CS2040C Tut 8

Binary Search Tree

AVL Tree

Binary Search Trees

Non-Balanced

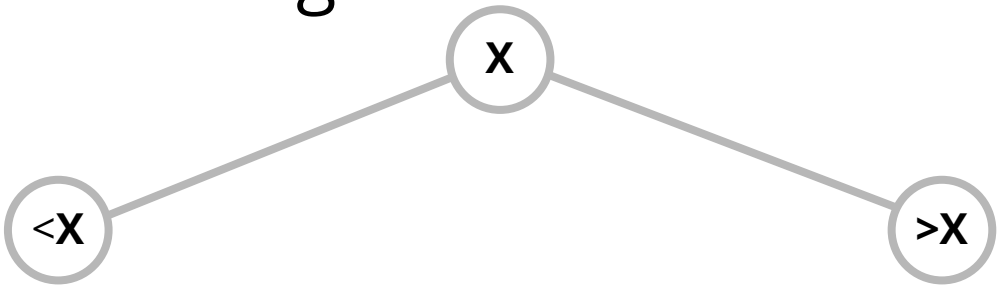
General Properties of BST

Main Idea

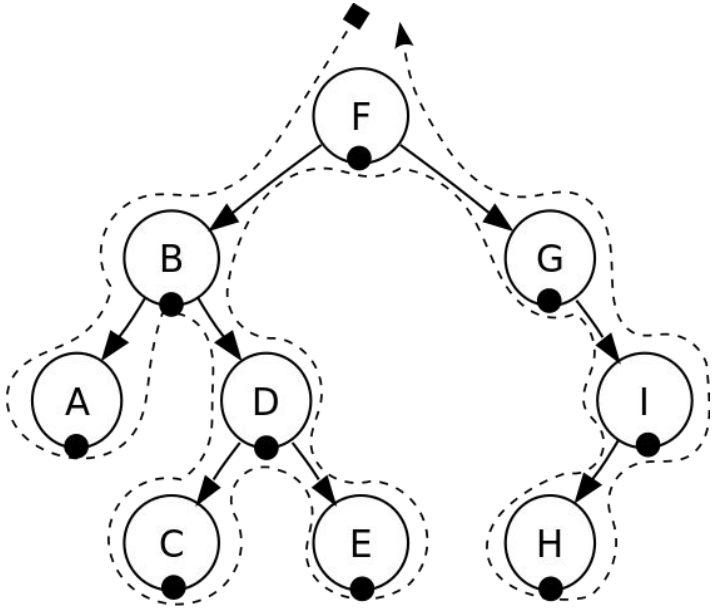
For every vertex, **X**:

Left child is smaller than **X**.

Right child is larger than **X**.

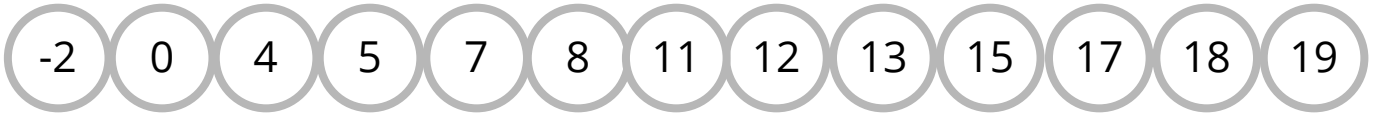


In-order Traversal [Credits: SG IOI Training 2017, Wikipedia]



A B C D E F G H I

BST



In-order traversal??

```
set<int> s;
for (int i = 0; i < N; i++) s.insert(i);
```

What does the code *below* do? (other than printing all the numbers)

What is the *time complexity*?

```
for (set<int>::iterator it = s.begin();
        it != s.end(); it++) {          //O(N) or O(N log N)?
    cout << *it << endl;
}
```

In-order traversal??

What is the time complexity of this code?

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

Range based for-loop

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

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

Balanced BST

AVL Tree

AVL Tree: Balanced

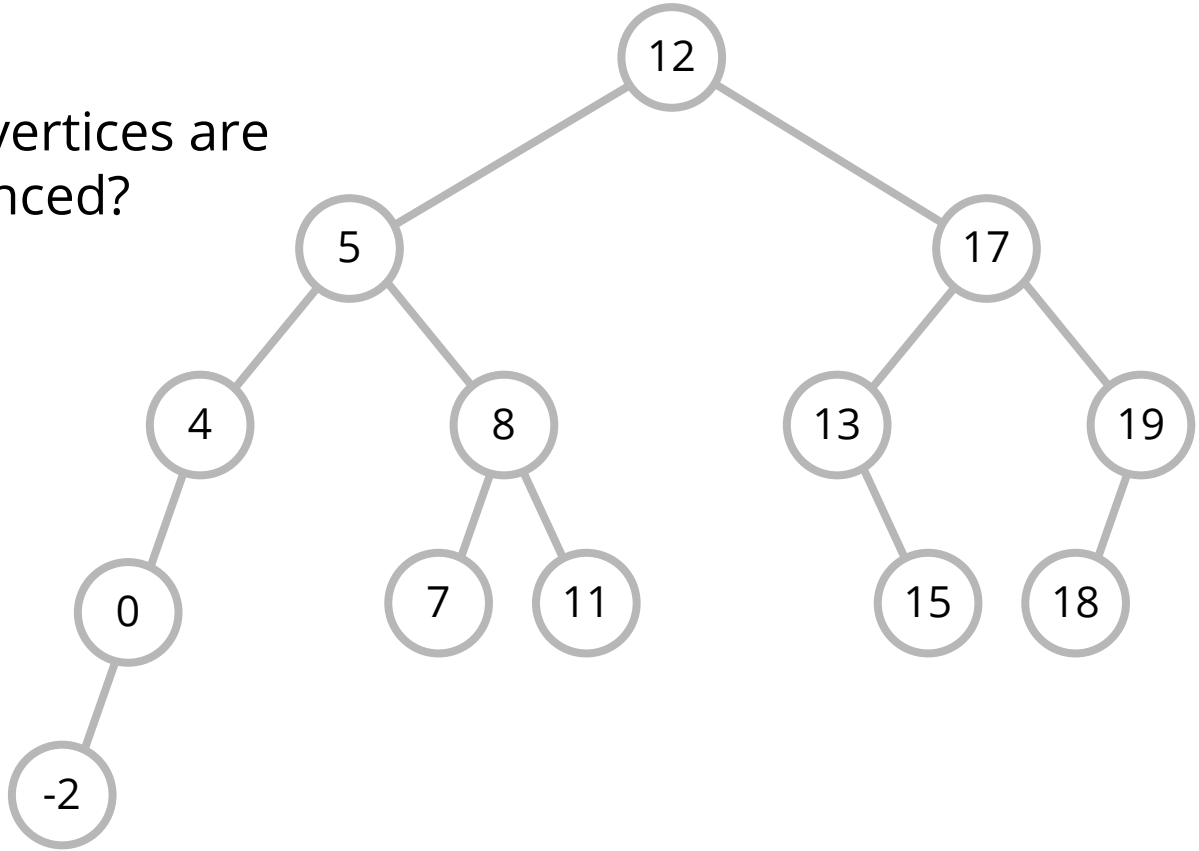
Height-balanced

A vertex is height-balanced if:

> *Difference* in height of left and right child is **not more than 1**.

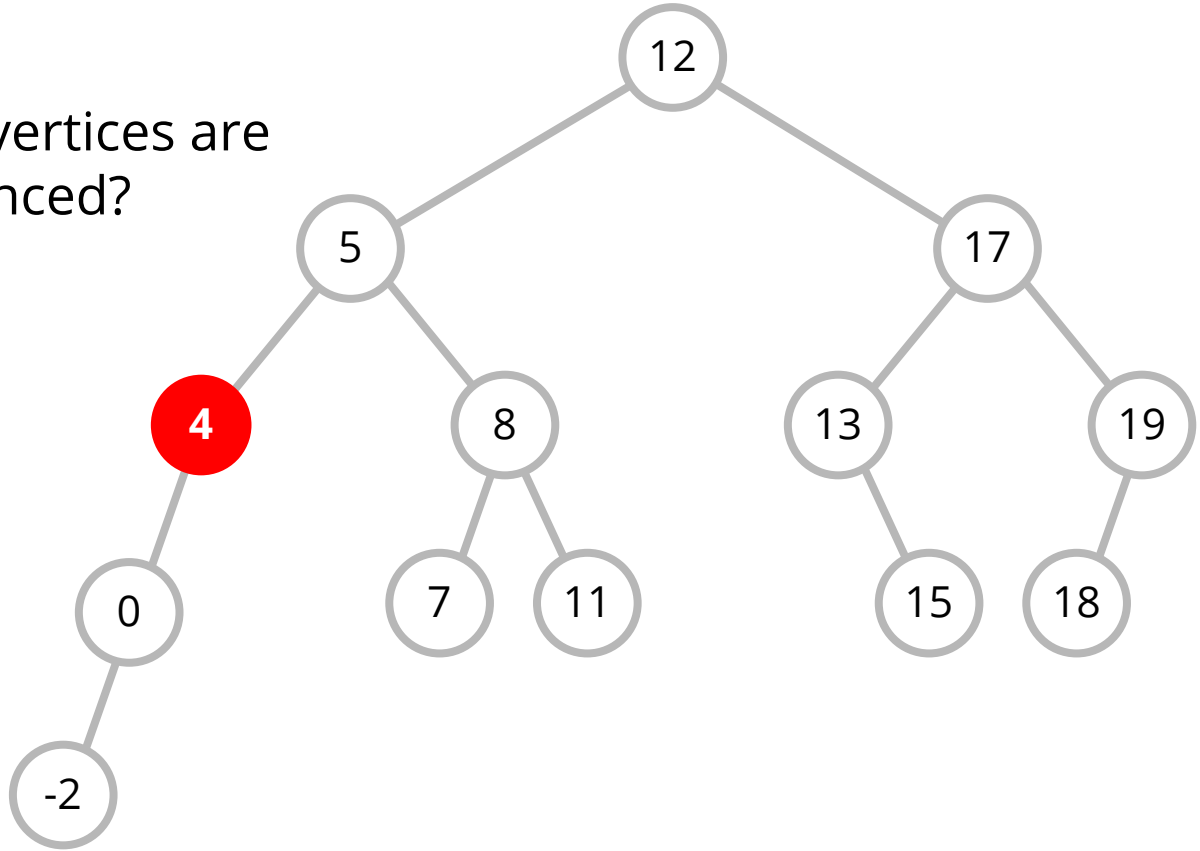
BST

Which vertices are unbalanced?



BST

Which vertices are unbalanced?



AVL Tree: Rotations

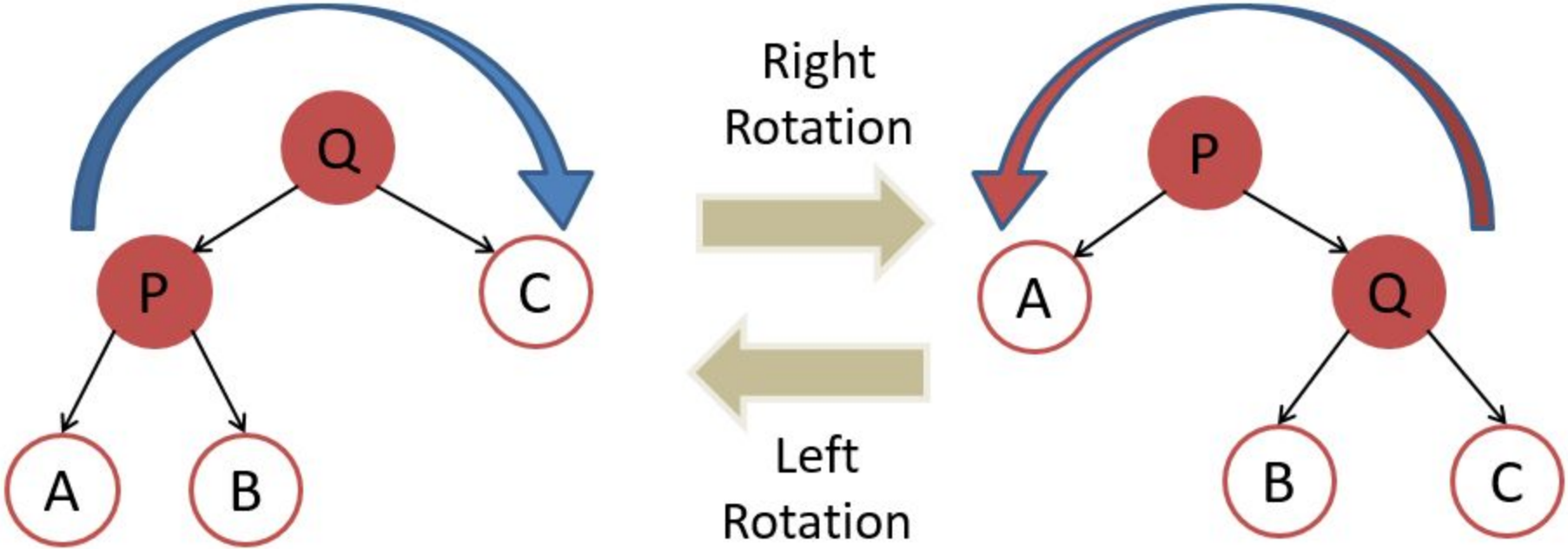
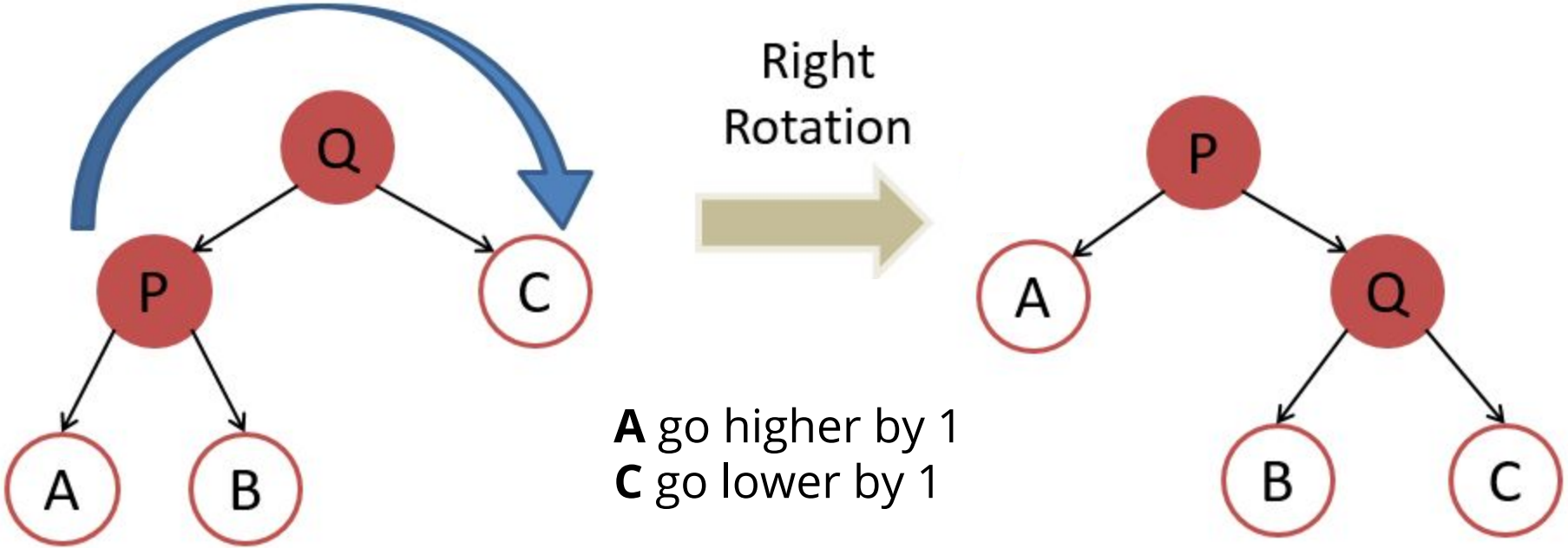


Image from VisuAlgo (and from previous CS2010/CS1102 lecture notes)

AVL Tree: Right Rotate



A go higher by 1
C go lower by 1

Image from VisuAlgo

AVL Tree: Left Rotate

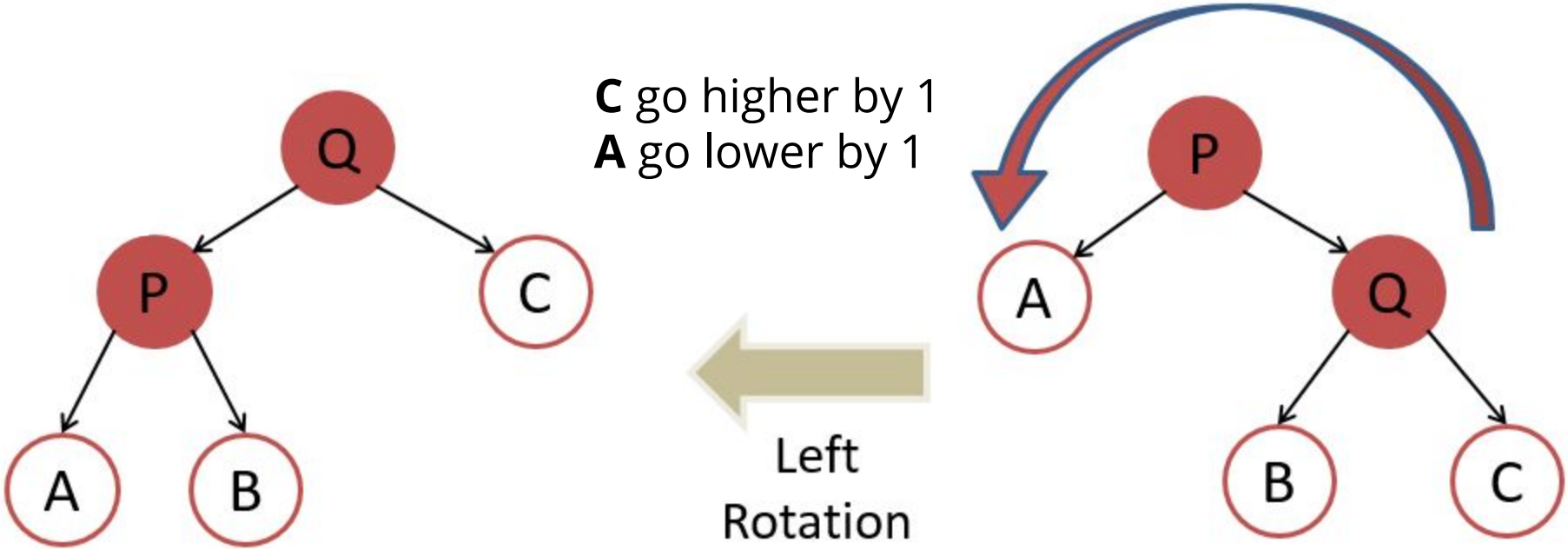


Image from VisuAlgo

AVL Tree: Balancing

Balance Factor

We let bf(**v**) be h(*v.left*) - h(*v.right*).

If left subtree is *higher* than right subtree,

bf(**v**) will be *positive*.

If left subtree is *lower* than right subtree,

bf(**v**) will be *negative*.

AVL Tree: Balancing

2 cases *first*:

A vertex **v** is unbalanced.

1. bf(**v**) > 1 [Left is higher]
   a. *Intuitively*, we need to rotate right
2. bf(**v**) < -1 [Right is higher]
   a. *Intuitively*, we need to rotate left

AVL Tree: Balancing

Observation

A right rotate will *balance* a subtree if:

- The *left* child not skewed to the right
- (i.e. **bf(v.left) ≥ 0**)

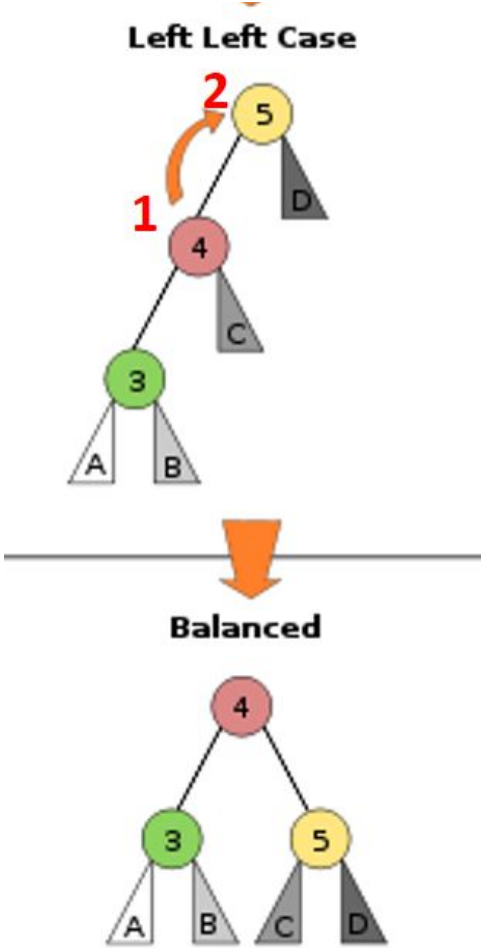


Image from VisuAlgo

AVL Tree: Balancing

Observation

A left rotate will *balance* a subtree if:

- The *right* child not skewed to the left
- (i.e. **bf(v.right) ≤ 0**)

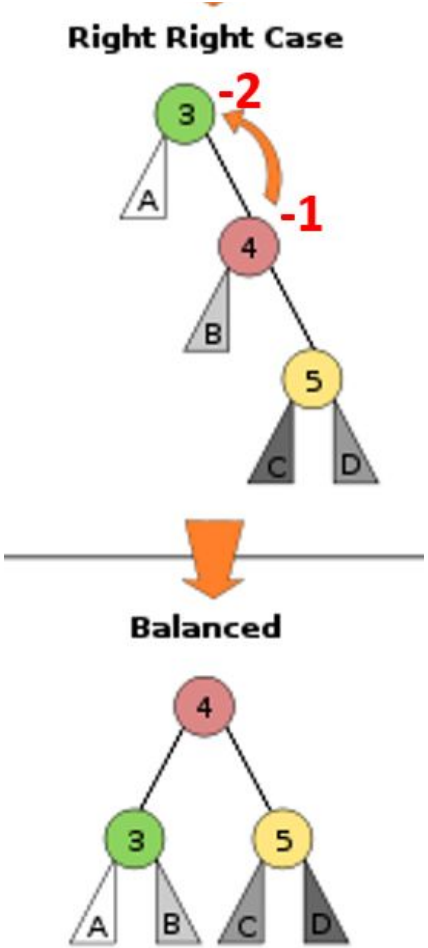


Image from VisuAlgo

AVL Tree: Balancing

Observation

In essence, they *cannot* have opposing signs.

0 is <u>neither</u> positive or negative.

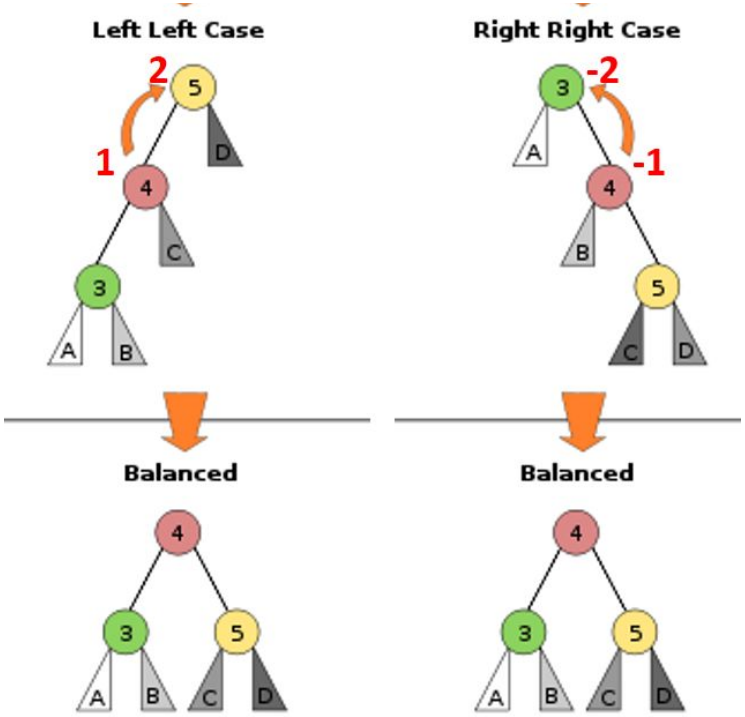


Image from VisuAlgo

AVL Tree: Balancing

Observation

If it is not the correct sign,

we can rotate the child to let it have the correct sign.

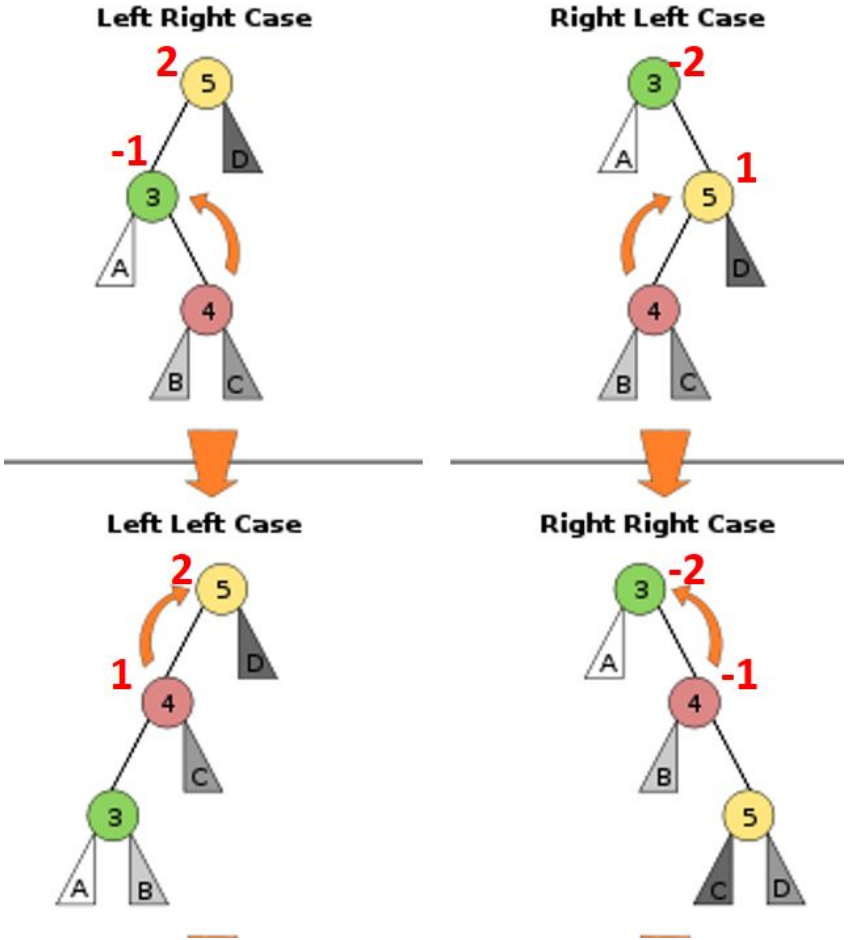


Image from VisuAlgo

Summary

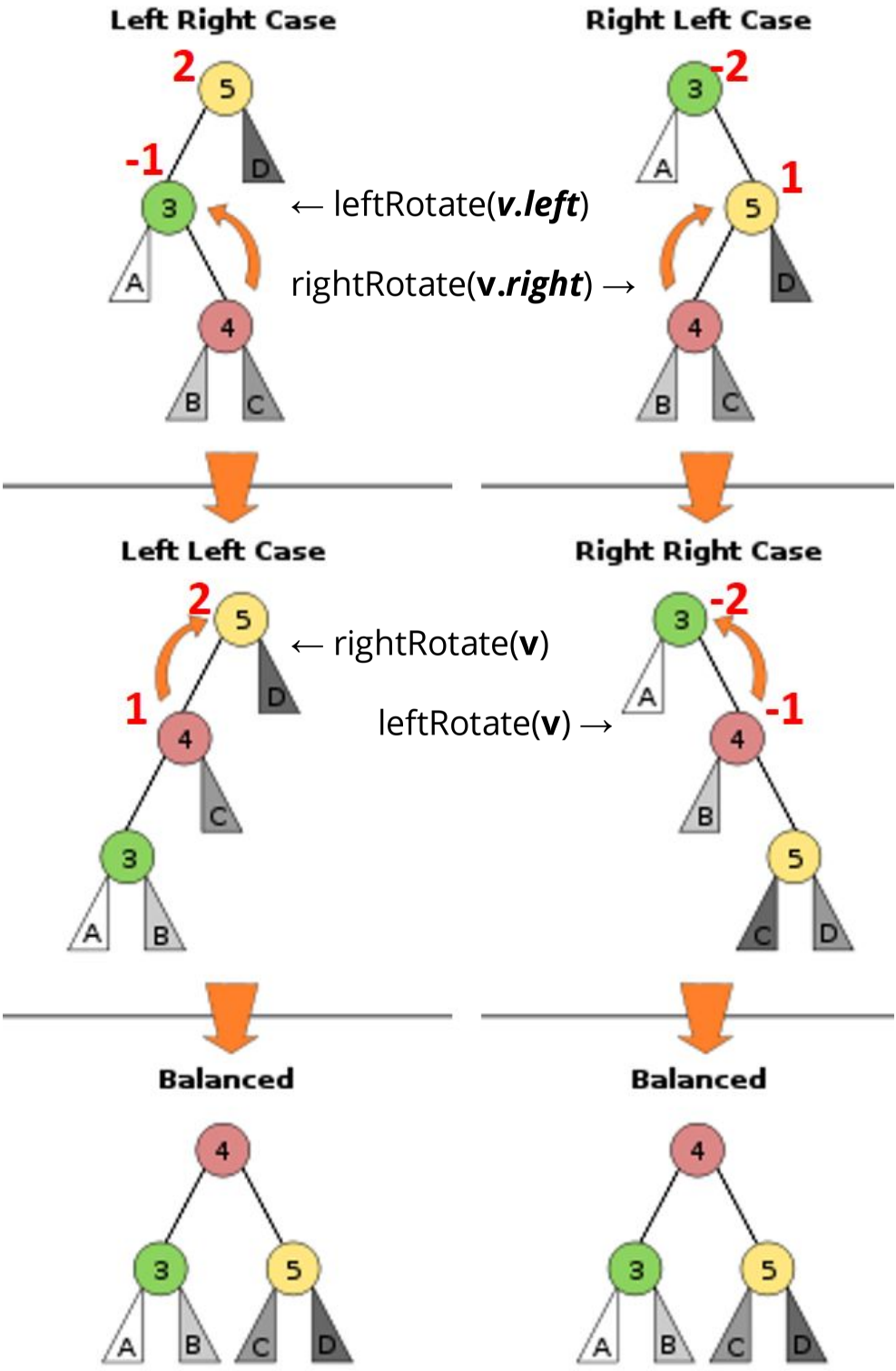


Image from VisuAlgo

Q1

VisuAlgo questions

Q1.1 & Q1.2

Prev PgUp

10-4. Remove(v) - Case 3 Discussion

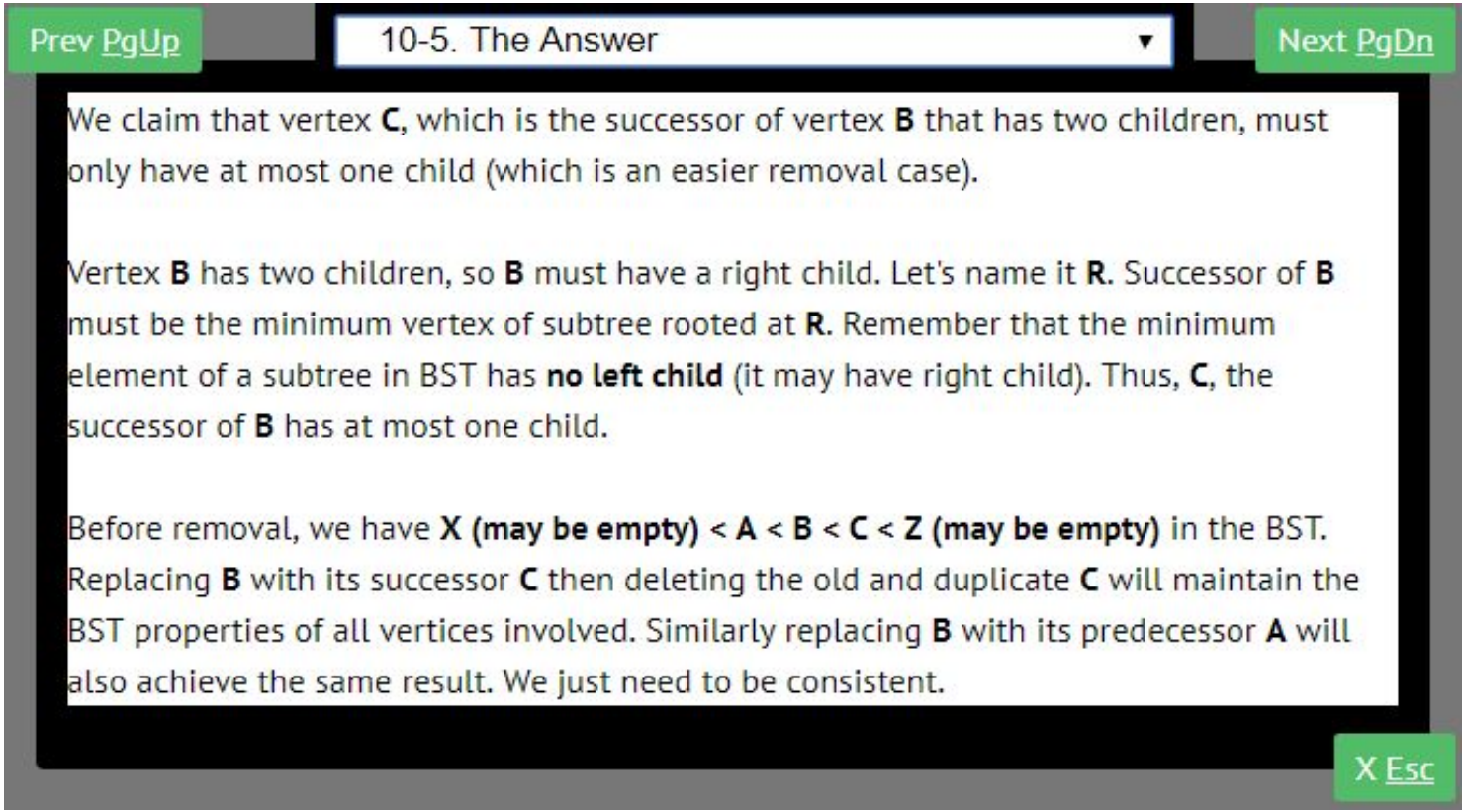
Next PgDn

This case 3 warrants further discussions:

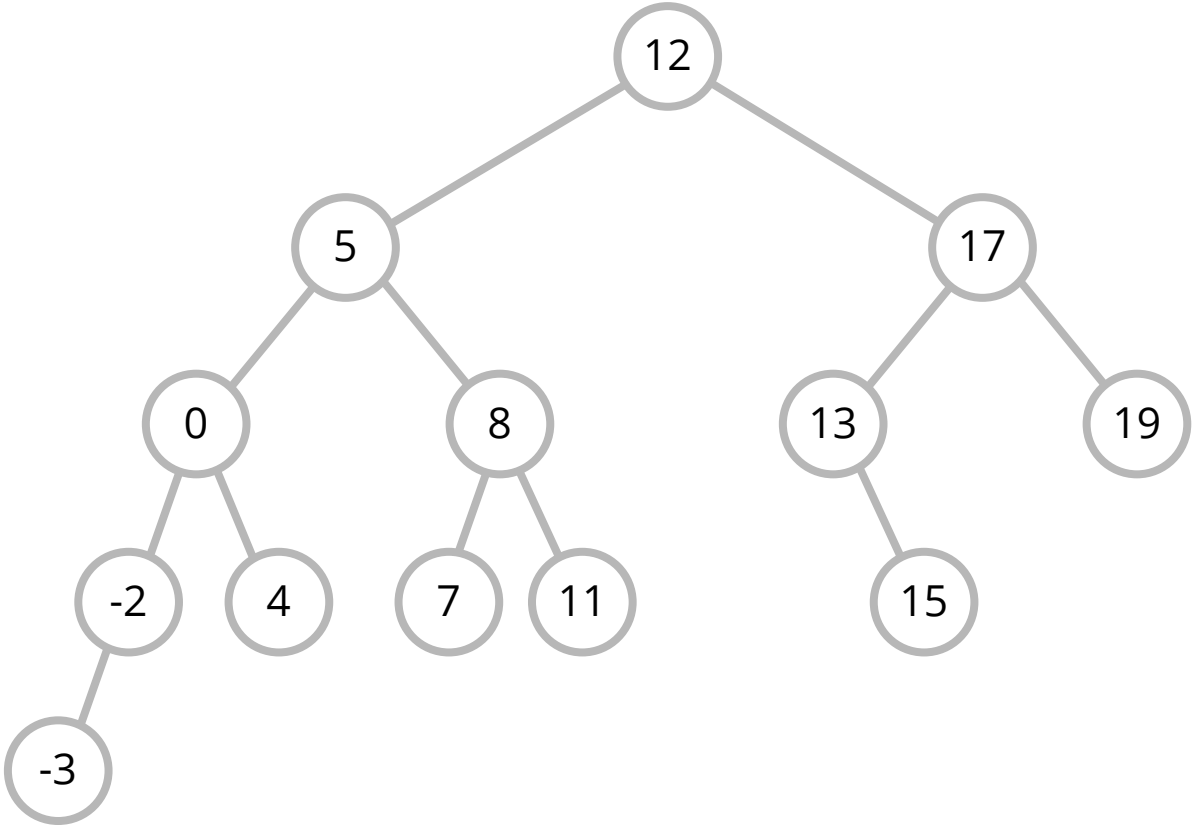
1. Why replacing a vertex **B** that has two children with its successor **C** is always a valid strategy?
2. Can we replace vertex **B** that has two children with its predecessor **A** instead? Why or why not?

X Esc

Q1.1 & Q1.2


Prev PgUp
10-5. The Answer
Next PgDn
We claim that vertex **C**, which is the successor of vertex **B** that has two children, must only have at most one child (which is an easier removal case).
Vertex **B** has two children, so **B** must have a right child. Let's name it **R**. Successor of **B** must be the minimum vertex of subtree rooted at **R**. Remember that the minimum element of a subtree in BST has **no left child** (it may have right child). Thus, **C**, the successor of **B** has at most one child.
Before removal, we have **X (may be empty) < A < B < C < Z (may be empty)** in the BST. Replacing **B** with its successor **C** then deleting the old and duplicate **C** will maintain the BST properties of all vertices involved. Similarly replacing **B** with its predecessor **A** will also achieve the same result. We just need to be consistent.
X Esc

BBST



Q1.3

Prev PgUp | 14-11. Insert(v) in AVL Tree ▾ | Next PgDn

1. Just insert **v** as in normal BST,
2. Walk up the AVL Tree from the insertion point back to the root and at every step, we update the height and balance factor of the affected vertices:
    a. Stop at the **first** vertex that is out-of-balance (+2 or -2), if any,
    b. Use **one** of the four tree rotation cases to rebalance it again, e.g. try Insert(37) on the example above and notice by calling **rotateLeft(29)** once, we fix the imbalance issue.

Discussion: Is there other tree rotation cases for Insert(v) operation of AVL Tree?

X Esc

Q1.3

Prev PgUp | 14-12. The Answer ▾ | Next PgDn

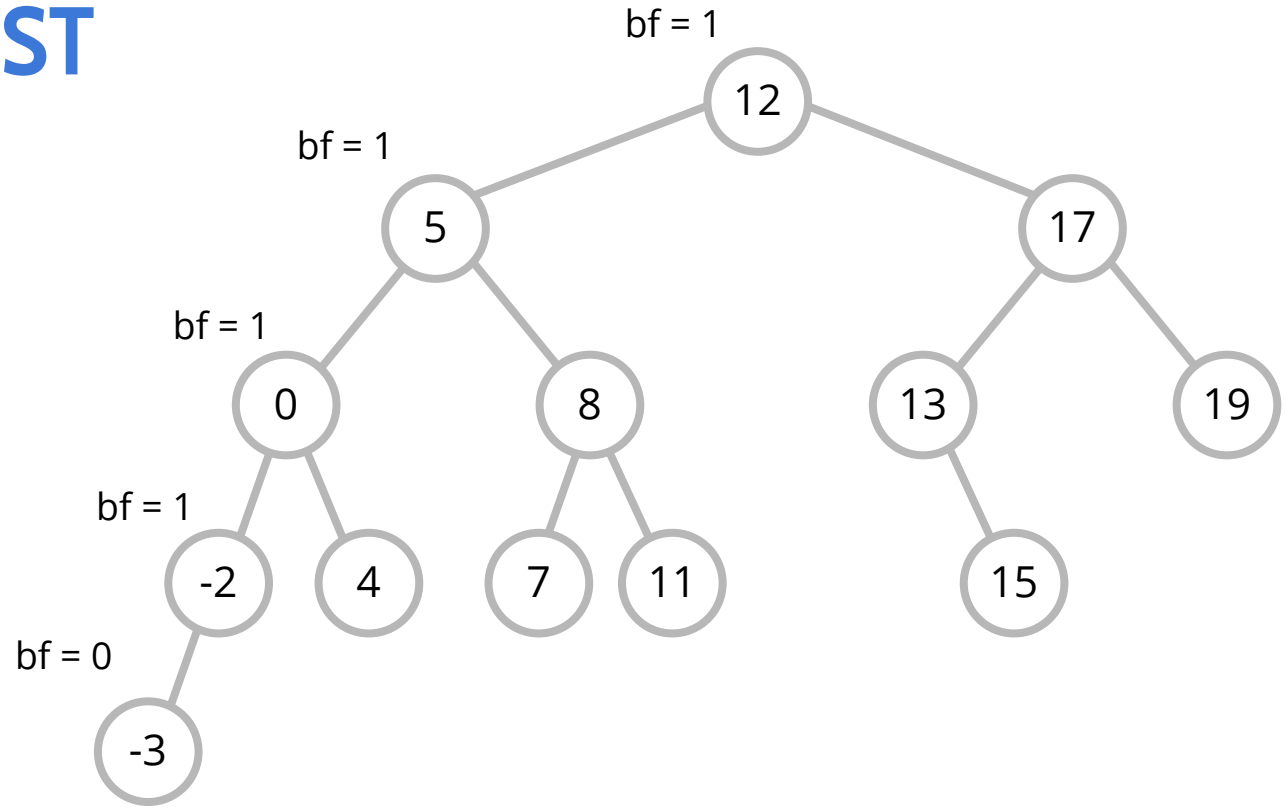
There is no other case.

Insert(v) may indeed cause more than one vertex along the insertion path to have its height increase by one unit.

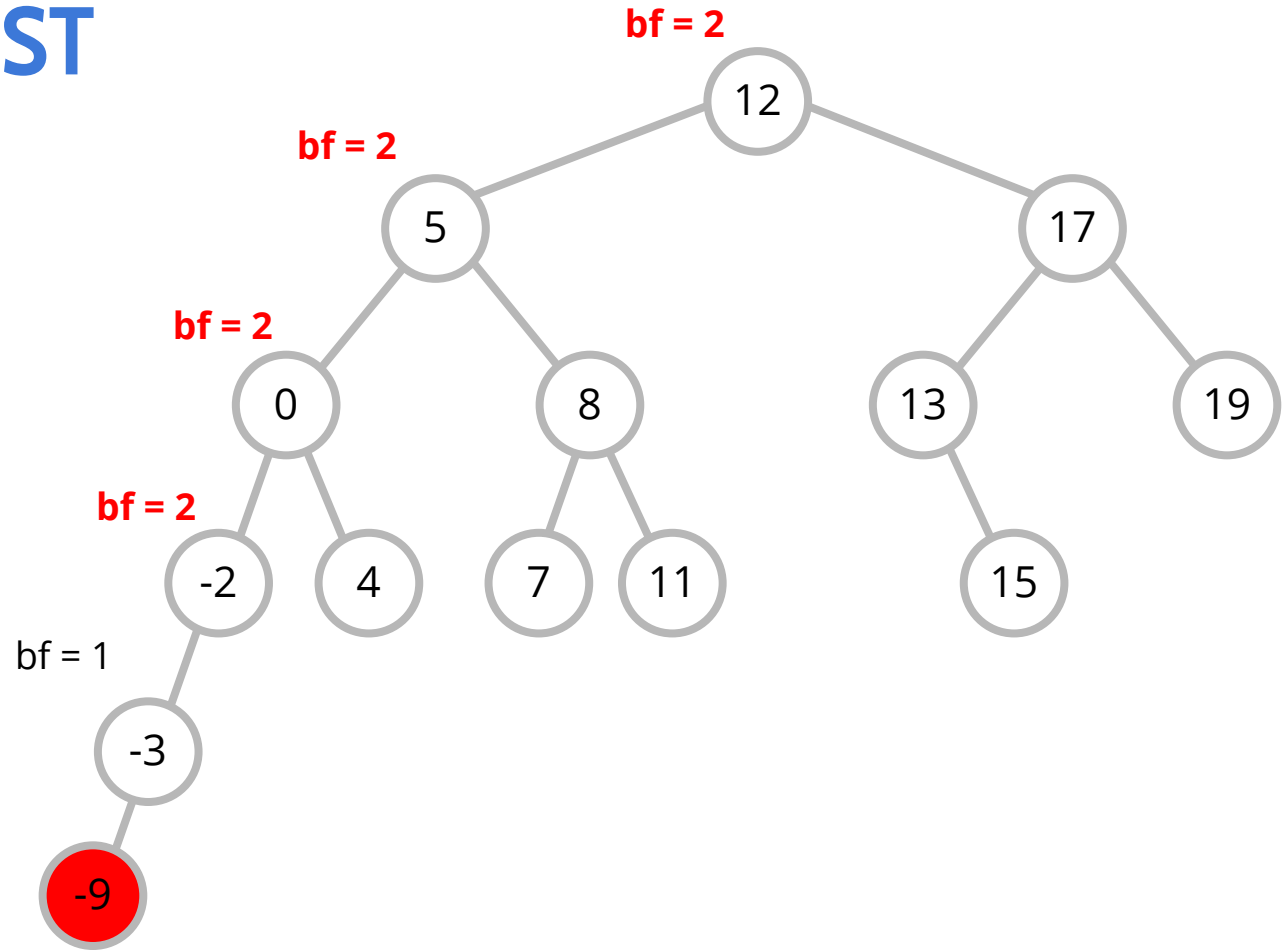
However, as soon as we fix the lowest vertex **x** that is out-of-balance by a bit (+2 or -2), the subsequent one (or double) rotation(s) will make the height of vertex **x** decreases by one again. Thus, any other vertices above **x** along the insertion path will no longer be out-of-balance anymore.

X Esc

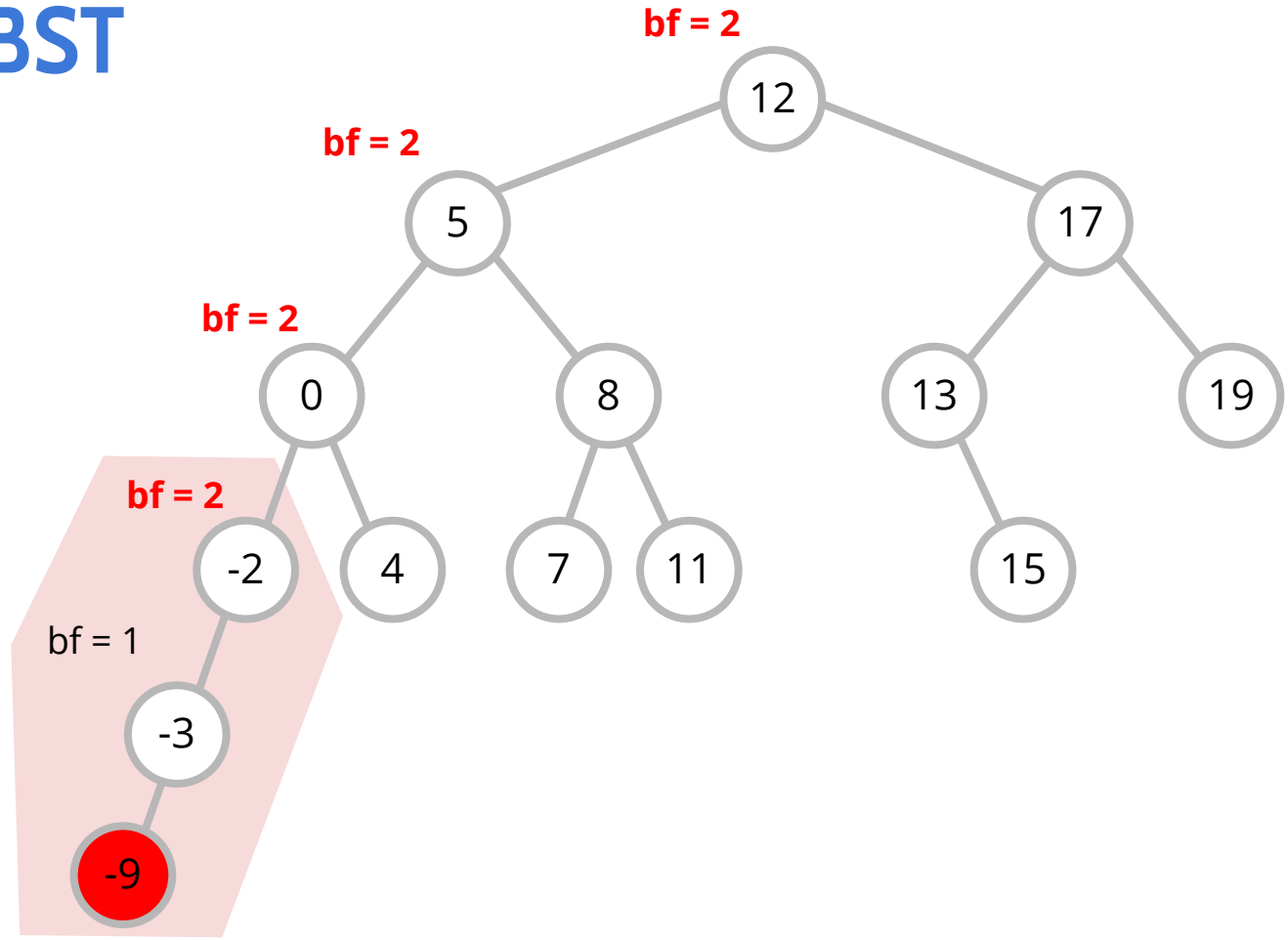
BBST



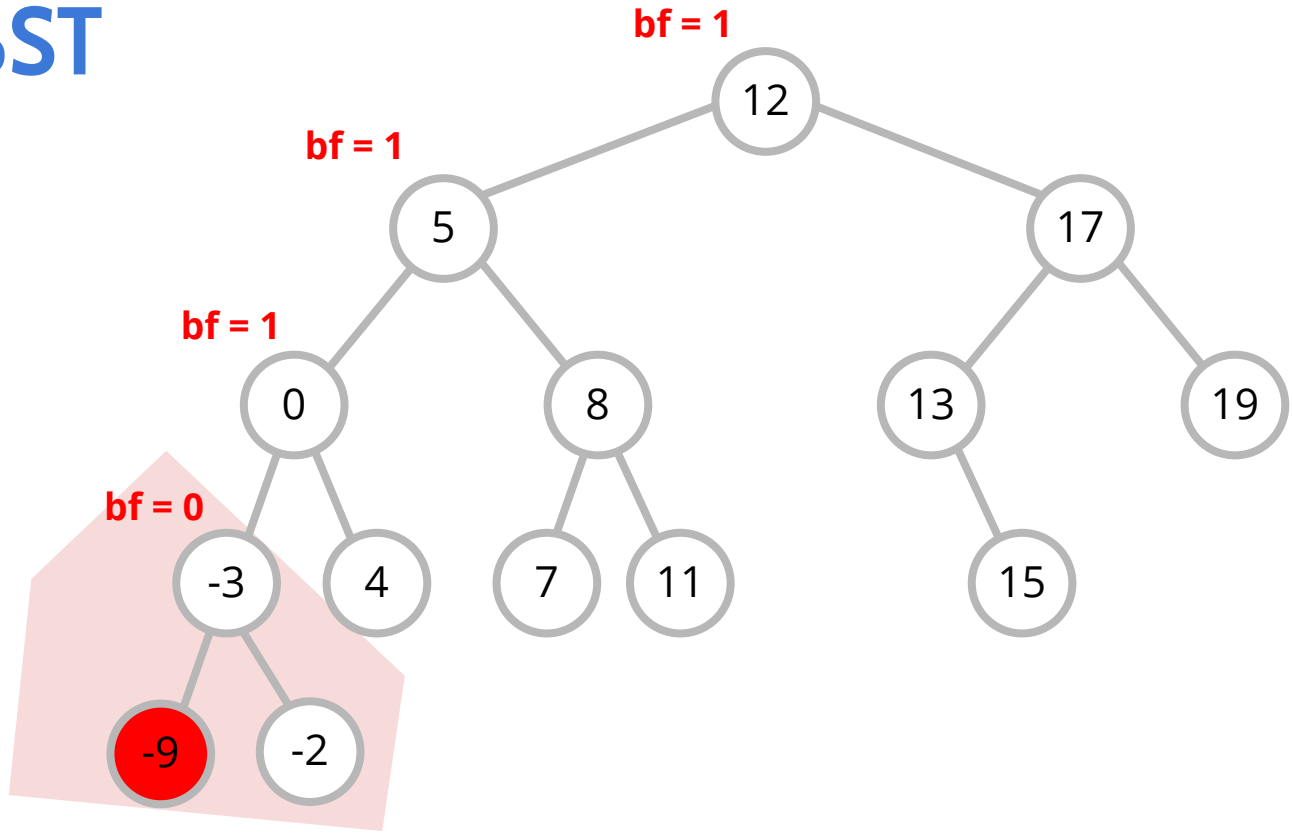
BBST



BBST



BBST



Q2: Deletion

What are the 3 cases for deletion?

- Case 1: 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

Q2: Rotation

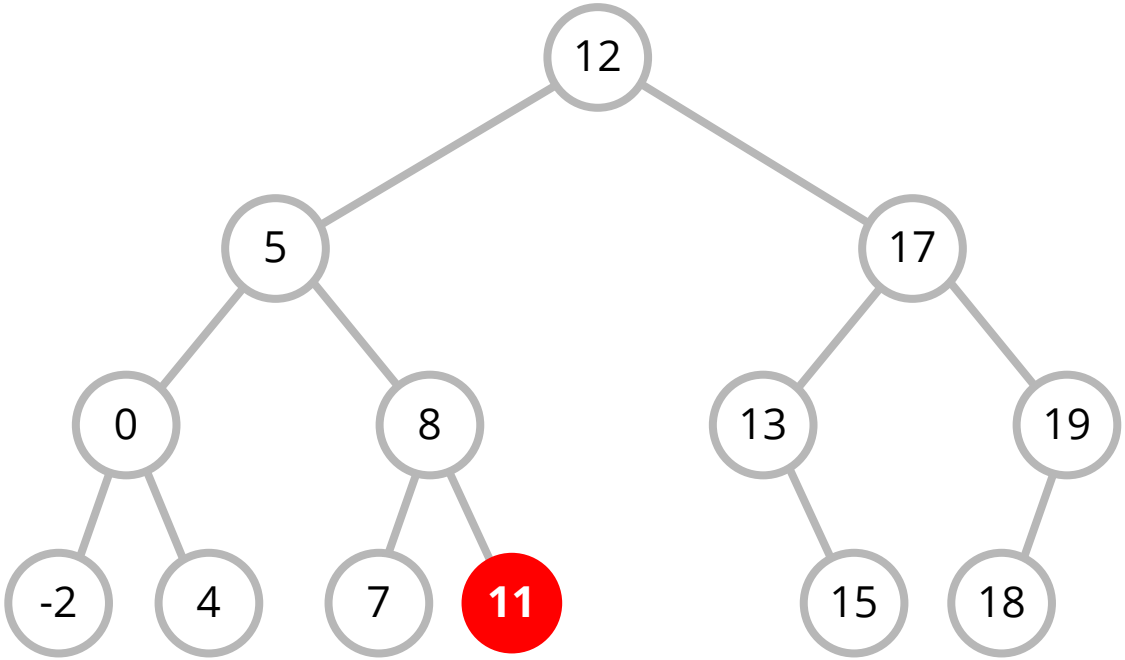
What are the conditions for rotation?

- Imbalanced subtree

Q2: Rotation

a. No rotation happens
    i. No imbalance
b. Only one rotation case
    i. Imbalance that can be *resolved* with one rotation
c. Exactly two rotation case
    i. “Many” rotations
    ii. Skewed BBST

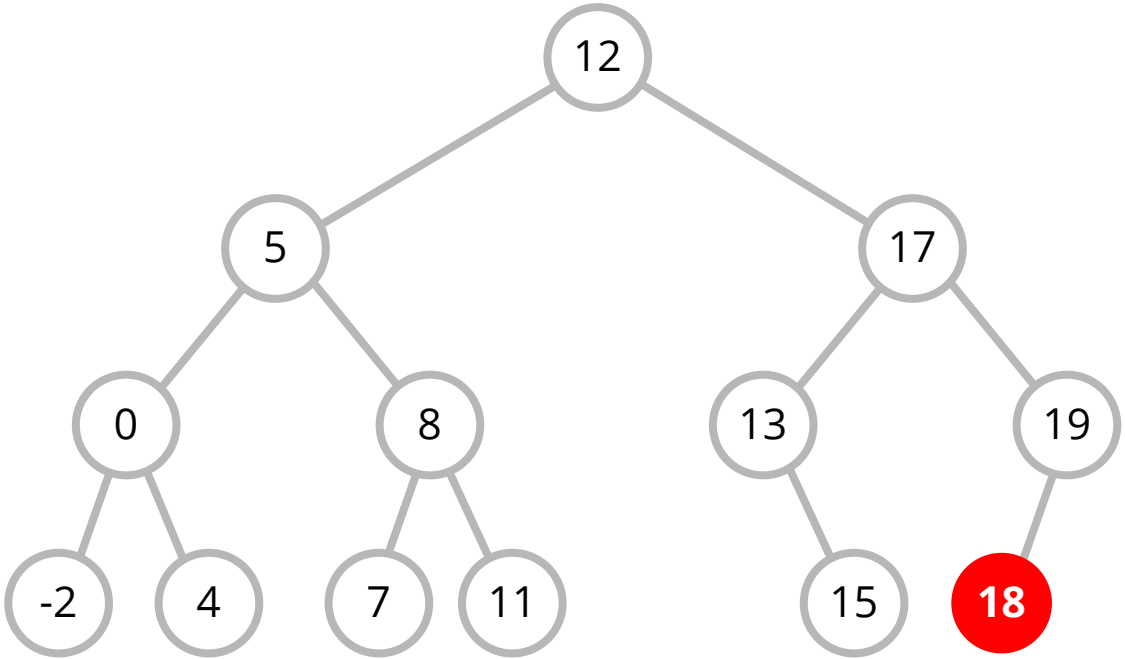
BBST



a. No rotation

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

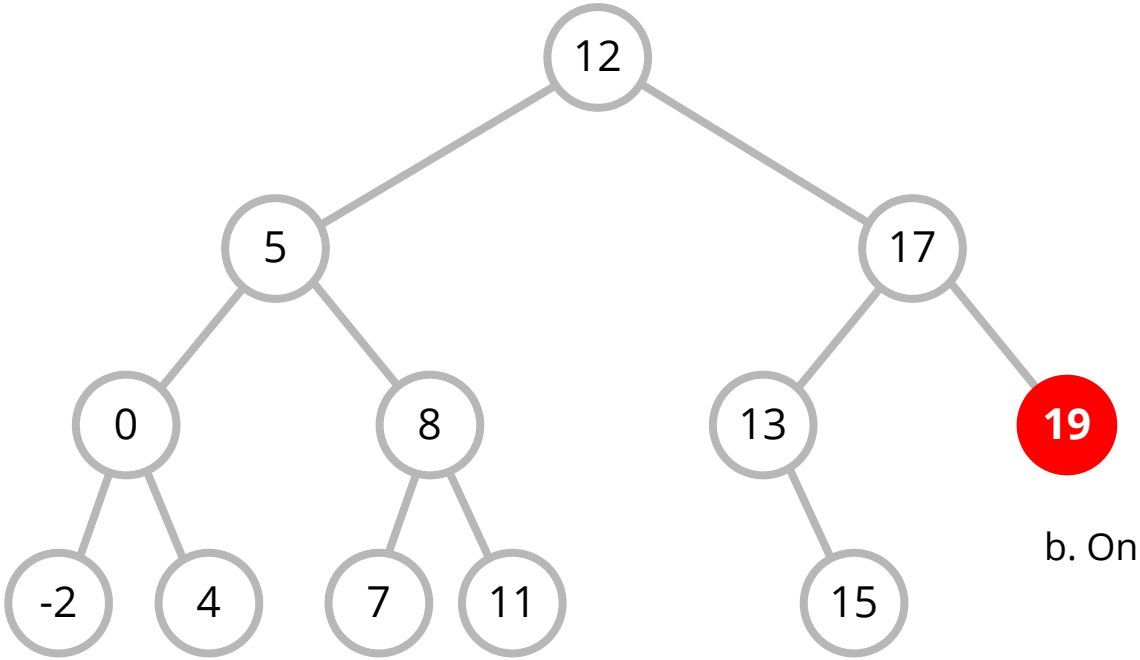
BBST



a. No rotation

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

BBST

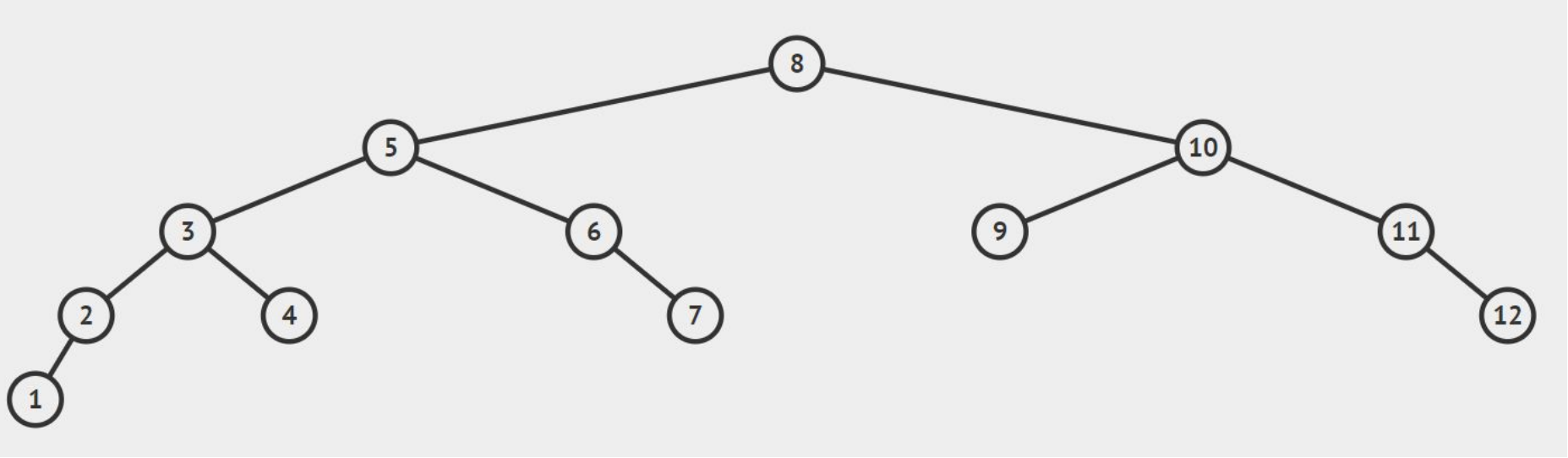


b. One rotation

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

BBST

c. Exactly 2 out of 4 rotations cases
Remove 8



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

Q2: Rotation

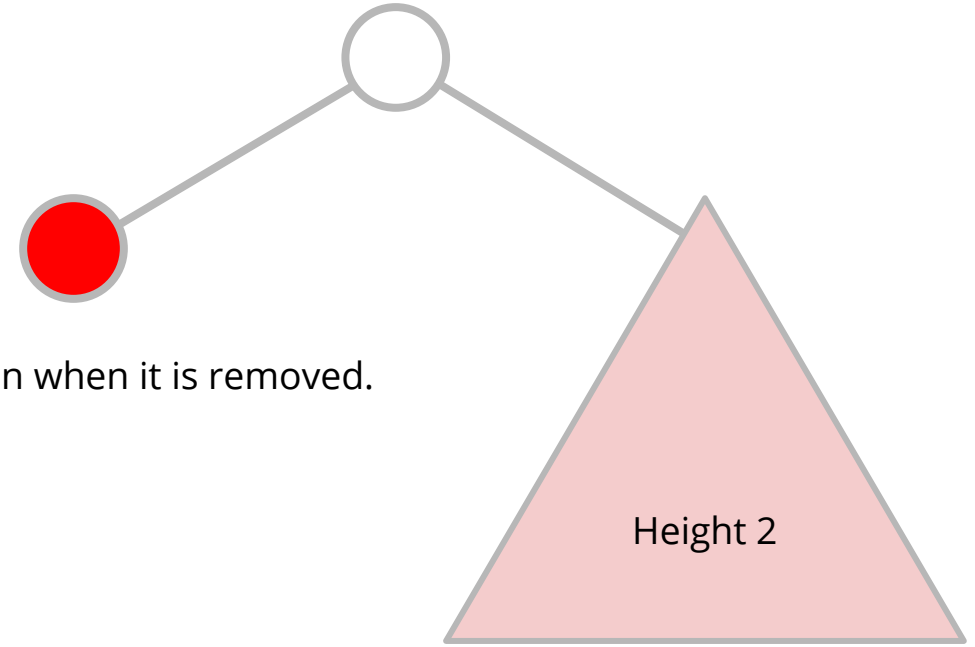
Observation

Each of the rotation case reduces height of the subtree by 1.

To get *2 rotations*, you must make it such that when you remove the vertex,

2 subtrees will be imbalanced.

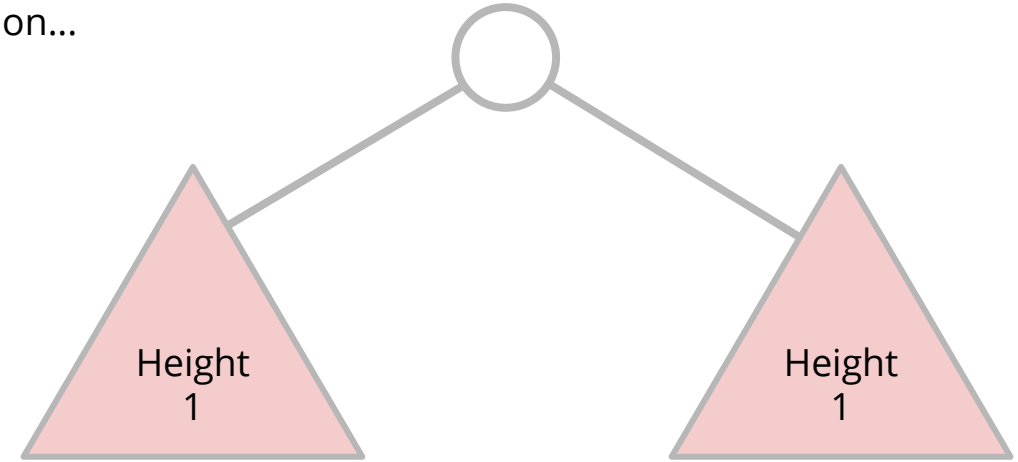
Q2: Rotation



One rotation when it is removed.

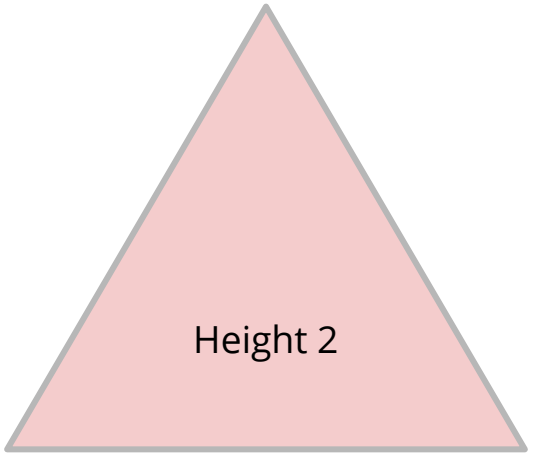
Q2: Rotation

After the first rotation...

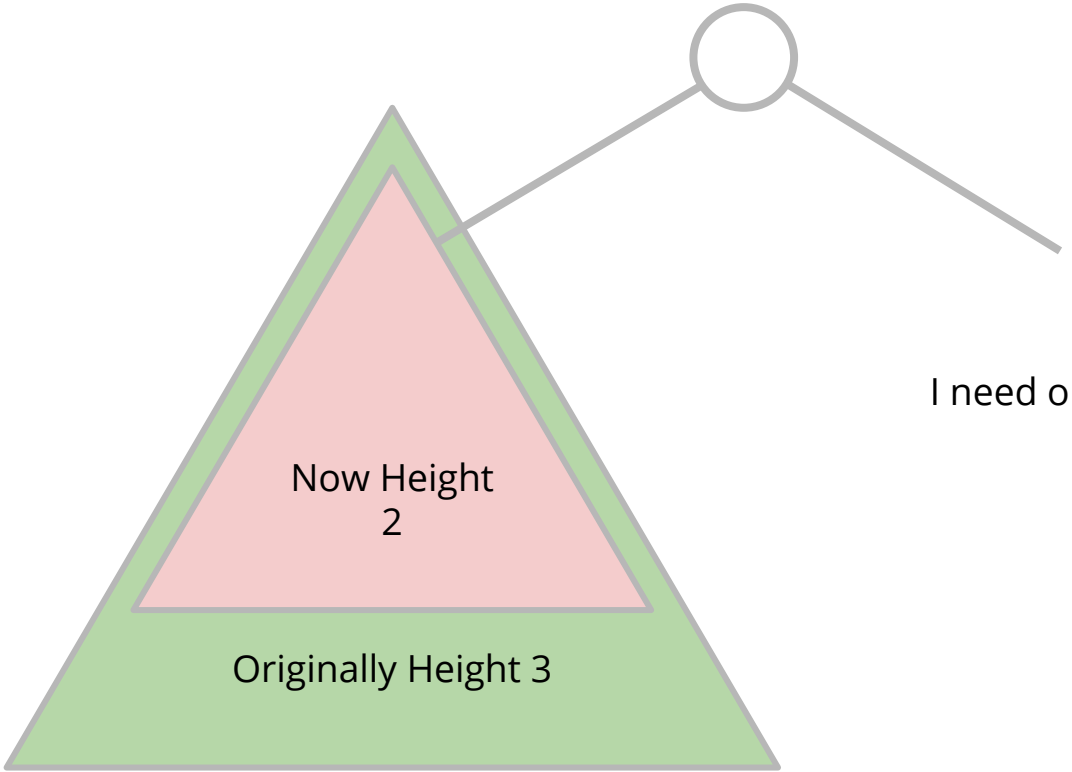


Q2: Rotation

After the first rotation...

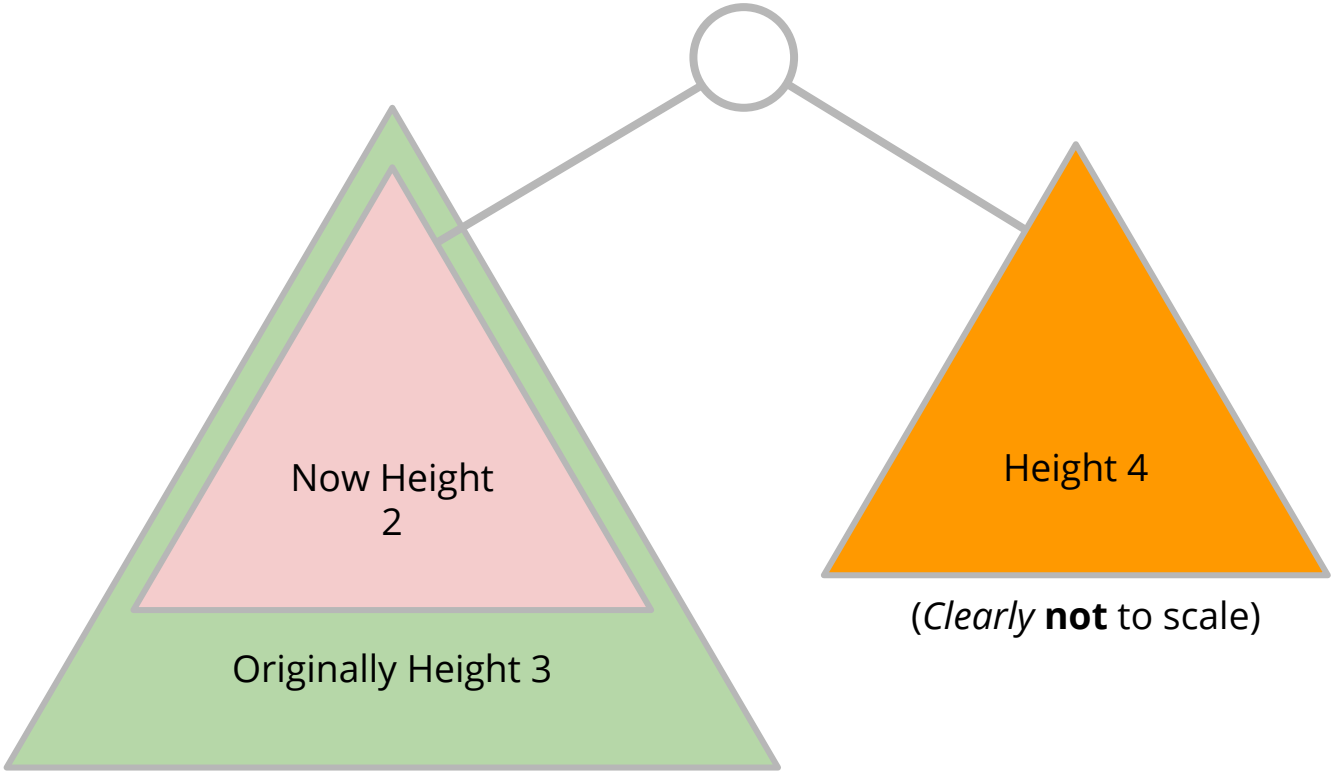


Q2: Rotation



I need one more rotation...

Q2: Rotation



(*Clearly* **not** to scale)

Select and Rank

Augmented BBST

Q3: Select and Rank

Rank (int key)

How many keys that are *smaller than* 'key' in the BBST **+ 1**. (1-indexed)

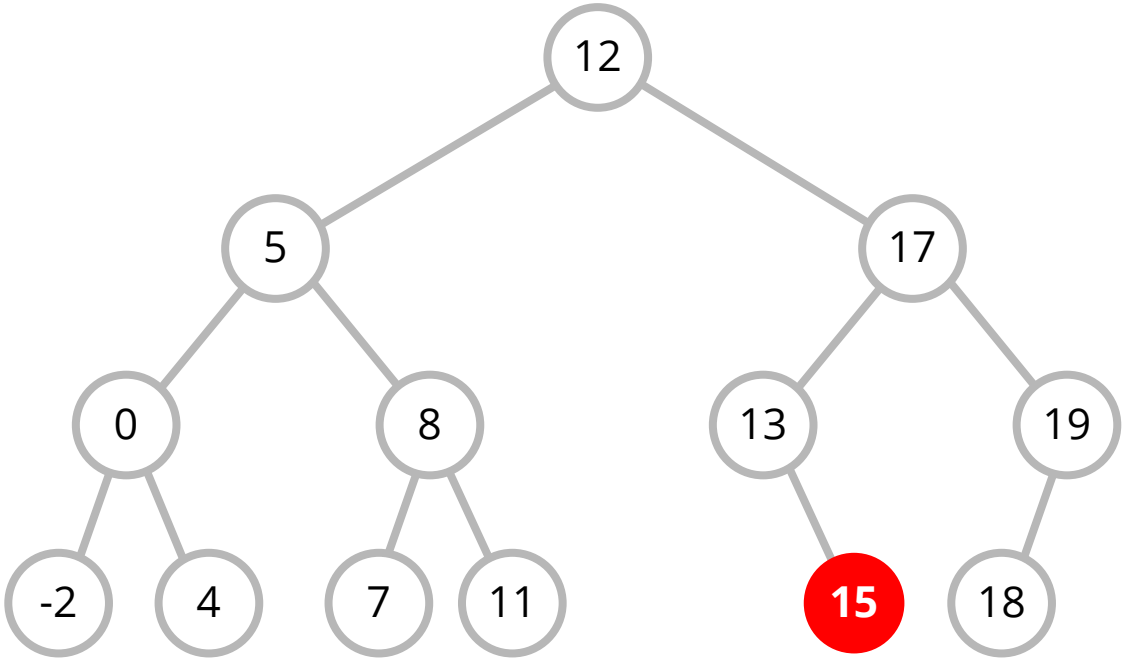
(Eg: smallest key has rank 1, largest key has rank **N**)

Select (int k)

Returns the element that is the **k**th smallest in the BBST.

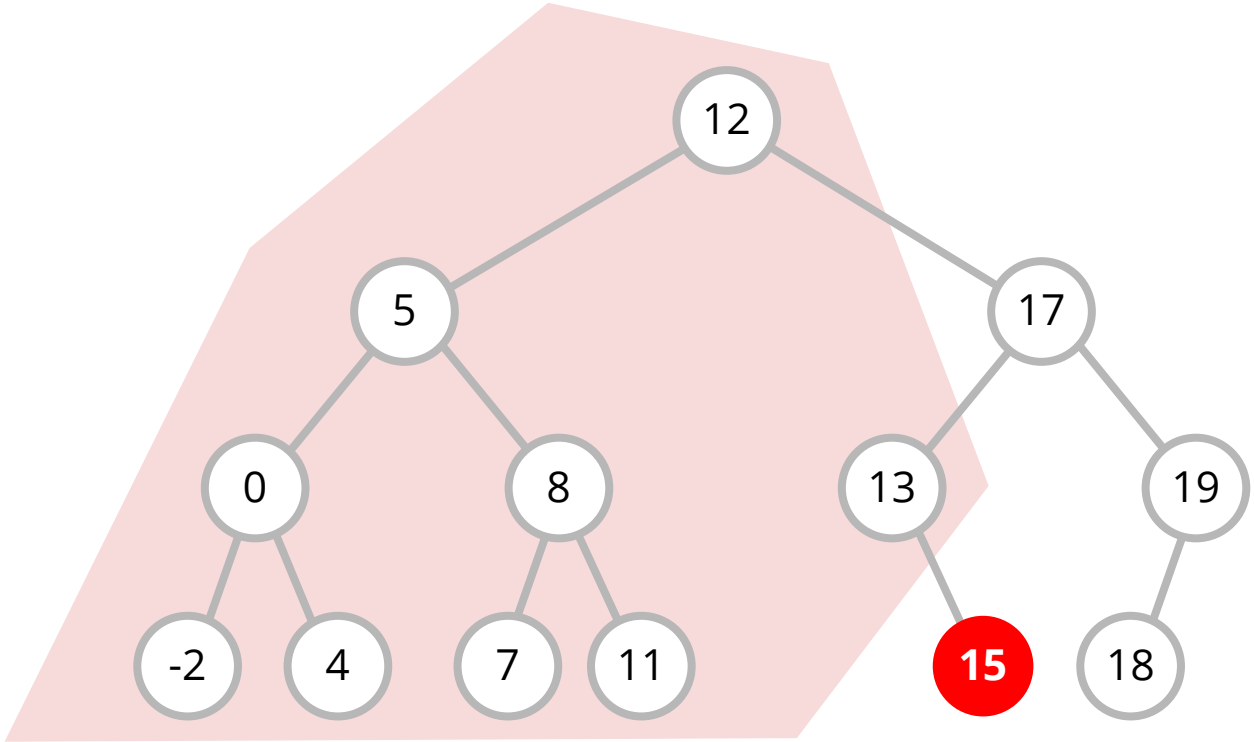
Basically get the element which is rank **k**.

Rank



Get rank of 15.

Rank



Get rank of 15:
Rank 10.

Q3: Select and Rank

Storing Subtree Size

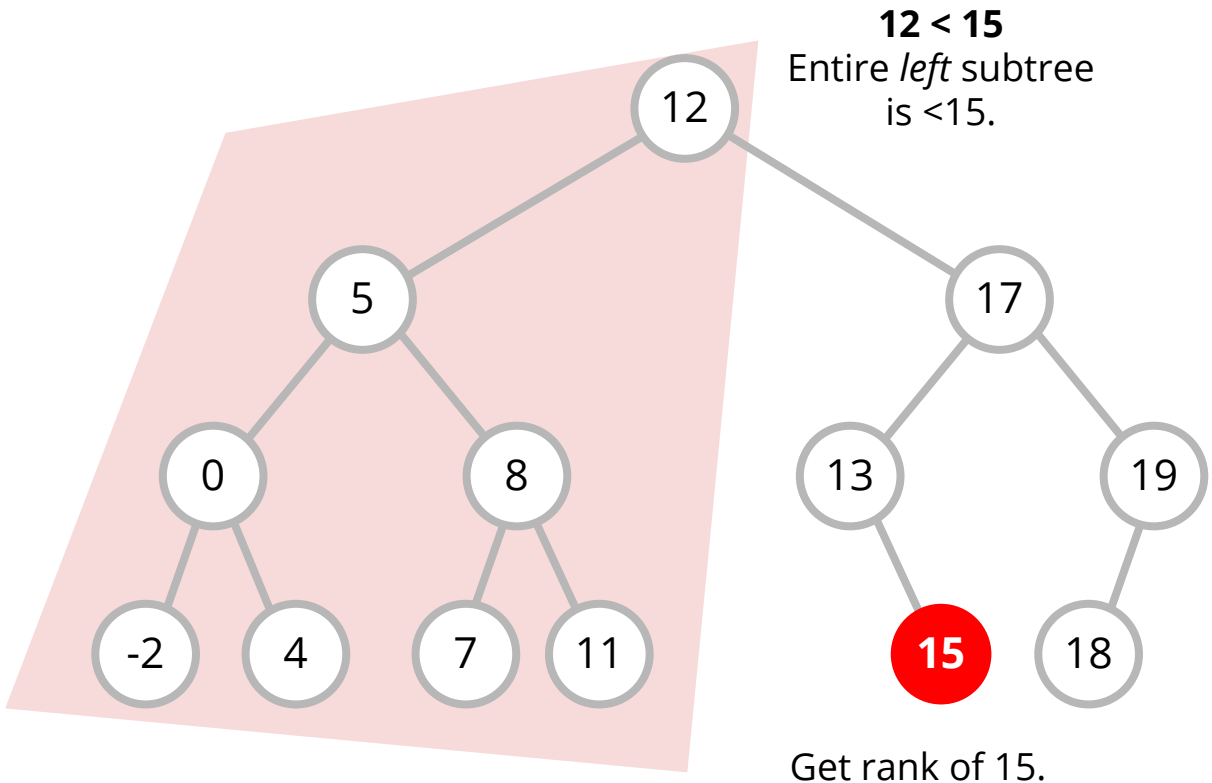
To implement this efficiently, we need to store:

Number of nodes in the subtree.

(Size of subtree)

Same idea as storing height of subtree in AVL Tree

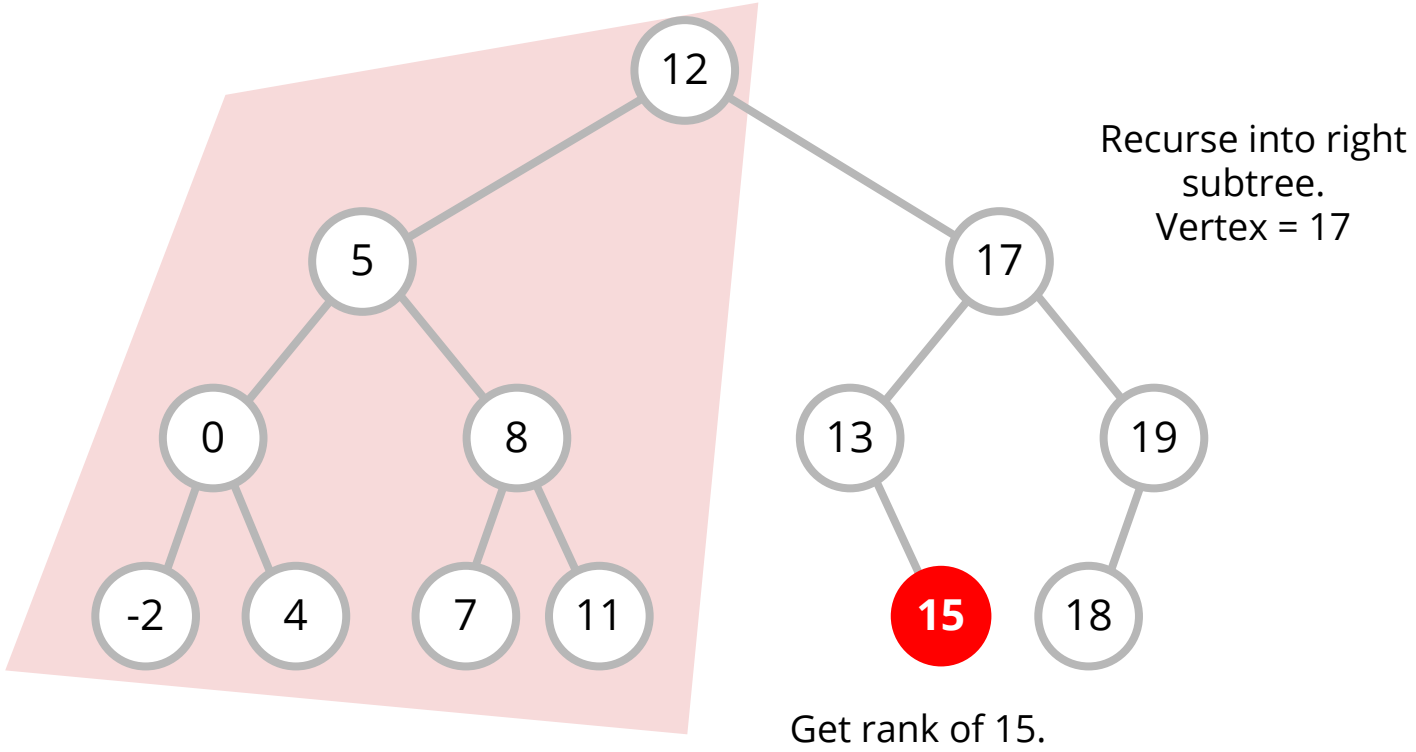
Rank



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

Get rank of 15.

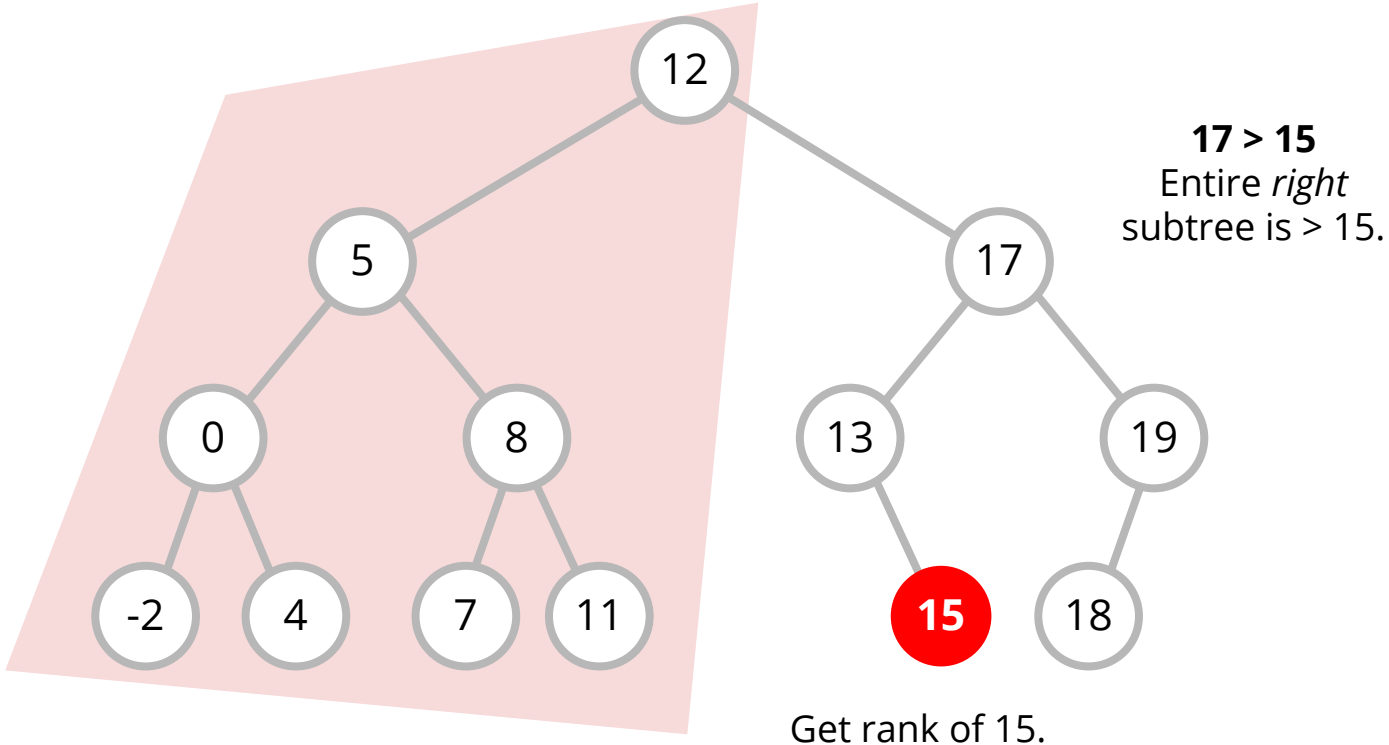
Rank



Recurse into right subtree.
Vertex = 17

Get rank of 15.

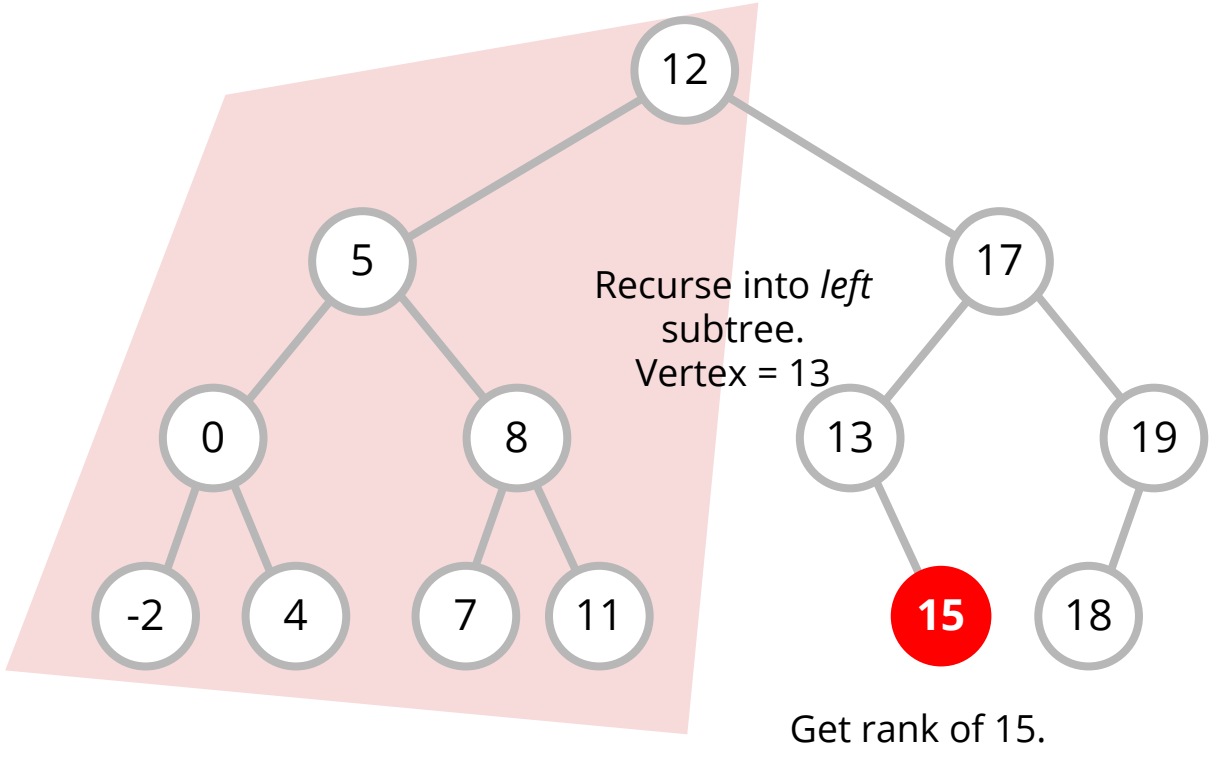
Rank



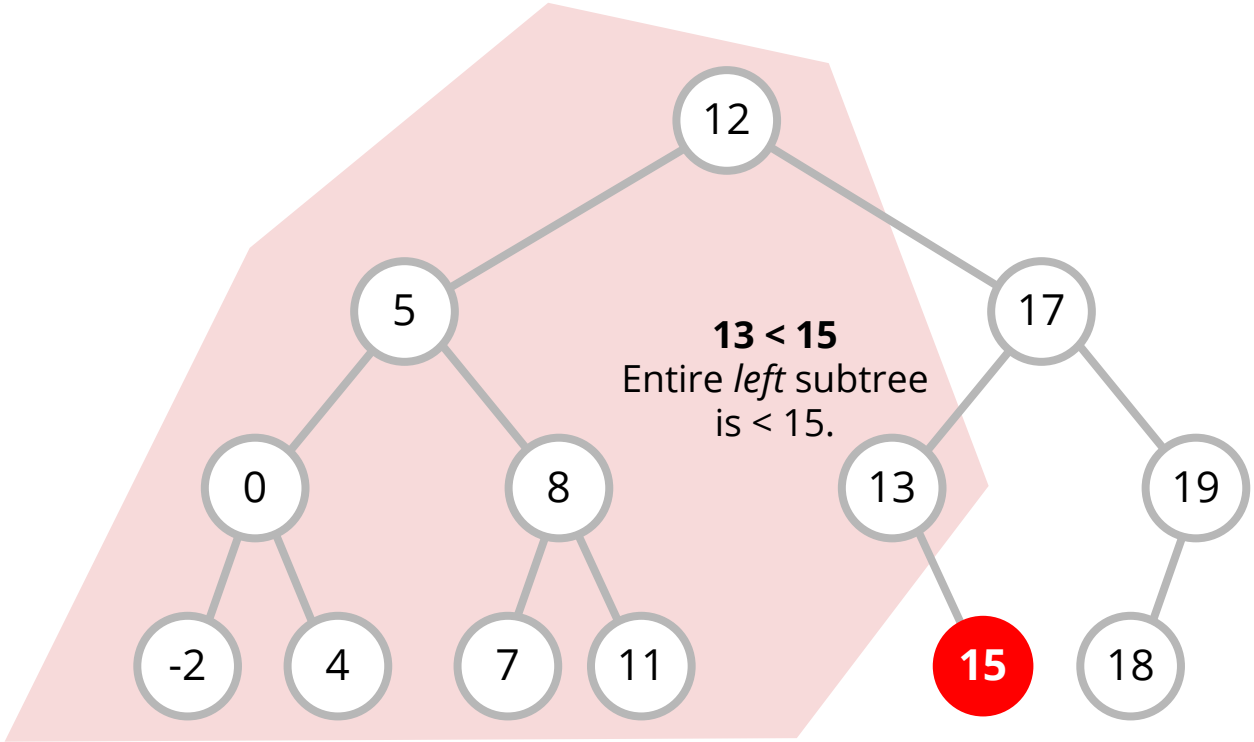
17 > 15
Entire *right* subtree is > 15.

Get rank of 15.

Rank

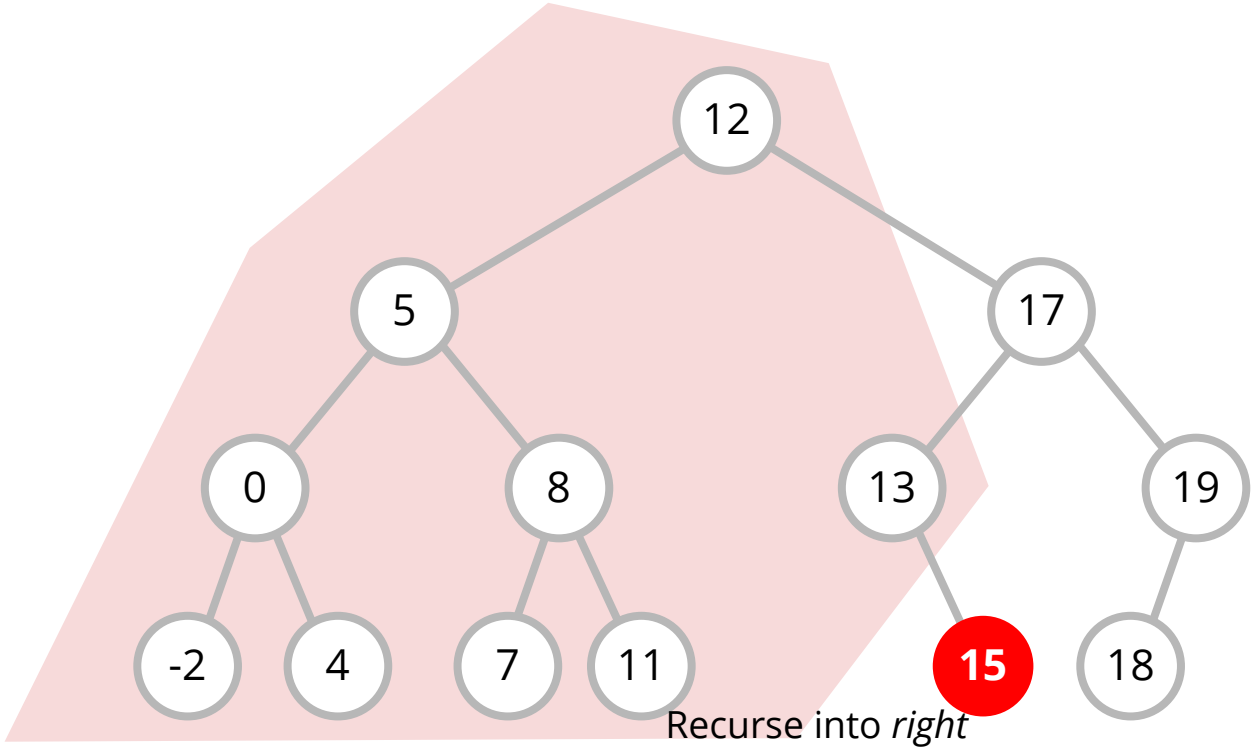

12
5
17
0
8
13
19
-2
4
7
11
15
18
Recurse into *left* subtree.
Vertex = 13
Get rank of 15.

Rank



Get rank of 15.

Rank



Recurse into *right* subtree.
Vertex = 15. **Stop**!

Q3: Rank

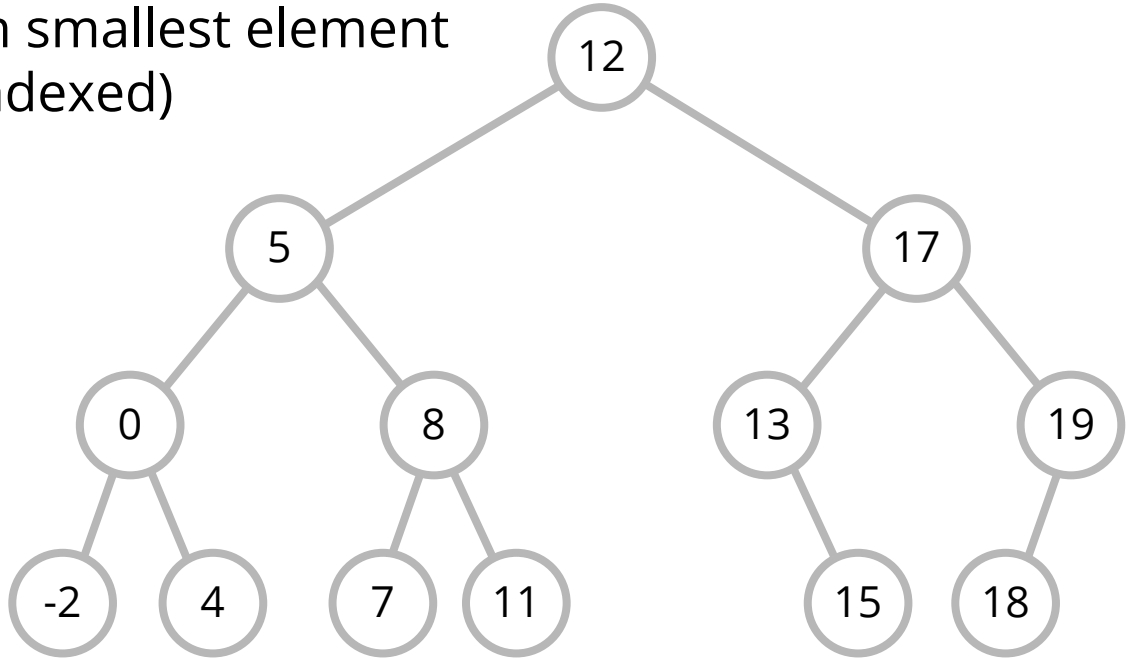
Copied from T08 answer :)

```
int rank(node, v) { // assume that v exists in the BST and size attribute is there
       if (node.key == v) return node.left.size + 1;
  else if (node.key >  v) return rank(node.left, v); // v must be on the left
  else                    return node.left.size+1 // v is > node's left and the node
                                 + rank(node.right, v); // and plus this rank
}

// select is very similar to rank and similar with QuickSelect
```

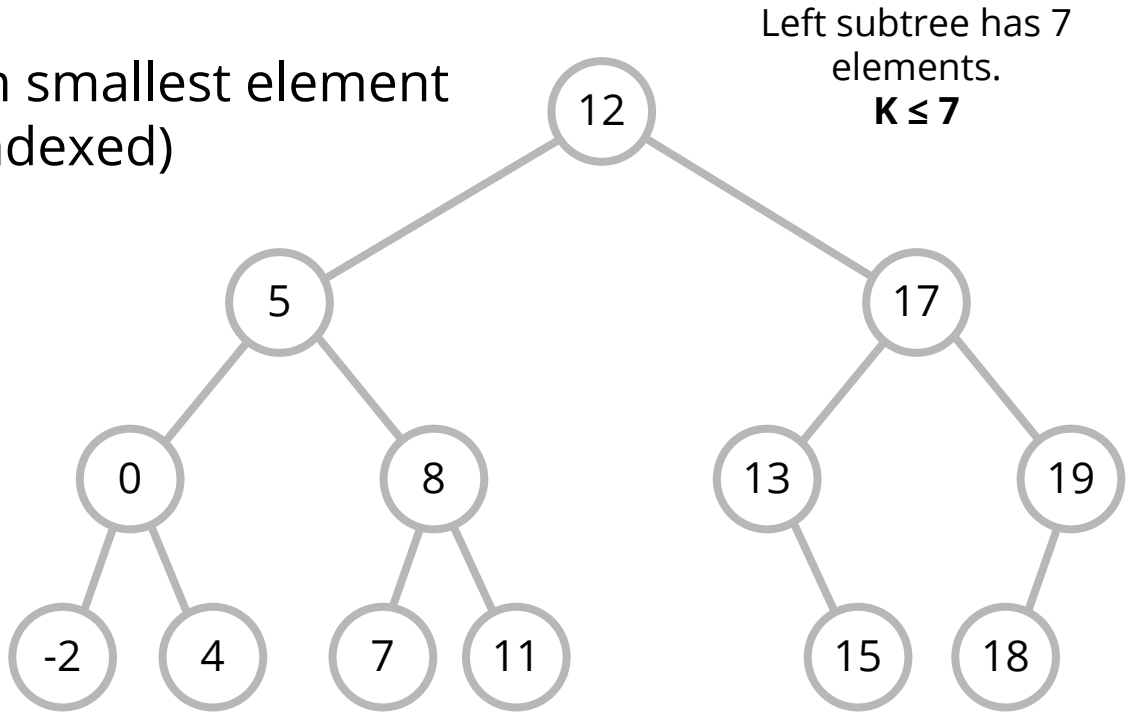
Select

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



Select

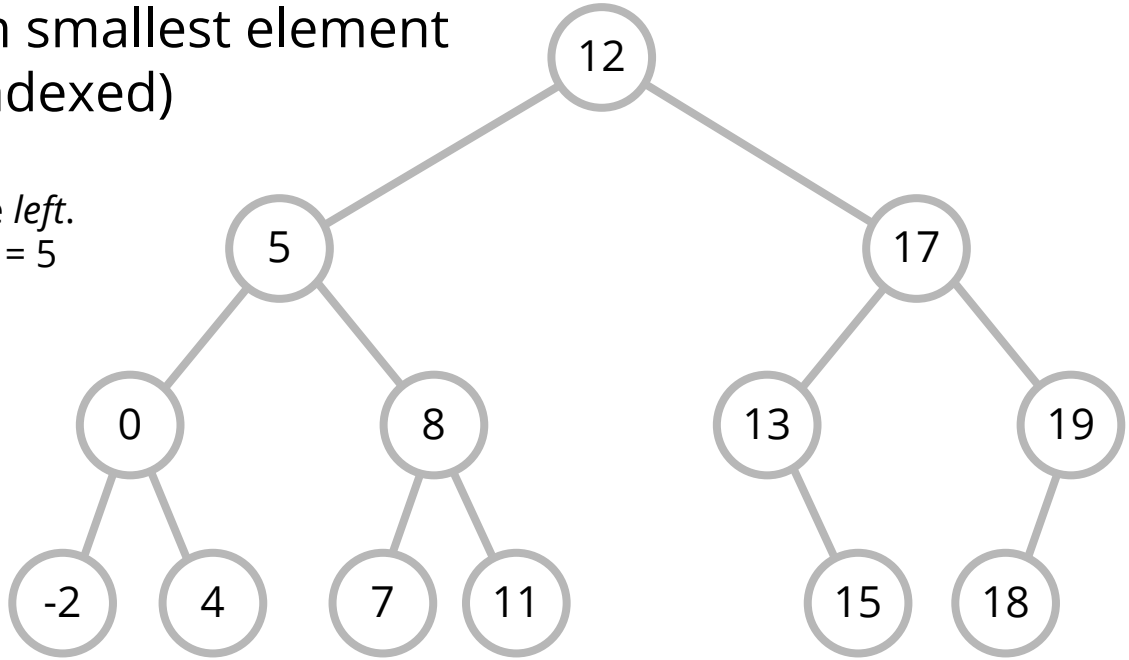
Select **K**-th smallest element
K = 5 (1 indexed)



Select

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

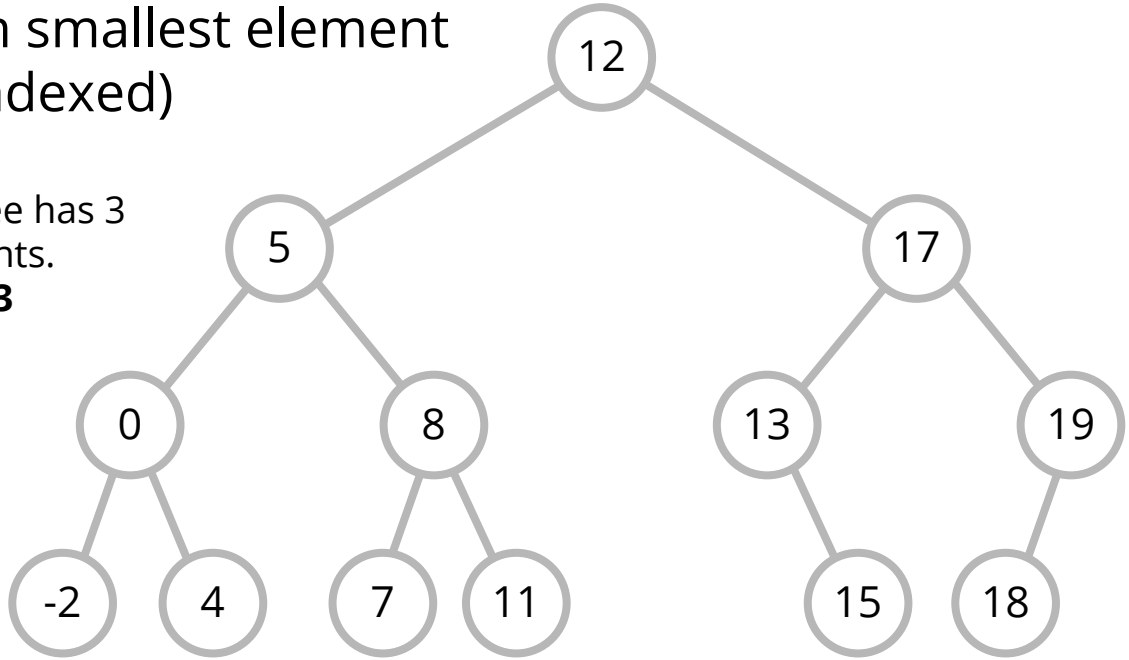
Recurse *left*.
Vertex = 5



Select

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

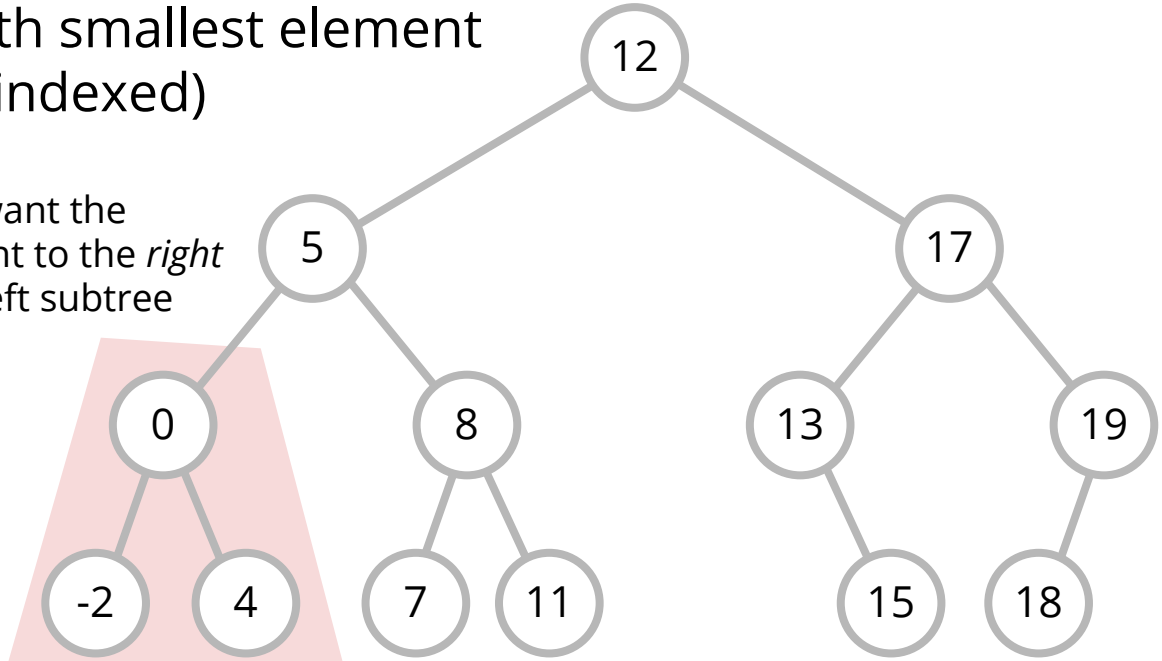
Left subtree has 3 elements.
K > 3



Select

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

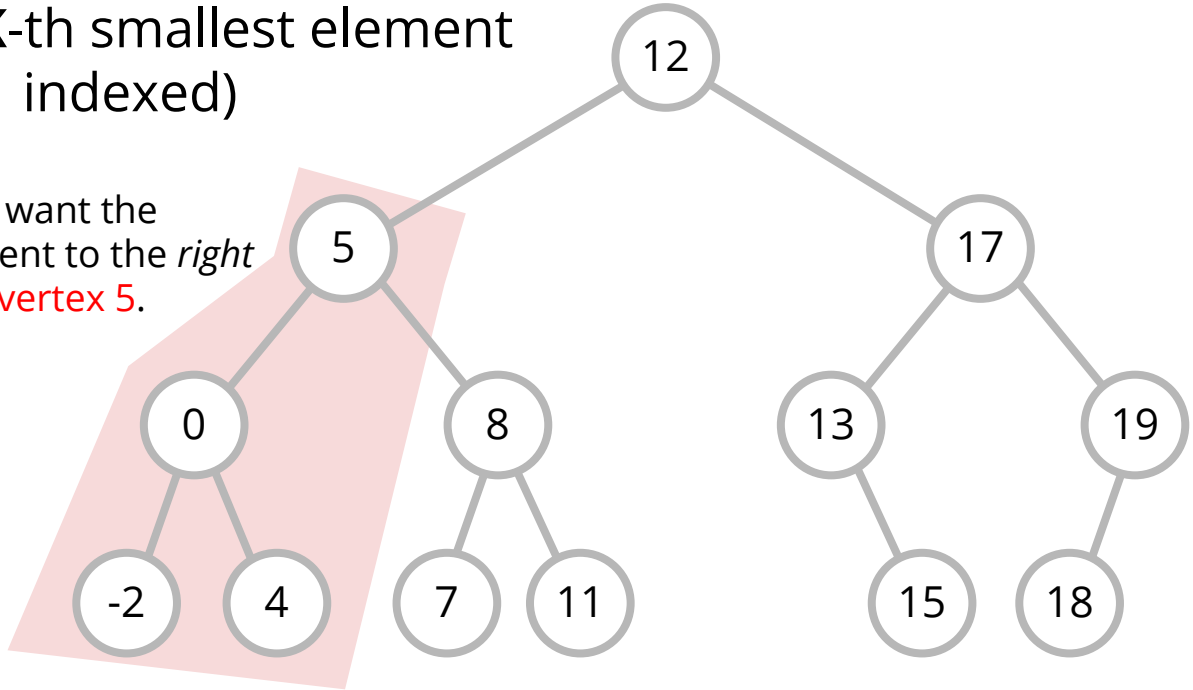
We want the
K-3 element to the *right*
of the left subtree



Select

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

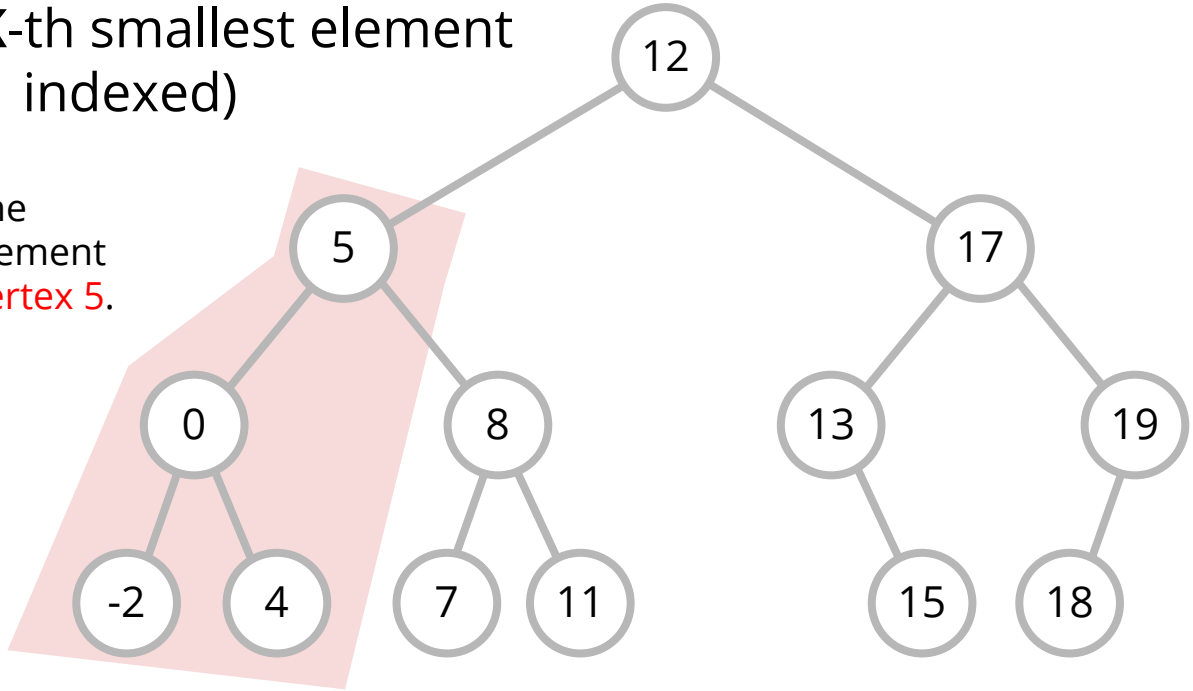
We want the
K-4 element to the *right*
of vertex 5.



Select

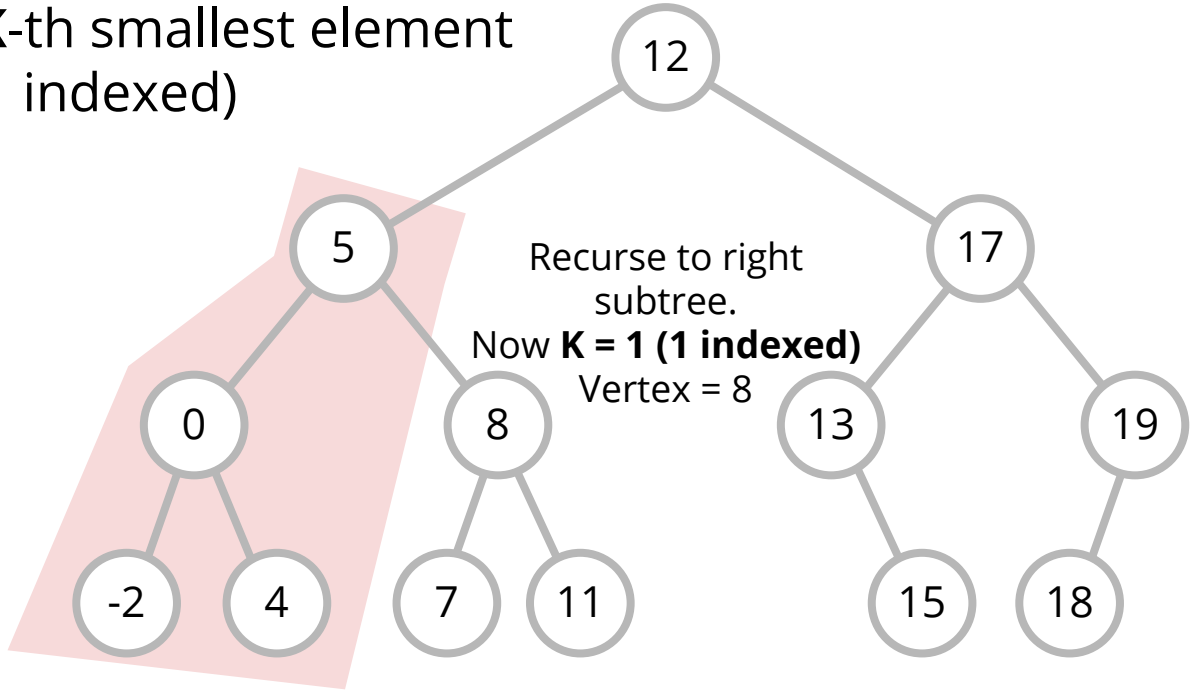
Select **K**-th smallest element
K = 5 (1 indexed)

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



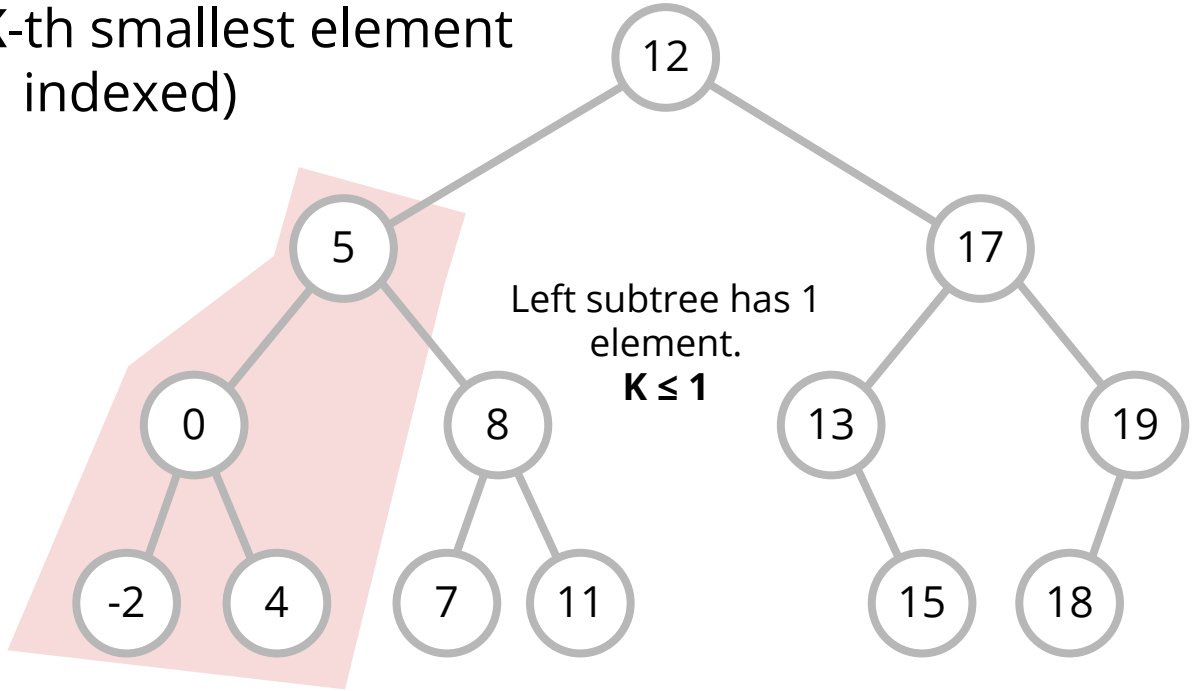
Select

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



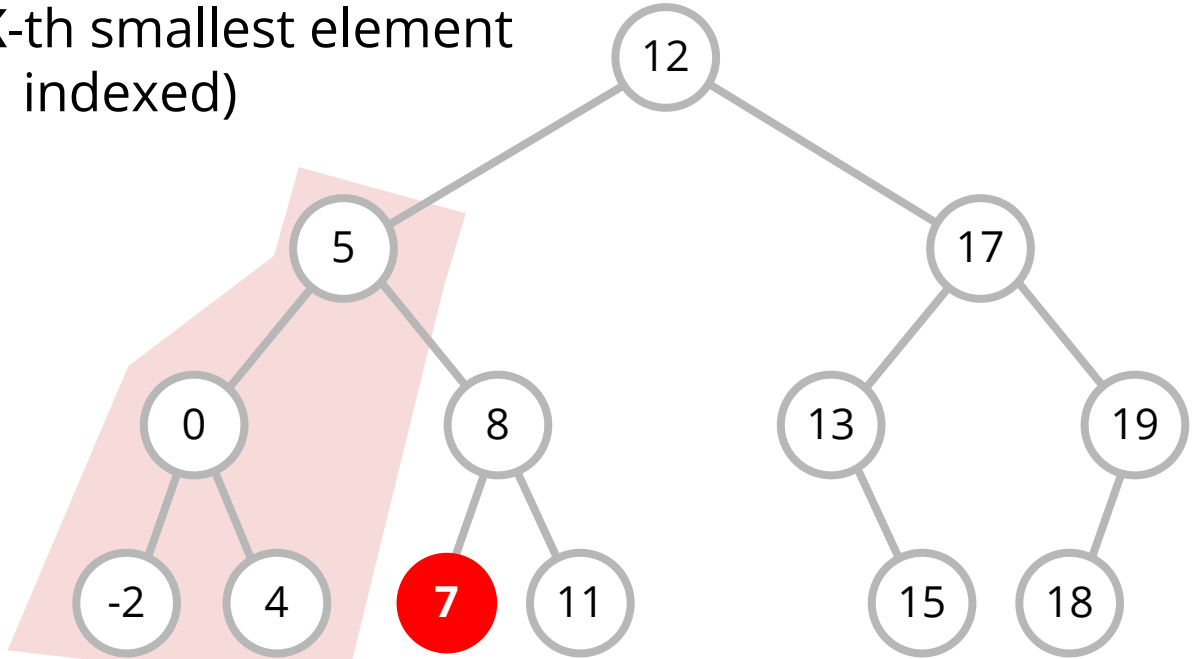
Select

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



Select

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



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

Q3: Select

Copied from T07 answer :)

```
int select(node, k) { // assume size attribute is there
  int q = node.left.size;
       if (q+1 == k) return node.key; // this node has rank k
  else if (q+1 >  k) return select(node.left, k); // rank k is in the left subtree
  else               return select(node.right, k-q-1); // do you understand why?
}
```

PS4

Baby Names

PS4A

Brute Force

Same, if your brute force gets WA.

Check your interpretation with one of the teaching staff.

PS4B

Hint

The first letter of the baby names are ***evenly distributed***.

This suggests that there is about **N/26** baby names starting with each letter.

PS4B

Quick Calculations

N = 20000, Q = 20000

N/26 = 770

Lets say each query takes N/26:

Total runtime: NQ/26 = 20000*770 = **15.3 mil**

PS4B

Helpful functions

- distance function
  - Returns the distance between 2 iterators

```
set<string>::iterator it_a, it_b;
distance(it_a, it_b);
```

PS4B

Helpful functions

- lower_bound/upper_bound function
  - Helps you find items within a set

```
set<string> s;
set<string>::iterator it = s.lower_bound("MIAO");
set<string>::iterator it2 = s.upper_bound("WOOF");
```

PS4C

Augmented BBST

- Strongly recommended to try
- **Rank** and **Select** are *very common* BBST operations but are not in C++ ***STL***.
  - Your code might serve you well in future modules!
- Good test of whether you understand BBST.
  - Can modify/use the provided implementation.