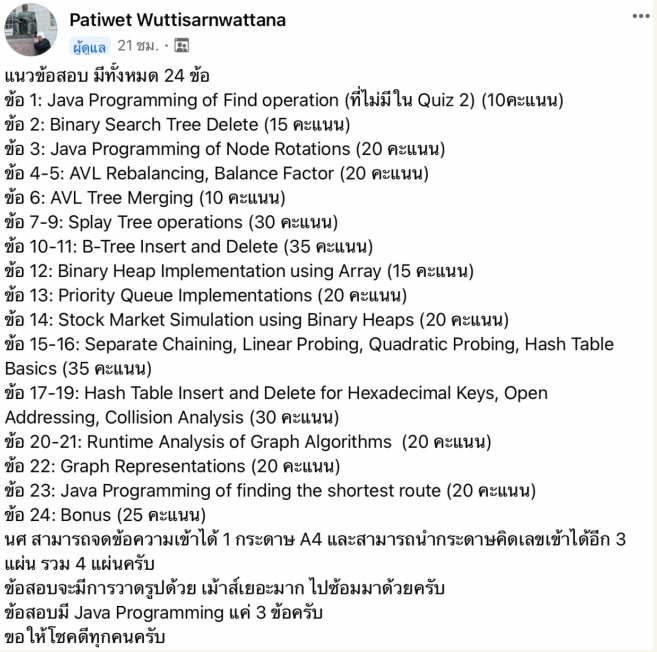
Data Structure

- PART FINAL -



Programming

- BST Find Operation
- AVL Node Rotation
- Graph Find Shortest Route

For the rest

= Understanding & Draw

BST Delete Operation

1. Empty Tree
2. Key not found

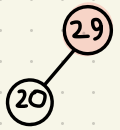
} ERROR

3. **Root ไม่มี child** "ลบได้เลย"

delete (29) (29) → ทางเปล่า root = null

4. **Root มี child คนเดียว**
"promote ลูกขึ้นมาแทน"

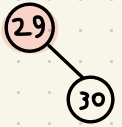
delete (29) (มีลูกซ้าย)



⇒

newRoot = root.left
newRoot.parent = null
root.left = null
root = newRoot

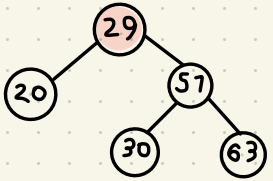
delete (29) (มีลูกขวา)



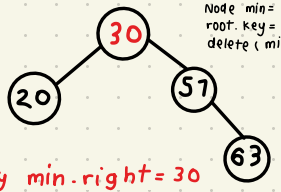
⇒

newRoot = root.right
newRoot.parent = null
root.right = null
root = newRoot

5. **Root มี child 2 คน**
"หา key min.right มาแทน key ของ root
แล้วลบ Node min.right ออก"



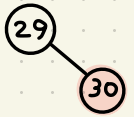
⇒



Node min = findMin(root.right)
root.key = min.key
delete (min)

key min.right = 30
ย้ายมาแทน 29 & ลบ node 30 ออก

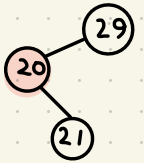
6. **Node ไม่มี child** "ลบเลย"



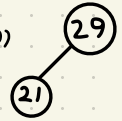
delete (30)
⇒

node30.parent = null
node29.right = null

7. **Node มี child คนเดียว** "promote ลูกขึ้นมาแทน"

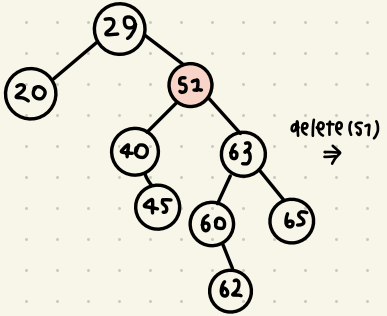


delete (20)
⇒

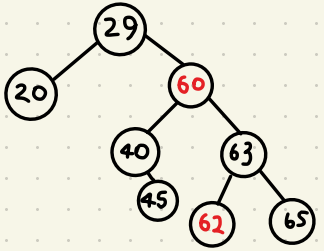


Node parent = Node20.parent
Node child = Node20.right
node.parent = null
node.child = null
parent.left = child
child.parent = parent

8. **Node มี child 2 คน**
"หา key min.right มาแทน key ของ Node
แล้วลบ Node min.right ออก"



delete (51)
⇒

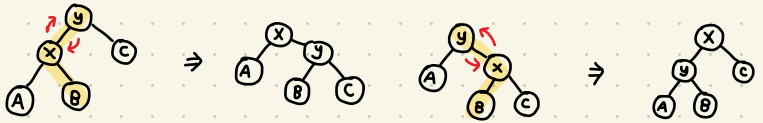


Rotation Concept ลืมคำว่า AVL ไปก่อน

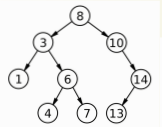
- Single Rotation

"วงนอกอยู่เฉยๆ เหมือนเดิม วงใน เปลี่ยน parent"

from Left

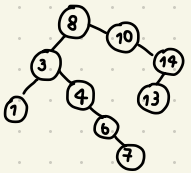


from Right



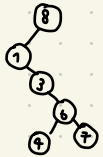
- SingleRotateFromLeft(6)
- SingleRotateFromRight(6)
- SingleRotateFromLeft(3)
- SingleRotateFromRight(3)
- SingleRotateFromLeft(8)
- SingleRotateFromRight(8)

✓1

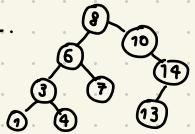


✓2.

✓3.

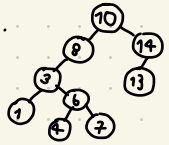


✓4.



✓5.

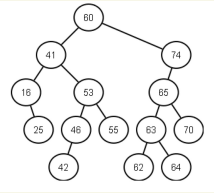
✓6.



- Double Rotation

→ Inner case อย่างเดียว ไม่มี out!

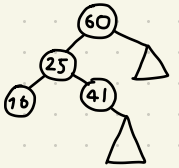
ยก z ขึ้น



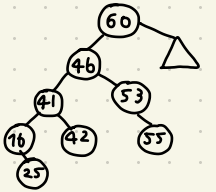
- DoubleRotateFromInnerLeft(41)
- DoubleRotateFromInnerRight(41)
- DoubleRotateFromInnerLeft(60)
- DoubleRotateFromInnerRight(60)

- DoubleRotateFromOuterLeft(60)

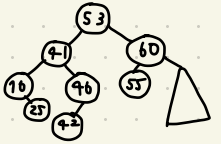
1.



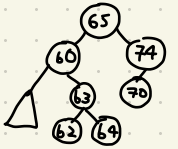
2.



3.



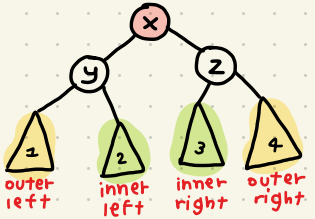
4.



Apply Rotation for Rebalancing!

ฟื้นความจำ AVL

- what if node x พัง ?



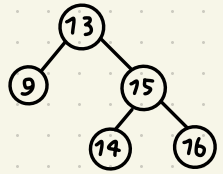
พังตรงไหน ?

- outer ? → Single Rotation
- inner ? → Double Rotation

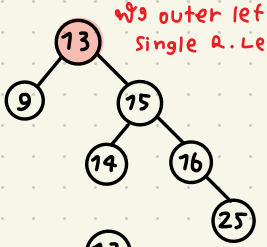
Insert & Delete

→ ใส่ node → ไล่เช็คขึ้นไปถึง root ว่า unbalanced ? : { rebalance } : { เช็ค parent }

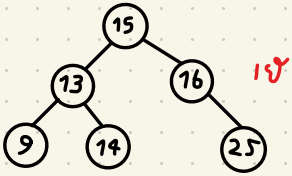
Ex.



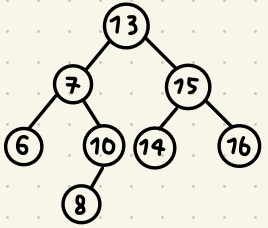
insert(25) ⇒



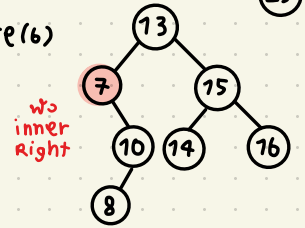
Single R. from Left ⇒



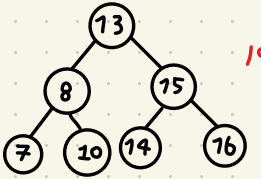
เช็ค!



delete(6) ⇒



Double R. from Right ⇒

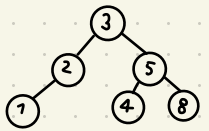


เช็ค!

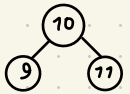
AVL Merge

- เช็คก่อนว่า mergeable มั้ย?
  - $max(R_1) < min(R_2)$
- $|R_1 - R_2| \leq 1$ merge เลย!

Ex

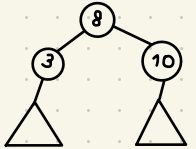


merge



⇒

เอา 8 มาเป็น root ใหม่



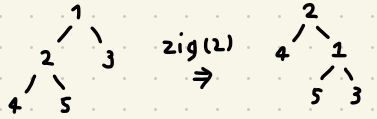
- else → ดูว่า R_1 หรือ R_2 สูงกว่า
  - R_1 สูงกว่า ⇒ จะหา R_1.right ที่ $|R.right - R_2| \leq 1$ มา merge ด้วย
  - R_2 สูงกว่า ⇒ จะหา R_2.left ที่ $|R_2.left - R_1| \leq 1$ มา merge ด้วย

Splay Tree

1985 By Sleator & Tarjan

- Node ในนั้นโดนอ่าน = ย้ายมาอยู่ข้างบน
- ไม่รับประกันความ balance

"การ zig" ~ rotation



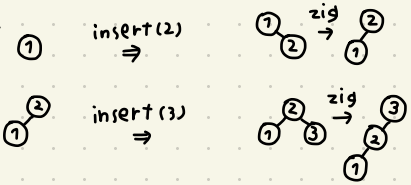
zigzig วงนอก
→ zig parent แล้ว zig r

zigzag วงใน
→ zig r แล้ว zig r

"Insert" เหมือน BST แล้ว splay มาที่ root

แต่ละครั้งใช้ O(1) n ครั้งก็ใช้ O(n)

Ex.

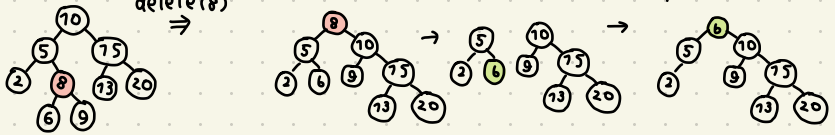


pro & cons.

+ รับประกัน log N เหมือน AVL
- ปรับข้อมูลบ่อย ทุกครั้งที่อ่าน

"Delete" splay มาที่ root → นักออก → splay max left มาเป็น root ใหม่

Ex.



B-Tree

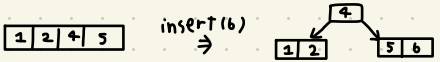
การันตี log N, balance ตลอดเวลา, hard-drive

- 1 Node มี key ได้หลาย key (min) M/2 ปัดขึ้น - 1
  (max) M-1
- 1 Node มีลูกได้หลายคน (min) M/2 ปัดลง
  (max) M
- Node ที่มี k keys ⇒ มีลูก k+1 คนเสมอ

ความลึก คือ $O(\log_{M/2} N)$

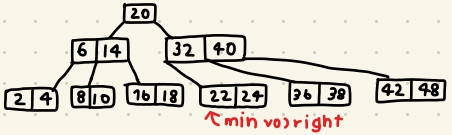
"Insert" เช็คว่า node เต็มมั้ย? → ไม่ : ใส่เลย
→ เต็ม : เอา node ตรงกลางมาเป็นแม่

Ex.

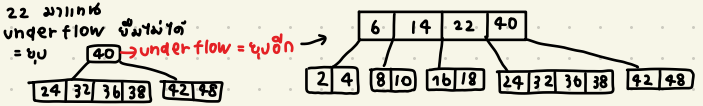


"Delete" key อยู่ในใบ → ลบ แล้วเช็คว่า underflow มั้ย?
key อยู่ที่ชั้นใน → เอา min ขวา right ที่ใบมาแทน แล้วเช็ค underflow

Ex.



delete(20)
→ เอา 22 มาแทน
→ underflow ยืมไม่ได้
= ยุบ



Priority Queue

Q. สามารถสร้าง priQ. ได้ด้วยอะไรบ้าง

Ans.

1. multiple queue
   - สร้าง queue ขึ้นมา m ตัว
   - แต่ละตัวจะผูกกับ priority แต่ละค่า
   - push obj. เข้าไปใน queue ที่ถูกต้องตาม priority
   - top, pop ก็จาก queue ที่มี prioty สูงก่อน

   ข้อดี : เขียนโค้ดง่าย ข้อเสีย: จน. priority มีจำกัด

2. AVL Tree
   - แต่ละ node เรียง priority เข้าไป
   - insert เข้าไปตาม priority ที่ควรเป็น
   - top, pop วิ่งซ้ายหาจุดที่น้อยสุด

   ข้อดี : เลย priority จะเป็นอะไรก็ได้ มากเท่าไหร่ก็ได้
   ข้อเสีย : ต้อง ปรับความสูงตลอดเวลา , ช้าไม่เป็น Big Theta(1)

3. Heap
   - Binary Tree ที่ root มีตัวเลข ≤ key ใน subtree ทั้งสอง โดยที่ subtree ก็เป็น Binary Min Heap ด้วย

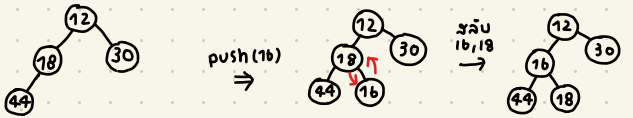
   ข้อดี : รู้เลยว่าใครออกก่อน Ⓗ(1)
   ข้อเสีย : อาจจะไม่ balance ถ้าไม่ทำให้เป็น AVL Tree

Binary Heap

complete tree
เติมจากบน → ล่าง, ซ้าย → ขวา

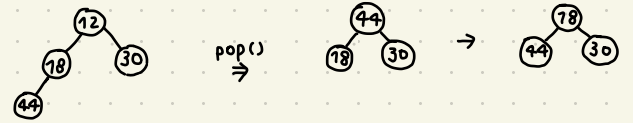
"Push" → ใส่แบบ complete tree
→ (min) swap ถ้า key ที่ใส่ < parent

Ex.



"Pop" → เอาตัวท้ายสุดมาขึ้นแทน root
→ swap ลงไป

Ex.



Using Array

- Travel แบบ BST
- เริ่มที่ index 1
- ลูกเขาอยู่ที่ 2K, 2K+1 , แม่เขาอยู่ที่ $\frac{K}{2}$ ปัดลง

Push = O(logN) , Pop = O(logN) , Top = O(1)

Merge = O(NlogN)

Stock Market using priority queue


```
market.submitSellOrder("Paint", "SCB", 5, 200);
market.submitSellOrder("Meen", "SCB", 10, 180);
```

จำนวน ราคา

10/180

5/200

```
market.submitBuyOrder("Gift", "SCB", 6, 250);
```

MATCH!!

4/180

5/200

```
market.submitBuyOrder("Kob", "SCB", 6, 250);
```

MATCH!!

3/200

Hash Table set, map

- ปัญหา hash table "chaining"
  → เอา obj. มาhash ได้ index เอาไปต่อใน List

$O(\overset{\text{จน.ข้อมูล}}{n}+m)$, ความเร็ว $O(\overset{\text{ยาวที่ยาวที่สุด}}{c}+1)$
↳ cardinality → hash แล้วมีค่าซ้ำกันน้อย

- Universal Family → อัตราการชน $\leq \frac{1}{m}$

$O(\alpha+1)$; $\alpha = \frac{n}{m}$
↳ ควรอยู่ประมาณ 0.5 - 1.0

Seperate Chaining

- array ของ Linked List
- ชน → จับต่อหลัง ไปเรื่อยๆ
- ควบคุม Load Factor (α) ไม่ได้ ($\alpha > 1$)

Open Address [Linear / Quadratic

- array เหยียบ
- ชน → ไปดูช่องอื่น
- Search O(1) เพราะควบคุม α ได้!

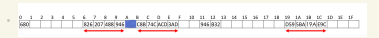
Linear Probing ชน → ไปดูช่องถัดไป

"Insert" hash หาindex แล้วเอาไปลง
- ชน ? ลงช่องถัดไปเรื่อยๆ

"Rehash" เมื่อ α เริ่มเยอะ ก็จะ
- เพิ่ม ขนาด array *2
- คิดการ hash ใหม่
- copy ข้อมูลเก่า → hash → เอาใส่ตารางใหม่

"Delete"
- ลบออกไป
- check จนกว่าจะเจอแจ้งว่า ข้อมูลนี้ hash แล้วได้หลายกว่า index ที่ลบมั้ย?
  ใช่ → ดึงกลับ , ไม่ใช่ → ไปต่อ

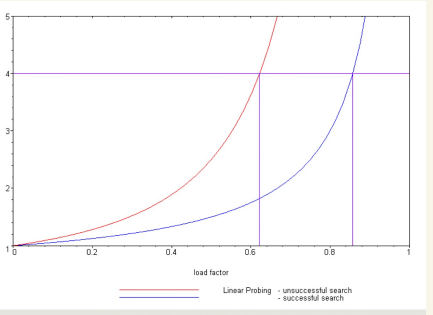
การชน collision
ข้อมูล ↑ การชน ↑



ค่าง่ายเป็นการลงช่องฟ้า = 5/32

"เกิดแพใหญ่"

Run-time Analysis



ชนก็ลองกว่าจะเจอ
- $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$ หาเจอ
- $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$ หาไม่เจอ

แกน x คือ α แกน y คือ collision

Load factor ↑ collision ↑
↳ ถ้า $\alpha > 1$ การชนจะวิ่งเข้าใกล้ ∞

ควบคุมได้! - ควรให้ m มีค่ามาก แต่ไม่เยอะจน mem จนเต็มนาน
- ควบคุม Load factor ด้วยการ rehash
- หรือ Quadratic Probing!

Quadratic Probing

→ ชน แล้ว ไปดู ช่องที่ bin + k

"Insert" hash หา index แล้วเอาไปลง

- ชน ? ลงช่องที่ (bin + k)%M

Ex. hash แล้วได้ 5 ชน → 5+1 = 6 ชน
→ 6+2 = 8 ชน
→ 8+3 = 11 ชน
→ 11+4 = 15 ชน
→ 15+5 = 20 %16 = 4 ชน
⋮

and it will visit every index

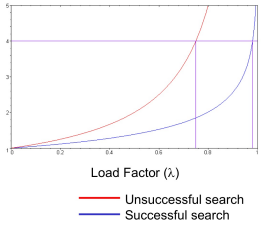
ขนาดตาราง 2^x เท่านั้น!

"Rehash" เหมือนเดิม

"Delete" Lazy Delete!

- คือเพิ่ม isDeleted ลงไป จะ mark ไว้ว่า true
- เหมือนไม่อยู่ แต่อยู่ แค่หาไม่เจอ

Run-time Analysis

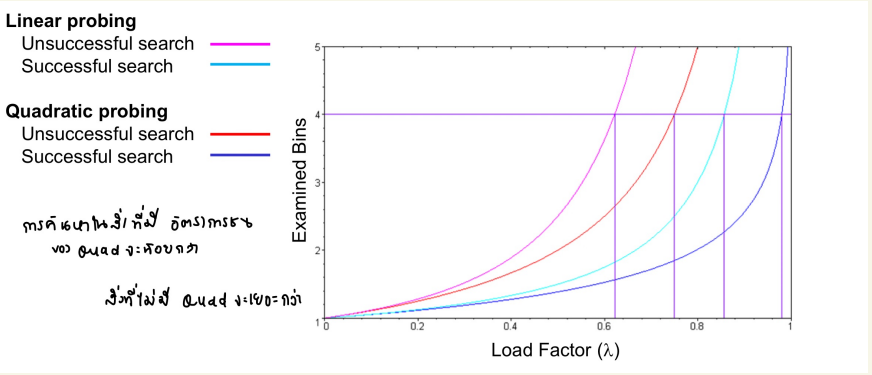


- $\frac{1}{1-\lambda}$ หาไม่เจอ
- $\ln(\frac{1}{1-\lambda})/\lambda$ หาเจอ

แกน x คือ α แกน y คือ collision

Load factor ↑ collision ↑
↳ ถ้า $\alpha > 1$ การชนจะวิ่งเข้าใกล้ ∞

เปรียบเทียบ collision



- ที่ Load Factor ที่มีค่าเดียวกัน
  อัตราการชนใน successful search ของ Quad จะน้อยกว่า
  ในขณะที่ อัตราการชนใน unsuccesful search ของ Quad จะเยอะกว่า

cache (ใน CPU) – สร้างโดย quad. probing

Graph

tree คือ subset ของกราฟ

- มี cycle → graph, ไม่มี cycle → tree

vertex จุด edge เส้น

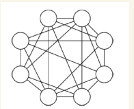
Graph Representation
- Edge List
- Adjacency Matrix
- Adjacency List *

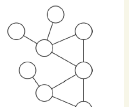
Run-time Analysis

Op	hasEdge(V1, V2)	find all Neighbors(V)	Memory
Edge List	O(|E|)	O(|E|)	O(|E|)
Adj Matrix	O(1)	O(|V|)	O(|V|^2)
Adj. List	O(|V|)	O(|V|)	O(|V|+|E|)
Adj. List (if deg << |V|)	O(deg)	O(deg)	O(|V|+|E|)

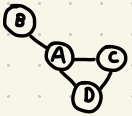
ความเร็วของกราฟ depends on V, E

"Density" ความหนาแน่น



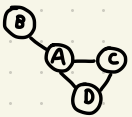


Edge List



Edge :
(A,B), (A,C), (A,D), (C,D)

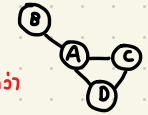
Adjacency Matrix



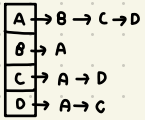
	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	1	0	0	1
D	1	0	1	0

Adjacency List

ใช้ mem เยอะกว่า



A	Ad to	B, C, D
B	〃	A
C	〃	A, D
D	〃	A, C



DFS

- set visited เป็น false → loop ทุก v
- จน. neighbor O(|E|)
- Runtime O(|V|+|E|)

BFS

- ค้นหา neighbor ทีละชั้นก่อน
- Length = จน. edge
- Distance = Length ที่สั้นที่สุด
  (∞ = ไม่มี, 0 = วนกลับมาที่ตัวเอง)
- Shortest Path O(|V|+|E|)