OOP & data struct

13. Binary Search Tree(BST) & sorting

BY SOMSIN THONGKRAIRAT


BINARY TREE
BINARY SEARCH TREE
imgflip.com

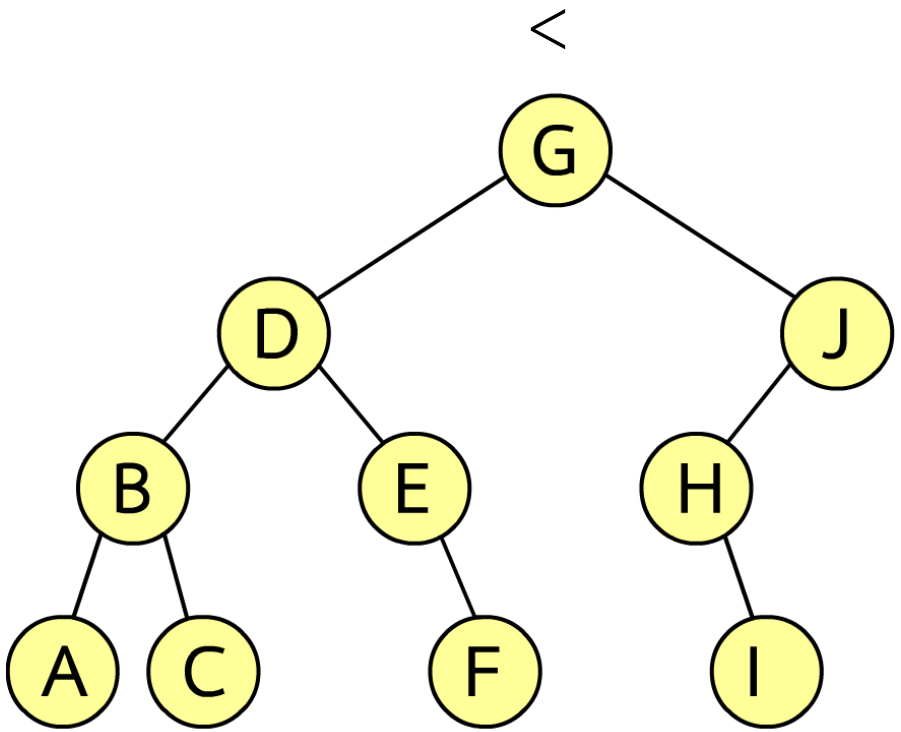
What is Binary search tree

A conditioned binary tree

Binary tree ที่มีเงื่อนไข

Fast lookup and fast remove

สามารถหาสิ่งของและลบได้เร็ว



* Invented in 1960s

Rules

For every node

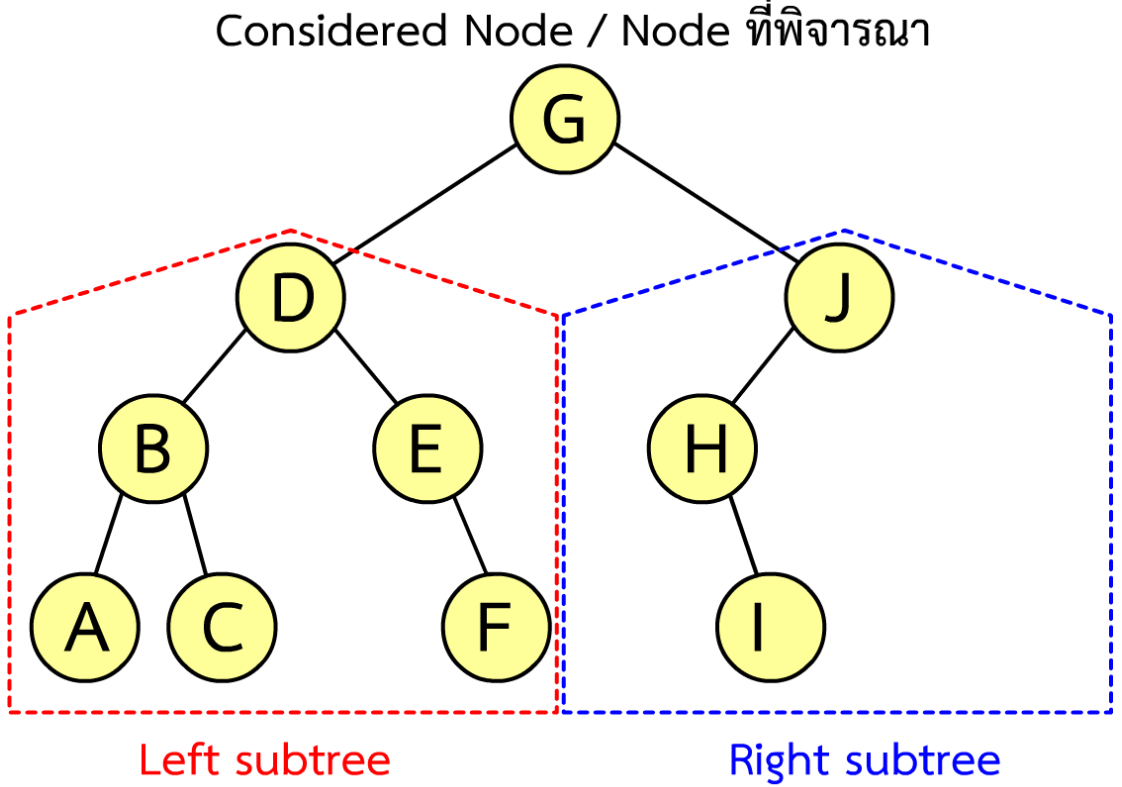
Left subtree contain only less value items

Right subtree contain only greater value items

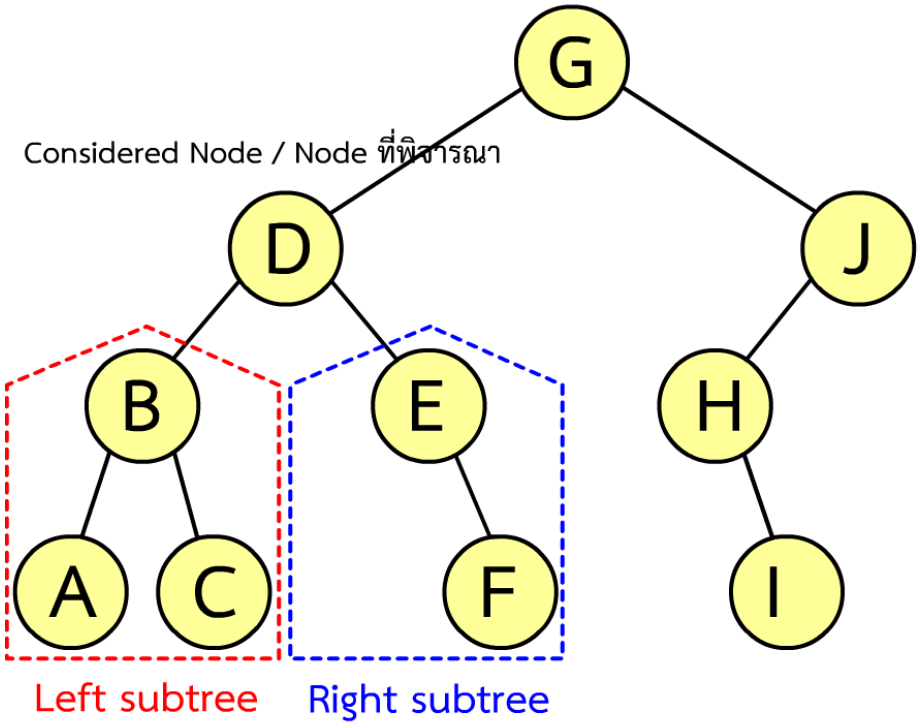
สำหรับทุกๆ node

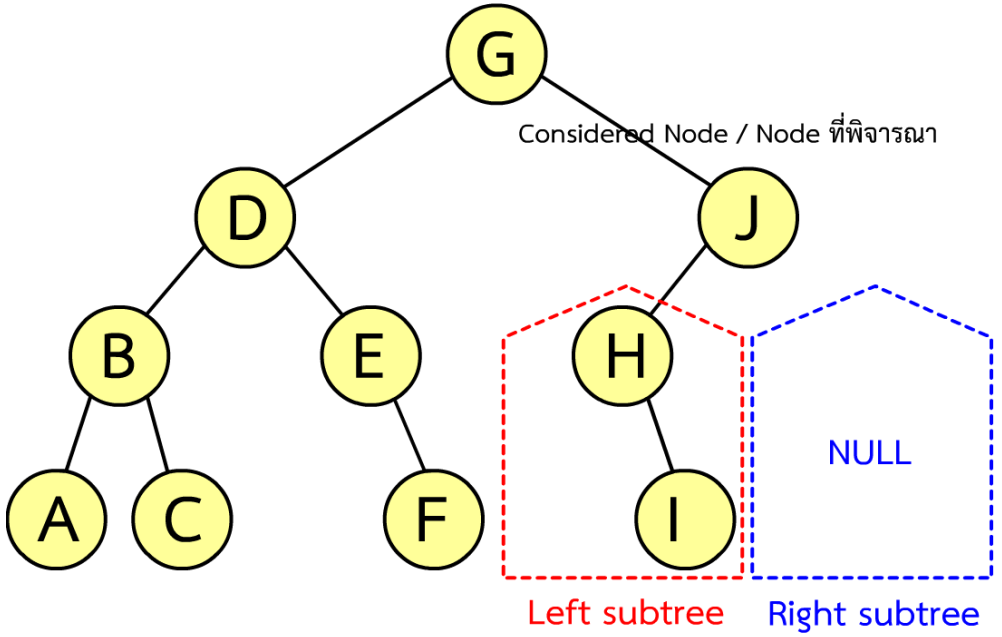
Left subtree จะมีเพียงแค่สมาชิกที่มีค่าต่ำกว่า

Right subtree จะมีเพียงแค่สมาชิกที่มีคู่งกว่า

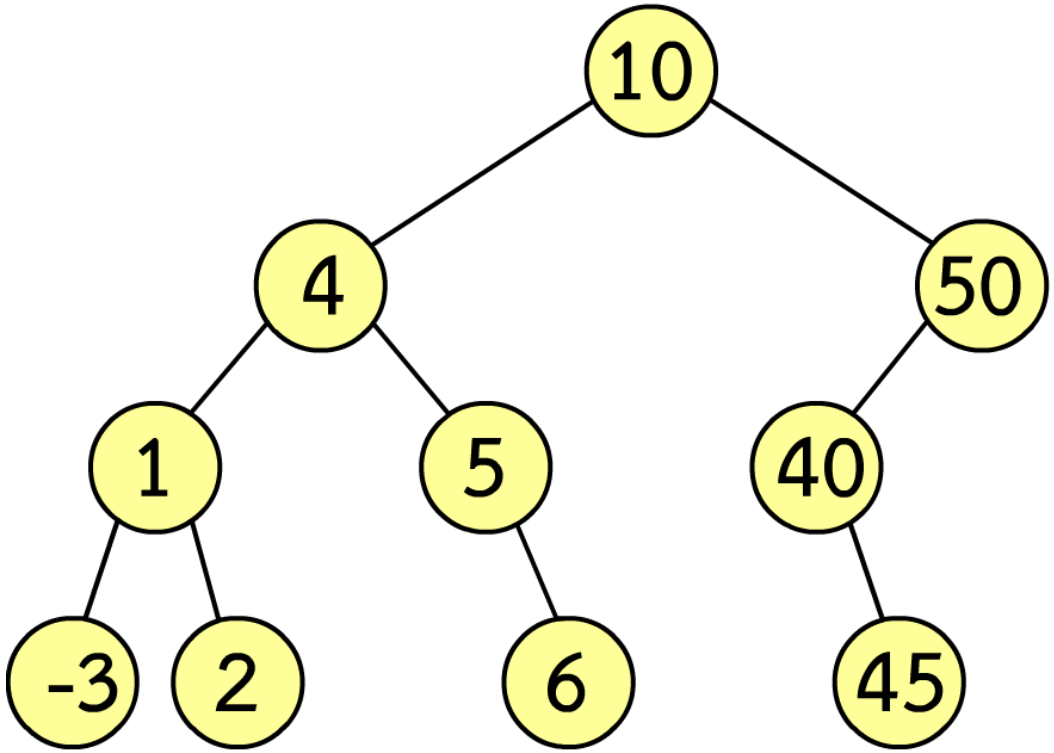


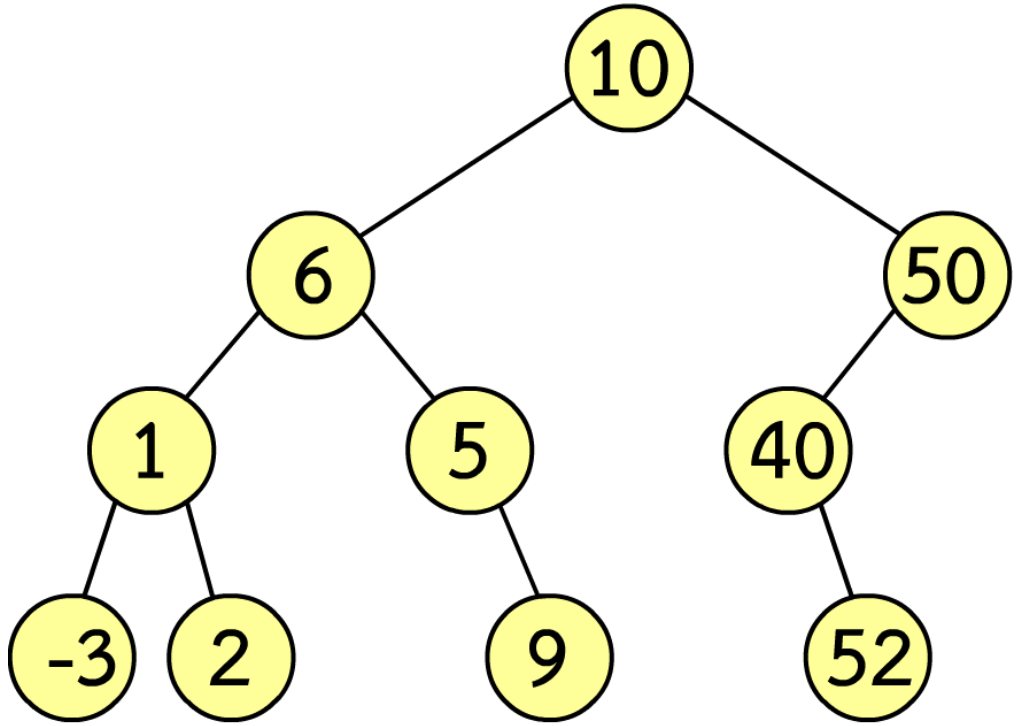
All subtree


Considered Node / Node ที่พิจารณา
G
D
J
B
E
H
A
C
F
I
Left subtree
Right subtree

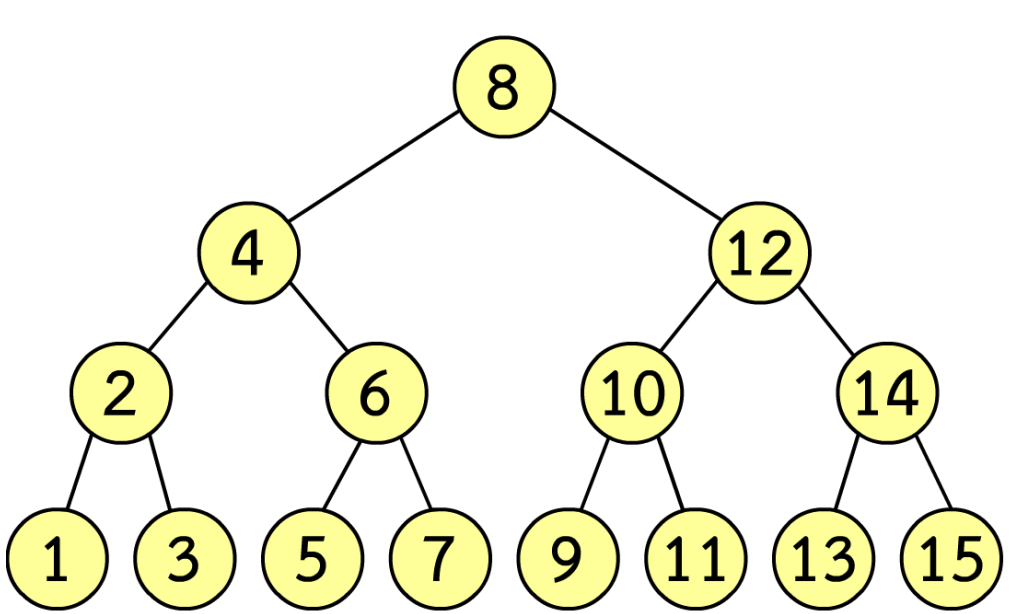

G
Considered Node / Node ที่พิจารณา
D
J
B
E
H
A
C
F
I
NULL
Left subtree
Right subtree

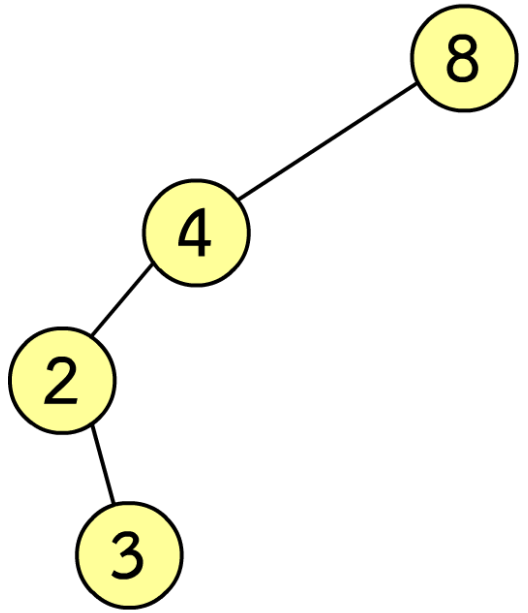
Witch one is follow BST rule





No need to be balance tree





Build the tree / insert(item)

```
Node *root = new Node('B');
root->left  = new Node('C');
root->right = new Node('D');
(root->left) ->left  = new Node('L');
(root->left) ->right = new Node('M');
```

- build with rule unlike unconditioned binary tree

- build by insert all node to tree start at root

- A new node is always inserted at the leaf of the tree by maintain BST rule for all subtree

- insert() function

Build the tree / insert(item)

```
Node *root = new Node('B');
root->left  = new Node('C');
root->right = new Node('D');
(root->left) ->left  = new Node('L');
(root->left) ->right = new Node('M');
```

- มีกฎในการสร้าง (insert) ไม่เหมือนกับ unconditioned binary tree
- สร้างโดยการ insert node ทุก node ไปใน tree เริ่มต้นที่ root
- node ใหม่ที่ถูก insert เข้ามาจะถูกนำไปไว้ที่ left เสมอ โดยที่จะยังคงกฎของ BST ไว้เสมอ
- insert() function

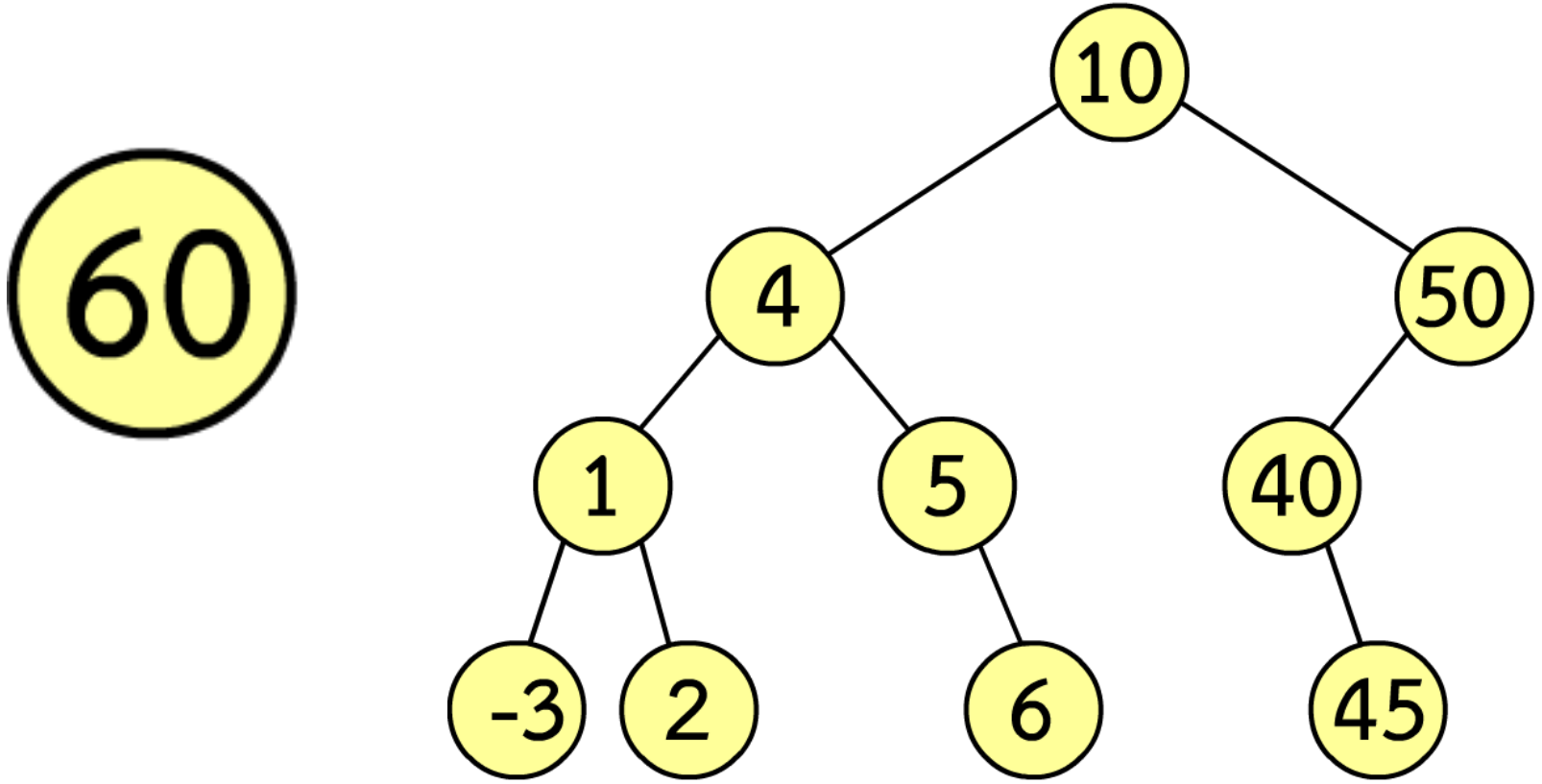
insert(item)

- start insert at root node
- if node is null build new node at this location
- if new item value less then node value go to left child
- if new item value greater than node value go to right child
- repeat until found null node

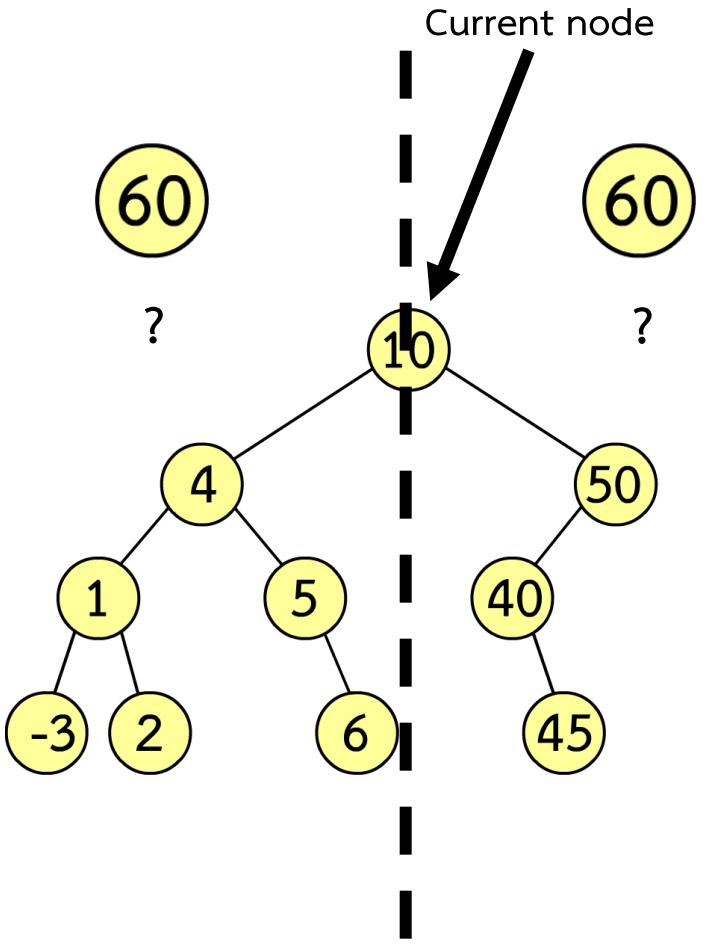
insert(item)

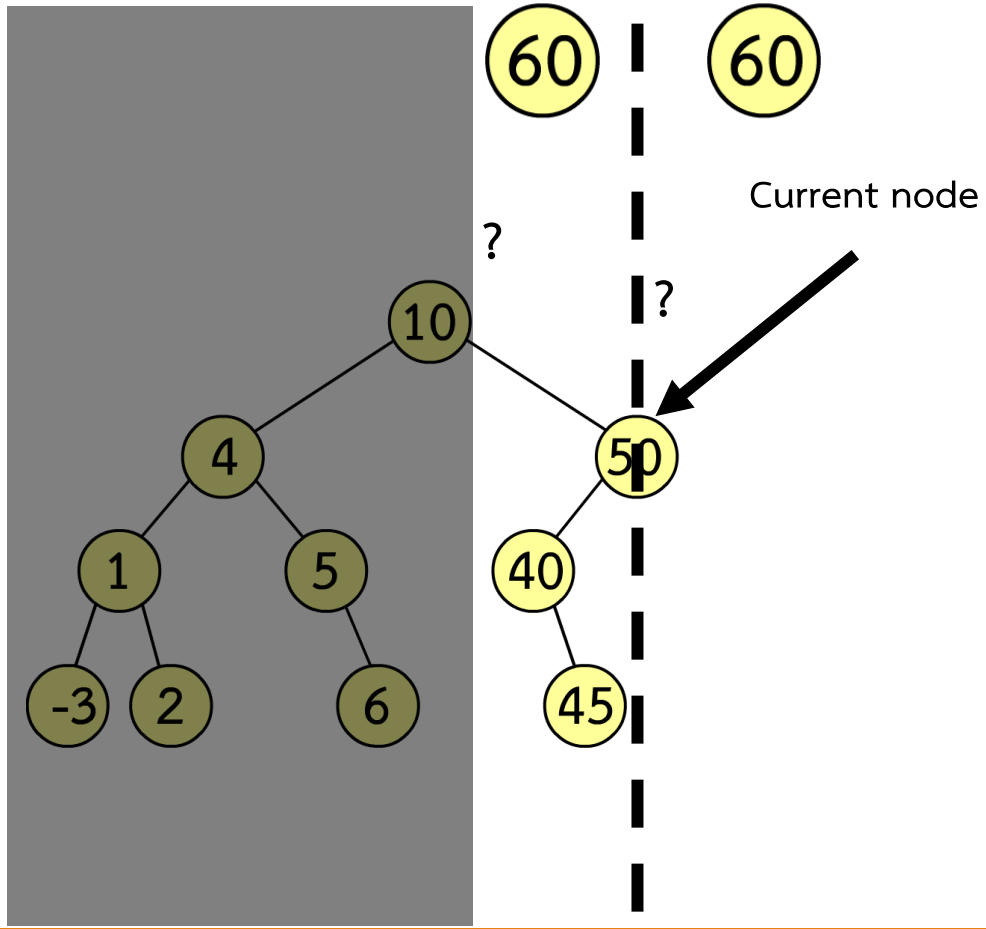
- เริ่ม insert จาก root
- หาก node ดังกล่าวเป็น null ให้สร้าง node ใหม่ด้วยข้อมูล item
- หาก item มีค่าน้อยกว่า node ดังกล่าว ให้ ท่องไป ทางซ้าย
- หาก item มีค่ามากกว่า node ดังกล่าว ให้ ท่องไป ทางขวา
- ซ้ำเรื่อย ๆ จนกว่าจะพบ null

Insert(60)

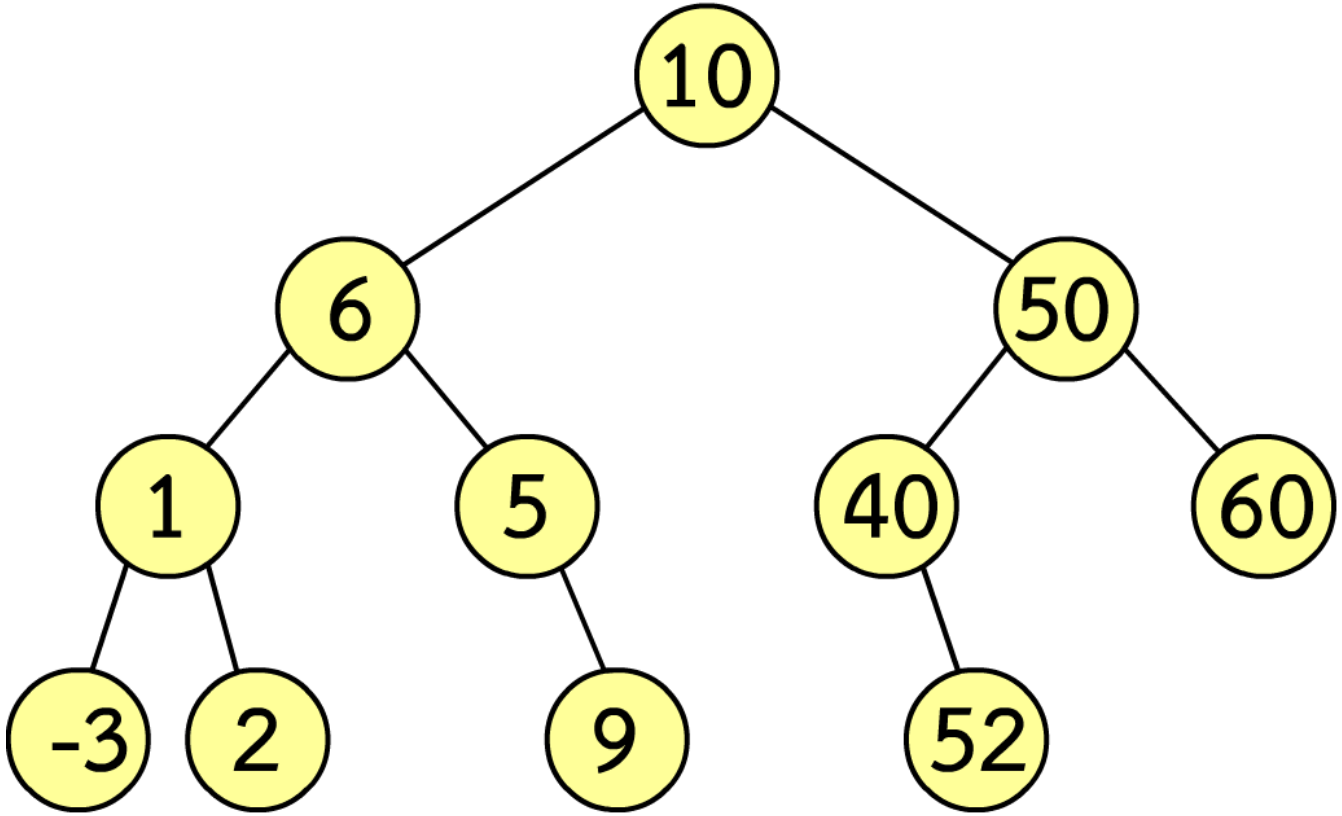


Insert(60) which side?

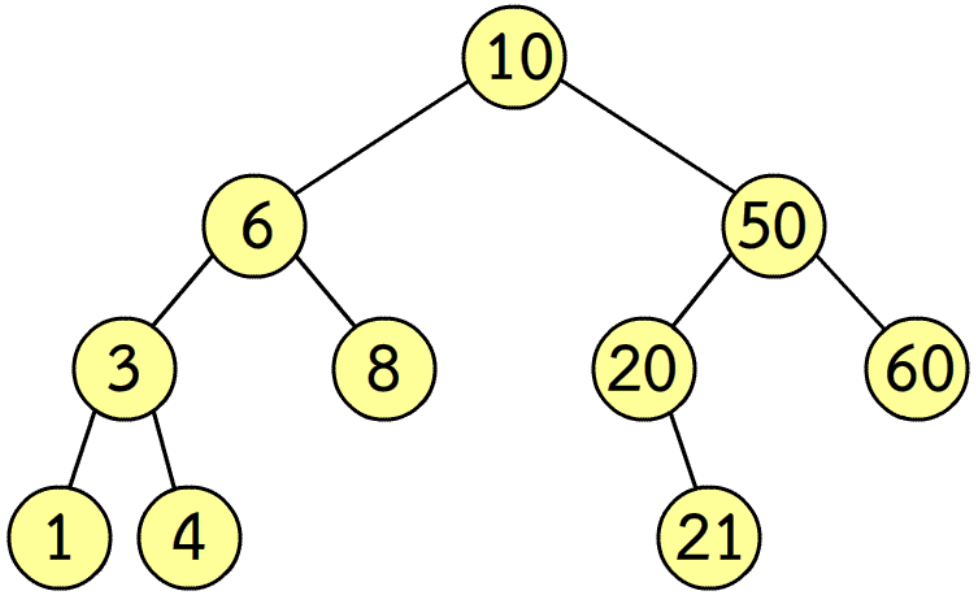




result

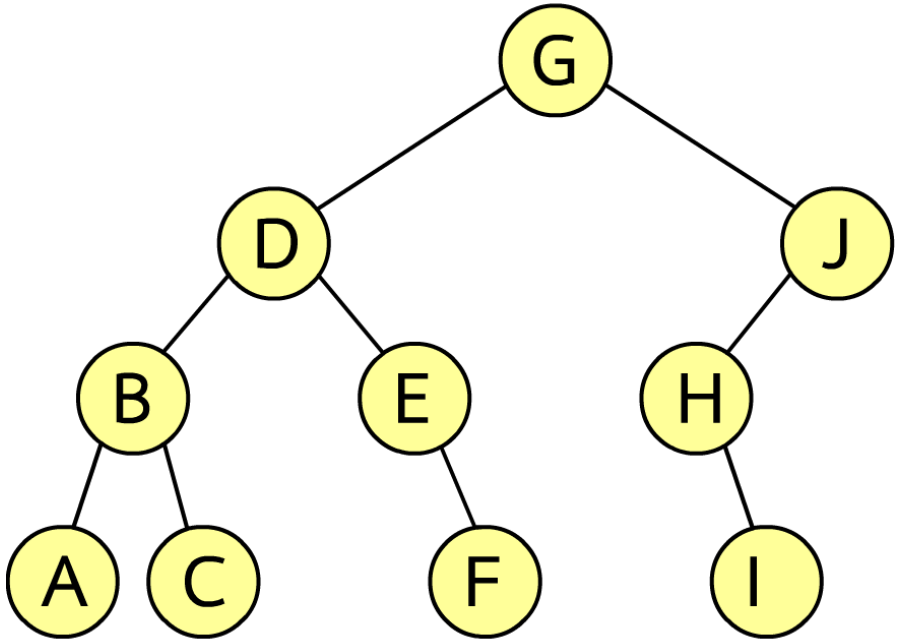


Another example



code

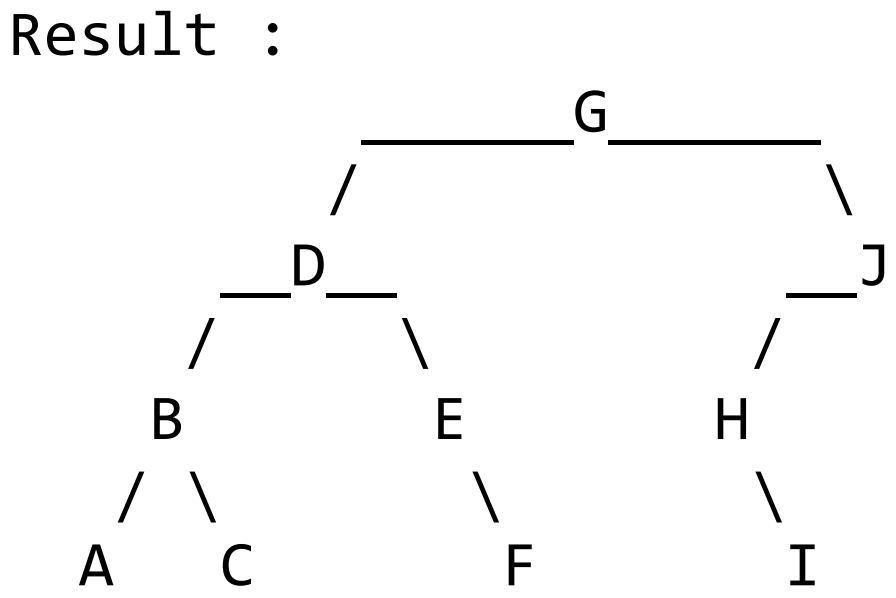
```
Node* p = root;
while(true){
    if(item < p->data){ // go left
        if(p->left == nullptr){
            p->left = new Node(item);
            break;
        }
        else p = p->left;
    }
    else{
        if(p->right == nullptr){
            p->right = new Node(item);
            break;
        }
        else p = p->right;
    }
}
```



code

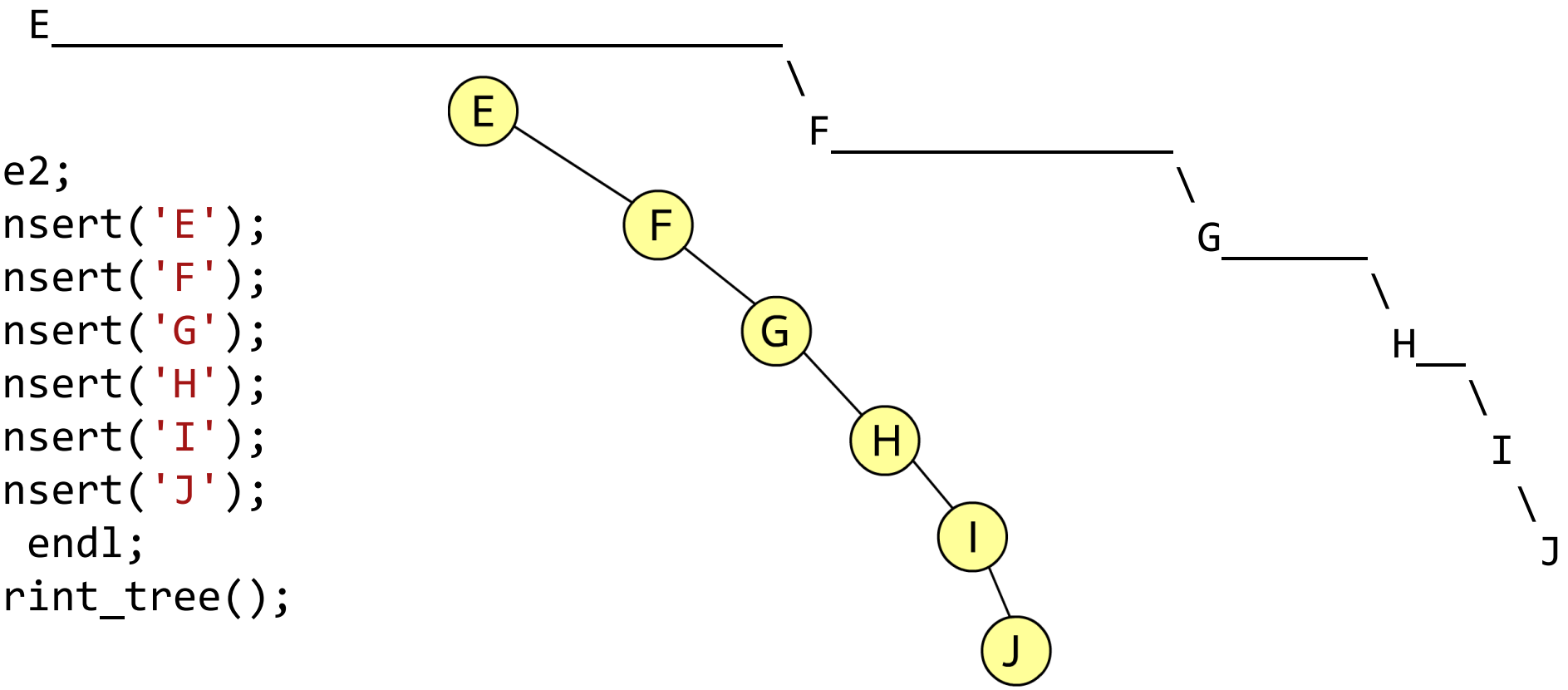
```
BST tree;
tree.insert('G');
tree.insert('D');
tree.insert('J');
tree.insert('B');
tree.insert('E');
tree.insert('A');
tree.insert('C');
tree.insert('F');
tree.insert('H');
tree.insert('I');
tree.print_tree();
```

Result :



Another example

```
BST tree2;
tree2.insert('E');
tree2.insert('F');
tree2.insert('G');
tree2.insert('H');
tree2.insert('I');
tree2.insert('J');
cout << endl;
tree2.print_tree();
```



Quiz

ASYMPTOTIC NOTATION

Asymptotic notation

Big O ->

Big theta ->

Big Omega ->

Search(item)

- one of purpose to invented BST / หนึ่งในจุดประสงค์ที่ BST ถูกสร้างขึ้นมา

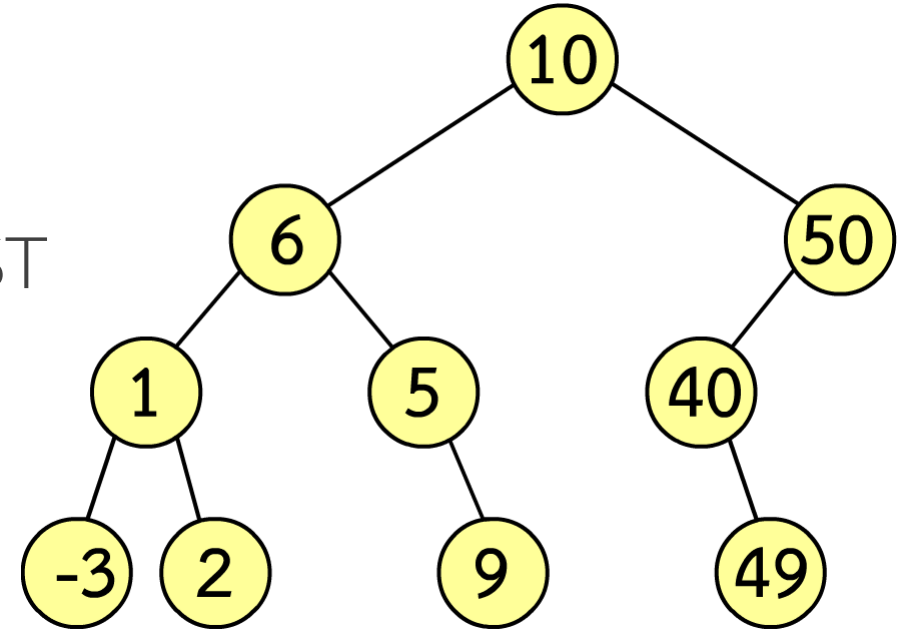
- find node that contain search value

- หา node ที่มีค่าเท่ากับ search value

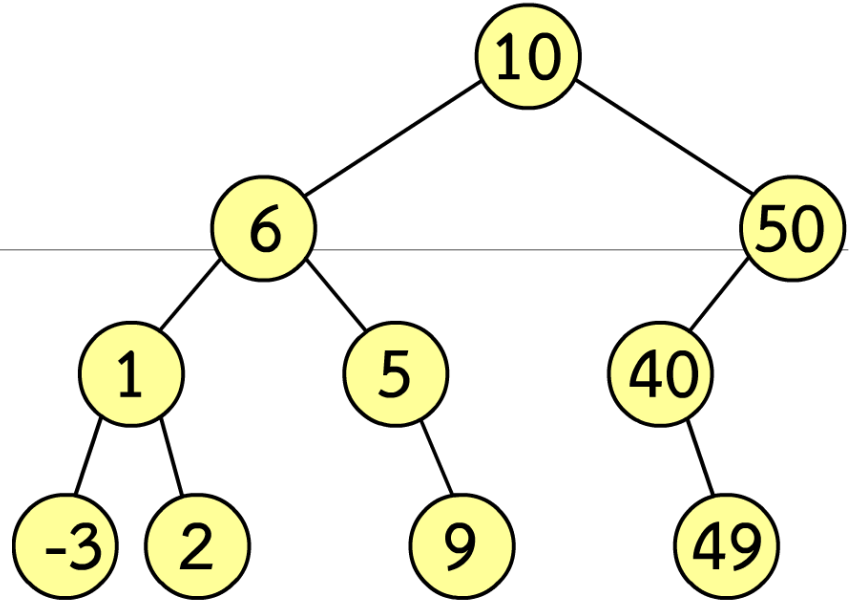
Search(item)

- because whole tree maintain BST rules so we can use this rule for searching

- เนื่องจากทั้ง tree นี้เป็นไปตามกฎของ BST เราจึงสามารถใช้กฎของ BST ในการ search ได้เ

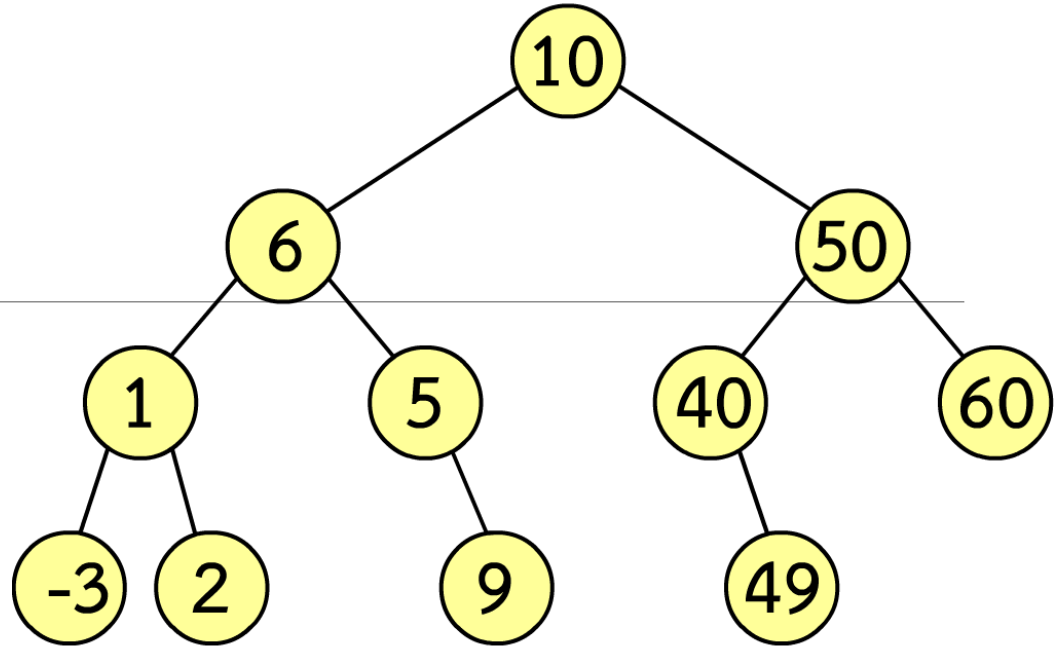


Search(item)



- start from root node
- check node that equal to search key
- if found return -> mission completed
- if search key less than node go to left node
- if search key greater than node go to right node
- repeat until found null and return not found

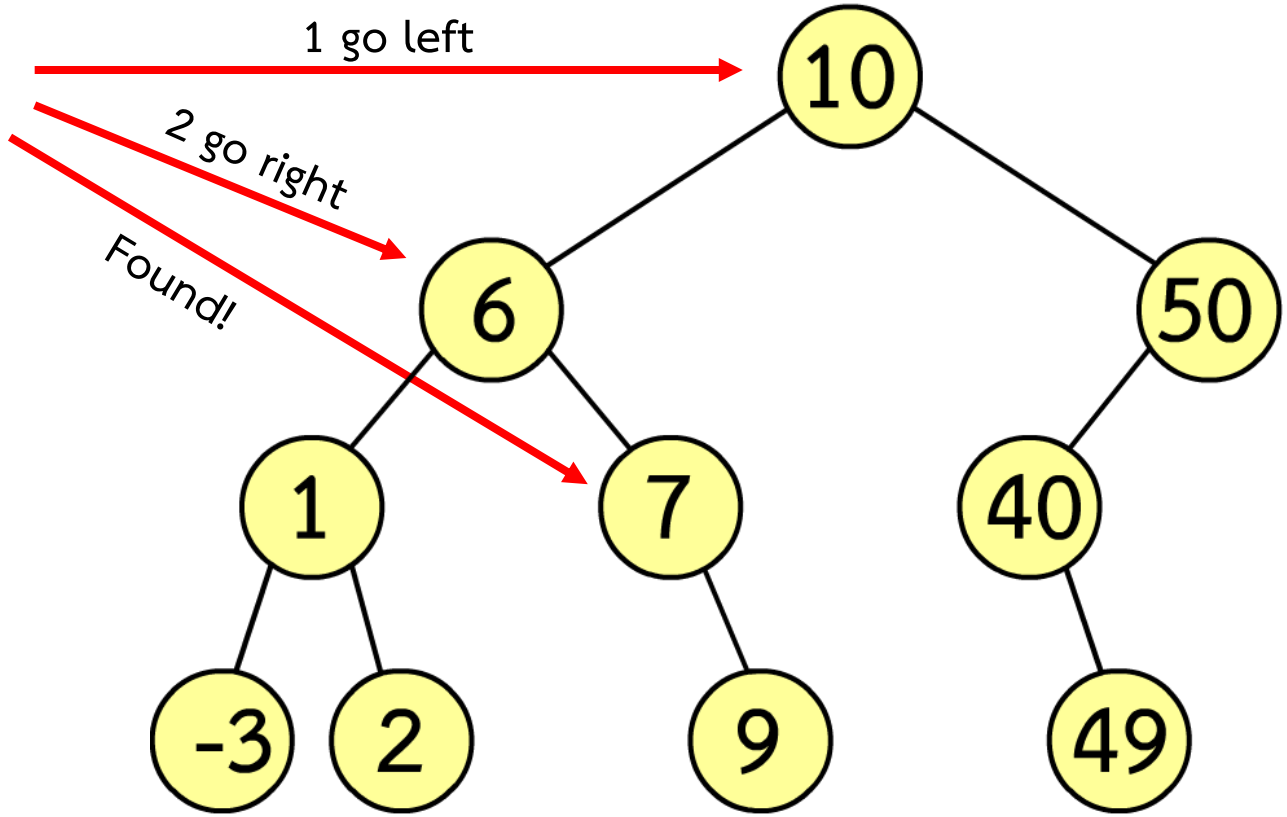
Search(item)



- เริ่มต้นจาก root node
- เช็ค node ว่าตรงตามที่ต้องการหรือไม่
- หากพบ return -> mission completed
- หาก search key มีค่าน้อยกว่า node ไปทำ left node
- หาก search key ค่ามากกว่า node ไปทำ right node
- ทำไปเรื่อย ๆ จนกว่าจะพบ NULL ให้ return Not found

Search(7)

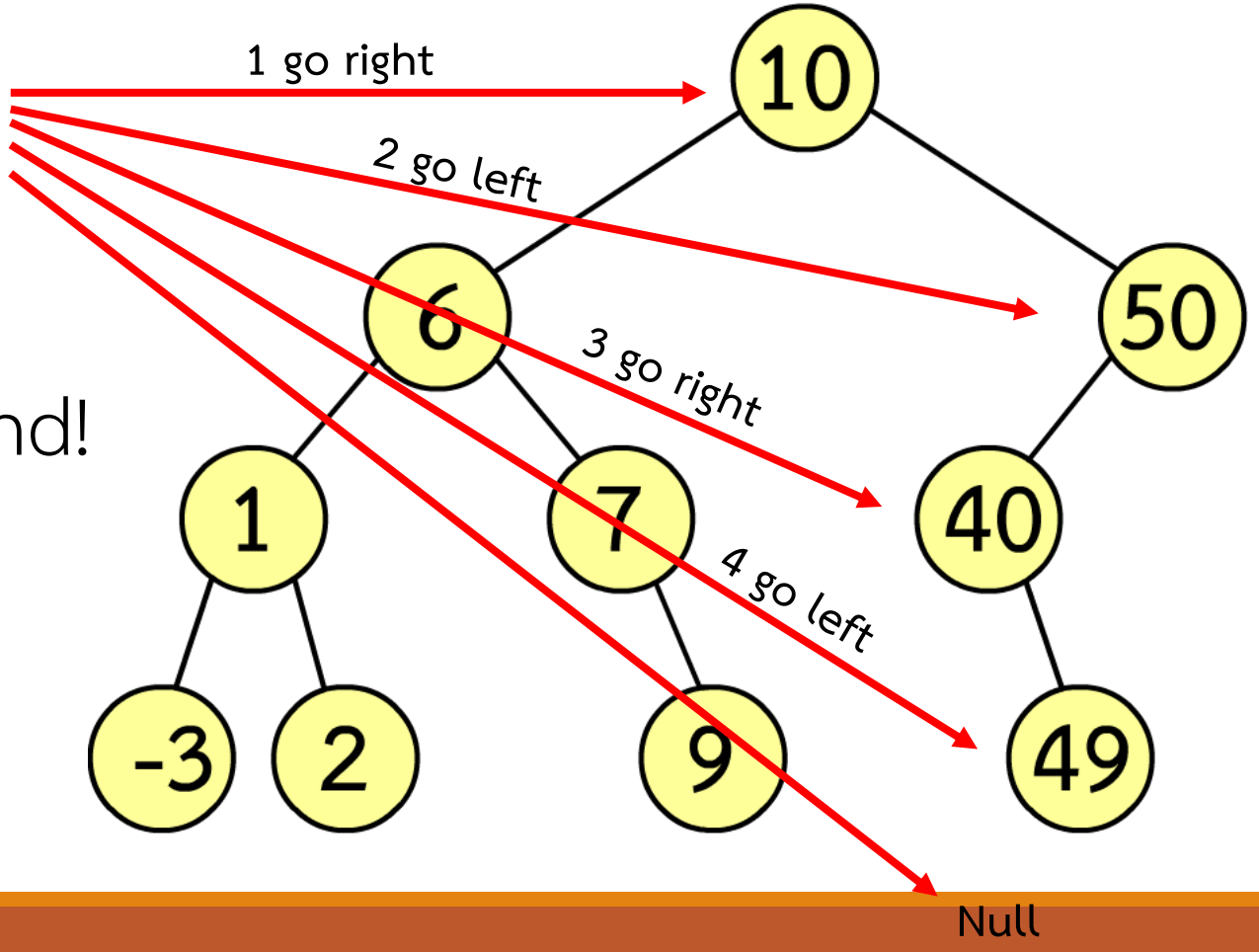
Search (7)



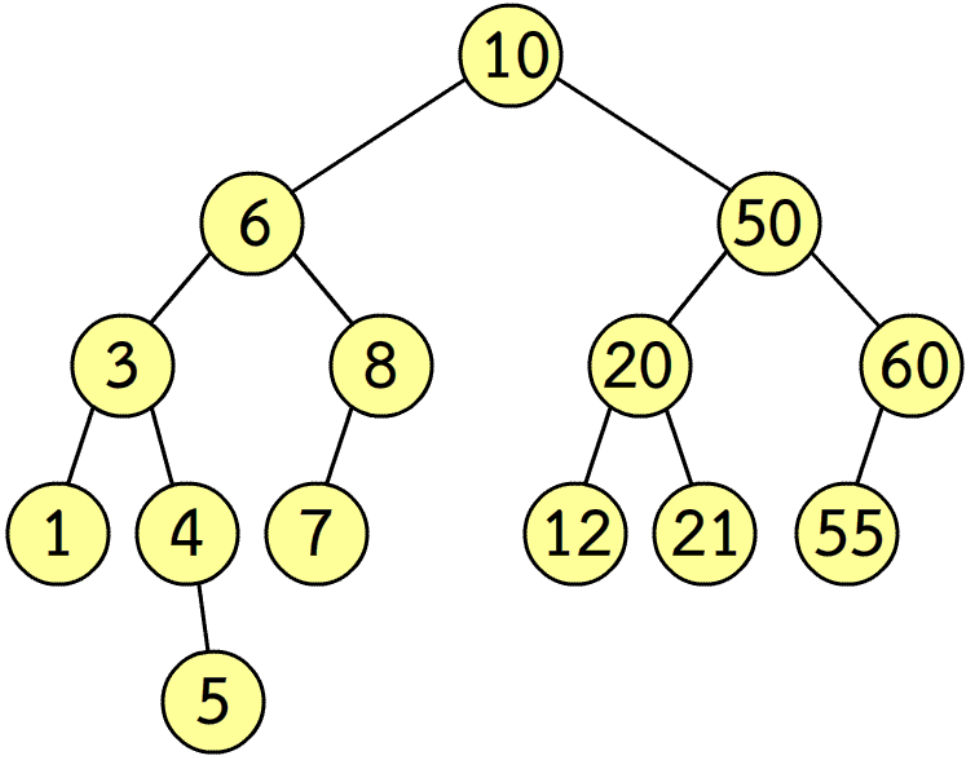
Search(41)

Search (41)

Return Not found!

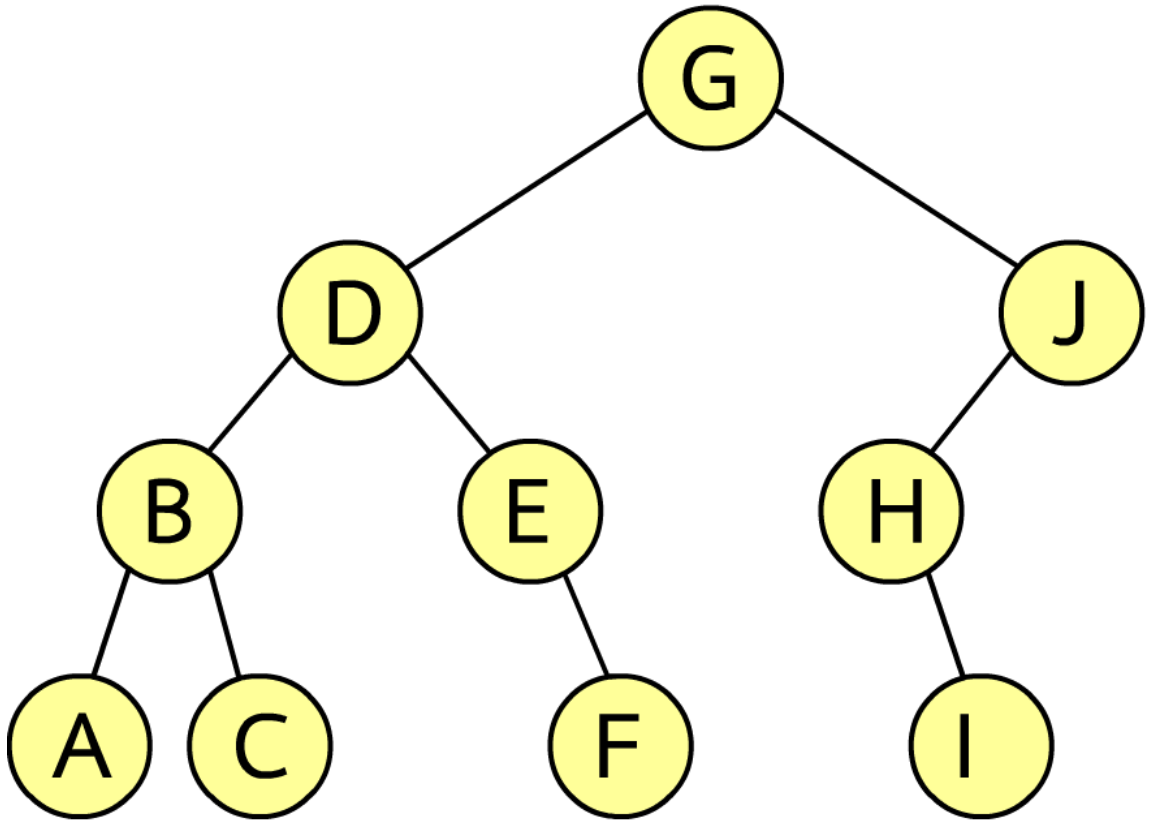


Another example



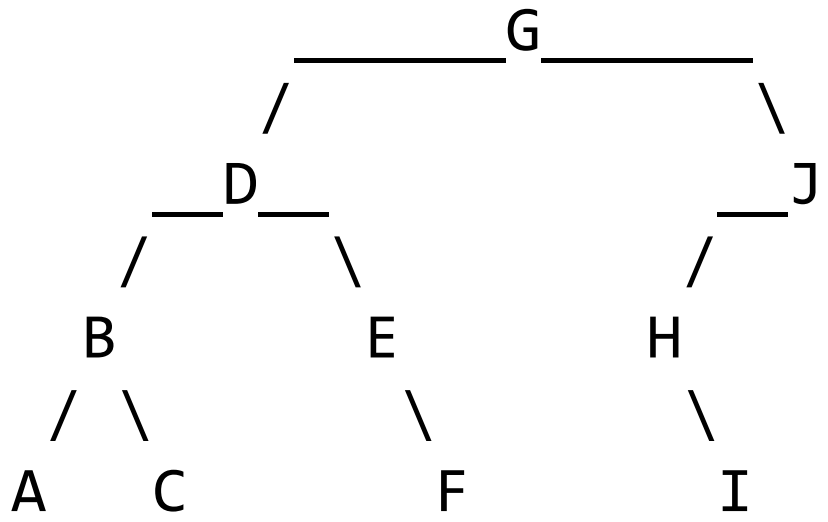
Code

```
bool search(char item){
    Node *p = root;
    while(p != nullptr){
        if(item == p->data){
            return true;
        }
        else if(item < p->data){
            p=p->left;
        }
        else{
            p=p->right;
        }
    }
    return false;
}
```

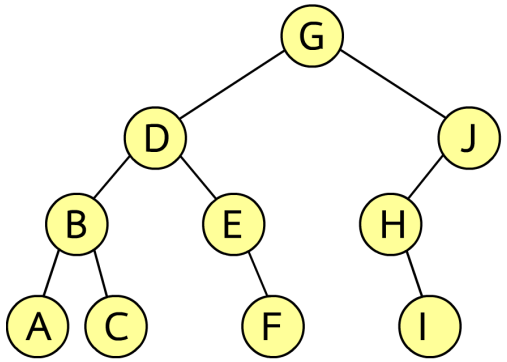


```
BST tree;
tree.insert('G');
tree.insert('D');
tree.insert('J');
tree.insert('B');
tree.insert('E');
tree.insert('A');
tree.insert('C');
tree.insert('F');
tree.insert('H');
tree.insert('I');
cout << endl;
tree.print_tree();

cout << tree.search('C') << endl;
cout << tree.search('X') << endl;
cout << tree.search('K') << endl;
cout << tree.search('H') << endl;
```

Result :




```
1
0
0
1
```



Search asymptotic notation ?

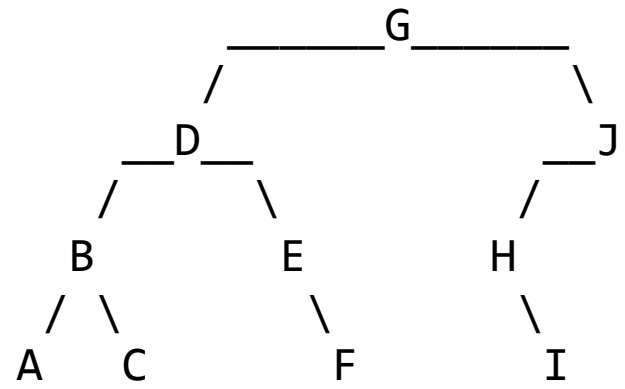
Big O

Big Theta

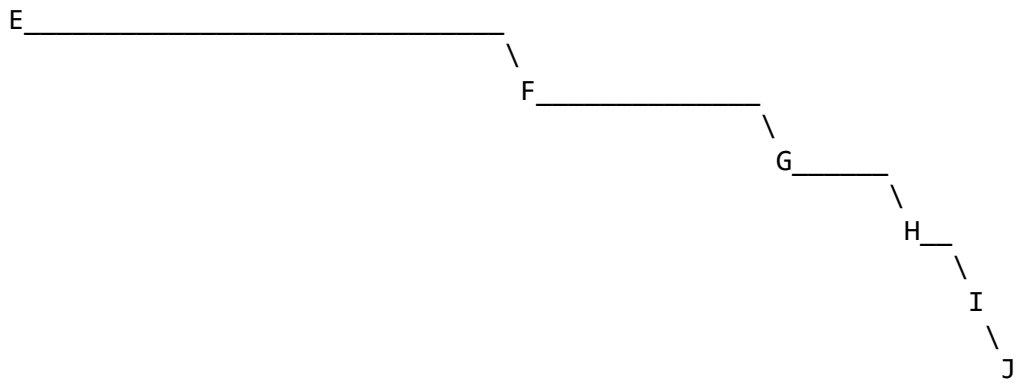
Big Omega

Traversal (in order) ?

```
tree.print_inorder();
cout << endl;
tree2.print_inorder();
cout << endl;
```



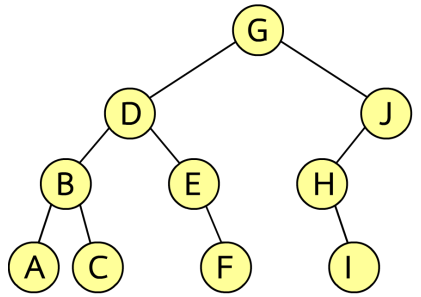

```
Result :
A B C D E F G H I J
E F G H I J
```

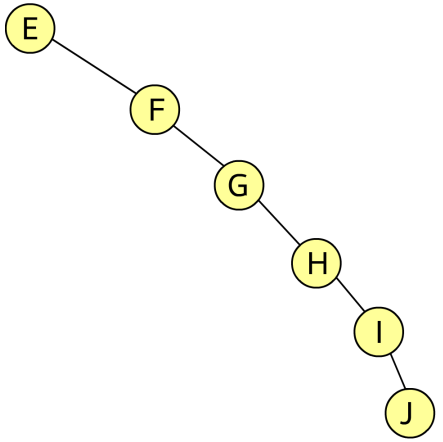


Traversal inorder

- Traversal inorder in BST always given sorted order item !
- หากท่อง tree แบบ inorder ใน BST จะได้ลำดับที่เรียงแล้วเสมอ

```
Result :
A B C D E F G H I J
E F G H I J
```





Delete! (hardest part)

Delete! (hardest part)

- after delete node, tree must remain BST rule

- หลังจากการ delete node ,tree จะต้องคงความเป็น BST ไว้

HOW !!?

Delete hardest part ()

- 3 case will happed

1. node to be deleted is the leaf node -> just delete node

2. node to be deleted has only one child -> replace node with child

3. Node to be deleted has 2 children -> replace successor(next) inorder member and repeat process to successor(next) inorder member

Delete hardest part ()

- จะมี 3 กรณีที่เกิดขึ้นได้

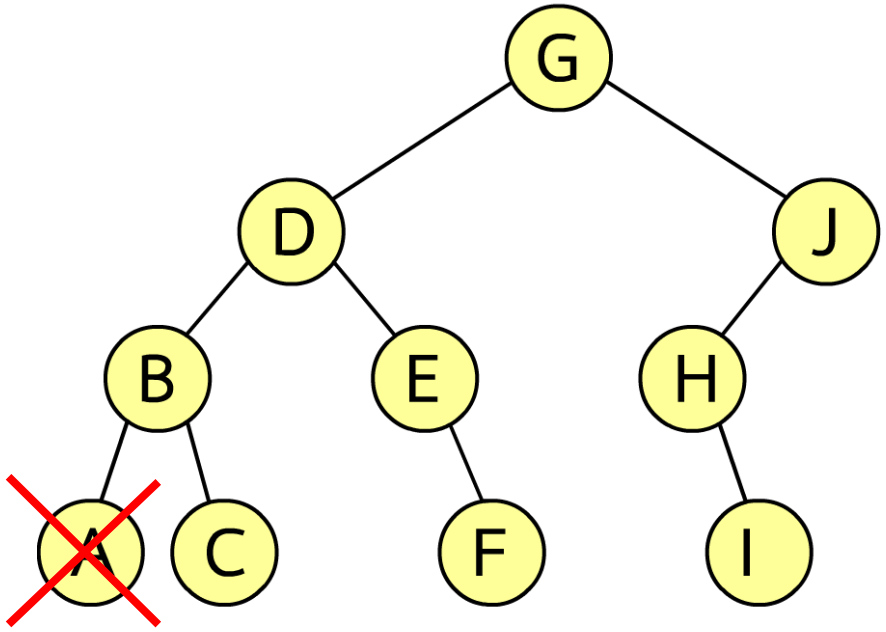
1. node ที่จะถูก delete เป็น leaf node -> ลบ node นั้นออกได้เลย

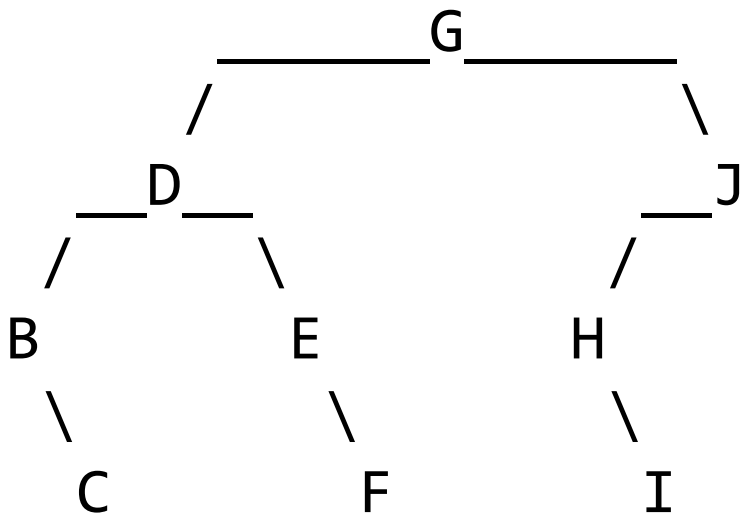
2. node ที่จะถูก delete มี child 1 ตัว -> เปลี่ยน child node ขึ้นมาเป็น node

3. node ที่จะถูก delete มี child 2 ตัว -> เปลี่ยน successor(next) inorder ขึ้นมาเป็น node และทำขั้นตอนเดียวกันกัน successor(next) inorder

Delete left node (case 1)

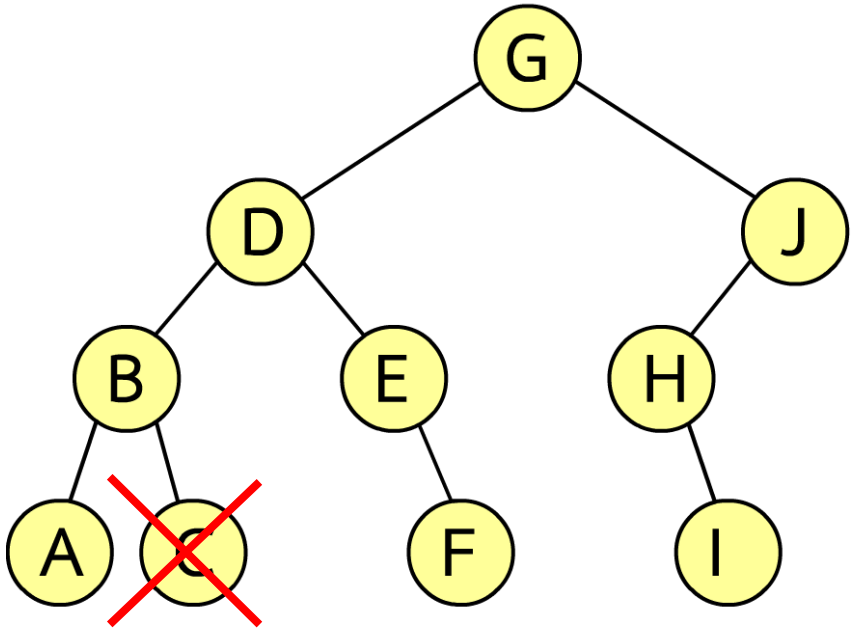
```
delete_node(&(((tree.root->left)->left)->left));
```

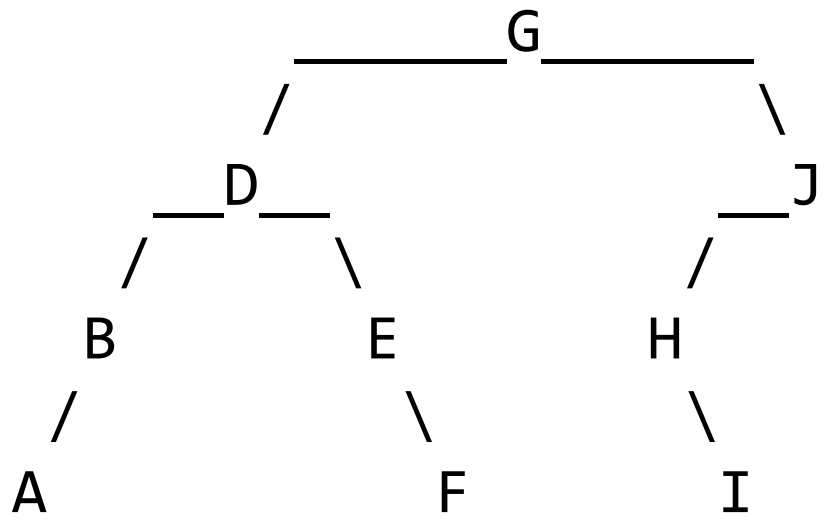




Delete left node (case 1)

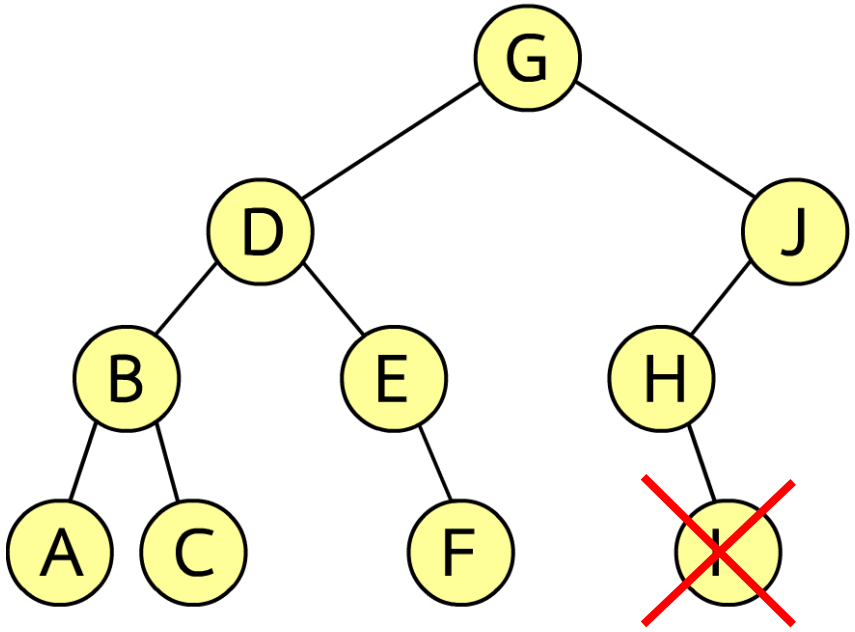
```
delete_node(&(((tree.root->left)->left)->right));
```

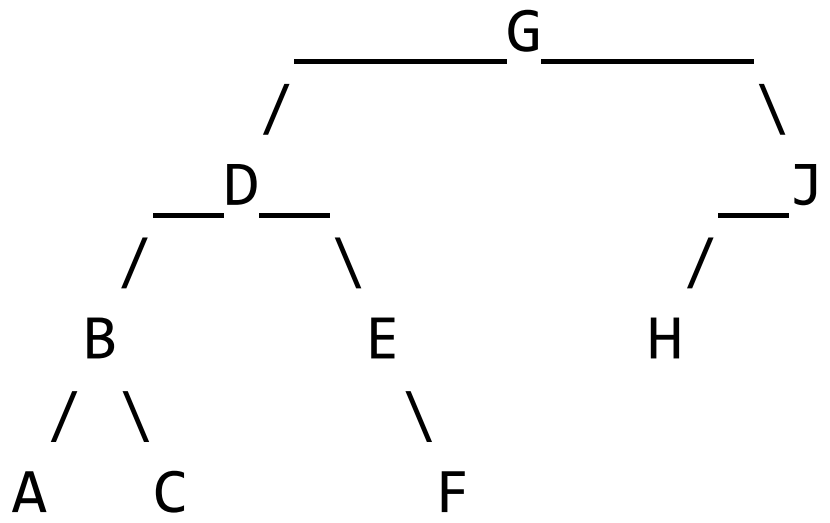




Delete left node (case 1)

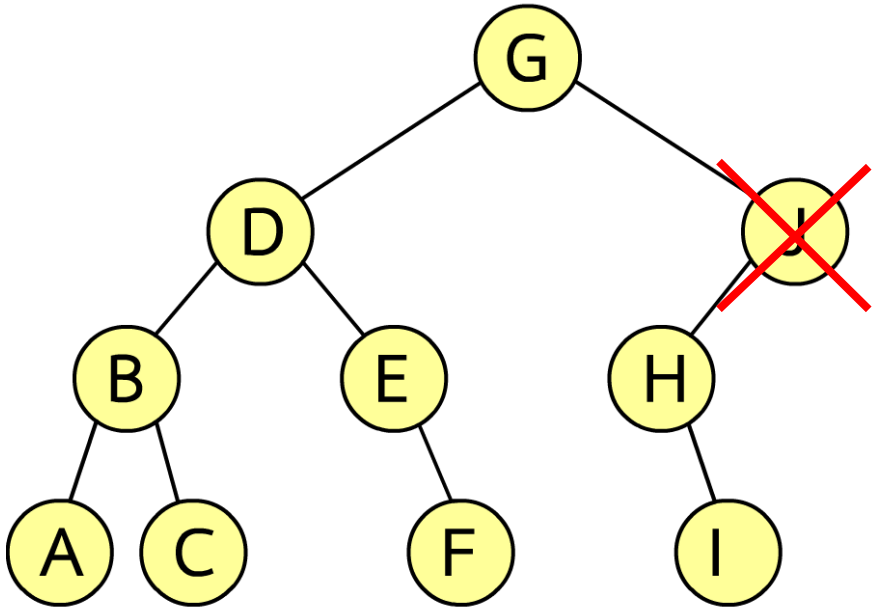
```
delete_node(&(((tree.root->right)->left)->right));
```

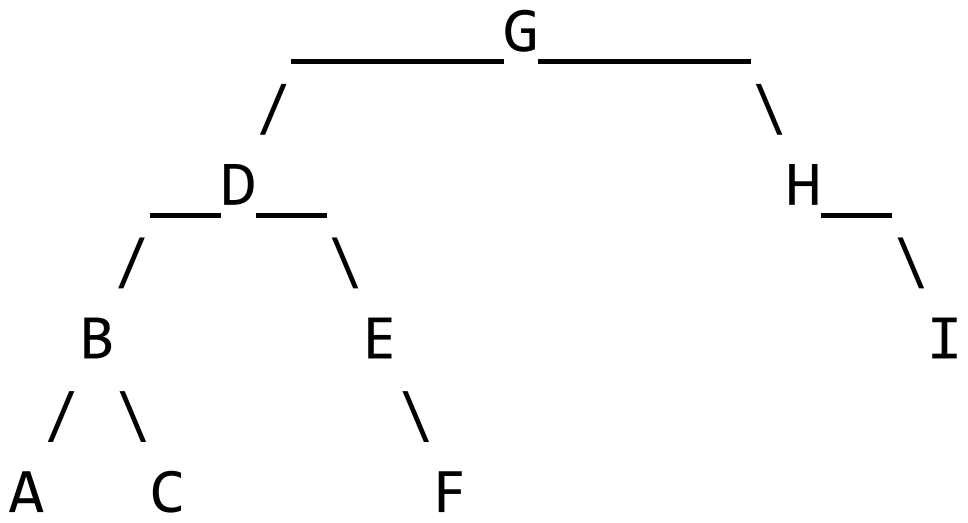




Delete left node (case 2)

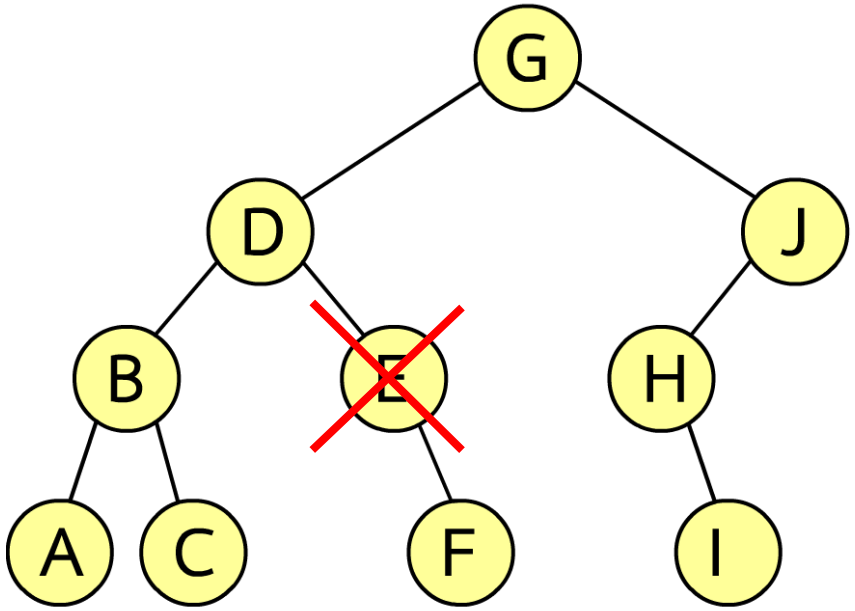
```
delete_node(&(tree.root->right));
```

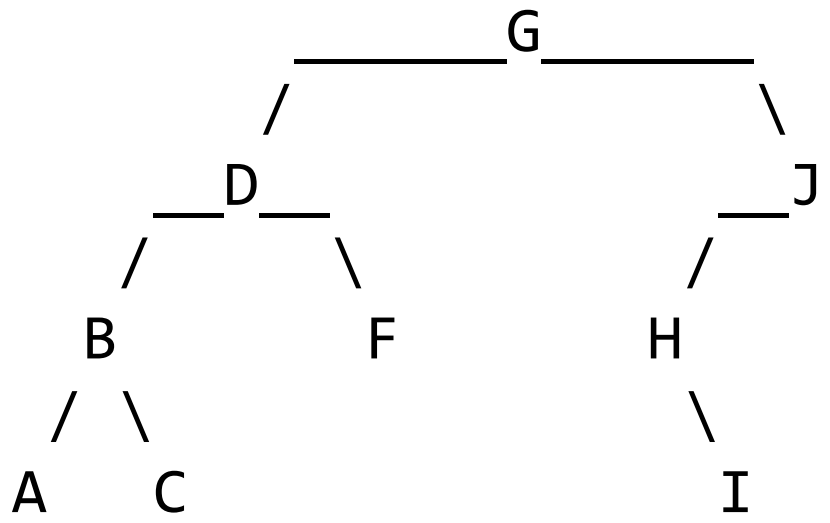




Delete left node (case 2)

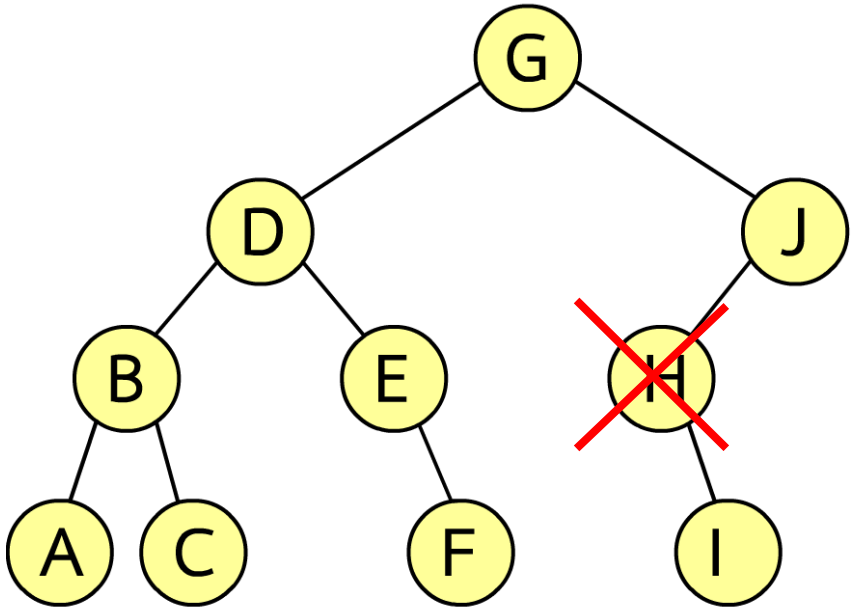
```
delete_node(&((tree.root->left)->right));
```

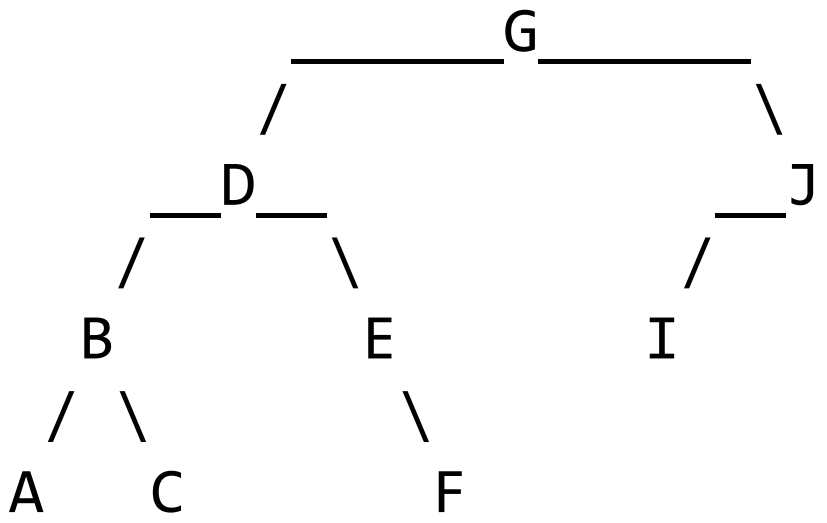


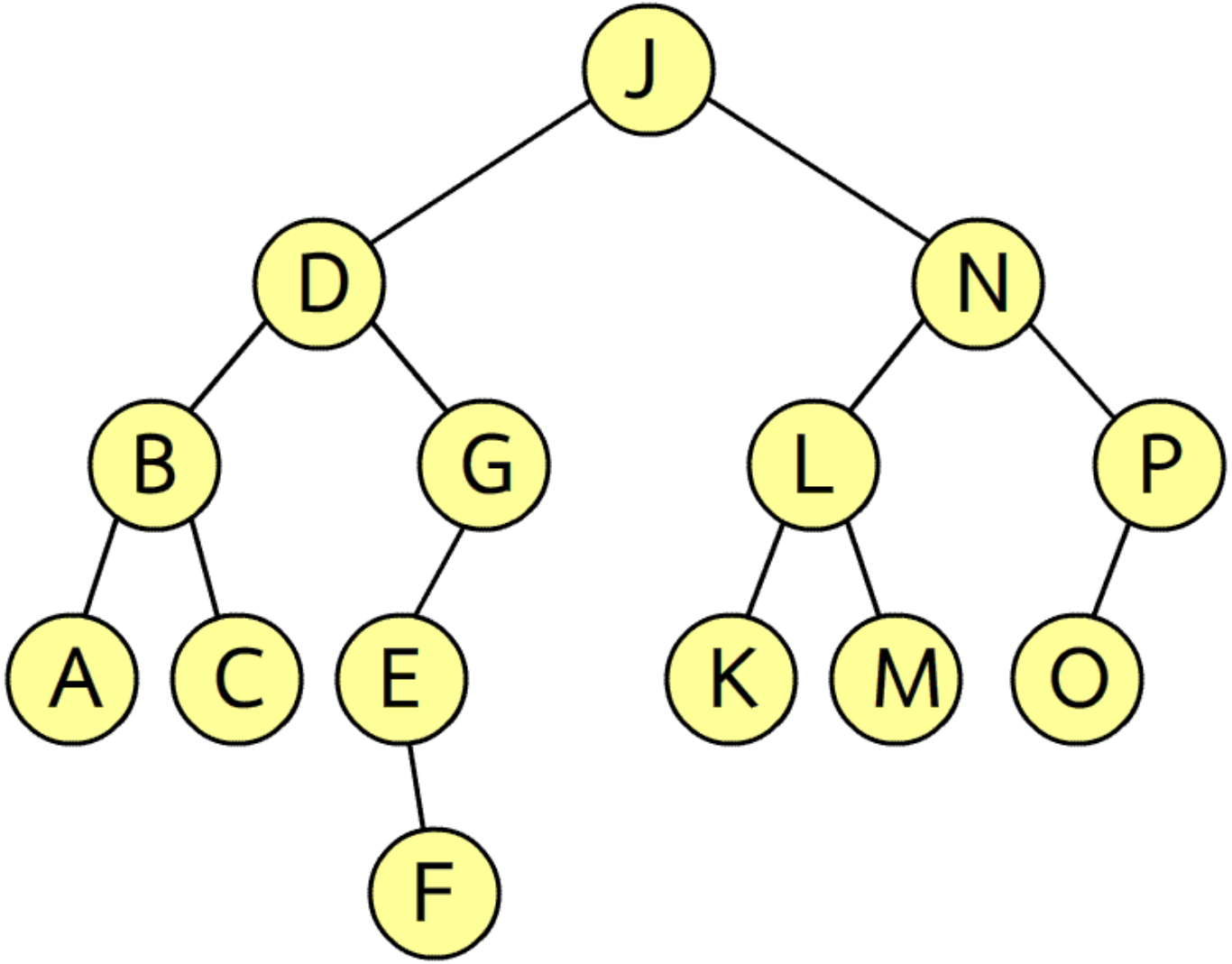


Delete left node (case 2)

```
delete_node(&((tree.root->right)->left));
```

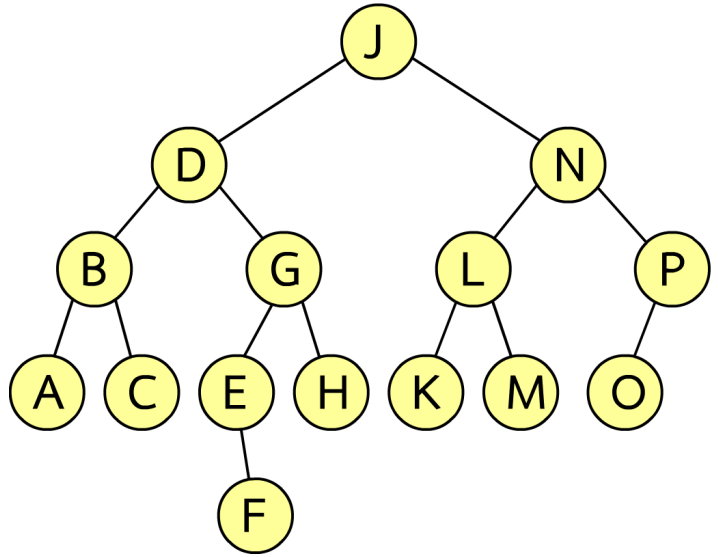


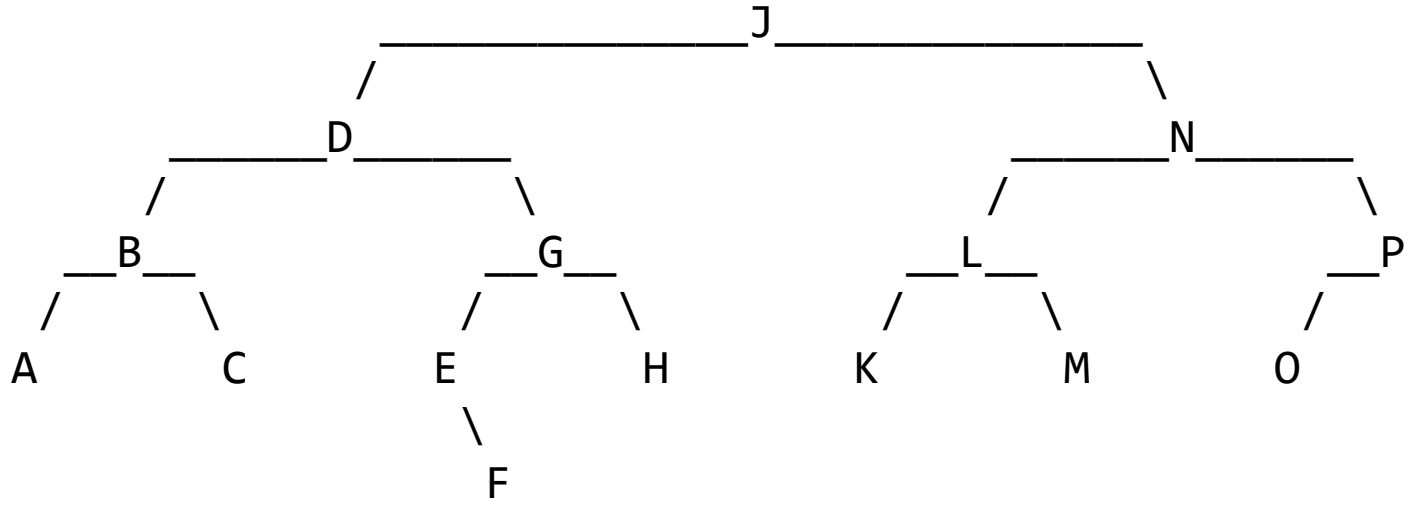



J
D
N
B
G
L
P
A
C
E
K
M
O
F

Delete left node (case 3)

```
BST tree2;
tree2.insert('J');
tree2.insert('D');
tree2.insert('N');
tree2.insert('B');
tree2.insert('G');
tree2.insert('L');
tree2.insert('P');
tree2.insert('A');
tree2.insert('C');
tree2.insert('E');
tree2.insert('H');
tree2.insert('K');
tree2.insert('M');
tree2.insert('O');
tree2.insert('F');

cout << endl;
tree2.print_tree();
tree2.print_inorder();
cout << endl;
```



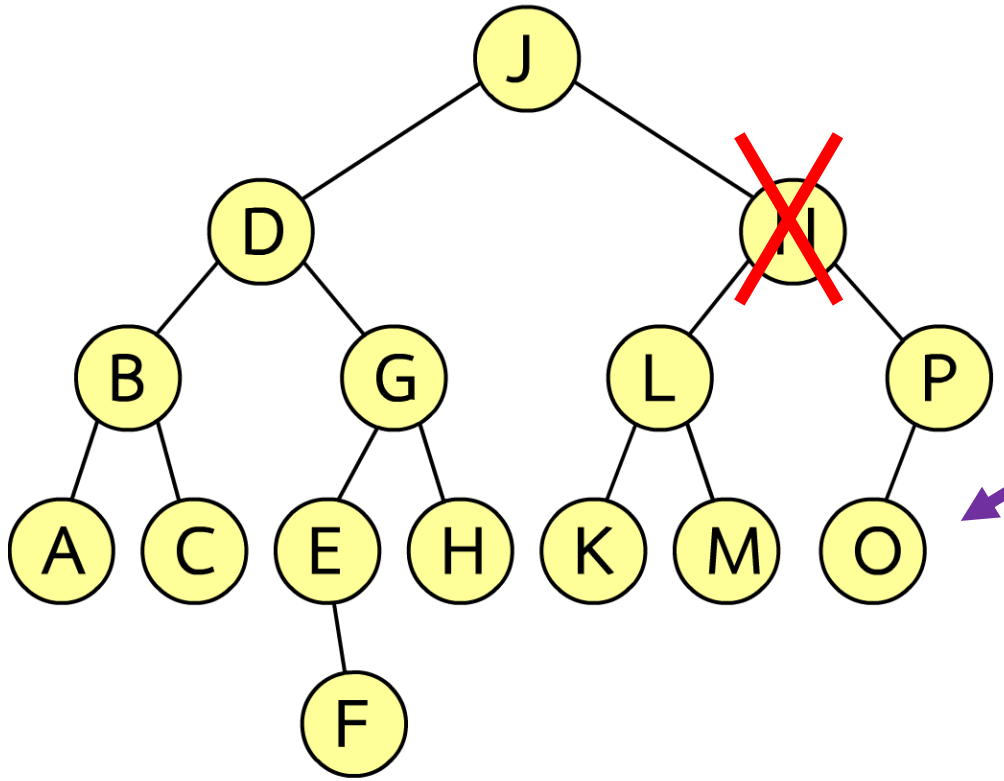



```
A B C D E F G H J K L M N O P
```

Delete left node (case 3)

A B C D E F G H J K L M N O P

```
delete_node(&(root->right));
```

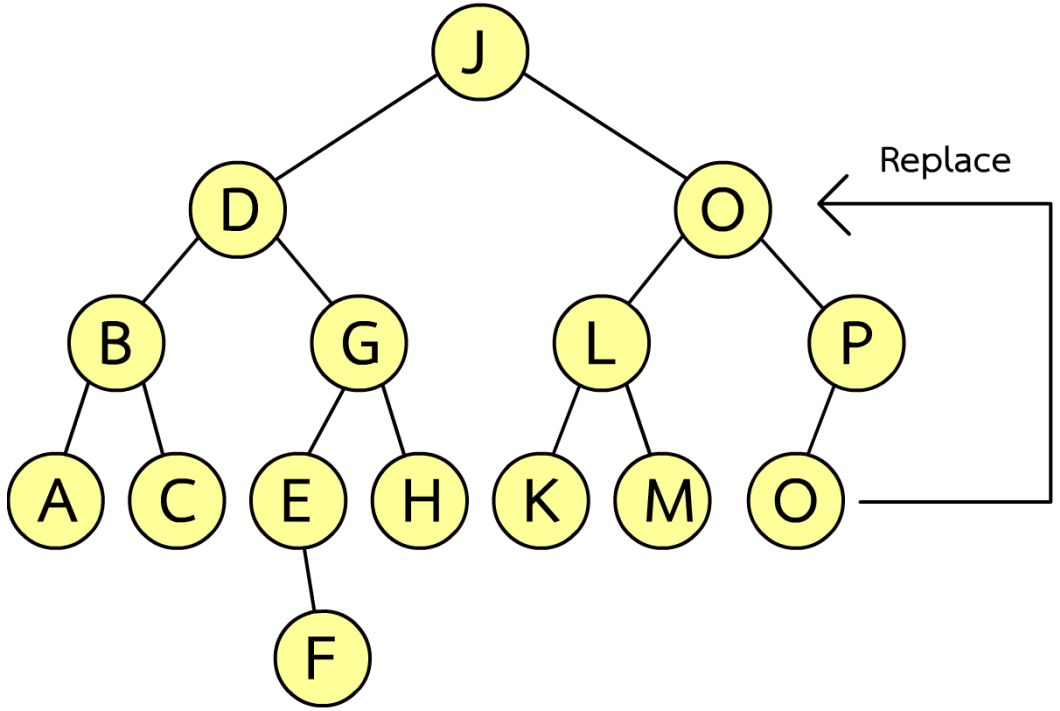


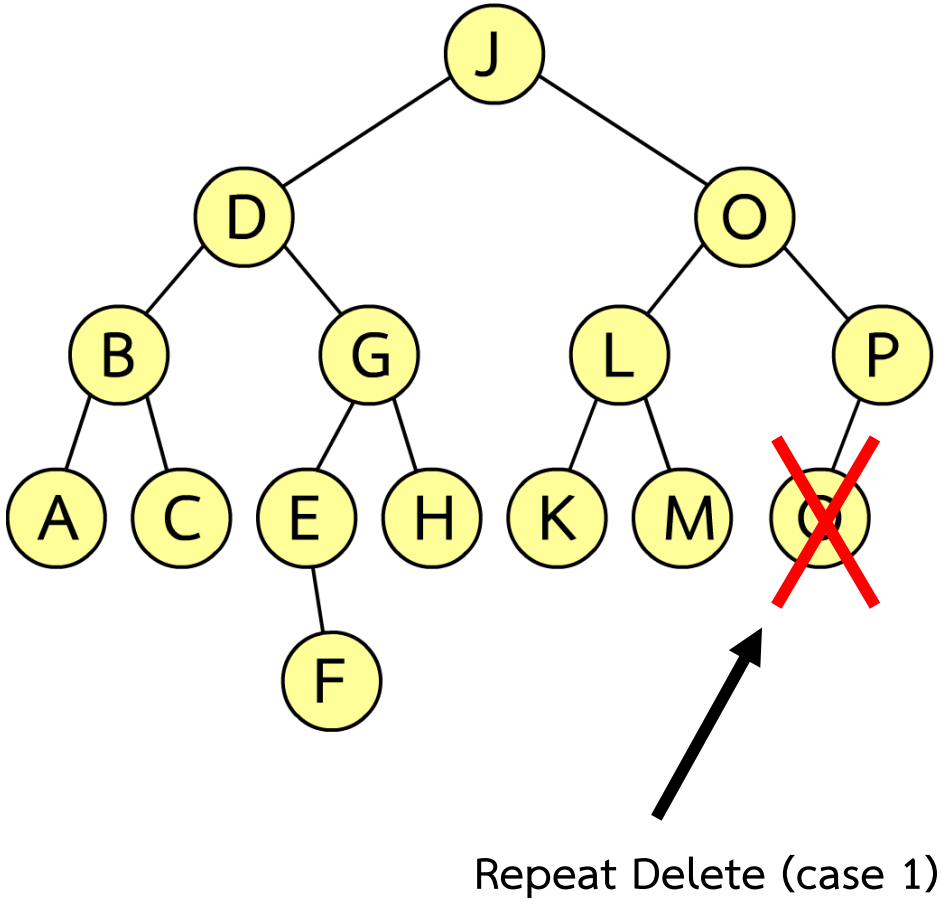
successor of N is O

Delete left node (case 3)

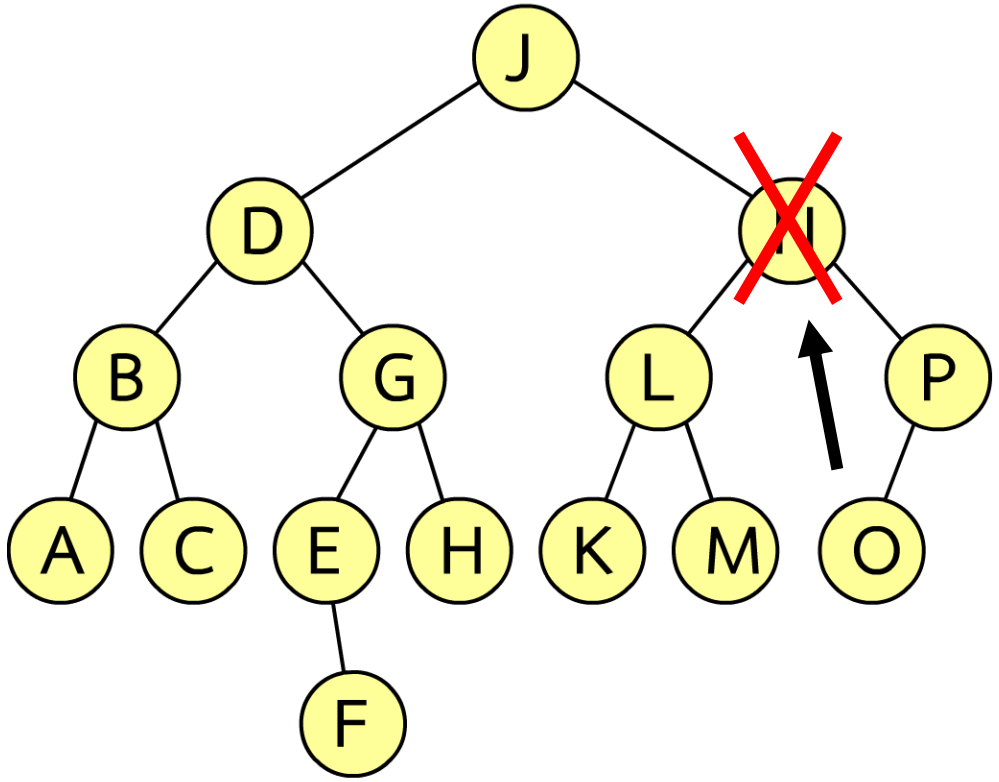
A B C D E F G H J K L M N O P

```
delete_node(&(root->right));
```





Delete left node (case 3)



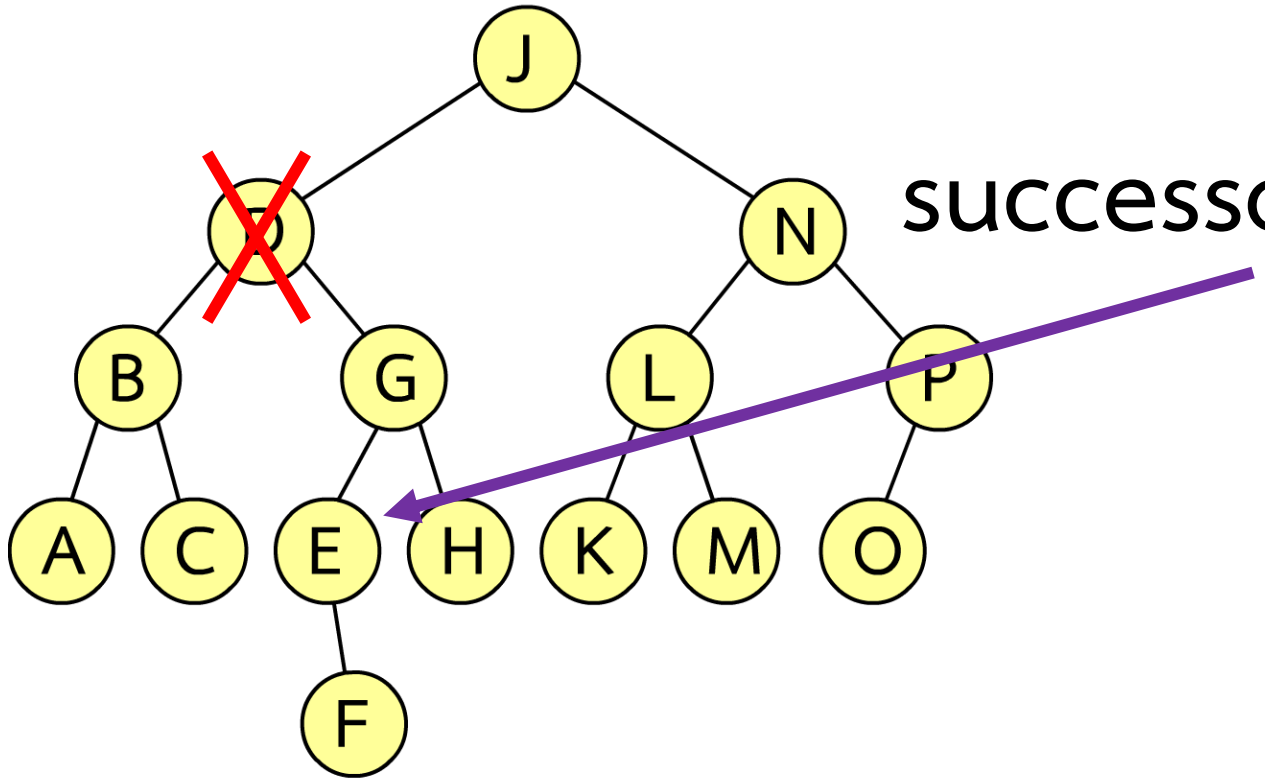

```
delete_node(&(tree2.root->right));

cout << endl;
tree2.print_tree();
```

```
                _______________J_______________
               /                               \
       _______D_______                 _______O_______
      /               \               /               \
   __B__           __G__           __L__             P
  /     \         /     \         /     \
 A       C       E       H       K       M
          \
           F
```

Delete left node (case 3)

A B C D E F G H J K L M N O P

```
delete_node(&(tree2.root->left));
```

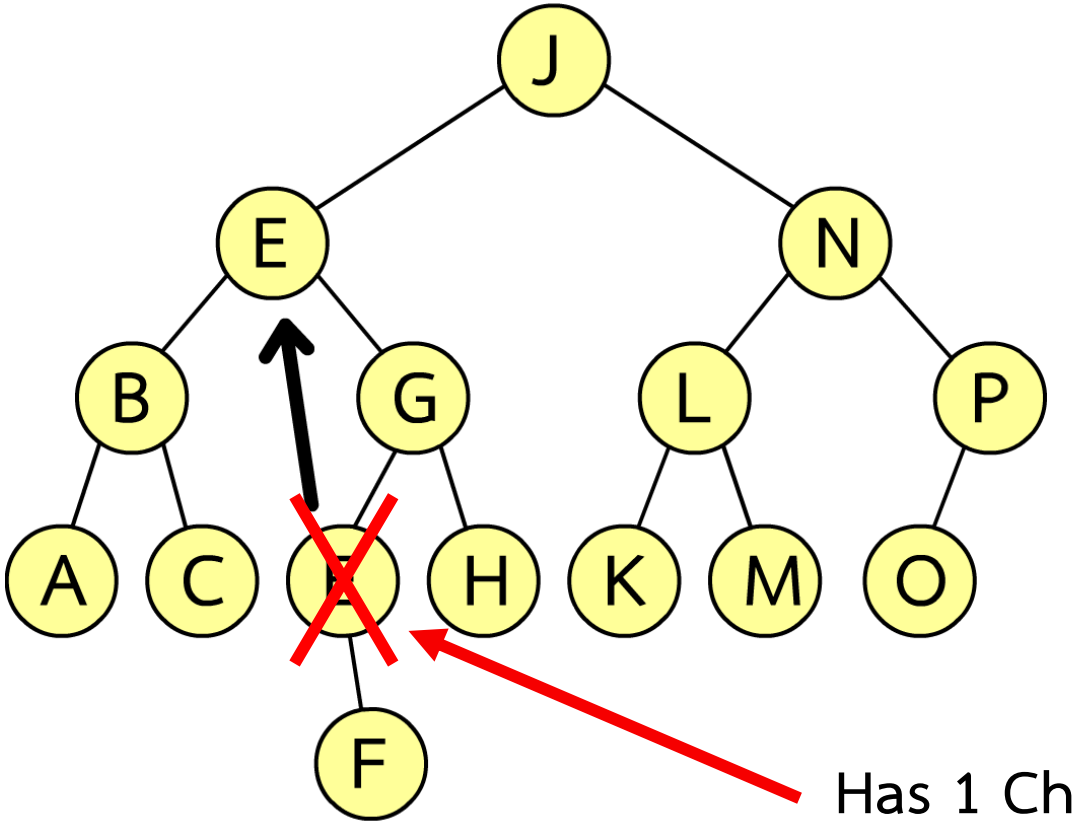


successor of D is E

Delete left node (case 3)

A B C D E F G H J K L M N O P

```
delete_node(&(tree2.root->left));
```

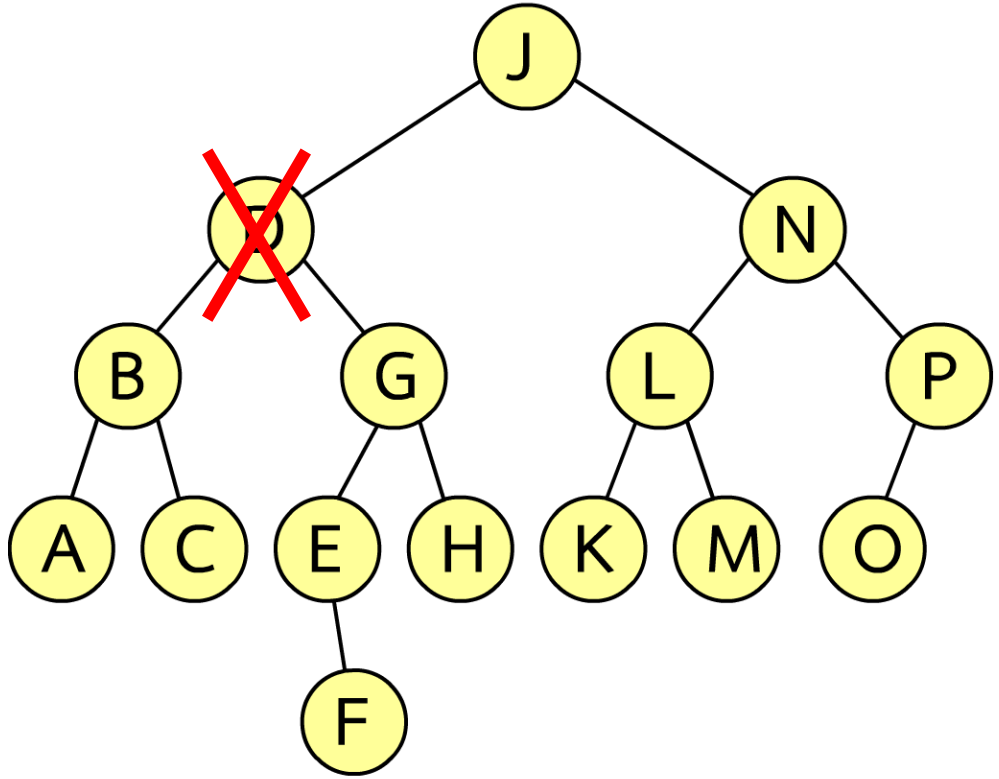


Has 1 Child (case 2) -> replace with child

Delete left node (case 3)

A B C D E F G H J K L M N O P

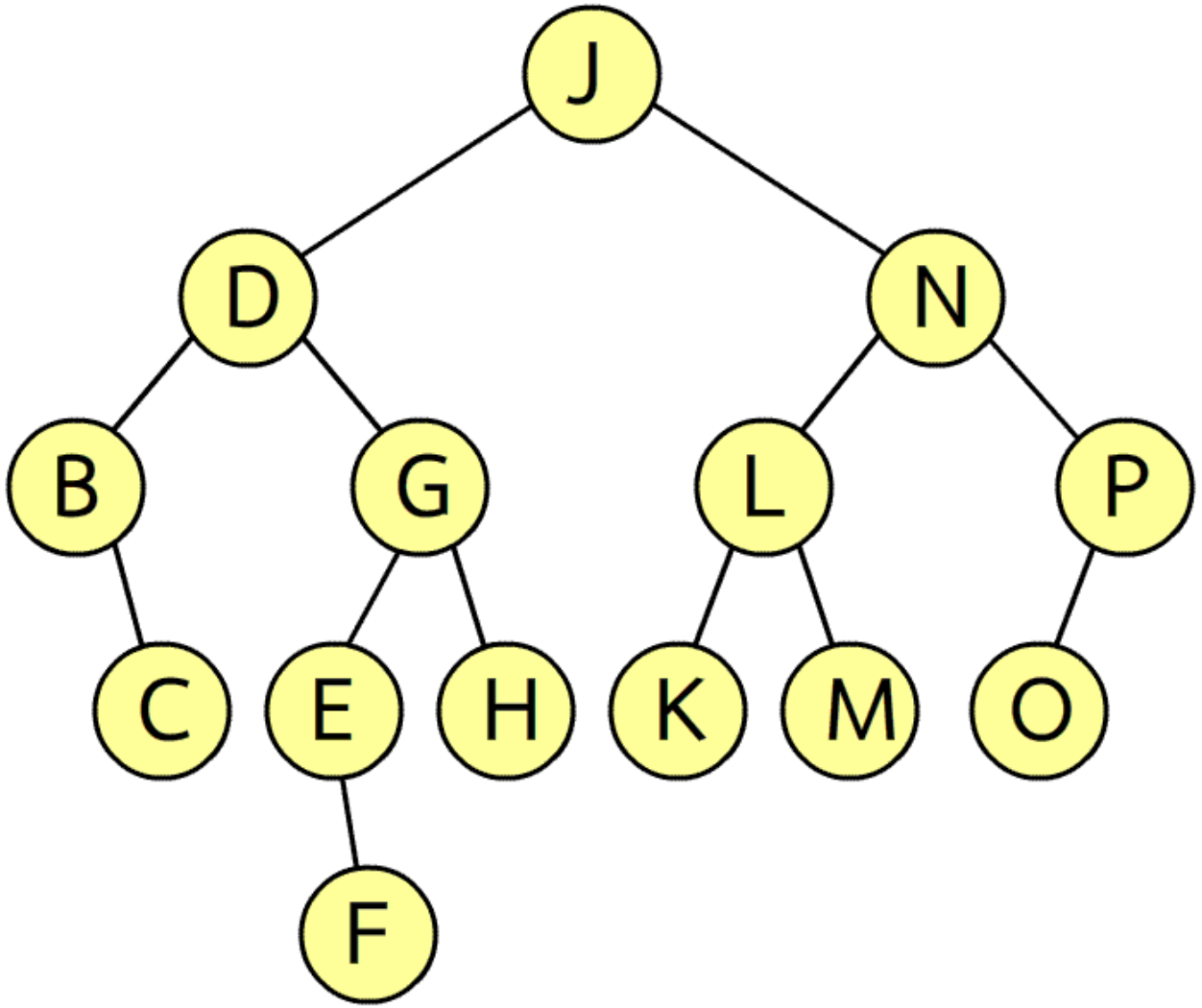
```
delete_node(&(tree2.root->left));
```



Result:

```
                _______J_______
               /               \
           __E__               __N__
          /     \             /     \
         B       G           L       P
        / \     / \         / \     /
       A   C   F   H       K   M   O
```

BCDEFGHJKLMNOP



```
void delete_node(Node** n){
    if((*n)->left == nullptr && (*n)->right == nullptr){ // leaf node (case 1)
        delete (*n);
        (*n) = nullptr;
    }
    else if((*n)->left == nullptr){ // replace with right node (case 2)
        (*n) = (*n)->right;
    }
    else if((*n)->right == nullptr){ // replace with left node (case 2)
        (*n) = (*n)->left;
    }
    else{ // replace with next inorder and process to next inorder node (case 3)
        Node** p = &((*n)->right);
        while((*p)->left != nullptr){
            p = &((*p)->left);
        }
        (*n)->data = (*p)->data;
        delete_node(p);
    }
}
```

Recursion!

Asymptotic analysis

Insert

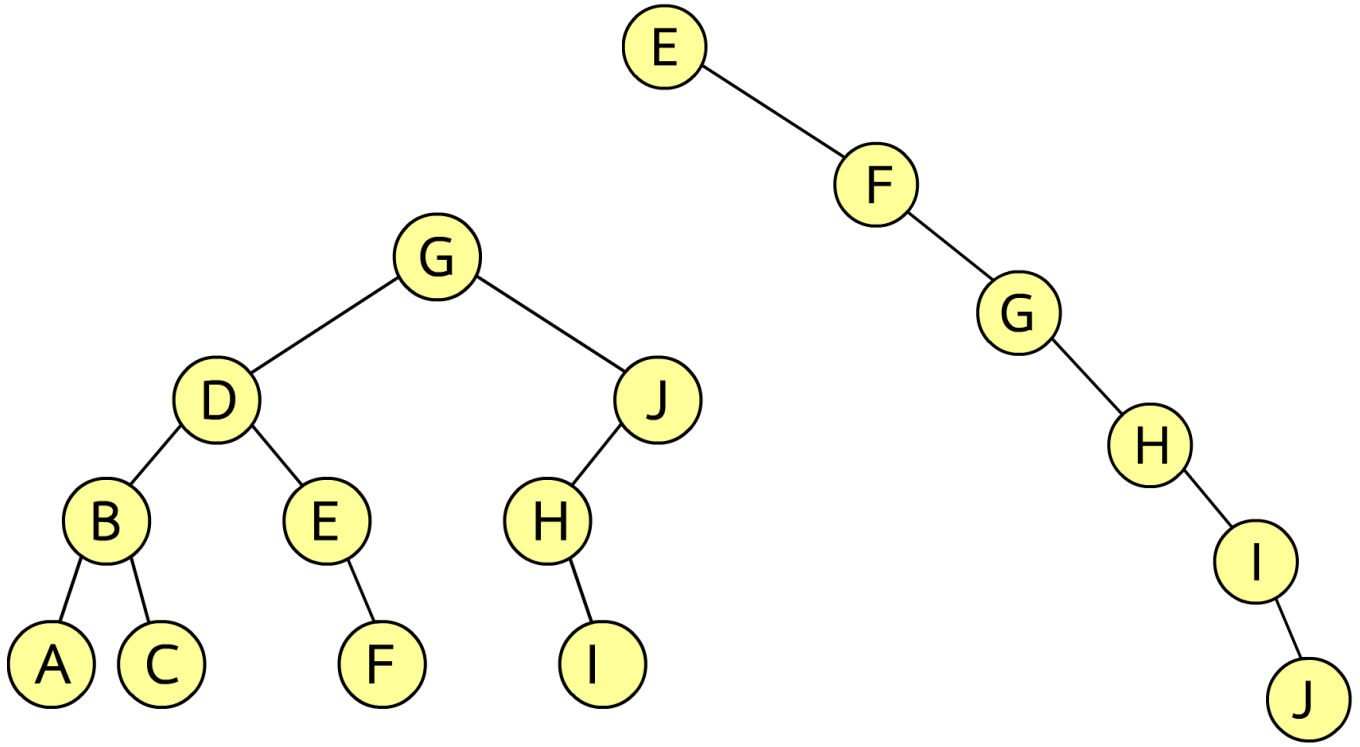
- base case (1)

- worse case (n)

Search

- base case (1)

- worse case (n)



Asymptotic analysis (balance binary tree)

Insert

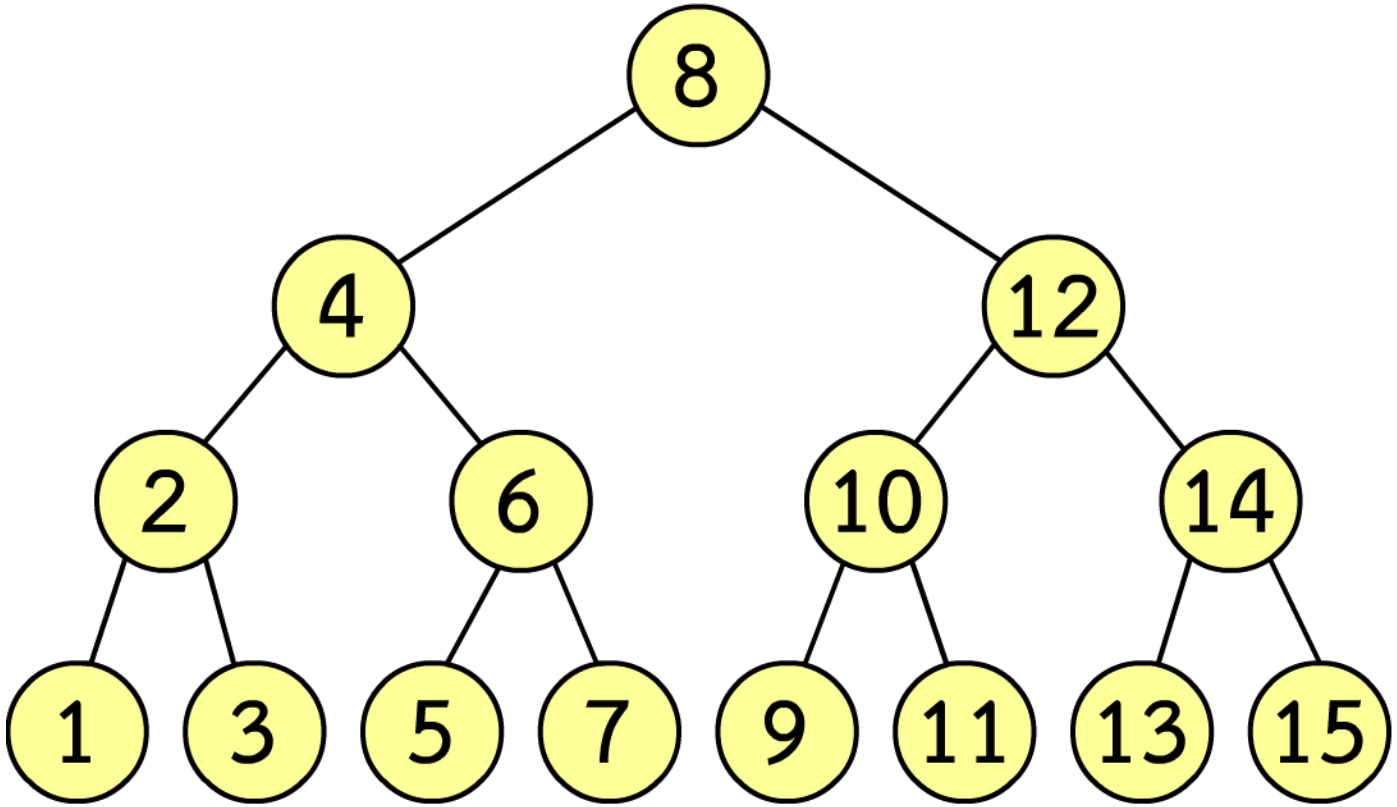
- base case (1)

- worse case (log n)

Search

- base case (1)

- worse case (log n)



advantage

- access nearly log n complexity in case of balance tree

- array and linked list using exact n complexity to access member

- เข้าถึงได้ด้วย log n complexity (ในกรณีที่เป็น balance tree)

- array และ linked list ใช้ n complexity ในการเข้าถึงเท่านั้น

log n ?

n	Log(n)
1	1
1,024	10
1,048,576	20
1,099,511,627,776	40

AVL tree

- self balance binary search tree
- always log n access (search and insert)
- invented in 1962 (2 year after binary tree)

https://en.wikipedia.org/wiki/AVL_tree#/media/File:AVL_Tree_Example.gif

Conclude

- what is BST

- insert , search and delete

- Big O

LAB