TUTORIAL 5

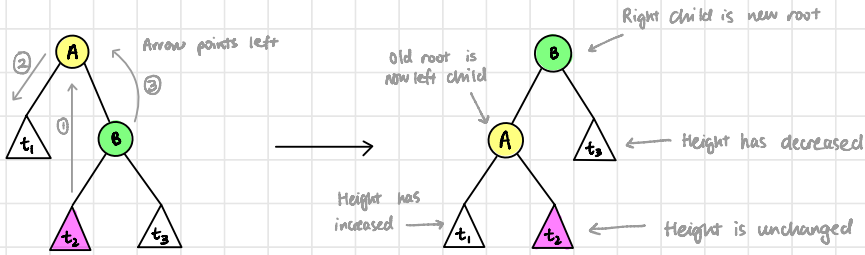
TREE ROTATIONS

Recall: Smaller tree heights ⇒ faster searches & insertions. So how can we try and control a tree's height as it grows?

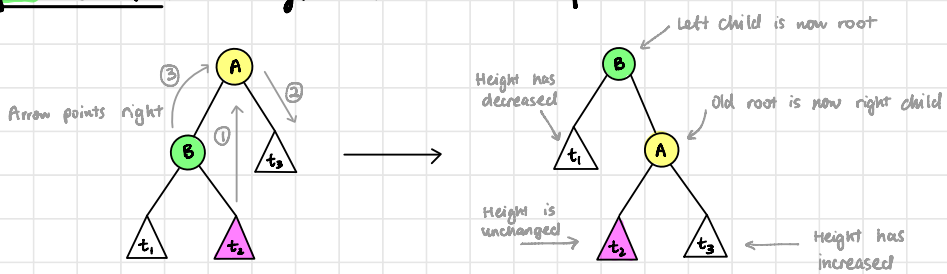
Solution: Tree rotations!

Rotation: A shape-changing operation which lowers the root one level and raises one of its children up one level to replace it. This lets us decrease overall tree height by moving tall subtrees up, short ones down.

Left rotation: Brings the right subtree up:

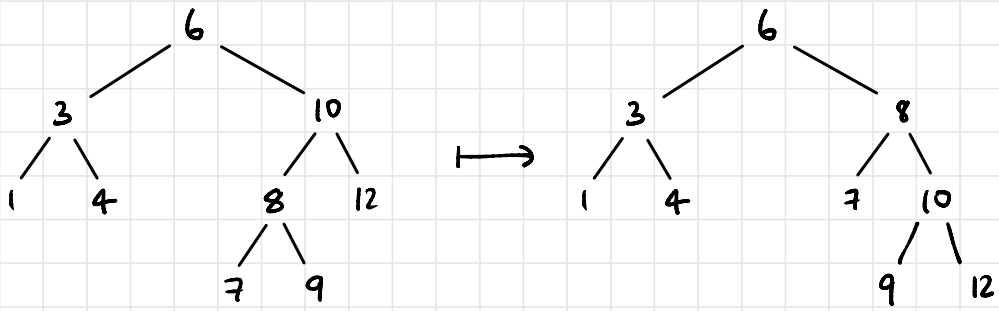


Right rotation: Brings left subtree up:

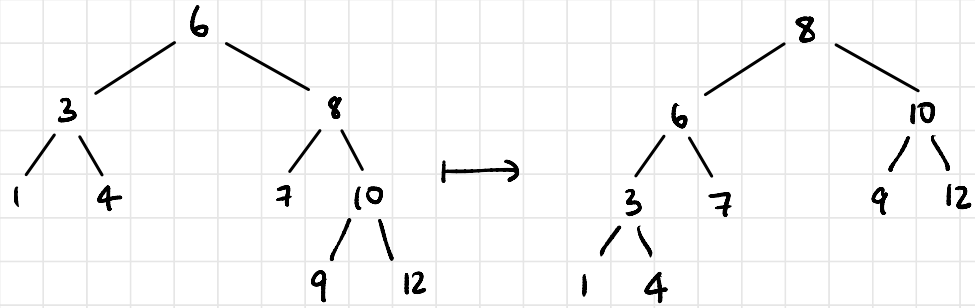


Implementation: Swap some pointers around ⇒ O(1) time complexity

1. Rotate right @ 10:



Rotate left @ 6:



AVL TREES: A SELF-BALANCING BST

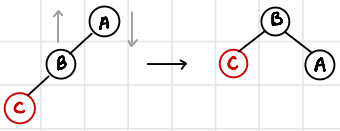
Goal: Have our tree automatically keep itself height-balanced.

Idea: Use rotations to correct any height imbalances caused during each insertion. But how many rotations do we need?

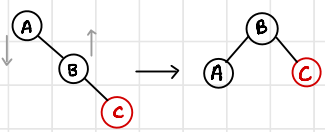
Fact: If the tree was balanced prior to inserting, we actually need at most 2 rotations to rebalance it!

We only need to consider checking the balance of the subtrees rooted at the inserted node's grandparents. There are 4 cases:

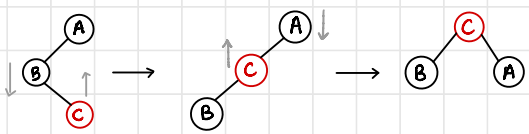
Left-left: R(A)



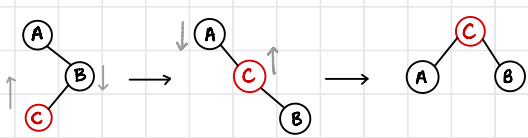
Right-right: L(A)



Left-right: L(B) + R(A)



Right-left: R(B) + L(A)



AVL Tree: Basically do the aforementioned per insert:

1. Insert value like normal.
2. Check subtree balance along the path you went down: is $|\text{left height} - \text{right height}| \leq 1$ for each subtree?
3. If not, then consider the 4 cases outlined above.

Complexity: If you have heights precomputed (i.e. each node contains its current height), then guaranteed $O(\log n)$. If you have to compute heights manually, then $O(n \log n)$.

To do: walk through simulation of inserts via web.

GRAPHS

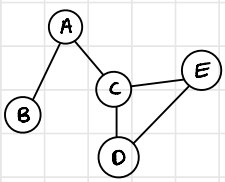
So far: arrays, linked lists, BSTs

- arrays: no inherent relation b/w items
- linked lists: each node is related to at most 1 other.
- BSTs: data in the tree is ordered, also each node is related to at most 2 others.

Problem: What if the data
* isn't ordered?
* can be related to >2 other pieces of data?

We need a new data structure ⇒ graphs!

Graph: a set of vertices/nodes and edges b/w vertices:

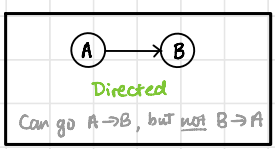


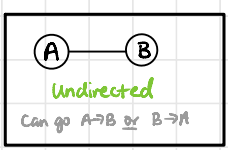
→ is the graph

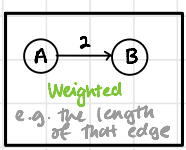
$V = \{A, B, C, D, E\}$

$E = \{(A,B), (A,C), (C,D), (C,E), (D,E)\}$

Edge: Can come in a variety of types:







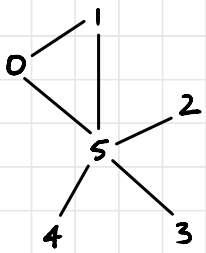
Representations: We usually represent vertices with (positive) integers, but how do we represent the edge set?

1. Array of edges: i.e. an array where each element is an edge
2. Adjacency matrix: 2D matrix A where A_{ij} = is there an edge $i \to j$?
3. Adjacency list: array with one slot per vertex, and each slot is a linked list of nodes for which there is an edge to

Tradeoffs: Edge lists suck, adj. matrix is fast but is memory inefficient, adj. lists are a bit slower but more memory efficient.

2. Show representations of these graphs:

(a)



Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	0	0	0	1
1	1	0				
2	0		0			
3	0			0		
4	0				0	
5	1					0

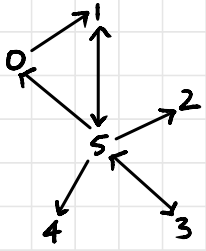
etc.

Adjacency list:

0 → 1 → 5 → X
1 → 0 → 5 → X
2 → 5
3
4
5

etc.

(b)



Adjacency Matrix:

	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0				
2	0		0			
3	0			0		
4	0				0	
5	1					0

etc.

Adjacency list:

0 → 1 → X
1 → 5 → X
2 → X
3
4
5

etc.