CSCI 104 Binary Search Trees and Balanced Binary Search Trees using AVL Trees

Mark Redekopp

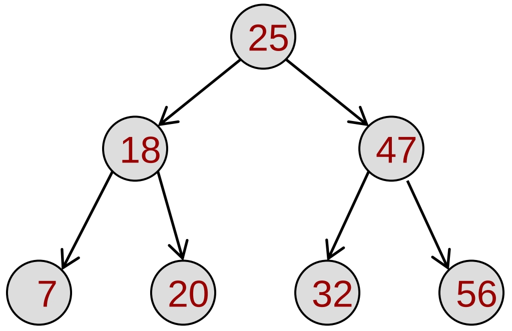
David Kempe

Properties, Insertion and Removal

BINARY SEARCH TREES

Binary Search Tree

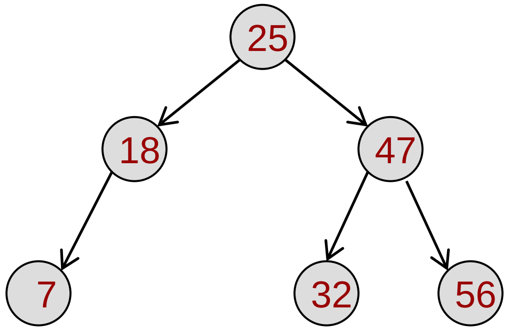
- Binary search tree = binary tree where all nodes meet the **binary search property** which states:
  - All values of nodes in **left** subtree are **less-than** (or equal in some implementations) than the parent's value
  - All values of nodes in **right** subtree are **greater-than** (or equal in some implementations) than the parent's value
- BSTs generally implement **maps/sets** (where keys must be unique) and support 3 primary operations:
  - Insert
  - Remove
  - Find/Lookup



If we wanted to print the values in sorted order would you use an pre-order, in-order, or post-order traversal?

BST-Find

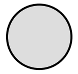
- To find a node with a given key
  - If node pointer is NULL, the key does NOT exist in the tree, STOP!
  - Otherwise, check if current node's key equals the desired key
    - If so, STOP! and return a pointer to that node
  - If desired key is LESS-THAN current node's key, go LEFT
  - If desired key is GREATER-THAN current node's key, go RIGHT



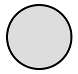
BST Insertion

- To insert an item walk the tree (go left if value is less than node, right if greater than node) until you find an empty location, at which point you insert the new value
- Practice: Build a BST from the data values below

Insertion Order: 25, 18, 47, 7, 20, 32, 56



Insertion Order: 7, 18, 20, 25, 32, 47, 56

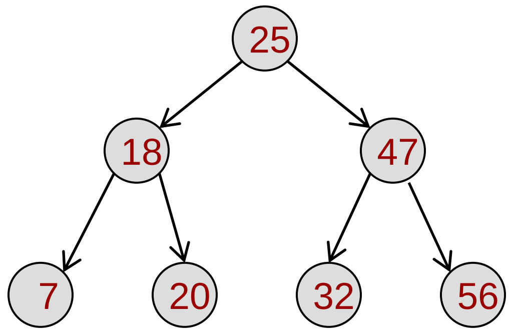


Important: To be efficient (useful) we need to keep the binary search tree balanced, but that is NOT guaranteed by the basic insert() algorithm.

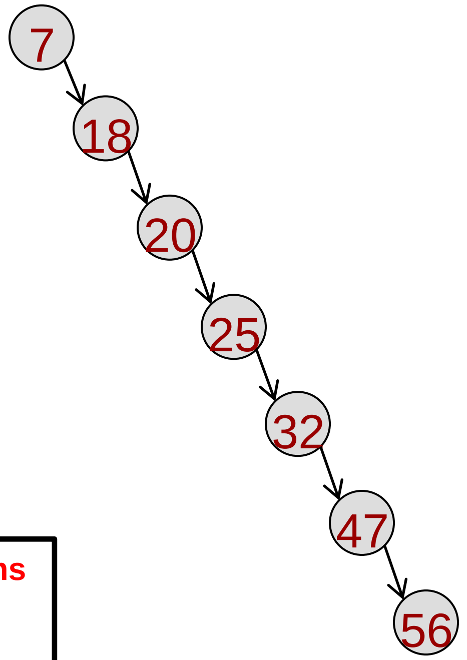
BST Insertion

- To insert an item, walk the tree (go left if value is less than node, right if greater than node) until you find an empty location, at which point you insert the new value
- Practice: Build a BST from the data values below
- https://www.cs.usfca.edu/~galles/visualization/BST.html

Insertion Order: 25, 18, 47, 7, 20, 32, 56



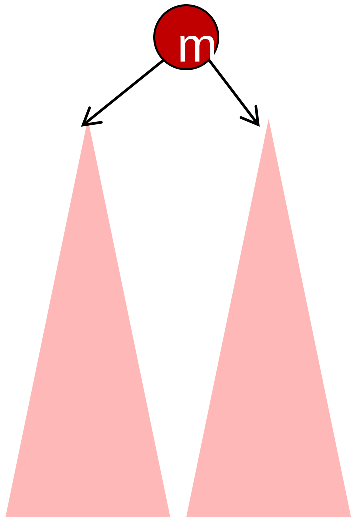
Insertion Order: 7, 18, 20, 25, 32, 47, 56



A major topic we will talk about is algorithms to keep a BST balanced as we do insertions/removals

Successors & Predecessors

- Let's take a quick tangent that will be necessary when we implement **BST Removal**
- Given a node in a BST
  - Its **predecessor** is defined as the **next smallest** value in the tree
  - Its **successor** is defined as the **next biggest** value in the tree
- Where would you expect to find a node's predecessor?
- Where would find a node's successor?



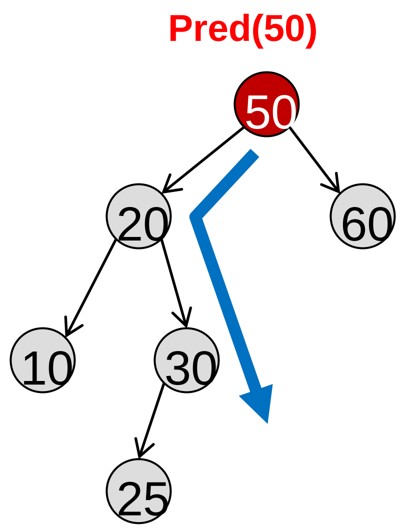
```
// Node definition
struct TNode
{
  int val;
  TNode *left, *right;
  Tnode *parent;
};
```

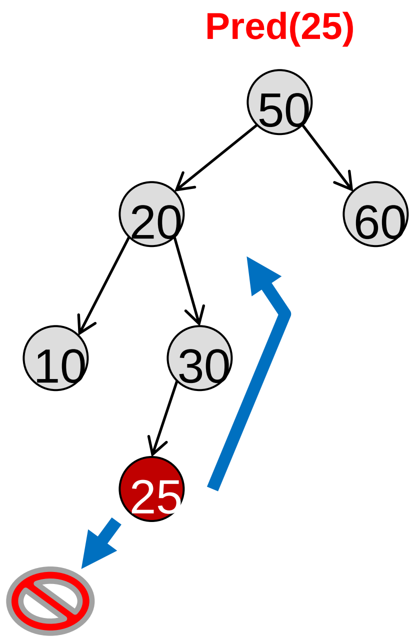
Predecessors

- If left child exists, predecessor is the **right most node** of the **left** subtree
- Else walk up the ancestor chain until you traverse the first right child pointer (find the first node who is a right child of his parent…that parent is the predecessor)
  - If you get to the root w/o finding a node who is a right child, there is no predecessor

If you have no left pointer, then realize that you must be someone's successor [succ(pred(m)) = m].
Think about who, if they wanted to find their successor (go right once and left as far as you can), would land on you.

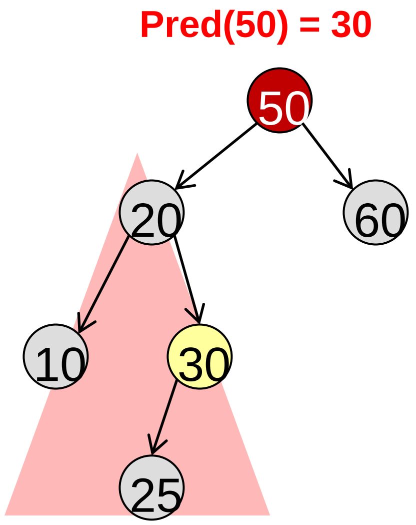
Code to check if you are the left child of your parent:

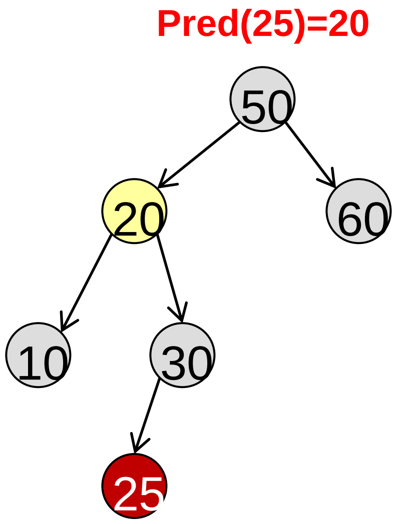




Predecessors

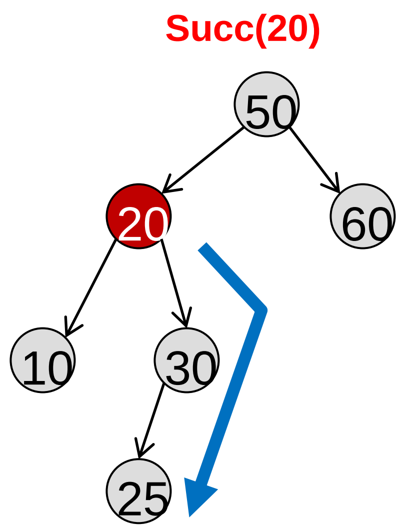
- If left child exists, predecessor is the right most node of the left subtree
- Else walk up the ancestor chain until you traverse the first right child pointer (find the first node who is a right child of his parent…that parent is the predecessor)
  - If you get to the root w/o finding a node who is a right child, there is no predecessor

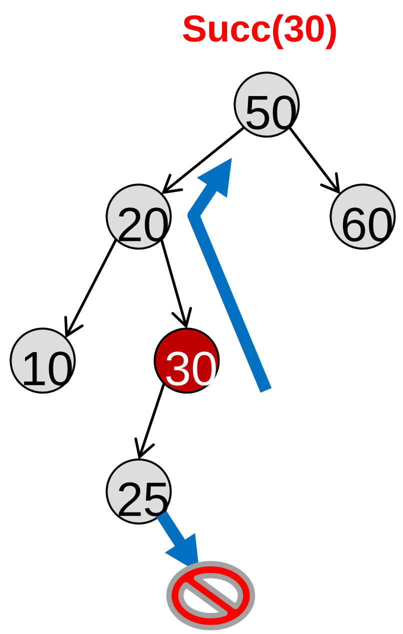




Successors

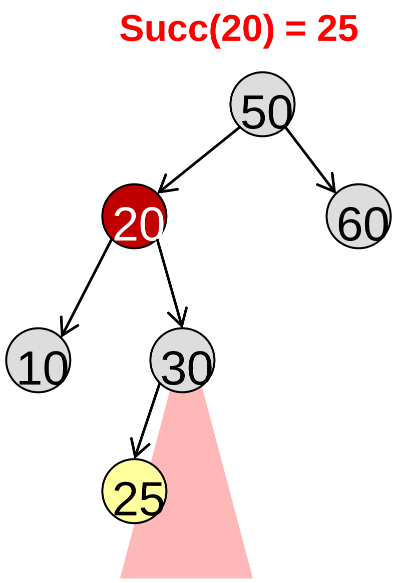
- If right child exists, successor is the left most node of the right subtree
- Else walk up the ancestor chain until you traverse the first left child pointer (find the first node who is a left child of his parent…that parent is the successor)
  - If you get to the root w/o finding a node who is a left child, there is no successor

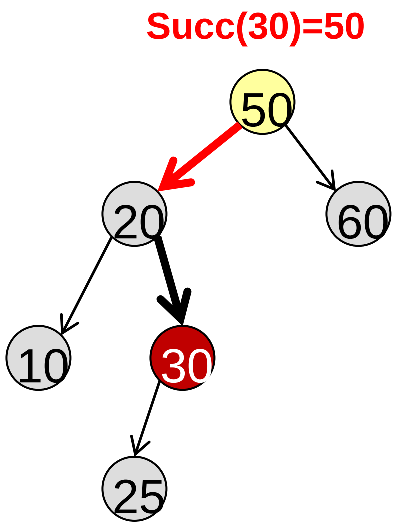




Successors

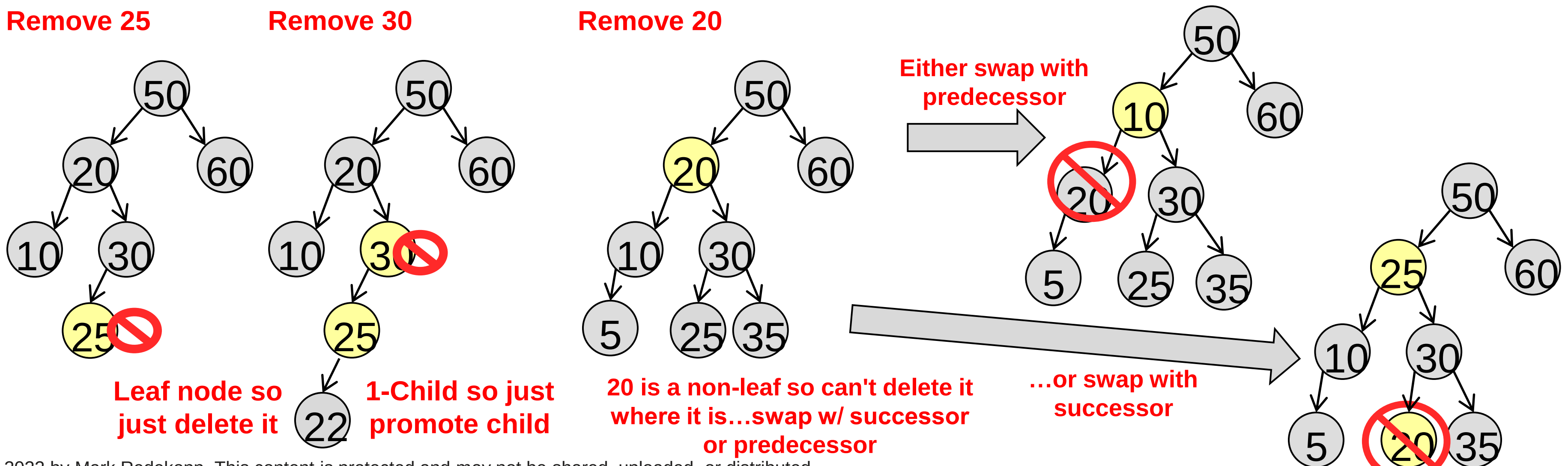
- If right child exists, successor is the left most node of the right subtree
- Else walk up the ancestor chain until you traverse the first left child pointer (find the first node who is a left child of his parent…that parent is the successor)
  - If you get to the root w/o finding a node who is a left child, there is no successor





BST Removal

- How we remove is based on the number of children the node has…
  - First find the value to remove by walking the tree
  - 0 children: If the value is in a leaf node, simply remove that leaf node
  - 1 child: Promote the child into the node's location
  - 2 children: Swap the value with its in-order successor or predecessor and then remove from its new location
    - We can maintain the BST properties by putting a value's successor or predecessor in its place
    - After swap, we have converted to 0-children or 1-child case (i.e. non-leaf node's successor or predecessor is guaranteed to not have 2 children) which we then perform


Remove 25
Remove 30
Remove 20
Either swap with predecessor
Leaf node so just delete it
1-Child so just promote child
20 is a non-leaf so can't delete it where it is…swap w/ successor or predecessor
…or swap with successor

Worst Case BST Efficiency

- Insertion
  - Balanced: ____________
  - Unbalanced: ____________
- Removal
  - Balanced: ___________
  - Unbalanced: ____________
- Find/Search
  - Balanced: _____________
  - Unbalanced: _____________

```
#include<iostream>

// Node definition
template <typename T>
struct TNode
{
  T val;
  TNode *left, *right;
  Tnode *parent;
};

// Bin. Search Tree
template <typename T>
class BST
{
public:
 BTree();
 ~BTree();
 virtual bool empty();
 virtual void insert(const T& v);
 virtual void remove(const T& v);
 virtual T* find(const T& v);
protected:
 TNode<T>* root_;
};
```

BST Efficiency

- Insertion
  - Balanced: O(log n)
  - Unbalanced: O(n)
- Removal
  - Balanced : O(log n)
  - Unbalanced: O(n)
- Find/Search
  - Balanced : O(log n)
  - Unbalanced: O(n)

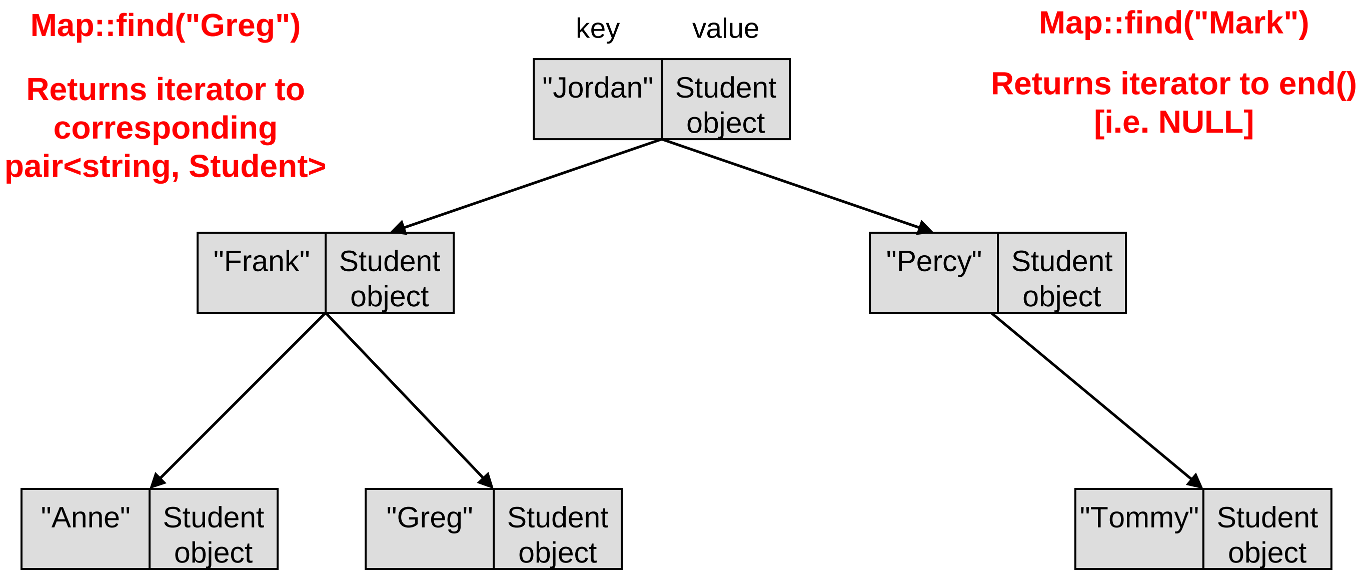
```
#include<iostream>

// Node definition
template <typename T>
struct TNode
{
  T val;
  TNode *left, *right;
  Tnode *parent;
};

// Bin. Search Tree
template <typename T>
class BST
{
public:
 BTree();
 ~BTree();
 virtual bool empty();
 virtual void insert(const T& v);
 virtual void remove(const T& v);
 virtual T* find(const T& v);
protected:
 TNode<T>* root_;
};
```

Trees & Maps/Sets

- C++ STL "maps" and "sets" use binary search trees internally to store their keys (and values) that can grow or contract as needed
- This allows O(log n) time to find/check membership
  - BUT ONLY if we keep the tree balanced!

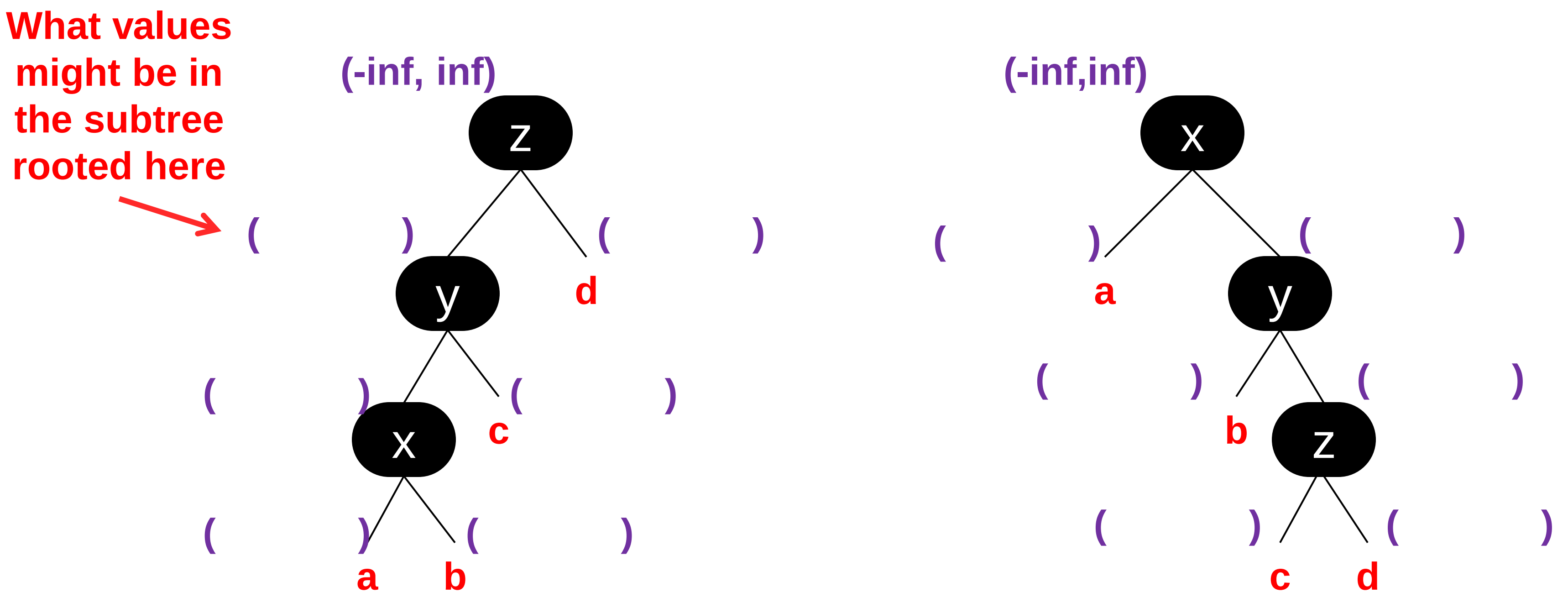


The key to balancing…

TREE ROTATIONS

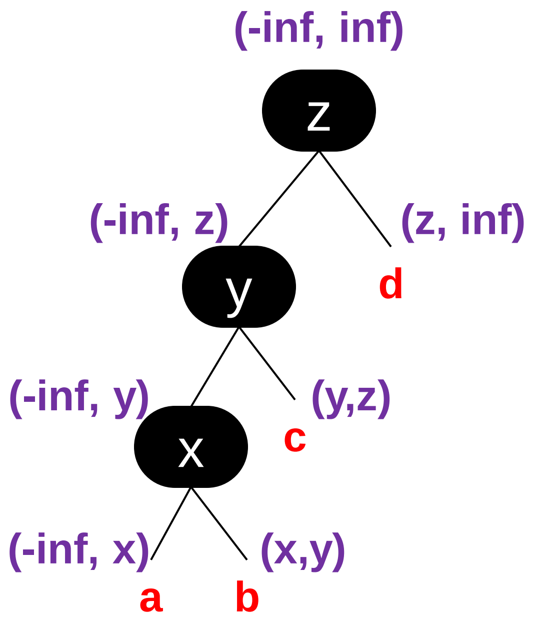
BST Subtree Ranges

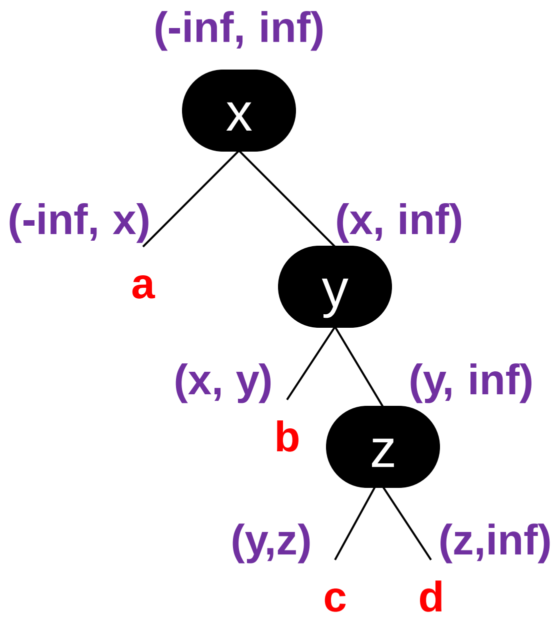
- Consider a binary search tree, what range of values could be in the subtree rooted at each node
  - At the root, any value could be in the "subtree"
  - At the first left child?
  - At the first right child?



BST Subtree Ranges

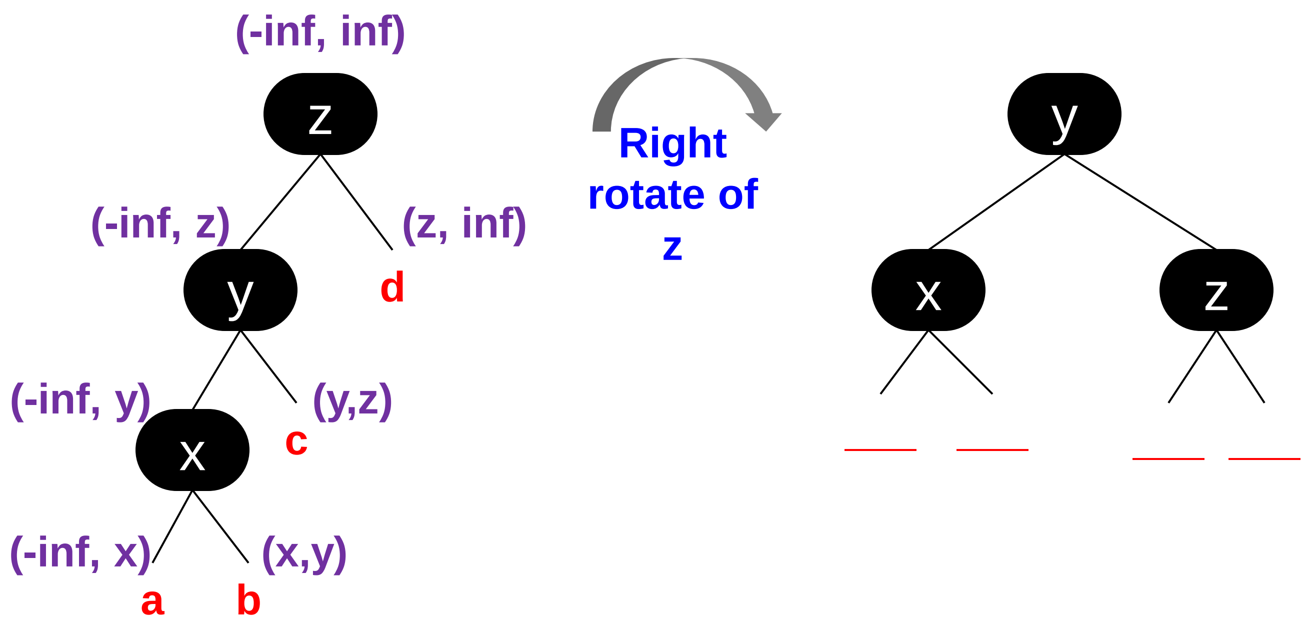
- Consider a binary search tree, what range of values could be in the subtree rooted at each node
  - At the root, any value could be in the "subtree"
  - At the first left child?
  - At the first right child?





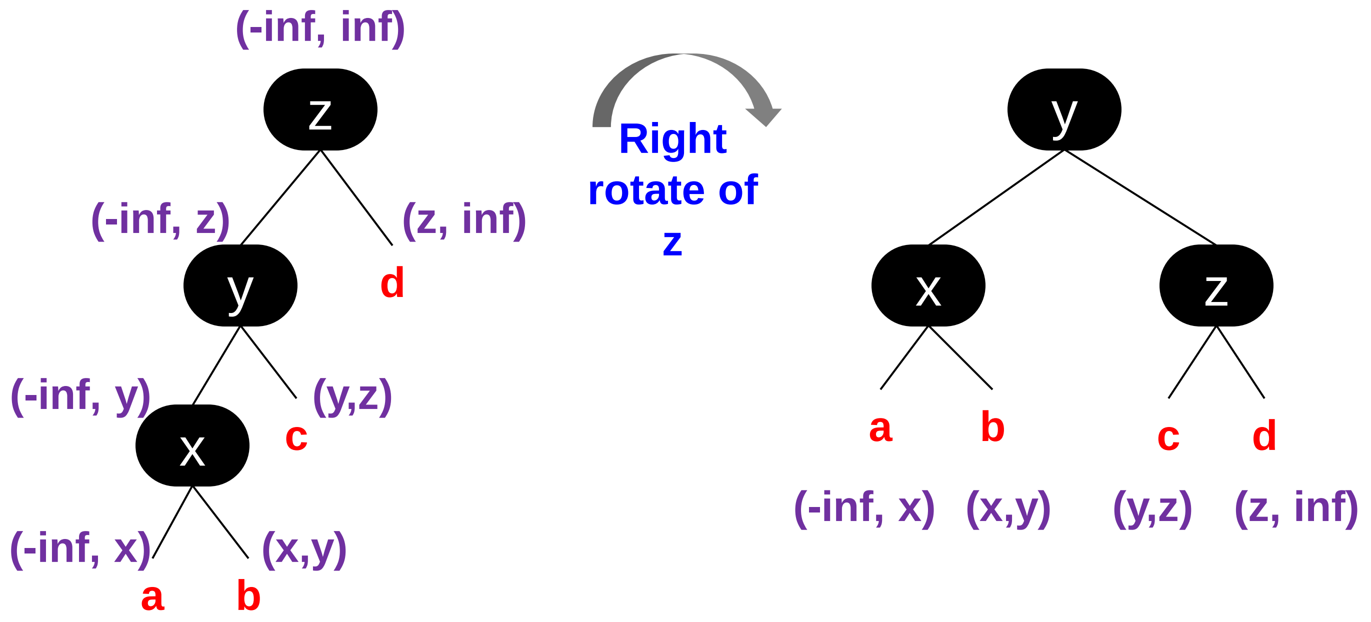
Right Rotation

- Define a right rotation as taking a left child, making it the parent and making the original parent the new right child
- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it…



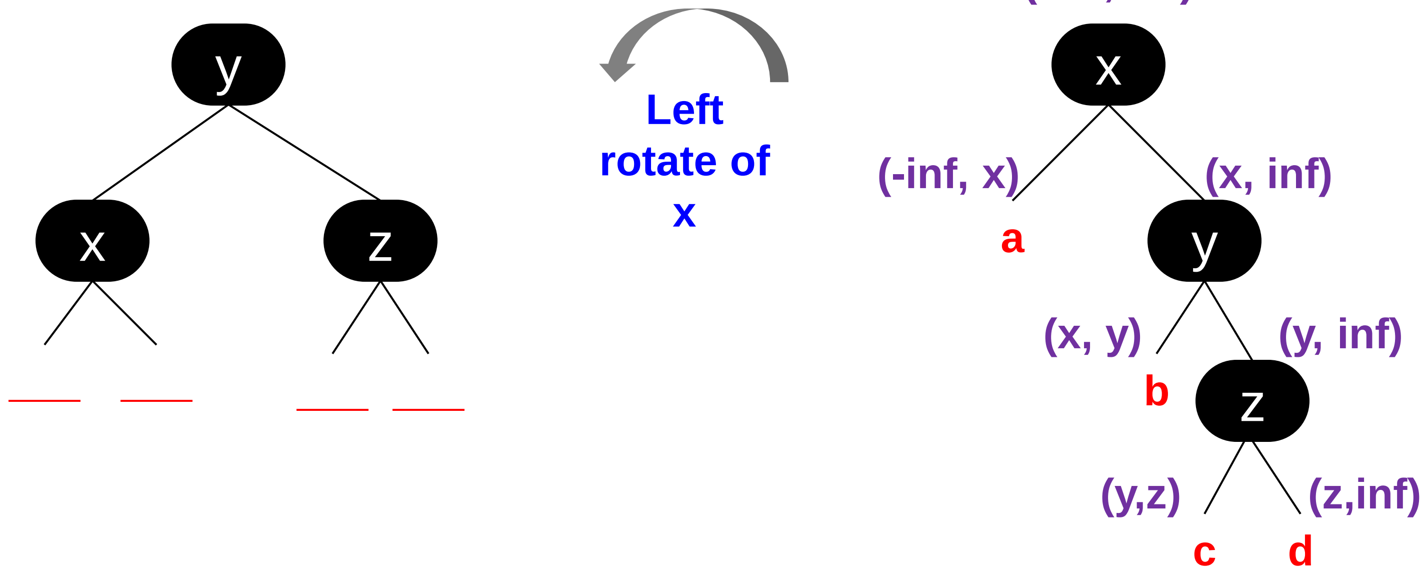
Right Rotation

- Define a right rotation as taking a left child, making it the parent and making the original parent the new right child
- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it…



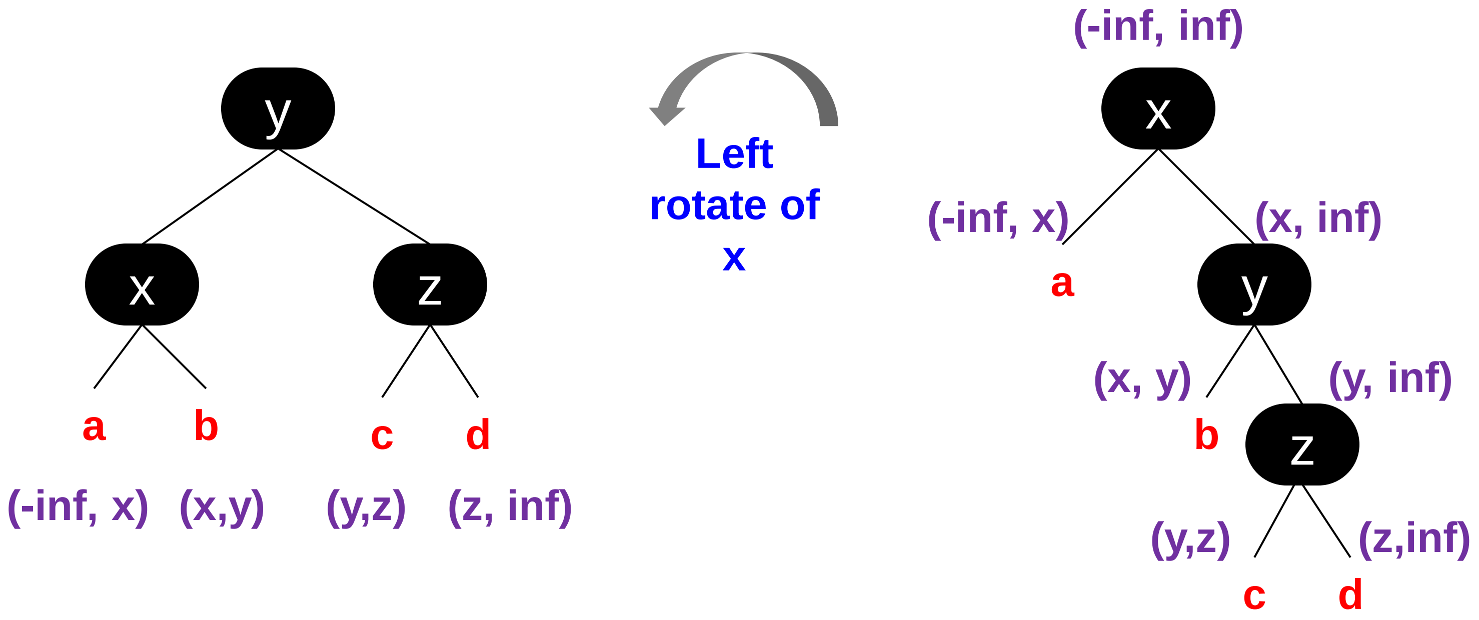
Left Rotation

- Define a left rotation as taking a right child, making it the parent and making the original parent the new left child
- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it…



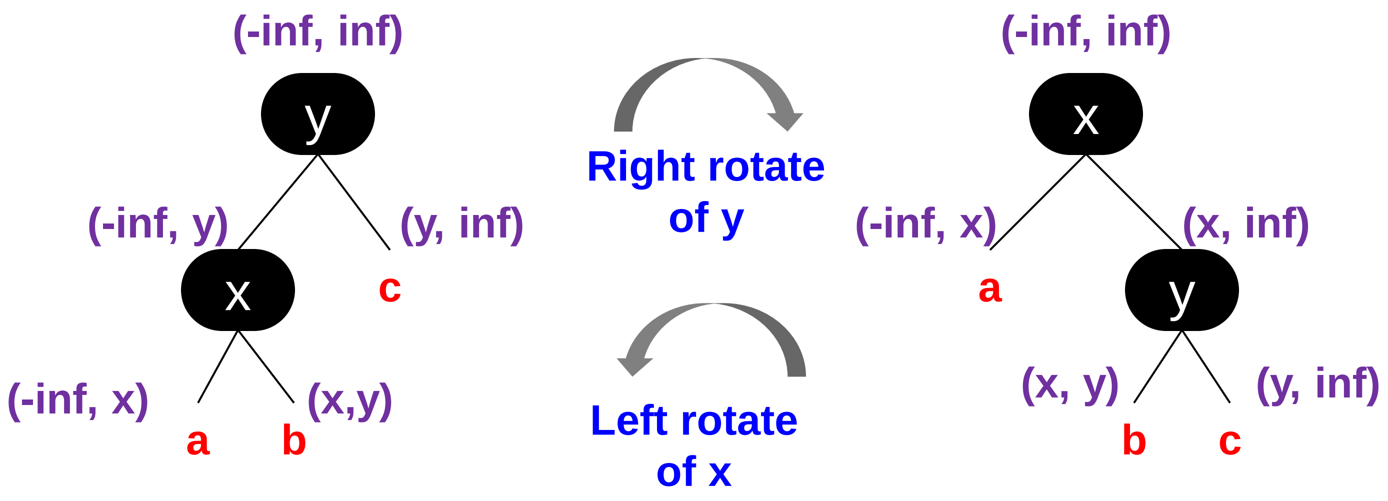
Left Rotation

- Define a left rotation as taking a right child, making it the parent and making the original parent the new left child
- Where do subtrees a, b, c and d belong?
  - Use their ranges to reason about it…



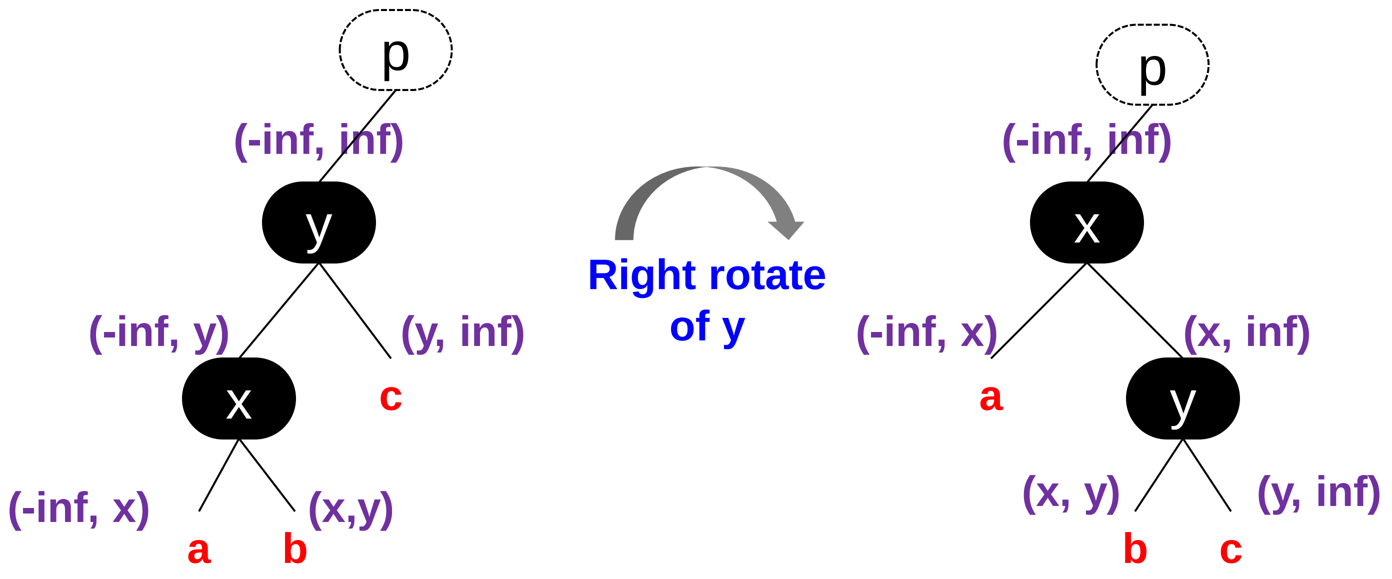
Rotations

- Define a right rotation as taking a left child, making it the parent and making the original parent the new right child
- Where do subtrees a, b, and c belong?
  - Use their ranges to reason about it…



Implementing Rotations

- Take a moment and identify how many and which pointers need to be updated to perform the below right rotation

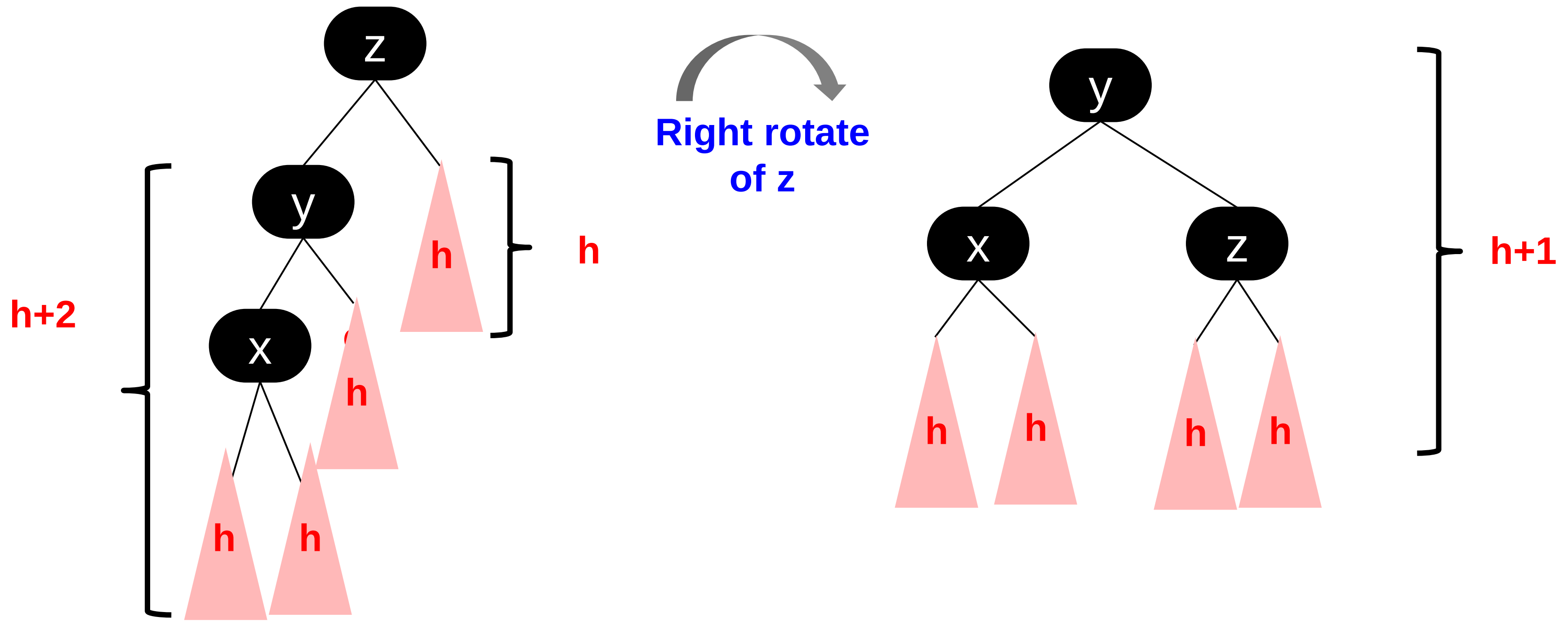


1.
2.
3.
4.

…

Rotation's Effect on Height

- When we rotate, it serves to re-balance the tree



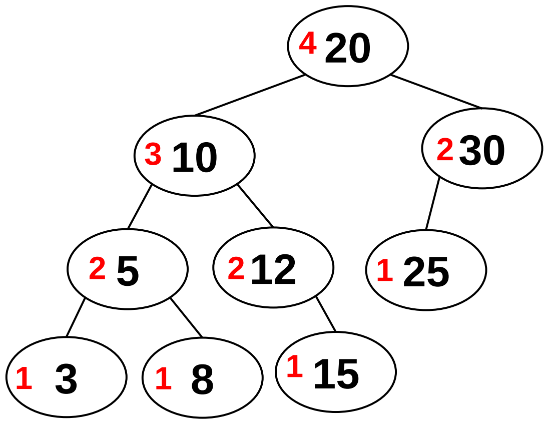
Let's always **specify the parent node** involved in a rotation (i.e. the node that is going to move **DOWN**).

Self-balancing tree proposed by Adelson-Velsky and Landis

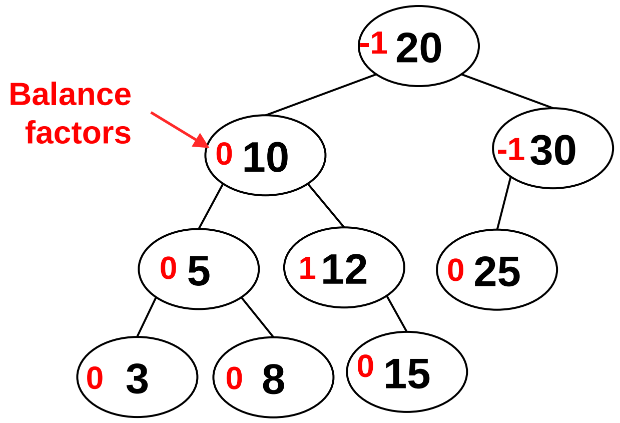
AVL TREES

AVL Trees

- A binary search tree where the **height difference** between left and right subtrees of a node is **at most 1**
  - Binary Search Tree (BST): Left subtree keys are less than the root and right subtree keys are greater
- Two implementations:
  - Height: Just store the height of the tree rooted at that node
  - Balance: Define b(n) as the balance of a node = Right – Left Subtree Height
    - Legal values are -1, 0, 1
    - Balances require at most 2-bits if we are trying to save memory.
    - **Let's use balance for this lecture.**



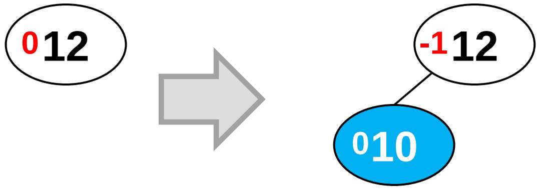
AVL Tree storing Heights



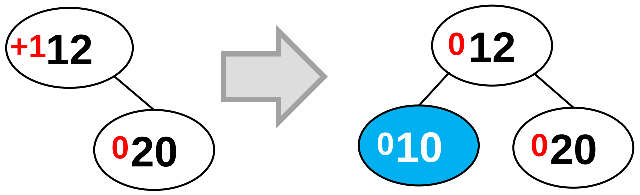
AVL Tree storing balances

Adding a New Node

- Once a new node is added, can its parent be out of balance?
  - No, assuming the tree is "in-balance" when we start.
  - Thus, our parent has to have
    - A balance of 0
    - A balance of 1 if we are a new left child or -1 if a new right child
  - Otherwise, it would not be our parent or the parent would have been out of balance already
- Cases for a newly inserted LEFT child



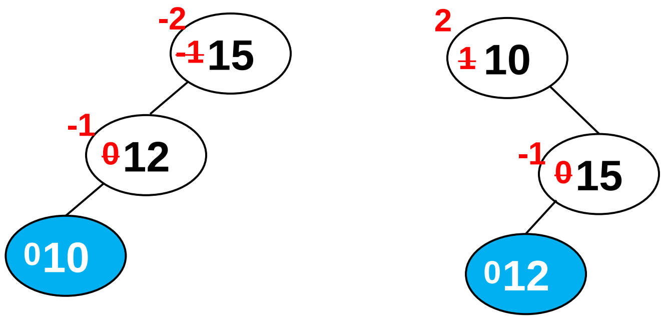
To be a newly inserted LEFT child - Option 1



To be a newly inserted LEFT child - Option 2

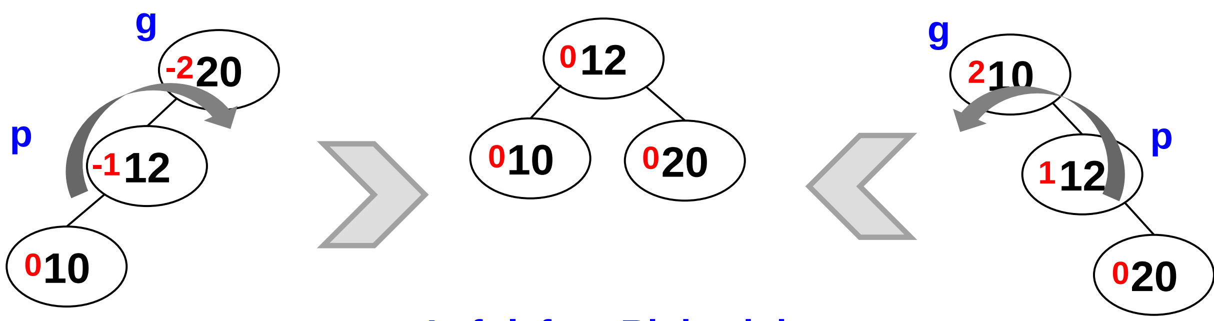
Losing Balance

- If our parent is NOT out of balance, is it possible our grandparent is out of balance?
- Sure, so we need a way to re-balance it

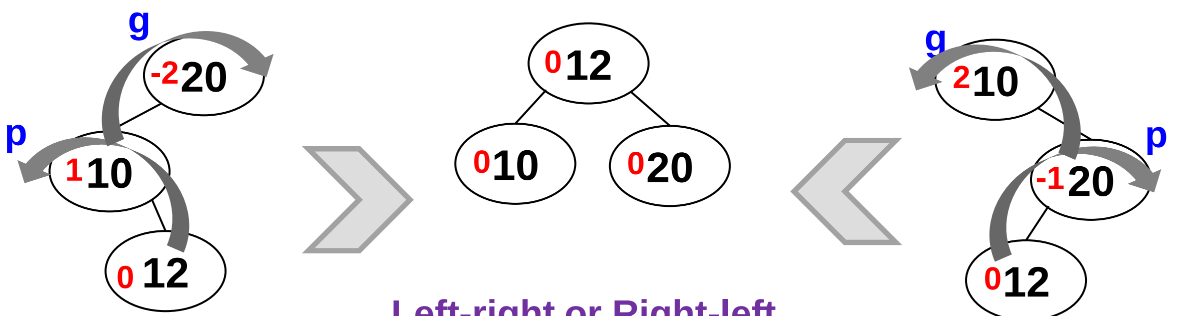


To Zig or Zag

- The rotation(s) required to balance a tree is/are dependent on the grandparent, parent, child relationships
- We can refer to these as the zig-zig (left-left or right-right) case and zig-zag case (left-right or right-left)
- Zig-zig requires 1 rotation
- Zig-zag requires 2 rotations (first converts to zig-zig)



Left-left or Right-right
(a.k.a. Zig-zig)
[Single left/right rotation at grandparent]



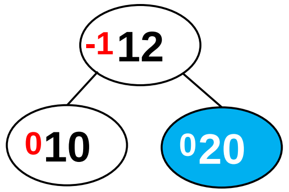
Left-right or Right-left
(a.k.a. Zig-zag)
[Left/right rotation at parent followed by rotation in opposite direction at grandparent]

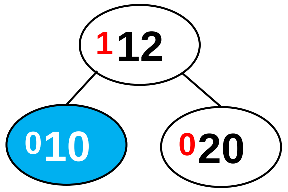
Disclaimer

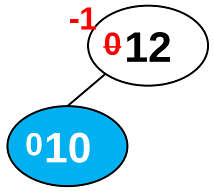
- There are many ways to structure an implementation of an AVL tree…the following slides represent just 1
  – Focus on the bigger picture ideas as that will allow you to more easily understand other implementations

Insert(n)

- If empty tree => set n as root, b(n) = 0, done!
- Else insert n (by walking the tree to a leaf, p, and inserting the new node as its child), set balance to 0, and look at its parent, p
  - If b(p) was -1, then b(p) = 0. Done!
  - If b(p) was +1, then b(p) = 0. Done!
  - If b(p) was 0, then update b(p) and call insert-fix(p, n)







Insert-fix(p, n)

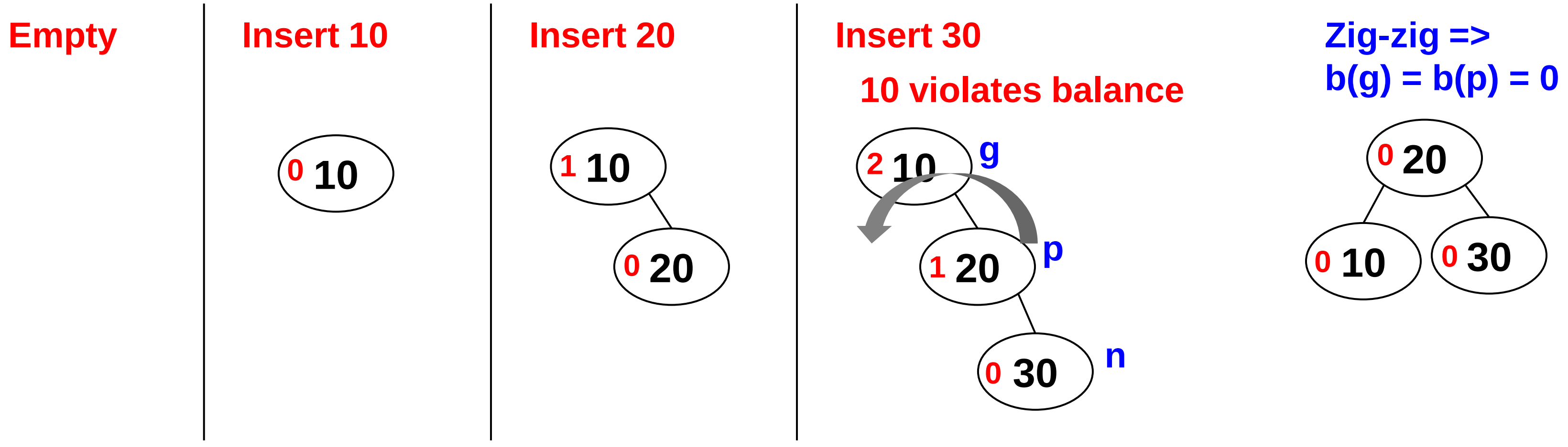
- **Precondition**: p and n are balanced: {-1,0,-1}
- **Postcondition**: g, p, and n are balanced: {-1,0,-1}
- If p is null or `parent(p)` is null, return
- Let g `= parent(p)`
- Assume p is left child of g [For right child swap left/right, +/-]
  - `b(g) += -1 //` Update g's balance to new accurate value for now
  - `Case 1: b(g) == 0, return`
  - `Case 2: b(g) == -1, insertFix(g, p) // recurse`
  - `Case 3: b(g) == -2`
    - If zig-zig then `rotateRight(g); b(p) = b(g) = 0`
    - If zig-zag then `rotateLeft(p); rotateRight(g);`
      - Case 3a: `b(n) == -1 then b(p) = 0; b(g) = +1; b(n) = 0;`
      - Case 3b: `b(n) ==  0 then b(p) = 0; b(g) =  0; b(n) = 0;`
      - Case 3c: `b(n) == +1 then b(p)= -1; b(g) =  0; b(n) = 0;`

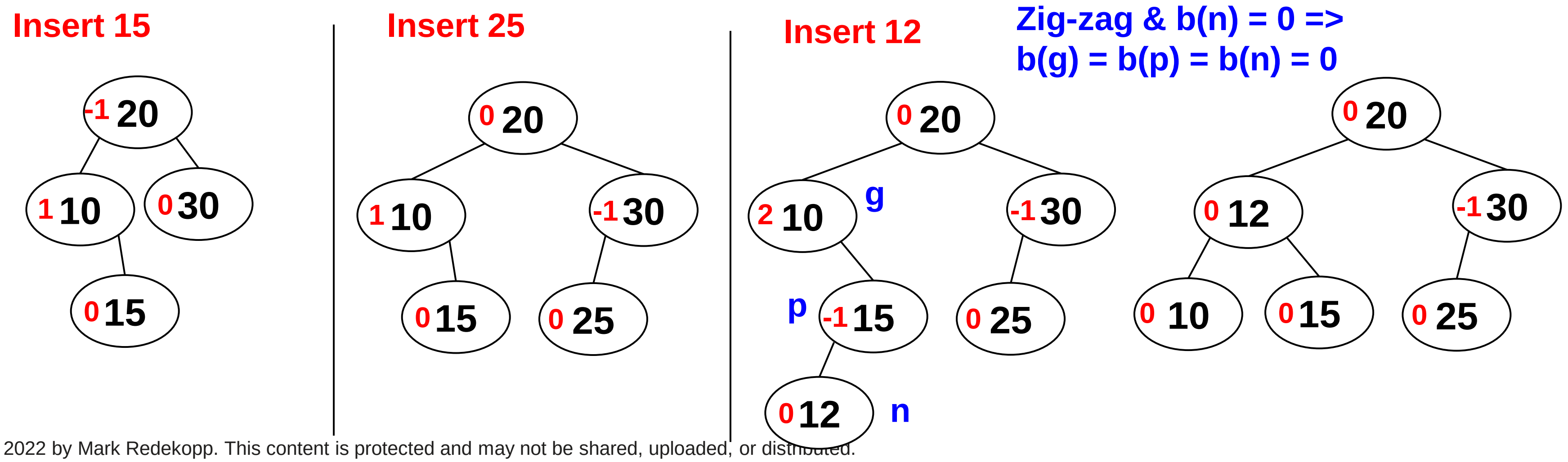
General Idea: Work up ancestor chain updating balances of the ancestor chain or fix a node that is out of balance.

Note: If you perform a rotation to fix a node that is out of balance you will NOT need to recurse. You are done!

Insertion

- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

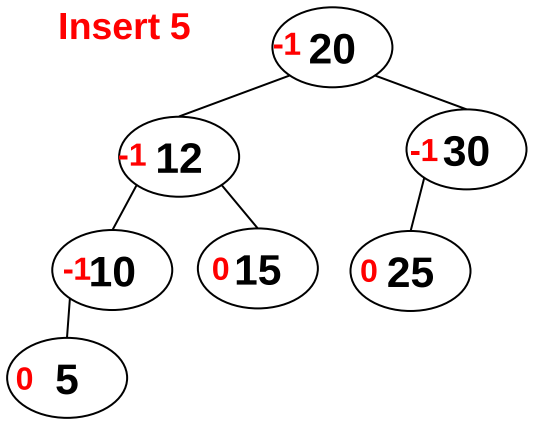




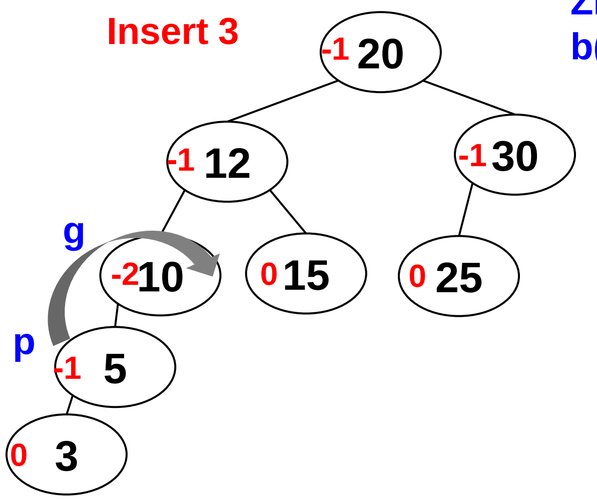
Insertion

- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8

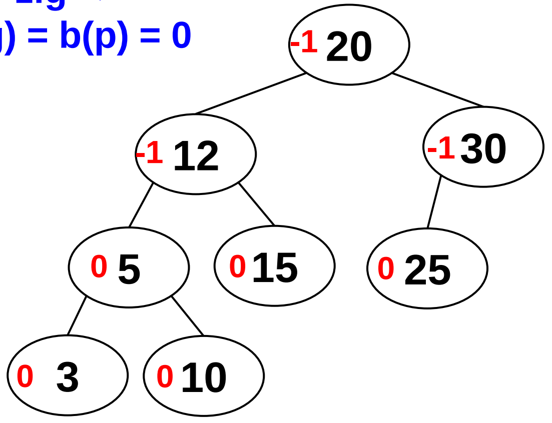
Insert 5



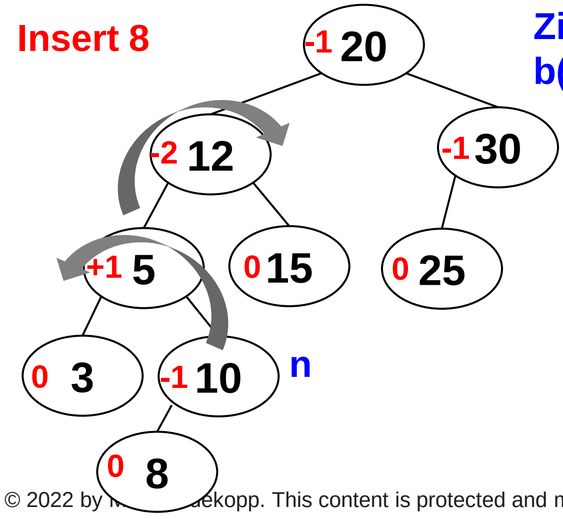
Insert 3



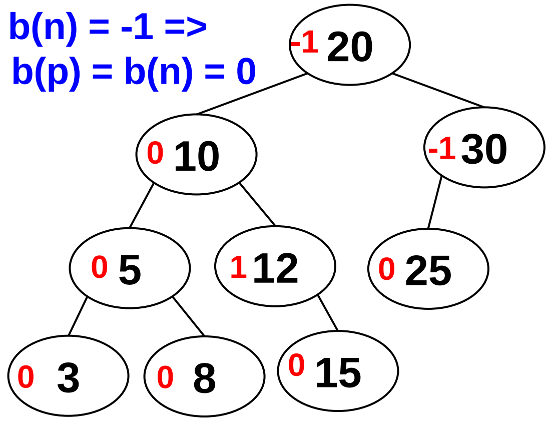
Zig-zig =>
b(g) = b(p) = 0



Insert 8

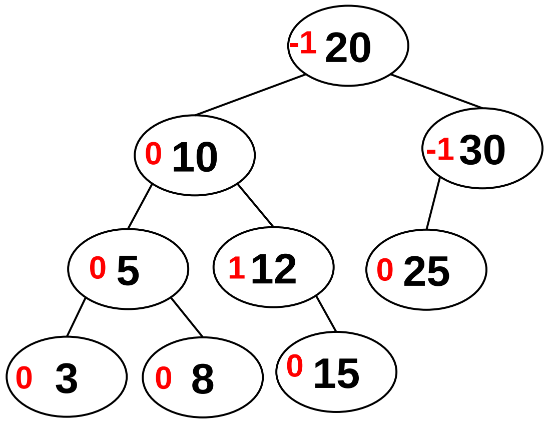


Zig-zag & b(n) = -1 =>
b(g) = 1 & b(p) = b(n) = 0



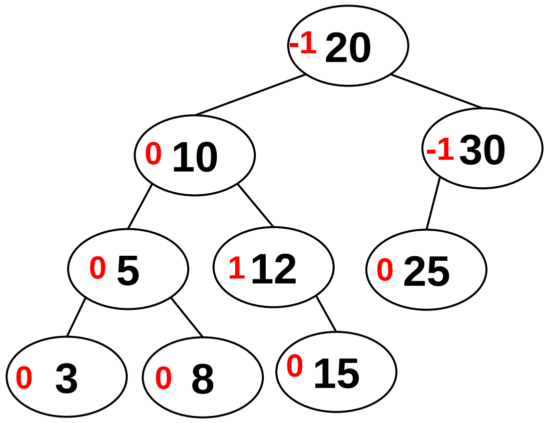
Insertion Exercise 1

- Insert key=28



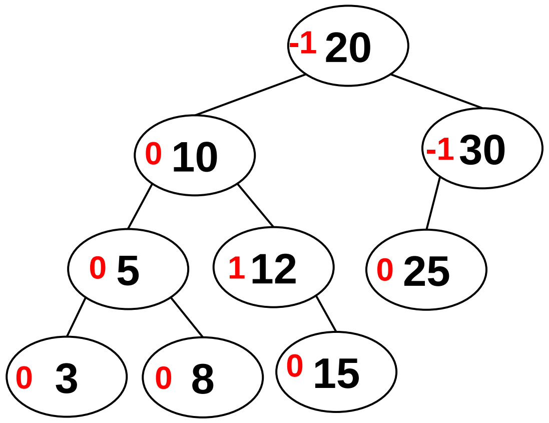
Insertion Exercise 2

- Insert key=17



Insertion Exercise 3

- Insert key=2



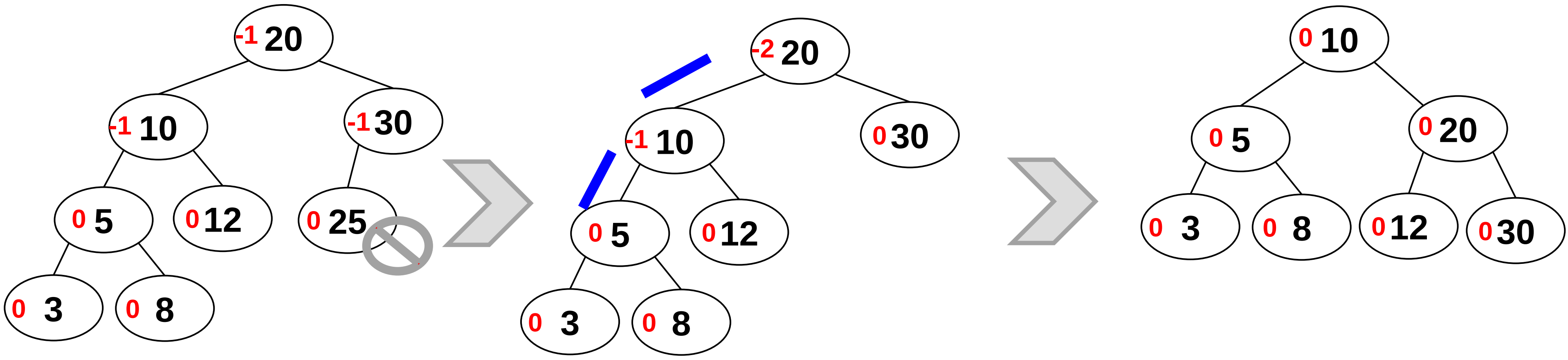
Remove Operation

- Remove operations may also require rebalancing via rotations
- The key idea is to update the balance of the nodes on the ancestor pathway
- If an ancestor gets out of balance then perform rotations to rebalance
  - Unlike insert, performing rotations during removal does not mean you are done, but need to continue recursing
- There are slightly more cases to worry about but not too many more

Removal: A First Look

- Let's try removal just by intuition…
  - Walk up ancestor chain updating balances
  - Fix any out-of-balance node by performing rotations

Remove 25



Update Balances

Perform rotations using "taller" of children

Remove

- Find node, n, to remove by walking the tree
- If n has 2 children, swap positions with in-order **successor** (or **predecessor**) and perform the next step
  – Recall if a node has 2 children we swap with its **successor or predecessor** who can have at most 1 child and then remove that node
- Let `p = parent(n)`
- If `p` is not NULL,
  – If n is a left child, let `diff = +1`
    – If n is a left child to be removed, the right subtree now has greater height, so add diff = +1 to balance of its parent
  – if n is a right child, let `diff = -1`
    – If n is a right child to be removed, the left subtree now has greater height, so add diff = -1 to balance of its parent
  – `diff` *will be the amount* ***added*** *to updated the balance of p*
- Delete n and update pointers
- "Patch tree" by calling `removeFix(p, diff);`

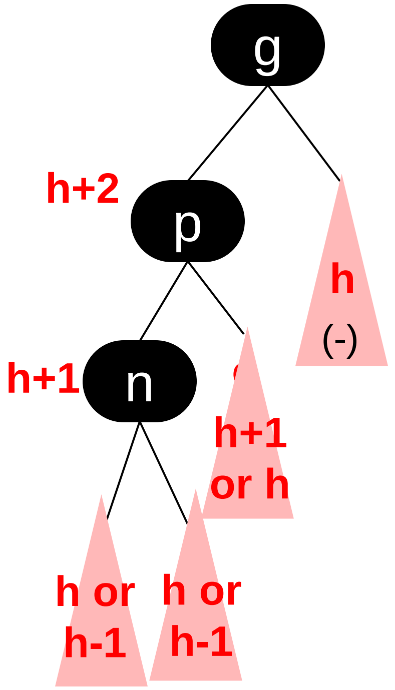
USC Viterbi School of Engineering

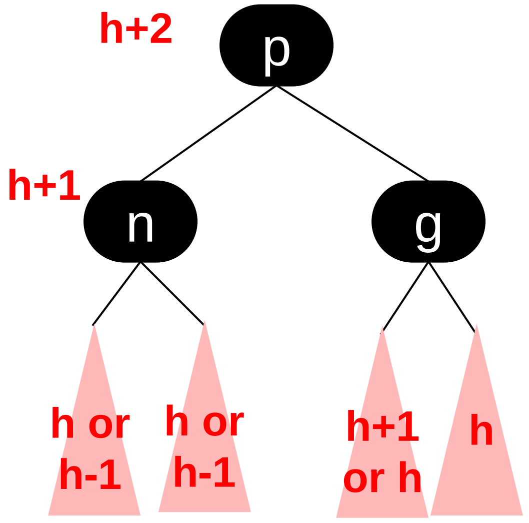
RemoveFix(n, diff)

- If n is null, return
- Compute next recursive call's arguments now before altering the tree
  - Let p = parent(n) and if p is not NULL let ndiff (nextdiff) = +1 if n is a left child and -1 otherwise
- Assume diff = -1 and follow the remainder of this approach, mirroring if diff = +1
- `Case 1: b(n) + diff == -2`
  - [Perform the check for the mirror case where b(n) + diff == +2, flipping left/right and -1/+1]
  - Let c = left(n), the taller of the children
  - `Case 1a: b(c) == -1   // zig-zig case`
    - `rotateRight(n), b(n) = b(c) = 0, removeFix(p, ndiff)`
  - `Case 1b: b(c) ==  0   // zig-zig case`
    - `rotateRight(n), b(n) = -1, b(c) = +1 // Done!`
  - `Case 1c: b(c) == +1   // zig-zag case`
    - `Let g = right(c)`
    - `rotateLeft(c) then rotateRight(n)`
    - `If b(g) was +1 then b(n) = 0,  b(c) = -1, b(g) = 0`
    - `If b(g) was  0 then b(n) = 0,  b(c) =  0, b(g) = 0`
    - `If b(g) was -1 then b(n) = +1, b(c) =  0, b(g) = 0`
    - `removeFix(p, ndiff);`
- `Case 2: b(n) + diff == -1: then b(n) = -1; // Done!`
- `Case 3: b(n) + diff ==  0: then b(n) = 0, removeFix(p, ndiff)`

Note:
p = parent of n
n = current node
c = taller child of n
g = grandchild of n

Why this Works (Zig-zig version)

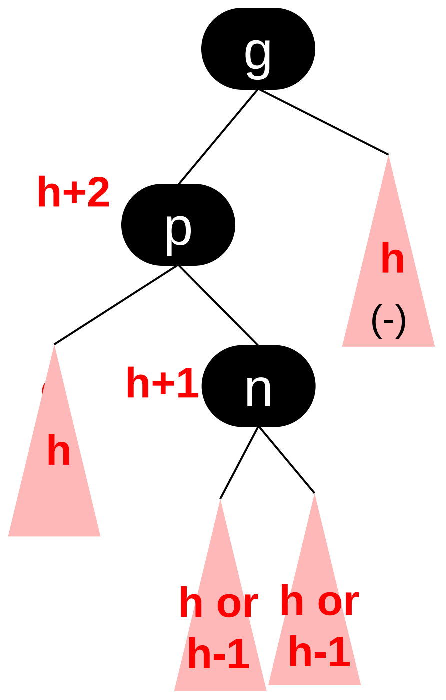


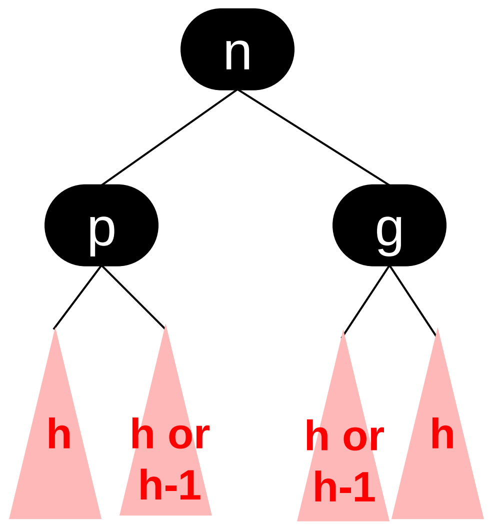


This is symmetrical if p and n are right children.

Note the change in height of the tree, thus the necessity to continue calling removeFix

Why this Works (Zig-zag version)



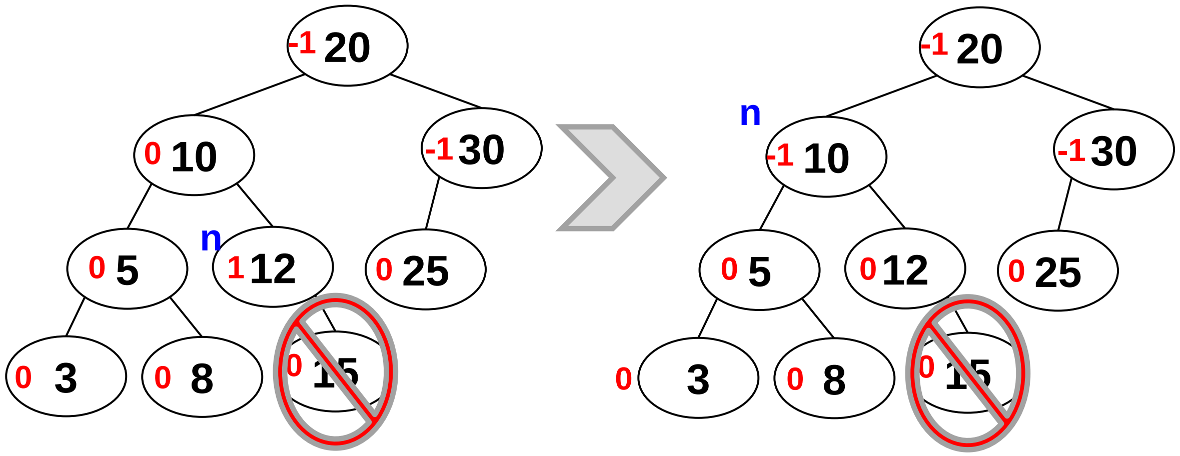


This is symmetrical if p is a right child while n is a left child.

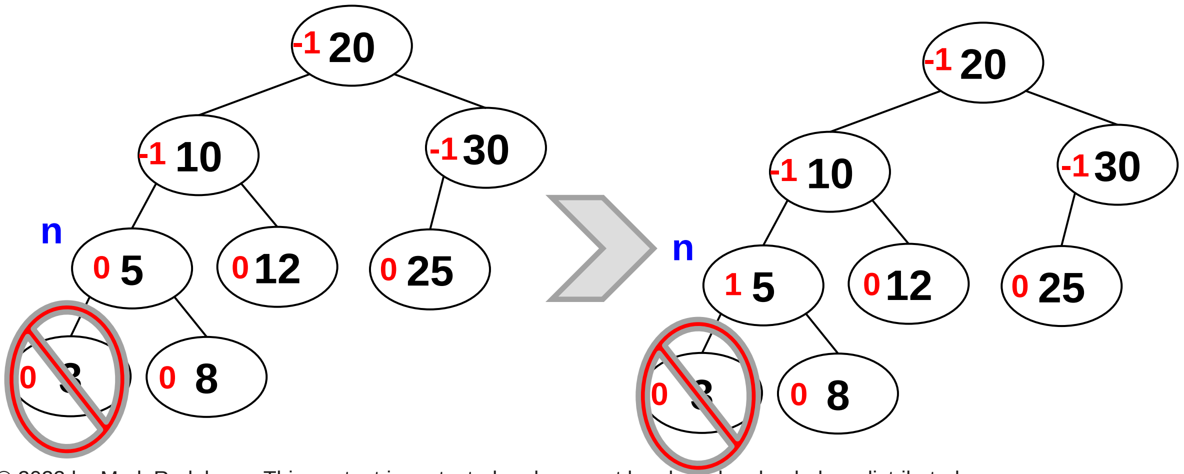
Note the change in height of the tree, thus the necessity to continue calling removeFix

Remove Examples

Remove 15

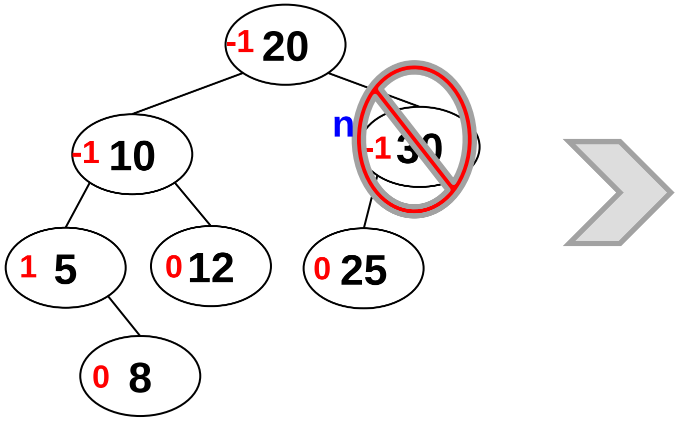


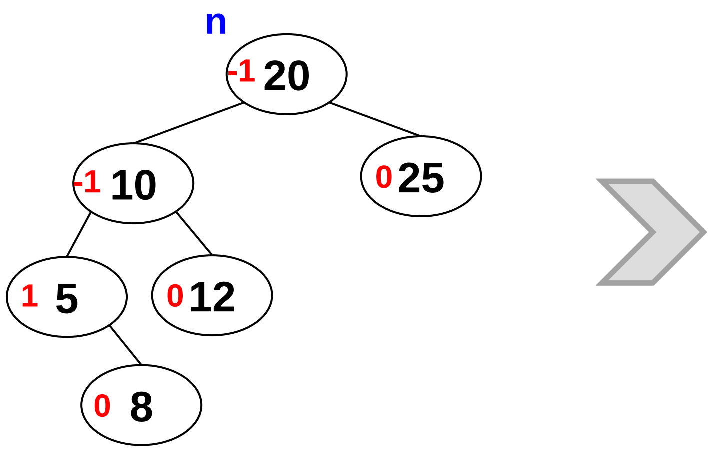
Remove 3



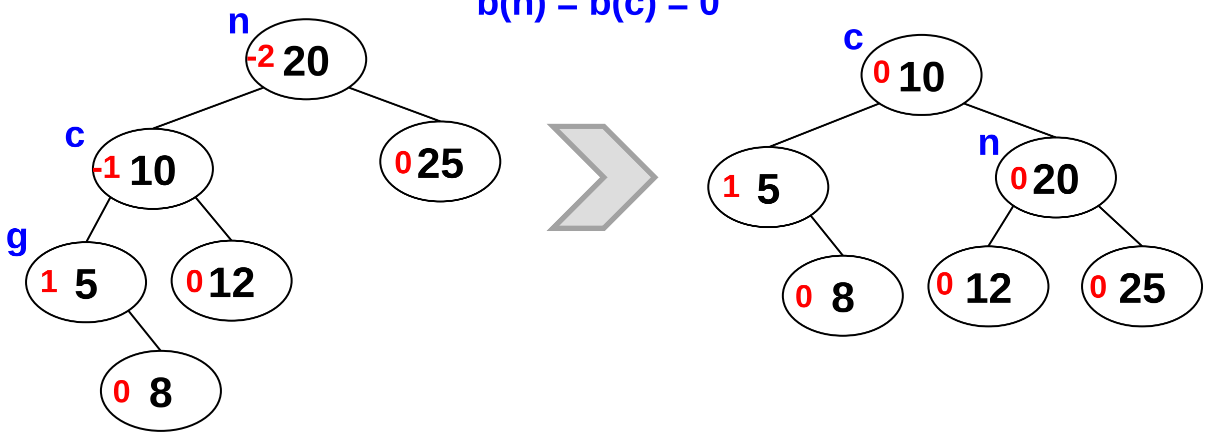
Remove Examples

Remove 30



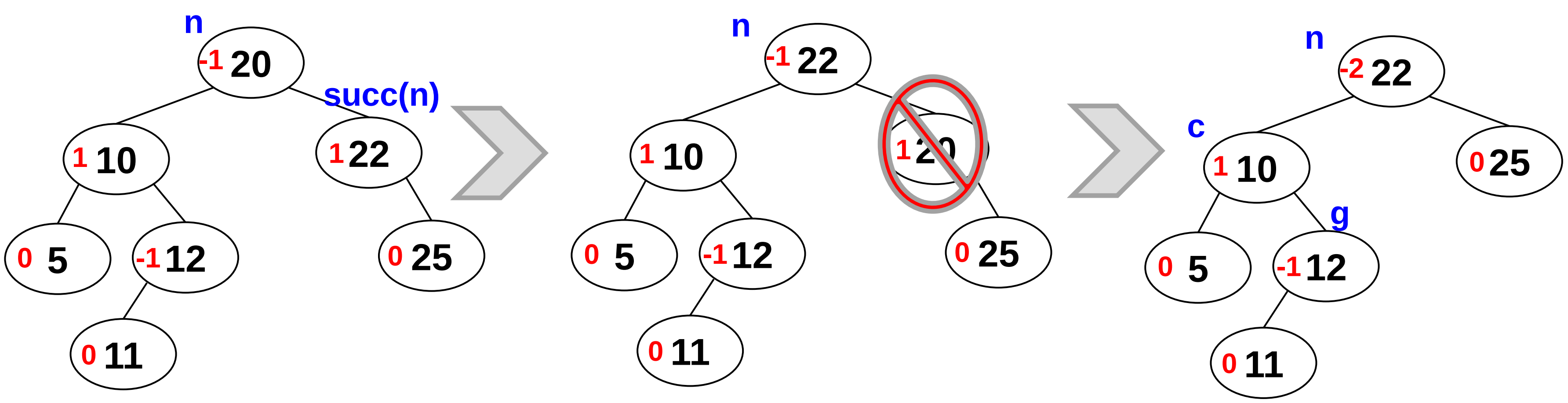


Zig-zig & b(c) = -1 =>
b(n) = b(c) = 0

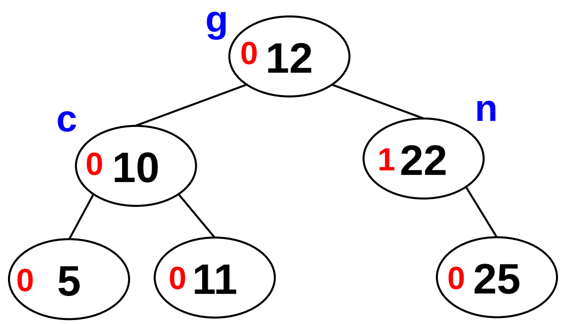


Remove Examples

Remove 20

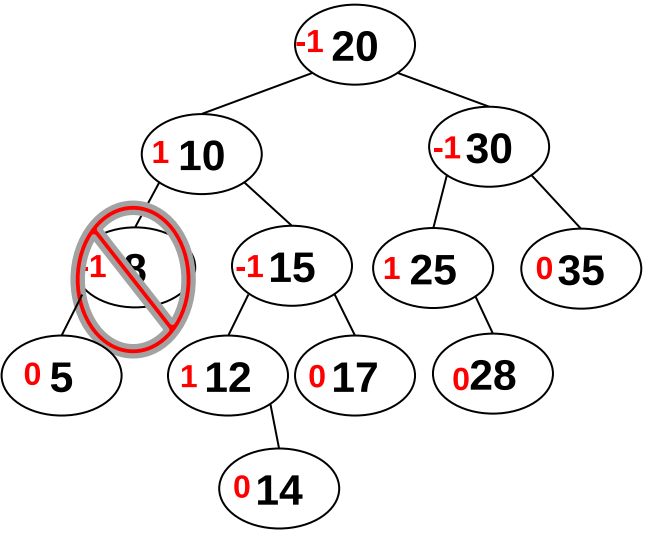


Zig-zag & b(g) = -1 =>
b(n) = +1, b(c) = 0, b(g) = 0



Remove Example 1

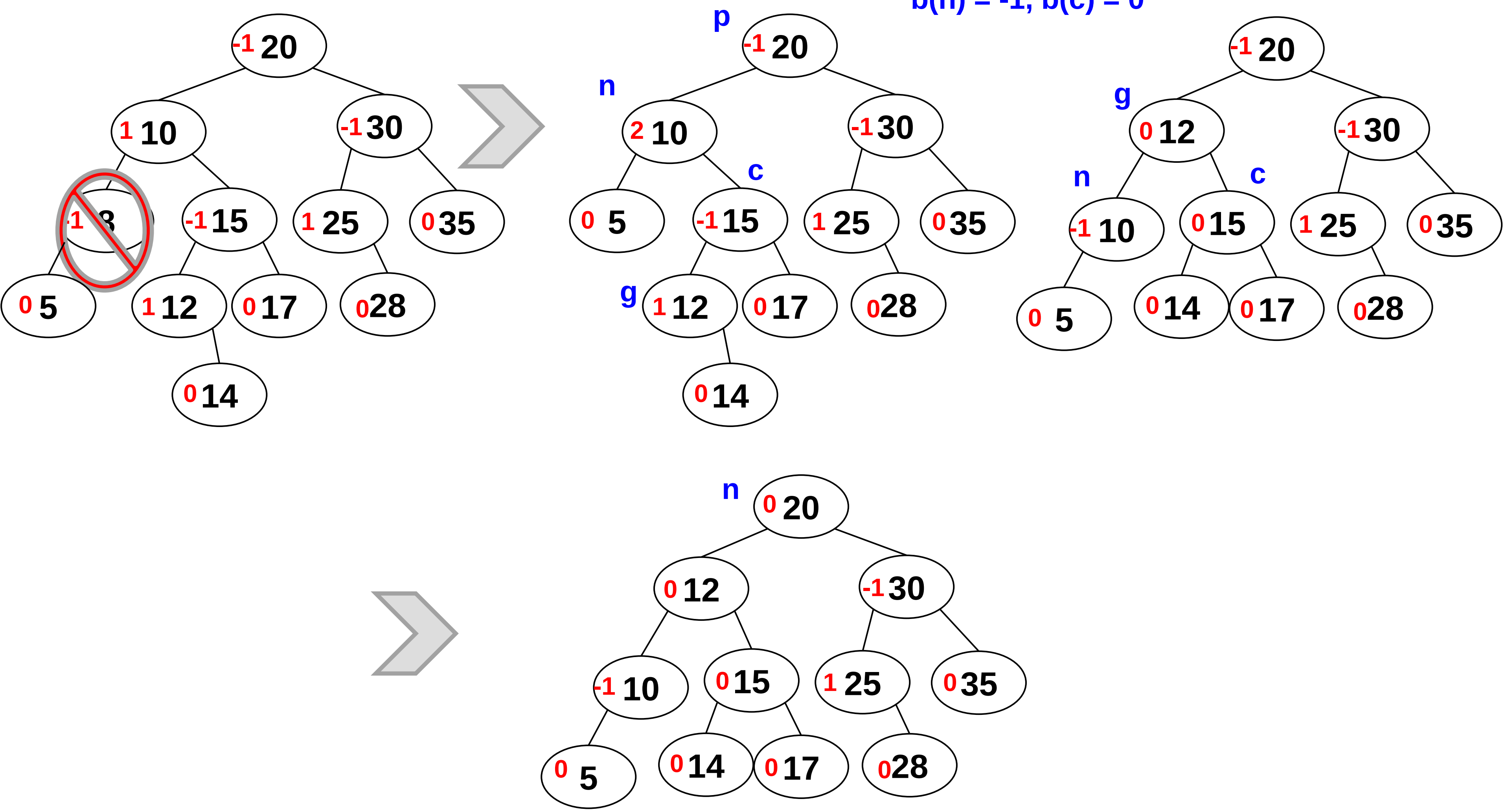
Remove 8



Remove Example 1

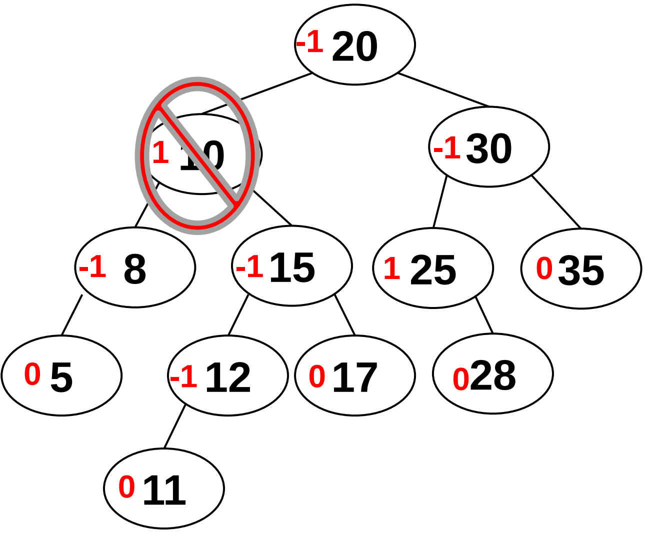
Remove 8

Zig-zag & b(g) = 0 => b(n) = -1, b(c) = 0



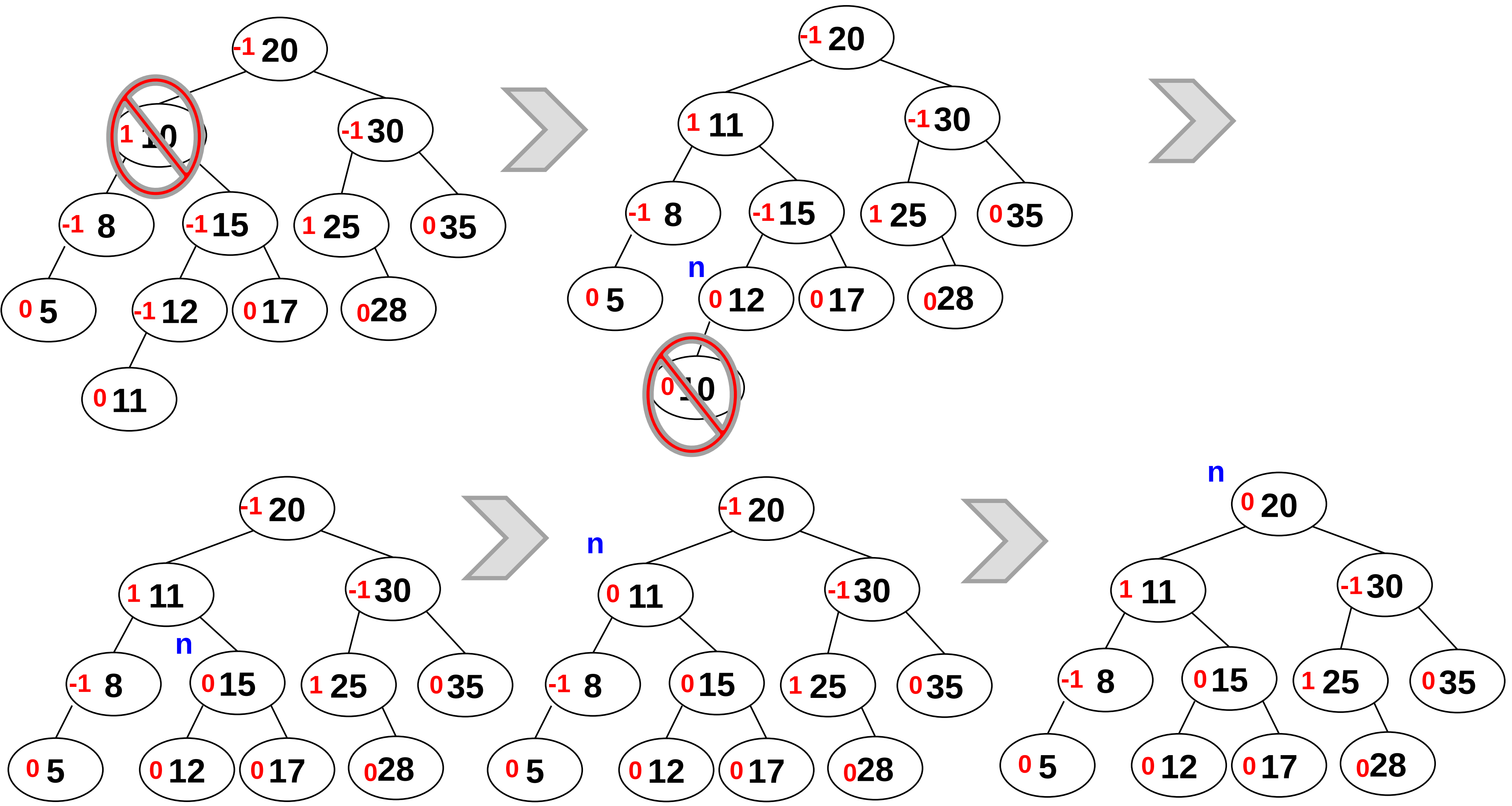
Remove Example 2

Remove 10



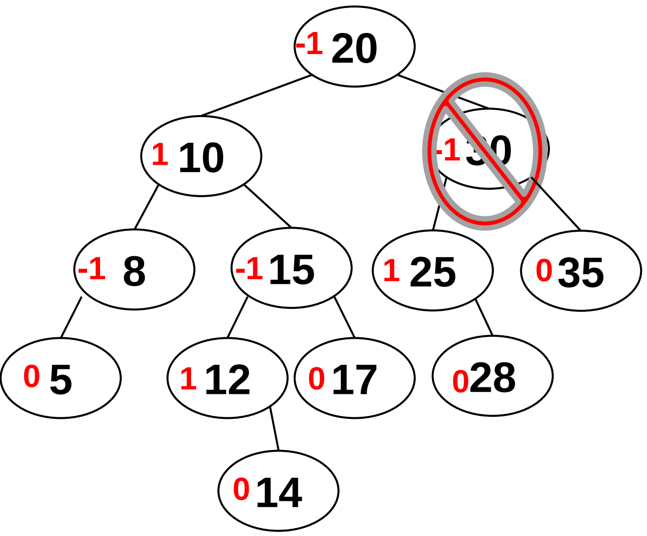
Remove Example 2

Remove 10



Remove Example 3

Remove 30

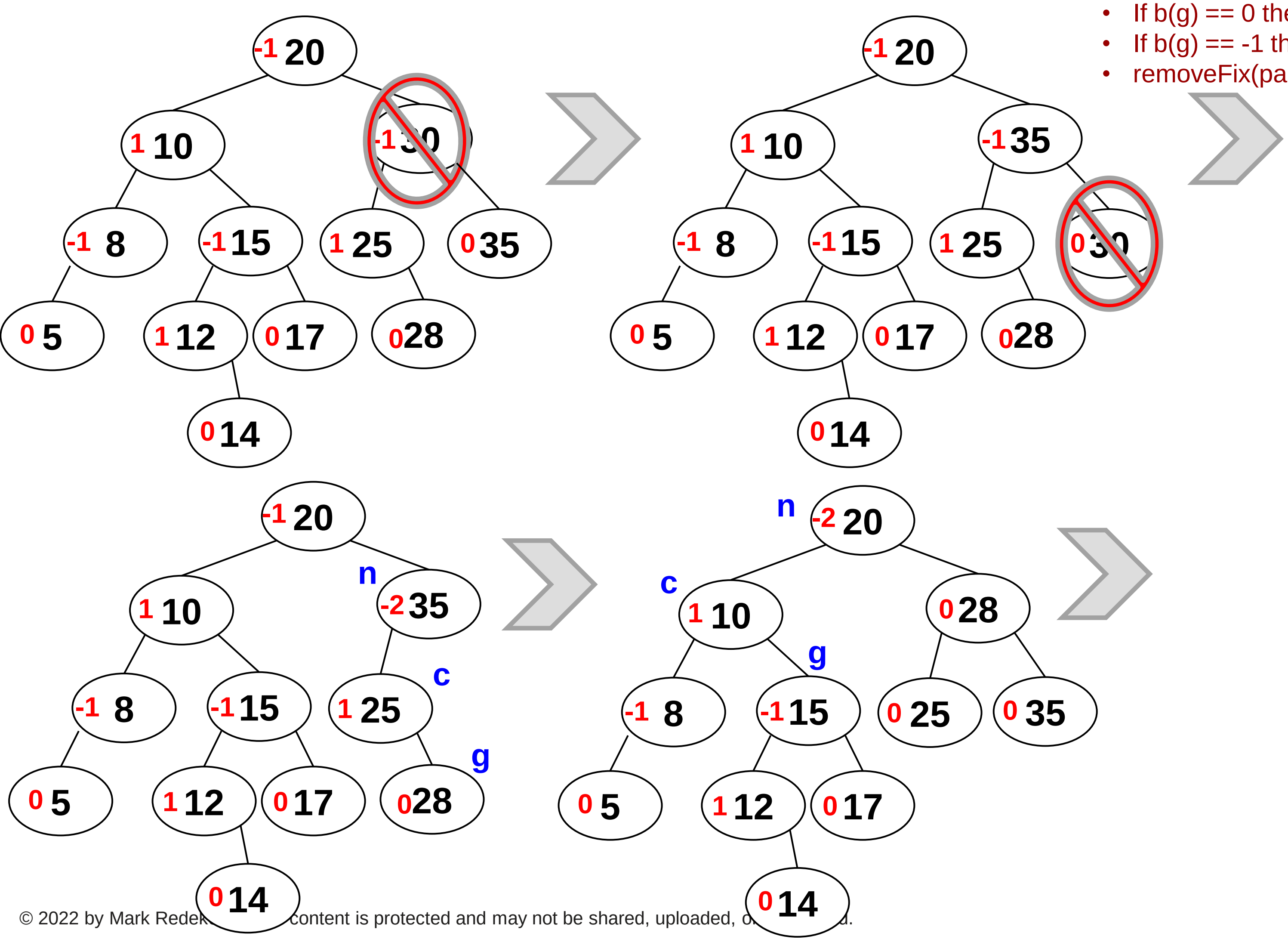


Remove Example 3

else if b(c) == 1 (zig-zag case)

- rotateLeft(c) then rotateRight(n)
- Let g = right(c), b(g) = 0
- If b(g) == +1 then b(n) = 0, b(c) = -1, b(g) = 0
- If b(g) == 0 then b(n) = b(c) = 0, b(g) = 0
- If b(g) == -1 then b(n) = +1, b(c) = 0, b(g) = 0
- removeFix(parent(p), ndiff);

Remove 30

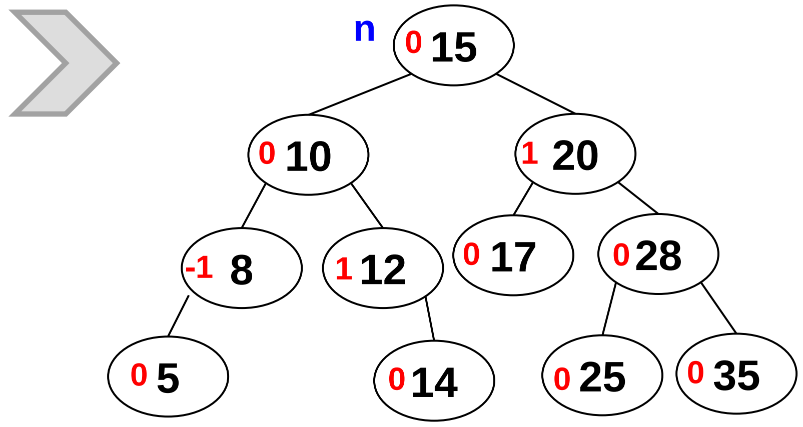


Remove Example 3 (cont)

else if b(c) == 1 (zig-zag case)

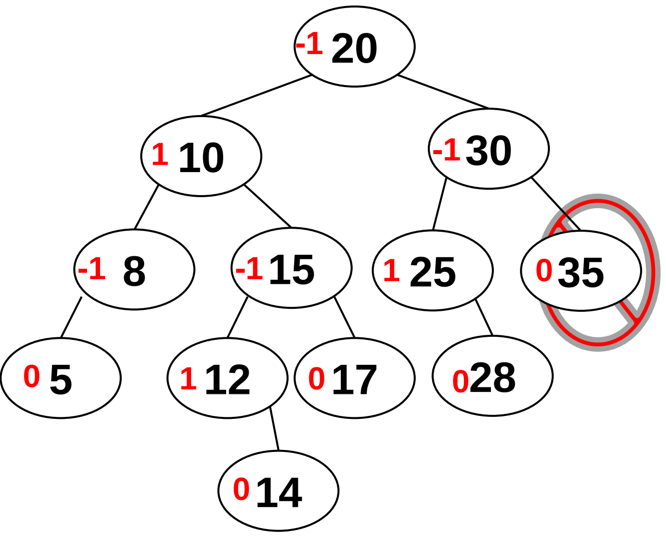
- rotateLeft(c) then rotateRight(n)
- Let g = right(c), b(g) = 0
- If b(g) == +1 then b(n) = 0, b(c) = -1, b(g) = 0
- If b(g) == 0 then b(n) = b(c) = 0, b(g) = 0
- If b(g) == -1 then b(n) = +1, b(c) = 0, b(g) = 0
- removeFix(parent(p), ndiff);

Remove 30 (cont.)



Remove Exercise

Remove 35



Online Tool

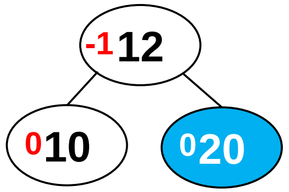
- https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

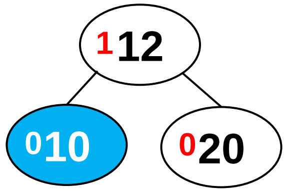
Distribute these 4 to students

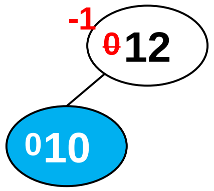
FOR PRINT

Insert(n)

- If empty tree => set n as root, b(n) = 0, done!
- Else insert n (by walking the tree to a leaf, p, and inserting the new node as its child), set balance to 0, and look at its parent, p
  - If b(p) was -1, then b(p) = 0. Done!
  - If b(p) was +1, then b(p) = 0. Done!
  - If b(p) was 0, then update b(p) and call insert-fix(p, n)







Insert-fix(p, n)

- **Precondition**: p and n are balanced: {-1,0,-1}
- **Postcondition**: g, p, and n are balanced: {-1,0,-1}
- If p is null or `parent(p)` is null, return
- Let g `= parent(p)`
- Assume p is left child of g [For right child swap left/right, +/-]
  - `b(g) += -1 //` Update g's balance to new accurate value for now
  - `Case 1: b(g) == 0, return`
  - `Case 2: b(g) == -1, insertFix(g, p) // recurse`
  - `Case 3: b(g) == -2`
    - If zig-zig then `rotateRight(g); b(p) = b(g) = 0`
    - If zig-zag then `rotateLeft(p); rotateRight(g);`
      - Case 3a: `b(n) == -1 then b(p) = 0; b(g) = +1; b(n) = 0;`
      - Case 3b: `b(n) ==  0 then b(p) = 0; b(g) =  0; b(n) = 0;`
      - Case 3c: `b(n) == +1 then b(p)= -1; b(g) =  0; b(n) = 0;`

General Idea: Work up ancestor chain updating balances of the ancestor chain or fix a node that is out of balance.

Note: If you perform a rotation to fix a node that is out of balance you will NOT need to recurse. You are done!

Remove

- Find node, n, to remove by walking the tree
- If n has 2 children, swap positions with in-order **successor** (or **predecessor**) and perform the next step
  - Recall if a node has 2 children we swap with its **successor or predecessor** who can have at most 1 child and then remove that node
- Let `p = parent(n)`
- If `p` is not NULL,
  - If n is a left child, let `diff = +1`
    - If n is a left child to be removed, the right subtree now has greater height, so add diff = +1 to balance of its parent
  - if n is a right child, let `diff = -1`
    - If n is a right child to be removed, the left subtree now has greater height, so add diff = -1 to balance of its parent
  - `diff` *will be the amount* ***added*** *to updated the balance of p*
- Delete n and update pointers
- "Patch tree" by calling `removeFix(p, diff);`

RemoveFix(n, diff)

- If n is null, return
- Compute next recursive call's arguments now before altering the tree
  - Let p = parent(n) and if p is not NULL let ndiff (nextdiff) = +1 if n is a left child and -1 otherwise
- Assume diff = -1 and follow the remainder of this approach, mirroring if diff = +1
- `Case 1: b(n) + diff == -2`
  - [Perform the check for the mirror case where b(n) + diff == +2, flipping left/right and -1/+1]
  - Let c = left(n), the taller of the children
  - `Case 1a: b(c) == -1   // zig-zig case`
    - `rotateRight(n), b(n) = b(c) = 0, removeFix(p, ndiff)`
  - `Case 1b: b(c) ==  0   // zig-zig case`
    - `rotateRight(n), b(n) = -1, b(c) = +1 // Done!`
  - `Case 1c: b(c) == +1   // zig-zag case`
    - `Let g = right(c)`
    - `rotateLeft(c) then rotateRight(n)`
    - `If b(g) was +1 then b(n) = 0,  b(c) = -1, b(g) = 0`
    - `If b(g) was  0 then b(n) = 0,  b(c) =  0, b(g) = 0`
    - `If b(g) was -1 then b(n) = +1, b(c) =  0, b(g) = 0`
    - `removeFix(p, ndiff);`
- `Case 2: b(n) + diff == -1: then b(n) = -1; // Done!`
- `Case 3: b(n) + diff ==  0: then b(n) = 0, removeFix(p, ndiff)`

Note:
p = parent of n
n = current node
c = taller child of n
g = grandchild of n

OLD ALTERNATE METHOD

Insert

- Root => set balance, done!
- Insert, v, and look at its parent, p
  - If b(p) = -1, then b(p) = 0. Done!
  - If b(p) = +1, then b(p) = 0. Done!
  - If b(p) = 0, then update b(p) and call insert-fix(p)

Insert-Fix

- For input node, v
  - If v is root, done.
  - Invariant: b(v) = {-1, +1}
- Find p = parent(v) and assume v = left(p) [i.e. left child]
  - If b(p) = 1, then b(p) = 0. Done!
  - If b(p) = 0, then b(p) = -1. Insert-fix(p).
  - If b(p) = -1 and b(v) = -1 (zig-zig), then b(p) = 0, b(v) = 0, rightRotate(p) Done!
  - If b(p) = -1 and b(v) = 1 (zig-zag), then
    - u = right(v), b(u) = 0, leftRotate(n), rightRotate(p)
    - If b(u) = -1, then b(v) = 0, b(p) = 1
    - If b(u) = 1, then b(v) = -1, b(p) = 0
    - Done!

Remove

- Let n = node to remove (perform BST find)
- If n has 2 children, swap positions with in-order **successor** (or predecessor) and perform the next step
  - If you had to swap, let n be the node with the original value that just swapped down to have 0 or 1 children guaranteed
- Let p = parent(n)
- If n is not in the root position (i.e. p is not NULL) determine its relationship with its parent
  - If n is a left child, let diff = +1
  - if n is a right child, let diff = -1
- Delete n and "patch" the tree (update pointers including root)
- removeFix(p, diff);

RemoveFix(n, diff)

- If n is null, return
- Compute next recursive call's arguments now before we alter the tree
  - Let p = parent(n) and if p is not NULL let ndiff = +1 if n is a left child and -1 otherwise
- Assume diff = -1 and follow the remainder of this approach, mirroring if diff = +1
- If (n.balance + diff == -2)
  - [Perform the check for the mirror case where n.balance + diff == +2, flipping left/right and -1/+1]
  - Let c = left(n), the taller of the children
  - If c.balance == -1 or 0 (zig-zig case)
    - rotateRight(n)
    - if c.balance was -1 then n.balance = c.balance = 0, removeFix(p, ndiff)
    - if c.balance was 0 then n.balance = -1, c.balance = +1, done!
  - else if c.balance == 1 (zig-zag case)
    - Let g = right(c)
    - rotateLeft(c) then rotateRight(n)
    - If g.balance was +1 then n.balance = 0, c.balance = -1, g.balance = 0
    - If g.balance was 0 then n.balance = c.balance = 0, g.balance = 0
    - If g.balance was -1 then n.balance = +1, c.balance = 0, g.balance = 0
    - removeFix(p, ndiff);
- else if (n.balance + diff == -1) then n.balance = -1, done!
- else (if n.balance + diff == 0) n.balance = 0, removeFix(p, ndiff)

Note:
p = parent of n
n = current node
c = taller child of n
g = grandchild of n