OOP & data struct

12. Binary tree

BY SOMSIN THONGKRAIRAT


TREE
POINTER
LINKED LIST
LINKED LIST
imgflip.com

What is tree

Basic non-linear data structural

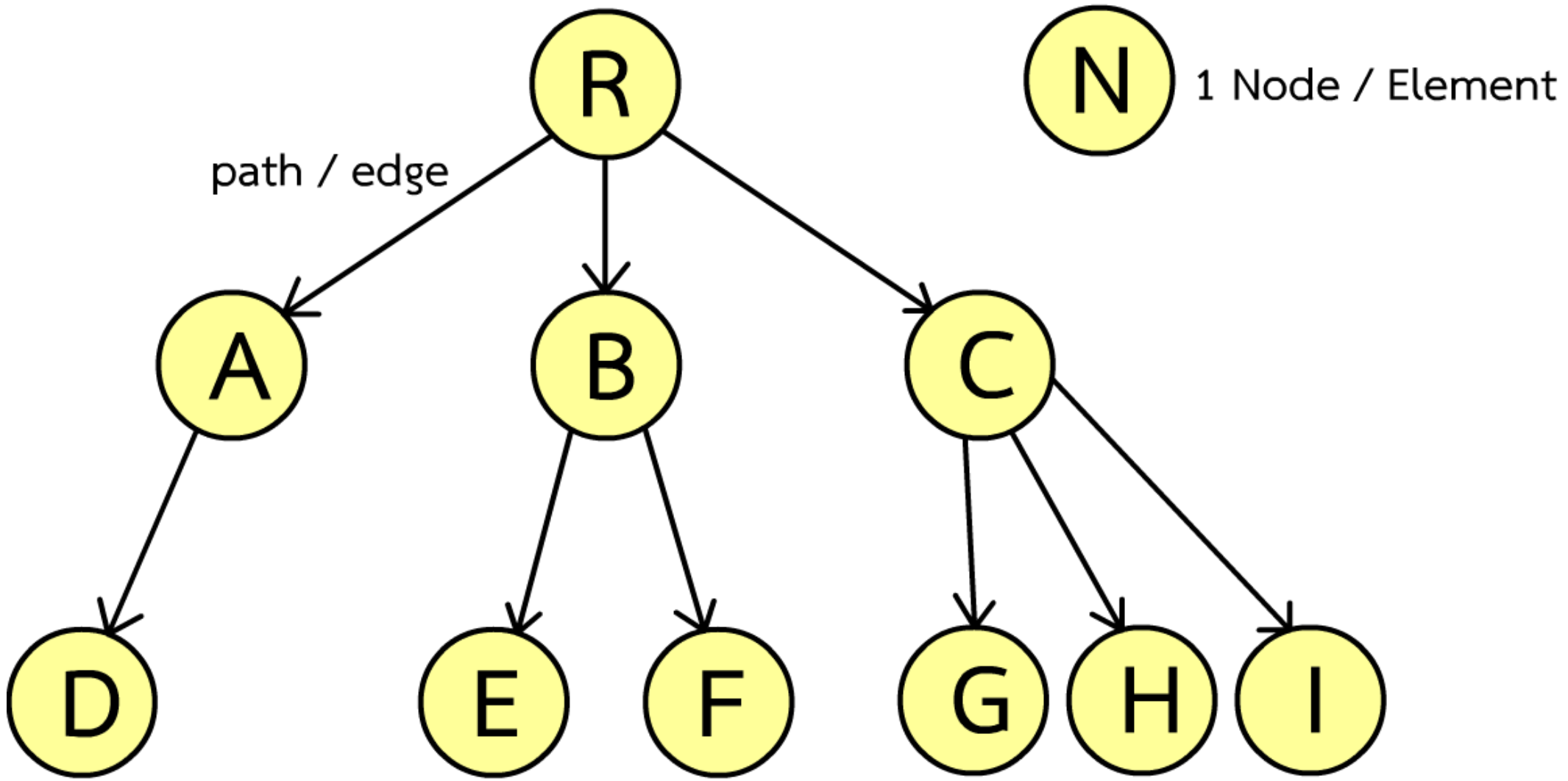
Tree represents hierarchical structure

Each element contain data and several paths to another members

เป็น structure พื้นฐานแบบ non-linear

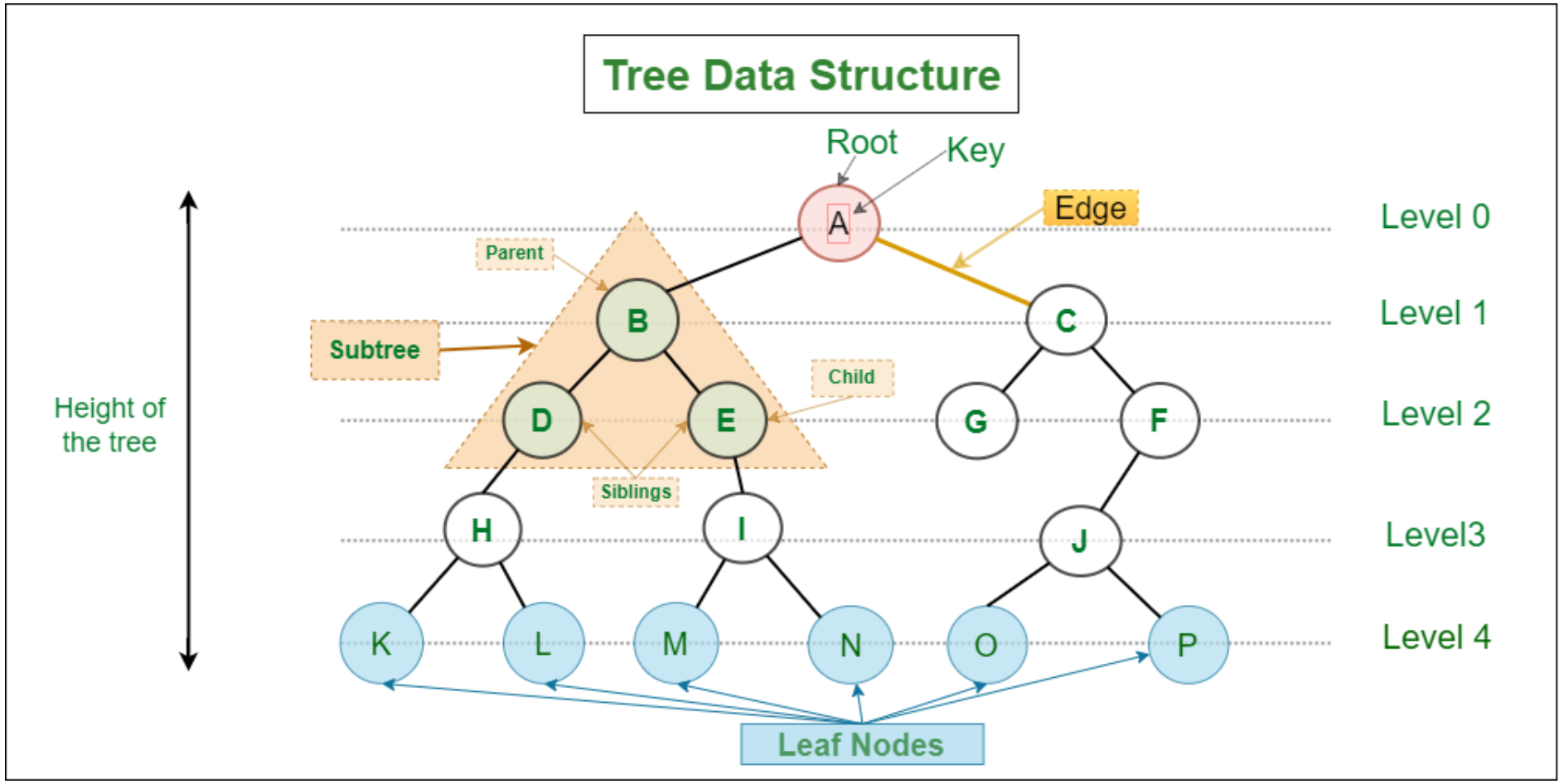
มีการเข้าถึงแบบเป็นชั้น (hierarchy)

สมาชิกแต่ละตัวจะเก็บ data และ เส้นทางไปยัง สมาชิกตัวต่อไป (มีได้หลายตัว)

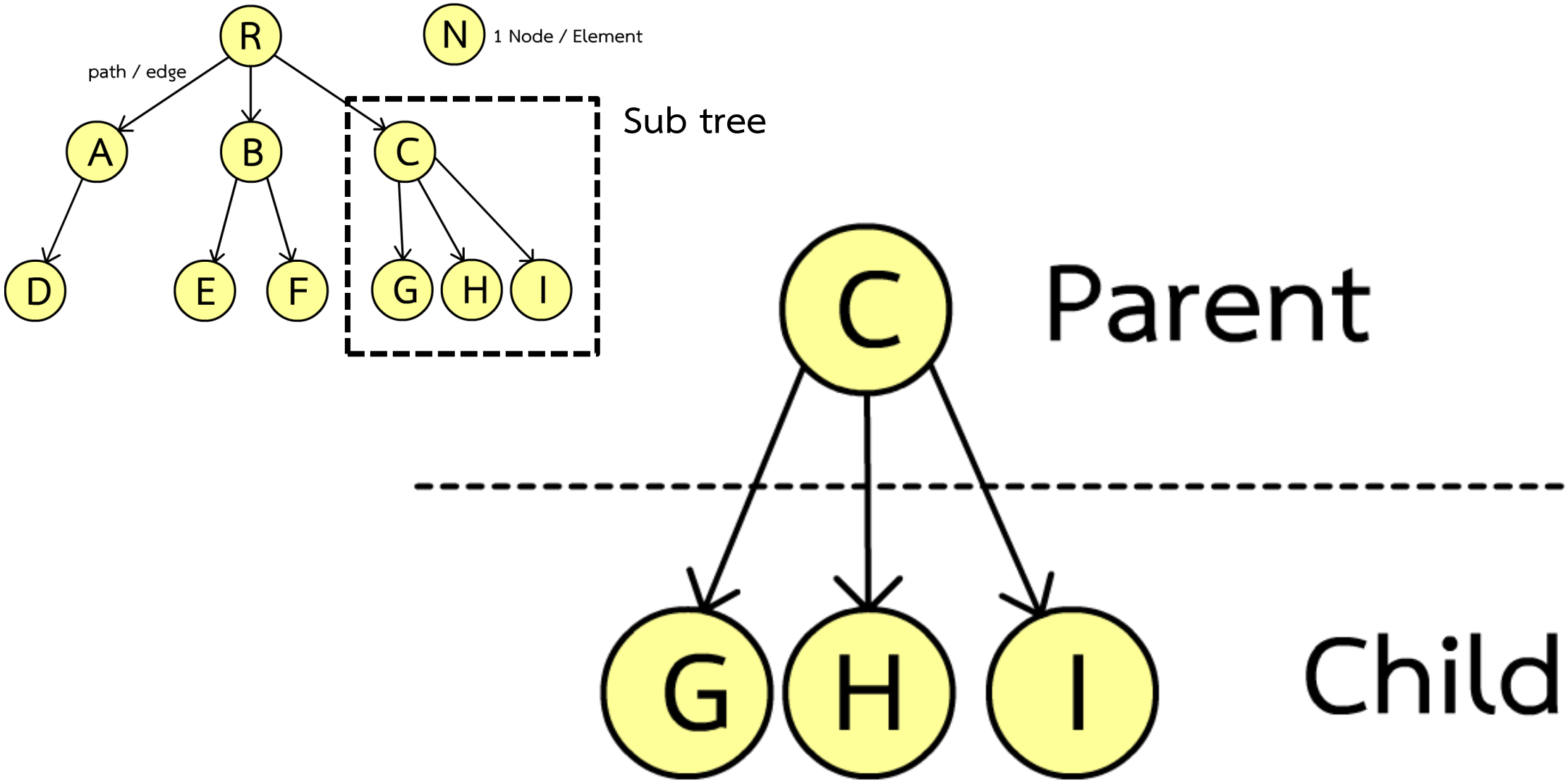

R
N
1 Node / Element
path / edge
A
B
C
D
E
F
G
H
I

Component

hierarchical structure

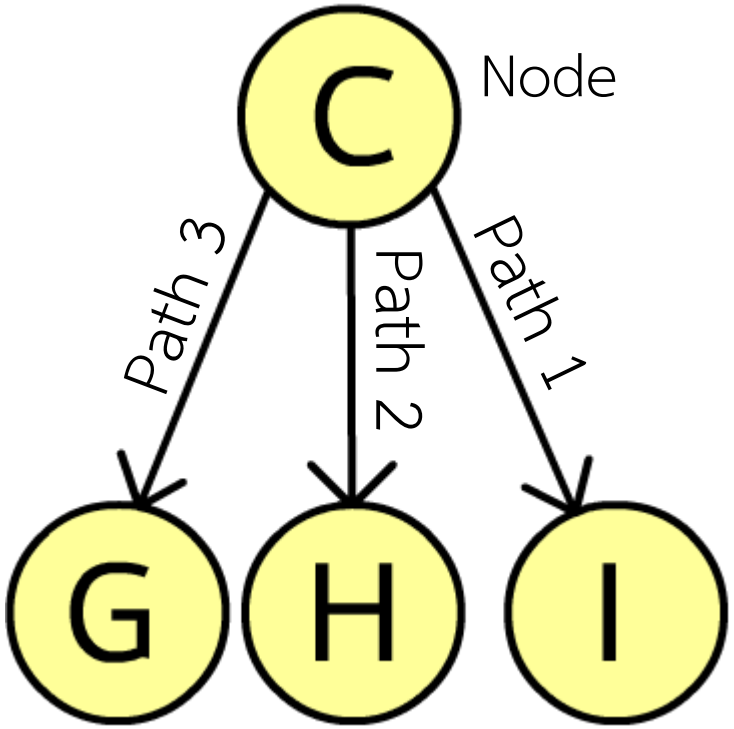


https://www.geeksforgeeks.org/introduction-to-binary-tree-data-structure-and-algorithm-tutorials/

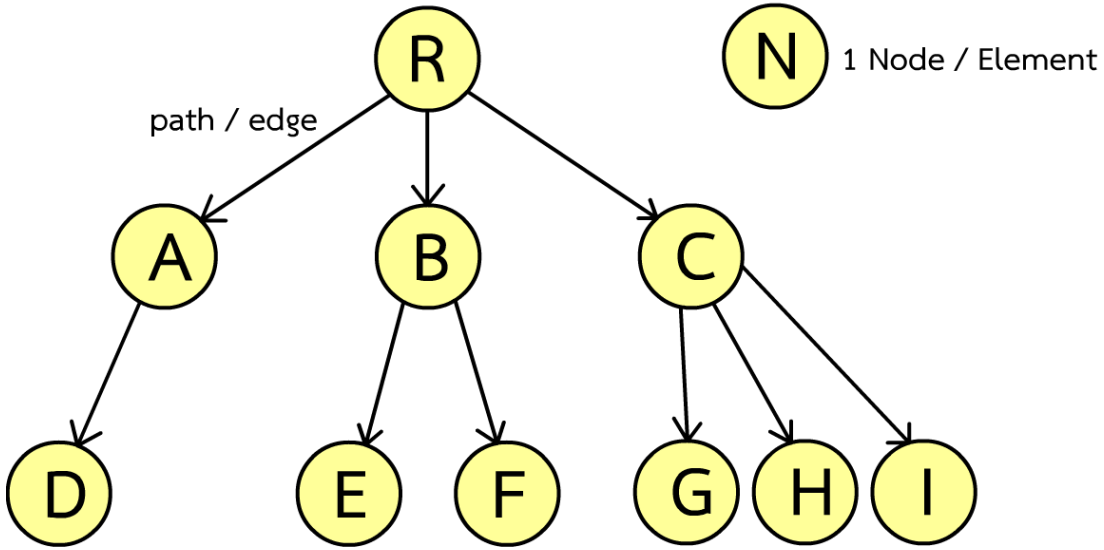

R
N
1 Node / Element
path / edge
A
B
C
Sub tree
D
E
F
G
H
I
C
Parent
G
H
I
Child

Tree node

```
class Node{
      char data;
      Node* path1;
      Node* path2;
      Node* path3;
      Node* ...
};
```



Tree!

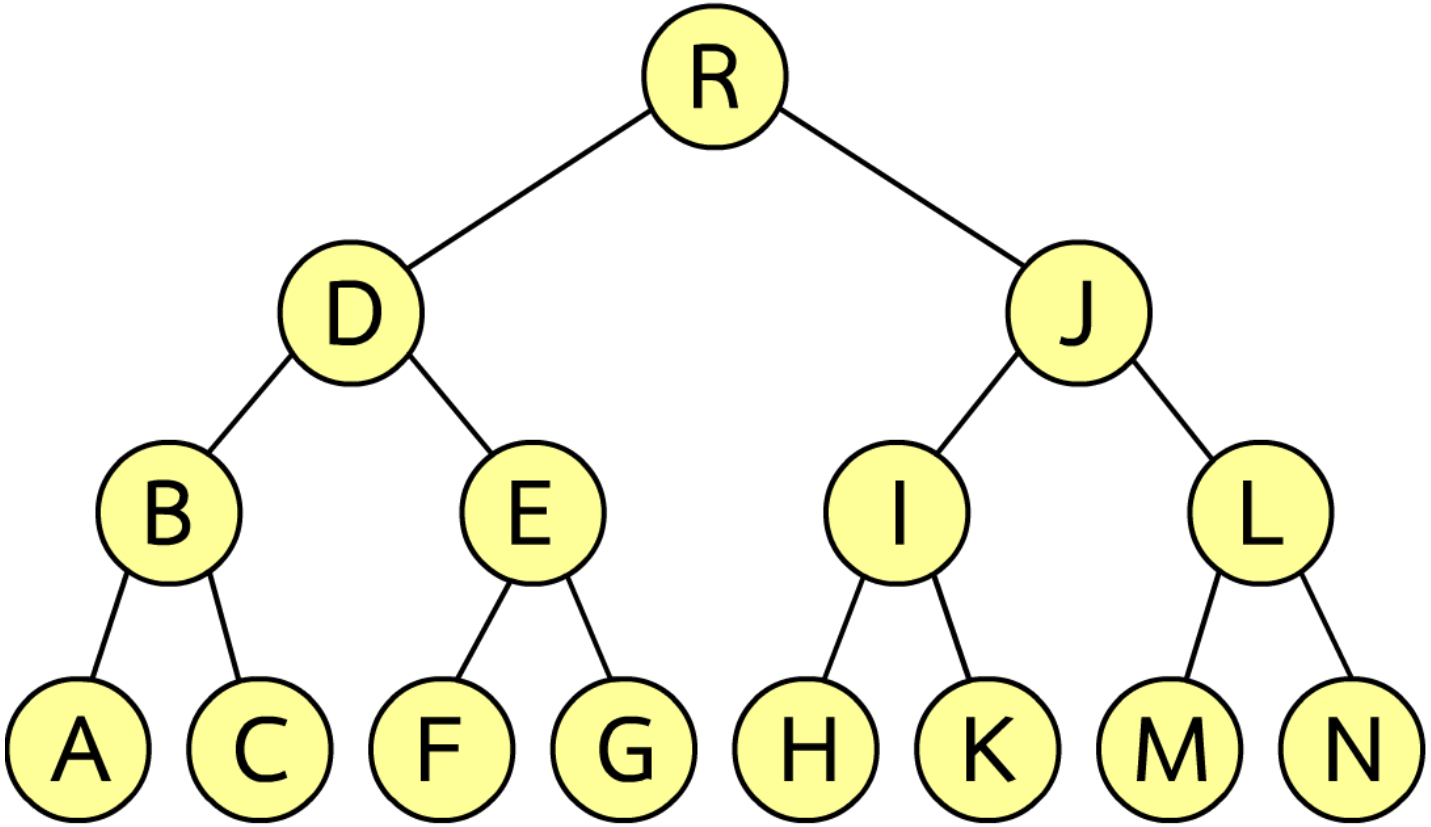


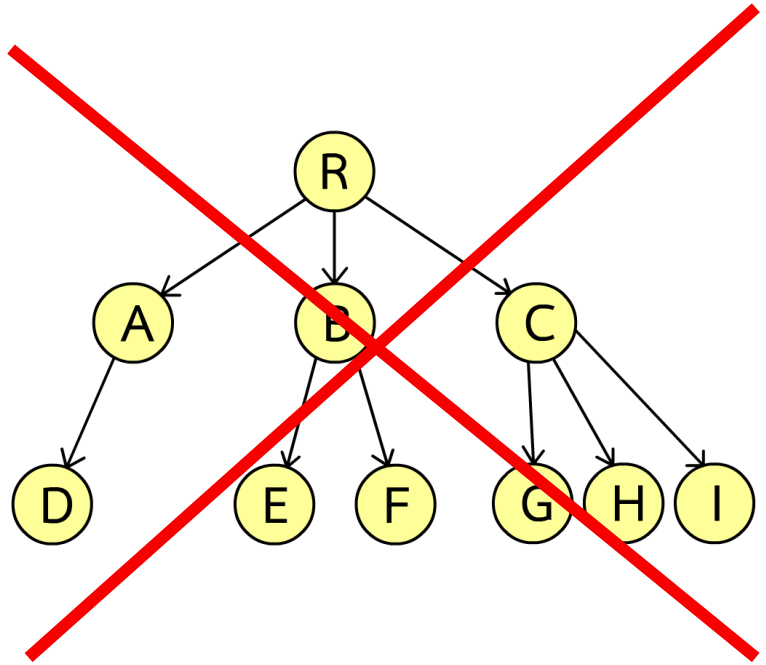
Binary tree

- a tree data structure where each node has at most 2 children usually called left and right node

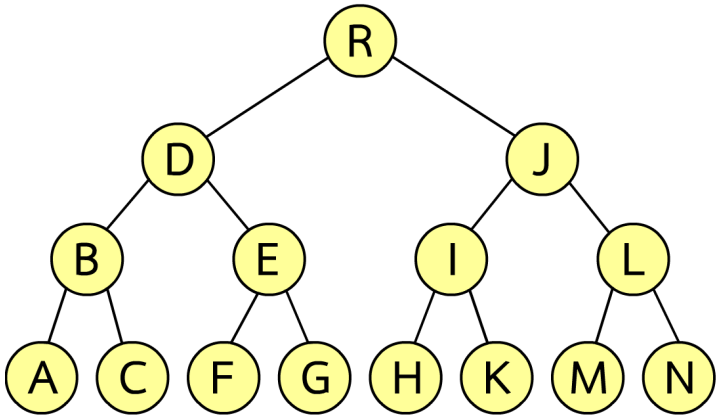
- data structure แบบ tree ที่มีเพียงแค่ 2 node ที่เป็น child node

Binary tree

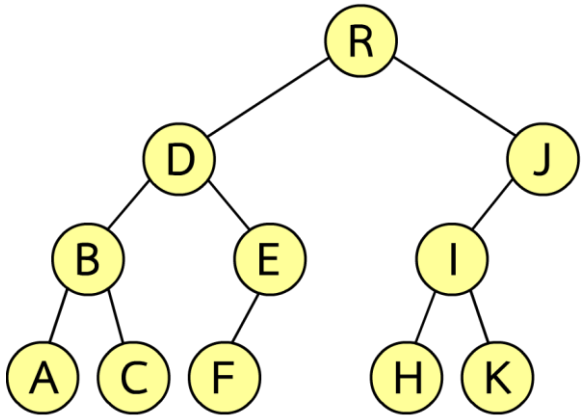


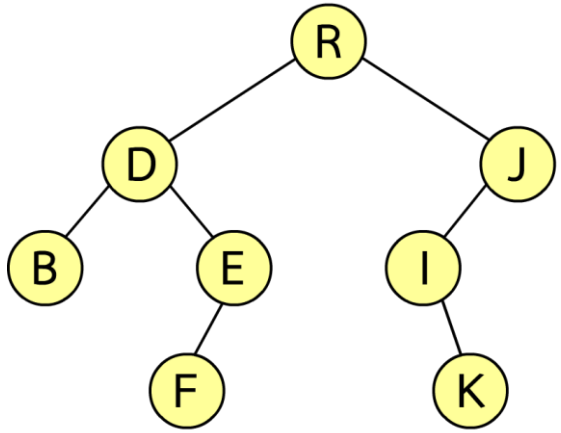


Balance tree? – not necessary



Balance tree



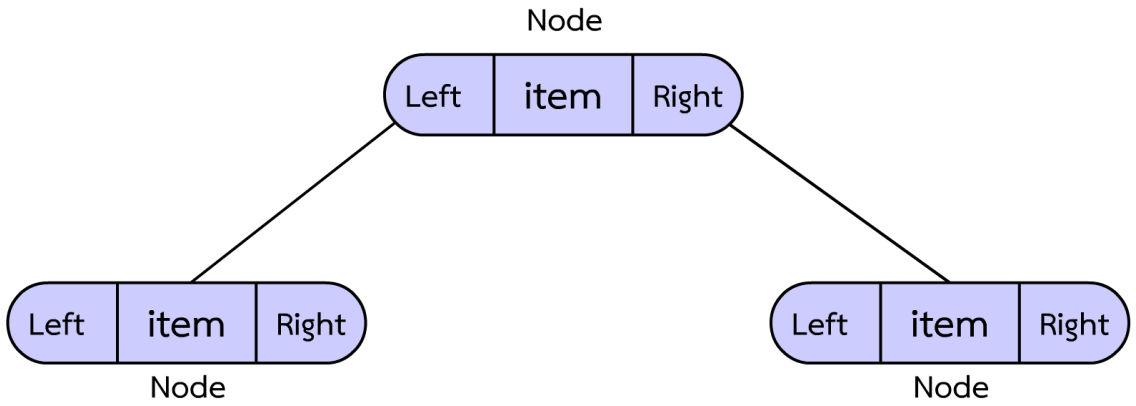


Unbalance tree (but still called “tree”)

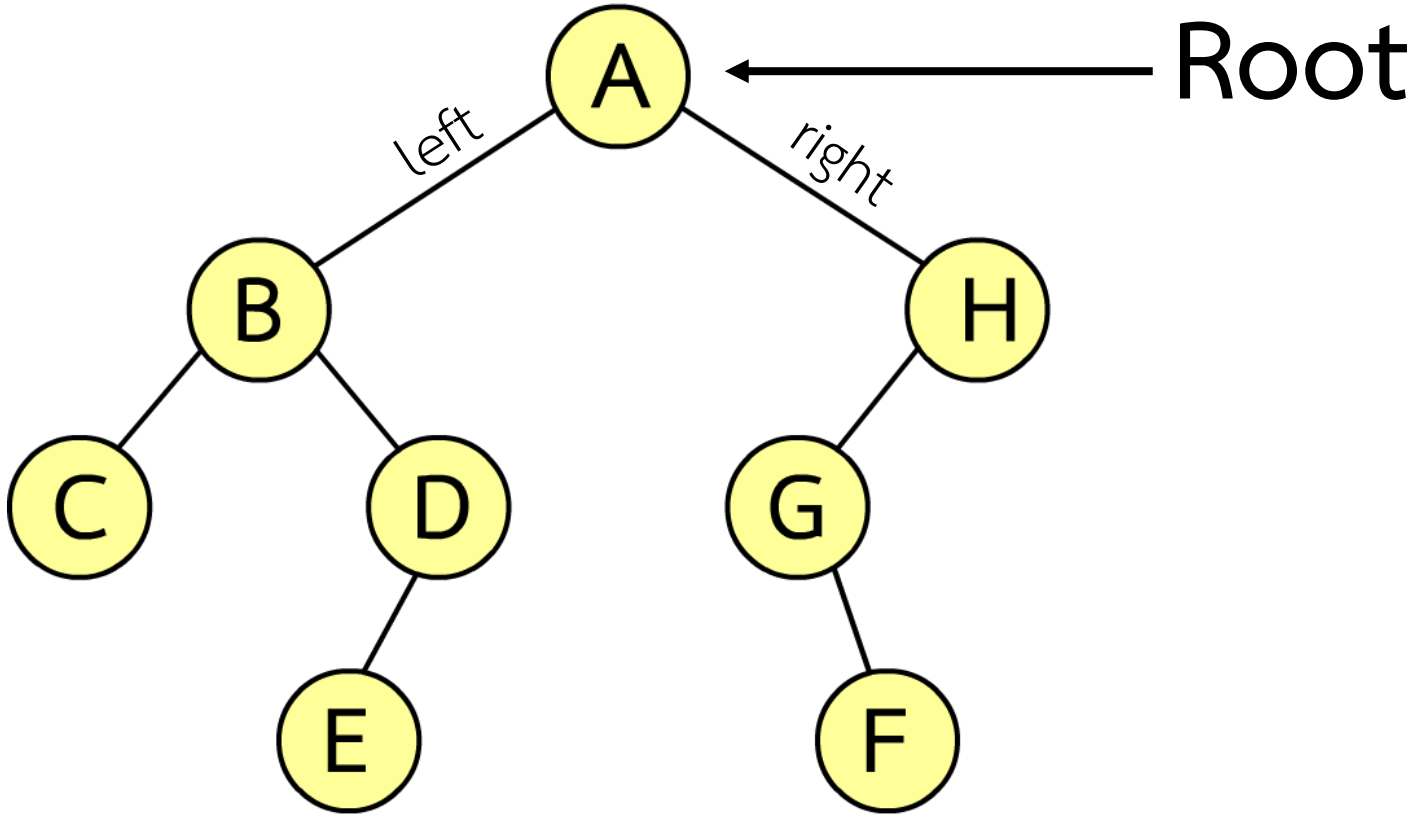
Tree node

```
class Node{
    char data;
    Node* left;
    Node* right;

    Node(char item){
        data = item;
        left = nullptr;
        right = nullptr;
    }
};
```



Build tree OOP



*ขออนุญาตใช้ main อีกนะครับเนื่องจากยุ่งยาก

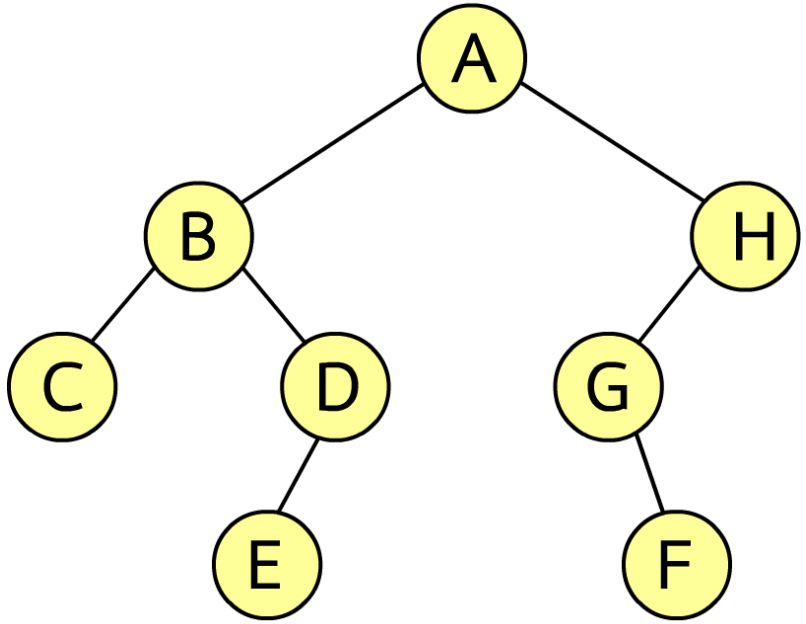
Build tree OOP

```
Node *root = new Node('A');

root->left  = new Node('B');
root->right = new Node('H');

(root->left) ->left  = new Node('C');
(root->left) ->right = new Node('D');
(root->right)->left  = new Node('G');

((root->left)->right)->left = new Node('E');
((root->right)->left)->right = new Node('F');
```

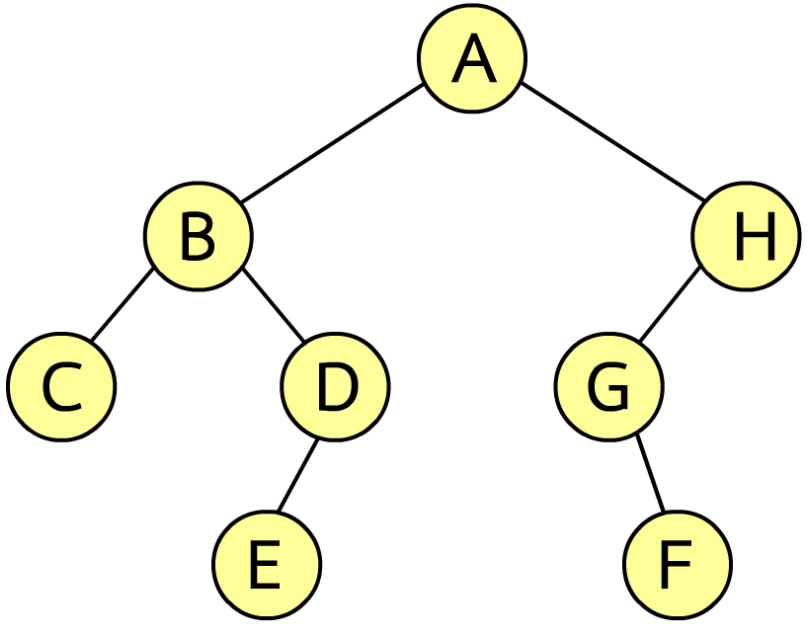


How to check (print) recursion!

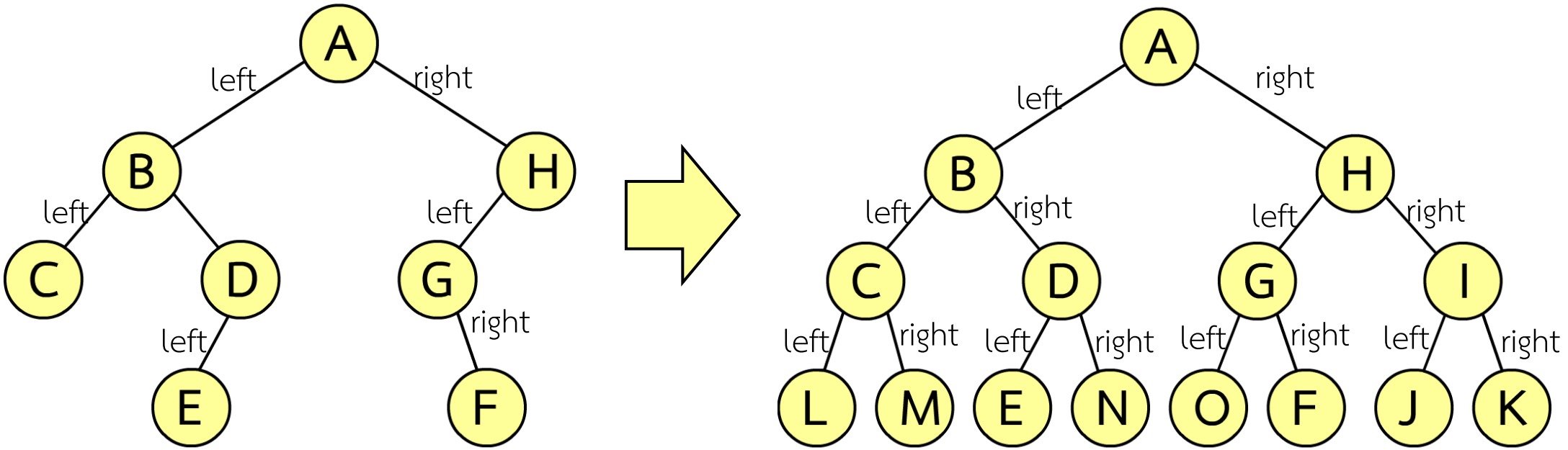
```
print_tree(root); // only 1 space character
```

Result :

```
                _______A_______
               /               \
           __B__               __H
          /     \             /
         C       D           G
                /             \
               E               F
```



Continue to build balance tree

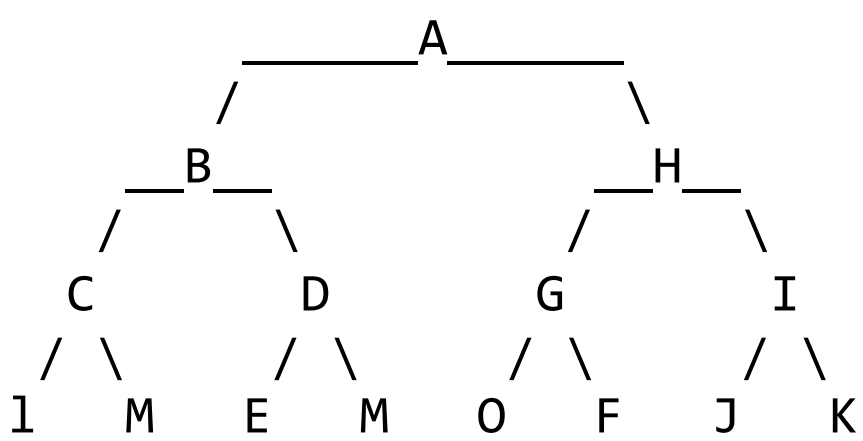


balance tree

```
((root->left)->left)->left = new Node('l');
((root->left)->left)->right = new Node('M');
((root->left)->right)->right = new Node('M');
((root->right)->left)->left = new Node('O');
((root->right)->right) = new Node('I');
((root->right)->right)->left = new Node('J');
((root->right)->right)->right = new Node('K');

cout << endl;
print_tree(root);
```

Result:



Access node

- access direct from root (only way to accesss)

- เข้าถึงได้โดยตรงจาก root (และเป็นทางเดียวเท่านั้น)

```
cout << root->data << endl;
cout << (root->left)->data << endl;
cout << ((root->left)->left)->data << endl;
cout << (((root->left)->left)->left)->data << endl;
cout << (((root->left)->left)->right)->data << endl;
```

Result :

A

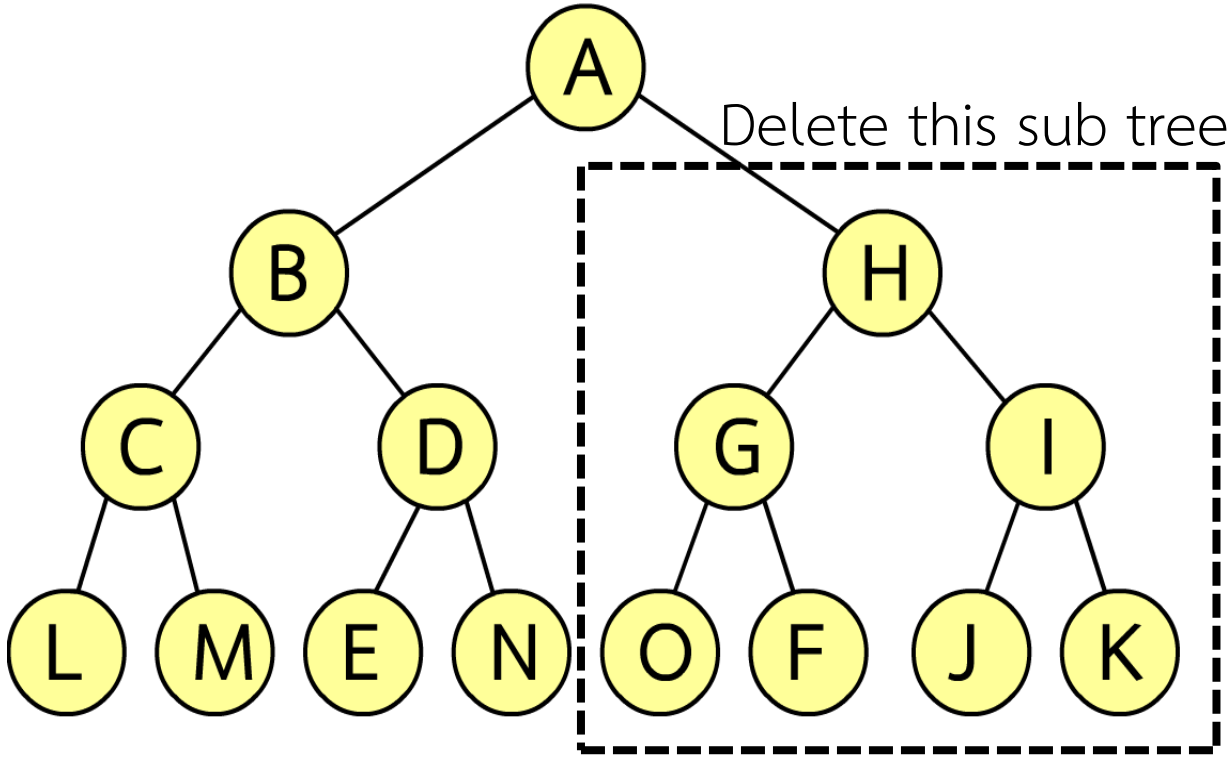
B

C

l

M

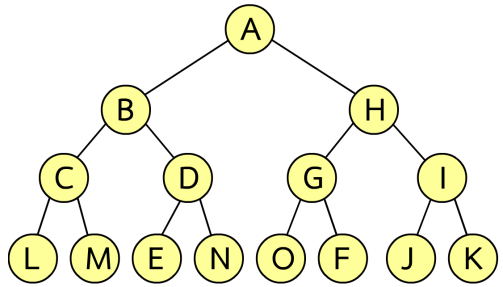
Delete subtree (if not care memory leak)




```
root->right = nullptr;
print_tree(root);
```

```
Result :
          ______A
         /
      __B__
     /      \
    C        D
   / \      / \
  l   M    E   M
```

Delete subtree (if not care memory leak)



- Try delete ‘L’ and ‘D’

```
((root->left)->left)->left = nullptr;
print_tree(root);
(root->left)->right = nullptr;
print_tree(root);
```

Result :

```
      ______A
     /
  __B__
 /     \
C       D
 \     / \
  M   E   M
```

```
      ______A
     /
  __B
 /
C
 \
  M
```

Happy? (quiz)

End of Binary tree how about delete just node

Delete node?

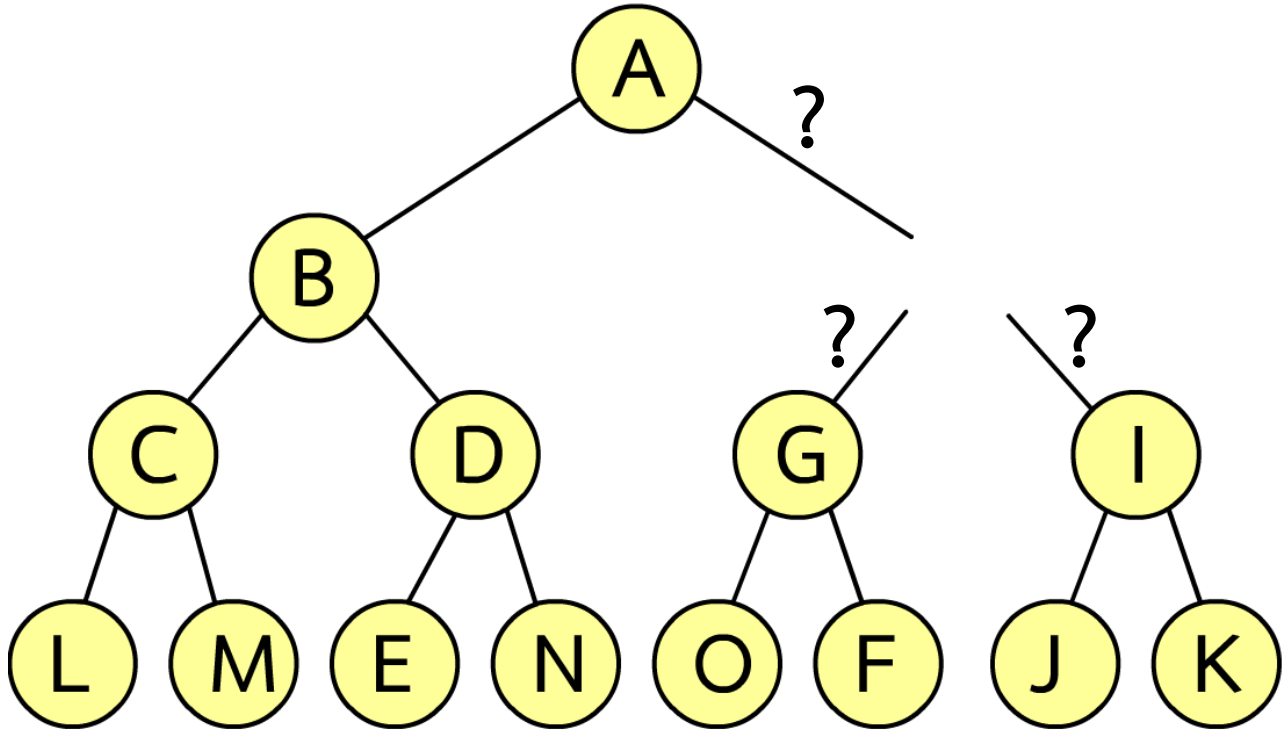
- to delete just a node and remain the rest of structure, we must manage left and right to replace with deleted node

- การ delete เพียงแค่ node และยังคง structure ที่เหลือไว้ เราต้องจัดการ (manage) node left และ node right ให้มาแทนที่ node ที่ถูกลบออกไป

Why?

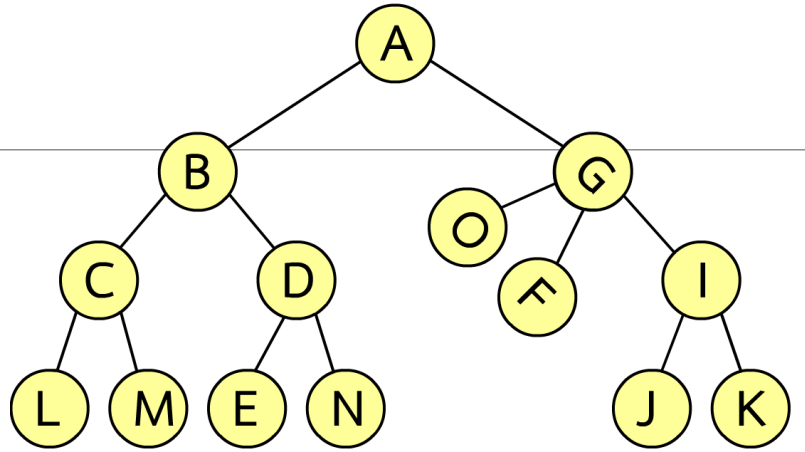
- To remain binary tree properties, we can’t just delete and merge like this

- เพื่อคงไว้ซึ่งสมบัติของ binary tree เราไม่สามารถแค่ลบออกไปเฉยๆ แบบนี้ได้

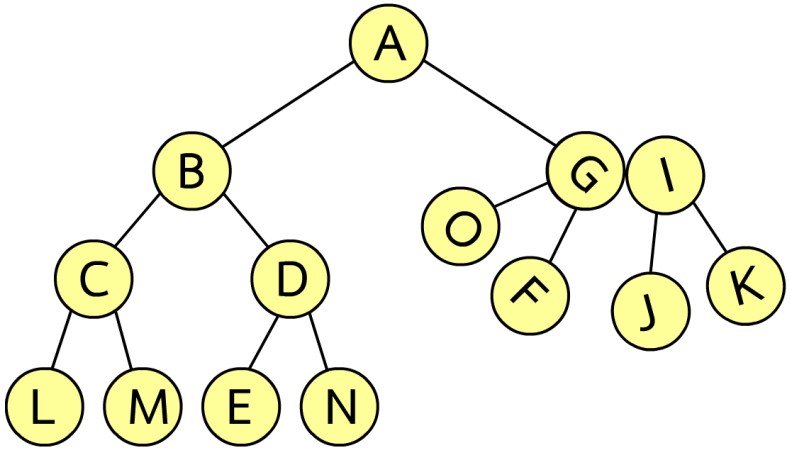


Unusable / ไม่สามารถนำไปใช้ได้

- Without well manage we will get unlikely good struct like this



- หากจัดการการ delete ไม่ดีเรา จะได้ structure ที่ไม่ค่อยดีแบบนี้

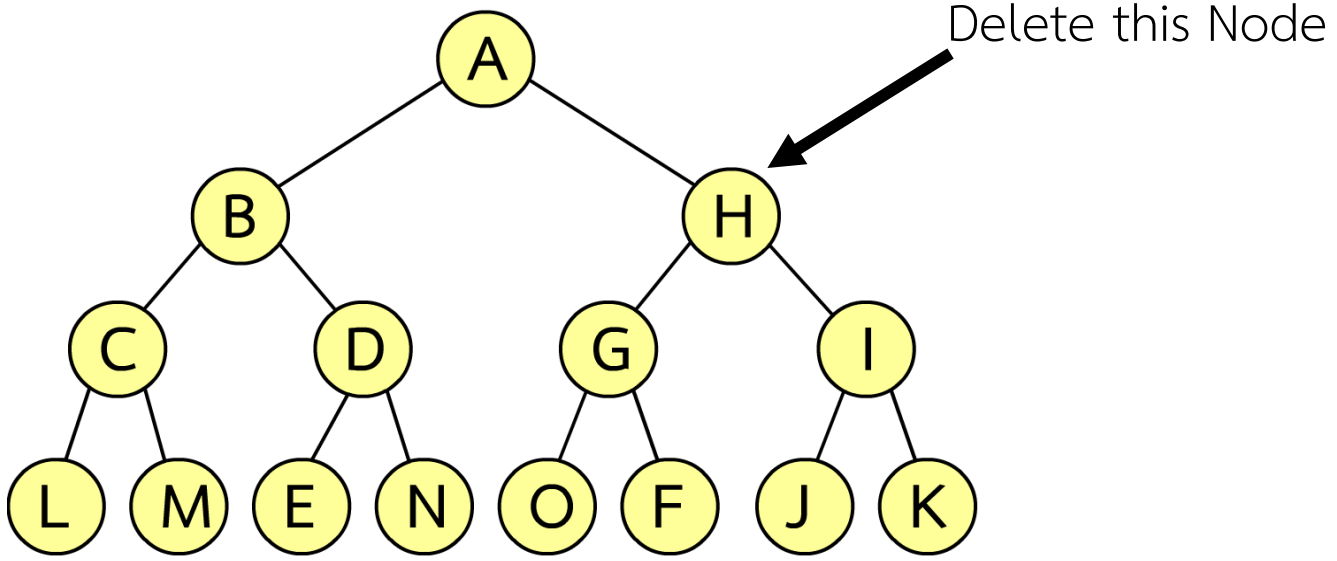


How to manage

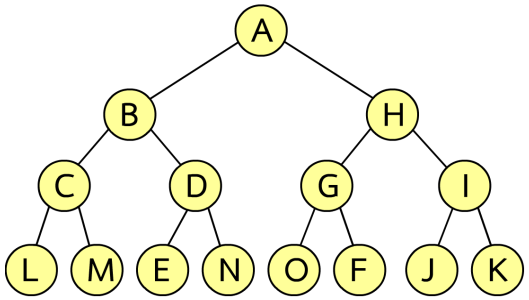
- replace left or right node with deleted node

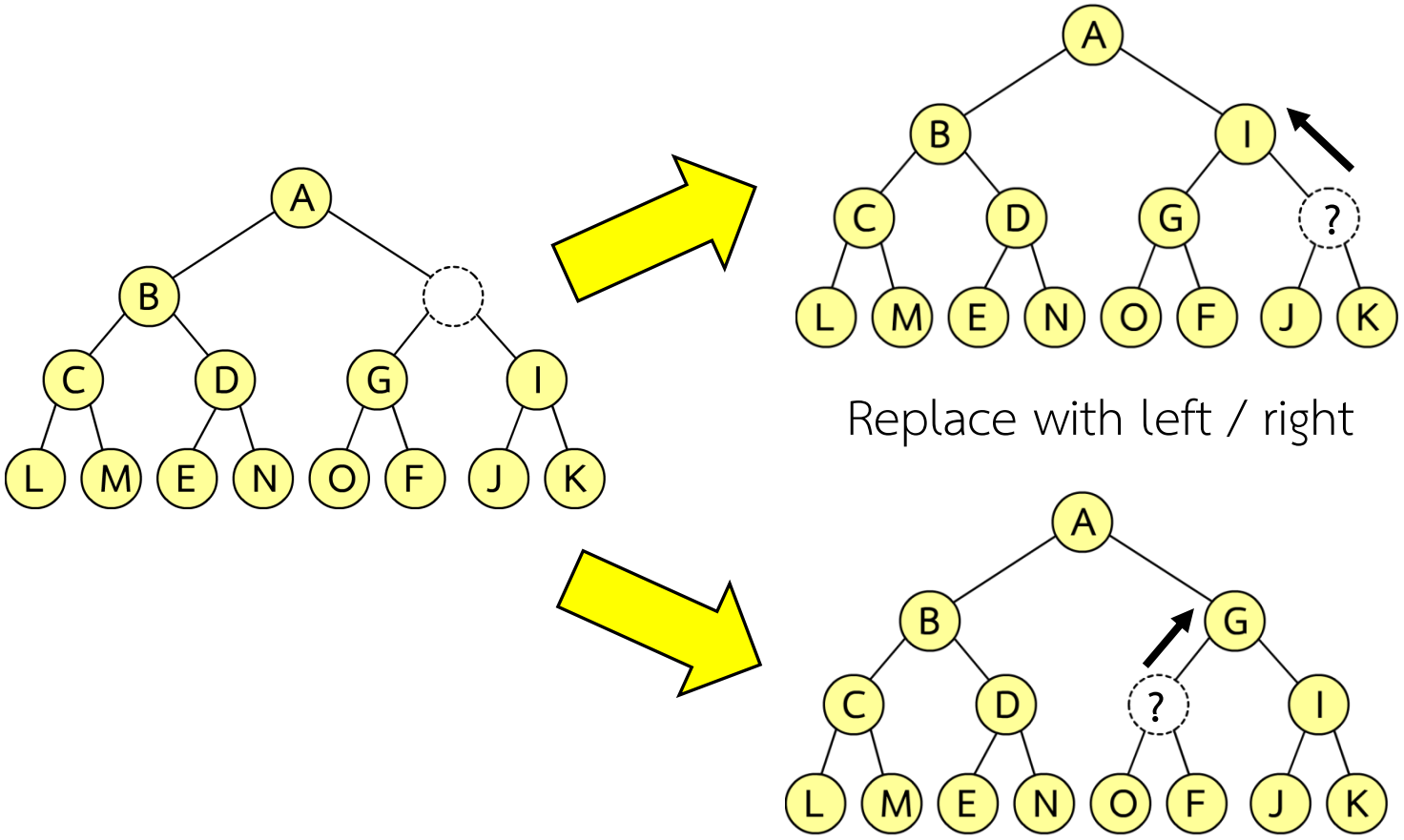
- แทนที่ node ที่ถูกลบไปด้วย node left หรือ node right

Example : Delete Node H



Problem ?





- Not just 1 node, We must Replace until reach left node

- ไม่ใช่แค่ node เดียว แต่เราต้อง replace จนกระทั้งไปถึง left node

Delete_left_first and Delete_right_first

- scene we can’t just delete note there are 2 type of deletion that shift up child to be a node : left first deletion and right first deletion

- ในเมื่อเราไม่สามารถแค่ delete node ออกไปเฉยๆ ได้เราจึงต้องมีคำสั่งที่ replace node ที่โดน delete ด้วย node ลูกได้แก่ left first deletion และ right first deletion

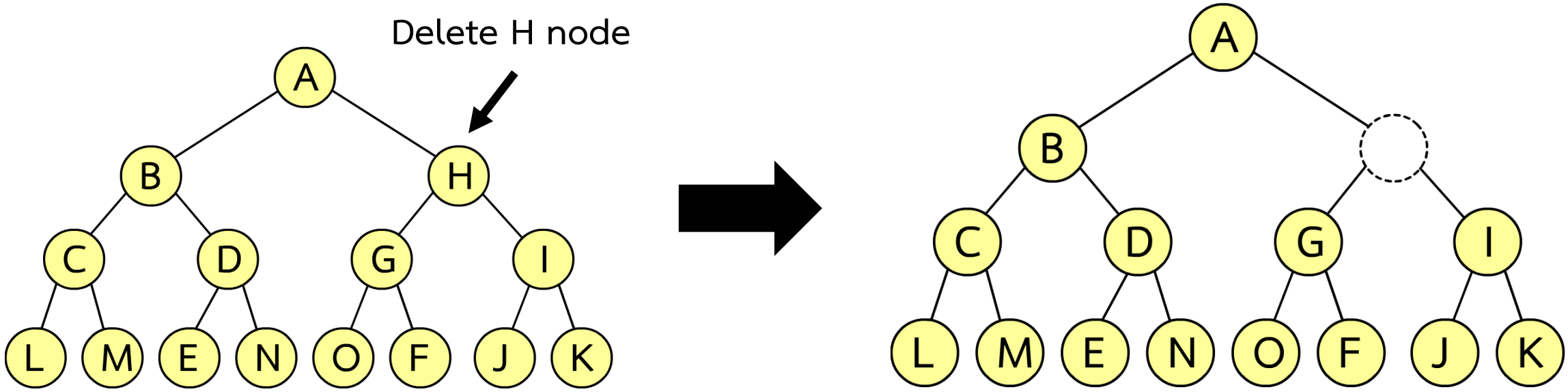
Delete_left_first(node)

- replace deleted node with left child node
- if left child node is null replace with right child node
- if both of children are null just delete node
- repeat this process for selected child node until found null

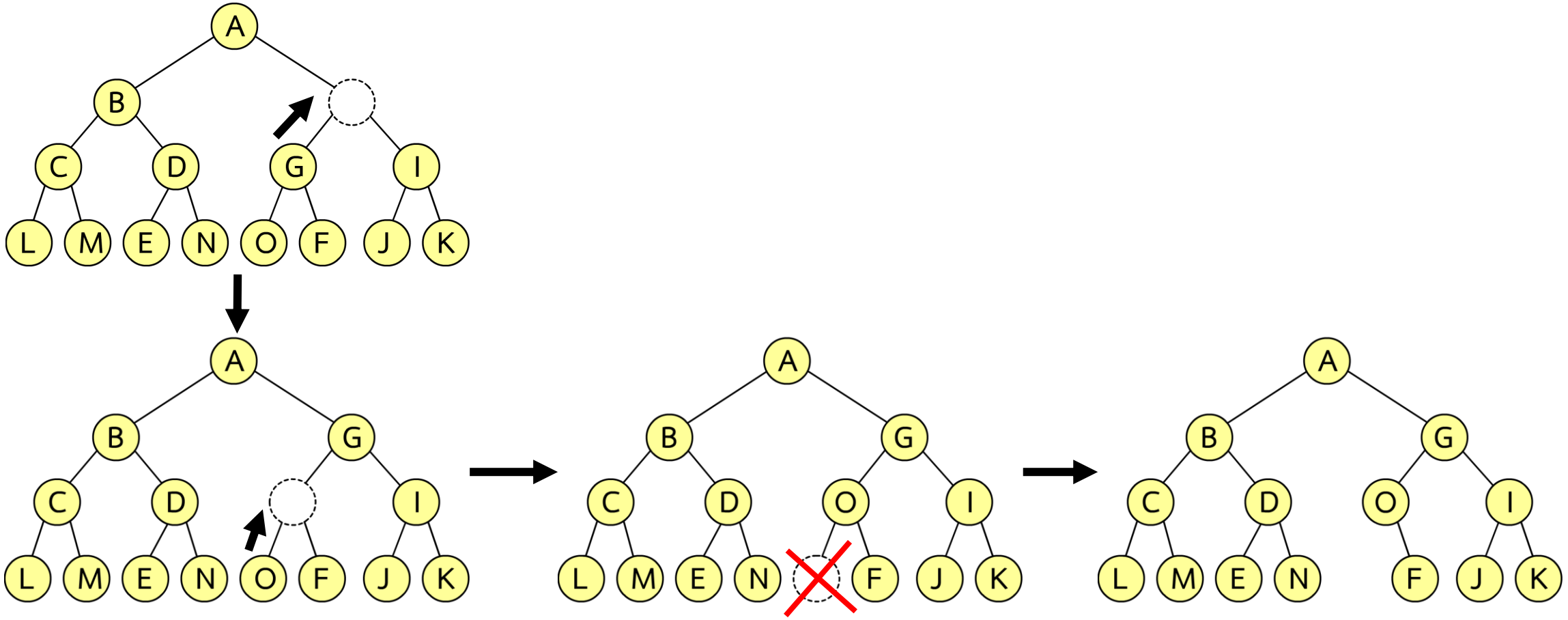
Delete_left_first(node)

- เปลี่ยน node ที่โดนลบด้วย left child
- ถ้า left child เป็น null ให้ใช้ right child แทน
- หากเป็น null ทั้ง 2 ข้างให้ลบ node นั้นออกได้เลย
- ทำซ้ำใน child node ที่ถูกเลือกจนกระทั้งพบ null

Delete_left_first example



Replace

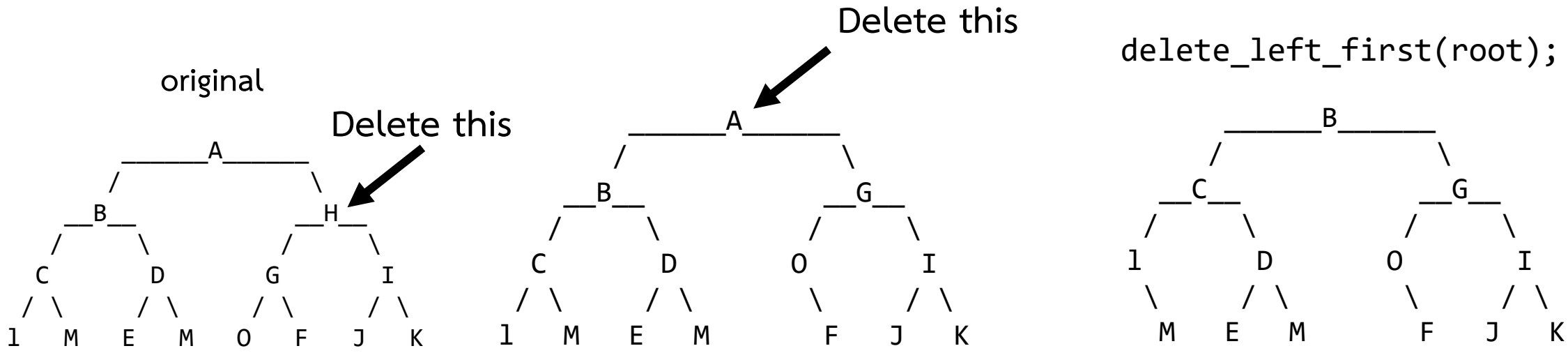


code

```
void delete_left_first(Node *target,Node *parent,bool from_left){
    while(target != nullptr){
        if(target->left != nullptr){ // replace with left ndoe
            target->data = (target->left)->data;
            parent = target;
            target = target->left;
            from_left = true;
        }
        else if(target->right != nullptr){ // replace with right ndoe
            target->data = (target->right)->data;
            parent = target;
            target = target->right;
            from_left = false;
        }
        else{ // delete node
            if(from_left == true) parent->left = nullptr;
            else parent->right = nullptr;
            target = nullptr;
        }
    }
}
```

example

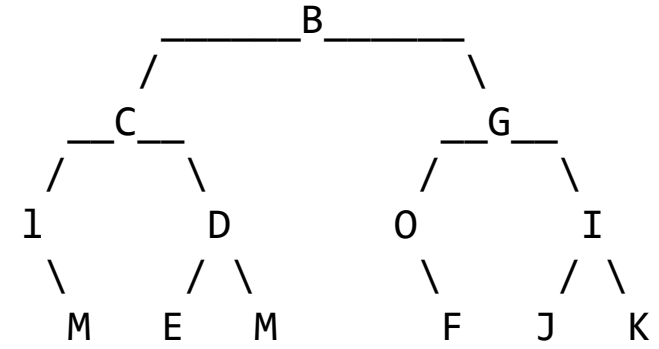
```
delete_left_first(root->right,root,false);
```

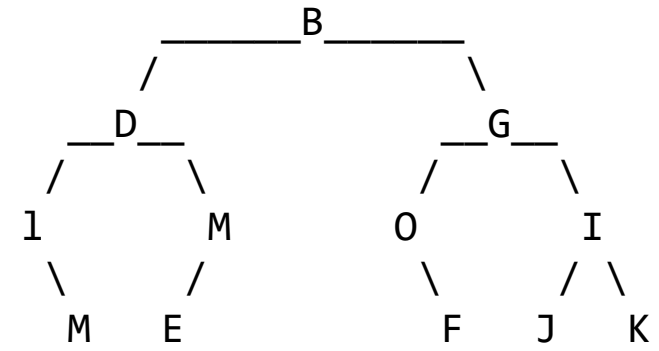


Delete_right_first(node) same as left

- same as left first but change priority to right first

- เหมือนกันกับ left first แต่เปลี่ยนความสำคัญ (piority) เป็นด้านขวาก่อน



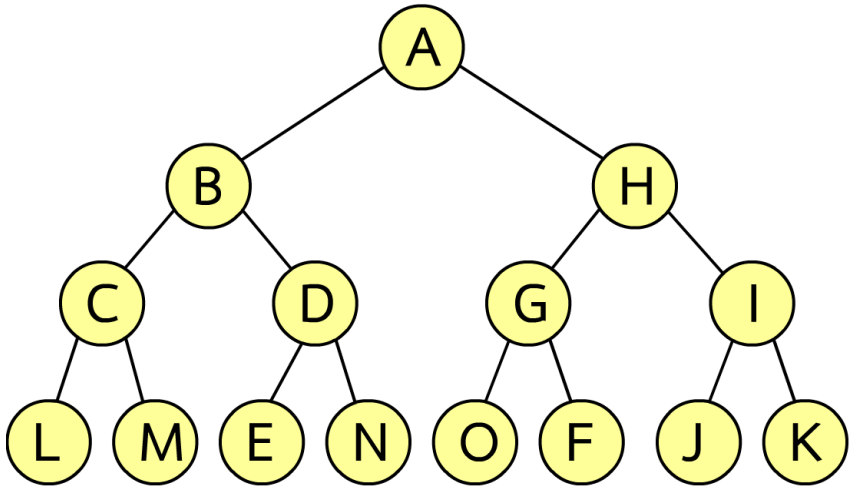



```
delete_right_first(root->left);

cout << endl;
print_tree(root);
```

Traversal

- There are several way to one-way traverse (discovery) in tree (unlike linear structure such as linked list or array)



- มากกว่า 1 วิธีในการท่องไปใน structure แบบ tree (ไม่เหมือนใน linear structure เช่น linked list or array)

Traversal (example)

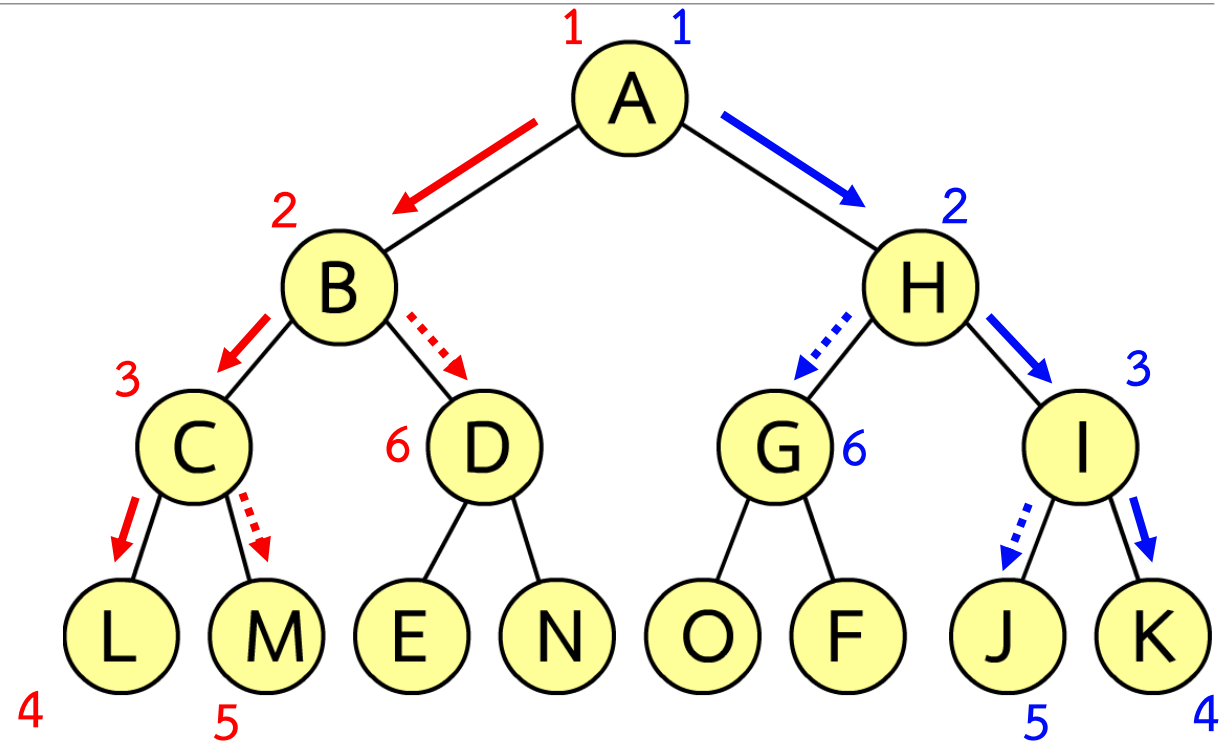
Left first traverse

or

Right first traverse

Primary rule

Secondly rule



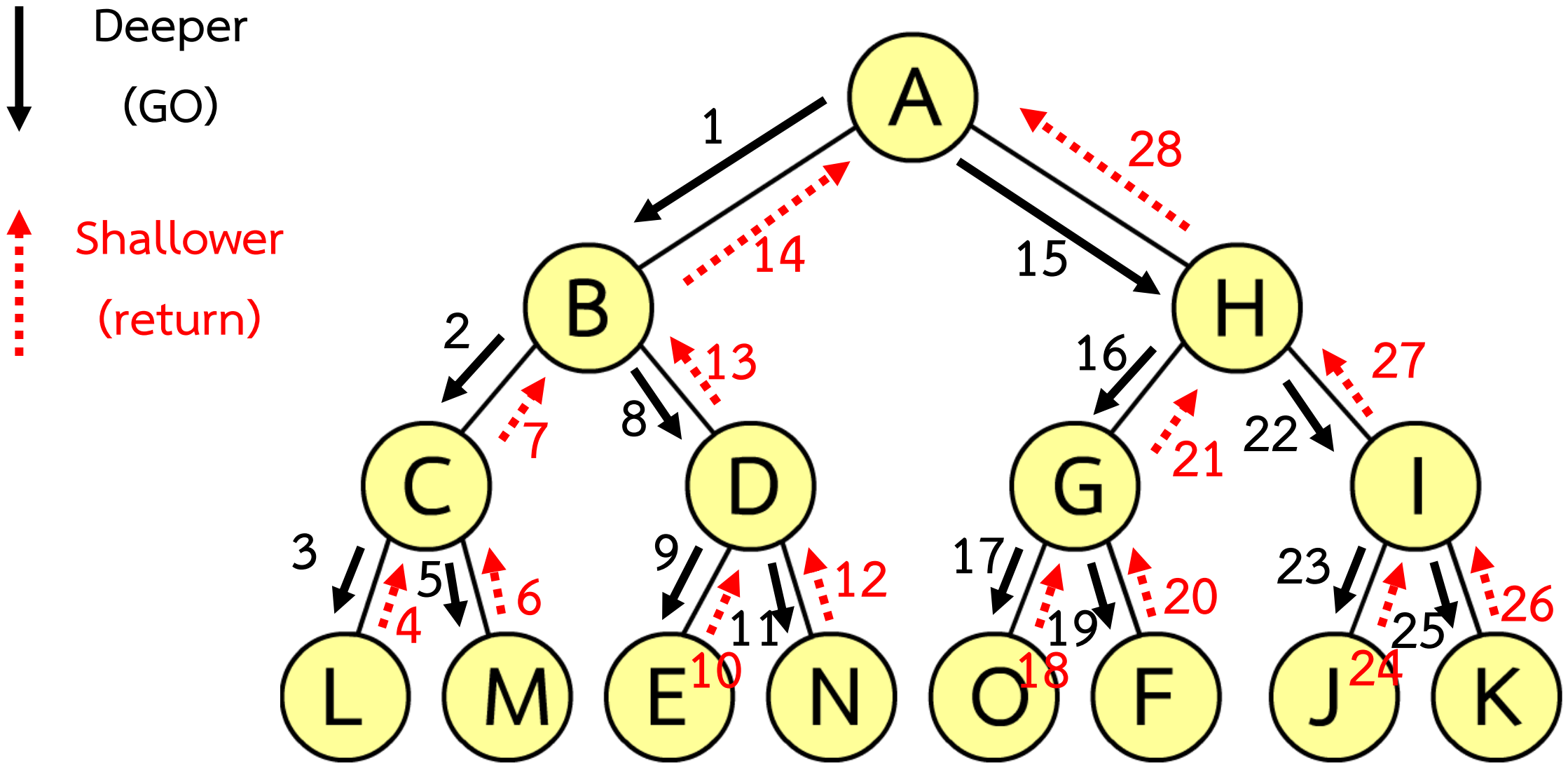
A,B,C,L,M,D,E,N,H,G,O,F,I,J,K

A->

Common traversal (recursively name)

If we want to travel to all node from root to leaf from left to right, it is only one path (left most recursively path)

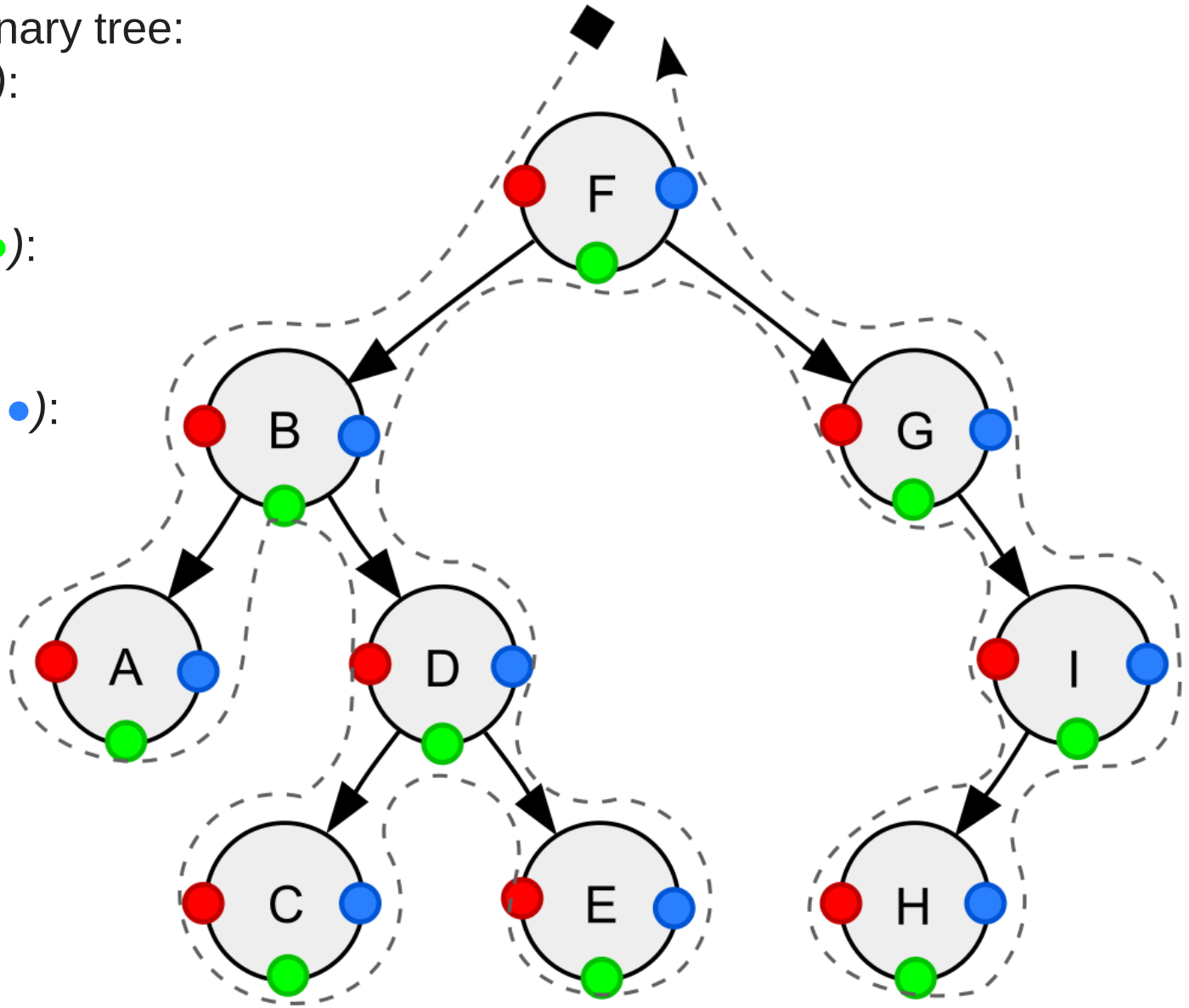
หากเราต้องการสำรวจทุก node จาก root ไปยัง leaf จากซ้ายไปขวา จะมีเส้นทางเดียวเท่านั้น (left most recursively path)


Deeper
(GO)
Shallower
(return)
A
B
H
C
D
G
I
L
M
E
N
O
F
J
K
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

Depth-first traversal (dotted path) of a binary tree:
Pre-order (node visited at position red ●):
F, B, A, D, C, E, G, I, H;

In-order (node visited at position green ●):
A, B, C, D, E, F, G, H, I;

Post-order (node visited at position blue ●):
A, C, E, D, B, H, I, G, F.



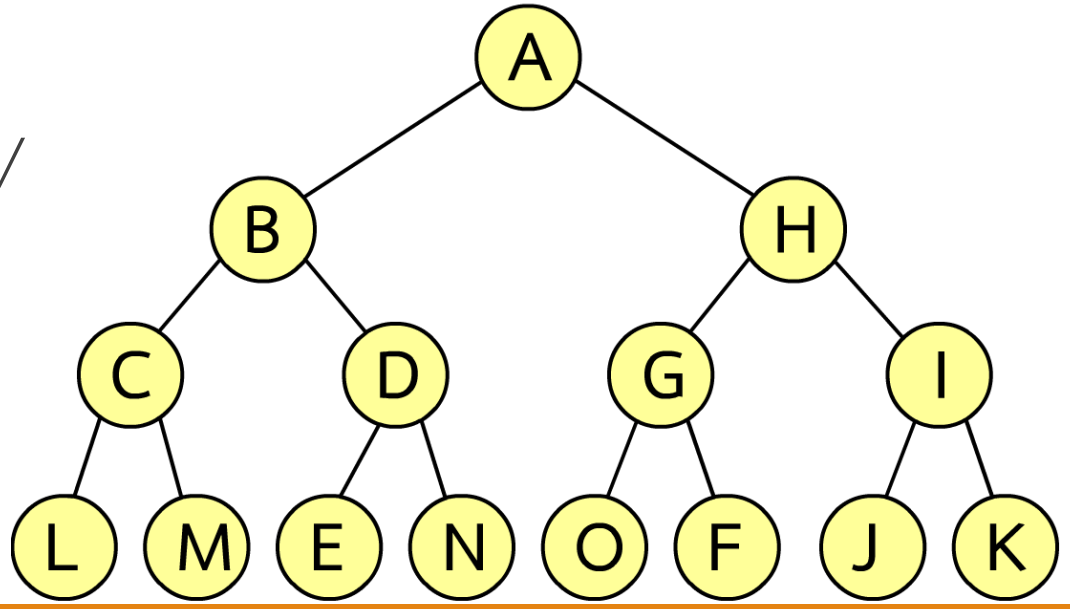
https://en.wikipedia.org/wiki/Tree_traversal#/media/File:Sorted_binary_tree_ALL_RGB.svg

Common traversal (recursively name)

Preorder -> visit when first found / สำรวจเมื่อเจอครั้งแรก

Inorder -> visit when visited all left child node / สำรวจเมื่อสำรวจทุก left child node

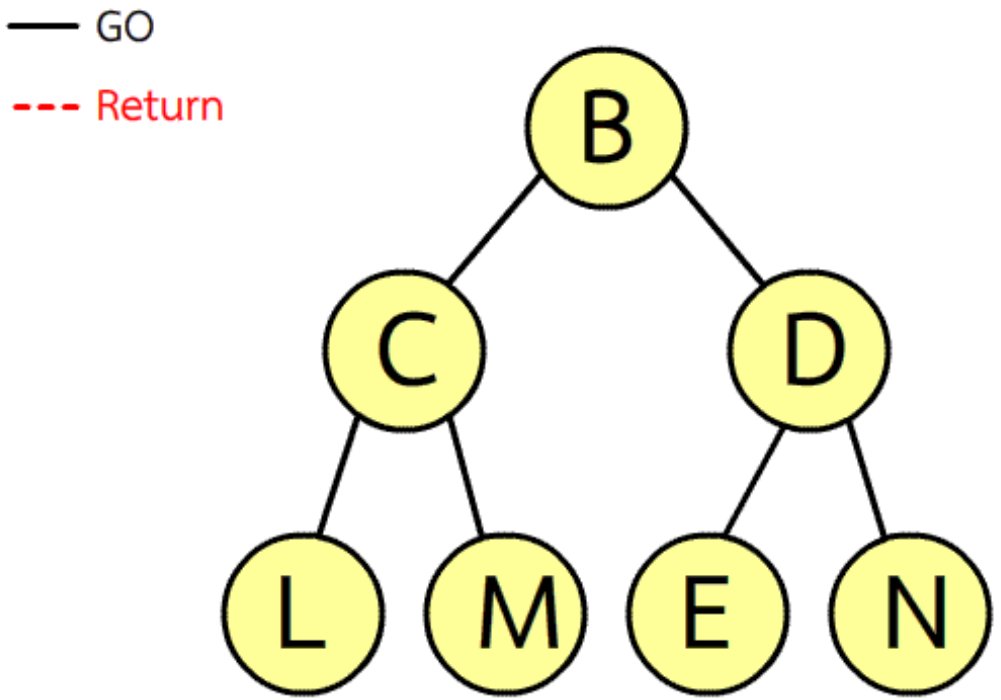
Postorder -> visit when leave node / สำรวจเมื่อออกจาก node



Preorder (visit when first found)

```
void print_preorder(Node* p){
    if(p == nullptr) return;

    cout << p->data << " ";
    print_preorder(p->left);
    print_preorder(p->right);
}
```

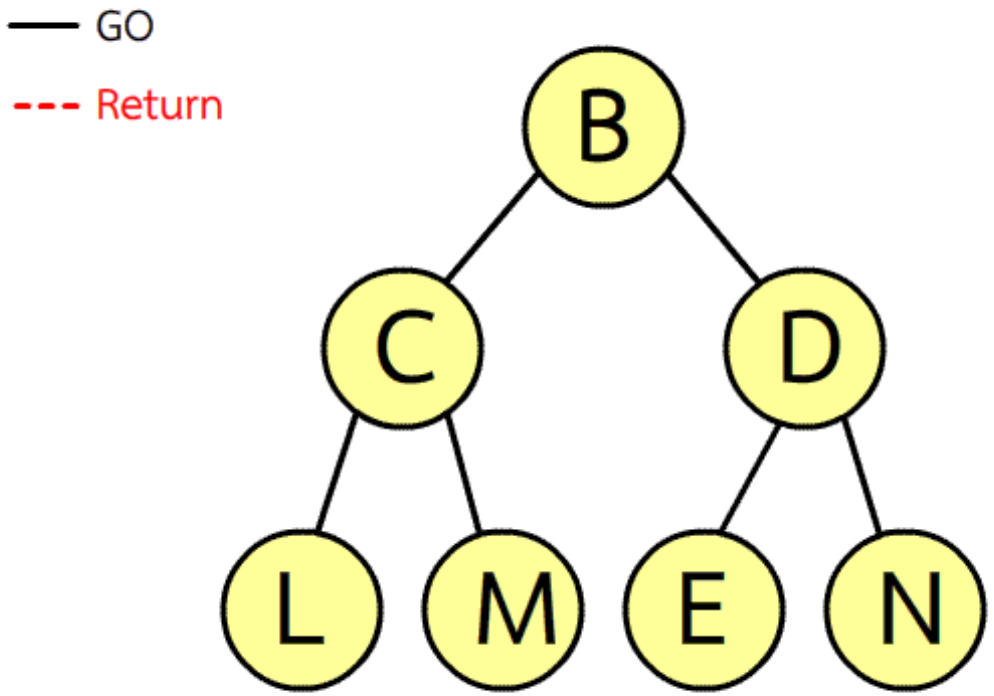


Output :

Inorder (visit when visited all left child node)

```
void print_inorder(Node* p){
    if(p == nullptr) return;

    print_inorder(p->left);
    cout << p->data << " ";
    print_inorder(p->right);
}
```

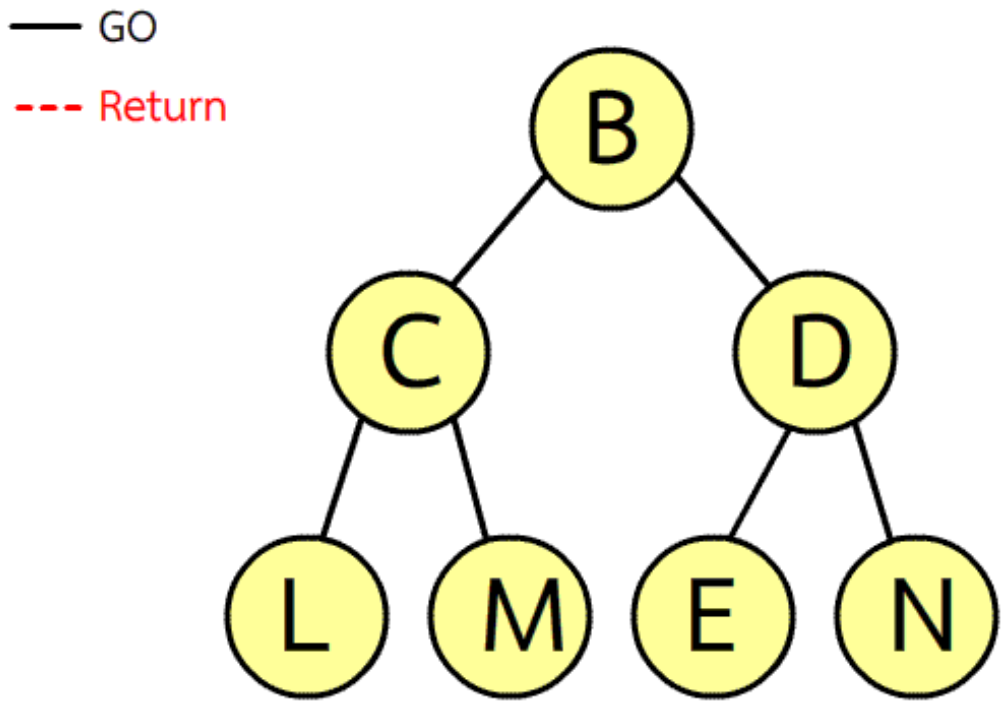


Output :

Postorder (visit when leave node)

```
void print_postorder(Node* p){
    if(p == nullptr) return;

    print_postorder(p->left);
    print_postorder(p->right);
    cout << p->data << " ";
}
```

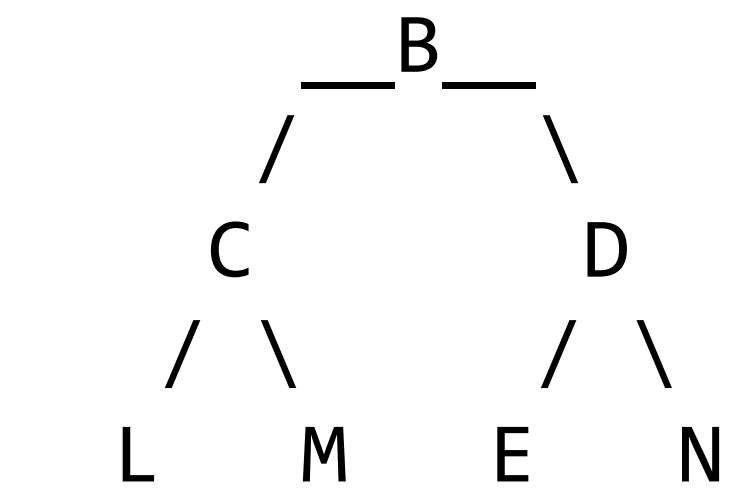


Output :

```
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');
(root->right)->left  = new Node('E');
(root->right)->right = new Node('N');

cout << endl;
print_tree(root);
print_preorder(root); cout << endl;
print_inorder(root); cout << endl;
print_postorder(root); cout << endl;
```

Result :




```
B C L M D E N
L C M B E D N
L M C E N D B
```

Big O

Can be anything because no insert or order rule

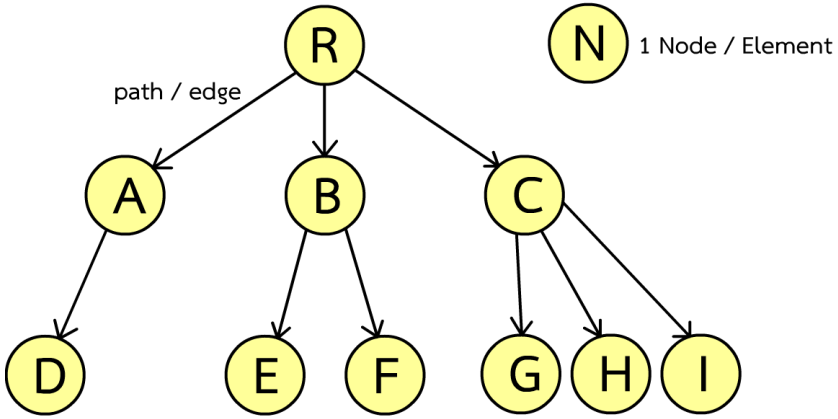
We don’t care!

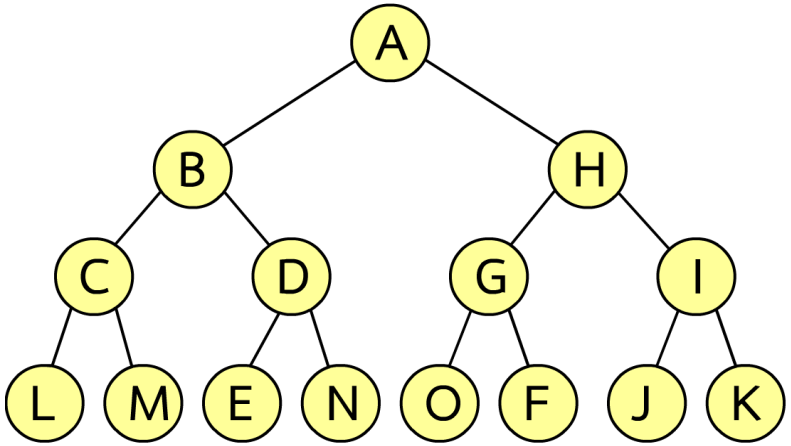
Easy Easy tree

Purpose of tree

- unconditioned tree rarely used

- conditioned tree are **power full**





Conditioned tree

- BTS binary tree search <- in this class
- AVL tree (self balance tree)
- Red Black Tree
- B – tree
- Segment Tree

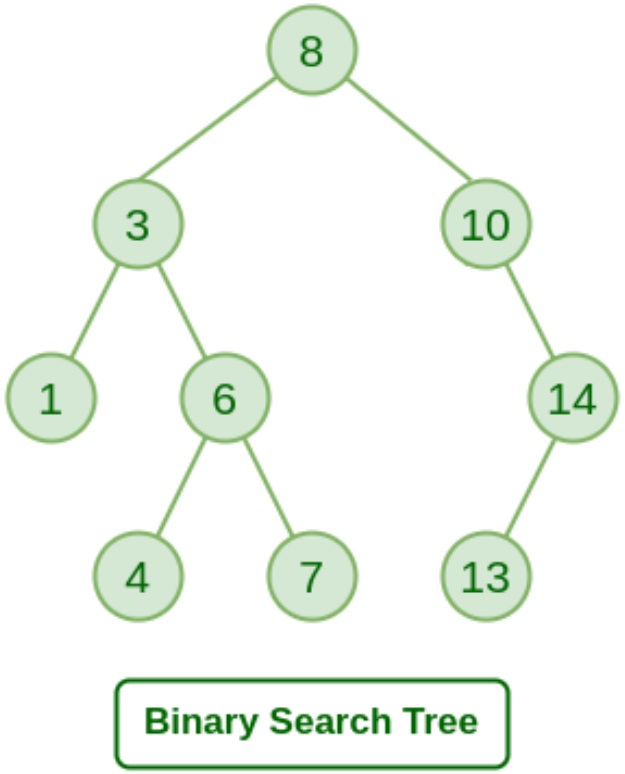
Binary Tree Search

Binary Search Tree is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with keys lesser 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.

https://www.geeksforgeeks.org/types-of-binary-tree/

Binary Tree Search



https://www.geeksforgeeks.org/types-of-binary-tree/

LAB