CS 1332R WEEK 8

SkipLists

AVLs

AVL Rotations

ANNOUNCEMENTS

- ❑

SkipList

- A **probability-based** data structure
  - The way we add to a SkipList is randomized using a coin flip.
- Outperform traditional data structures like BSTs when it comes to concurrent access, and other use cases that are outside the scope of this course!

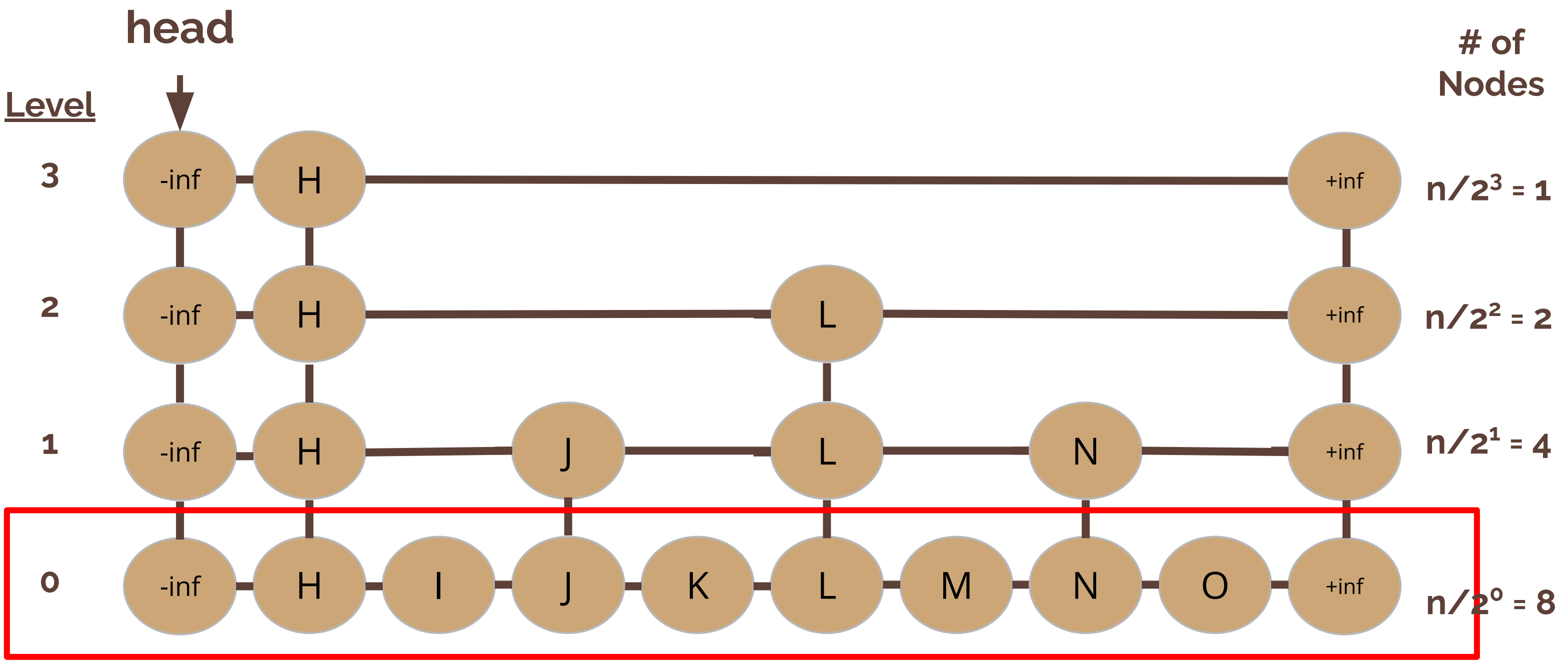
*We do **not** expect you to implement (code) a SkipList in this course. We do expect you to understand its structure & operations.*

SkipList: Structure

- A series of ***sorted*** LinkedLists, each occupying its own level
- Each level contains a subset of the data in the level directly beneath it.
- **Node**: contains data and ***4 references***:
  - Left and Right - normal for a LinkedList
  - Up and Down - to access the nodes directly above and below it
- **Level**: Each level is its own LinkedList, beginning and ending with a phantom node of -inf, +inf respectively

SkipList: Structure

The maximum number of levels in a SkipList is typically log(n).



Bottom level contains all the data in the list.

SkipList: Search

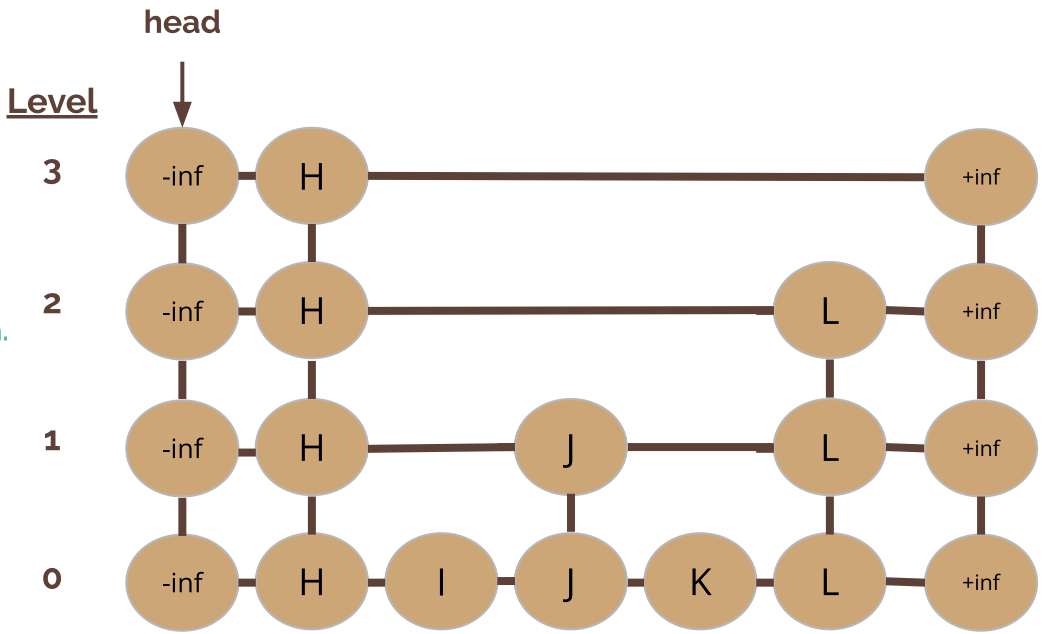
➔ Search is the primary purpose of a SkipList because we should be able to ***skip over*** a lot of our data when searching.

PROCEDURE - Look Ahead

1. Start at the head node.
2. Iterate through the LinkedList at the current level ***i***. Compare with the node ahead. There are **3 cases**:
   a. Data is equal?
      End search, data is found!
   b. Data is greater than our data?
      ***Go down a level*. Continue search.**
   c. Data is less than our data?
      Continue search.

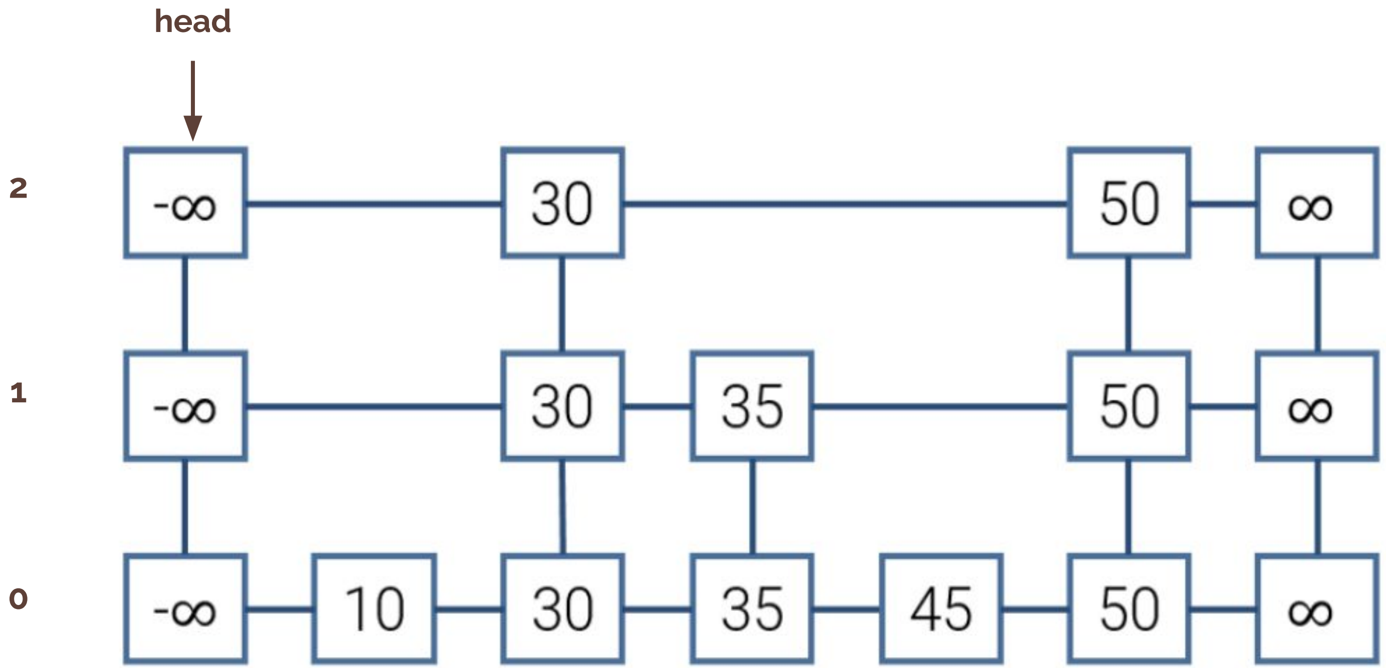
How do we know data is not in the SkipList?

The node ahead is larger than `data` and we are on level 0



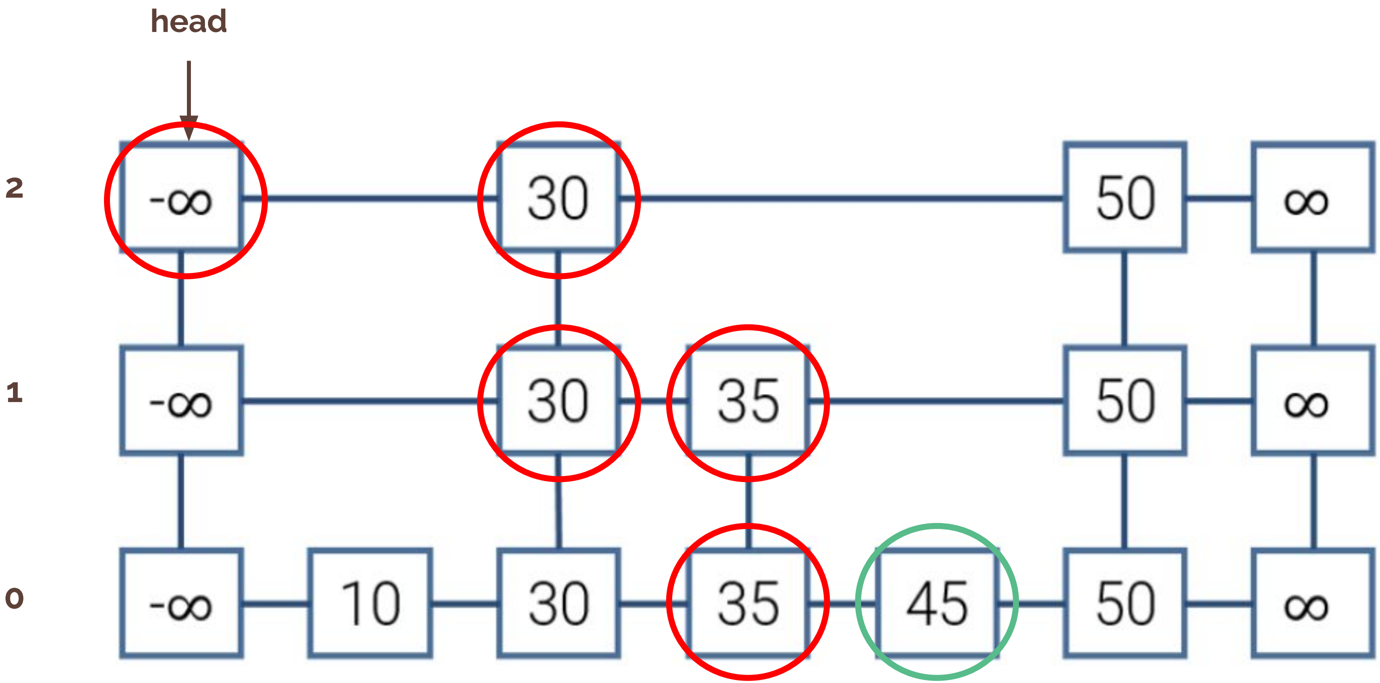
SkipList: Practice

Trace the path to 45. Include nodes that we actually reach, not nodes we only compare with.



SkipList: Practice

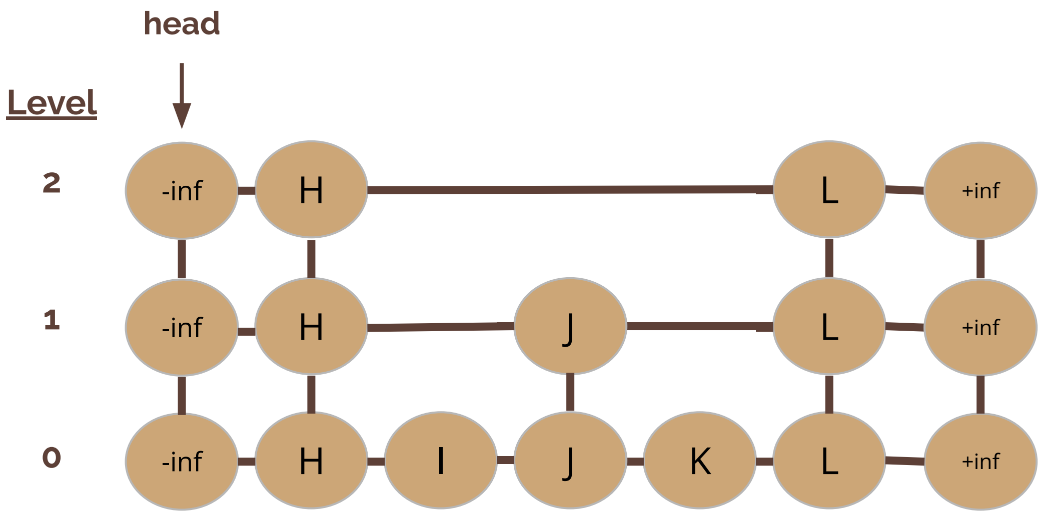
Trace the path to 45. Include nodes that we actually reach, not nodes we only compare with.



SkipList: Add

PROCEDURE

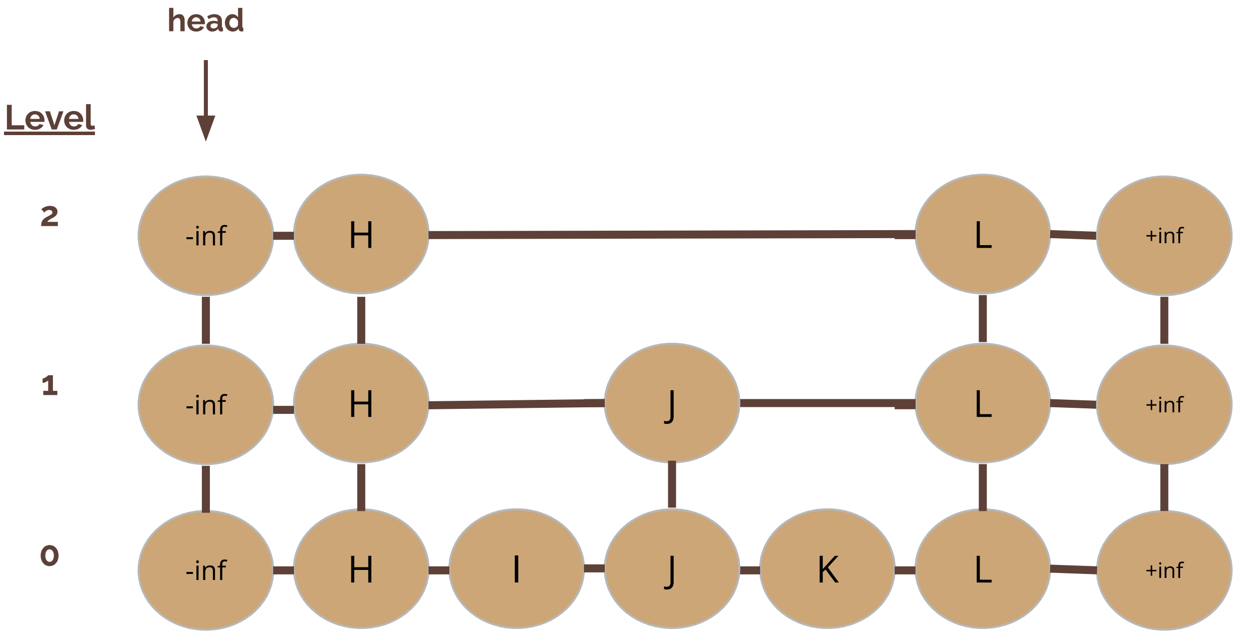
1. Search for the data. When we run into a null node, add the data to the right of our current node.
2. **We use randomness in determining how many levels to promote the new data:**



Flip a coin repeatedly, stop when we flip a Tails.

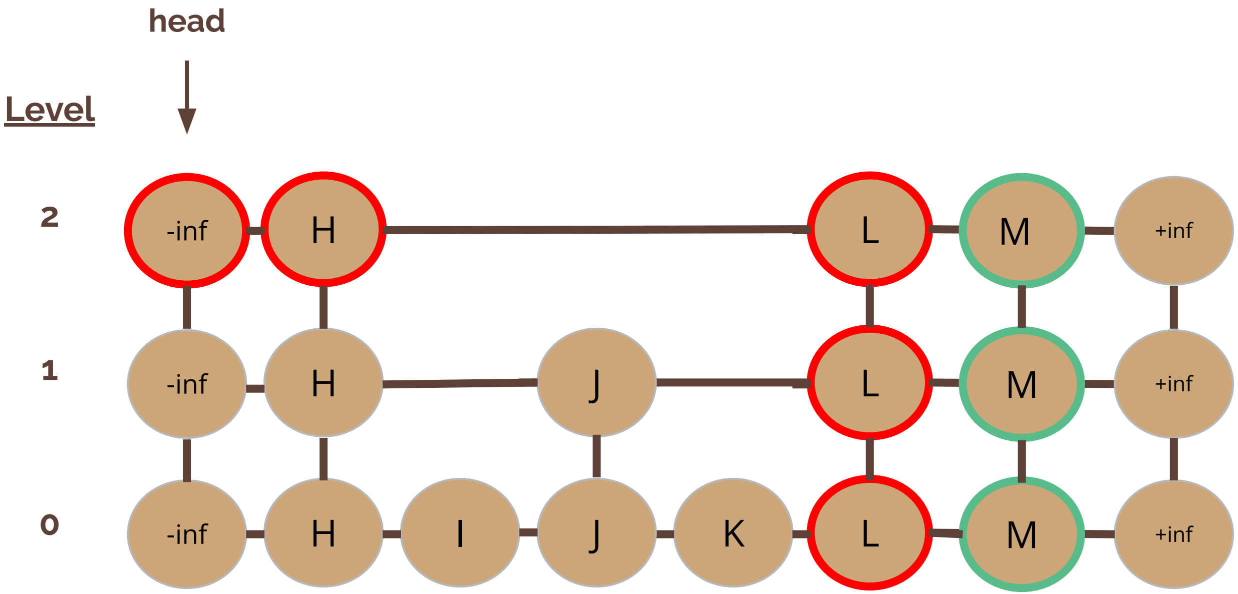
# of heads = # of levels to promote the data

SkipList: Add



Add “M” with the following coin flip:
HHT

SkipList: Add

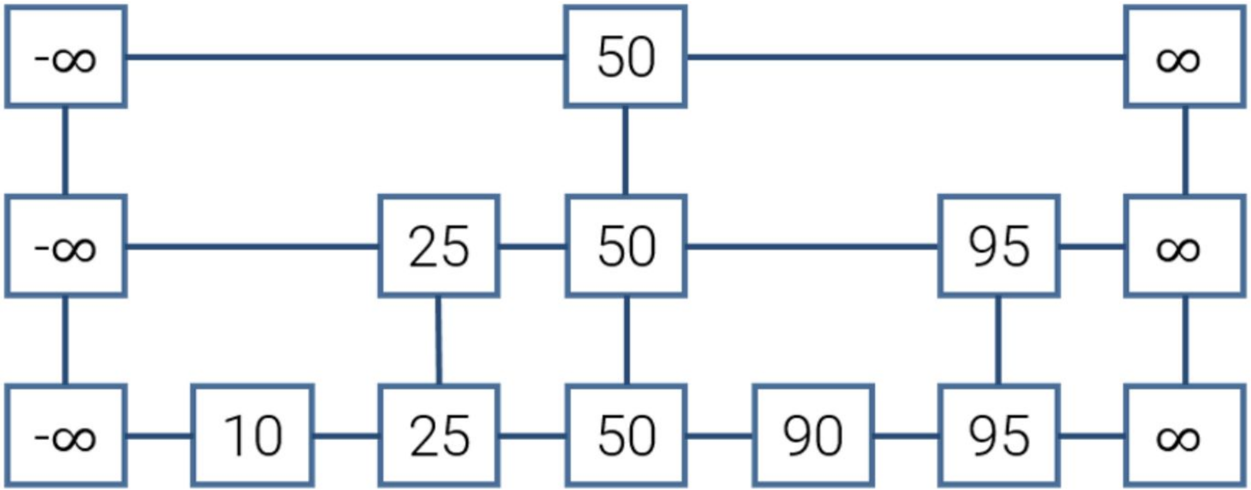


Add “M” with the following coin flip:

HHT

SkipList: Add

Add 10, 90, 25, 50, 95 to an empty SkipList with the following coin flip.
TTHTHHHTHTT

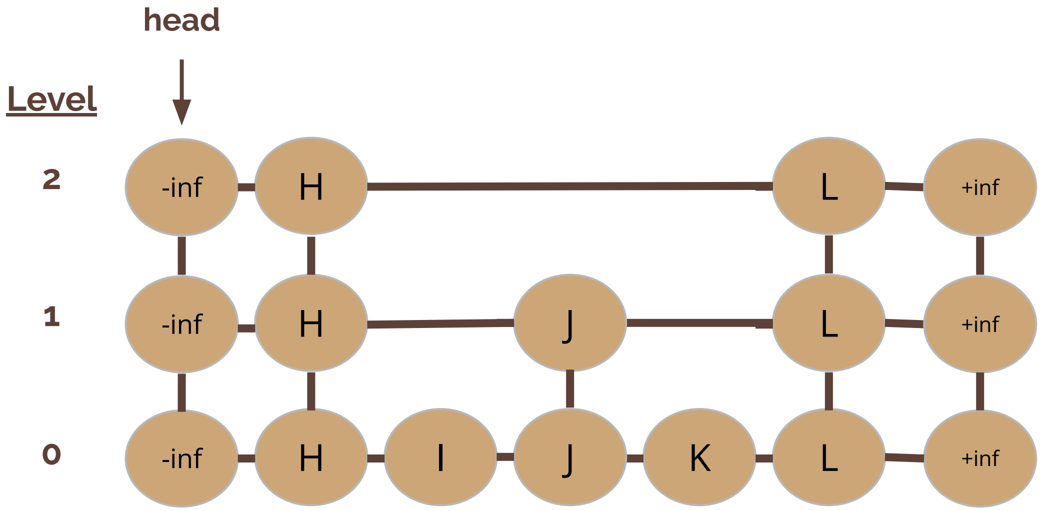


SkipList: Remove

PROCEDURE

1. Search for the data.
2. When we find the data, remove the node from the current level and all levels below it - same as removing from a DLL.

No randomness involved in removing.



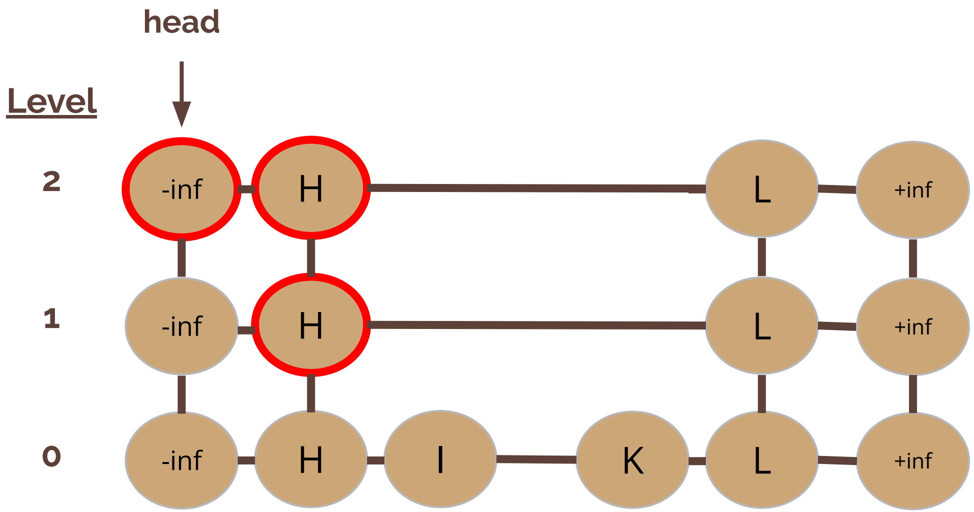
Remove "J".
What nodes do we access?

SkipList: Remove

PROCEDURE

1. Search for the data.
2. When we find the data, remove the node from the current level and all levels below it - same as removing from a DLL.

No randomness involved in removing.



Remove "J".
What nodes do we access?

(-inf, 2), (H, 2), (H, 1), (J, 1), (J, 0)

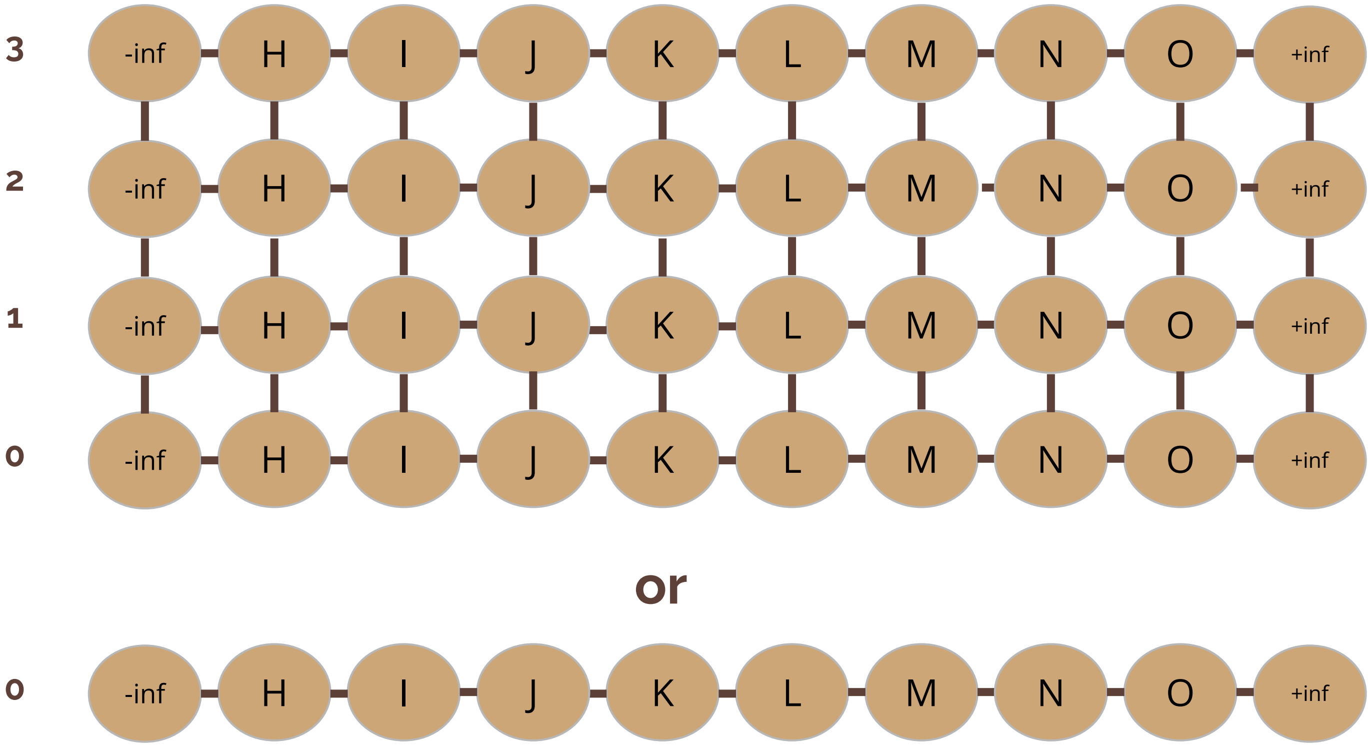
SkipList: Efficiencies

	Best/Average Case	Worst Case
Searching	*O(log(n))*	*O(n)*
Adding	*O(log(n))*	*O(n)*
Removing	*O(log(n))*	*O(n)*
Space	*O(n)*	*O(nlog(n))*

What would the worst case of a SkipList look like?

a LinkedList or a series of a LinkedLists with all of the data

SkipList: Efficiencies



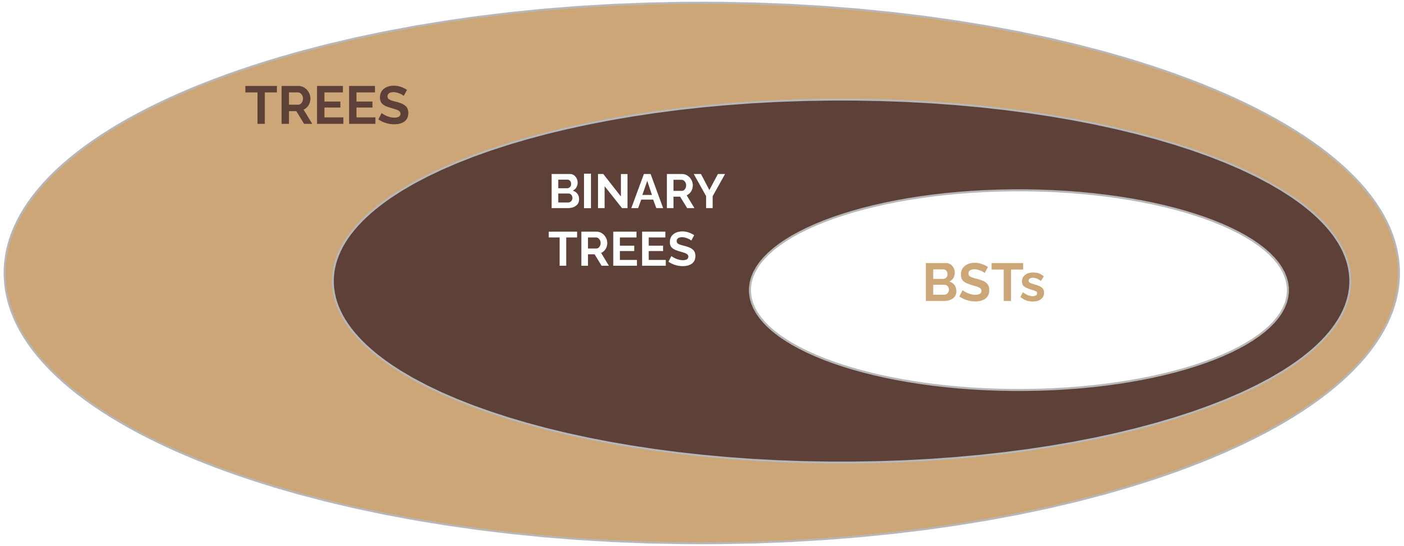
REFRESHER: Binary Search Tree

SHAPE Property

A node cannot have more than two children.

ORDER Property

1. The left child's data must be less than the parent's data.
2. The right child's data must be greater than the parent's data.



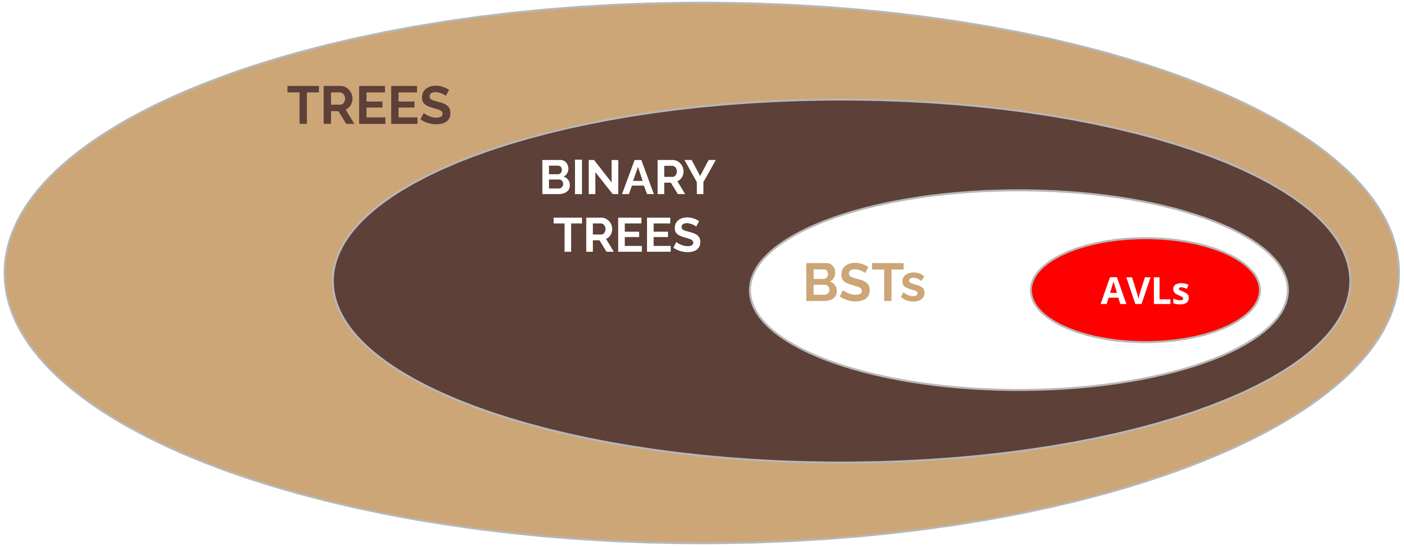
AVL

SHAPE Property

1. A node cannot have more than two children.
2. **Every node in the AVL must be balanced.**

ORDER Property

1. The left child's data must be less than the parent's data.
2. The right child's data must be greater than the parent's data.



AVL

➔ Solves the problem of the O(n) degenerate case in a BST because ***a balanced tree always has logn levels → always O(logn) search.***

WHAT'S NEW?

1. AVL Nodes store 5 pieces of information:
   a. Data
   b. Left child
   c. Right child
   d. **Height**
   e. **Balance Factor**
2. **Balance Factor = node.left.height - node.right.height**

A node is considered balanced if -1 <= node.bf <= 1.

AVL: Rotations

What is a rotation?

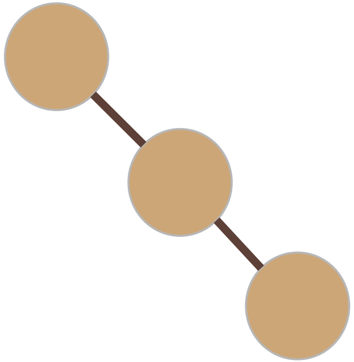
- A set of operations that re-balances a tree
- Implemented as methods
- **4 TYPES:**
  - **Left**
  - **Right**
  - **Left-Right**
  - **Right-Left**

AVL: Rotations

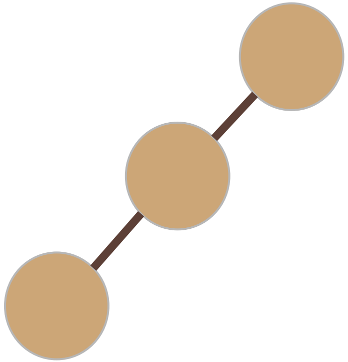
When do we rotate?

- ➔ Performed in add() and remove() methods to restructure the tree
- ➔ **CONDITION: Node.BF == 2 or Node.BF == -2**

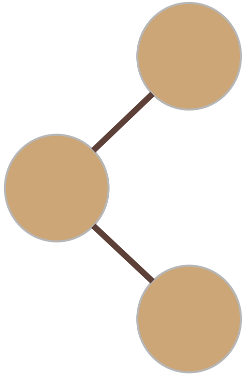
LEFT



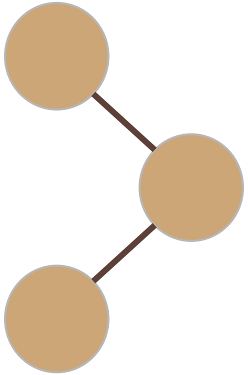
RIGHT



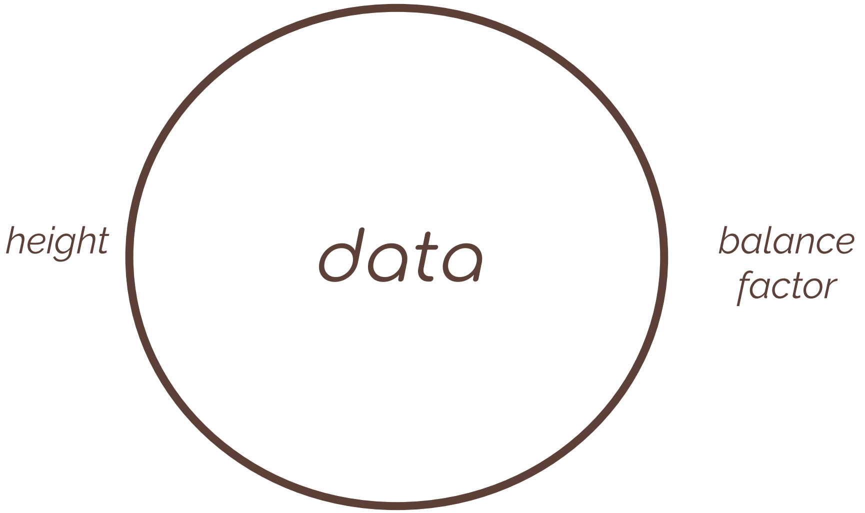
LEFT-RIGHT



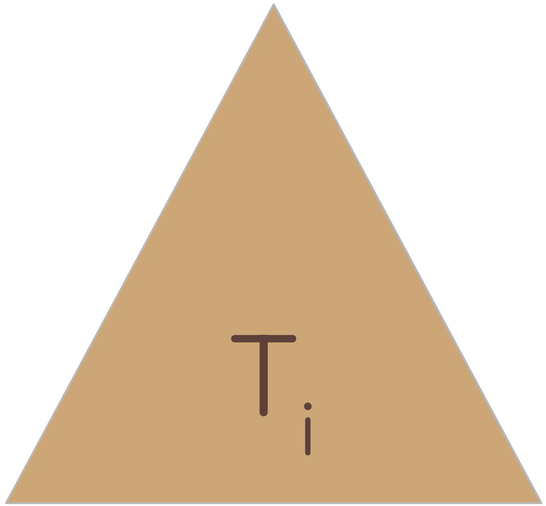
RIGHT-LEFT



NODE

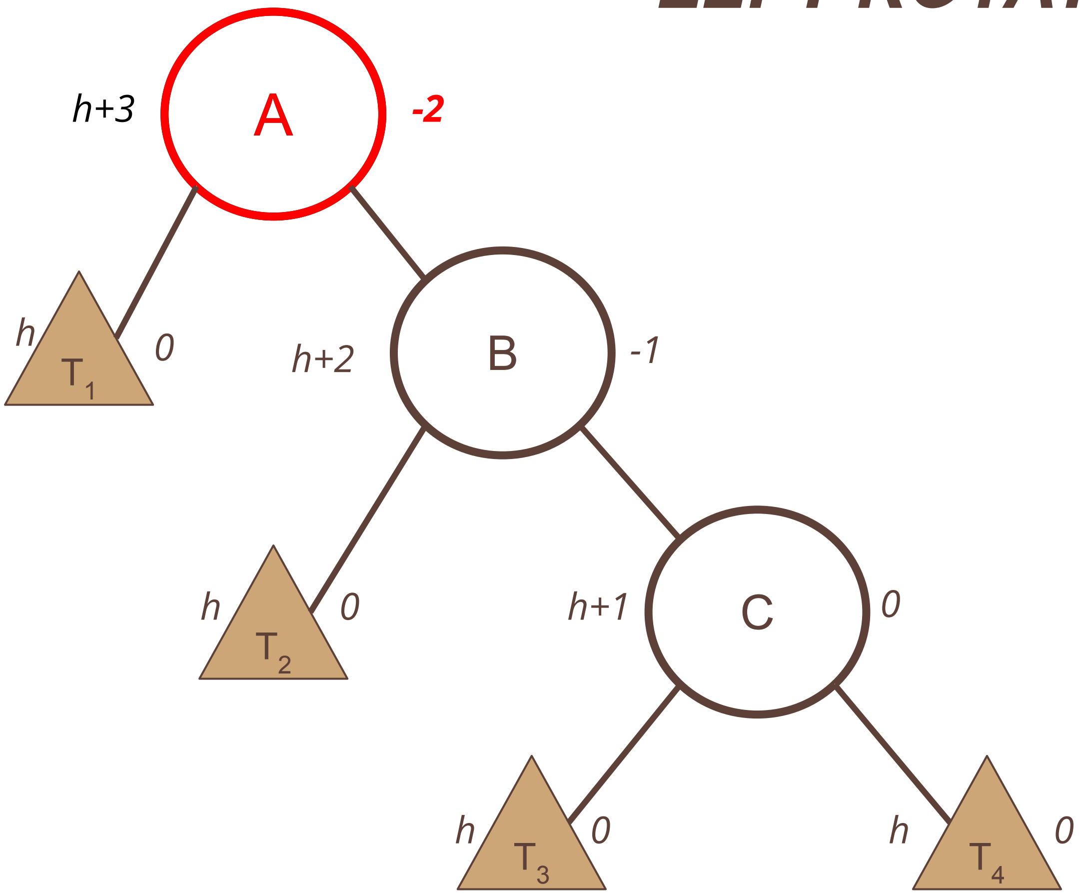


TREE



A balanced tree
of some height h

LEFT ROTATION



Conditions:

1. A.balanceFactor == -2
2. B.balanceFactor == -1

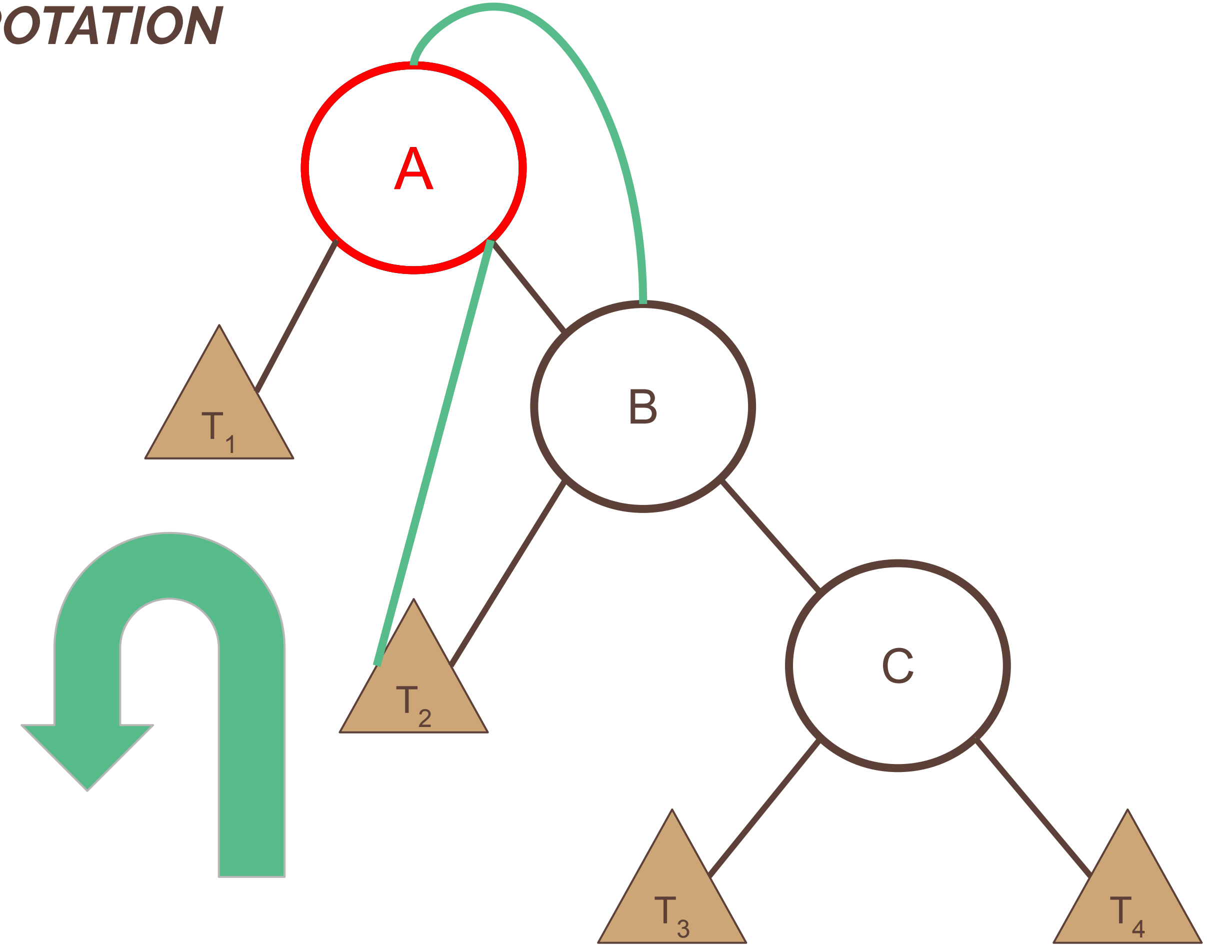
In words:

1. A is right-heavy beyond our limit.
2. A's right child is right-heavy.

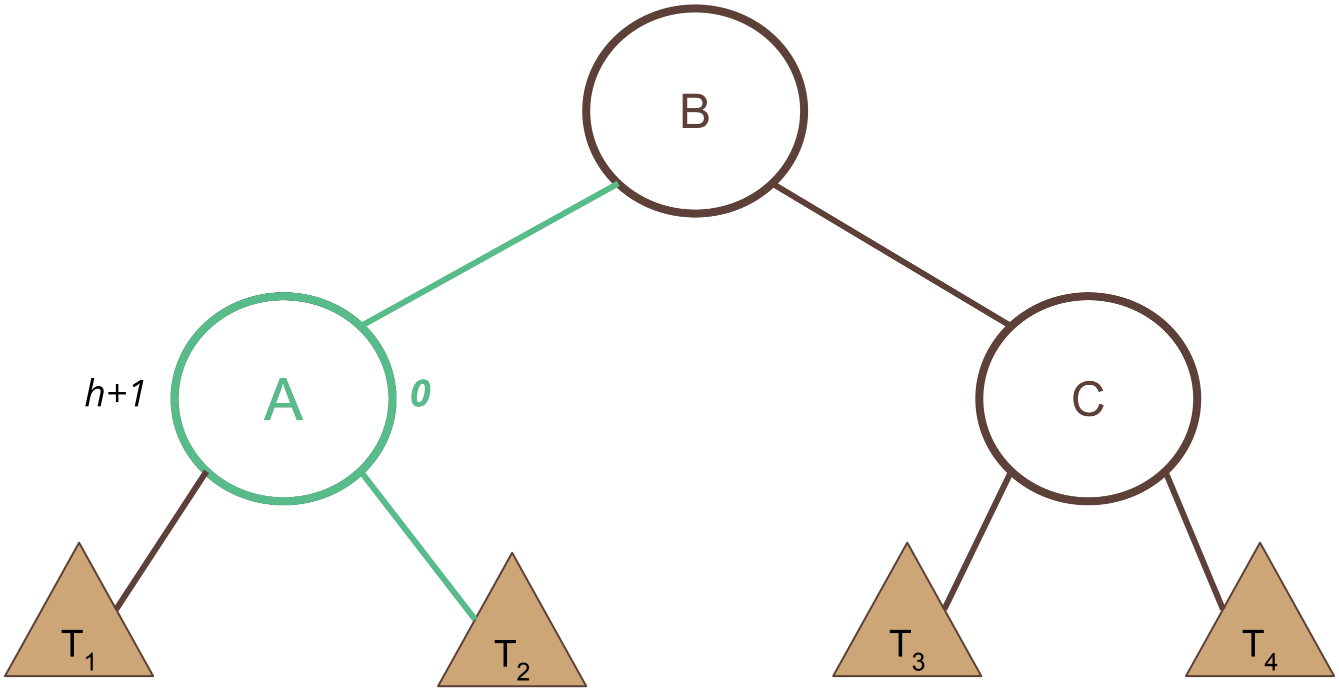
↓

LEFT ROTATION

LEFT ROTATION



LEFT ROTATION



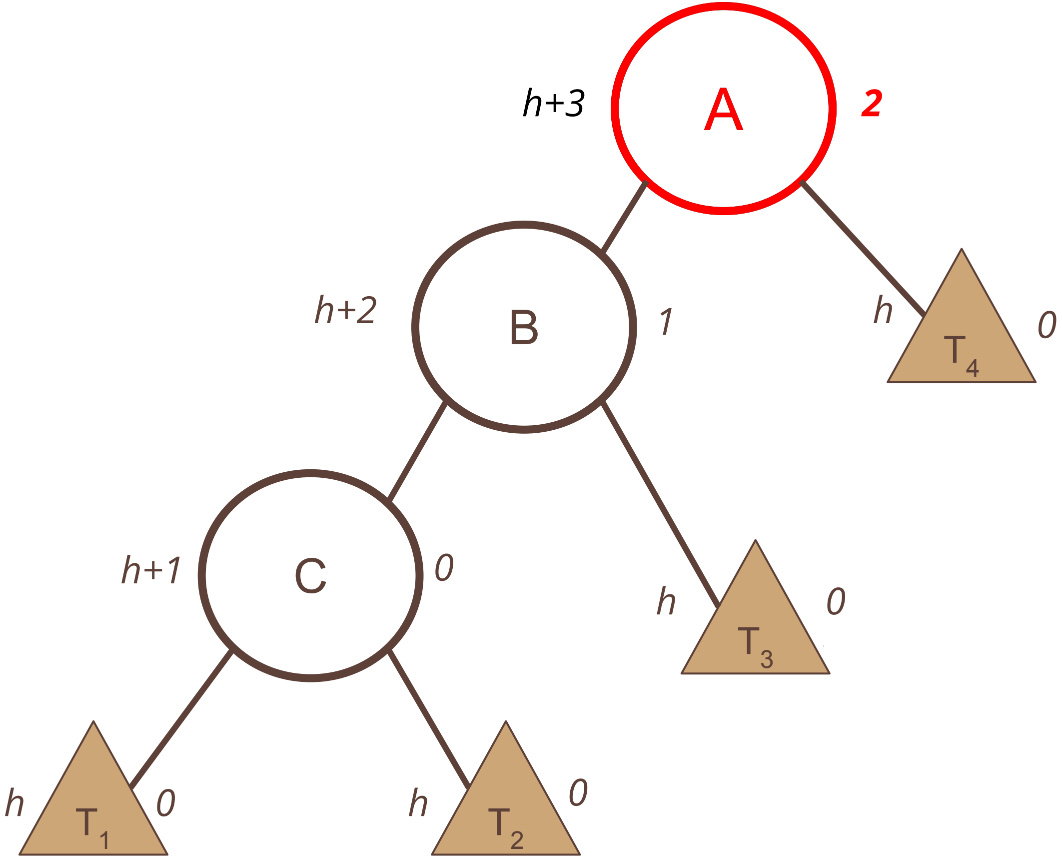
LEFT ROTATION: Implementation

1. Set A's right child to B's left child.
2. Set B's left child to A.
3. Updated A's height and balance factor.
4. Updated B's height and balance factor.

PSEUDOCODE

```
Algorithm leftRotation(A)
    B = A.right
    A.right = B.left
    B.left = A
    update(A)
    update(B)
    return B
```

RIGHT ROTATION



Conditions:

1. A.balanceFactor == 2
2. B.balanceFactor == 1

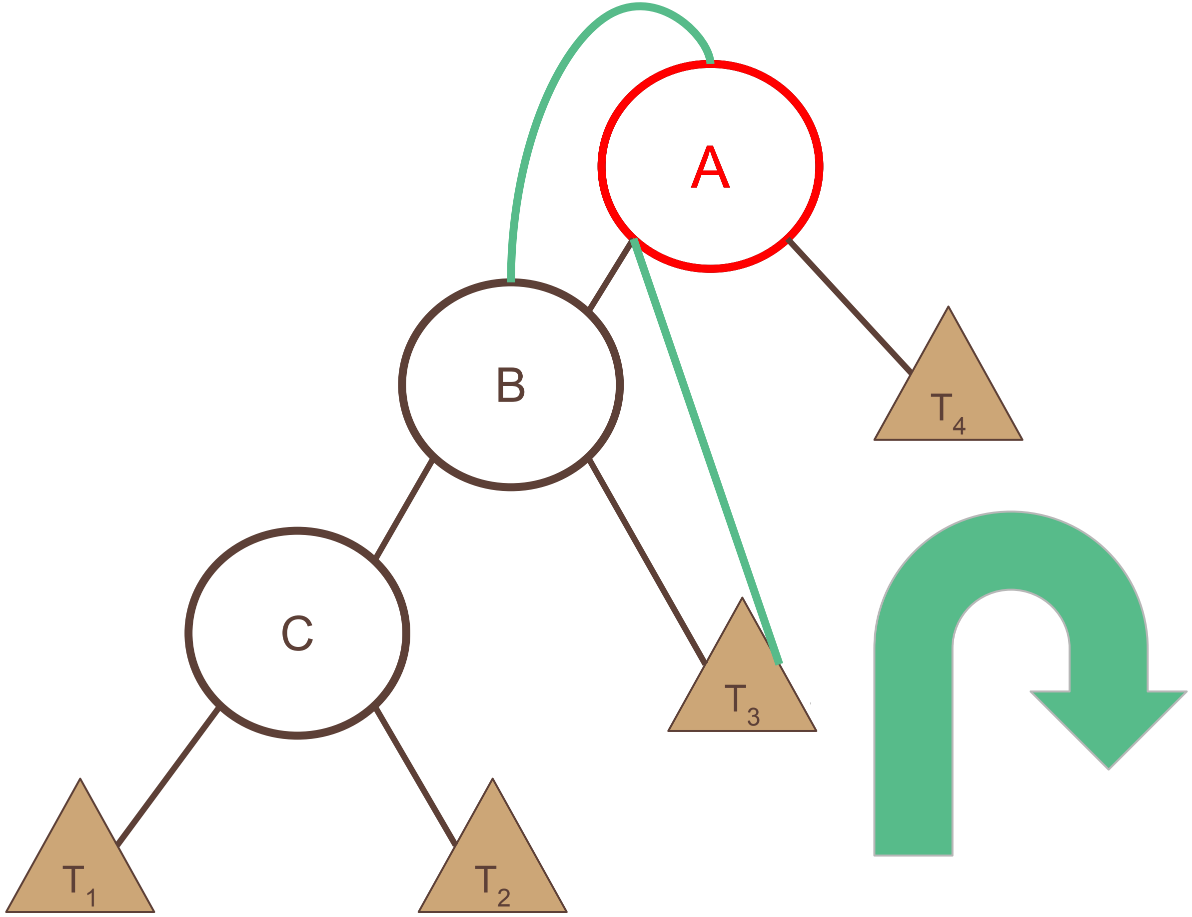
In words:

1. A is left-heavy beyond our limit.
2. A's left child is right-heavy.

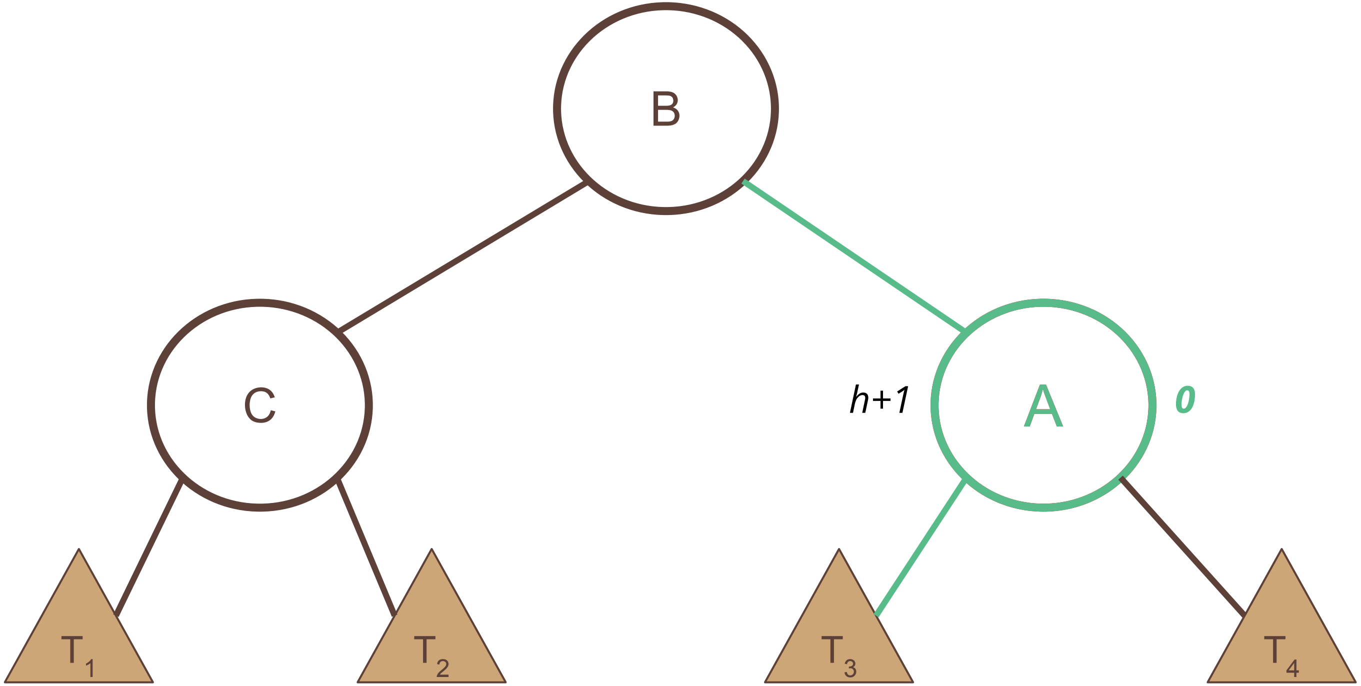
↓

RIGHT ROTATION

RIGHT ROTATION



RIGHT ROTATION



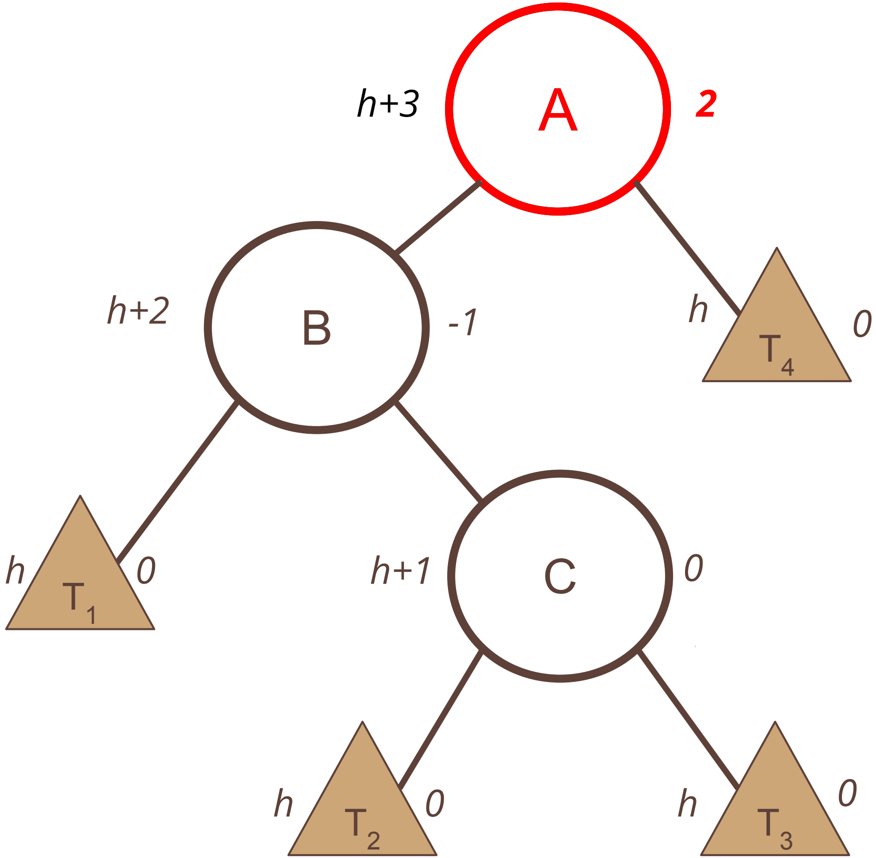
RIGHT ROTATION: Implementation

1. Set A's *left* child to B's *right* child.
2. Set B's *right* child to A.
3. Updated A's height and balance factor.
4. Updated B's height and balance factor.

PSEUDOCODE

```
Algorithm rightRotation(oldRoot):
    newRoot = oldRoot.left
    oldRoot.left = newRoot.right
    newRoot.right = oldRoot
    update(oldRoot)
    update(newRoot)
    return newRoot
```

LEFT-RIGHT ROTATION



Conditions:

1. A.balanceFactor == 2
2. B.balanceFactor == -1

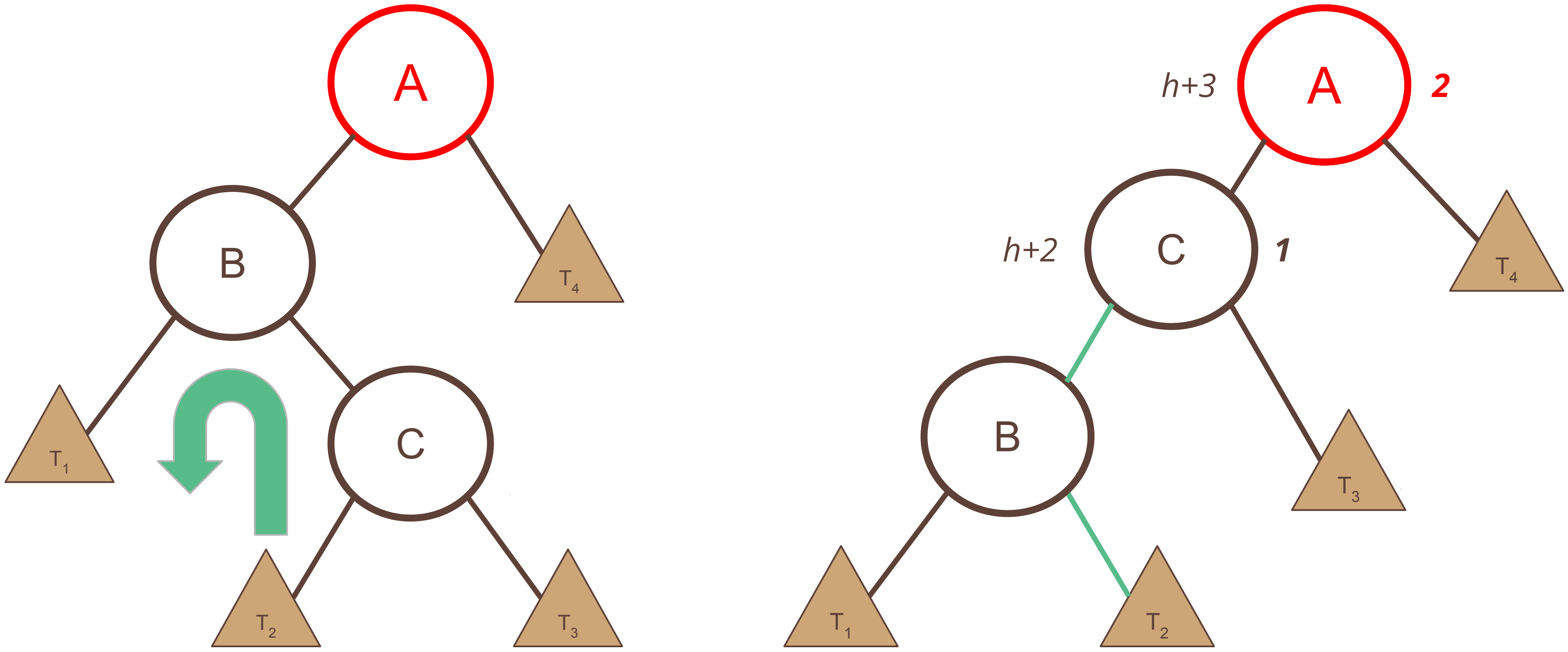
In words:

1. A is *left*-heavy beyond our limit.
2. A's left child is *right*-heavy.

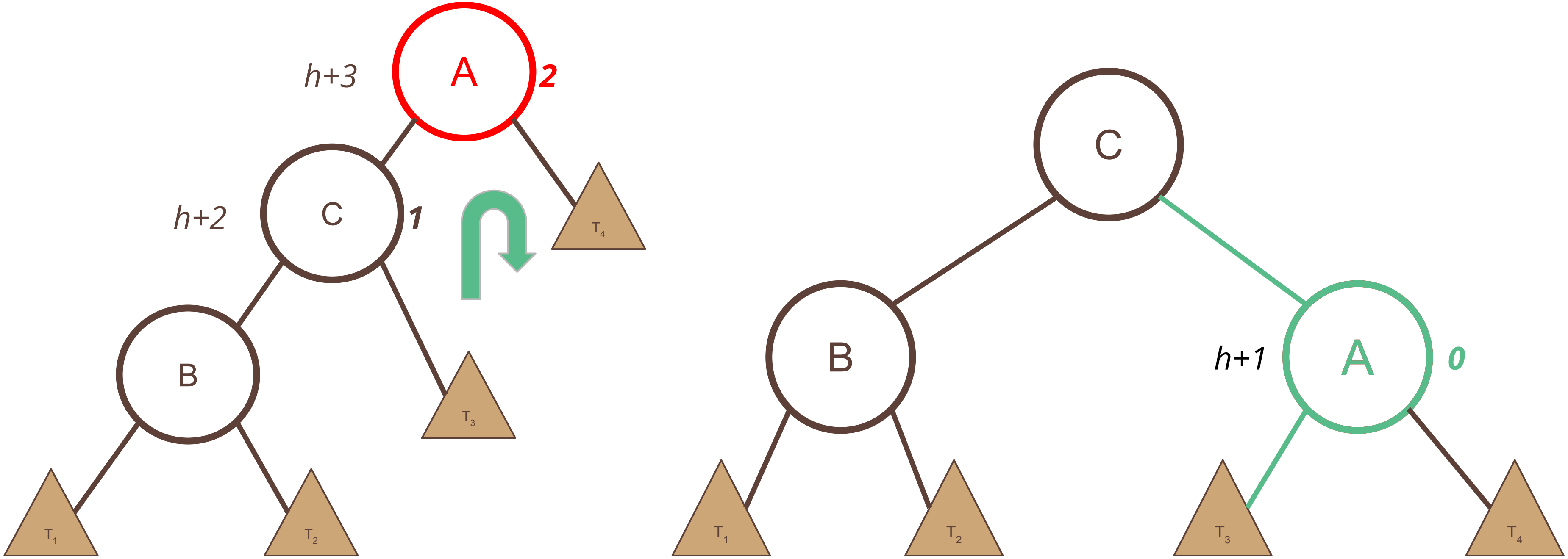
↓

LEFT-RIGHT ROTATION

LEFT-RIGHT ROTATION: Part 1



LEFT-RIGHT ROTATION: Part 2



LEFT-RIGHT ROTATION: Implementation

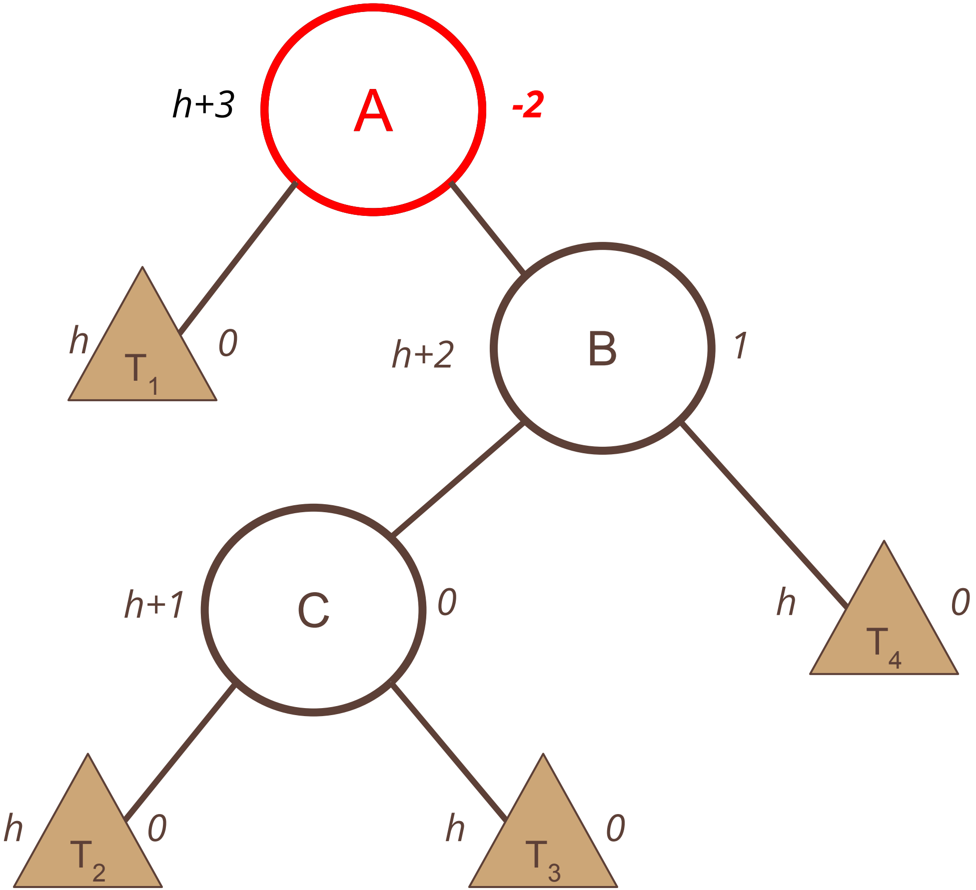
1. Performed a left-rotation on A's left child, B.

2. Performed a right-rotation on A.

PSEUDOCODE

```
Algorithm leftRight(A):
    A.left = leftRotation(A)
    A = rightRotation(A)
```

RIGHT-LEFT ROTATION



Conditions:

1. A.balanceFactor == -2
2. B.balanceFactor == 1

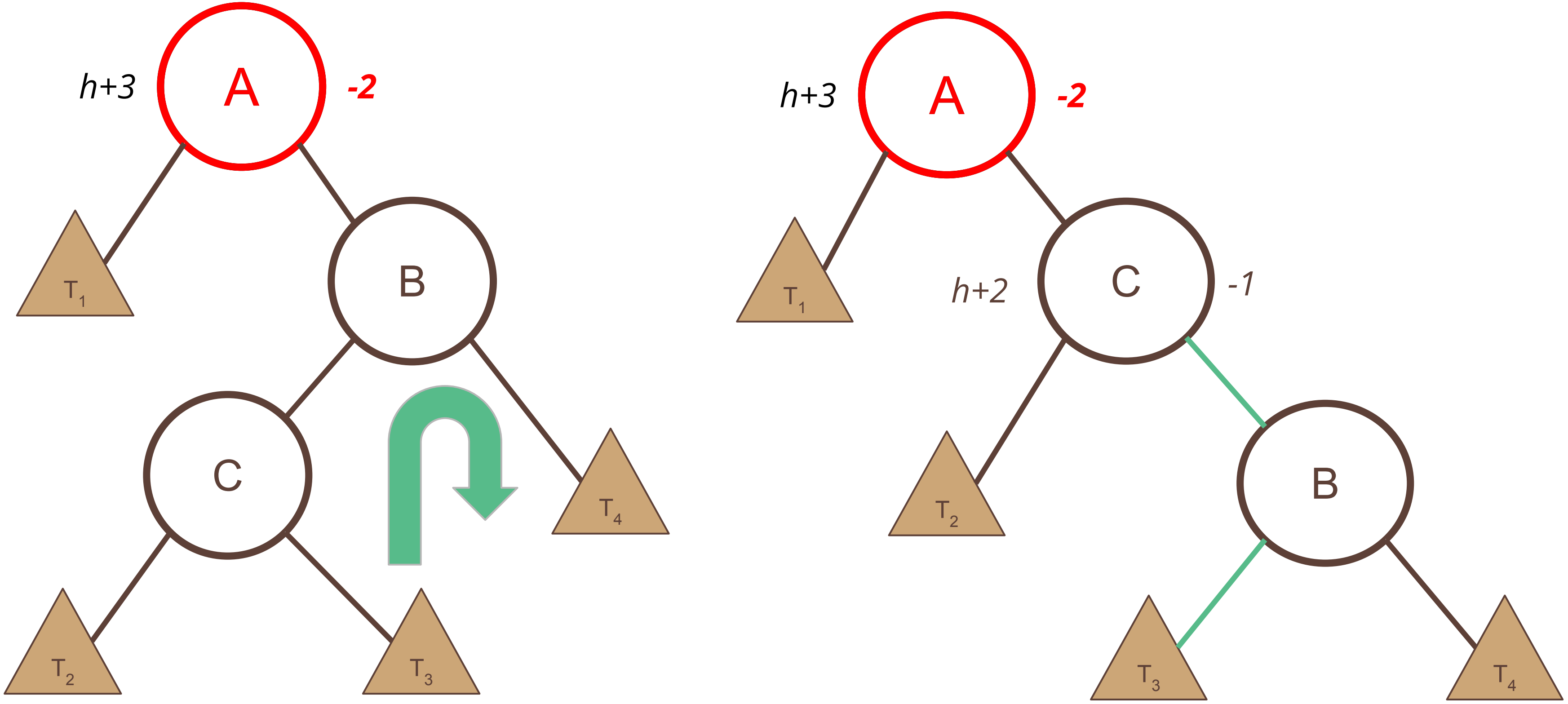
In words:

1. A is *right*-heavy beyond our limit.
2. A's right child is *left*-heavy.

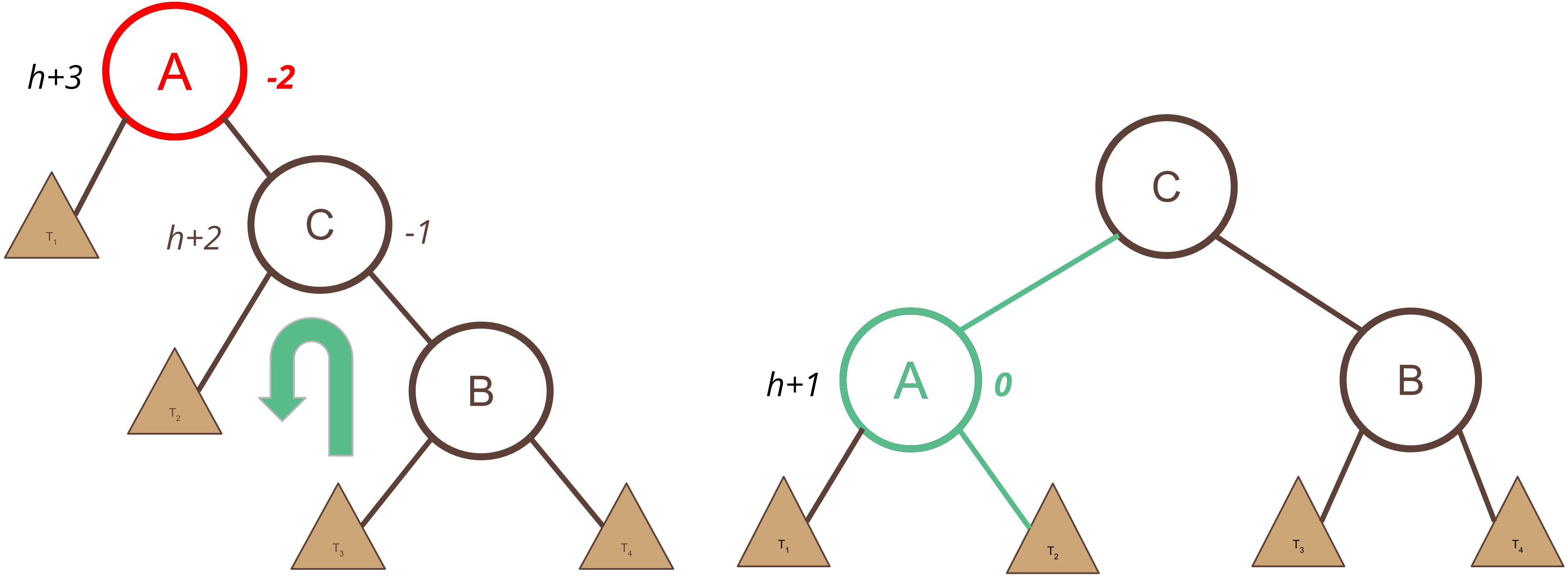
↓

RIGHT-LEFT ROTATION

RIGHT-LEFT ROTATION: Part 1



RIGHT-LEFT ROTATION: Part 2



RIGHT-LEFT ROTATION: Implementation

1. Performed a right-rotation on A's right child, B.

2. Performed a left-rotation on A.

PSEUDOCODE

```
Algorithm rightLeft(A):
    A.right = rightRotation(A)
    A = leftRotation(A)
```

AVL: Rotations

Parent BF	Heavier Child	Child BF	Rotation Shape	Rotation Type
2	Left	0, 1		Right
2	Left	-1		Left-Right
-2	Right	-1, 0		Left
-2	Right	1		Right-Left

AVL: Practice

When do we have to update a node's height and balance factor?

Any time we alter its children.

Do we update and balance from the bottom-up or top-down?

bottom-up

What is the worst case run time of adding to an AVL where the balance factor can range from -n to n?

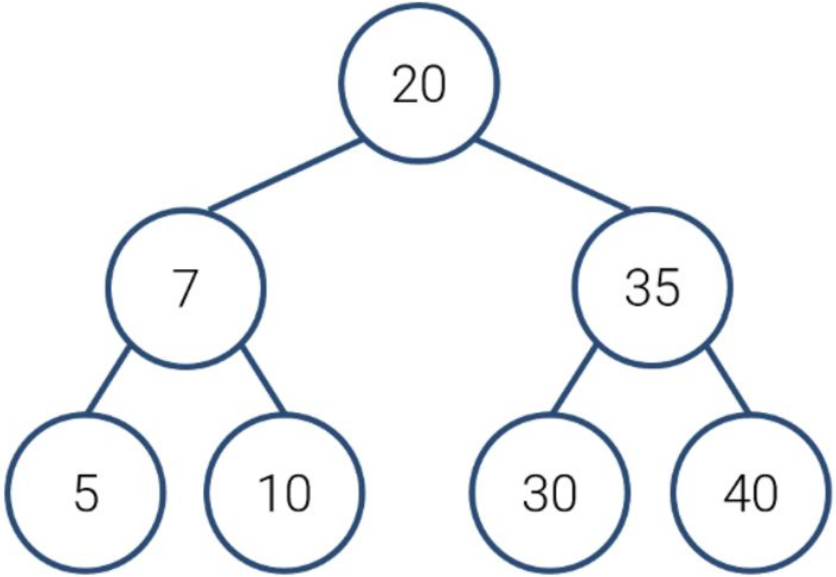
O(n)

What is the worst case run time of calculating the height of an AVL?

O(1)

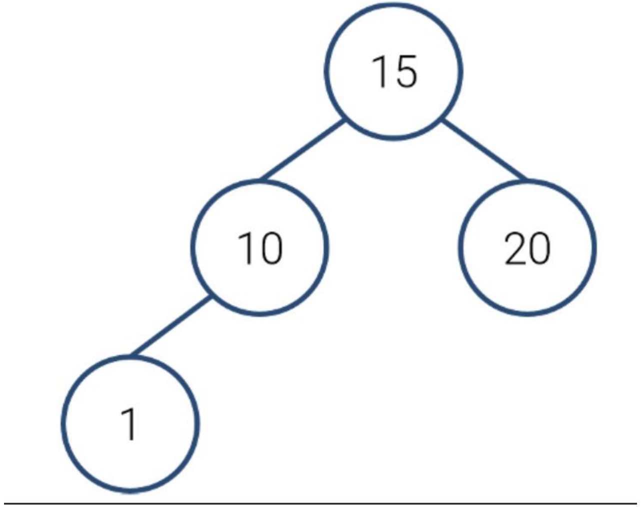
AVL: Add Practice

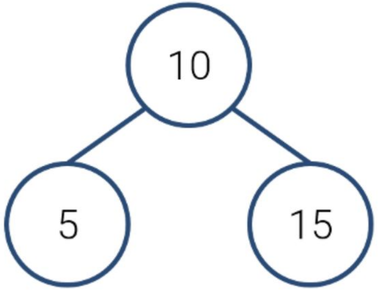
Create an AVL by adding the data in order:
10, 30, 20, 5, 7, 40, 35



AVL: Remove Practice

Add 5 to the AVL below. Then remove 20, then 1.
Use the predecessor if necessary.





AVL: Efficiencies

	Adding	Removing	Accessing	Height	Traversals
Average	*O(log n)*	*O(log n)*	*O(log n)*	*O(1)*	*O(n)*
Worst	*O(log n)*	*O(log n)*	*O(log n)*	*O(1)*	*O(n)*

All rotations are **O(1)**.

Rotations do not depend on the size of any tree or subtree involved.

LEETCODE PROBLEMS

[105. Construct Binary Tree from Preorder and Inorder Traversal]

[1206. Design Skiplist]

Any questions?

Name
Office Hours
Contact

Name
Office Hours
Contact

Let us know if there is anything specific you want out of recitation!