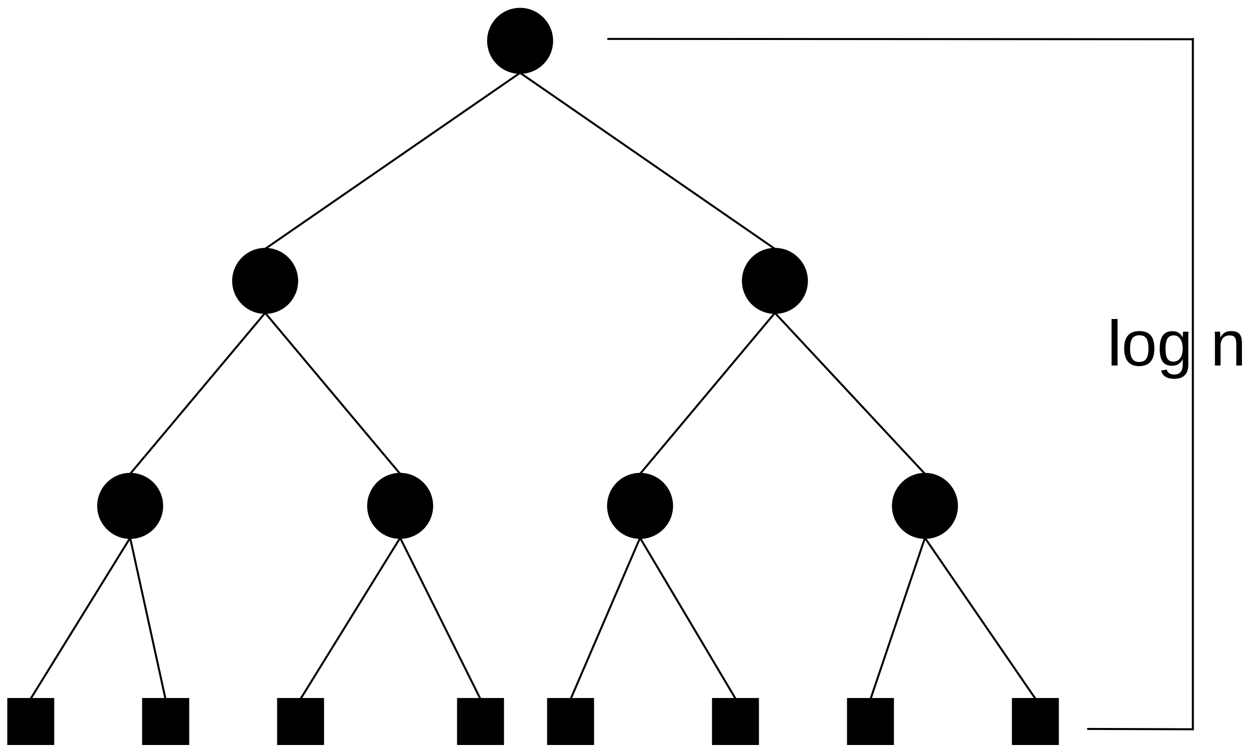
Red Black Tree

A balanced binary search tree

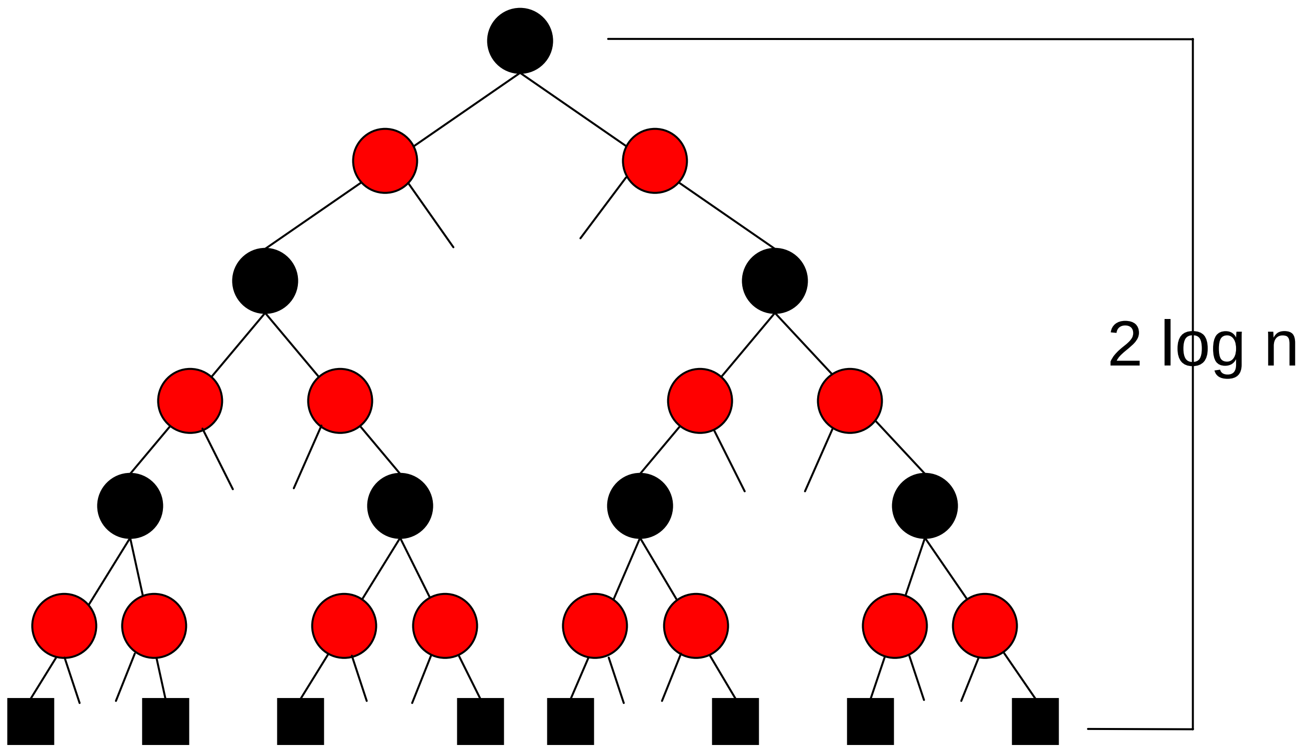
Red Black Tree

1. Every node is either red or black
2. For each node, all paths from the node to descendant leaves contain the same number of black nodes
3. If a node is red, then both its children are black
4. The root is black
5. Every dummy leaf is black

Balance

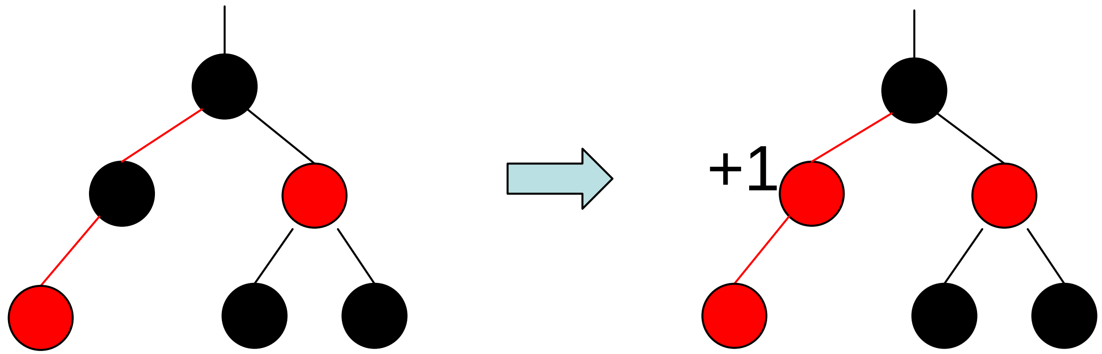


Balance



Notation

- +1 means need one more black for the node



Insertion

- Set inserted node to be red
- Fix the violation of Properties 3 and 4

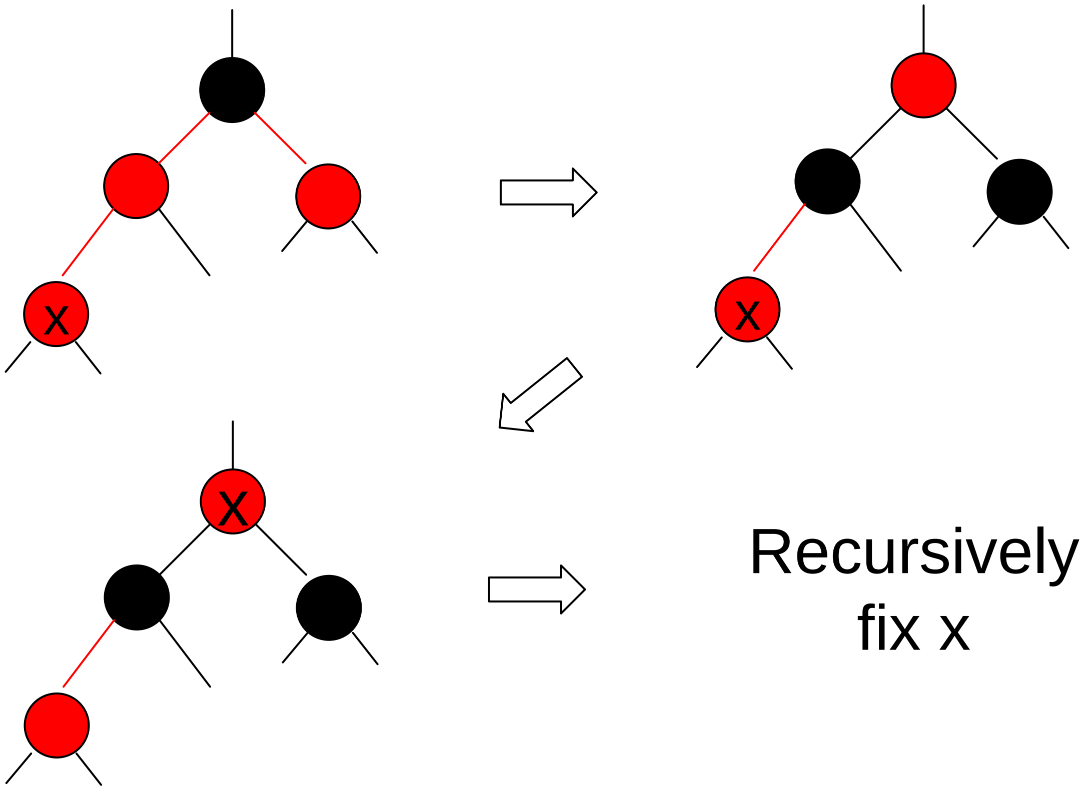
Insertion

To insert x:

- Uncle is red (Case 1)
- Uncle is black
  - Both x and parent are left (or right) child (Case 2)
  - Others (Case 3)

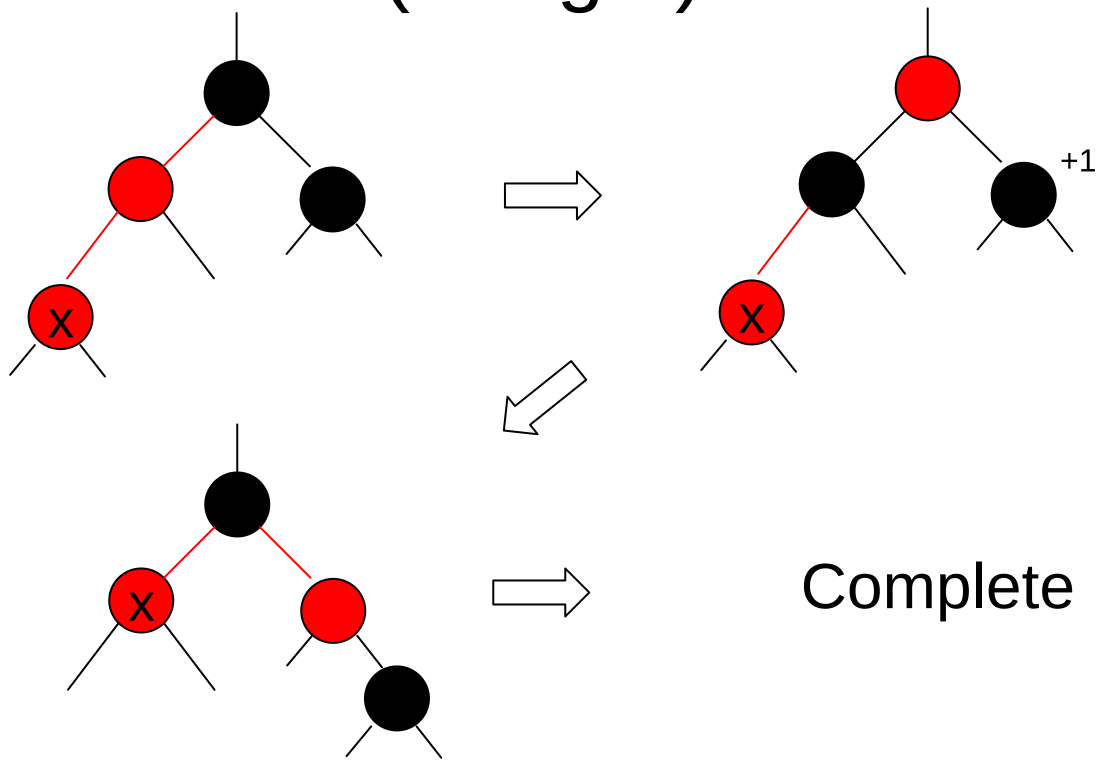
Insertion

- Case 1: Uncle is red



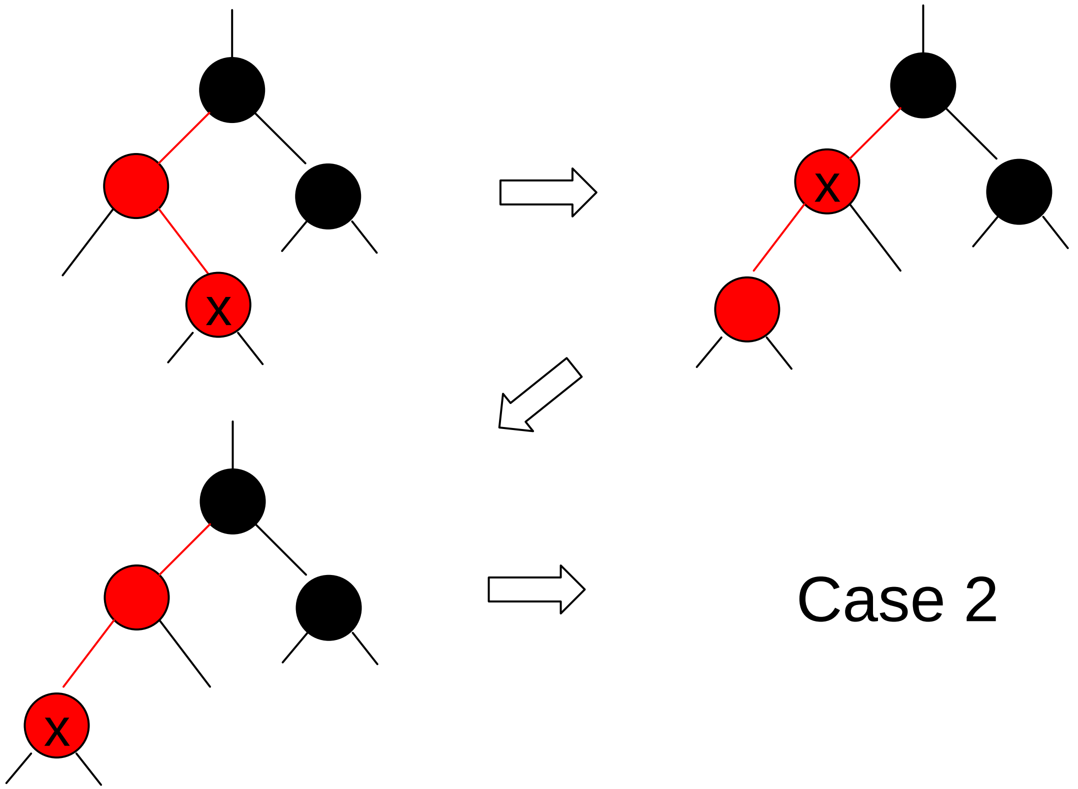
Insertion

- Case 2: Uncle is black. Both x and parent are left (or right) child



Insertion

- Case 3: Others



Delete vs Remove

- To delete z, node z may not be removed in the tree

- Denote y as the removed node

- Let x be the child of y

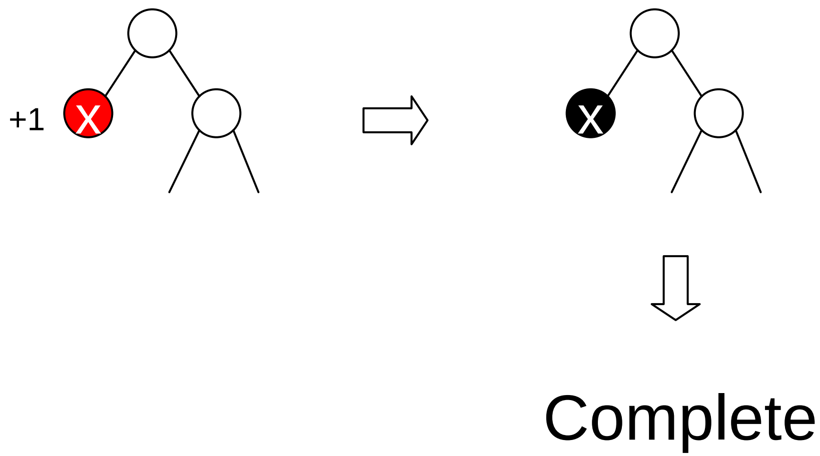
Deletion

- No violation when the removed node y is red
- Otherwise, fix the violation of property 2 and 4

- x is red (Case 1)
- x is black
  - Sibling is red (Case 2)
  - Sibling, denoted as s, is black
    - Both s’s children are black (Case 3)
    - The children of s, left are black, right are red (Case 4)
    - Right children of s are red
      - Parent are black (Case 5)
      - Parent are red (Case 6)

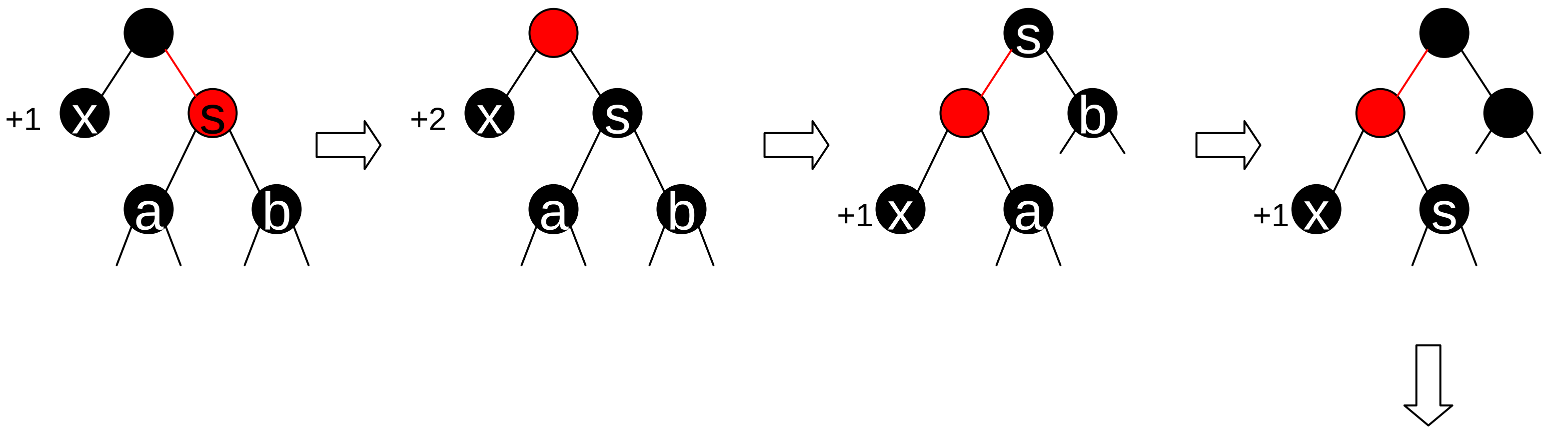
Deletion

- Case 1: x is red



Deletion

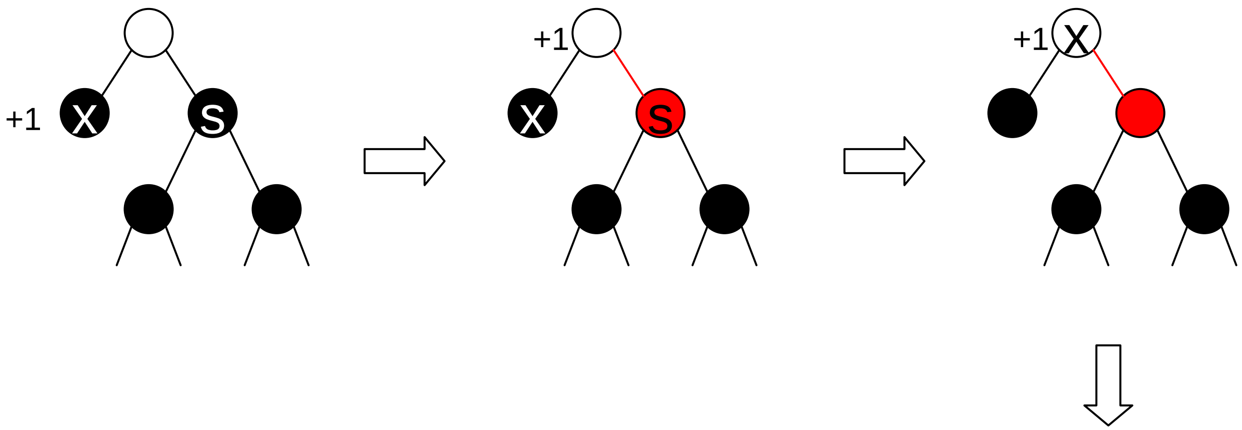
- Case 2: Sibling is red



Case 3~6

Deletion

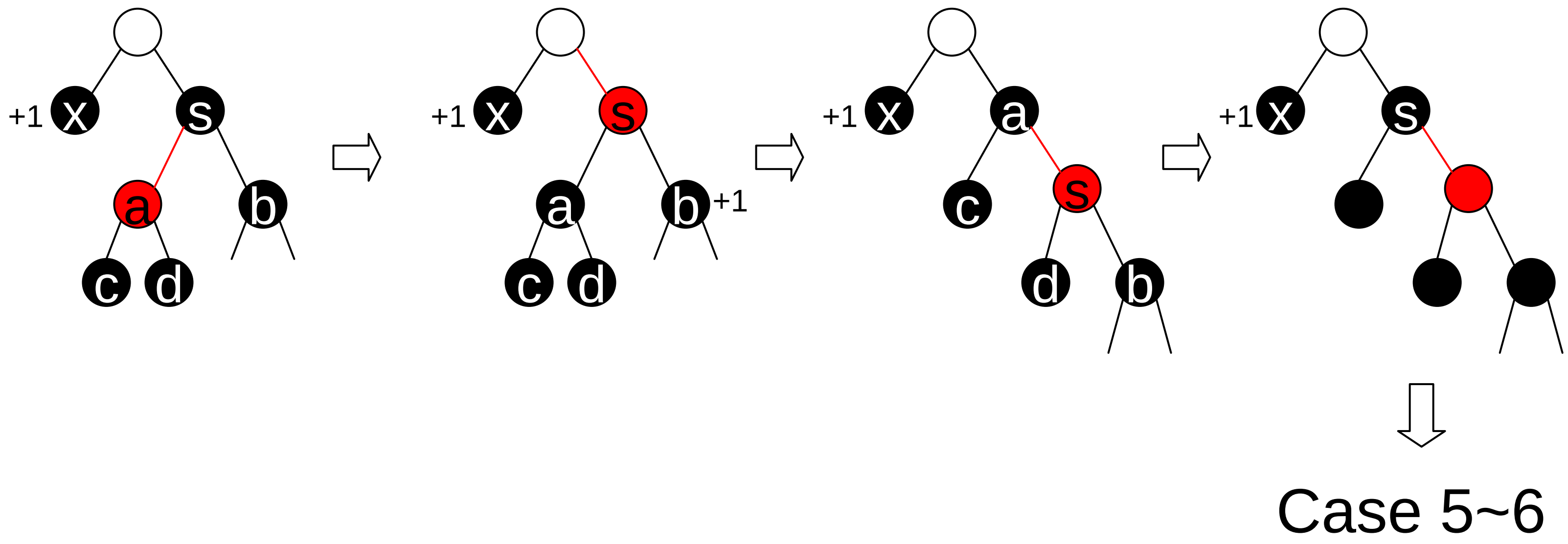
- Case 3: Both sibling's children are black



Recursively fix x

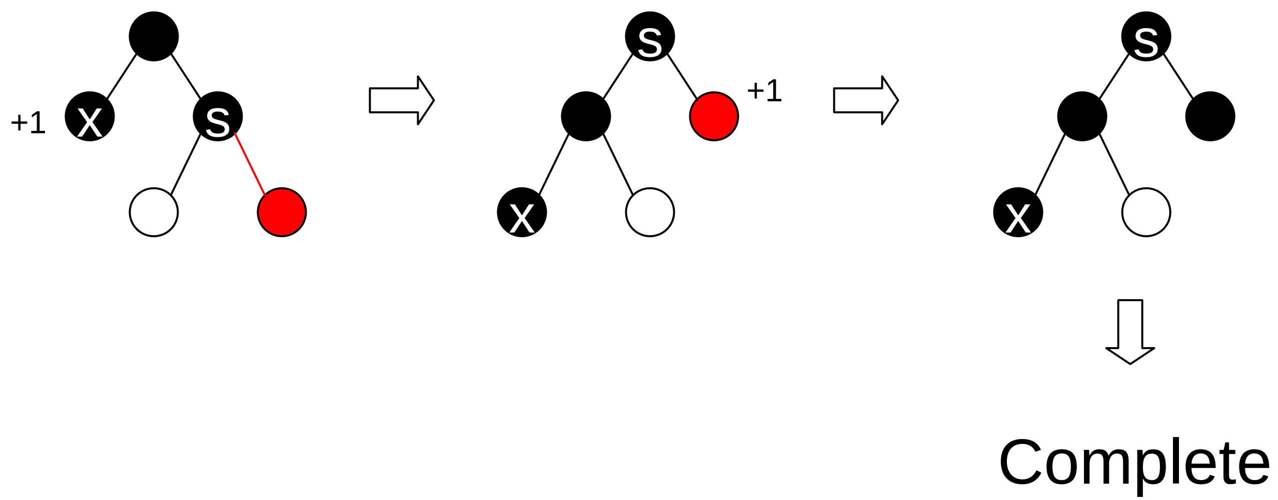
Deletion

- Case 4: Sibling's left child is red, right child is black



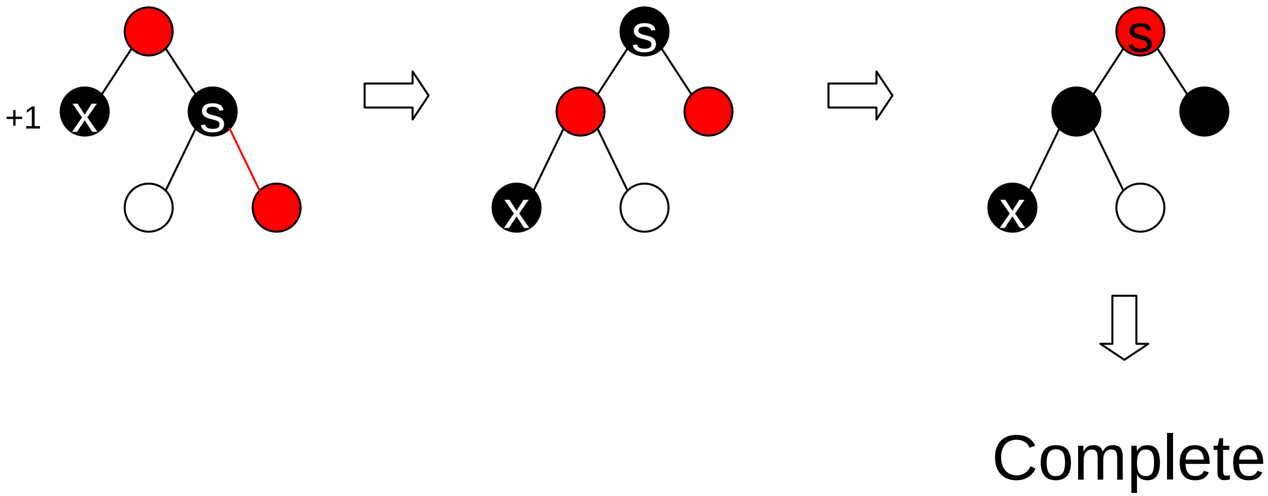
Deletion

- Case 5: Sibling's right child is red



Deletion

- Case 6: Sibling's right child is red



2 colors, why?

- Tree height is not that tight

- Reduce the cost to balance tree