NUS
National University of Singapore
School of Computing

Tutorial 7: BST and bBST

October 10, 2022

Gu Zhenhao

** Partly adopted from tutorial slides by Wang Zhi Jian.*

Ordered Map ADT

Why do we need the Ordered Map ADT?

Ordered Map ADT

- **Purpose**: keep both searching and comparing fast.
- **Operations**:
  1. `insert(i)`, `delete(i)`.
  2. `find(i)`: Check if a key exists.
  3. `predecessor(i)`, `successor(i)`: find the keys next to `i` in the sorted list.
  4. …

We assume all keys are unique.

Implementation of Ordered Map

- As **unsorted array**:

6	4	9	2	5	8

- Finding a key is very slow.

`insert`	`find`	`predecessor`
$O(1)$	$O(n)$	$O(n)$

Implementation of Ordered Map

- As **sorted array**:

2	4	5	6	8	9

- Can use **binary search** to find keys, faster to find predecessor/successors.
- Slower insertion (need a step in **insertion sort**).

`insert`	`find`	`predecessor`
$O(n)$	$O(\log n)$	$O(1)$

Implementation of Ordered Map

- As **hash map**:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		2		4	5	6		8	9

- Fast insertion/deletion and finding.
- Need to check all keys for predecessor/successor!

`insert`	`find`	`predecessor`
$O(1)$	$O(1)$	$O(n)$

Summary

Operations	Unsorted Array	Sorted Array	Hash Map
`insert`	$O(1)$	$O(n)$	$O(1)$
`delete`	$O(n)$	$O(n)$	$O(1)$
`find`	$O(n)$	$O(\log n)$	$O(1)$
`predecessor`	$O(n)$	$O(1)$	$O(n)$

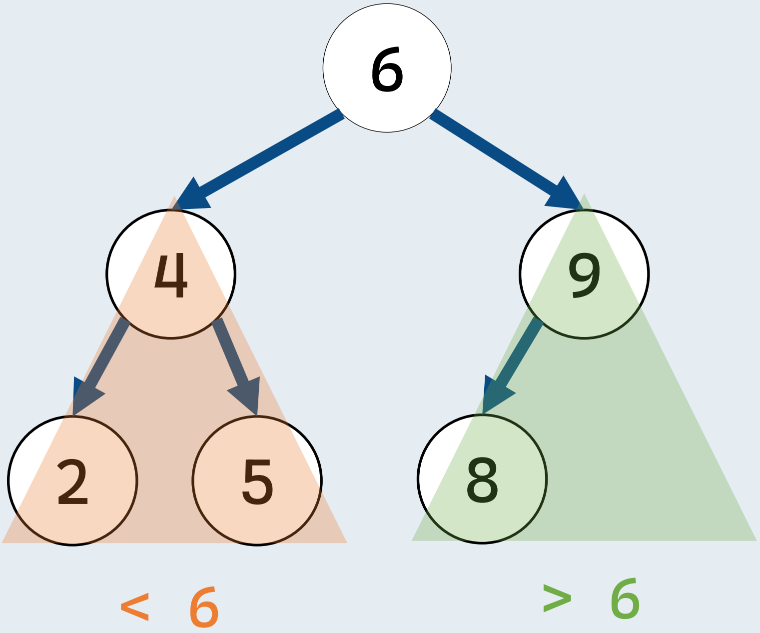
Better at **comparing**

Better at **searching/insert**

- Is there a way to take the merits of these two?

Implementation of Ordered Map

- As binary tree:



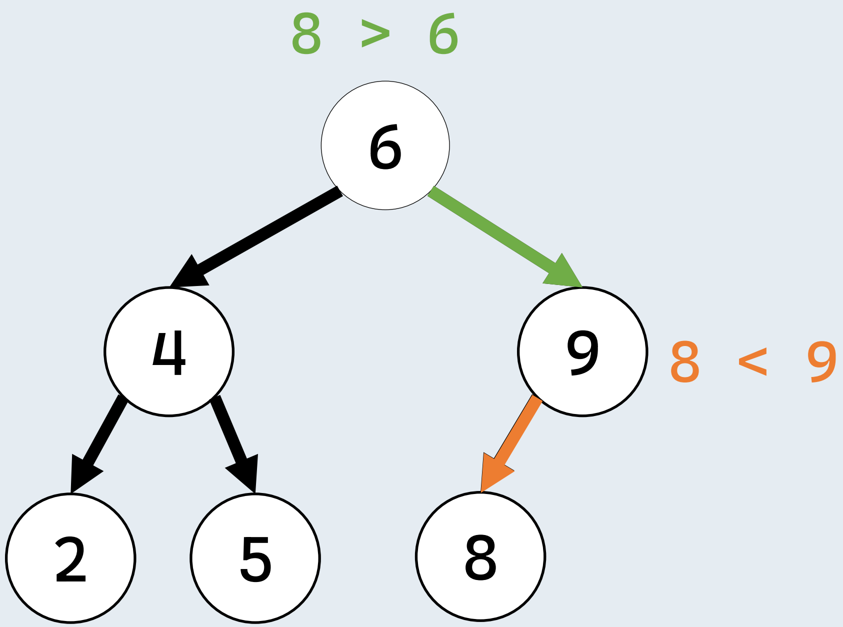
For **every node**:

Each node in **left subtree** is smaller than it,

Each node in the **right subtree** is larger than it.

Implementation of Ordered Map

- `find(8)`:



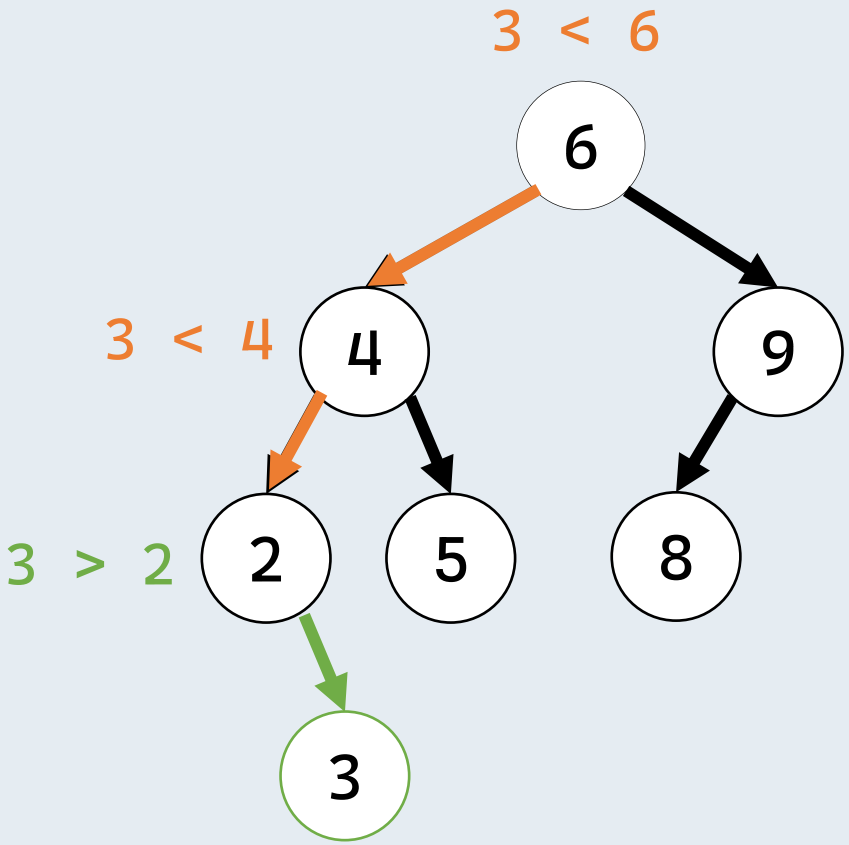
Kind of like **binary search**!

To **`find`** a key:

- Start from **root**,
- If the node is null, the key is not found.
- If the searched key is larger than root, search **right subtree**.
- Otherwise search **left subtree**.

Implementation of Ordered Map

- `insert(3)`:

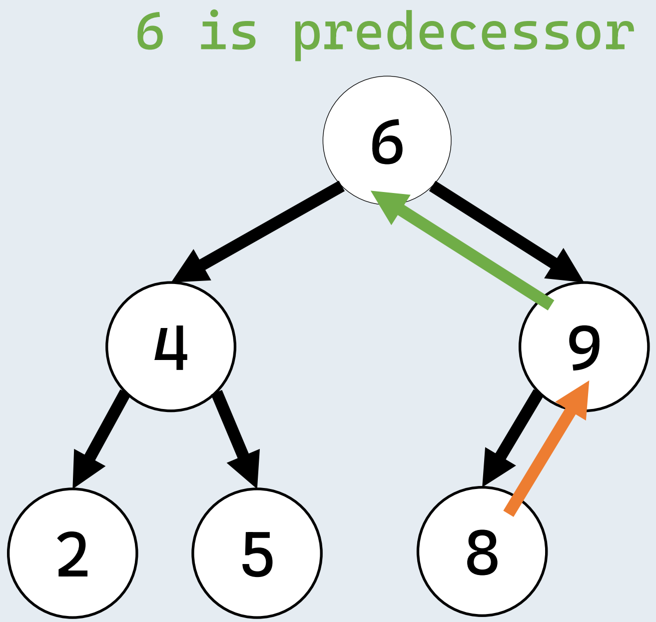


To **`insert`** a key:

- First find the key.
- If it doesn't exist, insert as a leaf node.

Implementation of Ordered Map

- `predecessor(8)`:

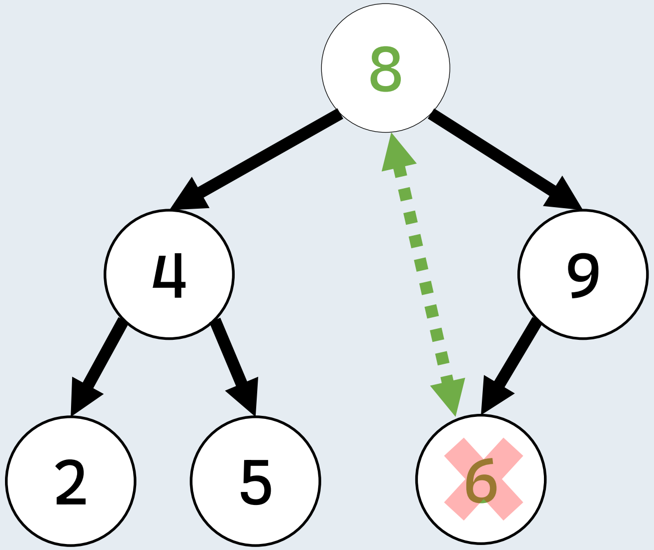


To find **`predecessor`** of a key:

- If it has a left child, find max in left subtree.
- Otherwise
  - if it is a right child, the predecessor is its parent.
  - Otherwise the predecessor is its "first left ancestor".

Implementation of Ordered Map

- `delete(6)`:



To **`delete`** a key:

- If it is a leaf, simply remove the node.
- If it has only one child, remove the node and attach the child to the node's parent.
- If it has 2 children, swap the node with its successor, then remove.

Summary

Operations	Sorted Array	Hash Map	BST
`insert`	$O(n)$	$O(1)$	$O(h)$
`delete`	$O(n)$	$O(1)$	$O(h)$
`find`	$O(\log n)$	$O(1)$	$O(h)$
`predecessor`	$O(1)$	$O(n)$	$O(h)$

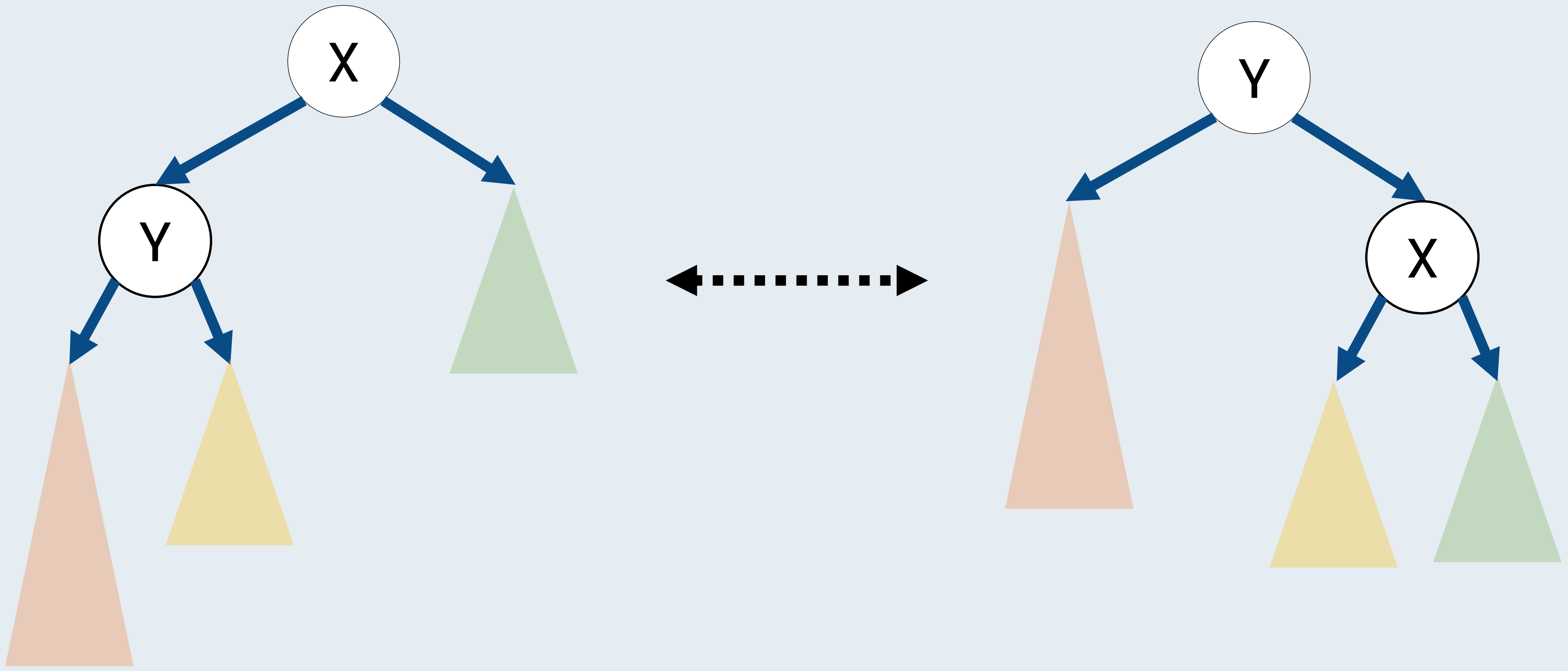
How to reduce the height of tree in worst case?

Problem:

- The height of the tree h can be $O(n)$ in the worst case!

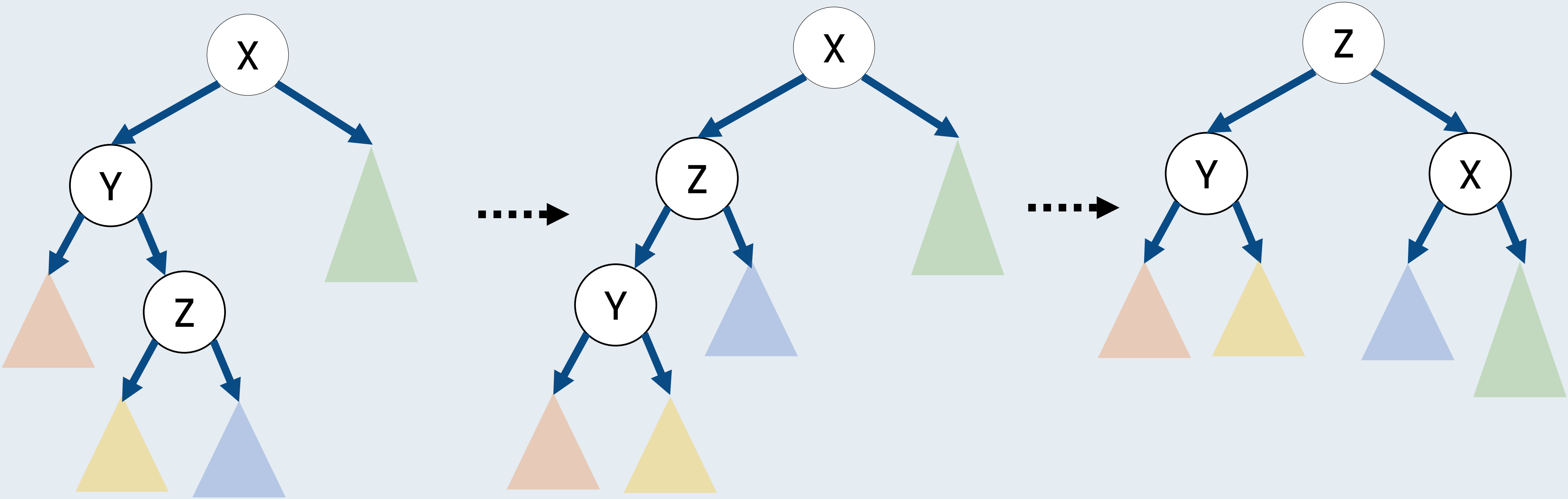
Rebalancing

- When tree is unbalanced (has a node where the height of left subtree and right subtree differ more than 1), needs rebalancing!
- **Single rotations**:



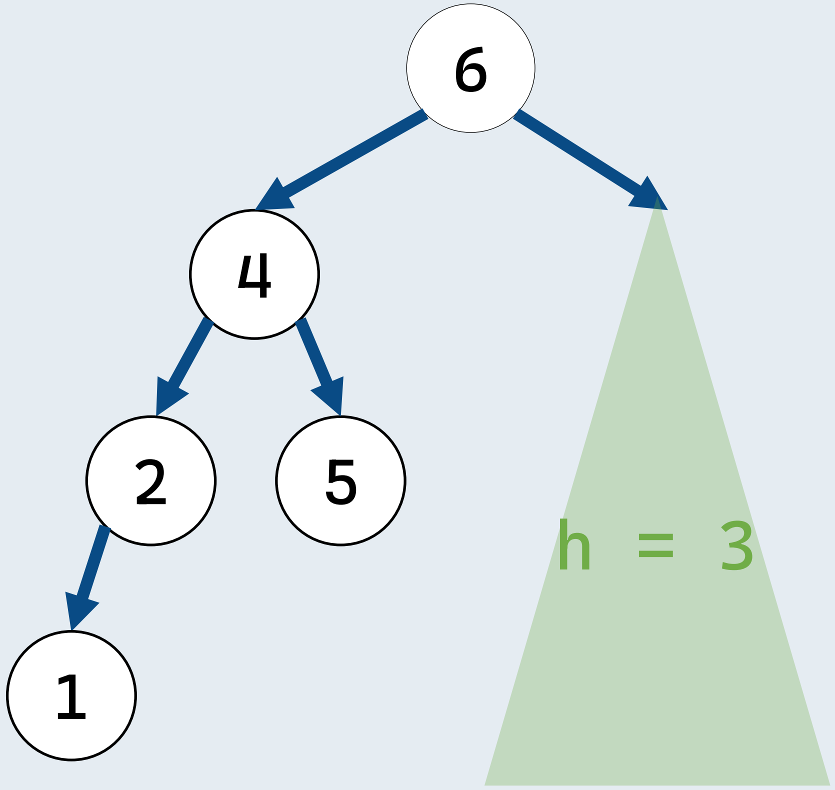
Rebalancing

- When tree is unbalanced (has a node where the height of left subtree and right subtree differ more than 1), needs rebalancing!
- **Double rotations**:



Problem 1.a

True or false: *Given any AVL tree of height 4, deleting any vertex in the tree will not result in more than 1 rebalancing operation (not rotation but rebalancing operations!).*



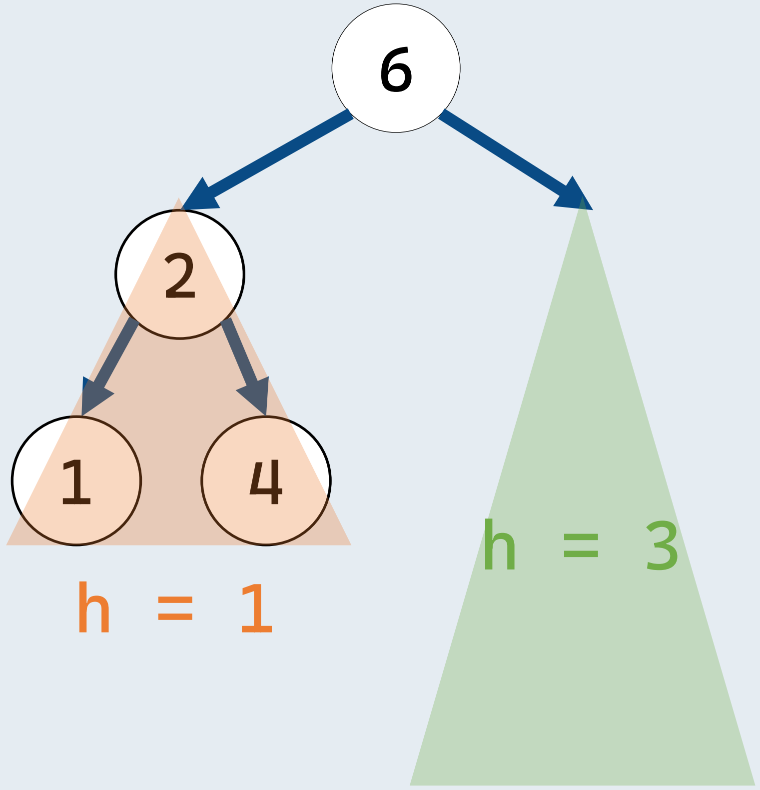
False. We can find a counter example:

if we delete 5,

- Need rebalancing in the left subtree,

Problem 1.a

True or false: *Given any AVL tree of height 4, deleting any vertex in the tree will not result in more than 1 rebalancing operation (not rotation but rebalancing operations!).*



False. We can find a counter example:

if we delete 5,

- Need rebalancing in the left subtree,
- Need another to balance the left and right subtree.

Problem 1.b

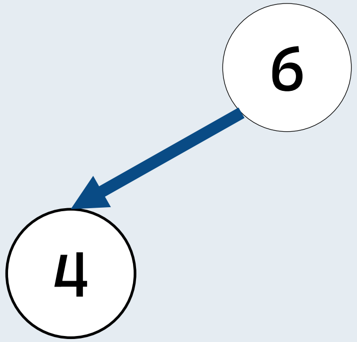
True or false: *The minimum number of vertices in an AVL tree of height 5 is 21.*

6

Height of AVL tree	Minimum no. of nodes
0	1
1	
2	
3	
4	
5	

Problem 1.b

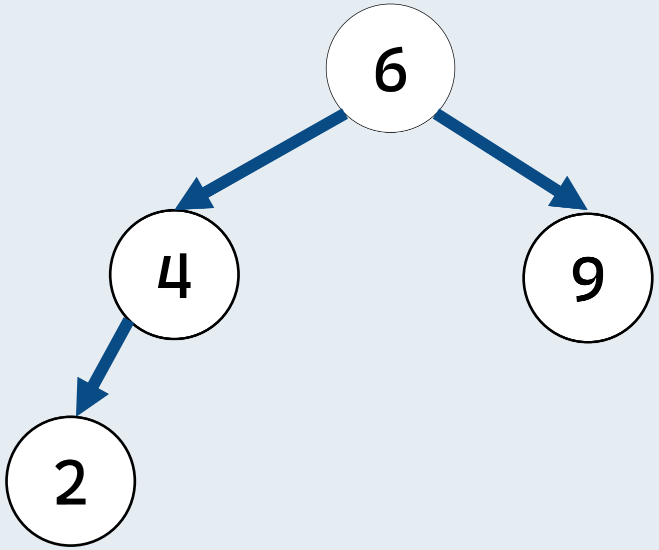
True or false: *The minimum number of vertices in an AVL tree of height 5 is 21.*



Height of AVL tree	Minimum no. of nodes
0	1
1	2
2	
3	
4	
5	

Problem 1.b

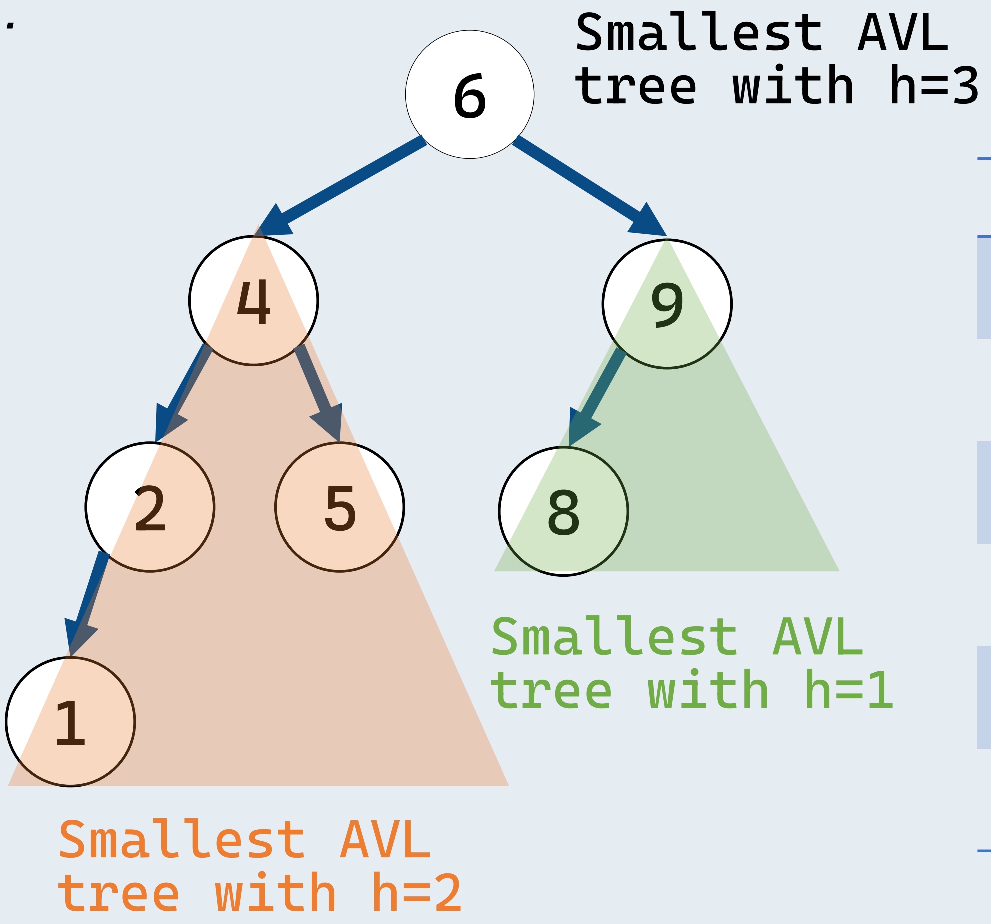
True or false: *The minimum number of vertices in an AVL tree of height 5 is 21.*



Height of AVL tree	Minimum no. of nodes
0	1
1	2
2	4
3	
4	
5	

Problem 1.b

True or false: *The minimum number of vertices in an AVL tree of height 5 is 21.*

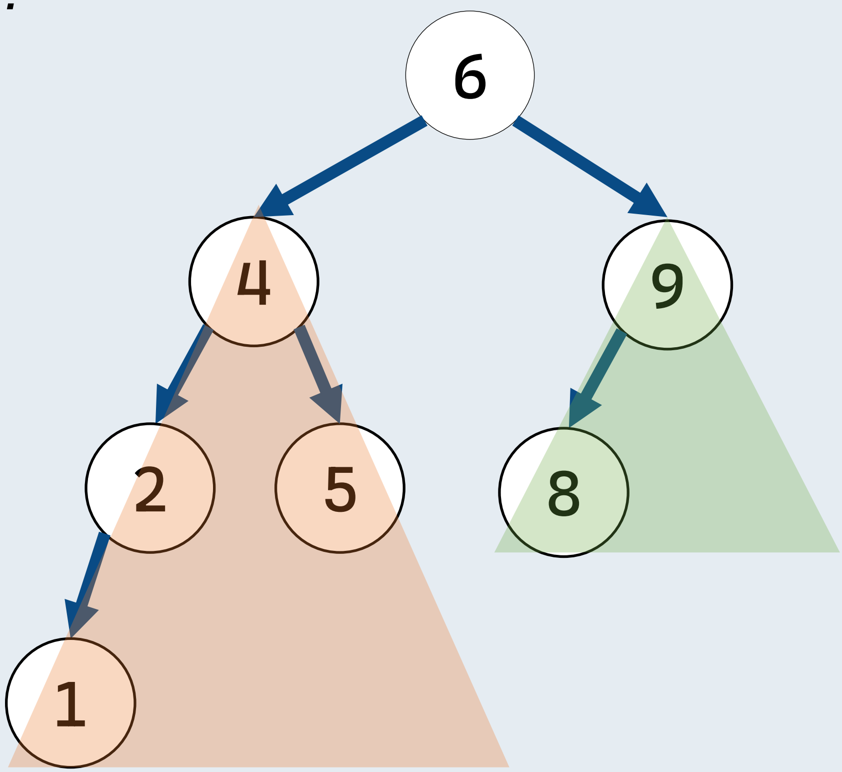


Height of AVL tree	Minimum no. of nodes
0	1
1	2
2	4
3	7
4	
5	

$$N_i = N_{i-1} + N_{i-2} + 1$$

Problem 1.b

True or false: *The minimum number of vertices in an AVL tree of height 5 is 21.*



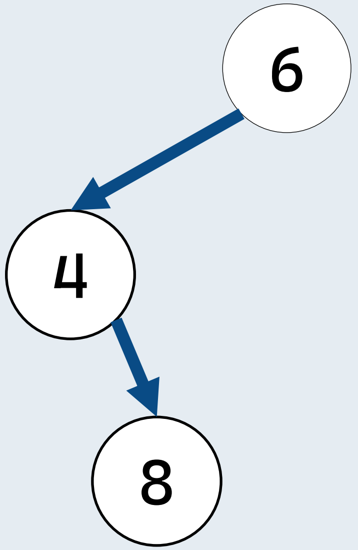
False. The minimum is 20.

Height of AVL tree	Minimum no. of nodes
0	1
1	2
2	4
3	7
4	12
5	**20**

$$N_i = N_{i-1} + N_{i-2} + 1$$

Problem 1.c

True or false: *In a tree, if for every vertex* `x` *that is not a leaf,* `x.left.key < x.key` *if x has a left child and* `x.key < x.right.key` *if* `x` *has a right child, the tree is a BST.*



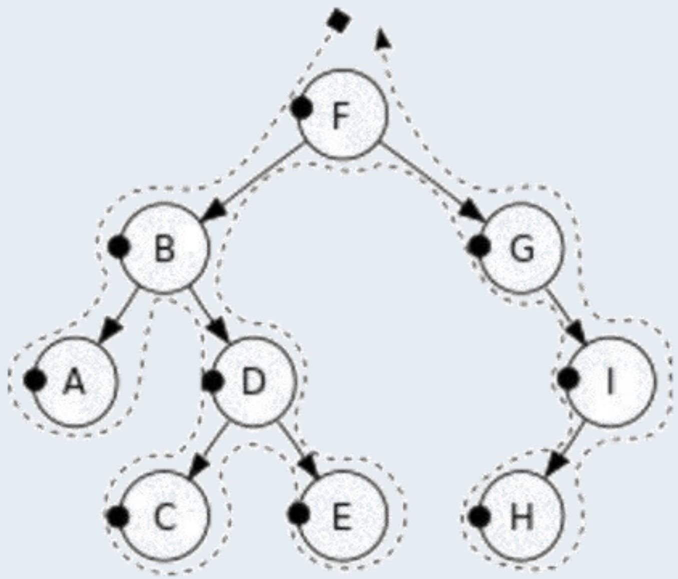
False. We can find a counter example:

The tree on the left is not a BST because we require **ALL** nodes in left subtree are smaller than root.

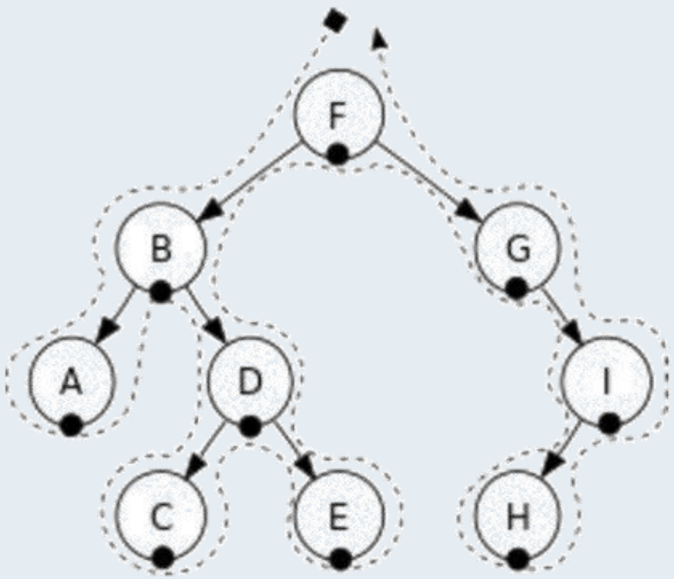
Tree Traversal

How to properly do traversal?

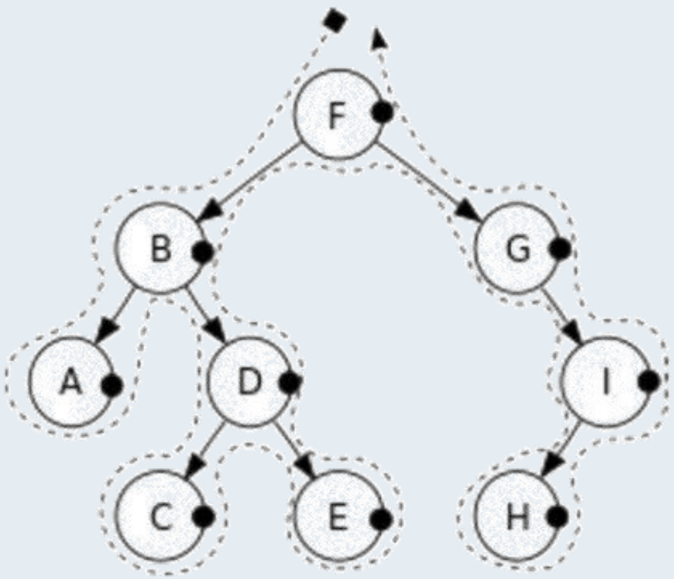
Three ways of tree traversal




```
pre-order(node)
    if node is null: return
    visit(node)
    pre-order(node.left)
    pre-order(node.right)
```




```
in-order(node)
    if node is null: return
    in-order(node.left)
    visit(node)
    in-order(node.right)
```




```
post-order(node)
    if node is null: return
    post-order(node.left)
    post-order(node.right)
    visit(node)
```

Problem 2

Algorithm 1 In-Order Traversal

procedure InOrderTraversal(T) ▷ bBST T is supplied as input
    $currentNodeValue \leftarrow T.findMin()$
    while $currentNodeValue \neq -1$ **do**
        output $currentNodeValue$
        $currentNodeValue \leftarrow T.successor(currentNodeValue)$
    end while ▷ $successor$ returns -1 if there is no successor
end procedure

Start from smallest node in BST...

What is the running time of Algorithm 1?

Problem 2

Algorithm 1 In-Order Traversal

procedure InOrderTraversal(T) ▷ bBST T is supplied as input
    $currentNodeValue \leftarrow T.findMin()$
    while $currentNodeValue \neq -1$ **do**
        output $currentNodeValue$ Visit current node...
        $currentNodeValue \leftarrow T.successor(currentNodeValue)$
    end while ▷ $successor$ returns -1 if there is no successor
end procedure

What is the running time of Algorithm 1?

Problem 2

Algorithm 1 In-Order Traversal

```
procedure InOrderTraversal(T)                              ▷ bBST T is supplied as input
    currentNodeValue ← T.findMin()
    while currentNodeValue ≠ −1 do
        output currentNodeValue
        currentNodeValue ← T.successor(currentNodeValue)
    end while                                               ▷ successor returns -1 if there is no successor
end procedure
```

Go to next smallest node, until all nodes are visited.

What is the running time of Algorithm 1?

Problem 2

Algorithm 1 In-Order Traversal

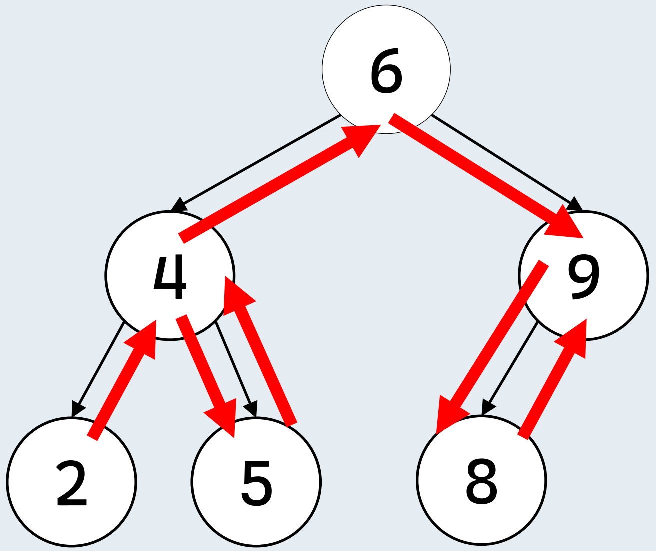
procedure InOrderTraversal(T) ▷ bBST T is supplied as input
  $currentNodeValue \leftarrow T.findMin()$ $O(\log n)$ time.
  while $currentNodeValue \neq -1$ **do**
    output $currentNodeValue$
    $currentNodeValue \leftarrow T.successor(currentNodeValue)$ $O(\log n)$ time, the `while` loop runs $O(n)$ times.
  end while ▷ *successor* returns -1 if there is no successor
end procedure

What is the running time of Algorithm 1?

In total $O(n \log n)$ time.

Problem 2

Propose modifications to the `successor` *function such that Algorithm 1 runs in $O(n)$ time.*



The effect is exactly the same as in-order traversal...

But in-order traversal runs in $O(n)$ time. **What's the difference?**

Problem 2

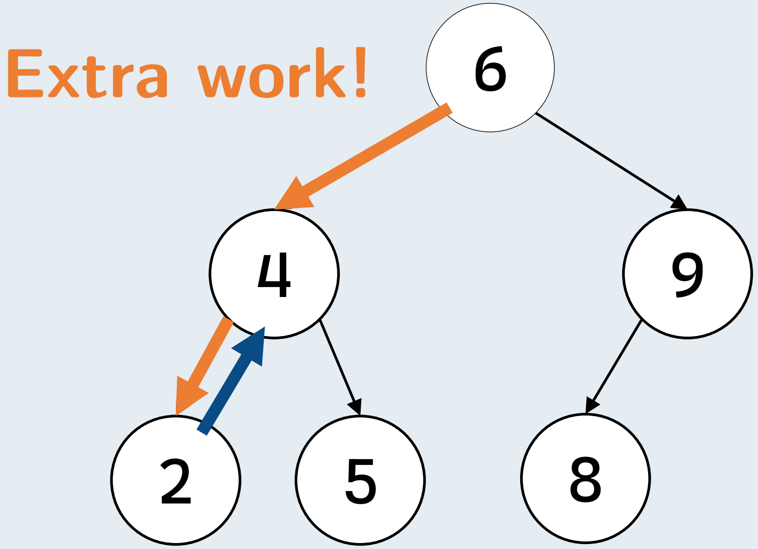
Algorithm 1 In-Order Traversal

procedure InOrderTraversal(T) ▷ bBST T is supplied as input
  $currentNodeValue \leftarrow T.findMin()$
  while $currentNodeValue \neq -1$ **do**
    output $currentNodeValue$
    $currentNodeValue \leftarrow T.successor(currentNodeValue)$
  end while ▷ $successor$ returns -1 if there is no successor
end procedure

We need to first find the node containing `currentNodeValue` from the root node, then go to its successor!

Problem 2

If we call **`successor(2)`**...



1. Find the node containing 2...
2. Starting from the node, find its successor.

Problem 2

Algorithm 1 In-Order Traversal

procedure InOrderTraversal(T) ▷ bBST T is supplied as input
  $currentNodeValue \leftarrow T.findMin()$
  while $currentNodeValue \neq -1$ **do**
    output $currentNodeValue$
    $currentNodeValue \leftarrow T.successor(currentNodeValue)$
  end while ▷ $successor$ returns -1 if there is no successor
end procedure

Idea: We feed a reference of `currentNode` to the successor function instead, instead of its value.

Each node is now visited at most $O(1)$ times ... $O(n)$ time in total!

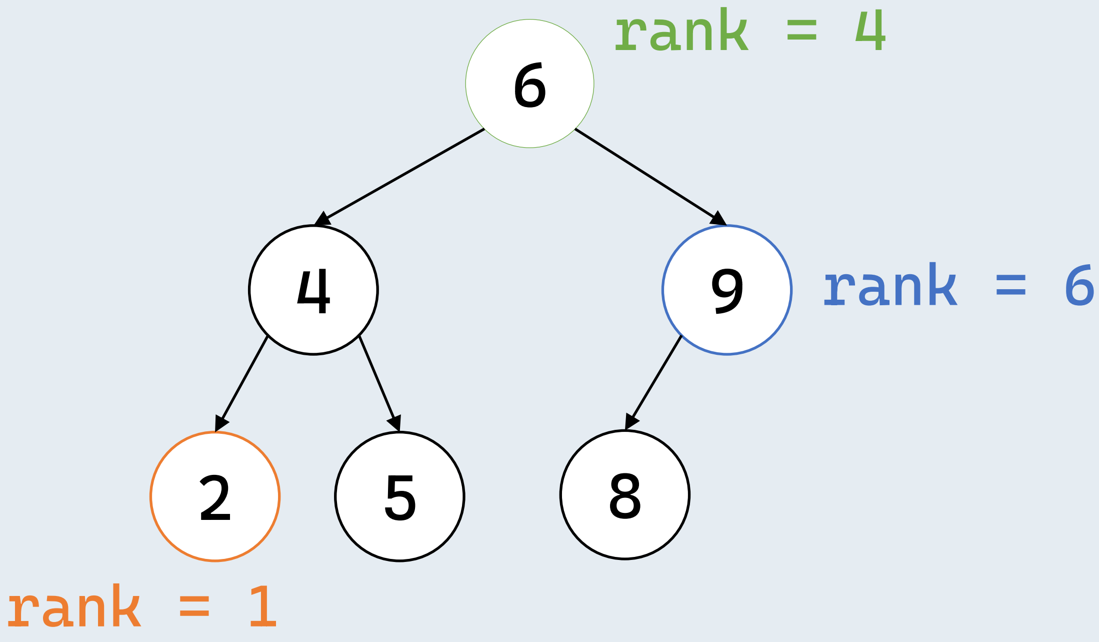
Applications

How to use BST's ability of comparing keys?

Problem 3

*A node x has **rank** k in a BST if there are $k-1$ nodes that are smaller than x in the BST.*

Goal: find the rank of a node x in $O(h)$ time.

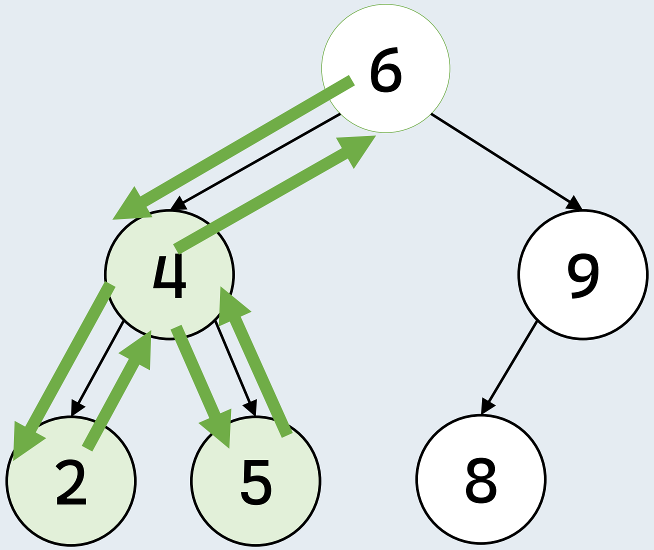


Problem 3

Trivial Solution: Find all nodes that are smaller than the node x.

We can use in-order traversal, count the visited nodes, until we visit x!

Time needed: $O(n)$ time.



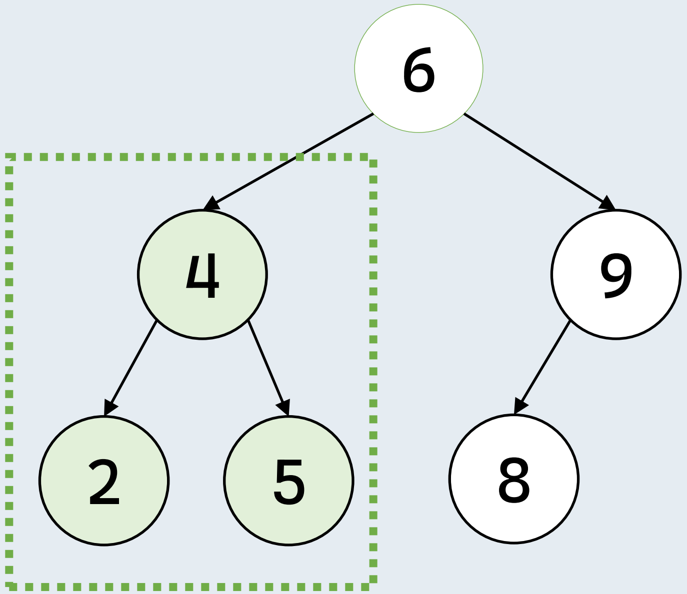
Where are the nodes smaller than the root **6** located?

Problem 3

The nodes that are smaller than root are all in left subtree.

Idea: just return the size of the left subtree!

But counting the size is still $O(n)$...

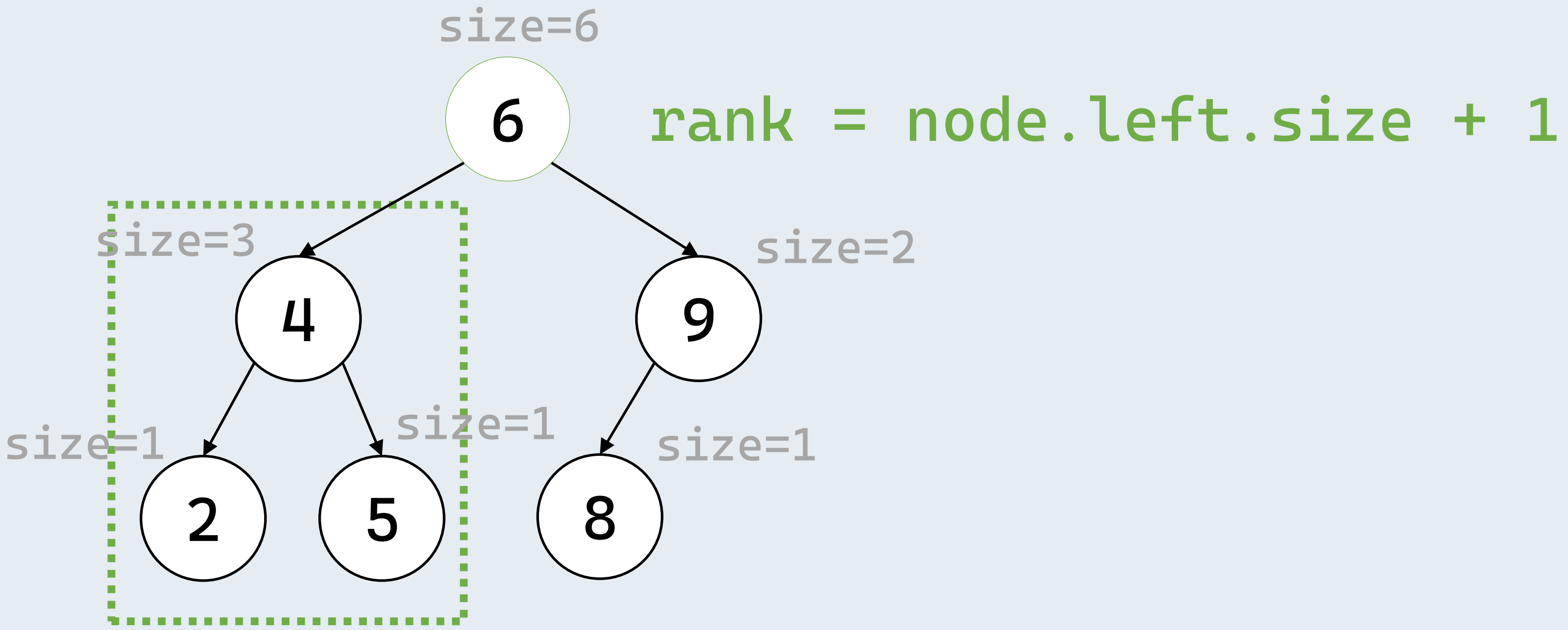


Problem 3

The nodes that are smaller than 6 are all in left subtree.

Idea: just return the size of the left subtree!

Count and store the `size` of subtree rooted at the node beforehand!

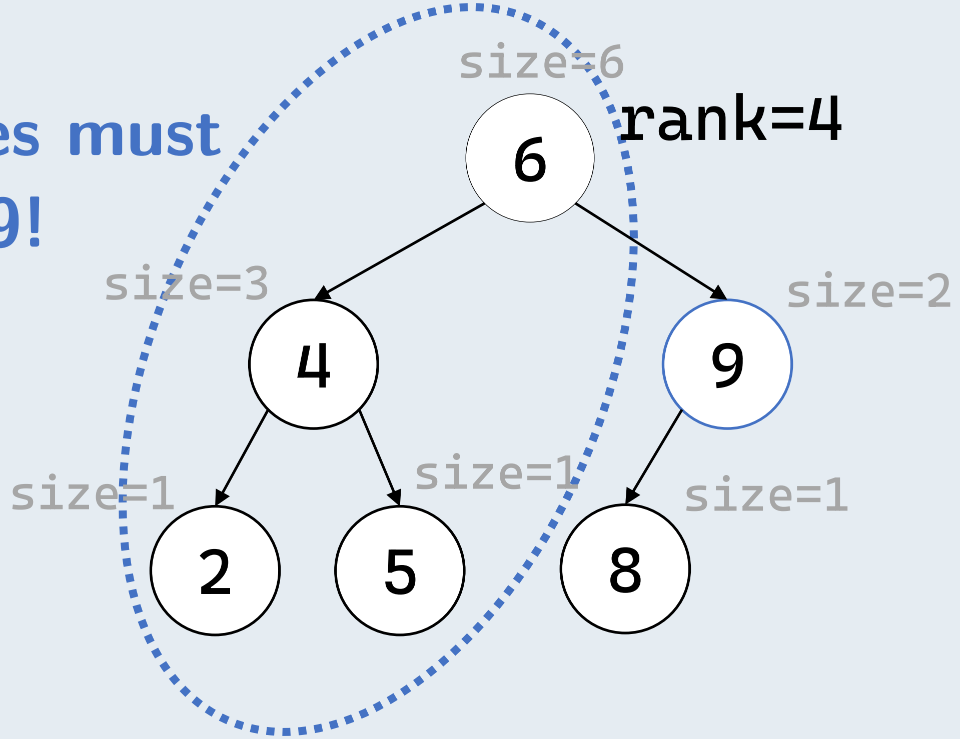


Problem 3

What if we want to find the rank of nodes other than root, e.g. 9?

1. Visit the root, and the root $6 < 9$.

All of these nodes must be smaller than 9!



Intuition

- We know that 6 is in rank 4 in sorted array … And all numbers before 6 are smaller than 6.

4	2	5	6	9	8

`rank=4`

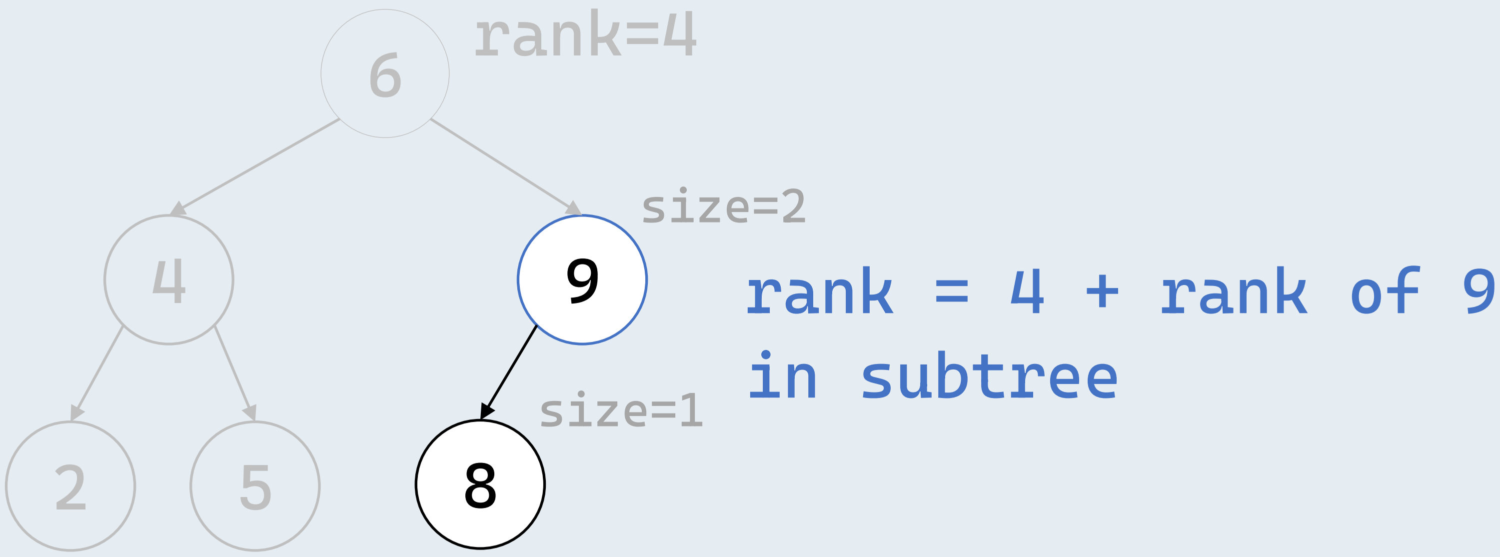
- rank of 9 in the whole array = rank of 6 + rank of 9 in right sub-array.

4	2	5	6	9	8

Problem 3

What if we want to find the rank of node other than root, e.g. 9?

1. Visit the root, and the root $6 < 9$.
2. Ignore the left subtree, and find the rank of 9 in right subtree.



Problem 3

Algorithm 2 BST Rank

```
function RANK(node, v)
    if node.key = v then
        return node.left.size + 1
    else if node.key > v then
        return RANK(node.left, v)
    else
        return node.left.size + 1 + RANK(node.right, v)
    end if
end function
```

rank of root node.

Recurrence relation:

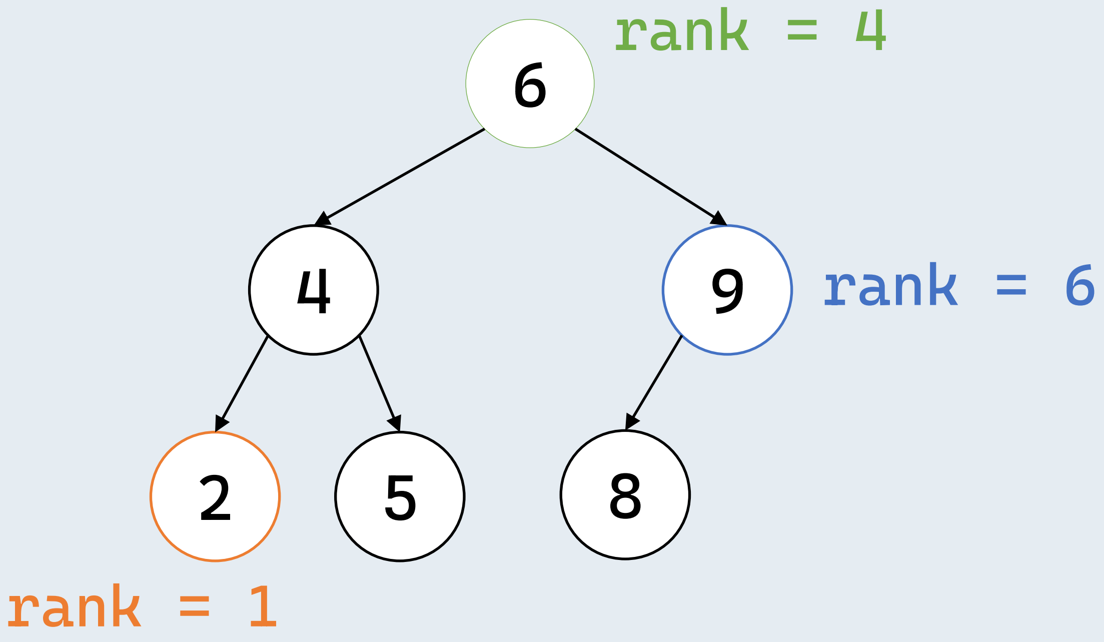
$$T(h) = T(h-1) + 1$$

In total $O(h)$ time.

Problem 3

*A node x has **rank** k in a BST if there are $k-1$ nodes that are smaller than x in the BST.*

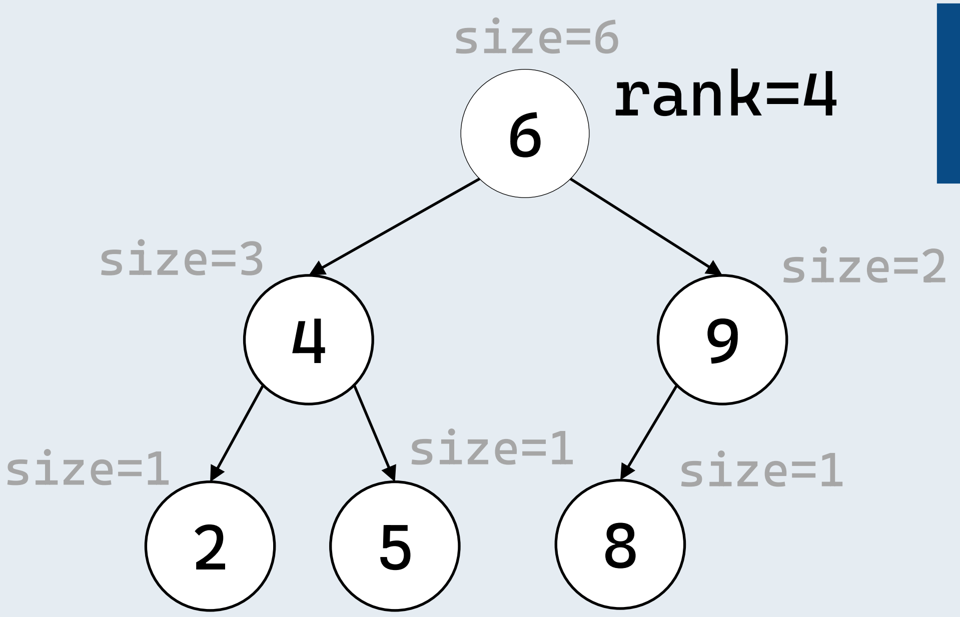
Goal: Find the node with rank k in $O(h)$ time.



Problem 3

Example: Find the node with rank $= 5$.

5 $>$ rank of root node $=$ 4.



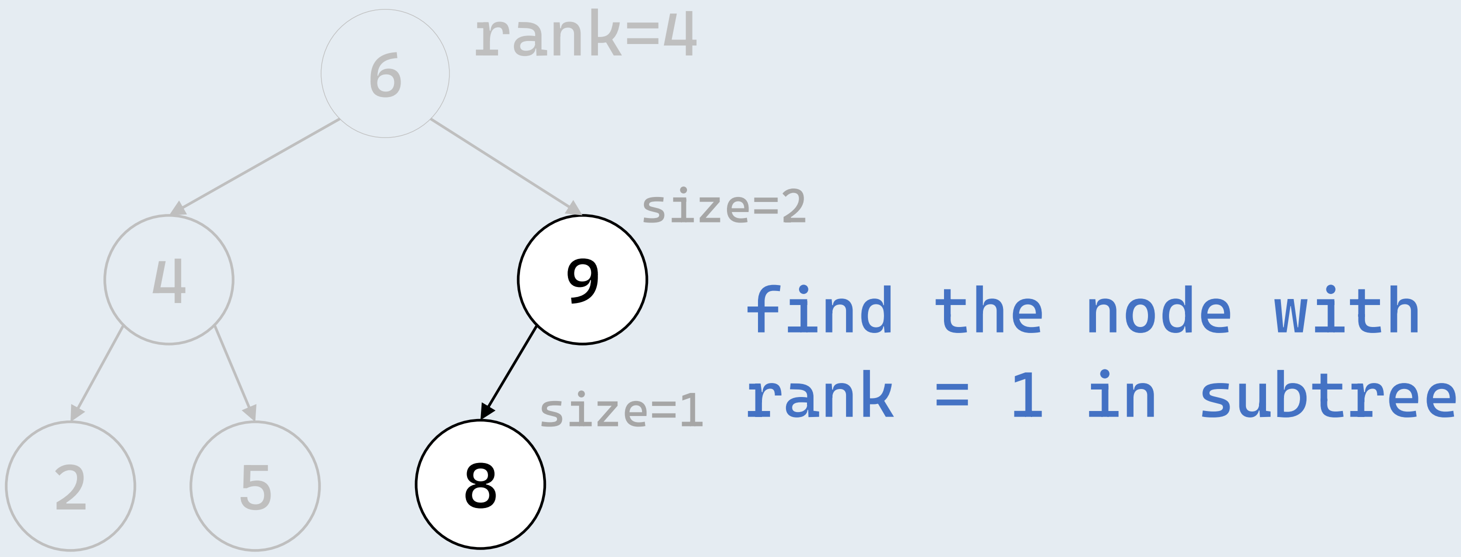
Do we need to check every node for their rank?

Problem 3

Example: Find the node with rank $= 5$.

5 $>$ rank of root node $=$ 4.

Idea: we only need to check the right subtree for the node!



Problem 3

Algorithm 3 BST Select

```
function SELECT(node, k)
    q ← node.left.size
    if q + 1 = k then
        return node.key
    else if q + 1 > k then
        return SELECT(node.left, k)
    else
        return SELECT(node.right, k − q − 1)
    end if
end function
```

This is very similar to `QuickSelect`!

… Except for the `partition` is already done by BST!

** It is very common to use recursions in tree-related algorithms! Get used to it ☺*

Problem 4

- n servers in a line, either enabled or disabled.
- Server i can send data to server j if all servers between i and j are enabled.

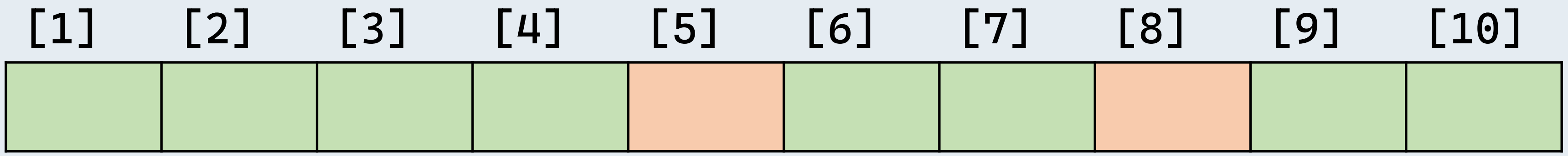
Goal: support the operations:

1. **`enable(i)`**: Enable the i-th server.
2. **`disable(i)`**: disable the i-th server.
3. **`send(i, j)`**: return true if we can send data from server i to j.

Problem 4

- n servers in a line, either enabled or disabled.
- Server i can send data to server j if all servers between i and j are enabled.

Example: Servers 5 and 8 are disabled.



`send(3, 10)` returns `false`.

Problem 4

Trivial Solution:

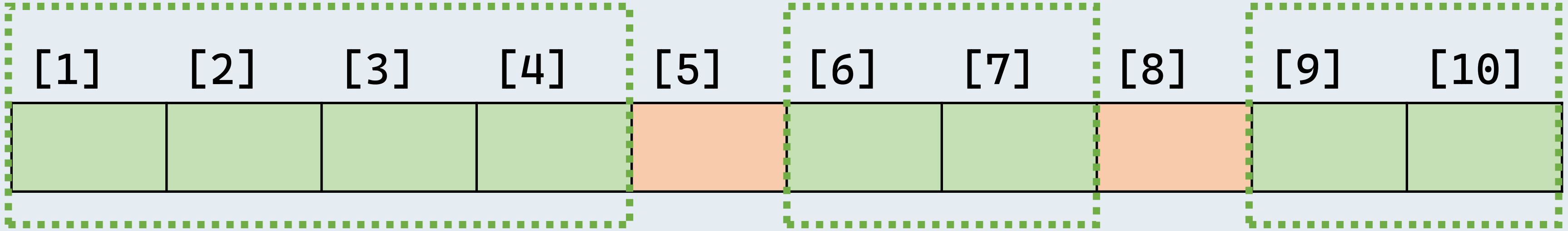
- Store everything in an array.
- Start from `i` and see if we encounter a disabled server, until we reach `j`.
- `enable` and `disable` are $O(1)$ time, `send` takes $O(n)$ time.

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
T	T	T	T	F	T	T	F	T	T

Problem 4

Another Solution:

- Store everything in UFDS.
- Group consecutive enabled servers in a set.
- `send` simply checks if `i` and `j` are in same set.
- `enable` and `send` are $O(\alpha(n))$ time, `disable` takes $O(n)$ time.



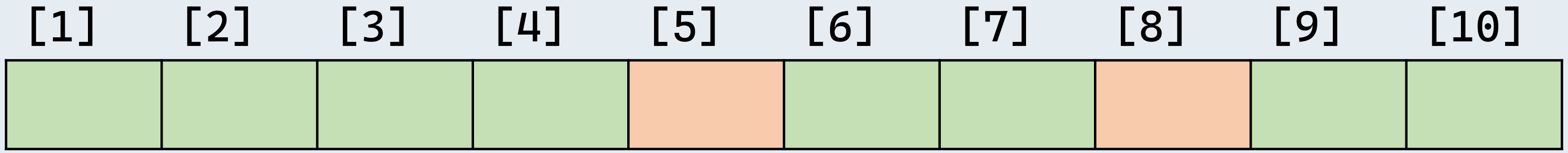
Problem 4

Observation:

Cannot send message from **`i`** to **`j`** if the first disabled server after **`i`** is **`< j`**.

e.g. **`send(3, 10)`** is false because <u>the first disabled server after 3</u> is 5.

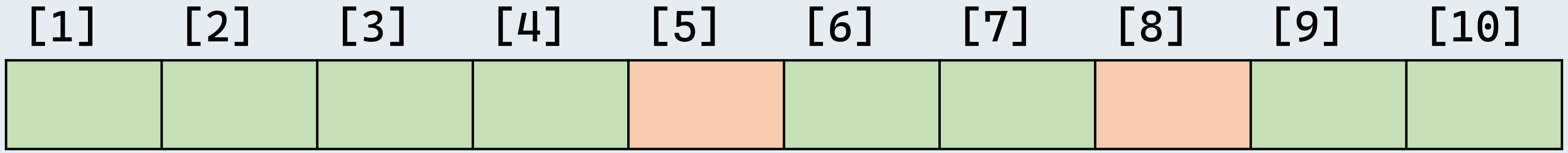
find the **successor** of 3!



Problem 4

Idea:

- Store all the **disabled servers** in an AVL tree.
- **`send(i, j)`** checks existence of **`i`**, and find the successor of **`i`** and check if it is **`> j`**.
- All 3 operations takes $O(\log n)$ time.



Problem 4

Algorithm 4 Solution to Problem 4

$T \leftarrow$ bBST, initially empty
procedure ENABLE(i)
    $T.delete(i)$
end procedure
procedure DISABLE(i)
    $T.insert(i)$
end procedure
function SEND(i, j)
    if i is in T **or** $T.successor(i) \leq j$ **then**
        return false
    else
        return true
    end if
end function

Note:

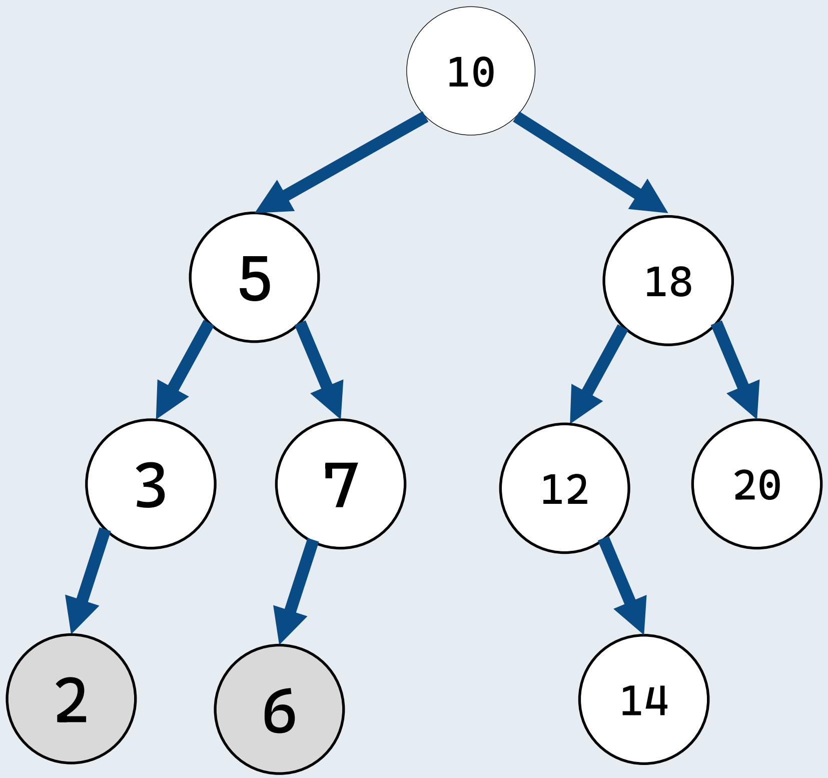
This **`successor`** is different from the one in the lecture: **`i`** may not be a node in the BST!

How to implement this `successor` function?

** See the appendix for hints!*

Problem 5

The Lowest Common Ancestor (LCA) of two nodes $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *in a BST is the node furthest from the root that is an ancestor of both* $\boldsymbol{a}$ *and* $\boldsymbol{b}$.

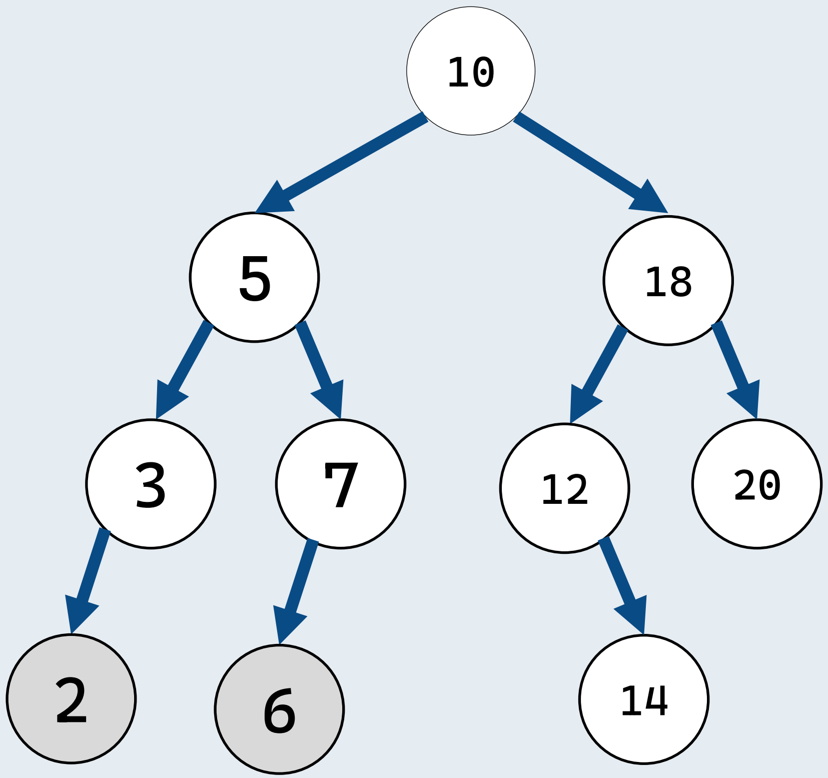


Example:

LCA of 2 and 6 is 5.

Problem 5

The Lowest Common Ancestor (LCA) of two nodes a *and* b *in a BST is the node furthest from the root that is an ancestor of both* a *and* b.



Do we need to store the path?

Idea:

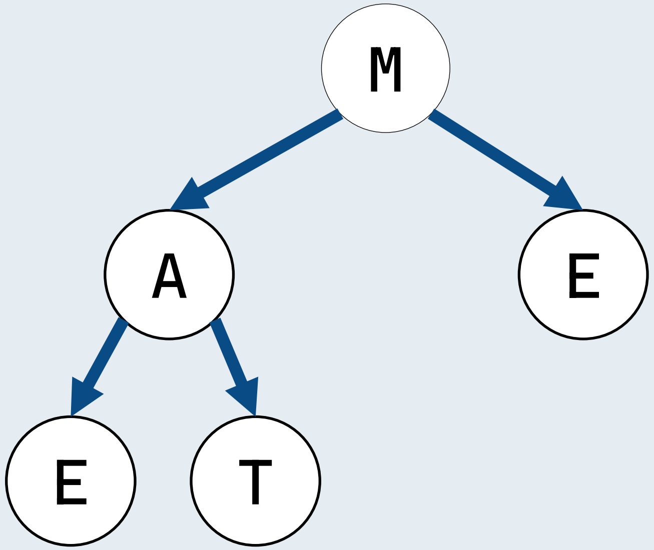
- We can first find the two nodes:
  - For node 2, we go through `10 -> 5 -> 3 -> 2`.
  - For node 6, we go through `10 -> 5 -> 7 -> 6`.
- **Just check the common path!**

Problem 6

A set of N strings of length L.

Goal: implement `gotPrefix(k)`: return true if string `k` is a prefix of some string in the set.

Example: Set is `{ME, MAE, MAT}`, `gotPrefix("MA")` returns `true`.



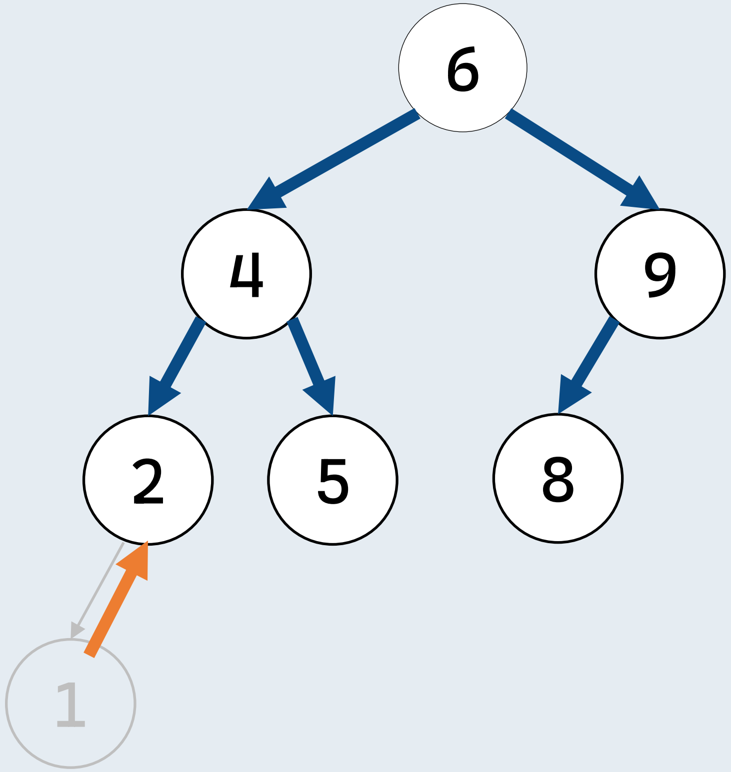
Idea: Simply use a trie!

- **preprocessing**: each insertion takes $O(L)$ time.
- **query**: check if there is a path matching k. Takes $O(k.\text{length})$ time.

Appendix

`successor(value)`

Problem: *How do we implement successor(value), where the value might not be among the nodes in BST?*



Example: `successor(1)`.

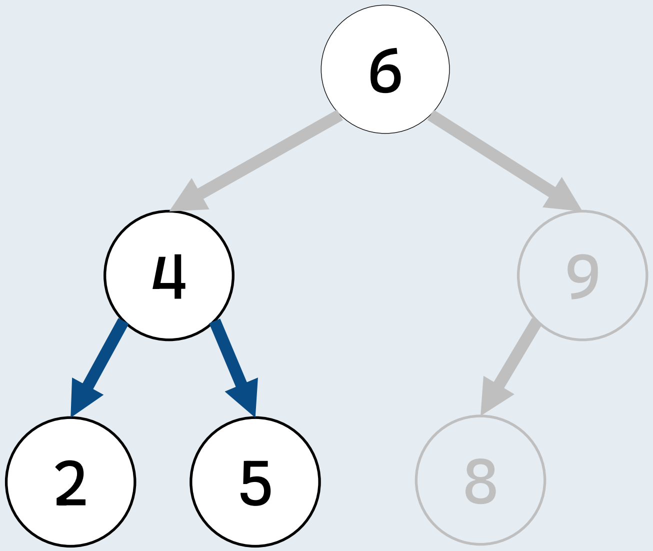
Trivial Solution:

- Insert a node 1,
- Find its successor,
- Delete the node 1.

Can we avoid creating the node?

`successor(value)`

Problem: *How do we implement successor(value), where the value might not be among the nodes in BST?*



Idea: Can also do it recursively!

We have the root $6 > 1$.

If 6 is not the successor, it must be in left subtree!

The successor must be: min{6, successor of 1 in left subtree}.

End of File

Thank you very much for your attention :-)