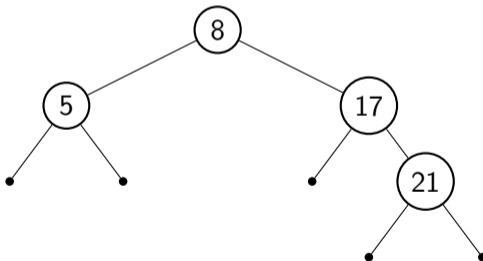Topic 7.2

Red-black tree

Null leaves

To describe a red-black tree, we replace the null pointers for absent children with dummy null nodes.

Example 7.2

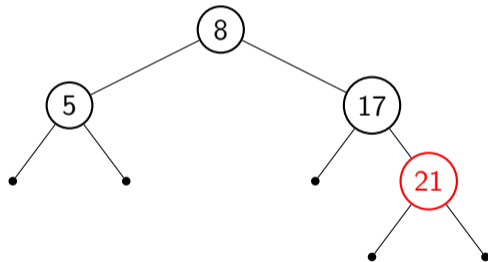The following tiny nodes are the dummy null nodes.

Red-black tree

Definition 7.1

A red-black tree T is a binary search tree such that the following holds.

- ▶ All internal nodes are colored either **red** or **black**
- ▶ Null leaves have no color
- ▶ Root is colored black
- ▶ No **red** node has a **red** child
- ▶ All paths from the root to null leaves have the same number of **black** nodes.
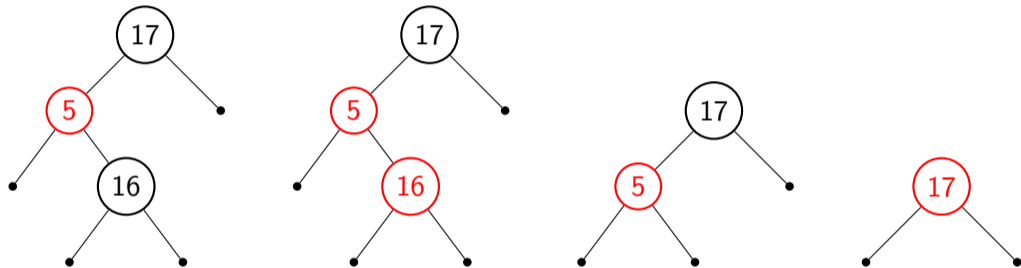
Example 7.3

An example of a red-black tree.

Exercise: Identify red-black trees

Exercise 7.1

Which of the following are red-black trees?

Observations:

▶ Red nodes are not counted in the imbalance. We need them only when there is an imbalance.

▶ There cannot be too many red nodes. (Why?)

▶ Red nodes can be at every level except the root.

Black height

Definition 7.2
The black height (bh) for each node is defined as follows.

$$bh(n) = \begin{cases} 0 & \text{n is a null leaf} \\ max(bh(right(n)), bh(left(n))) + 1 & \text{n is a \textbf{black} node} \\ max(bh(right(n)), bh(left(n))) & \text{n is a \textcolor{red}{red} node} \end{cases}$$

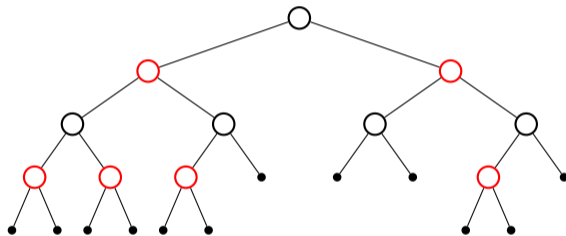Until it is stated otherwise, all heights mentioned in this lecture are black heights.

Exercise 7.2
Prove that for each node n in a red-black tree, $bh(right(n)) == bh(left(n))$.

Example: black height

Example 7.4

The black height of the following red-black tree is 2.

Exercise 7.3

Can we change the color of some nodes without breaking the conditions of a red-black tree?

Bound on the height of a red-black tree

Let h be the black height of a red-black tree containing n nodes.

- n is the smallest when all nodes are **black**. Therefore, the tree is a complete binary tree. Therefore, $n = 2^h - 1$.

- n is largest when the alternate levels of the tree are red. The height of the tree is $2h$. Therefore, $n = 2^{2h} - 1$.

$$\log_4 n < h \le \log_2(n+1)$$

Exercise 7.4
Define red-black tree inductively and write the proof of the above bounds using induction.

Commentary: Our definition of red-black tree is not inductive therefore the above proof does not read so formal. However, an inductive definition would make proof formal. Can you write an inductive definition of red-black tree and prove the above statements?

CS213/293 Data Structure and Algorithms 2025   Instructor: Ashutosh Gupta   IITB India   13

Search, Maximum/Minimum, and Successor/Predecessor

We can search, maximum/minimum, and successor/predecessor on the red-black tree as usual.

Their running time will be $O(\log n)$ because $h < 1 + \log_2 n$.

How do we do insertion and deletion on a red-black tree?

Topic 7.3

Insertion in red-black tree*

BST insertion in red-black tree

1. Follow the usual algorithm of insertion in the BST, which inserts the new node n as a leaf.
   - ▶ Note that there are dummy nodes. n is inserted as the parent of a dummy node.

2. We color n red.

▶ Good news: No change in the black height of the tree.
▶ Bad news: n may have a *red* parent.

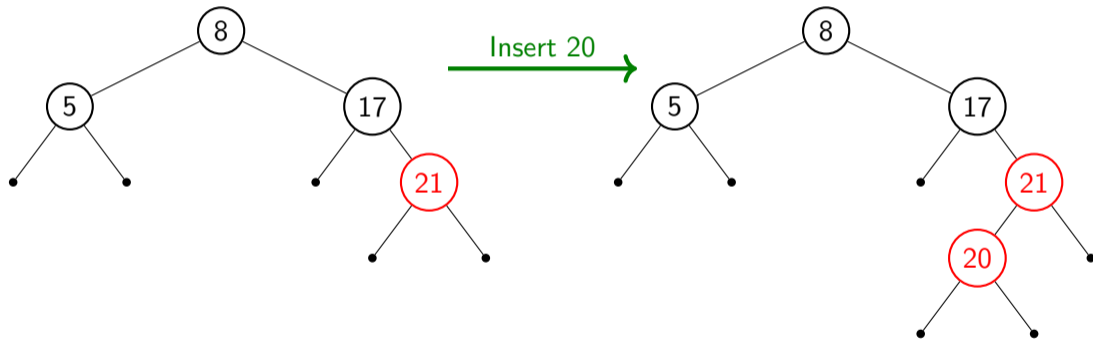After insertion, we may have a **red**-**red** violation, where a **red** node has a **red** child.

Commentary: We have null nodes in this setting. We need to add nulls as children to the new node.

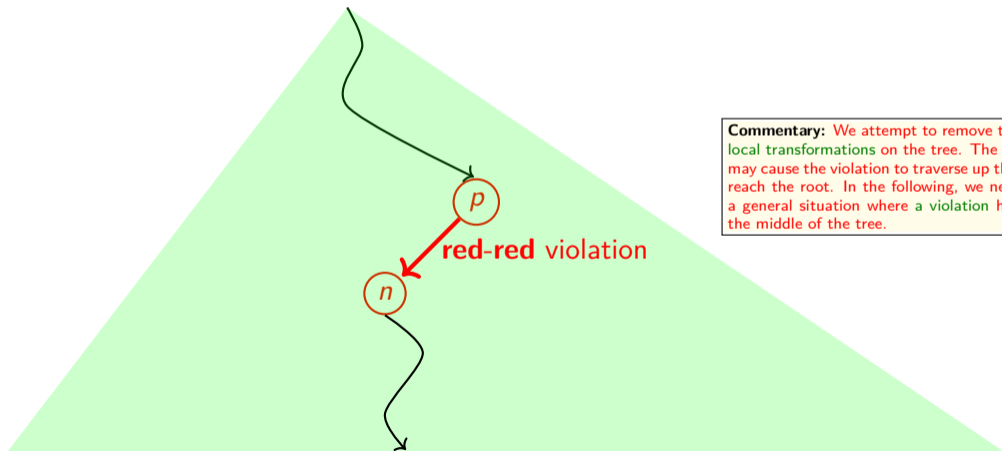# Example: insert in red-black tree

Example 7.5

Inserting 20 in the following tree.

The insertion results in violation of the condition of the red-black tree, which says red nodes can only have black children.

Iteratively remove **red**-**red** violation

A **red**-**red** violation starts at a leaf. However, an attempt to remove the violation may push up the violation to the parent.
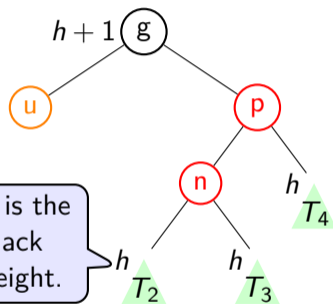

Commentary: We attempt to remove the violation by local transformations on the tree. The transformation may cause the violation to traverse up the tree until we reach the root. In the following, we need to consider a general situation where a violation has occurred in the middle of the tree.

Pattern around red-red violation

If n has a **red** parent, we correct the error by rotation and re-coloring operations.

h+1 (g)

(u)   (p)

(n) h
      T_4

h is the black height.

h h
T_2 T_3

Orange means that we need to consider all possible colors of the nodes.

We have three cases.

▶ Case 1: u is **red**

"not **red**" means either **black** node or null node.

▶ Case 2: u is not **red** and the g to n path is not straight
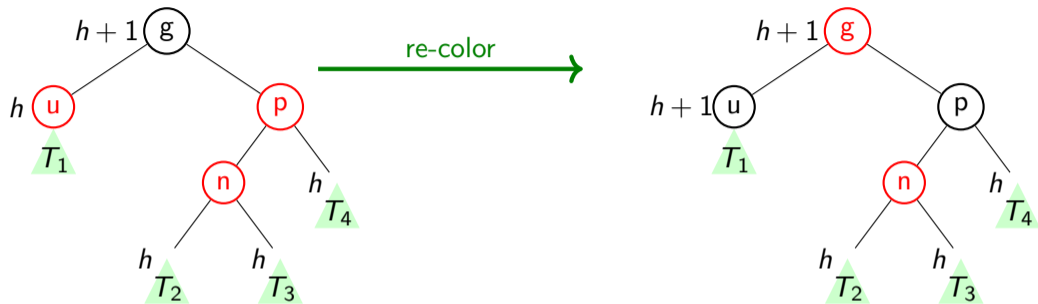
▶ Case 3: u is not **red** and the g to n path is straight

Exercise 7.5
Why must g exist and be black?

No transformation should change the black height of g.

Case 1: The uncle is red

In the subtree of g, there is no change in the black height and no **red-red** violation.

Now g is red. We have three possibilities: the parent of g is **black**, the parent of g is **red**, and g is the root.
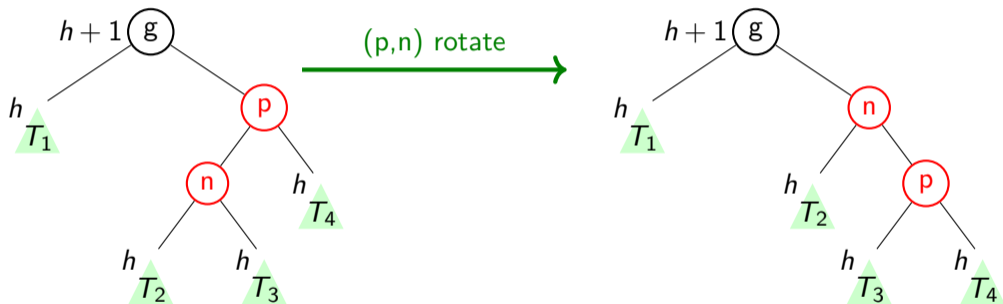
Exercise 7.6
What do we do in each of the possibilities?

Commentary: Possibility 1: Nothing. Possibility 2: We have a red-red violation a level up and need to apply the transformations there. Possibility 3: turn g back to black.
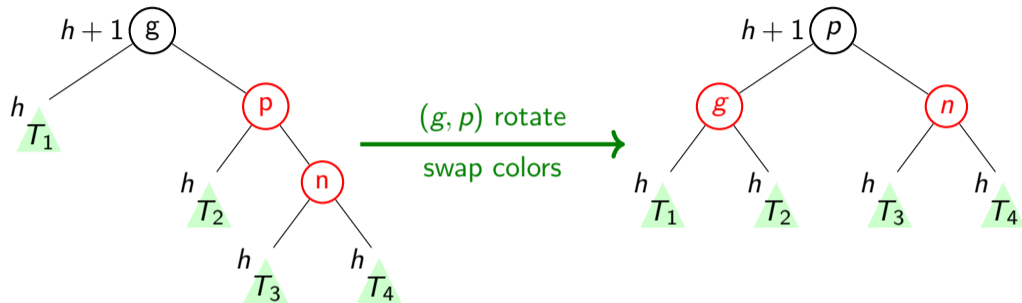
Case 2: The uncle is not red and the path to the grandparent is not straight

straight means $left^2(parent^2(n)) = n$ or $right^2(parent^2(n)) = n$

This transformation does not resolve the violation but converts the violation to case 3.

Case 3: The uncle is not red and the path to the grandparent is straight
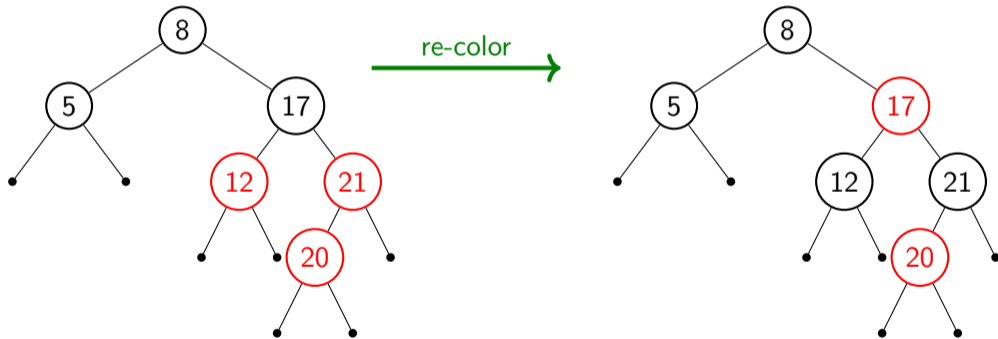

The transformation removes the red-red violation.

Exercise 7.7

a. Why are the roots of T_2, T_3, and T_4 not **red**?

b. Show that if the root of T_1 is **red** then the above operation does not work.

Example: red-red correction case 1
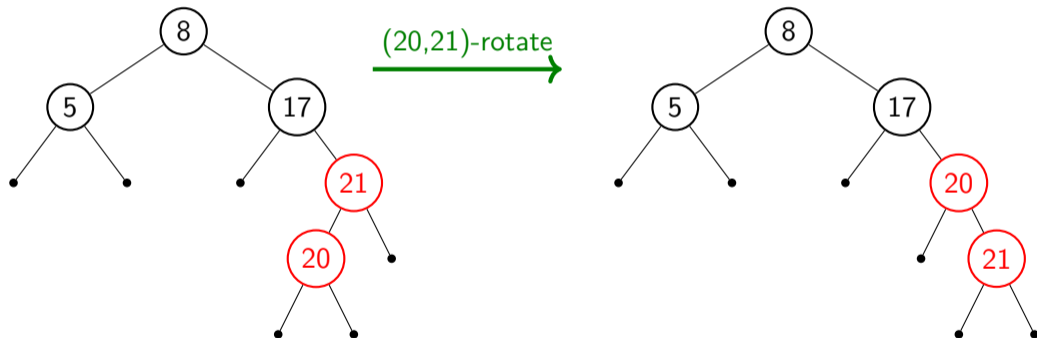
Example 7.6

We just inserted 20 in the following tree. We need to apply case 1 to obtain a red-black tree.

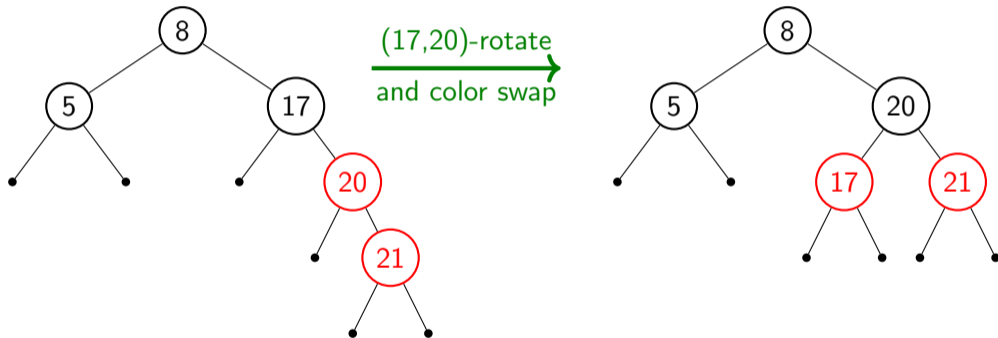re-color

Example: red-red correction case 2

Example 7.7

Consider the following example. We are attempting to insert 20. We apply case 2 to move towards a red-black tree.

$(20,21)$-rotate

The above is not a red-black tree. Furthermore, we need to apply case 3 to obtain the red-black tree.

Example: red-red correction case 3 (continued)

We apply case three as follows.

(17,20)-rotate
and color swap

Summary of insertion

Insert like BST, make the new node **red**, and run the following algorithm.

Algorithm 7.1: $\textsc{RedRedRepair}$(Node n)

---

1 ASSUME(there is no red-red violation at any other node);
2 **if** *n is root* **then** n.color := black; **return**;
3 p := parent(n);
4 **if** *p is black* **then return**;
5 g := parent(p); u := sibling(p);
6 **if** *u.color is* **red** **then** g.color:=**red**; u.color:=p.color:=**black**; $\textsc{RedRedRepair}$(g); **return**; // Case 1 ;
7 **if** *$left^2(g) \neq n$ and $right^2(g) \neq n$* **then** Rotate(p,n);$\textsc{RedRedRepair}$(p);**return**;   //Case 2->3 ;
8 Rotate(g,p); g.color,p.color := p.color,g.color;   //Case 3 ;
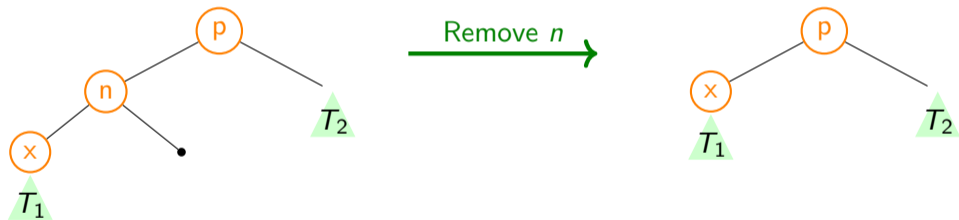

Topic 7.4

Deletion in red-black tree**

BST deletion in red-black tree

▶ Delete a node as if it is a binary search tree.

▶ Recall: In the BST deletion we always delete a node n with at most one non-null child.

x can be either a null or non-null node.

What can go wrong with a red-black tree?

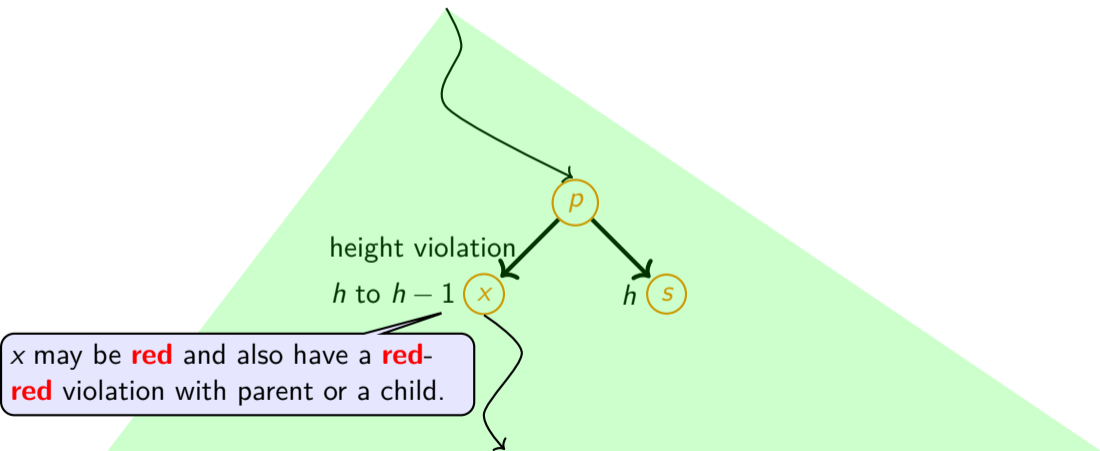Since a child x of n takes the role of n, we need to check if x can replace n.

- If n was **red**, no violations occur. (Why?)
- If n was **black**, $\underbrace{bh(x) = bh(n) - 1}_{\text{black height violation}}$, or it is possible that $\underbrace{\text{both } x \text{ and } p \text{ are } \textbf{red}}_{\text{red-red violation}}$.

> The leaves of the subtree rooted at x will have one less black depth.

Commentary: The or in the second bullet point is not xor. Both violations are possible at the same time.

Violation removal procedure

To remove the violation at x, we may recolor and rotate around x, which may push the violation to the parent. Therefore, we assume that the violation is somewhere in the middle of the tree.
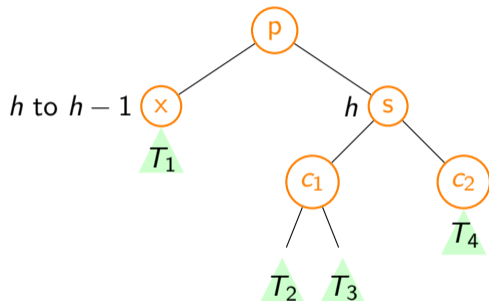

height violation

h to $h-1$ x

h s

x may be **red** and also have a **red-red** violation with parent or a child.

Orange means that we need to consider all possible colors of the nodes.

Violation pattern

After the deletion, we may need to consider the following five nodes around x.

h to $h-1$

h

Exercise 7.8
Show if x is not root and not **red**,
s is non-null.

We correct the violation either by rotation or re-coloring.

There are six cases

1. x is **red**
2. x is not **red** and root.
3. x is not **red** and s is **red**
4. x is not **red** and s is **black**
   - 4.1 c_2 is **red**
   - 4.2 c_2 is not **red** and c_1 is **red**
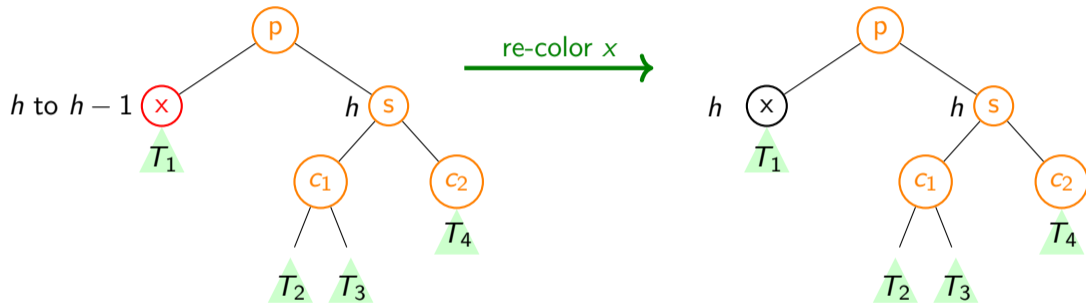   - 4.3 c_2 is not **red** and c_1 is not **red**

The goal is to restore the black height of p.
Trick: find a **red** node and turn it **black**.

Case 1: x is **red**

re-color x

h to $h-1$
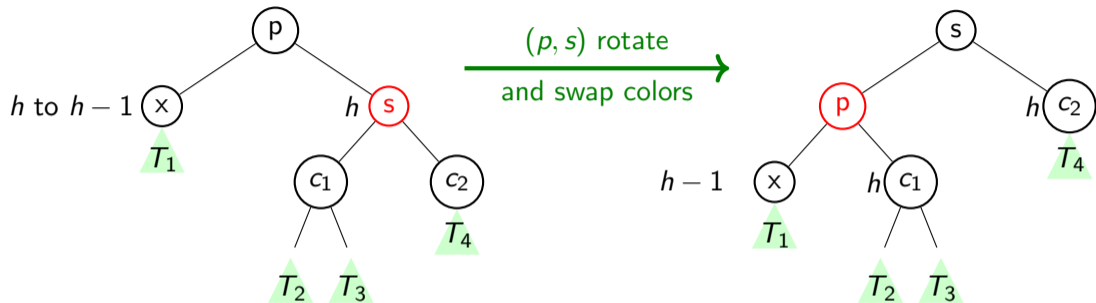
Violation solved!

Case 2: x is not **red** and root.

Do nothing.

Case 3: x is not **red** and the sibling of x is **red**

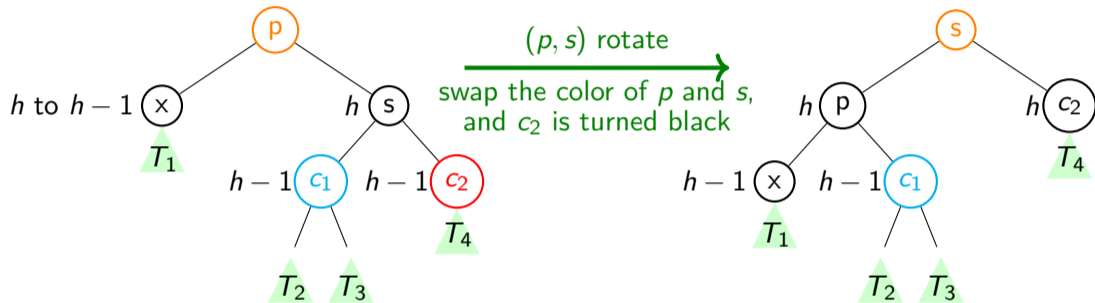h to $h-1$ (p, s) rotate and swap colors

The transformation does not solve the height violation at the parent of x but changes the sibling of x from **red** to **black**.

Exercise 7.9
Why must p, c_1, and c_2 be non-null and **black**?

Case 4.1: x is not **red**, the sibling of x is **black**, and the far nephew is **red**

The color of p and c_1 does not matter.

(p, s) rotate
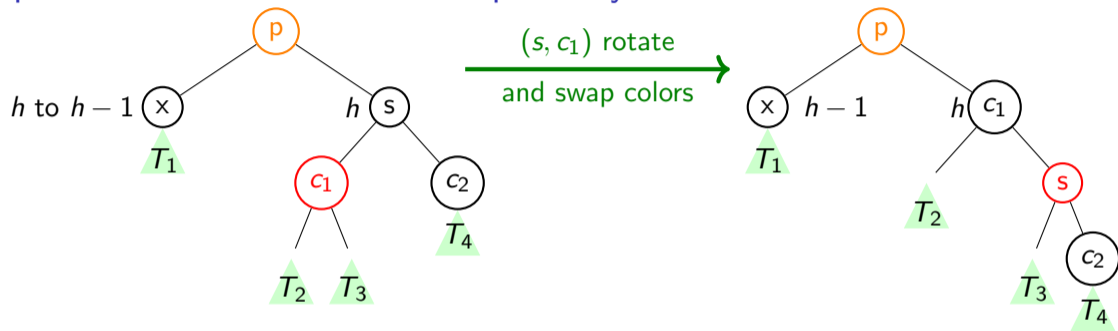swap the color of p and s,
and c_2 is turned black

The above transformation solves the black height violation.

Exercise 7.10
Write the above case if x is the right child of p.

Case 4.2: x is not **red** and the sibling is **black**, and the near and far nephews are **red** and not **red** respectively
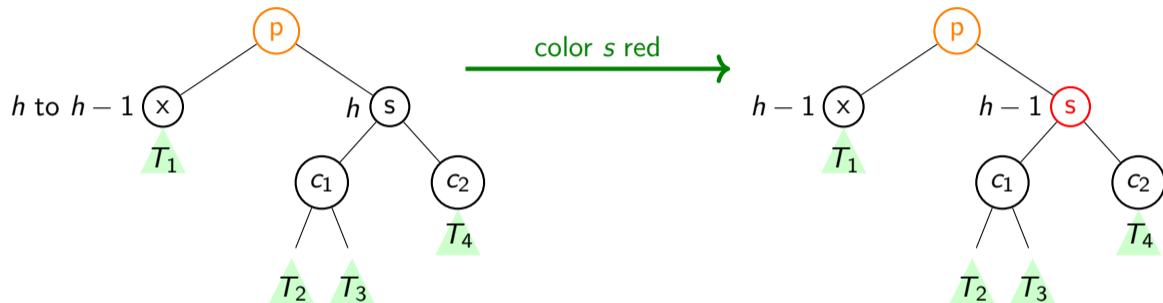

The above transformation does not solve the height violation. It changes the right child of the sibling from not **red** to **red**, which is the case 4.1.

Exercise 7.11

a. Why can case 4.1 transformation not be applied to case 4.2?

b. Write the above case if x is the right child of p.

Case 4.3: x is not **red**, the sibling of x is **black**, and the nephews of x are not **red**

The above transformation reduces $bh(p)$ by 1. There may be a potential violation at p, which is at the lower level. If p is red, then p and s will also have red-red violation.

We apply the case analysis again at node p. The only case that kicks the can upstairs!! All cases are covered.

Structure among cases

▶ Cases 1, 2, and 4.1 solve the violation at the node.
▶ Case 3 turns the violation into 4.1, 4.2, or $(4.3 \rightarrow 1)$.
▶ Case 4.2 turns the violation into 4.1.
▶ Case 4.3 moves the violation from x to its parent p.

Summary of deletion

Delete *n* like BST, if there is a height violation at the parent of *x* then run the following algorithm.

Algorithm 7.2: HEIGHTREPAIR(Node x)

1 ASSUME(there is no **red-red** violation at any other node);
2 ASSUME(there is no height violation below x);
3 **if** *x is* **red** **then** x.color := black; **return**;   // Case 1;
4 **if** *x is root* **then** **return**;   // Case 2;
5 p := parent(x); s := sibling(x);
6 c1 := (x = left(p)) ? left(s) : right(s);   // near nephew;
7 c2 := (x = left(p)) ? right(s) : left(s);   // far nephew;
8 **if** *s is* **red** **then** RotateAndSwapColor(p,s); HEIGHTREPAIR(x); **return**;   // Case 3 $\rightarrow$ 4.1, 4.2, or (4.3$\rightarrow$1);
9 **if** *c2 is* **red** **then** RotateAndSwapColor(p,s); color(c2) := black; **return**;   // Case 4.1;
10 **if** *c1 is* **red** **then** RotateAndSwapColor(s,c1);HEIGHTREPAIR(x); **return**;   // Case 4.2 $\rightarrow$ 4.1;
11 color(s) := **red**;
12 HEIGHTREPAIR(p);   // Case 4.3
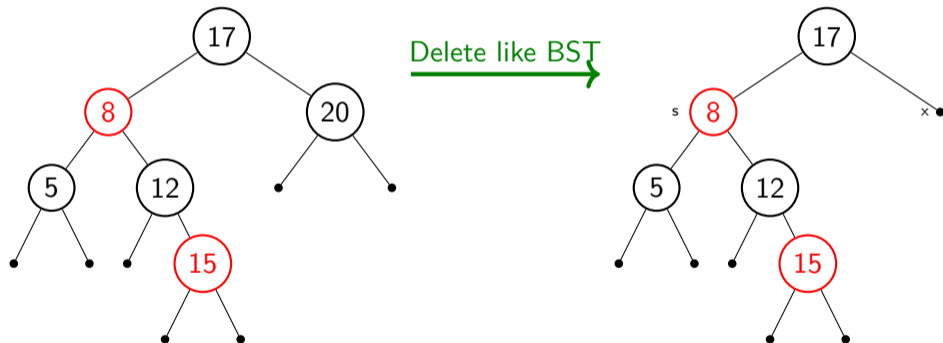

Exercise 7.12
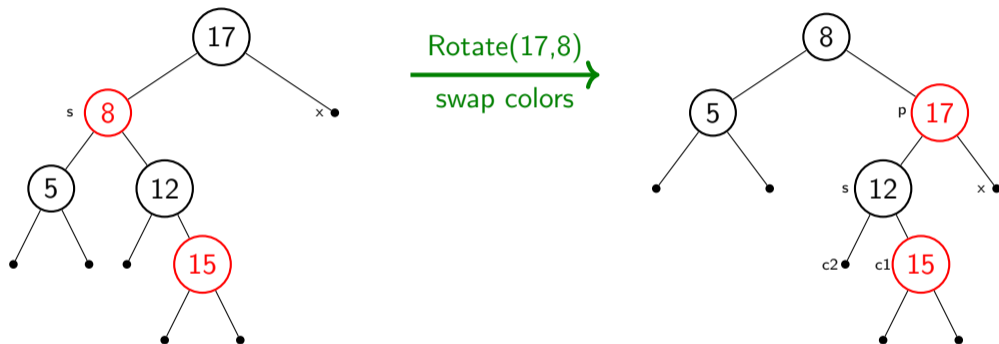Is the above a tail-recursive function?

Example: deletion in Red-black tree

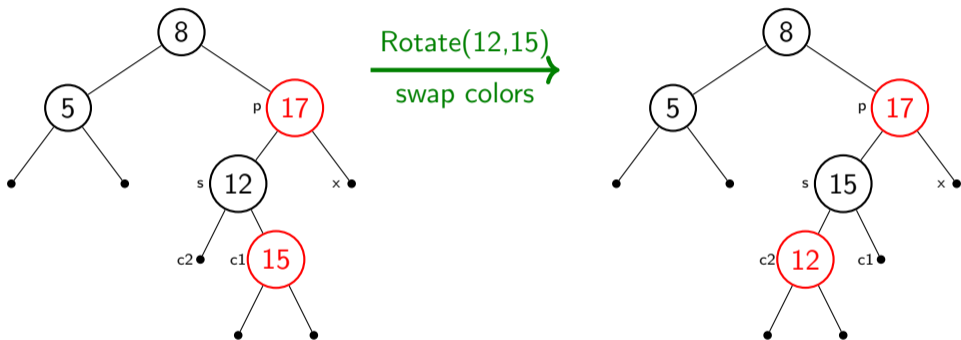## Example 7.8

Let us delete 20 in the following red black tree

Delete like BST

Node x is in Case 3, where the sibling s is **red**.

Example: deletion in red-black tree (continued) – applying Case 3
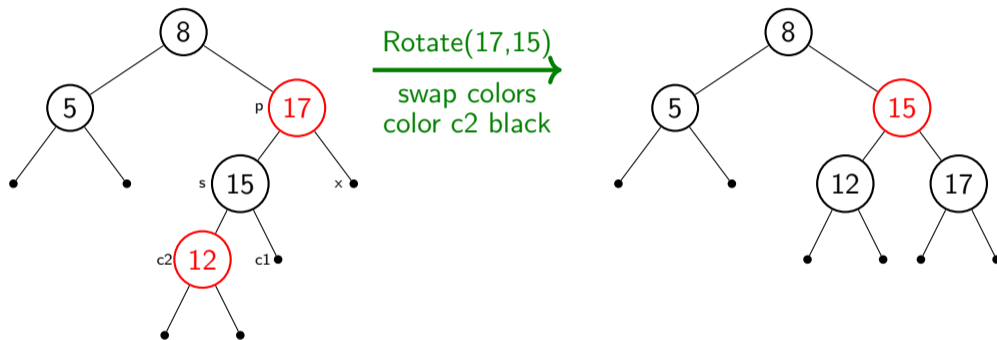

Rotate(17,8)
swap colors

Now node x is in Case 4.2, where the sibling s is not **red**, nephew c2 is not **red**, nephew c1 is **red**.

Example: deletion in red-black tree (continued) – applying Case 4.2

Now node x is in Case 4.1, where the sibling s is not **red** and nephew c2 is **red**.

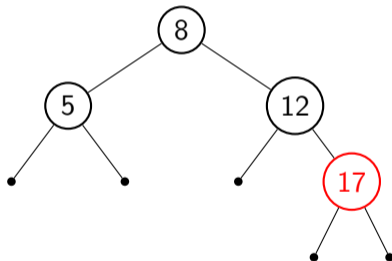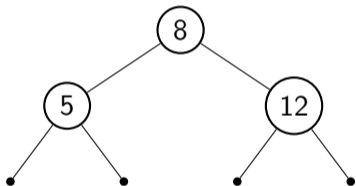# Example: deletion in red-black tree (continued) – applying Case 4.1

Rotate(17,15)

swap colors
color c2 black

Height imbalance due to the deletion is repaired.

Exercise: identify the case for deletion and apply deletion

Exercise 7.13

On deletion of 12 in the following red-black trees, which case of deletion will be applied?

Summary of deletion

1. Delete like BST. There may be a black height violation at the child of the deleted node.
2. While case 4.3 occurs, re-color nodes and move up the black height violation.
3. For all the other cases, we rotate or re-color, and the violation is finished.
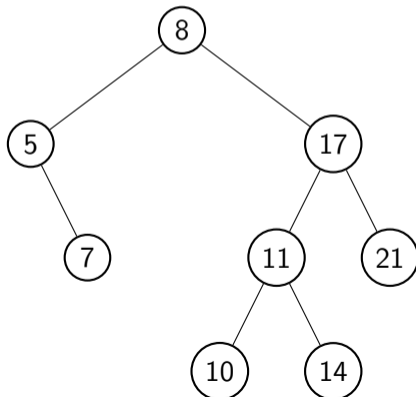
AVL (Adelson, Velsky, and Landis) tree

Definition 7.3

An AVL tree is a binary search tree such that for each node *n*

$$|height(right(n)) - height(left(n))| \leq 1.$$
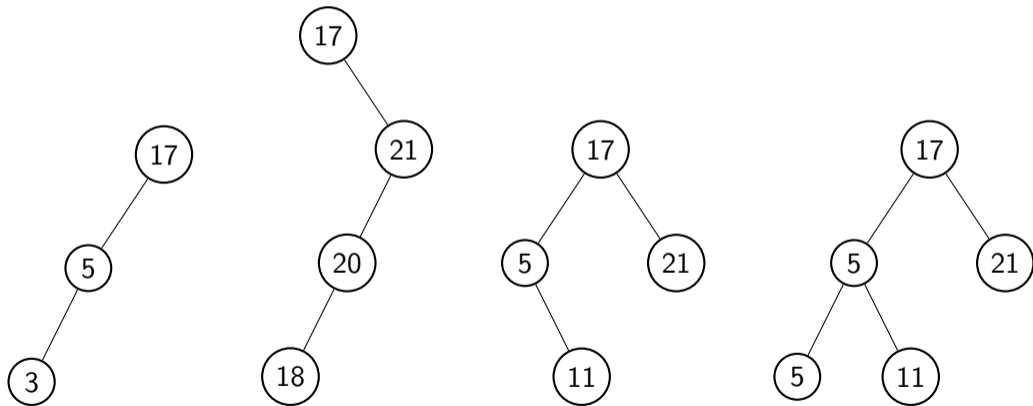
Example 7.9

An example of an AVL tree.

Exercise: Identify the AVL trees

Exercise 7.22
Which of the following are AVL trees?

Topic 7.8

Height of AVL tree

AVL tree height

Theorem 7.1
The height of an AVL tree T having n nodes is $O(\log n)$.

Proof.
Let $n(h)$ be the minimum number of nodes for height h.

Base case:
$n(1) = 1$ and $n(2) = 2$.

Induction step:
Consider an AVL tree with height $h \geq 3$. In the minimum case, one child will have a height of $h - 1$ and the other child will have a height of $h - 2$. (Why?)

Therefore, $n(h) = 1 + n(h - 1) + n(h - 2)$. ...

Commentary: We need to show that $n(h) > n(h - 1)$ is monotonous. Ideally, $n(h) = 1 + n(h - 1) + min(n(h - 2), n(h - 1))$. This proves that $n(h) > n(h - 1)$.

AVL tree height(2)

Proof(continued.)

Since $n(h-1) > n(h-2)$,

$$n(h) > 2n(h-2).$$

Therefore,

$$n(h) > 2^i n(h-2i).$$

For $i = h/2 - 1$ (Why?),

$$n(h) > 2^{h/2-1} n(2) = 2^{h/2}.$$

> **Commentary:** Here is the explanation of the last step. Consider an AVL tree with m nodes and h height. By definition, $h(n) \leq m$. Since $h < 2\log n(h)$, $h < 2\log m$. Therefore, h is $O(\log m)$.

Therefore,

$$h < 2\log n(h).$$

Therefore, the height of an AVL tree is $O(\log n)$. (Why?)
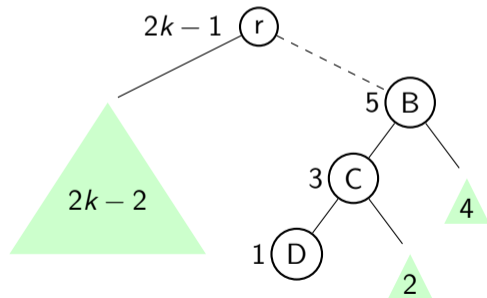
Closest leaf

Theorem 7.2
Let T be an AVL tree. Let the level of the closest leaf to the root of T is k.

$$height(T) \leq 2k - 1$$

Proof.
Let D be the closest leaf of the tree.

- ▶ The height of $right(C)$ cannot be more than 2. (Why?)
- ▶ Therefore, the maximum height of C is 3.
- ▶ Therefore, the maximum height of $right(B)$ is 4.
- ▶ Therefore, the maximum height of B is 5.
- ▶ Continuing the argument, the maximum height of root r is $2k - 1$.

□

A part of AVL is a complete tree

Theorem 7.3
Let T be an AVL tree. Let the level of the closest leaf to the root of T is k. Upto level $k-2$ all nodes have two children.

Proof.
A node at level $k-2-i$ cannot be a leaf for $i \geq 0$. (Why?)

Let us assume that a node n at level $k-2-i$ has a single child n'.

The height of n' cannot be more than 1. (Why?)

Therefore, n' is a leaf. Contradiction. □

Exercise 7.23
Show T has at least 2^{k-1} nodes.

Another proof of tree height bound

Let T have n nodes and the height of T be h.

We know the following from the previous theorems.

- $n \geq 2^{k-1}$, and
- $2k - 1 \geq h$.

Therefore,

$$n \geq 2^{k-1} \geq 2^{(h-1)/2}$$

Exercise 7.24
What is the maximum number of nodes given height h?

Problem: A sharper bound for the AVL tree

Exercise 7.25

a. Find largest c such that $c^{k-2} + c^{k-1} \geq c^k$

b. Recall $n(h) = 1 + n(h-1) + n(h-2)$. Let c_0 be the largest c. Show that $n(h) \geq c^{h-1}$.

c. Prove that the above bound is a sharper bound than our earlier proof.

Topic 7.9

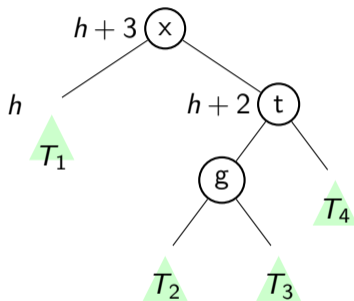Insertion and deletion in AVL trees

Insert and delete

Insert and delete like BST.

At most a path to one node may have height imbalances of 2. (Why?)

We have to repair height imbalances by rotations around the deepest imbalanced node.
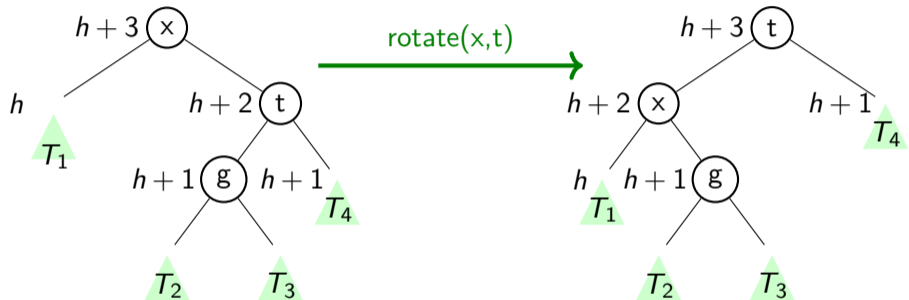
Rebalancing AVL trees

Let x be the deepest imbalanced node. Let t be the taller child. Let g be the grandchild via t that is not on the straight path from x.
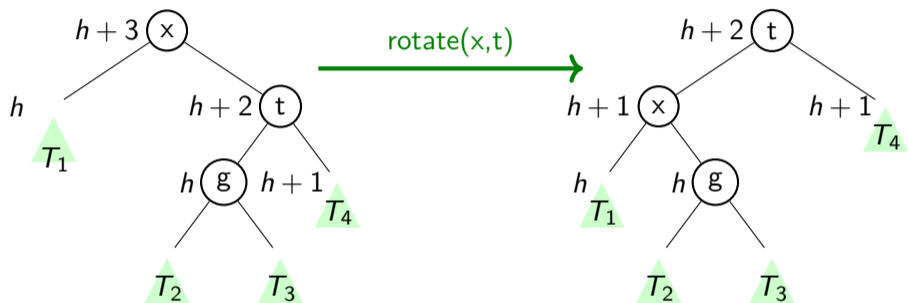

Three cases:

1. Case 1: Height of g is $h+1$ and T_4 is $h+1$.
2. Case 2: Height of g is h and T_4 is $h+1$.
3. Case 3: Height of g is $h+1$ and T_4 is h.

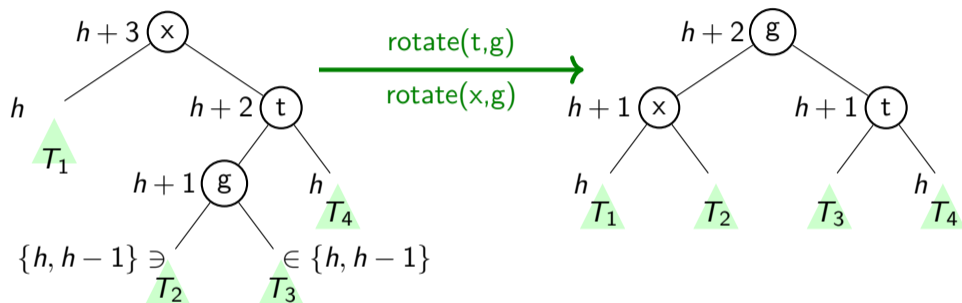# Case 1: Both grandchildren via t have height $h+1$

The imbalance in the subtree is repaired. We check the parent of t.

Case 2: Right-left grandchild has height h

Imbalance is repaired. But, the parent of t may need repair.

Case 3: Right right grandchild has height h

Imbalance is repaired. But, the parent may need repair.