Storing a Binary Tree in an Array

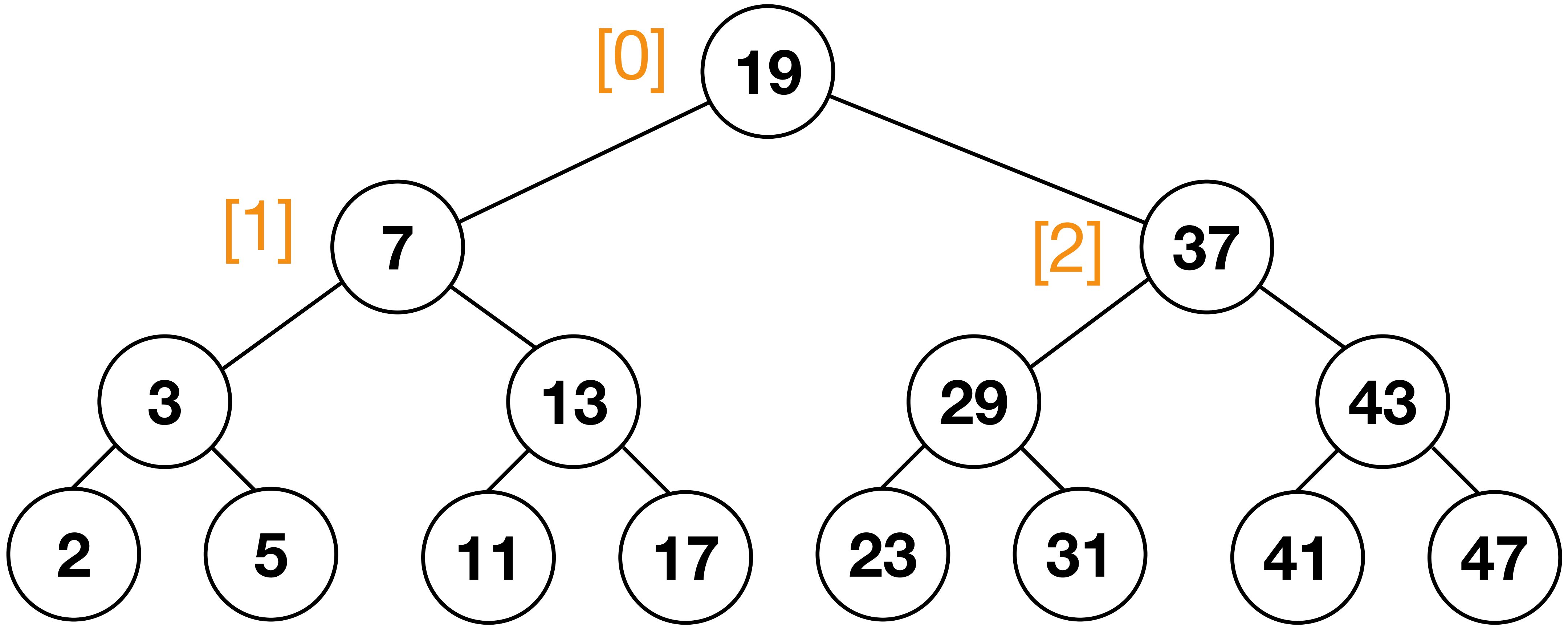
- We can store a list of elements either in a linked structure or an array. Similarly, we can store a binary tree in a linked structure (what we've learned so far), or we can store it in an array too!

- As you will see, the array representation has a pre-determined indexing scheme so it eliminates the the left and right child pointers. **This saves memory space**!

- The downside is that it's not very flexible and if the tree is very sparse you will waste a lot of memory.

Storing a Binary Tree in an Array

Imagine we have a full tree as below. We store the nodes into an array in **breadth-first order** (i.e. level order), where the root node is stored at index [0], its left child at [1], right child at [2], and so on.

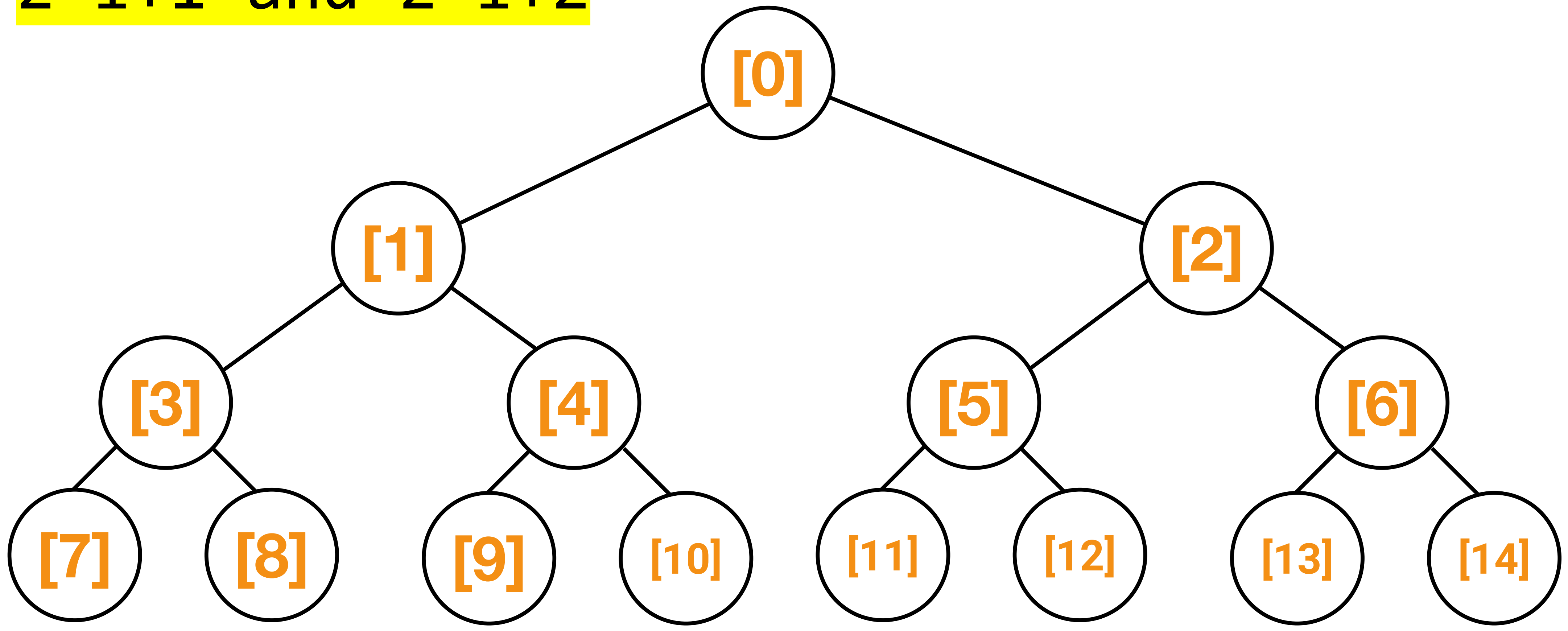


19	7	37	3	13	29	43	2	…	…

Storing a Binary Tree in an Array

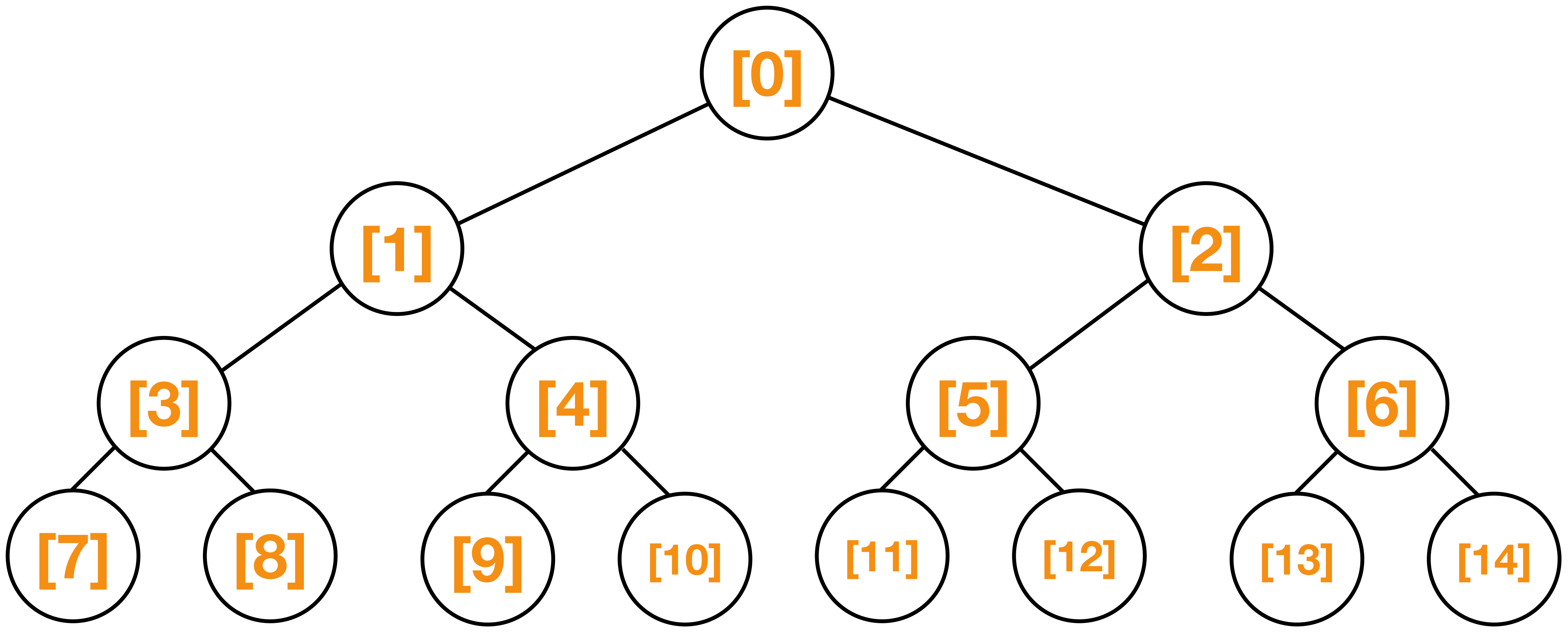
- The index of every node is shown below.
- If a node is at index i, observe from the patterns below, what indices would its two children have?

`2*i+1 and 2*i+2`

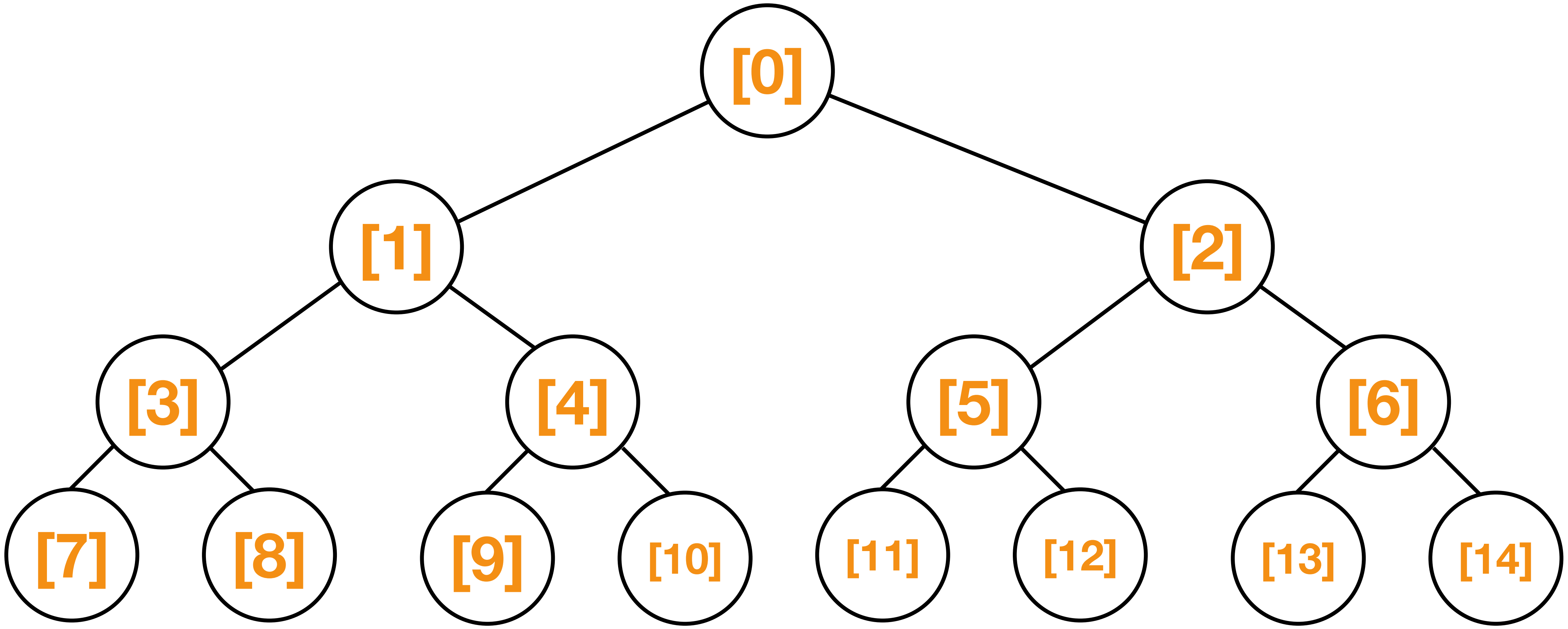


Storing a Binary Tree in an Array

- This indexing scheme is pre-determined, therefore there is no need to store pointers to children, and finding the children of a node involves just some arithmetic calculations (i.e. 2*i+1 and 2*i+2).



Clicker Question #1



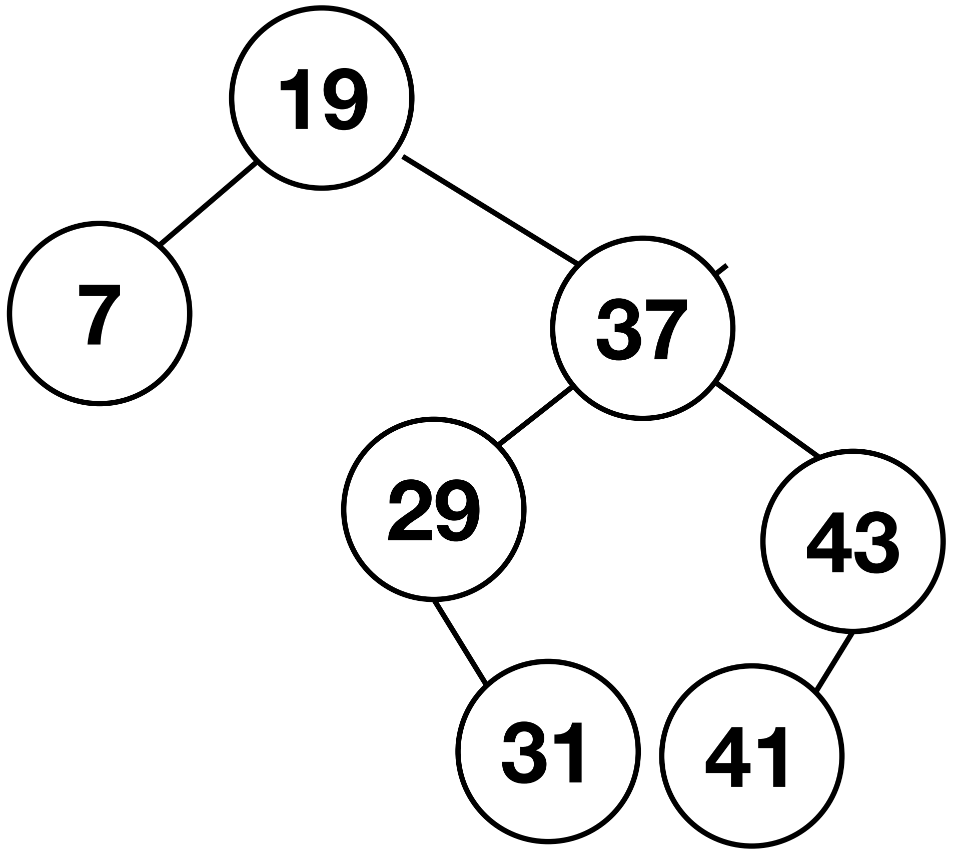
- If a node (not root) is at index `i`, what would be the index of its parent? (Assume integer division)

```
(a)  i/2        (b) (i+1)/2    (c) (i-1)/2

(d)  (i/2)+1    (e) (i/2)-1
```

Storing a Binary Tree in an Array

If a node does not exist, its corresponding array element will be `null`. If the tree is very sparse, a lot of elements will be `null`, in this case the array representation can waste a lot of memory storing `null` pointers.



19	7	37	null	null	29	43	null	null	null	null	null	31	41	null

Clicker Question #2

- What's the maximal difference in the number of null element stored in the array if we use array to represent a BST contains only three element

(a) 2

(b) 3

(c) 4

(d) 5

(e) 6

From Last Lecture: BST

- BST is a binary tree with a special ordering property.
- In-order traversal of BST will visit all nodes in ascending order.
- Searching, insertion, deletion costs are all bounded by **`O(h)`** where h is the tree height.
- If the tree is balanced, the height is approximately `log(N)`, therefore searching, insertion, deletion all cost **`O(log(N))` for a balanced BST**.

Comparing BST to Linear List

Assume N elements stored in each and the BST is balanced

	BST (balanced)	Sorted Array	Linked List
Find	O(log N)	O(log N)	O(N)
Insert			
Find			
Process			
Total			
Remove			
Find			
Process			
Total			

Comparing BST to Linear List

Assume N elements stored in each and the BST is balanced

	BST (balanced)	**Sorted Array**	**Linked List**
Find	O(log *N*)	O(log *N*)	O(*N*)
Insert			
Find		O(log *N*)	
Process		O(*N*)	
Total		O(*N*)	
Remove			
Find		O(log *N*)	
Process		O(*N*)	
Total		O(*N*)	

Comparing BST to Linear List

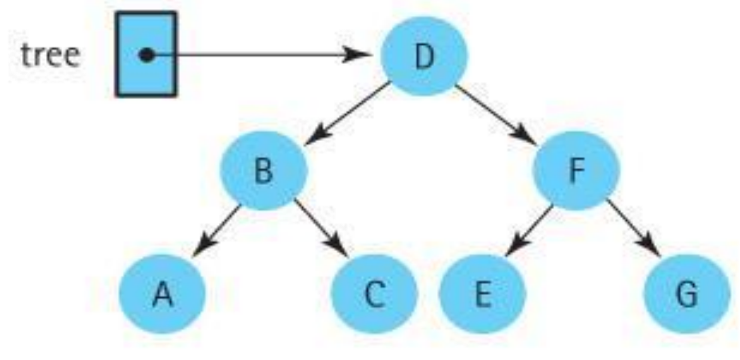
Assume N elements stored in each and **the BST is balanced**

	BST (balanced)	**Sorted Array**	**Linked List**
Find	O(log *N*)	O(log *N*)	O(*N*)
Insert			
Find	O(log *N*)	O(log *N*)	O(*N*)
Process	O(1)	O(*N*)	O(*1*)
Total	O(log *N*)	O(*N*)	O(*N*)
Remove			
Find	O(log *N*)	O(log *N*)	O(*N*)
Process	O(1)	O(*N*)	O(1)
Total	O(log *N*)	O(*N*)	O(*N*)

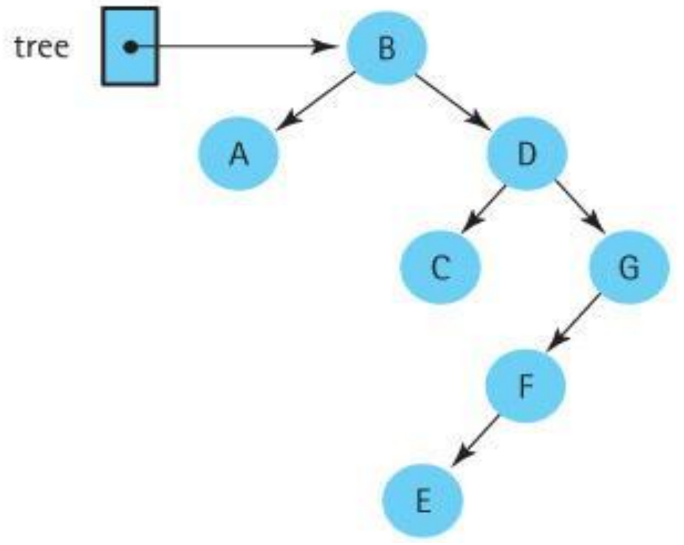
What is Balance?

- There are many ways to define **balance.** For example, standing at any node, we can define balance as the difference between the height of its right subtree vs. the height of the left subtree.

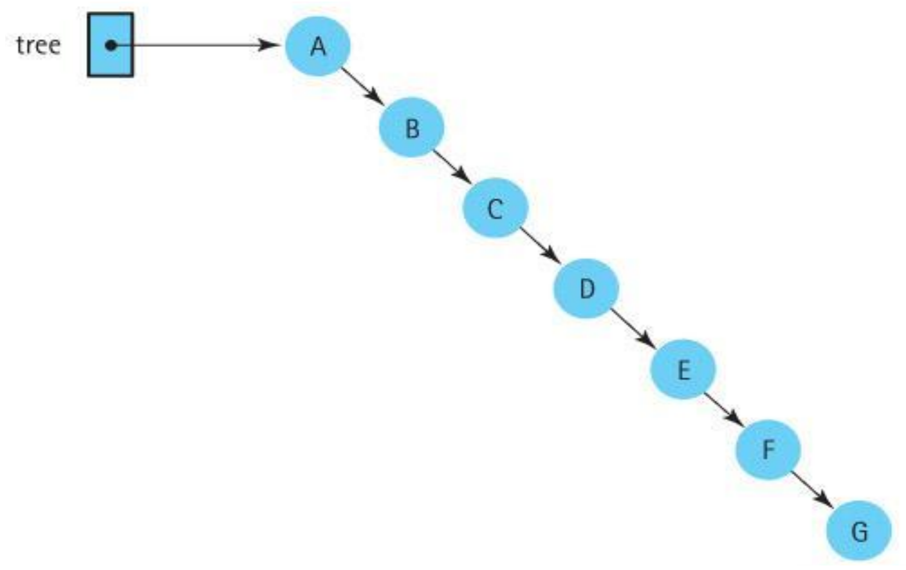
```
balance(n)=|height(n.right)-height(n.left)|
```



perfectly balanced



Not very balanced



Extremely unbalanced

What is Balance?

- Another way to think about balance is to look at how the tree height `h` is related to the number of nodes `N`. Ideally we want to **guarantee that `h = O(log(N))`**.
- Because insertion and deletion can cause the BST to become unbalanced, we need to perform extra steps to restore the balance (preserve the height guarantee).
- Such trees are called **self-balancing trees** (a.k.a. **height-balanced trees**).

Examples Self-Balancing Trees

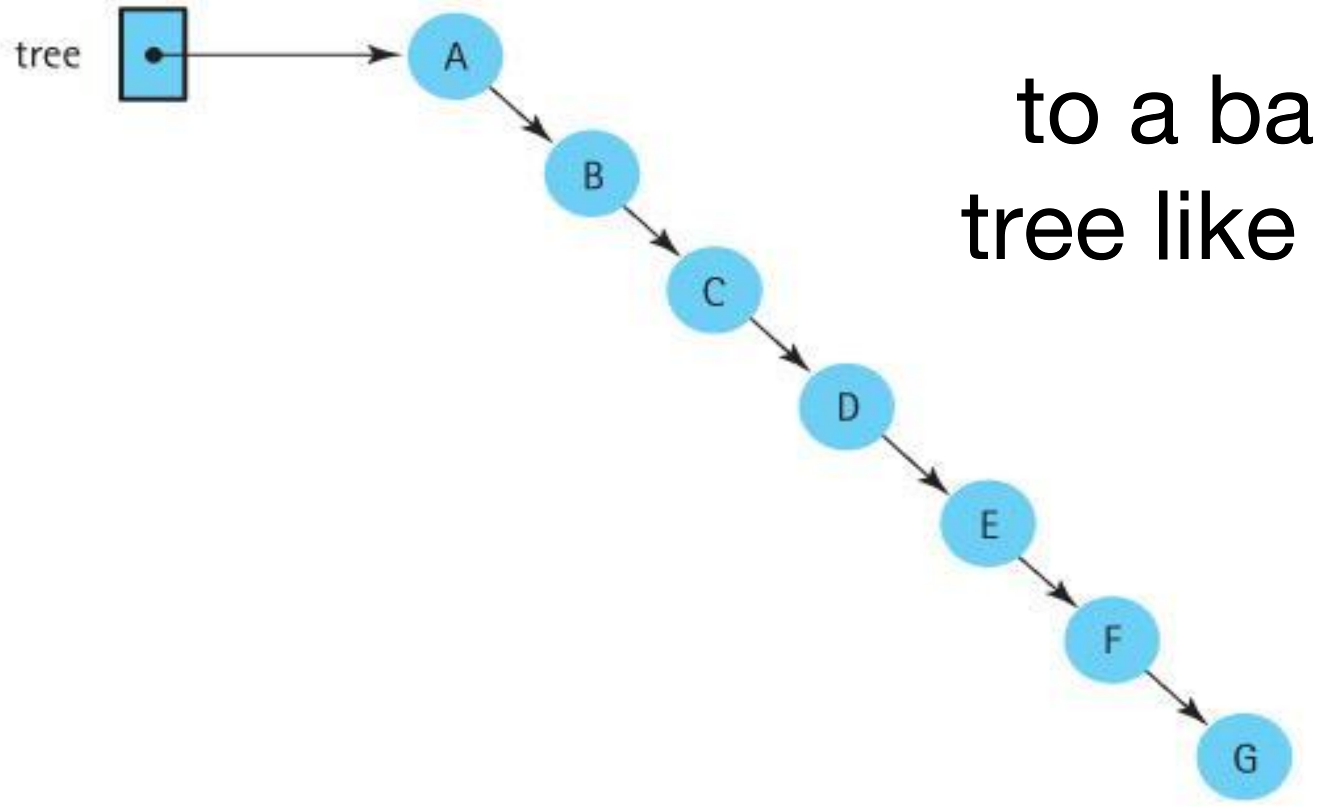
- **AVL tree:** guarantees that at any node, the height difference between its left and right subtrees is no more than 1.
- **Red-black tree:** colors each node with red/black (requires extra bit per node), and enforces color compatibility rules during insertion / deletion to maintain height guarantee.
- **Scapegoat tree:** when tree becomes unbalanced, find a ‘scapegoat’ and re-balances the sub-tree rooted at the scapegoat.
- **2-3 tree** (not binary tree)**:** each interior node either stores 1 data element and has 2 children, or stores 2 data elements and has 3 children.

First Approach to Balancing

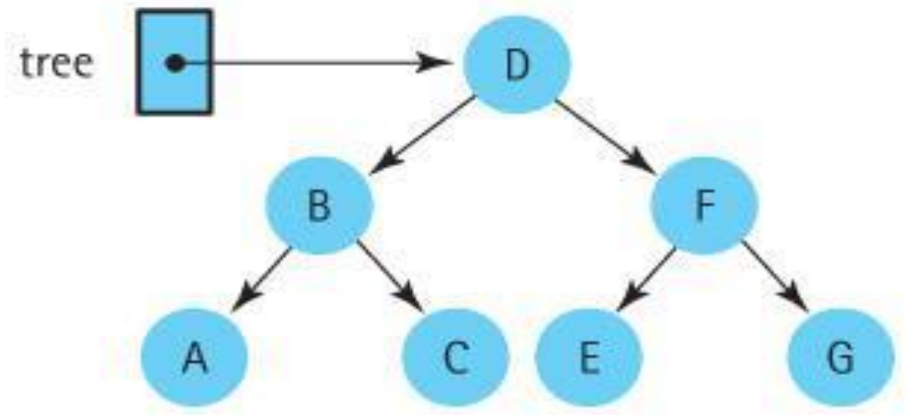
- The user manually calls a **`balance()`** method from time to time (at the user's discretion) to rebuild the tree into a balanced version.
- This is NOT done automatically so it doesn't qualify as a self-balancing tree.
- The `balance()` method is useful for some self-balance trees.

Manual Balancing

- We want to rebuild a tree like this:



to a balanced tree like this —>



- To do so, we will:
  1. Output the elements into a sorted array
  2. Build a new, balanced tree from the sorted array

Manual Balancing

- How do we output all elements in a BST into an array in sorted (ascending) order?

  Do an **in-order traversal** and save the result to an array (or a queue if you want)!

- How do we build a **balanced** tree from a sorted array? For example, if the sorted array is:

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

It's similar to a binary search: you start from the entire array, pick the middle element to make it a tree node, then recurse on the left and right sub-arrays.

```
// jump start the recursion by:
// root = sortedArray2BST(0, array.length-1);

BSTNode<T> sortedArray2BST(int lower, int upper) {
    if (lower > upper)
        return null;
    int mid = (low + high) / 2;
    BSTNode<T> node = new BSTNode<T>(array[mid]);
    node.left = sortedArray2BST(lower, mid - 1);
    node.right= sortedArray2BST(mid + 1, upper);
    return node;
}
```

Clicker Question #3

```
BSTNode<T> sortedArray2BST(int lower, int upper) {
    if (lower > upper) return null;
     int mid = (low + high) / 2;
     BSTNode<T> node = new BSTNode<T>(array[mid]); //1
     node.left = sortedArray2BST(lower, mid - 1);
     node.right= sortedArray2BST(mid + 1, upper);
     return node;
}
```

What's the cost of the above method, if the sorted array has N elements? Remind you that you start the call by:

```
sortedArray2BST(0, N-1);
```

(a) O(log N)

(b) O(N)

(c) O($\log^2$(N))

(d) O(N*log(N))

(e) O(N^2)

Comparing BST to Linear List

Assume N elements stored in each and **the BST is balanced**

	BST (balanced)	**Sorted Array**	**Linked List**
Find	O(log *N*)	O(log *N*)	O(*N*)
Insert			
Find	O(log *N*)	O(log *N*)	O(*N*)
Process	O(1)	O(*N*)	O(*1*)
Total	O(log *N*)	O(*N*)	O(*N*)
Remove			
Find	O(log *N*)	O(log *N*)	O(*N*)
Process	O(1)	O(*N*)	O(1)
Total	O(log *N*)	O(*N*)	O(*N*)

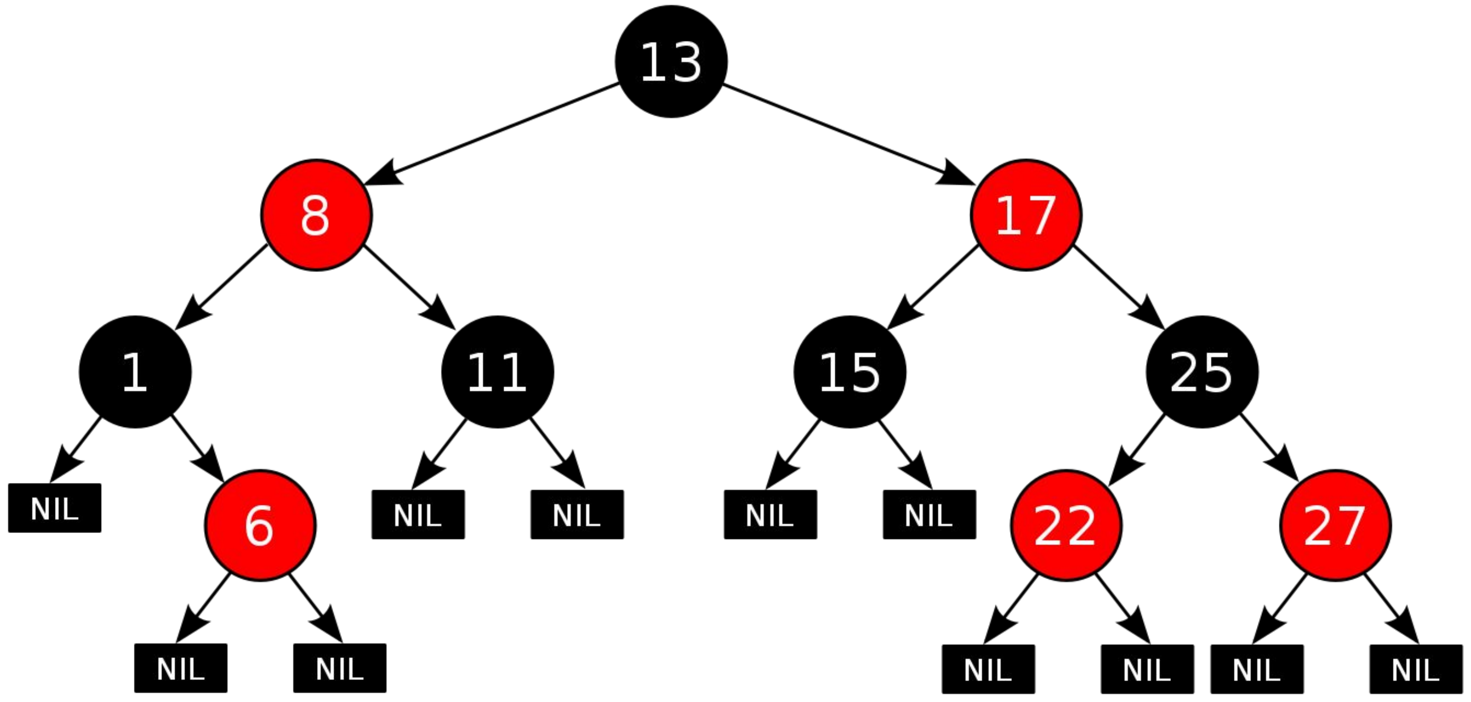
The `balance()` method

- To summarize, the `balance()` method first performs an in-order traversal to output all elements into a sorted array; then it calls the recursive `sortedArray2BST` to build a balanced tree from the sorted array.
  - The cost of this is `O(N)`
- Instead of balancing the whole tree, you can choose to **balance any sub-tree**.

Red-Black Tree

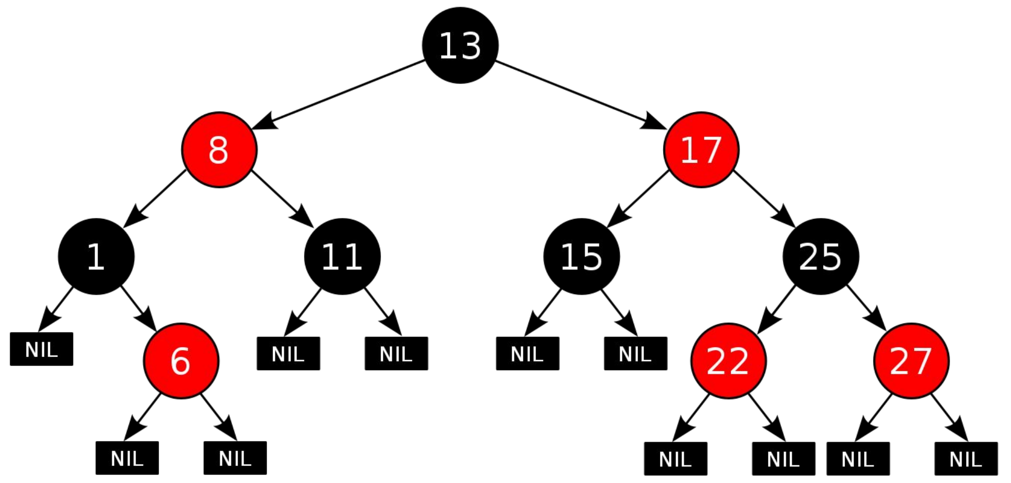
Main idea: add new attribute **color** inside the node to control level of tree

A red-black tree with n internal nodes has height at most 2*log(n+1)



Red-Black Tree

<Rules>



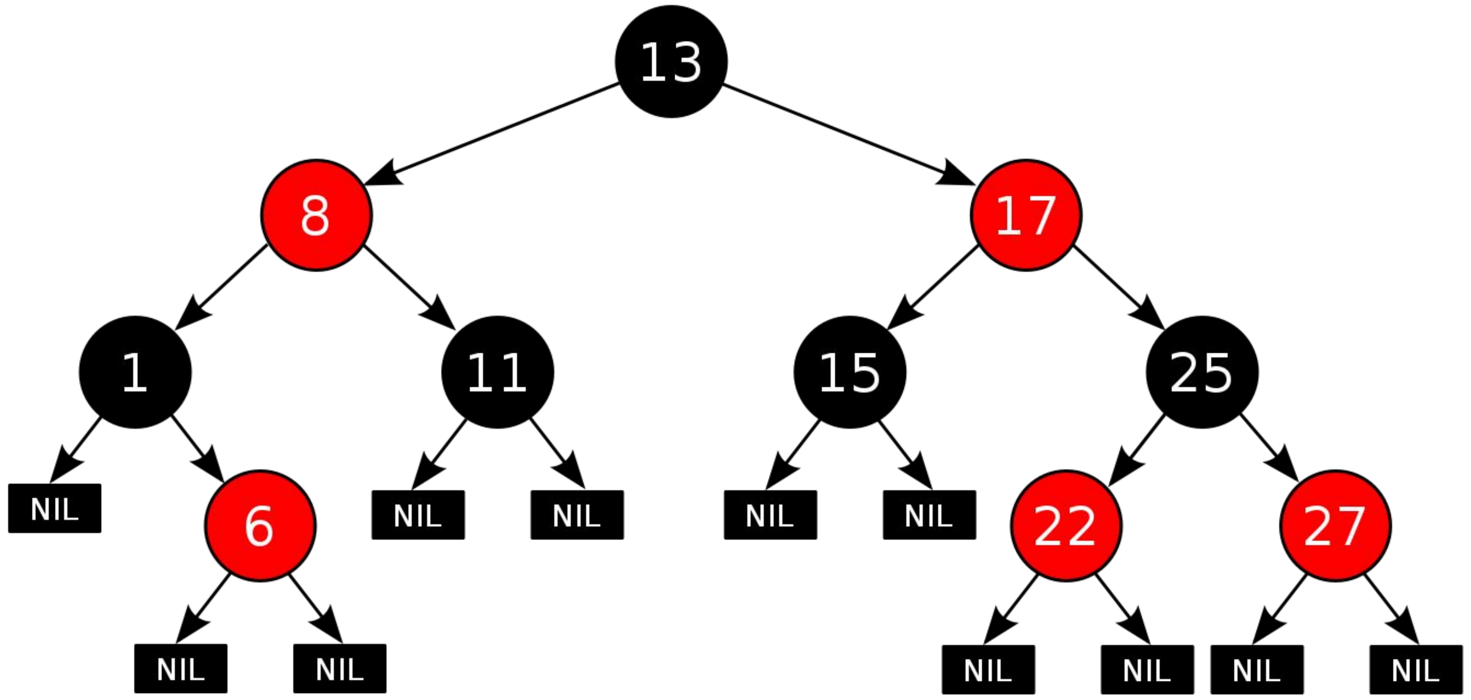
1. Each node is either **red** or **black**.
2. The root is black.
3. All leaves (NIL) are black.
4. If a node is red, then both its children are black.
5. Every **path** from a given node to any of its descendant NIL nodes contains the **same number** of **black** nodes.

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

Red-Black Tree

A red-black tree with n internal nodes has height at most 2*log(n+1)

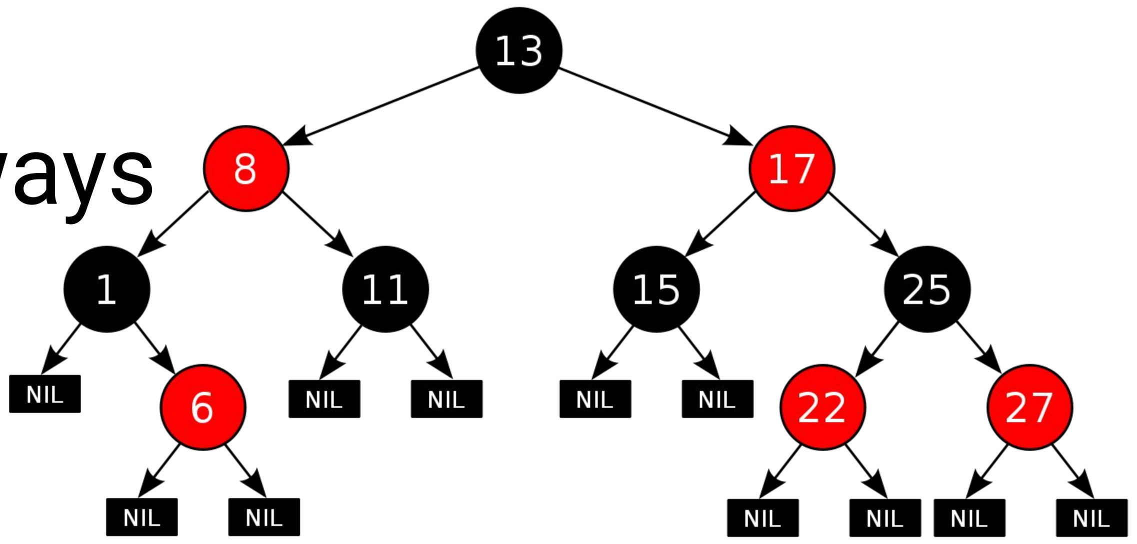
Proof: $n >= 2^{(\text{black height})} - 1$



Red-Black Tree

Grandparent node: the parent node of its parent node. Ex: 13 is the grandparent node of 1, 11, 15, or 25.
Sibling node: the another child node of the parent node. Ex: 17 is the sibling node of 8
Uncle node: the another child node of the grandparent node that is not its parent node. Ex: 8 is the uncle node of 15 and 25.

- In red-black tree, the node always has uncle and sibling since NIL is also a node (leaf node)
- Ex: 6's sibling is NIL



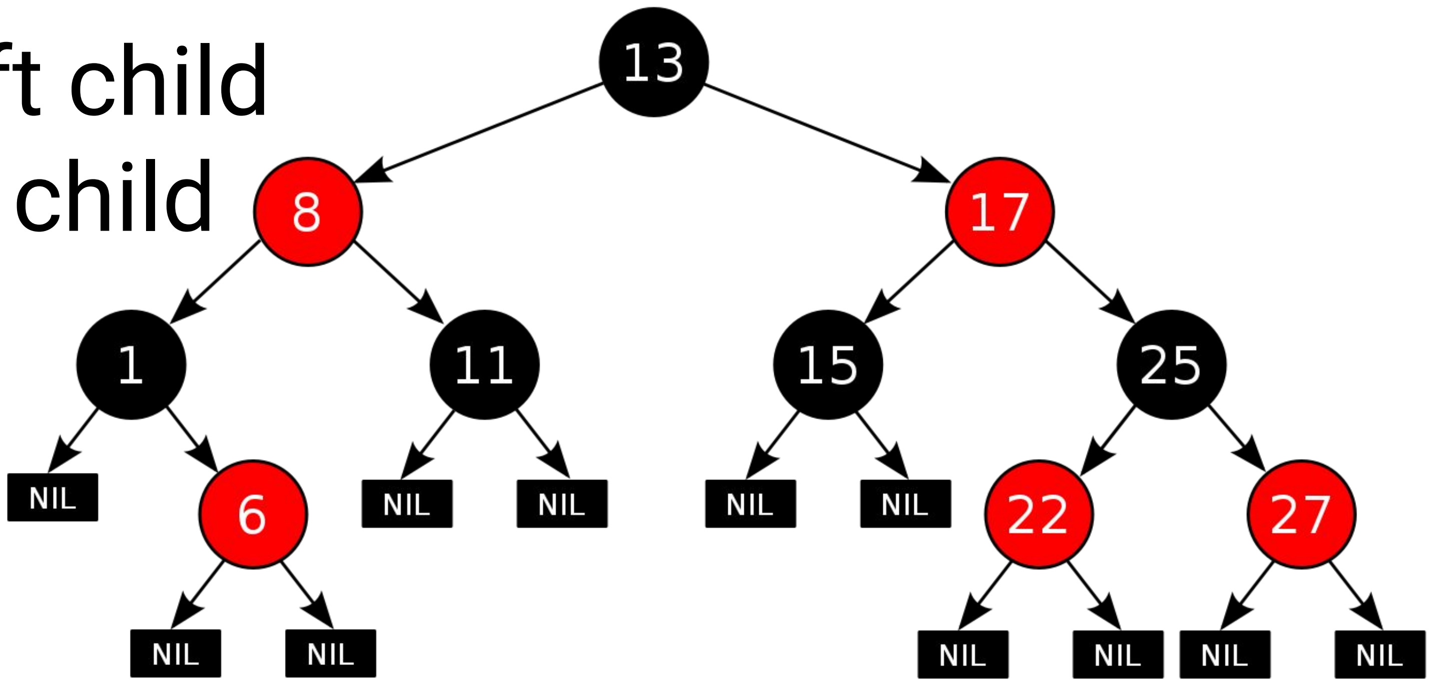
https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

Red-Black Tree

Outside: the node is the left child of left child of grandparent node or the right child of right child of grandparent node. Ex: 27 is outside under its grandparent node 17.

Inside: the node is the left child of right child or the right child of left child of grandparent node.
EX: 11 is inside under
Its grandparent node 13.

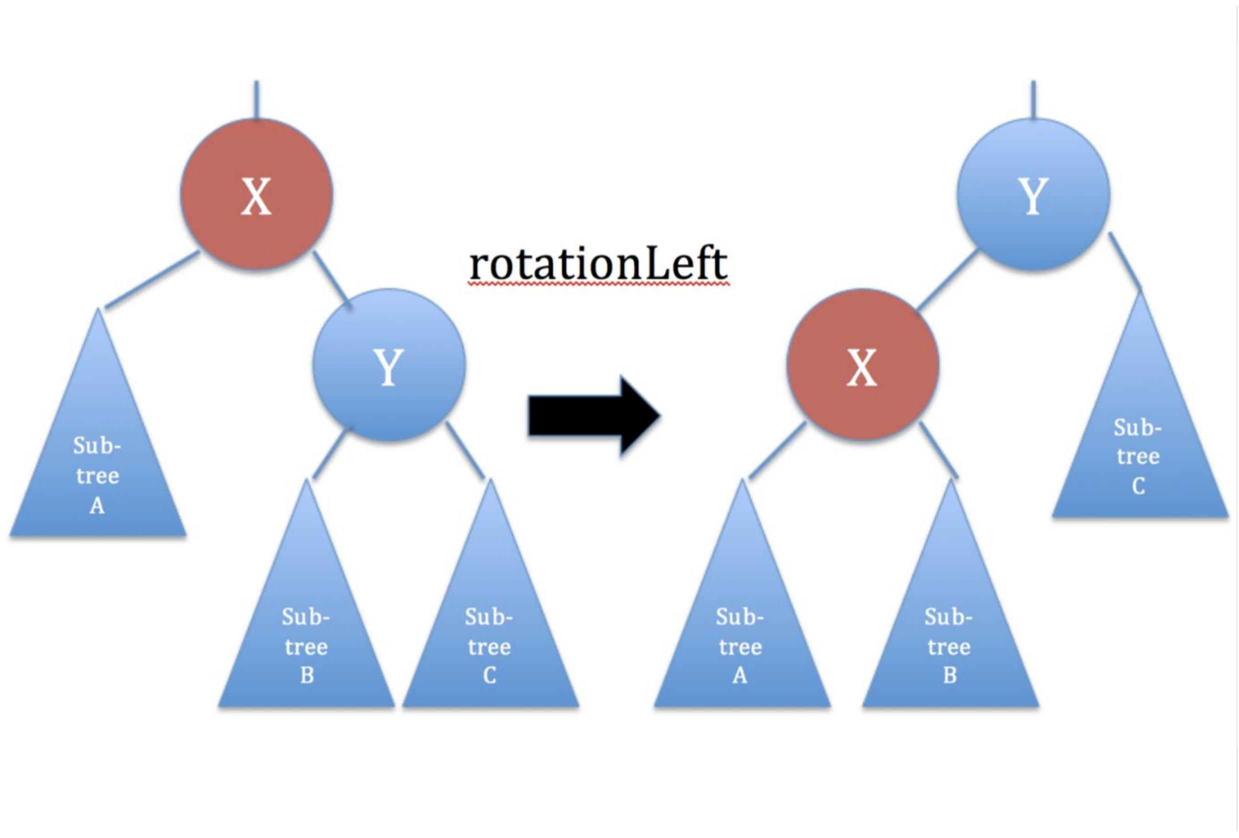
-

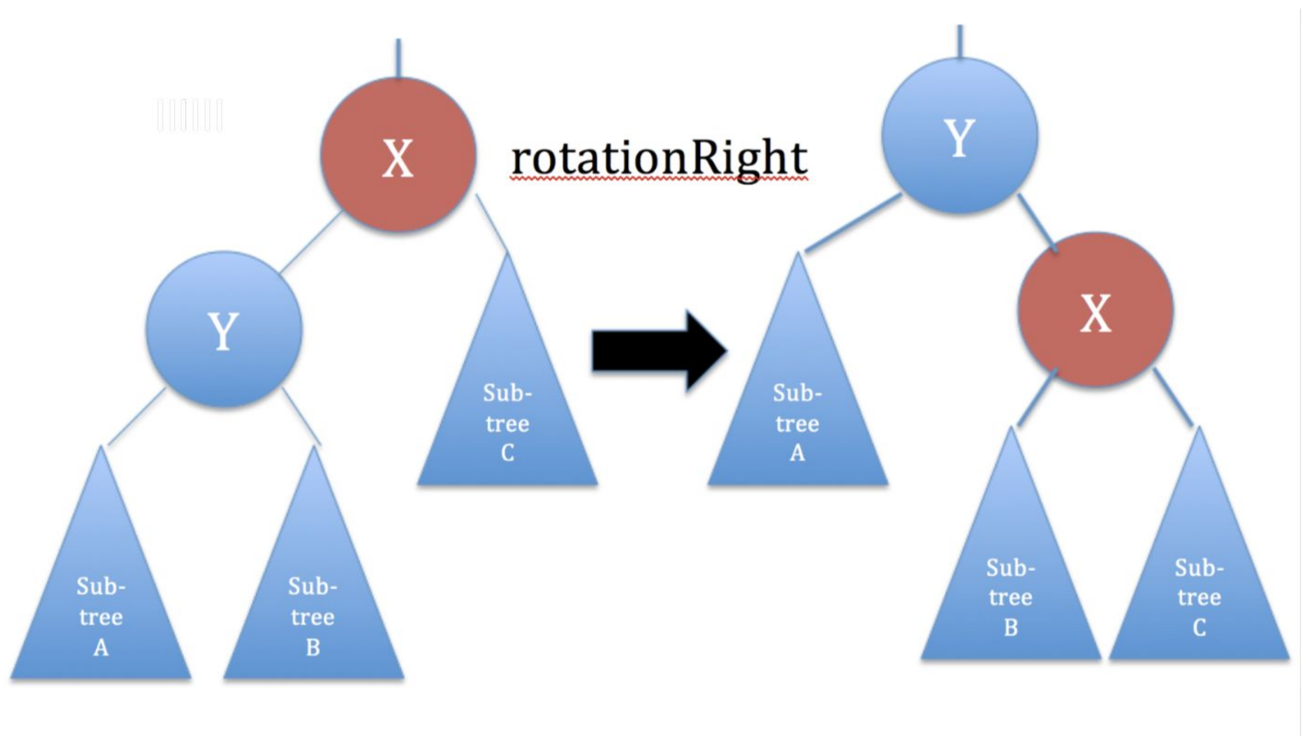


https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

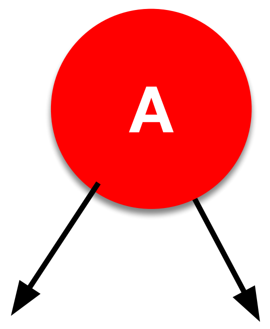
Example: Insertion

Rotation: left rotation and right rotation on X





Example: Insertion



Usual BST insertion

Coloring the new node as red

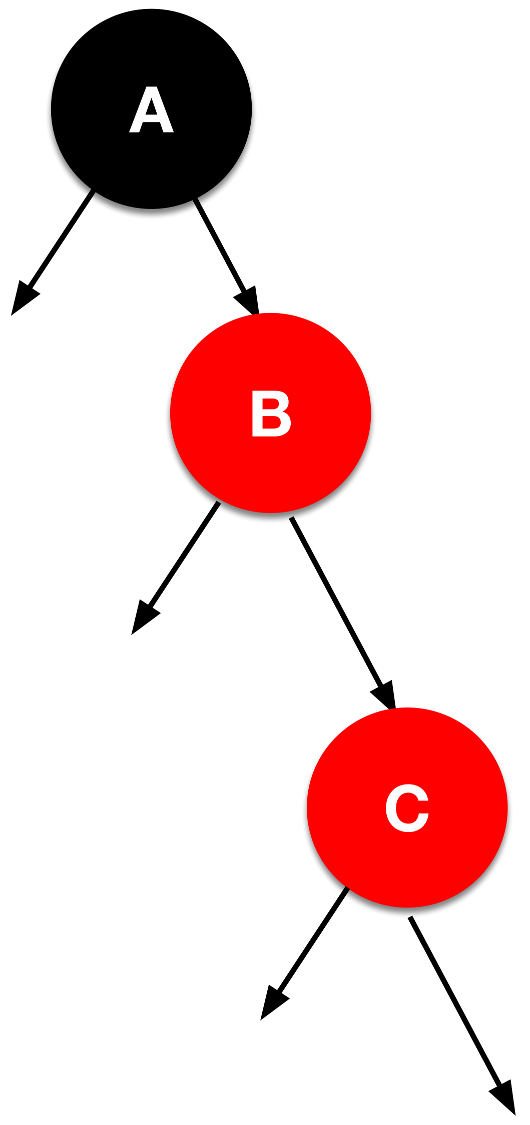
Based on the result to update it

Example: Insertion

Usual BST insertion

Coloring the new node as red

Based on the result to update it

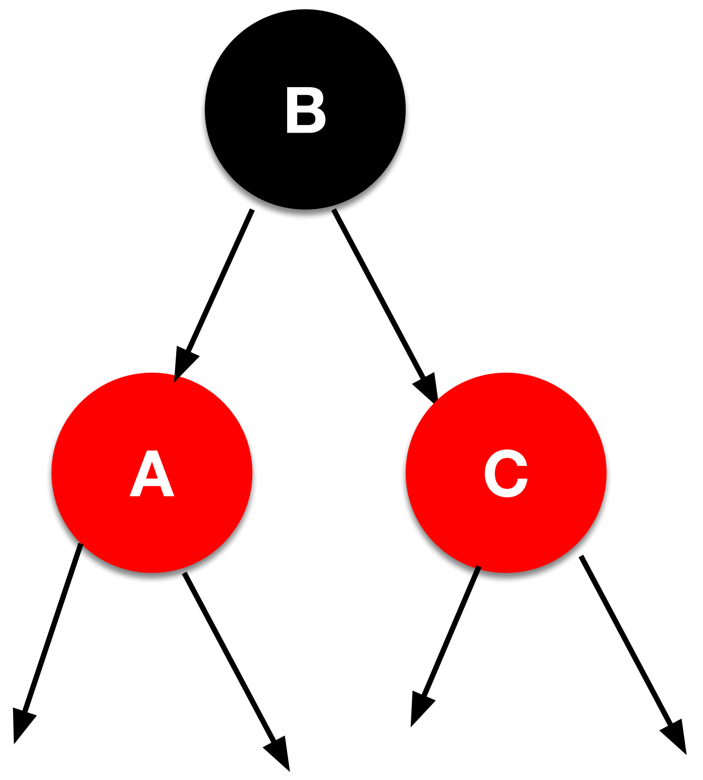


Example: Insertion

Usual BST insertion

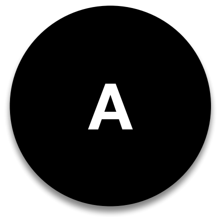
Coloring the new node as red

Based on the result to update it



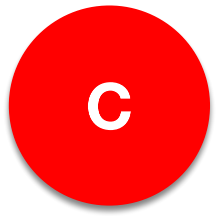
Example: Insertion

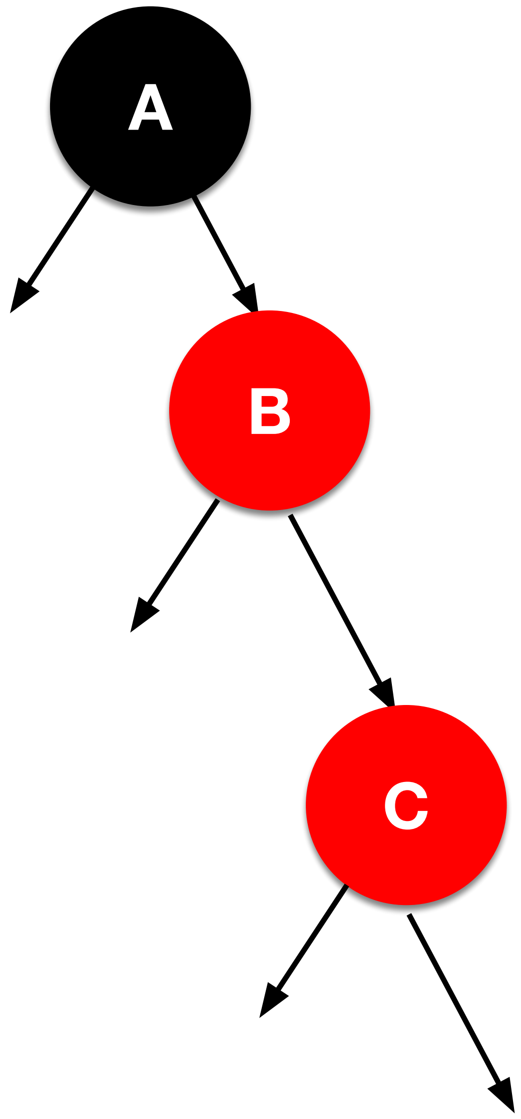
1. If the new node is root, coloring Black



2. If the new node's parent is black, then do nothing

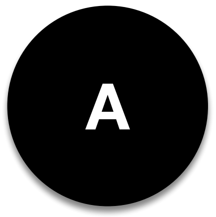
3. If the parent node is red, then need to rebalance.

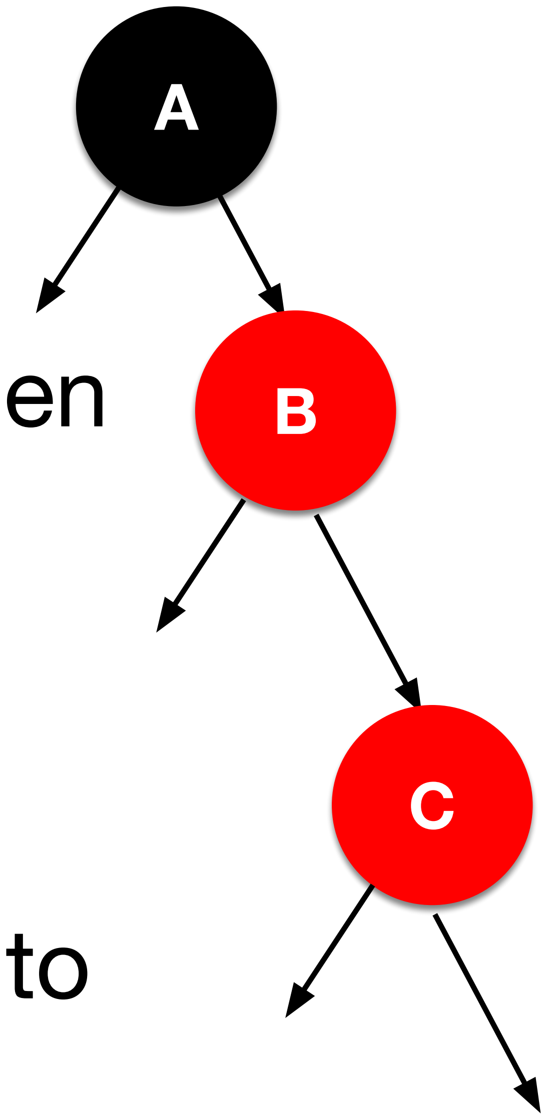




Example: Insertion

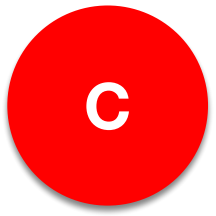
1. If the new node is root, coloring Black





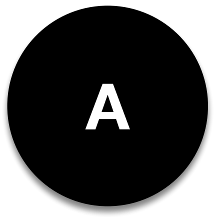
2. If the new node's parent is black, then do nothing

3. If the parent node is red, then need to rebalance



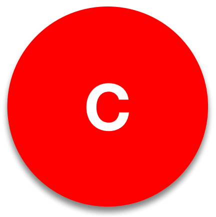
Example: Insertion

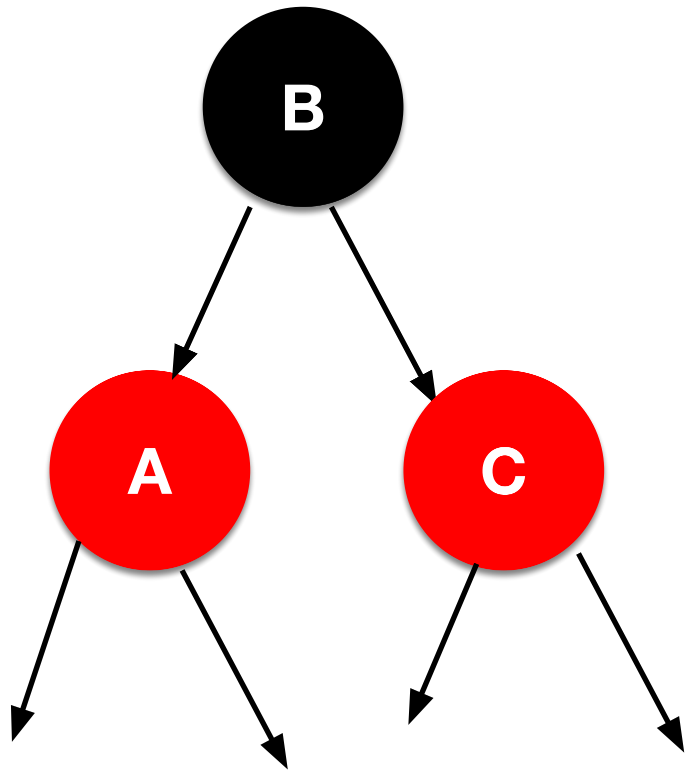
1. If the new node is root, coloring Black



2. If the new node's parent is black, then do nothing

3. If the parent node is red, then need to rebalance

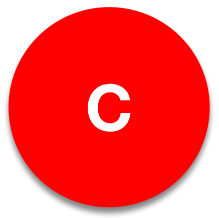




Example: Insertion

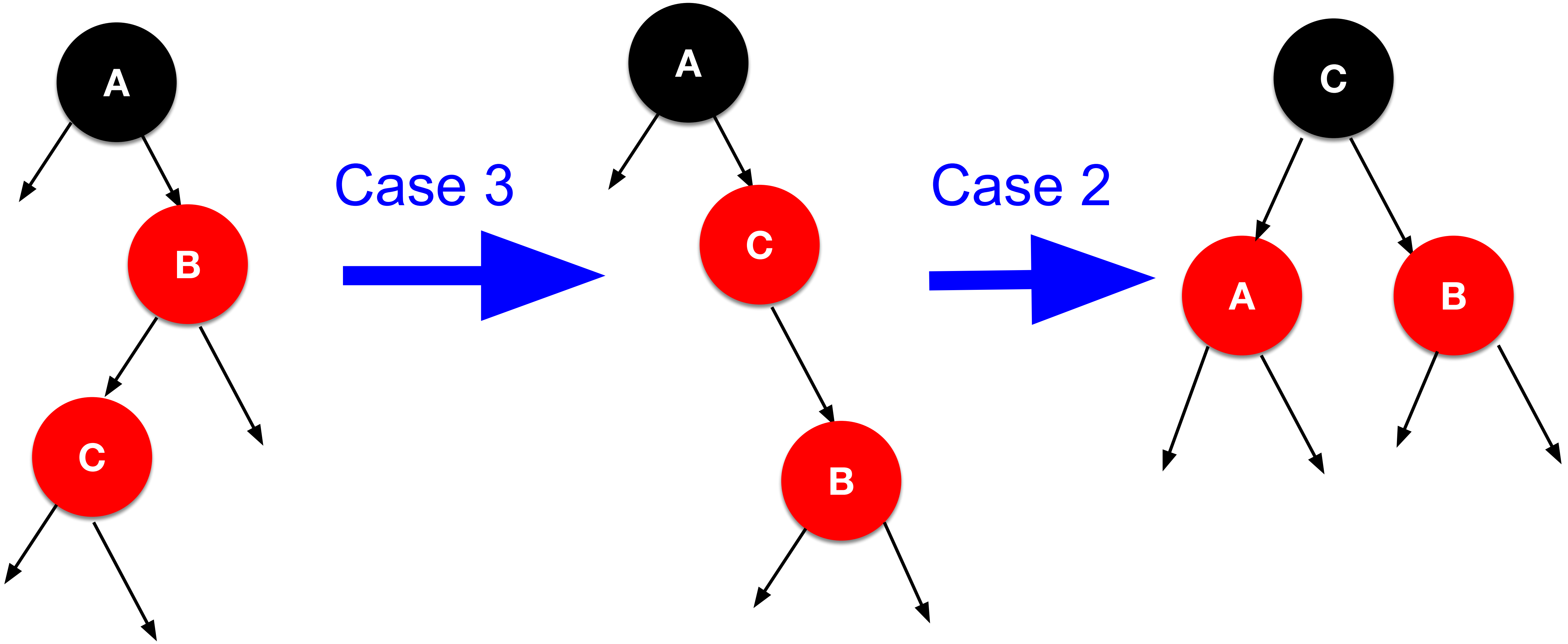
How to rebalance tree when parent node is red?

1. If uncle node is red, we repaint both parent and uncle node to **black**, then repaint grandparent node to **red** and do recursion on grandparent node.
2. If uncle node is black and new node is on "outside" of subtree under grandparent node, we repaint parent node to black and grandparent node to red, then do **rotation** on grandparent node.



3. If uncle node is black and new node is on "inside" of subtree under grandparent node, we do **rotation** on parent node for swapping and do recursion on parent node.

Example: Insertion



Example: Removal

When removing node from BST,

1. If the node has no child, remove it

2. If the node has one child, replace it by the child

3. If the node has two children, replace it by its successor and do recursion on successor node

Then repainting red-black tree if needed.

Example: Removal

When removing node from red-black tree,

If target node’s (you will remove) and its child’s color is different, coloring child node black then remove target node.

If both target node’s and its child’s color is black, then repainting.

Example: Removal

How to repaint when both node and its child (used to replace the node) is black?

1. If the child is new root, color it black and done.

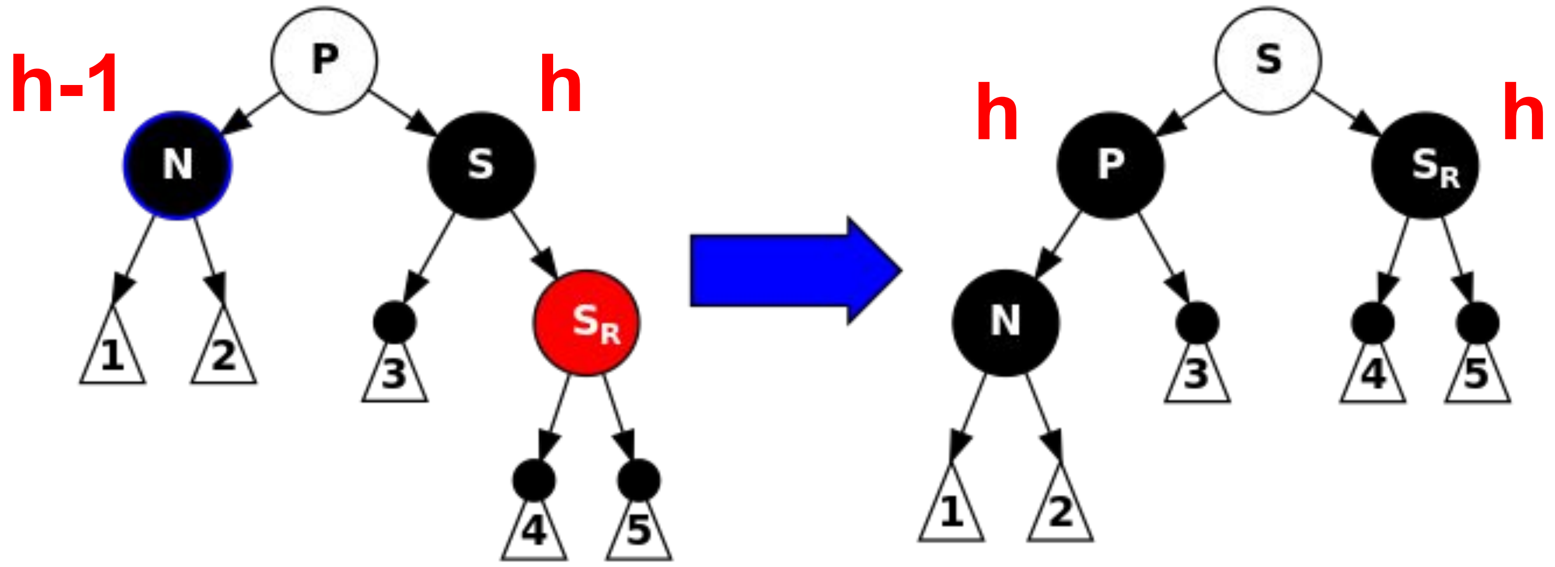
2. If sibling node and its both children are black, coloring sibling node red, then coloring parent node to black if it’s red.

Example: Removal

3. If sibling node is black and at least one of sibling's children is red

   a. if the sibling's child is "outside" of subtree under parent node, do **rotation** on parent node and coloring the child to black, then swap parent's and sibling's color.

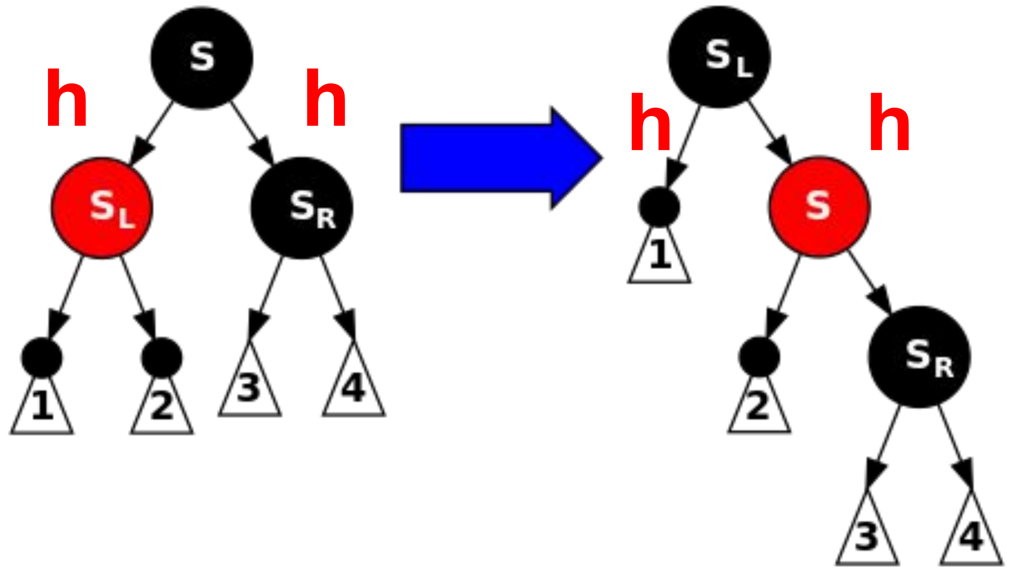
   b. ...



Example: Removal

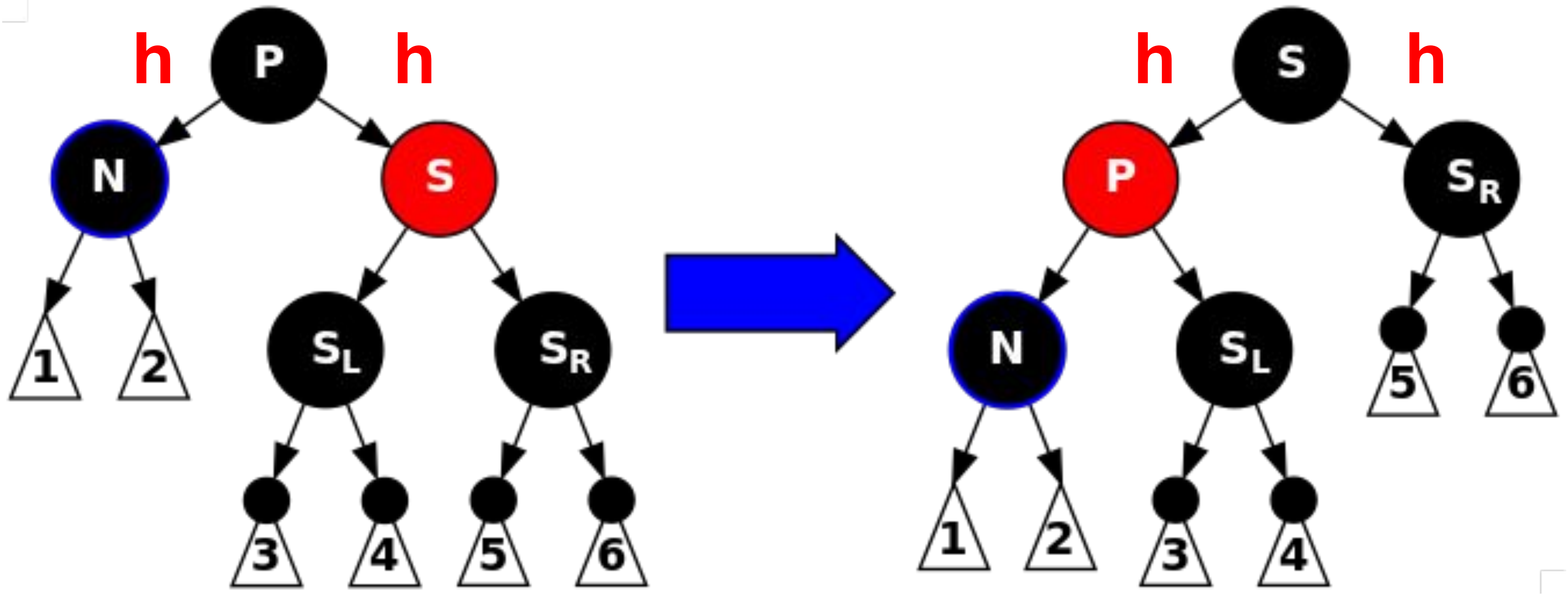
3. If sibling node is black and at least one of sibling's children is red

   a. ...
   b. if the sibling's children is "inside" of subtree under parent node, we swap color between sibling node and it's "inside "child and do **rotation** on sibling node, then do recursion for case 3.a



Example: Removal

4. If sibling node is **red**, change it to black and parent node to red, then do rotation for generating other cases that node has a **black** sibling (case 2 or 3).



Scapegoat Tree

The name comes from the common wisdom that, when something goes wrong, the first thing people tend to do is find someone to blame (the **scapegoat**). Once blame is established, we will demand the scapegoat to fix the problem.

Scapegoat Tree Rules

- Keep track of **N (# of nodes)**, **h (height)**, and **q (upper bound)**.
  - At insertion, q increments for each new node.
  - At deletion, q remains unchanged.
- After **inserting** a new node, check the **height condition**. If violated, find scapegoat and rebuild the **subtree rooted at the scapegoat**
- After **deleting** a node, check the **upper bound condition**, if violated, **rebuild the entire tree** and **reset q to N**

Scapegoat Tree

Main ideas:

Two conditions that must be satisfied at all times:

```
1.  h <= log_{3/2} q        // height cond.
2.  q/2 <= N <= q           // upper bound cond.
```

where N is the # of nodes and **q is called an upper bound** (how it's calculated will become clear later)

- If insertion violates condition 1, find a scapegoat node w, and then call `balance()` to rebuild subtree rooted at w.
- If removal violates condition 2, rebuild the entire tree.

Scapegoat Tree

Main ideas:

Two conditions that must be satisfied at all times:

1. $q/2 <= N <= q$ *// upper bound cond.*
2. $h <= \log_{3/2} q$ *// height cond.*

Why do these conditions help us maintain a self-balancing tree? It's because:

If the two conditions are met at all times, we have

$$h <= (\log_{3/2} q) <= (\log_{3/2} 2N) < (2 + \log_{3/2} N)$$

Thus $h = O(\log N)$ and this is what we wanted.

Example: Insertion

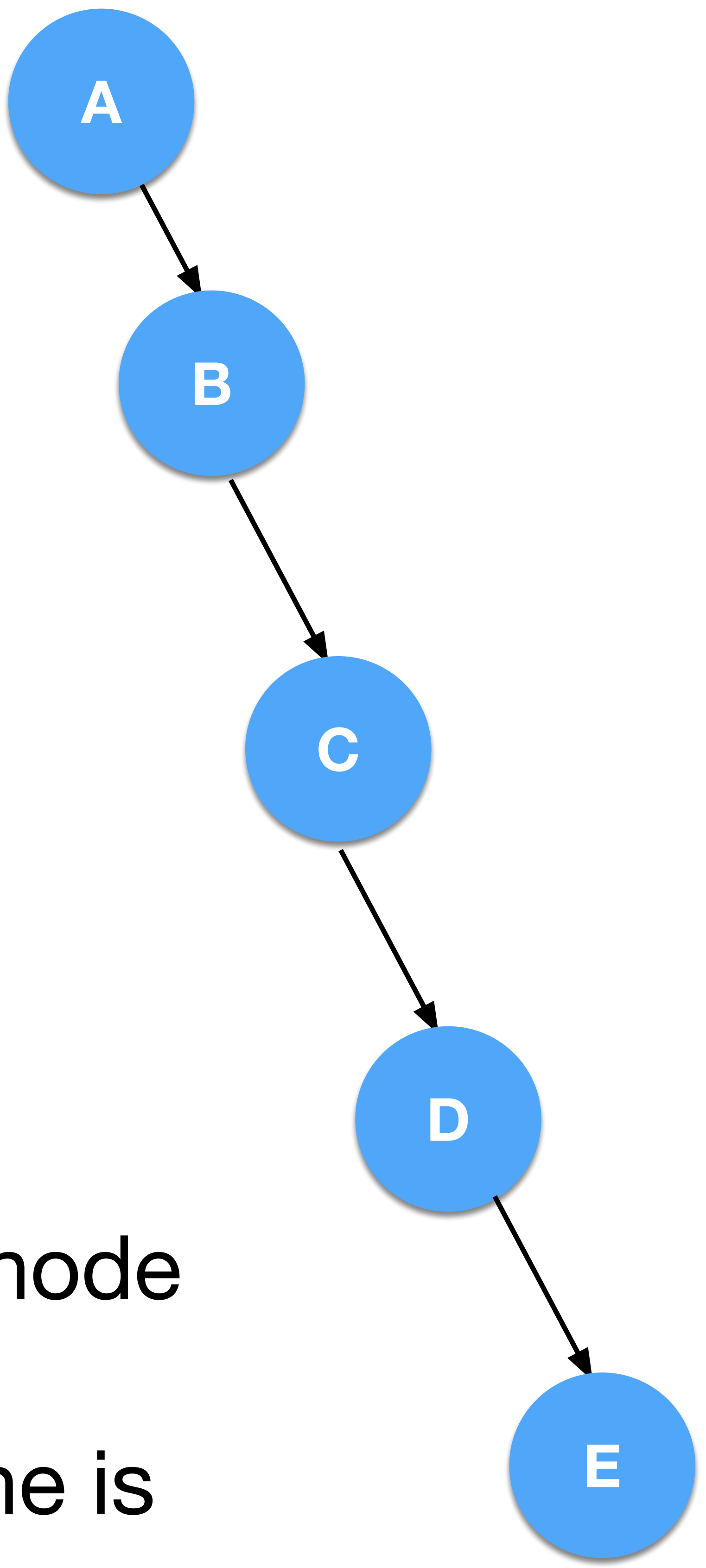
Here we insert 5 elements in ascending order: a worst-case for standard, non-balancing BST.
During insertion, q increments for each new node.

operation	**N**	**q**	**height**	**$\log_{3/2}(q)$**
`insert A`	1	1	0	0
`insert B`	2	2	1	1.79
`insert C`	3	3	2	2.70
`insert D`	4	4	3	3.42
`insert E`	5	5	**4**	**3.97**

- Inserting element E violated height condition (`h <= log`$_{3/2}$ `q`)
- Now find scapegoat node `w` (**on the path from the newly added node up to the root**) such that:

$$\frac{\texttt{sizeOfSubtree(w.child)}}{\texttt{sizeOfSubtree(w)}} > 2/3$$

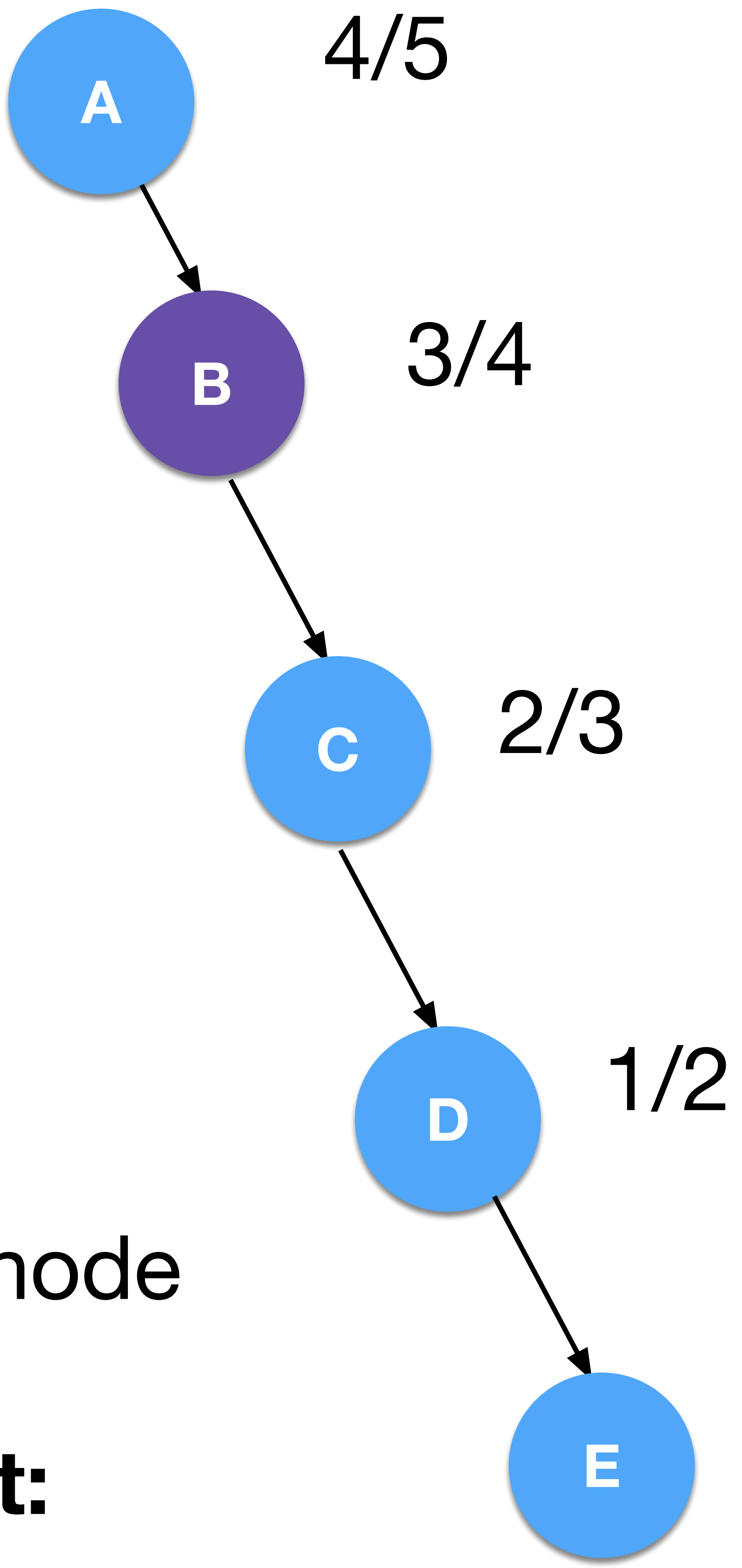
It can be shown that such a scapegoat node must exist (proof can be done using contradiction). In this example, which one is the scapegoat?



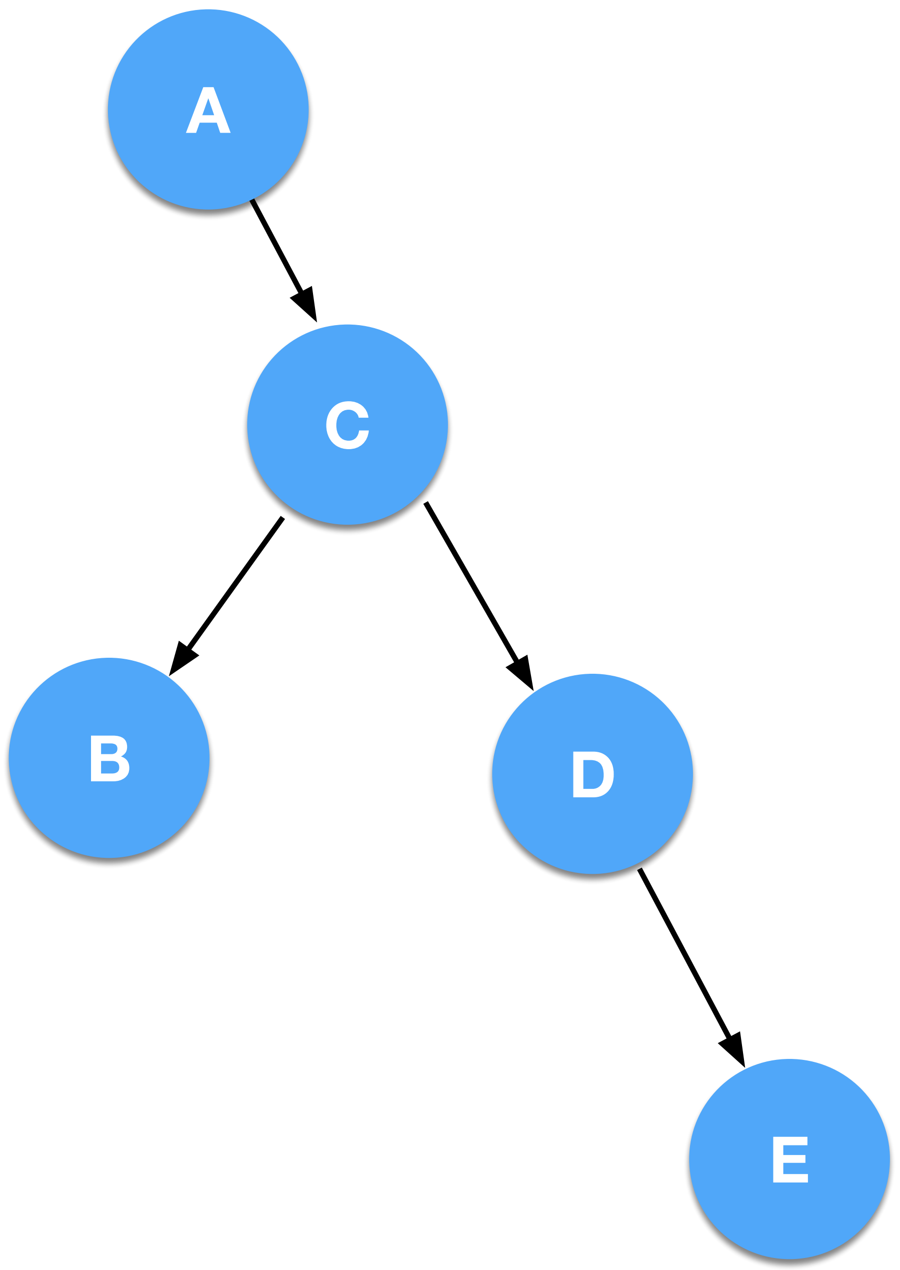
- Inserting E violated height condition (h `<= log`$_{3/2}$ `q`)
- Now find scapegoat node `w` (**on the path from the newly added node up to the root**) such that:

$$\frac{\texttt{sizeOfSubtree(w.child)}}{\texttt{sizeOfSubtree(w)}} > 2/3$$

It can be shown that such a scapegoat node must exist (proof can be done using contradiction). **Here B is the scapegoat: since `3/4 > 2/3`**



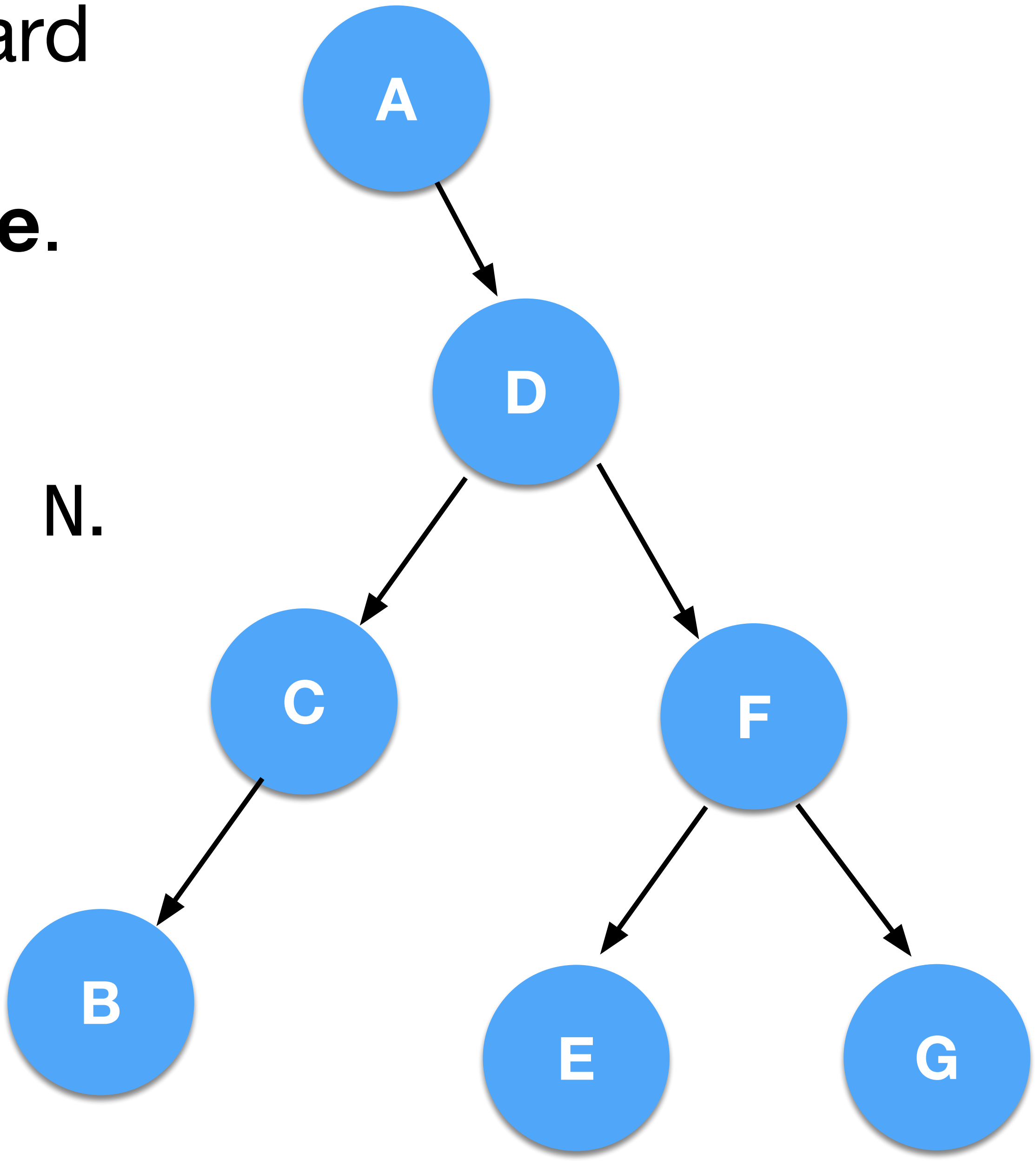
- Once we identified the scapegoat, we rebuild the subtree rooted at the scapegoat, using the same **balance()** method we just covered.
- This is guaranteed to fix the violated condition (proof is omitted here).
- Let's verify if it's true for this example. After rebuilding, what's the tree height? Is it less than $\log_{3/2}(q)$ (3.97 at the moment)?



Example: Removal

We perform removal by following standard BST removal. At each removal, we **decrement `N` while `q` remains the same**. Then check if **`q > 2 N`**. If so, the upper bound condition is violated, we fix it by **rebuilding the entire tree**, then set `q = N`.

operation	N	q	height	$\log_{3/2}(q)$
`remove G`	6	7	3	4.8
`remove F`	5	7	3	4.8
`remove E`	4	7	3	4.8
`remove D`	$\boxed{\textbf{3} \quad \textbf{7}}$		2	4.8



Scapegoat Tree

Summary

- Keep track of **N (# of nodes)**, **h (height)**, and **q (upper bound)**.
  - At insertion, q increments for each new node.
  - At deletion, q remains unchanged.

- After **inserting** a new node, check the **height condition**. If violated, find scapegoat and rebuild the **subtree rooted at the scapegoat** (height condition is guaranteed to be satisfied after the rebuilding)

- After **deleting** a node, check the **upper bound condition**, if violated, **rebuild the entire tree** and **reset q to N** (upper bound condition is satisfied after rebuilding)

Scapegoat Tree

- Why not simply rebuild the tree every time we insert or delete a node?
  - Rebuilding takes `O(N)` time. Doing so will make insertion and deletion degrade to `O(N)`. Bad!
- How does Scapegoat tree address this?
  - Turns out that rebuilding only occurs sparingly (not every time), and it can be mathematically shown that the amortized cost of insertion and deletion (over a large number of elements) are both `O(log N)`. Great!