Module 4 - Trees

SIT320 – Advanced Algorithms

Dr. Nayyar Zaidi

Motivations for BST

1. The punchline is **important**:
   - A data structure with O(log(n)) INSERT/DELETE/SEARCH

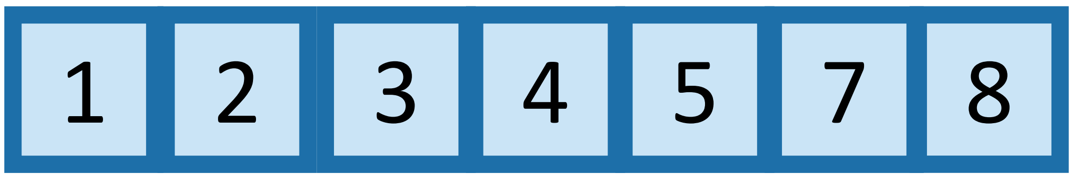
2. The idea behind AVL Trees and **Red-Black Trees** is clever
   - It's good to be exposed to clever ideas.
   - Also it's just aesthetically pleasing.

Some data structures

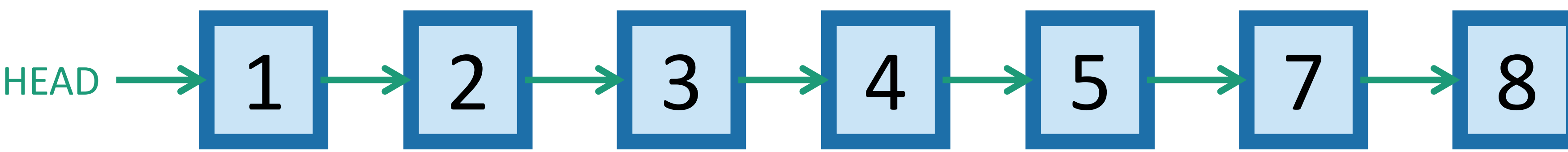
for storing objects like 

(aka, nodes with keys)

- (Sorted) arrays:



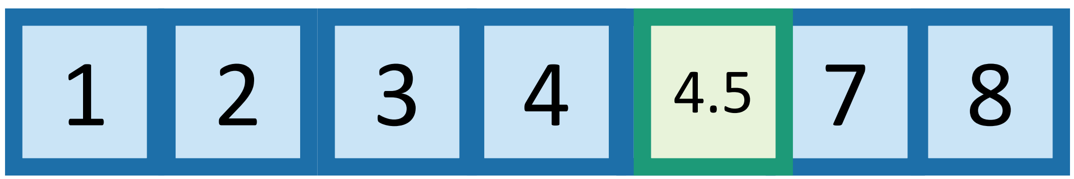
- (Sorted) linked lists:



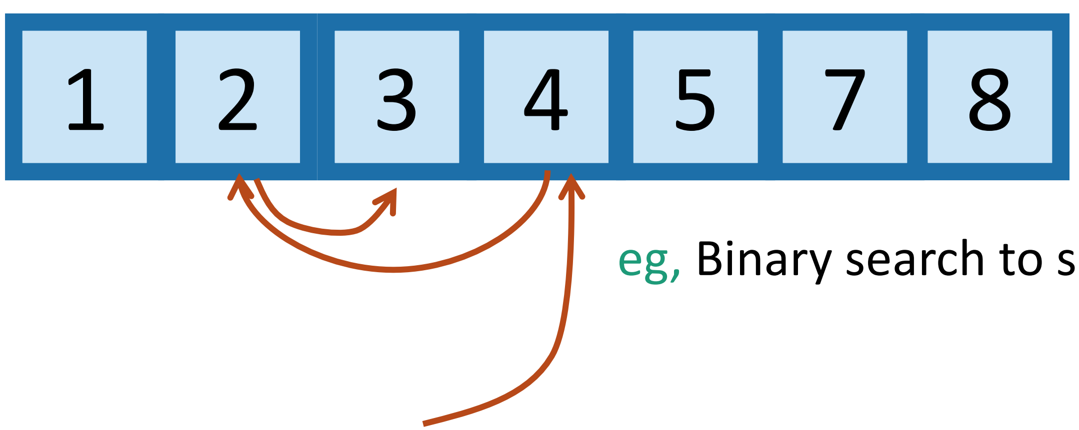
- Some basic operations:
  - INSERT, DELETE, SEARCH

Sorted Arrays

- O(n) INSERT/DELETE:

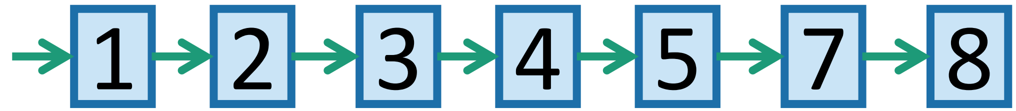


- O(log(n)) SEARCH:

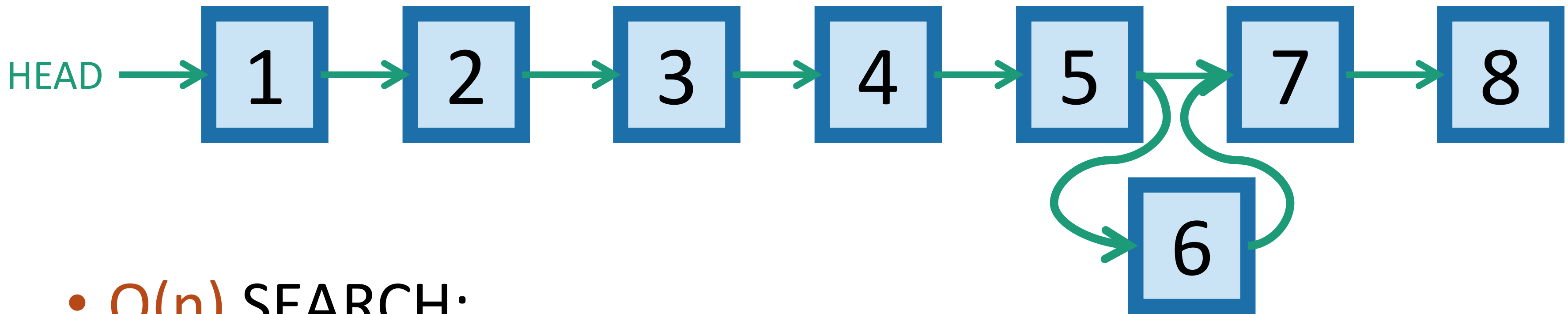


eg, Binary search to see if 3 is in A.

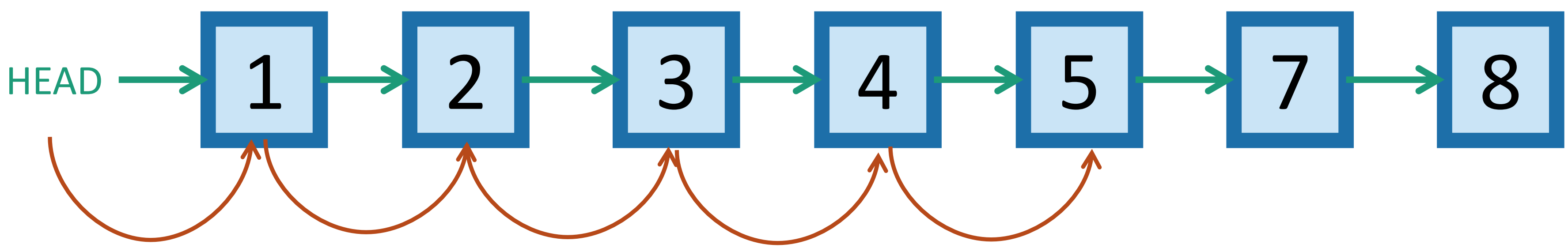
Sorted linked lists



- O(1) INSERT/DELETE:
  - (assuming we have a pointer to the location of the insert/delete)



- O(n) SEARCH:



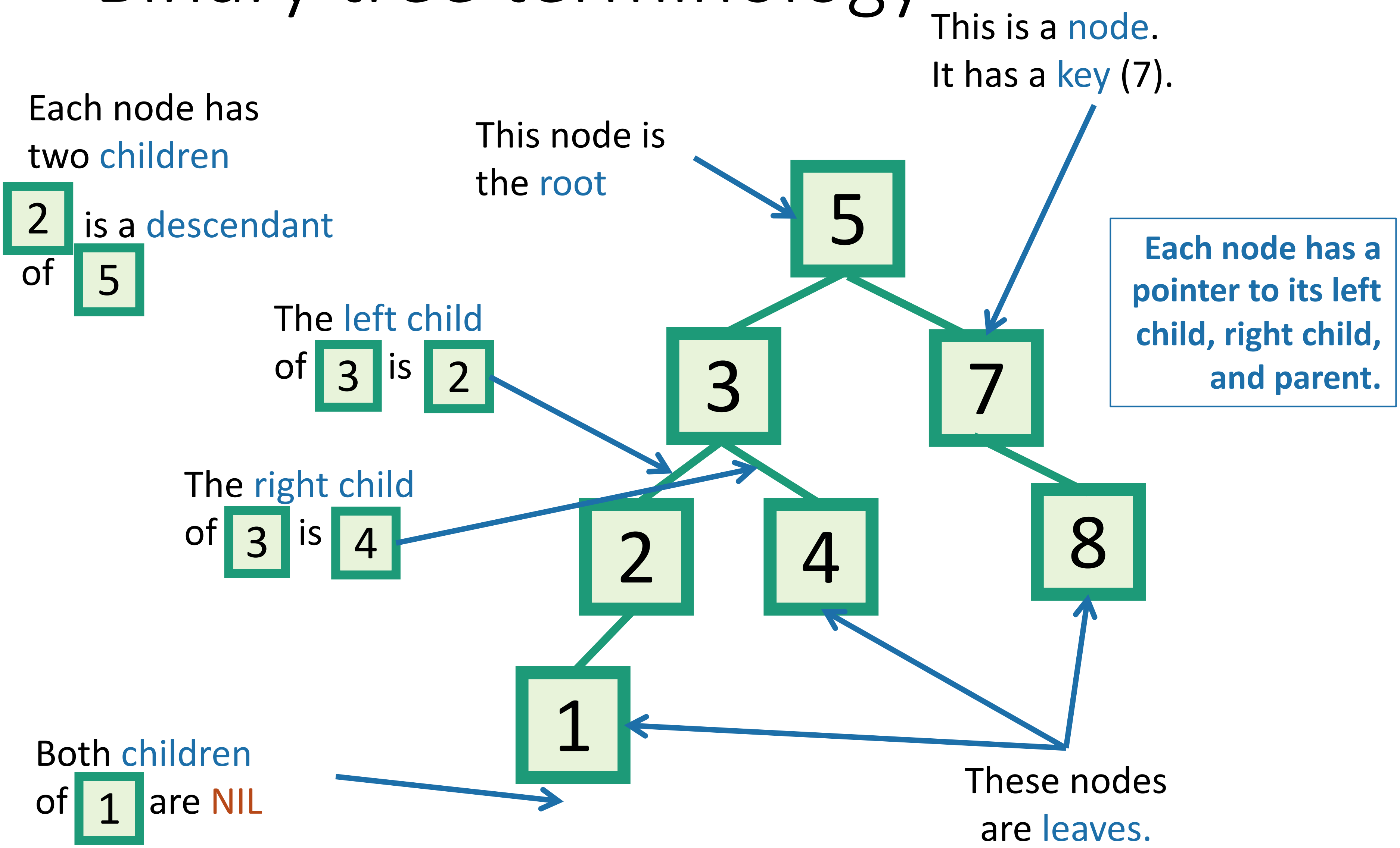
Motivation for Binary Search Trees

TODAY!

	Sorted Arrays	Linked Lists	Binary Search Trees*
Search	O(log(n)) 😃	O(n) ☹️	O(log(n)) 😃
Insert/Delete	O(n) ☹️	O(1) 😃	O(log(n)) 😃

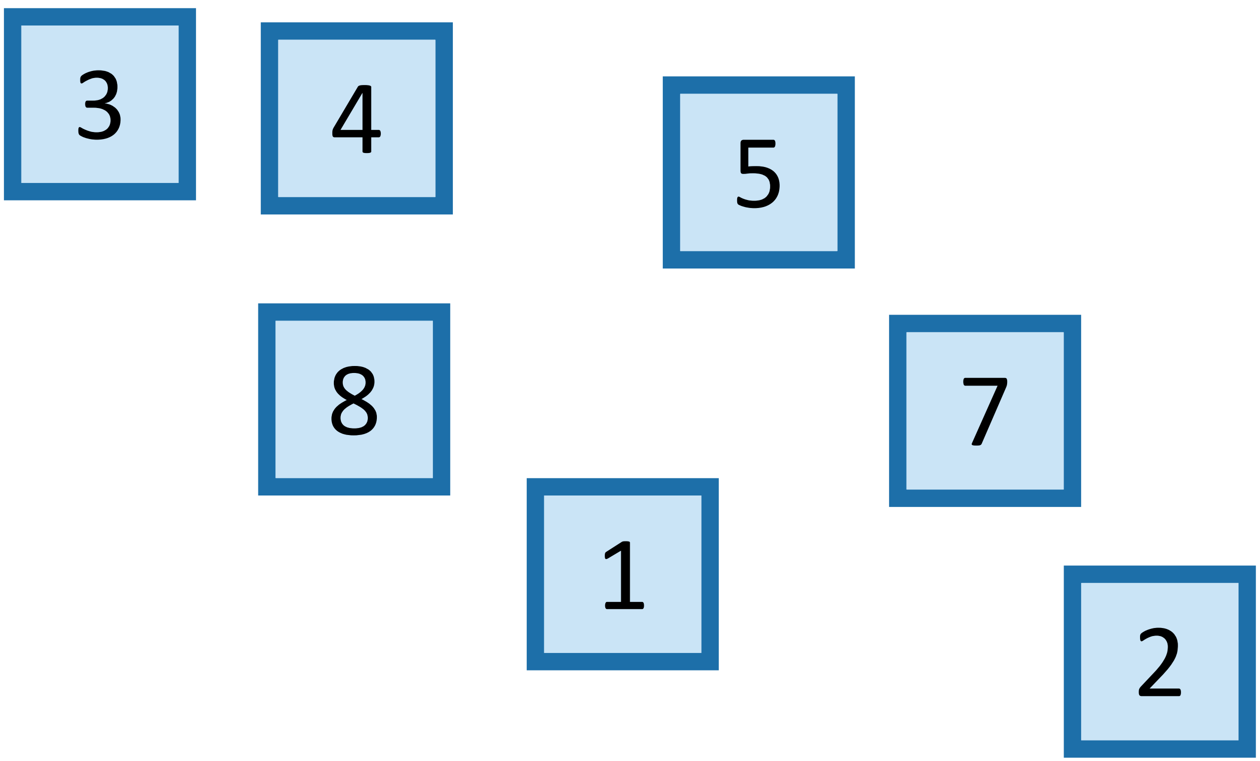
For today all keys are distinct.

Binary tree terminology



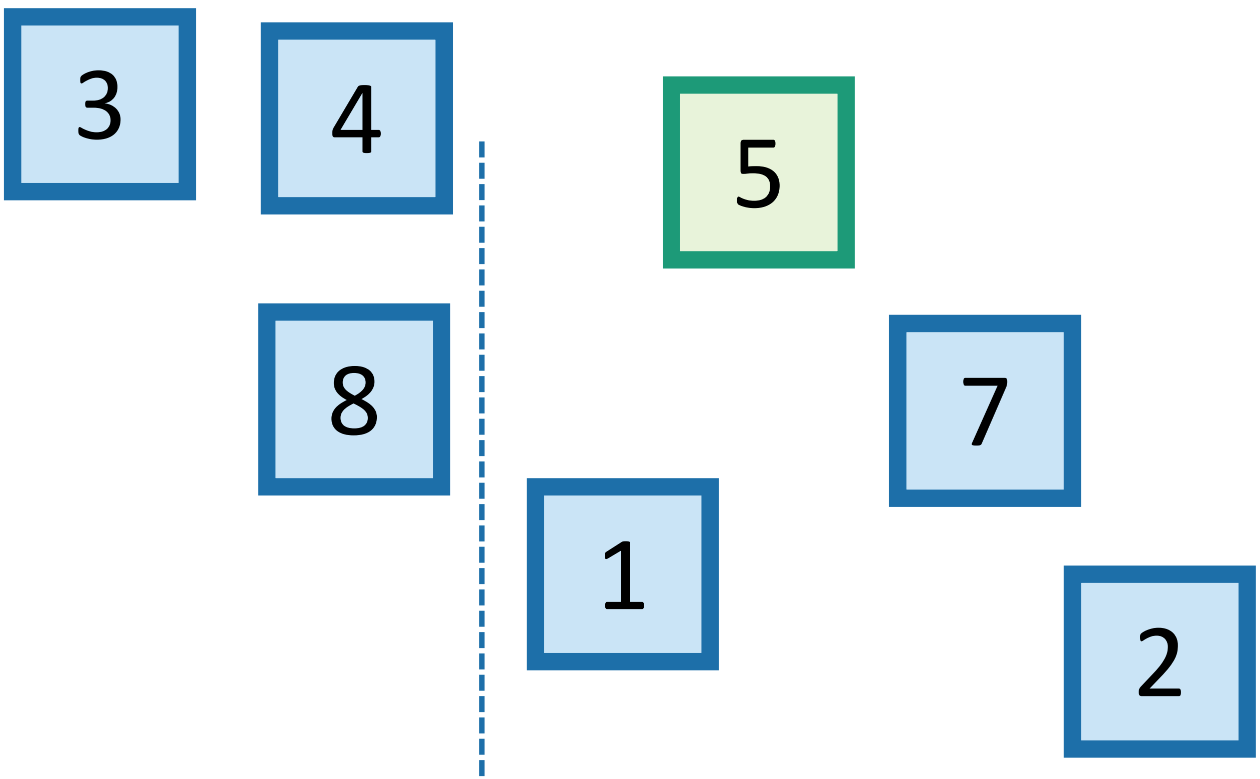
Binary Search Trees

- It's a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



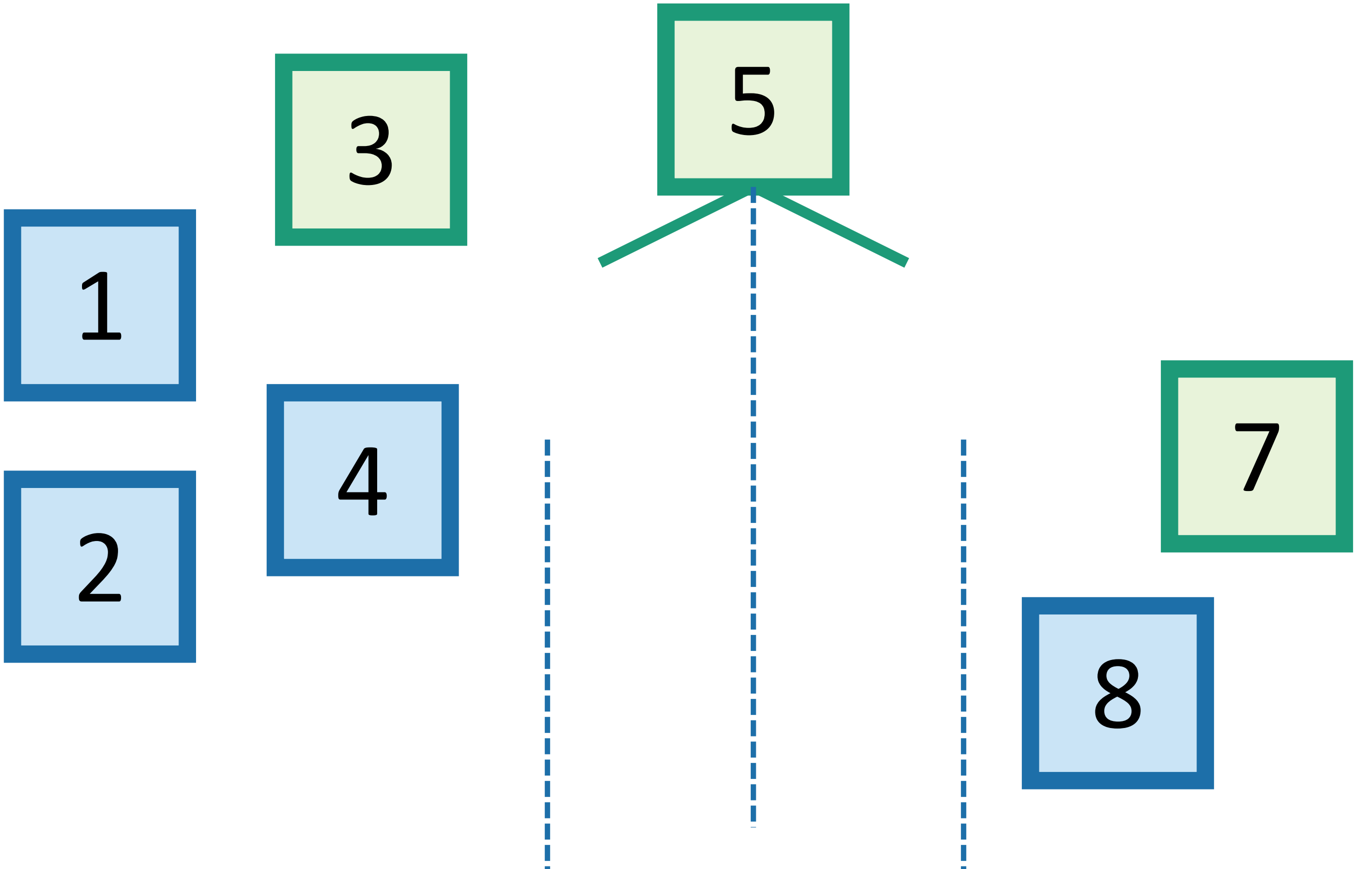
Binary Search Trees

- It's a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



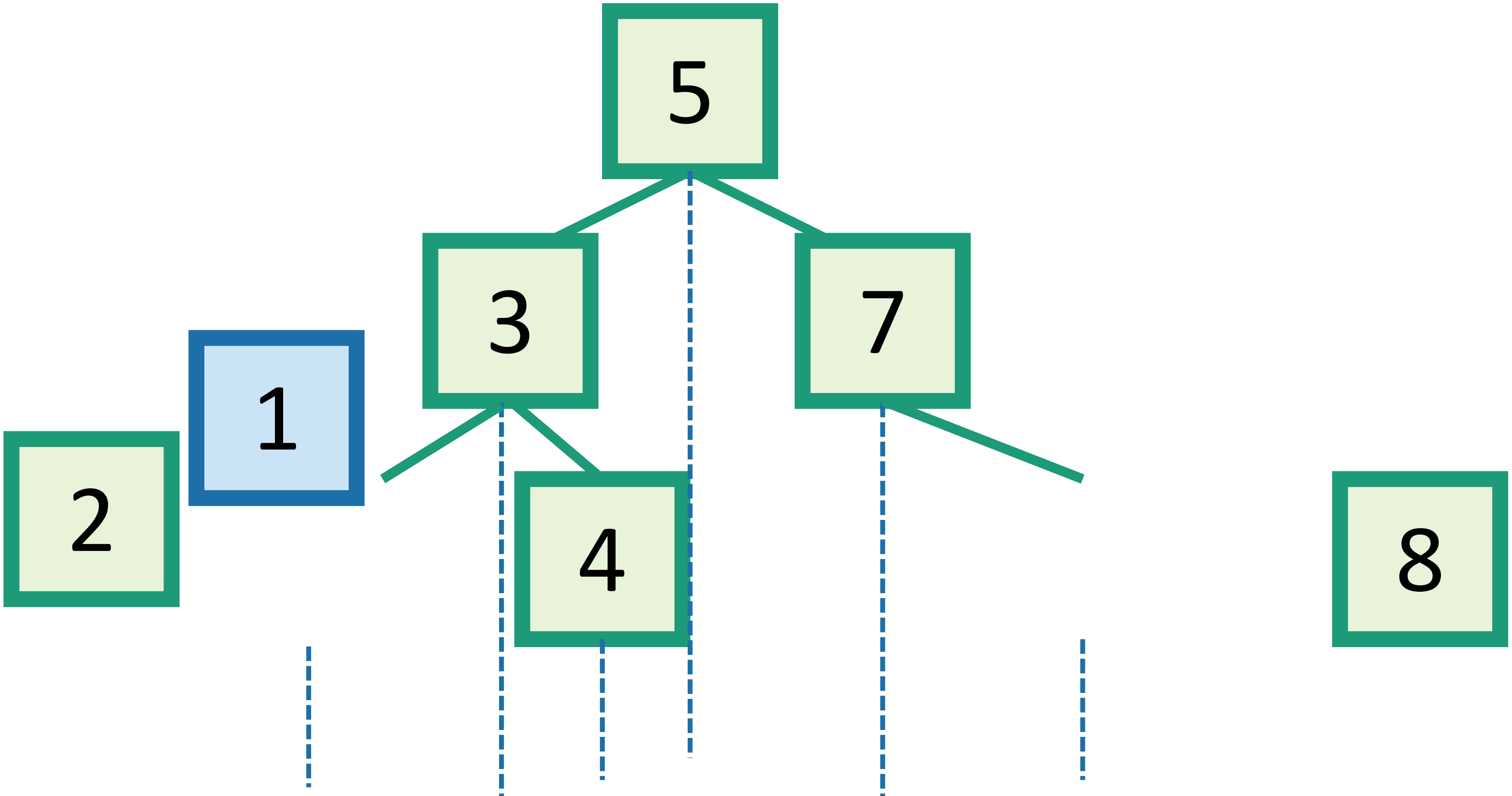
Binary Search Trees

- It's a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



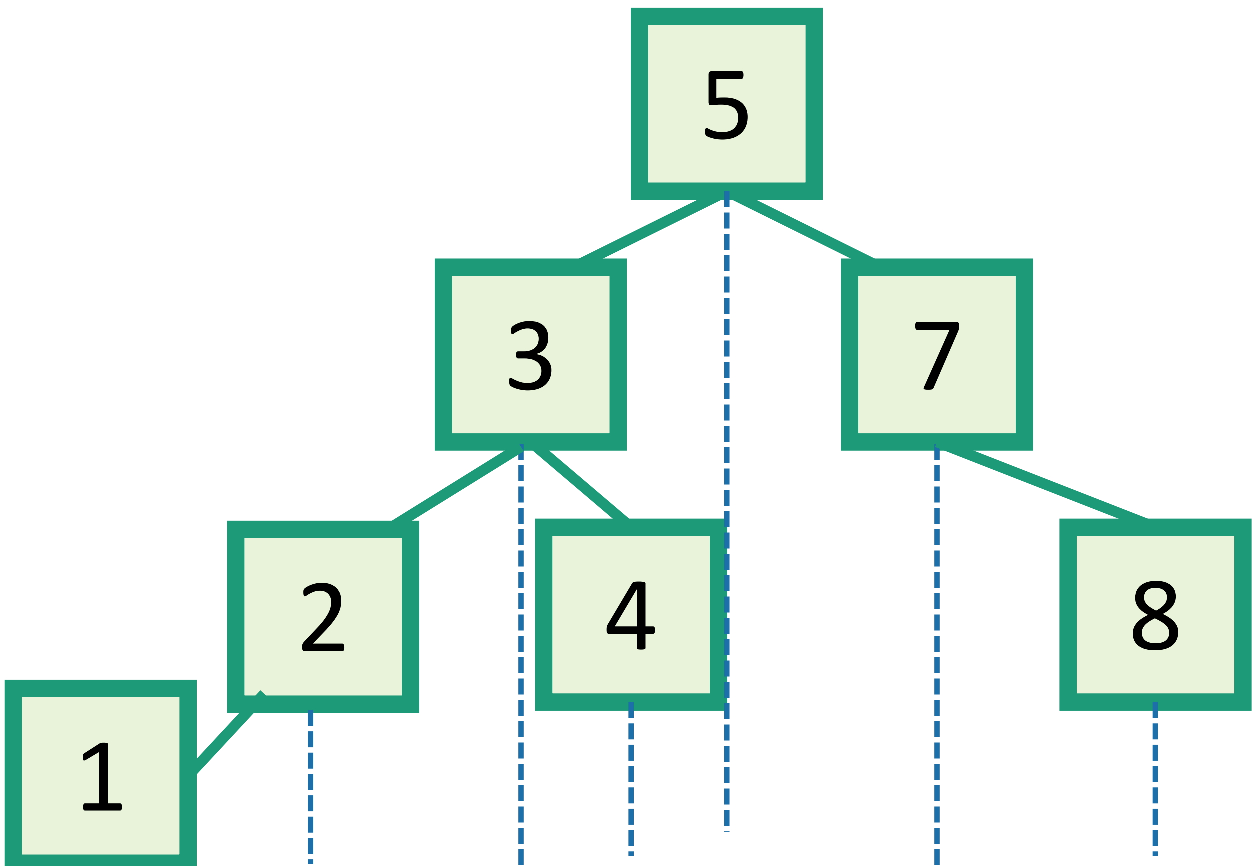
Binary Search Trees

- It's a binary tree so that:
    - Every LEFT descendant of a node has key less than that node.
    - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



Binary Search Trees

- It's a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



Q: Is this the only binary search tree I could possibly build with these values?

A: **No.** I made choices about which nodes to choose when. Any choices would have been fine.

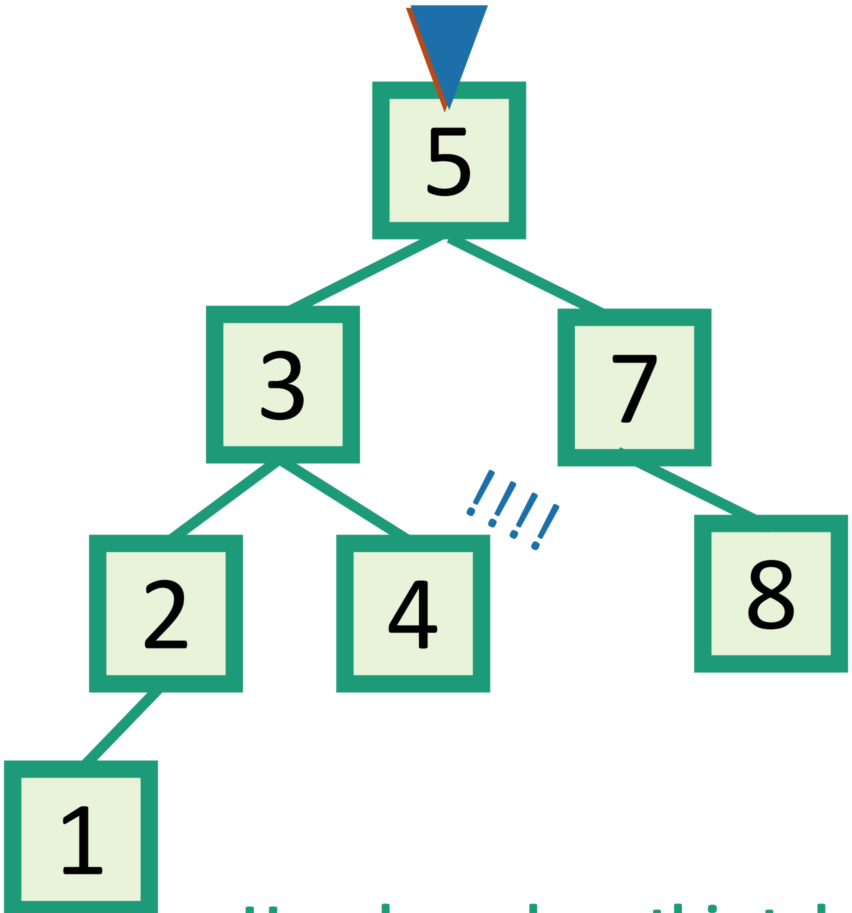
Remember the goal

Fast SEARCH/INSERT/DELETE

Can we do these?

SEARCH in a Binary Search Tree

definition by example



EXAMPLE: Search for 4.

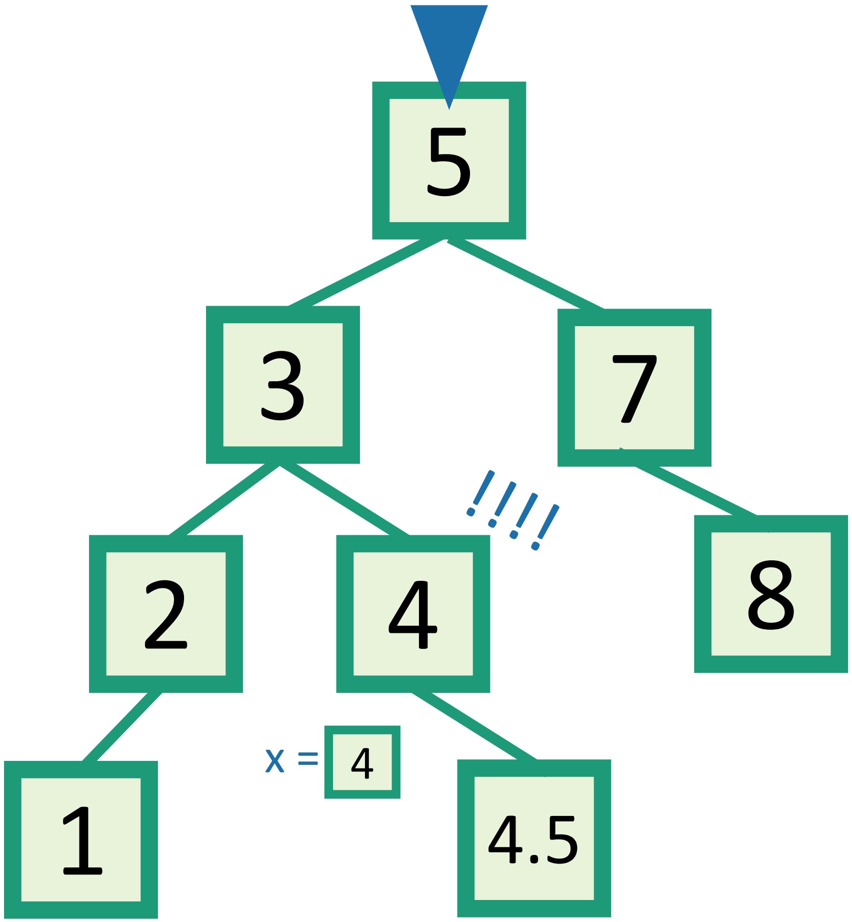
EXAMPLE: Search for 4.5

- It turns out it will be convenient to **return 4** in this case
- (that is, **return** the last node before we went off the tree)

How long does this take?

O(length of longest path) = O(height)

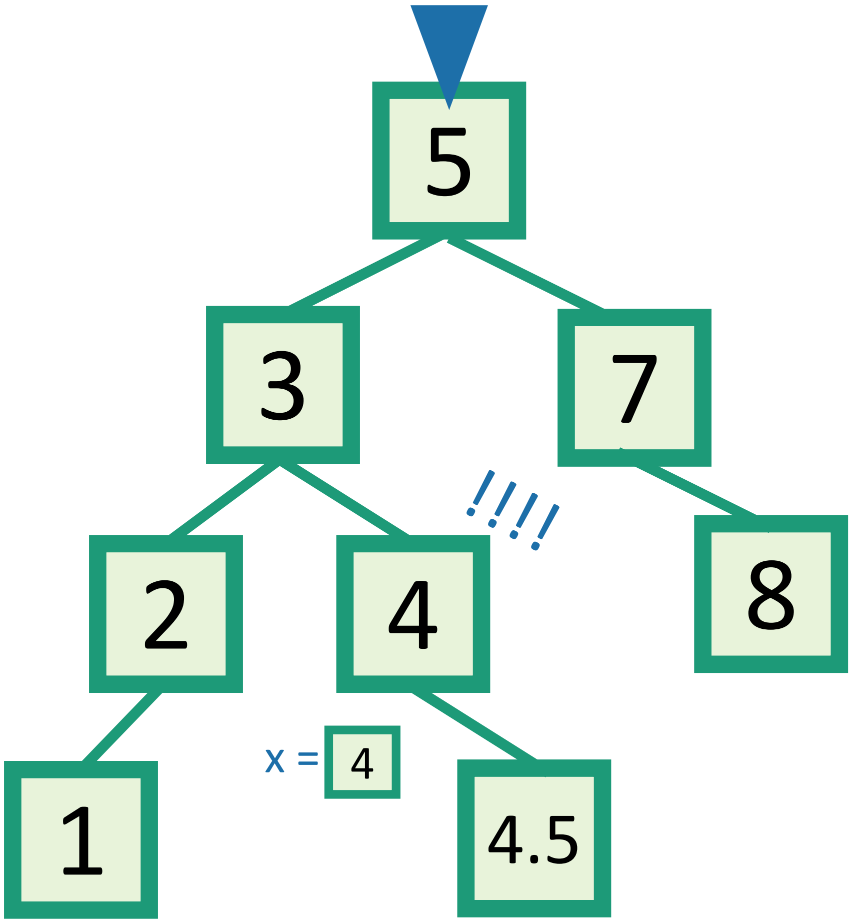
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

- INSERT(key):
  - x = SEARCH(key)
  - **Insert** a new node with desired key at x…

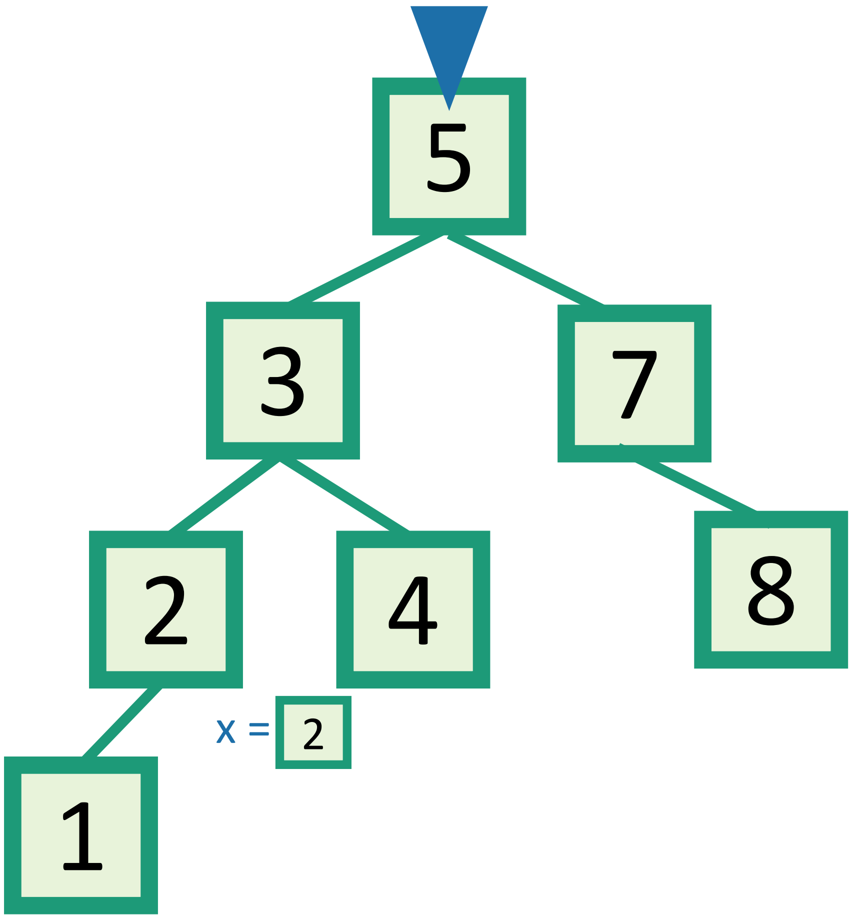
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

- INSERT(key):
  - x = SEARCH(key)
  - **if** key > x.key:
    - Make a new node with the correct key, and put it as the right child of x.
  - **if** key < x.key:
    - Make a new node with the correct key, and put it as the left child of x.
  - **if** x.key == key:
    - **return**

DELETE in a Binary Search Tree



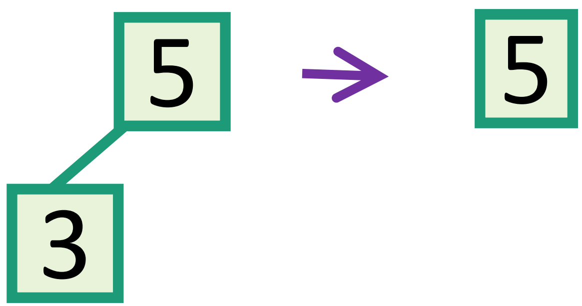
EXAMPLE: Delete 2

- DELETE(key):
  - x = SEARCH(key)
  - **if** x.key == key:
    - ….delete x….

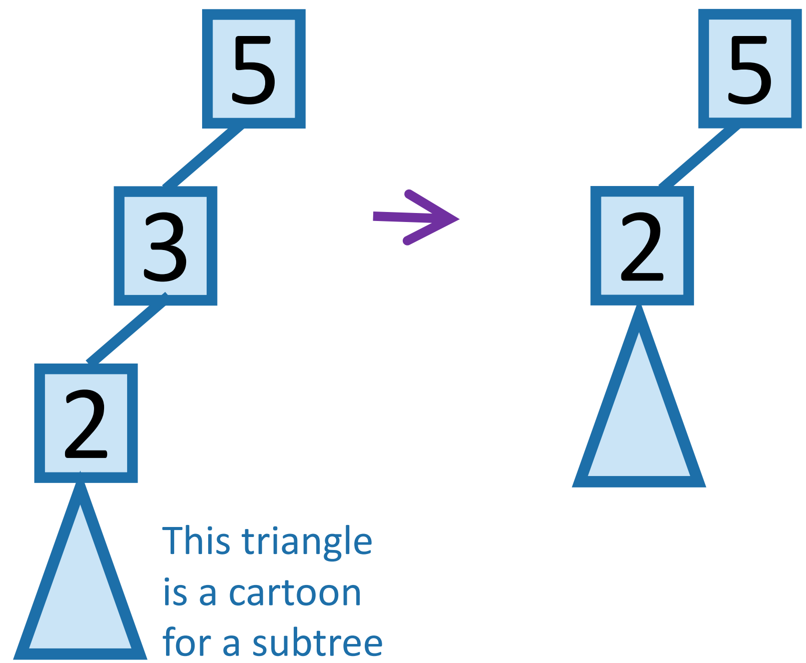
DELETE in a Binary Search Tree

several cases (by example)

say we want to delete 3



Case 1: if 3 is a leaf, just delete it.

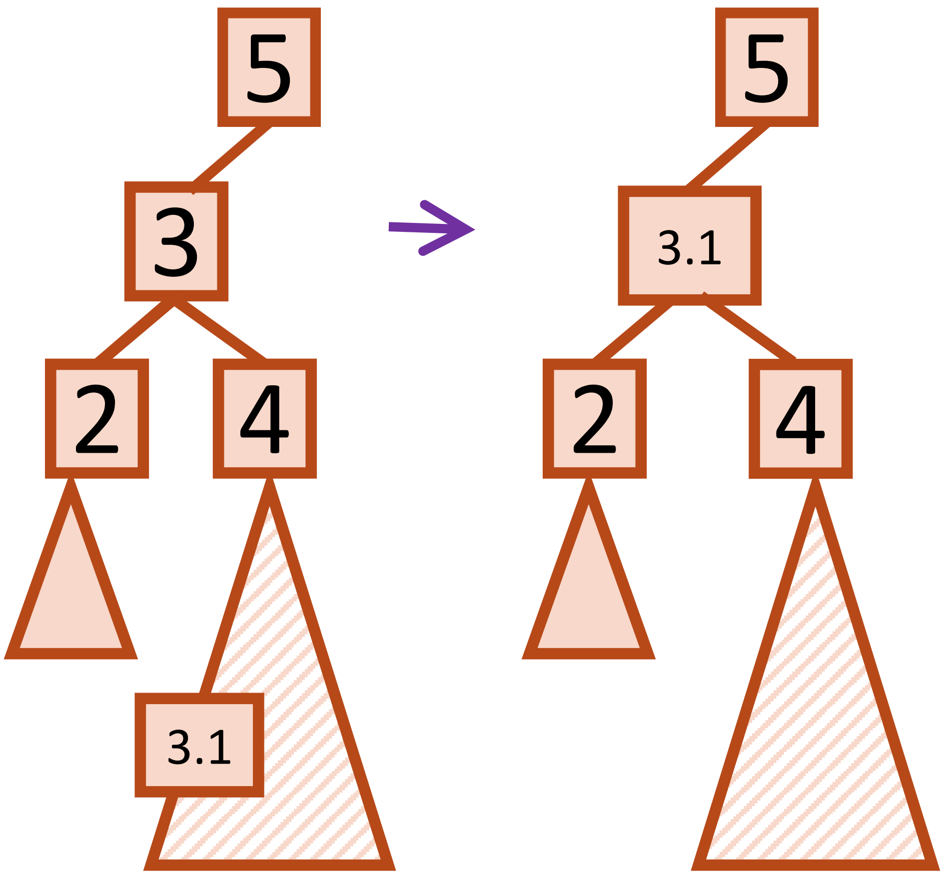


Case 2: if 3 has just one child, move that up.

DELETE in a Binary Search Tree

ctd.

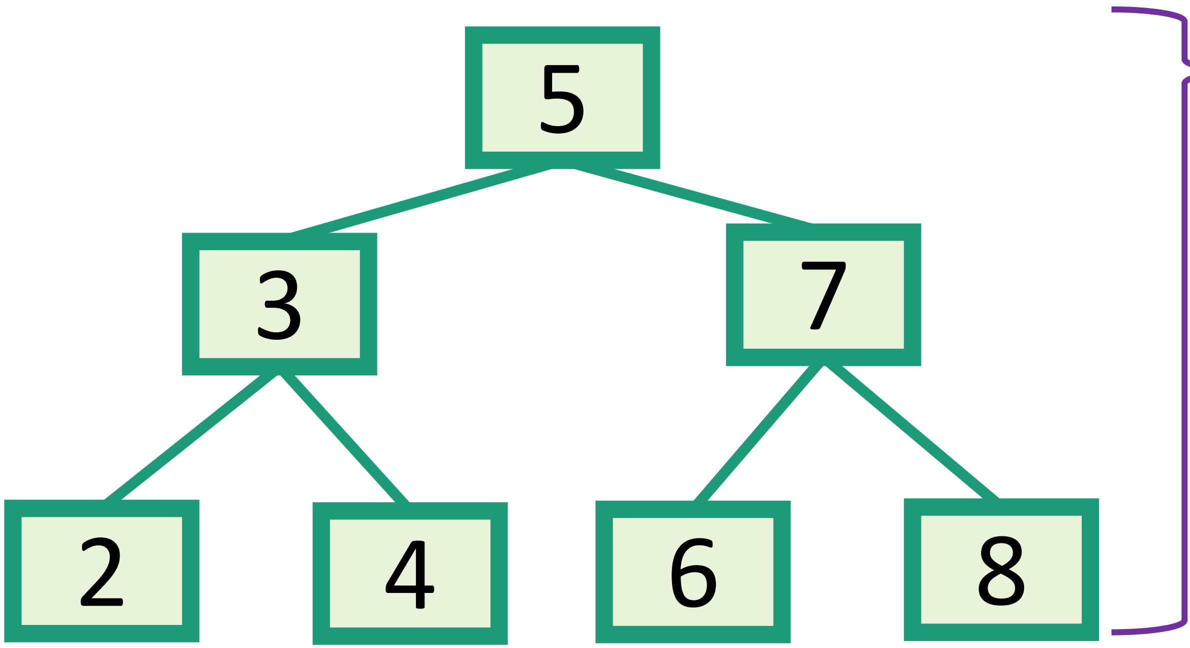
Case 3: if 3 has two children,
replace 3 with it's immediate successor.
(aka, next biggest thing after 3)



- Does this maintain the BST property?
  - Yes.
- How do we find the immediate successor?
  - SEARCH for 3 in the subtree under 3.right
- How do we remove it when we find it?
  - If [3.1] has 0 or 1 children, do one of the previous cases.
- What if [3.1] has two children?
  - It doesn't.

How long do these operations take?

- SEARCH is the big one.
  - Everything else just calls SEARCH and then does some small O(1)-time operation.

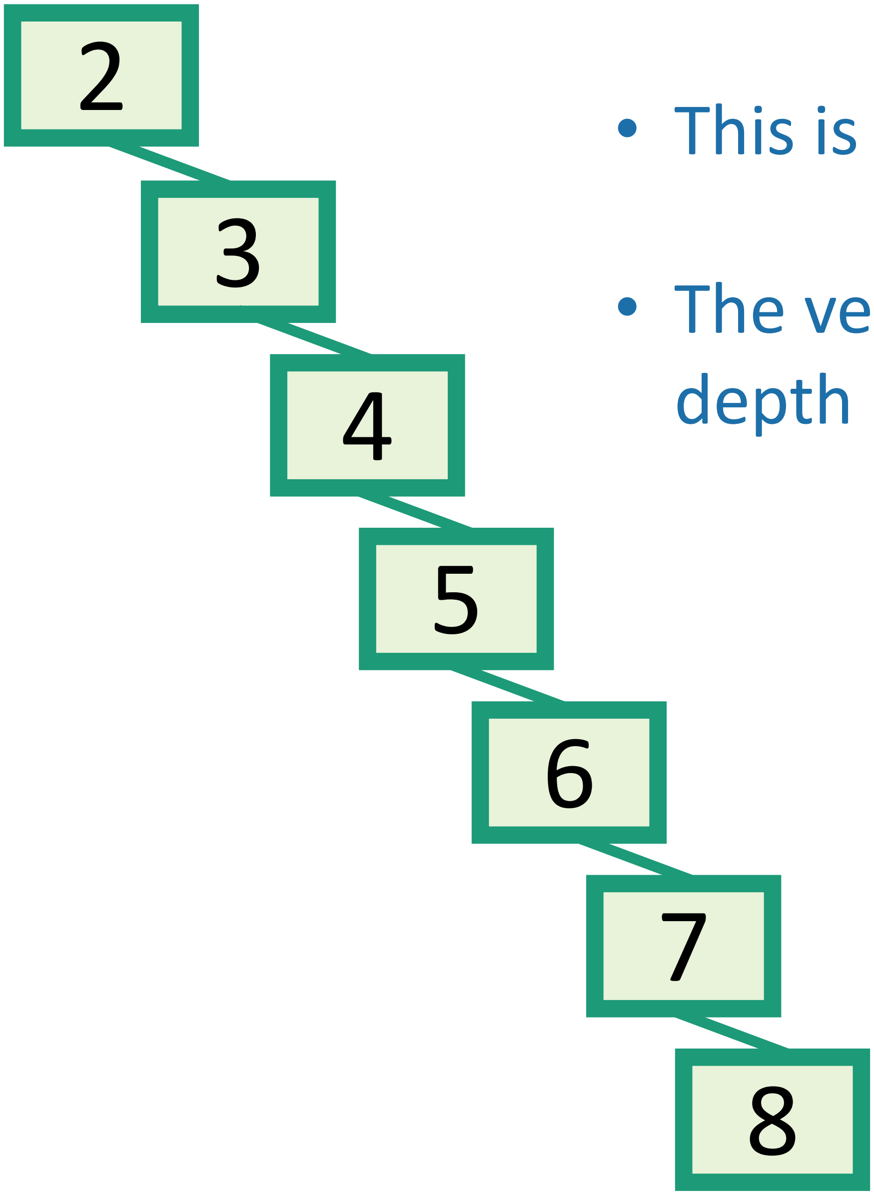


Time = O(height of tree)

Trees have depth O(log(n)). **Done!**

How long does search take?

Wait…



- This is a valid binary search tree.
- The version with n nodes has depth n, **not** O(log(n)).

Could such a tree show up?
In what order would I have to insert the nodes?

Inserting in the order 2,3,4,5,6,7,8 would do it.

So this ***could*** happen.

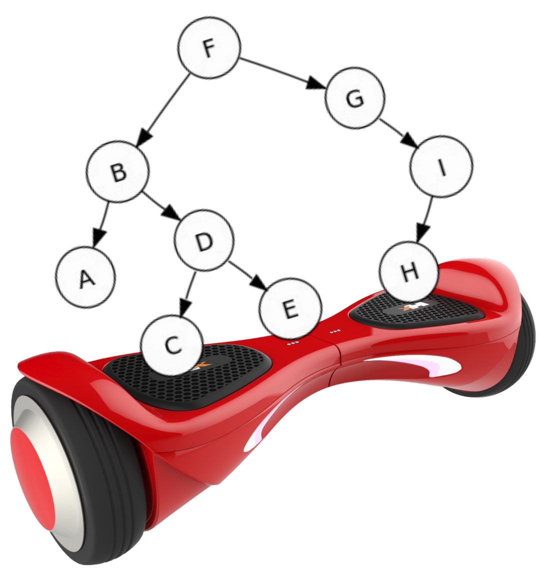
What to do?

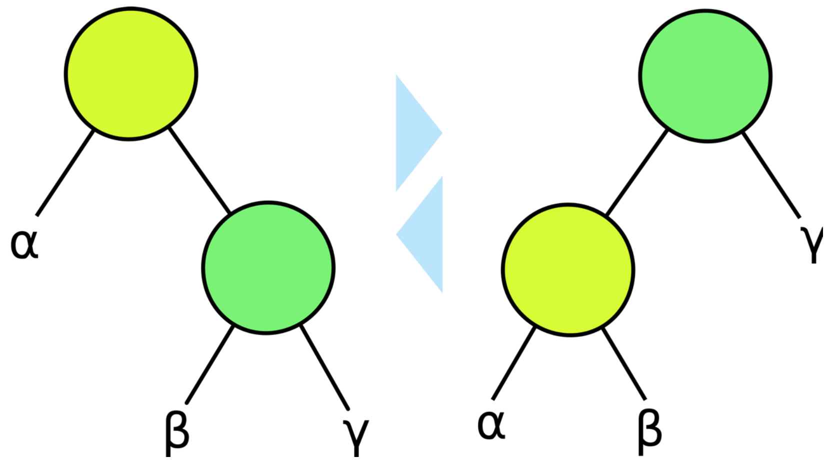
- Goal: Fast SEARCH/INSERT/DELETE
- All these things take time O(height)
- And the height might be big!!! ☹

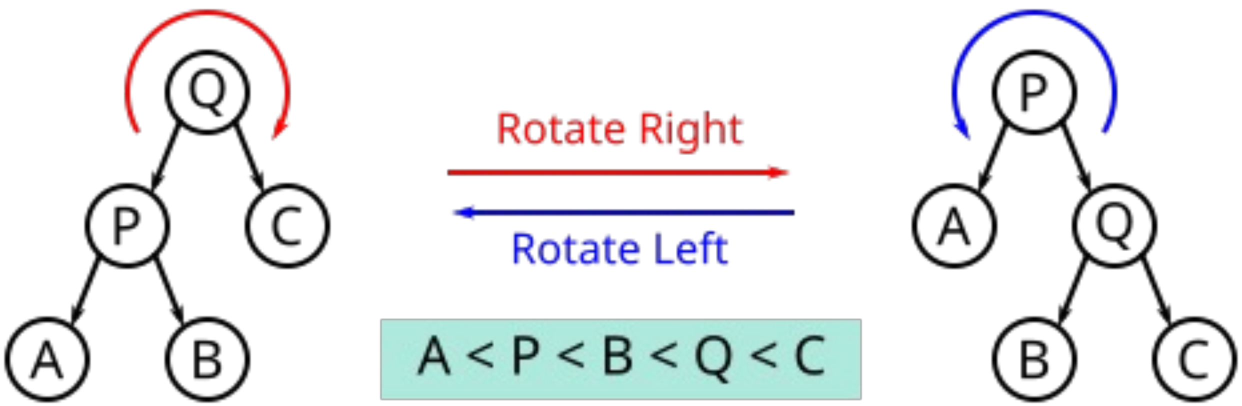
- Idea 0:
  - Keep track of how deep the tree is getting.
  - If it gets too tall, re-do everything from scratch.

- Turns out that's not a great idea. Instead we turn to…

Self-Balancing Binary Search Trees




α
β
γ
α
β
γ

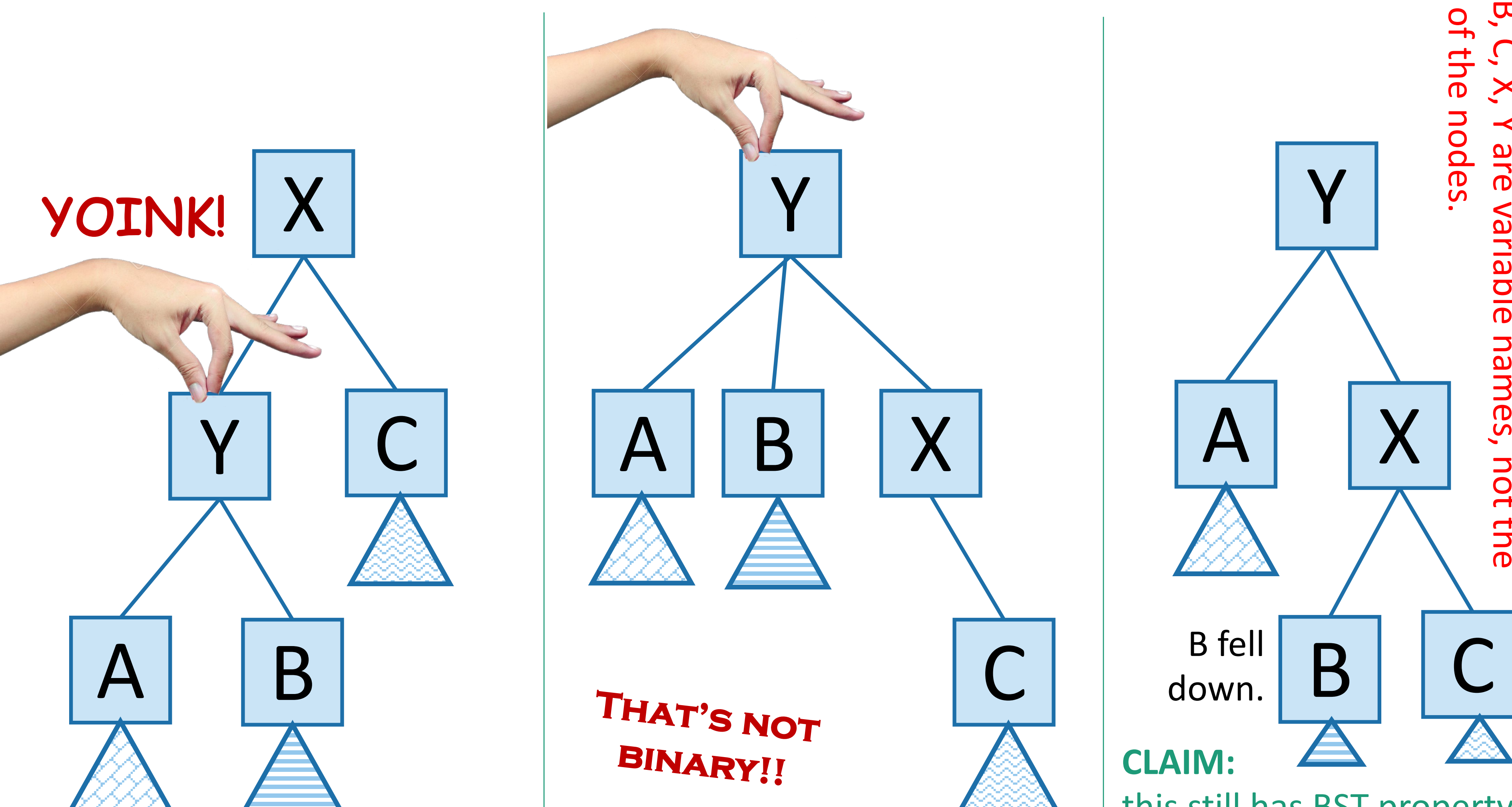

Q
P
C
A
B
Rotate Right
Rotate Left
A < P < B < Q < C
P
A
Q
B
C

Idea 1: Rotations

No matter what lives underneath A,B,C,
this takes time O(1). (Why?)

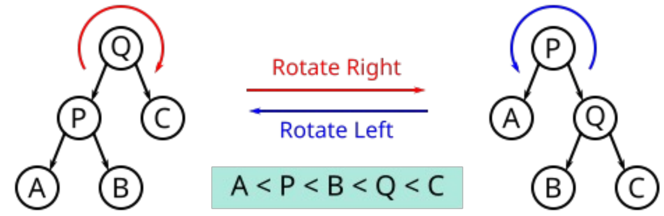
- Maintain Binary Search Tree (BST) property, while moving stuff around.

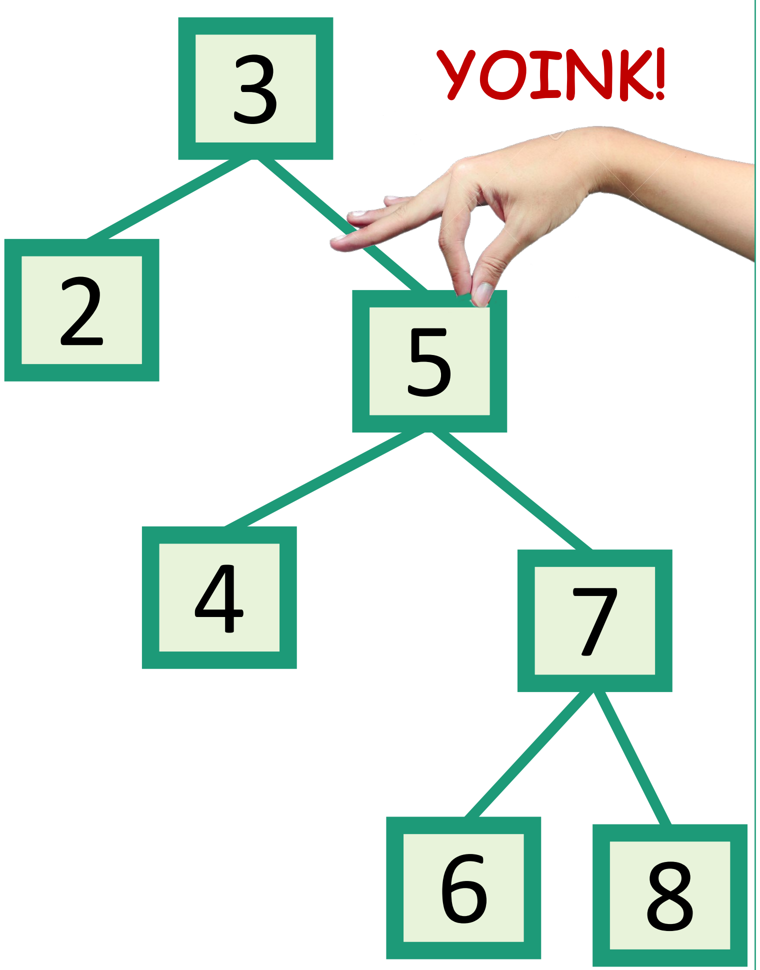
Note: A, B, C, X, Y are variable names, not the contents of the nodes.

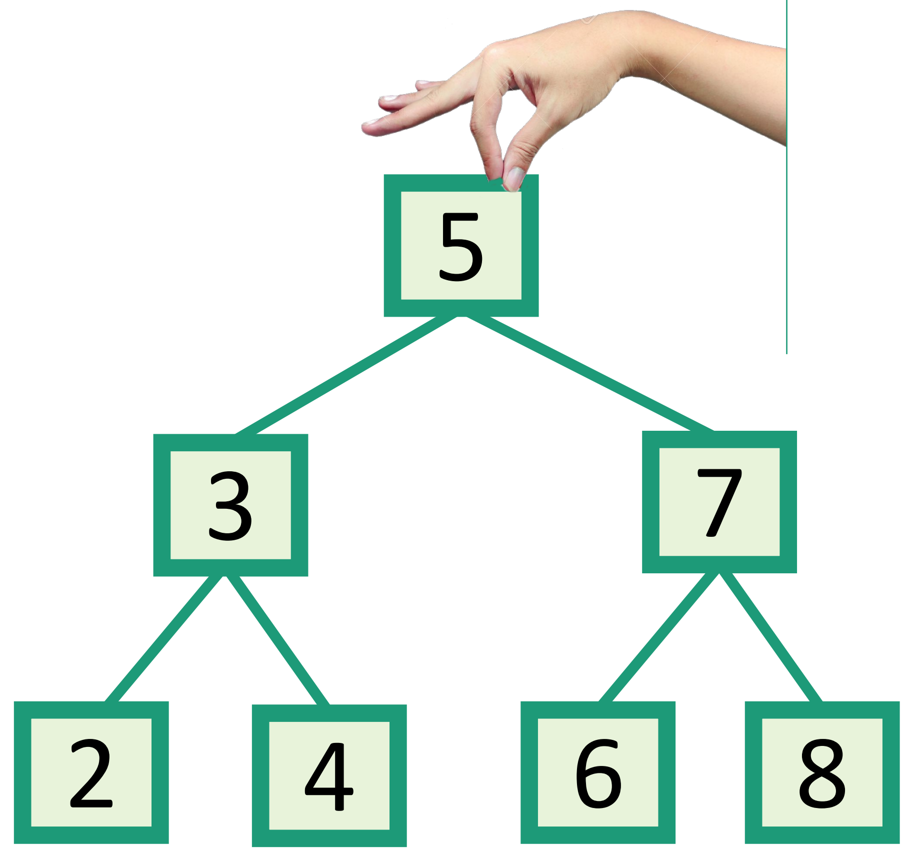


CLAIM:
this still has BST property.

This seems helpful







Does this work?

- Whenever something seems unbalanced, do rotations until it’s okay again.

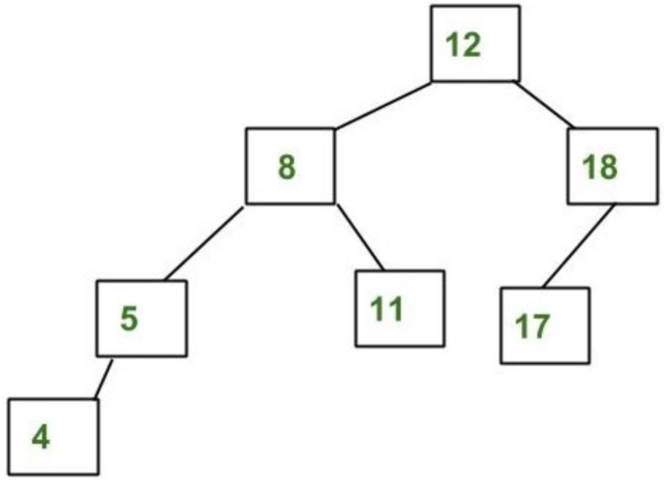
Idea 2: have some proxy for balance

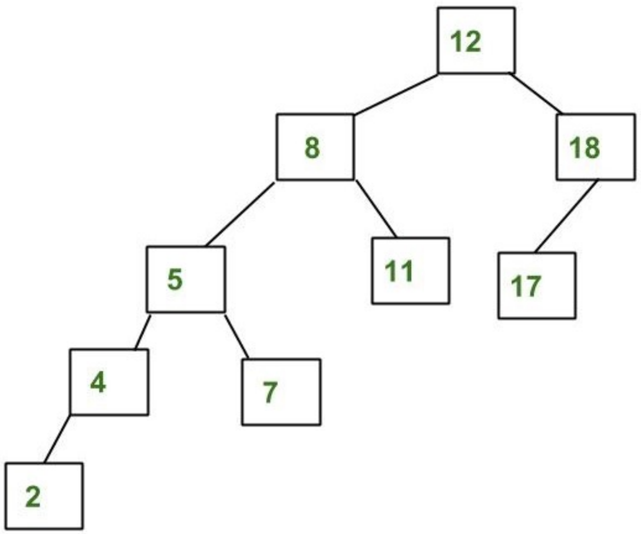
- Maintaining perfect balance is too hard.
- Instead, come up with some proxy for balance:
  - If the tree satisfies [SOME PROPERTY], then it's pretty balanced.
  - We can maintain [SOME PROPERTY] using rotations.

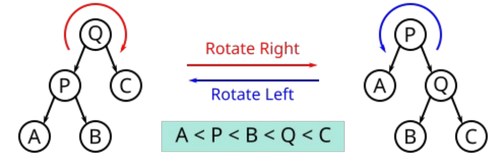
There are actually several ways to do this, but today we'll see…

AVL Trees

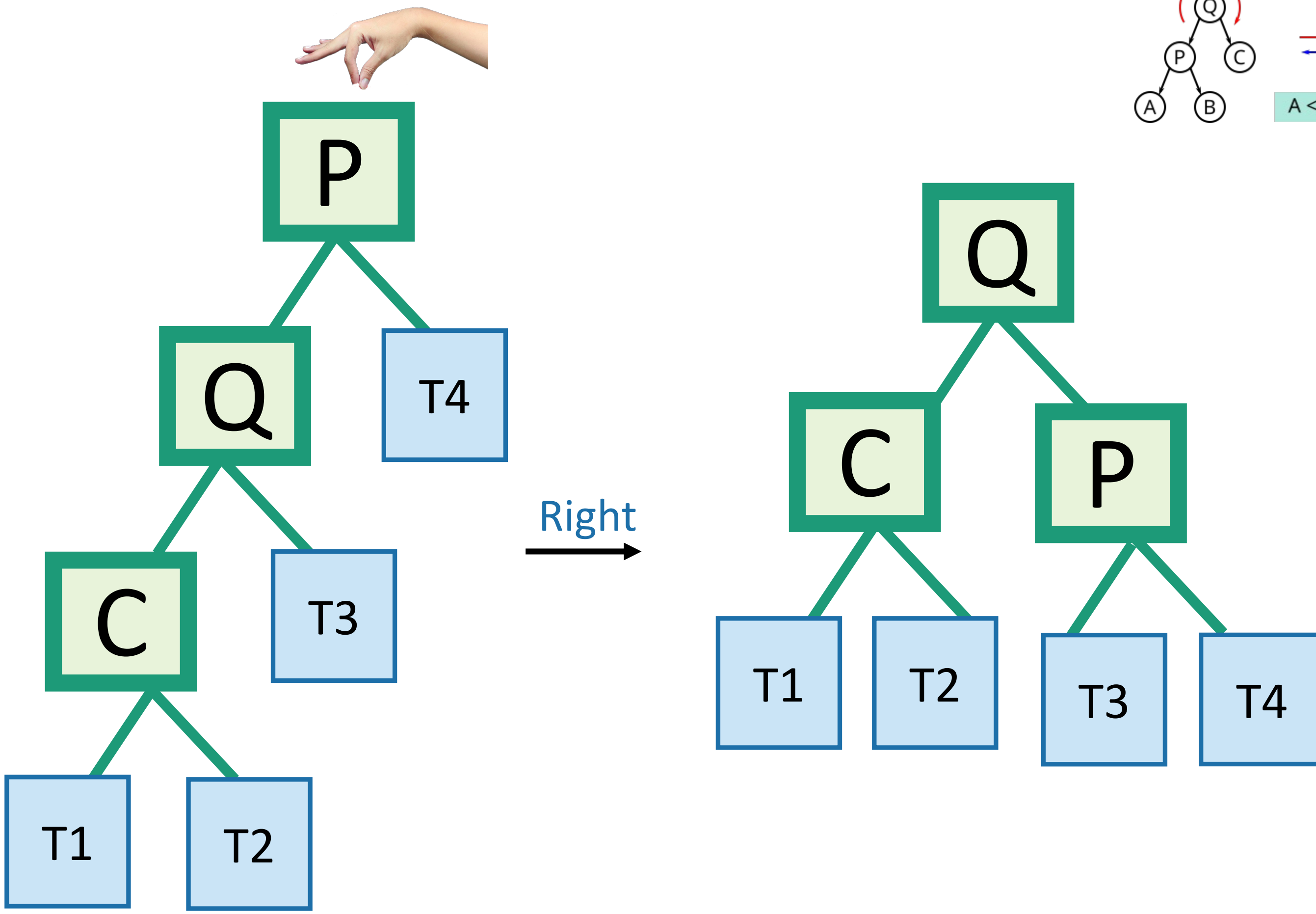
- **Height-Balancing Property**





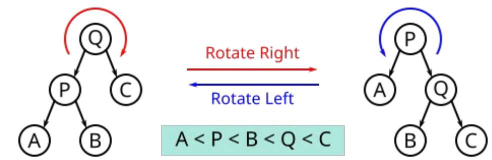


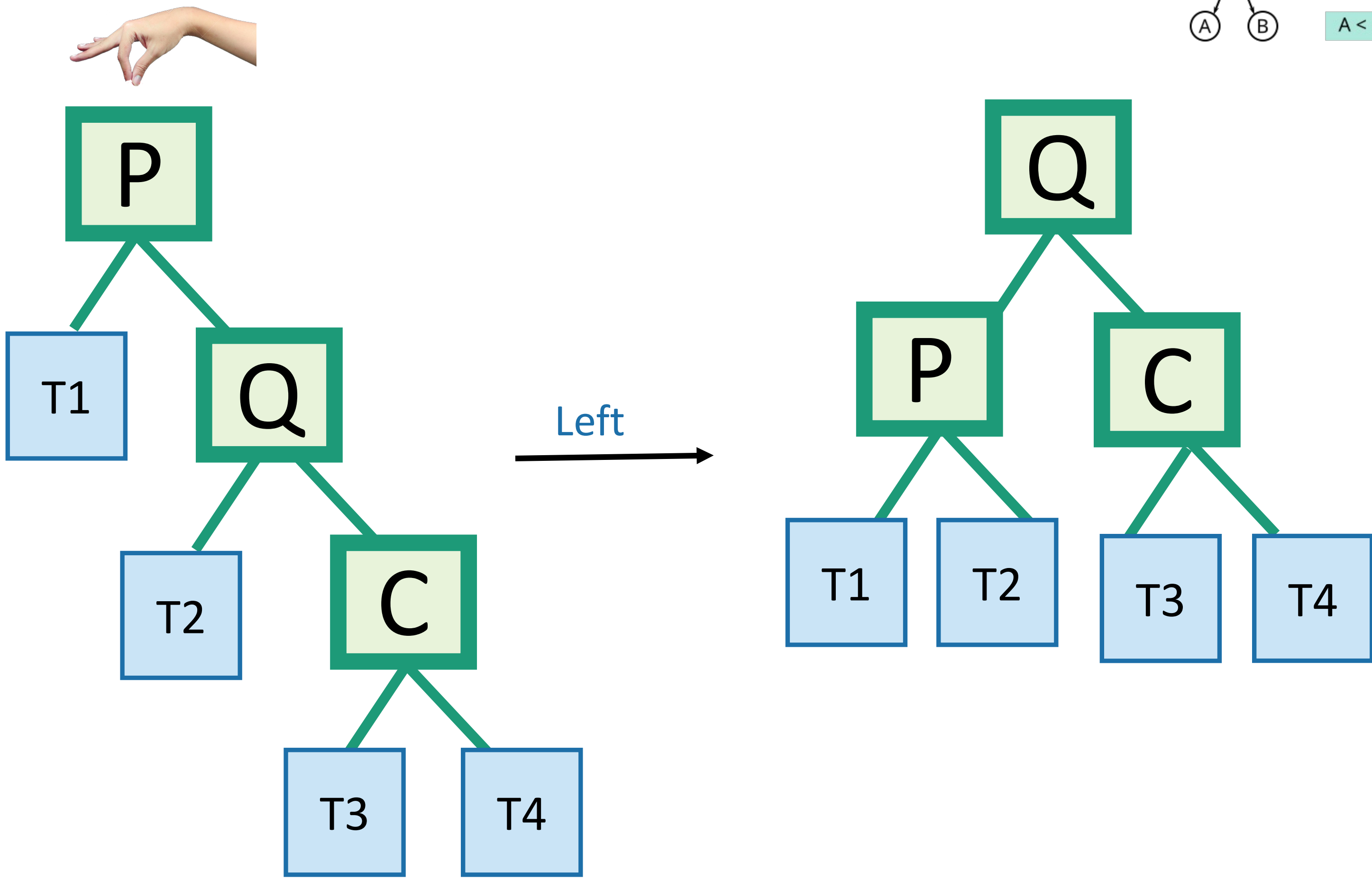
Q
P
C
A
B
Rotate Right
Rotate Left
A < P < B < Q < C
P
A
Q
B
C



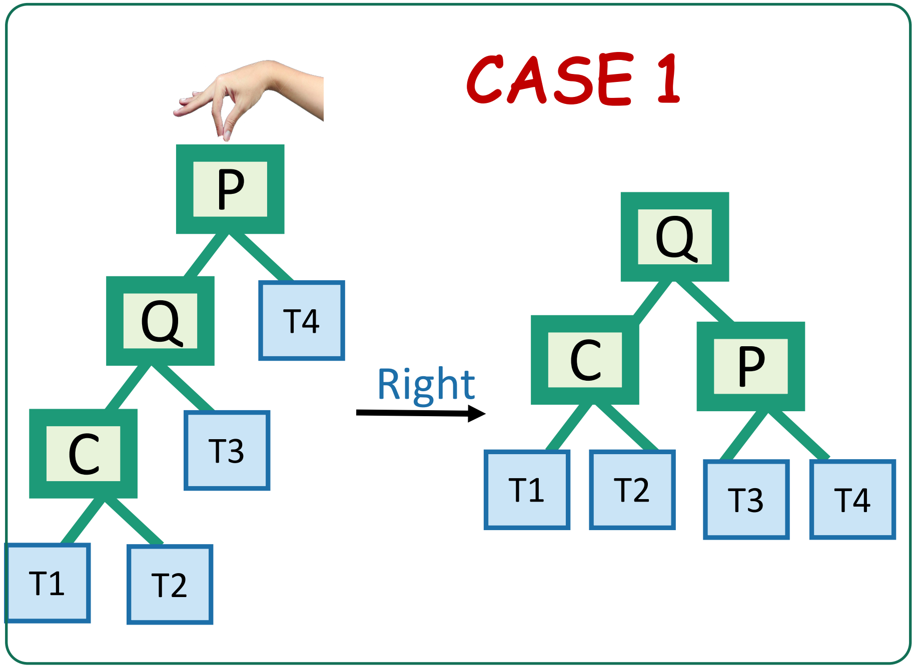
P
Q
T4
C
T3
T1
T2
Right
Q
C
P
T1
T2
T3
T4

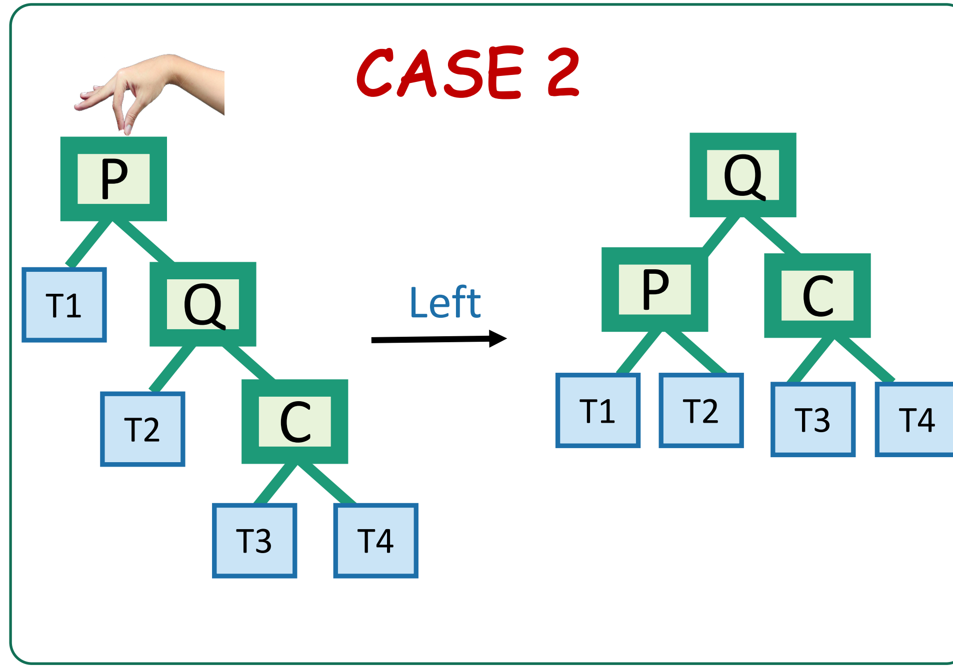
CASE 1

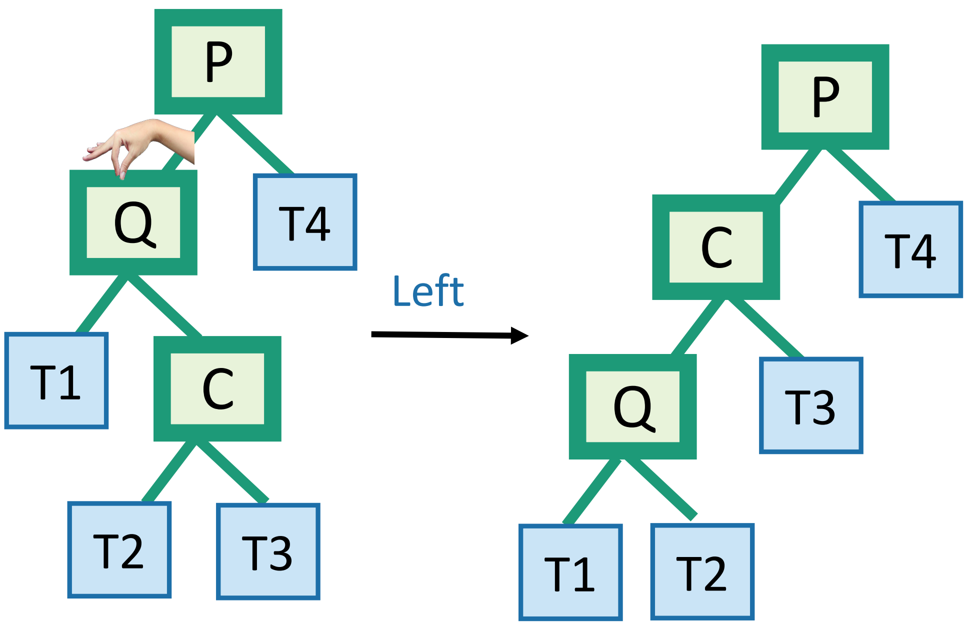

Q
P
C
A
B
Rotate Right
Rotate Left
A < P < B < Q < C
P
A
Q
B
C


P
T1
Q
T2
C
T3
T4
Left
Q
P
C
T1
T2
T3
T4

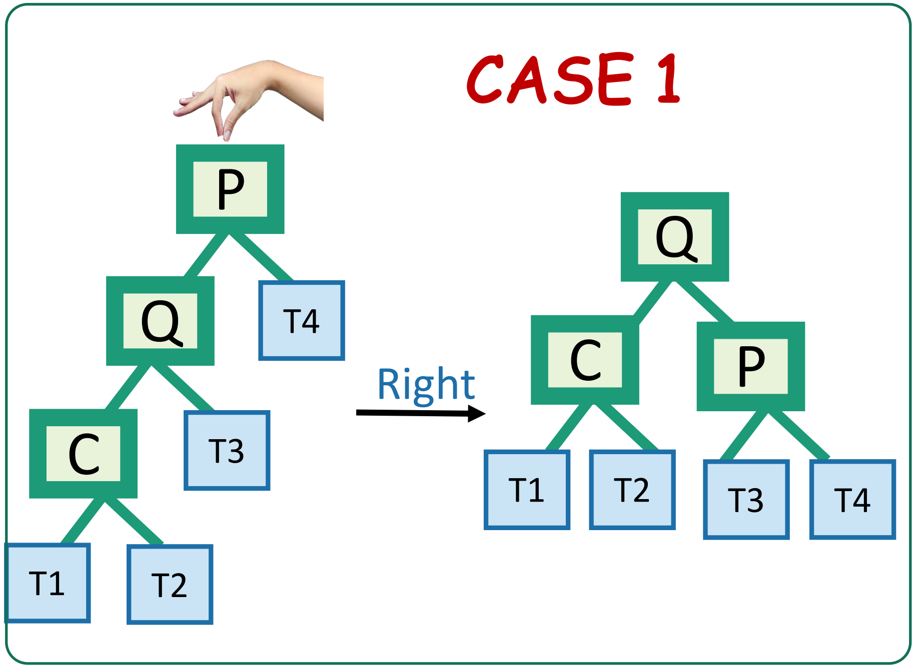
CASE 2

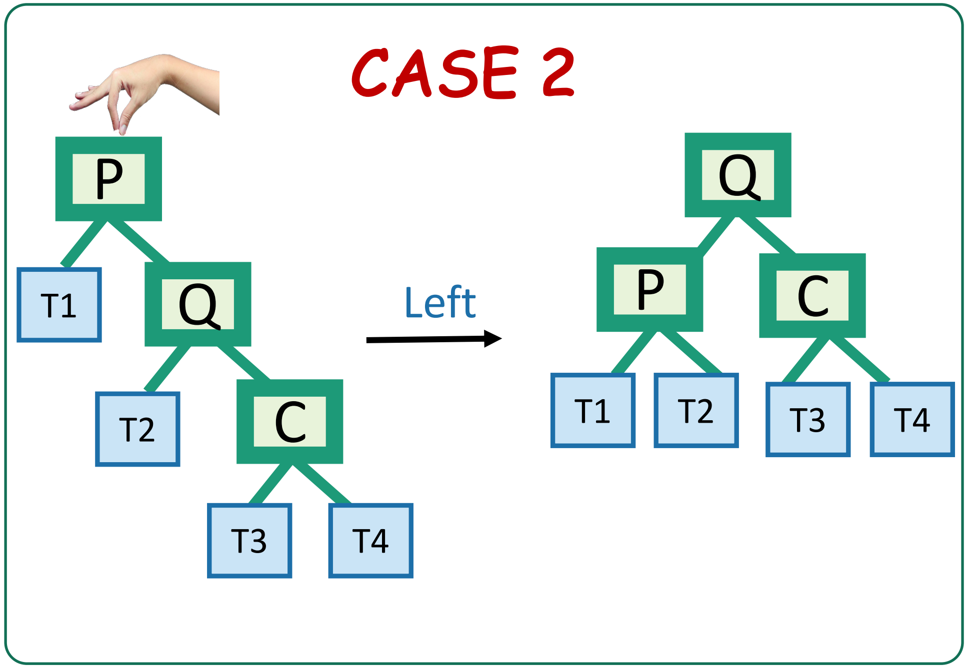

CASE 1
P
Q
T4
C
T3
T1
T2
Right
Q
C
P
T1
T2
T3
T4

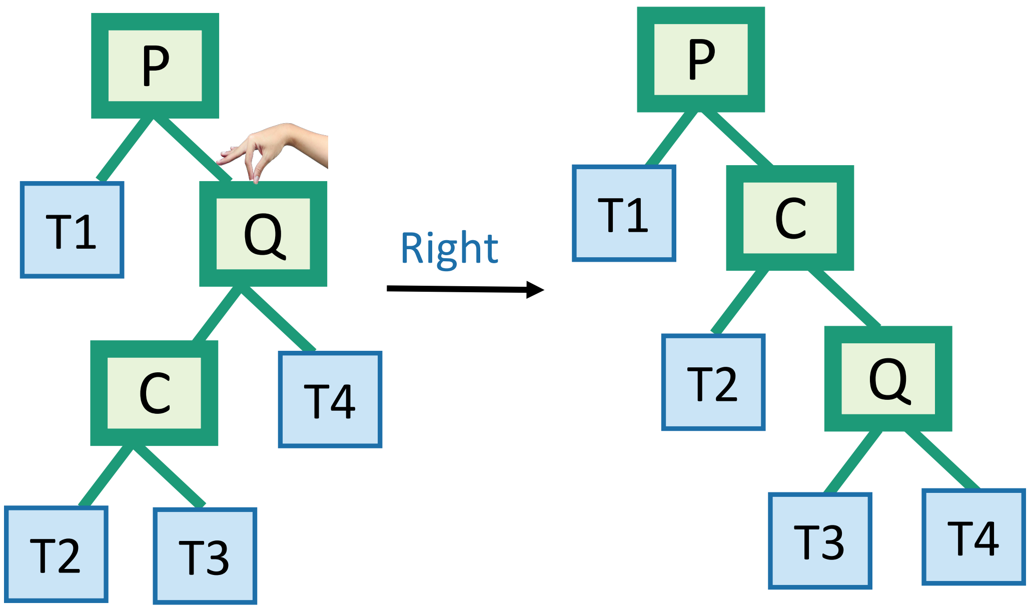

CASE 2
P
T1
Q
T2
C
T3
T4
Left
Q
P
C
T1
T2
T3
T4


P
Q
T4
T1
C
T2
T3
Left
P
C
T4
Q
T3
T1
T2

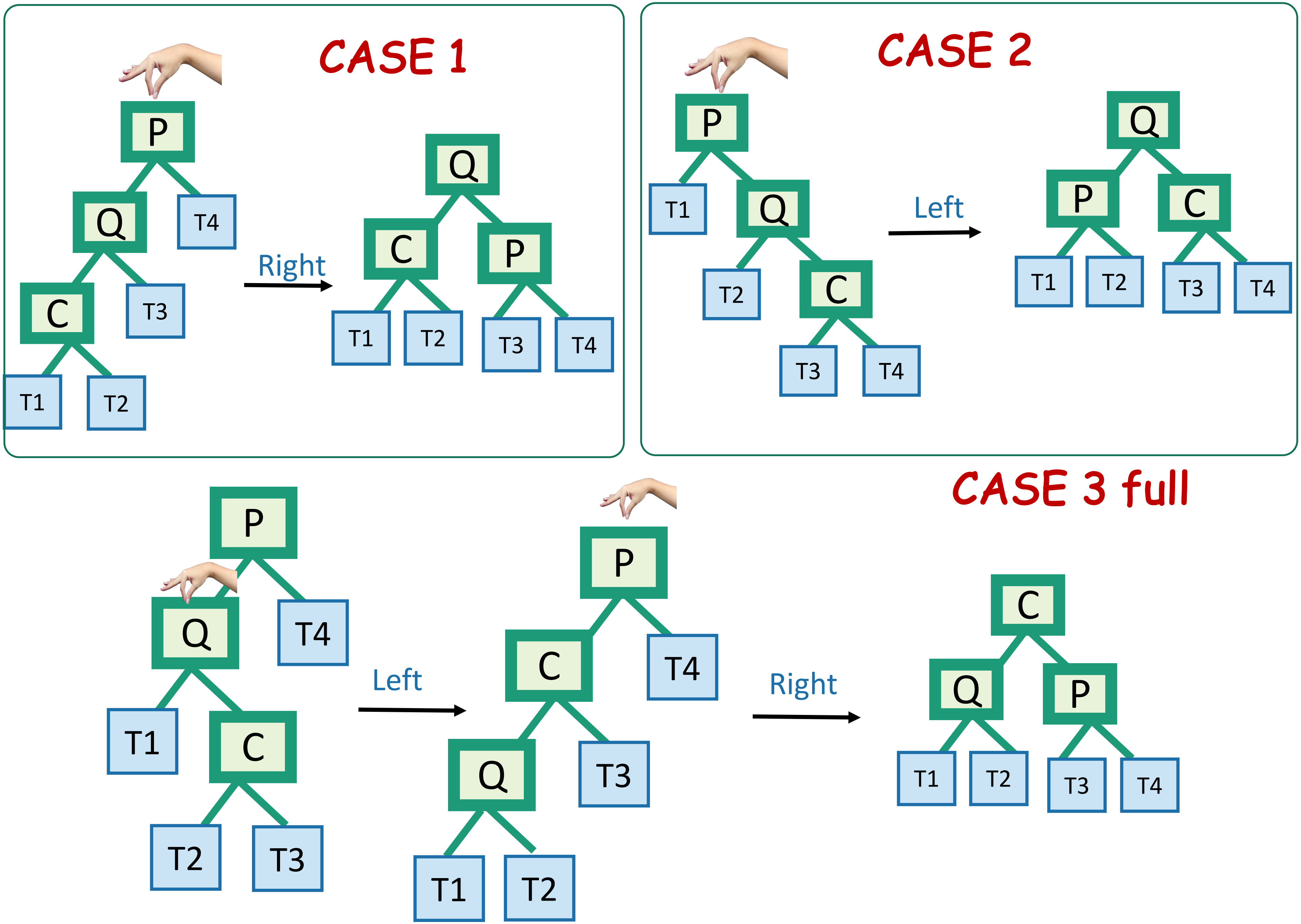
CASE 3


CASE 1
P
Q
T4
C
T3
T1
T2
Right
Q
C
P
T1
T2
T3
T4

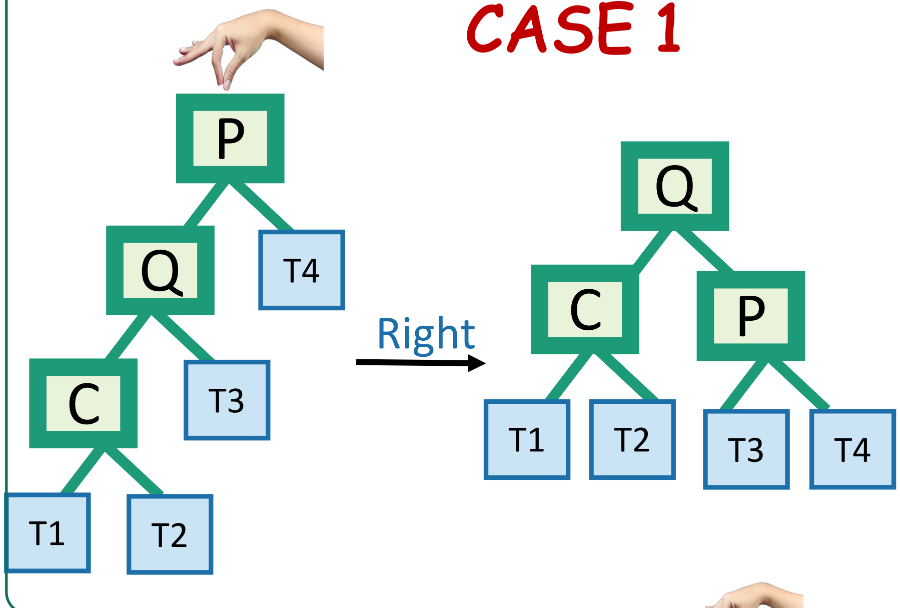

CASE 2
P
T1
Q
T2
C
T3
T4
Left
Q
P
C
T1
T2
T3
T4


P
T1
Q
C
T4
T2
T3
Right
P
T1
C
T2
Q
T3
T4

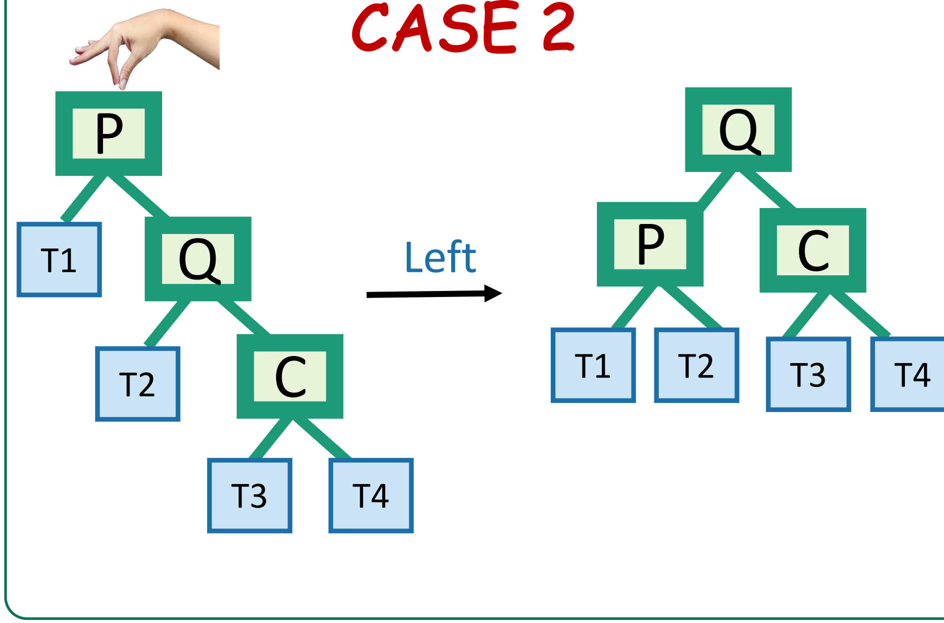
CASE 4


CASE 1
P
Q
T4
C
T3
T1
T2
Right
Q
C
P
T1
T2
T3
T4
CASE 2
P
T1
Q
T2
C
T3
T4
Left
Q
P
C
T1
T2
T3
T4
CASE 3 full
P
Q
T4
T1
C
T2
T3
Left
P
C
T4
Q
T3
T1
T2
Right
C
Q
P
T1
T2
T3
T4

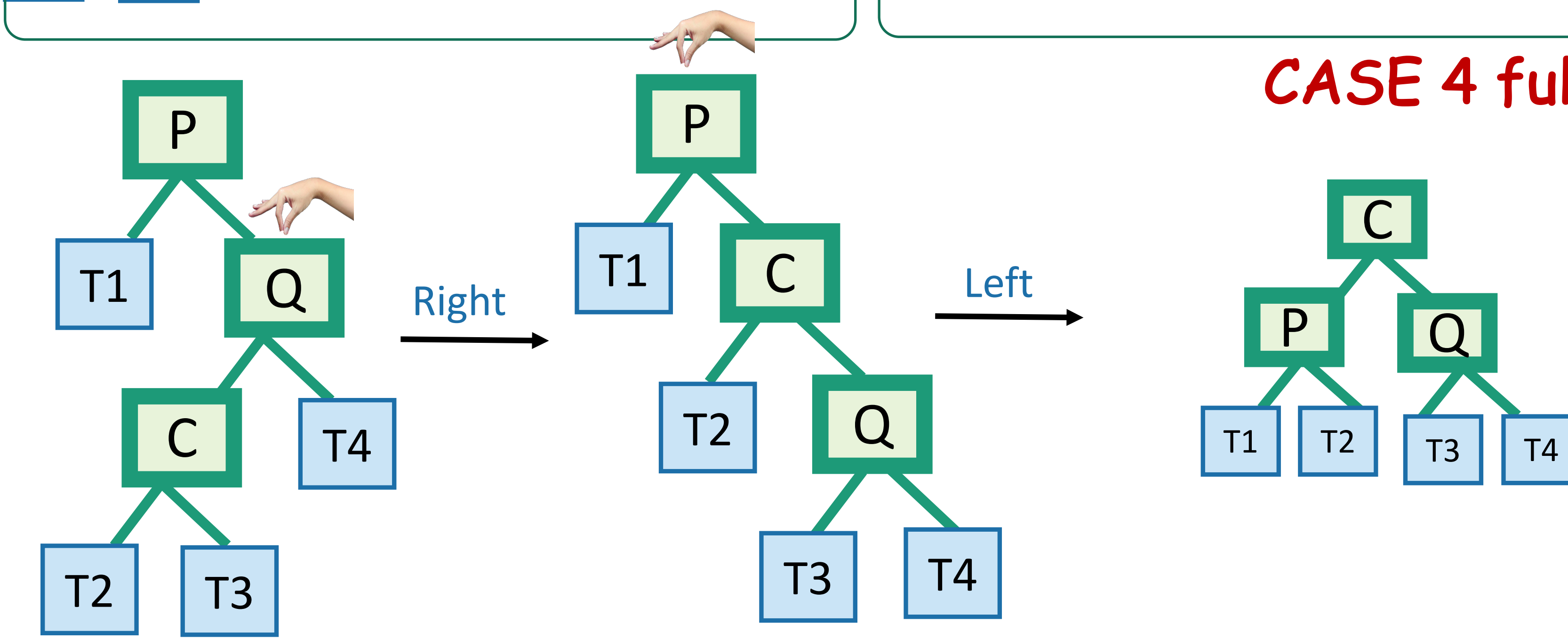
CASE 1

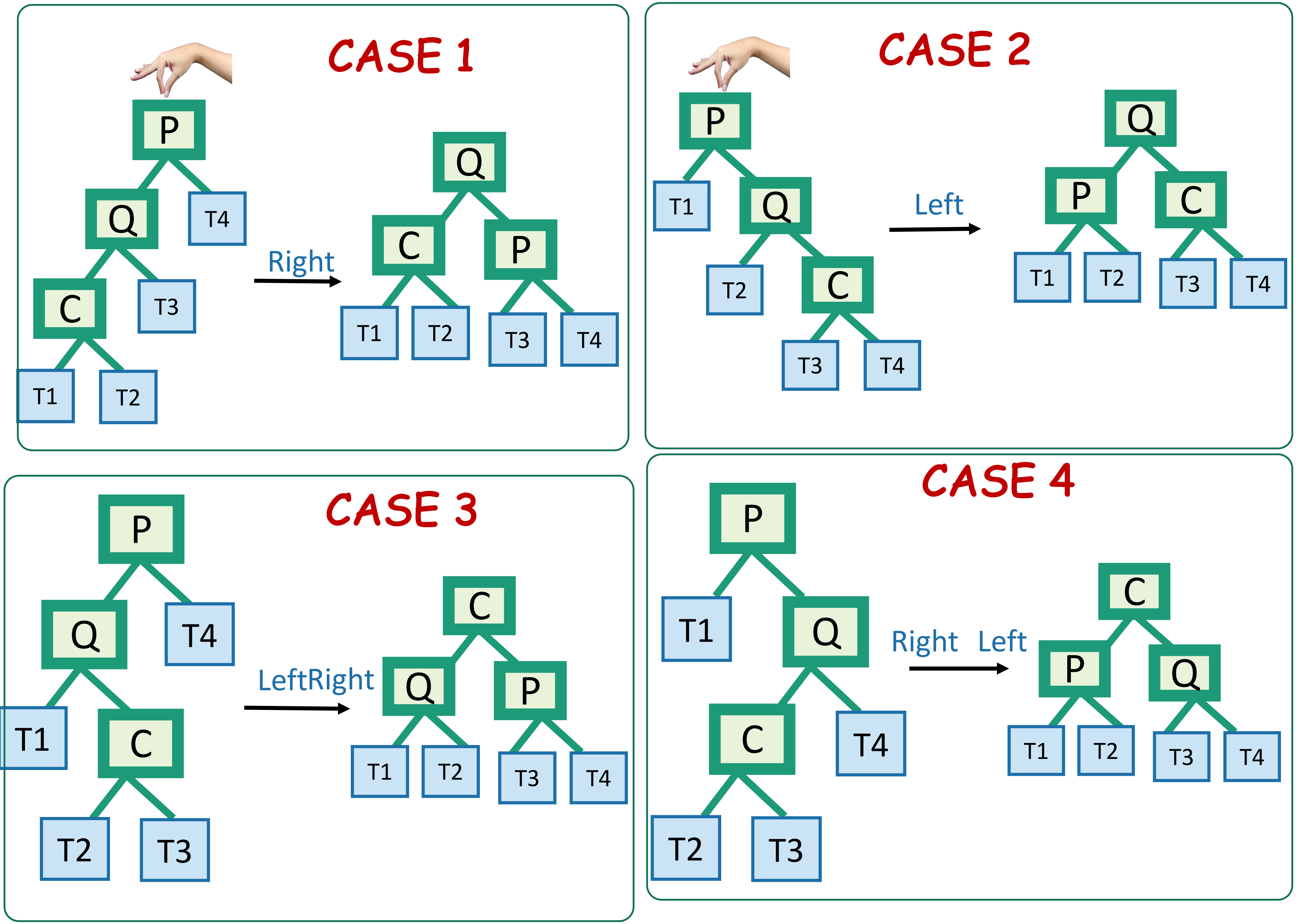


CASE 2

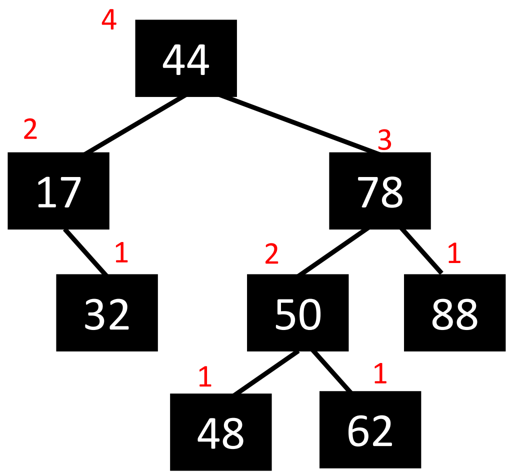


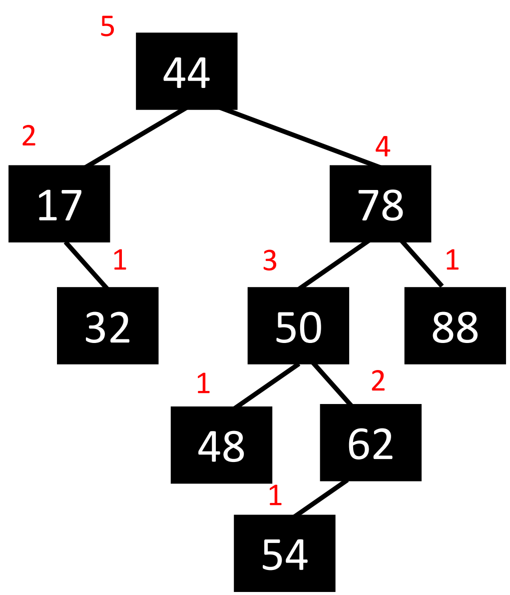
CASE 4 full

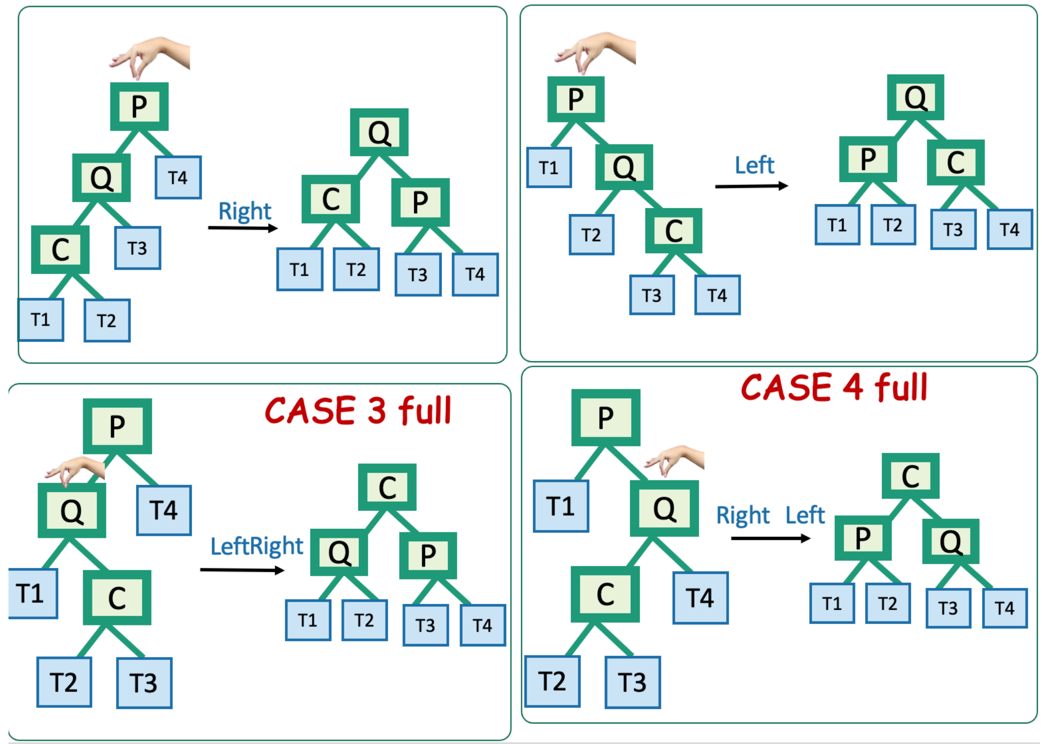



CASE 1
P
Q
T4
C
T3
T1
T2
Right
Q
C
P
T1
T2
T3
T4
CASE 2
P
T1
Q
T2
C
T3
T4
Left
Q
P
C
T1
T2
T3
T4
CASE 3
P
Q
T4
T1
C
T2
T3
LeftRight
C
Q
P
T1
T2
T3
T4
CASE 4
P
T1
Q
C
T4
T2
T3
Right Left
C
P
Q
T1
T2
T3
T4

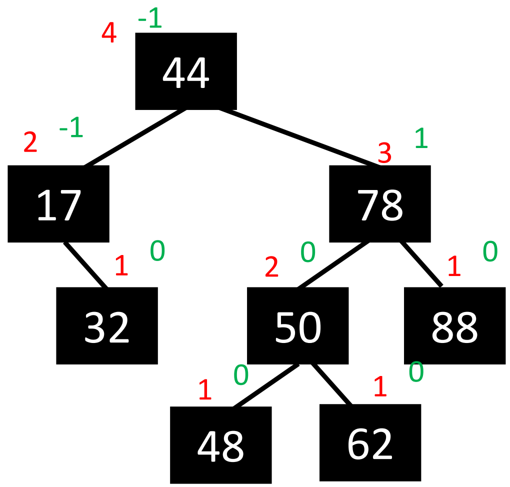
Inserting into an AVL Tree

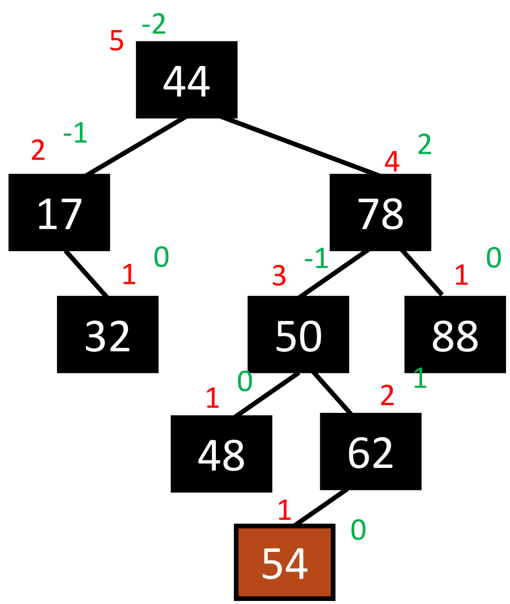




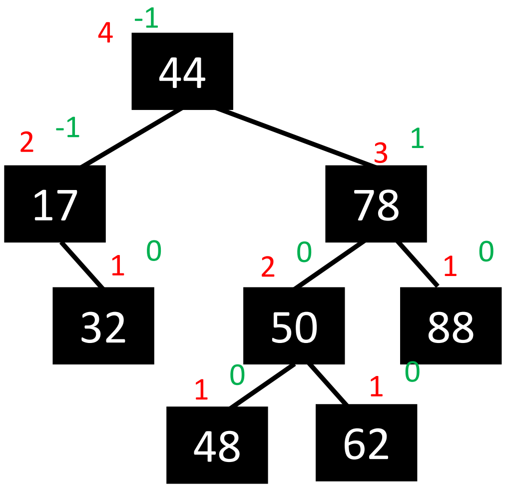


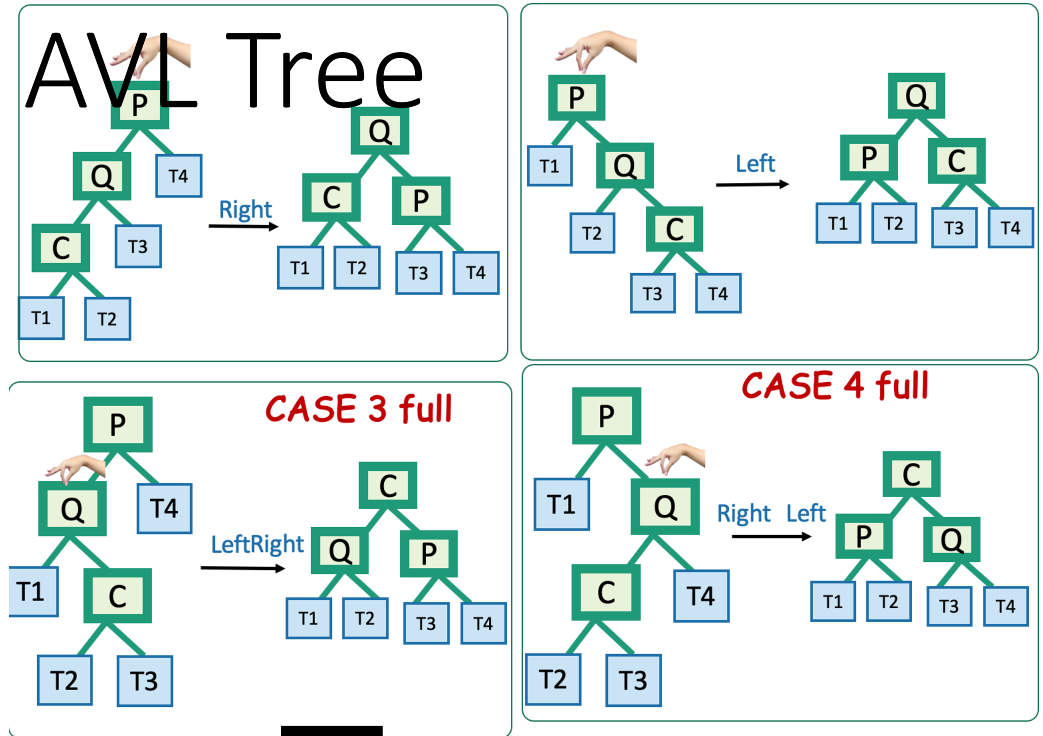
Inserting into an AVL Tree

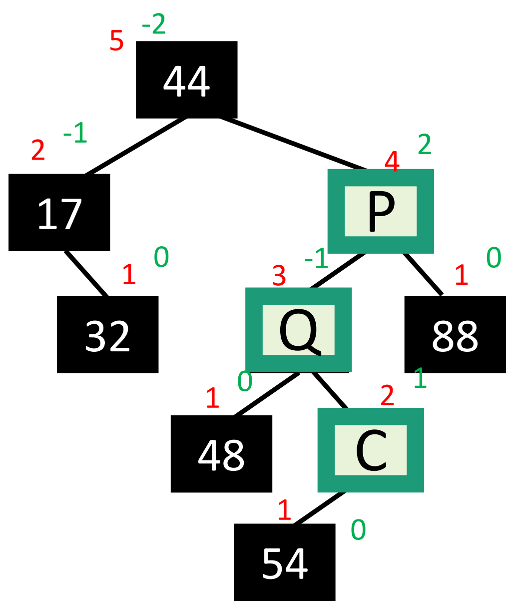


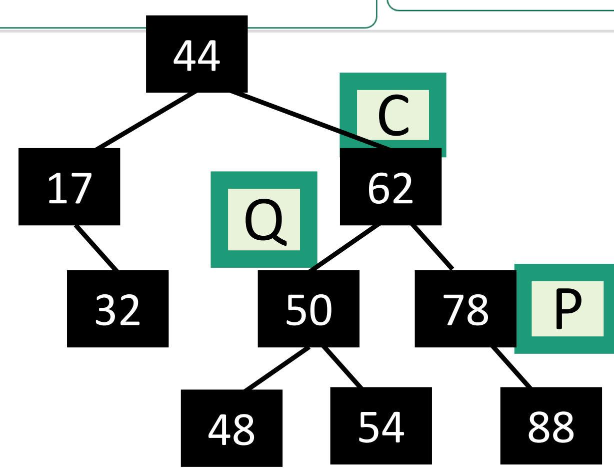


Inserting into an AVL Tree



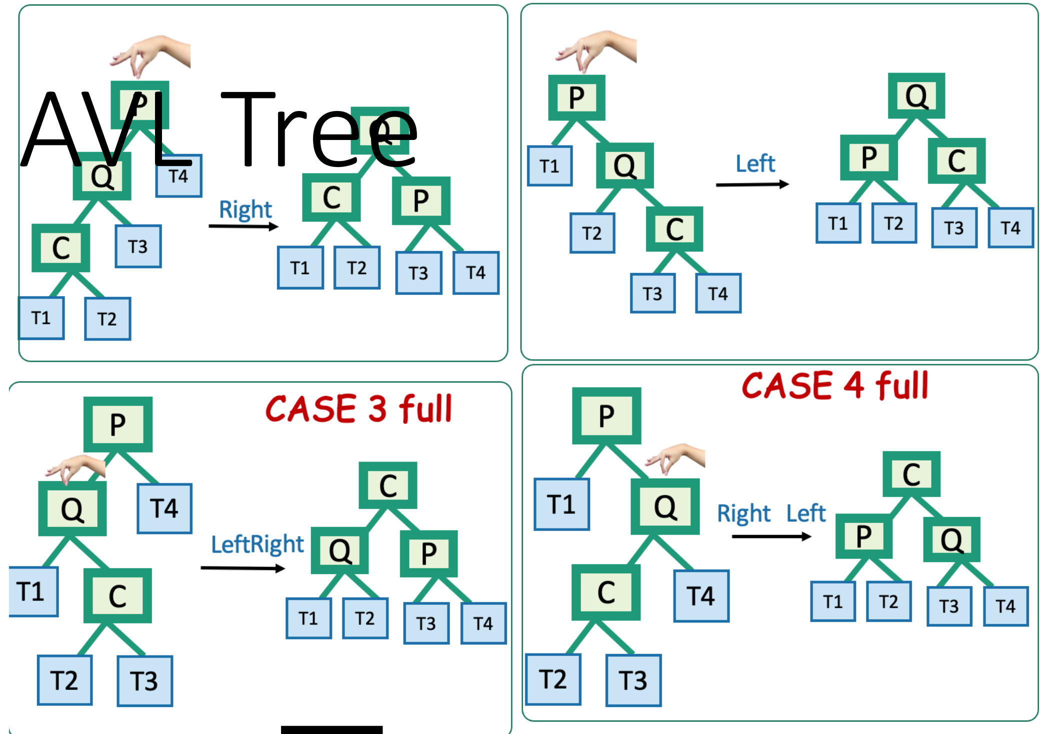


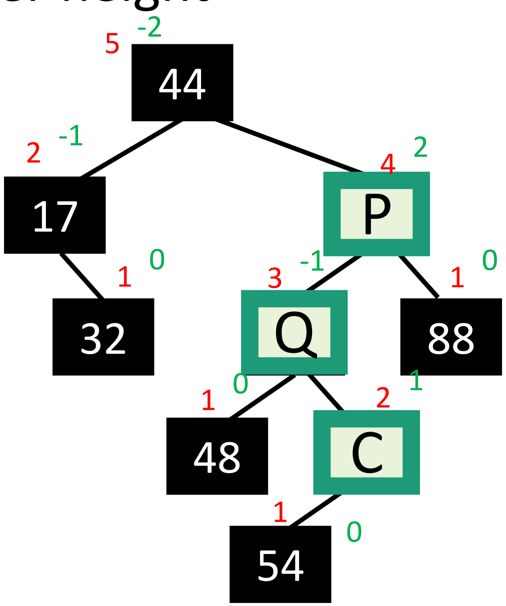


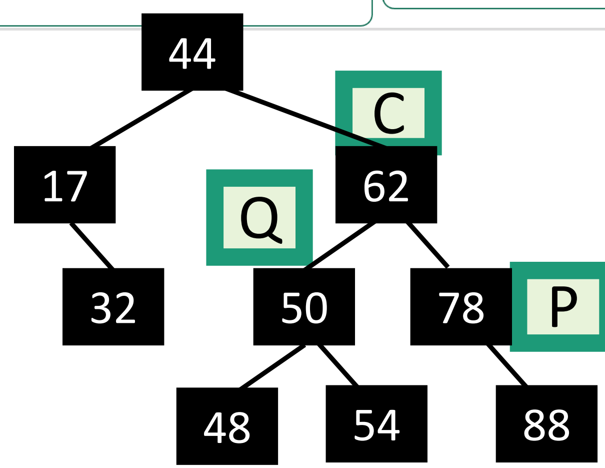


Inserting into an AVL Tree

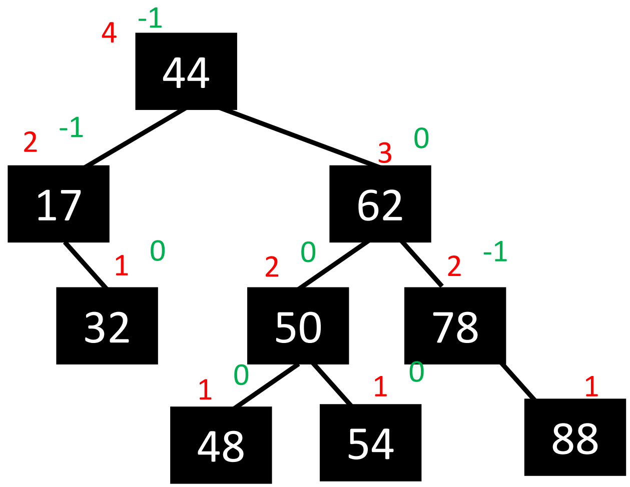
- P is the first unbalanced node you encountered going up from p, where p is the inserted node
- Q is the child of P with larger height (must be ancestor of p)
- C is the child of Q with larger height

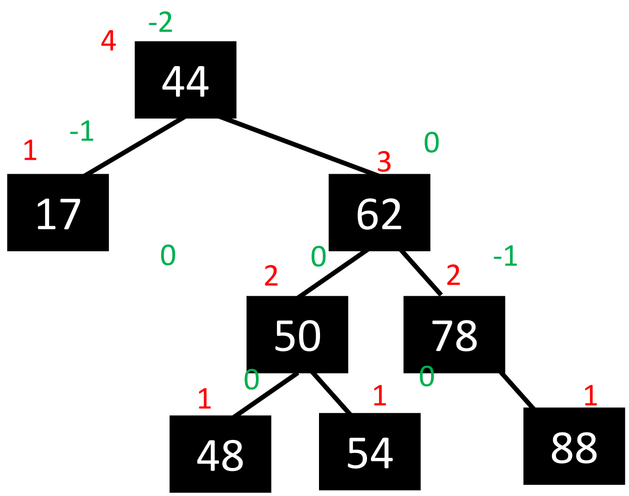


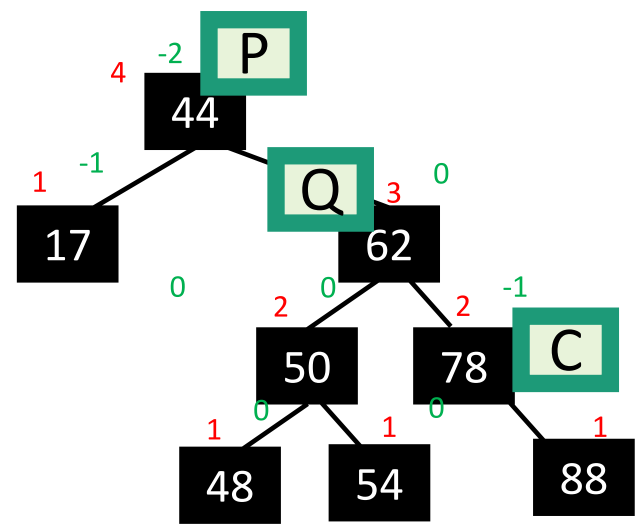




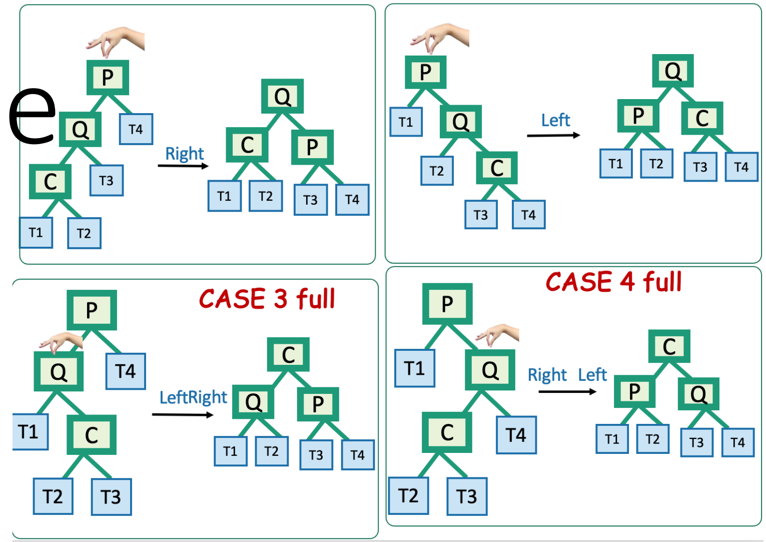
Deletion in an AVL Tree



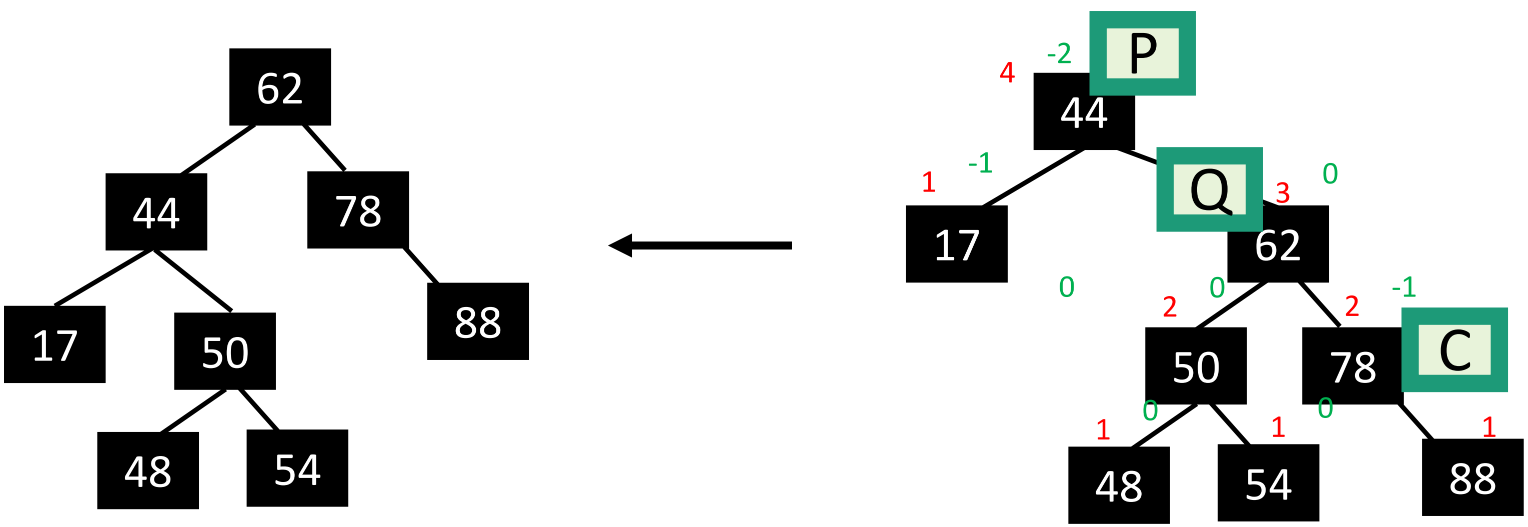




Deletion in an AVL Tree



- P is the unbalanced first node you encountered going up from p, where p is the parent of removed node
- Q is the child of P with larger height
- C is the child of Q with larger height (if equal: try growing on the same side where P-Q are growing)

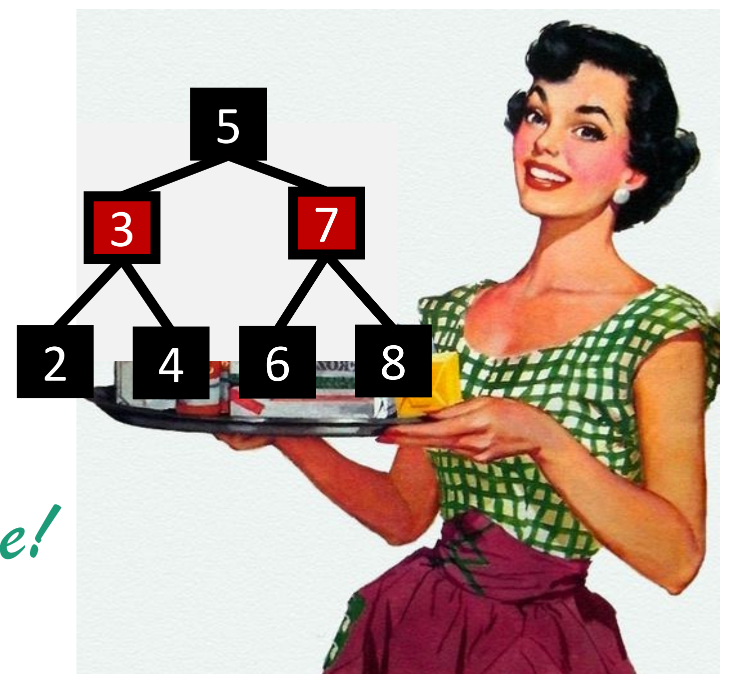


Red-Black Trees

- A Binary Search Tree that balances itself!
- No more time-consuming by-hand balancing!
- Be the envy of your friends and neighbors with the time-saving...

Red-Black tree!

Maintain balance by stipulating that **black nodes** are balanced, and that there *aren't too many* **red nodes.**

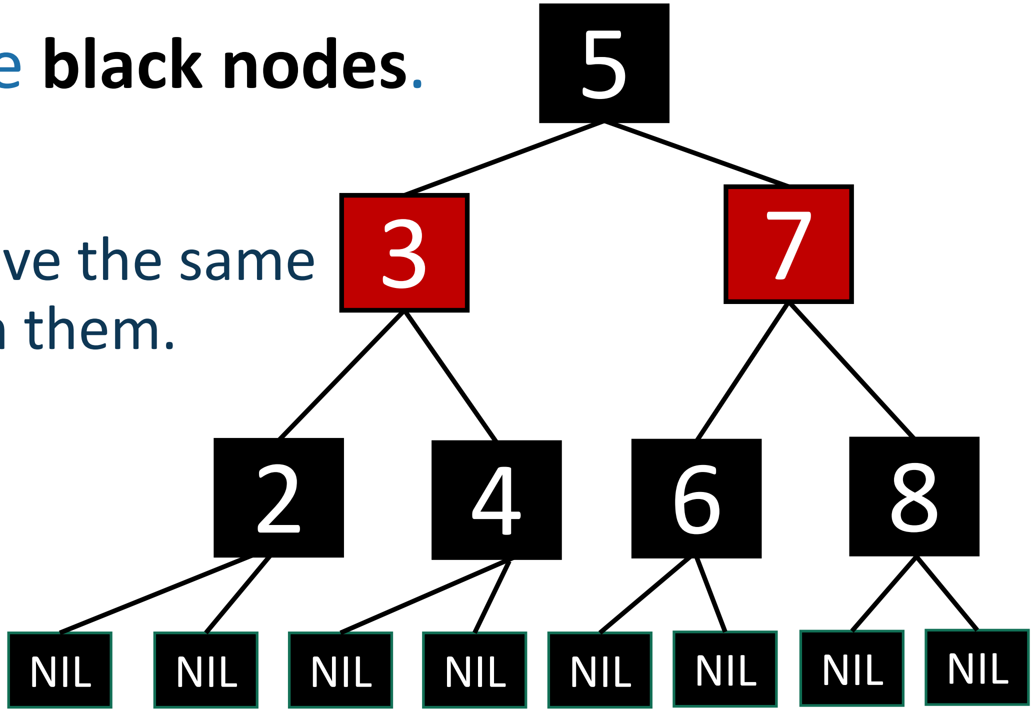


It's just good sense!

Red-Black Trees

these rules are the proxy for balance

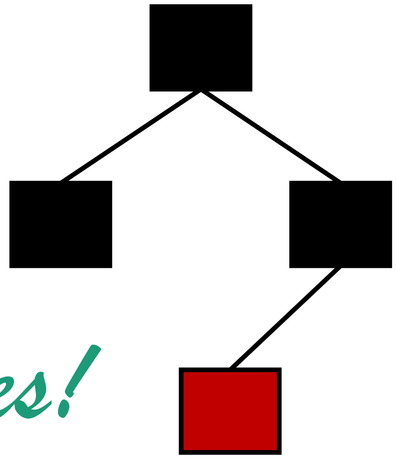
- Every node is colored red or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



I'm not going to draw the NIL children in the future, but they are treated as black nodes.

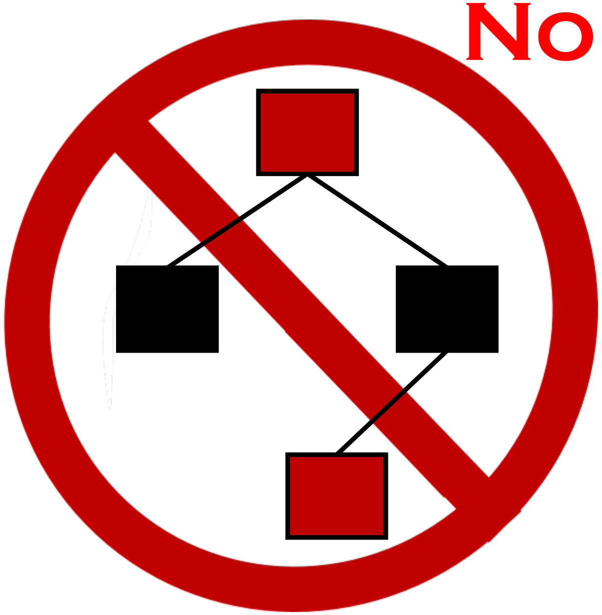
Examples(?)

- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.

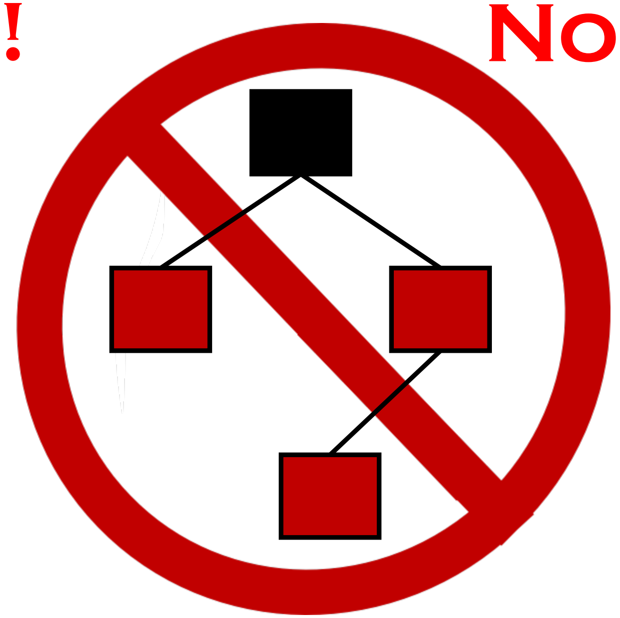


Yes!

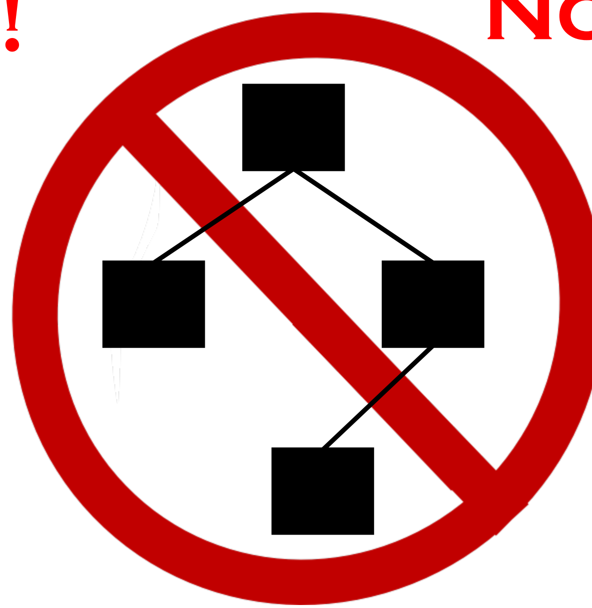
No!



No!

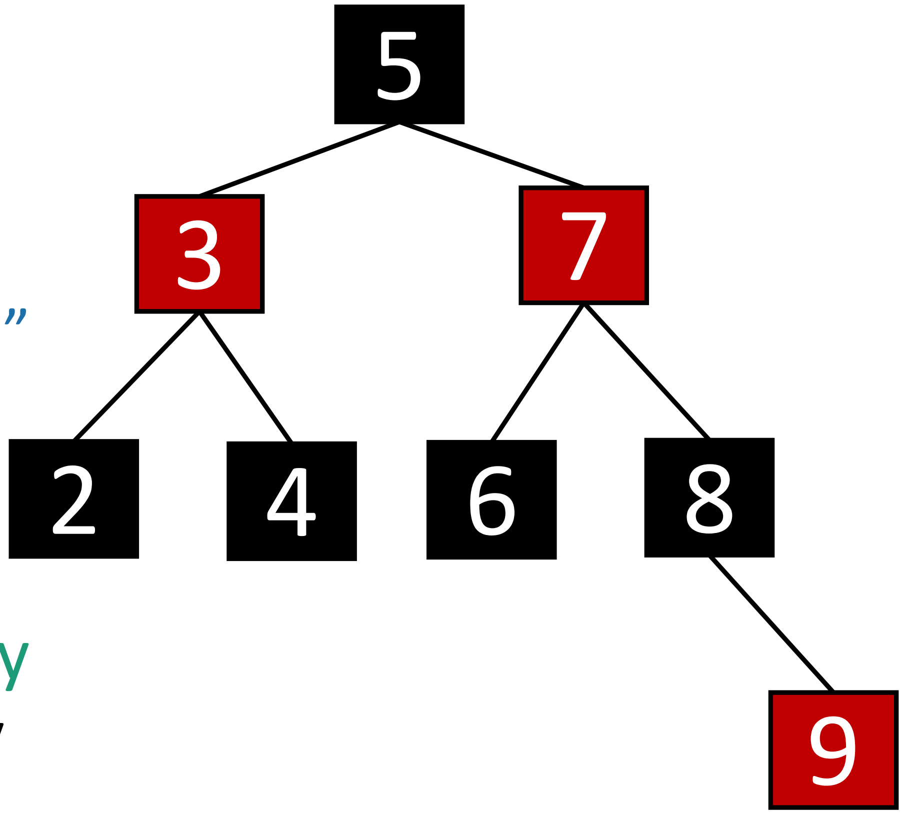


No!



Why???????

- This is pretty balanced.
  - The **black nodes** are balanced
  - The **red nodes** are "spread out" so they don't mess things up too much.



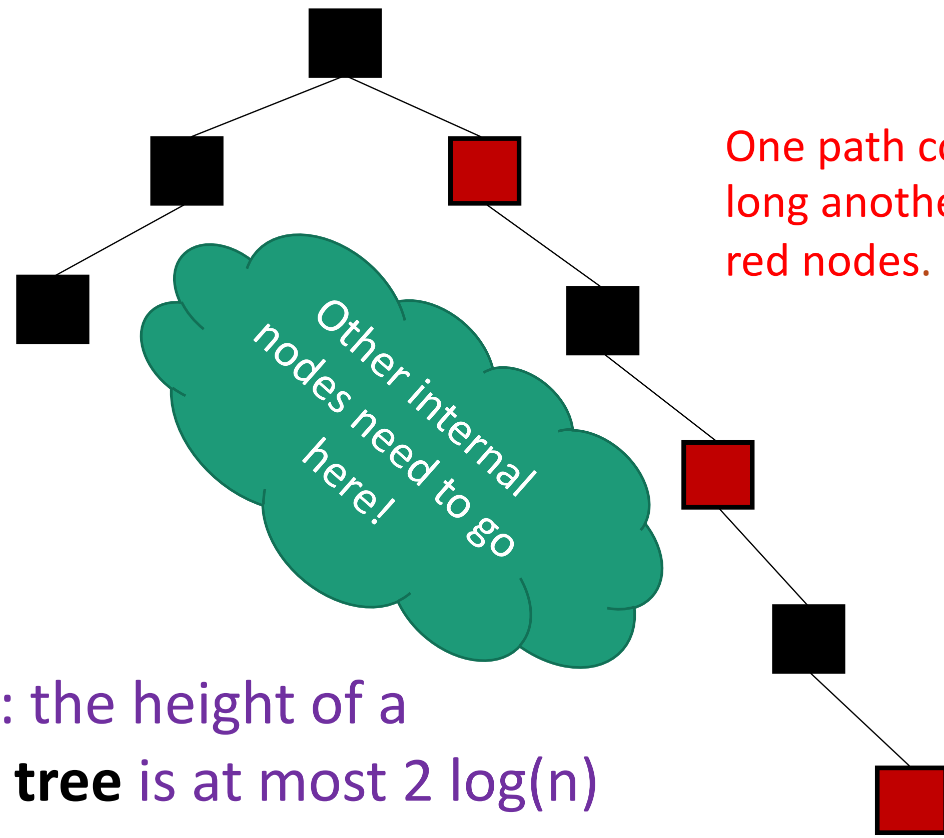
- We can maintain this property as we insert/delete nodes, by using rotations.

This is the really clever idea!

This **Red-Black** structure is a proxy for balance.

This is "pretty balanced"

- To see why, intuitively, let's try to build a Red-Black Tree that's unbalanced.



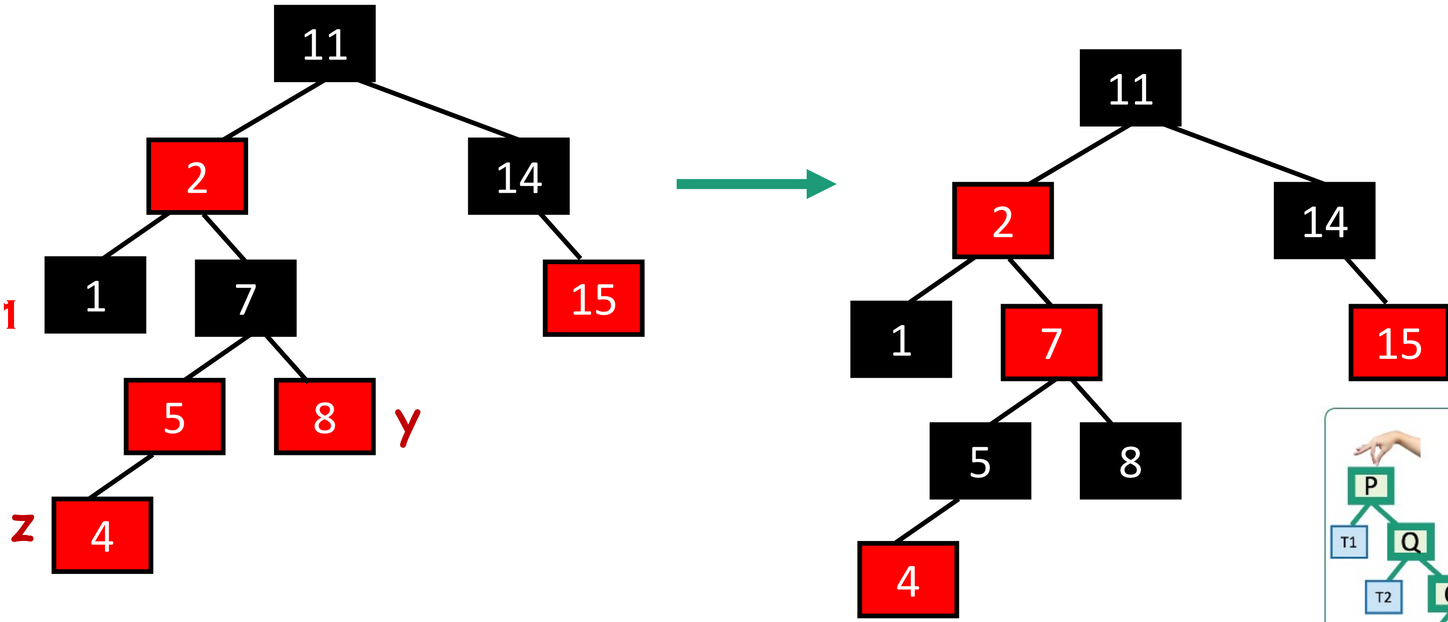
One path could be twice as long another if we pad it with red nodes.

Conjecture: the height of a **red-black tree** is at most 2 log(n)

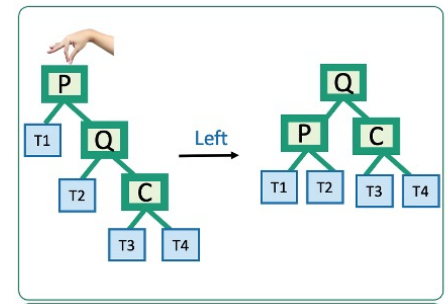
Okay, so it's balanced…
…but can we maintain it?

- Yes!
- See CLRS for more details.
- (You are not responsible for the details for this class – but you should understand the main ideas).

Case 1

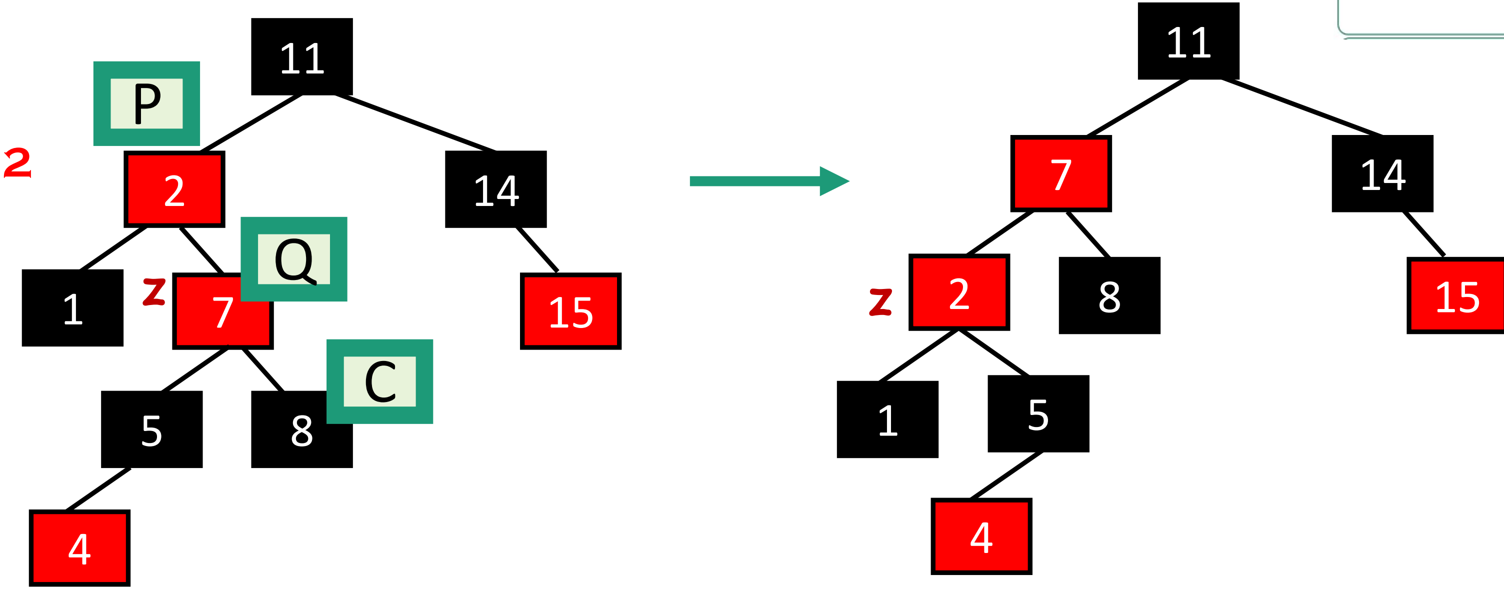


11
2
14
1
7
15
5
8
y
z
4
11
2
14
1
7
15
5
8
4



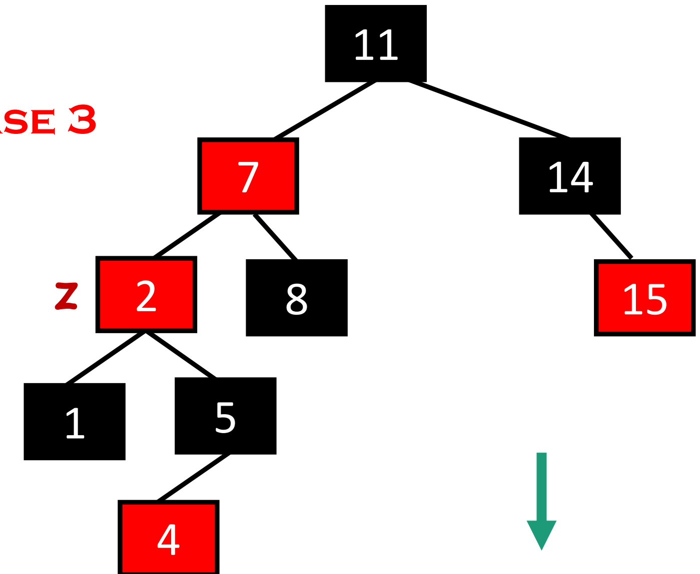
P
T1
Q
T2
C
T3
T4
Left
Q
P
C
T1
T2
T3
T4

case 2

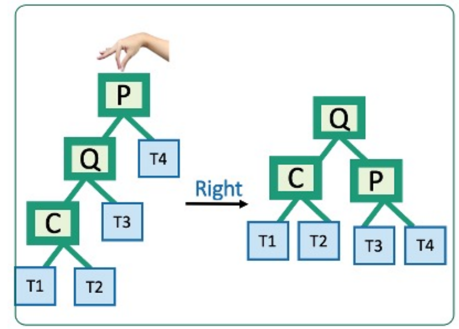


P
11
2
14
Q
1
z
7
15
5
8
C
4
11
7
14
z
2
8
15
1
5
4

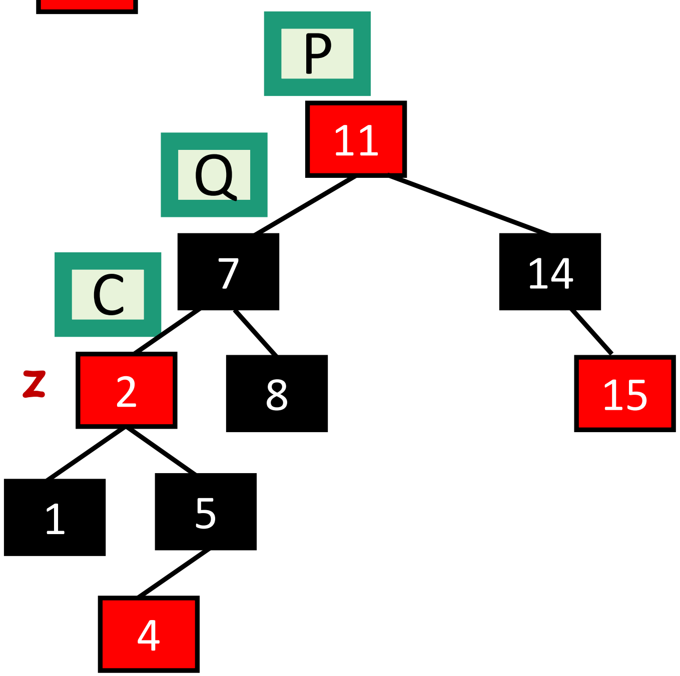
CASE 3



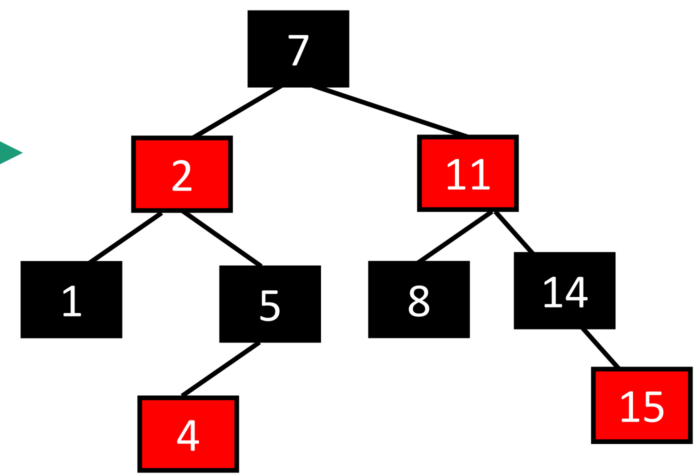
11
7
14
z
2
8
15
1
5
4



P
Q
Right
C
T1
T2
T3
T4

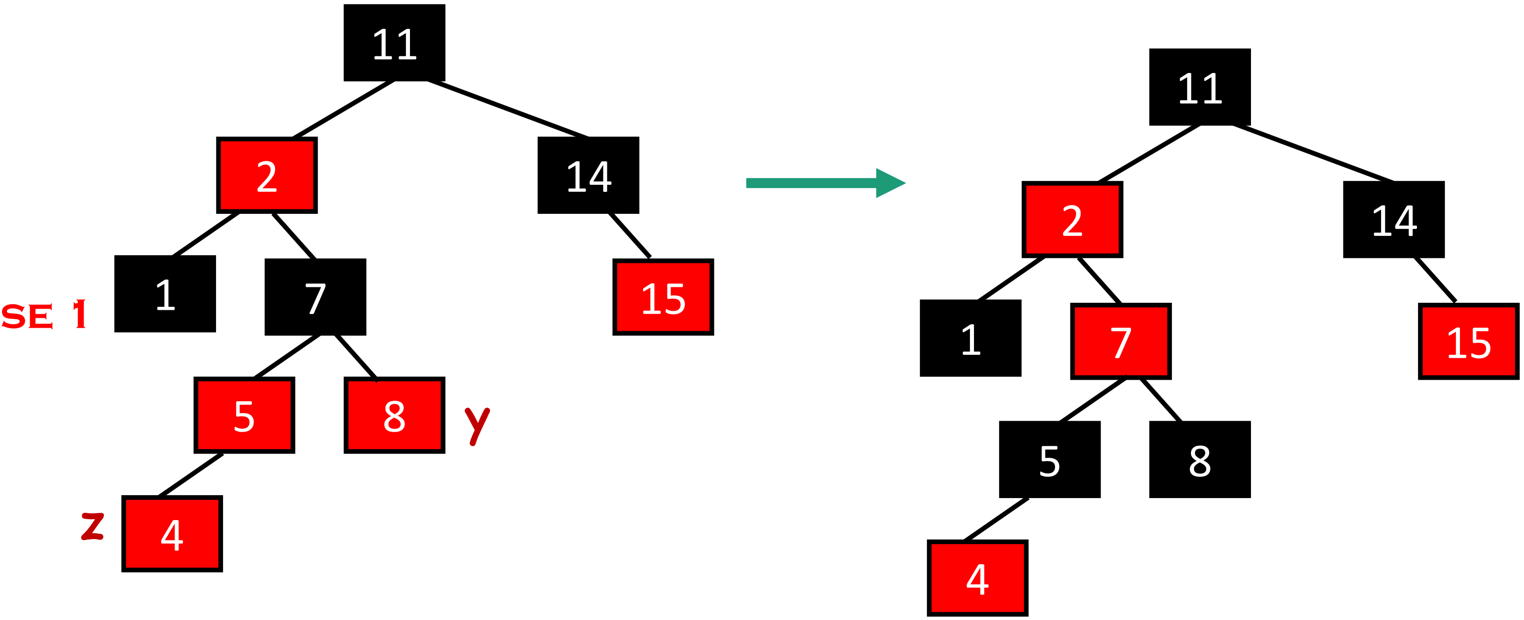


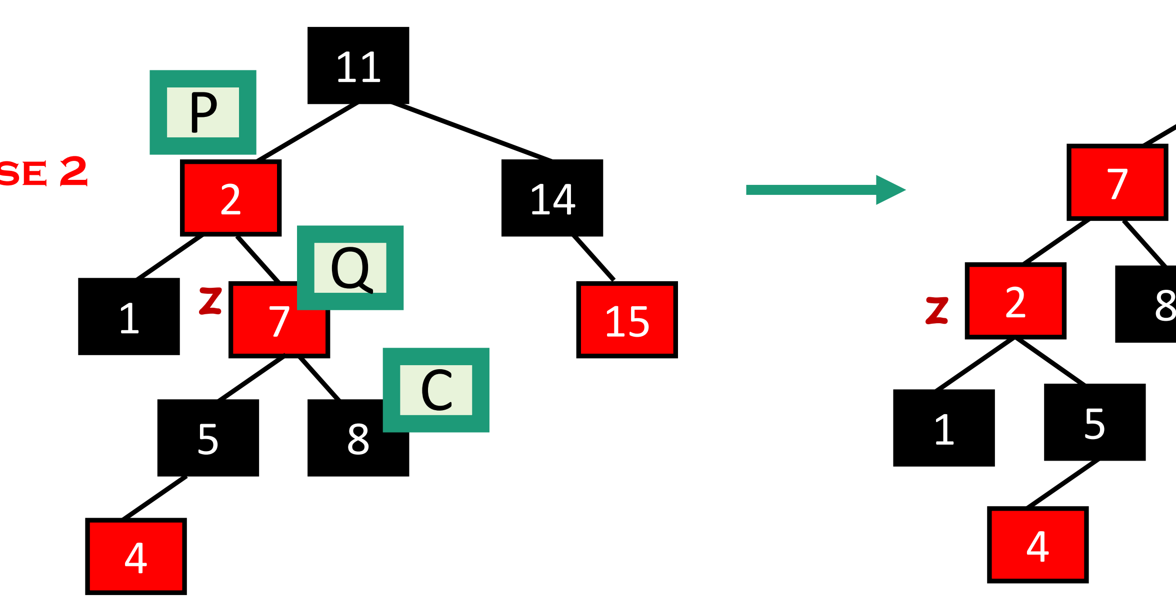
P
11
Q
7
14
C
z
2
8
15
1
5
4



7
2
11
1
5
8
14
4
15

```
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                           // case 1
            y.color = BLACK                             // case 1
            z.p.p.color = RED                           // case 1
            z = z.p.p                                   // case 1
        else if z == z.p.right
                z = z.p                                 // case 2
                LEFT-ROTATE(T, z)                       // case 2
            z.p.color = BLACK                           // case 3
            z.p.p.color = RED                           // case 3
            RIGHT-ROTATE(T, z.p.p)                      // case 3
    else (same as then clause
            with “right” and “left” exchanged)
T.root.color = BLACK
```






```
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                      // case 1
            y.color = BLACK                        // case 1
            z.p.p.color = RED                      // case 1
            z = z.p.p                              // case 1
        else if z == z.p.right
                z = z.p                            // case 2
                LEFT-ROTATE(T, z)                  // case 2
            z.p.color = BLACK                      // case 3
            z.p.p.color = RED                      // case 3
            RIGHT-ROTATE(T, z.p.p)                 // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

Deleting from a Red-Black tree

Fun exercise!

That’s a lot of cases

- You are **not responsible** for the nitty-gritty details of Red-Black Trees. (For this class)
  - Though implementing them is a great exercise!
- You should know:
  - What are the properties of an RB tree?
  - And (more important) why does that guarantee that they are balanced?

What was the point again?

- Red-Black Trees always have height at most 2log(n+1).
- As with general Binary Search Trees, all operations are O(height)
- So all operations are O(log(n)).

Conclusion: The best of both worlds

	Sorted Arrays	Linked Lists	Balanced Binary Search Trees
Search	O(log(n)) 😃	O(n) ☹️	O(log(n)) 😃
Insert/Delete	O(n) ☹️	O(1) 😃	O(log(n)) 😃