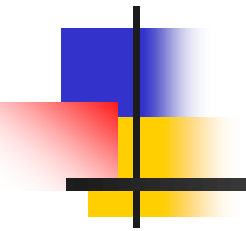
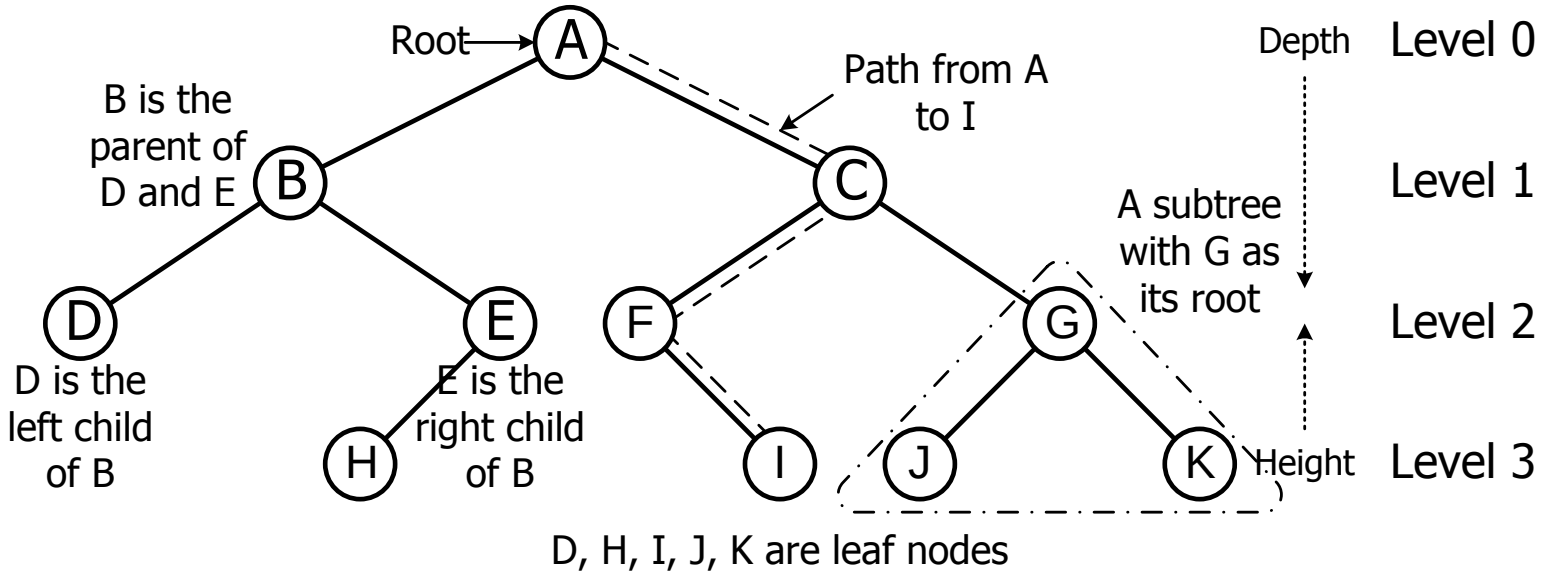


Chapter 05

Binary Trees

Trees Data Structure

- **Trees เป็นโครงสร้างข้อมูลแบบต้นไม้ ที่ประกอบด้วยความสัมพันธ์ของโหนด (Nodes) ซึ่งใช้เก็บข้อมูล เชื่อมต่อกันด้วยเอดจ์ (edges) ลดหลั่นตามลำดับชั้น**
- **โหนดแรกของทรี เรียกว่าราก (root) เป็นทางเข้า มีได้เพียงโหนดเดียว สามารถแตกออกเป็นโหนดลูก(child) ได้หลายโหนด**
- **โหนดที่แตกออกมาจากโหนดราก จะมีคุณสมบัติเหมือนกับเดิม คือสามารถแตกออกเป็นโหนดลูกย่อยๆได้อีก และแสดงคุณสมบัติเป็นต้นไม้ย่อย(Subtree)**
- **เส้นทางในการเข้าถึงโหนดต่างๆ เรียกว่าพาท(Path) แต่ละโหนดจะมีทางเข้าถึงได้เพียงพาทเดียว**
- **Binary Trees คือ Trees ที่แต่ละโหนดสามารถแตกโหนดลูกได้ไม่เกิน 2 โหนด**

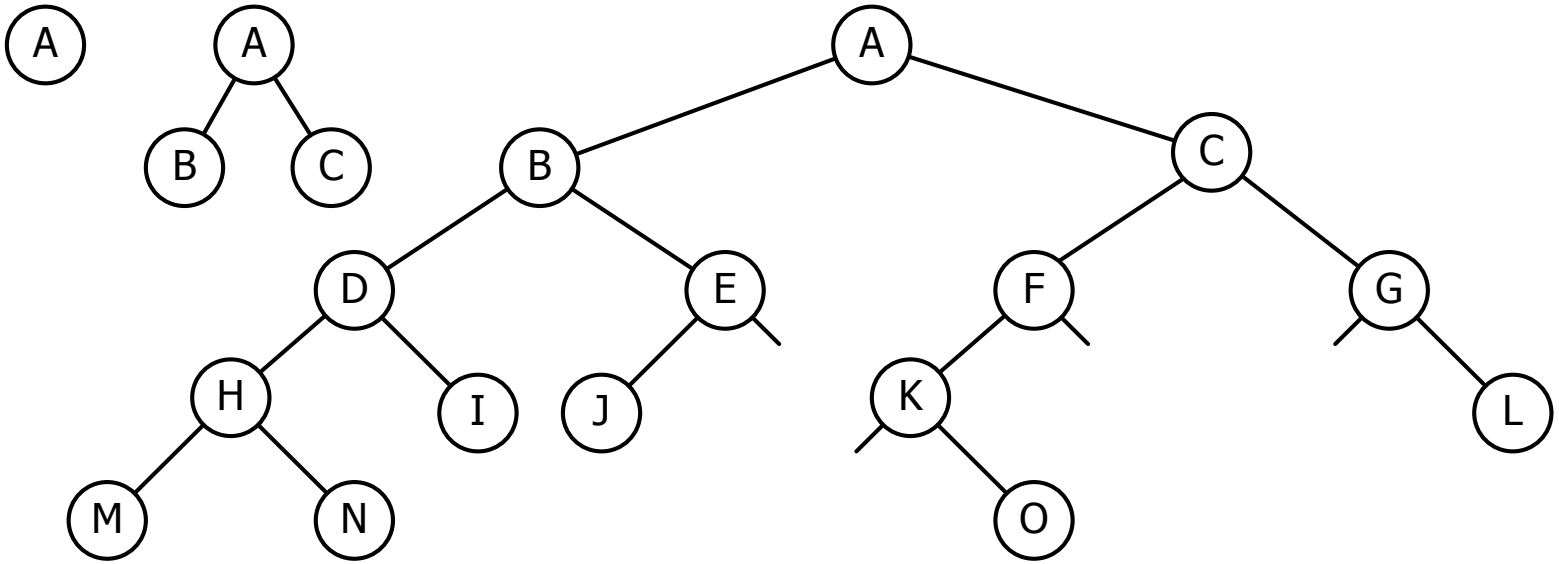


นิยามคำศัพท์เกี่ยวกับ Tree

- **Tree** ประกอบด้วยโหนด (nodes) ที่เชื่อมต่อกันด้วย edges
- **Edge** เส้นที่เชื่อมต่อระหว่างโหนด
- **Path** เส้นทาง(edges) จากโหนดหนึ่งไปยังอีกโหนดหนึ่ง ซึ่งจะมีได้เพียงเส้นทางเดียว
- **Path length** จำนวน edge ที่เชื่อมระหว่างโหนดหนึ่งไปยังอีกโหนดหนึ่ง
- **Root** คือโหนดสูงสุดของ Tree (มีได้เพียงตัวเดียว)
- **Parent** โหนดใดๆ (ยกเว้น Root) ที่อยู่เหนือโหนดอื่น(Child)
- **Child** โหนดใดๆ ที่อยู่ใต้โหนดอื่น(Parent)
- **Leaf** โหนดสุดท้ายที่ไม่มีโหนดลูก
- **Subtree** ต้นไม้ย่อยที่อยู่ใต้ Root
- **Levels** ระดับชั้นของต้นไม้ เริ่มตั้งแต่ Root เป็น level 0 และลดหลั่นลงมาเป็น level 1, 2, 3, ... ตามลำดับ
- **Height** ความสูง(level) ของโหนด คือระยะพาทที่วัดจากโหนดไปยังโหนดสุดท้ายที่ต่ำที่สุดของ subtree นั้น
- **Depth** ความลึกของโหนด คือระยะพาทที่วัดจากโหนดรูทมายังโหนดนั้น
- **Binary Tree** ต้นไม้ที่มีโหนดลูกได้ไม่เกิน 2 โหนด (2-Subtree)
- **M-ary Tree** ต้นไม้ที่มีโหนดลูกได้ไม่เกิน M โหนด
- **Full node** ทุกโหนดที่มีโหนดลูก จะต้องแตกเป็น M โหนด เท่านั้น
- **Perfect Tree** ต้นไม้ที่โหนดสุดท้าย(leaf) ทุกโหนดจะอยู่ใน level เดียวกันหมด
- **Complete Tree** ต้นไม้ที่แตกโหนดลูกตามลำดับจากซ้ายไปขวา
- **Balance Tree** ต้นไม้ที่ความสูงของต้นไม้ย่อยทางซ้าย กับทางขวาต่างกันไม่เกิน 1

Binary Trees

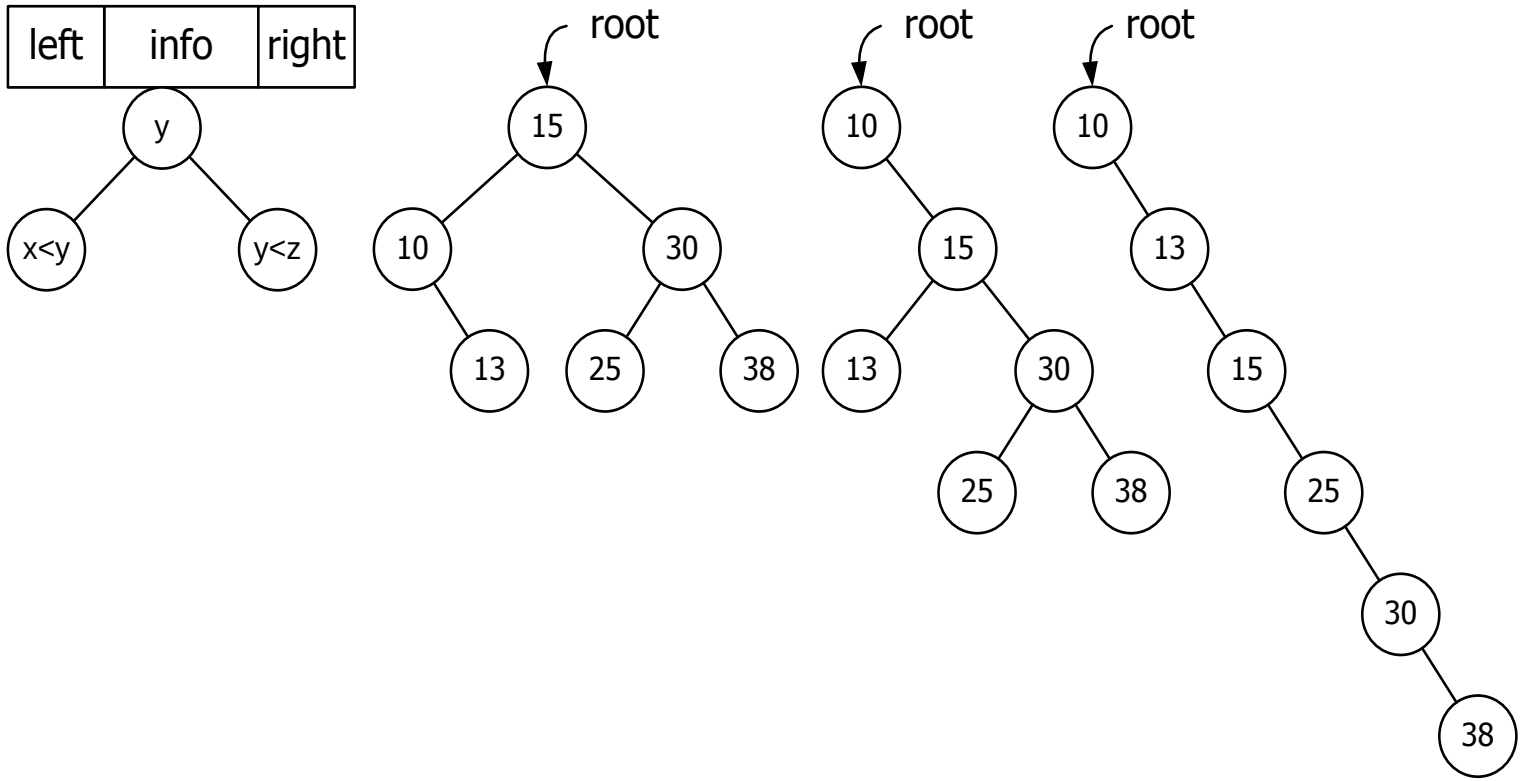
- **Binary trees คือต้นไม้ที่มีโหนดลูกได้มากที่สุด 2 โหนด คือ โหนดทางซ้าย (left child หรือ left subtree) และ โหนดทางขวา (right child หรือ right subtree)**



Binary Search Trees

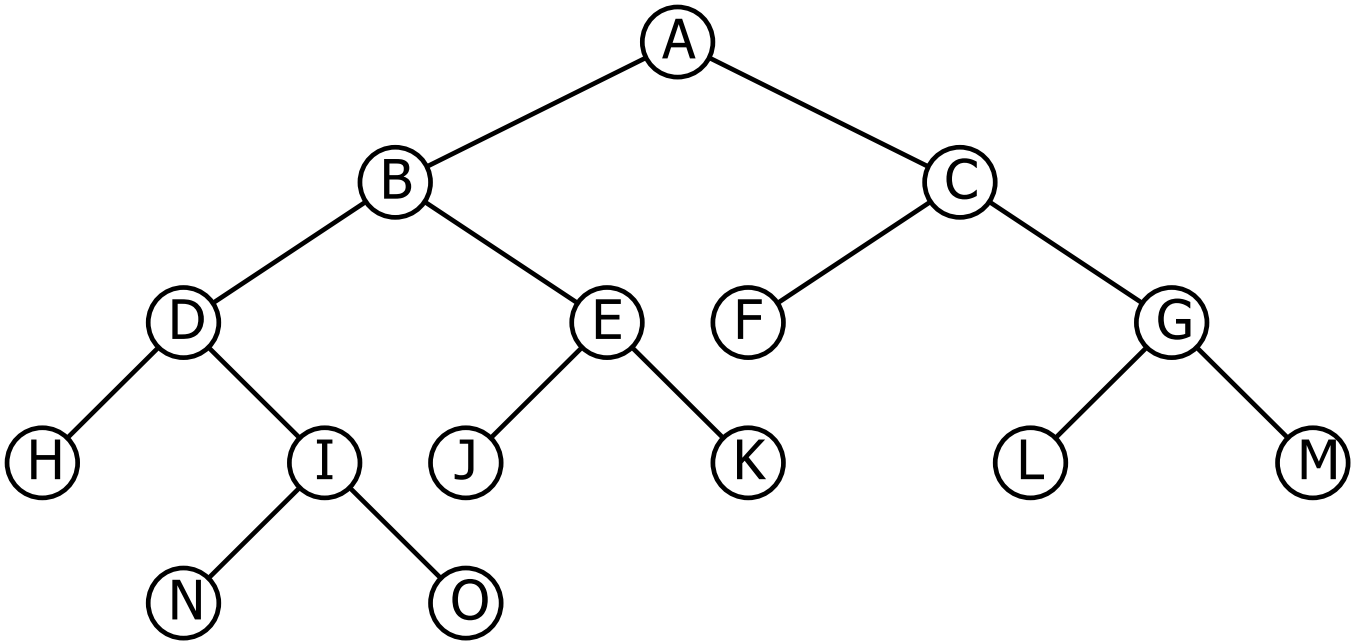
- **Binary search tree** คือ **Binary Tree** ที่ค่าของทุกโหนดที่อยู่ทางซ้าย จะต้องมีค่าคีย์น้อยกว่า **parent** และทุกโหนดที่อยู่ทางขวาจะมีค่ามากกว่า (หรือเท่ากับ ถ้ายอมให้มีโหนดซ้ำ) **parent**

all key in left subtree < root < all key in right subtree



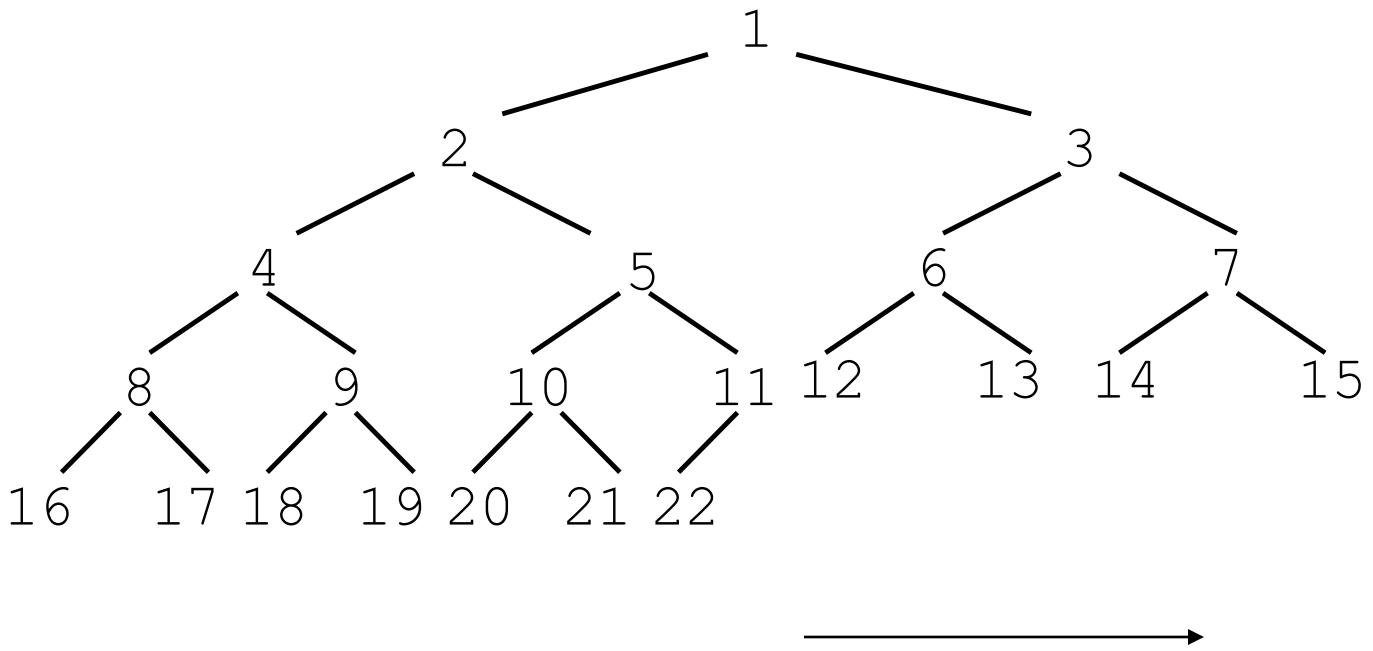
Full Binary Trees (2-trees)

- **Full binary tree (2-tree)** ต้นไม้ที่ทุกๆโหนดที่มีโหนดลูกจะต้องมีโหนดลูกทั้ง 2โหนด



Complete Binary Trees

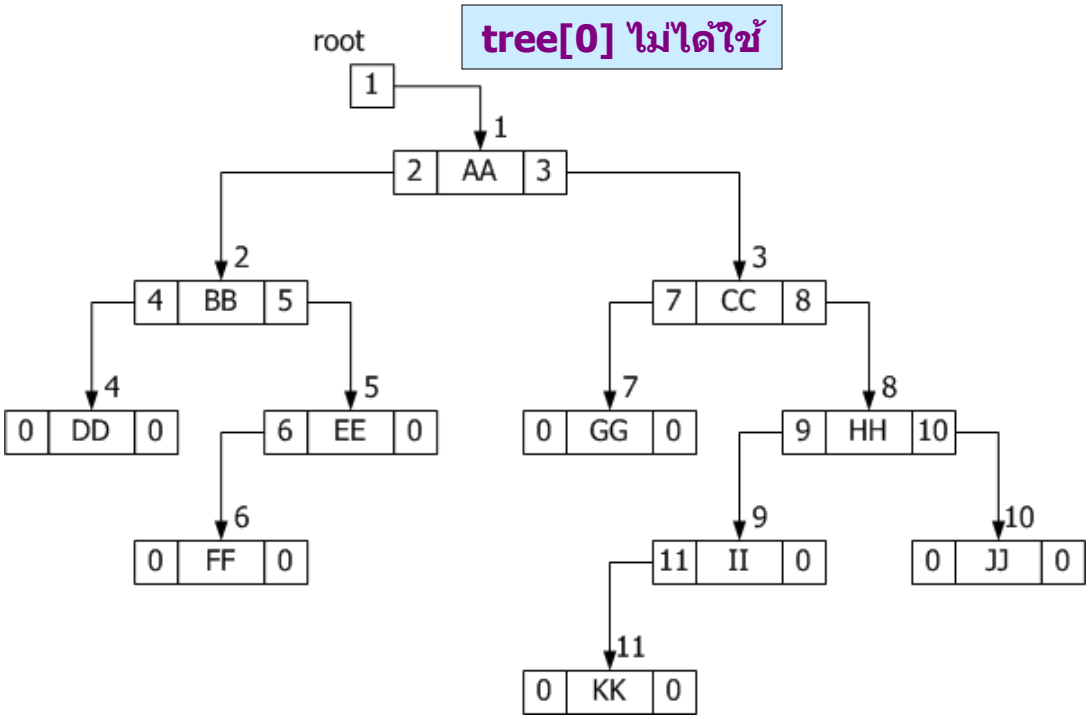
- **Complete binary tree ต้นไม้ที่มีการเกิดของโหนดในแต่ละระดับเพิ่มขึ้นจากซ้ายไปขวา**



Implement Binary Trees

- ตัวอย่างการสร้าง Binary tree โดยใช้ Linear Array

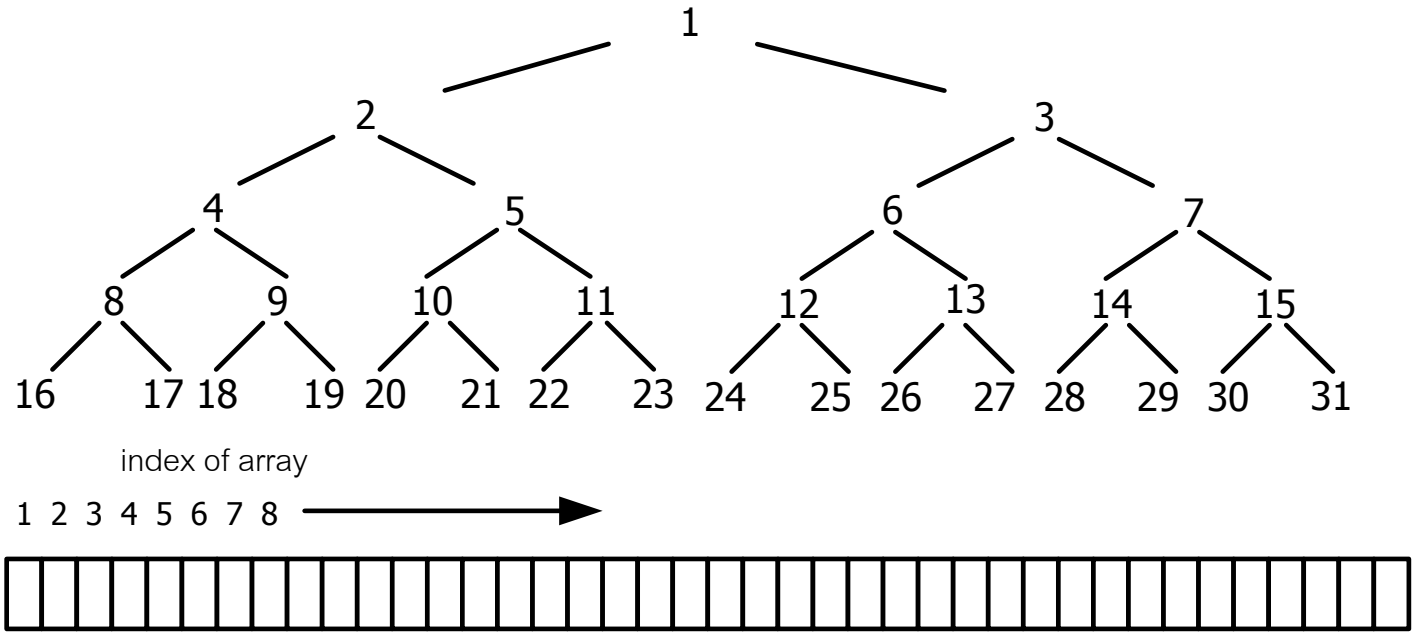
```
class  NodeArray { String info;
                   int left, right; } ;
NodeArray [] tree = new NodeArray[16];
int root, avail;
```



		info	left	right
root	1	AA	2	3
	2	BB	4	5
	3	CC	7	8
	4	DD	0	0
	5	EE	6	0
	6	FF	0	0
	7	GG	0	0
	8	HH	9	10
	9	II	11	0
	10	JJ	0	0
	11	KK	0	0
avail	12			
	13			
	14			
	15			

Implement Binary tree in Sequential Array

- **Complete Binary Trees ต้นไม้ที่มีการสร้างโหนดตามลำดับจากซ้ายไปขวา**
- **สามารถนำอาร์เรย์มาใช้แทนได้ โดยการคำนวณตำแหน่งของโหนดทางซ้าย และขวาแทนการใช้ตัวชี้ตำแหน่ง**



ตำแหน่ง index ของ root node = 1

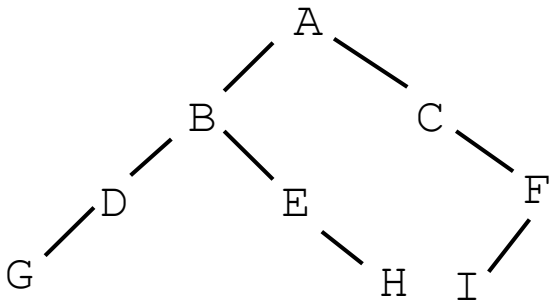
ตำแหน่ง index ของ node ใดๆ = K

ตำแหน่ง index ของโหนดที่อยู่ทางซ้ายของโหนด K = 2*K

ตำแหน่ง index ของโหนดที่อยู่ทางขวาของโหนด K = 2*K+1

Example

- ตัวอย่างตำแหน่งที่เก็บข้อมูลในอาร์เรย์ของไบนารี่ทรี



A	B	C	D	E		F	G			H			I
1	2	3	4	5	6	7	8	9	10	11	12	13	14

ตำแหน่งที่อยู่ของโหนด A = 1
ตำแหน่งที่อยู่ของโหนด B = 2*ตำแหน่งของ A
= 2*1 =2
ตำแหน่งที่อยู่ของโหนด E = 2*ตำแหน่งของ B + 1
= 2*2 + 1 = 5

Class Node & Binary Tree

```
class  Node {   long info ;  // คลาสสำหรับ โหนดเก็บข้อมูล
                Node  left, right;     // อาจเพิ่มให้มี parent เพิ่มเพื่อชี้กลับไปโหนดแม่
        public    Node (long info) {  // constructor method ชื่อเดียวกับ class แต่ไม่ return
                this.info = info;   // ใช้ this. ถ้าชื่อฟิลด์ ซ้ำกับชื่อพารามิเตอร์
                this.left = this.right = null;
        }
}
class  BinaryTree {  // คลาสสำหรับใช้เป็น binary tree
                        Node   root;
        public  BinaryTree () {          // constructor ใช้สำหรับ initial object อัตโนมัติ
                root = null;
        }
        public void insertNewNode(long info) { ..... }
        ..........................................................
}
```

insert a new node

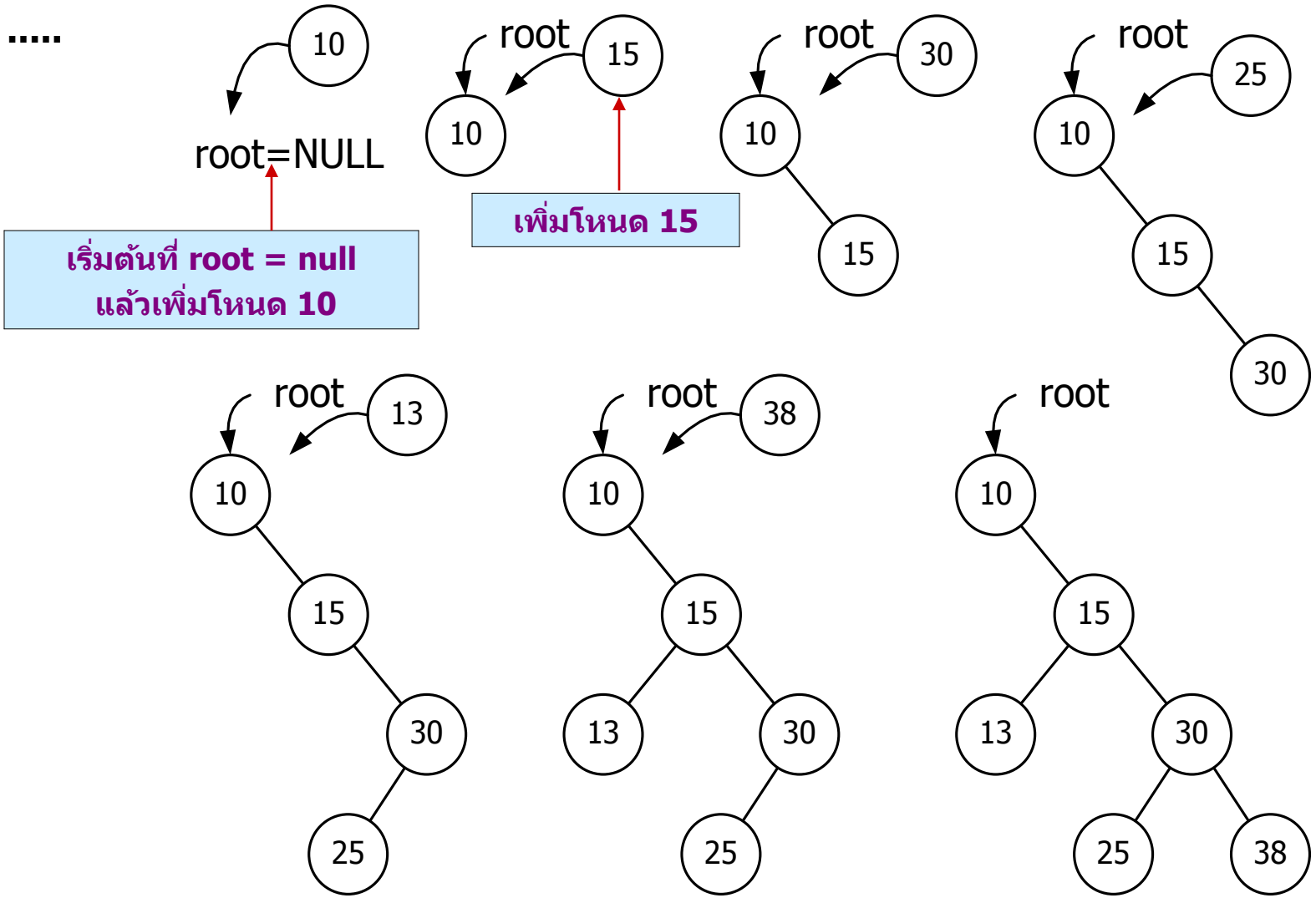
```
public void insertNewNode(long info)
{  Node current = null , parent = null ; // ตัวแปรสำหรับชี้ข้อมูล ไม่ต้องสร้าง Object
   Node newNode = new Node(info);   // สร้างโหนดเก็บข้อมูล info
   if  (root == null)  root = newNode ; // root คือตัวแปรในคลาส BinaryTree
   else {current = root ; //หาตำแหน่งโหนดสุดท้าย ที่จะเพิ่มโหนดลูก
          while (current !=null)
              { parent = current;  //จำโหนดตัวบน
               if (current.info > newNode.info)  //หาตำแหน่งซ้ายขวาโหนดลูก
                   current = current.left;
               else current = current.right;  }
         if   (parent.info  > newNode.info)  //เช็คว่าโหนดที่จะเพิ่ม ควรอยู่ซ้ายหรือขวา
               parent.left = newNode; //ปรับตัวชี้ไปยังโหนดลูก
        else parent.right = newNode;  }
}
```

Example Insert node (10 15 30 25 13 38)

ลักษณะของต้นไม้ที่เกิดขึ้น ขึ้นอยู่กับลำดับของโหนดที่ถูกเพิ่มเข้าไป

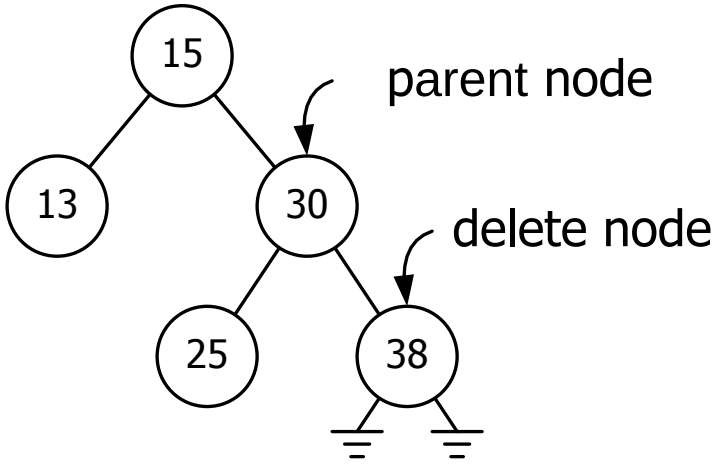
insertNewNode(10);

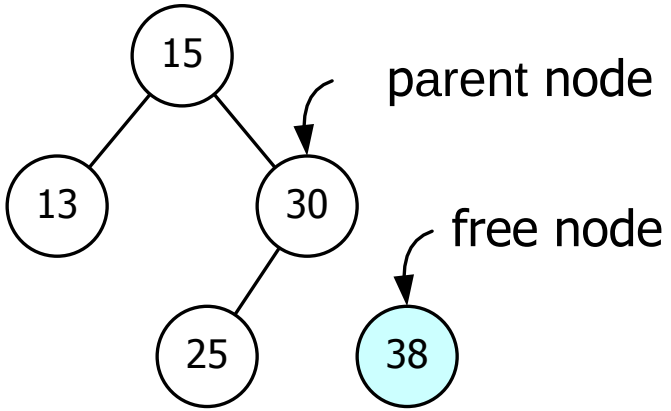
.....



Delete node examples

- การลบโหนด จะต้องรู้ตำแหน่งของโหนดแม่ก่อนเพื่อจัดการตัวชี้ของแม่ (ยกเว้น root)
- ลบลีฟโหนดที่ไม่มีลูก (Deleting a leaf node)
  - ลบโหนดสุดท้าย (leaf node) ทางขวา





parent = โหนด 30

current = parent.right;

current = โหนด 38

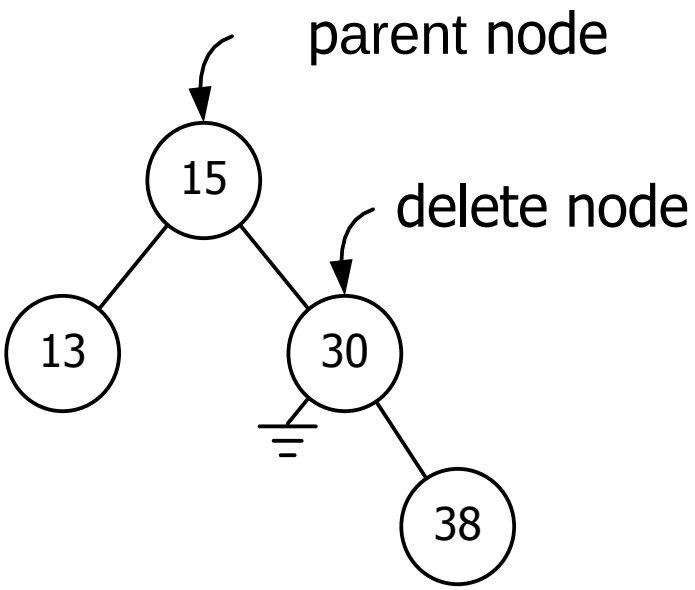
parent.right = null;

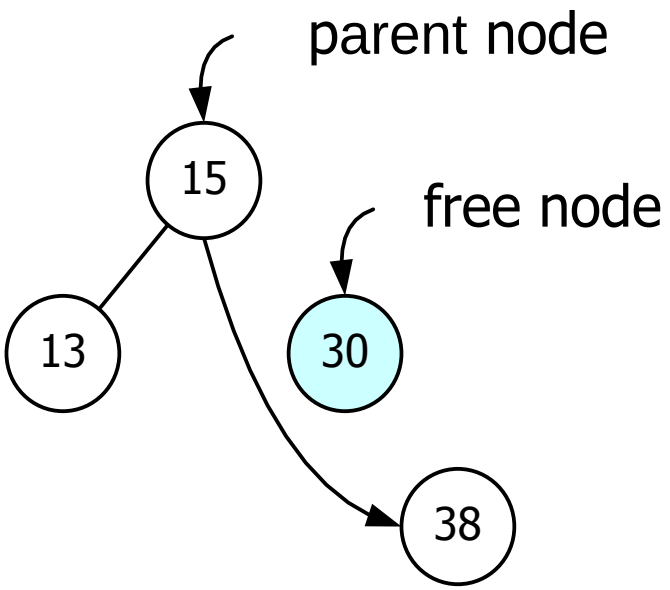
ลบทางขวาของ 30

ถ้าต้องการลบโหนดสุดท้ายทางซ้ายให้ทำกลับกัน

Delete node (continue)

- ลบโหนดที่มีโหนดลูกข้างเดียว
  - ลบโหนดทางขวา ที่มีลูกทางขวาอีก 1 ชุด (ไม่มีลูกทางซ้าย)





parent = โหนด 15

ถ้าต้องการลบโหนดทางขวาที่มีลูกทางซ้ายให้ทำกลับกัน

current = parent.right;

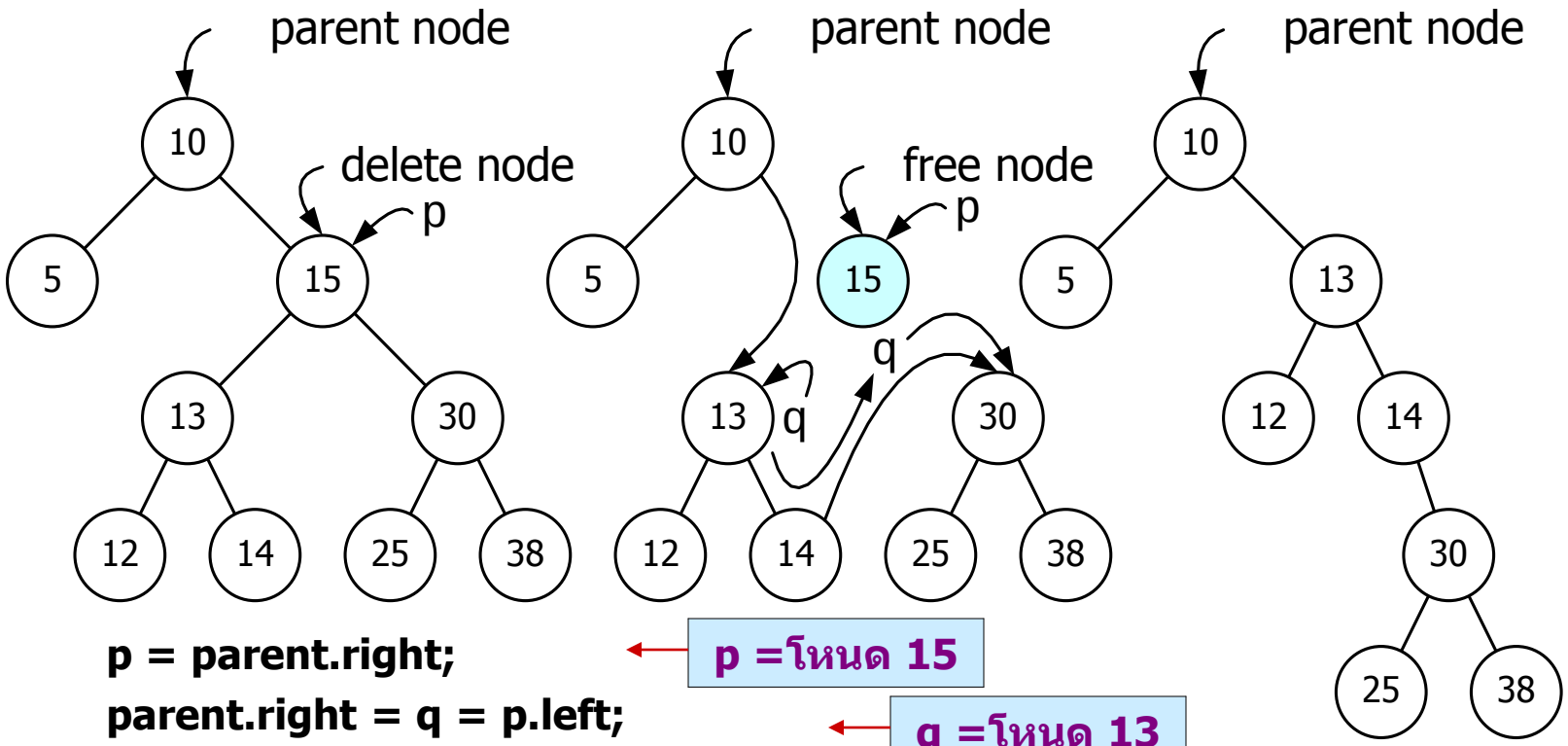
current = โหนด 30

parent.right = current.right;

ทางขวาของ 15 ชี้ไปทางขวาของโหนด 30 (38)

Delete node (continue)

- ลบโหนดที่มีโหนดลูก ทั้งซ้ายและขวา




```
p = parent.right;
parent.right = q = p.left;
while (q.right !=null) q = q.right ;
q.right = p.right;
```

p =โหนด 15

q =โหนด 13

หาโหนดทางขวาสุดของโหนด 13 (q = โหนด 14)

ทางขวาของ 14 ชี้ไปยัง โหนด 30

Traversing in binary trees

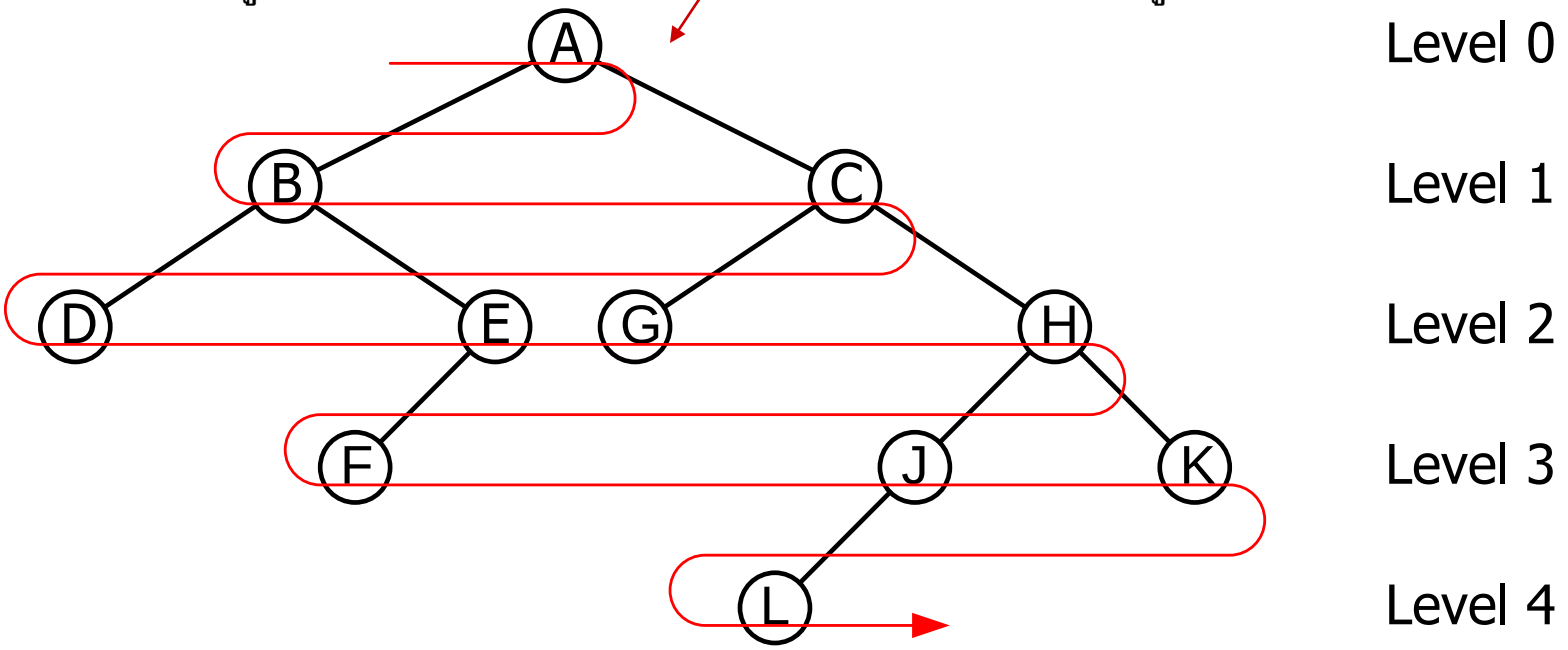
- **Breadth First Search**

  Level0 -> Level1 -> Level2 -> ...

  K = 1, 2, 3, 4 ...

- **วิธีการค้นหา**
  1. **เริ่มต้นที่ root โหนด**
  2. **Access โหนด**
  3. **นำโหนดทางซ้าย และ ขวา มา add ใส่ในคิวตามลำดับ(ถ้ามี)**
  4. **ดึงข้อมูลออกจากคิว และทำซ้ำในข้อ 2-3 จนกระทั่งข้อมูลหมดจากคิว**

ค้นตามแนวราบทีละตัวตามลำดับ
ใช้ queue เก็บสถานะการค้น



Breadth First : A B C D E G H F J K L

isEmpty()

Breadth First Traversing

```
void breadth_search_sequence(Node root)
{   Node  current = root;
    ArrayList <Node> Queue = new ArrayList <Node> ();
    while (current != null) {
     System.out.println("access node "+current. info);
      if (current.left != null)
            Queue.add(current.left);
      if (current.right != null)
            Queue.add(current.right);
     if Queue.Isempty()
        current = null;
     else
         current = Queue.get(0);
    }
}
```

- ตัวอย่างการเรียกใช้

  breadth_search_sequence(root);

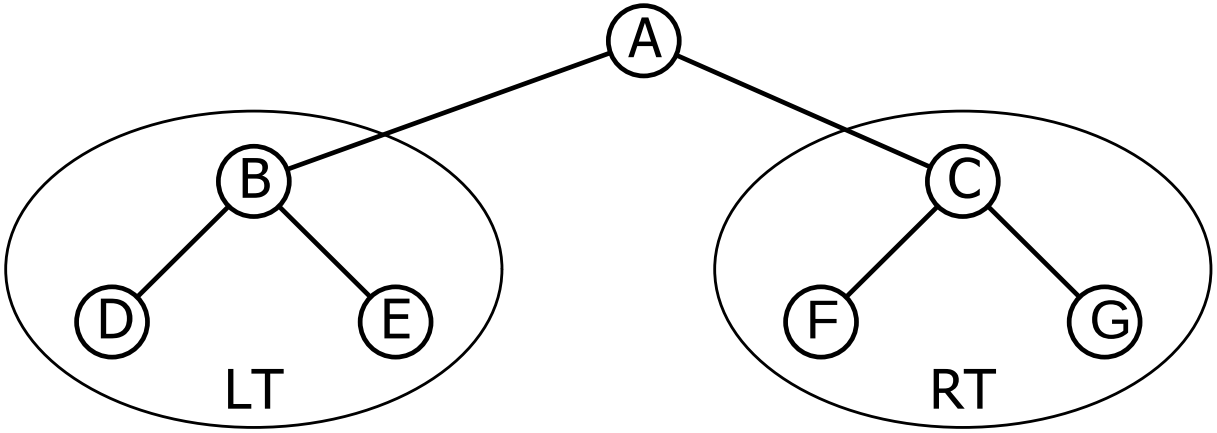
สร้าง queue ด้วย ArrayList สำหรับเก็บ address ของโหนด

นำลิงค์ของโหนดทางซ้าย และทางขวา มาสร้างโหนด(copy) ใส่เข้าไปในคิวตามลำดับ

ดึงโหนดใหม่ออกจากคิว จนกว่าจะหมด

Traversing in binary trees

- Depth First Search
  - Preorder: Root -> Left subtree -> Right subtree
  - Inorder: Left subtree -> Root -> Right subtree
  - Postorder: Left subtree -> Right subtree -> Root

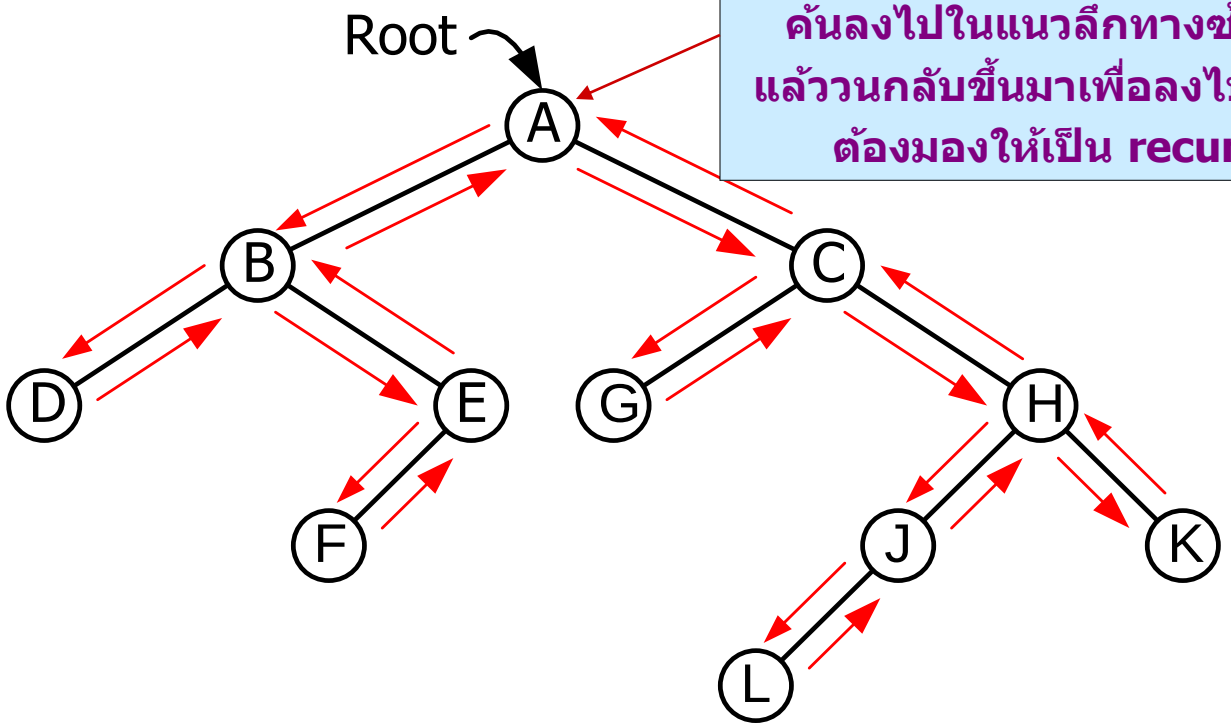


Preorder: A -> LT(B -> D -> E) -> RT(C -> F -> G)

Inorder: LT(D -> B -> E) -> A -> RT(F -> C -> G)

Postorder: LT(D -> E -> B) -> RT(F -> G -> C) -> A

Depth First Traversing



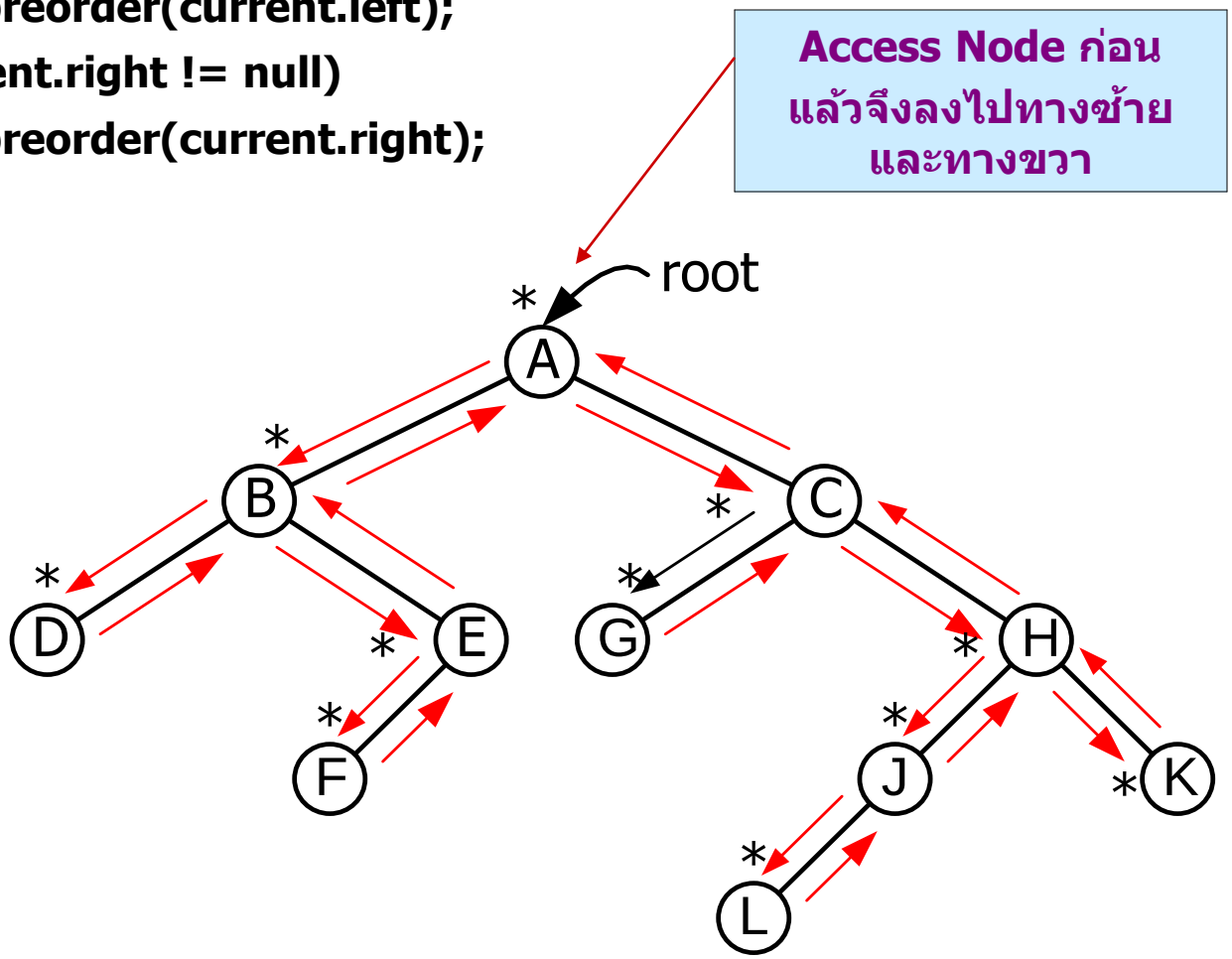
ค้นลงไปในแนวลึกทางซ้ายก่อน แล้ววนกลับขึ้นมาเพื่อลงไปทางขวา ต้องมองให้เป็น recursion

- **Preorder:** in ->(left) -> (right) — A B D E F C G H J L K
- **Inorder:** (left) -> in -> (right) — D B F E A G C L J H K
- **Postorder:** (left) -> (right) -> in — D F E B G L J K H C A

สามารถใช้ stack ช่วยในการจัดลำดับการเข้าถึงข้อมูลแทนการรีเคอร์ชันได้ แต่ต้องสร้างสถานะเพิ่มให้รู้ว่า โหนดใดสำรวจเสร็จแล้ว โหนดใดค้างอยู่ใน stack และโหนดใดที่ยังไม่เคยสำรวจ

Preorder Traversing

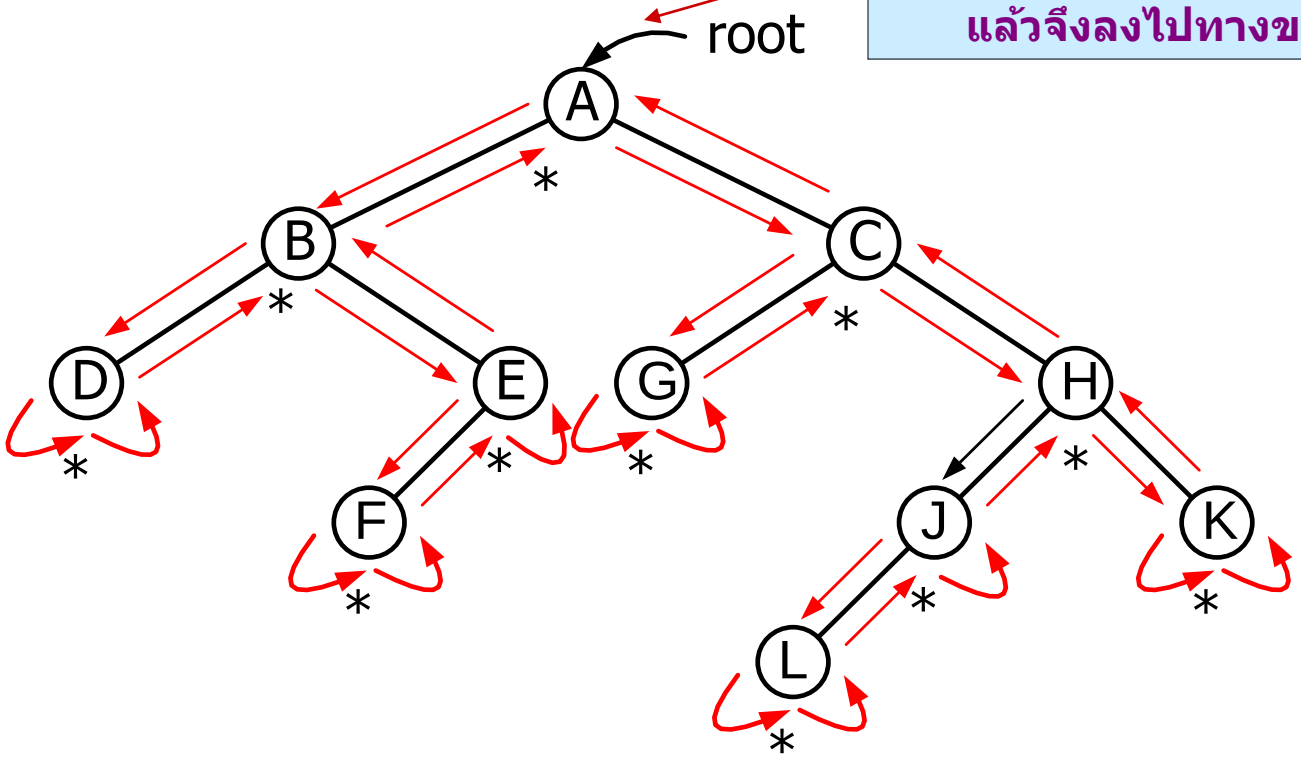
```
void preorder(Node current)
{  System.out.println("access node "+current.info);
   if (current.left != null)
          preorder(current.left);
   if (current.right != null)
          preorder(current.right);
}
```



Inorder Traversing

```
void inorder(Node current)
{ if (current.left != null)
        inorder(current.left);
   System.out.println("access node "+current. info);
   if (current.right != null)
        inorder(current.right);
}
```

ลงไปทางซ้ายสุดก่อน
แล้วกลับมา Access Node
แล้วจึงลงไปทางขวา



Postorder Traversing

```
void postorder(Node current)
{  if (current.left != null)
            postorder(current.left);
   if (current.right != null)
            postorder(current.right);
   System.out.println("access node "+current. info);
}
```

ลงไปทางซ้ายและขวาสุดก่อน
แล้วกลับมา Access Node หลังสุด