

Mid-Sem ❤:-)

order/complexity	Time Adj
常数阶	constant
对数阶	Logarithmic
对数-平方阶	Log - squared
线性阶	Linear
对数-线性阶	Log-linear
二次阶	Quadratic
对数-二次阶	Log-quadratic
三次阶	Cubic
指数阶	Exponential
阶乘阶	Factorial

* ลองแทนค่าจะเห็นภาพ

Big-Theta (Θ)

- $g(n) = \Theta(f(n))$ อ่านว่า $f(n)$ เป็น Big Theta ของ $g(n)$.
- ปิดบนปิดล่าง มี Upper-lower bound
- $K_1 f(n) \leq g(n) \leq K_2 f(n)$
- $3n+100$ ทั้งตัว degree สูสูด \therefore ตาม n , $6n^2+7n+30$ ตาม n^2
ex. $K_1 n^2 \leq 6n^2+7n+30 \leq K_2 n^2$
 $5n^2 \leq 6n^2+7n+30 \leq 7n^2$
 $0 \leq n^2+7n+30$ (แล้วเวลา 2 ช่วงมา ก กัน)

```
for i=1:n
    print(i) ∴ Big theta = n
```

การวนลูป 2 รอบ Big theta = n^2 แต่ถ้า loop โนน · วน
Big theta = n เป็นๆ

```
for i=1:n
    for i=i           ∴ ทั้ง  $\frac{n(n+1)}{2}$  “แล้ว  $\Theta = n^2$ 
        print(i)
```

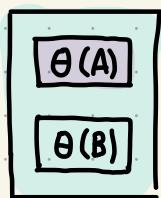
```
for i=½:n
    print(i)           Big theta = กอยู่
                        แบ่ง: กที่ร่องเดียว
```

```
for i = n-1; n
    print(i)
```

ตอบ 2 例
 $n=100$; $n-1=95$
 \therefore จะ 2 ครั้ง

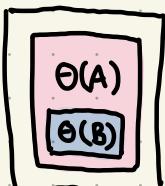
กรณีไม่ใช่สมการเส้นตรง

- เอกสาร Big theta ต้อง ถ้าตอบเป็น \log ตอบฐานไรก็ได้
 bc มันแปลงฐานได้ $\log_a n = \log_b n \times \frac{1}{\log_b a}$
- $3n\sqrt{n} + \sqrt{n} + 2n^2\sqrt{n} + n + 10$, $\therefore \Theta = n^2\sqrt{n} = n^{\frac{5}{2}}$
 $K_1 = 1$ and $K_2 = 3$ ($n \geq 9$?)



เวลาเขามาบวกกัน
 $\max(\Theta A, \Theta B)$
 (compare บันทึกกันตอบอันนั้น)
 แล้ว \max ที่โต๊ะ, Algo ซึ่ง

$$\begin{aligned}\Theta(\log N) + \Theta(N) &= \Theta(N) \\ \Theta(N \log N) + \Theta(N) &= \Theta(N \log N) \\ \Theta(N \log N) + \Theta(N^2) &= \Theta(N^2) \\ \Theta(2^N) + \Theta(N^2) &= \Theta(2^N)\end{aligned}$$



เวลาคูณกัน (loop ซ้อนกัน, func รีเควฟ์func, recursive)
 เอาท์ 2 มาคูณกันเลย

$$\begin{aligned}\Theta(\log N) * \Theta(N) &= \Theta(N \log N) \\ \Theta(N \log N) * \Theta(N) &= \Theta(N^2 \log N) \\ \Theta(N) * \Theta(1) &= \Theta(N) \\ \Theta(N) * \Theta(2^N) &= \Theta(N \times 2^N)\end{aligned}$$

Big-O (O) : worst case

- คือ ปิดหนอย่างเดียว = ช้าสุดไม่เกินนี้
- running time $\leq K \times f(n)$
- $T(n) = 2n^2 + 3n^3 + 100$ ถ้า $n \geq n+1$ $\leq 4n^3$
 $T(n) = O(n^3)$ ช้าสุดไม่เกินนี้
- $T(n) = (n+1)(1+n\log n) + \log n^{20}$
 $\text{in } n^2 \log n$
- $T(n) = O(n^2 \log n)$

ex. $2n^2 + 3n^3 + 100 \leq 4n^3$
 $0 \leq n^3 - 2n^2 - 100$
 $\therefore n \geq 6$ ทำให้สมการเป็นจริง

Order	Time Adj.
$O(1)$	const.
$O(\log n)$	Logarithmic
$O(\log^2 n)$	Log-squared
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^2 \log n)$	Log-quadratic
$O(n^3)$	Cubic
$O(a^n), a > 1$	Exponential
$O(n!)$	Factorial

- $5000 = O(n)$
- $n^2 = O(n^3)$
- $\log n = O(n)$
- $3^n = O(5^n)$
- take max เนื่องใน Θ

Big-Ω (Ω) : best case

- ปิดล่น คือเร็วสุดไม่เกิน ($f(n)$) = บันทึกไปกว่า $f(n)$
- $\Omega = \Omega(50000)$
- $n^3 = \Omega(n^2)$
- $n = \Omega(\log n)$
- $5^n = \Omega(3^n)$
- $\log_x n$ เปิด $\log_x n$ ถ้าห้องศึกษาแต่ยกกำลังไม่ได้ X

Abstract Data Type

อธิบายโครงสร้างข้อมูลแบบไม่สนใจ code
แต่ว่ารู้ว่ามันทำอะไรได้

→ วิจัยผู้ใช้ (วิจัยผู้ใช้กับ programmer)

- Queue** First come first serve, First in First out
- Priority queue** VIP always comes first (same rank, 1st come 1st serve)
- Balanced Trees** โครงสร้างข้อมูลที่平衡 (平衡)
- Mysterious Data structures** Big O เป็นการวิเคราะห์

// Array // : พน. ใน mem ที่ติดต่อกันเนื่องกัน ประกอบด้วย element
ที่มีขนาดเท่ากัน การเข้ากันด้วยนั้น เท่ากับ
- การเข้ากัน เป็น $O(1)$ ex. $a[5]$ $0x0007$ เลบฐาน 16 หรือ 10_{16}

$$\text{accessing_addr} = \text{array_addr} + \text{element_size} \times (\text{i} - \text{first_index})$$

& array ตัวแรก
แต่ละ block ที่อยู่ติดกัน
ต้องห่างกัน element_size

Array 2 มิติ

$$\text{accessing_addr} = \text{array_addr} + \text{element_size} \times ((\text{row} - 1) \times \text{L} + (\text{col} - 1))$$

& array ตัวแรก
แต่ละ block ที่อยู่ติดกัน
มี row และ col

Row Major Indexing : Fill row first, then column

$L=6$	1	2	3	4	5	6
1	0	1	2	3	4	5
2	6	7	8	9	10	11
3	12	13	14	15	16	17

$$\text{row } z = (x-1) \times L + (y-1)$$

$$= (3-1) \times 6 + (4-1)$$

$$= 2 \times 6 + 3$$

$$12+3 = 15$$

(3,4)

1. $(x,y) \therefore x=3, y=4$
2. ความยาวของ $y=L$
3. ใช้สูตรจะได้ค่าใน array

Column major Indexing

$L=3$	1	2	3	4	5	6
1	0	3	6	9	12	15
2	1	4	7	10	13	16
3	2	5	8	11	14	17

$$\text{col } z = (x-1) \times L + (y-1)$$

$$z = (4-1) \times 3 + (3-1) = 9+2 = 11$$

* จำ Row major 1. สับ x,y
2. คิด L ใหม่

Array as a data structure

- Add obj (method) ex. push ตัวหนึ่ง
- Remove obj
- size or length = field (ขนาด array ที่ตัว)
- Capacity ของพื้นที่

Add Obj

- Add ตัวสุดท้าย $a[\text{size}] = \text{value}, \text{size}++, \text{return } a[\text{size}-1]$

$O(1)$ เพราะเข้ากันแล้วรอบเดียว ต้อง $a[\text{size}] = 4$

กรณี condition ถ้า $\text{size} != \text{cap}$ ให้ return else print error

หรือ $\text{size} == \text{cap}$ แล้ว จึงส่งเพิ่ม error โดยเราใช้ condition ถ้า $\text{size} != \text{cap}$ {return} else {print error}

condition ถ้า $\text{size} == \text{cap}$ {return} else {print error}

- Add ตัวหนึ่งสุด ใช้ for loop = ต้องย้ายทุกตัวอยู่บนลับไป 1 step $O(n)$

Remove Obj

- ลบหนึ่งบันทึก → ลบได้เลย → ย้ายทุกตัวไปบันทึก $O(n)$

	Add	Remove	Read/Write
Beginning	O(n)	O(n)	O(1)
End	O(1)	O(1)	O(1)
Middle	O(n)	O(n)	O(1)
	ຈົນໆ $O(\frac{n}{2})$ ດີນປະສົງ $= O(n)$		



Summary : พก.ใน mem ต้องต่อเนื่องกัน แต่ละตัวมี size เท่ากัน
การเข้ากันเป็น constance function ($O(1)$)
การ add-remove ตัวกันยังบวกกันเป็น $O(1)$
ก้า add-remove ตัว ใหม่ๆ เป็น Linear func. $O(n)$

default vars int=0, float=0.0, String = "/o"

int[] myArray = <u>new int[5];</u>	0 0 0 0 0	default int = 0 float = 0.0 string = "0"
System.out.println(<u>myArray[0]</u>);	= 0	Your choice?
myArray[0] = 1;		1. Compilation error 2. Runtime Exception 3. Nothing 4. Print out numbers?
System.out.println(<u>myArray[0]</u>);	= 1	
myArray = <u>new int[10];</u>	มีรันไทม์เอนซิเมชัน เนื่องจาก myArray ไม่ได้ตั้งค่ามา	ไม่สามารถเข้าถึง index ที่ไม่ได้ตั้งค่ามา
System.out.println(<u>myArray[0]</u>);		

static array តើ fixed size និង size ពេលចែករាយ
"ArrayIndexOutOfBoundsException"

* Dynamic array ก็ array ที่ always accept new data โดยมีขนาดใหญ่ capacity เมื่อ size is full.
if size is full 1. create อันใหม่แบบ bigger size และ copy ข้อมูลมาลง
* หมายเหตุ size++;

- ▢ Get(i) = return នៅលើ i
 - ▢ Set(i,value) = set value នៅ i
 - ▢ PushBack(value) = ដើរ value នៅបន្ទាន់ស្តីក
 - ▢ Remove(i) = លួចចាប់នៅ i

field = - arr = pointer ទៅ array
- capacity = ពន្លេដែល array អាចរាយទៅបាន ក្នុងក្រុមហ៊ុន
- size = array តម្លៃរបស់វា

|| Linked List ||

- ปน. ของ array : ต้องจดงพท. ที่ memory
• ก้าพท. เต็ม = บัน ArrayIndexOutOfBoundsException
• การ allocate Dynamic array = $O(n)$
แต่ $O(1)$ เริ่บกการวิเคราะห์แบบ Amortization (ด้วยการวิเคราะห์ใน การ add
เป็น $O(1)$ เพราะมัน copy ข้อมูลลงเรื่อยๆ ดังเดิม คือครั้งลุก ก็ยัง เป็นการวิเคราะห์แบบ ณ ลุก)
- △ **Linked list** add ข้อมูลเป็น $O(1)$ เนื่องจาก
สปก. มี key, head, next

→ Singly Linked list ←

- มี head เป็น pointer ที่ชี้ไปที่ตัว node แรก
- ถ้า Linked list is empty , head จะชี้ไปที่ null
- key จะเก็บดำเนิน node
- next จะชี้ไปที่ node ต่อไปนี้ หรือ null (final)

PushFront : ต้องการ add new key : $O(1)$

- สร้าง node ที่จะเพิ่ม
- เอา next ของ node ที่สร้างมาชี้ไป head.next
- เอา head ชี้ node ใหม่

PopFront : ต้องการ remove ตัว Front : $O(1)$

- $head = head.next$ (ทำแล้วเดาแล้ว java 0: ลบเอง)

PushBack : ต้องการ insert ที่สุดท้าย : $O(n)$

- สร้าง pointer : current ไปชี้ head ก่อน
- while (current != null) { current = current.next; } จนชี้ไป node ที่ต้องการเพิ่ม
- current.next = node ใหม่ ก็ต้องการเพิ่ม

PopBack : ต้องการ remove ที่สุดท้าย : $O(n)$

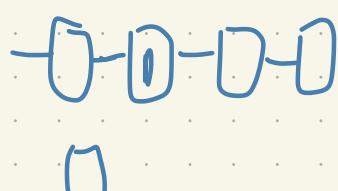
- สร้าง current = head และวนไปกับตัวก่อนหน้า : ลบ บันทึก รวมสุดท้าย
 $while (current.next != null) { current = current.next; }$
- current.next = null เลย

AddAfter (Node, Key) : เวลา key add หลัง node : $O(1)$ หรือ $O(n)$ ถ้าไม่รู้ node add อยู่ไหน

- สร้าง node ใหม่ ก่อน (แล้ว) สร้าง current วนครูไปปะบุถูก node ที่ต้องการ add after
- ให้ node ใหม่. next ชี้ไป current.next (แล้วให้ node ใหม่. next ชี้ไป key/node ใหม่)

Add Before (Node, Key) . เวลา key add ก่อน node : $O(n)$ bc ว่า while วิ่งรอบ

- วนไป current.next = Node

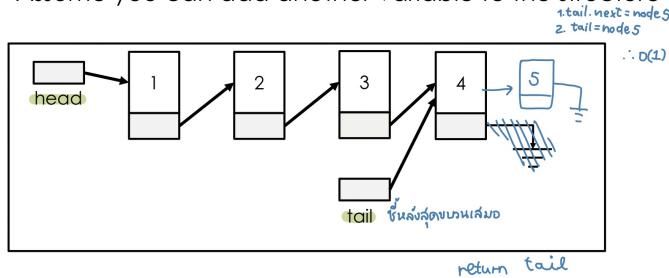


Singly-linked list operations	Running time
O(1) ตั้งหัวจ้า O(1) ตีบหัวจ้า	PushFront (Key)
	TopFront()
	PopFront()
	PushBack(Key)
	TopBack()
	PopBack()
	Find(Key)
	Erase(Key)
	Empty()
AddBefore(Node, Key)	O(1) return head==null ;
AddAfter(Node, Key)	O(n) หาปัจจุบันก่อน

ឧបាទក្នុង Push Back (key), TopBack() មែនស្ថិតុយនាក់ $O(n)$ ពីរ $O(1)$

The current complexity is $O(n)$

Assume you can add another variable to the structure



پر. ویں Singly-Linked list

- One-way traveling
 - ไปบ้านคน only
 - Add After = Superfast
 - Add Before = ចំណាំការណើនីតារសម្រាប់បាន (សំណើនីតារ head)

→ Doubly-Linked List with tail ←



Node contains

- key
 - next pointer
 - prev pointer

PopBack() : ลบโนนตสุดท้าย

Time complexity = $O(n)$

1. `tail = tail.previous`

2. `tail.next = null`

Singly-linked list operations	instead	No tail	With tail	binary search $\Rightarrow O(\log n)$
<code>PushFront (Key)</code>		$O(1)$		
<code>TopFront()</code>		$O(1)$		
<code>PopFront()</code>		$O(1)$		
<code>PushBack(Key)</code>		$O(n)$	$O(1)$	Doubly
<code>TopBack()</code>		$O(n)$	$O(1)$	
<code>PopBack()</code>		$O(n)$	$O(n)$	$O(1)$
<code>Find(Key)</code>		$O(n)$		
<code>Erase(Key)</code>		$O(n)$		ไม่ใช่แล้ว!
<code>Empty()</code>		$O(1)$		
<code>AddBefore(Node, Key)</code>		$O(n)$		$O(1)$
<code>AddAfter(Node, Key)</code>		$O(1)$		

Summary

- $O(1)$ ตอนแรก/ลบ จากตัวน้ำ
- ดับ tail ของ Doubly linked-list, $O(1)$ ตอนแรก - ลบจากตัวหลัง
- $O(n)$ ตอนหา element วงจร while loop
- list ไม่ต้องมีพ.ติดต่อกัน
- Doubly-linked list, $O(1)$ ตอนแรก: หัว node หรือลบโนนต

// Stack // Array + linked-list

Array VS Linked list

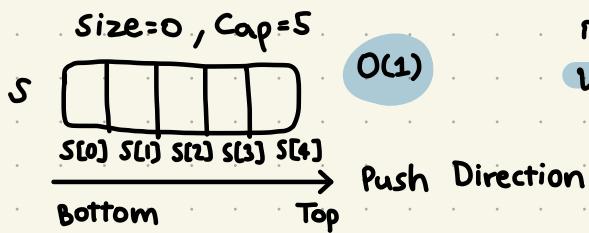
- Linear data structures
- Key Variable
 - Array : size, capacity, arr
 - Linked-list : head
- $O(n)$ ตอน key
 - for loop for array Condition: for (int i=0; i<size; i++)
 - while loop for linked-list while (node.next != null) { node.moveNext(); }
- $O(1)$ ตอน access ตั้ง 1st กับ last
 - Array 1st $\rightarrow a[0]$, last $\rightarrow a[size-1]$
 - linked-list 1st $\rightarrow head$, last $\rightarrow tail$
- "Array" arr=new int[10] // ขยายนัดต่อไปได้เรียบร้อยของพท. เยอะๆ ก่อน
 - "linked list" - สร้าง obj และจดพท. เวลาเดีบกัน + รักษา address ตัวต่อไป
 - wh. ไม่ติดกัน, ข้อมูลกราฟ, ตัว 1 นำไป ข้อมูลหาย

Midi
Sem

Stack = LIFO "Last in, first out" บนล์สุด, ออกก่อน

- push: เอา element บนล์สุด
- pop: ลบและ return element บนล์สุด
- IsEmpty: ทดสอบว่า ว่างไหม

ถ้าไม่มีว่าไว้ใน `pop` เลยว่าต่อไปจะ `pop` = Stack Underflow Exception
ถ้า stack เต็ม = Stack Overflow Exception



กันต้องอยู่ $S[0]$ เสมอ
ห้ามสวับป้ายบังหนูเป็น $O(n)$

mid-sem

Exception

ถ้า array เต็ม เราก็ 5 options :

1. ฟื้น size ของ array
2. Throw an exception (StackOverflow Exception)
3. Ignore new element ไม่เก็บ
4. Replace ตัว current top ของ stack
5. ตั้งแต่ sleep ก่อน เลี้ยวอนน์ `pop` ออกจาก queue และคุยกับผู้มีอำนาจตัดสินใจ

Array: ทั้ง `push(value)`, `Pop()` = $O(1)$

Linked-list : push-pop first = $O(1)$

push direction
top ← → Bottom

Stack Applications

Balance checking : พวกร์ควงเล็บ

1. for loop ต้องแต่ตัวเรียกบันลอดท้าย โดยสร้าง current เป็น
2. เช็คว่า current (ตัวเรียก) เป็นเปิดหรือปิด

open: push stack → เลือกเลื่อน current check ต่อ

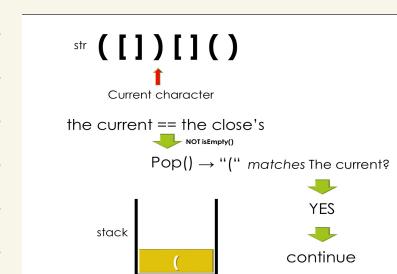
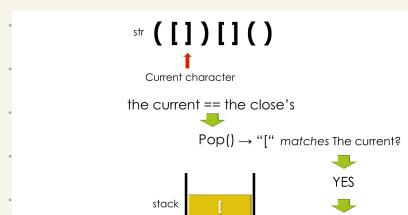
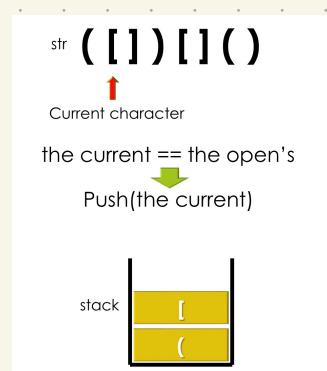
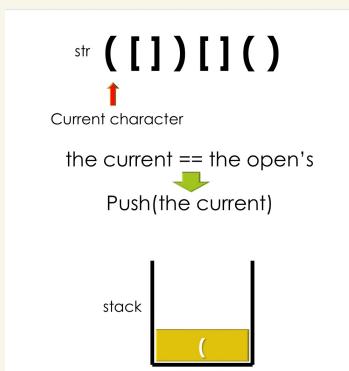
close: เช็ค empty [empty : false

not empty, ให้ pop หาซึ่งค่าว่า matches กับ current ไหม

yes: continue

no: return false

3. ถ้า return true ถ้า stack is empty มีเห็นนี่ false (ถ้ามีปิด) ปิด



Undo in Word using a stack

- "Undo" can reverse the state to previous states.
- "Redo" reverses the "Undo"
- Undo/Redo in Word Processing: implementation using one stack:
 - When typing new words (ending with white space), the stack is cleared.
 - When undoing, the current word is pushed into the stack and then erased.
 - When redoing, the stack is popped, and the top word is added to the paragraph.
- Example:
 - "I" "love" "cats" [UNDO] [UNDO] [UNDO] [REDO] "like" "dogs" [UNDO] [UNDO] [REDO] "cats" "and" "dogs" [UNDO] [UNDO] "very" "much" [UNDO] [UNDO] [REDO] [REDO]

UNDO - push in stack

REDO - pop from stack (ยกเลิก UNDO)

ถ้าพิมค์ใหม่ stack จะถูก clear ทั้งหมด stack

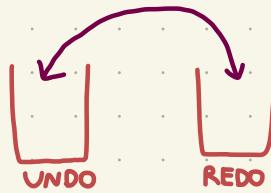
Undo Redo using stacks

- Undo คือ การกลับไป state ก่อนหน้า

- Redo คือ การย้อนกลับของ Undo

การกด undo 1 ครั้ง จะถูกลบใน stack

ex. "I" "love" "cat" [Undo][Undo] [Undo] STACK UNDO: cat → love → I
 [Redo] STACK UNDO: cat → love STACK REDO: I
 "like" "dogs" [UNDO] STACK UNDO: cat → love → dogs STACK REDO: I
 "cat" "and" "dogs" "very" "much"



Parsing XHTML

โปรเซสซิ่ง: มีตัว open & close

ex. <html> (ตัวเปิด), </html> (ตัวปิด)

- matching closing tags, e.g., </some_identifier>

```
<html>
  open   close   open   close   close
  <head><title>Hello</title></head>
  open
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

ภาษา html

ผู้ใช้ต้องปิดตัวก่อนตามลำดับ.

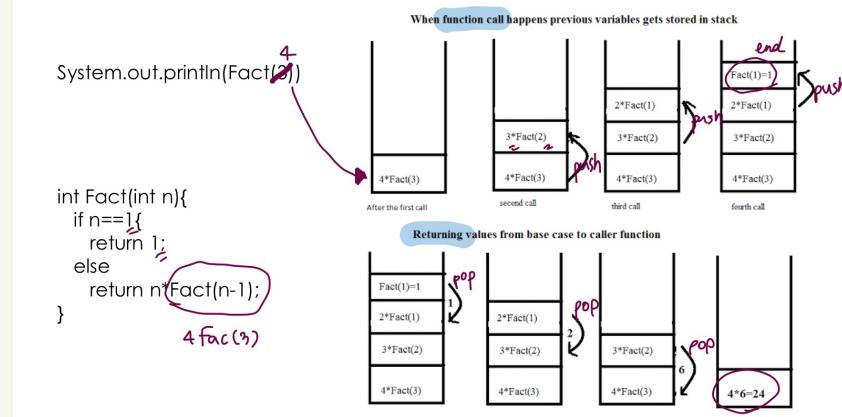
มีตัว stack ตัวมันจะไปหัว tag

1. เจอตัว open → push ลง stack

2. พอดีเจอตัว close มันจะ pop stack → check match [NO = false]
 Yes = continue

Function call : func ชื่อ func ปักไว้วางก่อน = stack

Function calls = STACK!



Reverse - Polish Notation

in-fix (จำเป็นต้องมีวงเล็บ)

- ตัวเลขอยู่ริมขวา operator
(ตัวเลข = operands)

Postfix (ไม่จำเป็นต้องมีวงลับ)

3 4 + 5 × 6 -

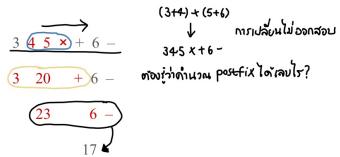
1. $(3+4) 5 \times 6 -$

2. $(7 \times 5) 6 -$

3. $35 - 6 = 29$

Postfix (Reverse-Polish) Notation

Other examples:



การเปลี่ยนไปดังนี้

3 4 + 5 6 + *

ลองเขียนตัวเอง postfix ໄດ້ຄົບໄວ?

3 4 5 6 × +

3 4 - 1 × +

3 - 4 +

-1

Benefits • ไม่ก่อความ + ไม่ต้องใส่ร่วงเลข

- วิธี 1. เจอตัวเลข → โนลต์ตัวเลขไปปุ่น stack
2. เต่าถ้าเจอ operator มันจะ pop stack 2 หีบะ-ปาก-ลบ-คูณ-หาร กันโดย 1st pop → ตัวเดียวหลังสุด ทางบน ก็ push ตัวเลขเป้าไปปุ่น stack

Summary

- ▢ Stack สร้างโดย array - linked list
- ▢ O(1): Push, Pop, Top, Empty
- ▢ Stack = LIFO queues
- ▢ ป.y. balancing symbols, parsing, func. call, reverse Polish postfix

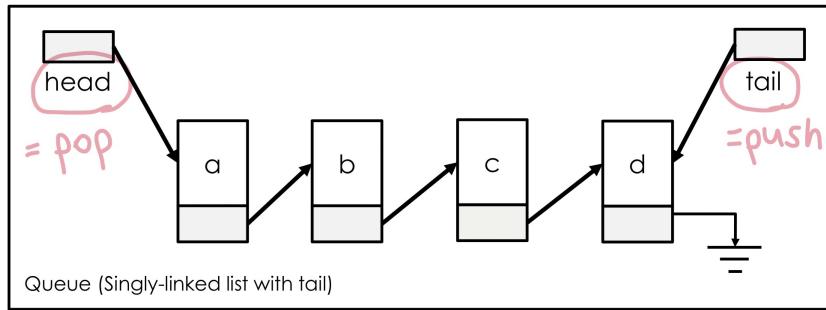
// Queue //

- One-ended list → stack
- Two-ended list → Queue
ที่เนื้องอกนก គົ່ວເປັນເພີ້ງເສັ້ນ (Linear)
- enqueue ຕົວສົດທ້າຍ ແລ້ວ Dequeue ຕົວແຮກ
- FIFO ('ໄວໄວ') First in, First out
- ensures "fairness"

- operation** • Enqueue (Key), Push (Key) ໄປຕົວກໍາຍ
• Dequeue(), Key Pop() ລົບແລ້ວ return ຕົວແຮກສຸດ
• Boolean IsEmpty() ເຊື້ອ empty

ประโยชน์ File servers, Printer Queue, Phone calls
(similar priority) ໂຕ Priority Queue ສັງໄດ້ສຳຄັນກໍາໄປກ່ວນໄດ້ກ່ວນໄດ້ລະຍ

Queue with Linked List



fn Enqueue(a.) = pushBack ຕົວກັບ

1. ສ່າງ Node a
2. ເອ head = Node a
tail = Node a ແລ້ວ Node a.next

fn Enqueue(b.) ເພີ່ມ

1. ສ່າງ Node b
2. ເວa Node a.next = Node b
tail ++;
3. check isEmpty(); ແລ້ວ return head == null;

fn Dequeue() ເກ a ວິກ

1. ເວ head.next = node b

fn Dequeue() ເກ b ວິກ

1. ເວ head = null;
tail = null;

Queue Operations	List Operations	Complexity
Enqueue(Key)	PushBack(Key)	O(1)
Key Dequeue()	Key TopFront(); PopFront();	O(1)
isEmpty() <i>head=null</i>	isEmpty()	O(1)

Thought Questions?

- ❑ What if we change the front of the Queue to be the end of the list (still singly-linked list with tail)?
mid -> m
- ❑ What would be the complexity for each operation?
 - ❑ Enqueue
 - ❑ Dequeue
 - ❑ isEmpty()
- ❑ How about Singly linked list without tail? (2 cases)
- ❑ How about Doubly linked list with/without tail? (4 cases)

1. ก้าวไปนั่งตัวๆ แรกของ Queue เป็นตัวสุดท้ายของ list
(ยังเป็น singly-linked list with tail)

and Enqueue O(1)

Dequeue O(n)

isEmpty() \rightarrow O(1)

2. without tail Enqueue O(1)

Dequeue O(n)

IsEmpty O(1)

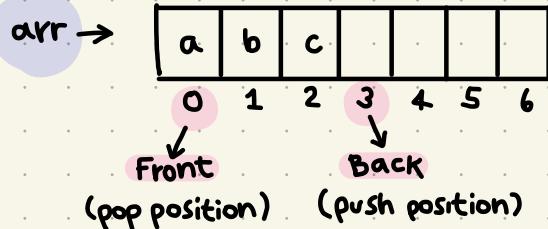
3. Doubly linked list with/without tail?

Enqueue O(1)

Dequeue O(1)

IsEmpty O(1)

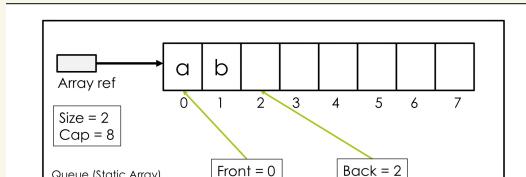
Queue Implementation with Array



Size = 3
Cap = 7
- Array Queue -
variable
arr, size, cap
front, back

- Enqueue(a) \rightarrow arr[back] = a;
back++;
size++;

(จะ front, back ซึ่ง nn. 0 ที่เดียวกัน , back=1 ถ้า front=0 เมื่อตอนเดิมแล้ว size ก็จะ +1)

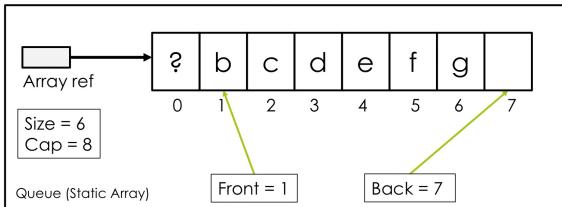


แบบนี้ไปเรื่อยๆ

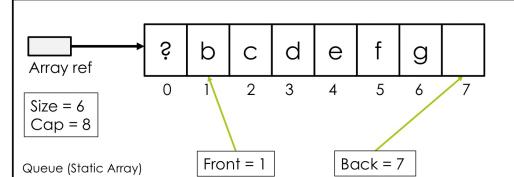
Enqueue(b)

-**Dequeue()** → $\text{temp} = \text{arr}[\text{front}]$;
 $\text{Front}++$;
 $\text{size}--$;
 return temp ;

(สร้าง pointer **temp** ไปที่เดียวกับ **front** ก่อน เล้า **front+1** เพื่อย้ายตัวหน้า เล็ก ลบ **size** ลง 1 ตัว และ **return** ตัวที่ลับไปหนึ่งตัว ตัวที่ **temp** ชี้ไป)

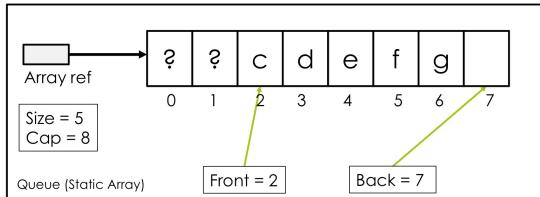


Dequeue() → a

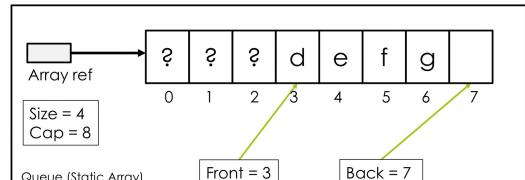


Dequeue()

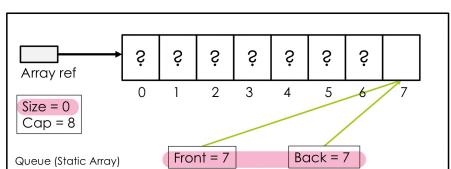
What are the corresponding Array operations?



Dequeue() → b



Dequeue() → c



isEmpty() → true
 $\hookrightarrow \text{size} == 0 ?$

- ถ้า **Dequeue()** ถูก → **ERROR**

∴ ควรใส่ condition นี้ด้วย
 $\text{if } (\text{size} > 0) {$
 $\quad // \text{statement}$
 $\} \text{ else} {$
 $\quad \text{sout} (" \text{ERROR} ");$
 $}$

Full Queue?

Queue of Array = Circular Array

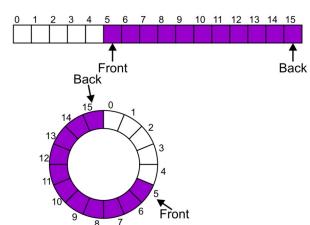
++iback

$\text{if } (\text{iback} == \text{capacity}()) {$
 $\quad \text{iback} = 0;$
 $}$

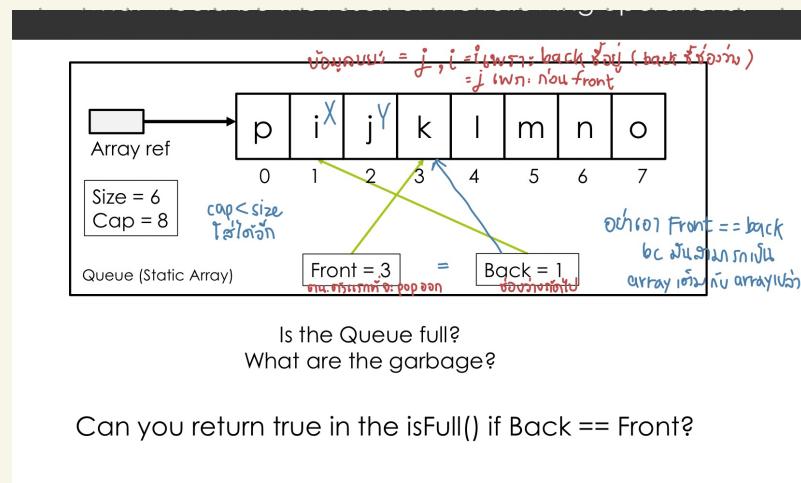
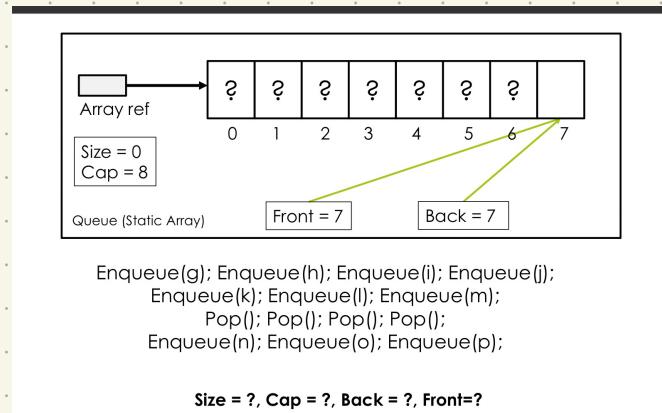
indices being cyclic.

..., 15, 0, 1, ..., 15, 0, 1, ..., 15, 0, 1, ...

This is referred to as a *circular array*



Ex.



* Mid-sem Exceptions

ก้า Array เต็ม มี 5 ตัวเลือก

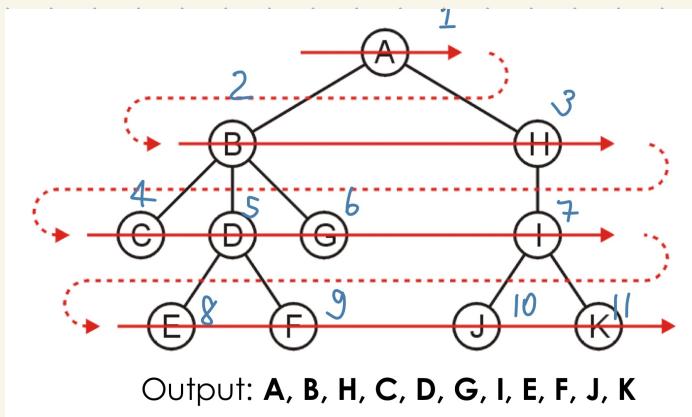
1. เพิ่ม size ของ Array
 2. Throw an exception
 3. Ignore element ที่ push เข้ามา
 4. ให้ process "sleep" จนตัวแรกของคิวโดน pop ออก
 5. replace ตัวนั้นด้วย
- ใช้ member function `bool full()`

คือ ก้าเลือก เพิ่ม size ของ Array เมื่อ ค่อนข้าง complex

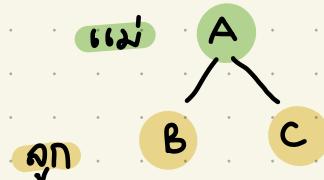
เพราะต้อง for loop, copy ข้อมูลนั้น เท่copy ตรงๆ ไม่ได้ เลยต้อง copy แบบ
เอาตัวแรกเป็น front และตัวสุดท้ายคือ back = การทำ **Normalization**

Normalization = เอา front ไปไว้ arr[0] และ back = arr[size-1]

Breadth-first : ลับหน้าในระนาบเดียวกัน ก่อนไปลังๆ



- ควรอ่านจากซ้ายไปขวา
- ใช้ Q ในการเดินเส้นทางนี้
- Algo มี 4 บรรทัด *JUM



Solution : 1. push root directory ลงไปใน Q
2. ถ้า Q ไม่ว่างให้ while

- 2.1. Pop Q 拿出根目 (pop ตัวแม่根目มาดื้อบนสุด)
- 2.2. Push ลูกของ directory แห่งไปใน Q และวนกลับไปกามว่า Q ว่างไหม

ดูอยู่ใน Slide ประจำวัน

Summary

- Q เป็น abstract ที่ common ที่สุดใน data structures
- เพราะว่า การทำงานของ queue มี trivial (ง่าย)
- แต่การ implementation (ใน coding) จะยากกว่า stack

Application

เข้าคิวลูกตัว , Breadth-first traversals of trees

// Tree //

□ List, Stack, Queues ต่อ linear relationships

• Accessing .. super fast $O(1)$

• Searching $O(n)$

↳ Binary search $O(\log n)$ แต่ condition: sorted

Total sorted = $O(n^2)$ อยู่ดี

$\therefore \text{Tree} = \text{answer}$

Tree (?)

- เป็น node และมีบันทุณล์ผ่องผู้
- บางไฟน์ node ก็ได้ = empty tree
- ต้องมีส่วนช่วง 1 node ไปมากกว่า 1 node ก็ได้ (diff from linked-list)
- node ที่ไม่ได้เป็นหัวโครง เรียกว่า null
- node ที่ไม่มีโครงไปหนาน = $\underset{= \text{head}}{\text{root node}}$ (ชี้ตัวเดียว)
- ต้องไม่มี loop / cycle

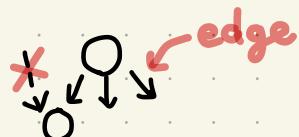
- บันทุณล์ hierarchical relationships (มีลำดับชั้น)
 - ex. Family, File directories (Folders), เกม XO, ลำดับชั้นโนองค์กร

Tree Terminology

- คล้าย linked list

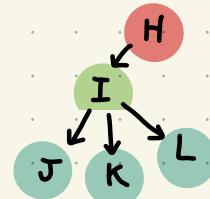
1. มี node แรกสุด = root
2. ในแต่ละ node ไปได้น้อยกว่า
3. แต่ละ node มีคนซึ่หานน์ได้แต่คนเดียวเท่านั้น ยกเว้น root ที่จะไม่มีโครง
4. เส้นทาง node = edge

โดย edge ต้องให้ทิศทางเดียว (Directed edge) ก้าวไปข้างหน้าเป็น root



Parent and Child

- ทุก node มีคูกันคนหรือคลายคนได้ เช่น child nodes or children
- ทุกๆ node จะมี parent 1 node เท่านั้น



Node Degree

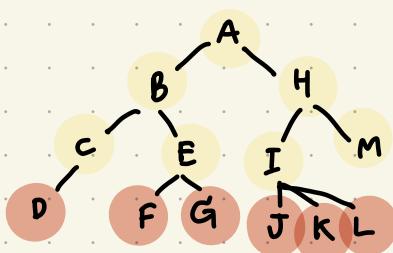
- degree of node = ลูกของมนูนี่ก็ล้วน = size
 - parent เดี๋ยวกันเรียกว่า siblings
- ex. $\text{deg}(I) = 3$, J, K and L are siblings

Phylogenetic tree

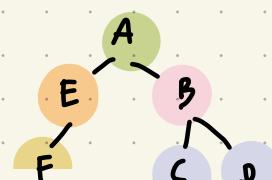
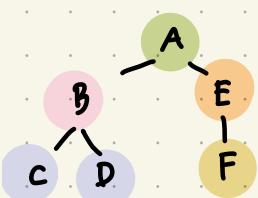
- ทุก node von Phylogenetic tree มี degree 2 หรือ 0 เท่านั้น
- ex. Carnivora morpha (สัตว์กินเนื้อ)

Leaf Nodes and Internal Nodes

- node ที่มี degree = 0 คือ leaf nodes
- node อื่นๆ คือ internal nodes



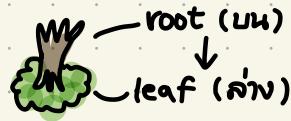
Unordered vs Ordered Trees



- ถ้า unordered trees = หนึ่งเดียว
(บรรทัดข้อมูลเดี่ยวกัน: ล้ำเดินไม่สำคัญ)
- ordered trees = สอง
bc ล้ำเดินต่าง

Root node

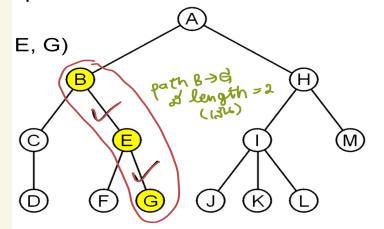
- Root ໃນ node ចົງໄດ້ສັນອາ



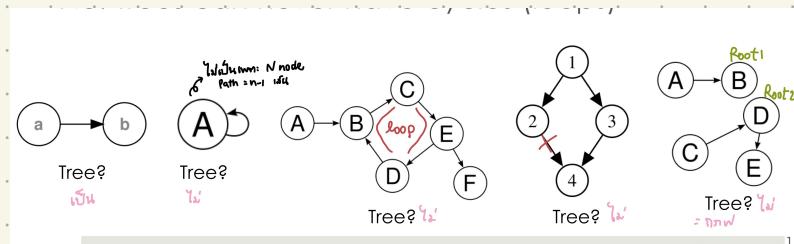
Path

- ຕົວເລີນທຸກຈາກ node ພຶບໄປຢືນກ node ນີ້
- length ຕົວ ດຣ. edge ໃນ path ນີ້
- tree N nodes has $N-1$ edges

Path of length

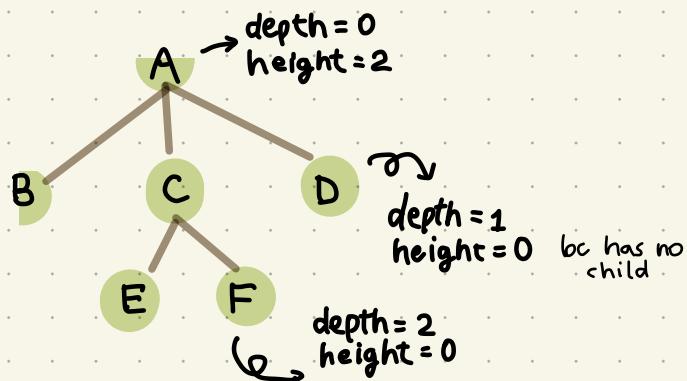


- Tree ແມ່ນ loop ໄວໄດ້ ໃນ 2 node ທັງນີ້ path ມາກສູດຕົວ 1 ເທົ່ານີ້



Distance Measurement

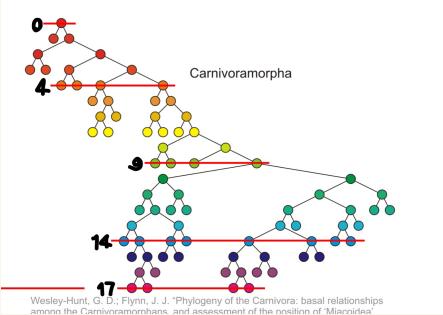
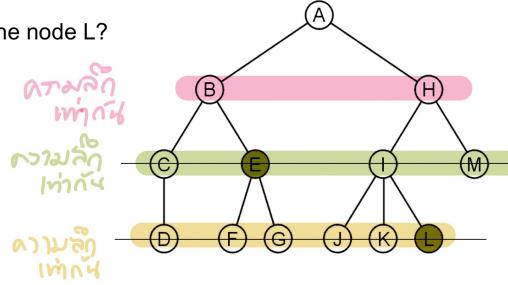
- Length of path = ດຣ. edges
- Depth of node N = ດາວວິກຂອງ node N (ວັດ root \rightarrow Node N)
- Height of node N = ດາວວິສູງຂອງ node N (ວັດ Node $N \rightarrow$ leaf (ຄູກໂນໂລຍົມ))
- Depth of tree = ວັດດາວວິກຂອງ Node ທີ່ລັກສູດ
- Height of tree = ວັດດາວວິສູງຂອງ root equal



Depth of tree = Height of tree

Node Depth

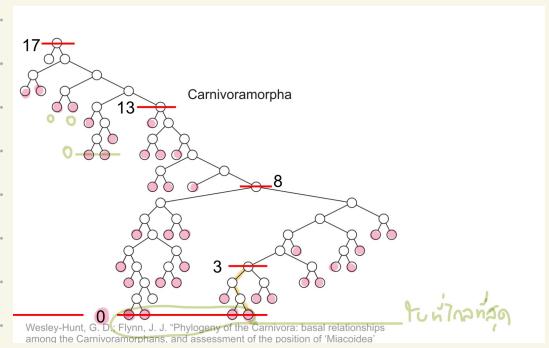
- What is depth of the node E?
 - 2
- What is depth of the node L?
 - 3



Height of Node and Tree

(ความสูงของ node)

- ก. สูง node = วัดจาก root ไป node ที่สุด
- ก. สูงต้นไม้ = จาก root ไป leaf ไกลสุด
- ก. สูงต้นไม้ที่ node เดียว = 0
- ก. สูงของ leaf node = 0 เป็น啥
- Height tree = Depth tree**
- ความสูงของ Empty tree = -1



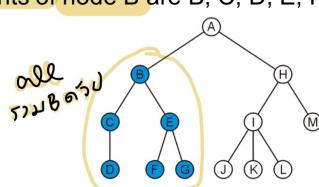
Ancestor and Descendent

จาก Node a → Node b

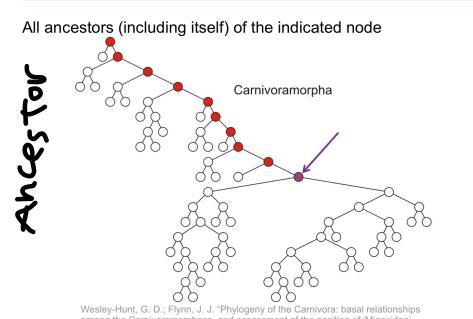
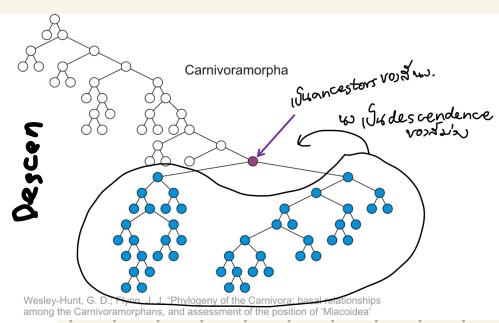
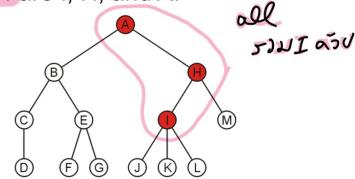
- a เป็น ancestor b
- b เป็น descendant a

* * node ใดๆ เป็น ancestor & decendent ของตัวมันเอง (ก็ไม่ใช่ strict)
strict : ก้ามอันนี้ ancestor & decendent จะไม่แบบตัวมันเอง

The descendants of node B are B, C, D, E, F, and G:



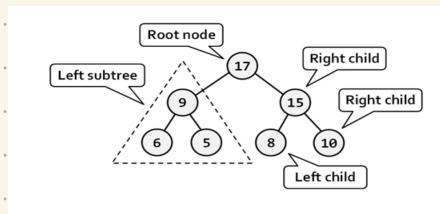
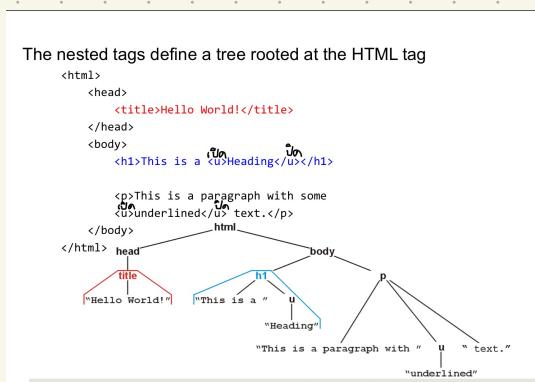
The ancestors of node I are I, H, and A:



Alternative Definition of tree กรณีที่ไม่ใช้ root node : recursive

- ถ้า node ไม่ใช่ root node
- node ที่เป็น root ไม่ต้องมี
- ถ้า node ไม่เป็น root node จะมี subtree (sub-tree)

Ex. XHTML



VER!

Implementation of Trees (K-ary Tree) pointer-based

1) สร้าง class Node

มี properties Object Key และ Node child₁, child₂, ..., child_K

optional กรณีเป็น双向 (bidirectional) เพิ่ม Node parent; (ไม่จำเป็นก็ได้)

2) สร้าง class Tree {

```

Node root
}
```

3)

```

T.root = new Node [2]
T.root.child1 = new 7
T.root.child2 = new 5
T.root.child1.child1 = new 2
```

Implementation of Binary Trees

1) class Node ทำ

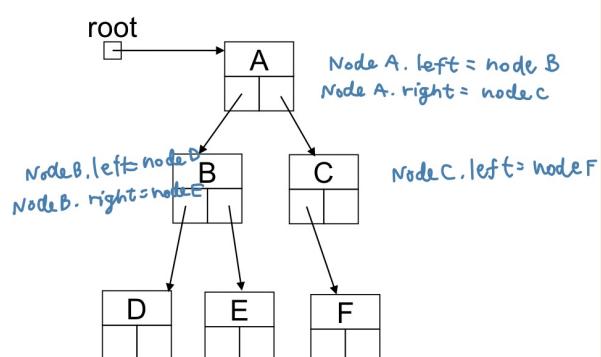
Object key

Node left

Node right

Node parent // optional

2.



class Node

Key

LeftChild		RightChild
-----------	--	------------

VER2

Use List of Node instead fixed num

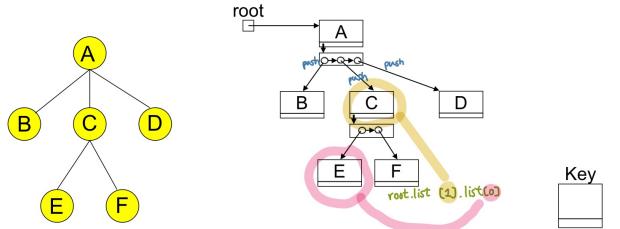
liked-list based

push မျှော်သုတ

- 1 สร้าง class Node (แล้วมีตัวแปร Object key, NodeList listofchildren)
2. class Tree { Node root }

List-based Implementation

LIST-based Implementation



- Now that each node can have a flexible number of children
- Slightly confusing for implementing
 - Need loop for accessing the children
 - Tree traversal/search implementation might be complicated

- မျိန်ဝါယာနံပါးယုံကြည်
- Accessing ရှုချင်ရန် အပြည့်စုံပြည့်စုံလုပ်
- Tree traversal ဆုံးလုပ်

□ sibling ကျော်စိုက်

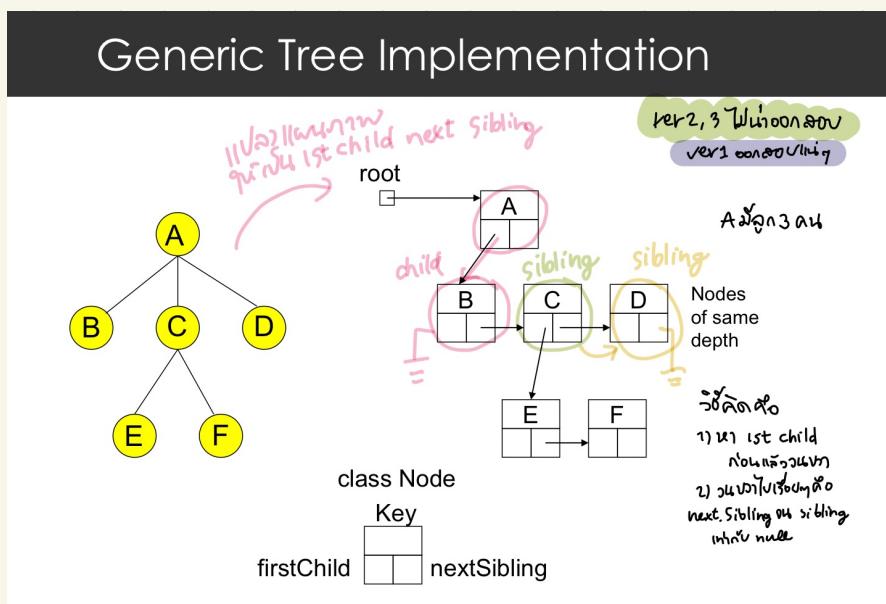
- ver2 : node ရှုတဲ့ children နှင့် parent (တာမျိုး parent)
- မျိန်ဝါယာနံပါးယုံကြည်
- Solution : ပေးသွေးသူ့ sibling pointer ပါ၏ node အားလုံး cycle ဖြစ်ခဲ့သော်လည်းကောင်း၊ အောက်လုပ်ရန် data structure မျိုး
- ver3 : ဂုဏ်ပိုင်

VER3 Generic Tree Implementation pointer-based

1. สร้าง class Node ပါးယုံကြည် Object key, Node firstchild , Node nextsibling
[Optional: parent]
2. tu class Tree { Node root }

class Node မျိုးမျိုးရှာ
first child ရှာရန်

next sibling ရှာရန်



Binary Tree

class node {
Object key;
Node left;
Node right;
Node parent;}

class Node

Parent

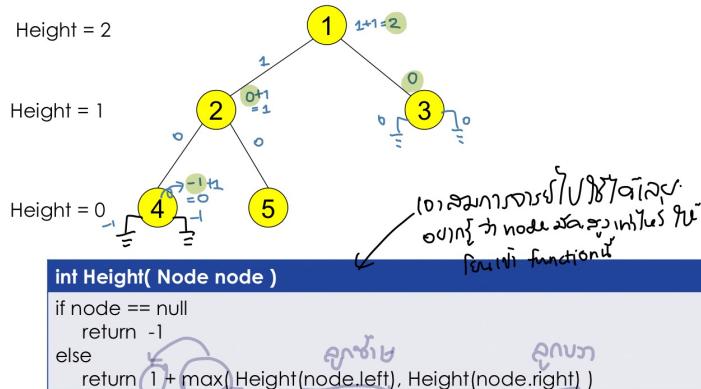
Key

Left Right

- ก้าใน binary tree ด้วย ความลึก d , Node ที่ปั๊ปด. $2^d - (2^{d+1} - 1)$
 - ก้าใน Node N ของ ความลึกน้อยสุดที่ปั๊ปด. $\log_2(N+1) - 1$
 - ความลึกสุดของ binary tree คือ Degenerate case : Tree เป็น linked-list
(worst case : มาก, ต่อๆ กัน)
 - complexity of searching for a key
 1. max depth (degenerate tree) = $O(n)$
 2. min. depth (complete tree) = $O(\log n)$

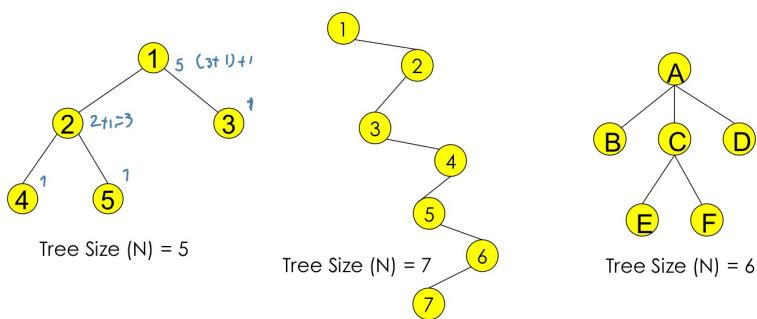
Height Measurement Algorithm

10



Tree Size Measurement Algorithm

20



```
int Size( Node node )
```

```
if node == null  
    return 0  
else
```

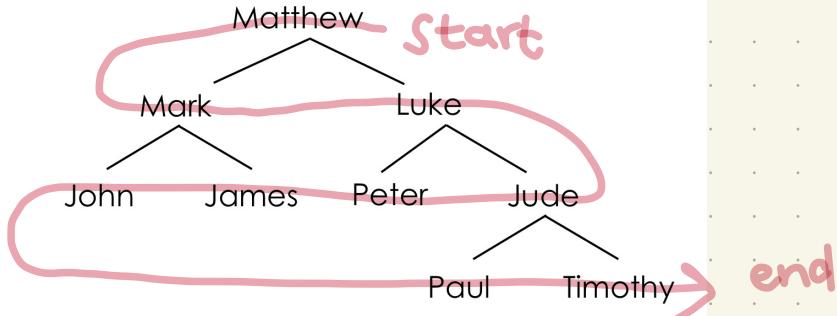
Yabot, take max.

Tree Traversal (Walking Tree)

แบบเป็น Breadth-first (บันทึกชั้น, ซ้ายไปขวา, ผ่านชั้นก่อน)
Depth-first (ไปลึกถัดลงก่อนกลับมาสายตาม, ซึ่งลึก)

- Breadth-first Traversal (Level Traversal) **BFT** using Queue

Breadth-first Traversal (Level Traversal)



Output: Matthew, Mark, Luke, John, James, Peter, Jude, Paul, Timothy

- เทคนิค BFT ไปริ้ว กับ Queue

Penso esame

```
using Queue
LevelTraversal ( Node node )
if node == null
    return
else
    Queue q
    q.Enqueue( node )
    while ( not q.Empty() )
        node = q.Dequeue()
        Print( node.key ) // Do something to the node
        if node.left != null
            q.Enqueue( node.left )
        if node.right != null
            q.Enqueue( node.right )
```

1. เช็คก่อนว่า node ว่างไหม
 2. Node ว่าง return
 3. Node ไม่ว่าง สร้าง Queue ของ併ๆ enqueue (node) แรก
 4. วนลูปต่อ ถ้า queue ไม่ empty()
 5. ให้ค่า ให้ dequeue() ออกจาก併ๆ print node.key
 6. if node.left != null ให้ enqueue node.左子
 6. if node.right != null ให้ enqueue node.右子
- * ซึ่งทำก่อนหน้า *

Depth-first Traversal

ใช้ 1. stack

2. Recursion

3. methodes

- PreOrder = ที่ node ใหม่ ต่อ node นั้นเลย
- InOrder = ที่ซ้าย → กลาง → ขวา (ต่อลูกก่อนเริ่ม)
- PostOrder = ไม่นำมา (ซ้าย → ขวา) ก่อนแล้วไปนำ เมื่อ

DFT using Recursion

- ใช้ stack ทำ

Depth-first Traversal

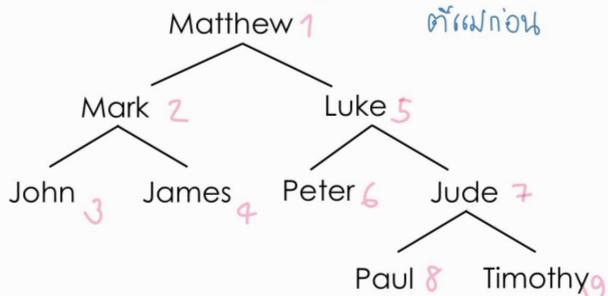
nonRecursiveDFT (Node node)

```
if node == null  
    return  
else  
    Stack s  
    s.push( node )  
    while ( not s.Empty() )  
        node = s.Pop()  
        Print( node.key ) // Do something to the node  
        if node.right != null  
            s.push( node.right )  
        if node.left != null  
            s.push( node.left )
```

เน้นอันเดียวเลย
แท้ไปทางกันซ้าย

- Preorder

Depth-first Traversal (PreOrder)



Output: Matthew, Mark, John, James, Luke, Peter, Jude, Paul, Timothy

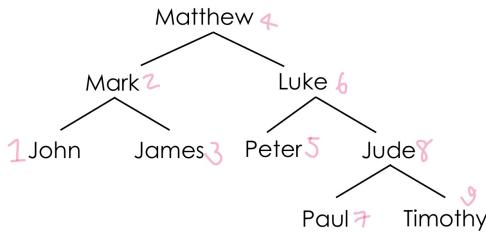
Depth-first Traversal (PreOrder)

PreOrderTraversal (Node node)

```
if node == null  
    return  
else  
    Print( node.key )  
    PreOrderTraversal (node.left)  
    PreOrderTraversal (node.right)
```

-InOrder

Depth-first Traversal (InOrder)



Output: John, Mark, James, Matthew, Peter, Luke, Paul, Jude, Timothy

Depth-first Traversal (InOrder)

InOrderTraversal (Node node)

```
if node == null  
    return  
else  
    InOrderTraversal (node.left)  
    Print (node.key)  
    InOrderTraversal (node.right)
```

start node

recursive

non-recursive

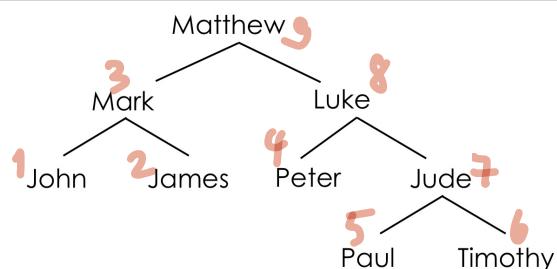
InOrderTraversal (node.left)

Print (node.key)

InOrderTraversal (node.right)

-PostOrder

Depth-first Traversal (PostOrder)



Output: John, James, Mark, Peter, Paul, Timothy, Jude, Luke, Matthew

Depth-first Traversal (PostOrder)

PostOrderTraversal (Node node)

```
if node == null  
    return  
else  
    PostOrderTraversal (node.left)  
    PostOrderTraversal (node.right)  
    Print (node.key)
```

recursive left

recursive right

print now

Unordered Tree Summary

* Exam

1. Tree เป็นโครงสร้างข้อมูลแบบบิโนเดที่ใช้ไปหลายโนนเดตได้
2. Searching a key สำหรับ unordered = $O(n)$
3. Degenerate (Tree เส้นเดียว) จะเป็นแบบ linked list เลยคือ $\Theta(n)$ big theta
4. ก้าวสี่ก้าวต่อไปของการค้นหาอยู่ node ให้ลึก ดึงตัวนั้นๆ, BFS และ
5. ก้าว: นำ: ไว้ลาก, ใช้ DFS
6. Traversal (DFT) ชื่อ Preorder, Inorder, PostOrder
7. Traversal เป็นได้ 2 แบบคือ recursive หรือ non-recursive
8. ก้าว non-recursive นำรากไป BFS ต่อ Q วน: ก้าว DFT ต่อ stack

Binary tree ถ้า 1 node ไม่ได้ 2 node

ถ้า ไม่ มี 1 node ก็ต่อ, 2 node ไม่ต่อ 3 node ไม่ต่อ

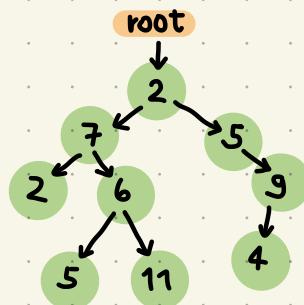
Def: binary tree นิยาม: node ที่มีลูก 2 คน

ลูก: ลูก: ลูก: ลูก: empty เลยก็ต่อ เป็น binary tree ลูกอ่อน
children ที่ left กับ right subtrees

* จงเขียน Binary tree ในรูปแบบ linked list

Implementation is super easy

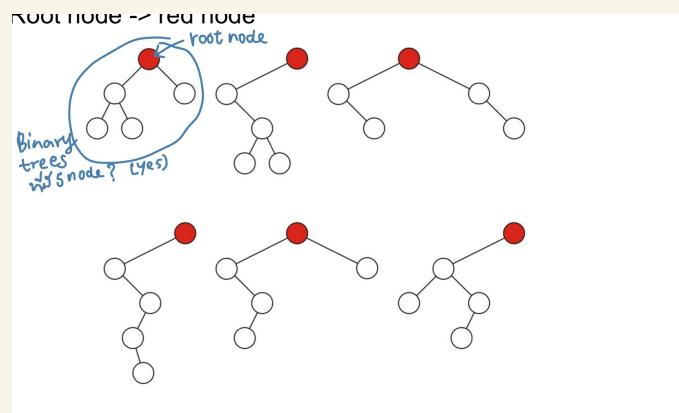
- ให้ class Node ที่ object Key, Node left, Node right
(ใน class Tree ที่ Node root)



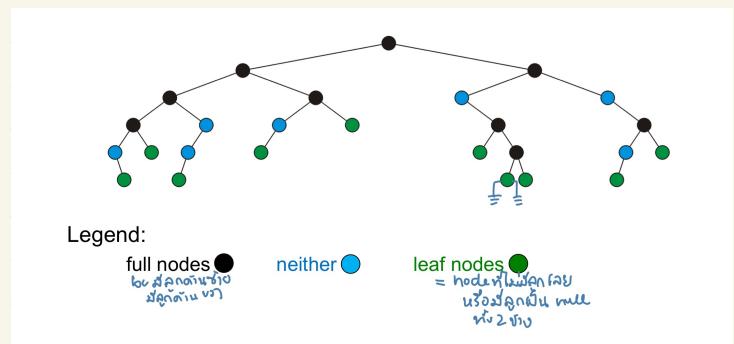
Sub-trees

- **Binary tree** ที่ 2 sub-trees คือ left-hand sub-tree, right-hand sub-tree

Binary Trees with 5 nodes



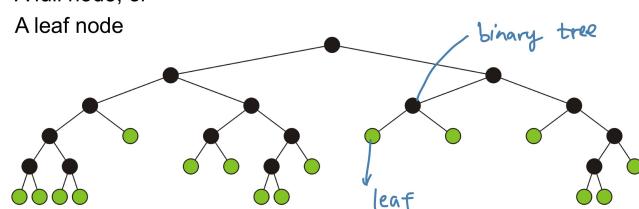
Full nodes : node ที่มีลูกเด็กทั้งสองข้าง - ลูกเด็กทั้งสองข้าง



Empty node : จุดที่จะเอามาไป (node ใหม่) ไม่น้อยไปกว่า

Full binary tree : ตัวต้นไม้แต่ละ node เป็น full node หรือ node ใหม่

- A full node, or
- A leaf node



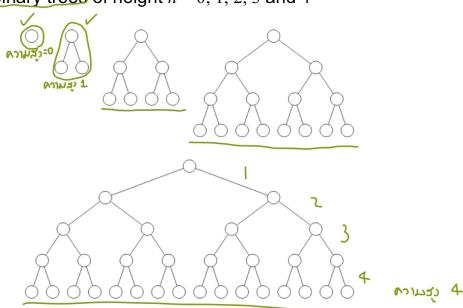
ด้วยมันจะมีแค่
สีเท่านั้นสีเดียว
only

Perfect Binary Tree : Tree ที่สมบูรณ์แบบมากที่สุด

Def :

- ใบตุกไม่ปูที่ตามลักษณะเดียวกันทั้งสิบห้าคู่ความสูง
- node ทุก node ที่เหลือ (ที่ไม่ใช่ leaf node) ต้องเป็น full node

Perfect binary trees of height $h = 0, 1, 2, 3$ and 4

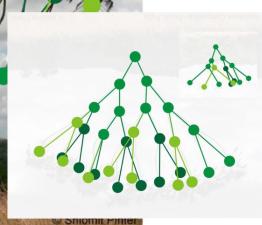
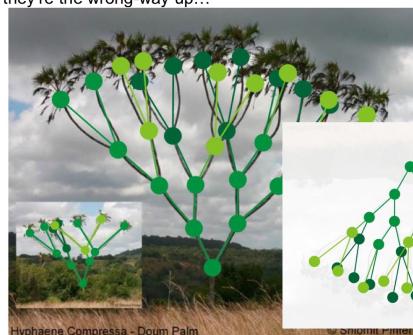


Examples

ใบตุก

Perfect binary trees of height $h = 3$ and $h = 4$

- Note they're the wrong-way up...



คงสูญตัวบ่ง Perfect Binary Trees

- ให้ความสูง n แล้ว node ก็จะ $2^{h+1} - 1$ nodes
- ก็ให้ n node มากแล้วหาความสูง $\log_2(n+1) - 1$
 - perfect tree มีความสูง $\Theta(\ln(n))$ big theta
- perfect tree จะมี 2^h leaf node ที่ความสูงที่ h
- ความลึกโดยเฉลี่ยอยู่ที่ $\Theta(\ln(n))$

Logarithmic Height Proof

กบ. บอกว่า ที่มี n node นั้นความสูง $\log_2(n+1) - 1$

Pf: $n = 2^{h+1} - 1$:

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



- ดึงพสูจน์ว่าการดันนาข้อมูลที่อยู่ที่ PBT ห้องยู่ไป Big theta = $\log_2 n$ หรือ $\ln n$

Pf: ผ่านไปแบบ take limit

$$= \lim_{n \rightarrow \infty} \frac{\log_2(n+1) - 1}{\ln(n)} = \frac{0}{0} \quad \dots \text{ โน่ปีศาจ }$$

$$= \lim_{n \rightarrow \infty} \frac{1}{\frac{(n+1)\ln 2}{n}} = \frac{0}{0} \quad \dots \text{ โน่ปีศาจ }$$

$$= \lim_{n \rightarrow \infty} \frac{n}{(n+1)\ln 2} = \lim_{n \rightarrow \infty} \frac{1}{\ln 2} = \frac{1}{\ln 2}$$

ดัง $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ โน่ปีศาจ $= 0$ เมื่อ $f(n) = O(g(n))$ สรุปได้เลย

$$\left[\begin{array}{l} 0 \\ \infty \end{array} \right] \text{ ex. } \lim_{x \rightarrow \infty} \frac{3x^2}{8x^2} \text{ เป็น Big theta ของกันและกัน}$$



Perfect Binary Tree is not practical

= ทางปฏิบัติแล้วรังยก (ป้อเวี้ย) bc จำกัดชน. node แบบต้องมี node ตามนี้ 1, 3, 7, 15, 31, ... ($n = 2^h - 1$)

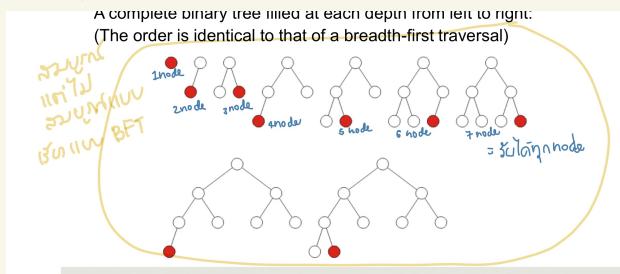
- การ Searching จะเปรียบเทียบความลึก ex. degenerate tree = $O(n)$
binary tree = $O(\log_2 N)$

แต่ PBT จะมี $O(\log_2 N)$ ซึ่งดีไม่เกินนั้น

- nothing is perfect -

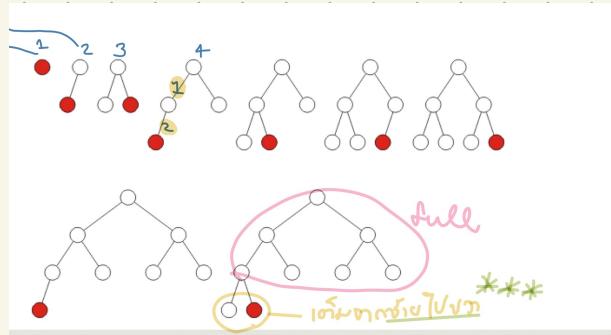
Complete Binary Tree : เป็น Binary tree ที่สมบูรณ์ 100%

- จะเป็นแบบนี้ - ล่าง, ซ้าย - ขวา, เหมือน FBT
- สมบูรณ์ (เต็ม) ในส่วนของ node แบบ



Height of Complete Binary tree

- ความสูงของ n node ก็คือ $\log_2(n)$

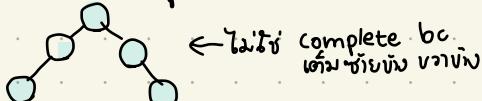


Level vs Depth (level = Depth + 1)

- ต้นเรือน (root tree) Depth = 0 หรือ level = 1

* Review: PBT, CBT, FBT

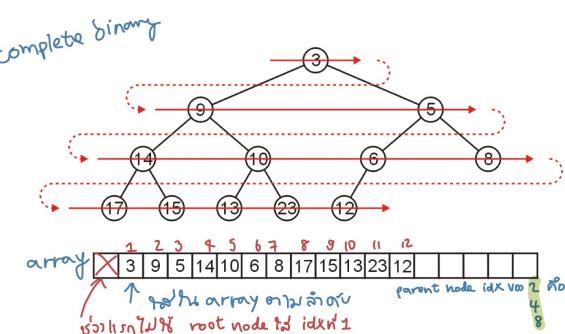
- Full binary tree : node ทุก node เป็น full node หรือ node ในระดับใดๆ ก็ 2 หรือมากกว่า
- Perfect binary tree : ทุก node อยู่ในความลึกเดียวกัน แล้วต้อง full
- Complete binary tree : ทุก level ยกเว้น level ล่างสุด (ไม่ซับ → ขวา) จะเต็มก่อน ต้องเต็มทุกบanch ล่าง , เต็มทุกชั้นไปขวา



Array Implementation for Complete Binary Tree

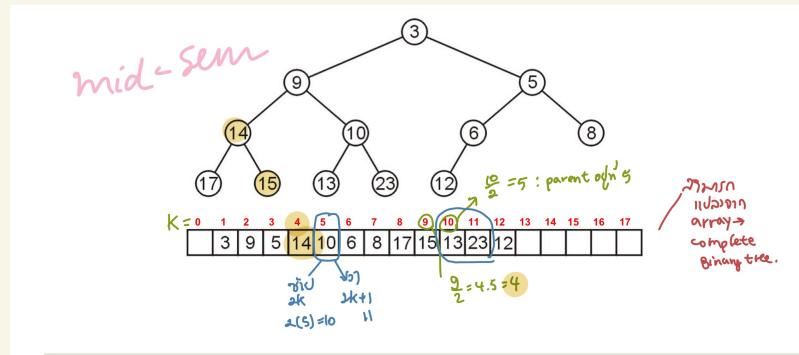
- เราสามารถ array ทุกสิ่งของ complete binary tree ได้ ตามที่ concept BFT บนต้นไม้ เลยว่าผลลัพธ์ไปเรียงลำดับ

We can store this in an array after a quick traversal.



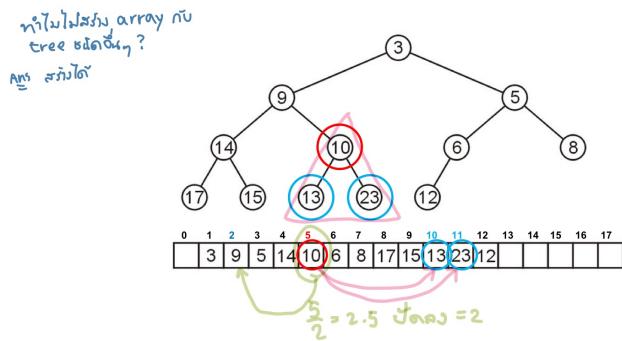
แล้ว ปน. ด้วย การลบ Node
คือ ต้องลบ Node ที่เก่าไป
เก่าไป bc ก็จะลบที่วัน จึงไม่
เป็น complete binary tree

- ถ้าต้องการหาลูกซ้าย at $2k$
หรือ at $2k+1$
parent at $\frac{k}{2}$ เลยปิดลง

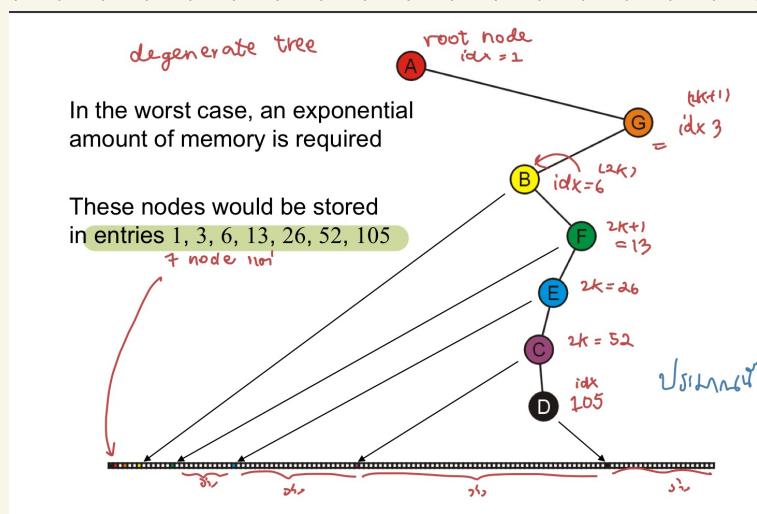


For example, node 10 has index 5:

- Its children 13 and 23 have indices 10 and 11, respectively

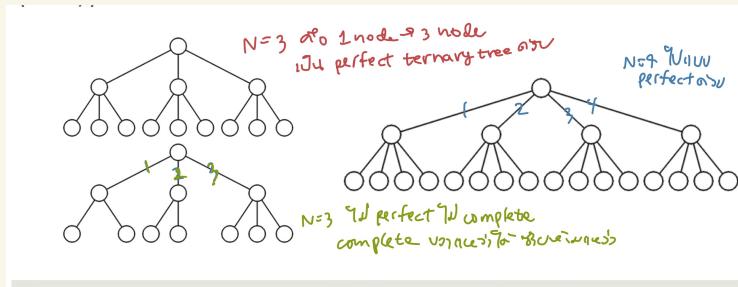


- สร้าง array รูป tree ขนาดเท่าไหร่ ?
ans: ໄດ້ຕັ້ງປລົງເມນອປັນນາກ (ພຣະມັນ): ຕົວແບບ expo ແກ່
ex. degenerate tree ຕ້ອງປລົງເມນສຸດຍ



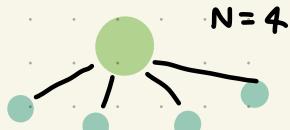
N-ary Trees : 1 node ໄປໄມາກສຸດ *n* node

- Binary \rightarrow ໄປໄມາກສຸດ 2 node , ternary (3-ary), quaternary (4)



Perfect *N*-ary Trees

- ມີຄວາມສຸດທ່ານ *n* node : $1 + N + N^2 + N^3 + \dots + N^h$
ex. $N=4$ ເຕັມ 5 node



- ໃຫ້ *n* node ນາຄາມສູງ : $h = \log_N(n(N-1)+1) - 1$

* ຕອນສອບຕາມບົດສົກໄດ້ເລີຍໄຟຕ້ອງດຳນວາກ

Complete *N*-ary Trees : 1 node ໄປໄມາກສຸດ *N* node



- ໃຫ້ *n* node ແລະ *h* : $h = \lceil \log_N((N-1)n) \rceil$
- Complete binary Tree ເວົາ array ສອບໄວ້
- ໃຫ້ root ໃໝ່ idx 0
- parent node ກົນ idx *K* ດະວິບຸກ $\lceil \frac{K-1}{N} \rceil$ (ເປີດສົງ)
- sibling node *i*:
- $\text{idx } K = \text{idx } K' + j$ \rightarrow $K' = \frac{K-1}{N}$ (ຕົວຢ່າງ $N=3$)
- j ຈຳນວນ *N*-ary

