CS213/293 Data Structure and Algorithms 2023

Lecture 9: Red-Black Trees

Instructor: Ashutosh Gupta

IITB India

Compile date: 2023-09-20

Topic 9.1

Balance and rotation

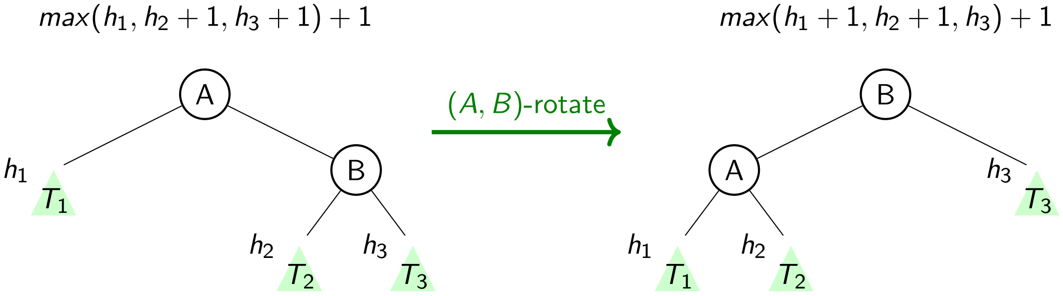
Maintain balance

BST may have a large height.

Height is directly related to branching. More branching implies a shorter height.

We call BST imbalanced when the difference between the left and right subtree height is large.

Balancing height by rotation



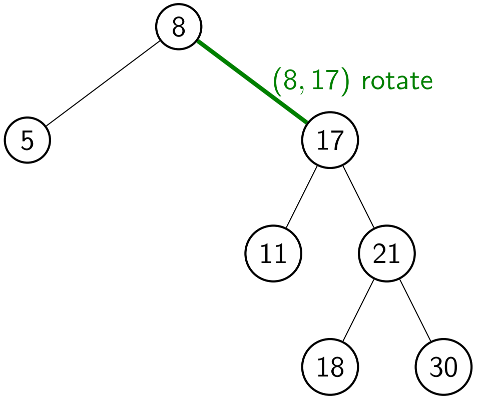
If $h_3 > h_2 = h_1$, if we rotate the BST, we will get a valid more balanced BST with less height.

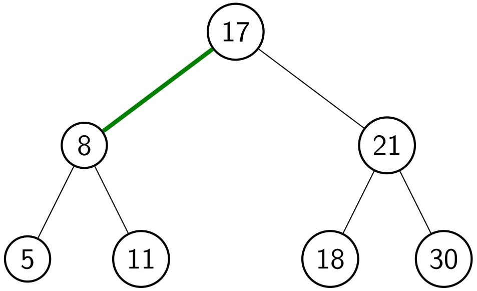
Note that B could have been the left child. For this situation, we can define symmetric rotation.

Example: rotation

Example 9.1

In the following BST, we can rotate 8-17 *edge.*





When to rotate? Can rotation only fix imbalance?

Rotation is a local operation, which must be guided by non-local measure height.

We need a definition of balance such that rotations operations should be able to achieve the definition.

Design principle:
We minimize the number of rotations while allowing some imbalance.

Topic 9.2

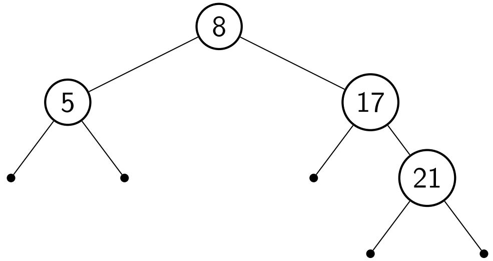
Red-black tree

Null leaves

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

Example 9.2

The following tiny nodes are the dummy null nodes.



Red-black tree

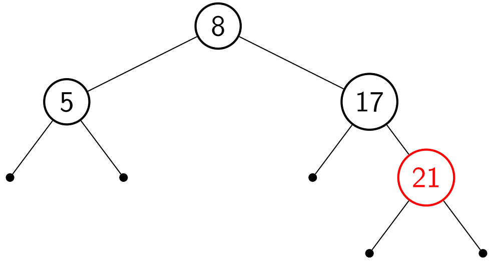
Definition 9.1

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

- *All nodes are colored either* **red** *or* **black**
- *Null leaves have no color*
- *Root is colored black*
- *All* **red** *nodes have* **black** *children*
- *All paths from the root to null leaves have the same number of* **black** *nodes.*

Example 9.3

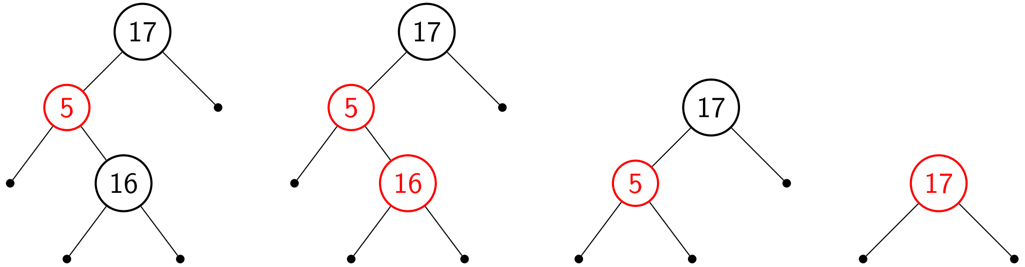
An example of a red-black tree.



Exercise: Identify red-black trees

Exercise 9.1

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 9.2

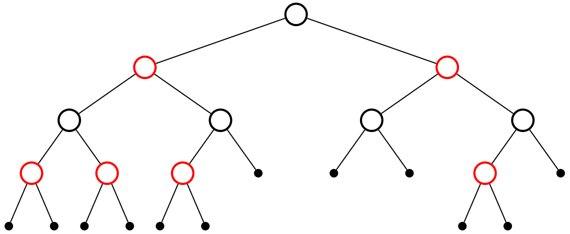
The black height (bh) for each node is defined as follows.

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

Example: black height

Example 9.4

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



Exercise 9.2

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, $n = 2^{2h} - 1$.

$$\log_4 n < h < 1 + \log_2 n$$

Search, Maximum, and Successor/Predecessor

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

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

The question is how do we do insertion and deletion on a red-black tree?

Topic 9.3

Insertion in red-black tree

BST insertion in red-black tree

1. Follow the usual procedure of insertion in the BST, which inserts the new node *n* as a leaf.
   - Note that there are dummy nodes in the red-black tree. *n* is inserted as the parent of a dummy node.
2. We color *n* red.

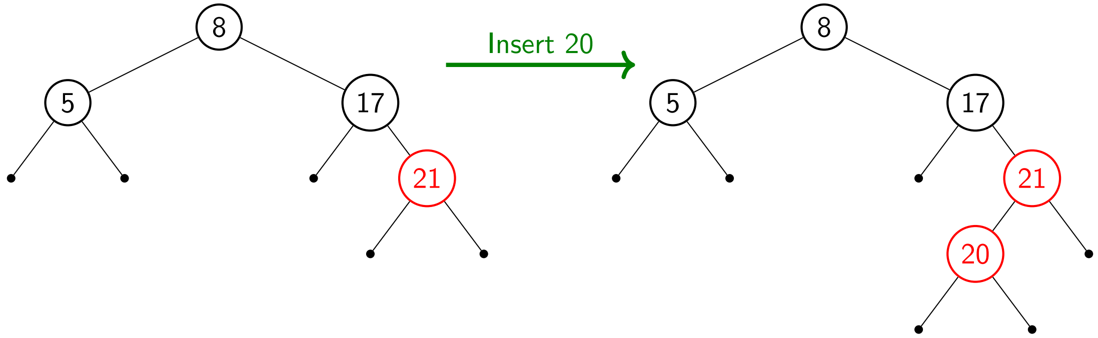
- Good news: No change in the black height of the tree.
- Bad news: *n* may have a *red* parent.

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 9.5

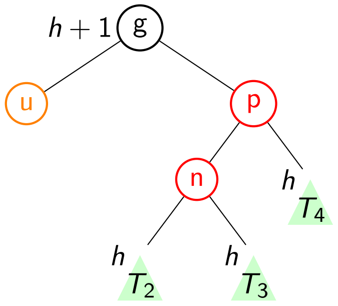
Inserting 20 *in the following tree.*



The insertion results in violation of the conditions of red-black tree that red nodes can only have black children.

Red-red violation

After insertion, we may have a **red-red** violation, where a **red** node has a **red** child. Orange color means that we need to consider all possible colors of the nodes.



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

We have three cases.

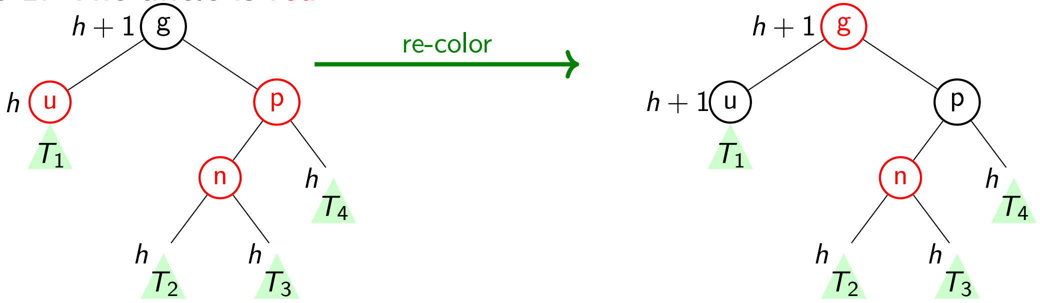
- Case 1: u is **red**
- Case 2: u is not **red** and g to n path is not straight
- Case 3: u is not **red** and g to n path is straight

No transformation should change the black height of g.

Exercise 9.3

Why g must exist and be black?

Case 1: The uncle is red



In the subtree of g, we have no change in the black height, and in the subtree, there is no red-red error.

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.

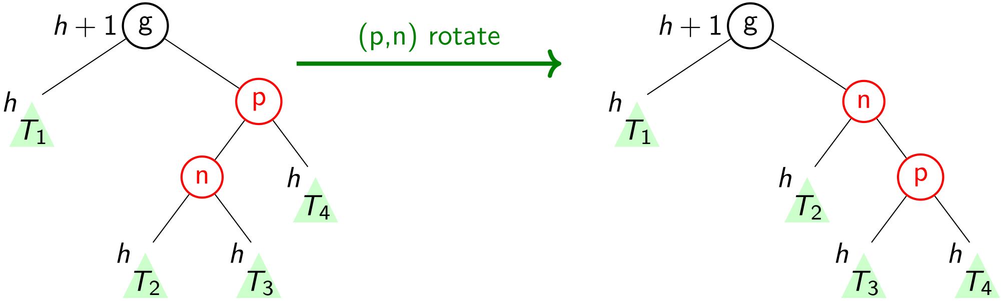
Commentary: Possibility 1: Nothing. Possibility 2: We have a red-red violation a level up and need to apply the transformations there. Possibility 3: turn g back to black.

Exercise 9.4

What do we do in each case?

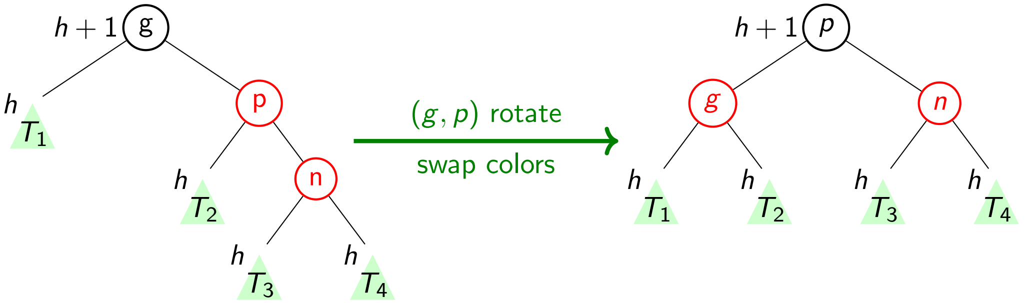
Case 2: The uncle is not red and the path to the grandparent is not straight

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 9.5

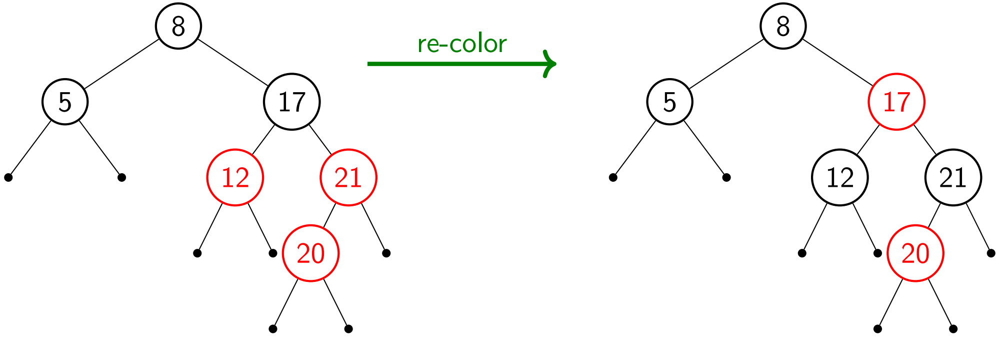
a. *Why roots of T_2, T_3, and T_4 are* **black***?*

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 9.6

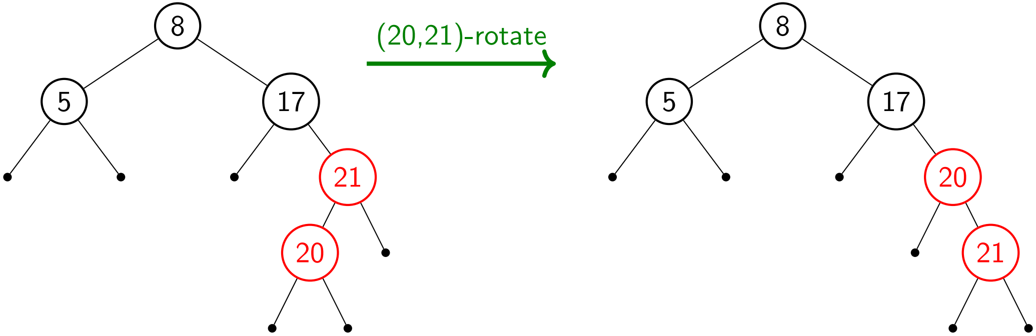
We just inserted 20 *in the following tree. We need to apply case 1 to obtain a red-black tree.*



Example: red-red correction case 2

Example 9.7

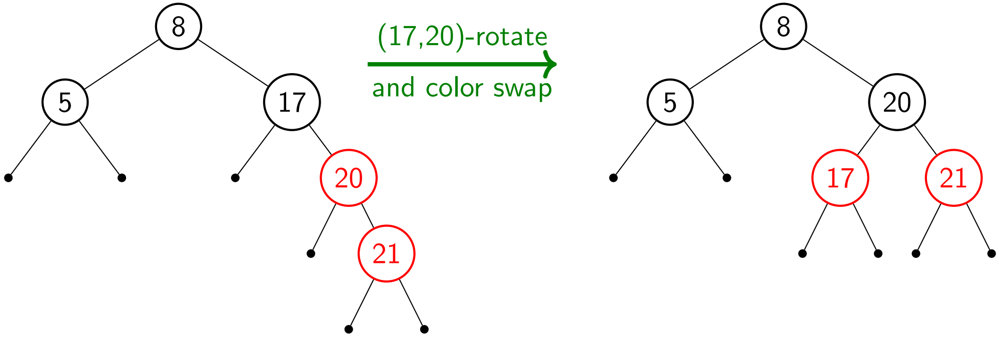
Consider the following example. We are attempting to insert 20. We apply case 2 to move towards a red-black tree.



The above is not a red-black tree. We need to further apply case 3 to finally obtain the red-black tree.

Example: red-red correction case 3 (continued)

We apply case three as follows.



Summary of insertion

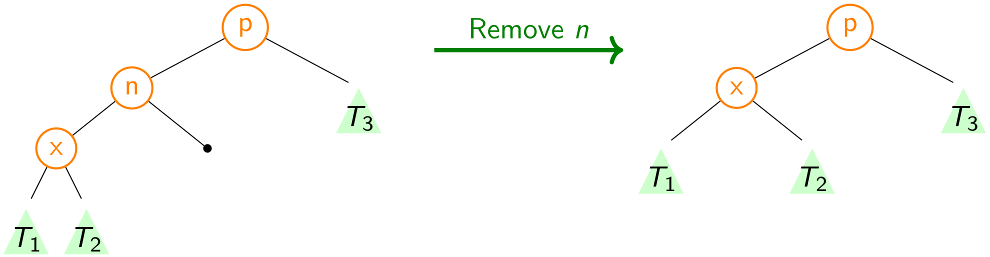
1. Insert like BST and assign **red** color to the new node.
2. While we have case 1, re-color nodes and move up the **red**-**red** violation.
3. If we find case 2 or 3, we rotate and the violation is finished.
4. If the root becomes **red** in the process, then turn it back to black.

Topic 9.4

Deletion 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 that has 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 the violations are possible at the same time.

Violation removal procedure

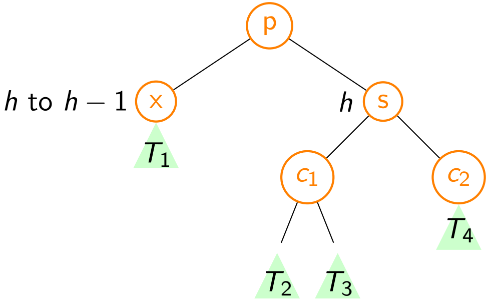
- To remove violation at the subtree of x, we may recolor and rotate around x.
- We may fix the violation at x or move the violation to the parent of x.

Therefore, we present the violation resolution cases in a generalized description.

In the following, x may be some internal node.

Violation pattern

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



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

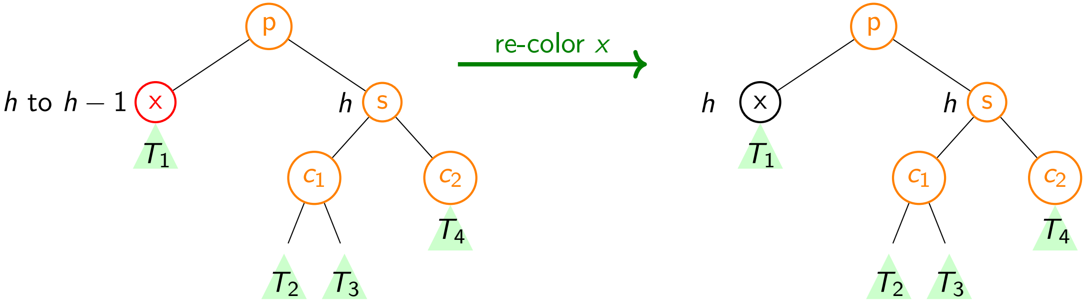
There are six cases

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.

Exercise 9.6
Show: If x is not root and not **red***, s must exist.*

Case 1: x is **red**

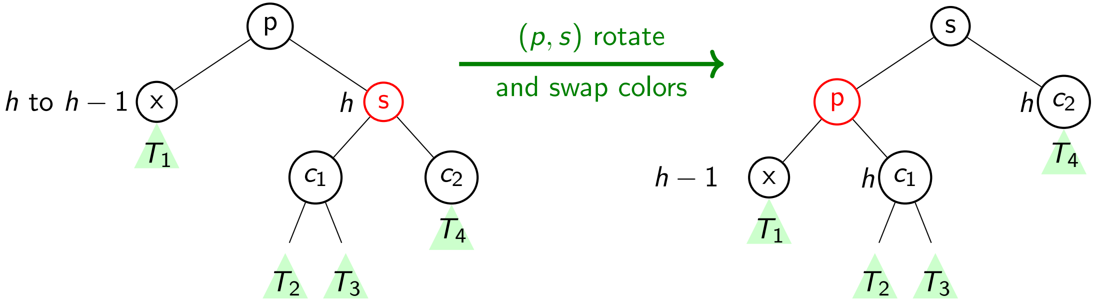


Violation solved!

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

Do nothing.

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

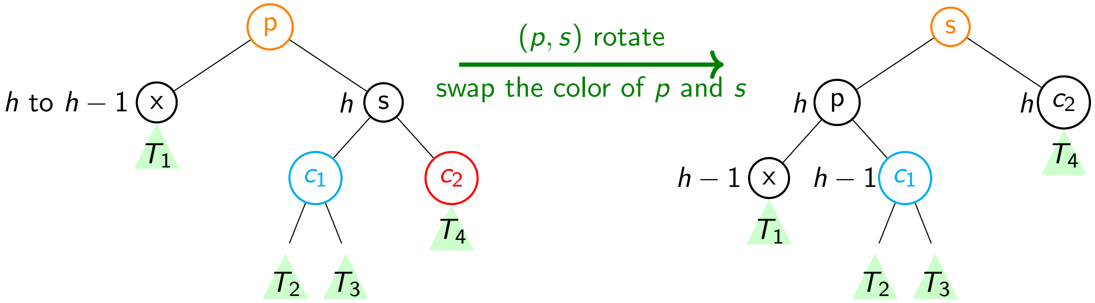


The transformation does not solve the height violation at parent of x but changes the sibling of x from **red** to **black**.

Exercise 9.7

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

Case 4.1: x is not **red** and sibling of x is **black**, and the right nephew is **red**(Assuming x is left child)

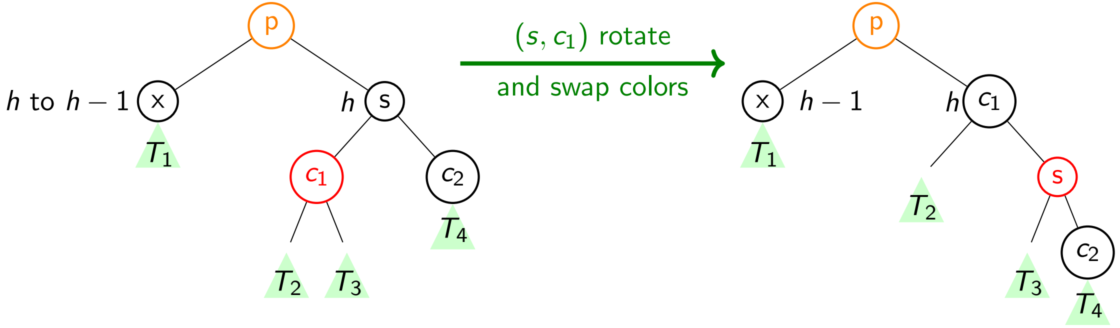


The above transformation solves the black height violation.

Exercise 9.8

Write the above case if x is the right child of p?

Case 4.2: x is not **red** and the sibling of x is **black**, and the left and right nephews are **red** and not **red** respectively



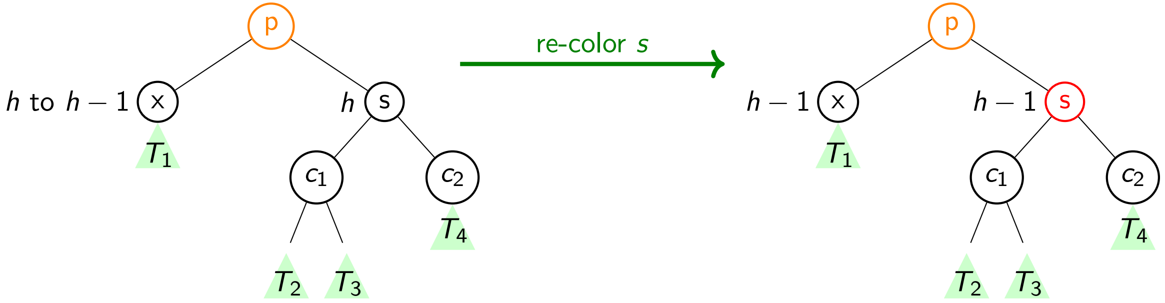
The above transformation does not solve the height violation. It changes the right child of the sibling from not **red** to **red**, which is the case 4.1.

Exercise 9.9

a. Why case 4.1 transformation cannot 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** and the sibling of x is **black**, and the nephews of x are not **red**



The above transformation reduces $bh(p)$ by 1 and there may be potential violation at p, which is at the lower level.

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

Structure among cases

- Case 1, 2, and 4.1 solve the violation at the node.
- Case 3 turns the violation into 4.1 or 4.2.
- Case 4.2 turns the violation into 4.1.
- Case 4.3 moves the violation from x to its parent p.

Summary of deletion

1. Delete like BST. There may be black height violation at the child of deleted node.
2. While we have case 4.3, 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.

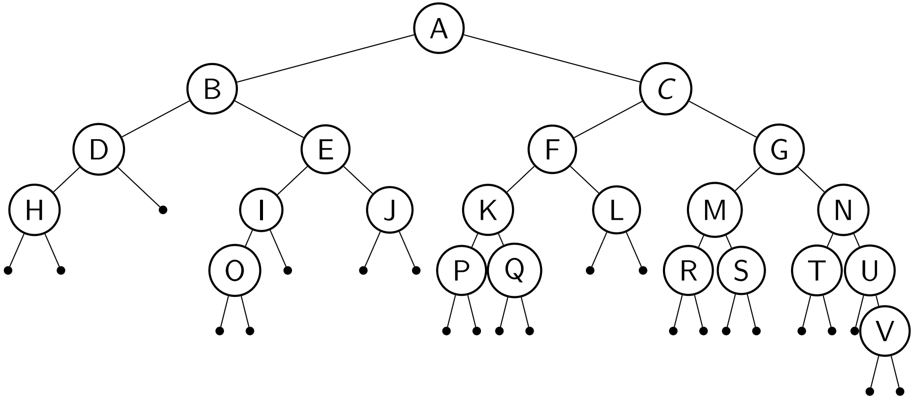
Topic 9.5

Problems

Insert and delete

Exercise 9.10

Consider the tree below. Can it be colored and turned into a red-black tree? If we wish to store the set 1, . . . , 22, label each node with the correct number. Now add 23 to the set and then delete 1. Also do the same in the reverse order. Are the answers the same? When will the answers be the same?



Topic 9.6

Extra slides: AVL trees

AVL (Adelson, Velsky, and Landis) tree

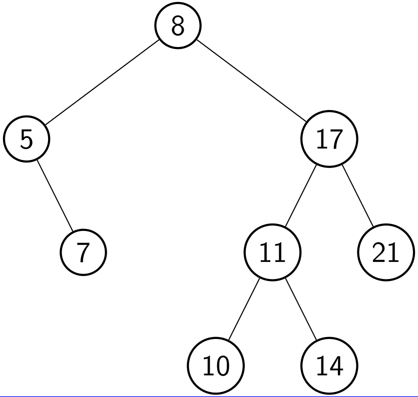
Definition 9.3

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

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

Example 9.8

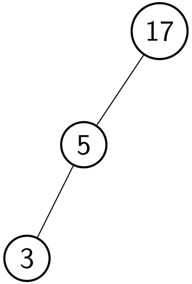
An example of an AVL tree.

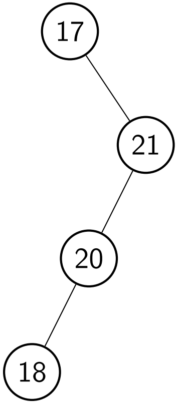


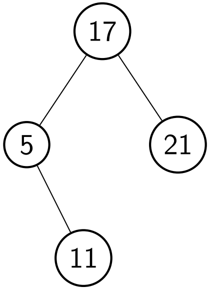
Exercise: Identify the AVL trees

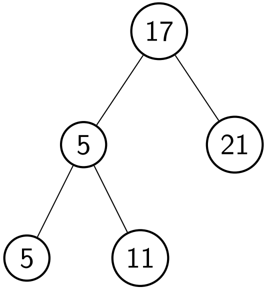
Exercise 9.11

Which of the following are AVL trees?









Topic 9.7

Height of AVL tree

AVL tree height

Theorem 9.1

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 height $n - 1$ and the other $n - 2$. (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 9.2

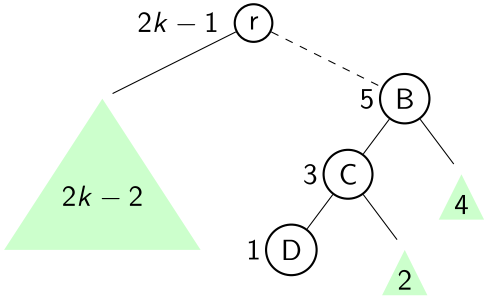
Let T be an AVL tree. Let the level of the closest leaf to the root of T is k.

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

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 9.3

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. (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 9.12

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 9.13

What is the maximum number of nodes given height h?

Problem: A sharper bound for the AVL tree

Exercise 9.14

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 9.8

Insertion and deletion

End of Lecture 9