Data Structures

AVL Deletion

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

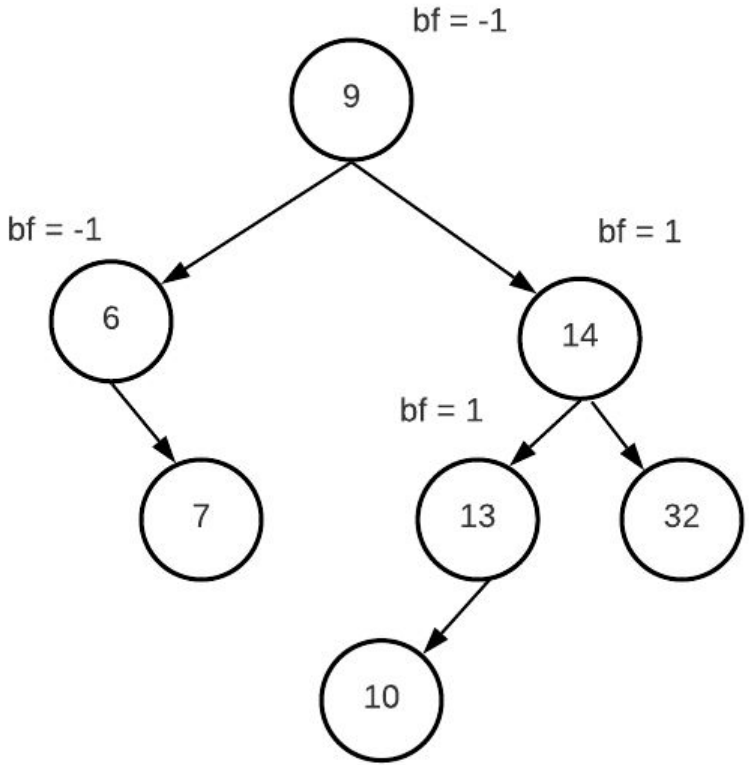
Node deletion

- Recall how deletion has several cases (successor replacement in the worst case)
- In fact, we follow in the logic as insertion. If a tree node has changes
  - Update height. Do rebalance. That is it ⇒ Simple code changes

```
def delete(self, val):
    def process(current, val):
        if not current:
            return
        if val < current.val:...
        if val > current.val:...
        if current.is_leaf():  # case 1: leaf
            return None  # Just remove

        if not current.right:  # case 2: has left only
            current = current.left
        elif not current.left:  # case 2: has right only
            current = current.right
        else:
            # 2 children: Use successor
            mn = self.min_node(current.right)
            current.val = mn.val  # copy data
            current.right = process(current.right, mn.val)

        current.update_height()
        return self.balance(current)

    self.root = process(self.root, val) # ** update root
```

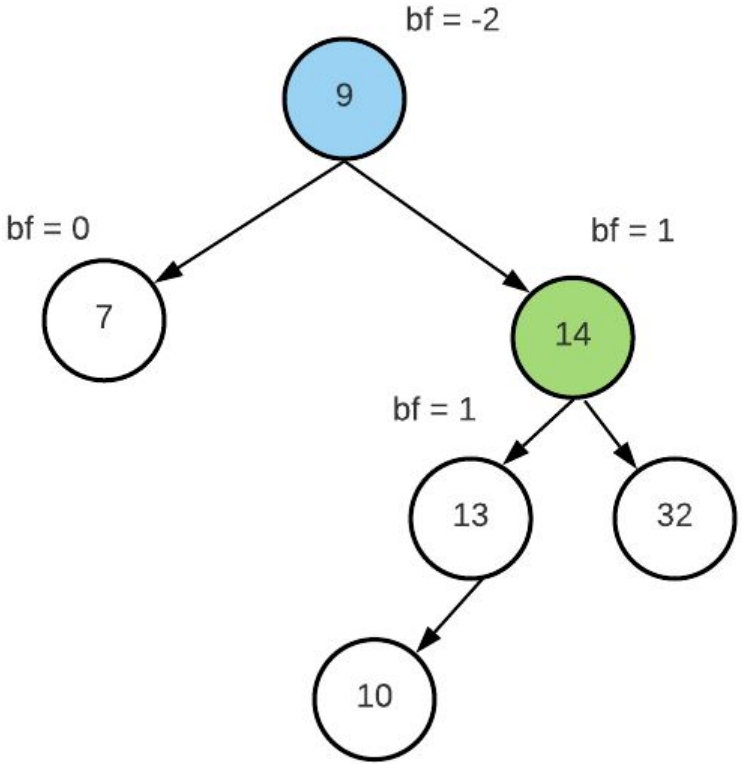
Let’s simulate

- Let’s create this tree first
  - Insert: 9 6 14 7 13 32 10
    - Tip: level-order traversal
- Next, delete 6
  - 6 only has one child node
  - Connect parent (9) to child (7)
  - Compute the BF for 9



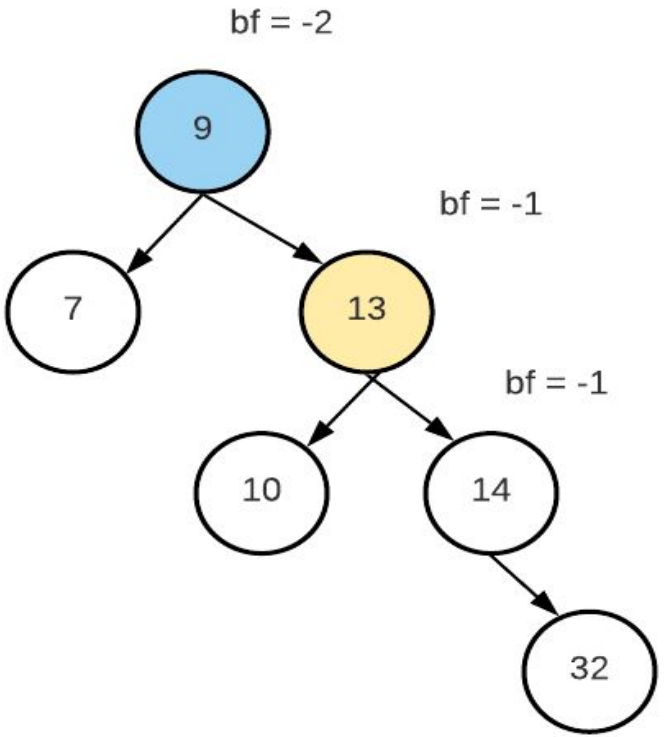
Delete 6

- BF [-2, 1] ⇒ Right-Left unbalanced tree
  - Right-Rotation(14)
    - Pushes 14 down and 13 up
  - Then Left-Rotation(9)



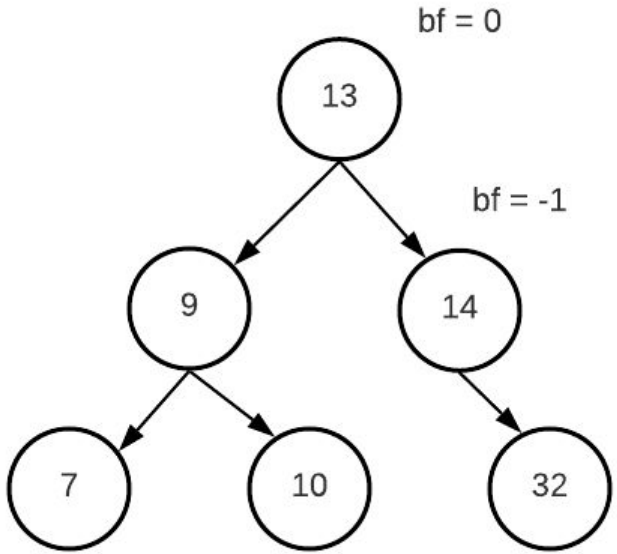
Delete 6

- Now we have BF [-2, -1]
  - Right-right unbalanced type
- Left-Rotation(9)
  - Pushes 9 down and 13 up
  - B-subtree(10)
    - Moves from left of 13 to right of 9



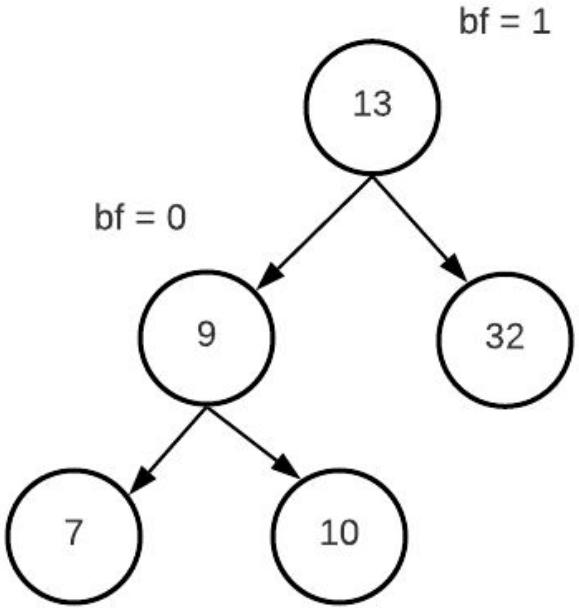
Delete 6

- It's fixed now!
- Next delete 14
  - Again, link 13 to 32
  - Update the BF, then check the balance



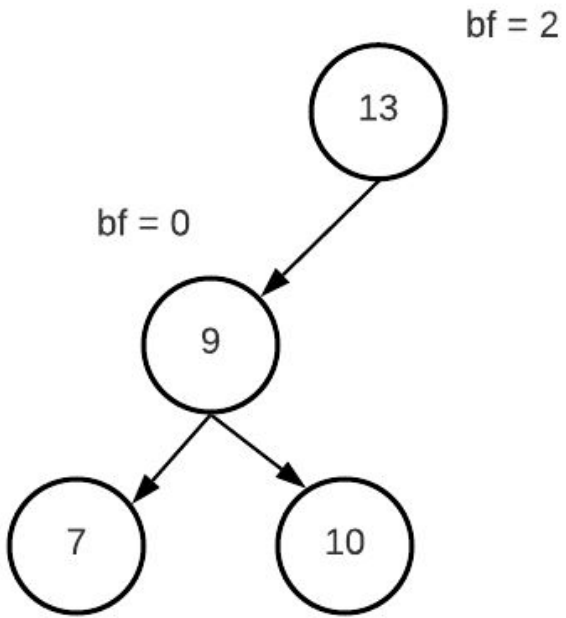
Delete 14

- No problems after deletion
- Next, delete 32
  - It's a leaf node, simply delete it as normal
  - Update the height and check the balance



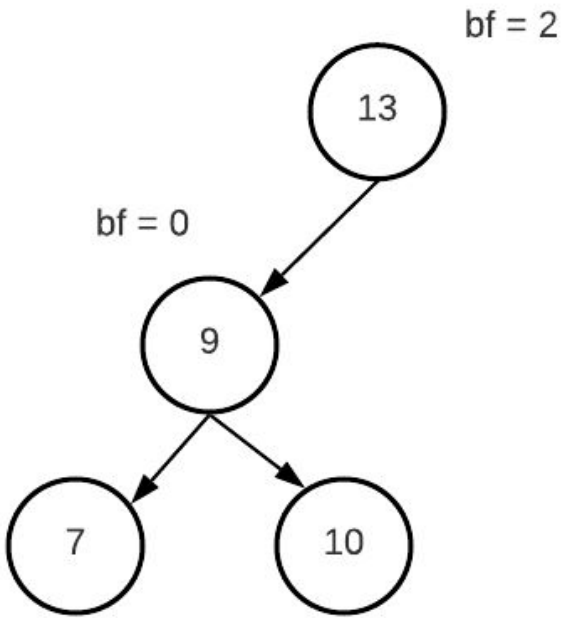
Delete 32

- Now we have an unbalanced tree/subtree
- The BF is 2, so we check the left subtree
- In insertion, left will be either 1 or -1
  - 1 ⇒ Left-Left rotation
  - -1 ⇒ Left-right rotation (requires two rotations to fix)
- Deletion here creates a NEW scenario!
  - BF = 0
  - Now, our possible values are: {-1, 0, 1}
  - It simply means **both** left-left or left-right are okay
  - Going left-left is more efficient
  - This means our balance code **doesn't need** changes
    - It only checks for left-right and right-left



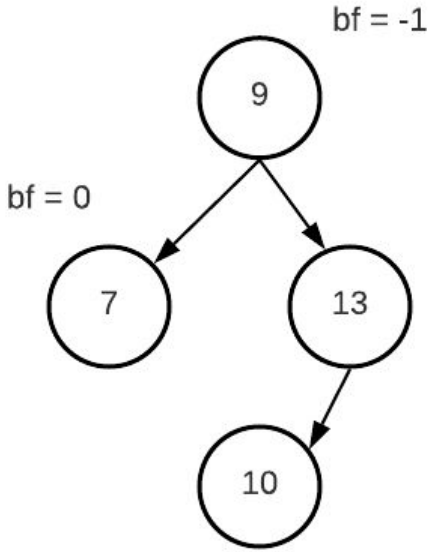
Delete 32

- BF [2, 0] ⇒ Left-left rotation
- Perform Right-rotation(13)
  - Pushes 13 down and 9 up
  - Subtree B(10) changes its parent
-



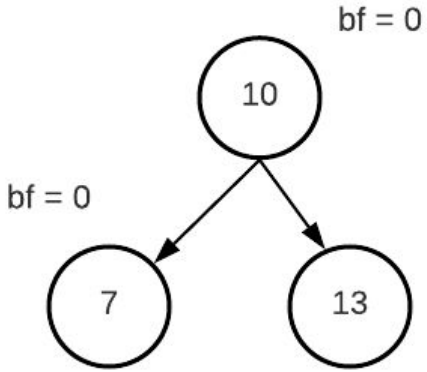
Delete 32

- Now it's fixed
- Next, delete 9
  - 9 has 2 children
  - Find successor(9) = 10
  - Copy the successor value
  - Remove node (10), which has no child
- So, deletion in the successor case ends up being jus
  with 0-1 child nodes - as we already learned



Done

- As you can see, the effect of deletion is direct
- Only it created the BF==0 case
  - BF {2, 0}
  - BF {-2, 0}
- However, this case can be handled easily, just like:
  - BF {2, 1}
  - BF {-2, -1}
  - So, the code for balance() didn't change
    - *Review code and verify*



"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."