Discussion 5: Balanced Trees

TIPs Session #2

Tech Interview & Internship TA Panel - - - -

Hear TAs share their experiences with interviews and internships and ask questions

February 9th @ 6PM

https://ufl.zoom.us/j/98850029166

Reminder that if you didn't elect to replace HTG with TIPs on Aman's spreadsheet - you are now **required** to do the HTG quiz

https://ufl.zoom.us/j/98850029166

Objectives

1. Project 1 Overview
2. Balanced Trees
    a. Balanced Binary Search Tree (BBST)
    b. AVL Trees
    c. Red-black trees
    d. Splay Trees
    e. B/B+ Trees
3. Discussion Participation Activity

Project 1: Gator AVL Tree

Implement a custom AVL tree to organize UF student accounts based on GatorIDs. Methods that will be called in the main method include:

1. insertion
2. remove
3. search
4. traversals (preorder, inorder, postorder)

You'll also need to implement rotation methods to fix unbalanced nodes

- You will only be expected to call rotations after insertions
- We **won't** ask you to rebalance on deletion, **only on** insertion!

Project 1: Gator AVL Tree

The following steps are required:

1. Designing the interface/modules/functions
2. Parsing the input and ensuring the constraints are met.
3. Building the main function to parse the inputs
4. Calling the respective functions to match the output
5. Testing your code within the constraints.

Project 1 - Getting Started

- A few TAs have made a presentation on getting started with the project
  - Here are the slides for the presentation
    - https://docs.google.com/presentation/d/1llw2u6PmbgUjE1nQM_Qyc4lVMeb84_UVINRpQvrycqE/edit#slide=id.p
  - Here is the YouTube video link
    - https://youtu.be/BjH_Pdu_2W4
  - Catch2 Tutorials
    - https://www.youtube.com/watch?v=DqE3UpOdBLw (CLion)
    - https://youtu.be/yj8baGjXmTU (VS Code)
  - Catch2 Template
    - https://github.com/COP3530/P1-Catch-Template
  - More information and resources on the canvas page!!

Balanced Trees

Wait -- why balanced trees?

Binary search trees typically have an average time complexity of O(log(n)) for searching, insertion, and deletion.

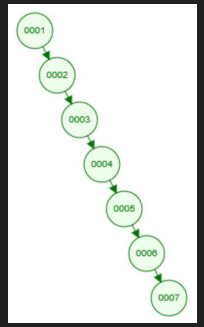
The worst case is O(n), which occurs in unbalanced trees as we iterate through a bunch of nodes

Operation	**Average Time Complexity**	**Worst Time Complexity**
Searching	O (log(n))	O (n)
Insertion	O (log(n))	O (n)
Deletion	O (log(n))	O (n)

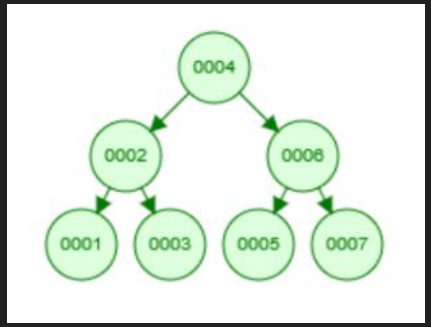
What is balance?

- The balance refers to the height of the left subtree - the height of the right subtree.
- The calculated balance value is called the balance factor:
  - Take a node
  - Get the height of both its left & right subtrees
  - Subtract the height of one subtree from another
    - You can do (height of Left subtree) - (height of Right subtree) or vice versa

FOR REFERENCE These slides will calculate height as [Left - Right]



Node 1 has a balance factor of [0 - 6 = -6]



Node 4 has a balance factor of [2 -2 = 0]

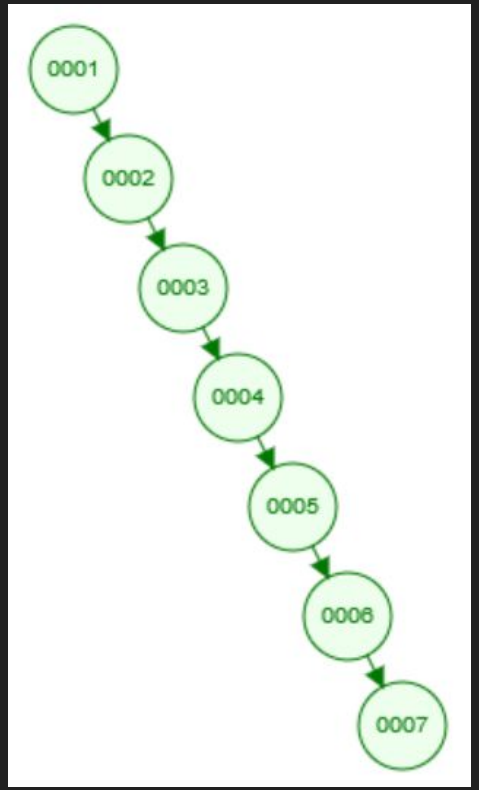
Insert Balanced Binary Search Trees (BBST)

Every node in a BBST has a balance factor that’s either -1, 0, or 1

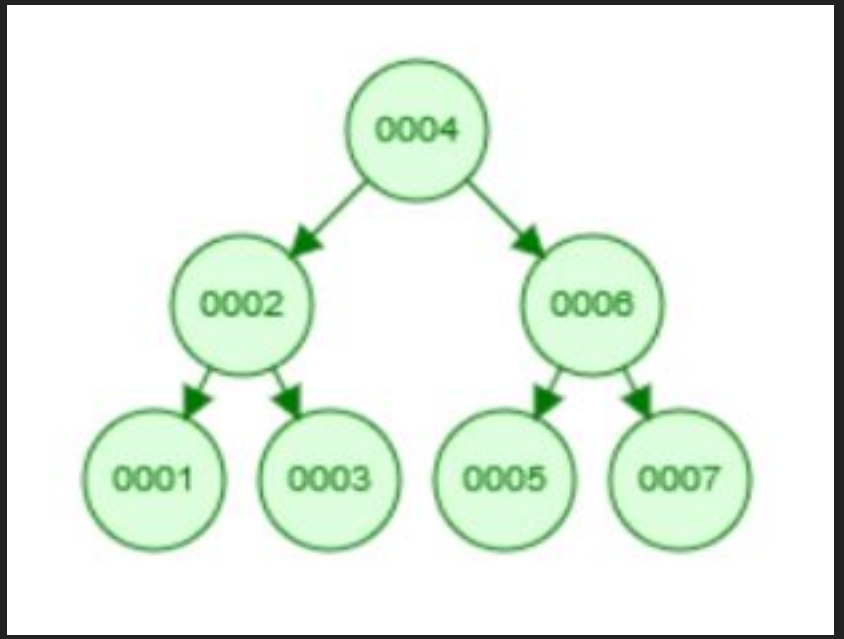
- Balance factor is the difference between the heights of the left & right subtrees

This ensures you always travel the minimum amount of nodes to get to a target, ensuring logarithmic growth for the worst case.

Operation	**Average Time Complexity**	**Worst Time Complexity**
Searching	O (log n)	O (log n)
Insertion	O (log n)	O (log n)
Deletion	O (log n)	O (log n)



Unbalanced Tree



Balanced Tree

Worst Case: Iterate through all 7 nodes

Worst Case: Iterate through 3 nodes

Rotations

Rotations move around nodes in a binary search tree without violating its properties.

This makes the tree balanced, thus, making operations on it more efficient.

There are 4 types of rotations, each based on the imbalance & alignment of the nodes:

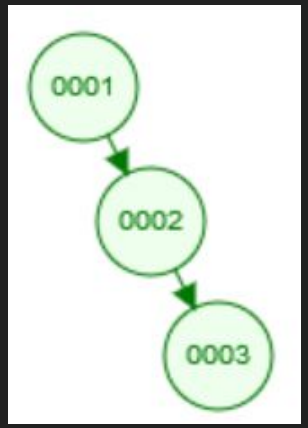
1. Left
2. Right
3. Left-Right
4. Right-Left

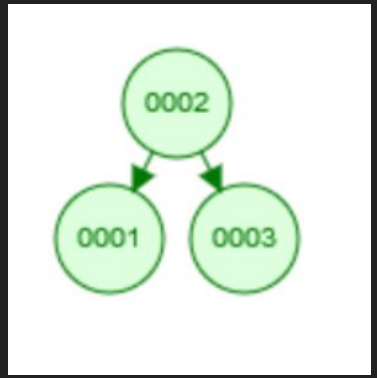
All of these rotations should have O(1) time complexity.

Left & Right Rotations

Right Right Case
Parent: -2
Child: -1

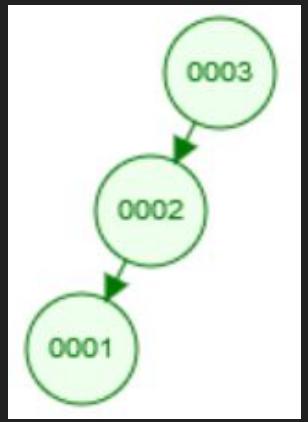
Left Rotation





Left Left Case
Parent: 2
Child: 1

Right Rotation



Left Rotation

Right Right Case
Parent: -2
Child: -1

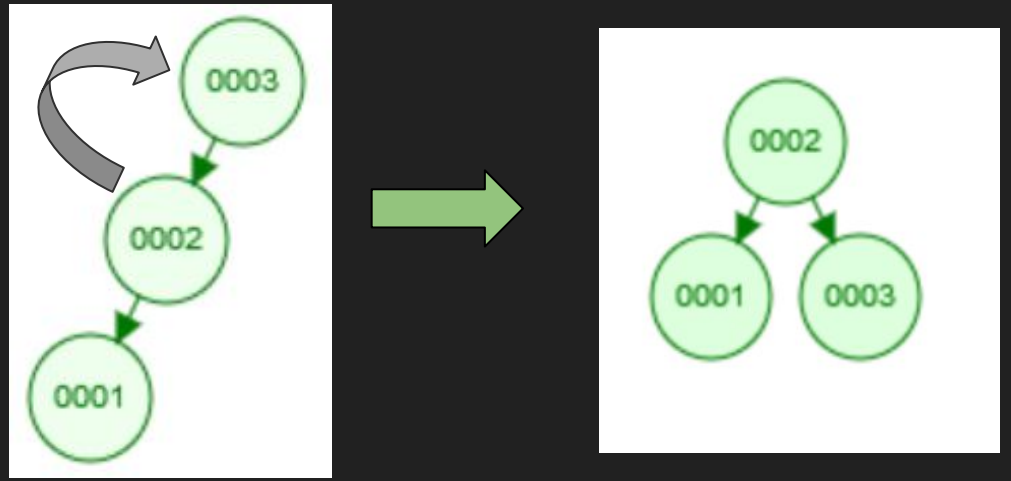
Left Rotation

Call leftRotate on the root
Set root = the result to
properly update the tree

Right Rotation

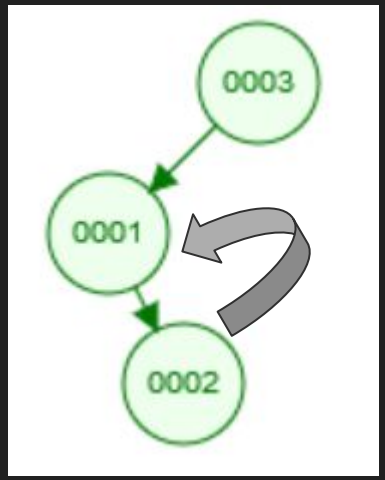
Left Left Case
Parent: 2
Child: 1

Right Rotation



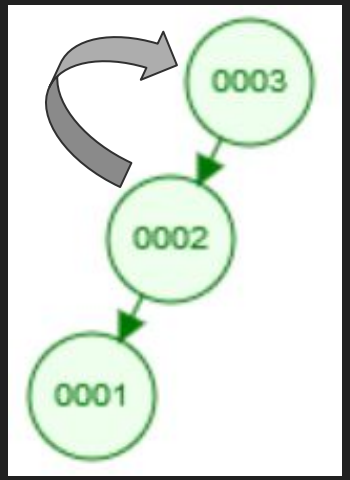
Call rightRotate on the root
Set root = the result to properly update the tree

Left-Right Rotation



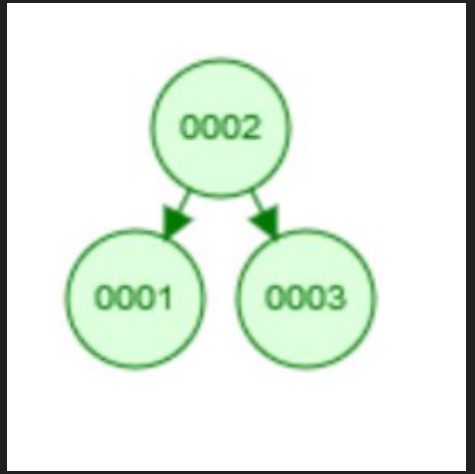
Left Right Case
Parent: 2
Child: -1

Left Right Rotation



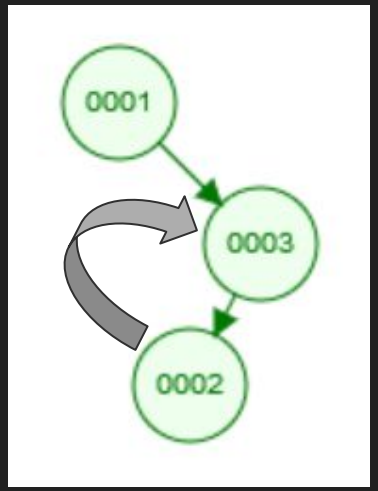
Left Left Case
Parent: 2
Child: 1

Right Rotation



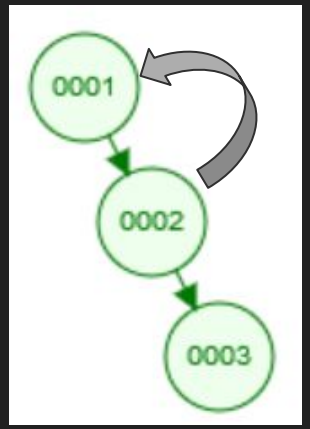
Call leftRotate on root's left
Set root's left = the result to properly update the tree. Then call rotateRight on root. Set root = the result to properly update the tree

Right-Left Rotation



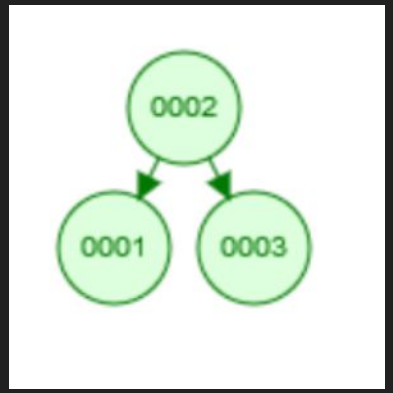
Right Left Case
Parent: -2
Child: 1

Right Left Rotation



Right Right Case
Parent: -2
Child: -1

Left Rotation



Call rightRotate on root's right
Set root's right = the result to properly update the tree. Then call rotateLeft on root. Set root = the result to properly update the tree

When to use each rotation

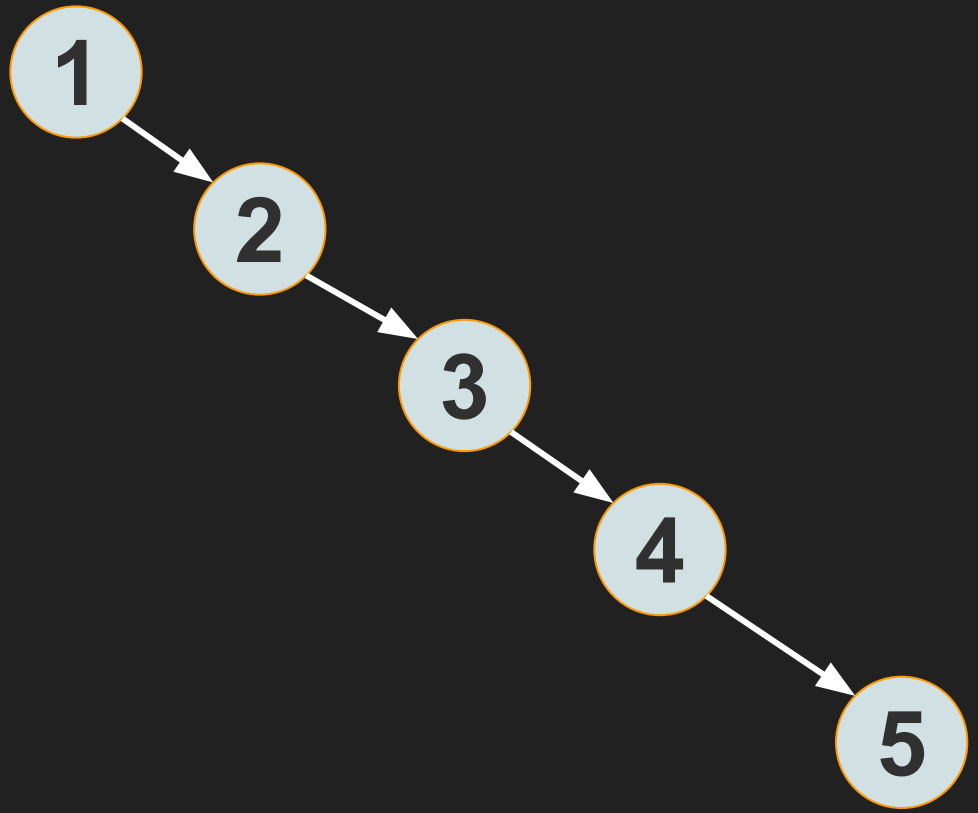
Case (Alignment)	Balance Factor		Rotation
	Parent	★ Child	
Left Left	+2	+1	Right
Right Right	-2	-1	Left
Left Right	+2	-1	Left Right
Right Left	-2	+1	Right Left

For this discussion, all balance factors are (height of left subtree) - (height of right subtree)

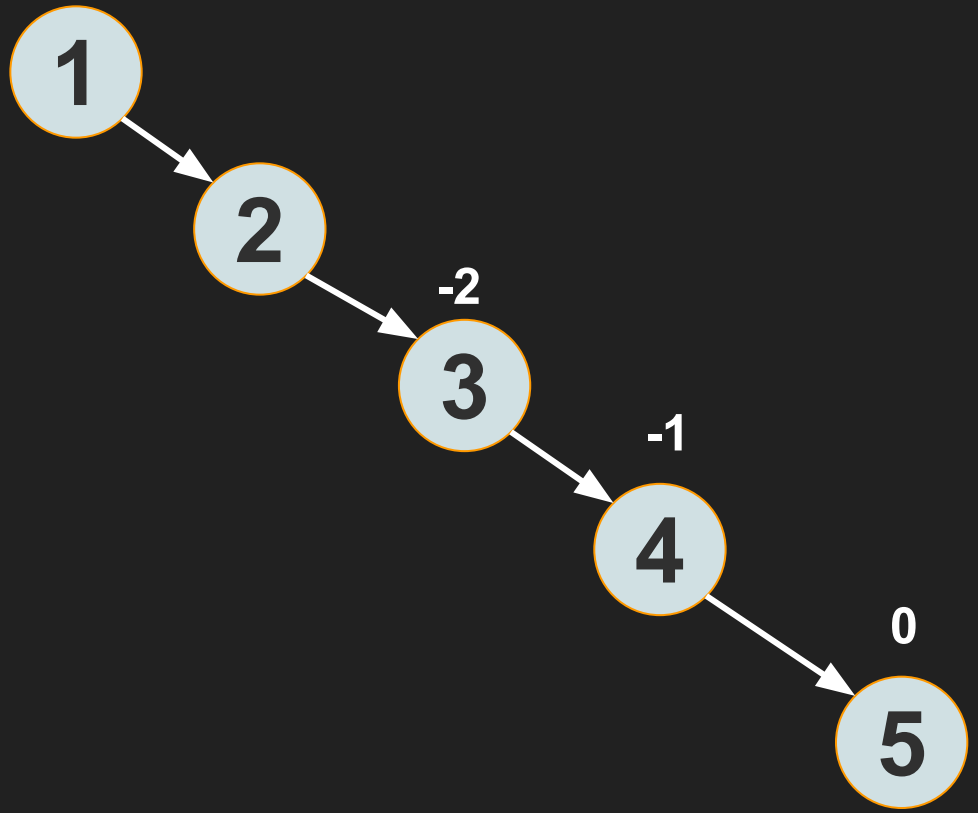
★ If the tree is left heavy, check the left child

If the tree is right heavy, check the right child

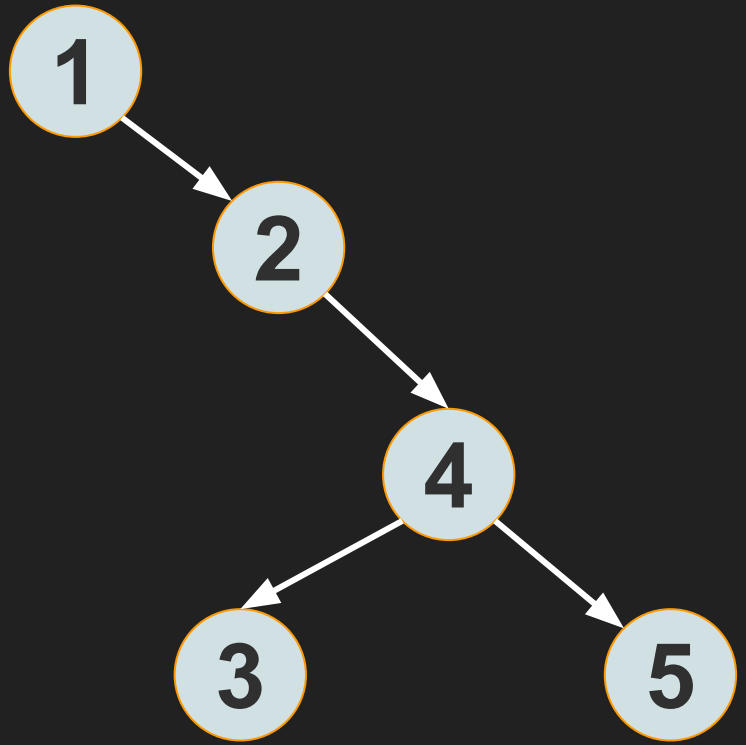
Question 1: What type of rotation does this tree need?



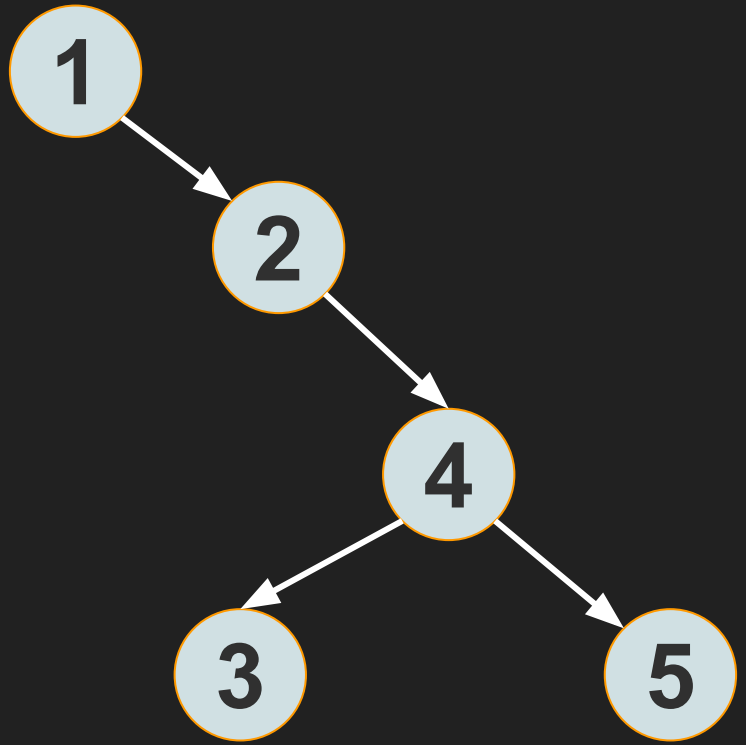
Must be a Left Rotation (look at the balance factors)



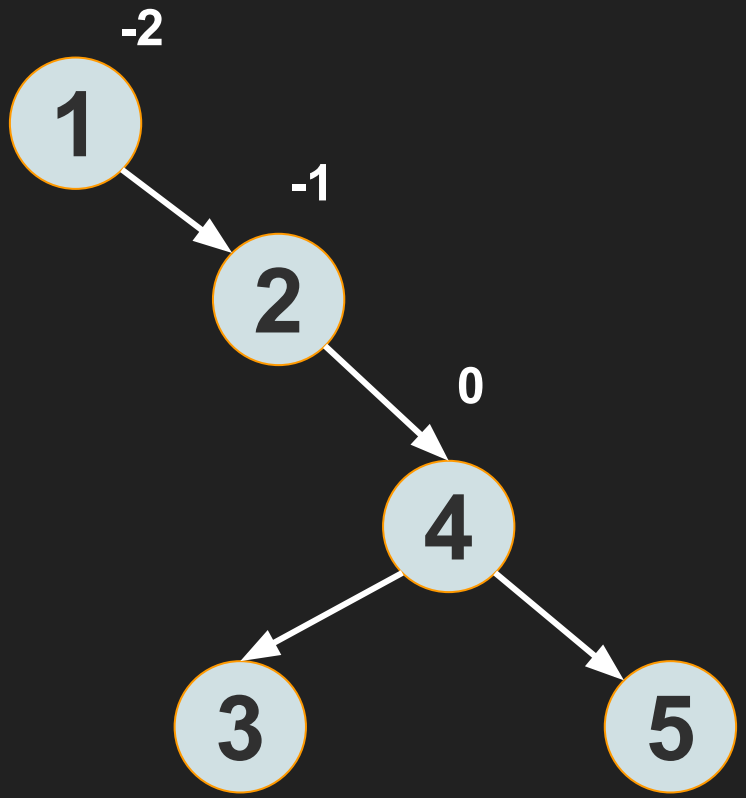
Perform the Rotation



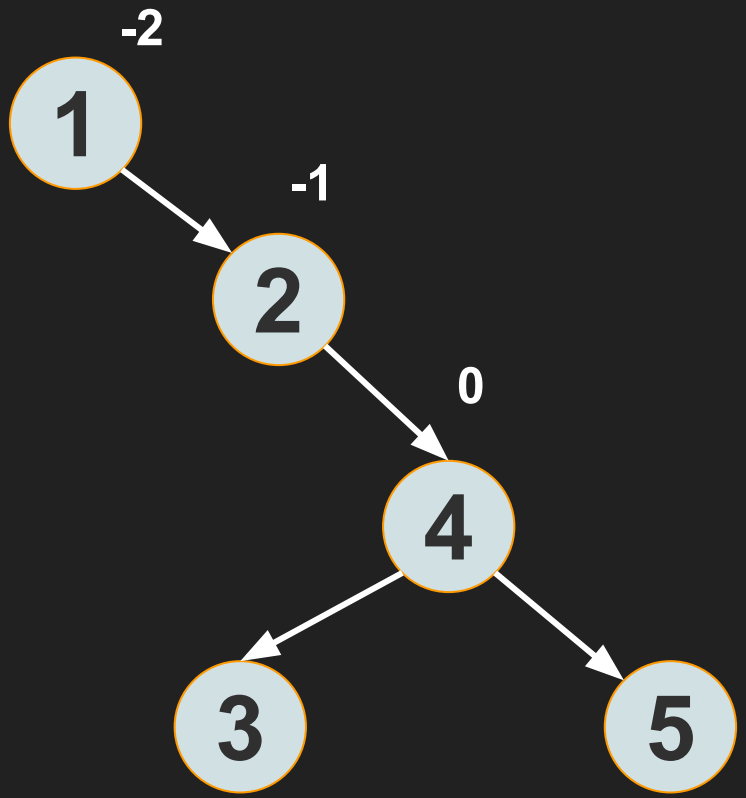
Question 2: Is the Tree balanced?



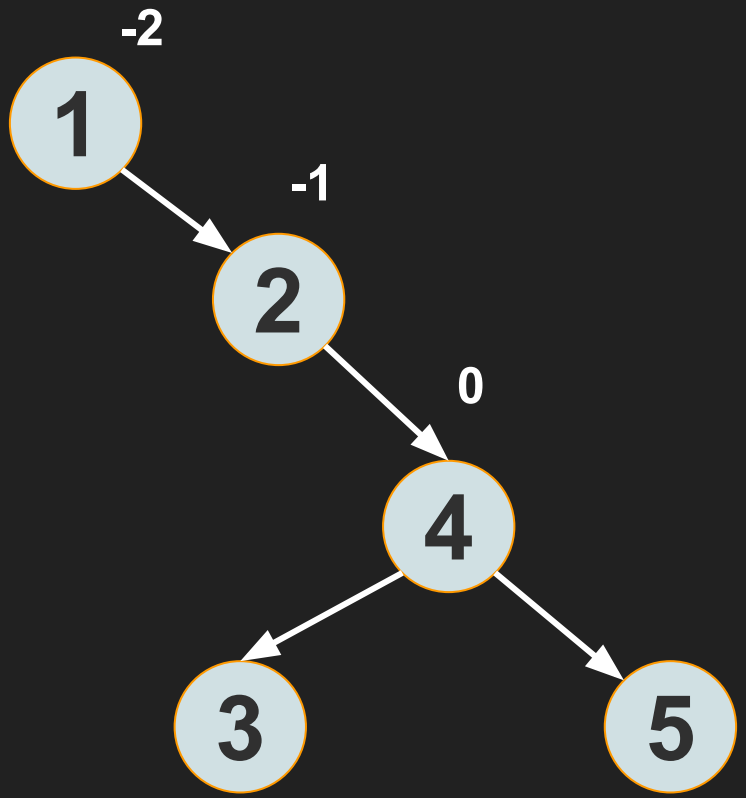
No, look at the balance factors of the top 3 nodes



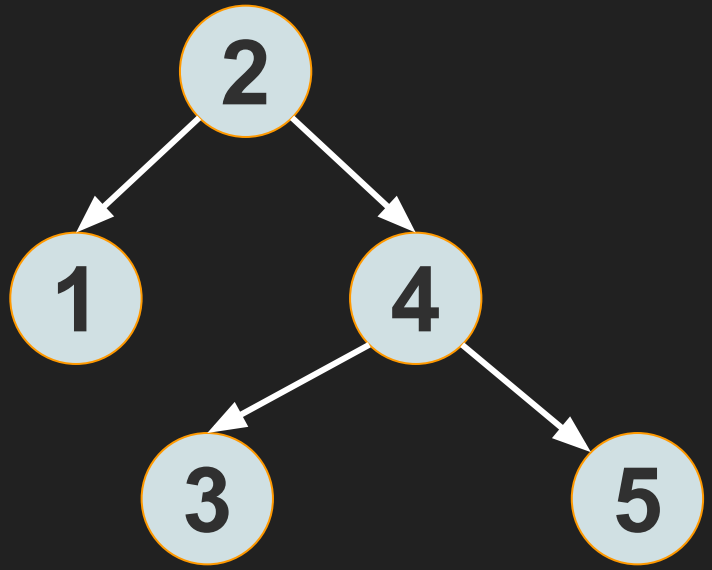
Question 3: What type of rotation does this tree need?



Left Rotation (same situation as before)

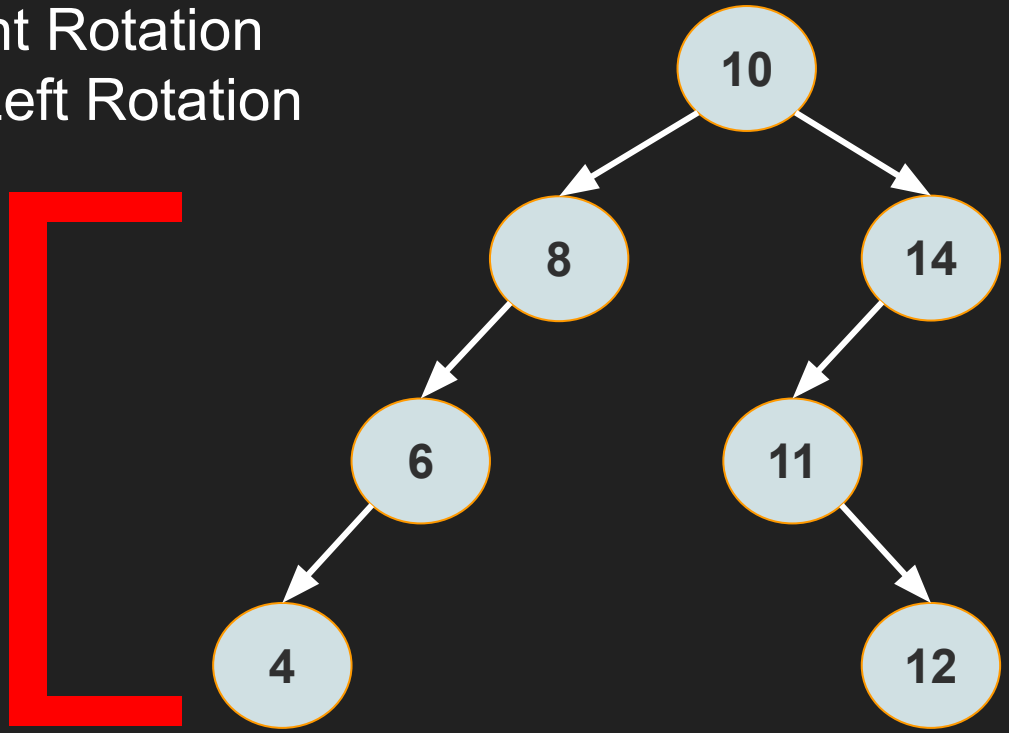


Perform the rotation, & the tree is now balanced



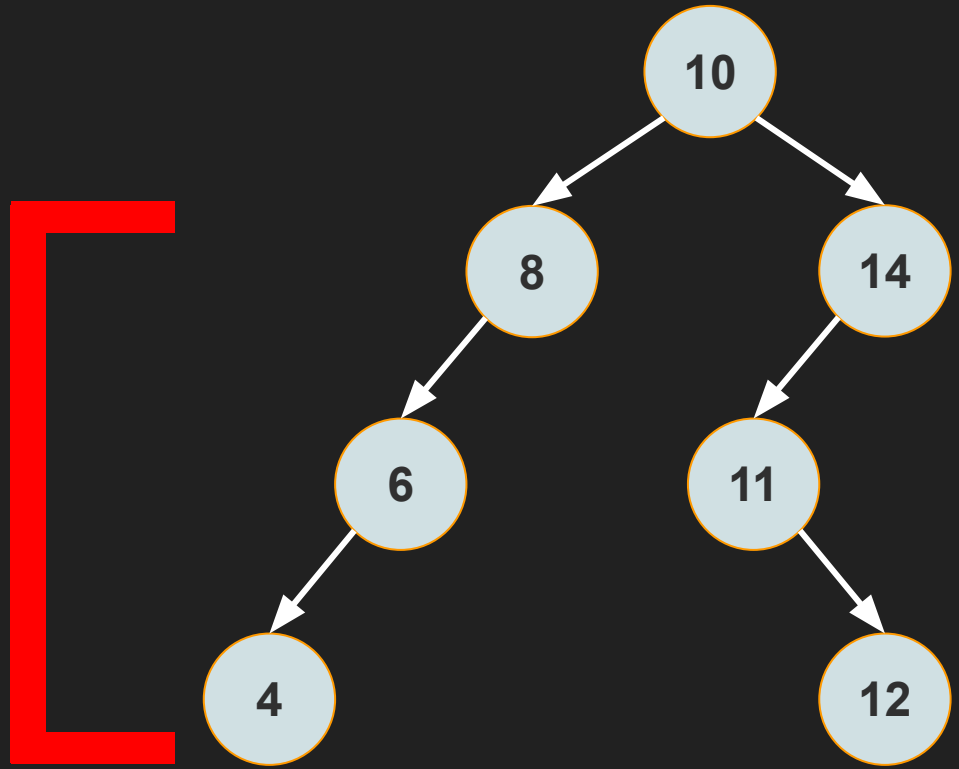
Question 1: For the left subtree [4, 6, and 8]:

- What imbalance is it? What rotation will fix it?
  - A. Left-Right Case; Right-Left Rotation
  - B. Right-Left Case; Right-Left Rotation
  - C. Left-Left Case; Right Rotation
  - D. Right-Right Case; Left Rotation

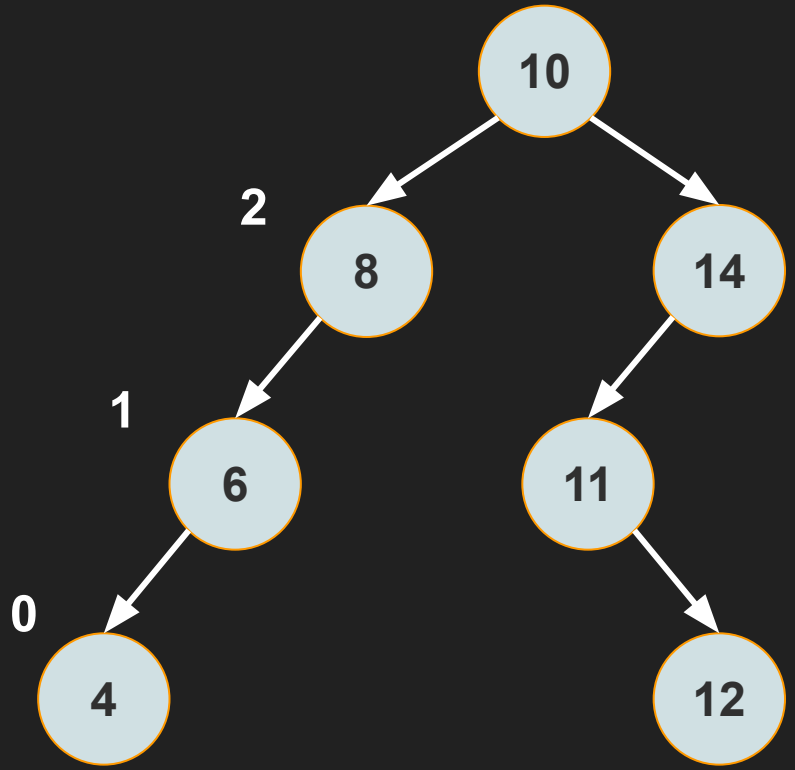


Question 1: For the left subtree [4, 6, and 8]:

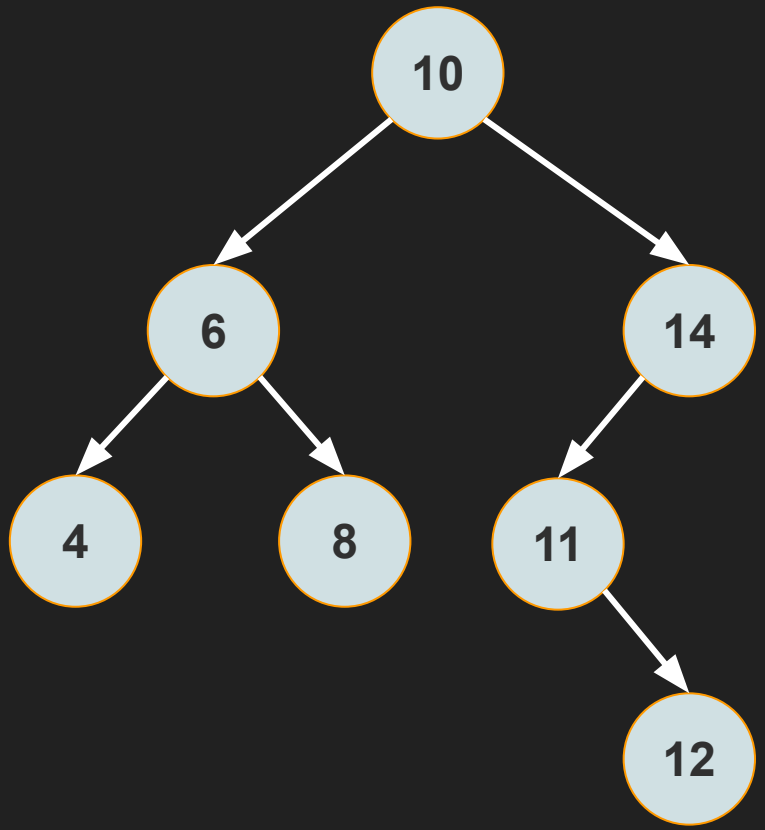
- What imbalance is it?
- What rotation will fix it?



- Question 1 Answer:
  - C. Left - Left Case; Right Rotation
  - Look at the balance factors

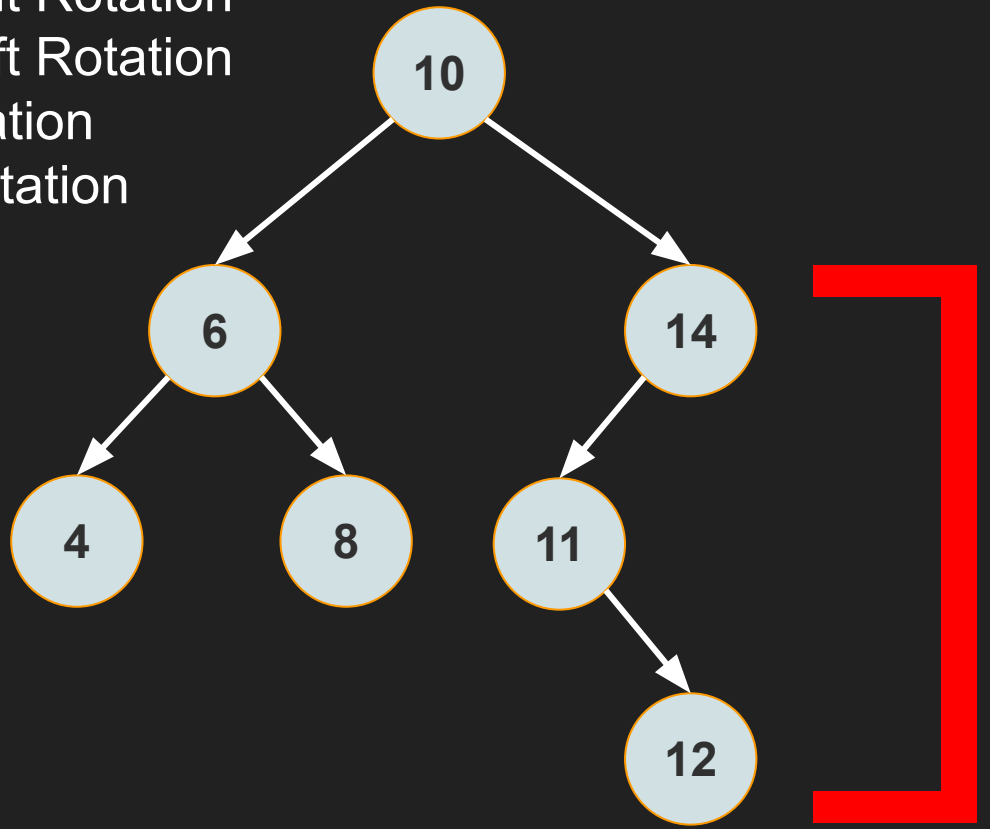


Perform the Rotation

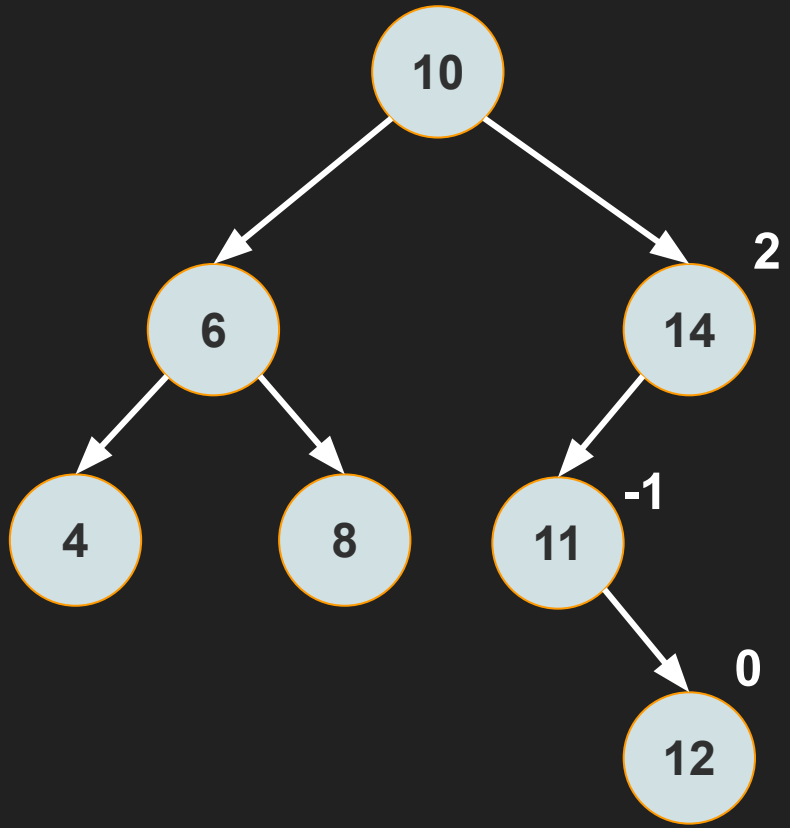


Question 2: For the right subtree [14, 11, and 12]:

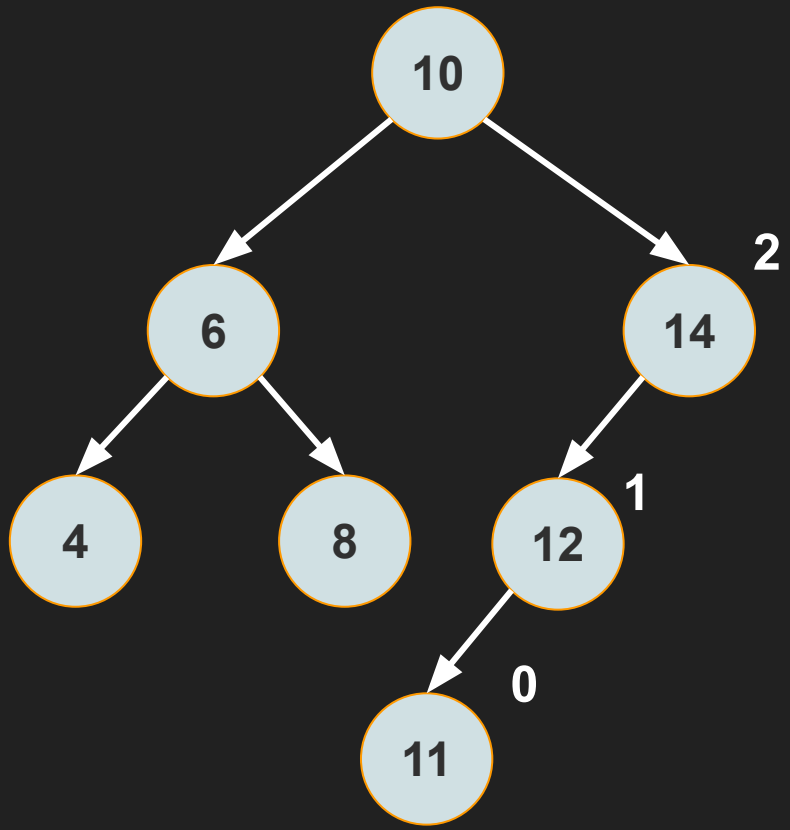
- What imbalance is it? What rotation will fix it?
  - A. Left-Right Case; Left-Right Rotation
  - B. Right-Left Case; Right-Left Rotation
  - C. Left-Left Case; Right Rotation
  - D. Right-Right Case; Left Rotation



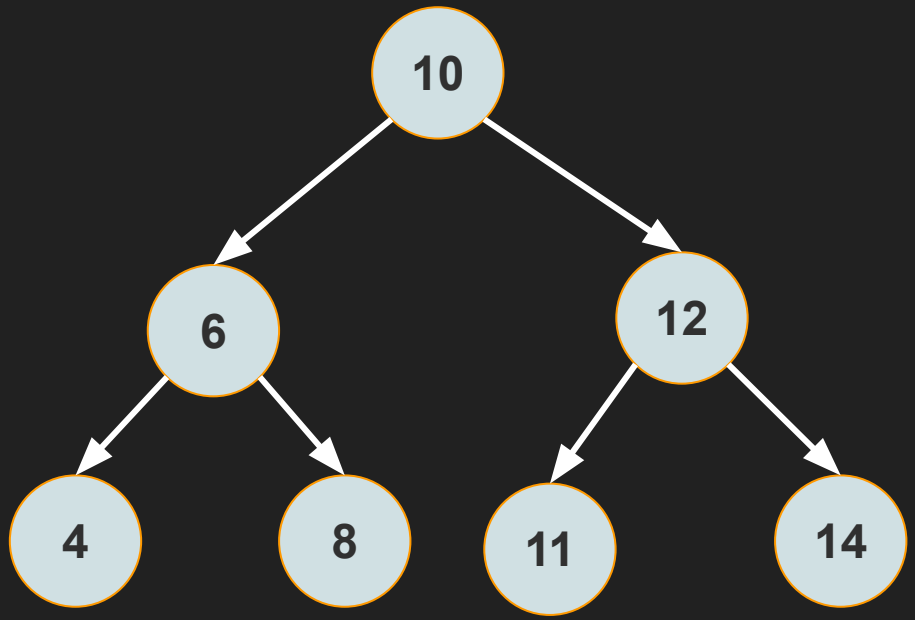
- Question 2 Answer:
  - A. Left - Right Case; Left-Right Rotation
  - Look at the balance factors



Perform the rotation (part 1)

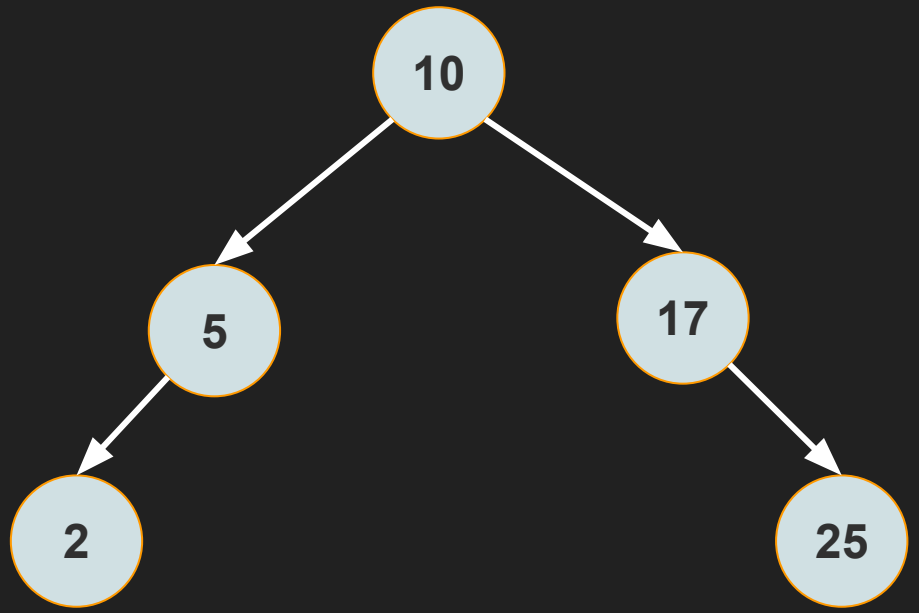


Perform the rotation (part 2)
And the tree is balanced!



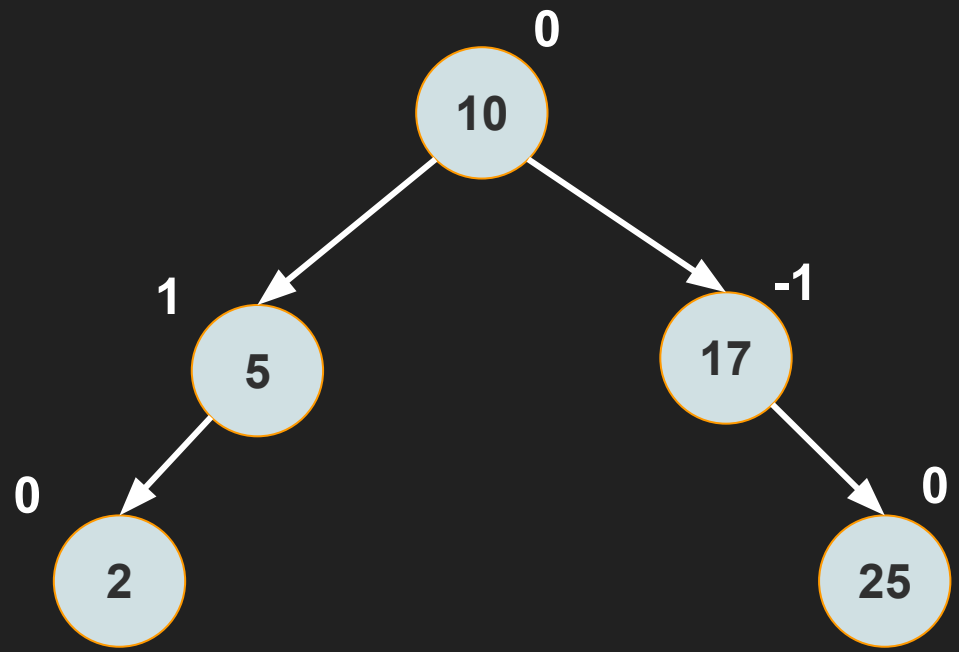
Question 3: What rotations does this tree need?

- A. Right-Left
- B. Left
- C. Right
- D. Both B and C
- E. None



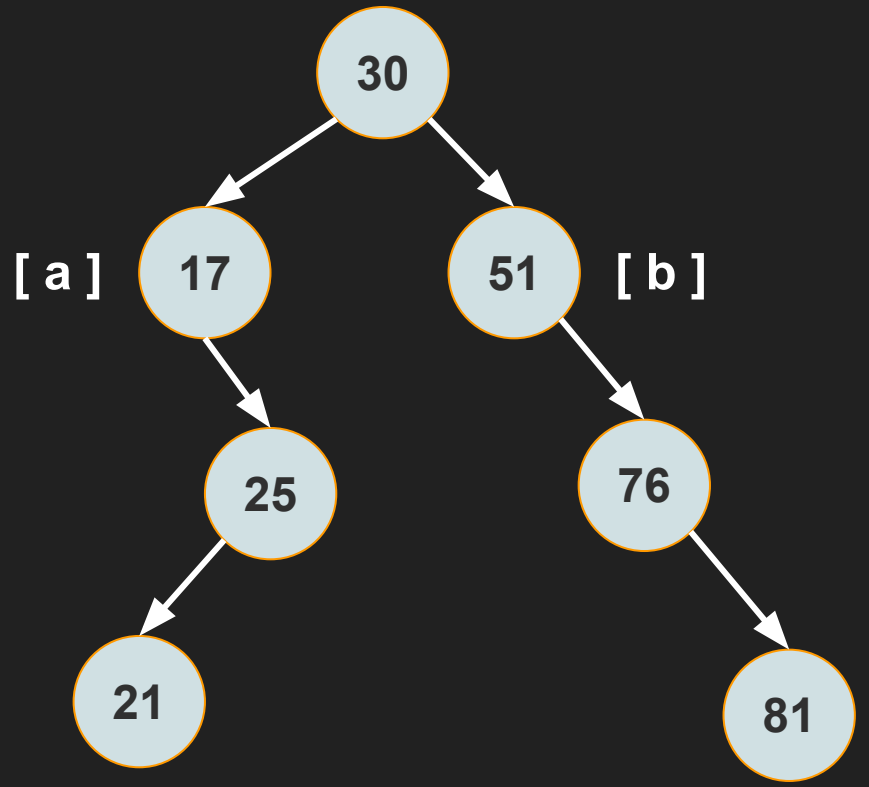
Question 3:

- E. None! It’s already balanced



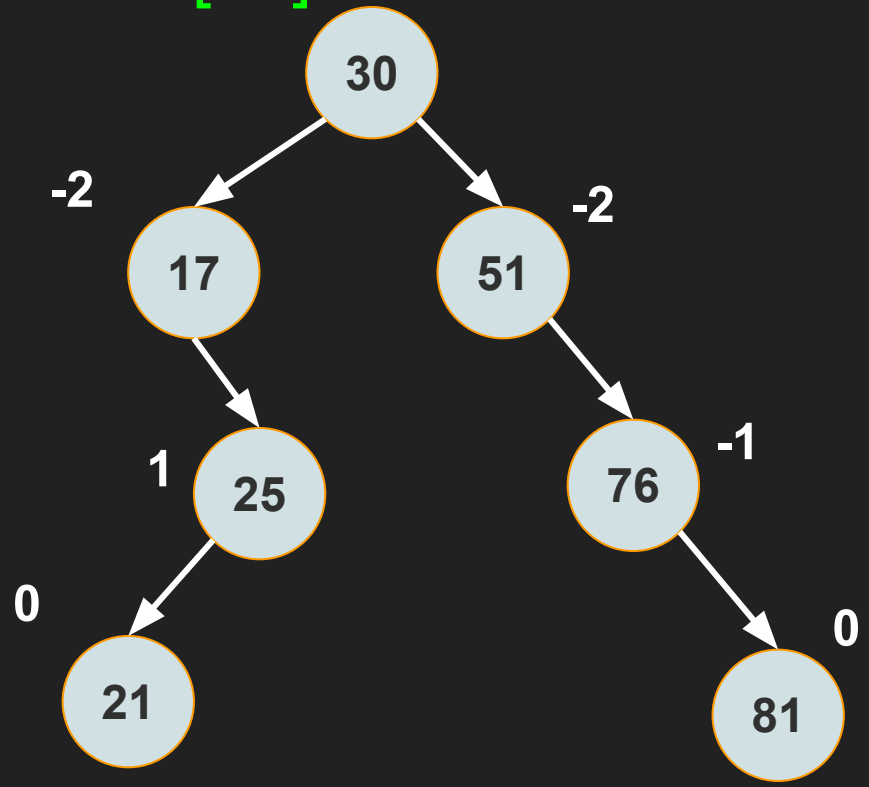
Question 4: What rotations are needed at [a] and [b]?

A. [a] Left [b] Right
B. [a] Right Left [b] Left
C. [a] Left Right [b] Right
D. [a] Right [b] Left



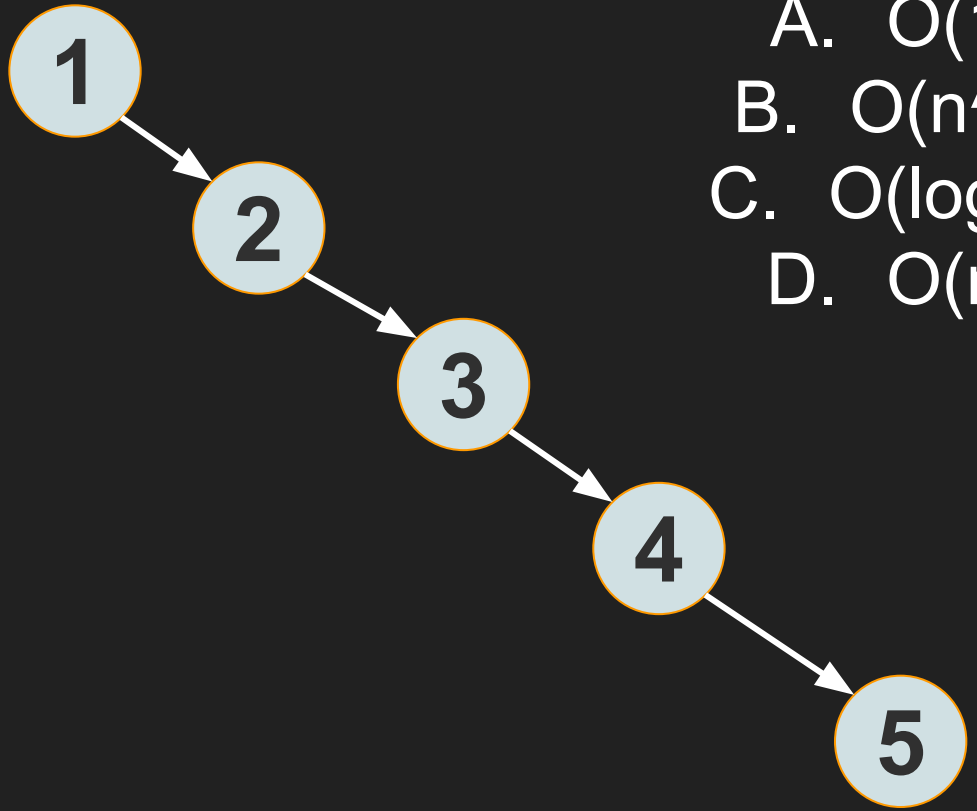
Question 4:

B. [a] - Right Left [b] - Left



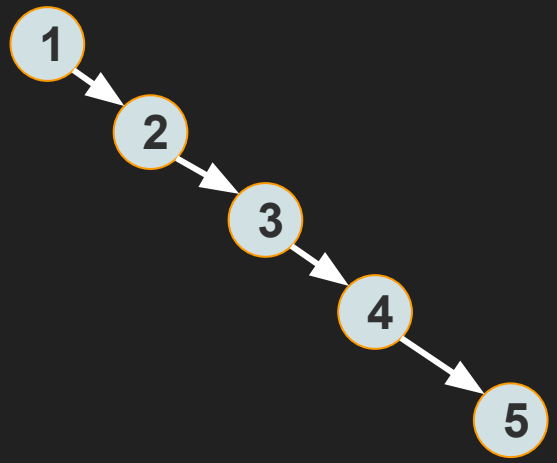
Question 5: What is the worst case time complexity of inserting into this tree?

A. O(1)
B. O(n^2)
C. O(log n)
D. O(n)

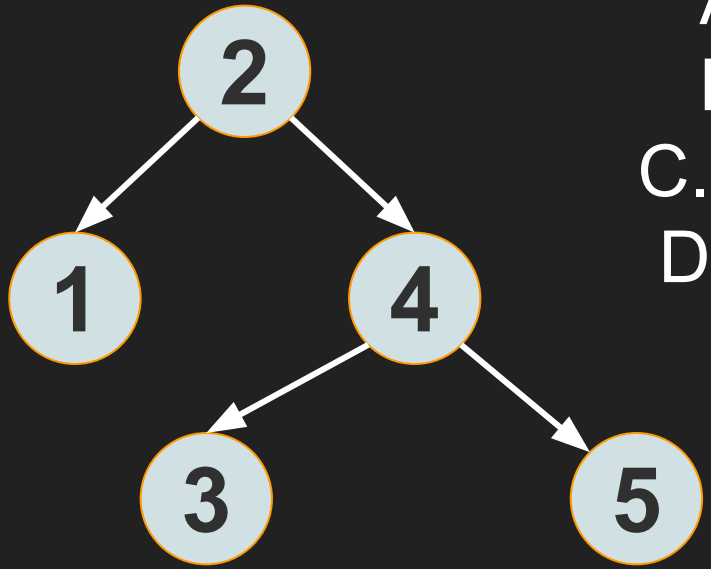


- Since this tree is unbalanced, the worst case insertion would have you iterate through n elements.
- Therefore, the worst case time complexity is D. O(n)

- Keep in mind that worst case time complexity for searching and deletion in an unbalanced tree is also O(n)



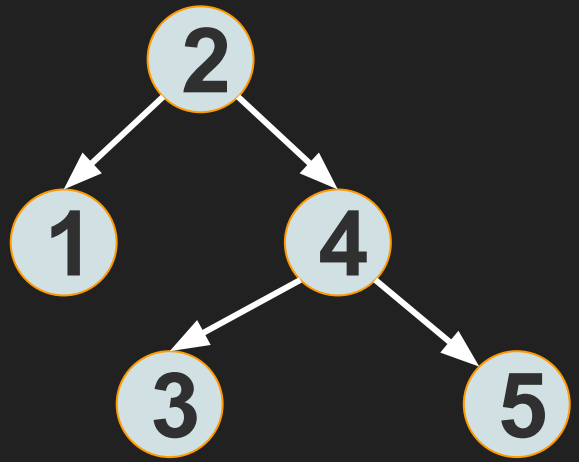
Question 6: What is the worst case time complexity of searching in this binary tree?



A. O(1)
B. O(n)
C. O(log n)
D. O(n^2)

- Since this tree is balanced, the worst case searching would have you iterate through log (n) elements.
- Therefore, the worst case time complexity is C. O(log n)

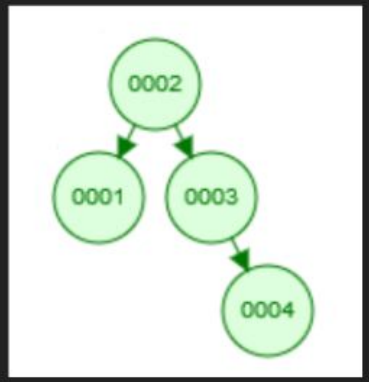
- Keep in mind that worst case time complexity for insertion and deletion in a balanced tree is also O(log n)



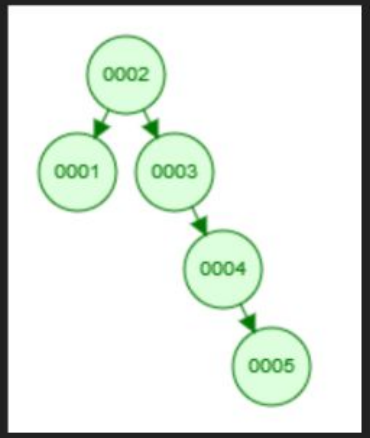
AVL Trees

- A self-balancing binary tree
- Every node has a height or balance attribute, which may be updated in an insertion or deletion based on the placement of the new node.
- After each insertion, at most one O(1) rotation happens. After each deletion, at most O(log n) rotations happen
- Since an AVL tree stays balanced, its insertion, deletion, and searching elements are O(log n)
- All elements should have balance factors of -1, 0, or 1

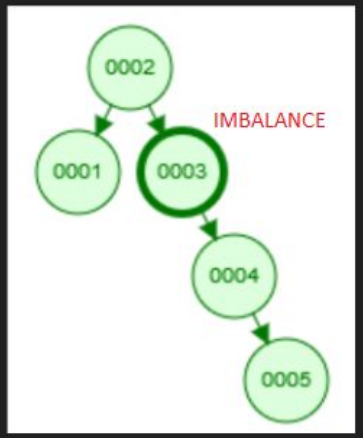
AVL Tree Insertion + Adjustment Example



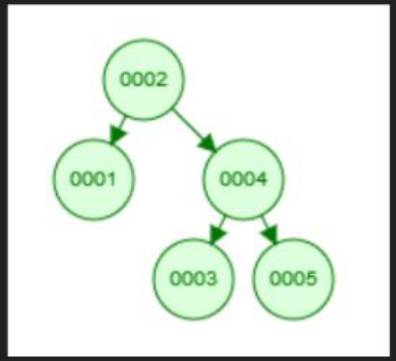
Original, balanced AVL Tree



Insert 5 into the tree, no longer balanced



Tree iterates through the nodes it passed on the way to insert to check if it is balanced



Automatically perform left rotation to balance the tree

AVL Tree Visualizer:
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Red-black trees

Self-balancing BST that follows these 4 rules:

1. A node is either red or black
2. The root is **always** black
3. A red node always has black children (a null reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

NOTE: You won't have to code these - just understand them conceptually.

Applications: Scheduling algorithms, backend to C++ map class

Visualization: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Question 7

Which of the following are true about a red-black tree? (select multiple)

- A. A node is either red or black
- B. The root is always red
- C. The number of black nodes in any path from the root to leaf is the same
- D. Red and black trees inspired the outfit that Deadpool wears
- E. If a node is black, its children must be black

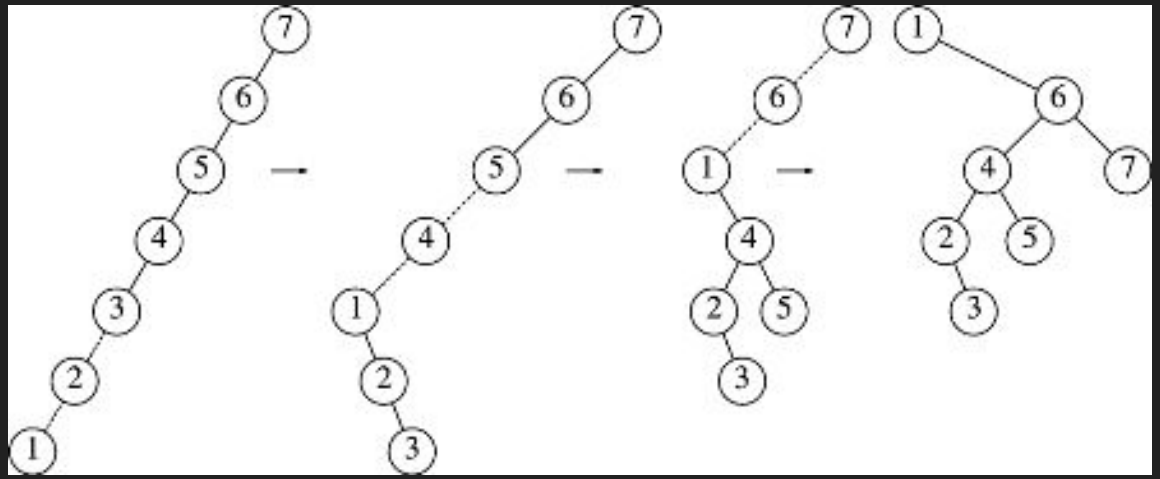
Question 7

Which of the following are true about a red-black tree? (select multiple)

- A. A node is either red or black
- B. The root is always red
- C. The number of black nodes in any path from the root to leaf is the same
- D. Red and black trees inspired the outfit that Deadpool wears
- E. If a node is black, its children must be black

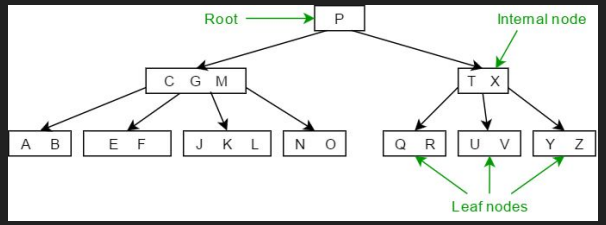
Splay Trees

- Self balancing binary search tree
- After a node is accessed, it is moved to the root via “zig zag” rotations
- Best for ensuring that most commonly accessed nodes have the quick access time



B/B+ Trees

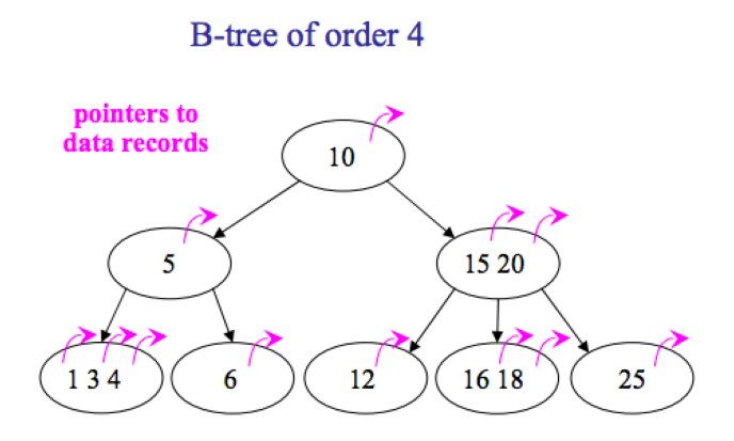
- N-ary tree, where each node has up to n children.
- Distinctinction between B/B+ Trees:
  a. In B+ trees, all data is stored in leaves.
  b. B+ trees' leaves are connected to each other.
- Important properties of B-trees
  a. All nodes have at most n children.
  b. Root is a leaf or has 2 to n children.
  c. Internal or non-leaf nodes have ceil(n/2) to n children.
  d. Non-leaf or internal nodes have a maximum of L keys, where L= n-1.
- Why another balanced tree?
  a. Data pulled from external "blocks" (like cache blocks)

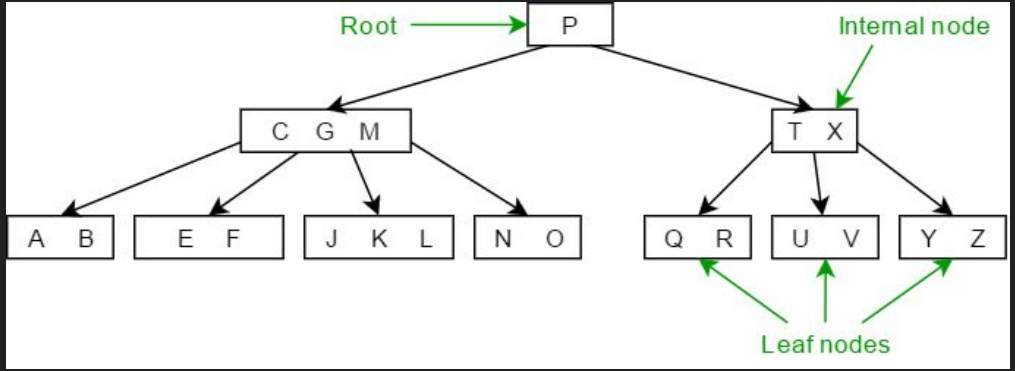


Used in indexing file systems and databases

B Trees

- Self-balancing tree used for key-value storage.
- Designed to store large amounts of data for fast query and retrieval.
- N-ary tree, where each node has up to n children.
- Can have multiple key-value pairs in a node, sorted ascendingly by keys.





B Tree Properties

Important properties of B-trees

a. All nodes have at most n children.
b. The root node is either a leaf, or has 2 to n children.
c. Internal or non-leaf nodes have ceil(n/2)* to n children.
d. Non-leaf or internal nodes have a maximum of n-1 keys.
e. L is the maximum number of keys/items a node can have

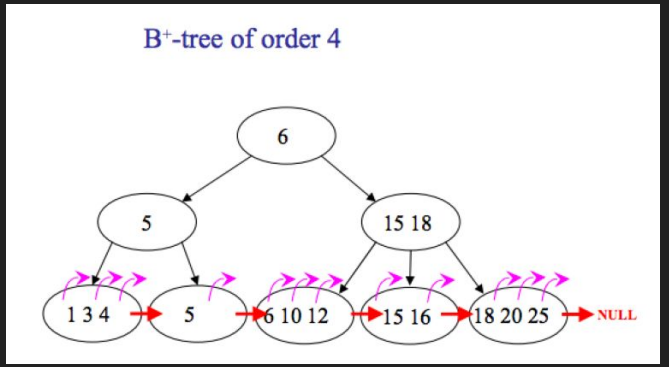
** If function returns a decimal, round up to the nearest integer*

B+ Trees

Can be viewed as a B-tree with the following distinctions:

a. Each node contains **only** keys (not pairs of key-value)
b. All data is **stored in leaves**
c. Additional level is added at the bottom with linked leaves

B-Tree	B+ Tree
Data is stored in leaf nodes as well as internal nodes.	Data is stored only in leaf nodes.
Searching is a bit slower as data is stored in internal as well as leaf nodes.	Searching is faster as the data is stored only in the leaf nodes.
No redundant search keys are present.	Redundant search keys may be present.
Deletion operation is complex.	Deletion operation is easy as data can be directly deleted from the leaf nodes.
Leaf nodes cannot be linked together.	Leaf nodes are linked together to form a linked list.



B/B+ Trees

Why another balanced tree?

- Data pulled from external "blocks" (like cache blocks, each block containing a lot of data)

What are the applications of B/B+ trees:

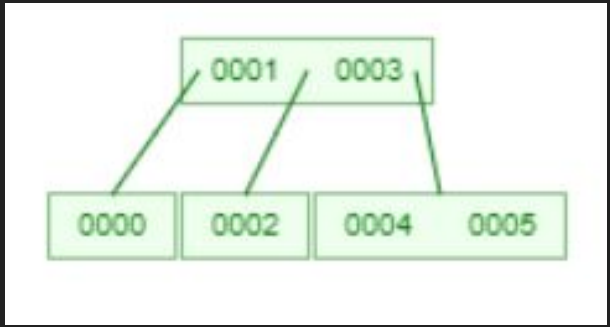
- indexing file systems and databases

Question 8:

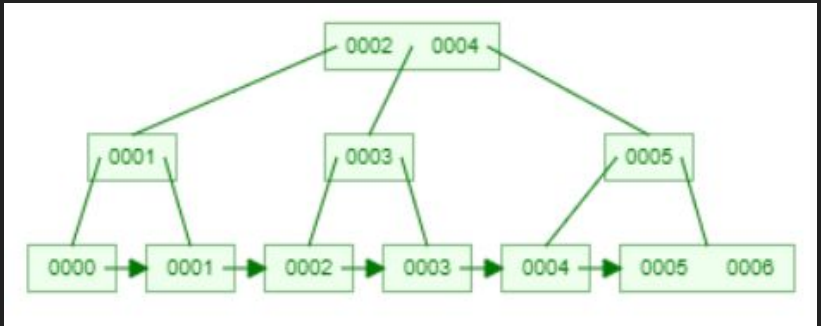
Which of these is a B tree? Which is a B+ tree?

A. T1 is a B tree, T2 is a B+ tree
B. T1 is a B+ tree, T2 is a B tree
C. Both are B trees
D. Both are B+ trees

Tree 1



Tree 2



Question 8:

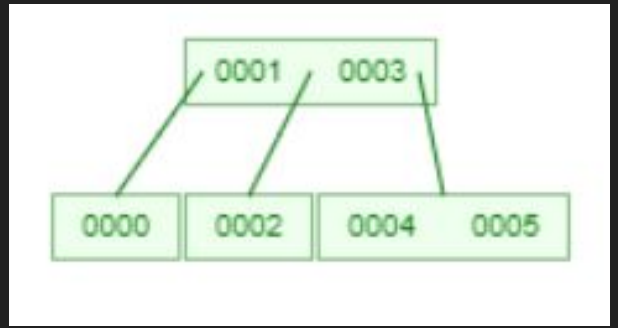
Which of these is a B tree? Which is a B+ tree?

A. T1 is a B tree, T2 is a B+ tree
B. T1 is a B+ tree, T2 is a B tree
C. Both are B trees
D. Both are B+ trees

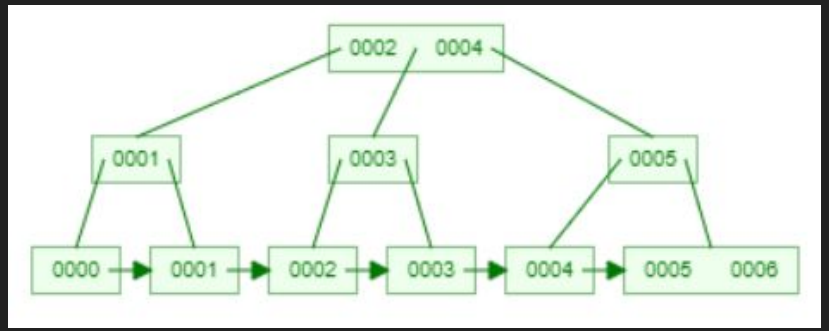
Remember:

- a B+ tree stores all of its data in its leaves and links the leaves together
- A B tree doesn't have duplicates

Tree 1



Tree 2



Question 9a:

You're presented with a **perfect** B+ tree with N=3 and L=3 and height = 2 (has levels 0, 1, and 2). How many unique keys are in this tree?

N = maximum children a node can have. L = maximum number of keys a node can have

a. 9 unique values
b. 27 unique values
c. 18 unique values
d. 35 unique values

Reminder:

a. All nodes have at most n children.
b. Root is a leaf or has min 2 to max n children.
c. Internal or non-leaf nodes have min ceil(n/2), max n children.
d. Non-leaf nodes have up to n-1 keys
e. All leaf nodes have maximum L keys.

Question 9a:

You're presented with a **perfect** B+ tree with N=3 and L=3 and height = 2 (has levels 0, 1, and 2). How many unique keys are in this tree?

N = maximum children a node can have. L = maximum number of keys a node can have

a. 9 unique values
b. 27 unique values
c. 18 unique values
d. 35 unique values

Reminder:

a. All nodes have at most n children.
b. Root is a leaf or has min 2 to max n children.
c. Internal or non-leaf nodes have min ceil(n/2), max n children.
d. Non-leaf nodes have up to n-1 keys
e. All leaf nodes have maximum L keys.

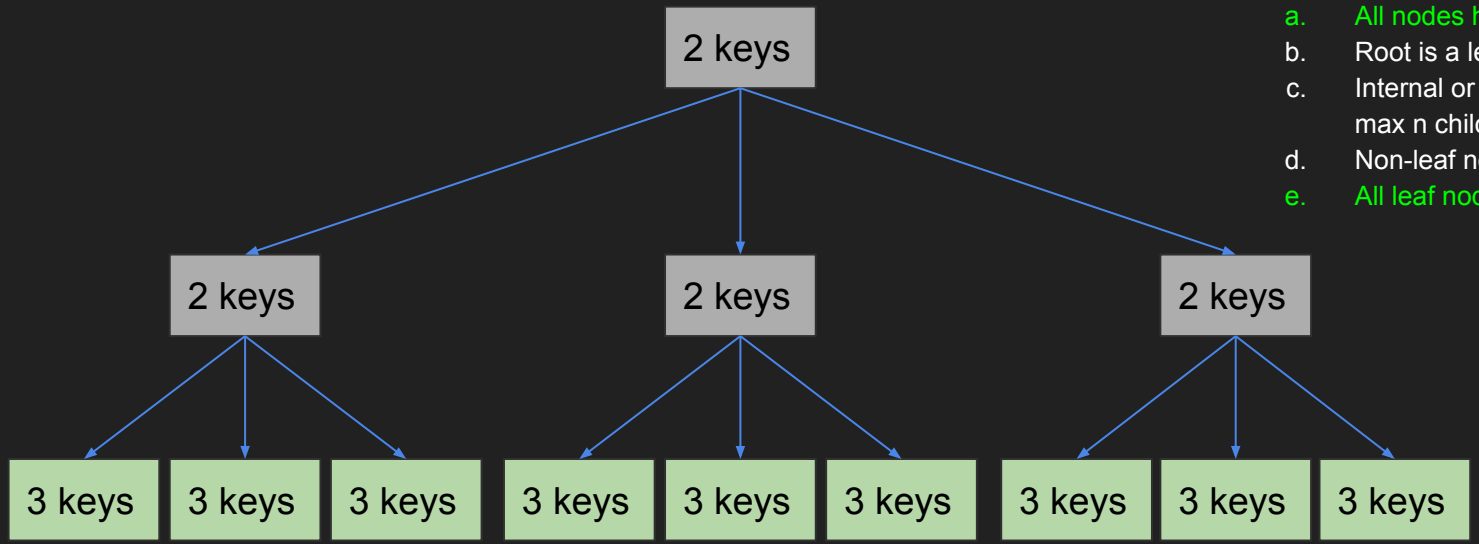
Level 0 = 1 node, each with 2 values
Level 1 = 3 nodes, each with 2 values
Level 2 = 9 nodes, each with 3 values = 27 unique values

Question 9a: (visual representation)

You're presented with a **perfect** B+ tree with N=3 and L=3 and height = 2 (has levels 0, 1, and 2). How many unique keys are in this tree?

Reminder:

a. All nodes have at most n children.
b. Root is a leaf or has min 2 to max n children.
c. Internal or non-leaf nodes have min ceil(n/2), max n children.
d. Non-leaf nodes have up to n-1 keys
e. All leaf nodes have maximum L keys.



Question 9b:

You're presented with a **perfect *B*** tree with N=3 and L=3 and height = 2 (has levels 0, 1, and 2). How many unique keys are in this tree?

N = maximum children a node can have. L = maximum number of keys a node can have

a. 9 unique values
b. 27 unique values
c. 18 unique values
d. 35 unique values

Reminder:

a. All nodes have at most n children.
b. Root is a leaf or has min 2 to max n children.
c. Internal or non-leaf nodes have min ceil(n/2), max n children.
d. Non-leaf nodes have up to n-1 keys
e. All leaf nodes have maximum L keys.

Question 9b:

You're presented with a **perfect *B*** tree with N=3 and L=3 and height = 2 (has levels 0, 1, and 2). How many unique keys are in this tree?

N = maximum children a node can have. L = maximum number of keys a node can have

a. 9 unique values
b. 27 unique values
c. 18 unique values
d. 35 unique values

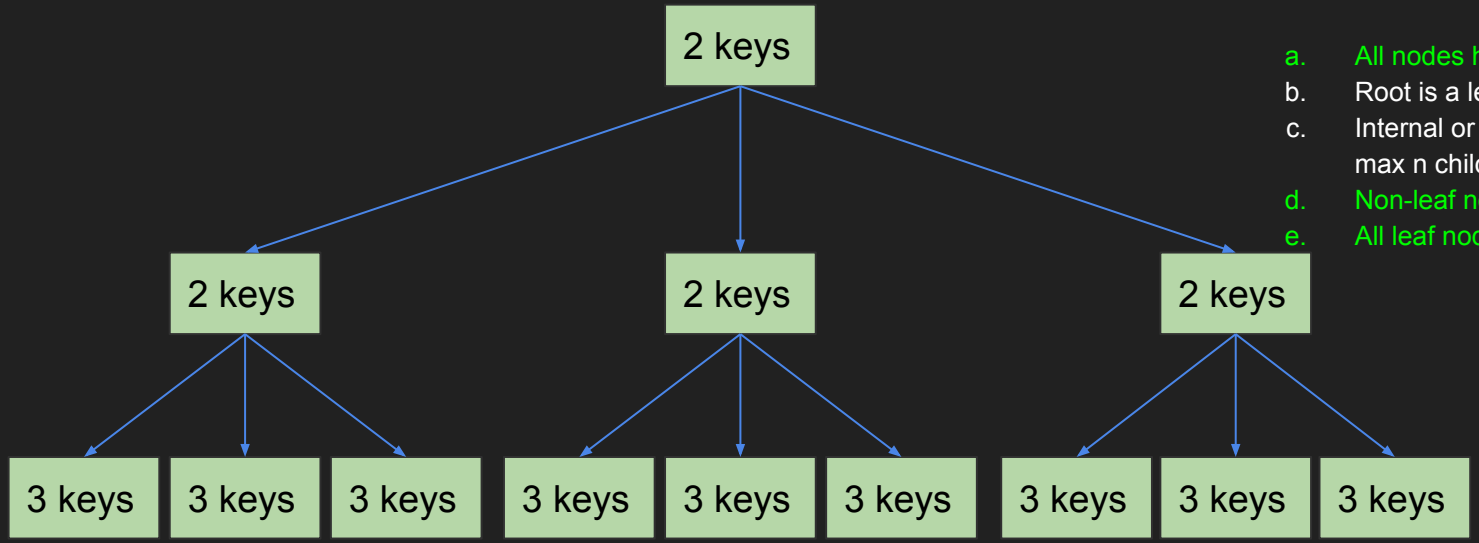
Reminder:

a. All nodes have at most n children.
b. Root is a leaf or has min 2 to max n children.
c. Internal or non-leaf nodes have min ceil(n/2), max n children.
d. Non-leaf nodes have up to n-1 keys
e. All leaf nodes have maximum L keys.

Level 0 = 1 node, each with 2 values = 2 unique values
Level 1 = 3 nodes, each with 2 values = 2 + 3(2) = 8 unique values
Level 2 = 9 nodes, each with 3 values = 8 + 9(3) = 35 unique values

Question 9b: (visual representation)

You're presented with a perfect B tree with N=3 and L=3 and height = 2 (has levels 0, 1, and 2). How many unique keys are in this tree?



Reminder:

a. All nodes have at most n children.
b. Root is a leaf or has min 2 to max n children.
c. Internal or non-leaf nodes have min ceil(n/2), max n children.
d. Non-leaf nodes have up to n-1 keys
e. All leaf nodes have maximum L keys.

B/B+ trees are confusing. How do I get used to them?

- Here are two awesome visualizations of B and B+ trees:
  - https://www.cs.usfca.edu/~galles/visualization/BTree.html
  - https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- Split your screen and open both visualizations in browser windows
- Try adding this sequence of keys to both trees: 0, 1, 2, 3, 4, 5, 6. How do the two trees change as you add more data?
- Seeing visualizations of both trees will help you recall their properties on quizzes and exams 😎
- It would also be valuable to view the construction of B/B+ trees to understand their properties. It might prove handy later on 👀

Other Visualizations

AVL Trees: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Splay Trees: https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

Red-Black Trees: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Questions?

Discussion Participation Activity

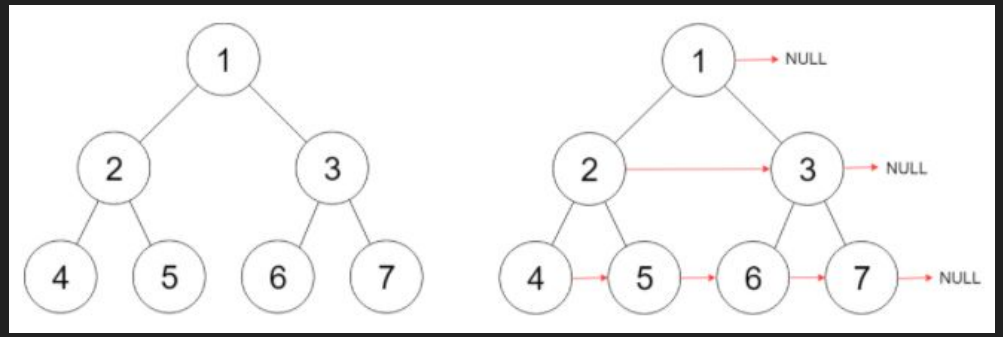
- Leapfrog through Balanced Trees!
  - https://www.educaplay.com/learning-resources/13951193-jump_through_balanced_trees.html
  - Rules
    - 20 seconds to jump to lily pad with correct answer
    - Wrong answer means you lose a life!
    - 3 lives for 20 Balanced Tree questions
  - Leaderboard for top 10 DSA-wide!
- Have fun! 🐸

Coding Question

You are presented with a perfect binary tree whose Node pointer looks like this:

```
1  struct Node {
2      int val;
3      Node *left;
4      Node *right;
5      Node *next;
6  };
```

The next pointer is curious. We want to connect each pointer's next value to the node to its right, like below. Write a way to make this modification to the tree.



Solution

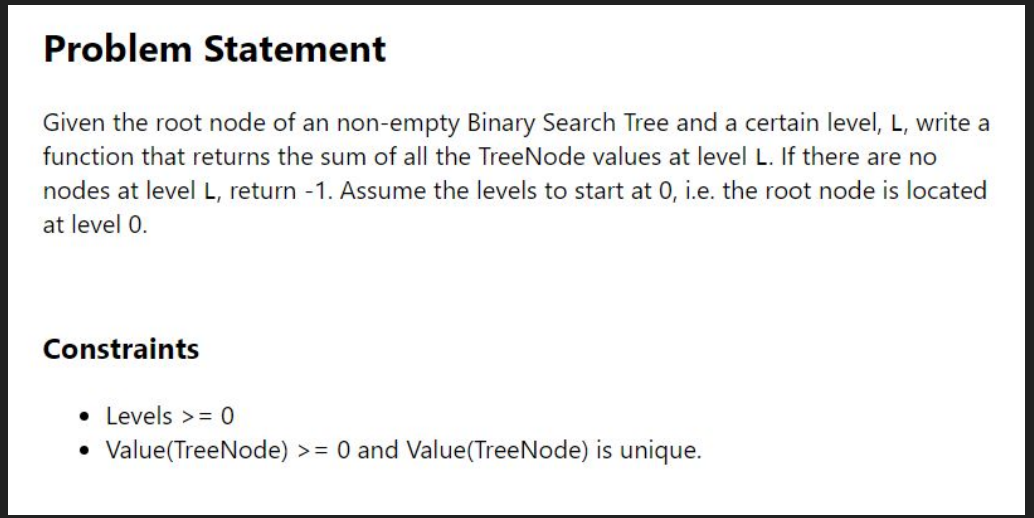
We're going to use the same idea of using a level order traversal.

The trick here is that we're setting the curr pointer's next to nullptr if i is at count - 1, and if not, we set curr's next to the value at the front of the queue.

```
Node* connect(Node* root) {
    int count = 0;
    queue<Node*> q;

    if(root)
        q.push(root);

    while(!q.empty())
    {
        int count = q.size();
        for (int i = 0; i < count; i++)
        {
            Node* curr = q.front();
            q.pop();

            // Make next pointer equal to next in traversal
            if(i == count - 1)
                curr->next = nullptr;
            else
                curr->next = q.front();

            if(curr->left)
                q.push(curr->left);
            if(curr->right)
                q.push(curr->right);
        }
    }
    return root;
}
```

Quiz 4 Question

Sum of a level of a binary tree



Problem Statement

Given the root node of an non-empty Binary Search Tree and a certain level, `L`, write a function that returns the sum of all the TreeNode values at level `L`. If there are no nodes at level `L`, return -1. Assume the levels to start at 0, i.e. the root node is located at level 0.

Constraints

- Levels >= 0
- Value(TreeNode) >= 0 and Value(TreeNode) is unique.

Common Issues: How to do level order traversal, how to keep track of the sum, how to keep track of the levels.

Quiz 4 solution

1. Initialize an empty queue, a sum variable, and a level variable

2. If the root is not empty, push it onto the queue

3. For each level of the loop:
- Get the size of the queue
- Loop through the size of the queue
- Add the values at the front of the queue together
- Add the left and right children of the node popped off

4. Return sum of current level is desired level

```
int sumOfLevel(TreeNode* root, int level)
{
    int sum = 0;
    std::queue <TreeNode*> t;
    t.push(root);
    int l = 0;
    while(!t.empty())
    {
        int level_queue = t.size();
        for(int i=0; i<level_queue; i++)
        {
            if(l==level)
            {
                sum+=t.front() -> val;
            }
            if(t.front()->left)
                t.push(t.front()->left);
            if(t.front()->right)
                t.push(t.front()->right);
            t.pop();
        }
        if(l==level)
            return sum;
        l++;
    }
    return -1;
}
```

Reminder:
You have HonorLock Quiz due this Saturday by 11:59pm

Collaborative Question Link

https://docs.google.com/presentation/d/1HgBq1Mld5a25lTqtzts1oIz-jQf0IYLVVwR8xUt2dnI/edit?usp=sharing