Tree Algorithms

Summer 2018 • Lecture 07/17

Announcements

- Alternate Midterm requests due 7/16.
- Homework 2
  - The hard deadline for **`hw2.zip`** is Thursday!
- Homework 3
  - **`hw3.pdf`** is live!
  - It's due next Tuesday 7/24 (hard deadline).
- Tutorial 4
  - Friday, 7/20 3:30-4:50 p.m. in STLC 115.
  - RSVP, so I can print enough copies for everyone: https://goo.gl/forms/eBupaH2NcwDRxuXS2 (requires Stanford email).

Course Overview

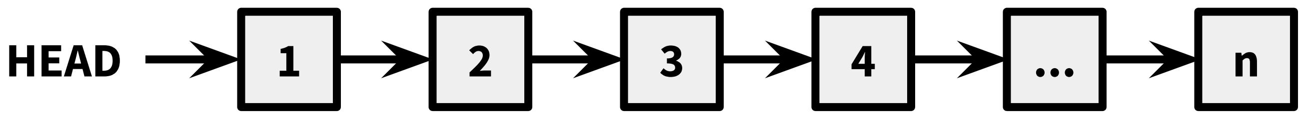
- Algorithmic Analysis
- Divide and Conquer
- Randomized Algorithms
- **Tree Algorithms**
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Algorithms

Today's Outline

- Tree Algorithms
  - Reading: CLRS: 12 and 13

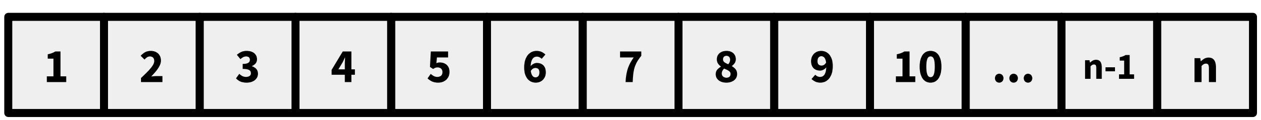
Binary Search Trees

Why BSTs?



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

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



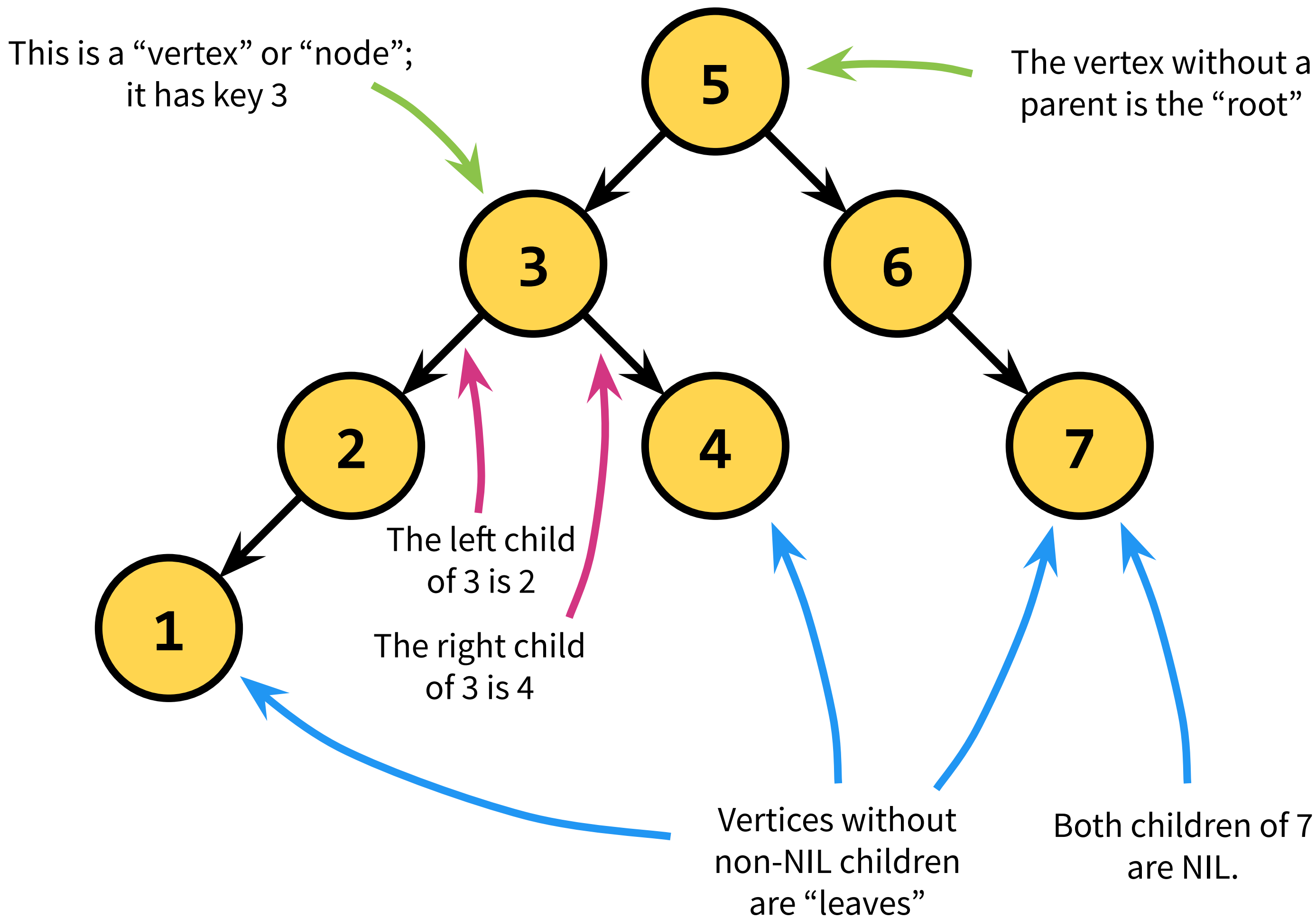
Sorted arrays: `O(log n)` search
`O(1)` select
`O(n)` insert/delete

“Get the k^{th} smallest element”

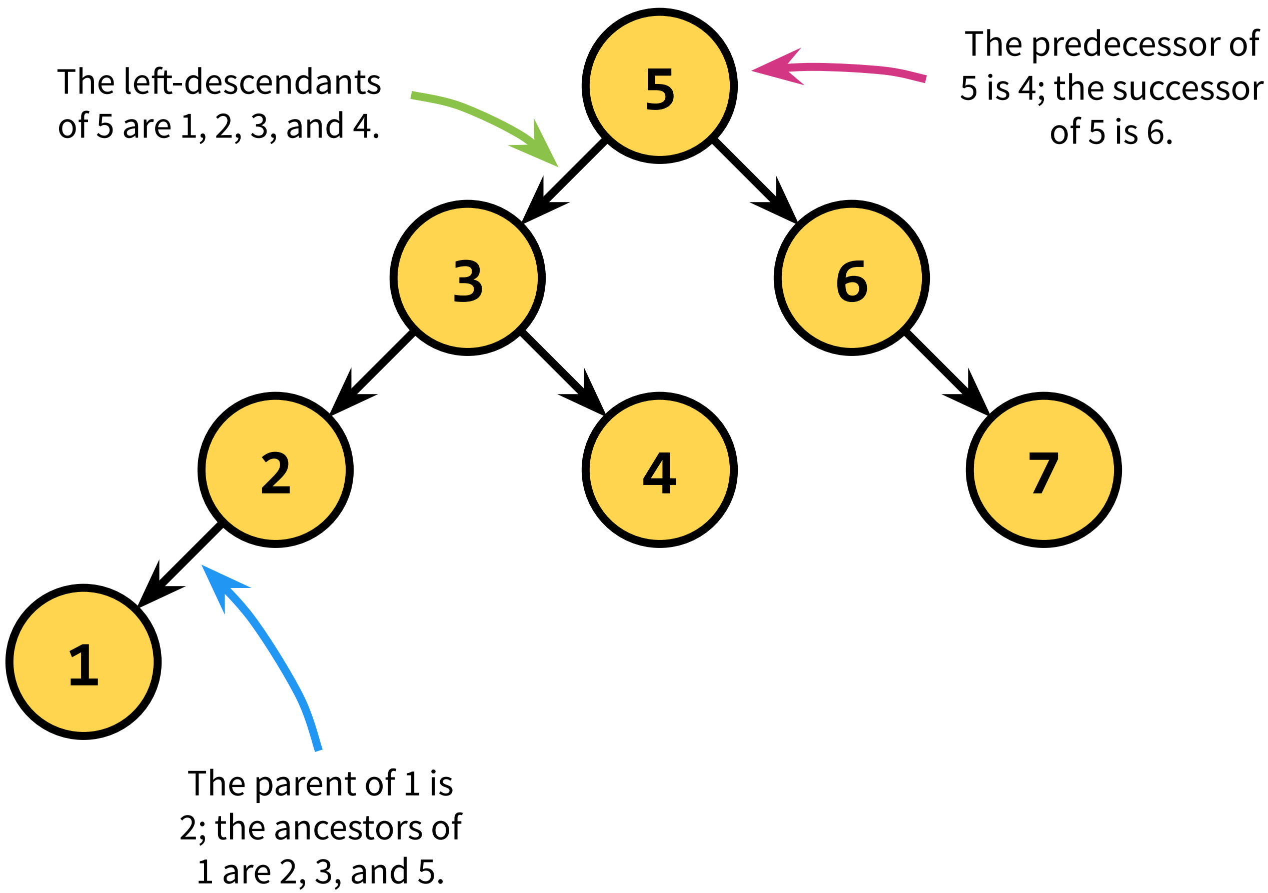
Why BSTs?

	Sorted linked lists	Sorted arrays	Binary search trees
Search	O(n)	O(log n)	O(log n)
Insert/Delete	O(1) given a pointer to the element; otherwise, O(n) to search for it	O(n)	O(log n)

Tree Terminology



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:

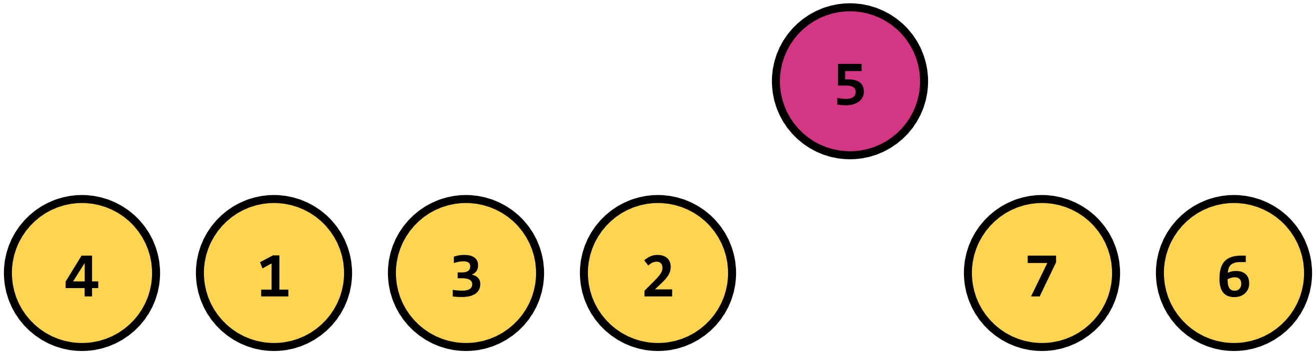
- Every left descendent of a vertex has key less than that vertex.
- Every right descendent of a vertex has key greater than that vertex.

Building BSTs by Example

Building BSTs by Example

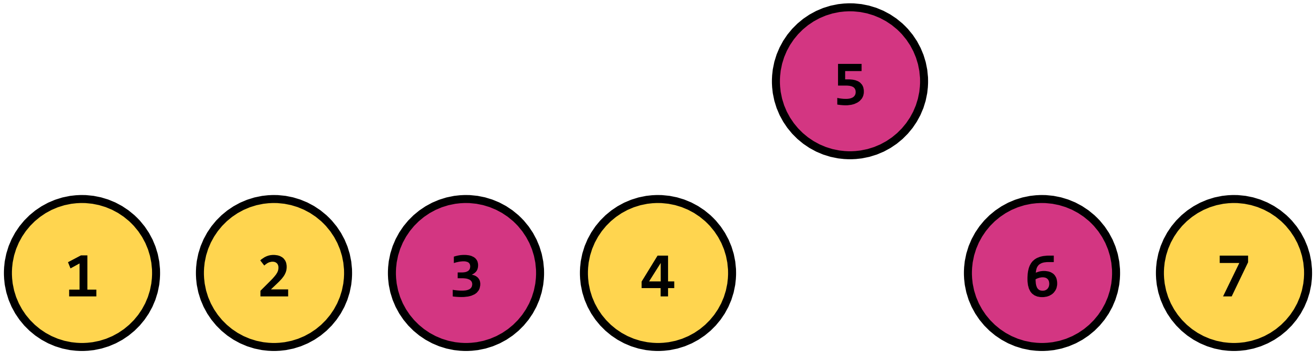
Let’s partition about one of the vertices …

Building BSTs by Example



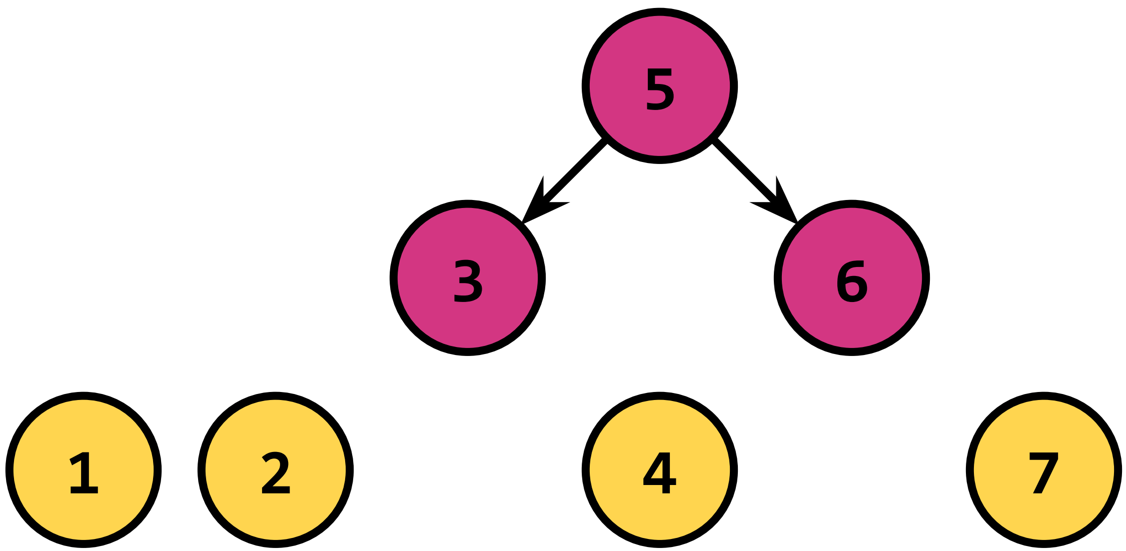
… and build a tree with that vertex as the root.

Building BSTs by Example



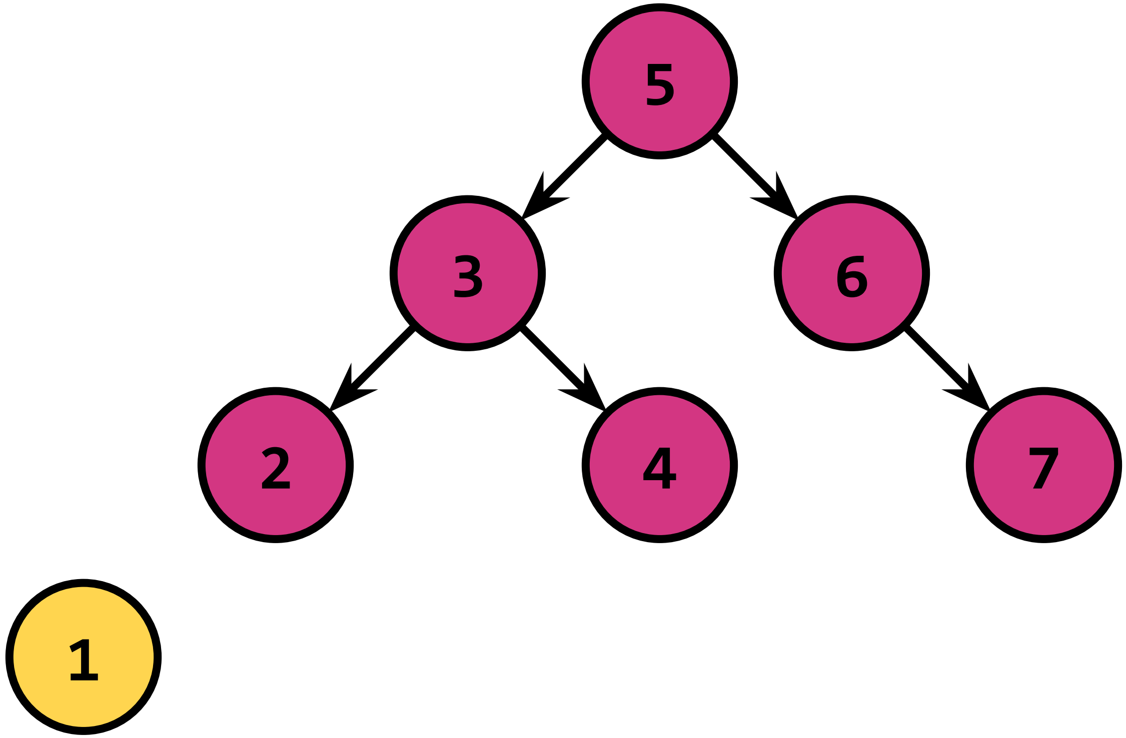
Then, recursively build trees out of its descendants.

Building BSTs by Example



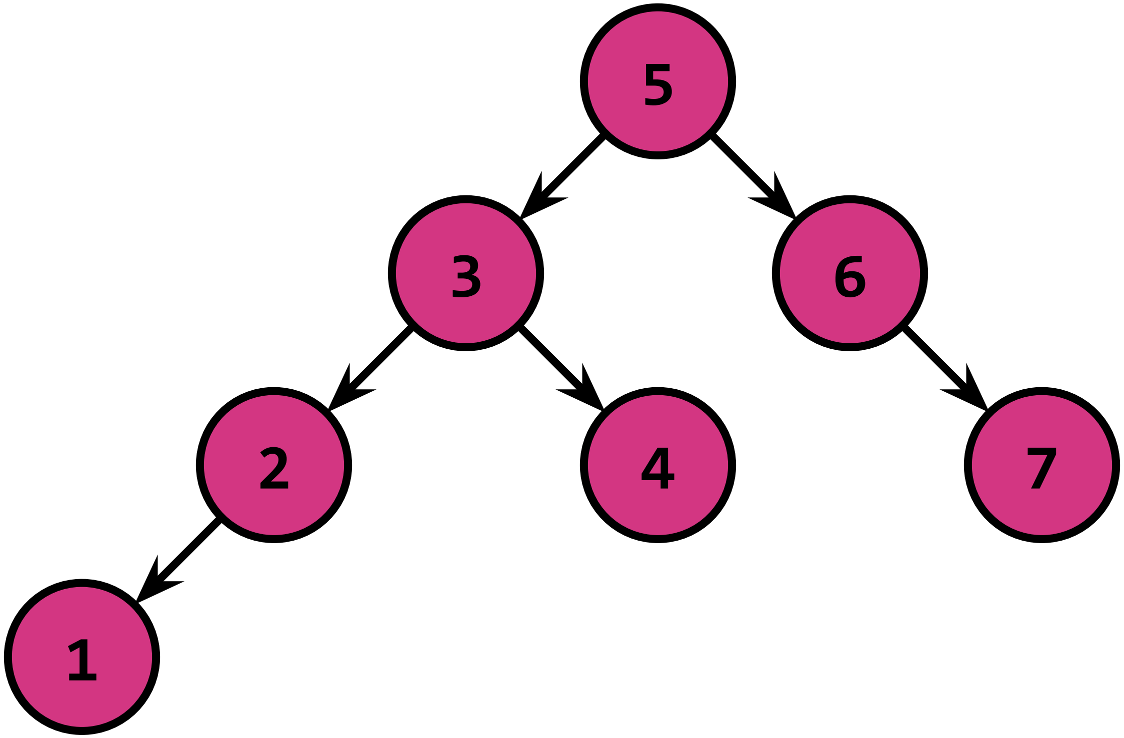
Then, recursively build trees out of its descendants.

Building BSTs by Example



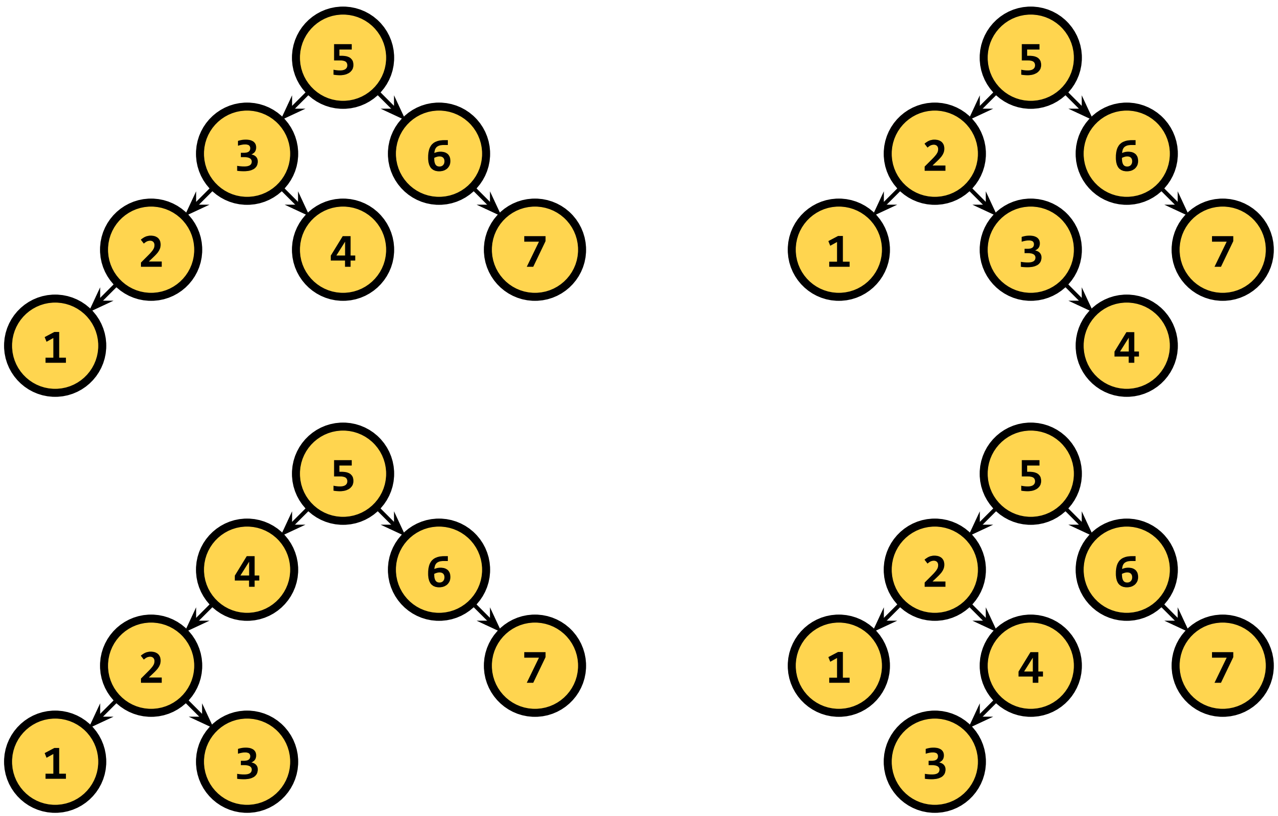
Then, recursively build trees out of its descendants.

Building BSTs by Example



Then, recursively build trees out of its descendants.

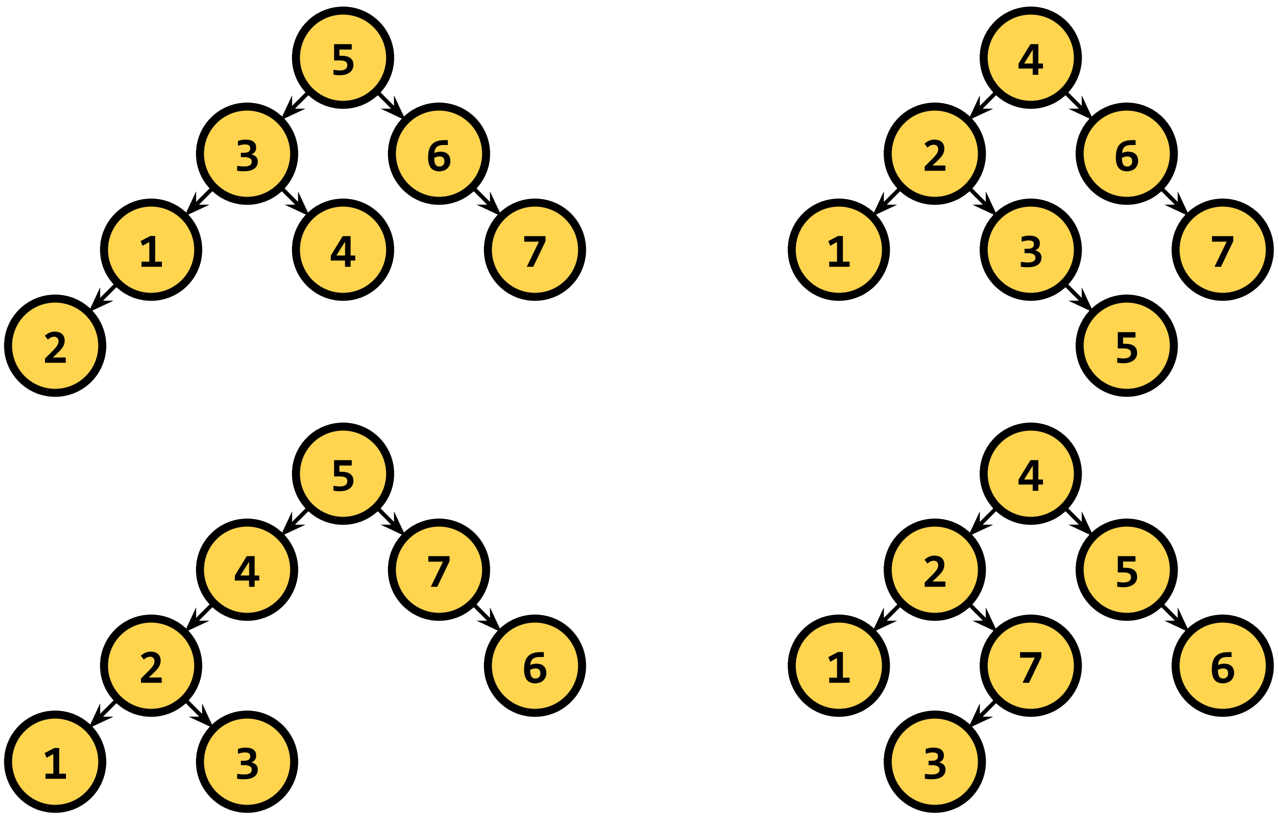
There Exist Many Valid BSTs



… and many more.

How many?

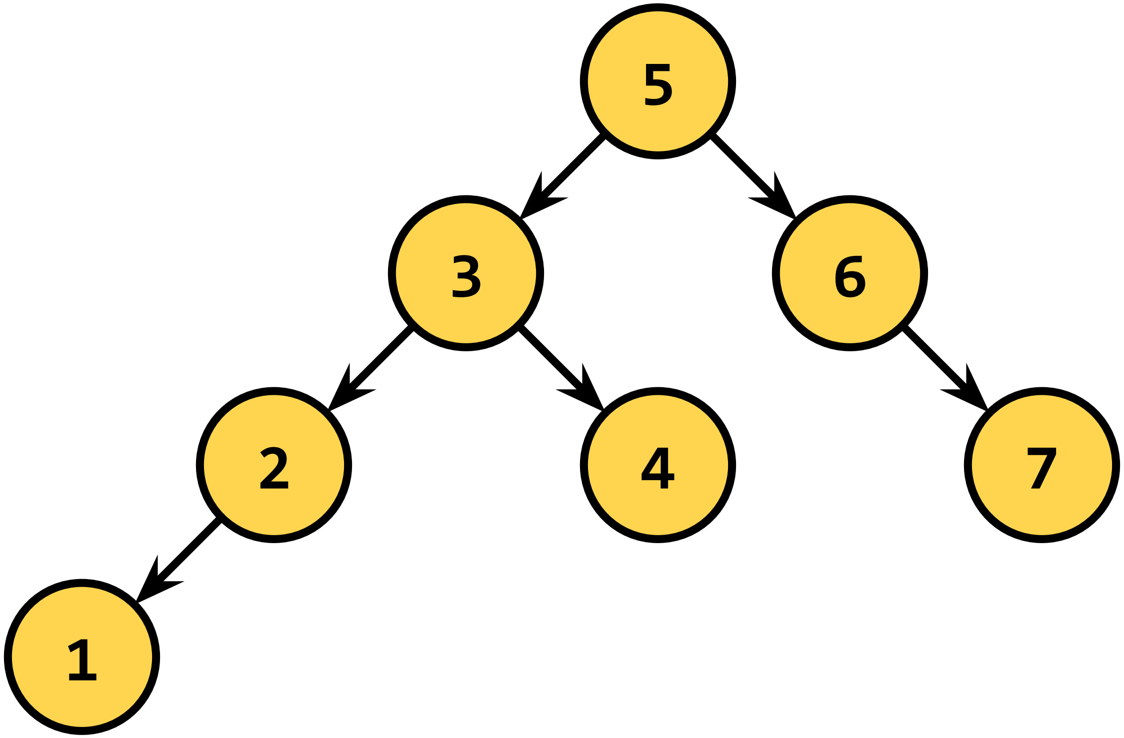
There Exist Many Invalid BSTs



… and many more.

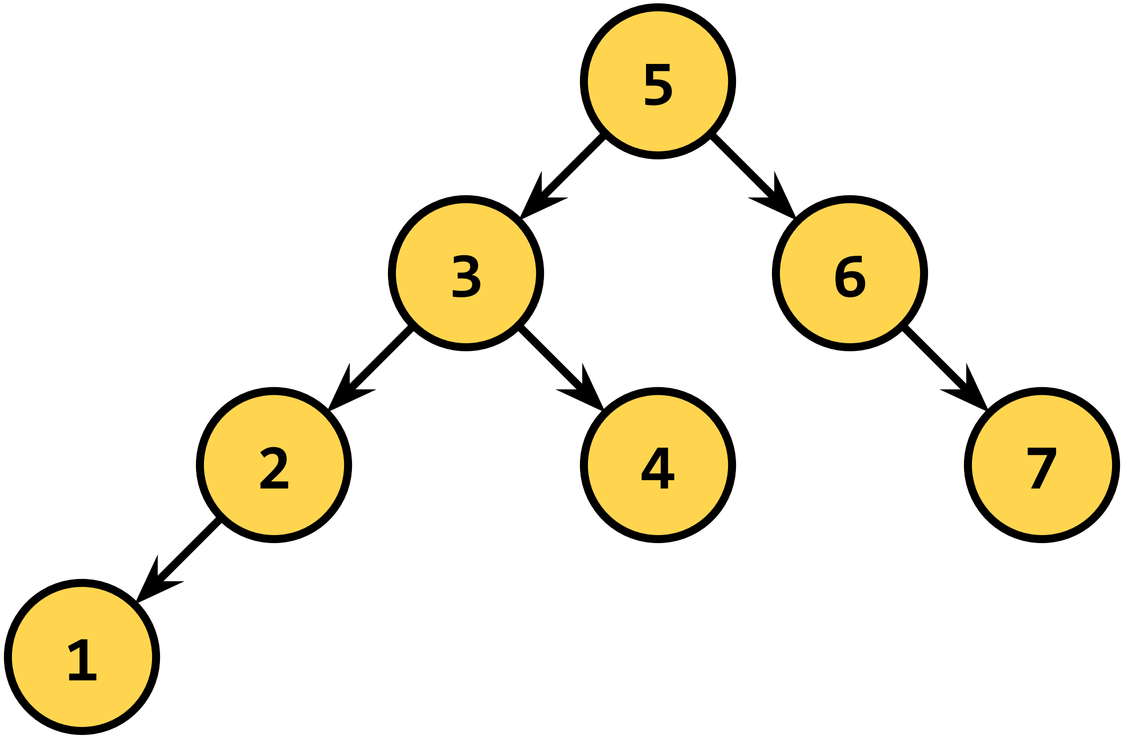
How many?

`search` in BSTs



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

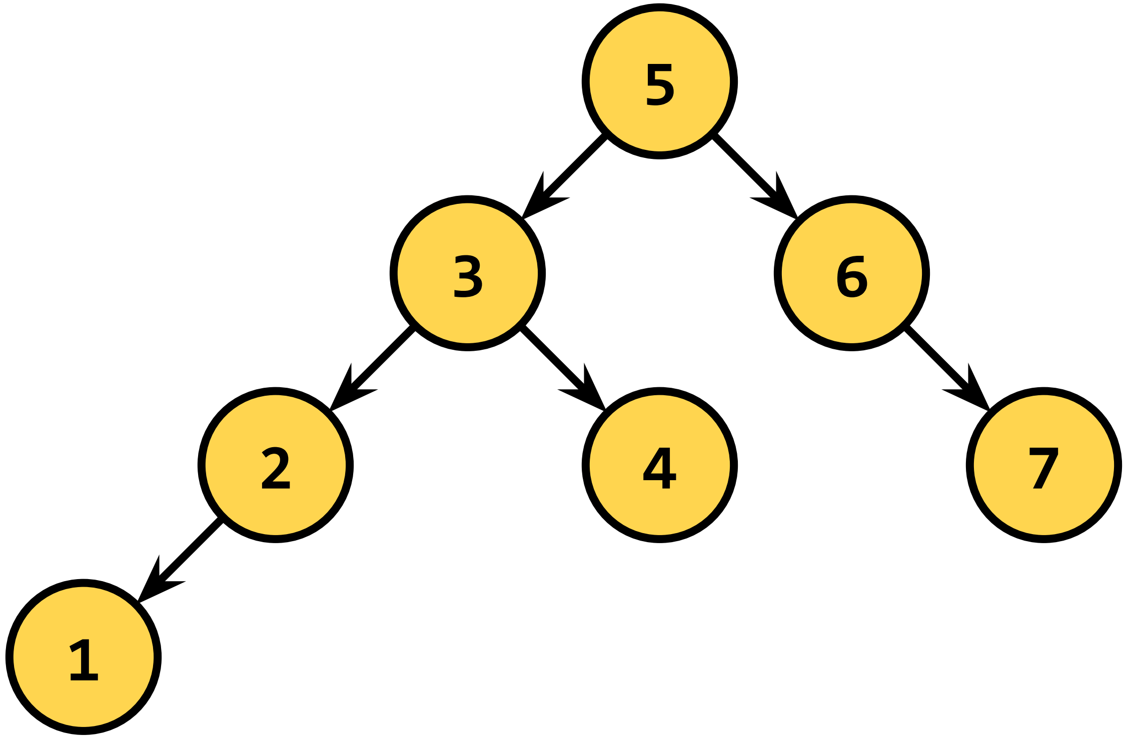
`search` in BSTs



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.

`insert` in BSTs

```
def 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
```

Runtime: `O(log n)` if balanced, `O(n)` otherwise

`delete` in BSTs

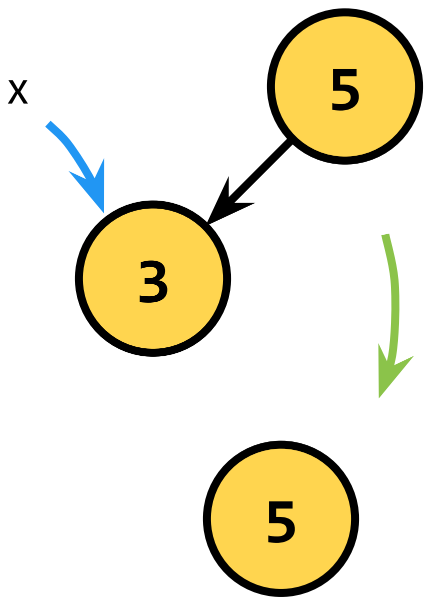
```
def delete(root, key_to_insert):
  x = search(root, key_to_insert)
  if key_to_insert == x.key:
    delete x
```

This is somewhat complicated …

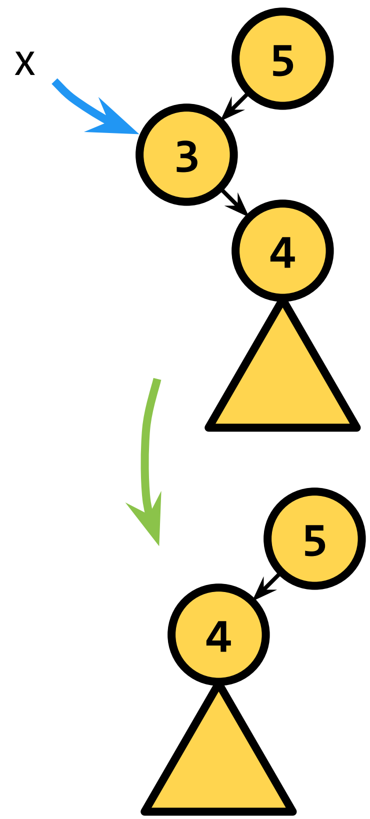
Runtime: `O(log n)` if balanced, `O(n)` otherwise

`delete` in BSTs

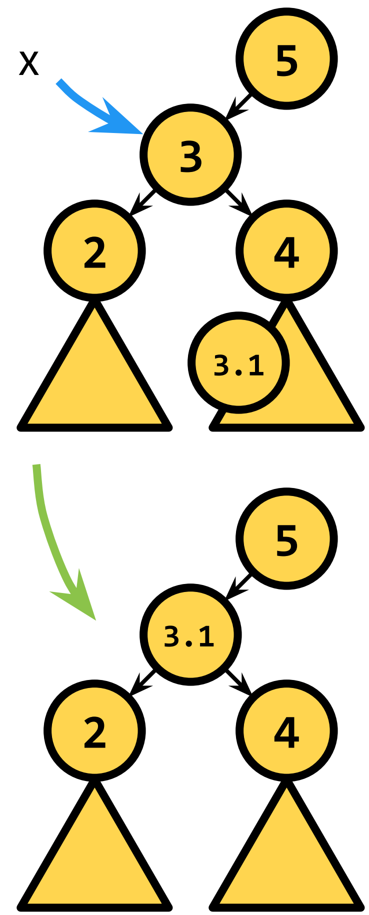
Case 1: x is a leaf
Just delete x



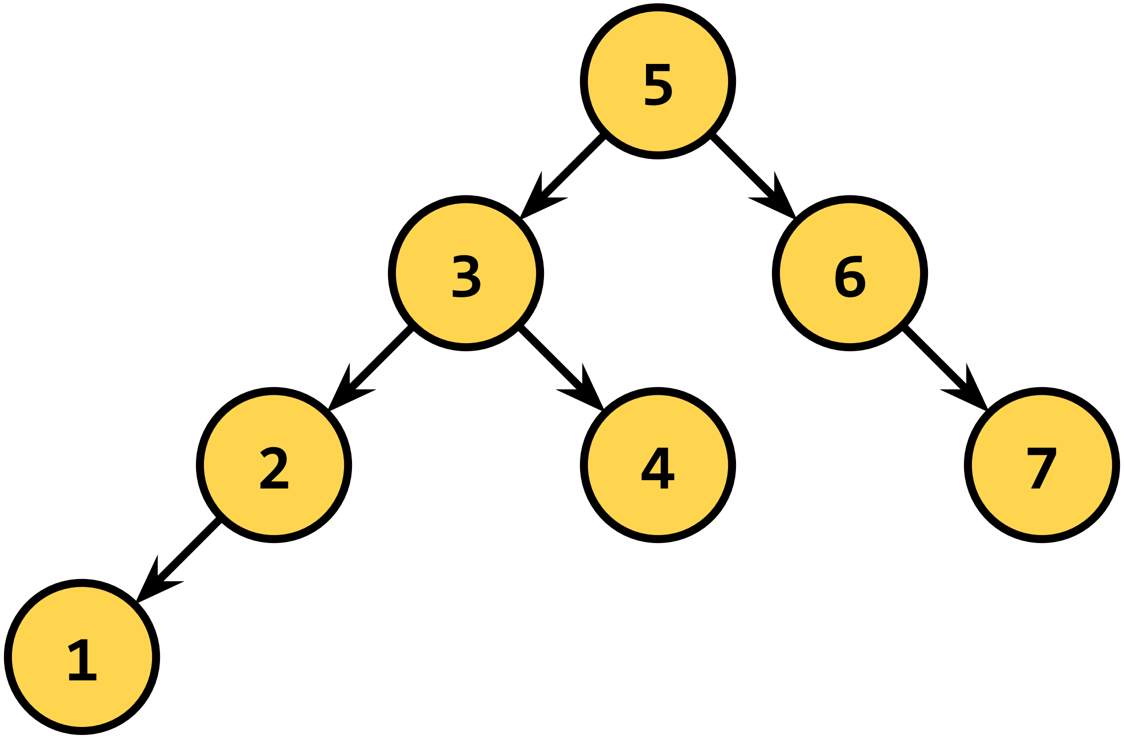
Case 2: x has 1 child
Move its child up



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

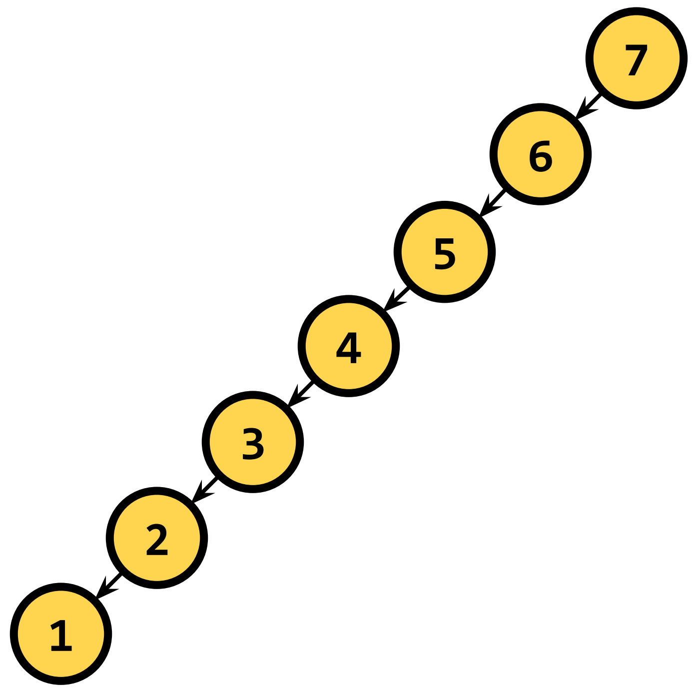


Runtime Analysis



Runtime of `search` (which `insert` and `delete` both call) is
`O(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.

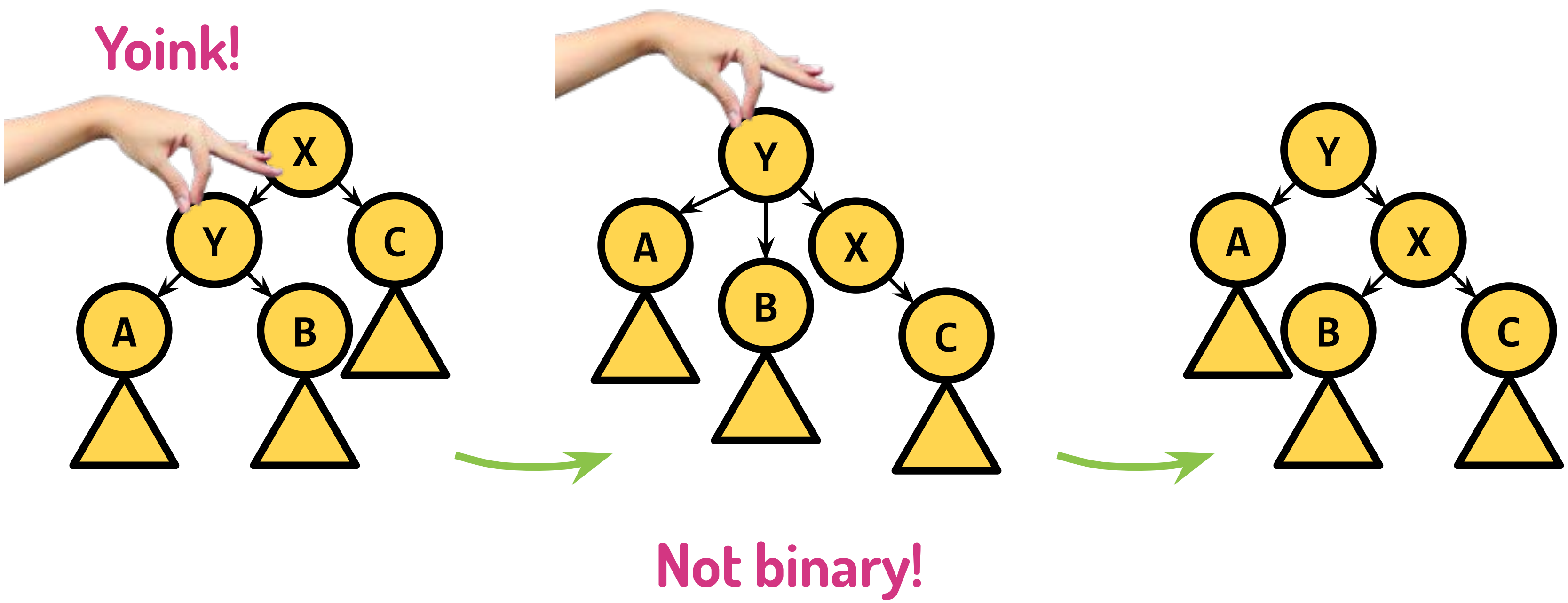
At least $\Omega(n)$ every so often …

In the worst case, how often is “every so often”?

Any other ideas?

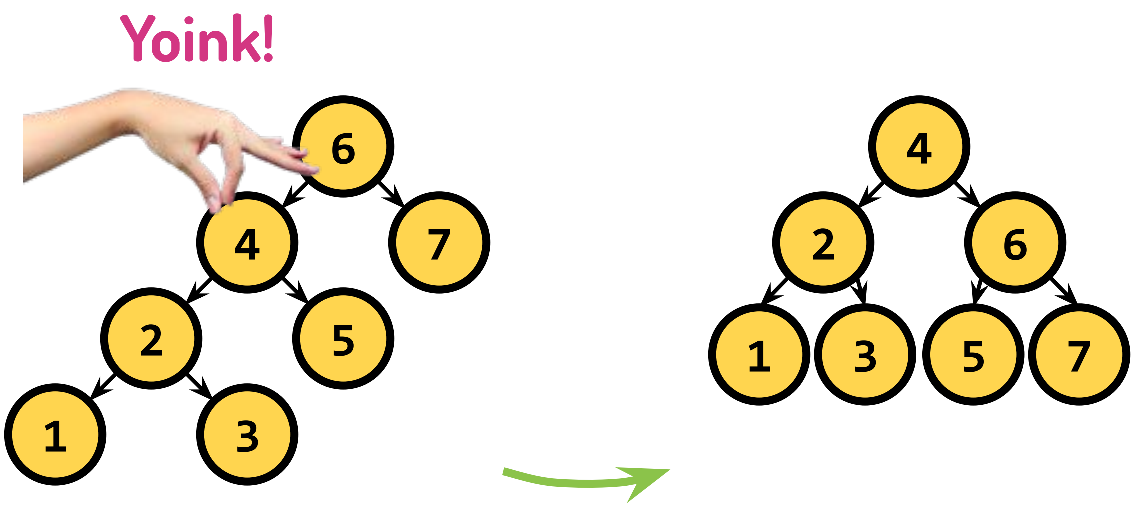
Idea 1: Rotations

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



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.

Instead, let's determine some proxy for balance.

- i.e. If the tree satisfies some property, then it's "pretty balanced."
- We can maintain this property using rotations.

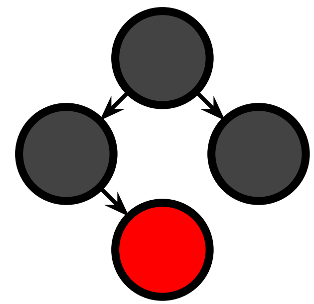
Red-Black Trees

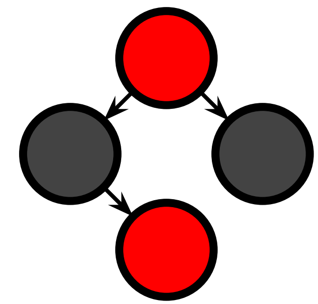
There exist many ways to achieve this proxy for balance, but here we'll study the **red-black tree**.

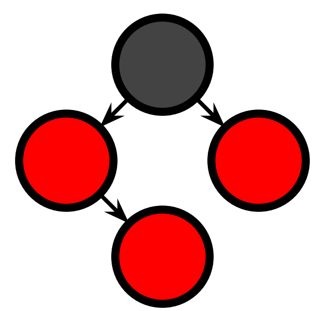
1. Every vertex is colored **red** or **black**.
2. The root vertex is a **black** vertex.
3. A NIL child is a **black** vertex.
4. The child of a **red** vertex must be a **black** vertex.
5. For all vertices v, all paths from v to its NIL descendants have the same number of **black** vertices.

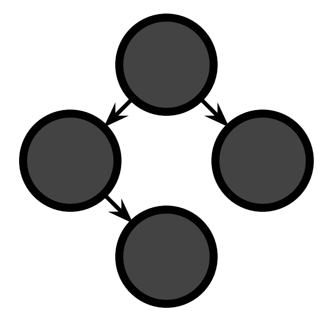
Red-Black Trees by Example

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.



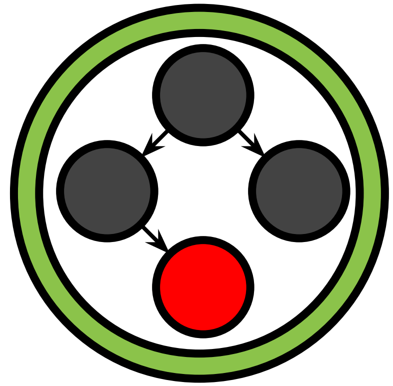


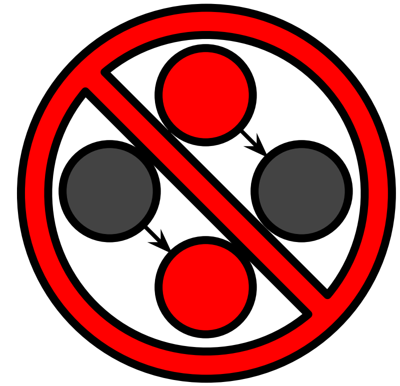




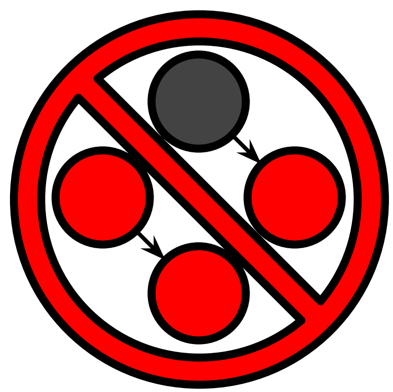
Red-Black Trees by Example

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.

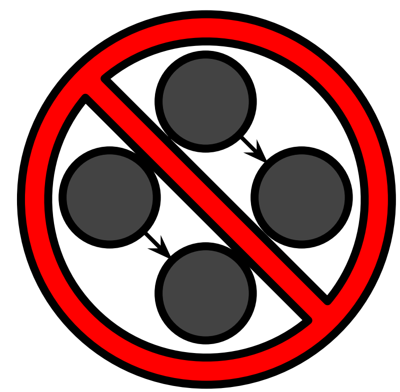




Violates 2



Violates 4



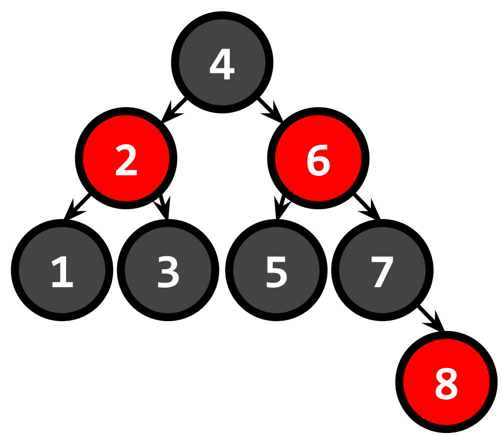
Violates 5

Red-Black Trees

Maintaining these properties maintains a "pretty balanced" BST.

- The **black** vertices are balanced.
- The **red** vertices are "spread out."

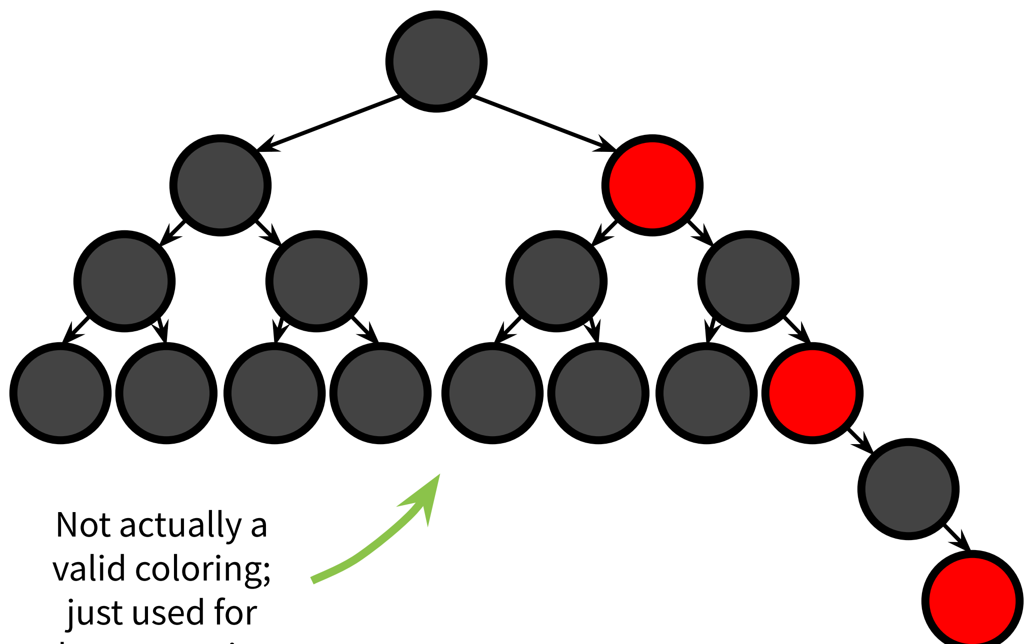
We can maintain this property as we insert/delete vertices, using rotations.



Red-Black Trees

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

One path could be twice as long as the others if we pad it with red vertices.



Not actually a valid coloring; just used for demonstration purposes.

Red-Black Trees

Lemma: The number of non-NIL descendants of x is at least $2^{b(x)} - 1$.

b(x) is the number of black nodes in any path from x to NIL, excluding x.

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.

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

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

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 $\leq 2 \log_2(n+1) = O(\log n)$.

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(x) is at least half of the height. Then $n \geq 2^{b(r)} - 1 \geq 2^{h/2} - 1$, and hence $h \leq 2 \log_2(n+1)$. ■

`insert` in Red-Black Trees

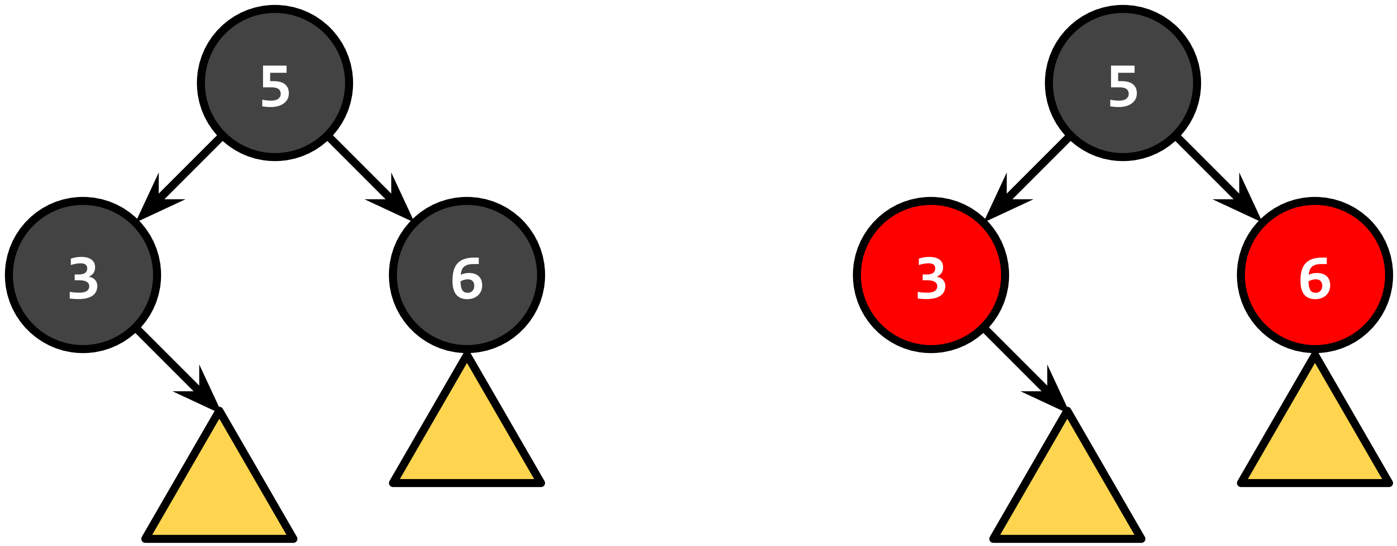
```
def rb_insert(root, key_to_insert):
  x = search(root, key_to_insert)
  v = new red vertex with key_to_insert
  if key_to_insert > x.key:
    x.right = v
    fix things up, if necessary
  if key_to_insert < x.key:
    x.left = v
    fix things up, if necessary
  if key_to_insert == x.key:
    return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

What does that mean?

`insert` in Red-Black Trees

What does “if necessary” mean?

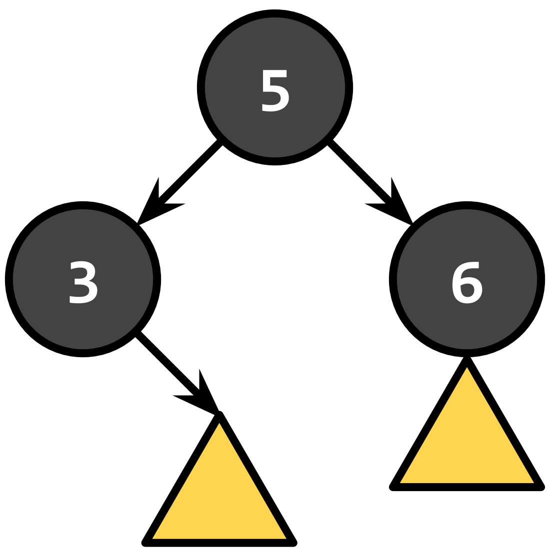
Suppose we want to `insert(1)`.

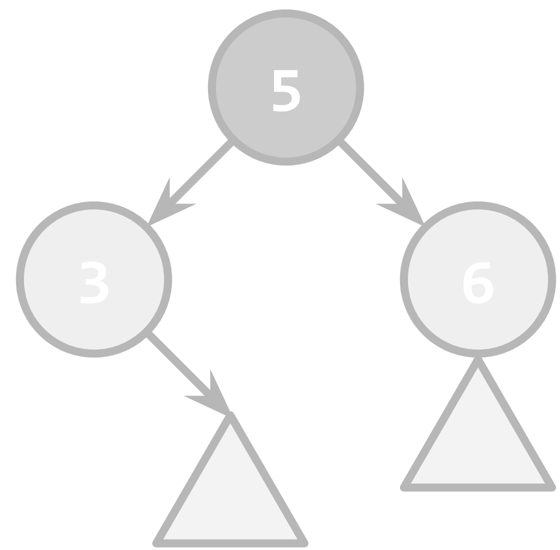


`insert` in Red-Black Trees

What does "if necessary" mean?

Suppose we want to `insert(1)`.

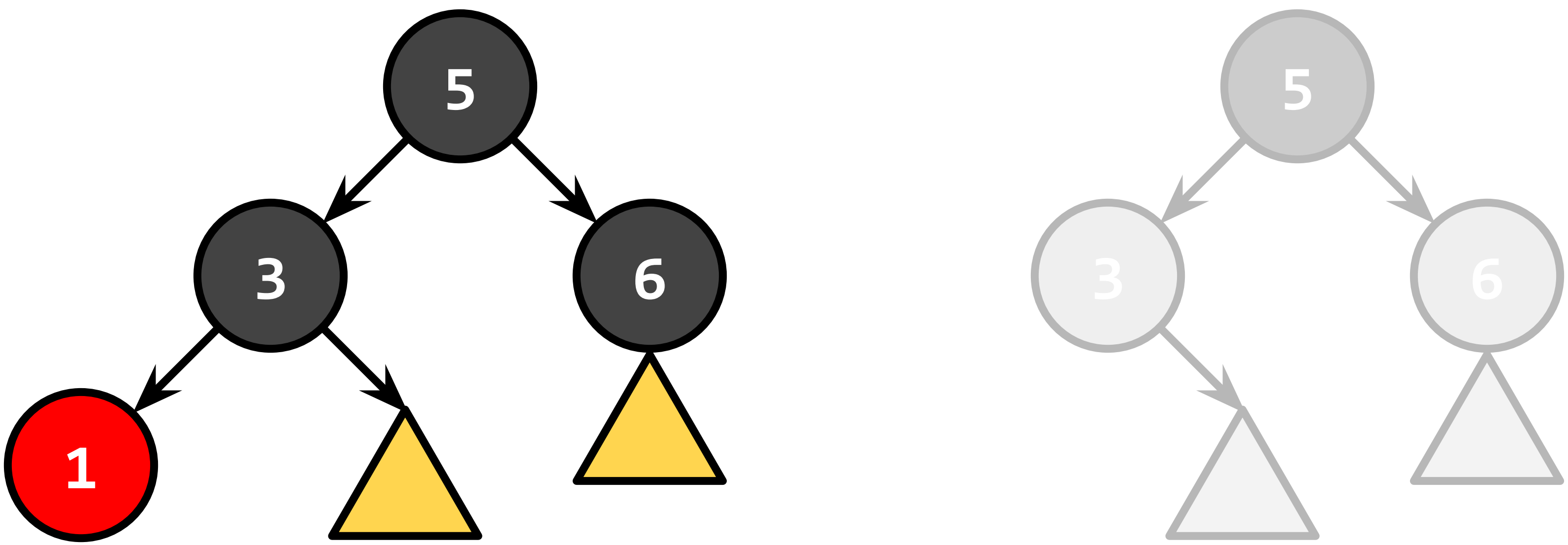




`insert` in Red-Black Trees

What does "if necessary" mean?

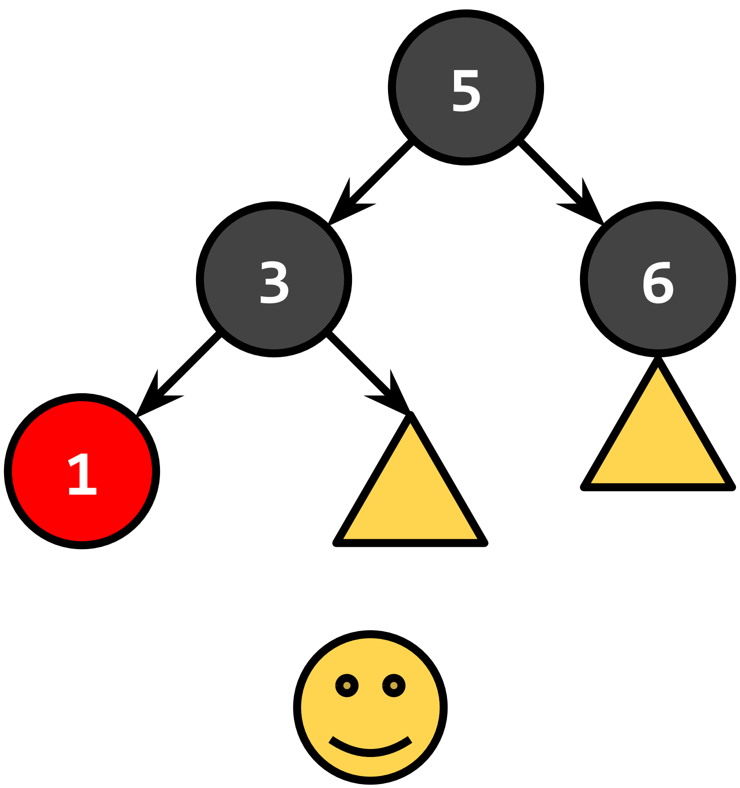
Suppose we want to `insert(1)`.

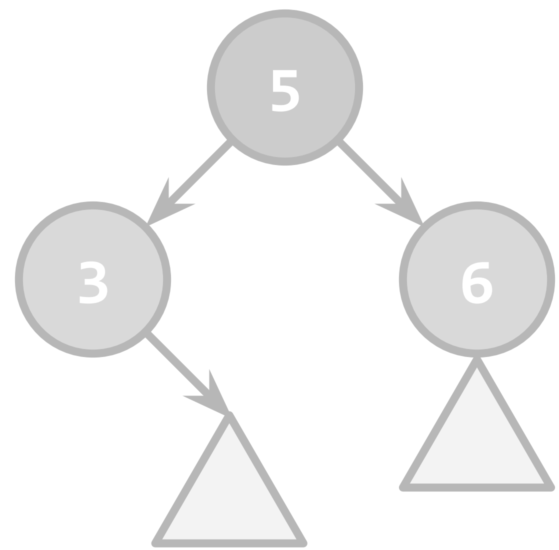


`insert` in Red-Black Trees

What does "if necessary" mean?

Suppose we want to `insert(1)`.

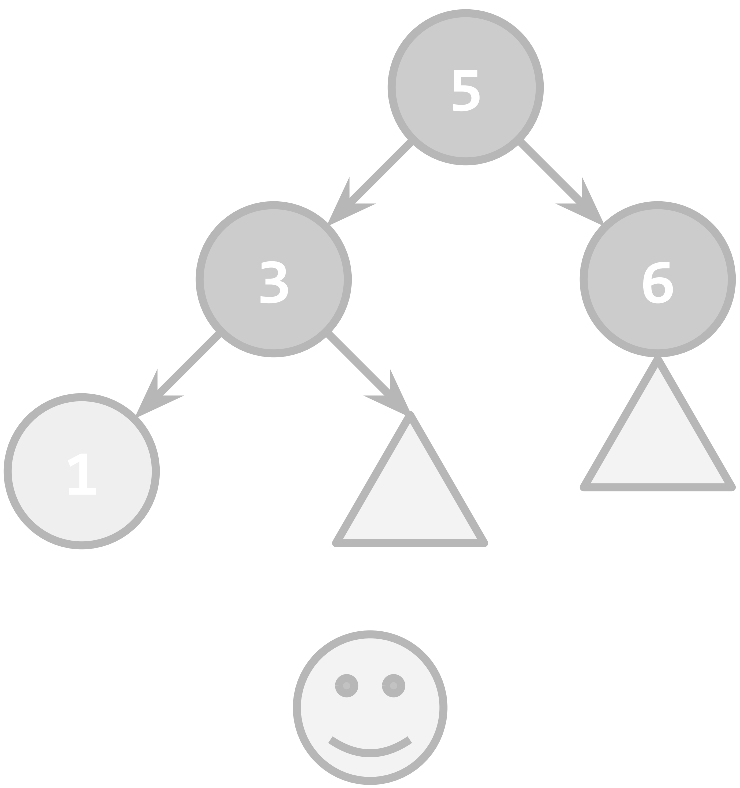


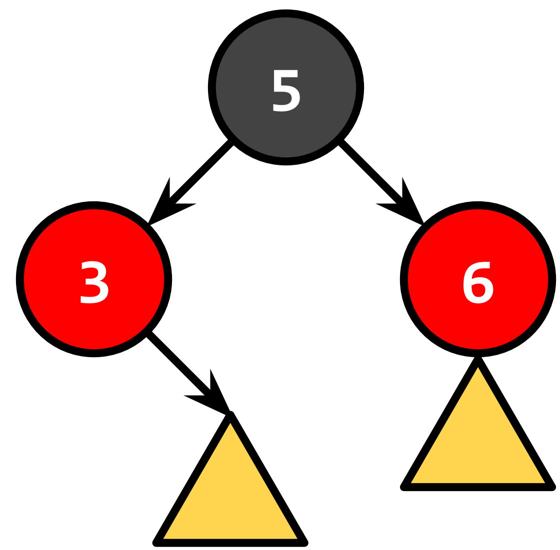


`insert` in Red-Black Trees

What does “if necessary” mean?

Suppose we want to `insert(1)`.

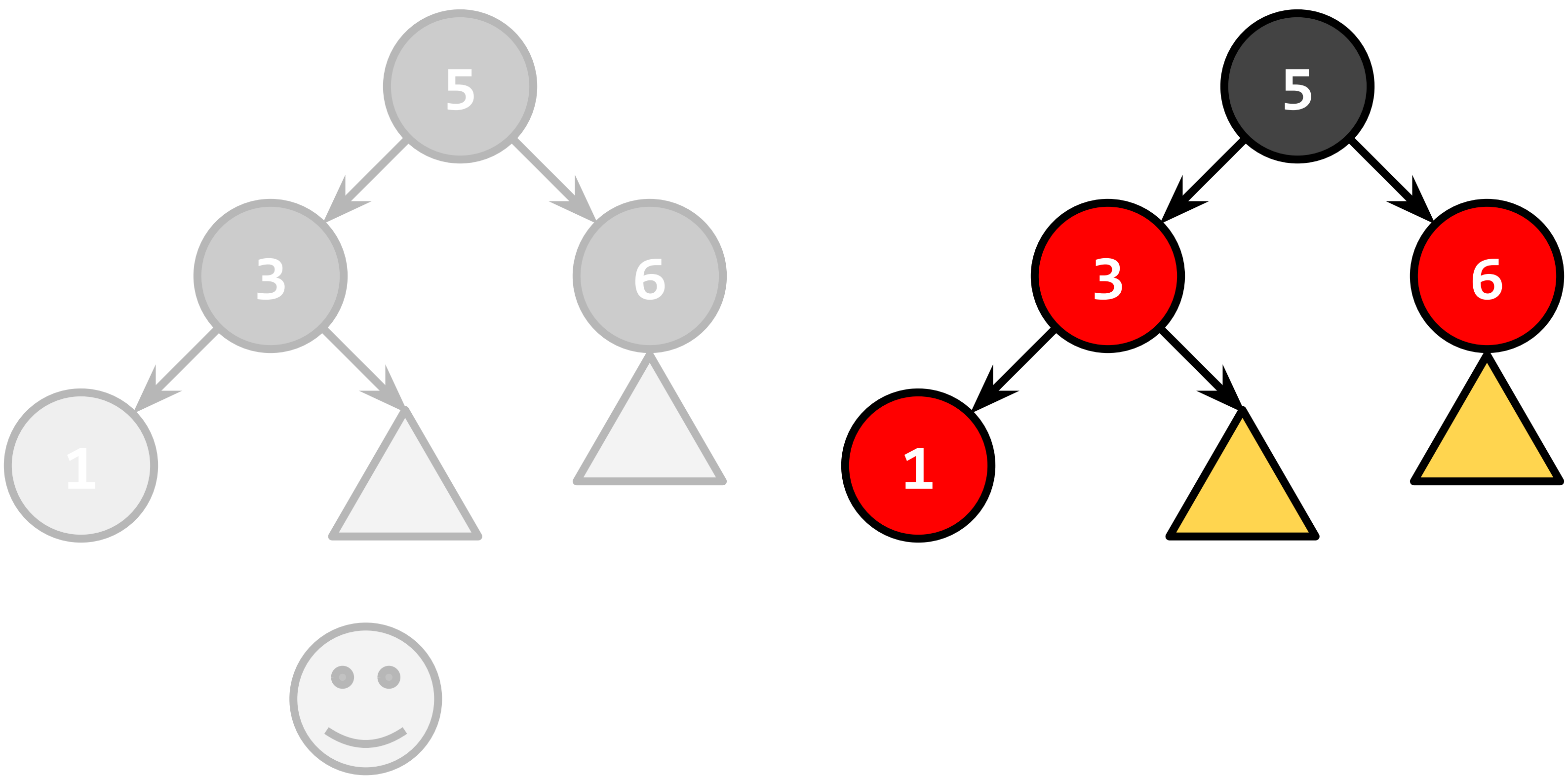




`insert` in Red-Black Trees

What does “if necessary” mean?

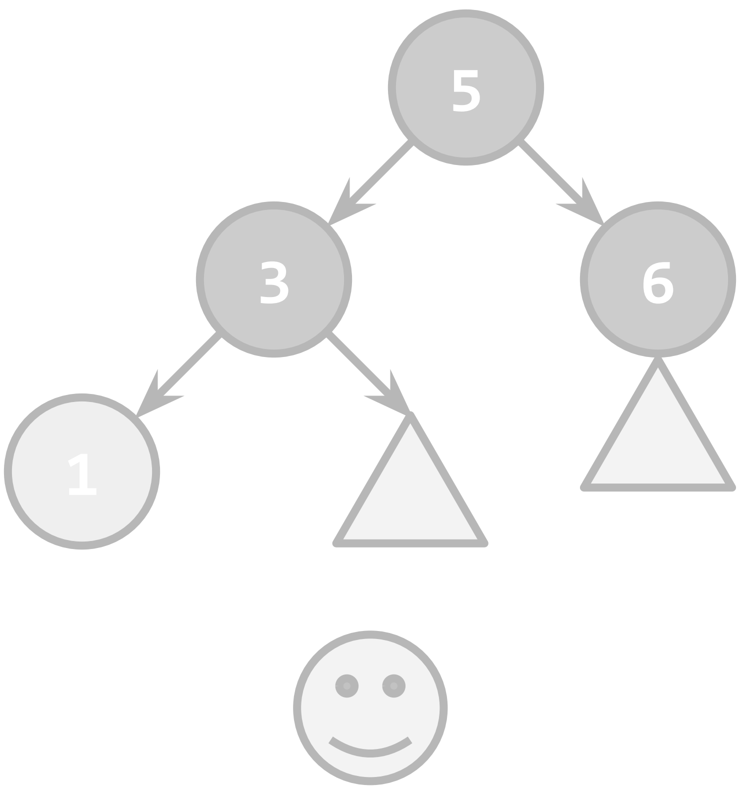
Suppose we want to `insert(1)`.

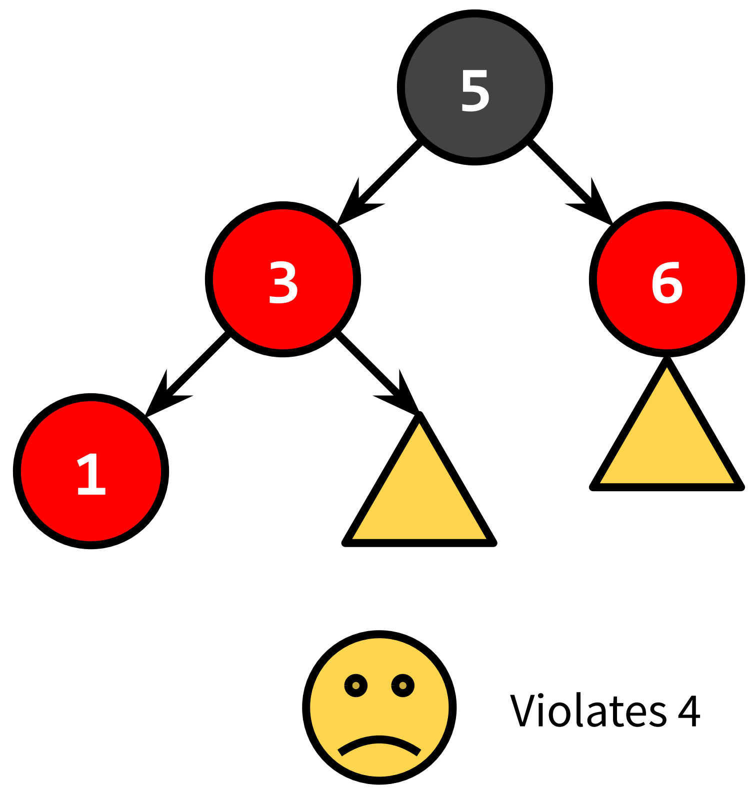


`insert` in Red-Black Trees

What does "if necessary" mean?

Suppose we want to `insert(1)`.

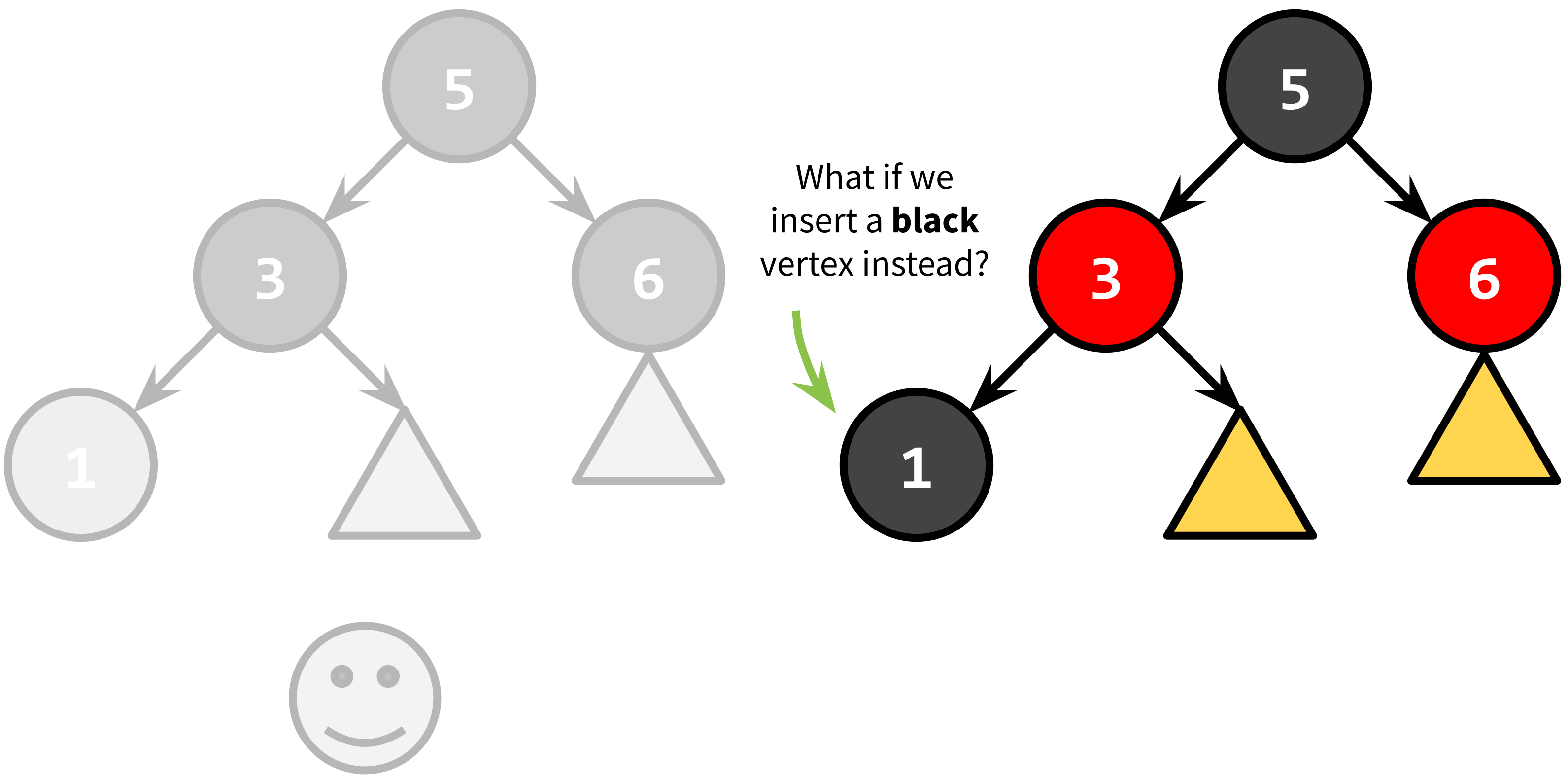




`insert` in Red-Black Trees

What does "if necessary" mean?

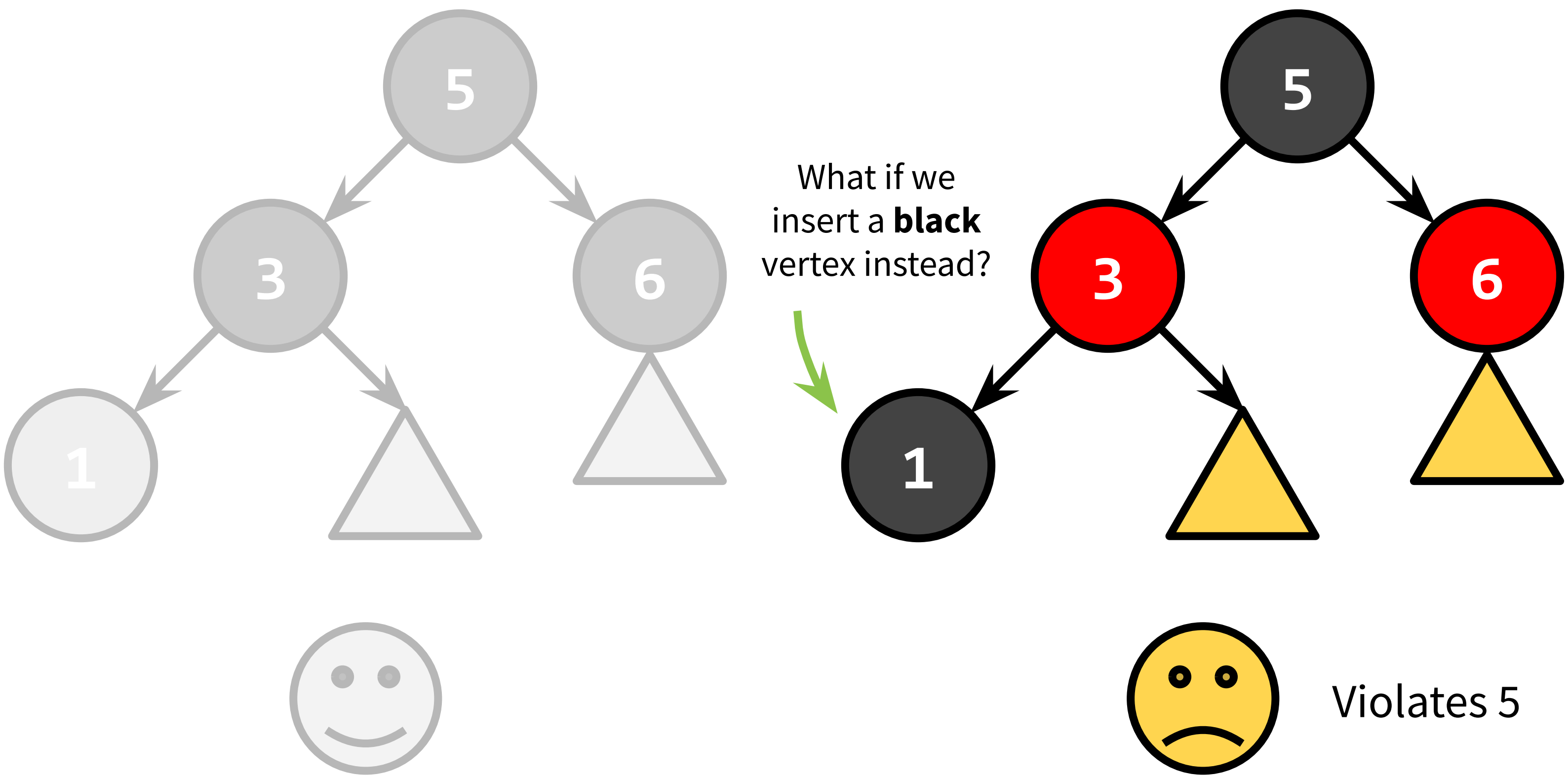
Suppose we want to `insert(1)`.



`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?

- So it seems we’re happy if the parent of the inserted vertex is **black**.
- But there’s an issue if the parent of the inserted vertex is **red**.

`insert` in Red-Black Trees

```
def 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

```
def recolor(v):
  p = parent(x)
  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
    recolor(grand_p)
  else:  # uncle.color == black
    p.color = black
    grand_p.color = red
    right_rotate(grand_p)  # yoink
```

Runtime: `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.

YAY!