CS2040S Tutorial 4

Admin matters

- Attendance taking
- Check in and PS4 discussion

PS4

- Some of you forgot to check for edge cases, such as when enumerateNodes is called on a node with no children
    - Be more careful!
- Start work early to prevent running out of time
- If your code has issues, please try to fix them because Problem Set 5 is a continuation of Problem Set 4
    - If you are unable to do so/want me to verify correctness, feel free to approach me

Recap (adapted from Christian's Slides)

- Quickselect
- Trees

Quickselect

Quickselect

- An algorithm to select the k^{th} smallest element in unsorted array
- Derives a similar idea from Quicksort
- Observation: When you finish partitioning, the pivot is already at the correct place!
- But instead of recursing on both sides, recurse on one side!

select(4)

Initialisation. Want to select 4th smallest elem.
Use 1-indexing because easier to think about

	10	14	35	32	40	22	7	6	8	1	0	5
Index:	1	2	3	4	5	6	7	8	9	10	11	12

select(4)

Select a random pivot and partition

	10	14	35	32	40	22	7	6	8	1	0	5
Index:	1	2	3	4	5	6	7	8	9	10	11	12

select(4)

Select a random pivot and partition

	10	14	7	0	5	1	8	6	22	35	32	40
Index:	1	2	3	4	5	6	7	8	9	10	11	12

select(4)

We know that our pivot is actually the 9th smallest element (i.e. we are too high up!)

	10	14	7	0	5	1	8	6	22	35	32	40
Index:	1	2	3	4	5	6	7	8	9	10	11	12

select(4)

So we recurse on the left side instead. All while choosing a new pivot

	10	14	7	0	5	1	8	6	22	35	32	40
Index:	1	2	3	4	5	6	7	8	9	10	11	12

select(4)

Partitioning...

	1	0	5	7	6	14	10	8	22	35	32	40
Index:	1	2	3	4	5	6	7	8	9	10	11	12

select(4)

Our pivot 5 was actually the 3rd smallest element.
Too low!

	1	0	5	7	6	14	10	8	22	35	32	40
Index:	1	2	3	4	5	6	7	8	9	10	11	12

select(4)

Recurse to the right, and so on!

	1	0	5	7	6	14	10	8	22	35	32	40
Index:	1	2	3	4	5	6	7	8	9	10	11	12

It is more important to understand the idea behind quickselect rather than being caught up in the indexing issues.

Quickselect Analysis

For an array of size n...

Recurrence Relation:

Time Complexity:

Quickselect Analysis

For an array of size n...

Recurrence Relation: $T(n) = T(n / 2) + O(n)$

Time Complexity: $O(n)$

Ordered Dictionary ADT

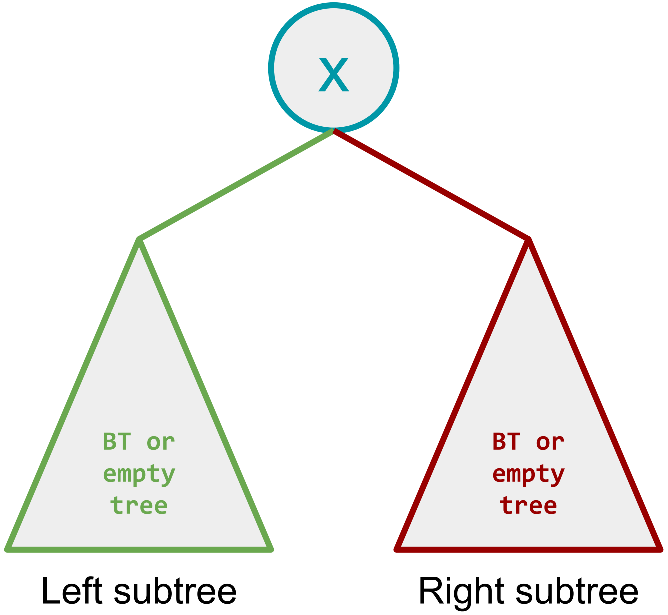
It should guarantee these operations:

Ordered Dictionary ADT

It should guarantee these operations:

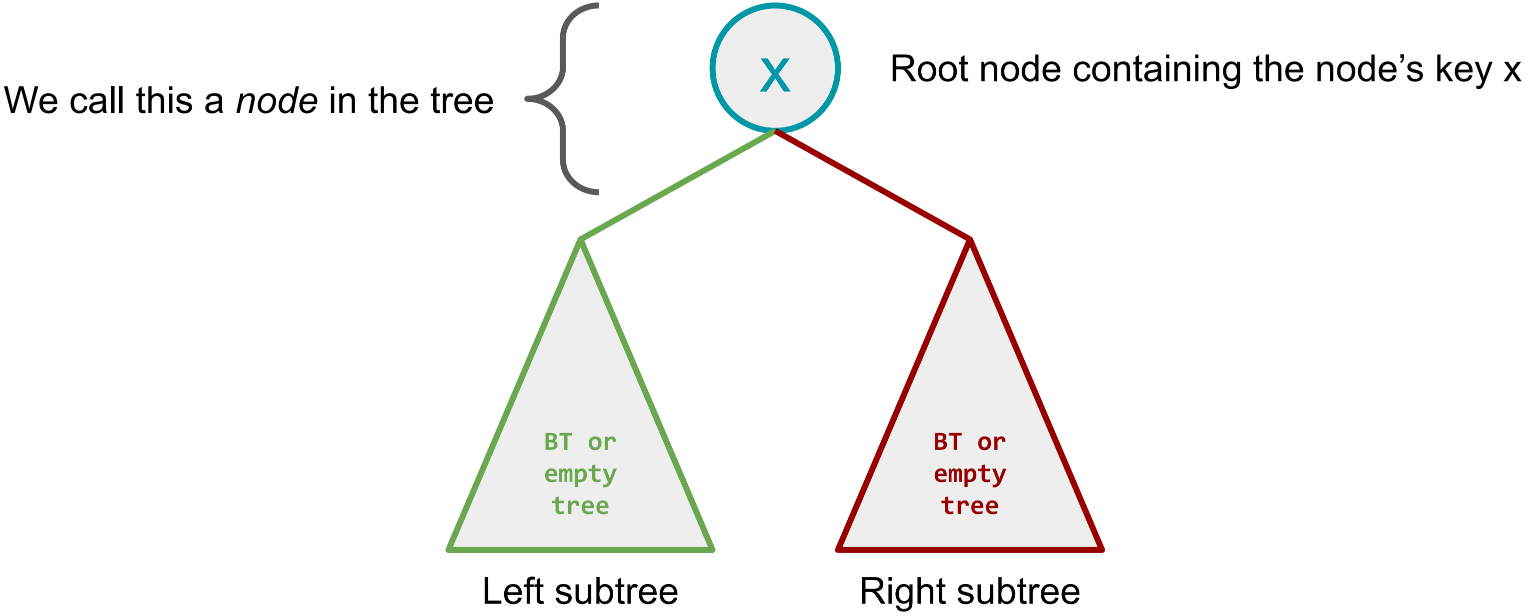
- `insert(key, value)`
- `search(key)`
- `delete(key)`
- `contains(key)`
- `successor(key)`
- `predecessor(key)`
- `size()`

Binary trees



Binary trees

Conceptually, we often visually represent a binary tree in the following form



Realize that this diagram applies to EVERY NODE in the BT?
I.E. every node in the BT is itself a BT!

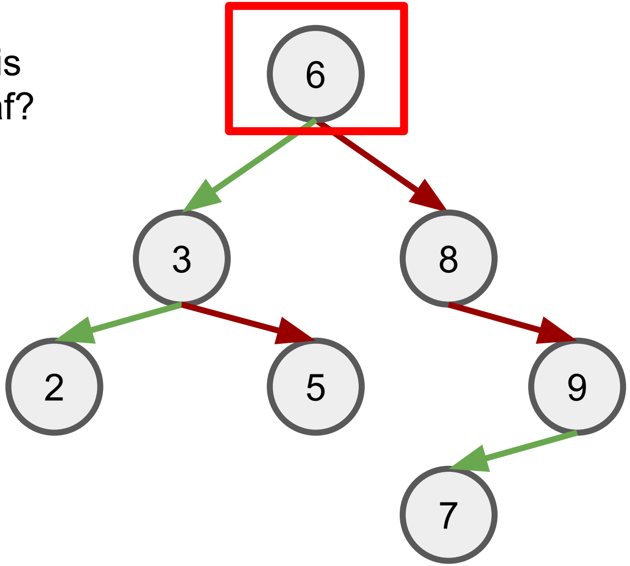
Height of a node

Definition: Number of **edges** on the *longest path* from from node to leaf

Height of a node

Definition: Number of **edges** on the *longest path* from from node to leaf

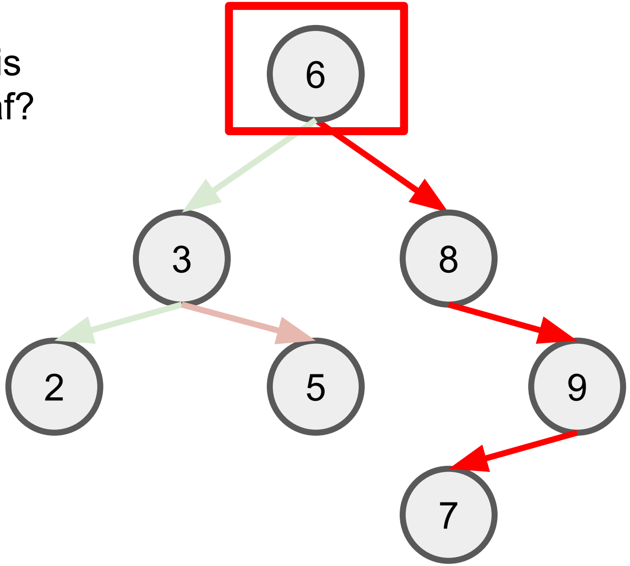
For this node, which is
the longest path to leaf?



Height of a node

Definition: Number of **edges** on the *longest path* from from node to leaf

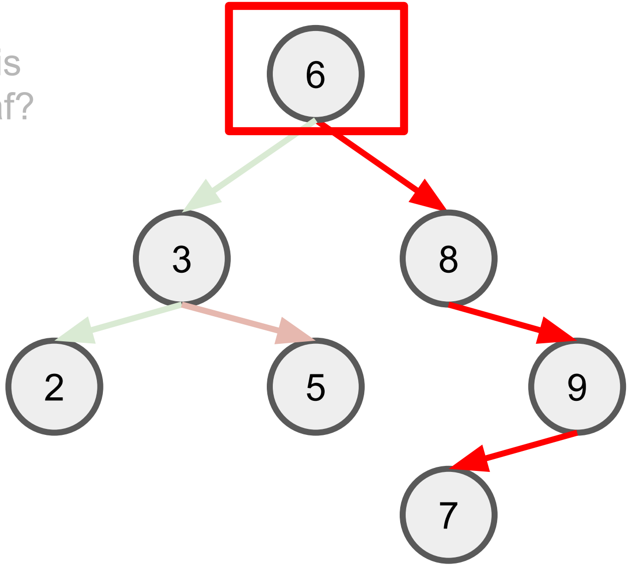
For this node, which is
the longest path to leaf?



Height of a node

Definition: Number of **edges** on the *longest path* from node to leaf

For this node, which is the longest path to leaf?

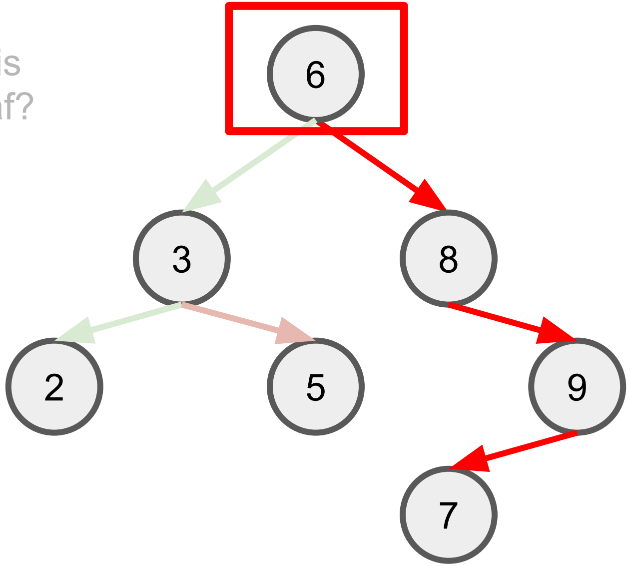


How many edges are there in that path?

Height of a node

Definition: Number of **edges** on the *longest path* from from node to leaf

For this node, which is the longest path to leaf?



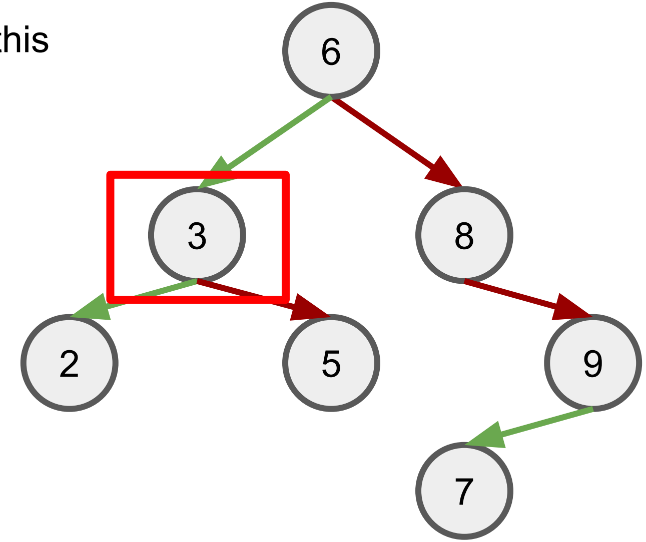
How many edges are there in that path?
3

Therefore this node has height 3

Height of a node

Definition: Number of **edges** on the *longest path* from from node to leaf

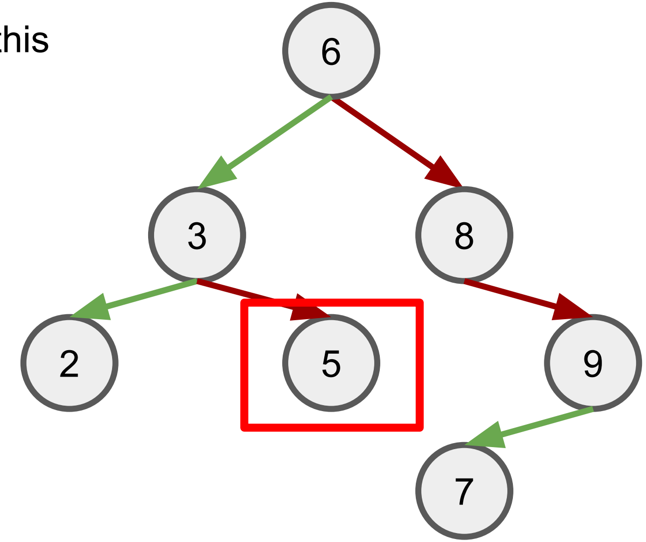
What's the height of this node?



Height of a node

Definition: Number of **edges** on the *longest path* from from node to leaf

What's the height of this node?

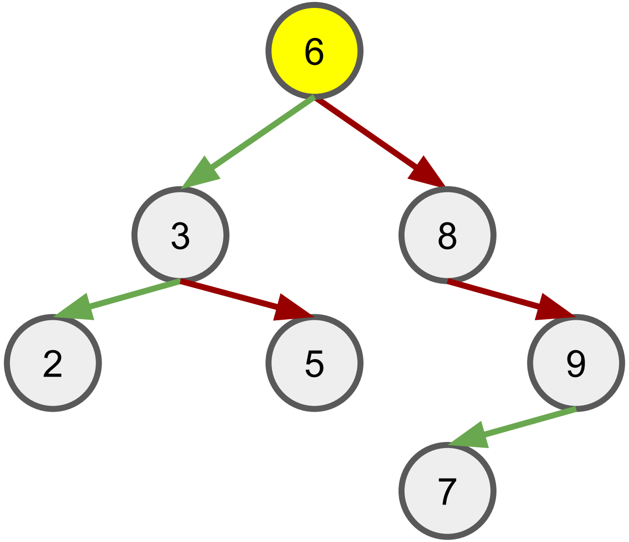


Depth of a node

Definition: Number of **edges** to the root

Depth of a node

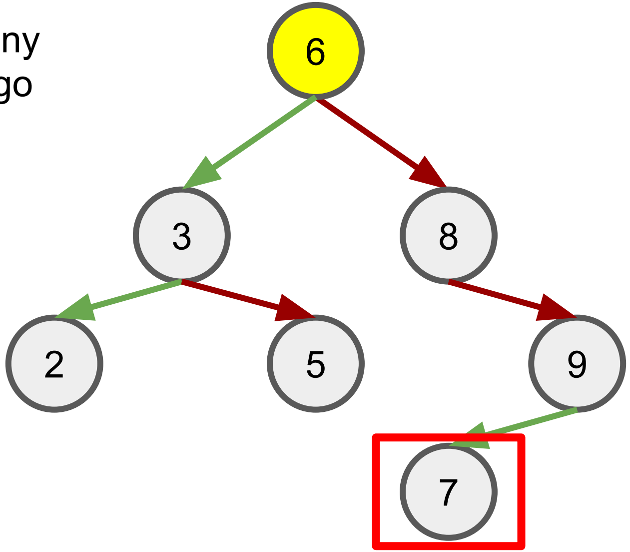
Definition: Number of **edges** to the root



Depth of a node

Definition: Number of **edges** to the root

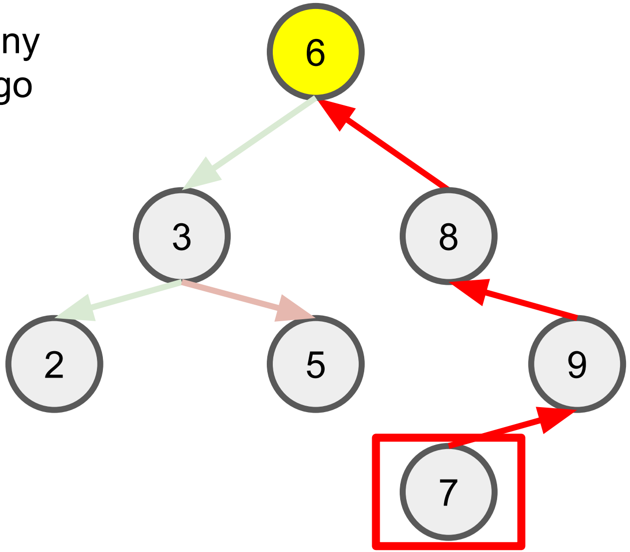
For this node, how many edges are needed to go up to the root?



Depth of a node

Definition: Number of **edges** to the root

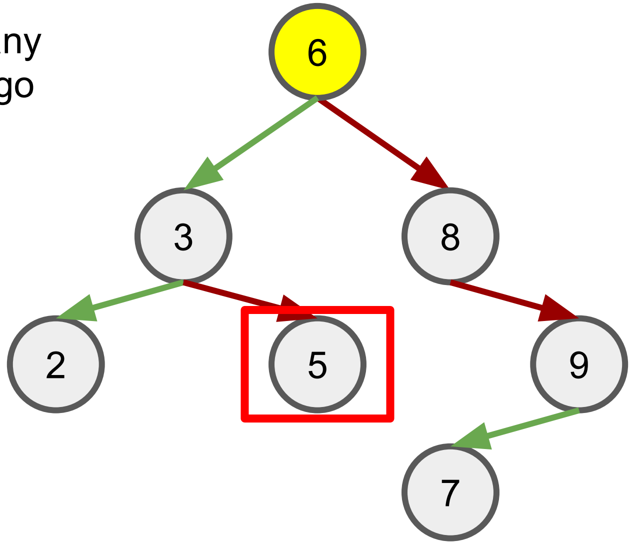
For this node, how many edges are needed to go up to the root?
3



Depth of a node

Definition: Number of **edges** to the root

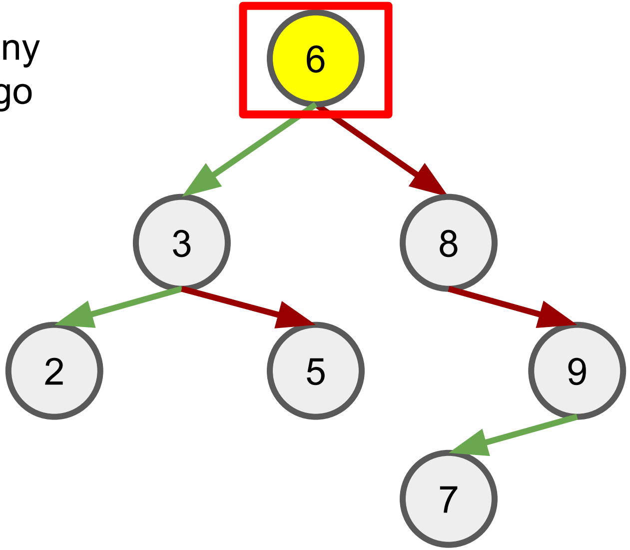
For this node, how many edges are needed to go up to the root?



Depth of a node

Definition: Number of **edges** to the root

For this node, how many edges are needed to go up to the root?



Binary Search Trees (BST)

Binary Search Trees (BST)

It is a binary tree with the following properties:

- A node’s left **subtree** contains nodes strictly less than the node’s key
- A node’s right **subtree** contains nodes strictly greater than the node’s key
- The left and right subtrees are binary trees
- All keys belong to a total order (no two different keys can be considered equal)

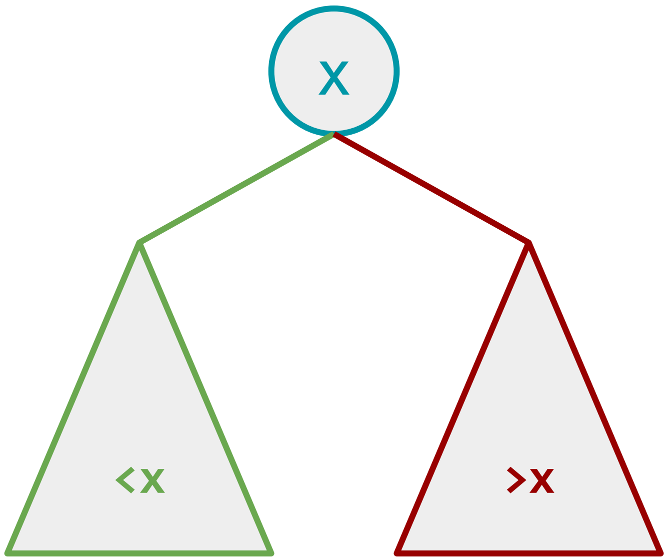
Binary Search Trees (BST)

It is a binary tree with the following properties:

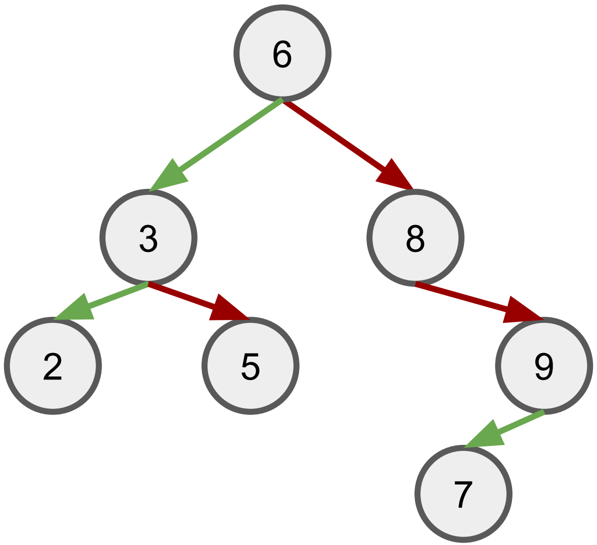
- A node’s left **subtree** contains nodes strictly less than the node’s key
- A node’s right **subtree** contains nodes strictly greater than the node’s key
- The left and right subtrees are binary trees
- All keys belong to a total order (no two different keys can be considered equal)

Common mistake: Thinking that only the direct left/right child have to be less/greater. It is ALL the nodes in the left/right SUBTREE

Binary Search Trees (BST)



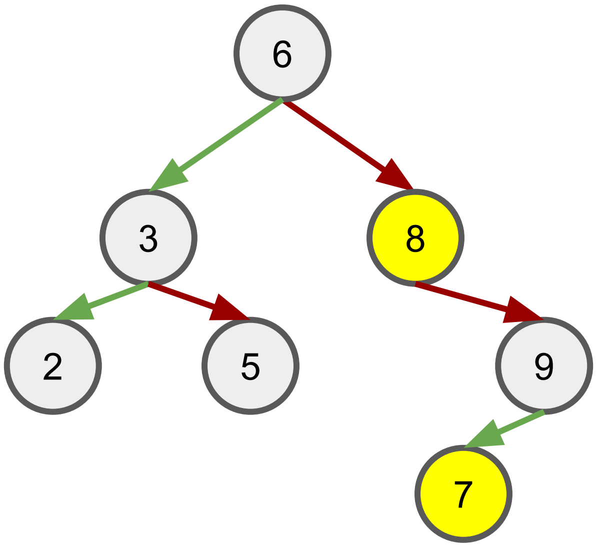
Is this a BST?



Question 1

Fun fact: There used to be CS2020, which is essentially a 6MC version of CS2030 + CS2040S in one mod

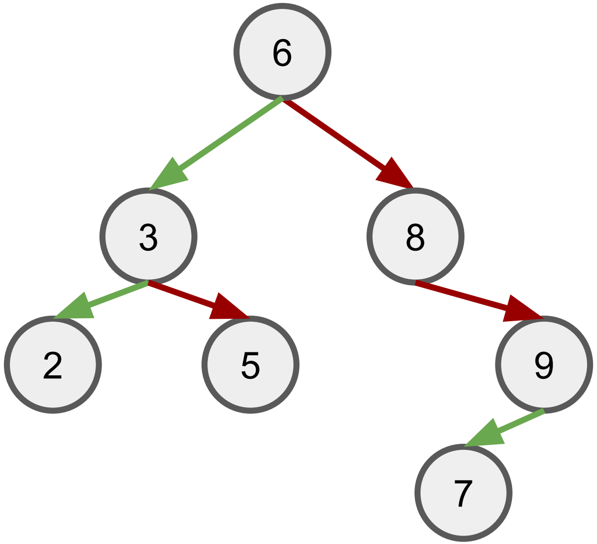
Is this a BST?



Question 1

NO! Compare these two nodes and notice that 7 belongs to the right subtree of 8

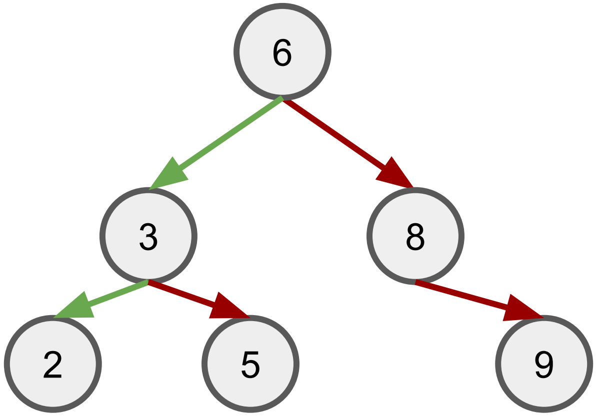
Is this a BST?



Question 1

If you were to (wrongly) use the idea that *only* the *direct left/right childs* have to be smaller/greater, then this fits. BUT THIS IS NOT A BST

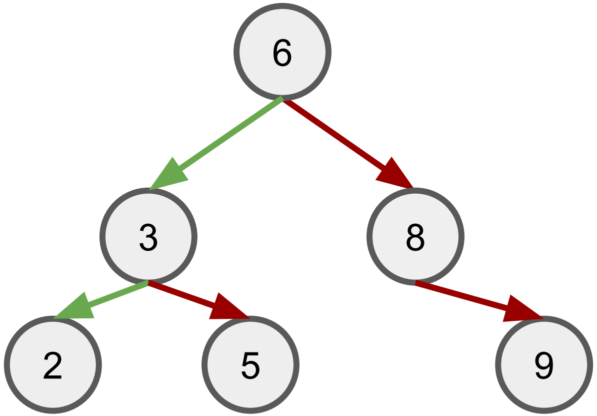
Is this a BST?



Question 2

Fun fact: Tony Hoare invented Quicksort at the age of 26 when he was a visiting student

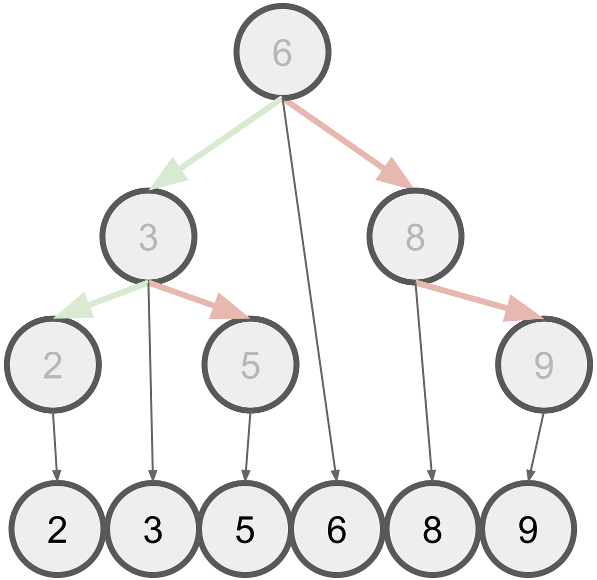
Is this a BST?



Question 2

Yes! Everything is gud

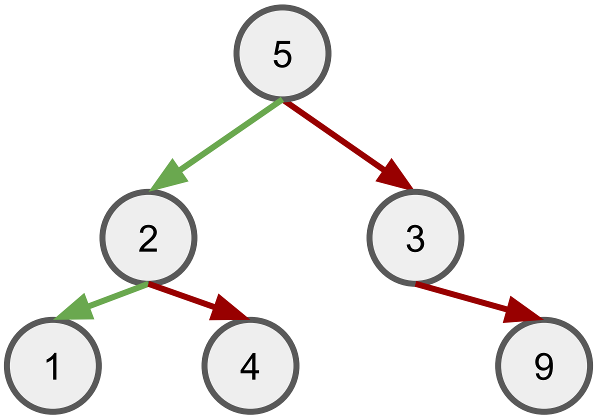
Is this a BST?



#LIFEHACKS

If you “drop” every node and it appears like a sorted sequence, then it is a BST

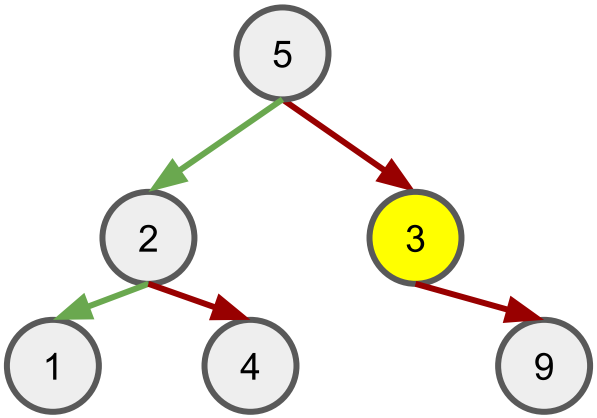
Is this a BST?



Question 3

Fun fact: Prof Seth Gilbert proved an important theorem called CAP theorem

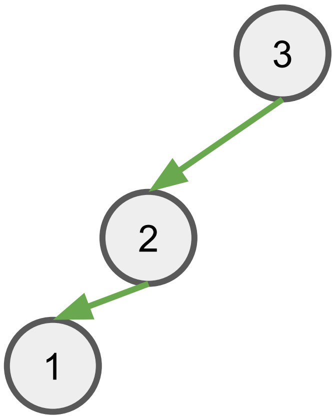
Is this a BST?



Question 3

Nope! Either use the “dropping” method or observe that 3 is in the right subtree of 5

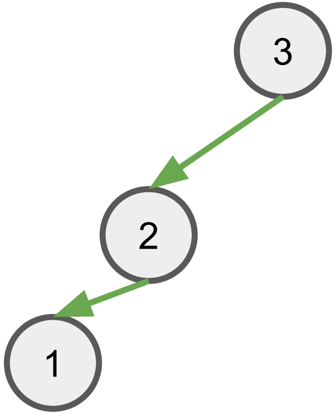
Is this a BST?



Question 4

Fun fact: Donald Knuth, the "father of analysis of algorithms" is *still* writing a book called The Art Of Computer Programming since 1968

Is this a BST?



Question 4

Yeboi

Is this a BST?

Question 5

Fun fact: COM3 which is being built, is the first building that is built for SoC (we inherited COM1 and COM2 from Law i think)

Is this a BST?

Question 5

This lonely boi is also a BST!
(You can think of the left and right childs being empty trees)

Summary

h(v) - height of a node: Number of **edges** on the *longest path* from node to leaf.

- h(v) = 0 (if v is a leaf)
- h(v) = max(h(v.left), h(v.right)) + 1

Questions?

Operations in a BST

- Searching
- Insertion
- Search Minimum and Maximum
- Successor and Predecessor
- Delete
- Traversal

Operations in a BST

- Searching
- Insertion
- Search Minimum and Maximum
- Successor and Predecessor
- Delete
- Traversal

Searching

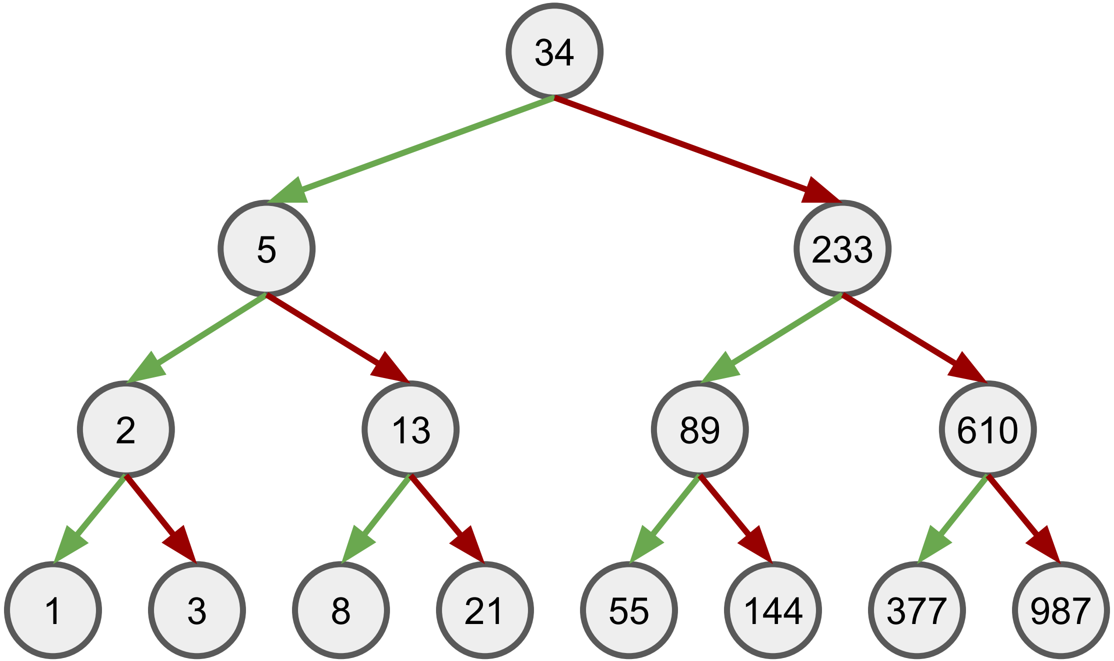
Idea:

- Compare root vs element you are looking for
- If element == root, found!
- Else if element < root, recurse to the left subtree
- Else element > root, recurse to the right subtree

Searching

search(144)

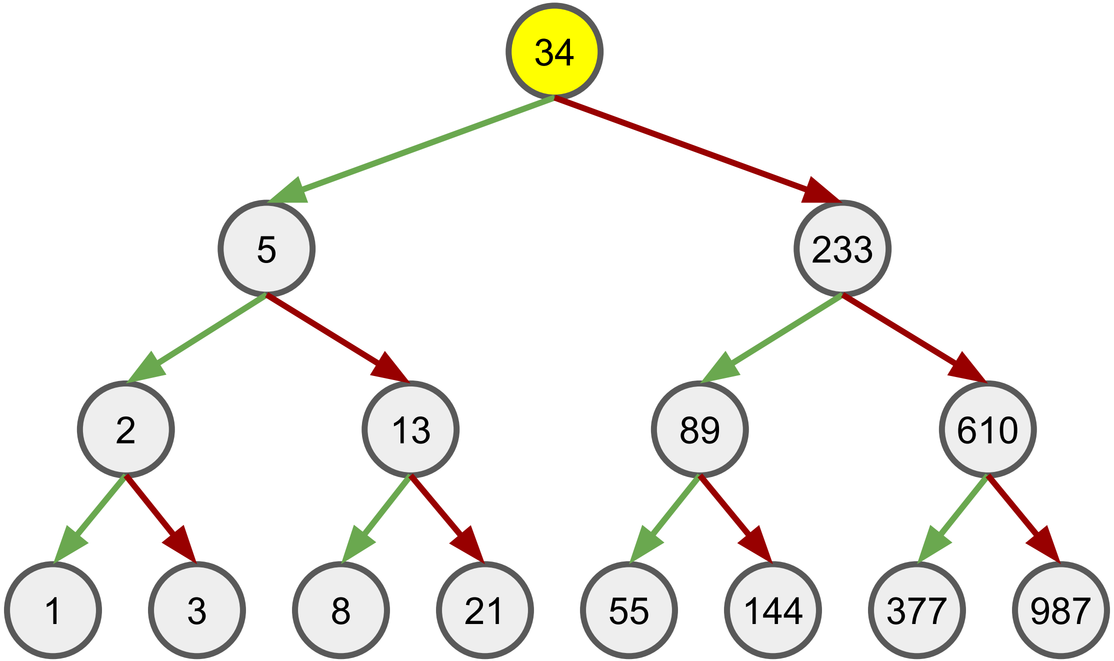
Initialisation



Searching

search(144)

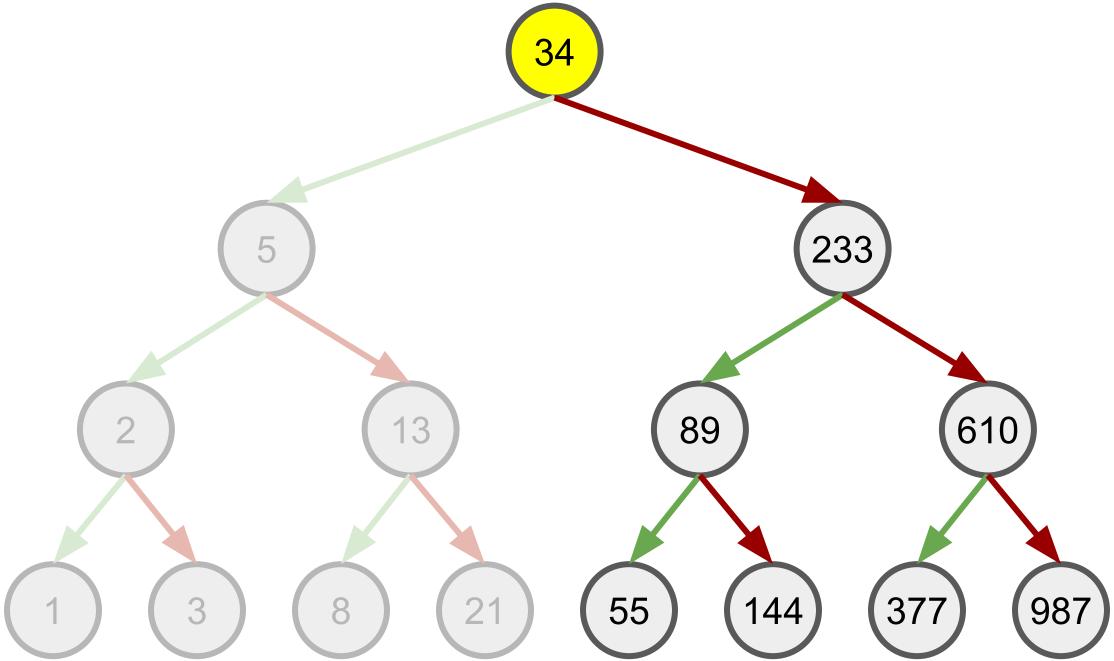
Compare 144 with 34



Searching

`search(144)`

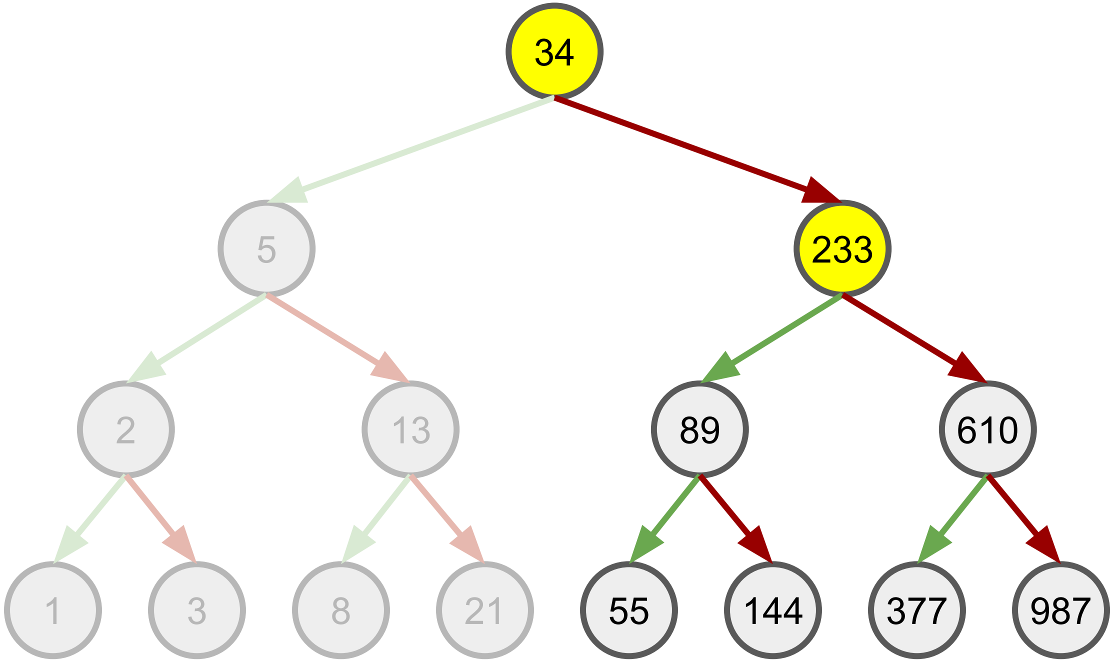
144 is greater than 34. Since we are in BST, we are guaranteed that 144 cannot be in the left subtree



Searching

search(144)

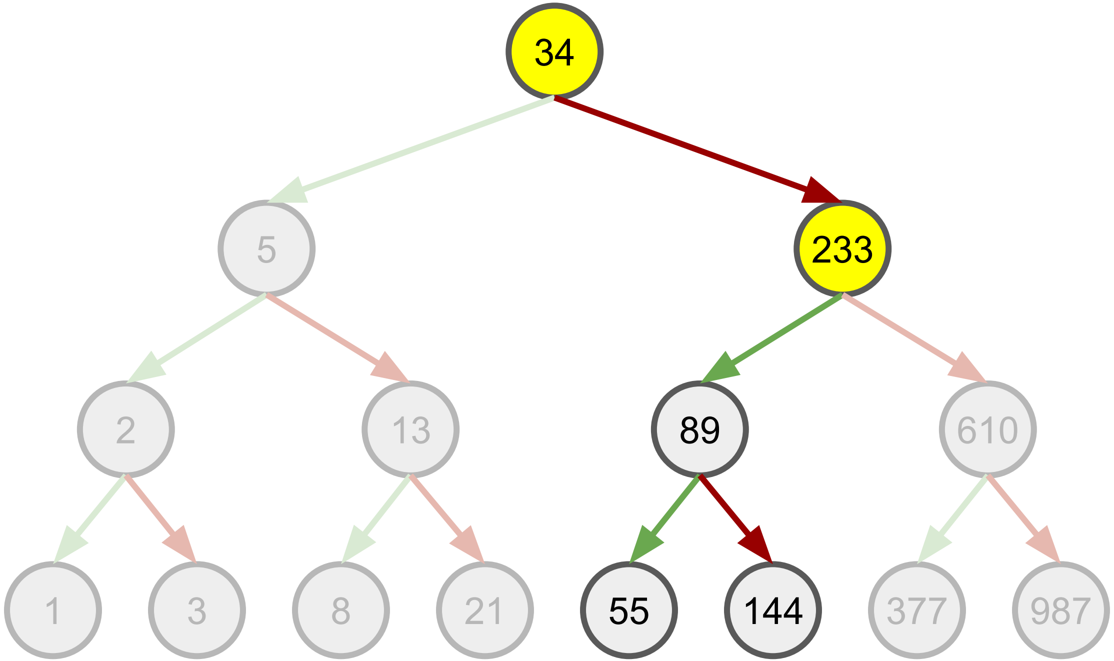
Compare 144 with 233



Searching

search(144)

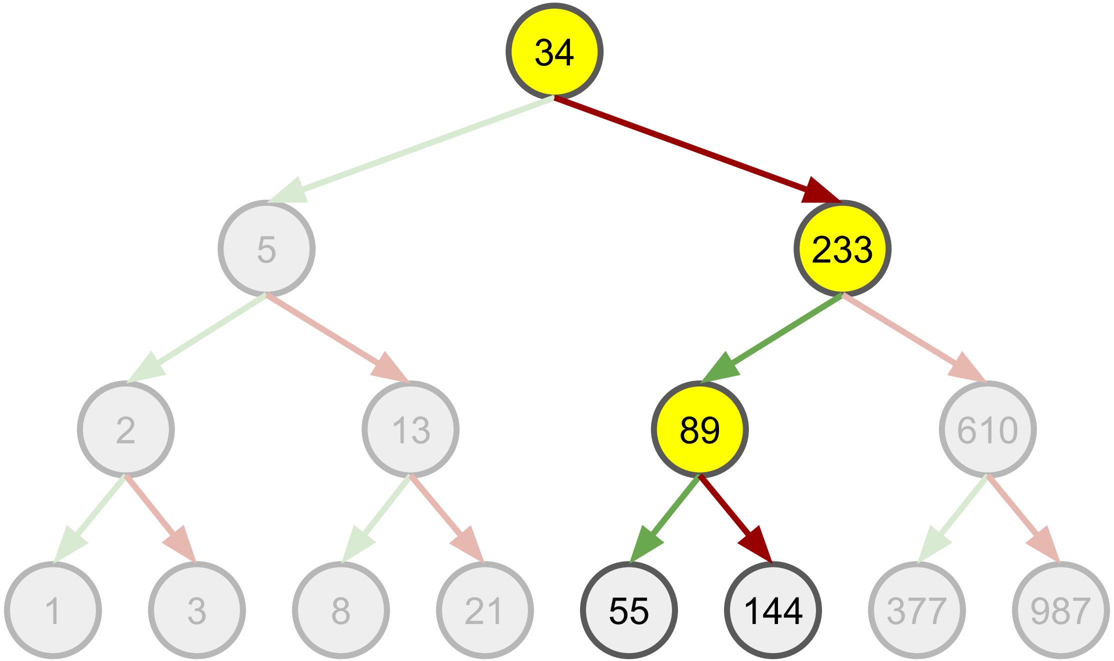
144 is less than 233. We can ignore the right subtree



Searching

search(144)

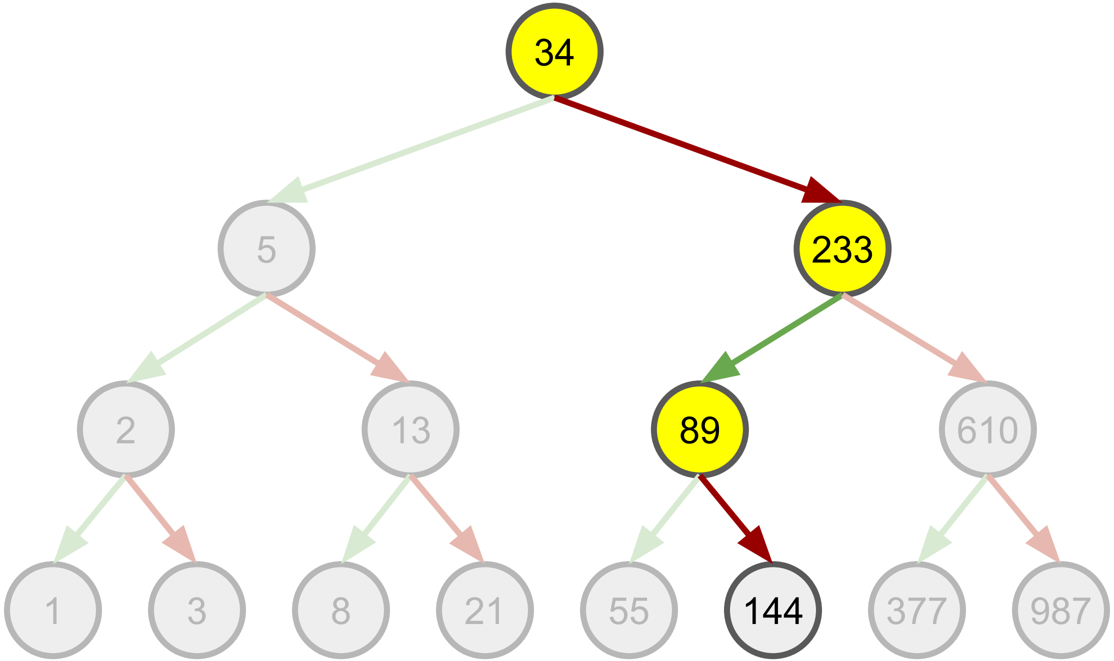
Compare 144 with 89



Searching

search(144)

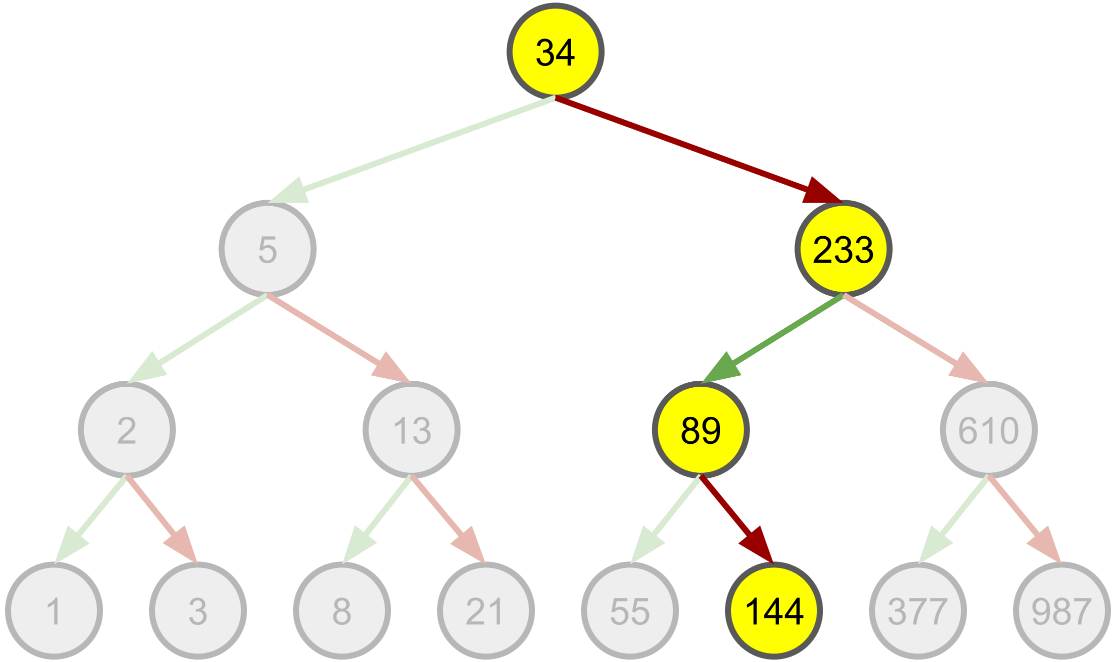
144 is greater than 89 so we can ignore the left subtree



Searching

search(144)

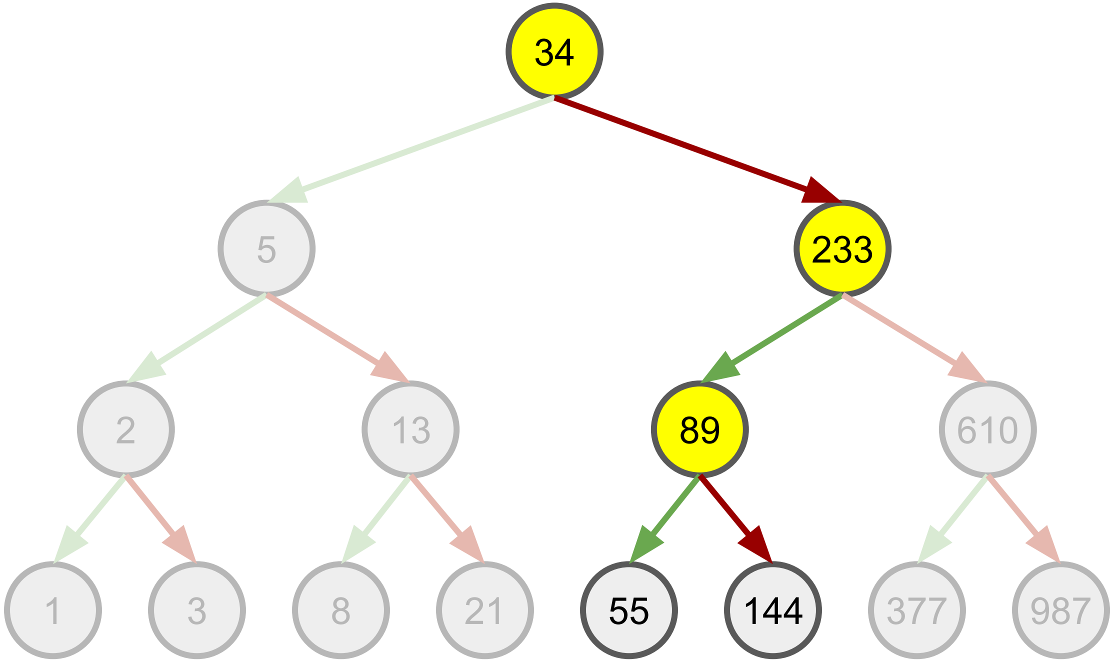
Compare 144 with 144, and found!



Searching (eg2)

search(145)

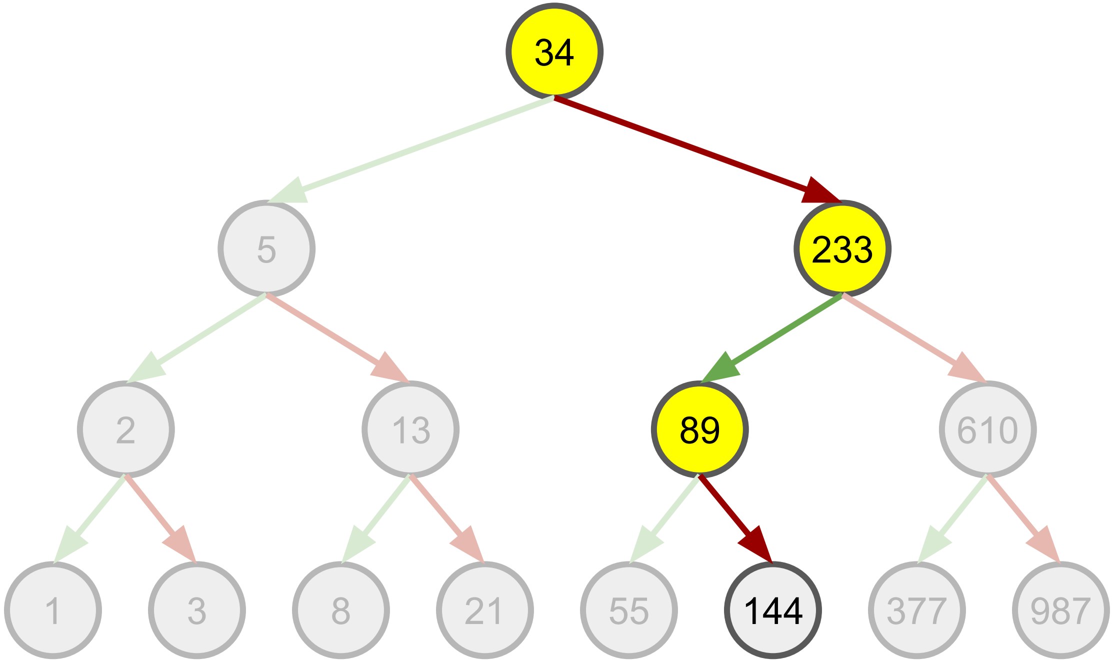
If for example we are searching for 145 instead of 144. Notice that everything is the same up to this point



Searching (eg2)

search(145)

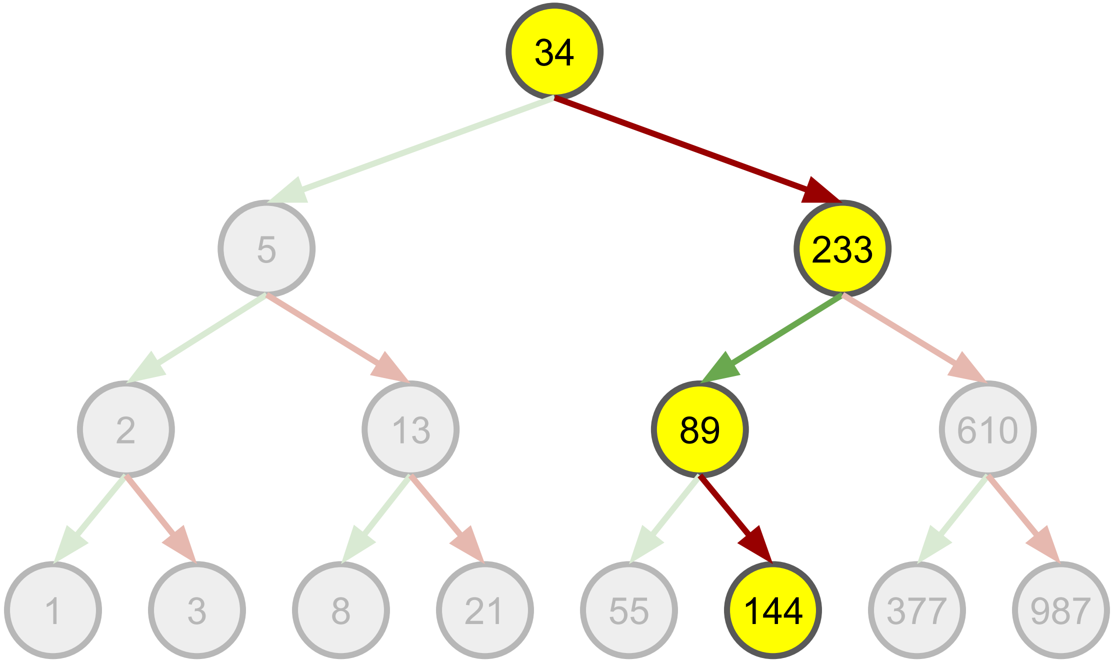
145 is greater than 89 so we can ignore the left subtree



Searching (eg2)

search(145)

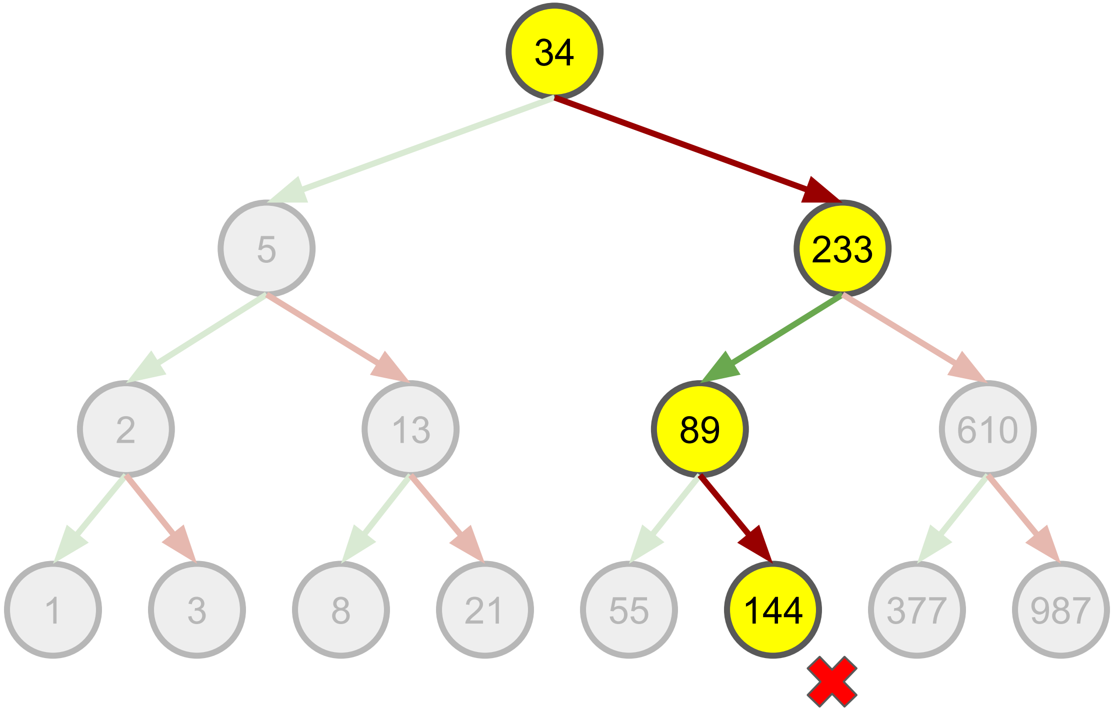
Compare 145 with 144



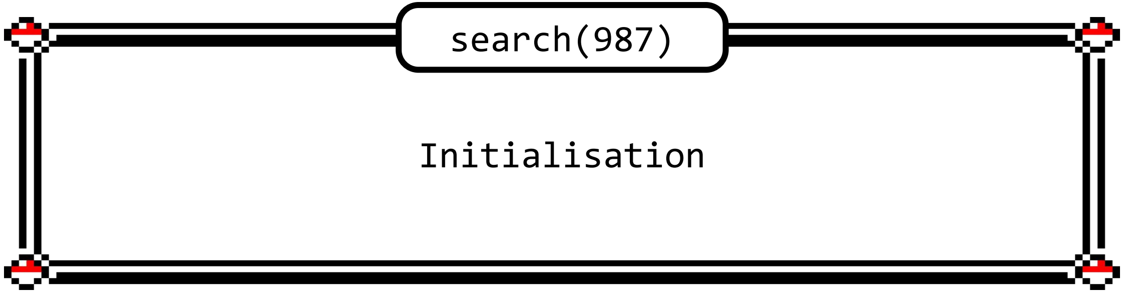
Searching (eg2)

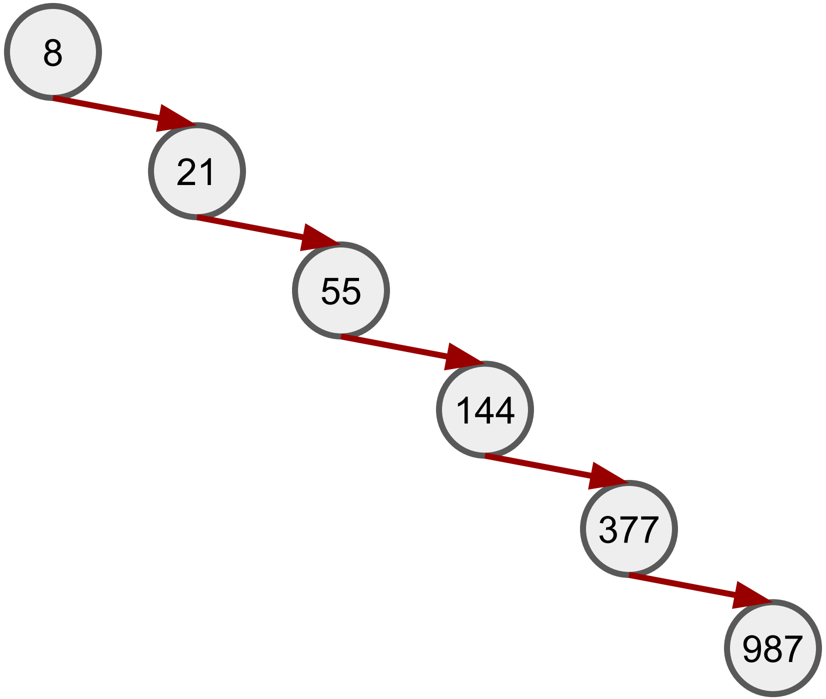
search(145)

We are supposed to explore the right subtree, but
144 does not have a right subtree!
We can report ‘not found’



Searching (eg3)

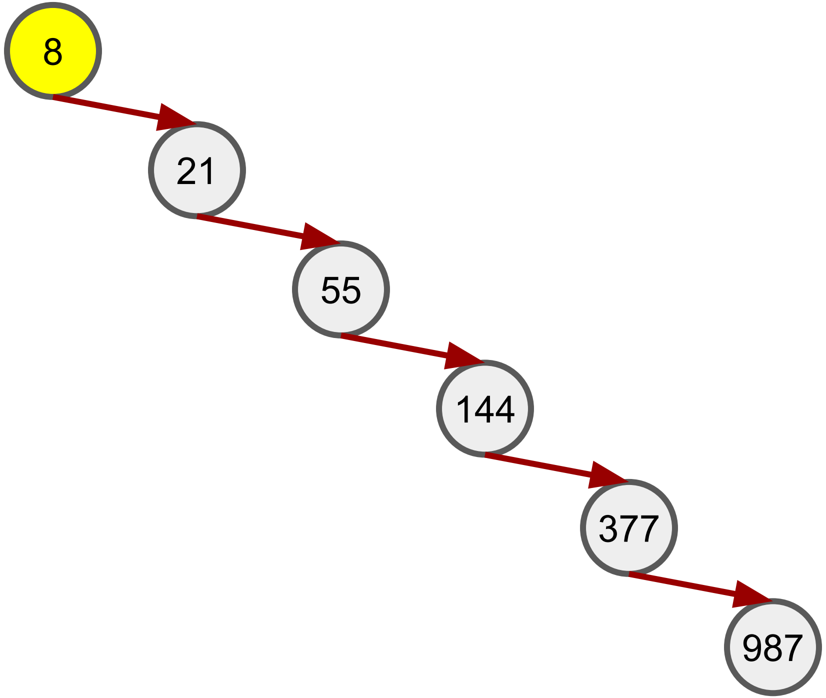




Searching (eg3)

search(987)

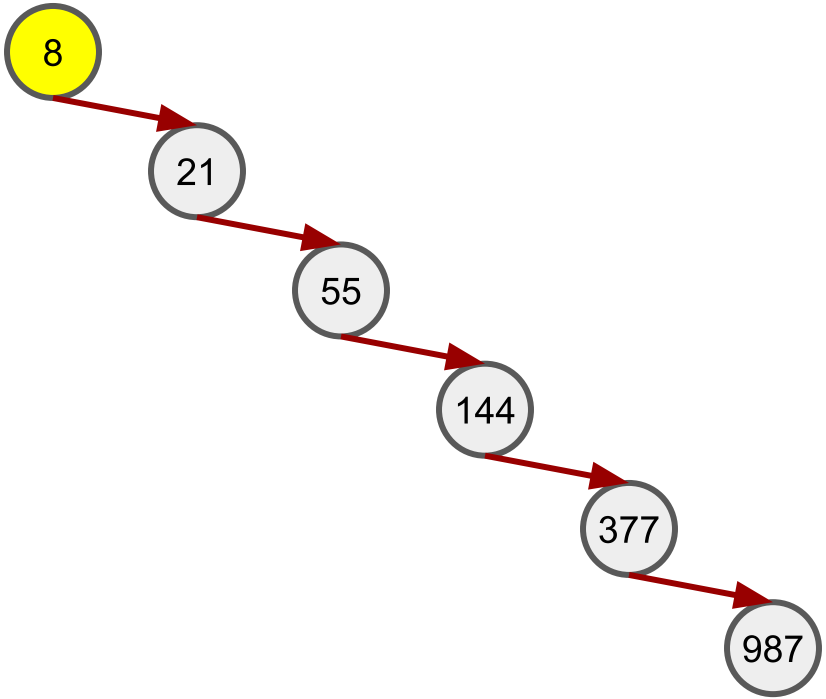
Compare 987 with 8



Searching (eg3)

search(987)

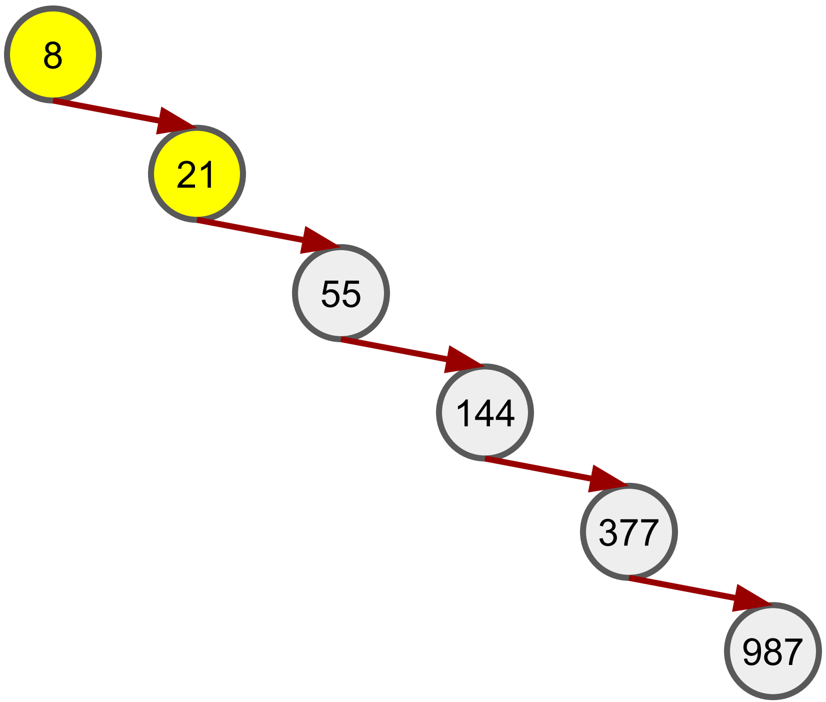
987 cannot be in the left subtree. So we explore the right subtree



Searching (eg3)

search(987)

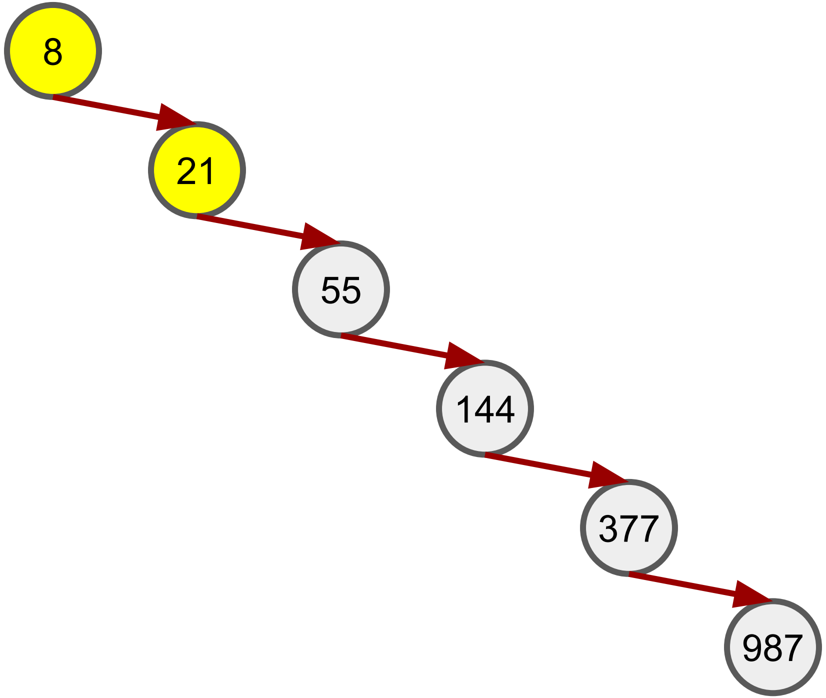
```
Compare 987 with 21
```



Searching (eg3)

search(987)

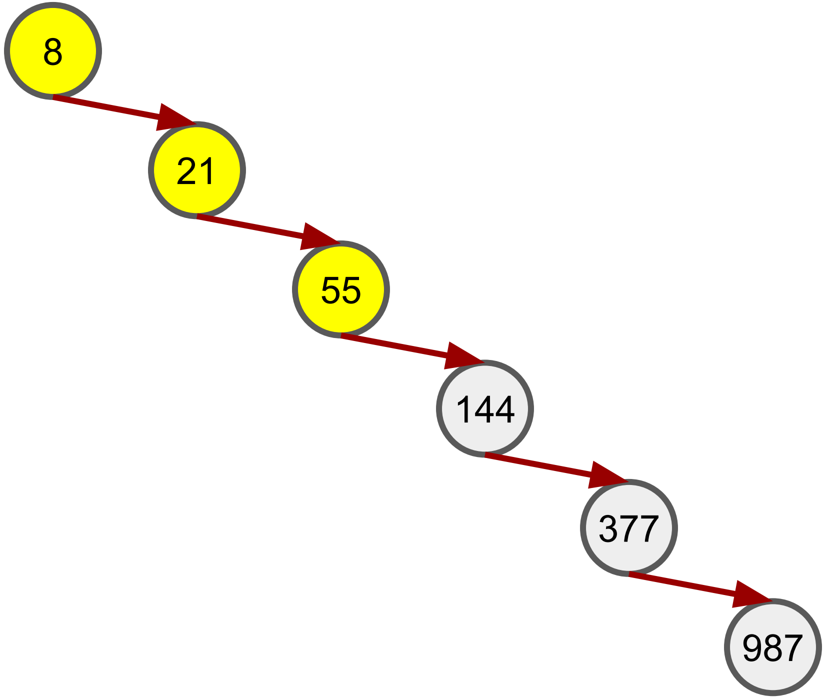
Also the same thing. Go right!



Searching (eg3)

search(987)

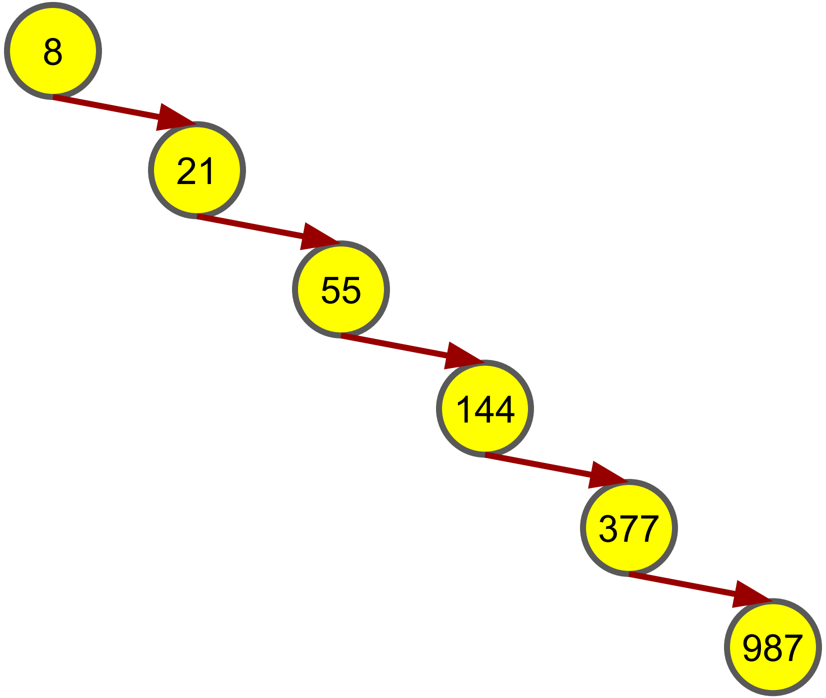
and so on...



Searching (eg3)

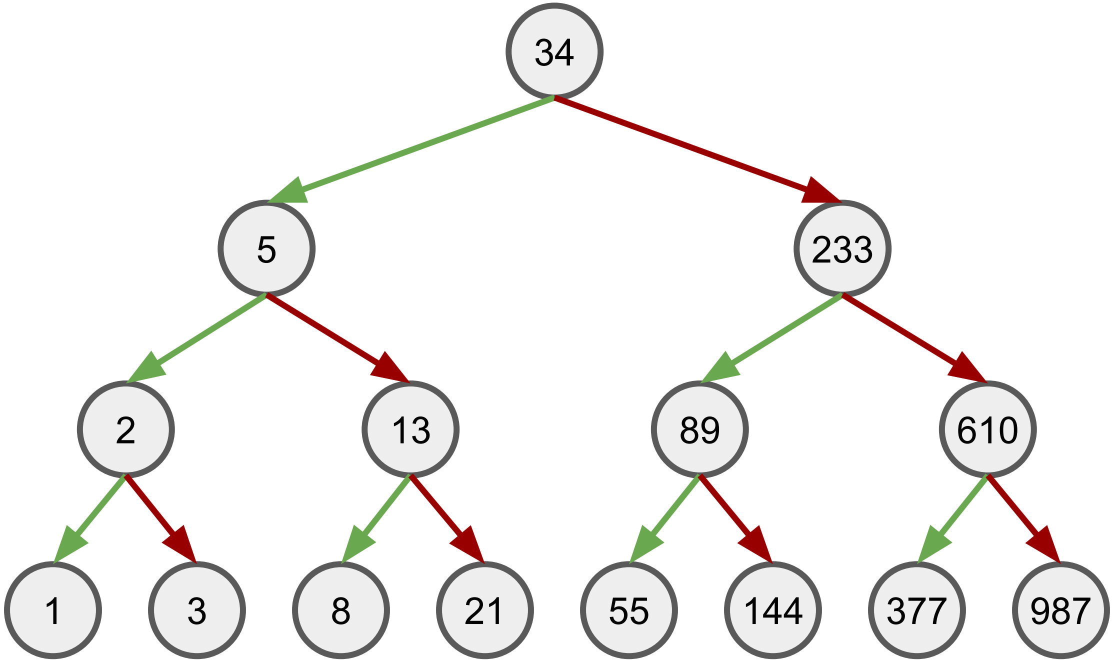
search(987)

will spare you the agony of going through several slides xD



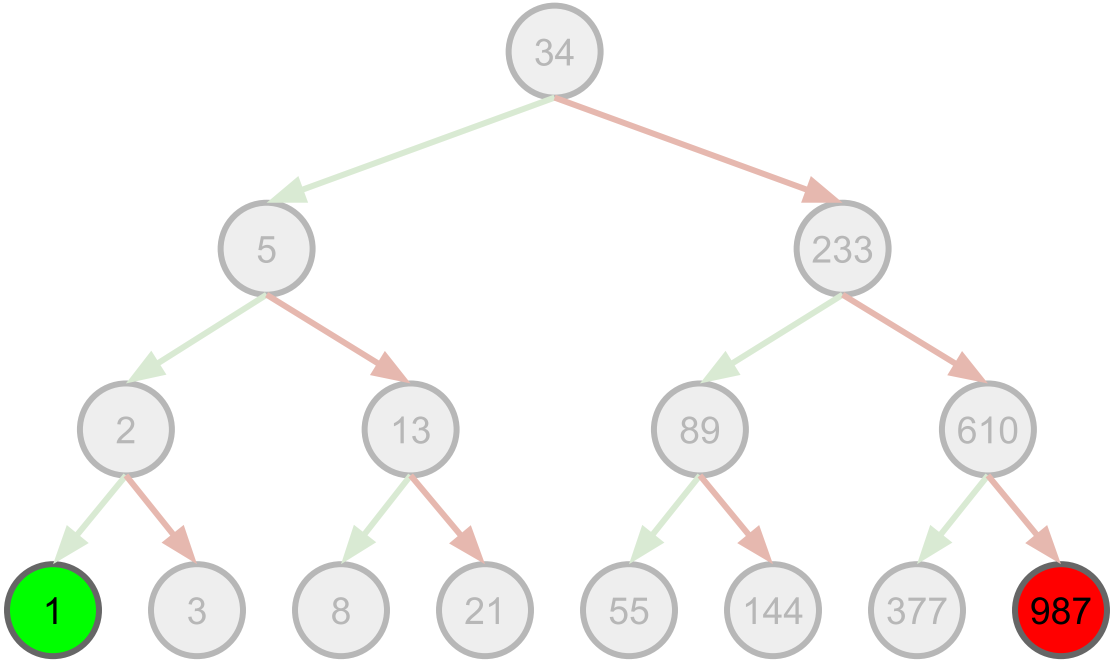
Searchmin and Searchmax

Where are the smallest and largest elements located in the tree?



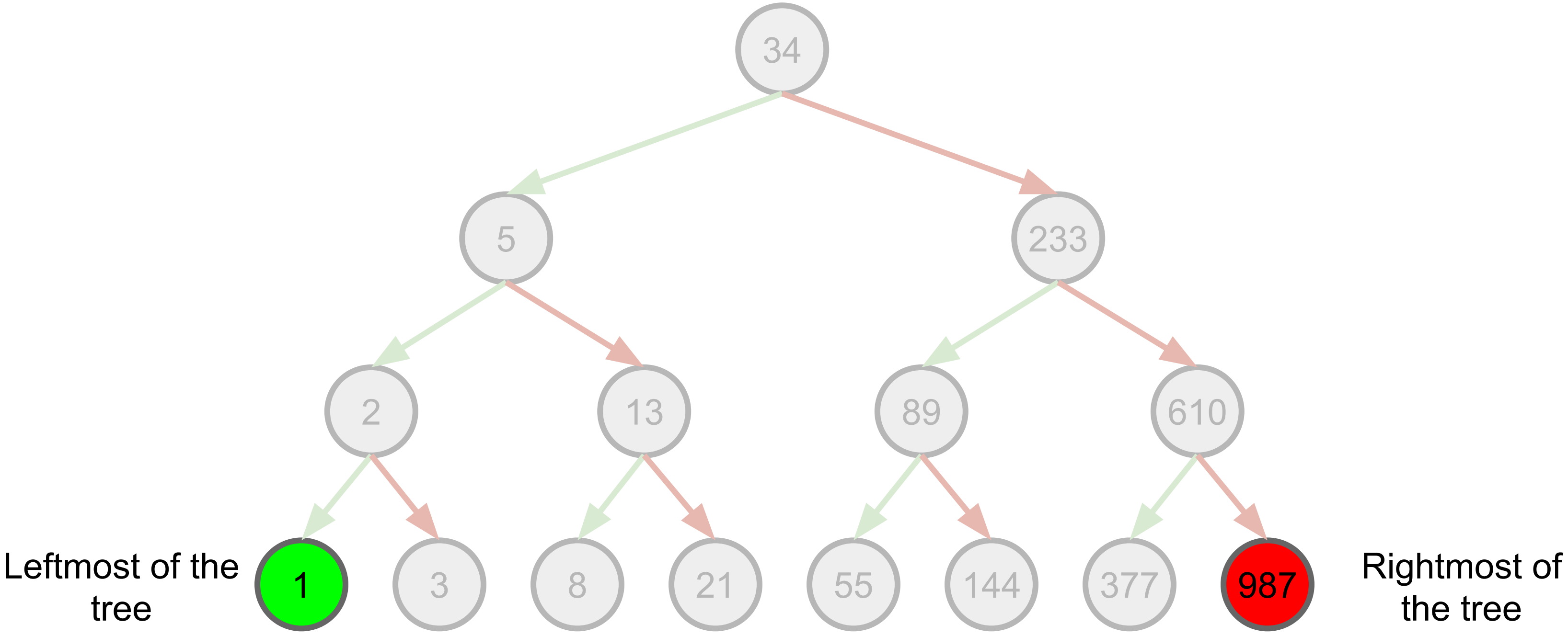
Searchmin and Searchmax

Where are the smallest and largest elements located in the tree?



Searchmin and Searchmax

Where are the smallest and largest elements located in the tree?



Searchmin and Searchmax

The idea:

- Just keep going down the left subtree (for min) or the right subtree (for max) ‘til you can’t no more

Questions?

Successor and Predecessor

Successor and Predecessor

The successor of key x is basically the “next bigger key” in the tree

The predecessor of key x is basically the “previous smaller key” in the tree

Successor and Predecessor

The successor of key x is basically the “next bigger key” in the tree

The predecessor of key x is basically the “previous smaller key” in the tree

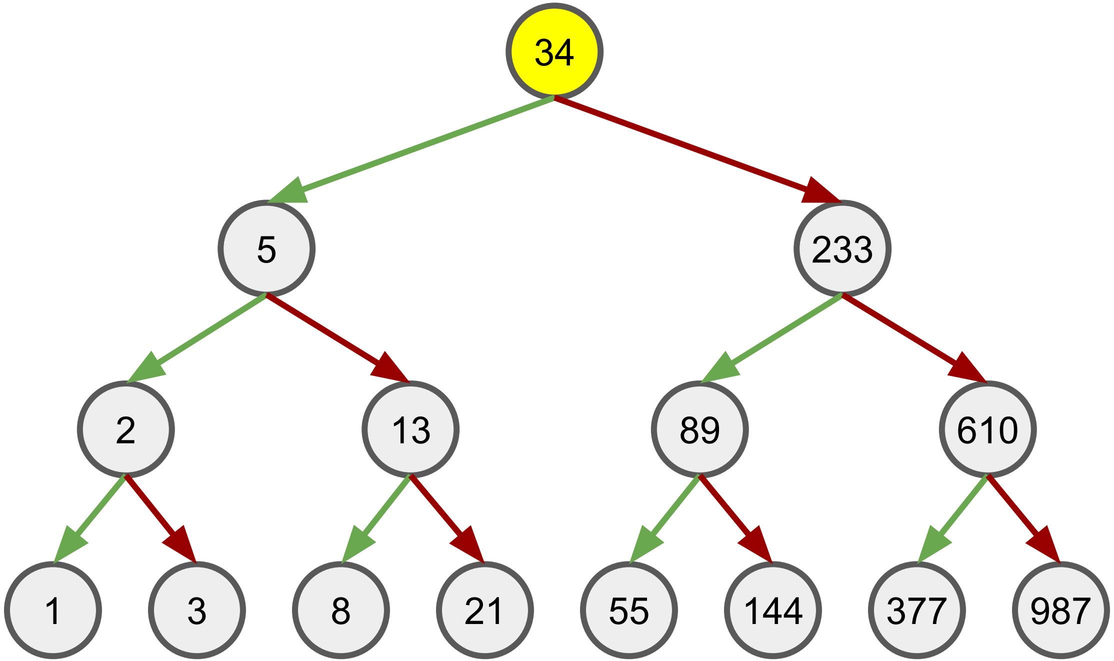
Another way to think about it: If you perform an in-order traversal (“write the keys in sorted order”), the successor is the next one in the sorted sequence and the predecessor is the previous one in the sorted sequence*

*If the key doesn’t exist in the tree, pretend you have inserted it in the sorted order for visualisation

Successor (Case 1)

successor(34)

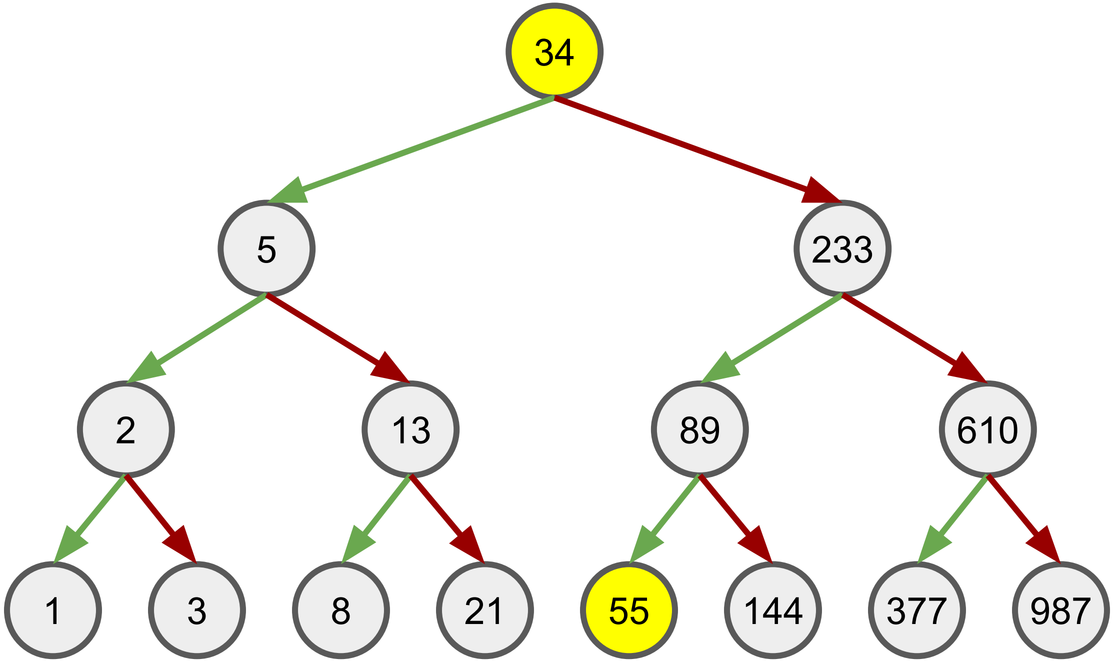
Who is the successor of 34?



Successor (Case 1)

successor(34)

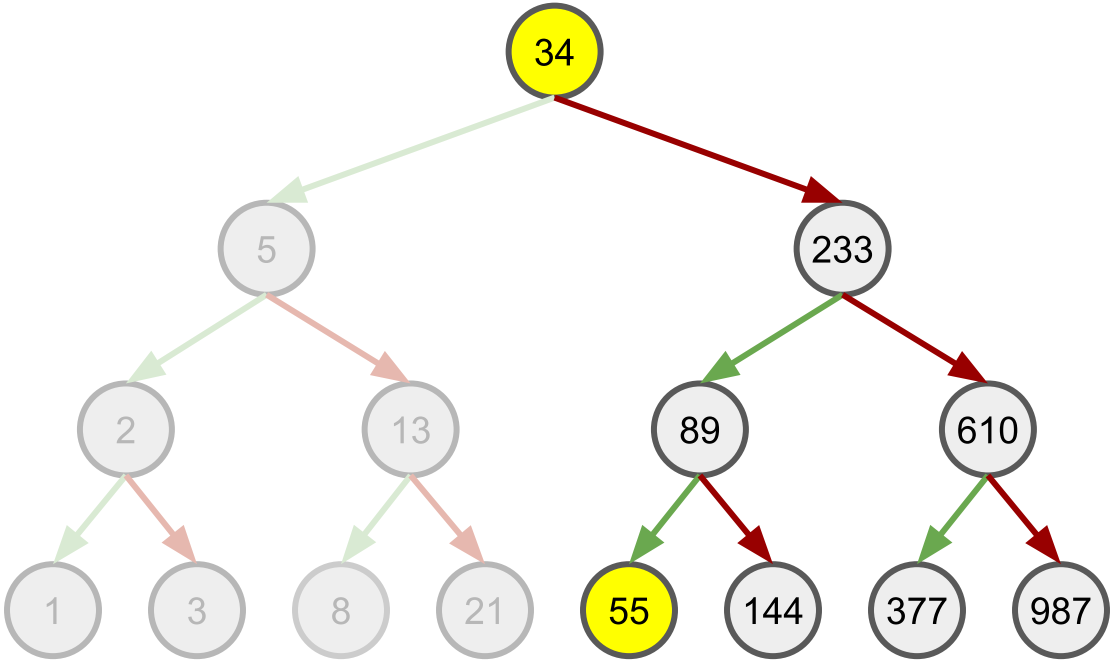
It’s 55!



Successor (Case 1)

successor(34)

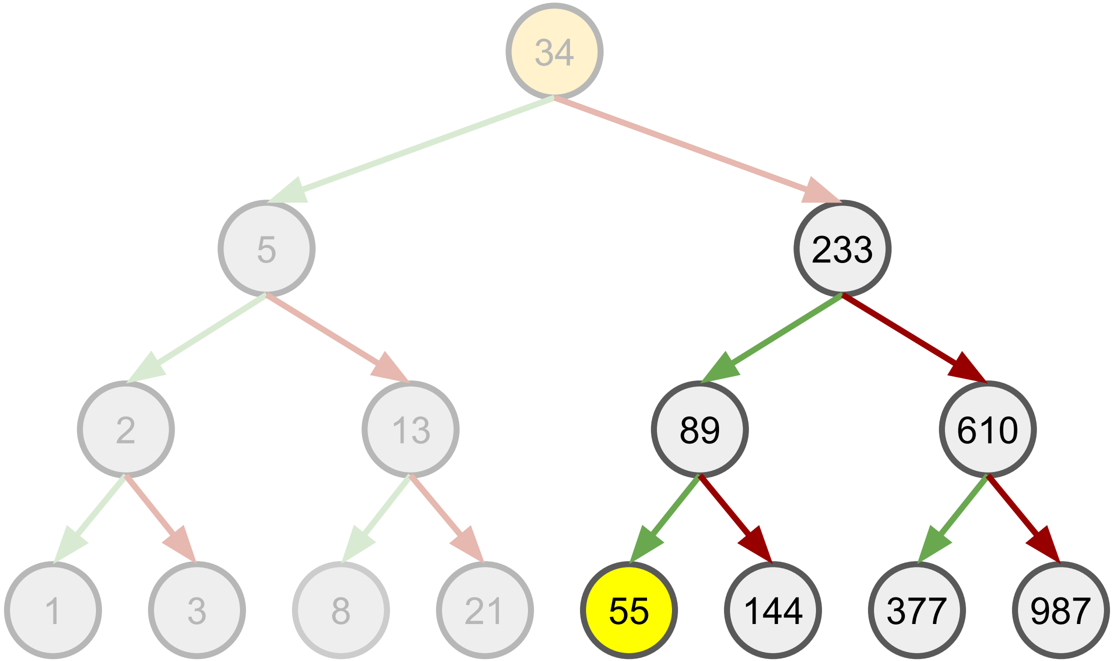
Do you notice something about where 55 is located?



Successor (Case 1)

successor(34)

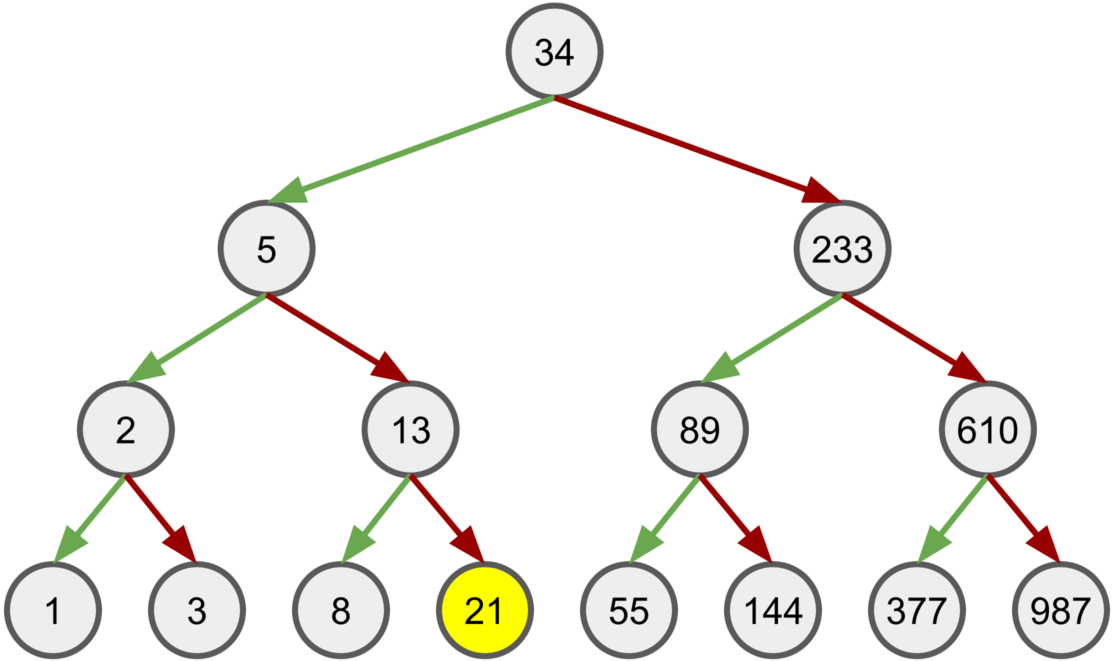
It's the smallest element of 34's right subtree!



Successor (Case 2)

successor(21)

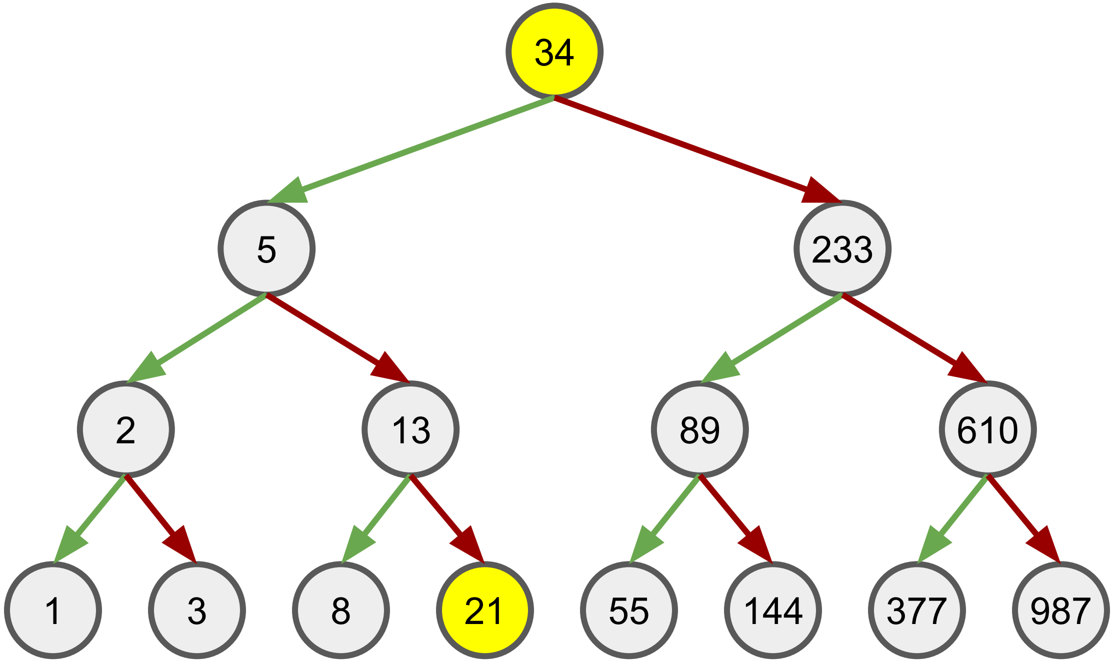
What if the node doesn't have a right subtree like this node with key 21 here?
First, what is this node's successor?



Successor (Case 2)

successor(21)

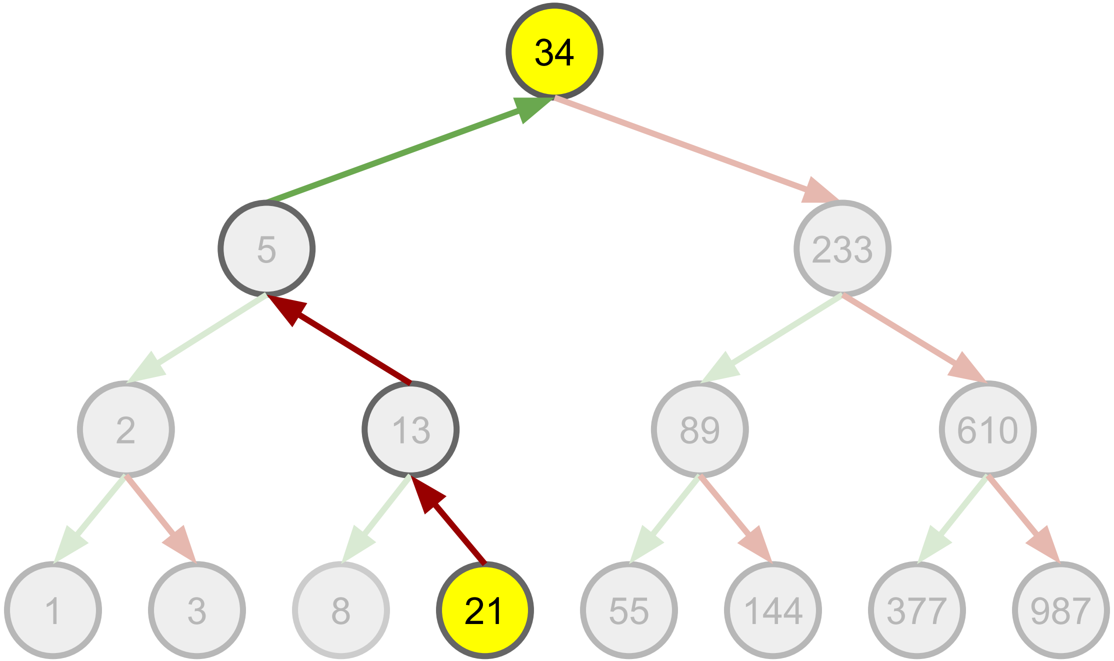
34 is the successor. How did we get to 34?



Successor (Case 2)

successor(21)

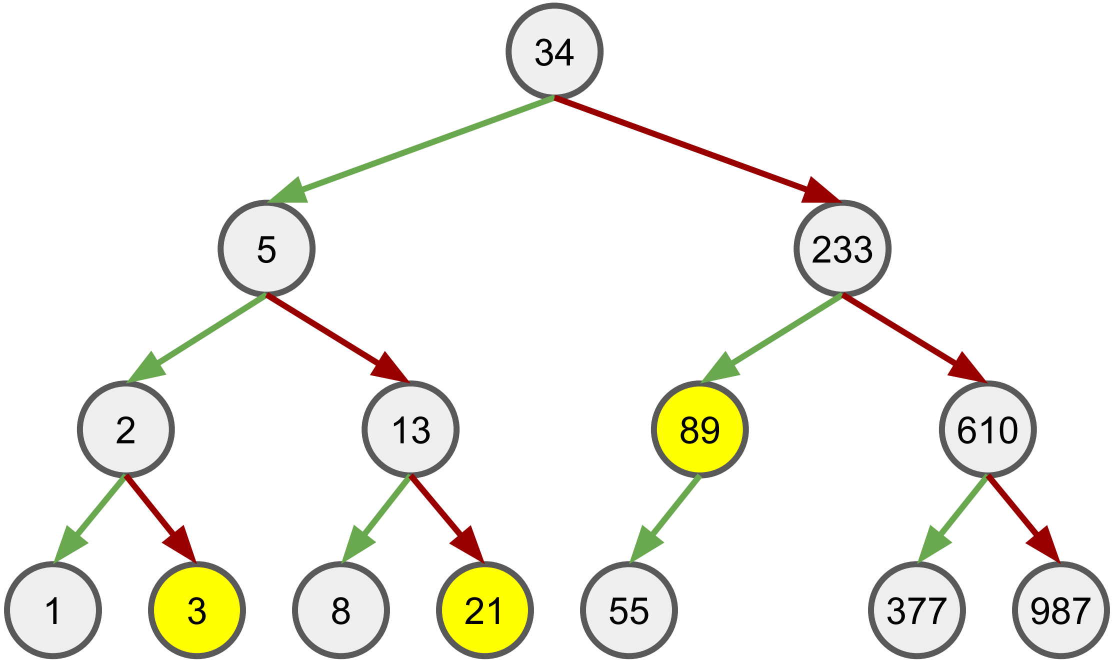
Essentially, keep going up the tree until we have to “climb to the right”



Successor (Case 2)

successor

Verify that this pattern holds for other cases



Questions?

Insert

The idea:

Insert

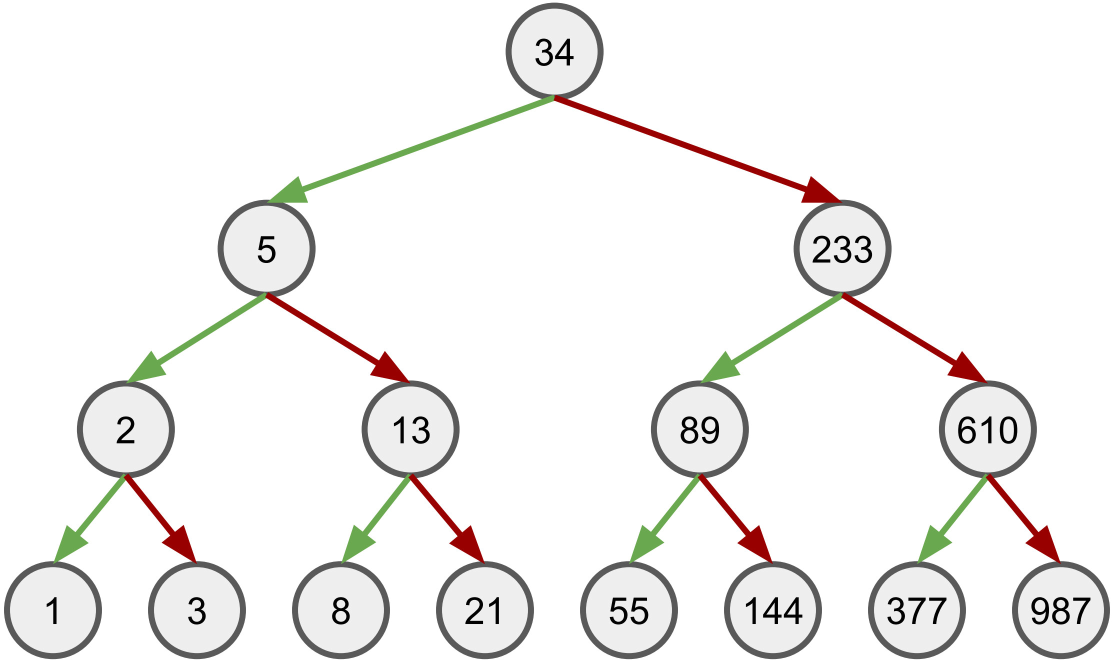
The idea:

- Keep going as if you are “searching”
- Once you cannot go on anymore (because you need to go to the right/left subtree but the right/left subtree is empty), you have found the position to insert

Insert

insert(155)

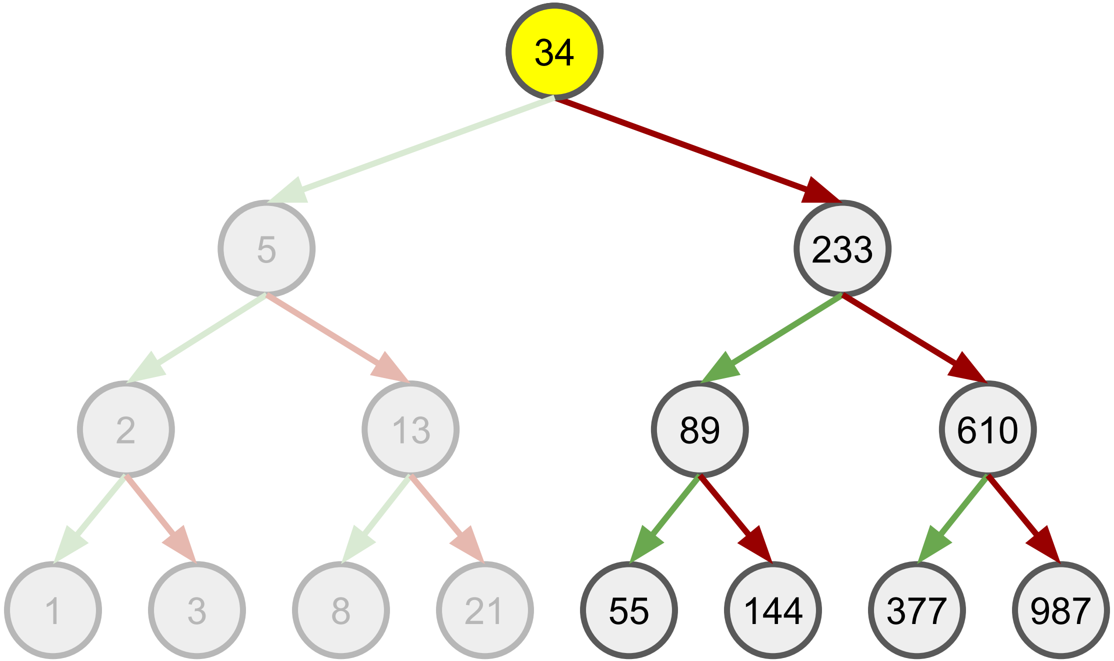
Initialisation



Insert

insert(155)

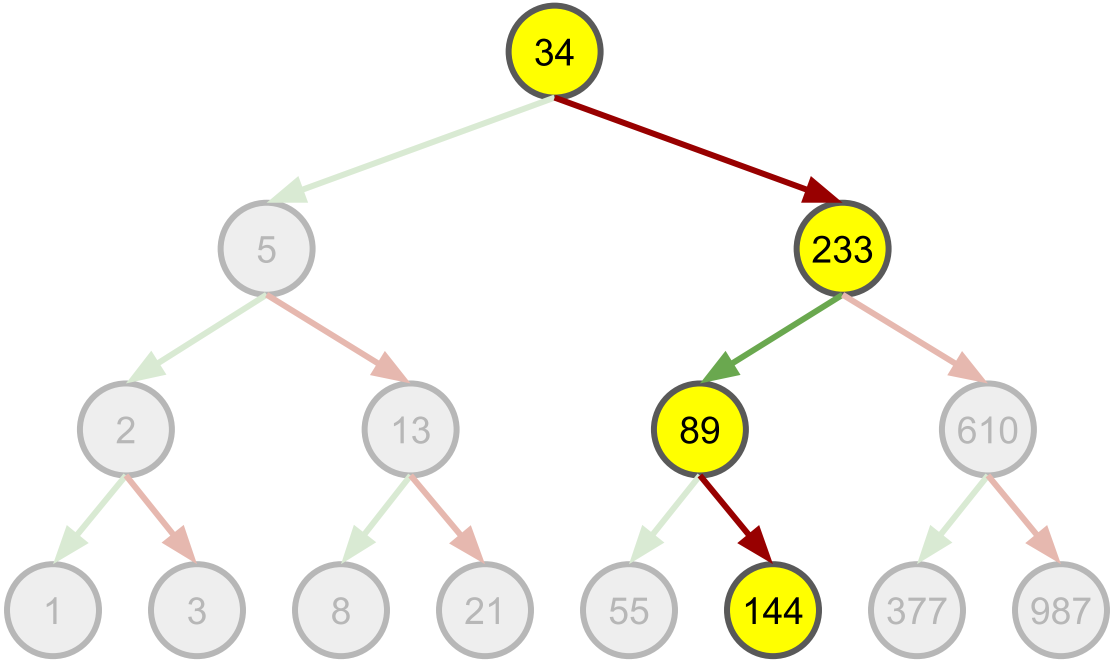
Go to the right subtree



Insert

insert(155)

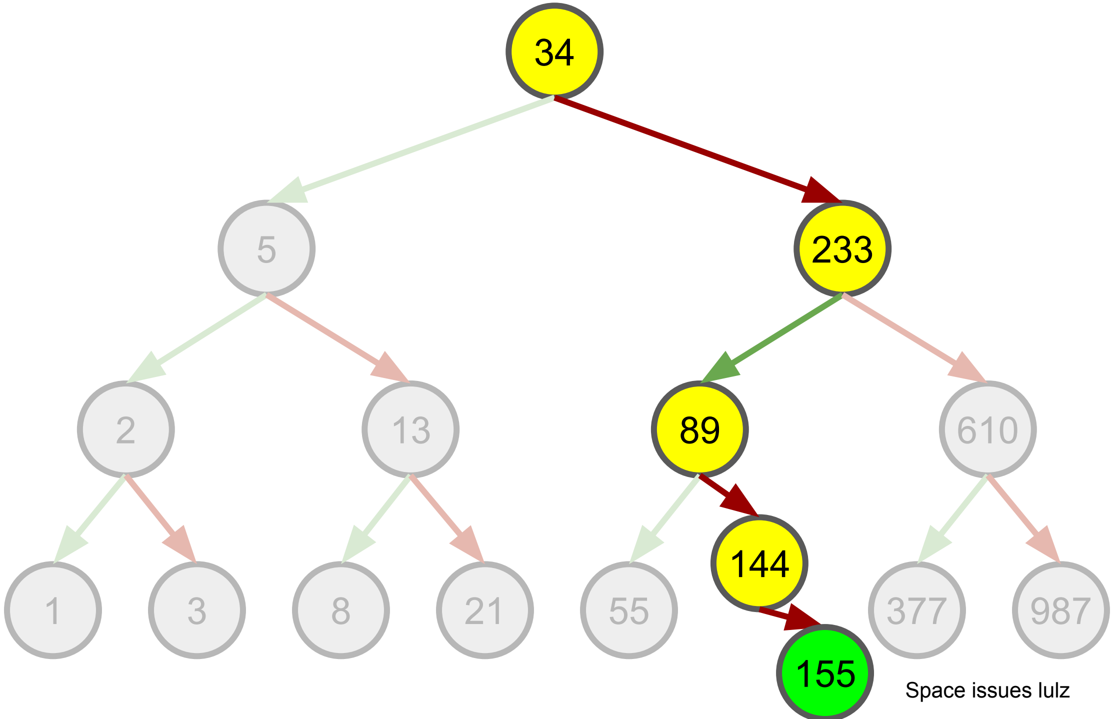
Same deal for the most part



Insert

insert(155)

Now you are supposed to go to the right subtree, but it’s empty! So that’s where you insert



Space issues lulz

Questions?

Deletion

A data structure is not interesting if we cannot remove anything from it! We need to be able to perform deletions:

There are multiple cases for deletion with respect to the target node to delete:

- Node is not found
- Node is at leaf
- Node is an internal node with 1 child
- Node is an internal node with 2 children

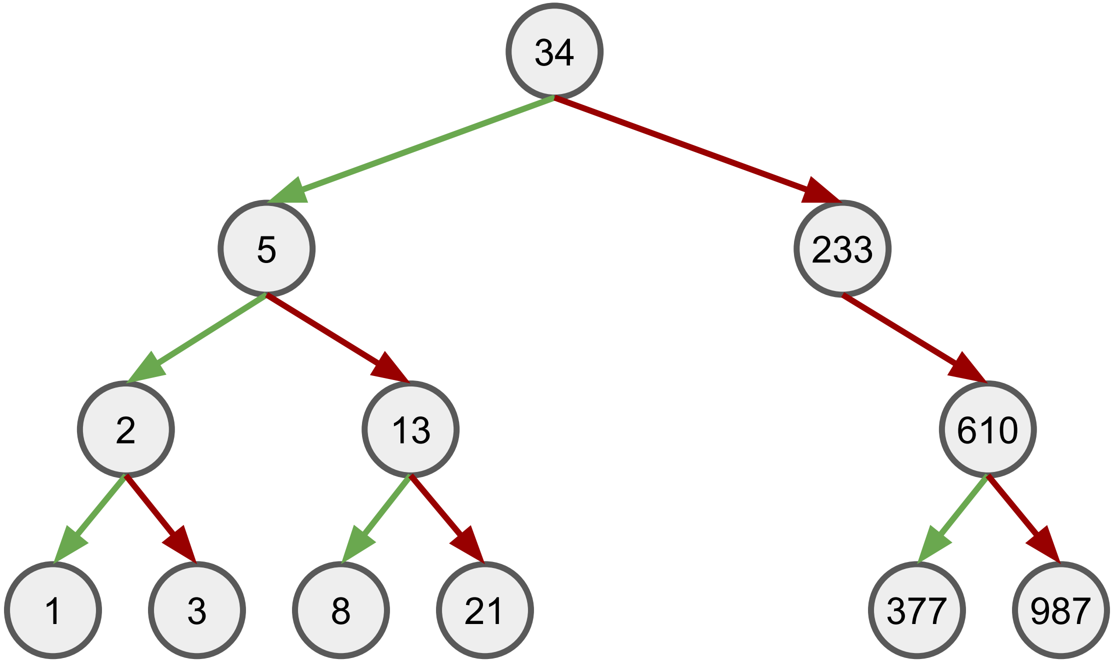
Note: The examples of binary trees in the following slides will differ by case for convenience in demonstration

Delete (Case 1)

[Node not found]

delete(40)

Initialisation

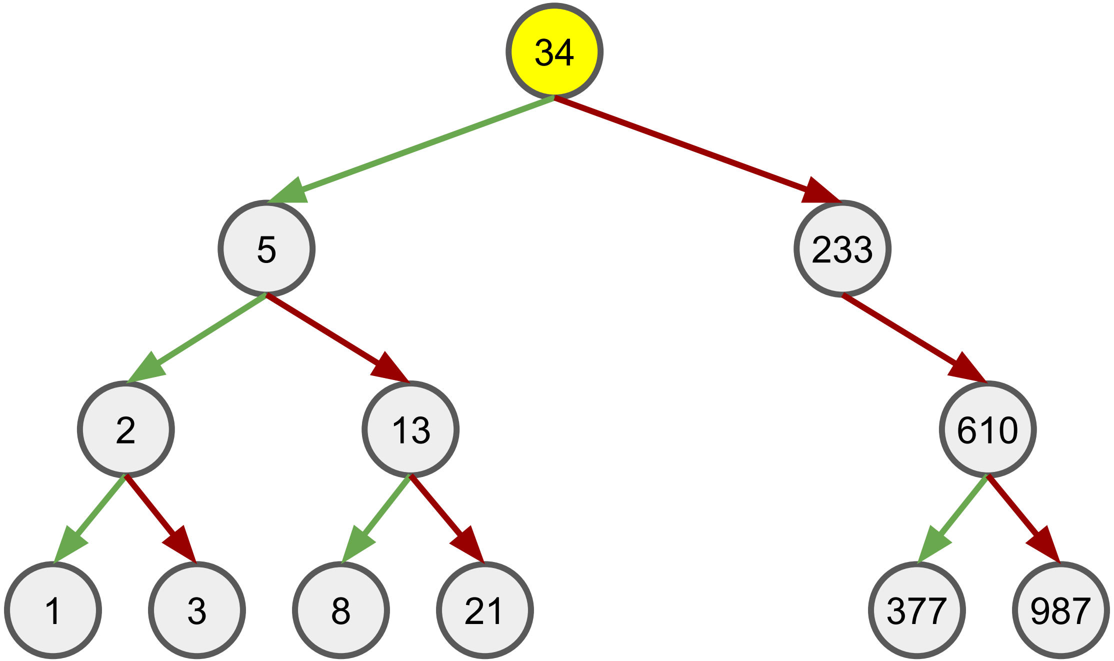


Delete (Case 1)

[Node not found]

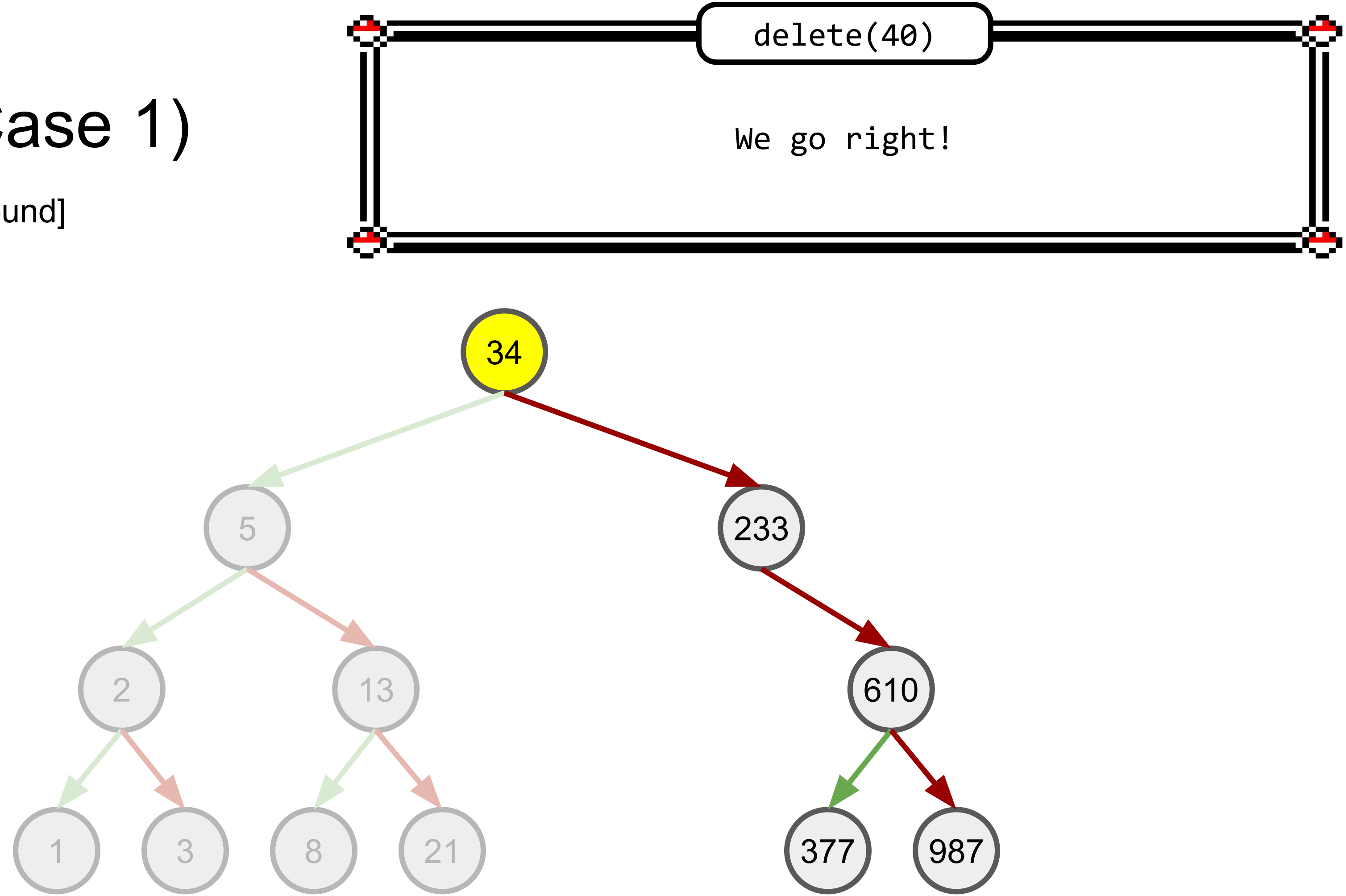
```
delete(40)

Compare 40 with 34
```



Delete (Case 1)

[Node not found]

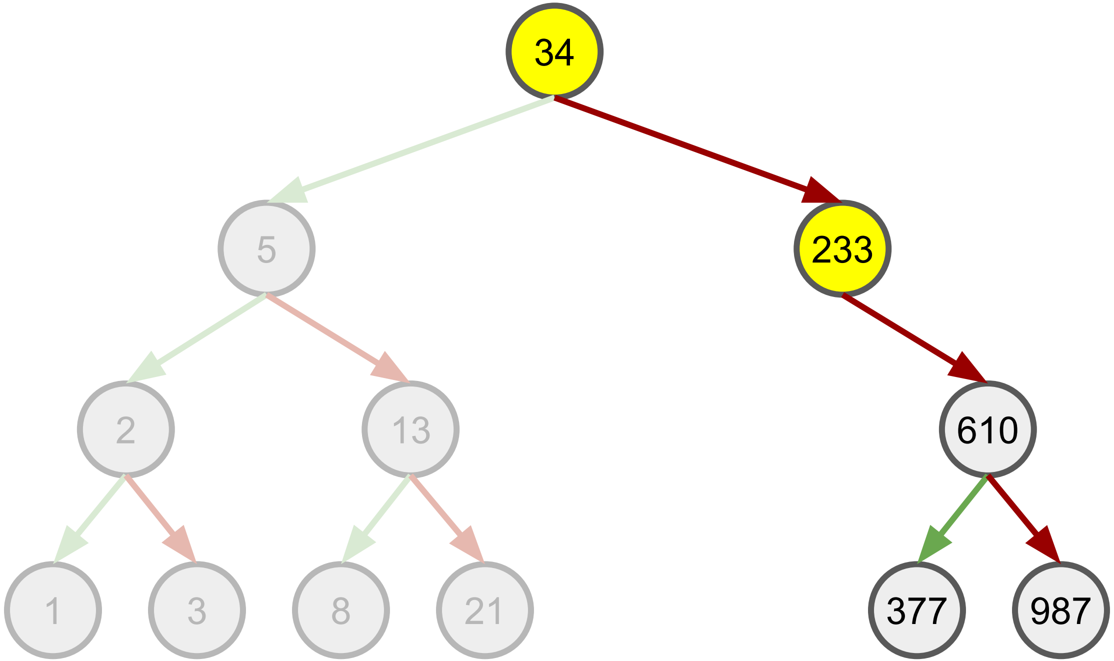


Delete (Case 1)

[Node not found]

delete(40)

40 is less than 233. But the left subtree is empty so it must mean that 40 does not exist. Do nothing!

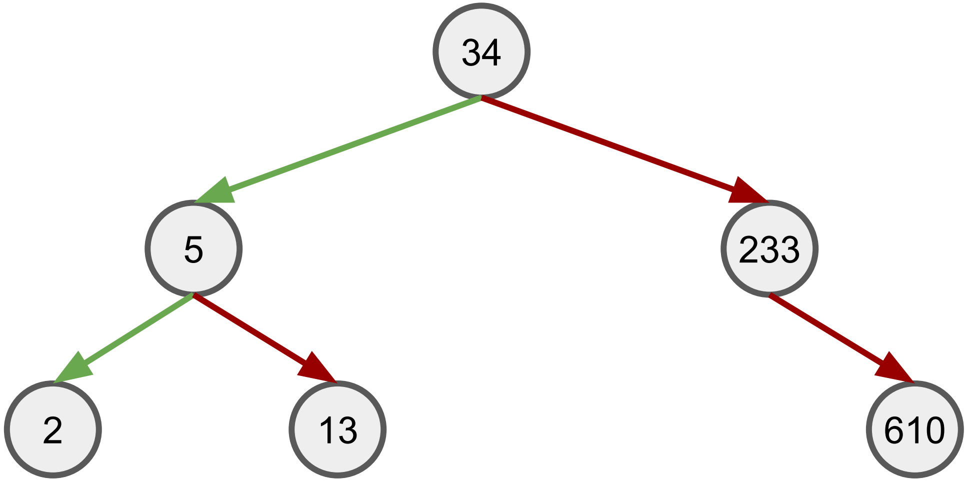


Delete (Case 2)

[Node is at a leaf]

delete(610)

Initialisation

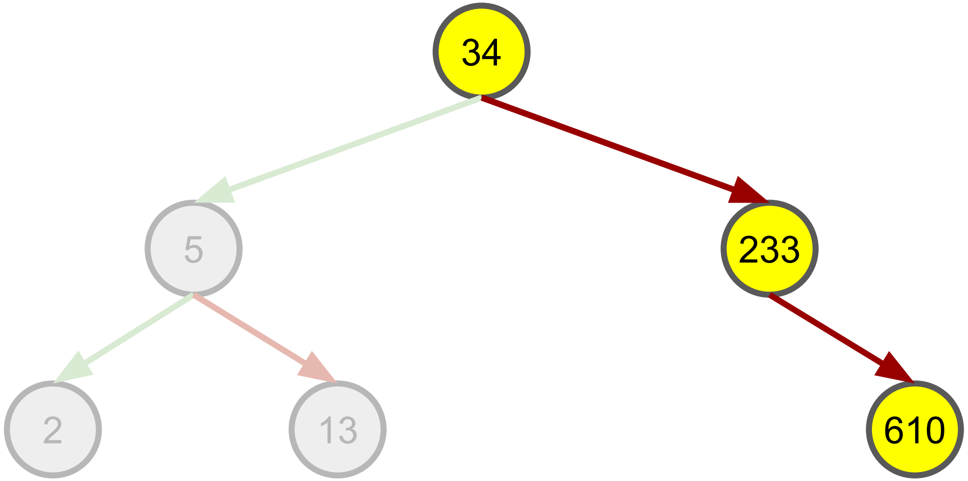


Delete (Case 2)

[Node is at a leaf]

delete(610)

I’ll skip the “searching” step :P
Let’s say we have traversed all the way to the right and found the node. wat do now

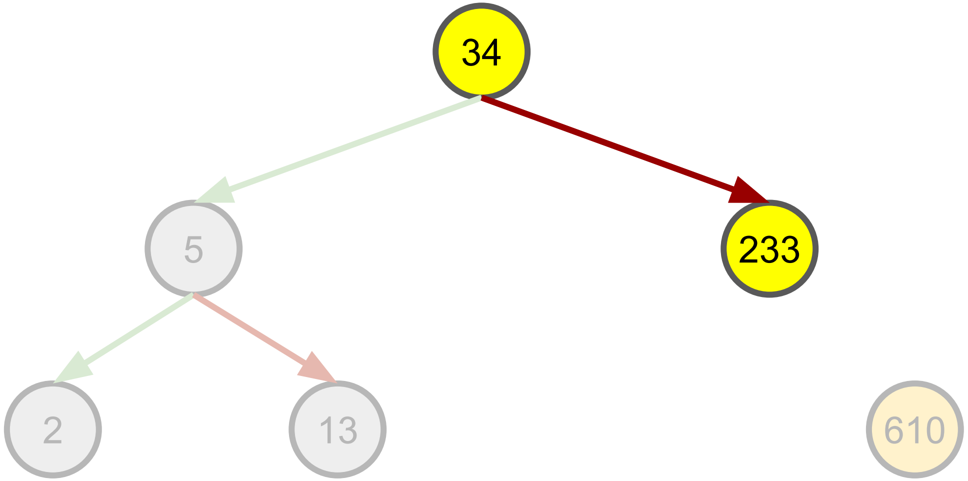


Delete (Case 2)

[Node is at a leaf]

```
delete(610)
```

```
Just remove! Nothing wrong because it doesn’t
break the BST property
```

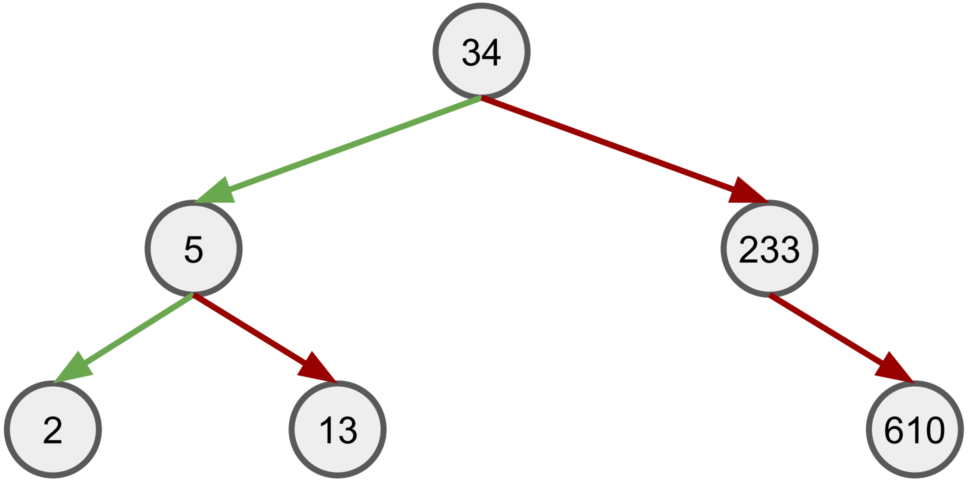


Delete (Case 3)

[Node has 1 child]

delete(233)

Initialisation

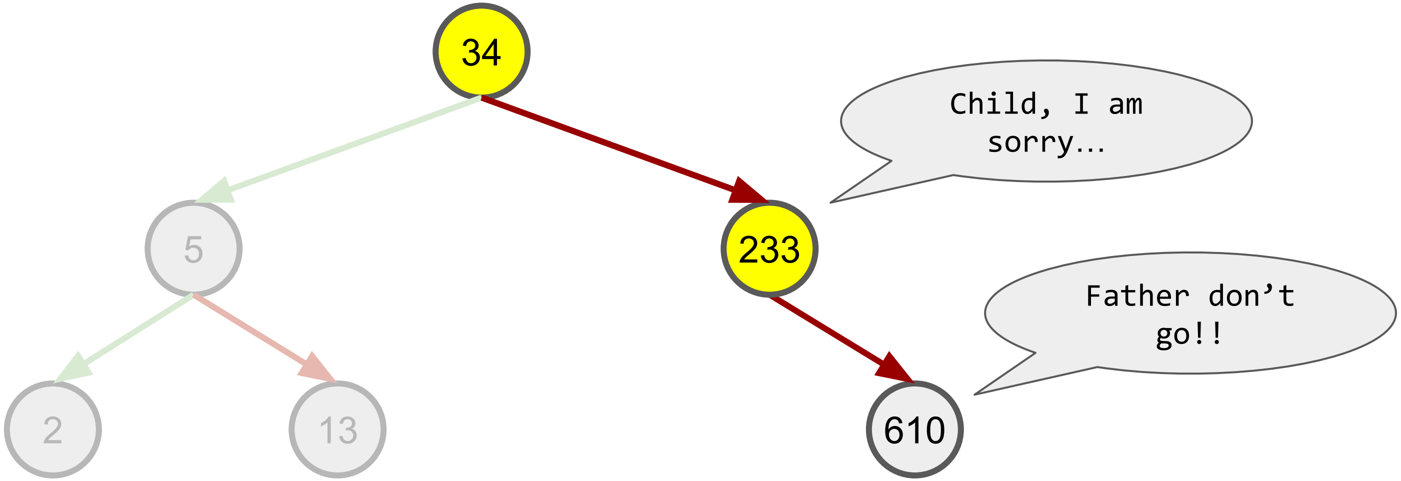


Delete (Case 3)

[Node has 1 child]

delete(233)

Gonna skip the “searching” again. Now if you directly remove this node, there will be orphans :O wat to do

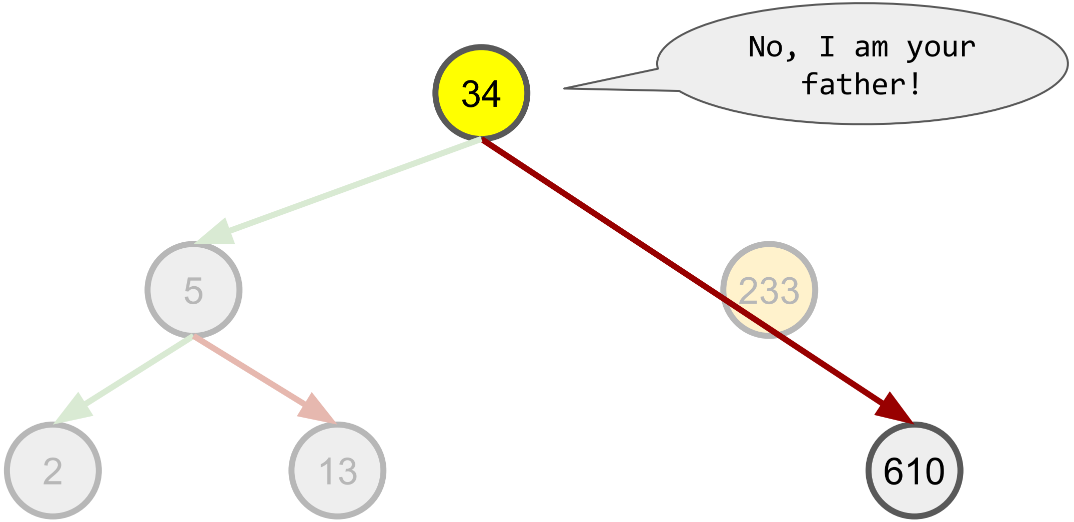


Delete (Case 3)

[Node has 1 child]

delete(233)

Let the grandpa adopt the poor child!

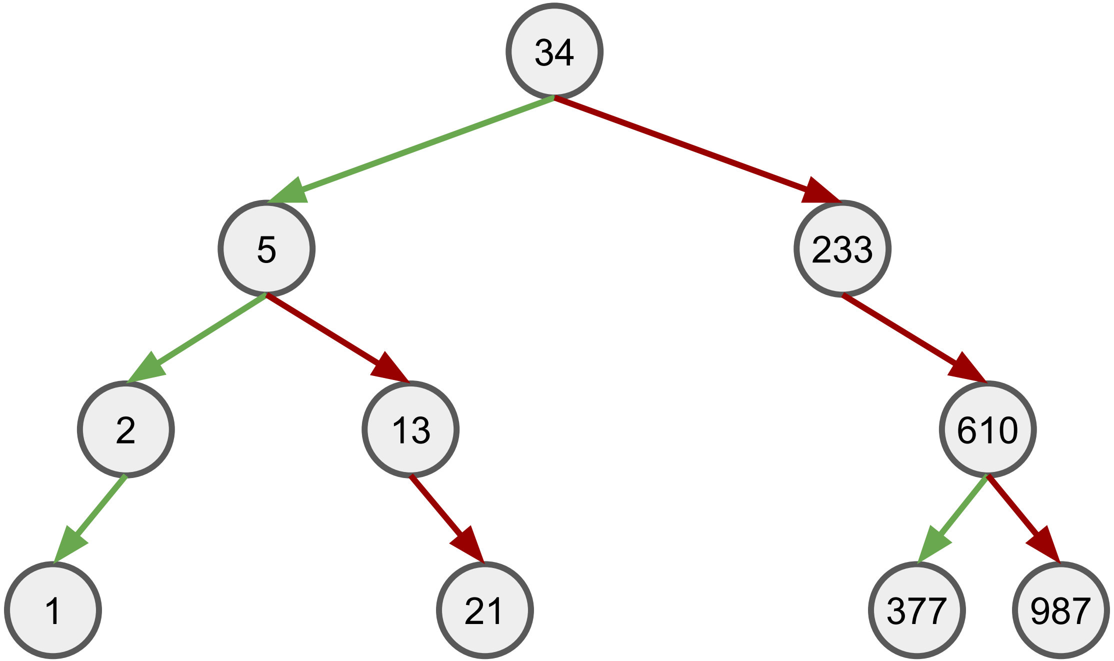


Delete (Case 4)

[Node has 2 children]

delete(5)

Initialisation

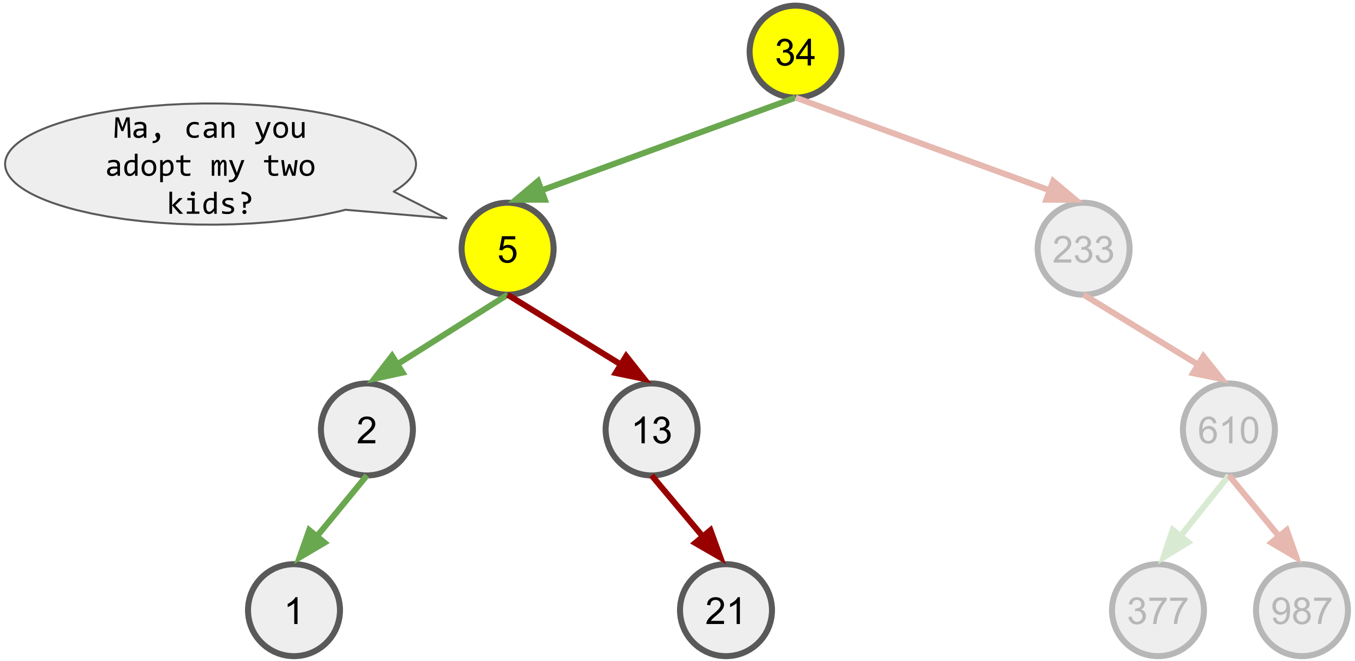


Delete (Case 4)

[Node has 2 children]

delete(5)

Skipping the search visualisation again… If we were to delete this node directly, there will be 2 orphans!

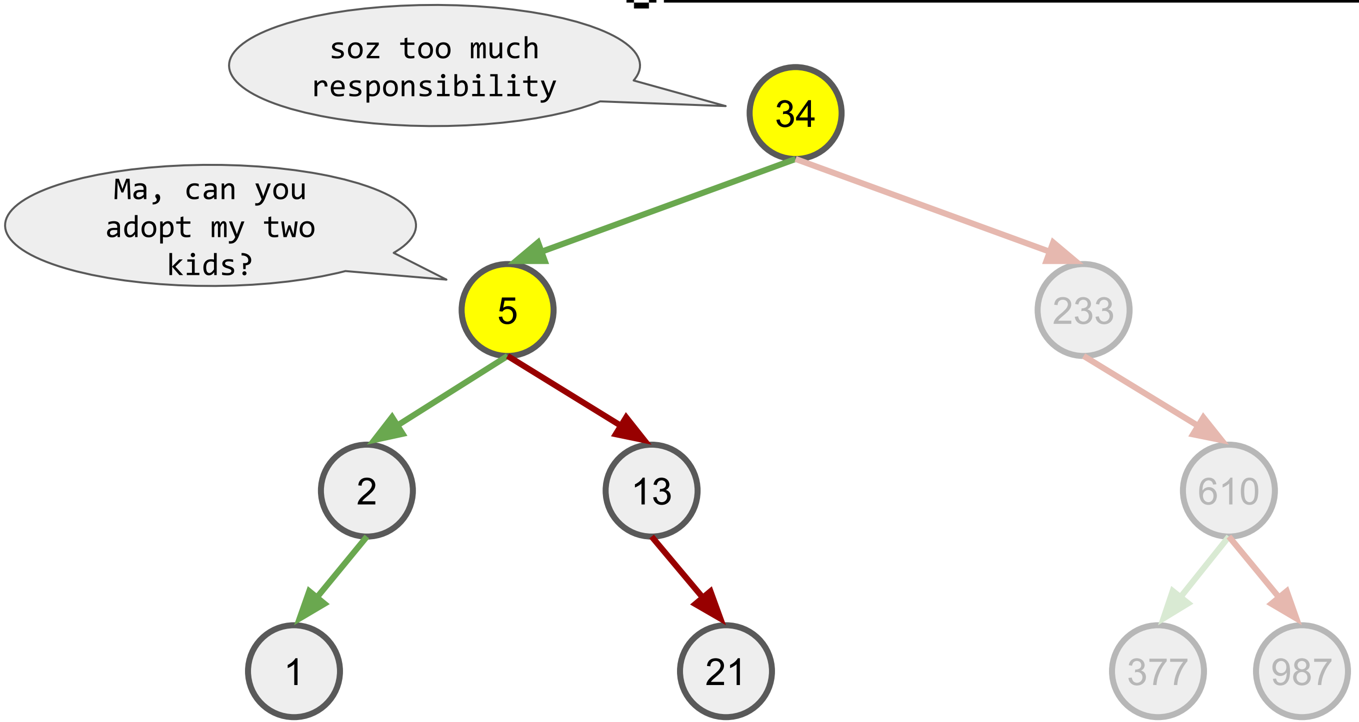


Delete (Case 4)

[Node has 2 children]

delete(5)

Adopting 2 children is a bit too much for the grandparent :(how how

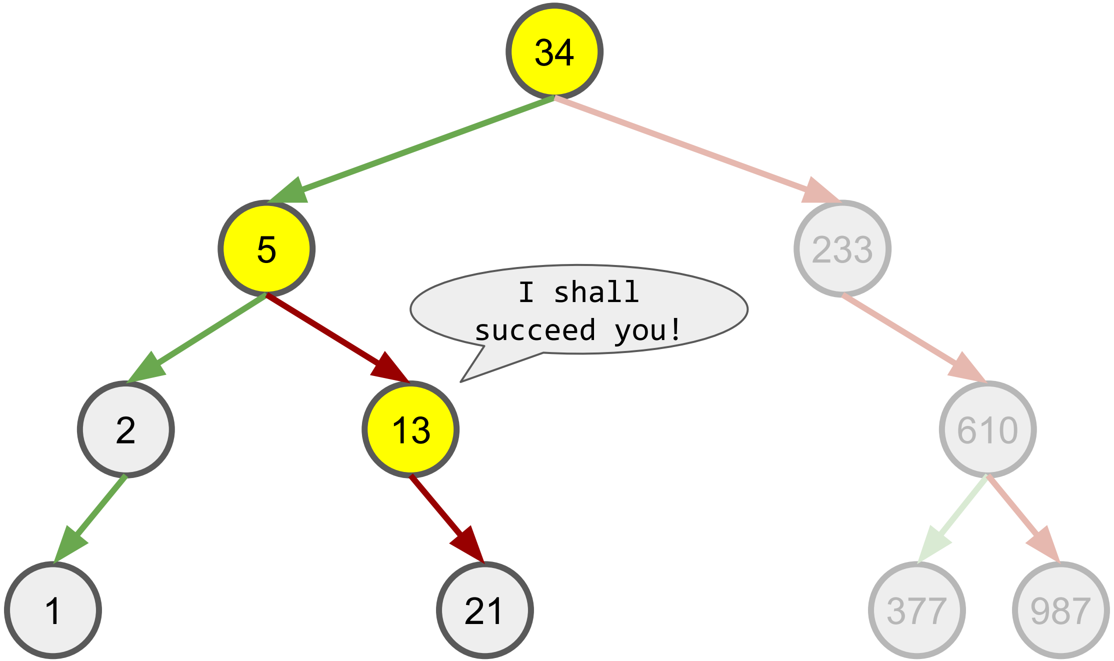


Delete (Case 4)

[Node has 2 children]

```
delete(5)
```

```
Solution: Successor shall take over
```

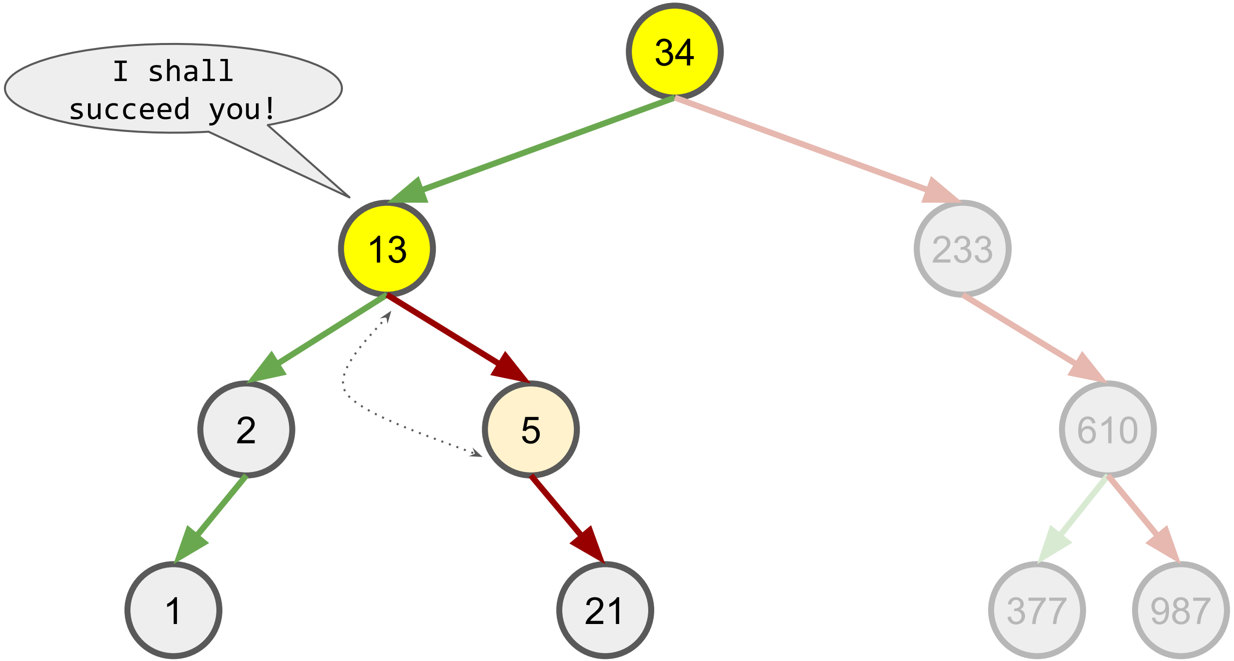


Delete (Case 4)

[Node has 2 children]

```
delete(5)
```

(Note that currently the BST property is violated. We shall fix it soon but how :O)

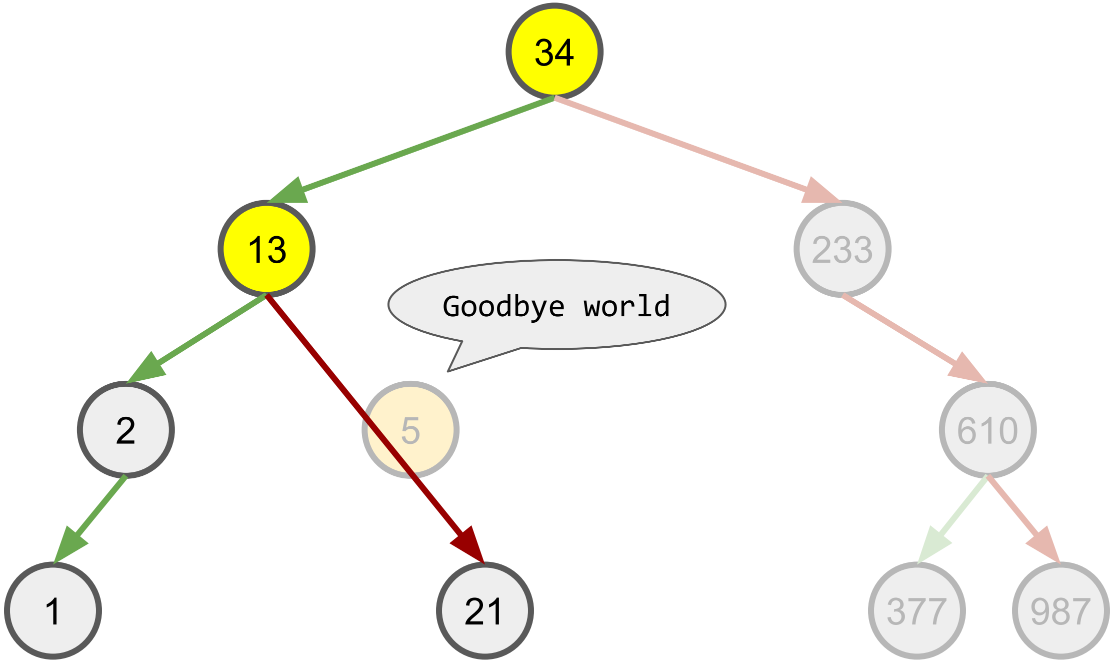


Delete (Case 4)

[Node has 2 children]

delete(5)

Deleting 5 is just the case of deleting node with 1 children!

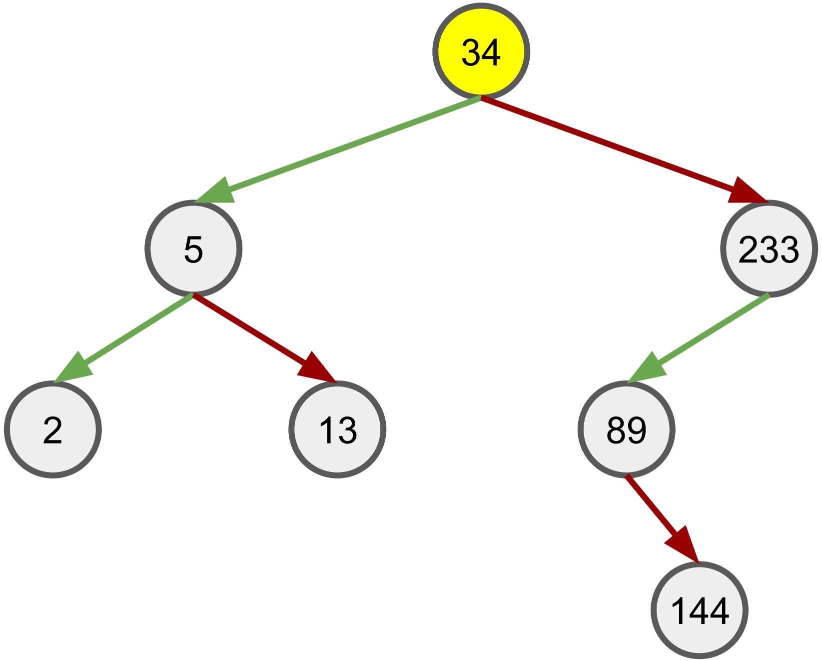


Delete (Case 4)

[Node has 2 children]

delete(34)

Initialisation: Another example for the case of node with 2 children

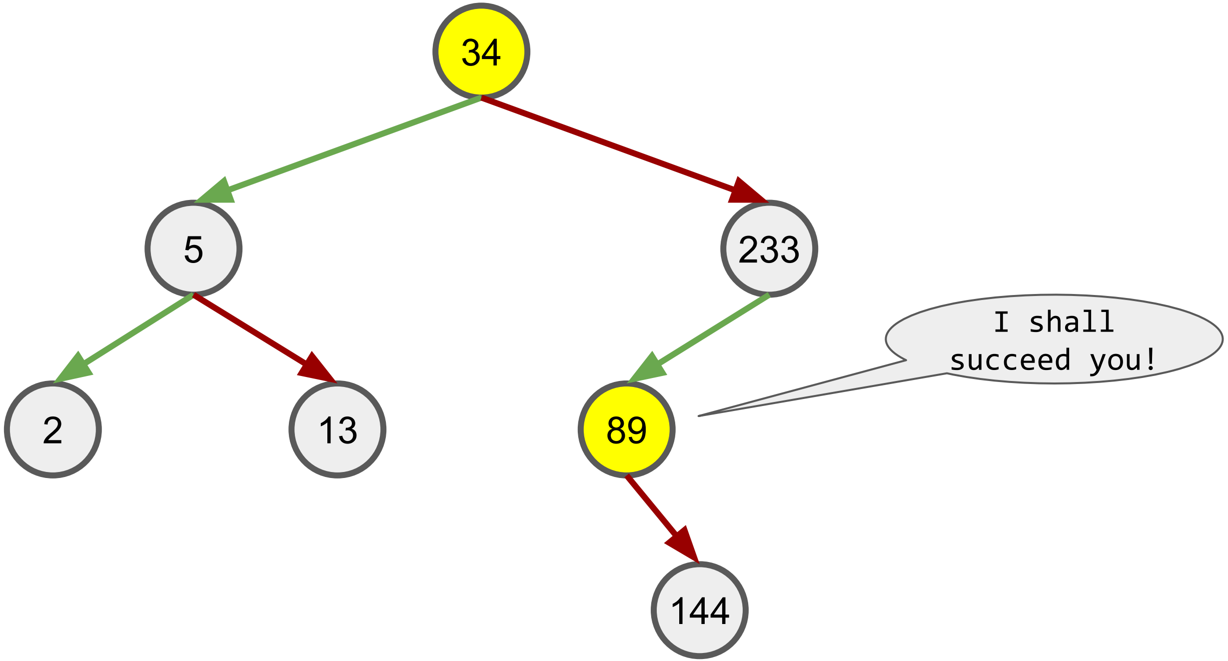


Delete (Case 4)

[Node has 2 children]

delete(34)

Successor will take over

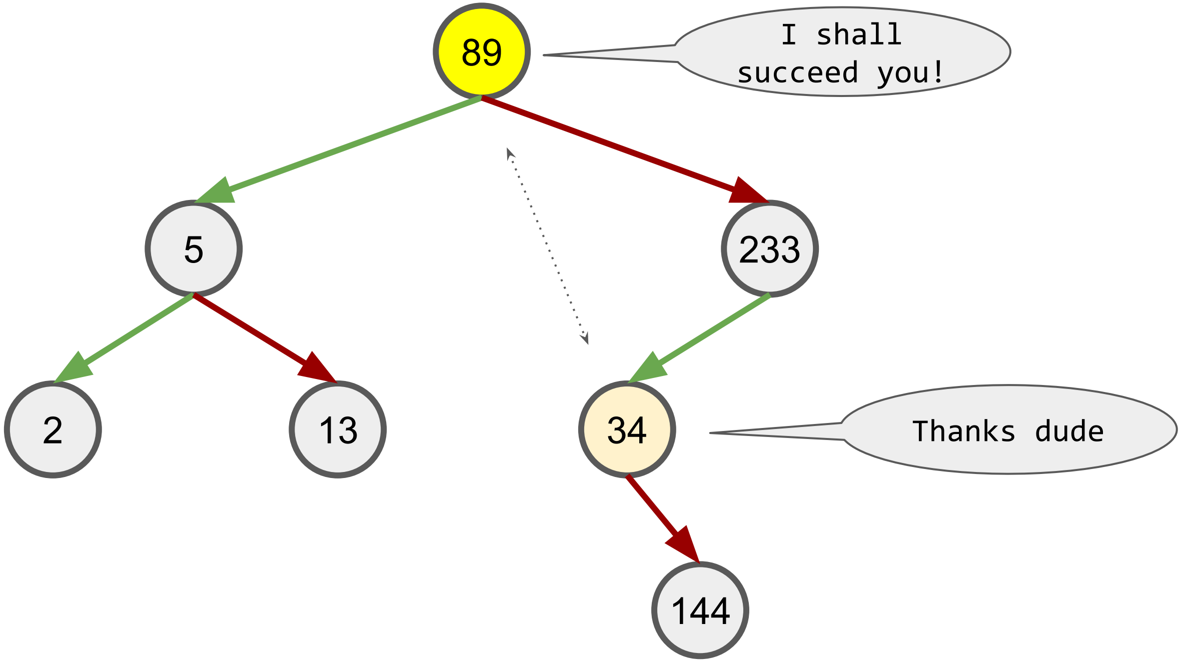


Delete (Case 4)

[Node has 2 children]

delete(34)

Swap anddddd

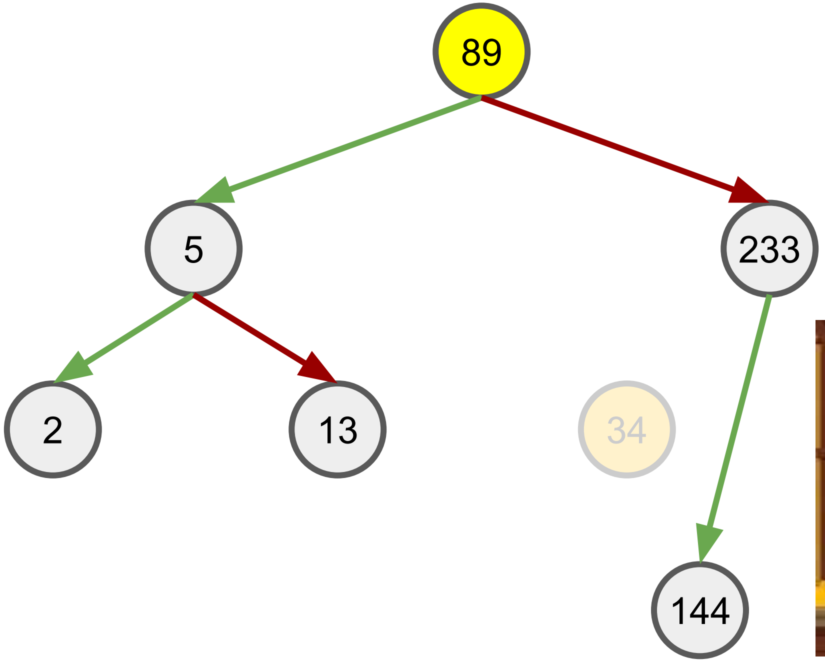


Delete (Case 4)

[Node has 2 children]

delete(34)

it’s gone



Deletion: Why does replacing with successor work?

We need to maintain the following:

Deletion: Why does replacing with successor work?

We need to maintain the following:

1. BST property is not violated when we swap
2. Successor has *at most* 1 child (why?)

Deletion: Why does replacing with successor work?

We need to maintain the following:

1. BST property is not violated when we swap
2. Successor has *at most* 1 child (because if 1 child, can just let grandparent adopt the child. If no child, can simply delete)

Questions?

Traversals

Different types of traversal:

- preorder:
- inorder:
- postorder:

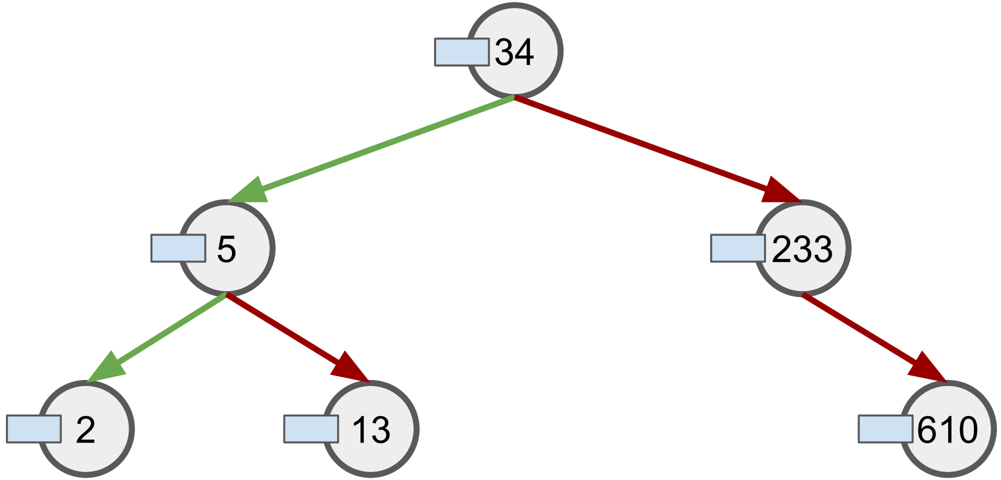
Traversals

Different types of traversal:

- preorder: *print(root) traverse(left) traverse(right)*
- inorder: *traverse(left) print(root) traverse(right)*
- postorder: *traverse(left) traverse(right) print(root)*

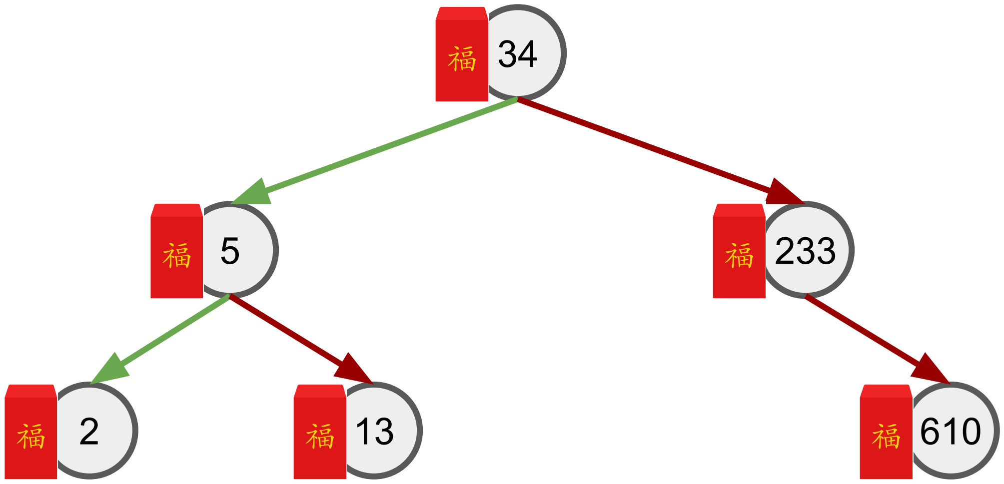
Pre-order Traversal

Lifehack: Put these “markers” at the left (pre)



Pre-order Traversal

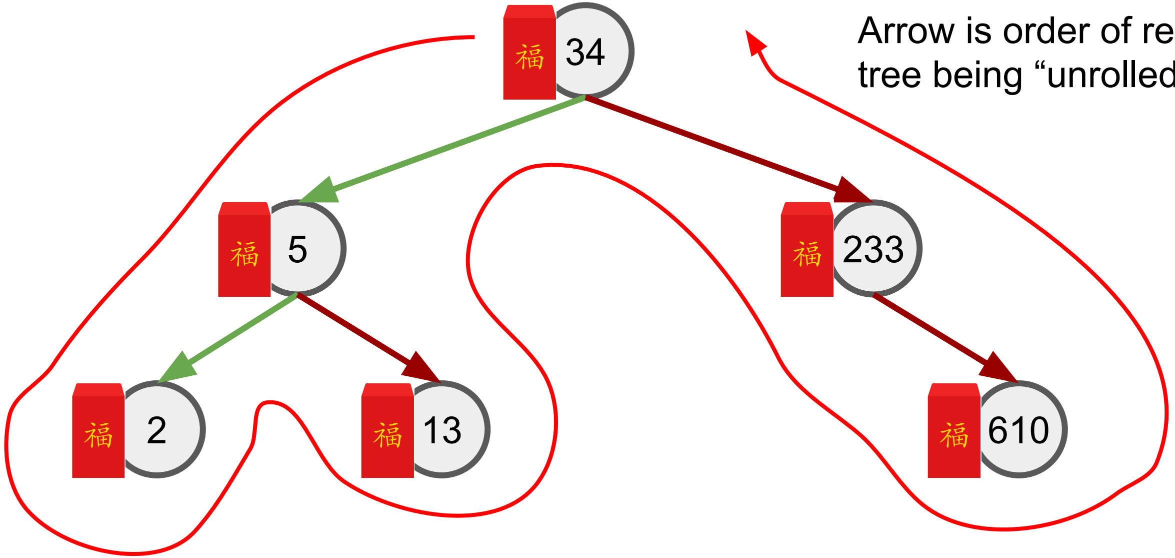
Lifehack: Put these hongbaos at the left (pre)



Pre-order Traversal

Lifehack: Put these hongbaos at the left (pre)

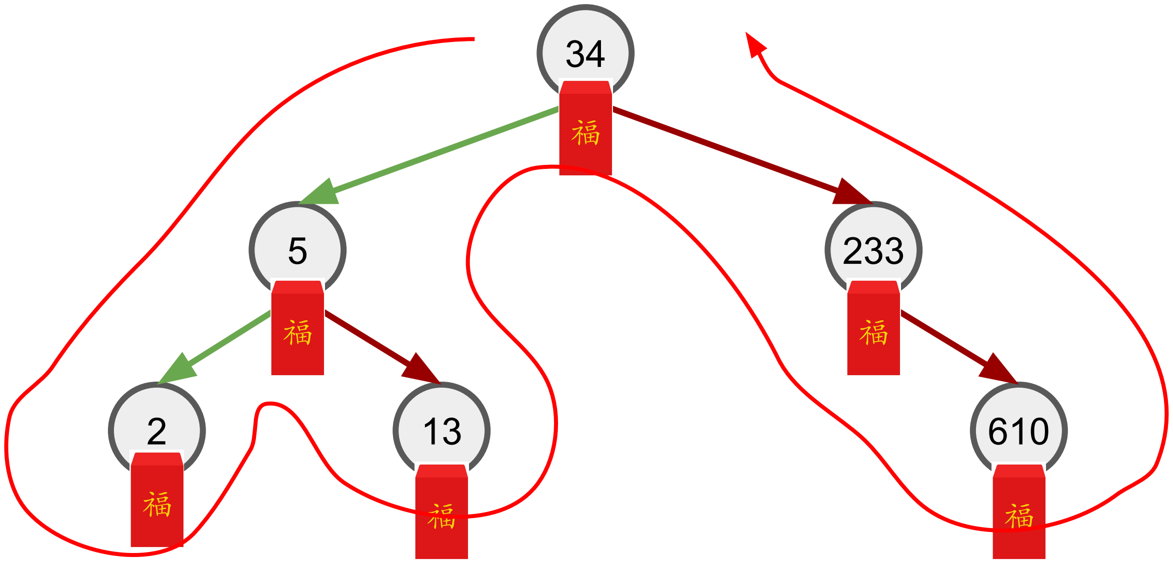
Arrow is order of recursion tree being "unrolled"



Order of hongbao collection: 34 5 2 13 233 610

In-order Traversal

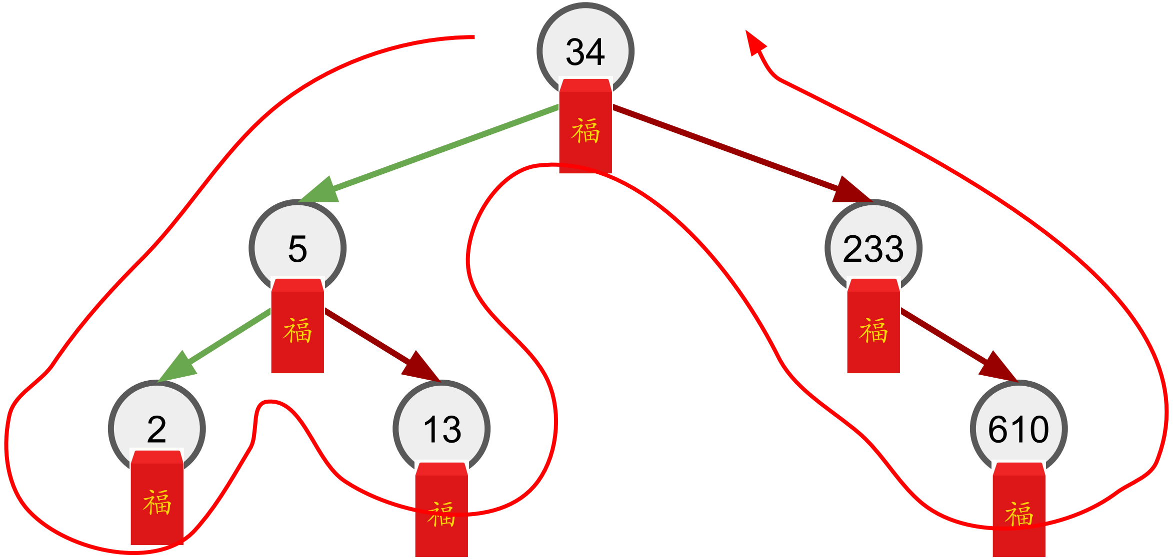
Lifehack: Put these hongbaos at the bottom (in)



Order of hongbao collection: ???

In-order Traversal

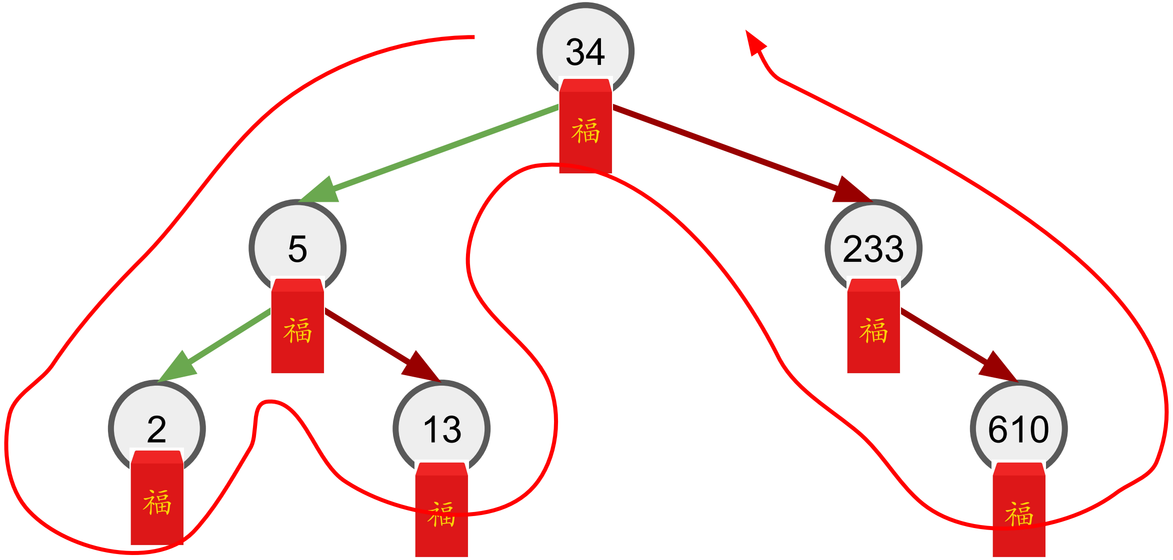
Lifehack: Put these hongbaos at the bottom (in)



Order of hongbao collection: 2 5 13 34 233 610

In-order Traversal

Lifehack: Put these hongbaos at the bottom (in)

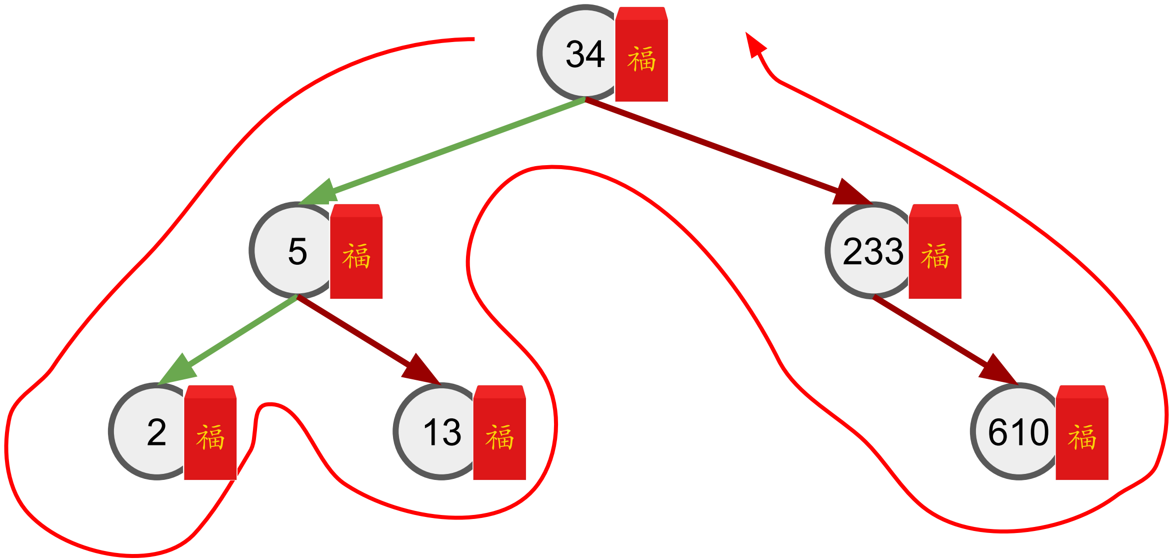


Order of hongbao collection: 2 5 13 34 233 610

Cool stuff: if your tree is BST, the inorder traversal appears in sorted order!

Post-order Traversal

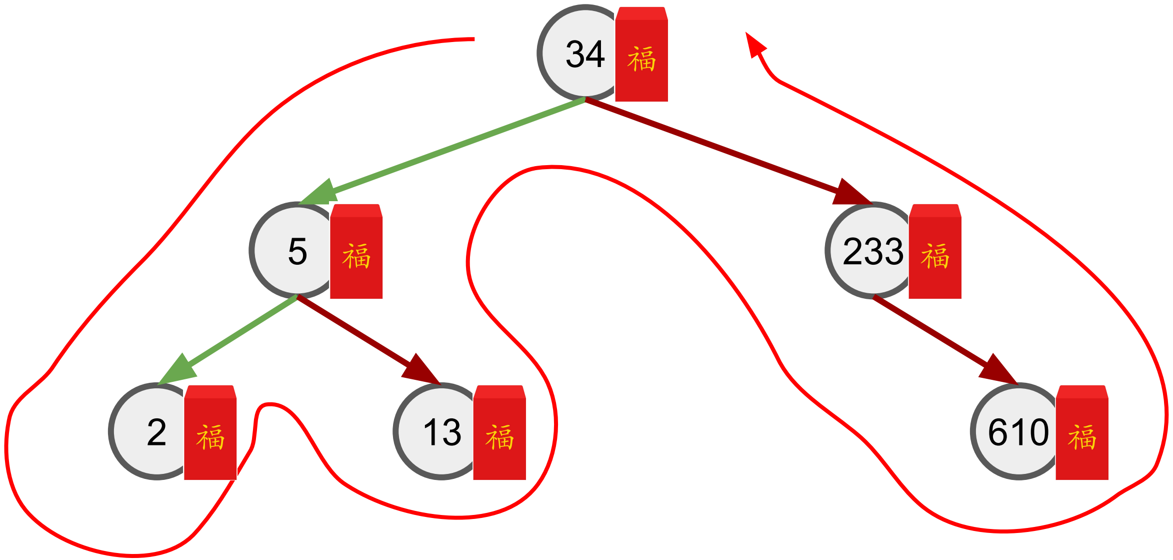
Lifehack: Put these hongbaos at the right (post)



Order of hongbao collection: ???

Post-order Traversal

Lifehack: Put these hongbaos at the right (post)



Order of hongbao collection: 2 13 5 610 233 34

Time to traverse?

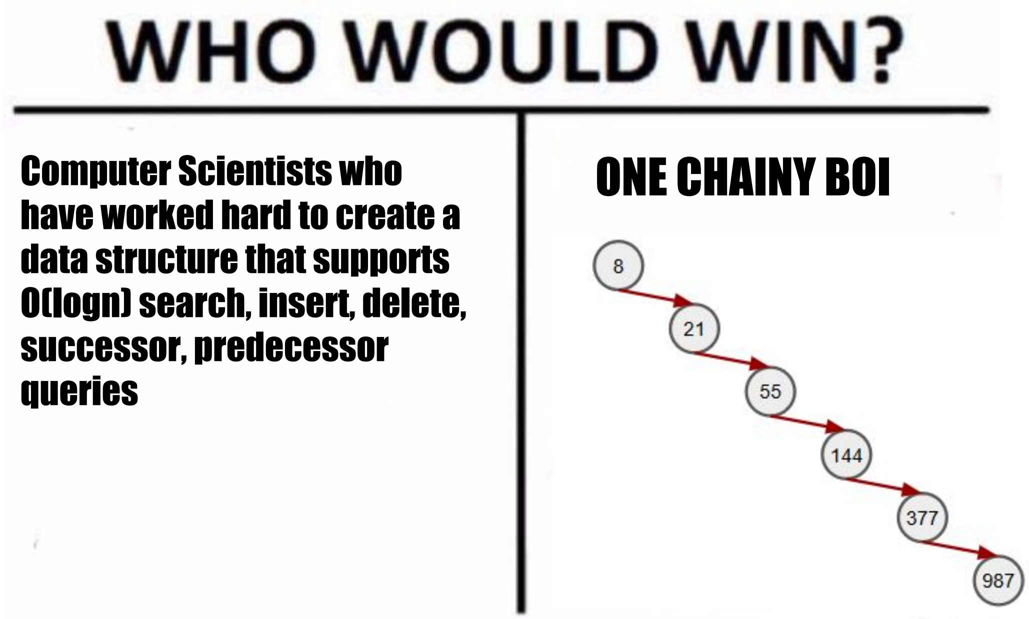
Time to traverse?

- Intuitively, you are going through the entire tree.
- Therefore it is $O(n)$ time

Time-complexity of BST operations

Time-complexity of BST operations

- Most of these operations are *O(h)* time where *h* is the height of the BST
- NOT necessarily *O(logn)* time where *n* is the number of elements in the tree



Enter the AVL trees!

Enter the AVL trees!

- Named after Adelson-Velsky and Landis (two people not three)
- Idea: Since the time-complexities of most operations in BST are *O(h)*, let's find a way to **bound *h* by *logn***!

- This is the concept of **height-balanced trees**. AVL tree is not the only way we can achieve this. Other trees such as Red-Black trees, B-Trees, Splay Trees exist.

How to AVL tree

- Every node contains the variable for **height**
  - `height = max(left.height, right.height) + 1`

How to AVL tree

- Every node contains the variable for **height**
  - `height = max(left.height, right.height) + 1`
- Invariant for height-balancing: For every node, the height of their children differ by **at most 1**.

How to AVL tree

- Every node contains the variable for **height**
    - `height = max(left.height, right.height) + 1`
- Invariant for height-balancing: For every node, the height of their children differ by **at most 1**.
- If this particular invariant is broken, then the tree is not an AVL-tree anymore

Rotations

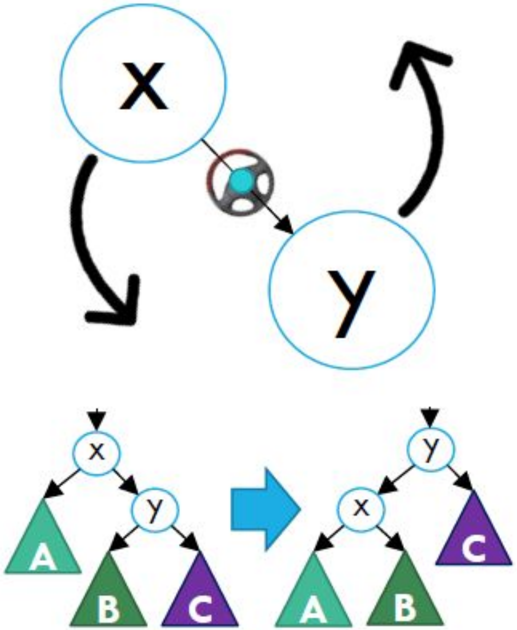
Rotations

- How we achieve balance!
- Different kinds: left rotate, right rotate, left-left rotate, right-right rotate
- **IMPORTANT:** Has to preserve the BST properties!

Taken from Prof Harold’s AY19/20 S1 slides (with permission)

HOW I REMEMBER IT

left / anti-clockwise

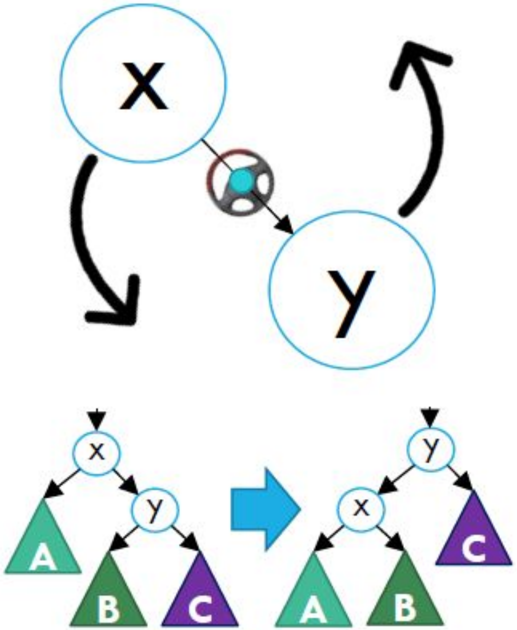


This tree is right-heavy! Position the steering wheel like this

Taken from Prof Harold's AY19/20 S1 slides (with permission)

HOW I REMEMBER IT

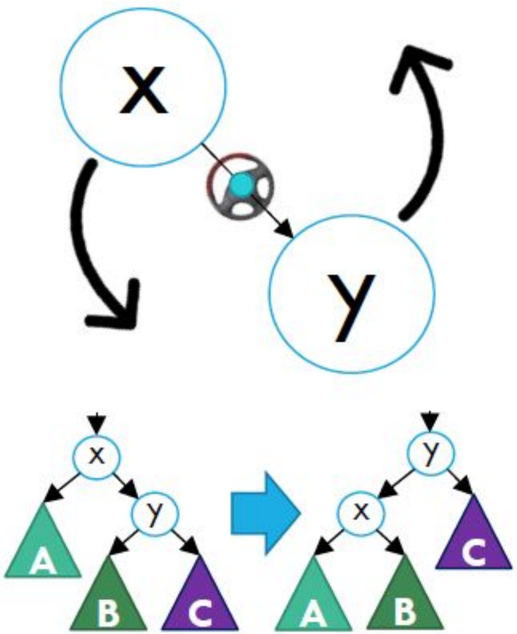
left / anti-clockwise



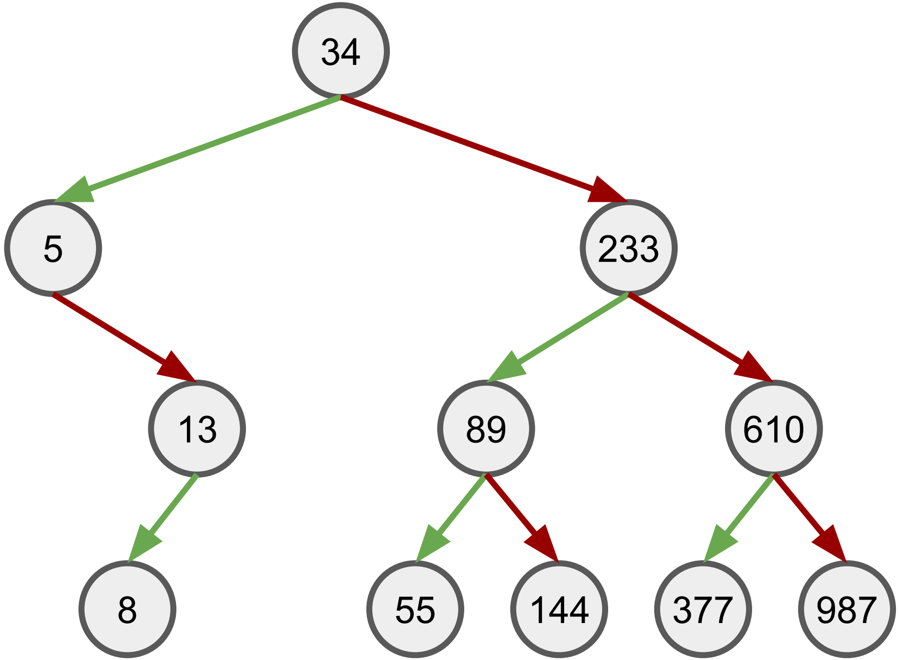
Taken from Prof Harold's AY19/20 S1 slides (with permission)

HOW I REMEMBER IT

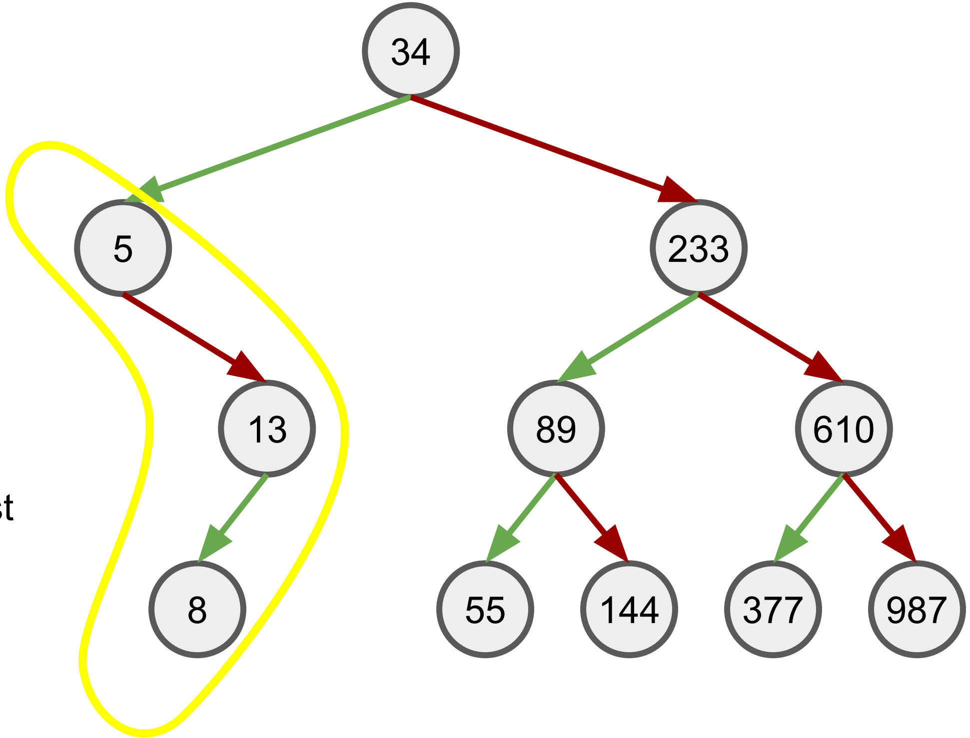
left / anti-clockwise



Imbalanced?? How should we rotate this?



Imbalanced?? How should we rotate this?

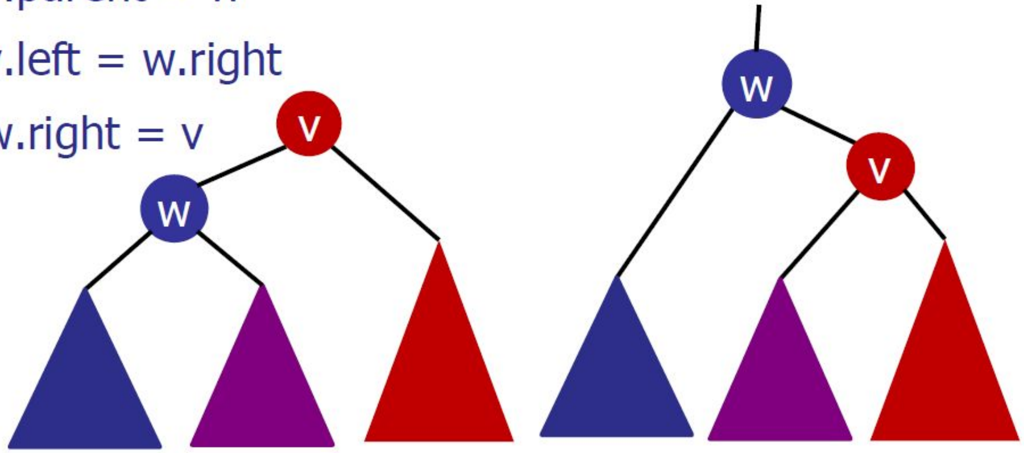


The zigzag pattern creates trouble!

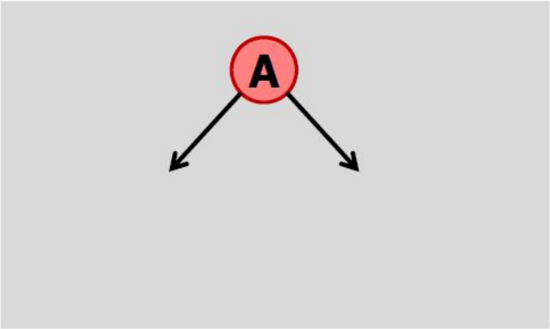
If you see zigzag, most likely need to double rotate

Rotations

```
right-rotate(v)          // assume v has left != null
    w = v.left
    w.parent = v.parent
    v.parent = w
    v.left = w.right
    w.right = v
```



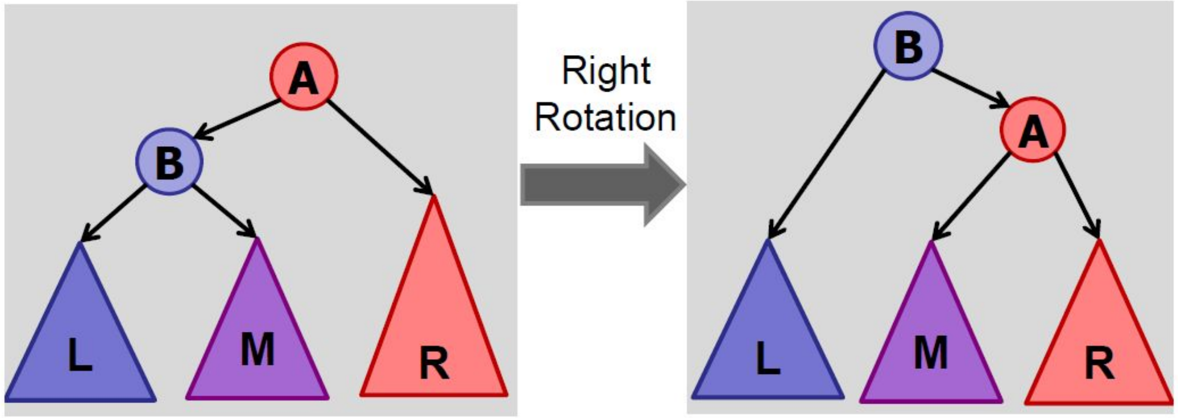
Tree Rotations



A is **LEFT-heavy** if left sub-tree has larger height than right sub-tree.

A is **RIGHT-heavy** if right sub-tree has larger height than left sub-tree.

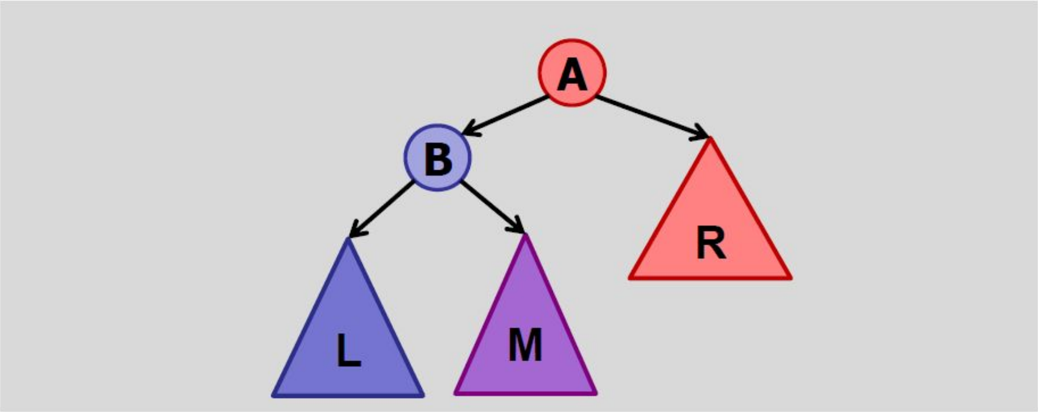
Tree Rotations



Use tree rotations to restore balance.

After insert, start at bottom, work your way up.

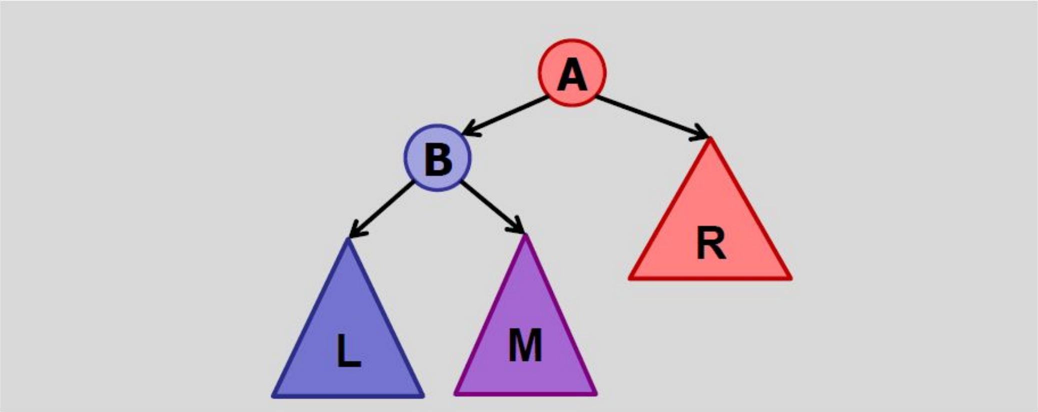
Tree Rotations



Assume **A** is the lowest node in the tree violating balance property.

Assume A is **LEFT-heavy**.

Tree Rotations (Left Heavy)

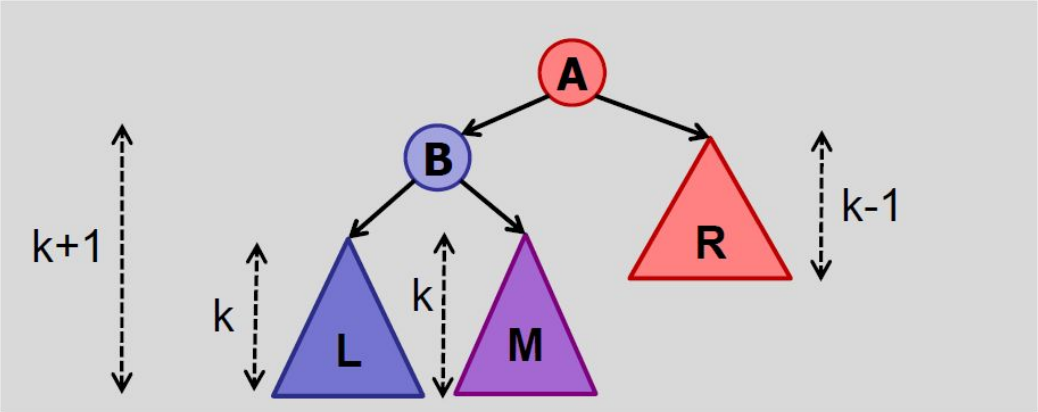


Assume **A** is the lowest node in the tree violating balance property.

Case 1: **B** is balanced : h(**L**) = h(**M**)

h(**R**) = h(**B**) – 2

Tree Rotations (Left Heavy)

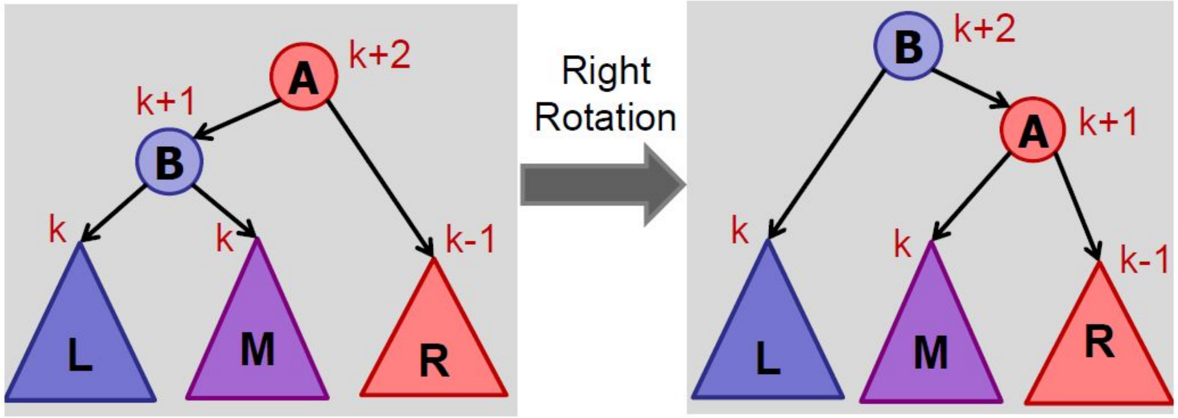


Assume **A** is the lowest node in the tree violating balance property.

Case 1: **B** is balanced : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations

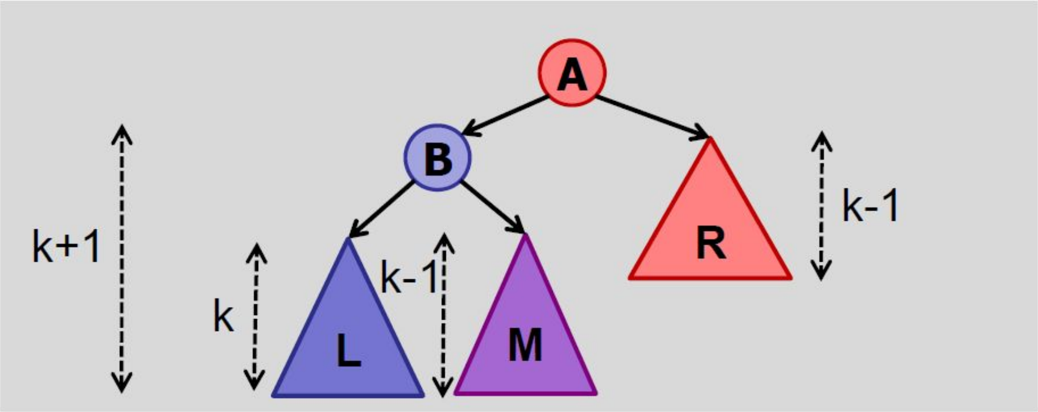


right-rotate:

Case 1: **B** is balanced : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations (Left Heavy)

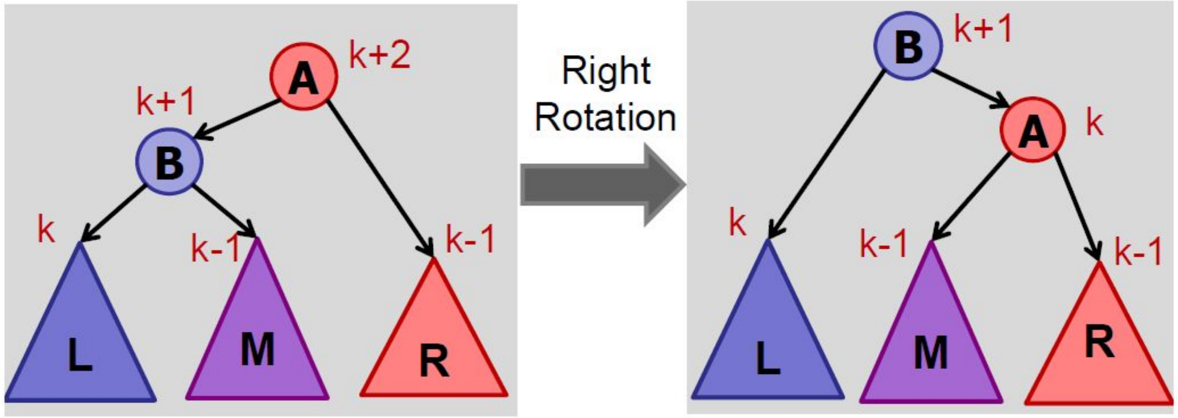


Assume **A** is the lowest node in the tree violating balance property.

Case 2: **B** is left-heavy : h(**L**) = h(**M**) + 1

h(**R**) = h(**M**)

Tree Rotations

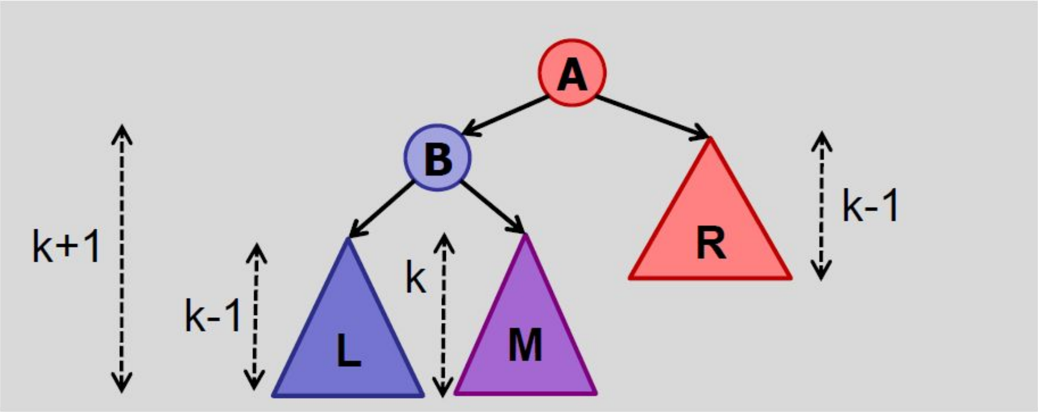


right-rotate:

Case 2: **B** is left-heavy: h(**L**) = h(**M**) +1

h(**R**) = h(**M**)

Tree Rotations (Left Heavy)

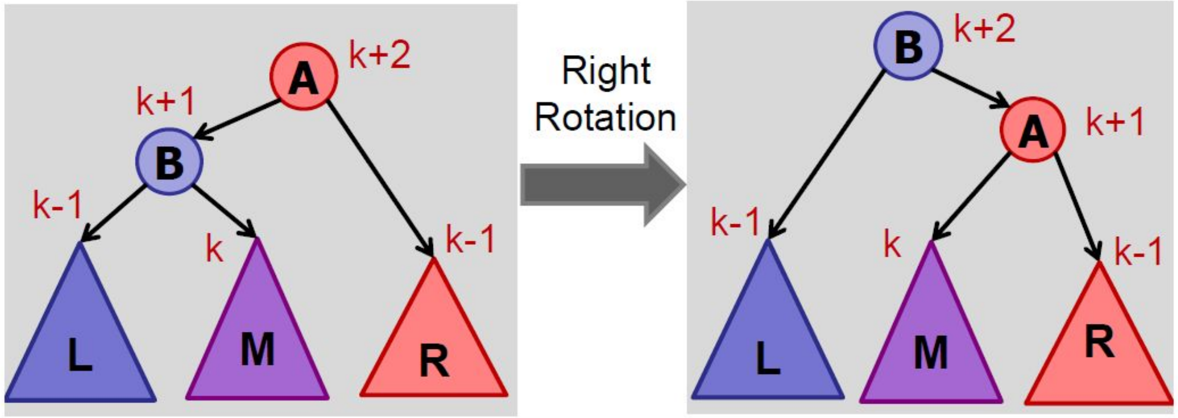


Assume **A** is the lowest node in the tree violating balance property.

Case 3: **B** is right-heavy : h(**L**) = h(**M**) - 1

h(**R**) = h(**L**)

Tree Rotations

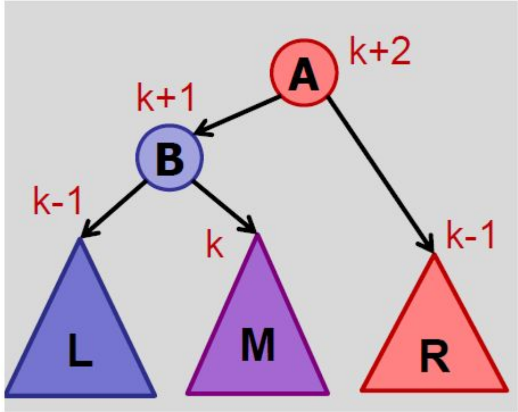


right-rotate:

Case 3: **B** is right-heavy: h(**L**) = h(**M**) – 1

h(**R**) = h(**L**)

Tree Rotations



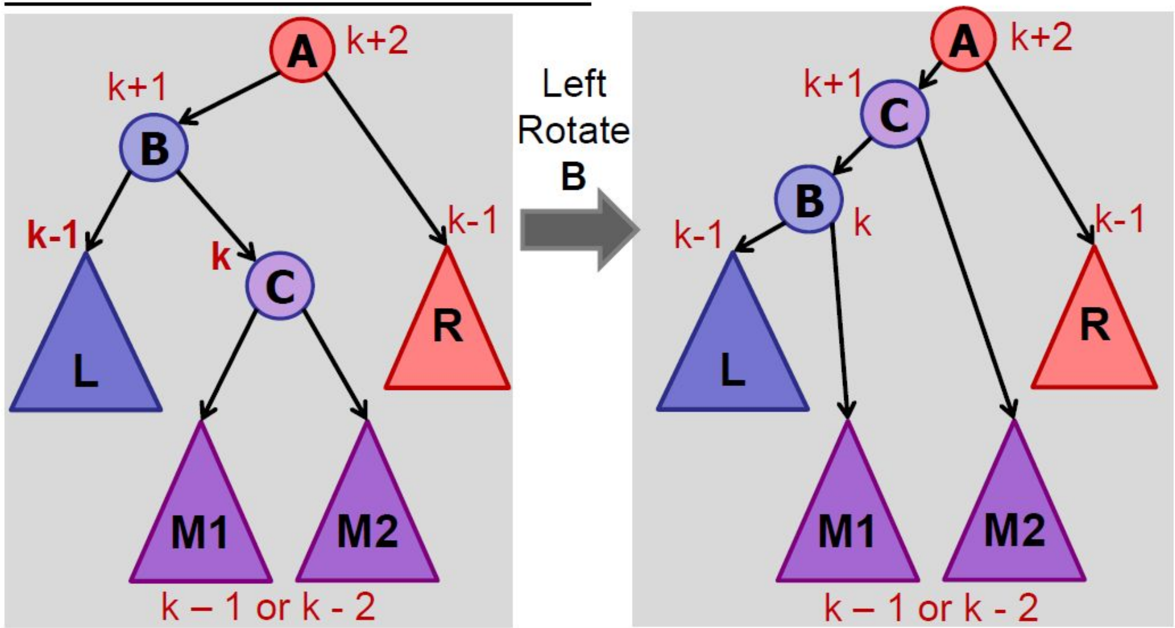
Let's do something first before we right-rotate(A)

right-rotate:

Case 3: **B** is right-heavy: h(**L**) = h(**M**) – 1

h(**R**) = h(**L**)

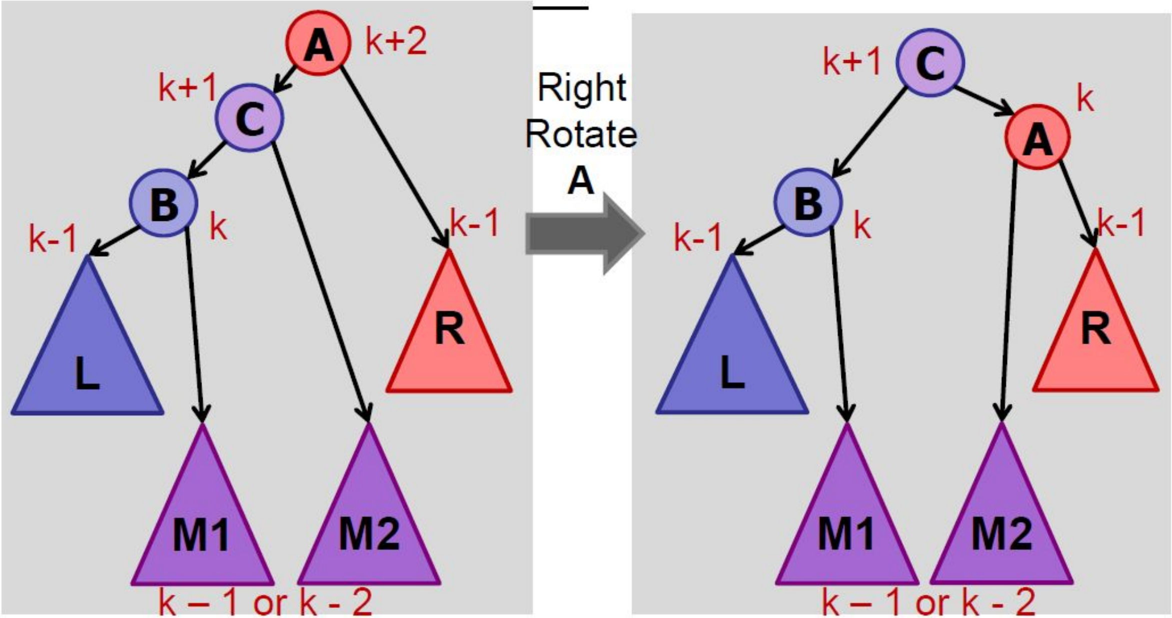
Tree Rotations



Left-rotate B

After left-rotate B: **A** and **C** still out of balance.

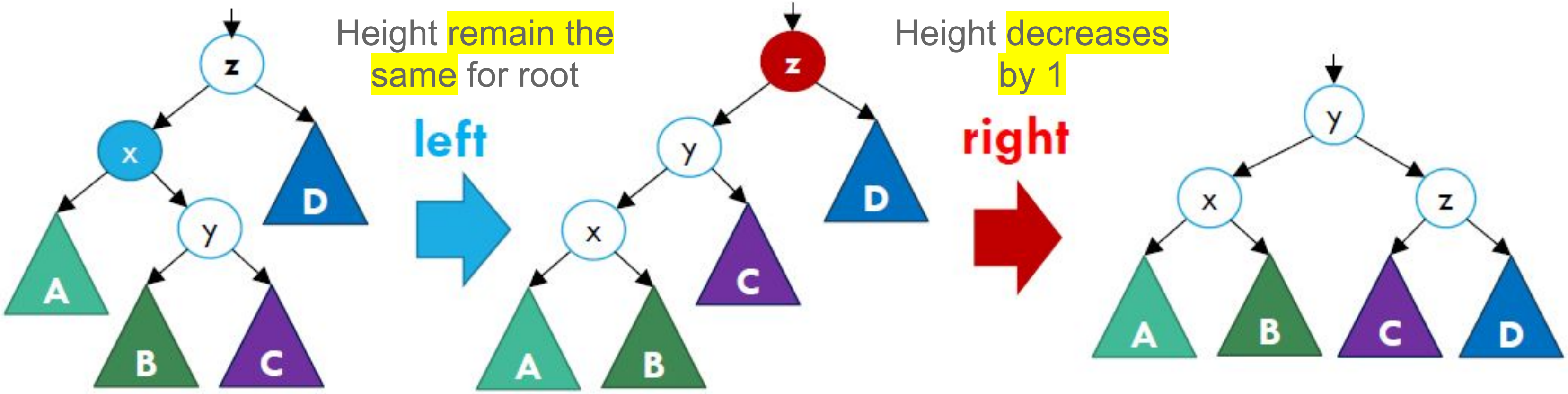
Tree Rotations



After right-rotate A: all in balance.

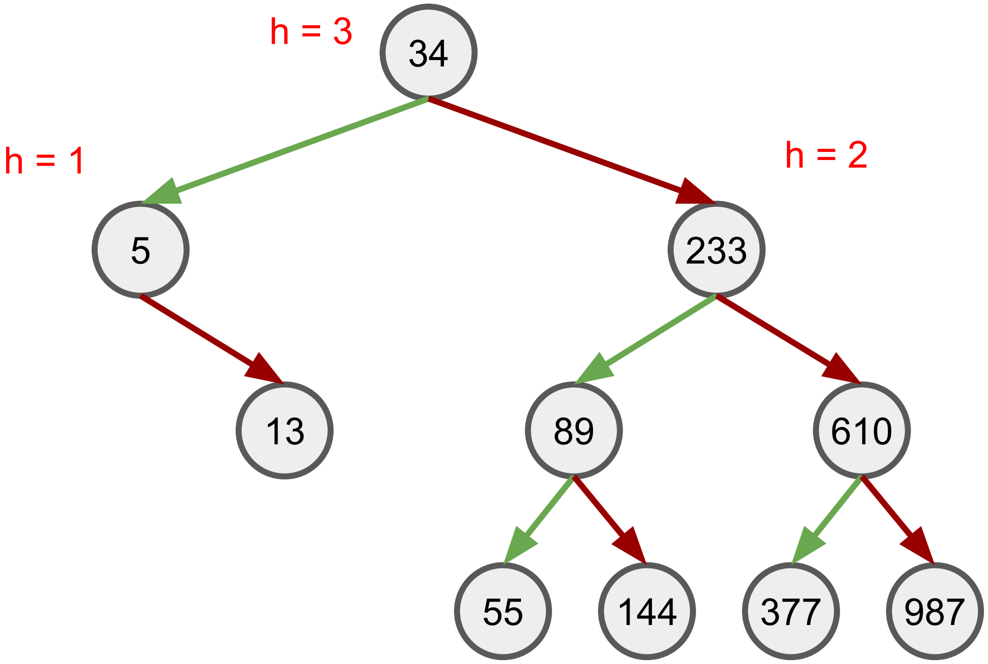
How is height affected after *rotation*?

- The goal of rotation is to *fix* height imbalances!
- Height should either decrease by 1 or remain the same (peek double rotation)



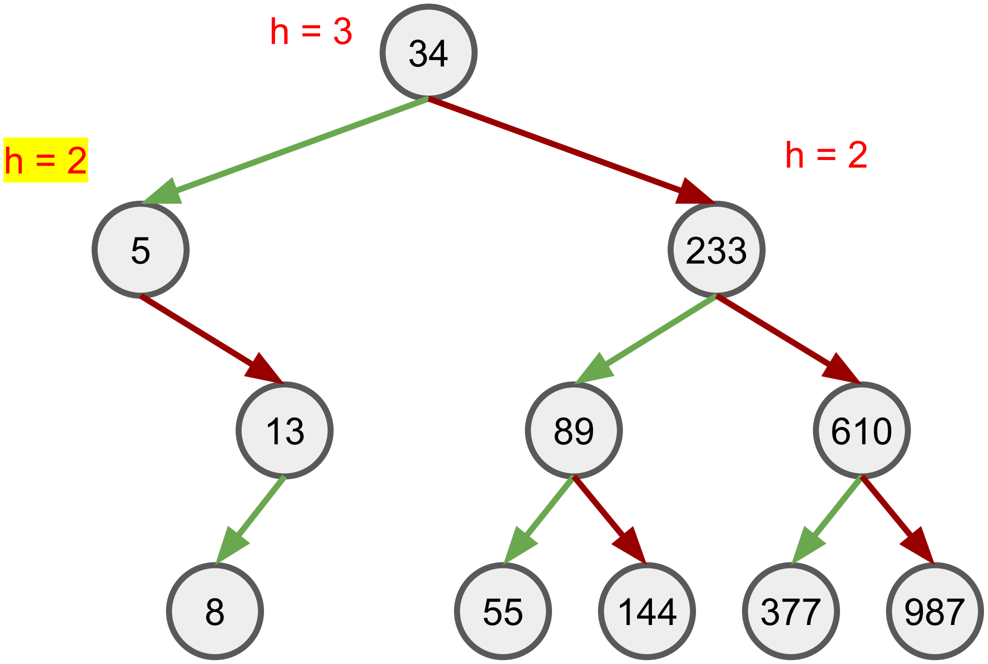
`insert(8)`

Where should 8 be inserted?



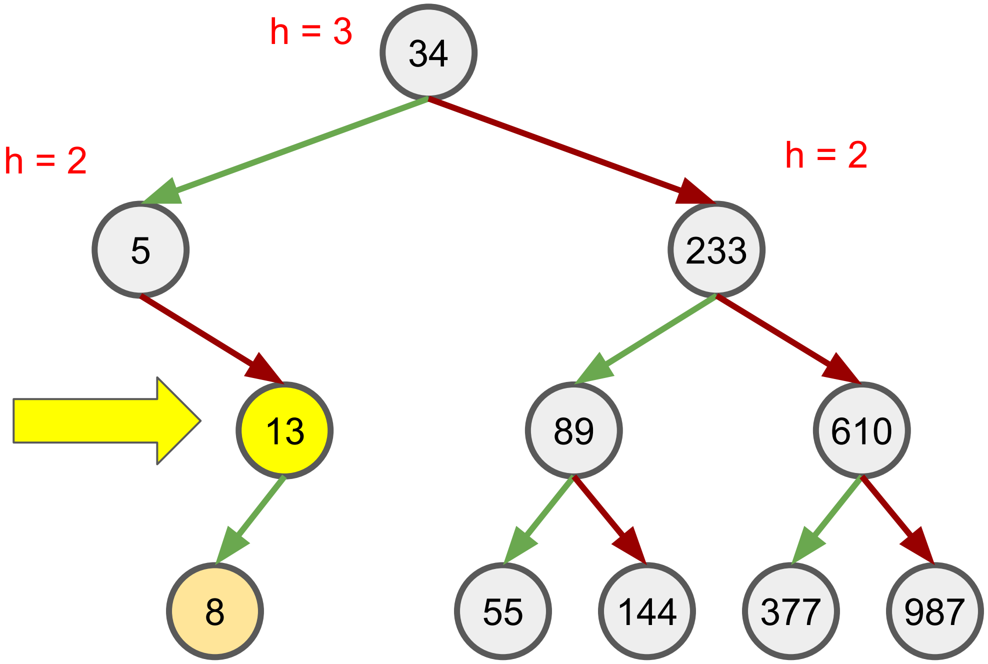
insert(8)

Notice that the height increases for that particular subtree. How should we fix this imbalance?



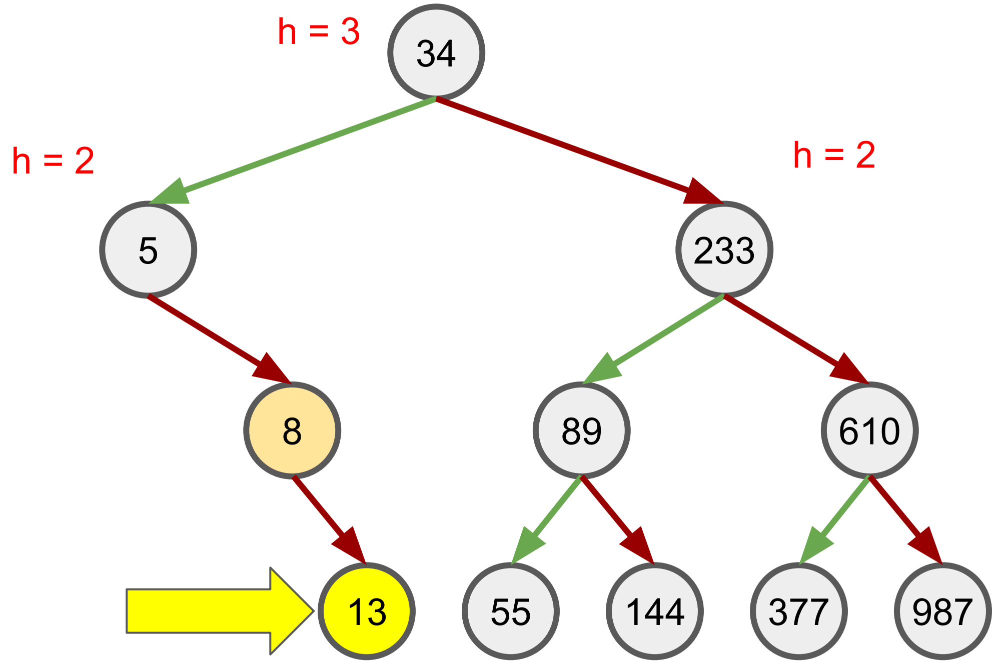
insert(8)

rightRotate(13)



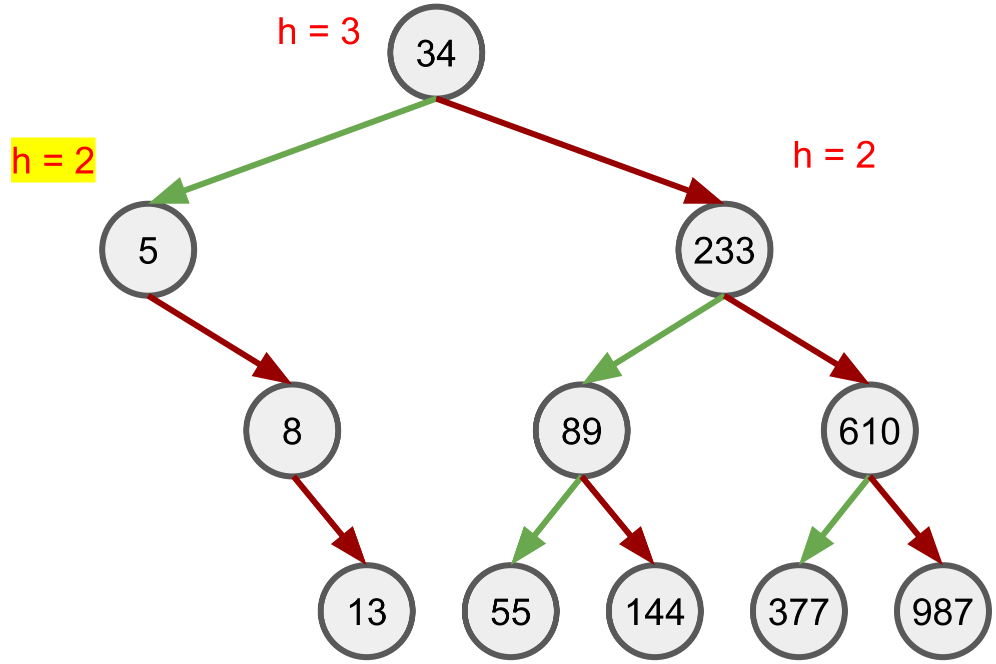
```
insert(8)
```

```
rightRotate(13)
```



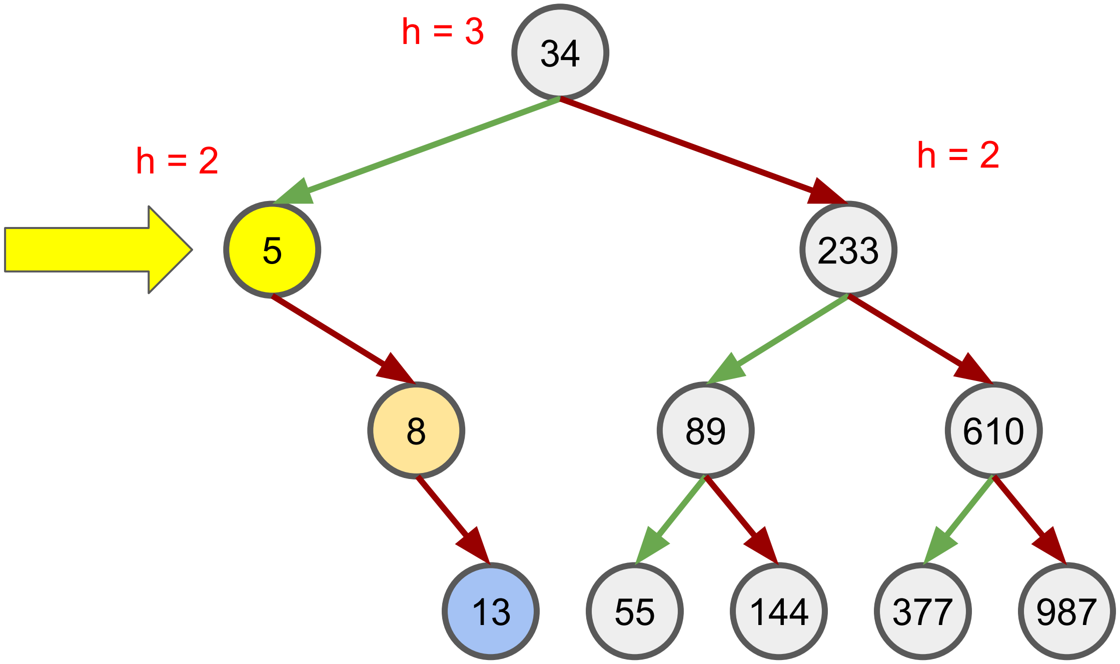
insert(8)

Notice the height didn’t change at all! Where should we perform the second rotation?



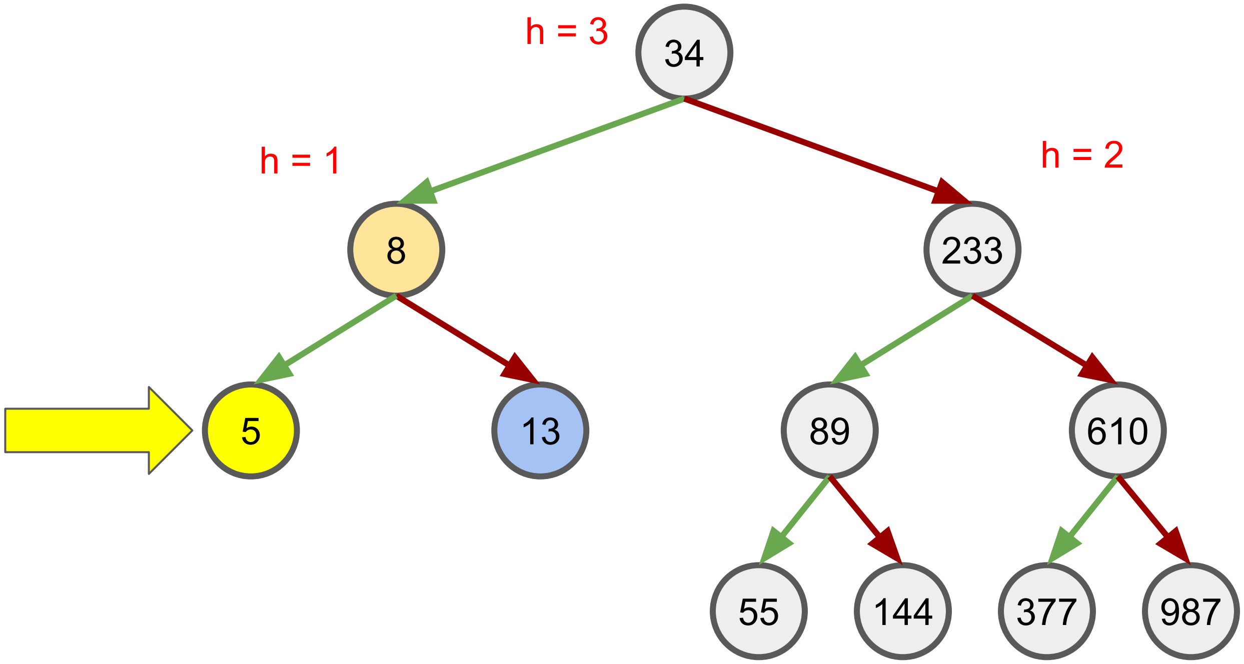
insert(8)

leftRotate(5)



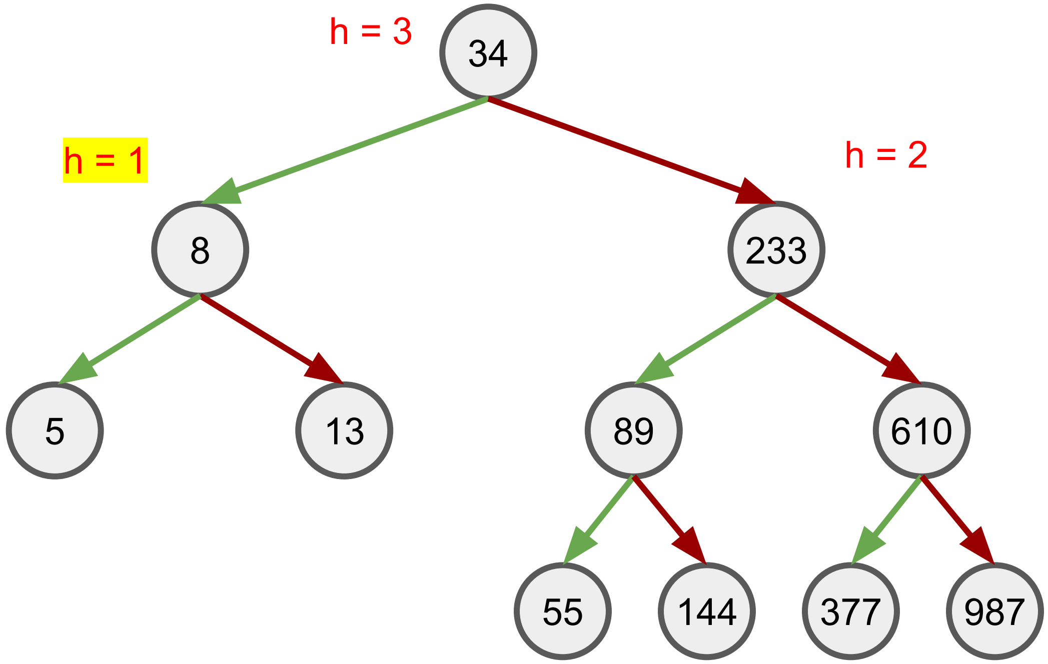
insert(8)

leftRotate(5)



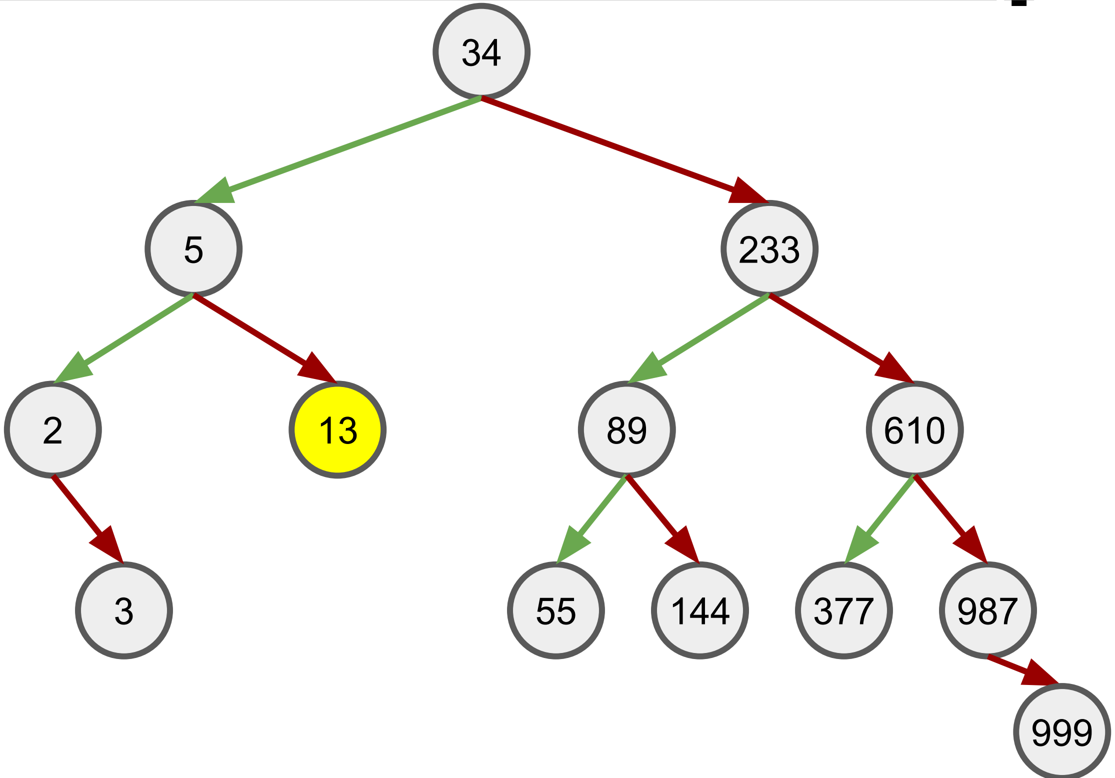
insert(8)

Notice the height of the subtree goes back to 1! It was *initially* 1, then insertion caused it to become 2, but balancings fixed it back to 1



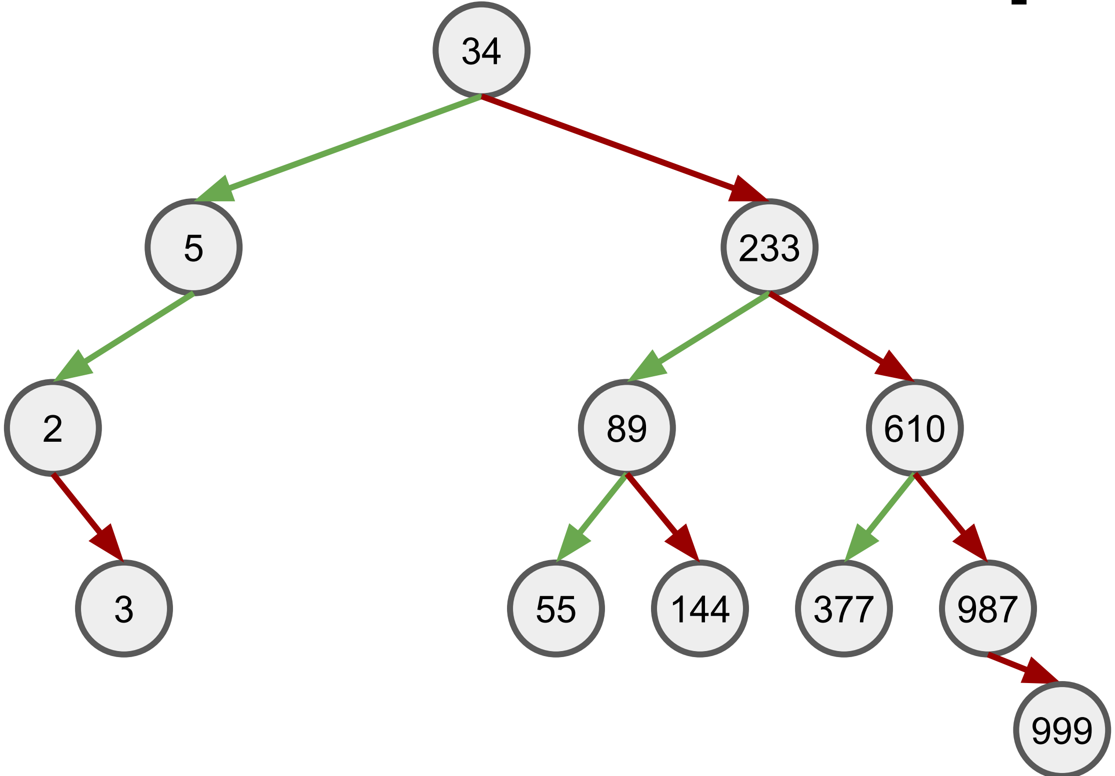
delete(13)

What happens after this deletion?



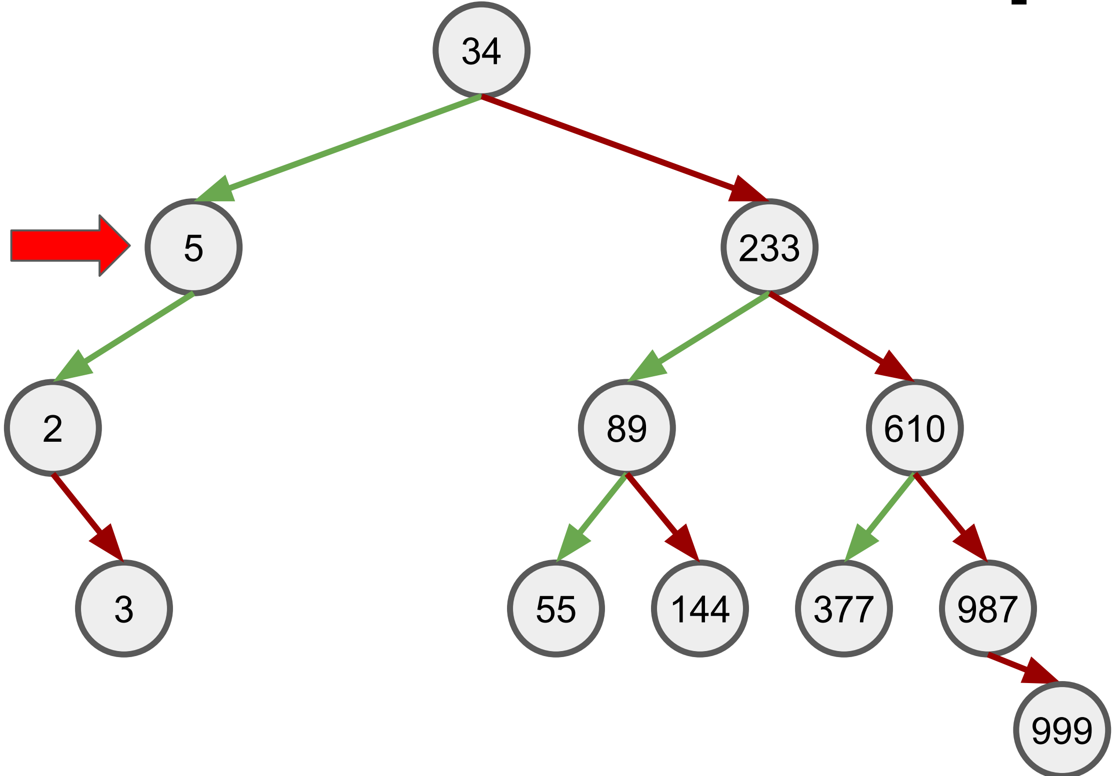
delete(13)

Is there an imbalance?



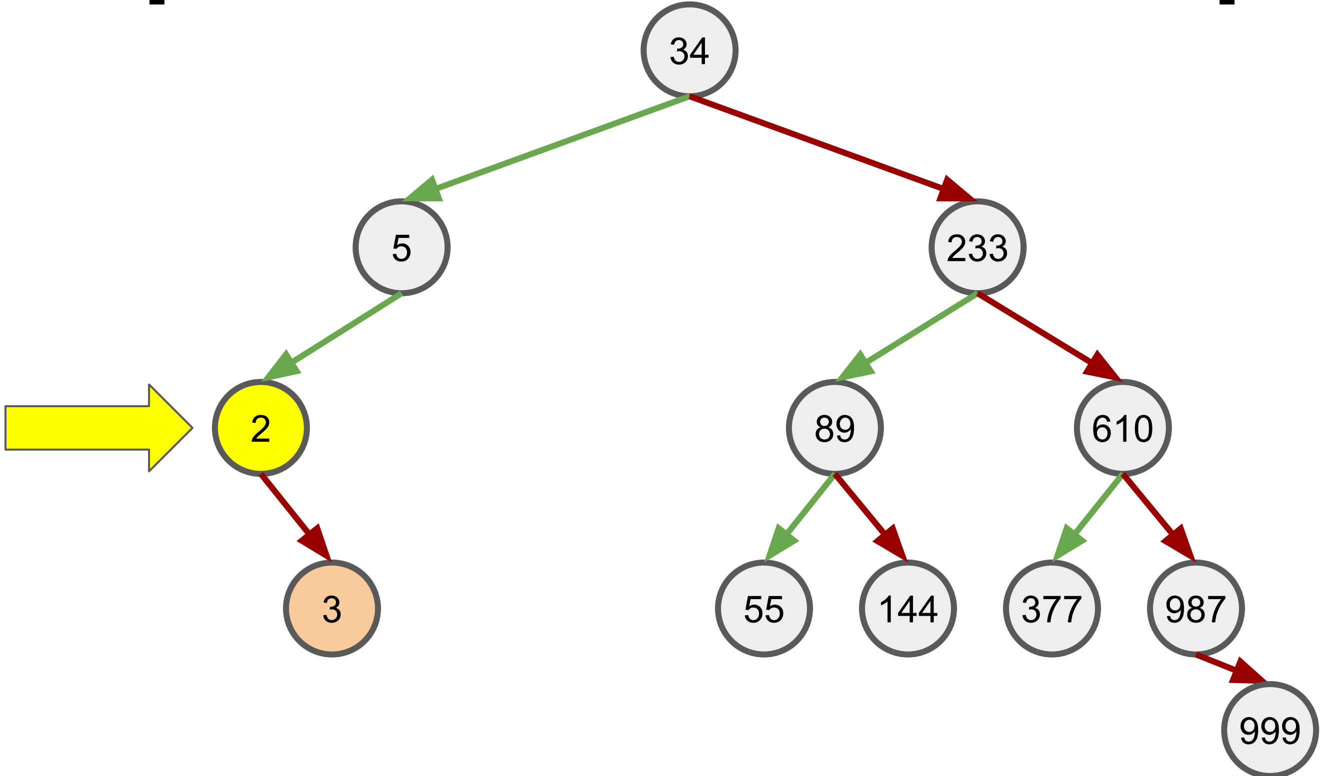
delete(13)

Imbalance at 5!! wat do



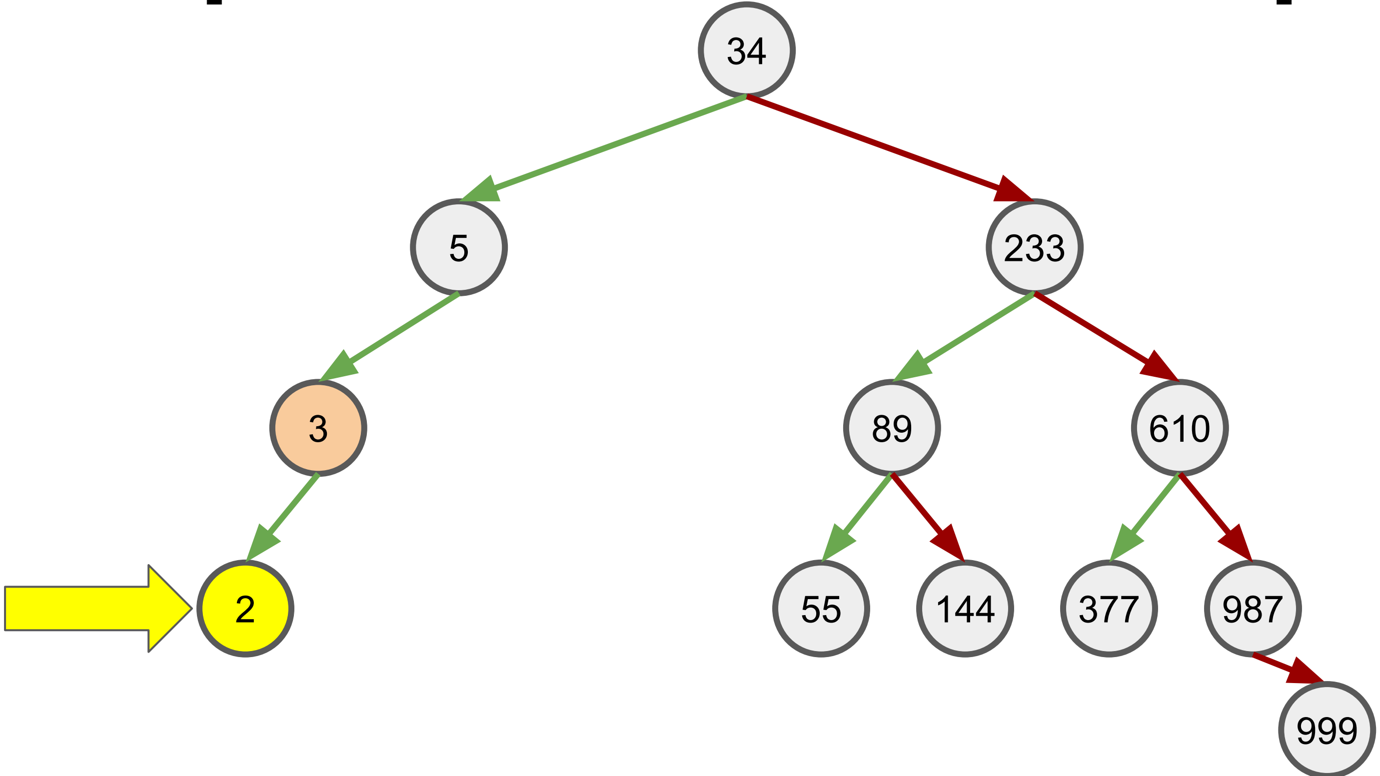
delete(13)

leftRotate(2)



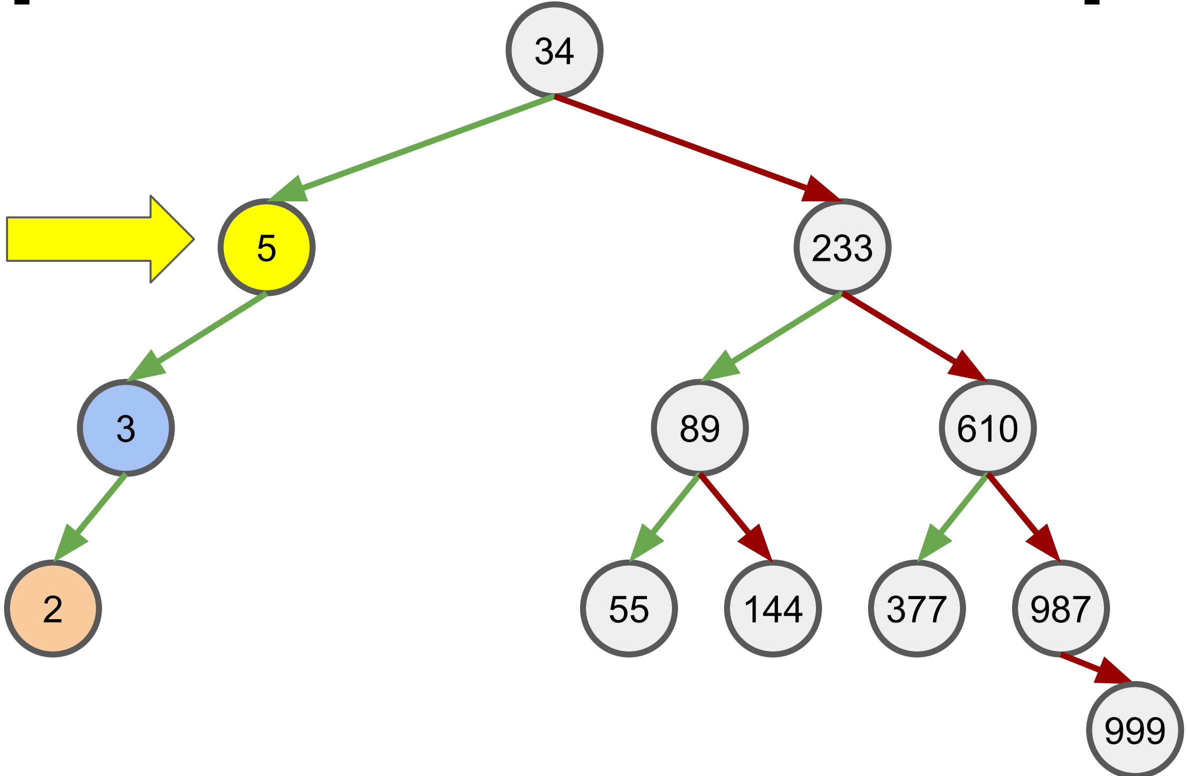
delete(13)

leftRotate(2)



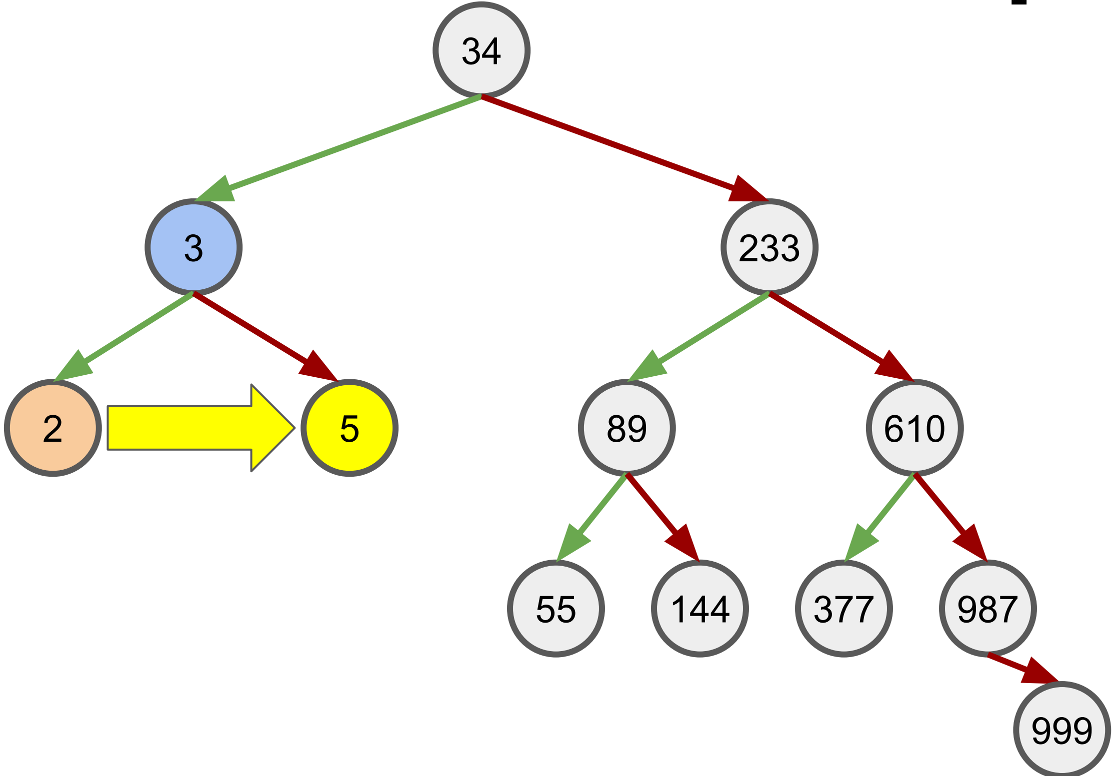
delete(13)

rightRotate(5)



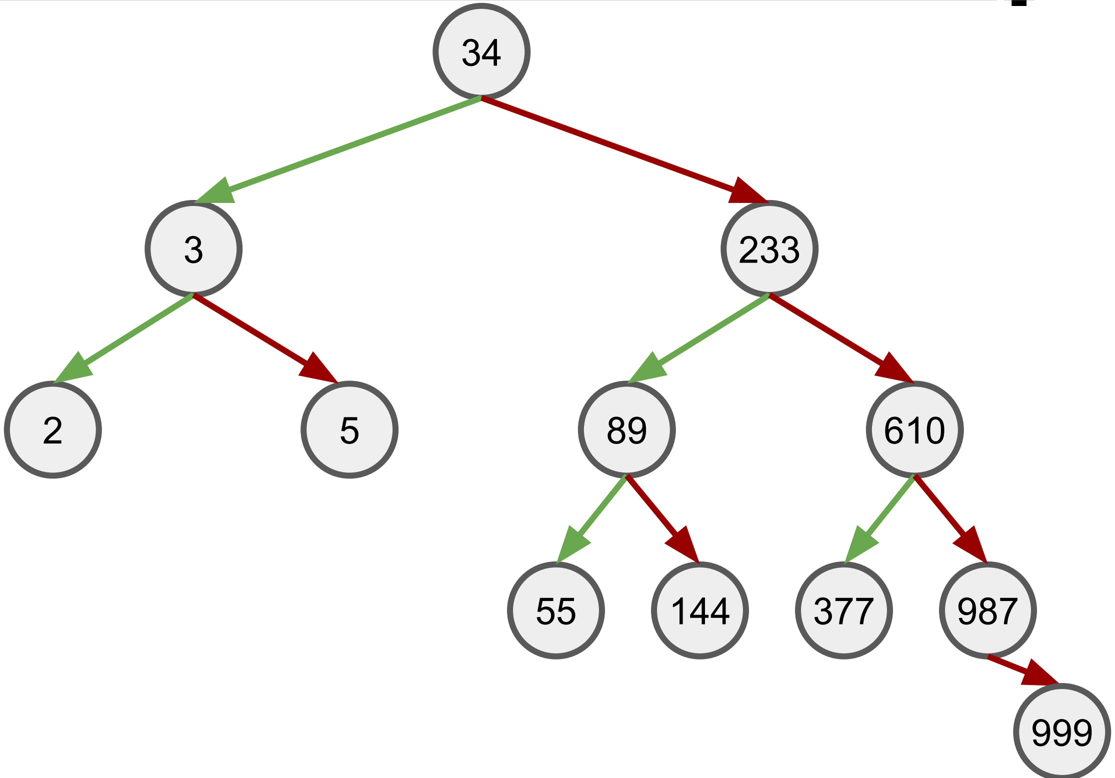
delete(13)

rightRotate(5)



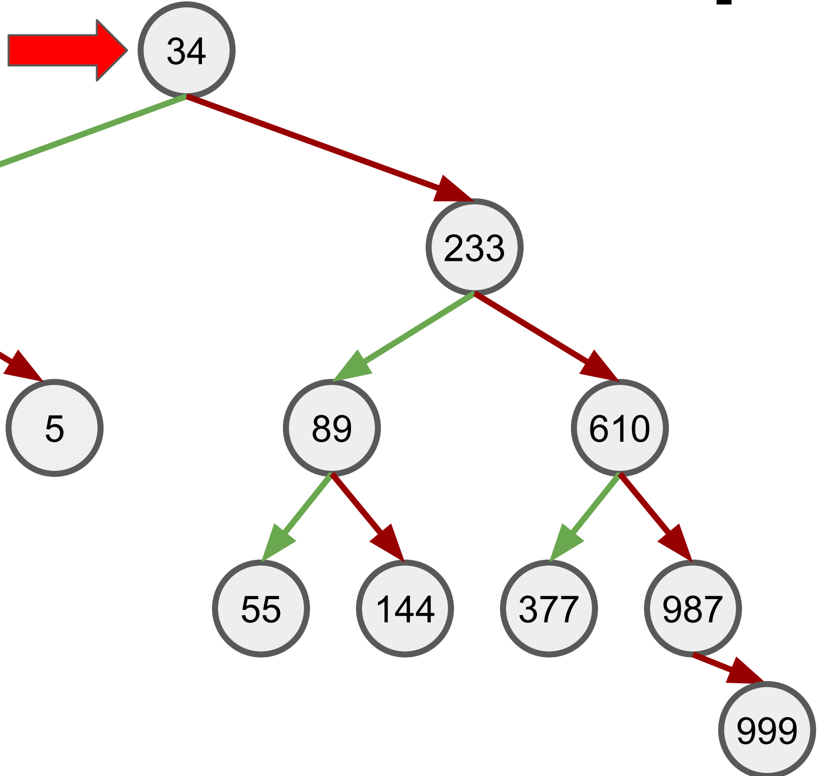
delete(13)

Done???



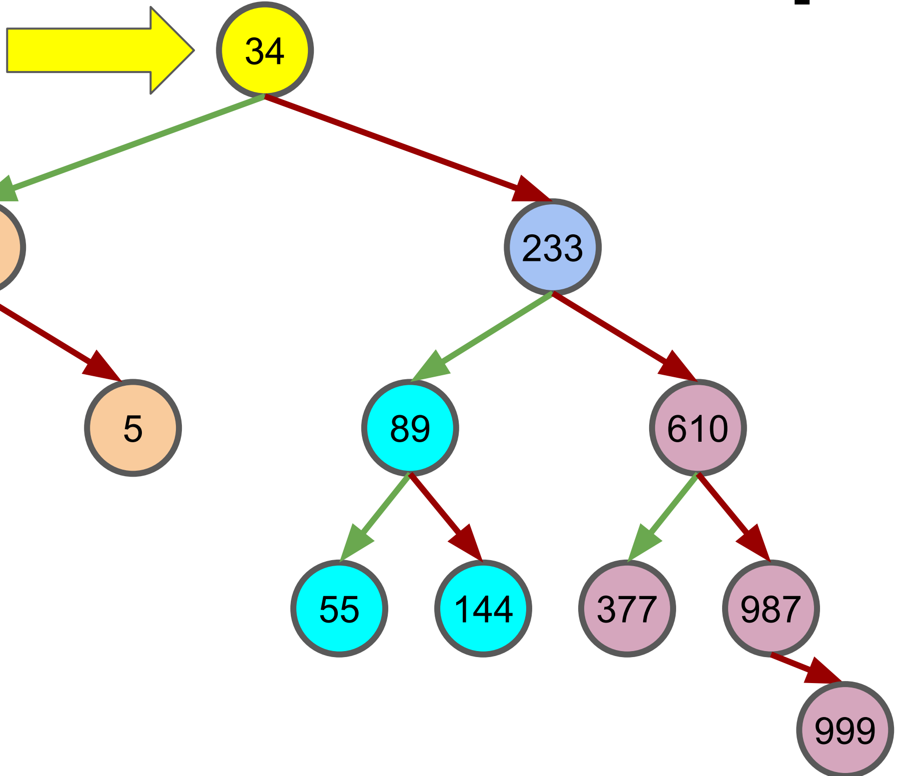
delete(13)

Imbalance here!!! wat do



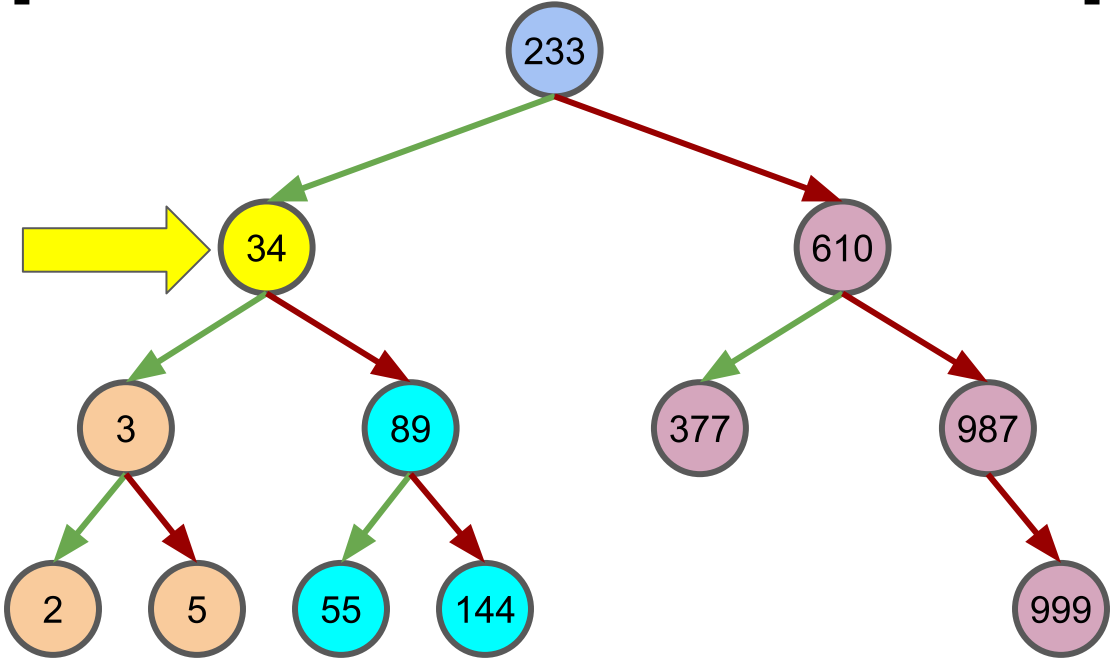
delete(13)

leftRotate(34)



delete(13)

leftRotate(34)



Rotations

Summary:

If v is out of balance and left heavy:

1. v.left is balanced: right-rotate(v)
2. v.left is left-heavy: right-rotate(v)
3. v.left is right-heavy: left-rotate(v.left)
   right-rotate(v)

If v is out of balance and right heavy:

Symmetric three cases….

Number of rotations

- For insert:
- For delete:

Number of rotations

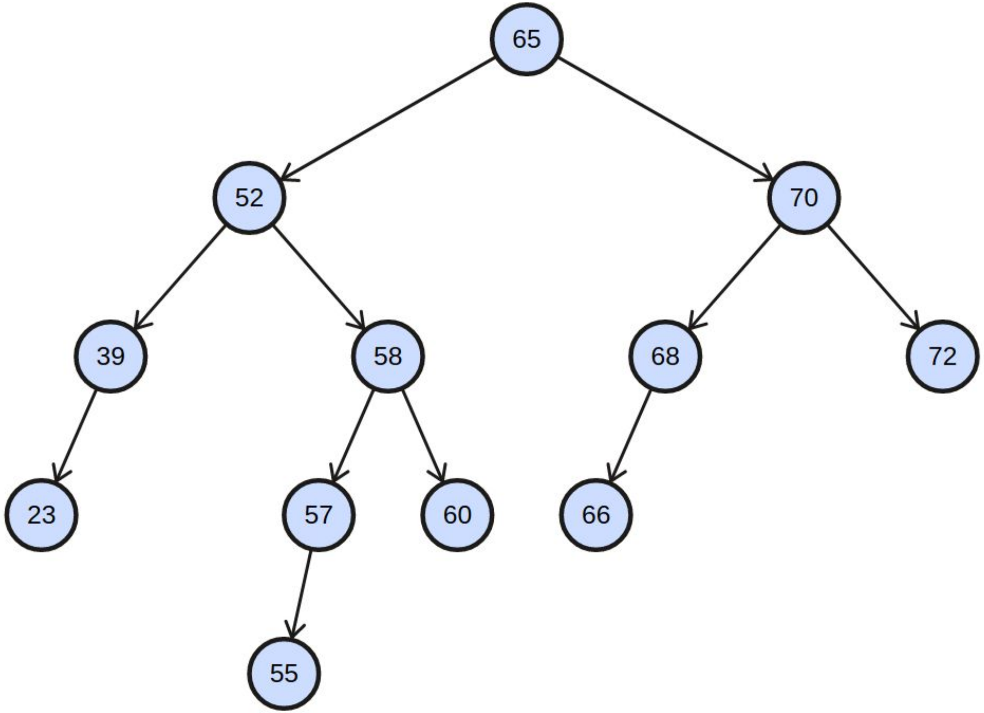
- For insert: at most 2
- For delete: *O(logn)*, because you may have to rotate all the way up to the root

Tutorial Time

Slides for tutorials are taken and adapted from Ian

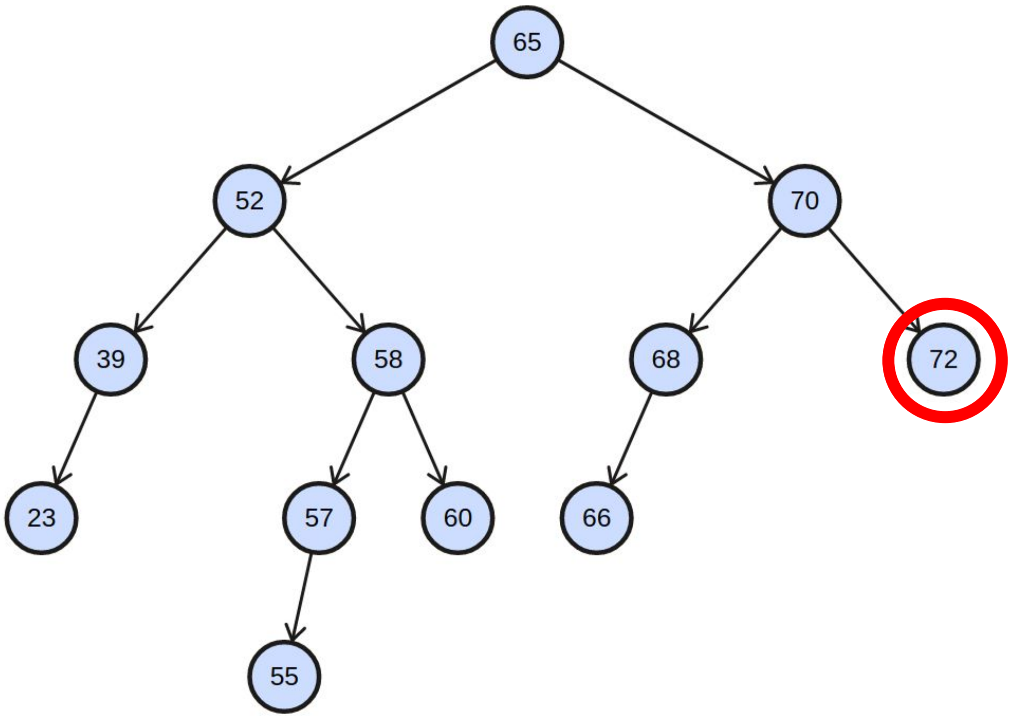
Problem 1: AVL Tree Review

Trace the deletion of the node with the key 70.



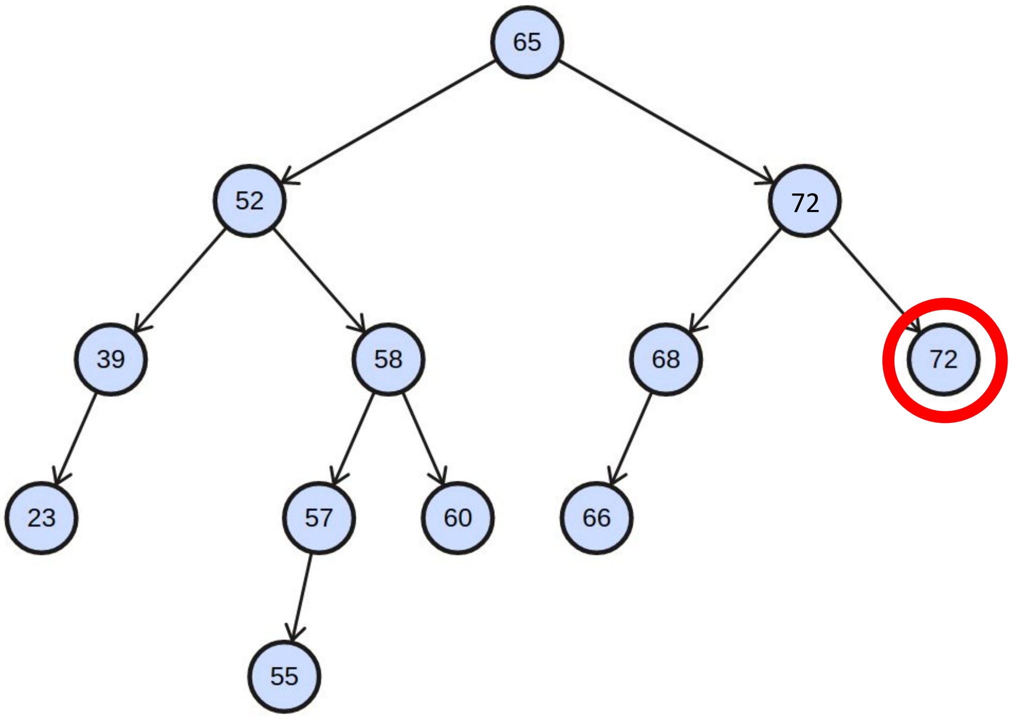
Problem 1: AVL Tree Review

First, find the successor of 70, which is 72.



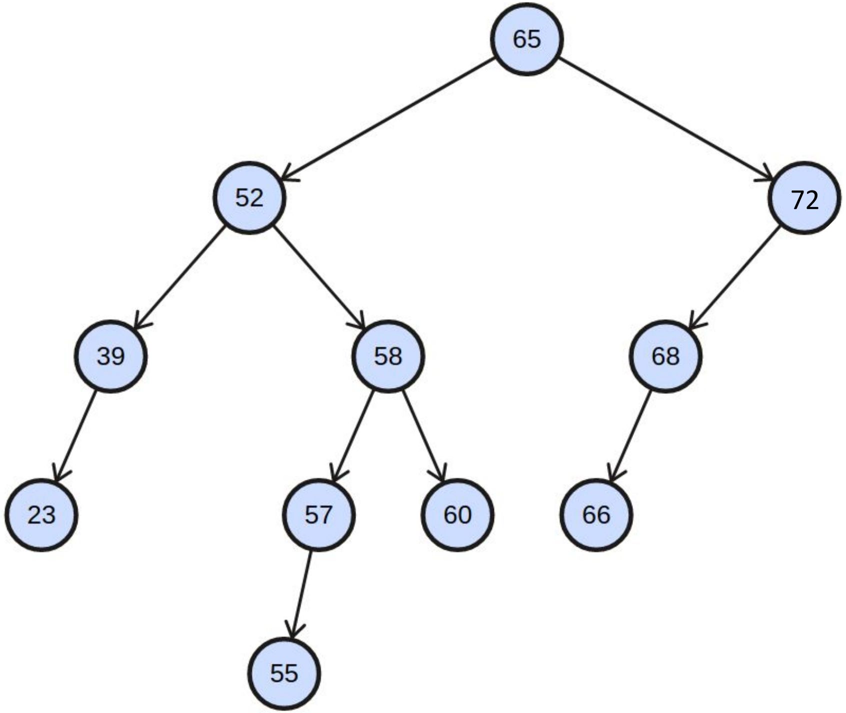
Problem 1: AVL Tree Review

Copy the value of the successor over.



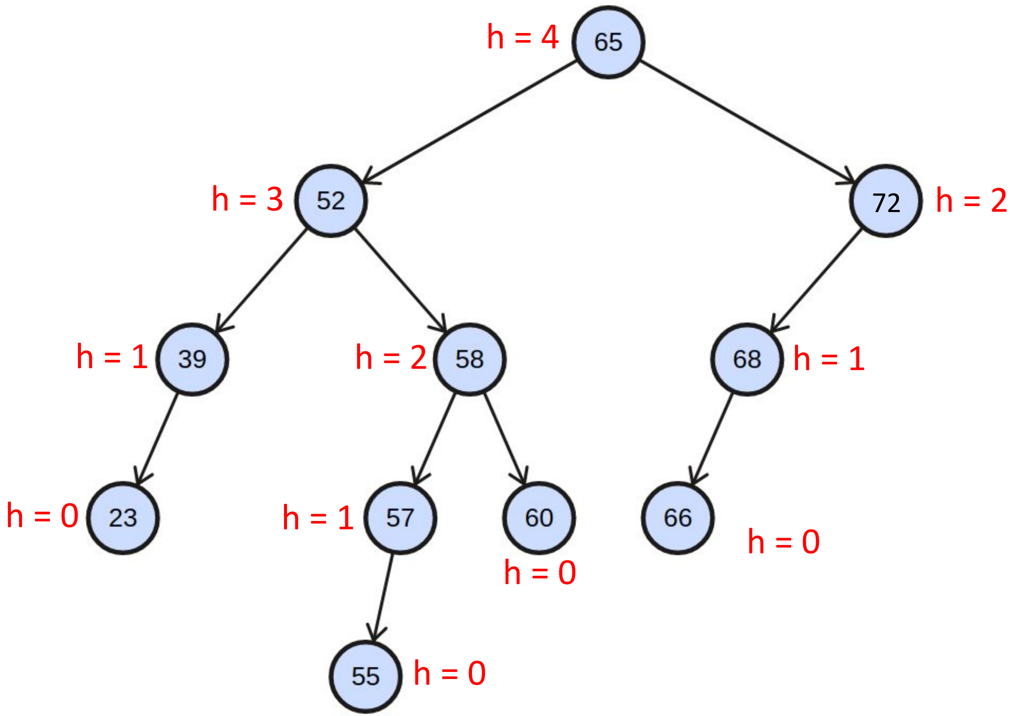
Problem 1: AVL Tree Review

Then, delete the successor.



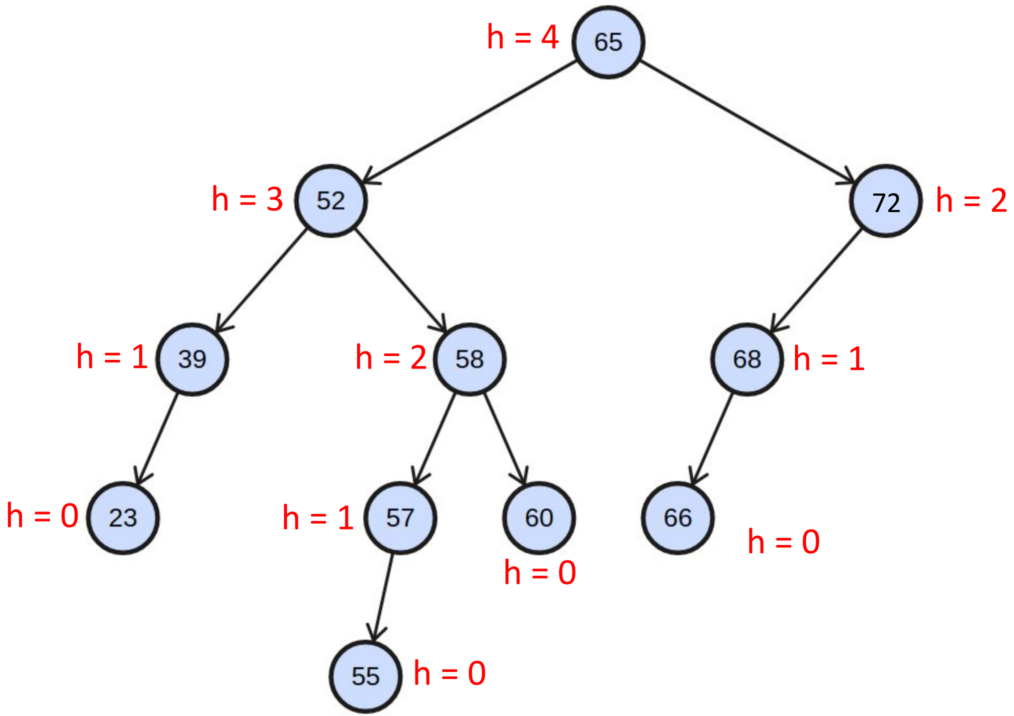
Problem 1: AVL Tree Review

Now, the subtree rooted at the node with key 72 is unbalanced!



Problem 1: AVL Tree Review

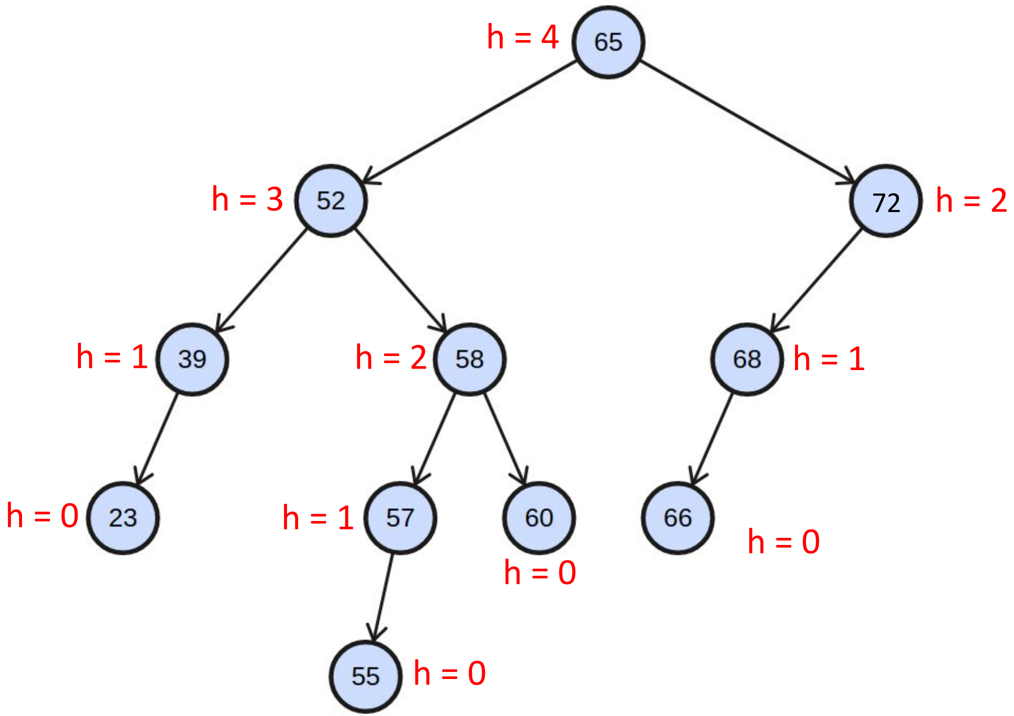
Let the node with key 72 be v.



Problem 1: AVL Tree Review

Let the node with key 72 be v.

v is out of balance and left heavy.

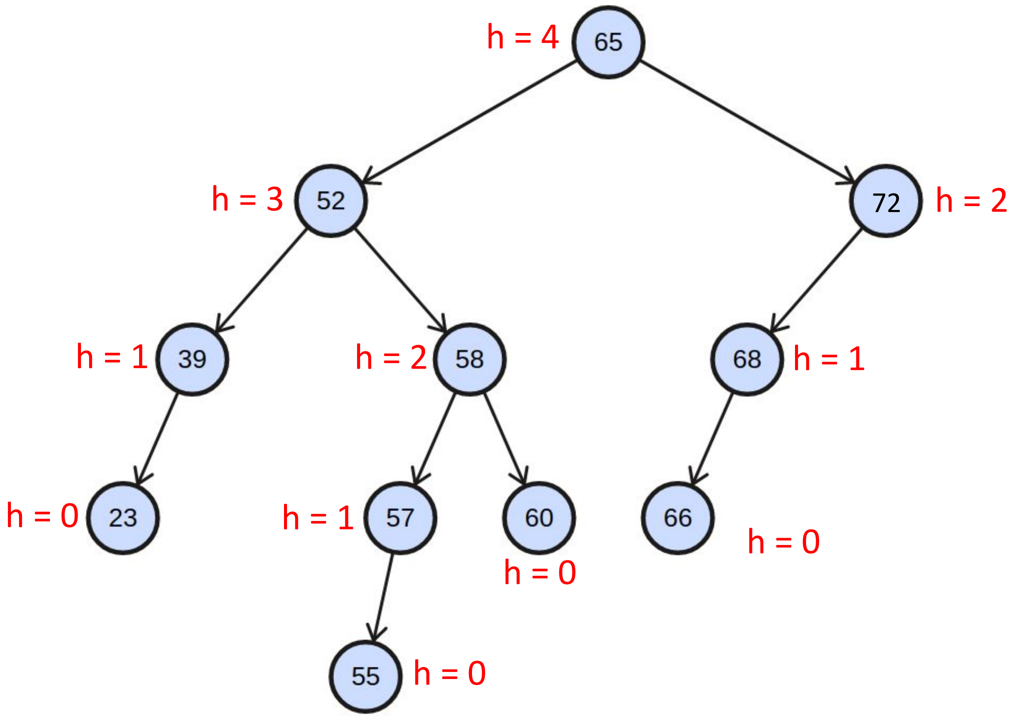


Problem 1: AVL Tree Review

Let the node with key 72 be v.

v is out of balance and left heavy.

v.left is left heavy.



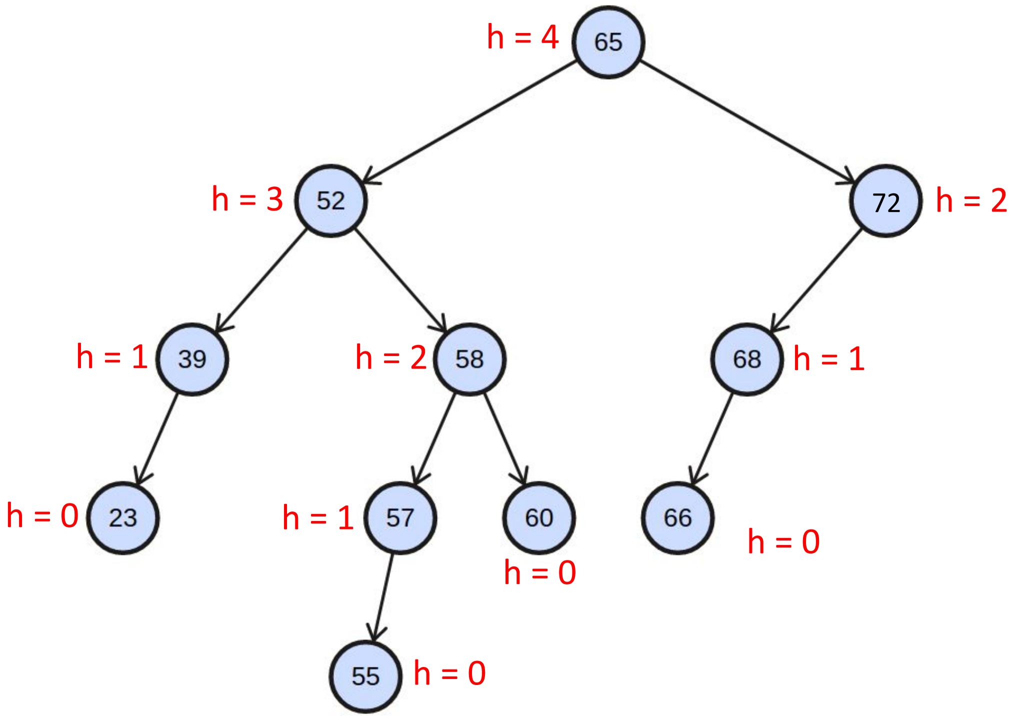
Problem 1: AVL Tree Review

Let the node with key 72 be v.

v is out of balance and left heavy.

v.left is left heavy.

Perform a right rotation on v!



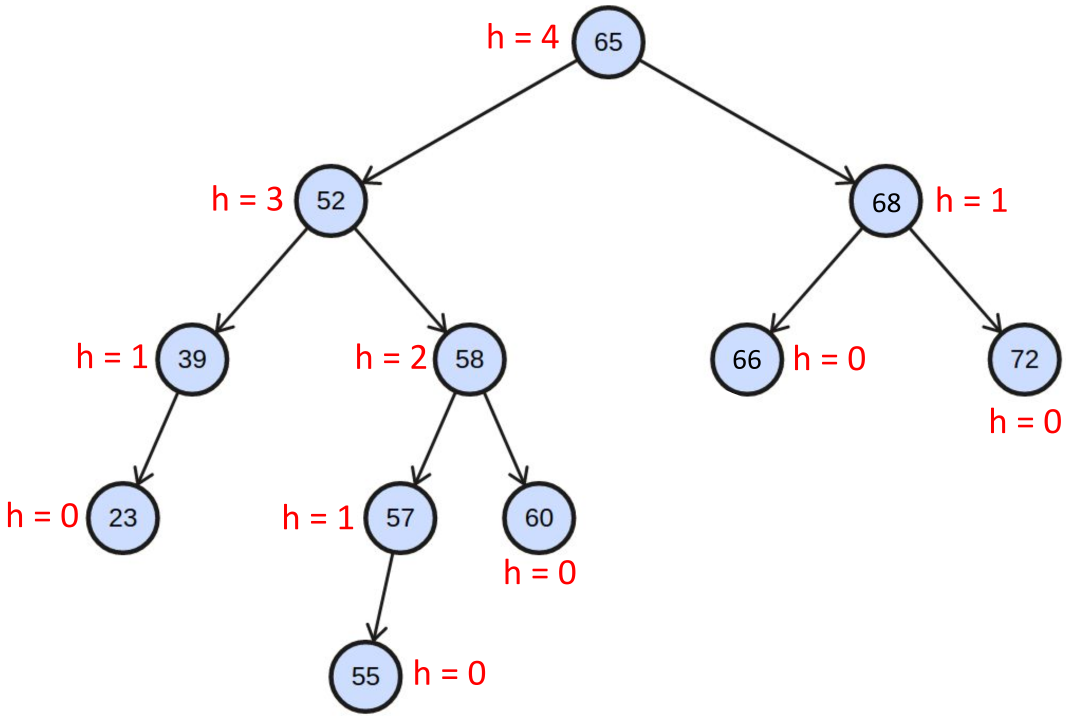
Problem 1: AVL Tree Review

Let the node with key 72 be v.

v is out of balance and left heavy.

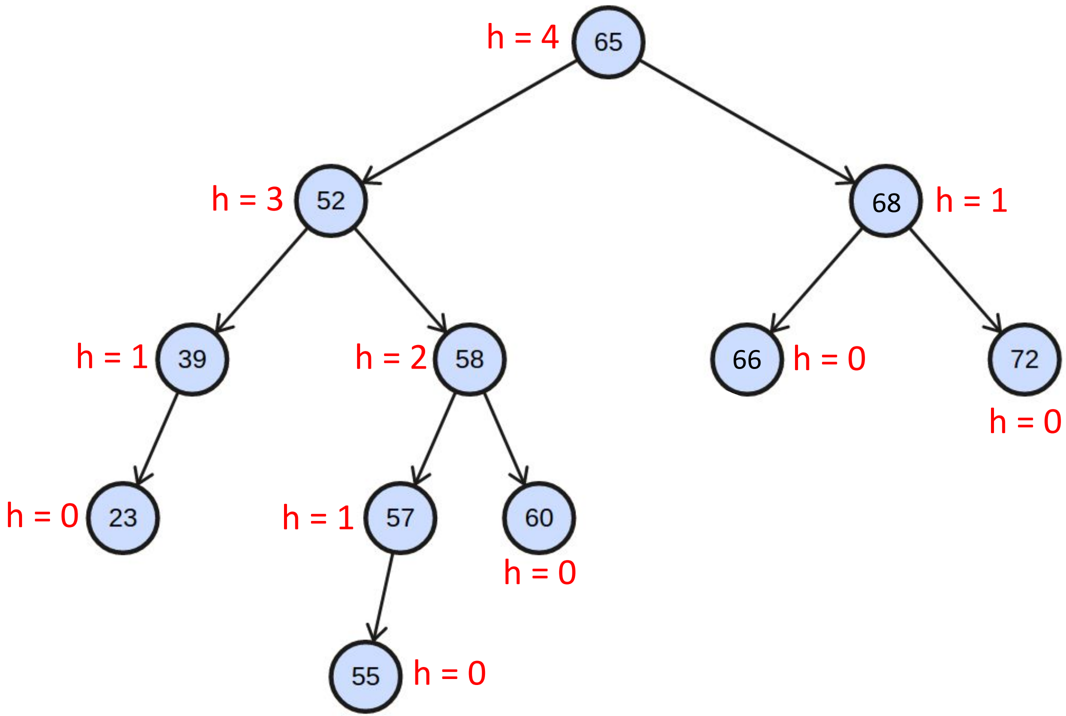
v.left is left heavy.

Perform a right rotation on v!



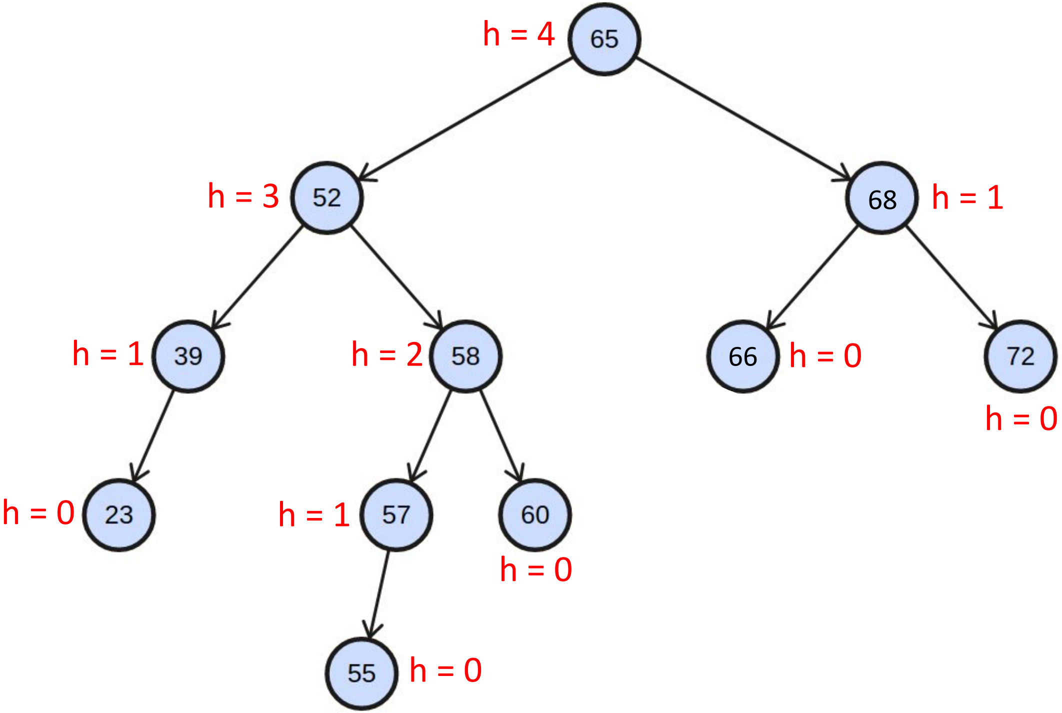
Problem 1: AVL Tree Review

Now, the subtree rooted at the node with key 65 is unbalanced.



Problem 1: AVL Tree Review

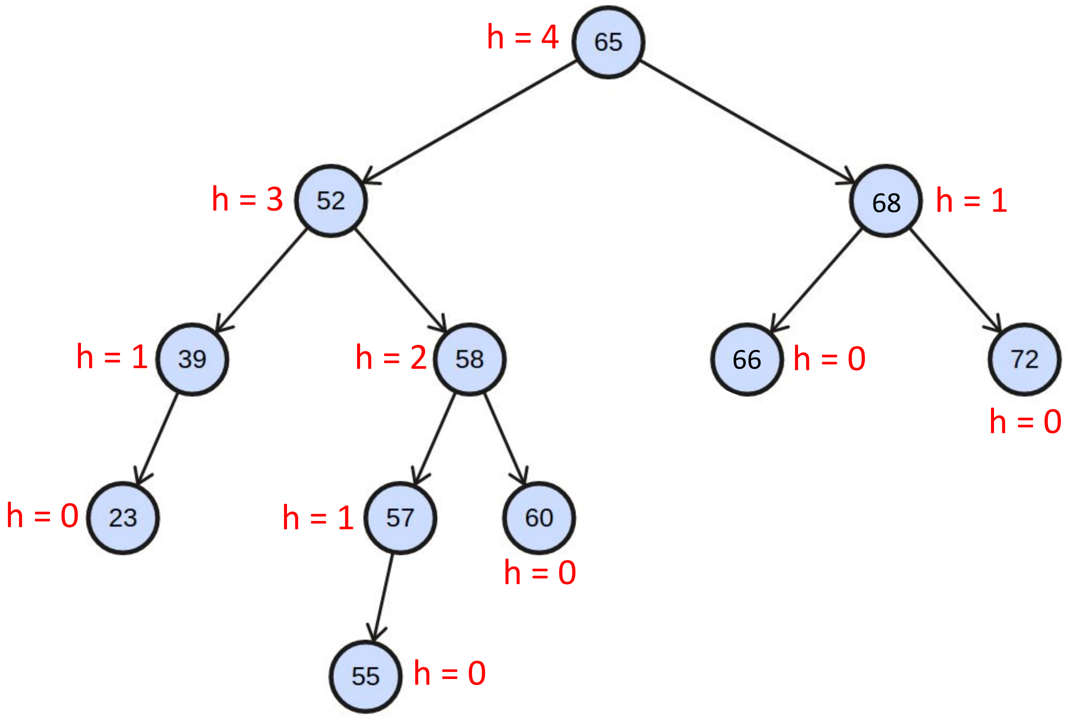
Let the node with key 65 be v.



Problem 1: AVL Tree Review

Let the node with key 65 be v.

v is out of balance and left heavy.

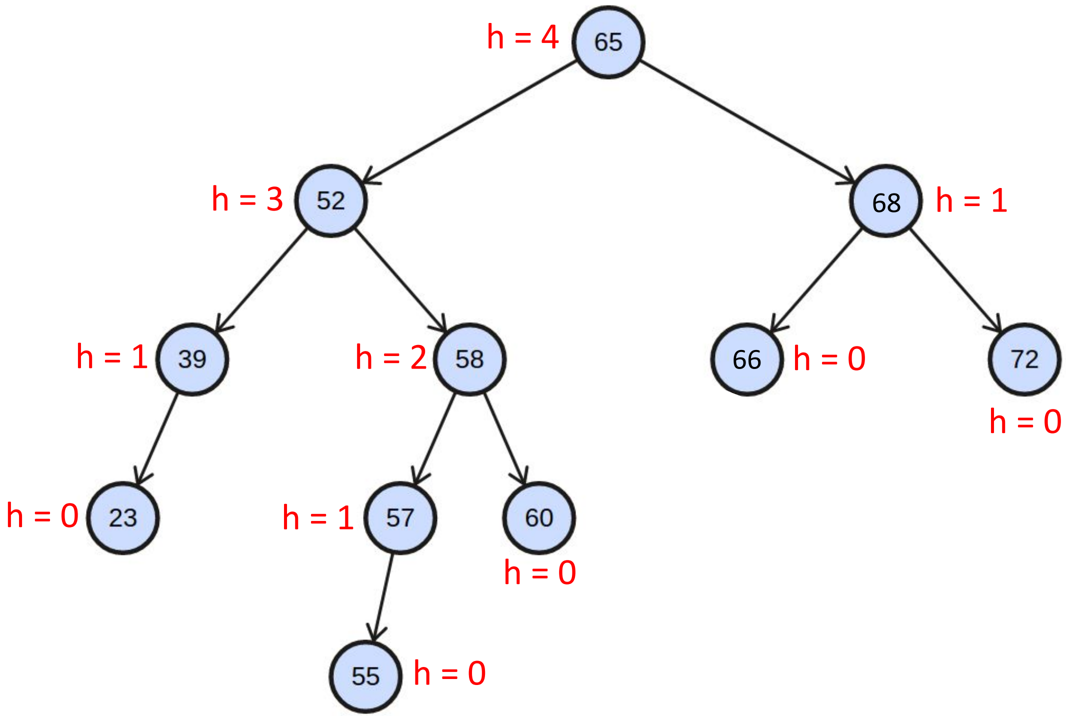


Problem 1: AVL Tree Review

Let the node with key 65 be v.

v is out of balance and left heavy.

v.left is right heavy.



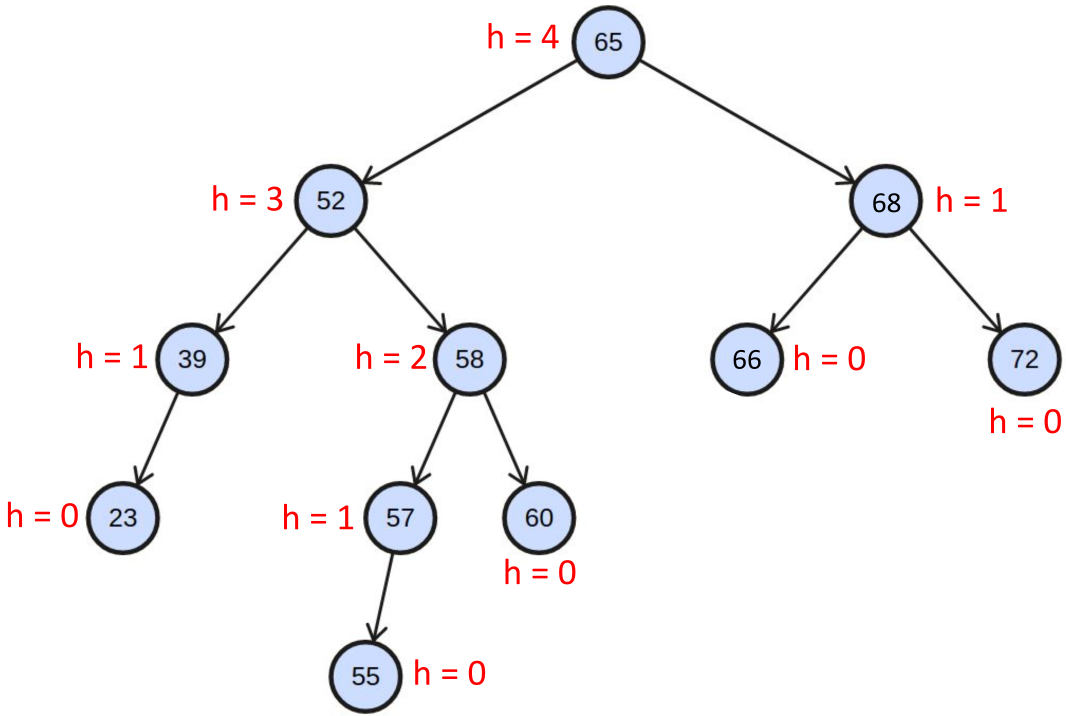
Problem 1: AVL Tree Review

Let the node with key 65 be v.

v is out of balance and left heavy.

v.left is right heavy.

Perform a left rotation on v.left, then a right rotation on v!



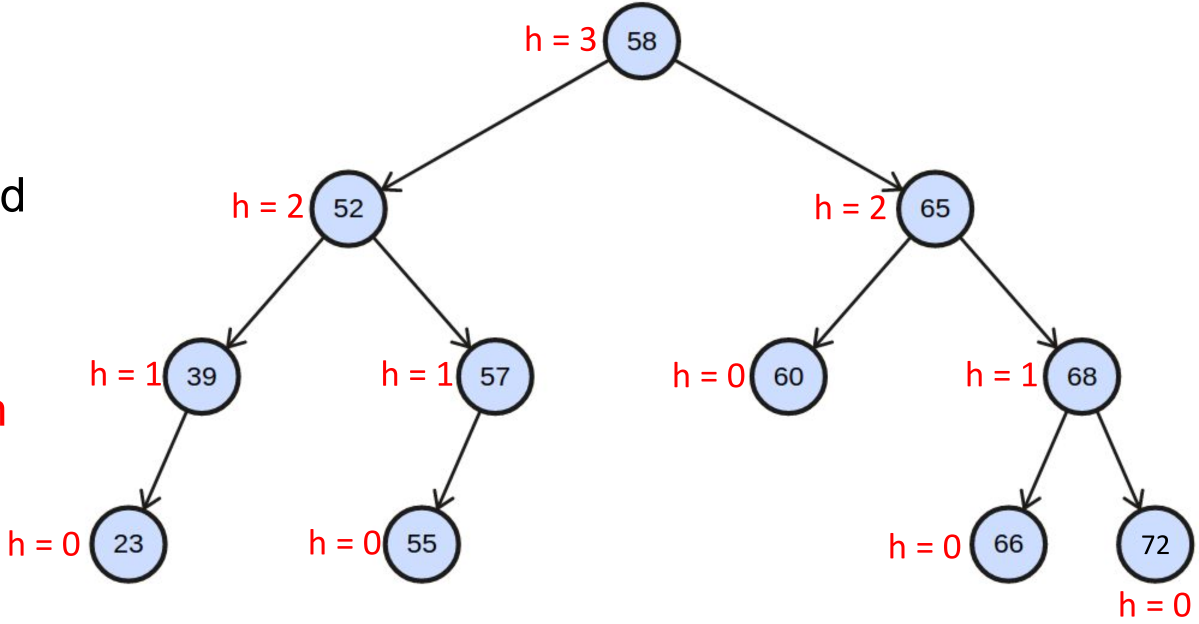
Problem 1: AVL Tree Review

Let the node with key 65 be v.

v is out of balance and left heavy.

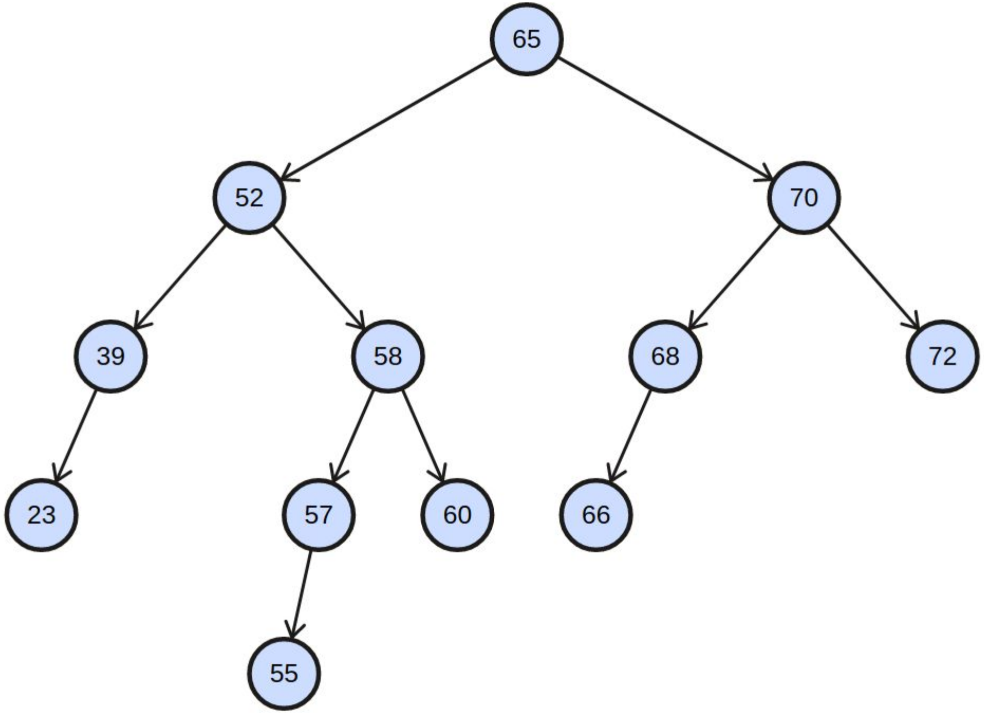
v.left is right heavy.

Perform a left rotation on v.left, then a right rotation on v!



Problem 1: AVL Tree Review

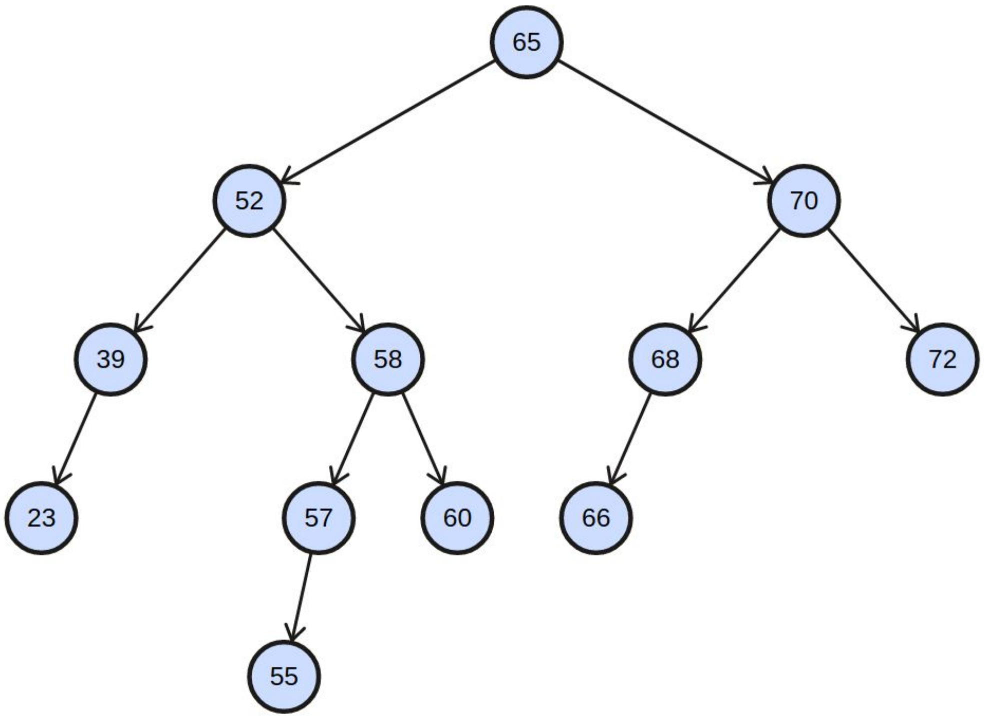
- Identify the roots of all maximally imbalanced AVL subtrees in the original tree. A maximal imbalanced tree is one with the minimum possible number of nodes given its height h.



Problem 1: AVL Tree Review

- Identify the roots of all maximally imbalanced AVL subtrees in the original tree. A maximal imbalanced tree is one with the minimum possible number of nodes given its height h.

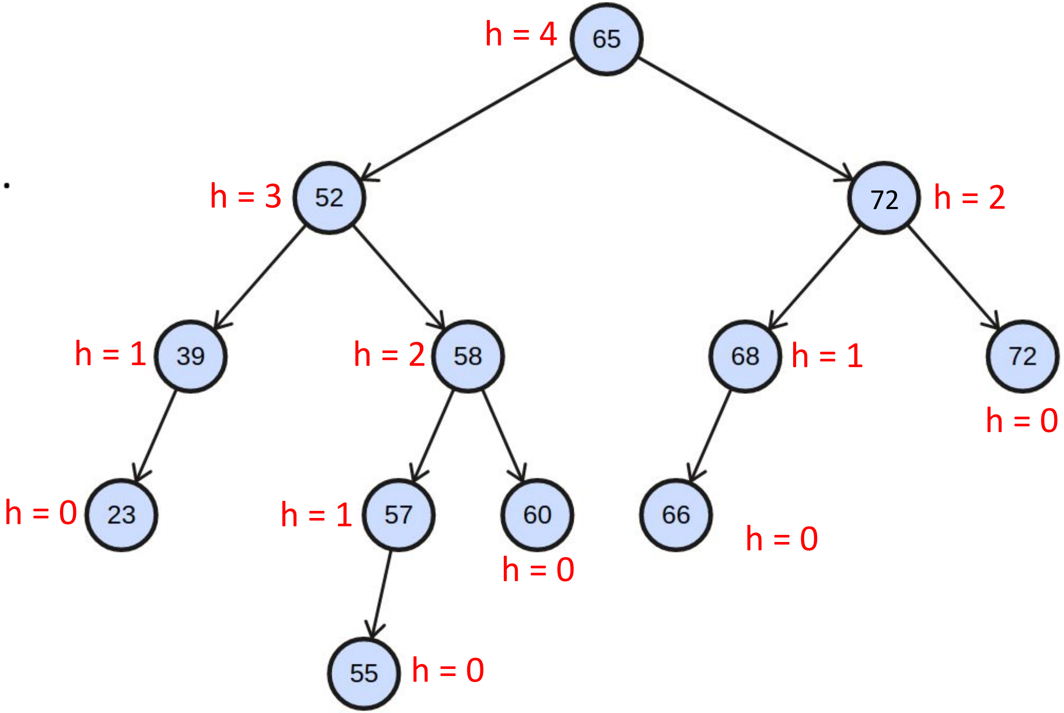
All of them!



Problem 1: AVL Tree Review

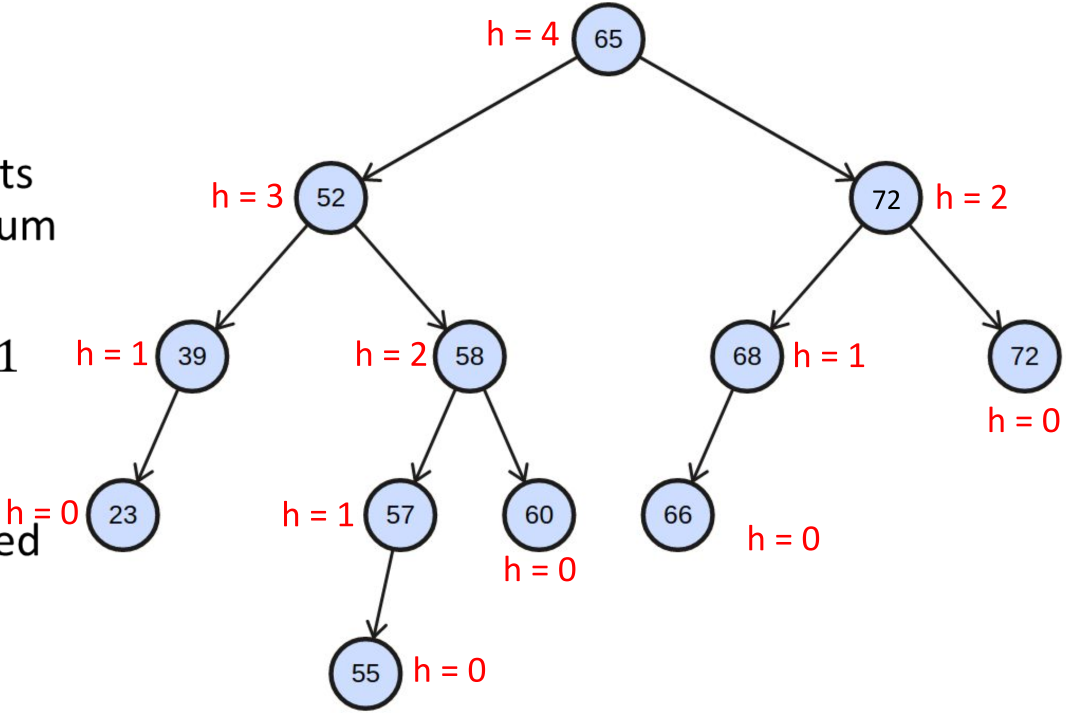
- Identify the roots of all maximally imbalanced AVL subtrees in the original tree. A maximal imbalanced tree is one with the minimum possible number of nodes given its height h.

All of them!



Problem 1: AVL Tree Review

- An AVL tree of height h with the minimum possible number of nodes has two subtrees of heights $h-1$ and $h-2$ with the minimum possible number of nodes
- $S(h) = S(h-1) + S(h-2) + 1$
- This means that a maximally imbalanced AVL tree has all its subtrees be maximally imbalanced



Lecture 1: AVL Tree Review

During lectures, we’ve learnt that we need to store and maintain height information for each AVL tree node to determine if there is a need to rebalance the AVL tree during insertion and deletion. However, if we store height as an `int`, each tree node now requires 32 extra bits. Can you think of a way to reduce the extra space required for each node to 2 bits instead?

Lecture 1: AVL Tree Review

- Instead of storing the height, we can store and maintain the balance factor for each node
- Balance factor is equal to the difference between the left and right subtrees of a node
- Remember our invariant!
  - An AVL tree is a height balanced tree
  - A tree is height balanced if every node in the tree is height balanced
  - A node v is height balanced if the difference between v.left.height and v.right.height is less than or equals to 1
- As such, the balance factor only needs to take up the values -1, 0 or 1.
- This requires only 2 bits which gives us $2^2 = 4 > 3$ distinct representations.

Problem 2a: Chicken Rice

- Imagine you are the judge of a chicken rice competition. You have in front of you n plates of chicken rice. Your goal is to identify which plate of chicken rice is best. In order to do so, you have devised the following algorithm:

1. Put the first plate on your table.
2. Go through all the remaining plates. For each plate, taste the chicken rice on the plate, as well as the chicken rice on the table to determine which is better.
   - If the new plate is better than the one on the table, replace the plate on the table with the new plate.
3. When you are done, the plate on your table is the winner!

Problem 2a: Chicken Rice

Assume each plate begins containing n bites of chicken rice. When you are done, in the worst-case, how much chicken rice is left on the winning plate?

- Only one bite left!
- In the worst case, the first plate on the table is already the best plate of chicken rice
- Thus, it is used to compare against all the remaining $n - 1$ plates, resulting in $n - 1$ bites being taken

Problem 2b: Chicken Rice

Oh no! We want to make sure that there is as much chicken rice left on the winning plate as possible (so you can take it home and give it to all your friends). Design an algorithm to maximise the amount of remaining chicken rice on the winning plate, once you have completed the testing/tasting process.

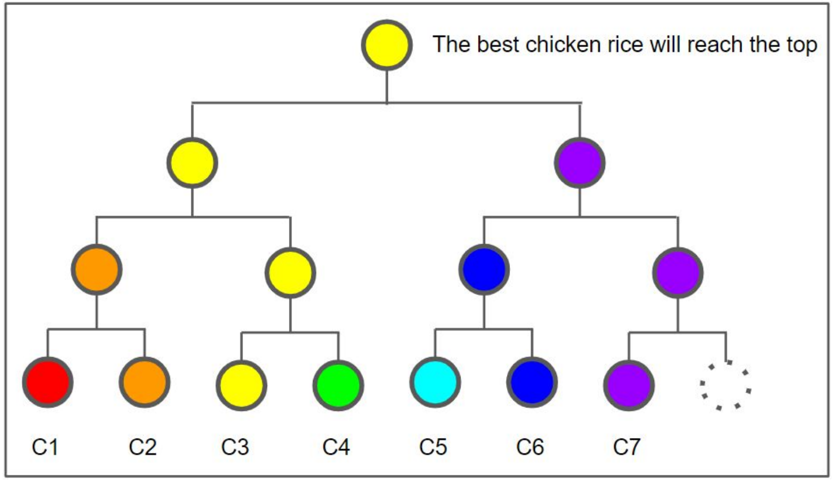
1. How much chicken rice is left on the winning plate?
2. How much chicken rice have you had to consume in total?

Give a tight asymptotic bound for both questions above.

Problem 2b: Chicken Rice

- Use a tournament tree!
  - Group the plates of chicken rice into pairs and compare within each pair to get a winner
  - Group the winners of the previous round into pairs and compare within each pair to get a winner
  - Repeat until we have only 1 plate left

- In the worst case, we consume
  - $O(\log n)$ bites from the winning plate
    - Height of the tree
  - $O(n)$ bites overall
    - Each comparison takes 2 bites
    - Each comparison removes 1 plate



Problem 2c: Chicken Rice

Now, I do not want to find the best chicken rice, but the **median** chicken rice. Again, design an algorithm to maximise the amount of remaining chicken rice on the median plate once you have completed the testing/tasting process.

1. How much chicken rice is left on median plate?
2. How much chicken rice have you had to consume in total?

Give a tight asymptotic bound for both questions above. If your algorithm is randomised, give your answers in expectation.

Problem 2c: Chicken Rice

- Use quickselect!
- Analysis
  - In one step of quickselect (with an array of size n)
    - The pivot plate has $n-1$ bites eaten
    - All other plates have 1 bite eaten
  - If the pivot plate is chosen at random, the median plate has an expected cost of $\frac{1}{n}(n-1) + \left(1 - \frac{1}{n}\right)1 = 2\left(1 - \frac{1}{n}\right) = 2 - \frac{2}{n} \leq 2$ bites eaten
    - In other words, at each level of recursion, there are at most 2 bites eaten from the median plate in expectation

Problem 2c: Chicken Rice

- Analysis
  - With high probability and in expectation, the recursion will terminate in $O(\log n)$ levels
  - In the average case, we consume
    - $O(\log n)$ bites from the median plate
    - $O(n)$ bites overall
      - If we select pivots randomly, we should get a good split most of the time

Problem 2c: Chicken Rice

- In the solution sheet, an alternative mentioned is to use an AVL tree
- However, since this solution is deterministic (not randomised), it is possible for there to be the following bad input:
  - Insert the median plate into the AVL tree
    - By default, the first plate inserted will be the root of the AVL tree
  - Insert all other plates in an order such that no rotations ever occur
    - This means that the median plate stays at the root throughout
- In the worst case, we consume $O(n)$ bites from the median plate!

Problem 3: Economic Research

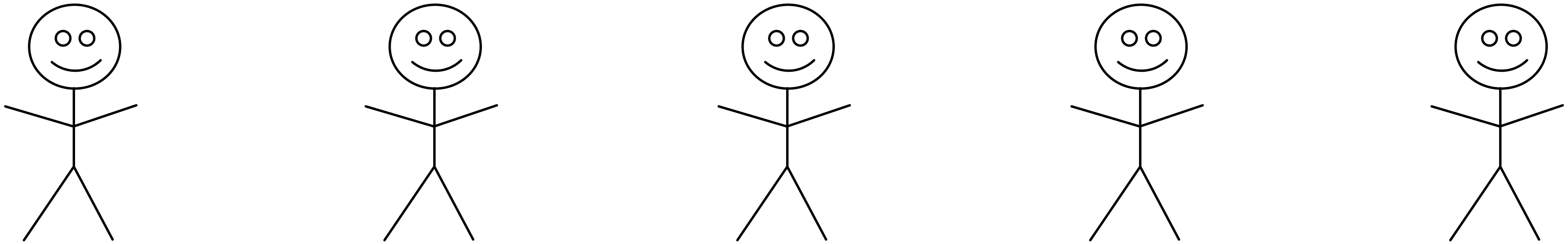
Problem 3: Economic Research

Main Question Idea: You want to divide the dataset into "equi-wealth" age ranges, and given parameter k, you should output k different lists A_1, A_2, ..., A_k with the following properties:

1. All the ages of people in set A_j should be less than or equal to the ages of people in A_{j+1}. That is, each set should be a subset of the original dataset containing a contiguous age range.

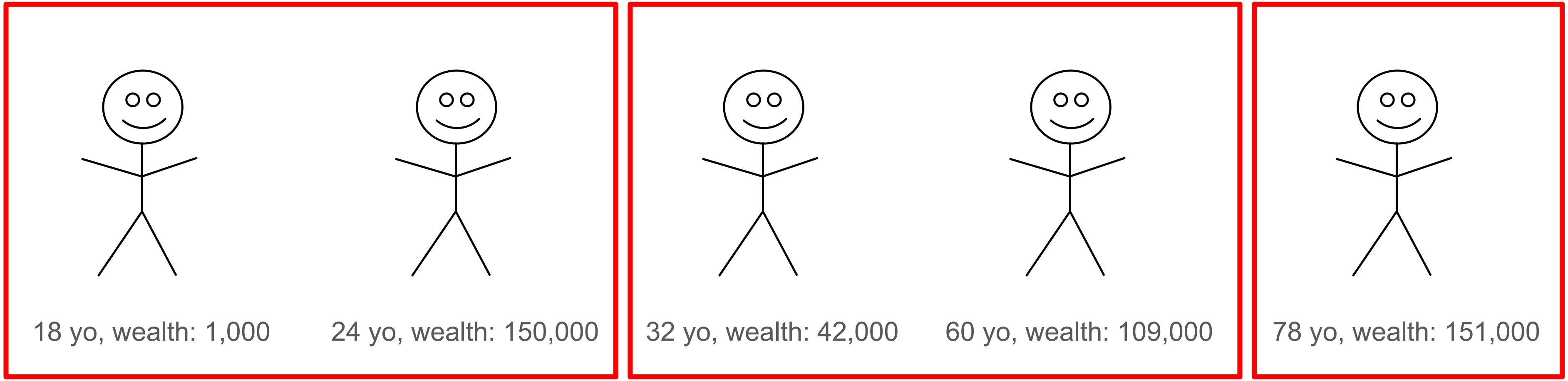
2. The sum of wealth in each set should be (roughly) the same

Problem 3: Example



18 yo, wealth: 1,000   24 yo, wealth: 150,000   32 yo, wealth: 42,000   60 yo, wealth: 109,000   78 yo, wealth: 151,000

Problem 3: Example



Equi-wealth partition: 151,000

Problem 3: Economic Research

Design the most efficient algorithm you can to solve this problem/do the partition, and analyse its time complexity.

Problem 3: Economic Research

Feeling lost?

Try to model this question to similar problems/algorithms you have seen before!

Problem 3: Economic Research

Order Statistics?

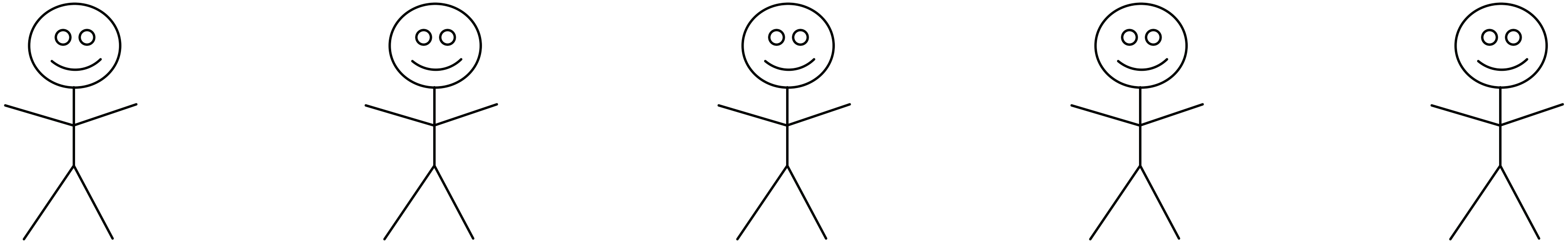
Problem 3: Economic Research

Order Statistics?

We are trying to find the $1/k$, $2/k$, … , $(k-1)/k$ order statistics of the weighted sum!

Problem 3: Economic Research

First and foremost, we would need to find the equi-wealth partition

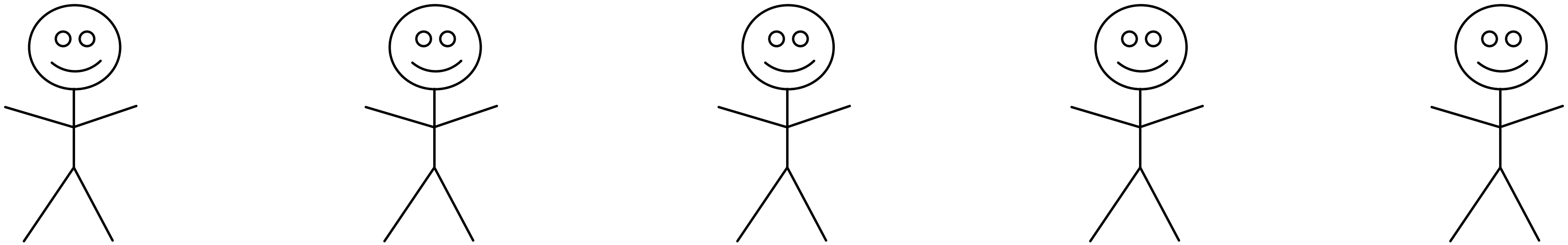


24 yo, wealth: 150,000   32 yo, wealth: 42,000   18 yo, wealth: 1,000   78 yo, wealth: 151,000   60 yo, wealth: 109,000

Equi-wealth partition: 151,000

Problem 3: Economic Research

First and foremost, we would need to find the equi-wealth partition

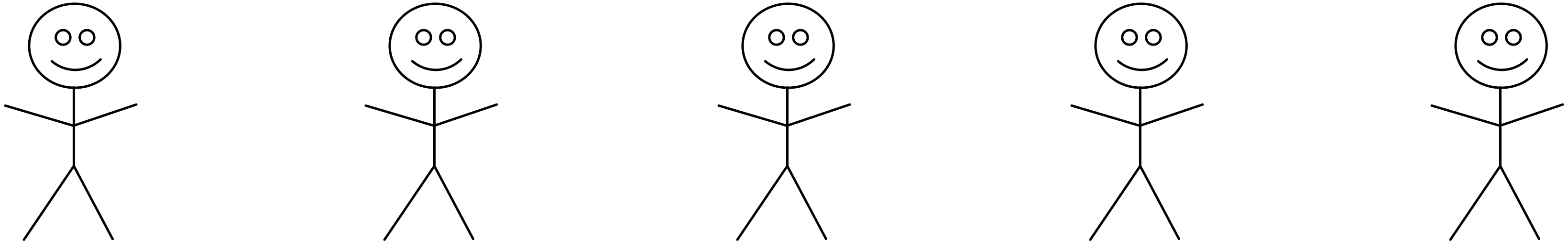


24 yo, wealth: 150,000 | 32 yo, wealth: 42,000 | 18 yo, wealth: 1,000 | 78 yo, wealth: 151,000 | 60 yo, wealth: 109,000

Go through everyone to find total wealth of the population, and divide by k

Problem 3: Economic Research

Next, we want to find the smallest ages with total wealth of at most 151,000

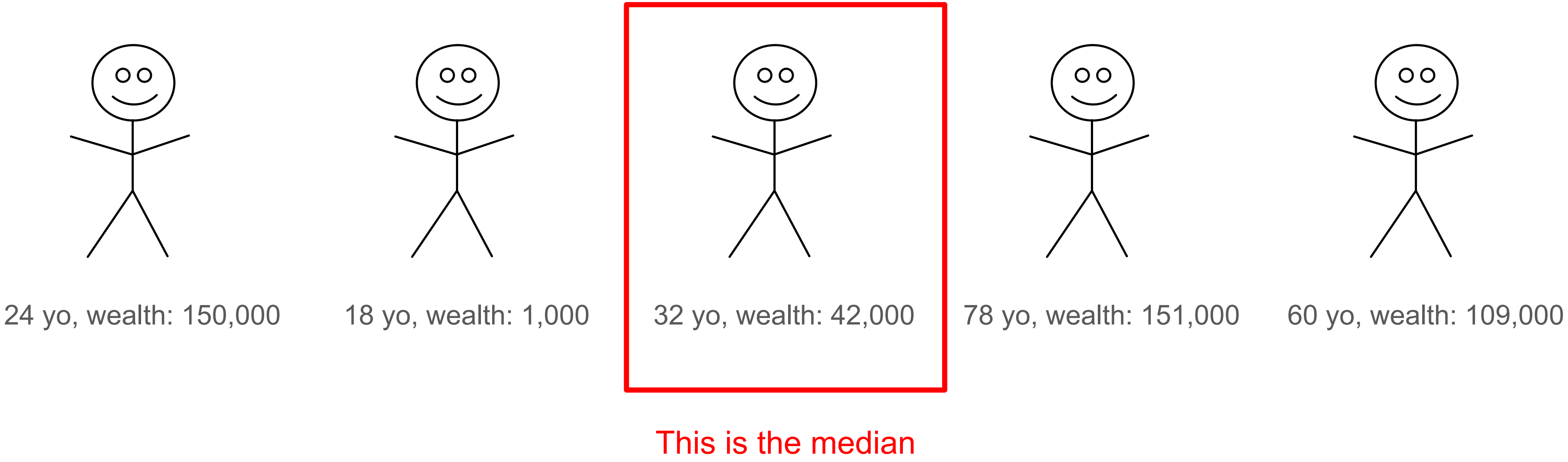


24 yo, wealth: 150,000 | 32 yo, wealth: 42,000 | 18 yo, wealth: 1,000 | 78 yo, wealth: 151,000 | 60 yo, wealth: 109,000

REMEMBER: this is not sorted! So what algorithm can we use?

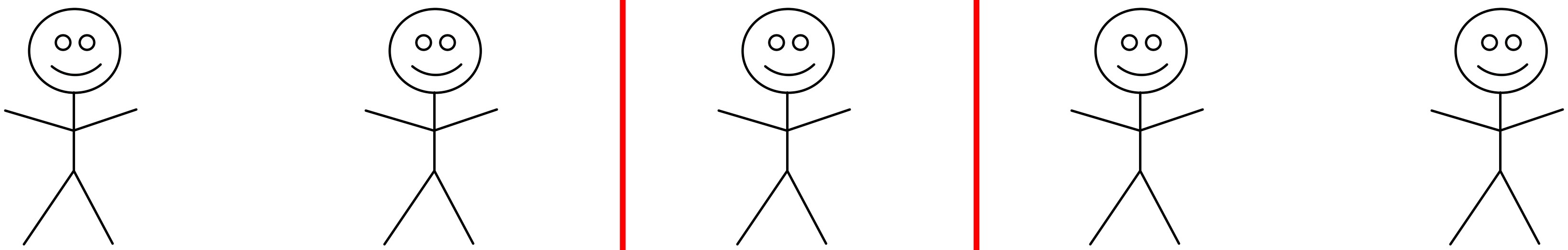
Problem 3: Economic Research

1. Use QuickSelect to find median based on age, and then partition around the median to find the first target (1/k)



Problem 3: Economic Research

2. Sum the totals on the left and right halves, and decide on which side to recurse on. (If target < median, recurse on the left. If target > medium, then target = target - total wealth of the left half.)



24 yo, wealth: 150,000

18 yo, wealth: 1,000

32 yo, wealth: 42,000

78 yo, wealth: 151,000

60 yo, wealth: 109,000

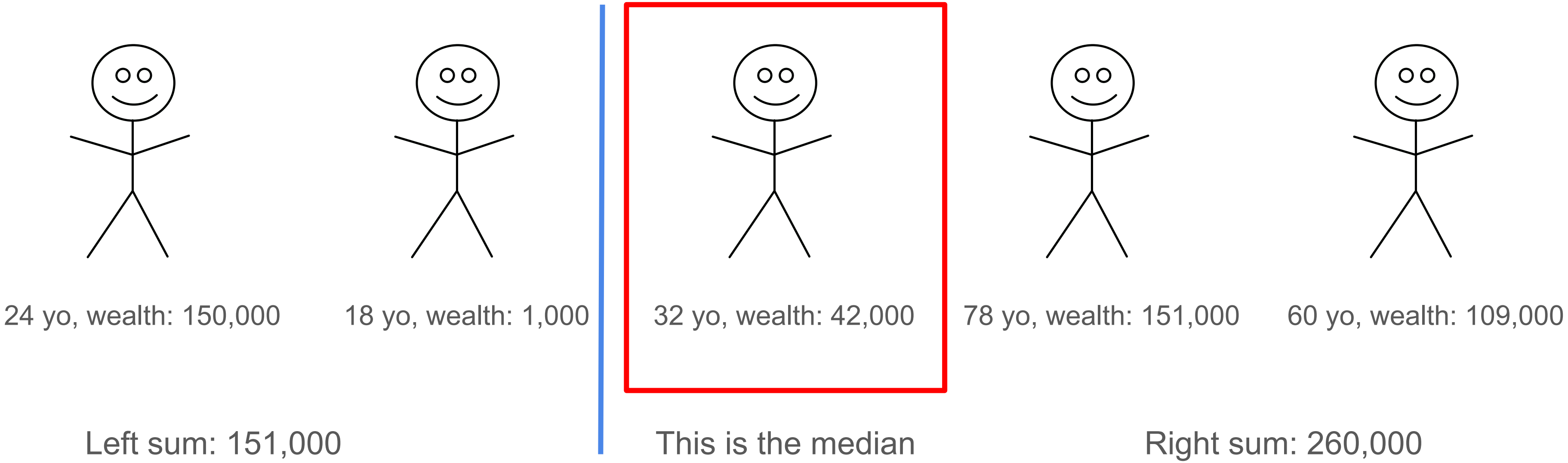
Left sum: 151,000

This is the median

Right sum: 260,000

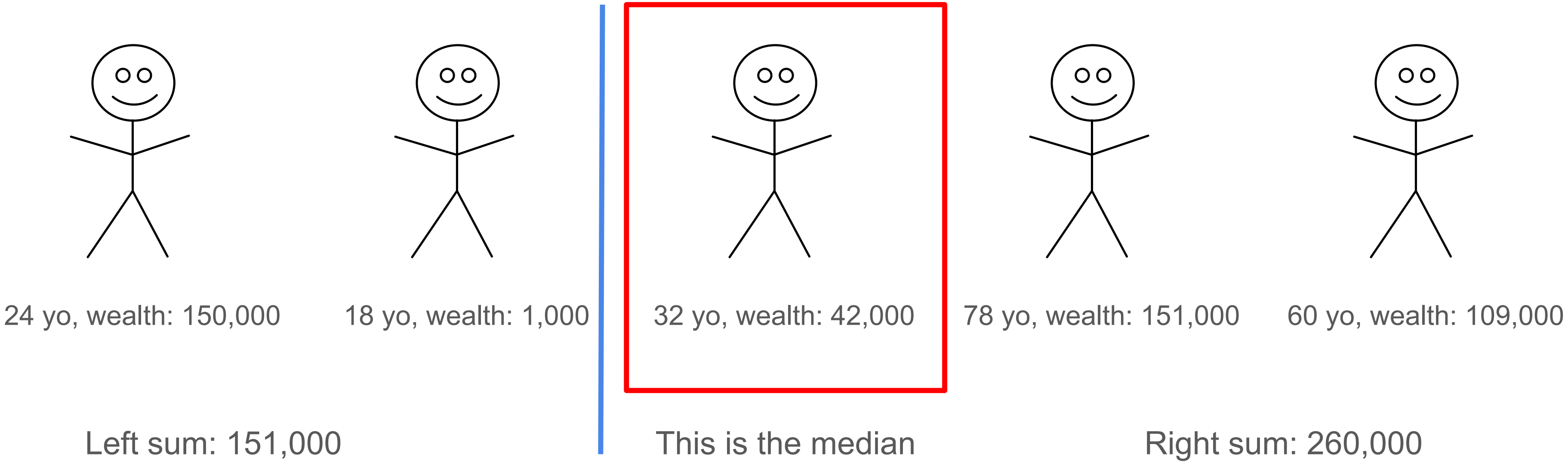
Problem 3: Economic Research

2. Here, there is no need to recurse to the left anymore, since you know the partition is between the 18 and 32 yo.



Problem 3: Economic Research

3. Repeat for the k - 1 targets (2/k, 3/k, … (k-1)/ k



Problem 3: Economic Research

Each QuickSelect: $O(n)$

Problem 3: Economic Research

Each QuickSelect: O(n)

Total time taken for k targets: O(nk)

Problem 3: Economic Research

Modifications?

Problem 3: Economic Research

Modifications

- Can use random pivot instead of median

Problem 3: Economic Research

Modifications

- Can use random pivot instead of median
- Can find all k “breakpoints” at once → less repeated work done partitioning the array

Problem 3: Economic Research

Modifications

- Can use random pivot instead of median
- Can find all k “breakpoints” at once → less repeated work done partitioning the array

Runtime?

Problem 3: Economic Research

Analysing runtime of more efficient method:

1. First divide the array up into k equal sized parts (in terms of the number of elements) by running QuickSort for log(k) levels of recursion.

1 … n/k	n/k+1…2n/k	…	n - k…n

Each level divides the array in half, so at this point, you have k different equal sized pieces → O(n log k) time

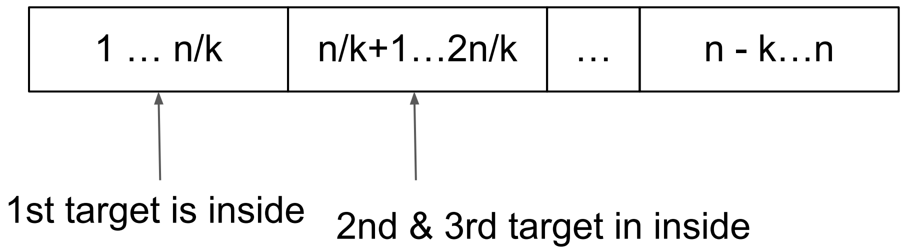
Problem 3: Economic Research

Analysing runtime of more efficient method:

2. For each of the k targets, figure out which piece it is in by:

    1. summing the wealth in each equally sized part → $O(n)$

    2. binary search k times for which of the k pieces it belongs to → $O(k \log k)$



Problem 3: Economic Research

Analysing runtime of more efficient method:

3.Now run the initial algorithm for each target on the correct subarray of size O(n/k), which takes O(n/k) time each. Since there are k targets, the total time will be O(n).

1 … n/k	n/k+1…2n/k	…	n - k…n

Initial algorithm:
Use QuickSelect to find the median based on age, and then partition around the median.
Sum the totals on the left and right halves, and decide on which side to recurse on.
If target is on the left, simply recurse on the left.
If your target is on the right, then subtract from your target the total wealth of the left half.

Problem 3: Economic Research

Analysing runtime of more efficient method:

Recurrence Relation:

$T(n, k) = O(n) + O(k) + T(n/2, k_1) + T(n/2, k_2)$ where $k_1 + k_2 = k$

At each level:
O(n) to partition the array into left and right using a pivot
O(k) to decide for each target if it is in the left and right halves
T (...) parts are for the two recursive calls

Problem 3: Economic Research

Modifications

- Can use random pivot instead of median
- Can find all k “breakpoints” at once → less repeated work done partitioning the array

Runtime? O(n log k) // yay better!

Problem 4: Order Maintenance

Design a data structure for Order Maintenance. The goal here is to maintain a total order over some arbitrary objects. The data structure supports two operations:

1. InsertBefore(A, B): insert B immediately before A
2. InsertAfter(A, B): insert B immediately after A.
3. IsAfter(A, B): is B after A in the total order?

Note: InsertAfter(A, B) adds B immediately after A, while the query operation IsAfter(A, B) asks whether B is anywhere after A in the total order

Problem 4: Order Maintenance

Expected time complexity of each operation is $O(\log n)$, where n is the number of items in the data structure.

Problem 4: Order Maintenance

Expected time complexity of each operation is O(log n), where n is the number of items in the data structure.

Hint: Which data structure has a time complexity of O(log n) for each operation?

Problem 4: Order Maintenance

AVL Trees!

Problem 4: Order Maintenance

InsertAfter(A, B):

1. If A has no right child, then insert B as the right child of A.

Problem 4: Order Maintenance

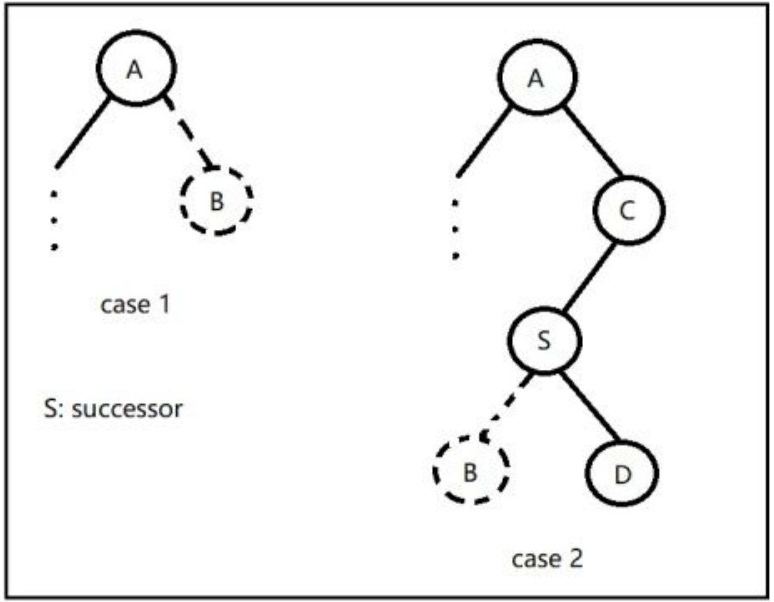
InsertAfter(A, B):

1. If A has no right child, then insert B as the right child of A.
2. Otherwise, find the successor of A and insert B as the left child of the successor.

Problem 4: Order Maintenance

InsertAfter(A, B):

1. If A has no right child, then insert B as the right child of A.
2. Otherwise, find the successor of A and insert B as the left child of the successor.



Problem 4: Order Maintenance

InsertBefore(A, B):

When inserting B before A, do the same thing in reverse:

1. Insert B as the left child (if none exists)
2. Insert B as the right child of the predecessor of A.

Problem 4: Order Maintenance

IsAfter(A, B):

Problem 4: Order Maintenance

IsAfter(A, B):

1. Walk up the tree, until we find a common ancestor of A and B (i.e. when their paths “meet”)

Problem 4: Order Maintenance

IsAfter(A, B):

1. Walk up the tree, until we find a common ancestor of A and B (i.e. when their paths “meet”)
2. Will need to store each step along the path (in an array), the key and whether the node was entered from the left or right.

Problem 4: Order Maintenance

IsAfter(A, B):

1. Walk up the tree, until we find a common ancestor of A and B (i.e. when their paths “meet”)
2. Will need to store each step along the path (in an array), the key and whether the node was entered from the left or right.
3. Compare the two tree walks and find the common ancestor where one path entered from the left and the other from the right.

Problem 4: Order Maintenance

Cost of all operations:

Problem 4: Order Maintenance

Cost of all operations: O(log n), since the height of an AVL tree of n nodes is at most O(log n)

Problem 5: Ancestor Queries

Our job is now to simulate a binary tree. Each node has zero, one, or two children, and the tree is of height h. Unfortunately, it is not a balanced tree. By preprocessing the binary tree, design and implement an auxiliary data structure to support the following operations efficiently:

1. InsertLeft(x, y): insert y as a left child of x in the binary tree.
2. InsertRight(x, y): insert y as a right child of x in the binary tree.
3. IsAncestor(x, y): is x an ancestor of y in the binary tree that contains them?

Problem 5: Ancestor Queries

Hint: Think about how you answer the IsAncestor(x, y) query without the extra data structure. What would the cost of that operation be? How can you improve this?

Problem 5: Ancestor Queries

Change way of traversal?

Problem 5: Ancestor Queries

New type of traversal in which each node in the tree appears twice:

Once before all of its children and once after all of its children.

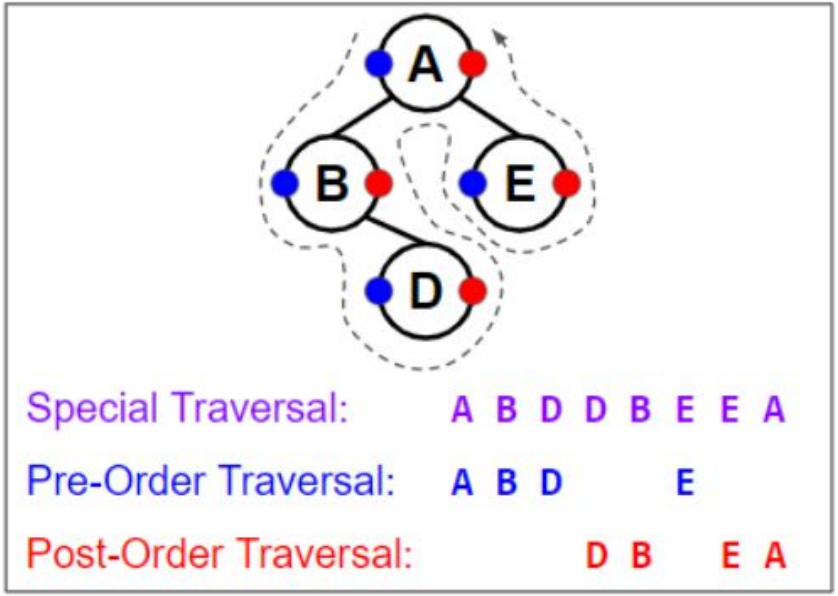
For a node v with children y and z, we define traverse(v) recursively as follows:

print(v) traverse(y) traverse(z) print(v)

Problem 5: Ancestor Queries

For a node v with children y and z, we define traverse(v) recursively as follows:

print(v) traverse(y) traverse(z) print(v)



Problem 5: Ancestor Queries

For a node v with children y and z, we define traverse(v) recursively as follows:

print(v) traverse(y) traverse(z) print(v)

If v has no children, then its traversal consists of just two elements: v v.

Problem 5: Ancestor Queries

For a node v with children y and z, we define traverse(v) recursively as follows:

print(v) traverse(y) traverse(z) print(v)

If v has no children, then its traversal consists of just two elements: v v.

Problem 5: Ancestor Queries

Let's consider the sequence that we obtain out of this traversal. We'll call the first time a node is printed as start(v), and the second time end(v).

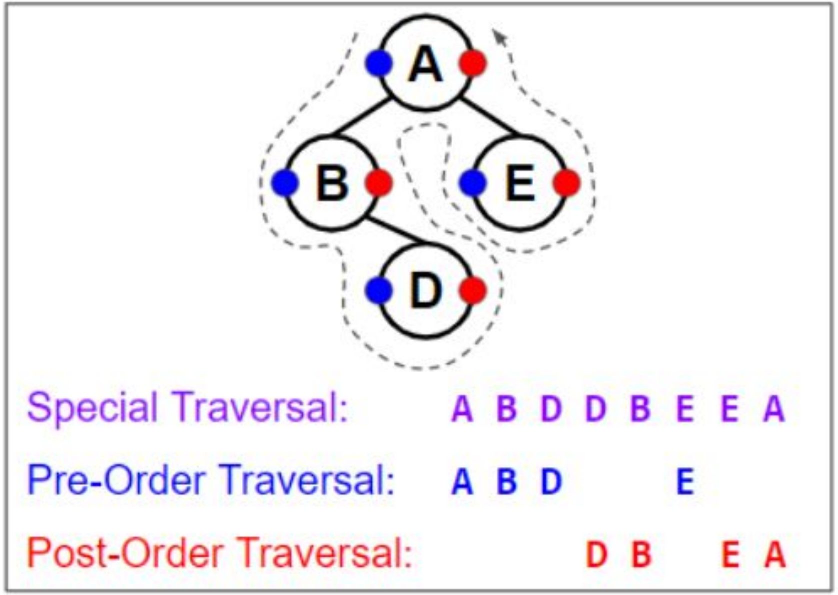
Problem 5: Ancestor Queries

Let’s consider the sequence that we obtain out of this traversal. We’ll call the first time a node is printed as start(v), and the second time end(v).

Focusing only on the first printings: resulting total order is exactly a pre-order traversal

Focusing only on the second printings, resulting total order is exactly a post-order traversal

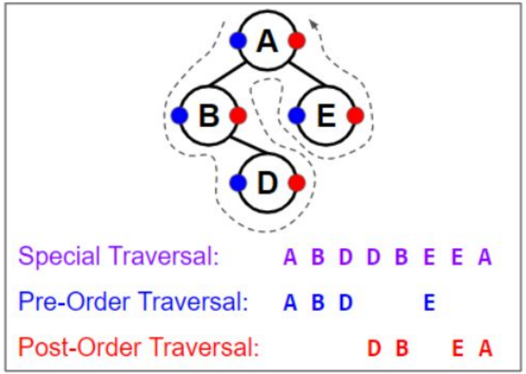
Problem 5: Ancestor Queries



Problem 5: Ancestor Queries

Given two nodes v and w, we observe that v comes before w in a pre-order traversal of the original (unbalanced) tree if start(v) comes before start(w) in the traversal.

Similarly, v comes before w in a post-order traversal of the original (unbalanced) tree if end(v) comes before end(w) in the traversal.



Problem 5: Ancestor Queries

IsAncestor(x,y):

- Check whether x precedes y in the pre-order traversal and whether y precedes x in the post-order traversal.

Problem 5: Ancestor Queries

IsAncestor(x,y):

- Check whether x precedes y in the pre-order traversal and whether y precedes x in the post-order traversal.

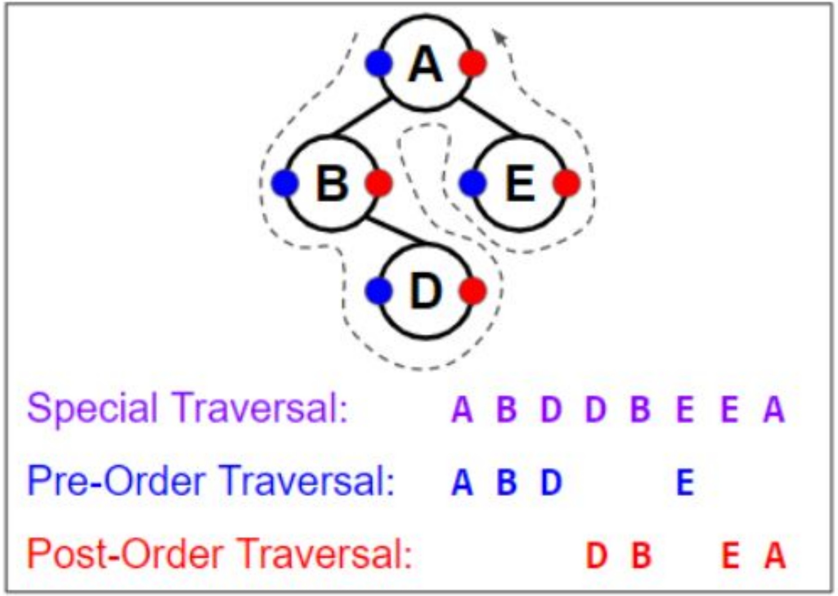
Lemma:

Node x is an ancestor of y if and only if x comes before y in a pre-order traversal and x comes after y in a post-order traversal.

Problem 5: Ancestor Queries

Lemma:

Node x is an ancestor of y if and only if x comes before y in a pre-order traversal and x comes after y in a post-order traversal.



Looking at BDDB here, B is an ancestor of D

Problem 5: Ancestor Queries

Insertions:

To insert a new node w as a child of x, we need to insert start(w) and end(w)

Problem 5: Ancestor Queries

Insertions:

To insert a new node w as a child of x, we need to insert start(w) and end(w)

If w is the only child of x, or w is the left child of x, then we insert start(w) after x and end(w) after start(w).

Otherwise, if w is the right child of x, then we insert end(w) immediately before end(x) and we insert start(w) immediately before end(w).

Problem 5: Ancestor Queries

So what kind of operations we need to support?

Problem 5: Ancestor Queries

So what kind of operations we need to support?

Given 2 items, we need to be able to check whether one is before the other, and also, we need to be able to insert items as either directly before or directly after them.

Problem 5: Ancestor Queries

So what kind of operations we need to support?

Given 2 items, we need to be able to check whether one is before the other, and also, we need to be able to insert items as either directly before or directly after them.

Exactly the previous question! Qn 4: Order Maintenance

Problem 5: Ancestor Queries

So what kind of operations we need to support?

Possibly maintain some additional index structures, e.g., a tree to translate the name of a node to its location in the new traverse data structure, and/or the name of a node to its location in the unbalanced binary tree. This tree would allow us to lookup x, find start(x) and end(x) in the tree, and use them to answer the IsAncestor(x,y) query.

Problem 5: Ancestor Queries

So what kind of operations we need to support?

Possibly maintain some additional index structures, e.g., a tree to translate the name of a node to its location in the new traverse data structure, and/or the name of a node to its location in the unbalanced binary tree. This tree would allow us to lookup x, find start(x) and end(x) in the tree, and use them to answer the IsAncestor(x,y) query.

- O(log n)

Problem 5: Ancestor Queries

So what kind of operations we need to support?

Can use hashing (taught later in the course) for O(1) time instead