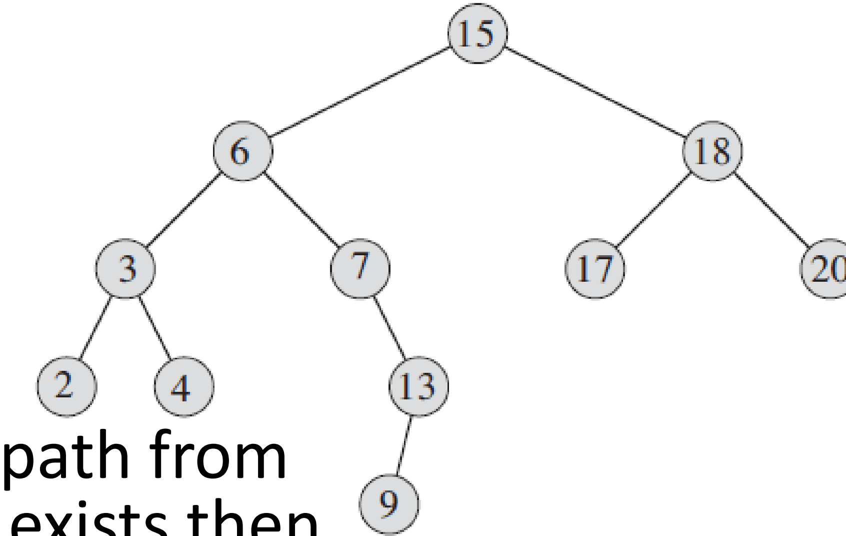
Binary Search Tree

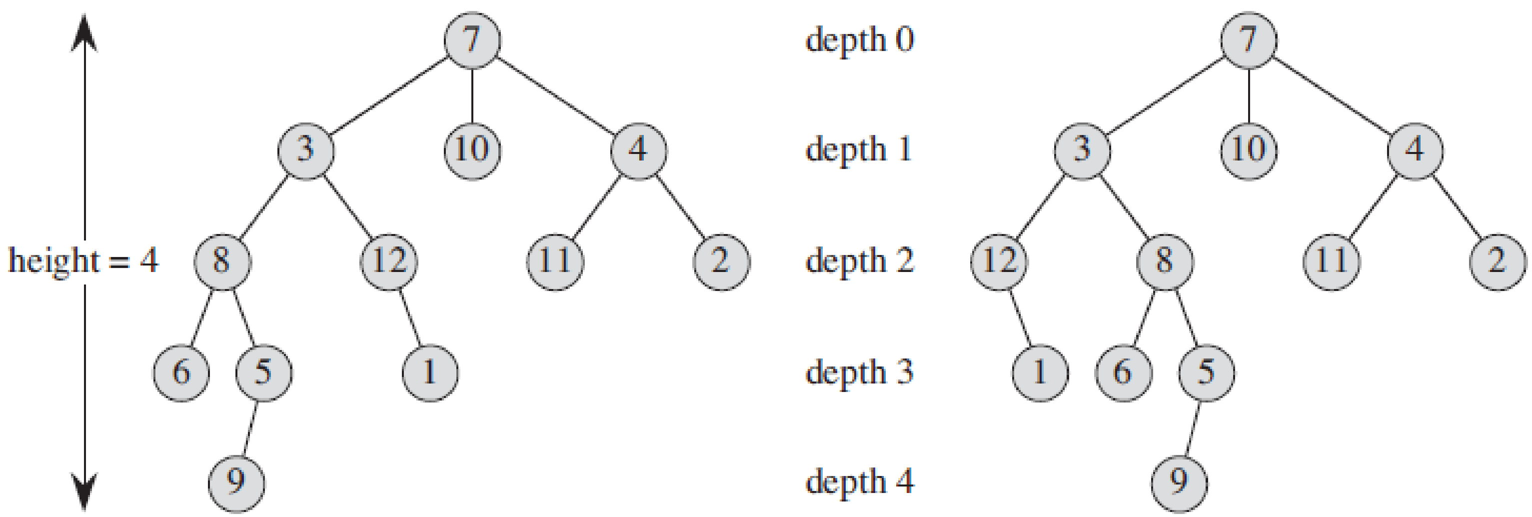
Terminology



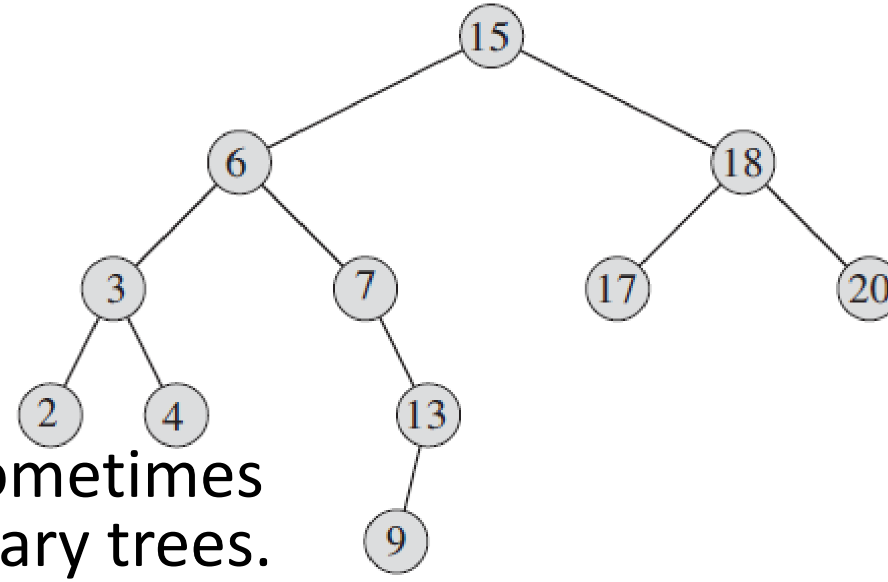
- **Root**: Topmost node in a tree
- **Child** and **Parent**: On a simple path from root to a node, if an edge (x,y) exists then x is the parent of y, and y is a child of x.
- **Siblings**: Nodes with the same parent
- **Descendant**: Node reachable by repeated proceeding from parent to child
- **Ancestor**: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

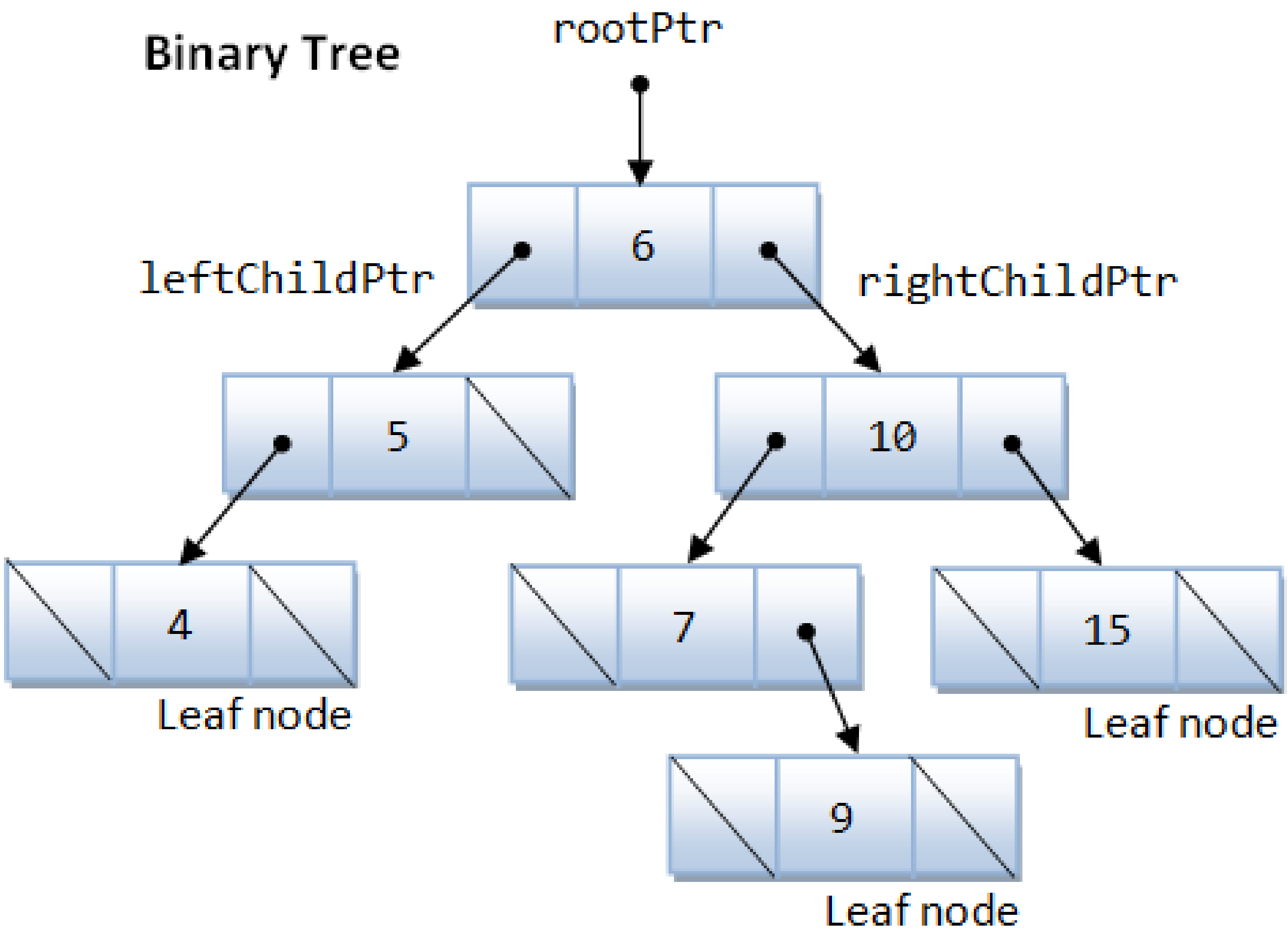


- Binary search trees (BST), sometimes called ordered or sorted binary trees.
- Properties:
  - The left subtree of a node contains only nodes with keys less than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - The left and right subtree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

Structure code of a tree node

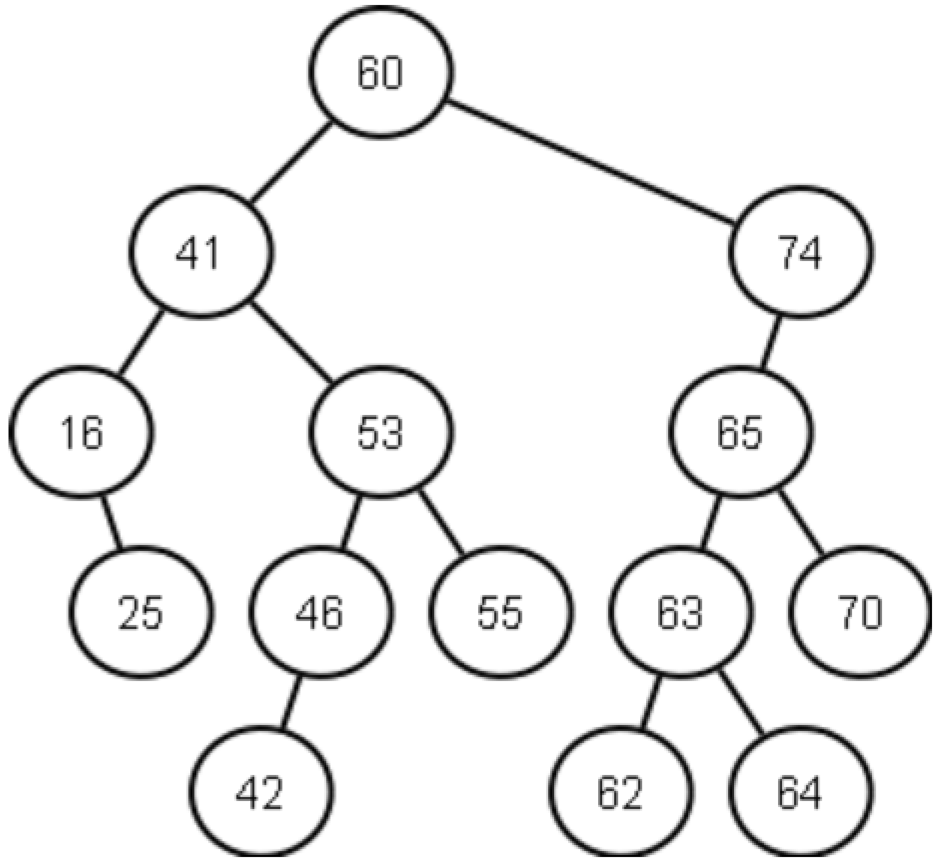
```
struct node
{ int data; //Data element
  struct node * left; //Pointer to left node
  struct node * right; //Pointer to right node
};

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

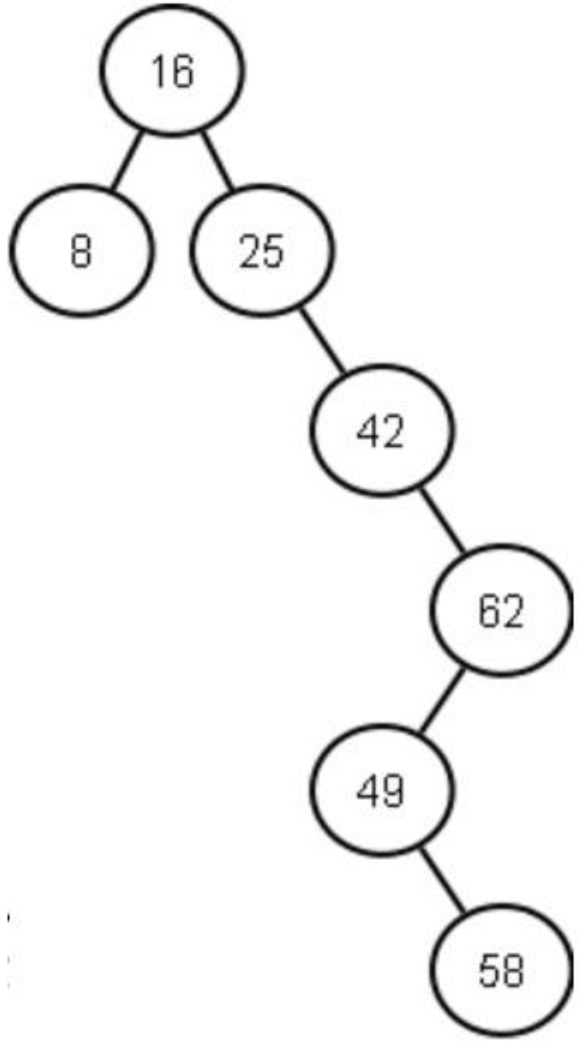
- Preorder traversal:
(parent, left, right)
- Inorder traversal:
(left, parent, right)
- Postorder traversal:
(left, right, parent)



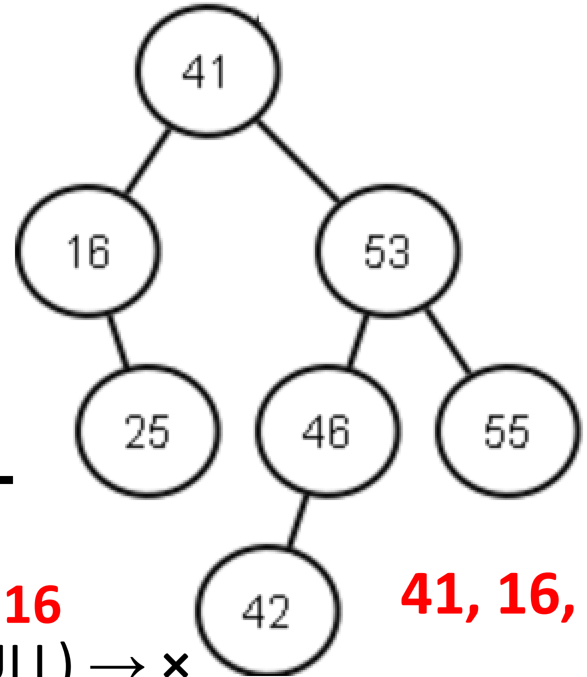
- Preorder traversal
60, 41, 16, 25, 53, 46, 42, 55, 74, 65, 63, 62, 64, 70
- Inorder traversal
16, 25, 41, 42, 46, 53, 55, 60, 62, 63, 64, 65, 70, 74
- Postorder traversal
25, 16, 42, 46, 55, 53, 41, 62, 64, 63, 70, 65, 74, 60

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



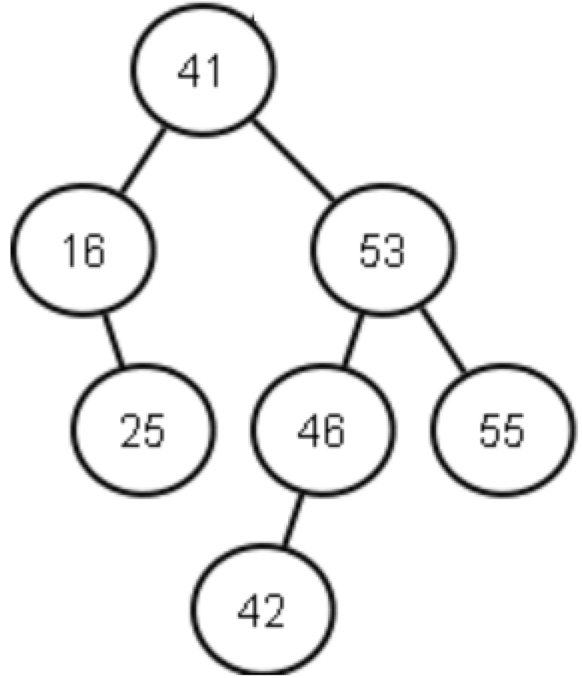
```
void preorder(node *temp)
{  if (temp != NULL)
   {  printf("%d", temp->data);
      preorder(temp->lchild);
      preorder(temp->rchild);
   } }
```



- 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) ⟶ ×

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

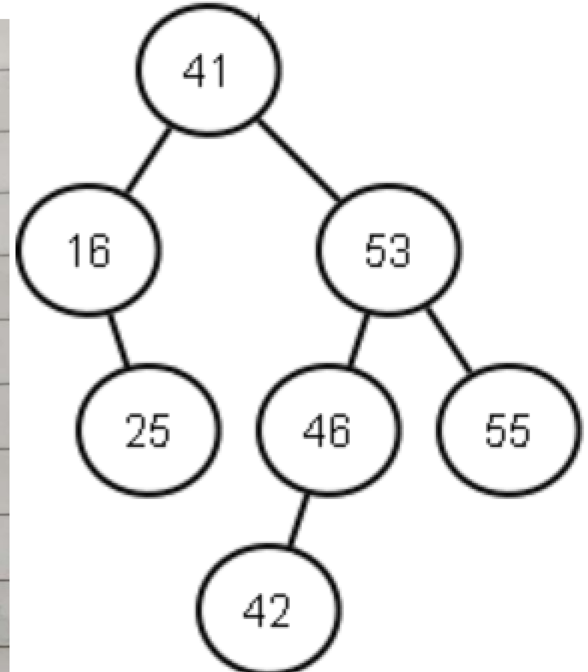
inorder (41)
- inorder (16)
  - inorder (NULL) → x
  - 16
  - inorder (25)
    - inorder (NULL) → x
    - 25
    - inorder (NULL) → x
- 41
- inorder (53)
  - inorder (46)
    - inorder (42)
      - inorder (NULL) → x
      - 42
      - inorder (NULL) → x
    - 46
    - inorder (NULL) → x
  - 53
  - inorder (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);
   } }
```

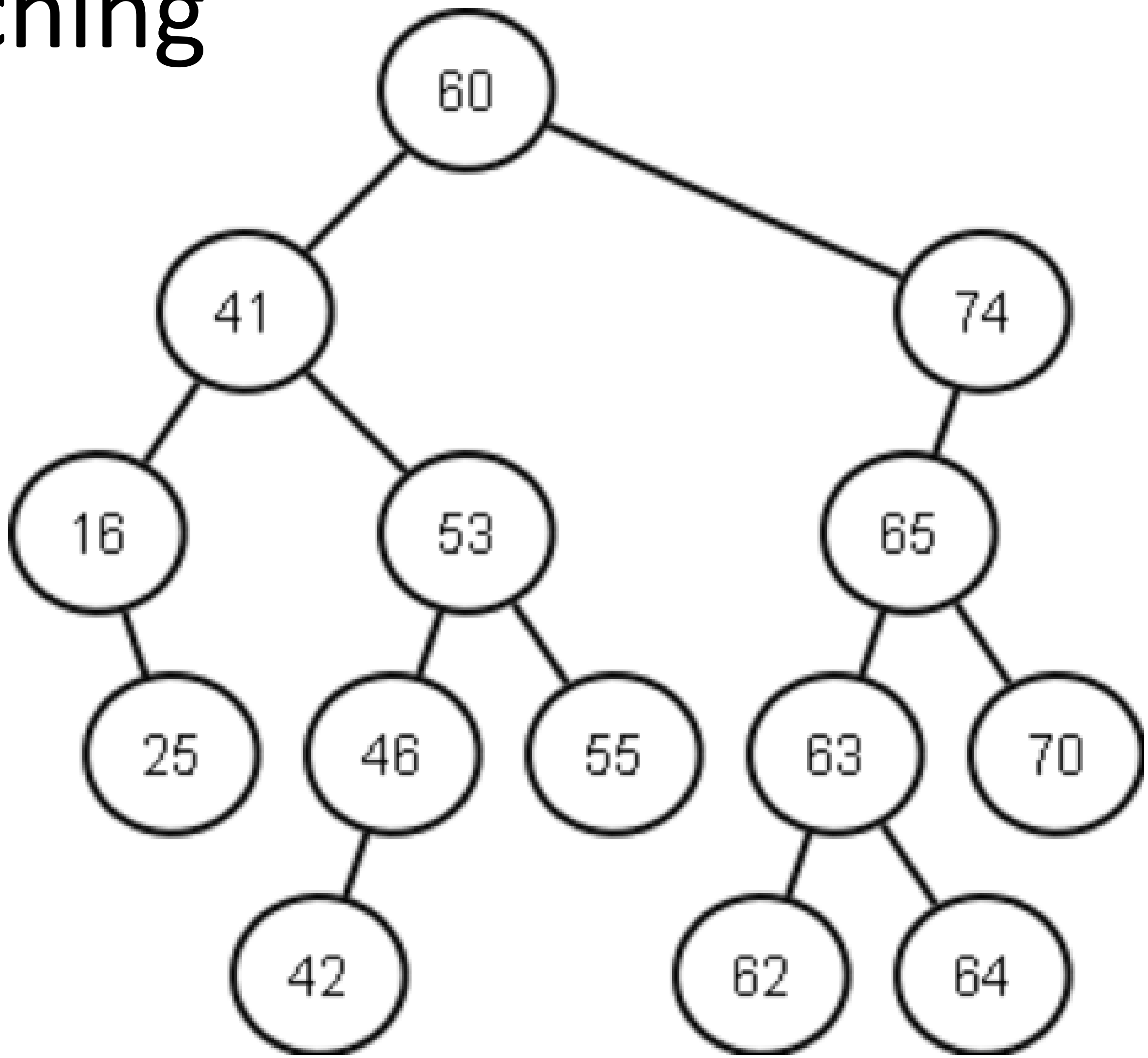
postorder (41)
- postorder(16)
  - postorder (NULL) → x
  - postorder (25)
    - postorder (NULL) → x
    - postorder (NULL) → x
    - 25
  - 16
- postorder(53)
  - postorder(46)
    - postorder(42)
      - postorder (NULL) → x
      - postorder (NULL) → x
      - 42
    - postorder (NULL) → x
    - 46
  - postorder(55)
    - postorder (NULL) → x
    - postorder (NULL) → x
    - 55
  - 53
- 41



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



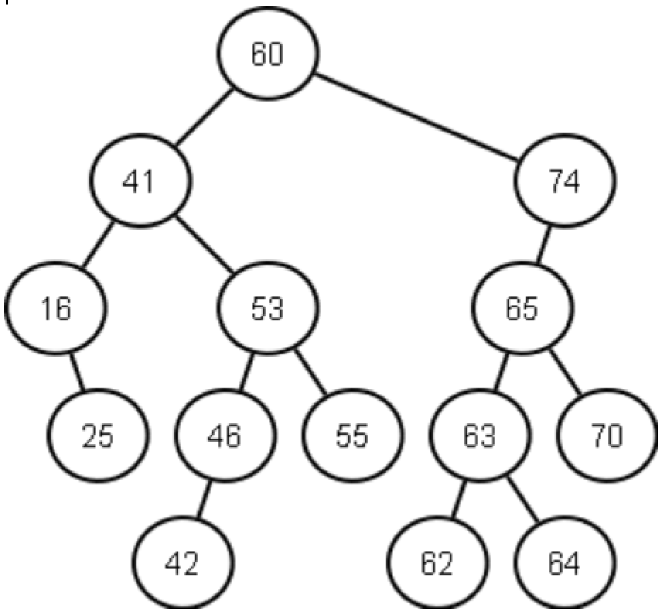
Contd…

```
TREE-SEARCH(x, k)
if x == 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 ≠ NIL and k ≠ x.key
        if k < x.key
                x = x.left
        else x = x.right
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)
```

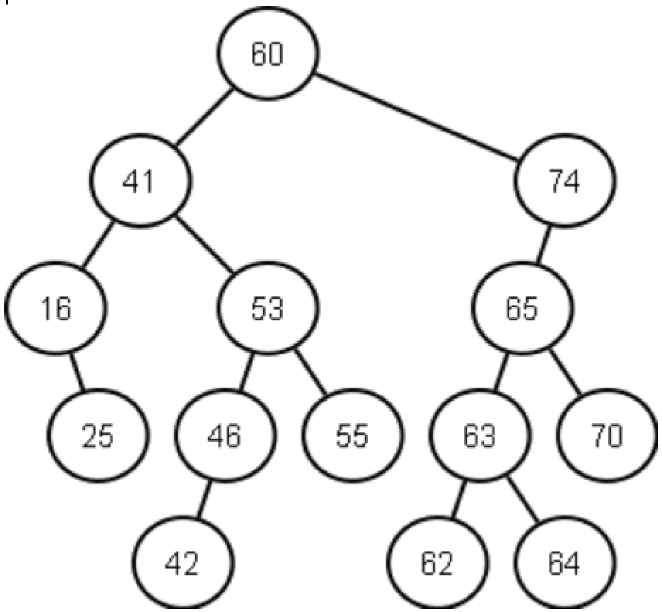


Tree-Search (root, 100)

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

TREE-SEARCH(x, k)

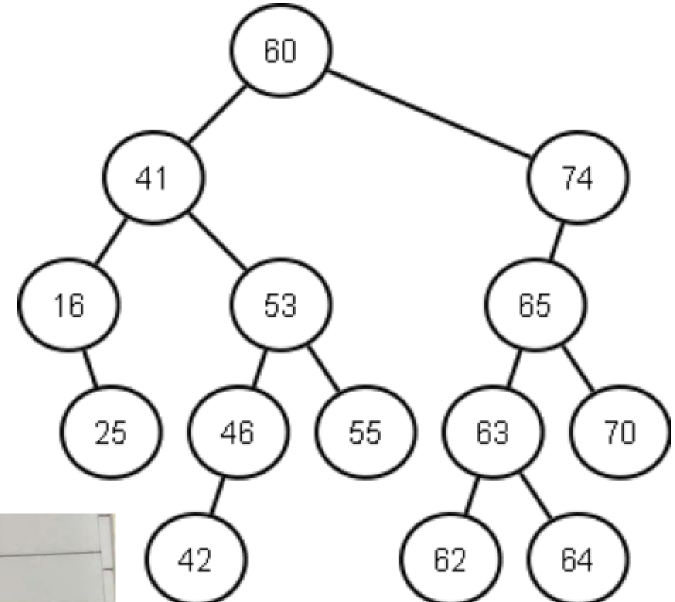
```
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```



Tree-Search (root, 53)

x	K	Execution
Node 60	53	① $x \neq$ NULL or $53 \neq 60$
		③ $53 < 60$ ✓
		④ return Tree-Search (Node 41, 100)
Node 41	53	① $x \neq$ NULL or $53 \neq 41$
		③ $53 < 41$ x
		⑤ return Tree-Search (Node 53, 53)
Node 53	53	① $x \neq$ NULL or $53 == 53$ ✓
		return (Node 53)

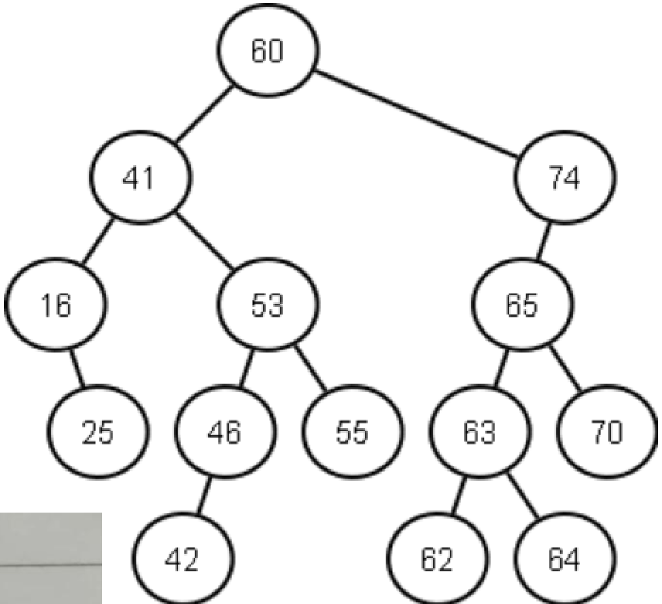
```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```



Iterative - Tree - Search (root, 100)

x	k	Execution
Node 60	100	x ≠ NULL ✓ 100 ≠ 60 ✓
		100 < 60 ✗
		x = x.right = Node 74
Node 74	100	x ≠ NULL ✓ 100 ≠ 74 ✓
		100 < 74 ✗
		x = x.right = Node (NULL)
NULL	100	x ≠ NULL ✗
		return (NULL)

```
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
```



Iterative - Tree - Search (root, 53)

x	K	Execution
Node 60	53	$x \neq$ NULL ✓ $60 \neq 53$ ✓
		$53 < 60$ ✓
		$x = x.left$ = Node 41
Node 41	53	$x \neq$ NULL ✓ $41 \neq 53$ ✓
		$53 < 41$ ✗
		$x = x.right$ = Node 53
Node 53	53	$x \neq$ NULL ✓ $53 \neq 53$ ✗
		return (Node 53)

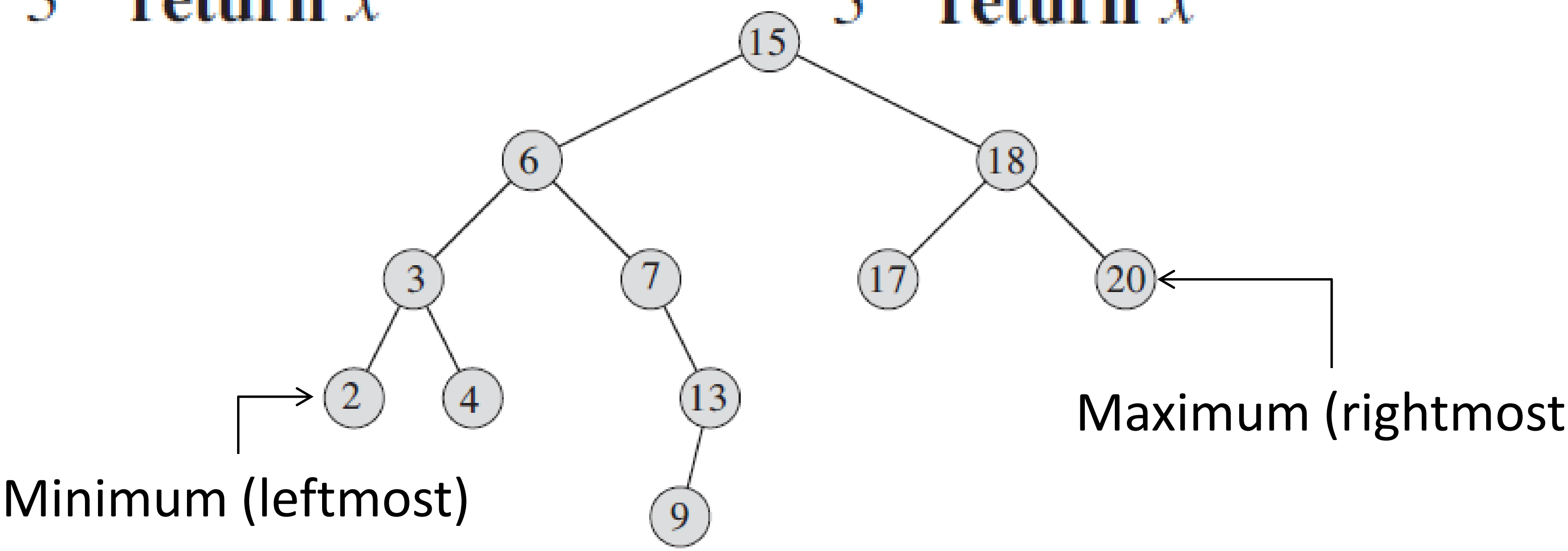
Minimum and Maximum

TREE-MINIMUM(x)

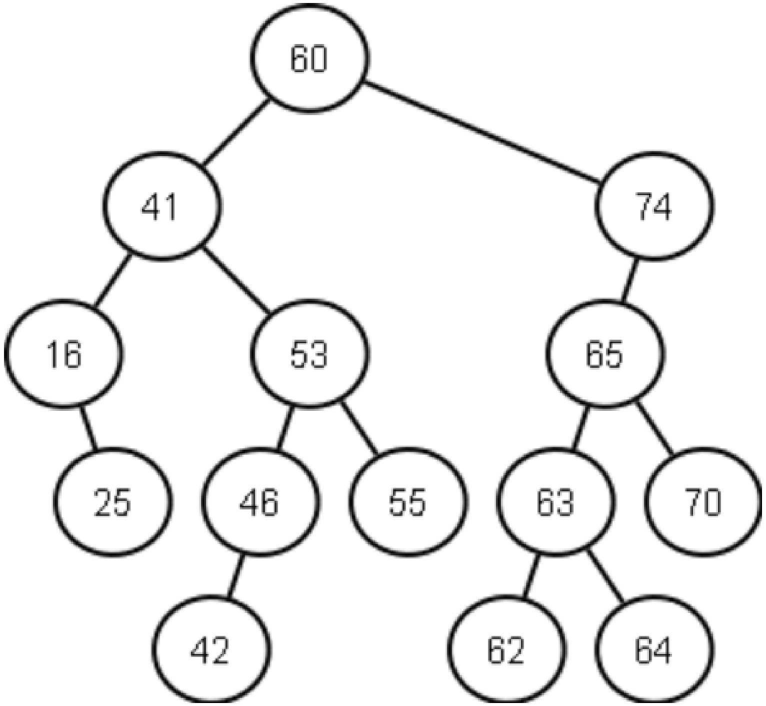
while $x.left \neq$ NIL
    $x = x.left$
return x

TREE-MAXIMUM(x)

while $x.right \neq$ NIL
    $x = x.right$
return x

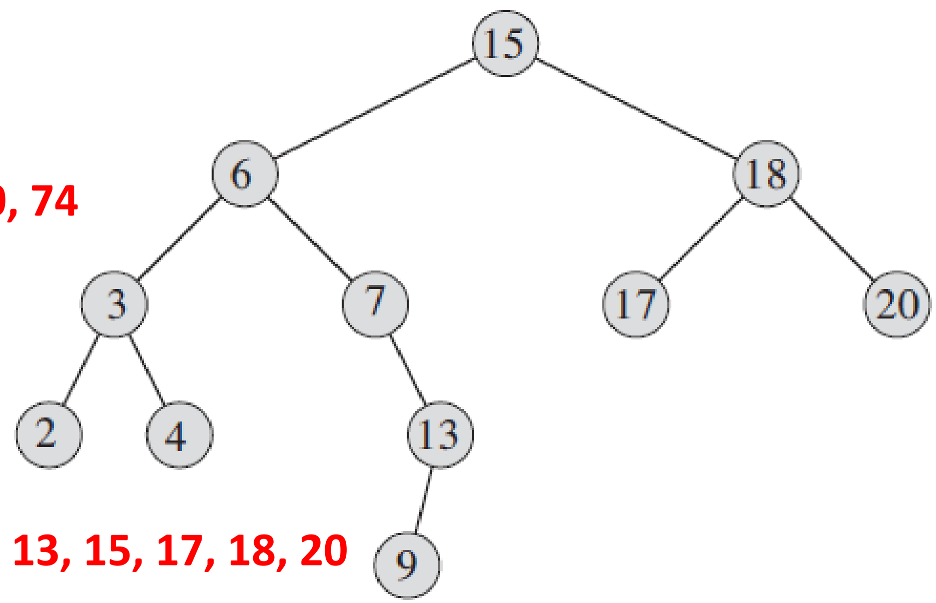


Successor and Predecessor



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

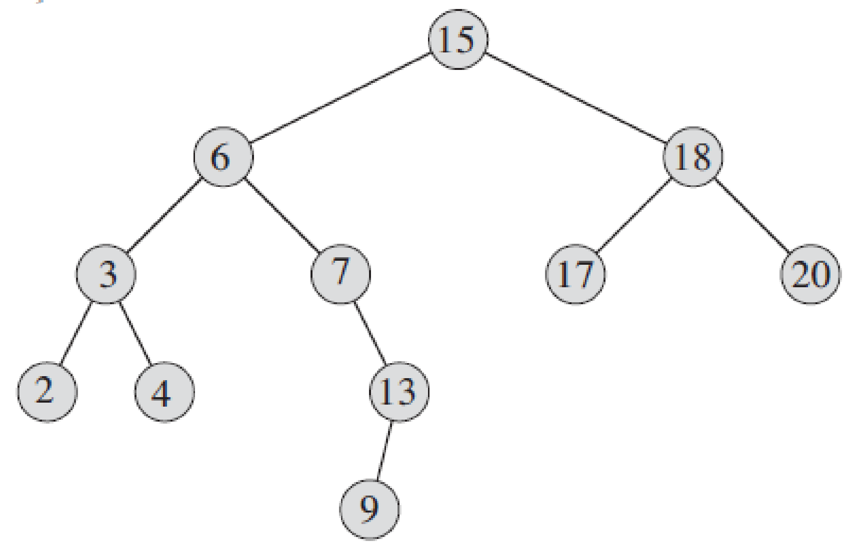
16, 25, 41, 42, 46, 53, 55, 60, 62, 63, 64, 65, 70, 74



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

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

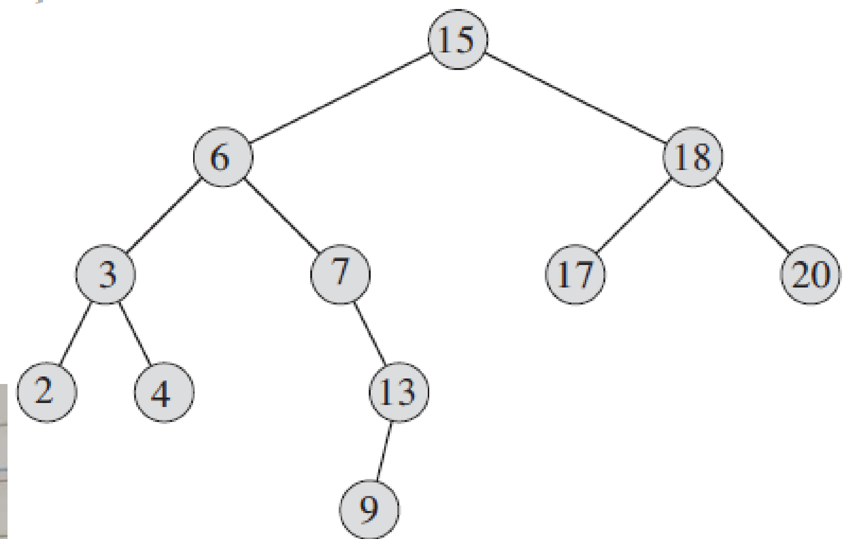


Tree-successor (6)

x	x.right	y	Execution
Node 6	Node 7		① Node 7 ≠ NULL ↳ Tree-Minimum(Node 7) ↳ 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
```

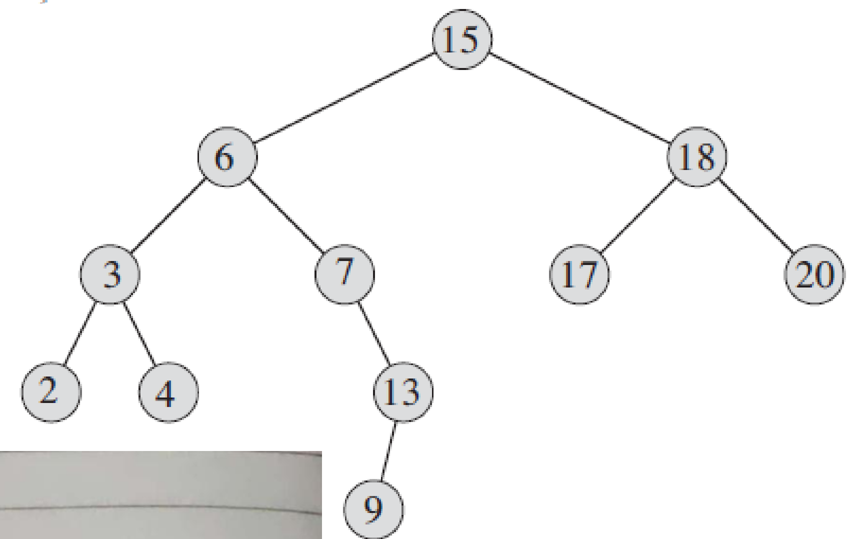


Tree-Successor (13)

x	x.right	y	Execution
Node 13	NULL	Node 7	1 NULL ≠ NULL X 4 Node 7 ≠ NULL and Node 13 == Node13
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 18X ↳ 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
```

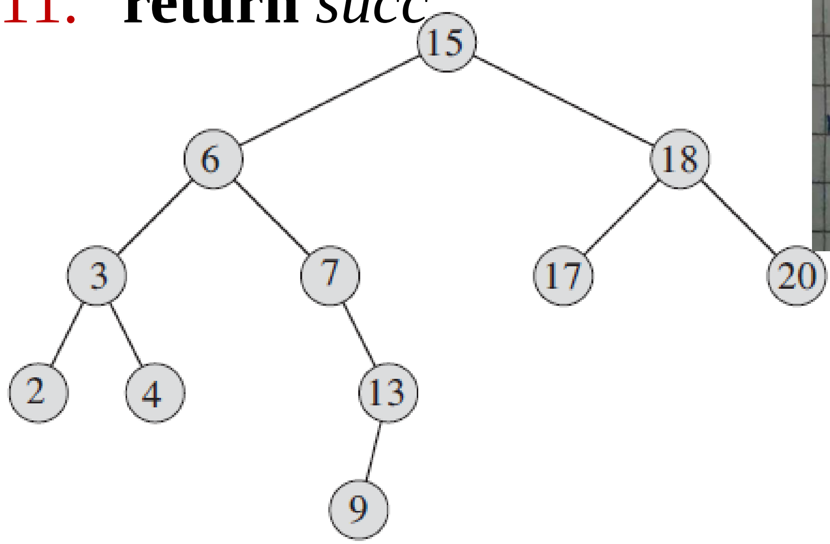


Tree - successor (20)

x	x.right	y	Execution
Node 20	NULL	Node 18	① NULL ≠ NULL ✗ ④ Node 18 ≠ NULL and node 20 == node 20
Node 18		Node 15	④ Node 15 ≠ NULL and Node 18 == Node 18
		NULL	④ NULL ≠ NULL ✗ ↳ $\boxed{\text{NULL}}$

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

```
if x.right ≠ 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
```



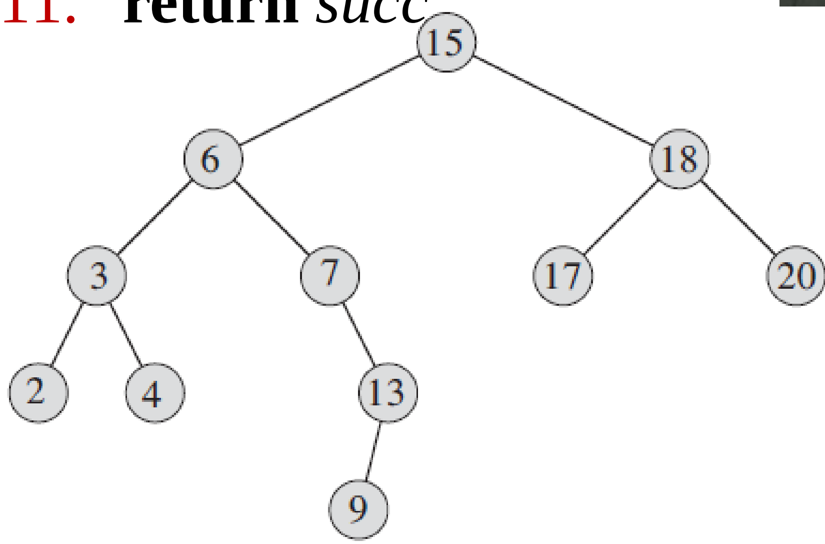
Tree-Successor-Without-Parent (root, 13)

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 × 13 > 6 ✓ y = Node 7
Node 15	Node 7	Node 13	13 < 7 × 13 > 7 ✓ y = Node 13
Node 15	Node 13	Node 13	13 < 13 × 13 > 13 × break; return 15

```
TREE-SUCCESSOR-WITHOUT-PARENT(y,x)
    if x.right ≠ 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 - Without - Parent (root, 20)

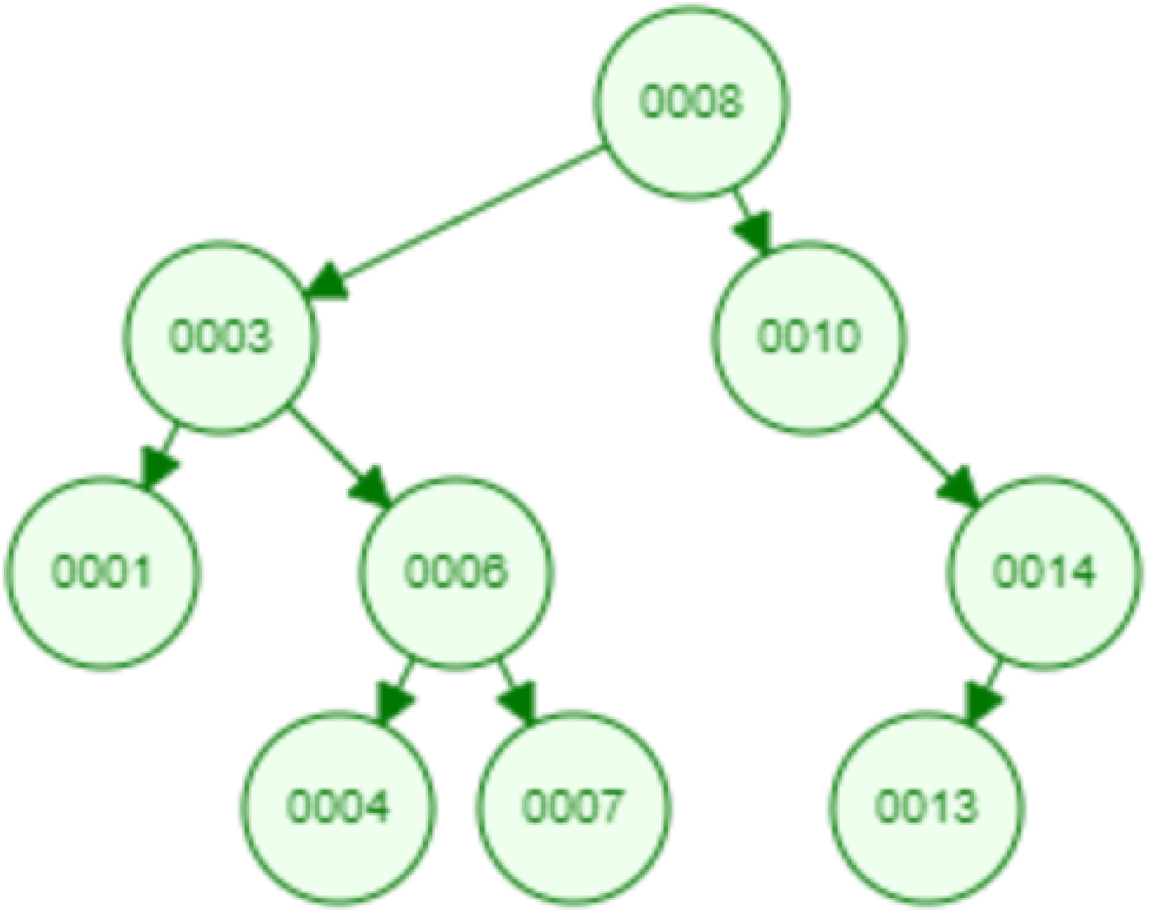
succ	y	x	Execution
NULL	Node 15	Node 20	20 < 15 x 20 > 15 ✓ y = Node 18
NULL	Node 18	Node 20	20 < 18 x 20 > 18 ✓ y = Node 20
ULL	Node 20	Node 20	20 < 20 x 20 > 20 x break; return $\boxed{\text{NULL}}$



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



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 ≠ 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
```

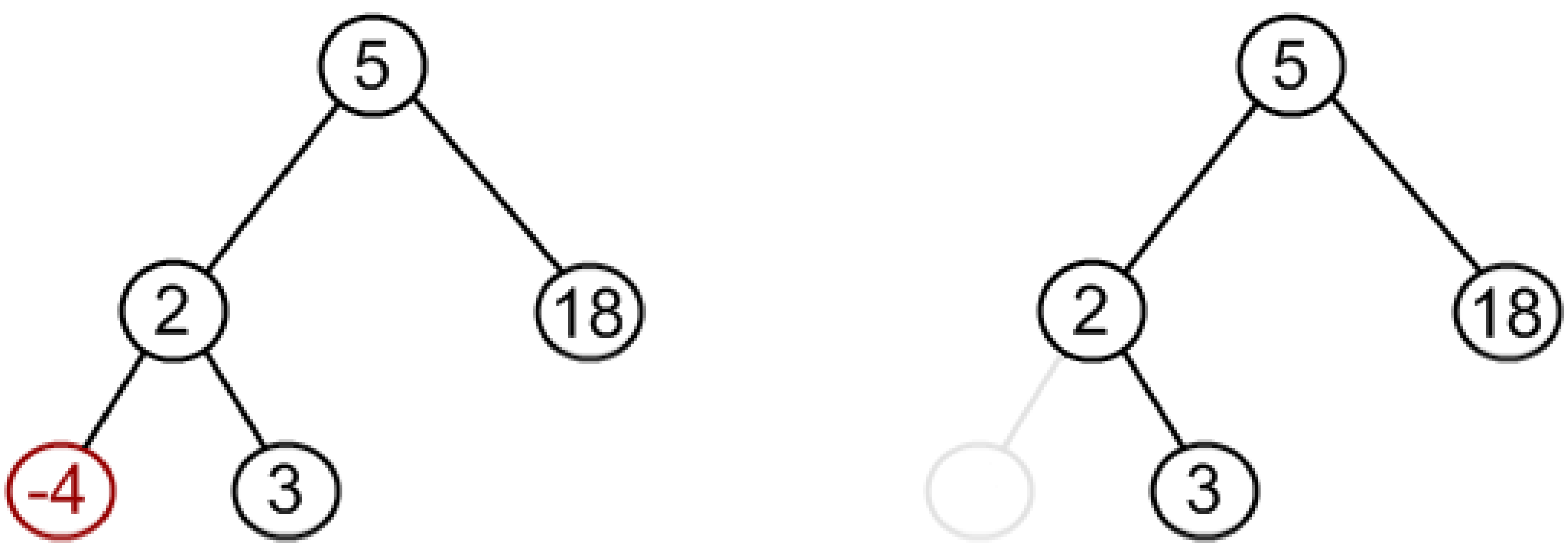
Deletion

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

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

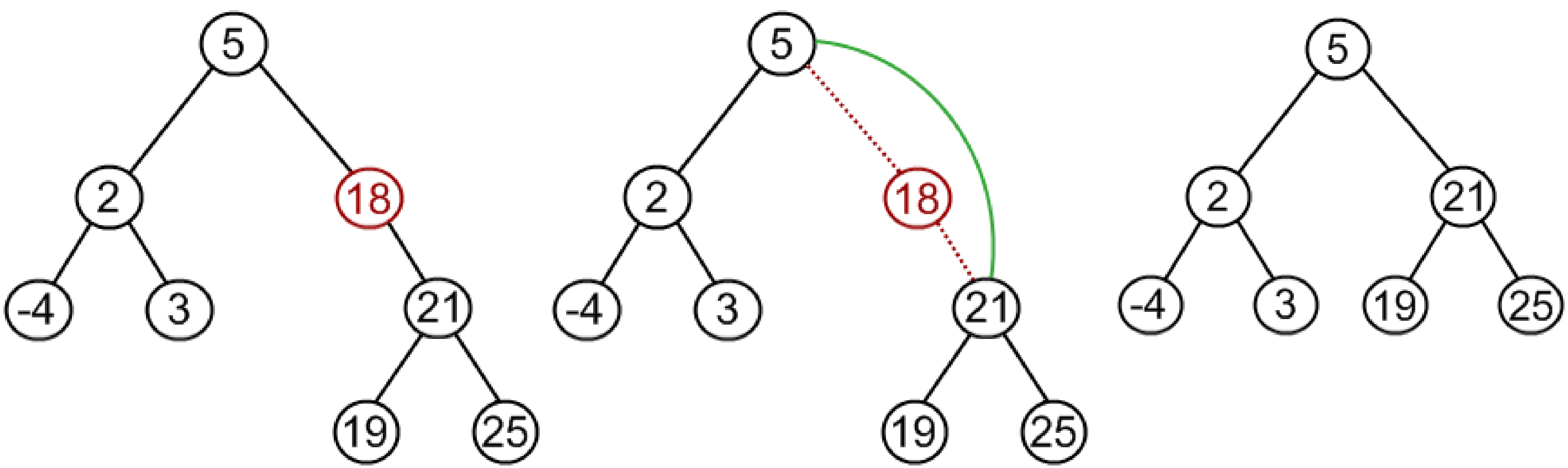
Node to be removed has no children

- 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.

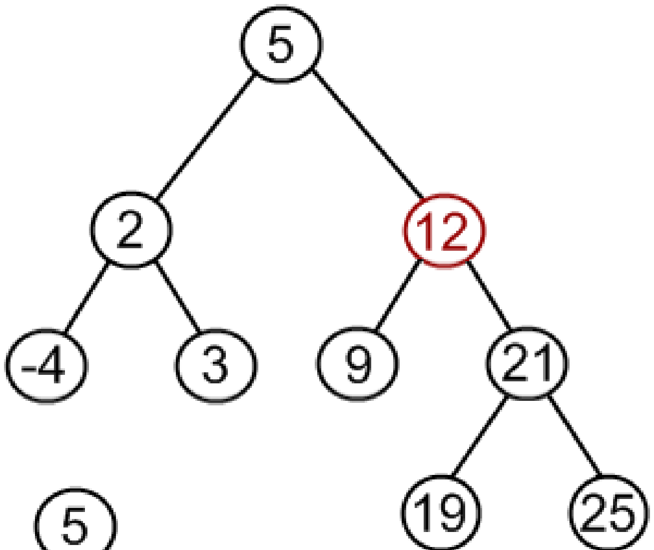


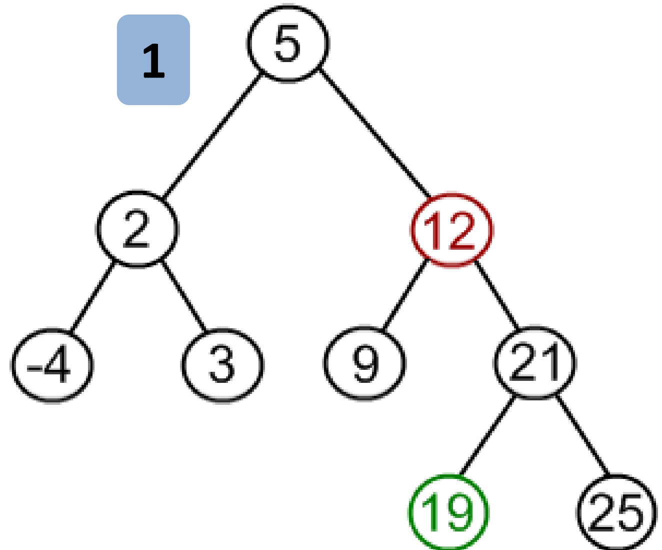
Node to be removed has two children

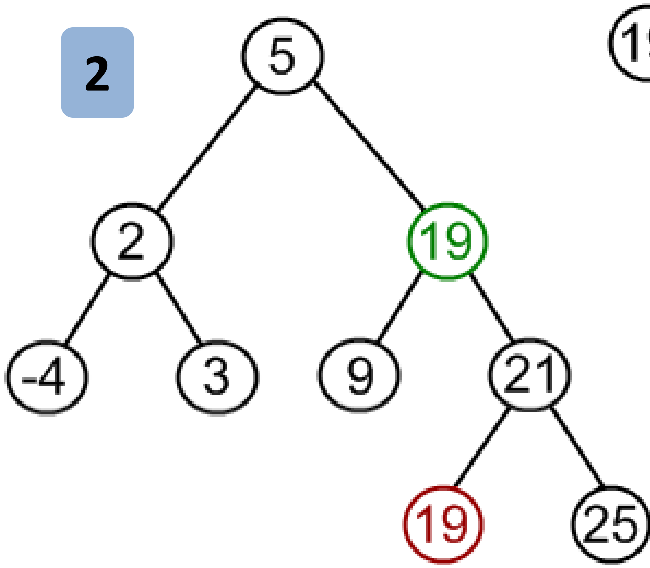
- Find inorder successor, i.e. find the minimum value in right child of the node.
- Copy contents of the inorder successor (or found minimum) to the node being removed with.
- Delete the inorder successor (or found minimum) from the right subtree.
- Note:
  - The node with minimum value has no left child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

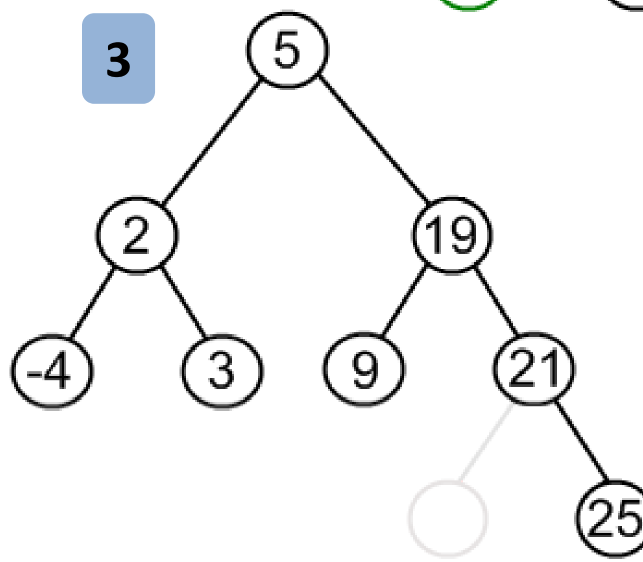
left ton maximum value kadlo;
jan
right ton minimum value kadlo;

Contd… Remove 12 from a BST.

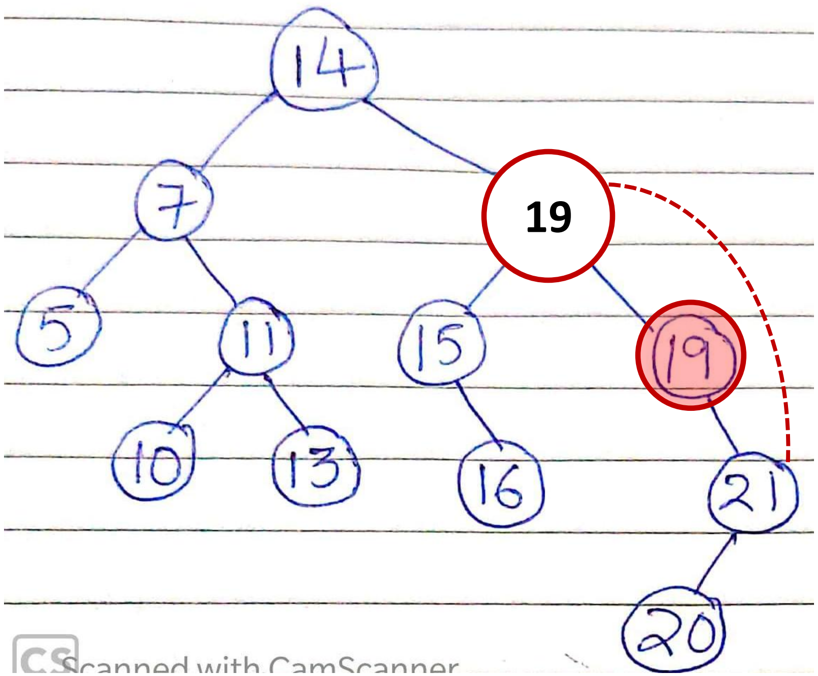


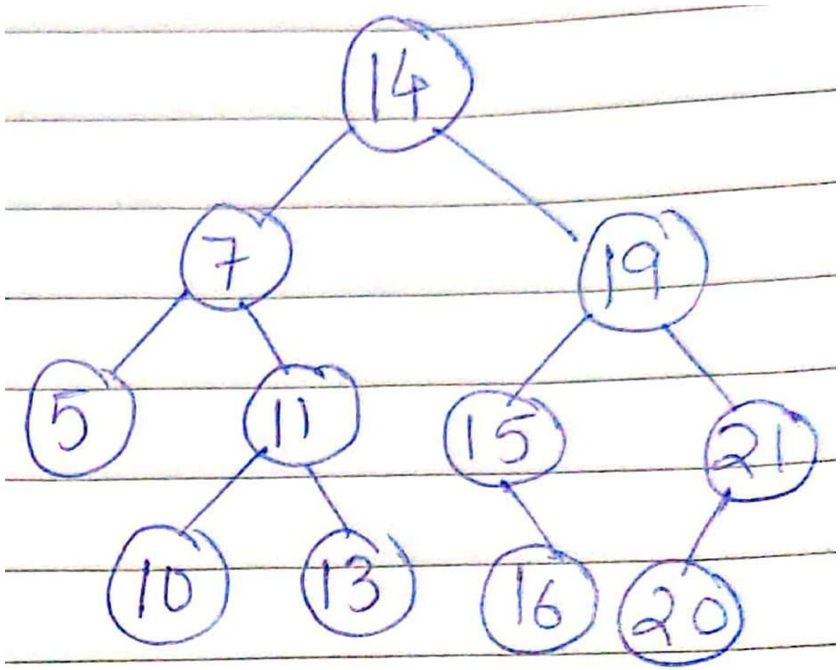






Delete 17





```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ 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
```

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

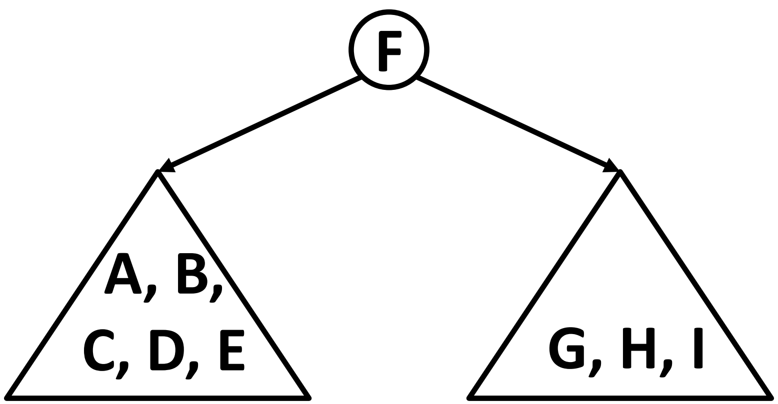
Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

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]

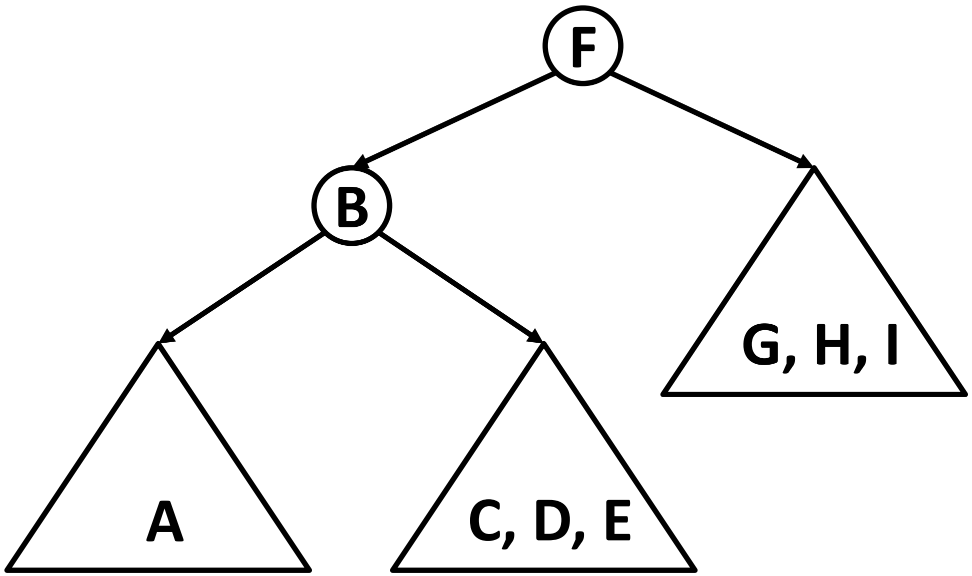
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



Inorder and Preorder

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

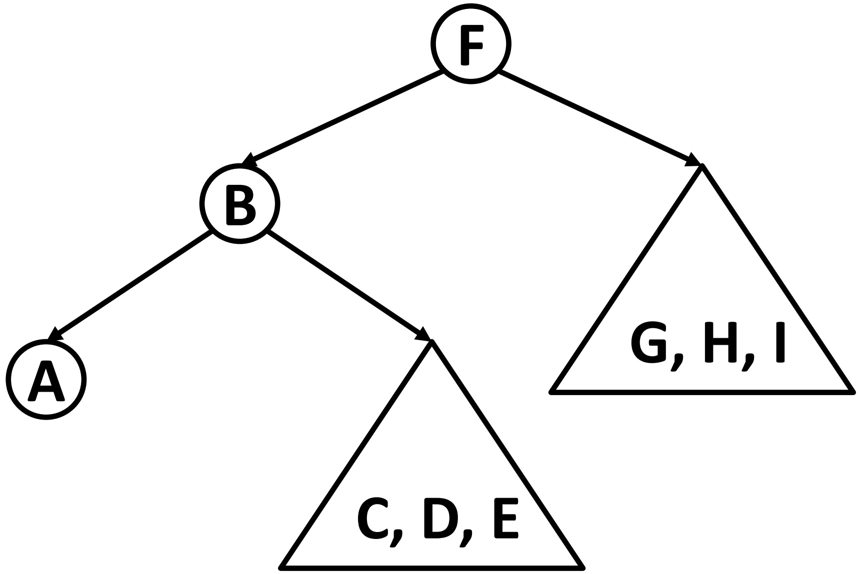
Preorder: <u>F, B</u>, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



Inorder and Preorder

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

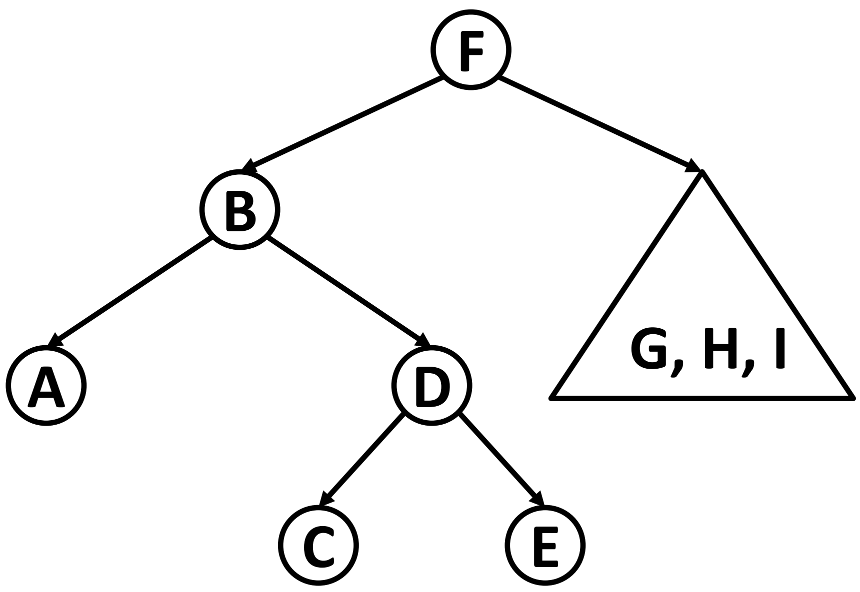
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



Inorder and Preorder

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

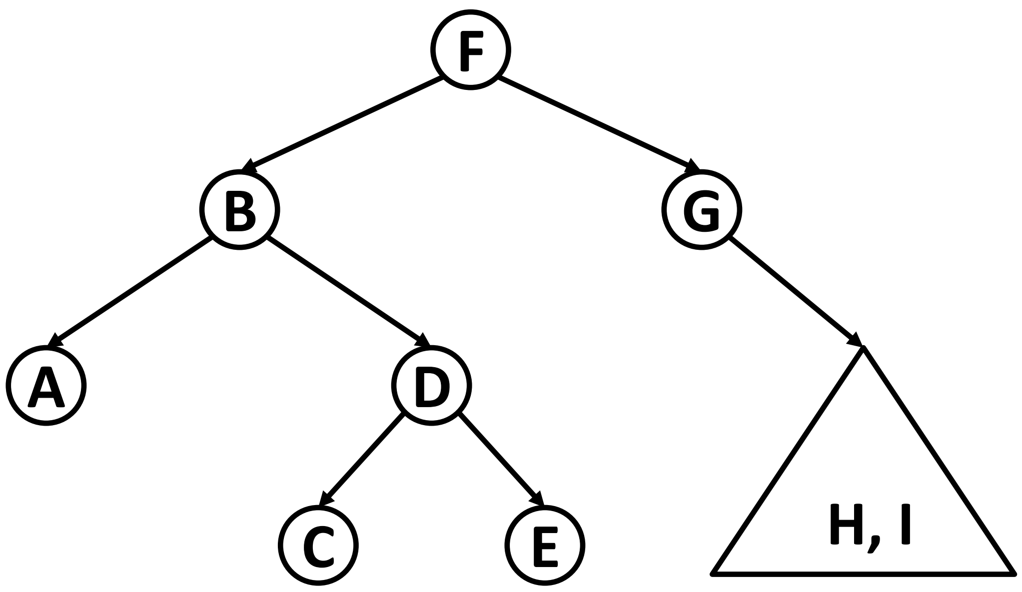
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



Inorder and Preorder

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

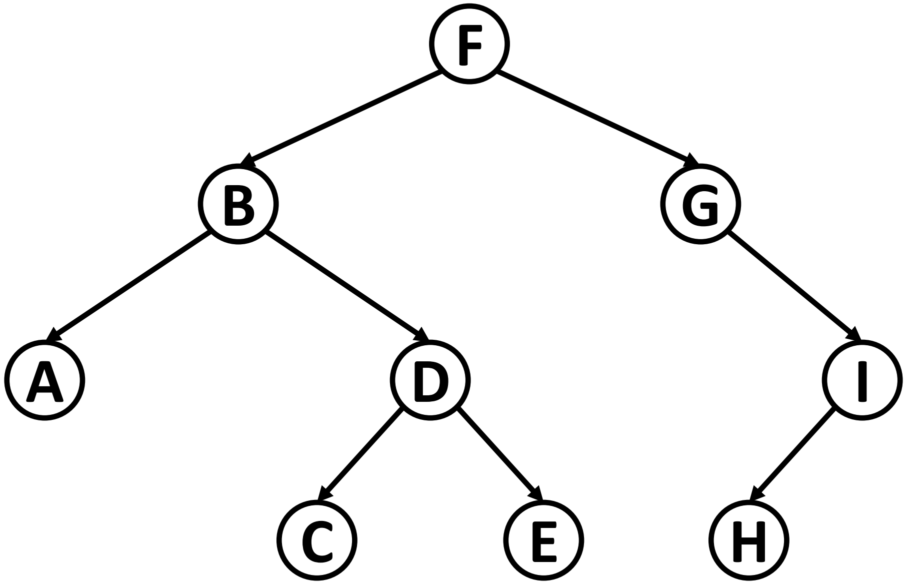
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]



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

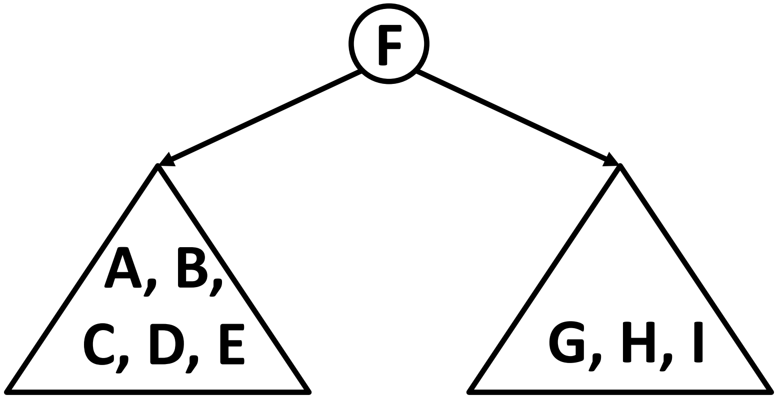
Inorder: A, B, C, D, E, F, G, H, I [LEFT PARENT RIGHT]

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]

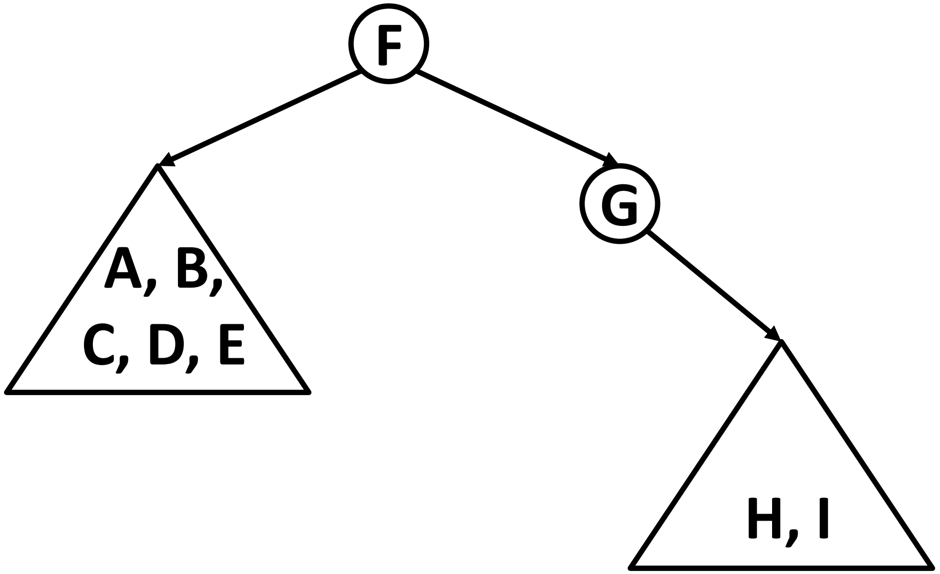
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



Inorder and Postorder

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

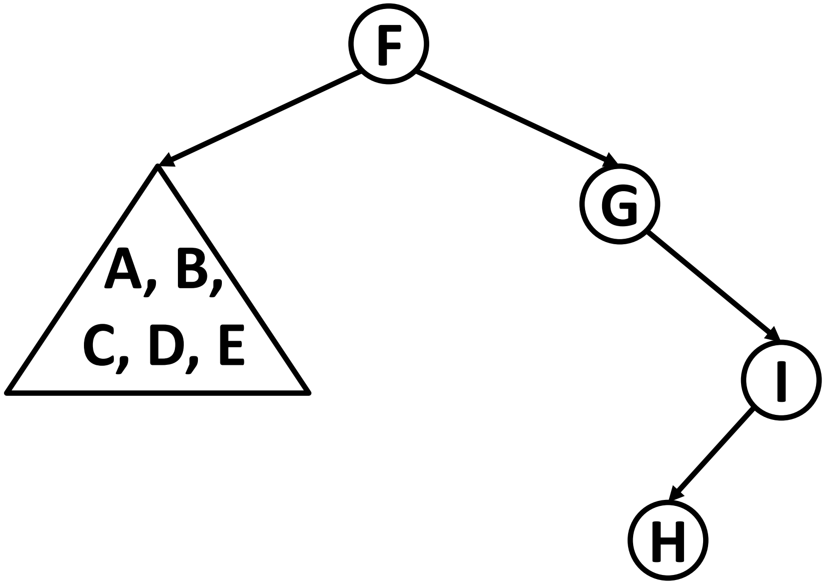
Postorder: A, C, E, D, B, H, I, <u>G, F</u> [LEFT RIGHT PARENT]



Inorder and Postorder

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

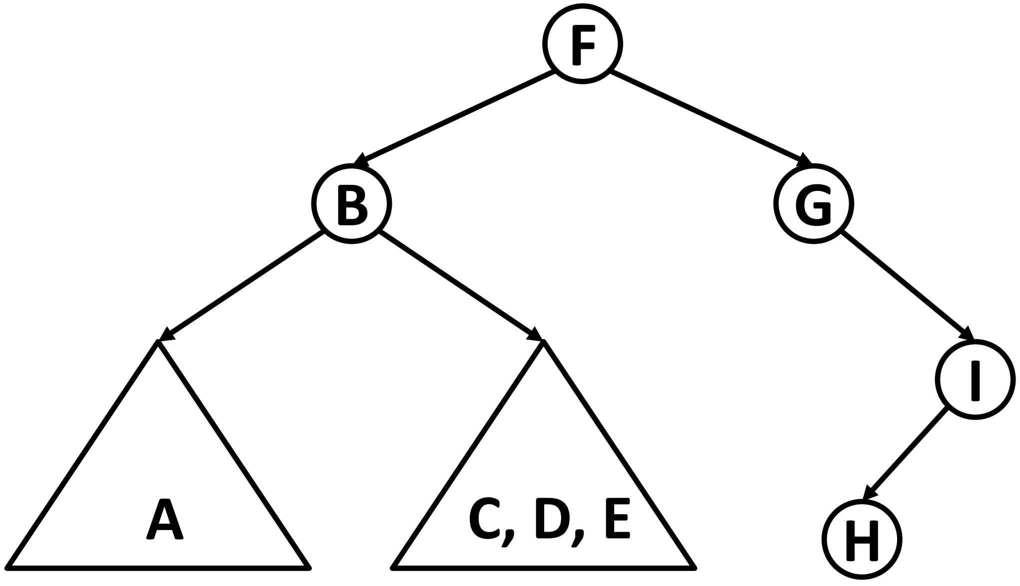
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



Inorder and Postorder

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

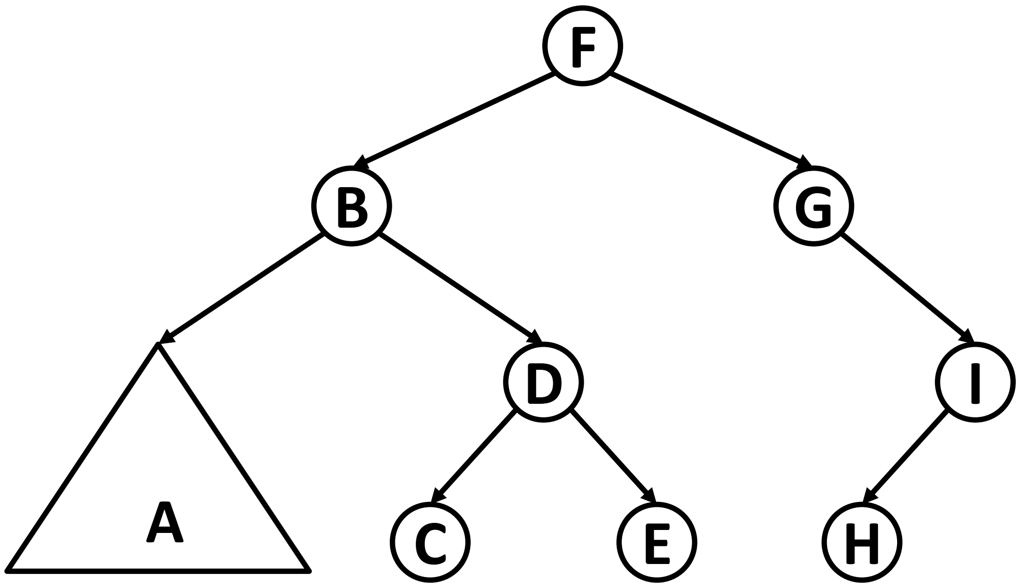
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



Inorder and Postorder

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

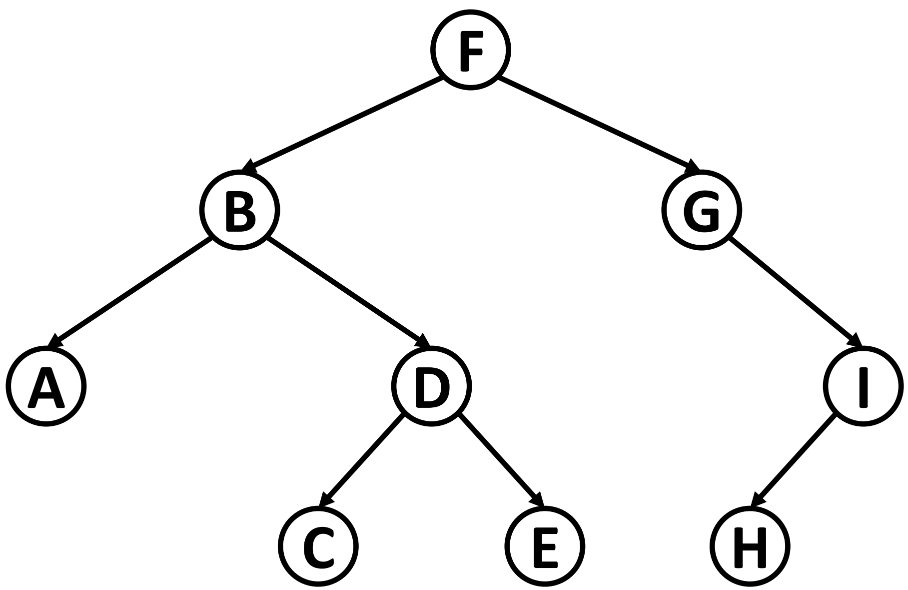
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



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

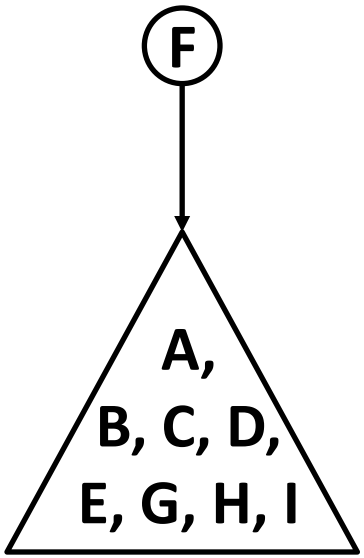
Preorder: F, B, A, D, C, E, G, I, H [PARENT LEFT RIGHT]

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]

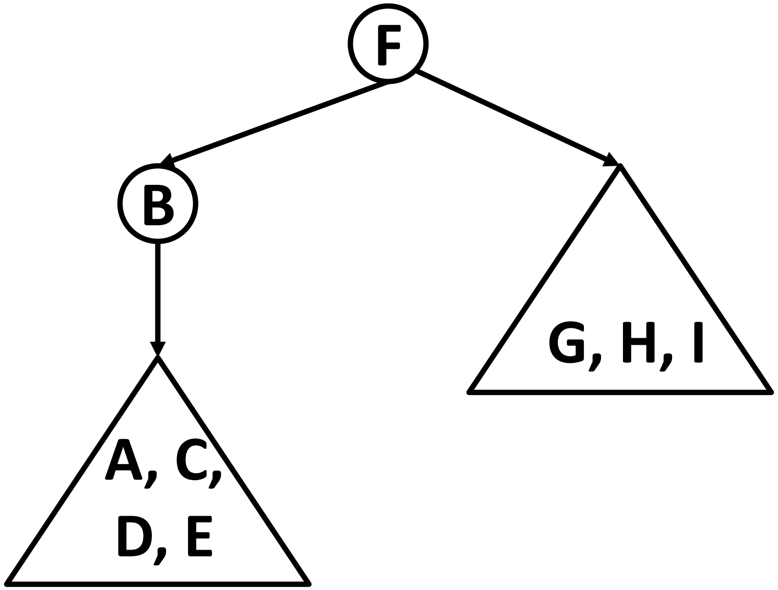
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



Preorder and Postorder

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

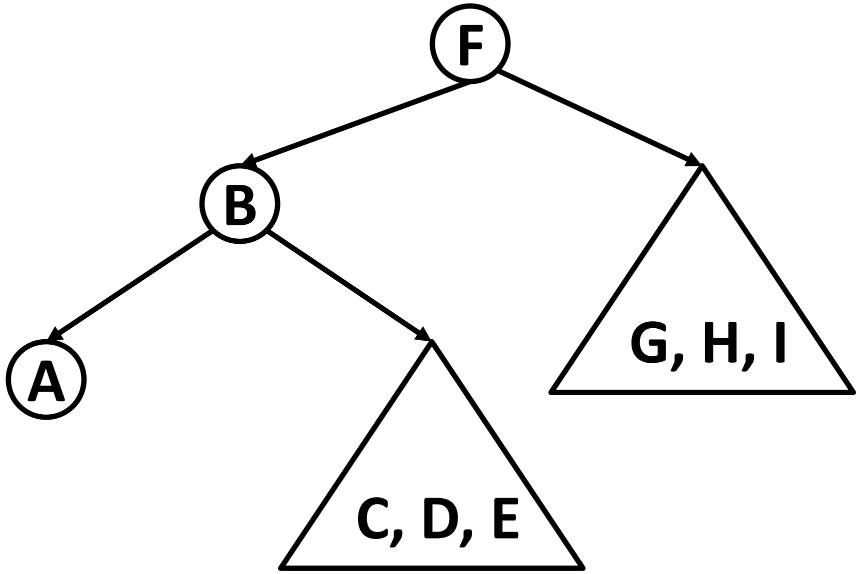
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



Preorder and Postorder

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

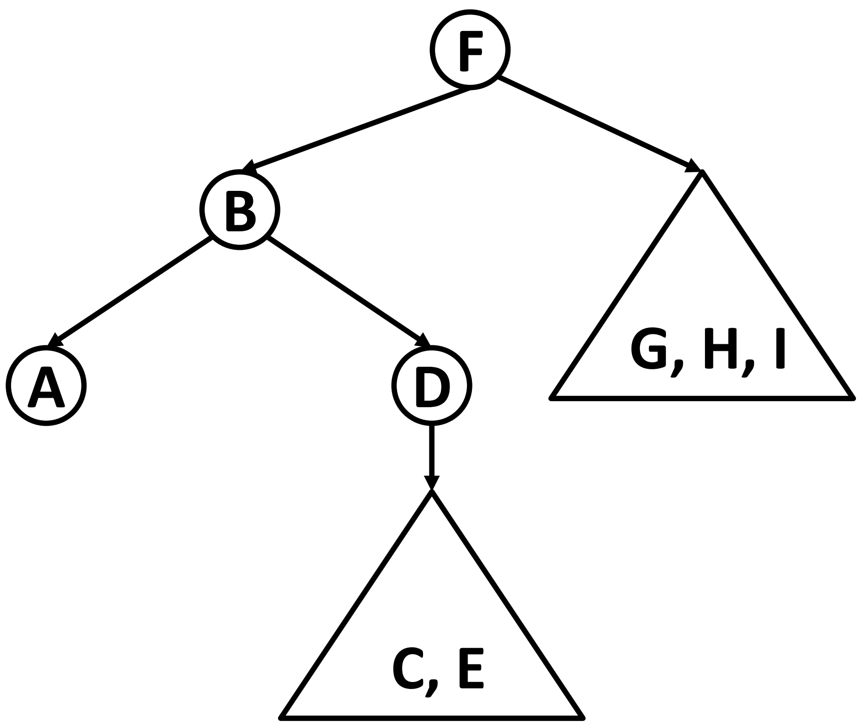
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



Preorder and Postorder

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

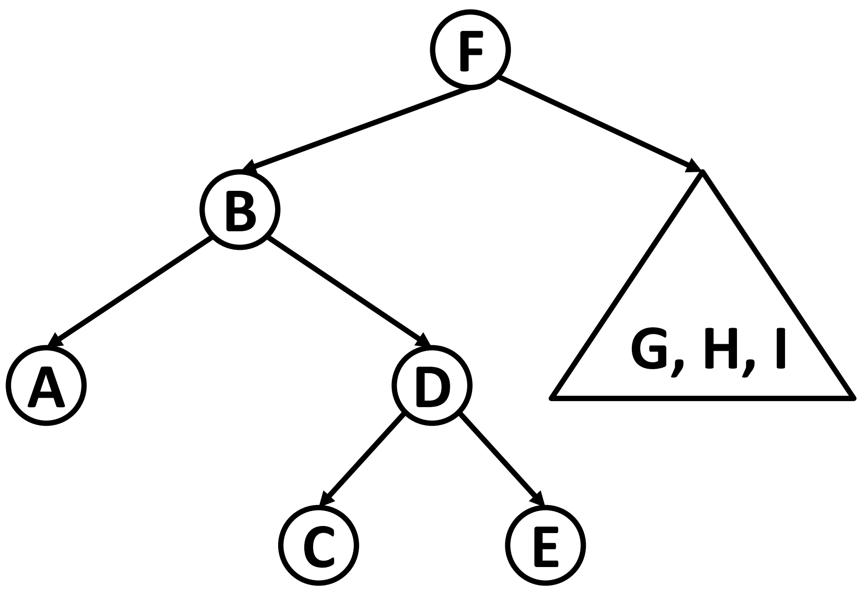
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



Preorder and Postorder

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

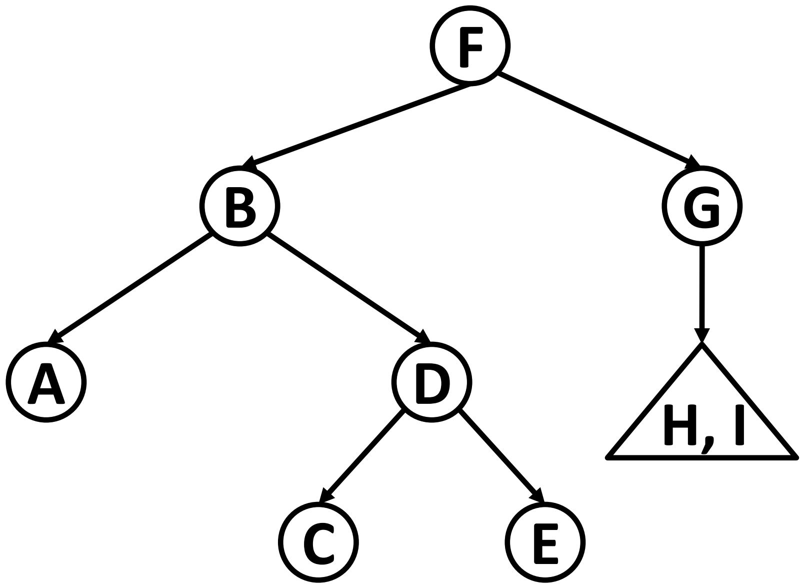
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]



Preorder and Postorder

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

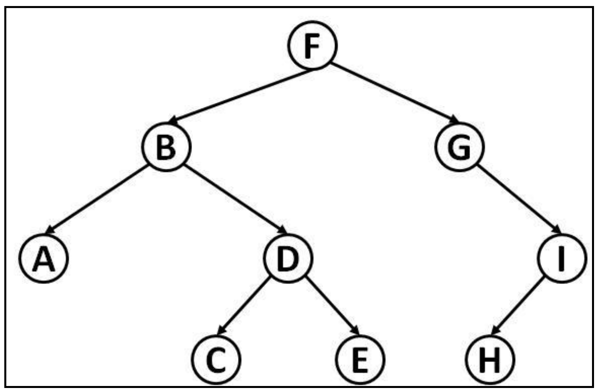
Postorder: A, C, E, D, B, H, I, G, F [LEFT RIGHT PARENT]

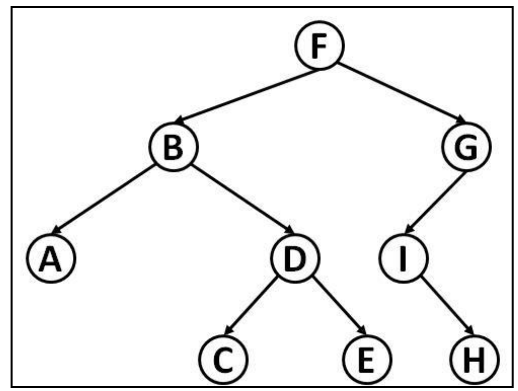


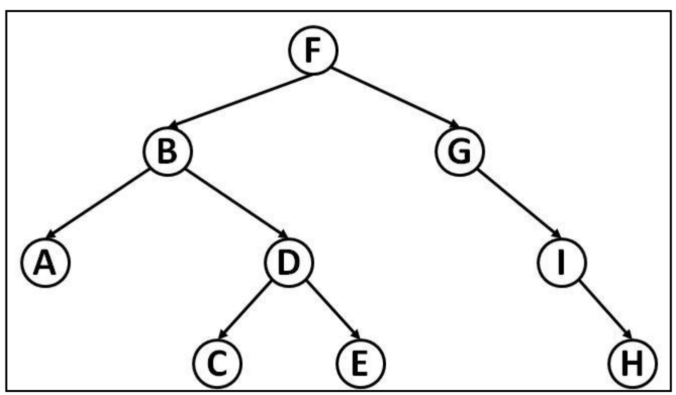
Preorder and Postorder

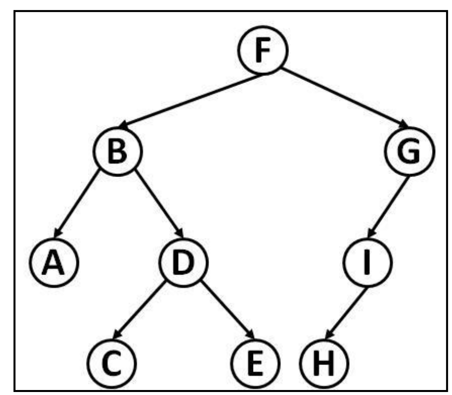
Preorder: <u>F, B, A, D, C, E, G</u>, I, H [PARENT LEFT RIGHT]

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





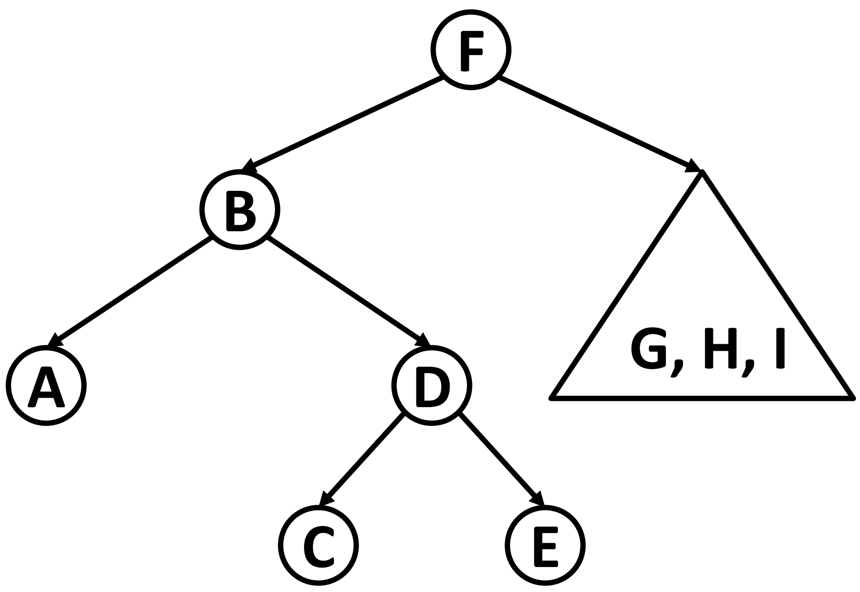




Preorder and Postorder (Full Tree)

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

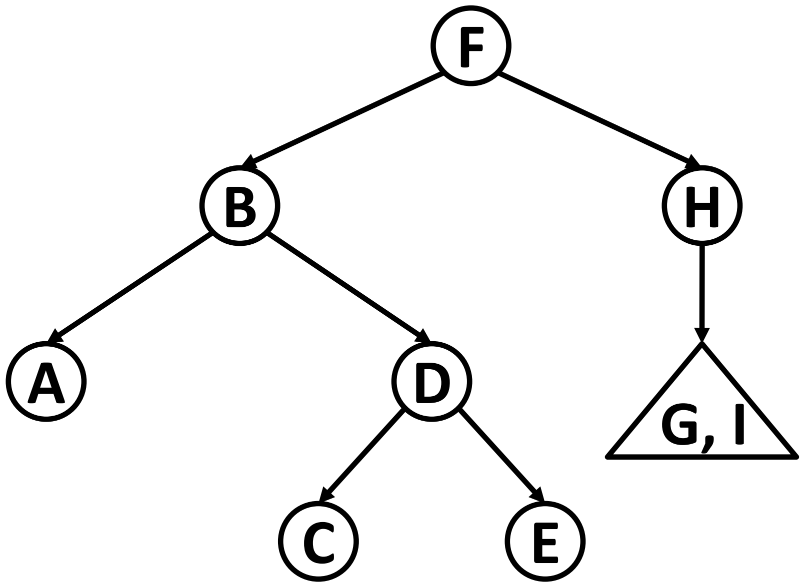
Postorder: A, C, E, D, B, G, I, H, F [LEFT RIGHT PARENT]



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]