CS 261-020
Data Structures

Lecture 11

AVL Trees

2/22/24, Thursday

Oregon State University

Odds and Ends

- This Friday is the last day to demo your assignment 2 w/o penalty!
- Due Sunday 2/25 11:59 pm:
  - Assignment 3
- Questions?

Lecture Topics:

- AVL Trees
  - Self-balancing BST

The Balance of BSTs

- Balance of BSTs:
  - All nodes have depths approximately log(n) or less

- Balance is important – primary operations on BSTs all have O(h) runtime complexity, where h is the height of the tree.

O(logn)

- With balanced BST, h → log(n), then O(h) will be fast
- With unbalanced BST, h → n, then O(h) will be slow

→ O(n)

- Problem: plain BSTs cannot ensure itself is balanced

Height Balance

- *Height Balance*: a measurable form of BST balance

- A BST is height balanced if, at every node in the tree, the subtree heights of the node's left and right subtrees differ by at most 1

- A height-balanced BST is guaranteed to have an overall height that's within a constant factor of log(n)
  - operations in a height-balanced BST are guaranteed to have *O(log n)* runtime complexity.

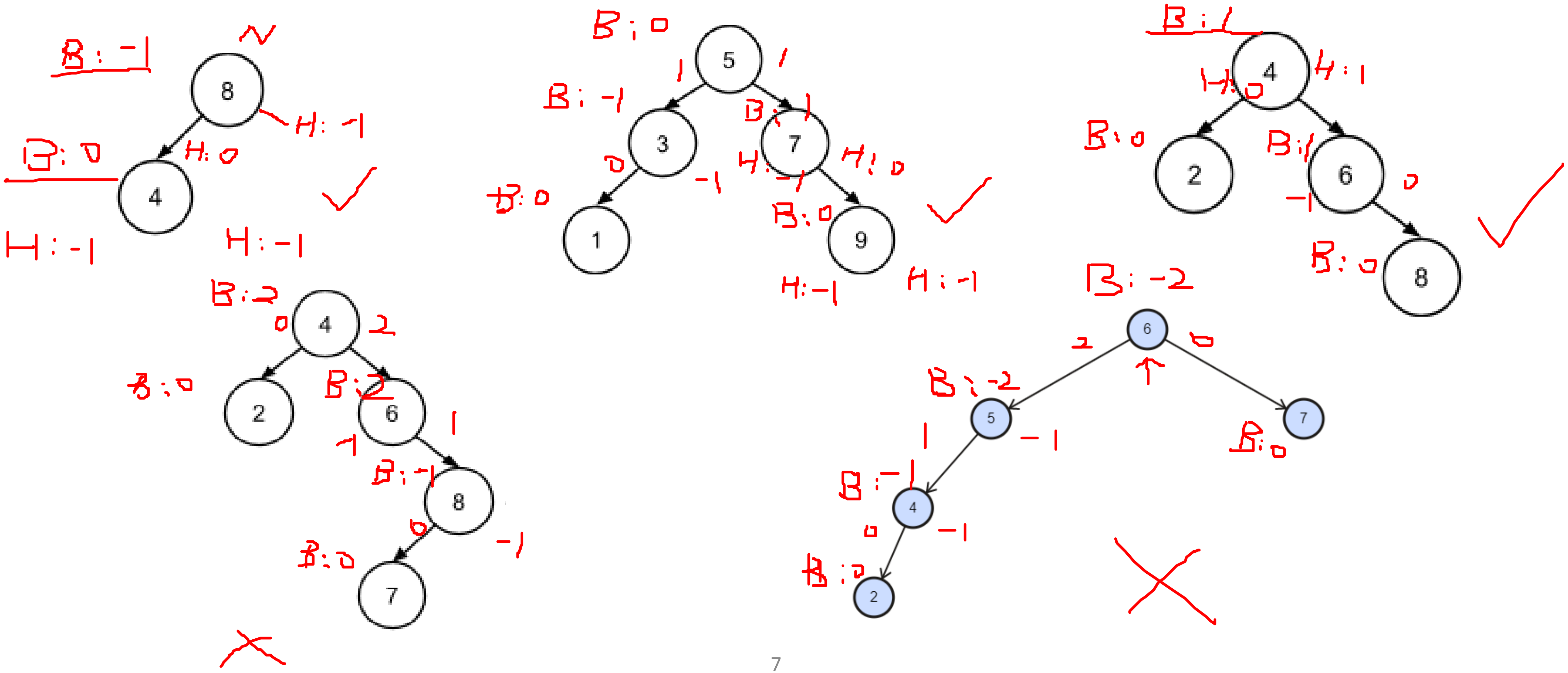
Balance Factor

- A BST node's *balance factor* – a metric to figure out whether the subtree rooted at that node is height balanced.

- the balance factor of the node N:
  - **`balanceFactor(N) = height(N.right) - height(N.left)`**
  - the height of a NULL node (i.e. an empty subtree) is -1

Height Balance and Balance Factor

- Height-balanced, or un-balanced? Write down balance factor for each node.



Restructuring AVL Trees via Rotations

- The AVL tree is one of several existing types of self-balancing BST.
  - AVL is derived from the initials of the names of the tree's inventors: Adelson-Velsky and Landis.
  - Another popular one is the red-black tree.

- An AVL tree's operations include mechanisms to ensure that the tree always exhibits height balance
  - check the height balance of the tree after each insertion and removal
  - perform rebalancing operations known as *rotations* whenever height balance is lost

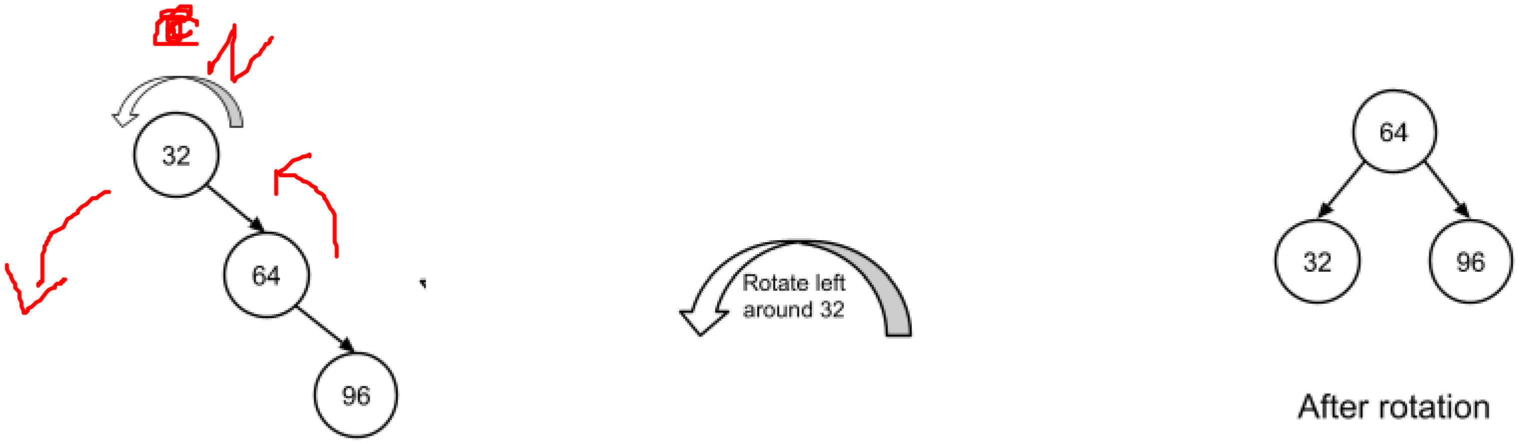
Restructuring AVL Trees via Rotations

- A rotation: an operation that restructures an isolated region of the tree by performing a limited number of *pointer updates* that result in one node moving “upwards” in the tree and another node moving “downwards.”
  - preserve the BST property among all nodes in the tree

- Sometimes, a single rotation will be enough to restore height balance.
- Sometimes, a double rotation will be needed.

Restructuring AVL Trees via Rotations

- Each rotation has a center and a direction
- The center is the node at which the rotation is performed
- Direction: perform either a left rotation or a right rotation around this center node
  - A left rotation moves nodes in a "counterclockwise" direction, with the center moving downwards and nodes to its right moving upwards.
  - A right rotation moves nodes in a "clockwise" direction, with the center moving downwards and nodes to its left moving upwards



Before rotation

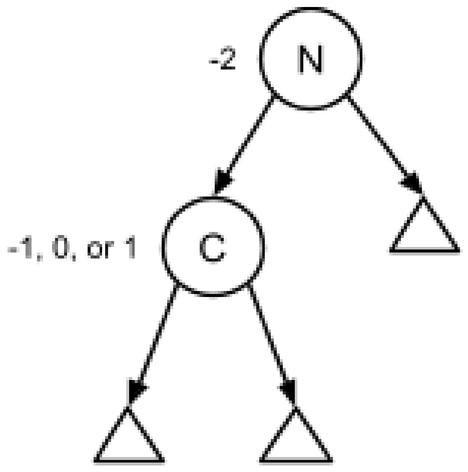
After rotation

How to rotate?

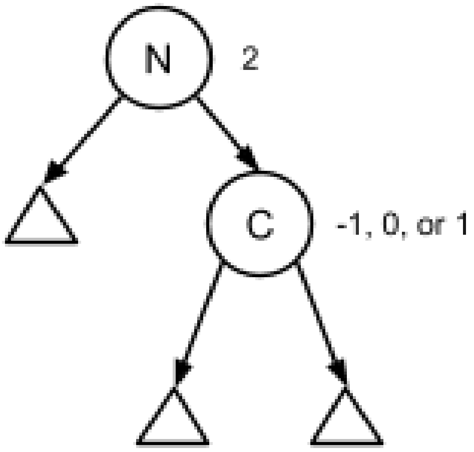
- A rotation (i.e. single or double) will be needed any time an insertion into or removal from an AVL tree that leaves the tree (temporarily) with a node whose balance factor is either -2 or 2
- In other words, a rotation is needed when height balance is lost at a specific node in the tree. Let's call this node N.

- If N has a balance factor of -2, this means N is left-heavy.
- If N has a balance factor of 2, this means N is right-heavy.
- Regardless of the direction of N's heaviness, let's refer to the heavier of N's children as C

- The node C itself will have a balance factor of -1, 0, or 1



N left-heavy



N right-heavy

In class activity: How to rotate?

- Get into small groups, on the worksheet, for each unbalanced tree,
  - Determine whether a single rotation / a double rotation is needed
  - draw the height-balanced BSTs after rotating

- Can you generalize the situations when a single rotation is needed?

- Can you generalize the situations when a double rotation is needed?

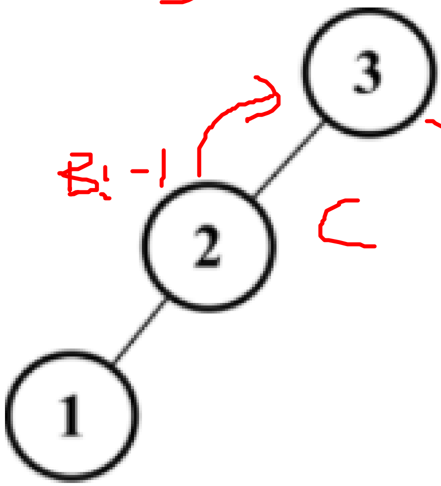
In class activity: How to rotate?

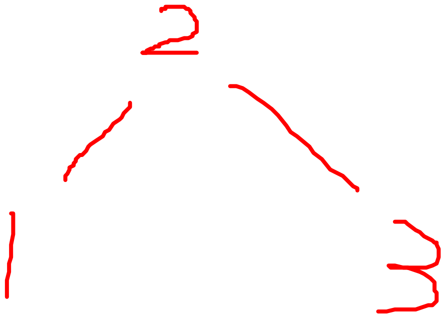
B: -2

N

B: -1

C

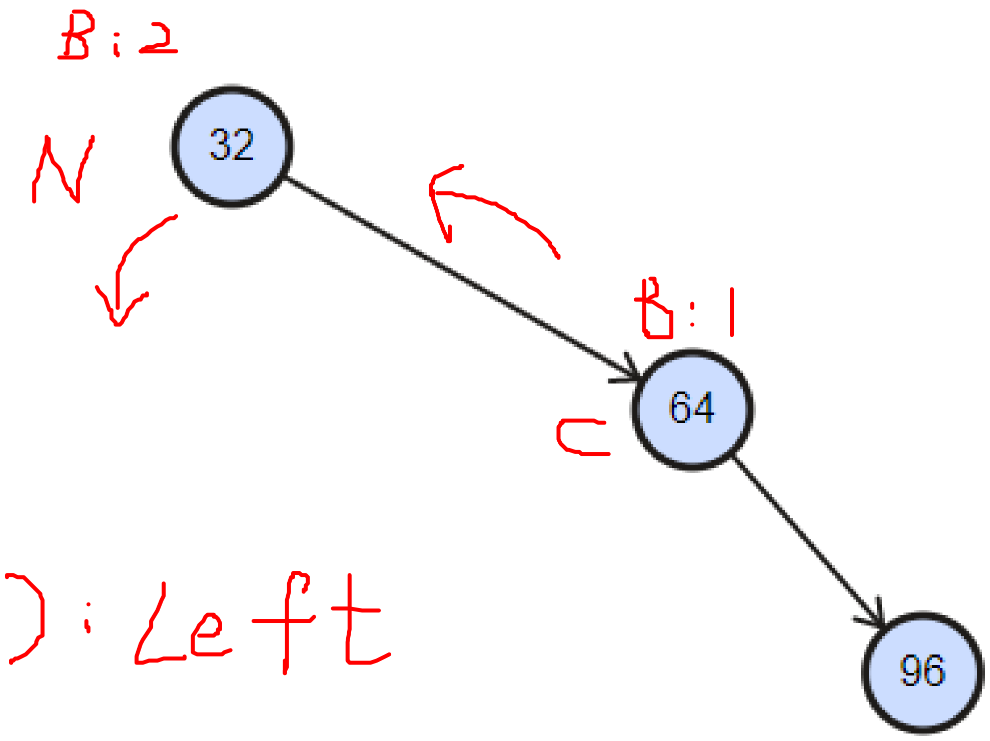




D: Right

C: 3

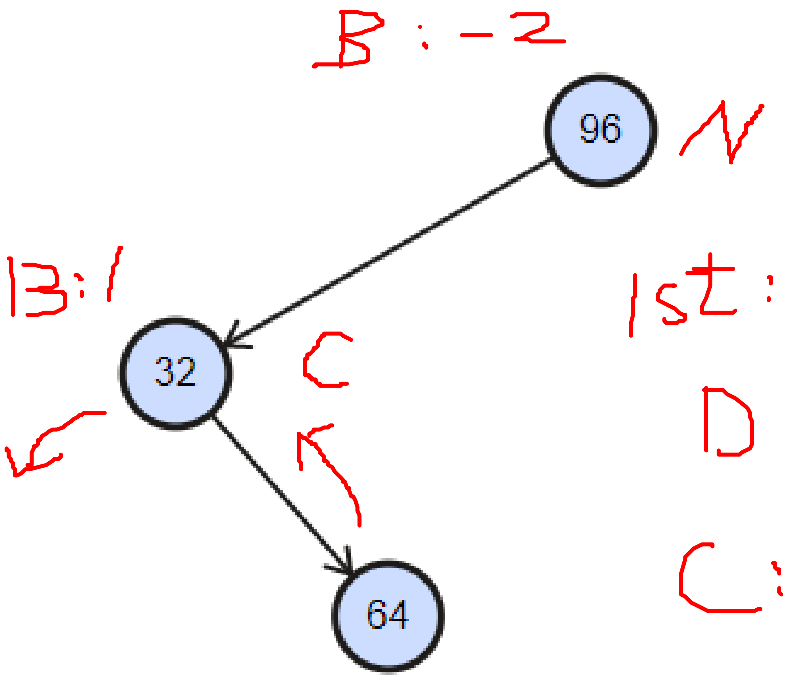
In class activity: How to rotate?



D: Left

C: 32

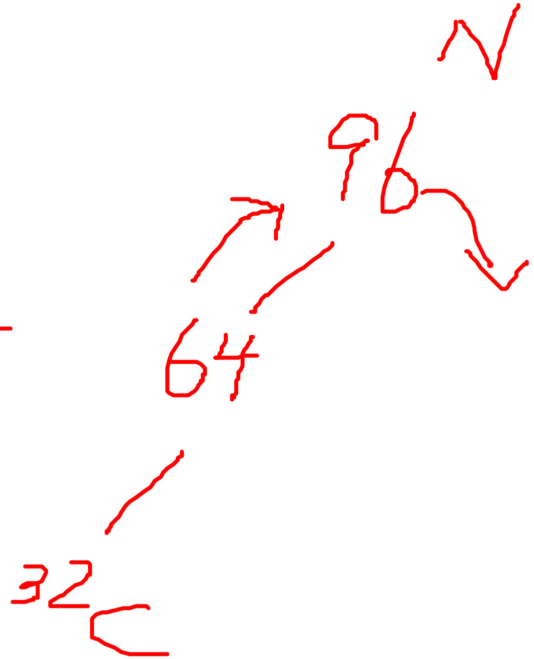
In class activity: How to rotate?



1st:

D: left

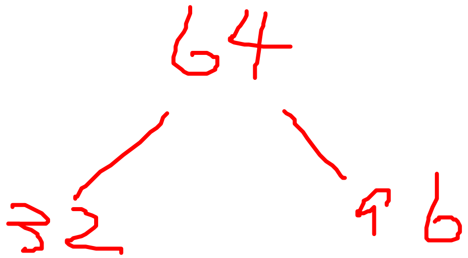
C: 32



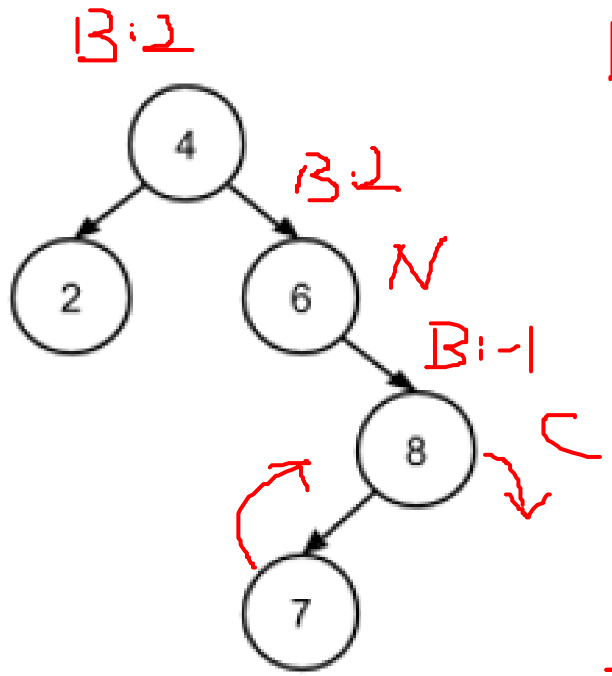
2nd:

D: right

C: N



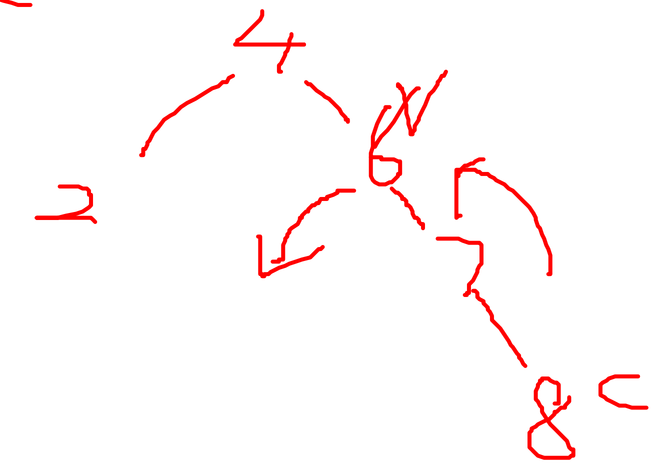
In class activity: How to rotate?



1st:

D: right

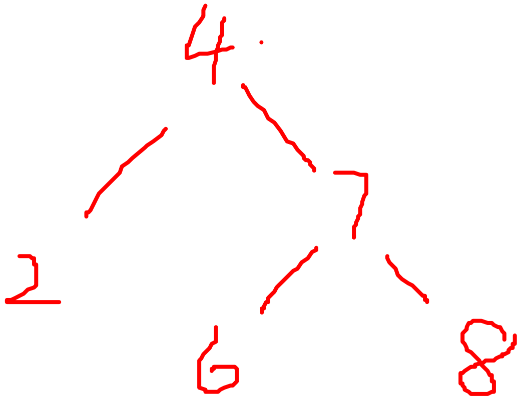
C: 8



2nd:

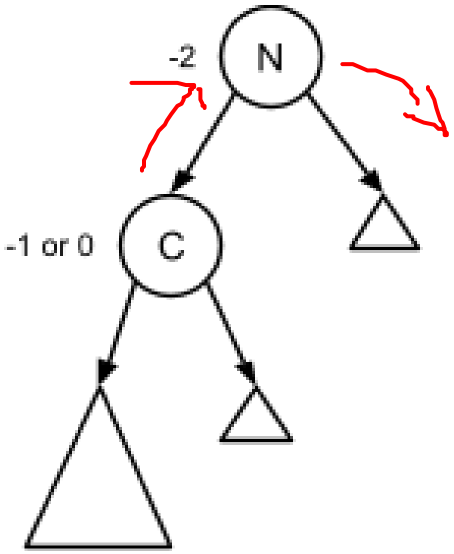
D: left

C: 6

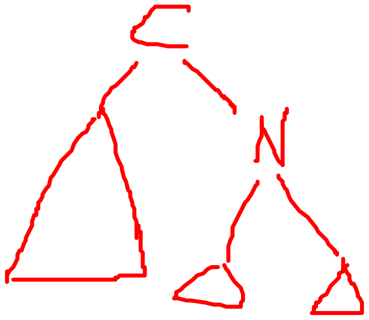


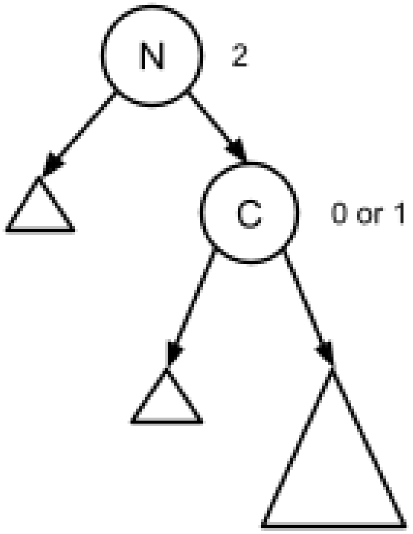
Single vs. Double Rotation

- If ***N*** **and** ***C*** **are heavy in the same direction**, then a single rotation is needed around N in the opposite direction as N's heaviness

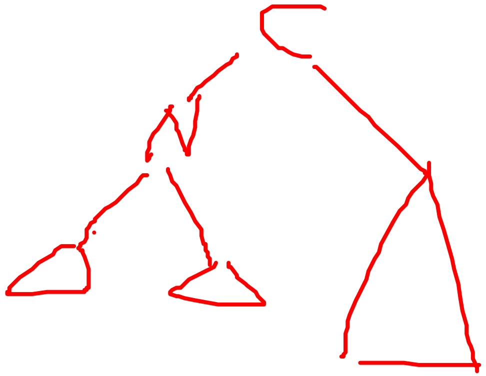


Single right rotation needed



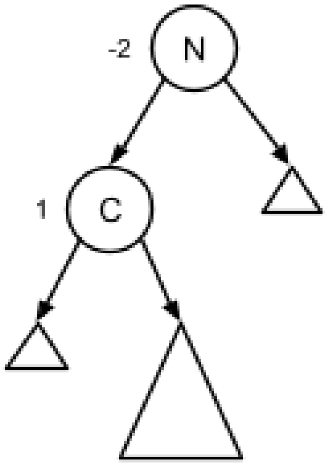


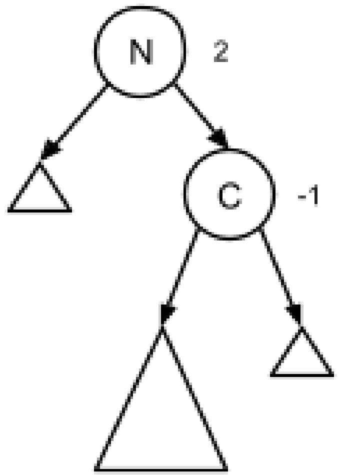
Single left rotation needed



Single vs. Double Rotation

- If ***N* and *C* are heavy in opposite directions**, then a double rotation is needed
  - If N is left-heavy and C is right-heavy, then we first rotate left around C then right around N.
  - If N is right-heavy and C is left-heavy, then we first rotate right around C then left around N





Single vs. Double Rotation

<table>
<tr><td colspan="2"></td><td colspan="2"><code>balanceFactor(N)</code></td></tr>
<tr><td colspan="2"></td><td>-2 (left-heavy)</td><td>2 (right-heavy)</td></tr>
<tr><td rowspan="3"><code>balanceFactor(C)</code></td><td>-1
(left-heavy)</td><td rowspan="2">Left-left imbalance
Single rotation:
right around N</td><td>Right-left imbalance
Double rotation:
1. right around C
2. left around N</td></tr>
<tr><td>0</td><td rowspan="2">Right-right imbalance
Single rotation:
left around N</td></tr>
<tr><td>1
(right-heavy)</td><td>Left-right imbalance
Double rotation:
1. left around C
2. right around N</td></tr>
</table>

Single Rotations

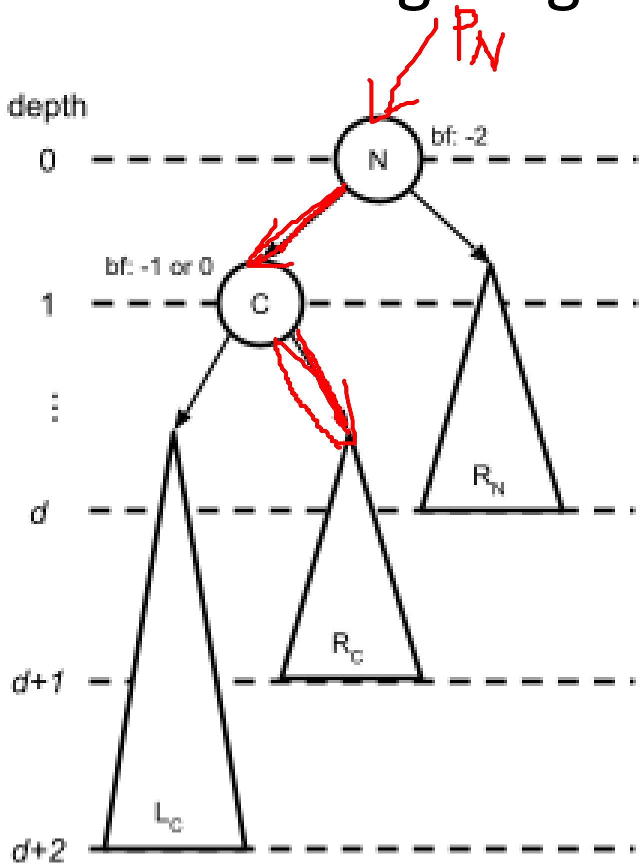
- Recall: a single rotation is needed if ***N* and *C* are heavy in the same direction.**

- Single rotation: always centered around the node N (where height balance is lost), and the rotation is in the opposite direction of the imbalance

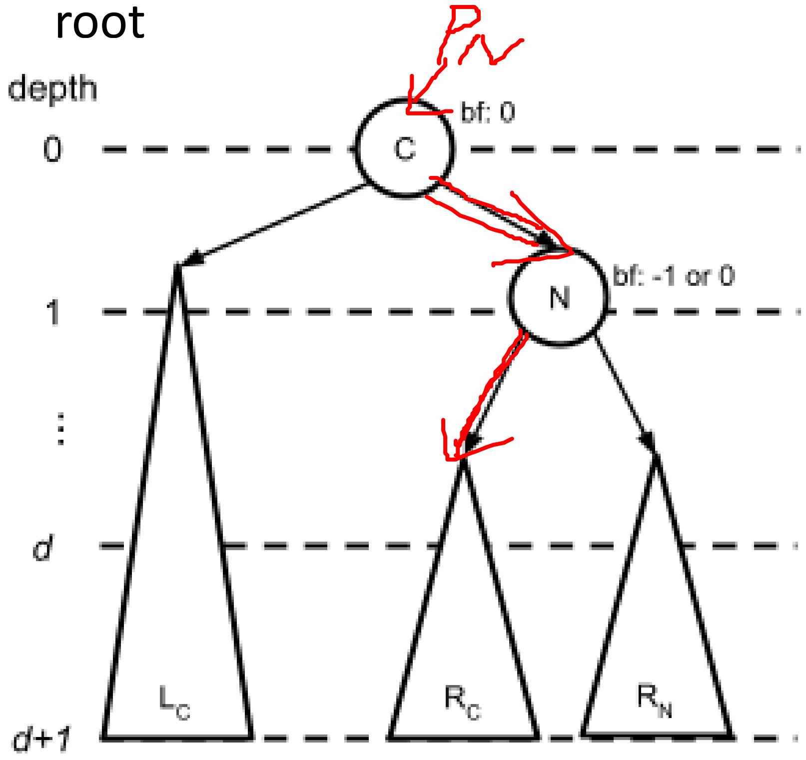
- Two situations:
  - **Left-left imbalance** – N is left-heavy and N's left child C is also left-heavy
    - Cause: insert an element into C's left subtree OR remove an element from N's right subtree
    - To fix: apply a single **right** rotation around N.
  - **Right-right imbalance** – N is right-heavy and N's right child C is also right-heavy
    - Cause: insert an element into C's right subtree OR remove an element from N's left subtree
    - To fix: apply a single **left** rotation around N

Single Rotations

- Visualize a single right rotation:



- In a right rotation around N:
  - N will become the right child of its current left child C.
  - C's current right child will become N's left child.
  - If N has a parent P_N, then C will replace N as P_N's child. Otherwise, if N was the root of the entire tree, C will replace N as the root



Double Rotations

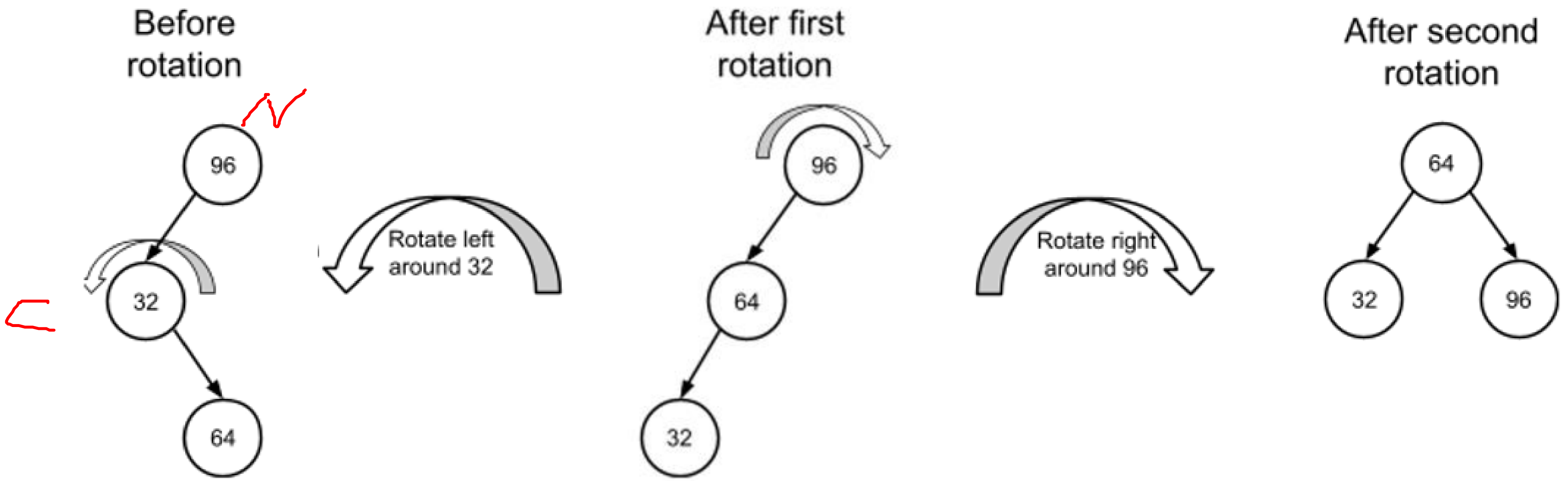
- Recall: a double rotation is needed if ***N* and *C* are heavy in the opposite direction.**

- A double rotation consists of two single rotations:
  - The first one is always centered around N's child C (align imbalances on the same side)
  - The second is always centered around N itself (where height balance is lost)

- Two situations:
  - **Left-right imbalance** – N is left-heavy and N's left child C is right-heavy
    - Cause: insert an element into C's right subtree OR remove an element from N's right subtree
    - To fix: apply a **left rotation around C** followed by a **right rotation around N**
  - **Right-left imbalance** – N is right-heavy and N's right child C is left-heavy
    - Cause: insert an element into C's left subtree OR remove an element from N's left subtree
    - To fix: apply a **right rotation around C** followed by a **left rotation around N**

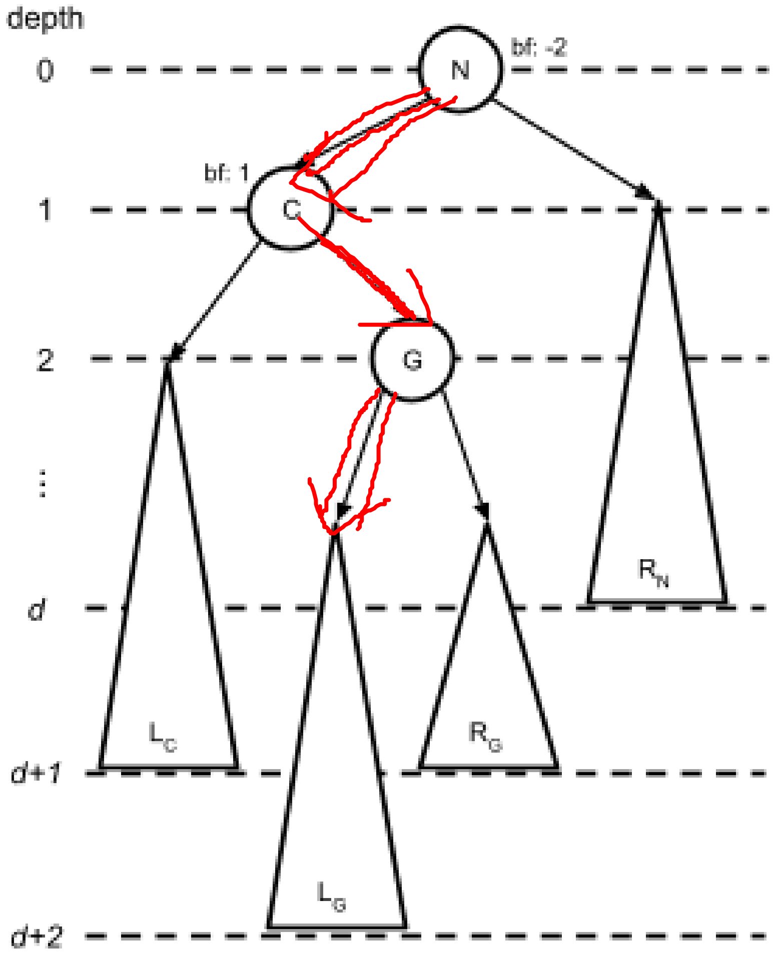
Double Rotations

- Example:

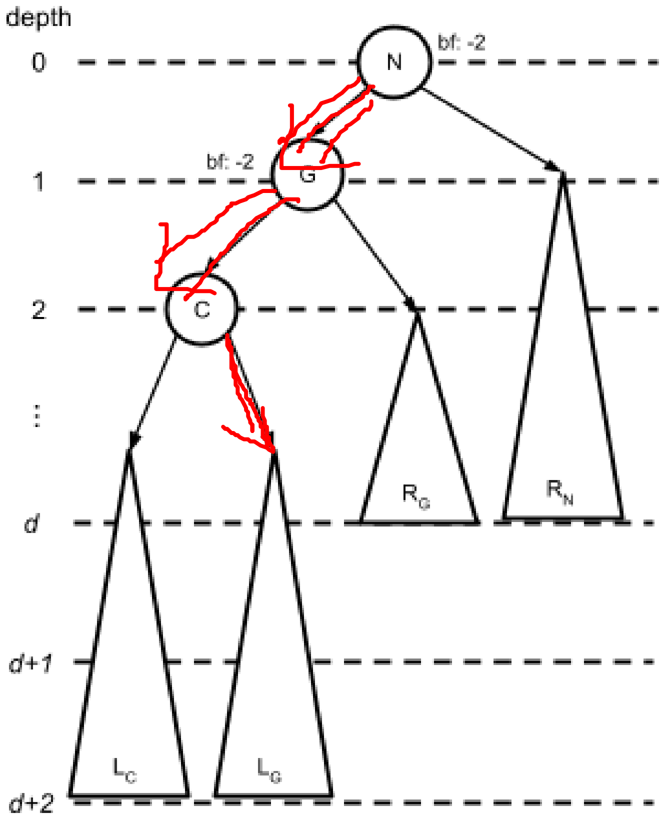


Double Rotations

- Visualize a left-right imbalance:

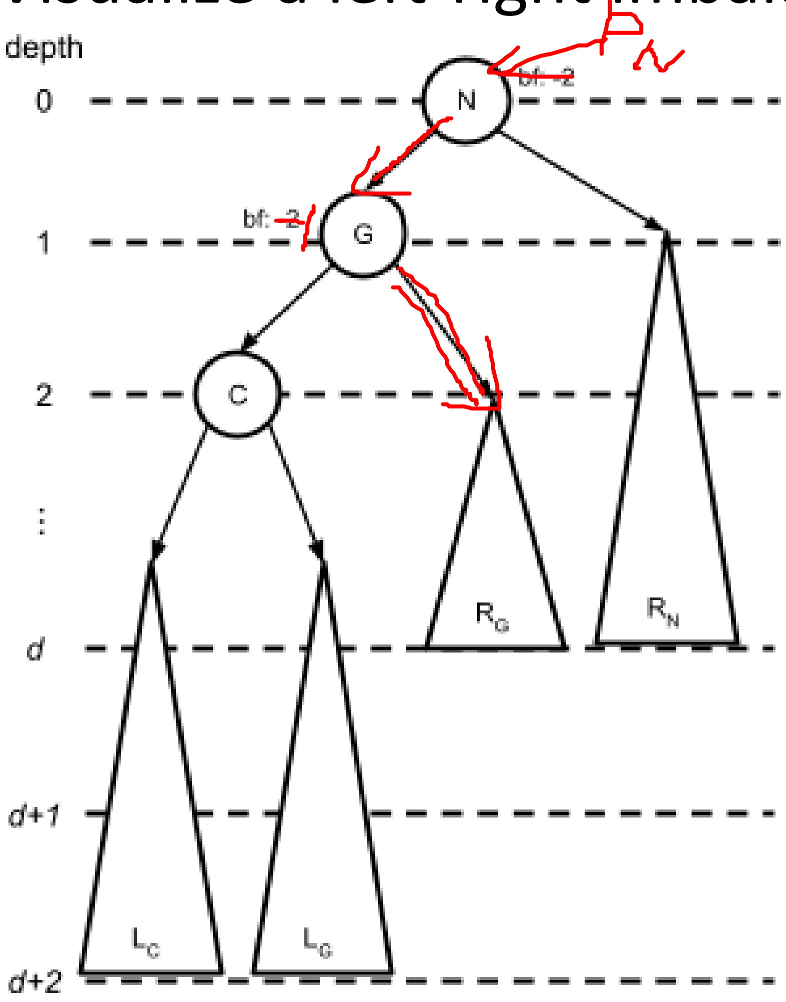


- First rotation: Center around C, opposite direction of C's imbalance, i.e., a left rotation around C:
  - G moves up in the tree to replace C as N's left child.
  - C moves down in the tree to become G's left child.
  - L_G becomes C's right child.

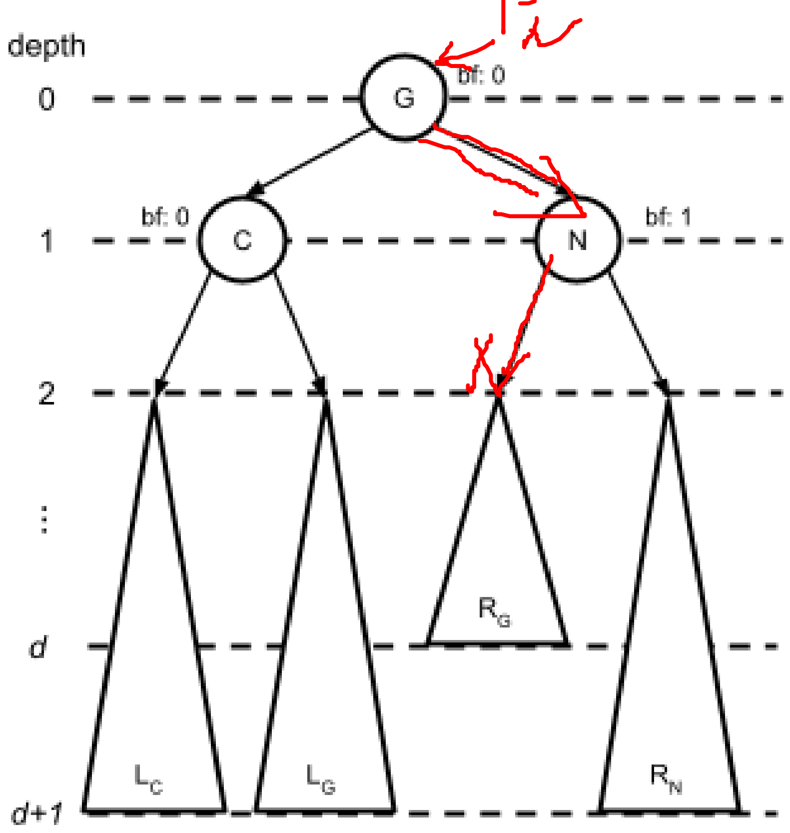


Double Rotations

- Visualize a left-right imbalance:

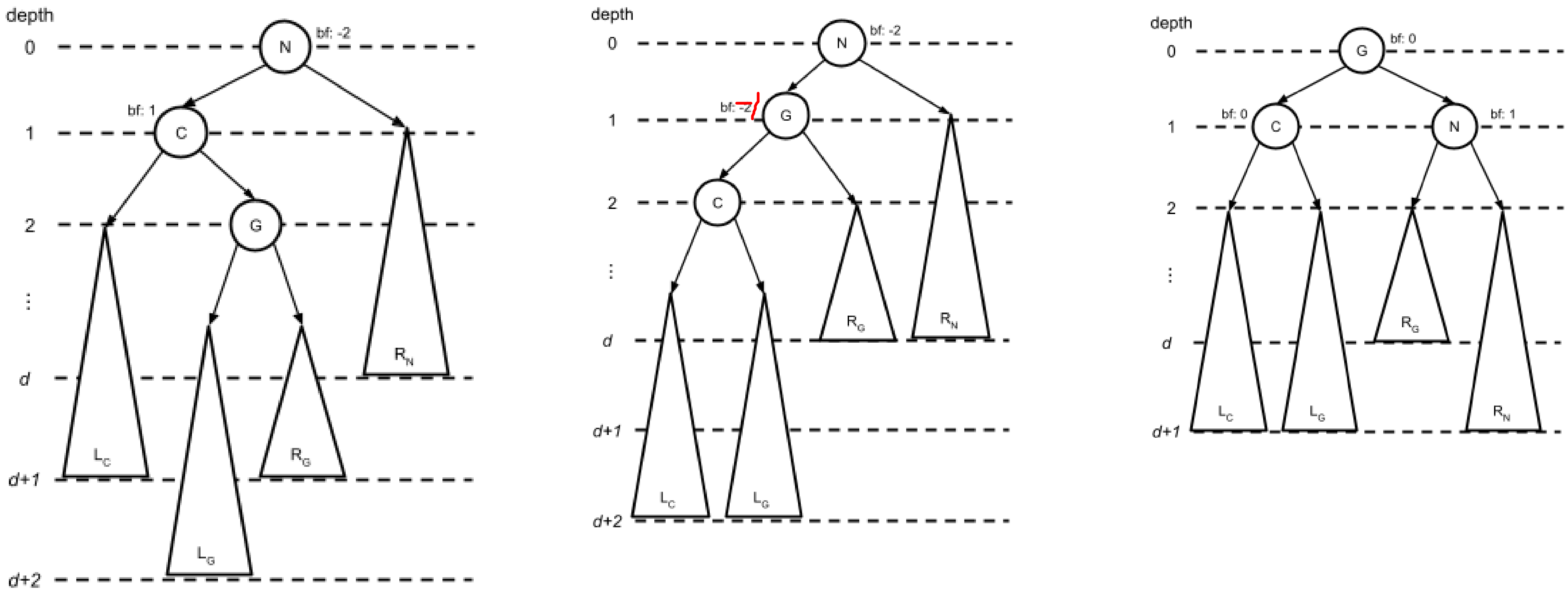


- Second rotation: Center around N, opposite direction of N's imbalance, i.e., a right rotation around N:
  - G moves up in the tree to become the new root of this subtree
  - N moves down in the tree to become G's right child.
  - If N had a parent, P_N, G would replace N as the child of P_N. If N was the root of the entire tree, G would become the new root



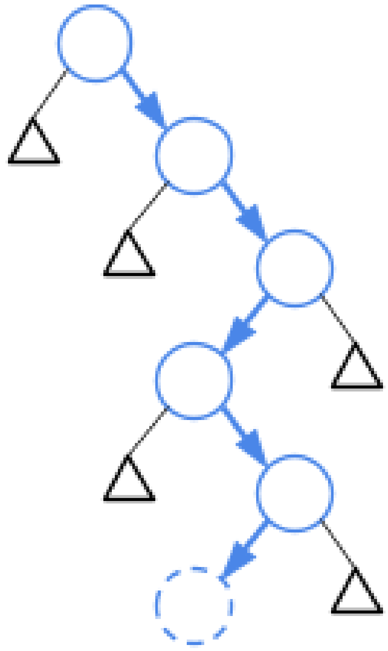
Double Rotations

- Visualize a left-right imbalance:

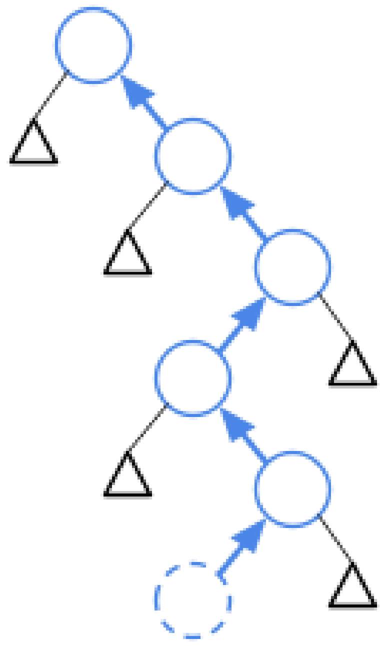


AVL Tree operations

- Note: an AVL tree will only ever need to be rebalanced in response to an operation that changes the structure of the tree
    - i.e. after inserting a new element or removing an element
- Rebalancing an AVL tree is a bottom-up operation
    - begins at the location in the tree where its structure was changed, and proceeds upwards from that location towards the root



Path taken downward to location of insertion/removal



Retraced path taken upward to rebalance tree

AVL Tree operations

- Need a mechanism to retrace a path *upwards* from a given node back to the root
- How: by adding a pointer to the AVL tree node structure that points to the node's parent
  - Then, retracing the path upwards from a node to the tree's root is as simple as following these parent pointers up the tree
- Add an additional field that allows us to track the height of the subtree rooted at each node.

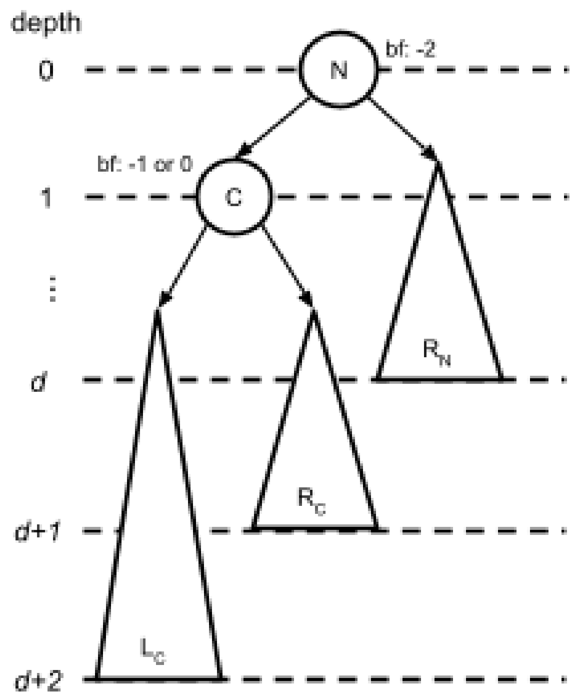
```
struct avl_node {
    int key;
    void* value;
    int height;
    struct avl_node* left;
    struct avl_node* right;
    struct avl_node* parent;
};
```

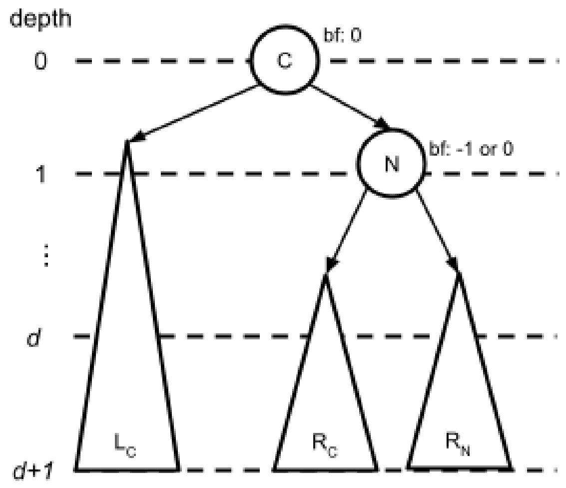
- When a node doesn't have a parent, parent = NULL.
- Specifically, the root node of the tree will always have a NULL parent pointer

AVL Tree operations

- Pseudocode for a right rotation:
- `rotateRight(N):`

```
rotateRight(N):
    C ← N.left
    N.left ← C.right
    if N.left is not NULL:
        N.left.parent ← N
    C.right ← N
  ✓ C.parent ← N.parent
  ✓ N.parent ← C
    updateHeight(N)
    updateHeight(C)
    return C
```





AVL Tree operations

- Pseudocode for a left rotation:
- **rotateLeft(N):**

```
rotateLeft(N):
    C ← N.right
    N.right ← C.left
    if N.right is not NULL:
        N.right.parent ← N
    C.left ← N
    C.parent ← N.parent
    N.parent ← C
    updateHeight(N)
    updateHeight(C)
    return C
```

```
updateHeight(N):
    N.height ← MAX(height(N.left), height(N.right)) + 1
```

AVL Tree operations

- How these pieces work:
  - Rotating left or right around a given node: simply involves trading a few pointers.
  - After every rotation, re-compute the subtree heights for both the node that moved downwards during the rotation (i.e. N) and the node that moved upwards during the rotation (i.e. C).

AVL Tree operations

- pseudocode for the insert operation:

```
avlInsert(tree, key, value):
      insert key, value into tree like normal BST insertion
      N ← newly inserted node
      P ← N.parent
      while P is not NULL:
            rebalance(P)
            P ← P.parent
```

- pseudocode for the remove operation:

```
avlRemove(tree, key):
      remove key from tree like normal BST removal
      P ← lowest modified node (e.g. parent of removed node)
      while P is not NULL:
            rebalance(P)
            P ← P.parent
```

The key piece: rebalance() function, which performs rebalancing at each node:

AVL Tree operations

- Pseudocode for rebalance():

- ```
rebalance(N):
      if balanceFactor(N) < -1:
            if balanceFactor(N.left) > 0:
                  N.left ← rotateLeft(N.left)
            newSubtreeRoot ← rotateRight(N)
            if newSubtreeRoot.parent is not NULL
                  if newSubtreeRoot.parent.left is N:
                        newSubtreeRoot.parent.left ← newSubtreeRoot
                  else:
                        newSubtreeRoot.parent.right ← newSubtreeRoot
      else if balanceFactor(N) > 1:
            if balanceFactor(N.right) < 0:
                  N.right ← rotateRight(N.right)
            newSubtreeRoot ← rotateLeft(N)
            if newSubtreeRoot.parent is not NULL
                  if newSubtreeRoot.parent.left is N:
                        newSubtreeRoot.parent.left ← newSubtreeRoot
                  else:
                        newSubtreeRoot.parent.right ← newSubtreeRoot
else:
      updateHeight(N)
```
```

Runtime Complexity of AVL Tree operations

- Single rotation (rotateLeft() and rotateRight()):
  - A limited number of pointers is updated
  - The height of two nodes is updated
  - Thus, O(1)
- rebalance() :
  - For each call, at most two rotations.
  - Thus, O(1)

- How many times will rebalance() be called?
  - once per node on a traversal upwards to the root of the tree
  - Thus, the maximum number of times rebalance() can be called is h (height of the tree)

- If a tree is height balanced, then h = log(n). Thus, the AVL tree's insert and remove operations each have overall complexity of O(h) = O(log n)