

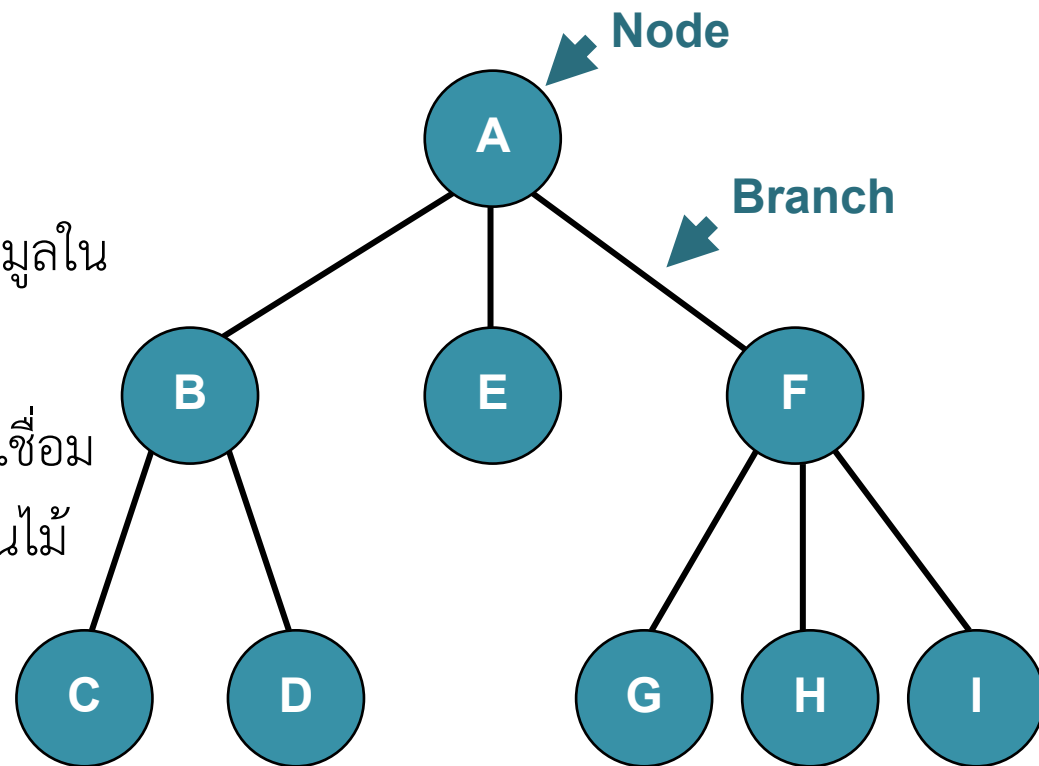


# Chapter 4

## Introduction to Trees & Binary Search Trees

# Trees

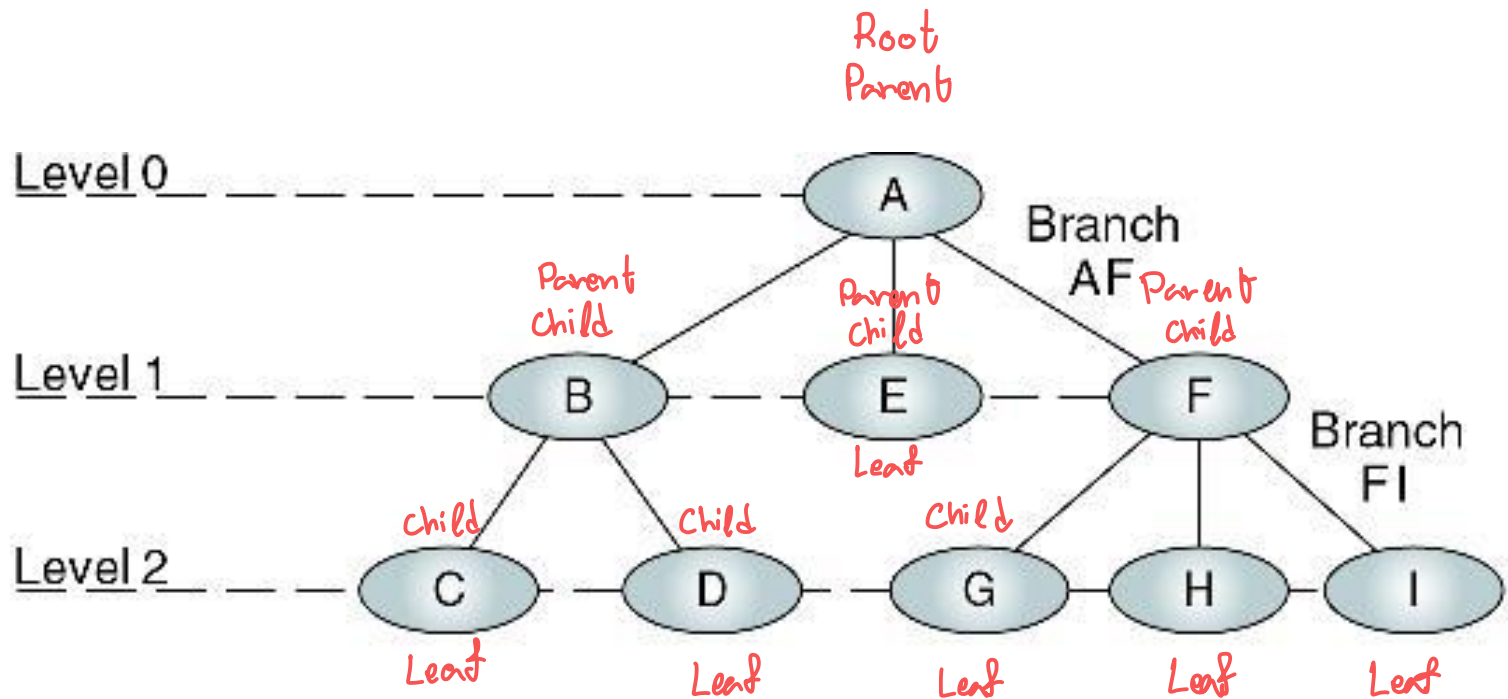
- เป็นโครงสร้างข้อมูล ที่ประกอบด้วย
  - โหนด (Node) เป็นกลุ่มข้อมูลในโครงสร้างต้นไม้
  - กิ่ง (Branch) เป็นส่วนที่ใช้เชื่อมต่อโหนดต่างๆ ในโครงสร้างต้นไม้



# Terminology

- Root : โหนดแรกสุดของต้นไม้
- Parent : โหนดที่อยู่สูงกว่าโหนดที่พิจารณา 1 ระดับ
- Child : โหนดที่อยู่ต่ำกว่าโหนดที่พิจารณา 1 ระดับ
- Sibling : โหนดที่อยู่ระดับเดียวกันและมีพ่อเดียวกัน
- Leaf : โหนดที่ไม่มีลูก
- Ancestor : โหนดที่อยู่ในเส้นทางการเดินจากรูทมายังโหนดที่พิจารณา
- Descendent : โหนดที่อยู่ในเส้นทางการเดินจากโหนดที่พิจารณา ไปจนหมดต้นไม้
- Degree : จำนวนลูกของโหนดที่พิจารณา
- Level : ระยะทางจากรูทมายังโหนดที่พิจารณา
- Depth : ความสูงของต้นไม้ (= Level ของโหนดที่อยู่ห่างจากรูทมากที่สุด + 1)
- Subtree : ต้นไม้ย่อย
- Internal : ไม่ใช่ root และไม่ใช่ leaf

# Terminology (cont.)

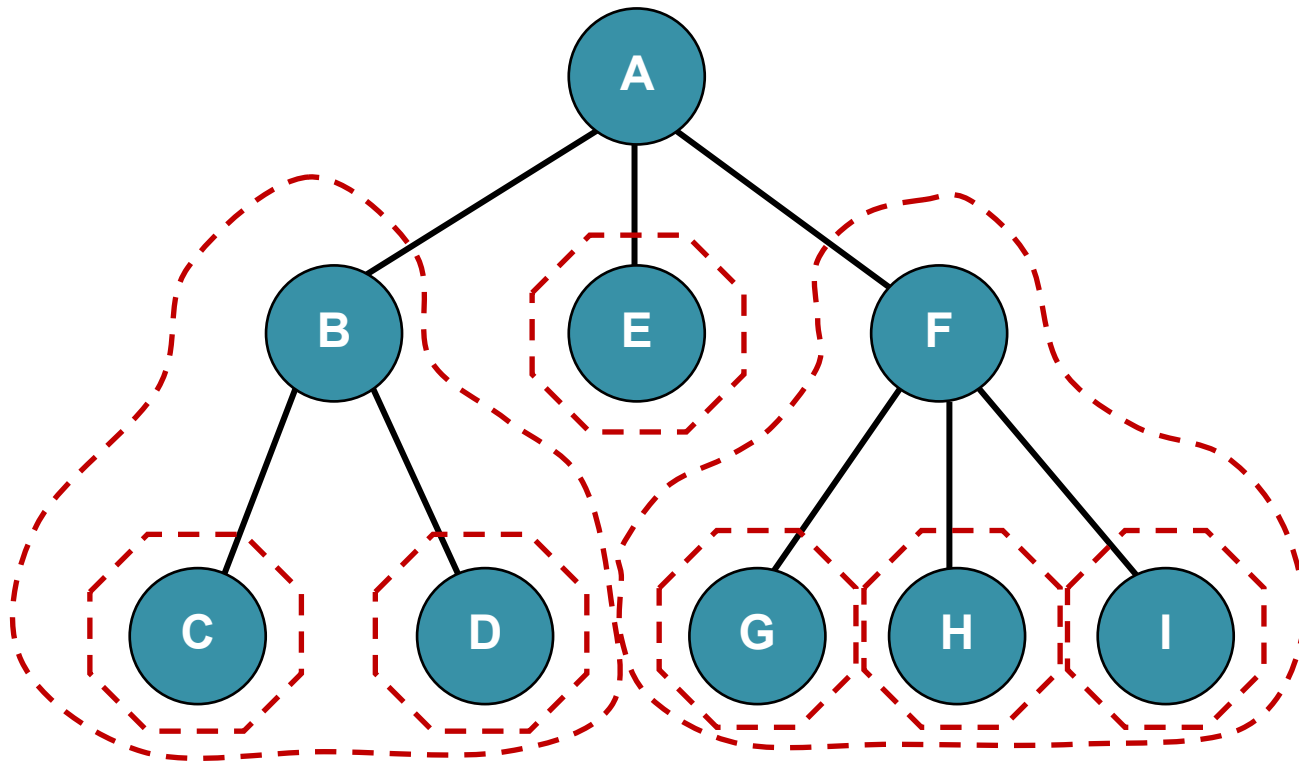


Root: A  
Parents: A, B, F  
Children: B, E, F, C, D, G, H, I

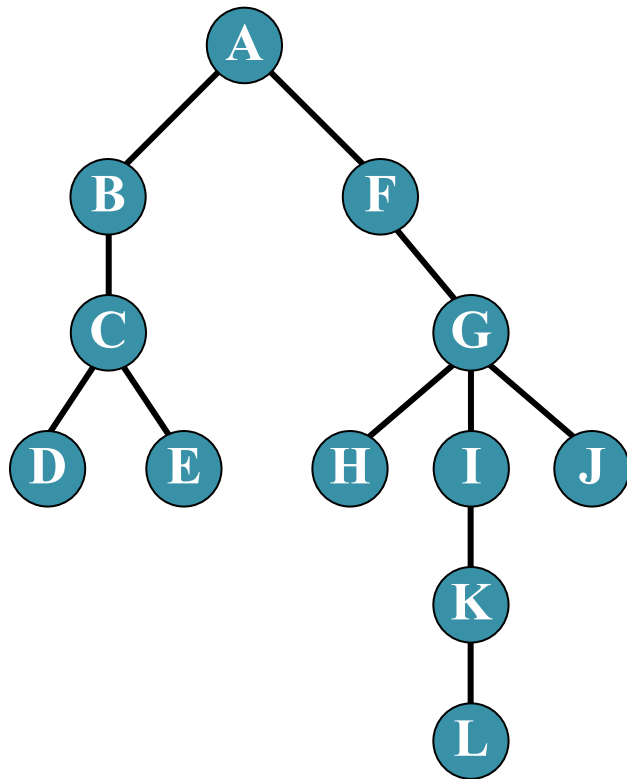
Siblings: {B, E, F}, {C, D}, {G, H, I}  
Leaves: C, D, E, G, H, I  
Internal nodes: B, F

# Subtrees

- โครงสร้างต้นไม้ย่อยๆ ในต้นไม้ใหญ่ทั้งต้น



# Terminology (cont.)

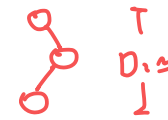


หาคำตอบต่อไปนี้

- Root : **A**
- Leaves : **D E H L J**
- Internal nodes : **B C F G I K**
- Ancestors of H : **A F G**
- Descendents of F : **G H I J K L**
- Siblings of I : **H J**
- Degrees of L : **0**
- Parent of K : **I**
- Children of C : **D E**
- Depth : **6**
- **Highest** Levels of the tree : **5**
- Level of J : **3**
- Number of all subtrees : **11**

# Binary Trees

Depth สูงสุด



Depth น้อยสุด



- เป็นโครงสร้างต้นไม้ ที่แต่ละโหนดจะมีโหนดลูกได้ไม่เกิน 2 ต้นไม้ย่อย
- ถ้าต้นไม้มีทั้งหมด  $N$  โหนด  $N = 3$

◦ Depth สูงสุด =  $N$  ; 3

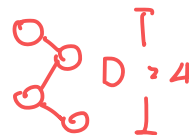
◦ Depth น้อยสุด =  $\log_2 N + 1$  ;  $\log_2 3 + 1$

- ถ้าต้นไม้มีความสูง  $D$   $D = 4$

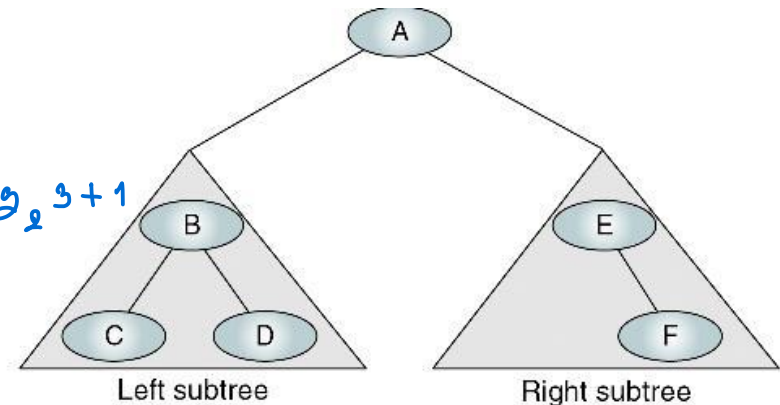
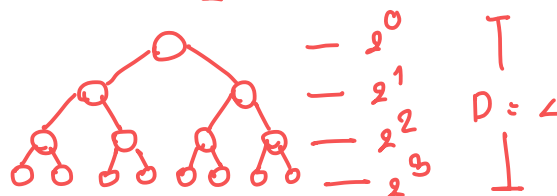
◦ จน.โหนดน้อยสุด =  $D$  ; 4

◦ จน.โหนดมากที่สุด =  $2^D - 1$  ;  $2^4 - 1$

Node น้อยสุด



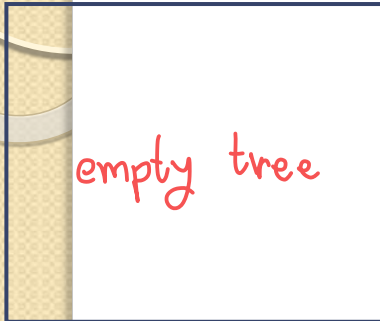
Node มากสุด



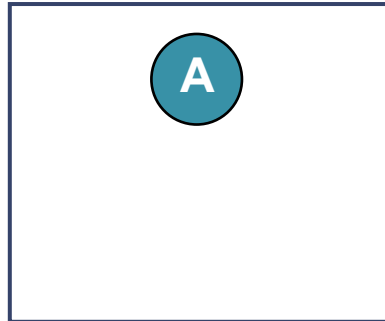


# Binary Trees

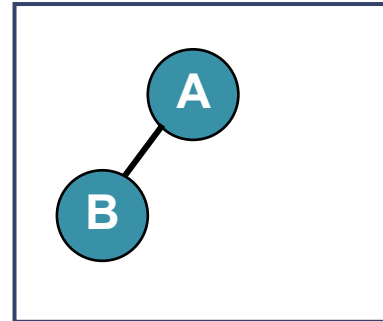
(a)



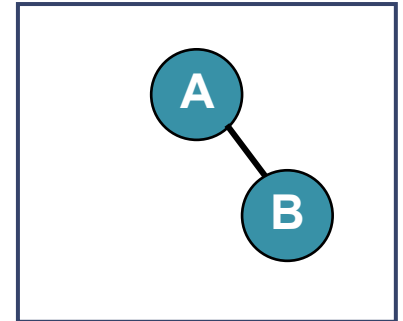
(b)



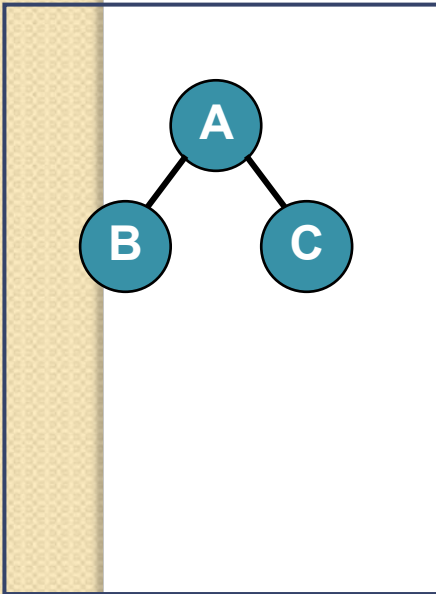
(c)



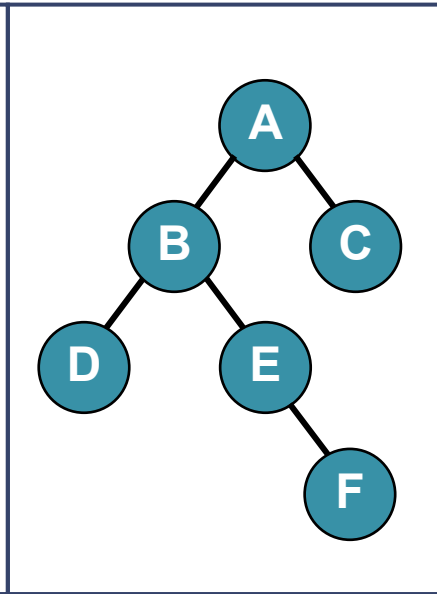
(d)



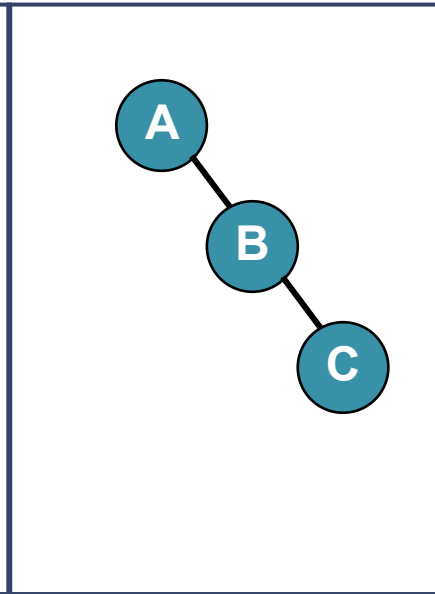
(e)



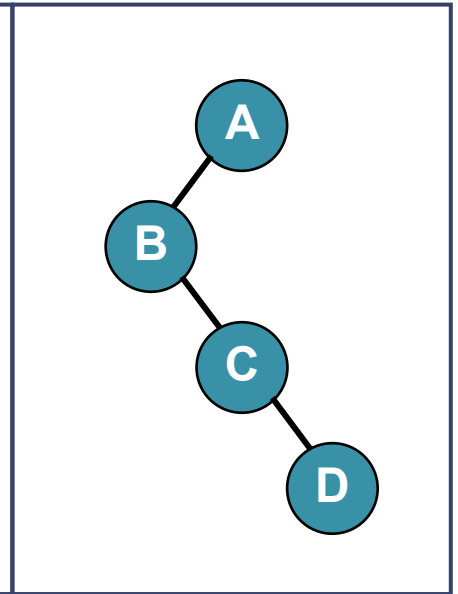
(f)



(g)



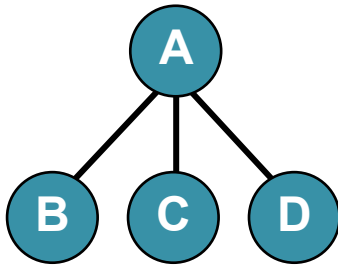
(h)



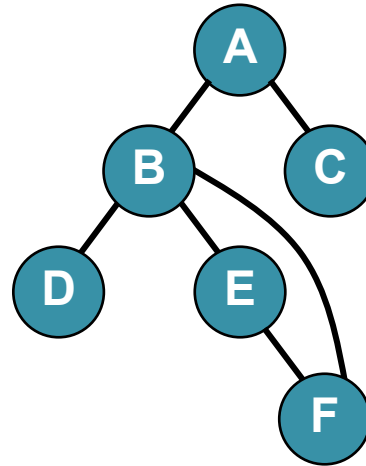


# Structures which are not binary trees

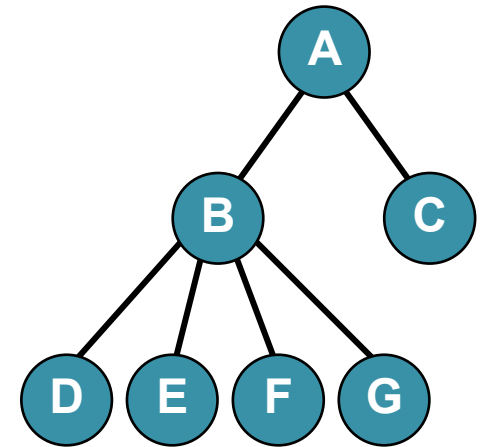
✓ Tree  
✗ Binary Tree



✗ Tree

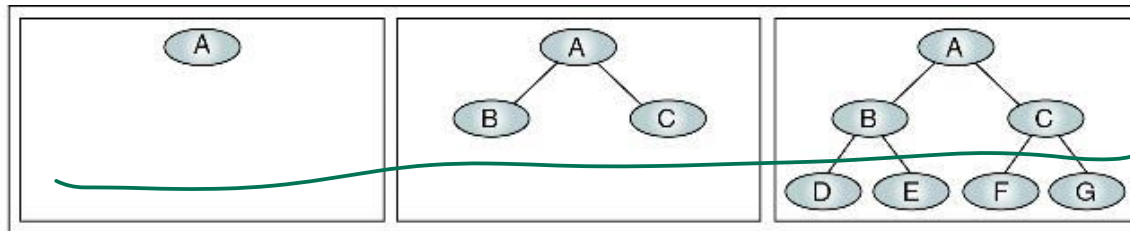


✓ Tree  
✗ Binary Tree



# Complete and Nearly Complete Trees

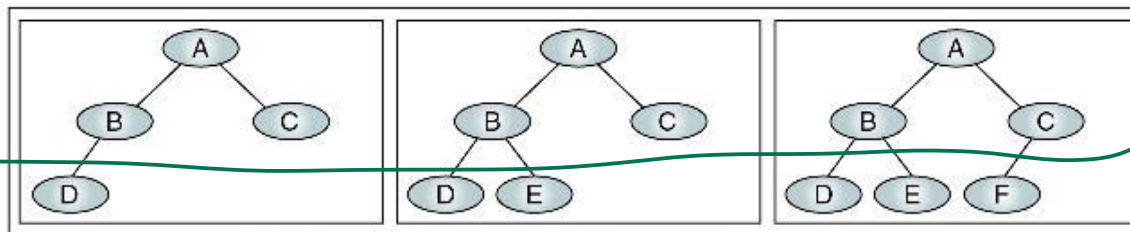
- Complete Trees : มีจ.โนหนดสูงสุด เมื่อมีความสูง D
- Nearly Complete Tree : มีความสูงน้อยสุด เมื่อมีโนหนด N โหนด  
*ทุก level except lowest level*



(a) Complete trees (at levels 0, 1, and 2)

$$D = 3$$

$$N = 2^3 - 1 = 7$$



(b) Nearly complete trees (at level 2)

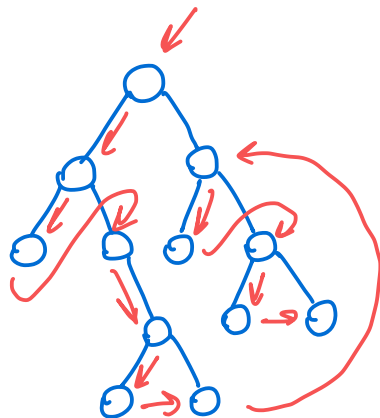
$$N = 4, 5, 6$$

$$D = \log_2 4 + 1 = 3$$

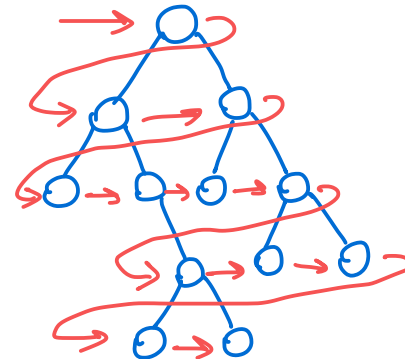
$$D = \log_2 5 + 1 = 3$$

# Binary Tree Traversals

- Depth-first traversal : เริ่มท่องจากรูท ไปใน descendent ของลูกตัวแรกจนหมด ค่อยไปท่องในลูกตัวถัด
- Breadth-first traversal : เริ่มท่องจากรูทแบบแนวกว้าง ไปยังลูกทุกตัวก่อน แล้วค่อยท่องไปที่ระดับลูกของลูกไปเรื่อยๆ



Depth-first

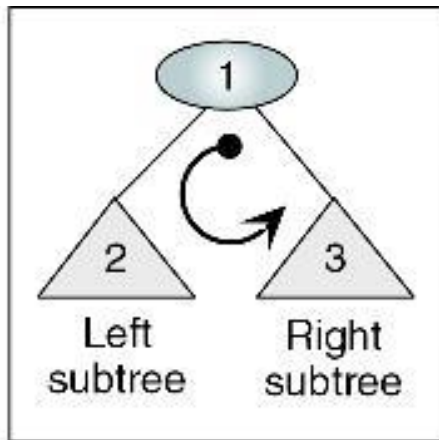


Breadth-first

# Depth-first Traversals

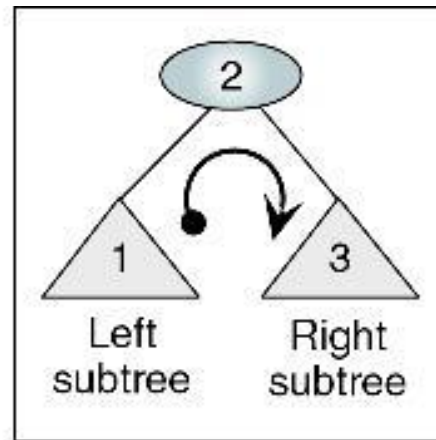
- เริ่มท่องจากรูท ไปใน descendent ของลูกตัวแรกจนหมด ค่อยไปท่องในลูกตัวถัด

Root  $\rightarrow$  L  $\rightarrow$  R



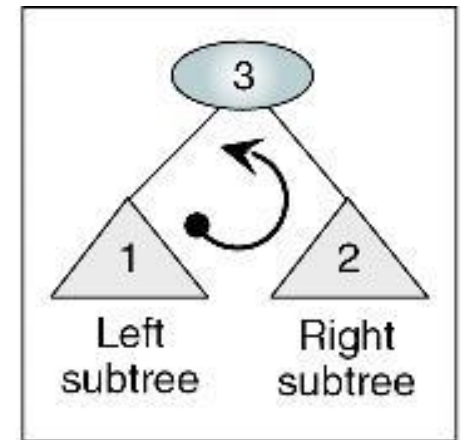
(a) Preorder traversal

L  $\rightarrow$  Root  $\rightarrow$  R



(b) Inorder traversal

L  $\rightarrow$  R  $\rightarrow$  Root



(c) Postorder traversal

วนกลับมาที่ root

# Preorder Traversal

- ท่องตามลำดับ root -> left subtree -> right subtree

```
Algorithm preOrder (root)
```

```
  Traverse a binary tree in node-left-right sequence.
```

```
    Pre  root is the entry node of a tree or subtree
```

```
    Post each node has been processed in order
```

```
1  if (root is not null)
```

```
    1  process (root)
```

```
    2  preOrder (leftSubtree)
```

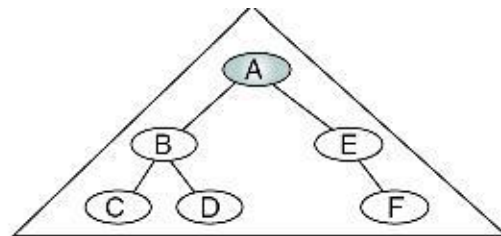
```
    3  preOrder (rightSubtree)
```

```
2  end if
```

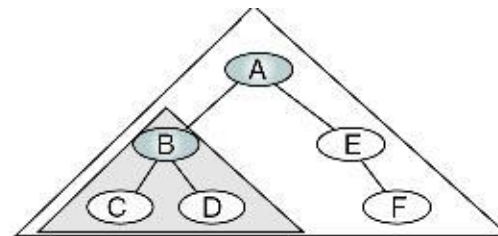
```
end preOrder
```



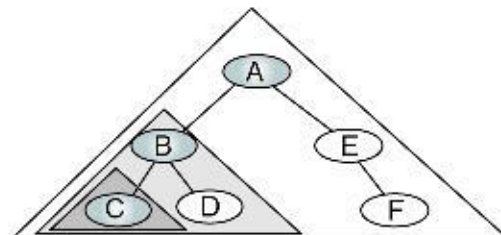
# Preorder Traversal (cont.)



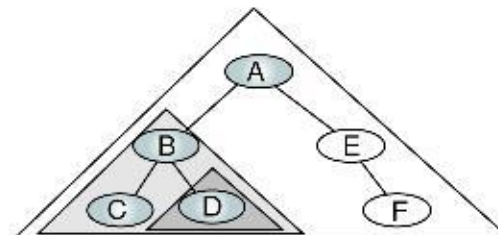
(a) Process tree A



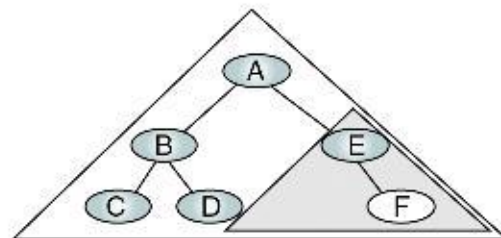
(b) Process tree B



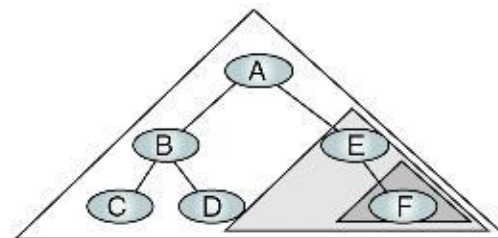
(c) Process tree C



(d) Process tree D



(e) Process tree E

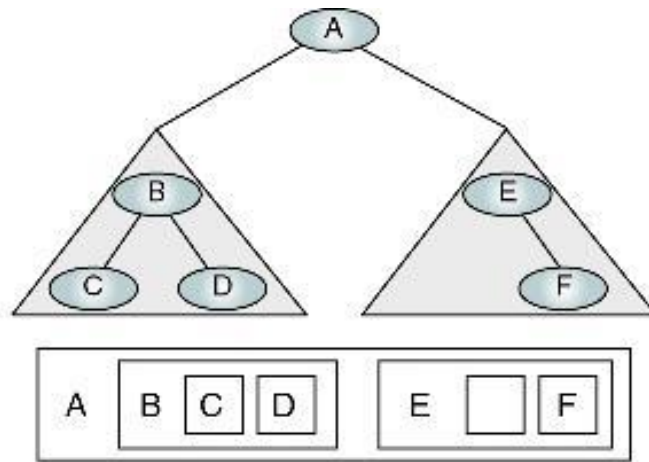


(f) Process tree F

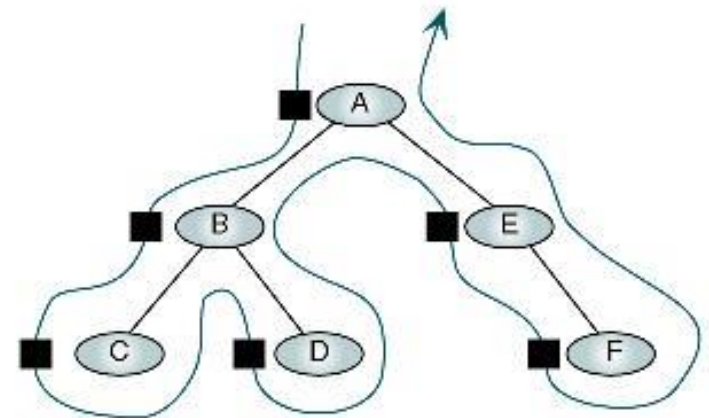
---

Algorithmic Traversal of Binary Tree

# Preorder Traversal (cont.)



(a) Processing order



(b) "Walking" order

---

Preorder Traversal—A B C D E F



# Inorder Traversal

- ท่องตามลำดับ left subtree -> root -> right subtree

```
Algorithm inOrder (root)
```

```
  Traverse a binary tree in left-node-right sequence.
```

```
    Pre  root is the entry node of a tree or subtree
```

```
    Post each node has been processed in order
```

```
1  if (root is not null)
```

```
    1  inOrder (leftSubTree)
```

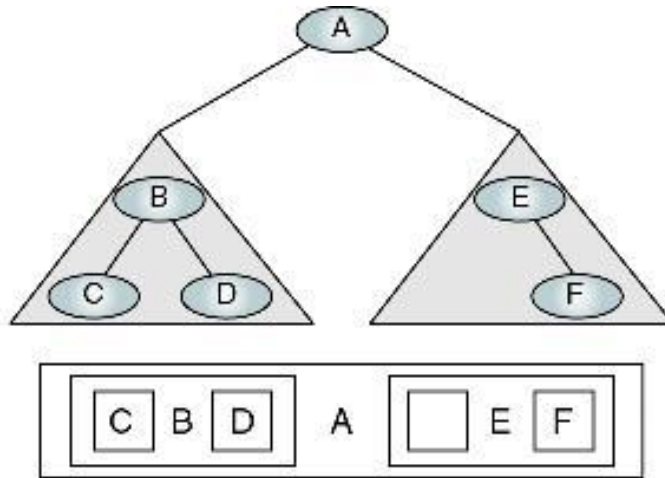
```
    2  process (root)
```

```
    3  inOrder (rightSubTree)
```

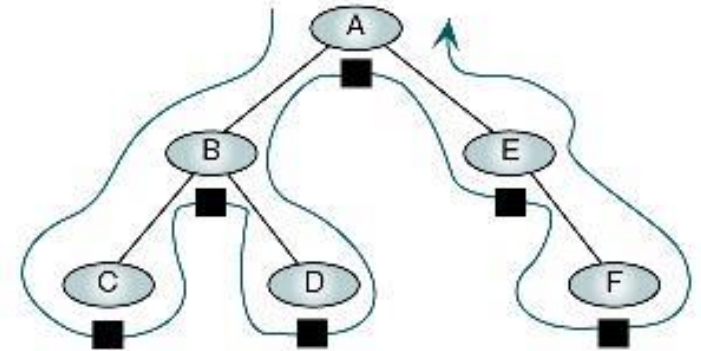
```
2  end if
```

```
end inOrder
```

# Inorder Traversal (cont.)



(a) Processing order



(b) "Walking" order

Inorder Traversal—C B D A E F

# Postorder Traversal

- ท่องตามลำดับ left subtree -> right subtree -> root

```
Algorithm postOrder (root)
```

```
  Traverse a binary tree in left-right-node sequence.
```

```
    Pre  root is the entry node of a tree or subtree
```

```
    Post each node has been processed in order
```

```
1  if (root is not null)
```

```
    1  postOrder (left subtree)
```

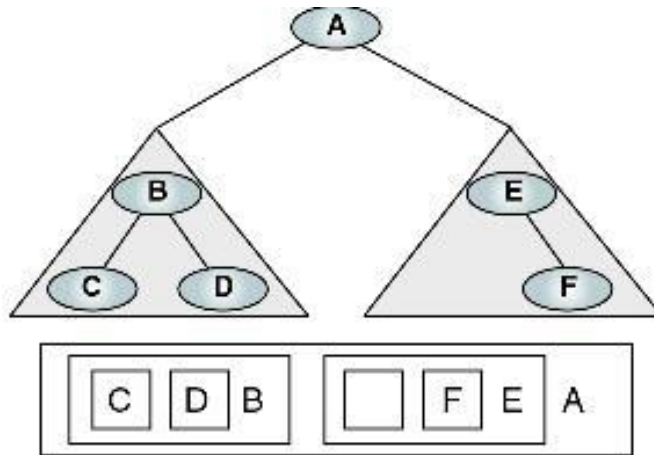
```
    2  postOrder (right subtree)
```

```
    3  process (root)
```

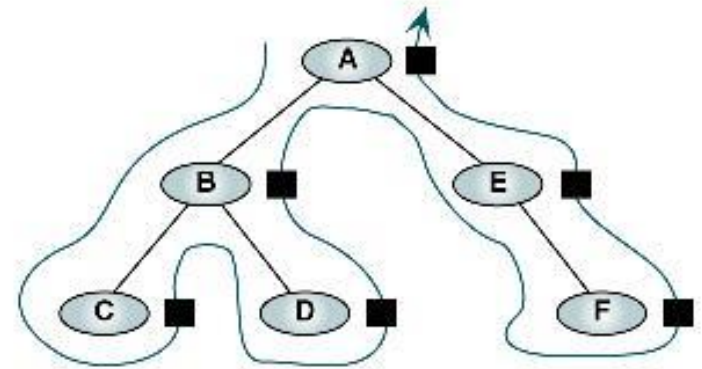
```
2  end if
```

```
end postOrder
```

# Postorder Traversal



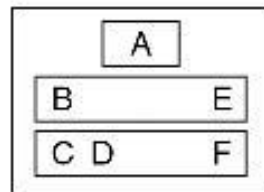
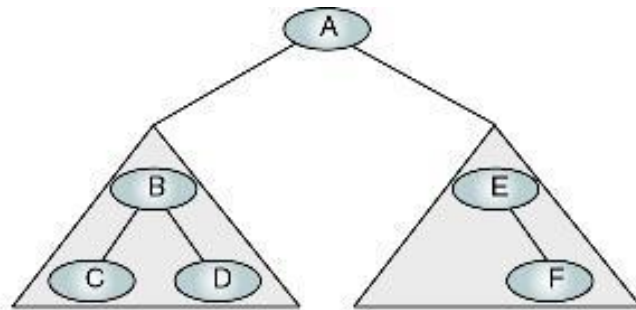
(a) Processing order



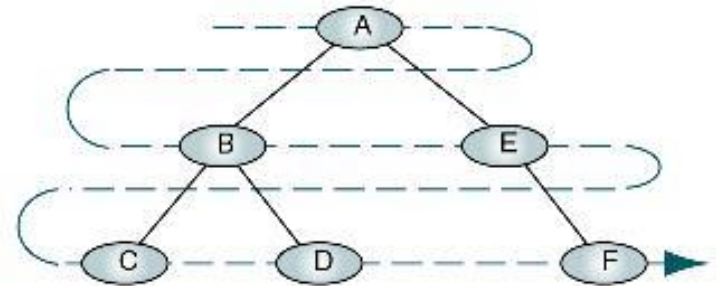
(b) "Walking" order

Postorder Traversal—C D B F E A

# Breadth-first Tree



(a) Processing order



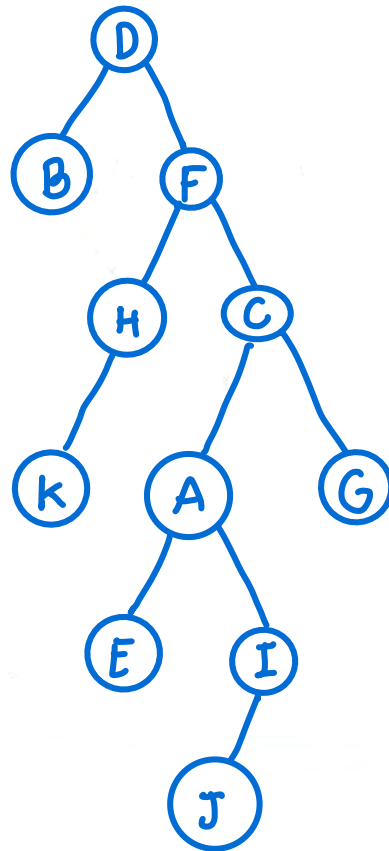
(b) "Walking" order

---

## Breadth-first Traversal



Root: D



Depth - first :

- Preorder

D B F H K C A E I J G

- Inorder

B D K H F E A J I C G

- Postorder

B K H E J I A G C F D

Breadth - first :

D B F H C K A G E I J

# Quiz

- ให้หาารูทของ Binary Tree เมื่อมีลำดับการท่องเข้าไปในต้นไม้ดังนี้
  - a) Tree with postorder traversal: FCBDG
  - b) Tree with preorder traversal: IBCDFEN
  - c) Tree with inorder traversal: CBIDFGE
- ให้วาดรูป Binary Tree เมื่อมีลำดับการท่องเข้าไปในต้นไม้ดังนี้
  - Preorder : JCBADEFIGH
  - Inorder : ABCEDFJGIH
- ให้วาดรูป Binary Tree ที่เป็นไปได้ เมื่อมีโหนดทั้งหมด 3 โหนด (A,B,C) (บอกด้วยว่าเป็น complete, nearly complete หรือไม่)
- ให้วาดรูป Nearly complete binary tree ที่มีการท่องไปในต้นไม้แบบ Breadth-first traversal เป็น JCBADEFIG

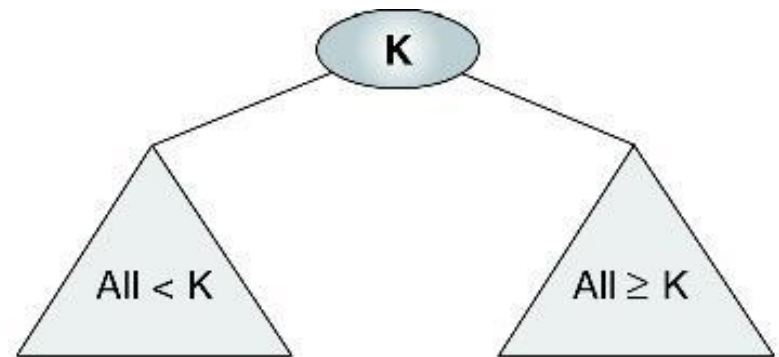




# Binary Search Tree

# Binary Search Tree

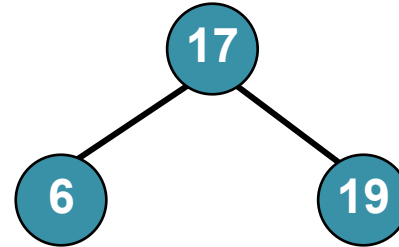
- เป็น Binary Tree ประเภทหนึ่ง โดยกำหนดให้
  - ทุกโหนดใน left subtree ต้องมีค่าน้อยกว่า root
  - ทุกโหนดใน Right Subtree ต้องมีค่ามากกว่าหรือเท่ากับ root
  - ในแต่ละ Subtree ต้องคงคุณสมบัติของ Binary Search Tree



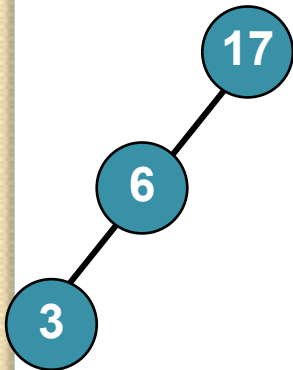
# Valid Binary Search Trees



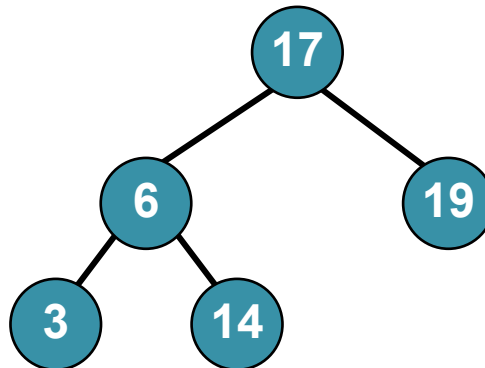
(a)



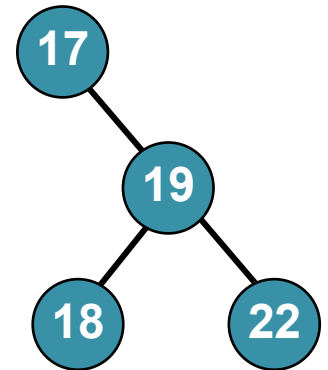
(b)



(c)

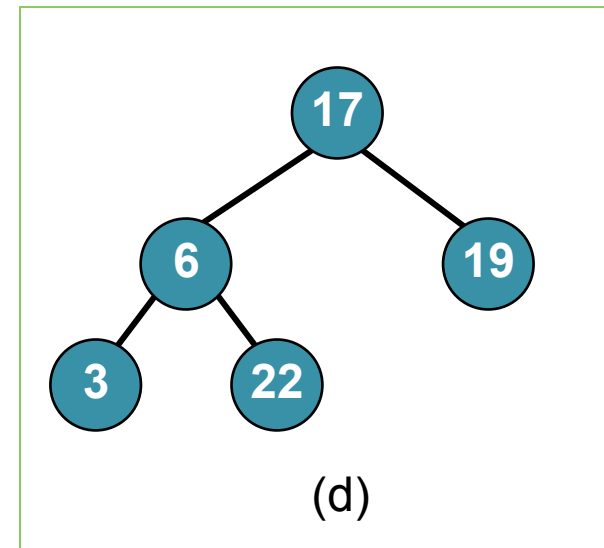
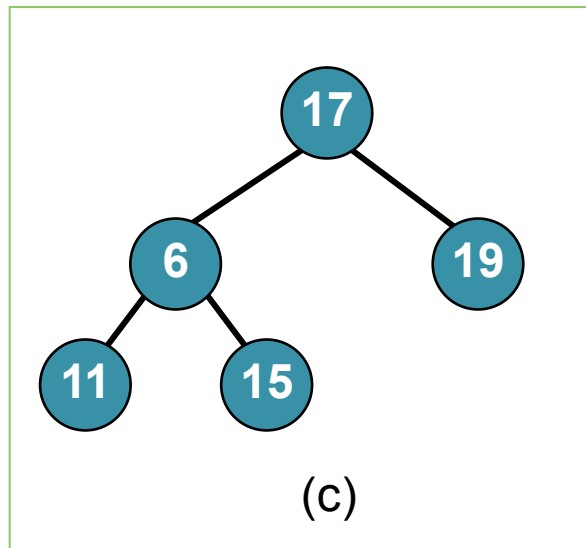
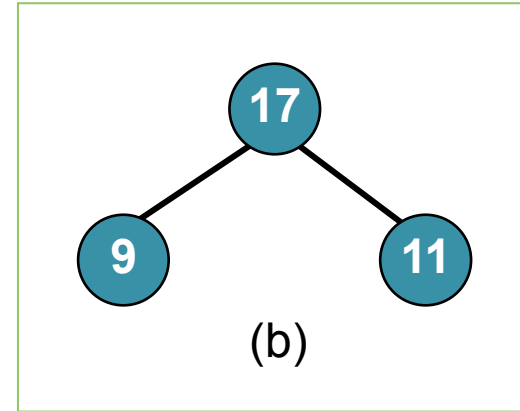
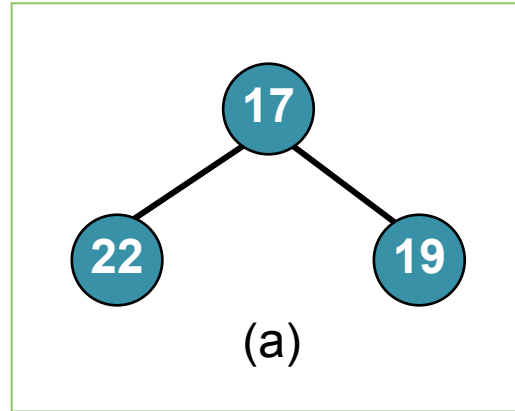


(d)

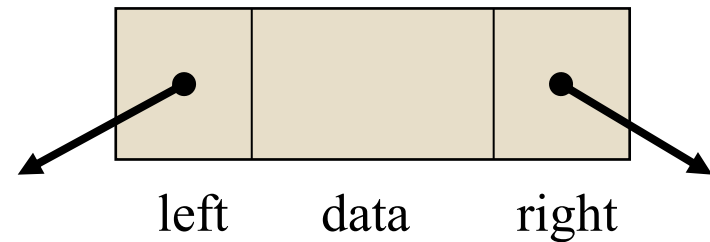
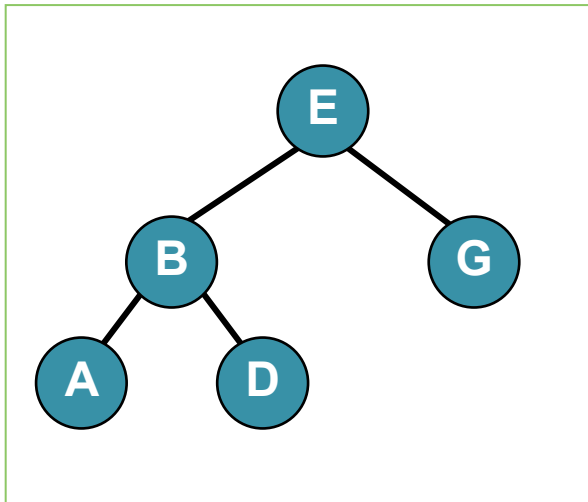


(e)

# Invalid Binary Search Trees



# BST Representations



BSTNode

dataType   data

BSTNode left, right

end BSTNode

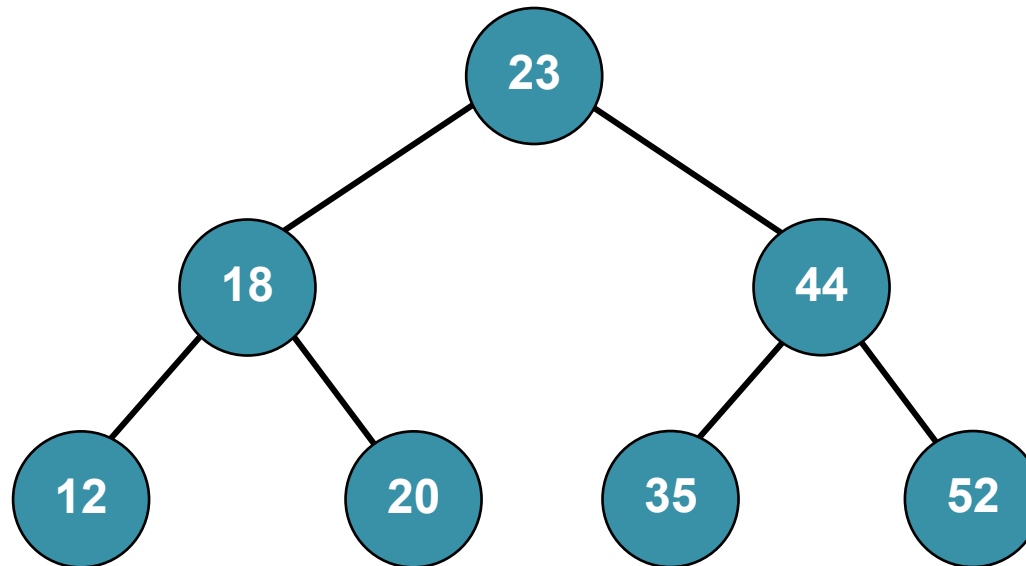
BSTNode root



# BST Operations

- Traversals
- Search
- Insertion
- Delete

# Traversals



- Preorder Traversal :
- Inorder Traversal :
- Postorder Traversal :

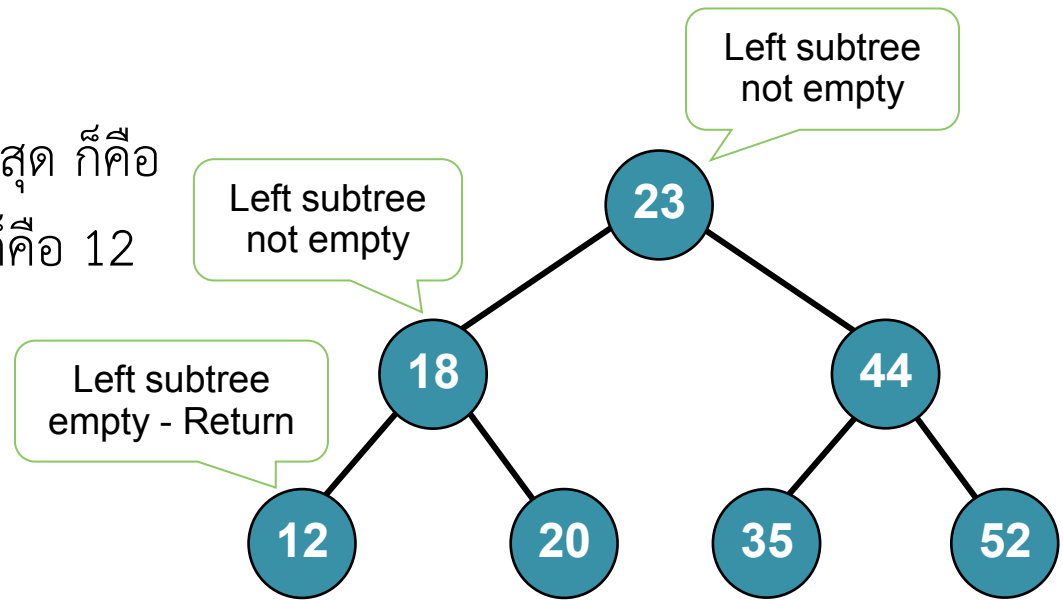


# Search

- การค้นหาโหนดที่มีค่าน้อยสุด
- การค้นหาโหนดที่มีค่ามากที่สุด
- การค้นหาโหนดที่ต้องการ

# Find the Smallest Node

- คิดง่าย ๆ -> โหนดที่มีค่าน้อยสุด ก็คือ โหนดที่อยู่ซ้ายสุดนั่นเอง ซึ่งก็คือ 12
- ท่องไปทางซ้ายเรื่อยๆ



**Algorithm** findSmallestBST (root)

This algorithm finds the smallest node in a BST.

Pre root is a pointer to a nonempty BST or subtree

Return address of smallest node

1 if (left subtree empty)

1 return (root)

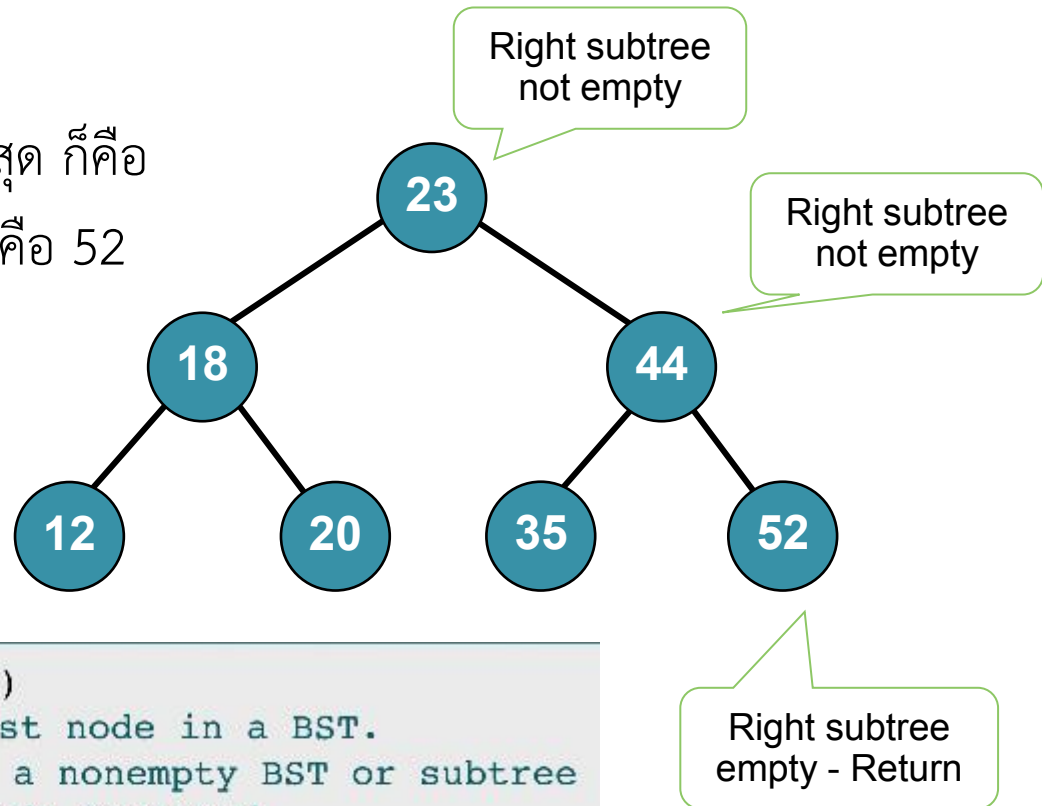
2 end if

3 return findSmallestBST (left subtree)

end findSmallestBST

# Find the Largest Node

- คิดง่าย ๆ -> โหนดที่มีค่ามากที่สุด ก็คือ โหนดที่อยู่ขวาสุดนั่นเอง ซึ่งก็คือ 52
- ท่องไปทางขวาเรื่อยๆ



**Algorithm** findLargestBST (root)

This algorithm finds the largest node in a BST.

Pre root is a pointer to a nonempty BST or subtree

Return address of largest node returned

1 if (right subtree empty)

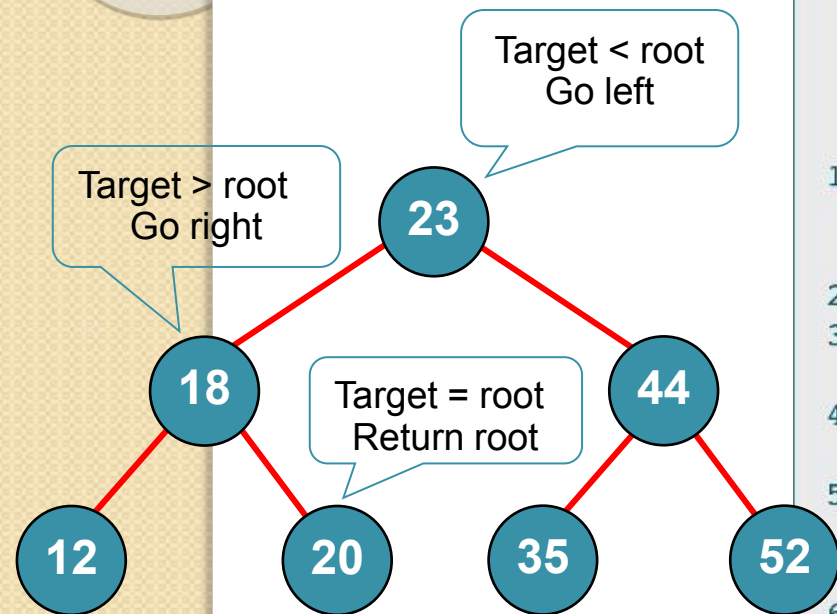
1 return (root)

2 end if

3 return findLargestBST (right subtree)

end findLargestBST

# Searching a Given Node in a BST

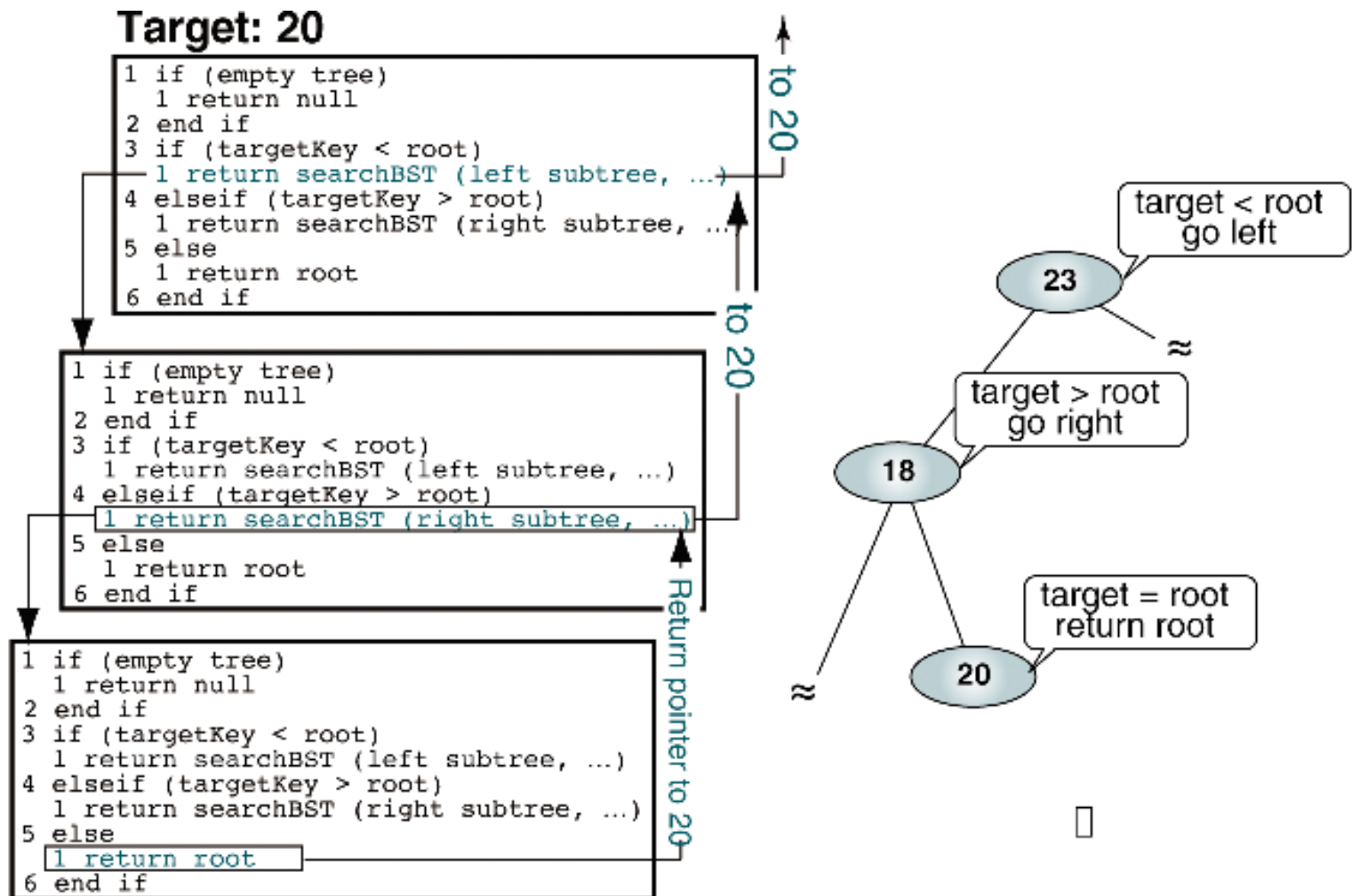


**Target : 20**

```
Algorithm searchBST (root, targetKey)
Search a binary search tree for a given value.
Pre    root is the root to a binary tree or subtree
       targetKey is the key value requested
Return the node address if the value is found
       null if the node is not in the tree

1  if (empty tree)
    Not found
    1  return null
2  end if
3  if (targetKey < root)
    1  return searchBST (left subtree, targetKey)
4  else if (targetKey > root)
    1  return searchBST (right subtree, targetKey)
5  else
    Found target key
    1  return root
6  end if
end searchBST
```

# Searching a Given Node in a BST

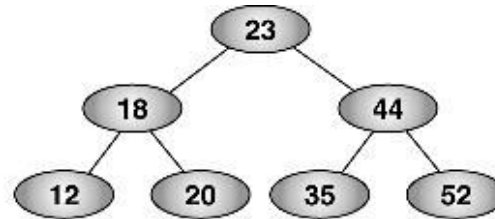




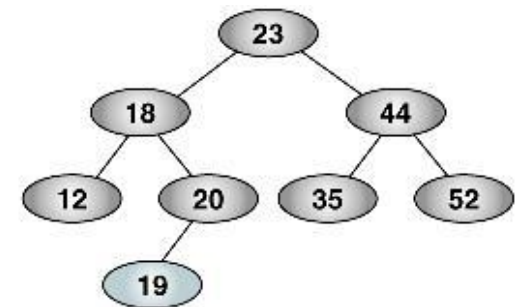
# Insertion

แทรกข้อมูลเข้าไปใน BST โดย

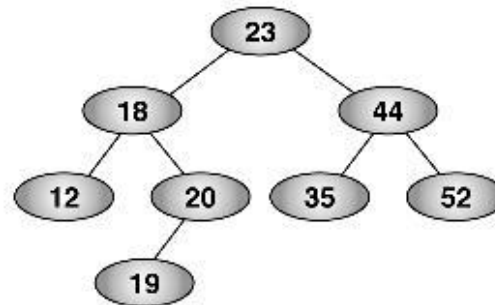
- ตามกิ่ง (Branch) ไปเรื่อยๆ จนเจอ subtree ที่ว่าง
- แล้วแทรกโหนดข้อมูลใหม่เข้าไป (แทรกจาก leaf node)



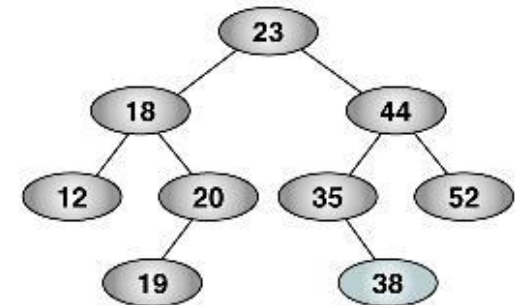
(a) Before inserting 19



(b) After inserting 19



(c) Before inserting 38



(d) After inserting 38

# Insertion (cont.)

**Algorithm** addBST (root, newNode)

Insert node containing new data into BST using recursion.

Pre      root is address of current node in a BST

          newNode is address of node containing data

Post     newNode inserted into the tree

Return address of potential new tree root

1 if (empty tree)

1    set root to newNode

2    return newNode

2 end if

Locate null subtree for insertion

3 if (newNode < root)

1    return addBST (left subtree, newNode)

4 else

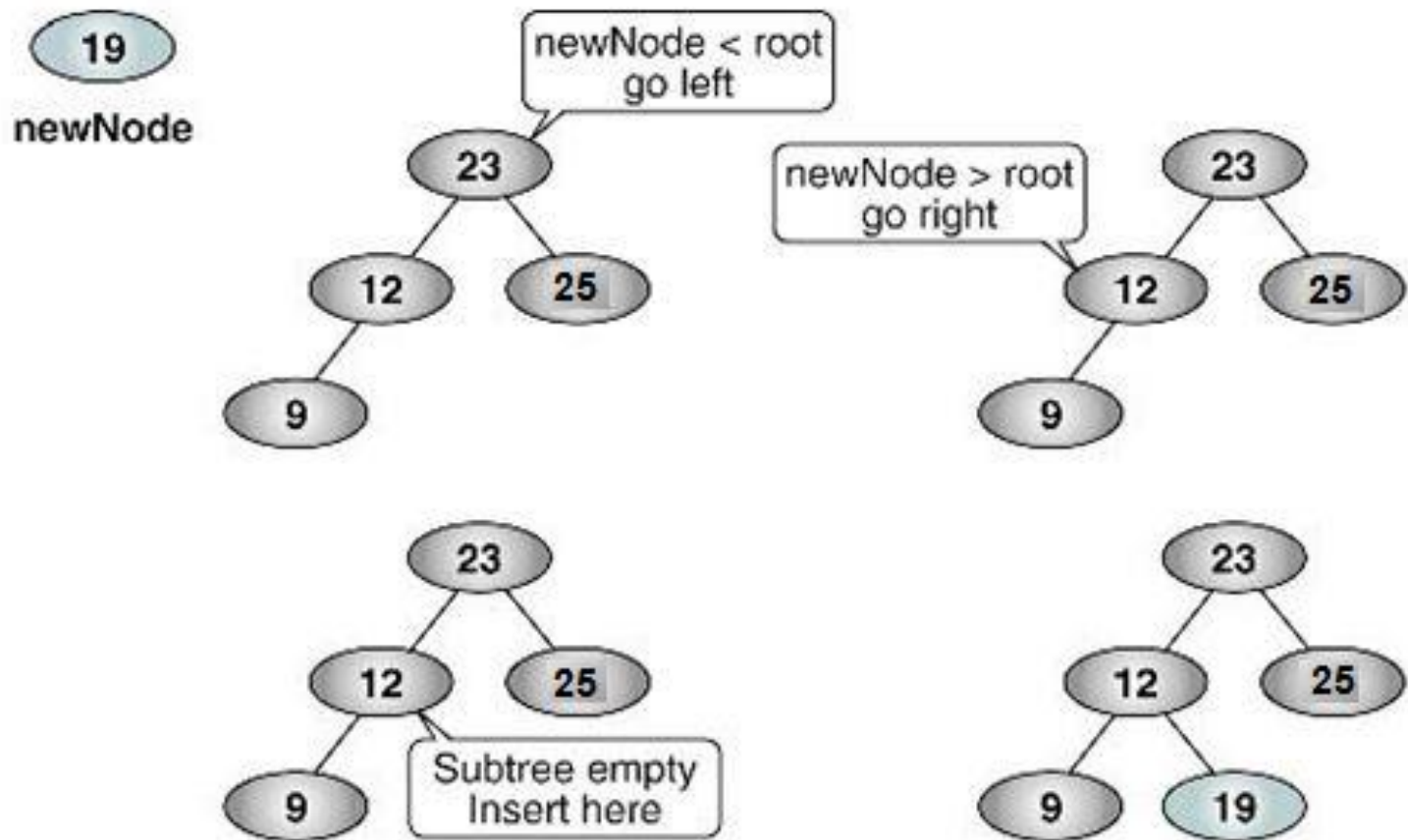
1    return addBST (right subtree, newNode)

5 end if

end addBST



# Insertion (cont.)



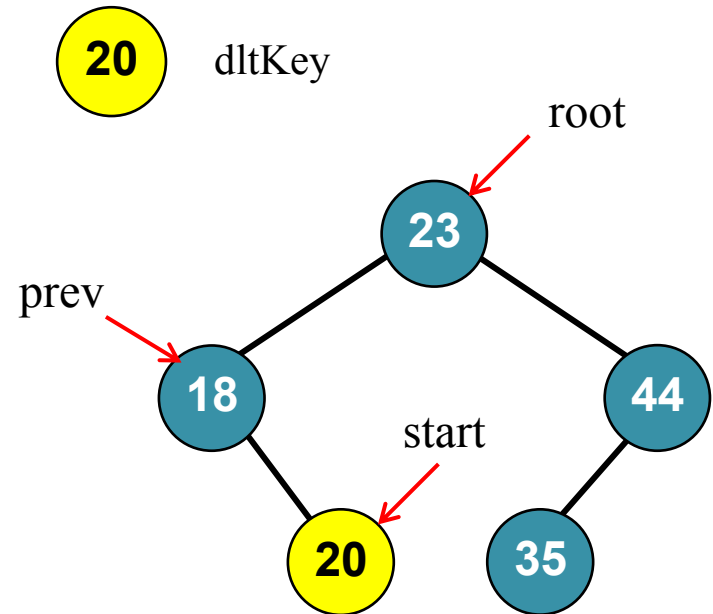
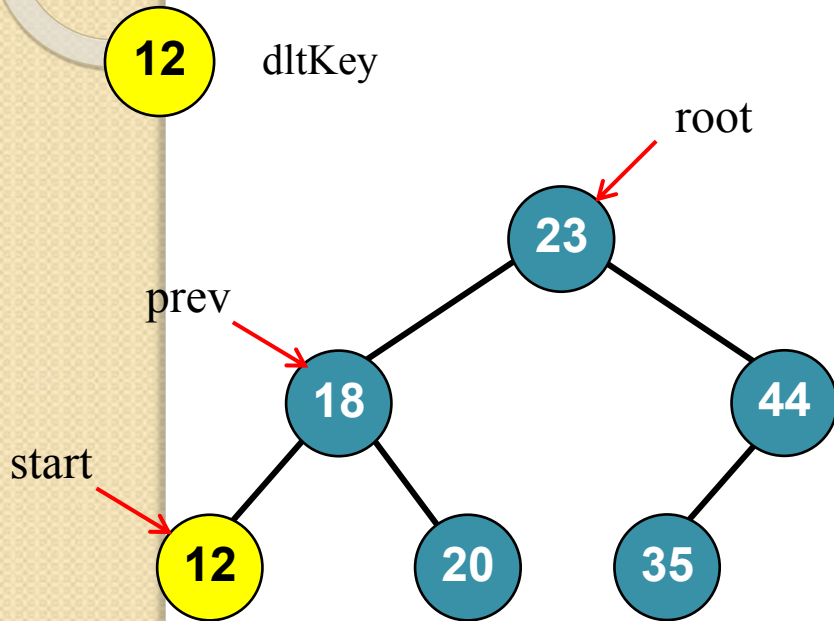
# Deletion

- จะลบข้อมูลจาก BST ต้องทำการค้นหาโหนดที่ต้องการจะลบให้ได้ก่อน ซึ่งโหนดเหล่านั้น แบ่งลักษณะได้เป็น 4 กรณี
  - โหนดที่จะลบเป็น leaf node (ไม่มีลูก) -> ลบได้เลย
  - โหนดที่จะลบ มีเพียง Right subtree -> เปลี่ยนให้พ่อของโหนดนั้นชี้ไปที่ Right subtree แทน แล้วลบโหนดนั้น
  - โหนดที่จะลบ มีเพียง Left subtree -> เปลี่ยนให้พ่อของโหนดนั้นชี้ไปที่ Left subtree แทน แล้วลบโหนดนั้น
  - โหนดที่จะลบมีทั้ง Right และ Left subtree -> คัดลอกข้อมูลมากที่สุดของ Left subtree มาที่โหนดที่จะลบ แล้วลบโหนด แล้วลบโหนดมากที่สุดของ Left subtree แทน

# Deletion (cont.)

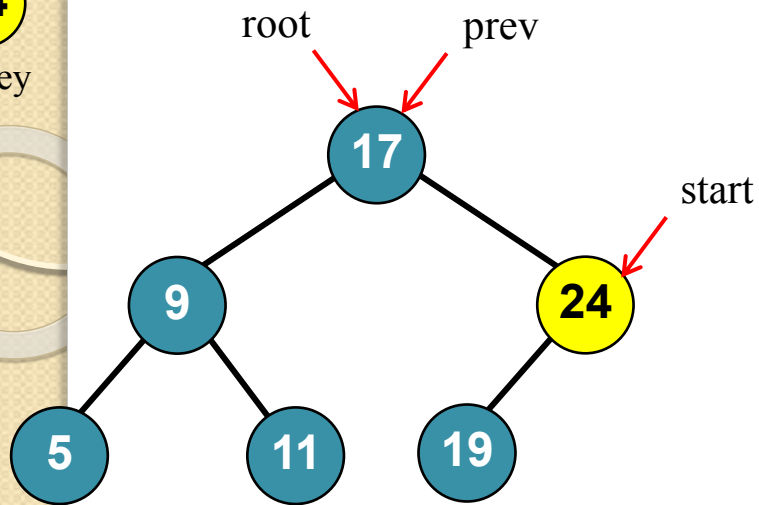
```
Algorithm deleteBST (root, dltKey)
This algorithm deletes a node from a BST.
    Pre    root is reference to node to be deleted
           dltKey is key of node to be deleted
    Post   node deleted
           if dltKey not found, root unchanged
    Return true if node deleted, false if not found
1  if (empty tree)
1  return false
2  end if
3  if (dltKey < root)
1  return deleteBST (left subtree, dltKey)
4  else if (dltKey > root)
1  return deleteBST (right subtree, dltKey)
5  else
    Delete node found--test for leaf node
1  If (no left subtree)
1  make right subtree the root
2  return true
2  else if (no right subtree)
1  make left subtree the root
2  return true
3  else
    Node to be deleted not a leaf. Find largest node on
    left subtree.
1  save root in deleteNode
2  set largest to largestBST (left subtree)
3  move data in largest to deleteNode
4  return deleteBST (left subtree of deleteNode,
                     key of largest
4  end if
6  end if
end deleteBST
```

# Case 1 : The node has no children



24

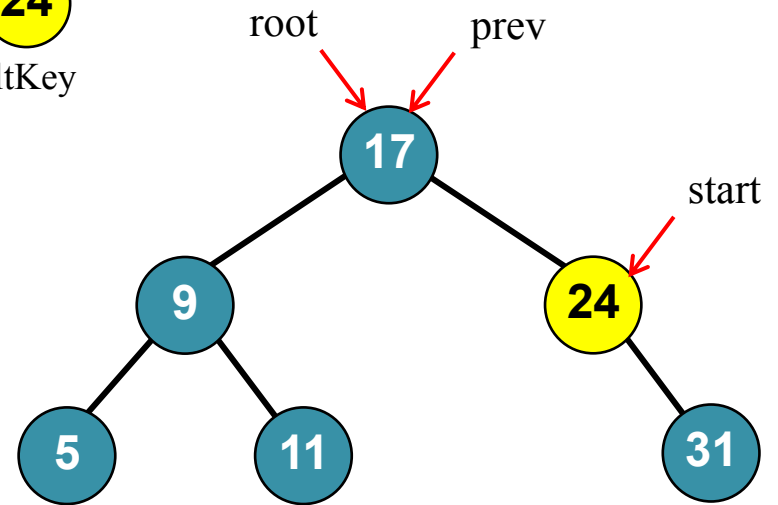
dltKey



prev.right = start.left

24

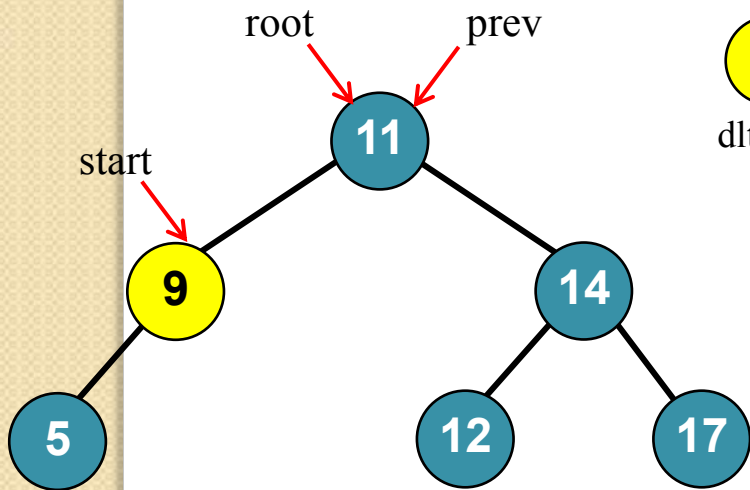
dltKey



prev.right = start.right

9

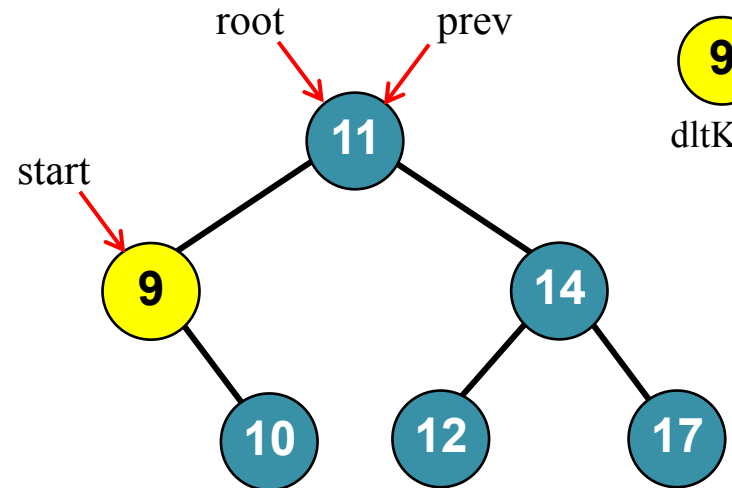
dltKey



prev.left = start.left

9

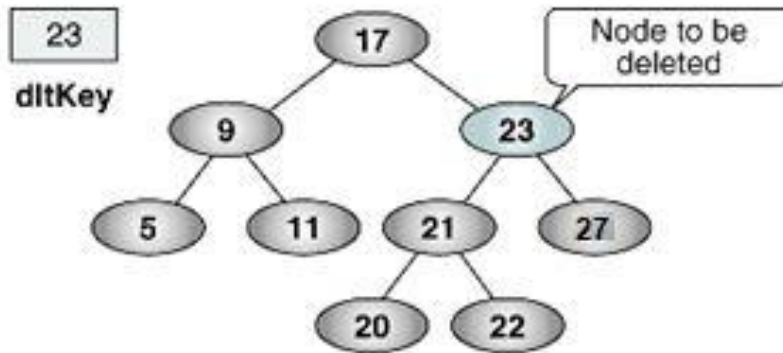
dltKey



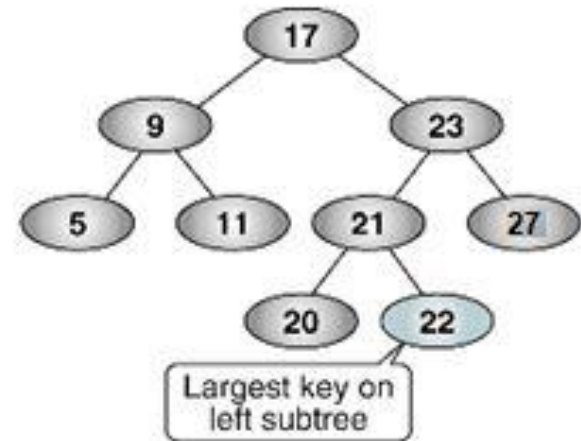
prev.left = start.right



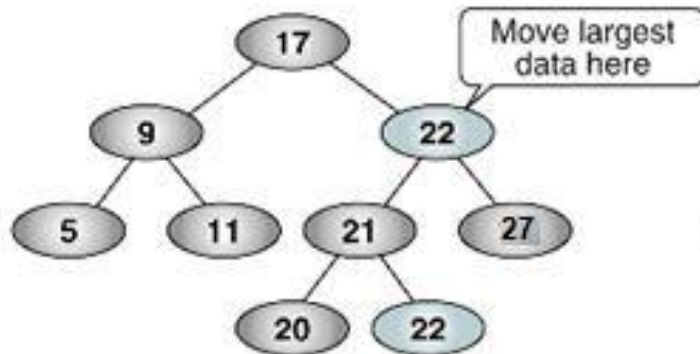
# Case 4 : The node have 2 subtrees



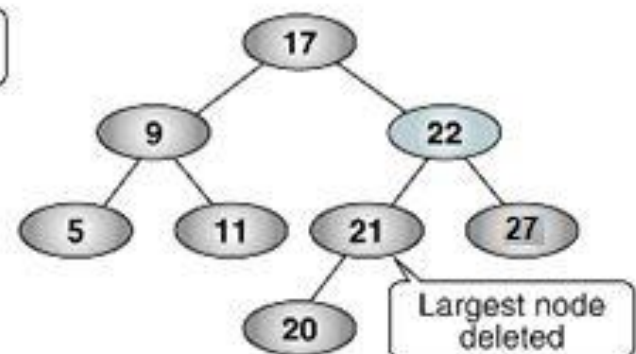
(a) Find dltKey



(b) Find largest



(c) Move largest data



(d) Delete largest node



# Quiz

- ให้วาดรูป BST ตามลำดับข้อมูลที่เข้าม้างนี้
  - 70, 80, 60, 90, 20, 40, 25, 82, 10, 33
- ให้วาดรูป BST ตามลำดับข้อมูลที่เข้าม้างนี้
  - 10, 20, 25, 33, 40, 60, 70, 80, 82, 90
- ให้ลบโหนดที่มีข้อมูล 70 ออกจาก BST แล้ววาดรูป BST ใหม่

