Design and Analysis of Data Structures and Algorithms :: Tree part 2

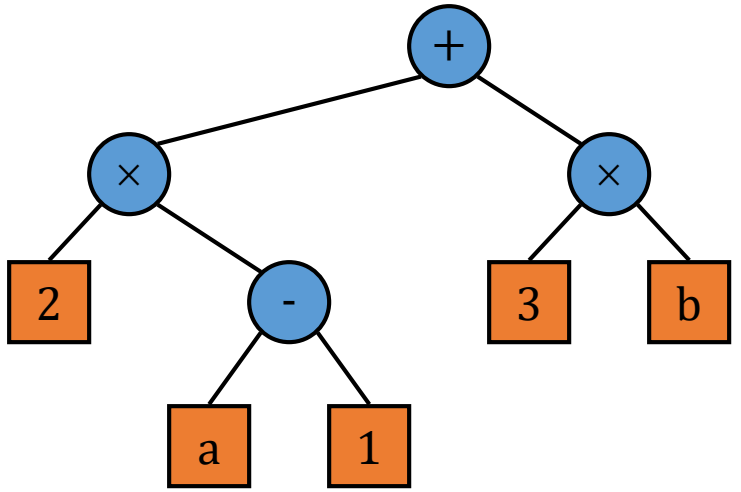
อ.ดร.วรินทร์ วัฒนพรพรหม

Binary Tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching

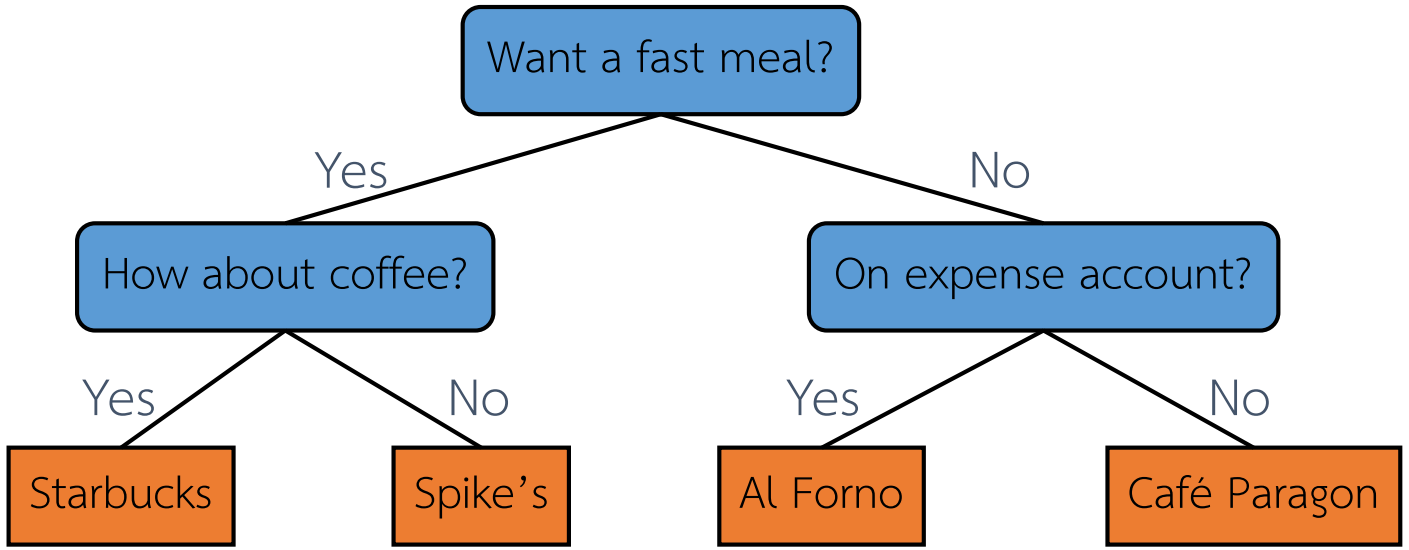
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with Yes/No answer
  - external nodes: decisions
- Example: dining decision



Comparing Binary Search Trees to Linear Lists

(presorted - insertion)

Big-O Comparison			
Operation	Binary Search Tree	Array-based List	Linked List
Constructor	O(1)	O(1)	O(1)
Destructor	O(n)	O(1)	O(n)
IsFull	O(1)	O(1)	O(1)
IsEmpty	O(1)	O(1)	O(1)
RetrieveItem	O(log n)*	O(log n)	O(n)
InsertItem	O(log n)*	O(n)	O(n)
DeleteItem	O(log n)*	O(n)	O(n)

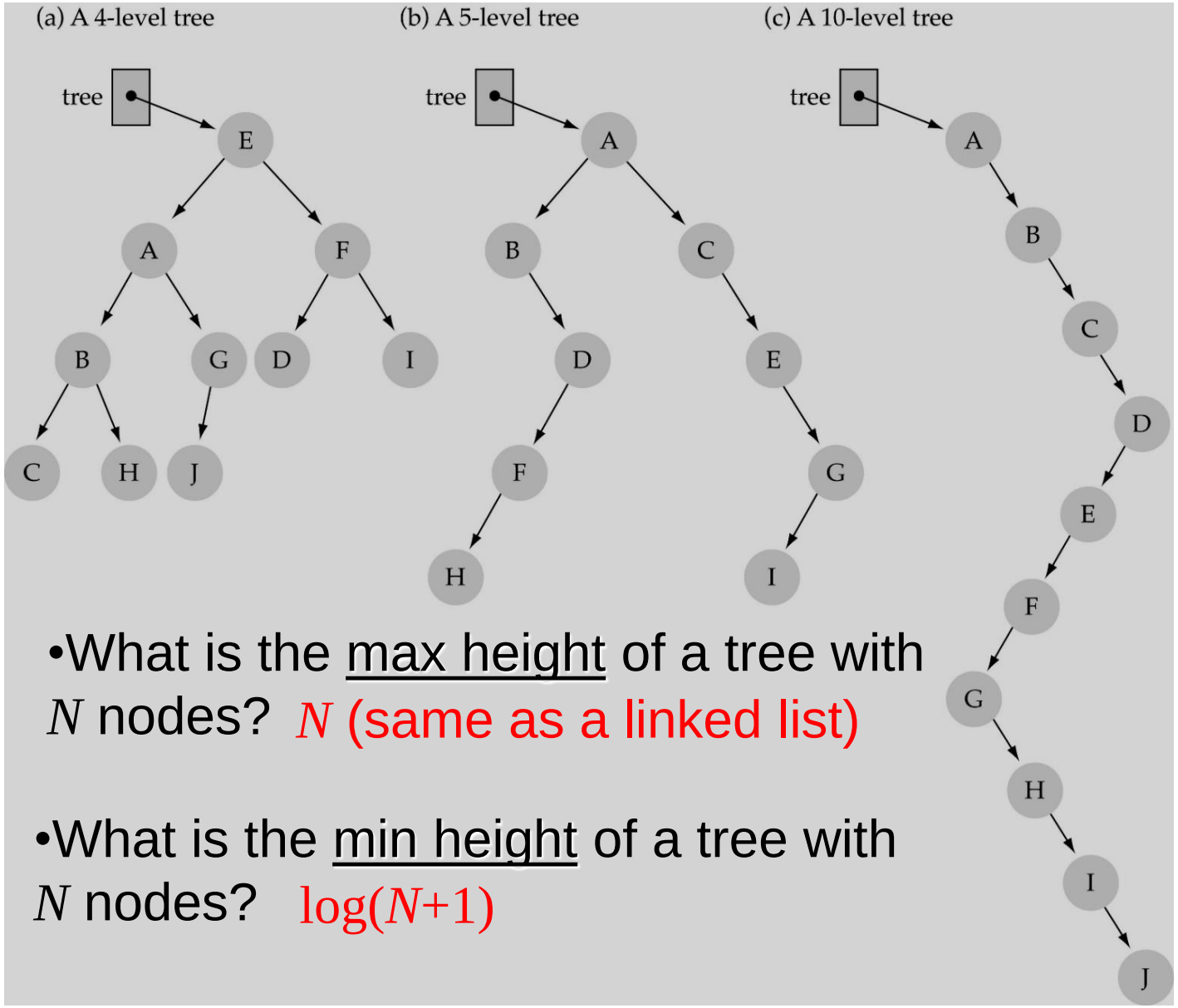
*assuming h=O(log n)

จะเกิดอะไรขึ้นถ้าความสูงของต้นไม้ h เท่ากับทุกๆ โหนด N

$$2^h - 1 = N$$
$$\Rightarrow 2^h = N + 1$$
$$\Rightarrow \ \ h = \log(N + 1) \rightarrow O(\log N)$$

ความสูงของต้นไม้ h สำคัญอย่างไร

- แต่ละ Operations ของต้นไม้ (insert, delete, retrieve etc.) ทำงานตามความสูง h.
- แปลได้ว่าความสูงของต้นไม้ h สามารถใช้ในการประเมินเวลา



- What is the max height of a tree with N nodes? N (same as a linked list)

- What is the min height of a tree with N nodes? $\log(N+1)$

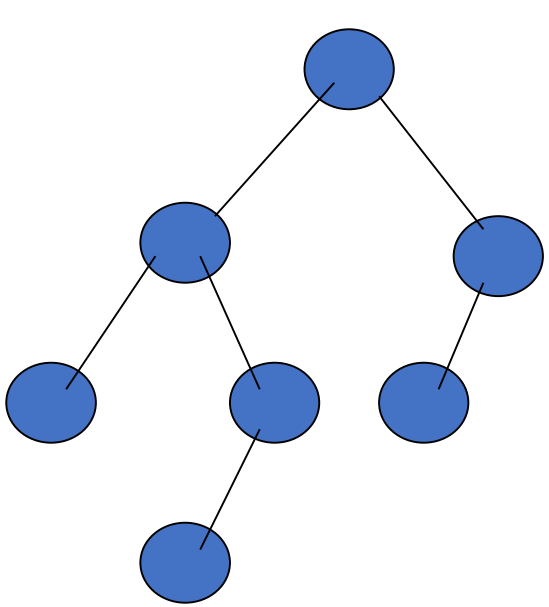
โครงสร้างข้อมูลแบบ AVL Tree

ต้นไม้เอวีแอล (AVL Tree - named after inventors **A**delson-**V**elsky and **L**andis)

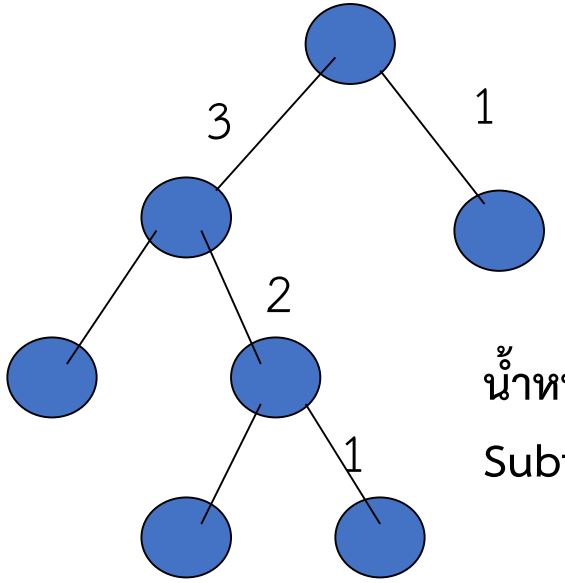
เป็นไบนารีเซิร์ชทรีแบบ**สมดุล** :

ถ้า T คือต้นไม้ใดๆ และ N คือโหนดใดๆ บนต้นไม้ T แล้ว ความแตกต่างระหว่างความสูงของต้นไม้ย่อย (Sub Tree) ที่อยู่ทางขวา และทางซ้าย ของ N จะถูกเรียกว่า บาลานซ์แฟคเตอร์ (Balance Factor) และต้นไม้ T จะถูกเรียกว่าต้นไม้สมดุล เมื่อบาลานซ์แฟคเตอร์ของ N ใดๆ ต้องมีค่าไม่เกิน |1| (ต้องมีค่าเป็น -1 หรือ 0 หรือ 1)

ต้นไม้เอวีแอล (AVL Tree)



AVL



Non-AVL

น้ำหนักความสูงของ node = ความสูง Subtree(ซ้าย) - ความสูง Subtree(ขวา)

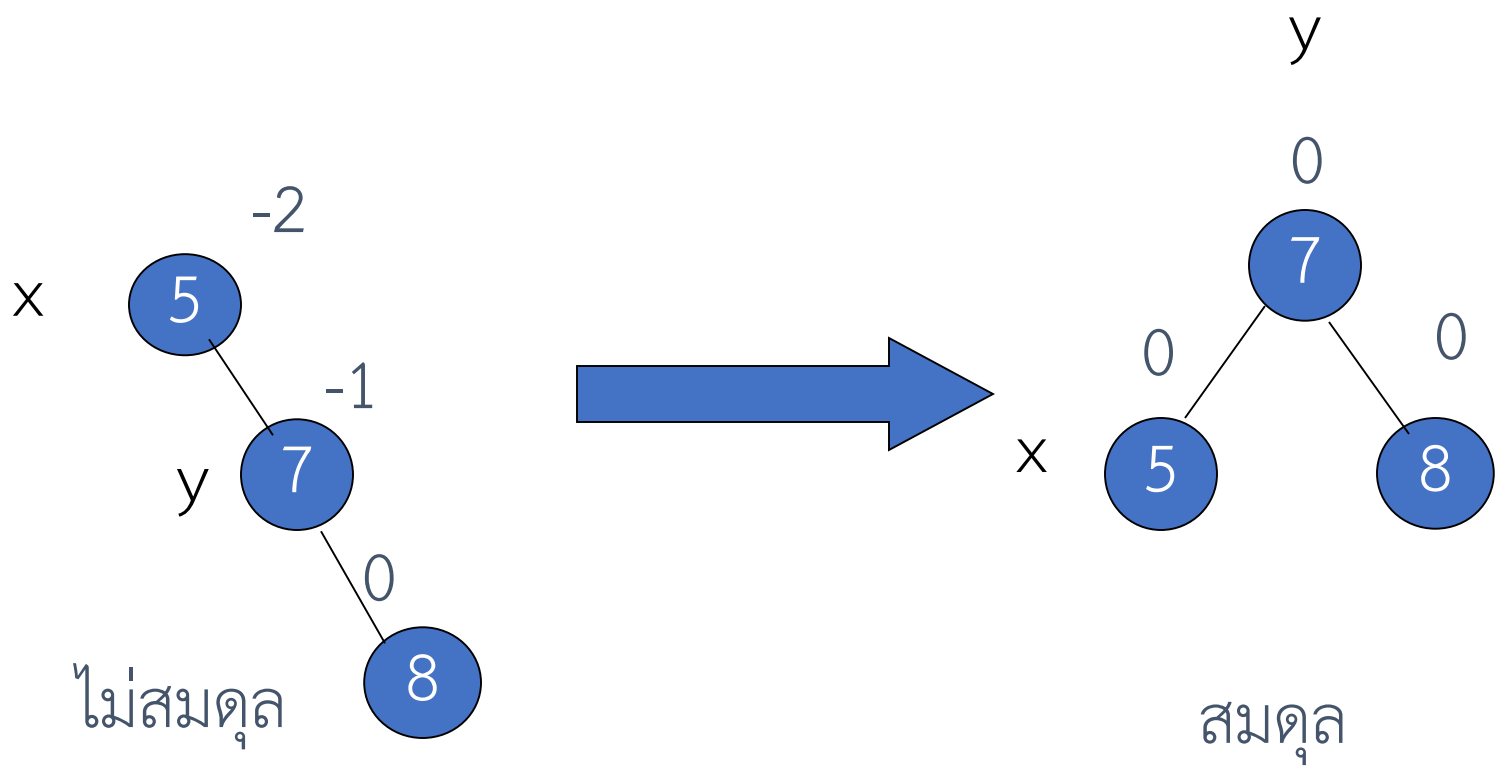
ต้นไม้เอวีแอล (AVL Tree)

AVL TREE เมื่อมี Tree ที่ไม่สมดุลจะเกิดการหมุนเพื่อให้สมดุลซึ่งมี 4 วิธี

1. การหมุนซ้าย : โหนดทางขวามากกว่าซ้าย ยกโหนดที่อยู่ขวาขึ้นมา
2. การหมุนขวา : โหนดทางซ้ายมากกว่าขวา ยกโหนดที่อยู่ซ้ายขึ้นมา
3. การหมุนขวาไปซ้าย : หมุนขวาก่อน แล้วค่อยหมุนซ้าย
4. การหมุนซ้ายไปขวา : หมุนซ้ายก่อน แล้วค่อยหมุนขวา

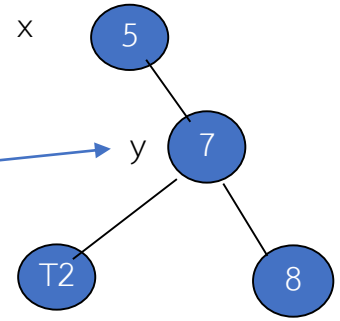
ต้นไม้เอวีแอล (AVL Tree) - การหมุนซ้าย

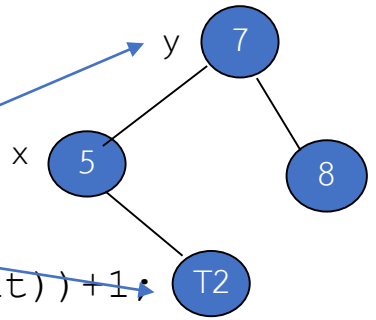
การหมุนซ้าย



ต้นไม้เอวีแอล (AVL Tree) - การหมุนซ้าย

```
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    //  Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}
```



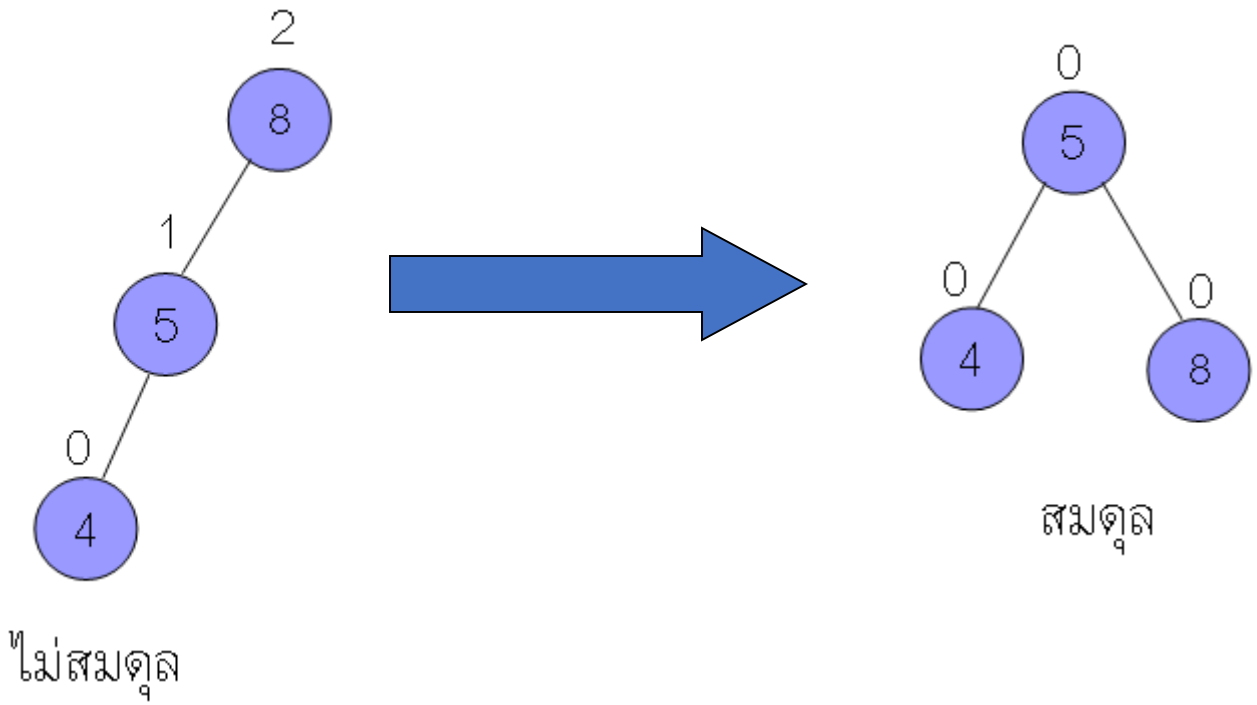


Get Balance

```
int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}
```

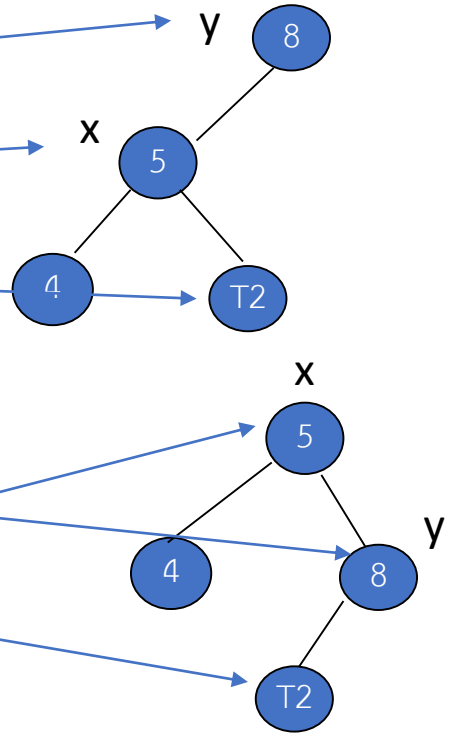
ต้นไม้เอวีแอล (AVL Tree) - การหมุนขวา

การหมุนขวา

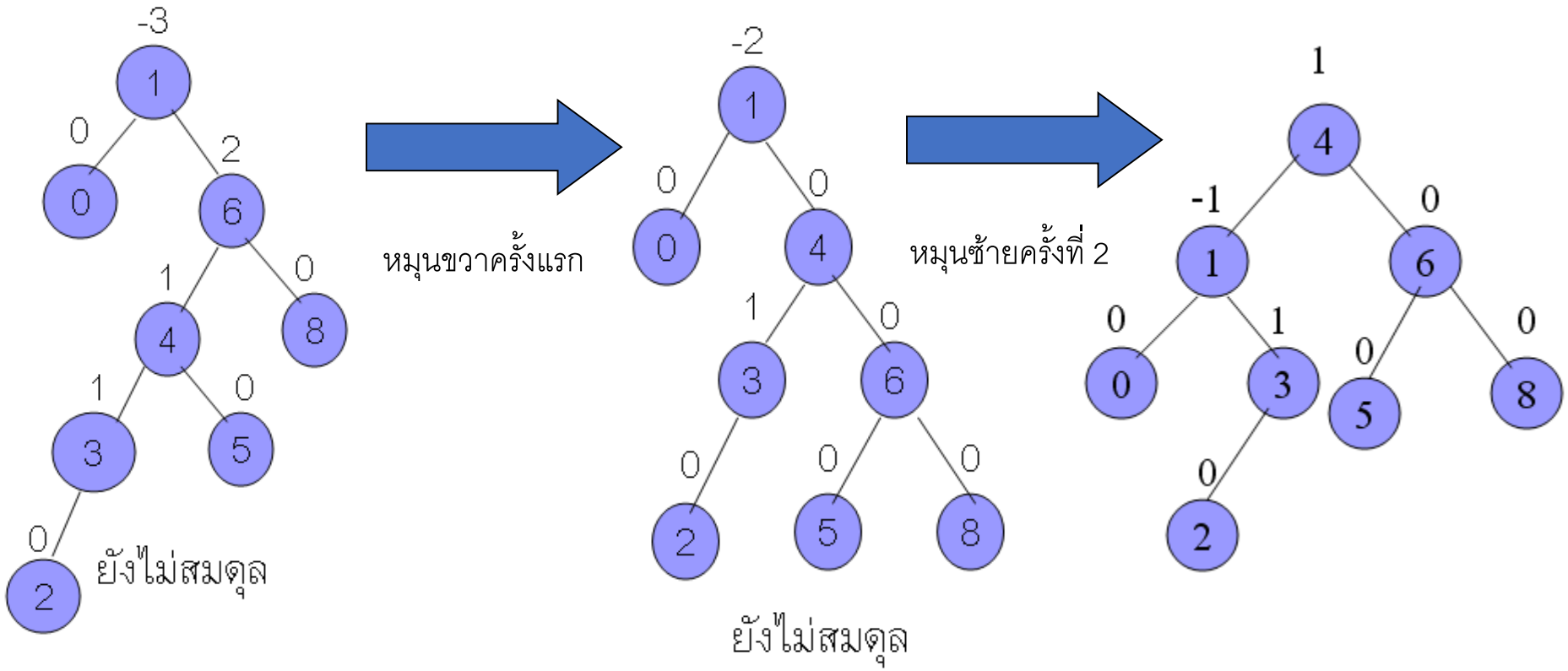


ต้นไม้เอวีแอล (AVL Tree) - การหมุนขวา

```
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}
```



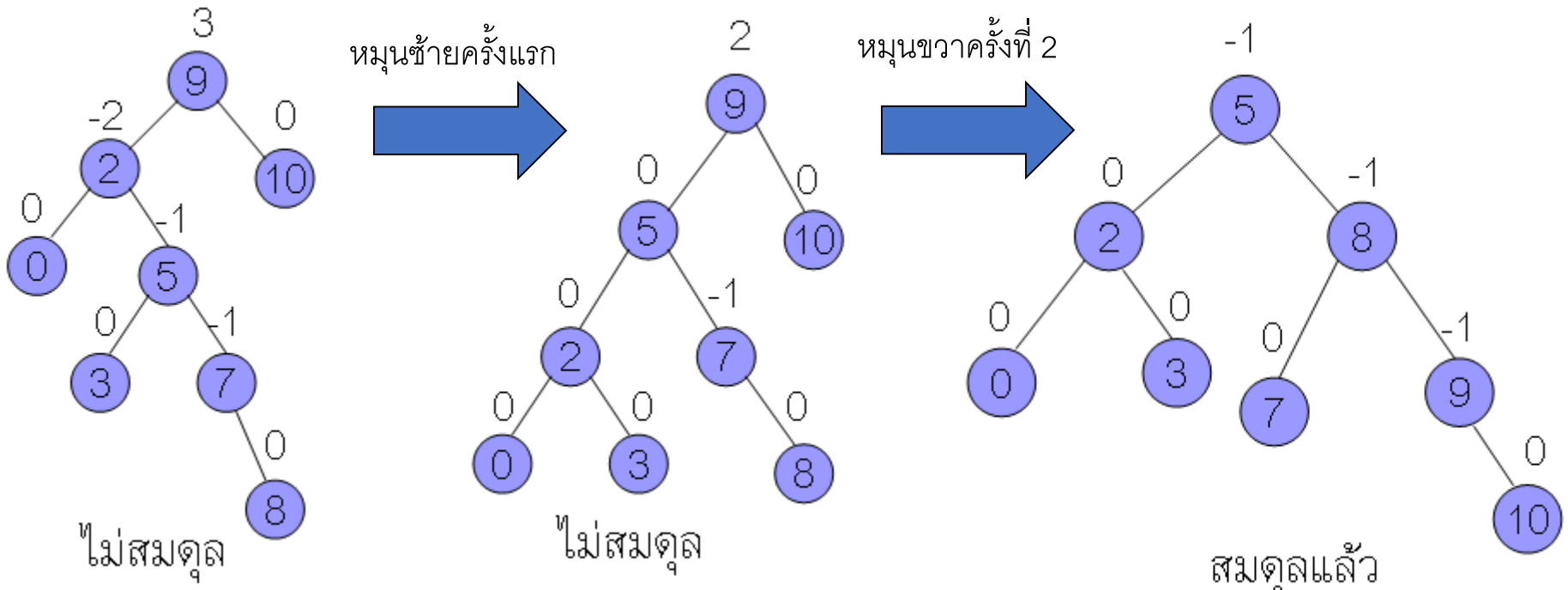
ต้นไม้เอวีแอล (AVL Tree) – การหมุนขวาไปซ้าย RightLeftCase



ต้นไม้เอวีแอล (AVL Tree) – การหมุนขวาไปซ้าย RightLeftCase

```
if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
```

ต้นไม้เอวีแอล (AVL Tree) – การหมุนซ้ายไปขวา LeftRightCase



ต้นไม้เอวีแอล (AVL Tree) – การหมุนซ้ายไปขวา LeftRightCase

```
if (balance > 1 && key > node->left->key)
    {
        node->left =  leftRotate(node->left);
        return rightRotate(node);
    }
```

LeftLeftCase

```
if (balance > 1 && key < node->left->key)
        return rightRotate(node);
```

RightRightCase

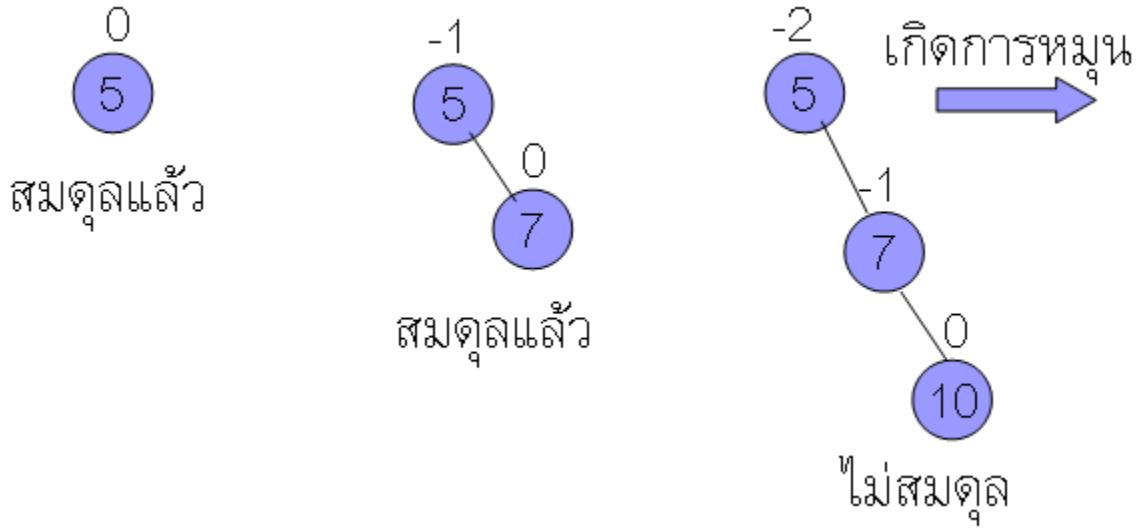
```
if (balance < -1 && key > node->right->key)
        return leftRotate(node);
```

การใส่โหนดใหม่เข้าไปใน AVL Tree

ถ้า node เข้าไปใหม่ทำให้ Tree เสียสมดุล ทำให้สมดุลได้โดยการหมุน

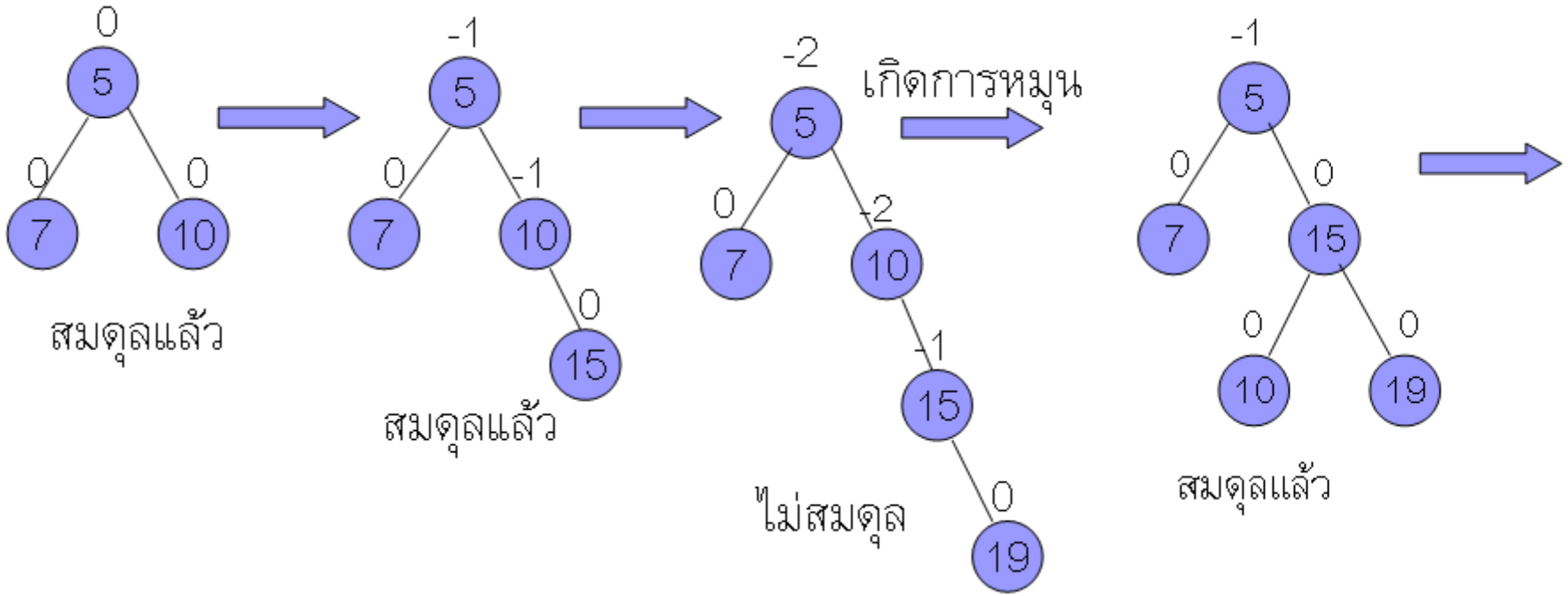
ตัวอย่าง

ใส่ node 5,7,10,15,19,20



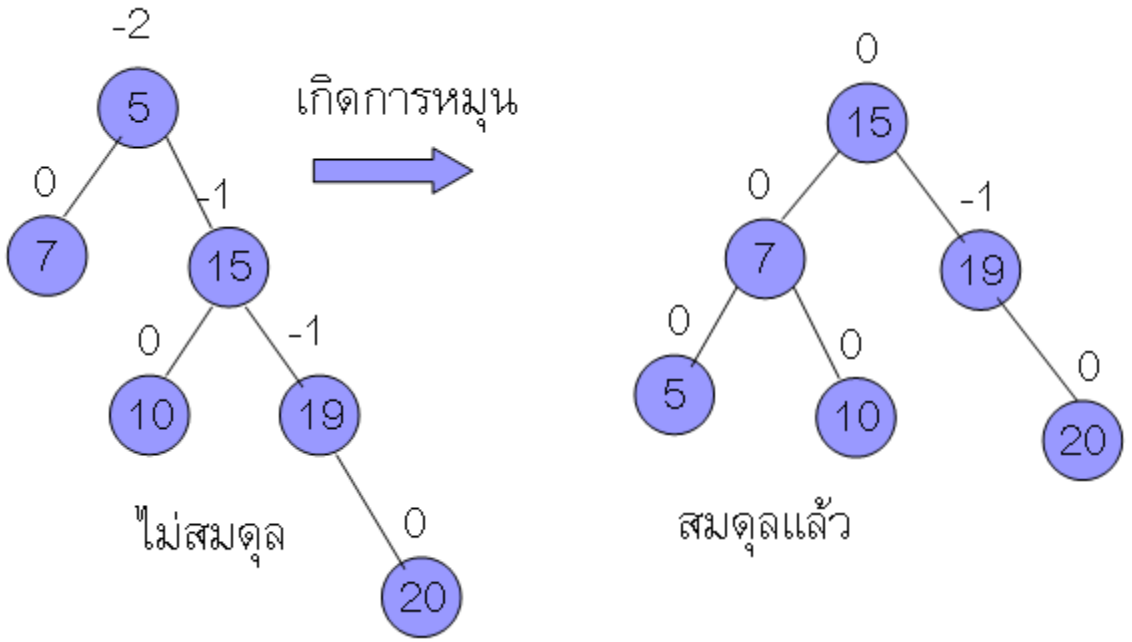
การใส่โหนดใหม่เข้าไปใน AVL Tree

ใส่ node 5,7,10,15,19,20



การใส่โหนดใหม่เข้าไปใน AVL Tree

ใส่ node 5,7,10,15,19,20



Insert

```
struct Node* insert(struct Node* node, int key)   // n
{
    /* 1.  Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key);     // n/2
    else if (key > node->key)
        node->right = insert(node->right, key);    // n/2
    else // Equal keys are not allowed in BST
        return node;
```

$O(\log n) = \Omega(\log n) \rightarrow \Theta(\log n)$

BST - $O(n)$, $\Omega(\log n)$

Insert

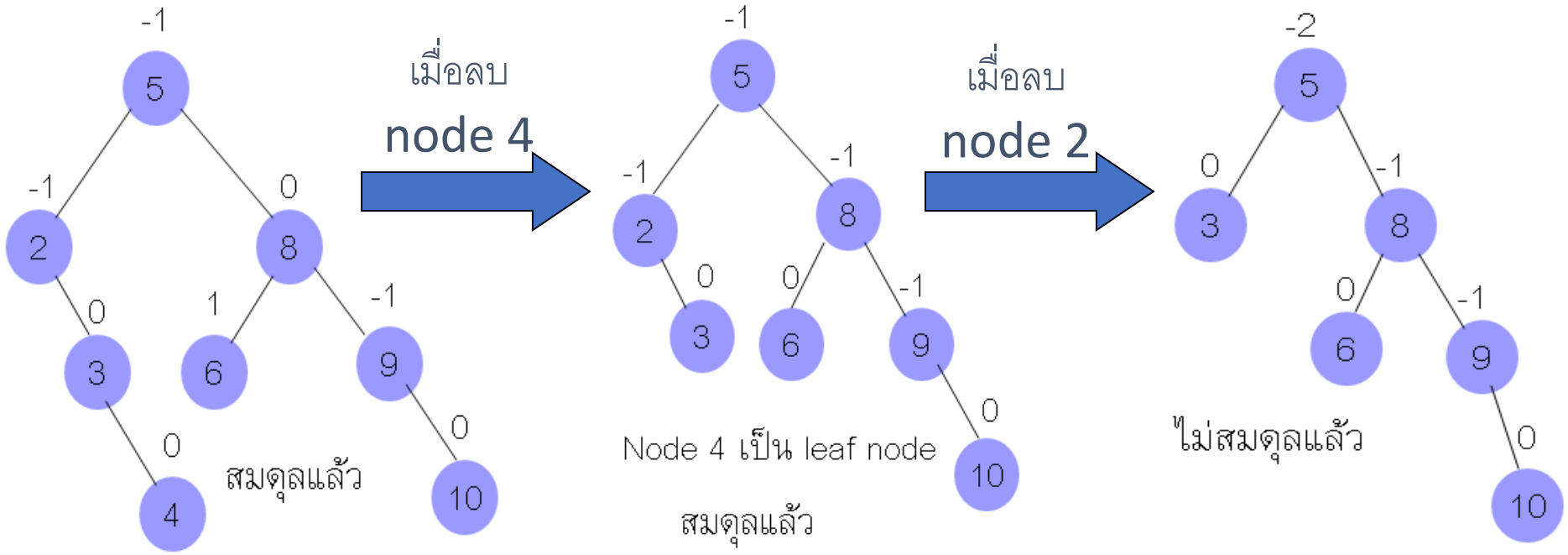
```
/* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                           height(node->right));
```

Insert

```
    /* 3. Get the balance factor of this ancestor node to check whether this node became
unbalanced */
    int balance = getBalance(node);
    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {   node->left =  leftRotate(node->left);
        return rightRotate(node);     }
    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {   node->right = rightRotate(node->right);
        return leftRotate(node);     }
return node;
}
```

การลบ node ใน AVL TREE

- ถ้าลบโหนดออกไปแล้วเกิดเสียสมดุล ก็ทำให้ต้นไม้สมดุล โดยการหมุน



Delete

```
struct Node* deleteNode(struct Node* root, int key)
{
 // STEP 1: PERFORM STANDARD BST DELETE

 // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE

 // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to
    // check whether this node became unbalanced)
  // If this node becomes unbalanced, then there are 4 cases

}
```

$O(\log n) = \Omega(\log n) \rightarrow \Theta(\log n)$

Priority queue representations

Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted array	$O(n)$	$\Theta(1)$
Sorted linked list	$O(n)$	$\Theta(1)$
Max heap	$O(\log_2 n)$	$O(\log_2 n)$

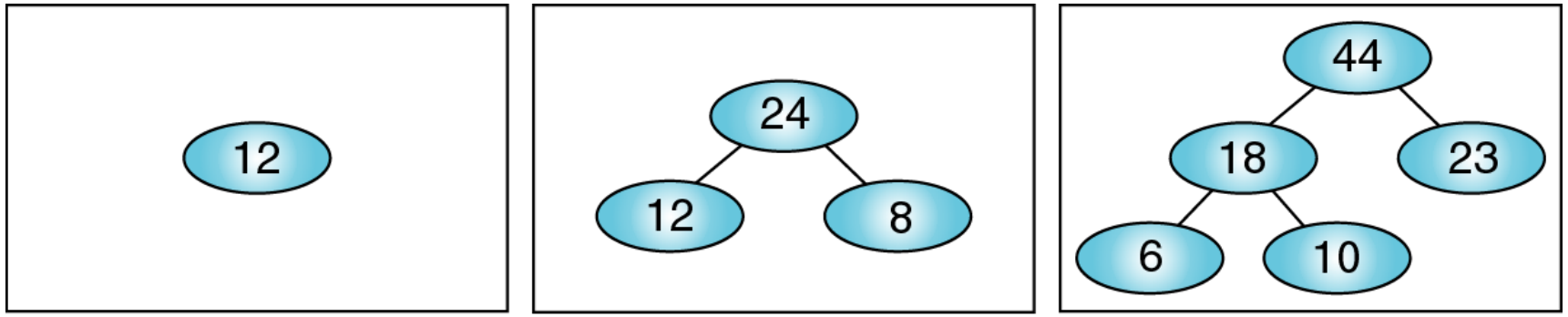
โครงสร้างข้อมูลแบบ Heap (กองซ้อน)

- ต้นไม้แบบฮีป (Heap Tree)

  เป็นการจัดโครงสร้างข้อมูลแบบต้นไม้ไบนารีเกือบสมบูรณ์แบบหนึ่ง โดยที่ต้นไม้ไบนารีเกือบสมบูรณ์แบบนี้ มีข้อกำหนดว่า ค่าที่บรรจุอยู่ภายในโหนดใดๆ ที่เป็นโหนดพ่อจะต้องมีค่าสูงกว่าหรือเท่ากับโหนดลูกทั้งสองของมัน (Max Heap Tree)

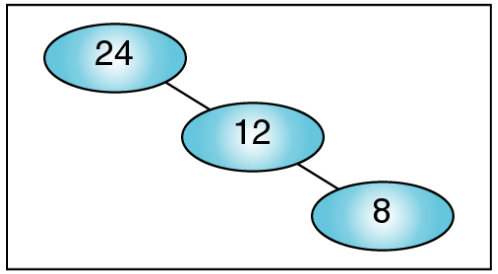
ต้นไม้แบบฮีป (Heap Tree)

- ตัวอย่างต้นไม้แบบฮีป

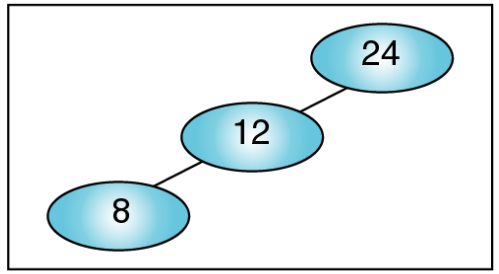


ต้นไม้แบบฮีป (Heap Tree)

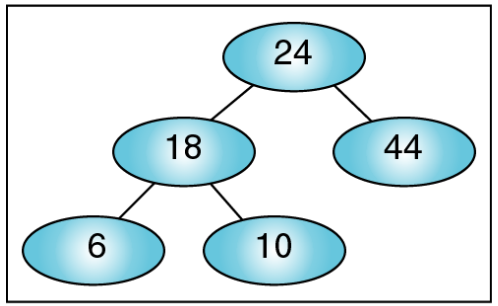
- ตัวอย่าง ที่ไม่ถูกต้อง



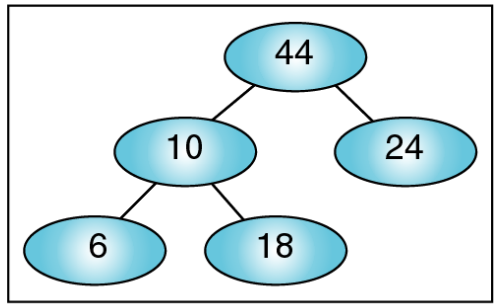
(a) Not nearly complete (rule 1)



(b) Not nearly complete (rule 1)



(c) Root not largest (rule 2)

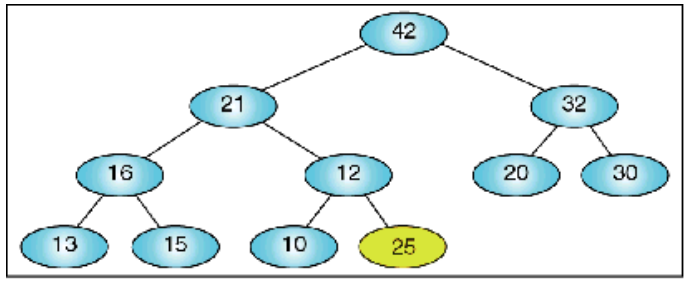


(d) Subtree 10 not a heap (rule 2)

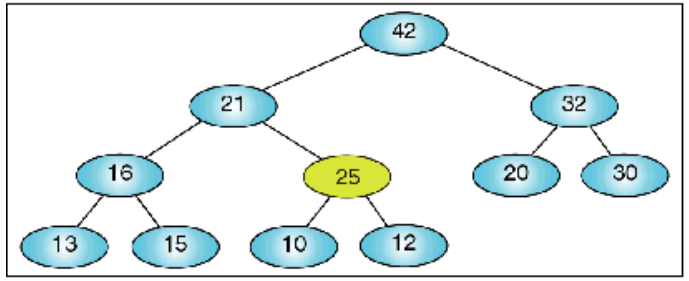
ต้นไม้แบบฮีป (Heap Tree)

Reheap Up

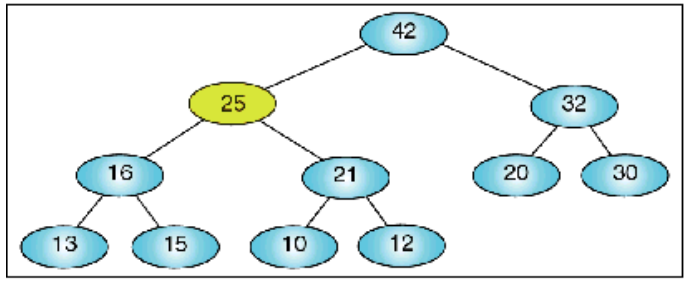
สลับ Node กับ Parent จนกว่าจะถูกตำแหน่ง



(a) Original tree: not a heap



(b) Last element (25) moved up

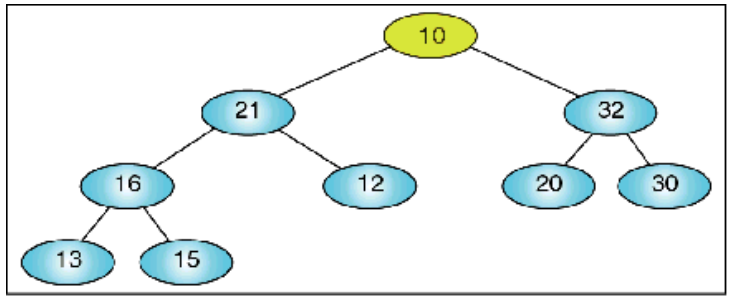


(c) Moved up again: tree is a heap

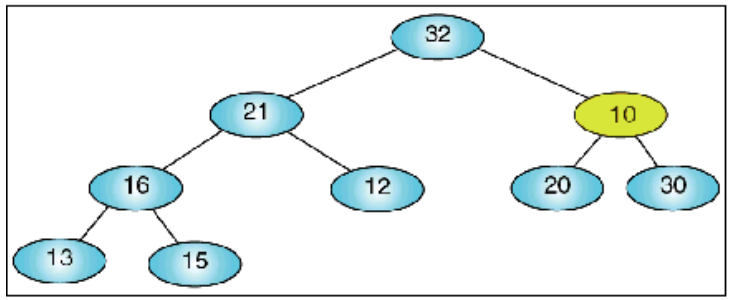
ต้นไม้แบบฮีป (Heap Tree)

Reheap Down

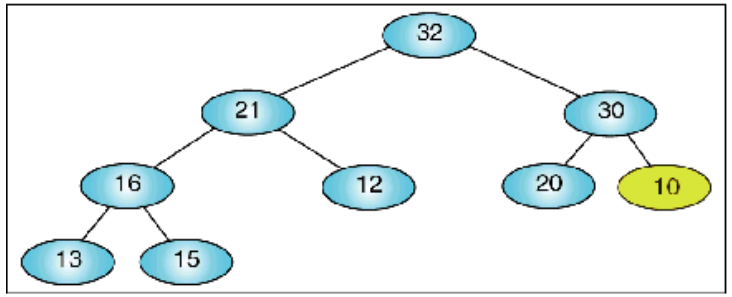
สลับ Node กับ Child ที่มีค่ามากกว่า



(a) Original tree: not a heap



(b) Root moved down (right)



(c) Moved down again: tree is a heap

ต้นไม้แบบฮีป (Heap Tree)

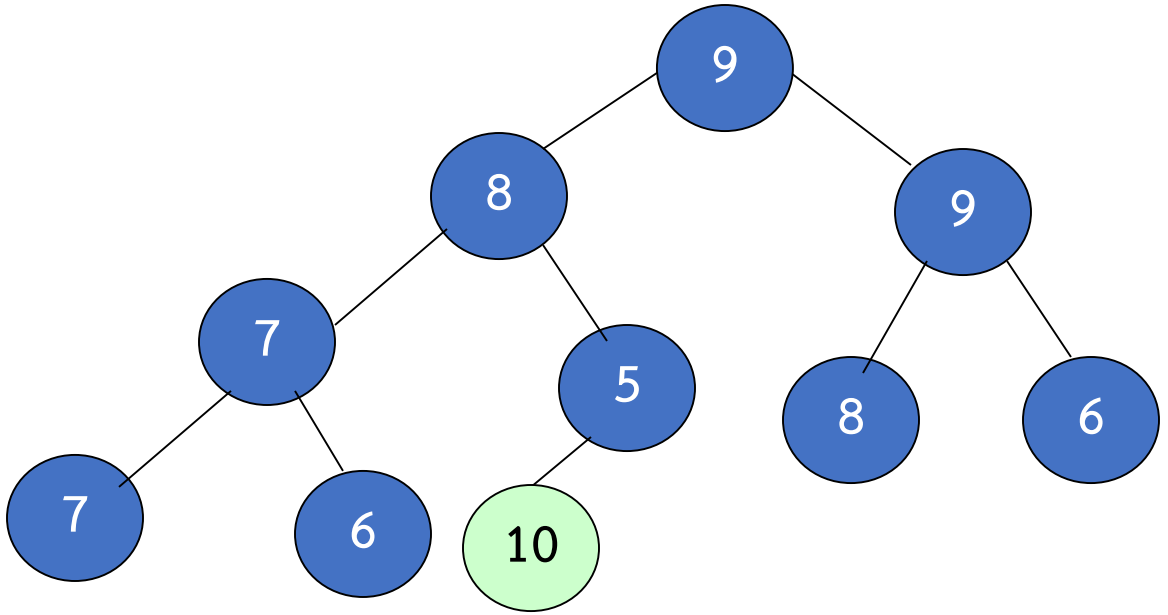
การใส่ข้อมูลเข้าไปในฮีปมีขั้นตอนดังนี้

1. ใส่ข้อมูลเข้าไปที่โหนดสุดท้ายของฮีป
2. ถ้าข้อมูลที่ใส่เข้าไปนั้นทำให้ได้เป็นต้นไม้ฮีปการใส่ก็เสร็จสิ้น
3. ถ้าต้นไม้ที่ได้ไม่เป็นต้นไม้ฮีปให้ทำการปรับต้นไม้ที่ได้ให้เป็นฮีป โดยการเปรียบเทียบค่าที่ใส่เข้ามาใหม่กับโหนดพ่อแม่

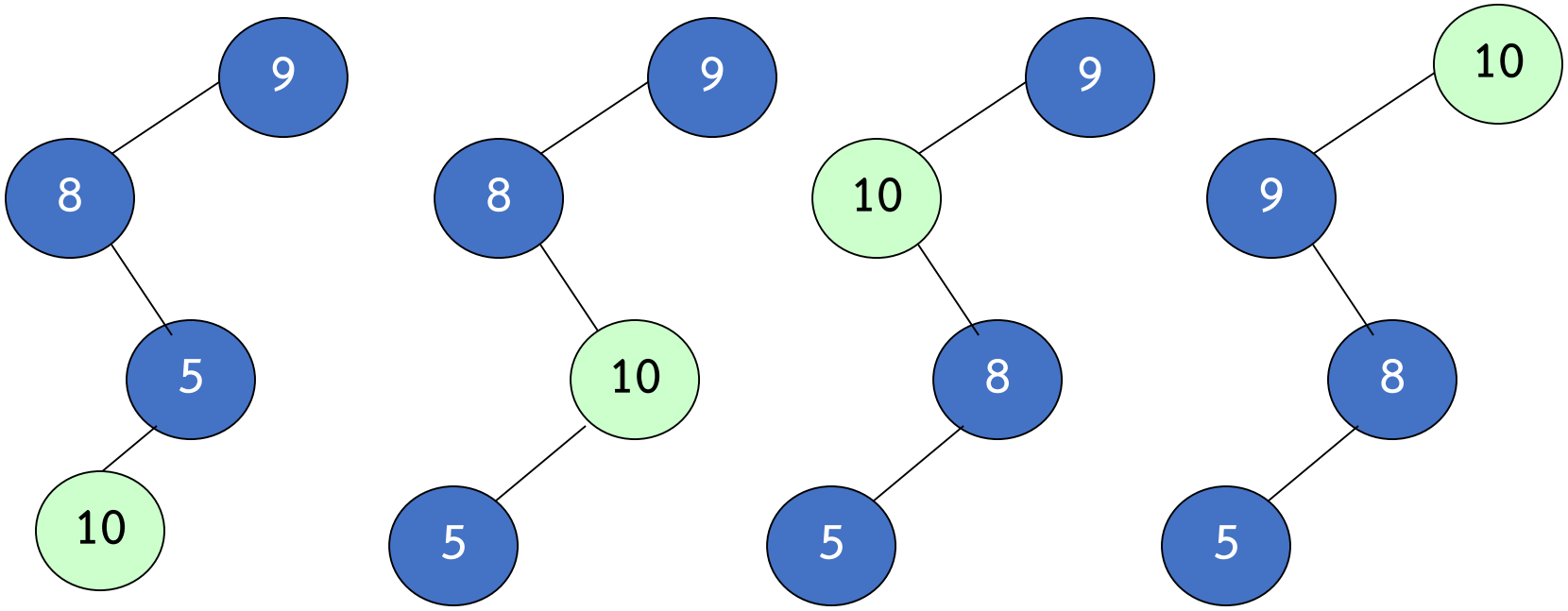
    ถ้าโหนดที่ใส่เข้าไปมีค่ามากกว่าให้สลับที่ ทำเรื่อยๆจนกระทั่งปรับสิ้นสุด

ต้นไม้แบบฮีป (Heap Tree)

- เติมค่า 10 เข้าไปในต้นไม้ฮีป

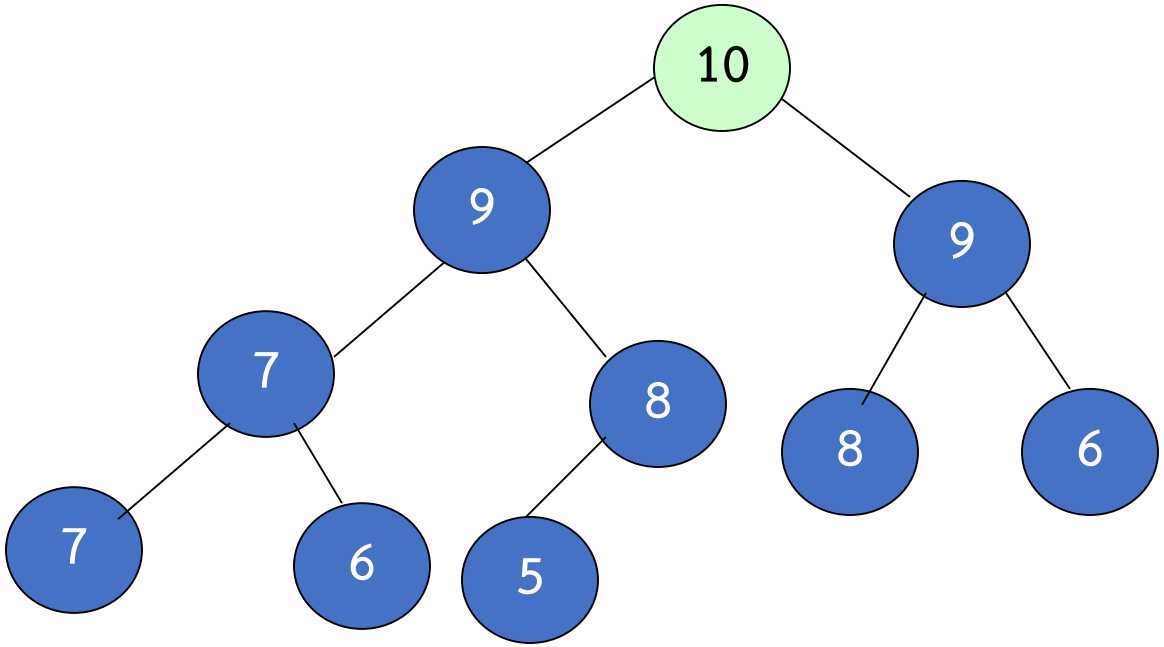


ต้นไม้แบบฮีป (Heap Tree)



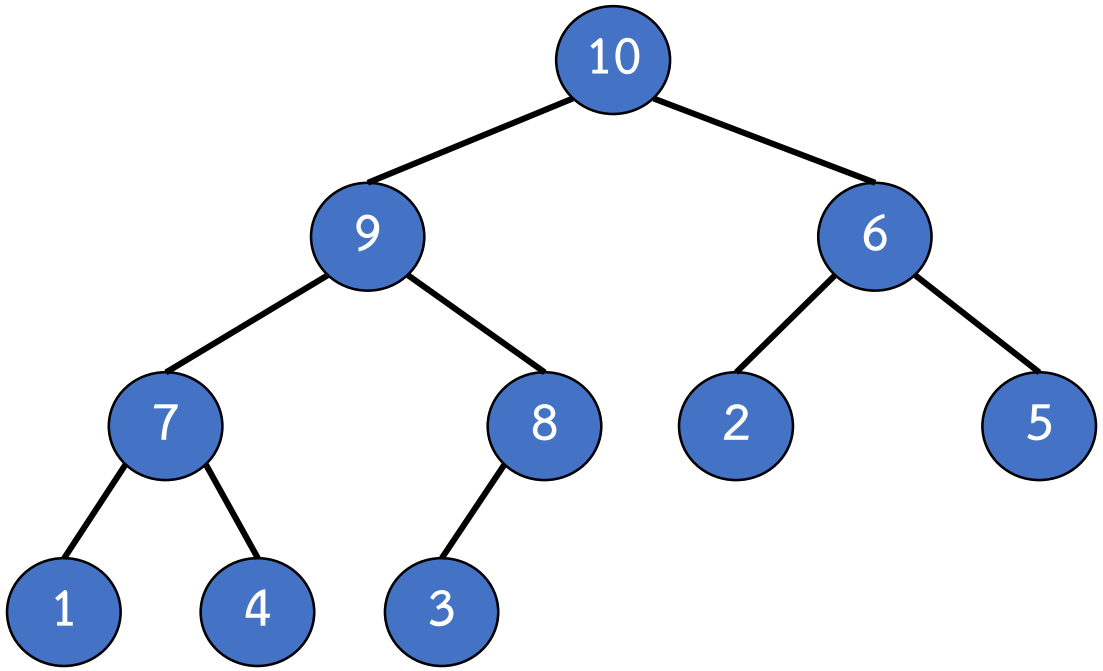
ต้นไม้แบบฮีป (Heap Tree)

- ฮีปใหม่ที่ได้

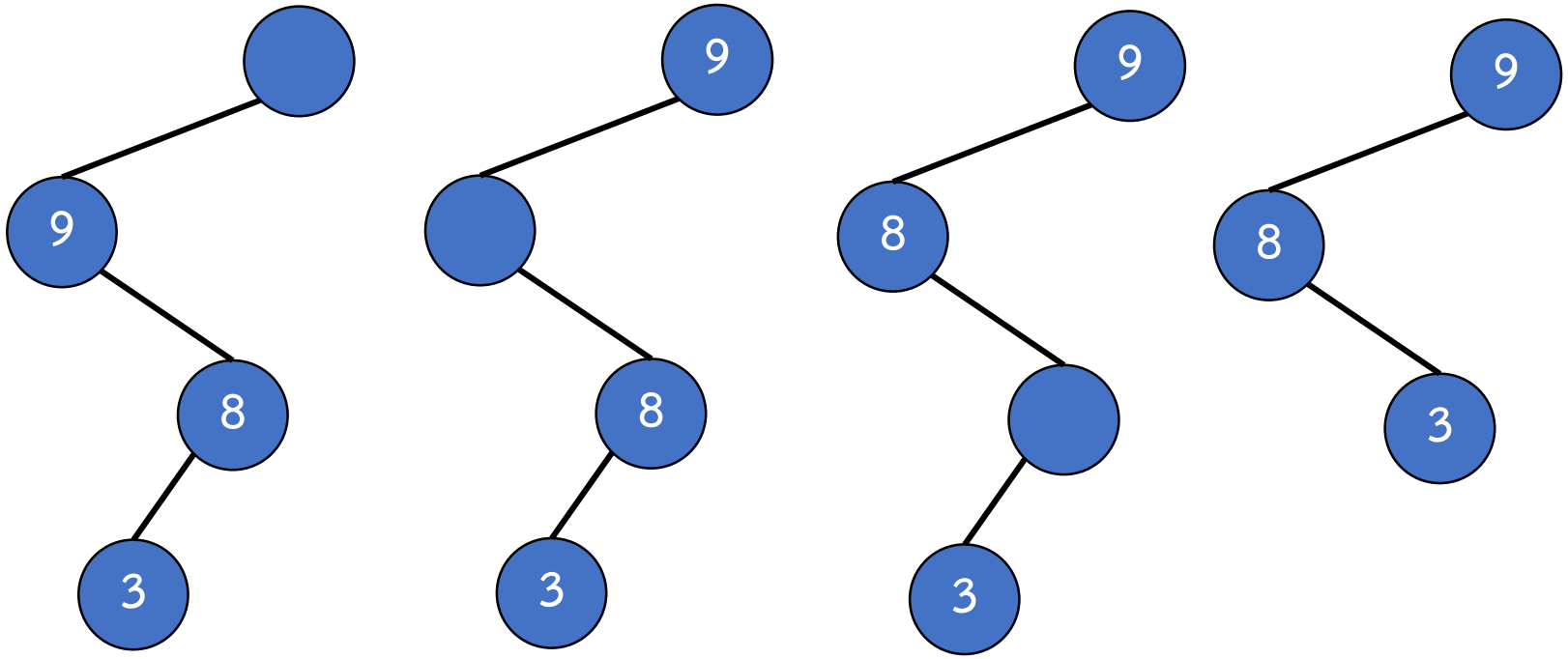


ต้นไม้แบบฮีป (Heap Tree)

- ลบค่า 10 ออกจากฮีป

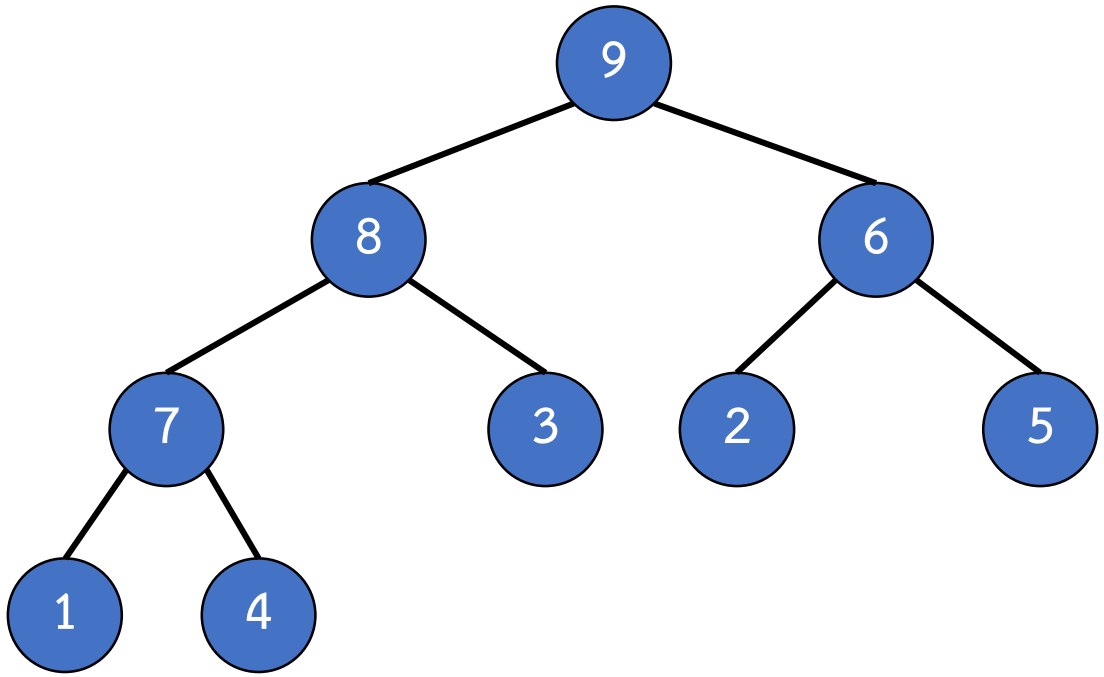


ต้นไม้แบบฮีป (Heap Tree)



ต้นไม้แบบฮีป (Heap Tree)

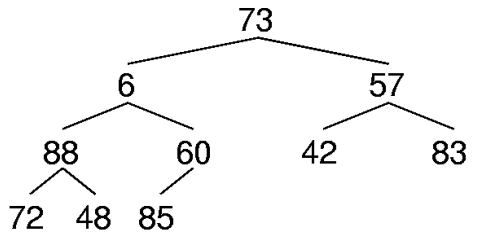
- ฮีปใหม่ที่ได้



Heap Sort

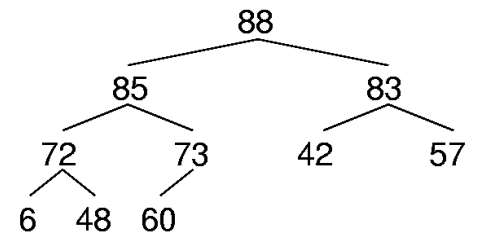
Original Numbers

73	6	57	88	60	42	83	72	48	85



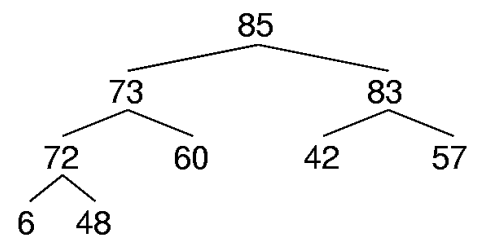
Build Heap

88	85	83	72	73	42	57	6	48	60



Remove 88

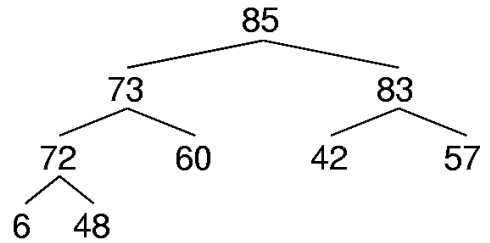
85	73	83	72	60	42	57	6	48	88



Heap Sort

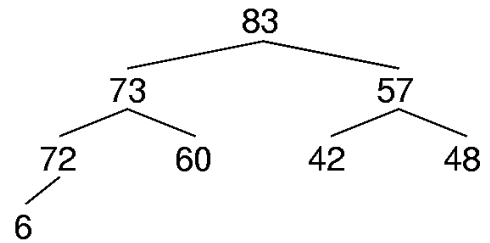
Remove 88

85	73	83	72	60	42	57	6	48	88



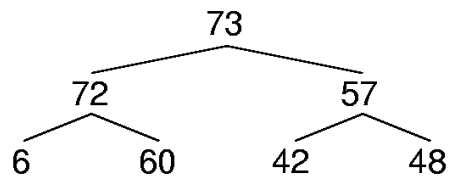
Remove 85

83	73	57	72	60	42	48	6	85	88



Remove 83

73	72	57	6	60	42	48	83	85	88



MinHeap with Array ADT

```
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    MinHeap(int capacity);
    void MinHeapify(int );
    int parent(int i) { return (i-1)/2; }
    int left(int i) { return (2*i + 1); }
    int right(int i) { return (2*i + 2); }
    int extractMin();
    void decreaseKey(int i, int new_val);
    int getMin() { return harr[0]; }
    void deleteKey(int i);
    void insertKey(int k);
};
```

Constructor

```
// Constructor: Builds a heap from an array a[]
MinHeap::MinHeap(int cap)
{
    heap_size = 0;
    capacity = cap;
    harr = new int[cap];
}
```

Insert Key

```
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }
    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])   //n
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);                              //n/2
    }
}
```

O(height)=O(log n) != $\Omega(1)$

Decrease Key – swap upward

```
void MinHeap::decreaseKey(int i, int new_val)
{
    harr[i] = new_val;
    while (i != 0 && harr[parent(i)] > harr[i])    //n
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);                              //n/2
    }
}
```

O(height)=O(log n)

Extract Min

```
int MinHeap::extractMin()
{
    if (heap_size <= 0)
        return INT_MAX;
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Store the minimum value, and remove it from heap
    int root = harr[0];
    harr[0] = harr[heap_size-1];
    heap_size--;
    MinHeapify(0);

    return root;
}
```

O(MinHeapify)=???

Extract Min - MinHeapify

```
void MinHeap::MinHeapify(int i) // n
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;              // smallest=i/2
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;              // smallest=i/2
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);     // n/2
    }
}
```

O(MinHeapify)=O(log *n*)

Delete Key

```
void MinHeap::deleteKey(int i)
{
    // move to top
     decreaseKey(i, INT_MIN);          //O(log n)
    // delete from top
     extractMin();                     //O(log n)
}
```

O(deleteKey)=O(Max(decreaseKey,extractMin))=O(log *n*)

ถ้าเราอยากได้ item ที่มีขนาดเล็กที่สุดจาก List จำนวน k ชิ้น

วิธีแรก Sorting + Select

เวลาที่ใช้ ?

วิธีที่สอง ดึงของออกจาก Heap k ชิ้น

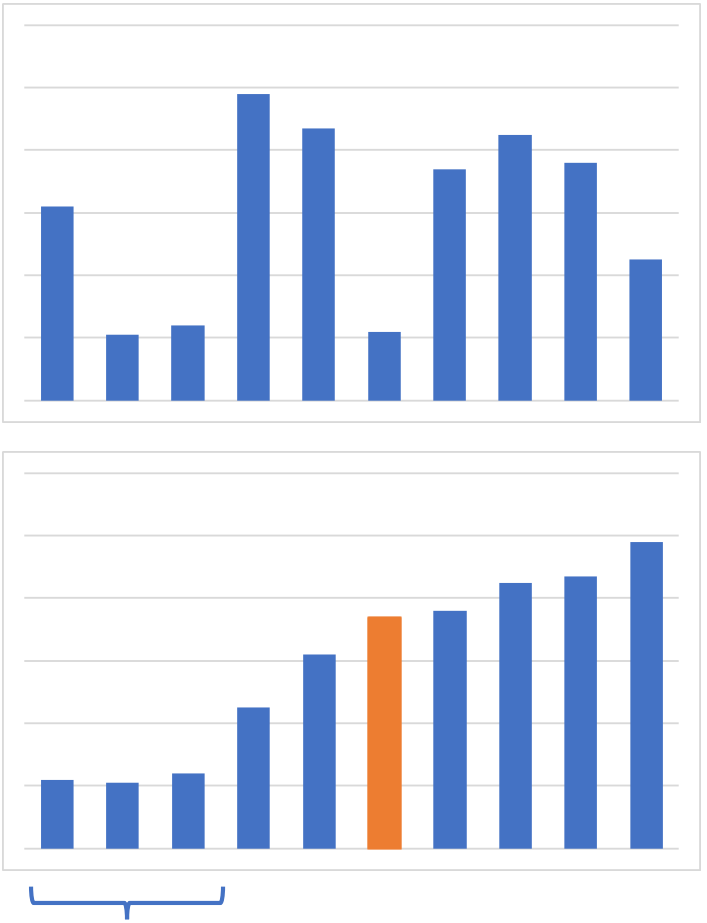
เวลาที่ใช้ ?

ถ้าของ k ชิ้นไม่ต้องเรียงกันก็ได้ล่ะ

Quick Selection (Partition and Push/Pull Back)

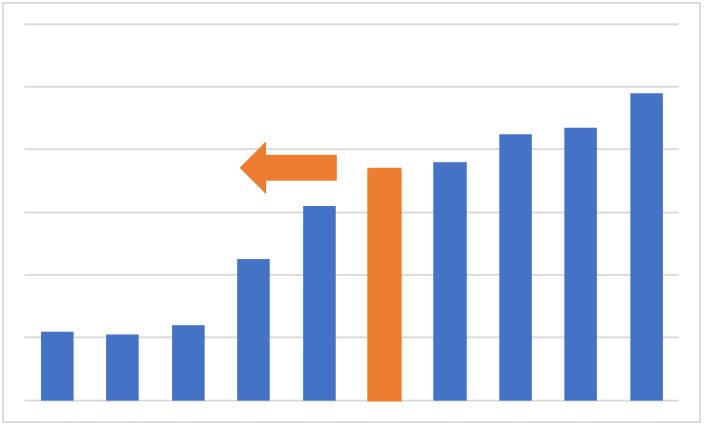
- ใน Quick Sort เรา รู้วิธีการหา Partition เพื่อแบ่งข้อมูลออกเป็นสองส่วน

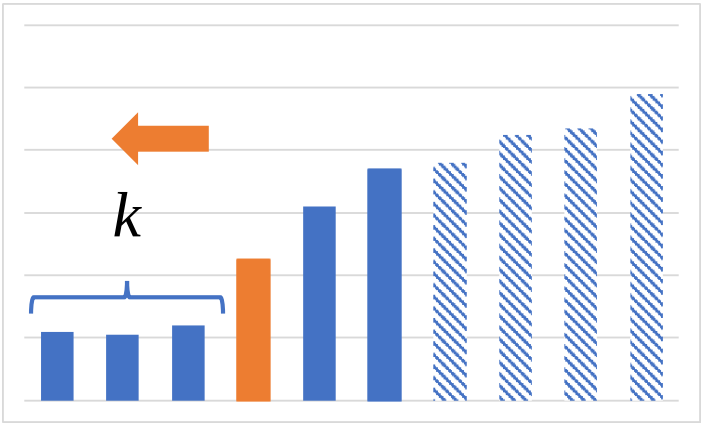
```
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++) {
        if (arr[j] <= x) {
            swap(arr[i], arr[j]);
            i++;
        }
    }
    swap(arr[i], arr[r]);
    return i;
}
```



Quick Selection (Partition and Push/Pull Back)

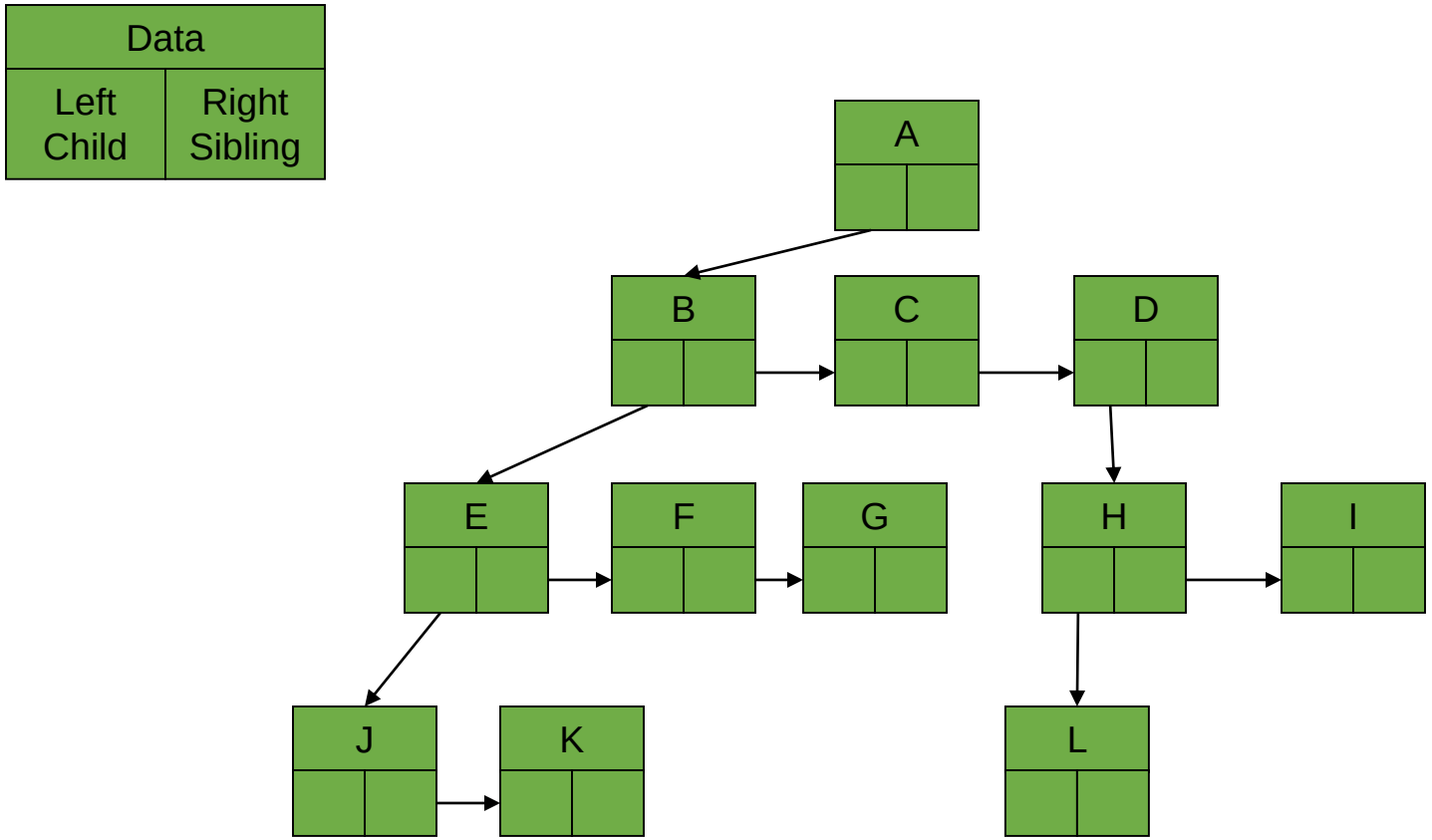
- ถ้ายังไม่ได้จำนวน k ตัวที่อยากได้ก็แค่ดึงหรือว่าดัน partition ต่อไป



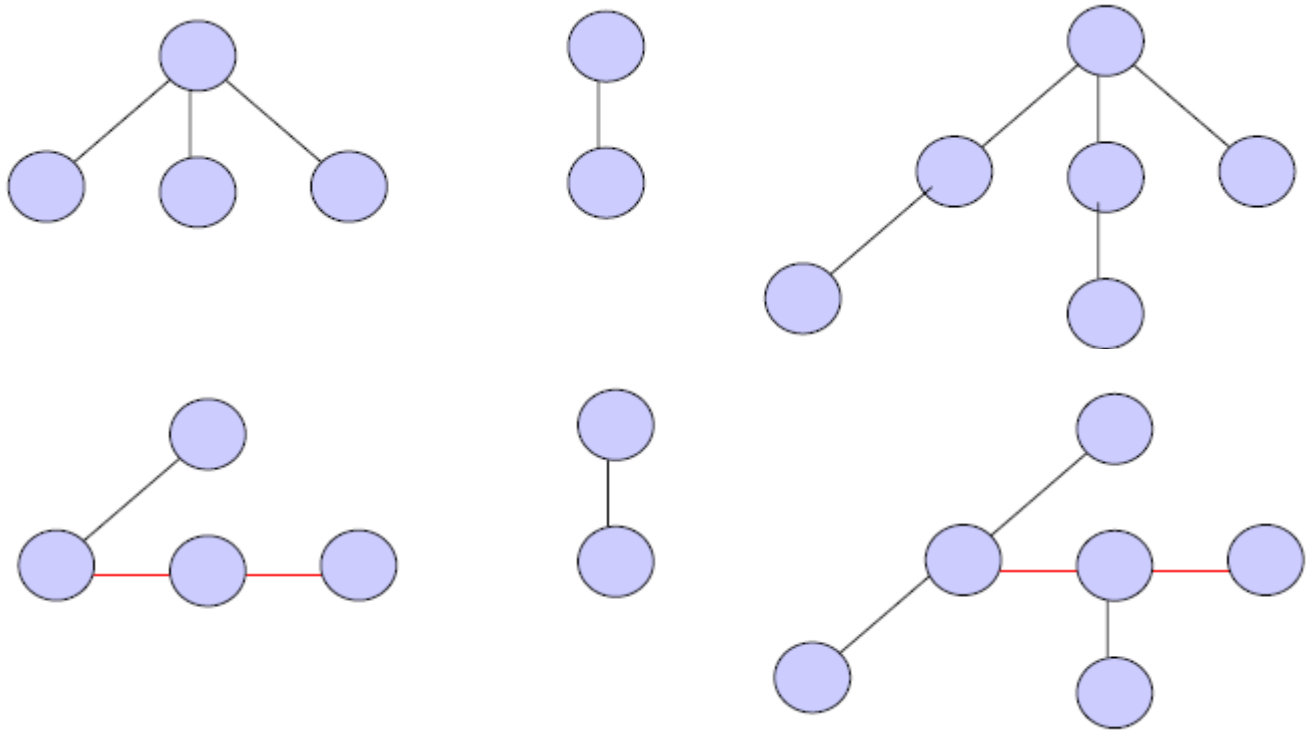



```
int kthSmallest(int arr[], int l, int r, int k)
{
    if (k > 0 && k <= r - l + 1) {
        int index = partition(arr, l, r);

        if (index - l == k - 1)
            return arr[index];

        // for left subarray
        if (index - l > k - 1)
            return kthSmallest(arr, l, index - 1, k);
        // else for right subarray
        return kthSmallest(arr, index + 1, r, k - index + l - 1);
    }
    return INT_MAX; //Index out of bound
}
```

General Tree - Left Child, Right Sibling Representation



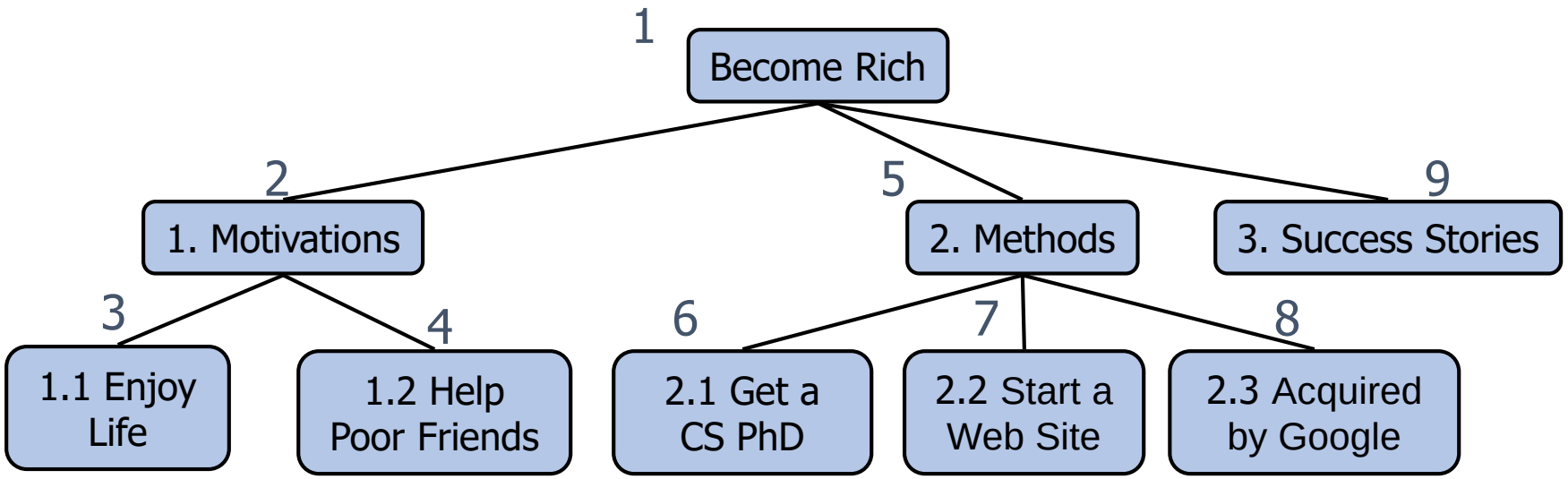
การแปลงต้นไม้ทั่วไปเป็นโครงสร้างต้นไม้ในโปรแกรม



Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

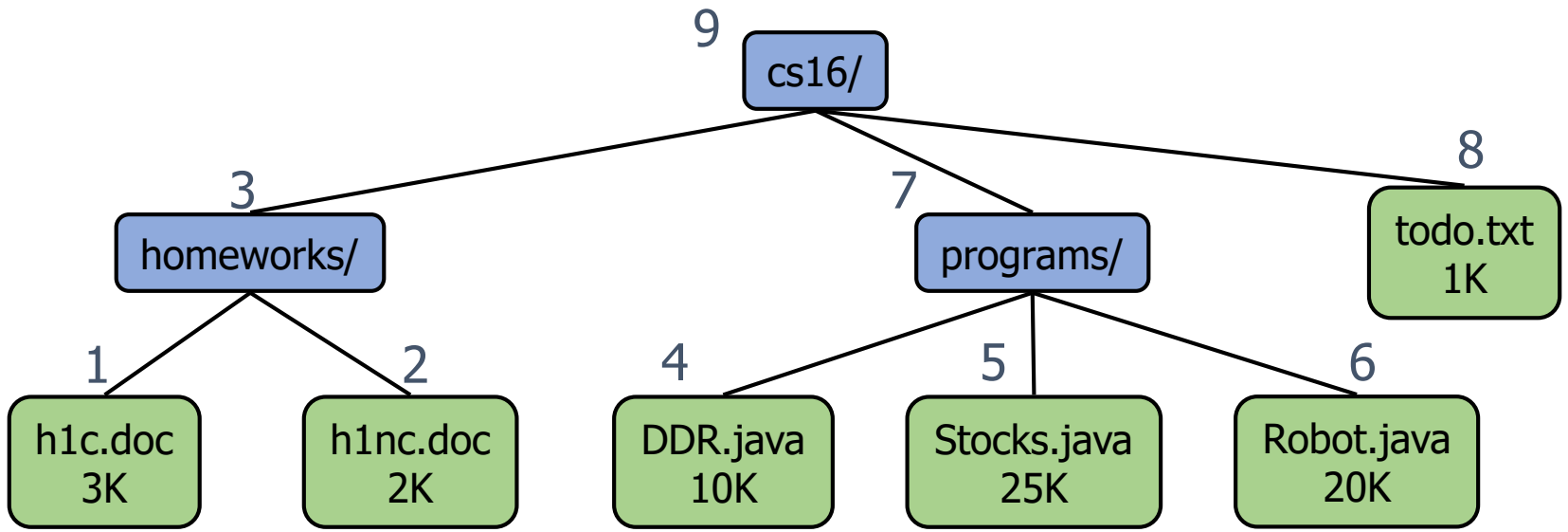
```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preorder (w)
```



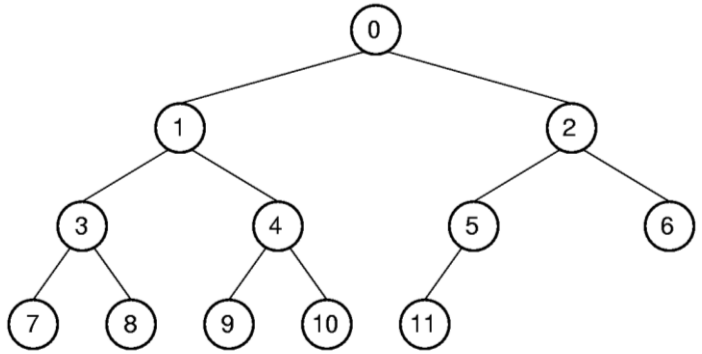
Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

```
Algorithm postOrder(v)
    for each child w of v
        postOrder (w)
    visit(v)
```

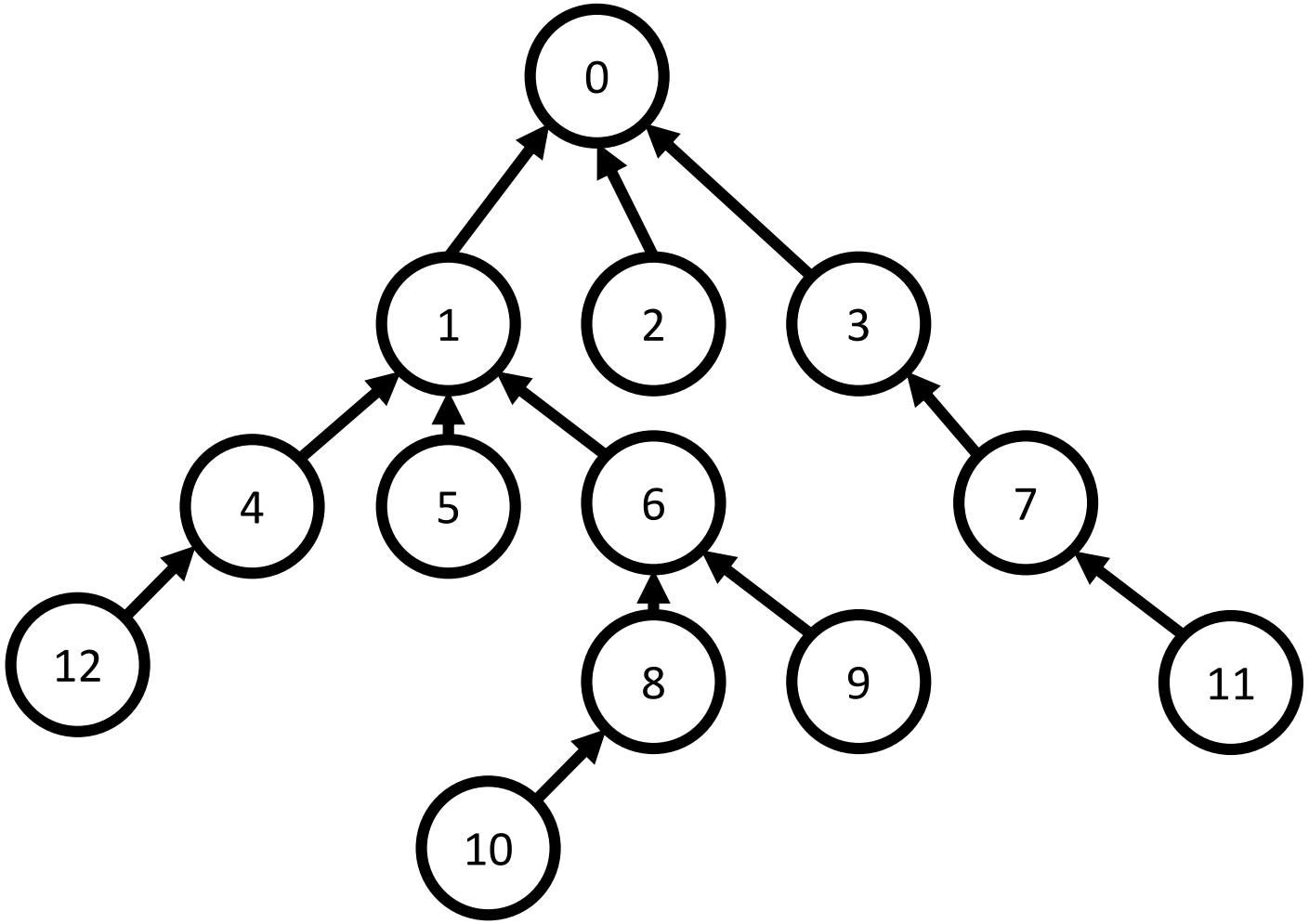


Up-Tree Implementation

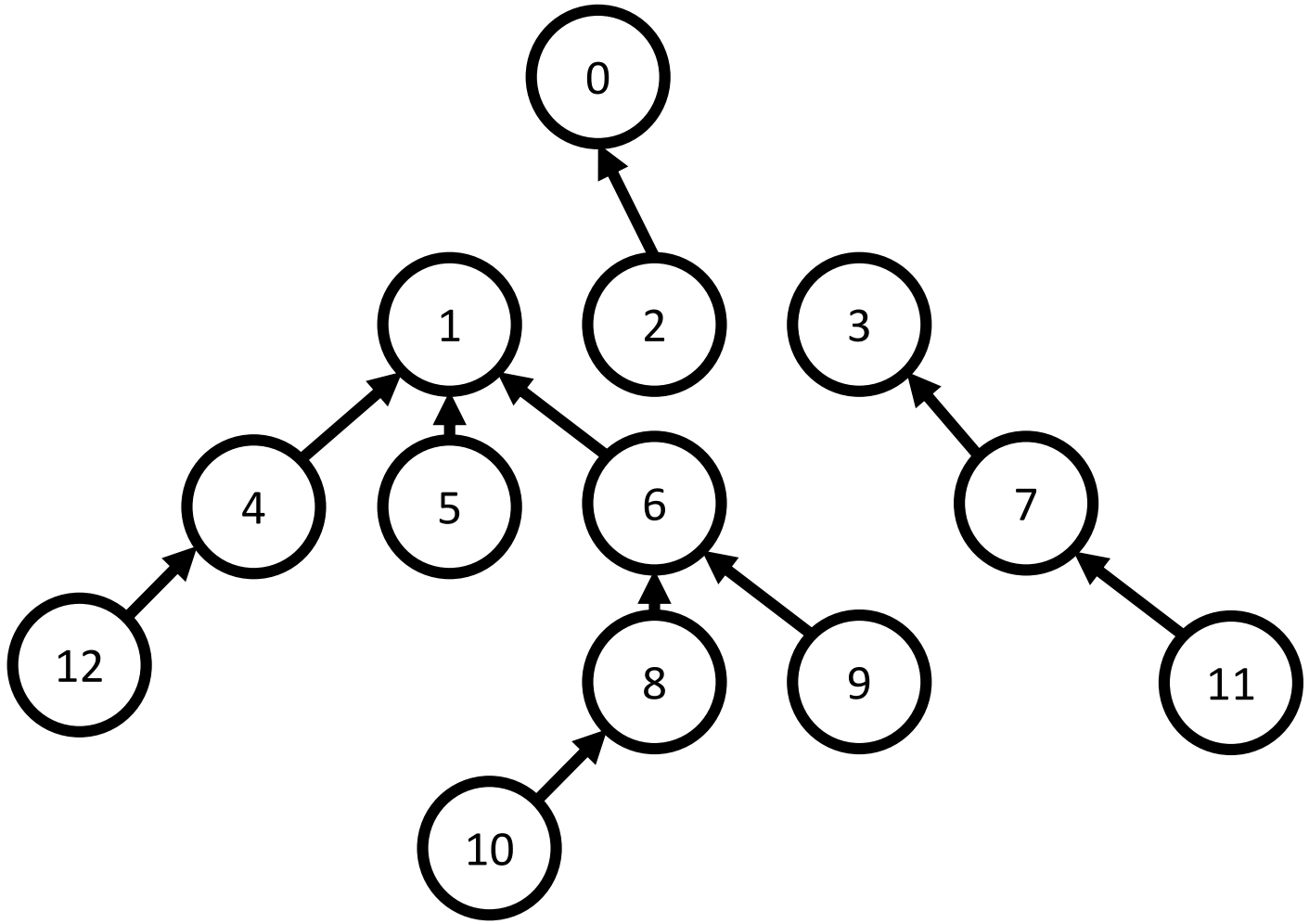


Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	--	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	--	--	--	--	--	--
Right Child	2	4	6	8	10	--	--	--	--	--	--	--
Left Sibling	--	--	1	--	3	--	5	--	7	--	9	--
Right Sibling	--	2	--	4	--	6	--	8	--	10	--	--

จงใช้ array สำหรับเก็บ up tree



จงใช้ array สำหรับเก็บ up tree



การบ้าน จงใช้ up tree สำหรับบันทึก state