การอบรมคอมพิวเตอร์โอลิมปิกค่าย 2

- Recursion
- Tree
- Binary Tree, Binary Search Tree

Recursion

Recursion

นิยาม

- **Recursion** คือการกำหนด object ใด ๆ ในเทอมที่ง่ายกว่าของตัวเอง โดยการทำงานจะต้องมีจุดจบที่แน่นอน

ตัวอย่าง Factorial, การคูณเลขสองจำนวน (Multiplication of natural number), Fibonacci sequence, Binary search

1. การหาค่า Factorial

5 factorial = 5 * 4 * 3 * 2 * 1 = 120

3 factorial = 3 * 2 * 1 =

0 factorial = 1

เราสามารถเขียนนิยามของ Factorial โดยสรุปได้ดังนี้

if n = 0 => n! = 1

if n > 0 => n! = n * (n-1) * (n-2) * ...* 1

การหาค่า Factorial แบบวน loop

รับค่าจำนวนเต็ม *n* และ return ค่า *n!*

```
int  iterativeFactorial (int n )
{
      int  prod  = 1,  x;
      for ( x = n; x > 0; x--)
            prod  = prod * x;
      return(prod);
}
```

เราสามารถเขียนนิยามของ factorial ใดๆ ใหม่ได้ดังนี้

$n! = 1$	if $n == 0$
$n! = n * (n-1)!$	if $n > 0$

Examples :

$0! = 1$	$3! = 3 * 2!$
$1! = 1 * 0!$	$4! = 4 * 3!$
$2! = 2 * 1!$	

$5! = 5 * 4!$

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1!$

$1! = 1 * 0!$

$0! = 1$

1

2

6

24

120

การหาค่า factorial แบบ recursive

รับตัวเลขจำนวนจำนวนเต็ม *n* and return ค่า *n!*

```
int recursiveFactorial ( int n )    {

    int  result;

    if (n == 0)
        return 1;
    else
        result = n * recursiveFactorial (n -1)
        return ( result);
}
```

2. การหาค่าผลคูณของเลข 2 จำนวน

- การหาค่าผลคูณแบบวน loop

ผลคูณของ a และ b จะเท่ากับการบวก a ทั้งหมด b ครั้ง

```
int mult (int a, int b)
{
   int sum = 0, i;
   for (i = 1; i <= b; i++ )
        sum  = sum +  a;
    return(sum)
}
```

- การหาค่าผลคูณ แบบ recursive

$$a * b = a \qquad \text{if } b == 1$$

$$a * b = a * (b - 1) + a \qquad \text{if } b > 1$$

```
int mult (int a, int b)
{
    int result;

    if (b ==1)
            return(a);
    else  {
            result = mult(a, (b-1)) + a;
            return(result);
    }
}
```

สรุปหลักการเขียนอัลกอริทึมแบบ recursive

- พยายามหา case ที่ง่ายที่สุดซึ่งจะเป็นจุดจบของการทำงานแบบ recursive เช่น

  factorial => 0! = 1

  multiple => $a * 1 = a$

- พยายามเขียนกรณีอื่น ๆ ในรูปแบบเดียวกับตัวเองแต่เป็นกรณีที่ง่ายกว่าและมีโอกาสที่จะถึงกรณีที่ง่ายที่สุด

  factorial => continue (n -1) until equal 0

  multiply => continue (b-1) until equal 1

3. การหาค่าลำดับของ Fibonacci

Fibonacci sequence คือ ชุดของเลขจำนวนเต็มที่ค่าในลำดับใด ๆ จะมีค่าเท่ากับค่าในลำดับสองตัวก่อนหน้ารวมกัน ยกเว้นในลำดับที่ 0 และ 1

<u>Example</u>

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

0+1

1+1

…… 2+3 8+13

การหาค่าของลำดับ Fibonacci แบบ recursive

กำหนดให้ fib(0) = 0, fib(1) = 1

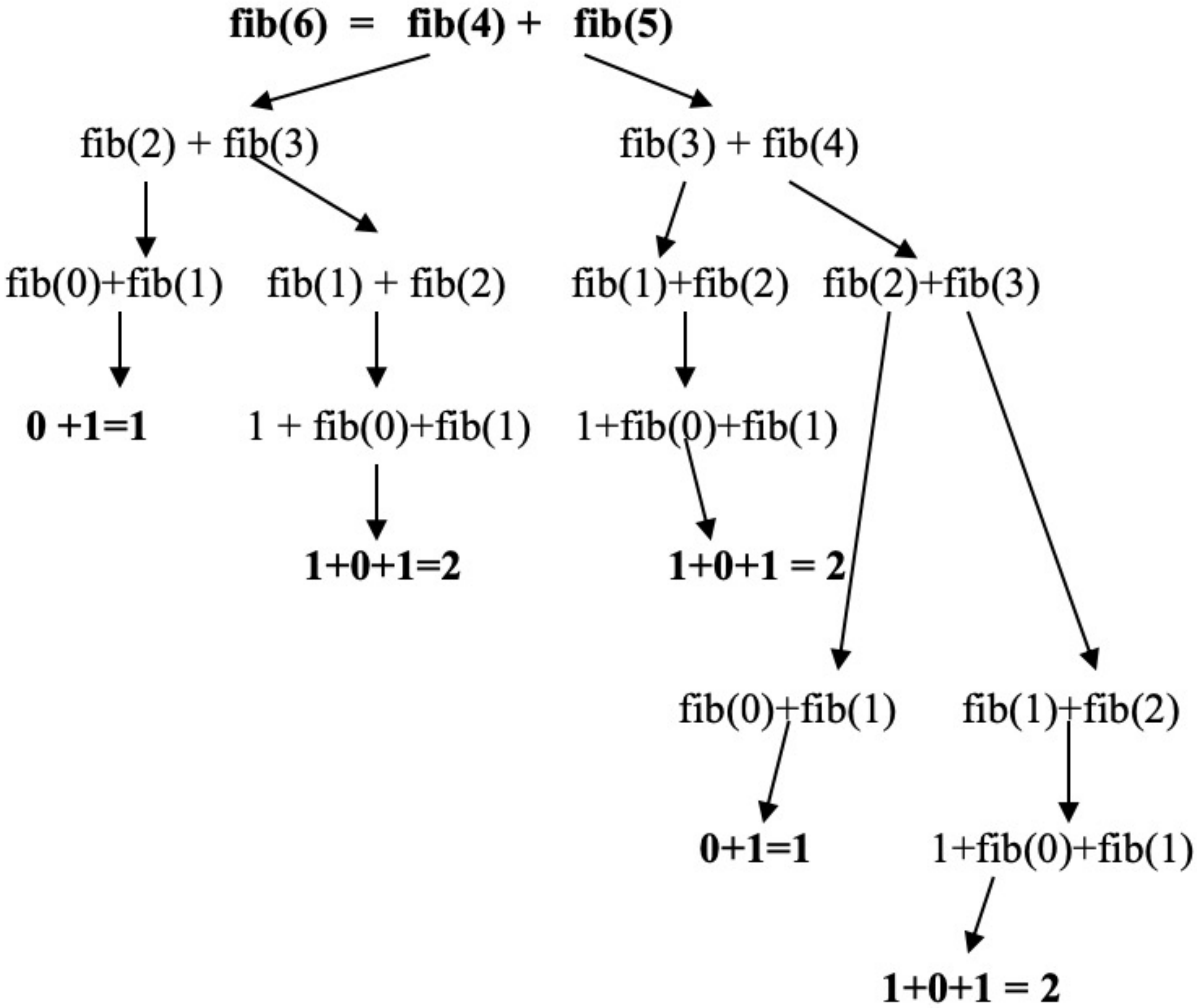
fib(0) = 0 and fib(1) = 1

fib(n) = fib(n-2) + fib(n-1) if n >= 2

```
int fib (int n)
{
      int result;

      if (n <= 1)
            return ( n );
      result = fib(n-1) + fib(n - 2);
      return(result)
}
```

ตัวอย่าง


fib(6) = fib(4) + fib(5)
fib(2) + fib(3)
fib(3) + fib(4)
fib(0)+fib(1)
fib(1) + fib(2)
fib(1)+fib(2)
fib(2)+fib(3)
0 +1=1
1 + fib(0)+fib(1)
1+fib(0)+fib(1)
1+0+1=2
1+0+1 = 2
fib(0)+fib(1)
fib(1)+fib(2)
0+1=1
1+fib(0)+fib(1)
1+0+1 = 2

การหาค่าของลำดับ Fibonacci แบบวน loop

```
int  fib( int n )
{
    int x, i, lofib, hifib;

    if  ( n <= 1 )
        return(n);
    lofib  =  0;
    hifib  =  1;
    for (i = 2; i <= n; i++ )  {
        x  =  lofib;
        lofib  =  hifib;
        hifib  =  x + lifib;
    }
    return(hifib);
}
```

Example fib(6)

initial :	lofib = 0, hifib = 1	
i = 2 :	x = 0; lofib = 1;	
	hifib = 0 + 1 = 1	fib(2)
i = 3 :	x = 1; lofib = 1;	
	hifib = 1 + 1 = 2	fib(3)
i = 4 :	x = 1; lofib = 2;	
	hifib = 1 + 2 = 3	fib(4)
i = 5 :	x = 2; lofib = 3;	
	hifib = 2 + 3 = 5	fib(5)
i = 6 :	x = 3; lofib = 5;	
	hifib = 3 + 5 = 8	fib(6)

4. การค้นหาแบบไบนารี (Binary Search)

- Binary Search จะใช้กับข้อมูลอินพุตที่ได้เรียงลำดับเรียบร้อยแล้ว

<u>**ตัวอย่าง**</u> ถ้าต้องการหาว่ามีข้อมูลเลข 9 อยู่ใน array ของตัวเลขข้างล่างนี้หรือไม่

1 2 3 4 5 6 7 8 9 10

- ถ้าใช้การเปรียบเทียบแบบธรรมดาจะต้องทำการเปรียบเทียบหลายครั้ง

- การใช้การค้นหาแบบ binary จะช่วยให้จำนวนครั้งของการเปรียบเทียบน้อยลง เพราะการเปรียบเทียบแต่ละครั้งสามารถตัดข้อมูลกึ่งหนึ่งออกจากการพิจารณาได้ ทำให้ลดจำนวนข้อมูลที่จะต้องทำการเปรียบเทียบลงมาก

<u>ตัวอย่าง</u> การค้นหาข้อมูล 17 ใน array *a*

0	1	2	3	4	5	6	7
1	**3**	**4**	**5**	**17**	**18**	**31**	**33**

low=0 high=7

mid=(0+7)/2=3

17 > *a*[3]

low = 4 high=7

mid=(4+7)/2=5

17 < *a*[5]

low=high=4

mid=(4+4)/2 =4

17 = *a*[4] ***found!***

<u>ตัวอย่าง</u> การค้นหาข้อมูล 2 ใน array *a*

0	1	2	3	4	5	6	7
1	**3**	**4**	**5**	**17**	**18**	**31**	**33**

low=0 high=7

mid=(0+7)/2=3

2 < *a*[3]

low = 0 high=2

mid=(0+2)/2=1

2 < *a*[1]

low=high=0 => mid = 0

low=1 high=0 => return (-1) ***not found!***

การค้นหาแบบไบนารี แบบ recursive

สำหรับการค้นหาข้อมูล x ใน array *a* ที่มีข้อมูลที่เรียงลำดับแล้วจาก *a*[low] ถึง *a*[high]

```
int binsrch (int a[], int x, int low, int high)
{
    int  mid;
    if ( low > high)
        return(-1);
    mid = (low + high) / 2;
    if (x == a[mid])
        result = mid;
    else if ( x < a[mid])
            result = binsrch(a, x, low, mid-1);
    else
            result = binsrch(a, x, mid+1,high);
    return(result);
}
```

คุณสมบัติของอัลกอริทึมแบบ recusive

- จะต้องมีการเรียกตัวเองแบบมีจุดจบ
- จะต้องประกอบไปด้วยส่วนที่สามารถเขียนให้อยู่ในรูปที่สามารถเรียกตัวเองได้ แต่ต้องมีอย่างน้อยส่วนหนึ่งที่ไม่ได้เขียนในรูปแบบเดียวกับตัวเองหรือไม่ได้เรียกตัวเอง ส่วนนั้นจะเป็นส่วนที่เป็นทางออกของ recursive เช่น

  factorial => 0! = 1

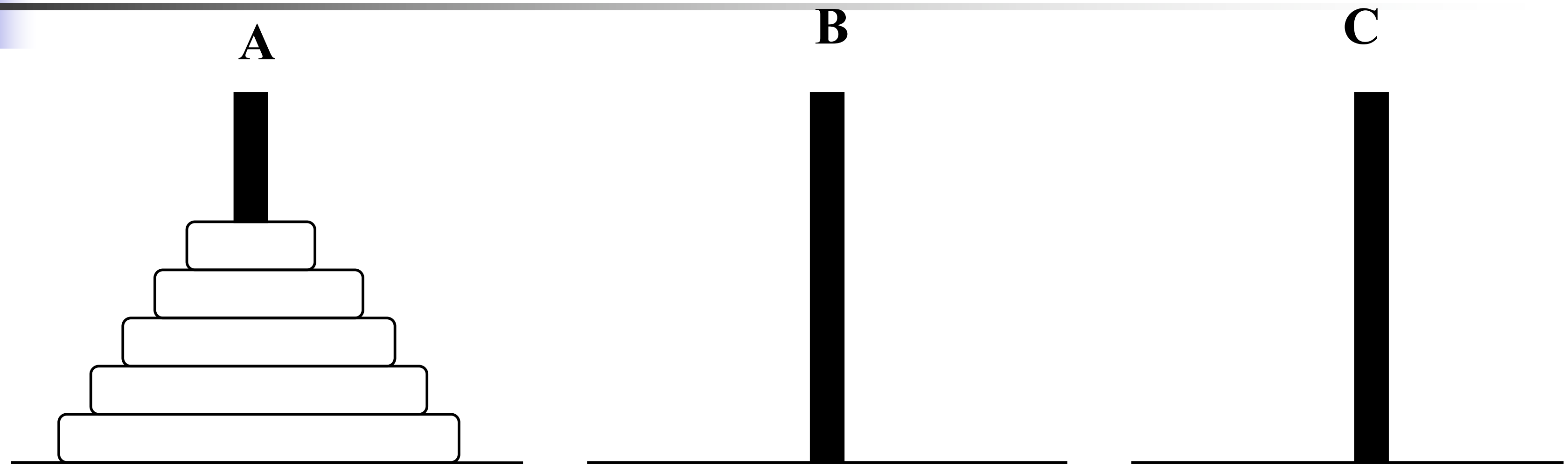
  multiple => a * 1 = a

  fibonacci seq : fib(0) = 0; fib(1) = 1

  binary search : if (low>high) return(-1);

  if (x==a[mid]) return(mid);

The Tower of Hanoi Problem

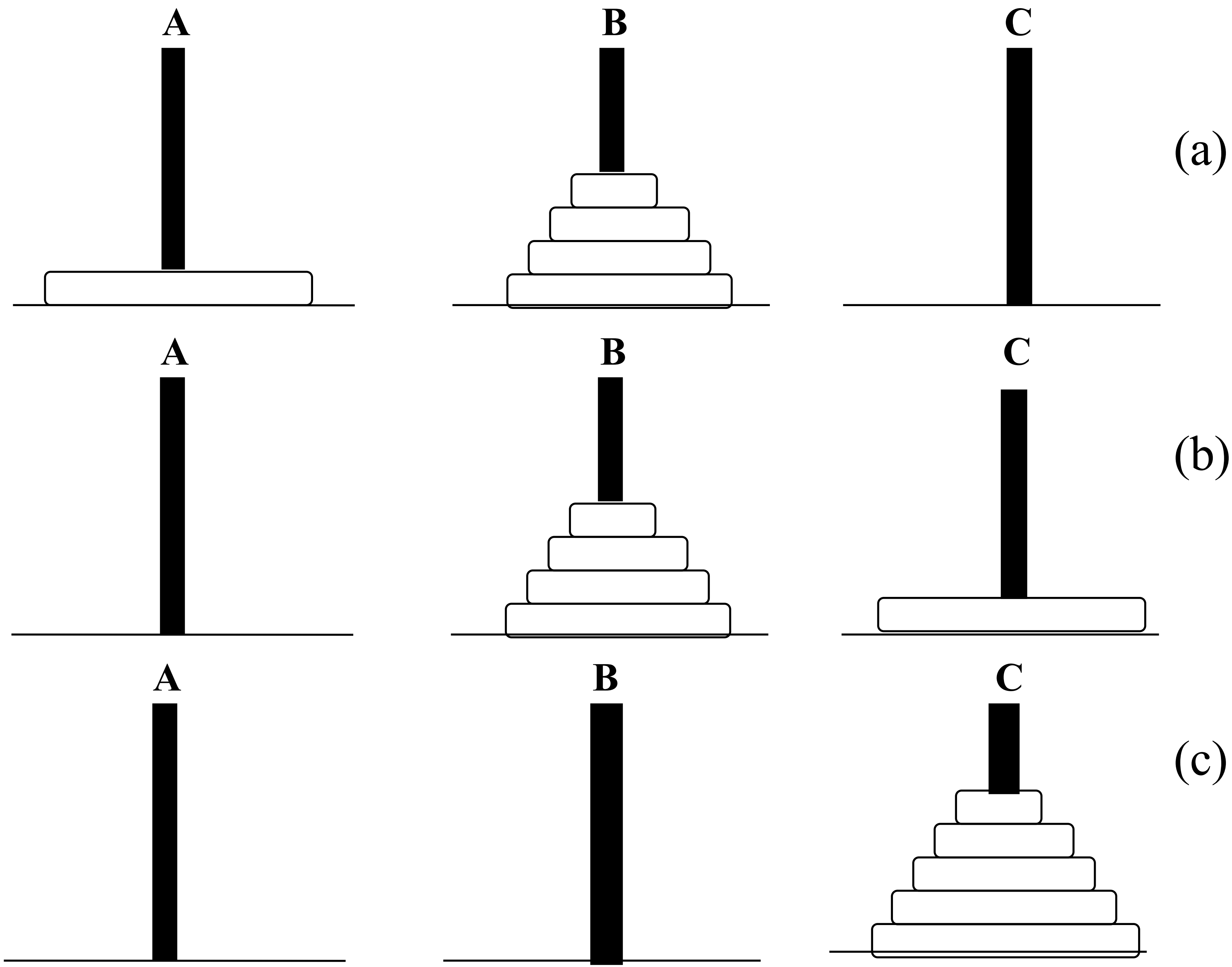


สมมติมีเสาทั้งหมด 3 ต้นให้ชื่อว่า ต้น A, B และ C และมีจานทั้งหมด 5 ใบ ที่มีเส้นผ่าศูนย์กลางต่างกันวางซ้อนกันอยู่บนเสาต้นแรก โดยให้จานใบใหญ่กว่าจะอยู่ใต้ใบที่เล็กกว่าเสมอ ดังรูป

The Tower of Hanoi Problem

ปัญหา: เราต้องการย้ายจานทั้ง 5 ใบจากต้น A ไปต้น C โดยใช้เสาต้น B เป็นต้นพัก โดยมีเงื่อนไขในการย้าย ดังนี้

- ในการย้ายครั้งหนึ่งๆ เราสามารถย้ายได้เฉพาะจานใบบนสุดเท่านั้น
- จานใบใหญ่กว่าไม่สามารถอยู่บนจานที่เล็กกว่า


A
B
C
(a)
A
B
C
(b)
A
B
C
(c)

<u>การแก้ปัญหาด้วยวิธี recursion:</u>

ถ้าสามารถแก้ปัญหาสำหรับจาน n-1 ใบได้ ก็สามารถแก้ปัญหาจาน n ใบได้ พยายามเขียนปัญหา n ใบให้อยู่ในรูปของปัญหา n-1 ใบ

<u>ขั้นตอนการแก้ปัญหา:</u> ให้ n แทนจำนวนจาน

1) ถ้า n = 1 ย้ายจานใบเดียวนี้จากเสาต้น A ไปต้น C

2) ถ้า n > 1

   ย้าย n-1 ใบจากเสาต้น A ไปต้น B โดยใช้เสาต้น C เป็นต้นพัก

   ย้ายใบที่เหลือจากเสาต้น A ไปรอไว้ที่ต้น C

   ย้าย n-1 ใบจากเสาต้น B ไปต้น C โดยใช้เสาต้น A เป็นต้นพัก

   ฉะนั้น Base case สำหรับกรณีนี้คือเมื่อ n = 1

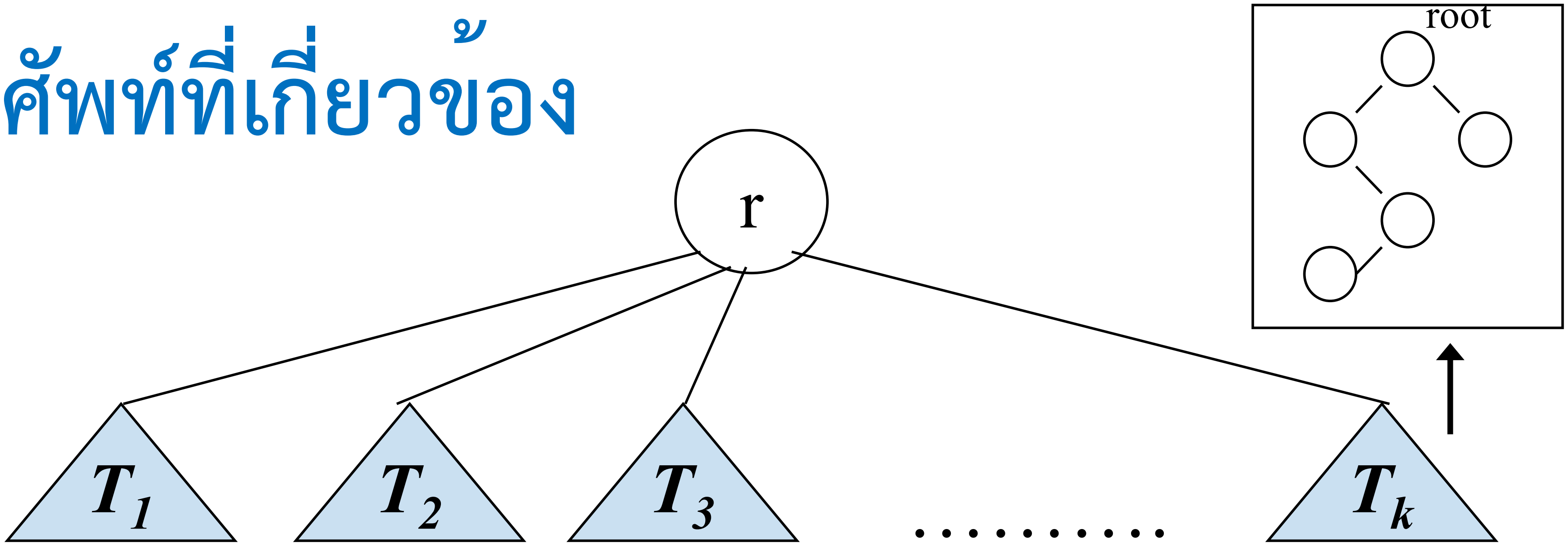
```
void towers(int n, char frompeg, char topeg, char auxpeg)
{
    // If only one disk, make the move and return
    if  (n == 1)  {
        cout << “move disk 1 from peg ” <<  frompeg,  << “ topeg ”<< topeg;
        return;
    }
     // move top n-1 disks from A to B using C as auxiliary
    towers(n-1, frompeg, auxpeg, topeg);
    // move remaining disk from A to C
    cout << “move disk” << n << “ from peg ” << frompeg << “ to peg ”<<  topeg;
    // move n-1 disk from B to C using A as auxiliary
    towers(n-1, auxpeg, topeg, frompeg);
}
```

ต้นไม้ (Tree)

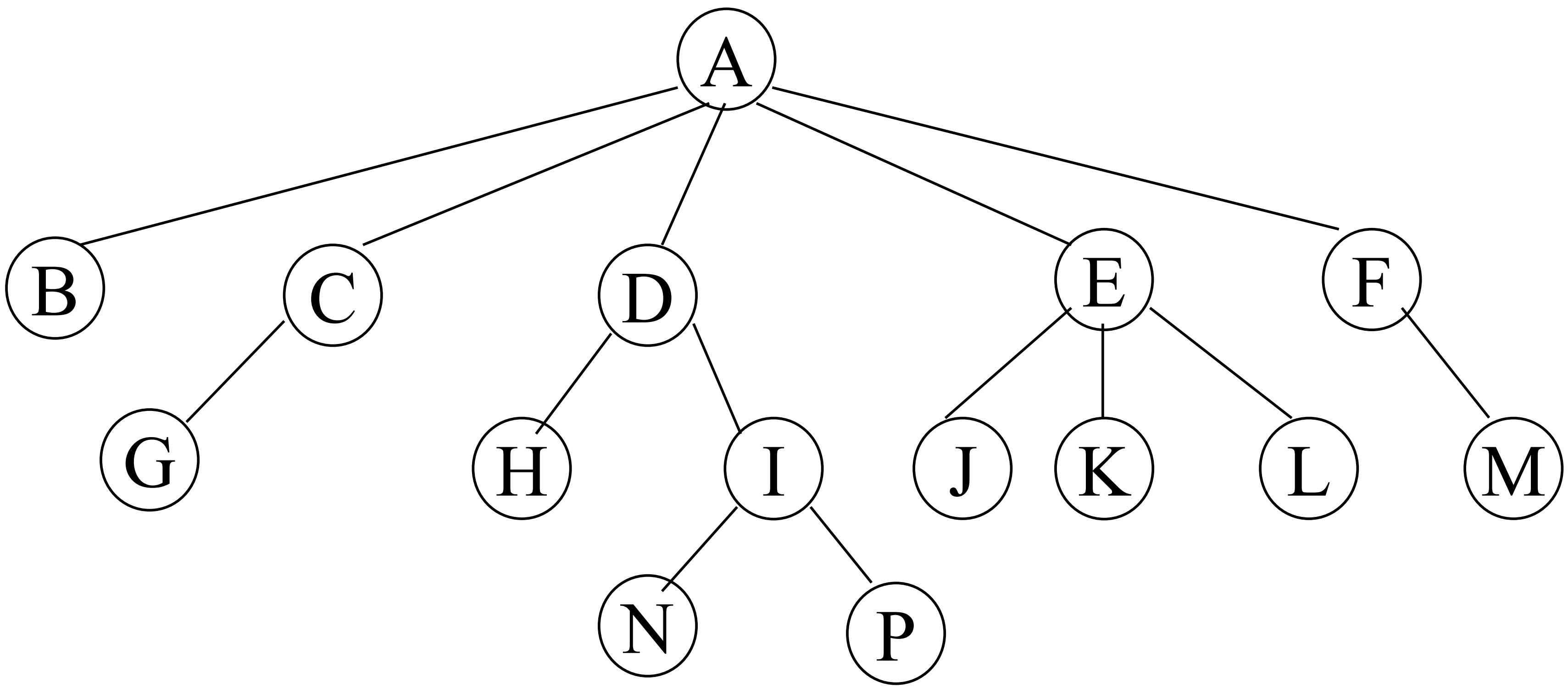
ต้นไม้ (Tree)

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

ศัพท์ที่เกี่ยวข้อง

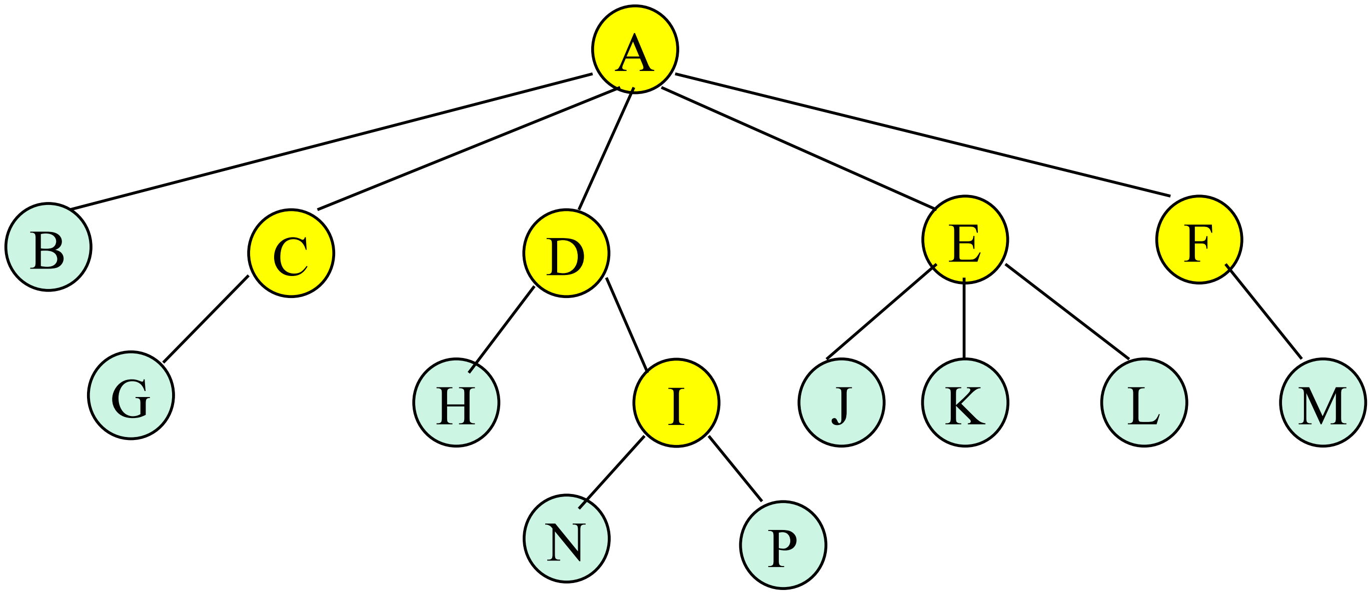


- โหนด *r* เป็น root node ของต้นไม้
- root node ของทุกๆต้นไม้ย่อยถือเป็น children ของโหนด *r* หรือ โหนด *r* เป็น parent ของ root nodes ของต้นไม้ย่อย
- ทุกๆ โหนดในต้นไม้ยกเว้น root node มีโหนด parent 1 โหนด
- สามารถกำหนดความสัมพันธ์ของโหนดอื่นๆ เช่นเดียวกับความสัมพันธ์ของครอบครัวคือ คือ โหนดลูก (child)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

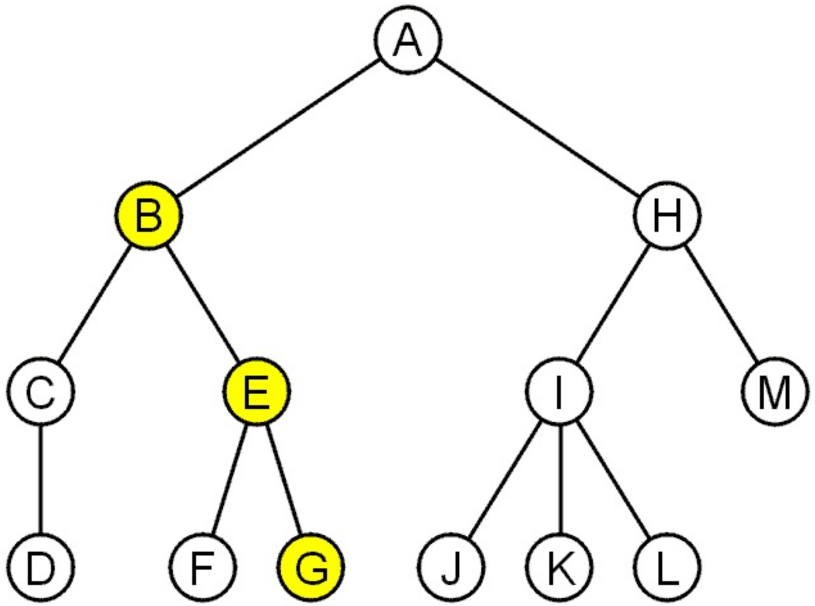
ศัพท์ที่เกี่ยวข้อง



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

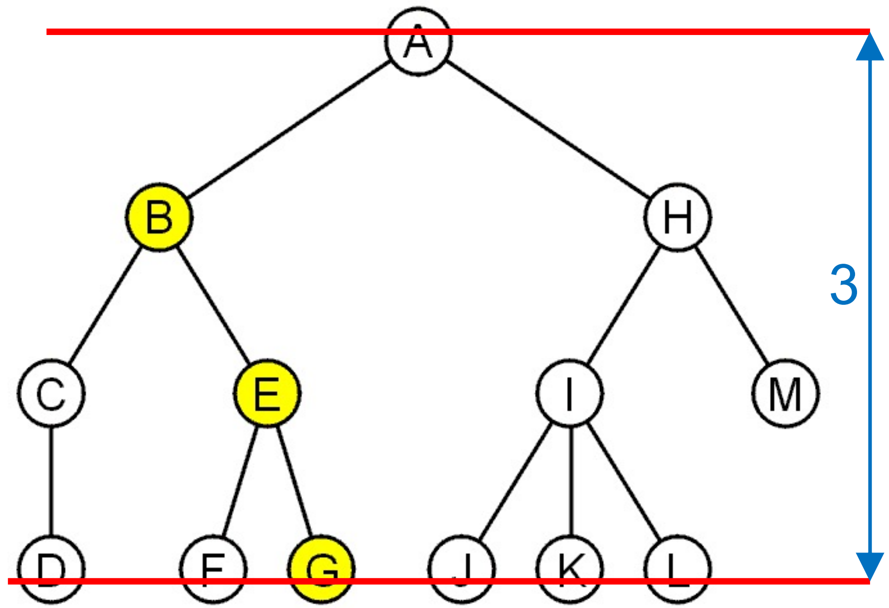
ศัพท์ที่เกี่ยวข้อง

- Path เป็นลำดับของโหนด (a_0, a_1, ..., 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



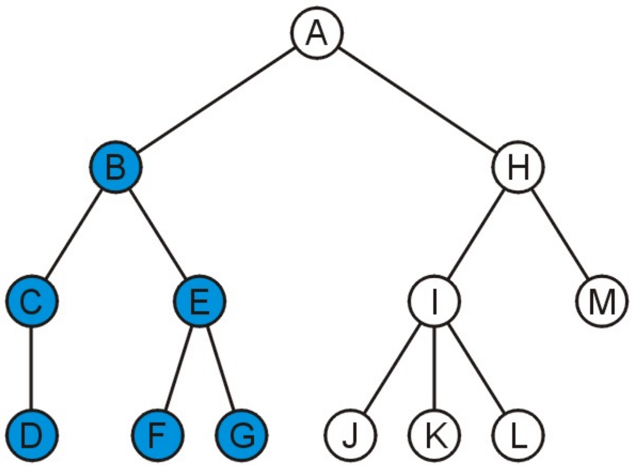
ศัพท์ที่เกี่ยวข้อง

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

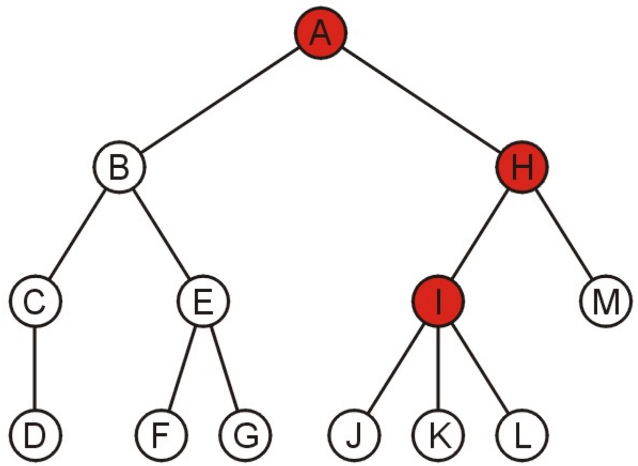


ศัพท์ที่เกี่ยวข้อง

- 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



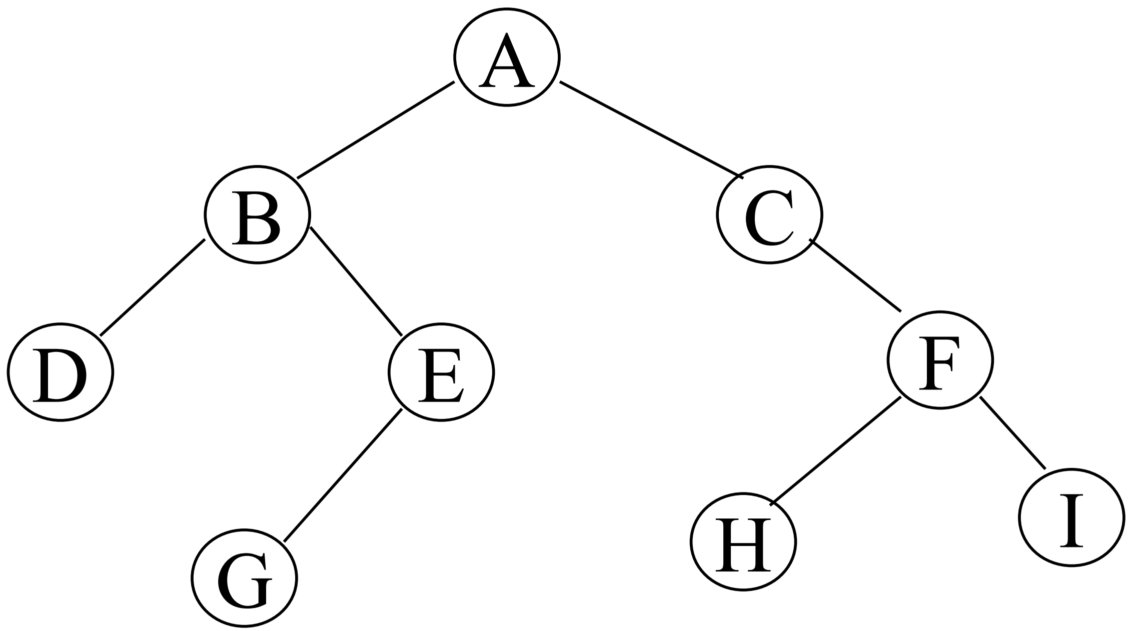
Descendants –ของโหนด B คือ B, C, D, E, F, และ G



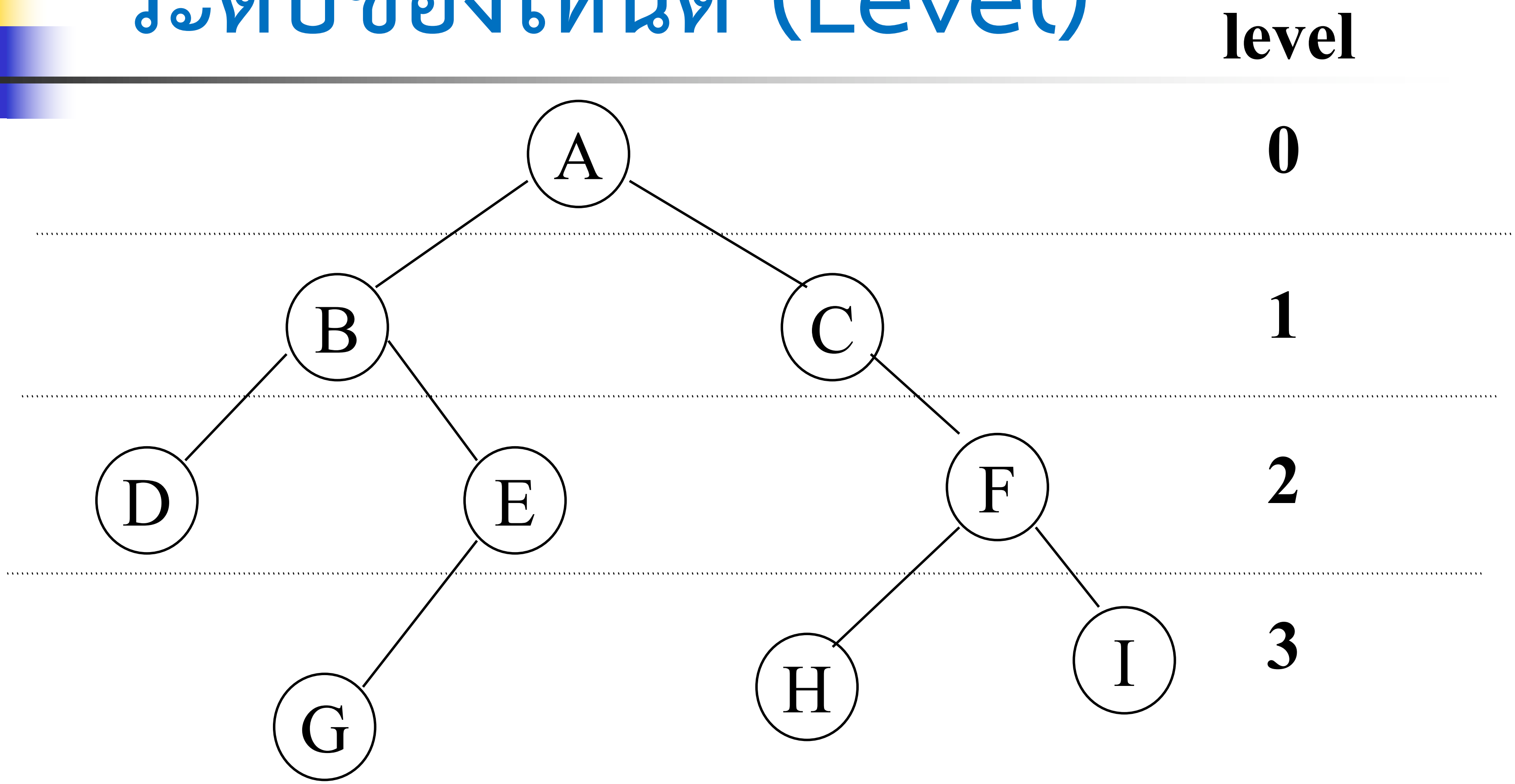
Ancestorsของโหนด I คือ I, H, และ A

ต้นไม้ไบนารี (Binary Trees)

ต้นไม้ไบนารีคือ ต้นไม้ที่ทุกโหนดมีลูกได้ไม่เกิน 2 โหนด อาจไม่มีลูกเลยก็ได้ หรือมีลูกหนึ่งโหนดหรือมีลูกสองโหนด



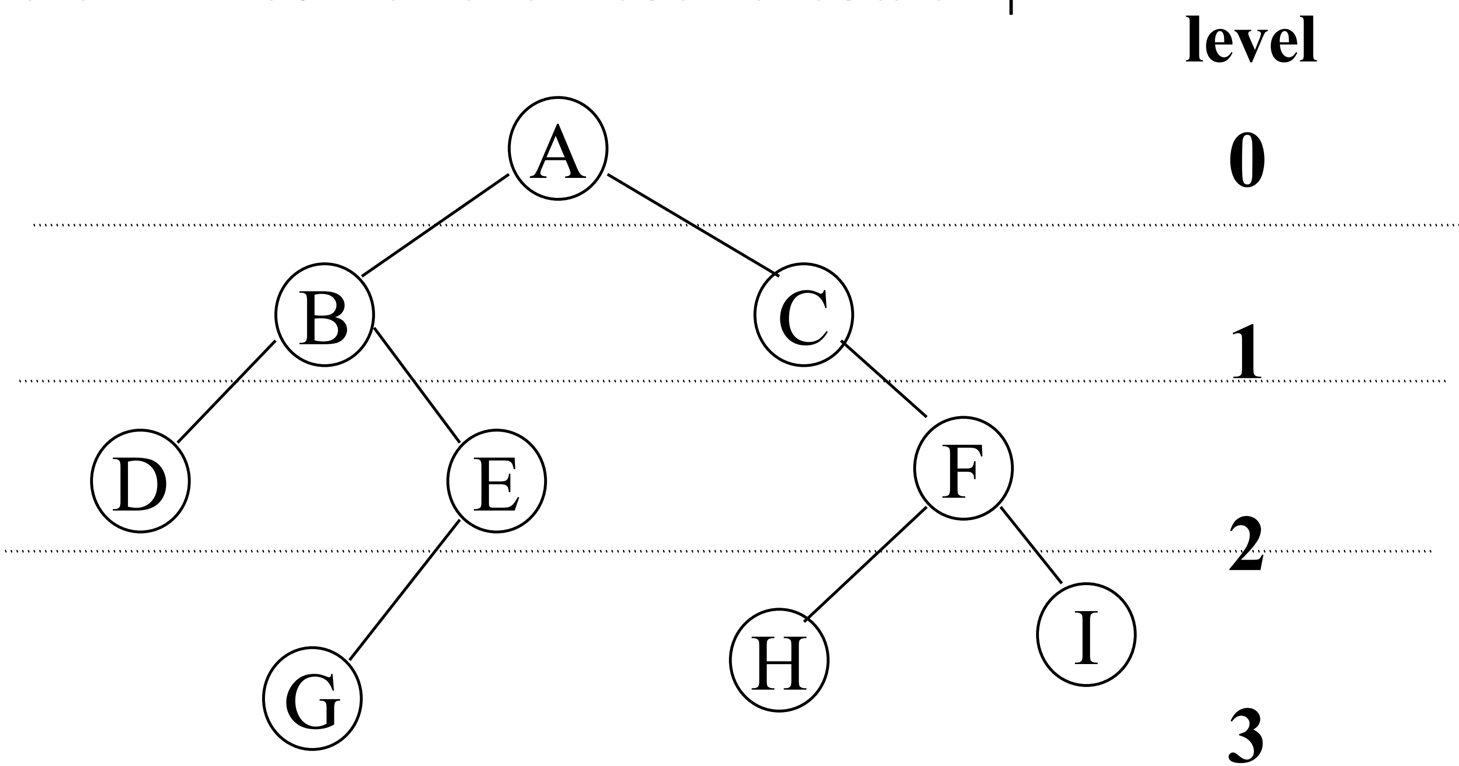
ระดับของโหนด (Level)



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

ความลึก (Depth)

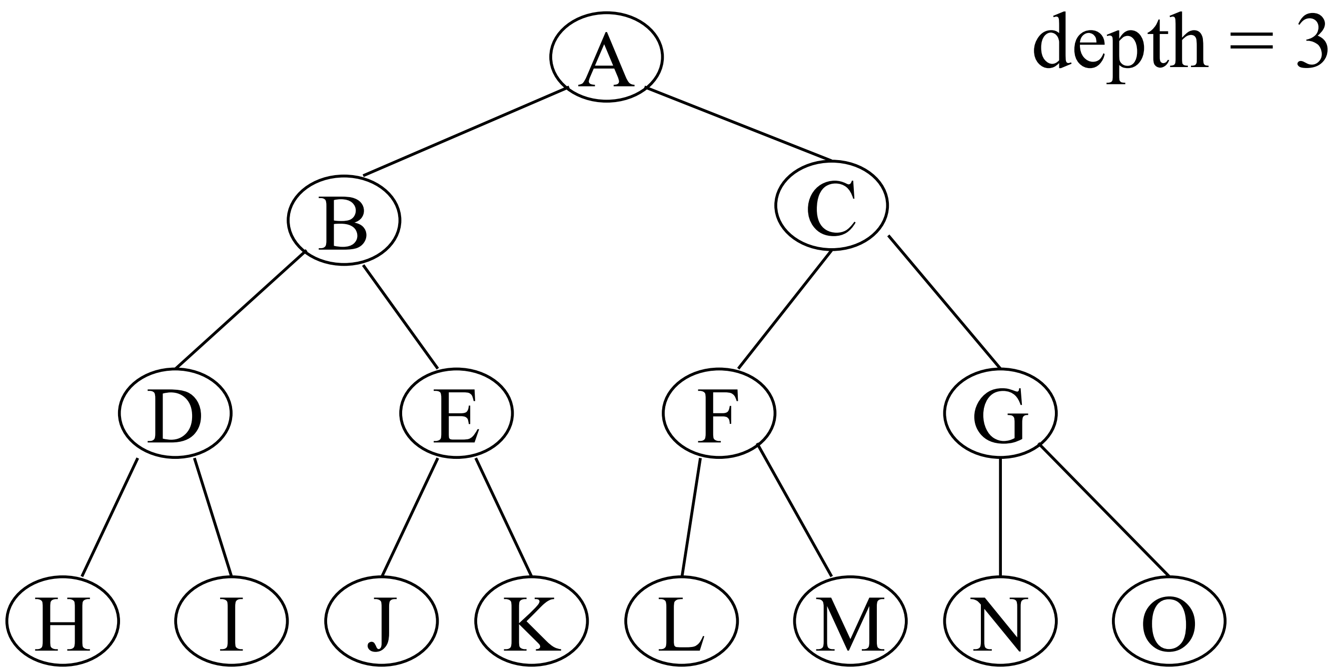
ความลึก คือค่าระดับที่สูงที่สุดของโหนดใบไม้ใดๆ ในต้นไม้ หรือความยาวสูงสุดของ เส้นทางเดินจากโหนดรากไปโหนดใบไม้ใดๆ



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 2L 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

$$
\begin{aligned}
tn &= 2^{d+1} - 1 \\
tn + 1 &= 2^{d+1} \\
\log_2 (tn + 1) &= d + 1 \\
d &= \log_2(tn + 1) - 1
\end{aligned}
$$

ตัวอย่าง

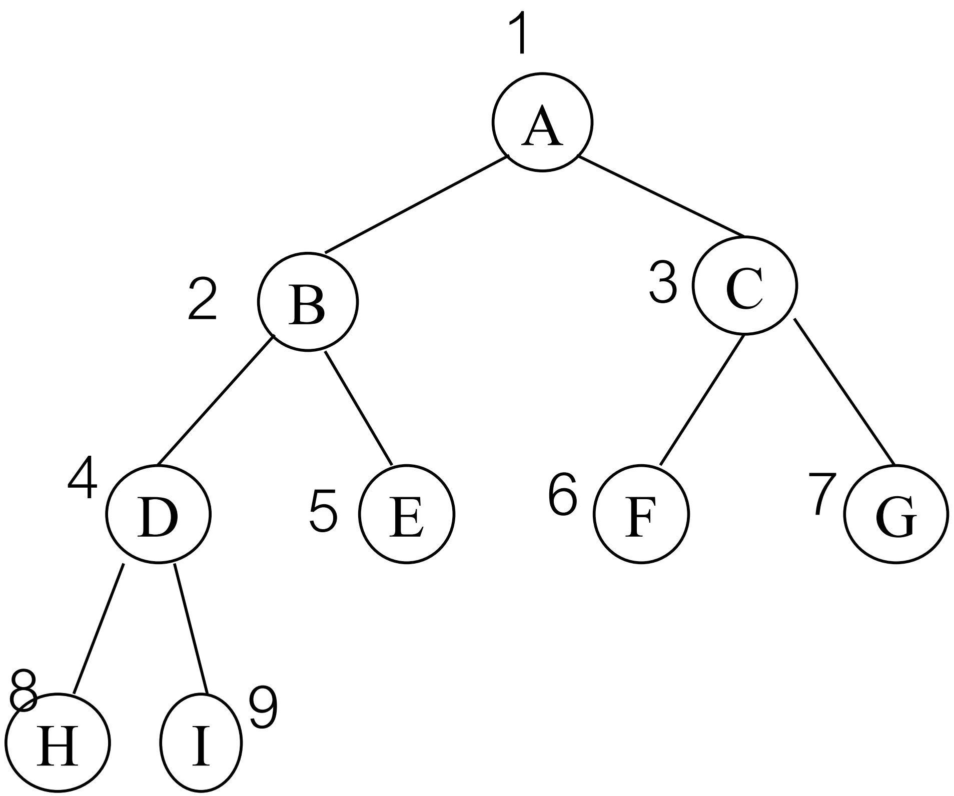
ต้นไม้ไบนารีที่สมบูรณ์มีจำนวนโหนดทั้งสิ้น 15 โหนด ต้นไม้ต้นนี้มีความลึกเท่าใด

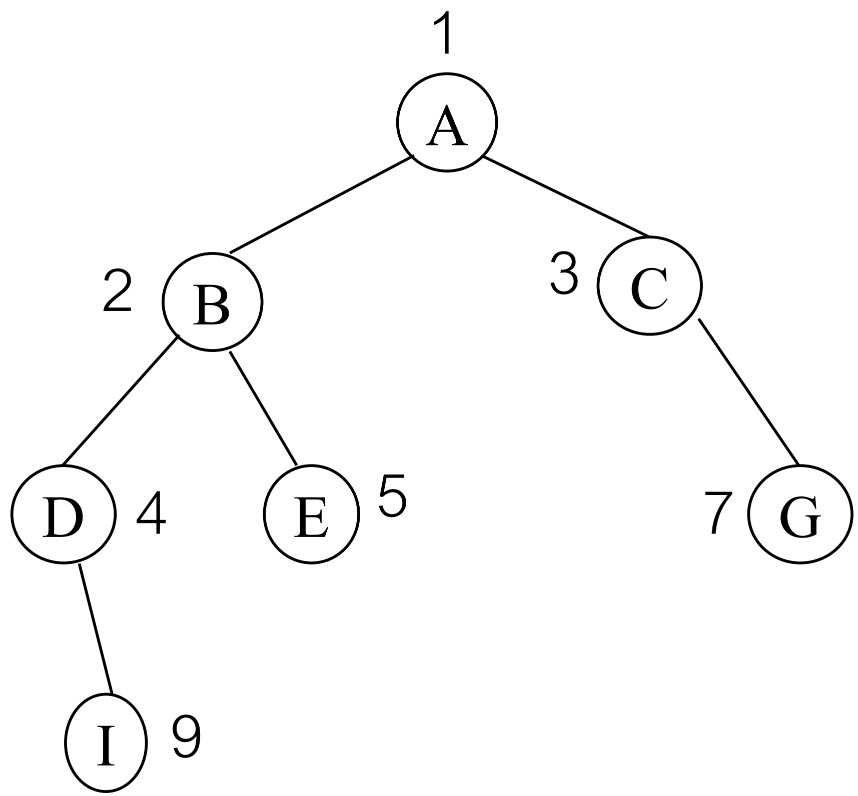
$$
\begin{aligned}
d &= \log_2 (tn + 1) - 1 \\
&= \log_2 (15 + 1) - 1 \\
&= 3 \text{ ระดับ}
\end{aligned}
$$

ตำแหน่งของโหนด

- โหนดราก (root node) มีเลขประจำตำแหน่งคือ 1
- โหนดที่เป็นลูกทางซ้าย จะมีค่าตำแหน่งเป็นสองเท่าของ ตำแหน่งของโหนดผู้ปกครอง
- โหนดที่เป็นลูกทางขวาจะมีค่าตำแหน่งเป็นสองเท่าบวก หนึ่ง ของโหนดผู้ปกครอง

ตัวอย่างการให้ค่าตำแหน่งของโหนดในต้นไม้ไบนารี





การเข้าถึงข้อมูลในต้นไม้

การเข้าถึงข้อมูลในต้นไม้แบบไบนารี (Tree Traversal)

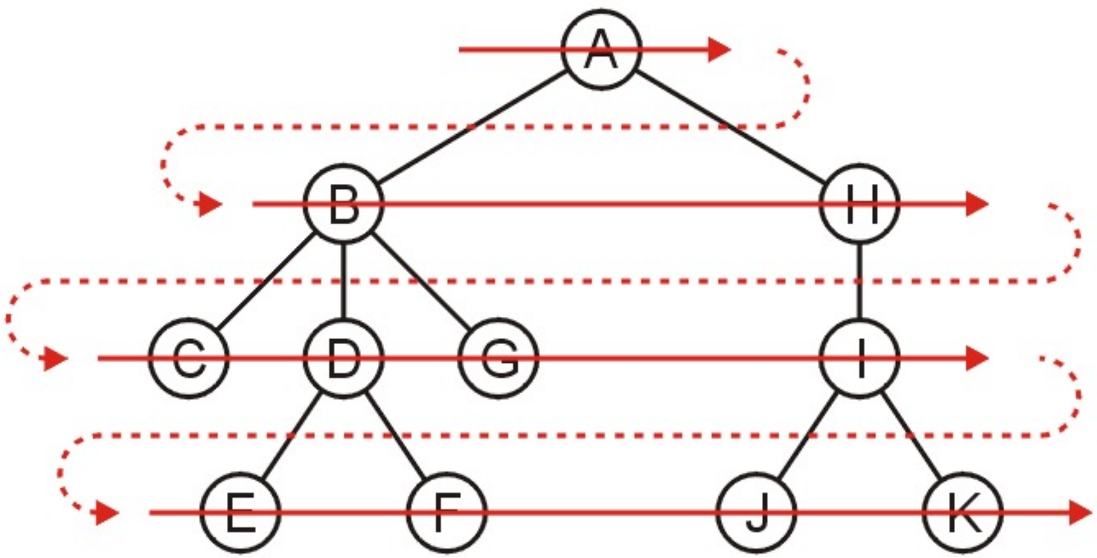
- การเข้าถึงโหนดใดๆ ในต้นไม้เราเรียกว่าการเยี่ยม (visiting)
- ลำดับของการเข้าถึงข้อมูลจึงขึ้นกับการนำไปใช้
- สามารถที่จะเข้าถึงโหนดได้ 2 รูปแบบ คือ

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

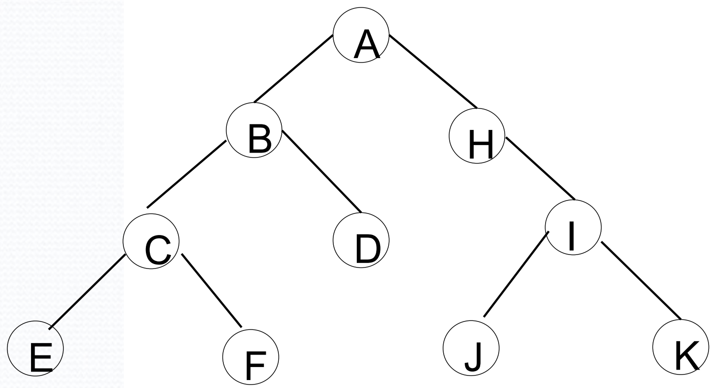
Breadth-First Traversal

ขั้นตอน

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

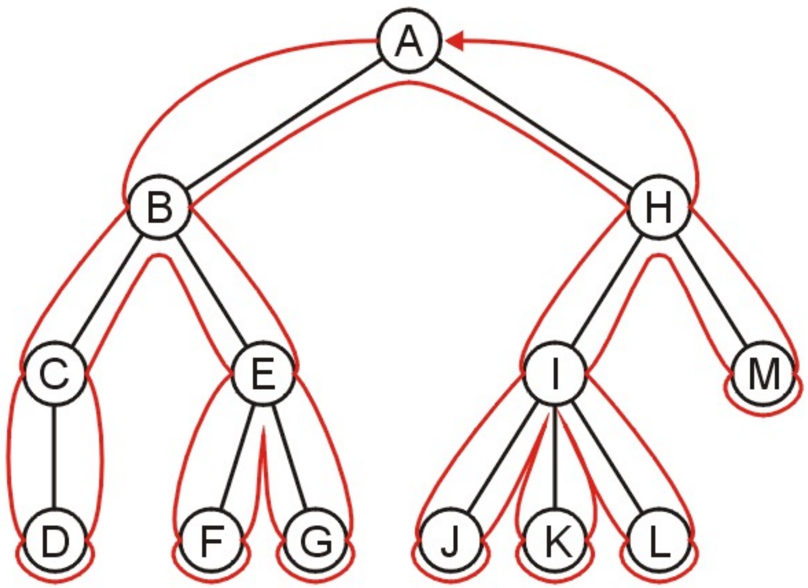
Breadth-First Traversal

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

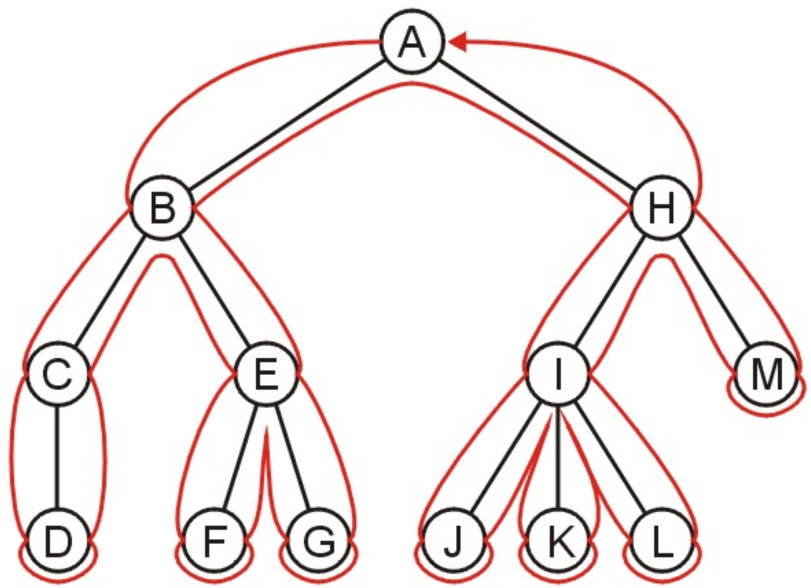


Order: A B C D E F G H I J K L M

Depth-First Traversal

<u>**ขั้นตอน**</u> **:**

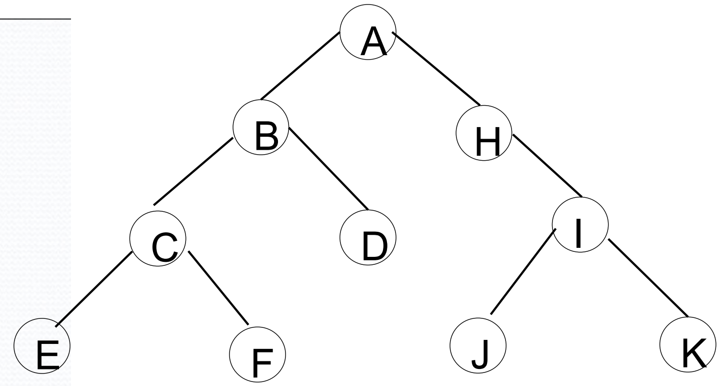
- เริ่มด้วยการใส่ root ไปที่ stack.
- Pop โหนดออกจาก stack แล้วใส่ลูกข้างขวา ตามด้วยลูกข้างซ้ายของโหนดนี้ลงไปในแสตก
- พิมพ์ค่าโหนดที่ pop ออกมา
- ทำซ้ำสองขั้นตอนก่อนหน้านี้ จนกว่าแสตกจะว่าง



Depth-First Traversal

Top ---------------------------->

ค่าใน Stack	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

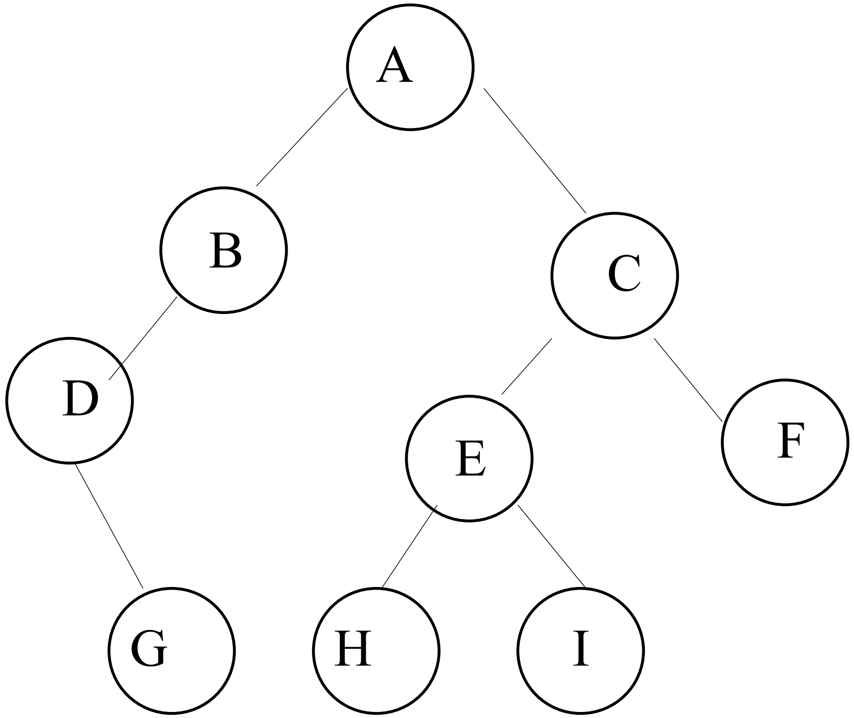
ลำดับแบบ Preorder

เป็นการเข้าถึงโหนดในต้นไม้แบบไบนารี ตามลำดับดังนี้

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

ตัวอย่าง การเข้าถึงโหนดแบบ preorder

$$\overset{\text{root}}{\downarrow}\atop{A}\quad \underbrace{B\quad D\quad G}_{\text{Left subtree}}\quad \overbrace{C\quad E\quad H\quad I\quad F}^{\text{Right subtree}}$$



ลำดับแบบ Inorder

เป็นการเข้าถึงโหนดในต้นไม้แบบไบนารี ตามลำดับดังนี้

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

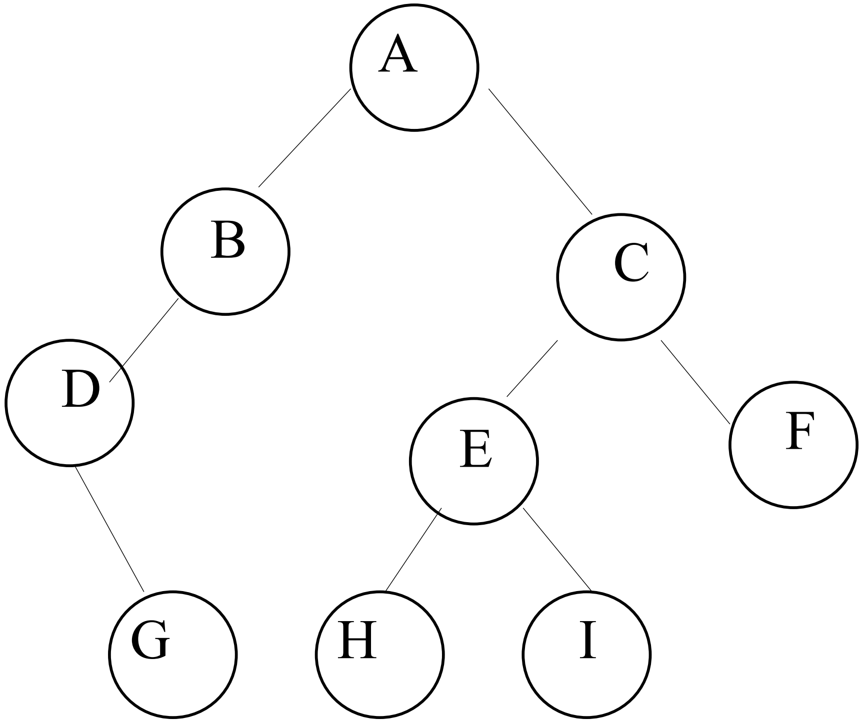
ตัวอย่าง การเข้าถึงโหนดแบบ inorder

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)

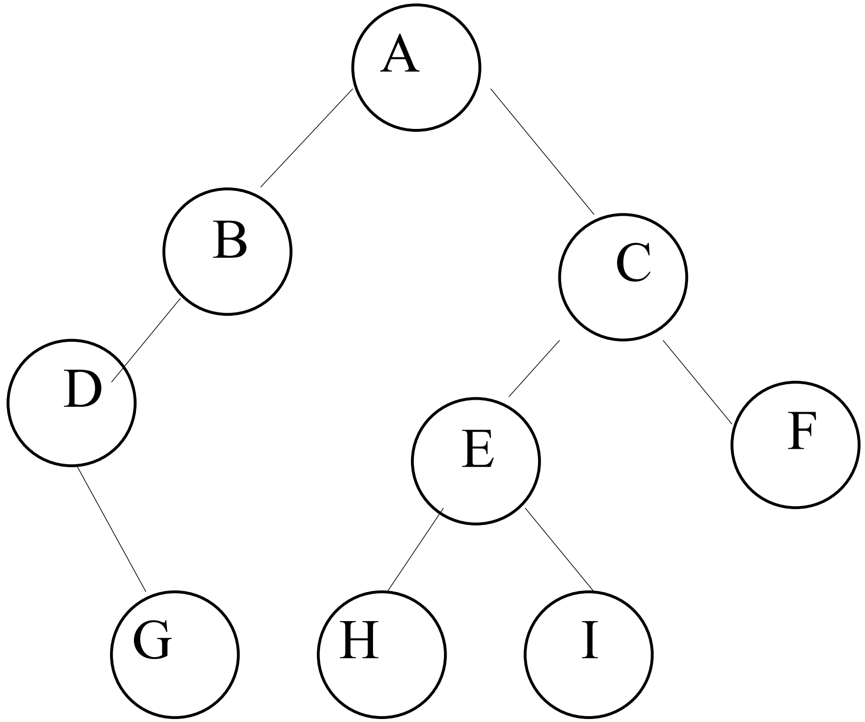
ตัวอย่าง การเข้าถึงโหนดแบบ postorder

root

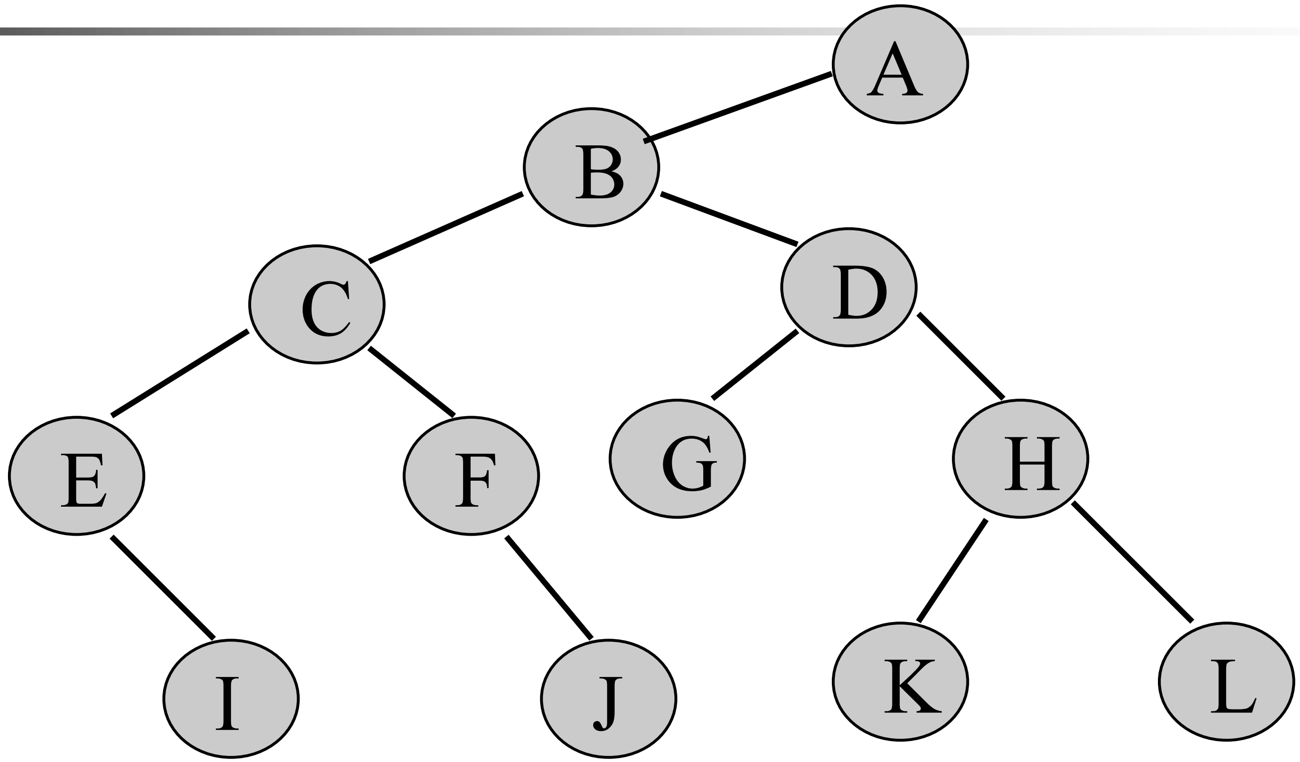
Right subtree

G D B H I E F C A

Left subtree



ตัวอย่าง



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

ต้นไม้ไบนารีกับการแก้ปัญหา

- การหาเลขซ้ำ
- การเรียงลำดับข้อมูล

การหาตัวเลขซ้ำ

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

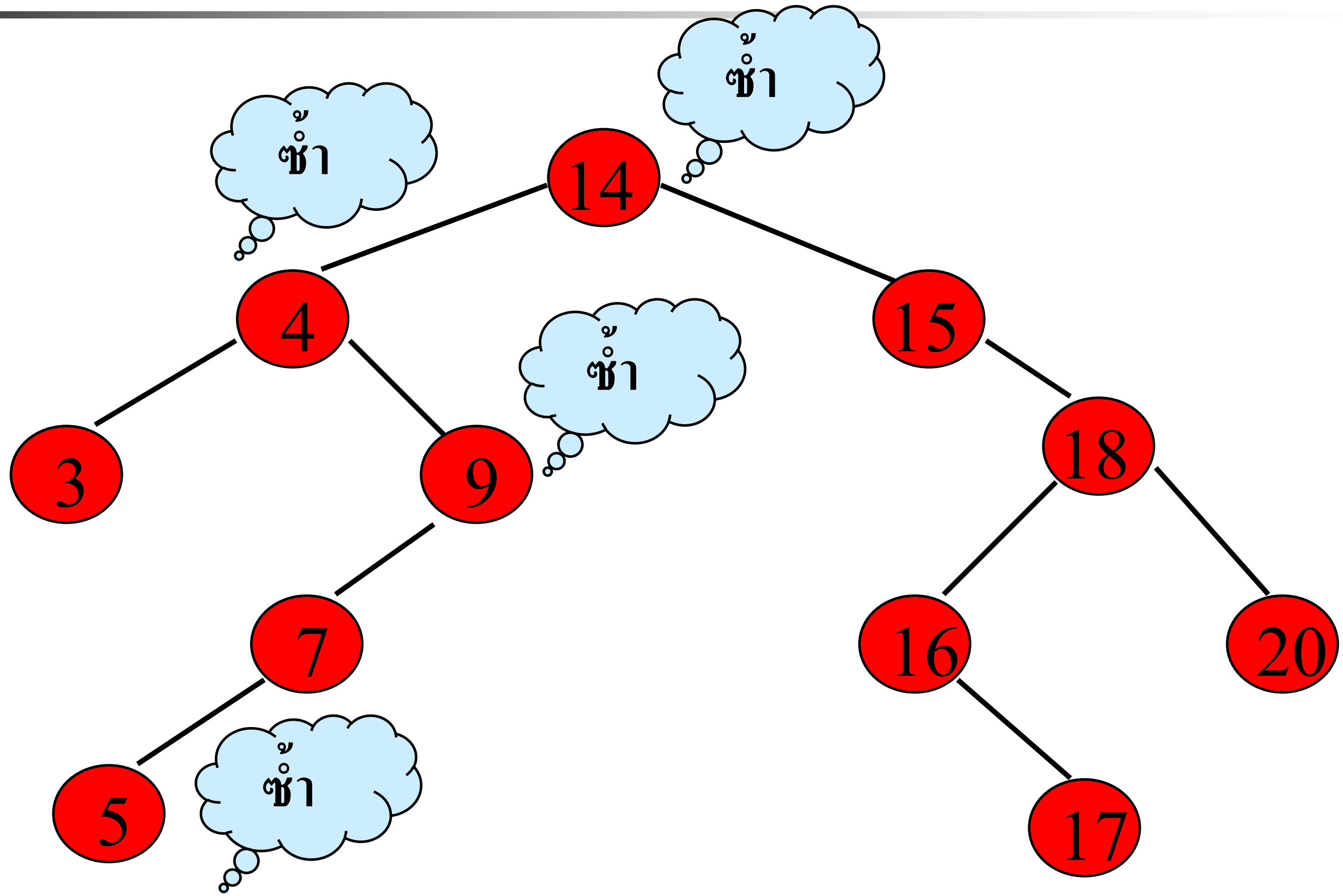
- วิธีการเปรียบเทียบทีละตัว
  - ต้องเปรียบเทียบข้อมูลทุกตัวกับข้อมูลทั้งหมดกว่าจะทราบว่ามีข้อมูลซ้ำกี่ตัวและซ้ำจำนวนเท่าใด
  - จำนวนครั้งของการเปรียบเทียบมาก
- สามารถที่จะใช้โครงสร้างต้นไม้แบบไบนารีมาแก้ปัญหาเพื่อลดจำนวนครั้งของการเปรียบเทียบ

การสร้างต้นไม้เพื่อหาเลขซ้ำ

- อ่านเลขเข้ามาทีละจำนวน
- เลขจำนวนแรกที่อ่านเข้ามา สร้างเป็นโหนดรากของต้นไม้
- เลขจำนวนถัดๆ มาให้ทำการเปรียบเทียบกับโหนดราก ซึ่งผลของการเปรียบเทียบแบ่งเป็น 3 กรณี
  1. เลขที่อ่านเข้ามาเท่ากับเลขที่โหนดรากแสดงว่าเกิดการซ้ำ
  2. เลขที่อ่านเข้ามาน้อยกว่าเลขที่โหนดราก ให้พิจารณาต้นไม้ย่อยทางซ้าย
  3. เลขที่อ่านเข้ามามากกว่าเลขที่โหนดราก ให้พิจารณาต้นไม้ย่อยทางขวา

- ถ้าต้นไม้ย่อยที่ทำการเปรียบเทียบเป็นต้นไม้ว่าง และเลขที่อ่านเข้ามายังไม่ซ้ำ ก็ให้สร้างโหนดใหม่สำหรับเลขจำนวนนั้น ณ ตำแหน่งนั้น
- ถ้าต้นไม้ย่อยที่พิจารณาไม่ว่างเราจะทำการเปรียบเทียบเลขที่อ่านเข้ามากับโหนดรากของต้นไม้ ย่อย 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

  น้อยกว่า เปรียบเทียบต่อไปที่ต้นไม้ย่อยทางด้านซ้าย

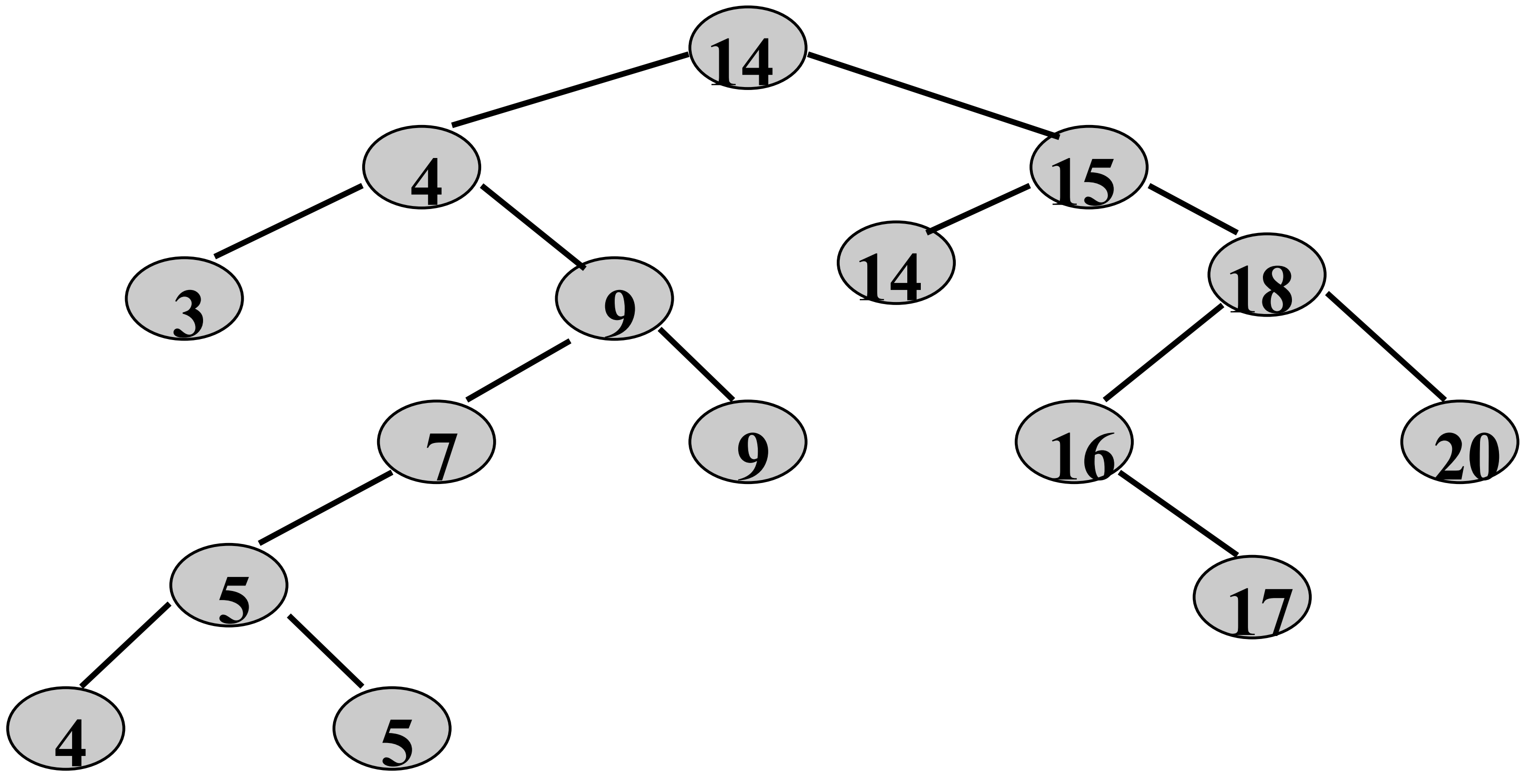
  มากกว่าเปรียบเทียบต่อไปที่ต้นไม้ย่อยทางด้านขวา

  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

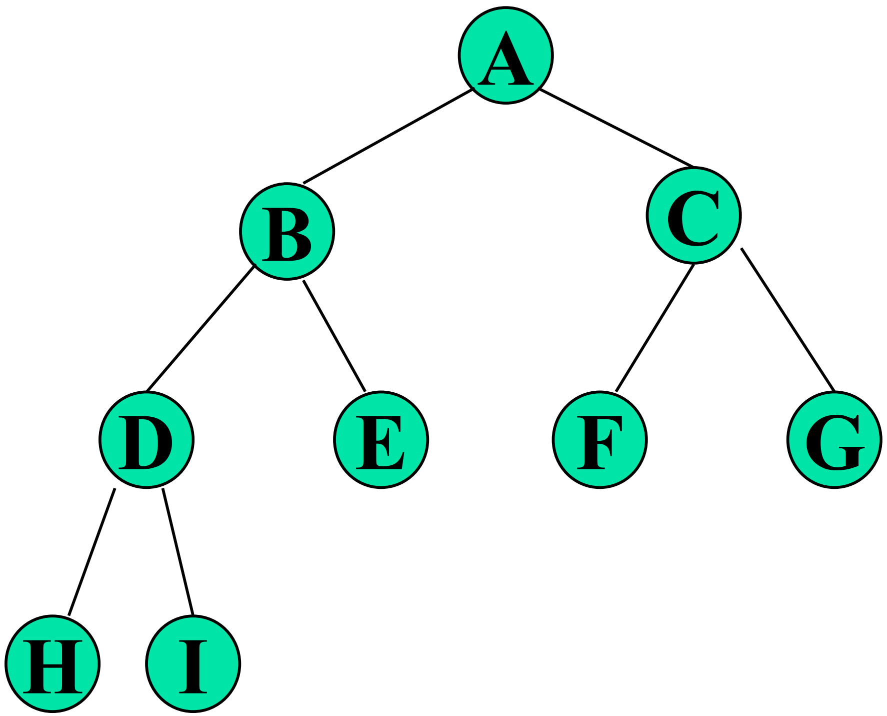
การสร้างต้นไม้ไบนารีในภาษา C

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

```
#define 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

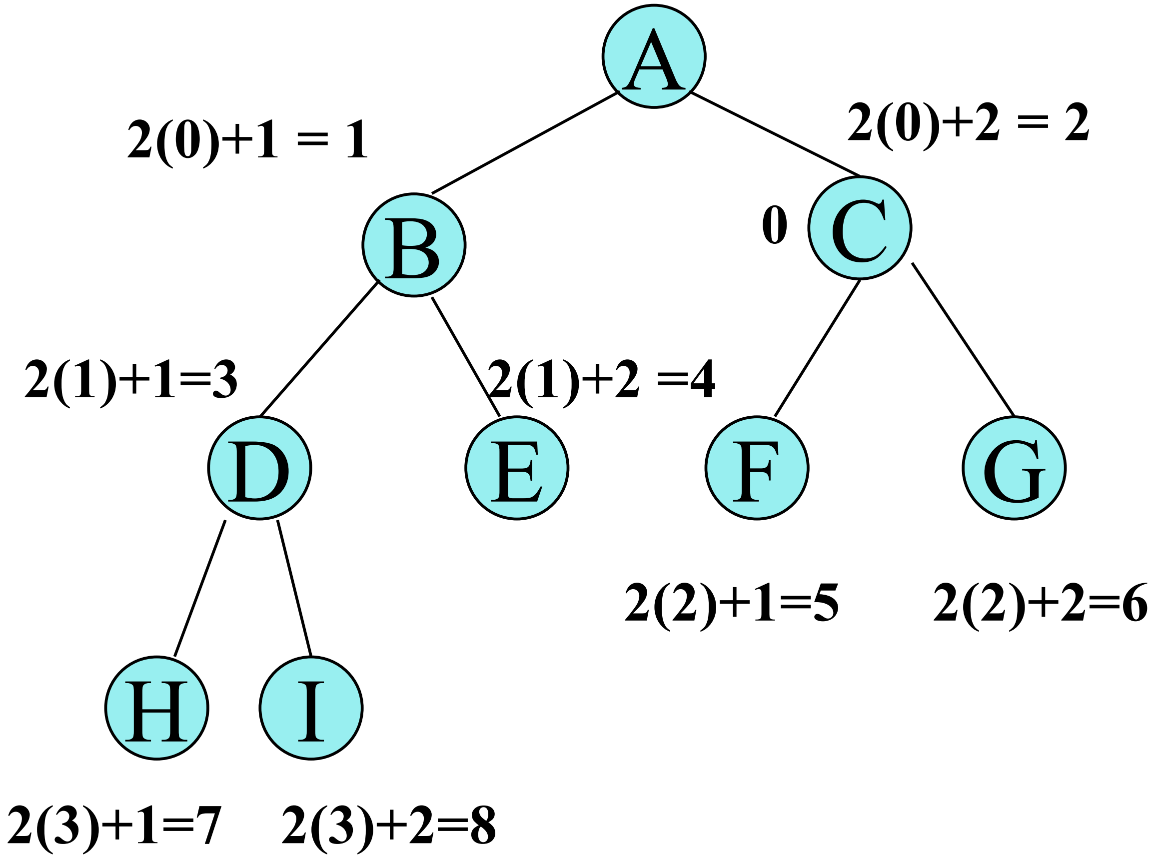
- ภาษา C มีการเก็บข้อมูลในอะเรย์ตั้งแต่ช่องที่ 0 แต่ไม่มีโหนดใดเลยที่มีค่าหมายเลขเป็น 0 ทั้งนี้เพื่อเป็นการใช้เนื้อที่ให้มีประสิทธิภาพ เราจึงกำหนดหมายเลขประจำโหนดใหม่ ดังนี้

  โหนดรากให้หมายเลข 0

  โหนดทางซ้ายของโหนด n ใดๆ มีหมายเลข 2n+1

  โหนดทางขวาของโหนด n ใดๆ มีค่า 2n+2

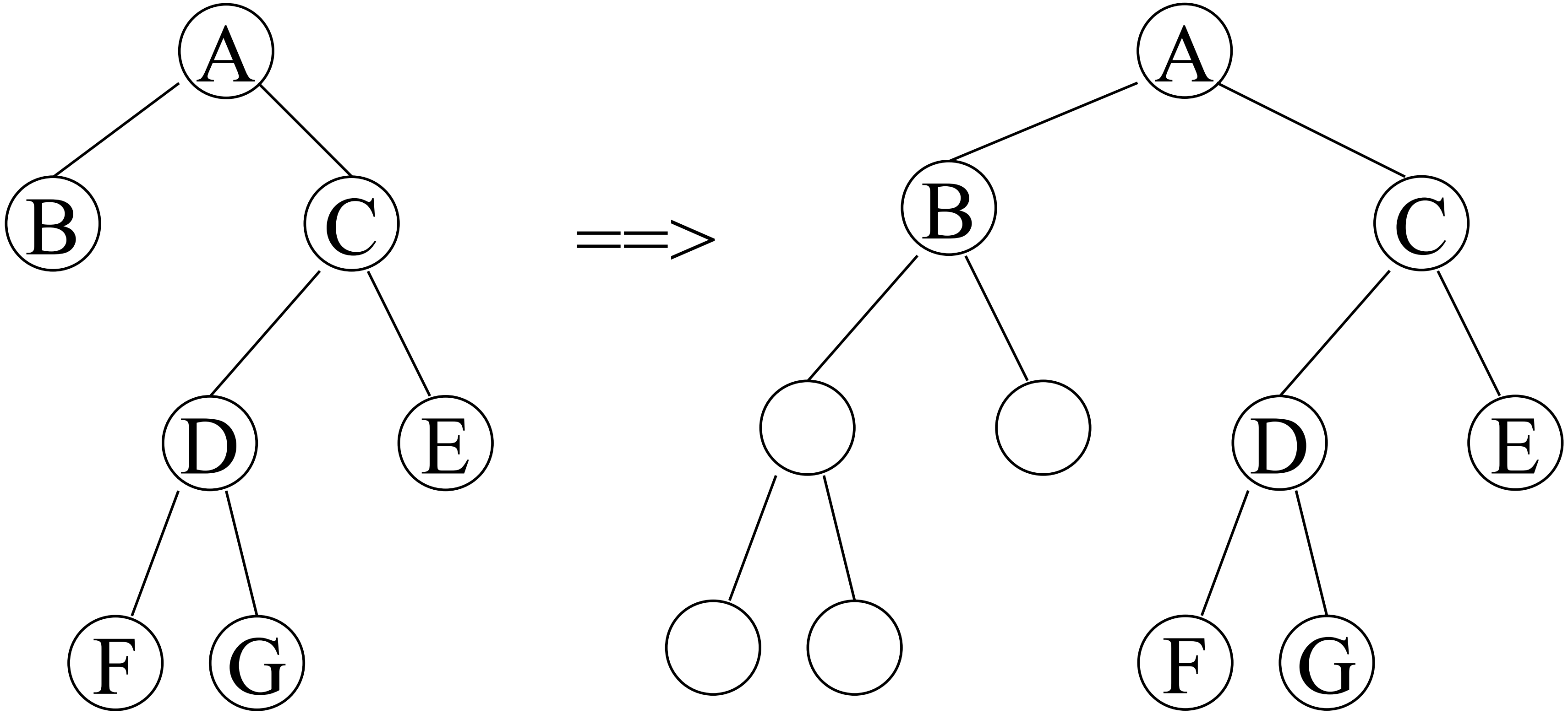
- โหนดรากของต้นไม้จะถูกเก็บในอะเรย์ที่ช่อง 0 เสมอ



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

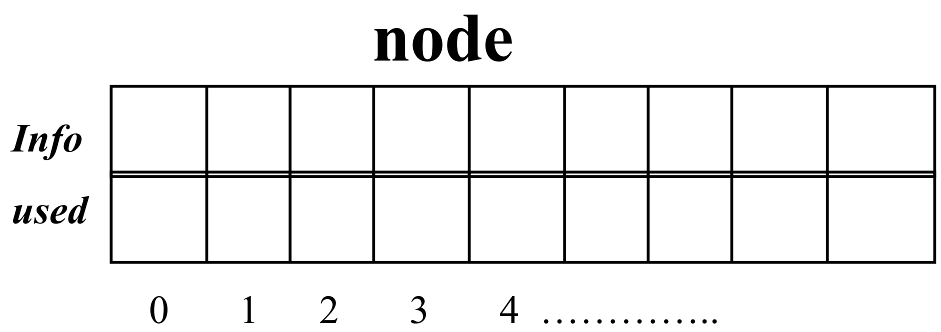
การจัดเก็บแบบนี้ถ้าต้นไม้ไบนารีไม่ถูกเติมเต็มดังรูป เราก็ต้องมีการ
เผื่อเนื้อที่สำหรับโหนดทุกตำแหน่งไว้



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

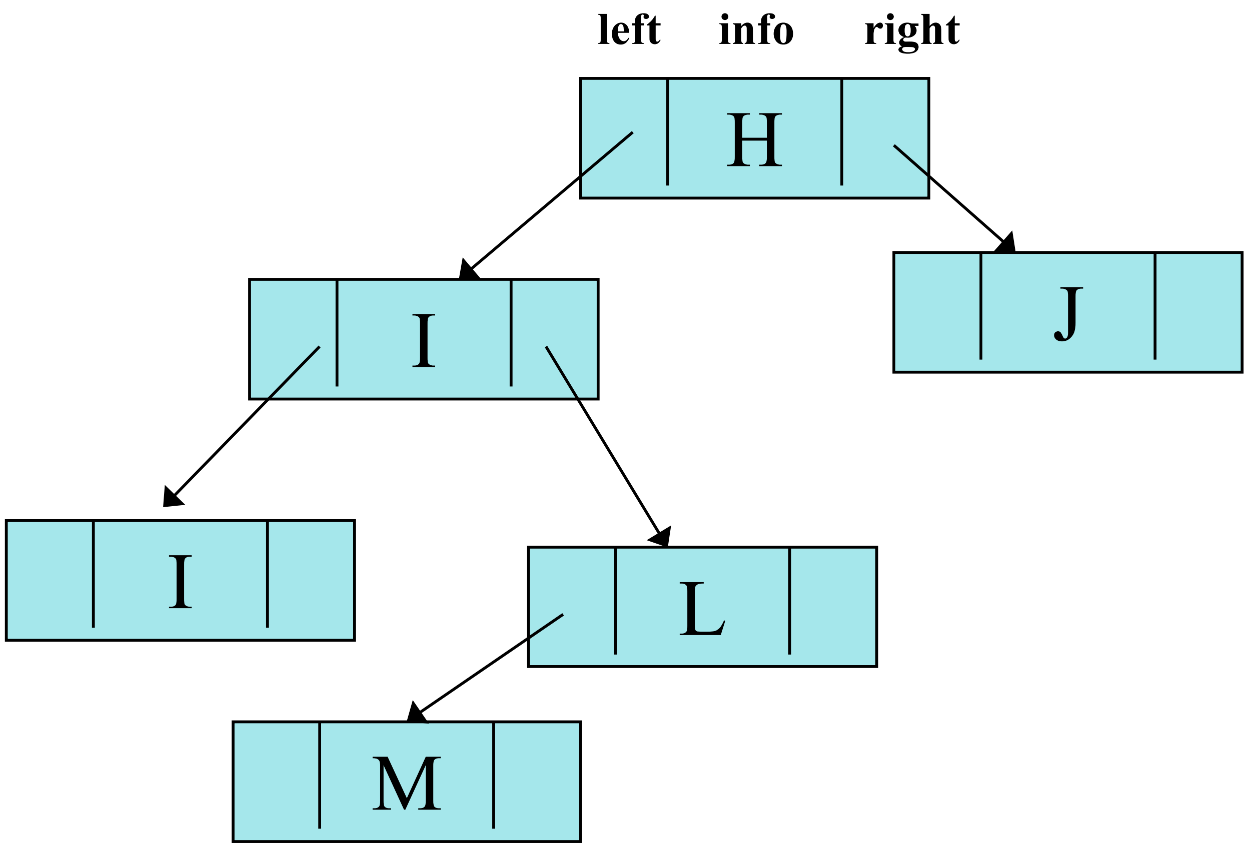
โครงสร้างของโหนด

```
#define NUMNODES 500
struct nodetype {
    char  info;
    int  used;
} node[NUMNODE];
```



ต้นไม้ไบนารีด้วยตัวแปรแบบพลวัตร

การใช้ตัวแปรแบบพลวัตร ไม่จองเนื้อที่ให้แต่ละโหนดล่วงหน้า



โครงสร้างของโหนด

```
struct nodetype {
    char  info;
    struct  nodetype  *left;
    struct  nodetype  *right;
    struct  nodetype  *father;      /* optional */
};
typedef struct  nodetype *NODEPTR;
```

Tree operations

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

maketree algorithm

algorithm makeTree (val x <key>)

allocates a node and sets it as the root of a single-node binary tree

Pre x is the key value requested

Return address of new node with value x

node = new

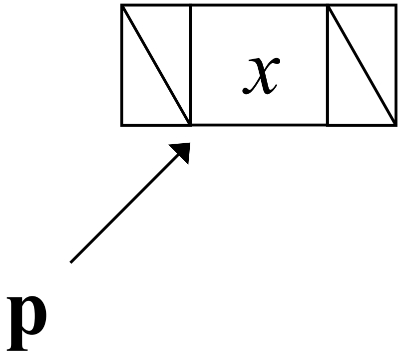
node->key = x

node->left = null

node->right = null

return node

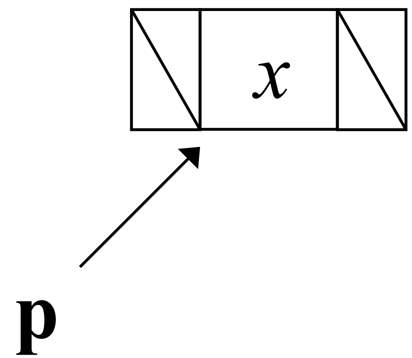
end makeTree



maketree algorithm

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

        p = (NODEPTR)malloc(sizeof(struct nodetype);
        p->info = x;
        p->left = NULL;
        p->right = NULL;
        return (p);
}
```



setLeft algorithm

algorithm setLeft (val tree <tree pointer> val x <key>)

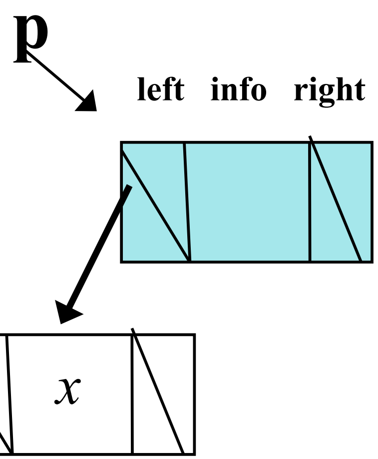
sets a node with contents *x* as the left son of *node(p)*

Pre tree is a pointer to a nod and x is the key value requested

Post tree has a left child with value x

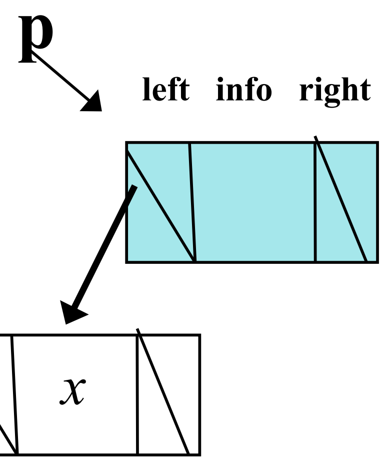
```
if (tree is null)
    print "can't set left child to tree"
else if (tree->left not null)
    print "tree already has left child .. "
else
    tree->left = makeTree(x)
```

end setLeft



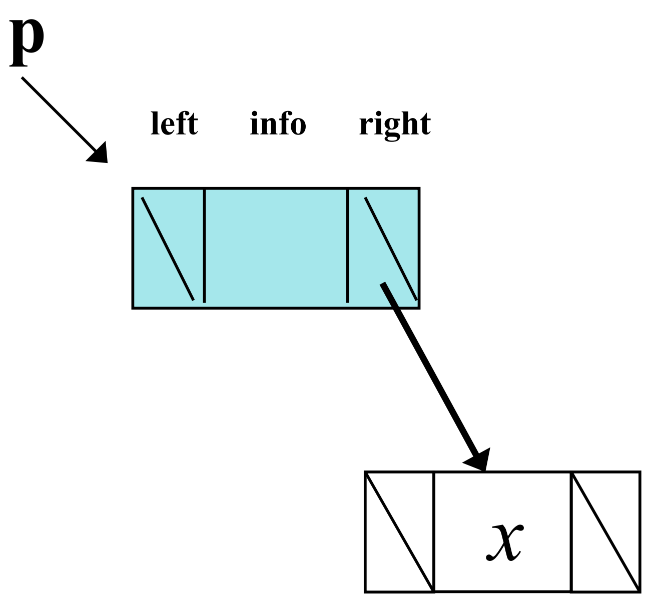
setLeft algorithm

```
void setLeft (NODEPTR p, char x)
{
        if (p == NULL)
            printf ("can't set left child to p\n");
        else if (p->left != NULLl)
            printf ("p already has left child \n");
        else
            p->left = makeTree(x);
}
```



setright Function

เหมือน setLeft แต่ทิศทางตรงกันข้าม



Preorder Traversal

algorithm preOrder (val root <node pointer>)

Traverse a binary tree in node-left-right sequence

Pre root is the entry node of a tree or subtree

Post each node has been processed in order

```
if (root is not NULL)
    process (root)
    preOrder(root->leftSubtree)
    preOrder(root->rightSubtree)
return
```

end preOrder

Preorder Traversal

```
void preOrder (NODEPTR tree)
{
    if (tree != NULL)
    {
        printf("%c  ", tree->info);
        preOrder(tree->left);
        preOrder(tree->right);
    }
}
```

Inorder Traversal

algorithm inOrder (val root <node pointer>)

Traverse a binary tree in left-node-right sequence

Pre root is the entry node of a tree or subtree

Post each node has been processed in order

```
if (root is not NULL)
        inOrder(root->leftSubtree)
        process (root)
        inOrder(root->rightSubtree)
return
```

end inOrder

Inorder Traversal

```
void InOrder (NODEPTR tree)
{
    if (tree != NULL)
    {
        InOrder(tree->left);
        printf("%c  ", tree->info);
        InOrder(tree->right);
    }
}
```

Postorder Traversal

algorithm postOrder (val root <node pointer>)

Traverse a binary tree in left-right-node sequence

Pre root is the entry node of a tree or subtree

Post each node has been processed in order

```
if (root is not NULL)
    postOrder(root->leftSubtree)
    postOrder(root->rightSubtree)
    process (root)
return
```

end postOrder

Postorder Traversal

```
void postOrder (NODEPTR tree)
{
    if (tree != NULL)
    {
        postOrder(tree->left);
        postOrder(tree->right);
        printf("%c  ", tree->info);
    }
}
```

Breath-First Traversal

```
algorithm breathFirst (val root <node pointer>)
Process tree using breath-first traversal
      Pre   root is a pointer to a tree node
      Post  tree has been processed
      pointer = root
      while (pointer not NULL)
                 process (pointer)
                     if (pointer->left not null)
                         enqueue(pointer->left)
               if (pointer->right not null)
                   enqueue(pointer->right)
                             if (not emptyQueue)
                                 dequeue(pointer)
                             else
                                  pointer = null
                return
end breathFirst
```

Breath-First Traversal

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

    while (p!= NULL)
        printf("%c  ",p->info);
        if (p->left != NULL)
           enqueue(p->left);
        if (p->right != NULL)
           enqueue(p->right);
        if (! emptyQ())
           p = dequeue()
        else
           p = NULL;
 }
```

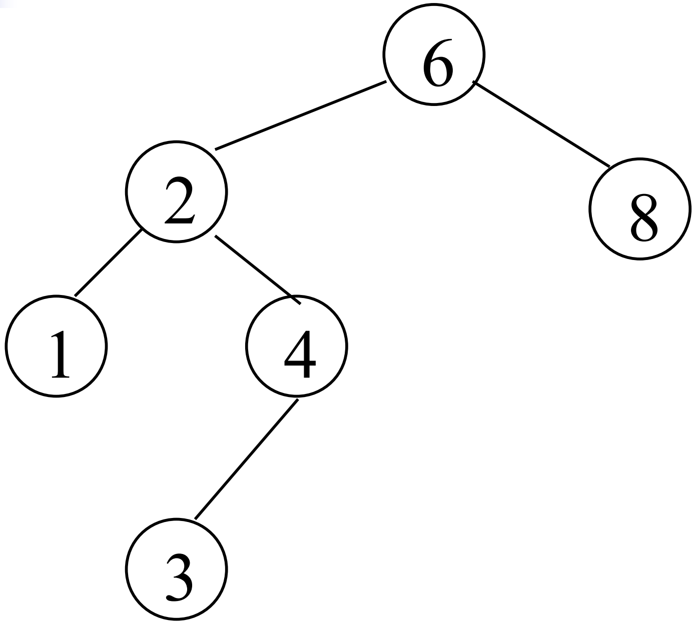
Binary Search Tree (BST)

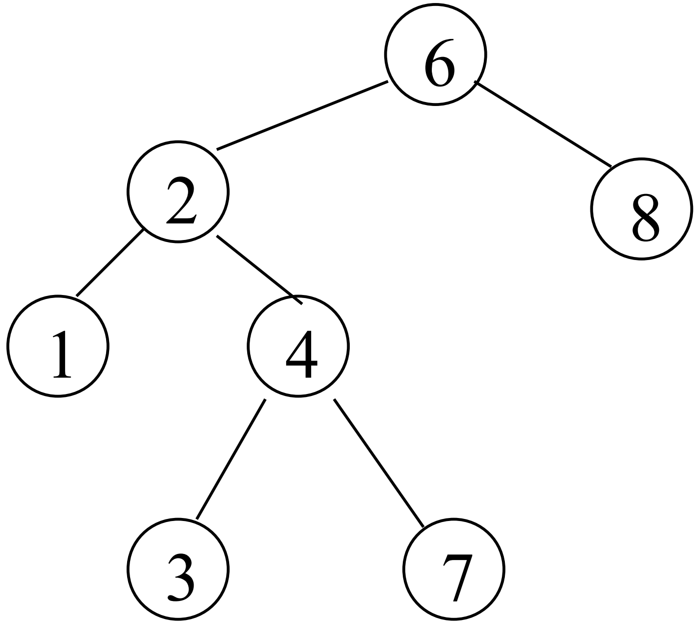
ต้นไม้ไบนารีสำหรับการค้นหา (BST)

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

Node

data <data type> // ตัวแปรที่ใช้เก็บข้อมูล

left <pointer to Node> //เก็บตำแหน่งของลูกทางซ้าย

right <pointer to Node> //เก็บตำแหน่งของลูกทางขวา

end Node

โครงสร้างของ BST

```
struct node {
    int info;                   // ตัวแปรที่ใช้เก็บข้อมูล
    struct node *left;          //เก็บตำแหน่งของลูกทางซ้าย
    struct node *right;         //เก็บตำแหน่งของลูกทางขวา
}

typedef struct node * NODEPTR;
```

Operations of Binary Search Tree

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

```
algorithm searchBST (val root <pointer>, val arg <key>)
Search BST for a given value
    Pre      root is a pointer to a BST or subtree
             arg is the key value requested
    Return the node address if the search value is found
             null if the node is not in the tree
    if ( root is null )
        return null
    if  (arg < root->key)
        return  searchBST(root->left, arg);
    else if  (arg > root->key)
        return  searchBST(root->right, arg);
    else
        return root
end searchBST
```

```
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;
}
```

algorithm findSmallestBST (val root <pointer>)

The algorithm finds the smallest node in a BST

Pre root is a pointer to a non-empty BST

Return address of the smallest node

if (root->left null)

return root;

return findSmallestBST(root->left);

end findSmallestBST

algorithm findLargestBST (val root <pointer>)

The algorithm finds the largest node in a BST

Pre root is a pointer to a non-empty BST

Return address of the largest node

```
if  ( root->right  null )
    return root;
return  findLargestBST(root->right);
```

end findLargestBST

algorithm insertBST (ref root <pointer>, val new <pointer>)

Insert node containing new node into BST

Pre root is address of the first node in a BST

new is address of node containing data to be inserted

Post new node inserted into the tree

Iteration

```
if  ( root is null )
     root = new
else
     pWalk = root
     Loop (pWalk not null)
           parent = pWalk
            if (new->key < pWalk->key)
                 pWalk = pWalk->left
            else
                 pWalk = pWalk->right
            Location for new node found
            if (new->key < parent->key)
                parent->left = new
            else
                parent->right = new
      return
end insertBST
```

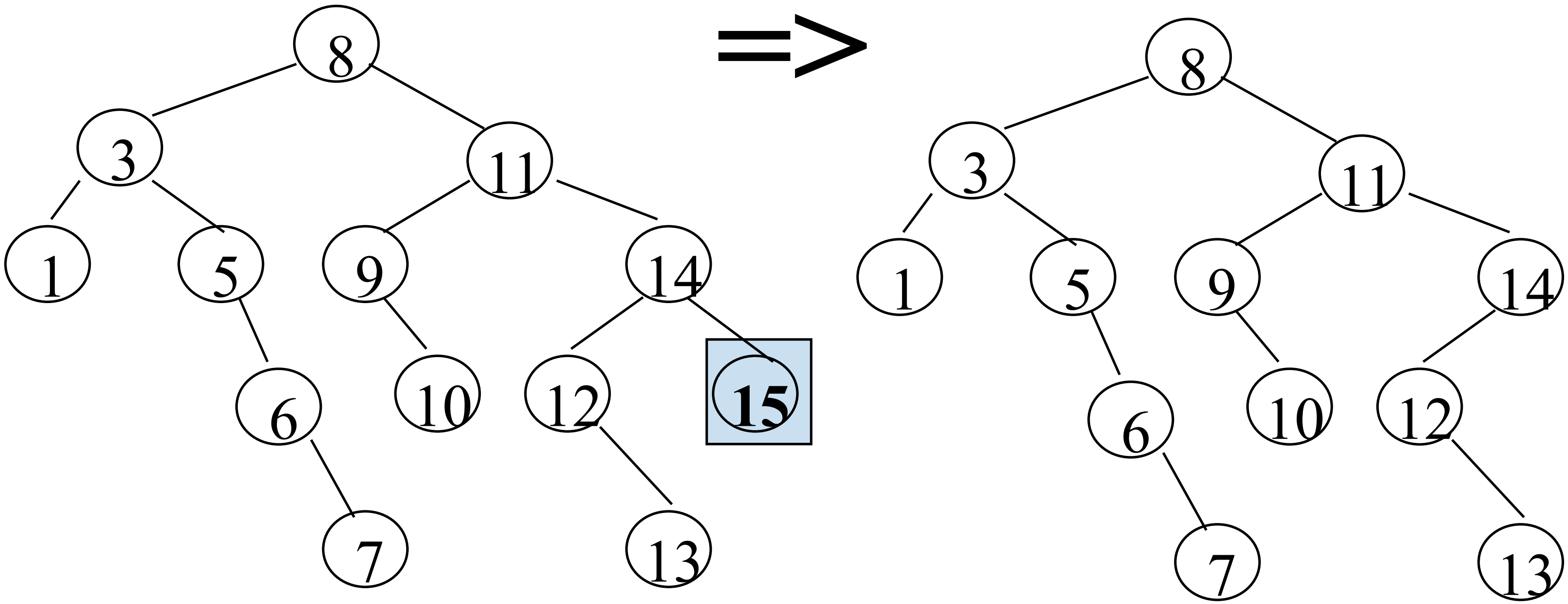
Recursion

```
    if  ( root is null )
        root = new
        root->left = null
        root->right = null
    else
        Locate null subtree for insertion
        pWalk = root
        if (new->key < root->key)
                addBST(root->left, new)
        else
                addBST(root->right, new)
    return
end insertBST
```

การ Delete ข้อมูล

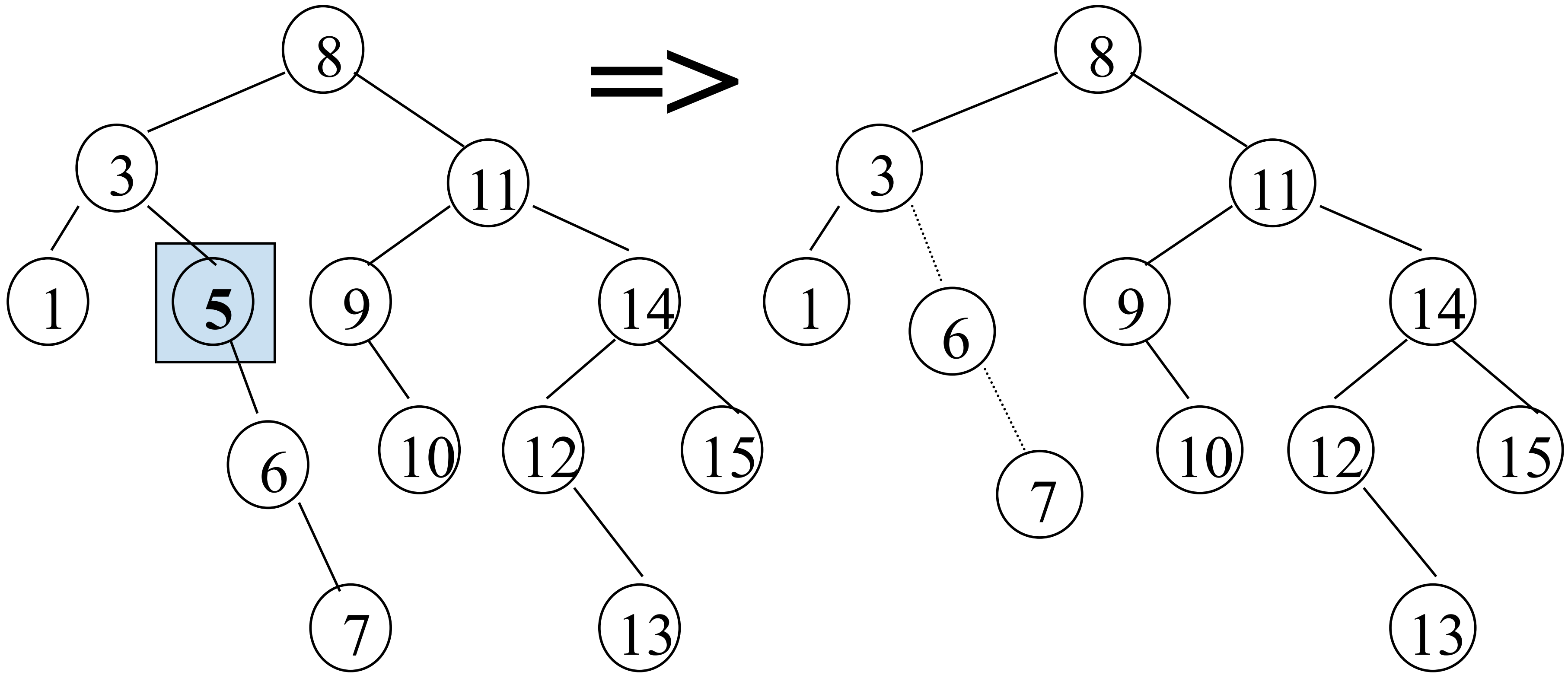
- การลบข้อมูลใน BST ค่อนข้างจะยุ่งยาก เนื่องจากพบลบแล้วก็ต้องทำการปรับ ลิงค์ต่าง ๆ ซึ่งแตกต่างกันเป็นกรณีๆ ไป
- หลังจากลบแล้ว ต้นไม้ที่เหลือจะต้องคง คุณสมบัติของ BST คือโหนดทางซ้ายมีค่าน้อยกว่าโหนดกลางและโหนดทางขวา มีค่ามากกว่าหรือเท่ากับโหนดกลาง
- เราแบ่งการพิจารณาออกได้เป็น 3 กรณี

กรณีที่1 โหนดที่ต้องการลบเป็นโหนดใบไม้



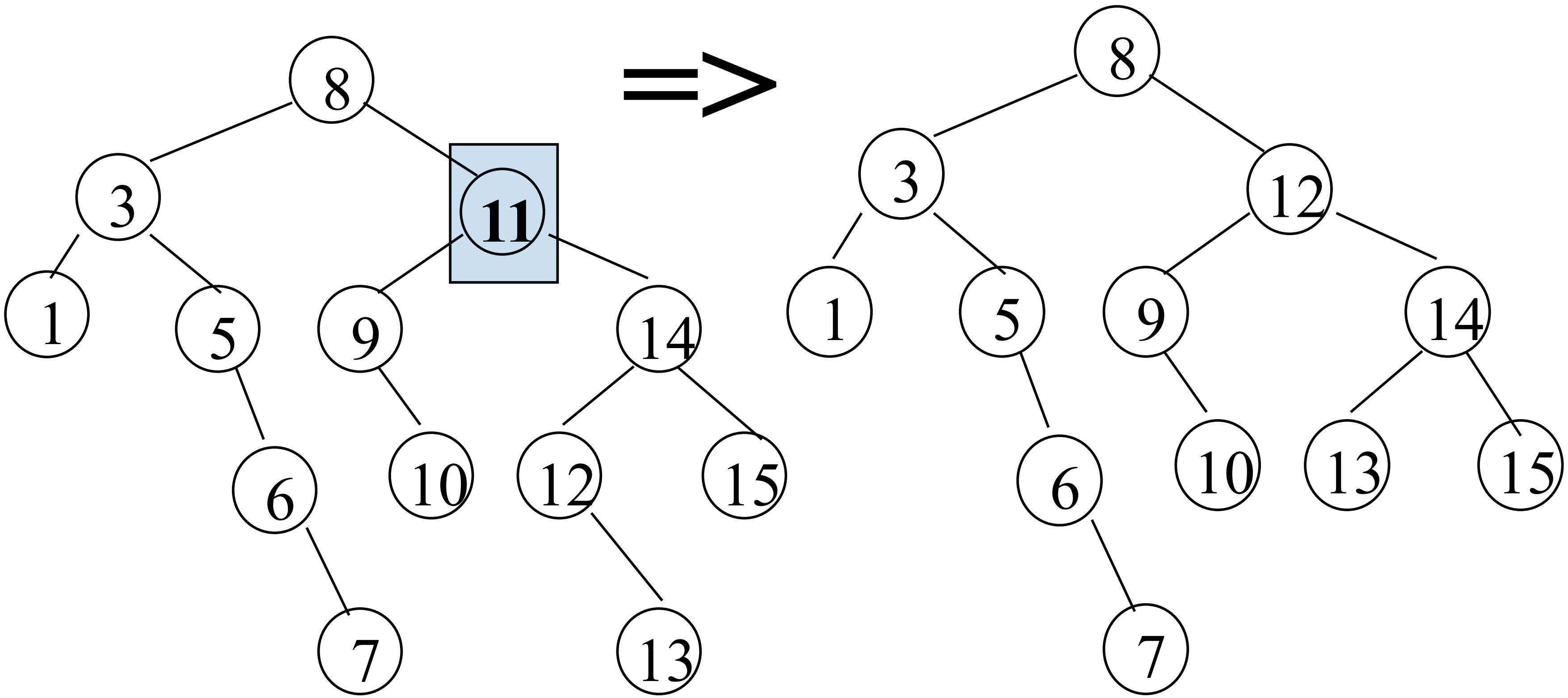
(a) deleting node with key 15.

กรณีที่ 2 โหนดที่ต้องการลบมีต้นไม้ย่อยเพียงข้างซ้ายข้างเดียวหรือข้างขวาเพียงข้างเดียว



(b) deleting node with key 5.

กรณีที่3 โหนดที่ต้องการลบมีต้นไม้ย่อยทั้งสองข้าง



(c) deleting node with key 11.

algorithm deleteBST (ref root <pointer>, val dltKey <key>)

This algorithm deletes a node from BST

Pre root is pointer to tree containing data to be deleted

dltKey is key of node to be deleted

Post node deleted and memory recycled

if dltKey not found, root unchanged

```
if  ( root null )
        return false
if (dltKey < root->key)
        return deleteBST(root->left, dltKey)
else if (dltKey > root->key)
        return deleteBST(root->right, dltKey)
else
```

Delete node found --- Test for leaf node

```
if  ( root->left null )
     dltPtr = root
     root = root->right
     recycle (dltPtr)
     return true
else if  ( root->right null )
     dltPtr = root
     root = root->left
     recycle (dltPtr)
     return true
else
     Node to be deleted not a leaf. Find largest node on left subtree
     dltPtr = root->left
     Loop (dltPtr->right not null)
          dltPtr = dltPtr->right
     Node found. Move data and delete leaf node
     root->data = dltPtr->data
     return deleteBST(root->left, dltPtr->data)
end deleteBST
```

ประสิทธิภาพการค้นหาของ BST

- ประสิทธิภาพการค้นหาข้อมูลใน BST มักจะขึ้นกับรูปร่างของต้นไม้ ว่ามีความสมดุลหรือ เอียงมากน้อยแค่ไหน