Binary Search Trees

What they are

How they are implemented

Binary Search Trees (BSTs)

Fairly fast search, insertion, deletion

maximum, minimum, successor, predecessor

Creators: Dumey (1952) Wheeler (1957) 

Berners-Lee (1959)  Windley (1960)

Booth & Colin (1960)   Hibbard (1962) 

Complexity (for n data items)

Typically, binary search trees are fairly fast:

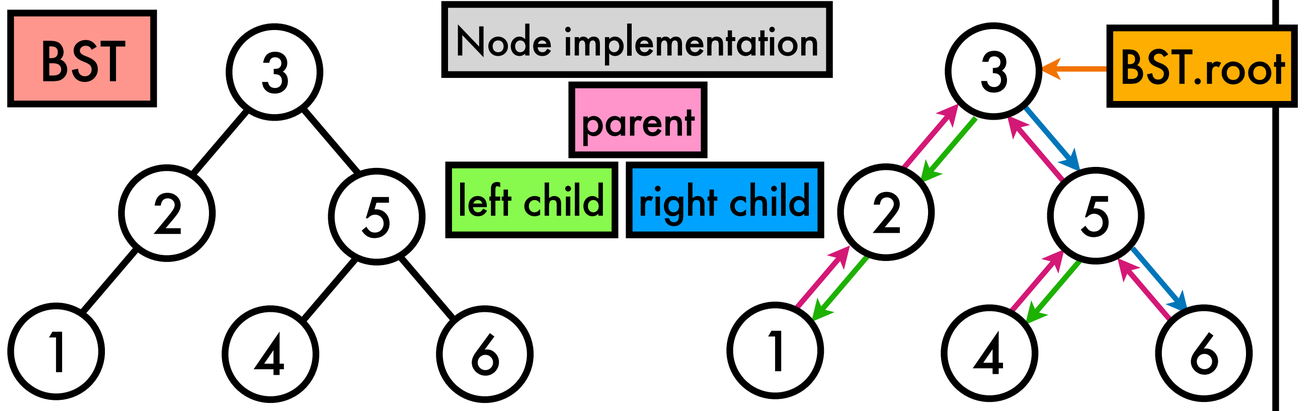
Avg. case: search, insert, delete $\rightarrow O(\log n)$

Worst case: search, insert, delete $\rightarrow \Theta(n)$

search, insert, delete $\rightarrow O(h)$ (h BST height)

Storage of binary search trees: $\Theta(n)$

Suits Abstract Data Types | Set | Map | Priority Queue



Binary Search Tree Property: for each node u, any node l in its left subtree satisfies $l.\text{key} \leq u.\text{key}$, any node r in its right subtree satisfies $r.\text{key} \geq u.\text{key}$

Note: this definition allows duplicate keys

References/Notes/Image credits:
(History of Binary Search Trees) D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 6.2.2 (1974)
(Wheeler/Berners-Lee) A. Douglas, "Techniques for the recording of, and reference to data in a computer", The Computer Journal (1959)
(D. Wheeler) https://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)#/media/File:EDSAC_(14)_(cropped).jpg
(C. Berners-Lee) https://www.bl.uk/voices-of-science/interviewees/conway-berners-lee
A. Booth and A. Colin, "On the efficiency of a new method of dictionary construction", Information and Control (1960)
(Booth) https://www.computerhope.com/people/andrew_booth.htm
(Andrew Colin) https://www.heraldscotland.com/opinion/16998308.obituary--andrew-colin-professor-computer-science/
T. Hibbard, "Some combinatorial properties of certain trees with applications to searching and sorting", JACM (1962)
(Hibbard) http://math.oxford.emory.edu/site/cs171/hibbardDeletion/
(Binary Search Trees) T. Cormen et al., "Introduction to algorithms", Chap 12.1, MIT press, (2022)

Tree traversals

Traversal algorithms

A traversal algorithm aims to "process" each node in the tree exactly once

The simplest traversals use depth-first-search (go deeper first, rather than "breadth first")

inorder preorder postorder

Inorder traversal: process each node in-between visiting left subtree and right subtree

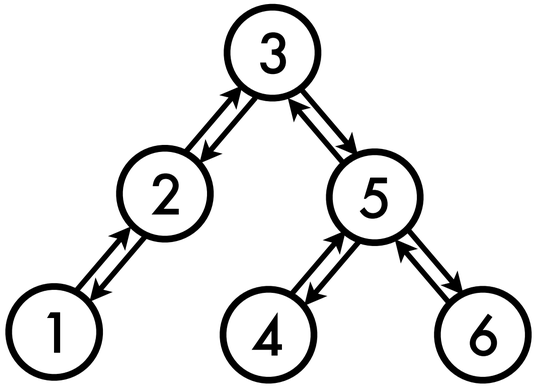
Recursive implementation:

```
def inorder(u):
    if u is not None:
        inorder(u.left)
        print(u.key)
        inorder(u.right)
```

Inorder traversal of BST

print out: 1 2 3 4 5 6

Note that keys were printed in order!




```
def preorder(u):
    if u is not None:
        print(u.key)
        preorder(u.left)
        preorder(u.right)
```

print out: 3 2 1 5 4 6

```
def postorder(u):
    if u is not None:
        postorder(u.left)
        postorder(u.right)
        print(u.key)
```

print out: 1 2 4 6 5 3

Traversals are $\Theta(n)$ - call themselves twice at each node (left child and right child)

References/Notes/Image credits:
(Tree traversals) http://webdocs.cs.ualberta.ca/~holte/T26/tree-traversal.html
(Traversal complexity) T. Cormen et al., "Introduction to algorithms", Chap 12.1, MIT press, (2022)

Binary Search Tree Queries

Minimum/Maximum

```
def minimum(u):
    while u.left:
        u = u.left
    return u
```

E.g. argument: root

returns node with key 1

```
def maximum(u):
    while u.right:
        u = u.right
    return u
```

E.g. argument: root

returns node with key 9

Search

```
def search(u, key):
    while u and key != u.key:
        if key < u.key:
            u = u.left
        else:
            u = u.right
    return u
```

query key: 6

returns node

query key: 7

returns None

Complexity: $O(h)$ where h is tree height

Inorder Predecessor

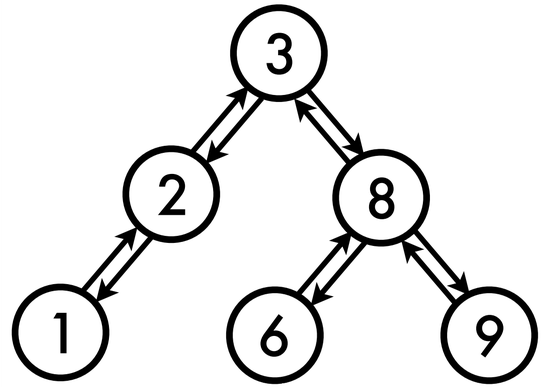
Element immediately before node in the inorder traversal ordering

Note: no key required!

```
def predecessor(u):
    if u.left:
        return maximum(u.left)
    else:
        par = u.parent
        while par and u != par.right:
            u = par
            par = par.parent
        return par
```

Note: inorder successor is symmetric

Complexity: $O(h)$ (h is tree height)



Example Binary Search Tree

Predecessor examples

First case

query node with key 3

returns node with key 2

Second case

query node with key 6

returns node with key 3

References:
(Binary search) https://webdocs.cs.ualberta.ca/~holte/T26/binary-search.html (algorithm pseudocode) https://en.wikipedia.org/wiki/Binary_search_tree#Searching (Querying a BST) T. Cormen et al., "Introduction to algorithms", Chap 12.2, MIT press, (2022)

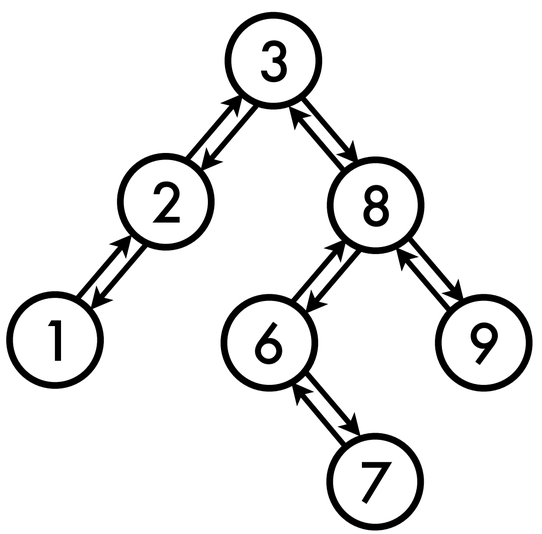
Binary Search Tree Insertion

Insertion

Insert new node `v` into binary search tree `bst`

```
def insert(bst, v):
    u = bst.root
    par = None
    while u:
        par = u
        u = u.left if v.key < u.key else u.right
    v.parent = par
    if not par: # handle case when bst was empty
        bst.root = v
    elif v.key < par.key:
        par.left = v
    else:
        par.right = v
```

Complexity: $O(h)$ where h is tree height



$3 \leq 7 \implies$ u.key $\leq$ v.key

$7 < 8 \implies$ v.key $<$ u.key

$6 \leq 7 \implies$ u.key $\leq$ v.key

Example Binary Search Tree

Insert example node with key 7

We follow Cormen (BST allows duplicate keys)

If not allowed, insert() must be modified

References:

(insertion pseudocode) https://en.wikipedia.org/wiki/Binary_search_tree#Insertion

(Insertion) T. Cormen et al., "Introduction to algorithms", Chap 12.3, MIT press, (2022) (a.k.a CLRS due to the authors names)

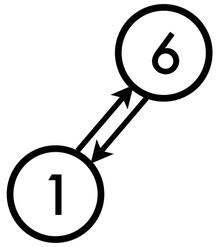
Binary Search Tree Deletion

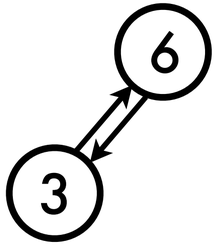
Deletion logic

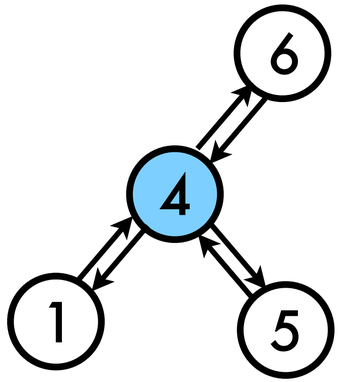
To delete a node from BST, there are four cases to consider

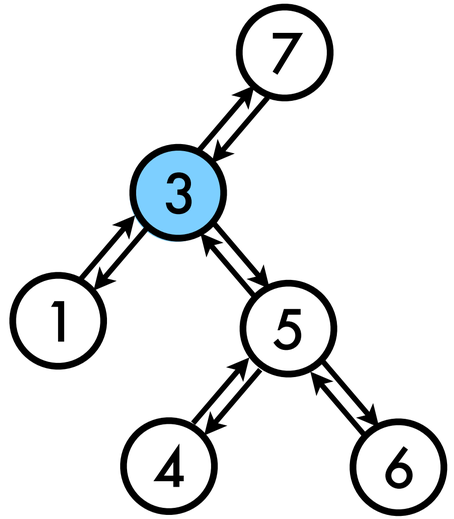
goal: preserve the BST property

In the following, we will focus on deleting the node with key 2









case 1:
no right child

case 2:
no left child

case 3: two children
successor is right child

case 4: two children
successor is **not** right child

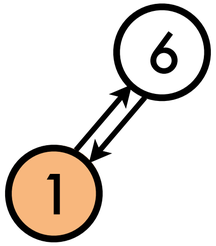
References:
https://en.wikipedia.org/wiki/Binary_search_tree#Deletion
(Deletion) T. Cormen et al., "Introduction to algorithms", Chap 12.3, MIT press, (2022)

Binary Search Tree Deletion

Deletion implementation

Delete node `u` from binary search tree `bst`

```
def delete(bst, u):
    if not u.right:
        shift_nodes(bst, u, u.left)
    elif not u.left:
        shift_nodes(bst, u, u.right)
    else:                               # u has two children
        u_successor = minimum(u.right)
        if u_successor != u.right:
            shift_nodes(bst, u_successor, u_successor.right)
            u_successor.right = u.right
            u_successor.right.parent = u_successor
        shift_nodes(bst, u, u_successor)
        u_successor.left = u.left
        u_successor.left.parent = u_successor
```




```
def shift_nodes(bst, old, src):
    if not old.parent:
        bst.root = src
    elif old == old.parent.left:
        old.parent.left = src
    else:
        old.parent.right = src
    if src:
        src.parent = old.parent
```

Complexity: $O(h)$ where h is tree height

References:
(Deletion pseudocode - we follow this naming convention) https://en.wikipedia.org/wiki/Binary_search_tree#Deletion
(Deletion) T. Cormen et al., "Introduction to algorithms", Chap 12.3, MIT press, (2022)

Red-Black Trees

What they are

How they are implemented

Red-Black Trees (RBTs)

Self-balancing binary search trees

Assuredly fast search, insertion, deletion

History

Adelson-Velsky & Landis (AVL tree) (1962) 

Bayer (Symmetric binary B-trees) (1972) 

Guibas & Sedgewick (RB trees) (1978) 

Sedgewick (Left-leaning RB trees) (2008) 

Complexity (for n data items)

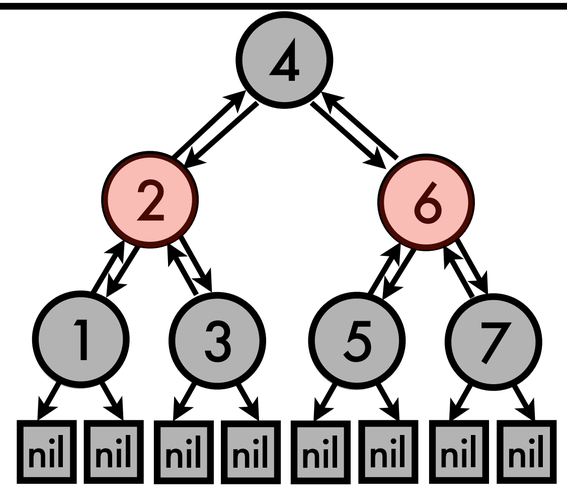
By balancing, red-black trees guarantee speed

Worst case: search, insert, delete $\rightarrow O(\log n)$

Storage of red-black trees: $\Theta(n)$

Suits Abstract Data Types | Set | Map

Applications: Container libraries (C++, Java) | Linux CFS



If node has no child, points to a special `nil` node

- treated as black
- simplify logic
- omitted from diagrams

Red-black tree: a Binary Search Tree where each node also has a colour (which can be **red** or **black**)

Approximately balanced: Tree height is $\Theta(\log n)$

References/Notes/Image credits:
https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
(AVL Trees) G. M. Adelson-Velsky and E. M. Landis, "An algorithm for the organization of information", Doklady Akademii Nauk (1962)
(G. M. Adelson-Velsky) https://www.math.toronto.edu/askold/2014-UMN-4-e-Adelson-.pdf
(E. Landis) https://opc.mfo.de/detail?photo_id=2447
R. Bayer, "Symmetric binary B-trees: Data structure and maintenance algorithms", Acta informatica (1972)
(R. Bayer) https://www.computerhope.com/people/rudolf_bayer.htm
L. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees", SFCS (1978)
(L. Guibas) https://geometry.stanford.edu/member/guibas/
(R. Sedgewick) https://sedgewick.io/
(Red-Black Trees) T. Cormen et al., "Introduction to algorithms", Chap 13.1, MIT press, (2022)
(Linux CFS) https://en.wikipedia.org/wiki/Completely_Fair_Scheduler

Red-Black Tree Properties

Five key properties of Red-Black Trees (CLRS)

RBTs are Binary Search Trees with:

Property 1 Every node is **red** or **black**

Property 2 The root node is **black**

Property 3 Every leaf node (nil node) is **black**

Property 4 If a node is **red**, both of its children are **black**

Property 5 Starting from any node, all simple paths down to leaf nodes hold the same number of **black** nodes

Tree height (h) is $\Theta(\log n)$

By **Property 5** all root-leaf paths have the same number of **black** nodes

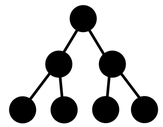
By **Property 4** **red** nodes have **black** children

$\Longrightarrow$ longest path, $h_{\max} \leq 2 \cdot$ shortest path, $h_{\min}$

Note: perfect binary tree has $n = 2^{h+1} - 1$ nodes

- $h = 0$, $n = 1$

$h = 1$, $n = 3$



$h = 2$, $n = 7$

$$\Longrightarrow 2^{h_{\min}+1} - 1 \leq n \leq 2^{h_{\max}+1} - 1$$

$$\Longrightarrow h_{\min} \leq \log(n+1) - 1 \leq h_{\max} \leq 2 \cdot h_{\min}$$

$$\Longrightarrow h_{\min} = \Theta(\log n)$$

Key "balanced tree" result

$$h_{\min} \leq h_{\max} \leq 2 \cdot h_{\min} \Longrightarrow h_{\max} = \Theta(\log n)$$

$\Longrightarrow$ Min | Max | Search | Successor | Predecessor | $O(\log n)$

References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap 13.1, MIT press, (2022)
(Proof of tree height) L. Arge and M. Lagoudakis, CPS 230 lecture notes, https://courses.cs.duke.edu/cps130/fall02/fall02lectures/lecture18/long_redblack.pdf (2002)

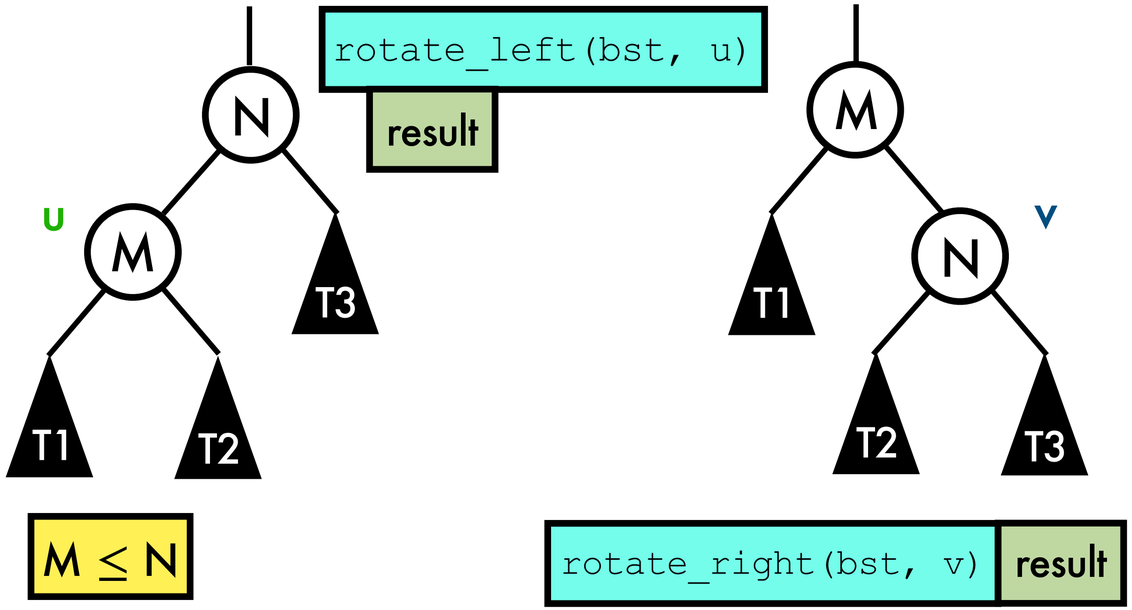
Rotation Operations

The rotation operation

Rotations restructure a tree locally without

breaking the Binary Search Tree Property

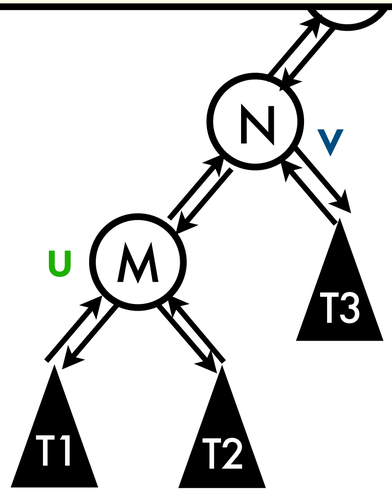
Rotations change links (not colours) at $O(1)$ cost



References:
(Rotations) L. Arge and M. Lagoudakis, CPS 230 lecture notes, https://courses.cs.duke.edu/cps130/fall02/fall02lectures/lecture18/long_redblack.pdf (2002)
(rotation logic/comments based on CLRS) T. Cormen et al., "Introduction to algorithms", Chap 13.2, MIT press (2022)
Python code snippet reference for rotate_left - https://blog.boot.dev/python/red-black-tree-python/

Rotate_left implementation in Python

```
def rotate_left(self, u): # self is BST instance
    v = u.right
    u.right = v.left
    if v.left != self.nil:
        v.left.parent = u
    v.parent = u.parent
    if u.parent is None:
        self.root = v
    elif u.parent.left == u:
        u.parent.left = v
    else:
        u.parent.right = v
    v.left, u.parent = u, v
```



Why Are Rotations Useful?

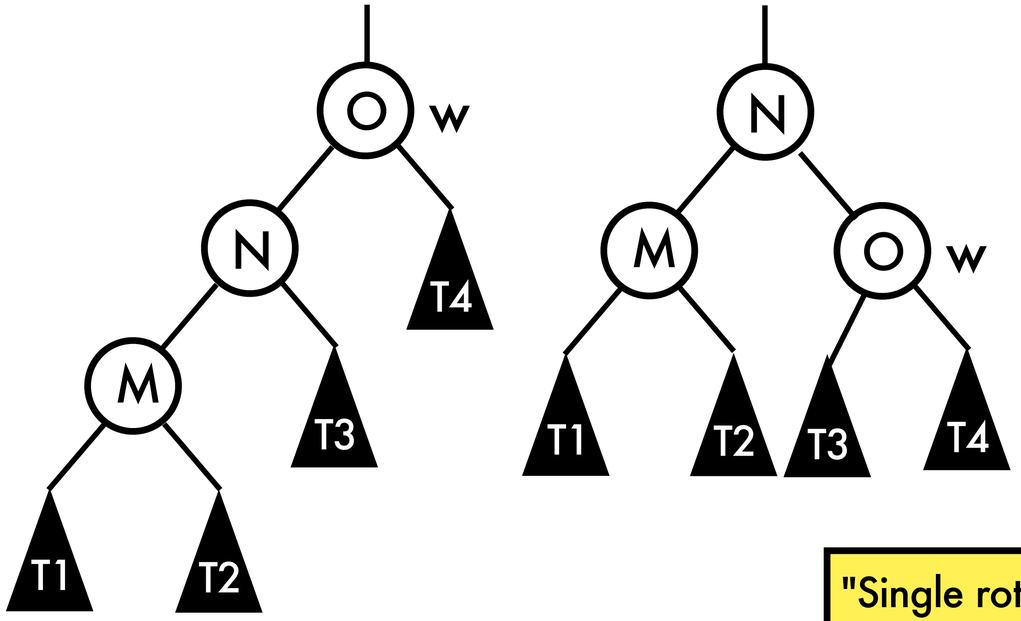
The benefits of rotation

When too many nodes are on a single path,

rotation distributes them to neighbour paths

This restores balance in the tree

It preserves the Binary Search Tree Property

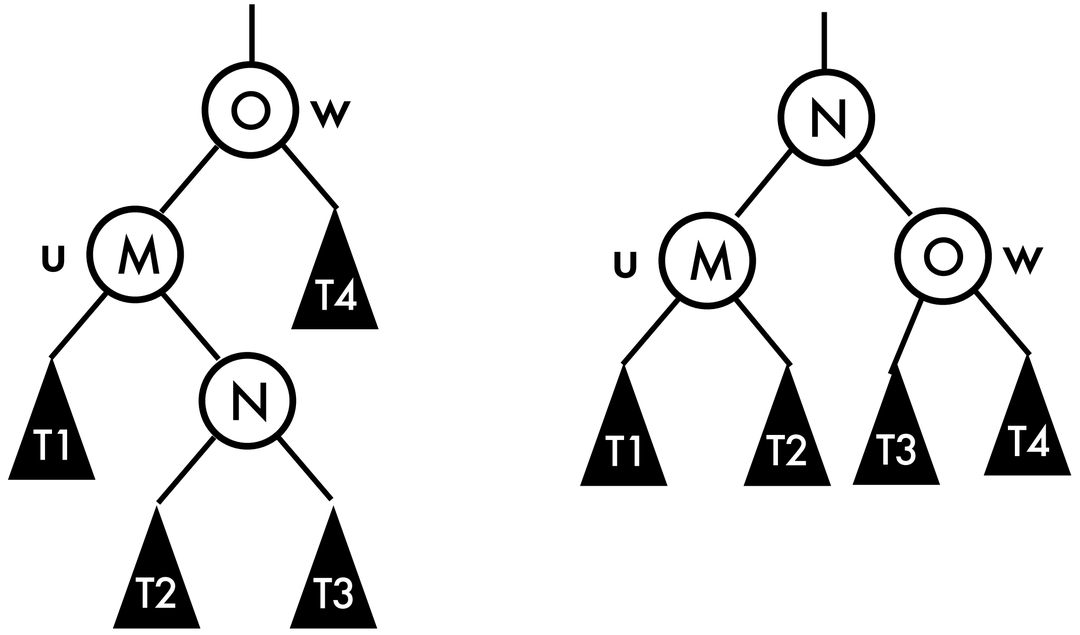


"Single rotation"

`rotate_right(bst, w)` — Produces more balanced structure

Double rotations

If we have a chain with both a left child and a right child, then a "double rotation" is needed



Function calls: `rotate_left(bst, u)` `rotate_right(bst, w)`

References:
(Rotations) L. Arge and M. Lagoudakis, CPS 230 lecture notes, https://courses.cs.duke.edu/cps130/fall02/fall02lectures/lecture18/long_redblack.pdf (2002)

Red-Black Tree Insertion

Insertion overview

Initially, RBT insertion is the same as BST insertion

What colour should the inserted node be?

We have two options: **red** or **black**

If we colour the inserted node **red** and its parent is **red**, we violate Property 4

If we colour the inserted node **black**, the extra black node will violate Property 5

Strategy: fix problem locally with rotations and recolouring

"Escalate" further issues up the tree

"Management strategy"

Tree height is $\Theta(\log n)$ so fixing is $O(\log n)$

Total Red-Black Tree insertion $O(\log n)$

Insertion - handling cases

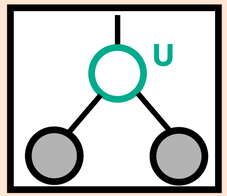
Red-black tree insertion involves patiently handling several cases

Precursor: whenever we have a RBT violation,

the node has **black** children

True after inserts (nils are **black**)

Also true further up the tree



Case 1 u has a black parent

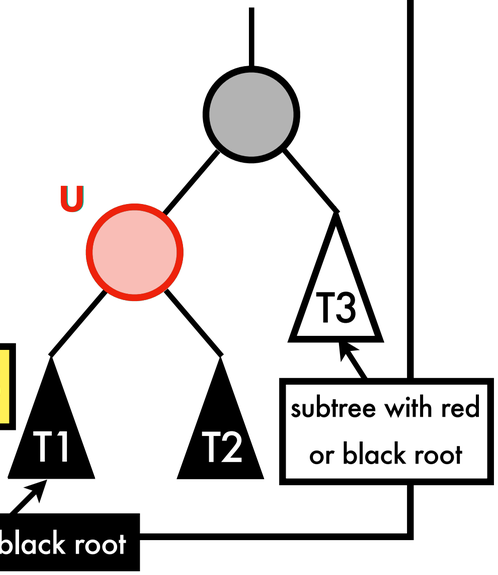
Simply colour node u **red**:

u is not the root Property 2

Only **black** children Property 4

No extra **black** nodes Property 5

Problem is fixed locally



Reference:
(Rotations) L. Arge and M. Lagoudakis, CPS 230 lecture notes, https://courses.cs.duke.edu/cps130/fall02/fall02lectures/lecture18/long_redblack.pdf (2002)

Red-Black Tree Insertion Cont.

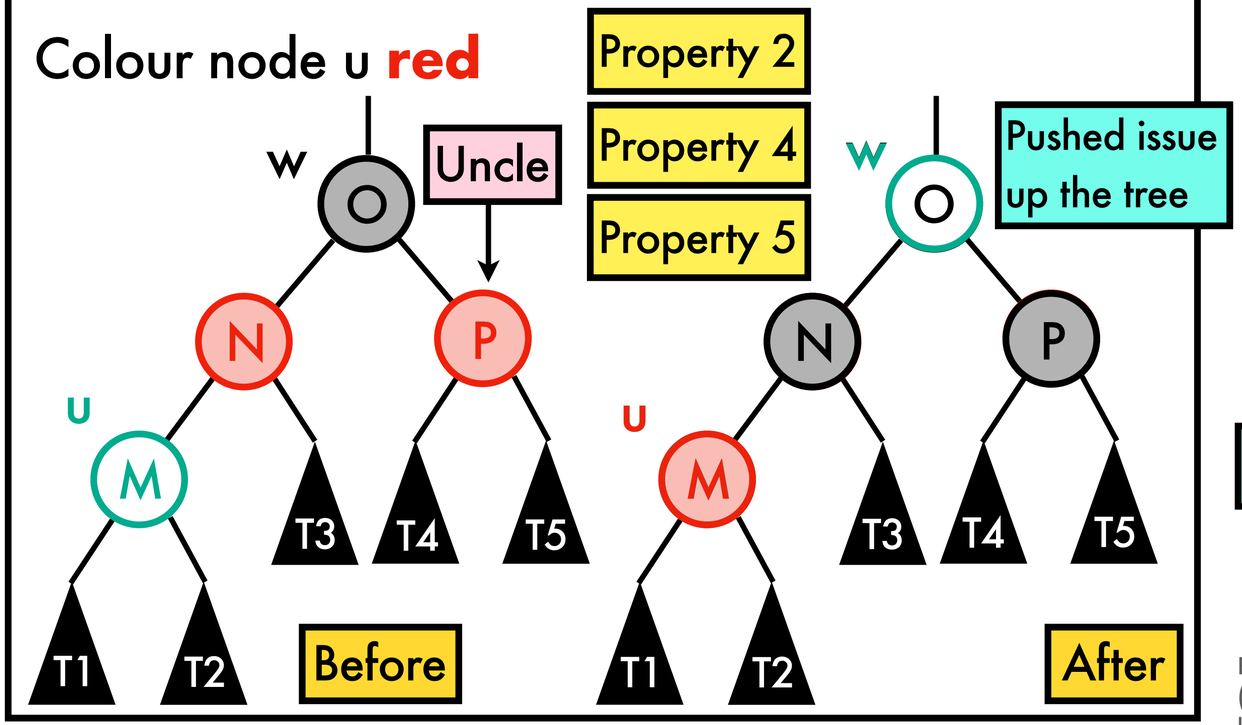
Parent of u is red

When parent of (u) is **red**, the steps depend on colour of its parent's sibling (its "uncle")

Case 2 (u) has **red** parent, **red** uncle

Push **black** colour of w down to each child

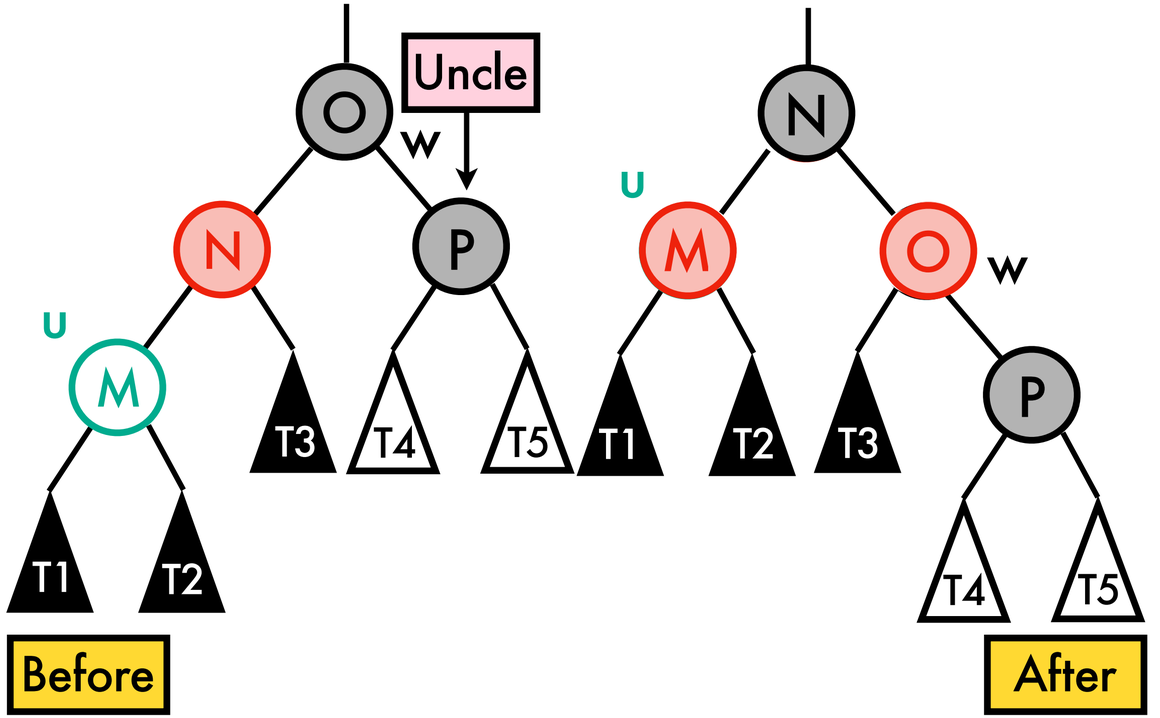
Colour node u **red**



Parent of u is red

Case 3 (u) has **red** parent, **black** uncle

Case 3a (u) is away from its uncle



`rotate_right(bst, w)` | u → red | u.parent → black | w → red

Problem is fully fixed (no further escalation)

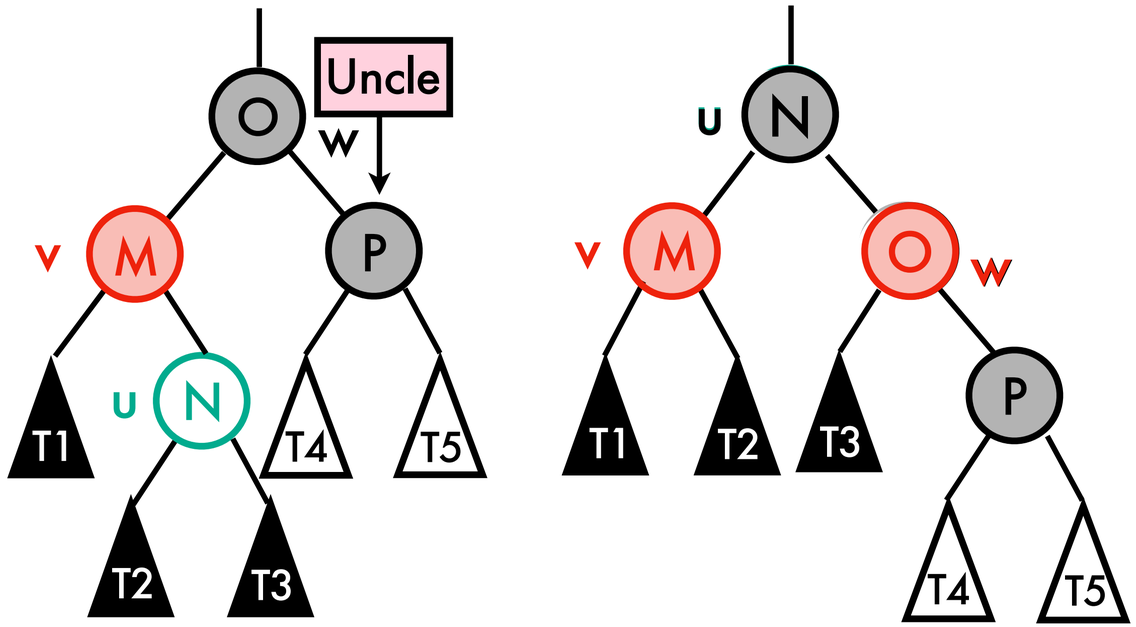
Reference:
(Rotations) L. Arge and M. Lagoudakis, CPS 230 lecture notes, https://courses.cs.duke.edu/cps130/fall02/fall02lectures/lecture18/long_redblack.pdf (2002)

Red-Black Tree Insertion Cont.

Parent of u is red

Case 3 (u) has **red** parent, **black** uncle

Case 3b (u) is towards its uncle



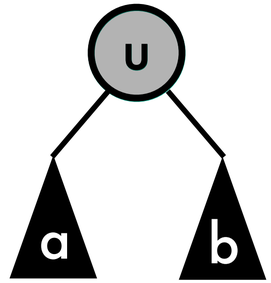
Before `rotate_left(bst, v)` After

`rotate_right(bst, w)` u → black w →red

Problem is fully fixed (no further escalation)

u has no parent

Case 4 (u) has no parent



Colour node u **black**

Property 2 Property 5

Problem is fully fixed (no further escalation)

Red-Black Tree Deletion

Widely feared

Deletion overview

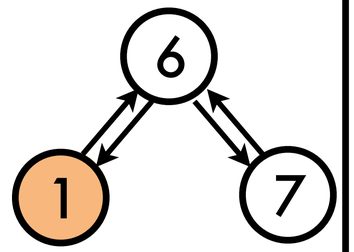
We build on top of Binary Search Tree deletion

Red-Black Tree deletion is $O(\log n)$

It is also quite complicated to implement

Shifting subtrees

```
def shift_nodes(self, old, new):
    if not old.parent:
        self.root = new
    elif old == old.parent.left:
        old.parent.left = new
    else:
        old.parent.right = new
    new.parent = old.parent
```



Reference:
T. Cormen et al., "Introduction to algorithms", Chap 13.4, MIT press (2022)

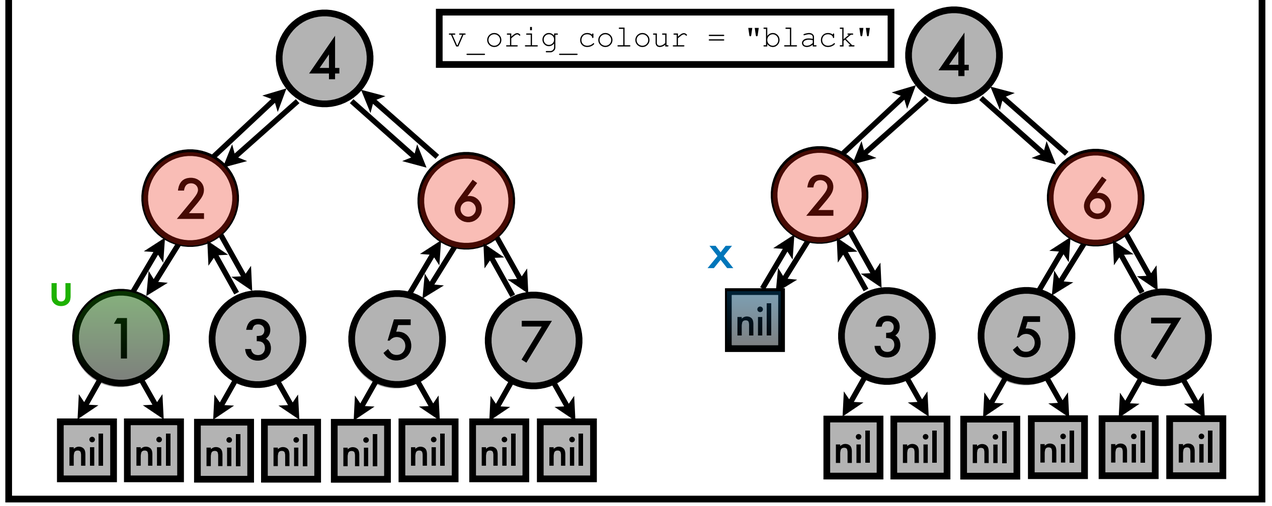
Red-Black Tree Deletion

```
def delete(self, u):  # self is an instance of a red-black tree
    v = u  # assign v to the node to be deleted
    v_orig_colour = v.colour  # track v's original colour
    if u.left == self.nil:  # u's left child is nil
        x = u.left
        shift_nodes(self, u, x)  # shift up x into u's place
    elif u.right == self.nil:  # u's right child is nil
        x = u.right
        shift_nodes(self, u, x)  # shift up x into u's place
    else:  # u has two children
        v = minimum(u.right)
        v_orig_colour = v.colour
        x = v.right
        if v != u.right:
            shift_nodes(self, v, v.right)
            v.right = u.right
            v.right.parent = v
        else:
            x.parent = v
        shift_nodes(self, u, v)
        v.left = u.left
        v.left.parent = v
        v.colour = u.colour
if v_orig_colour == "black":
    fix_delete_violations(self, x)
```

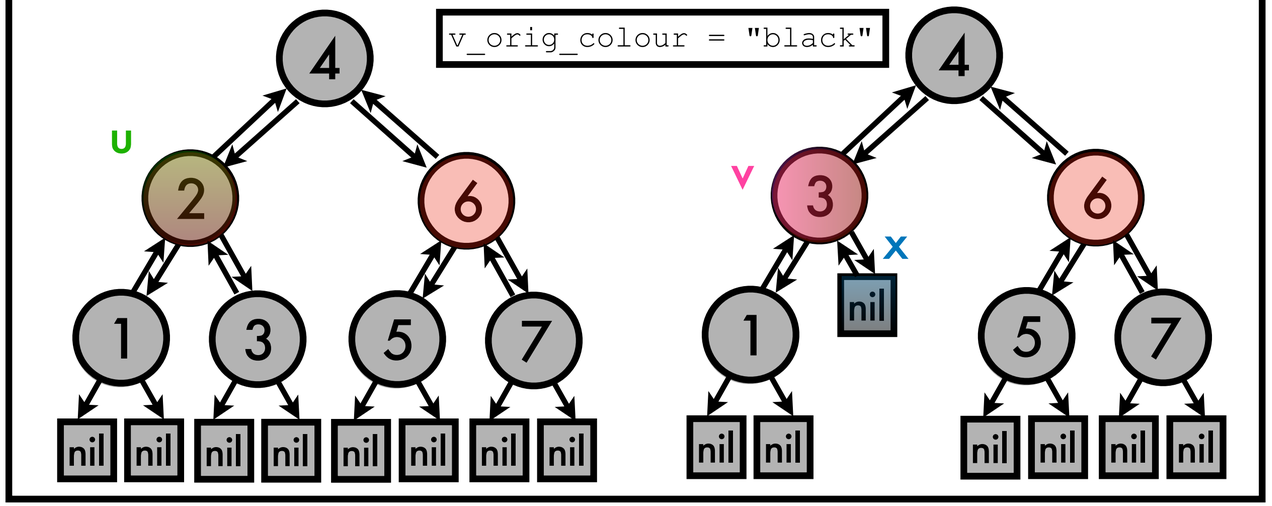
If `v_orig_color` was red:

- v wasn't the root — Property 2
- no red nodes have become neighbours — Property 4
- num black nodes on paths haven't changed — Property 5

u's left child is nil (right case is symmetric)



u has two children

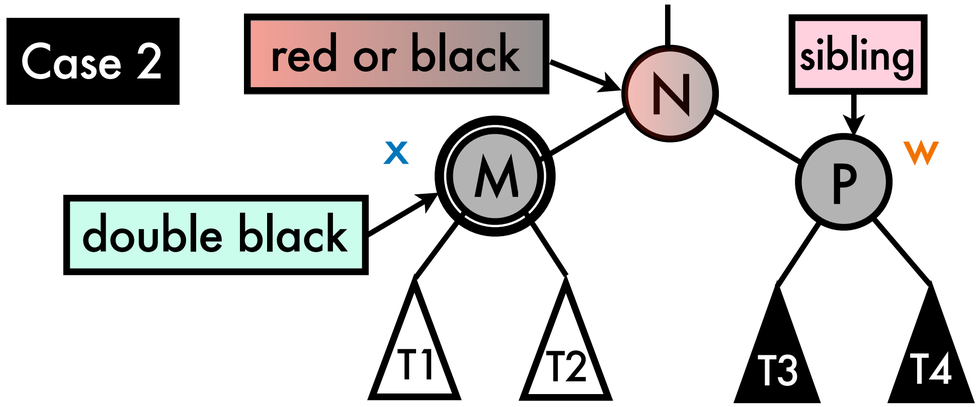


Reference:
Code adapted from T. Cormen et al., "Introduction to algorithms", Chap 13.4, MIT press (2022)

Fixing RBT Violations

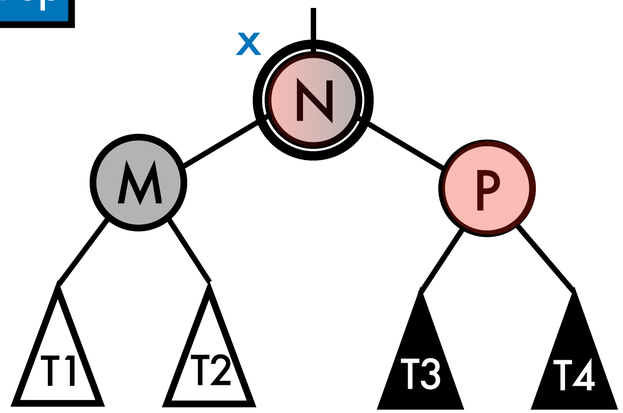
w (x's sibling) is black, w's children are black

Case 2



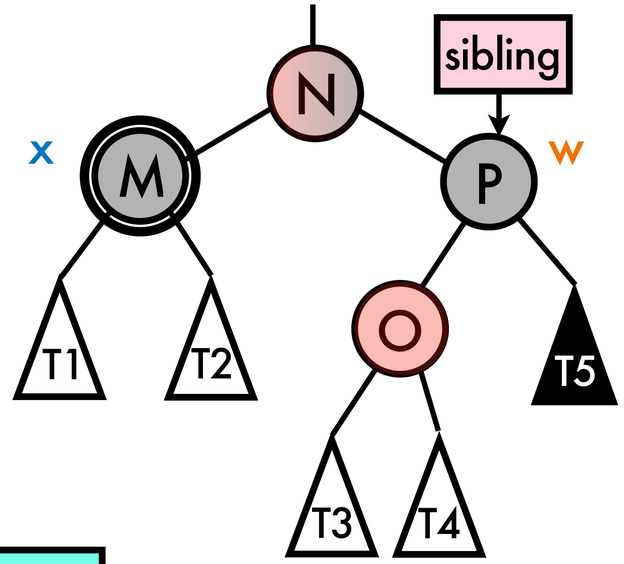
w → red

Move x up



w is black with red left and black right child

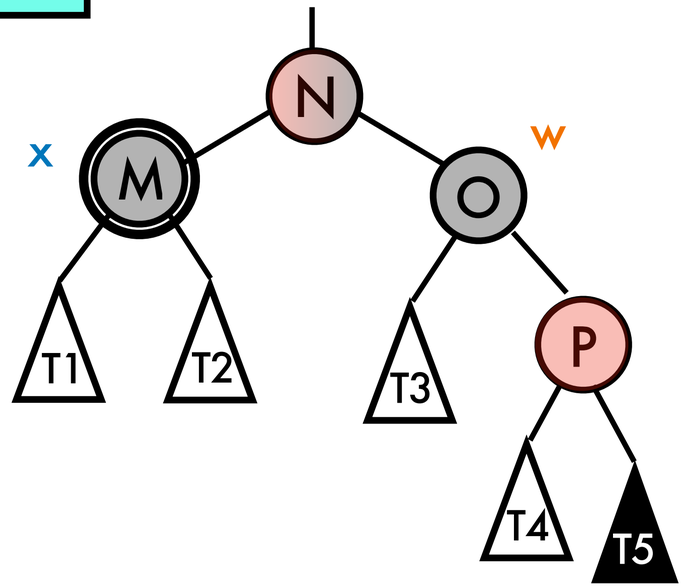
Case 3



`rotate_right(bst, w)`

w → red

w.parent → black

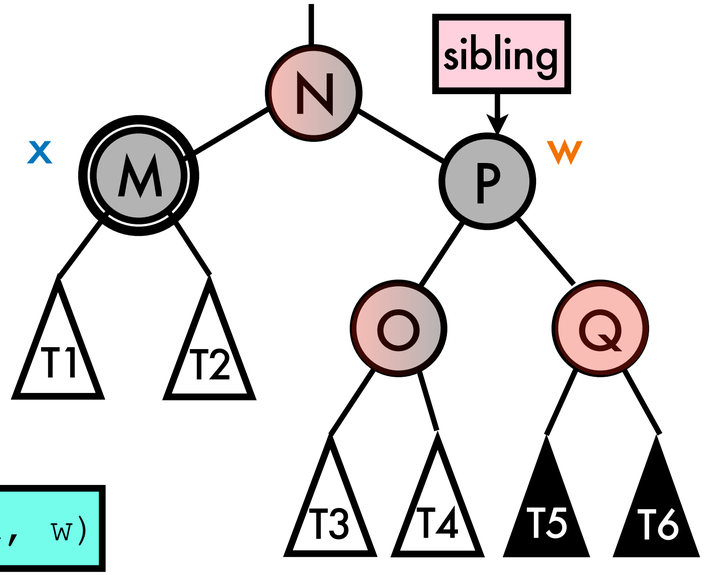


Reference:
T. Cormen et al., "Introduction to algorithms", Chap 13.4, MIT press (2022)

Fixing RBT Violations

w is black and w's right child is red

Case 4



`rotate_left(bst, w)`

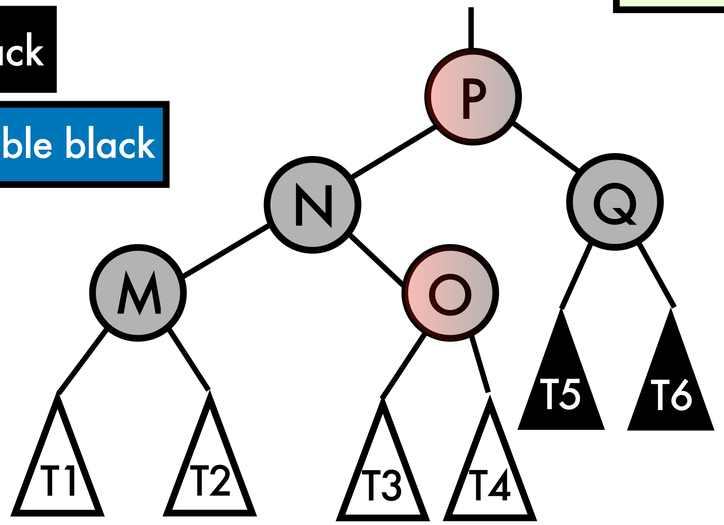
w.right → black

w → w.left's colour

x = root

w.left → black

Remove double black



Case flow

Case 1 → Case 2 → resolves (x red-and-black)

Case 1 → Case 3

Case 1 → Case 4

Case 2 → Case 2 → loops at most $O(\log n)$ times

Case 3 → Case 4

Case 4 → resolves (x set to root)

All paths through case handling are $O(\log n)$

$\Longrightarrow$ `fix_delete_violations()` is $O(\log n)$

$\Longrightarrow$ Red-Black Tree deletion is $O(\log n)$

Reference:
T. Cormen et al., "Introduction to algorithms", Chap 13.4, MIT press (2022)