Introduction to Algorithms

L6. Tree

Instructor : Kilho Lee

Course Overview

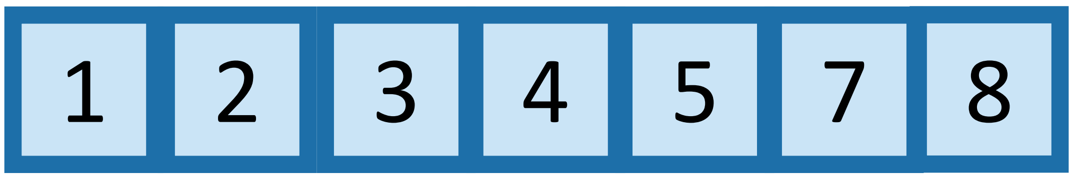
- Algorithmic Analysis
- Divide and Conquer
- Randomized Algorithms
- **Tree Algorithms**
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Algorithms

Today's Outline

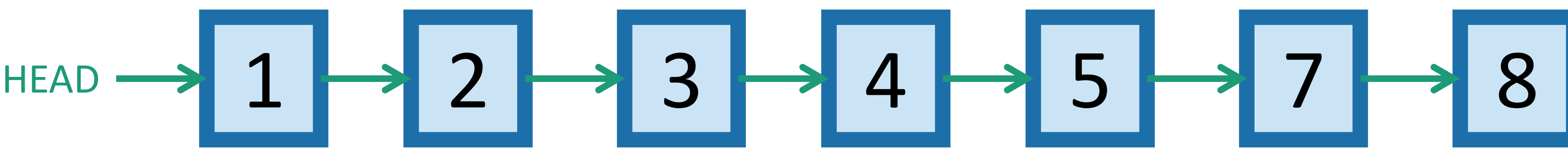
- Tree Algorithms
  - Tree basics, Binary search tree, Red-black tree
  - Reading: CLRS 12.1, 12.2, 12.3 and 13

Some data structures
for storing objects like $\boxed{5}$ (aka, **nodes** with **keys**)

- (Sorted) arrays:

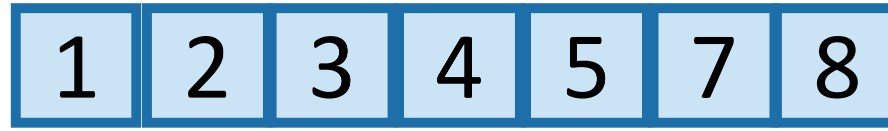


- (Sorted) linked lists:

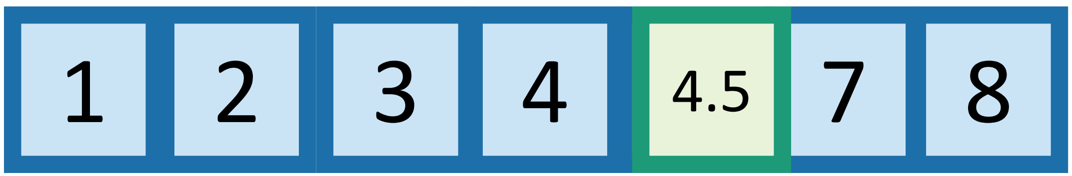


- Some basic operations:
  - INSERT, DELETE, SEARCH

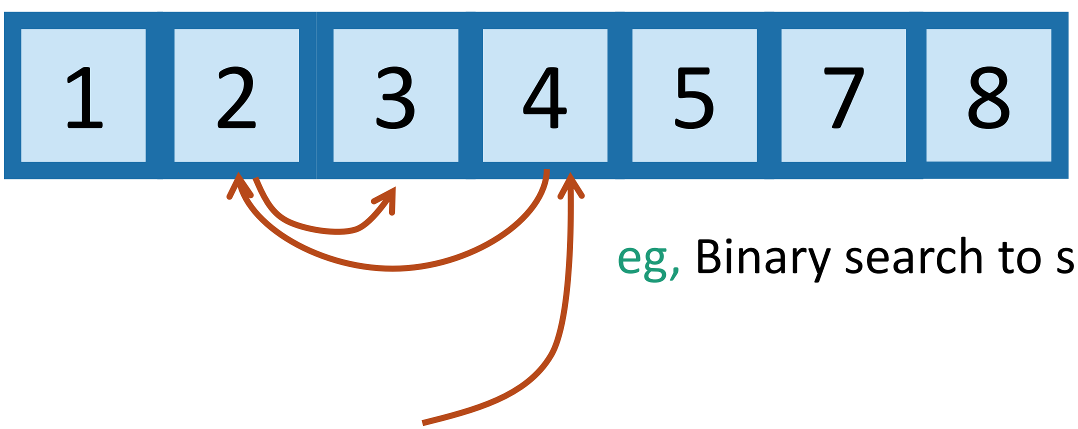
Sorted Arrays



- O(n) INSERT/DELETE:

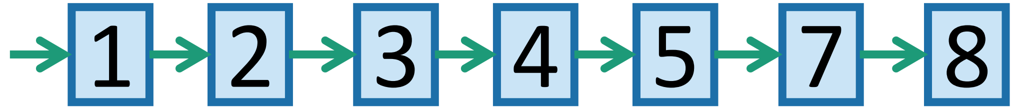


- O(log(n)) SEARCH:

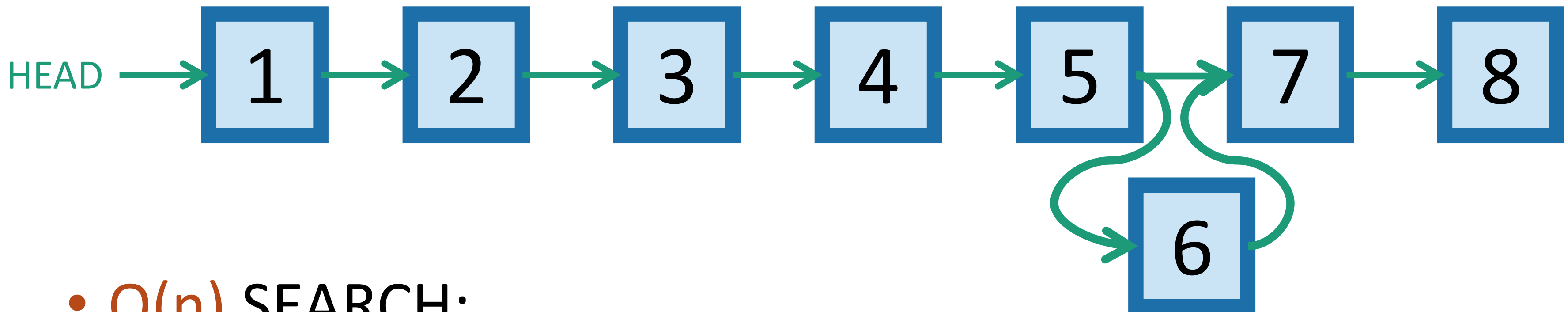


eg, Binary search to see if 3 is in A.

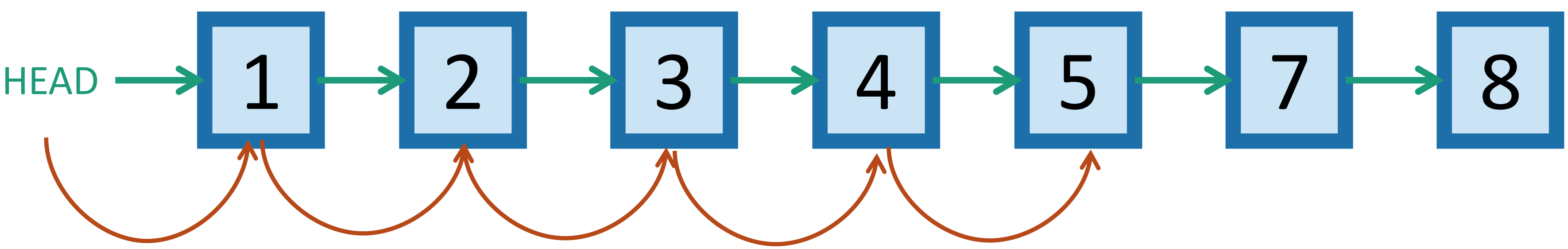
Sorted linked lists



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



- O(n) SEARCH:



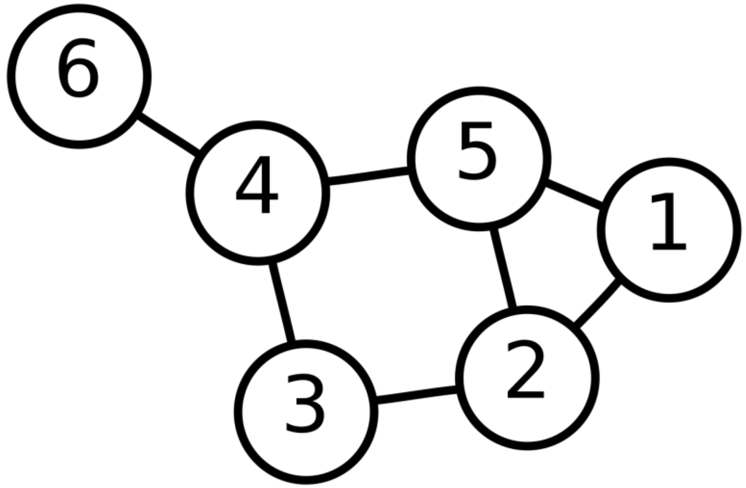
Motivation for Binary Search Trees

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))

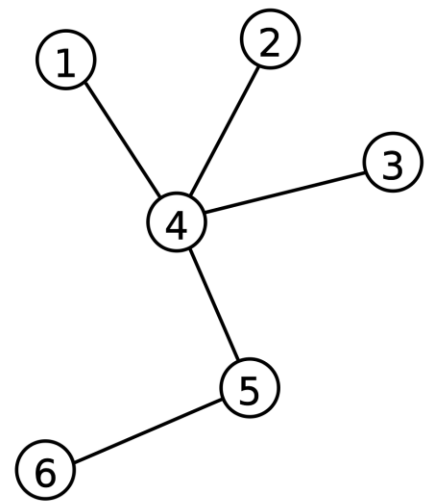
Binary Search Trees

Recall the data structure



- Graph
  - Graph G = (V,E) is a pair of the vertex set V and the edge set of E

Recall the data structure



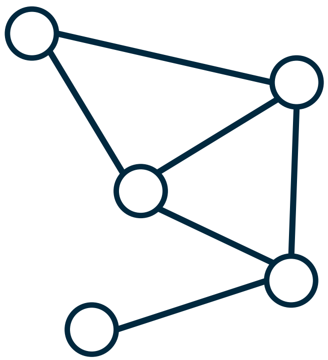
- Tree

  - Let G = (V,E) be an undirected graph, G is a (free) tree iff,

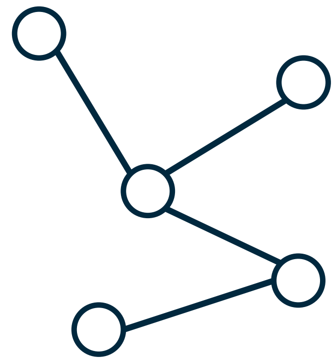
    1. Any two vertices in G are connected by a unique simple path
    2. G is connected, but if any edge is removed from E, the resulting graph is disconnected
    3. G is connected, and |E| = |V| - 1
    4. G is acyclic, and |E| = |V| - 1
    5. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

  - In short, a connected acyclic undirected graph

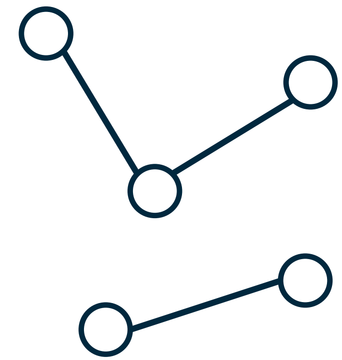
Recall the data structure



Not tree
(graph)

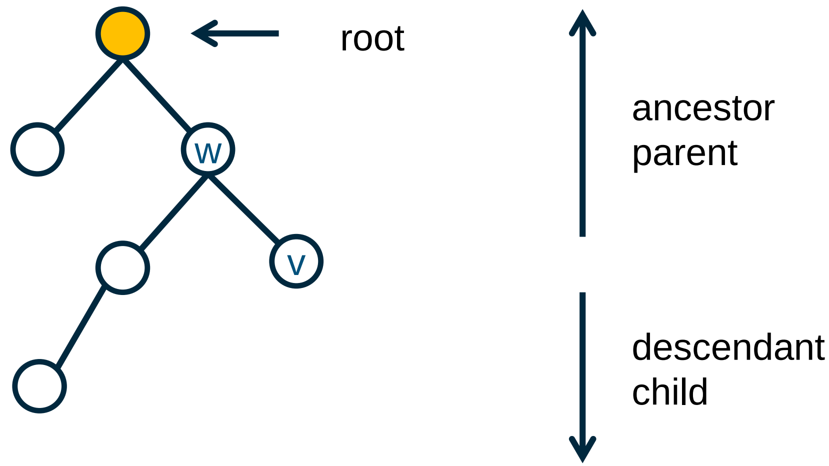


Tree



Not tree
(forest)

Recall the data structure



- Rooted tree
  - A free tree in which one of the vertices is distinguished from the others (the root)
  - In a rooted tree, each node has parent-child and ancestor-descendant relationships.
  - The root has no parent/ancestor.

For today all keys are distinct.

Binary tree terminology

Each node has two children.

The left child of 3 is 2

The right child of 3 is 4

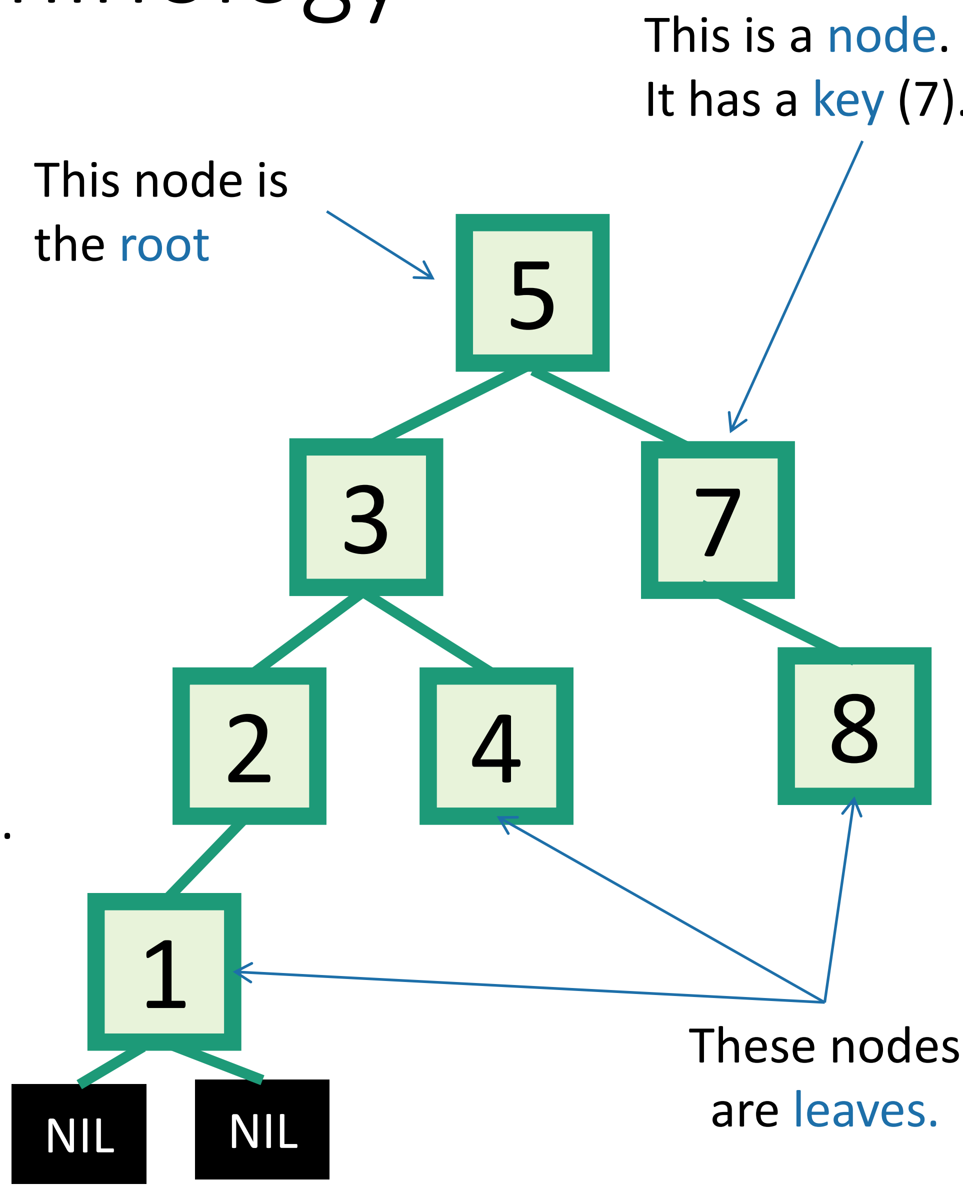
The parent of 3 is 5

2 is a descendant of 5

Each node has a pointer to its left child, right child, and parent.

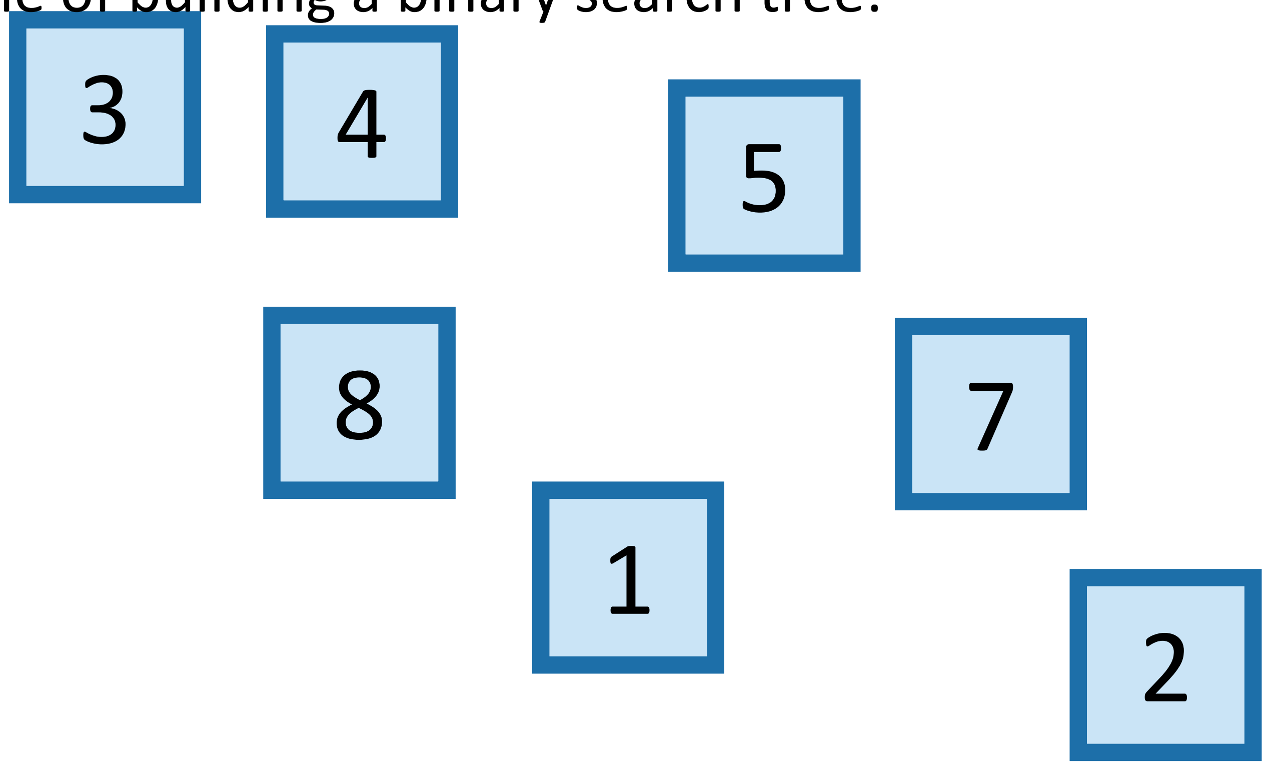
Both children of 1 are NIL.
(I won't usually draw them).

The height of this tree is 3.
(Max number of edges from the root to a leaf).



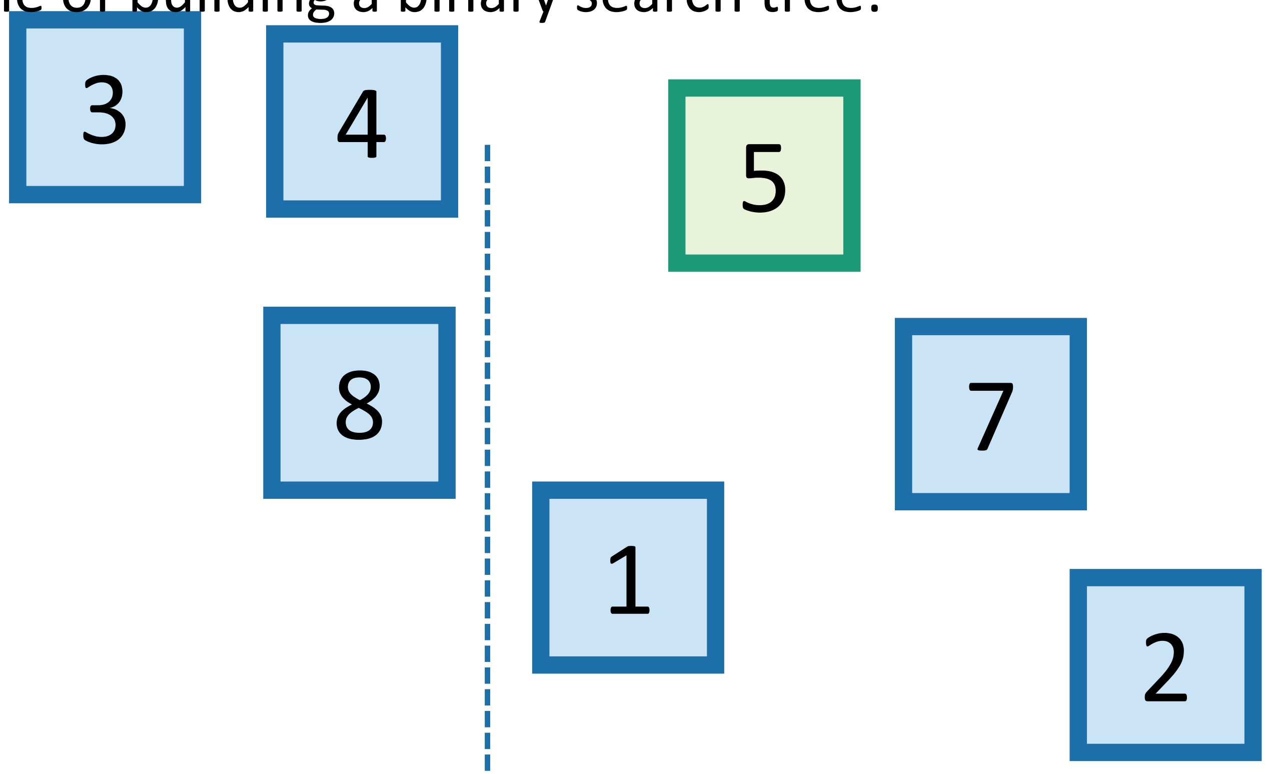
Binary Search Trees

- It's a binary search tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - (노드 x의 모든 왼쪽 후손들의 key는 x의 key 보다 작다)
  - Every RIGHT descendant of a node has key larger than that node.
  - (노드 x의 모든 오른쪽 후손들의 key는 x의 key 보다 크다)
- Example of building a binary search tree:



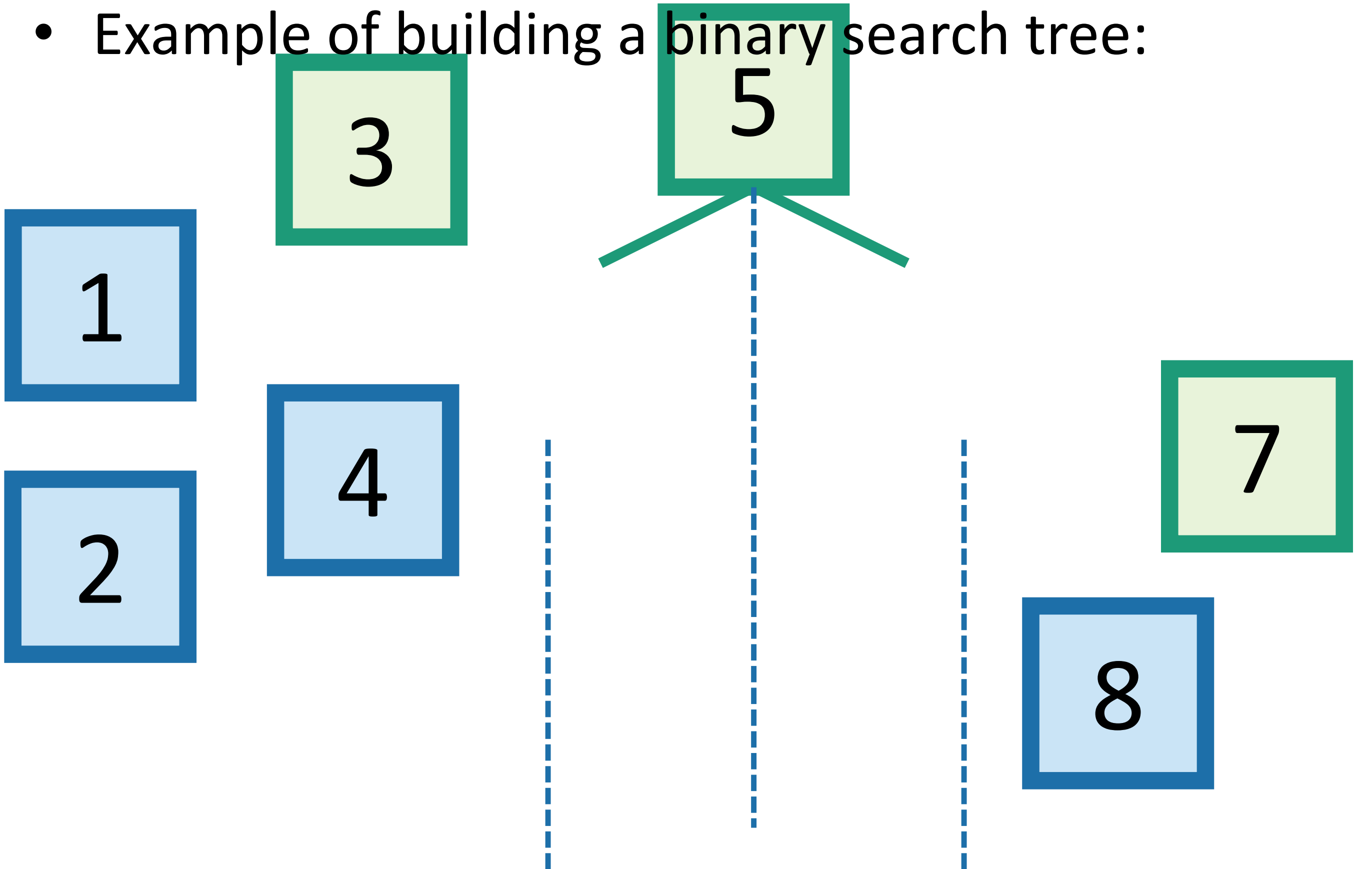
Binary Search Trees

- It’s a binary search tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - (노드 x의 모든 왼쪽 후손들의 key는 x의 key 보다 작다)
  - Every RIGHT descendant of a node has key larger than that node.
  - (노드 x의 모든 오른쪽 후손들의 key는 x의 key 보다 크다)
- Example of building a binary search tree:



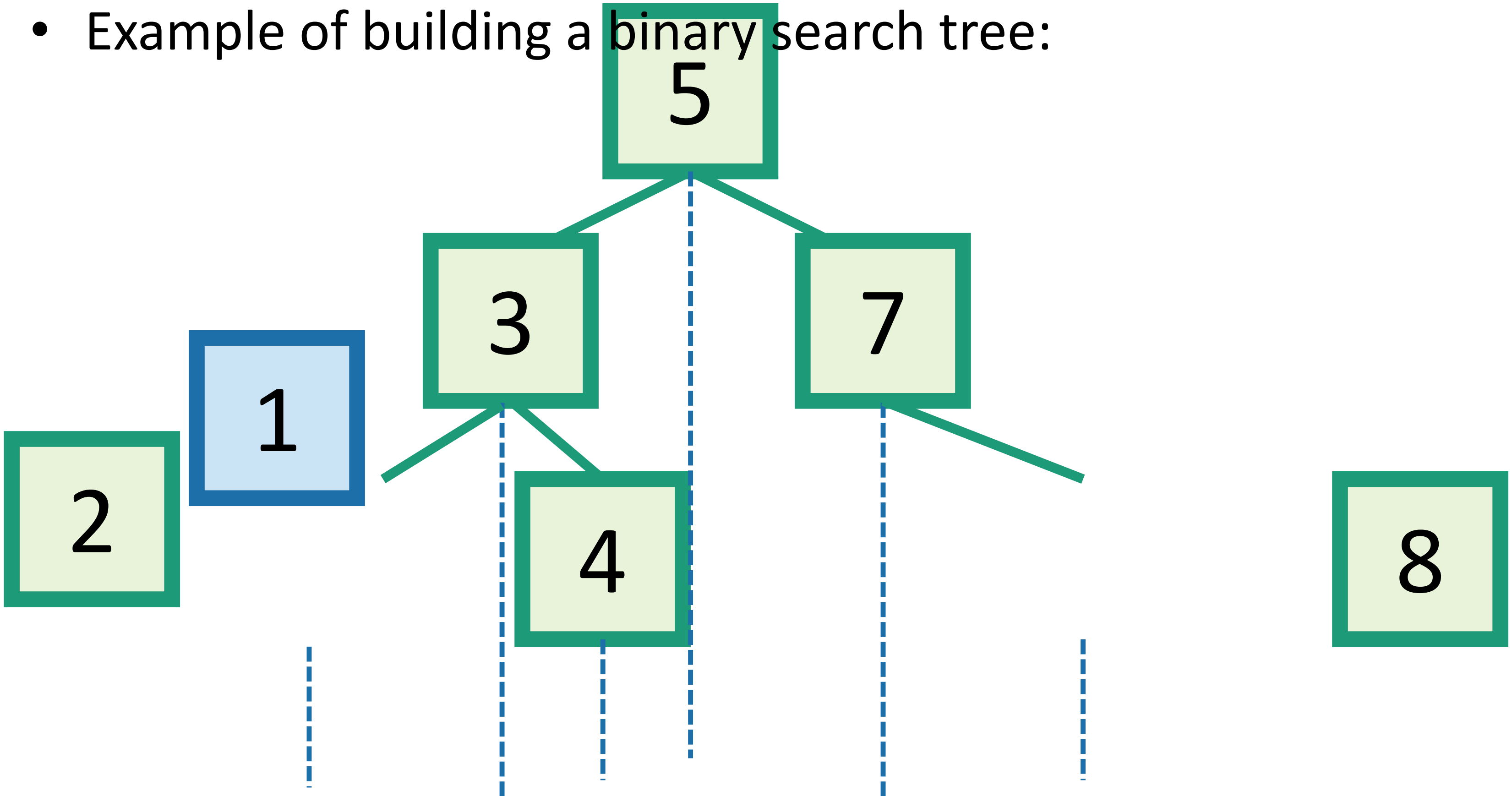
Binary Search Trees

- It's a binary search tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - (노드 x의 모든 왼쪽 후손들의 key는 x의 key 보다 작다)
  - Every RIGHT descendant of a node has key larger than that node.
  - (노드 x의 모든 오른쪽 후손들의 key는 x의 key 보다 크다)
- Example of building a binary search tree:



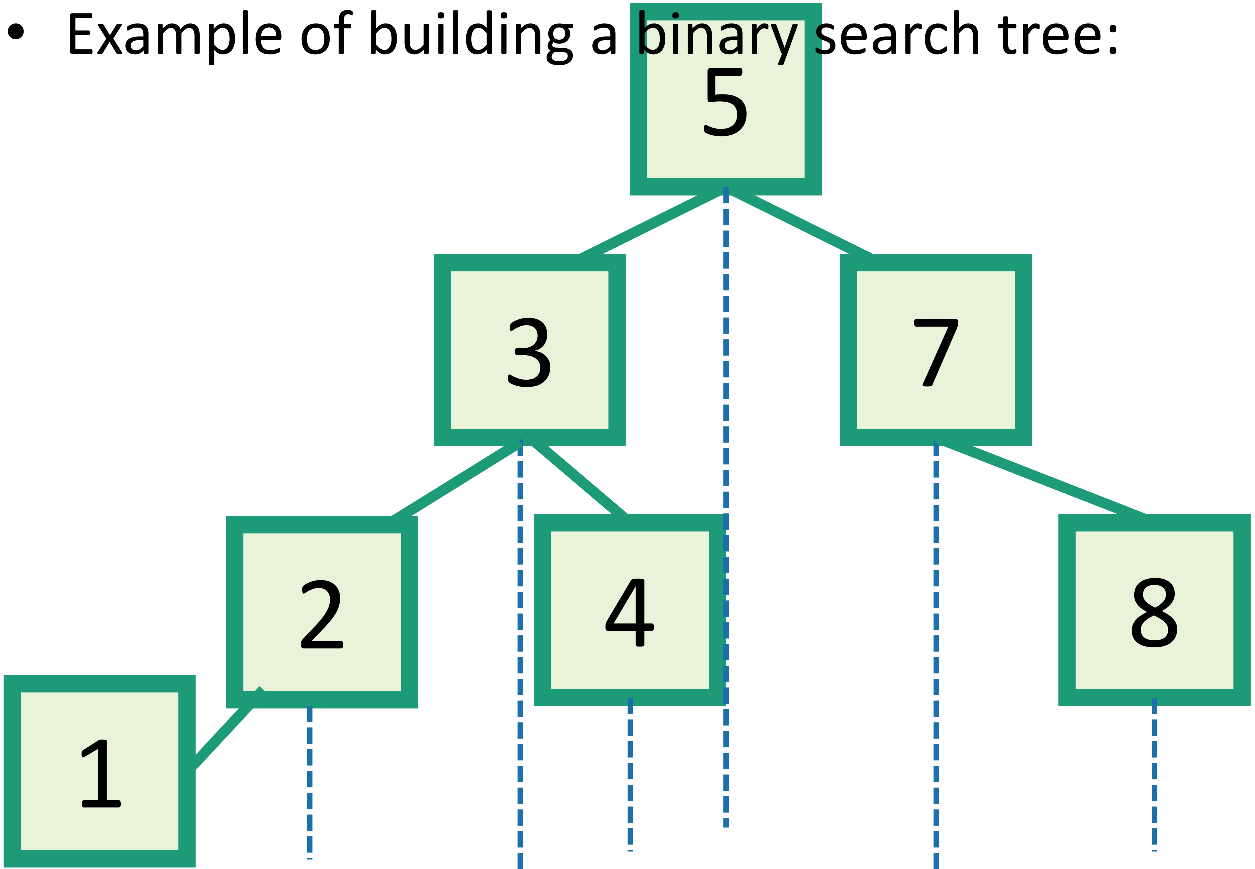
Binary Search Trees

- It's a binary search tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - (노드 x의 모든 왼쪽 후손들의 key는 x의 key 보다 작다)
  - Every RIGHT descendant of a node has key larger than that node.
  - (노드 x의 모든 오른쪽 후손들의 key는 x의 key 보다 크다)
- Example of building a binary search tree:



Binary Search Trees

- It's a binary search tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - (노드 x의 모든 왼쪽 후손들의 key는 x의 key 보다 작다)
  - Every RIGHT descendant of a node has key larger than that node.
  - (노드 x의 모든 오른쪽 후손들의 key는 x의 key 보다 크다)
- Example of building a binary search tree:

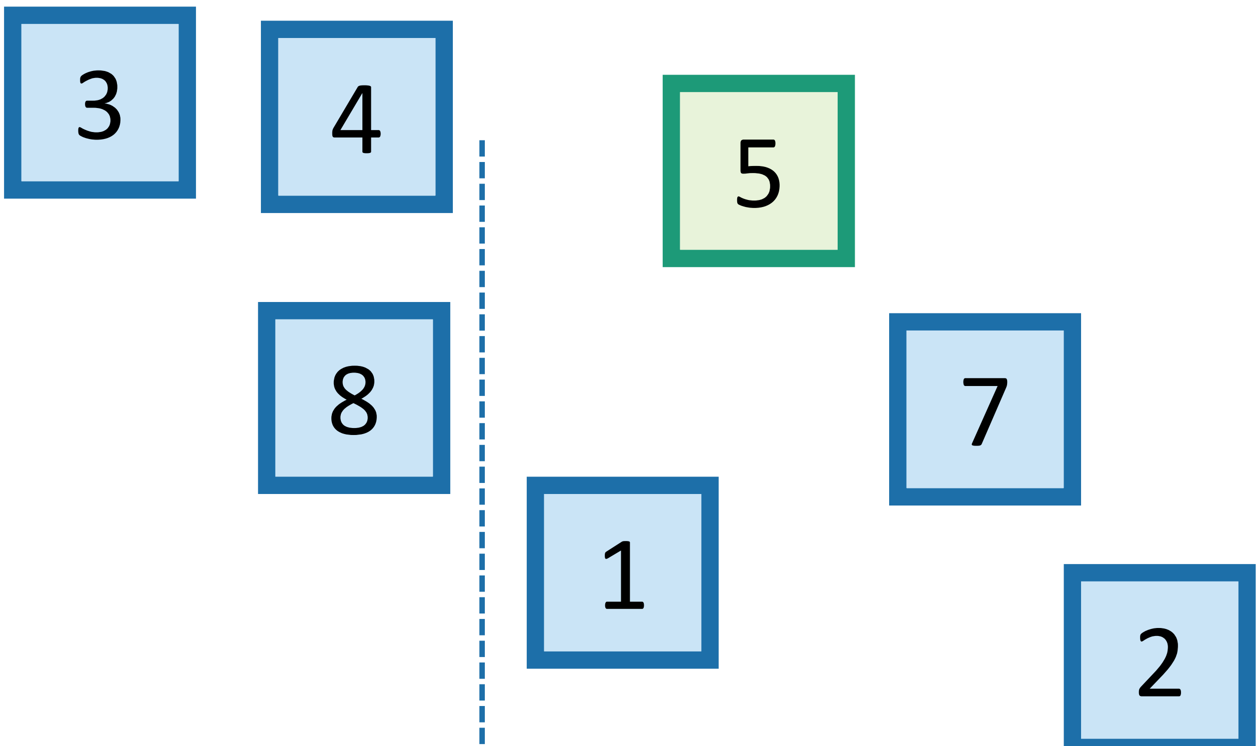


Q: Is this the only binary search tree I could possibly build with these values?

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

Aside: this should look familiar

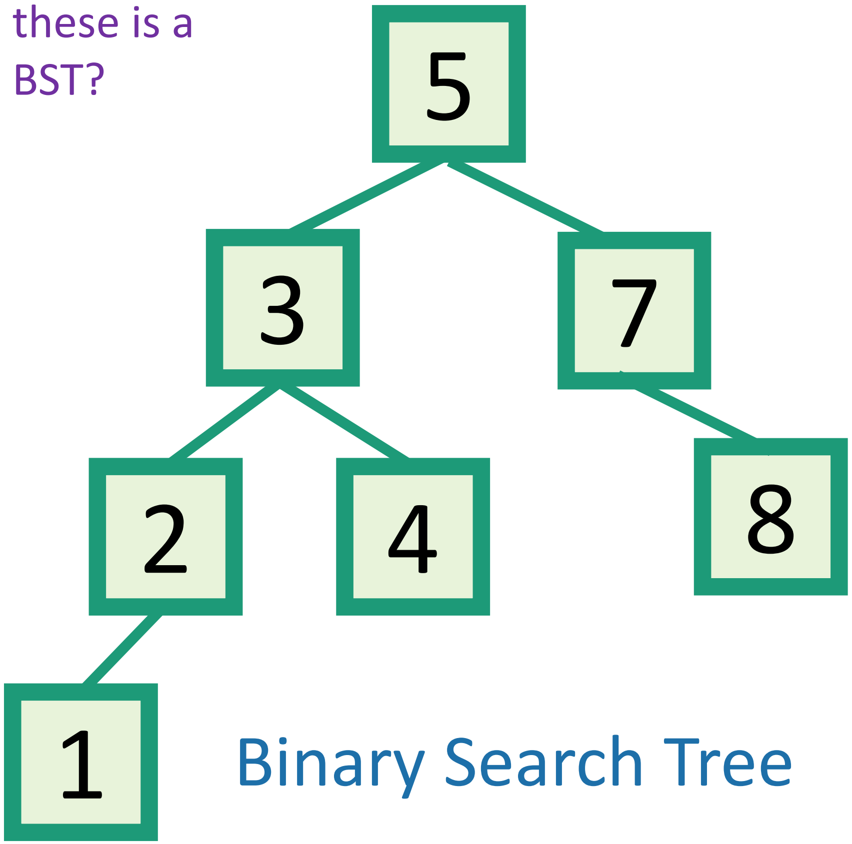
kinda like QuickSort



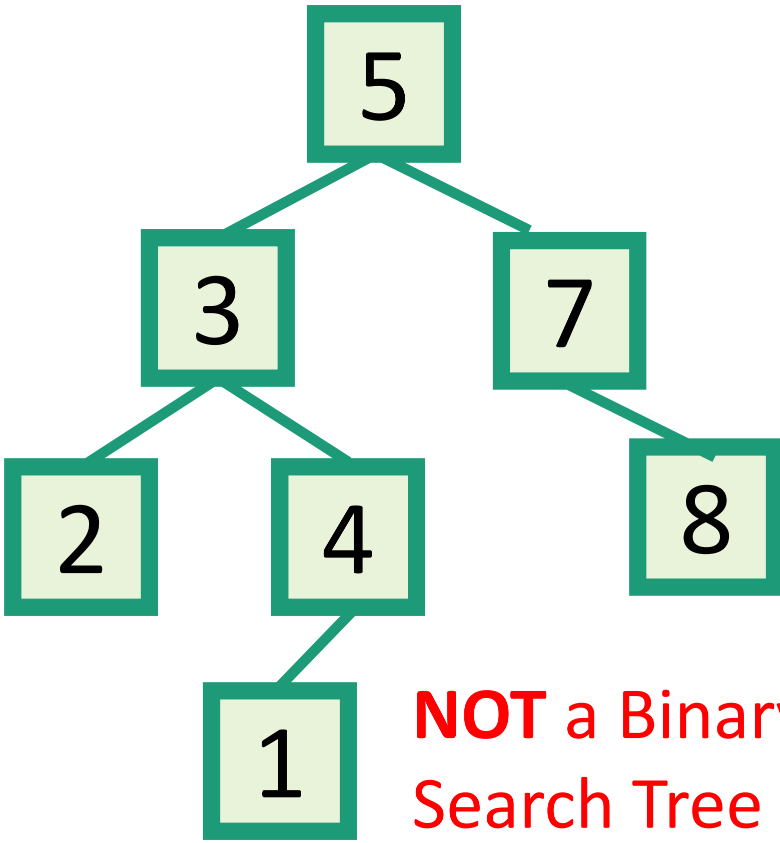
Binary Search Trees

- 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.

Which of these is a BST?



Binary Search Tree



NOT a Binary Search Tree

Remember the goal

Fast SEARCH/INSERT/DELETE

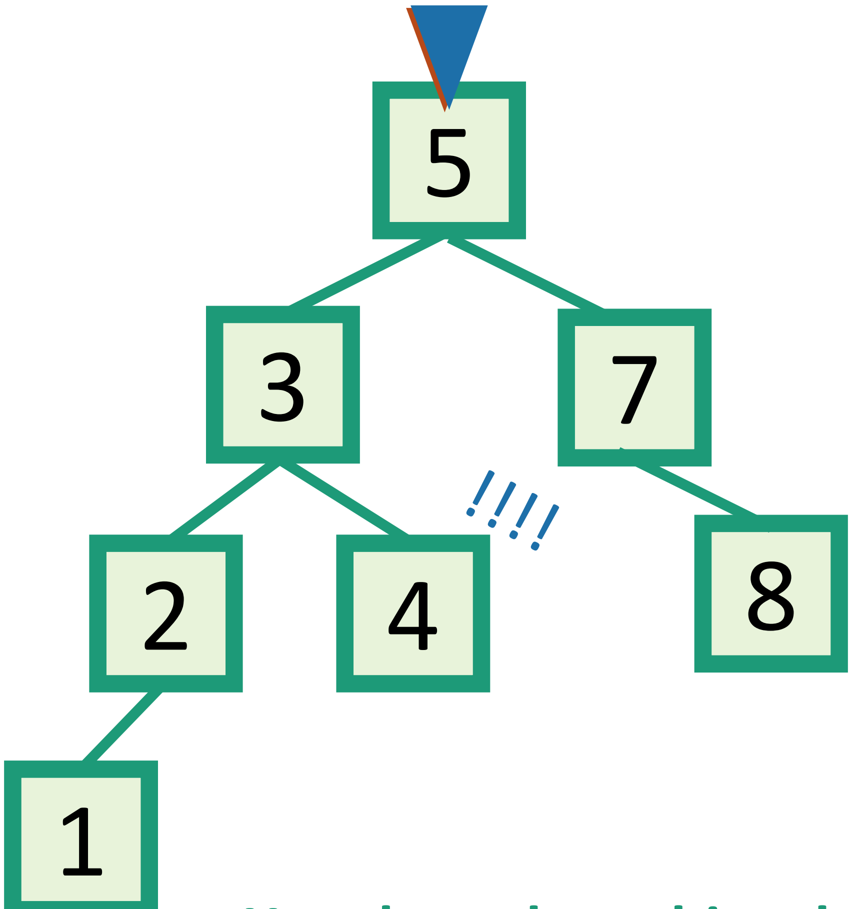
Can we do these?

Motivation for Binary Search Trees

	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))

SEARCH in a Binary Search Tree

definition by example



EXAMPLE: Search for 4.

EXAMPLE: Search for 4.5

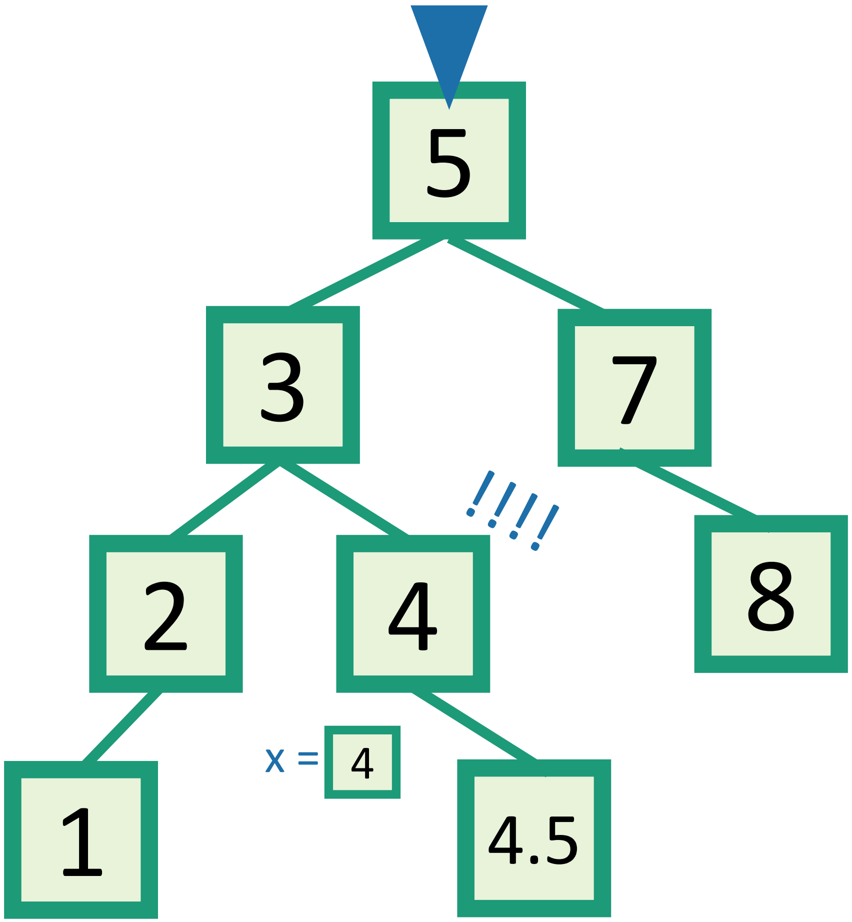
- It turns out it will be convenient to **return 4** in this case
- (that is, **return** the last node before we went off the tree)

Write pseudocode (or actual code) to implement this!

How long does this take?

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

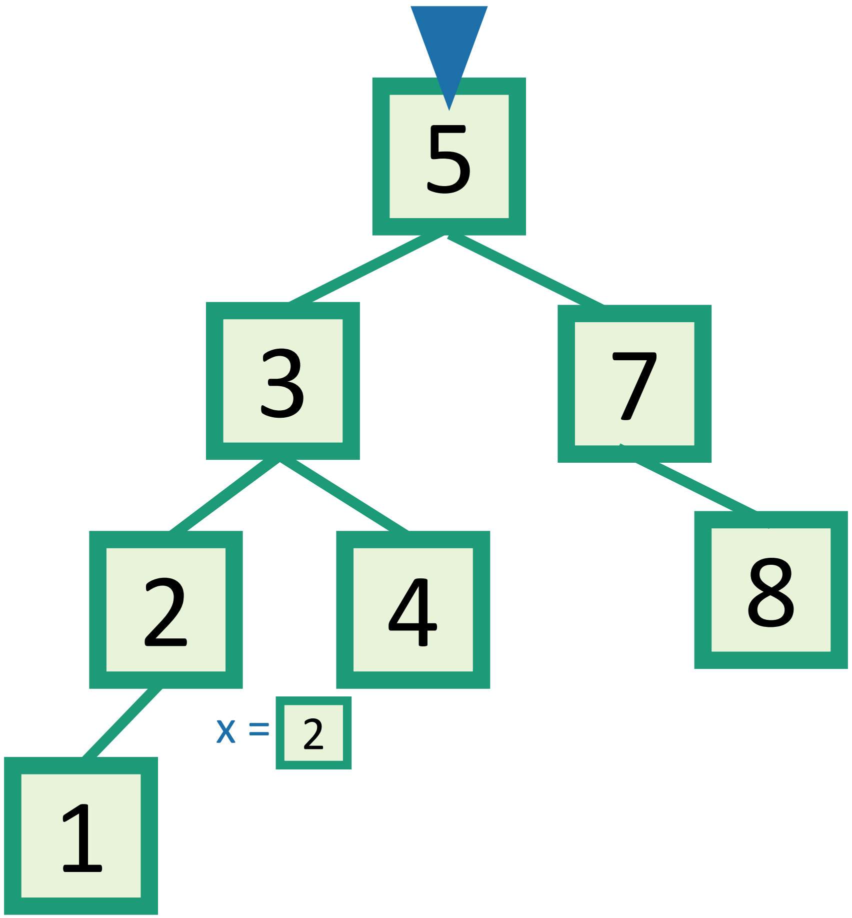
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

- INSERT(key):
  - x = SEARCH(key)
  - **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

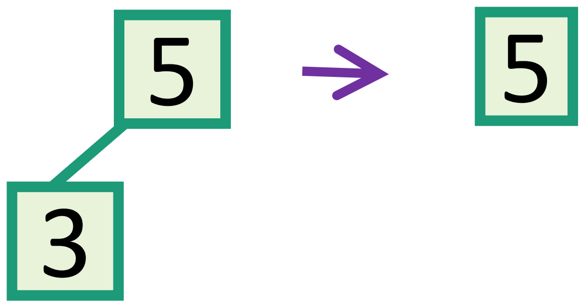
- DELETE(key):
  - x = SEARCH(key)
  - **if** x.key == key:
    - ….delete x….

This is a bit more complicated…

DELETE in a Binary Search Tree

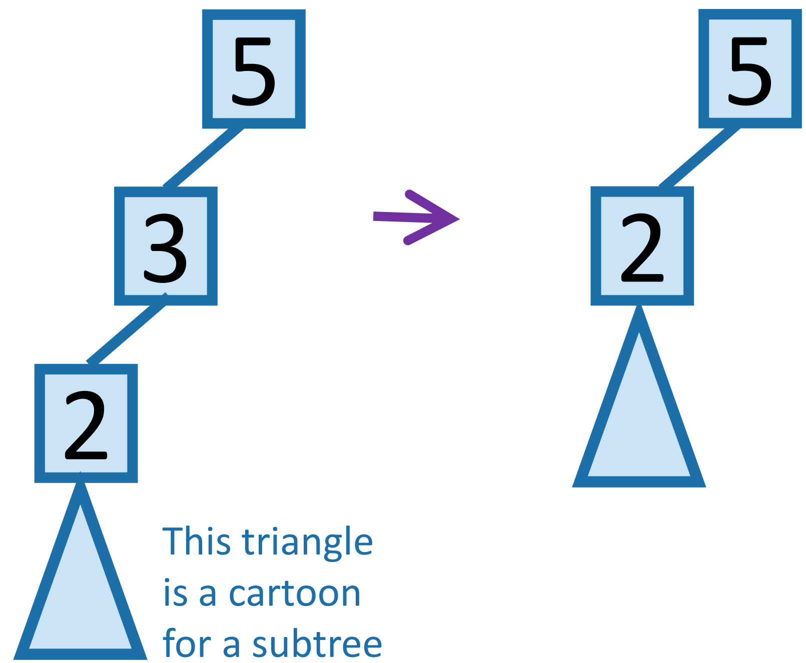
several cases (by example)

say we want to delete 3



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

Write pseudocode for all of these!

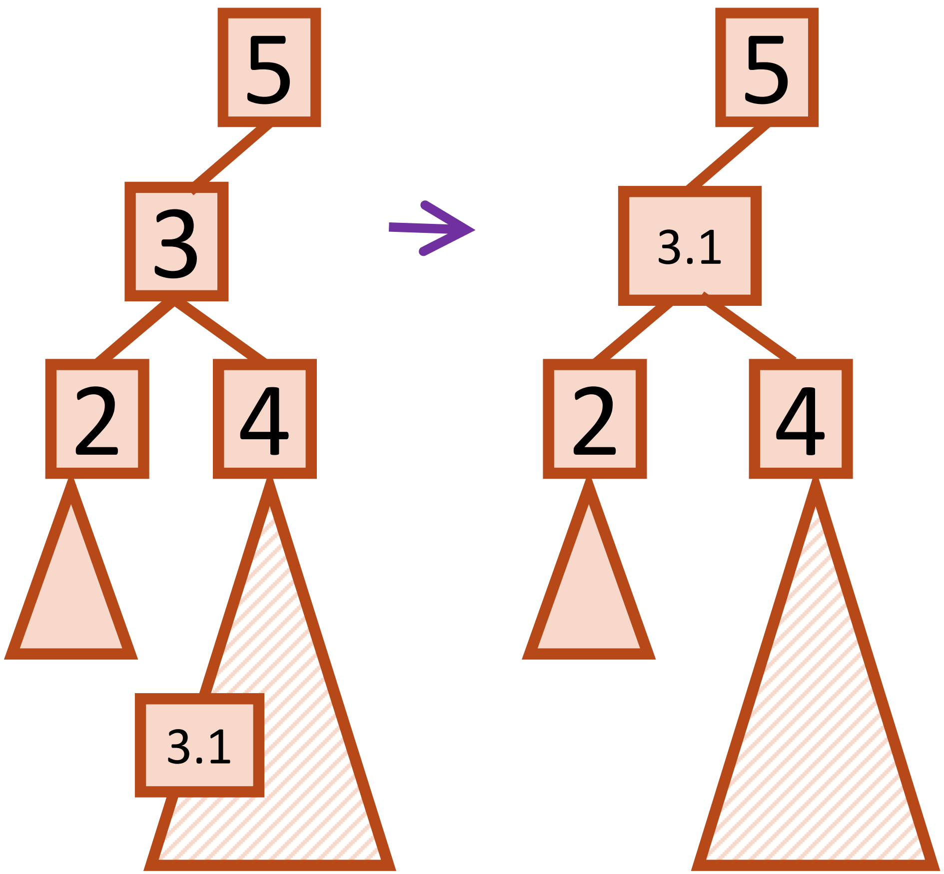


Case 2: if 3 has just one child,
move that up.

DELETE in a Binary Search Tree

ctd.

Case 3: if 3 has two children,
replace 3 with it's immediate successor.
(aka, next smallest 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.

```
#include <iostream>
using namespace std;

struct node {
  int key;
  struct node *left, *right;
};

// Create a node
struct node *newNode(int item) {
  struct node *temp = (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}
```

```c
// Insert a node
struct node *insert(struct node *node, int key) {
  // Return a new node if the tree is empty
  if (node == NULL) return newNode(key);

  // Traverse to the right place and insert the node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}

struct node *search(struct node* node, int key) {
  if (node == NULL) return NULL;
  if (key == node->key)  return node;
  else if (key < node->key)  return search(node->left, key);
  else if (key > node->key)  return search(node->right, key);
}
```

```c
// Deleting a node
struct node *deleteNode(struct node *root, int key) {
  // Return if the tree is empty
  if (root == NULL) return root;

  // Find the node to be deleted
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    // If the node is with only one child or no child
    if (root->left == NULL) {
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }

    // If the node has two children
    struct node *temp = minValueNode(root->right);

    // Place the inorder successor in position of the node to be deleted
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}
```

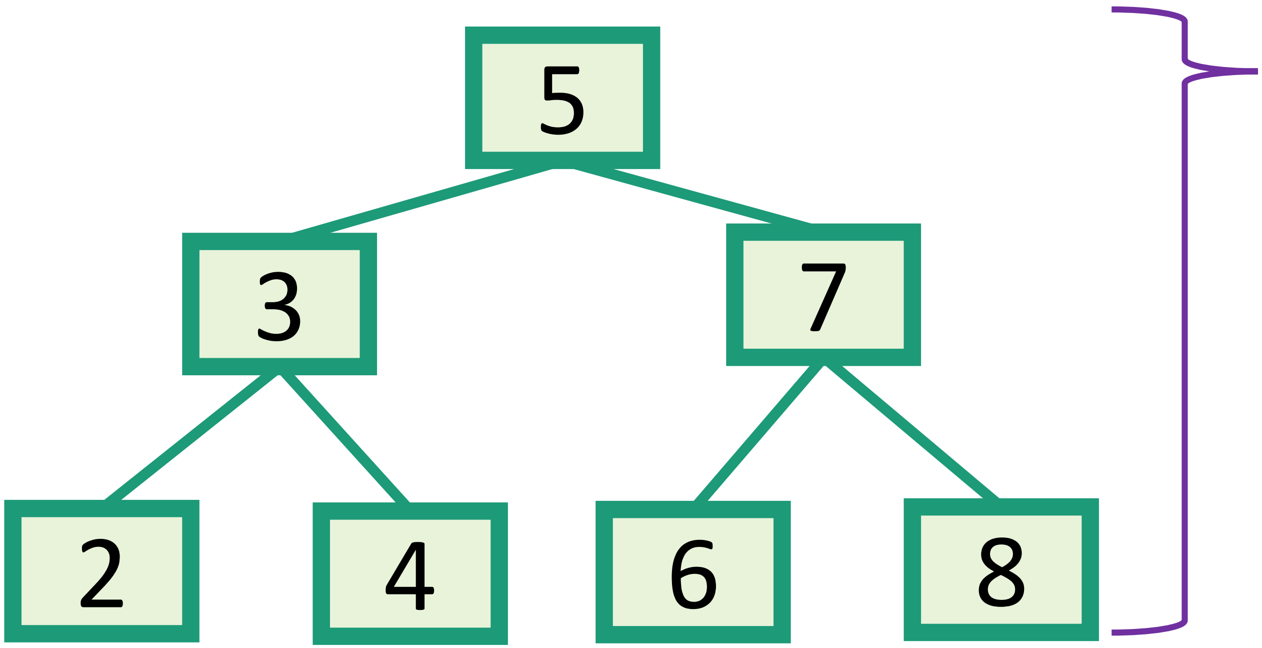
```c
// Find the inorder successor
struct node *minValueNode(struct node *node) {
  struct node *current = node;

  // Find the leftmost leaf
  while (current && current->left != NULL)
    current = current->left;

  return current;
}
```

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.

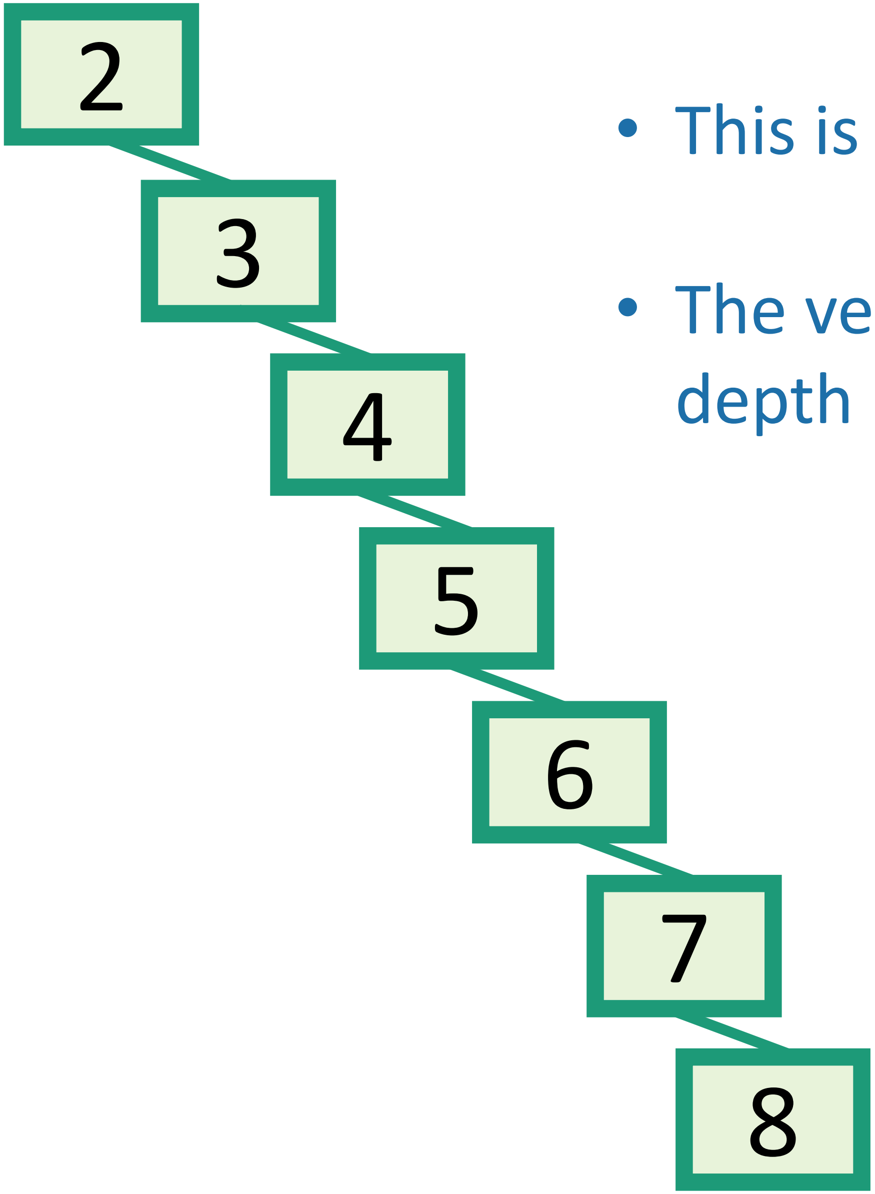


Time = O(height of tree)

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.

How often is "every so often" in the worst case? It's actually pretty often!

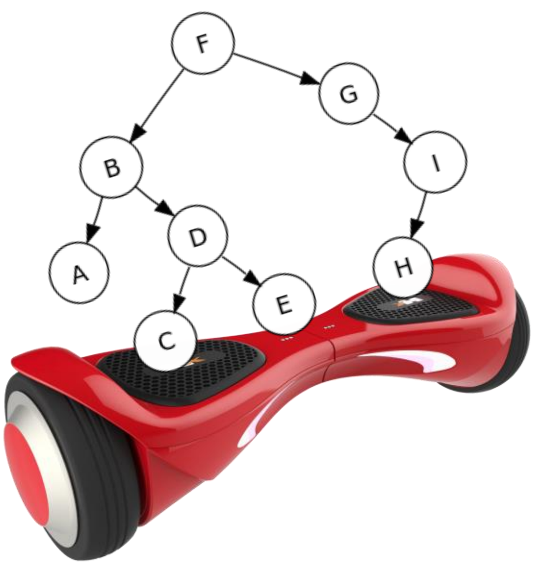
What to do?

- 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.
    - At least $\Omega(n)$ every so often....

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

Self-Balancing Binary Search Trees



Today’s Outline

- Tree Algorithms
  - Tree basics, Binary search tree, Red-black tree
  - Reading: CLRS 12.1, 12.2, 12.3 and 13

How often is “every so often” in the worst case? It’s actually pretty often!

What to do?

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

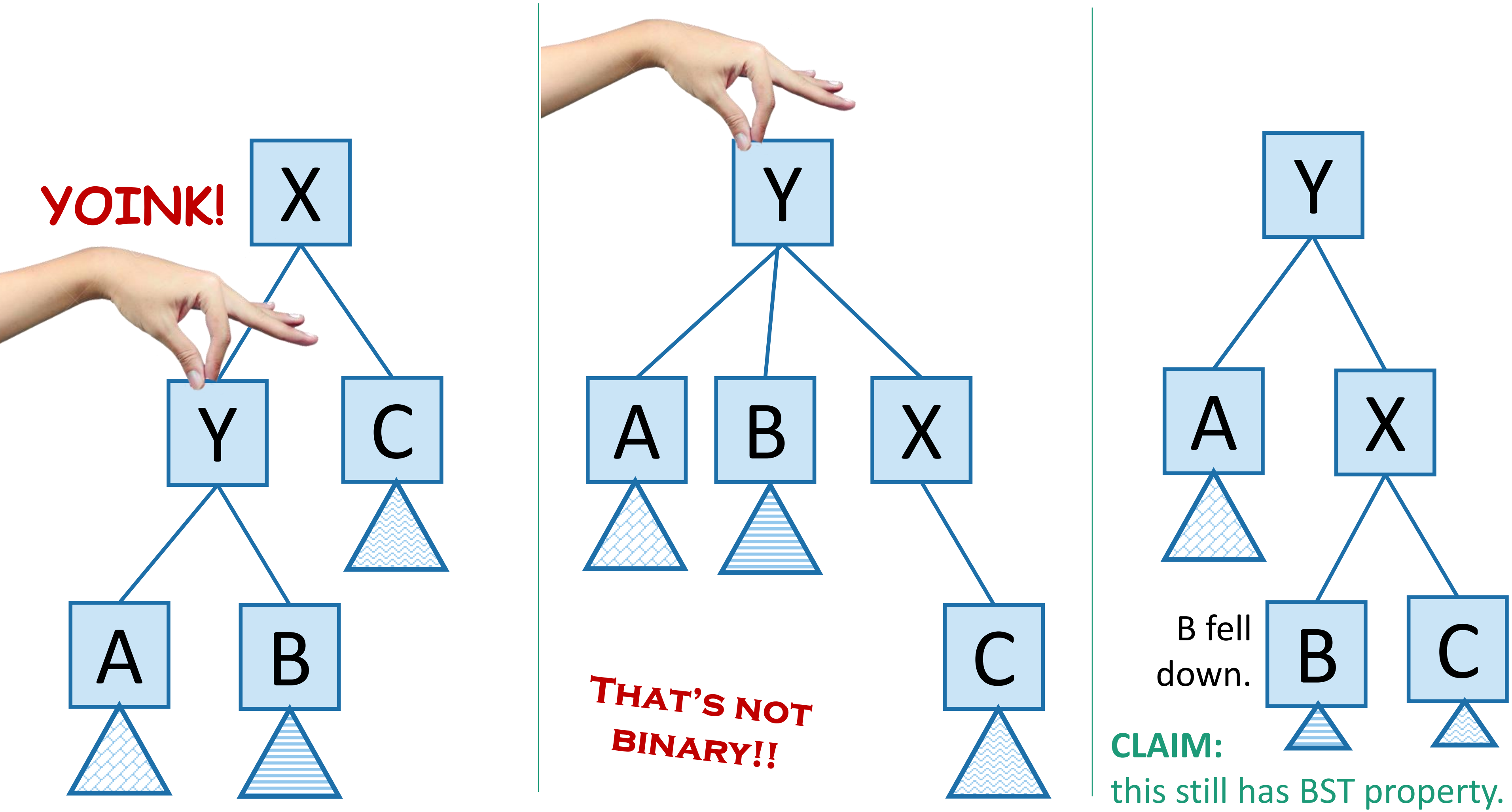
- Idea 0:
  - Keep track of how deep the tree is getting.
  - If it gets too tall, re-do everything from scratch.
    - At least $\Omega(n)$ every so often….

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

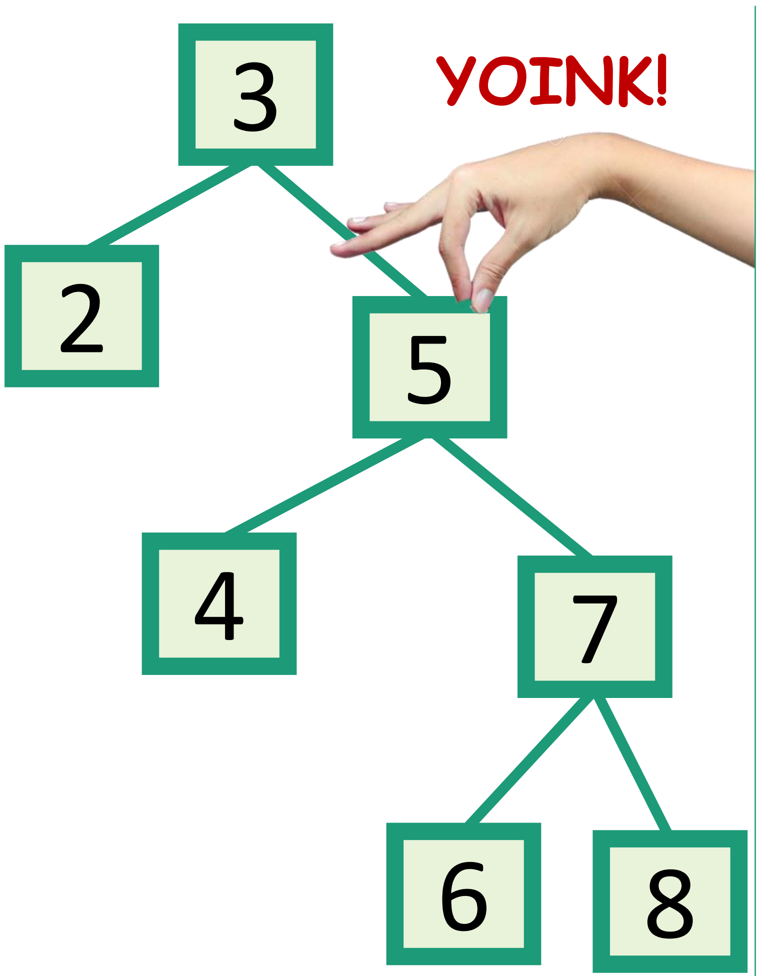
Idea 1: Rotations

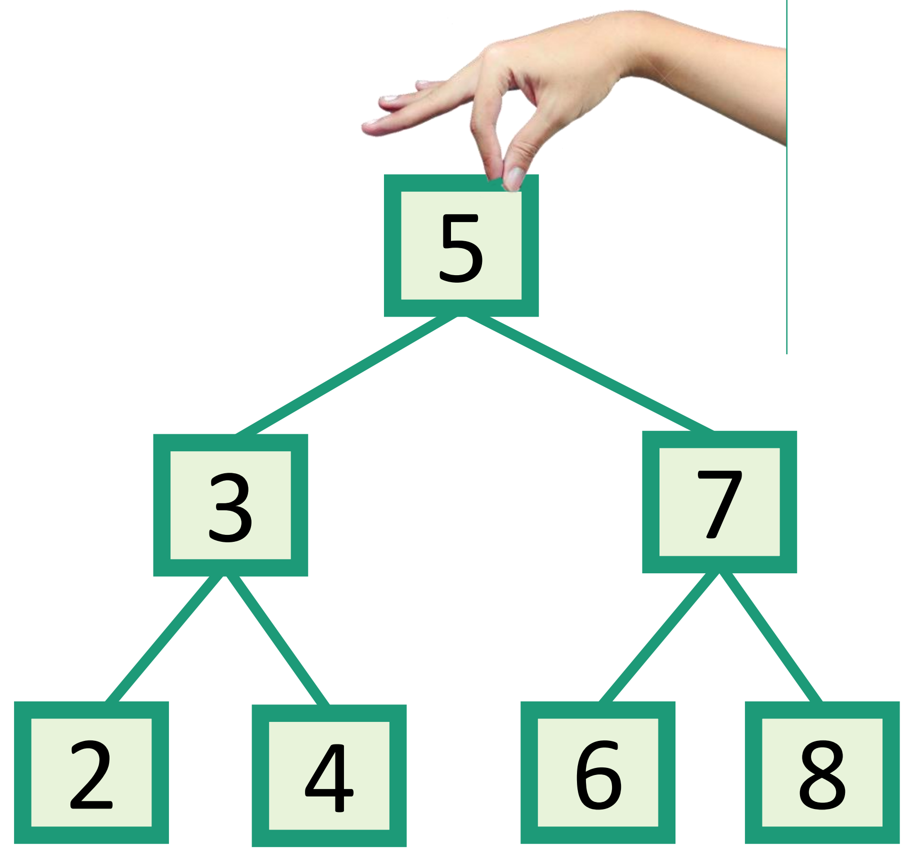
No matter what lives underneath A,B,C,
this takes time O(1). (Why?)

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

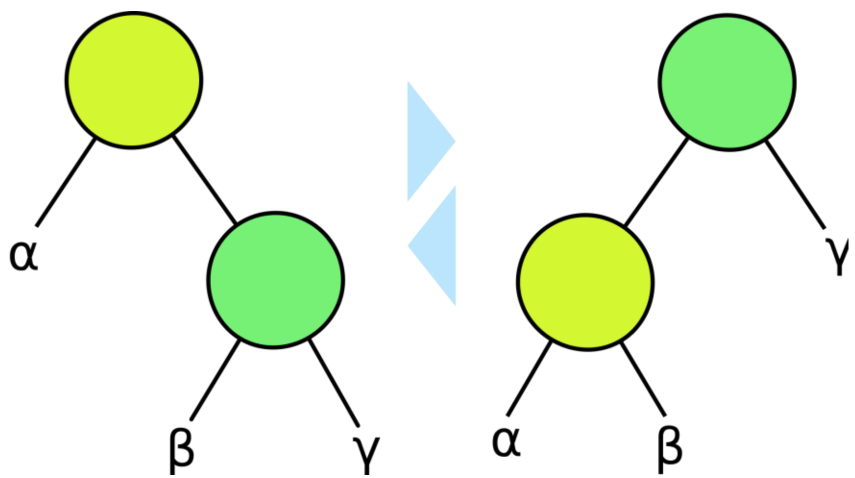


This seems helpful

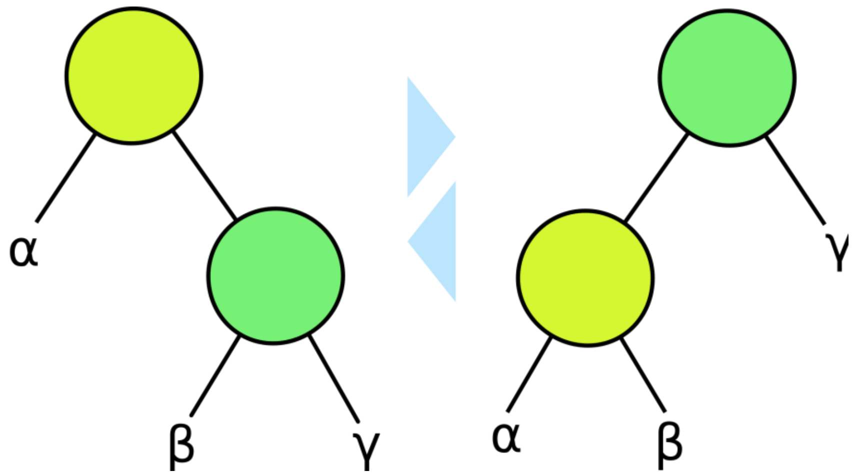


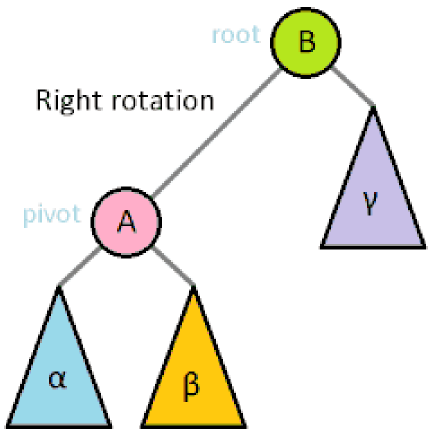


Tree rotation



Tree rotation





Does this work?

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

Even for me this is pretty vague.
What do we mean by "seems unbalanced"?
What's "okay"?

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.

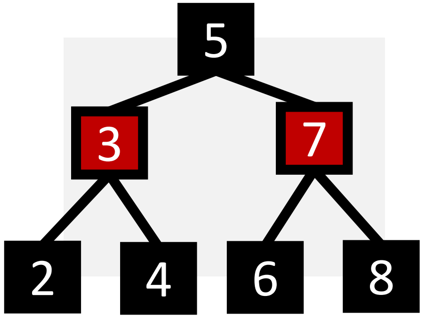
There are actually several ways to do this, but today we'll see…

Red-Black Trees

- A Binary Search Tree that balances itself!
- No more time-consuming by-hand balancing!

Red-Black tree!

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

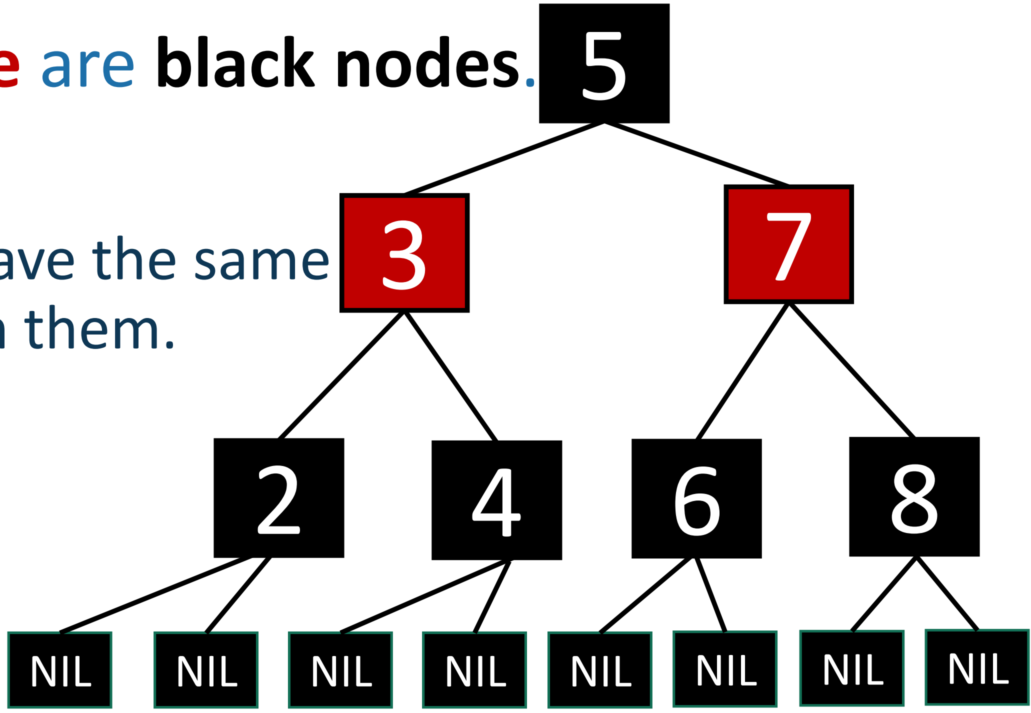


It's just good sense!

Red-Black Trees

these rules are the proxy for balance

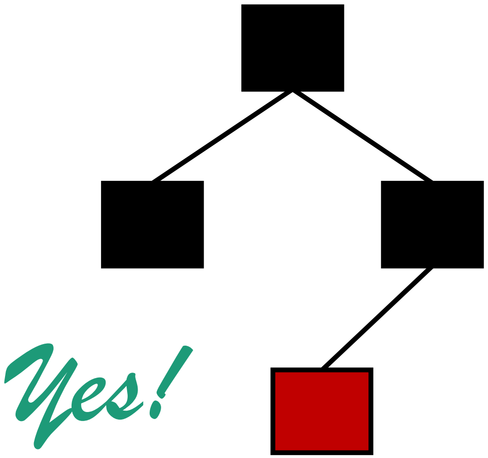
- P1. Every node is colored **red** or **black.**
- P2. The root node is a **black node**.
- P3. NIL children count as **black nodes**.
- P4. Children of a **red node** are **black nodes**.
- P5. For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.

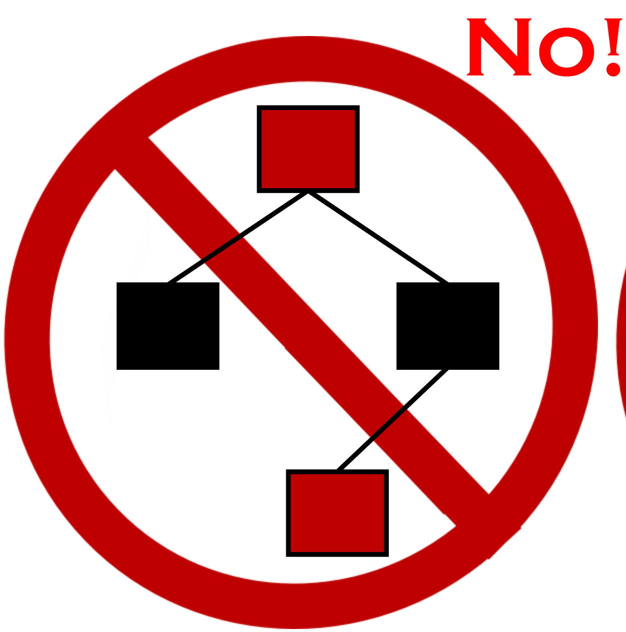


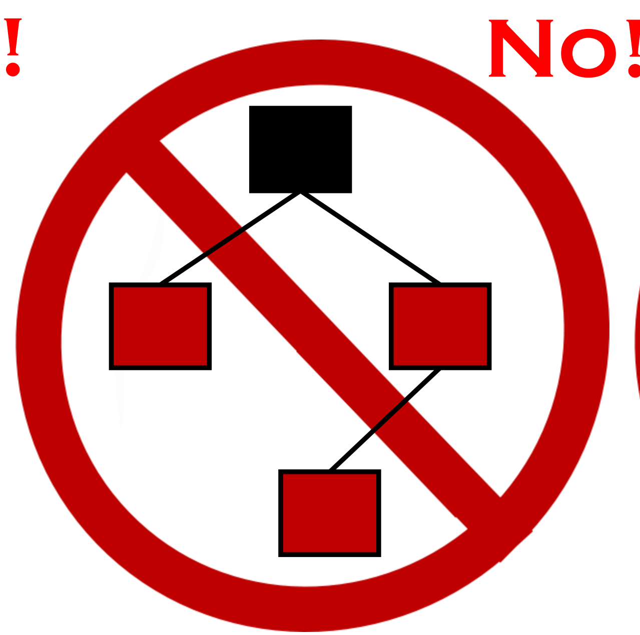
I'm not going to draw the NIL children in the future, but they are treated as black nodes.

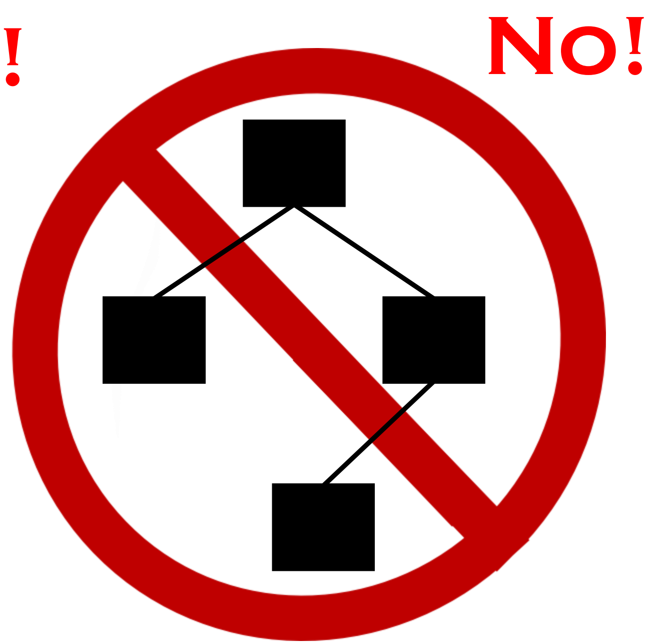
Examples(?)

- P1. Every node is colored **red** or **black.**
- P2. The root node is a **black node**.
- P3. NIL children count as **black nodes**.
- P4. Children of a **red node** are **black nodes**.
- P5. For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



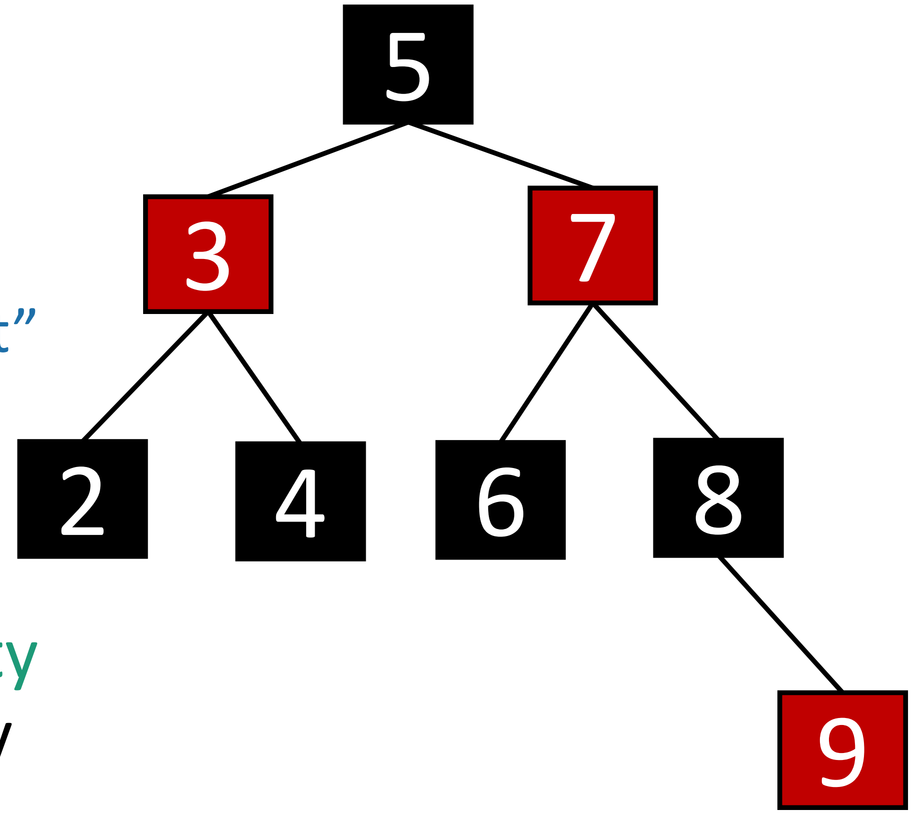






Why???????

- 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!

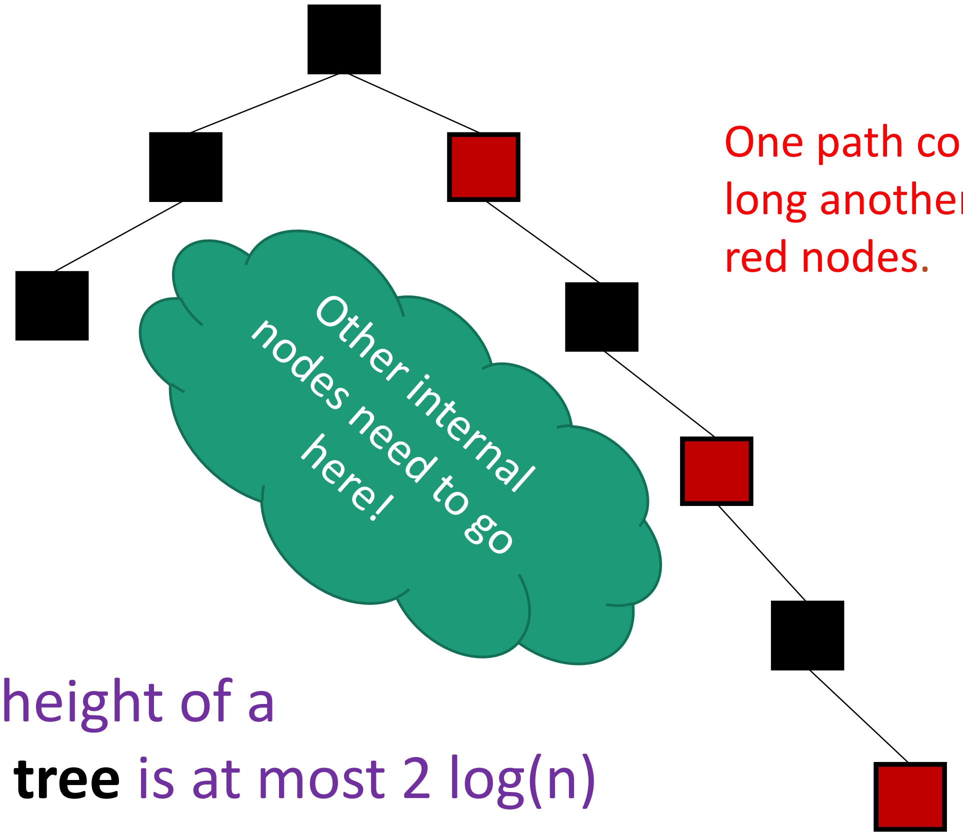
This **Red-Black** structure is a proxy for balance.

It's just a little weaker than perfect balance, but we can actually maintain it!

Let’s build some intuition!

This is “pretty balanced”

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



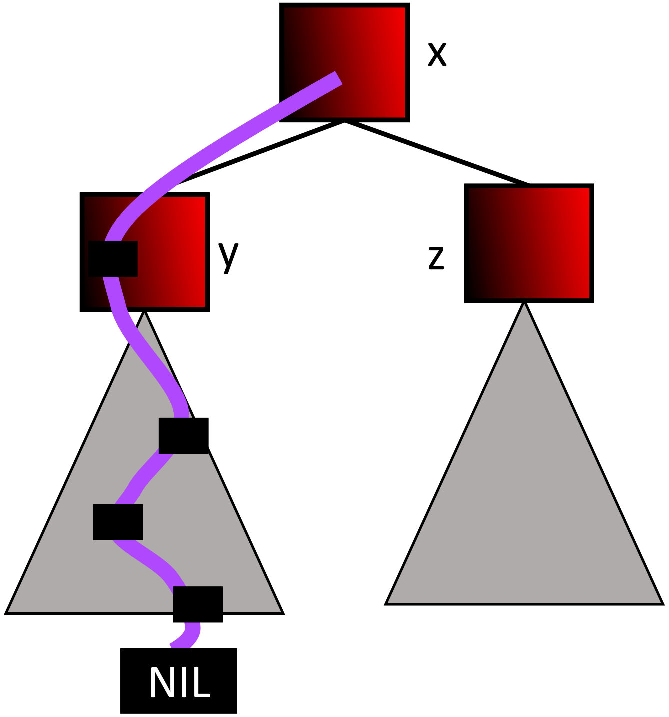
One path could be twice as long another if we pad it with red nodes.

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

That turns out to be basically right.

- **Intuition:**
  - The height of completely balanced RBTree is *floor(logn)+1*.
  - The number of black nodes from the root to any leaf is bounded to *floor(logn)+1.*
  - And, the number of red nodes from the root to any leaf is less than *floor (logn)+1*.
  - Then the longest path length is less than *2*(floor(logn)+1).*

  - Time complexity of each operation: ***O (log n)***



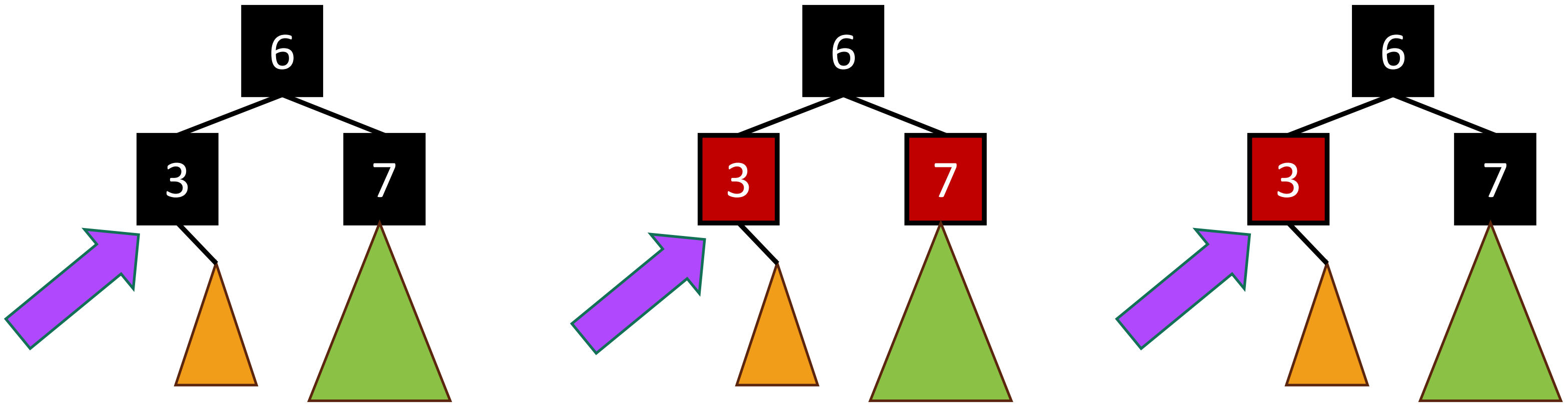
Okay, so it's balanced…
…but can we maintain it?

- **Yes!**

- For the rest of lecture:
  - sketch of how we'd do this.

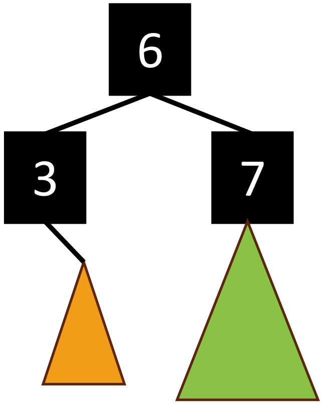
- See CLRS for more details.
- (You are not responsible for the details for this class – but you should understand the main ideas).

Many cases



- Suppose we want to insert **here**.
  - eg, want to insert 0.

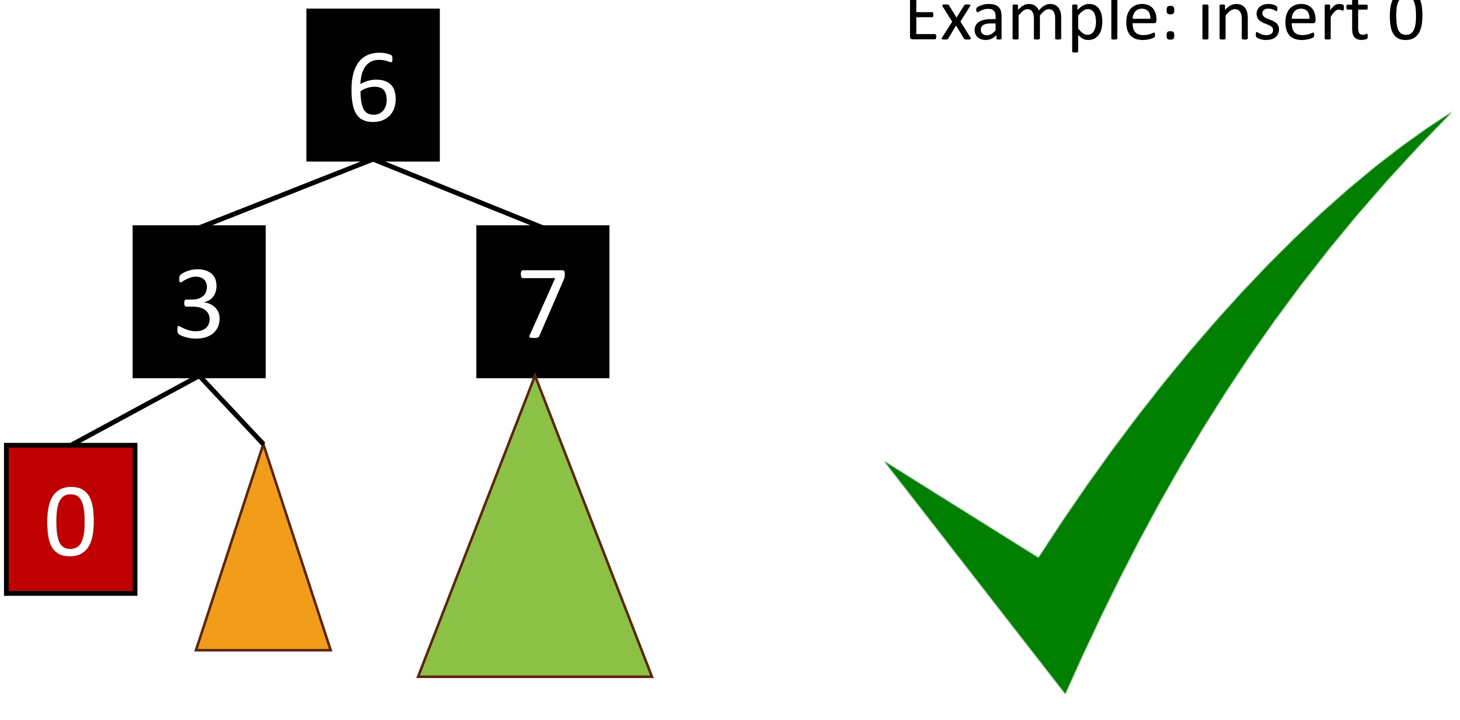
Inserting into a Red-Black Tree



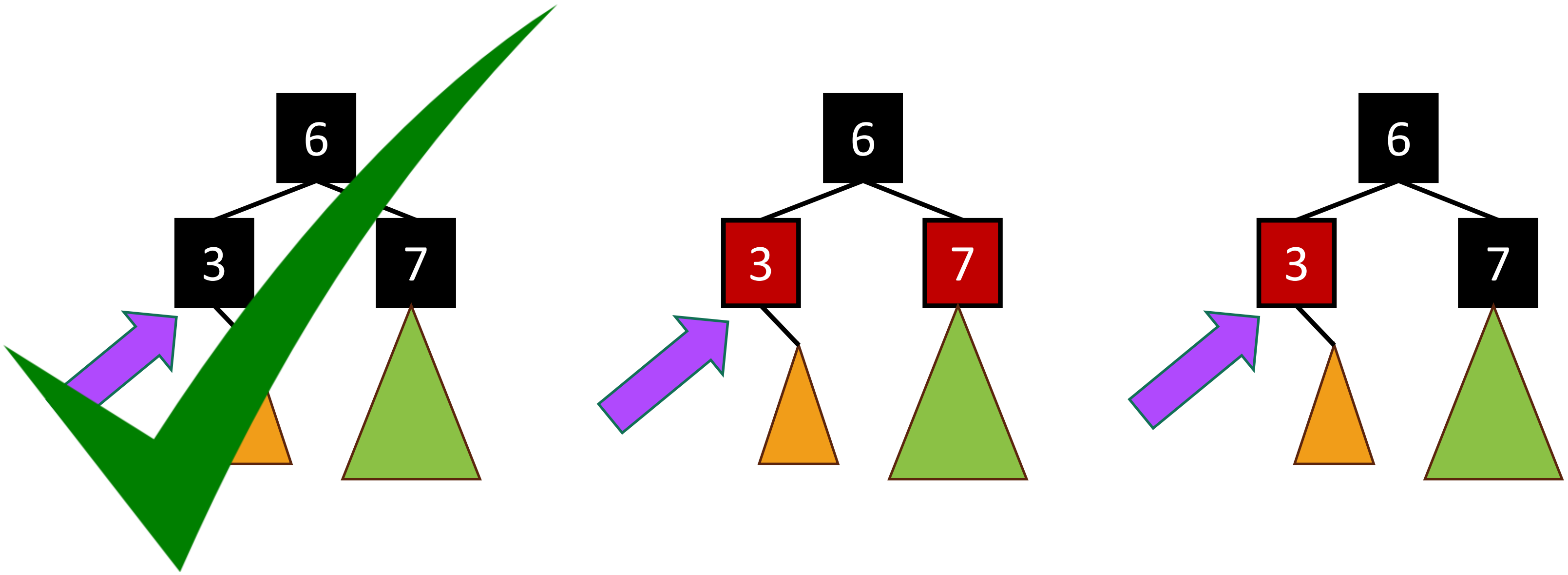
What if it looks like this?

- Make a new **red node**.
- Insert it as you would normally.

Example: insert 0

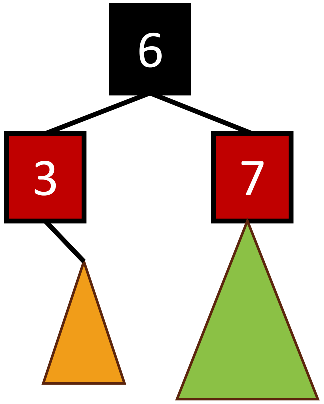


Many cases



- Suppose we want to insert **here**.
  - eg, want to insert 0.

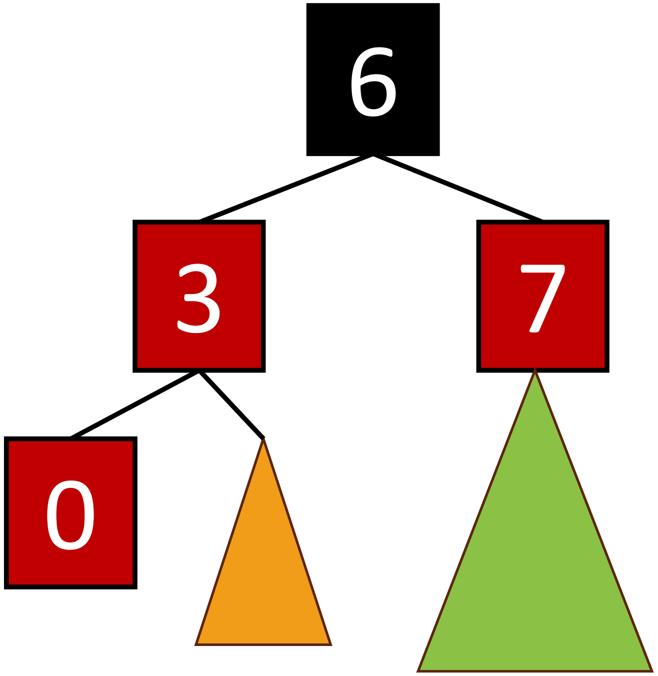
Inserting into a Red-Black Tree



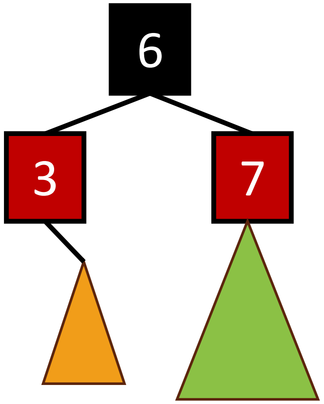
What if it looks like this?

- Make a new **red node.**
- Insert it as you would normally.
- Fix things up if needed.

Example: insert 0

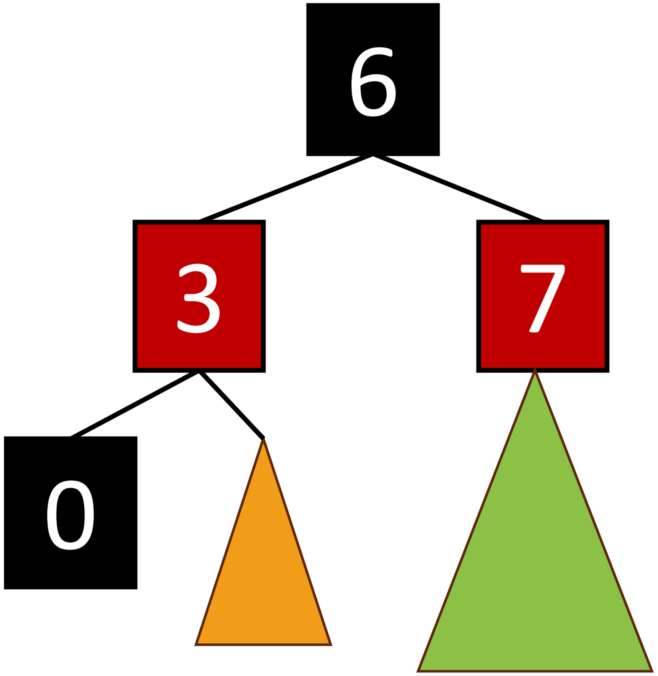


Inserting into a Red-Black Tree



What if it looks like this?

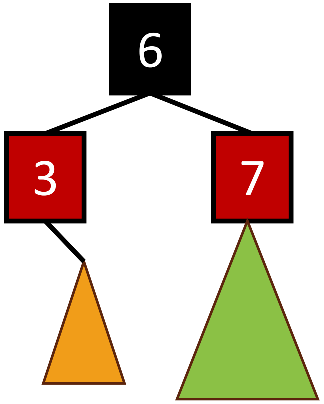
- Make a new **red node.**
- Insert it as you would normally.
- Fix things up if needed.



Example: insert 0

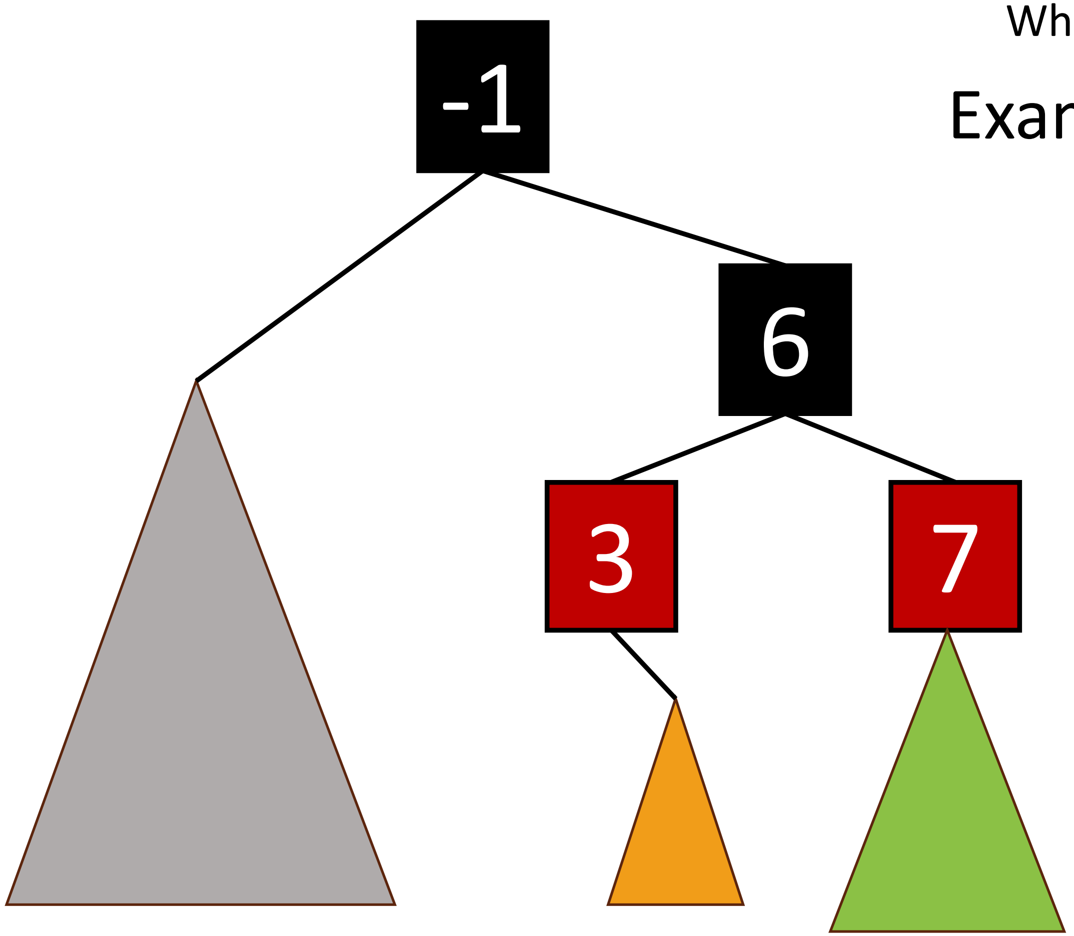
Can't we just insert 0 as a **black node?**

We need a bit more context



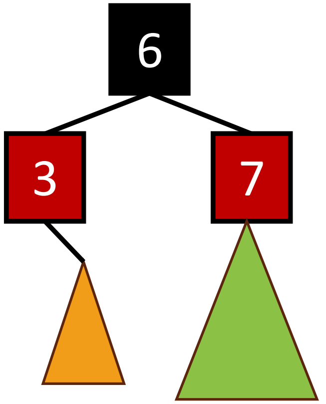
What if it looks like this?

Example: insert 0



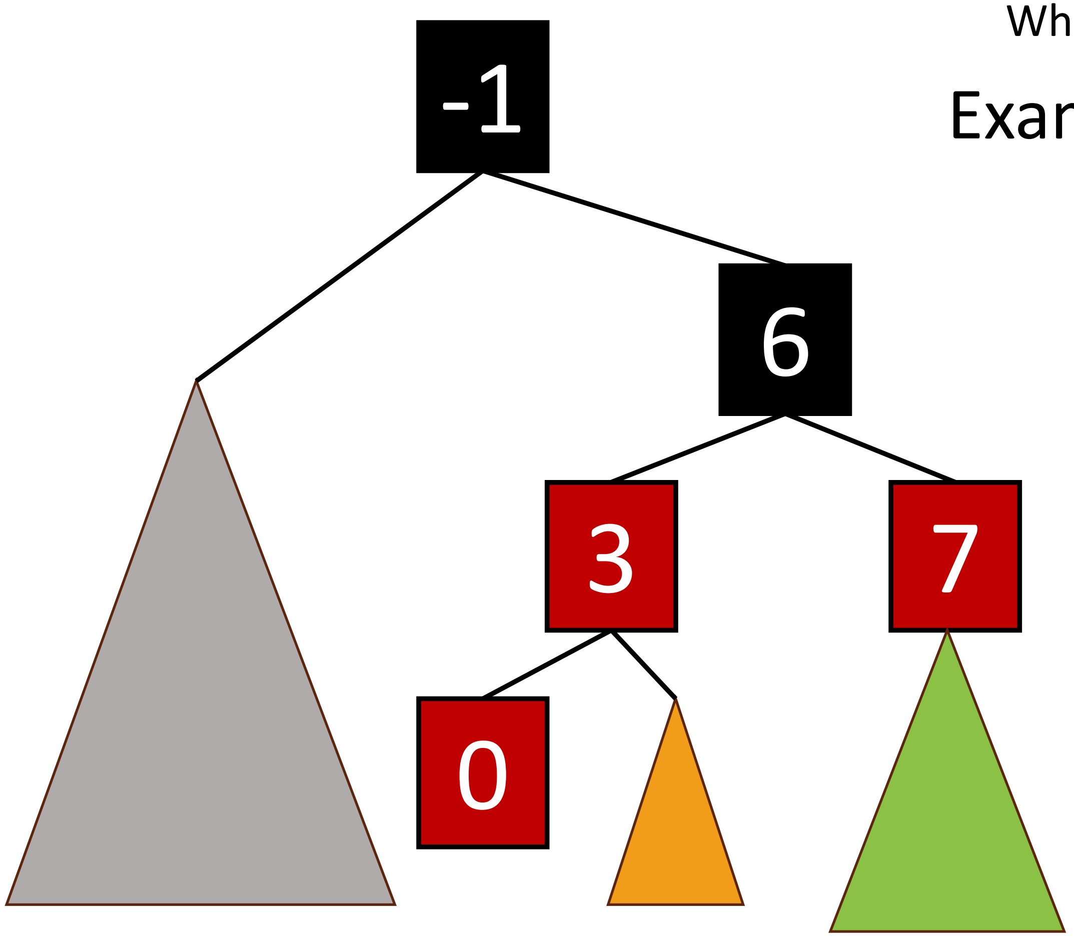
We need a bit more context

- Add 0 as a red node.



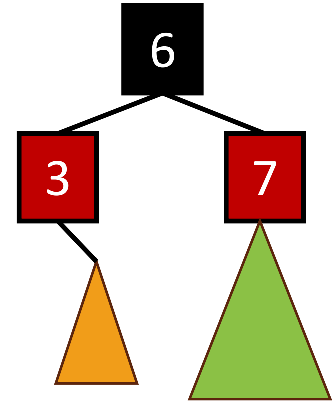
What if it looks like this?

Example: insert 0



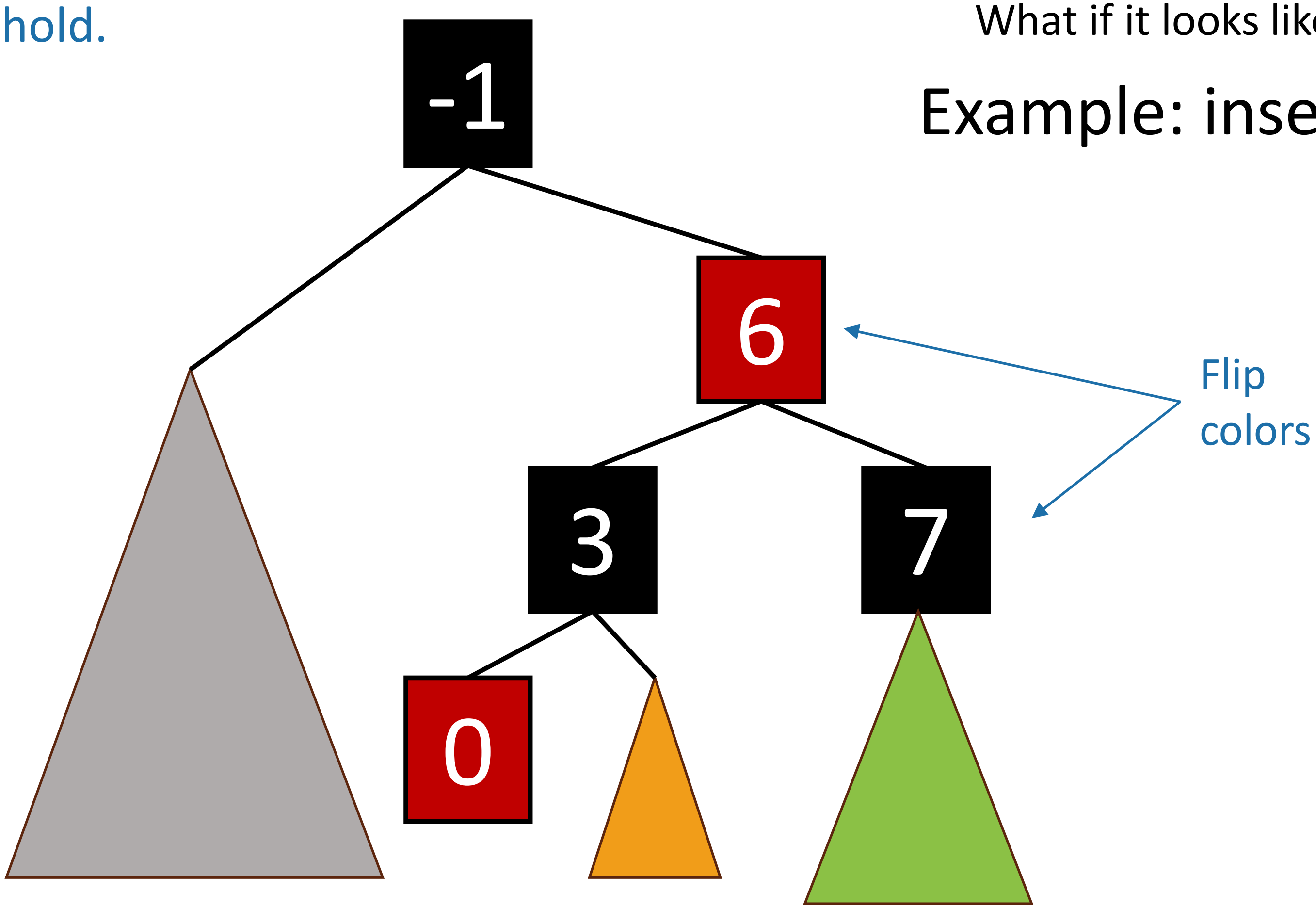
We need a bit more context

- Add 0 as a red node.
- **Claim:** RB-Tree properties still hold.

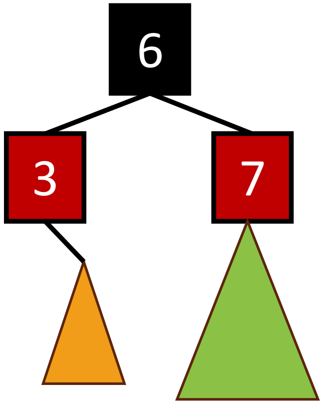


What if it looks like this?

Example: insert 0

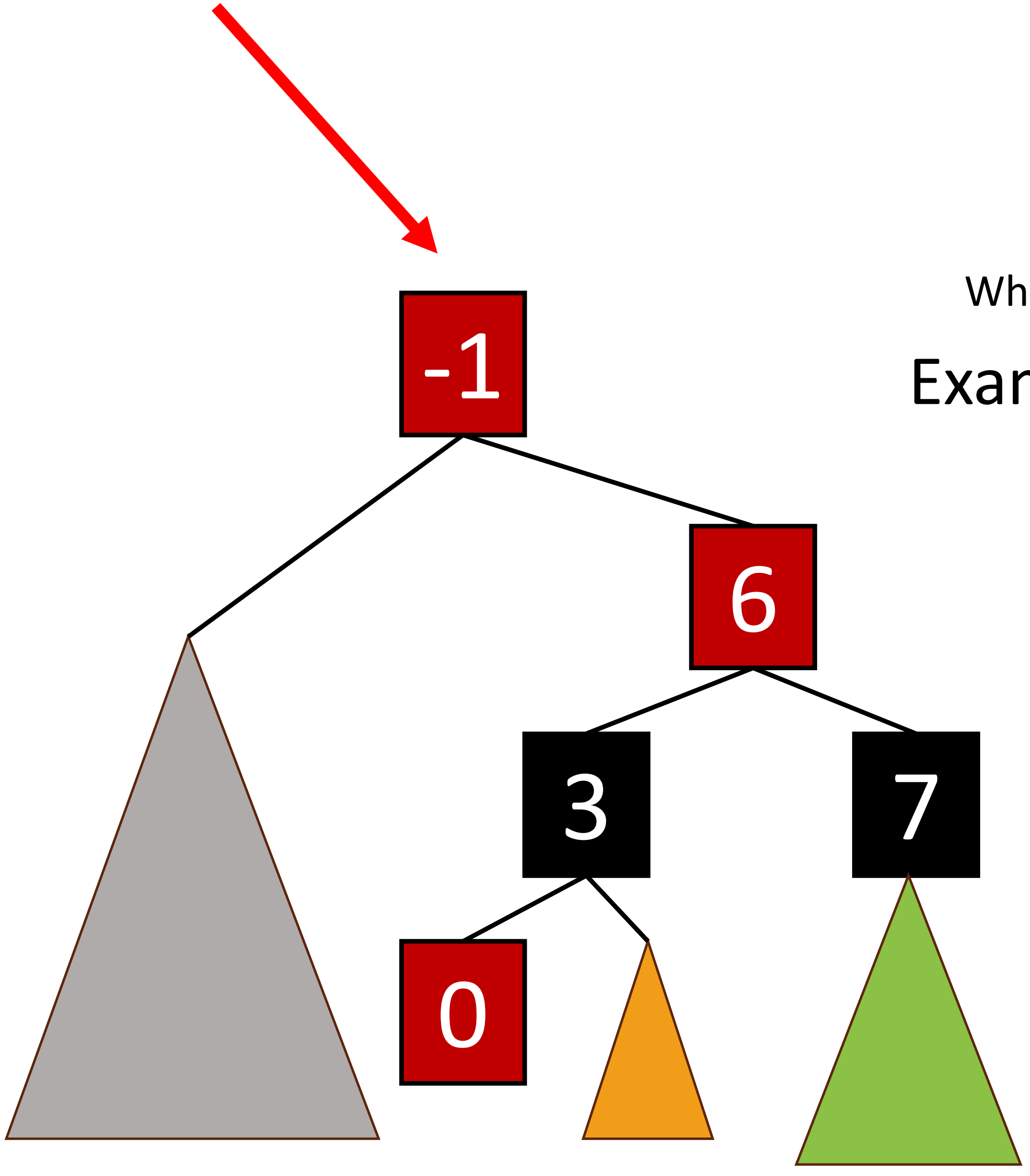


But what if that was red?

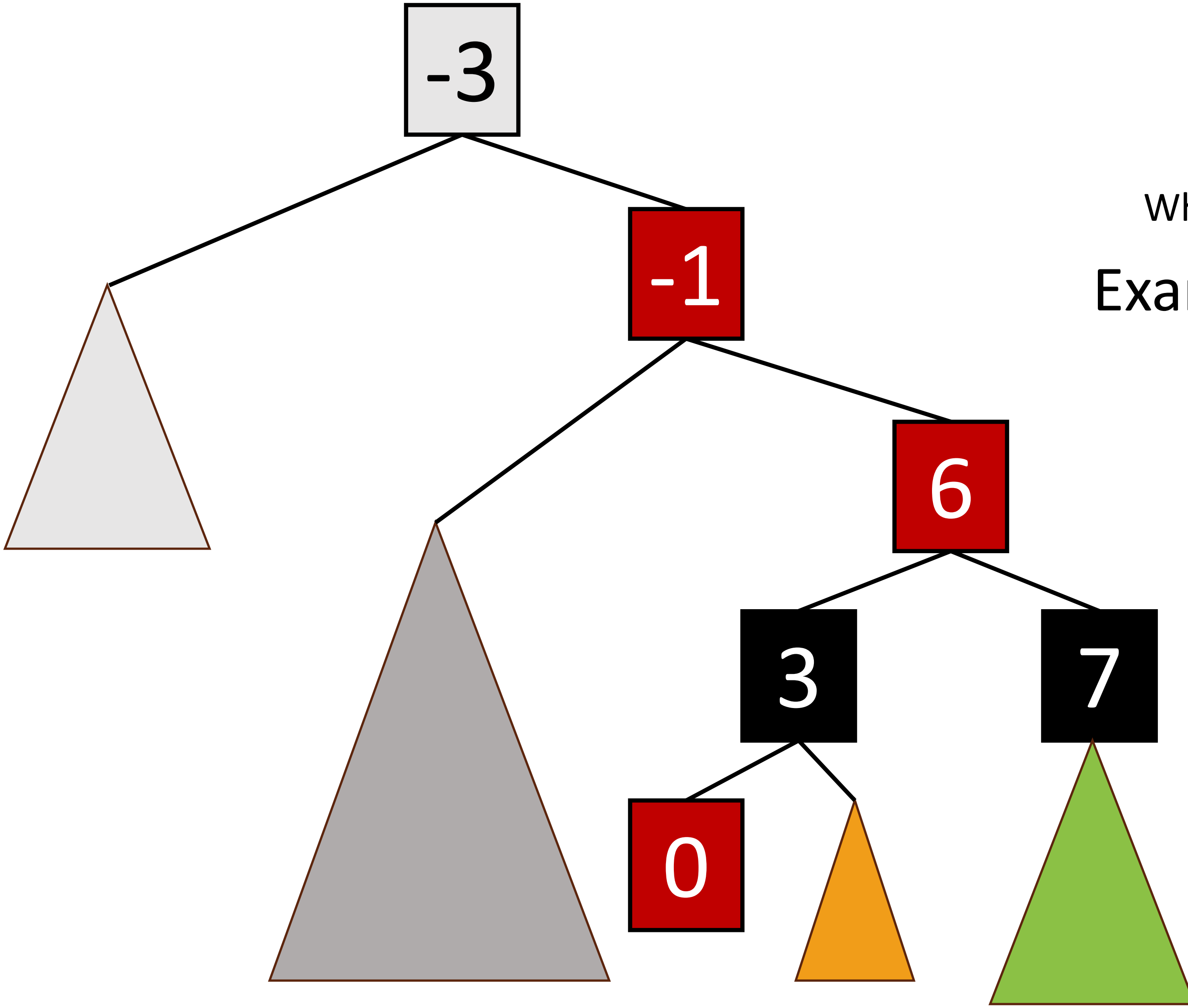


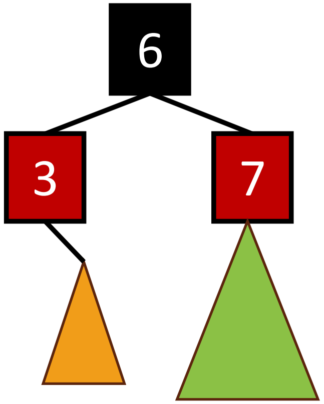
What if it looks like this?

Example: insert 0



More context…

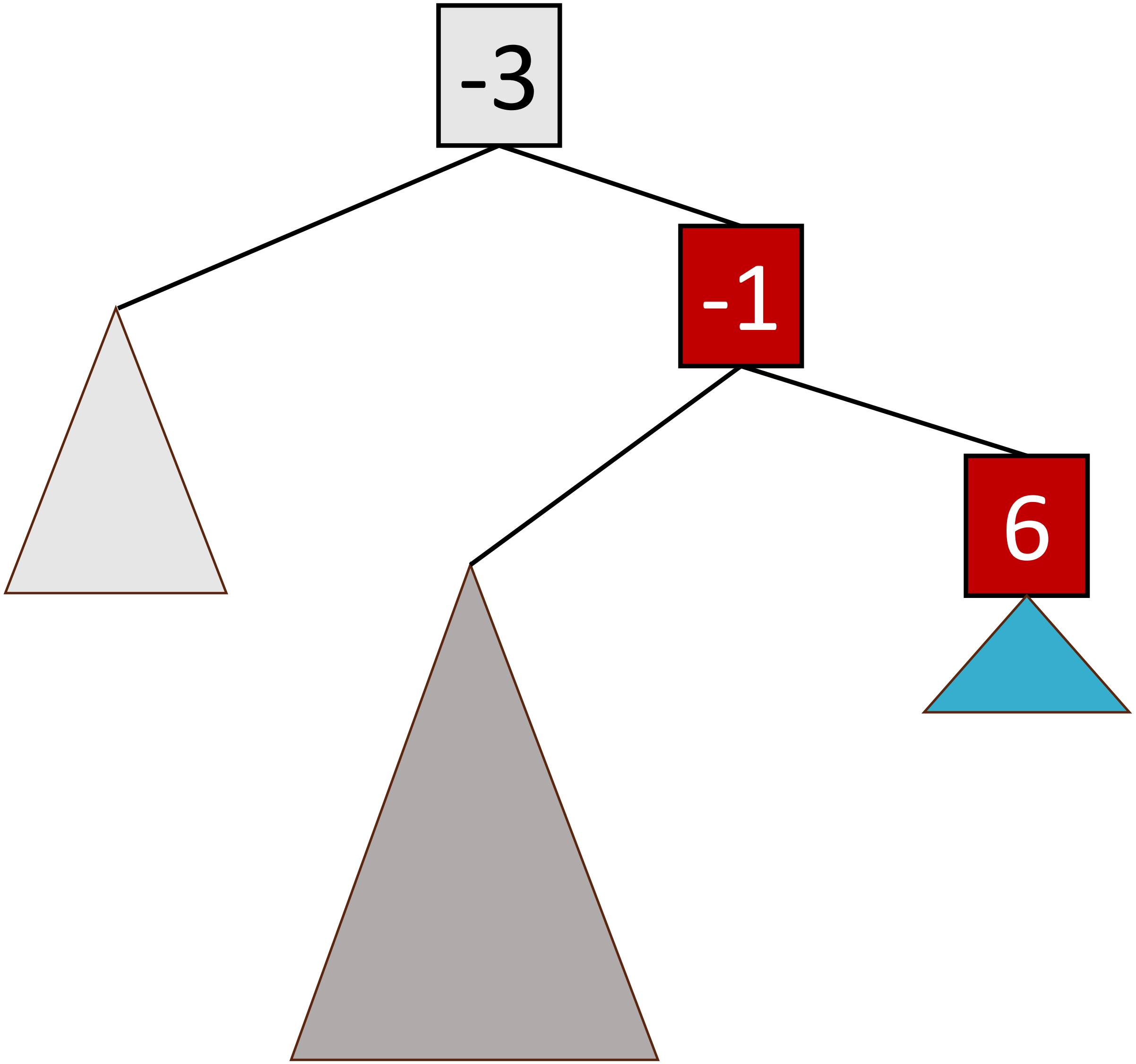


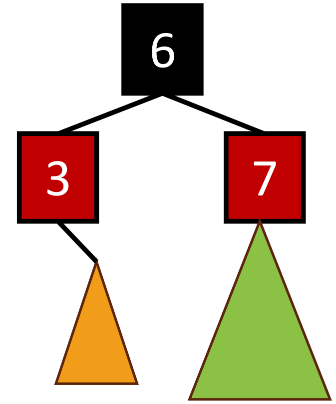


What if it looks like this?

Example: insert 0

More context…



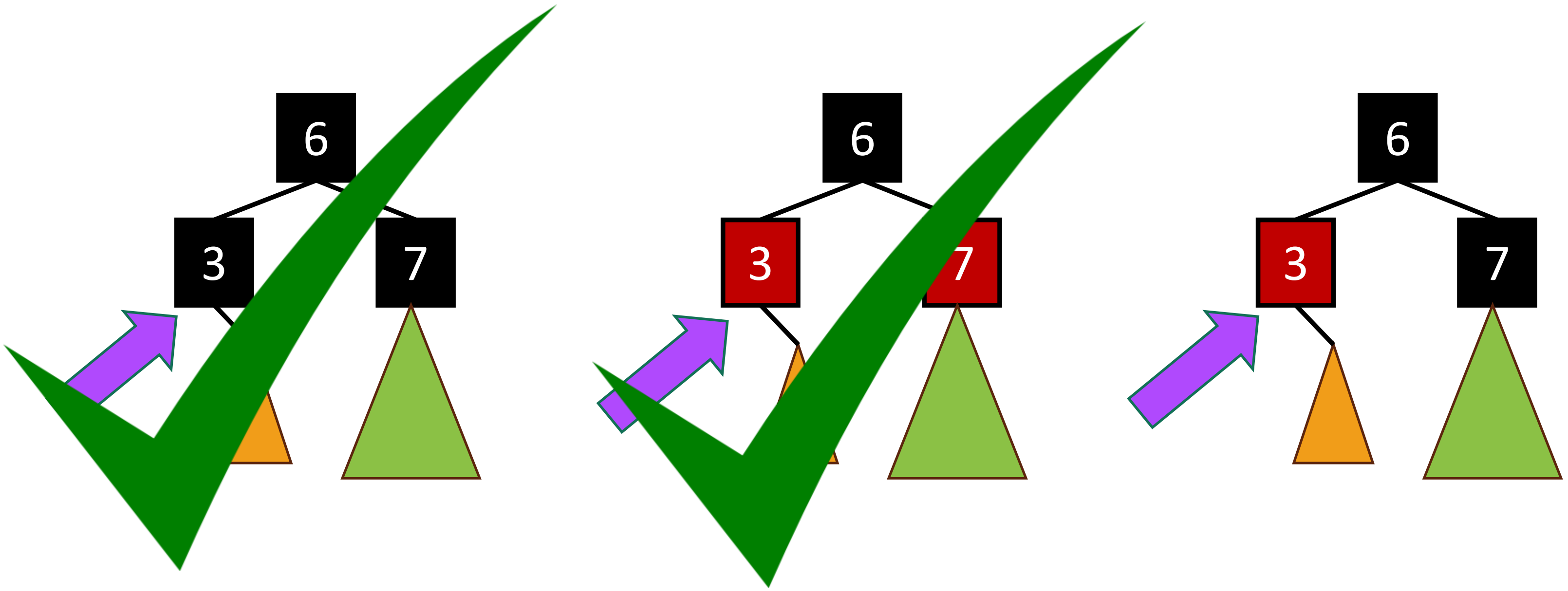


What if it looks like this?

Example: insert 0

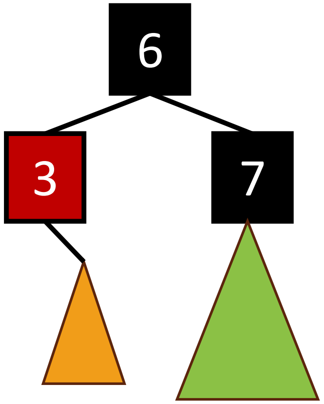
Now we're basically inserting 6 into some smaller tree. Recurse!

Many cases



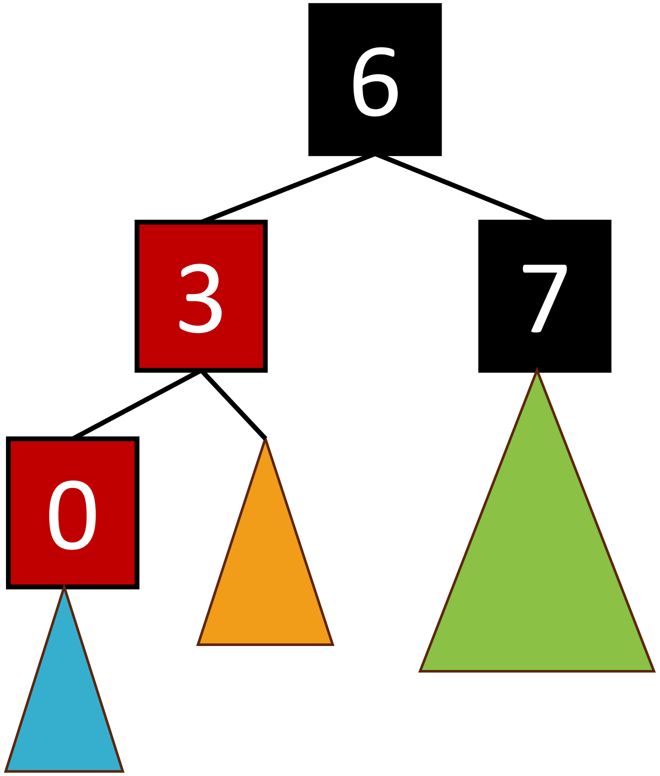
- Suppose we want to insert **here**.
    - eg, want to insert 0.

Inserting into a Red-Black Tree



What if it looks like this?

- Make a new **red node.**
- Insert it as you would normally.
- Fix things up if needed.

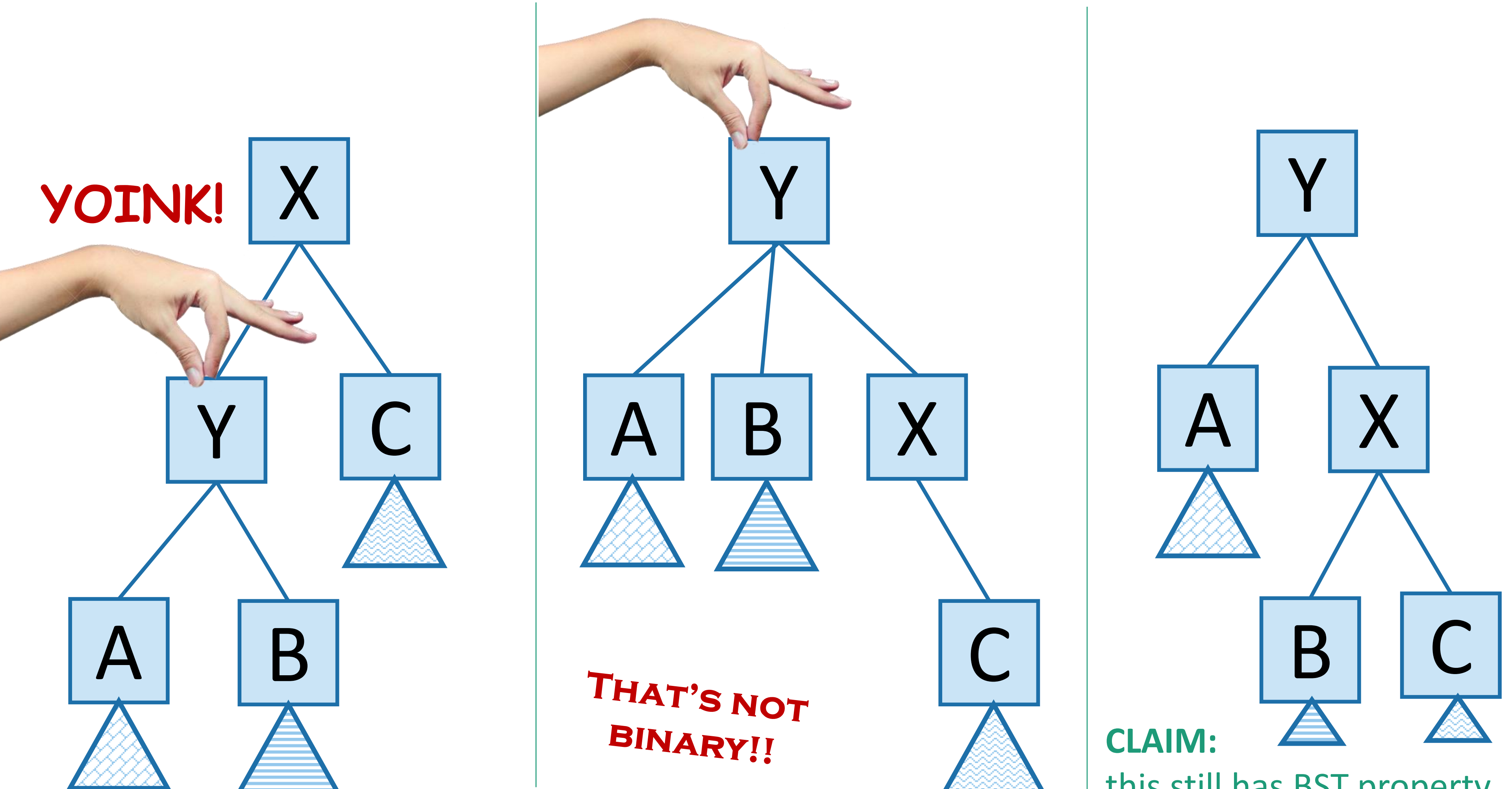


Example: Insert 0.

- Actually, **this can't happen?**
  - **6**-**3** path has one black node
  - **6**-**7**-… has at least two
- It might happen that we just turned 0 red from the previous step.
- Or it could happen if 7 is actually **NIL**.

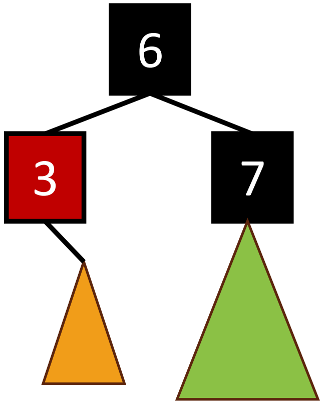
Recall Rotations

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



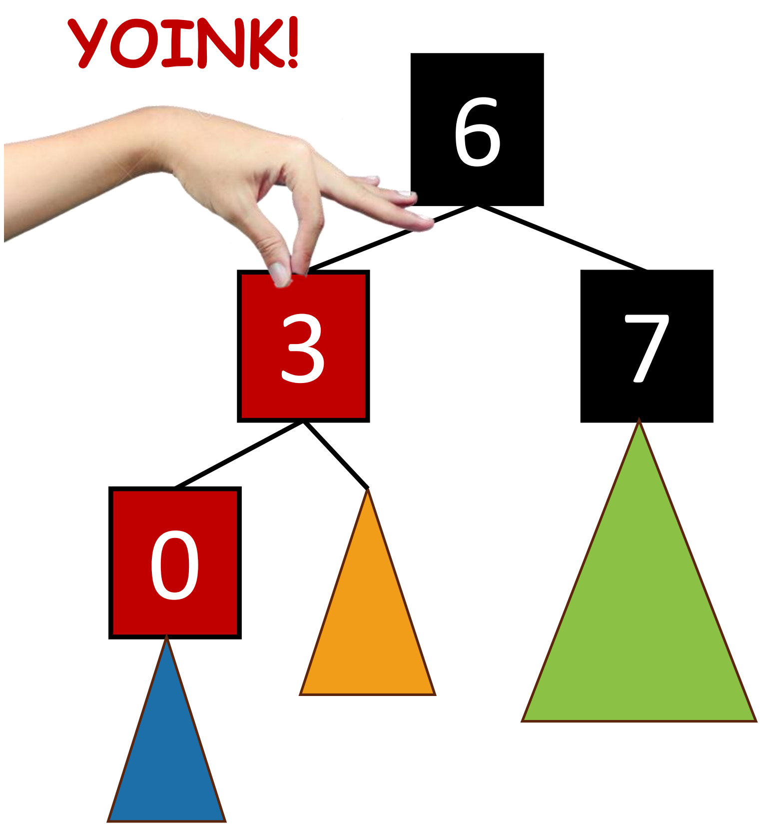
CLAIM:
this still has BST property.

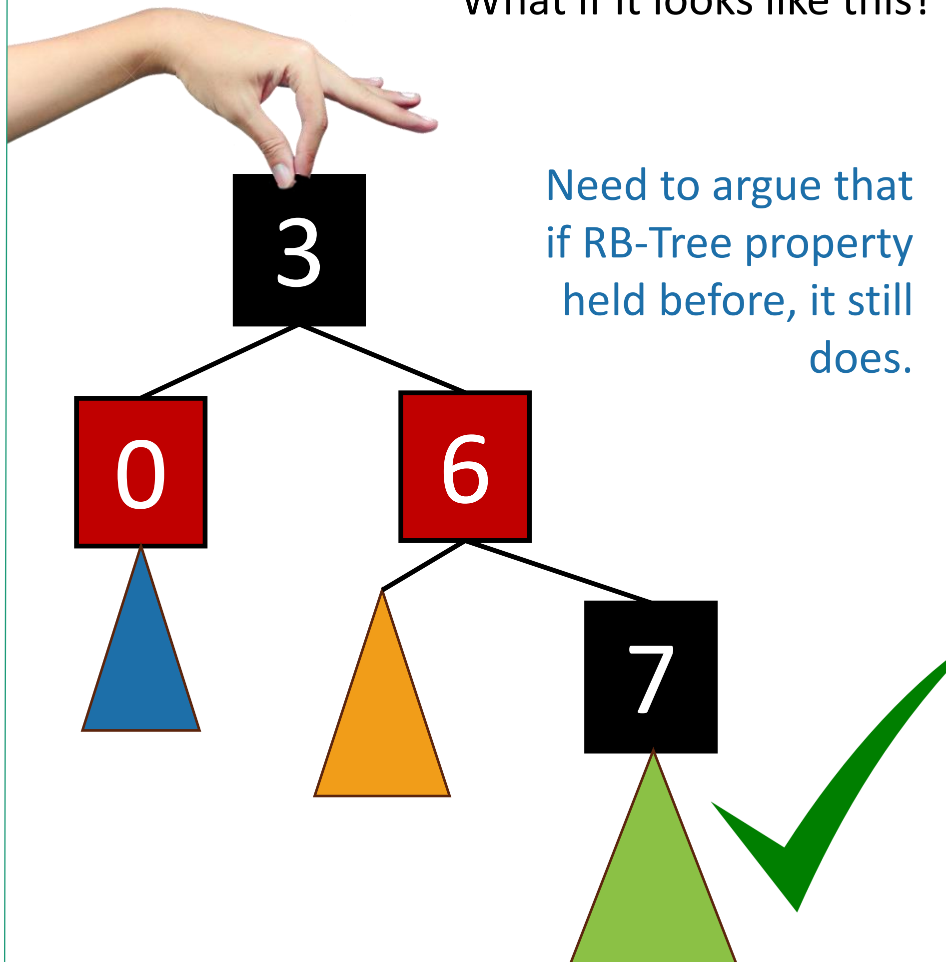
Inserting into a Red-Black Tree



- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.

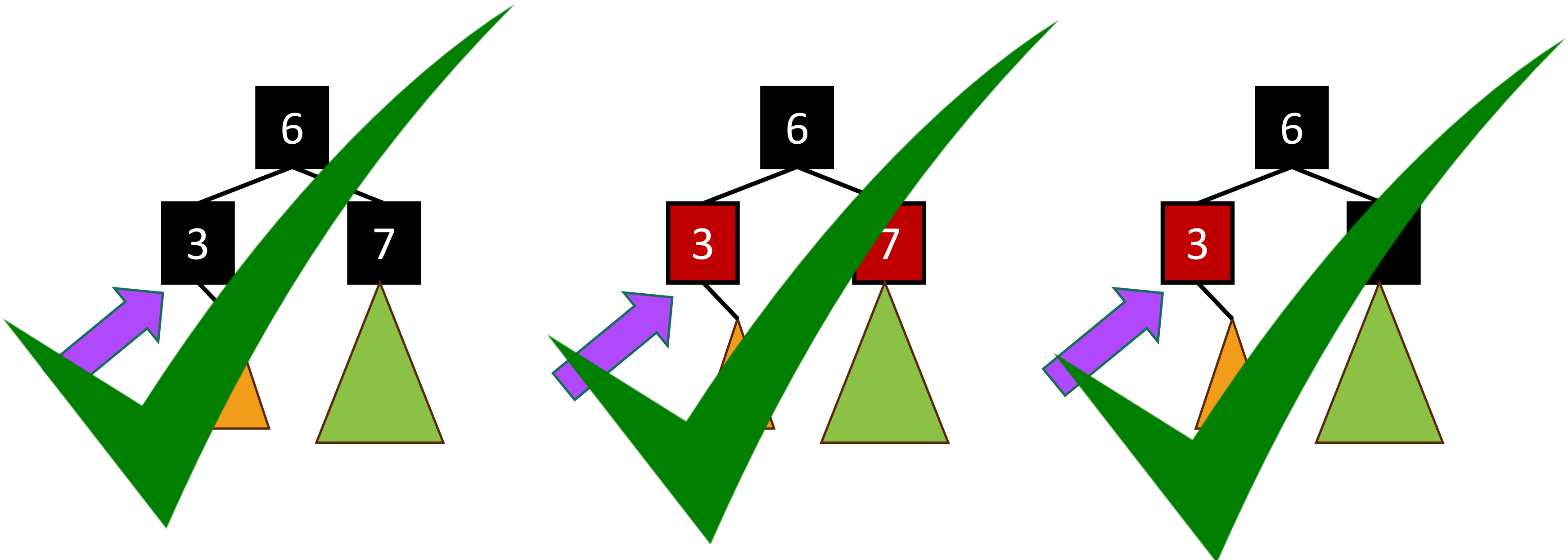
What if it looks like this?





Need to argue that if RB-Tree property held before, it still does.

Many cases



- Suppose we want to insert **here**.
    - eg, want to insert 0.

`insert` in Red-Black Trees

```
enum Color {RED, BLACK};

struct Node
{
    int data;
    bool color;
    Node *left, *right, *parent;
};

void rotateLeft(Node *&root, Node *&pt)
{
    Node *pt_right = pt->right;
    pt->right = pt_right->left;

    if (pt->right != NULL)
        pt->right->parent = pt;

    pt_right->parent = pt->parent;
    if (pt->parent == NULL)
        root = pt_right;

    else if (pt == pt->parent->left)
        pt->parent->left = pt_right;

    else
        pt->parent->right = pt_right;

    pt_right->left = pt;
    pt->parent = pt_right;
}
```

insert in Red-Black Trees

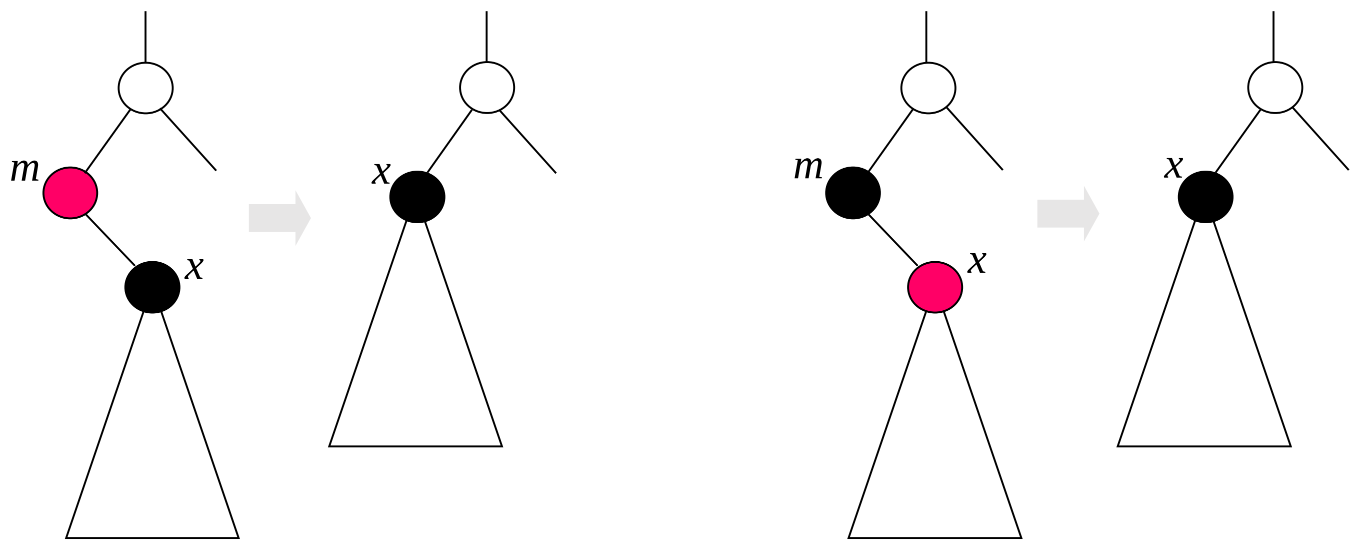
```
// This function fixes violations caused by BST insertion
// It only contains a specific case. Other cases must be implemented!
void RBTree::fixViolation(Node *&root, Node *&pt)
{
    Node *parent_pt = NULL;
    Node *grand_parent_pt = NULL;

    while ((pt != root) && (pt->color != BLACK) &&
           (pt->parent->color == RED))
    {
        parent_pt = pt->parent;
        grand_parent_pt = pt->parent->parent;

        /*  Case : 3
            Parent of pt is left child of Grand-parent of pt */
        if (parent_pt == grand_parent_pt->left)
        {
            Node *uncle_pt = grand_parent_pt->right;
            if (uncle_pt != NULL && uncle_pt->color == BLACK)
            {
                if (pt == parent_pt->left)
                {
                rotateRight(root, grand_parent_pt);
                swap(parent_pt->color, grand_parent_pt->color);
                pt = parent_pt;
                }
            }
        }
    }
    root->color = BLACK;
}
```

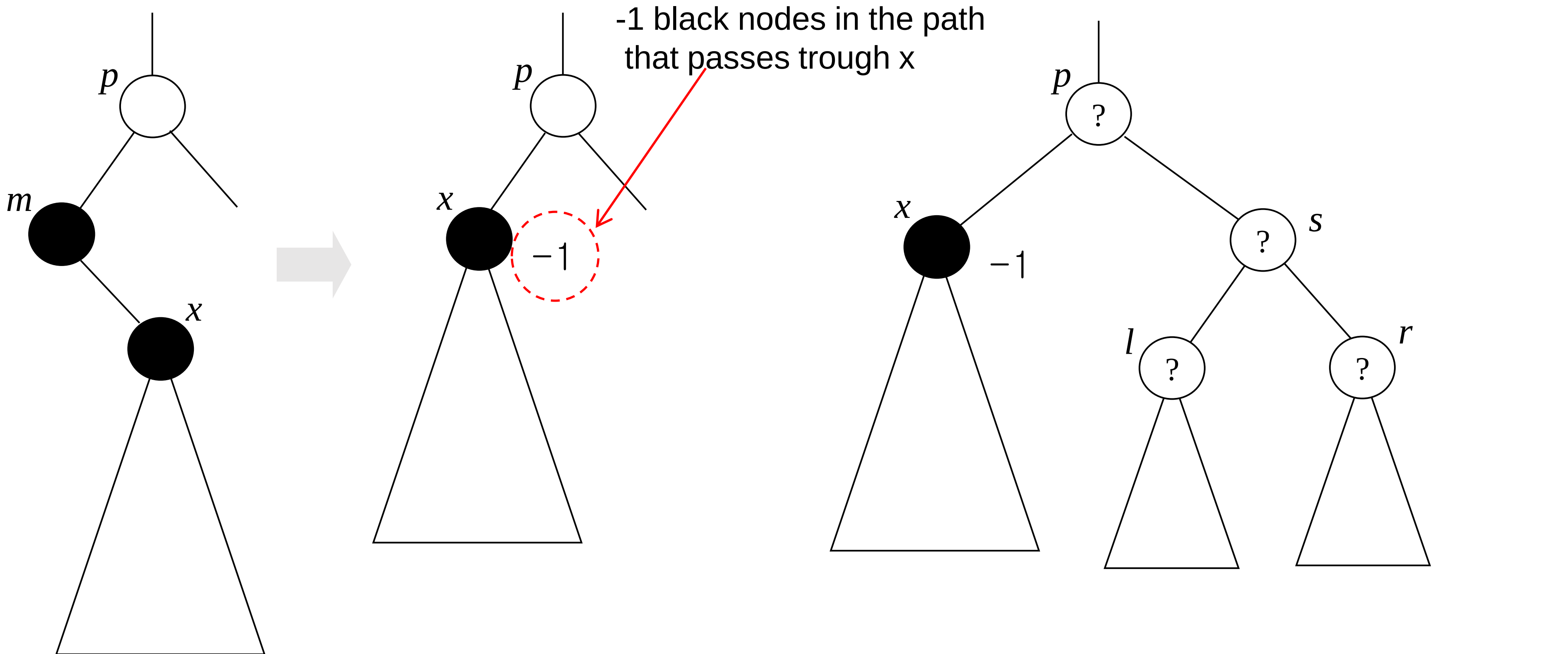
Deleting from a Red-Black tree

- Some easy cases
  - If deleting node is RED
  - If deleting node is BLACK, and its unique child is RED



Deleting from a Red-Black tree

- Little-bit complicated
  - If deleting node is BLACK
    - In this case, we have address each case that (p,s,l,r) has distinct color combination (skip in this class)



That’s a lot of cases

- You are **not responsible** for the 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?

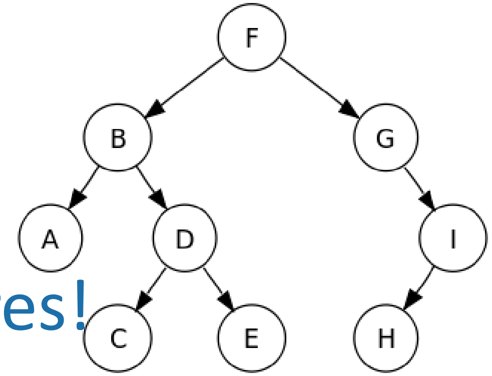
- Red-Black Trees always have height at most 2log(n+1).
- As with general Binary Search Trees, all operations are O(height)
- So all operations are O(log(n)).

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))

Today

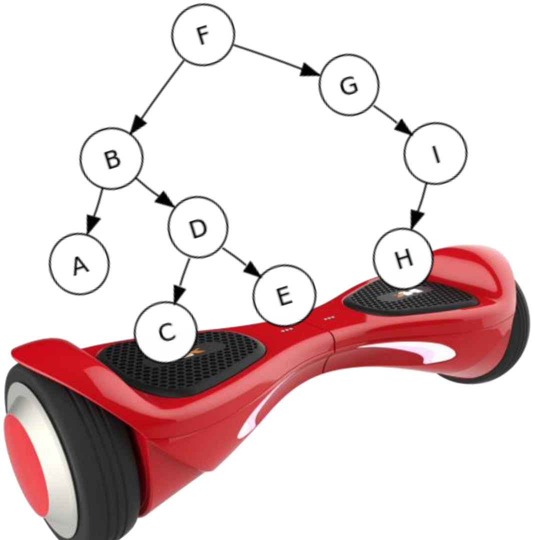
- Begin a brief summary into data structures!



- Binary search trees
  - They are better when they're balanced.

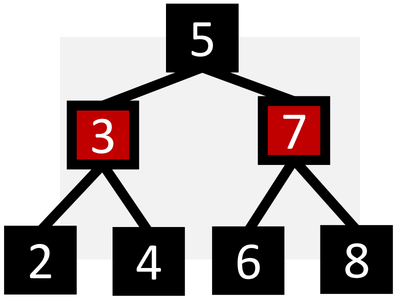
this will lead us to…

- Self-Balancing Binary Search Trees
  - **Red-Black** trees.



Recap

- Balanced binary trees are the best of both worlds!
- But we need to keep them balanced.
- **Red-Black Trees** do that for us.
  - We get O(log(n))-time INSERT/DELETE/SEARCH
  - Clever idea: have a proxy for balance



Post any question
On the Q&A board.