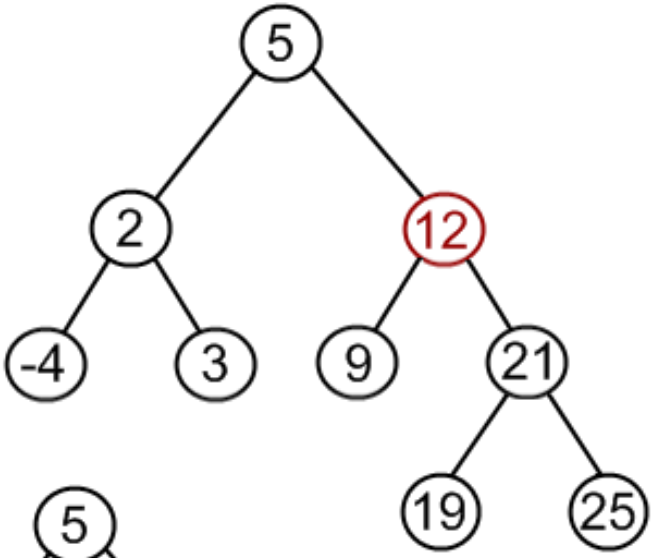
- Link single child (with it's subtree) directly to the parent of the removed node.
- Example. Remove 18 from a BST.

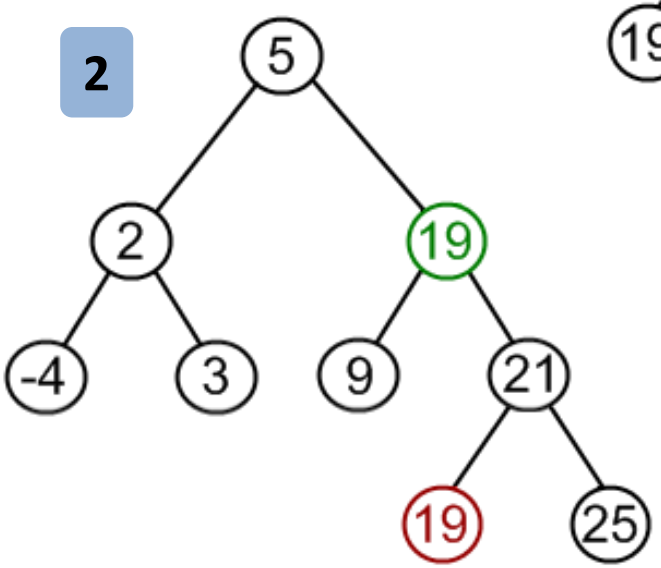


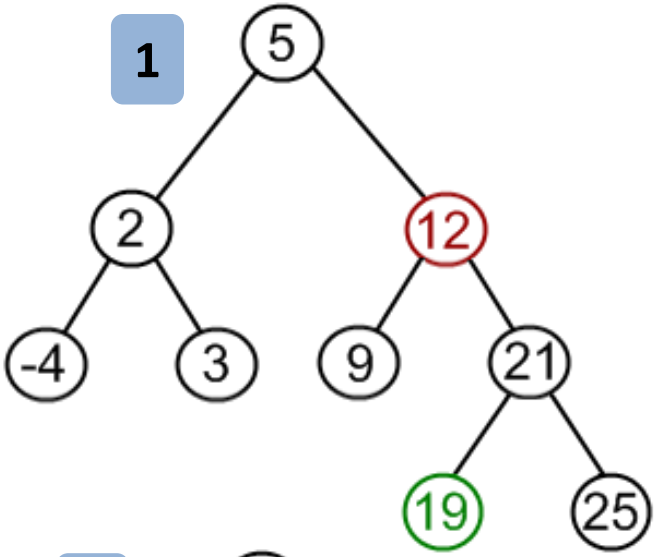
# Node to be removed has two children

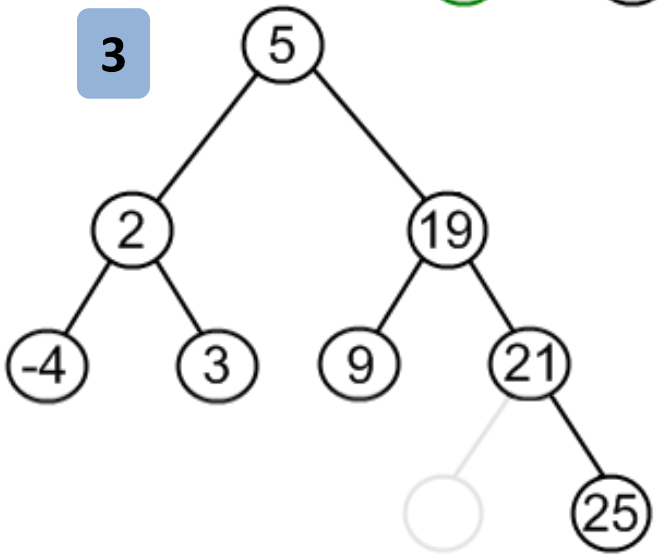
- Find inorder successor, i.e. find the minimum value in right child of the node.
- Copy contents of the inorder successor (or found minimum) to the node being removed with.
- Delete the inorder successor (or found minimum) from the right subtree.
- Note:
  - The node with minimum value has no left child 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.

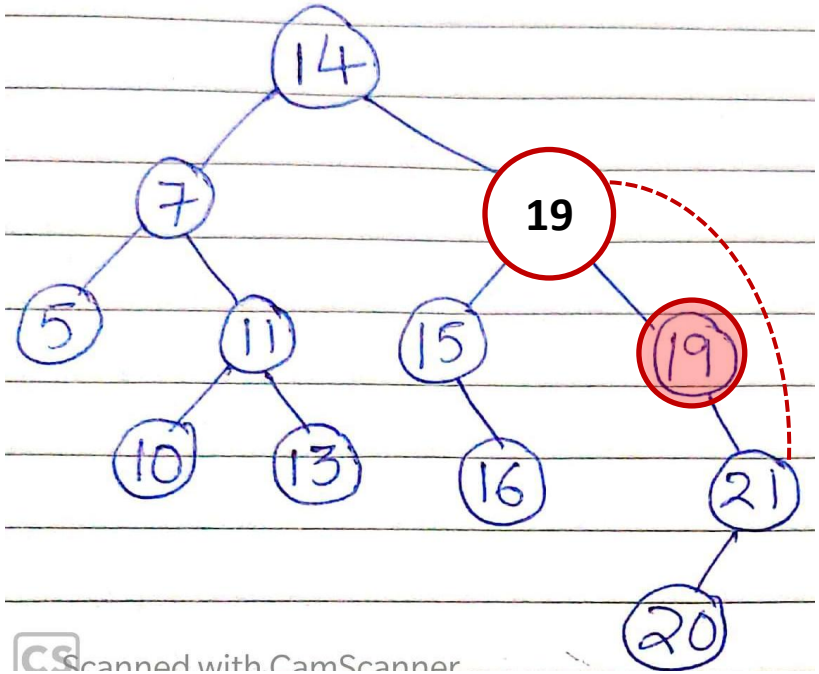


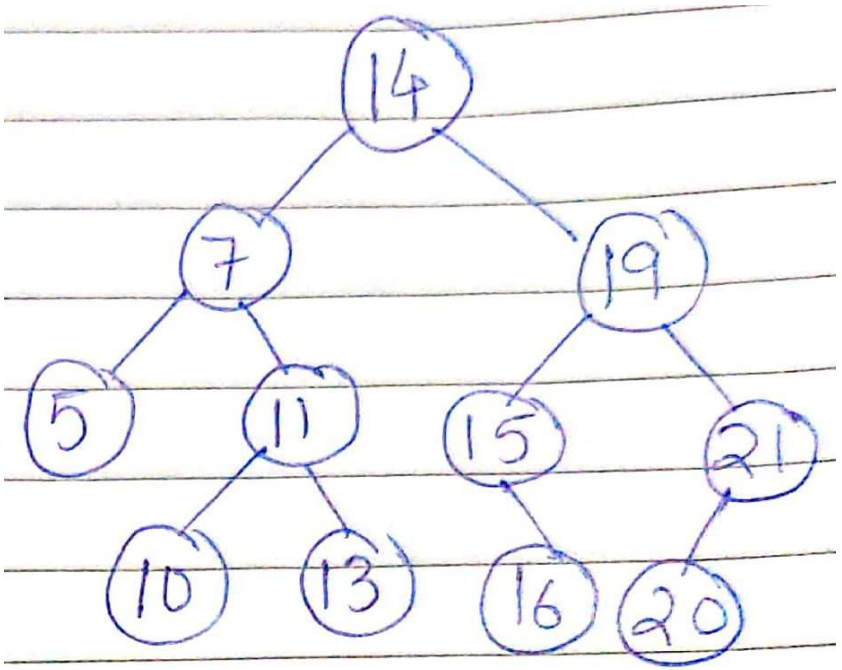



# Delete 17





```
TRANSPLANT(T, u, v)
1   if u.p == NIL
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else u.p.right = v
6   if v ≠ NIL
7       v.p = u.p
```

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

```
TREE-DELETE-WITHOUT-PARENT(T.root,k)
1.    x = T.root
2.    if x == NIL
3.        return x
4.    if k < x.key
5.        x.left = TREE-DELETE-WITHOUT-PARENT(x.left,k)
6.    else if k > x.key
7.        x.right = TREE-DELETE-WITHOUT-PARENT(x.right,k)
8.    else
9.        if x.left == NIL
10.                temp = x.right
11.                delete x
12.                return temp
13.       else if x.right == NIL
14.                temp = x.left
15.                delete x
16.                return temp
17.       temp = TREE-MINIMUM(x.right)
18.       x.key = temp.key
19.       x.right = TREE-DELETE-WITHOUT-PARENT(x.right,temp.key)
20.   return x
```

# 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.

# Inorder and Preorder

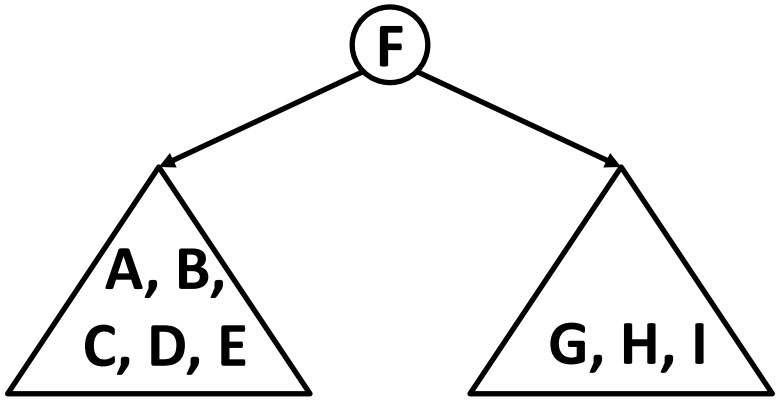
Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]

# Inorder and Preorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

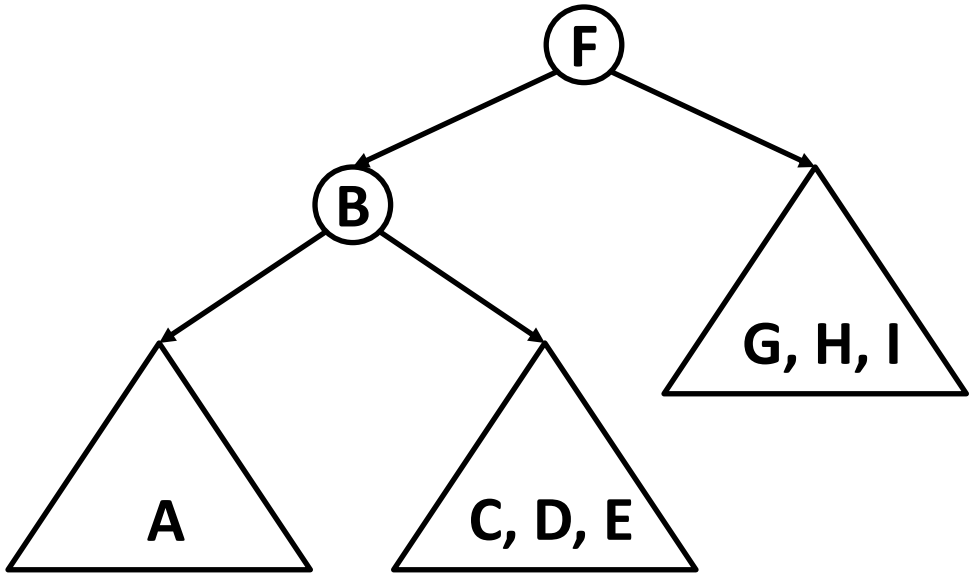
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



# Inorder and Preorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

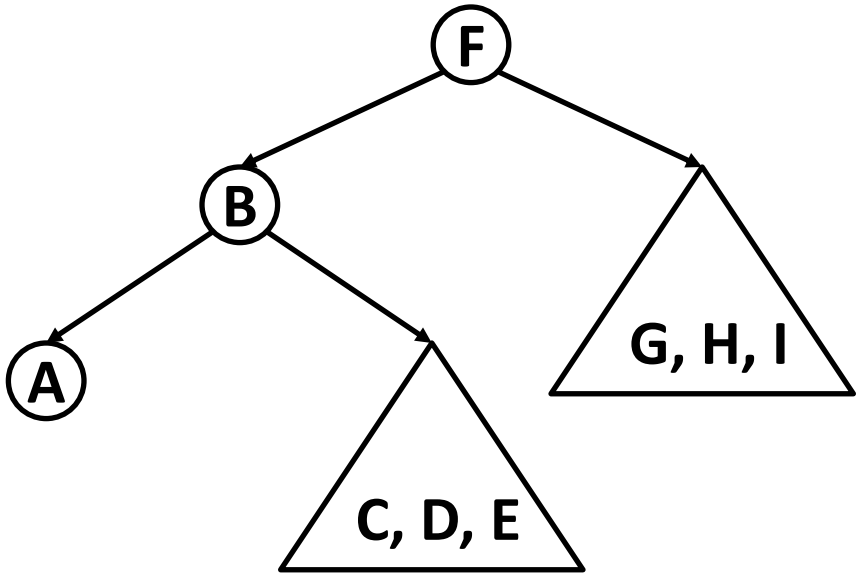
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



# Inorder and Preorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

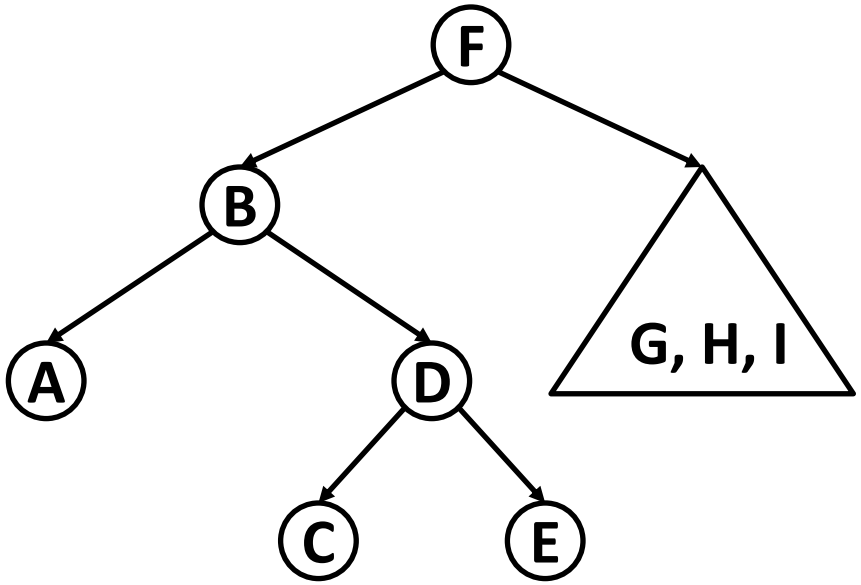
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



# Inorder and Preorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

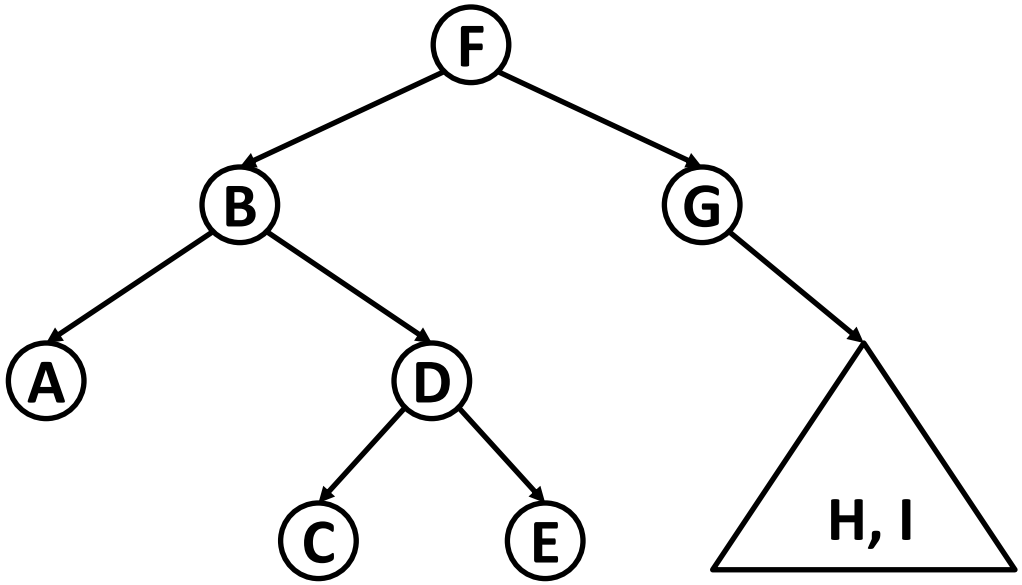
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



# Inorder and Preorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

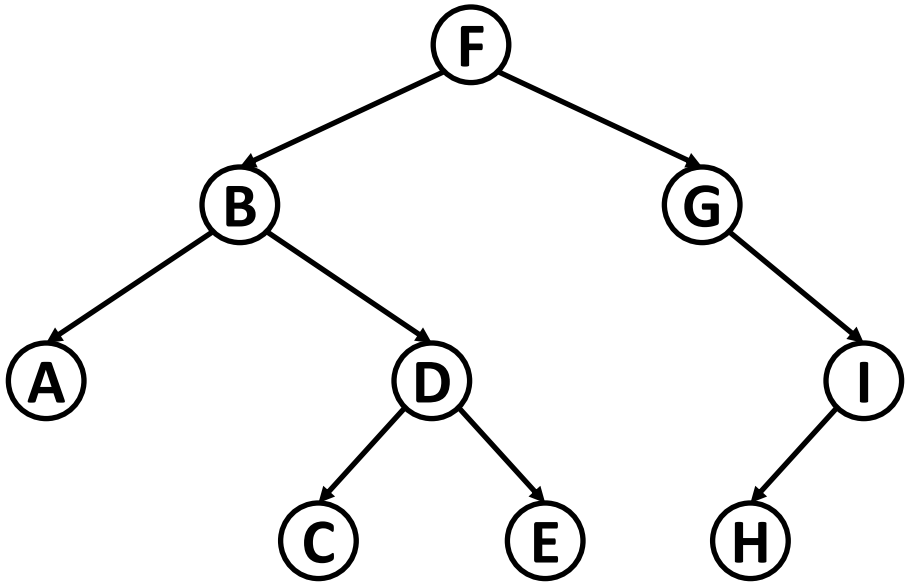
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



# Inorder and Preorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



# Inorder and Postorder

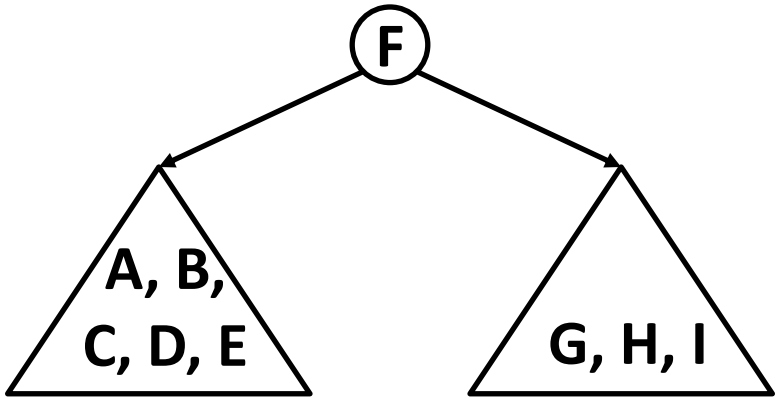
Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]

# Inorder and Postorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

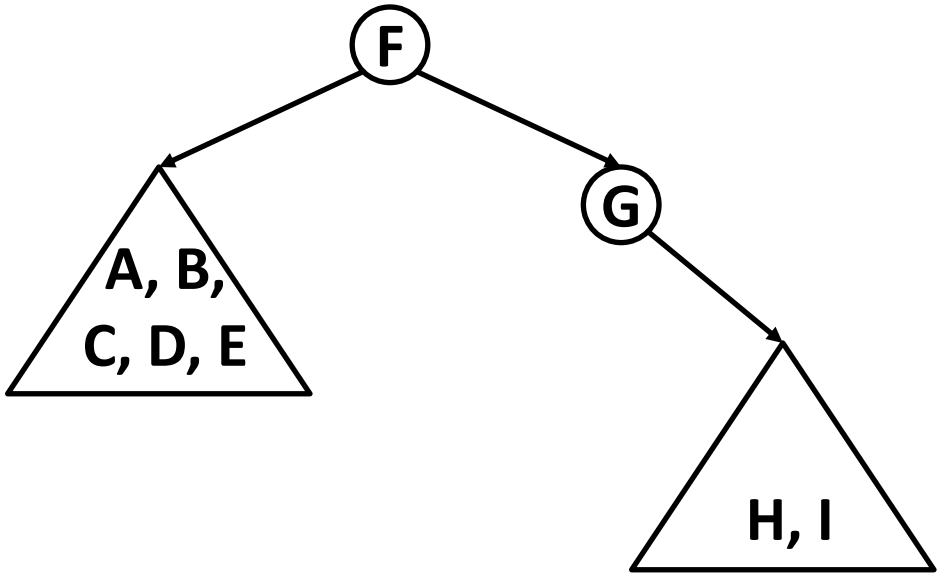
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Inorder and Postorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

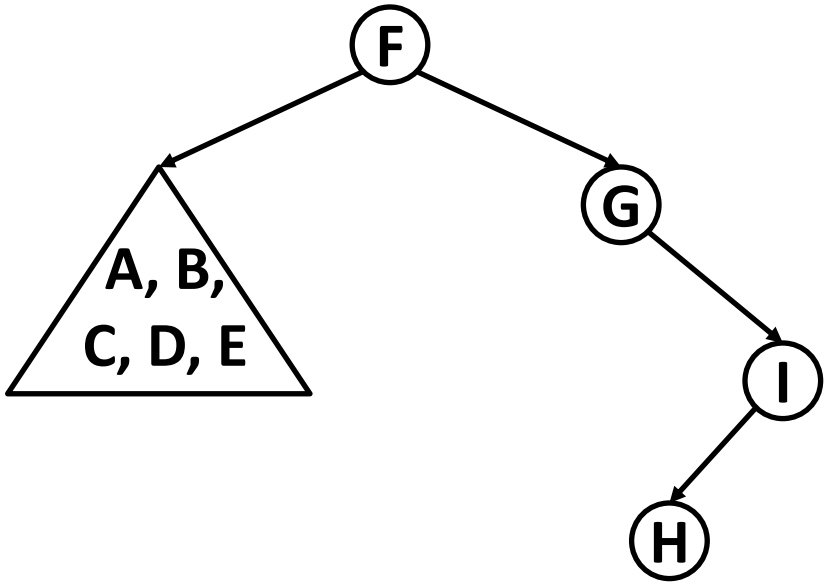
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Inorder and Postorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

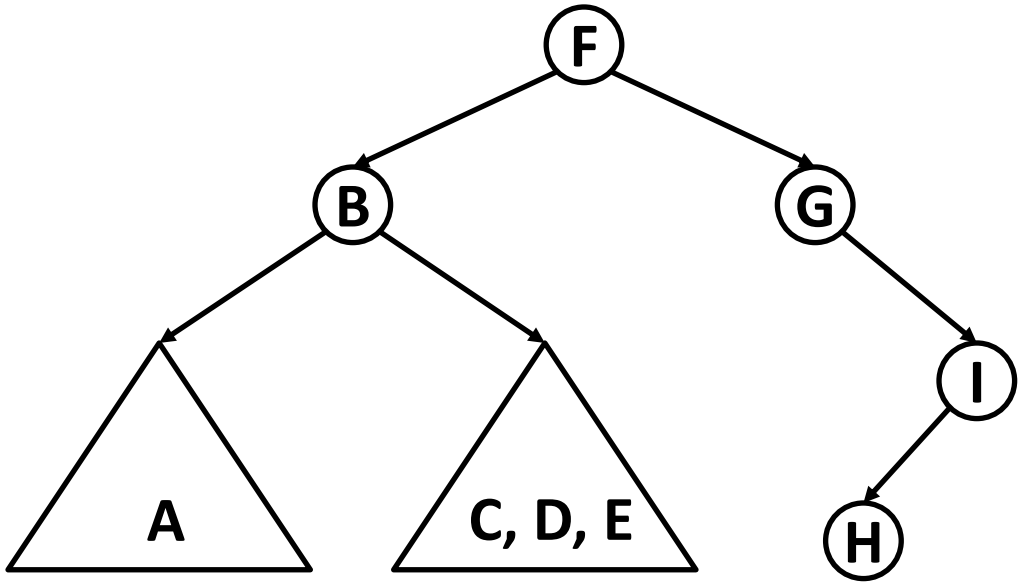
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Inorder and Postorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

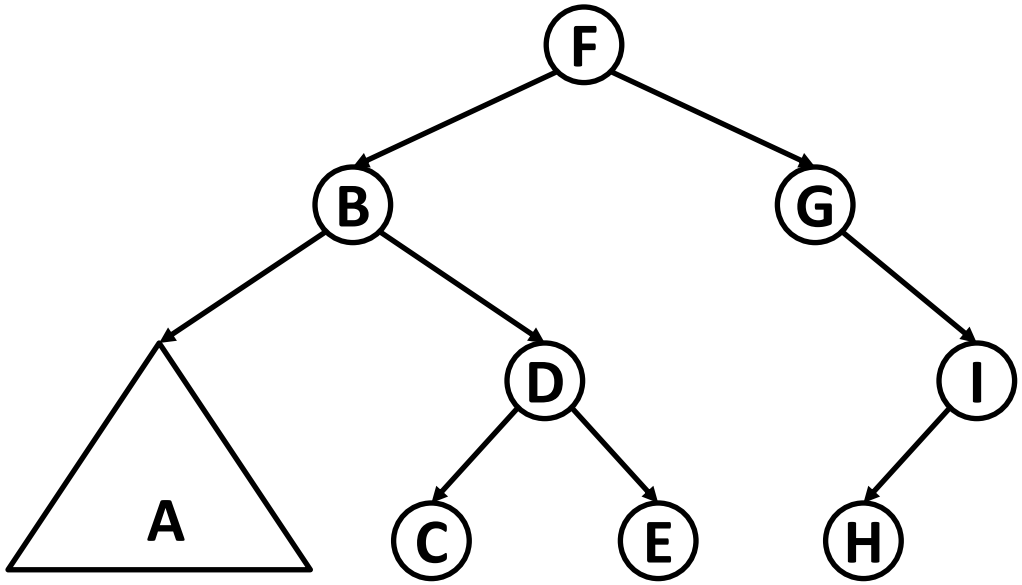
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Inorder and Postorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

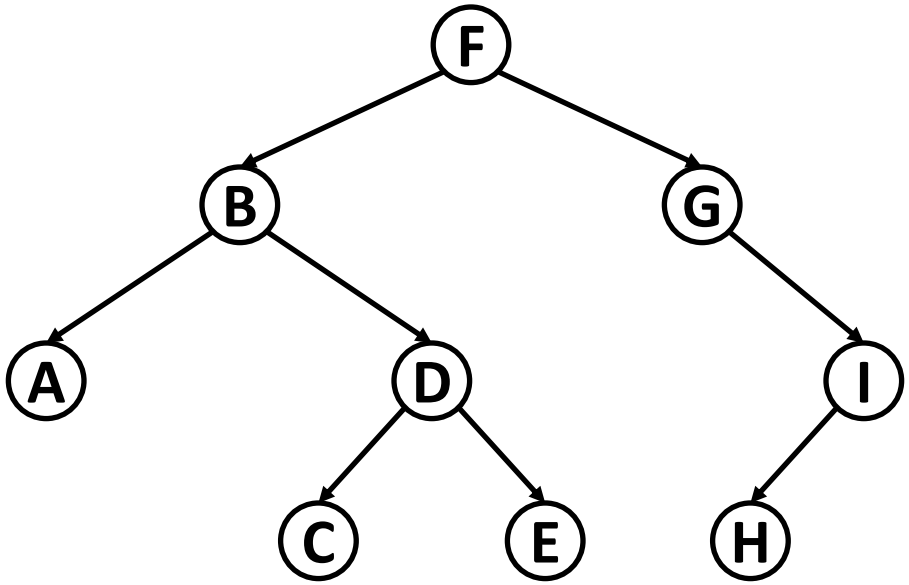
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Inorder and Postorder

Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Preorder and Postorder

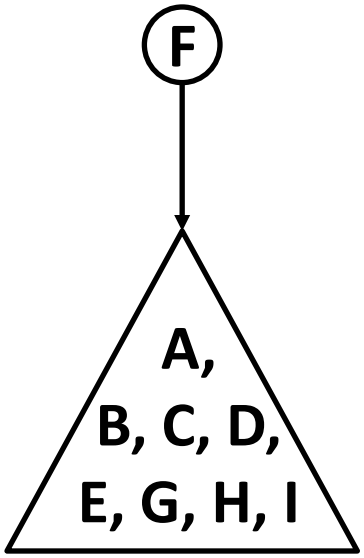
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]

Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]

# Preorder and Postorder

Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]

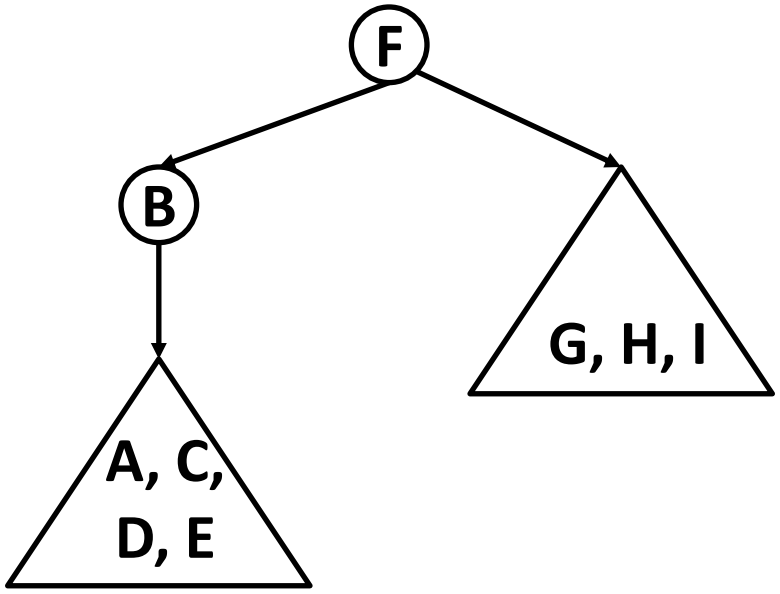
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Preorder and Postorder

Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]

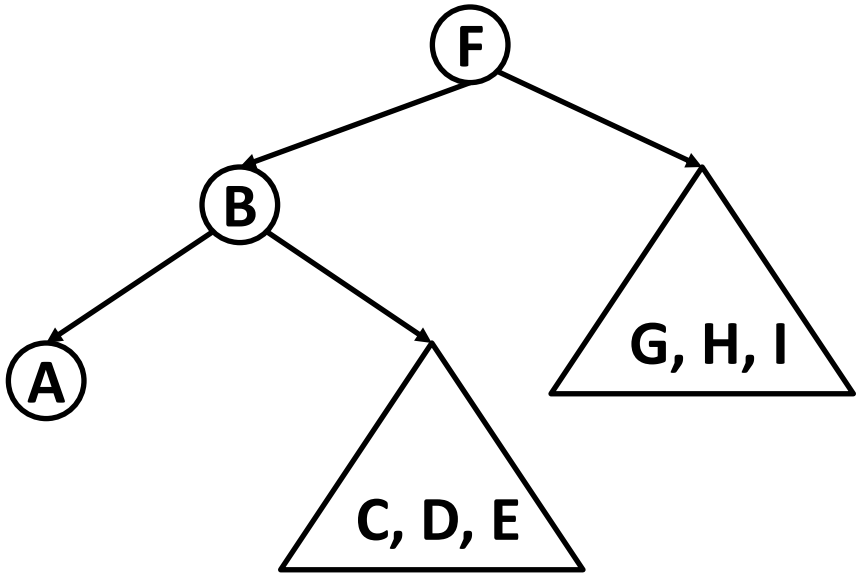
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Preorder and Postorder

Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]

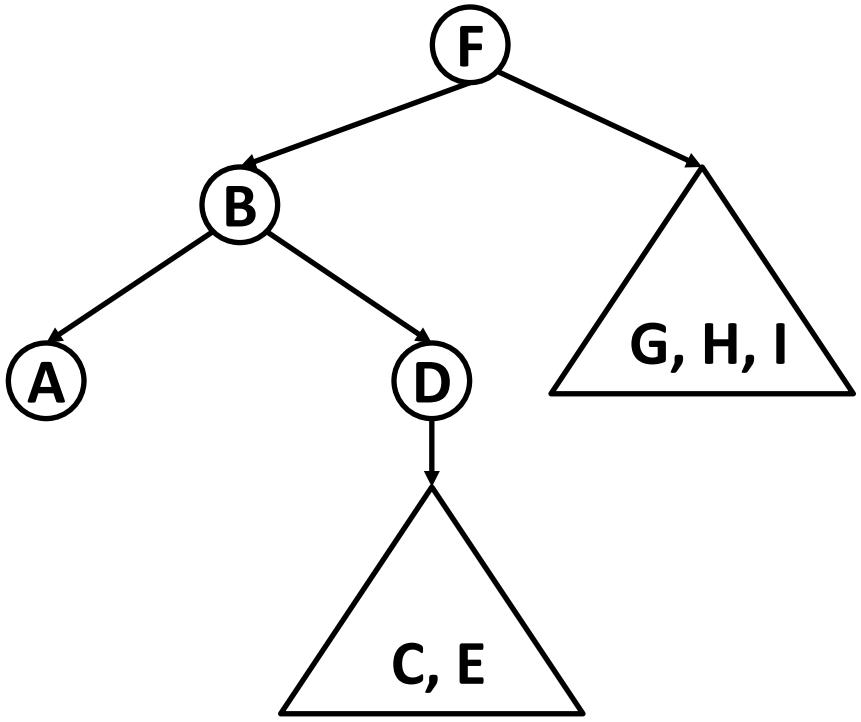
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Preorder and Postorder

Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]

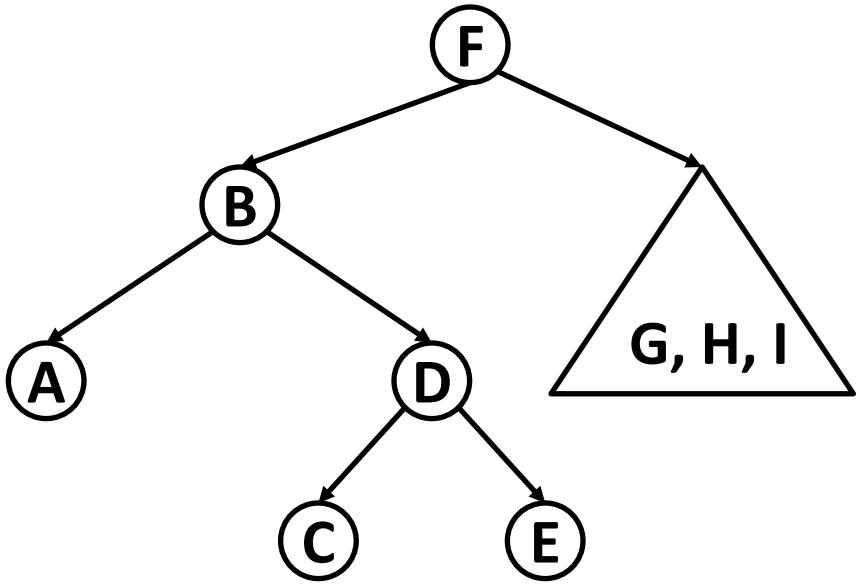
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Preorder and Postorder

Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]

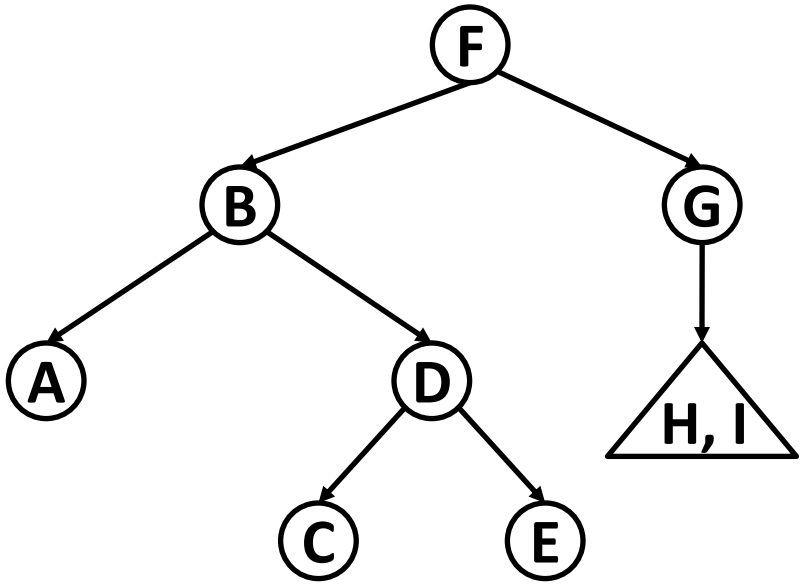
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



# Preorder and Postorder

Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]

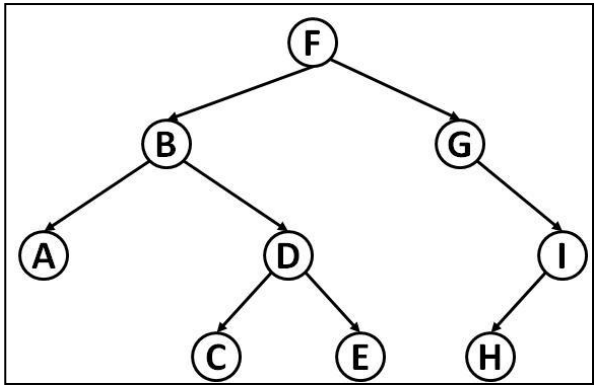
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]

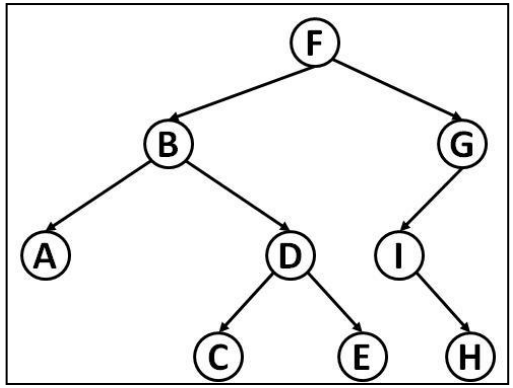


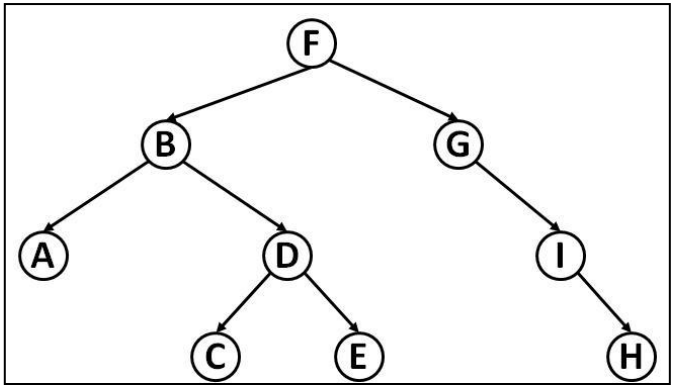
# Preorder and Postorder

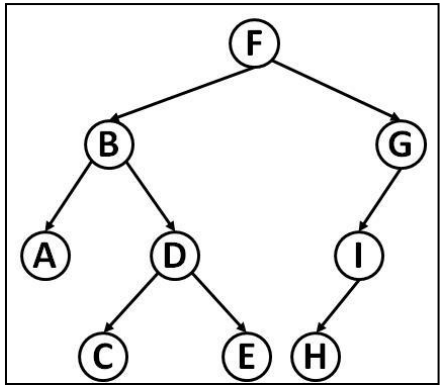
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]

Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



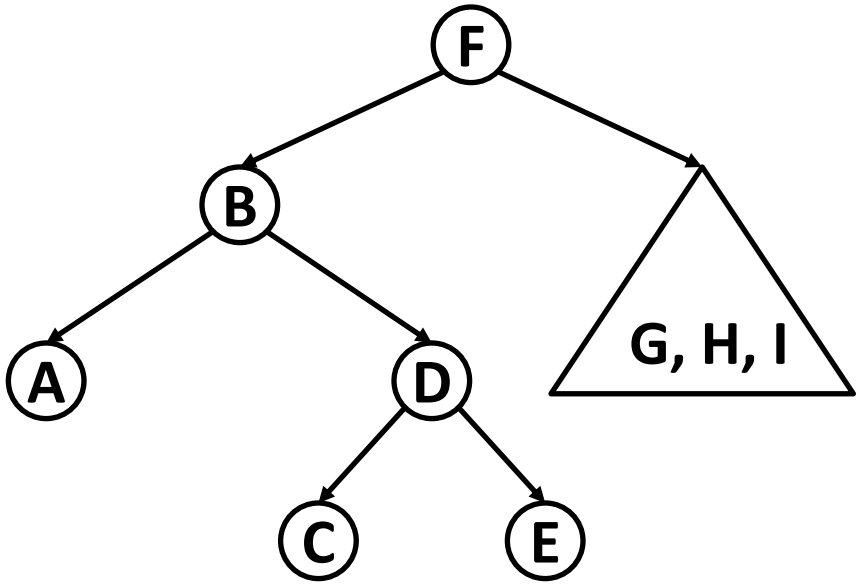



# Preorder and Postorder (Full Tree)

Preorder: F, B, A, D, C, E, H, G, I [LEFT PARENT RIGHT]

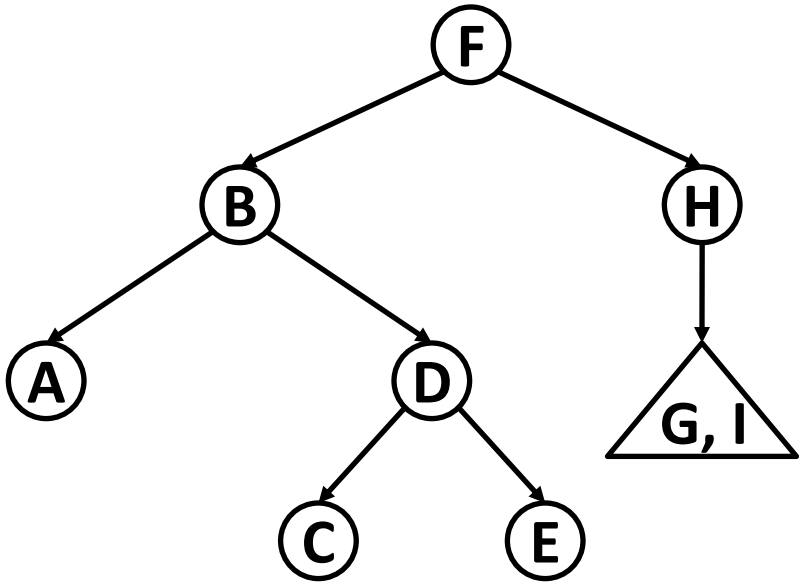
Postorder: A, C, E, D, B, G, I, H, F [LEFT RIGHT PARENT]



# Preorder and Postorder (Full Tree)

Preorder: F, B, A, D, C, E, H, G, I [LEFT PARENT RIGHT]

Postorder: A, C, E, D, B, G, I, H, F [LEFT RIGHT PARENT]



# Preorder and Postorder (Full Tree)

Preorder: F, B, A, D, C, E, H, G, I [LEFT PARENT RIGHT]

Postorder: A, C, E, D, B, G, I, H, F [LEFT RIGHT PARENT]