

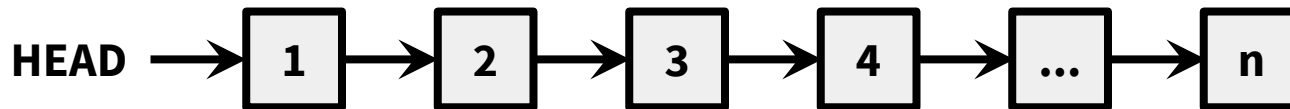
# Binary Search Trees

## Red-Black Trees

# Binary Search Trees

# Why BSTs?

Good data structures help us to design more efficient algorithms!



**Sorted linked lists:**  $O(n)$  search/select  
 $O(1)$  insert/delete

Assuming we already have a  
pointer to the location of the  
insert/delete



**Sorted arrays:**  $O(\log n)$  search  
 $O(1)$  select

“Get the  $k^{\text{th}}$  smallest element”

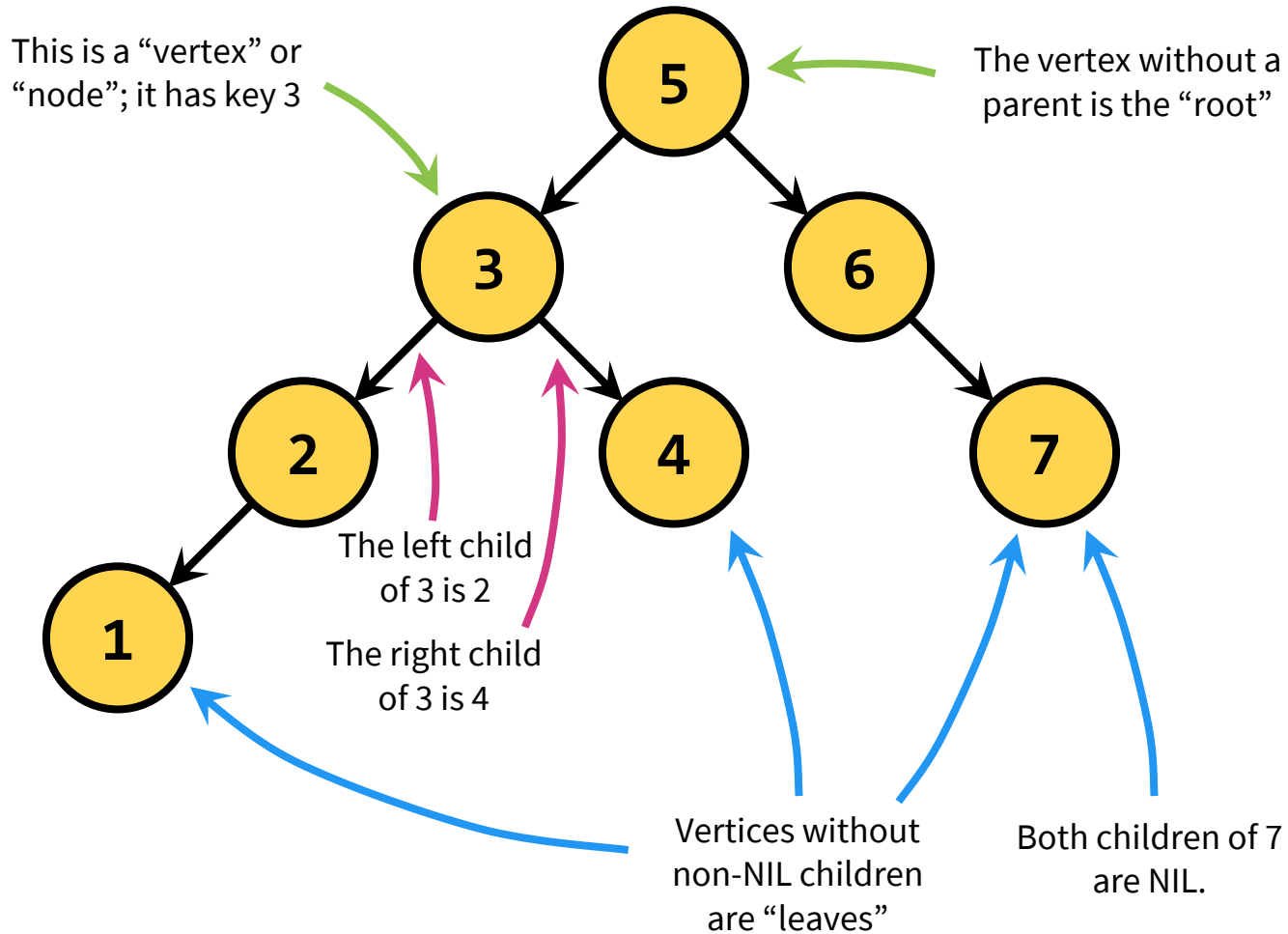
$O(n)$  insert/delete



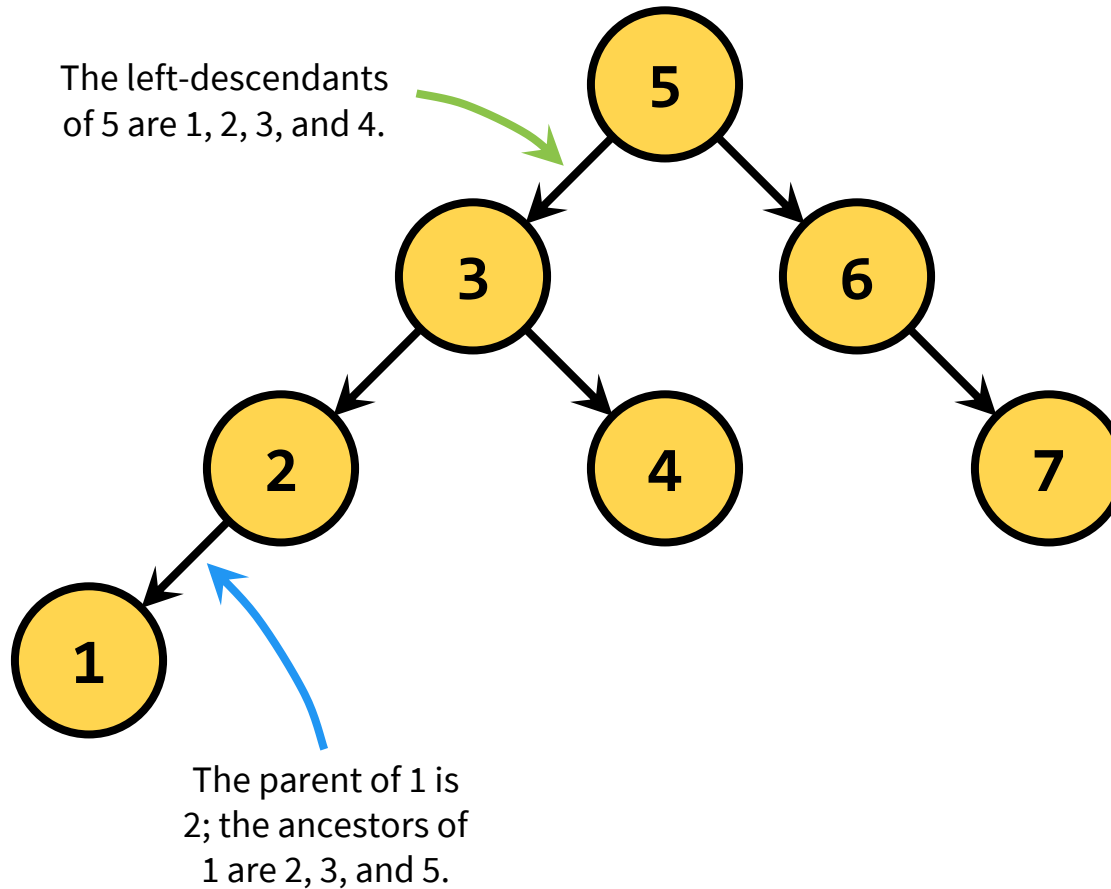
# Why BSTs?

	Sorted linked lists	Sorted arrays	Binary search trees
Search	$O(n)$	$O(\log n)$	$O(\log n)$
Insert/Delete	$O(n)$ If we already find the place to insert or delete, then $O(1)$	$O(n)$	$O(\log n)$

# Tree Terminology



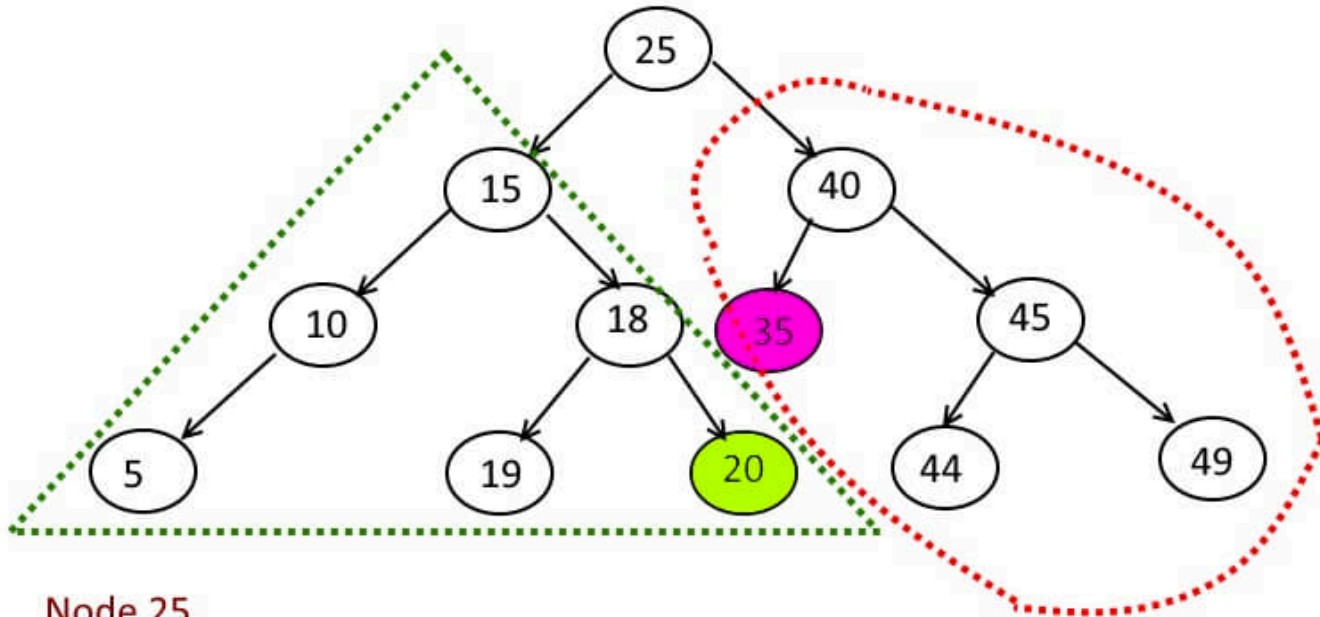
# Tree Terminology



# Tree Terminology

The **Predecessor** of a node is the **right most** element in its **left subtree**.

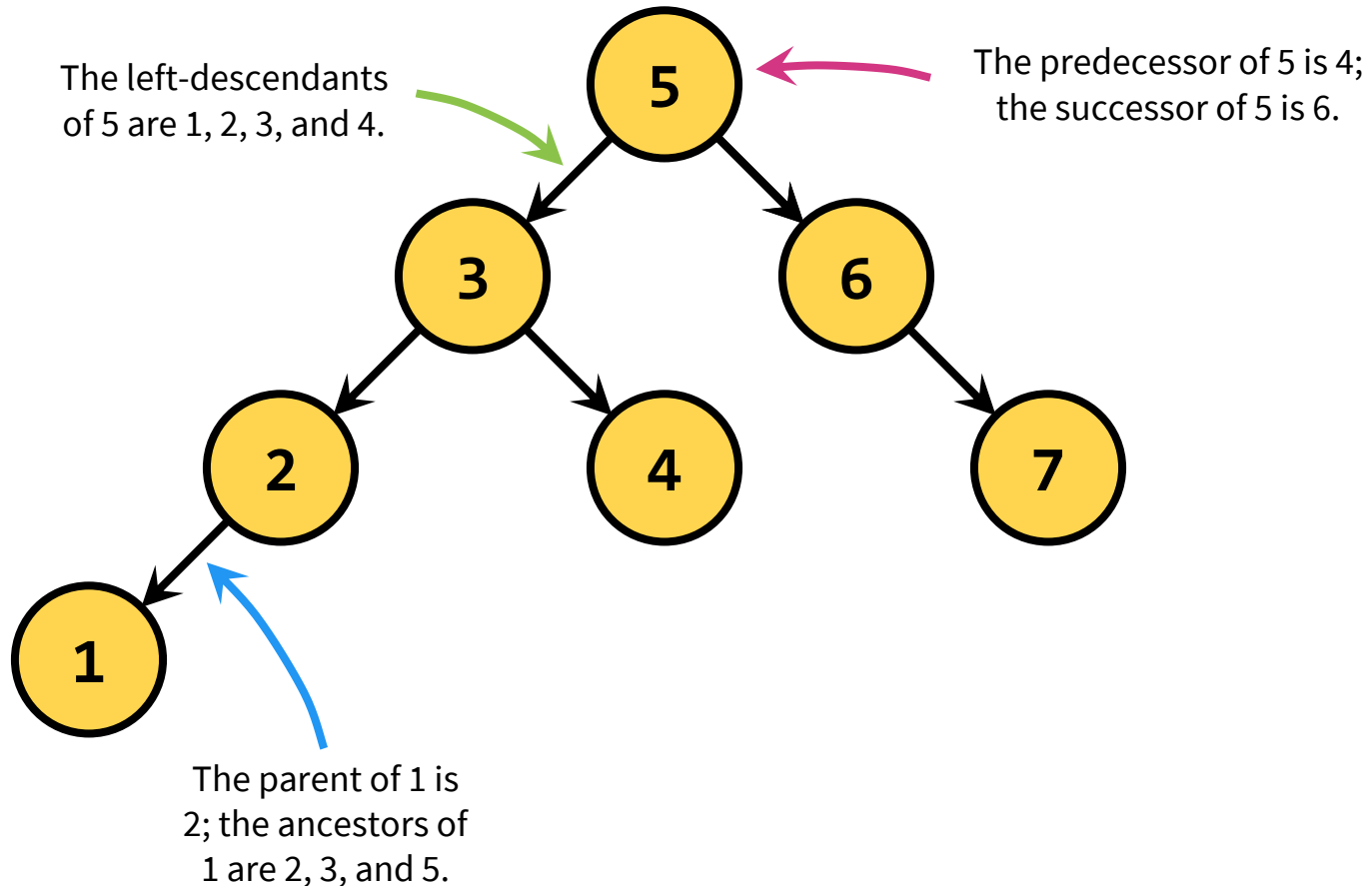
The **Successor** of a node is the **left most** element in its **right subtree**.



The predecessor of node 25 is the right most node in its left subtree, which is 20

The successor of node 25 is the left most node in its right subtree, which is 35

# Tree Terminology





# Binary Search Trees

Binary Trees are trees such that each vertex has at most 2 children.

Binary Search Trees are Binary Trees such that:

**Every** left descendent of a vertex has key smaller than that vertex.

**Every** right descendent of a vertex has key greater than that vertex.

Take care: Not only the left and right child, but all the left and right descendants!

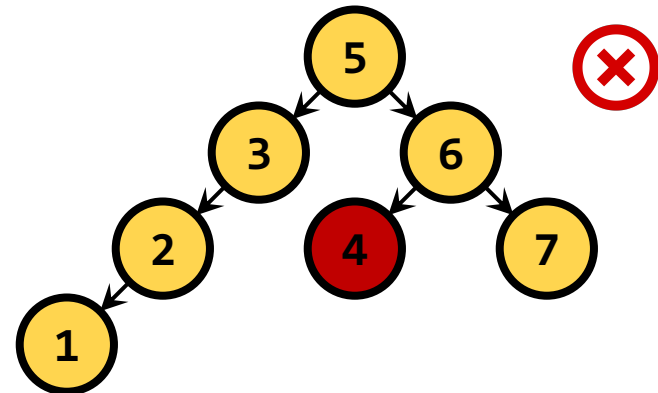
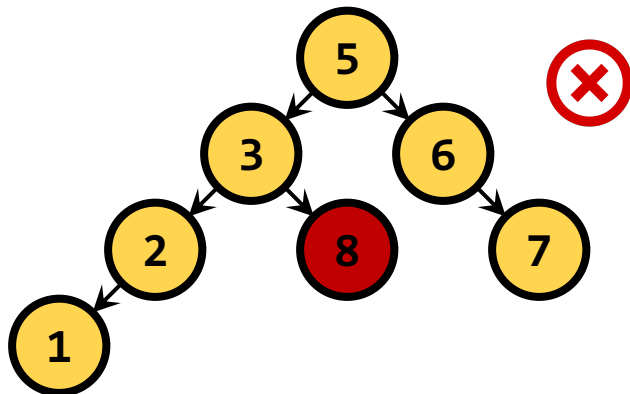
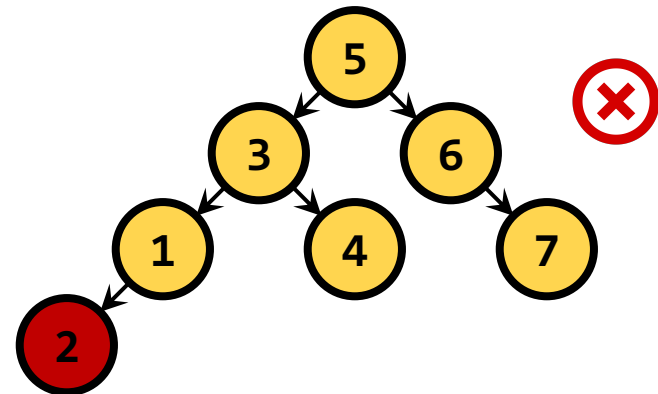
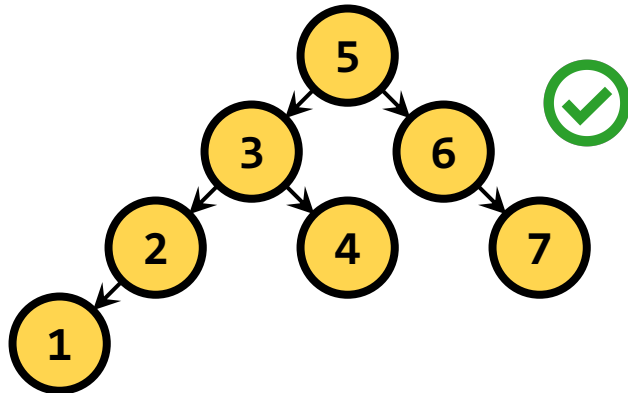
# Binary Search Trees

Binary Search Trees are Binary Trees such that:

**Every** left descendent of a vertex has key **smaller than that vertex**.

**Every** right descendent of a vertex has key **greater than that vertex**.

Take care: Not only the left- and right-child, but all the left and right descendants!



# Building BSTs by Example

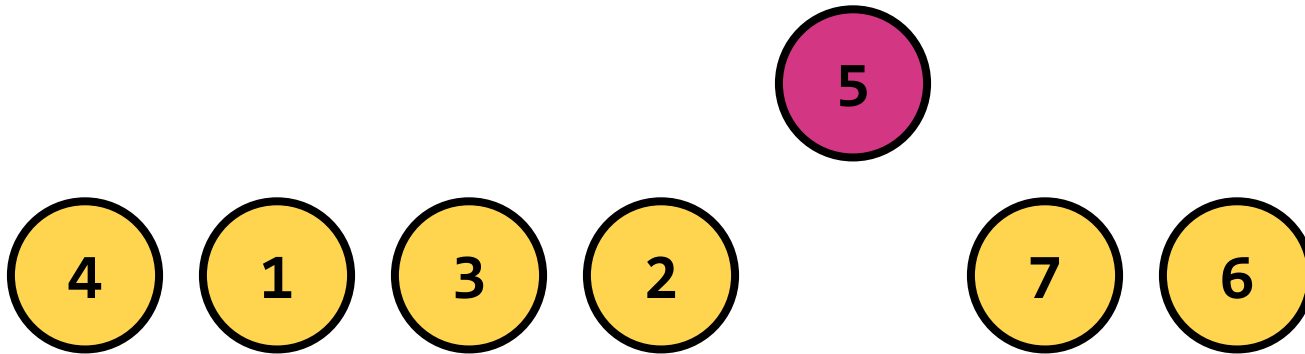


# Building BSTs by Example



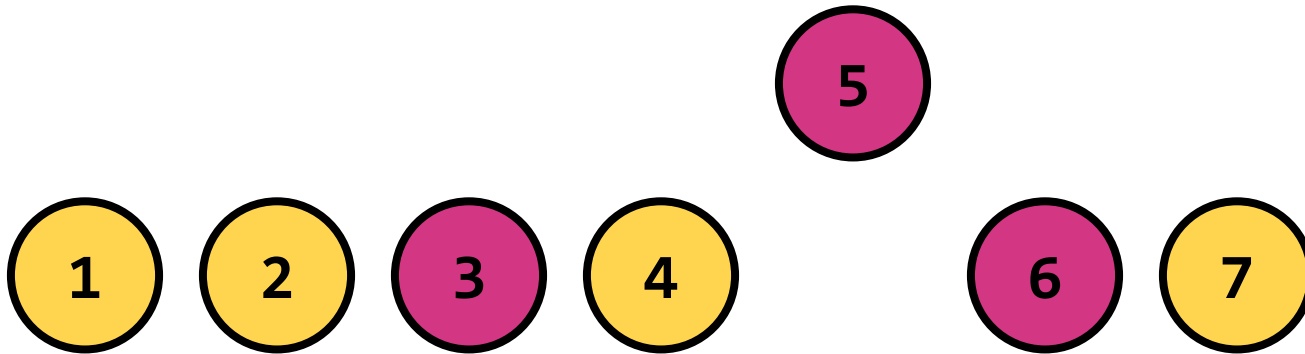
Let's partition about one of the vertices ...

# Building BSTs by Example



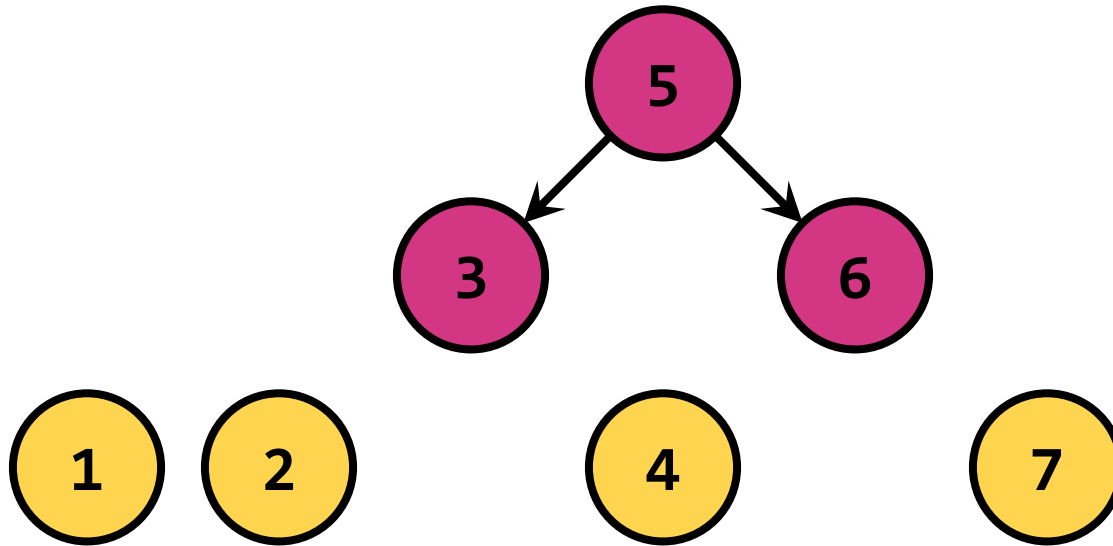
... and build a tree with that vertex as the root.

# Building BSTs by Example



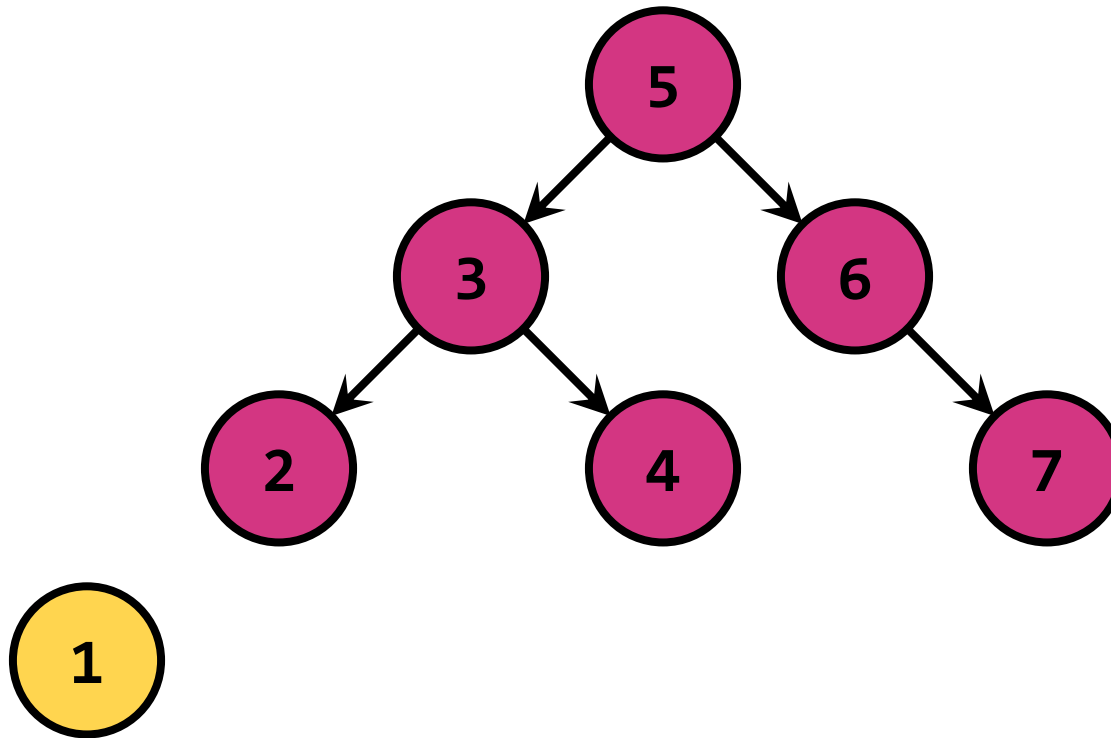
Then, recursively build trees out of its descendants.

# Building BSTs by Example



Then, recursively build trees out of its descendants.

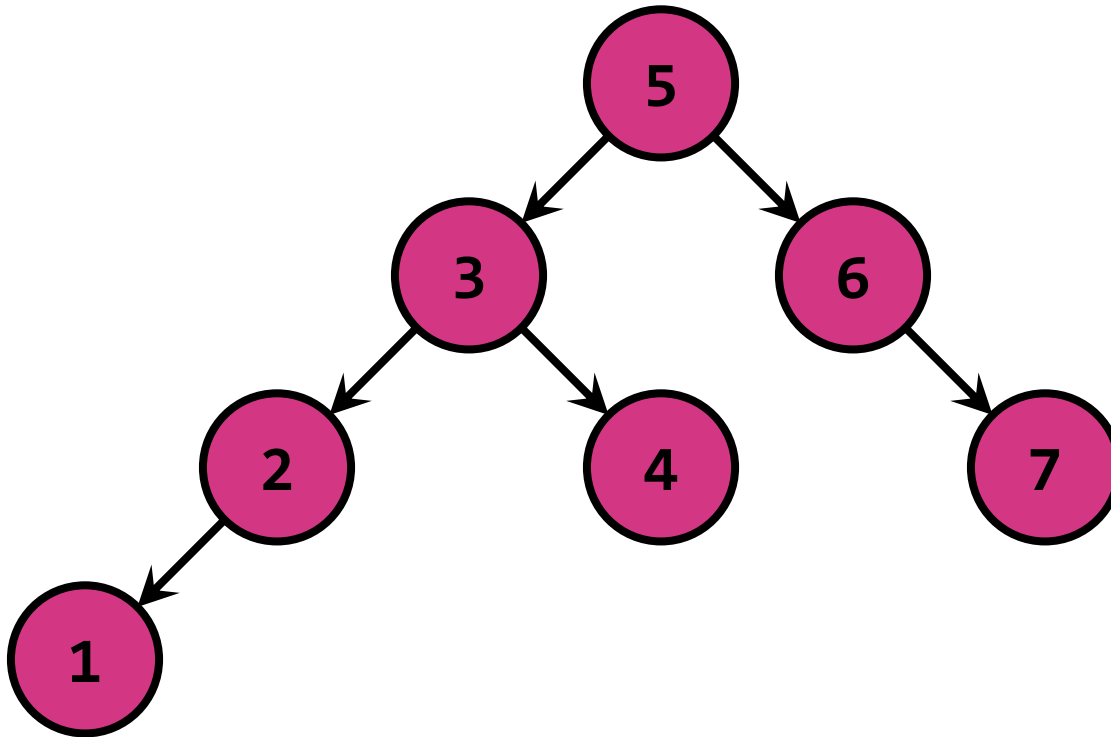
# Building BSTs by Example



Then, recursively build trees out of its descendants.



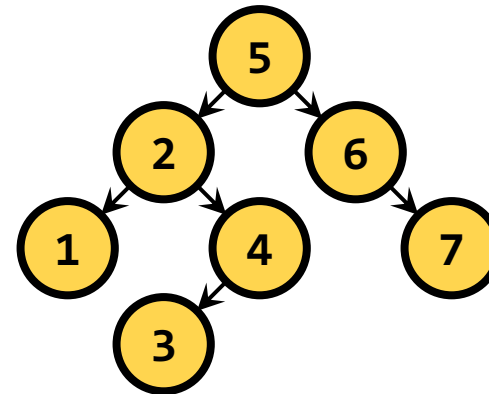
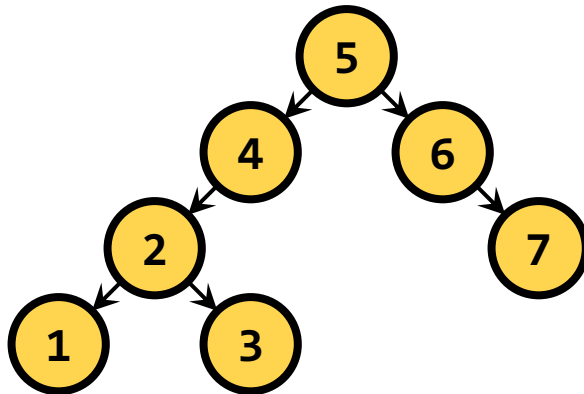
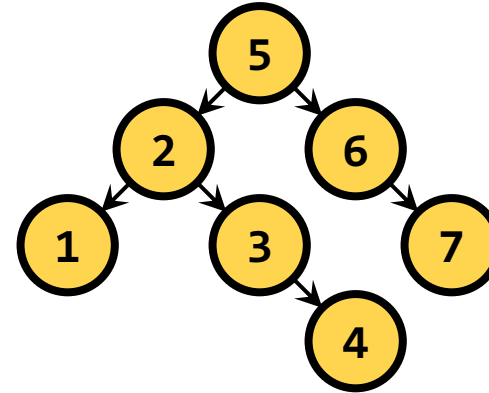
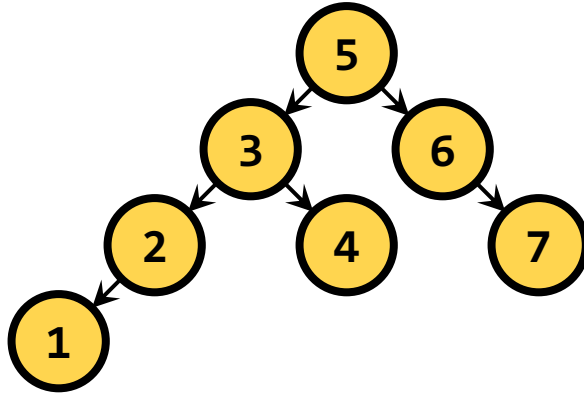
# Building BSTs by Example



Then, recursively build trees out of its descendants.

# There Exist Many Valid BSTs

Explanation on board: construction of another BST by selecting 4 as the root.



... and many more.

How many?



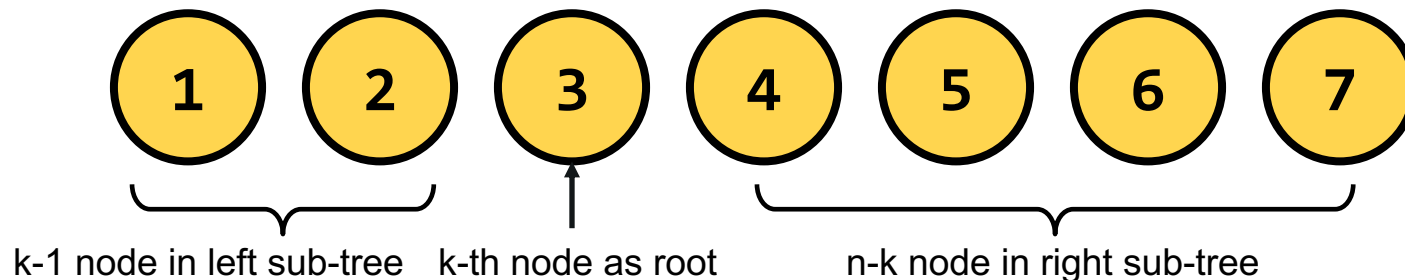
Catalan number:

$$C(n) = \frac{(2n)!}{n!(n+1)!}$$

# There Exist Many Valid BSTs

Given  $n$  vertices, how many valid BSTs can we possibly build?

Let  $C(n)$  be the number of valid BSTs using  $n$  nodes.



$$C(0) = C(1) = 1$$

Rewrite  $C(n)$  by construction:  $C(n) = \sum_{k=1}^n C(k-1)C(n-k)$

$C(n)$  is a series of numbers defined by the following recurrence relation:

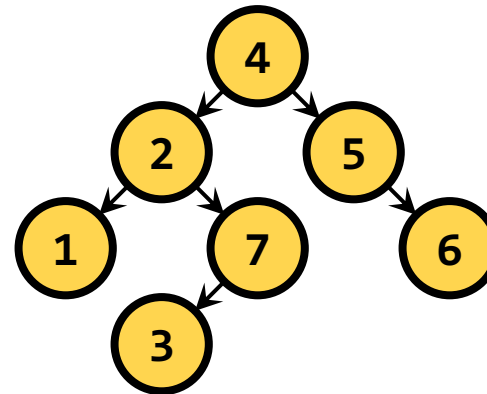
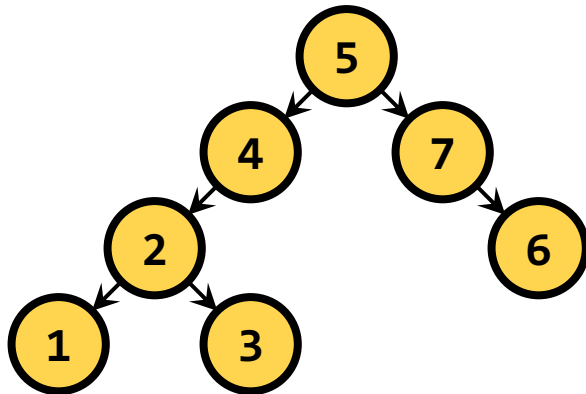
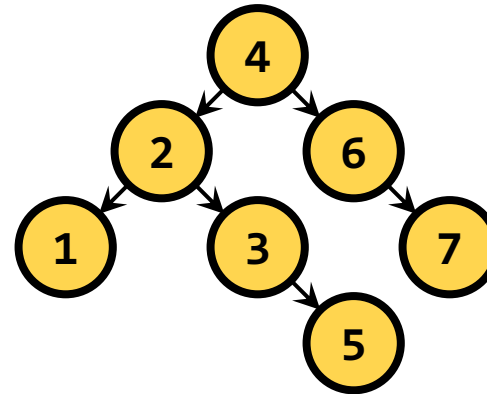
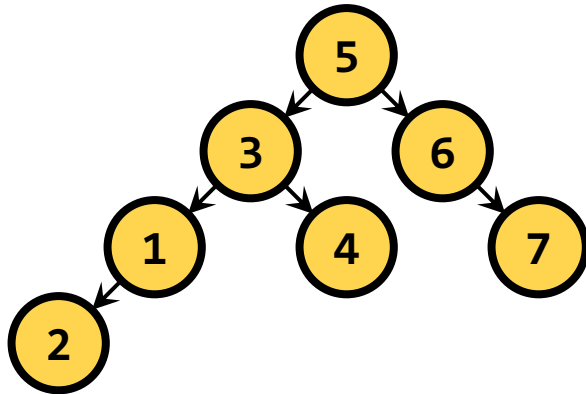
$$C(0) = C(1) = 1$$
$$C(n) = C(0)C(n-1) + C(1)C(n-2) + \dots + C(n-1)C(0)$$

This is the Catalan series: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, ...

Answer to this series is:

$$C(n) = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{n!(n+1)!}$$

# There Exist Many Invalid BSTs

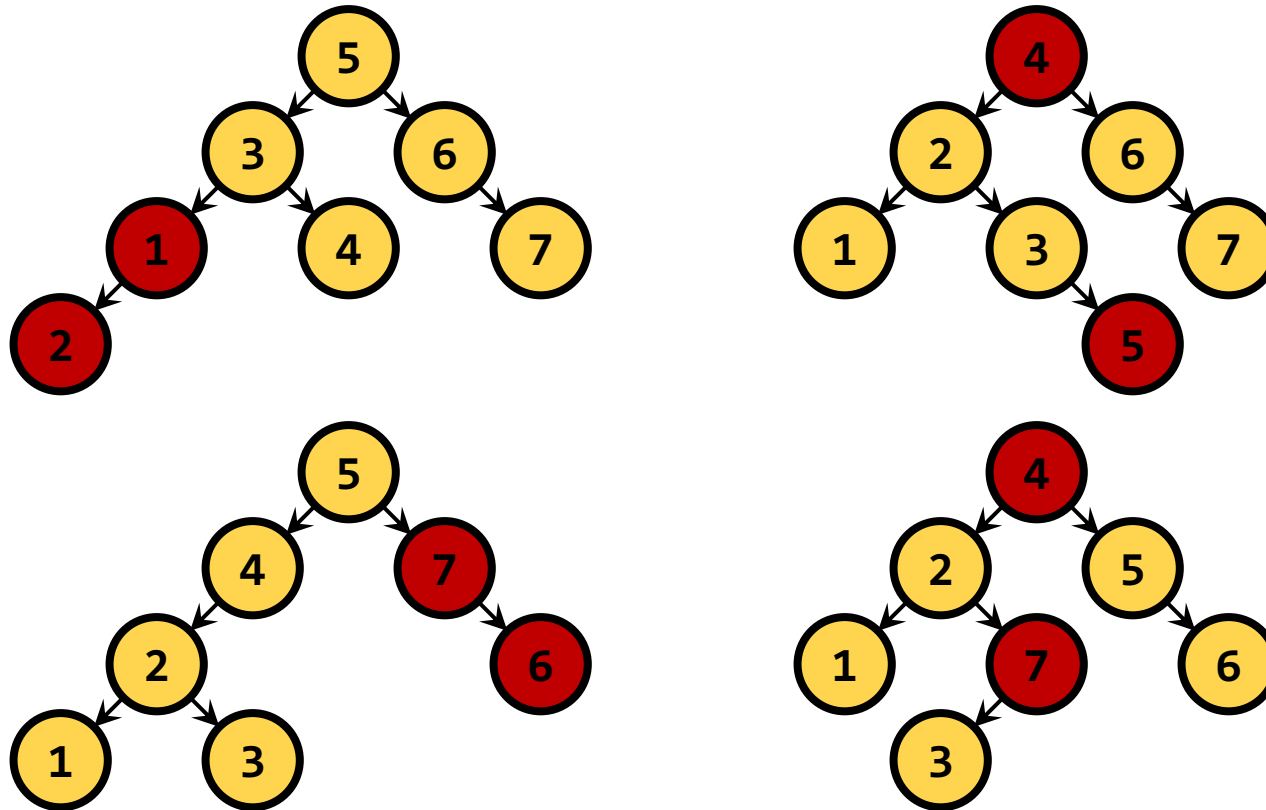


... and many more.

How many?



# There Exist Many Invalid BSTs



... and many more.

How many?

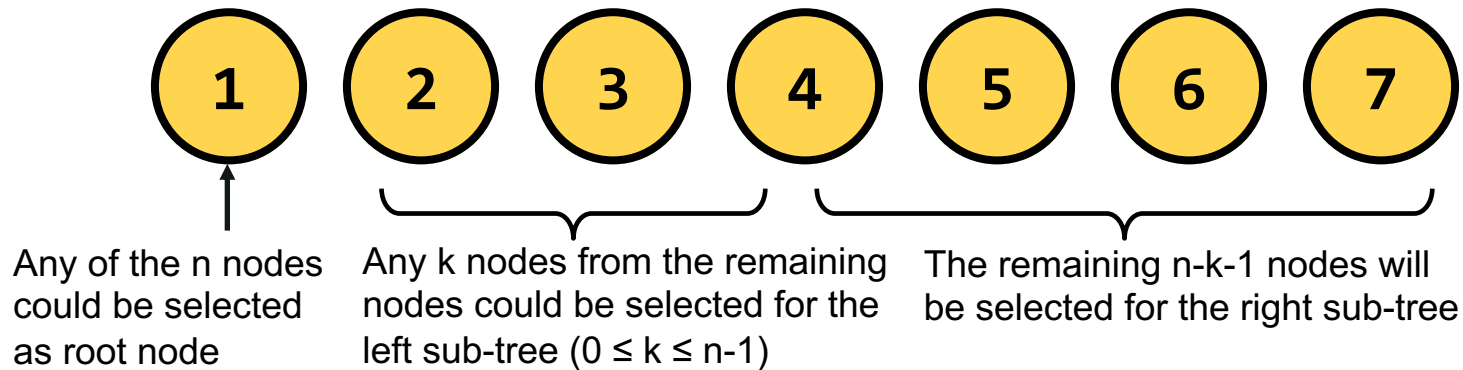


$$\frac{(2n)!}{(n+1)!} - \frac{(2n)!}{n!(n+1)!}$$

# There Exist Many Invalid BSTs

Given  $n$  vertices, how many BTs can we possibly build?

Let  $T(n)$  be the number of **binary trees (BTs)** using  $n$  nodes, **no matter being valid or invalid BST.**



$$T(0) = T(1) = 1$$

Rewrite  $T(n)$  by construction:  $T(n) = \binom{n}{1} \sum_{k=0}^{n-1} \binom{n-1}{k} T(k) T(n-k-1)$

Answer to this series is: 
$$T(n) = \frac{(2n)!}{(n+1)!}$$

# There Exist Many Invalid BSTs

Given  $n$  vertices, how many BTs can we possibly build?

Let  $T(n)$  be the number of binary trees (BTs) using  $n$  nodes, no matter being valid or invalid BST.

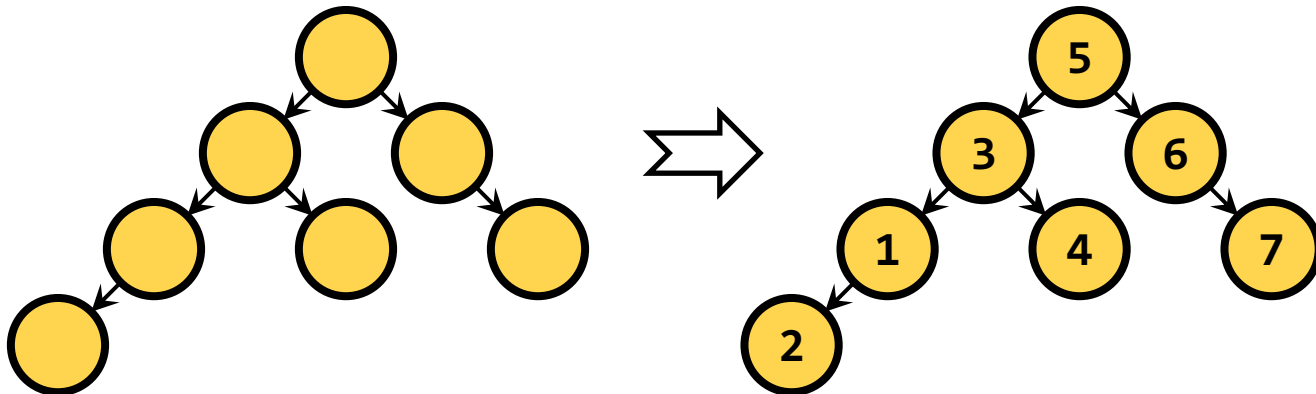
$$T(0) = T(1) = 1$$

$$\text{Rewrite } T(n) \text{ by construction: } T(n) = \sum_{k=0}^{n-1} \binom{n-1}{k} T(k) T(n-k-1)$$

$$\text{Answer to this series is: } T(n) = \frac{(2n)!}{(n+1)!}$$

An easier way to get the answer:

Each **unique BT structure** corresponds to a **unique valid BST**, e.g., given the following structure, the root must be 5, because there are 4 nodes smaller than root and 4 greater than root. Similar for remaining nodes.

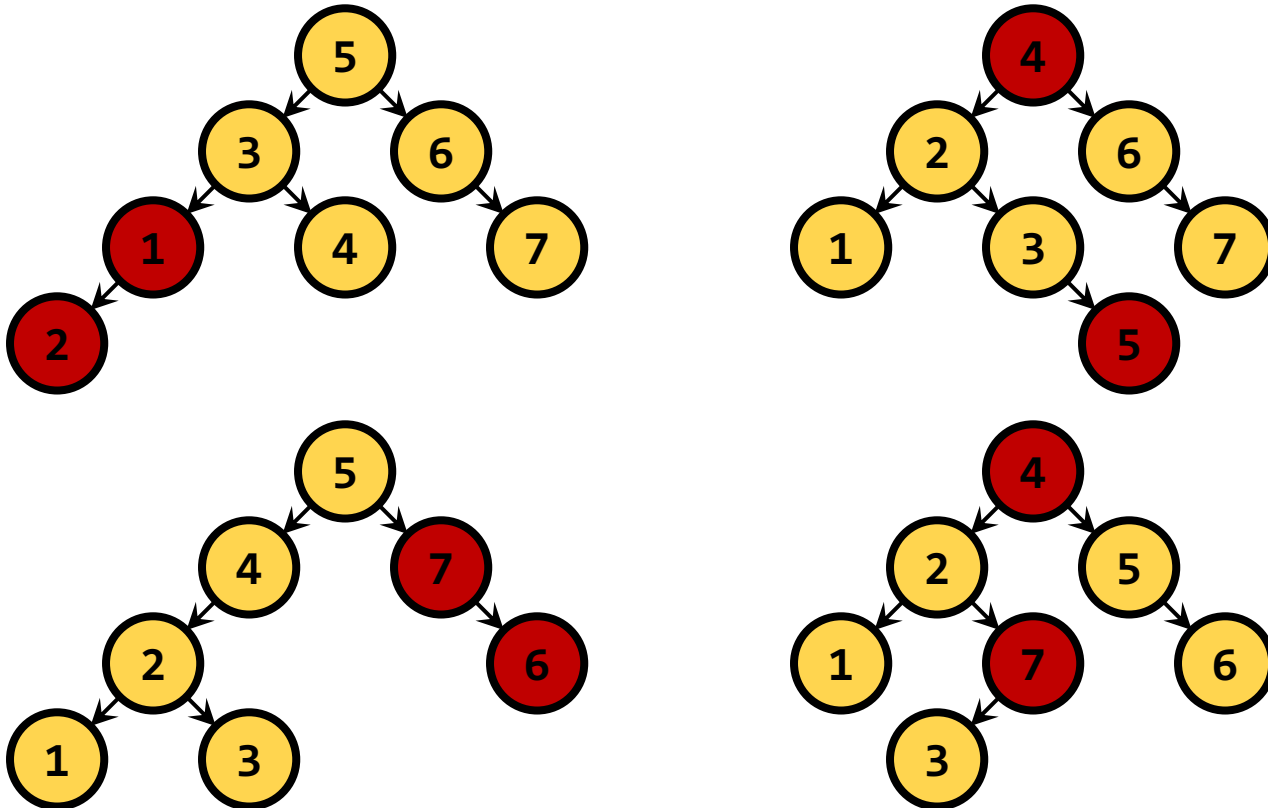


If we do not care about validity of BST, for each BST, we can permute  $n$  numbers arbitrarily to create BTs.  $n$  numbers will give  $n!$  permutations. As a result:

$$T(n) = C(n) \cdot n! = \frac{(2n)!}{n! (n+1)!} \cdot n! = \frac{(2n)!}{(n+1)!}$$

# There Exist Many Invalid BSTs

Given  $n$  vertices, how many invalid BSTs can we possibly build?

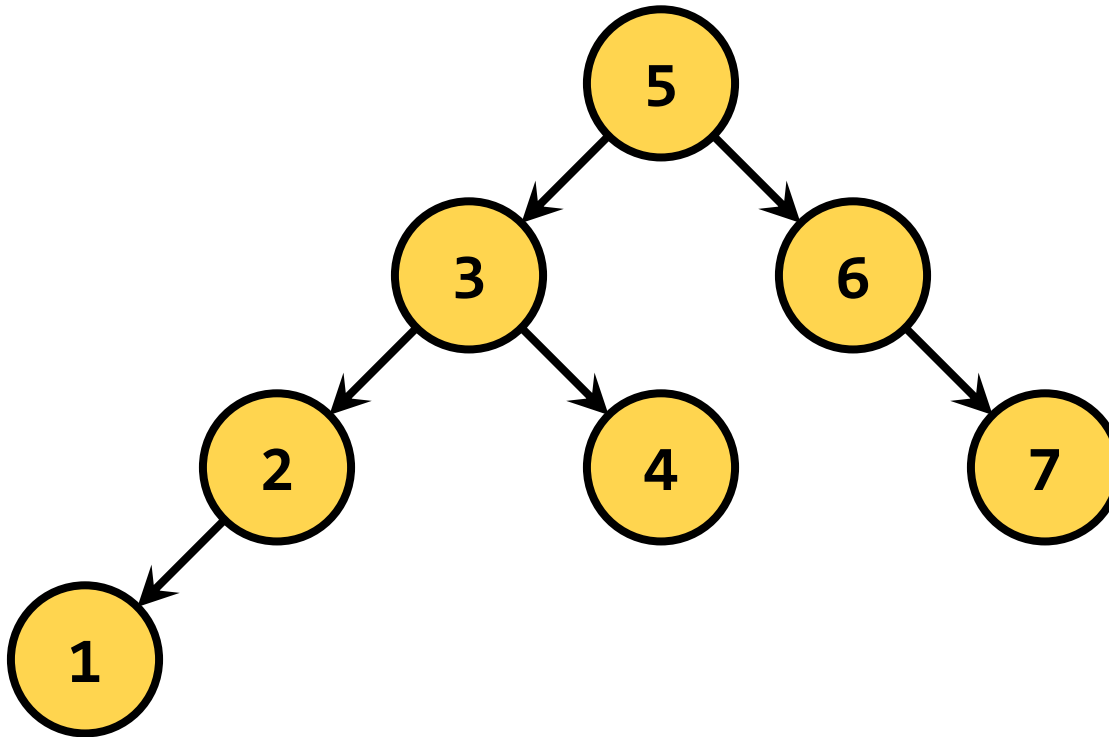


$$\# \text{ Invalid BSTs} = \# \text{ BTs} - \# \text{ Valid BSTs} = T(n) - C(n) = \frac{(2n)!}{(n+1)!} - \frac{(2n)!}{n!(n+1)!}$$



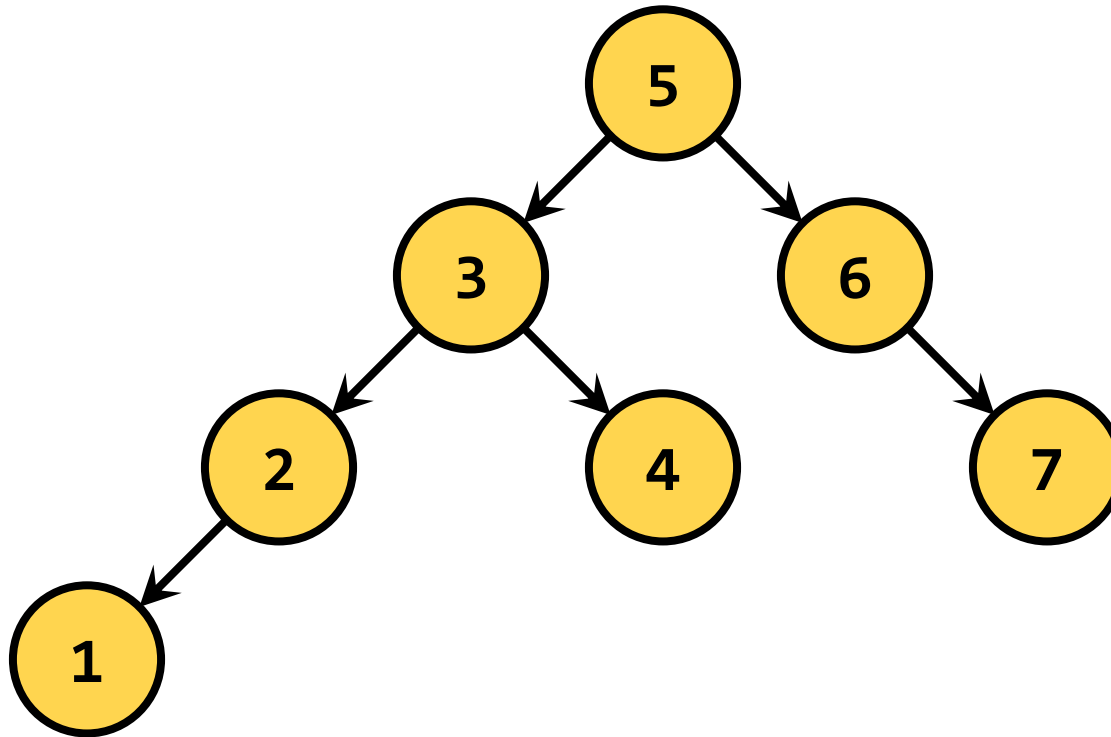
# Operations on BSTs

# search in BSTs



search compares the desired key to the current vertex,  
visiting left or right children as appropriate.

# search in BSTs

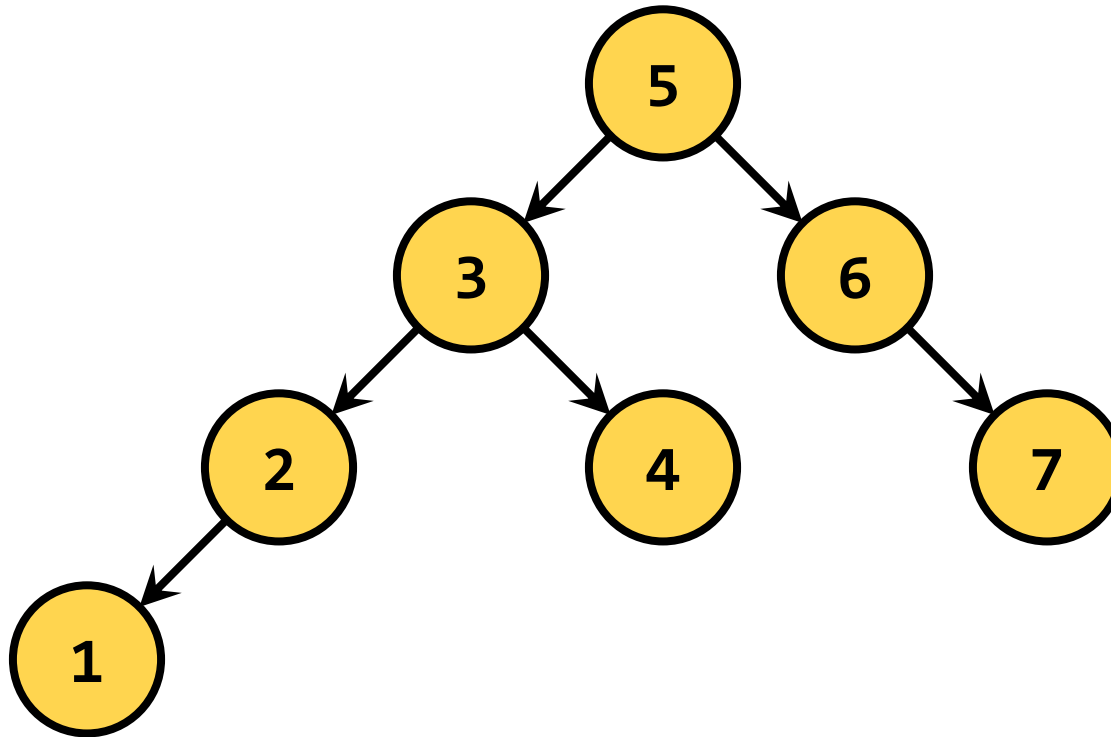


For example, `search(4)` compares the 4 to the 5, then visits its left child of 3, then visits its right child of 4.

Write pseudocode to implement this algorithm!



# search in BSTs



If we desire a **non-existent** key, such as `search(4.5)`, we can either **return the last seen node** (in this case, 4) or we can throw an exception. For now, let's do the former (helpful to other algorithms that may use `search()` function, e.g., **insert**. 28

# insert in BSTs

```
algorithm insert(root, key_to_insert):  
    x = search(root, key_to_insert)  
    v = new vertex with key_to_insert  
    if key_to_insert > x.key:  
        x.right = v  
    if key_to_insert < x.key:  
        x.left = v  
    if key_to_insert == x.key:  
        return
```


Explain on board: insert(4.5), insert(6.5), insert(4)

**Runtime:**  $O(\log n)$  if balanced,  $O(n)$  otherwise

Explain on board: an extremely unbalanced BST

# delete in BSTs

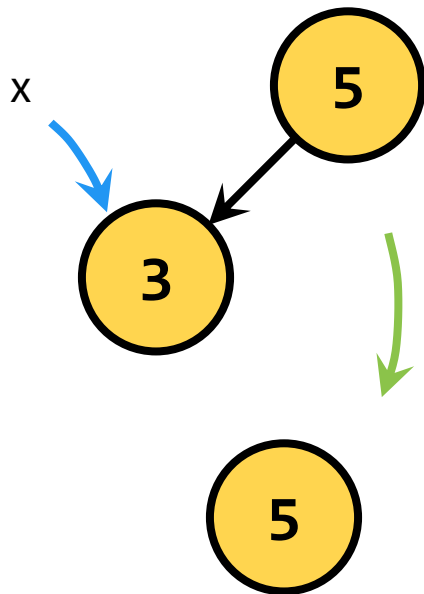
```
algorithm delete(root, key_to_delete):  
  x = search(root, key_to_delete)  
  if key_to_delete == x.key:  
    delete x
```

 This is somewhat complicated ...

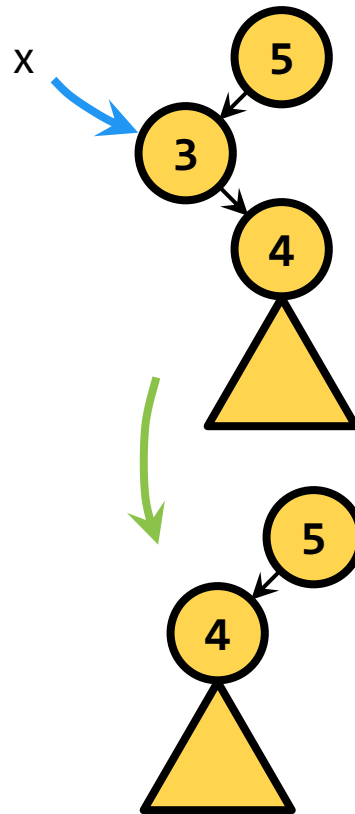
**Runtime:**  $O(\log n)$  if balanced,  $O(n)$  otherwise

# delete in BSTs

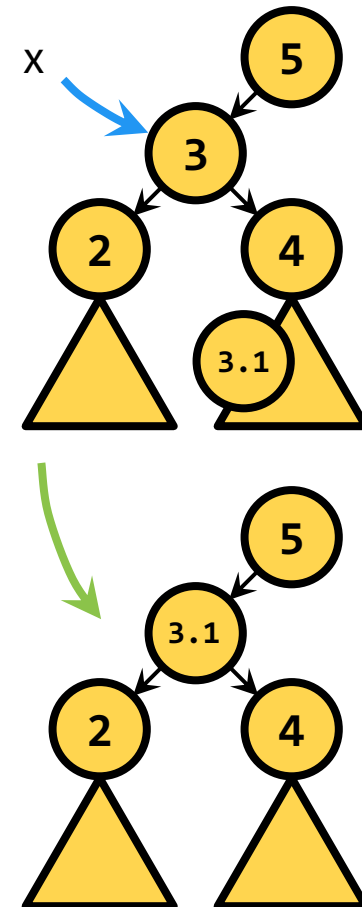
**Case 1:** x is a leaf  
Just delete x



**Case 2:** x has 1 child  
Move its child up

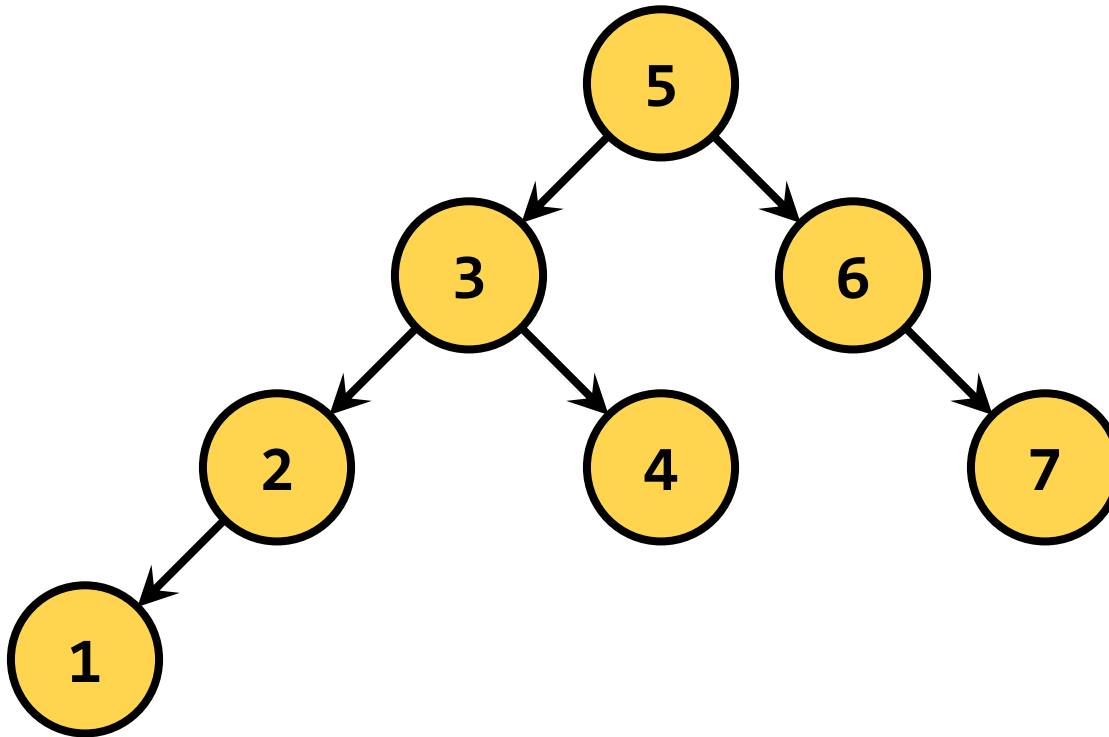


**Case 3:** x has 2 children  
Replace x with its **successor**



Explain on board: why the successor is selected in case 3.  
It should be greater than 2 and all its descendants, while being less than 4 and all its descendants. So we choose the smallest value in the right-descendants of 3, i.e., successor.

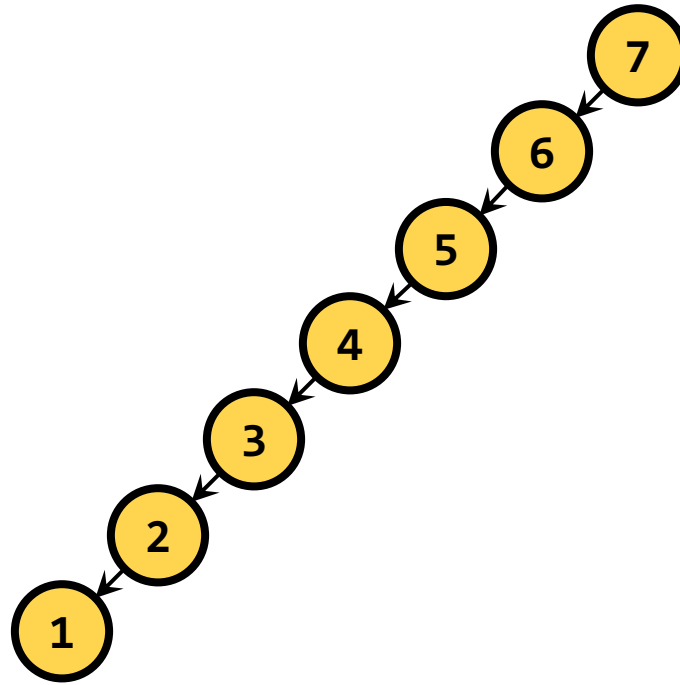
# Runtime Analysis



Runtime of **search** (which insert and delete both call) is  **$O(\text{depth of tree})$** .



# Runtime Analysis



But this is a valid BST and the depth of the tree is  $n$ , resulting in a runtime of  $O(n)$  for search.

In what order would we need to insert vertices to generate this tree?

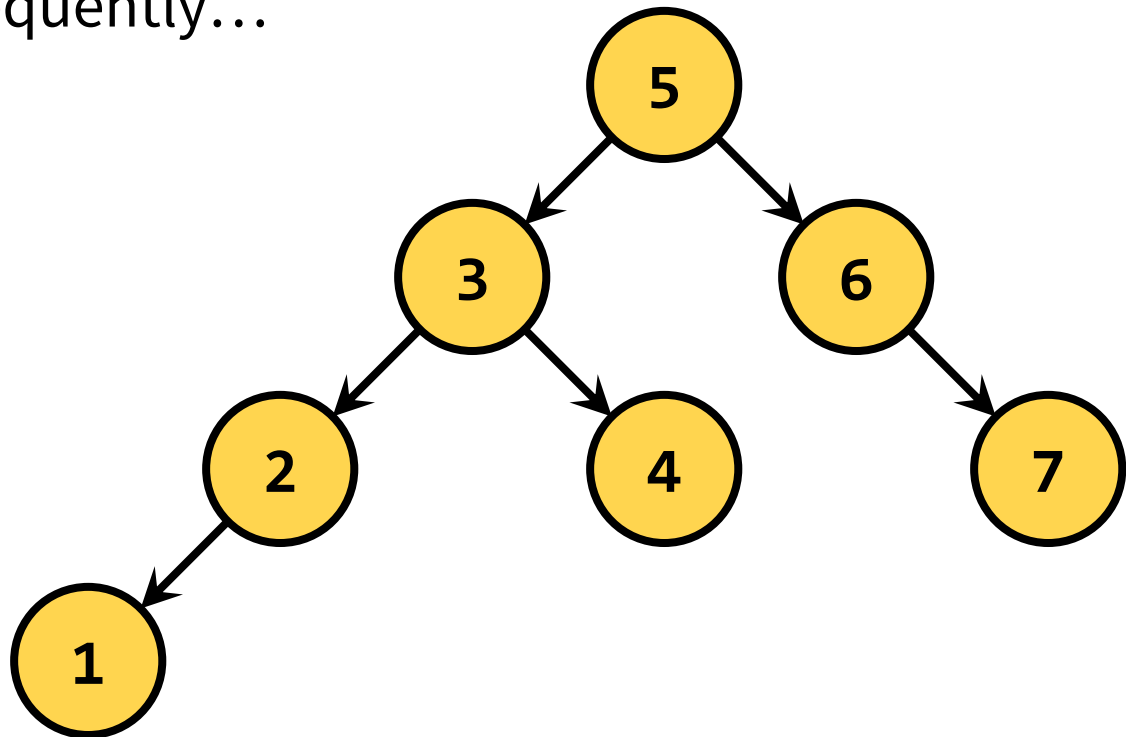


# What To Do?

We could keep track of the depth of the tree. If it gets too tall, re-do everything from scratch.

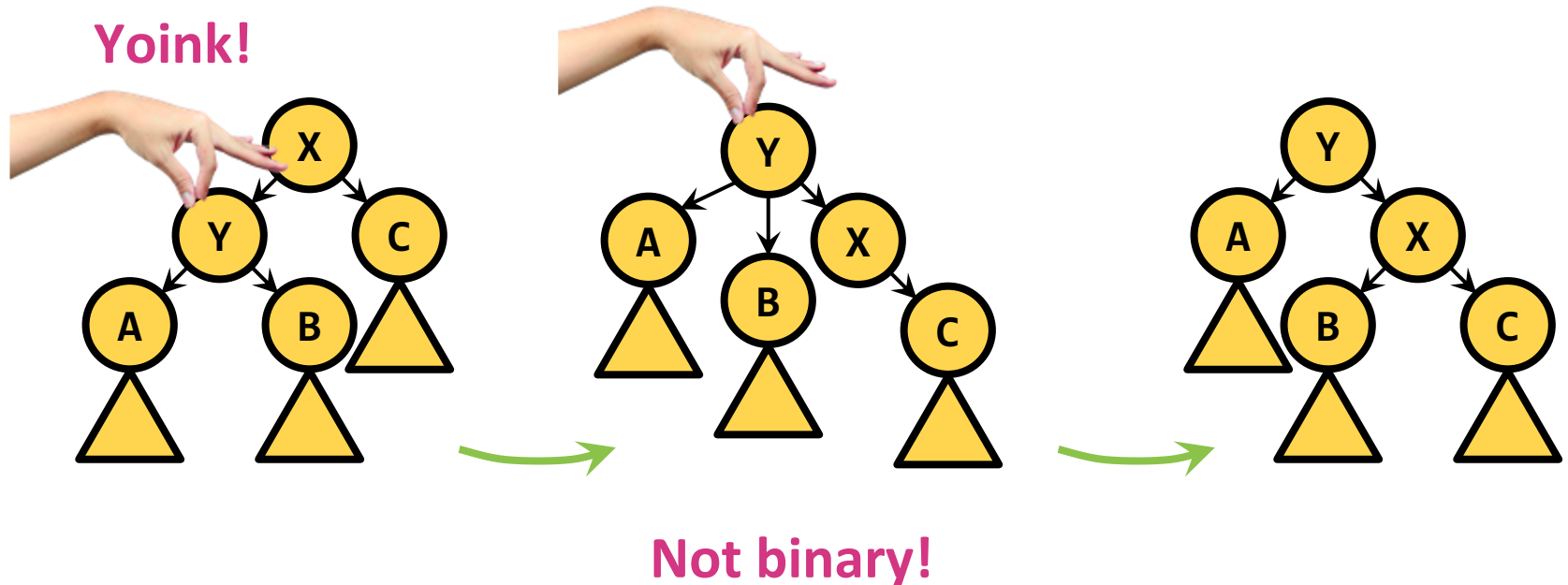
But this is time-consuming because we have to reconstruct the whole tree frequently...

Any other ideas?



# Idea 1: Rotations

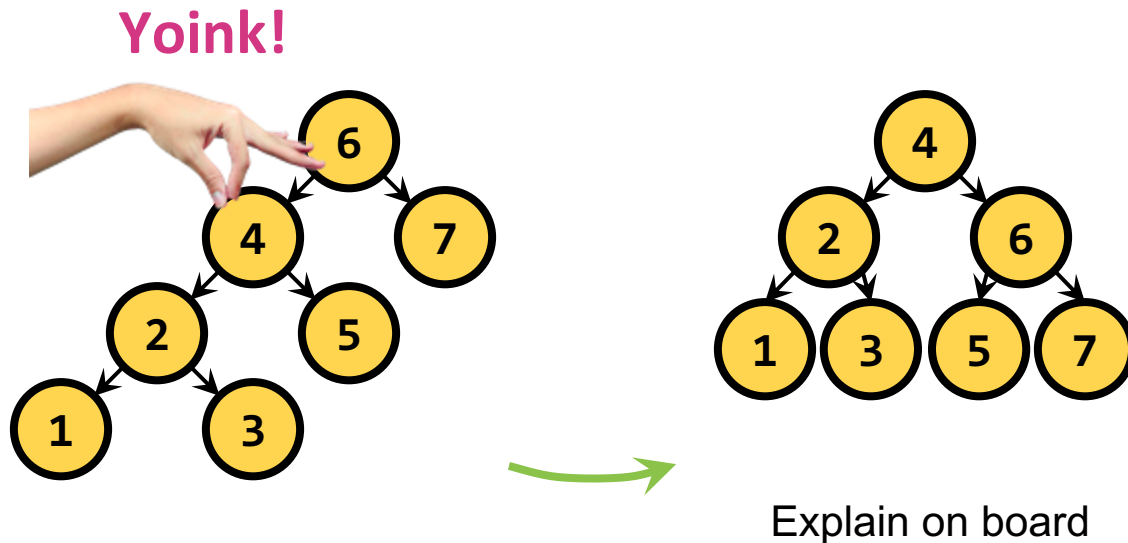
Maintain the BST property, and move some of the vertices (but not all of them) around.



Check the BST properties on Y and X

# Idea 1: Rotations

Maintain the BST property, and move some of the vertices (but not all of them) around.



# Idea 2: Proxy for Balance

Maintaining **perfect balance** is too difficult.

Checking for balance and which node to rotate is difficult

Instead, let's determine some proxy for balance.

i.e. If the tree satisfies **some property**, then it's "pretty balanced."

If during tree modification (construction, insert, delete, etc.) the property no longer holds, we can **maintain this property** using **rotations**.

# Red-Black Trees (RB-Tree)

Advanced Topic (Optional)

# Red-Black Trees

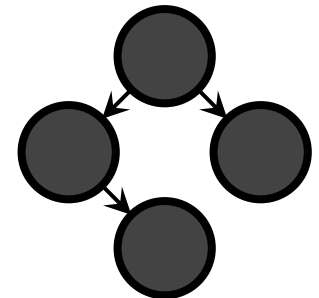
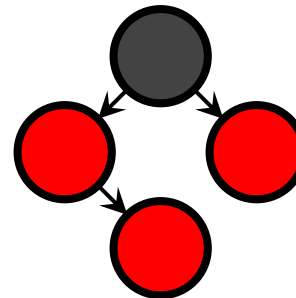
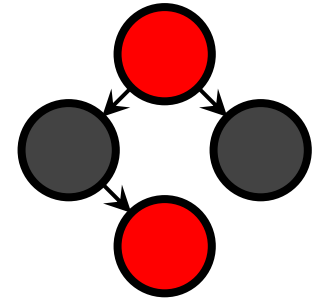
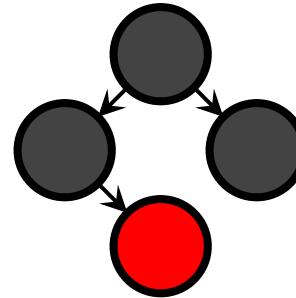
There exist many ways to achieve this proxy for balance, but here we'll study the **red-black tree**.

1. Every vertex is colored **red** or **black**.
2. The root vertex is a **black** vertex.
3. A NIL child is a **black** vertex.
4. The child of a **red** vertex must be a **black** vertex.
5. For all vertices  $v$ , all paths from  $v$  to its NIL descendants have the same number of **black** vertices.

We can be sure that the tree is **pretty balanced** as long as these **proxy properties hold**.

# Red-Black Trees by Example

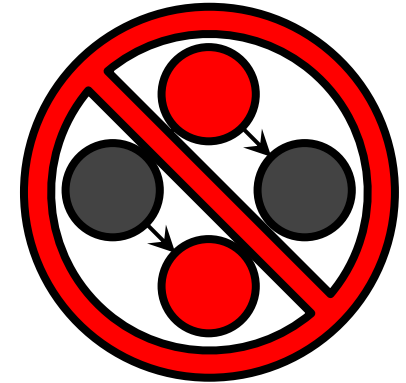
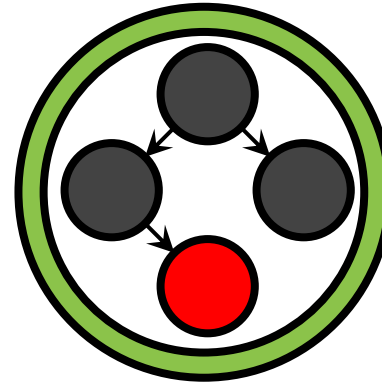
1. Every vertex is colored **red** or **black**.
2. The root vertex is a **black** vertex.
3. A NIL child is a **black** vertex.
4. The child of a **red** vertex must be a **black** vertex.
5. For all vertices  $v$ , all paths from  $v$  to its NIL descendants have the same number of **black** vertices.



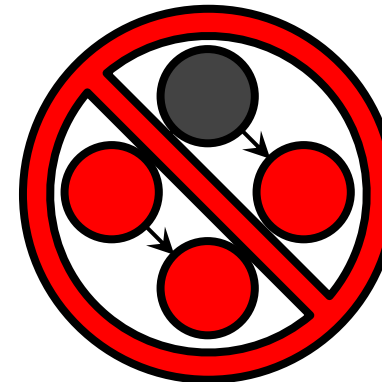


# Red-Black Trees by Example

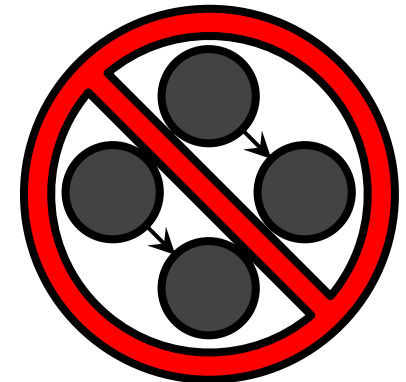
1. Every vertex is colored **red** or **black**.
2. The root vertex is a **black** vertex.
3. A NIL child is a **black** vertex.
4. The child of a **red** vertex must be a **black** vertex.
5. For all vertices  $v$ , all paths from  $v$  to its NIL descendants have the same number of **black** vertices.



Violates 2



Violates 4



Violates 5

# Red-Black Trees

Maintaining these properties maintains a “pretty balanced” BST.

Intuition:

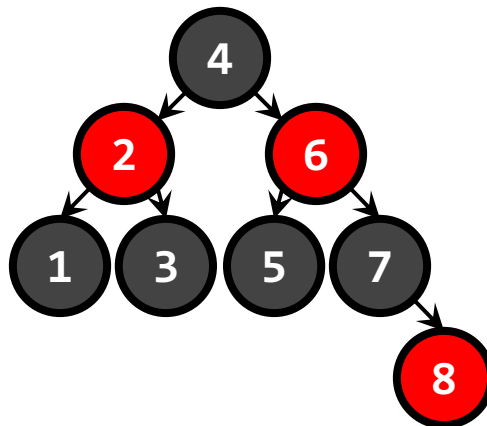
The **black** vertices are balanced.

Rule #5: For all vertices  $v$ , all paths from  $v$  to its NIL descendants have the same number of **black** vertices

The **red** vertices are “spread out.”

Rule #4: The child of a **red** vertex must be a **black** vertex

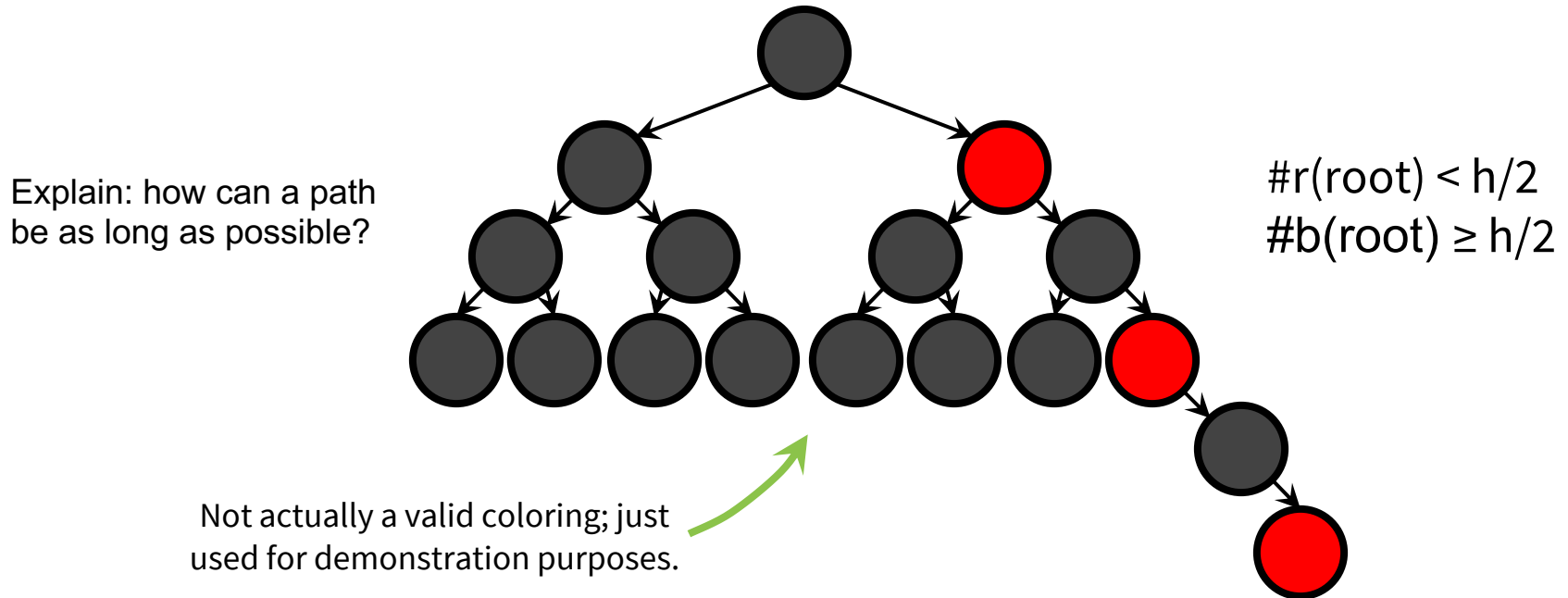
We can maintain this property as we insert/delete vertices, using rotations.



# Red-Black Trees

To see why a red-black tree is “pretty balanced,” consider that its height is at most  $O(\log(n))$ .

Property: One path could be twice as long as the others if we pad it with red vertices, but **at most twice** as long as the others.



# Red-Black Trees

**Lemma:** The number of **node in a subtree of x** is at least  $n(x) \geq 2^{b(x)} - 1$ ,  $b(x)$  is the **number of black nodes** from x to NIL descendants.

**Proof:**

To prove this statement, we proceed by induction.

For our **base case**, note that a NIL node has  $b(x) = 0$  and at least  $2^0 - 1 = 0$  non-NIL descendants; same for a single black or red node.

For our **inductive step**, let  $n(x)$  be the number of non-NIL descendants of x. Then:

$$\begin{aligned} n(x) &= 1 + n(x.\text{left}) + n(x.\text{right}) \\ &\geq 1 + (2^{b(x)-1} - 1) + (2^{b(x)-1} - 1) \\ &= 2^{b(x)} - 1 \end{aligned}$$

Thus, the number of non-NIL descendants of x is at least  $2^{b(x)} - 1$ .

# Red-Black Trees

**Theorem:** A Red-Black Tree has height  $h \leq 2 \log_2(n+1) = O(\log n)$ .

Proof:

By our lemma, the number of non-NIL descendants of  $x$  is at least  $2^{b(x)} - 1$ .

Notice that on any root to NIL path there are no two consecutive red vertices (otherwise the tree violates rule 4);

So the number of black vertices is at least the number of red vertices.


Thus,  $b(\text{root})$  is at least half of the height, i.e.,  $b(\text{root}) \geq h/2$

Let  $n$  be the number of vertices in the tree

then  $n \geq 2^{b(\text{root})} - 1 \geq 2^{h/2} - 1$ , and hence  $h \leq 2 \log_2(n+1)$ .

# insert in Red-Black Trees

```
algorithm rb_insert(root, key_to_insert):  
  x = search(root, key_to_insert)  
  v = new red vertex with key_to_insert  
  if key_to_insert > x.key:  
    x.right = v  
    fix things up, if necessary  
  if key_to_insert < x.key:  
    x.left = v  
    fix things up, if necessary  
  if key_to_insert == x.key:  
    return
```

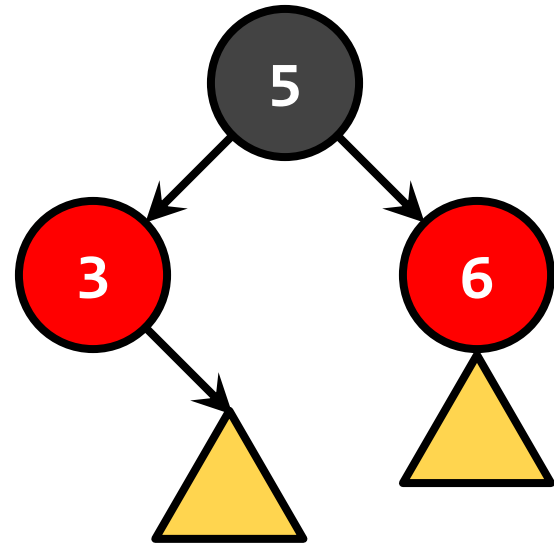
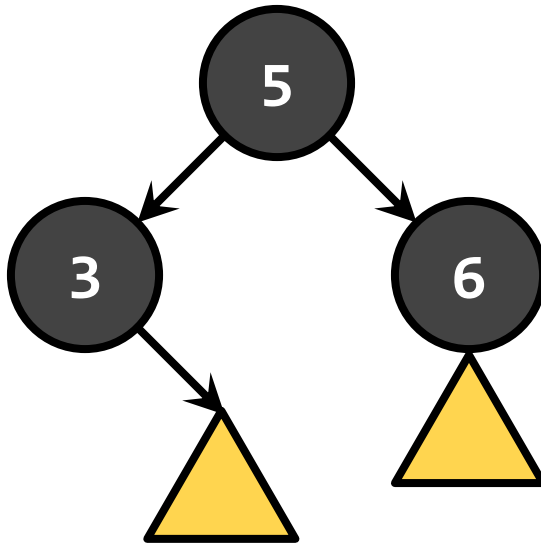


What does that mean?

# insert in Red-Black Trees

What does “if necessary” mean?

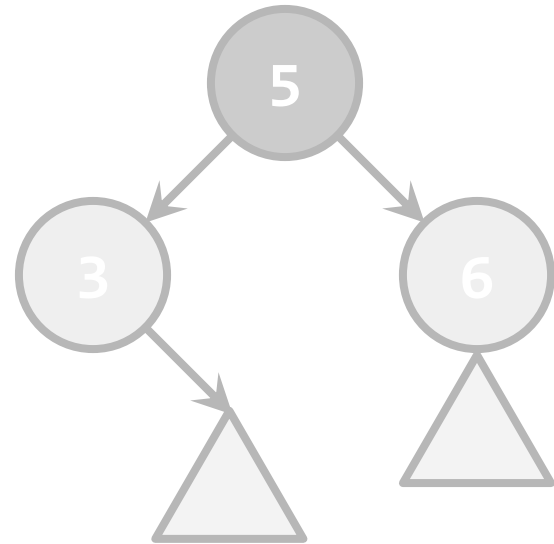
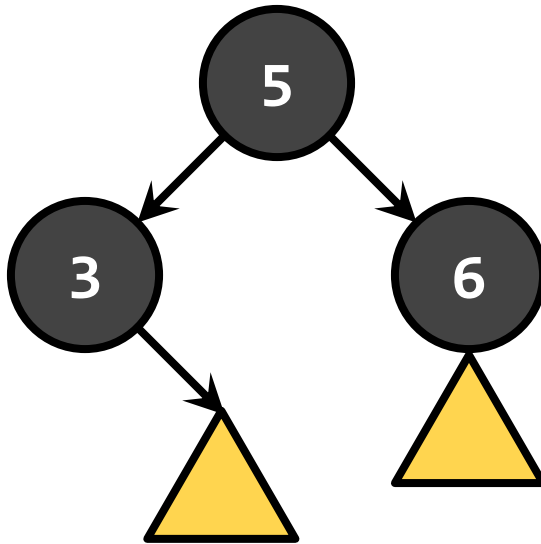
Suppose we want to insert(1).



# insert in Red-Black Trees

What does “if necessary” mean?

Suppose we want to insert(1).

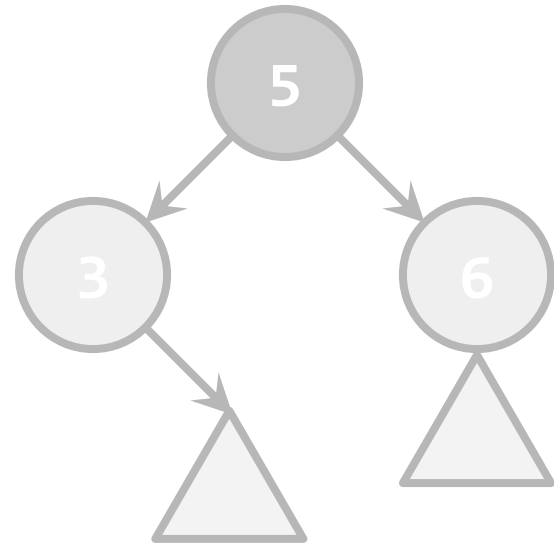
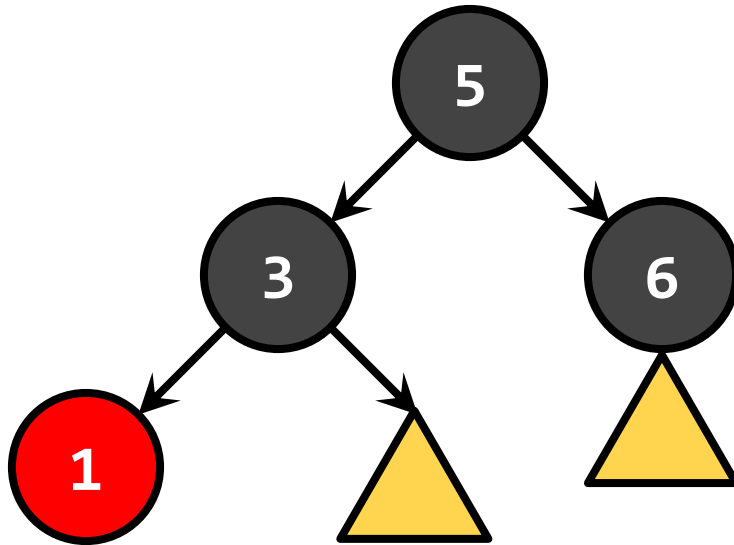




# insert in Red-Black Trees

What does “if necessary” mean?

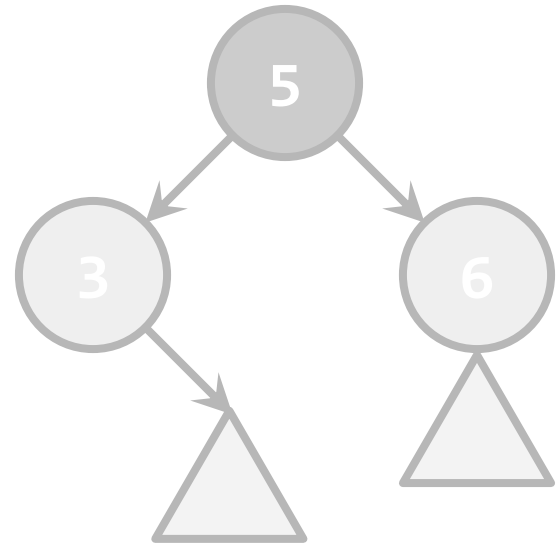
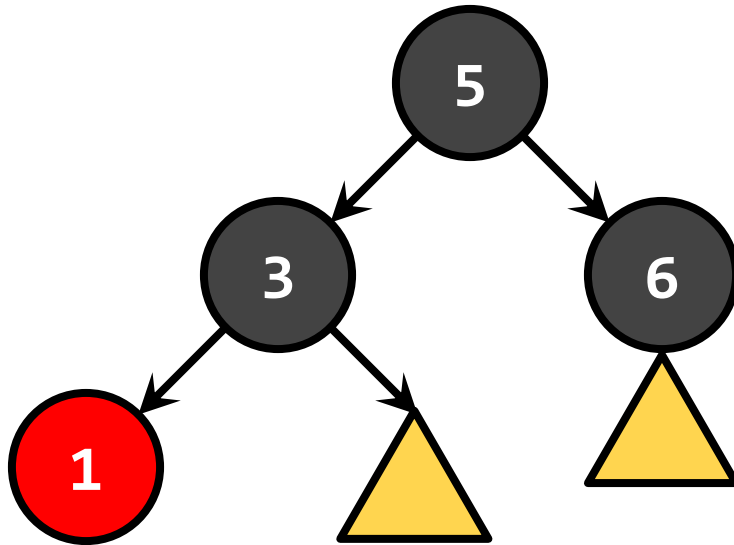
Suppose we want to insert(1).



# insert in Red-Black Trees

What does “if necessary” mean?

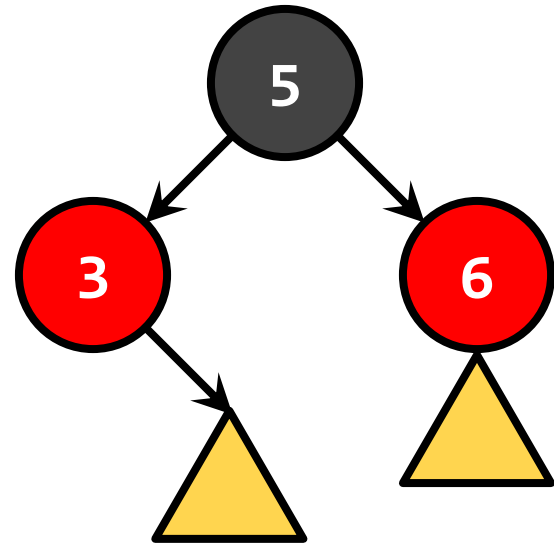
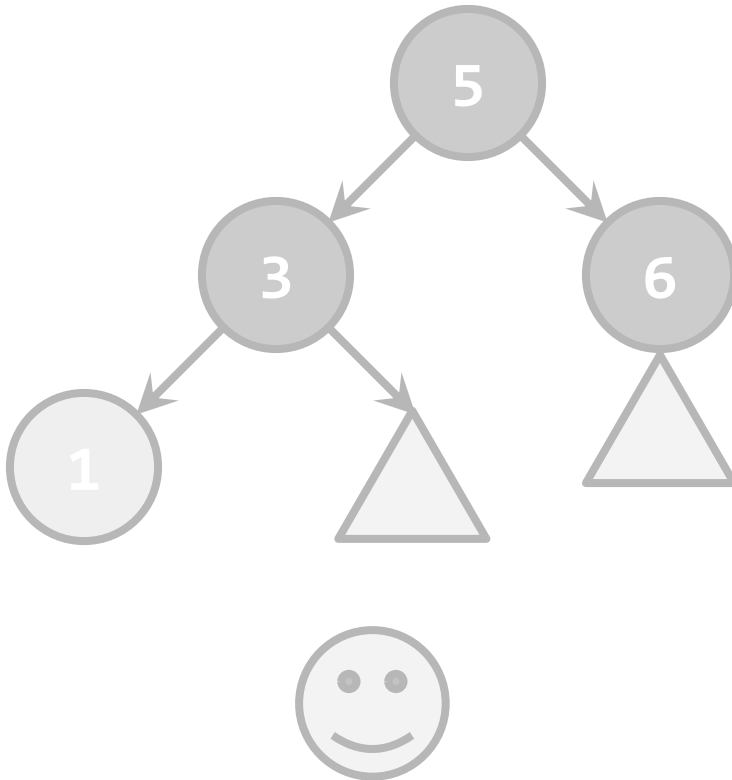
Suppose we want to insert(1).



# insert in Red-Black Trees

What does “if necessary” mean?

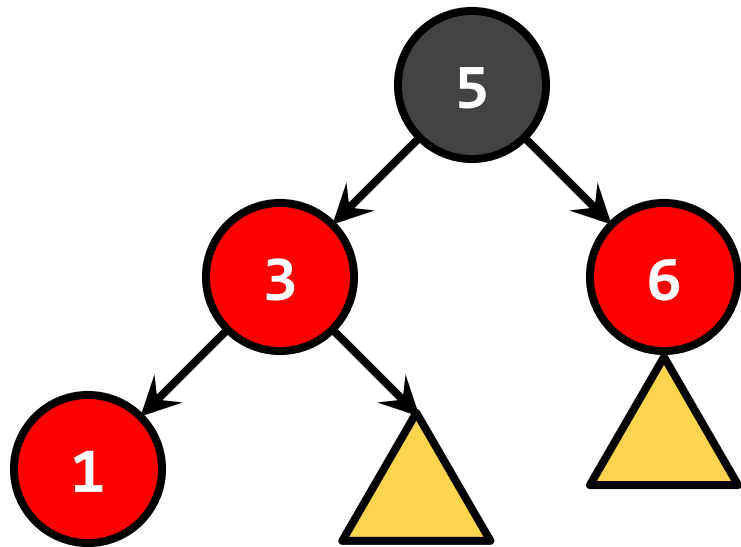
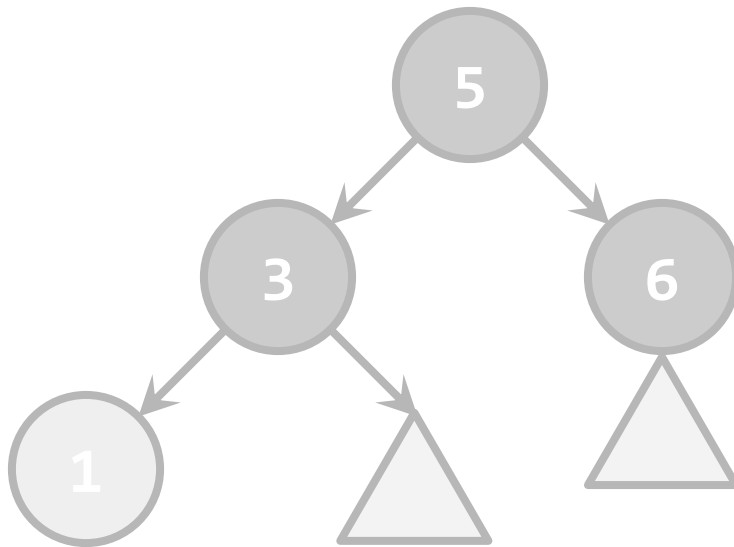
Suppose we want to insert(1).



# insert in Red-Black Trees

What does “if necessary” mean?

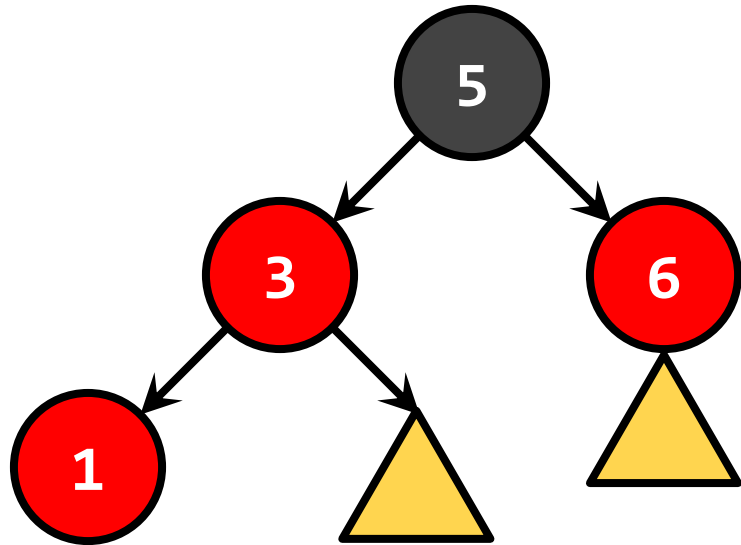
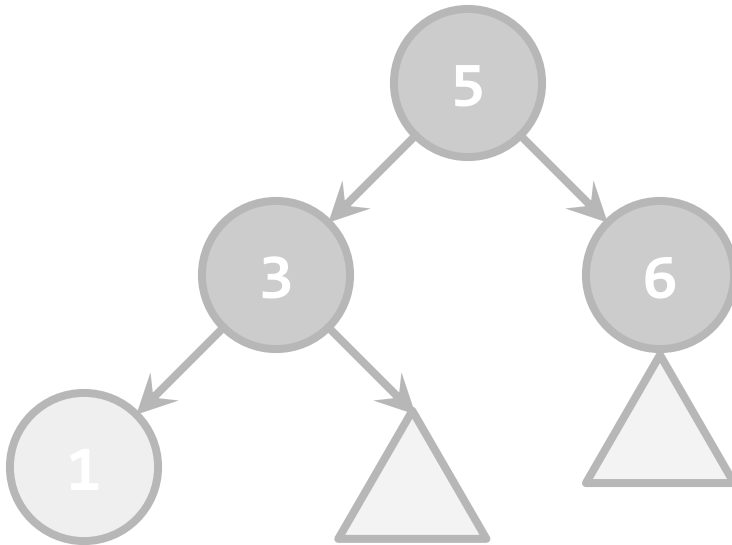
Suppose we want to insert(1).



# insert in Red-Black Trees

What does “if necessary” mean?

Suppose we want to insert(1).

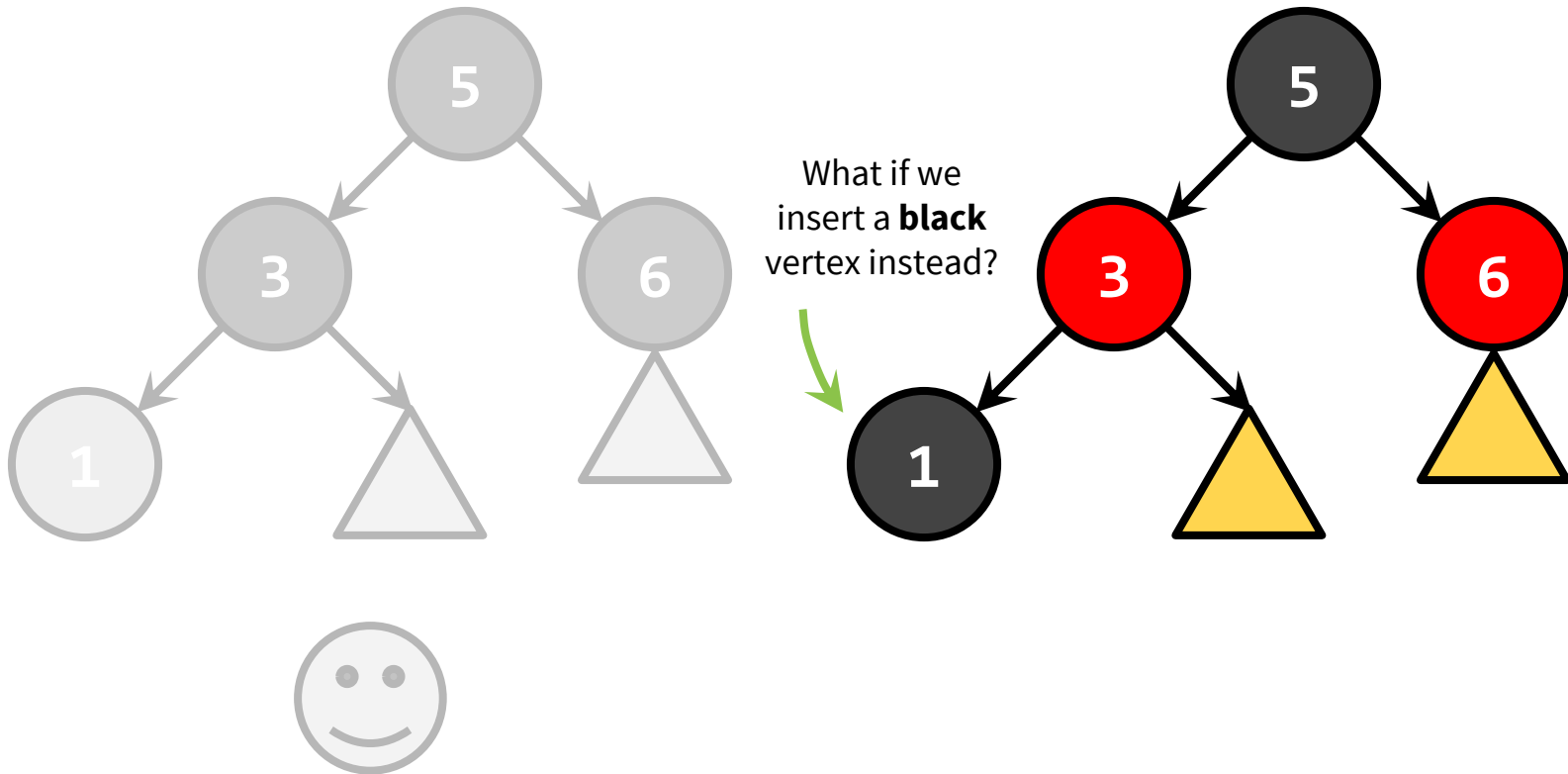


Violates 4

# insert in Red-Black Trees

What does “if necessary” mean?

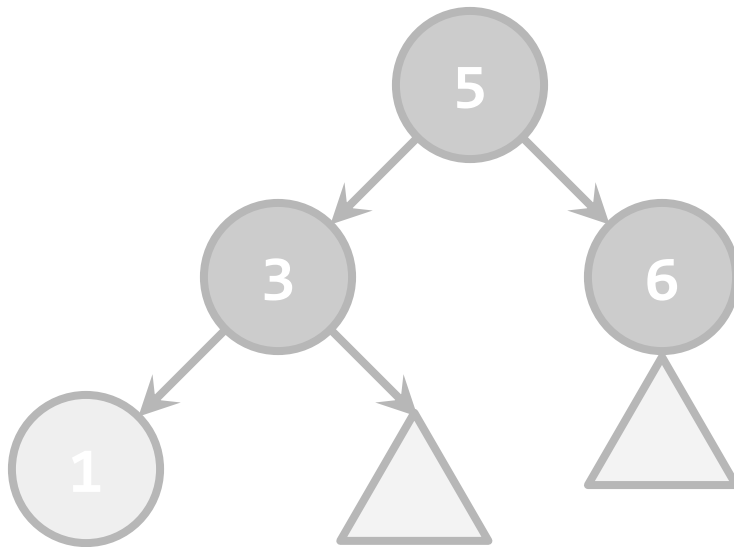
Suppose we want to insert(1).



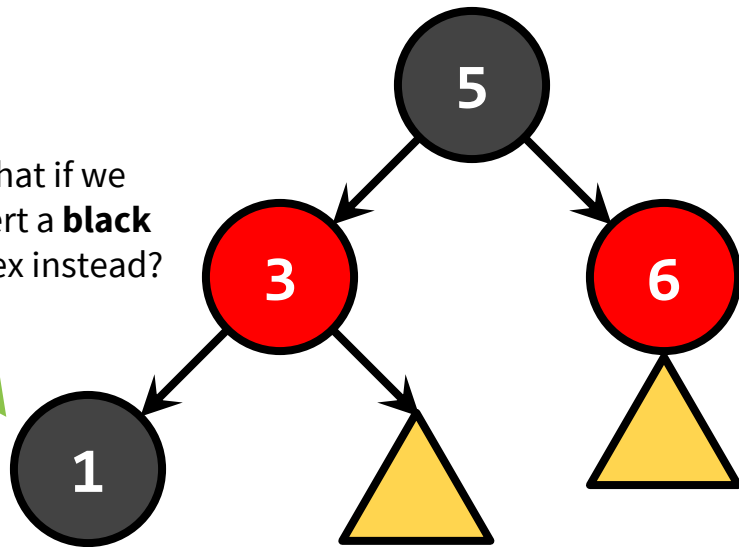
# insert in Red-Black Trees

What does “if necessary” mean?

Suppose we want to insert(1).



What if we  
insert a **black**  
vertex instead?



Violates 5

# insert in Red-Black Trees

What does “if necessary” mean?

So it seems we’re happy if the parent of the inserted vertex is **black**.

Check the rules:

1. Every vertex is colored **red** or **black**.
2. The root vertex is a **black** vertex.
3. A NIL child is a **black** vertex.
4. The child of a **red** vertex must be a **black** vertex.
5. For all vertices  $v$ , all paths from  $v$  to its NIL descendants have the same number of **black** vertices.

But there’s an issue if the parent of the inserted vertex is **red**.



# insert in Red-Black Trees

```
algorithm rb_insert(root, key_to_insert):  
    x = search(root, key_to_insert)  
    v = new red vertex with key_to_insert  
    if key_to_insert > x.key:  
        x.right = v  
        recolor(v)  
    if key_to_insert < x.key:  
        x.left = v  
        recolor(v)  
    if key_to_insert == x.key:  
        return
```

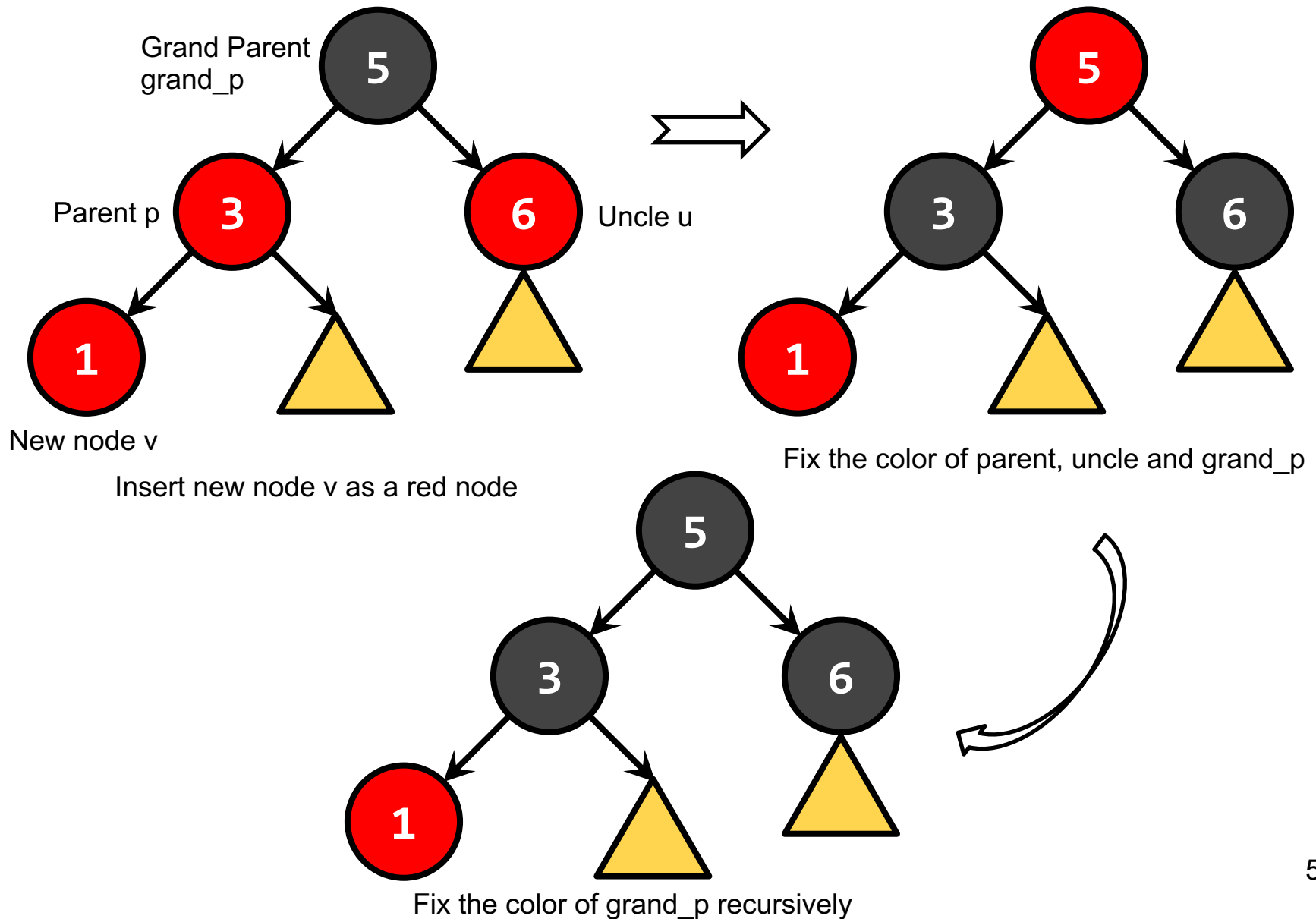
**Runtime:**  $O(\log n)$

# insert in Red-Black Trees

```
algorithm recolor(v):
    if v == root:
        v.color = black
        return
    p = parent(v)
    if p.color == black:
        return
    grand_p = p.parent
    uncle = grand_p.right
    if uncle.color == red:
        p.color = black
        uncle.color = black
        grand_p.color = red # maintain number of black vertices
        recolor(grand_p) # fix up color recursively
    else: # uncle.color == black
        p.color = black
        grand_p.color = red
        right_rotate(grand_p) # yoink
```

**Runtime:  $O(\log n)$**

# insert in Red-Black Trees



# Red-Black Trees

Since we maintain the red-black property in  $O(\log n)$ , then insert, delete, and search all require  $O(\log n)$ -time.

	Search	Insertion	Deletion
Linked list	$O(n)$	$O(n)$	$O(n)$
Arrays	$O(\log n)$	$O(n)$	$O(n)$
BST (unbalanced)	$O(n)$	$O(n)$	$O(n)$
BST (balanced)	$O(\log n)$	$O(\log n)$	$O(\log n)$
RBT (always balanced)	$O(\log n)$	$O(\log n)$	$O(\log n)$

# Red-Black Trees

Since we maintain the red-black property in  $O(\log n)$ , then insert, delete, and search all require  $O(\log n)$ -time.

	Search	Insertion	Deletion
Linked list	$O(n)$	$O(n)$	$O(n)$
Arrays	$O(\log n)$	$O(n)$	$O(n)$
BST (unbalanced)	$O(n)$	$O(n)$	$O(n)$
BST (balanced)	$O(\log n)$	$O(\log n)$	$O(\log n)$
RBT (always balanced)	$O(\log n)$	$O(\log n)$	$O(\log n)$

Acknowledgement: Part of the materials are adapted from Mary Wootter, Virginia Williams and David Eng's lectures on algorithms. We appreciate their contributions.