

Foundations

🕒 Created	@January 8, 2025 9:18 AM
📁 Class	DS 4300
☑ Reviewed	<input type="checkbox"/>

Searching

- Most common operation
- In SQL, the SELECT is most versatile / complex
- Baseline for efficiency is Linear Search: $\theta(n)$
- Record - a collection of values for attributes - row
- Collections - a set of records of the same entity type, a table
- Search Key - A value for an attribute from the entity type
- Contiguously Allocated List
 - All $n \times x$ bytes are allocated as a single chunk of memory
- Linked List
 - Each record needs x bytes + additional space for 1 or 2 memory addresses
 - Individual records are linked together in a type of chain using memory addresses
- Arrays are faster for random access, but slow for insertion anywhere besides end
- Linked Lists are faster for inserting anywhere in the list, but slower for random access
- Arrays
 - fast random access
 - slow for random insertions

- Linked Lists
 - slow for random access
 - fast for random insertions
- Binary Search is faster: $\theta(\log n)$
- Database Searching:
 - Searching for specific id = fast
 - Searching for specific specialVal needs linear search of that column
 - Can't store data on disk sorted by both id and specialVal (at the same time)
 - data duplicated, inefficient space
 - Options:
 1. An array of tuples (specialVal, rowNum) sorted by specialVal
 - We could use Binary Search to quickly located specialVal
 - But every insert into the table would be slow
 2. A linked list of tuples (specialVal, rowNum) sorted by specialVal
 - a. searching for specialVal would be slow - linear scan required
 - b. Inserting would be faster

Binary Search Tree

BST

- Binary Search tree - fast insert and fast search
 - Creating / inserting into BST
 - EX: 23, 17, 20, 42, 31, 50

```

      23
     / \
    17  42
  
```

\ / \
20 31 50

- Always a reference to a root node
- Tree Traversals:
 - Pre-Order
 - Post Order
 - In Order
 - Level Order: BFS search
 - $23 \rightarrow 17, 42 \rightarrow 20, 31, 50$
 - Python doesn't have queue: use deque
- Want to minimize number of levels

For Homework:

```
class BinaryTreeNode(self, value, left=None, right=None)
    value int
    left BinaryTreeNode
    right BinaryTreeNode

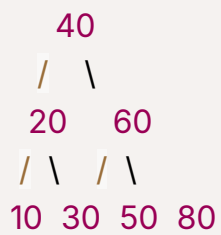
''' creating 23
root = BinaryTreeNode(23)
root.left = BinaryTreeNode(17)
root.right = BinaryTreeNode(42)
```

AVL Tree

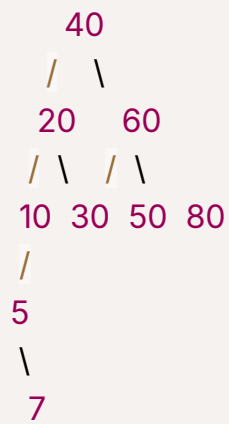
- Approximately balanced binary search tree
- AVL Property Balancing by $|h(LST) - h(RST)| \leq 1$
- Self balancing



Insert 50



Insert 5 then 7



Unbalanced: node of imbalance is 5 - 7 (denoted by alpha)

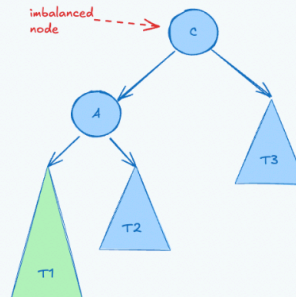
To balance:



▼ Four Cases of Imbalance:

1. Left Left Insertion (LL)

Case 1: Left-Left Imbalance (Single Rotation)



Then, there is an insertion somewhere in T1 that causes A's height to increase by 1. Now, height(C.left) and height(C.right) differ by 2.

Important Observations:

- All values in T1 are smaller than both A and C.
- All values in T2 are bigger than A but smaller than C
- All values in T3 are bigger than C.

→ Since all values in T2 are bigger than A but smaller than C, that subtree could live to the right of A or the left of C

```

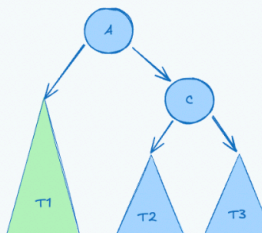
fun rotateWithLeftChild(C, parentOfC):
  A = C.left
  C.left = A.right
  A.right = C

  if C == root of tree:
    root of tree = A
  else:
    if parentOfC.left = C:
      parentOfC.left = A
    else:
      parentOfC.right = A

  updateHeight(C)
  updateHeight(A)
  
```

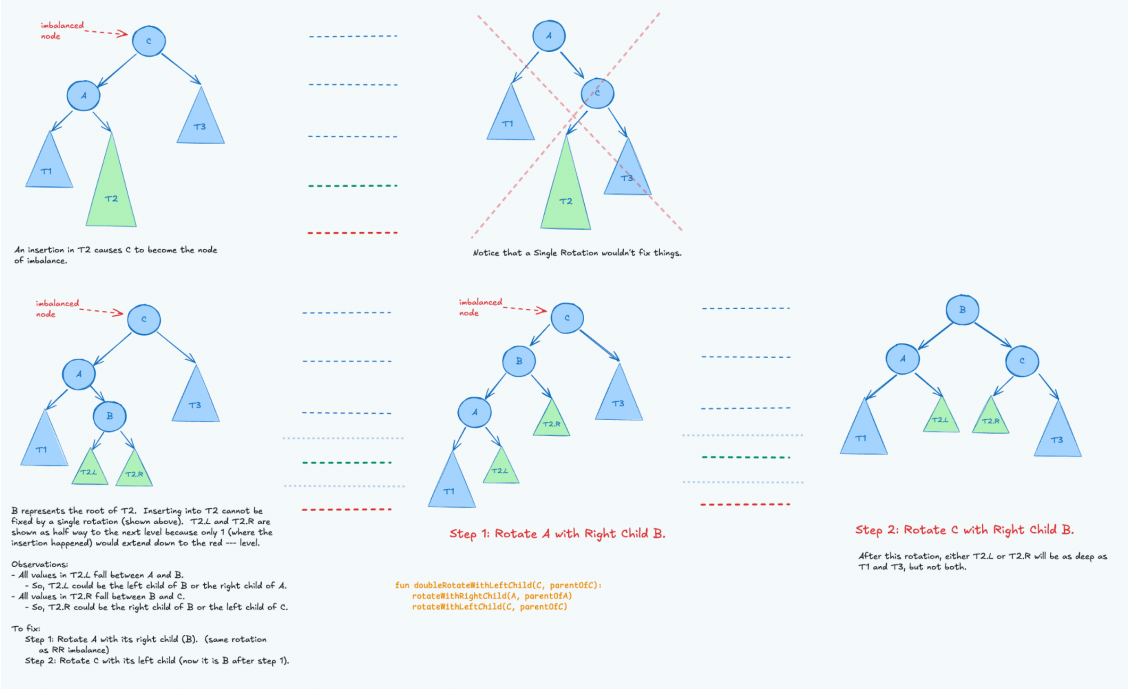
We can adjust for this imbalance with a *SINGLE ROTATION* by rotating the node of imbalance with its left child.

Notice after the re-balancing, A has the same height as C did in the original tree.



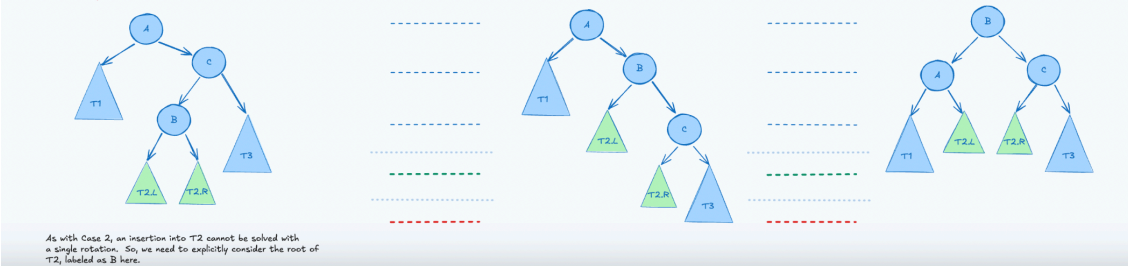
2. Left Right Insertion (LR)

Case 2: Left-Right Imbalance (Double Rotation)



3. Right Left Insertion (RL)

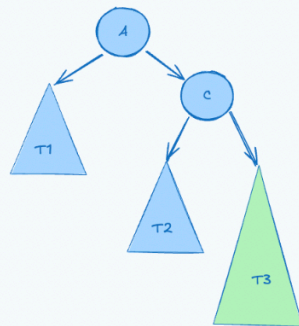
Case 3: Right Left Imbalance (Double Rotation)



- Rotate B with C, rotate A with B

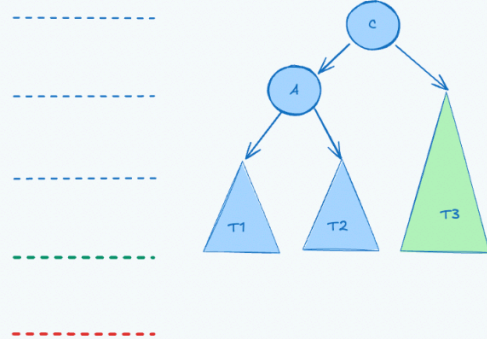
4. Right Right Insertion (RR)

Case 4: Right Right Imbalance (Single Rotation)



Insertion into T3 causes it to be 2 levels deeper than T1, making A the node of imbalance.

Remember: All values in T2 fall between A and C. So T2 could be connected to the left of C or the Right of A.



- Reassign c.left to A, then a.right to T2
- Case 1 & 4 are mirrors; 2 & 3 are also mirrors
-