Tutorial 07 — BST

CS2040C Semester 1 2020/2021

By Jin Zhe, adapted from slides by Ranald, AY1819 S2 Tutor

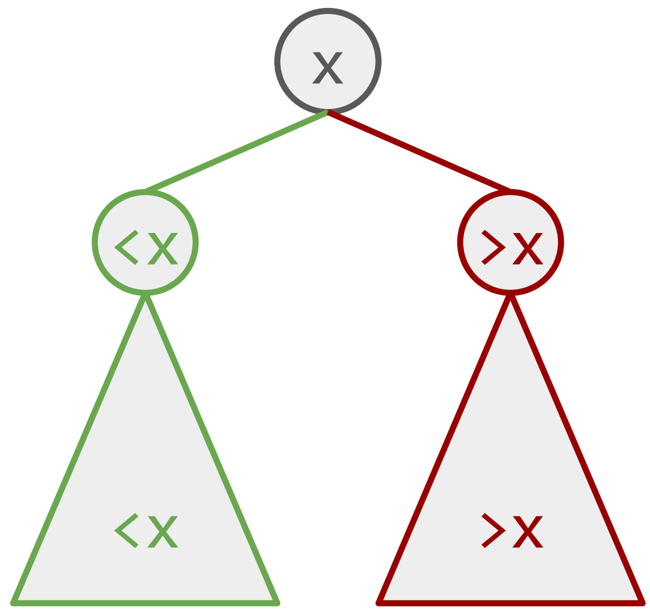
General Properties of BST

Non-Balanced

Definition

A Binary Search Tree (BST) is a *binary tree* where for every vertex **X**:

- It has **at most** 2 children
- Every vertex in left subtree is lesser than **X**
- Every vertex in right subtree is greater than **X**



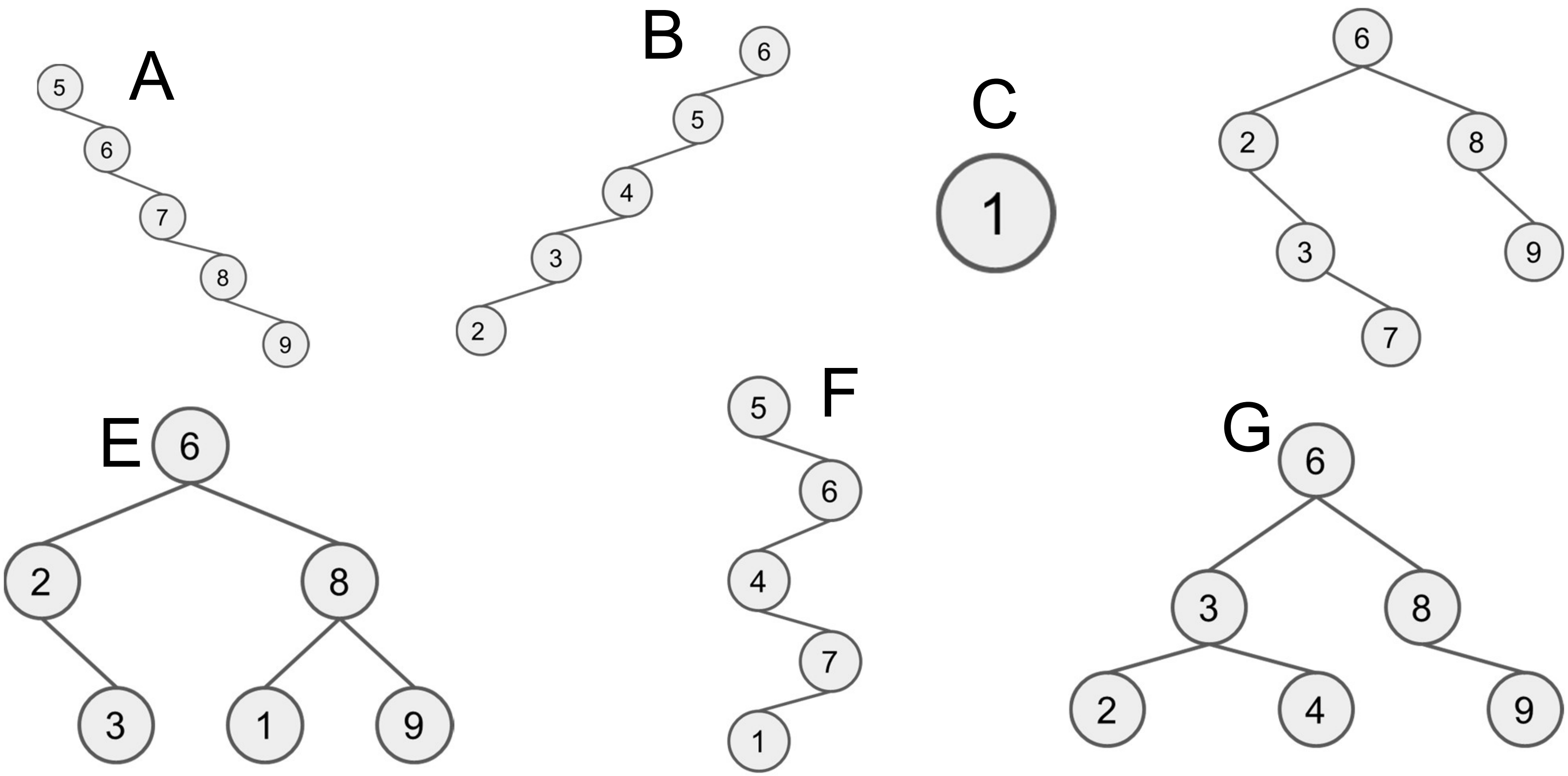
Note: We assume no duplicates for now

Test yourself!

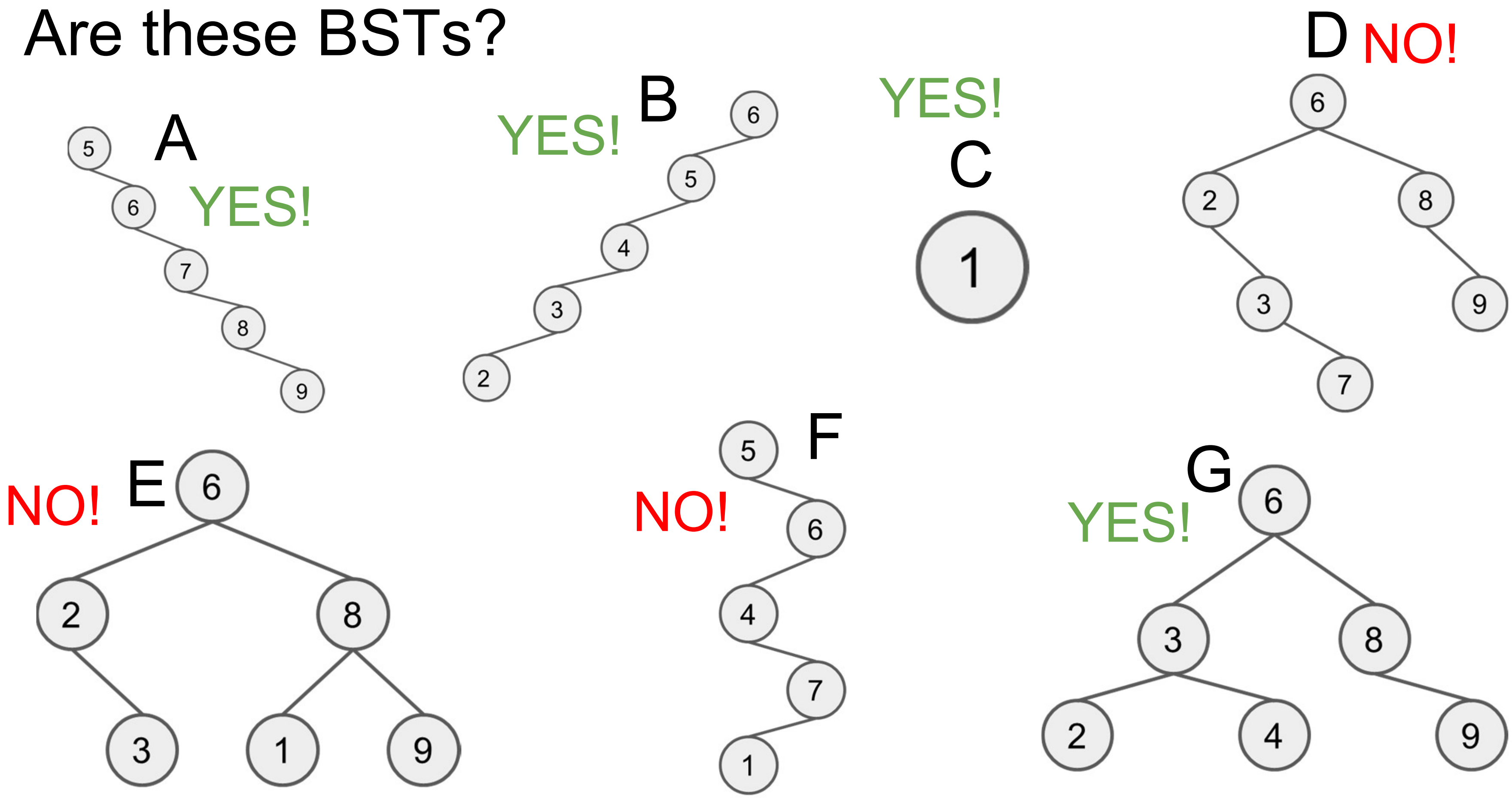
In the next few slides we quickly shall test your understanding of BSTs!

For the sake of brevity, we shall denote a Binary Search Tree rooted at vertex x to be BST_x

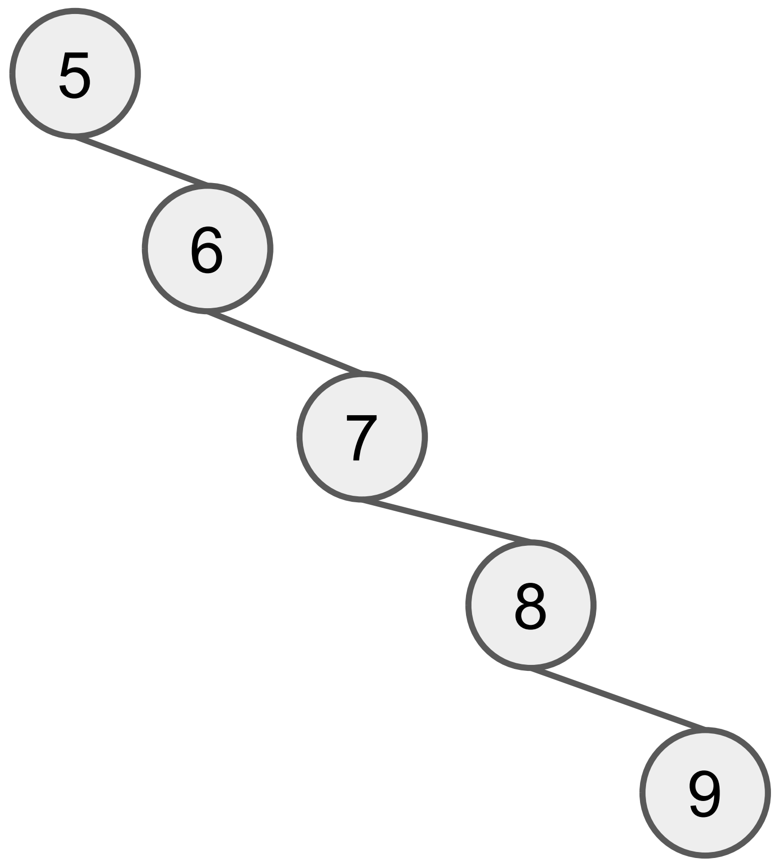
Are these BSTs?



Are these BSTs?



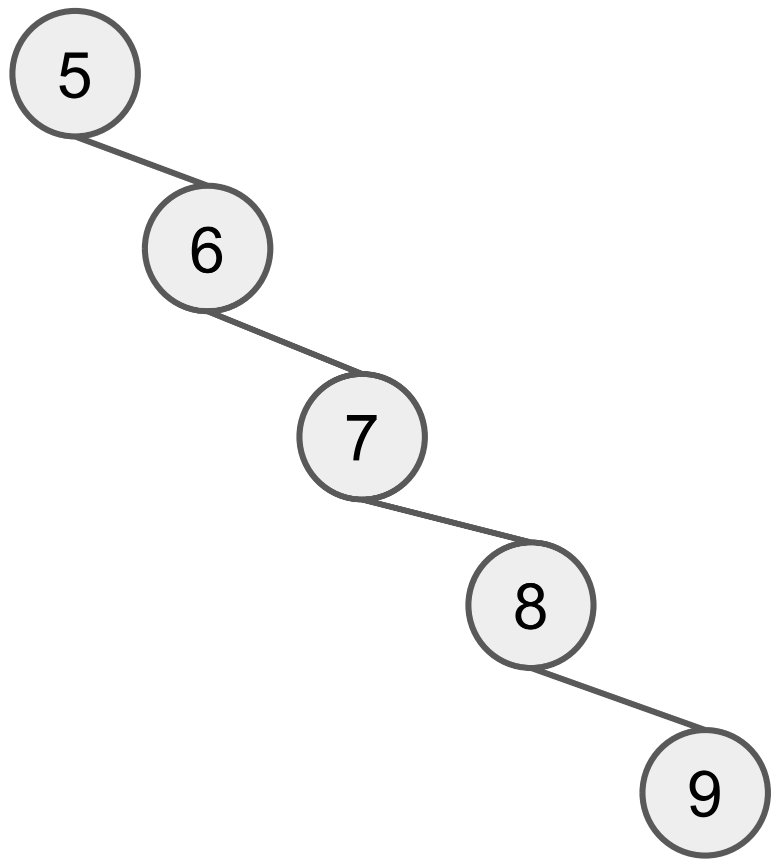
Is this a BST?



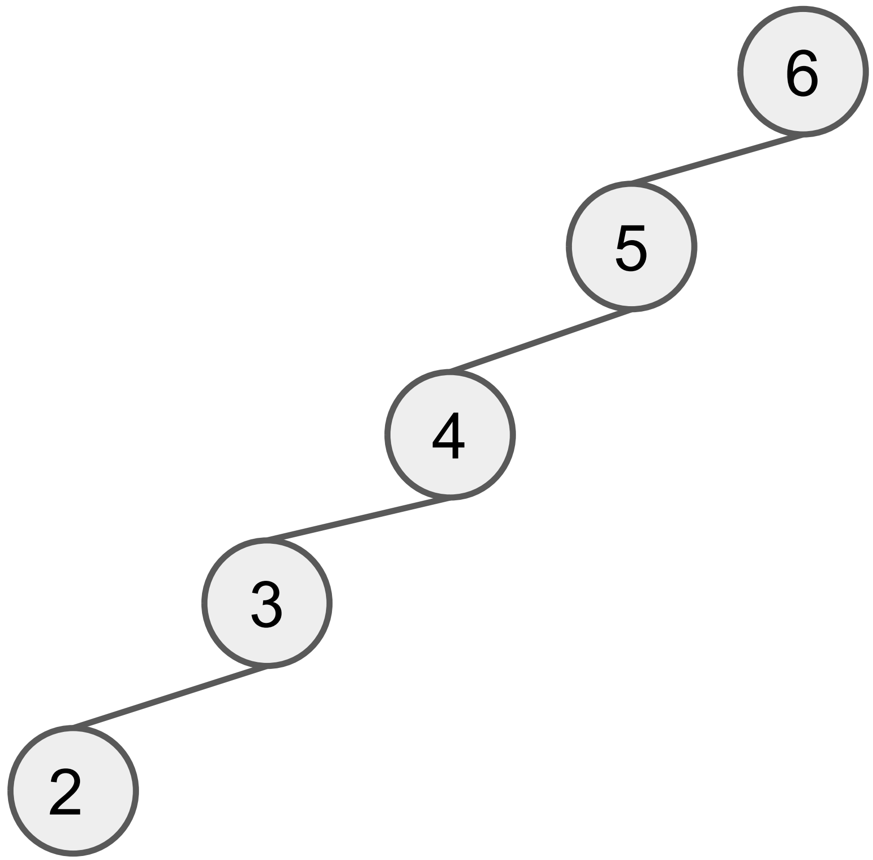
Is this a BST?

YES!

Unlike in binary heap, a BST vertex can have right child without left child

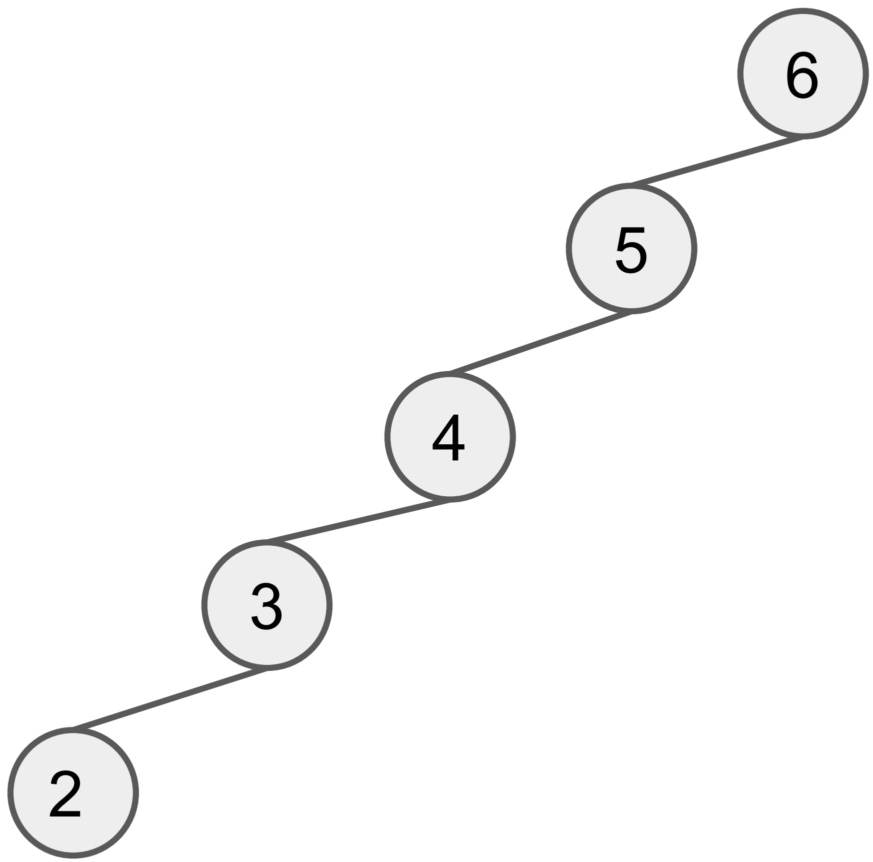


Is this a BST?



Is this a BST?

YES!



Is this a BST?

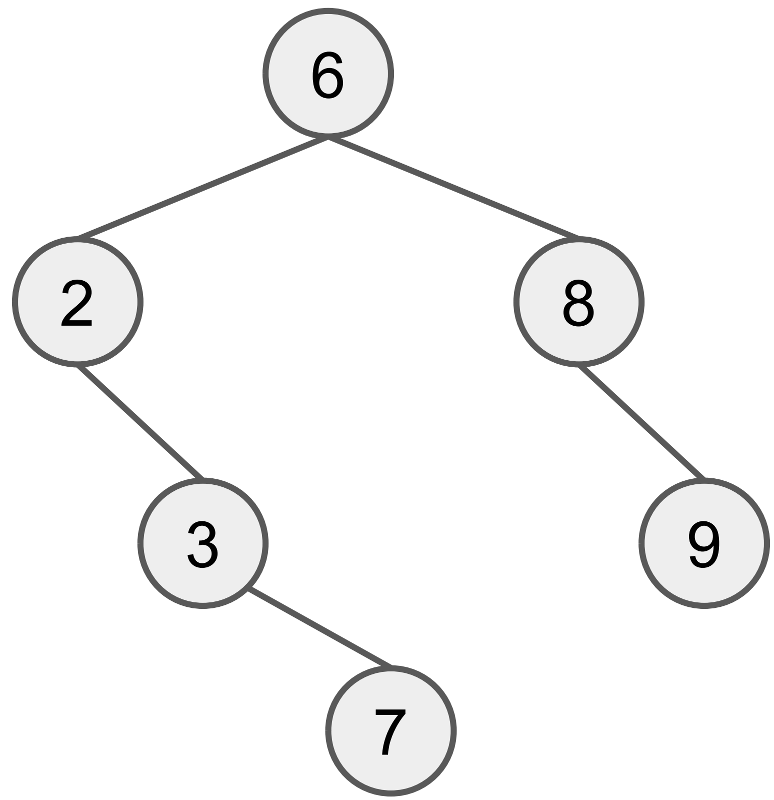
Is this a BST?

YES!

A BST vertex can have no children.

In fact we need this condition for binary trees to allow for leaves! :O

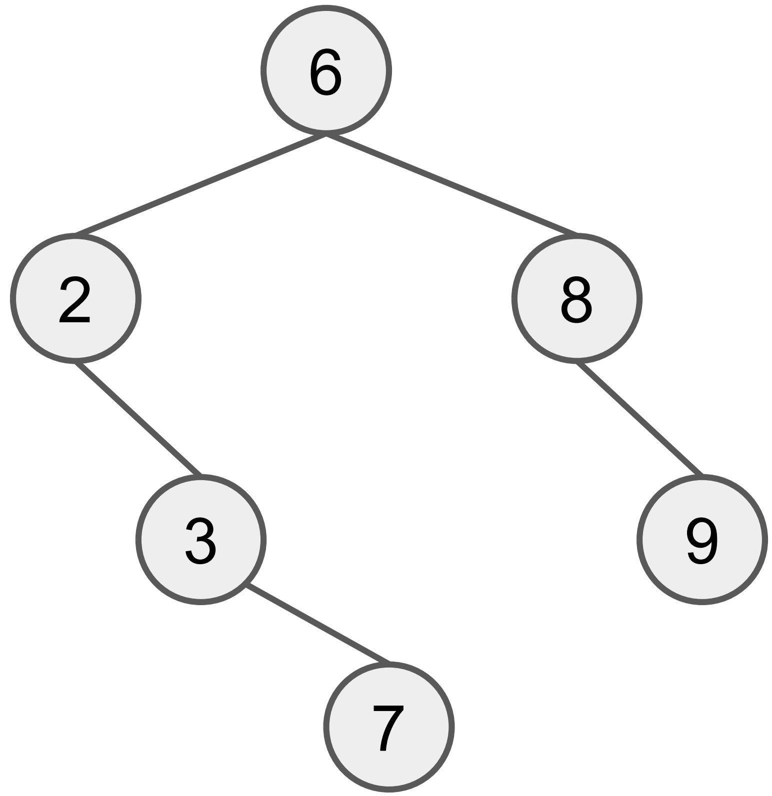
Is this a BST?



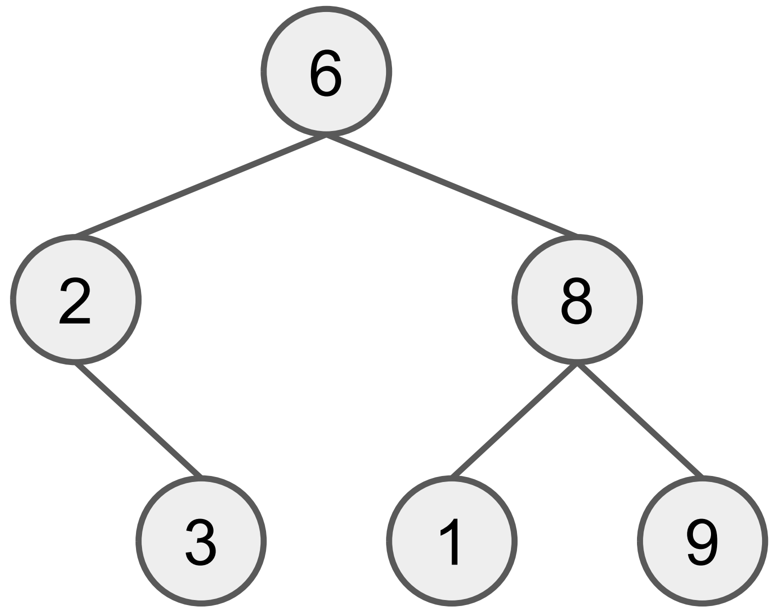
Is this a BST?

NO!

BST_7 violates BST_6



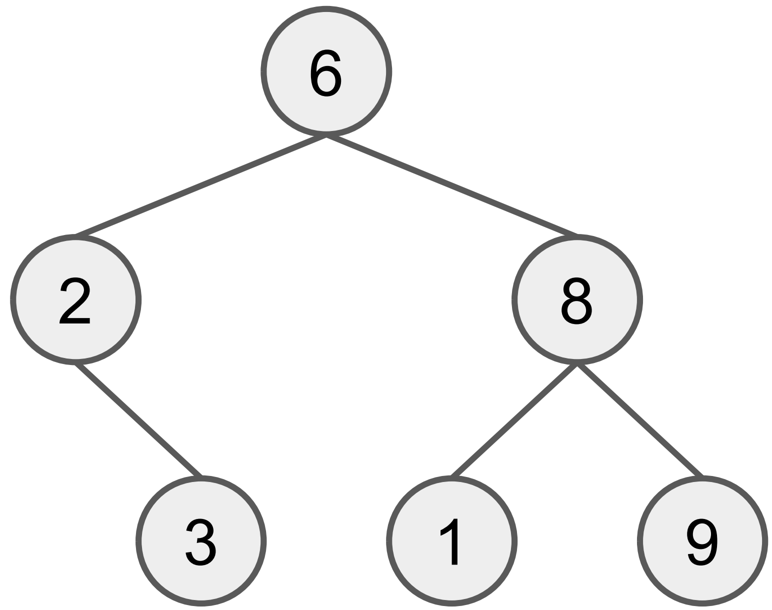
Is this a BST?



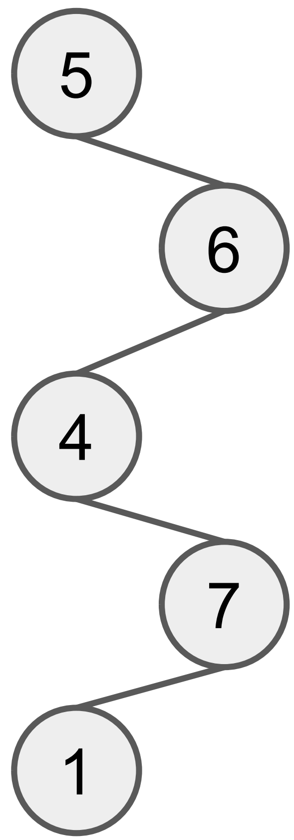
Is this a BST?

NO!

BST_1 violates BST_6



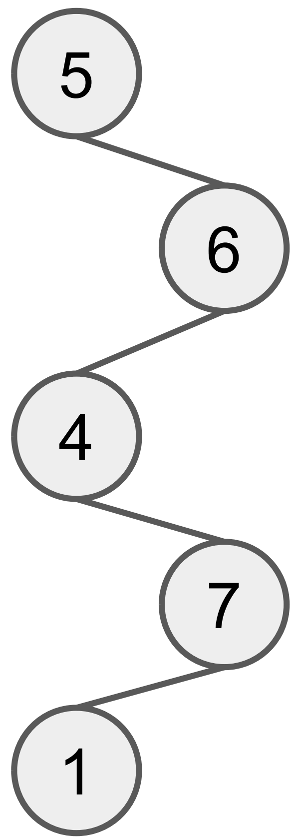
Is this a BST?



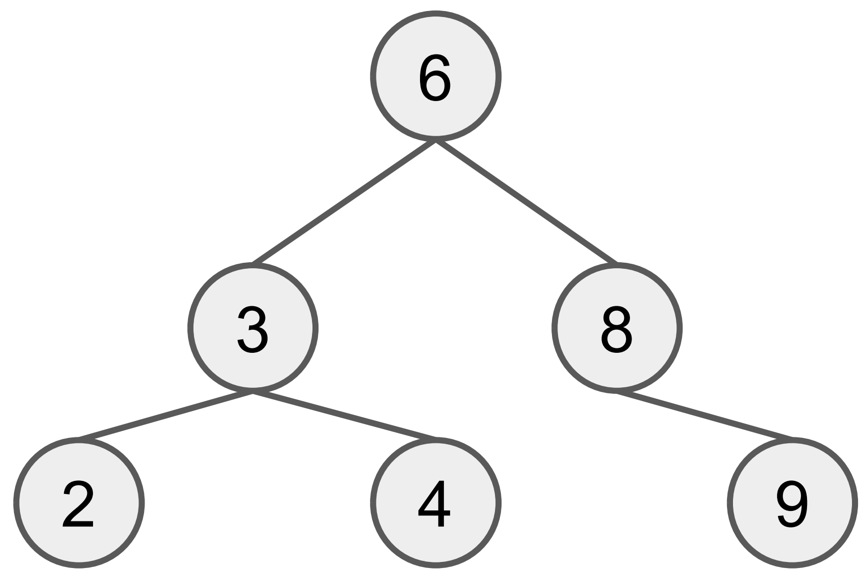
Is this a BST?

NO!

BST$_4$ violates BST$_5$
BST$_1$ violates BST$_5$
BST$_7$ violates BST$_6$
BST$_1$ violates BST$_4$

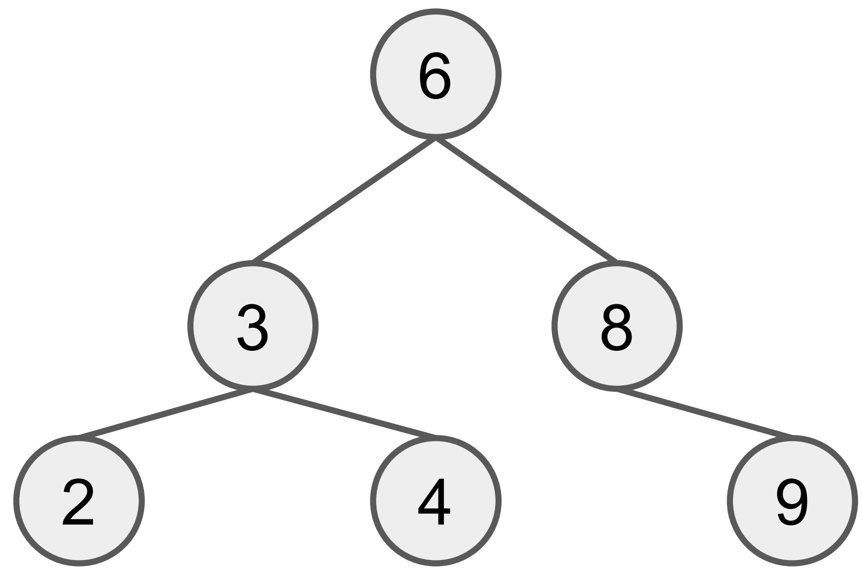


Is this a BST?



Is this a BST?

YES!



Test yourself!

Is a BST <u>always</u> a *complete binary tree*?

What about a **balanced** BST?

Hint: see previous slide

Not always!

Test yourself!

- What is the **min/max** height of a BST with **31** vertices?
- Height = number of edges from root to deepest leaf
- How did you get those values?

Test yourself!

- What is the **min/max** height of a BST with 31 vertices?
- Height = number of edges from root to deepest leaf
- How did you get those values?

Answer:

- Minimum: $\lfloor log_2 31 \rfloor = 4$ *Recall tutorial 4 slide 7*
- Maximum: 30
- Also see VisuAlgo BST slides 13-5 and 13-7

In-order Traversal

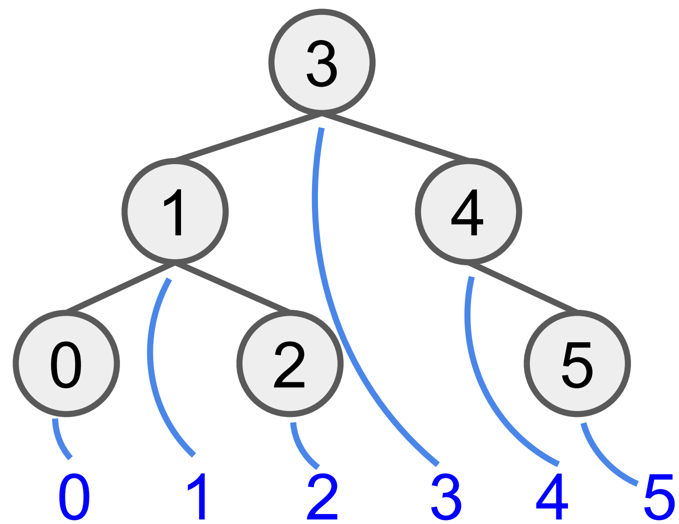
Recall from Tutorial 4 self-read slides (57-60):

- In-order traversal performs operations on the vertex after completing the left subtree, but before commencing the right subtree
- When performed on a BST, In-order traversal will *operate on* each vertex “in order” (i.e. in sorted order)

In-order Traversal

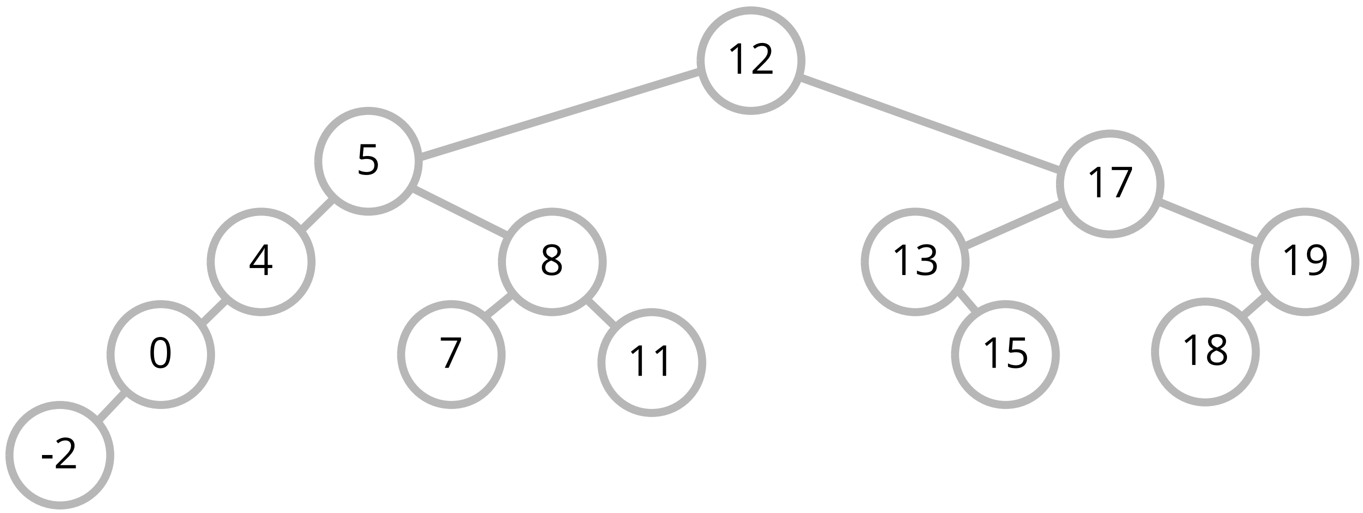
1. Recurse left
2. Current operation
3. Recruse right

```
void in_order(vertex v) {
    if (v) {
        in_order(v.left);
        cout << v.value << " ";
        in_order(v.right);
    }
}
```



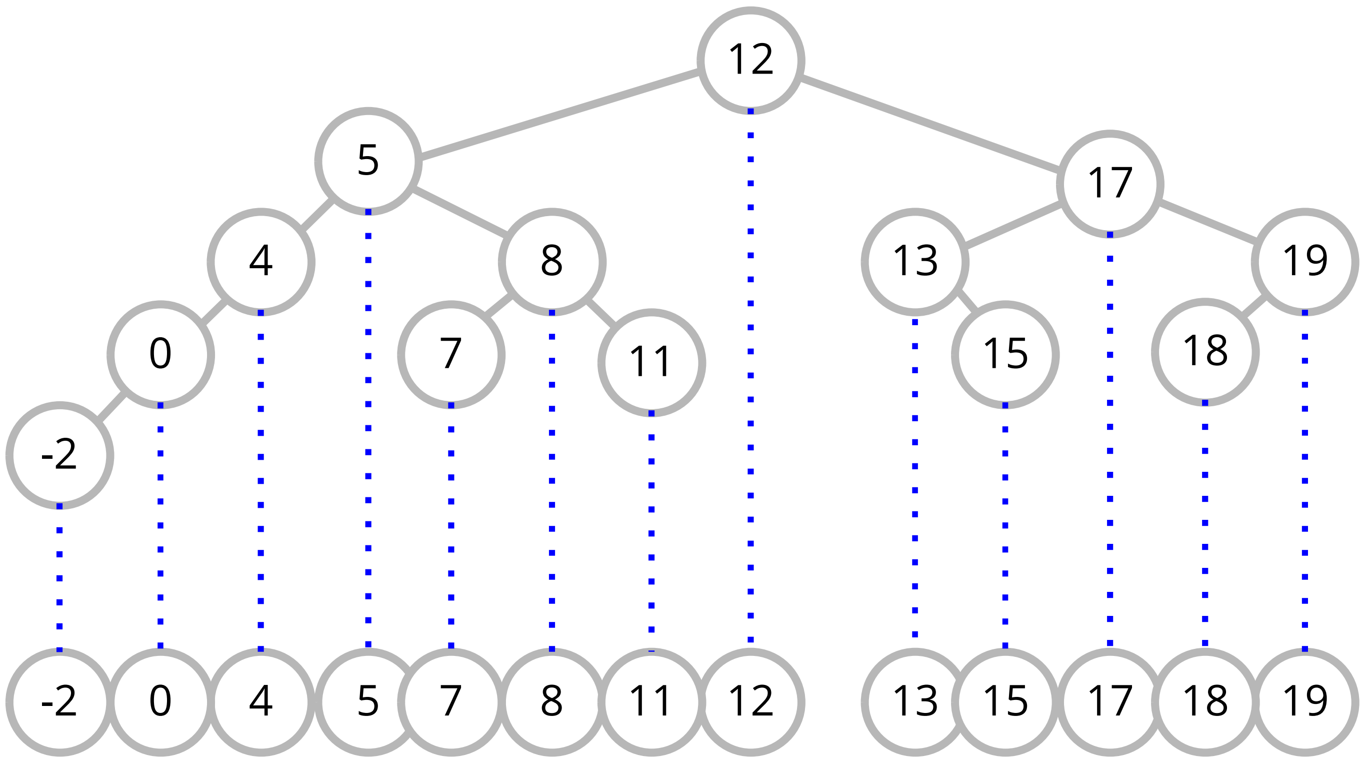
Print order in blue
Resembles "downward projection" or "flattening" of the tree!

Test yourself!



What's the sequence of operations by in-order traversal on this BST?

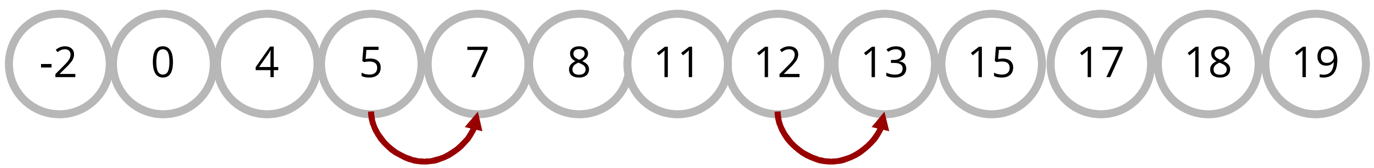
Test yourself!



Finding successor

To find successor of a key **k**, it's equivalent to finding:

- The *next highest* key
- *The next* vertex after **k** in in-order traversal sequence

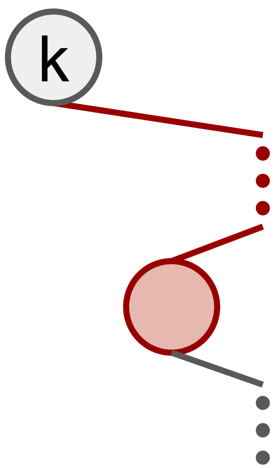


Finding the successor of keys 5 and 12 respectively

Finding successor

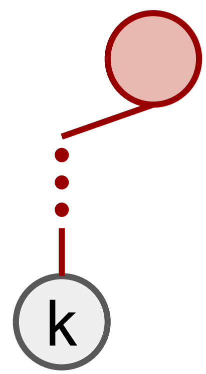
Case 1: Has **right** child:

- **Leftmost** vertex in **right** subtree

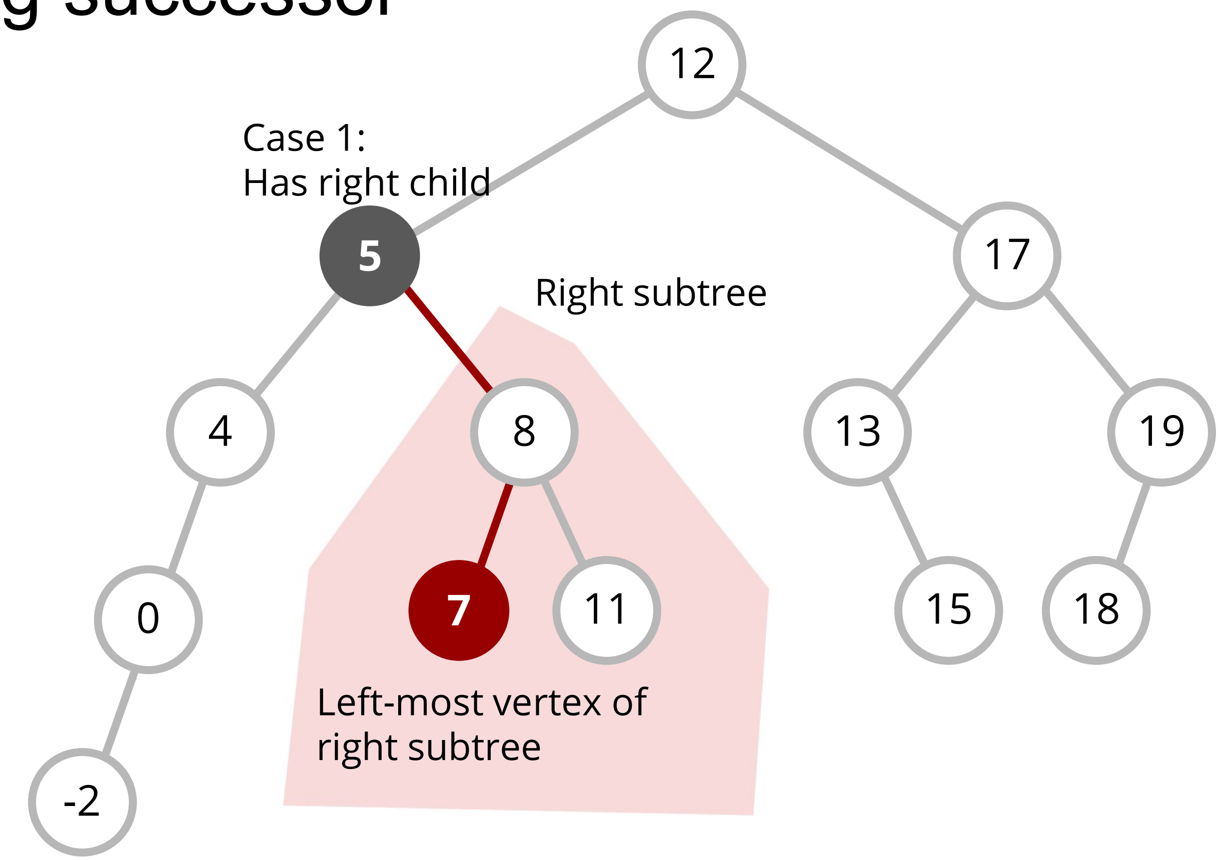


Case 2: No **right** child

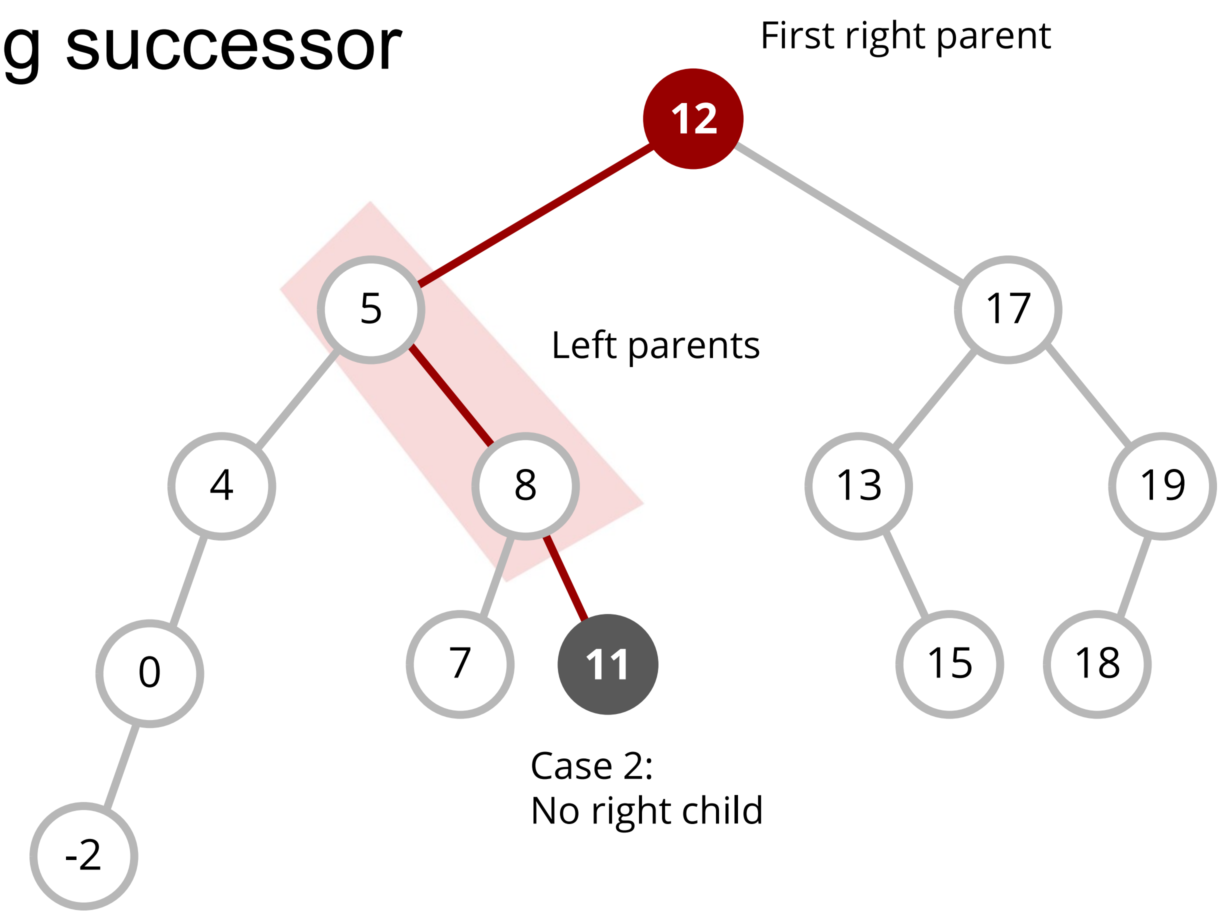
- First **right** parent
- What if there isn't a **right** parent?



Finding successor



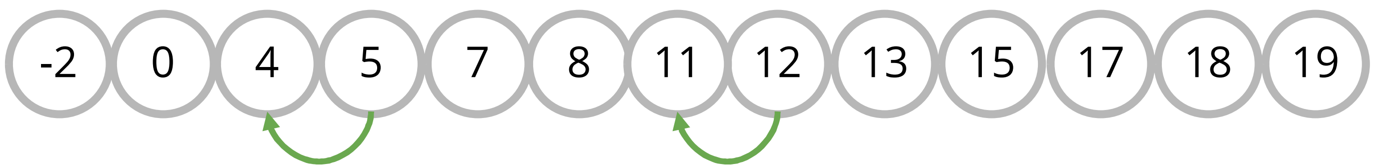
Finding successor



Finding predecessor

To find predecessor of a key **k**, it's equivalent to finding:

- The *previous largest* key
- *The previous* vertex of **k** in in-order traversal sequence

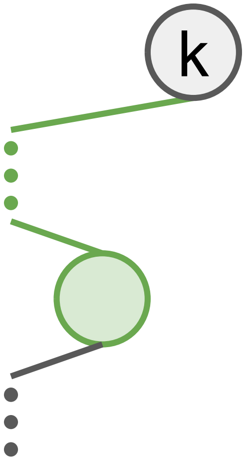


Finding the predecessor of keys 5 and 12 respectively

Finding predecessor

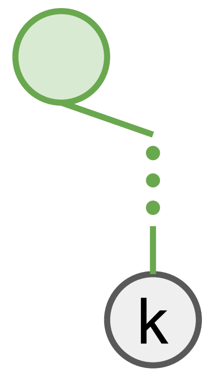
Case 1: Has **left** child:

- **Rightmost** vertex in **left** subtree

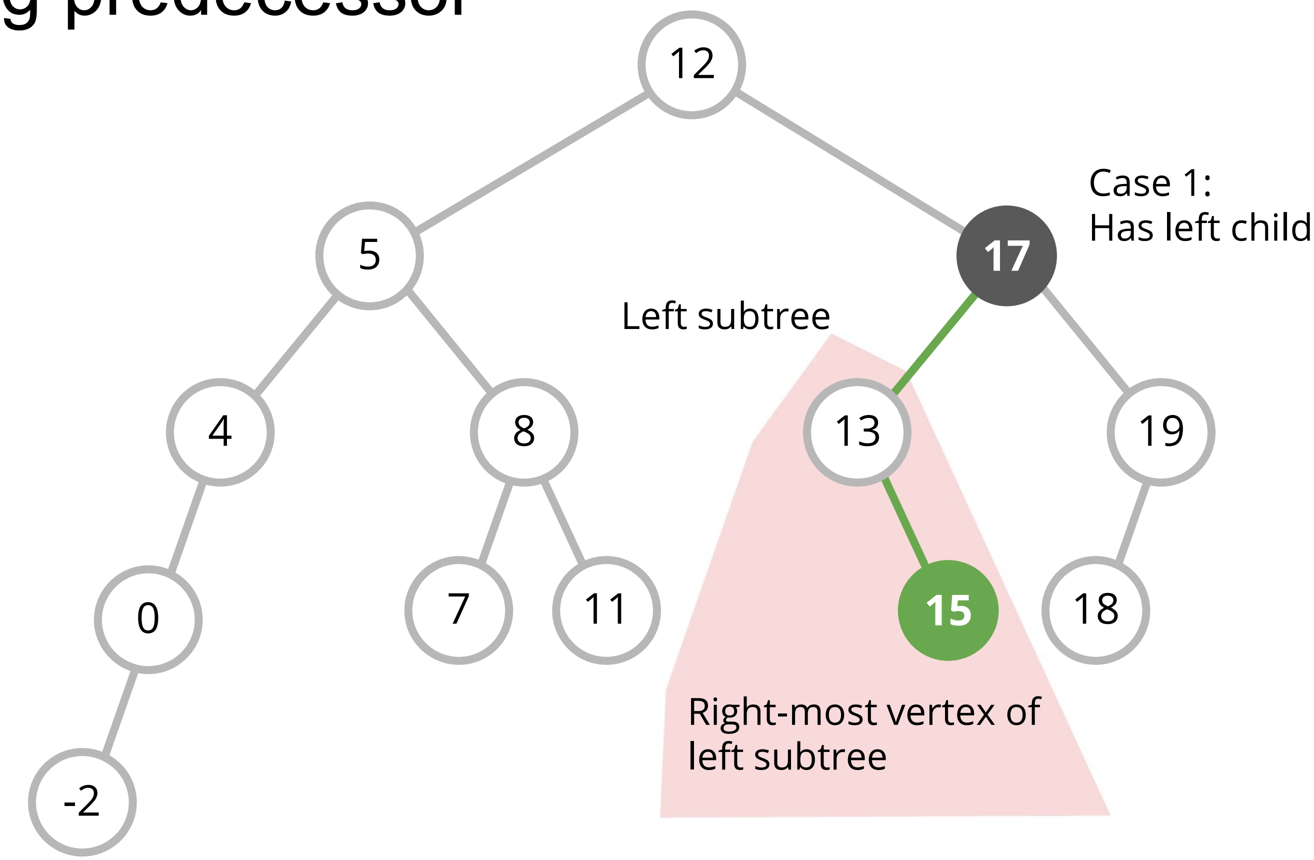


Case 2: No **left** child

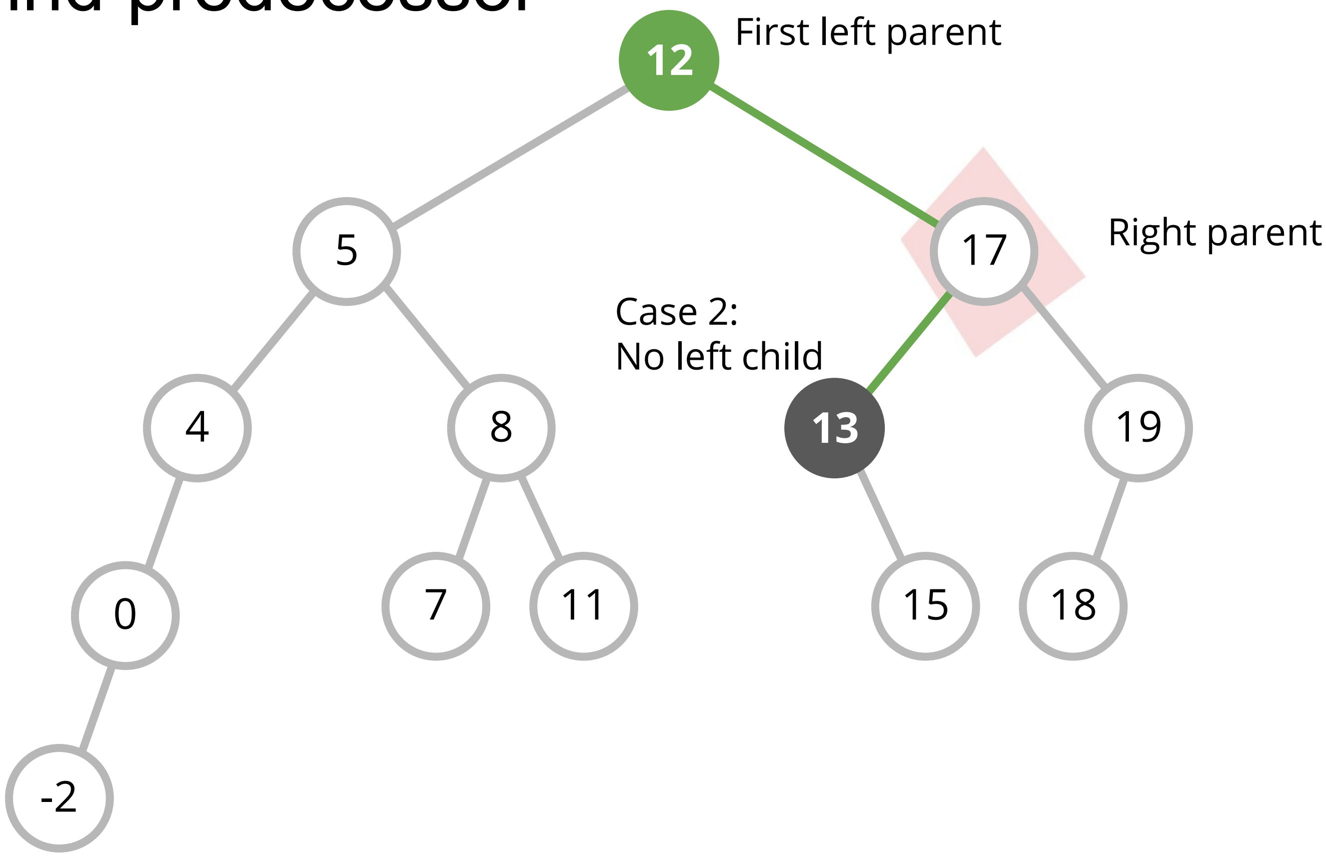
- First **left** parent
- What if there isn't a **left** parent?



Finding predecessor



BST: Find predecessor



In-order traversal??

Assume

```
set<int> s{11,15,13,-2,0,12,8,5,4,19,17,18,7};
```

What is the code *below* doing besides printing all the numbers? What's the time complexity? *O(N)* or *O(N log N)*?

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << endl;
```

Note: `auto` is preferred over `set<int>::iterator`

In-order traversal??

Assume

```
set<int> s{0, 1, 2, 3, 4, 5, …, N-1};
```

What is the time complexity of this code? *O(N)* or *O(N log N)*?

```
auto median = s.find(N/2);
for (int i = 0; i < N; i++) {
    auto it = median;
    it++; // O(log N) or O(1)?
    cout << *it << endl;
}
```

Printing the successor of median vertex at every iteration

Range based for-loop

The 2 methods below for in-order traversal of `set<int> s` are equivalent

```
for (auto &i : s) { // Preferred
    // auto here is int
    cout << i << endl;
}
```

```
for (auto it = s.begin(); it != s.end(); it++) {
    // auto here is set<int>::iterator
    cout << *it << endl;
}
```

Handling duplicates

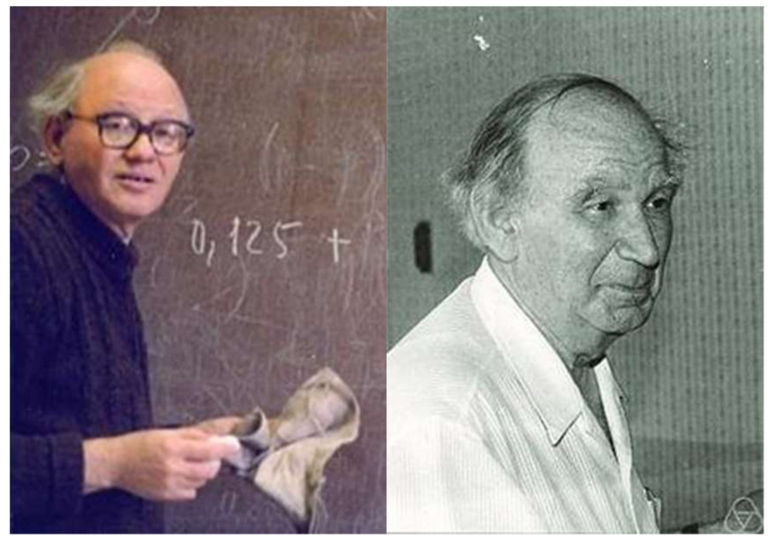
- So far for simplicity sake we assumed the values stored in BST are unique
- How shall we handle duplicate values?
- One reasonable approach is to include an additional attribute for each vertex to **count** the number of times the value was encountered. i.e. keep a frequency counter for every vertex

Balanced BST

AVL Tree

Did you know?

The AVL tree is named after its two Soviet inventors, Georgy ***A****delson-*___V___*elsky and Evgenii* ***L****andis, who published it in their 1962 paper "An algorithm for the organization of information"*



Source: [Wikipedia]

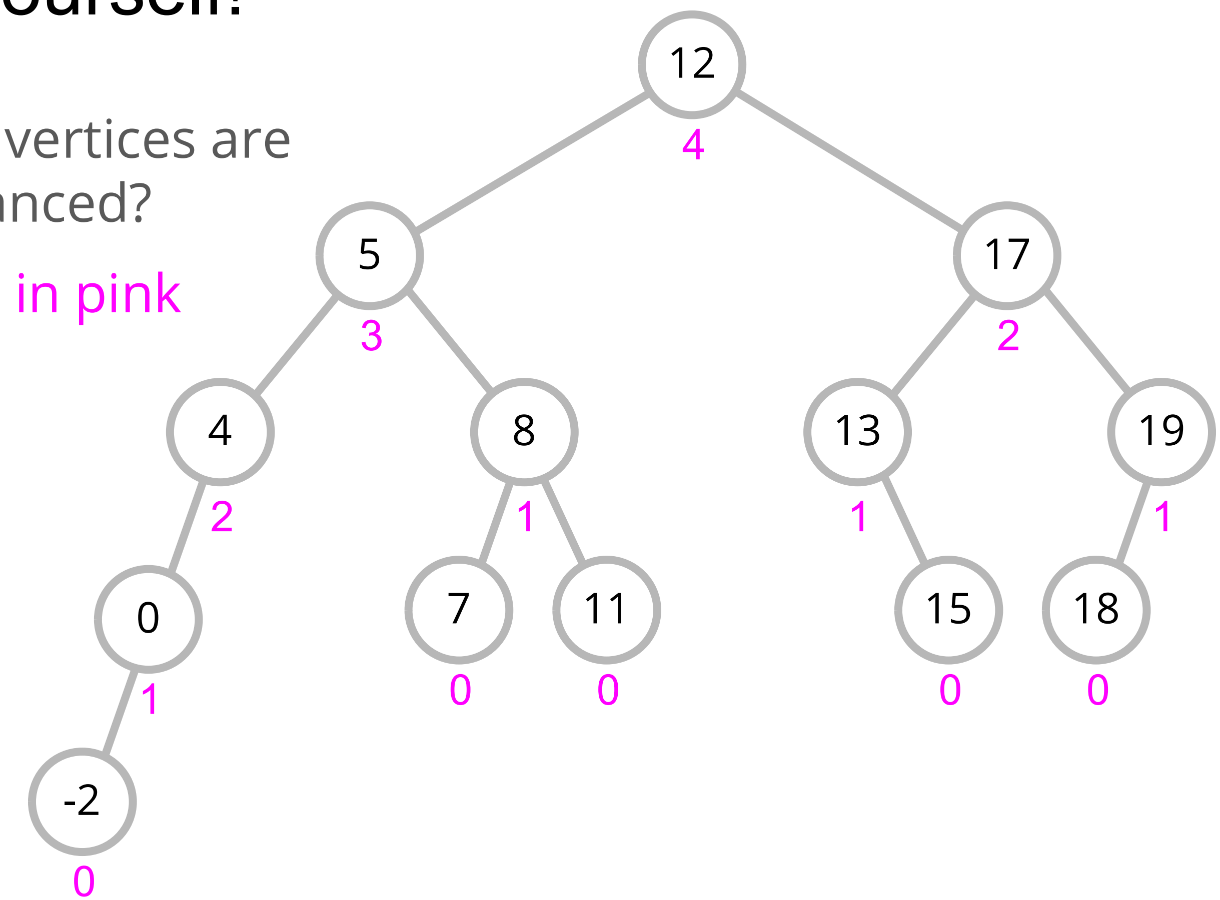
AVL Tree

- An AVL tree is a *self-balancing* BST where every vertex is *height-balanced*
- A vertex is height-balanced if difference in height between left and right subtree is not more than 1

Test yourself!

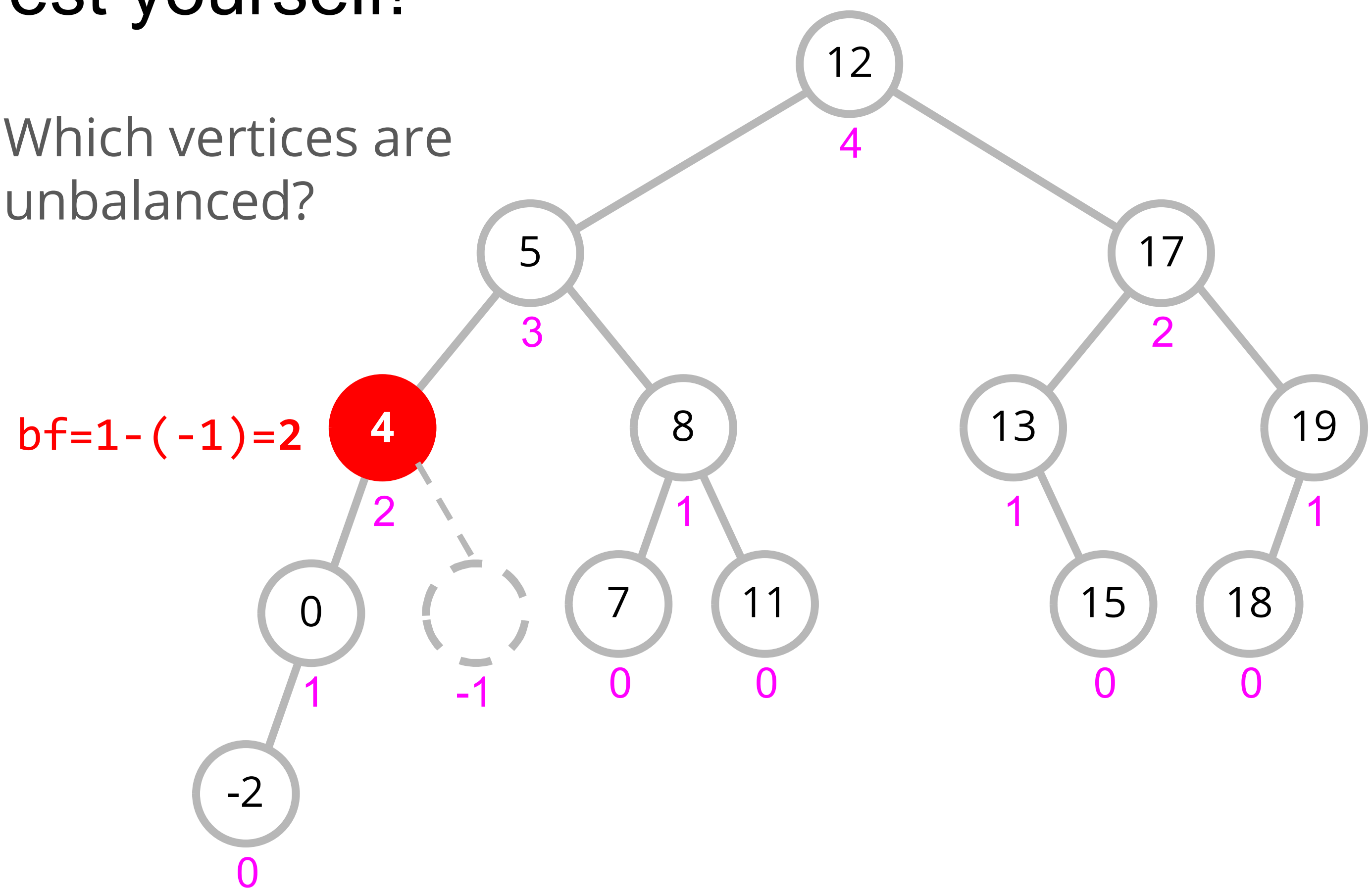
Which vertices are unbalanced?

Height in pink



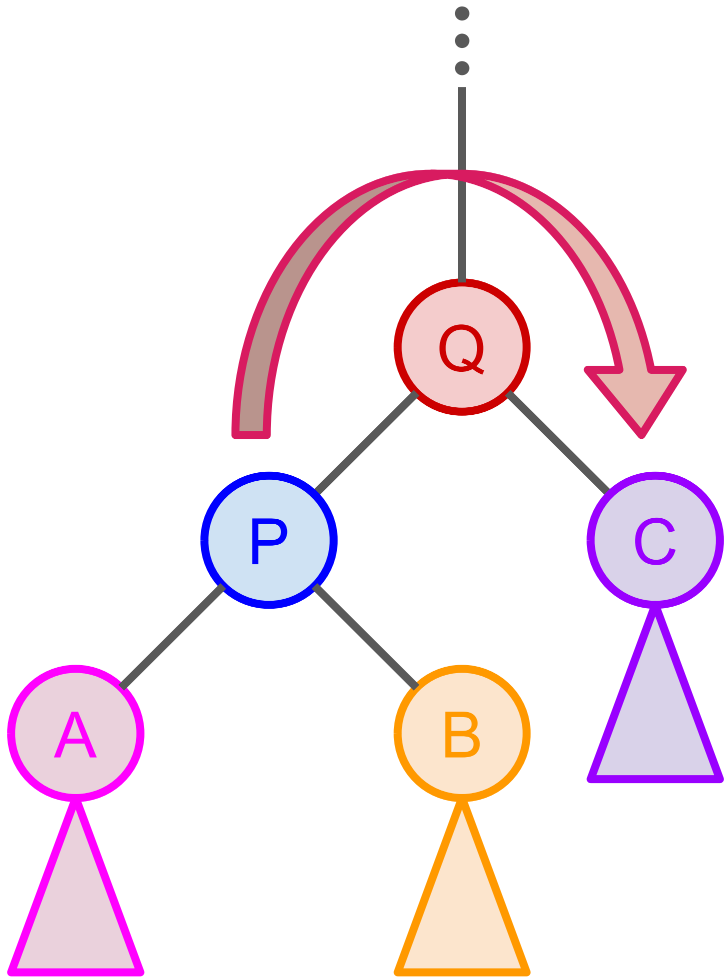
Test yourself!

Which vertices are unbalanced?



Right rotate

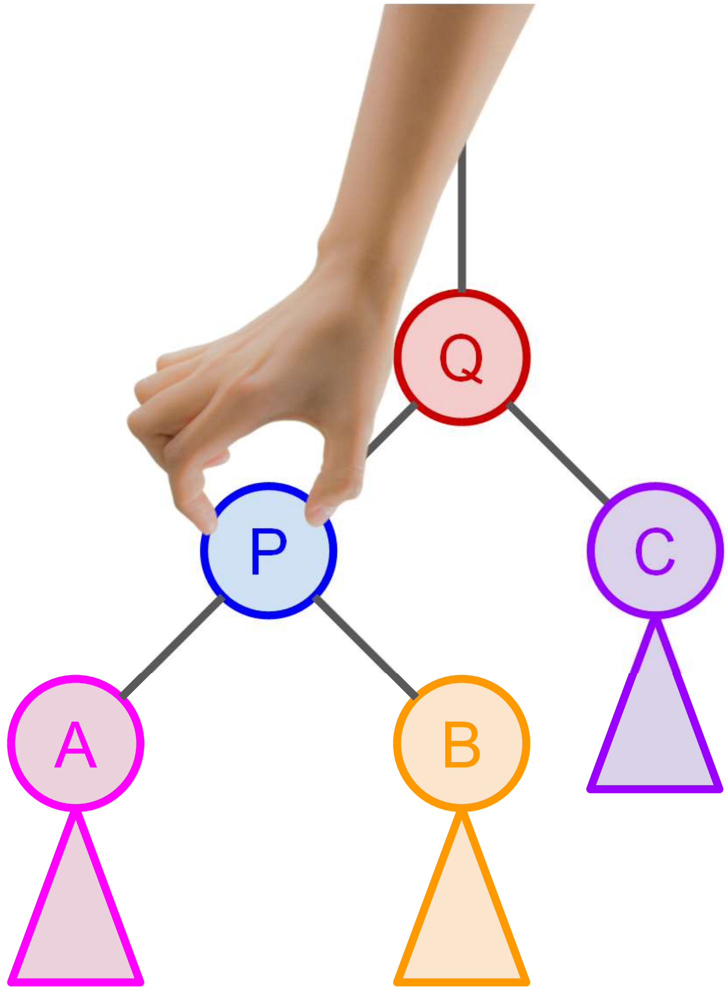
Right rotation
on Q



Right rotate

Right rotation on Q

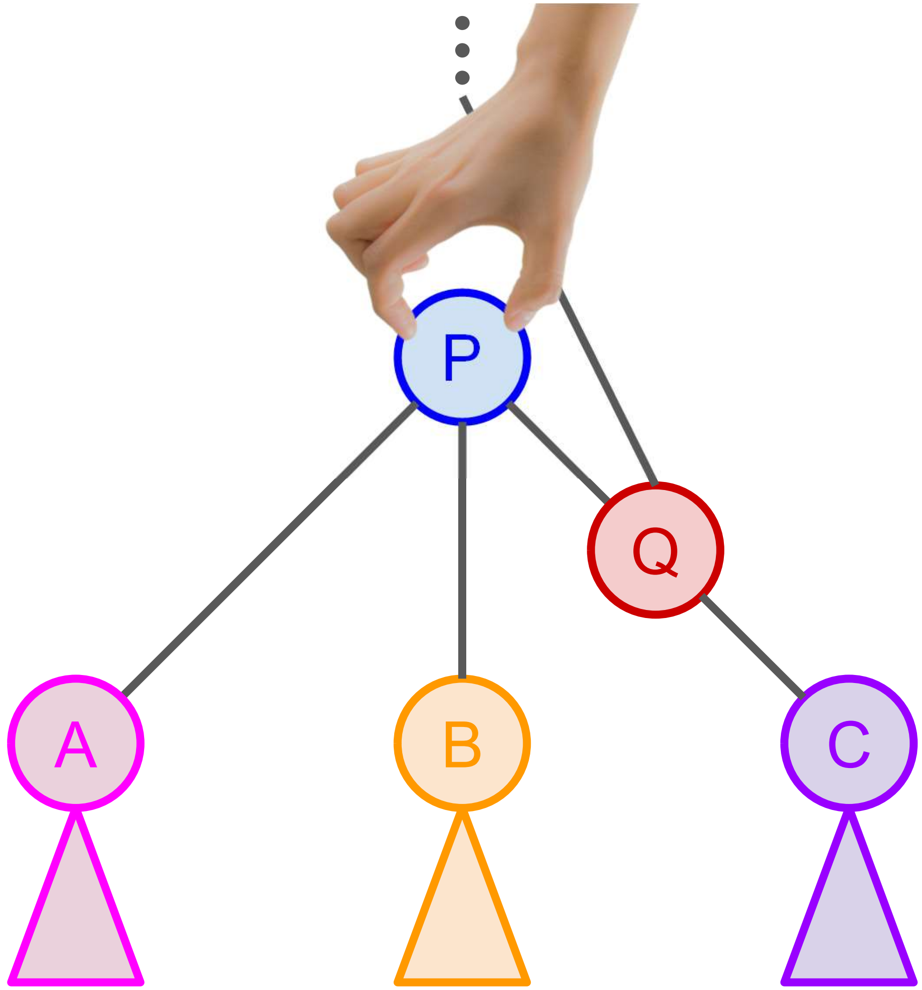
Imagine we pinch P and bring it **above and over** Q



Right rotate

Right rotation on Q

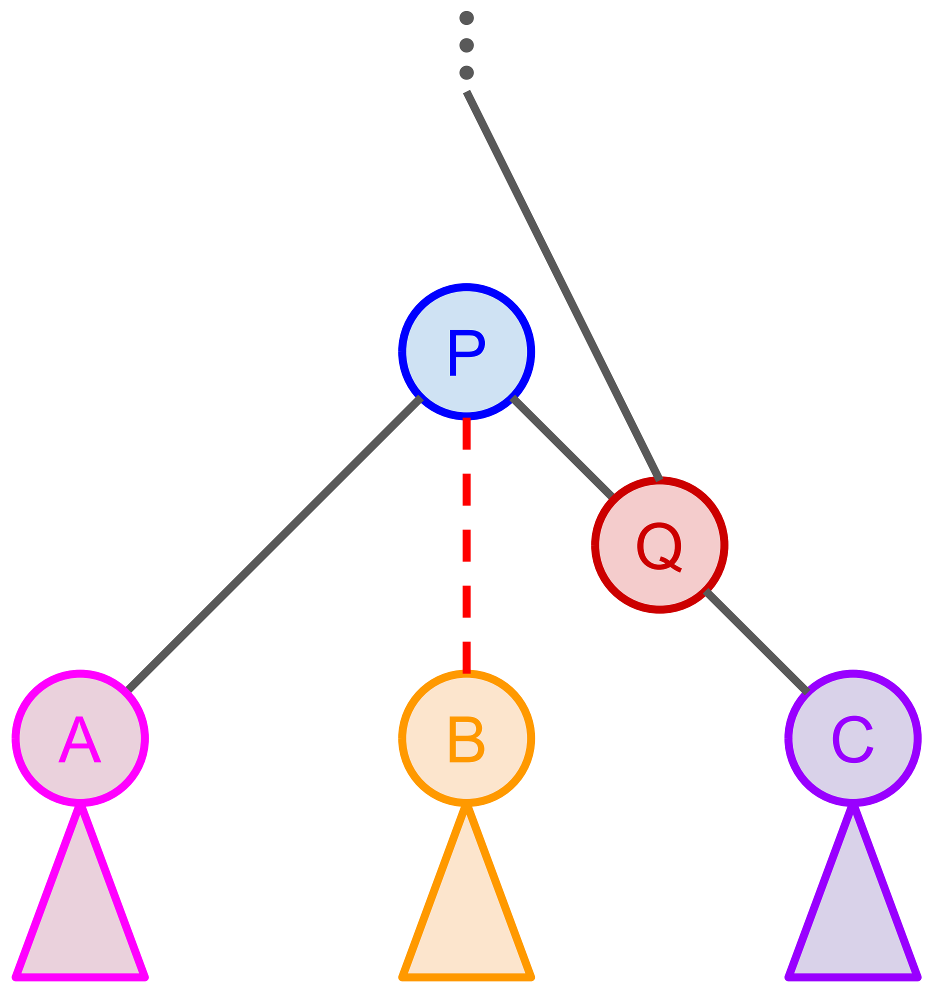
Imagine we pinch P and bring it **above and over** Q



Right rotate

Right rotation on Q

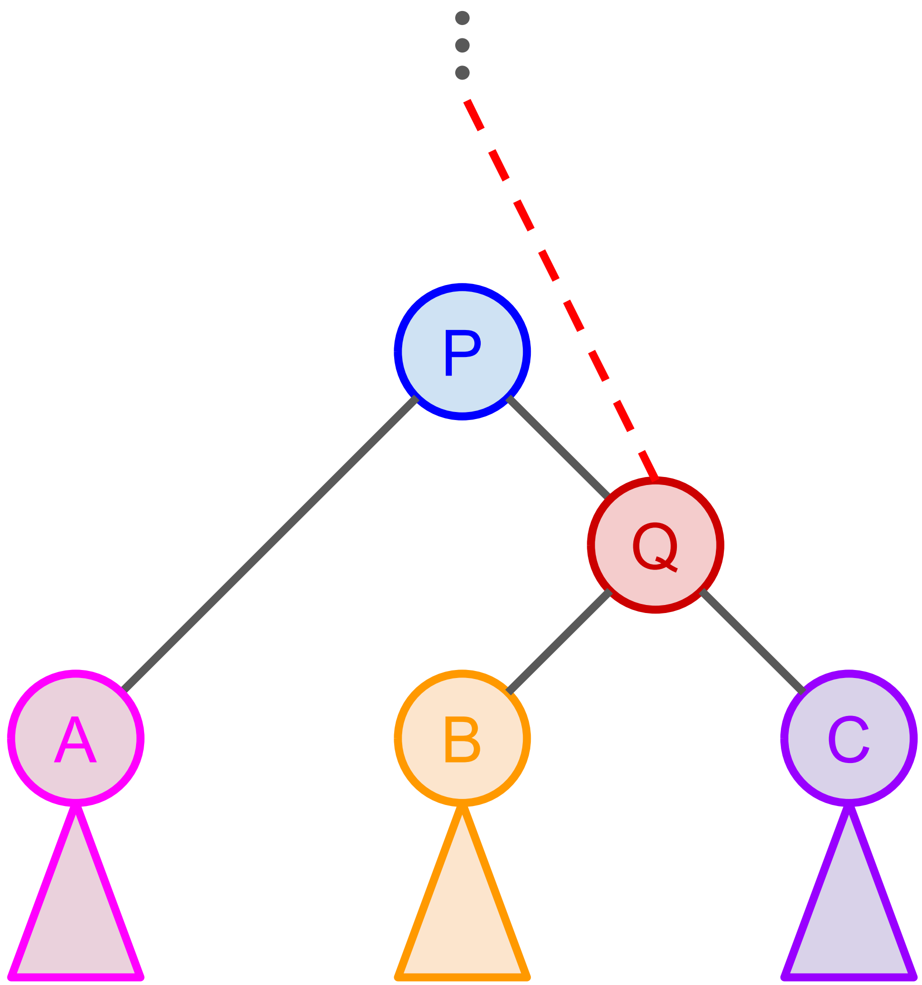
P cannot have 3 children! So we shall make subtree B the left subtree of Q.



Right rotate

Right rotation on Q

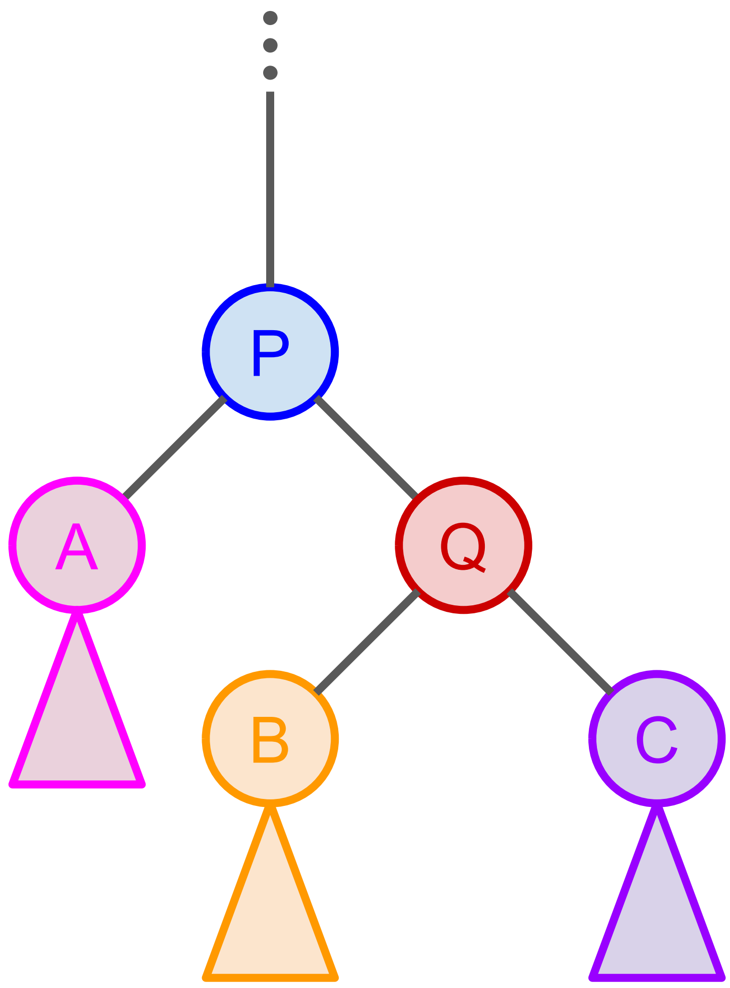
Q cannot have 2 parents and P cannot be without a parent! So we shall make the previous parent of Q the new parent of P instead.



Right rotate

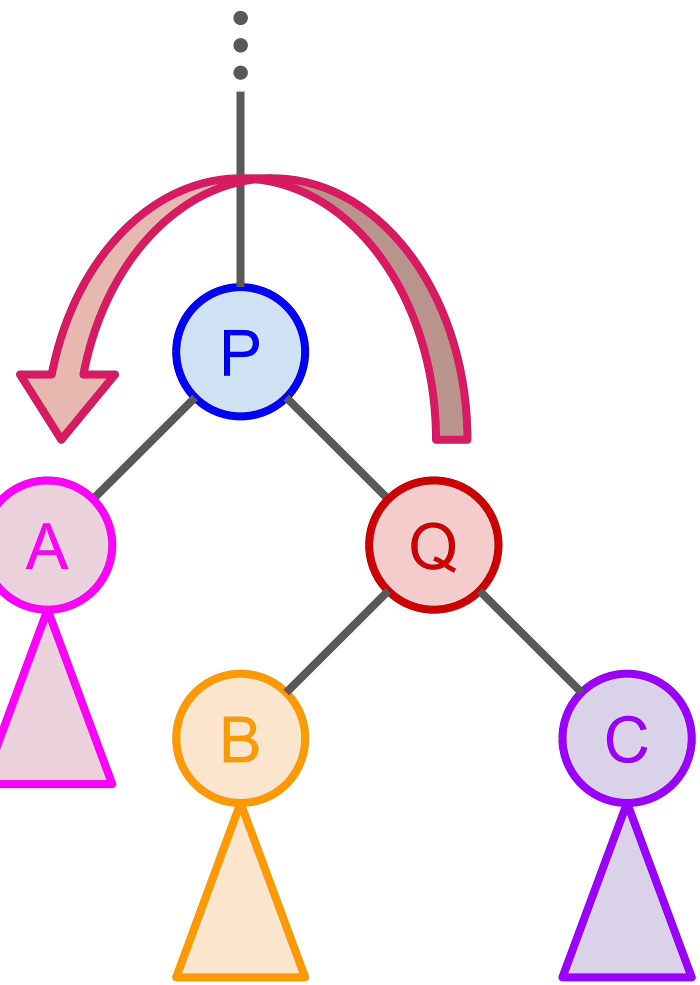
Right rotation on Q complete!

Notice A went up 1 level, C went down 1 level



Left rotate

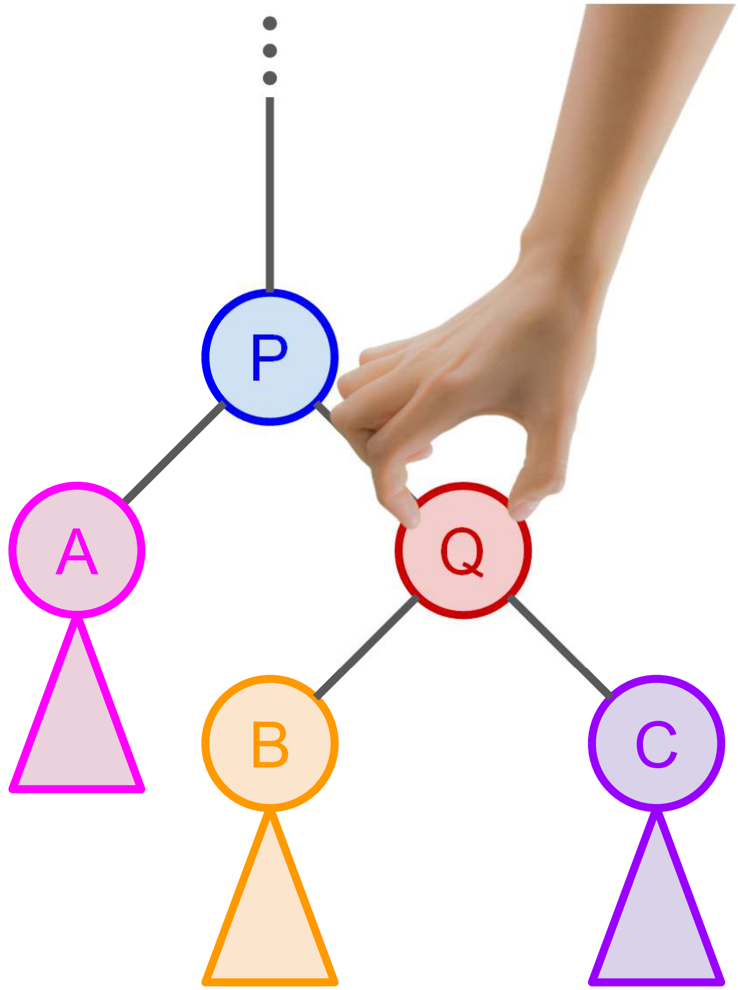
Left rotation on P



Left rotate

Left rotation on P

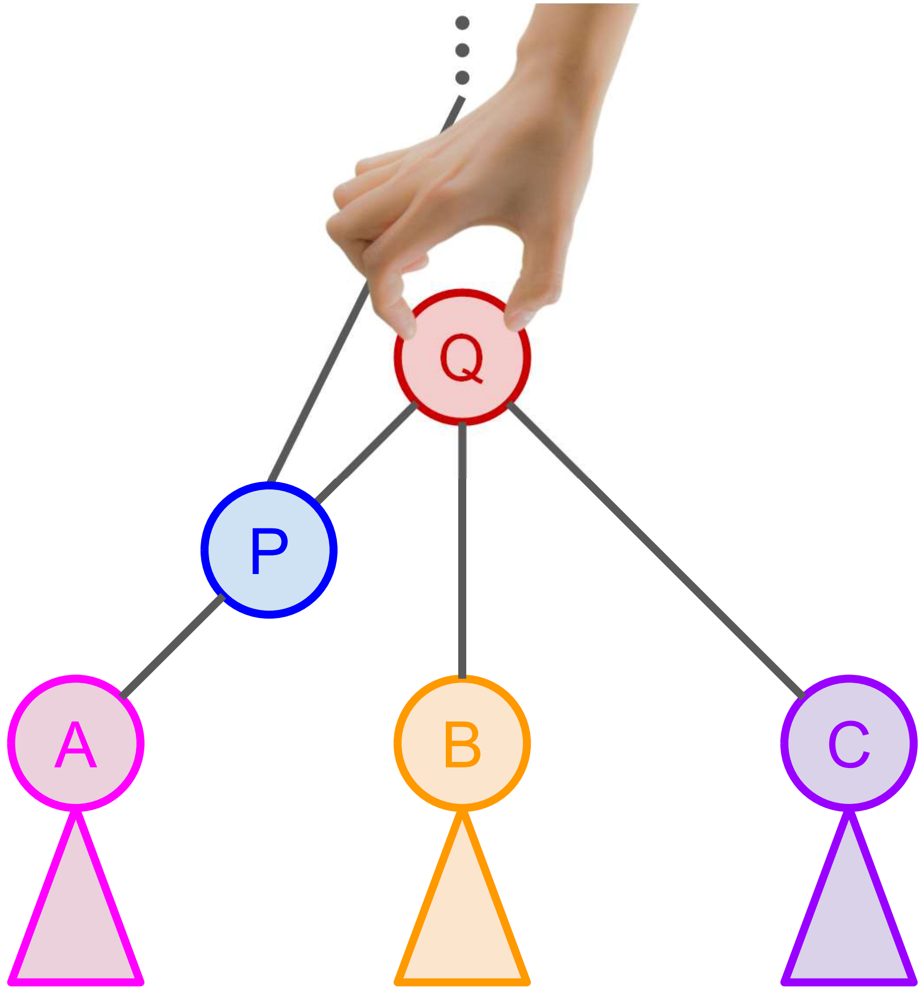
Imagine we pinch Q and bring it **above and over** P



Left rotate

Left rotation on P

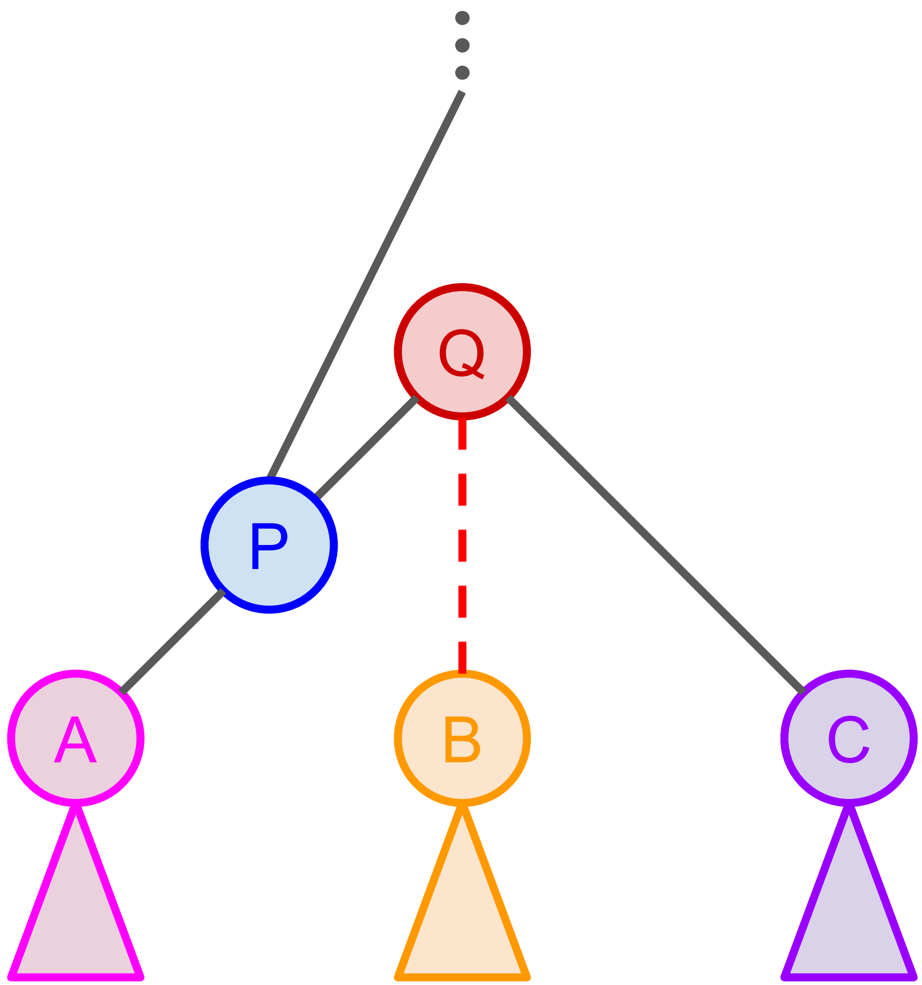
Imagine we pinch Q and bring it **above and over** P



Left rotate

Left rotation on P

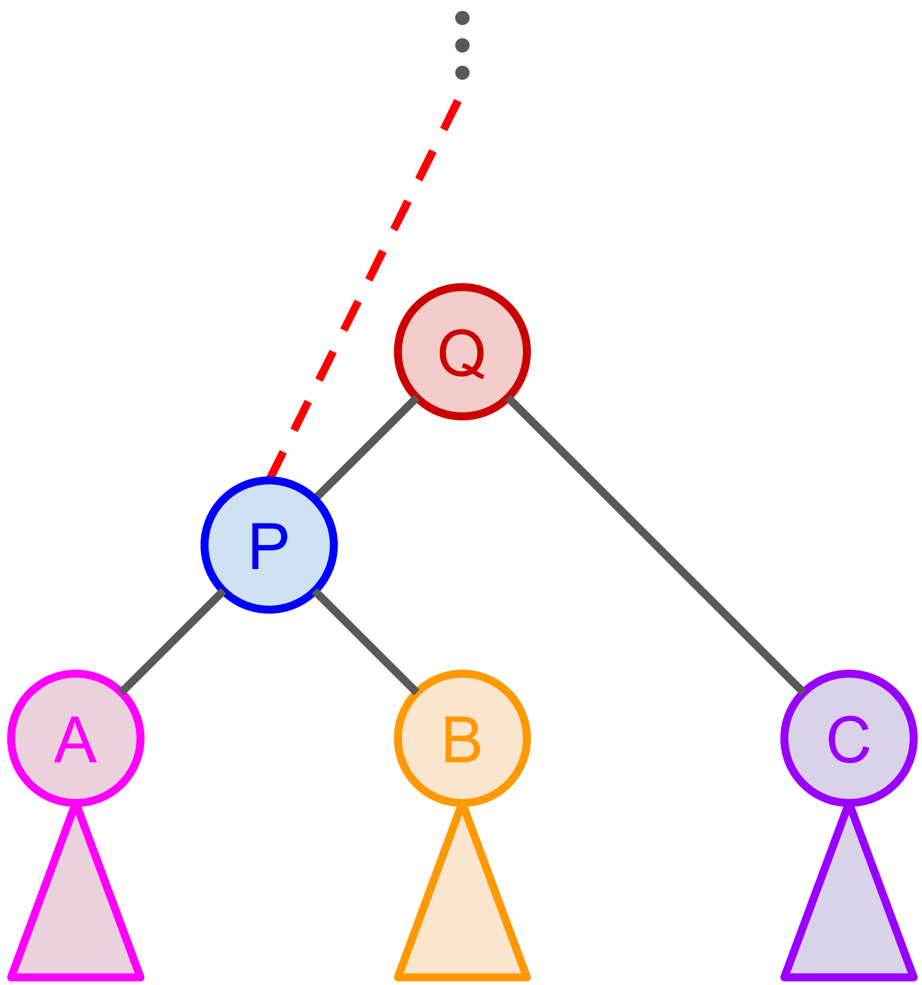
Q cannot have 3 children! So we shall make subtree B the right subtree of P.



Left rotate

Left rotation on P

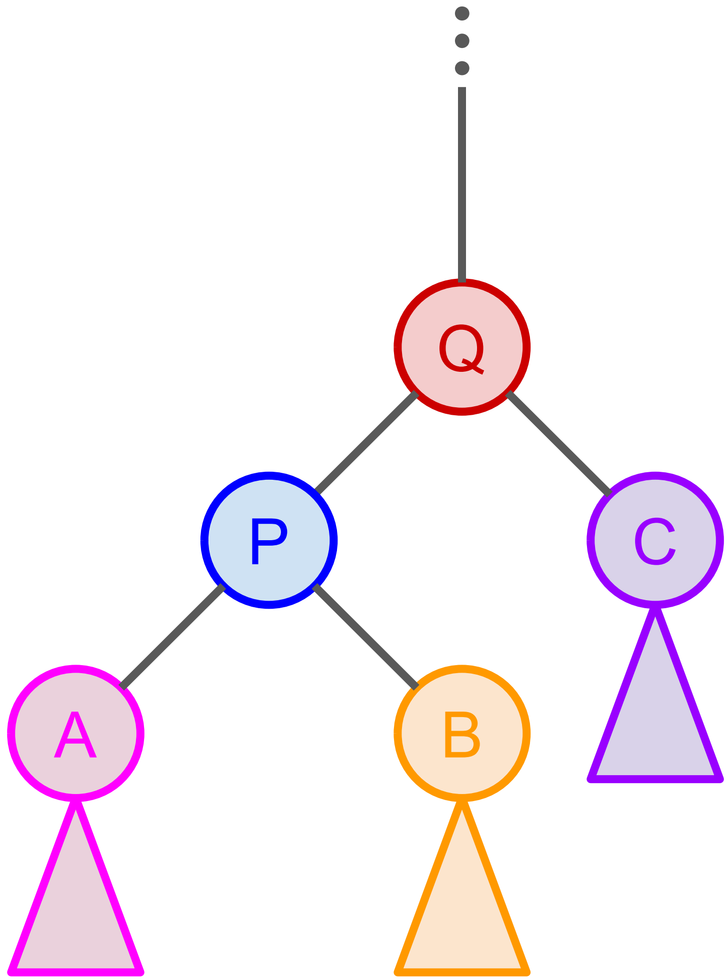
P cannot have 2 parents and Q cannot be without a parent! So we shall make the previous parent of P the new parent of Q instead.



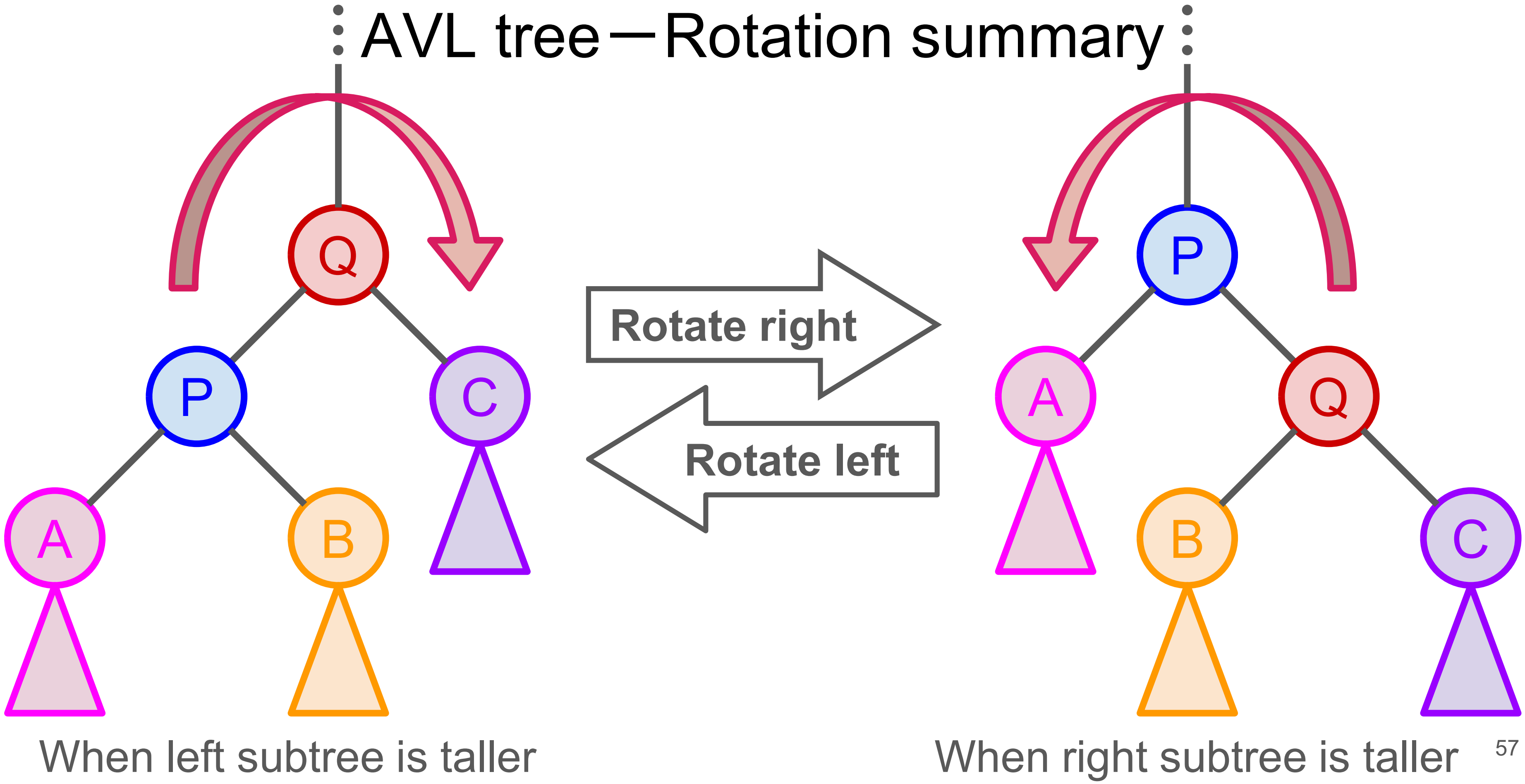
Left rotate

Left rotation on P complete!

Notice C went up 1 level, A went down 1 level



AVL tree－Rotation summary



AVL Tree—Balancing

We define *balance factor* for a vertex `v` to be:

`bf(v) = h(v.left) - h(v.right)`

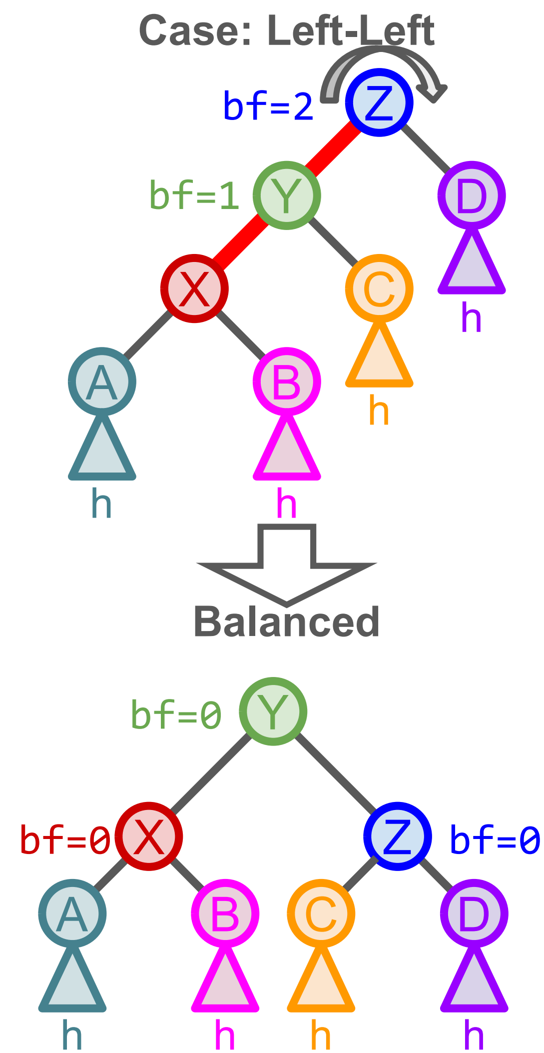
`bf(v)`	Balance type
[- 1, 1]	Balanced
>1	left subtree *taller* than right subtree
<1	left subtree *shorter* than right subtree

AVL Tree—Balancing

Observation

A right rotate will *balance* a subtree if:

- The *left* child has a taller left subtree
- We call it *left-left* case because that's the 2 turns we take to traverse down to the taller subtree
- `bf(v.left) ≥ 0`

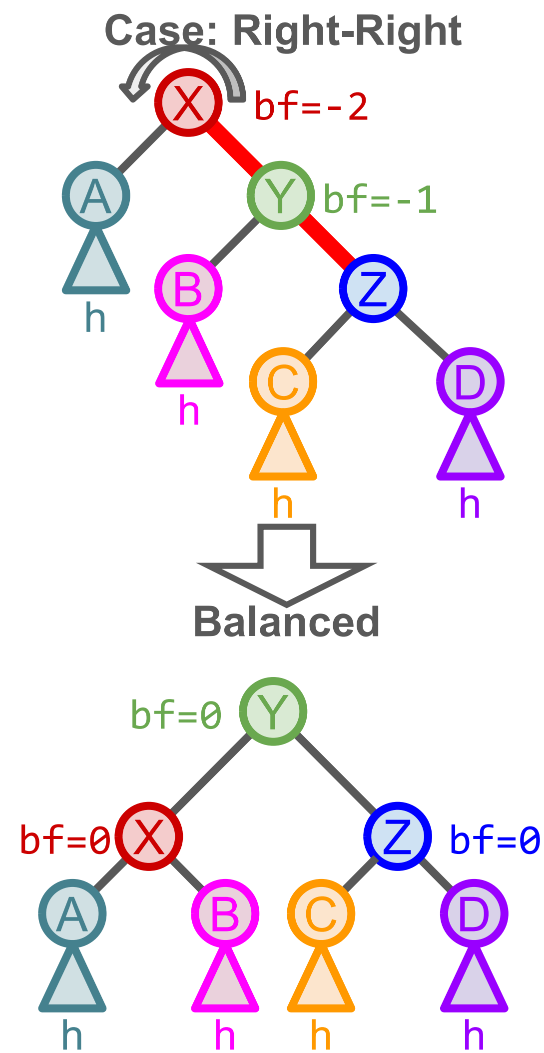


AVL Tree—Balancing

Observation

A left rotate will *balance* a subtree if:

- The *right* child has a taller right subtree
- We call it *right-right* case because that's the 2 turns we take to traverse down to the taller subtree
- **`bf(v.right) ≤ 0`**



AVL Tree—Balancing

Observation

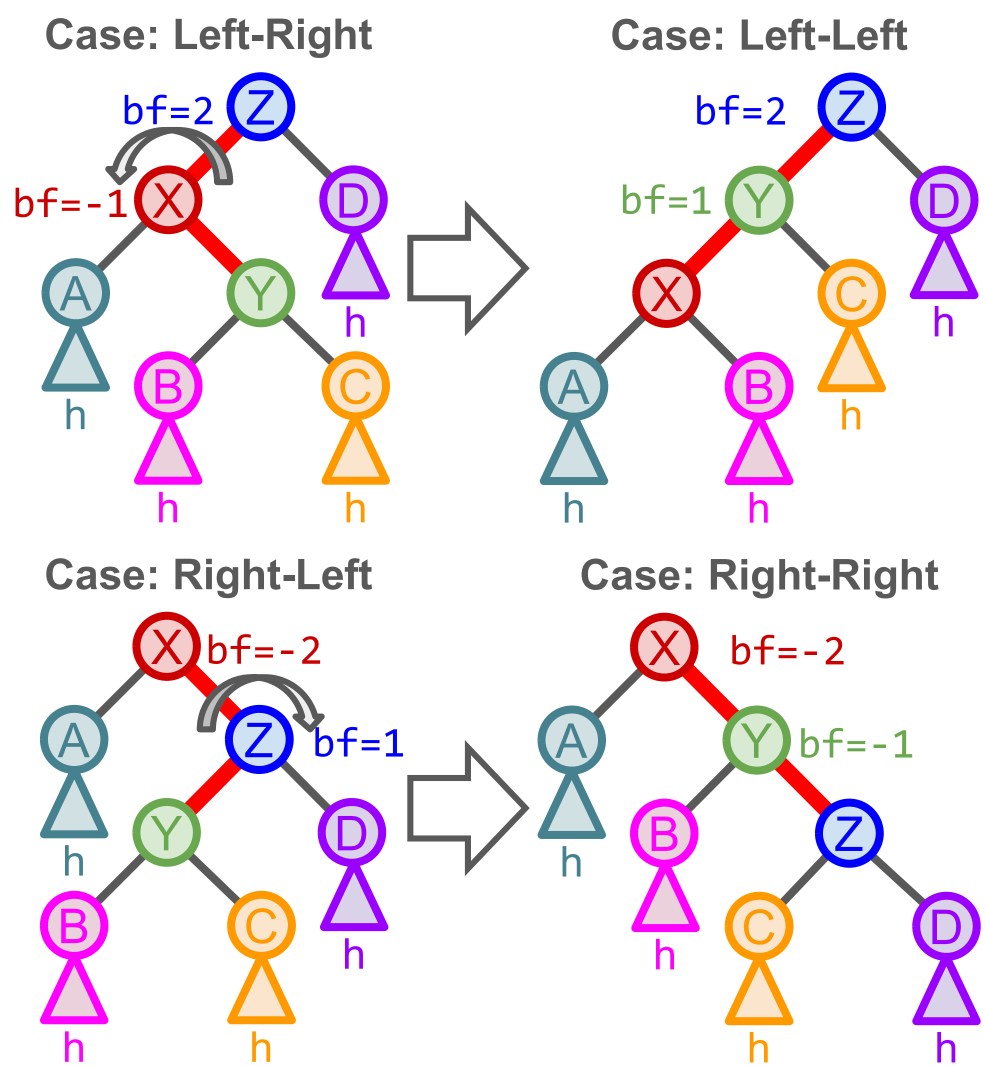
In essence, the previous 2 balancing tricks applies *iff* `bf(v)` and `bf(v.taller_child())` *do not* have opposing signs!

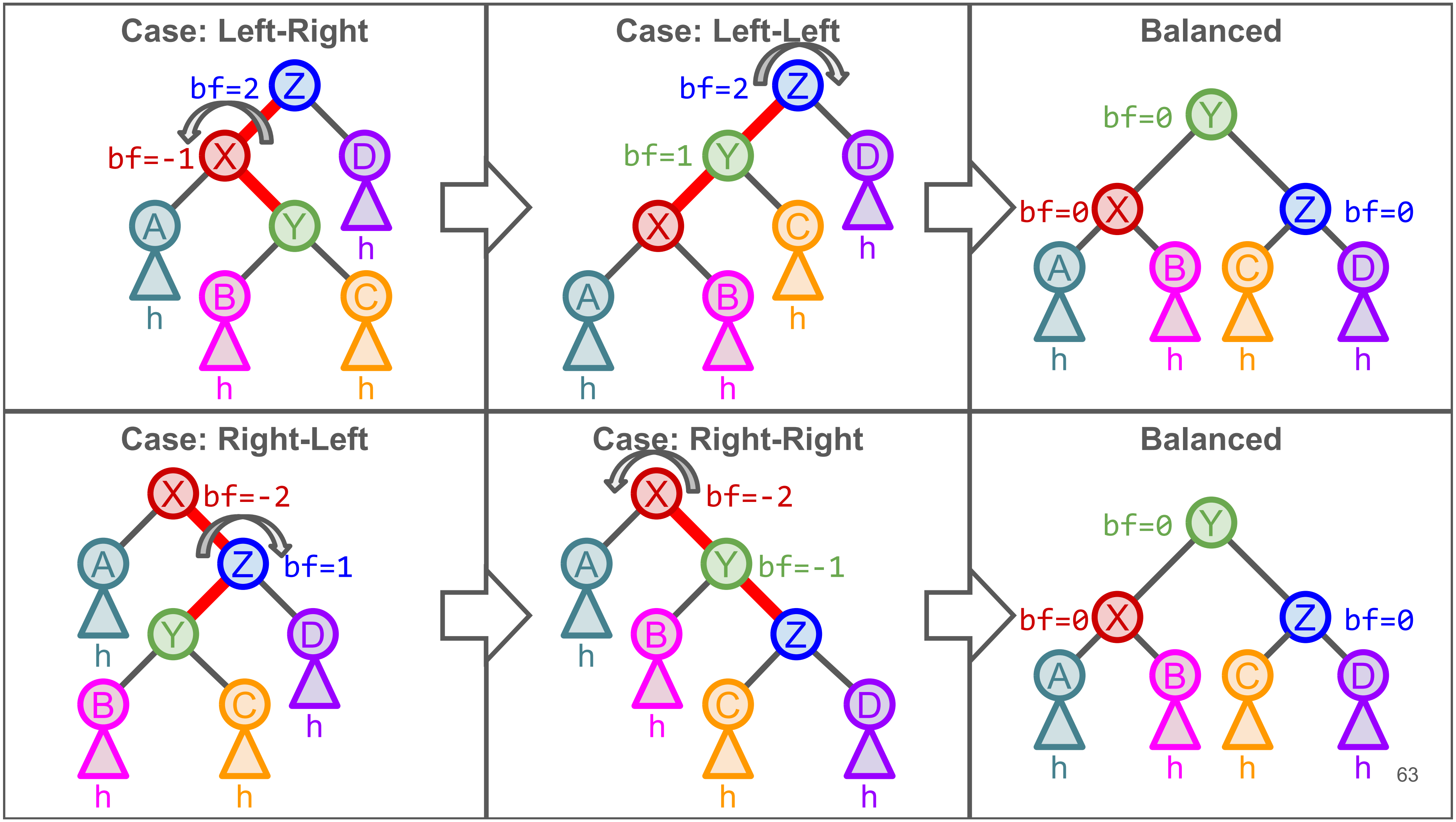
We regard `0` as neither positive or negative so they do not oppose any sign.

AVL Tree—Balancing

Observation

If `bf(v.taller_child())` has an opposite sign to `bf(v)`, we can **rotate** `v.taller_child()` to transform its `bf` into the desired sign!




Case: Left-Right
bf=2
Z
bf=-1
X
D
A
Y
B
C
h
h
h
h
Case: Left-Left
bf=2
Z
bf=1
Y
D
X
C
A
B
h
h
h
h
Balanced
bf=0
Y
bf=0
X
Z
bf=0
A
B
C
D
h
h
h
h
Case: Right-Left
X
bf=-2
A
Z
bf=1
Y
D
B
C
h
h
h
h
Case: Right-Right
X
bf=-2
A
Y
bf=-1
B
Z
C
D
h
h
h
h
Balanced
bf=0
Y
bf=0
X
Z
bf=0
A
B
C
D
h
h
h
h

AVL Tree－Balancing summary

`bf(v)`	`bf(v.left)`	`bf(v.right)`	Case	Rotation
$[-1, 1]$	Don’t care	Don’t care	Balanced	NIL
>1	≥ 0	Don’t care	Left-Left	`rotate_right(v)`
	< 0		Left-Right	`rotate_left(v.left)`
<1	Don’t care	≤ 0	Right-Right	`rotate_left(v)`
		> 0	Right-Left	`rotate_right(v.right)`

Question 2

Problem statement

Draw a valid AVL Tree and nominate a vertex to be deleted such that if that vertex is deleted:

a. No rotation happens
b. Exactly one of the four rotation cases happens
c. Exactly two of the four rotation cases happens (you cannot use the sample given in VisuAlgo which is https://visualgo.net/en/bst?mode=AVL&create=8,6,16,3,7,13,19,2,11,15,18,10, delete vertex 7; think of your own test case)

Test yourself!

Recall that an unbalanced subtree is the condition for rebalancing rotation(s).

After a subtree has been re-balanced with rotation(s), which other vertices do we have to check for imbalances?

Question 2: Deletion

What are the 3 cases for deletion?

- Case 1: Is leaf vertex
  - Just remove it
- Case 2: Has one child
  - Connect the child subtree to the deleted vertex’s parent
- Case 3: Has 2 children
  - Replace with successor, delete successor instead
  - Can also use predecessor

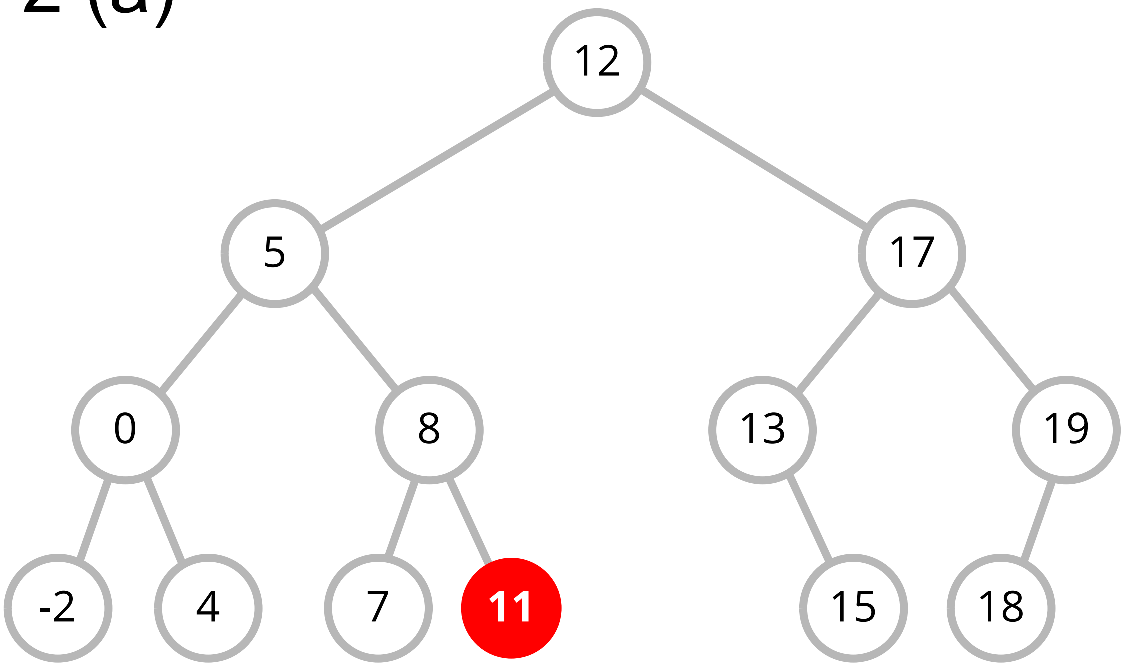
Question 2: Rotation

a. No rotation happens
   - No imbalance

b. Only one rotation case
   - Imbalance that can be *resolved* with one rotation

c. Exactly two rotation case
   - Multiple sequential rotations
   - “Skewed” BBST

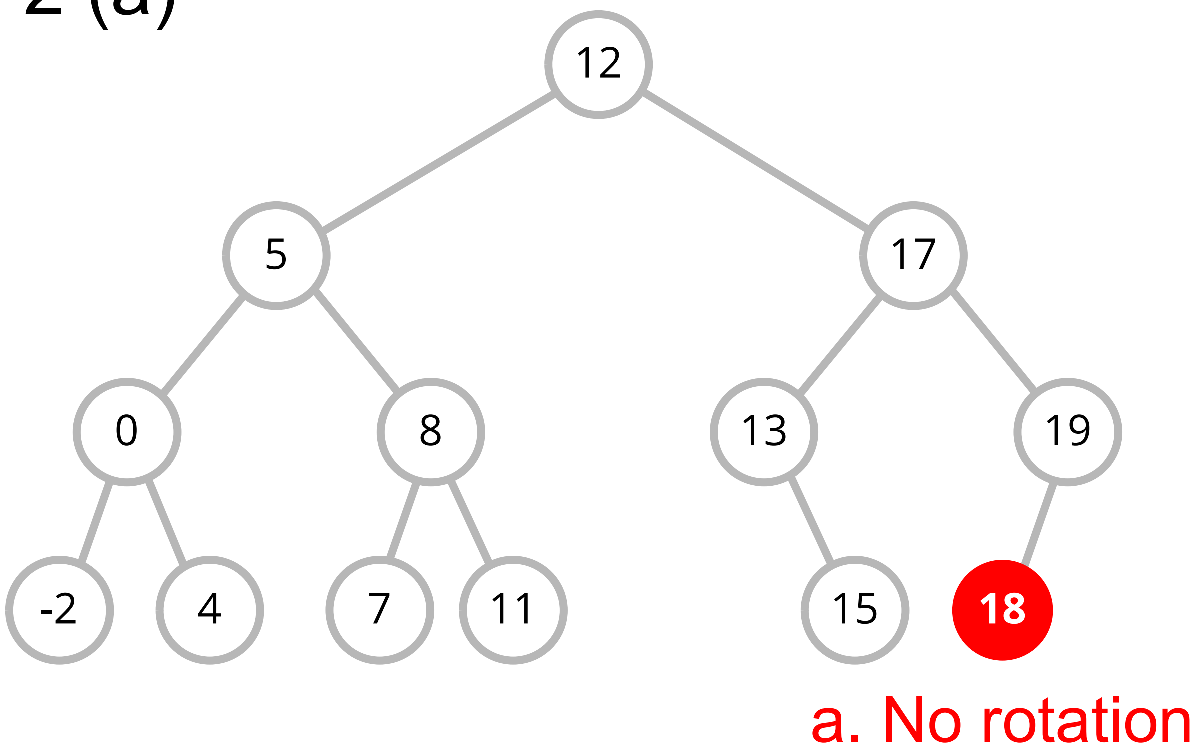
Question 2 (a)



a. No rotation

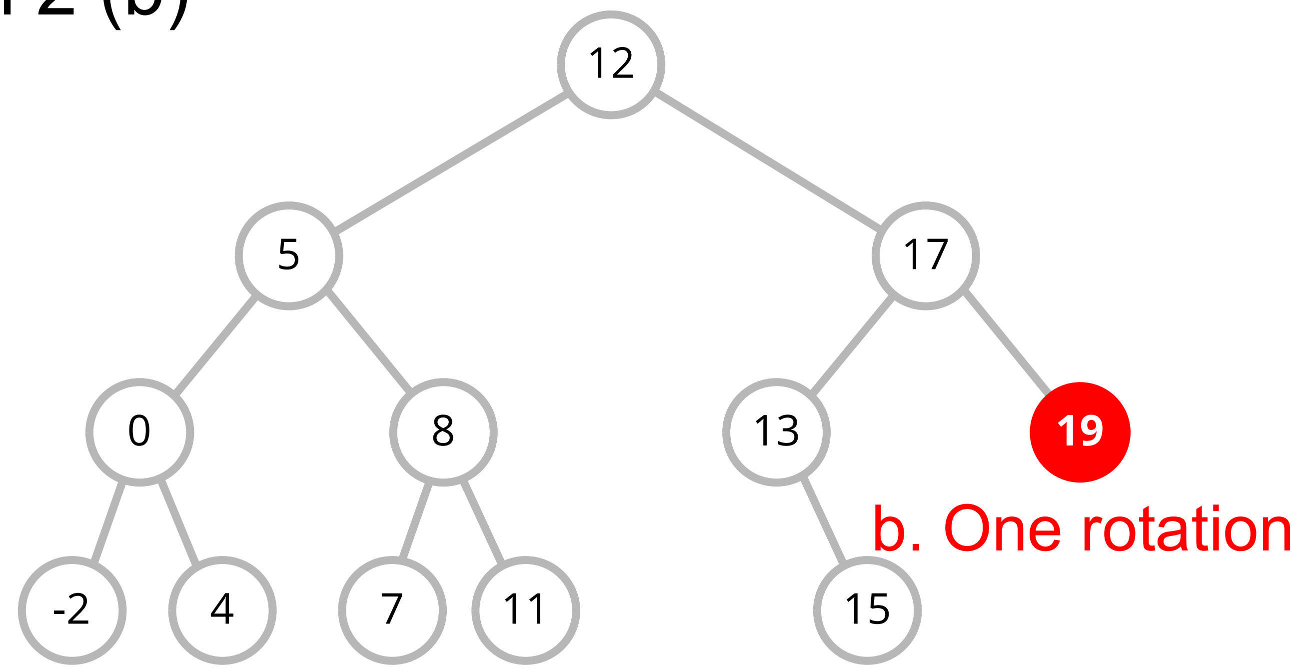
https://visualgo.net/en/bst?create=12,5,17,0,8,13,19,-2,4,7,11,15,18&mode=AVL

Question 2 (a)



https://visualgo.net/en/bst?create=12,5,17,0,8,13,19,-2,4,7,11,15,18&mode=AVL

Question 2 (b)



https://visualgo.net/en/bst?create=12,5,17,0,8,13,19,-2,4,7,11,15&mode=AVL

Question 2 (c)

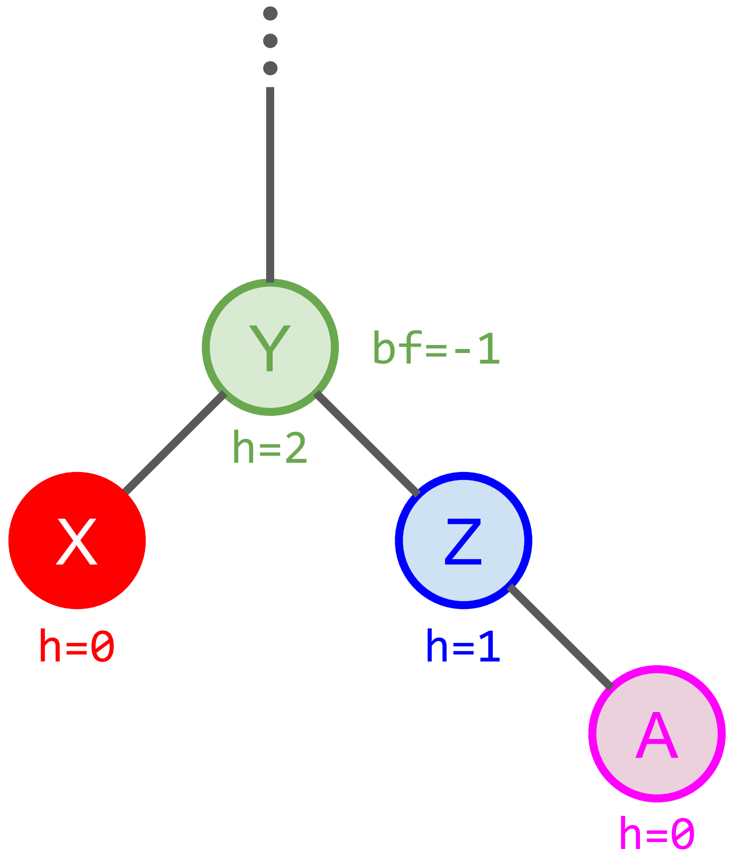
Observation

- Rebalancing a subtree can reduce its height by **at most** 1
- To get *2 rotations*, you must make it such that when you remove the vertex, *2 subtrees* will be imbalanced in the process

Question 2 (c)

We first show a scenario where rebalancing reduces the subtree's height by 1.

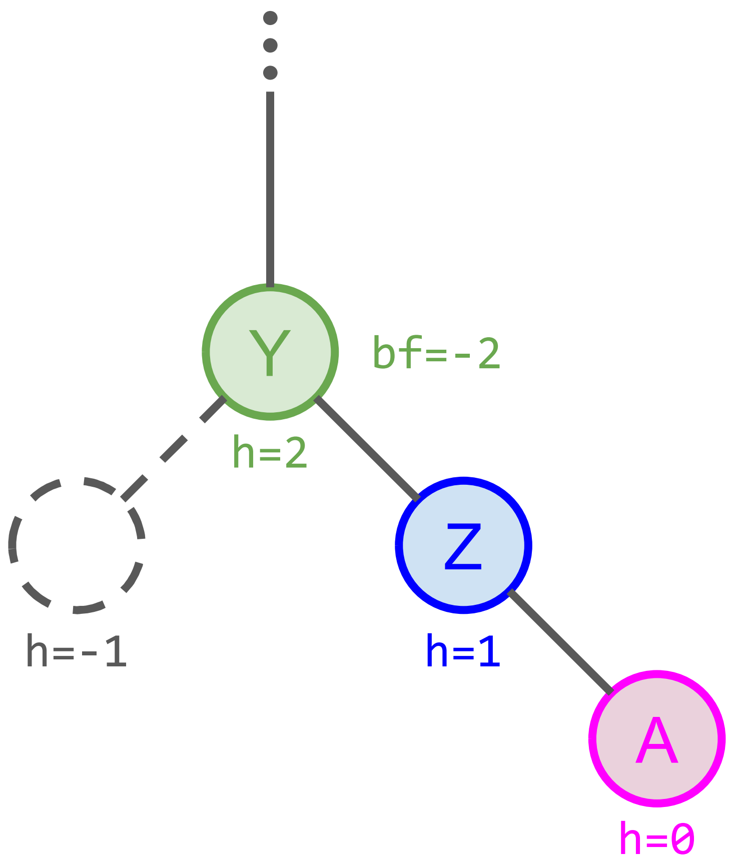
We shall remove X from this subtree Y with height 2.



Question 2 (c)

X removed.

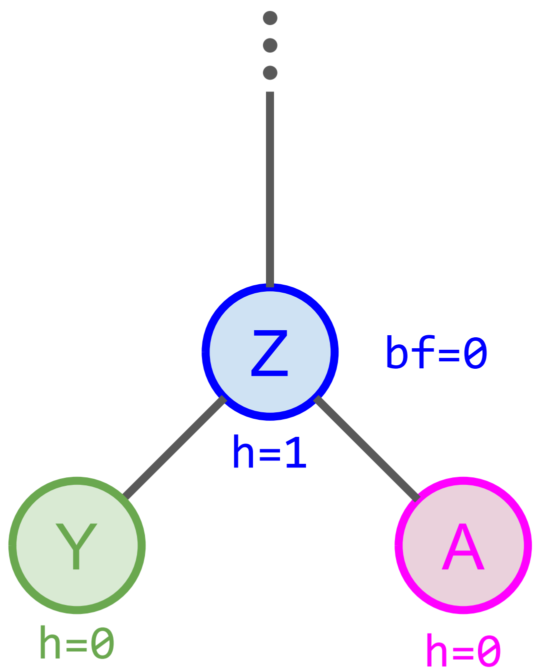
`bf(Y)` is upset.



Question 2 (c)

After rotation, this subtree has height 1.

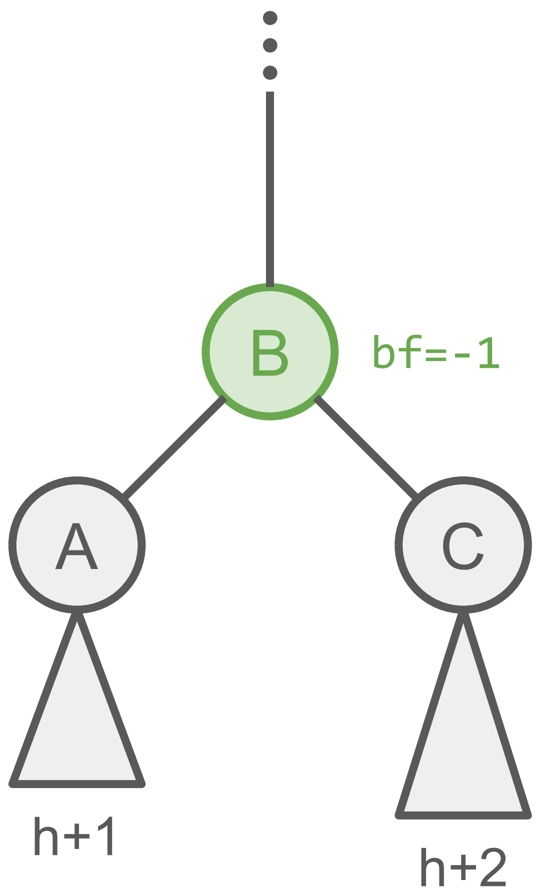
Realize the height will also drop by 1 if we simply removed A instead of X, in which case no rotation would be required.



Question 2 (c)

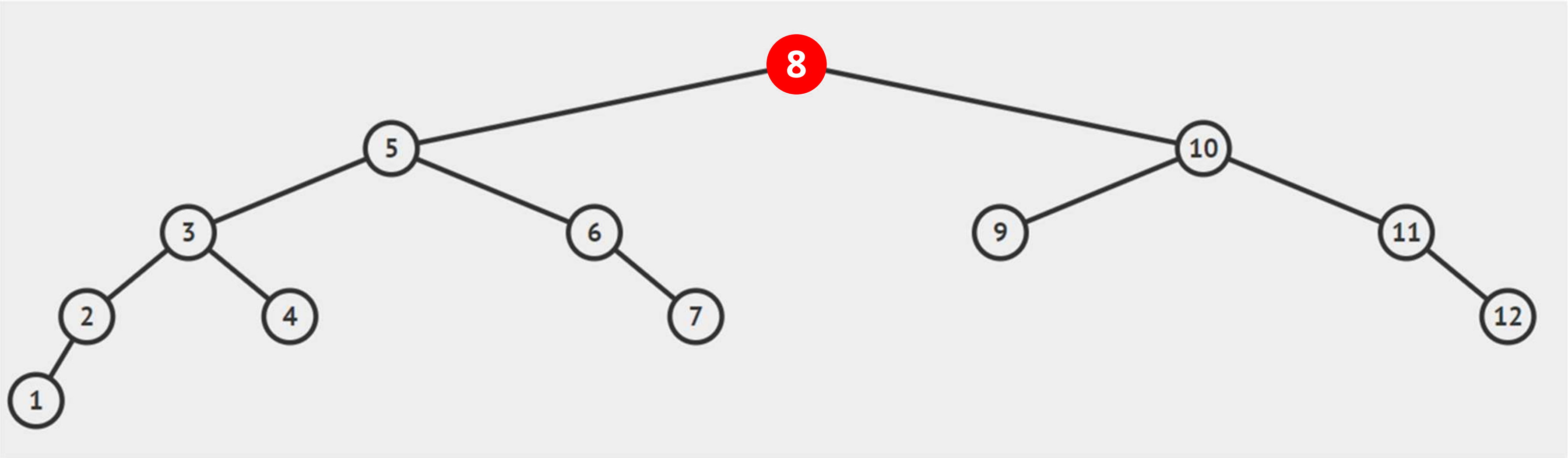
So here's a general strategy, given a subtree of this form:

To cause at least 2 rotations, we can simply find a vertex to remove in A such that A's height will be decreased by 1 after a rebalancing rotation within.



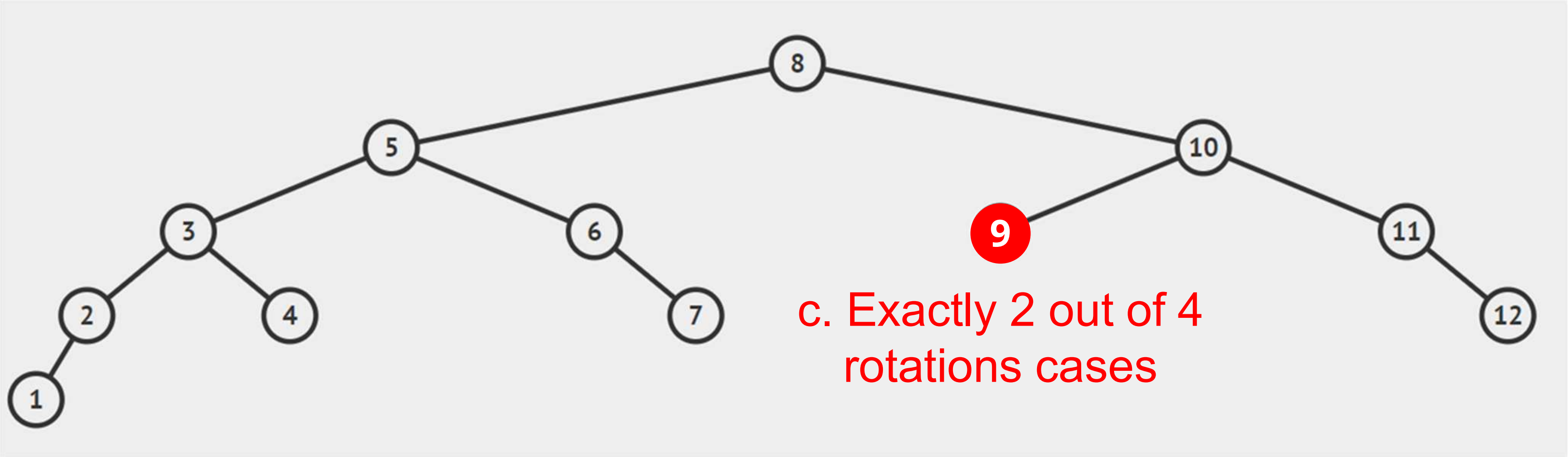
Question 2 (c)

c. Exactly 2 out of 4 rotations cases



https://visualgo.net/en/bst?create=8,5,10,3,6,9,11,2,4,7,12,1&mode=AVL

Question 2 (c)



https://visualgo.net/en/bst?create=8,5,10,3,6,9,11,2,4,7,12,1&mode=AVL

Question 3

Augmented BBST

Problem statement

There are two important BST operations: *Rank* and *Select* that are not included in VisuAlgo yet

(overview at https://visualgo.net/en/bst?slide=5-1) but can be quite useful for some order statistics problems. Please discuss on how to implement these two operations efficiently

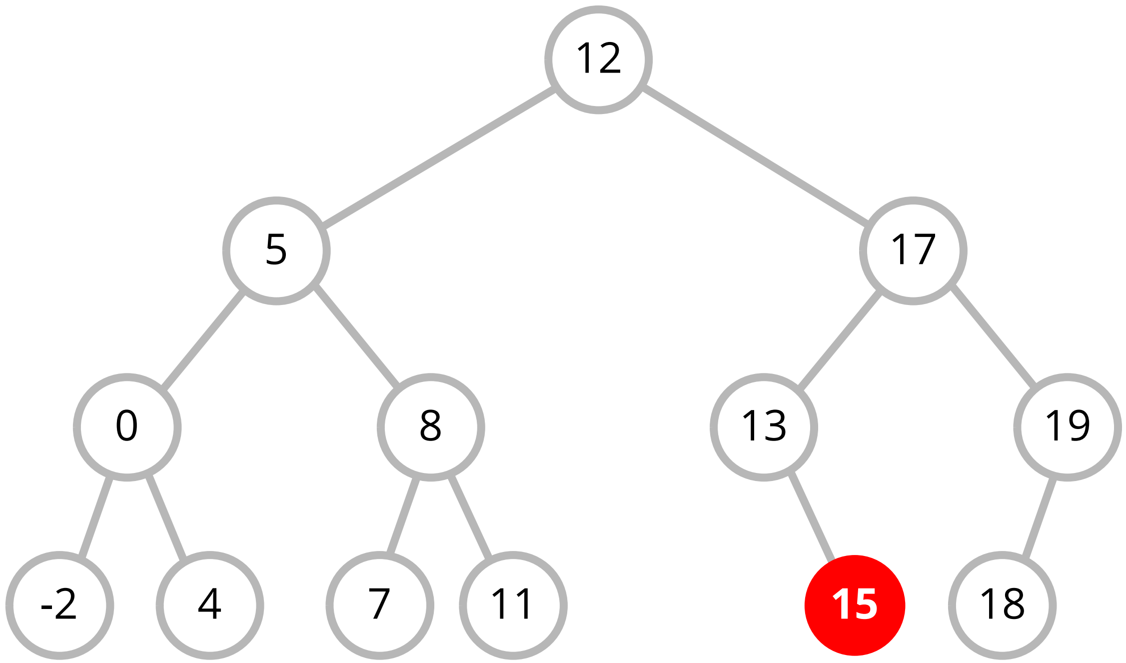
Question 3: Rank

`rank(key)` returns the 1-based index of `key` in the sorted ordering of all keys in the BST.

E.g. `rank(5)` on a BST containing `{2,1,7,5,0}` is `4` because it belongs to that 1-based index position in the sorted sequence `{0,1,2,5,7}`.

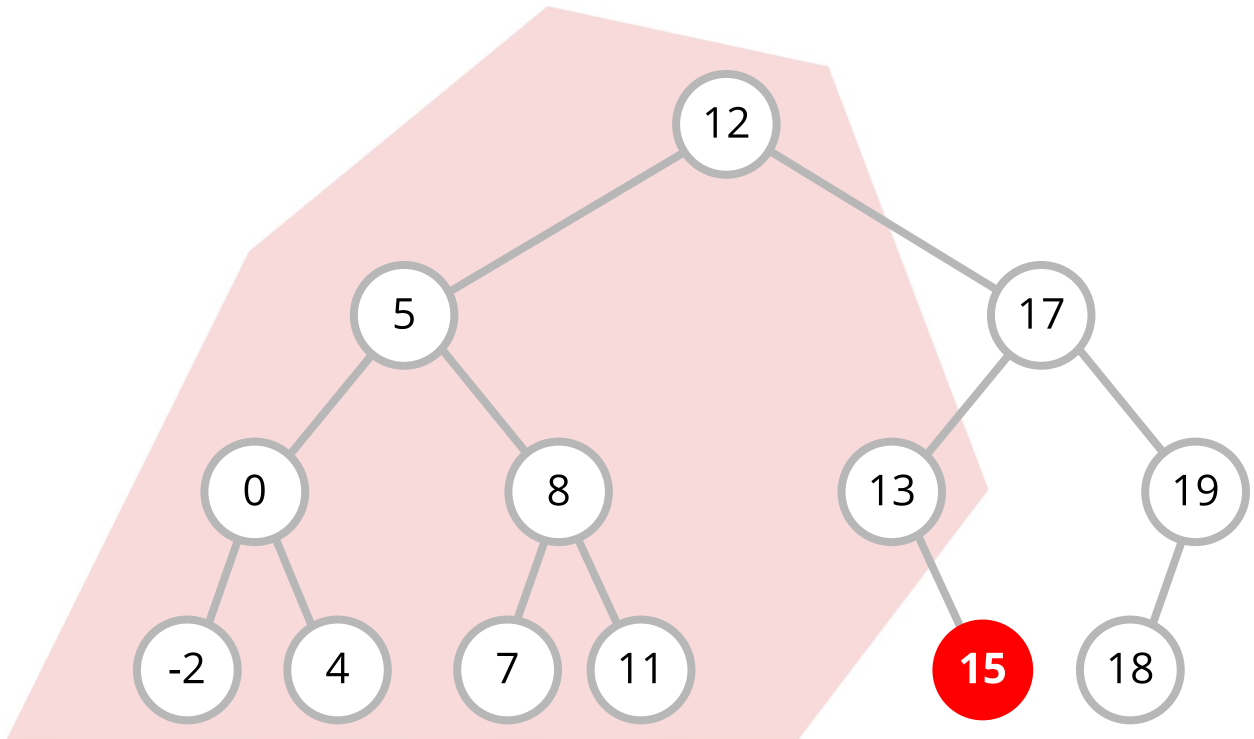
Realize, this is simply 1 + the number of keys in the BST that are *smaller than* `key`.

Question 3: Rank



Get rank of 15.

Question 3: Rank



Get rank of 15:
Rank 10.

Question 3: Rank

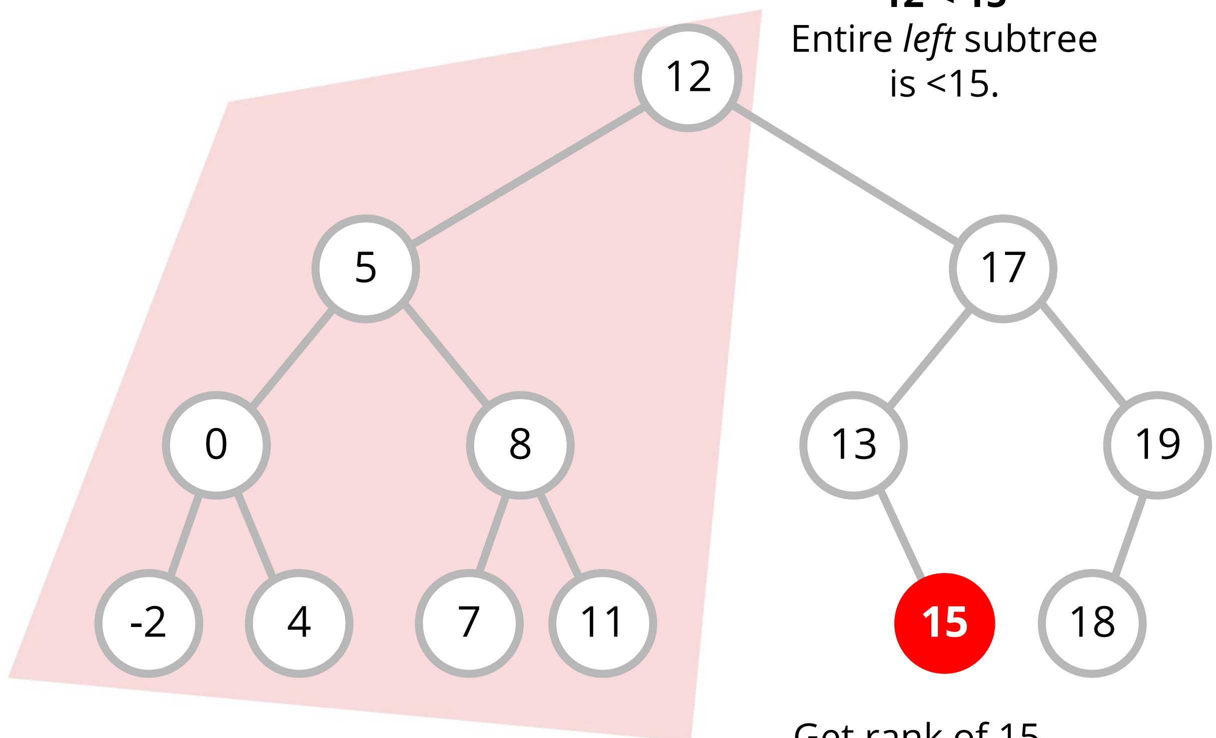
Storing Subtree Size

To implement this efficiently, at every vertex, we need to store number of vertices in the subtree (i.e size) rooted at that vertex.

Same idea as “caching” height of subtree in AVL Tree

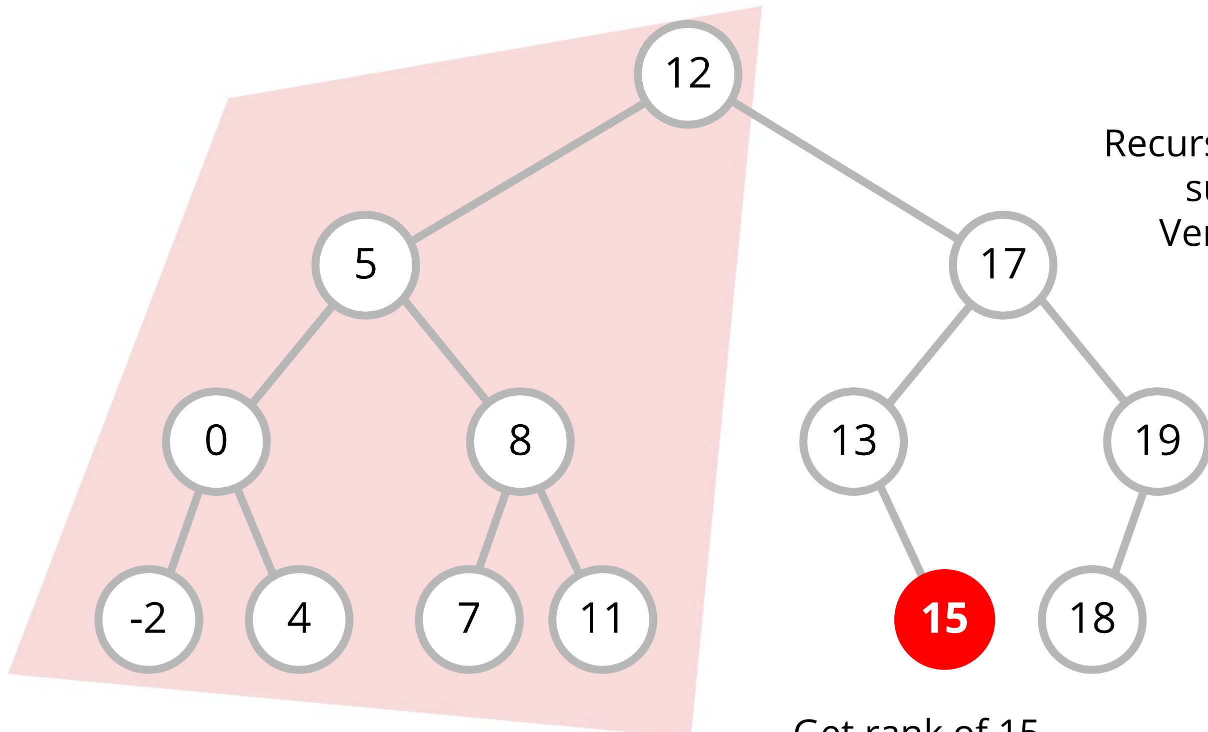
Question 3: Rank

12 < 15
Entire *left* subtree
is <15.



Get rank of 15.

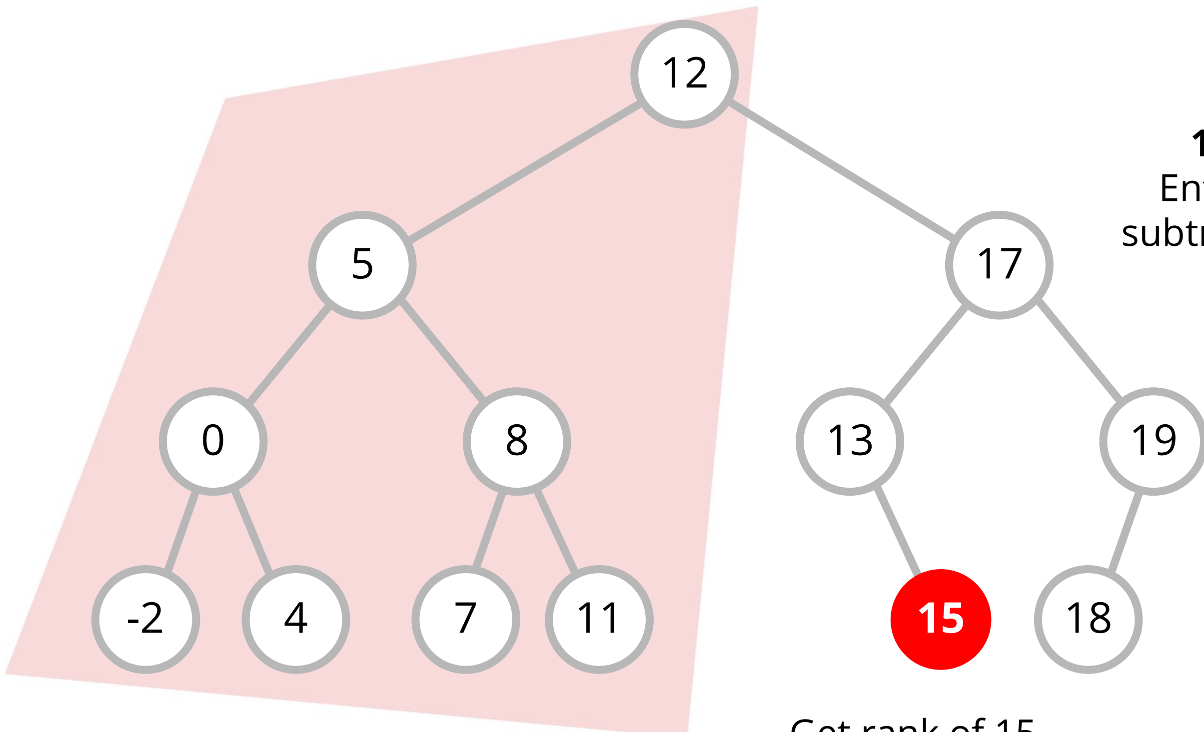
Question 3: Rank



Recurse into right subtree.
Vertex = 17

Get rank of 15.

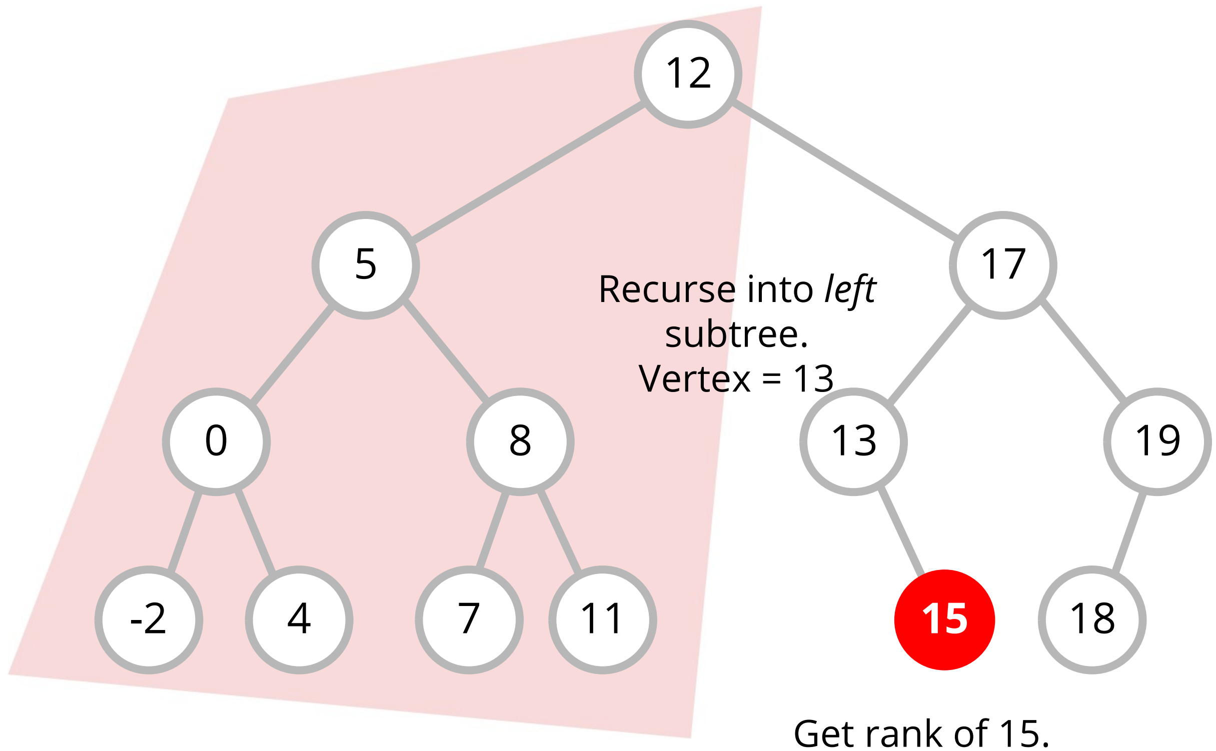
Question 3: Rank



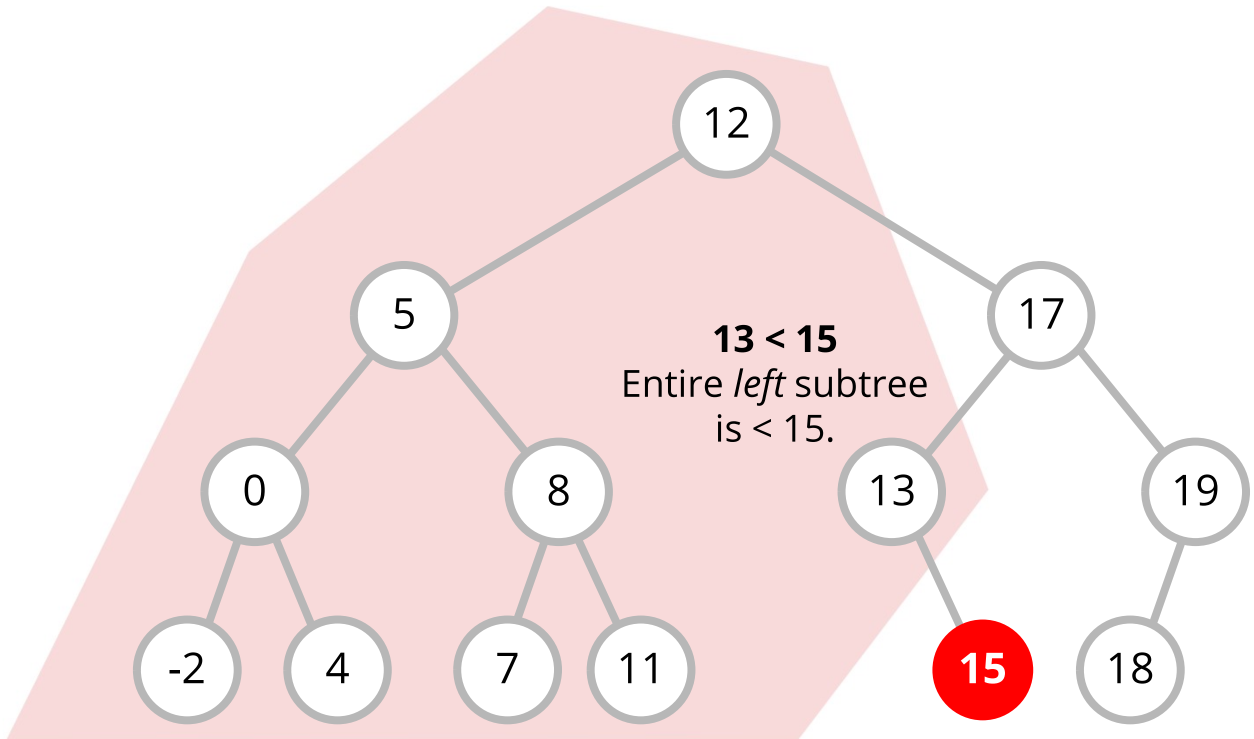
17 > 15
Entire *right* subtree is > 15.

Get rank of 15.

Question 3: Rank

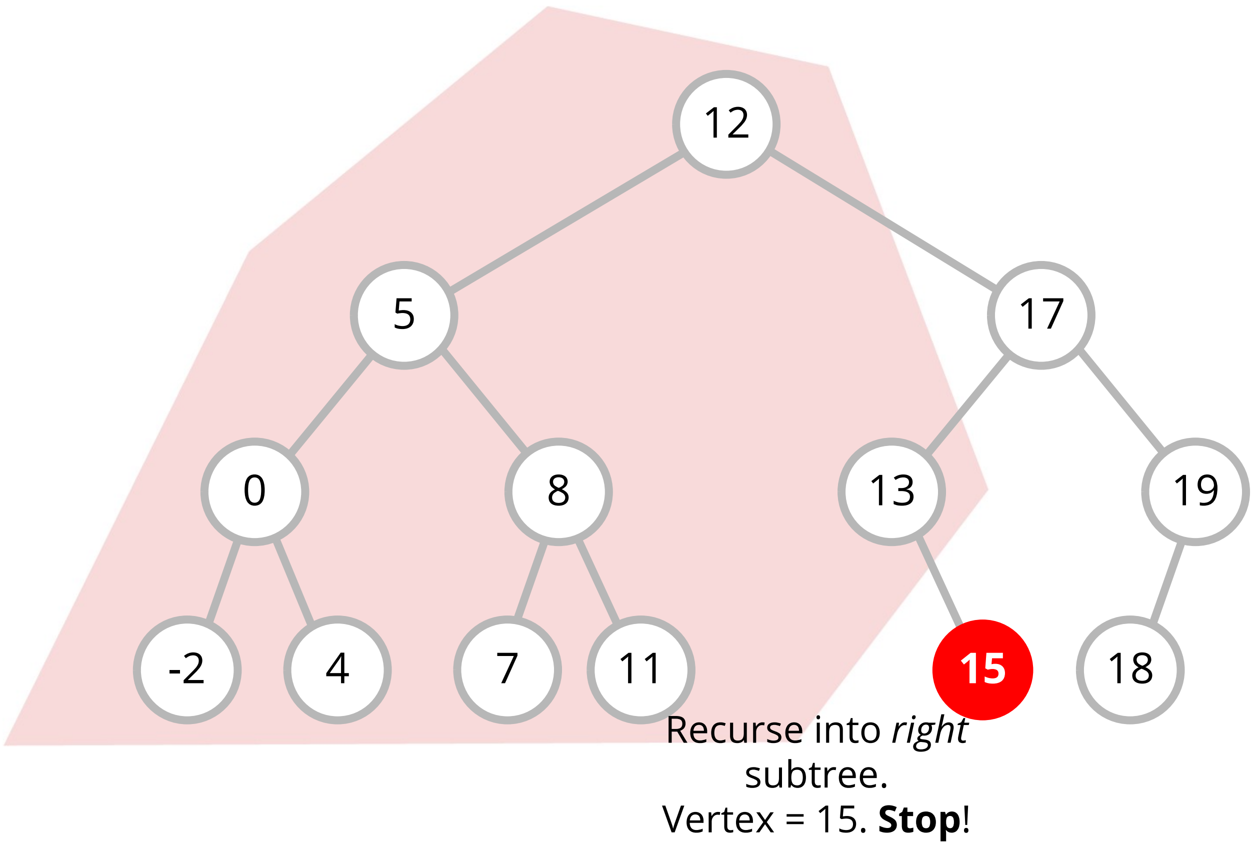


Question 3: Rank



Get rank of 15.

Question 3: Rank



Question 3: Rank

Solution 1: Tail-recursion

```
int rank(vertex v, int key, int accum) {
    if (v.key == key)       // Base case: at key
        return v.left.size + accum + 1;
    else if (v.key < key)   // Recurse right
        return rank(v.right, key, accum + v.left.size + 1);
    else                    // Recurse left
        return rank(v.left, key, accum);
}
```

Question 3: Rank

Solution 2

```
int rank(vertex v, int key) {
    if (v.key == key)          // Base case: at key
        return v.left.size + 1;
    else if (v.key < key)      // Recurse right
        return v.left.size + 1 + rank(v.right, key);
    else                       // Recurse left
        return rank(v.left, key);
}
```

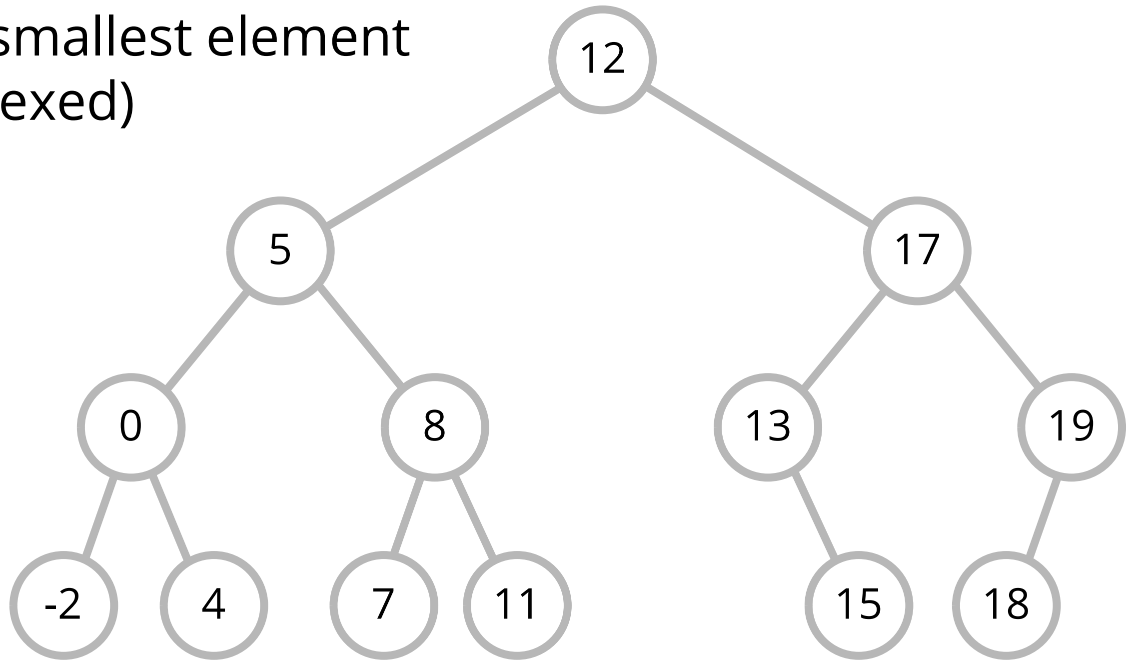
Question 3: Select

`select(k)` returns the **k^{th} smallest** key in the BST.

This is same as retrieving the key which is of **rank k**.

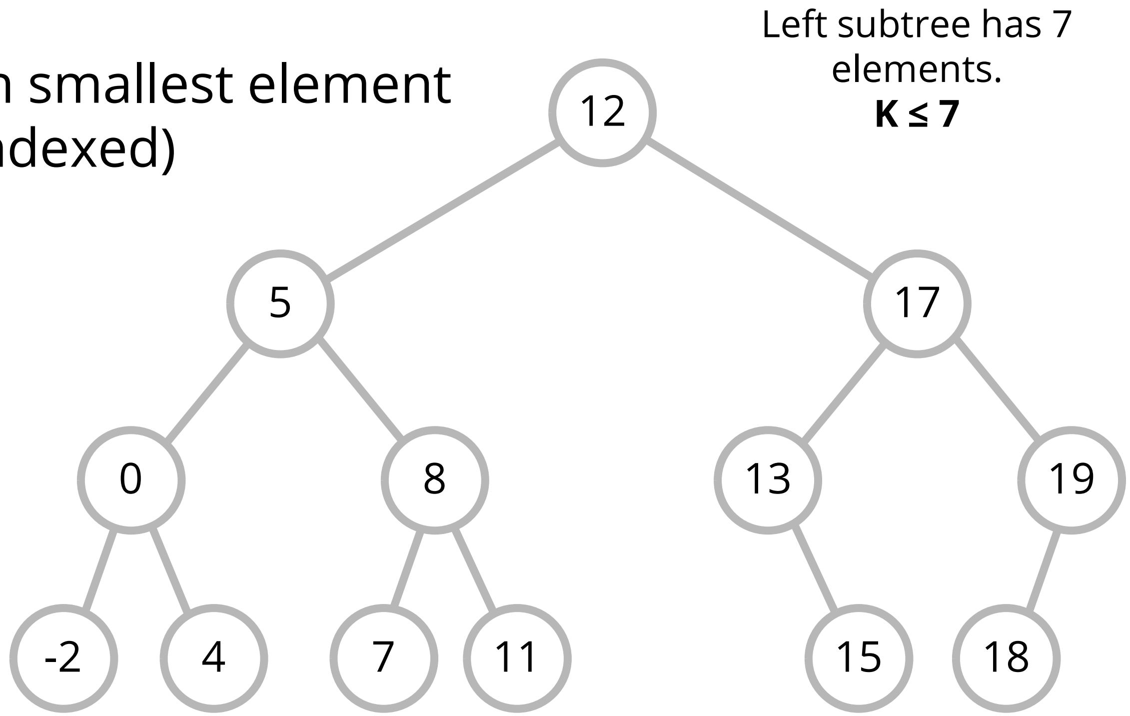
Question 3: Select

Select **K**-th smallest element
K = 5 (1 indexed)



Question 3: Select

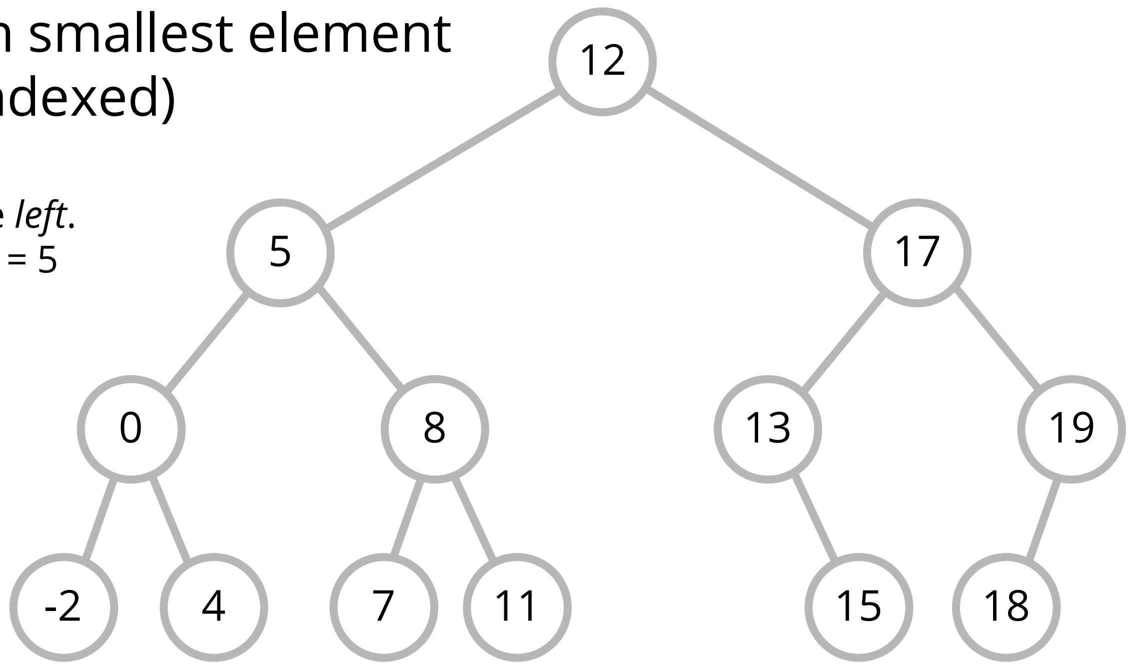
Select **K**-th smallest element
K = 5 (1 indexed)



Question 3: Select

Select **K**-th smallest element
K = 5 (1 indexed)

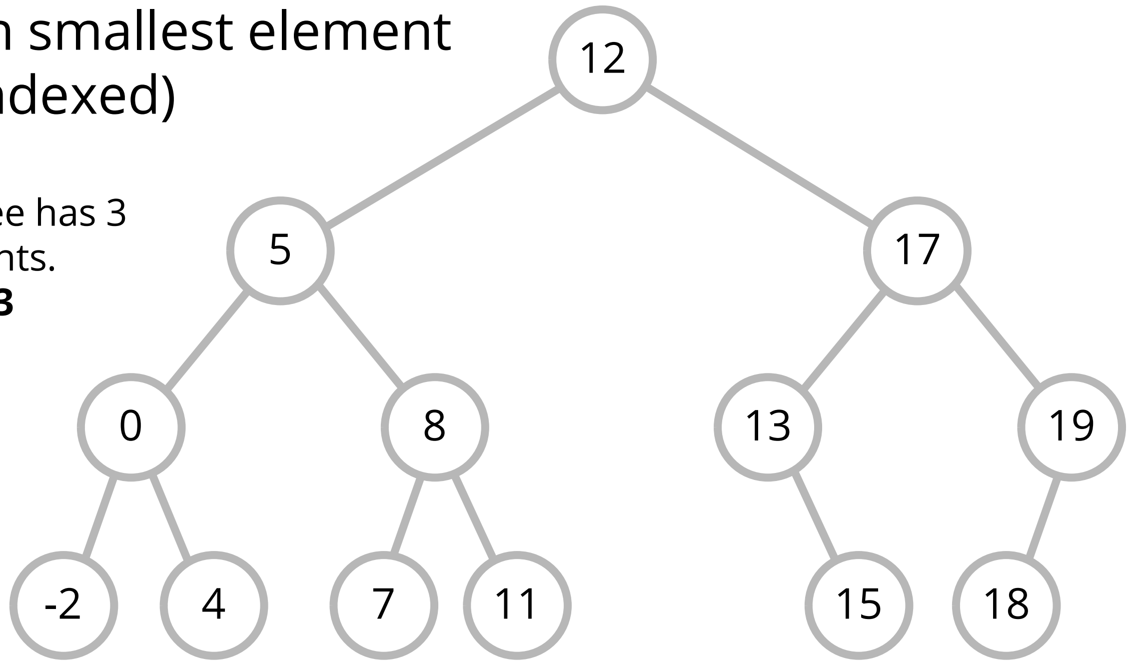
Recurse *left*.
Vertex = 5



Question 3: Select

Select **K**-th smallest element
K = 5 (1 indexed)

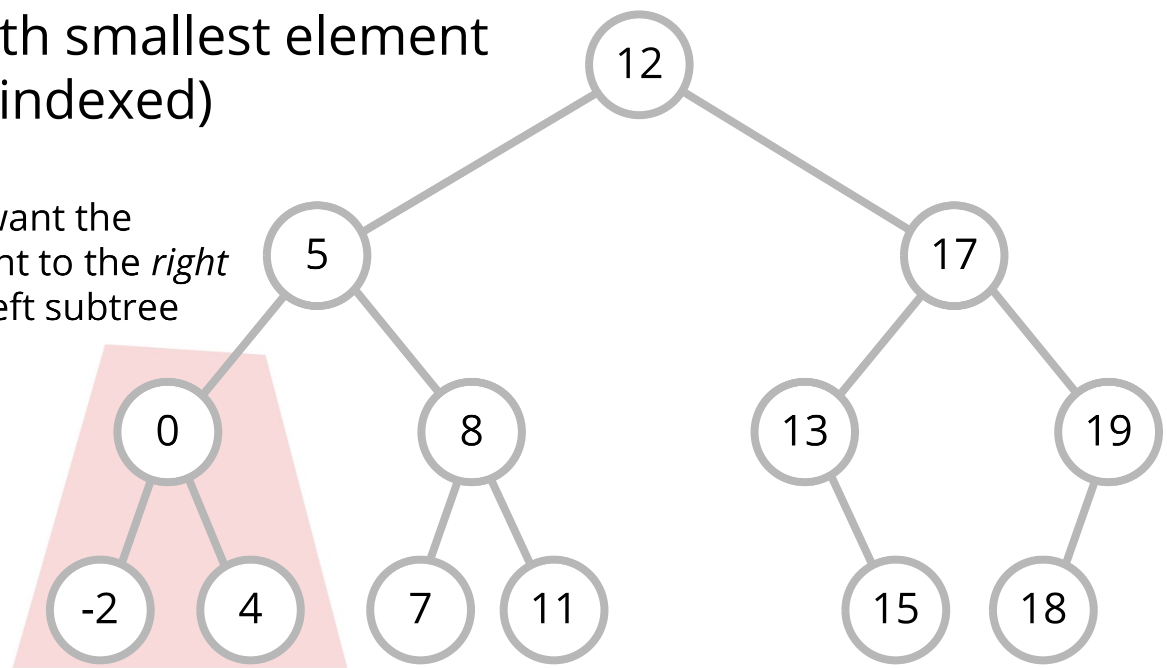
Left subtree has 3 elements.
K > 3



Question 3: Select

Select **K**-th smallest element
K = 5 (1 indexed)

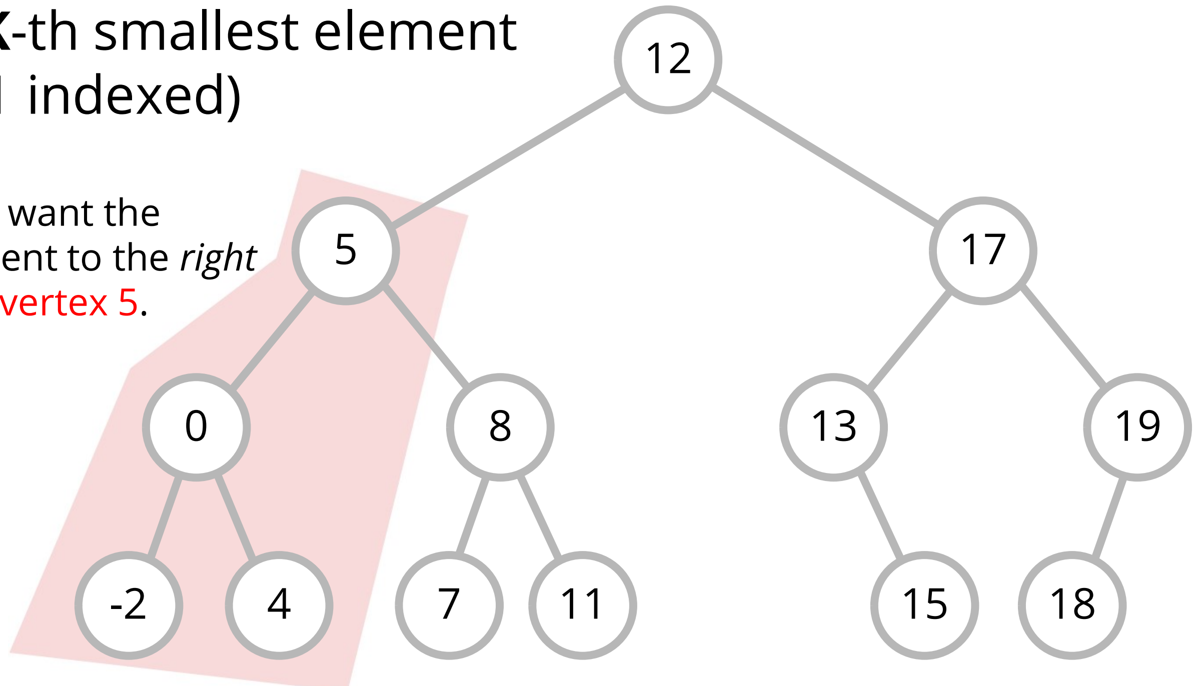
We want the
K-3 element to the *right*
of the left subtree



Question 3: Select

Select **K**-th smallest element
K = 5 (1 indexed)

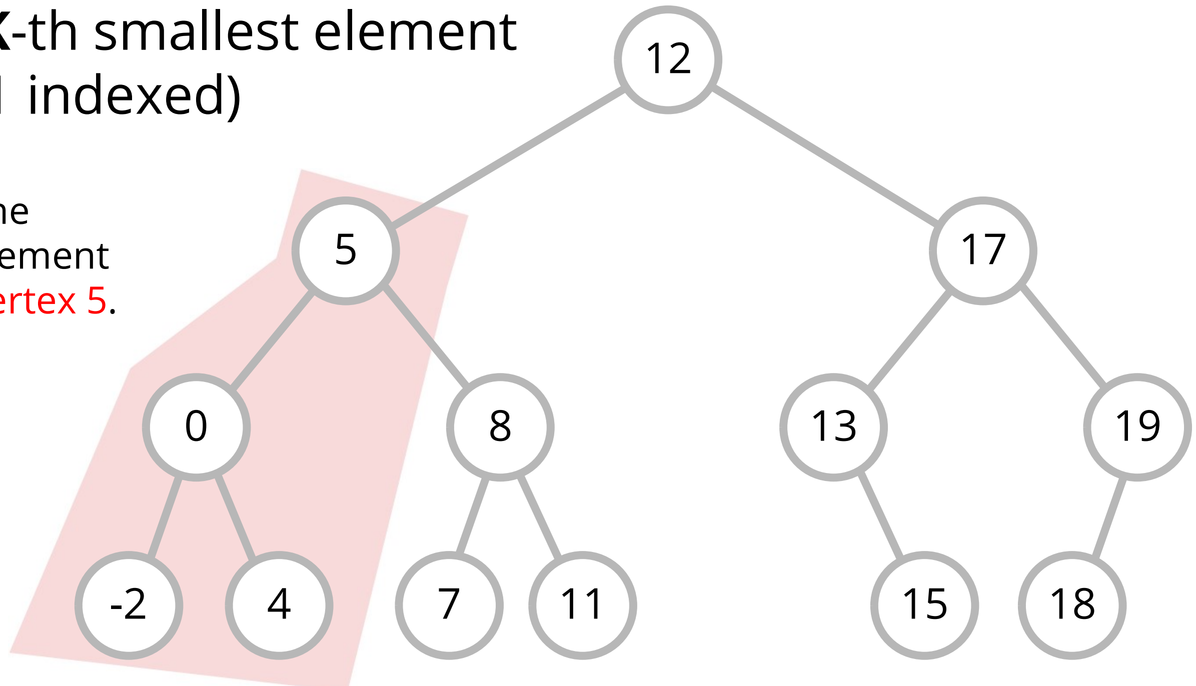
We want the
K-4 element to the *right*
of vertex 5.



Question 3: Select

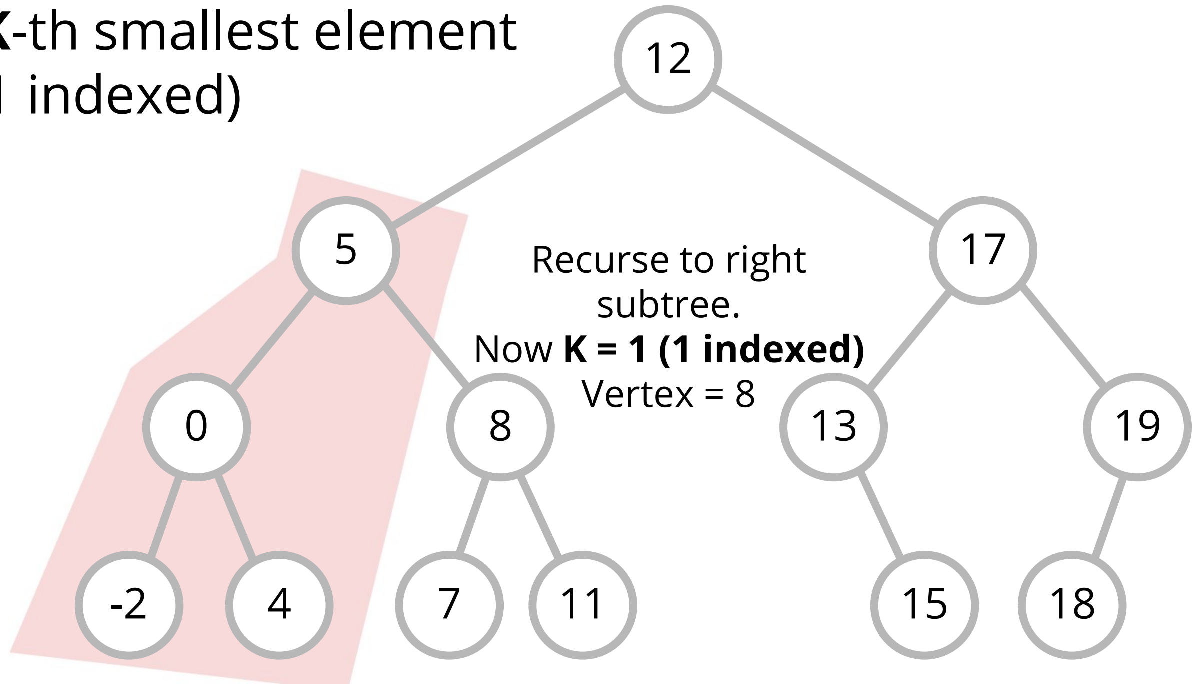
Select **K**-th smallest element
K = 5 (1 indexed)

We want the
1st smallest element
to the *right* of vertex 5.



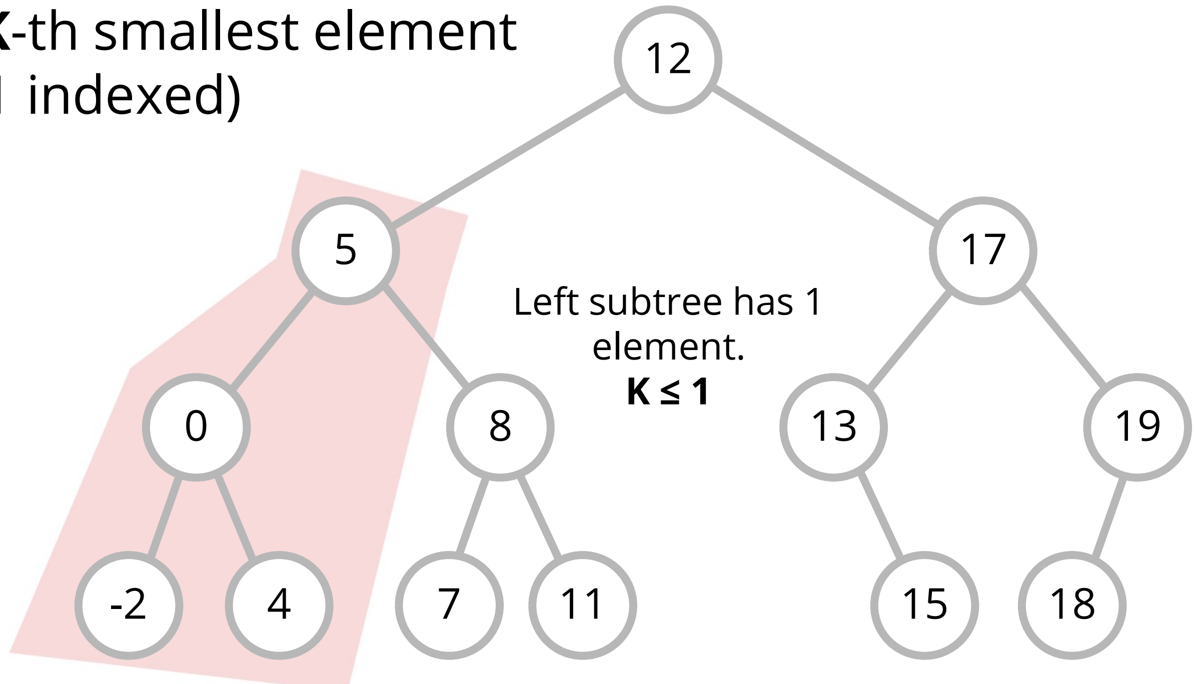
Question 3: Select

Select **K**-th smallest element
K = 5 (1 indexed)



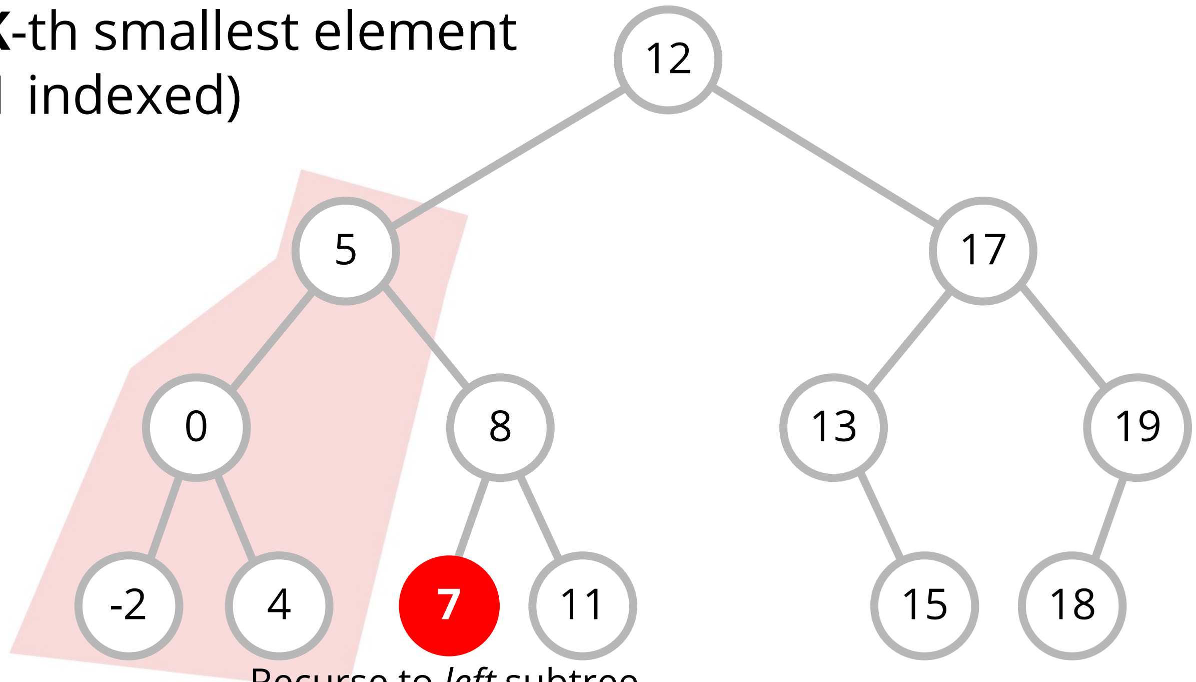
Question 3: Select

Select **K**-th smallest element
K = 5 (1 indexed)



Question 3: Select

Select **K**-th smallest element
K = 5 (1 indexed)



Recurse to *left* subtree.
K = 1
At leaf node. **Stop!**

Question 3: Select

Solution

```
int select(vertex v, int k) {
    // Base case: found it
    if (k == v.left.size + 1)
        return v.key;
    // If in left subtree: recurse left
    else if (k < v.left.size + 1)
        return select(v.lect, k);
    // If in right subtree: recurse right with updated k
    else
        return select(v.right, k - v.left - 1);
}
```

Self-read

Question 4

Priority Queue ADT

Table ADT

Self-read

Problem statement

We mentioned that Binary (Max) Heap can be used to implement Priority Queue ADT

How can we modify the implementation such that for *n* elements

- Both `ExtractMax()` and `ExtractMin()` done in *O(log n)* time
- Every other Priority Queue related-operations, especially `insert`/`enqueue` retains the same *O(log n)* time

Hint: What is the topic of this tutorial?

Self-read

ADT Recap

Analogy

Company is looking to hire a new employee.

Employee needs to be able to perform certain tasks:

- `enqueue()`
- `ExtractMax()`
- `top()`

Self-read

ADT Recap

Analogy

Many potential candidates applied for the job:

- Array
- Binary Heap
- Hash Table
- Binary Search Tree
- Balanced Binary Search Tree

Self-read

ADT Recap

Analogy

All of them *technically* can perform the task.

But some do it more *efficiently* than others.

Job Specifications $\leftrightarrow$ Abstract Data Type

Candidate $\leftrightarrow$ DS implementations

Self-read

Question 4: Modified Priority Queue ADT

Original Functions

- `enqueue()`
- `ExtractMax()`
- `top()`

New Functions

- `ExtractMin()`

Self-read

Question 4: Modified Priority Queue ADT

We can use a BBST to implement PQ ADT with all the desired functions aforementioned

Operations	BBST Implementation
`enqueue()`	Insert into BBST. *O(log n)*
`top()`	Find the maximum vertex. *O(log n)*
`ExtractMax()`	Find the maximum vertex and remove it. *O(log n)*
`ExtractMin()`	Find the minimum vertex and remove it. *O(log n)*

Self-read

Test yourself!

With a BBST implementation for Priority Queue ADT, how can you achieve `top()` in *O(1)* time? Can you employ the same strategy to peek at the minimum value in *O(1)* time as well?

Self-read

Question 5

Self-read

Problem statement

- Follow up from Question 4
- Let's revisit Question 3 of Tutorial 05 (on right)
- Would you answer that question differently?

There are two interesting features of Binary Heap data structure that are not available in C++ STL priority queue and Java PriorityQueue yet: `Increase`/`Decrease`/`UpdateKey(old k, new k)` and `DeleteKey(k)` where `v` is not necessarily the max element. These two operations are not yet included in VisuAlgo .

Self-read

Question 5: `DeleteKey(k)`

- Lazy deletion
  - Spoiler given 2 tutorials ago
  - Can keep track of deleted elements using another priority queue
- BBST
  - Delete any element in $O(\log N)$

Self-read

Question 5: `UpdateKey(old k, new k)`

- Lazy update
  - Spoiler given 2 tutorials ago
  - Use lazy deletion approach to find and mark `old k` as invalid, then enqueue `new k` as new element
- BBST
  - Find key in *O(log N)*, delete key in *O(log N)*, finally insert `new k` as new element in *O(log N)*

Self-read

Relevant discussion: BBST for Table ADT

- Up till now, we have only seen examples of BBST storing simple values
- Realize that BBST vertices can also store (Key, Value) pairs (i.e. Table ADT's satellite data)!
- This allows us to effectively implement Table ADT using BBST!

Self-read

Relevant discussion: BBST for Table ADT

- With a BBST implementation for Table ADT, the comparison between vertices is based on **Key**. i.e. Successor to a vertex represents the next largest key
- We can thus conduct *binary search* for keys!
- Therefore no hash function is necessary, and consequently collision resolution is also not required
- How might we handle (Key, Value) pairs sharing the same key? We can use employ separate chaining idea where each vertex is a now a bucket!

Some remarks

- Priority Queue ADT
  - Doesn't *necessarily* have to be a Binary Heap!
  - For exposure: There are many other heaps as well
- Table ADT
  - Doesn't *necessarily* have to be a Hash Table!
- We have seen how BBST can serve as a reasonable implementation for both ADTs

Question 6

Problem statement

As of now, you have been exposed with both possible implementations of Table ADT:

1. Hash Table (and its variations)
2. BST (including Balanced BST like AVL Tree)

Now write down **4 potential usage scenarios** of Table ADT

- Two scenarios should favor the usage of Hash Table
- The other two scenarios should favor using Balanced BST

Question 6: When to use Hash Table

1. Pure key-to-value mapping without ordering of keys
2. No order statistic queries needed
3. Time limit is tight: Need to choose $O(1)$ over $O(\log N)$
4. Hashing of keys is easier/well-known
5. Occasional re-hashing is tolerated

Question 6: When to use BBST

1. Key-to-value mapping with **keys in sorted order**
2. Need to support order statistic queries: `min`/`max`/`lower_bound`/`upper_bound`/`select`/`rank` etc.
3. Keys are harder to be hashed (but easier to compare, like a tuple)
4. Memory constraints imposed where a Hash Table would be considered wasteful in comparison
5. Consistent and deterministic time complexity requirements where re-hashing in Hash Table cannot be tolerated

C++ library containers

Container	Satellite data	Allow duplicates?	DS / ADT
`set`	Key	No	BBST
`multiset`	Key	Yes	BBST
`map`	(Key, Value)	No	BBST
`multimap`	(Key, Value)	Yes	BBST
`unordered_set`	Key	No	Hash Table
`unordered_multiset`	Key	Yes	Hash Table
`unordered_map`	(Key, Value)	No	Hash Table
`unordered_multimap`	(Key, Value)	Yes	Hash Table

Questions?

PS3

- Can use std::priority_queue (binary heap) with std::unordered_map (hash table)
    - Lazy deletion – hash table to keep track of non-deleted keys, and mapping strings
- Can also use std::set (BBST) with std::unordered_map
    - Hash table to map strings

/fantasydraft

PS4

- A: seems … could a naïve solution work?
- B: unweighted SSSP (Lecture 10b), can self learn first