Sorting Lower Bounds & Binary Search Trees

Summer 2017 • Lecture 3

A Few Notes

- Homework 1
  - ~~Due tomorrow night at 11:59 p.m. on Gradescope.~~
  - Remember, you must type your solutions!
  - ~~You can use a max of 2 out of your 3 late days.~~
- Homework 2
  - ~~Released tomorrow night~~.
  - ~~Due Friday 7/14 at 11:59 p.m. on Gradescope~~.
- Lecture Notes are from past quarters, for now.

Outline for Today

- Sorting Lower Bounds
  - [Example] Mergesort, revisited
  - Sorting Lower Bounds
    - Comparison-based sorting algorithms
    - Linear-time sorting algorithms
  - Binary Search Trees
    - Red-black trees

Sorting Lower Bounds

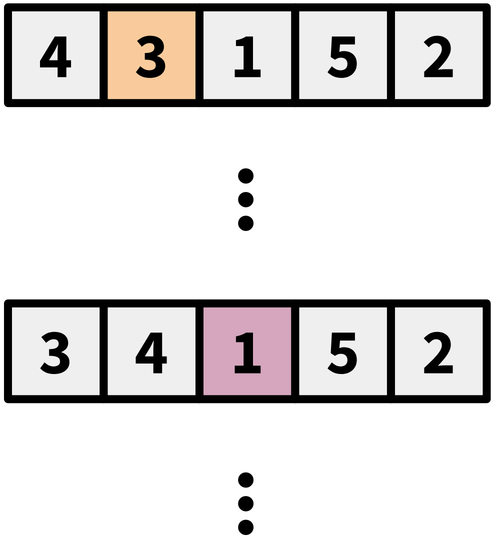
Comparison-Based Sorting

These algorithms use “comparisons” to achieve their output.

`insertion_sort` and `mergesort` are comparison-based sorting algorithms. `select_k` is a comparison-based algorithm. Next week, we’ll learn about a randomized comparison-based sorting algorithm called `quicksort`.

A comparison compares two values. e.g. Is **A[0]** < **A[1]**? Is **A[0]** < **A[4]**?

Recall, insertion sort.



4	3	1	5	2

Is 3 < 4?

⋮

3	4	1	5	2

Is 1 < 4? Is 1 < 3?

⋮

Comparison-Based Sorting

Theorem: Any deterministic comparison-based sorting algorithm requires Ω(n log(n))-time.

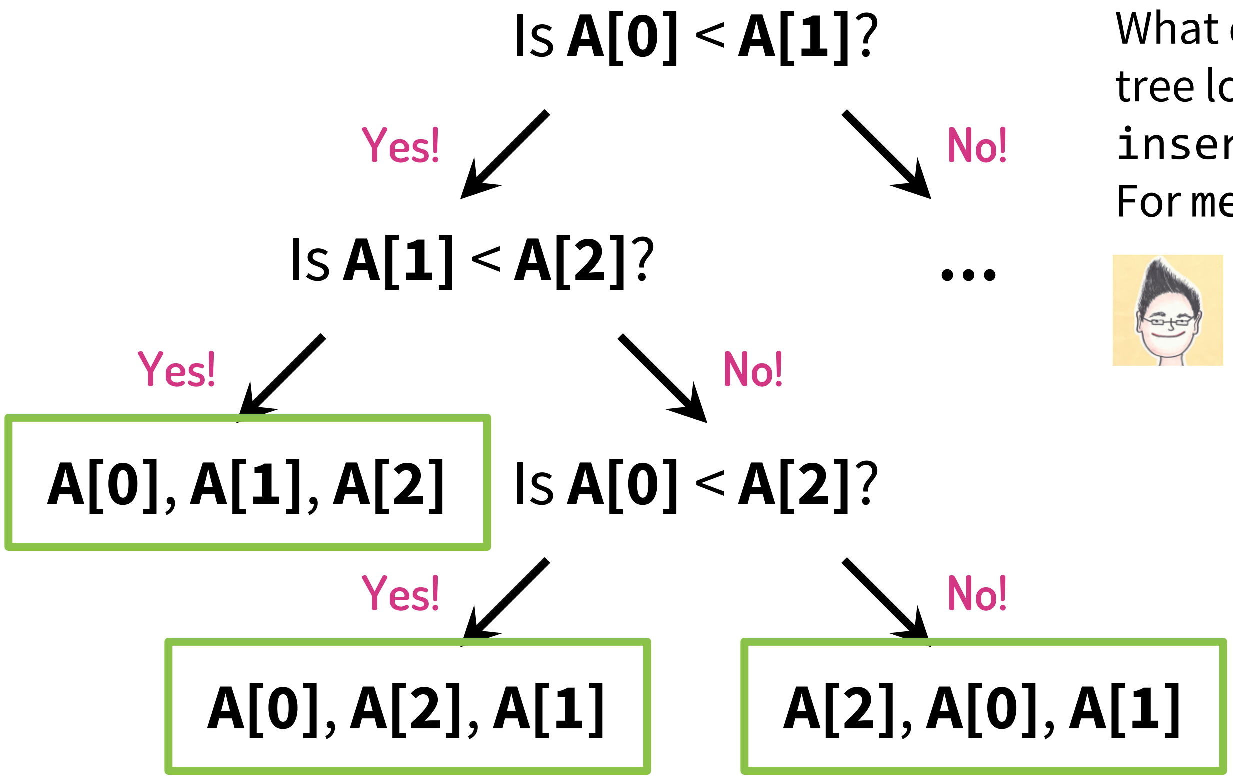
Proof:

Hmm …

Comparison-Based Sorting

We can represent the comparisons made by a comparison-based sorting algorithm as a decision tree.

Suppose we want to sort three items in **A**.



Is **A[0]** < **A[1]**?

Yes!

Is **A[1]** < **A[2]**?

Yes!

A[0], A[1], A[2]

No!

Is **A[0]** < **A[2]**?

Yes!

A[0], A[2], A[1]

No!

A[2], A[0], A[1]

No!

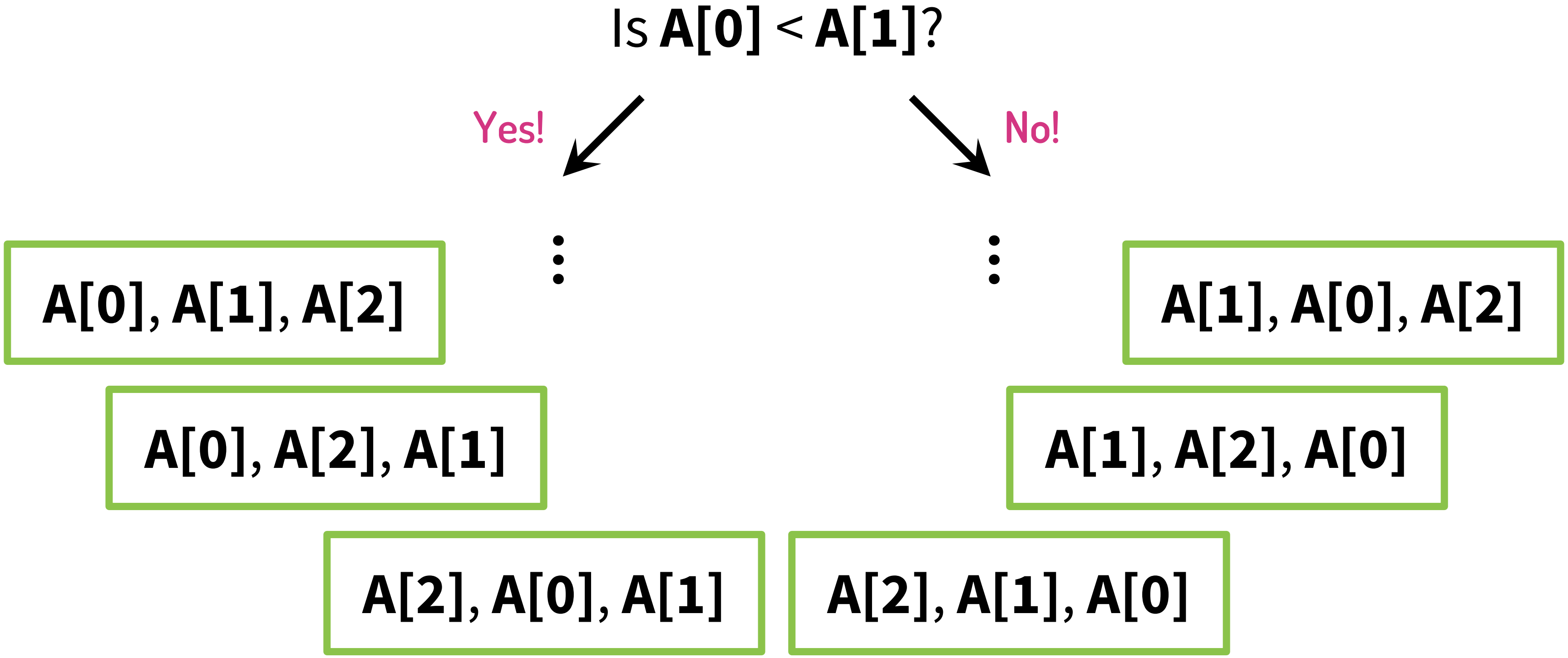
…

What does the decision tree look like for `insertion_sort`? For `mergesort`?

Comparison-Based Sorting

The leaves are all of the possible orderings of the items.

The worst-case runtime must be at least
Ω(length of the longest path).



Is **A[0]** < **A[1]**?

Yes!

No!

A[0], A[1], A[2]

A[0], A[2], A[1]

A[2], A[0], A[1]

A[2], A[1], A[0]

A[1], A[2], A[0]

A[1], A[0], A[2]

Comparison-Based Sorting

How long is the longest path?

- At least how many leaves must this decision tree have?
- What is the depth of the shallowest tree with this many leaves?

Comparison-Based Sorting

How long is the longest path?

- At least how many leaves must this decision tree have? **n!**
- What is the depth of the shallowest tree with this many leaves? **log(n!)**

The longest path is at least log(n!), so the worst-case runtime must be at least **$\Omega(\log(n!))$** = **$\Omega(n \log(n))$**.

Comparison-Based Sorting

Theorem: Any deterministic comparison-based sorting algorithm requires Ω(n log(n))-time.

Proof:

Any deterministic comparison-based sorting algorithm can be represented as a decision tree with n! Leaves.

The worst-case runtime is at least the depth of the decision tree.

All decision trees with n! leaves have depth Ω(n log(n)).

Therefore, any deterministic comparison-based sorting algorithm requires Ω(n log(n))-time

Beyond Comparisons

But then what’s this nonsense about linear-time sorting algorithms?

- We achieve O(n) worst-runtime if we make assumptions on the input.
- e.g. They are integers that range from 0 to k-1.

Counting sort

```
algorithm counting_sort(A, k):
  # A consists of n ints, ranging from
  # 0 to k-1
  counts = [0 * k]  # list of k zeros
  for a_i in A:
    counts[a_i] += 1
  result = []
  for a_i = 0 to length(counts)-1:
    append counts[a_i] a_i's to results
  return results
```

Runtime: O(n+k)

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```

0	**0**	**3**	**1**	**1**	**3**	**1**	**0**

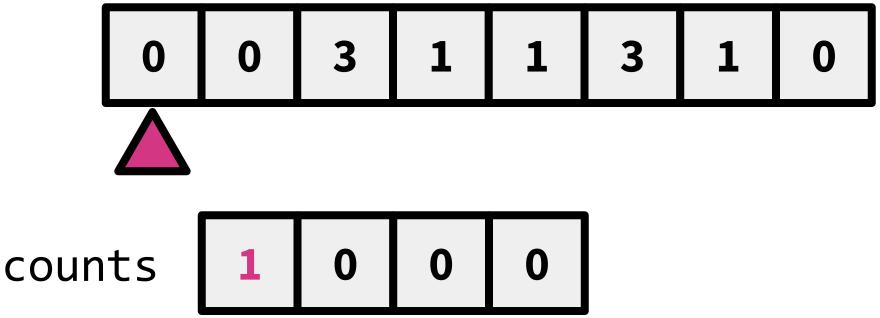
`counts`

0	**0**	**0**	**0**

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



0	0	3	1	1	3	1	0

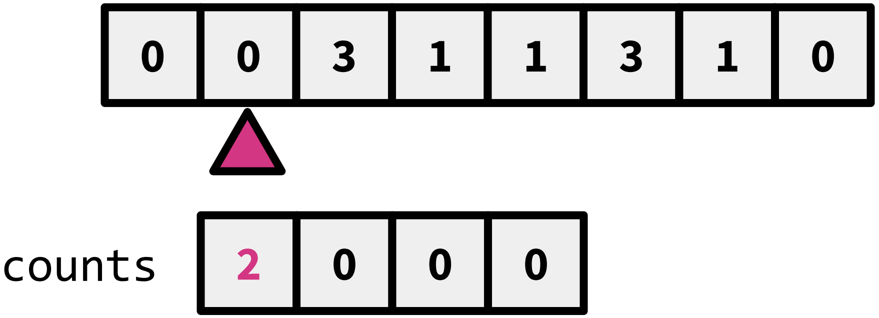
counts

1	0	0	0

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



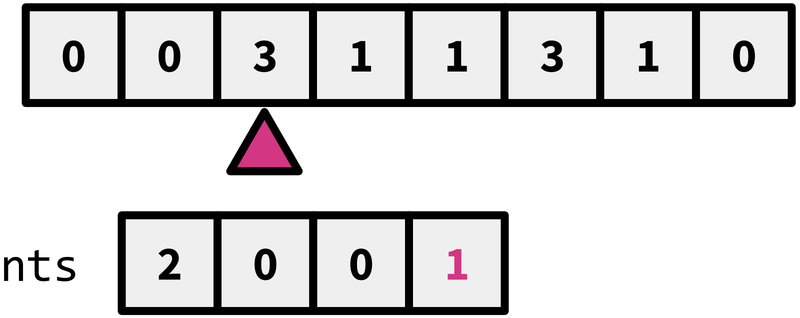
0	0	3	1	1	3	1	0

counts	2	0	0	0

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



0	0	3	1	1	3	1	0

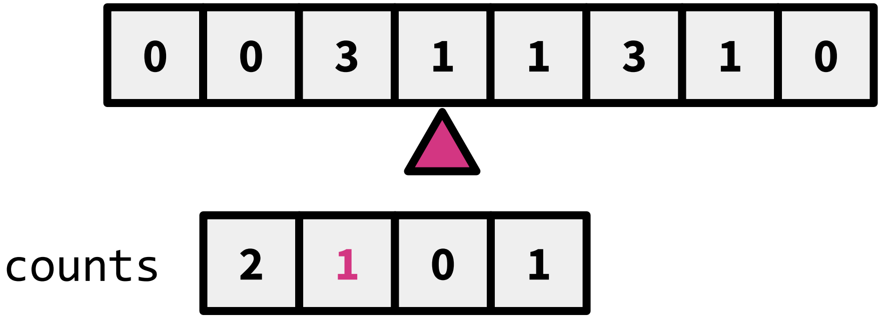
counts

2	0	0	1

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



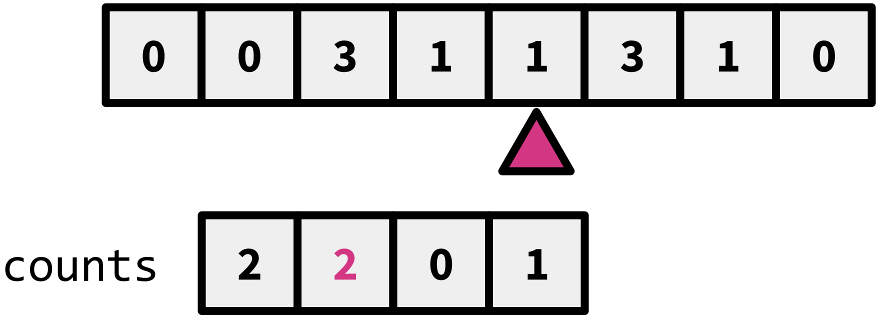
0	0	3	1	1	3	1	0

counts	2	1	0	1

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



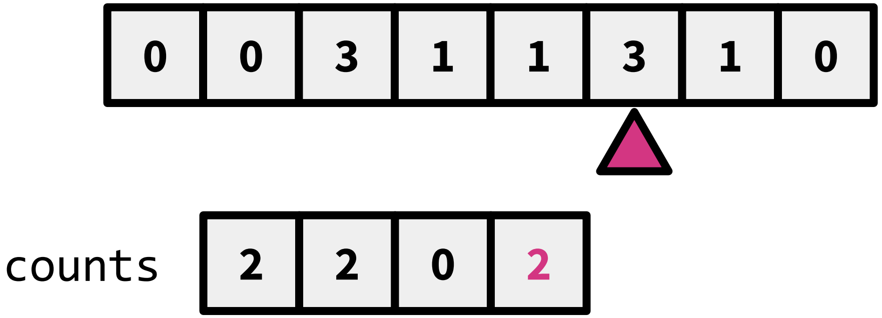
0	0	3	1	1	3	1	0

counts	2	2	0	1

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



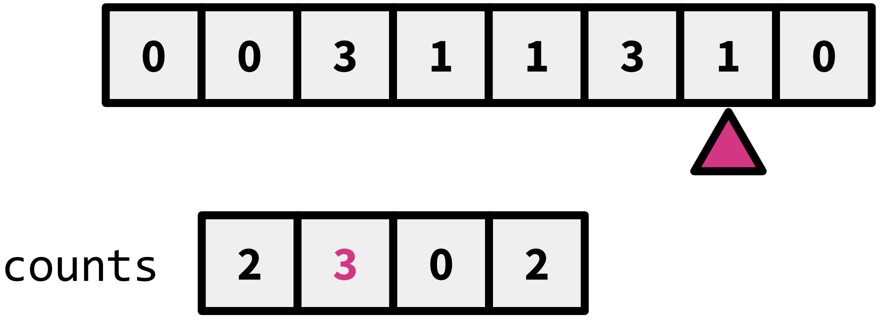
0	0	3	1	1	3	1	0

counts	2	2	0	2

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



0	0	3	1	1	3	1	0

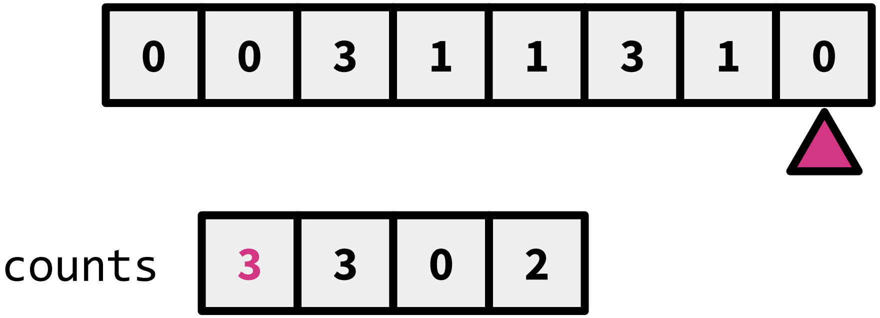
counts

2	3	0	2

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



0	0	3	1	1	3	1	0

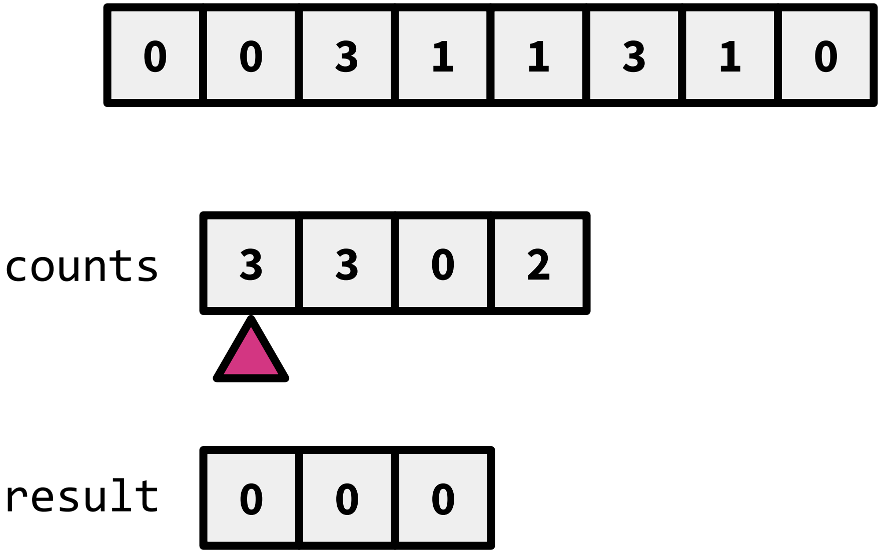
counts

3	3	0	2

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



0	0	3	1	1	3	1	0

counts

3	3	0	2

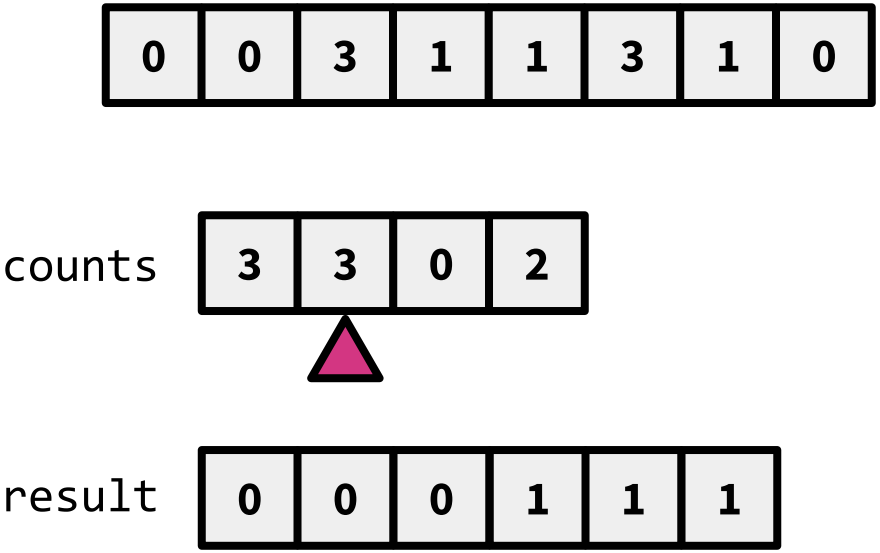
result

0	0	0

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

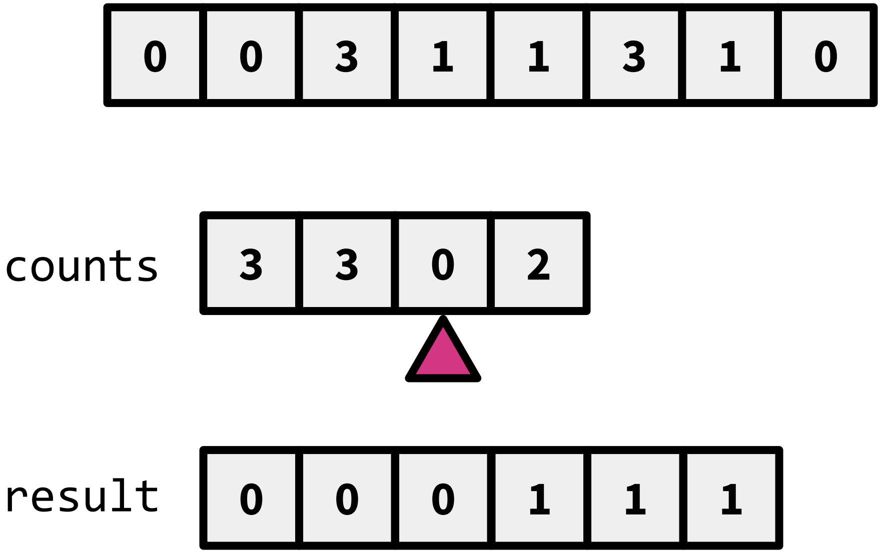
```
counting_sort(A, 4)
```



Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



0	0	3	1	1	3	1	0

counts

3	3	0	2

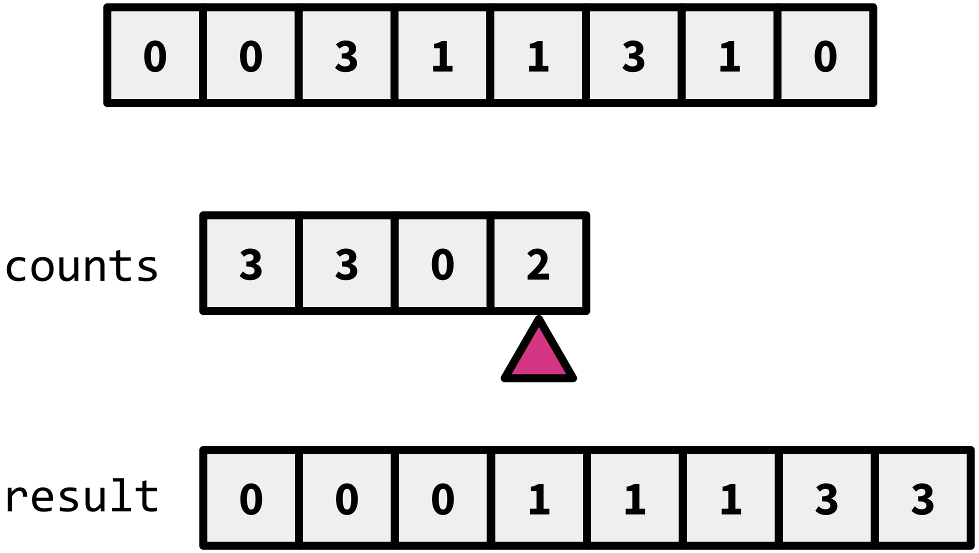
result

0	0	0	1	1	1

Counting sort

Suppose **A** consists of 8 ints ranging from 0 to 3.

```
counting_sort(A, 4)
```



0	0	3	1	1	3	1	0

counts

3	3	0	2

result

0	0	0	1	1	1	3	3

Bucket sort

```
algorithm bucket_sort(A, k, num_buckets):
  # A consists of n (key, value) pairs,
  # with keys ranging from 0 to k-1
  buckets = [[] * num_buckets]
  for key, value in A:
    buckets[get_bucket(key)].append((key, value))
  if num_buckets < k:
    for bucket in buckets:
      stable_sort(bucket) by their keys
  result = concatenate buckets by their values
  return result
```

Runtime: O(n+k) or O(nlogn)

Only guaranteed if
num_buckets >= k

Bucket sort

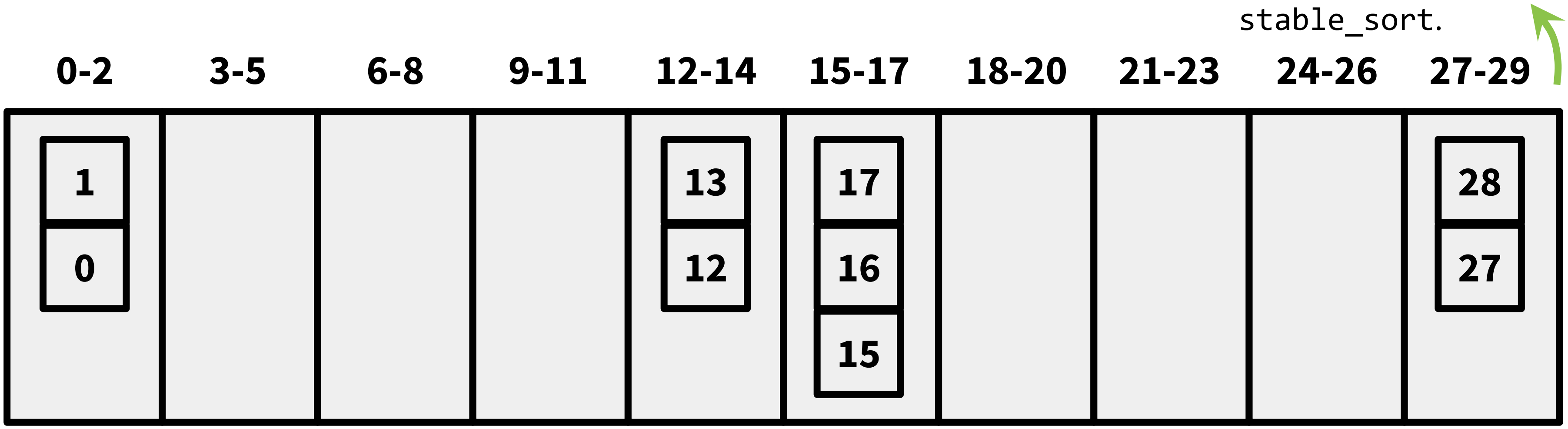
Two cases for k and num_buckets in `bucket_sort`:

(1) k ≤ num_buckets: At most one key per bucket, so buckets do not require an additional `stable_sort` to be sorted (similar to `counting_sort`).

(2) k > num_buckets: Maybe multiple keys per bucket, so buckets require an additional `stable_sort` to be sorted.

Suppose k = 30 and num_buckets = 10. Then we group keys 0 to 2 in the same bucket, 3 to 5 in the same bucket, etc.

A= [17, 13, 16, 12, 15, 1, 28, 0, 27] produces:



Only the keys in the (key, value) pairs are shown here, and all of the buckets require `stable_sort`.

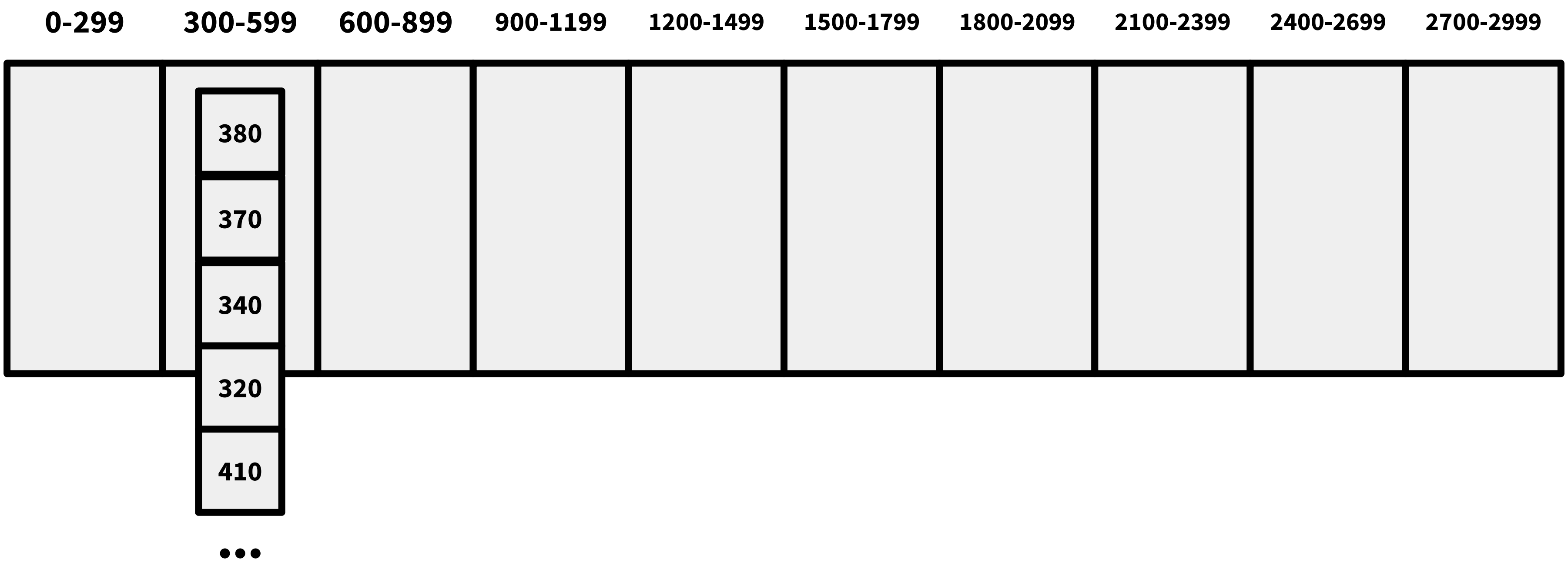
Bucket sort, case (2)

Why `O(nlogn)` in case (2)?

With multiple keys per bucket, a bucket might receive all of the inserted keys.

Suppose the `bucket_sort` caller specifies k = 3000 and num_buckets = 10, but then inserts elements all from the same bucket.

A = [380, 370, 340, 320, 410, …] would need to `stable_sort` all of the elements in the original list since they all fall in the same bucket.



Radix sort

```
algorithm radix_sort(A, d, k):
  # A consists of n d-digit ints, with
  # digits ranging 0 -> k-1
  for j = 0 to d-1:
    A_j = A converted to (key, value) pairs, where
          key is the jth digit of value
    result = bucket_sort(A_j, k, k)
    A = result
  return A
```

Runtime: O(d(n+k))

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```

A	31	5	210	14	95	477	555	125

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

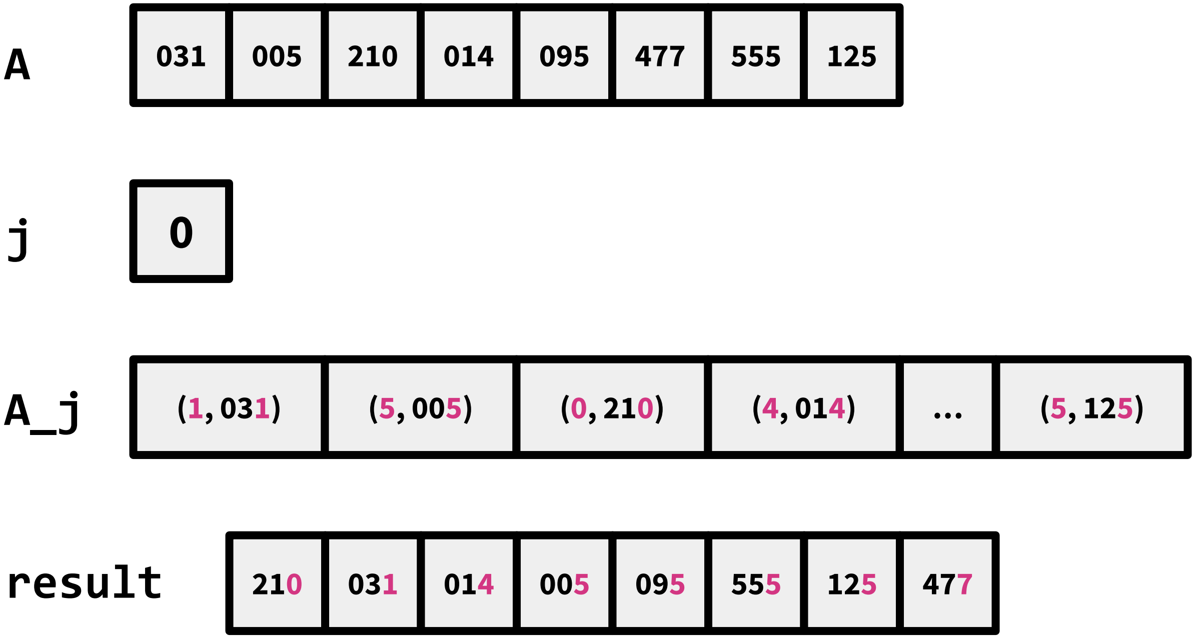
```
radix_sort(A, 3, 10)
```

A	031	005	210	014	095	477	555	125

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```



A: 031 005 210 014 095 477 555 125

j: 0

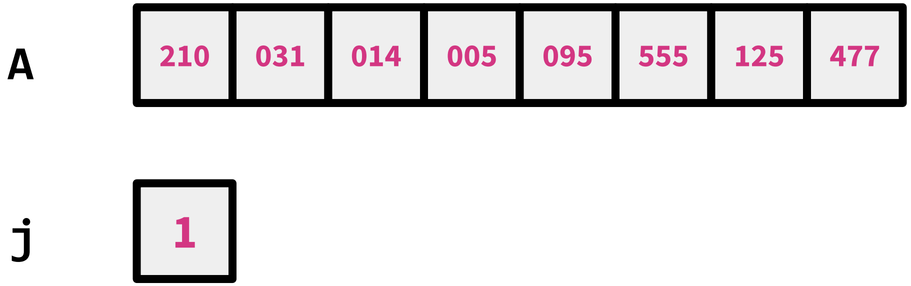
A_j: (1, 031) (5, 005) (0, 210) (4, 014) ... (5, 125)

result: 210 031 014 005 095 555 125 477

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```

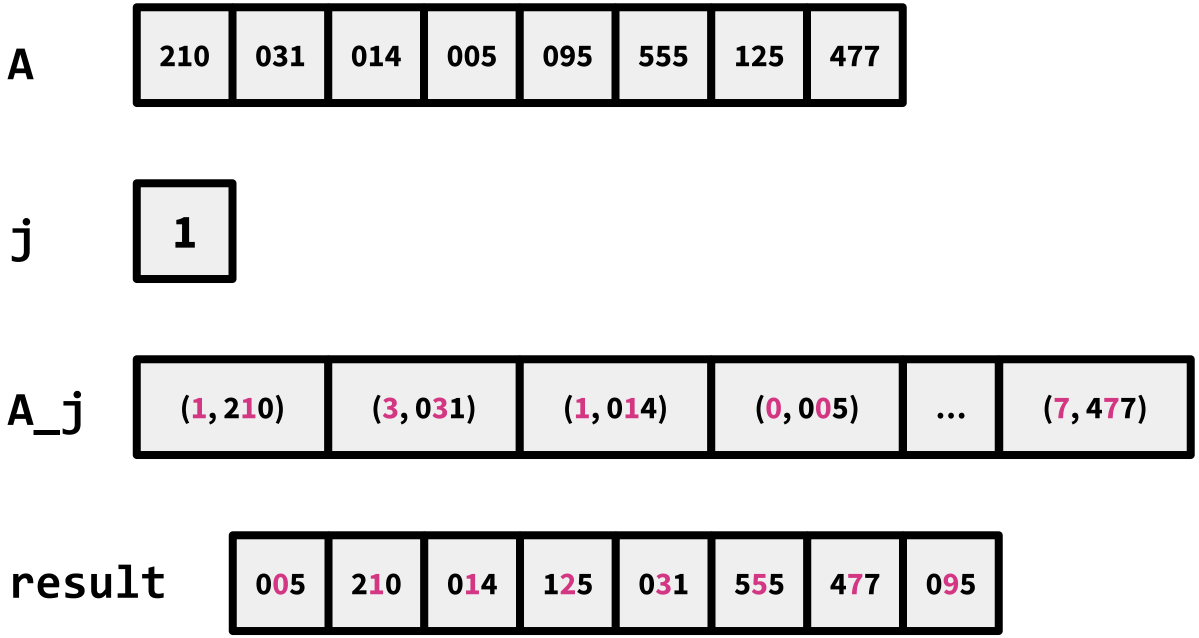


A	210	031	014	005	095	555	125	477
j	1							

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```



A							
210	031	014	005	095	555	125	477

j
1

A_j					
(1, 210)	(3, 031)	(1, 014)	(0, 005)	...	(7, 477)

result							
005	210	014	125	031	555	477	095

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

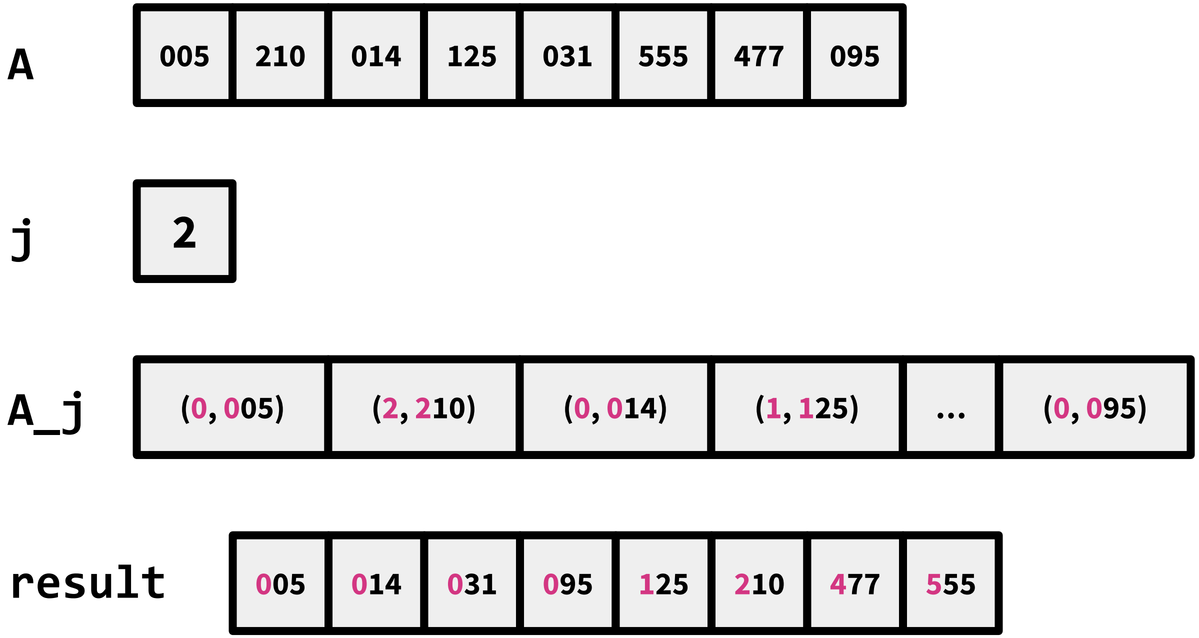
```
radix_sort(A, 3, 10)
```

A	005	210	014	125	031	555	477	095
j	2							

Radix sort

Suppose **A** consists of 8 3-digit ints, with digits ranging from 0 to 9.

```
radix_sort(A, 3, 10)
```



A 005 210 014 125 031 555 477 095

j 2

A_j (0, 005) (2, 210) (0, 014) (1, 125) … (0, 095)

result 005 014 031 095 125 210 477 555

Radix sort

Lemma: If **A** is sorted by its x least-significant digits at the start of iteration j = x of the loop, then **A** will be sorted by its x+1 least-significant digits at the start of iteration j = x+1 of the loop.

Proof:

Since `bucket_sort` is stable, the elements within each bucket are still sorted by their x least-significant digits. `bucket_sort` sorts **A** by the x+1 digit of the elements, so the elements are sorted by their x+1 least-significant digits. □

Radix sort

Theorem: Radix sort sorts the input list.

Proof:

At the start of the first iteration of the loop, **A** is trivially sorted by its 0 least-significant digits.

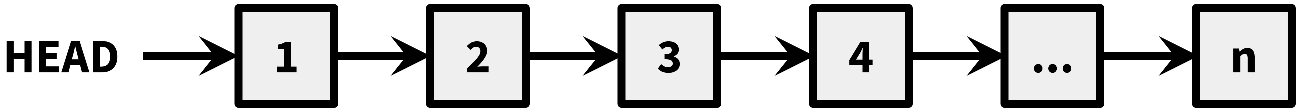
By our lemma, if **A** is sorted by its x least-significant digits at the start of iteration j = x of the loop, then **A** will be sorted by its x+1 least-significant digits at the start of iteration j = x+1 of the loop.

The loop terminates at the start of iteration j = d. The collection of d-digit integers in **A** are sorted by their d least-significant digits, which implies that **A** is sorted when the loop ends. □

3 min break

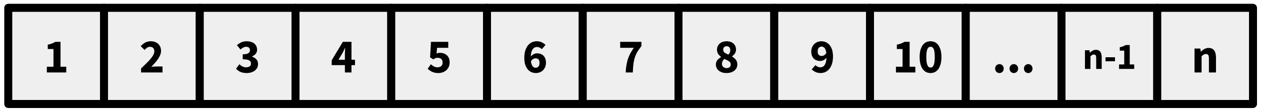
Binary Search Trees

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



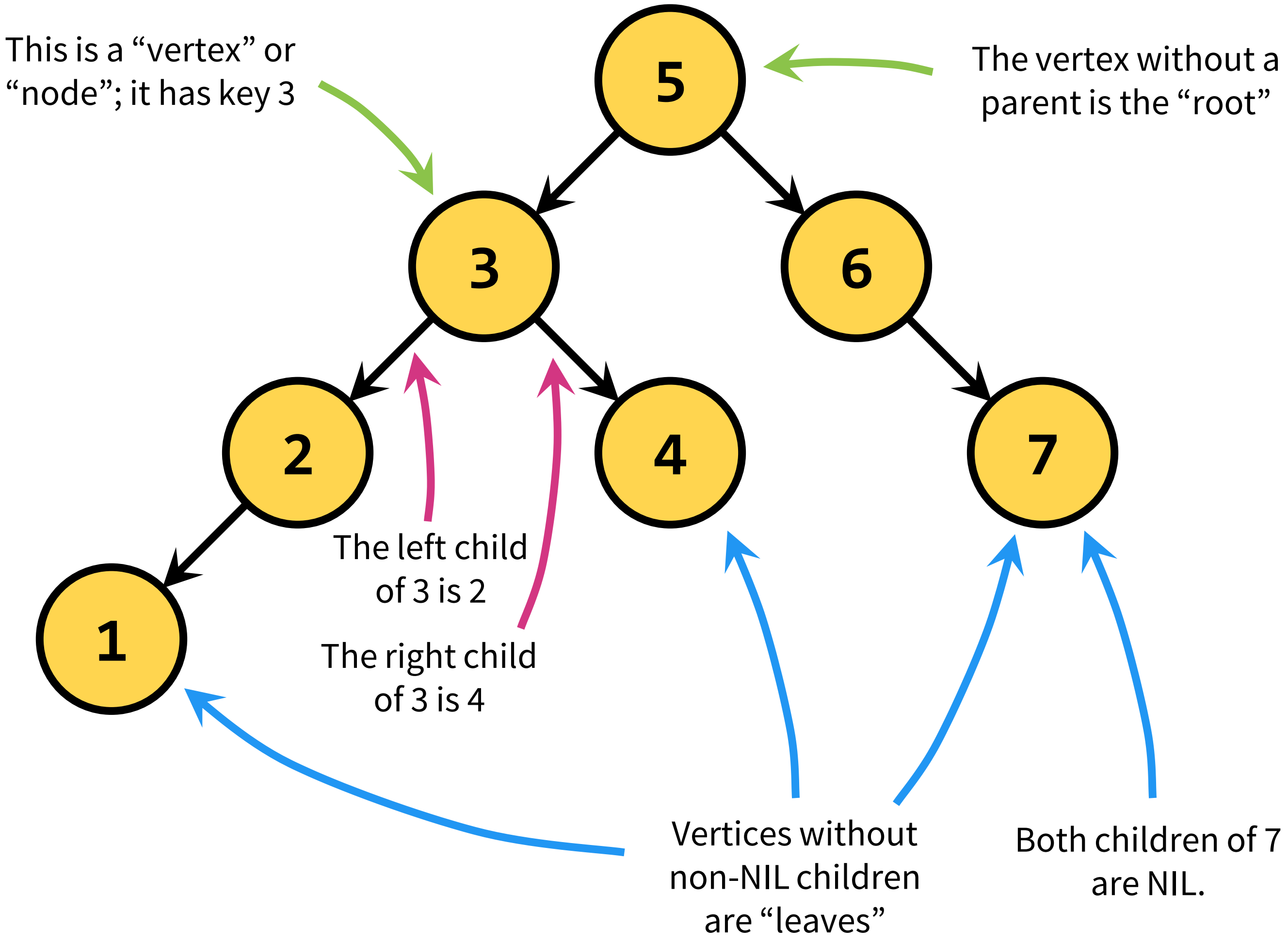
Sorted arrays: O(log n) search
O(1) select
O(n) insert/delete

“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



This is a "vertex" or "node"; it has key 3

The vertex without a parent is the "root"

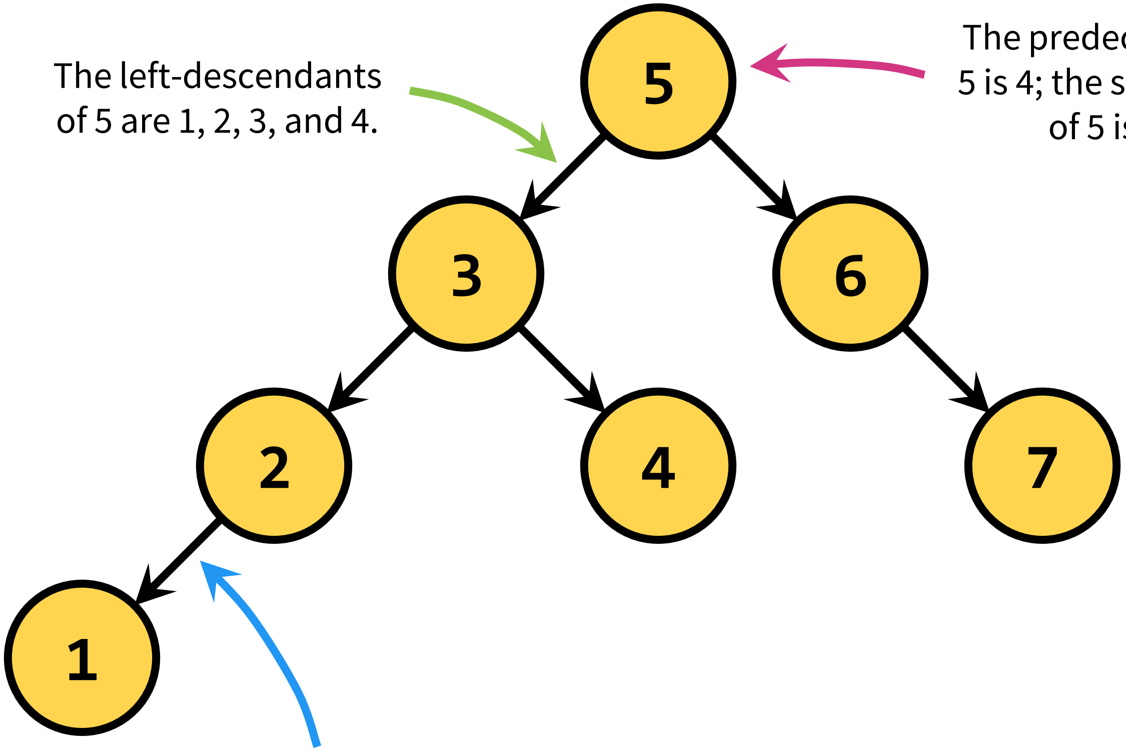
The left child of 3 is 2

The right child of 3 is 4

Vertices without non-NIL children are "leaves"

Both children of 7 are NIL.

Tree Terminology



The left-descendants of 5 are 1, 2, 3, and 4.

The predecessor of 5 is 4; the successor of 5 is 6.

The parent of 1 is 2; the ancestors of 1 are 2, 3, and 5.

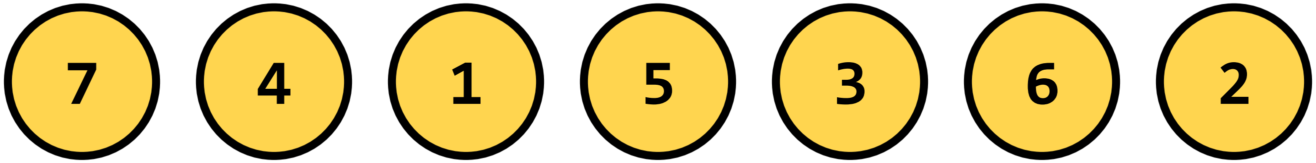
Binary Search Trees

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

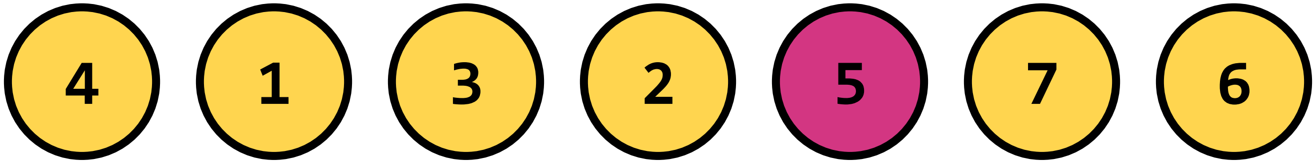
Binary Search Trees are Binary Trees such that:

- 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

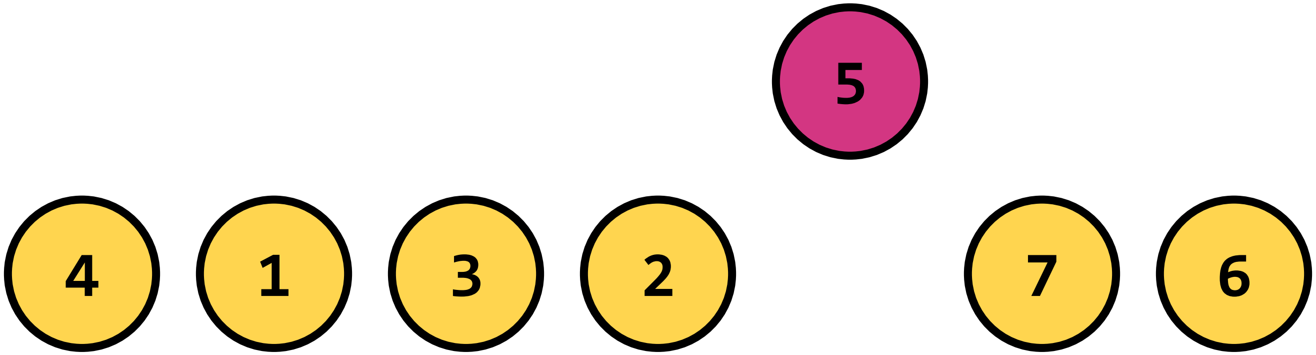


Building BSTs by Example



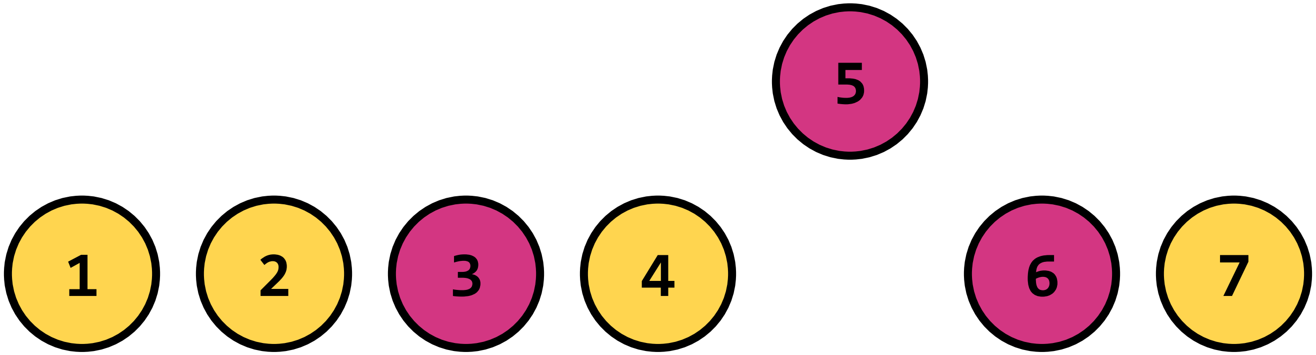
Let’s partition about one of the vertices …

Building BSTs by Example



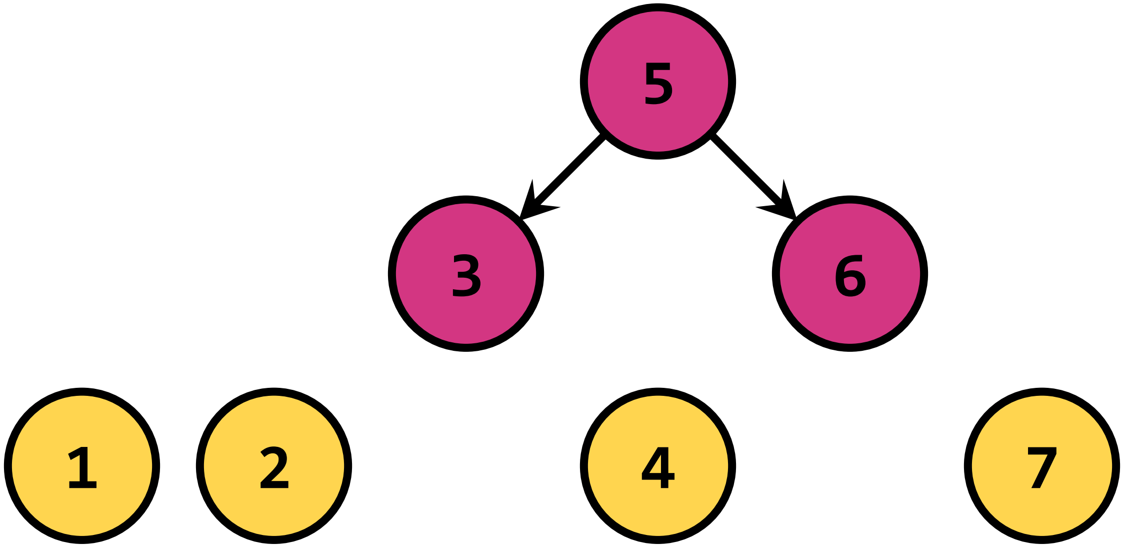
… and build a tree with that vertex as the root.

Building BSTs by Example



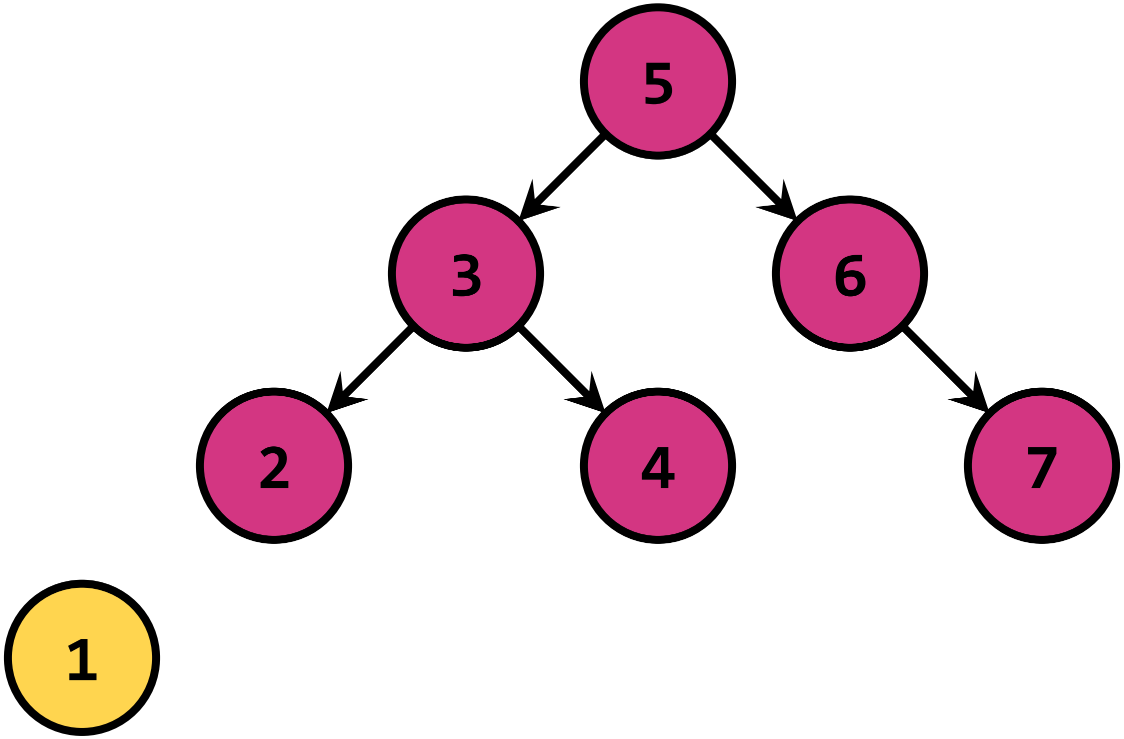
Then, recursively build trees out of its descendants.

Building BSTs by Example



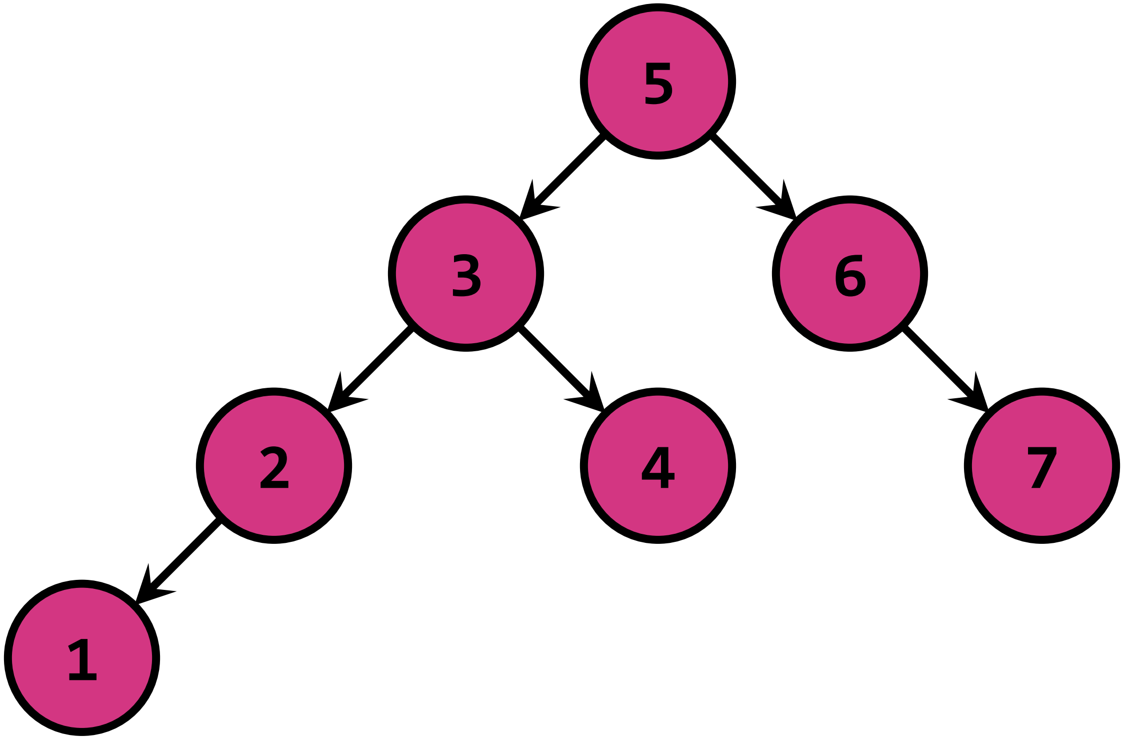
Then, recursively build trees out of its descendants.

Building BSTs by Example



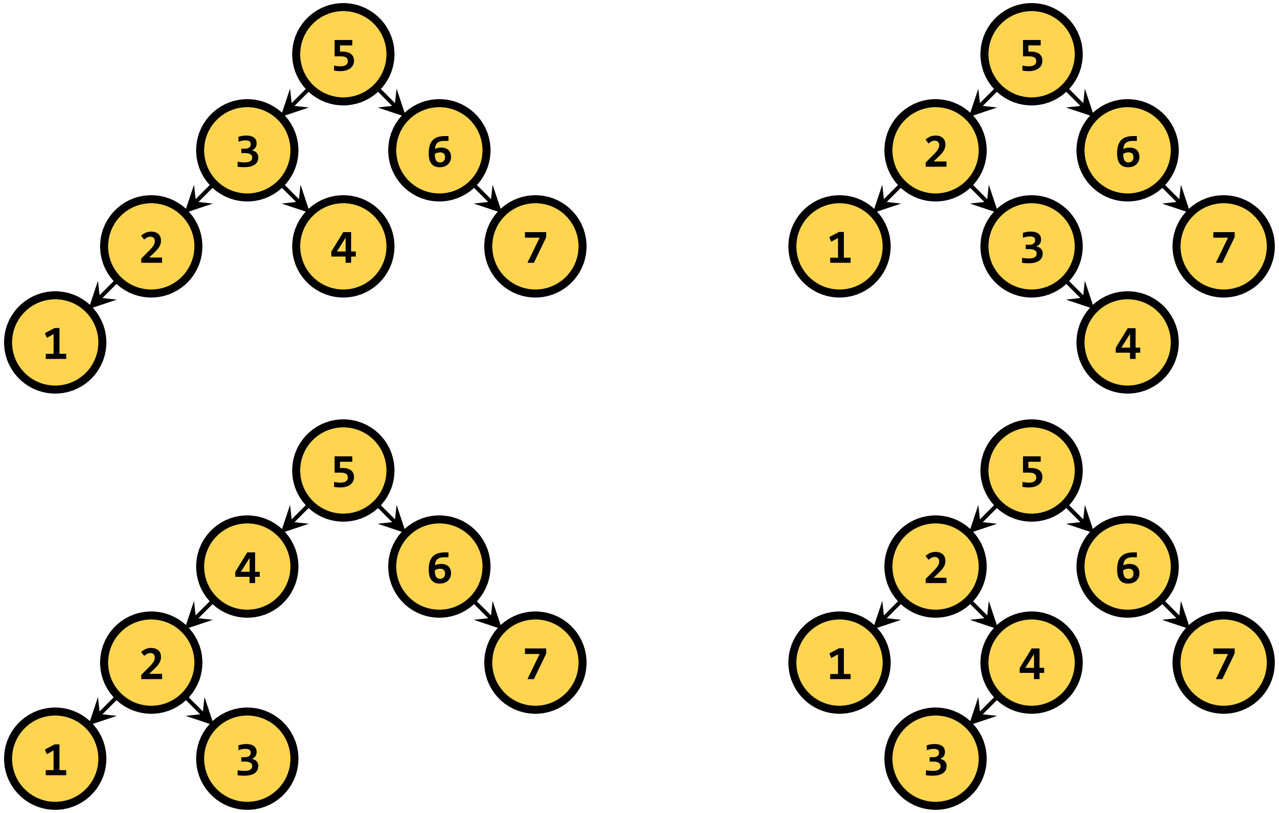
Then, recursively build trees out of its descendants.

Building BSTs by Example



Then, recursively build trees out of its descendants.

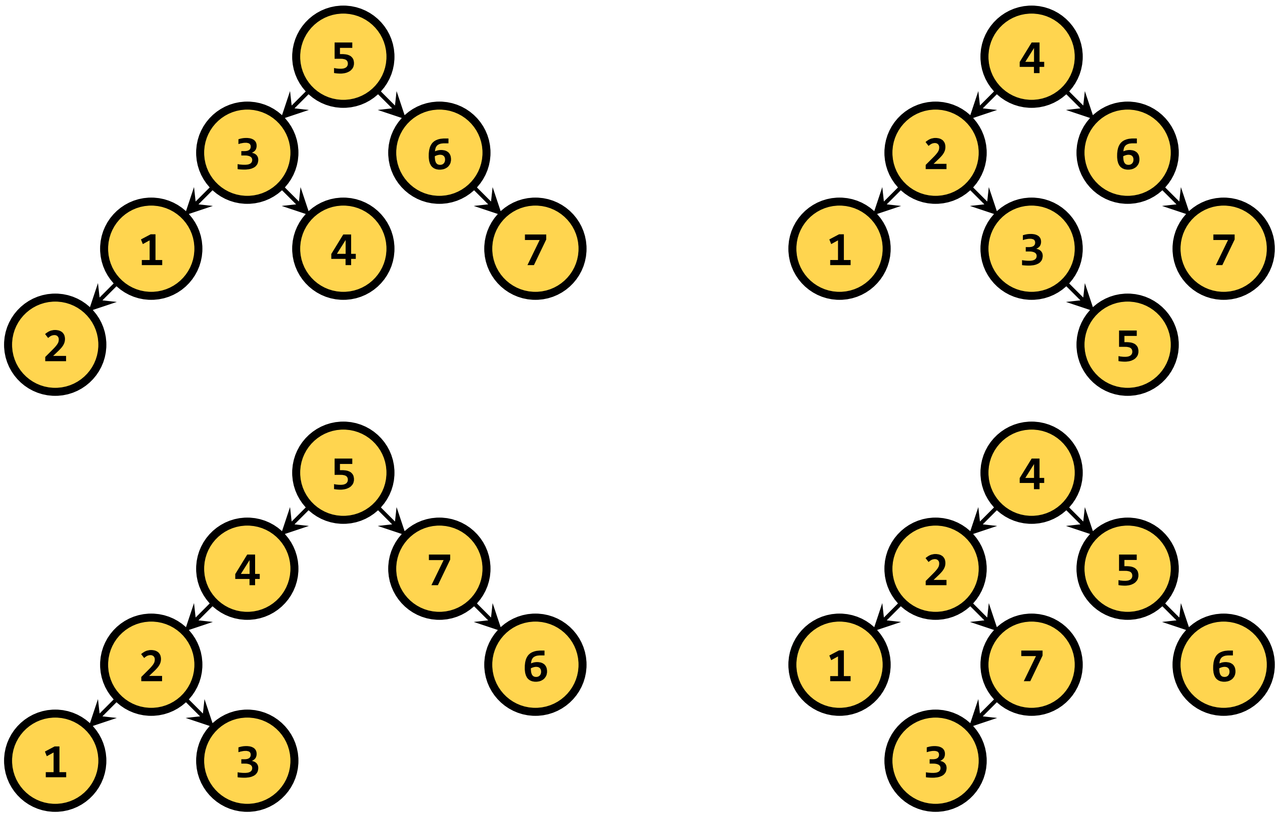
There Exist Many Valid BSTs



… and many more.

How many?

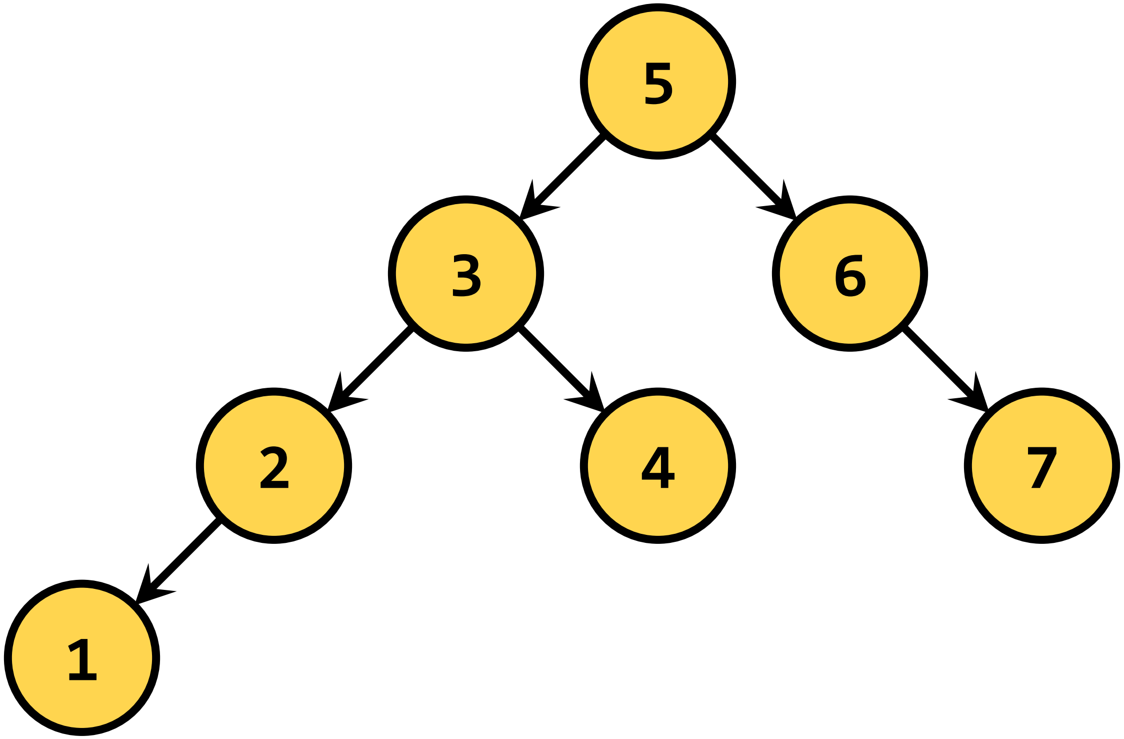
There Exist Many Invalid BSTs



… and many more.

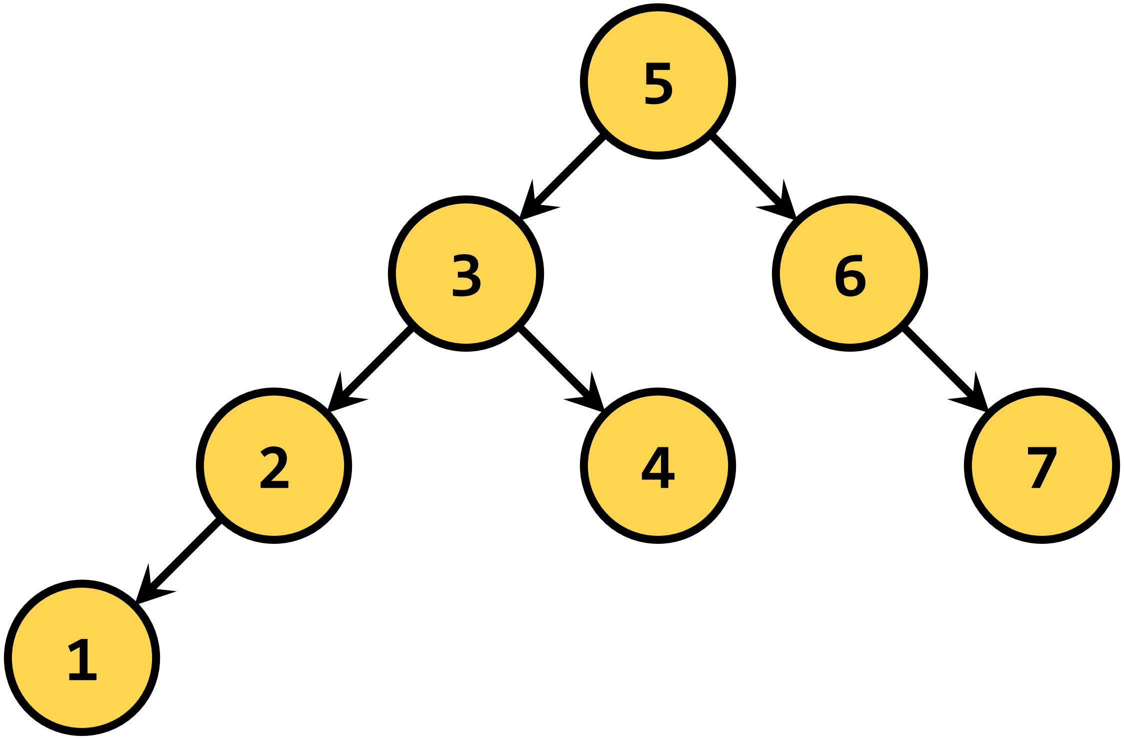
How many?

`search` in BSTs



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

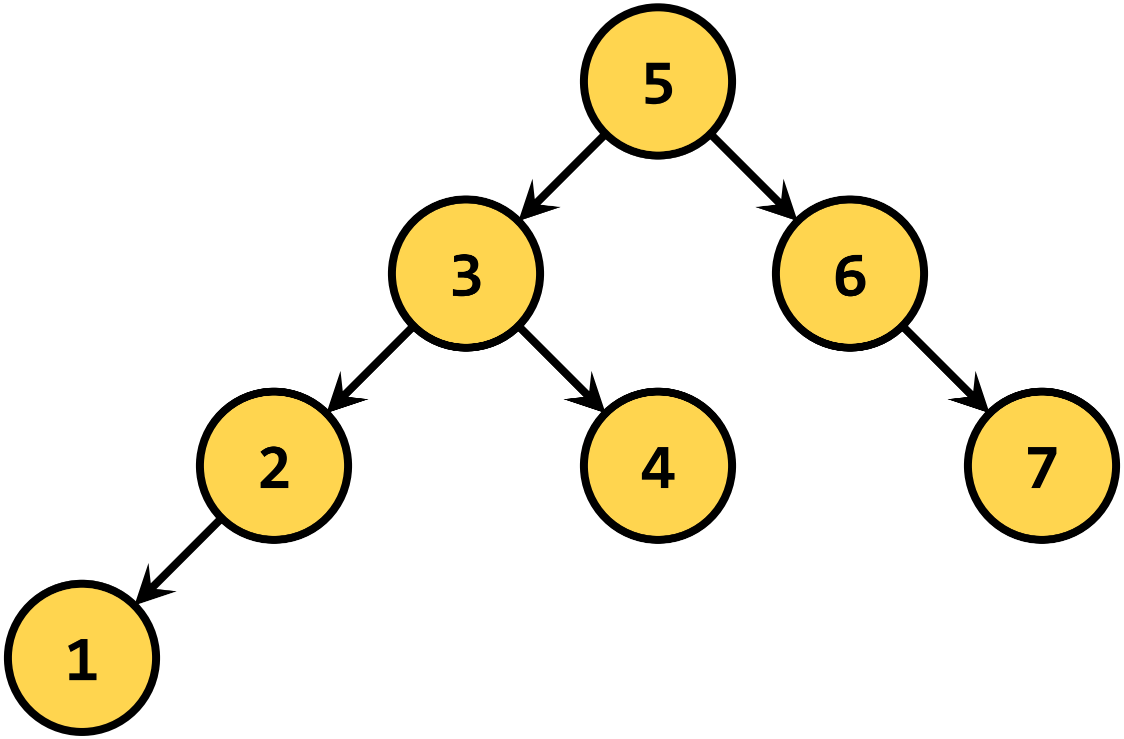
`search` in BSTs



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

```
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
```

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

delete in BSTs

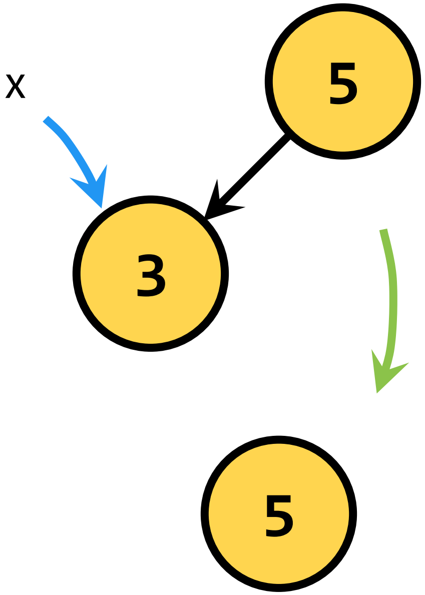
```
algorithm delete(root, key_to_insert):
  x = search(root, key_to_insert)
  if key_to_insert == x.key:
    delete x
```

This is somewhat complicated …

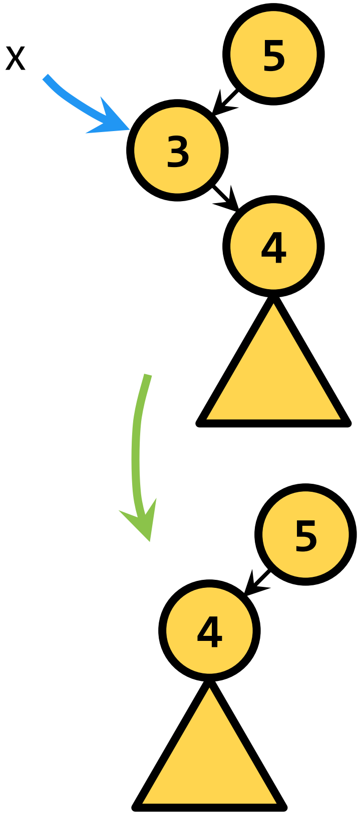
Runtime: `O(log n)` if balanced, `O(n)` otherwise

`delete` in BSTs

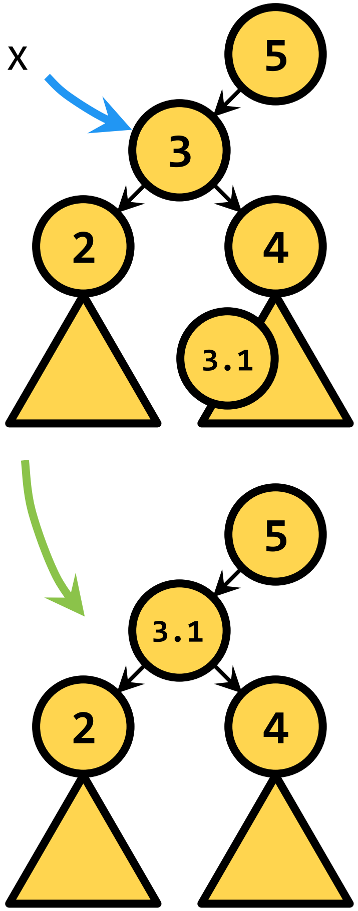
Case 1: x is a leaf
Just delete x



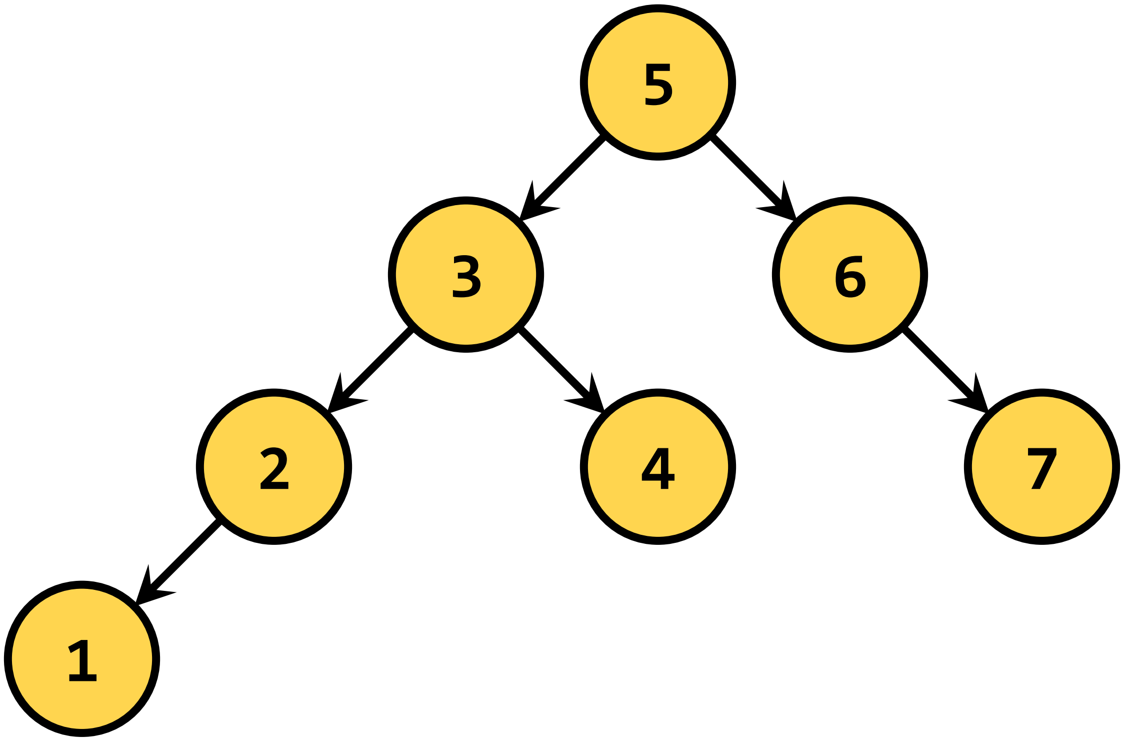
Case 2: x has 1 child
Move its child up



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

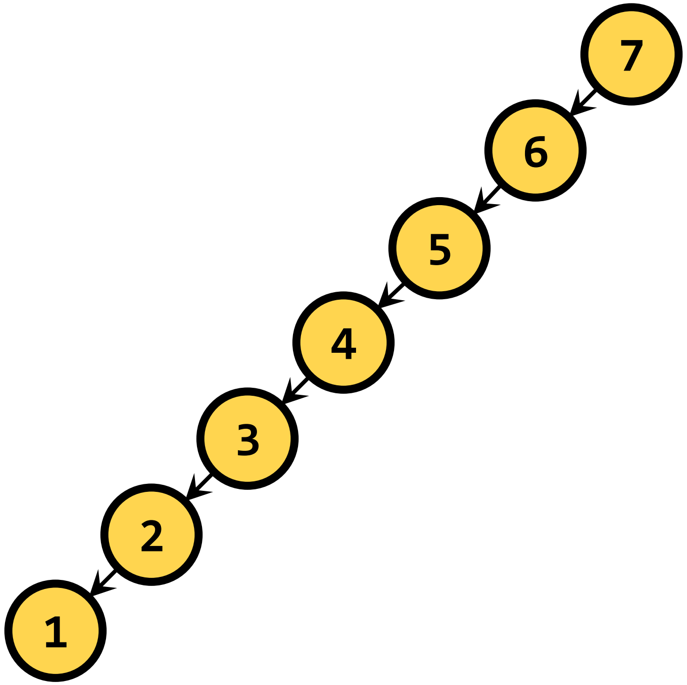


Runtime Analysis



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 Ω(n) every so often …

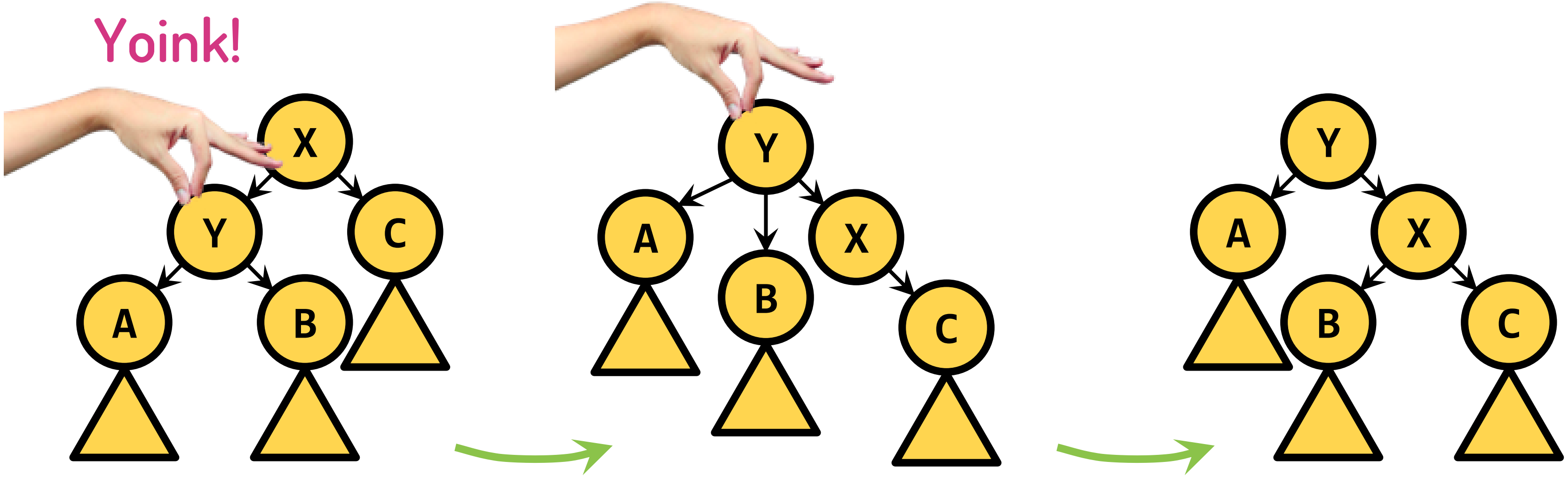
In the worst case, how often is "every so often"?

Any other ideas?

Idea 1: Rotations

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

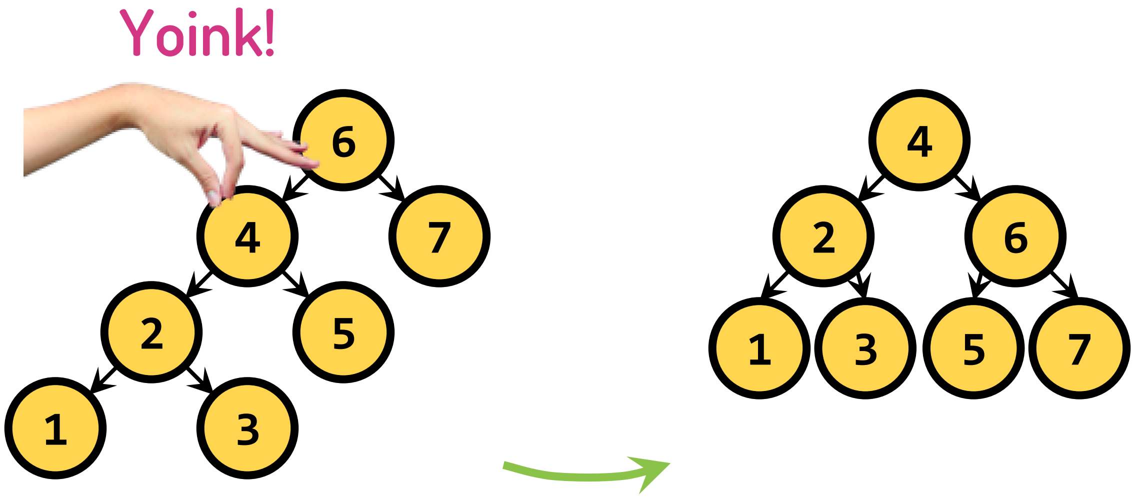
Yoink!



Not binary!

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.

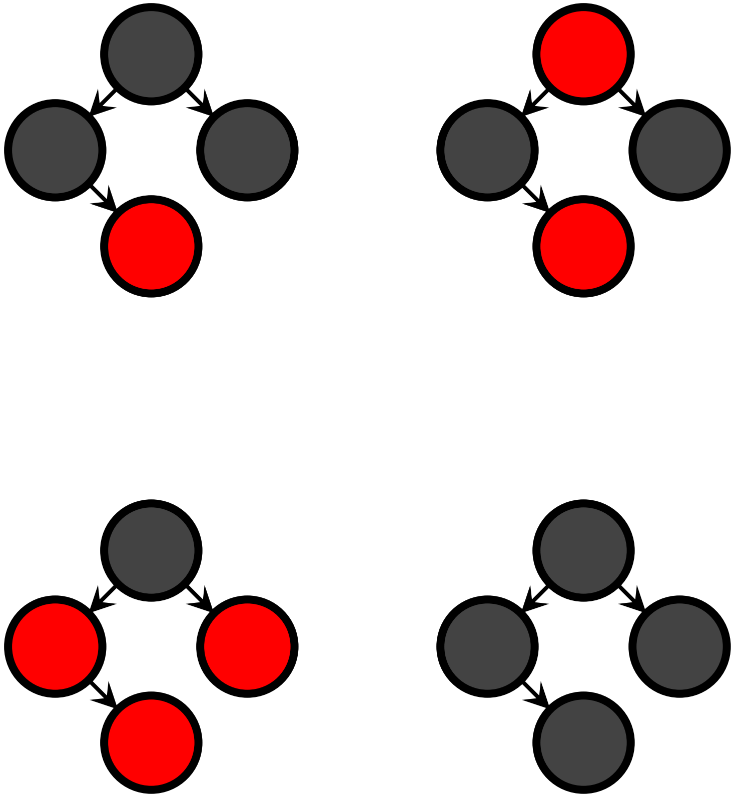
Red-Black Trees

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.

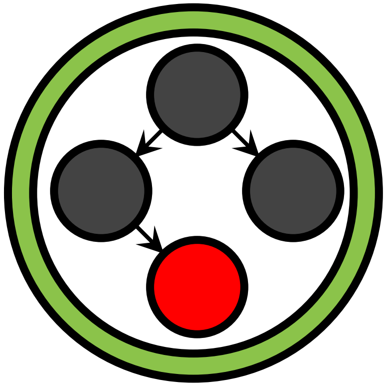
Red-Black Trees by Example

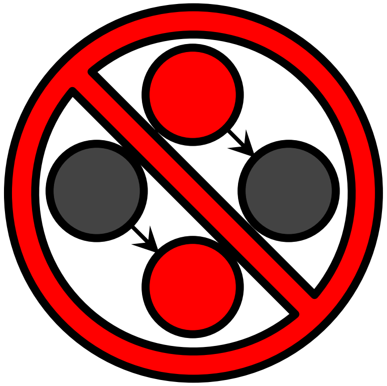
1. Every vertex is colored **red** or **black**.
2. The root vertex is a **black** vertex.
3. A NIL child is a **black** vertex.
4. The child of a **red** vertex must be a **black** vertex.
5. For all vertices v, all paths from v to its NIL descendants have the same number of **black** vertices.



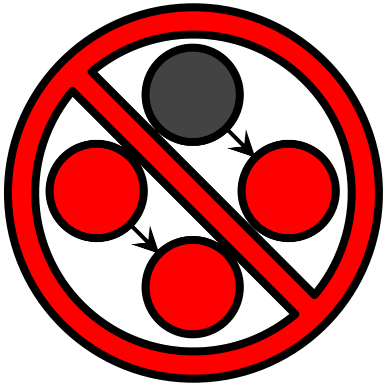
Red-Black Trees by Example

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.

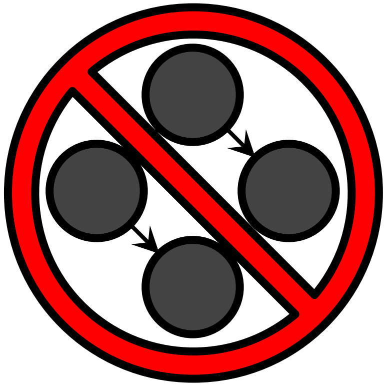




Violates 2



Violates 4



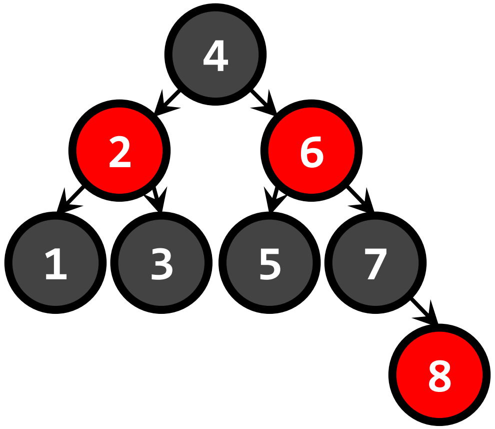
Violates 5

Red-Black Trees

Maintaining these properties maintains a “pretty balanced” BST.

- The **black** vertices are balanced.
- The **red** vertices are “spread out.”

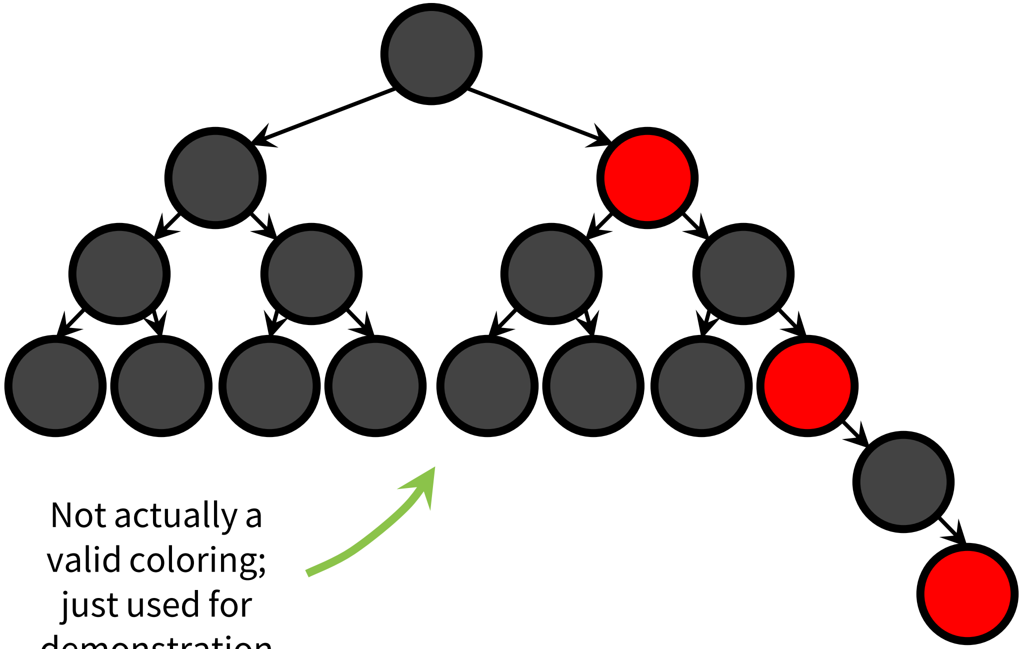
We can maintain this property as we insert/delete vertices, using rotations.



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 $2^{b(x)} - 1$.

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:

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

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 ≤ 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

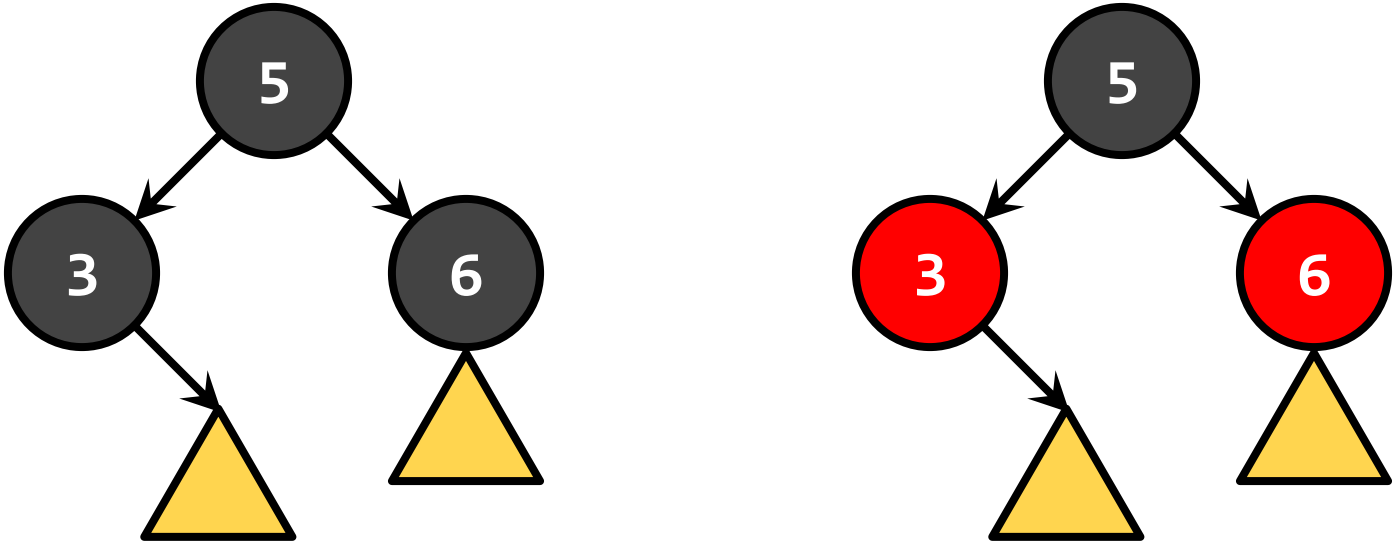
```
algorithm rb_insert(root, key_to_insert):
  x = search(root, key_to_insert)
  v = new red vertex with key_to_insert
  if key_to_insert > x.key:
    x.right = v
    fix things up, if necessary
  if key_to_insert < x.key:
    x.left = v
    fix things up, if necessary
  if key_to_insert == x.key:
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

What does that mean?

`insert` in Red-Black Trees

What does "if necessary" mean?

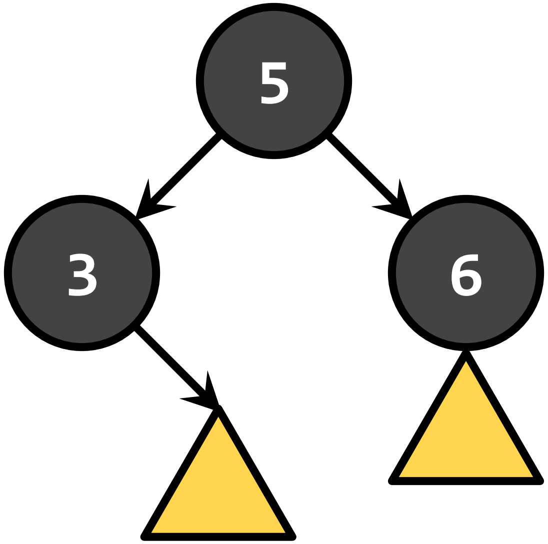
Suppose we want to `insert(1)`.

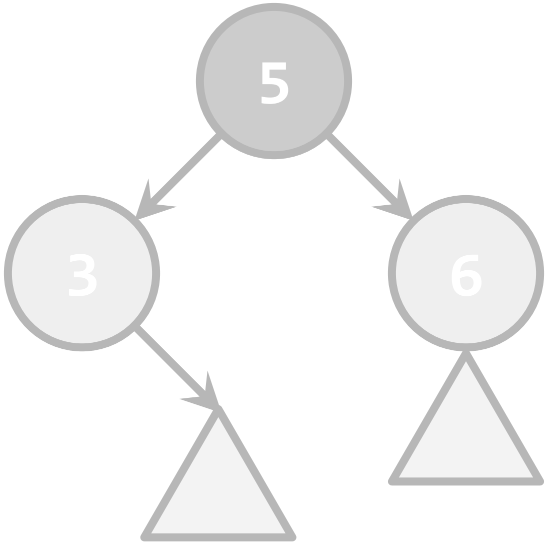


`insert` in Red-Black Trees

What does "if necessary" mean?

Suppose we want to `insert(1)`.

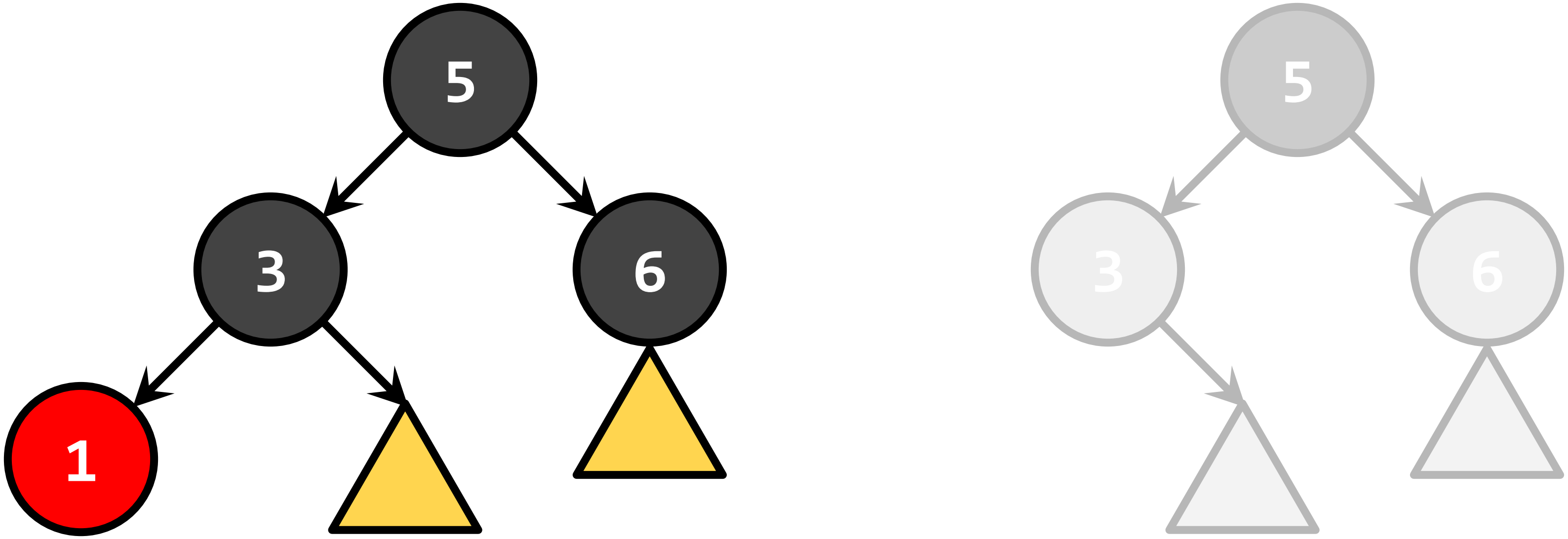




`insert` in Red-Black Trees

What does “if necessary” mean?

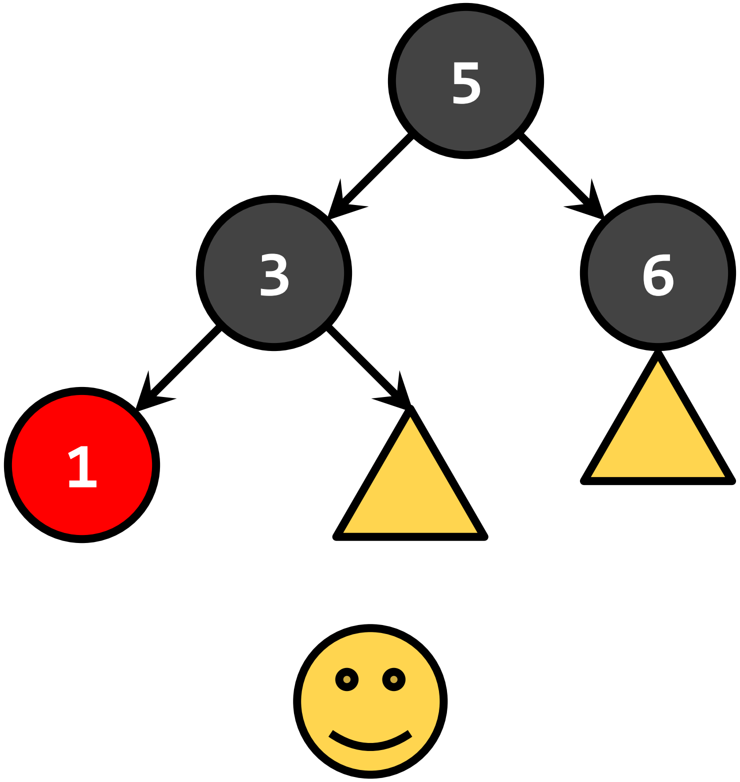
Suppose we want to `insert(1)`.

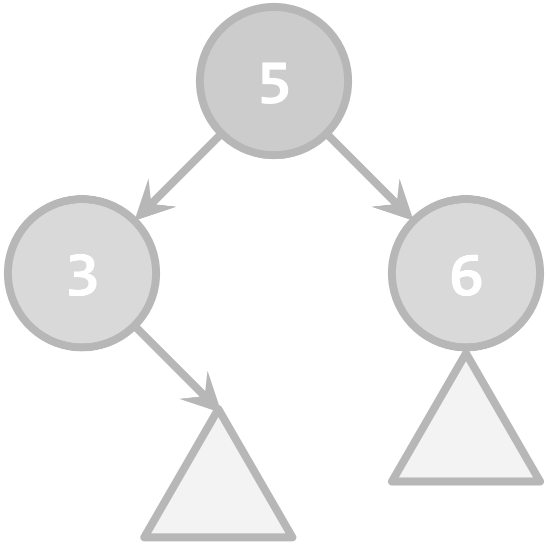


`insert` in Red-Black Trees

What does "if necessary" mean?

Suppose we want to `insert(1)`.

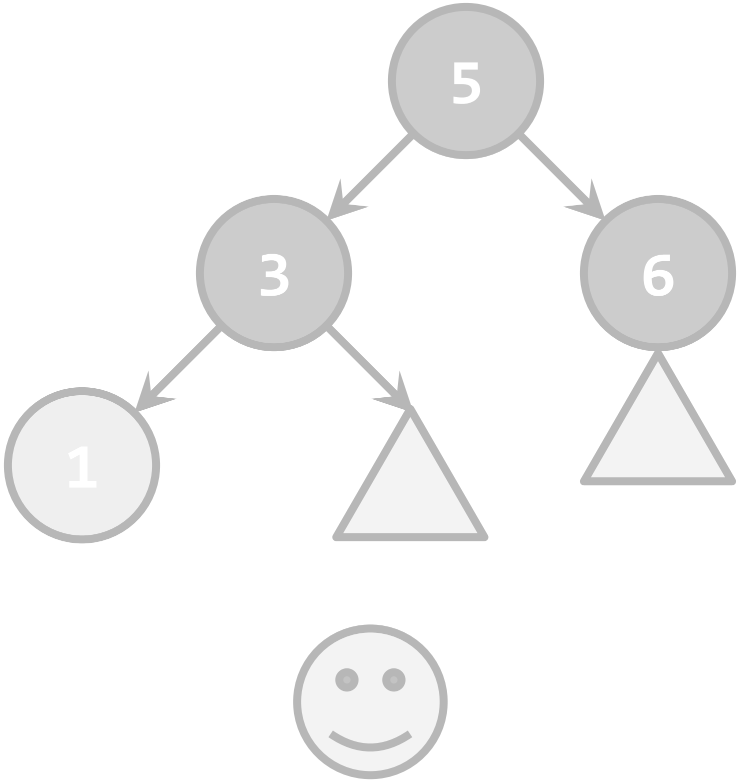


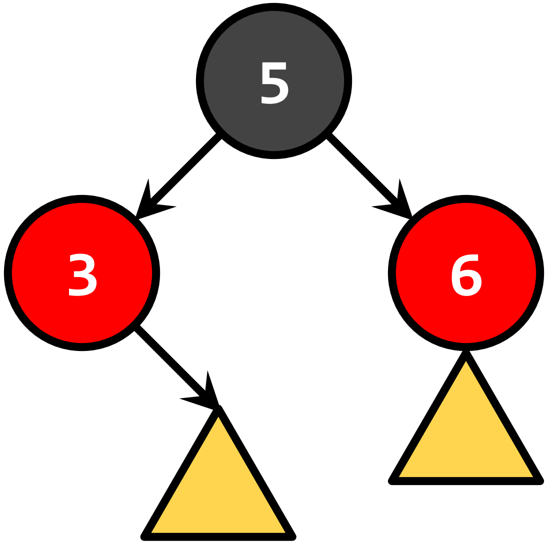


`insert` in Red-Black Trees

What does "if necessary" mean?

Suppose we want to `insert(1)`.

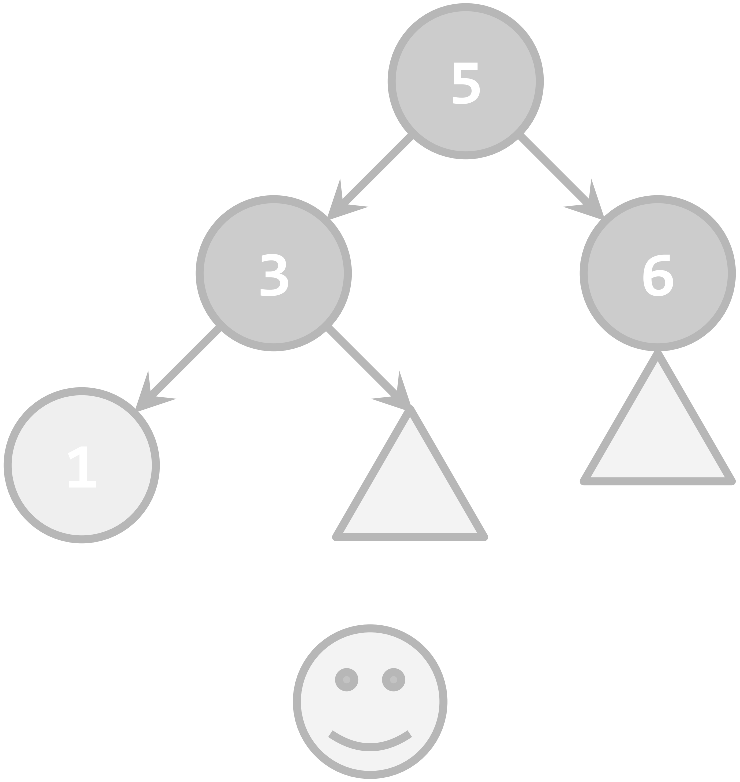


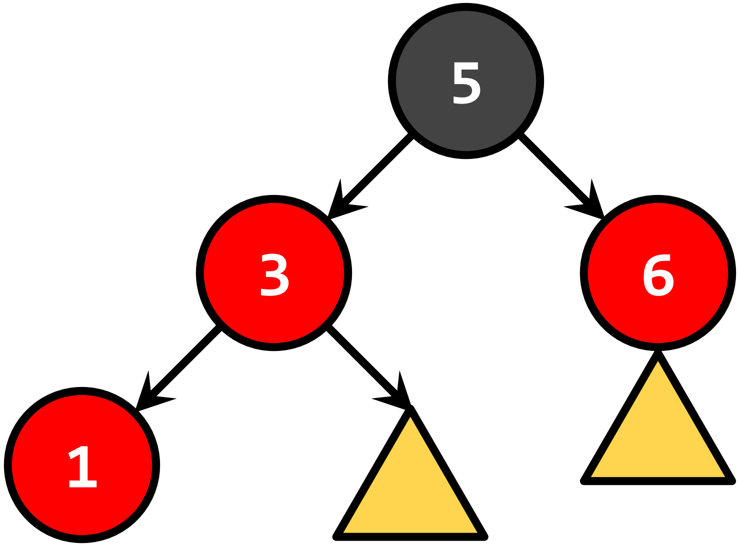


`insert` in Red-Black Trees

What does “if necessary” mean?

Suppose we want to `insert(1)`.

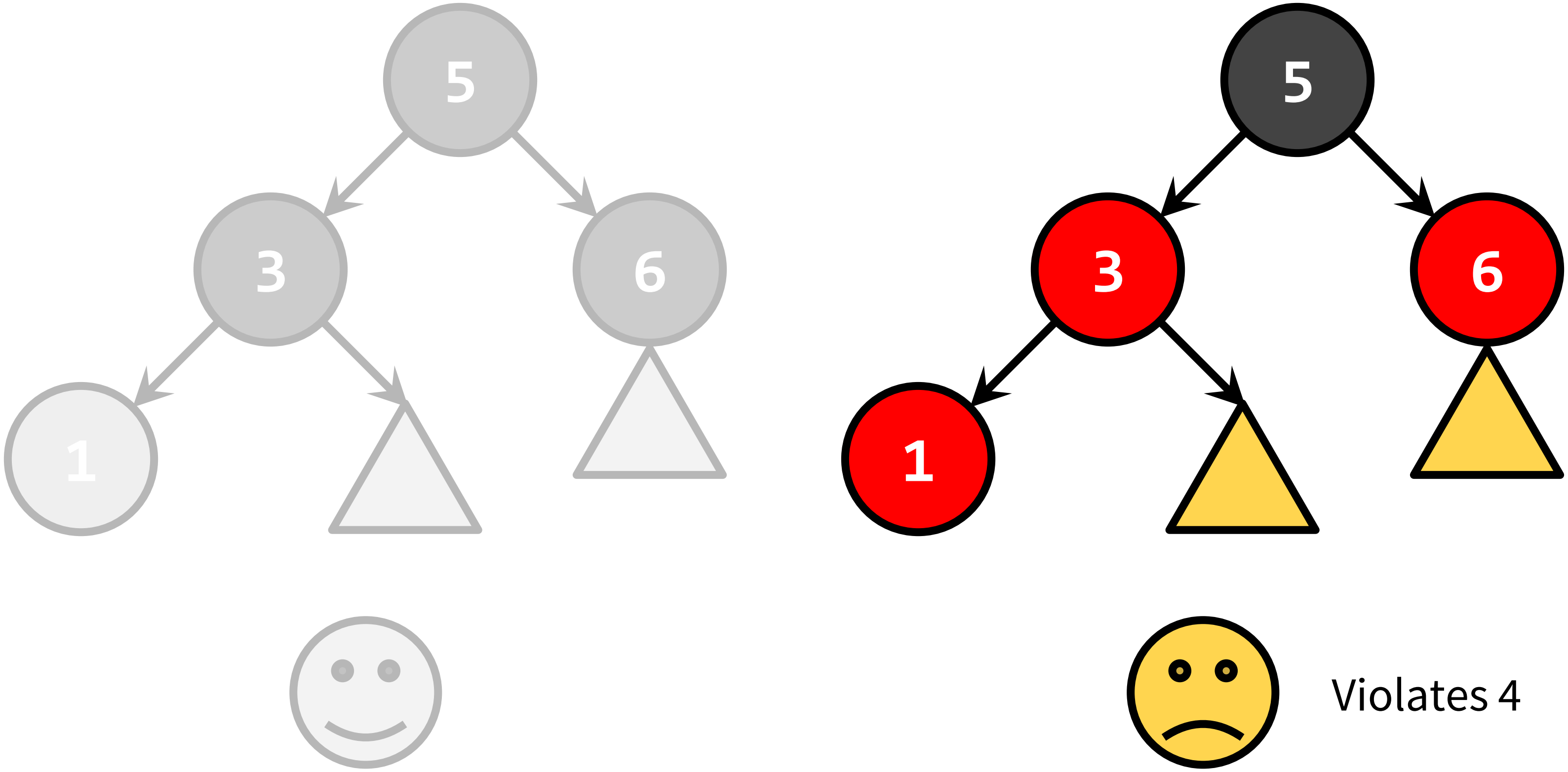




`insert` in Red-Black Trees

What does “if necessary” mean?

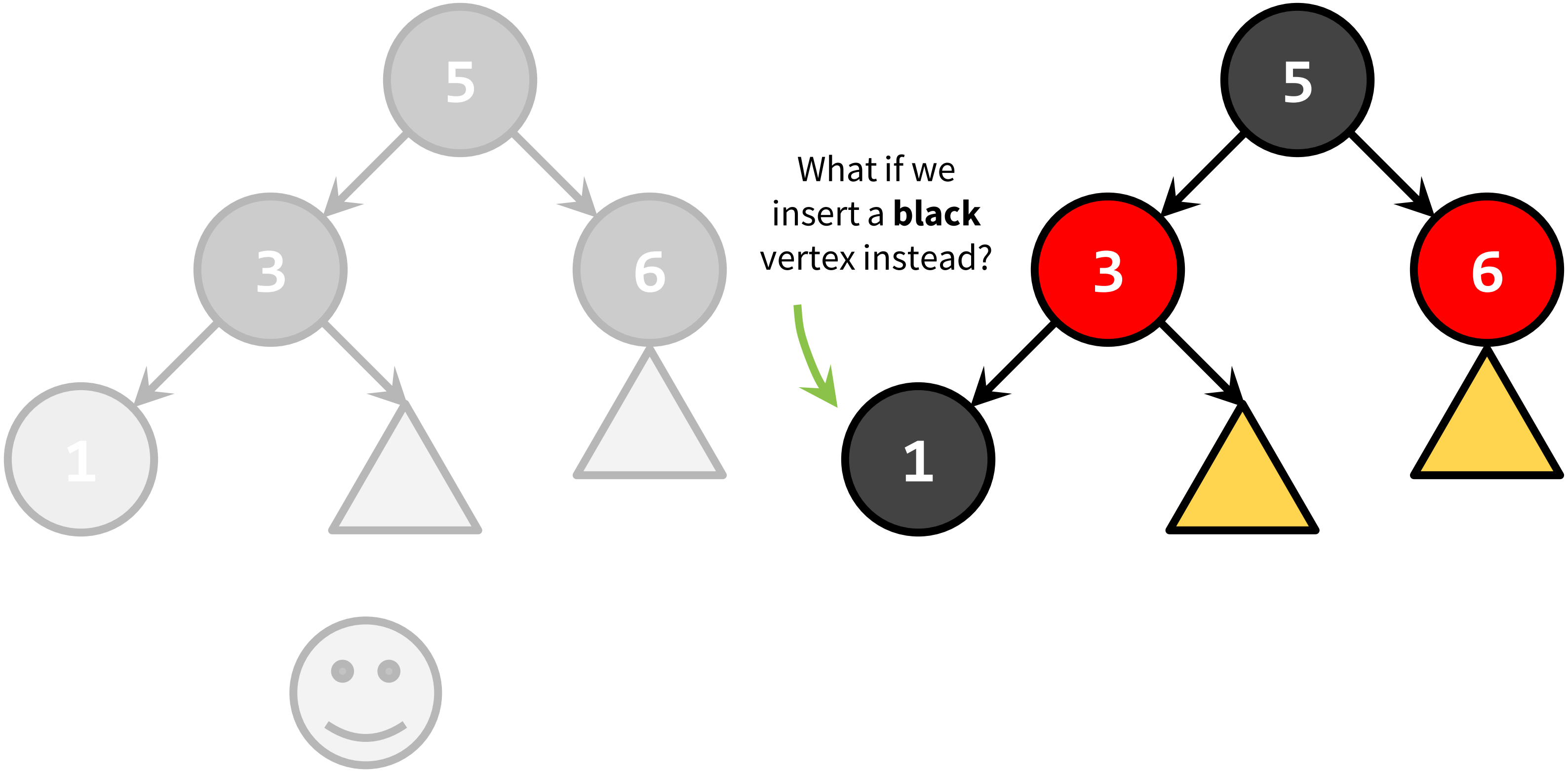
Suppose we want to `insert(1)`.



`insert` in Red-Black Trees

What does "if necessary" mean?

Suppose we want to `insert(1)`.

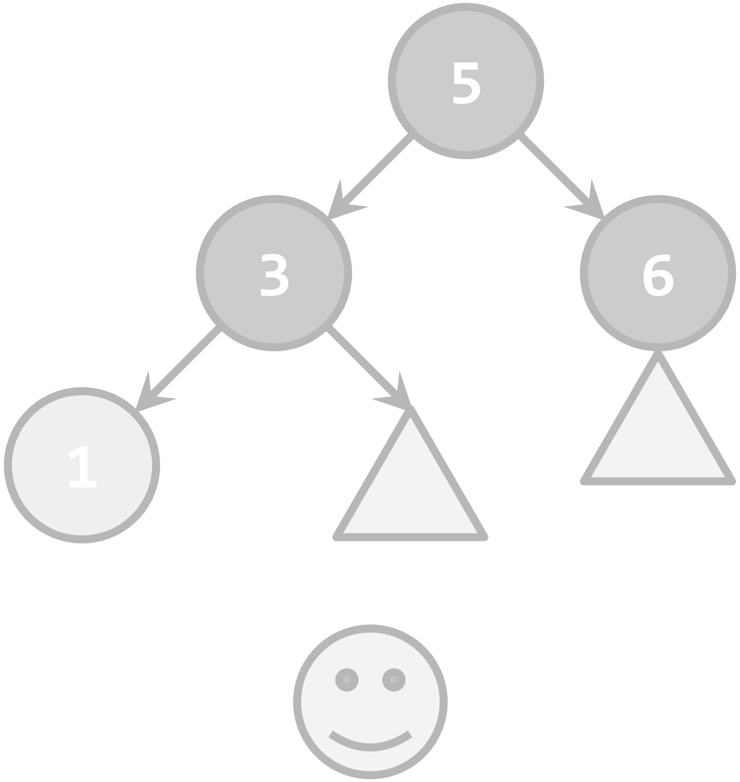


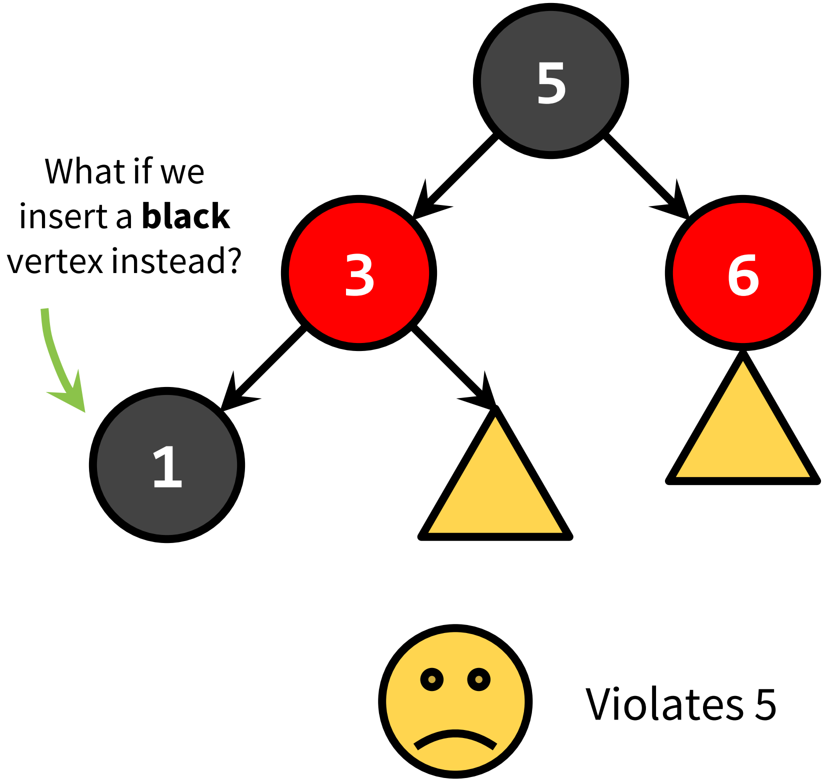
What if we insert a **black** vertex instead?

`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**.

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):
  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!