## Trees

นิยาม กลุ่มของ nodes ซึ่งอาจจะว่าง (empty) มี node พิเศษเรียกว่า root เป็น node เริ่มต้น จากนิยามข้างต้น tree ที่มี n nodes จะประกอบด้วย n-1 edges

นิยาม path จาก node $n_1$ ถึง $n_k$ คือ sequence ของ nodes $n_1, n_2, \ldots, n_k$ โดยที่ $n_i$ เป็น parent ของ $n_{i+1}$, $1 <= i < k$

นิยาม ความยาว (length) ของ path หนึ่งๆ คือ จำนวน edges บน path นั้น ซึ่งก็คือ k-1 (จากนิยามของ path)

นิยาม ความลึก (depth) ของ node $n_i$ คือ ความยาว (length) ของ unique path จาก root ถึง node $n_i$ นั้น

นิยาม ความสูง (height) ของ $n_i$ คือ ความยาวที่สุด (longest path) จาก $n_i$ ถึงปลาย (leaf)

นิยามง่าย ความลึกของ $n_i$ วัด (หรือนับ) จาก root ถึง $n_i$

นิยามง่าย ความสูงของ $n_i$ วัด (หรือนับ) จาก $n_i$ ถึง leaf

## Binary trees

นิยาม เป็น tree ที่ทุก node มีได้ไม่เกิน 2 children

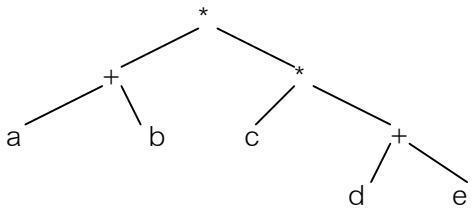
มีความลึกสูงสุด N – 1 (สำหรับ binary tree ที่มี N nodes)

ความลึกเฉลี่ย $O(N^{0.5})$ ในกรณีของ binary search tree คือ O(log N)

สำหรับ complete binary tree ที่มีความสูง h จะมี nodes ทั้งสิ้น $2^{h+1} – 1$

binary search tree เป็น binary tree ที่มีลำดับสอดคล้องตามความสัมพันธ์ left < root < right และมีการประยุกต์ใช้งานอย่างกว้างขวาง เช่น การแทน expression ด้วย tree (ดังตัวอย่างในหน้า 97-100) แต่จุดแข็งที่โดดเด่นที่สุด น่าจะเป็นการเก็บข้อมูลที่มีการเรียงลำดับอย่างถูกต้องตลอดเวลา ซึ่งทำให้ประหยัดเวลาในการ sort ข้อมูล (อันจะกล่าวในบทต่อไป) หลักการที่ใช้ก็คือ tree traversal สามแบบ ได้แก่ pre-order, in-order, และ post-order สำหรับข้อมูลทั่วไป in-order จะเหมาะสมที่สุดในการใช้งาน แต่สำหรับ expression tree แล้ว ทั้งสามแบบจะแสดงนิพจน์ในลักษณะ prefix, infix, และ postfix ซึ่งทำให้การเก็บโดยใช้ tree เป็นโครงสร้างข้อมูลที่สมบูรณ์แบบที่สุด (ตัวอย่าง directory listing หน้า 92-95 ก็เป็นอีกตัวอย่างของการประยุกต์ tree)

ตัวอย่างที่ 1 จงสร้าง expression tree จากนิพจน์ที่กำหนดให้ (a + b) * (c * (d + e))
เริ่มจากซ้ายสุด (ตามตัวอย่าง) จะได้ expression tree ดังรูป




ความสะดวกของ expression tree คือ การแทนนิพจน์ได้ทั้ง 3 รูปแบบ คือ infix, prefix, และ postfix notation ซึ่งก็สอดคล้องกับการเดินตาม tree ในลักษณะ in-order, pre-order และ post-order ตามลำดับ จากตัวอย่างข้างต้น เราจะได้ผลลัพธ์ของ tree traversal ดังนี้

| | | |
|---|---|---|
| in-order: | (a + b) * (c * (d + e)) | (left-root-right) |
| pre-order: | * + a b * c + d e | (root-left-right) |
| post-order: | a b + c d e + * * | (left-right-root) |

การวิเคราะห์ประสิทธิภาพของ binary search tree อาศัยหลัก recurrence เพราะ tree จะอยู่ในรูปที่สร้างตามนิยามวนซ้ำ (recursive definition) กล่าวคือ tree ประกอบด้วย root และ left/right subtree ซึ่งแต่ละ subtree ก็ใช้นิยามเดิม คือมี root ของ subtree นั้นและ sub- ของ left/right subtree อีก เช่นนี้เรื่อยไปจนถึง leaf node

ถ้าให้ D(n) เป็น internal path length ของ tree T ที่มี n nodes จะได้ว่า D(1) = 0 สำหรับ left subtree ขนาด i-node ขนาดของ right subtree จะเป็น (n-i-1) การคำนวณ internal path length D (n) หาได้จาก

$$D(n) = D(i) + D(n-i-1) + n - 1$$

ซึ่งก็คือ internal path length ของ T วัดจาก root คำนวนได้จาก internal path length ของ left subtree ขนาด i-node ข้างต้น บวกกับ internal path length ของ right subtree และ internal path length ของ T ไม่นับ root

<u>ตัวอย่างที่ 2</u> สมมุติ (complete) binary search tree T ขนาด 7 node ค่าของ internal path length จะเป็น $D(7) = D(3) + D(3) + 7 - 1 = [D(1) + D(1) + 3 - 1] + [D(1) + D(1) + 3 - 1] + 7 - 1 = 10$ (ลองเขียนรูปแล้วนับดู ที่ความลึก 1 มี 2 nodes รวม path length เป็น 2 ที่ความลึก 2 มี 4 nodes ซึ่งแต่ละ node ค่า path length เป็น 2)

## AVL (Adelson-Velskii and Landis) trees

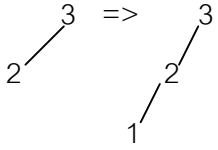
เป็น binary search tree ที่สมดุลย์ (balance) มีความลึกเป็น O(log n) ดังนั้นสำหรับทุกๆ โหนดใน AVL tree ความสูงของ left และ right subtrees จะต่างกันเพียง 1 (โดยกำหนดว่า ความสูงของ empty tree เป็น -1)

การ balance AVL tree สำหรับโหนด $\alpha$ ใดๆ แบ่งได้เป็น 4 กรณีคือ

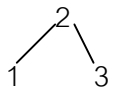
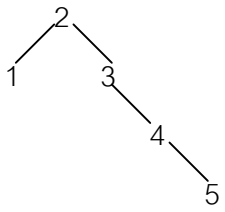
1. left child ของ $\alpha$ ใน left subtree (เกิดด้านนอก เรียกว่า L-L เป็นแบบ single rotation)
2. left child ของ $\alpha$ ใน right subtree (เกิดด้านใน เรียกว่า L-R เป็นแบบ double rotation)
3. right child ของ $\alpha$ ใน left subtree (เกิดด้านใน เรียกว่า R-L เป็นแบบ double rotation)
4. right child ของ $\alpha$ ใน right subtree (เกิดด้านนอก เรียกว่า R-R เป็นแบบ single rotation)

(ดูรูปแบบของ single และ double rotation ใน PowerPoint)

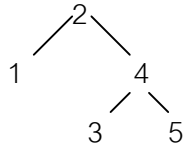
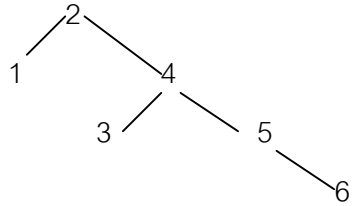
<u>ตัวอย่างที่ 3</u> กำหนดข้อมูลดังต่อไปนี้ 3, 2, 1, 4, 5, 6, 7 ให้สร้าง AVL tree สำหรับข้อมูลชุดนี้

3 =>  =>  ณ จุดนี้เกิดความไม่สมดุลย์ ใช้ single rotation

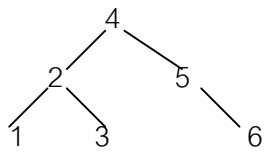
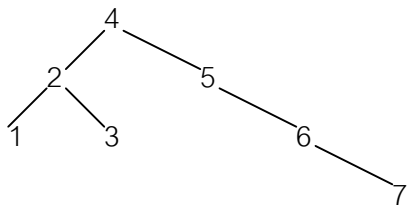


=>  =>  => เกิดไม่สมดุลย์ ต้องจัดใหม่ด้วย single rotation

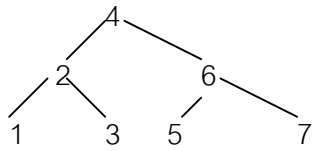
พิจารณา subtree 3-4-5 สามารถ rotate ครั้งเดียวก็จะได้รูปใหม่ดังนี้

=>  => 

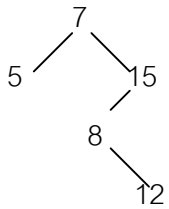
ต้องจัดใหม่ด้วย single rotation คราวนี้พิจารณาทั้ง tree ที่โหนด 2 ซึ่งเป็น root และ 4 ซึ่งเป็น root ของ right subtree

=>  => 

จะเห็นว่า right subtree ไม่สมดุลย์ หลังจากปรับ right subtree ด้วย single rotation ก็จะได้รูปใหม่เป็น

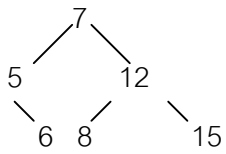


<u>ตัวอย่างที่ 4</u> กำหนดข้อมูลดังต่อไปนี้ 7, 15. 5, 8, 12, 6 ให้สร้าง AVL tree สำหรับข้อมูลชุดนี้



=> เมื่อเพิ่ม 12 เข้าไปใน tree ทำให้ right subtree เสียสมดุลย์ ซึ่งในกรณีนี้ single rotation ไม่สามารถจัดการได้ ต้องใช้ L-R double rotation (คือ โย้ซ้ายหมุนขวาแก้) โดยให้ 15 เป็น $k_3$ 8 เป็น $k_1$ และ 12 เป็น $k_2$ ส่วน subtree A, B, C, D ถือเป็น NULL ทั้งหมด

รูปใหม่ที่ได้จะเป็น เมื่อเพิ่ม 6 ก็ยังคงสมดุลย์เหมือนเดิม



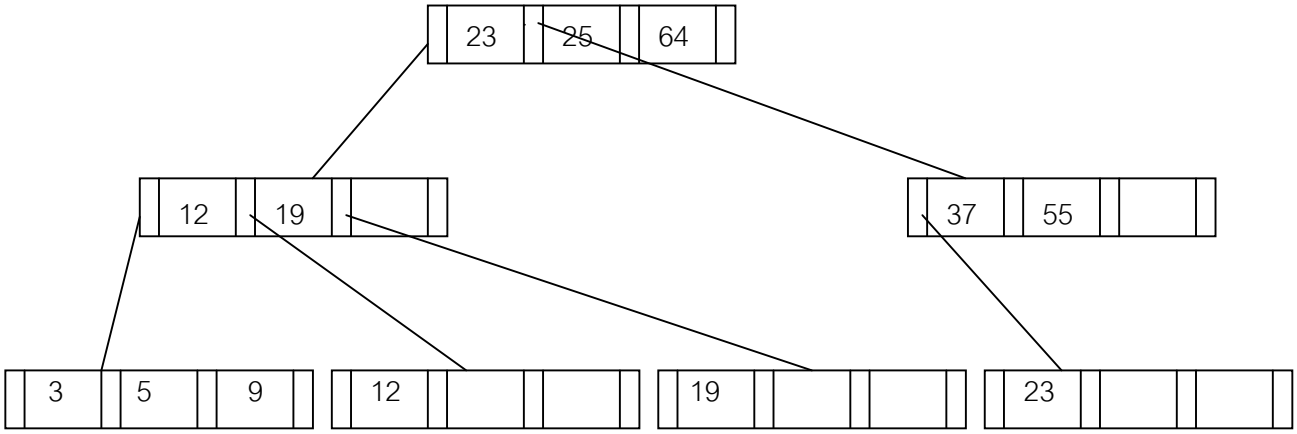
เวลาที่ใช้ในการเพิ่มข้อมูลแต่ละตัวเข้าใน tree ก็ไม่เกิน O(log n) ส่วนเวลาที่ใช้ในการทำ single และ double rotation (แทนด้วย $T_L$, $T_R$, $T_{LR}$, $T_{RL}$) ก็ขึ้นอยู่กับขนาดของ subtree ซึ่งสามารถคำนวณได้จาก $n = 2^{h+1} - 1$ โดย h เป็นความสูงของ subtree ที่พิจารณา

## B-Trees

B-tree ขนาด (order) m มีลักษณะโครงสร้างดังต่อไปนี้

- root จะเป็น leaf หรือมีโหนดลูกได้ระหว่าง 2 ถึง m
- nonleaf nodes (หรือเรียกว่า interior nodes) ทั้งหมดจะมีโหนดลูกได้ระหว่าง floor(m/2) ถึง m
- leaf nodes ทั้งหมดอยู่ที่ระดับความลึกเดียวกัน
- ข้อมูลจะเก็บไว้ใน leaves เท่านั้น

ตัวอย่างที่ 5 สร้าง B-tree ขนาด 4




หลักการใส่ key และข้อมูลลงใน interior nodes และ leaves มีดังนี้

เริ่มใส่ค่าลงในแต่ละ leaf node เรื่อยๆ จนเมื่อเต็มโหนดก็จะ split ออกเป็น 2 โหนด โดยยกค่าแรกของ leaf node หลังขึ้นเป็น key ตัวแรกของ interior node ในระดับที่สูงถัดไป ในกรณีนี้ คือ 12 ทำเช่นนี้กับระดับที่สูงขึ้นไปเรื่อยๆ สังเกตว่า ค่า pointer ทางซ้ายของ key ตัวแรกใน root ชี้ไปยัง “กลุ่ม” ของโหนดลูกที่น้อยกว่า key (ถ้าจะเปรียบเทียบกับ binary tree ก็เสมือน left subtree < key) pointer ทางขวาก็จะมากกว่า key (แต่ก็จะมีค่าน้อยกว่า key ตัวถัดไป คือ 19) ซึ่งถ้าพิจารณา interior node ในระดับต่ำลงมา 1 ระดับ จะเห็นว่า 23 น้อยกว่า 37 จึงเป็นกลุ่มโหนดลูกซึ่งถูกชี้โดย pointer ด้านซ้ายของ 37 (เสมือน left subtree เช่นกัน) การ split ของโหนดทั้งที่เป็น leaf และ interior nodes จะเกิดขึ้นในลักษณะดังกล่าวเรื่อยไปจนกว่าแต่ละโหนดจะเต็ม (ซึ่งหมายถึงแต่ละโหนดมีค่าเต็มทุกช่อง รวมทั้ง pointer ที่ชี้ไปยังโหนดอื่นด้วย) นั่นหมายความว่า ที่ root ของ B-tree มีโหนดลูกได้ถึง 4 โหนด (และ 3 keys) แต่ละ interior node มีโหนดลูก 4 โหนด (และ 3 keys) เช่นกัน สำหรับ B-tree ขนาด 4 ที่มีความสูง 2 จะมีโหนดมากที่สุดถึง 21 โหนด และบรรจุข้อมูลได้ทั้งสิ้น 48 ตัว ในขณะที่ complete binary search tree จะมีความสูงถึง floor(log 48) = 5 ดังนั้น ความได้เปรียบในการสืบค้นที่รวดเร็ว (เนื่องจากจำนวนการเปรียบเทียบค่าของ key ในแต่ละระดับคือ 1 ครั้งเท่ากันใน tree ทั้งสองชนิด) ของ B-tree จึงทำให้ได้รับความนิยมในการประยุกต์ใช้งานอย่างกว้างขวาง

เป็นที่น่าสังเกตว่า B-tree จะไม่สูงมากเหมือน binary tree แม้ว่าใน worst case ก็ตาม (worst case ของ binary tree คือ n -1) เพราะข้อกำหนดหนึ่งของโครงสร้าง คือ แต่ละ interior node จะมีโหนดลูกระหว่าง floor(m/2) ถึง m โหนด ทำให้ B-tree ไม่มีสภาพสูงเอียงไปด้านใดด้านหนึ่งเช่นเดียวกับ binary tree เพราะข้อมูลจะกระจายไปยังโหนดในระดับเดียวกัน (ซึ่งจะมีจำนวนมากกว่า binary tree )

ก่อนที่จะขยับสูงขึ้นสู่ระดับต่อไป นั่นหมายถึง B-tree มักเริ่มต้นจาก “ล่าง-ขึ้น-บน” (bottom-up) แทนที่จะเป็นแบบ “บน-ลง-ล่าง” เช่น binary tree

## Implementation

ความเหมาะสมของ self-reference structures ในลักษณะที่เป็น dynamic memory allocation เป็นข้อได้เปรียบในการประยุกต์ใช้งานของ trees ทุกชนิด แต่สำหรับงานขนาดเล็กที่ใช้ binary tree เป็นโครงสร้างหลักนั้น อาจใช้ array แทน self-reference structure ได้ แต่มีข้อแม้ว่า binary tree ต้องมีขนาดจำกัด (ถึงแม้ว่า array จะสามารถ implement ในลักษณะ dynamic ก็จริง แต่ก็มีความไม่สะดวกบางประการในการรองรับ binary tree ที่มีขนาดไม่จำกัด)

## Applications

Trees เป็น nonlinear data structure ที่มีประโยชน์ใช้สอยมาก โดยเฉพาะ B-tree ที่มักใช้ในการทำดรรชนี (indexing) ของระบบแฟ้มข้อมูลและฐานข้อมูล ระบบ directory systems ประเภทต่างๆ รวมทั้งการแทนนิพจน์ทางคณิตศาสตร์ เป็นต้น