Algorithm and Object-Oriented Programming for Modeling
Part 3: Trees and related algorithms

MSDM 5051, Yi Wang (王一), HKUST

Side remark: why pulling trees to a new part?

This is refactoring of teaching, like refactoring of code

- Open to addition of new features (e.g. search in addition to sort)
- Modularity (functions should be small)
- Single responsibility (do one thing and do it well)

(I need to make O(n) change to rename files…)

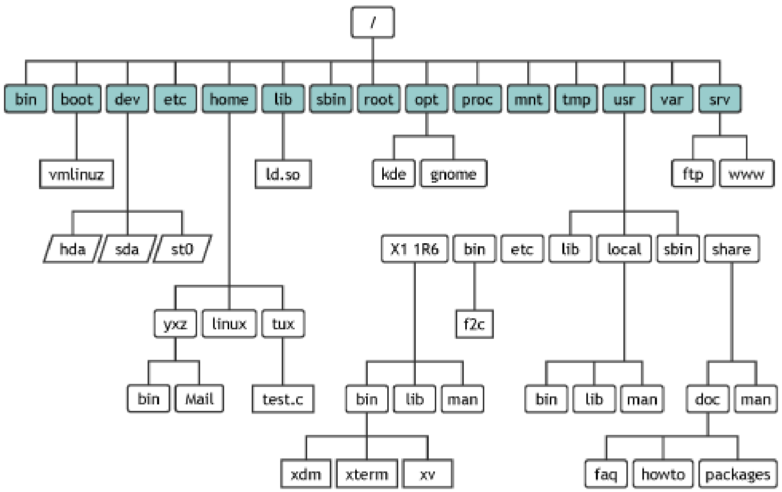
Section 1. Why trees?

Why not the good old array, link list, heap

or even the magic dict?

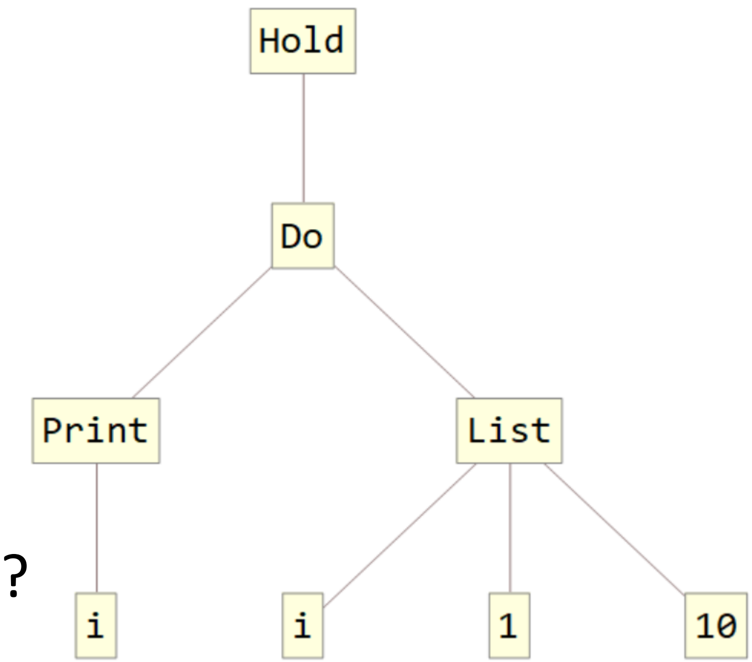
1. Natural trees for the logical connection of data

   e.g., file system, xml, grammar parsing, …



In[3]:= `Hold[Do[Print[i], {i, 1, 10}]] // TreeForm`

Out[3]//TreeForm=



2. Make trees

   Why making lists into trees?

   (1) Search (more later)

   (2) encoding/decoding (e.g., Huffman tree)

Recall example: how to insert a card?

Need:

- Search for insertion place: O(logn)
- Insertion: O(logn)

Want O(logn)?
Plant a tree!

Recall example: how to insert a card?

Need:

- Search for insertion place: O(logn)
- Insertion: O(logn)

Want O(logn)?
Plant a tree!

Key question:

how to do fast insertion/deletion/search?

Q: Why not dict?

Recall example: how to insert a card?

Need:

- Search for insertion place: O(logn)
- Insertion: O(logn)

Want O(logn)?
Plant a tree!

Key question:

how to do fast insertion/deletion/search?

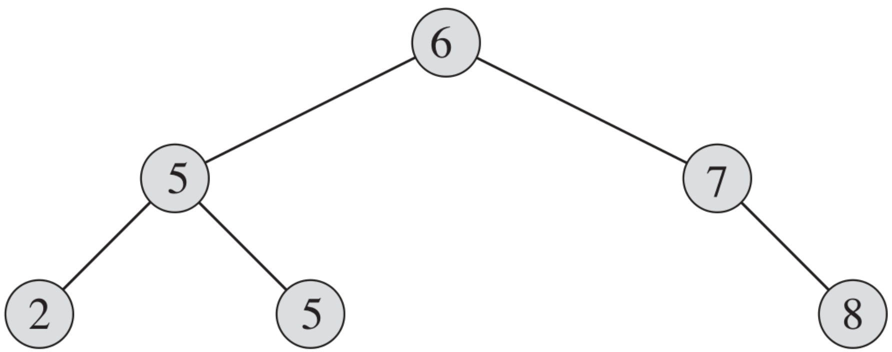
Q: Why not dict?

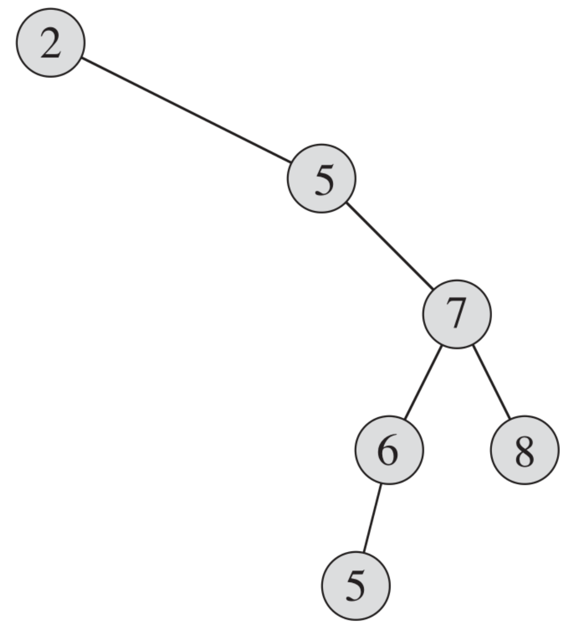
Q: How to use a tree to realize dict?

Section 2: Binary Search Tree

Key question: how to do fast insertion/deletion/search?

Binary search tree (BST): (left sub tree) $\leq$ parent $\leq$ (right sub tree) for all nodes (Not unique for a set of data)





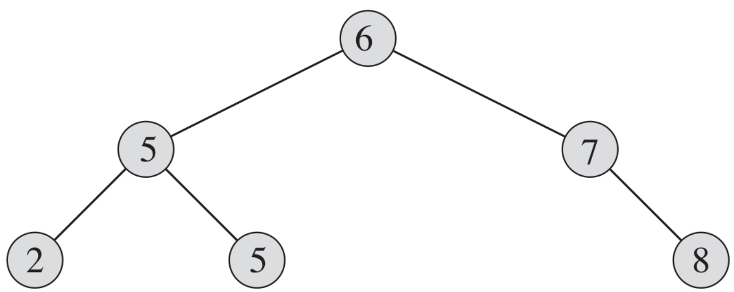
Walk (traversal) methods:

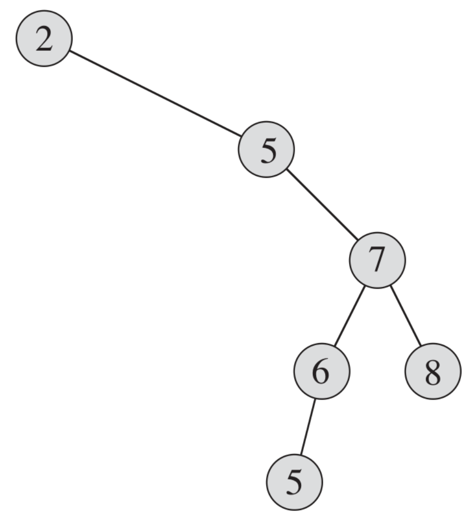
- Depth first search (DFS)
    - in-order, pre-order, post-order (recursion or stack)

```python
def inorder(self, node = 0, result = None):
    if result is None:
        result = []
    if node == 0:
        node = self.root
    if node:
        self.inorder(node.left, result)
        result.append(node.data)
        self.inorder(node.right, result)
    return result
```

- Breadth first (BFS), queue (or in Python can use dict)

Once BST constructed, in-order walk => sort



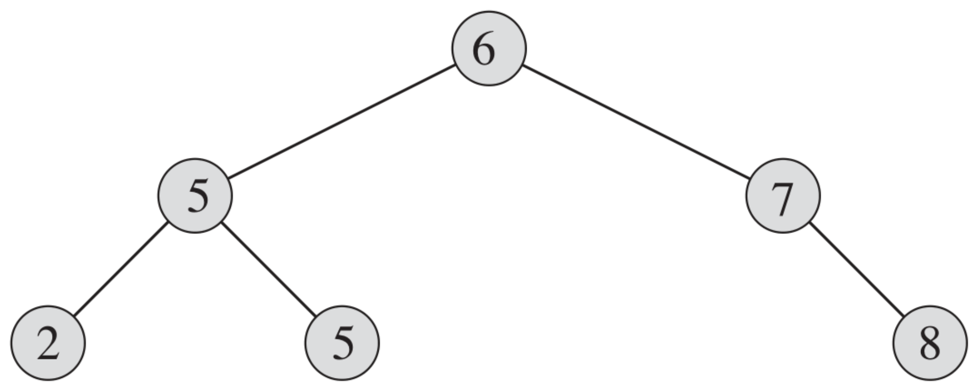


How to construct a BST?

How to insert it to BST?

Simplest way:

- Compare with node
- Smaller: left, greater/equal: right
- Until child = None, add

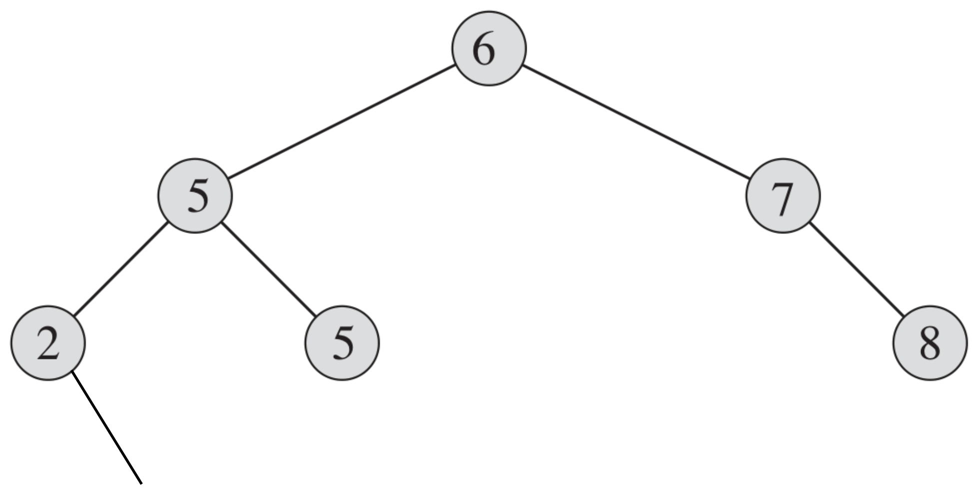


How to construct a BST?

Simplest way:

- Compare with node
- Smaller: left, greater/equal: right
- Until child = None, add

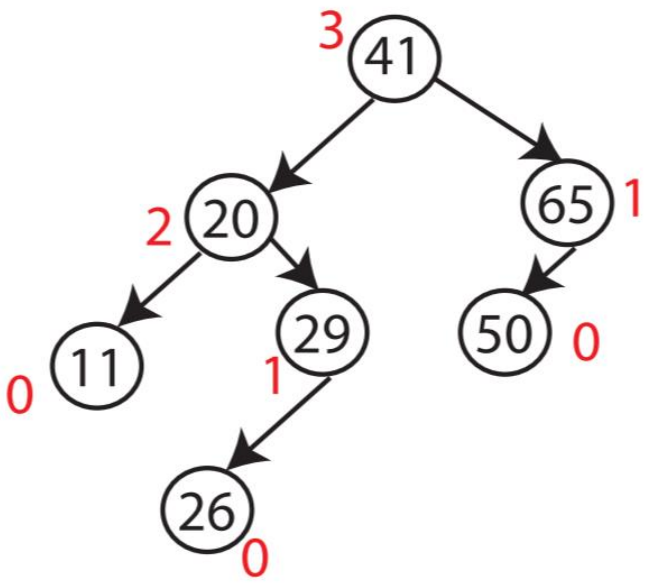
How to insert it to BST?



Height of node

(not essential for BST

but here we will keep height)



binary_tree.py

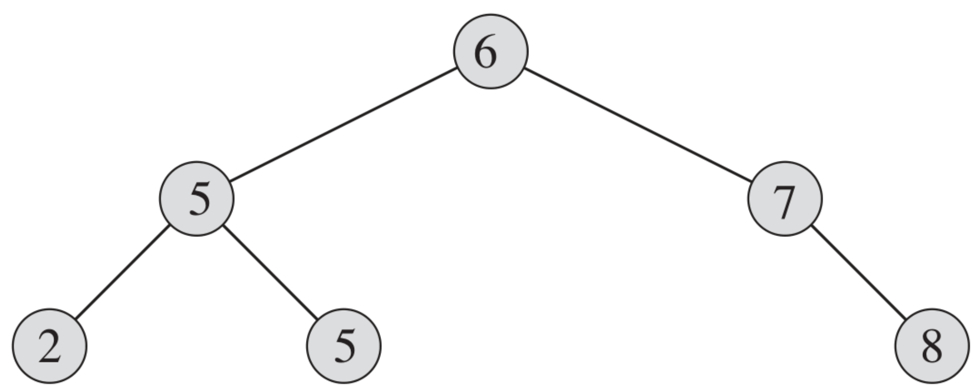
```python
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.parent = None
        self.data = data
        self.height = 0

class BinaryTree:

    def __init__(self):
        self.root = None

    def __str__(self, node = 0, depth = 0, direction_label = ""):
        "The tree structure in string form, to be used in str(my_node) or print(my_node)."
        if node == 0:
            node = self.root
        if node:
            height_info = "(H"+str(node.height)+")" if node.height > 0 else ""
            return depth * "\t" + direction_label + height_info + str(node.data) + "\n" + \
                self.__str__(node.left, depth+1, "L:") + self.__str__(node.right, depth+1, "R:")
        else:
            return ""

    def inorder(self, node = 0, result = None):
        if result is None:
            result = []
        if node == 0:
            node = self.root
        if node:
            self.inorder(node.left, result)
            result.append(node.data)
            self.inorder(node.right, result)
        return result
```

binary_search_tree.py

```python
import binary_tree

class Node(binary_tree.Node):
    def left_height(self):
        return -1 if self.left is None else self.left.height
    def right_height(self):
        return -1 if self.right is None else self.right.height
    def update_height(self):
        self.height = max(self.left_height(), self.right_height()) + 1
    def balance(self):
        "-2, -1: left heavy, 1, 2: right heavy"
        return self.right_height() - self.left_height()

class BinarySearchTree(binary_tree.BinaryTree):
    def __init__(self, data_array = []):
        self.root = None
        for data in data_array:
            self.insert(Node(data))

    def insert(self, new_node, node = 0):
        if not self.root:
            self.root = new_node
            return new_node
        if node == 0:
            node = self.root
        if new_node.data < node.data:
            if node.left:
                self.insert(new_node, node.left)
            else:
                new_node.parent = node
                node.left = new_node
        else:
            if node.right:
                self.insert(new_node, node.right)
            else:
                new_node.parent = node
                node.right = new_node
        node.update_height()
```

```
    def right_height(self):
        return -1 if self.right is None else self.right.height
    def update_height(self):
        self.height = max(self.left_height(), self.right_height()) + 1
    def balance(self):
        "-2, -1: left heavy, 1, 2: right heavy"
        return self.right_height() - self.left_height()

class BinarySearchTree(binary_tree.BinaryTree):
    def __init__(self, data_array = []):
        self.root = None
        for data in data_array:
            self.insert(Node(data))

    def insert(self, new_node, node = 0):
        if not self.root:
            self.root = new_node
            return new_node
        if node == 0:
            node = self.root
        if new_node.data < node.data:
            if node.left:
                self.insert(new_node, node.left)
            else:
                new_node.parent = node
                node.left = new_node
        else:
            if node.right:
                self.insert(new_node, node.right)
            else:
                new_node.parent = node
                node.right = new_node
        node.update_height()

    def sort(self):
        return self.inorder()

def BST_sort(array):
    my_tree = BinarySearchTree(array)
    return my_tree.sort()
```

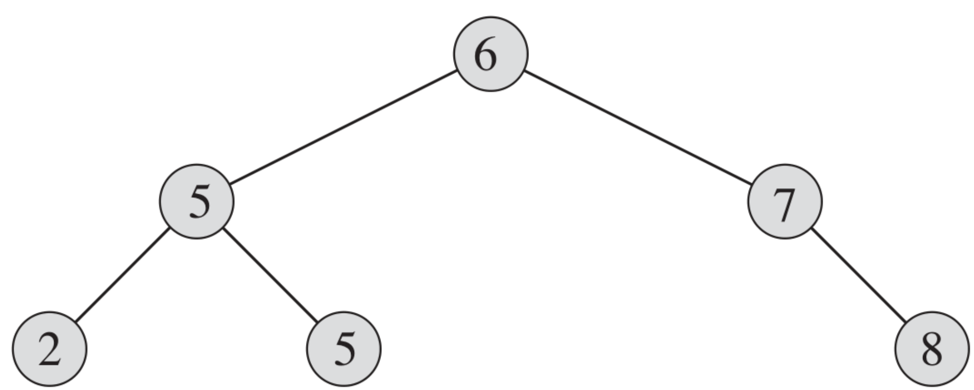
How to delete data from BST?



How to delete data from BST?

1) ***Node to be deleted is the leaf:*** Simply remove from the tree.

```
              50                            50
           /     \         delete(20)      /   \
          30      70       --------->    30     70 
         /  \    /  \                     \    /  \ 
       20   40  60   80                   40  60   80
```



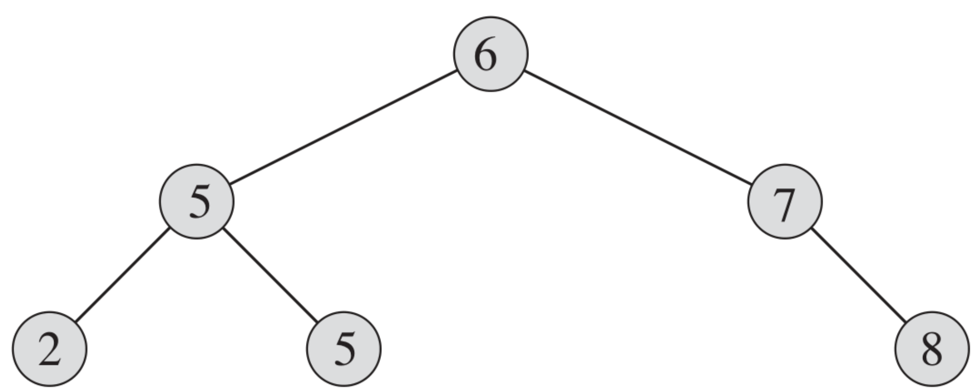
How to delete data from BST?

1) ***Node to be deleted is the leaf:*** Simply remove from the tree.

```
              50                            50
           /     \         delete(20)      /   \
          30      70       --------->    30     70
         /  \    /  \                     \    /  \
       20   40  60   80                   40  60   80
```

2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

```
              50                            50
           /     \         delete(30)      /   \
          30      70       --------->    40     70
            \    /  \                          /  \
            40  60   80                       60   80
```



How to delete data from BST?

1) *Node to be deleted is the leaf:* Simply remove from the tree.

```
              50                            50
           /     \         delete(20)      /   \
          30      70       --------->    30     70
         /  \    /  \                     \    /  \
       20   40  60   80                   40  60   80
```

2) *Node to be deleted has only one child:* Copy the child to the node and delete the child

```
              50                            50
           /     \         delete(30)      /   \
          30      70       --------->    40     70
            \    /  \                          /  \
            40  60   80                       60   80
```

3) *Node to be deleted has two children:* Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
              50                            60
           /     \         delete(50)      /   \
          40      70       --------->    40    70
                 /  \                            \
                60   80                           80
```

image from [Binary Search Tree | Set 2 (Delete) - GeeksforGeeks]

Problem: the tree may get too high

i.e. not “balanced”

(Balanced: height of all sub-trees differ by at most 1)

For example, insert 5.1, 5.2, 5.3, …

Then insertion is O(n) instead of O(logn)

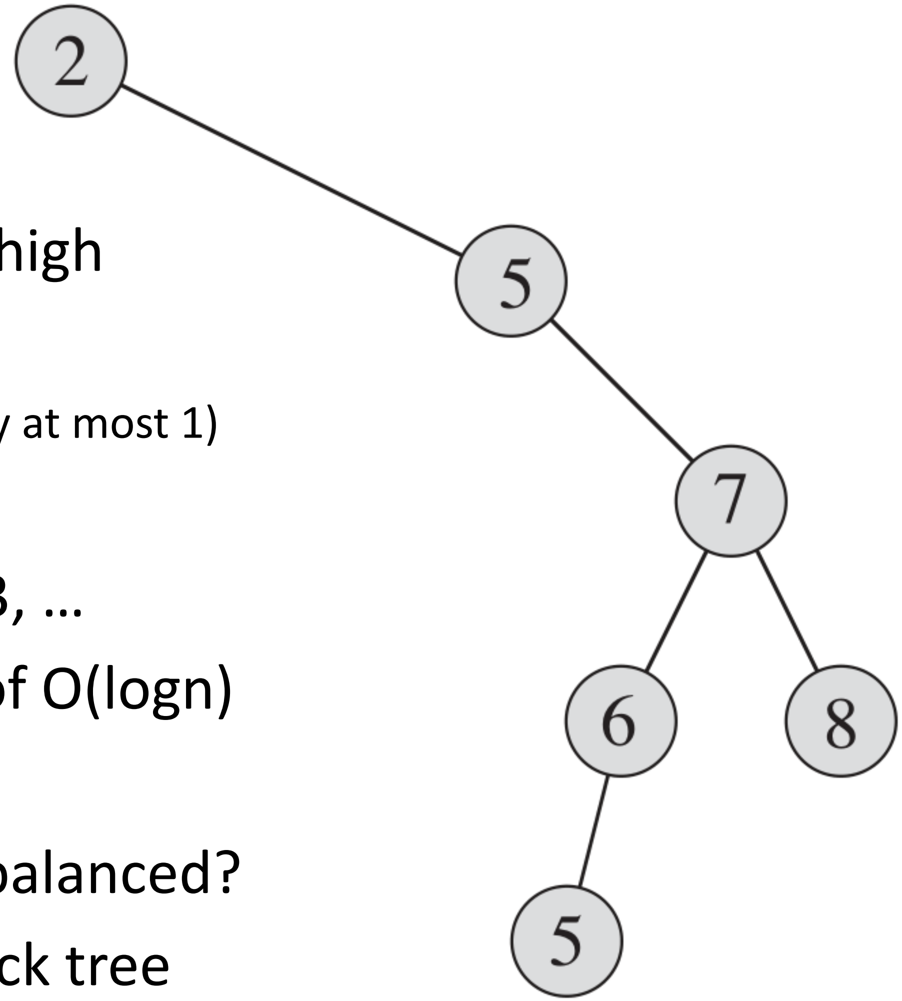
How to insert while keep tree balanced?

Algorithms: AVL tree or red black tree

usually lower height

usually faster insertion

Georgy Adelson-Velsky and Evgenii Landis, 1962



Section 3: AVL Tree

BST is not unique.

If a tree is not balanced, reconnect to make it balanced.

Q1: How to measure balanced or not?

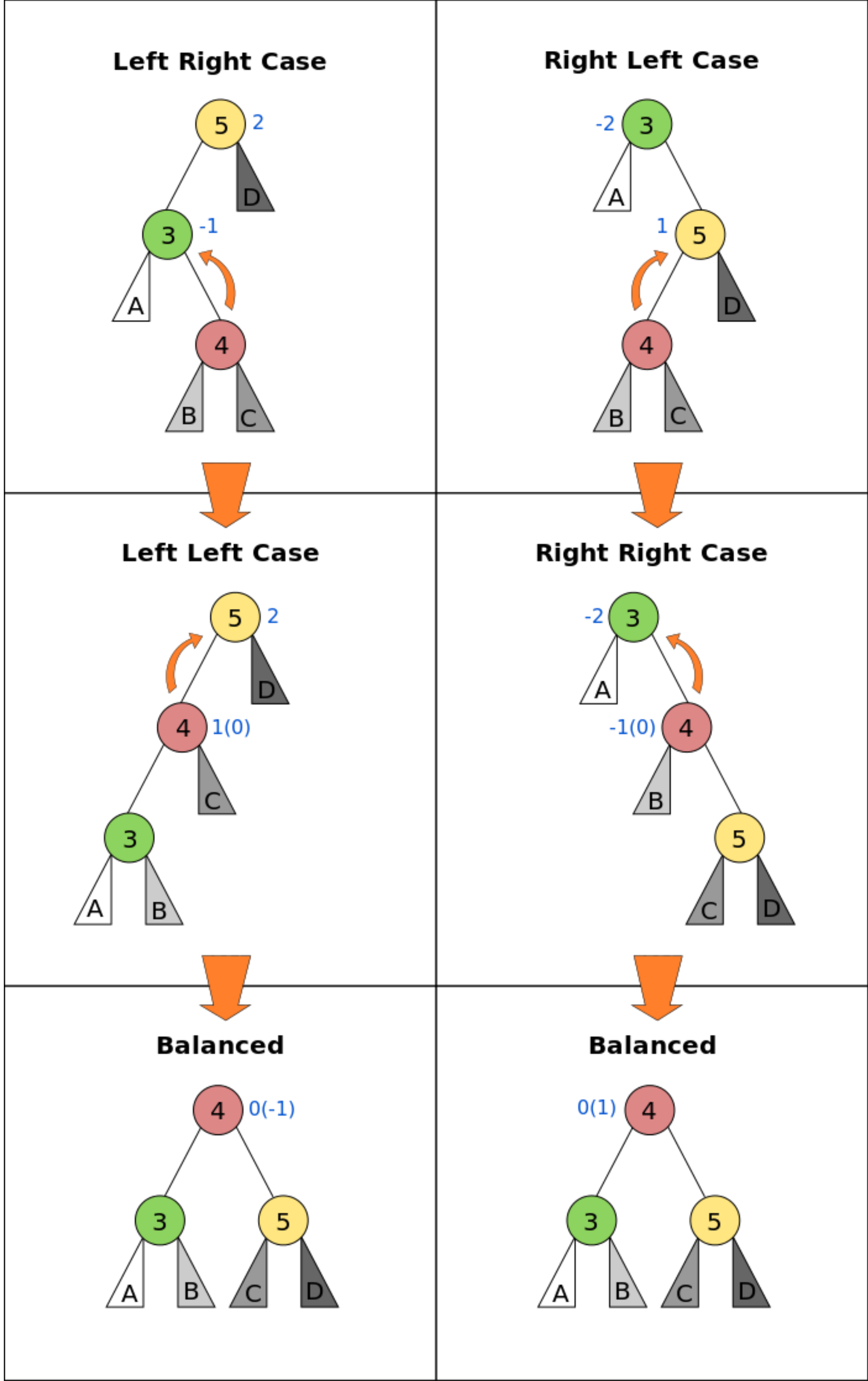
Q2: How to reconnect to make sure balance?

AVL algorithm:

1. BST insert
2. Fix unbalanced case by "rotation"

Can-do spirit
(trail and error)
(see also heap)

You can you up,
No can be children



Concept: height of nodes:

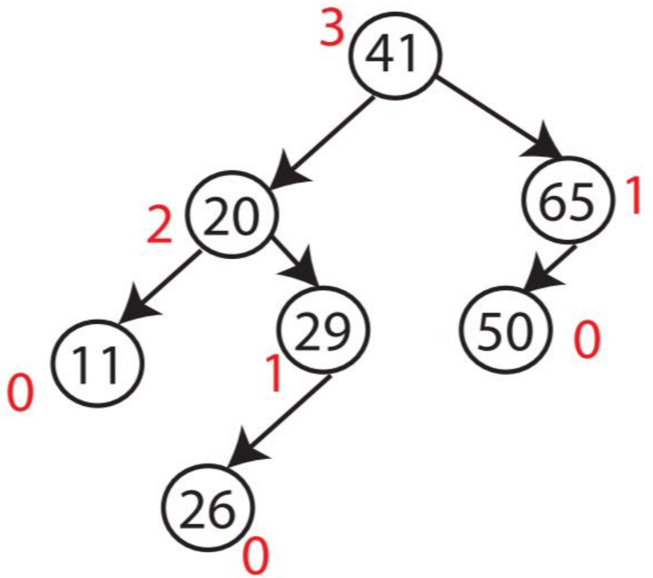
- Count from leaves
- Add one from greater child

AVL property:

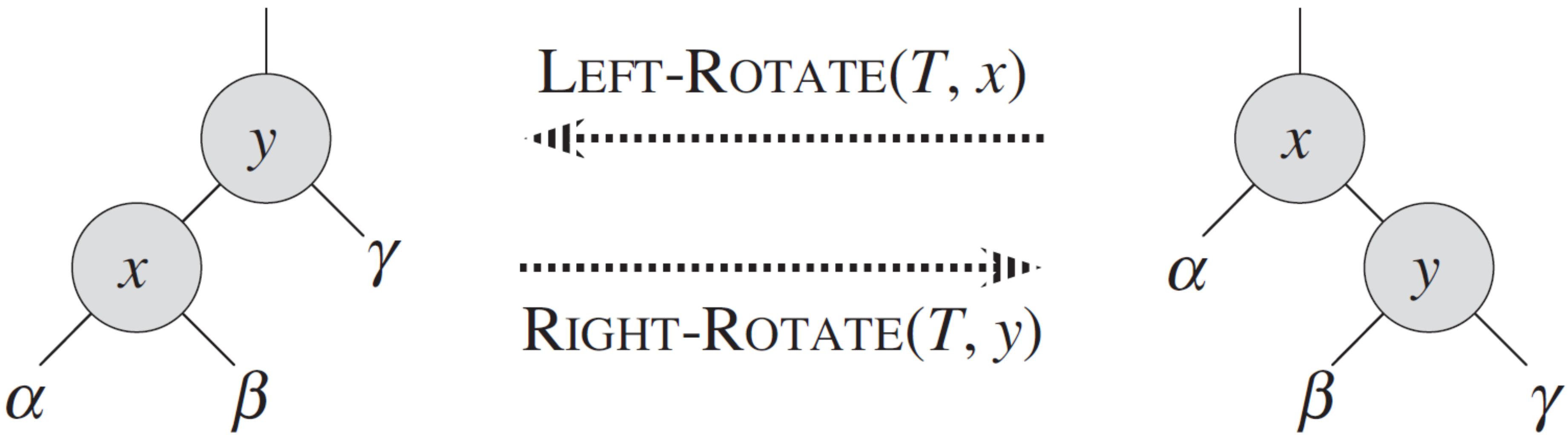
We require:

Height of children differ by at most 1

Then height is O(logN) ⇒ Balanced



Insertion: may break AVL property => need fix-up => How?



y
x
γ
α
β
LEFT-ROTATE(T, x)
RIGHT-ROTATE(T, y)
x
α
y
β
γ

```python
def _left_rotate(self, x):
    # x, y, B notation follows MIT 6.006 Lecture 6.
    # First define y and B:
    y = x.right
    B = y.left
    # Setup y:
    y.parent = x.parent
    y.left = x
    # Setup y's parent
    if y.parent is None:
        self.root = y
    elif y.parent.left is x:
        y.parent.left = y
    else:
        y.parent.right = y
    # Setup x:
    x.parent = y
    x.right = B
    # Setup B:
    if B is not None:
        B.parent = x
    self.update_all_heights_upwards(x)
```

```python
def _right_rotate(self,x):
    # First define y and B:
    y = x.left
    B = y.right
    # Setup y:
    y.parent = x.parent
    y.right = x
    # Setup y's parent
    if y.parent is None:
        self.root = y
    elif y.parent.right is x:
        y.parent.right = y
    else:
        y.parent.left = y
    # Setup x:
    x.parent = y
    x.left = B
    # Setup B:
    if B is not None:
        B.parent = x
    self.update_all_heights_upwards(x)
```

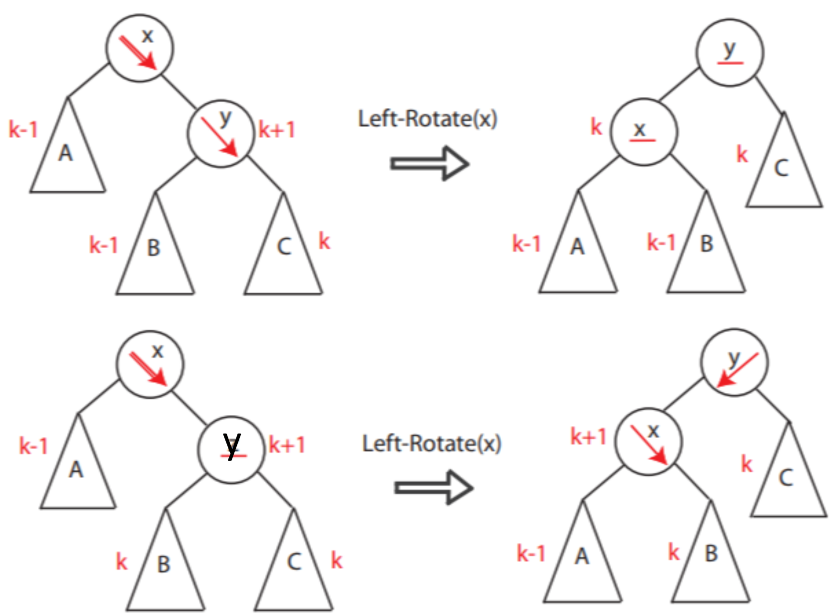
AVL Insert:

1. insert as in simple BST

2. work your way up tree, restoring AVL property (and updating heights as you go).

Each Step:

- suppose x is lowest node violating AVL
- assume x is right-heavy (left case symmetric)
- if x's right child is right-heavy or balanced: follow steps in Fig. 5



Ref: MIT 6.006 Lecture Notes

- if x's right child is right-heavy or balanced: follow steps in Fig. 5

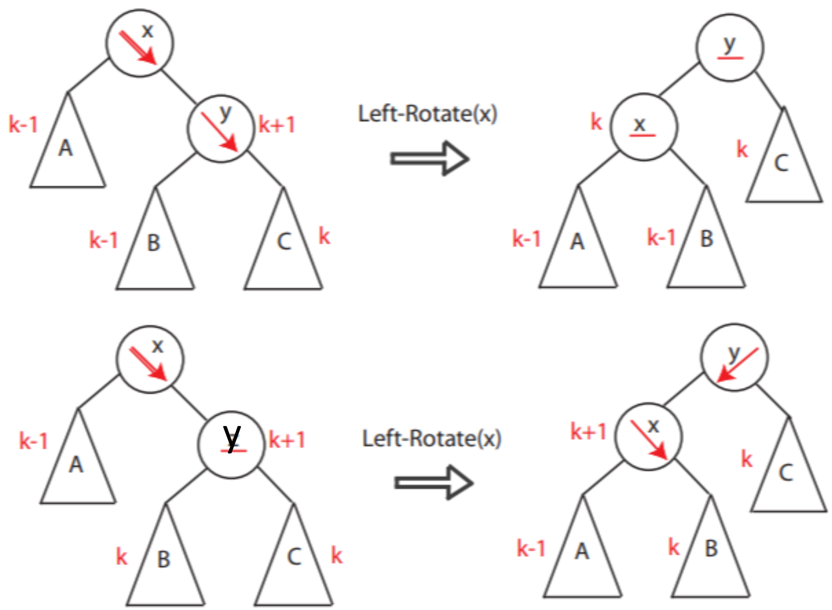
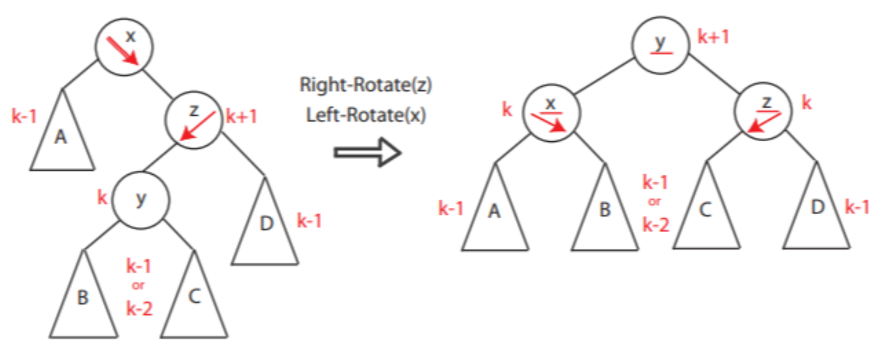


Figure 5: AVL Insert Balancing

- else: follow steps in Fig. 6



Ref: MIT 6.006

Lecture Notes

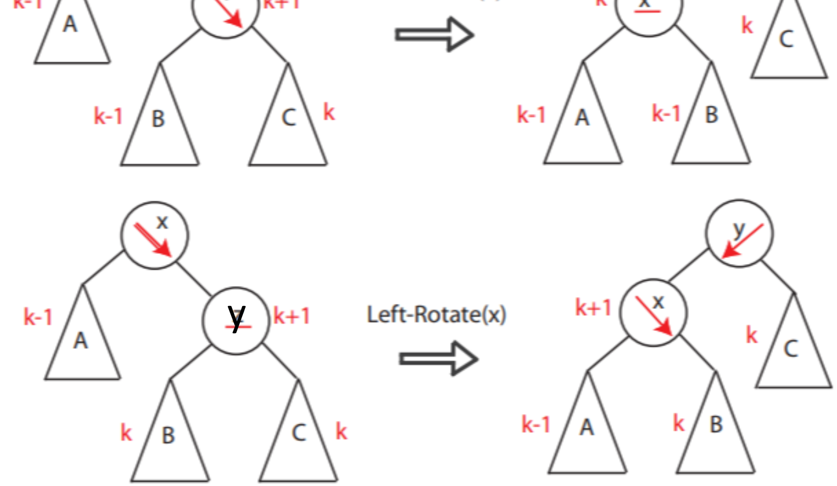


Figure 5: AVL Insert Balancing

- else: follow steps in Fig. 6

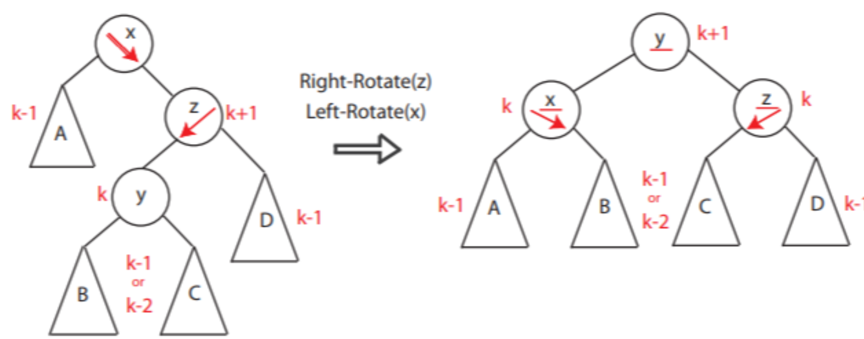


Figure 6: AVL Insert Balancing

- then continue up to x's grandparent, greatgrandparent ...

Ref: MIT 6.006 Lecture Notes

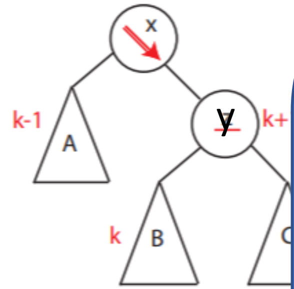


Figure 5

Note: only need to update height, no need for rotation for grandparent, …

At most 2 rotations needed to get a AVL tree balanced (all shown here).
[Proof: note that height of subtree restored after insertion + rotation]

In MIT 6.006, after rotation, only the height of x, y, z are updated, in rotation module. So they have to "continue up to grandparent…"

In our sample code, update height is an independent module, so no need to recursively check grandparents, …

- else: follow steps in Fig. 6

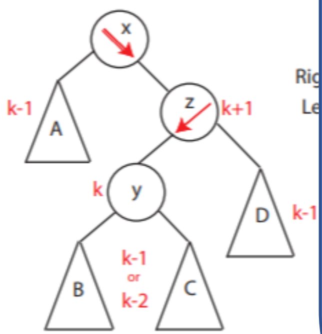


Figure 6: A

- then continue up to x's grandparent, greatgrandparent …

Ref: MIT 6.006 Lecture Notes

```python
import binary_search_tree

class Node(binary_search_tree.Node):
    pass

class AVLTree(binary_search_tree.BinarySearchTree):
    def insert(self, new_node, node = 0):
        super().insert(new_node, node)
        self.check_fix_AVL(new_node.parent)
        return new_node

    def update_all_heights_upwards(self, node):
        node.update_height()
        if node is not self.root:
            self.update_all_heights_upwards(node.parent)
```

```python
def check_fix_AVL(self, node):
    if node is None:
        return
    if abs(node.balance()) < 2:
        self.check_fix_AVL(node.parent)
        return
    if node.balance() == 2: # right too heavy
        if node.right.balance() >= 0:
            self._left_rotate(node)
        else:
            self._right_rotate(node.right)
            self._left_rotate(node)
    else: # node.balance() == -2, left too heavy
        if node.left.balance() <= 0:
            self._right_rotate(node)
        else:
            self._left_rotate(node.left)
            self._right_rotate(node)
    self.check_fix_AVL(node.parent)
```

AVL deletion:

Let w be the node to be deleted

1) Perform standard BST delete for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case)

b) y is left child of z and x is right child of y (Left Right Case)

c) y is right child of z and x is right child of y (Right Right Case)

d) y is right child of z and x is left child of y (Right Left Case)

Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well (See this video lecture for proof)

AVL Tree | Set 2 (Deletion) - GeeksforGeeks

a) Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

b) Left Right Case

```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

c) Right Right Case

```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

d) Right Left Case

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \    - - - - - - - -> / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                              /  \
T2   T3                           T3   T4
```

[AVL Tree | Set 2 (Deletion) - GeeksforGeeks]

AVL tree vs heap

	Type	AVL tree	Heap
Insert	Average	O(logN)	**O(1)**
Insert	Worst	O(logN)	O(logN)
Find	Worst	O(logN)	O(N)
Find min/max	Worst	O(logN) [note1]	O(1) / O(N)
Create	Worst	O (N logN)	O(N)
Delete	Worst	O(logN)	O(logN)
Space usage		N x sizeof(Node)	**N x sizeof(data)**

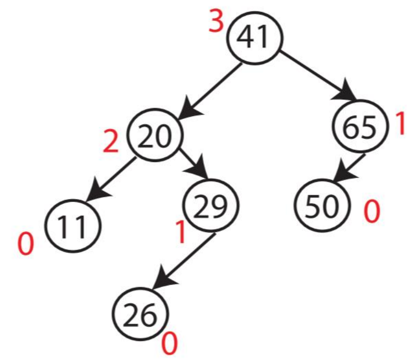
[note1] Find min/max of AVL tree can be improved to O(1) by caching left-most and right-most elements.

See here for a nice discussion (with caution of online discussions)

AVL tree:

require balance → small (but may not be minimal) height

max height $\sim 1.44 \log_2 n$



Note:

Frequent rotations (once not balanced)

Can we relax it, as long as still $h \simeq O(\log_2 n)$, to insert/delete faster?

Idea:

- Construct a balanced tree as the **main part**
- Add a limited number of **other nodes** (distinguish & limit the #)

Section 4: Red Black Tree (RBT)

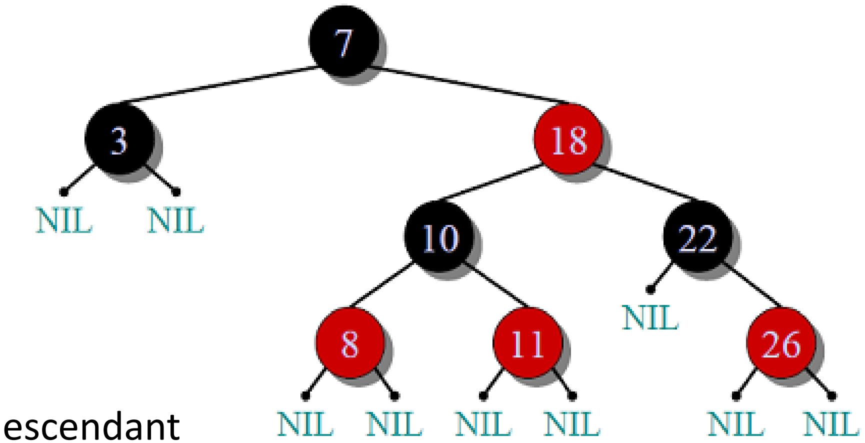
Idea:

- Construct a balanced tree as the **main part**
- Add a limited number of **other nodes** (distinguish & limit the #)

How to realize?

- Each node colored red or black
- Root & NIL are black
- Balanced black:

For each node, all simple path from the node to descendant leaves contains the same number of black nodes.



- Limited red: If a node is red, then both its children are black

Height: $h(\text{Black}) \sim O(\log N)$, $h \lesssim 2h(\text{Black}) \sim O(\log N)$

Insertion: what to take care of?

RBT Insertion:

Step 1: BST insertion

(here iteration version)

(T.nil instead of NIL)

Step 2: Color new node red

Step 3: Check & fix properties

RB-INSERT(T, z)

```
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

(CLRS)

How to fixup when inserting a node z?

What to fixup? i.e., which property may be violated?

- Each node colored red or black (not violated)
- Root & NIL are black (iff z is the root, just recolor root black **in the end**)
- Balanced black (not violated)
- Limited red
    - Can z's children be red? (no, its children is T.nil, black)
      (note: when we fix upwards, we will keep z's children black)
    - Can z's parent be red? (check & fix needed)

What to do if z’s parent is red?

Assume z’s parent is a left child (otherwise symmetric)

- Case 1: z’s uncle is red
- Case 2: z’s uncle is black, and z is a right child
- Case 3: z’s uncle is black, and z is a left child

Case 1: z's uncle is red:

(1) parent ⇒ Black, (2) uncle ⇒ Black, (3) grandpa ⇒ Red

(4) check grandpa (as the new z)

Note:

(1) $\alpha, \beta, \gamma, \delta, \epsilon$ has the same black-height ⇒ still have "balanced black"

(2) No rotation needed (before checking new z)

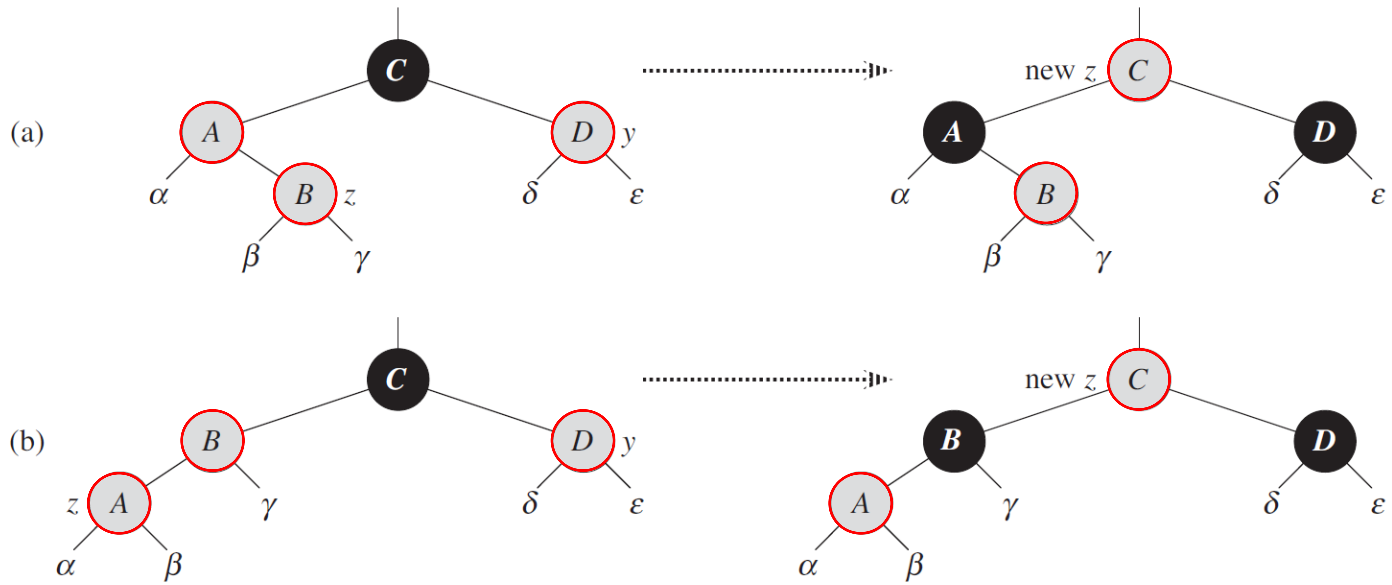


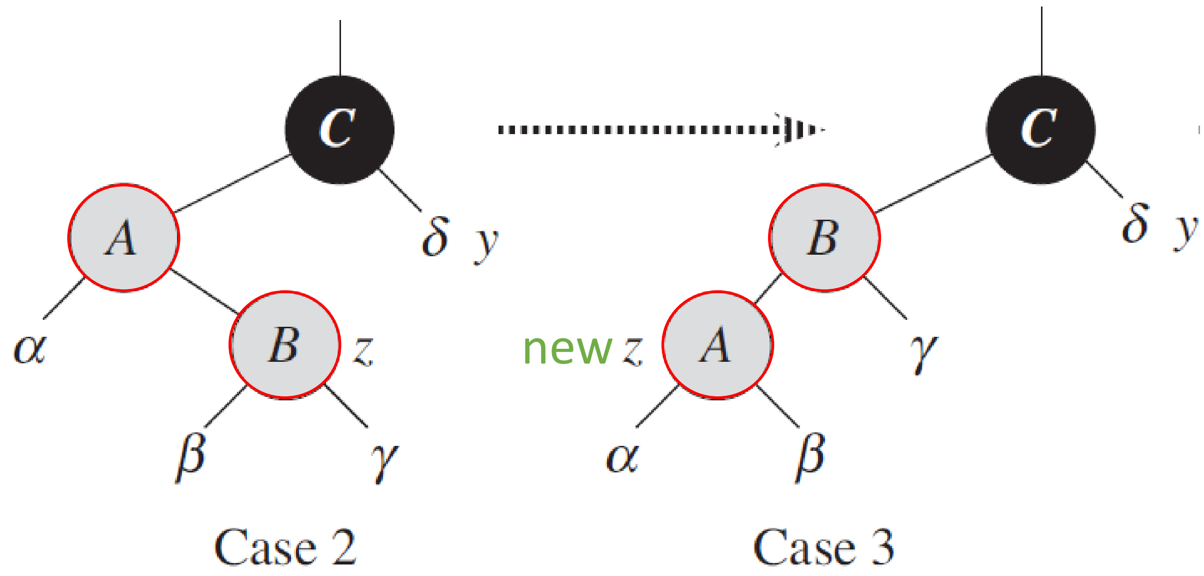
Figure 13.5 Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since z and its parent $z.p$ are both red. We take the same action whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $z.p.p$ as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.

(CLRS)

Case 2: z's uncle is black, and z is a right child

left-rotate z's parent, to convert it to Case 3

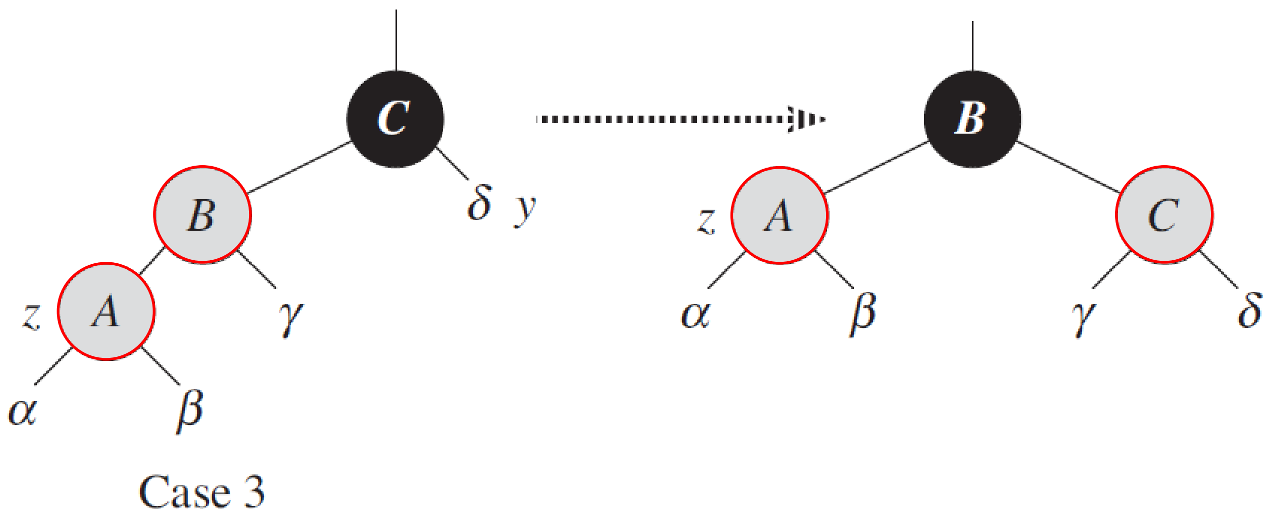
with new z = z's original parent (now left child)



(CLRS)

Case 3: z’s uncle is black, and z is a left child

(1) parent ⇒ black, (2) grandpa ⇒ Red, (3) right-rotate grandpa



(CLRS)

左爹诀
红叔染色去查爷
黑叔右子转查爹
黑叔左子爷转染
爹黑就染根节点

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right          // y is defined as z's uncle
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK      // Case 1: z's uncle is red
            z.p.p.color = RED    // then recolor and fixup z.p.p
            z = z.p.p
        else if z == z.p.right
                z = z.p
                LEFT-ROTATE(T, z)        // Case 2: z's uncle is black
            z.p.color = BLACK            // && z is a right child
Case 3      z.p.p.color = RED            // then left-rotate to convert
            RIGHT-ROTATE(T, z.p.p)       // it to Case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

(CLRS)

左爹诀

红叔染色去查爷

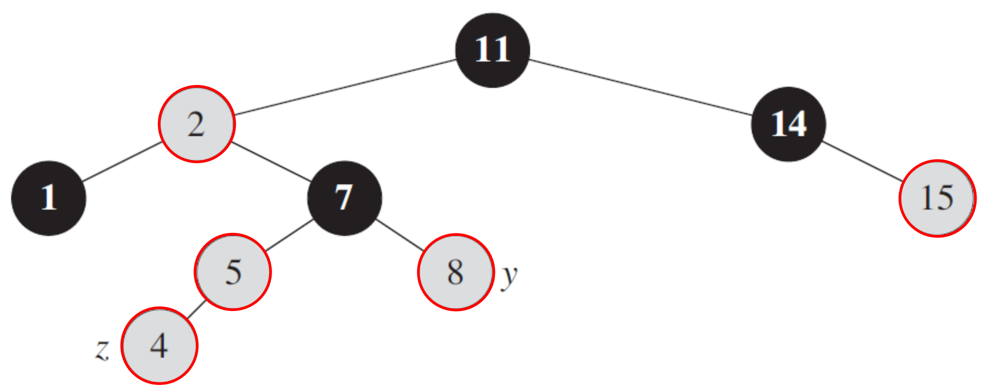
黑叔右子转查爹

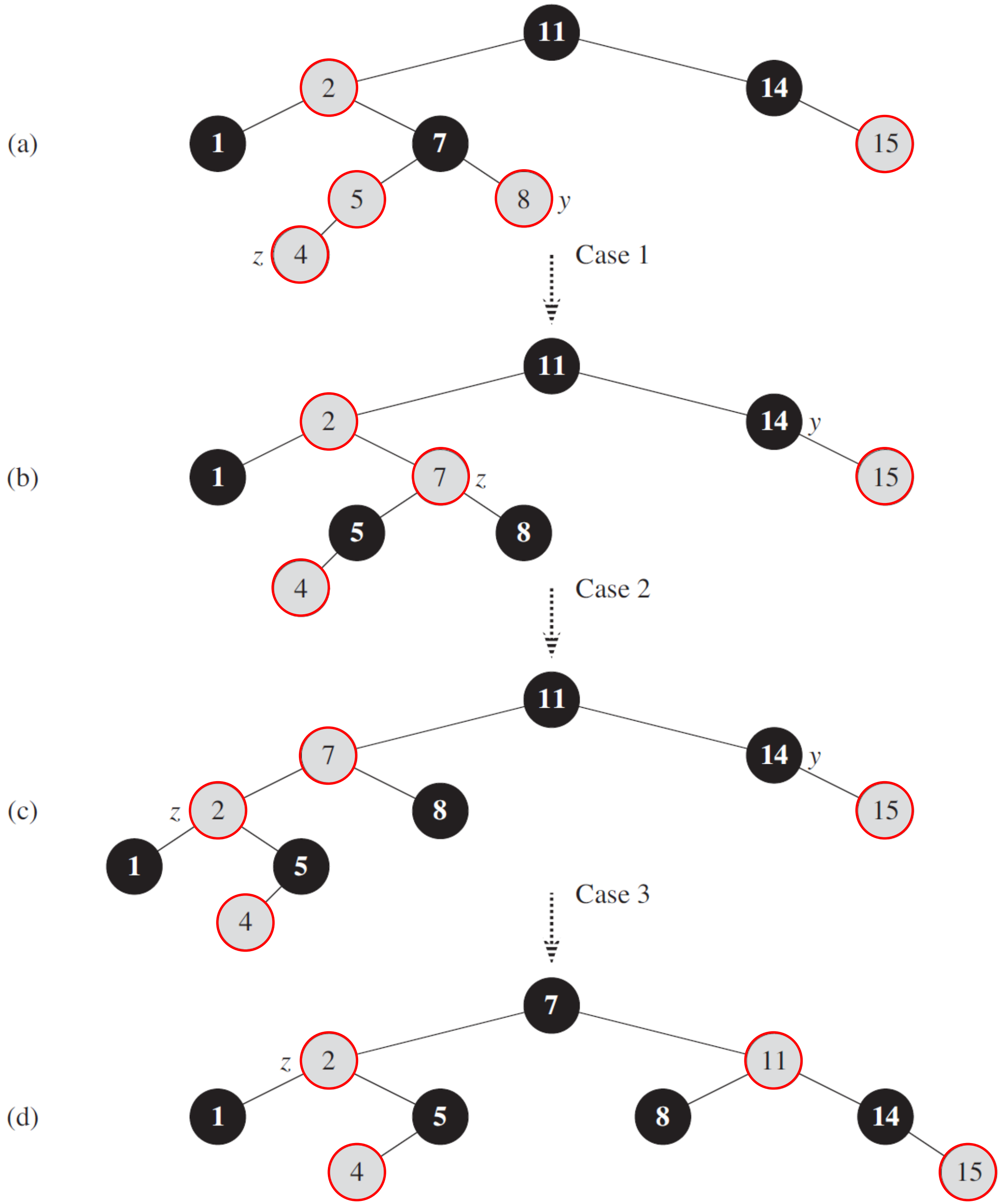
黑叔左子爷转染

爹黑就染根节点

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
                z = z.p
                LEFT-ROTATE(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

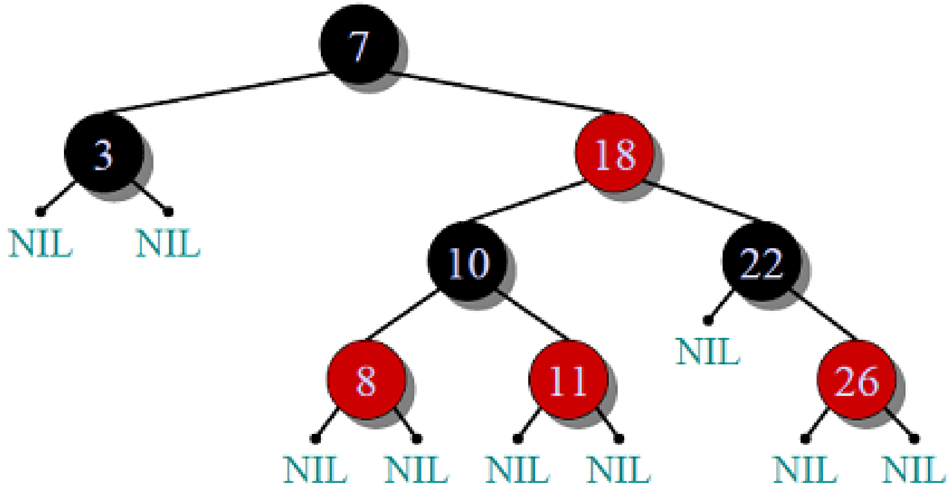
Exercise




(a)
Case 1
(b)
Case 2
(c)
Case 3
(d)

Comparison between AVL and red-black tree:

- Both AVL tree and red-black tree has O(lgN) insertion and lookup.
- AVL tree is better balanced and better if more lookup operations.
- Red-black tree is faster in insertions, and needs a little less storage space.



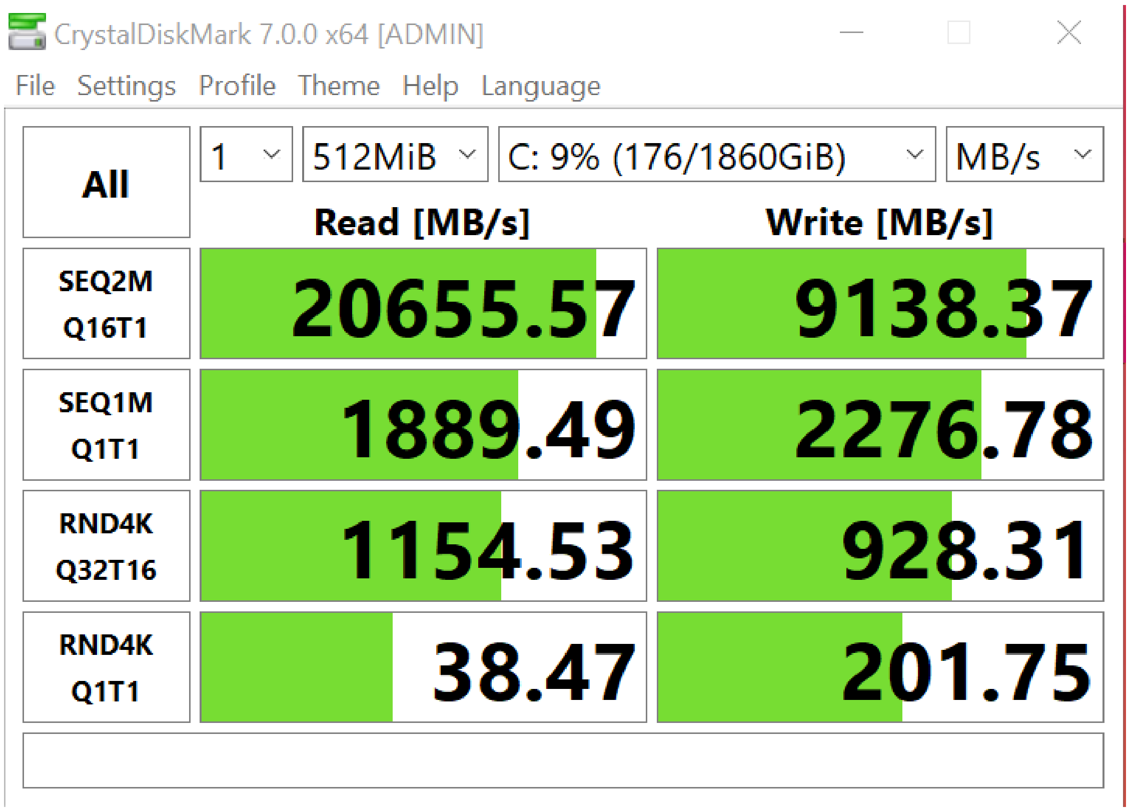
Section 5: Other tree related topics

Can we get even better real-world search performance?

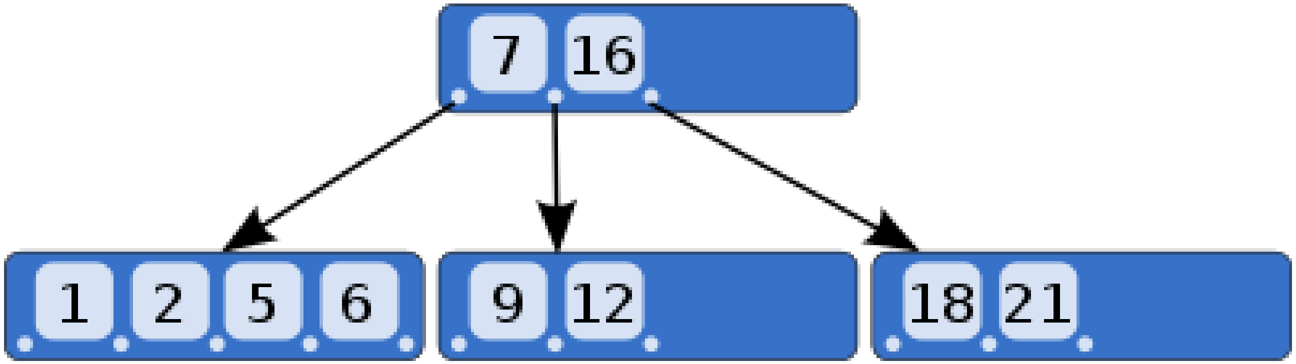
Imagine:

searching in PBs of indexed data

only small part of the tree fits in memory, rest on disk



B-tree: generalization of balanced BST



Summary: BST, AVL, RBT