

ค่ายอบรมโอลิมปิกวิชาการ 2



โครงสร้างข้อมูล: ต้นไม้

Data Structure: Tree

รัชดาพร คณาวงษ์

16 มีนาคม 2562

ศูนย์มหาวิทยาลัยศิลปากร



ทรี (Trie)

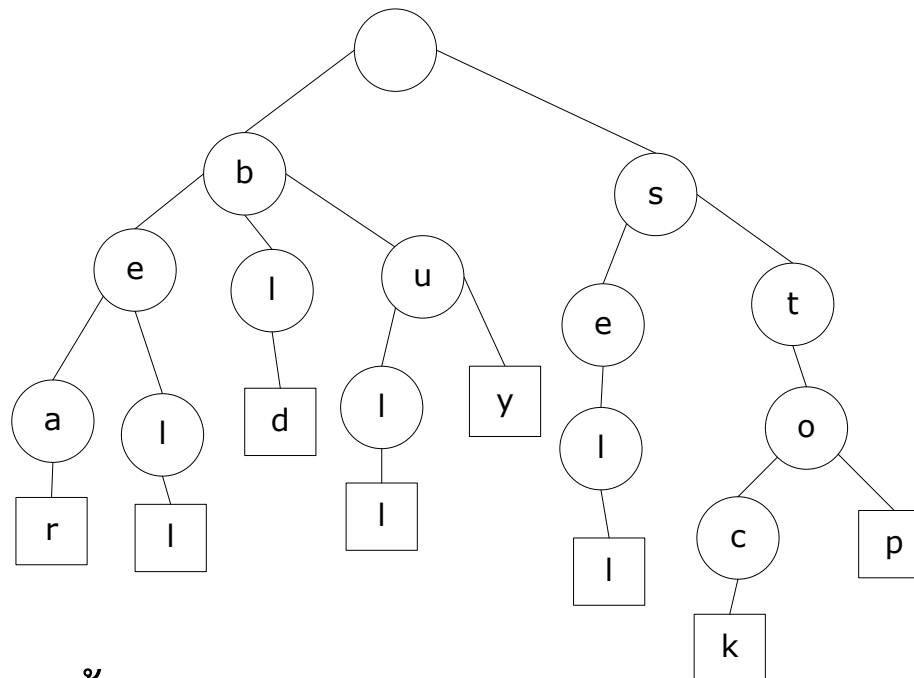
- เป็นต้นไม้ที่ออกแบบมาเพื่อเก็บสตริงและค้นหาสตริง
- มาจากคำว่า “retrieval”
- ใช้ในการค้นคืนข่าวสาร เช่นการค้นหาลำดับของ DNA ในฐานข้อมูล genomic
- ถ้าส่วนด้านหน้า (prefix) ของข้อมูลหรือข้อความคือองค์ประกอบหลักในการค้นคืนข้อมูล ทรีถือได้ว่าเป็นโครงสร้างข้อมูลที่มีความเหมาะสม เช่น ต้องการค้นหาคำทุกคำที่ขึ้นต้นด้วย cat

คุณสมบัติของทรี



คุณสมบัติของทรีมีดังนี้ [อ้างอิง: ผศ. นันทน์ภัส โตอดีเทพย์]

- ทุกโหนดยกเว้นรากมีเลเบลกำกับเป็นแคแรคเตอร์ในเซตของอักขรทั้งหมด
- ลำดับของโหนดกิ่งเป็นลำดับแบบบัญญัติ (canonical) ของอักขร (หรือออกแบบใหม่)
- โหนดภายนอก (ใบ) แทนสตริง S ที่ได้จากการเชื่อมแคแรคเตอร์จากโหนดราก
- แต่ไม่ใช่โหนดใบก็เก็บสตริงได้เหมือนกัน เช่น กรณีที่มีทั้งคำว่า do, done และ dog อยู่ในทรี



ภาพ: ผศ. นันทน์ภัส โตอดีเทพย์

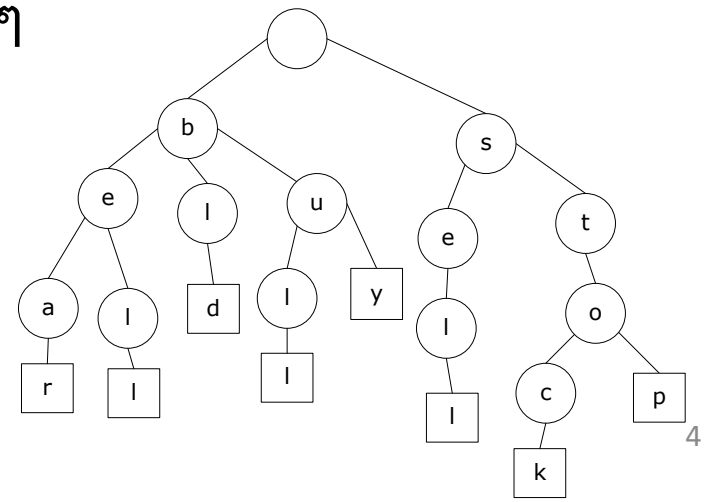
จากภาพข้างบน ทรีนี้เก็บคำว่า bear, bell, bid, bull, buy, sell, stock, และ stop

ข้อสังเกตเกี่ยวกับทรี



- ทรีทำหน้าที่คล้าย search tree ได้เหมือนกัน เช่น การจัดเรียงคำตามแบบบัญญัติ ภาษาอังกฤษเราสามารถเปรียบเทียบค่า a, b, c, ..., z เป็น 1, 2, 3, ..., 26 ได้ แล้วกำหนดให้พวกตัวอักษรแรกซ้ายสุด เป็นต้น
- การอ่านค่าของตัวอักษรในแต่ละโหนดแบบ preorder จะทำให้ได้คำที่เก็บไว้ในทรี (ต้องลบตัวอักษรเวลากลับไปหาโหนดพ่อด้วย)
- การค้นหาคำในทรีจะเร็วมาก แต่การลบคำจะช้า เพราะต้องตามลบโหนดด้านบนด้วย เช่น การลบคำว่า bear ออกจากทรีในตัวอย่าง ทั้งโหนด a และ r จะต้องถูกลบด้วย ทรีจึงไม่เหมาะกับการที่ต้องลบข้อมูลออกบ่อย ๆ

ทรีไม่ใช่ binary search tree เพราะสามารถมีโหนดลูกได้มากมายตามจำนวนอักษร



ทดลองสร้างทรีสำหรับเก็บคำศัพท์



เช่นเดิม เราเริ่มจากที่เก็บข้อมูล

```
#include <string.h>
```

```
using namespace std;
```

```
class TrieNode {  
public:
```

```
    bool end;
```

```
    char key;
```

```
    TrieNode* parent;
```

```
    TrieNode* link[26];
```

```
    TrieNode(char* word) {
```

```
        this->key = word[0];
```

```
        end = false;
```

```
        parent = NULL;
```

```
        for (int i = 0; i < 26; ++i)
```

```
            link[i] = NULL;
```

```
    };
```

```
};
```

ตัวระบุว่ามีคำที่สิ้นสุดในโหนดนี้หรือไม่

แต่เดิมเก็บตัวเลข ตอนนี้เก็บตัวอักษร

ลิงค์ไปโหนดลูกทั้งหมด เนื่องจากตัวอักษรภาษาอังกฤษมี 26 ตัว
จึงเตรียมไว้ทั้งหมด 26 ชุด

เก็บตัวอักษรตัวแรกของคำไว้
ในฐานะ key ของโหนด

คล้ายกับการกำหนดค่าเริ่มต้น left กับ right ซึ่ง
เป็นลิงค์ไปหาโหนดลูกใน binary search tree

แล้วตัวอักษรสัมพันธ์กับตัวเลขยังไง



วิธีเก็บตัวอักษรมีอยู่หลายมาตรฐาน เช่น ASCII, ISO 8859-11, และ Unicode

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

ขอบคุณข้อมูลจาก 2

Source: www.LookupTables.com

เราจะเลือกแบบ ASCII เพราะง่ายและเข้ากันได้กับหลายระบบ

แค่จะหาลิงค์ไปไหนดูลูกก็เป็นเรื่องที่ต้องคิด



```
int getSlot(char key) {  
    if (key >= 97) // อักษรตัวเล็กเริ่มจากค่า 97 ขึ้น  
        ไป  
          
        // convert to uppercase.  
        key -= 32;  
  
    if (key < 65 || key > 90)  
        return -1; // invalid slot  
    else  
        return (key - 65);  
}
```

อักษรตัวใหญ่จะมีค่าน้อยกว่าตัวเล็กอยู่ 32

อักษรตัวใหญ่จะมีค่าตั้งแต่ 65 ถึง 90

แปลงค่าให้กลายเป็น 0 - 25



ถึงคราวต้อง insert ข้อมูลกันแล้ว

แบ่งออกเป็นสองฟังก์ชันเพราะโหนดรากของทรียังไม่ได้เก็บตัวอักษรอะไรไว้และทำหน้าที่พิเศษ

ตัวจัดการโหนดราก

```
void insert(char* word,
TrieNode*& root) {
    if (root == NULL)
        root = new
TrieNode("*");

    int slot = getSlot(word[0]);
    if (slot == -1)
        return;

    insert2(word, root->link[slot], root);
}
```




```
void insert2(char* word, TrieNode*& subtree, TrieNode* parent)
{
    char key = word[0];
    if (key == '\\0') {
        parent->end = true;
    } else {
        int slot = getSlot(word[0]);
        if (slot == -1)
            return;        // invalid character
        else {

            // need new node
            if (subtree == NULL) {
                subtree = new TrieNode(word);
                subtree->parent = parent;
            }

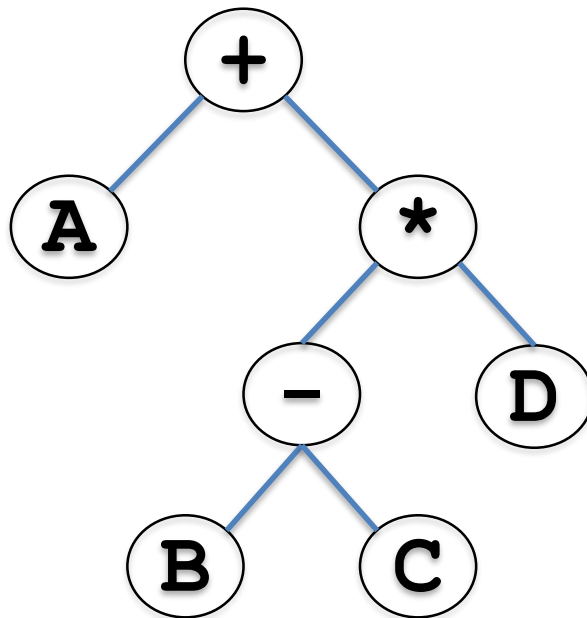
            if (word[1] == '\\0') {
                subtree->end = true;
            } else {
                int nextSlot = getSlot(word[1]);
                insert2(word+1, subtree->link[nextSlot], subtree);
            }
        }
    }
}
```

เทคนิคการบวกเลข **pointer** ไม่ค่อยมีใน
ภาษายุคใหม่เพราะทำให้โปรแกรมมีช่องโหว่
ด้านความปลอดภัยได้ง่าย



การประยุกต์ต้นไม้กับนิพจน์การคำนวณ

- ให้เครื่องหมายการคำนวณ (Operator) เป็นโหนดพ่อของโอเปอเรนด์ทั้งสองที่เป็นลูกทางซ้ายและลูกทางขวา [อ้างอิง: ผศ. นันทน์ภัท โตอติเทพย์] จากรูป แทนนิพจน์ $A+(B-C)*D$ ด้วยต้นไม้



ซึ่งเราสามารถแปลงนิพจน์ infix notation เป็น postfix notation โดยการแวะผ่านต้นไม้แบบ inorder

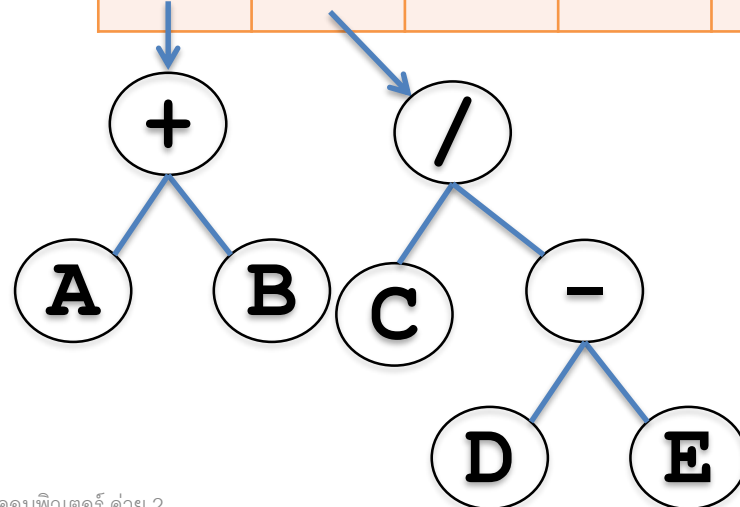
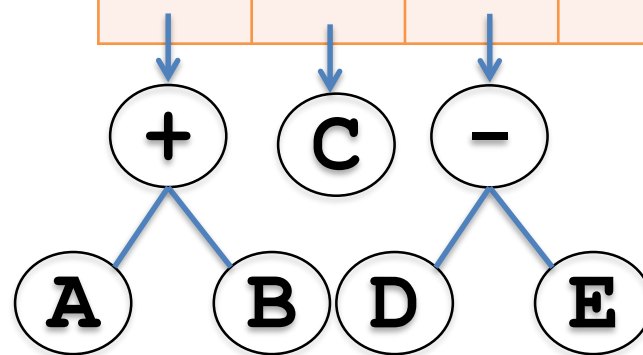
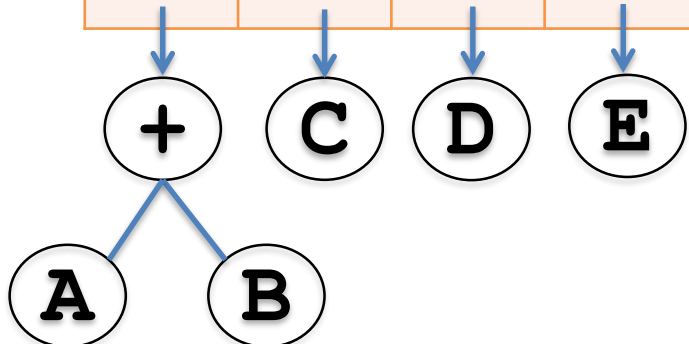
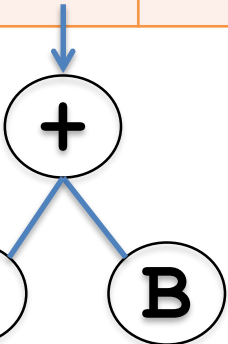
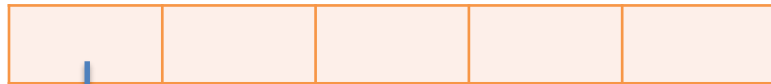
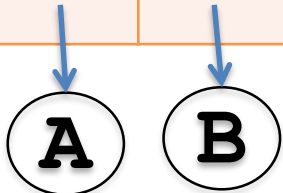


การสร้างต้นไม้เก็บนิพจน์จากนิพจน์ postfix

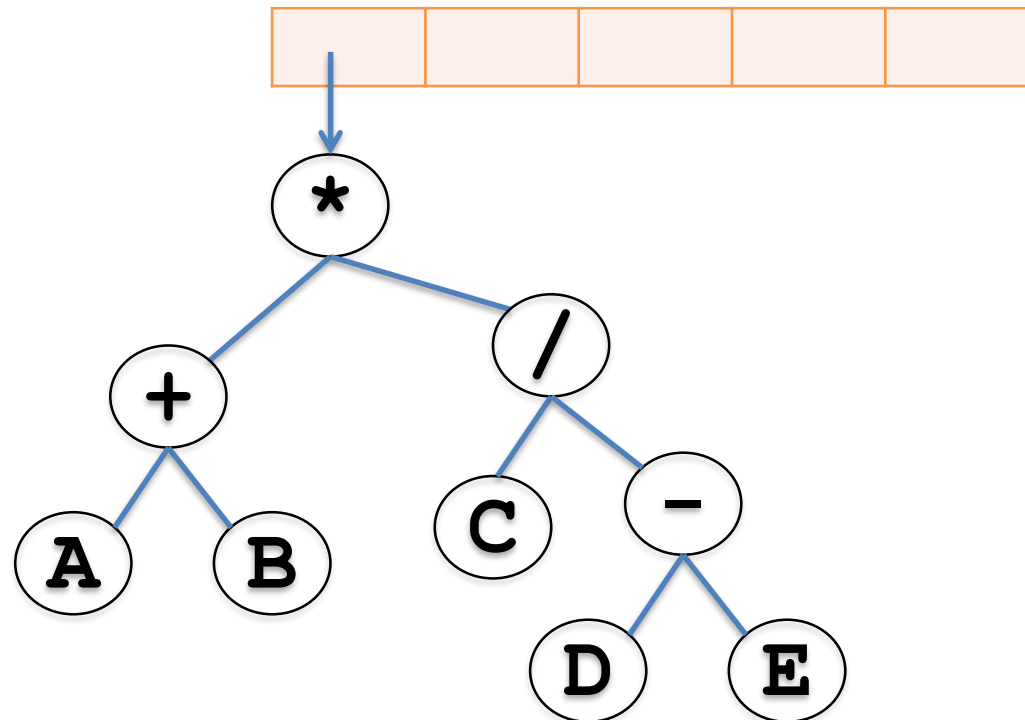
- อ่านนิพจน์ postfix ครั้งละ 1 สัญญลักษณ์จนหมด
 - ถ้าเป็นโอเปอเรนด์ สร้างโหนด และพুষค่าพอยเตอร์ของโหนดนั้นลงสแตค
 - ถ้าเป็นโอเปอเรเตอร์ ให้สร้างต้นไม้โดยมีโอเปอเรเตอร์เป็นโหนดแรกและต้นไม้ย่อยขวาและซ้ายได้จากพอยเตอร์ที่ป้อนจากสแตคและพুষพอยเตอร์ของต้นไม้ใหม่ลงสแตค



postfix expression: A B + C D E - / *



postfix expression: A B + C D E - / *



เรื่อนำรู้



- ในหลาย ๆ ปัญหาทรีของเราอาจจะไม่ต้องยุ่งกับการลบโหนด แต่เราก็ใช้การลบโหนดเพื่อช่วยในการอัปเดตคีย์ได้
- ต้นไม้ที่สมดุลอย่าง AVL กับ Red-Black Tree รับประกันความเร็วในการทำงาน แต่ก็ไม่เหมาะที่จะเอาไปใช้ในการแข่ง เพราะใช้เวลาสร้างนาน
อย่างไรก็ตามต้นไม้สองแบบนี้อาจจะเหมาะในการใช้งานจริง
- การรู้ข้อกำหนดของโจทย์ เช่น “จะมีข้อมูลเข้าไม่เกิน 1000 บรรทัด” จะทำให้เราสามารถสร้างที่เก็บข้อมูลแบบตายตัว เช่น `Titem item[1000]`; ขึ้นมาได้ โดยไม่ต้องกังวลว่าจะต้องไปหาขนาดของจำนวนข้อมูลก่อน
➔ โปรแกรมจะเขียนง่าย เหมาะกับการแข่งที่มีเวลาน้อยอย่างโอลิมปิกวิชาการ



Self-balancing Binary Search Tree

- โครงสร้างที่การันตีความสูงของต้นไม้จะเป็น $O(\log_2 n)$ แม้จะมีการปรับเปลี่ยนข้อมูลในโครงสร้างแบบไดนามิก
 - ตัวอย่างของโครงสร้างแบบนี้เช่น
 - AVL Trees
 - B-trees
 - Red-black Trees
- เป็นต้น



AVL Trees

- ถึง AVL Trees ไม่เหมาะกับการแข่งแต่เหมาะสำหรับทำแอปพลิเคชัน
- เพื่อให้การค้นหาใช้เวลาน้อยที่สุด เราจำเป็นต้องทำให้โครงสร้างต้นไม้มีความสมดุลด้านความสูงให้มากที่สุด ซึ่งมีนักคณิตศาสตร์ชาวรัสเซียสองคนคือ G.M. ADEL'SON-VEL'SKII และ E.M. LANDIS ได้คิดค้นรูปแบบไว้แล้ว ซึ่งวิธีการก็ได้ให้ชื่อเพื่อเป็นเกียรติกับพวกเขาชื่อ AVL Trees
- AVL Trees จะใช้เวลาในการค้นหา แทรกข้อมูล และลบข้อมูลในต้นไม้ขนาด n โหนด เสร็จในเวลา $O(\log n)$ เท่านั้น แม้แต่ในกรณีที่แย่ที่สุดก็ตาม

ตัวอย่าง <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

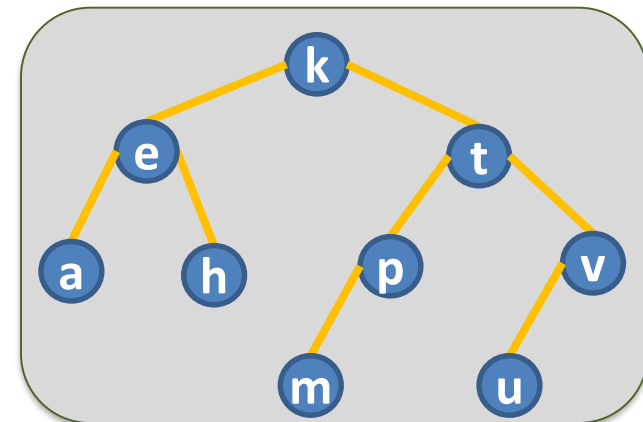
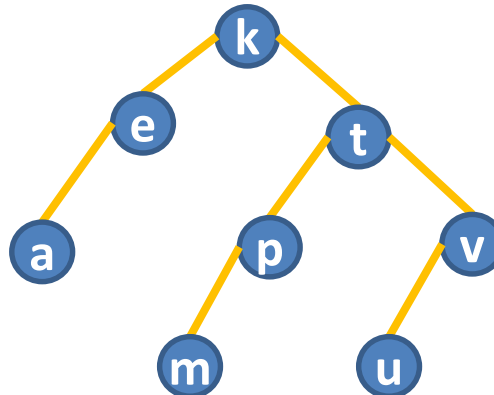
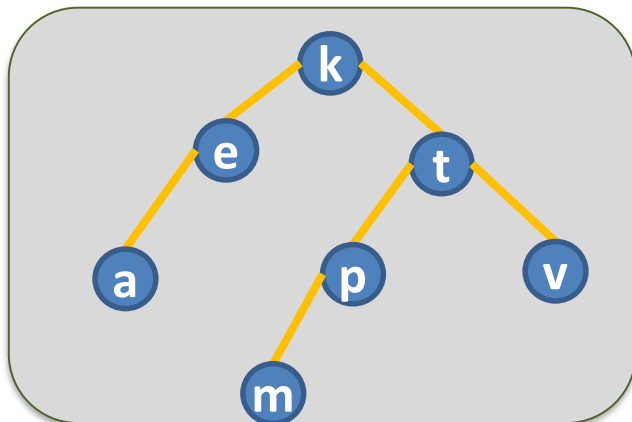
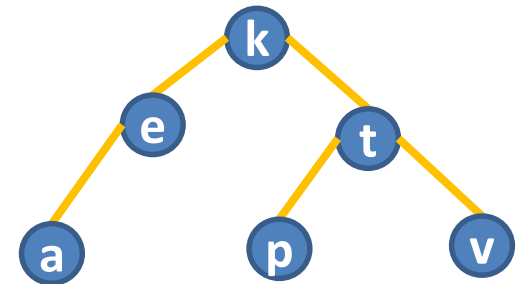
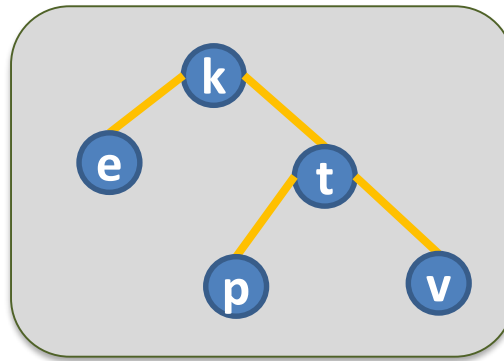
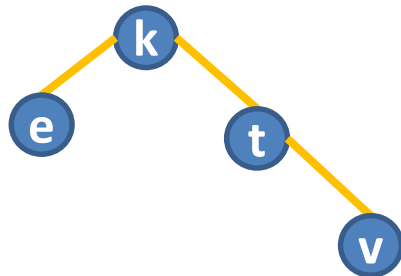
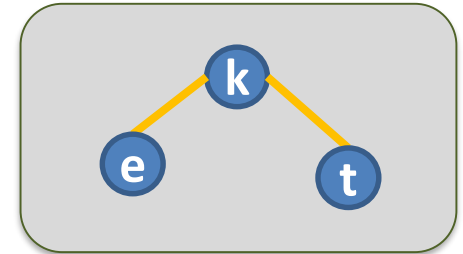
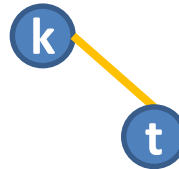
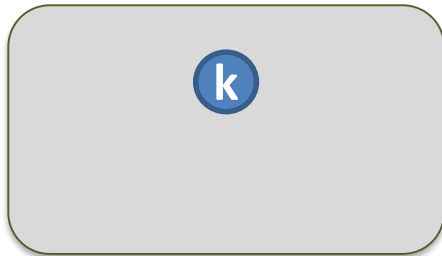


AVL Trees (ต่อ)

- AVL Trees คือ binary search tree ที่มีความสูงของต้นไม้ย่อยทางขวาและต้นไม้ย่อยทางซ้ายมีความแตกต่างไม่เกิน 1 และต้นไม้ย่อยทั้งซ้ายและขวาต้องเป็นโครงสร้าง AVL Trees เช่นกัน
- การดำเนินการกับโครงสร้าง AVL Trees จะต้องทำให้คงไว้ซึ่งโครงสร้าง AVL Trees เช่นเดิม ดังนั้นเราต้องมาพิจารณาว่าทำสิ่งต่อไปนี้อย่างไร
 - Insertion of a node
 - Deletion of a node



Insertion of a node



การดำเนินงาน 2 อย่างที่ช่วยรักษาสมดุล

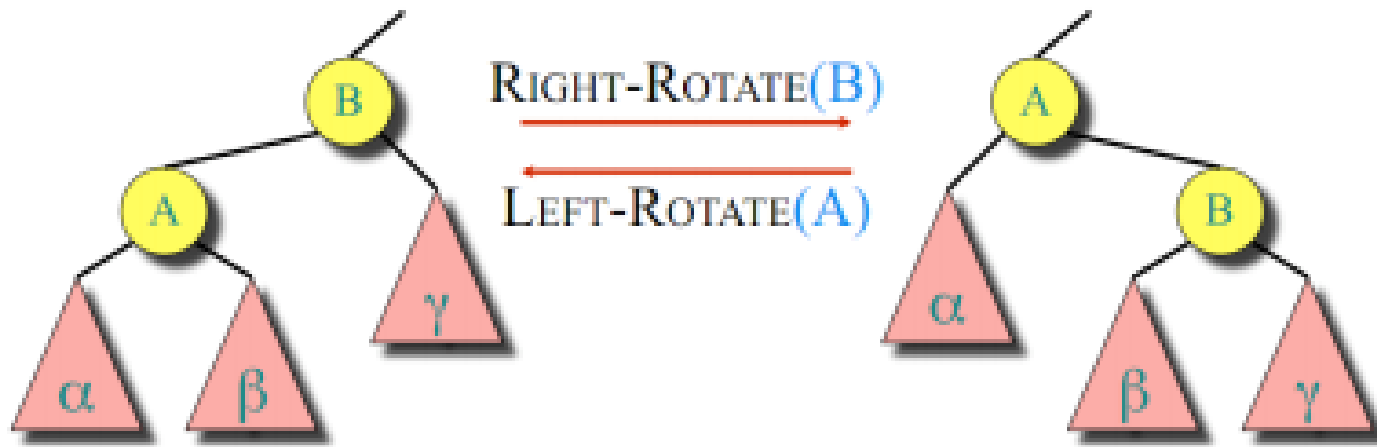


- การหมุนเวียน (rotation) เป็นการจัดลำดับโหนดใหม่เพื่อให้ความสูงของต้นไม้ของต้นไม้ย่อยเปลี่ยนตำแหน่ง ทำให้รักษาสมดุลได้
- Rotation ต้อนการเปลี่ยนตำแหน่งแค่ left, right และ parent ของบางโหนดเท่านั้น
- มี Rotation ที่น่าสนใจ 2 คือ Left rotation และ Right rotation



Rotation

ภาพแสดงการดำเนินงาน Rotation ทั้ง 2 แบบ



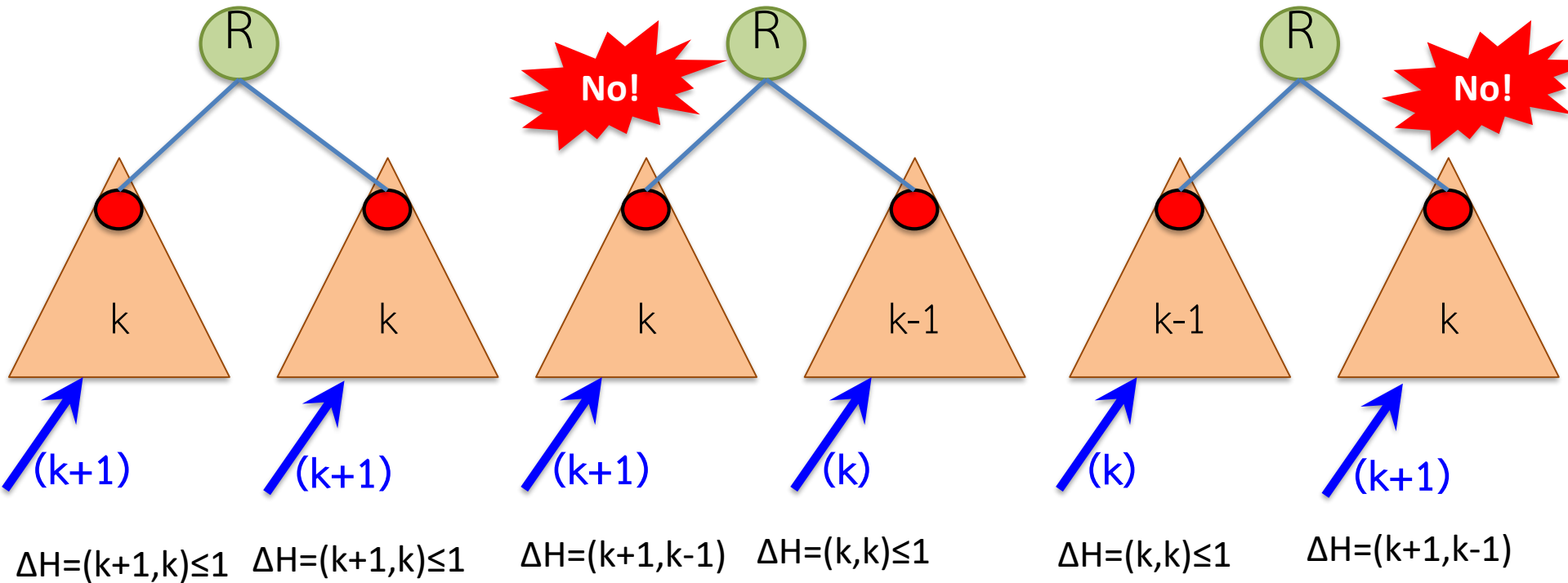
ที่มาภาพ : <http://courses.csail.mit.edu/6.006/spring11/>

Right_Rotate (B) กลับไปเป็นลูกทางขวา (ต้องไปอยู่ใต้ลูกทางซ้าย)

Left_Rotate(A) กลับไปเป็นลูกทางซ้าย (ต้องไปอยู่ใต้ลูกทางขวา)



AVL Tree ก่อน insert ในสถานการณ์ต่างๆ

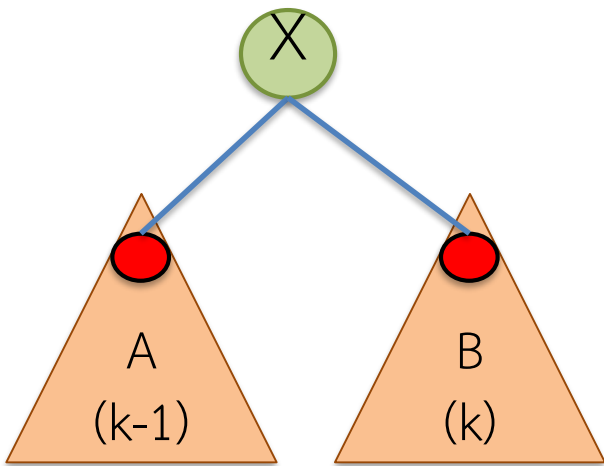


ข้อสังเกต ถึงแม้ว่าต้นไม้ย่อยจะมีความสูง k แต่เมื่อเราแทรกโหนดใหม่ ไม่จำเป็นว่าความสูงของต้นไม้ย่อยจะต้องเปลี่ยนไปเป็น k+1 เสมอไป

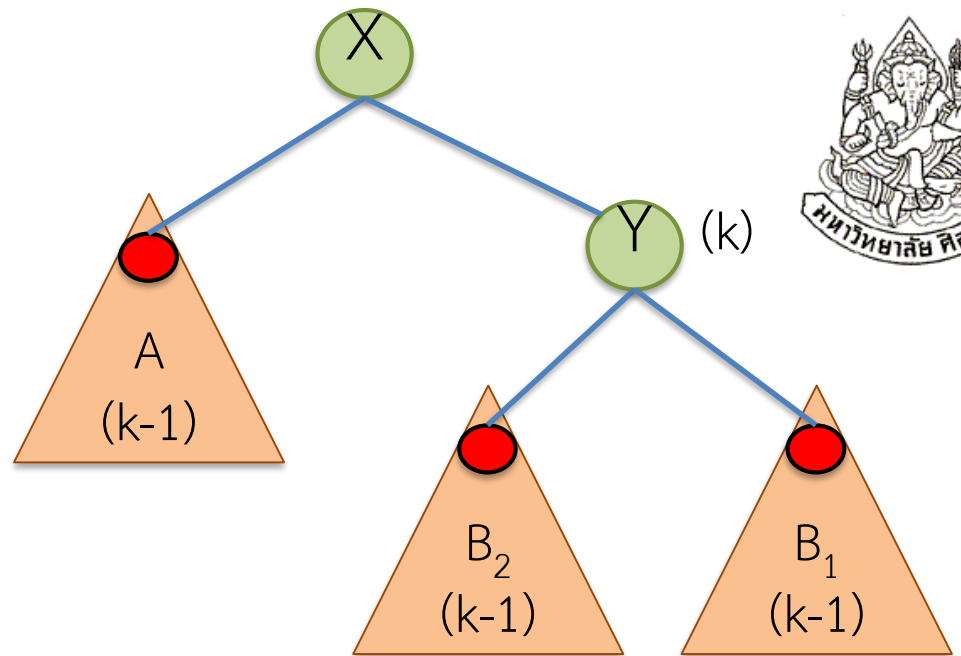


มีความเป็นไปได้ 3 กรณี

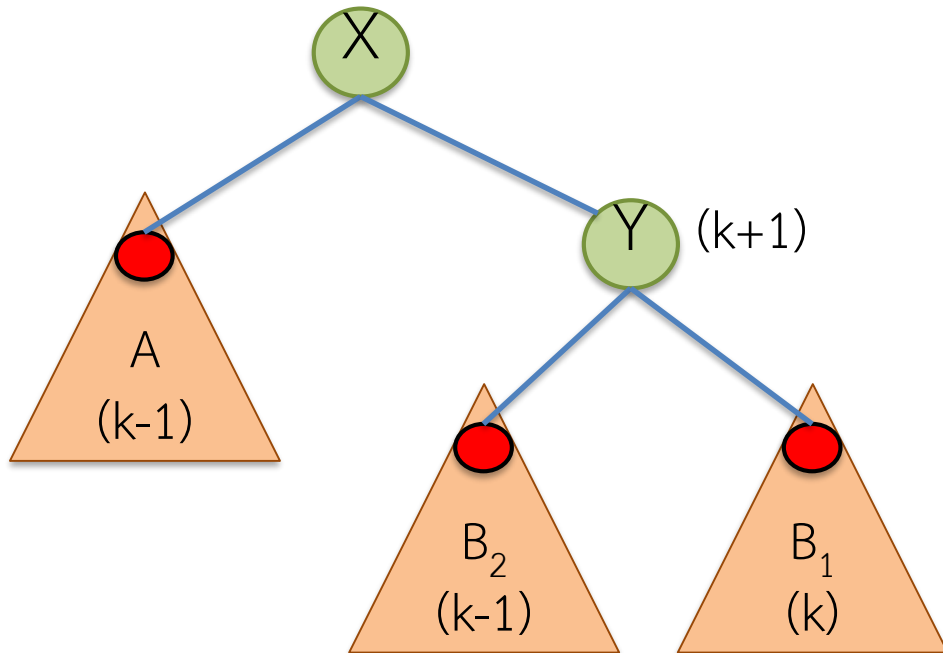
- โหนดลูกของ R มีความสูงเท่ากันทั้งสองด้าน ในกรณีนี้ ไม่ว่าโหนดใหม่จะถูกแทรกไว้ที่ด้านใดก็ไม่ทำให้ความสูงของต้นไม้ย่อยทั้งสองต่างกันเกินกว่า 1 ไปได้
- โหนดลูกของ R ด้านซ้ายมีความสูงมากกว่าโหนดลูกของ R ด้านขวาอยู่ 1 ถ้าโหนดใหม่ถูกเพิ่มที่ด้านซ้ายจะทำให้ต้นไม้ไม่สมดุล
- โหนดลูกของ R ด้านขวามีความสูงมากกว่าโหนดลูกของ R ด้านซ้ายอยู่ 1 ถ้าโหนดใหม่ถูกเพิ่มที่ด้านขวาจะทำให้ต้นไม้ไม่สมดุล
- เมื่อใดก็ตามที่ AVL Tree ไม่สมดุล เราจะต้องทำการ rotation เพื่อจัดโหนดให้ต้นไม้เกิดความสมดุล



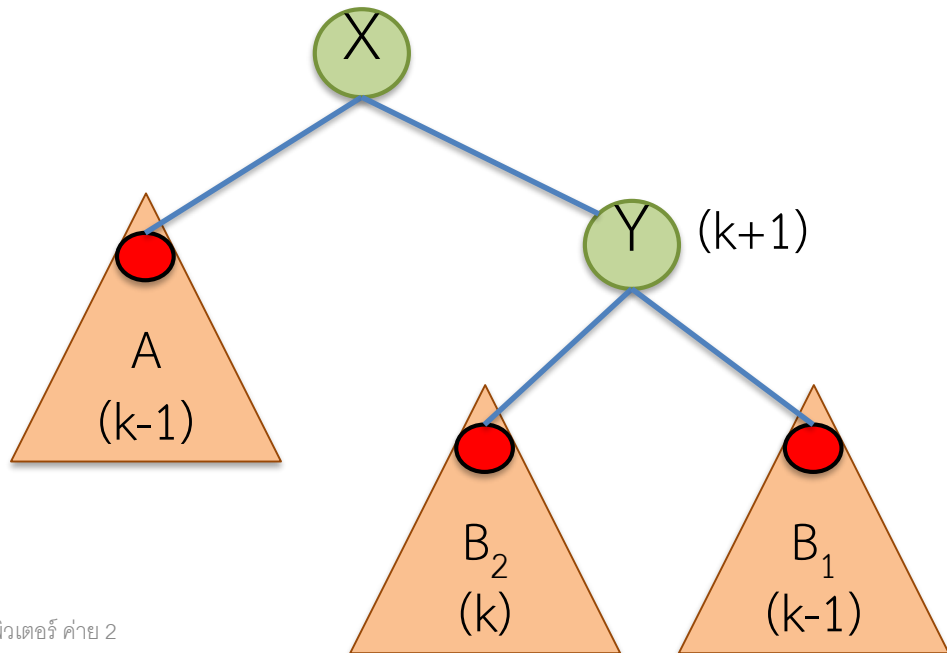
||



CASE 1 : โหนดใหม่แทรกที่ B_1



CASE 2 : โหนดใหม่แทรกที่ B_2

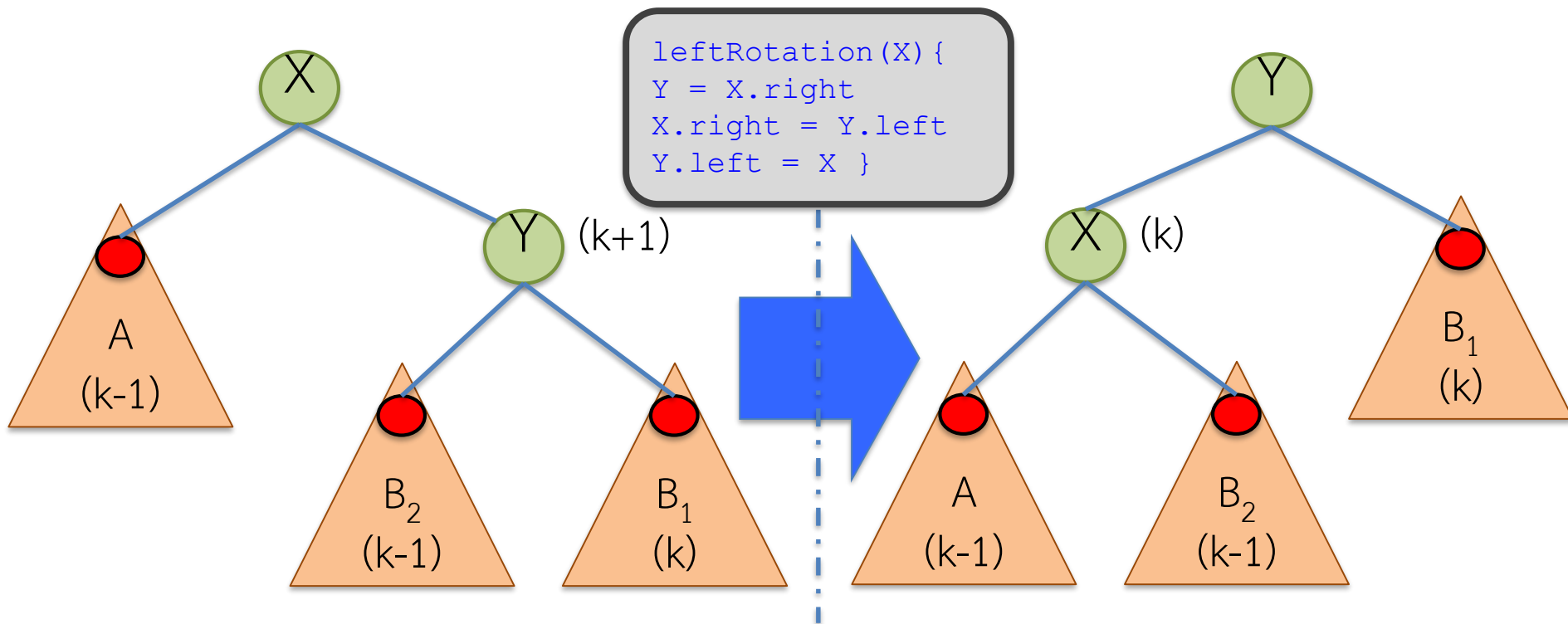




Case 1: โหนดใหม่ถูกแทรกที่ต้นไม้อย่อยขวาสุด

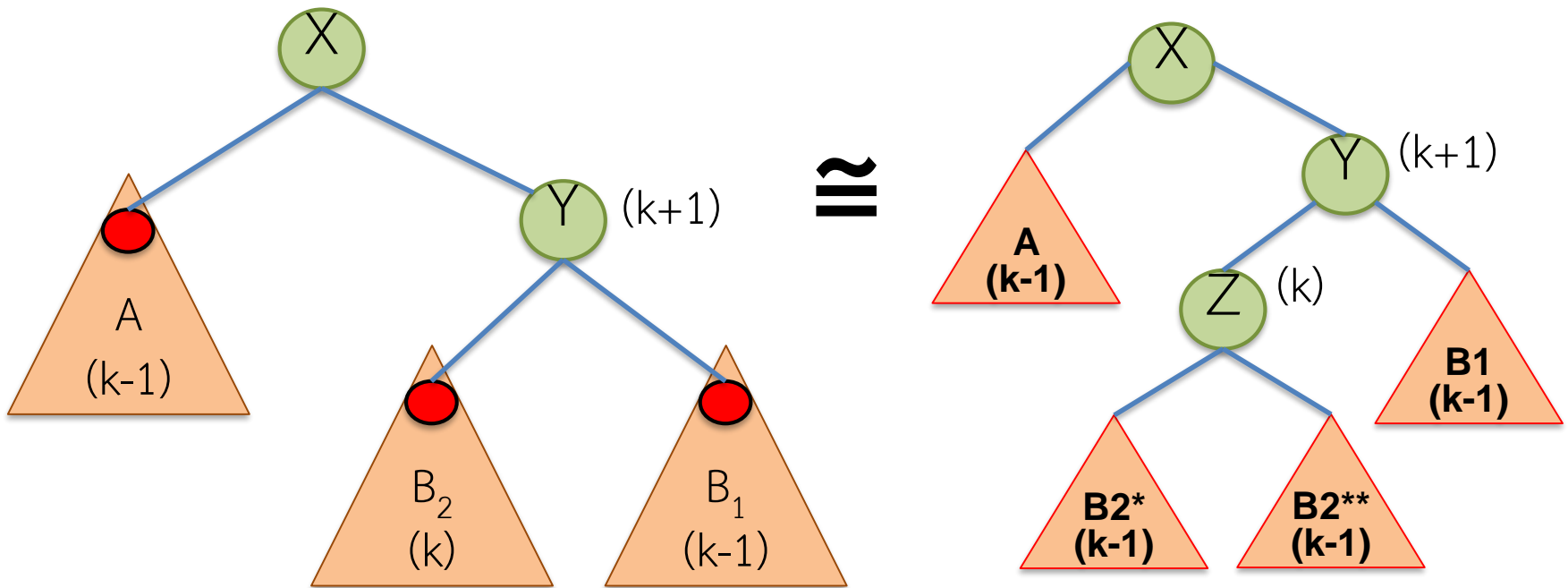
- สามารถใช้ left rotation เพื่อช่วยจัดลำดับในต้นไม้ใหม่

```
leftRotation(X) {  
  Y = X.right  
  X.right = Y.left  
  Y.left = X  
}
```



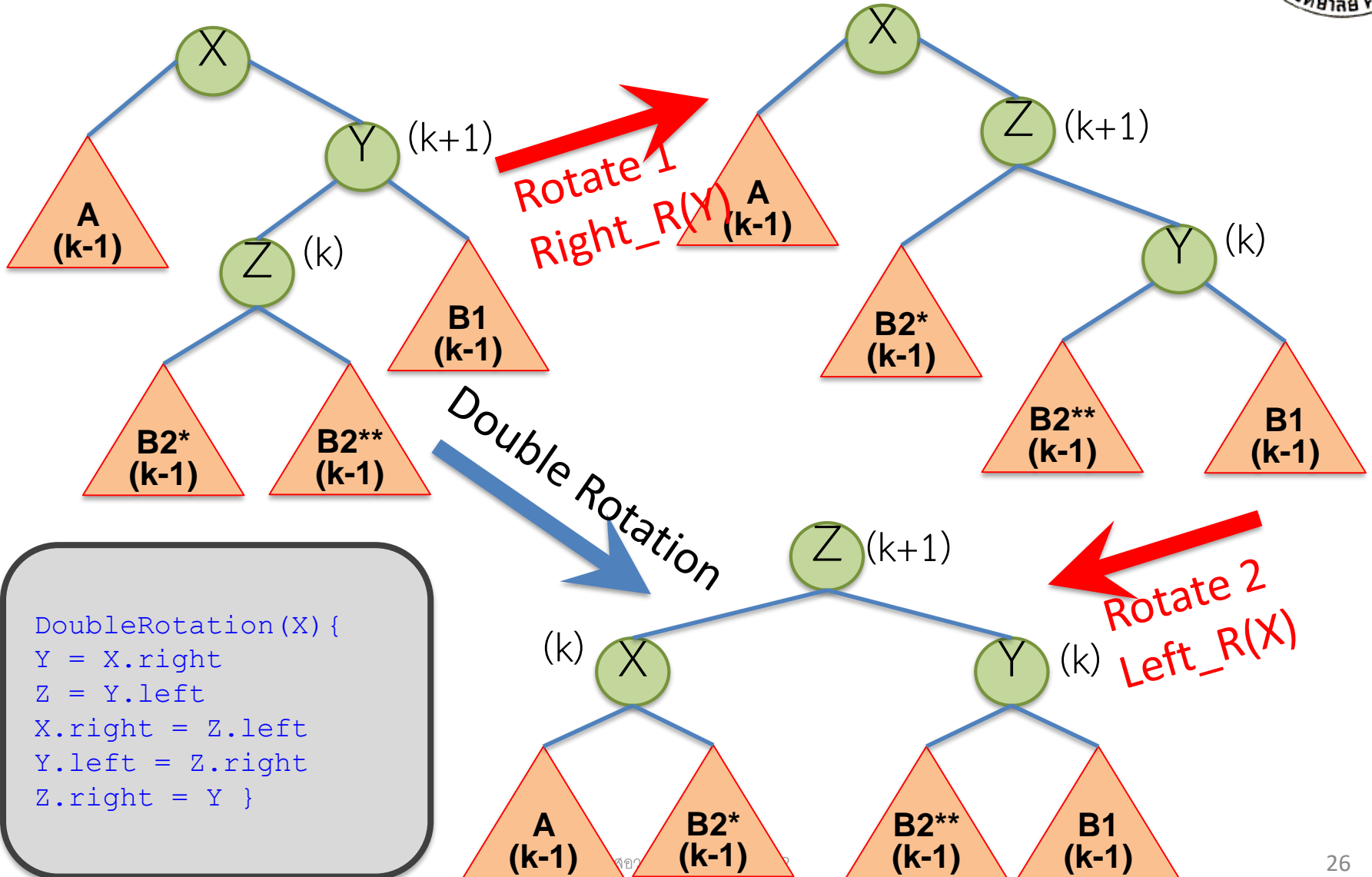
ลำดับของสมาชิกยังคงเหมือนเดิมคือ $A < X < B_2 < Y < B_1$

CASE 2: โหนดใหม่ถูกแทรกที่ต้นไม้ย่อยซ้ายของ Y





Case 2: ต้องใช้ double rotation



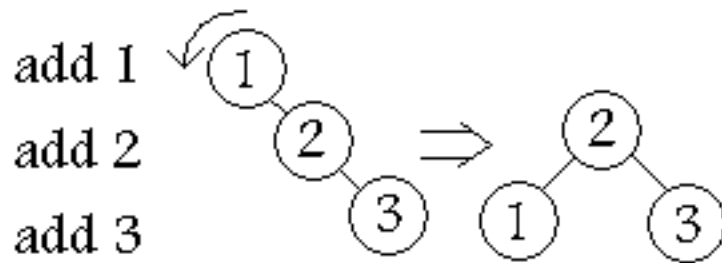


การลบโหนดออกจาก AVL Tree

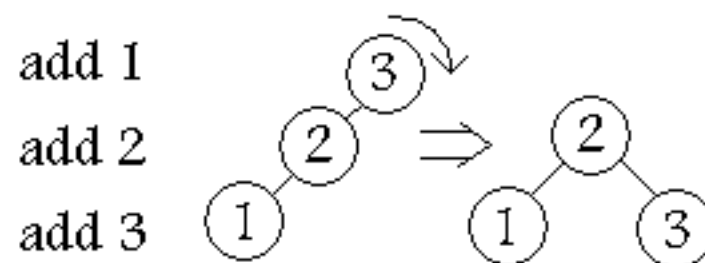
- การลบโหนดใดๆ ออกจาก AVL Tree สามารถใช้วิธีการลบโหนดเหมือน BST แต่ต้องมีการทำสมดุลให้ต้นไม้ โดยใช้วิธีการ rotation ซึ่งสิ่งที่ต้องคำนึงถึงคือเมื่อเราทำสมดุลให้กับโหนดใดแล้วโหนดพ่อของโหนดดังกล่าวอาจเกิดความไม่สมดุลได้ เราจึงต้องตามไปปรับสมดุลด้วย left rotation หรือ double rotation ไปเรื่อยๆ ตามแต่สถานการณ์ว่าต้นไม้ย่อยด้านใดมีความสูงมากกว่ากัน ซึ่งก็ใช้วิธีการเหมือนกับการเพิ่มโหนดนั่นเอง



Single Rotations



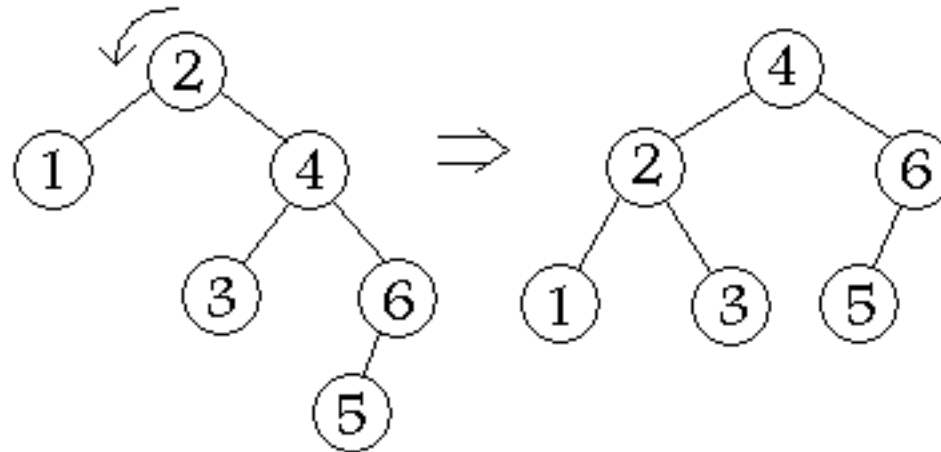
simple left rotation



simple right rotation

add 2
add 1
add 4
add 3
add 6
add 5

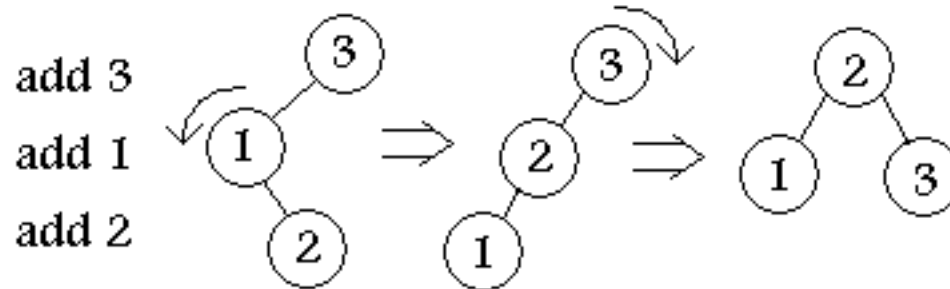
complex left rotation



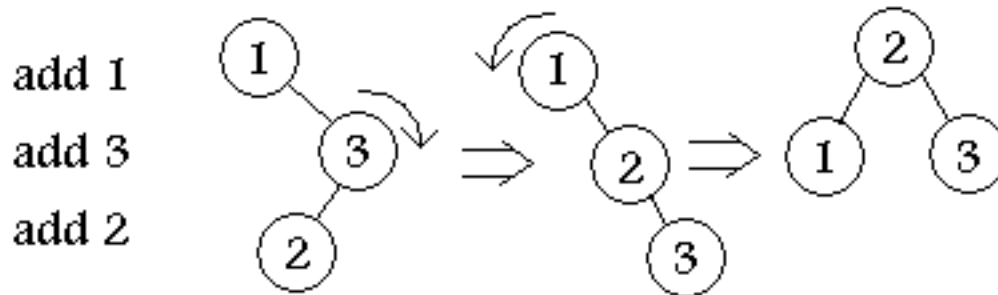


Double Rotations

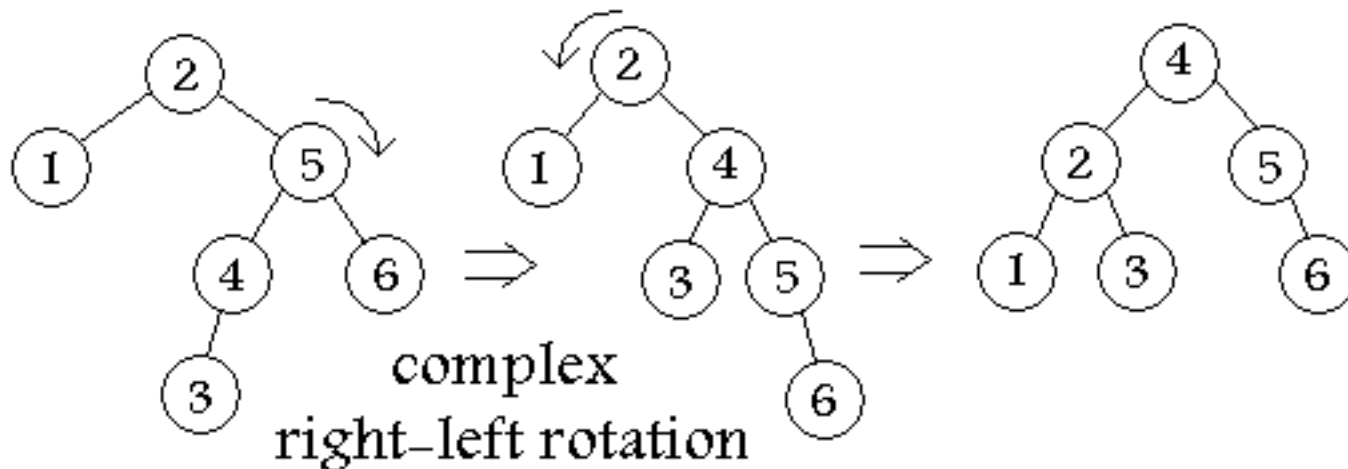
simple
left-right
rotation



simple
right-left
rotation



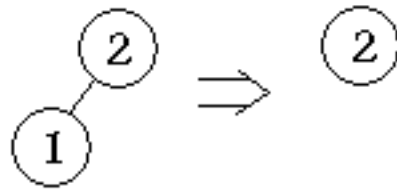
add 2
add 1
add 5
add 4
add 6
add 3





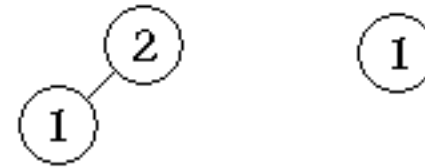
Simplified Deletion

add 2
add 1
delete 1



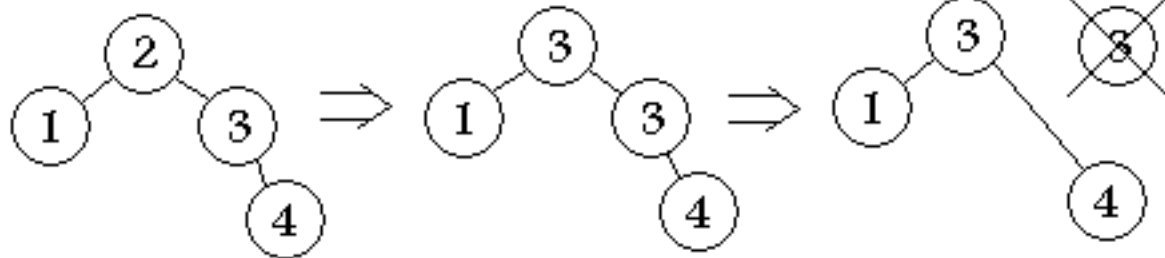
A leaf

add 2
add 1
delete 2



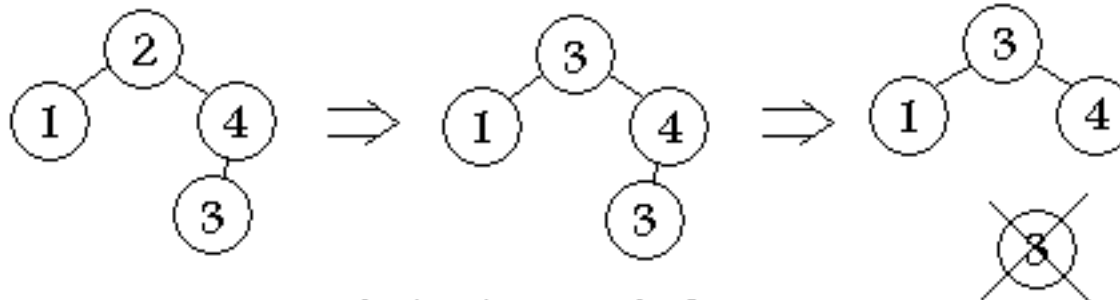
has no right

add 2
add 1
add 3
add 4
delete 2



right has no left

add 2
add 1
add 4
add 3
delete 2



right has a left

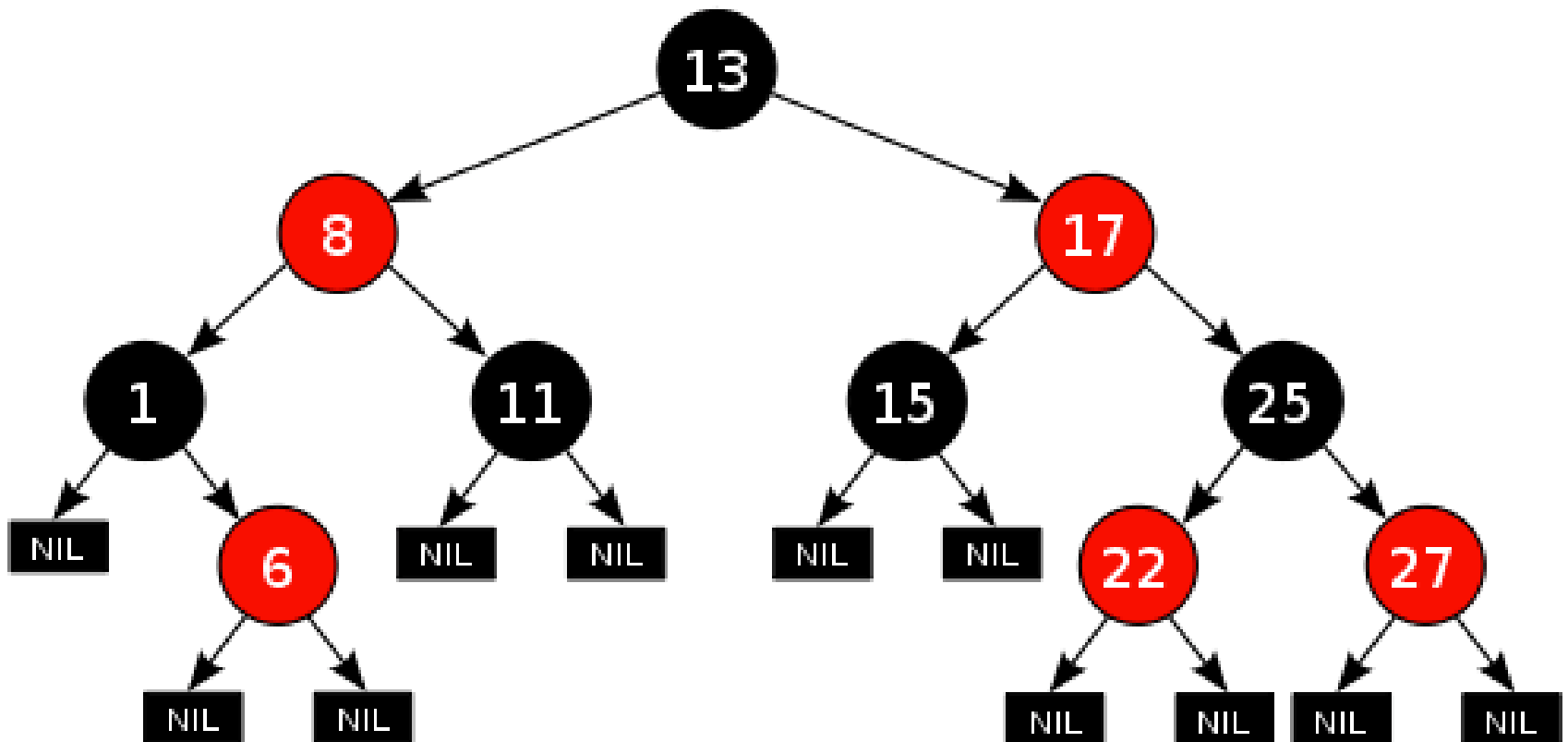


Red-black trees

- ต้องเพิ่มฟิลด์พิเศษลงในโหนดนั้นคือสี โดยกำกับให้แต่ละโหนดต้องมีสีซึ่งจะเป็นได้แค่ 2 สีคือสีดำ (black) หรือสีแดง (red)
- ต่อไปนี้คือคุณสมบัติของ Red-black trees
 1. ทุกโหนดจะต้องมีสีแดงหรือสีดำเท่านั้น
 2. รากและใบ (คือค่า NULL เพราะถือเป็นโหนดสิ้นสุดจริงๆ) มีสีดำ
 3. ถ้าโหนดใดเป็นสีแดง โหนดลูกจะต้องเป็นสีดำทั้งสองโหนด
 4. เส้นทางใดๆ (path) จากโหนด x ไปโหนดลูกที่เป็นใบของมันจะมีจำนวนโหนดสีดำเท่ากันเสมอ เรียก $\text{black-height}(x)$



ตัวอย่าง Red-black Trees





Trees in STL

- น่าเสียดาย The Standard Template Library ไม่ได้มีเทมเพลตต้นไม้ในชื่อว่า tree แต่มี container ที่เมื่อใส่ข้อมูลไป ตัวเทมเพลตได้สร้างการเก็บข้อมูลเป็นแบบ self-balancing binary search tree และเนื่องจาก the self-balancing BST จะรักษาสสมดุลของต้นไม้ ดังนั้นจึงมั่นใจได้ว่าเวลาในการค้นหาจะมีค่า $O(\log_2 n)$ เสมอ แม้จะมีการเปลี่ยนโครงสร้างข้อมูลภายในก็ตาม
- ตัวคอนเทนเนอร์ที่เรียกว่าคือ `map<T1,T2>`



std::map<T1,T2> container

- The STL map<T1,T2> บางครั้งถูกเรียกว่า associative array เพราะถูกออกแบบมาให้ทำการแมปค่าจะชนิดข้อมูล T1 ไปยังชนิดข้อมูล T2
- เริ่มจากอาร์เรย์ธรรมดา ซึ่งจริงๆ แล้วเป็นการแมปค่าจะจำนวนเต็มไปยังชนิดข้อมูลที่กำหนด ตัวอย่างเช่น

```
string fruits[3] = {"Apple", "Orange", "Banana"};
```

เมื่อเราต้องการอ้างถึงผลไม้ก็ใช้ตัวเลขเป็นตัวชี้ ดังนี้

```
cout << fruits[0];  
cout << fruits[1];
```

0	1	2
"Apple"	"Orange"	"Banana"



std::map<T1,T2> Container (2)

- จากอาร์เรย์ เราสามารถอ้างอิงถึงชื่อผลไม้จากตัวเลข

```
string fruits[3] = {"Apple", "Orange", "Banana"};
```

0	1	2
"Apple"	"Orange"	"Banana"

- แต่ถ้าเราต้องการใช้ชื่อผลไม้อ้างอิงตัวเลขล่ะ อาร์เรย์ทำไม่ได้ แต่ map<T1,T2> สามารถทำได้ โดย

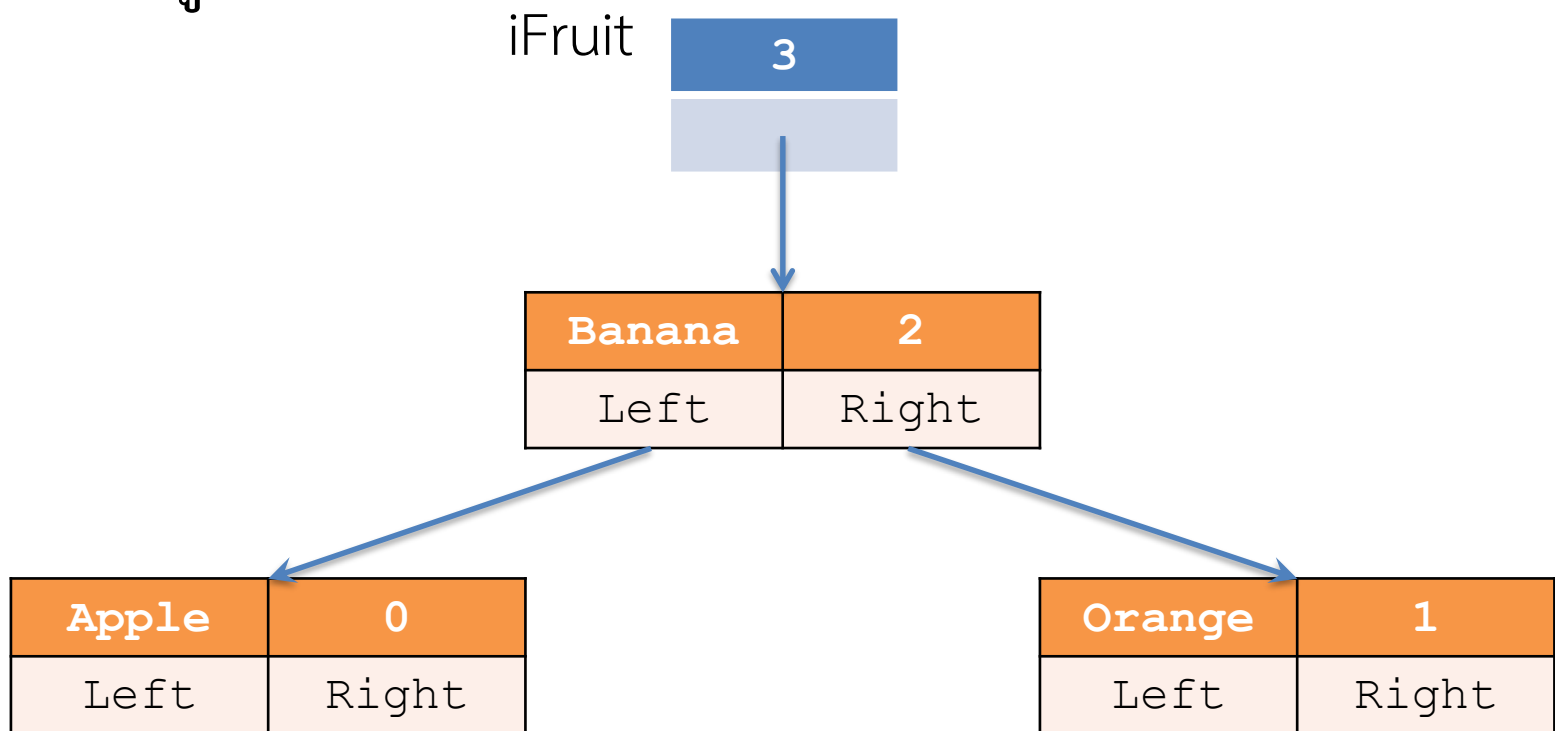
```
map<string, int> iFruit;  
iFruit["Apple"] = 0;  
iFruit["Orange"] = 1;  
iFruit["Banana"] = 2;
```

["Apple"]	["Orange"]	["Banana"]
0	1	2



`std::map<T1,T2>` container

- ซึ่งความจริงแล้วโครงสร้างภายในของ `iFruit` จะเป็น self-balancing BST ดังรูป





std::map<T1,T2> & Operator

ตัวดำเนินการ	ความหมาย
bool empty()	เป็นจริงเมื่อไม่มีค่า
Int size()	จำนวนค่าที่มีใน map
Int erase(T1 aValue)	ลบค่า aValue
Void clear()	ลบค่าทุกค่าใน map
Iterator find (T1 aValue)	คือค่าที่คู่กับ aValue ทุกค่า
Int count(T1 aValue)	นับค่าชนิดข้อมูล T1 มีค่า aValue
Iterator begin()	คือค่าแรกของ map
Iterator end()	คือพอยเตอร์ของค่าสุดท้ายของ map



ตัวอย่างโปรแกรม

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() {
    map<string, string> mascots;
    mascots["China"] = "Panda";
    mascots["Thailand"] = "Elephant";
    mascots["Malaysia"] = "Tiger";
    cout << "enter the name of country:";
    string country;
    getline(cin, country);
    map<string, string>::iterator it = mascots.find(country);
    if (it != mascots.end())
        cout << "Answer " << mascots[country] << endl;
    else
        cout << "missing country from database" << endl;
}
```



มุมมองคำถามและทบทวน

สิ่งที่เรียนไปทั้งหมด

- นิยามต้นไม้ และส่วนประกอบต่างๆ ของต้นไม้
- Binary Search Tree
- Self-balancing BST
- `std::map<T1,T2>`



คำถาม

- จงสร้าง self-balancing BST จากลำดับต่อไปนี้

23 44 20 2 30 56 60 32 65 48 46