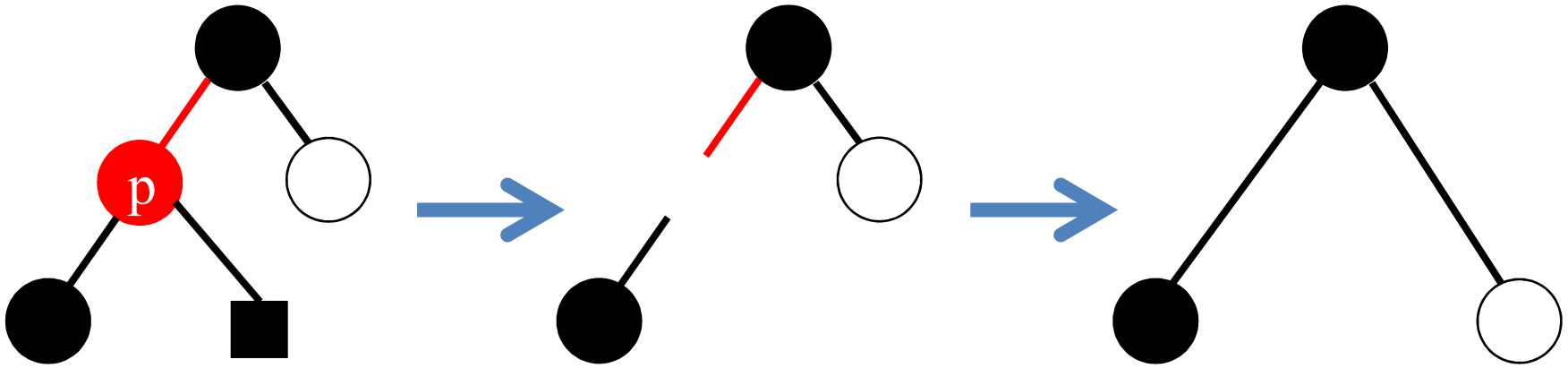
Delete

- Perform standard binary search tree delete
  - Node to be deleted is leaf
  - Node to be deleted has only one child
  - Node to be deleted has two children
    - “Copy” a successor (or a predecessor) appeared in inorder traversal and then recursively call delete
- We always end up deleting a node which is either leaf or has only one child

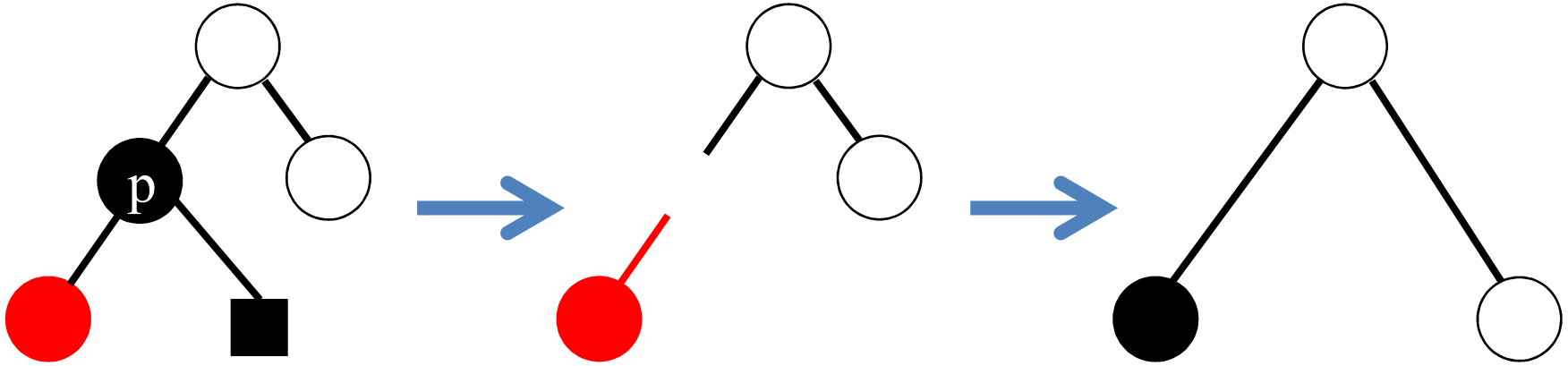
Delete

white circle : any color can be placed



Delete

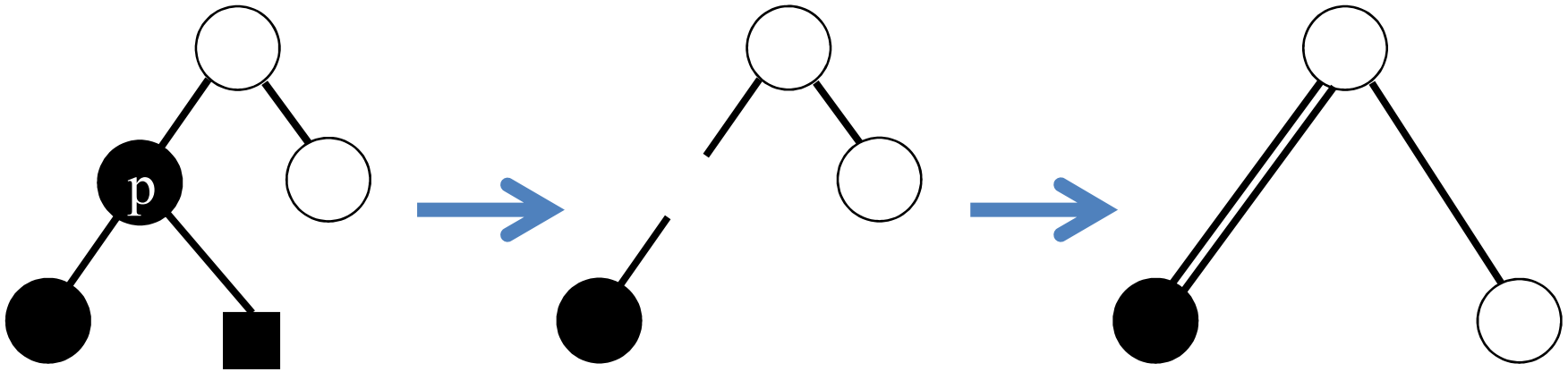
white circle : any color can be placed



Delete

- Delete black will violate red-black property
  - Path passing through deleted node will have fewer number of black nodes
  - Delete 1 child node or leaf → double black pointer

white circle : any color can be placed



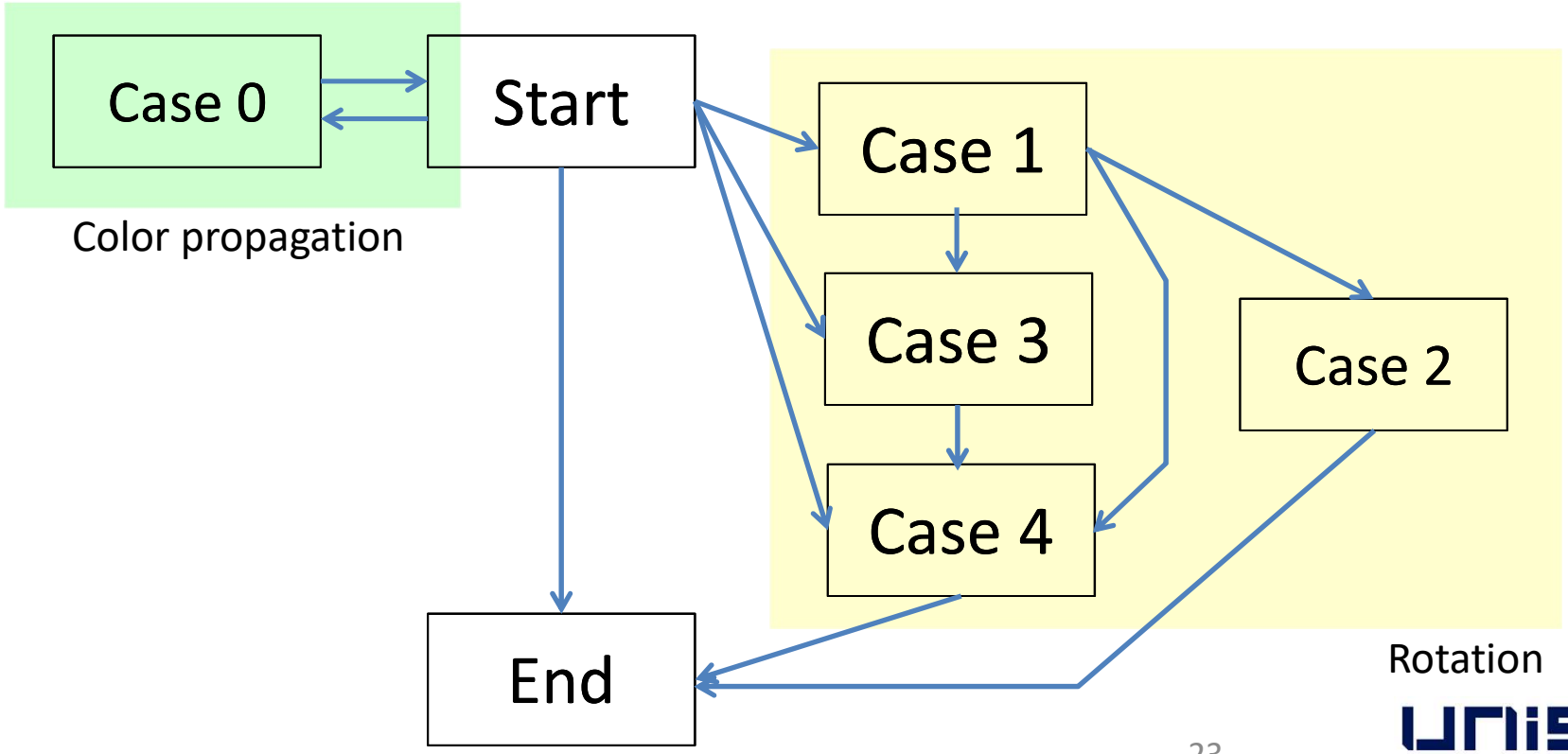
Deleting right child is symmetric

Dealing with Fewer # of Black Nodes

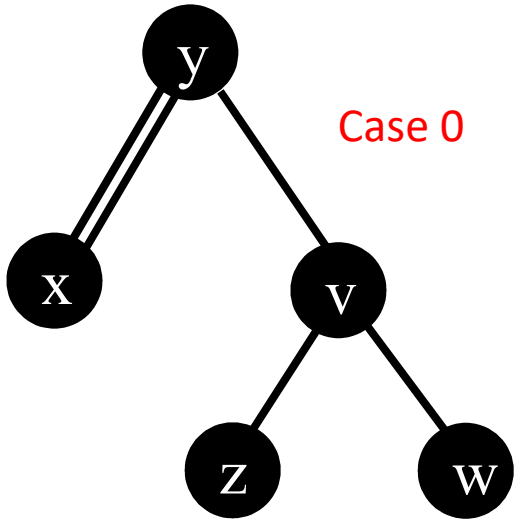
1) Reduce # of black nodes at all other paths
2) Increase # of black nodes for the path containing double black pointer

Preview: Delete Workflow

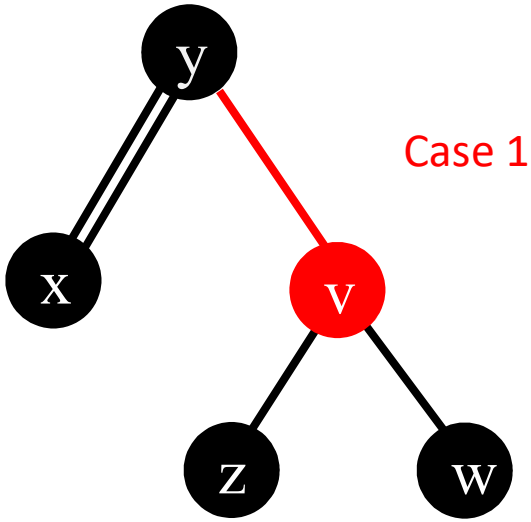
- At most 3 rotations are needed



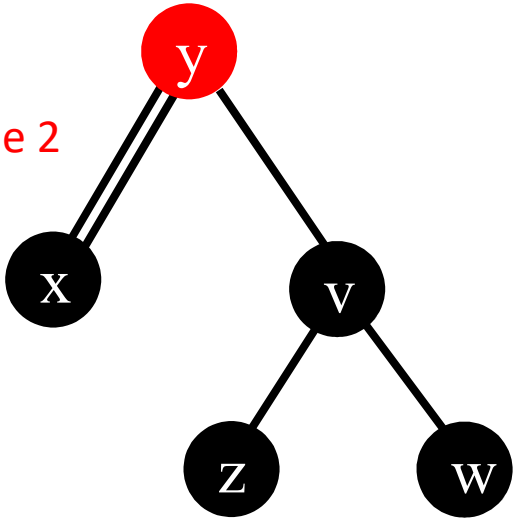
Five Cases



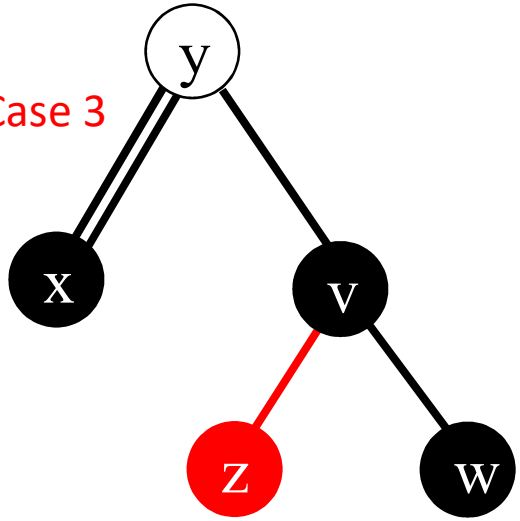
Case 0



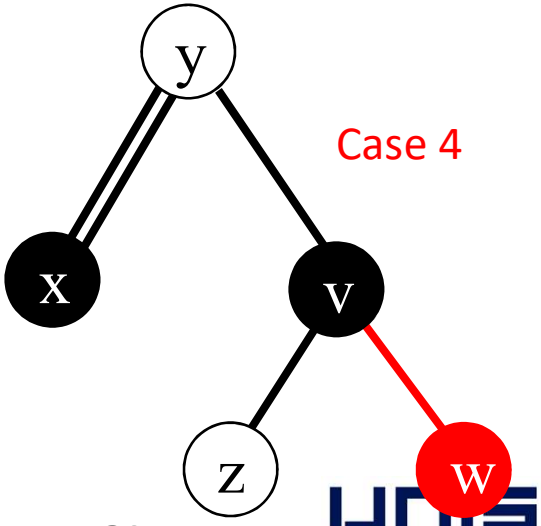
Case 1



Case 2



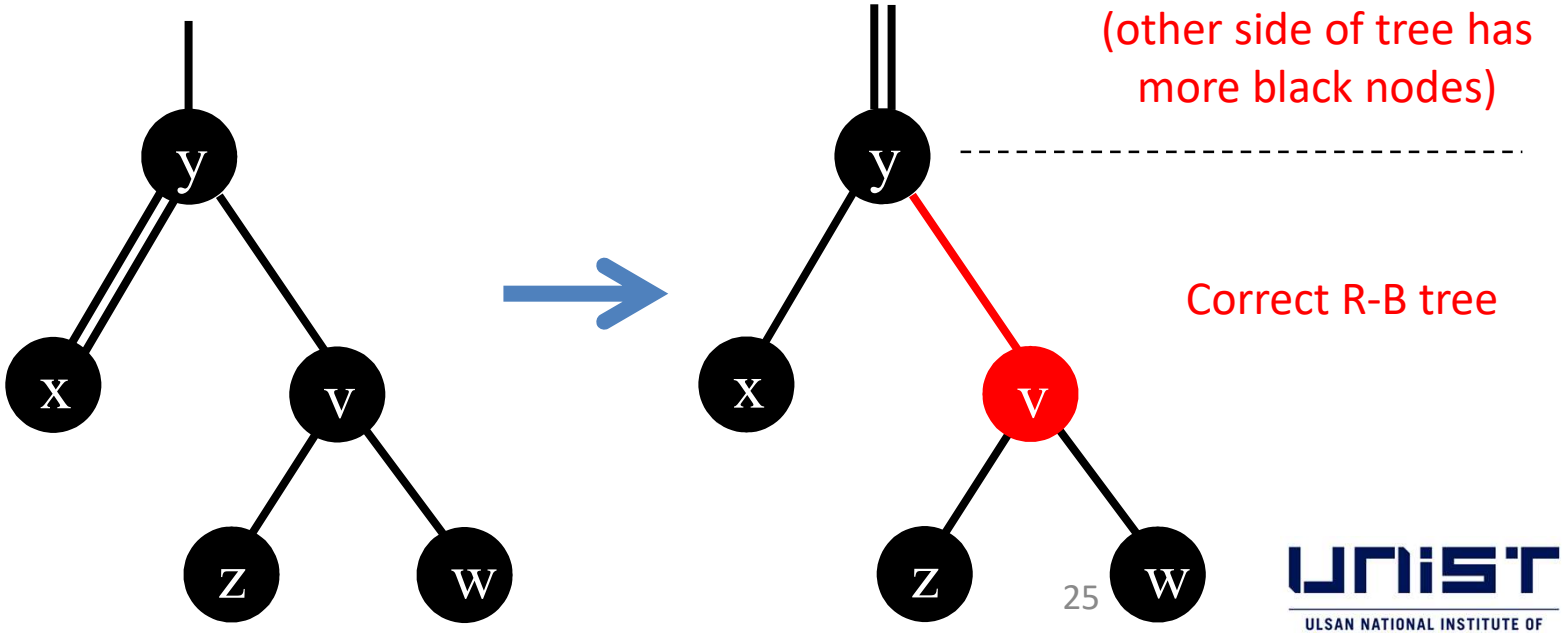
Case 3



Case 4

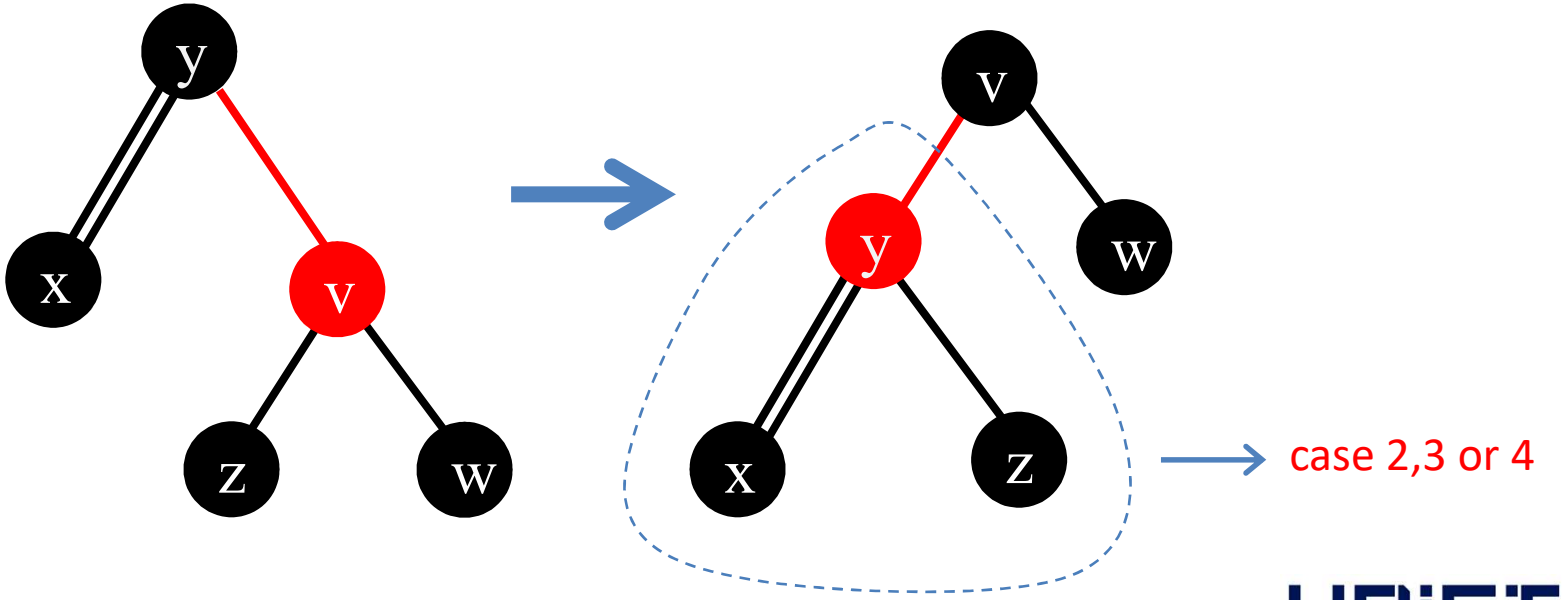
Delete

- Case 0: v and its two children are black
  - Make v red
  - If y is black, then move up double black pointer
    - x = y, y = y's parent, and restart



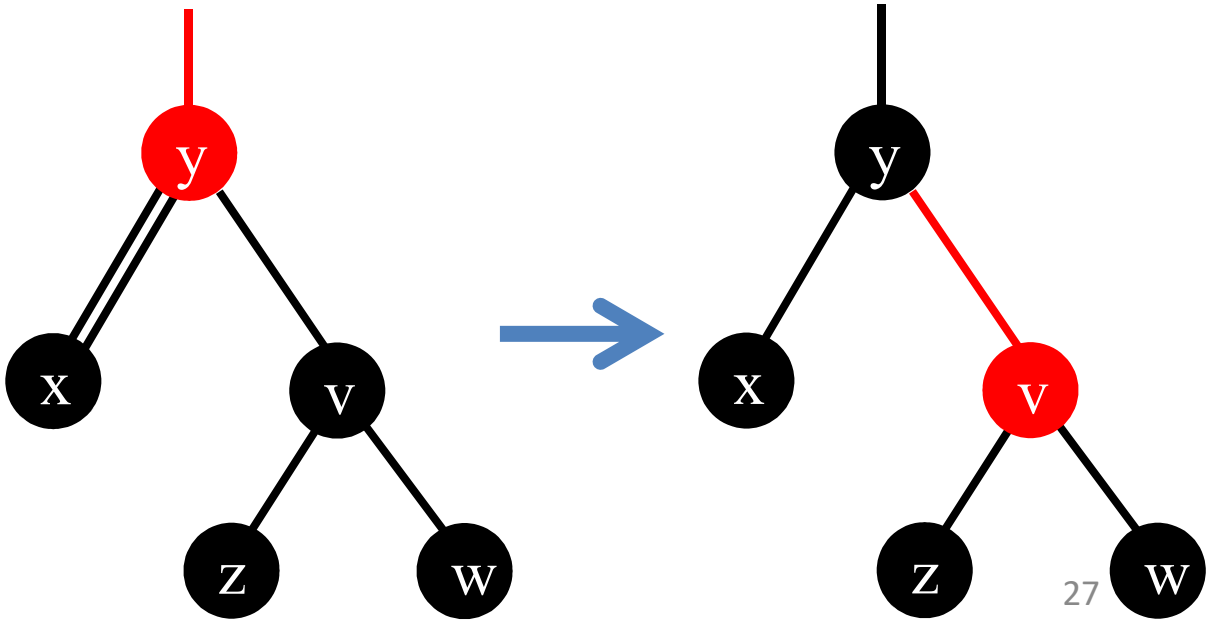
Delete

- Case 1: y is black and v is red
  - –Left-rotate at y and exchange colors of y & v
  - –Go to case 2, 3, or 4 for subtree of y



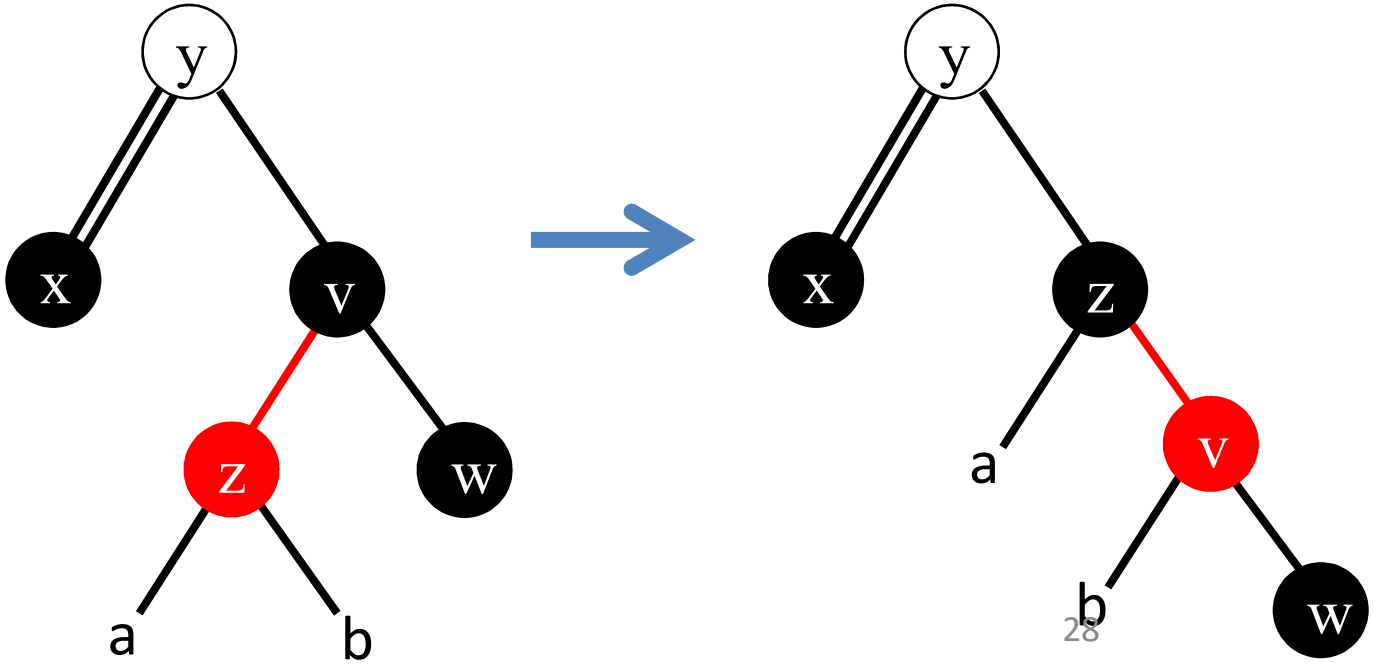
Delete

- Case 2: v and its two children are black
  - Make v red
  - If y is red, then make y black and remove double pointer. Done.



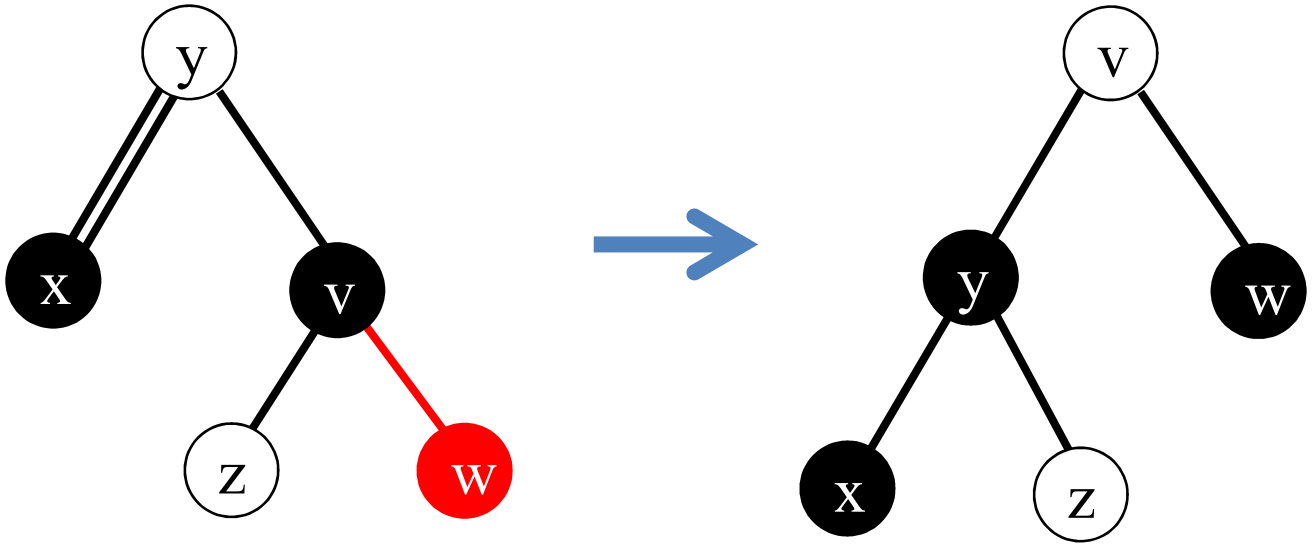
Delete

- Case 3: v and its right child are black while its left child is red
  - Right-rotate at v and exchange colors of v and its left child. Go to case 4.



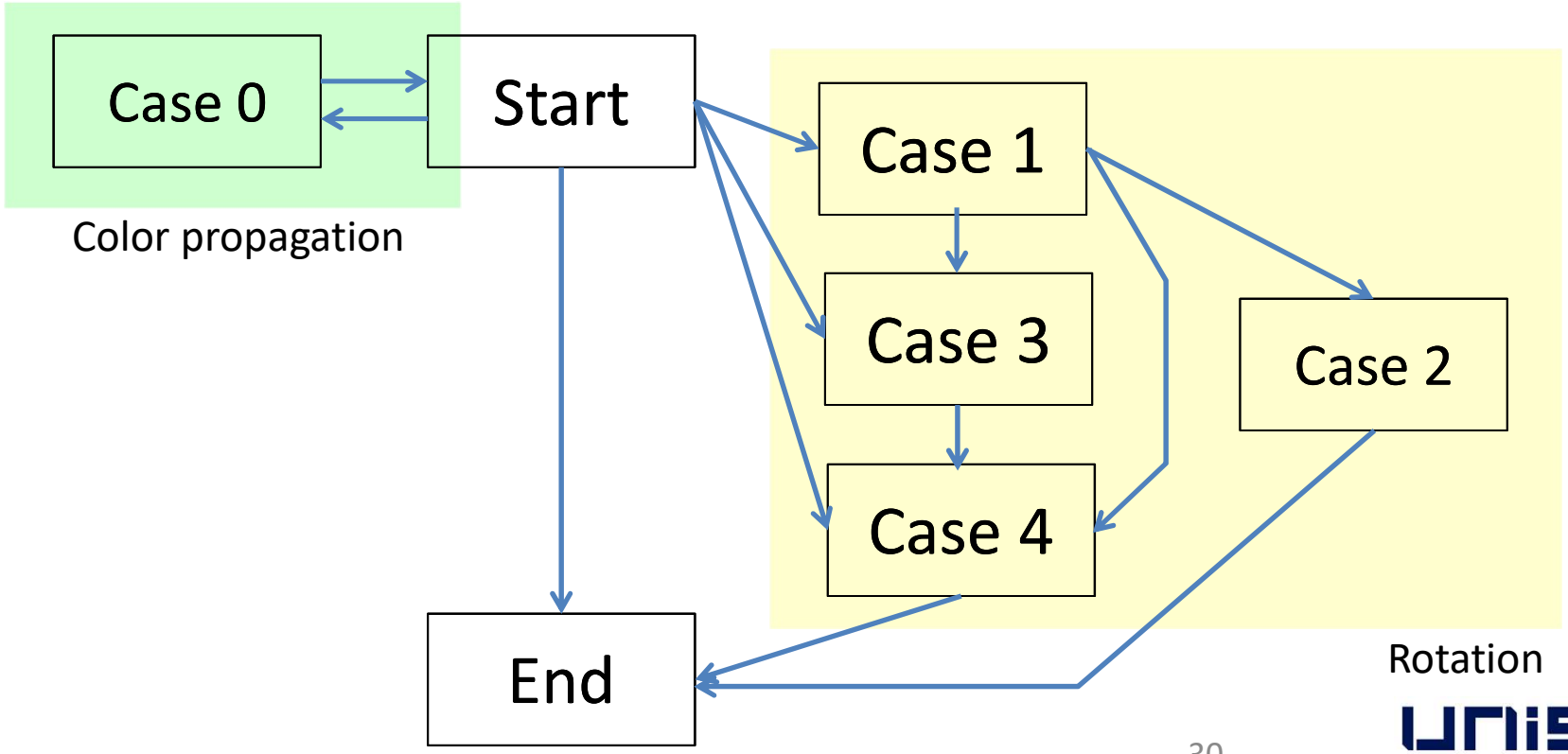
Delete

- Case 4: v is black and its right child is red
  - Left rotate at y, exchange colors of y & v
  - Remove double black pointer, change w to black. Done.



Delete Workflow

- At most 3 rotations are needed
- Color exchange can propagate log n times



Discussion

- Red-Black trees use color as balancing information instead of height as in AVL trees
- Insert/delete may cause a perturbation (if two consecutive red nodes exist)
- Perturbation is either
  - propagated to a higher level in the tree by color flip, or
  - resolved locally by rotations
- O(1) for a rotation or O(log n) color flips
- Total time: O(log n)

AVL Trees v.s. Red-Black Trees

- AVL trees provide faster search: more balanced
- Red-Black trees provide faster insert/delete
  - Fewer rotations due to relaxed balancing
- AVL trees store **balance factors or heights** for each node
- Red-Black trees require only 1 bit of information per node

Questions?