Balanced
Binary Search Trees

Red Black and 2-4

Binary Search Tree (BST)

The binary search tree was used to organized comparable data

Binary Search Tree (BST)

The binary search tree was used to organized comparable data

Operations allowed included,

Binary Search Tree (BST)

The binary search tree was used to organized comparable data

Operations allowed included,

- Insert
- Remove
- Contains
- Traverse

Binary Search Tree (BST)

The binary search tree was used to organized comparable data

Operations allowed included,

- Insert
- Remove
- Contains
- Traverse

What was the average runtime for insert remove and contains?

Binary Search Tree (BST)

The binary search tree was used to organized comparable data

Operations allowed included,

- Insert
- Remove
- Contains
- Traverse

What was the average runtime for insert remove and contains?

What was the worst case runtime for insert remove and contains?

Balanced BSTs (BBSTs)

In CS I at least one BBST was covered

Balanced BSTs (BBSTs)

In CS I at least one BBST was covered. AVL

Balanced BSTs (BBSTs)

In CS I at least one BBST was covered. AVL

Balanced by enforcing children depth restrictions

Balanced BSTs (BBSTs)

In CS I at least one BBST was covered. AVL

Balanced by enforcing children depth restrictions

More BBSTs exist

Balanced BSTs (BBSTs)

In CS I at least one BBST was covered. AVL

Balanced by enforcing children depth restrictions

More BBSTs exist

Most commonly Red-Black Trees and 2-4 Trees

Red Black Trees

Personal Favorite

Red Black Trees

Personal Favorite

Uses 5(ish) rules to ensure reduced worst case performance

Red Black Trees

Personal Favorite

Uses 5(ish) rules to ensure reduced worst case performance

1. Node must be colored red or black

Red Black Trees

Personal Favorite

Uses 5(ish) rules to ensure reduced worst case performance

1. Node must be colored red or black
2. All NULL nodes are considered black

Red Black Trees

Personal Favorite

Uses 5(ish) rules to ensure reduced worst case performance

1. Node must be colored red or black
2. All NULL nodes are considered black
3. No red node can have a red child

Red Black Trees

Personal Favorite

Uses 5(ish) rules to ensure reduced worst case performance

1. Node must be colored red or black
2. All NULL nodes are considered black
3. No red node can have a red child (insertion constraint)

Red Black Trees

Personal Favorite

Uses 5(ish) rules to ensure reduced worst case performance

1. Node must be colored red or black
2. All NULL nodes are considered black
3. No red node can have a red child (insertion constraint)
4. The **path** from the root to any NULL node must traverse an equal number of black node

Red Black Trees

Personal Favorite

Uses 5(ish) rules to ensure reduced worst case performance

1. Node must be colored red or black
2. All NULL nodes are considered black
3. No red node can have a red child (insertion constraint)
4. The **path** from the root to any NULL node must traverse an equal number of black nodes (removal constraint)

Red Black Trees

Personal Favorite

Uses 5(ish) rules to ensure reduced worst case performance

1. Node must be colored red or black
2. All NULL nodes are considered black
3. No red node can have a red child (insertion constraint)
4. The **path** from the root to any NULL node must traverse an equal number of black nodes (removal constraint)
5. The root is colored black

Red Black Trees

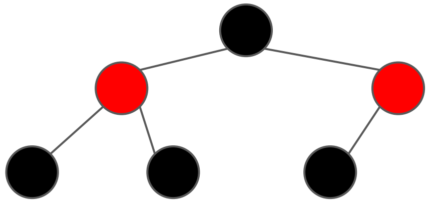
Personal Favorite

Uses 5(ish) rules to ensure reduced worst case performance

1. Node must be colored red or black
2. All NULL nodes are considered black
3. No red node can have a red child (insertion constraint)
4. The **path** from the root to any NULL node must traverse an equal number of black nodes (removal constraint)
5. (optional) The root is colored black

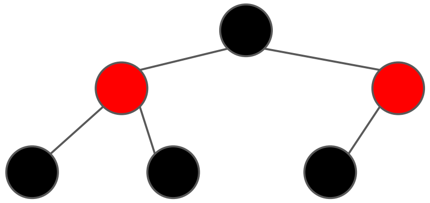
Examples?

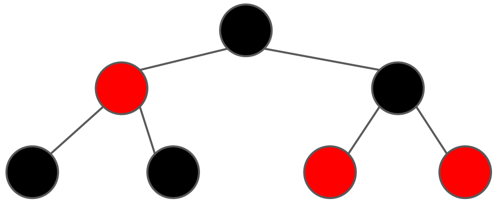
Assume circles are non-NULL nodes.



Examples?

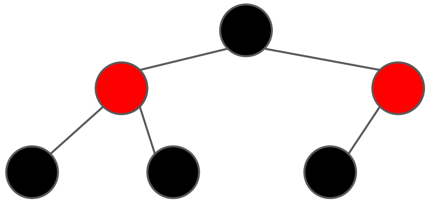
Assume circles are non-NULL nodes.

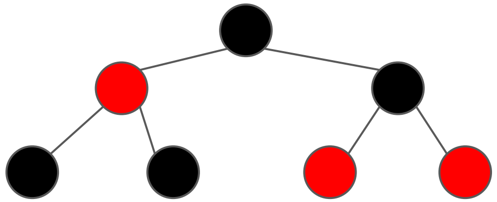


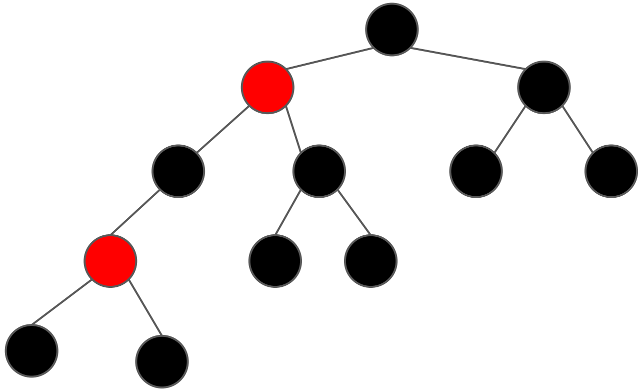


Examples?

Assume circles are non-NULL nodes.

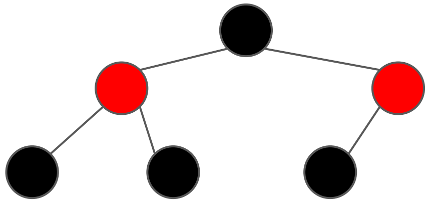


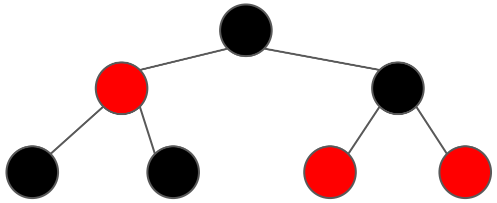


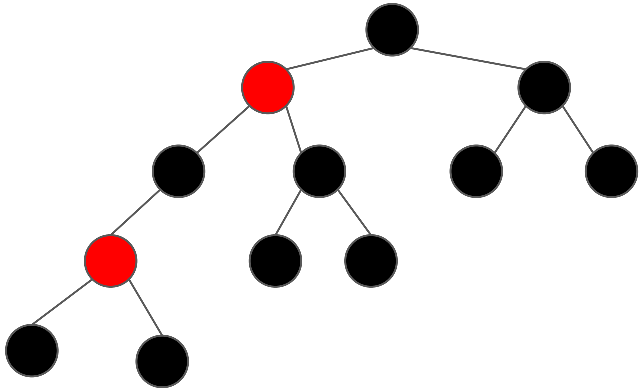


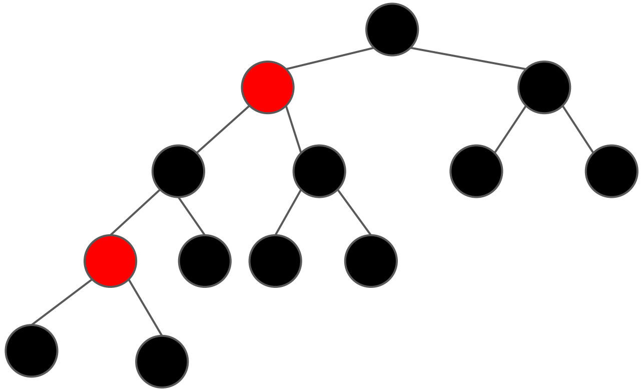
Examples?

Assume circles are non-NULL nodes.









Red Black Insertion

Insertion()

Red Black Insertion

Insertion()

- Insert the node using normal BST insertion

Red Black Insertion

Insertion()

- Insert the node using normal BST insertion
- Let the node be red

Red Black Insertion

Insertion()

- Insert the node using normal BST insertion
- Let the node be red
- Verify the rules have not been violated

Red Black Insertion

Insertion()

- Insert the node using normal BST insertion
- Let the node be red
- Verify the rules have not been violated (**not trivial**)

Red Black Insertion

Insertion()

- Insert the node using normal BST insertion
- Let the node be red
- Verify the rules have not been violated (**not trivial**)

Case 1: The node is the root

Case 2: The node has a black parent

Case 3: Red-Red

Red Black Insertion

Insertion()

- Insert the node using normal BST insertion
- Let the node be red
- Verify the rules have not been violated (**not trivial**)

Case 1: The node is the root

Red Black Insertion

Insertion()

- Insert the node using normal BST insertion
- Let the node be red
- Verify the rules have not been violated (**not trivial**)

Case 1: The node is the root

Case 2: The node has a black parent

Red Black Insertion

Insertion()

- Insert the node using normal BST insertion
- Let the node be red
- Verify the rules have not been violated (**not trivial**)

Case 1: The node is the root

Case 2: The node has a black parent

Case 3: Red-Red

Red Red

Case 3 sub cases

Red Red

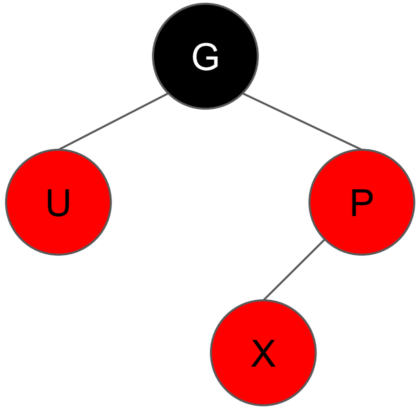
Case 3 sub cases

- Red Uncle (easy-ish)

Red Red

Case 3 sub cases

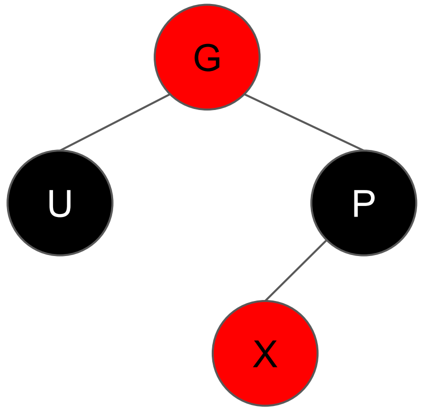
- Red Uncle (easy-ish)



Red Red

Case 3 sub cases

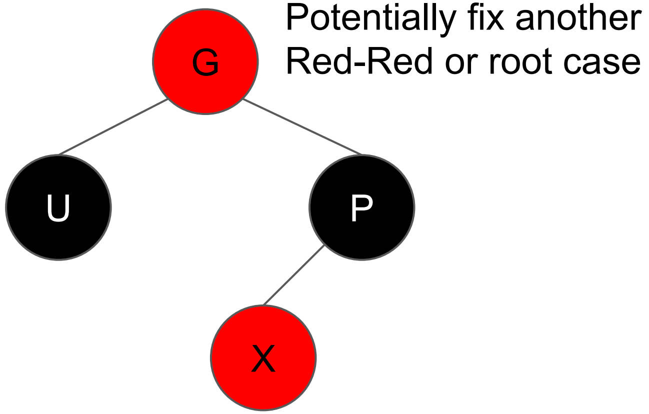
- Red Uncle (easy-ish)



Red Red

Case 3 sub cases

- Red Uncle (easy-ish)



Red Red

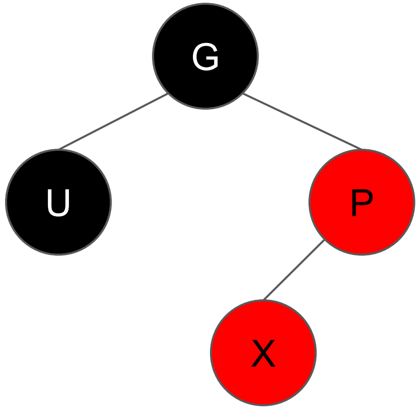
Case 3 sub cases

- Red Uncle (easy-ish)
- Black Uncle (rotations)

Red Red

Case 3 sub cases

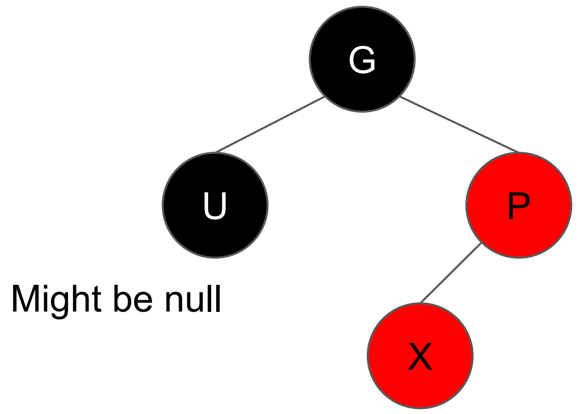
- Red Uncle (easy-ish)
- Black Uncle (rotations)



Red Red

Case 3 sub cases

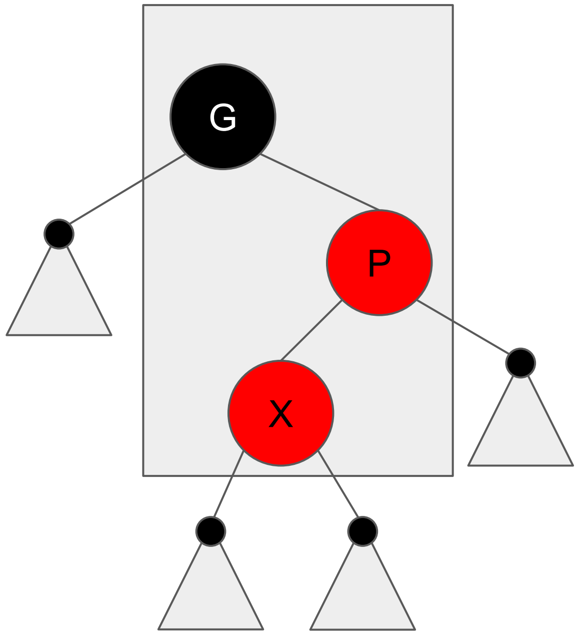
- Red Uncle (easy-ish)
- Black Uncle (rotations)



Red Red

Case 3 sub cases

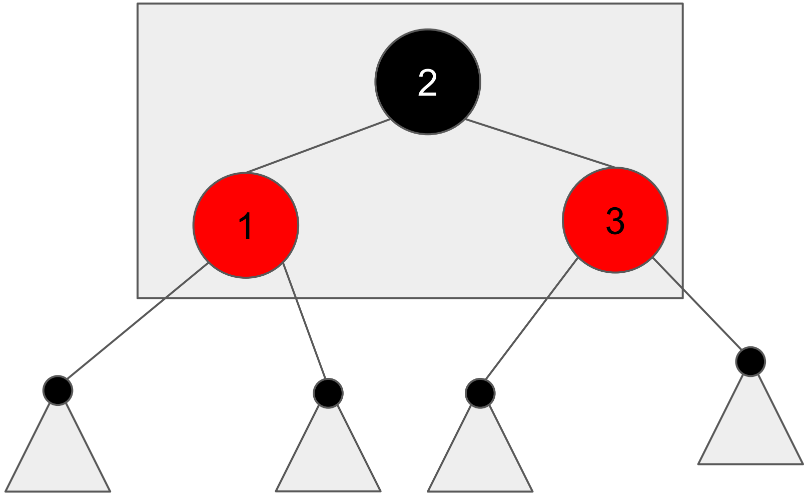
- Red Uncle (easy-ish)
- Black Uncle (rotations)



Red Red

Case 3 sub cases

- Red Uncle (easy-ish)
- Black Uncle (rotations)

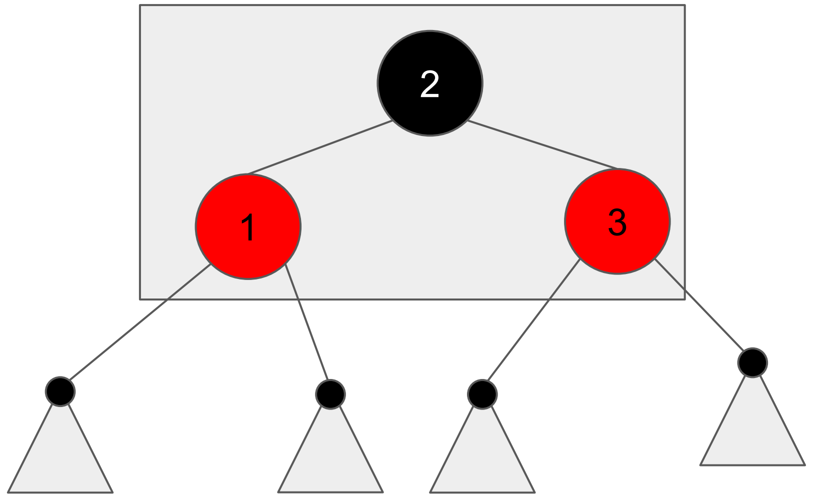


Red Red

Case 3 sub cases

- Red Uncle (easy-ish)
- Black Uncle (rotations)

Note: Only 1 rotation is needed



Red Black Removal

Remove()

Red Black Removal

Remove()

- Do the normal BST remove

Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)

Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove

Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring

Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Node is red

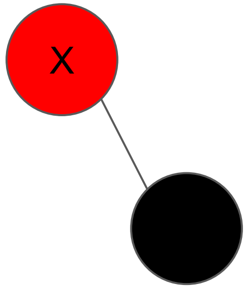
Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Node is red



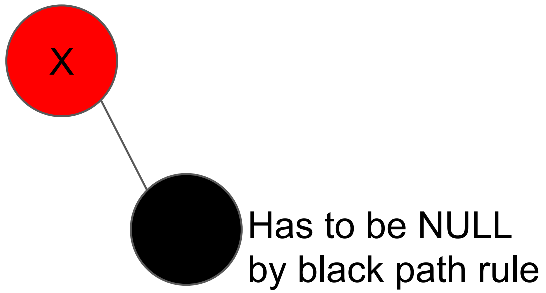
Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Node is red



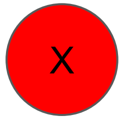
Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Node is red (easy removal)



Red Black Removal

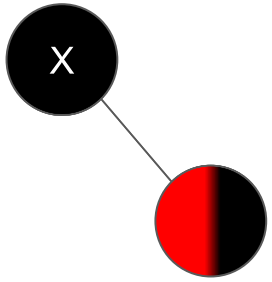
Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Node is red (easy removal)

Node is black



Has to be NULL or **red** by black path rule

Red Black Removal

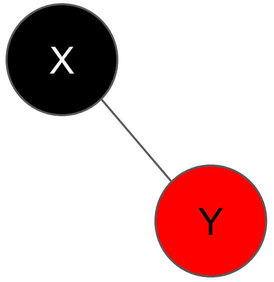
Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Node is red (easy removal)

Node is black



Has to be NULL or **red** by black path rule

Red Black Removal

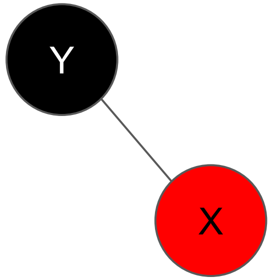
Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Node is red (easy removal)

Node is black



Has to be NULL or **red** by black path rule

Red Black Removal

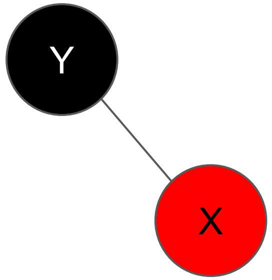
Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Node is red (easy removal)

Node is black



Has to be NULL or **red** by black path rule. Red is easy

Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Node is red (easy removal)

Node is black

Has to be NULL or **red** by black path rule

Red Black Removal

Remove()

- Do the normal BST remove (swap element into leaf/one child)
- Remove
- Fix potential problems with node coloring (also not trivial)

What cases could we encounter?

Node is red (easy removal)

Node is black (potentially not as easy removal)

Double Black Node

Caused by removing a black node with no replacement red child.

Double Black Node

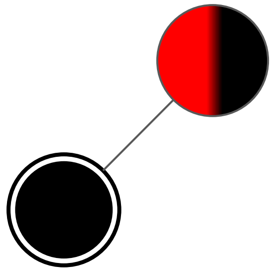
Caused by removing a black node with no replacement red child.

A node counts as two black nodes for the sake of the black path rule

Double Black Node

Caused by removing a black node with no replacement red child.

A node counts as two black nodes for the sake of the black path rule

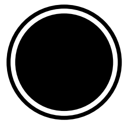


Double Black Node

Caused by removing a black node with no replacement red child.

A node counts as two black nodes for the sake of the black path rule

Case 1: The double black node is the root.

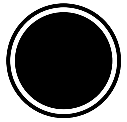


Double Black Node

Caused by removing a black node with no replacement red child.

A node counts as two black nodes for the sake of the black path rule

Case 1: The double black node is the root. (easy do nothing)



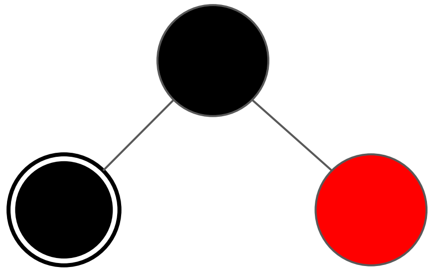
Double Black Node

Caused by removing a black node with no replacement red child.

A node counts as two black nodes for the sake of the black path rule

Case 1: The double black node is the root.

Case 2: The double black node has a red sibling



Double Black Node

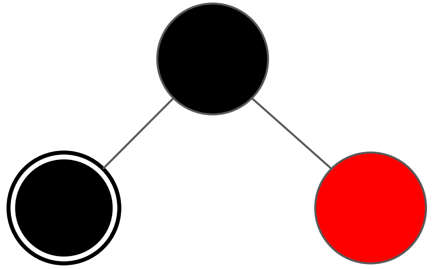
Caused by removing a black node with no replacement red child.

A node counts as two black nodes for the sake of the black path rule

Case 1: The double black node is the root.

Case 2: The double black node has a red sibling

Make it black



Double Black Node

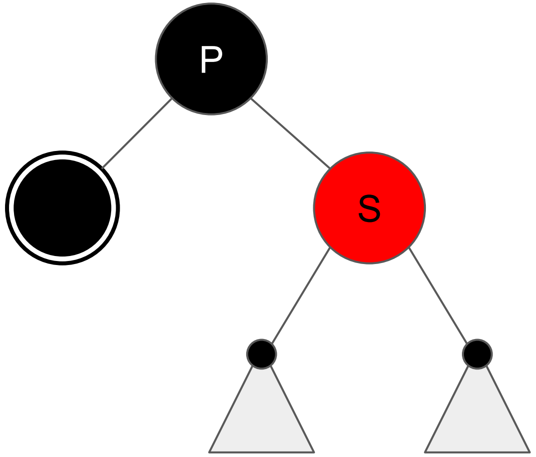
Caused by removing a black node with no replacement red child.

A node counts as two black nodes for the sake of the black path rule

Case 1: The double black node is the root.

Case 2: The double black node has a red sibling

Make it black through rotation



Double Black Node

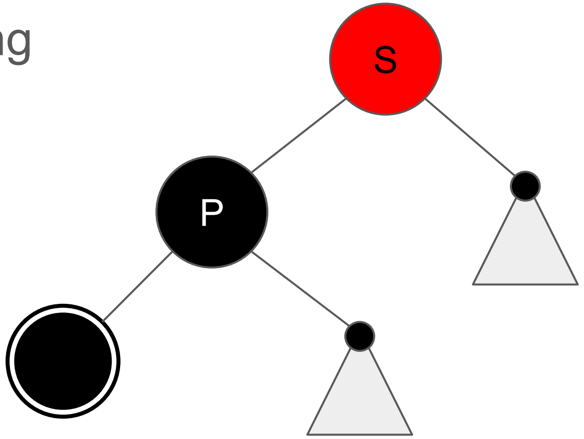
Caused by removing a black node with no replacement red child.

A node counts as two black nodes for the sake of the black path rule

Case 1: The double black node is the root.

Case 2: The double black node has a red sibling

Make it black through rotation



Double Black Node

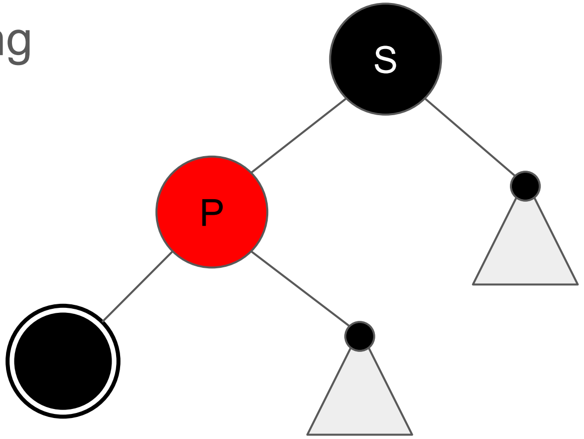
Caused by removing a black node with no replacement red child.

A node counts as two black nodes for the sake of the black path rule

Case 1: The double black node is the root.

Case 2: The double black node has a red sibling

Make it black through rotation



Double Black Node

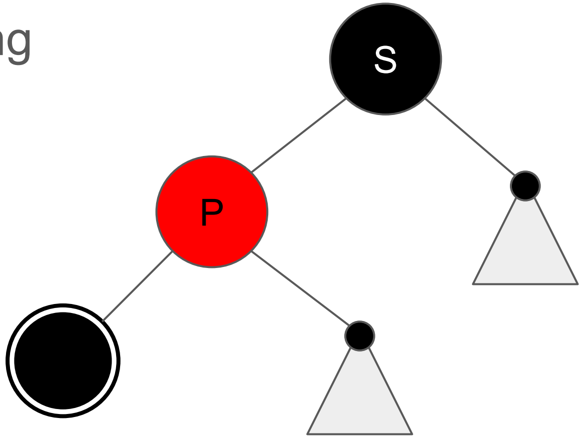
Caused by removing a black node with no replacement red child.

A node counts as two black nodes for the sake of the black path rule

Case 1: The double black node is the root.

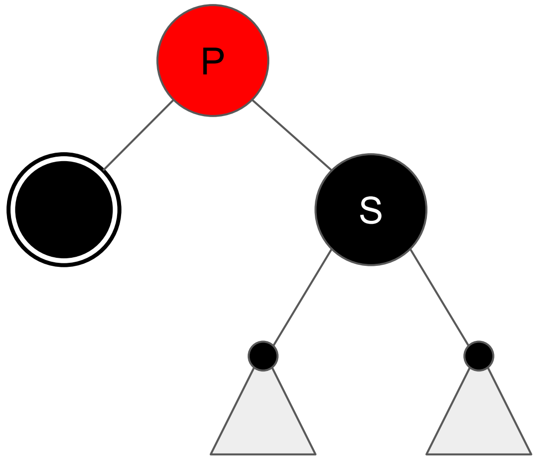
Case 2: The double black node has a red sibling

Make it black through rotation



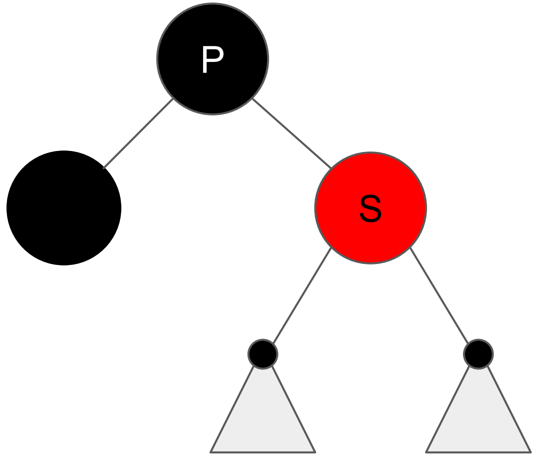
Double Black Node

Case 3: Parent is red, but nephews are black



Double Black Node

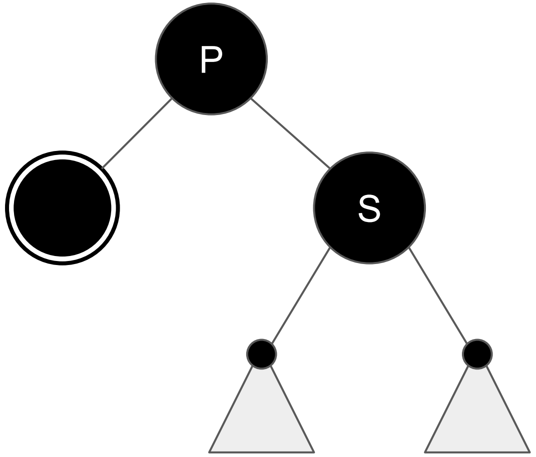
Case 3: Parent is red, but nephews are black (swap parent and sibling color)



Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

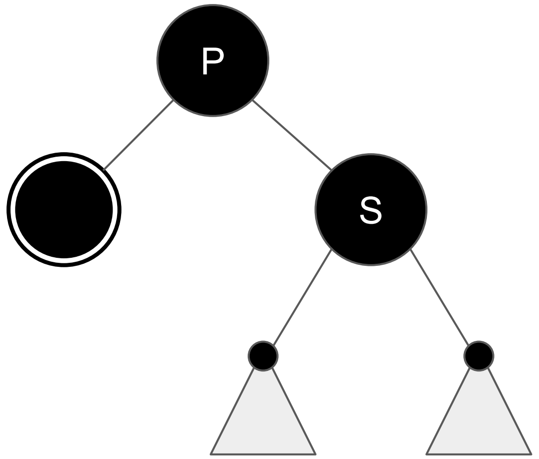
Case 4: Parent is black and so are nephews



Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

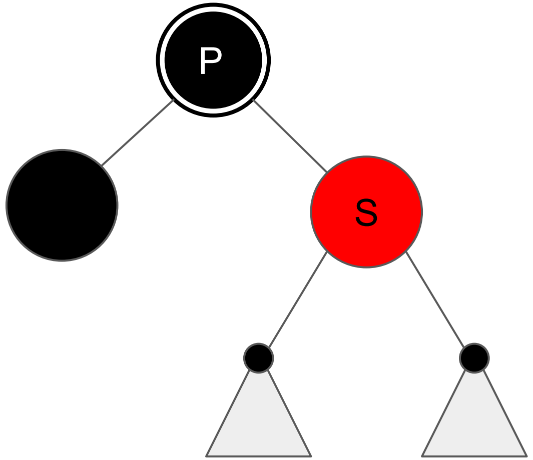
Case 4: Parent is black and so are nephews (Let our parent inherit our problem)



Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

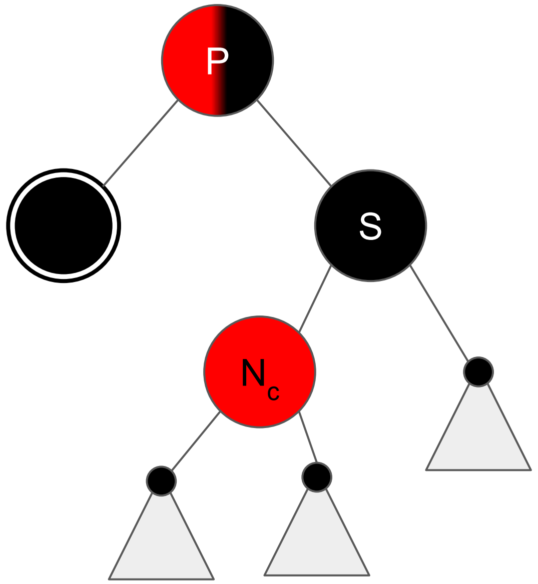


Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black



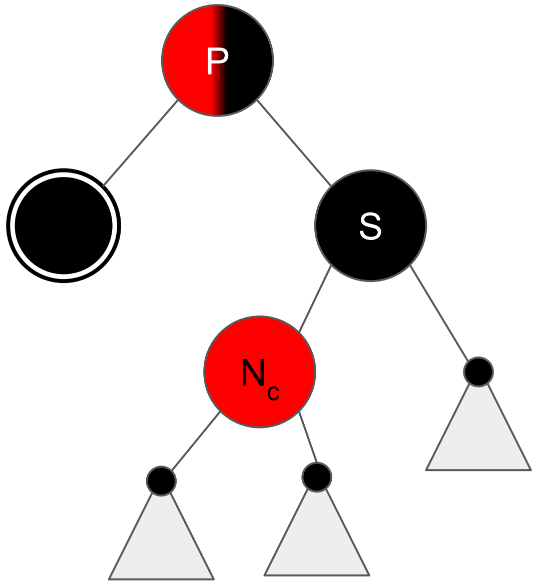
Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)



Double Black Node

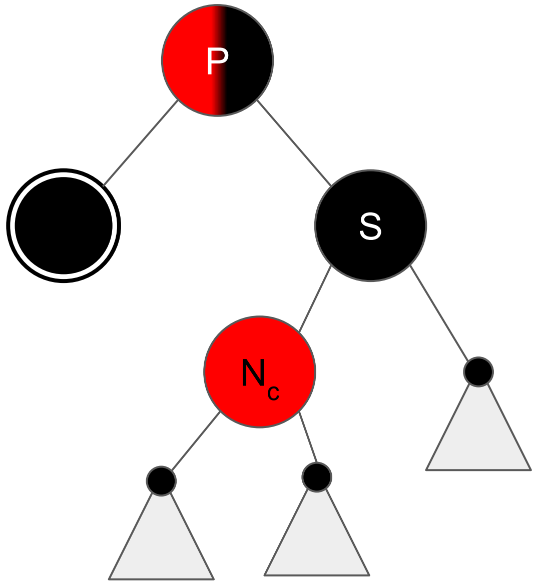
Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)

Consider the sibling



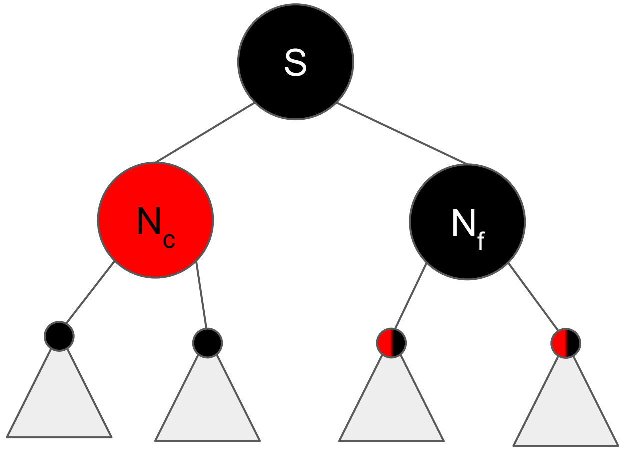
Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)



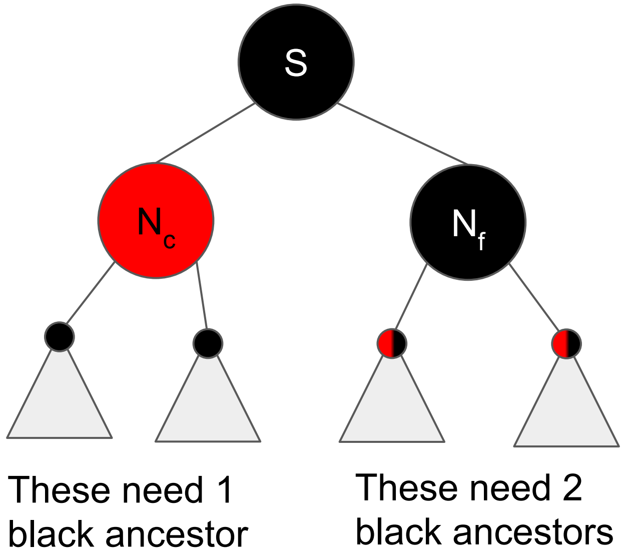
Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)



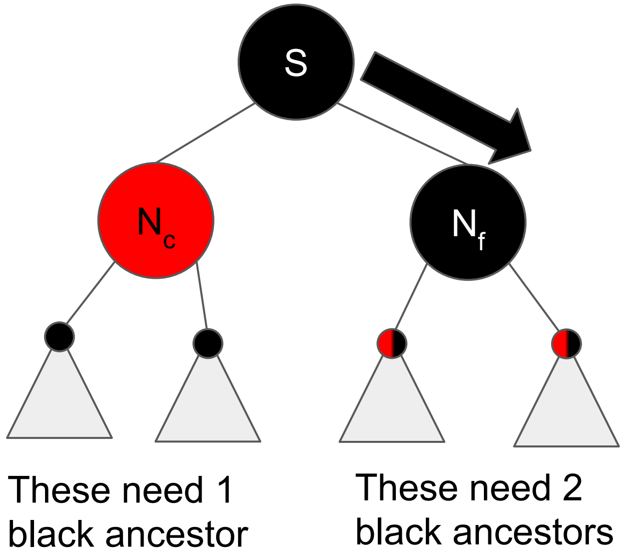
Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)



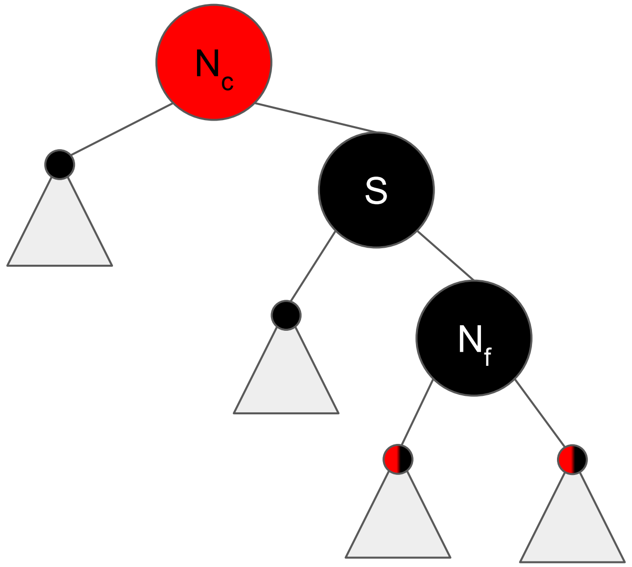
Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)



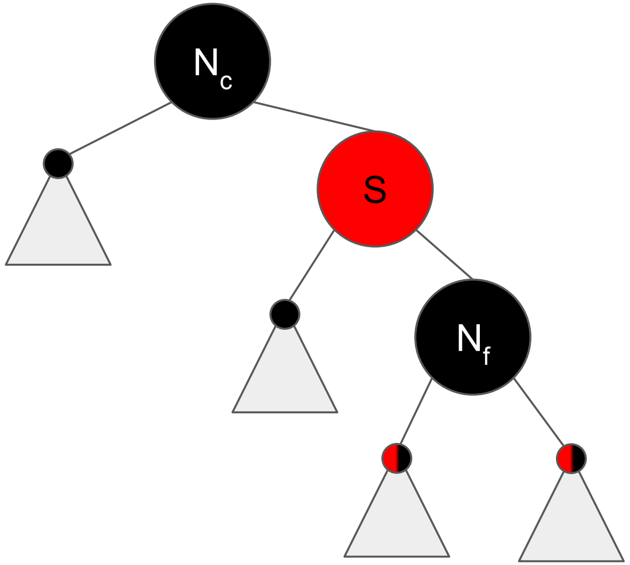
Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)



Double Black Node

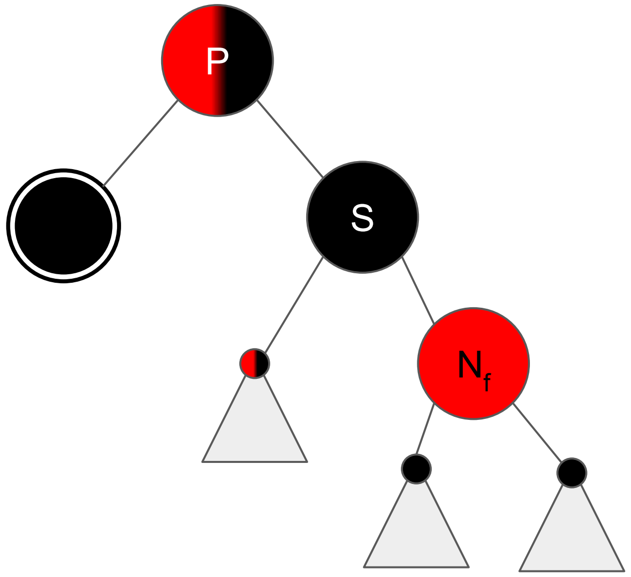
Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)

Case 6: Far nephew is red



Double Black Node

Case 3: Parent is red, but nephews are black (swap parent and sibling color)

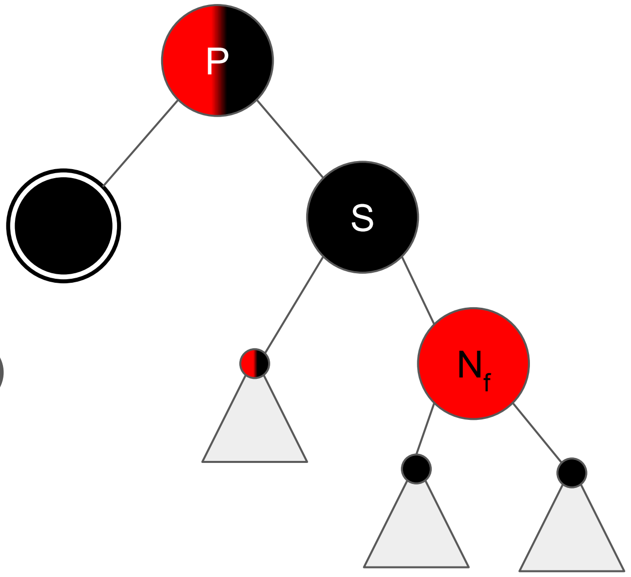
Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)

Case 6: Far nephew is red

(rotate the parent towards the double black node)



Double Black Node

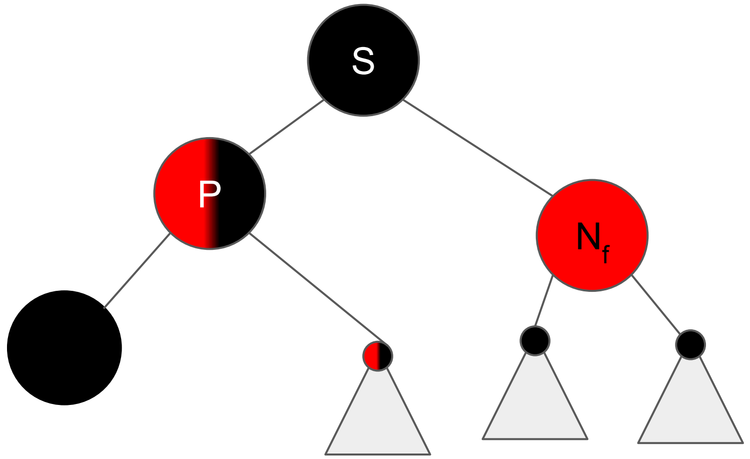
Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)

Case 6: Far nephew is red



Double Black Node

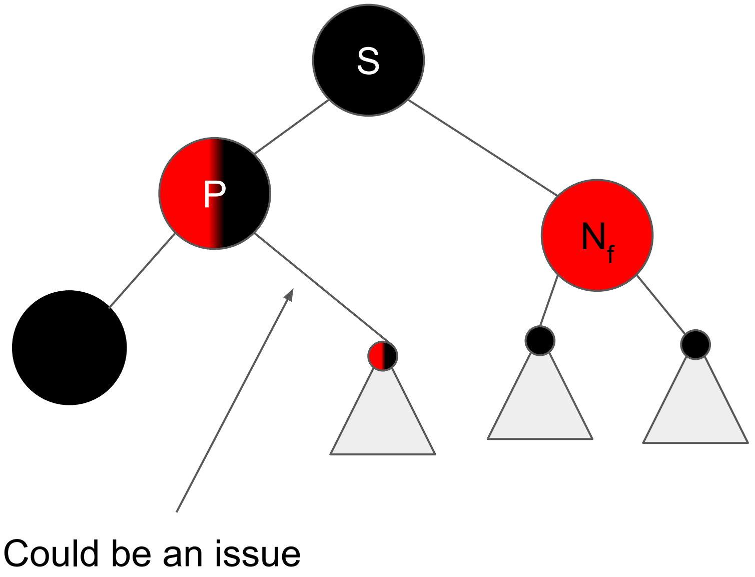
Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)

Case 6: Far nephew is red



Double Black Node

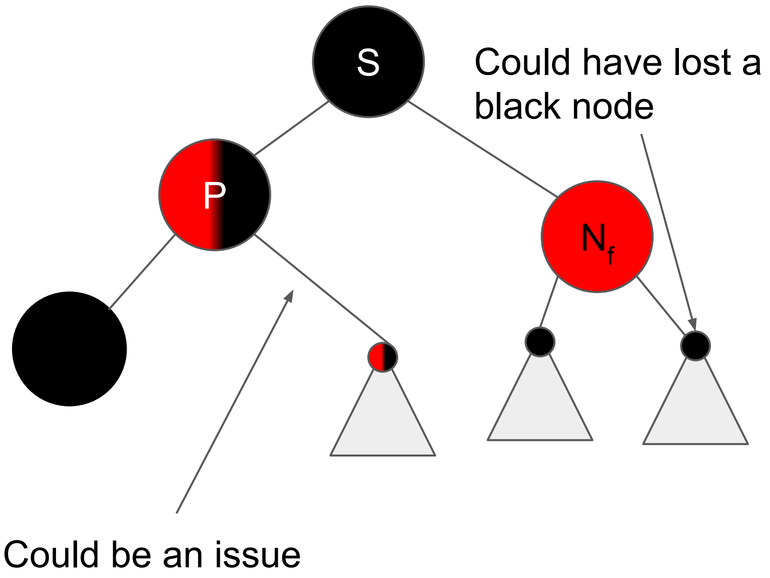
Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)

Case 6: Far nephew is red



Double Black Node

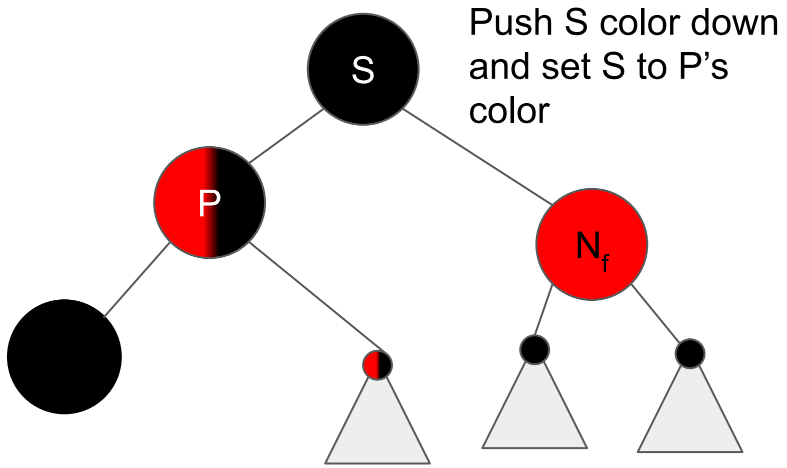
Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)

Case 6: Far nephew is red



Double Black Node

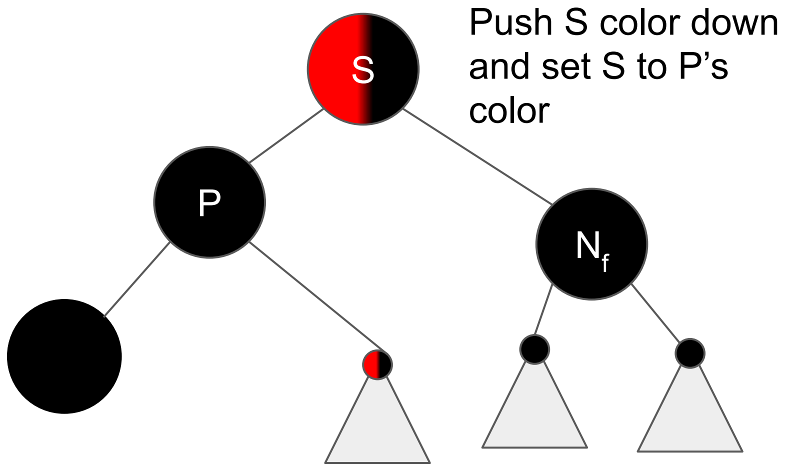
Case 3: Parent is red, but nephews are black (swap parent and sibling color)

Case 4: Parent is black and so are nephews (Let our parent inherit our problem)

Case 5: Close nephew is red and far is black

(Make far red)

Case 6: Far nephew is red



Runtime Proof

Insert when fixing the color will always work its way up the tree with each recursive call.

Runtime Proof

Insert when fixing the color will always work its way up the tree with each recursive call.

Remove will perform a “constant” number adjustments before moving the double black node up the tree.

Runtime Proof

Insert when fixing the color will always work its way up the tree with each recursive call.

Remove will perform a “constant” number adjustments before moving the double black node up the tree.

The maximum number of operations is thus the height of the tree.