Balanced Trees (***Red-Black*** Trees)

Warren Kim

Applications

Red-Black Trees have a variety of applications. Some include:

(i) Database indexing (2-4 trees[1] $\simeq$ ***red-black*** trees)[e.g. VoltDB]

[1] 2-4 (sometimes called 2-3-4) trees are a subset of B^+-trees.

Applications

Red-Black Trees have a variety of applications. Some include:

(i) Database indexing (2-4 trees[1] $\simeq$ ***red-black*** trees)[e.g. VoltDB]

(ii) Linux CPU scheduler (Completely Fair Scheduler)

[1] 2-4 (sometimes called 2-3-4) trees are a subset of B^+-trees.

Applications

Red-Black Trees have a variety of applications. Some include:

(i) Database indexing (2-4 trees[1] $\simeq$ ***red-black*** trees)[e.g. VoltDB]

(ii) Linux CPU scheduler (Completely Fair Scheduler)

(iii) Linux Virtual Memory Areas (VMA)

[1] 2-4 (sometimes called 2-3-4) trees are a subset of B^+-trees.

Applications

Red-Black Trees have a variety of applications. Some include:

(i) Database indexing (2-4 trees[1] $\simeq$ ***red-black*** trees)[e.g. VoltDB]

(ii) Linux CPU scheduler (Completely Fair Scheduler)

(iii) Linux Virtual Memory Areas (VMA)

(iv) STL Data Structures (e.g. C++'s `std::map`, Java's `HashMap`)

[1] 2-4 (sometimes called 2-3-4) trees are a subset of B^+-trees.

Applications

Red-Black Trees have a variety of applications. Some include:

(i) Database indexing (2-4 trees[1] $\simeq$ ***red-black*** trees)[e.g. VoltDB]

(ii) Linux CPU scheduler (Completely Fair Scheduler)

(iii) Linux Virtual Memory Areas (VMA)

(iv) STL Data Structures (e.g. C++'s `std::map`, Java's `HashMap`)

(v) Graph algorithm optimizations (for AI/ML)[e.g. K-mean clustering]

---

[1] 2-4 (sometimes called 2-3-4) trees are a subset of B^+-trees.

Motivation

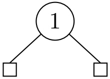
(i) Raw binary search tree performance is highly dependant on input order

(ii) We want to ensure $\mathcal{O}(\log n)$ performance

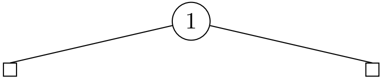
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



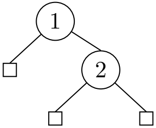
$\{1, 7, 2, 6, 3, 5, 4\}$



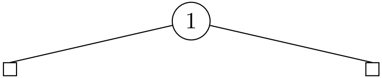
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



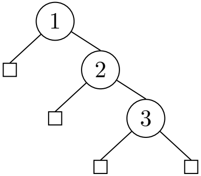
$\{1, 7, 2, 6, 3, 5, 4\}$



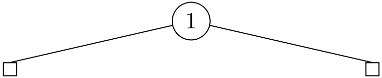
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



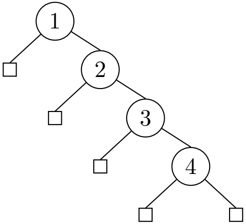
$\{1, 7, 2, 6, 3, 5, 4\}$



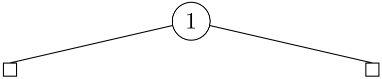
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



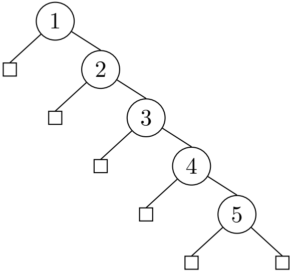
$\{1, 7, 2, 6, 3, 5, 4\}$



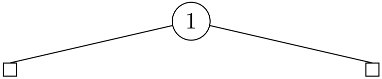
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



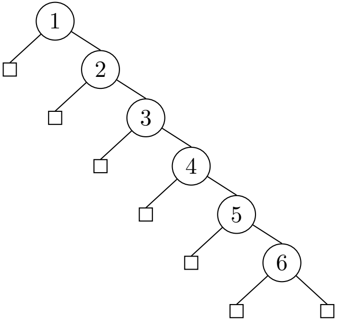
$\{1, 7, 2, 6, 3, 5, 4\}$



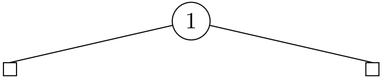
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



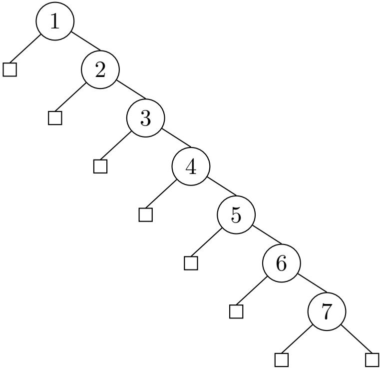
$\{1, 7, 2, 6, 3, 5, 4\}$



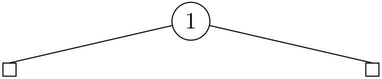
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



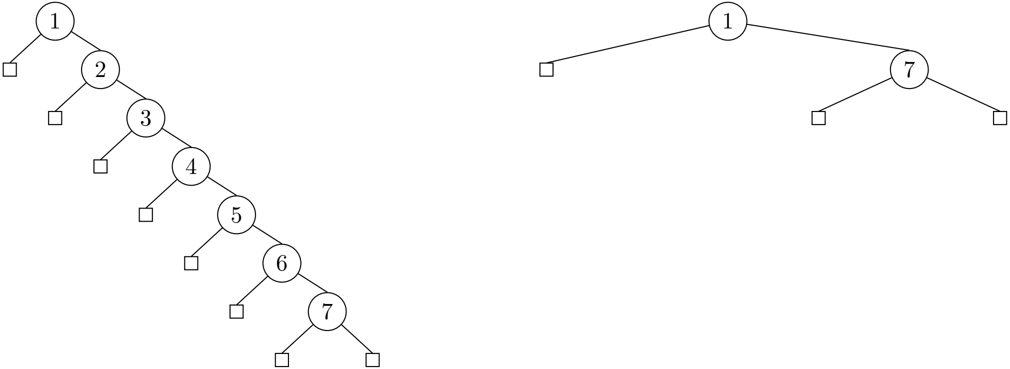
$\{1, 7, 2, 6, 3, 5, 4\}$



Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

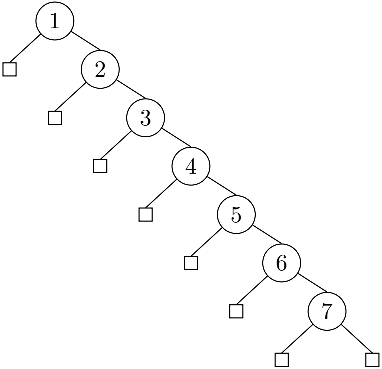
$\{1, 2, 3, 4, 5, 6, 7\}$ $\{1, 7, 2, 6, 3, 5, 4\}$



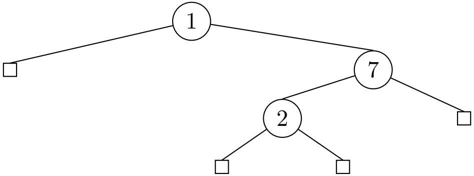
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



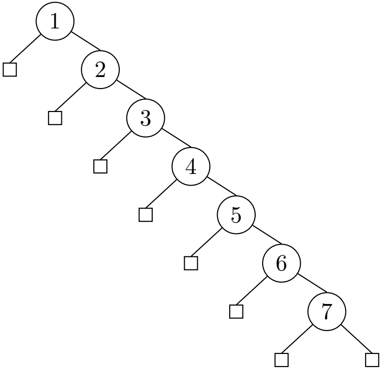
$\{1, 7, 2, 6, 3, 5, 4\}$



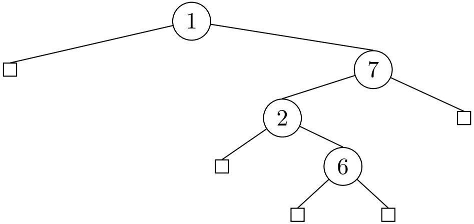
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



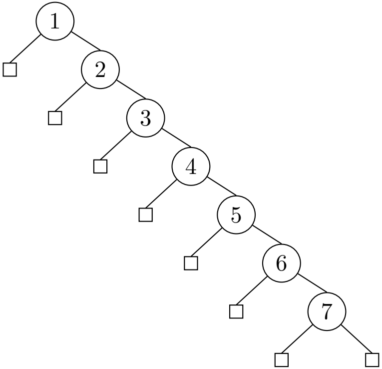
$\{1, 7, 2, 6, 3, 5, 4\}$



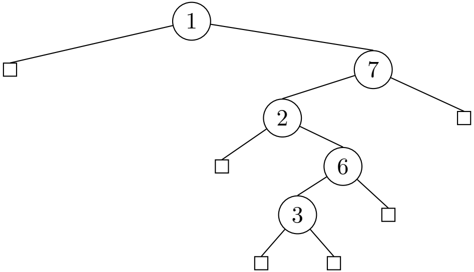
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



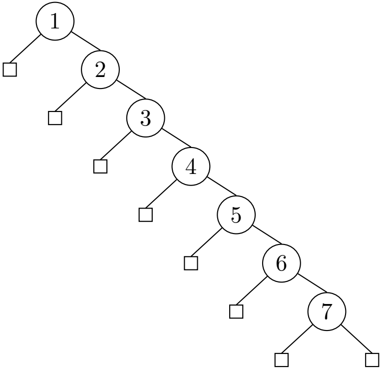
$\{1, 7, 2, 6, 3, 5, 4\}$



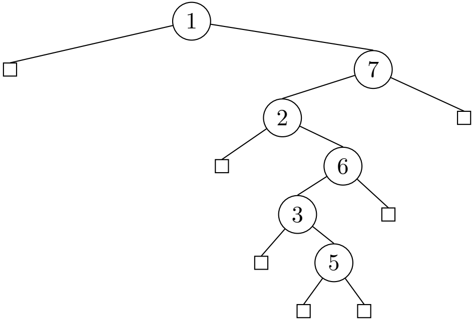
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



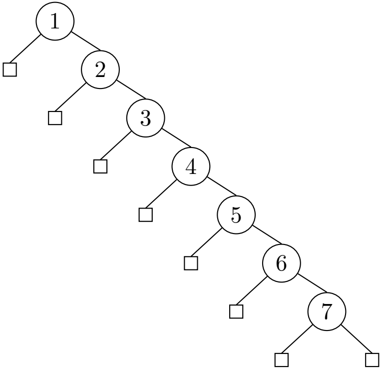
$\{1, 7, 2, 6, 3, 5, 4\}$



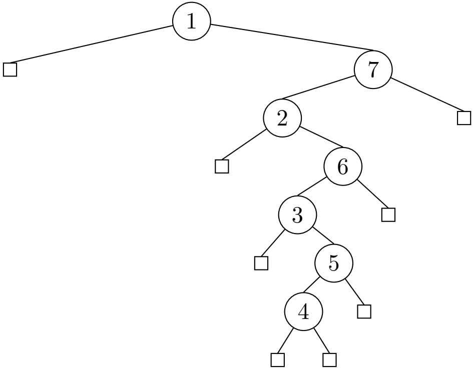
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$



$\{1, 7, 2, 6, 3, 5, 4\}$



Intuition

How do we balance binary trees?

Intuition

How do we balance binary trees?

We can *dynamically* balance the tree!

Intuition

How do we balance binary trees?

We can *dynamically* balance the tree!

We want to enforce a set of well-defined conditions. We can achieve this by adding additional member variables in our `Node` struct.

Definition and Properties

Definition

*A **red-black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ search, insertion, and deletion operations.*

Definition and Properties

Definition

A ***red-black*** *tree is a type of* ***self-balancing*** *binary search tree that guarantees* $\mathcal{O}(\log n)$ *search, insertion, and deletion operations.*

(i) *Color:* Every node is either ***red*** or ***black***

```
typedef enum Color { RED, BLACK };

struct Node {
    Color color;
    int data;
    Node *left;
    Node *right;
    Node *parent;
};
```

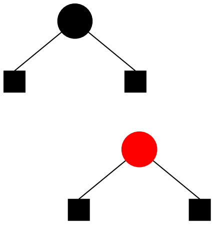
Definition and Properties

Definition

*A **red-black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ search, insertion, and deletion operations.*

(i) *Color:* Every node is either ***red*** or ***black***

(ii) *External:* All nil nodes are ***black***



Definition and Properties

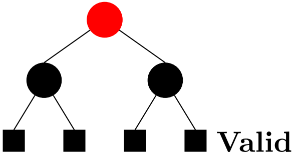
Definition

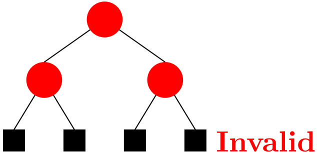
*A **red-black** tree is a type of **self-balancing** binary search tree that guarantees* $\mathcal{O}(\log n)$ *search, insertion, and deletion operations.*

(i) *Color:* Every node is either ***red*** or ***black***

(ii) *External:* All nil nodes are ***black***

(iii) *Internal:* A ***red*** node does not have a ***red*** child





Definition and Properties

Definition

*A **red-black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ search, insertion, and deletion operations.*

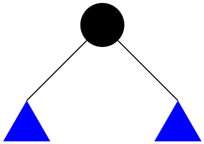
(i) *Color:* Every node is either ***red*** or ***black***

(ii) *External:* All nil nodes are ***black***

(iii) *Internal:* A ***red*** node does not have a ***red*** child

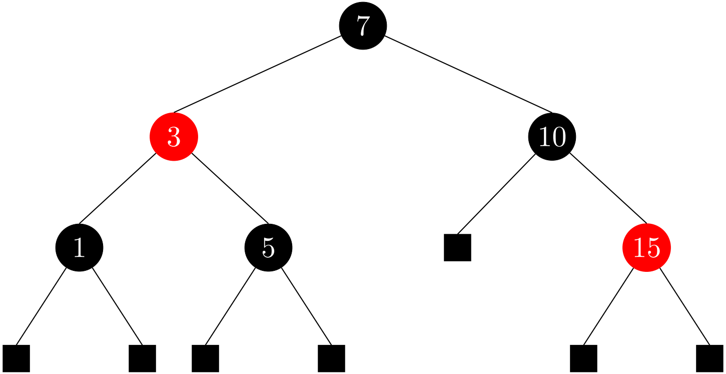
(iv) *Depth:* Every path from the root to *any* leaf node passes through the same number of ***black*** nodes

(v) *Root*: The root node is always ***black***



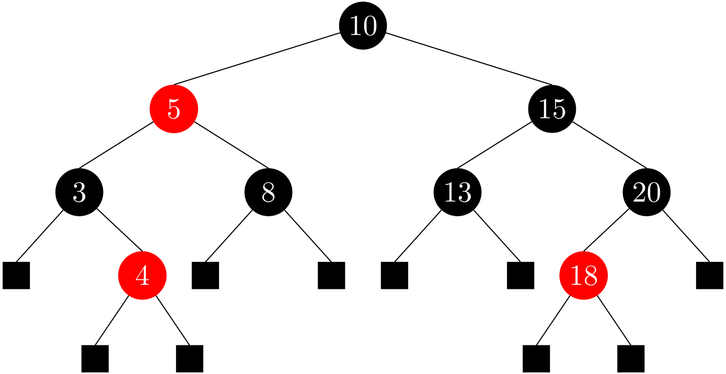
Depth Property

(iv) Depth: Every path from the root to *any* leaf node passes through the same number of ***black*** nodes



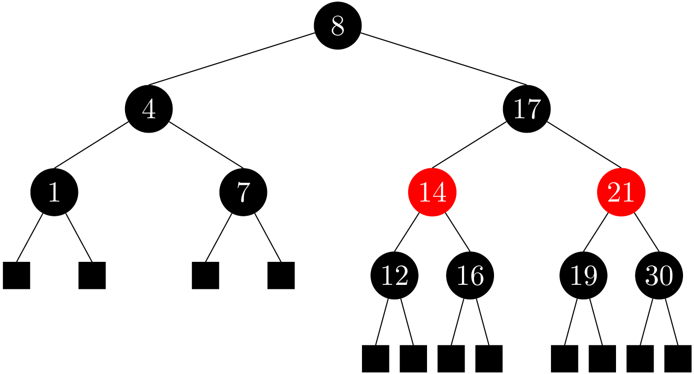
Depth Property

(iv) Depth: Every path from the root to *any* leaf node passes through the same number of ***black*** nodes



Depth Property

(iv) Depth: Every path from the root to *any* leaf node passes through the same number of ***black*** nodes



Definition and Properties

Definition

*A **red-black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ search, insertion, and deletion operations.*

(i) *Color:* Every node is either ***red*** or ***black***

(ii) *External:* All nil nodes are ***black***

(iii) *Internal:* A ***red*** node does not have a ***red*** child

(iv) *Depth:* Every path from the root to *any* leaf node passes through the same number of ***black*** nodes

(v) *Root*: The root node is always ***black***

Insertion

Suppose we have a node z to insert into our ***red-black*** tree. Then,

Insertion

Suppose we have a node z to insert into our ***red-black*** tree. Then,

(i) Like a BST, insert z.

Insertion

Suppose we have a node z to insert into our ***red-black*** tree. Then,

(i) Like a BST, insert z.

(ii) Color z ***red***.

Insertion

Suppose we have a node z to insert into our ***red-black*** tree. Then,

(i) Like a BST, insert z.

(ii) Color z ***red***.

(iii) Fix double ***red*** violations, if any.

Insertion

Suppose we have a node z to insert into our ***red-black*** tree. Then,

(i) Like a BST, insert z.

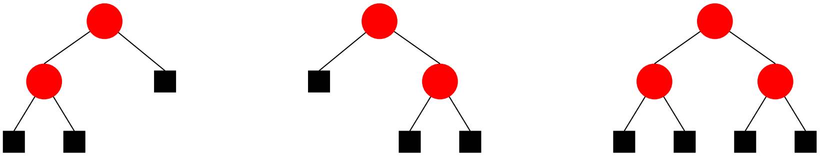
(ii) Color z ***red***.

(iii) Fix double ***red*** violations, if any.

(iv) Recursively fix violations upward.

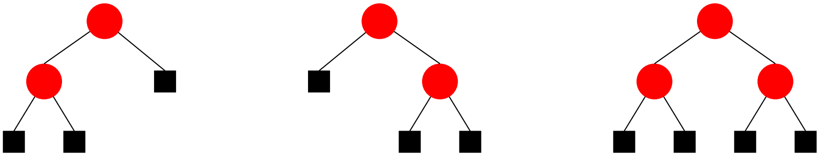
Double Red Violations

Recall *Property (iii): A **red** node does not have a **red** child.* All of the diagrams shown below are examples of ***invalid red-black*** trees.



Double Red Violations

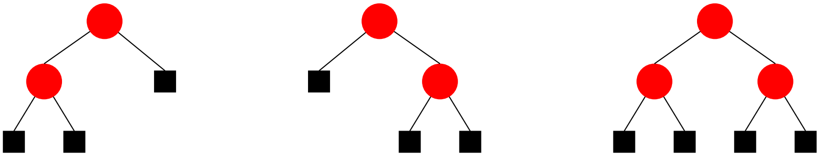
Recall *Property (iii): A* ***red*** *node does not have a* ***red*** *child.* All of the diagrams shown below are examples of ***invalid red-black*** trees.



There are two cases:

Double Red Violations

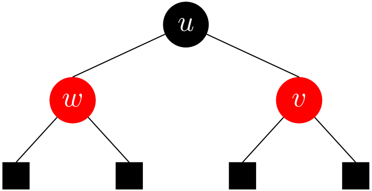
Recall *Property (iii): A **red** node does not have a **red** child.* All of the diagrams shown below are examples of ***invalid red-black*** trees.



There are two cases:

(i) ***Recolor:*** If both the *parent* and *uncle* are ***red***, perform a *recolor*.

Recolor



where

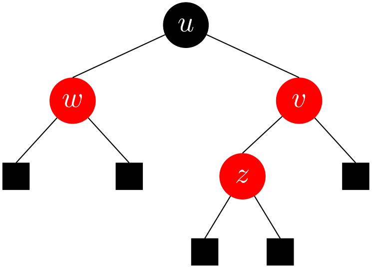
z is the new node

v is z's parent

u is z's grandparent

w is z's uncle

Recolor



where

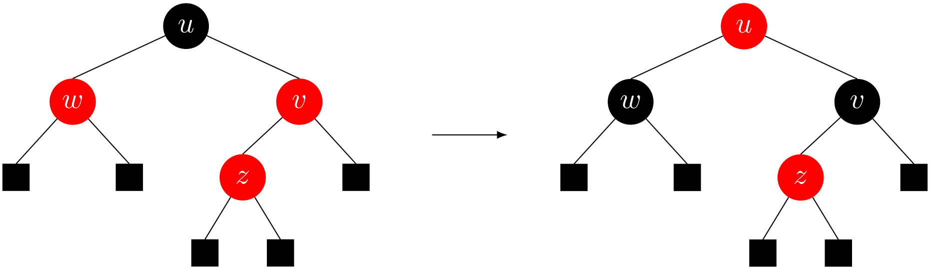
z is the new node

v is z's parent

u is z's grandparent

w is z's uncle

Recolor



where

z is the new node

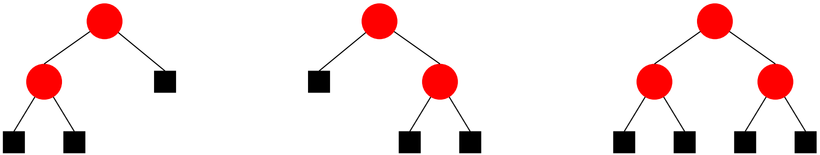
v is z's parent

u is z's grandparent

w is z's uncle

Double Red Violations

Recall *Property (iii): A* ***red*** *node does not have a* ***red*** *child.* All of the diagrams shown below are examples of ***invalid red-black*** trees.



There are two cases:

(i) ***Recolor:*** If both the *parent* and *uncle* are ***red***, perform a *recolor*.

(ii) ***Restructure:*** If the *parent* is ***red*** but the *uncle* is ***black***, perform a *tri-node restructure*.

Tri-Node Restructure

There are four cases:

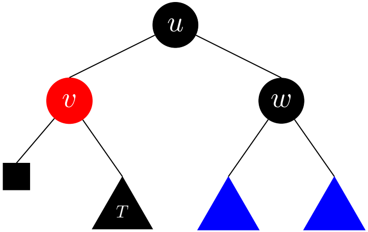
(i) Left-Left

(ii) Right-Right

(iii) Left-Right

(iv) Right-Left

Case: Left-Left



where

z is the new node

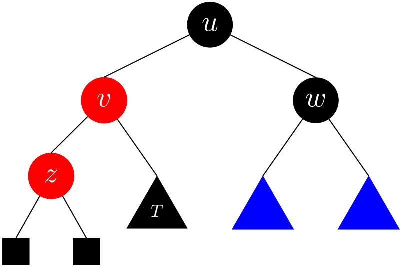
v is z's parent

u is z's grandparent

w is z's uncle

T is v's subtree

Case: Left-Left



where

z is the new node

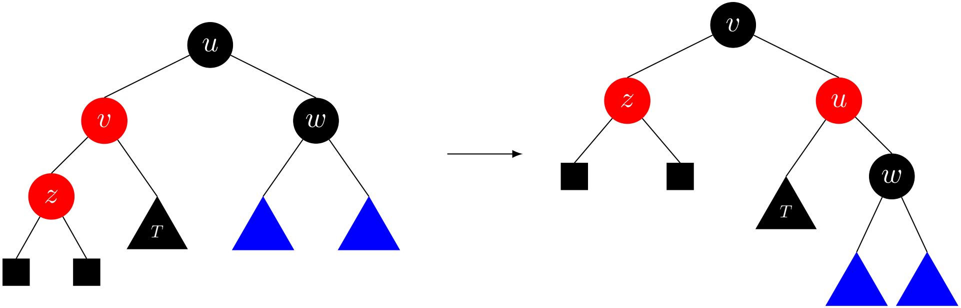
v is z's parent

u is z's grandparent

w is z's uncle

T is v's subtree

Case: Left-Left



where

z is the new node

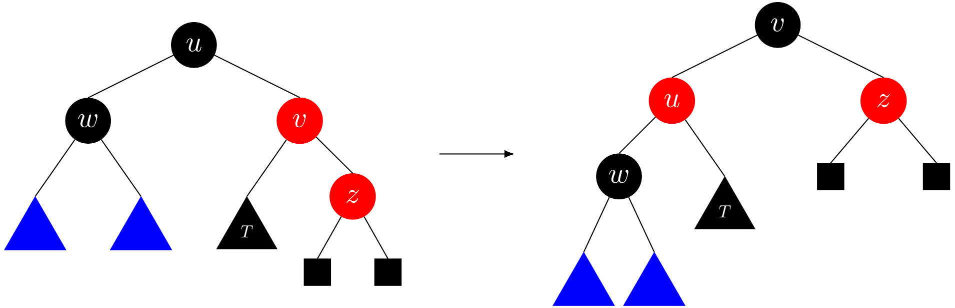
v is z's parent

u is z's grandparent

w is z's uncle

T is v's subtree

Case: Right-Right



where

z is the new node

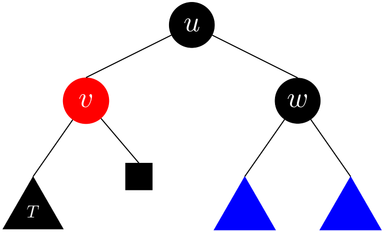
v is z's parent

u is z's grandparent

w is z's uncle

T is v's subtree

Case: Left-Right



where

z is the new node

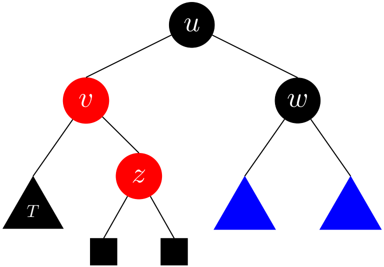
v is z's parent

u is z's grandparent

w is z's uncle

T is v's subtree

Case: Left-Right



where

z is the new node

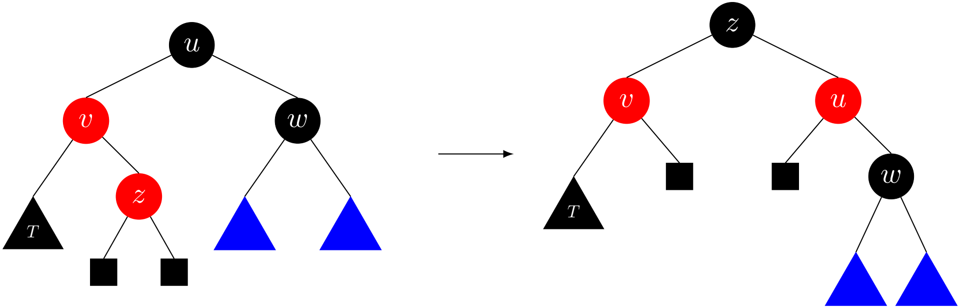
v is z's parent

u is z's grandparent

w is z's uncle

T is v's subtree

Case: Left-Right



where

z is the new node

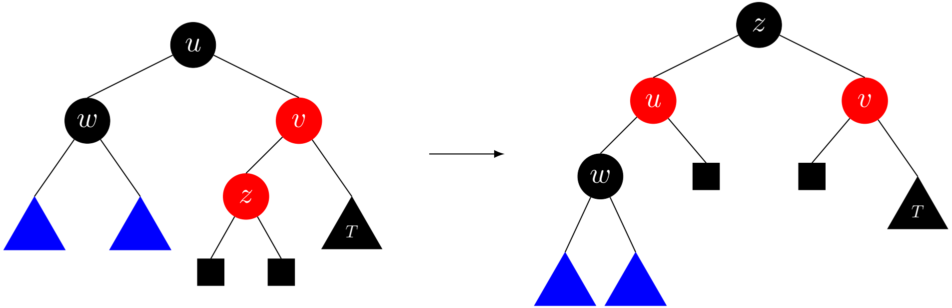
v is z's parent

u is z's grandparent

w is z's uncle

T is v's subtree

Case: Right-Left



where

z is the new node

v is z's parent

u is z's grandparent

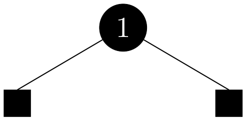
w is z's uncle

T is v's subtree

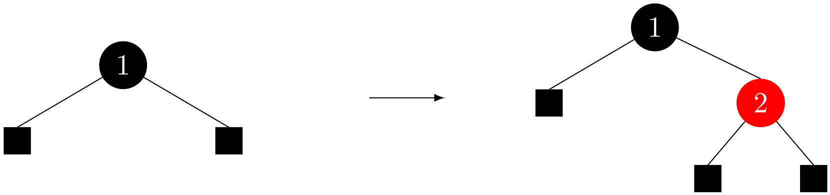
Time and Space Complexities

Insertion: $\mathcal{O}(\log n)$
Deletion: $\mathcal{O}(\log n)$
Search: $\mathcal{O}(\log n)$

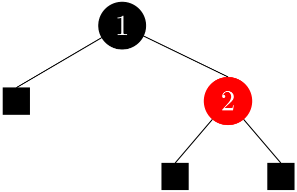
Red-Black Tree: Example



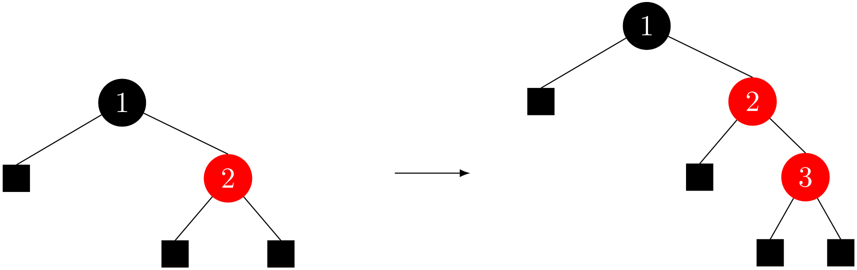
Red-Black Tree: Example



Red-Black Tree: Example

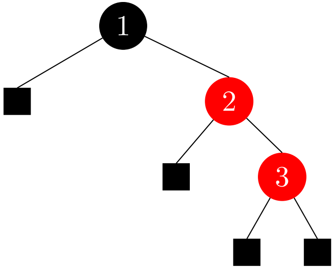


Red-Black Tree: Example



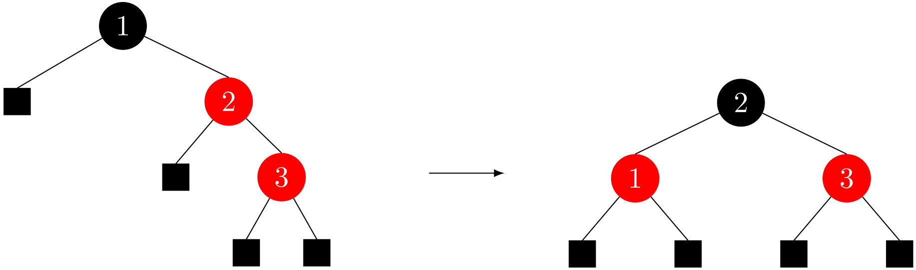
Case: Right-Right

Red-Black Tree: Example



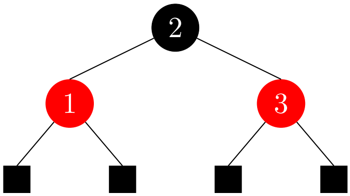
Case: Right-Right

Red-Black Tree: Example

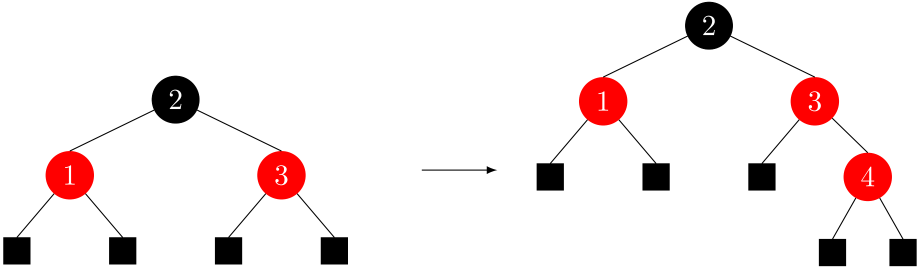


Case: Right-Right

Red-Black Tree: Example

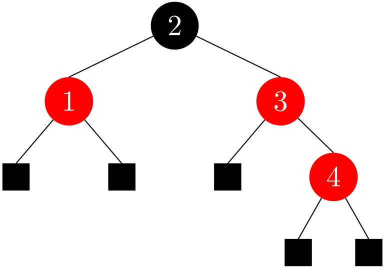


Red-Black Tree: Example



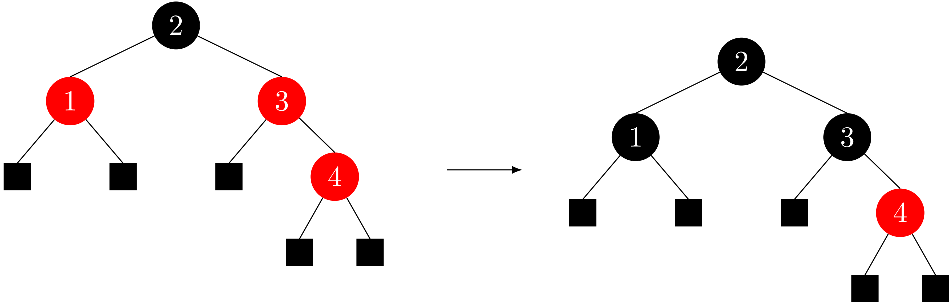
Case: Recolor

Red-Black Tree: Example



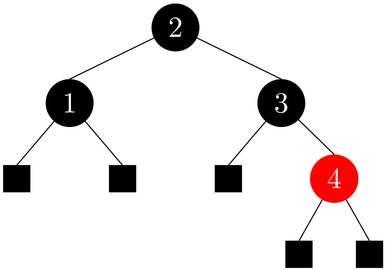
Case: Recolor

Red-Black Tree: Example

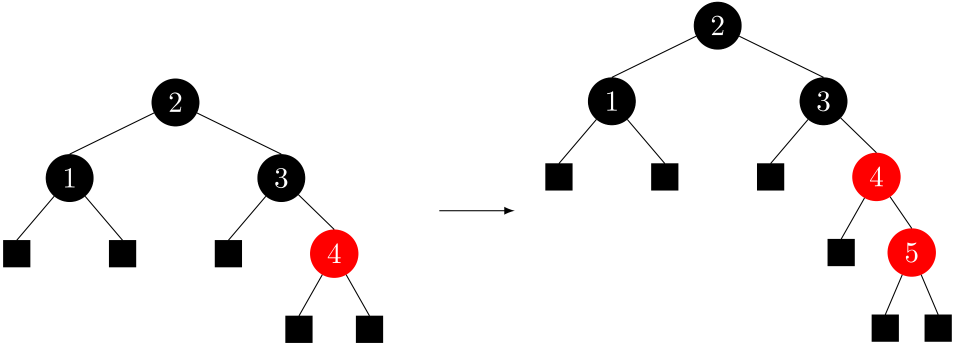


Case: Recolor

Red-Black Tree: Example

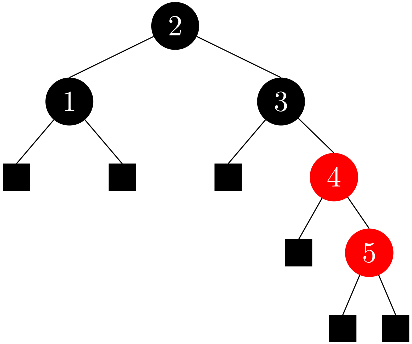


Red-Black Tree: Example



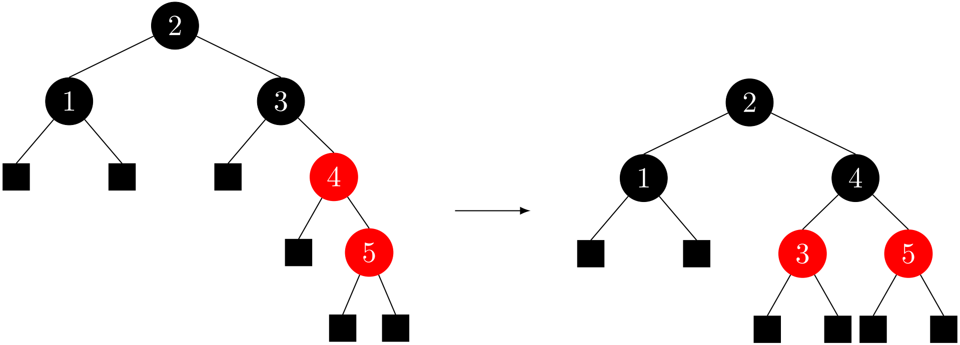
Case: Right-Right

Red-Black Tree: Example



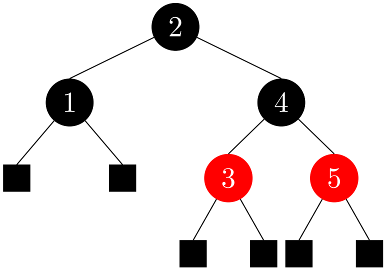
Case: Right-Right

Red-Black Tree: Example

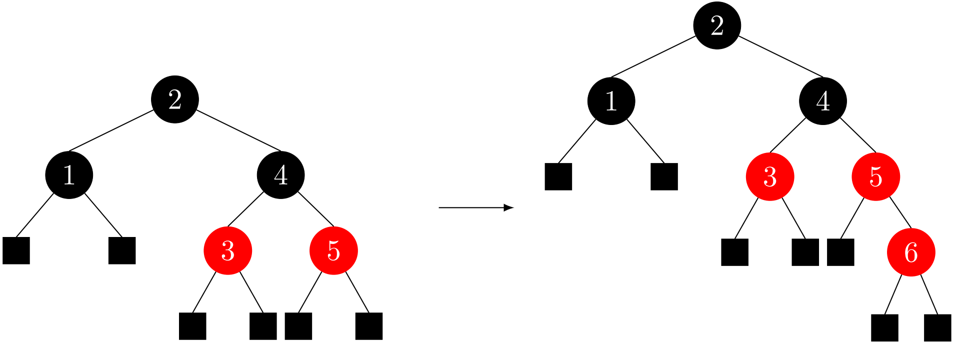


Case: Right-Right

Red-Black Tree: Example

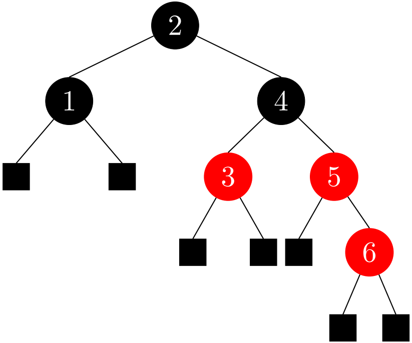


Red-Black Tree: Example



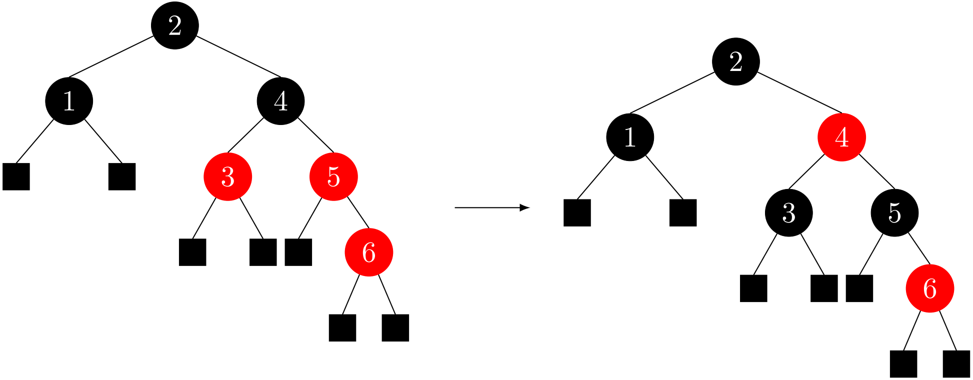
Case: Recolor

Red-Black Tree: Example



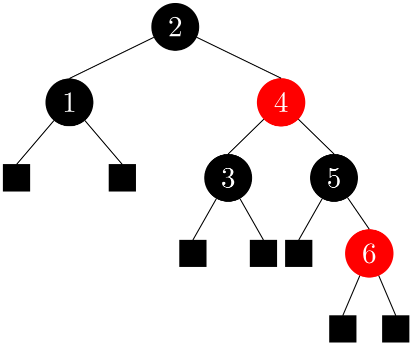
Case: Recolor

Red-Black Tree: Example

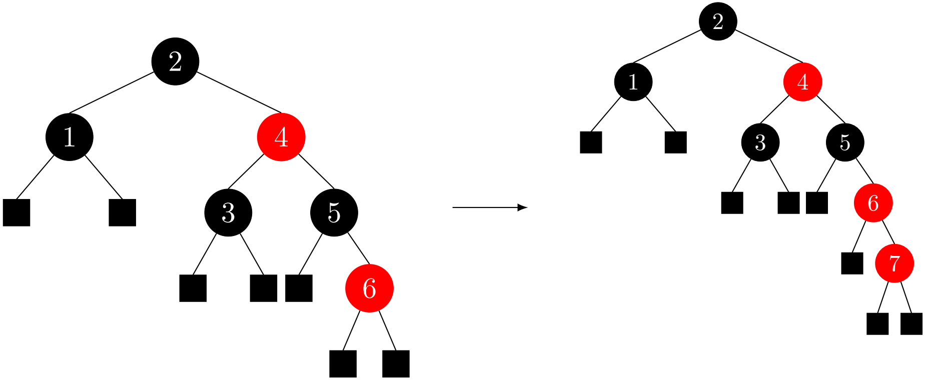


Case: Recolor

Red-Black Tree: Example

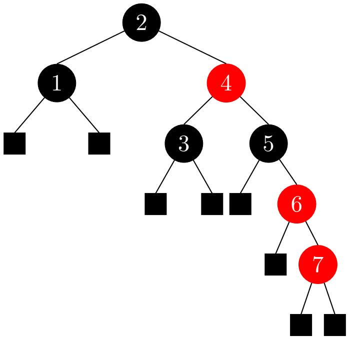


Red-Black Tree: Example

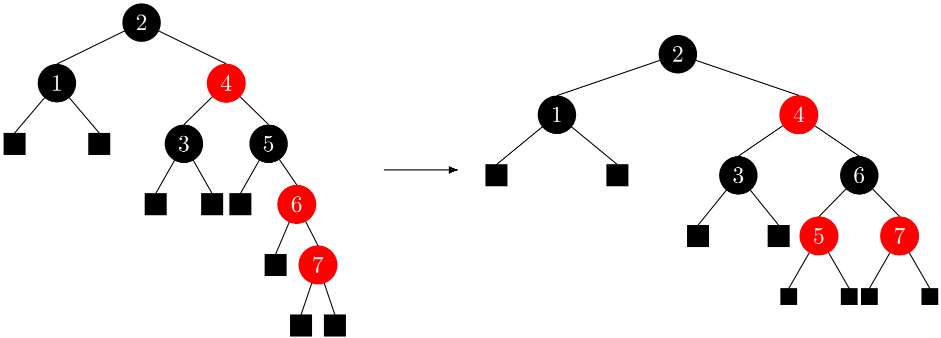


Case: Right-Right

Red-Black Tree: Example



Red-Black Tree: Example



End

Thank you!

Appendix

Below are slides that didn't make the cut :(

Corollaries

Proposition

If a node n has exactly one child, c, then **(a)** *c is* ***red*** *,* **(b)** *n is* ***black****, and* **(c)** *c has no children.*

Proof. Suppose we have a valid red-black tree. Consider a node n with exactly one child. Without loss of generality, choose n's left node to be the child and call it c.

(a) n passes through no ***black*** nodes on the right side by assumption. If c were ***black***, then n would pass through 1 ***black*** node, a contradiction since this violates the *depth property*.

(b) By **(a)**, n's child is ***red*** and by the *internal property*, n is ***black***.

(c) Since n passes through no ***black*** nodes on the right side by assumption, n cannot pass through any ***black*** nodes on the left side by the *the depth property*. Then, since c is ***red*** by **(a)**, c has only nil nodes, which are ***black*** by the *external property*. □

Height of a Red-Black Tree

Theorem

A red-black tree with n nodes has a height h that is $\mathcal{O}(\log n)$.

Proof. Suppose we have an arbitrary (valid) red-black tree. Let b be the number of ***black*** nodes on the shortest path from root to any leaf. In the worst case, the shortest path alternates between ***red*** and ***black*** nodes and thus has a height of $2b$. Then, h is bounded above by $2b$; that is, $h \leq 2b$. There are $2^b - 1 \leq n$ nodes in this tree. Solving for b, we get $b \leq \log(n+1)$. Substituting b, we get $b \leq \log(n+1) \leq h \leq 2b \leq 2\log(n+1)$ so h is bounded below by $\log(n+1)$ and above by $2\log(n+1)$; that is, $\log(n+1) \leq h \leq 2\log(n+1)$. So, h is $\mathcal{O}(\log n)$. $\square$

Deletion

Suppose we have a node z to delete from our red-black tree.

(i) If z is a ***red*** leaf node, simply remove z.

(ii) If z only has a left child s, swap *values* and remove s.

(iii) If z has a right child, swap its *value* with its in-order successor s and remove s.

If there is a double ***black*** in *(ii)* or *(iii)*, perform a *fixup* at y's original position.

Deletion Fixup

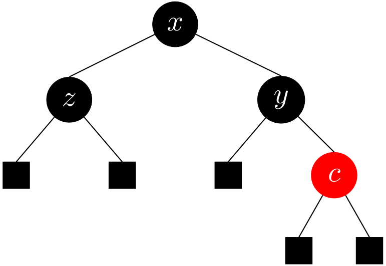
There are three cases:

(i) y has a ***red*** child. Then, we perform a ***restructure***.

(ii) y's sibling w is ***black***. Then, we perform a ***recolor***.

(iii) y's sibling w is ***red*** . Then, we perform an ***adjustment*** followed by either case *(ii)* or *(iii)*.

Restructure



where

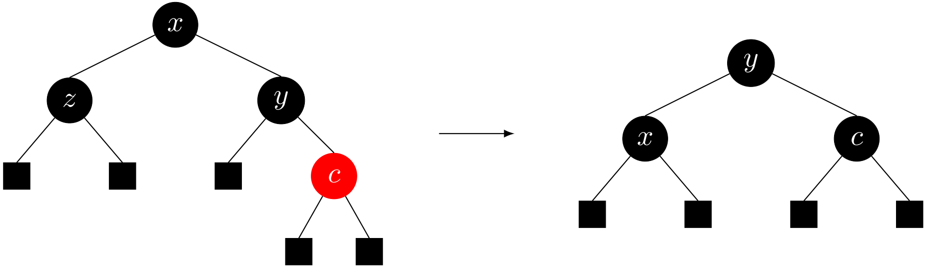
z is the node to delete

y is z's sibling

x is y's parent

c is y's child

Restructure



where

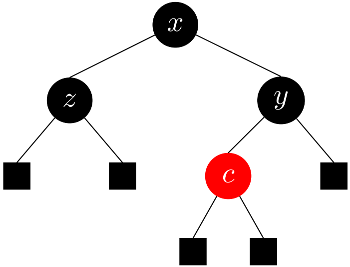
z is the node to delete

y is z's sibling

x is y's parent

c is y's child

Restructure



where

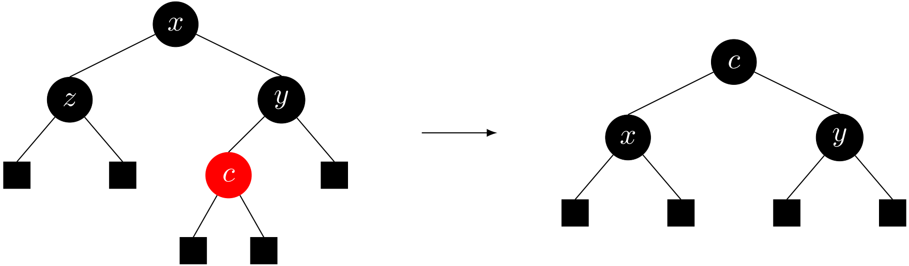
z is the node to delete

y is z's sibling

x is y's parent

c is y's child

Restructure



where

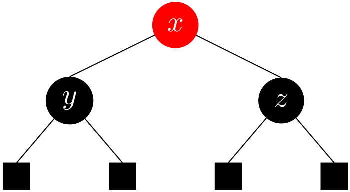
z is the node to delete

y is z's sibling

x is y's parent

c is y's child

Recolor



where

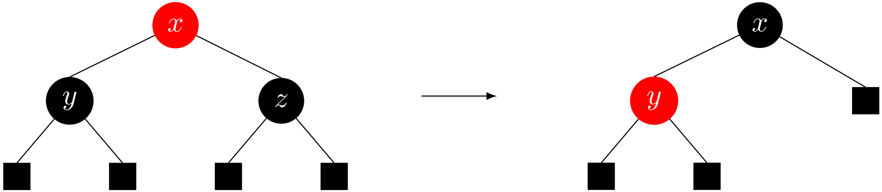
z is the node to delete

y is z's sibling

x is y's parent

c is y's child

Recolor



where

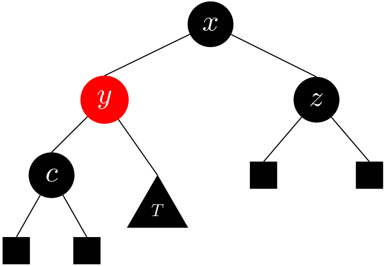
z is the node to delete

y is z's sibling

x is y's parent

c is y's child

Adjustment



where

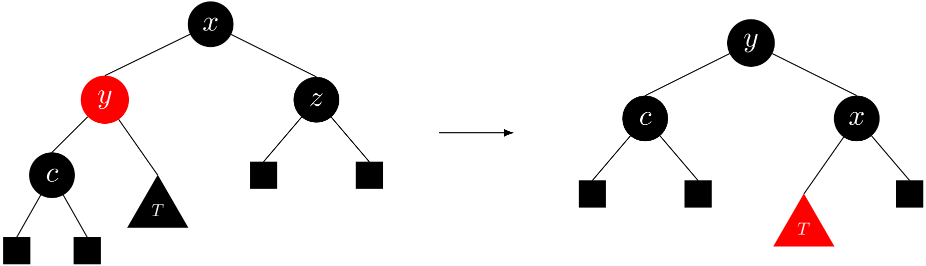
z is the node to delete

y is z's sibling

x is y's parent

c is y's child

Adjustment



where

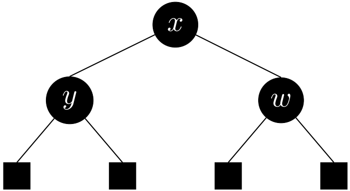
z is the node to delete

y is z's sibling

x is y's parent

c is y's child

Case: y is a Leaf Node or has Two ***Black*** Children



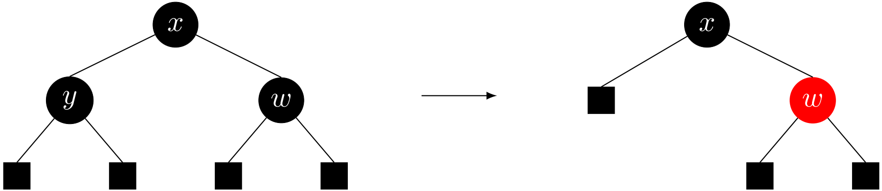
where

y is the node to delete

w is y's sibling

x is y's parent

Case: y is a Leaf Node or has Two ***Black*** Children



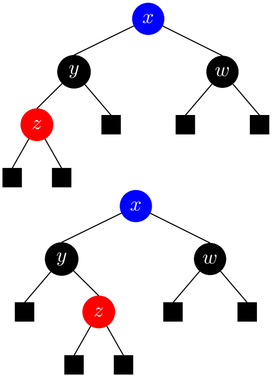
where

y is the node to delete

w is y's sibling

x is y's parent

Case: y has a ***Red*** Child



where

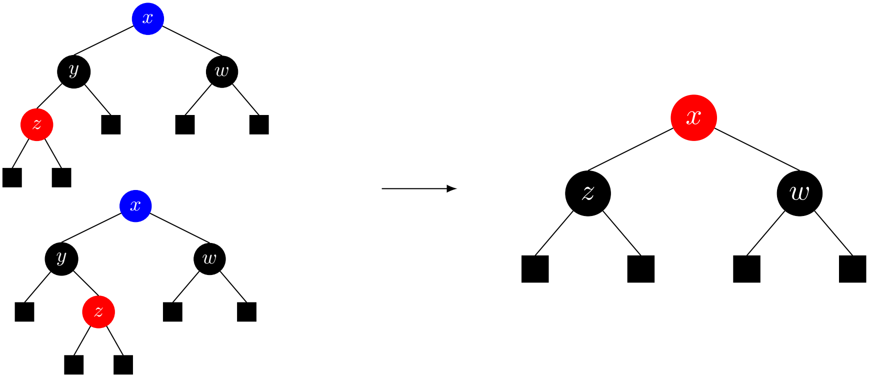
y is the node to delete

w is y's sibling

x is y's parent

z is y's child

Case: y has a ***Red*** Child



where

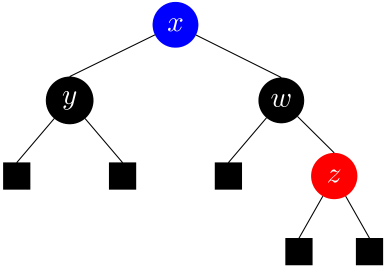
y is the node to delete

w is y's sibling

x is y's parent

z is y's child

Case: w has a ***Red*** Child



where

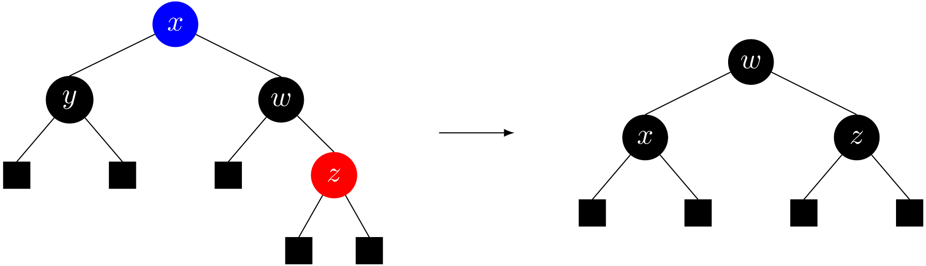
y is the node to delete

z is y's child

w is y's sibling

x is y's parent

Case: w has a ***Red*** Child



where

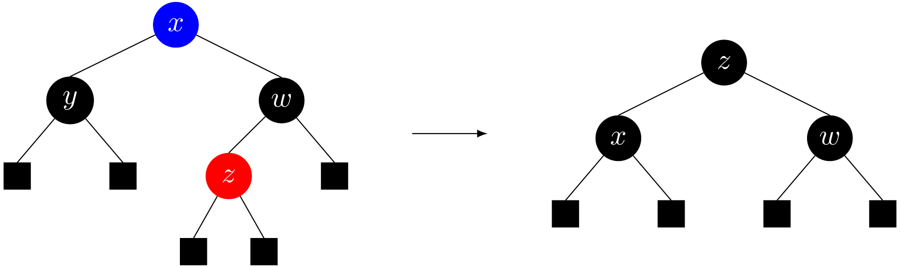
y is the node to delete

z is y's child

w is y's sibling

x is y's parent

Case: w has a ***Red*** Child



where

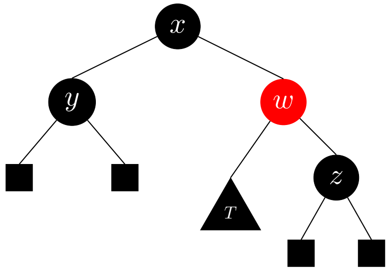
y is the node to delete

z is y's child

w is y's sibling

x is y's parent

Case: w is ***Red***



where

y is the node to delete

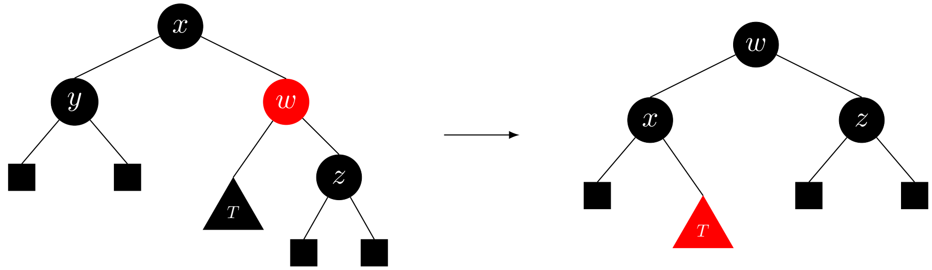
z is y's child

w is y's sibling

x is y's parent

T is w's subtree

Case: w is ***Red***



where

y is the node to delete

z is y's child

w is y's sibling

x is y's parent

T is w's subtree

Red-Black Tree Example