EC504 ALGORITHMS AND DATA STRUCTURES
FALL 2020 MONDAY & WEDNESDAY
2:30 PM - 4:15 PM

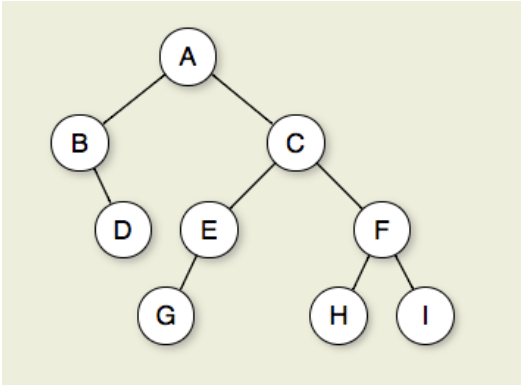
Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

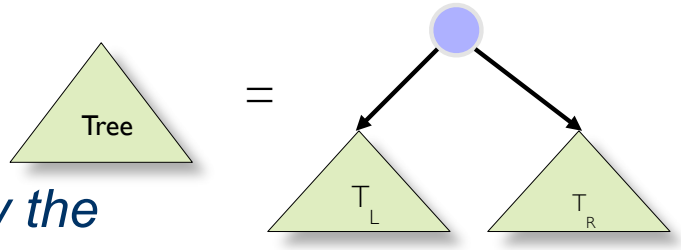
Haoyang Wang: haoyangw@bu.edu

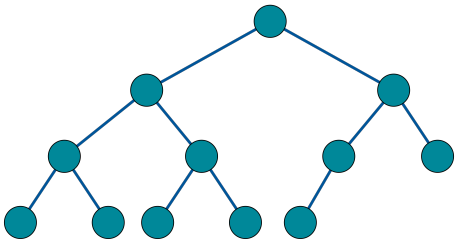
Christopher Liao: cliao25@bu.edu

Special Case: Binary Trees

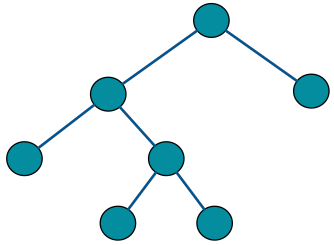


- Binary Tree: Every node has at most 2 children (0, 1 or 2)
- Recursive definition: A binary trees is null, or a singe node with a
  Right and Left Child that is a binary tree!
- Special cases:
  - Full Binary Tree: Every node has 0 or 2 children
  - Perfect Binary Tree: all interior nodes have two children and all leaves have the same depth
  - Complete Binary Tree: every level, *except possibly the last*, is completely filled, and all nodes in the last level are as far left
    as possible





Complete binary tree



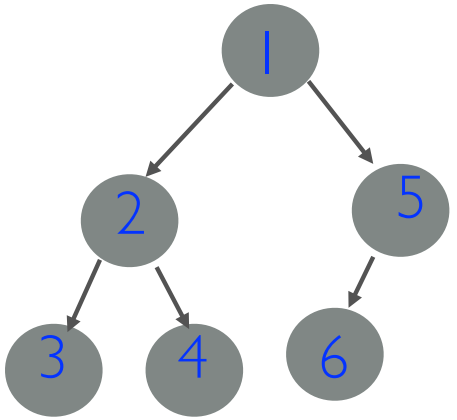
Full binary tree

Perfect binary tree

- Simple results:
  - Full binary tree: number of leaves = number of internal nodes + 1
  - In non-empty binary tree, number of null children = number of nodes in trees+1
  - In perfect binary tree, number of nodes is $2^{height+1} - 1$
    - For n nodes, height = $log_2(n+1) - 1$

Binary tree traversals

- Preorder:
```
Print [Tree]{
    Print root;
    Print Tree[LeftTree];
    Print Tree:[RightTree];
}
```
- Inorder:
```
Print [Tree]{
    Print  Tree[LeftTree];
    Print root;
    Print Tree:[RightTree]
}
```
- Postorder:
```
Print [Tree]{
    Print  Tree[LeftTree];
    Print Tree:[RightTree]
    Print root;
}
```



Pre: 1➔2➔3➔4➔5➔6

In: 3➔2➔4➔1➔6➔5

Post: 3➔4➔2➔6➔5➔1

Search with Dynamic Data

- Problem: finding an element among a group of elements
  - Frequent insertions, frequent deletions, modifications of data
- Operations of interest:
  - Find, insert, delete, find minimum, find maximum

- If we use a sorted list:
  - Takes O(n log(n)) to form sorted list initially from n elements
  - Takes O(n) to insert: Either because of time to find or data movements
  - Takes O(n) to delete: Either because of time to find or data movements
  - Takes O(log(n)) to find: Binary search

- Alternative: Search trees

Binary Search Tree: BST

1. BST is a Binary Tree with keys stored in each node.

2. The key (K_0) in each node is: greater or equal to all keys in T_L , the Left subtree ($K_{left} <= K_0$) less than all keys in T_R, the Right subtree ($K_0 < K_{Right}$)

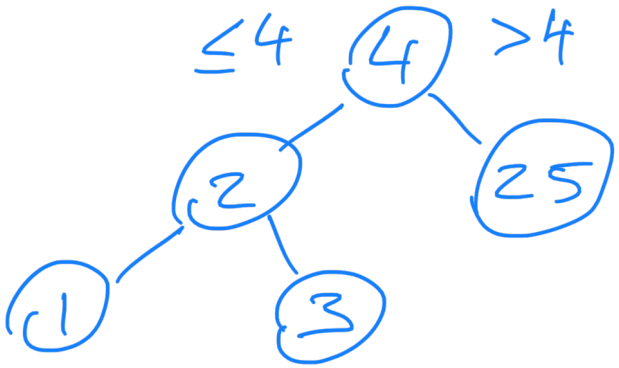
3. The BST defines a partial ordered set --- as you move down to the left/right the keys decrease/increase.

4. Insert new K_{new} push down to subtree Left/Right if K_{new} <=/> K_0.

5. Delete (remove): if K_0 has two children, find **SMALLEST** key in T_R, the Right subtree.

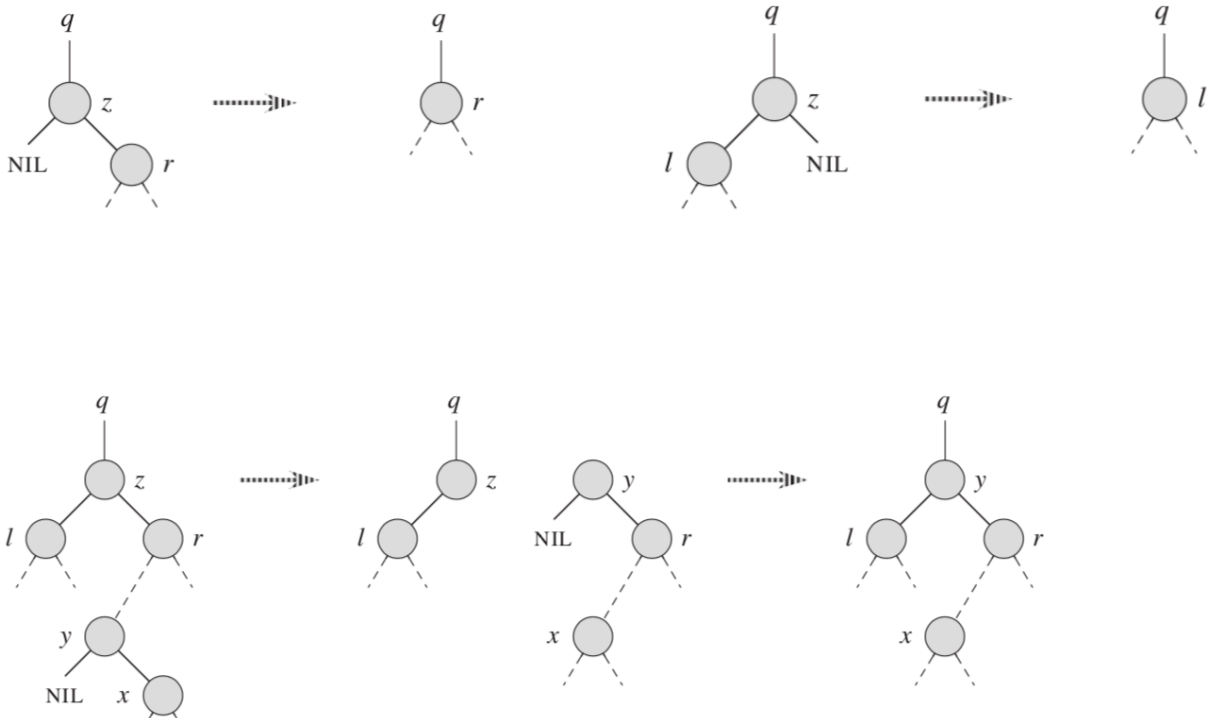
   SMALLEST only has a right child. Right child replaces **SMALLEST** in tree, **SMALLEST** replaces K_0. K_0 is deleted.

   If K_0 has only one child, child to replace K_0 .



Sort → Do inorder traversal

BST Delete



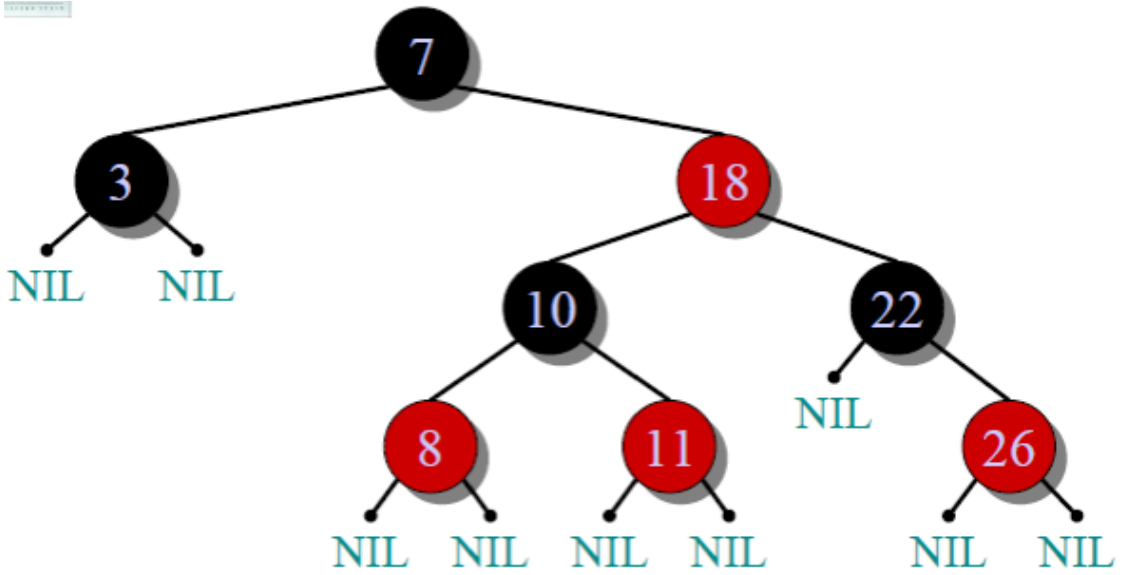
Binary Search Tree: Comments

- In-order traversal yields sorted list!
- Operations
  - Insert, Delete, Find: O(h), where h is height of tree
  - Find minimum, maximum: O(h)
  - Find range: O(n)
- Problem: h can be O(n), for n nodes! Trees can be unbalanced…
- Solution: Use height-balanced binary search trees: h is O(log(n))
  - AVL trees (not that interesting)
  - Red-Black trees (used in many applications: java.util.TreeMap , java.util.TreeSet, java.util.HashMap; C++ STL: map, set, multimap, multiset; Linux kernel: completely fair scheduler, …)

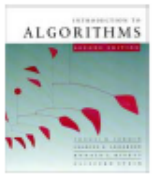
Red Black Trees

- Binary search tree
  - Each node is colored red or black.
  - The root is black
  - If a node is red, its children must be black
  - Every path from a node to a NULL reference must contain the same number of black nodes —> Black height
  - Null references are colored black
- Data structure requires an extra one- bit color field in each node

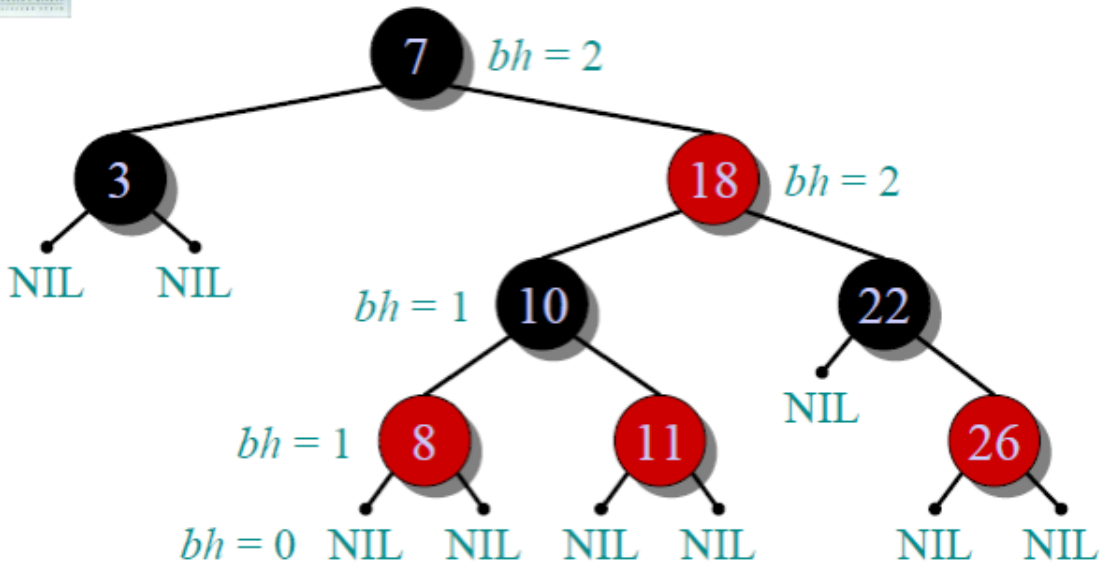
Example of Red Black Trees



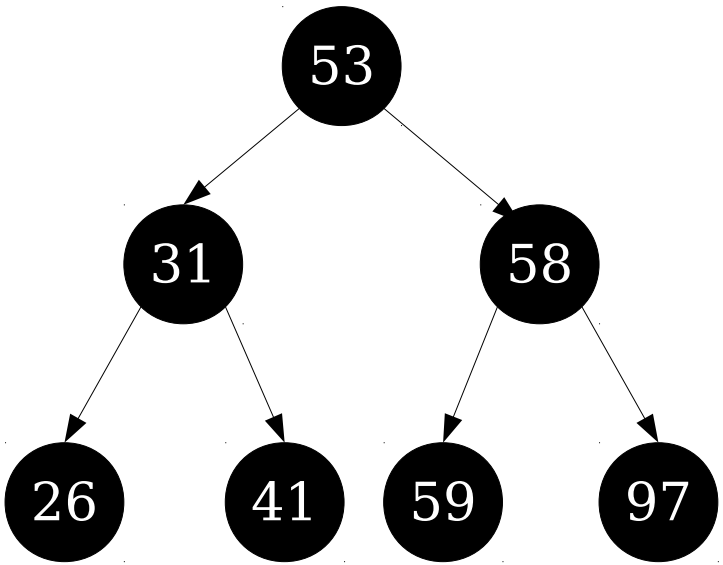
- Each node is colored red or black.
- Root is black
- No red node with red parent
- Every path from a node to a leaf contains the same number of black nodes
- —> Black height
- Null Nodes are black

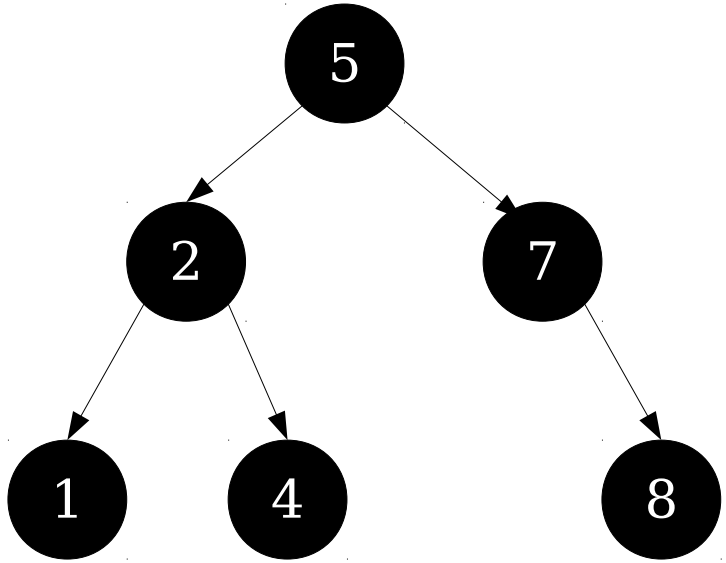


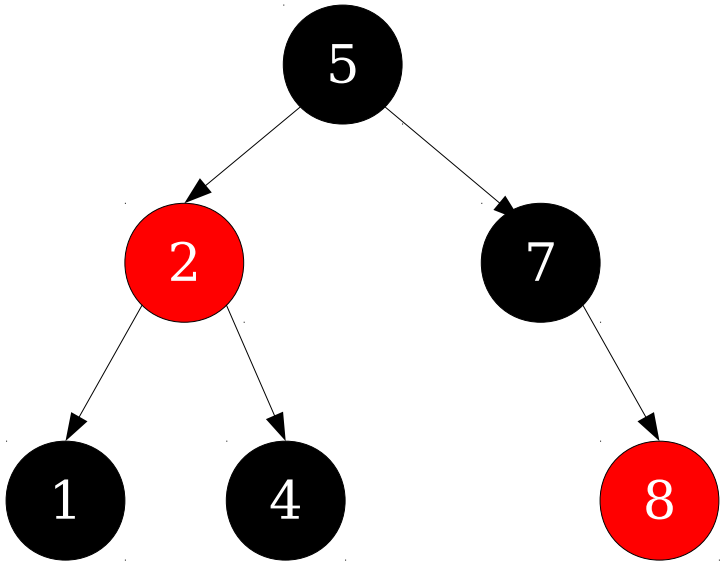
Example of a red-black tree

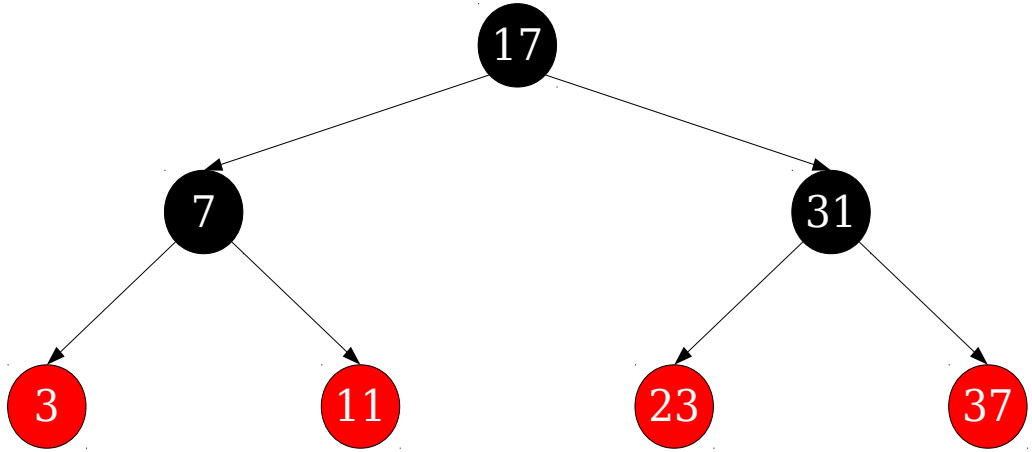


4. All simple paths from any node x to a descendant leaf have the same number of black nodes = *black-height*(x).


53
31
58
26
41
59
97


5
2
7
1
4
8


5
2
7
1
4
8


17
7
31
3
11
23
37

Red Black Tree Properties

- Node of black rank k has height at most 2k,
- Node of black rank k has at least 2^{k-1} descendants,
- RB Tree with n nodes has height at most 2 log(n+1)
- Proof: (More detail in book…merge red nodes with parent black nodes…)
  - Bottom layer has at least 2^k nodes. If k is black rank of node, then max number of nodes in path to leaf is K,
    min number of nodes n in path to leaf is K/2.
  - So, tree is **full** to level K/2.
  - Number of nodes n is at least $2^{K/2}$
  - So, $n + 1 \geq 2^{K/2} \Rightarrow L \leq 2\log_2(n+1)$

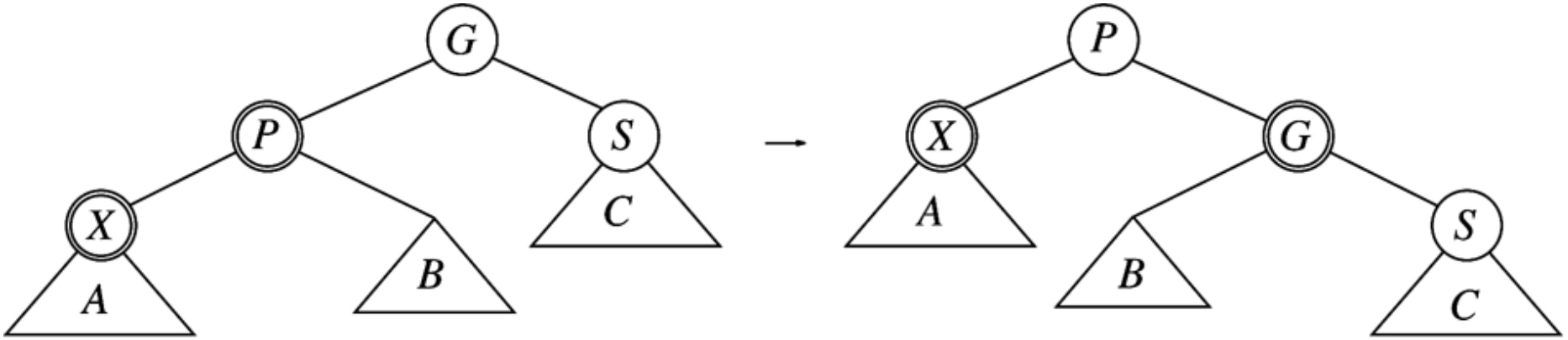
Nice Result for RB Trees

- We will be able to do normal binary tree insertions and deletions
- They will result in temporary violation of RB tree properties
- We will develop an algorithm that restores the RB tree properties using:
  - Recoloring
  - Rotations
- Complexity of insertion and deletion will be O(log(n))

- **Bad news**: lots of cases to handle
  - Detailed in the textbook
  - We will overview; if you need to implement, look up details for each case

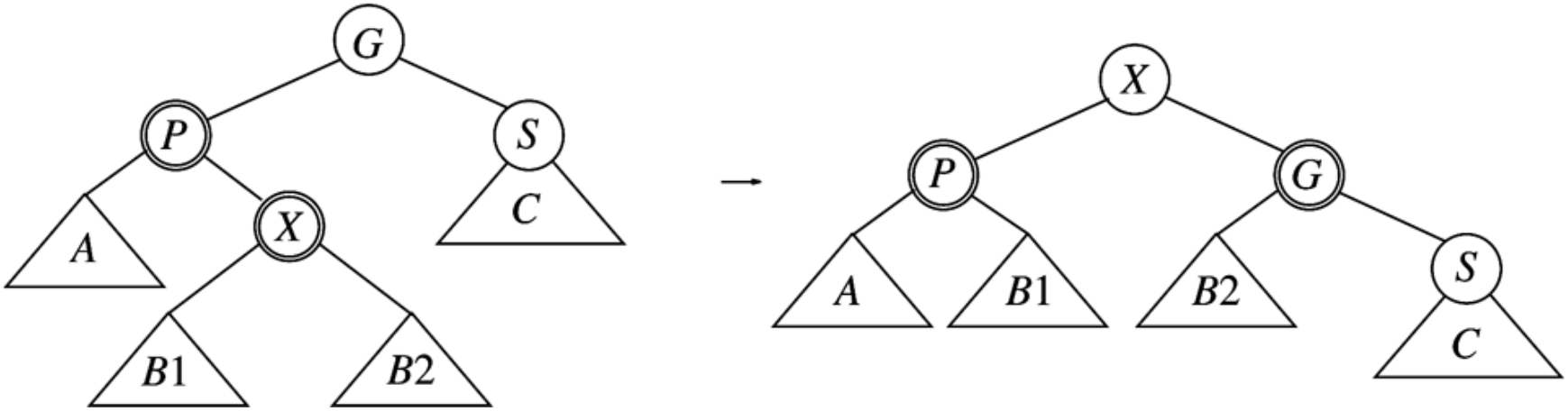
Tree Rotations

Right rotate (Zig) G



Tree Rotations

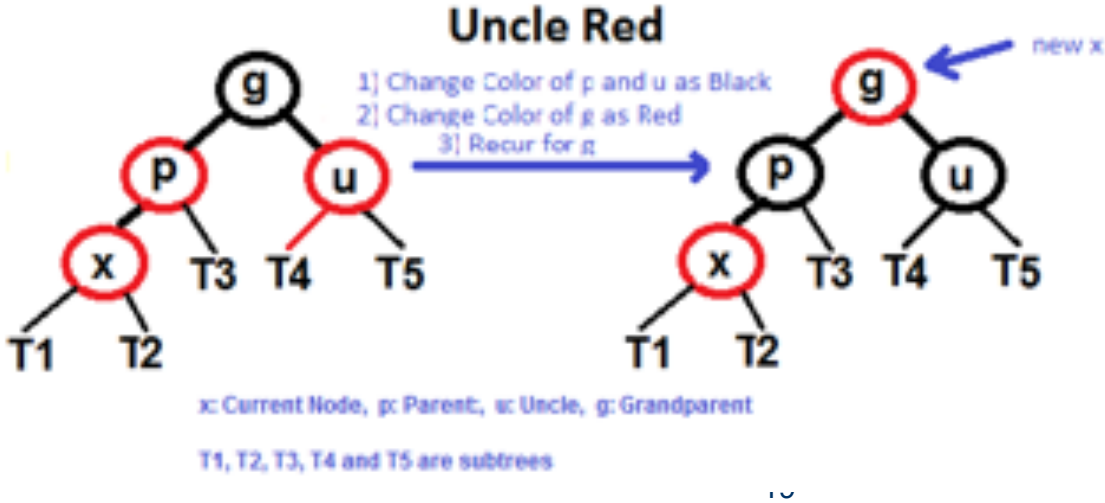
Left-Right rotate (Zag-Zig): Left rotate P, right rotate G



Insertion in RB Trees

Insert as in RB tree and color the new node Red.

1. If parent of new node is Black, nothing needs to be done.
   Tree is RB tree
2. If parent is RED, and uncle is also RED,
   A. Change color of parent **and** uncle as **BLACK**
   B. Change color of grandparent as RED
   C. Continue fixing up tree if grandparent has RED parent



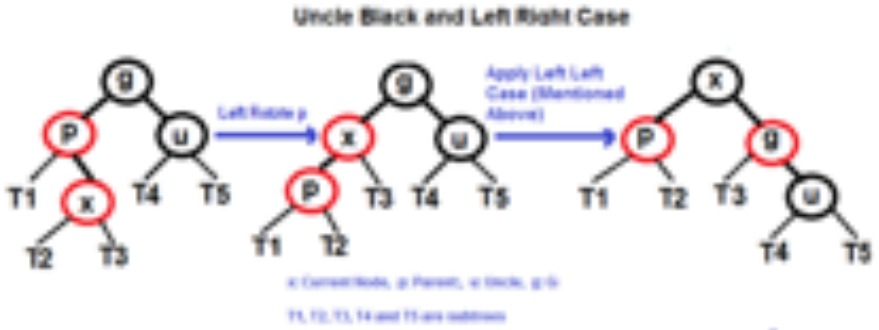
Insertion in RB Trees (2)

Insert as in RB tree and color the new node Red.

3. If parent is RED, and uncle is **BLACK**, 4 possible cases with parent (p), grandparent (g)
   A. x left of p left of g (Left-left)
   B. x left of p right of g (left-right)
   C. x right of p right of g (right-right) mirror A
   D. x right of p, left of g (right-left) mirror B

CASE A

CASE D



Insertion in RB Trees (3)

Complexity of insertion

A. At most O(log(n)) recoloring

B. At most two rotations

Result: O (log(n)).

Example: Insert 1, 2, 3, 4, 5, 6, 7

Deletion in RB Trees

Remove node as in regular binary search trees

Fix resulting BST to restore red-black property

Problem: Many special cases - Discussed in great detail in CLRS

Will illustrate with examples, show they can be done in O(log(n)) time.

Reminder: BST Delete Node X

- X is leaf: Simply delete it
- X has only one child: Copy the child to node X, **delete** the child
- X has two children: Find inorder successor of the node. Copy right child of successor to replace successor. Copy contents of successor to X into X; **delete** inorder successor.

Deletion in RB Trees - 2

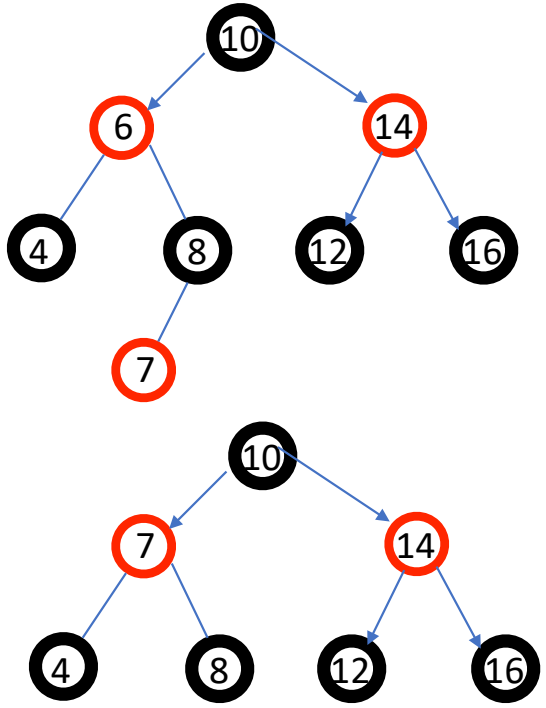
Easy case: Node that will be deleted in BST Remove is RED. Then, just delete it as in BST case and all will be well.

e.g. Remove 7. Simply delete node 7.

e.g Remove 6.

6's data replaced by 7's data, same color

7 is deleted



Deletion in RB Trees - 3

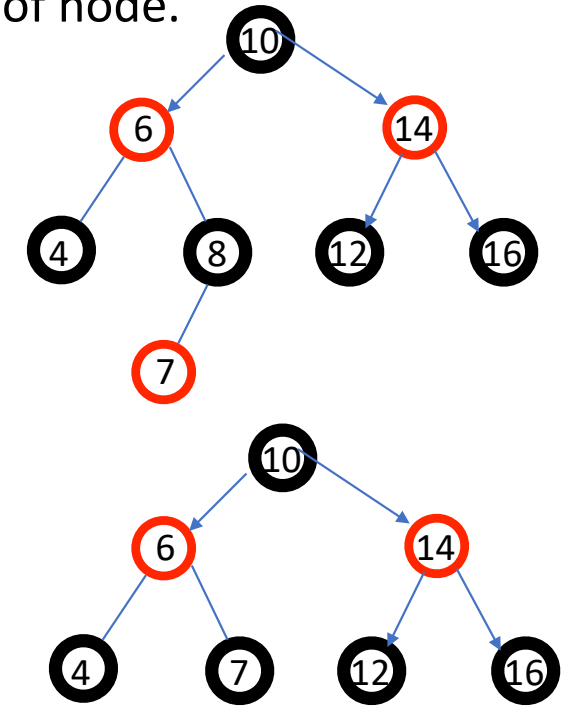
Easy case: Node that will be deleted in BST Remove is black and has single red child. Then, just delete it as in BST case, but keep color of node.

e.g. Remove 8.

Find node to delete as 7.

Copy 7 to 8, keeping original color of 8.

-> Preserves black height.



Deletion in RB Trees - 4

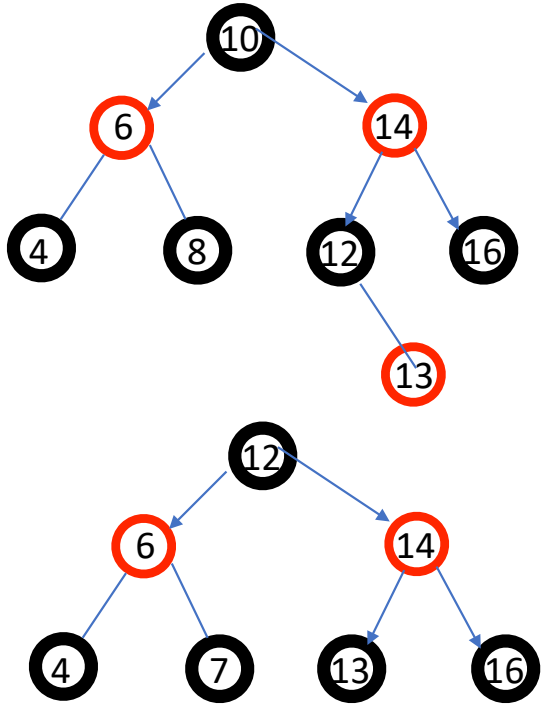
Easy case: Node that will be deleted in BST Remove is black and has single red child. Then, just delete it as in BST case, but keep color of node.

e.g. Remove 10.

Find node to delete as 12.

Copy 12 to 10, keeping original color of 10.

Copy 13 to 12, change color of 13 to 12's color



Deletion in RB Trees - 5

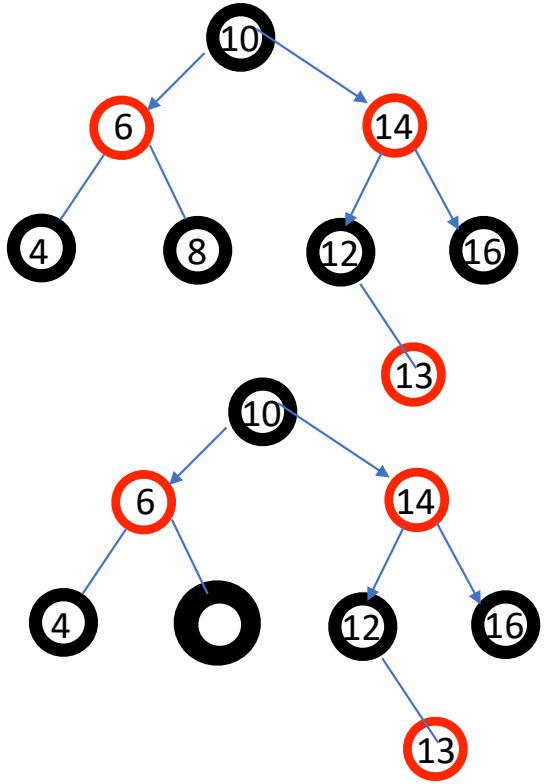
Hard case: Node that will be deleted in BST Remove is black and has two black children! (e.g. Null). This creates black height imbalance

e.g. Remove 8.

Creates an imbalance in black height.

We identify this as a “double black”

Must correct with rotations, recoloring.



Deletion in RB Trees - 6

Hard case: Node that will be deleted in BST Remove is black and has two black children! (e.g. Null). This creates black height imbalance

e.g. Remove 8.

Creates an imbalance in black height.

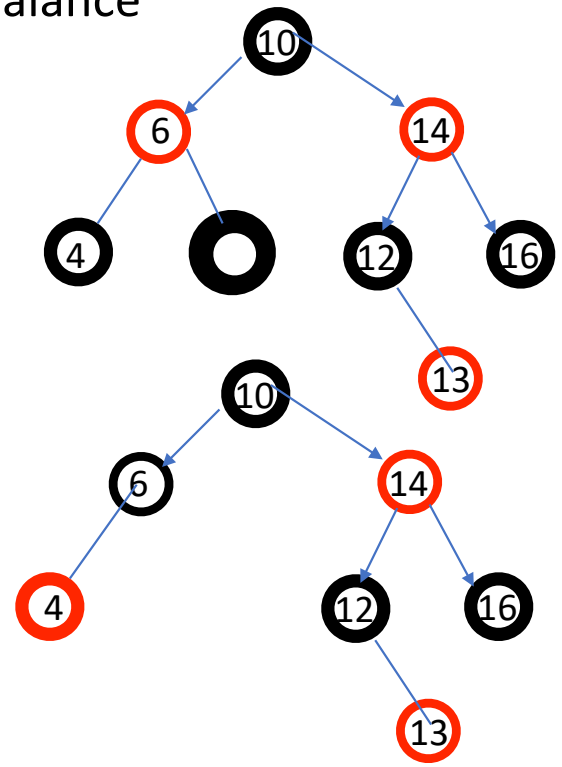
We identify this as a “double black”

Must correct with rotations, recoloring.

Sibling black —> recolor as red; if parent red

Recolor as black. If parent is black, color as

“double black” and recur up the tree!



Deletion in RB Trees - 7

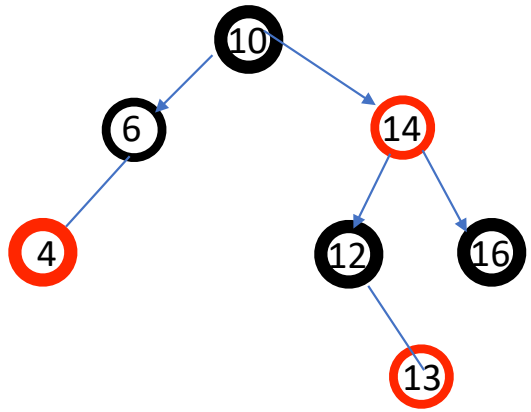
Hard case: Node that will be deleted in BST Remove is black and has two black children! (e.g. Null). This creates black height imbalance

Sibling is black, has red child

e.g. delete 16

a) Rotate 13 left, make it black, make 12 red

b) Rotate 13 right, restoring black height



Deletion in RB Trees - 8

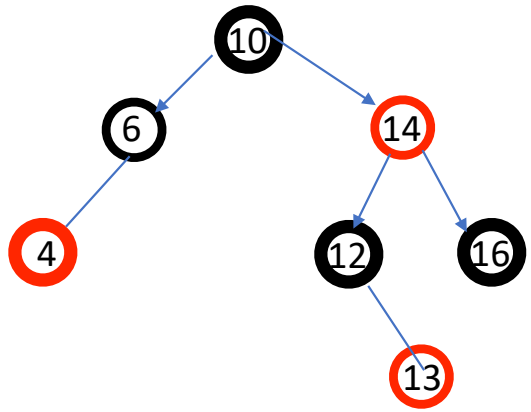
Hard case: Node that will be deleted in BST Remove is black and has two black children! (e.g. Null). This creates black height imbalance

Sibling is red

e.g. delete 6

a) Rotate 14 left, make it black

b) Adds black height automatically



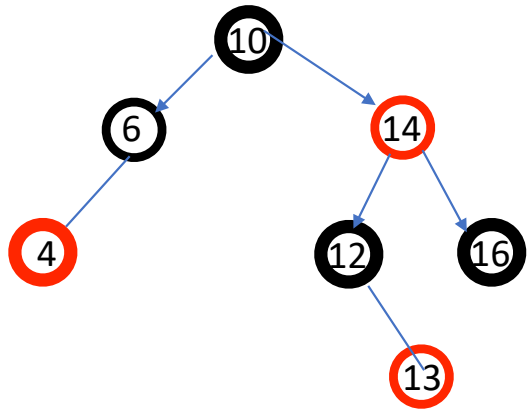
RB Trees Summary

1) Can perform insertion in O(log(n))

2) Can perform deletions in O(log(n))

   O(log(n)) recoloring maximum

   One or two rotations



Conclusion: find, insert, delete are O(log(n))

So why are RB trees used more than AVL trees?

   AVL trees take more space

   AVL trees do more rotations on insert, delete —> more overhead

Drawbacks of RB trees: overhead to maintain RB property, 1 bit extra

Splay Trees

Another binary tree approach: Splay trees are self-adjusting binary trees.

- "Lazy" data structure: Don't insist on balance all the time, but recover balance when needed
- If elements of a BST are not to be accessed uniformly, then "balance" may not be

Begin with an arbitrary BST.

After looking up an element, repeatedly rotate that element with its parent until it becomes the root.

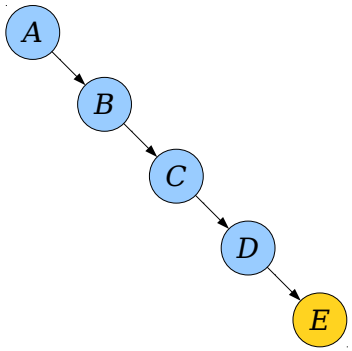
Recently-accessed elements will be up near the root of the tree, lowering access time.

Unused elements stay low in the tree.

Simple Idea

After accessing node, use single rotations to rotate it to root.

e.g. access E, then D, then C, then B, then A.



Splay Tree Operations

- Insert: Standard BST insert
- Find: find as in BST, then splay to root using splay operation
- Delete: find as in BST, then splay to root using splay operation.
  - Subsequently, remove root, divide into 2 subtrees, do another splay, merge into single tree

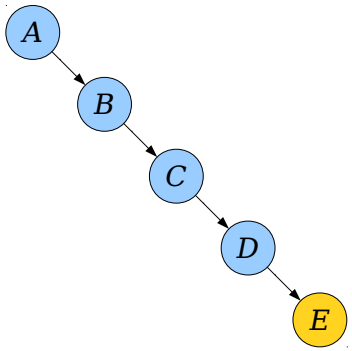
Key: Splay operation

Double rotations to avoid problems above!

Simple Idea

After accessing node, use single rotations to rotate it to root.

e.g. access E, then D, then C, then B, then A.



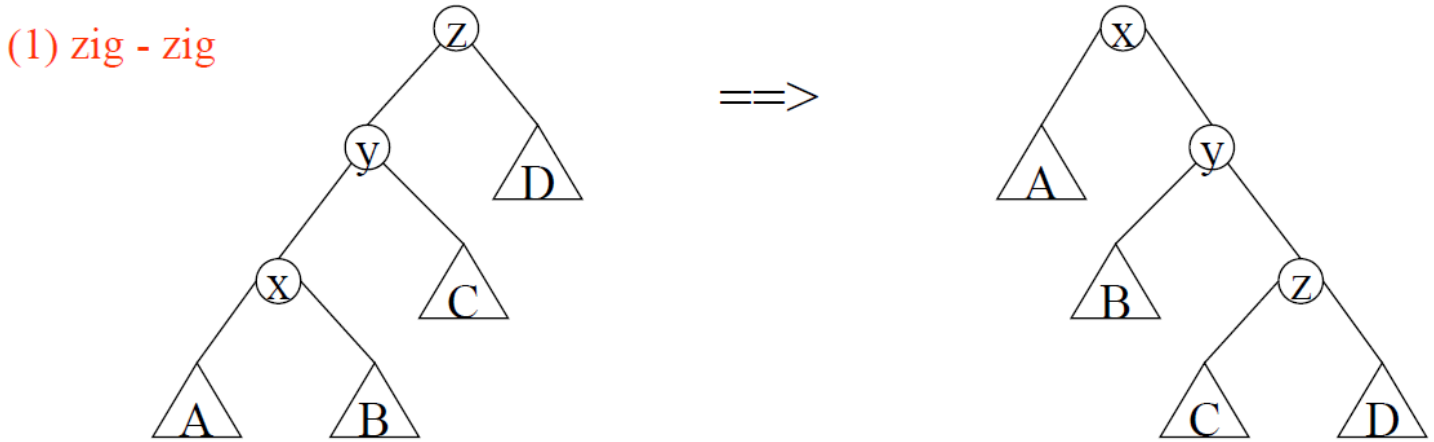
Splay Operation

Splay node X:

- If parent is root, do rotation towards parent and end
- If parent is not root, and grandparent is aligned with parent (zig-zig):
  - Rotate parent towards grandparent
  - Then rotate X towards parent
- If parent is not aligned with grandparent (zig-zag):
  - Rotate X towards parent
  - Then rotate X towards grandparent

Splay Operation - 1

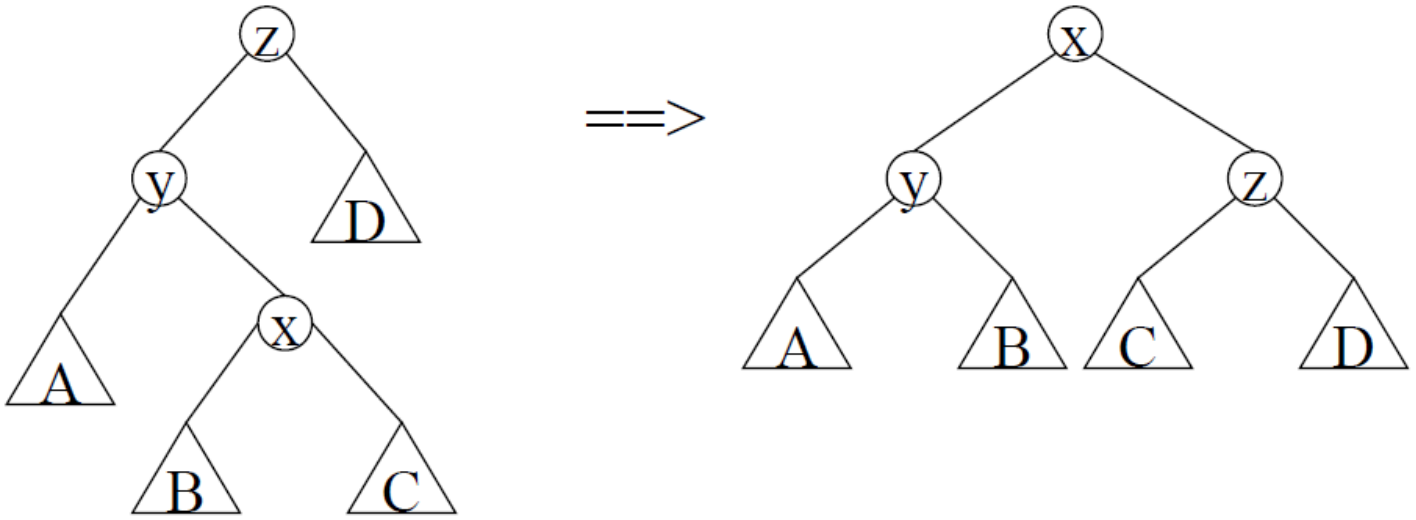
- Splay(x): Parent is not root, and grandparent is aligned with parent (zig-zig):
  - Rotate parent towards grandparent
  - Then rotate X towards parent



Splay Operation - 2

- Splay(x): Parent is not root, and grandparent is not aligned with parent (zig-zag):
  - Rotate X towards parent
  - Then rotate X towards grandparent

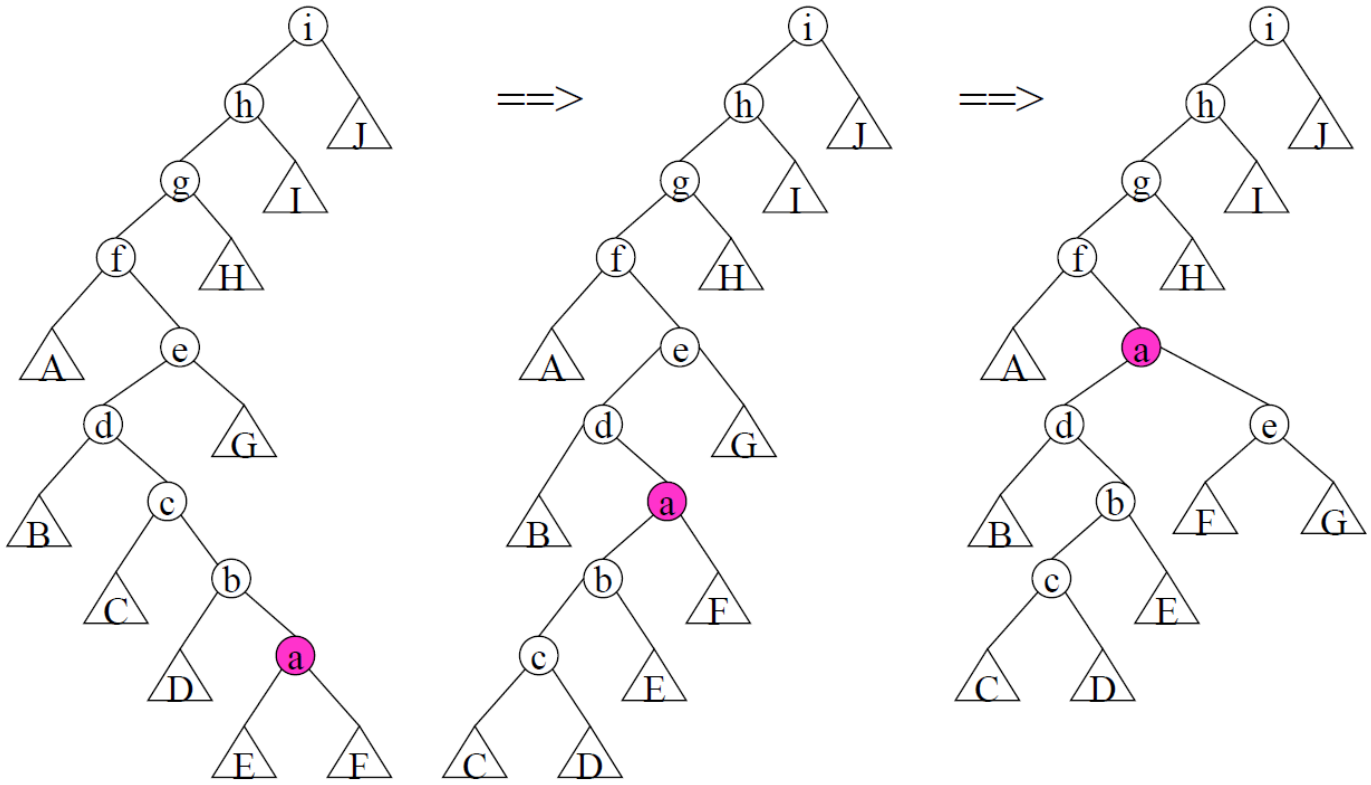
(2) zig - zag

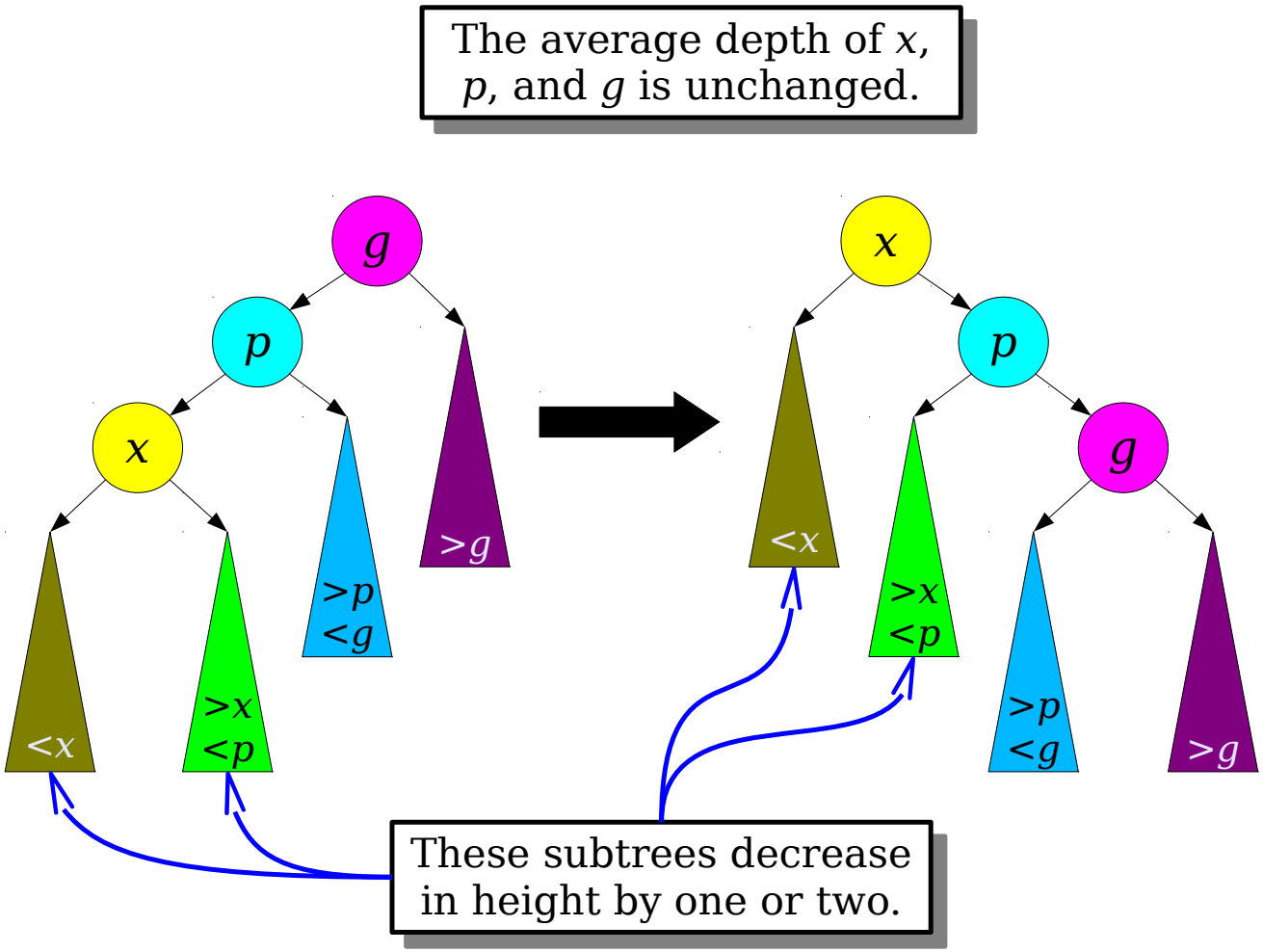


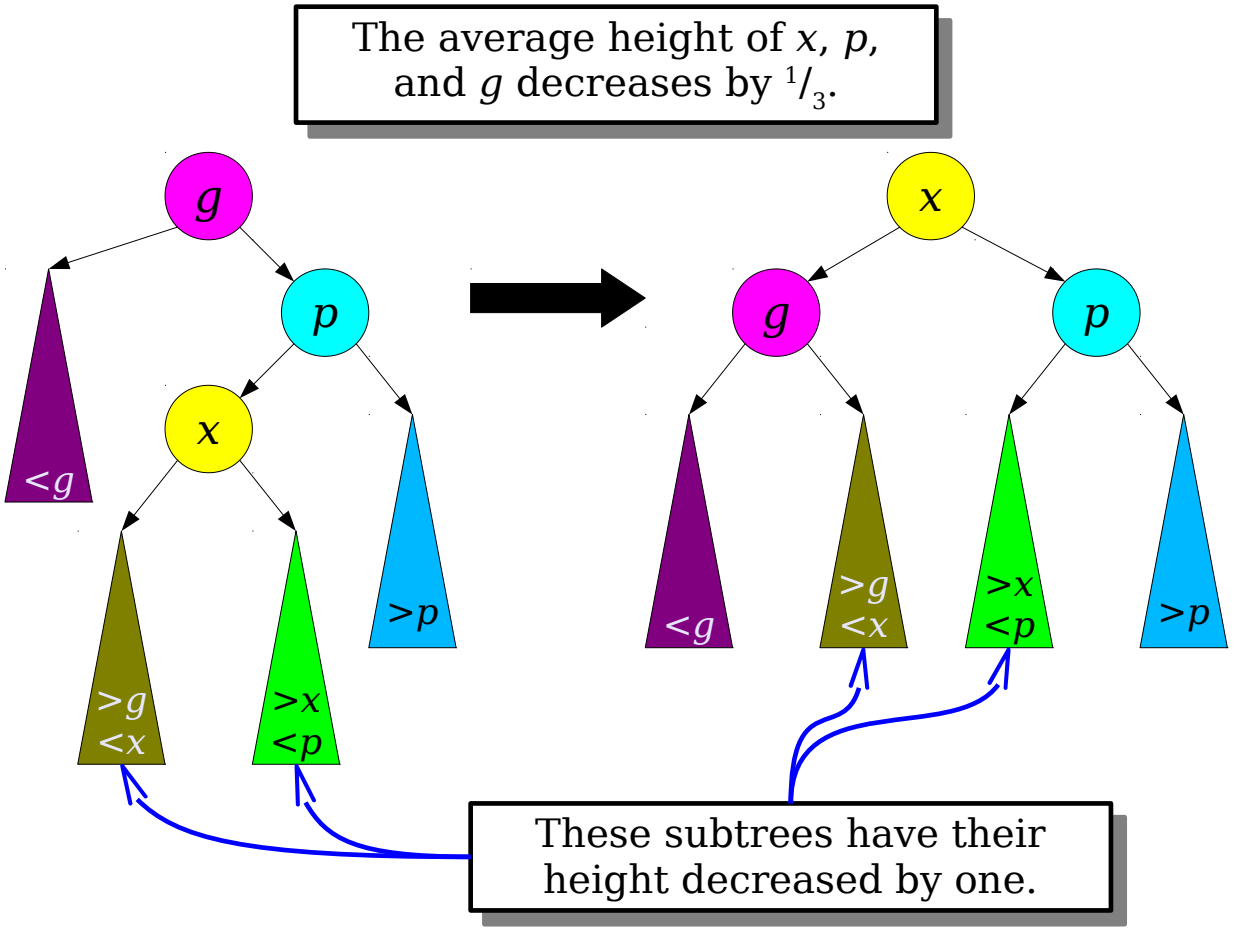
Splay Operation - example

- Splay(a)

Observation: Splaying at a node of depth d takes $\Theta(d)$ time. It moves x to the root, but it also halves the average depth of nodes on the path from x to the root




The average depth of x, p, and g is unchanged.
g
p
x
<x
>x <p
>p <g
>g
x
p
g
<x
>x <p
>p <g
>g
These subtrees decrease in height by one or two.


The average height of x, p, and g decreases by 1/3.
g
p
x
<g
>p
>g <x
>x <p
x
g
p
<g
>g <x
>x <p
>p
These subtrees have their height decreased by one.

Complexity of Splay Trees

Amortized complexity is O(log(n))

- Will show (after we learn about amortized complexity

Splay trees have excellent locality properties. Frequently accessed items are easy to find. Infrequent items are out of way.

Splay trees are simpler compared to AVL and Red-Black Trees as no extra field is required in every tree node.

Splay trees are widely used basic data structure, because they're the fastest type of balanced search tree for many applications

- Splay trees are used in Windows NT (in the virtual memory, networking, and file system code),
- the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers,
- Unix malloc, Linux loadable kernel modules, and in much other software