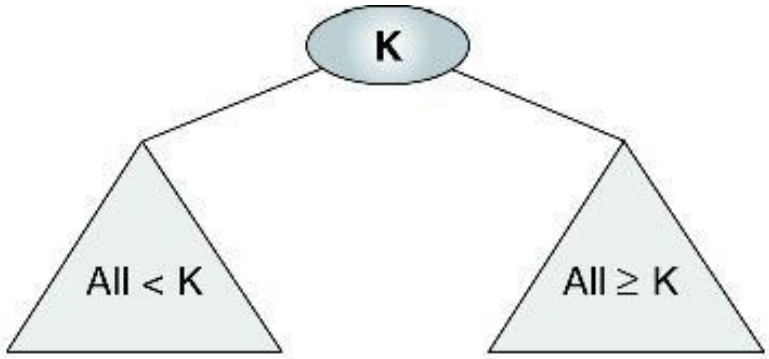
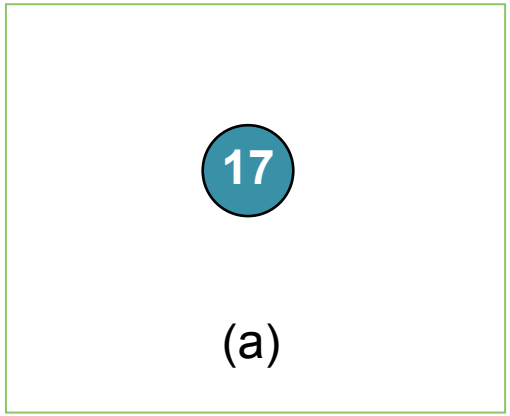
Binary Search Tree

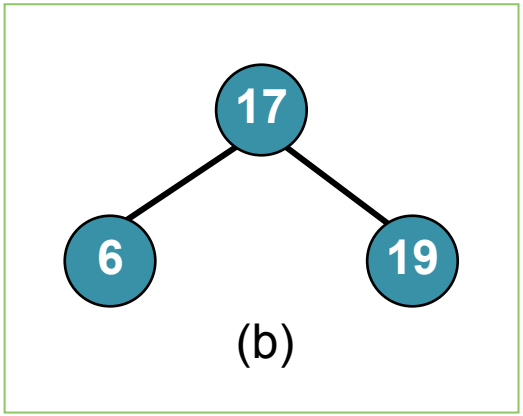
Binary Search Tree

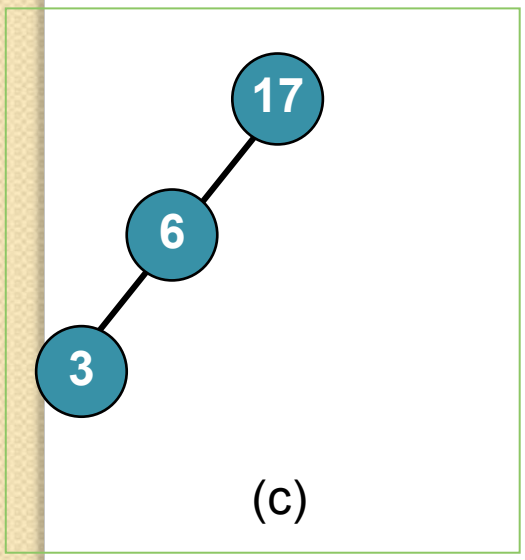
- เป็น Binary Tree ประเภทหนึ่ง โดยกำหนดให้
  - ทุกโหนดใน left subtree ต้องมีค่าน้อยกว่า root
  - ทุกโหนดใน Right Subtree ต้องมีค่ามากกว่าหรือเท่ากับ root
  - ในแต่ละ Subtree ต้องคงคุณสมบัติของ Binary Search Tree

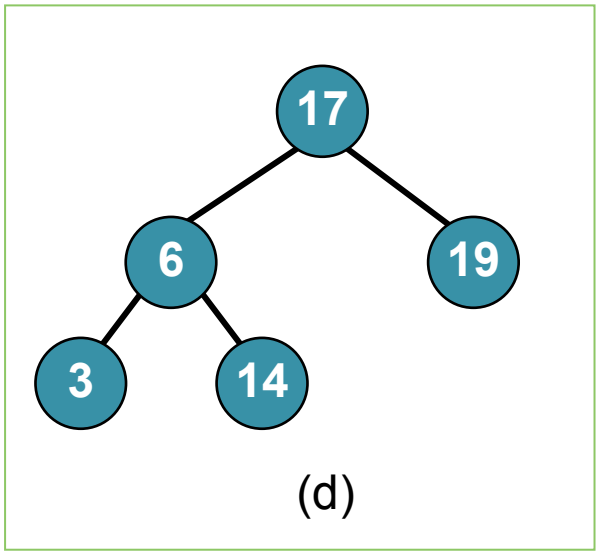


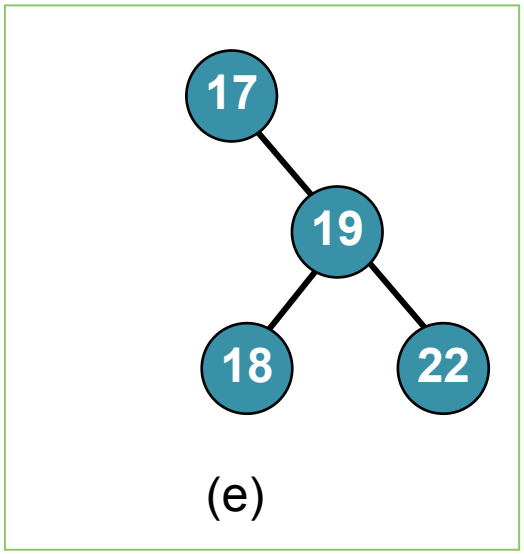
Valid Binary Search Trees



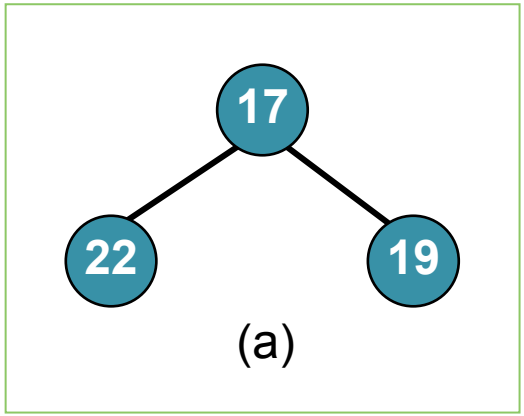


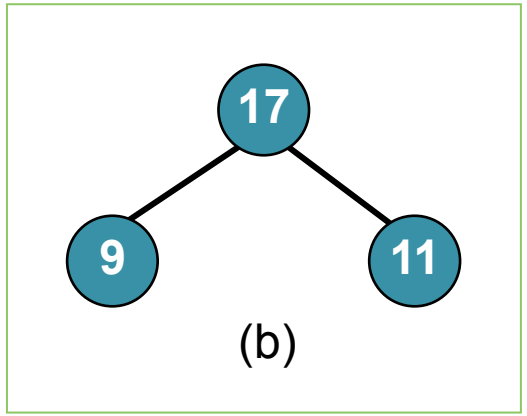


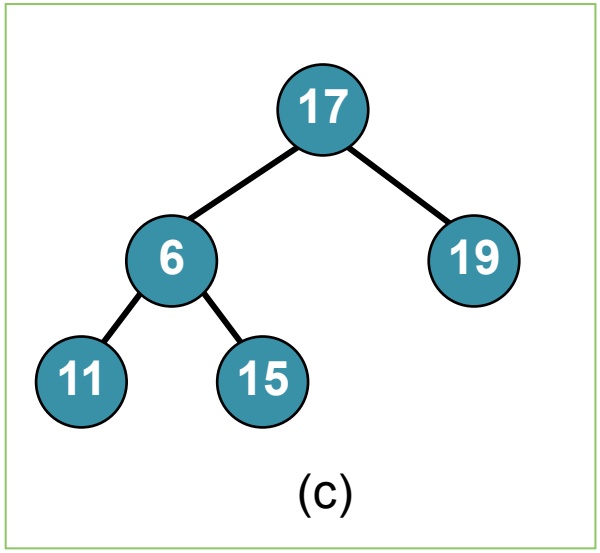


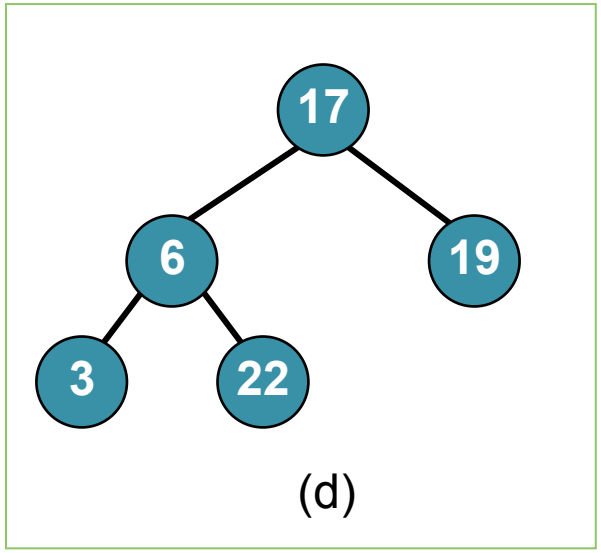


Invalid Binary Search Trees

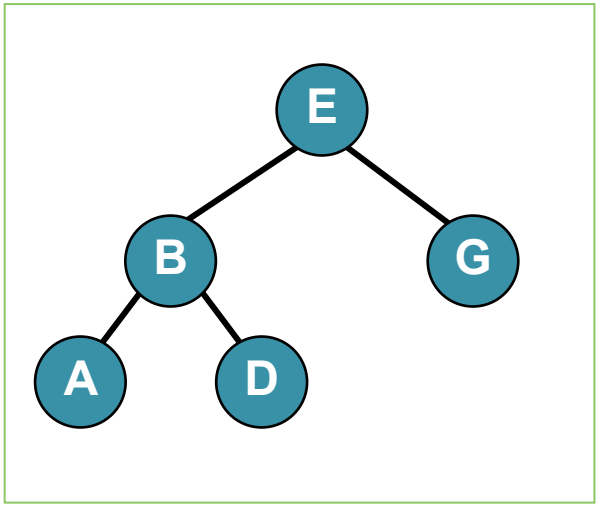


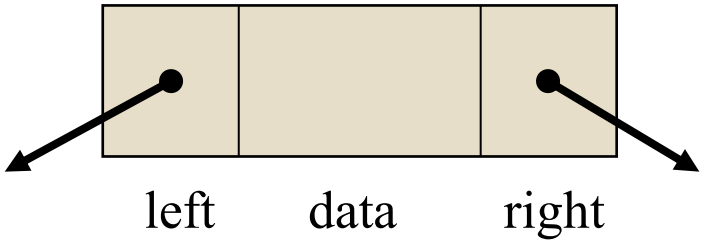






BST Representations



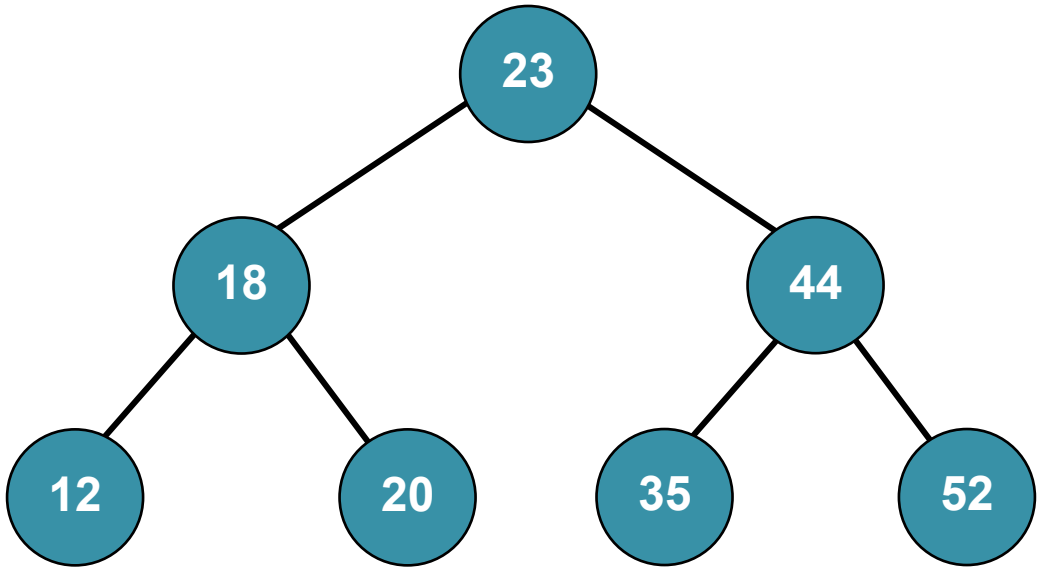



```
BSTNode

        dataType    data

        BSTNode left, right

end BSTNode

BSTNode root
```

BST Operations

- Traversals
- Search
- Insertion
- Delete

Traversals



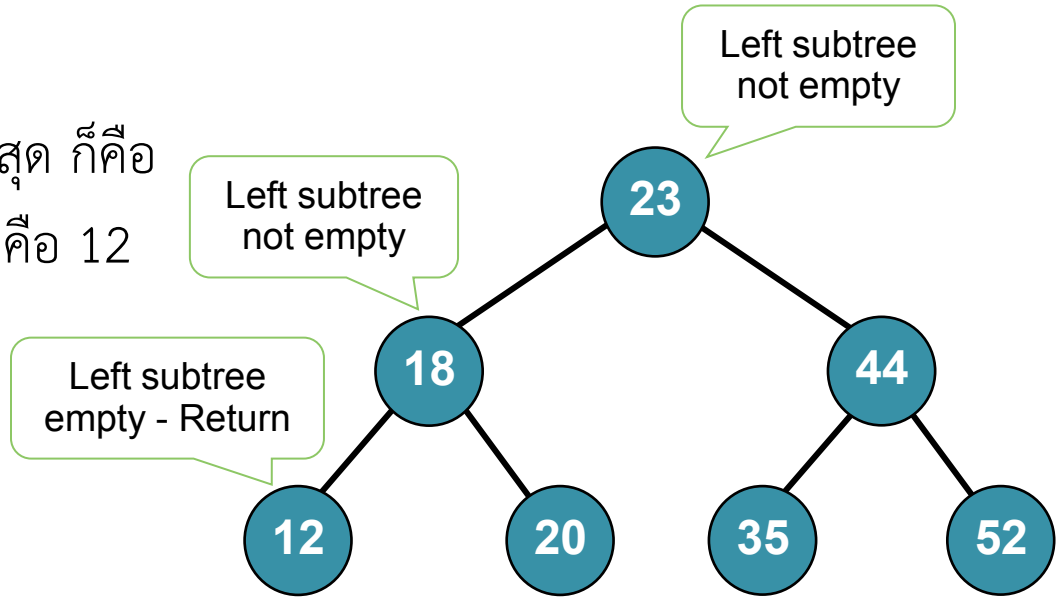
- Preorder Traversal : 23 18 12 20 44 35 52
- Inorder Traversal : 12 18 20 23 35 44 52
- Postorder Traversal : 12 20 18 35 52 44 23

Search

- การค้นหาโหนดที่มีค่าน้อยสุด
- การค้นหาโหนดที่มีค่ามากสุด
- การค้นหาโหนดที่ต้องการ

Find the Smallest Node

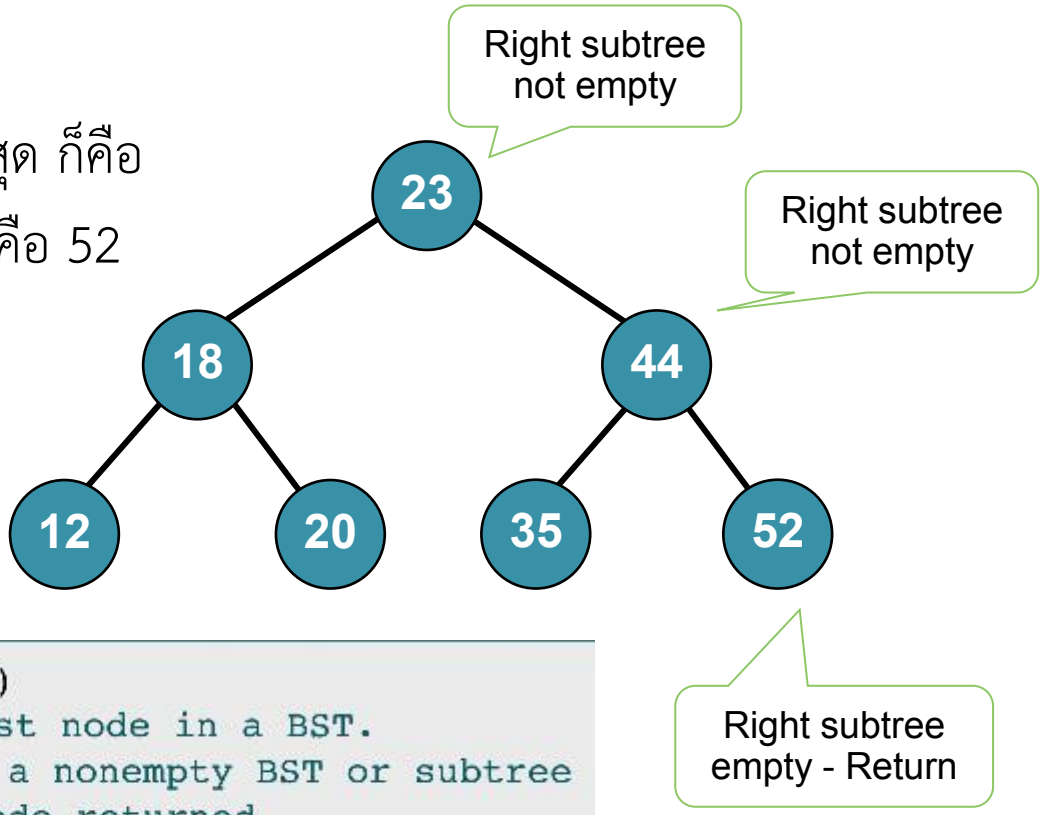
- คิดง่ายๆ -> โหนดที่มีค่าน้อยสุด ก็คือ โหนดที่อยู่ซ้ายสุดนั่นเอง ซึ่งก็คือ 12
- ท่องไปทางซ้ายเรื่อยๆ




```
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
```

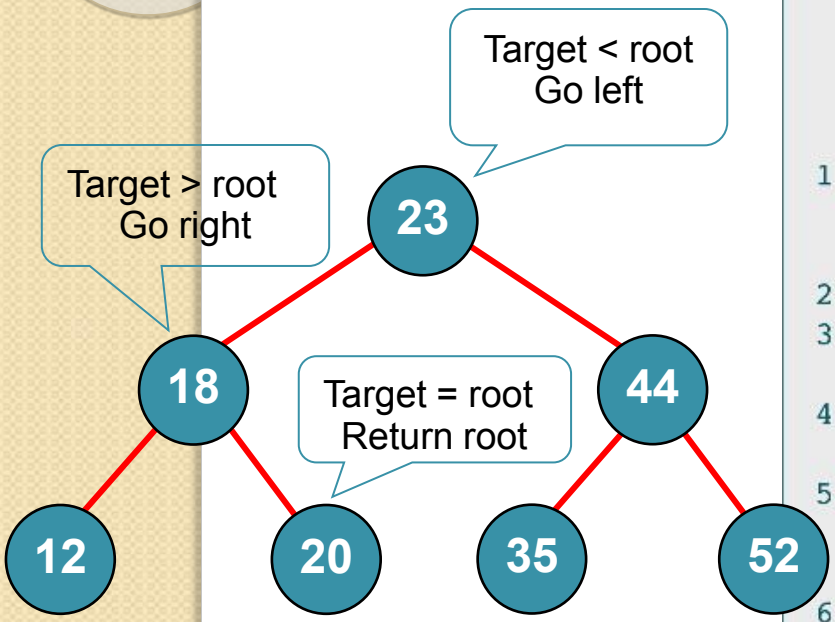
Find the Largest Node

- คิดง่ายๆ -> โหนดที่มีค่ามากสุด 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
```

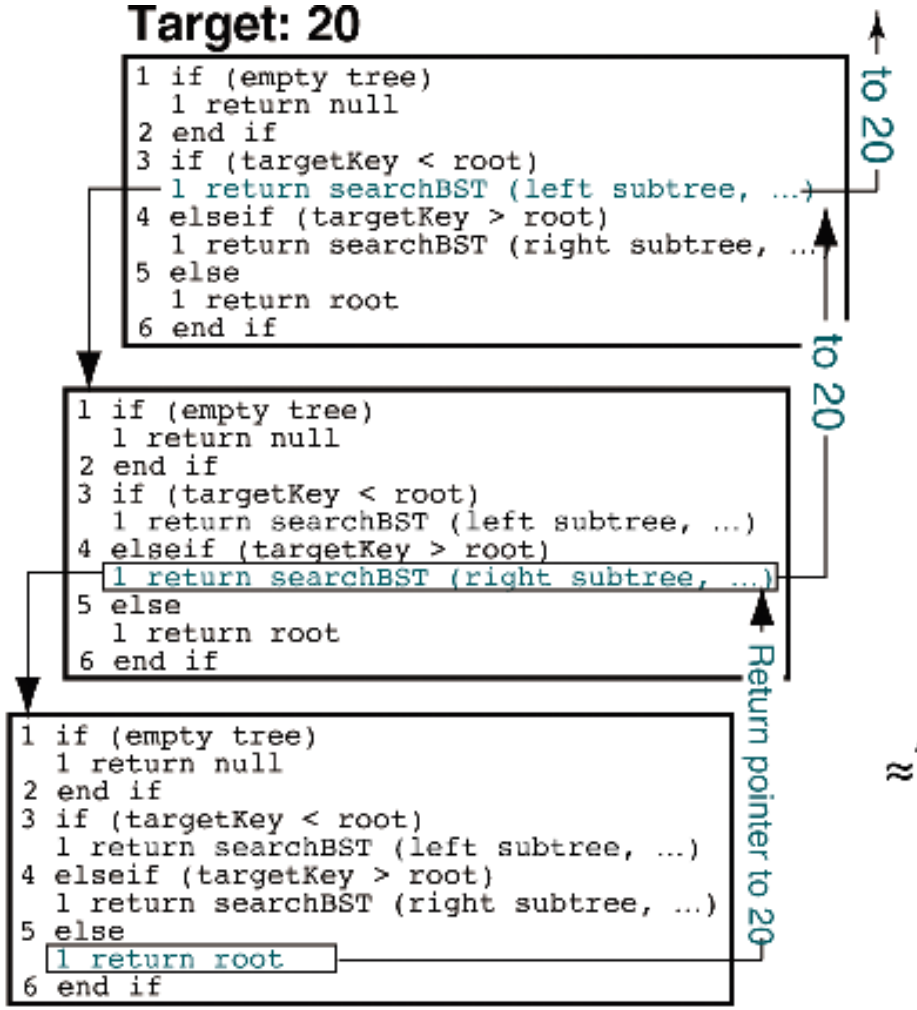
Searching a Given Node in a BST

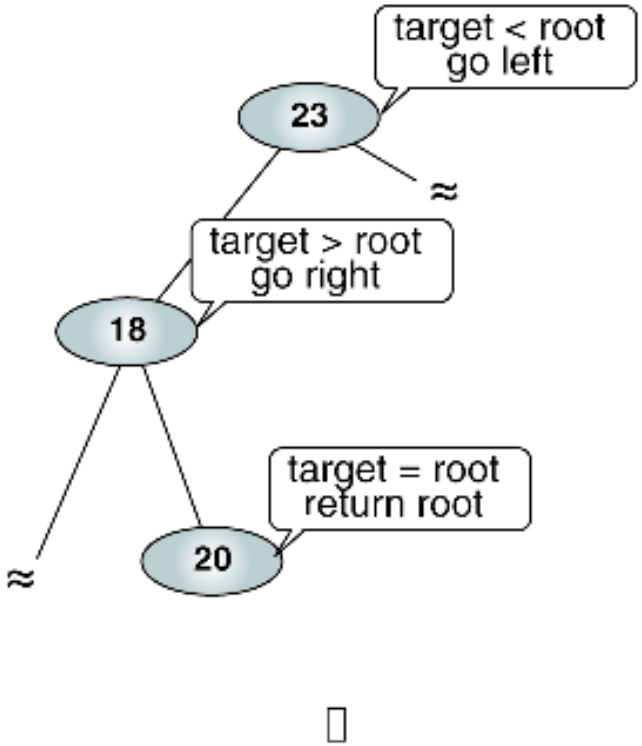



```
Algorithm searchBST (root, targetKey)
Search a binary search tree for a given value.
   Pre    root is the root to a binary tree or subtree
          targetKey is the key value requested
   Return the node address if the value is found
          null if the node is not in the tree
1  if (empty tree)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

Target : 20

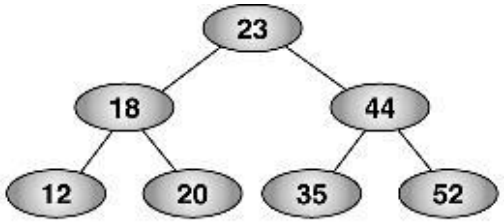
Searching a Given Node in a BST



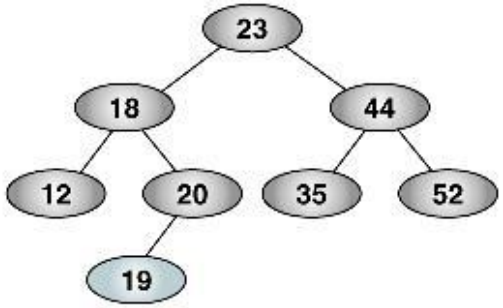


Insertion

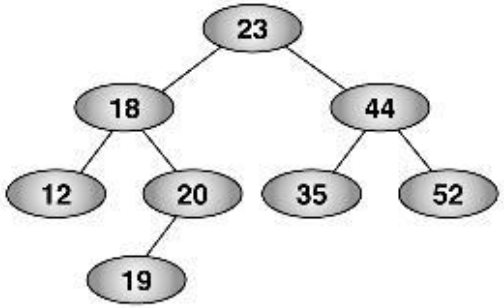
- แทรกข้อมูลเข้าไปใน BST โดย
  - ตามกิ่ง (Branch) ไปเรื่อยๆ จนเจอ subtree ที่ว่าง
  - แล้วแทรกโหนดข้อมูลใหม่เข้าไป (แทรกจาก leaf node)



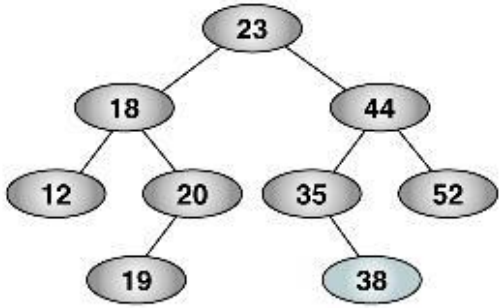
(a) Before inserting 19



(b) After inserting 19



(c) Before inserting 38

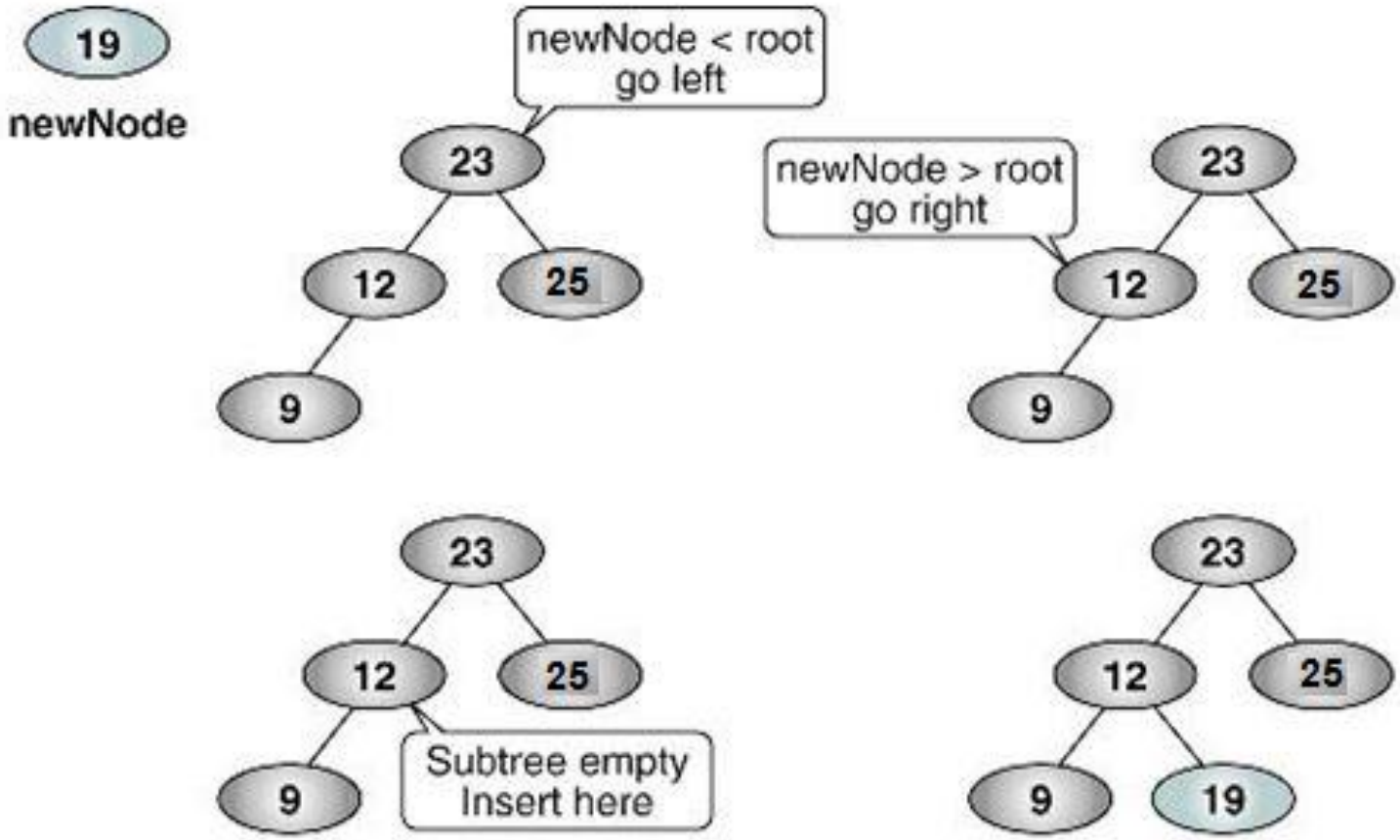


(d) After inserting 38

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.)



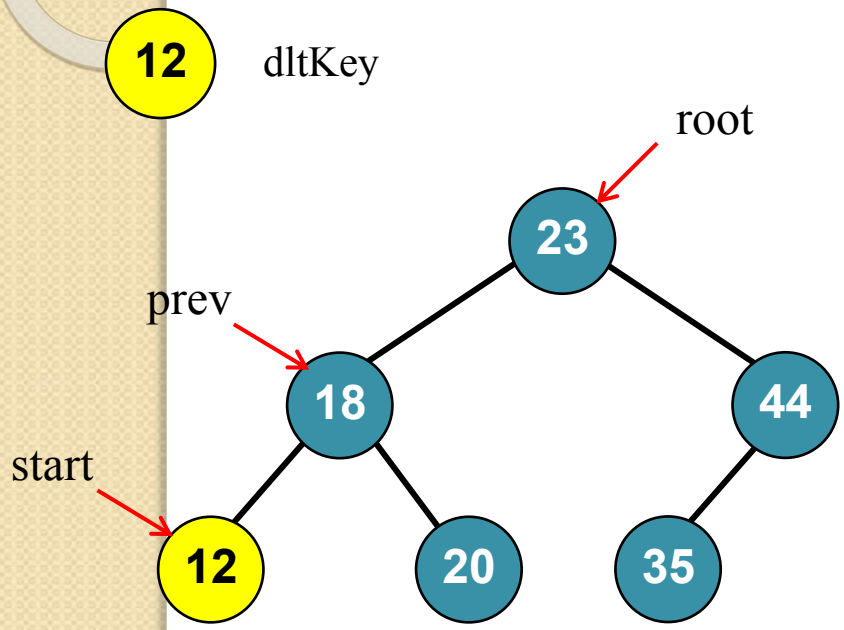
Deletion

- 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 แล้วลบโหนดมากสุดของ Left subtree แทน

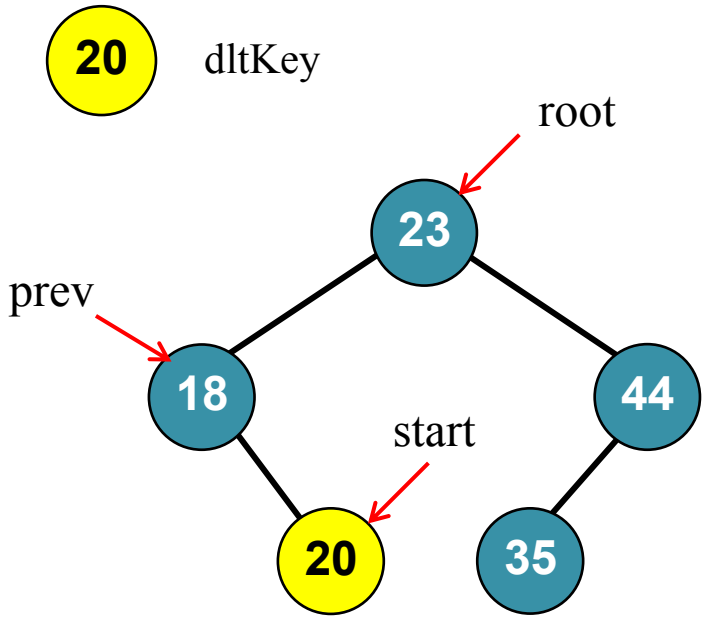
Deletion (cont.)

```
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
```

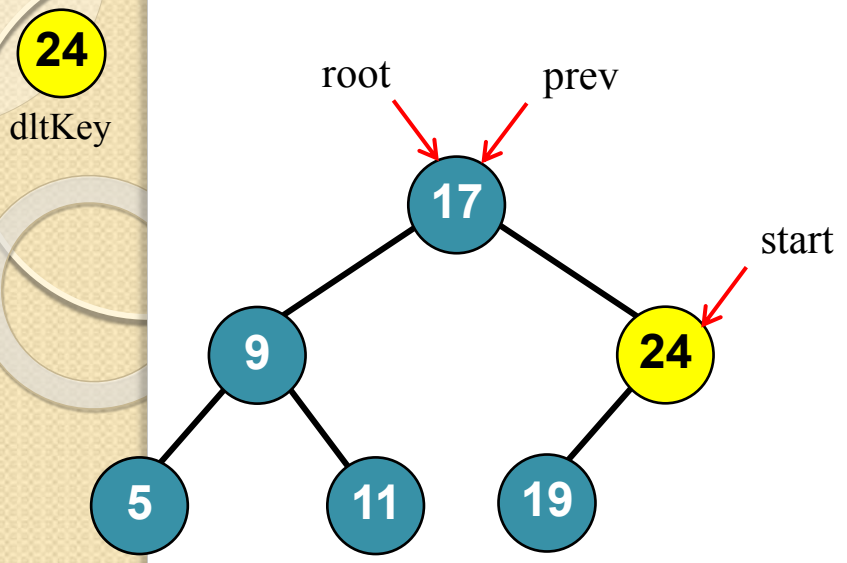
Case 1 : The node has no children



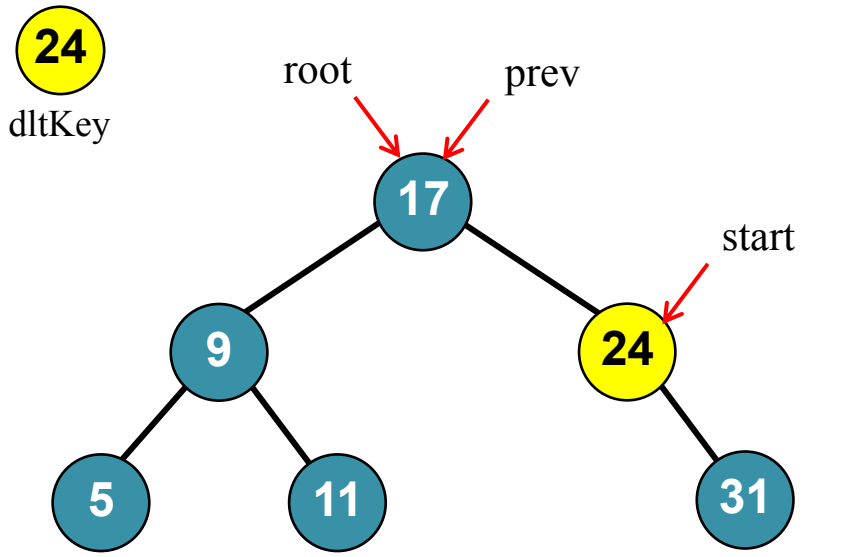
prev.left = NULL;



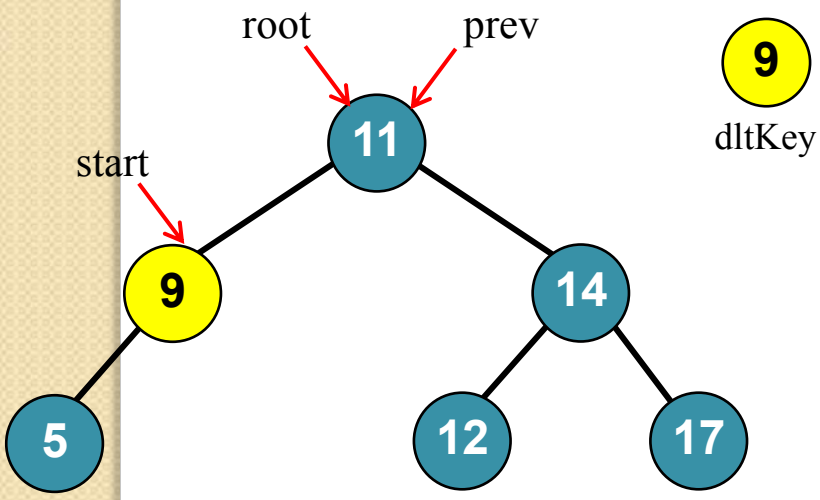
prev.right = NULL;



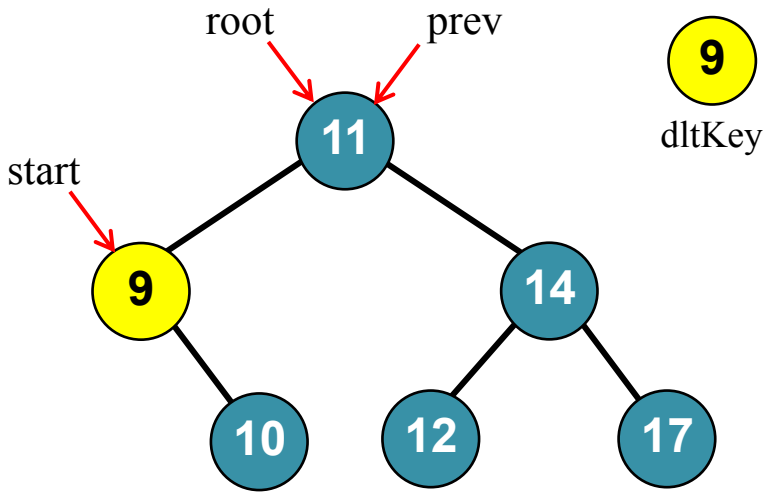
prev.right = start.left



prev.right = start.right

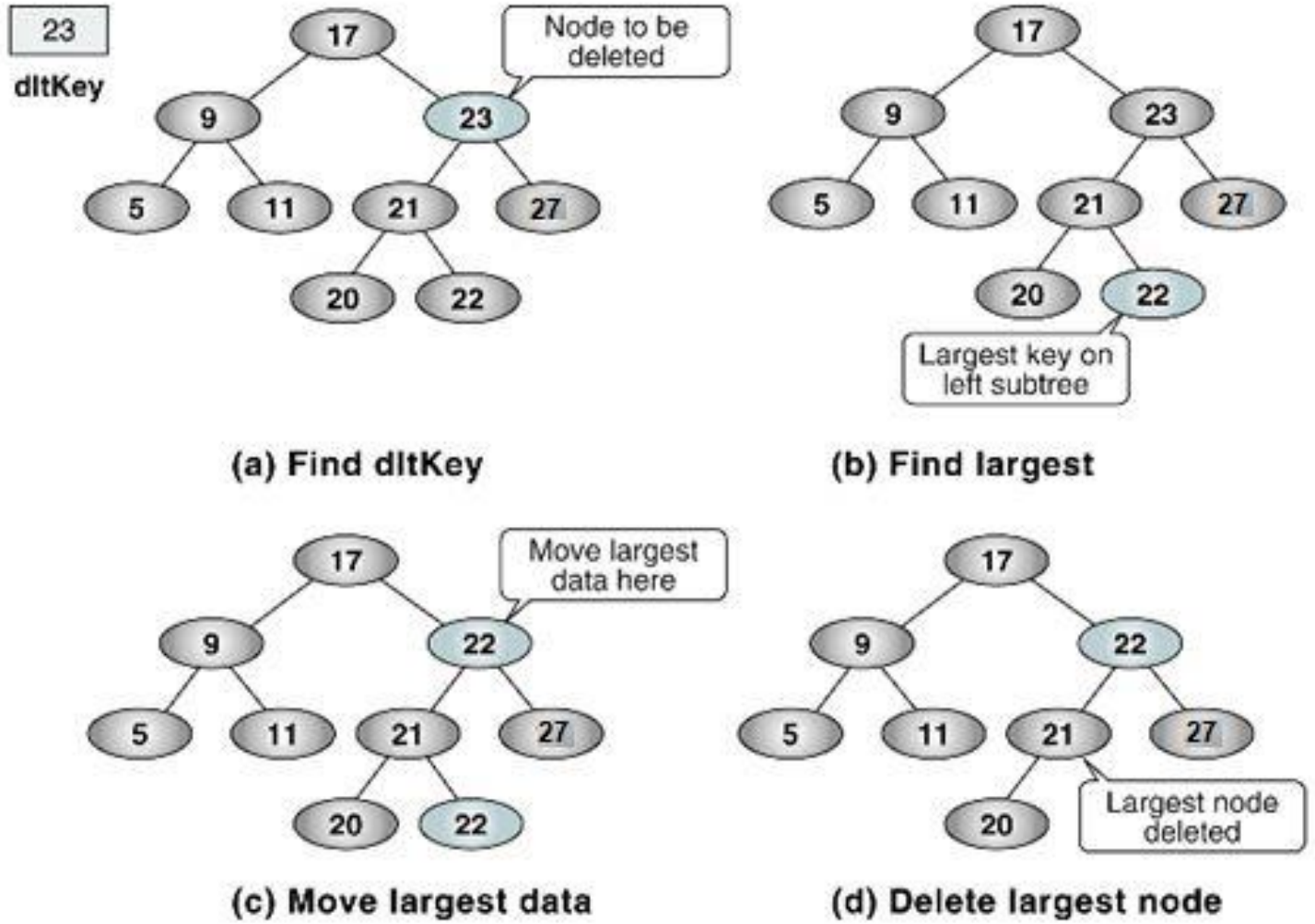


prev.left = start.left



prev.left = start.right

Case 4 : The node have 2 subtrees



Binary Expression Tree

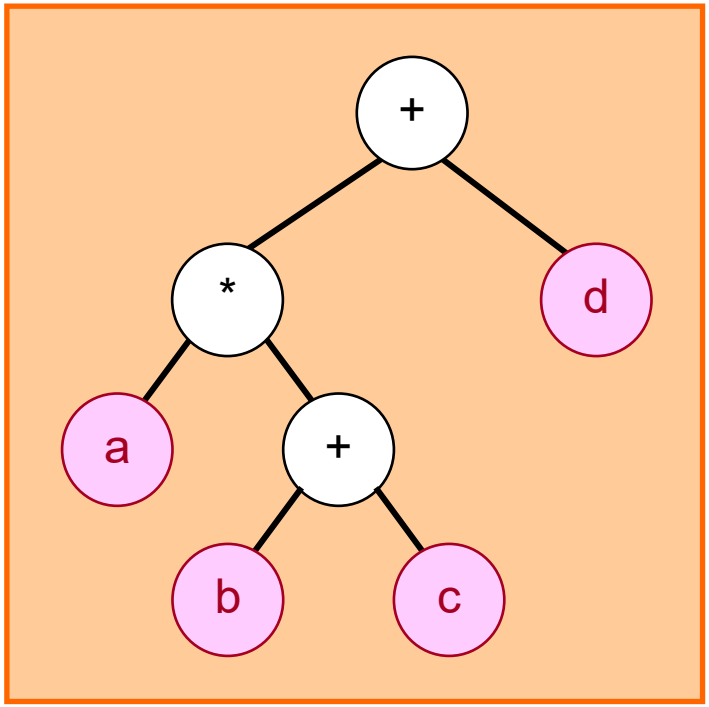
Binary Expression Tree

- Expression คือ นิพจน์การคำนวณซึ่งประกอบด้วย
  - Operator คือ ตัวดำเนินการ เช่น +, -, *, /
  - Operand คือ ตัวถูกดำเนินการ

  Ex. 52 + 10 --> Operand : 52, 10 Operator : +
- Expression Tree : เป็น Binary tree ที่ใช้สำหรับจัดเก็บ Expression โดยมีลักษณะดังนี้
  - Leaf Node จะเป็น Operand เสมอ
  - Root และ Internal Node จะเป็น Operator
  - Subtree ที่มี root เป็น Operator จะเป็น subexpression

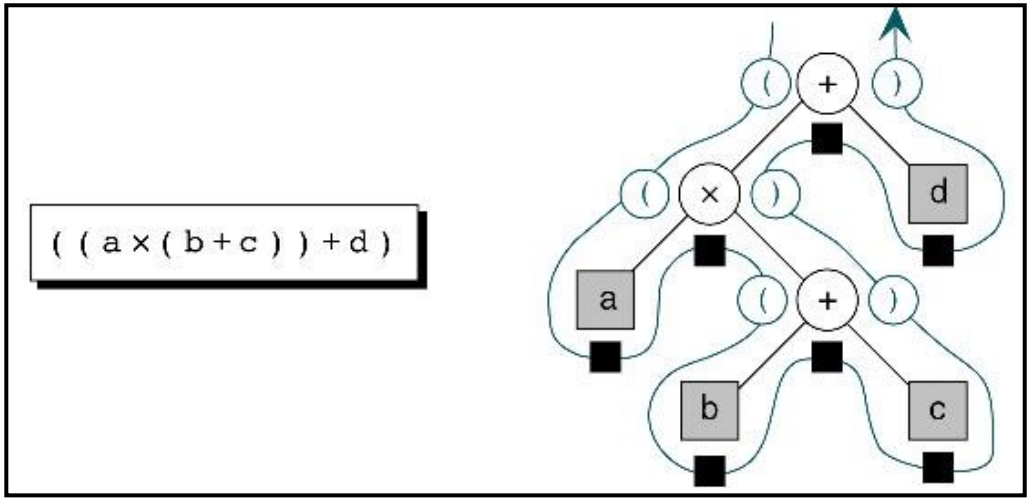
Expression Tree Example

- a*(b+c)+d

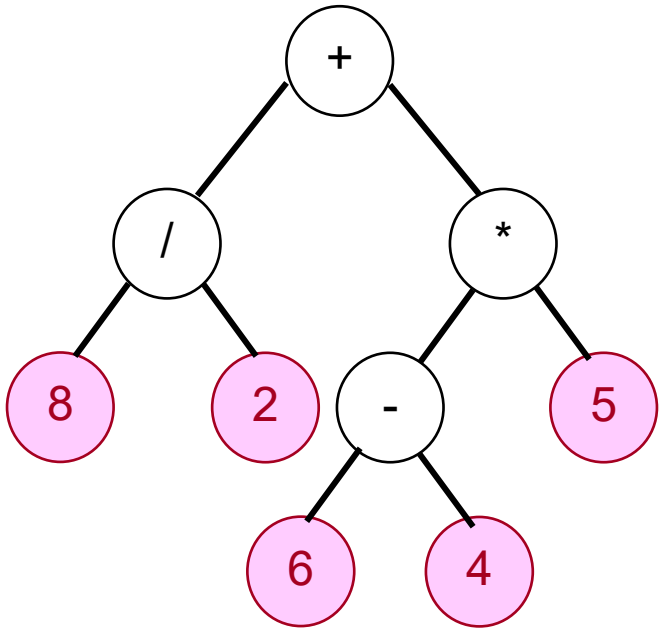


Expression Tree : Traversal

- ในกรณีของการท่องแบบ Infix จะต้องมีการเพิ่มวงเล็บเปิดและปิดเมื่อเริ่มต้นและจบเอ็กซ์เพรสชั่น ตามลำดับ
  - เปิดวงเล็บเมื่อเริ่มต้น tree หรือ subtree
  - ปิดวงเล็บเมื่อทำงาน tree หรือ subtree นั้นเสร็จ
- ในกรณีการท่องแบบ Prefix และ Postfix ไม่จำเป็นต้องมีวงเล็บเปิดปิด



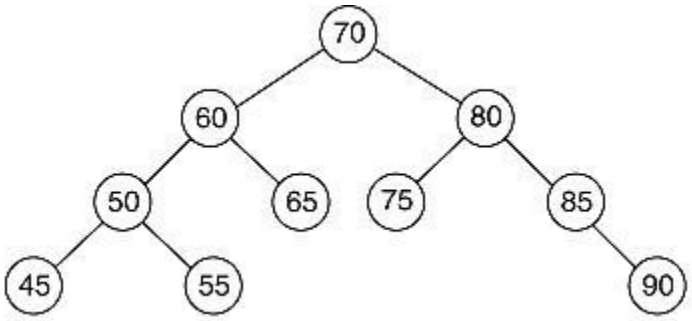
Example : Traversal



- Preorder Traversal : +/82*-645
- Inorder Traversal : ((8/2)+((6-4)*5))
- Postorder Traversal : 82/64-5*+

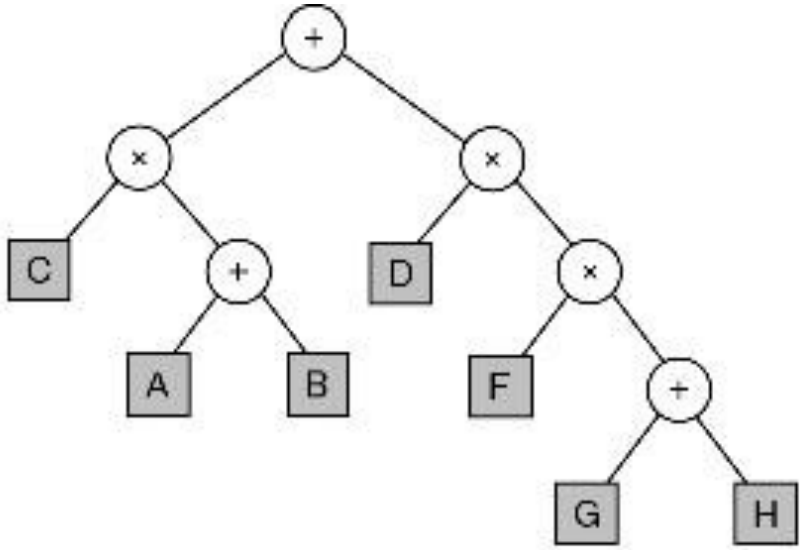
Quiz

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



Quiz

- จาก Expression Tree ให้หา infix, prefix, postfix expression



- ให้เขียน Expression Tree, Infix, Postfix สำหรับ Prefix expression ดังนี้

  + - * A + B C / D E * F G