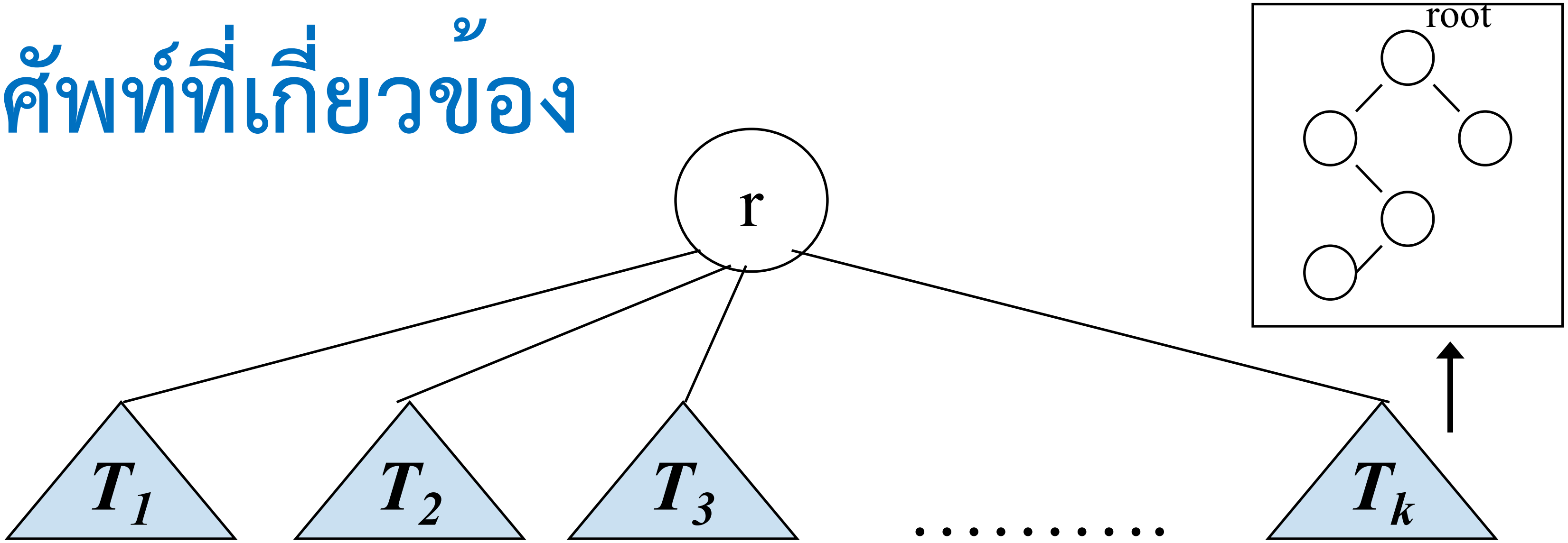
ต้นไม้ (Tree)

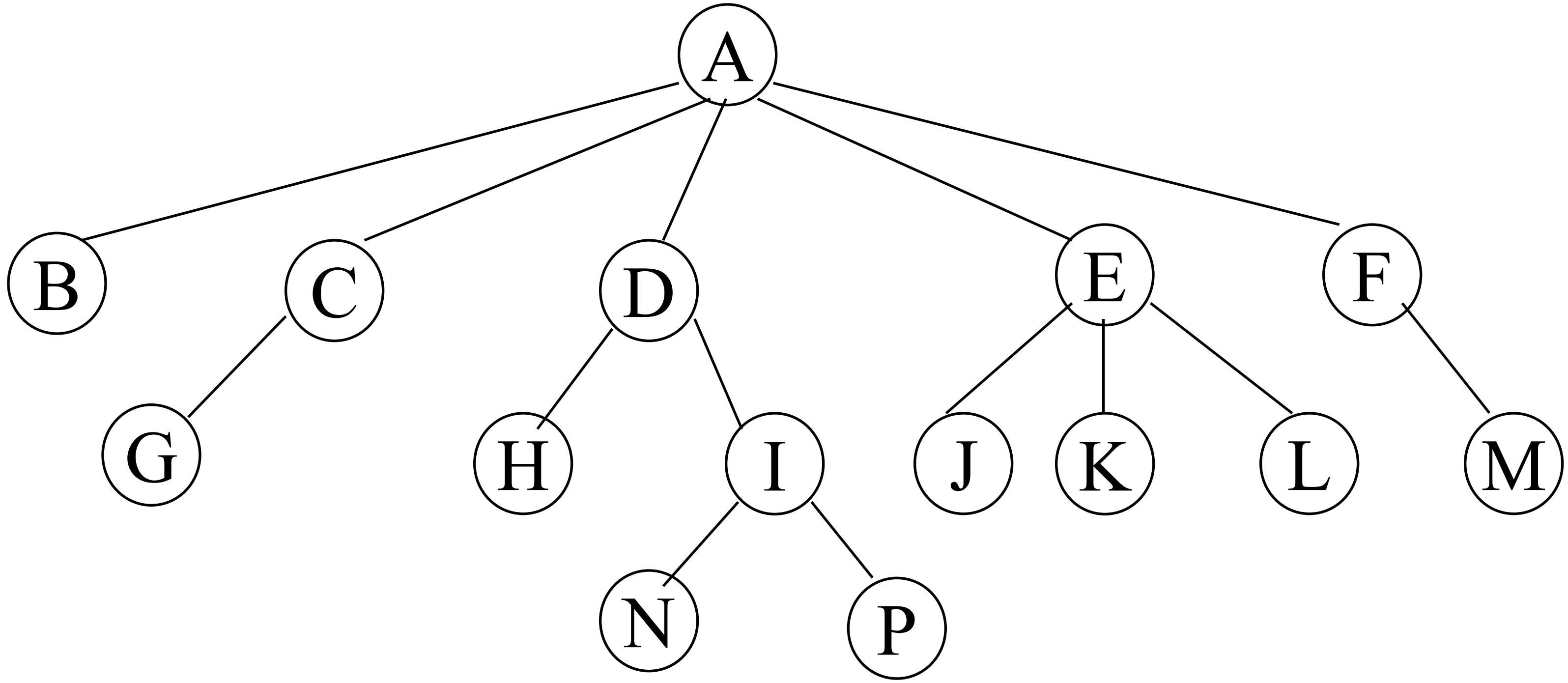
ต้นไม้ (Tree)

- โครงสร้างต้นไม้เป็นโครงสร้างที่ประกอบไปด้วยโหนดที่เก็บข้อมูล
- ต้นไม้อาจจะเป็น ต้นไม้ว่าง คือไม่มีข้อมูลอยู่เลยหรือ
- ประกอบไปด้วยโหนดที่เป็นรากของต้นไม้หรือ root node และต้นไม้ย่อย (subtrees) ซึ่งต้นไม้ย่อยก็มีคุณสมบัติเช่นเดียวกับต้นไม้ใหญ่ คือเป็นต้นไม้ย่อยที่ว่างหรือเป็นต้นไม้ย่อยที่มีโหนดรากและต้นไม้ย่อยลงไปอีก

ศัพท์ที่เกี่ยวข้อง

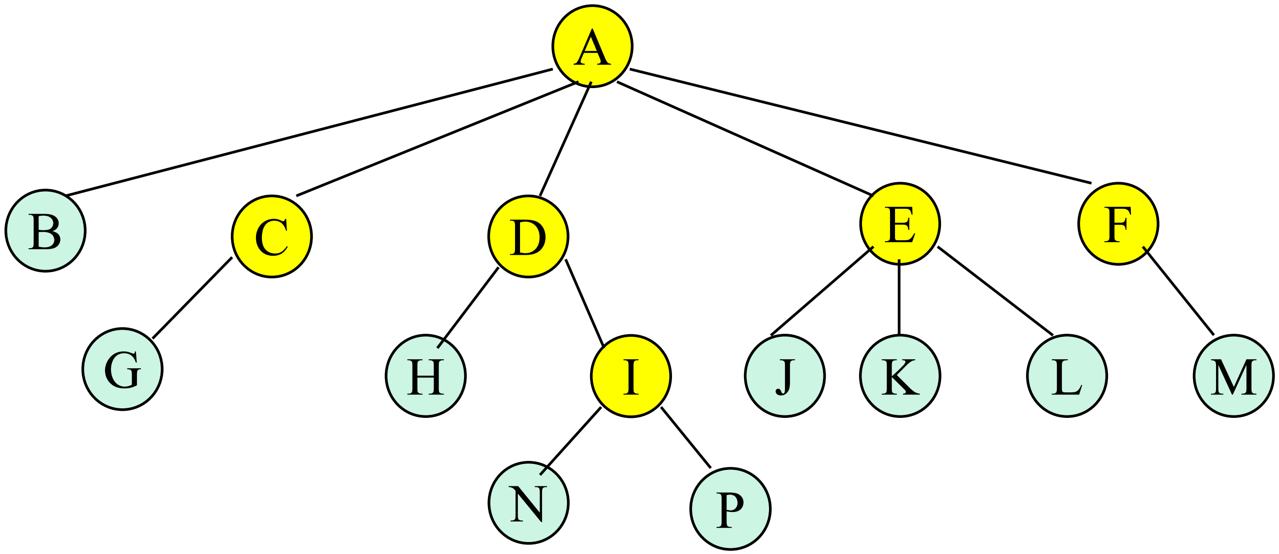


- โหนด *r* เป็น root node ของต้นไม้
- root node ของทุกๆต้นไม้ย่อยถือเป็น children ของโหนด *r* หรือ โหนด *r* เป็น parent ของ root nodes ของต้นไม้ย่อย
- ทุกๆ โหนดในต้นไม้ยกเว้น root node มีโหนด parent 1 โหนด
- สามารถกำหนดความสัมพันธ์ของโหนดอื่นๆ เช่นเดียวกับความสัมพันธ์ของครอบครัวคือ คือ โหนดลูก (child) โหนดหลาน (grandchild) โหนดปู่และทวด (grandparent)



- จากรูป โหนด A เป็น root node และเป็น parent ของโหนด B, C, D, E และ F
- โหนด J, K, L เป็น children ของโหนด E
- โหนดที่ไม่มีลูกเลยเรียกว่าโหนดใบไม้ (leaves) ได้แก่โหนด B, G, H, N, P, J, K, L และ M
- โหนดที่มี parent เดียวกันเป็นพี่น้องกัน (siblings) เช่น โหนด J, K, L

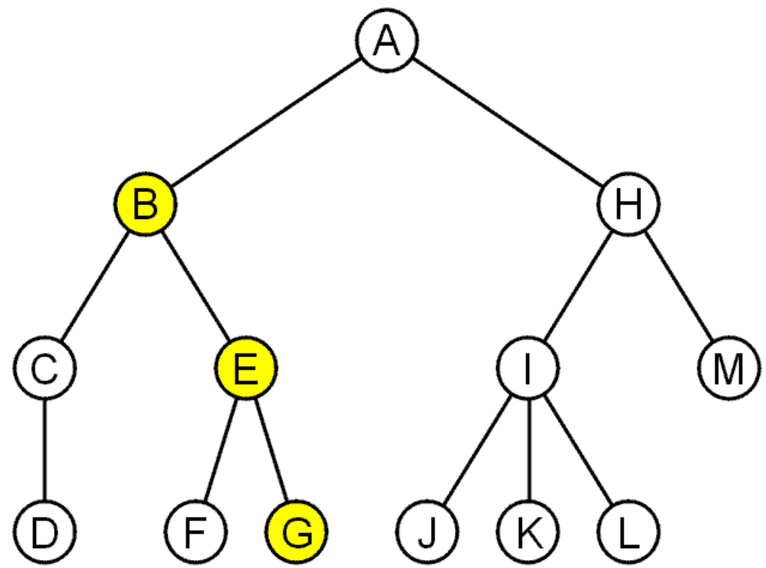
ศัพท์ที่เกี่ยวข้อง



- Degree ของโหนดคือ จำนวนลูกของโหนดนั้น เช่น degree(E) = 3
- โหนดที่มี degree เป็น 0 คือ leaf nodes
- โหนดอื่นๆ ถือเป็น internal nodes หรือโหนดที่อยู่ด้านในของต้นไม้

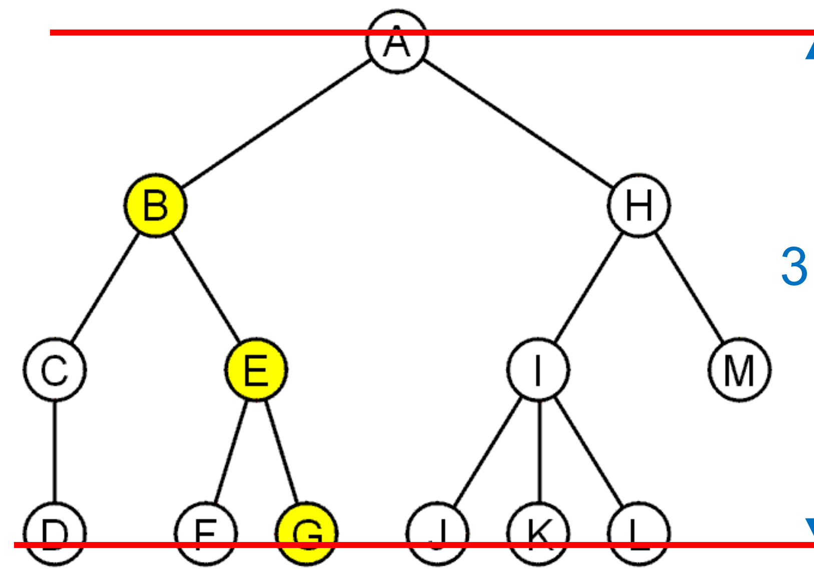
ศัพท์ที่เกี่ยวข้อง

- Path เป็นลำดับของโหนด ($a_0, a_1, \ldots, a_n$) โดย a_{k+1} เป็นลูกของโหนด a_k
- **ความยาว (length) ของ path** คือจำนวนเส้นที่เชื่อมโหนด เช่น path (B, E, G) มีความยาว 2
- แต่ละโหนดในต้นไม้ จะต้องมี path จาก root ไปที่โหนดนั้นเสมอ
- **ความลึก (depth)** ของโหนดจะเท่ากับ length ของ path จาก root ไปที่โหนดนั้น
  - E มีความลึก 2
  - L มีความลึก 3



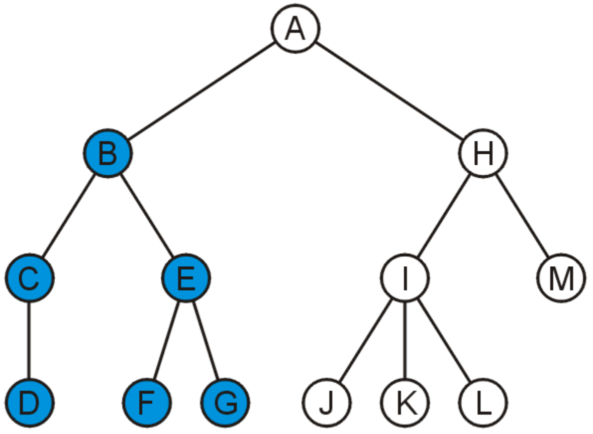
ศัพท์ที่เกี่ยวข้อง

- ความสูง (height) ของต้นไม้ คือ ความลึกสูงสุดของความลึกของทุกๆ โหนดในต้นไม้
- ความสูงของต้นไม้ที่มีหนึ่งโหนดคือ 0 (มีแค่ root node)
- ความสูงของต้นไม้ว่างคือ -1

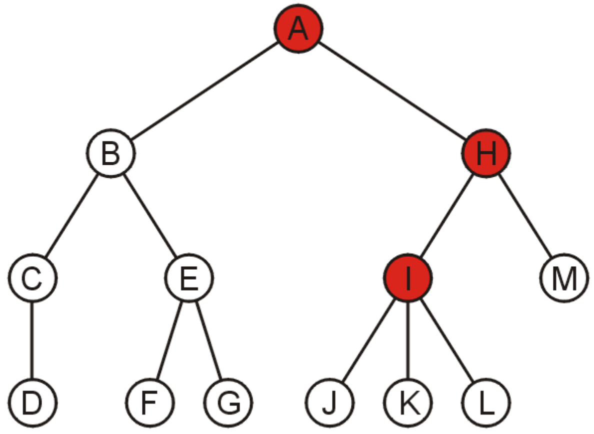


ศัพท์ที่เกี่ยวข้อง

- ถ้ามี path จากโหนด *a* ไป โหนด *b* แล้ว
  - โหนด *a* เป็น ancester ของ โหนด *b* และ โหนด *b* เป็น descendant ของโหนด *a*
- ดังนั้นโหนดหนึ่งๆ จะเป็นทั้ง ancester และ descendant ของตัวเอง
- root เป็น ancester ของทุกโหนด



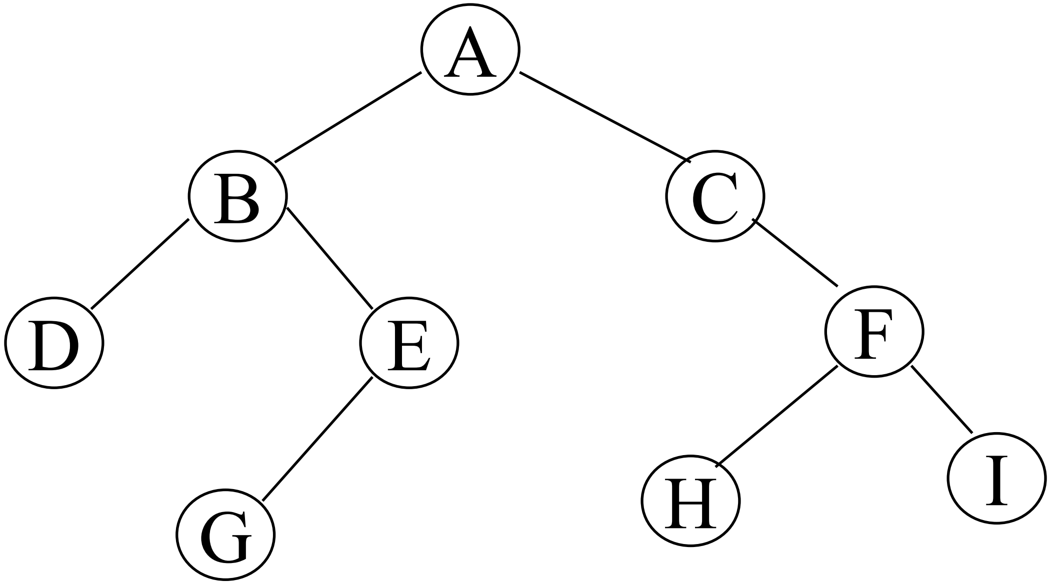
Descendants –ของโหนด B คือ B, C, D, E, F, และ G



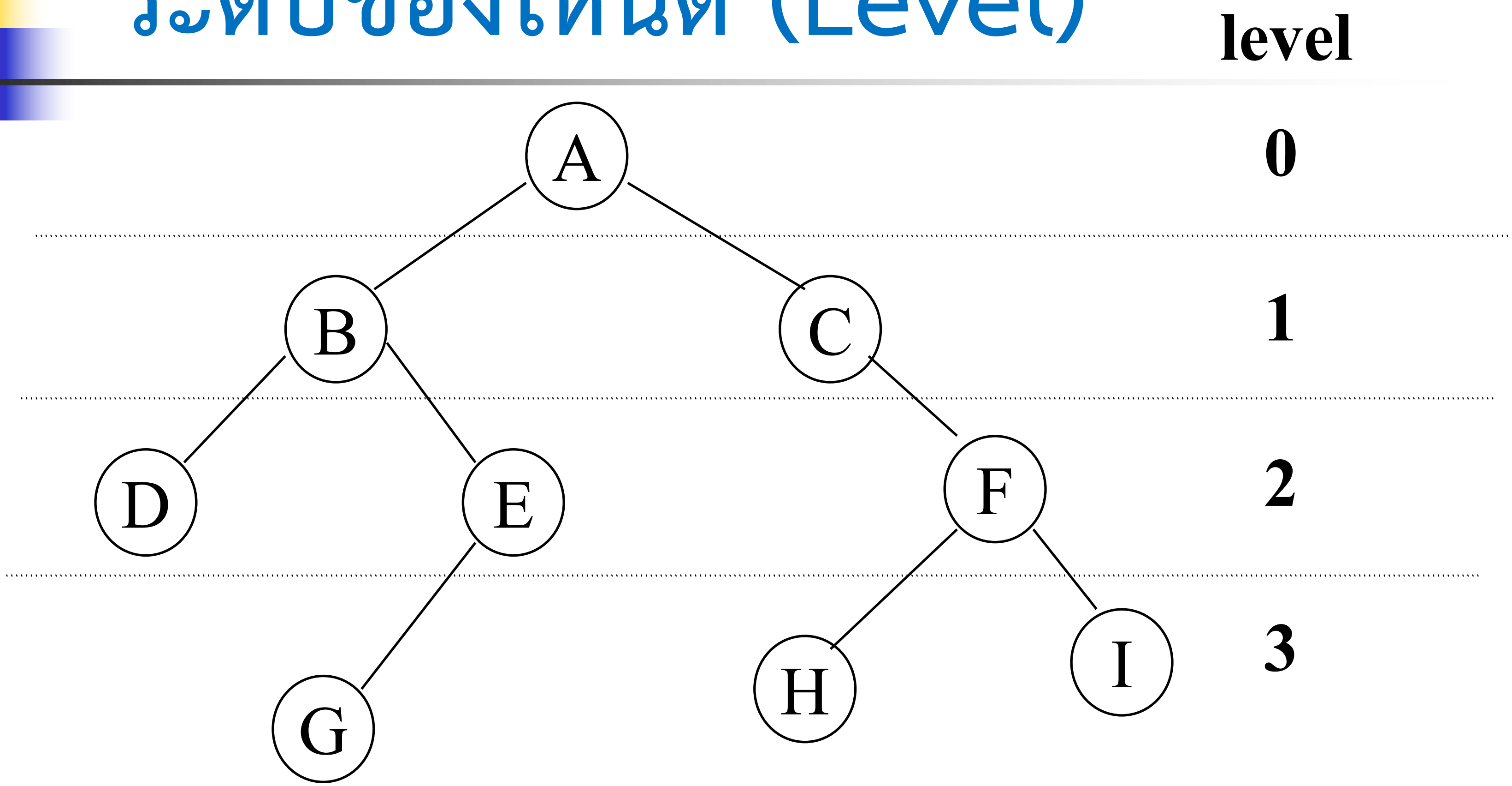
Ancestorsของโหนด I คือ I, H, และ A

ต้นไม้ไบนารี (Binary Trees)

ต้นไม้ไบนารีคือ ต้นไม้ที่ทุกโหนดมลูกได้ไม่เกิน 2 โหนด อาจไม่มีลูกเลยก็ได้ หรือมีลูกหนึ่งโหนดหรือมีลูกสองโหนด



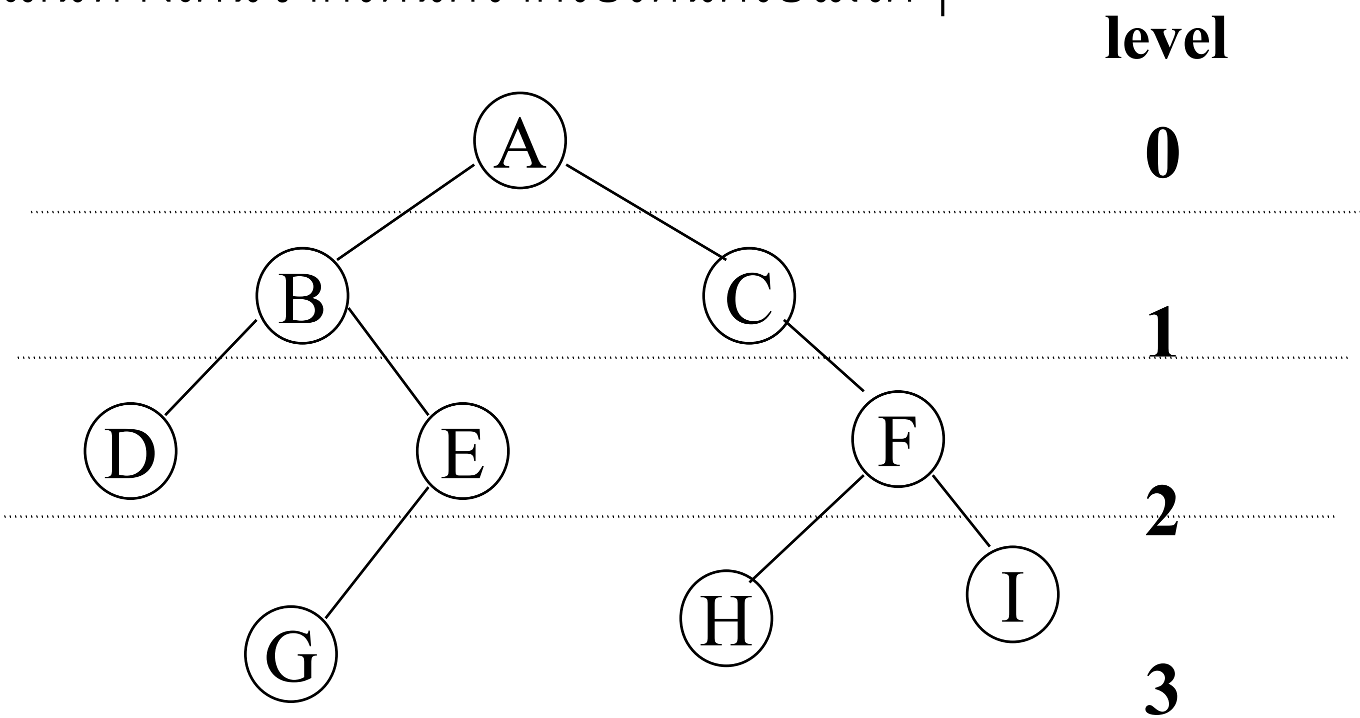
ระดับของโหนด (Level)



โหนดรากของต้นไม้อยู่ที่ระดับ 0 ระดับของโหนดอื่นๆ จะมีค่ามากกว่าค่าระดับของโหนดผู้ปกครองอยู่หนึ่งระดับ เช่น โหนด E อยู่ที่ระดับ 2 และโหนด H อยู่ที่ระดับ 3

ความลึก (Depth)

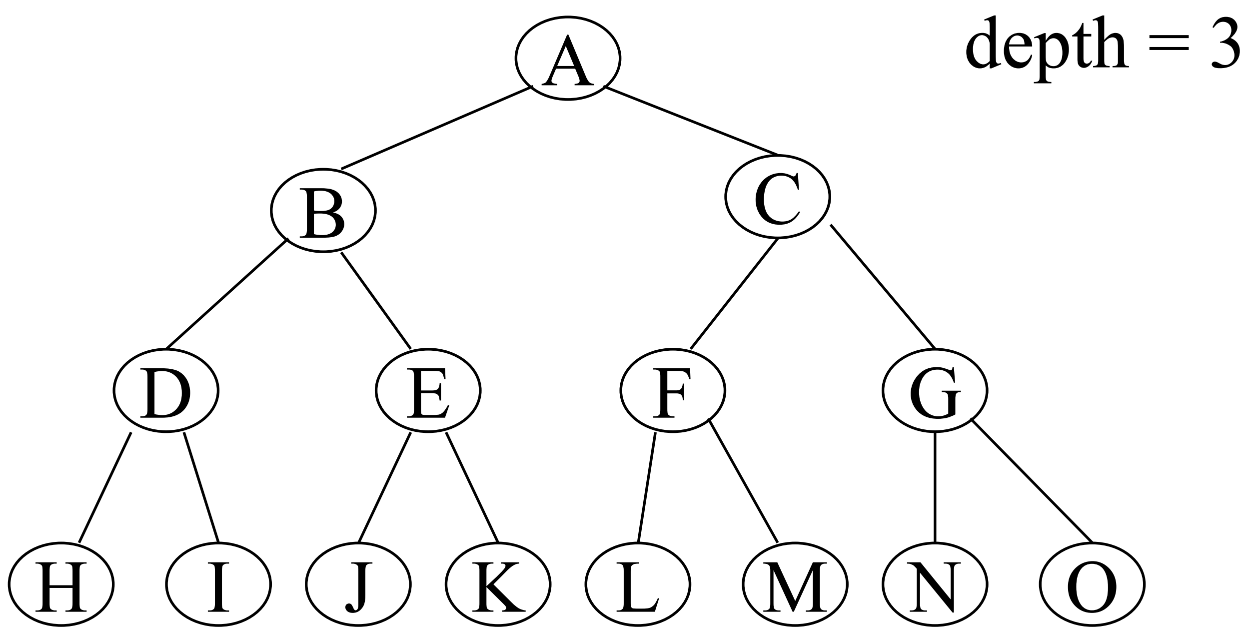
ความลึก คือค่าระดับที่สูงที่สุดของโหนดใบไม้ใดๆ ในต้นไม้ หรือความยาวสูงสุดของ เส้นทางเดินจากโหนดรากไปโหนดใบไม้ใดๆ



ต้นไม้นี้มีความลึก 3 ระดับ

ต้นไม้ไบนารีแบบสมบูรณ์

ต้นไม้ไบนารีแบบสมบูรณ์ (Complete binary tree) ที่มีความลึก d เป็นต้นไม้ไบนารีที่โหนดใบไม้ทุกๆ โหนดจะอยู่ที่ระดับเดียว กันคือ ระดับ d หรือระดับที่เป็นความลึกของต้นไม้



ต้นไม้ไบนารีมีโหนด m โหนด ที่ระดับ L จะมีจำนวนโหนดมากที่สุด 2m ที่ระดับ L+1 และเนื่องจากต้นไม้ไบนารีสามารถมีโหนดได้เพียงหนึ่งโหนดที่ระดับ 0 เราสามารถกล่าวได้ว่า ต้นไม้ต้นนี้จะมีโหนดได้มากที่สุด 2^L โหนดที่ระดับ L

จำนวนโหนดที่ระดับ 0 คือ $2^0 = 1$ โหนด

จำนวนโหนดที่ระดับ 1 คือ $2^1 = 2$ โหนด

จำนวนโหนดที่ระดับ 2 คือ $2^2 = 4$ โหนด

ต้นไม้ไบนารีแบบสมบูรณ์จะมีจำนวนโหนด 2^L โหนดพอดีที่ระดับ L ใดๆ โดย L มีค่าระหว่าง 0 และ d (0 <= L <= d)

ดังนั้นจำนวนโหนดทั้งหมดในต้นไม้ จึงหาได้จากผลรวมของจำนวนโหนดในแต่ละระดับจากระดับ 0 จนถึงระดับที่เป็นความลึกหรือระดับ d

$$\text{จำนวนโหนดทั้งหมด} = 2^0 + 2^1 + 2^2 + \ldots\ldots + 2^d$$

$$= \sum_{j=1}^{d} 2^J$$

$$= 2^{d+1} - 1$$

จากจำนวนโหนดทั้งหมดของต้นไม้ไบนารีแบบสมบูรณ์ที่มีความลึก d สามารถนำมาแยกเป็นโหนดใบไม้และโหนดที่ไม่ใช่ใบไม้ ดังนี้

- จำนวนโหนดใบไม้ทั้งหมดคือ 2^d
- จำนวนโหนดที่ไม่ใช่โหนดใบไม้ $2^d - 1$

ถ้าทราบจำนวนโหนดทั้งหมดในต้นไม้ไบนารีแบบสมบูรณ์ จะสามารถหาความลึกของต้นไม้ได้ โดย

จำนวนโหนดทั้งหมด

$$tn = 2^{d+1} - 1$$

$$tn + 1 = 2^{d+1}$$

$$\log_2 (tn + 1) = d + 1$$

$$d = \log_2(tn + 1) - 1$$

ตัวอย่าง

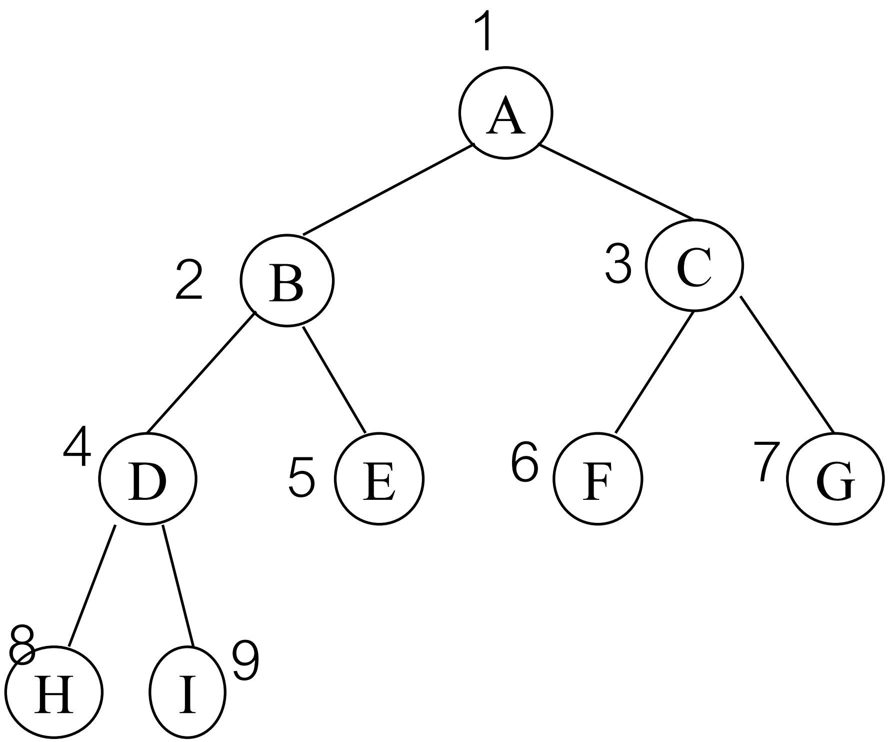
ต้นไม้ไบนารีที่สมบูรณ์มีจำนวนโหนดทั้งสิ้น 15 โหนด ต้นไม้ต้นนี้มีความลึกเท่าใด

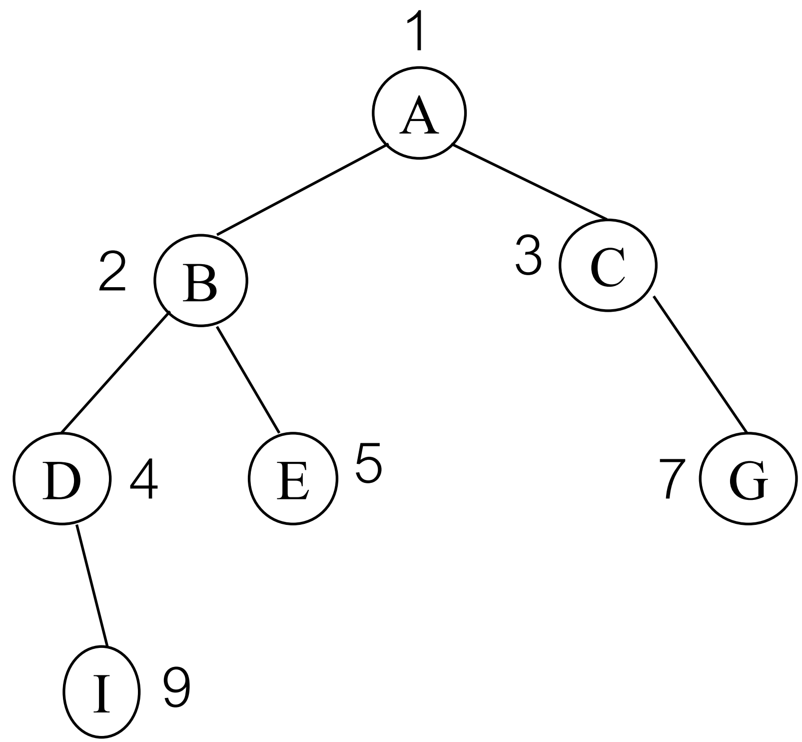
$$d = \log_2 (tn + 1) - 1$$
$$= \log_2 (15 + 1) - 1$$
$$= 3 \text{ ระดับ}$$

ตำแหน่งของโหนด

- โหนดราก (root node) มีเลขประจำตำแหน่งคือ 1
- โหนดที่เป็นลูกทางซ้าย จะมีค่าตำแหน่งเป็นสองเท่าของตำแหน่งของโหนดผู้ปกครอง
- โหนดที่เป็นลูกทางขวาจะมีค่าตำแหน่งเป็นสองเท่าบวกหนึ่ง ของโหนดผู้ปกครอง

ตัวอย่างการให้ค่าตำแหน่งของโหนดในต้นไม้ไบนารี





การเข้าถึงข้อมูลในต้นไม้

การเข้าถึงข้อมูลในต้นไม้แบบไบนารี (Tree Traversal)

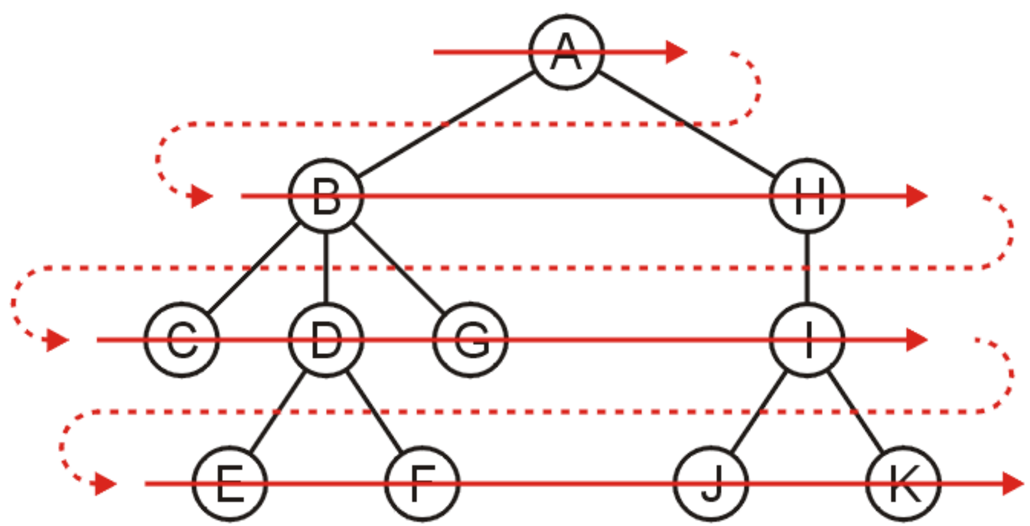
- การเข้าถึงโหนดใดๆ ในต้นไม้เราเรียกว่าการเยี่ยม (visiting)
- ลำดับของการเข้าถึงข้อมูลจึงขึ้นกับการนำไปใช้
- สามารถที่จะเข้าถึงโหนดได้ 2 รูปแบบ คือ

Breadth-First Traversal

Depth-First Traversal

Breadth-First Traversal

- Visit แต่ละโหนดโดยเริ่มที่ root
- เข้าถึงโหนดทีละระดับ
- ในแต่ละระดับเข้าถึงโหนดจากซ้ายไปขวา



Order: A B H C D G I E F J K

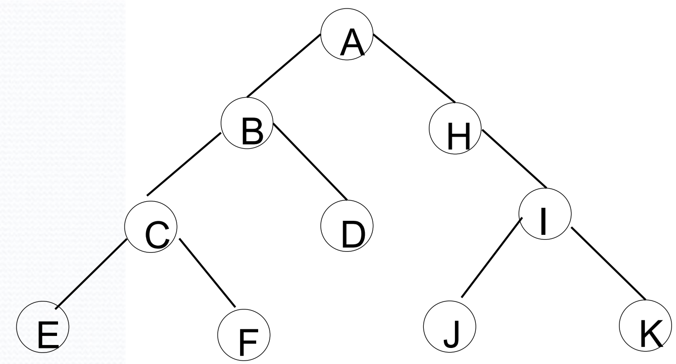
Breadth-First Traversal

ขั้นตอน

- กำหนดให้ queue ว่าง
- เริ่มเดินจาก root โดยเพิ่ม โหนด root ไปที่ queue
- ทำซ้ำด้านล่างนี้ ถ้า queue ยังไม่ว่าง
  - ลบหนึ่งโหนดออกจากคิว เอาลูกทั้งหมดของโหนดนี้เพิ่มเข้าไปในคิว
  - พิมพ์โหนดที่ถูกลบออกทางหน้าจอ

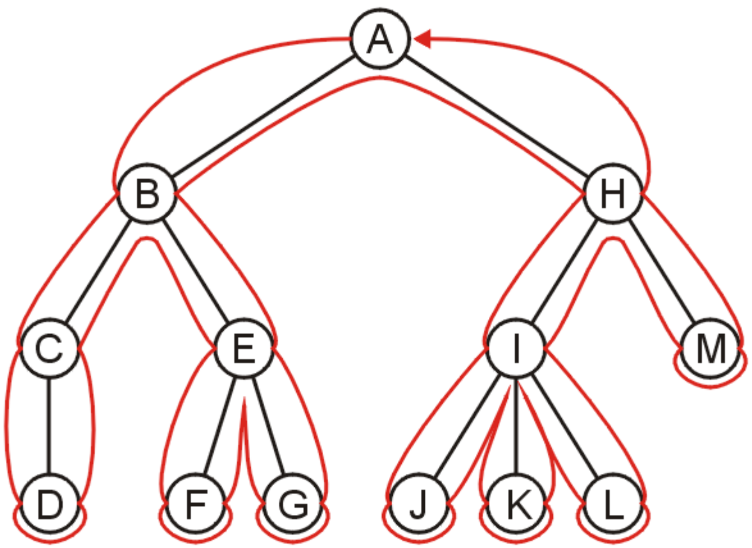
Breadth-First Traversal

front ค่าใน queue	ลำดับโหนดที่เข้าเยี่ยม
A	
B H	A
H C D	A B
C D I	A B H
D I E F	A B H C
I E F	A B H C D
E F J K	A B H C D I
F J K	A B H C D I E
J K	A B H C D I E F
K	A B H C D I E F J
	A B H C D I E F J K



Depth-First Traversal

- เข้าถึงโหนดตามเส้นทางจากโหนดรากไปยังลูกข้างใดข้างหนึ่งและลงไปถึงลูกหลานทั้งหมดของลูกข้างนั้น ก่อนที่จะเข้าถึงโหนดของลูกอีกข้างและโหนดลูกหลานของลูกข้างที่เหลือ

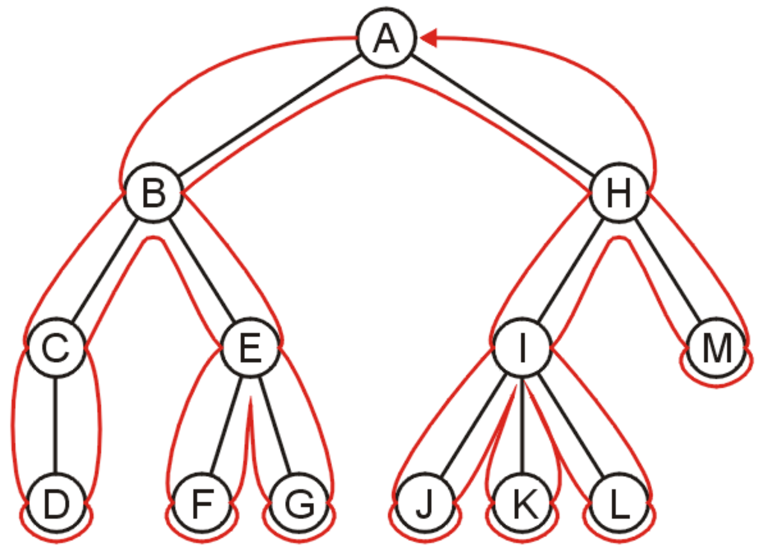


Order: A B C D E F G H I J K L M

Depth-First Traversal

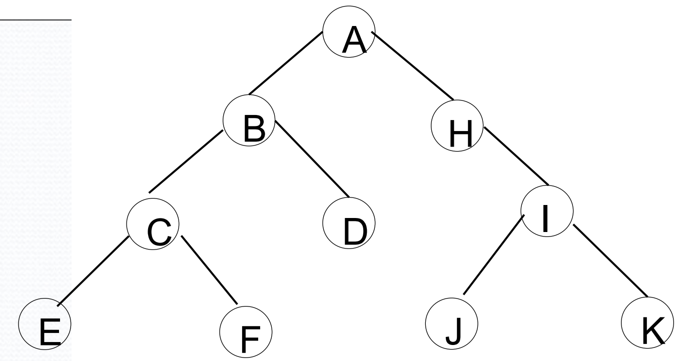
ขั้นตอน :

- เริ่มด้วยการใส่ root ไปที่ stack.
- Pop โหนดออกจาก stack แล้วใส่ลูกข้างขวา ตามด้วยลูกข้างซ้ายของโหนดนี้ลงไปในแสตก
- พิมพ์ค่าโหนดที่ pop ออกมา
- ทำซ้ำสองขั้นตอนก่อนหน้านี้ จนกว่าแสตกจะว่าง



Depth-First Traversal

ค่าใน Stack Top ---------------------------->	ลำดับของโหนดที่เข้าเยี่ยม
A	
H B	A
H D C	A B
H D F E	A B C
H D F	A B C E
H D	A B C E F
H	A B C E F D
I	A B C E F D H
K J	A B C E F D H I
K	A B C E F D H I J
	A B C E F D H I J K



Depth-First Traversal for Binary Tree

- แบ่งการเข้าถึงได้เป็น 3 งานย่อย
  - V การเข้าถึงโหนดราก
  - L การเข้าถึง left subtree
  - R การเข้าถึง right subtree

- ถ้าเข้าถึงโหนดทางซ้ายก่อนขวาเสมอ จะมีการเข้าถึง 3 รูปแบบ คือ
  - VLR -- preorder tree traversal
  - LVR -- inorder tree traversal
  - LRV -- postorder tree traversal

ลำดับแบบ Preorder

เป็นการเข้าถึงโหนดในต้นไม้แบบไบนารี ตามลำดับดังนี้

1. การเข้าถึงโหนดราก (root node)
2. การเข้าถึงต้นไม้ย่อยทางด้านซ้ายแบบ preorder
3. การเข้าถึงต้นไม้ย่อยทางด้านขวาแบบ preorder

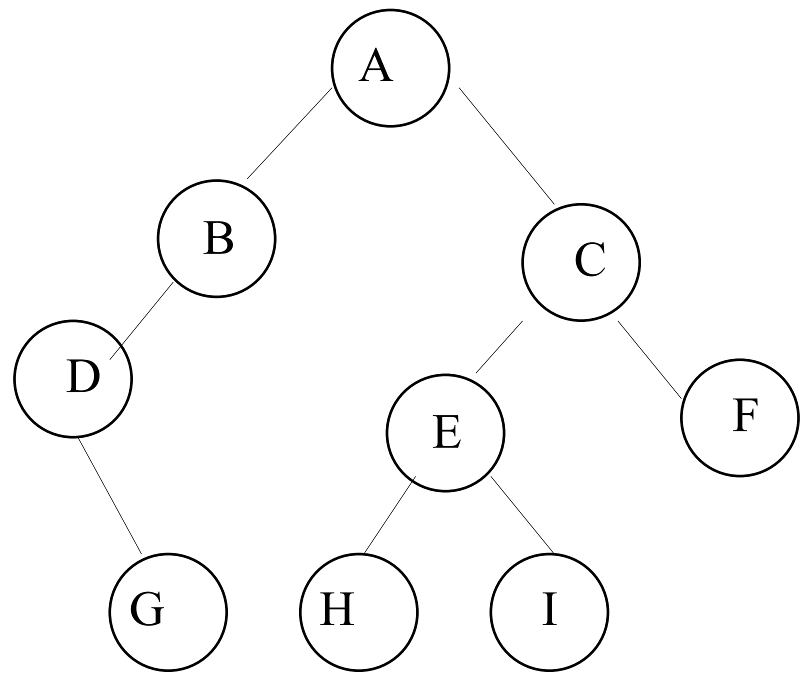
ตัวอย่าง การเข้าถึงโหนดแบบ preorder

root

Right subtree

A B D G C E H I F

Left subtree



ลำดับแบบ Inorder

เป็นการเข้าถึงโหนดในต้นไม้แบบไบนารี ตามลำดับดังนี้

1. การเข้าถึงต้นไม้ย่อยทางด้านซ้ายแบบ inorder
2. การเข้าถึงโหนดราก (root node)
3. การเข้าถึงต้นไม้ย่อยทางด้านขวาแบบ inorder

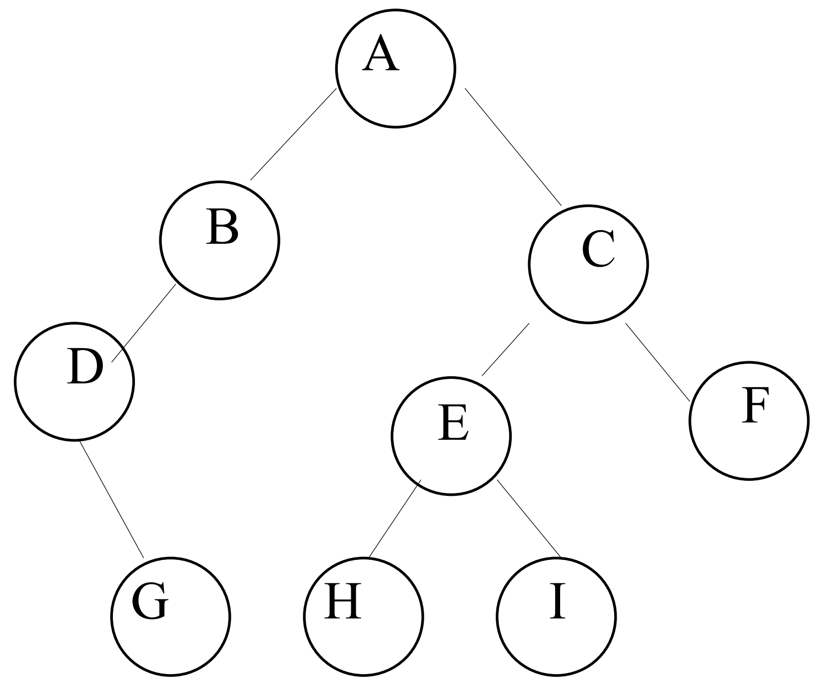
ตัวอย่าง การเข้าถึงโหนดแบบ inorder

root

Right subtree

D G B A H E I C F

Left subtree



ลำดับแบบ Postorder

เป็นการเข้าถึงโหนดในต้นไม้แบบไบนารี ตามลำดับดังนี้

1. การเข้าถึงต้นไม้ย่อยทางด้านซ้ายแบบ postorder
2. การเข้าถึงต้นไม้ย่อยทางด้านขวาแบบ postorder
3. การเข้าถึงโหนดราก (root node)

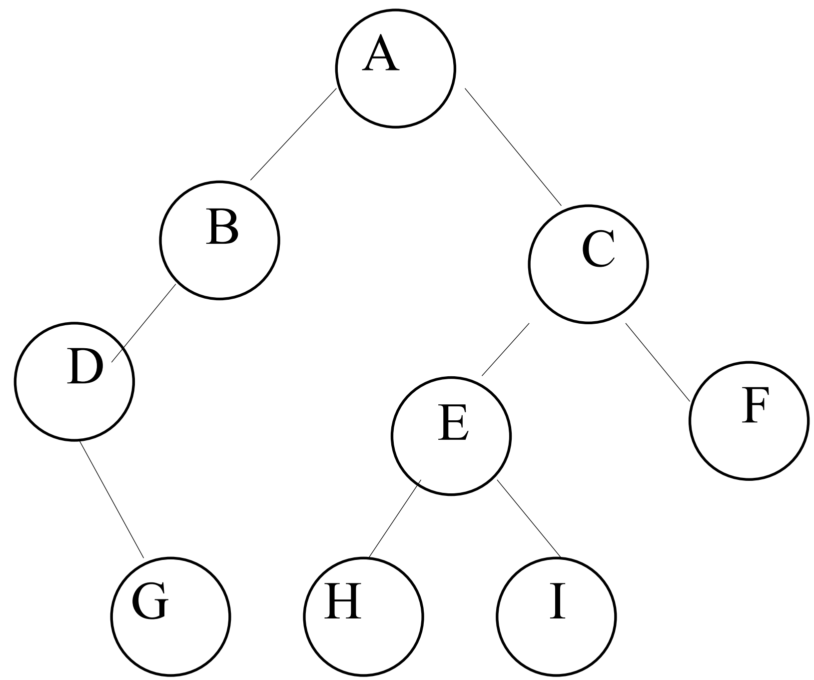
ตัวอย่าง การเข้าถึงโหนดแบบ postorder

root

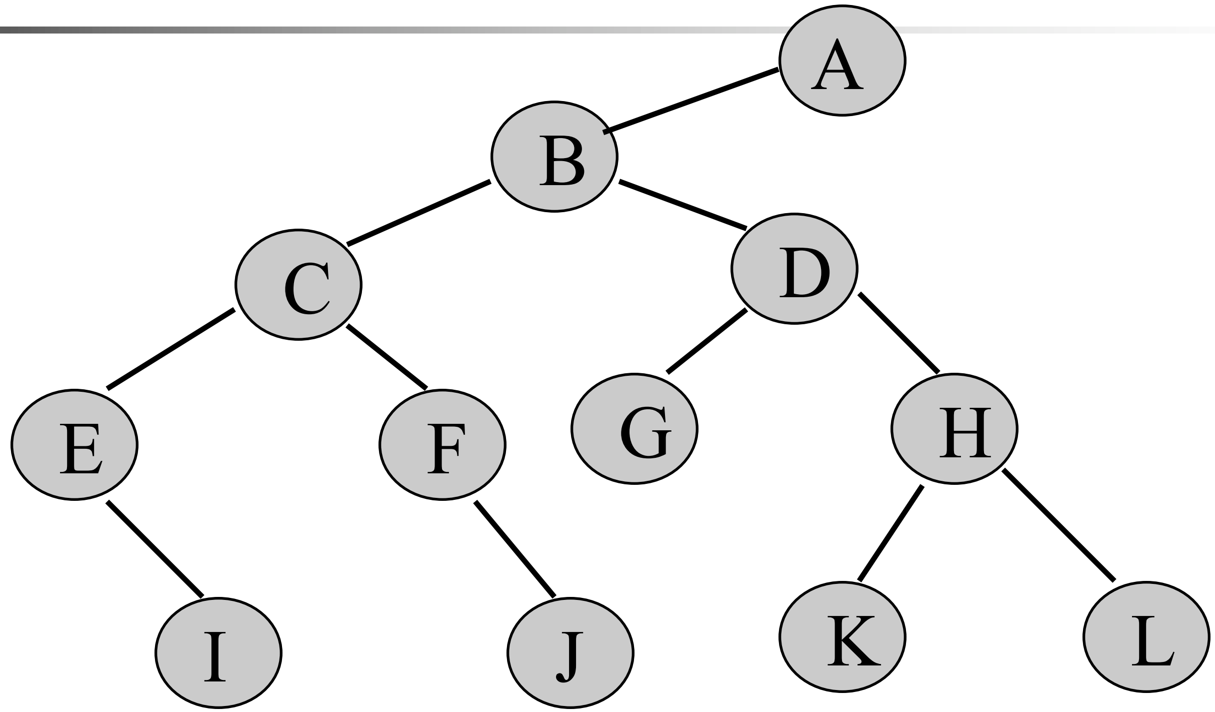
Right subtree

G D B H I E F C A

Left subtree



ตัวอย่าง



Preorder : *ABCEIFJDGHKL*
Inorder : *EICFJBGDKHLA*
Postorder : *IEJFCGKLHDBA*

ต้นไม้ไบนารีกับการแก้ปัญหา

- การหาเลขซ้ำ
- การเรียงลำดับข้อมูล

การหาตัวเลขซ้ำ

ข้อมูล 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

- วิธีการเปรียบเทียบทีละตัว
  - ต้องเปรียบเทียบข้อมูลทุกตัวกับข้อมูลทั้งหมดกว่าจะทราบว่ามีข้อมูลซ้ำกี่ตัวและซ้ำจำนวนเท่าใด
  - จำนวนครั้งของการเปรียบเทียบมาก
- สามารถที่จะใช้โครงสร้างต้นไม้แบบไบนารีมาแก้ปัญหาเพื่อลดจำนวนครั้งของการเปรียบเทียบ

การสร้างต้นไม้เพื่อหาเลขซ้ำ

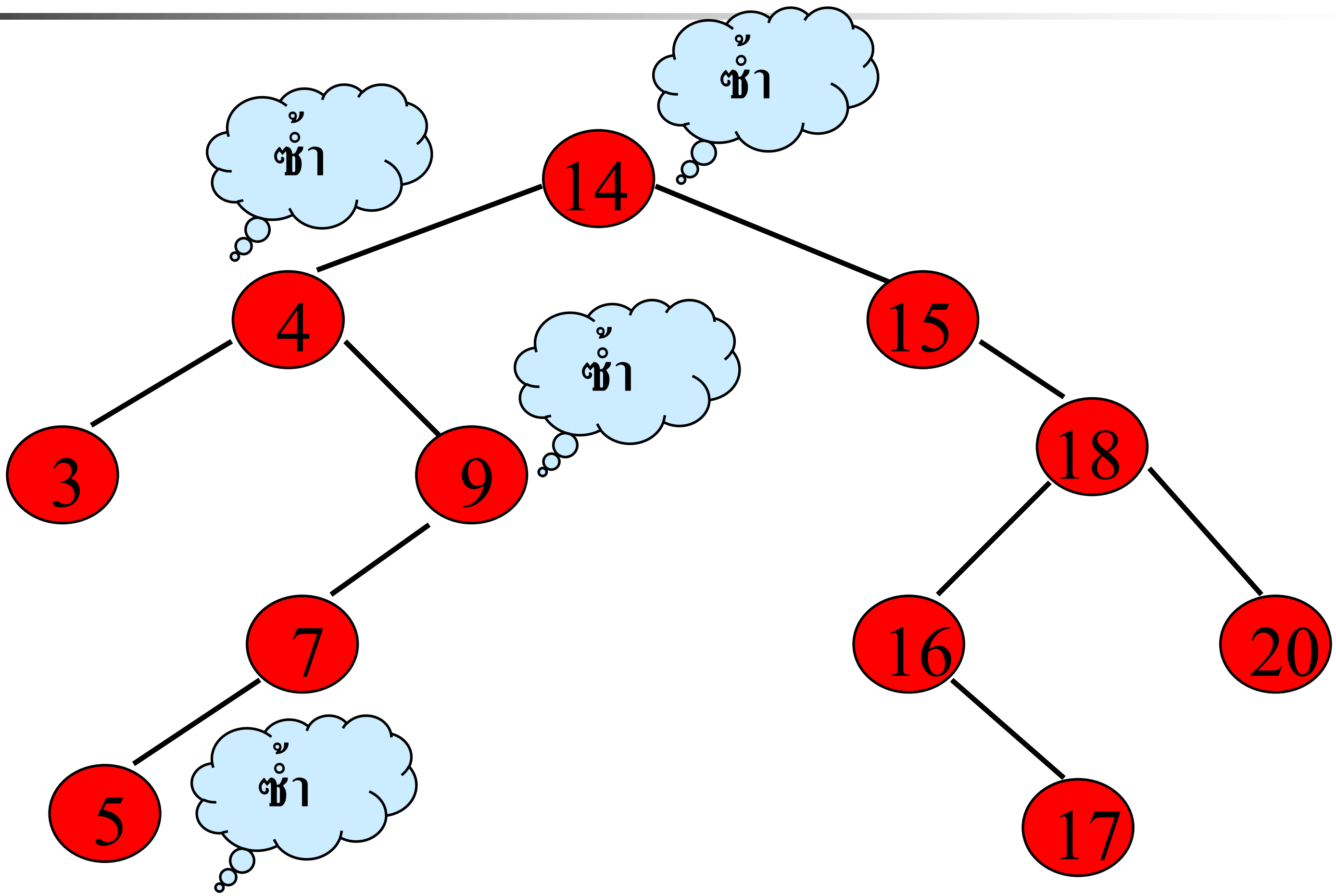
1. อ่านเลขเข้ามาทีละจำนวน
2. เลขจำนวนแรกที่อ่านเข้ามา สร้างเป็นโหนดรากของต้นไม้
3. เลขจำนวนถัดๆ มาให้ทำการเปรียบเทียบกับโหนดราก ซึ่ง

ผลของการเปรียบเทียบแบ่งเป็น 3 กรณี

- เลขที่อ่านเข้ามาเท่ากับเลขที่โหนดรากแสดงว่าเกิดการซ้ำ
- เลขที่อ่านเข้ามาน้อยกว่าเลขที่โหนดราก ให้พิจารณาต้นไม้ย่อยทางซ้าย
- เลขที่อ่านเข้ามามากกว่าเลขที่โหนดราก ให้พิจารณาต้นไม้ย่อยทางขวา

- ถ้าต้นไม้ย่อยที่ทำการเปรียบเทียบเป็นต้นไม้ว่าง และเลขที่อ่านเข้ามายังไม่ซ้ำ ก็ให้สร้างโหนดใหม่สำหรับเลขจำนวนนั้น ณ ตำแหน่งนั้น
- ถ้าต้นไม้ย่อยที่พิจารณาไม่ว่างเราจะทำการเปรียบเทียบเลขที่อ่านเข้ามากับโหนดรากของต้นไม้ย่อย แล้วทำซ้ำตั้งแต่ขั้นตอนที่ 3 จนกว่าข้อมูลจะหมด

ต้นไม้ไบนารีเพื่อหาเลขซ้ำ



14 , 15 , 4 , 9 , 7 , 18 , 3 , 5 , 16 , 4 , 20 , 17 , 9 , 14 , 5

การเรียงลำดับข้อมูล

<u>ชุดข้อมูล</u> 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

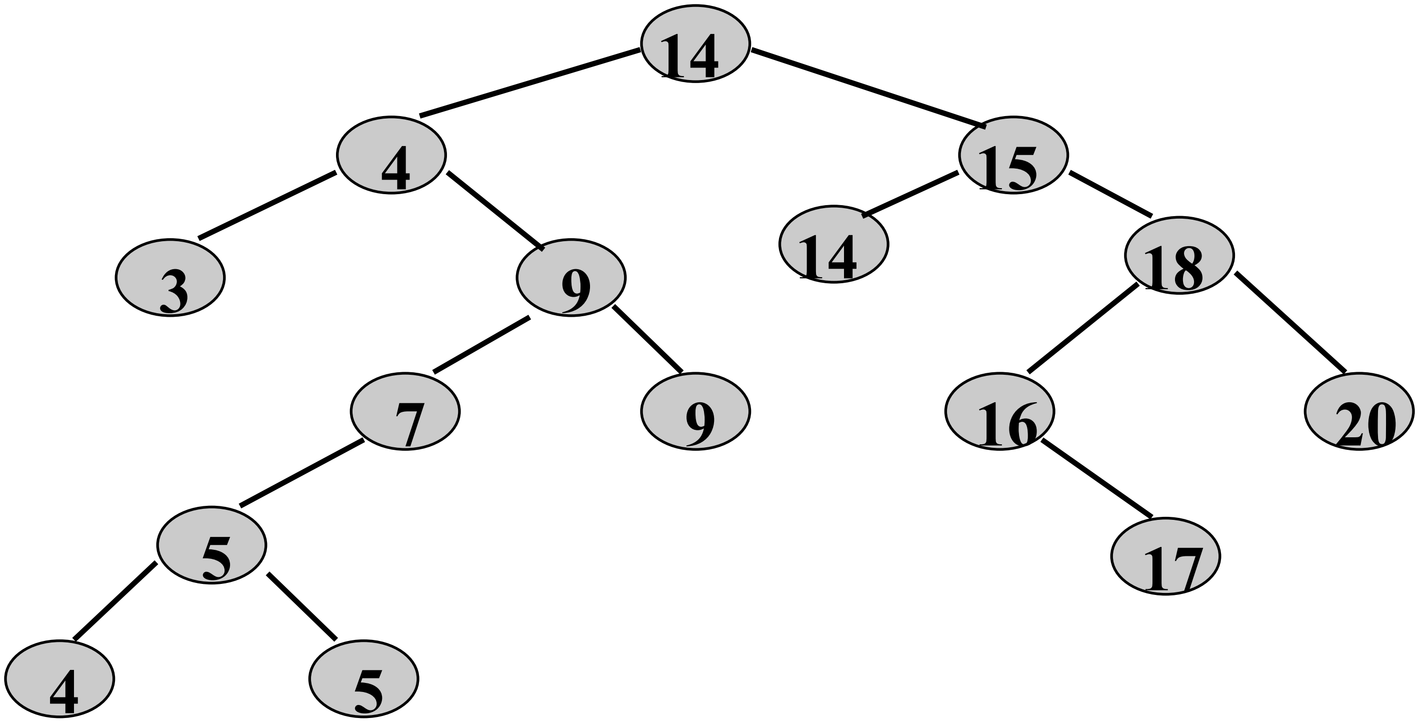
- การเรียงลำดับข้อมูลนั้นมีหลายวิธี
- ส่วนใหญ่ต้องใช้การเปรียบเทียบข้อมูลจำนวนมาก
- สามารถใช้ต้นไม้ไบนารีมาช่วยแก้ปัญหานี้ดังนี้

ต้นไม้ไบนารีเพื่อการเรียงลำดับ

- สร้างต้นไม้ไบนารีด้วยชุดของข้อมูลข้างต้น
- ทำการเปรียบเทียบเลขที่อ่านกับข้อมูลในโหนด ถ้า
  น้อยกว่า เปรียบเทียบต่อไปที่ต้นไม้ย่อยทางด้านซ้าย
  มากกว่าเปรียบเทียบต่อไปที่ต้นไม้ย่อยทางด้านขวา
  เท่ากันพิจารณาที่ต้นไม้ย่อยทางด้านขวา
- เข้าถึงของข้อมูลในต้นไม้ไบนารีและพิมพ์ข้อมูลในโหนดด้วยลำดับแบบ inorder

ต้นไม้ไบนารีเพื่อการเรียงลำดับ

[14 15 4 9 7 18 3 5 16 4 20 17 9 14 5]



ลำดับแบบ inorder คือ 3 4 4 5 6 7 9 9 14 14 15 16 17 18 20

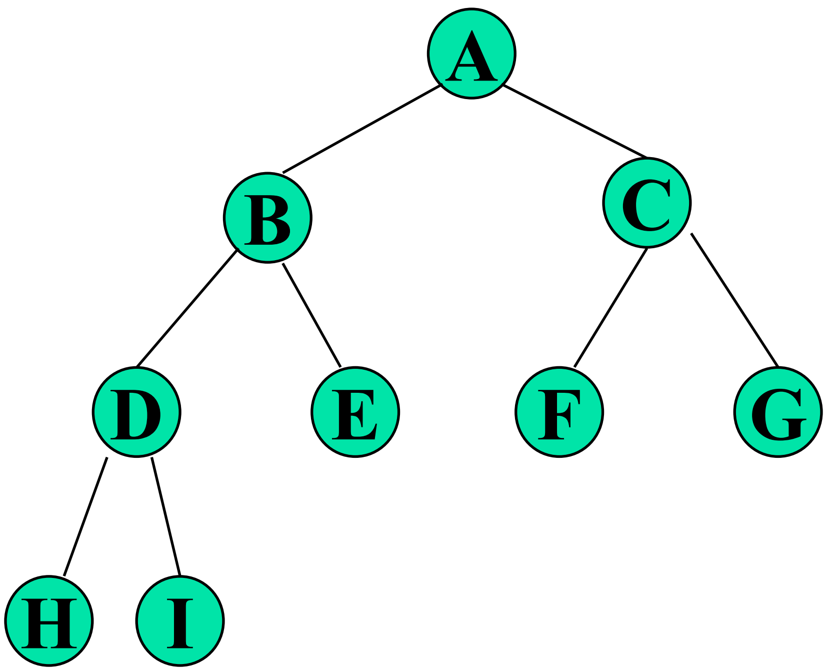
การสร้างต้นไม้ไบนารีในภาษา C/C++

เราสามารถสร้างโครงสร้างต้นไม้ได้ 3 แบบ ดังนี้

- **การสร้างด้วยอะเรย์**
  - อะเรย์แบบมีลิงค์ (Linked Array Representation)
  - อะเรย์แบบต่อเนื่อง (Sequential Array Representation)
- **การสร้างด้วยตัวแปรแบบพลวัตร**

  (Dynamic Node Representation)

ต้นไม้ไบนารีด้วยอะเรย์แบบมีลิงค์



	left	info	father	right
0	1	A	-1	2
1	3	B	0	4
2	5	C	0	6
3	7	D	1	8
4	-1	E	1	-1
5	-1	F	2	-1
6	-1	G	2	-1
7	-1	H	3	-1
8	-1	I	3	-1
9				
10				
:				

โครงสร้างของโหนด

```
const int NUMNODES = 500;

struct nodetype {
    char info;
    int left;
    int right;
    int father;
};

struct nodetype node[NUMNODES];
```

ต้นไม้ไบนารีด้วยอะเรย์แบบต่อเนื่อง

ใช้หมายเลขลำดับของโหนดที่เคยกำหนดมาก่อนหน้านี้เป็นตัวบอกตำแหน่งที่เก็บข้อมูลในอะเรย์

- กำหนดให้โหนดรากมีหมายเลข 1
- ลูกทางซ้ายของโหนด n ใด ๆจะมีค่าหมายเลข 2 n
- ลูกทางขวาของโหนด n ใดๆ จะมีค่าหมายเลข 2 n + 1

อะเรย์แบบต่อเนื่องในภาษา C/C++

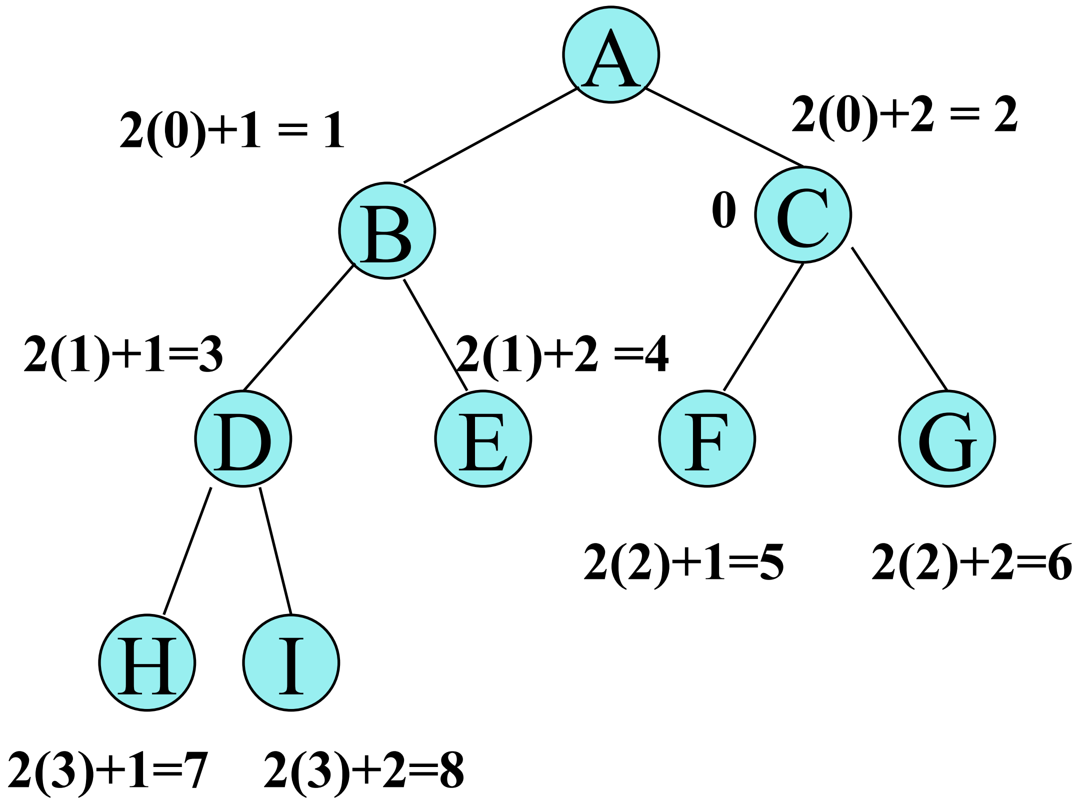
- ภาษา C/C++ มีการเก็บข้อมูลในอะเรย์ตั้งแต่ช่องที่ 0 แต่ไม่มีโหนดใดเลยที่มีค่าหมายเลขเป็น 0 ทั้งนี้เพื่อเป็นการใช้เนื้อที่ให้มีประสิทธิภาพ เราจึงกำหนดหมายเลขประจำโหนดใหม่ ดังนี้

    โหนดรากให้หมายเลข 0

    โหนดทางซ้ายของโหนด n ใดๆ มีหมายเลข 2n+1

    โหนดทางขวาของโหนด n ใดๆ มีค่า 2n+2

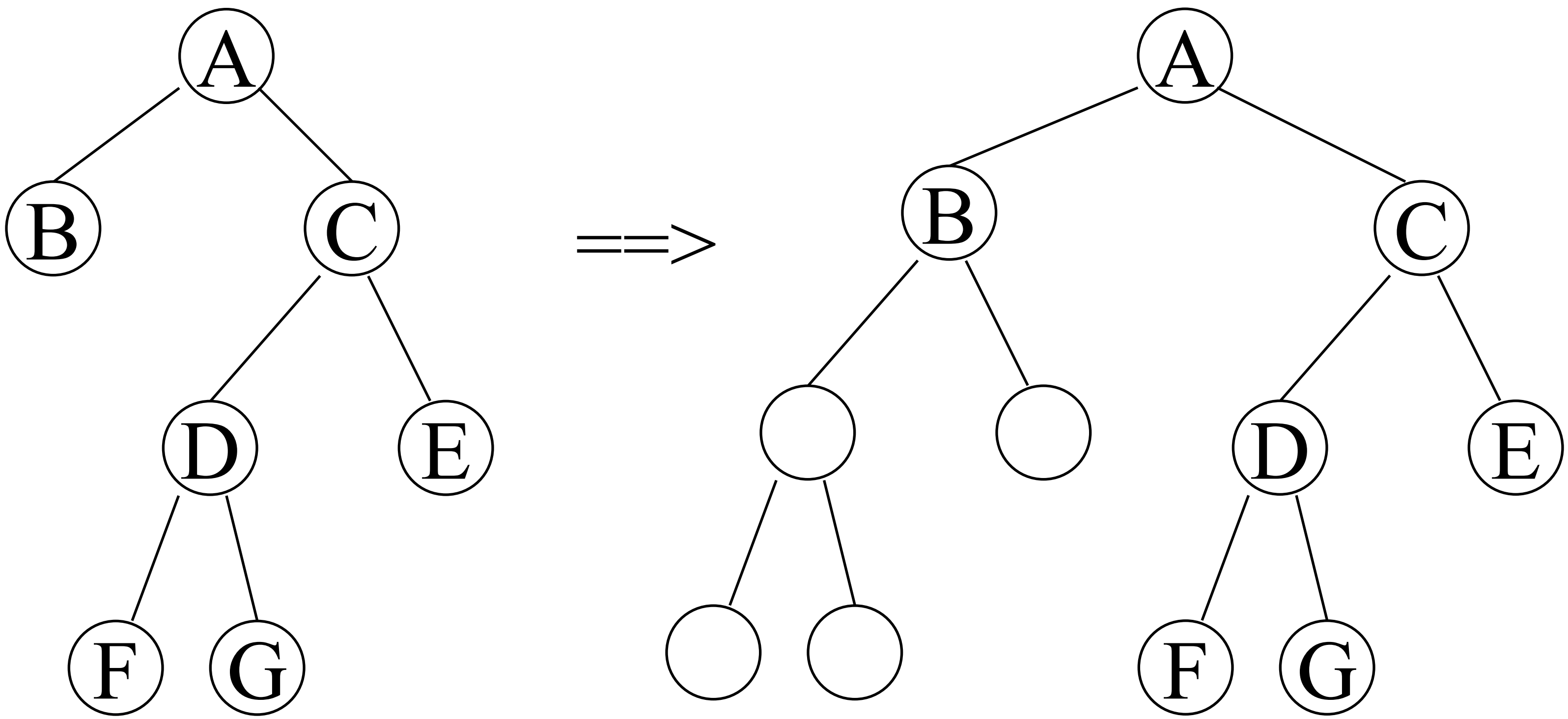
- โหนดรากของต้นไม้จะถูกเก็บในอะเรย์ที่ช่อง 0 เสมอ



A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8

- ถ้าลูกทางซ้ายอยู่ที่ตำแหน่ง p ในอะเรย์ ลูกทางขวาจะอยู่ที่ตำแหน่ง $p + 1$
- ถ้าลูกทางขวาอยู่ที่ตำแหน่ง p ในอะเรย์ ลูกทางซ้ายจะอยู่ที่ตำแหน่ง $p - 1$
- ถ้าโหนดที่ตำแหน่ง p เป็นลูกทางซ้าย โหนดพ่อจะอยู่ที่ตำแหน่ง $(p – 1)/2$
- ถ้าโหนดที่ตำแหน่ง p เป็นลูกทางซ้าย ก็ต่อเมื่อ p เป็นเลขคี่

การจัดเก็บแบบนี้ถ้าต้นไม้ไบนารีไม่ถูกเติมเต็มดังรูป เราก็ต้องมีการเผื่อเนื้อที่สำหรับโหนดทุกตำแหน่งไว้

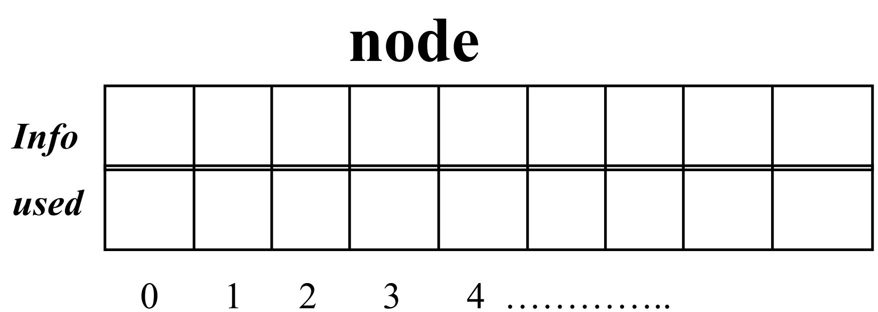


0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C			D	E					F	G

โครงสร้างของโหนด

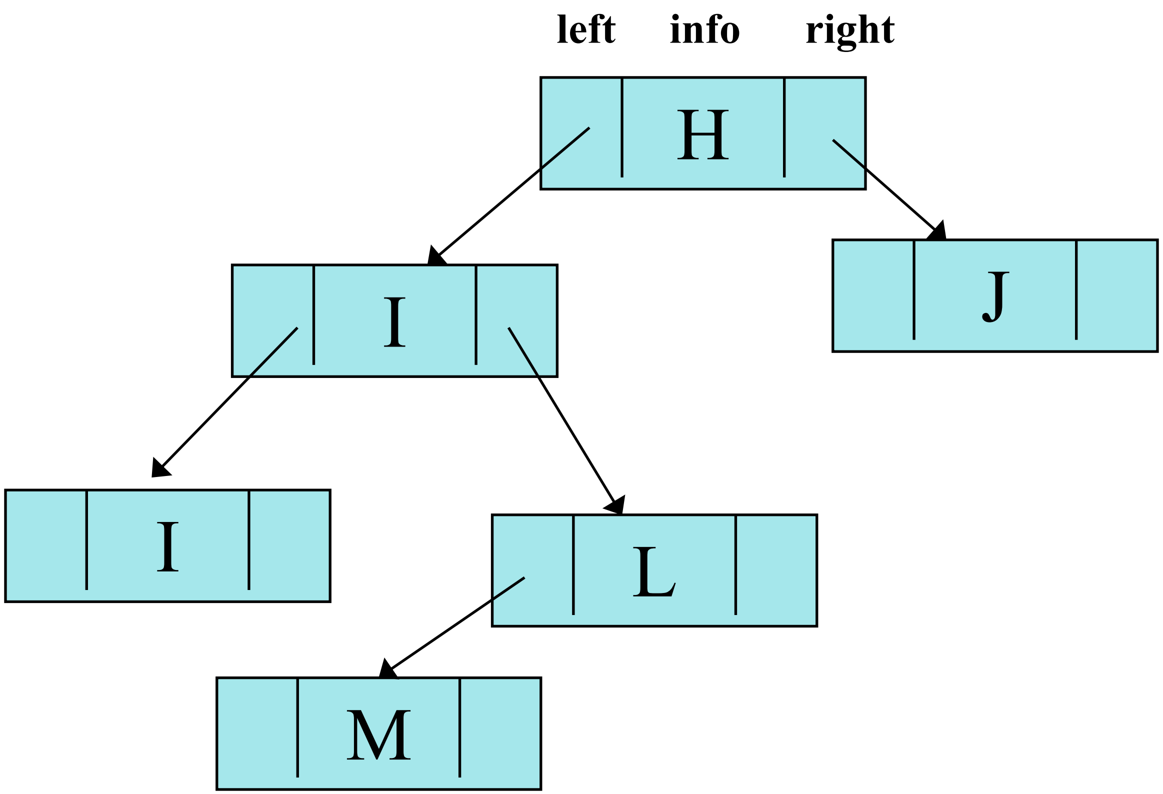
```
const int NUMNODES = 500;

struct nodetype {
    char  info;
    int   used;
} node[NUMNODES];
```



ต้นไม้ไบนารีด้วยตัวแปรแบบพลวัตร

การใช้ตัวแปรแบบพลวัตร ไม่จองเนื้อที่ให้แต่ละโหนดล่วงหน้า



โครงสร้างของโหนด

```
struct node {
    int info;
    struct node *left;
    struct node *right;
    struct node *father;  //optional
};

typedef struct node *NODEPTR;

NODEPTR root = NULL;
```

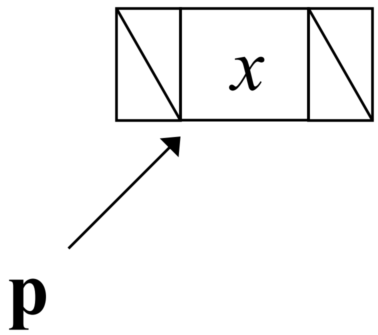
Tree operations

- **maketree(x)** – ขอที่สำหรับหนึ่งโหนดให้เป็นโหนดรากของต้นไม้
- **setleft (p, x)** – กำหนดให้โหนดมีค่า *X* และให้โหนดนี้เป็นลูกทางซ้ายของโหนด *p*
- **setright(p, x)** - กำหนดให้โหนดมีค่า *X* และให้โหนดนี้เป็นลูกทางซ้ายของโหนด *p*

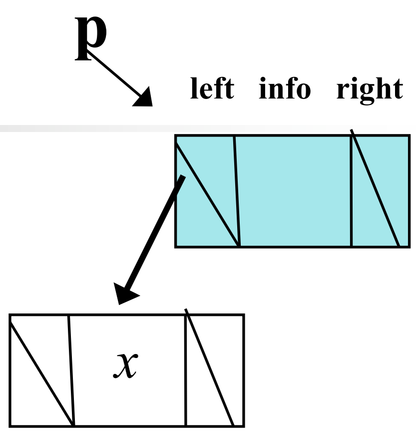
maketree function

```
NODEPTR makeTree (int x)
{
    NODEPTR p;

    p = new node;
    p->info = x;
    p->left = NULL;
    p->right = NULL;
    return (p);
}
```



setLeft function

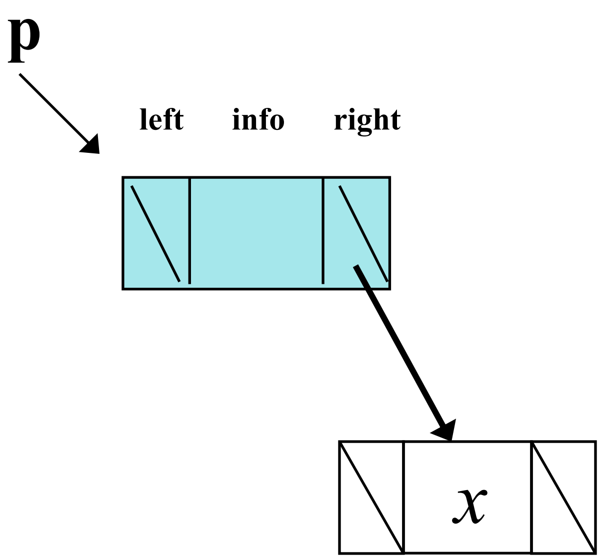



```
void setLeft (NODEPTR p, int x)
{
    if (p == NULL)
        cout << "can’t set left child to p" << endl;
    else if (p->left != NULL)
        cout << "p already has left child" << endl;
    else
        p->left = makeTree(x);

}
```

setright Function

เหมือน setLeft แต่ทิศทางตรงกันข้าม



Preorder Traversal

```
void preOrder (NODEPTR tree)
{
    if (tree != NULL)
    {
        cout << tree->info << " ";
        preOrder(tree->left);
        preOrder(tree->right);
    }
}
```

Inorder Traversal

```
void inOrder(NODEPTR tree)
{
    if (tree != NULL)
    {
        inOrder(tree->left);
        cout << tree->info << " ";
        inOrder(tree->right);

    }
}
```

Postorder Traversal

```
void postOrder (NODEPTR tree)
{
    if (tree != NULL)
    {
        postOrder(tree->left);
        postOrder(tree->right);
        cout << tree->info << " ";
    }
}
```

Breath-First Traversal

```
void breathFirst (NODEPTR root)
{
    NODEPTR p = root;

    while (p!= NULL)
         cout << p->info << " ";
         if (p->left != NULL)
            enqueue(p->left);
         if (p->right != NULL)
            enqueue(p->right);
         if (!emptyQ())
            p = dequeue()
         else
            p = NULL;
 }
```

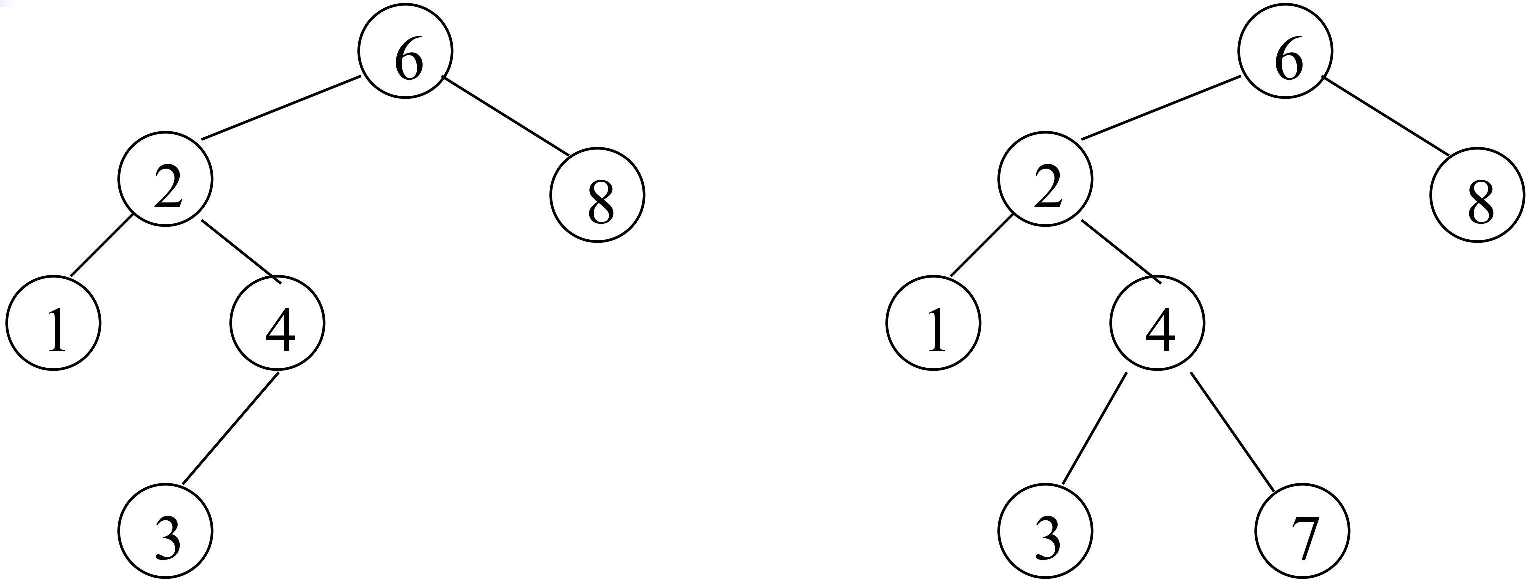
Binary Search Tree (BST)

ต้นไม้ไบนารีสำหรับการค้นหา (BST)

ต้นไม้ไบนารีสำหรับการค้นหา (Binary Search Tree) เป็นต้นไม้ไบนารีที่สำหรับโหนด x ใดๆ

- โหนดที่อยู่ในต้นไม้ย่อยทางซ้ายของโหนด x มีค่าน้อยกว่าโหนด x และ
- โหนดที่อยู่ในต้นไม้ย่อยทางขวาของโหนด x จะมีข้อมูลที่มากกว่าหรือเท่ากับโหนด x

และ ถ้าเราทำการท่องเข้าไปใน BST แบบ inorder เราจะได้ข้อมูลที่เรียงลำดับ จากน้อยไปมากเสมอ



ต้นไม้ทั้งสองเป็นต้นไม้ไบนารีทั้งคู่ แต่เฉพาะต้นซ้ายเท่านั้นที่มีคุณสมบัติเป็น BST

โครงสร้างของ BST

```
struct node {
    int info;
    struct node *left;
    struct node *right;
};
typedef struct node* NODEPTR;

NODEPTR root = NULL;
```

Operations of Binary Search Tree

- searchBST ใช้ในการค้นหาข้อมูลใน BST
- findSmallestBST หาข้อมูลที่มีค่าน้อยที่สุด
- findLargestBST หาข้อมูลที่มีค่ามากที่สุด
- insertBST เพิ่มโหนดของข้อมูลใน BST
- deleteBST ลบโหนดของข้อมูลใน BST

```c
NODEPTR searchBST (NODEPTR t, int key)
{
    if (t == NULL)
        return NULL;
    if (key < t->info)
        return  searchBST(t->left, key);
    else if (key > t->info)
        return  searchBST(t->right, key);
    else
        return t;
}
```

```
void findSmallest()
{
   if (root != NULL) {
      NODEPTR tmp;
      for(tmp=root; tmp->left!=NULL; tmp=tmp->left);
        cout << "The smallest is" << tmp->info << endl;
    }
}
```

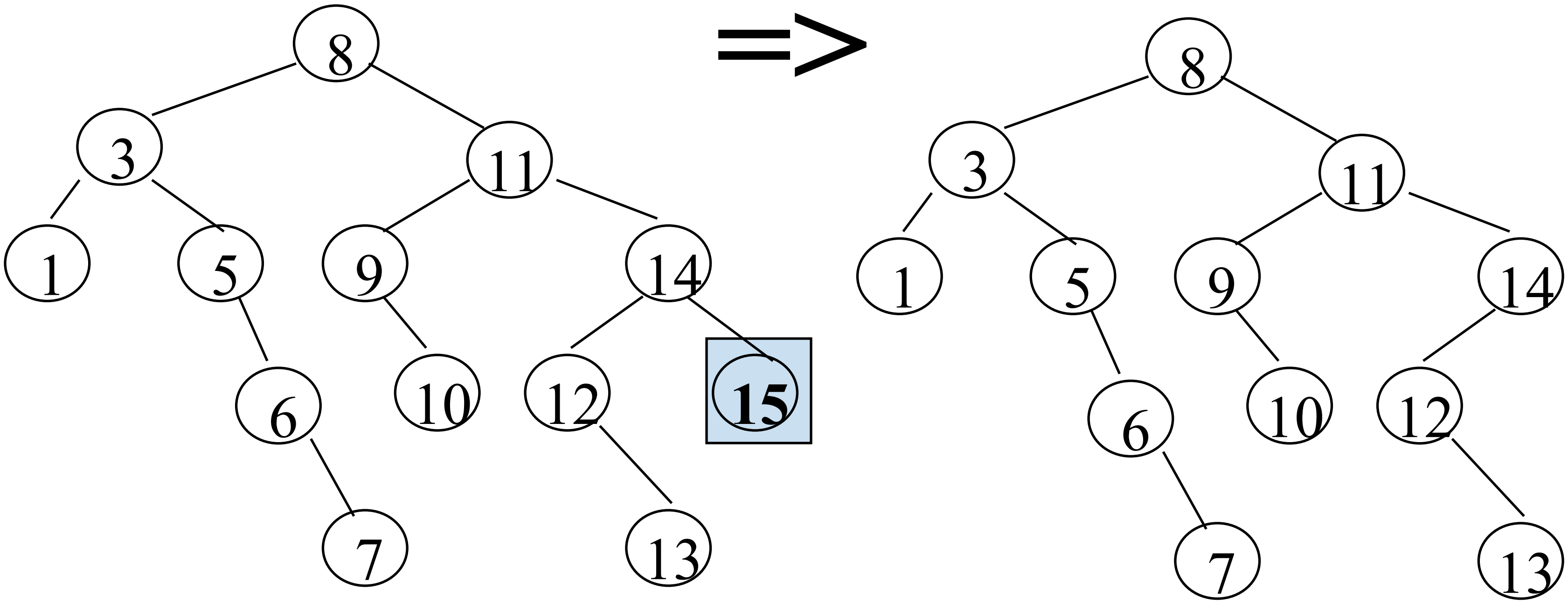
```cpp
void findLargest()
{
   if (root != NULL) {
      NODEPTR tmp;
      for(tmp=root;tmp->right!=NULL;tmp=tmp->right);
        cout << "The largest is " <<tmp->info<< endl;
    }
}
```

```
void insertBST( NODEPTR &t, int input)
{
    if (t == NULL) {
        t = new node;
        t->info = input;
        t->left = NULL;
        t->right = NULL;
    }
    else
    {
        if (input < t->info)
            insertBST(t->left, input);
        else
            insertBST(t->right, input);
    }
}
```

การ Delete ข้อมูล

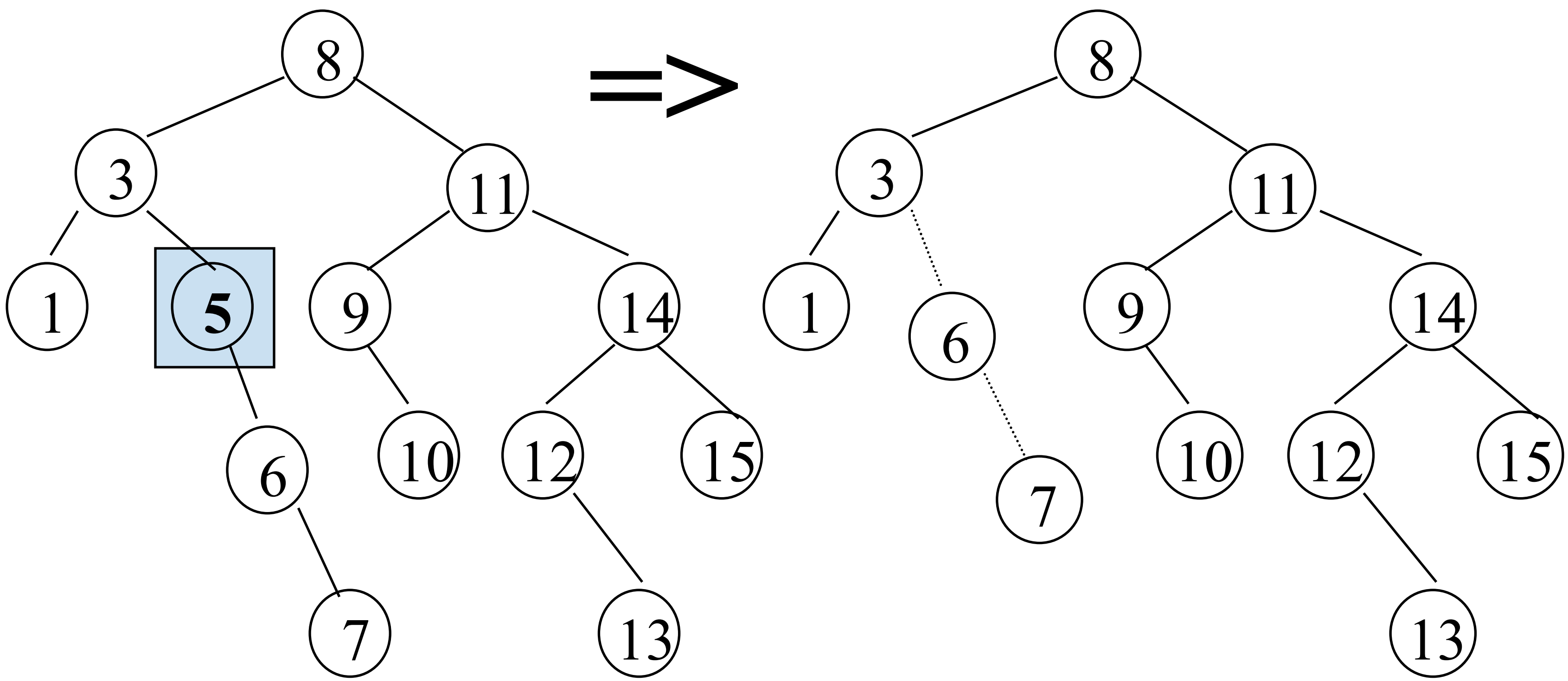
- การลบข้อมูลใน BST ค่อนข้างจะยุ่งยาก เนื่องจากพบลบแล้วก็ต้องทำการปรับ ลิงค์ต่าง ๆ ซึ่งแตกต่างกันเป็นกรณีๆ ไป
- หลังจากลบแล้ว ต้นไม้ที่เหลือจะต้องคง คุณสมบัติของ BST คือโหนดทางซ้ายมีค่าน้อยกว่าโหนดกลางและโหนดทางขวา มีค่ามากกว่าหรือเท่ากับโหนดกลาง
- เราแบ่งการพิจารณาออกได้เป็น 3 กรณี

<u>กรณีที่1</u>   โหนดที่ต้องการลบเป็นโหนดใบไม้



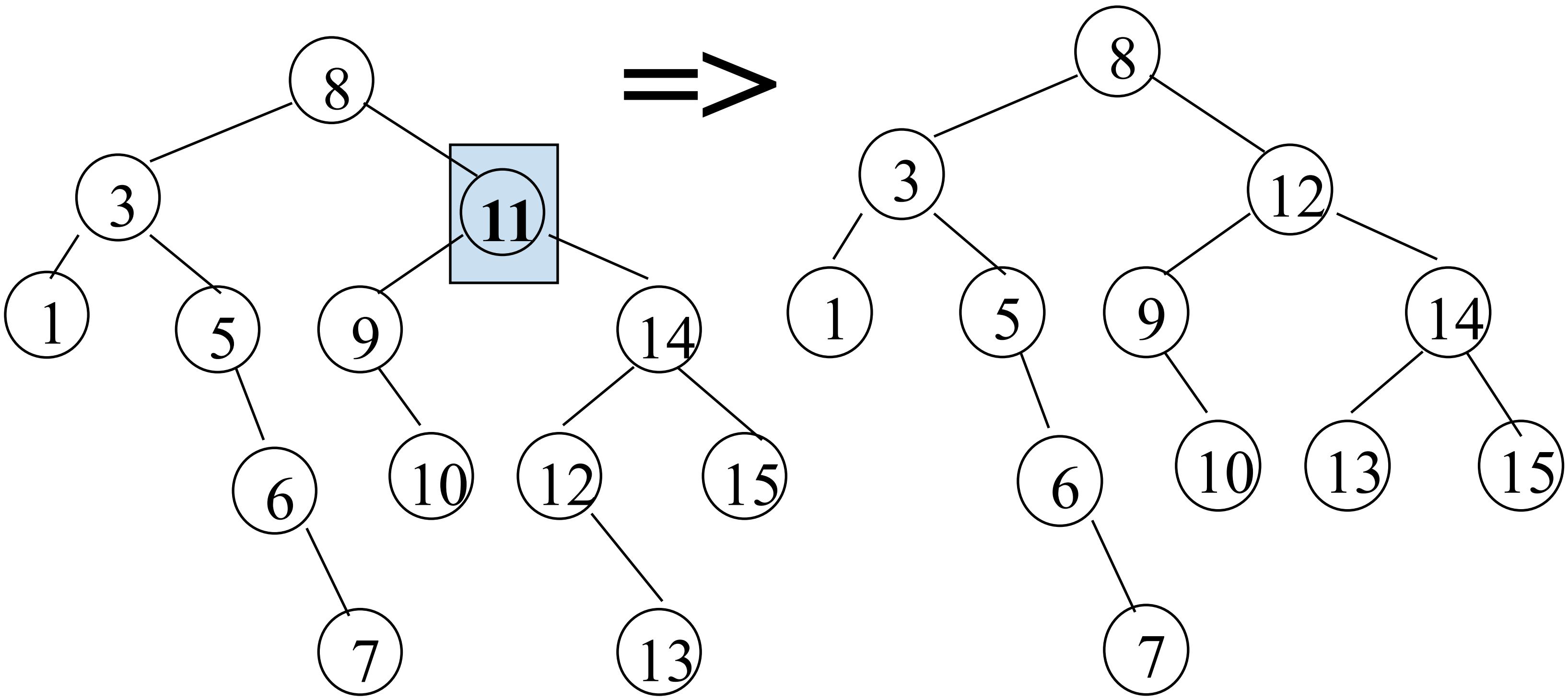
(a) deleting node with key 15.

กรณีที่ 2 โหนดที่ต้องการลบมีต้นไม้ย่อยเพียงข้างซ้ายข้างเดียวหรือข้างขวาเพียงข้างเดียว



(b) deleting node with key 5.

<u>กรณีที่3</u> โหนดที่ต้องการลบมีต้นไม้ย่อยทั้งสองข้าง



(c) deleting node with key 11.

ประสิทธิภาพการค้นหาของ BST

- ประสิทธิภาพการค้นหาข้อมูลใน BST มักจะขึ้นกับรูปร่างของต้นไม้ว่ามีความสมดุลหรือ เอียงมากน้อยแค่ไหน