LECTURE 8: BALANCED BINARY SEARCH TREES (AVL)

Harold Soh
harold@comp.nus.edu.sg

ADMINISTRATIVE ISSUES

Problem 2 is out. Please get started!

Duplicate Kattis IDs

- Please email me and I will remove one.

Discussion Group / Tutorial movements.

QUESTIONS?

RECALL: BST OPERATIONS & COSTS

height(): $O(n)$

search(k) : $O(h)$

searchMin() : $O(h)$

searchMax() : $O(h)$

successor(): $O(h)$

predecessor(): $O(h)$

insert(k,v): $O(h)$

delete(k) : $O(h)$

if the tree is imbalanced, $O(n)$

But what if the tree is balanced?

$$O(h = \log n)$$

RECALL: BST OPERATIONS & COSTS

`height()`: $O(n)$

`search(k)` : $O(\log n)$

`searchMin()` : $O(\log n)$

`searchMax()` : $O(\log n)$

`successor()`: $O(\log n)$

`predecessor()`: $O(\log n)$

`insert(k,v)`: $O(\log n)$

`delete(k)` : $O(\log n)$

if the tree is imbalanced, $O(n)$

But what if the tree is balanced?

$$O(h = \log n)$$

LEARNING OUTCOMES

By the end of this session, students should be able to:

- Derive how **height-balanced trees** ensure O(log n) operations
- Describe **how balance is maintained in an AVL tree.**
- **Explain rotations** and **how they are used to correct height imbalances.**

DIFFERENT BALANCED SEARCH TREES

- AVL trees (Adelson-Velsii & Landis, 1962)
- B-trees / 2-3-4 trees (Bayer & McCreight, 1972)
- BB[α] trees (Nievergelt & Reingold 1973)
- Red-black trees (see CLRS 13) – Discussion Group
- Splay trees (Sleator and Tarjan 1985)
- Treaps (Seidel and Aragon 1996)
- Skip Lists (Pugh 1989)
- Scapegoat Trees (Anderson 1989)

TODAY: HOW TO ENSURE TREES ARE BALANCED

The key idea:

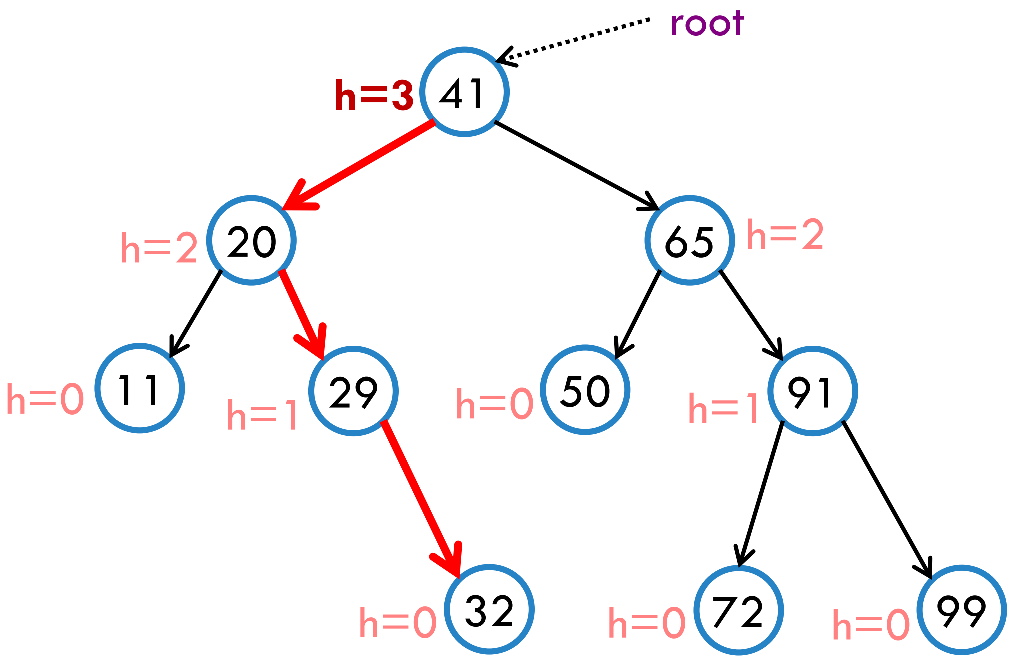
Define a good property of the tree

Show that if the good property holds, then the tree is balanced

If tree changes, ensure that good property holds. If not, fix it.

Another word for this "good property" is

____



TODAY: HOW TO ENSURE TREES ARE BALANCED

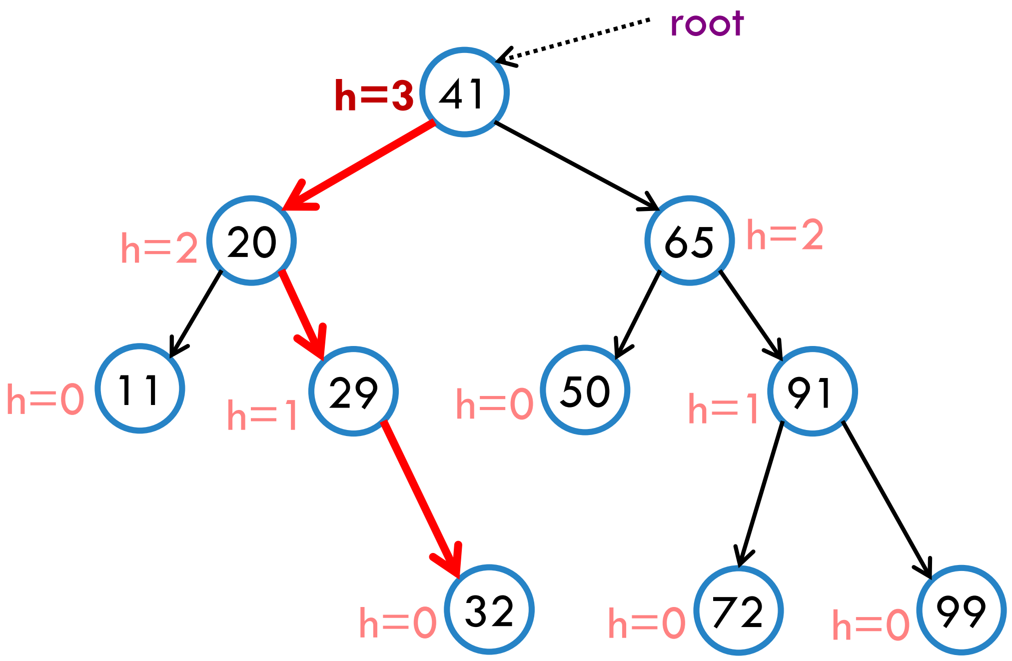
The key idea:

Define a good property of the tree

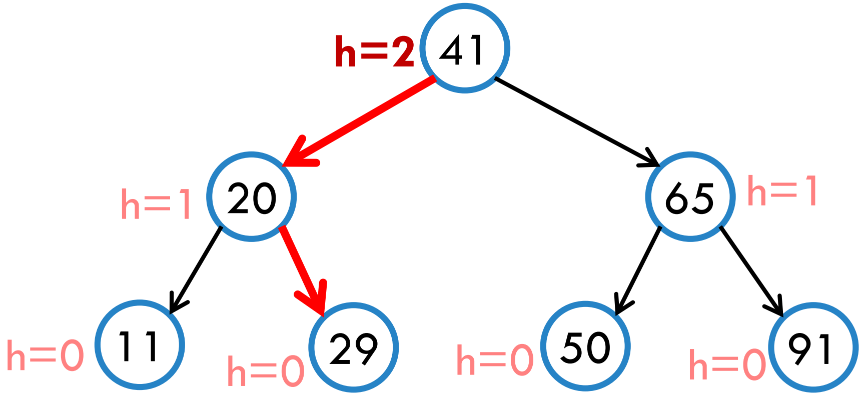
Show that if the good property holds, then the tree is balanced

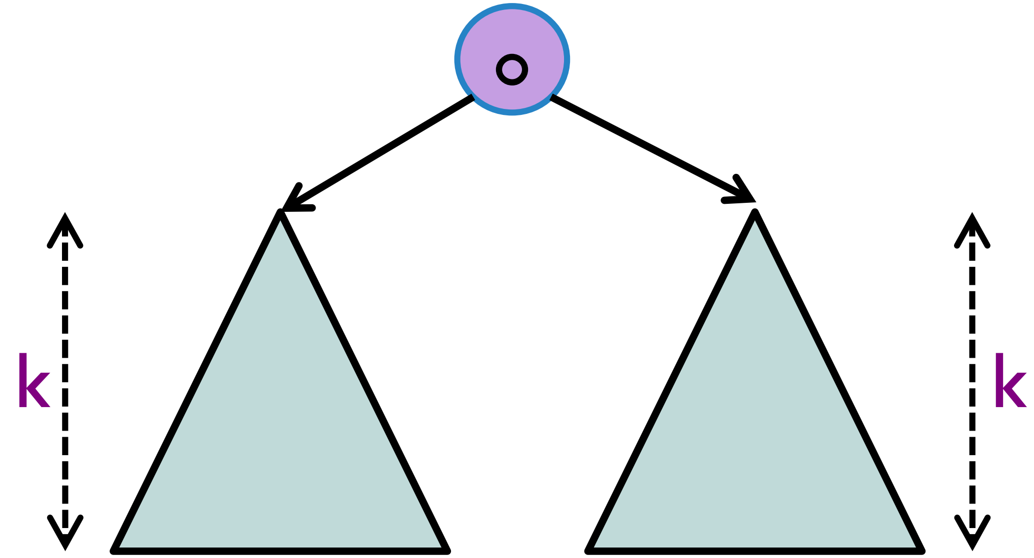
If tree changes, ensure that good property holds. If not, fix it.

Another word for this "good property" is INVARIANT



PERFECTLY BALANCED





AVL TREES: HEIGHT-BALANCED

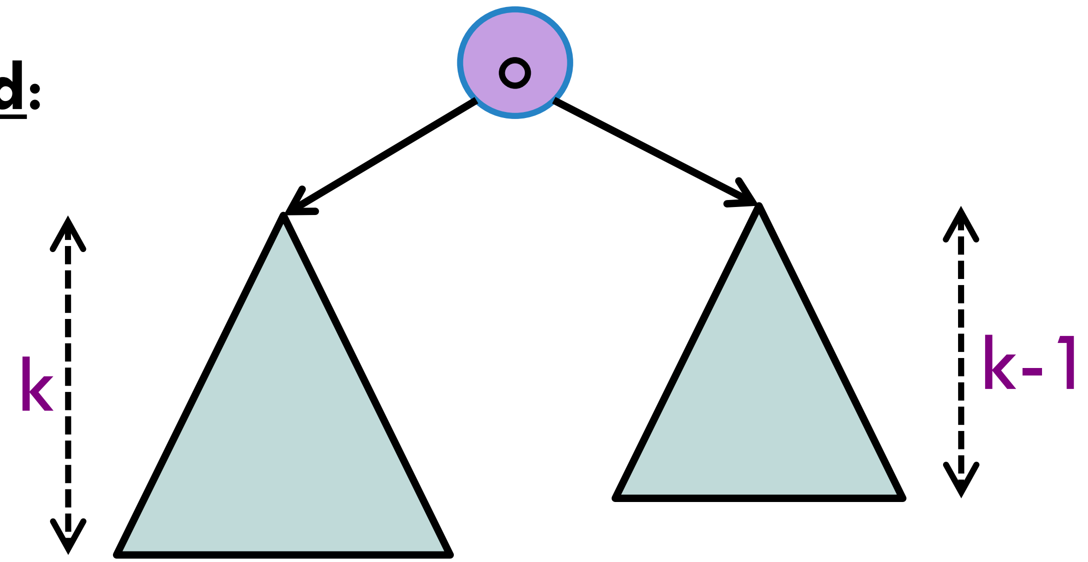
1. Keep a height variable at each node

```
o.height = max(o.left.height, o.right.height) + 1
```

2. Maintain the following invariant:

all nodes in the BST are **height balanced**:

$$|\text{o.left.height()} - \text{o.right.height()}| \leq 1$$



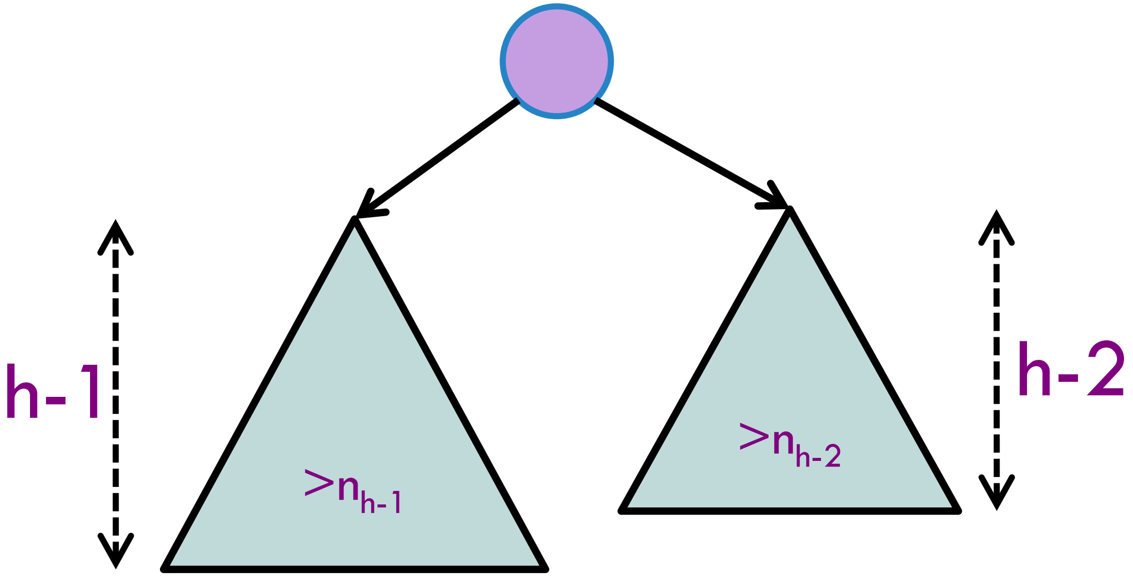
HEIGHT OF A HEIGHT-BALANCED TREE

Claim: A height-balanced tree with n nodes has at most height $h < 2\log n$ so, $h = O(\log n)$

Let us define n_h as the minimum number of nodes in an AVL tree

$$n_h = 1 + n_{h-1} + n_{h-2}$$

Assume without loss of generality (WLOG), $n_{h-1} > n_{h-2}$



Post-lecture: "I realized from questions after lecture that I didn't cover this well enough (why minimum num nodes and why WLOG). Please see the provided link on piazza (https://people.csail.mit.edu/alinush/6.006-spring-2014/avl-height-proof.pdf) for more specifics"

HEIGHT OF A HEIGHT-BALANCED TREE

Claim: A height-balanced tree with n nodes has at most height $h < 2\log n$ so, $h = O(\log n)$

Let us define n_h as the minimum number of nodes in an AVL tree

$$n_h = 1 + n_{h-1} + n_{h-2}$$

Then,

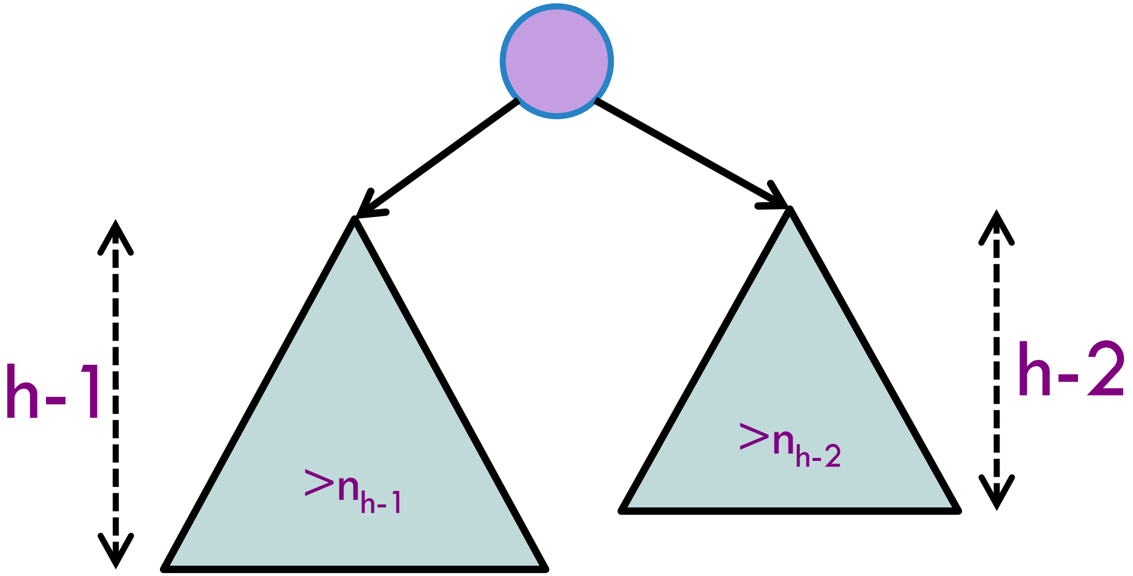
$$n_h > 1 + n_{h-2} + n_{h-2} > 2n_{h-2}$$

So,

$$\underbrace{n_h > 2n_{h-2}}_{\text{This is a recurrence}}$$

Assume without loss of generality (WLOG),

$n_{h-1} > n_{h-2}$



HEIGHT OF A HEIGHT-BALANCED TREE

Claim: A height-balanced tree with n nodes has at most height $h < 2\log n$ so, $h = O(\log n)$

Solve this recurrence where $n_0 = 1$

$n_h > 2n_{h-2} > 2 \cdot 2n_{h-4} > \cdots > 2^{h/2}n_0$

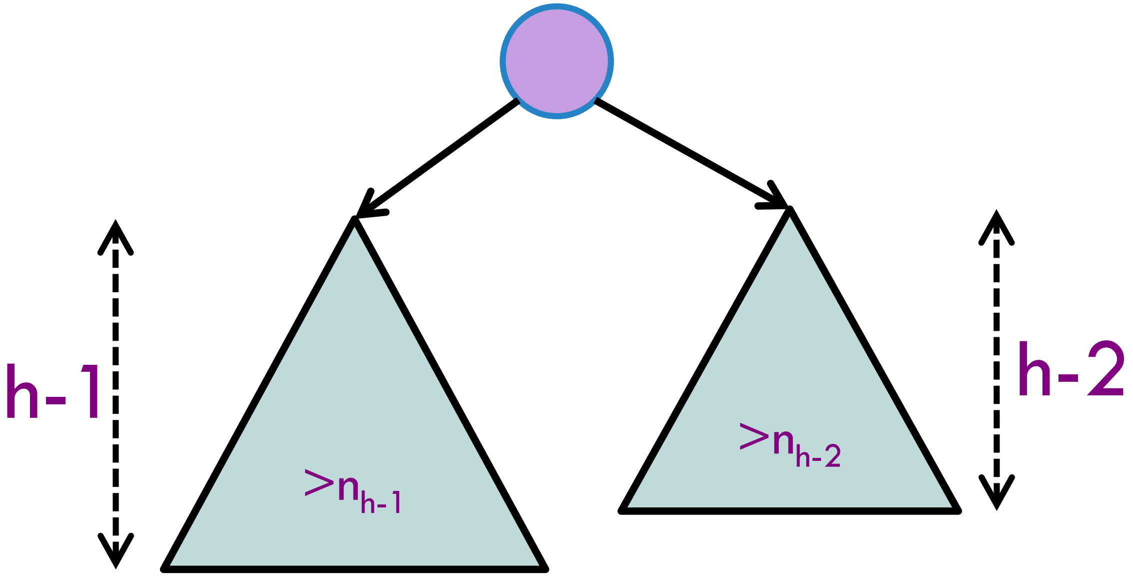
Next, we bound h:

$$n_h > 2^{h/2}$$

Take log:

$$\log n_h > \log 2^{h/2}$$
$$h < 2\log n_h \quad \blacksquare$$

Assume without loss of generality (WLOG), $n_{h-1} > n_{h-2}$



RECALL: BST OPERATIONS & COSTS

height(): $O(n)$

search(k) : $O(\log n)$

searchMin() : $O(\log n)$

searchMax() : $O(\log n)$

successor(): $O(\log n)$

predecessor(): $O(\log n)$

insert(k,v): $O(\log n)$

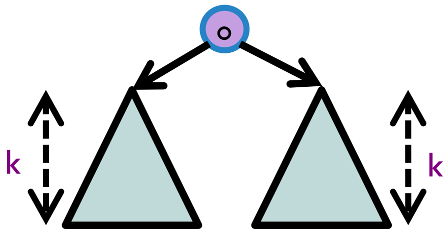
delete(k) : $O(\log n)$

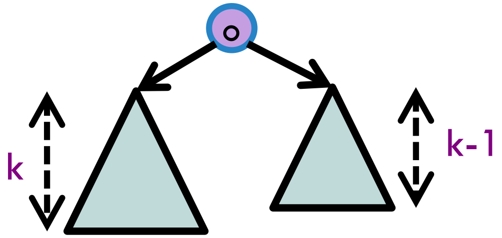
if the tree is imbalanced, $O(n)$

But what if the tree is height-balanced?

$\boldsymbol{O(\log n)}$

DON’T HAVE TO PERFECTLY BALANCE!



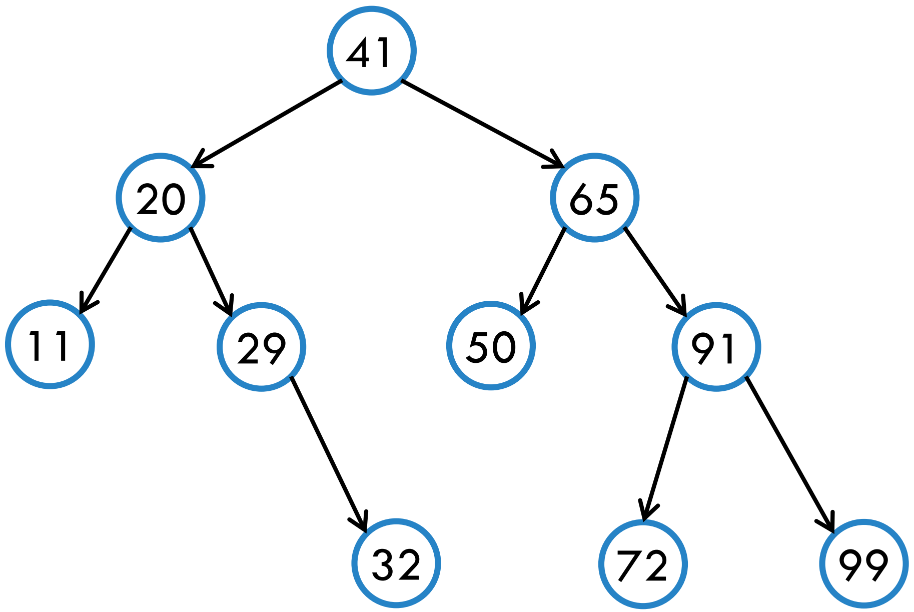


AVL TREES

1. Keep a height variable at each node

```
o.height = max(o.left.height, o.right.height) + 1
```

2. When you change the tree, it may not longer be height balanced, i.e., for all nodes $|\text{o.left.height()} - \text{o.right.height()}| \leq 1$
3. Rebalance via rotations.



AVL TREES

1. Keep a height variable at each node

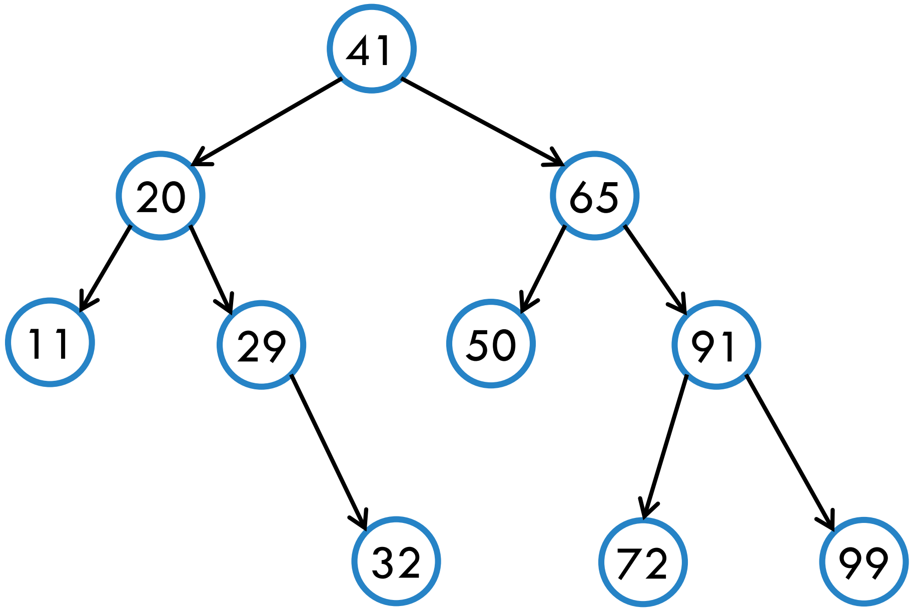
   `o.height = max(o.left.height, o.right.height) + 1`

2. When you change the tree, it may not longer be height balanced, i.e., for all nodes $|$o.left.height() – o.right.height()$| \leq 1$

3. Rebalance via rotations.

Balance Factor

Define b(o) = o.left.height() – o.right.height()
b(o) of an empty tree (null) is -1
if b(o) > 1 or b(o) < -1 for any o, the tree is no longer height balanced.



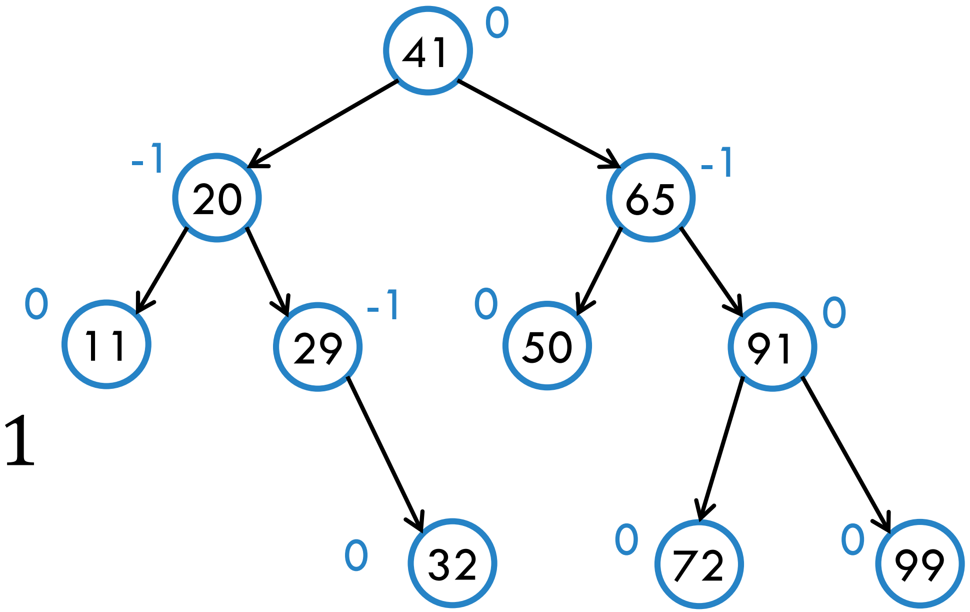
AVL TREES

1. Keep a height variable at each node

```
o.height = max(o.left.height, o.right.height) + 1
```

2. When you change the tree, it may not longer be height balanced, i.e., for all nodes $|\text{o.left.height()} - \text{o.right.height()}| \leq 1$

3. Rebalance via rotations.



Balance Factor

Define b(o) = o.left.height() – o.right.height()
b(o) of an empty tree (null) is -1
if b(o) > 1 or b(o) < -1 for any o, the tree is no longer height balanced.

AVL TREES

1. Keep a height variable at each node

   `o.height = max(o.left.height, o.right.height) + 1`

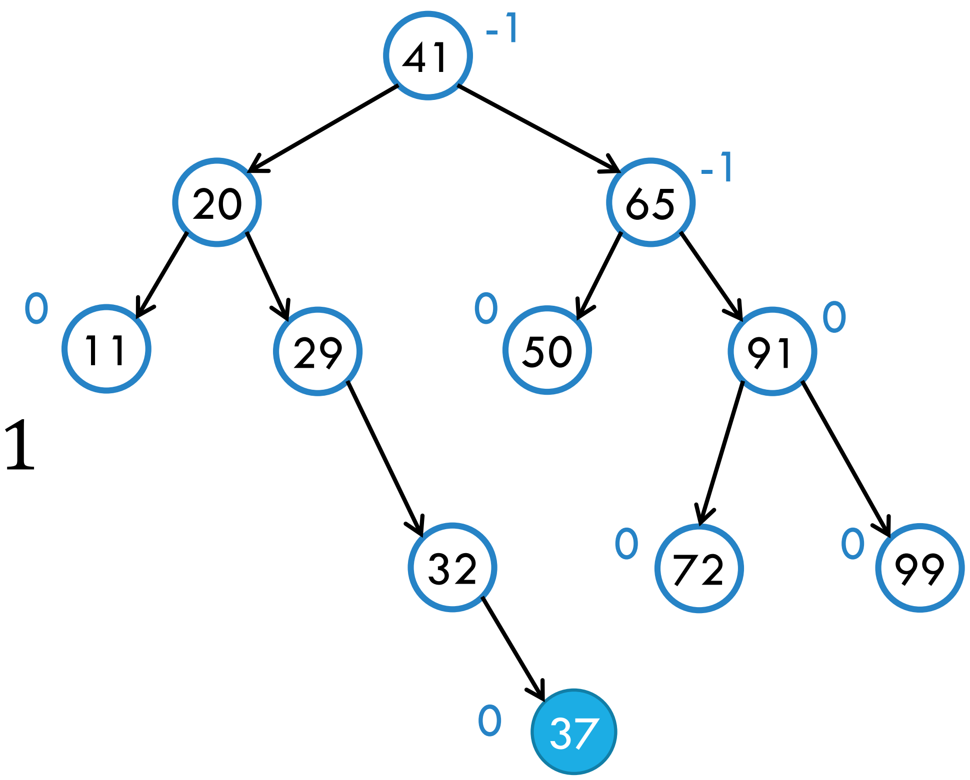
2. When you change the tree, it may not longer be height balanced, i.e., for all nodes $|o.left.height() - o.right.height()| \leq 1$

3. Rebalance via rotations.

Balance Factor

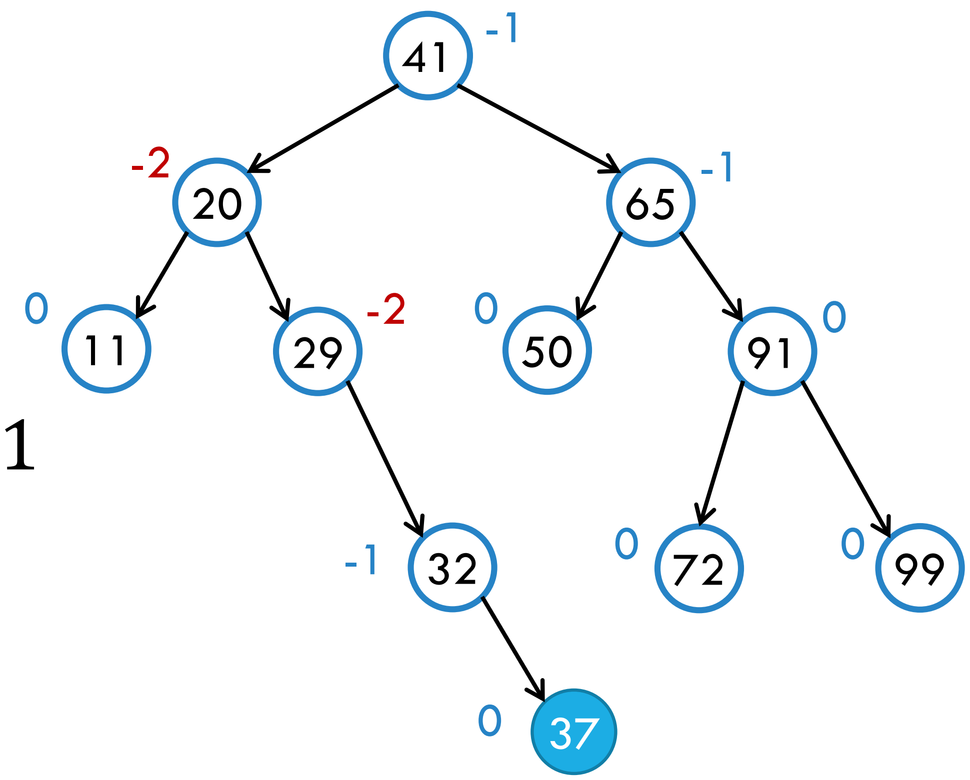
Define b(o) = o.left.height() – o.right.height()
b(o) of an empty tree (null) is -1
if b(o) > 1 or b(o) < -1 for any o, the tree is no longer height balanced.

insert(37) yields an imbalanced tree



AVL TREES

insert(37) yields an imbalanced tree



1. Keep a height variable at each node

   `o.height = max(o.left.height, o.right.height) + 1`

2. When you change the tree, it may not longer be height balanced, i.e., for all nodes |o.left.height() – o.right.height()| ≤ 1

3. Rebalance via rotations.

Balance Factor

Define b(o) = o.left.height() – o.right.height()
b(o) of an empty tree (null) is -1
if b(o) > 1 or b(o) < -1 for any o, the tree is no longer height balanced.

ROTATIONS

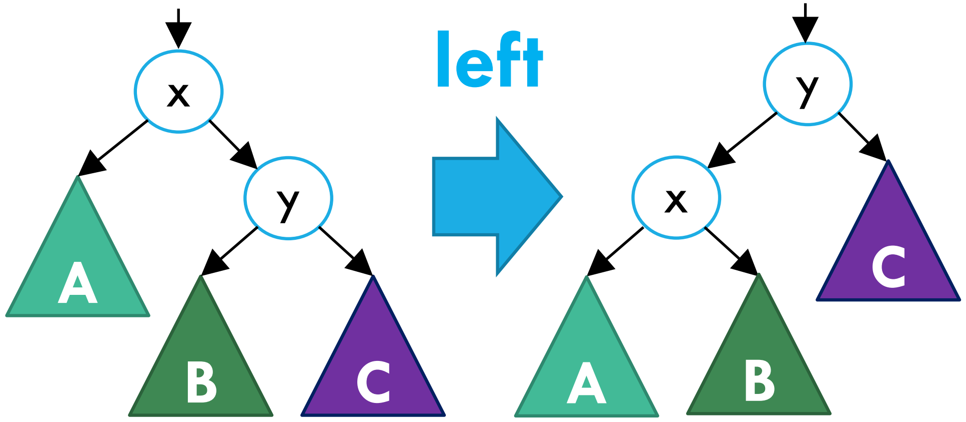
Common "primitive" operation used in balanced search trees.

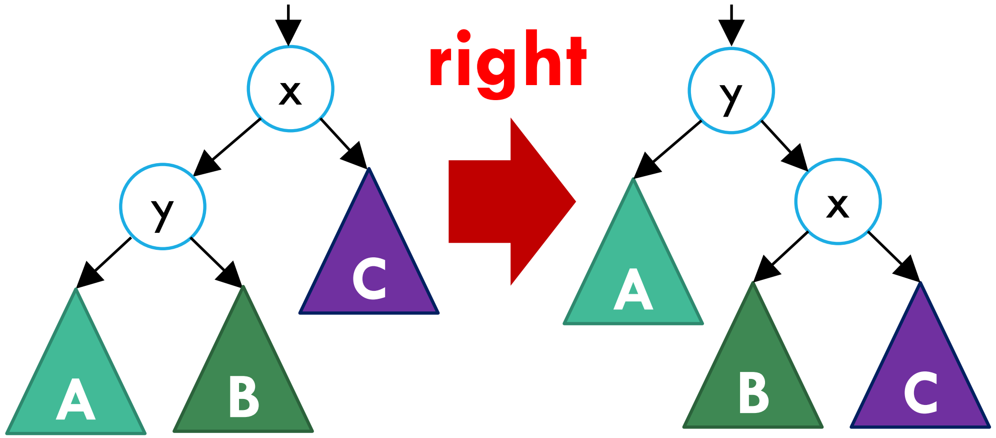
Idea: Locally rebalance subtrees at a given node

4 types:

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

Basic rotations



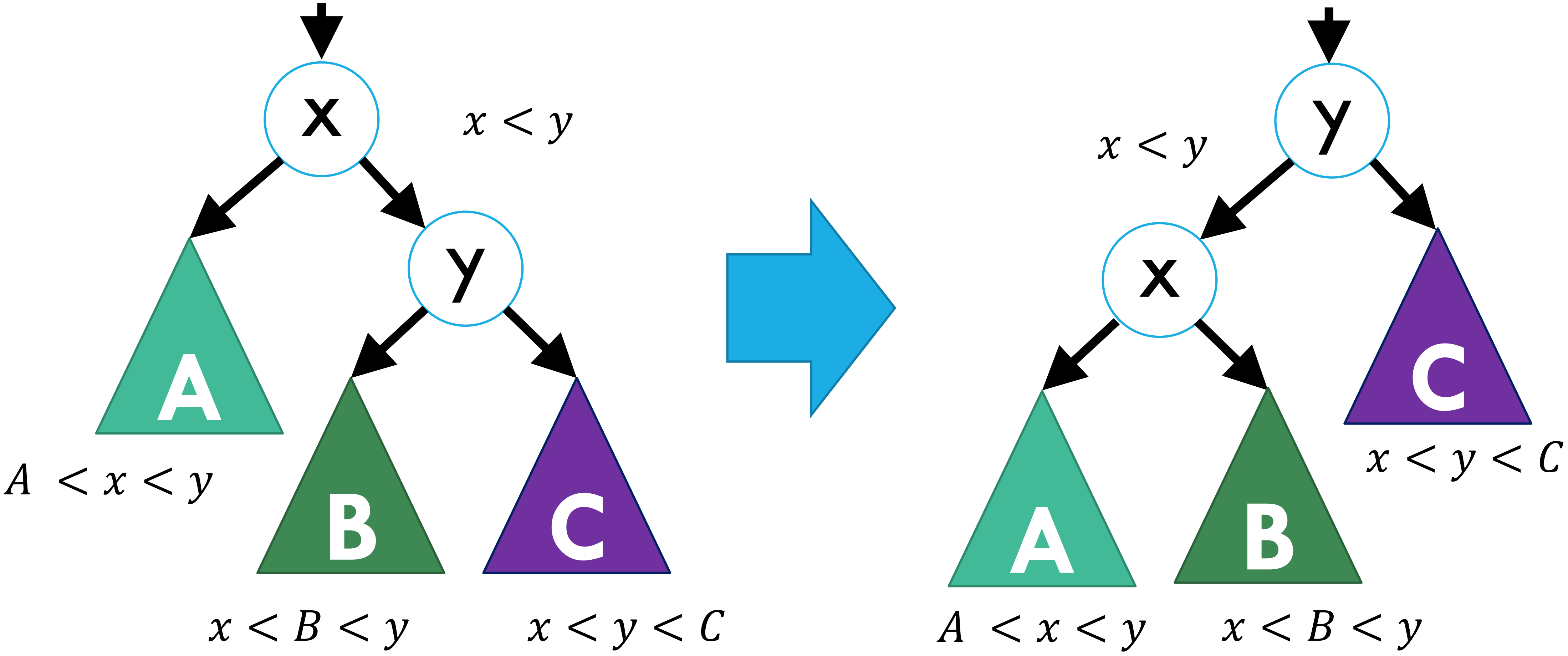


LEFT ROTATION

Applies: y is x's right child.

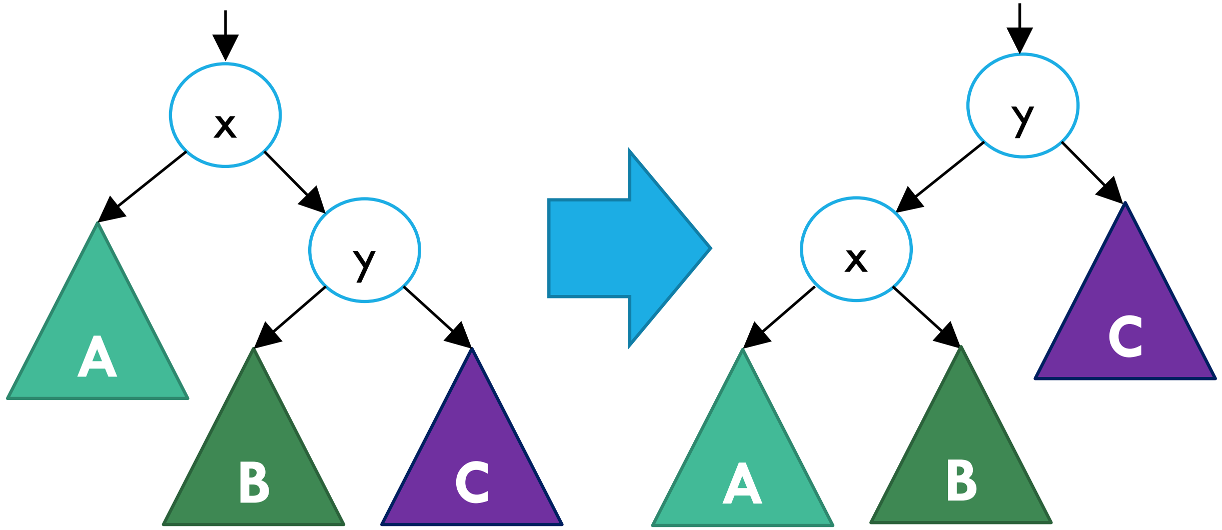
Objective: Want y to be x's parent

But must preserve BST properties.



Poll Everywhere
https://bit.ly/2LvG9bq

LEFT ROTATION



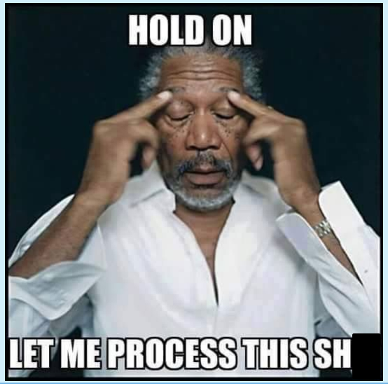
Applies: y is x’s right child.

Objective: Want y to be x’s parent

<u>But</u> must preserve BST properties.

What is the worst time time complexity of a left rotation?

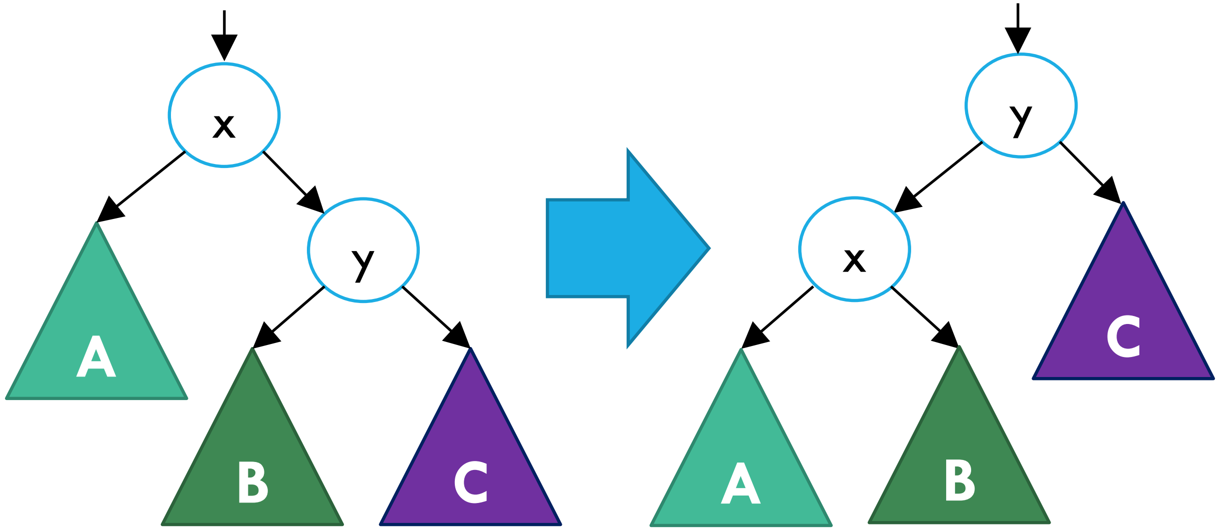
A. $O(1)$
B. $O(n)$
C. $O(\log n)$
D.



Poll Everywhere

https://bit.ly/2LvG9bq

LEFT ROTATION



Applies: y is x’s right child.

Objective: Want y to be x’s parent

<u>But</u> must preserve BST properties.

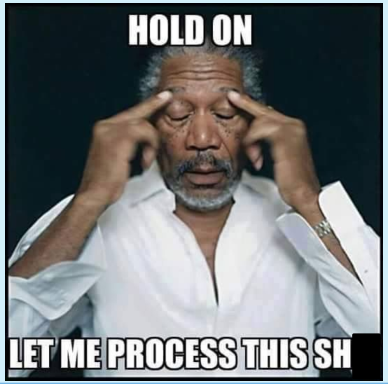
What is the worst time time complexity of a left rotation?

A. $\boldsymbol{O(1)}$

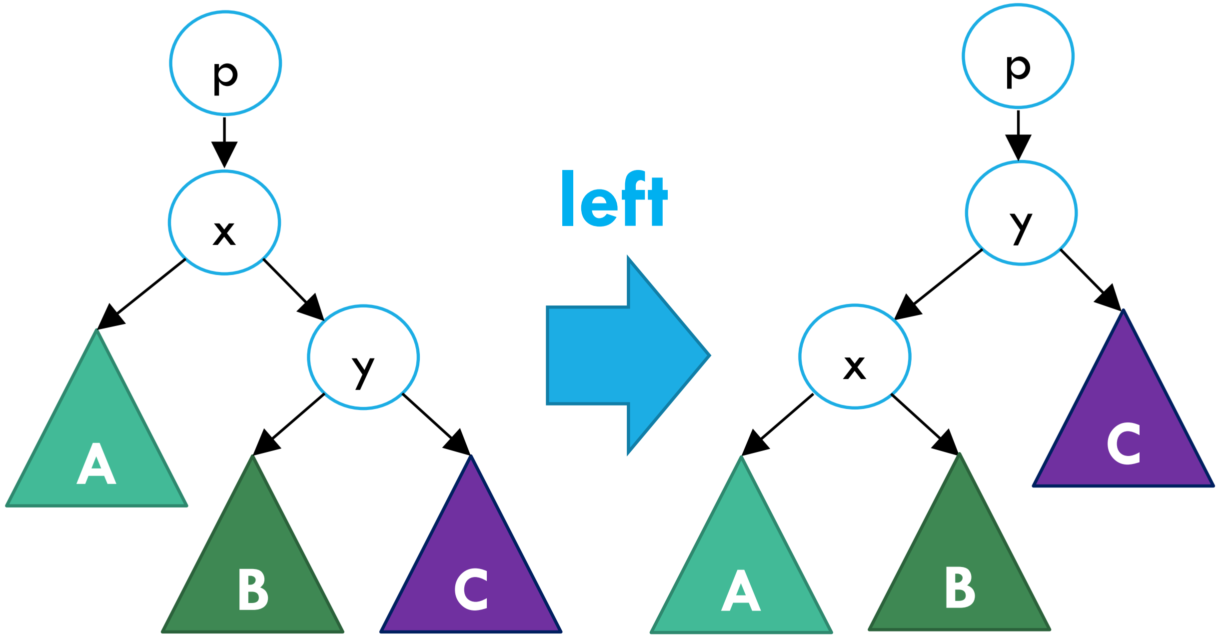
B. $O(n)$

C. $O(\log n)$

D.



ROTATE LEFT PSEUDOCODE

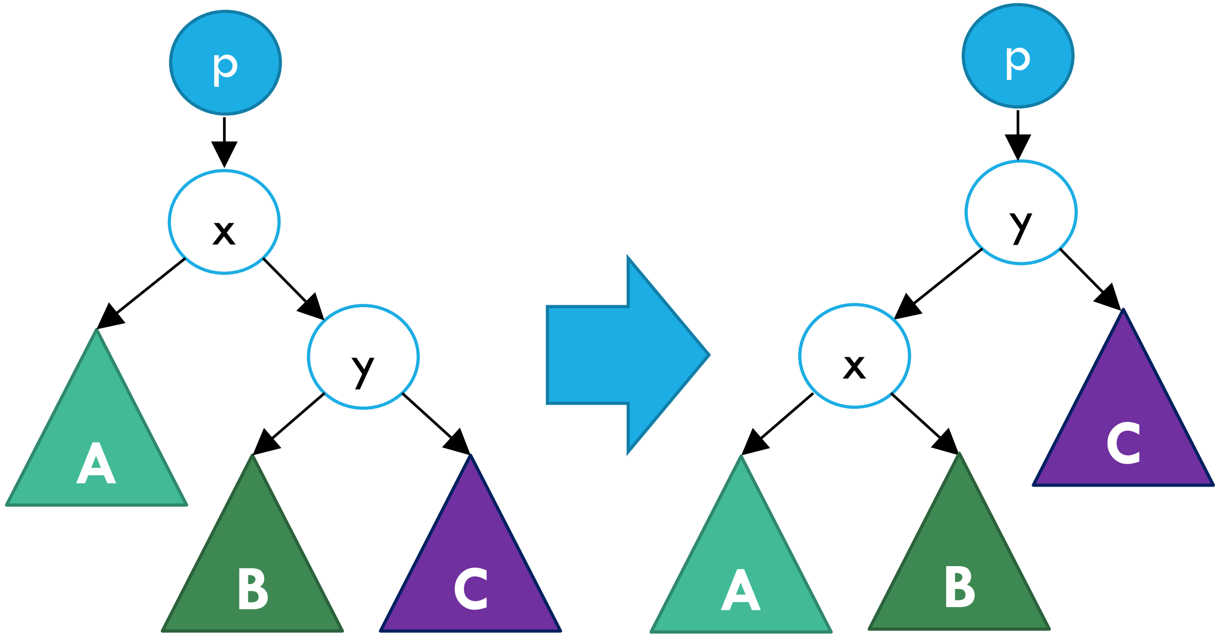



```
function rotateLeft(x)
    y = x.right
    // reassign parent
    y.parent = x.parent
    // make y into x's parent
    x.parent = y
    // move B
    x.right = y.left
    if (x.right is not null)
        x.right.parent = x
    // make x into y's child
    y.left = x
    // update heights and parent reference
    return y
```

Poll Everywhere

https://bit.ly/2LvG9bq

LEFT ROTATION



Applies: y is x's right child.

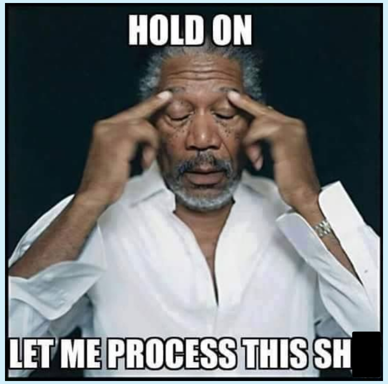
Objective: Want y to be x's parent

<u>But</u> must preserve BST properties.

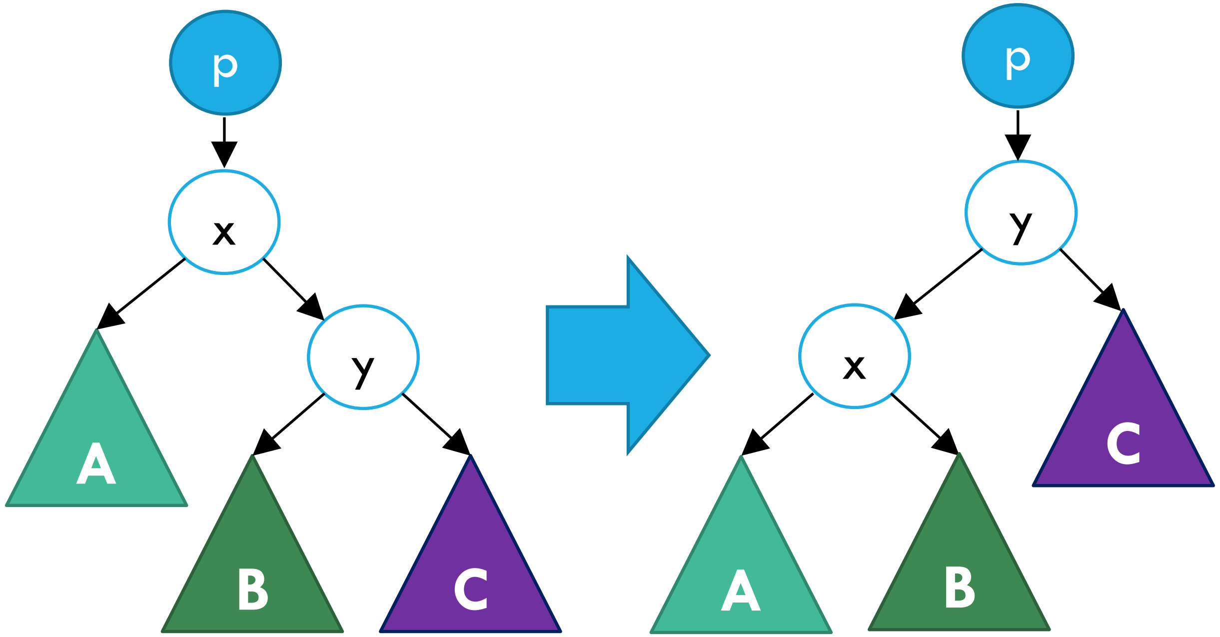
Is the BST property preserved for the parent?

A. Yeah! Of course!

B. Nope. We need to fix it.

C. 

LEFT ROTATION



Applies: y is x's right child.

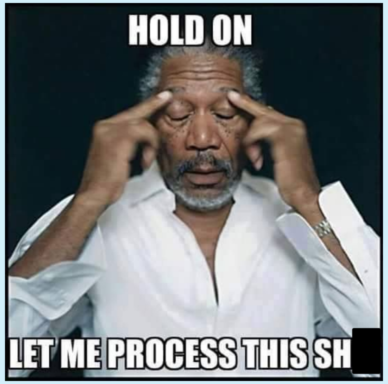
Objective: Want y to be x's parent

<u>But</u> must preserve BST properties.

Is the BST property preserved for the parent?

A. **Yeah! Of course!**

B. Nope. We need to fix it.

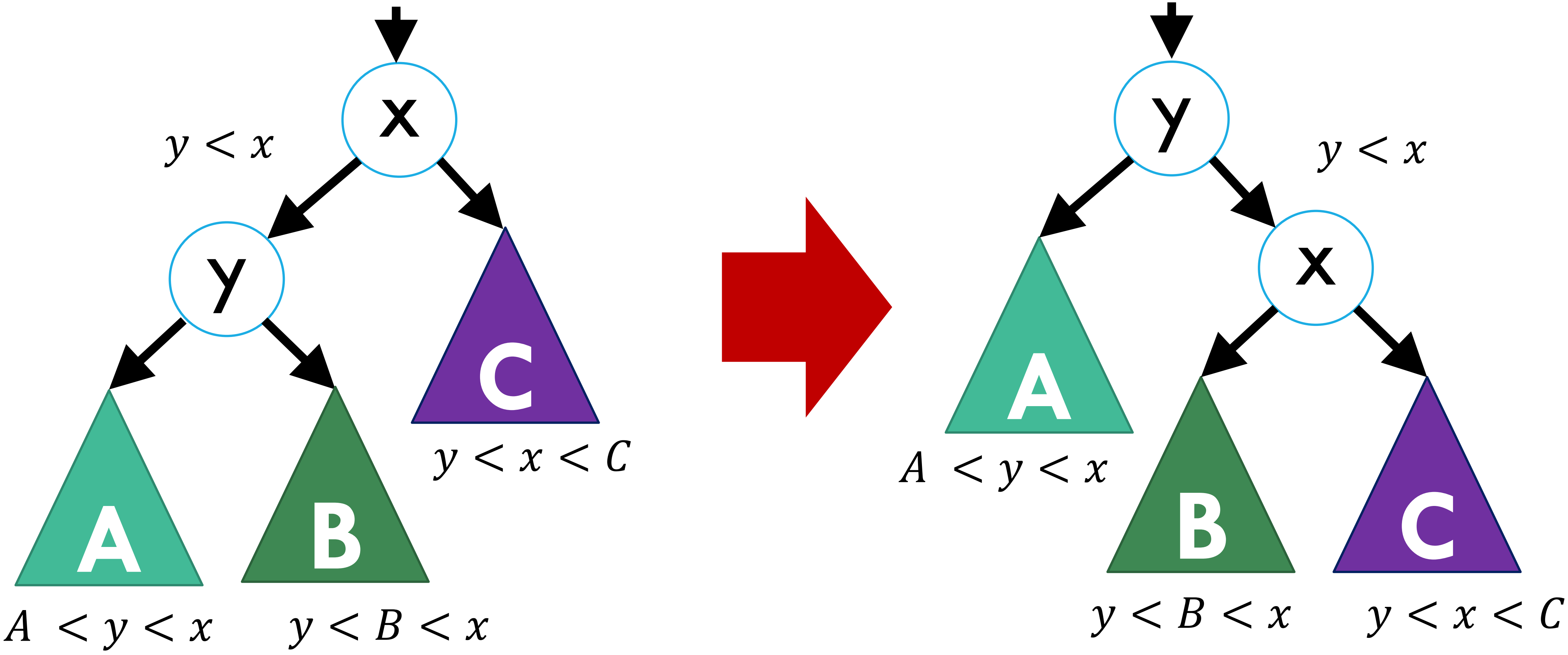
C. 

RIGHT ROTATION

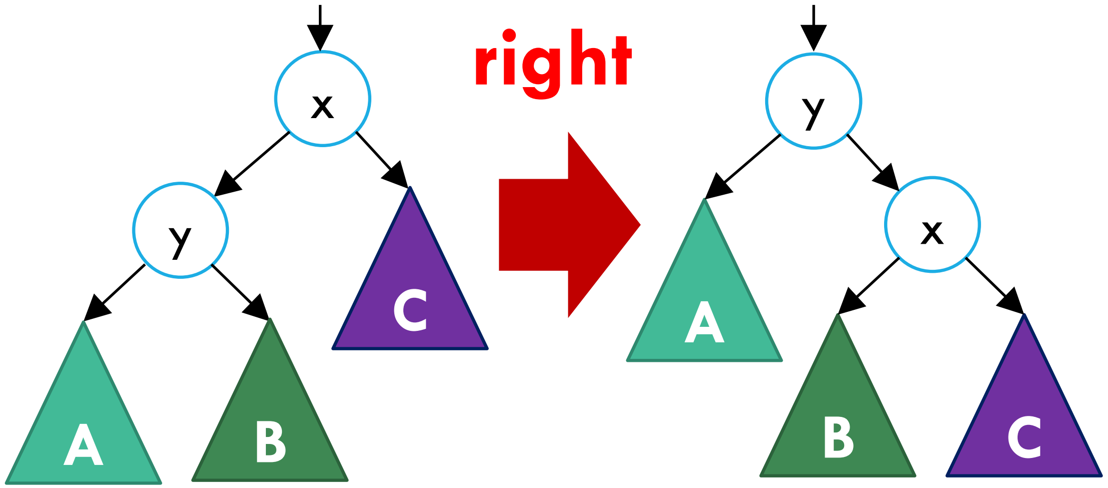
Applies: y is x's left child.

Objective: Want y to be x's parent

<u>But</u> must preserve BST properties.



ROTATE RIGHT PSEUDOCODE




```
function rotateRight(x)
    y = x.left
    // reassign parent
    y.parent = x.parent
    // make y into x's parent
    x.parent = y
    // move B
    x.left = y.right
    if (x.left is not null)
        x.left.parent = x
    // make x into y's child
    y.right = x
    // update heights and parent reference
    return y
```

ROTATIONS

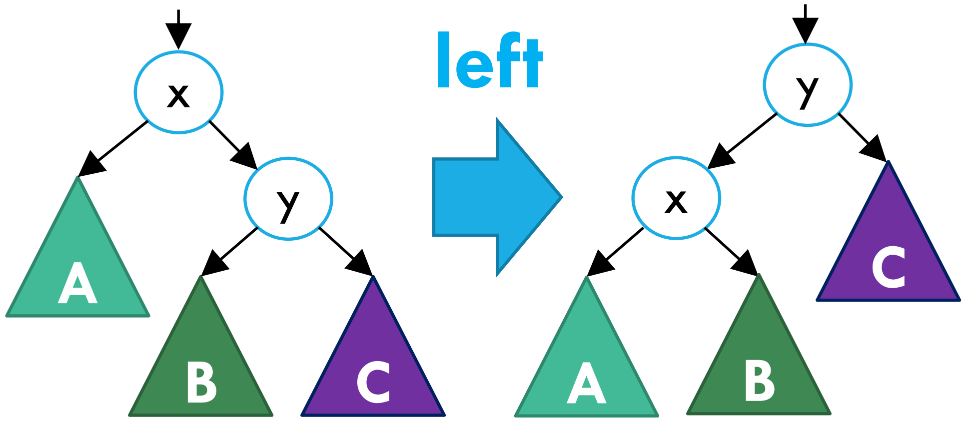
Common “primitive” operation used in balanced search trees.

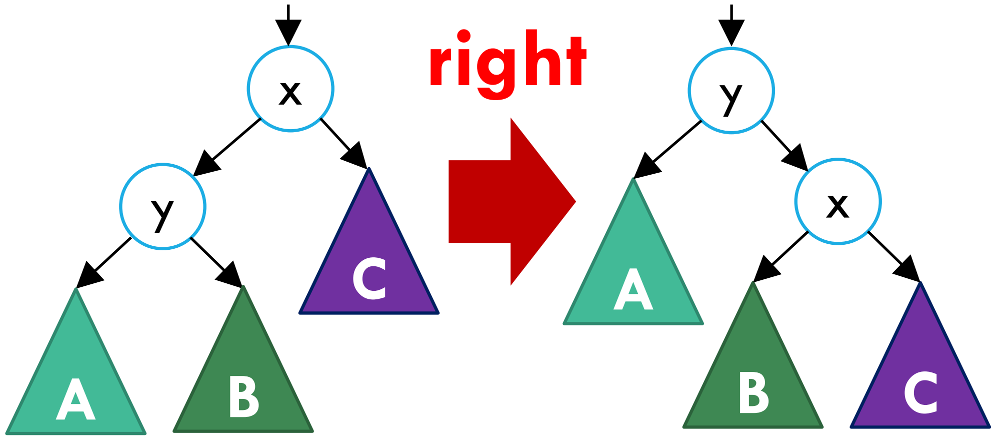
Idea: Locally rebalance subtrees at a given node

4 types:

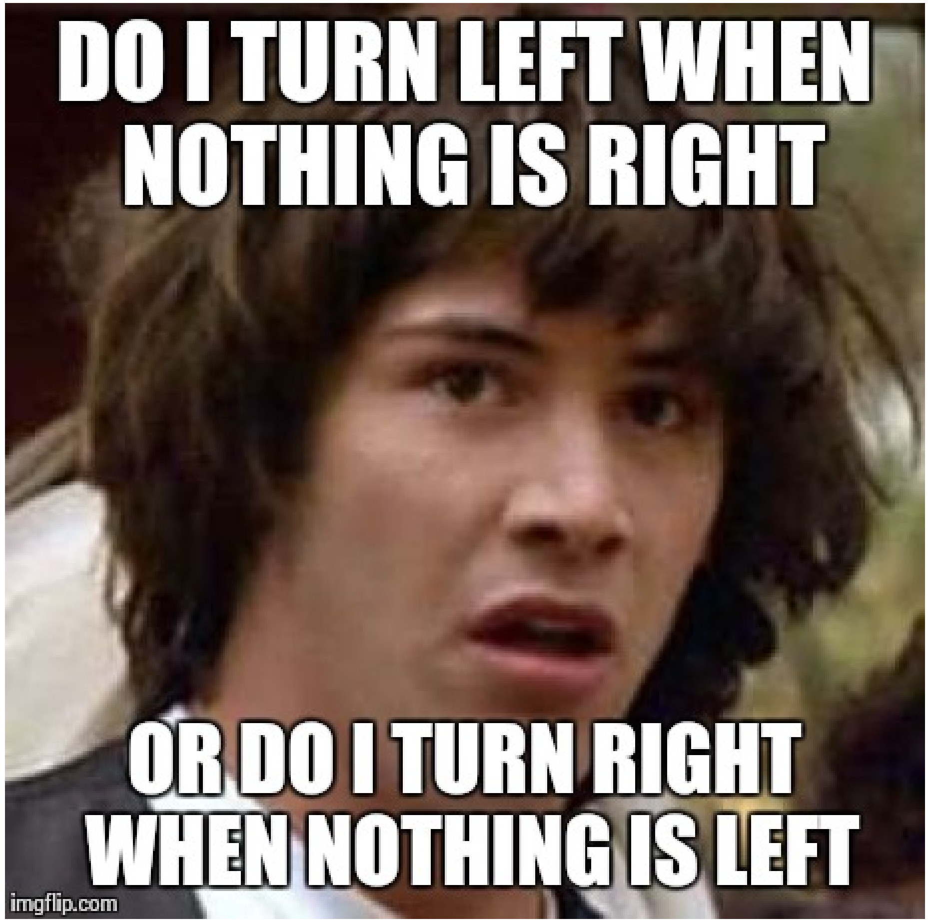
- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

Basic rotations





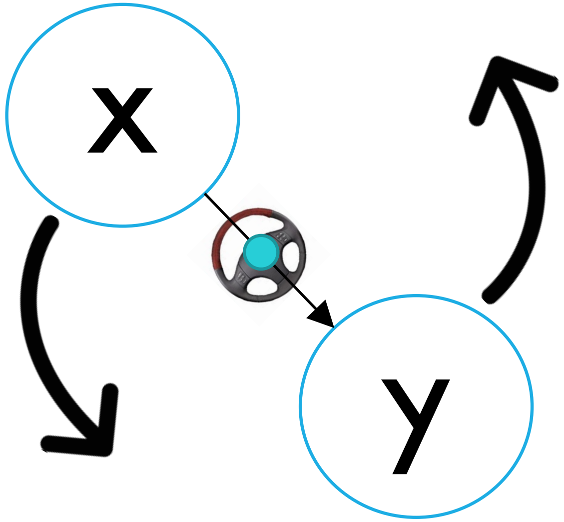
LEFT OR RIGHT ROTATION ?!?

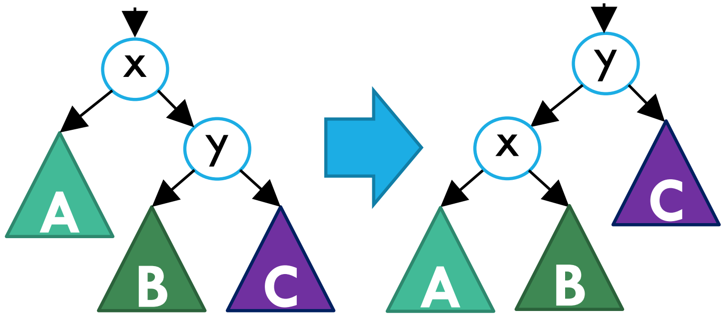


HOW I REMEMBER IT

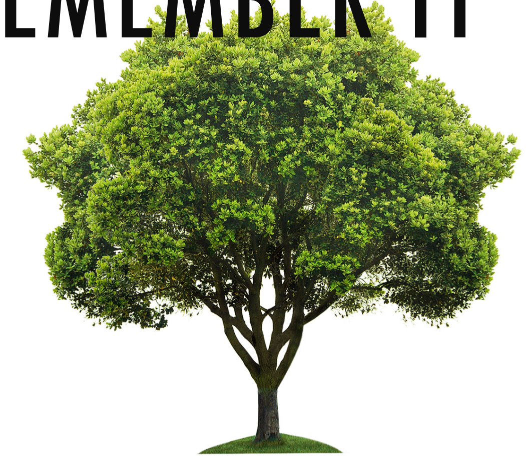
HOW I REMEMBER IT

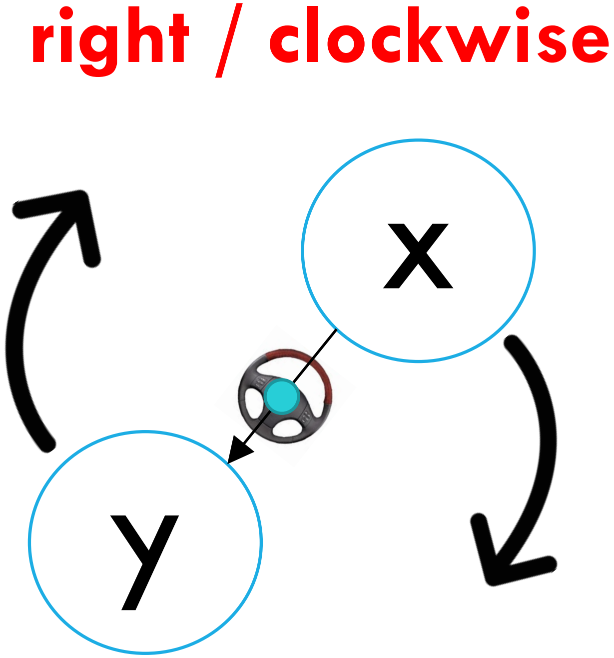
left / anti-clockwise

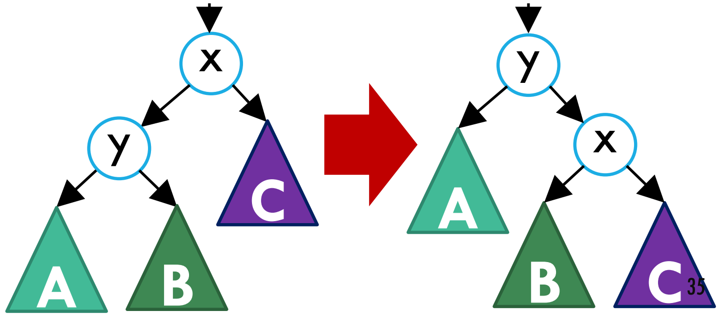




HOW I REMEMBER IT

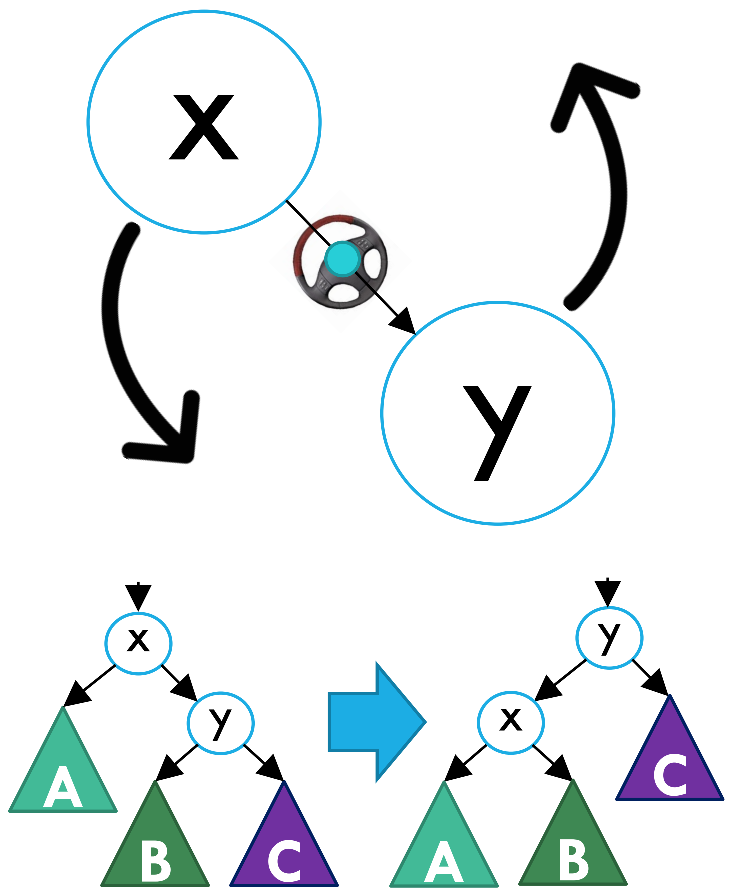




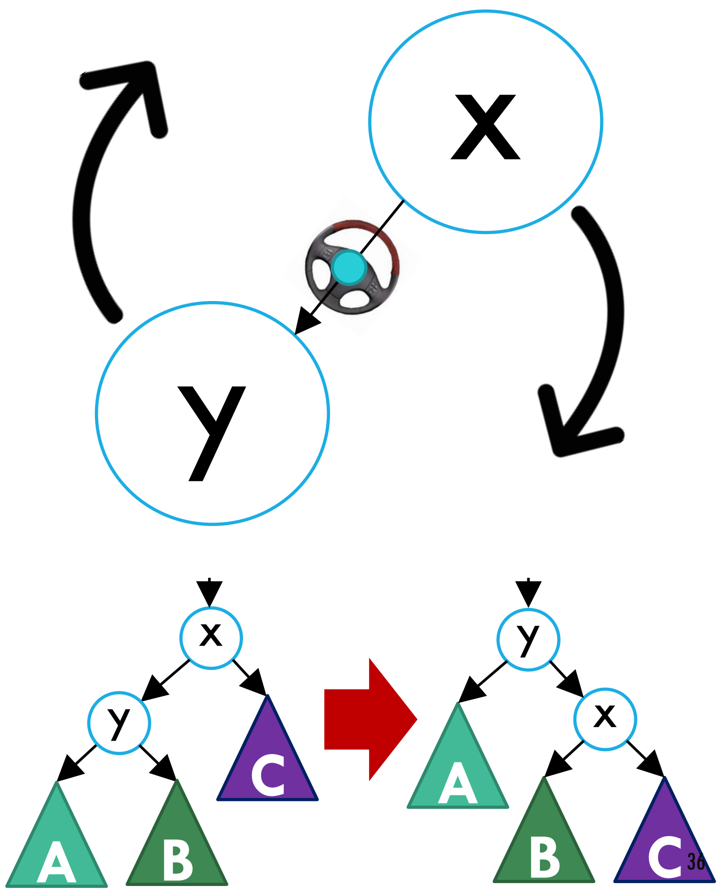


HOW I REMEMBER IT

left / anti-clockwise

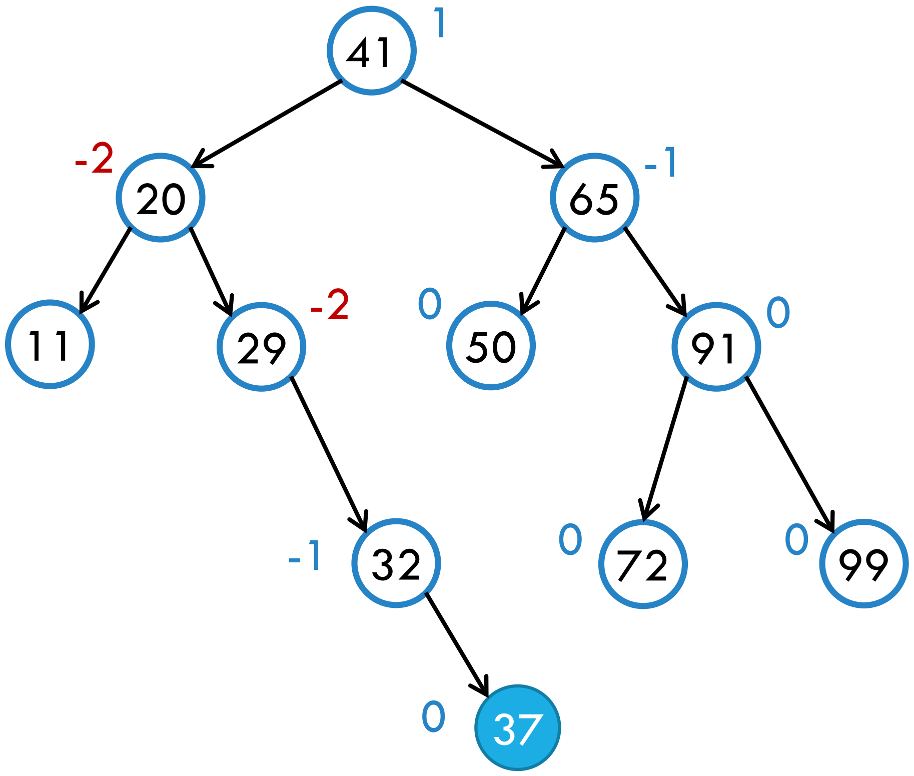


right / clockwise



BACK TO OUR EXAMPLE

insert(37) yields an imbalanced tree



1. Keep a height variable at each node

   `o.height = max(o.left.height, o.right.height) + 1`

2. When you change the tree, it may not longer be height balanced, i.e., for all nodes |o.left.height() – o.right.height()| ≤ 1

3. Rebalance via rotations.

   Balance Factor

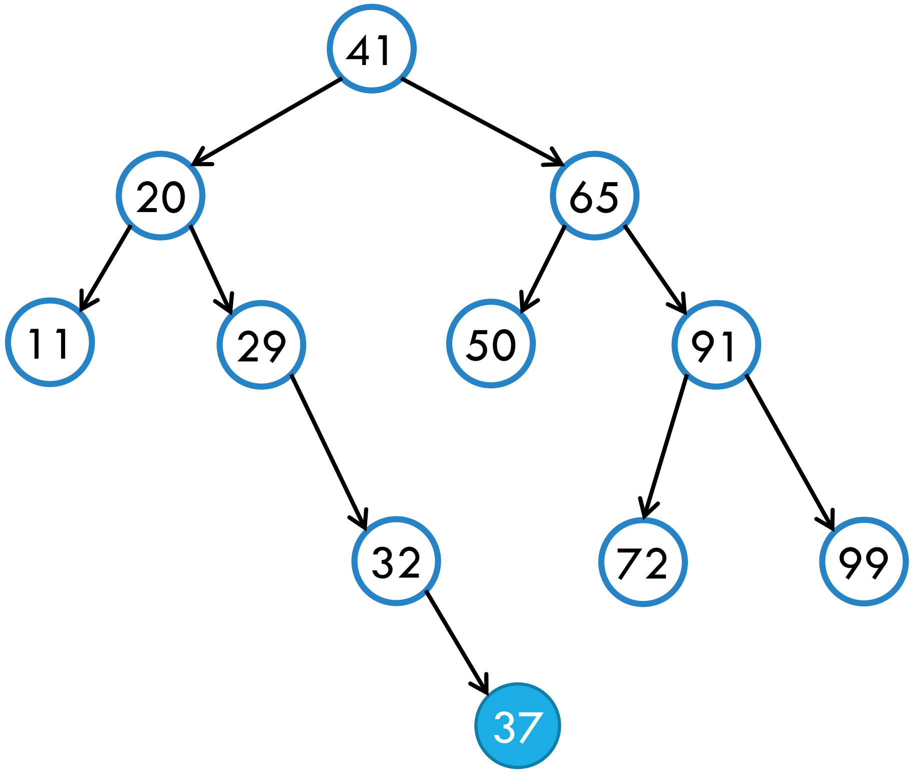
   Define b(o) = o.left.height() – o.right.height()

   b(o) of an empty tree (null) is -1

   if b(o) > 1 or b(o) < -1 for any o, the tree is no longer height balanced.

ROTATE WHERE?

insert(37) yields an imbalanced tree



Which node should we rotate and how?

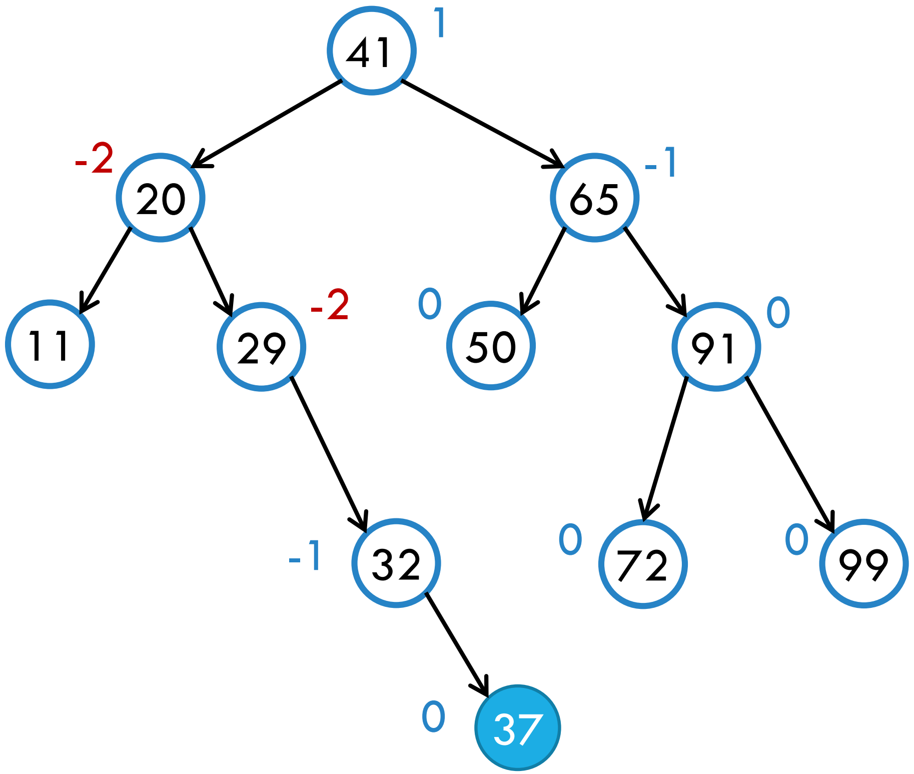
A. 29, left
B. 32 right
C. 11, left
D. 41, right
E.

ROTATE WHERE?

Which node should we rotate and how?

A. 29, left
B. 32 right
C. 11, left
D. 41, right
E. 

insert(37) yields an imbalanced tree

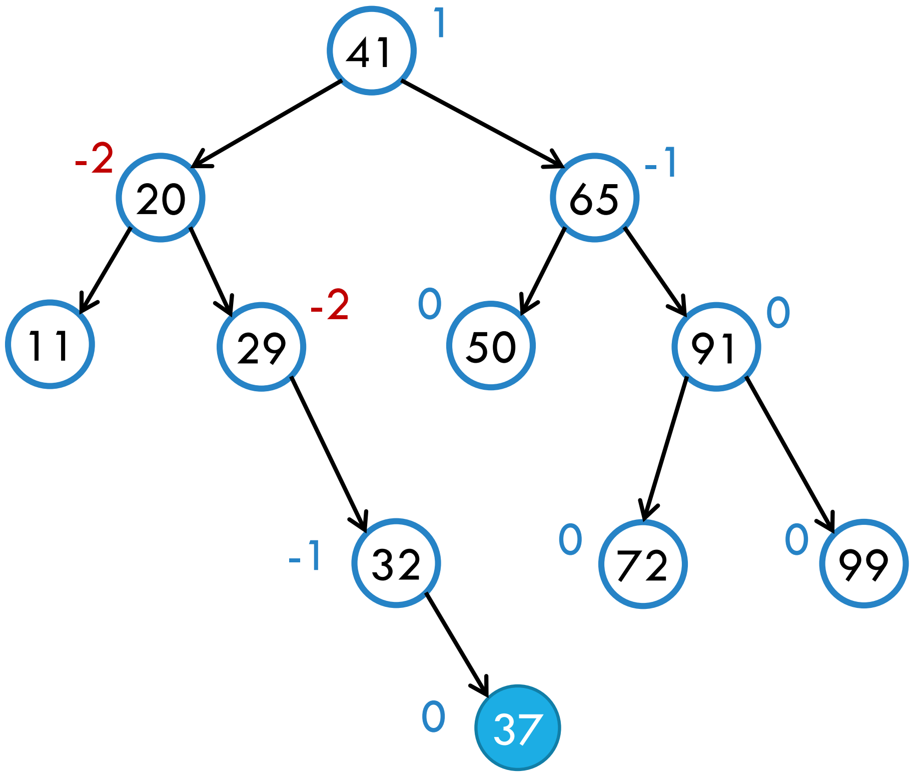


ROTATE WHERE?

Which node should we rotate and how?

A. **29, left**
B. 32 right
C. 11, left
D. 41, right
E. 

insert(37) yields an imbalanced tree

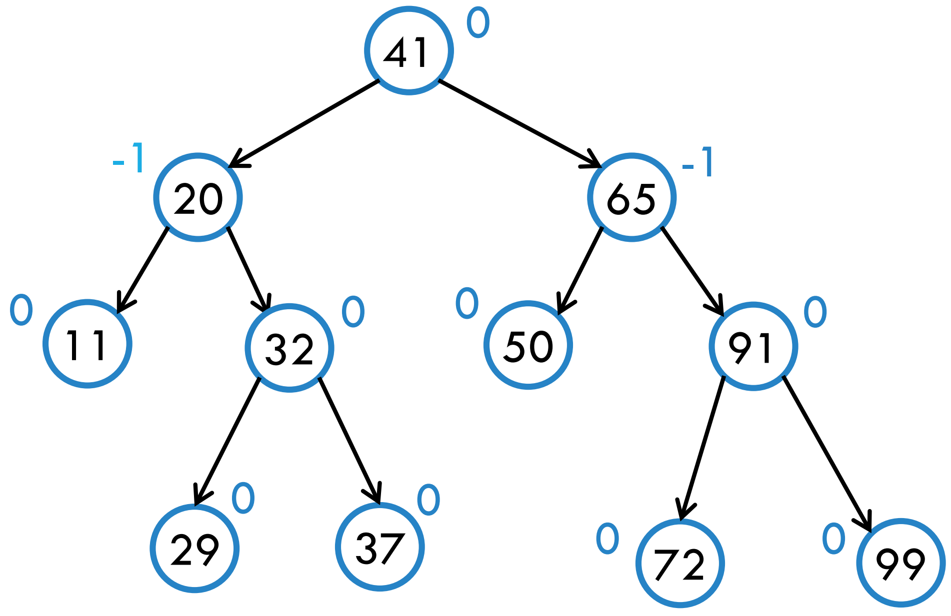


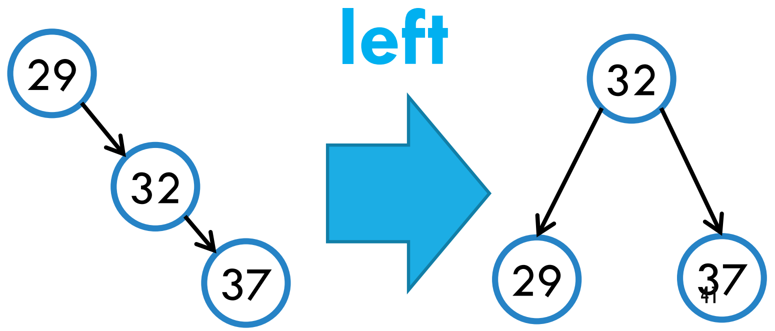
ROTATE WHERE?

Which node should we rotate and how?

A. **29, left**
B. 32 right
C. 11, left
D. 41, right
E.

insert(37) yields an imbalanced tree





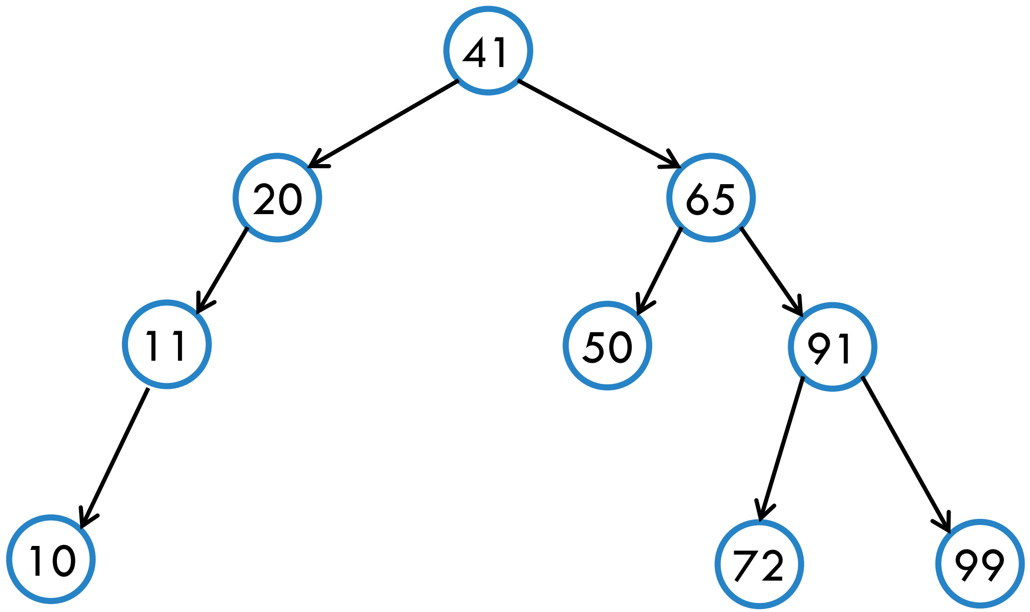
Poll Everywhere

https://bit.ly/2LvG9bq

ROTATE WHERE?

Which node should we rotate and how?

A. 11, left
B. 20 right
C. 11, right
D. 20, left
E. Ok, I'm not getting this rotation stuff.



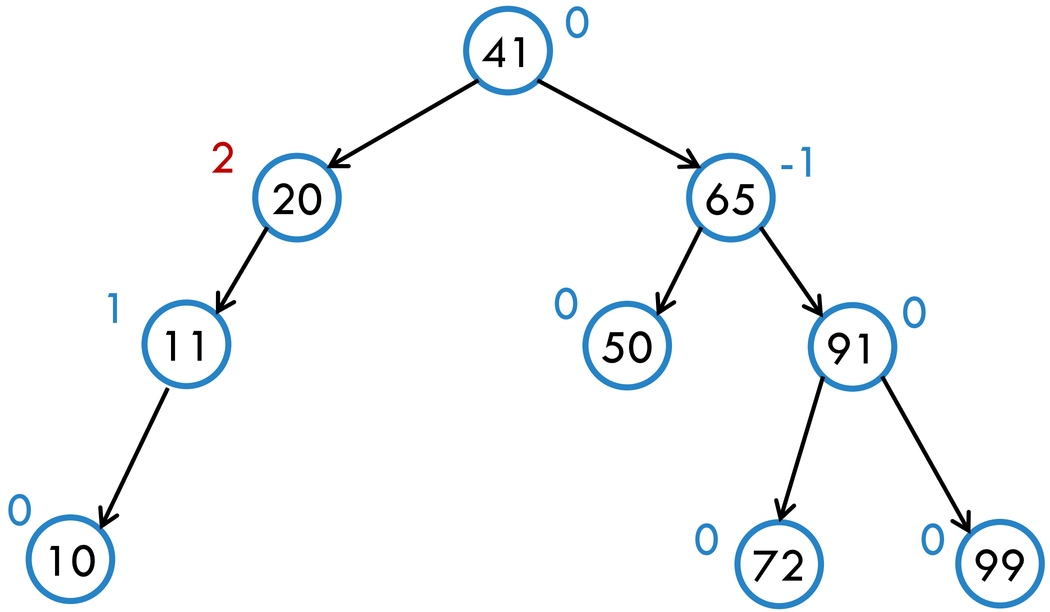
ROTATE WHERE?

Poll Everywhere

https://bit.ly/2LvG9bq

Which node should we rotate and how?

A. 11, left
B. 20 right
C. 11, right
D. 20, left
E. Ok, I'm not getting this rotation stuff.



Poll Everywhere

https://bit.ly/2LvG9bq

ROTATE WHERE?

Which node should we rotate and how?

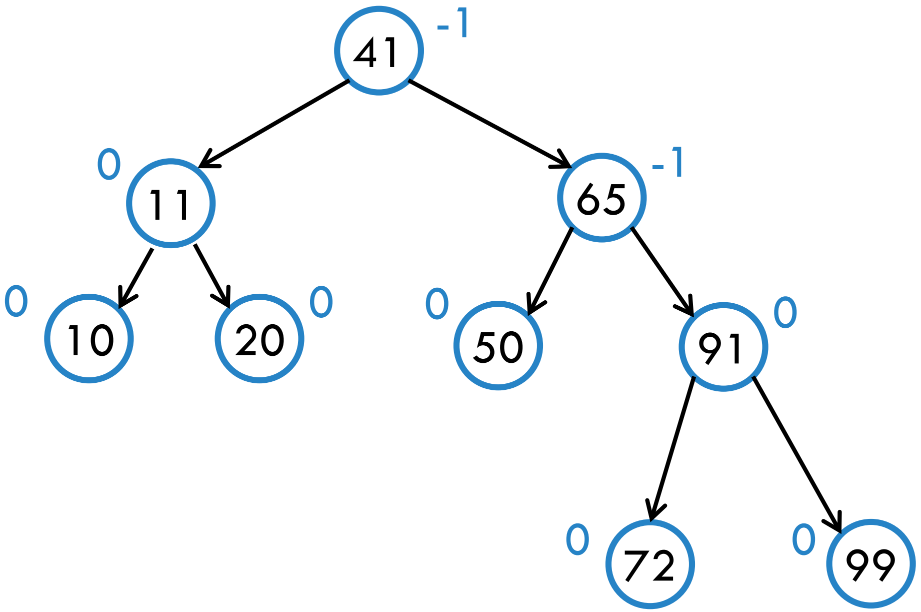
A. 11, left

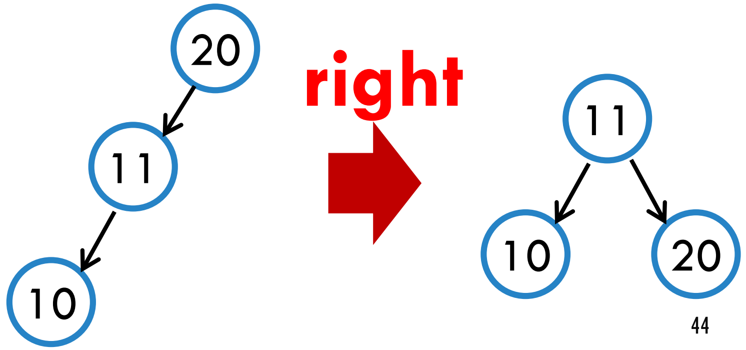
B. 20 right

C. 11, right

D. 20, left

E. Ok, I'm not getting this rotation stuff.



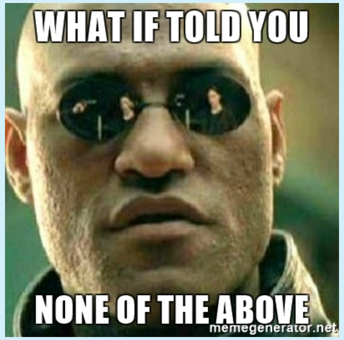


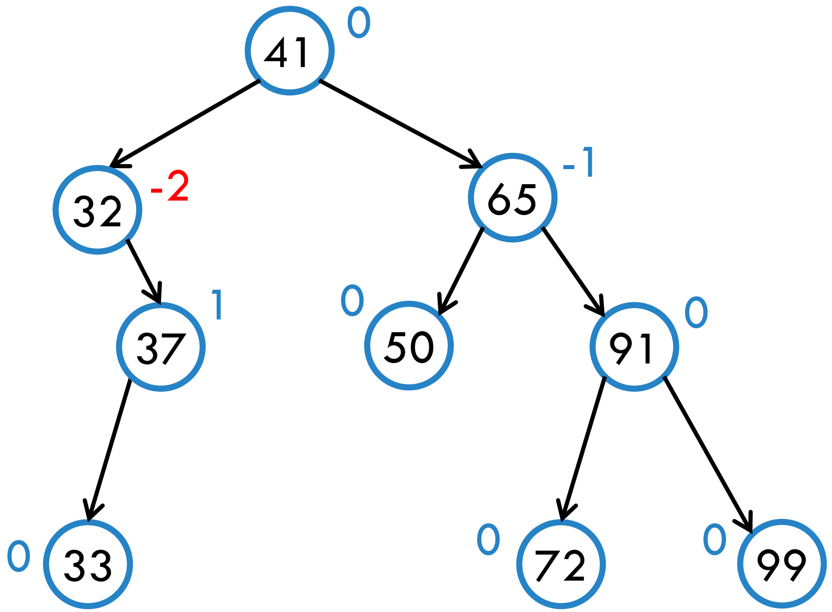
Poll Everywhere

https://bit.ly/2LvG9bq

HOW ABOUT THIS ONE?

Which node should we rotate and how?

A. 32, left
B. 32 right
C. 37, right
D. 41, right
E. 

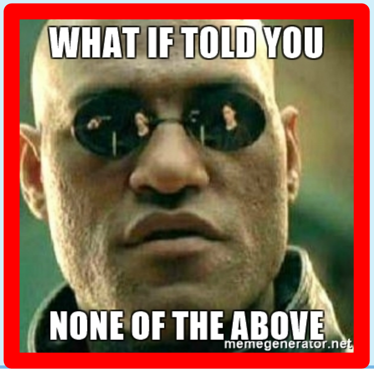


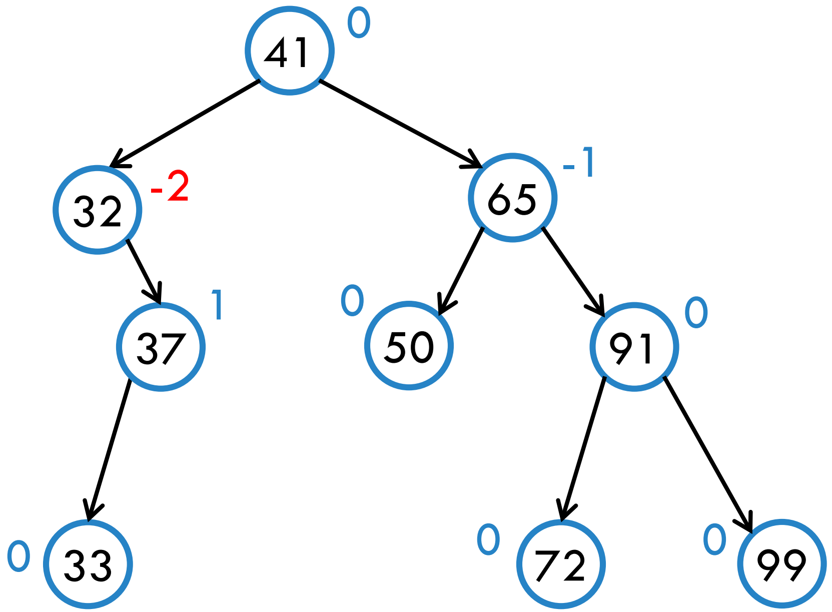
Poll Everywhere

https://bit.ly/2LvG9bq

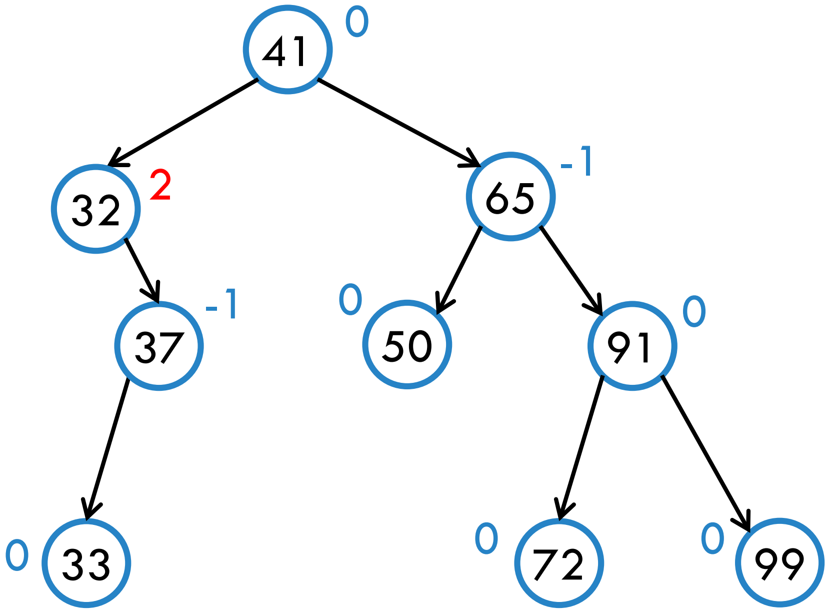
HOW ABOUT THIS ONE?

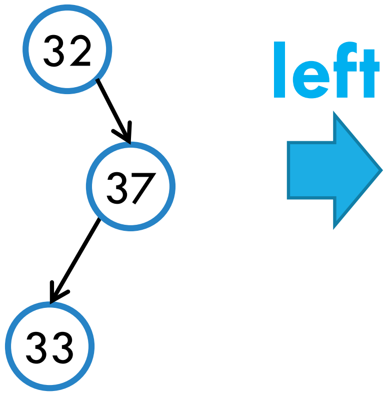
Which node should we rotate and how?

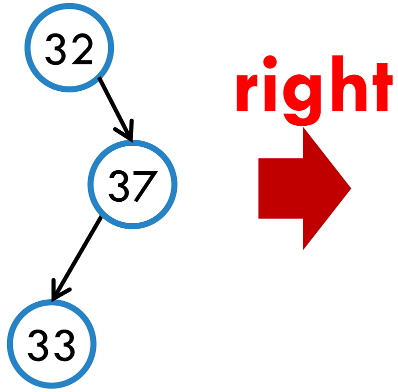
A. 32, left
B. 32 right
C. 37, right
D. 41, right
E. 



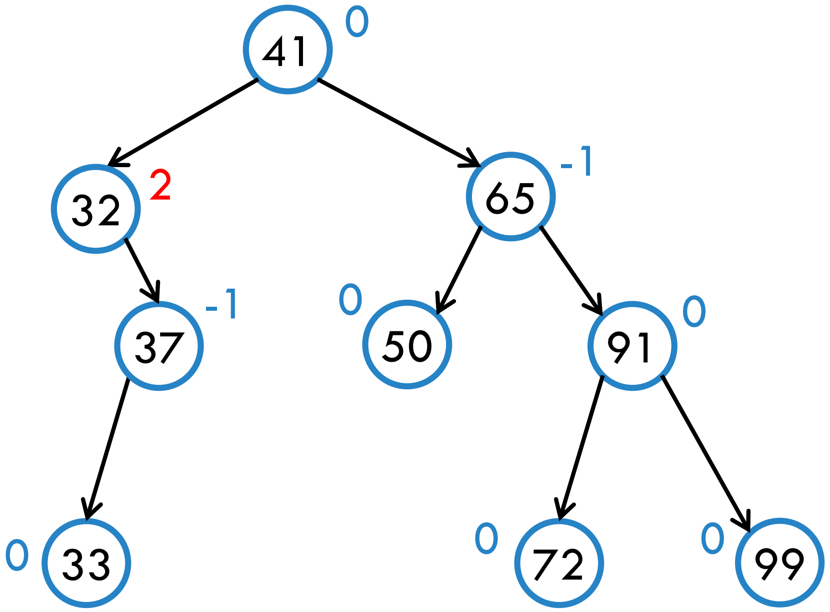
NONE OF THE ABOVE?!

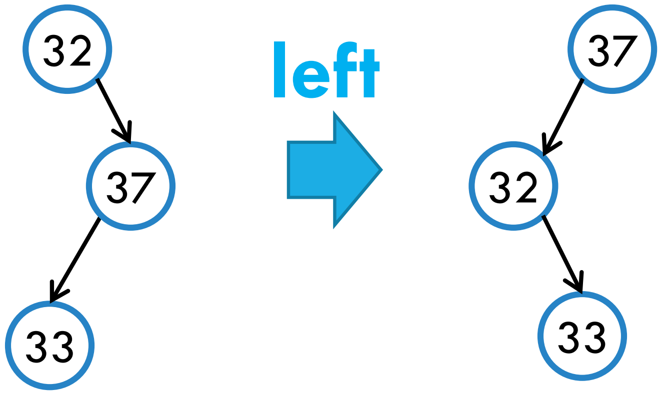


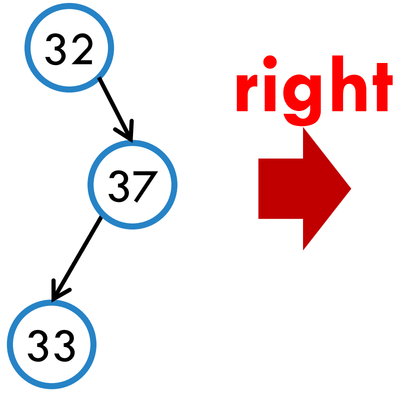




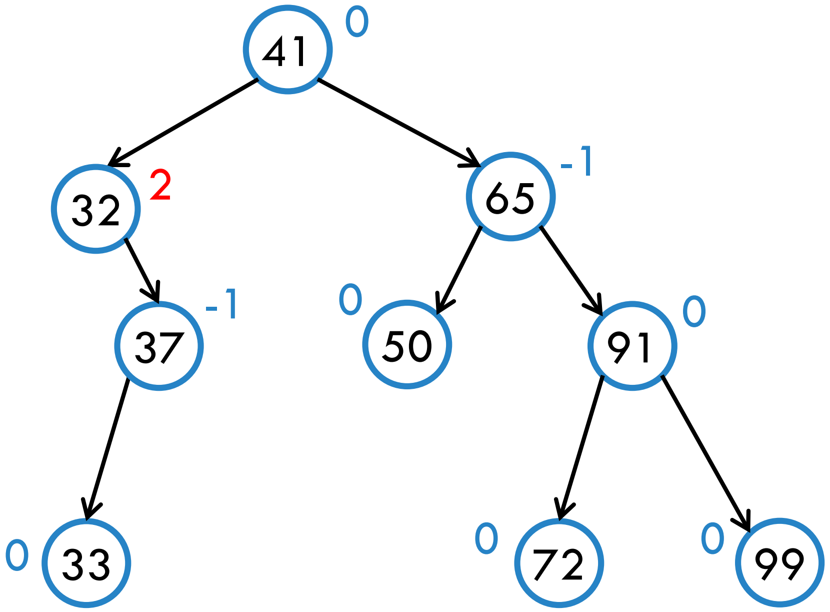
NONE OF THE ABOVE?!

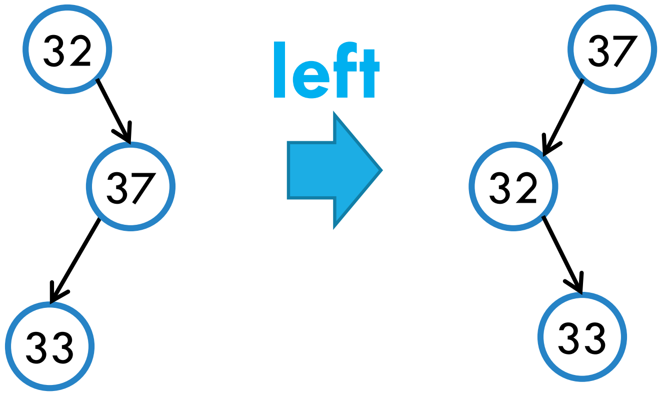


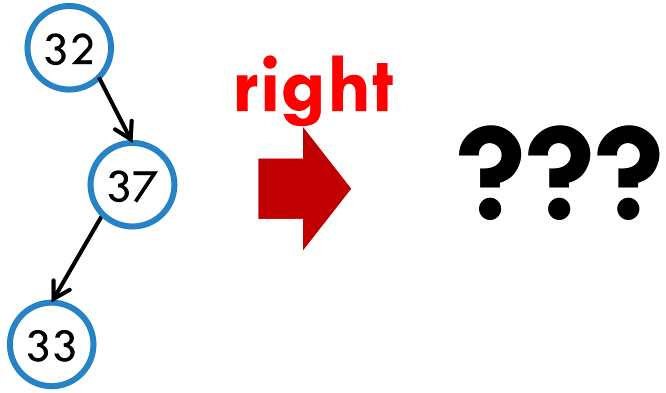




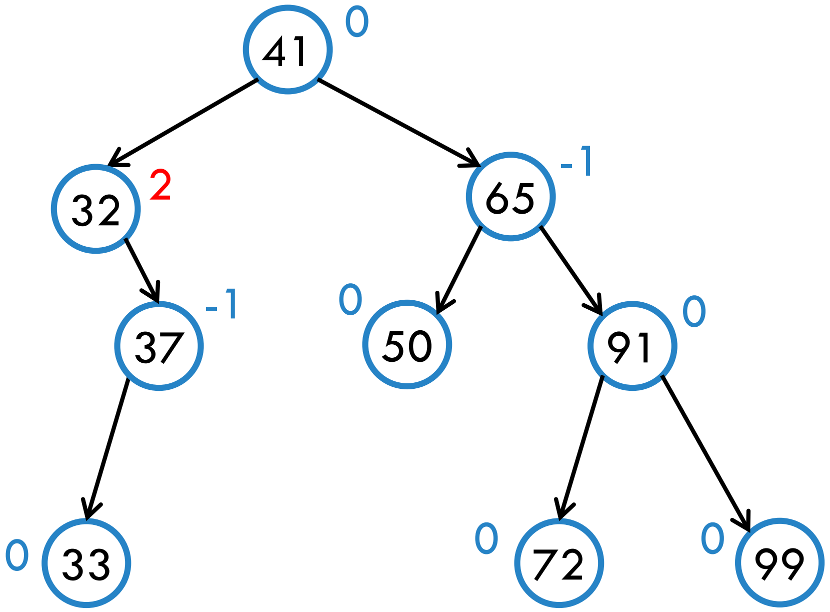
NONE OF THE ABOVE?!

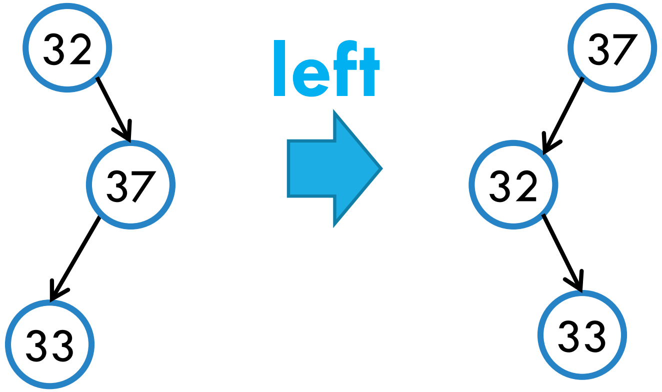


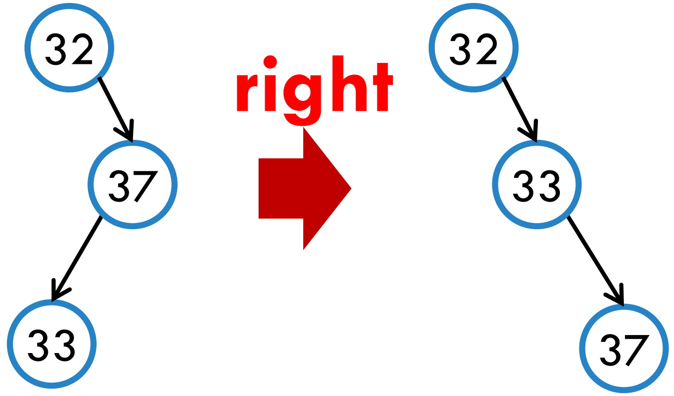




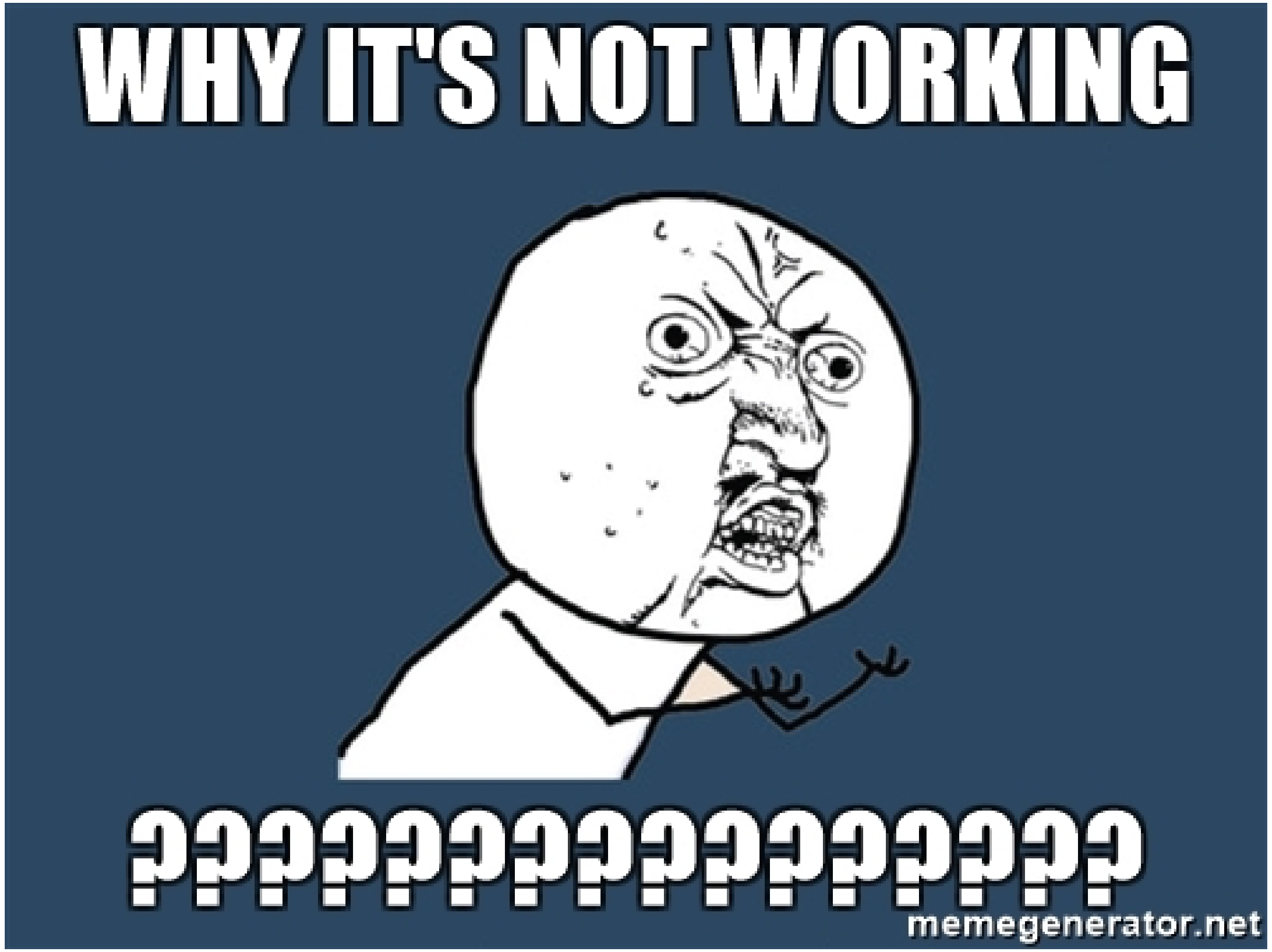
NONE OF THE ABOVE?!







maybe 37 right?


WHY IT'S NOT WORKING
????????????????
memegenerator.net

ROTATIONS

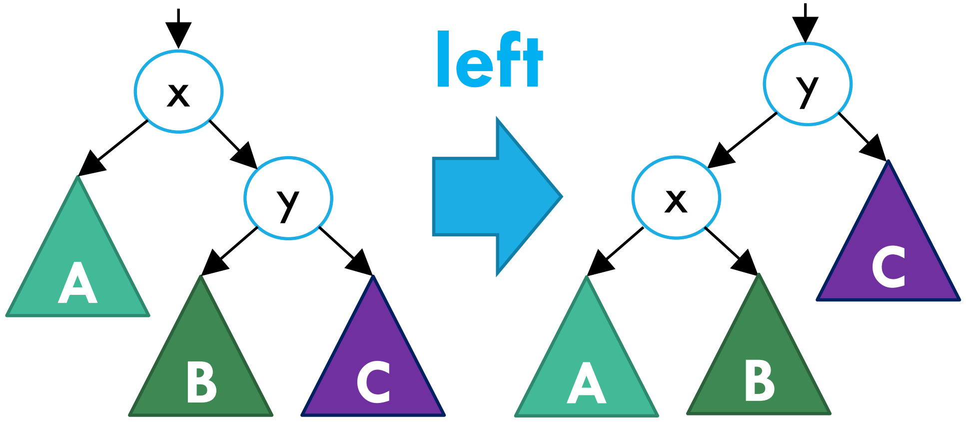
Common "primitive" operation used in balanced search trees.

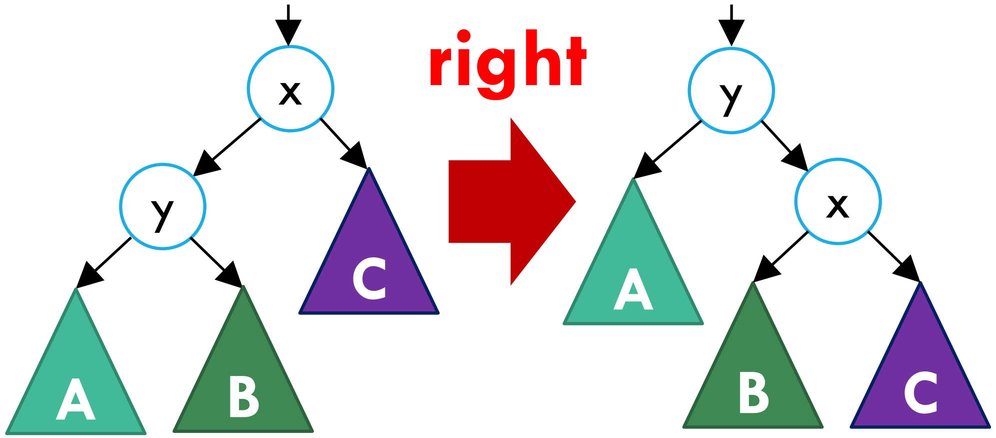
Idea: Locally rebalance subtrees at a given node

4 types:

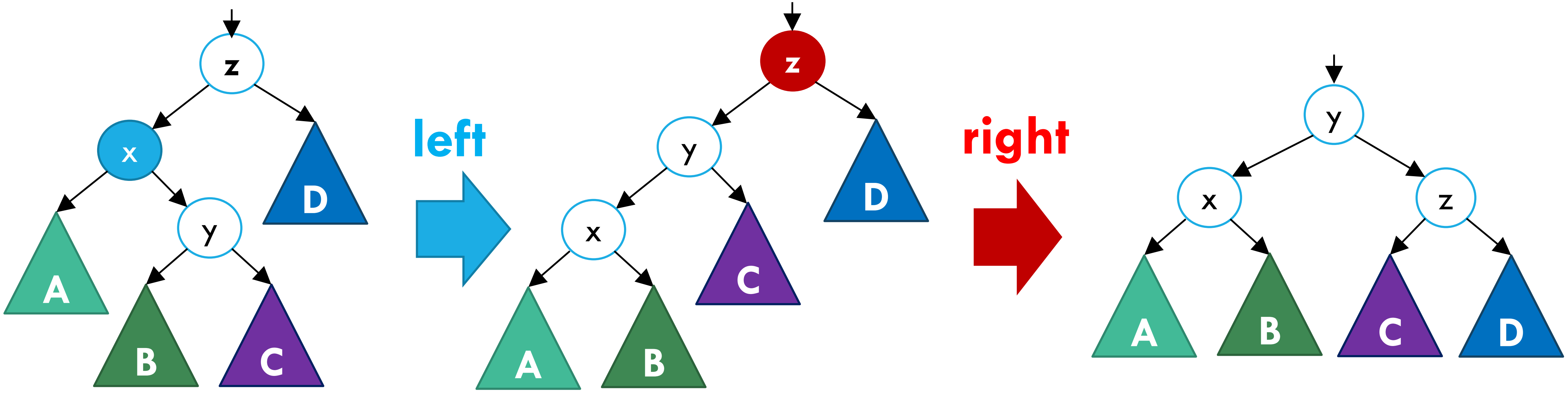
- Left rotation
- Right rotation
- **Left, Right rotation**
- **Right, Left rotation**

Basic rotations



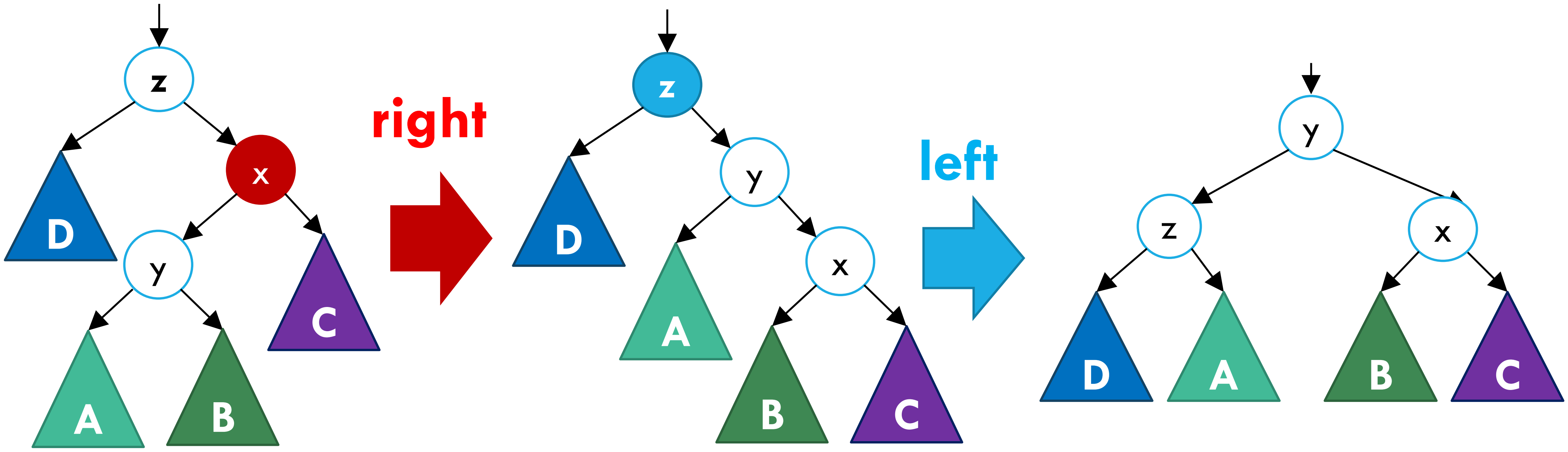


DOUBLE ROTATIONS: LEFT, RIGHT CASE



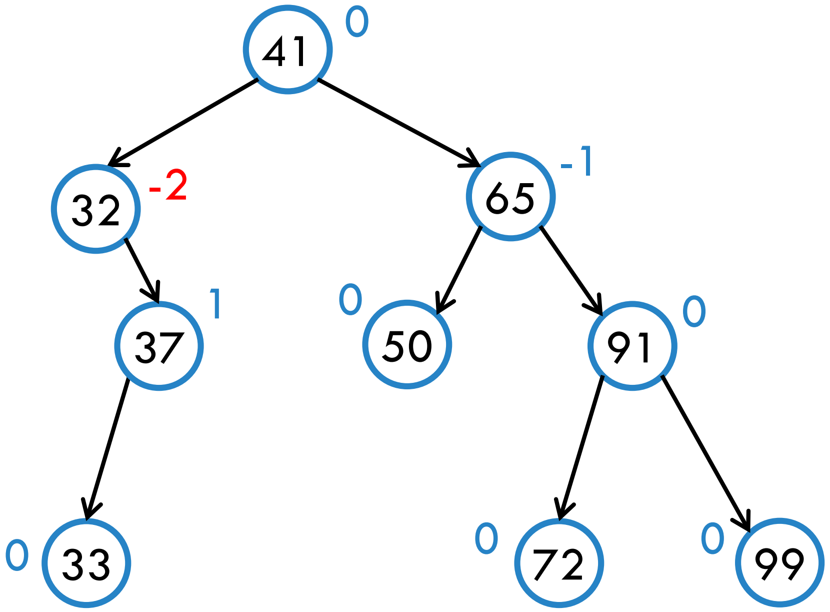

```
function doubleRight(z)
    leftRotate(z.left)
    rightRotate(z)
```

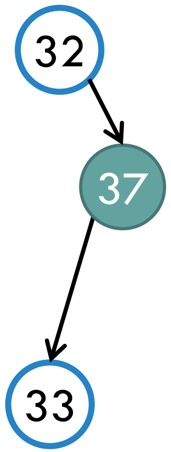
DOUBLE ROTATIONS: RIGHT, LEFT CASE




```
function doubleLeft(z)
    rightRotate(z.right)
    leftRotate(z)
```

LET'S TRY THE DOUBLE ROTATION!

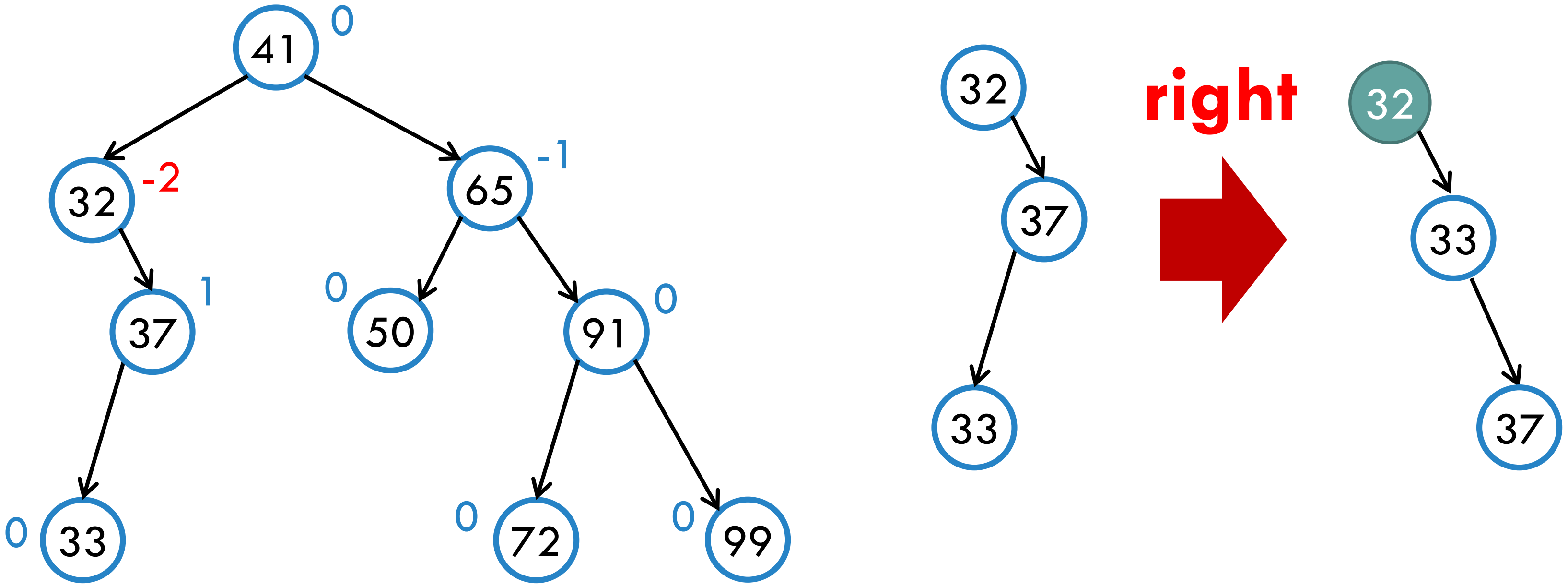




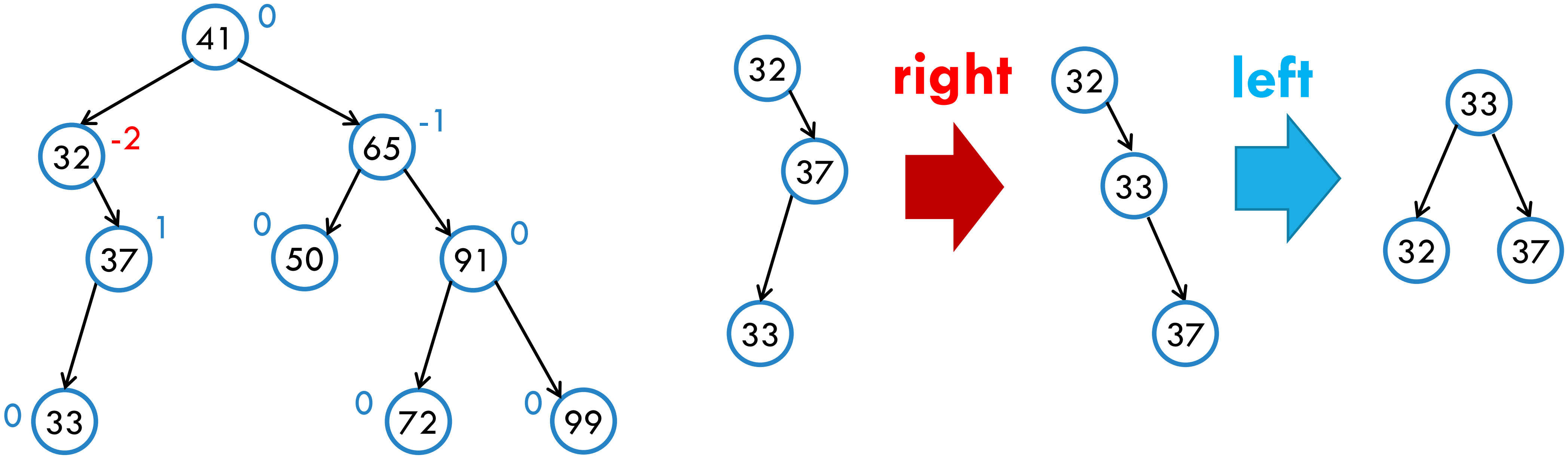
left or

right

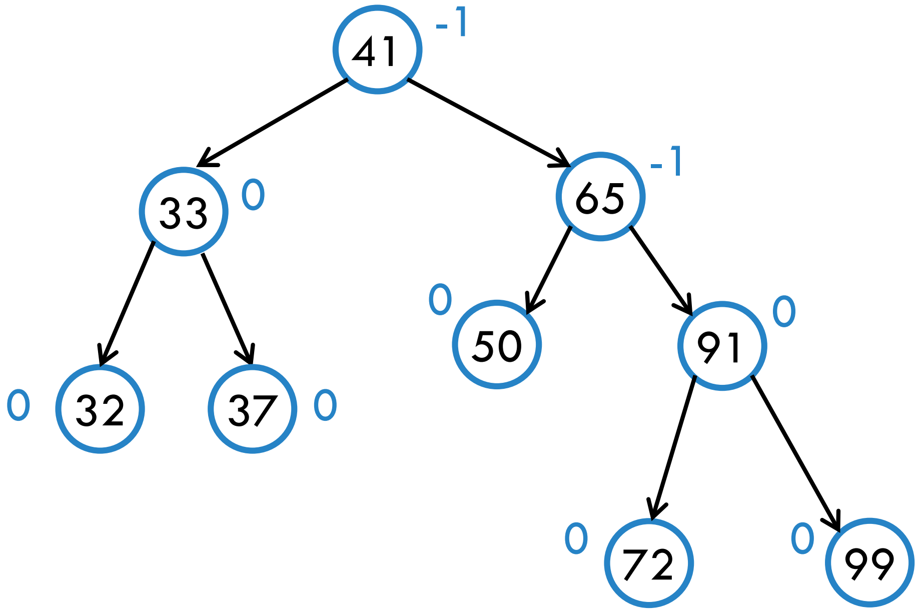
LET'S TRY THE DOUBLE ROTATION!

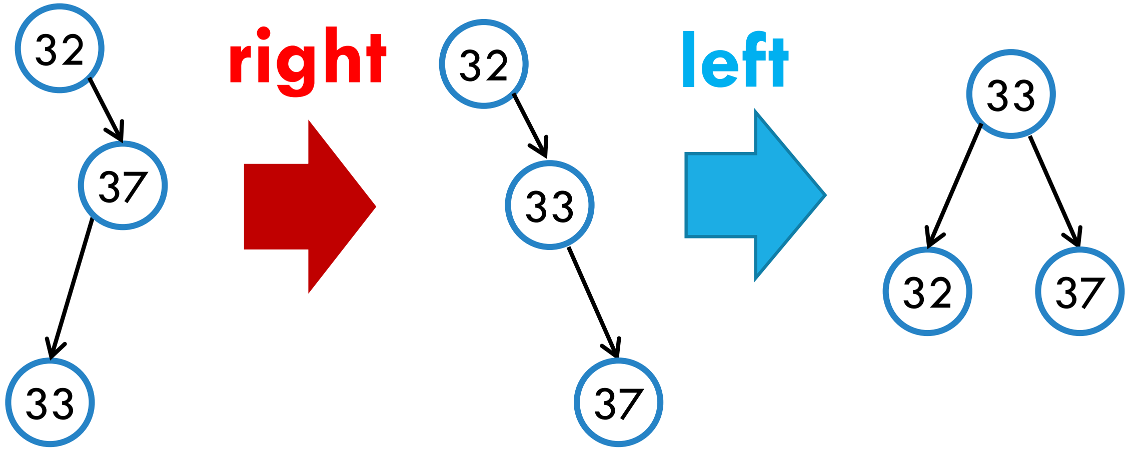


LET'S TRY THE DOUBLE ROTATION!



LET'S TRY THE DOUBLE ROTATION!





It worked!

Poll Everywhere

https://bit.ly/2LvG9bq

WHICH OPERATIONS CAN CAUSE AN IMBALANCE?

Which of these can cause an imbalance in the ~~force~~ tree?

A. insertion
B. deletion
C. search
D. both A & B
E.

Poll Everywhere

https://bit.ly/2LvG9bq

WHICH OPERATIONS CAN CAUSE AN IMBALANCE?

Which of these can cause an imbalance in the ~~force~~ tree?

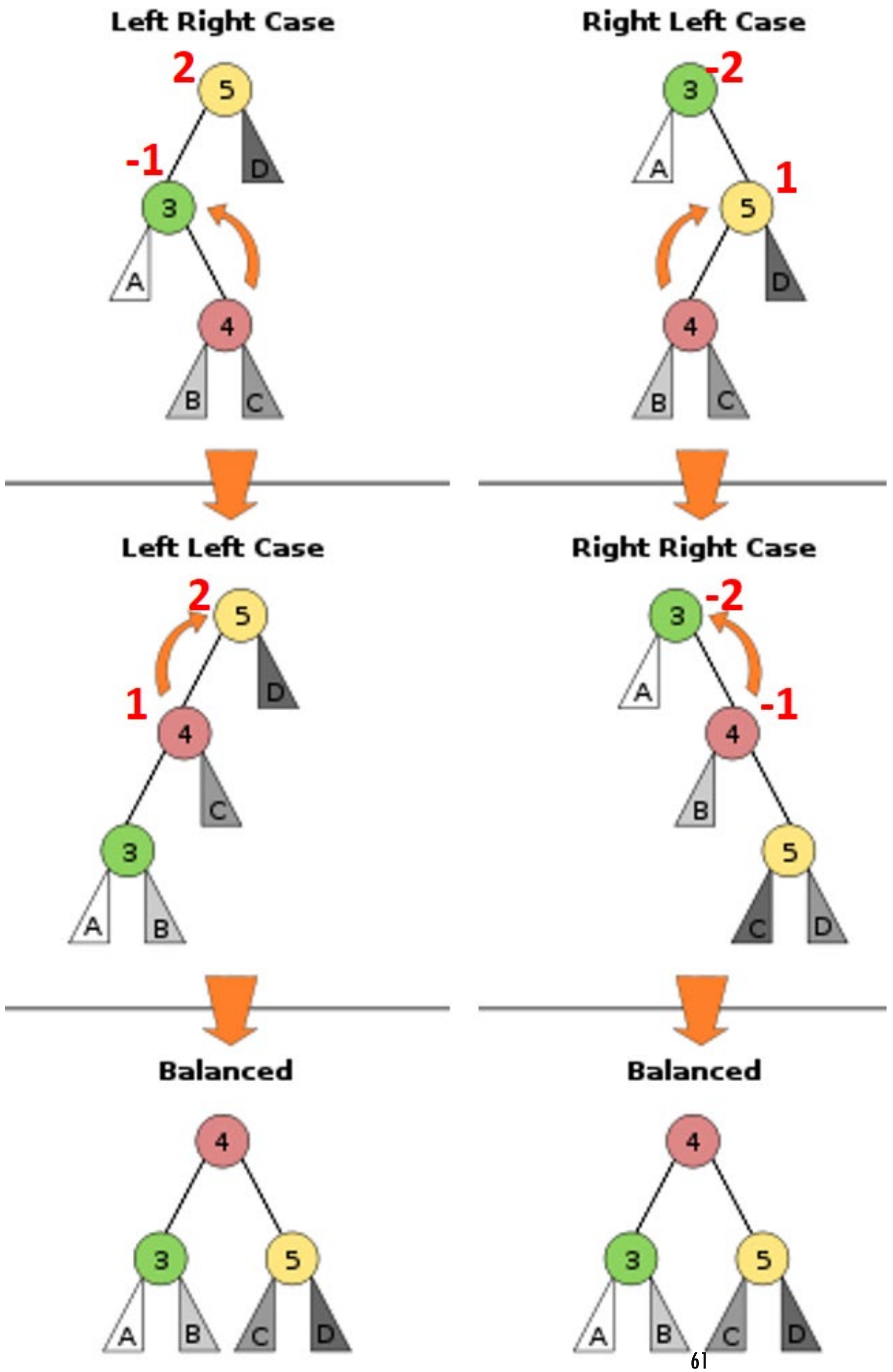
A. insertion
B. deletion
C. search
D. both A & B
E.

ALGORITHM TO FIX IMBALANCES

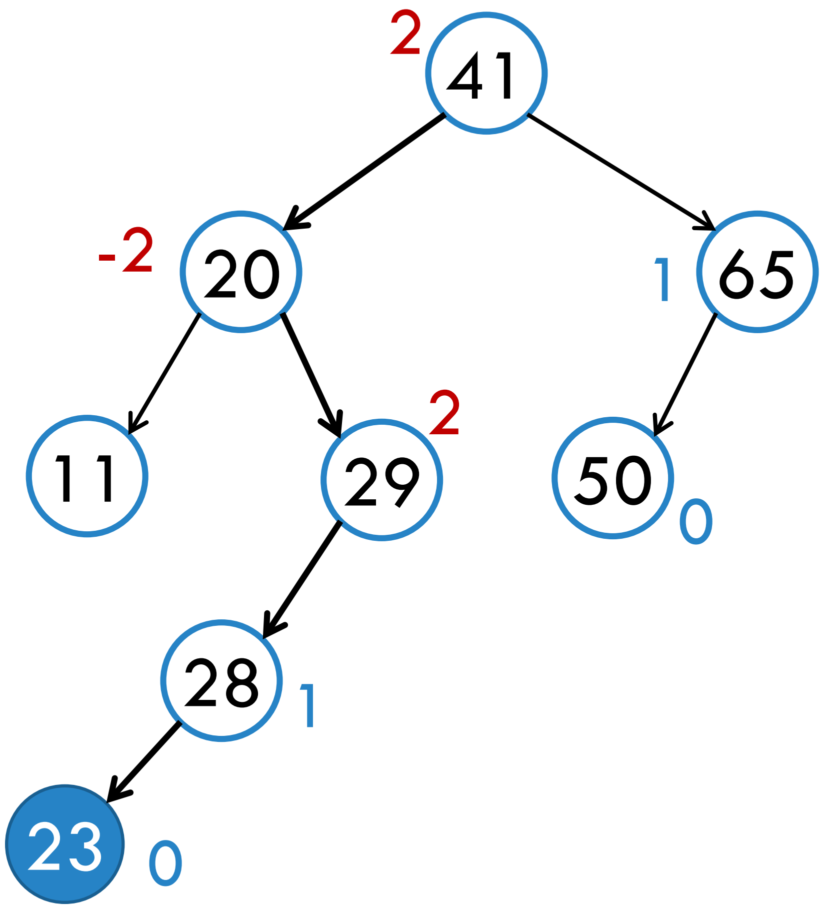
For both insertion and deletions:

- walk up the tree (to the root) from the inserted/deleted node & update balance factors.
- if we find a height-balance violation, fix it via a rotation:

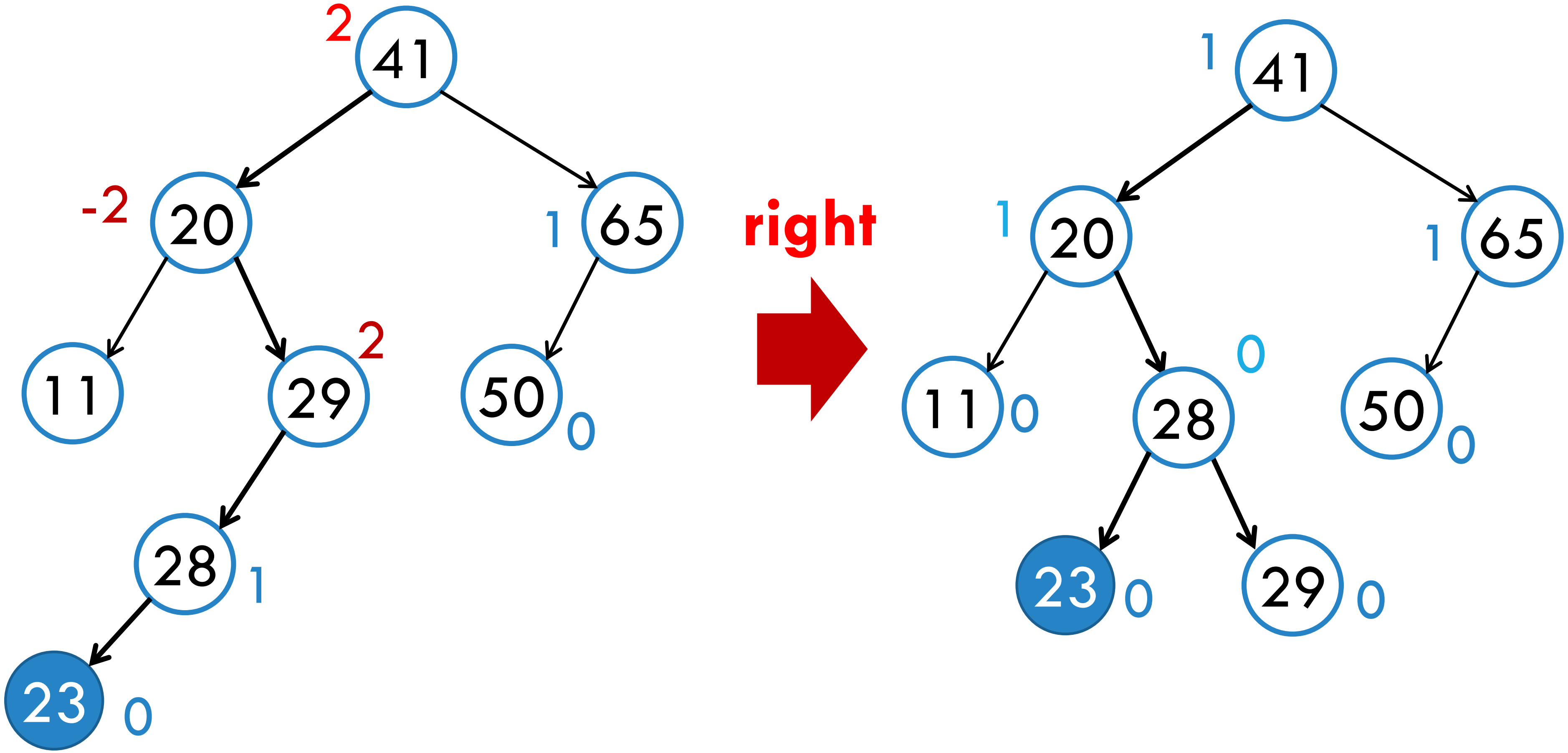
```
if tree is right heavy
      if tree's right subtree is left heavy
            right rotate, left rotate
      else
            left rotate
else if tree is left heavy
      if tree's left subtree is right heavy
            left rotate, right rotate
      else
            right rotate
```



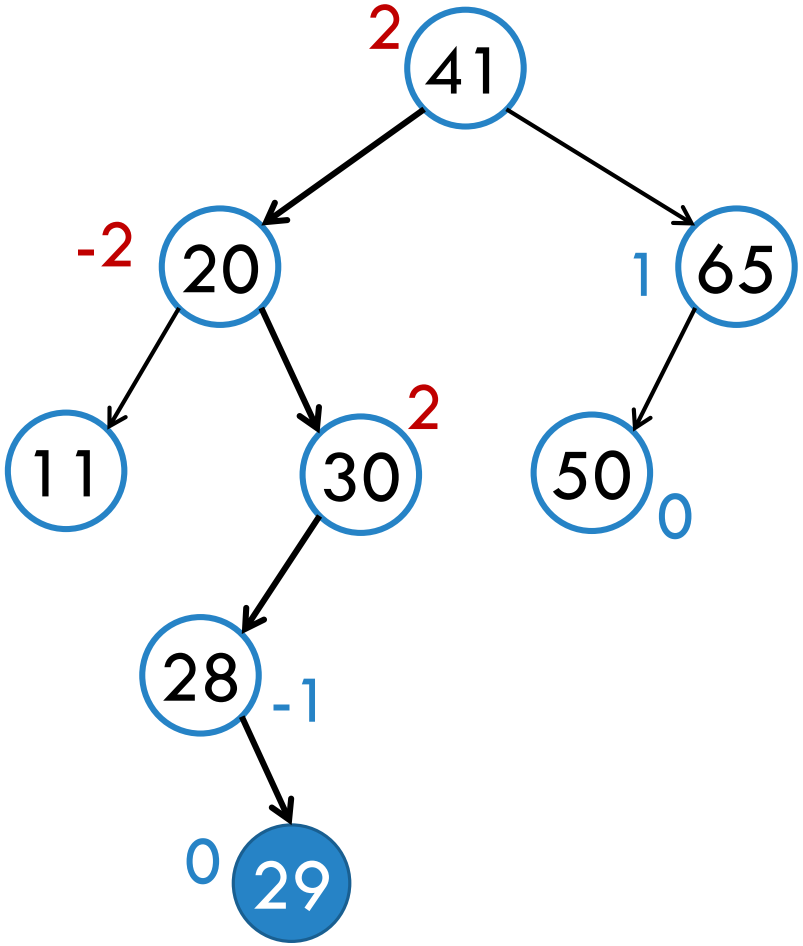
INSERTION EXAMPLE



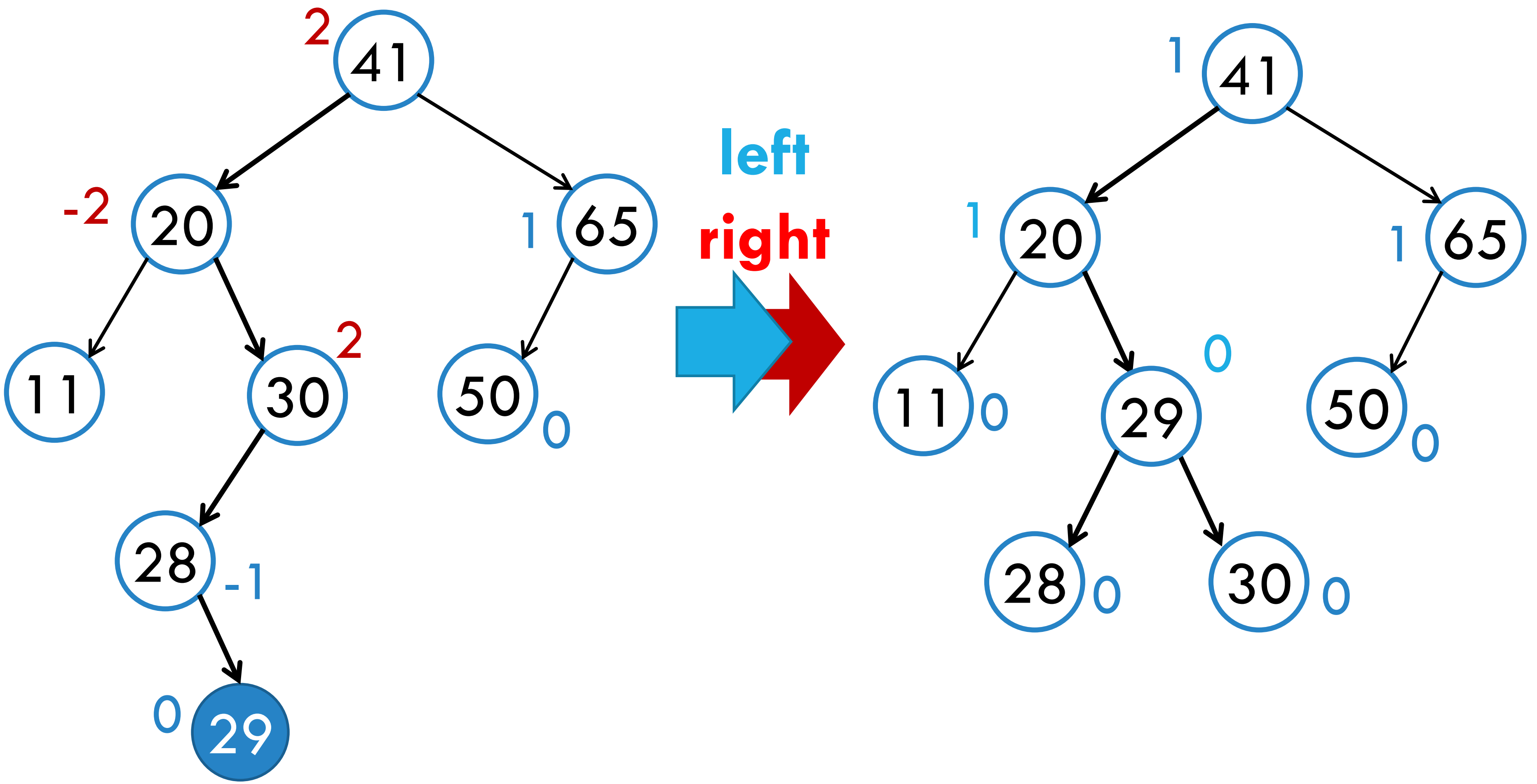
INSERTION EXAMPLE



ANOTHER INSERTION EXAMPLE



ANOTHER INSERTION EXAMPLE



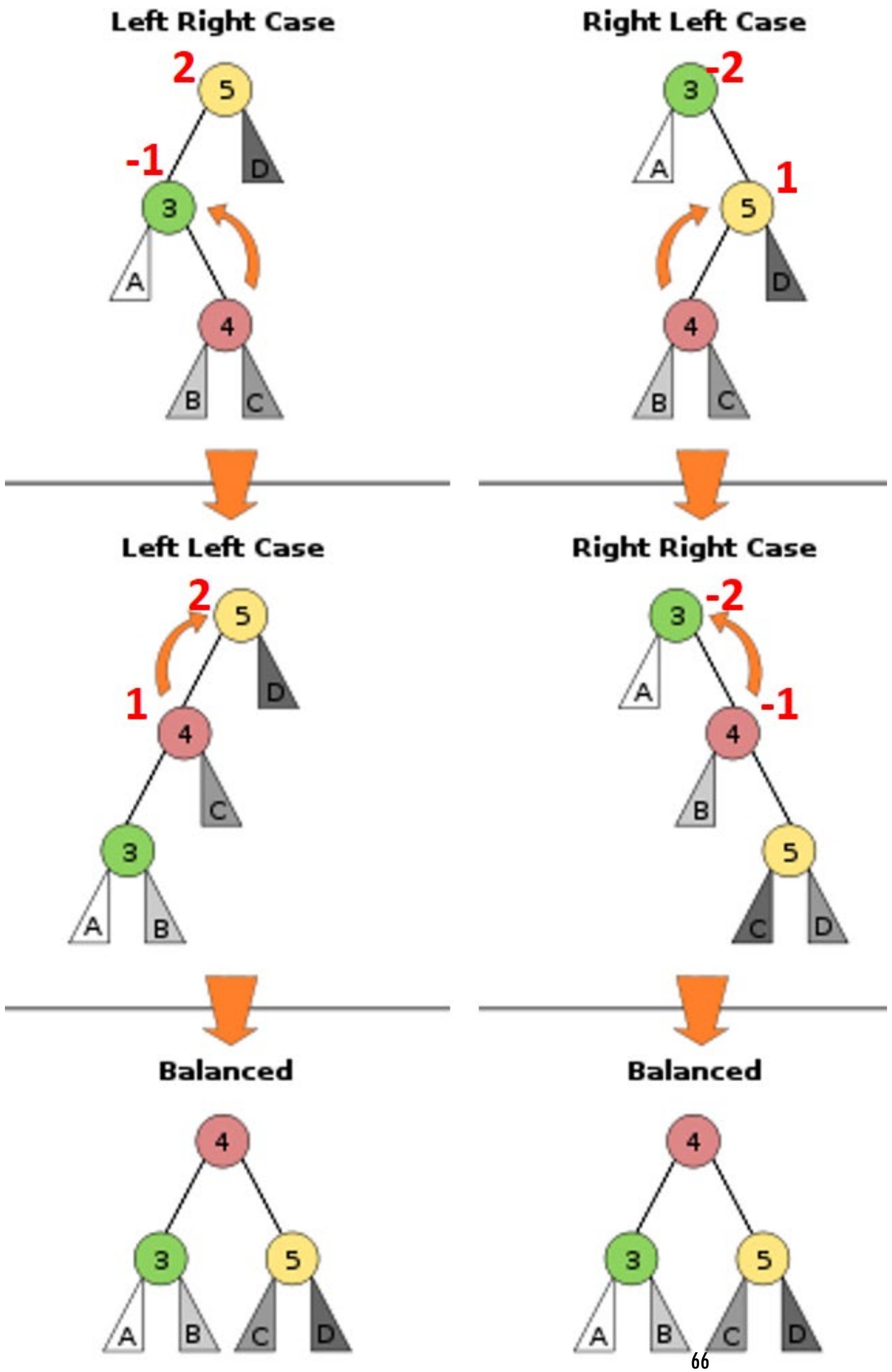
HEIGHT INCREASES?

Can a rotation increase the height of a tree?

A. Yes!

B. No. Rotations either decrease height by one or leave it the same.

C. Why more questions?



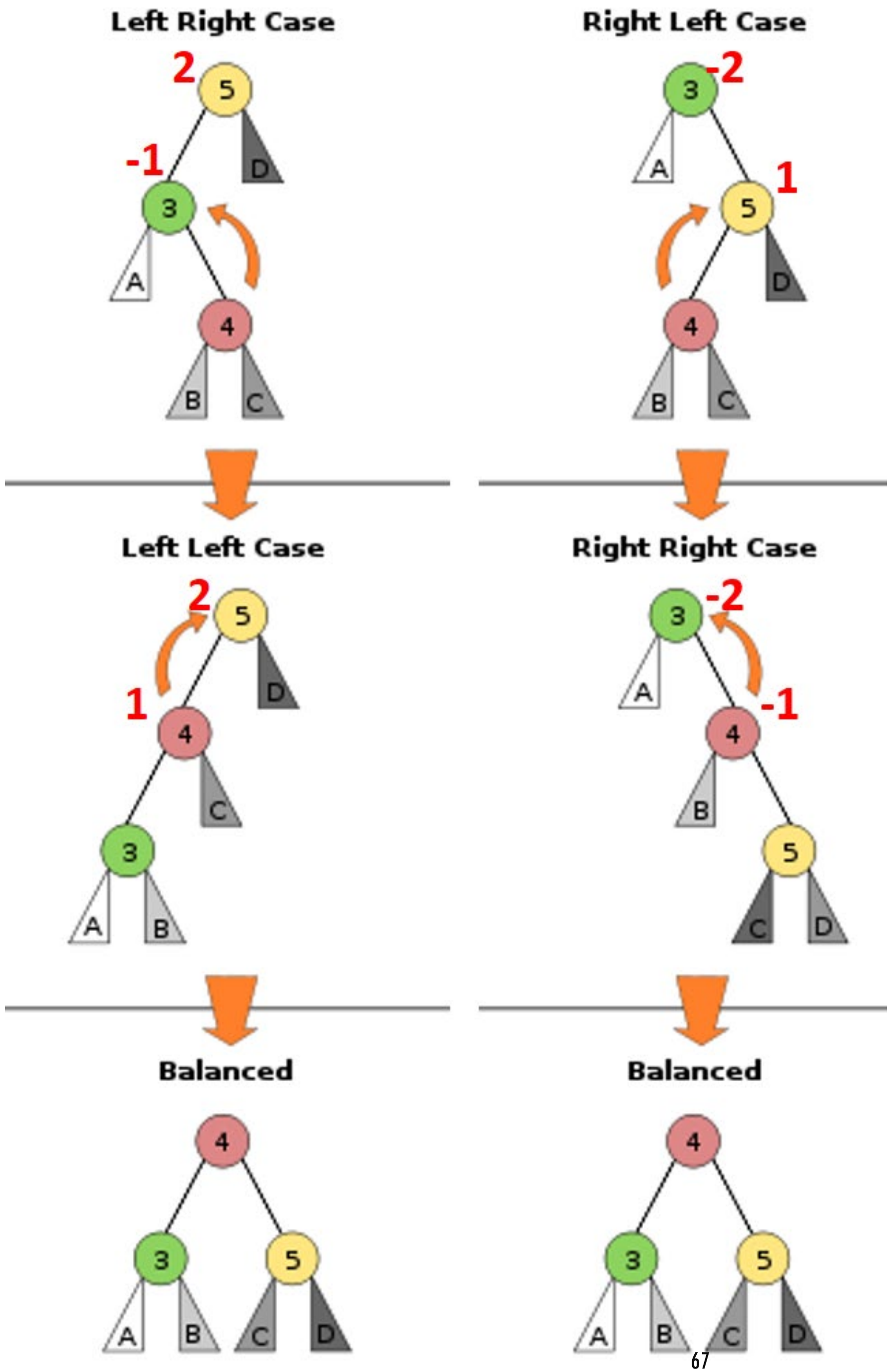
HEIGHT INCREASES?

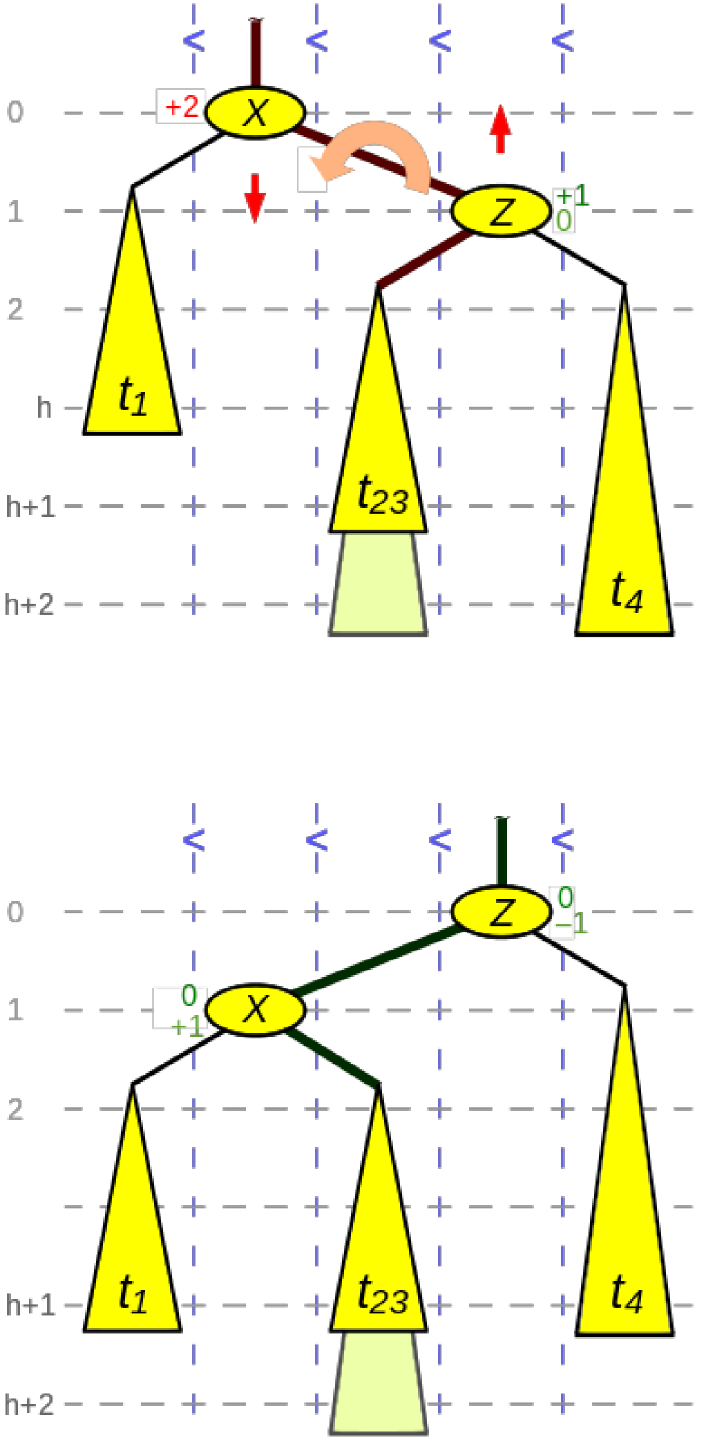
Can a rotation increase the height of a tree?

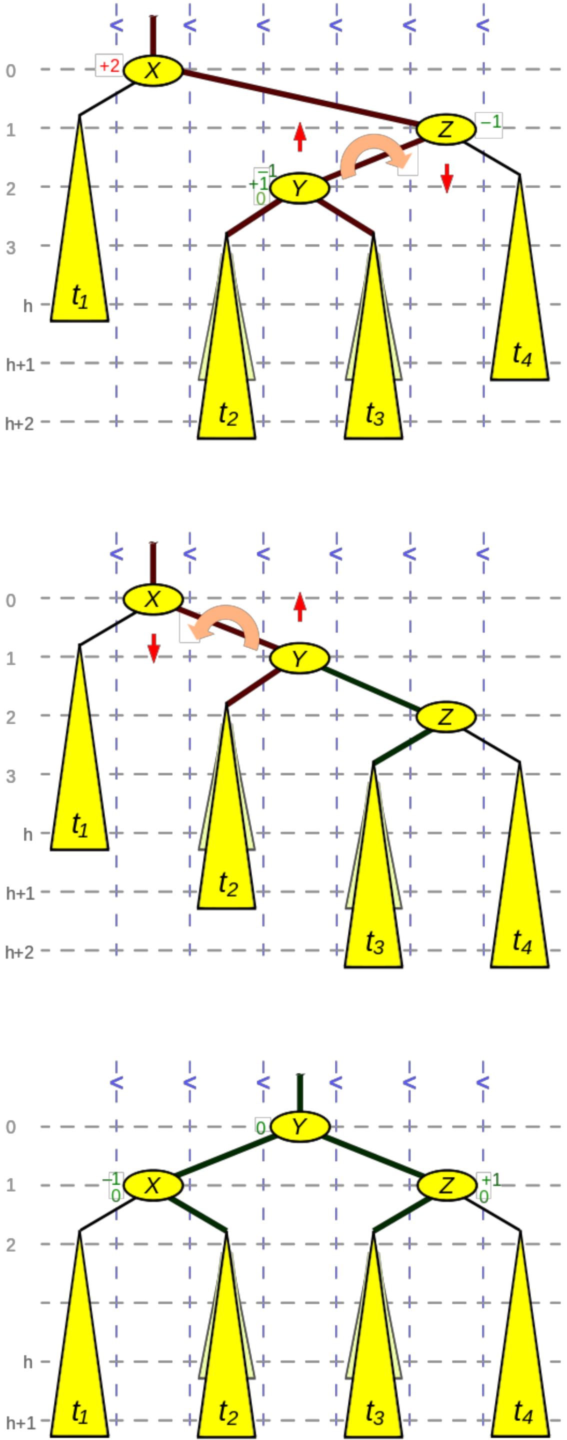
A. Yes!

B. No. Rotations either decrease height by one or leave it the same.

C. Why more questions?





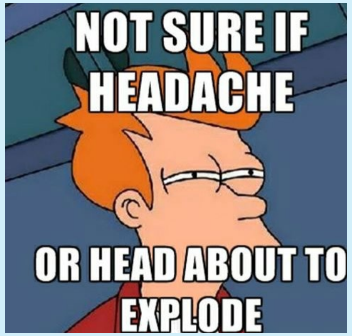


[from wikipedia]

ROTATIONS AFTER AN INSERT?

How many rotations may be required after an insert?

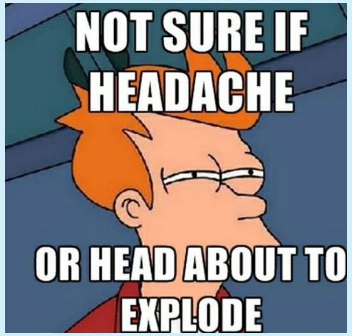
A. O(log n)
B. O(n)
C. O(1)
D.



ROTATIONS AFTER AN INSERT?

How many rotations may be required after an insert?

A. O(log n)
B. O(n)
C. O(1)
D.



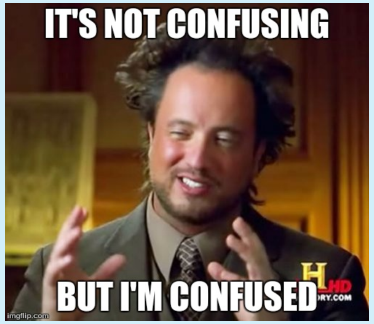
Just have to restore the subtree rooted at the imbalanced node.

ROTATIONS AFTER A DELETE

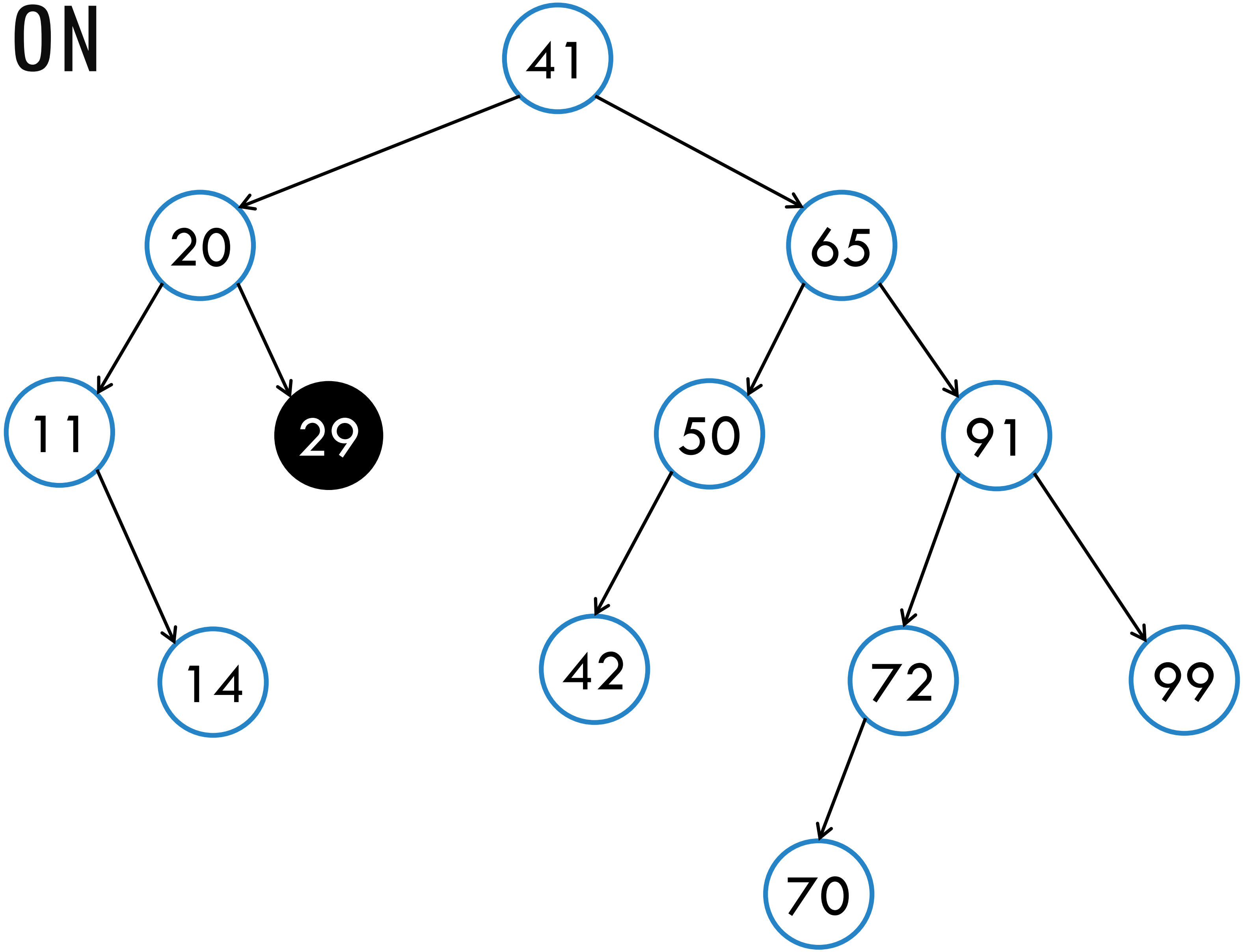
Poll Everywhere

https://bit.ly/2LvG9bq

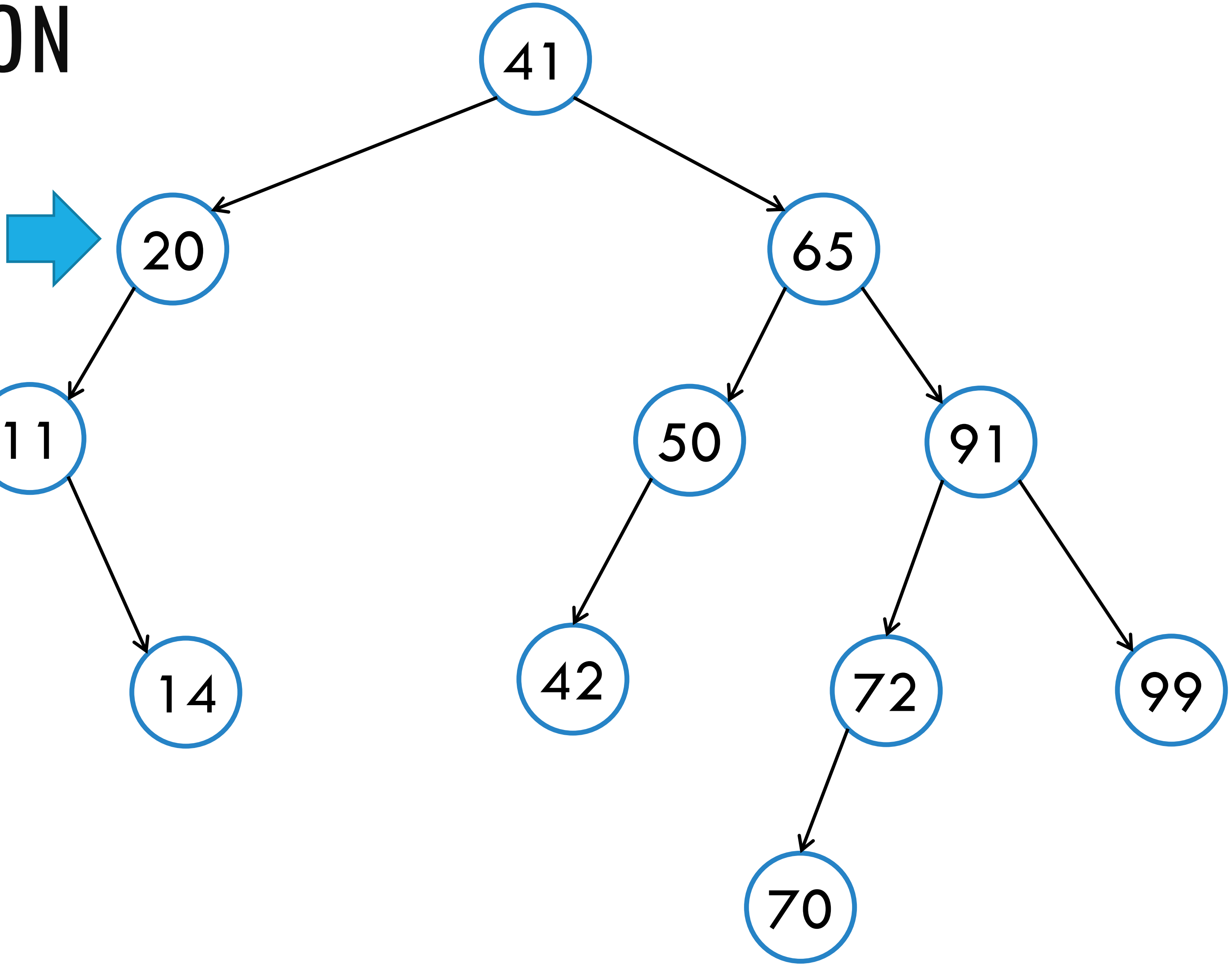
How many rotations may be required after a deletion?

A. O(log n)
B. O(n)
C. O(1)
D. 

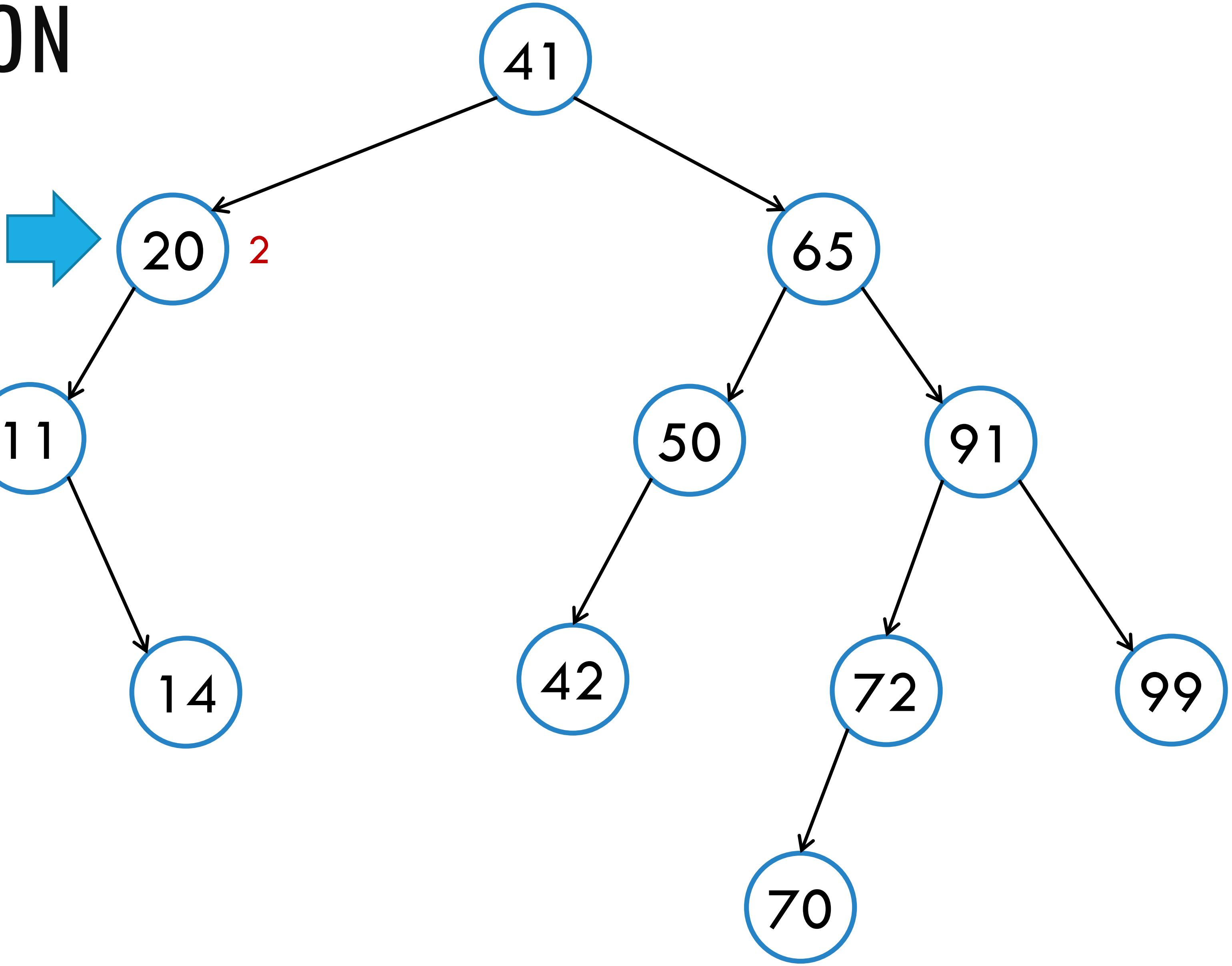
DELETION



DELETION



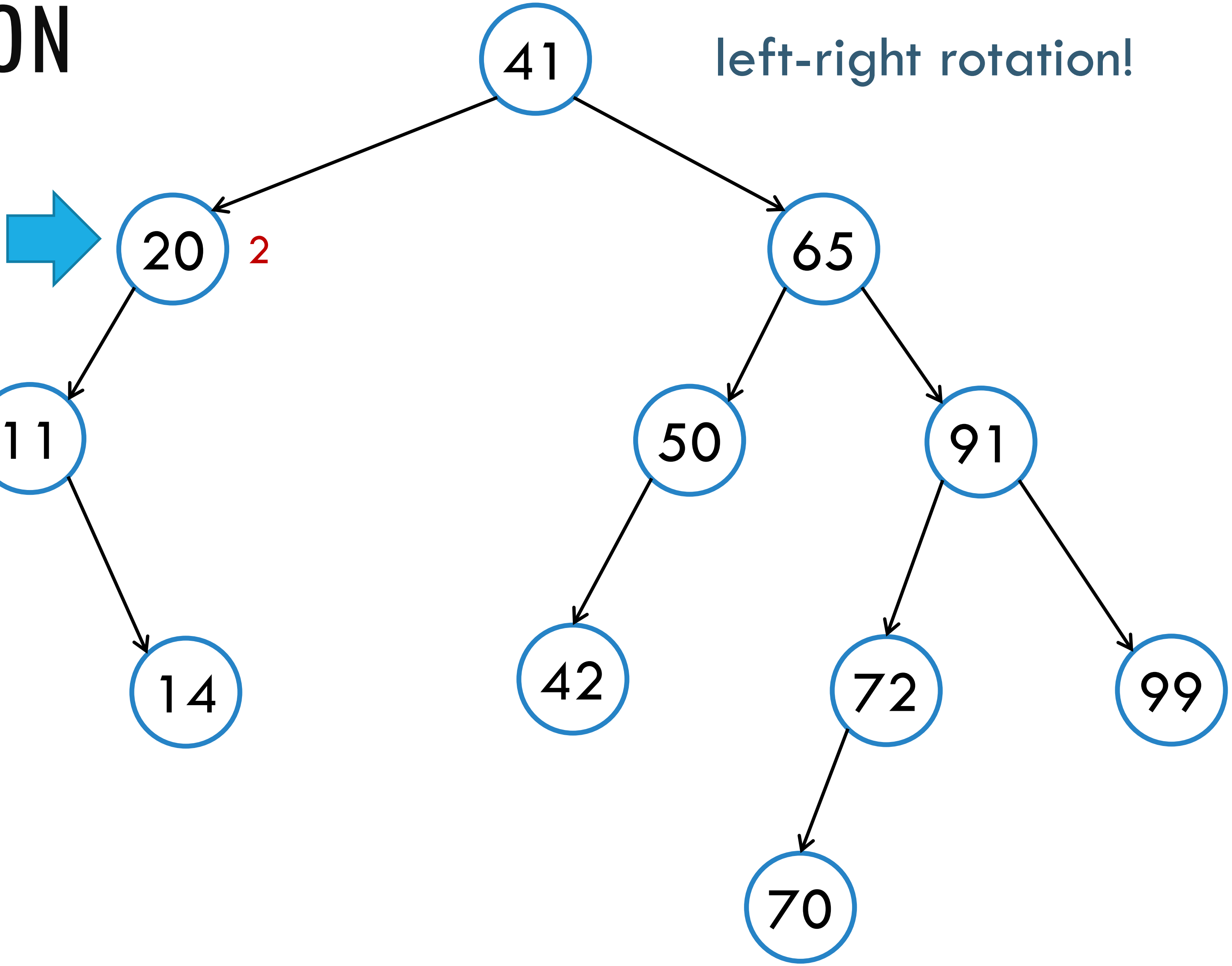
DELETION



DELETION

what kind of rotation do we need?

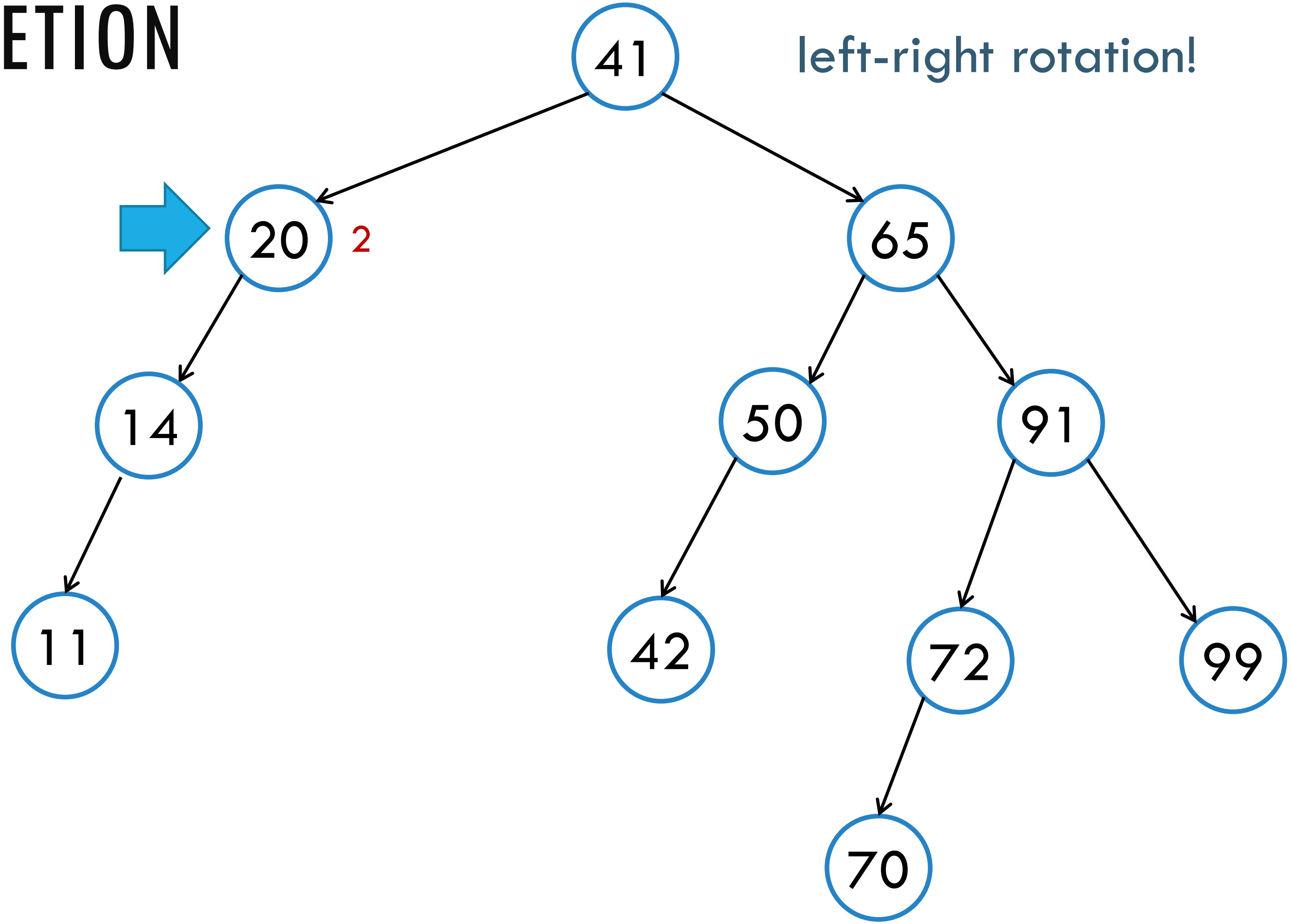
left-right rotation!



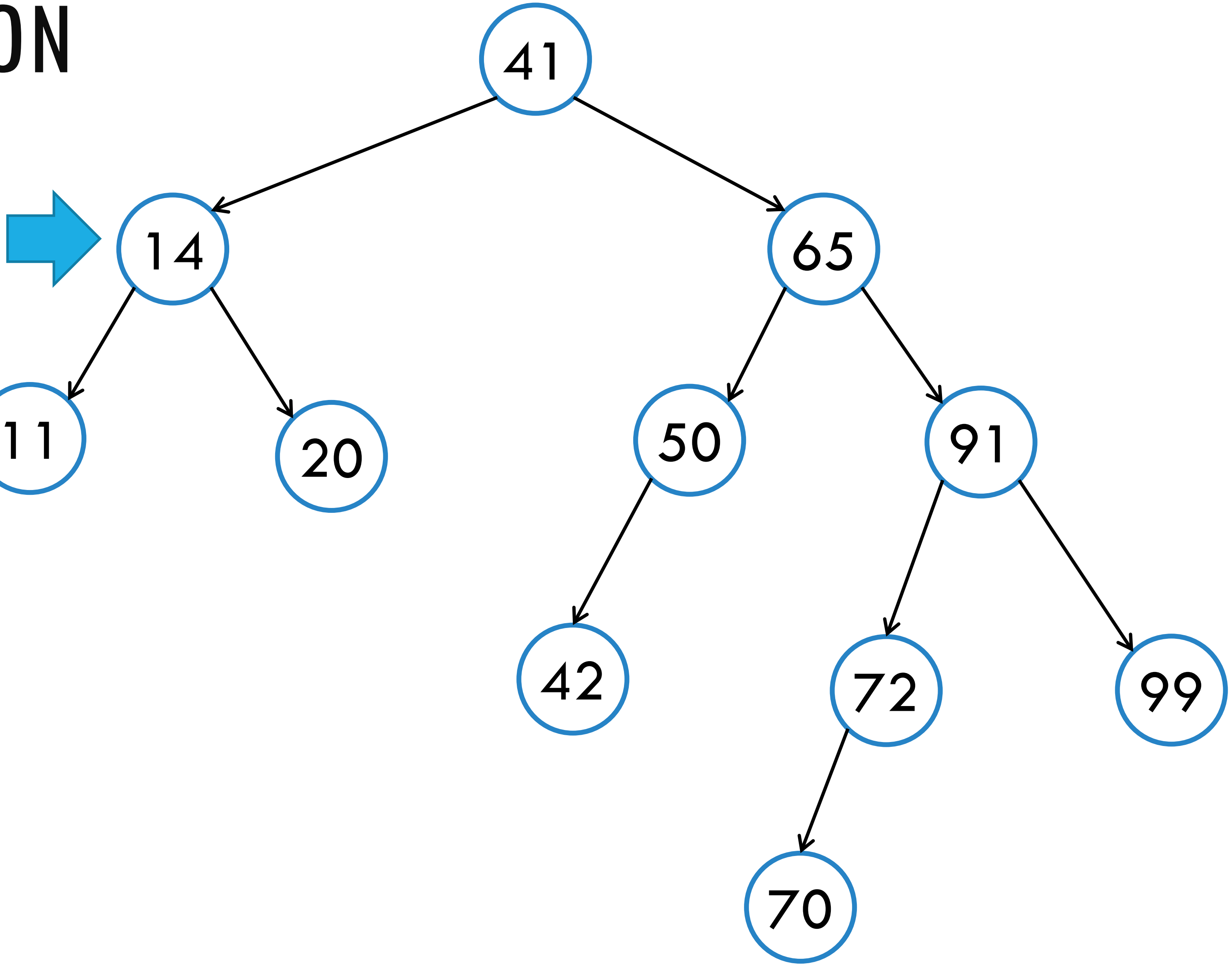
DELETION

what kind of rotation do we need?

left-right rotation!



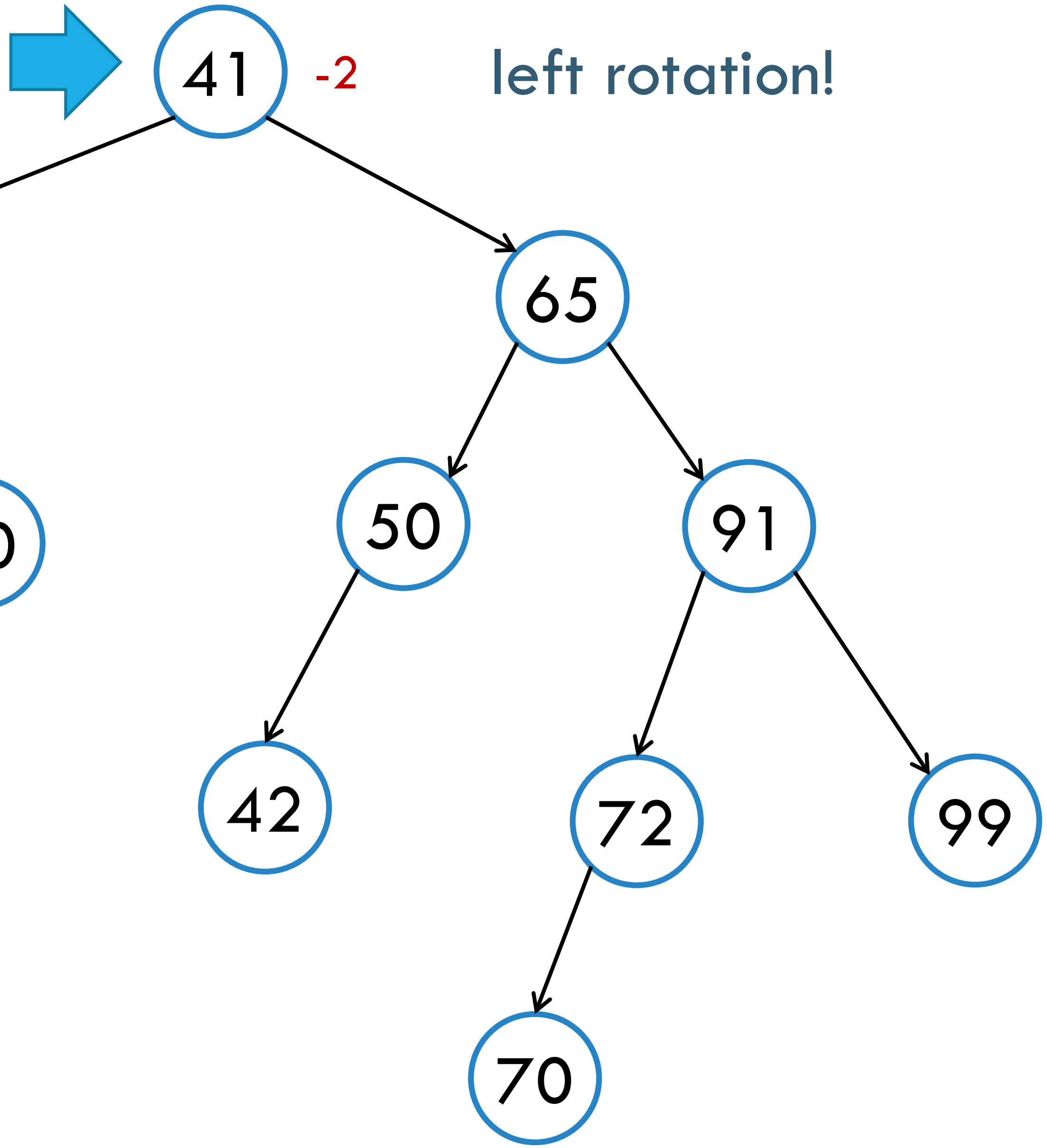
DELETION



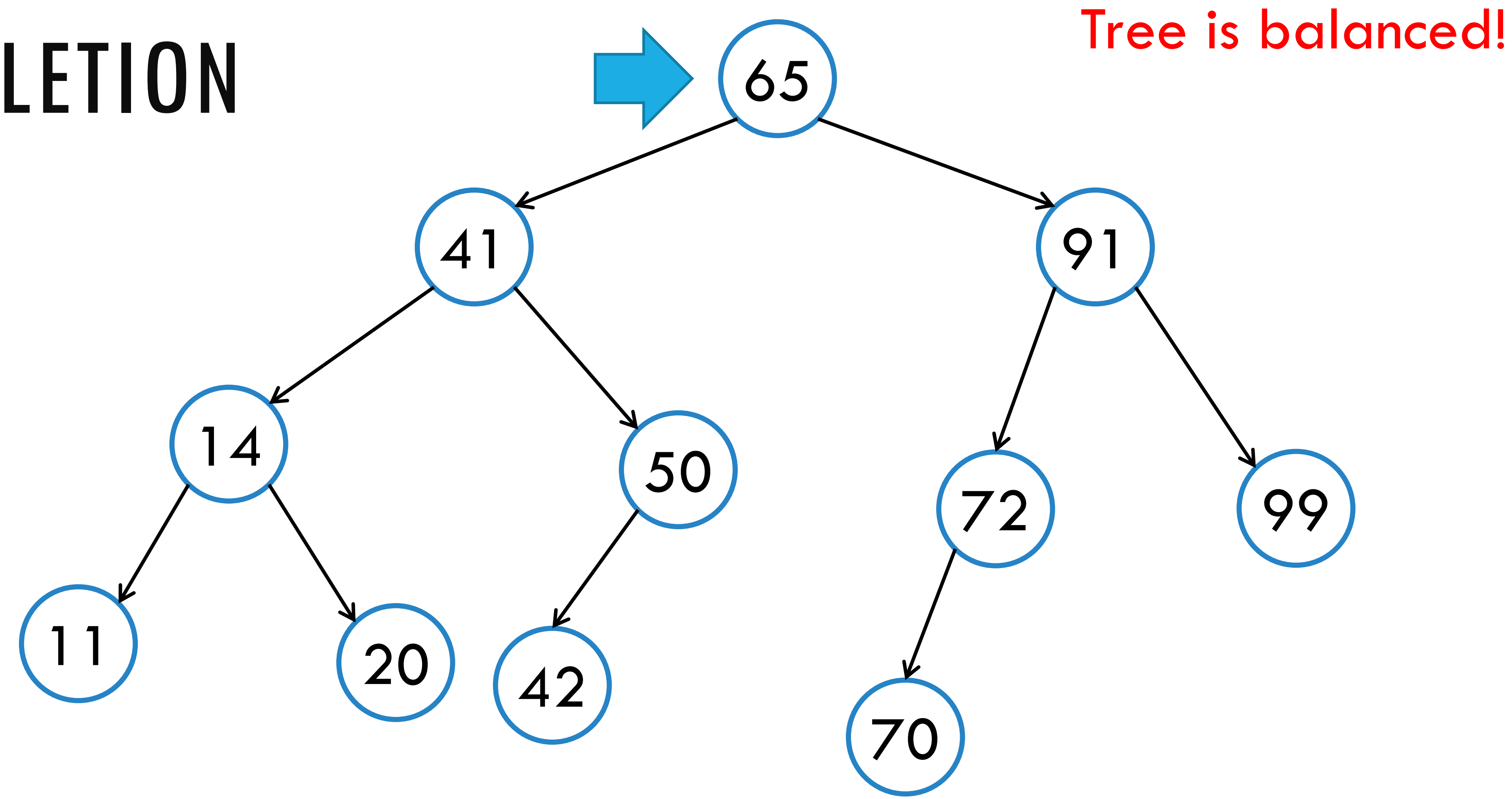
DELETION

what kind of rotation do we need?

left rotation!



DELETION

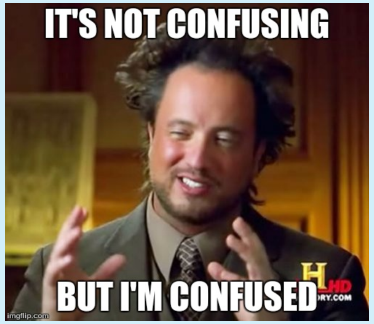


ROTATIONS AFTER A DELETE

Poll Everywhere

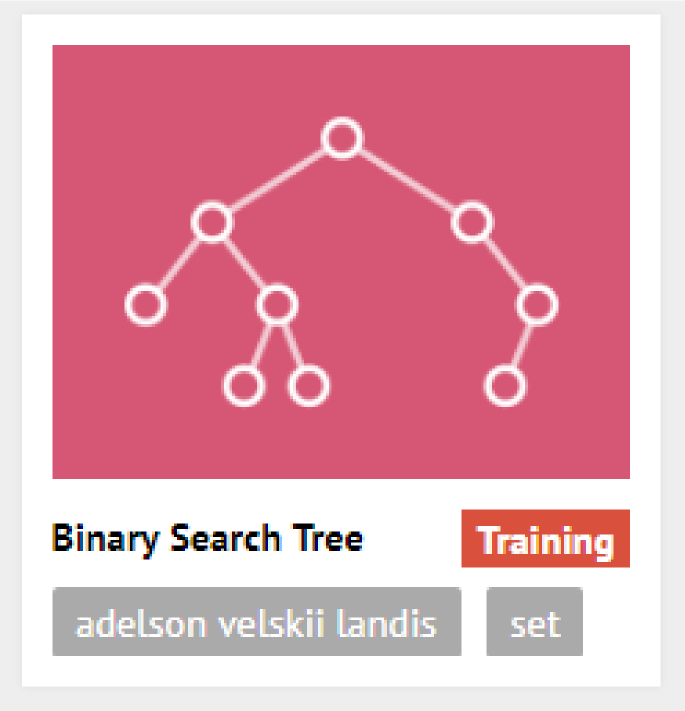
https://bit.ly/2LvG9bq

How many rotations may be required after a deletion?

A. **O(log n)**
B. O(n)
C. O(1)
D. 

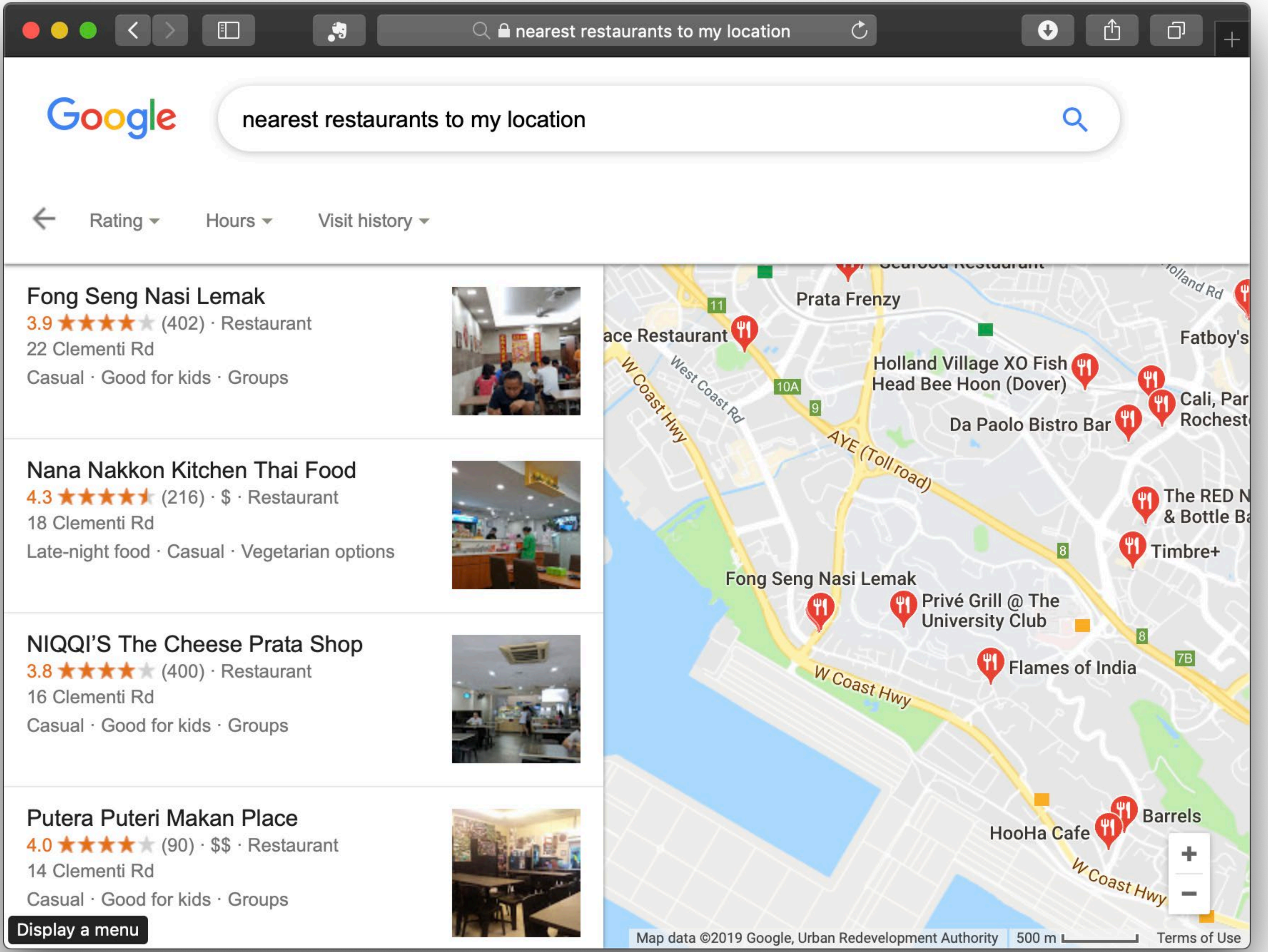
May have to rotate all the way up to the root.

SOME PRACTICE ON VISUALGO



PROBLEM: NEAREST RESTAURANT

Given n two-dimensional points $P = \{p_i = (x_i, y_i)\}_{i=1}^{N}$, can find the closest point to a query point $q = (a, b)$?



PROBLEM: NEAREST RESTAURANT

Given n two-dimensional points $P = \{p_i = (x_i, y_i)\}_{i=1}^{N}$, can find the closest point to a query point $q = (a, b)$?

What is the most straight-forward algorithm?

Just compute the distance of each point p_i to q, and report the one with the minimum distance.

Time-Complexity?

$O(n)$

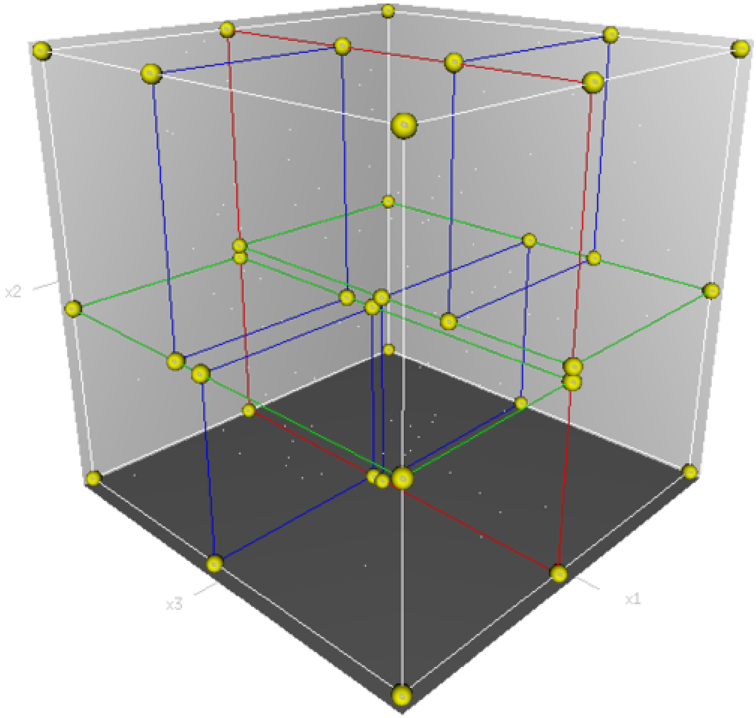
Can we do better?

A BETTER WAY: KD-TREES

Invented by Jon Bentley in the 1970s

Name originally meant for k to represent number of dimensions, e.g., "3d-trees", "4d-trees" etc.

Idea: we will store points in a binary tree



Slides borrow material from: https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf and Wikipedia article in kdtrees https://en.wikipedia.org/wiki/K-d_tree

KD-TREES

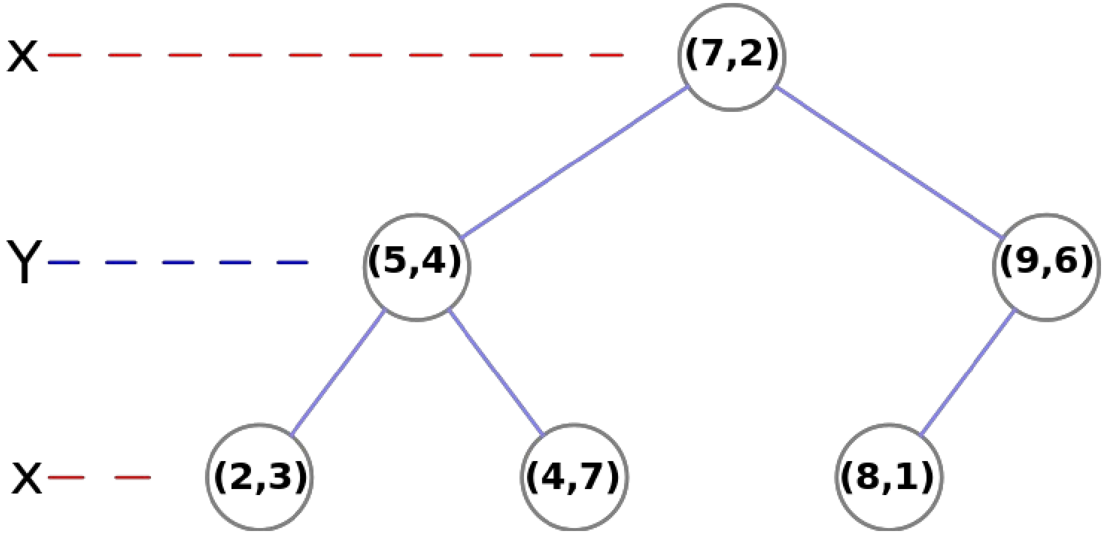
An extension of BSTs to k-dimensional keys!

Each level has a “cutting/split dimension”

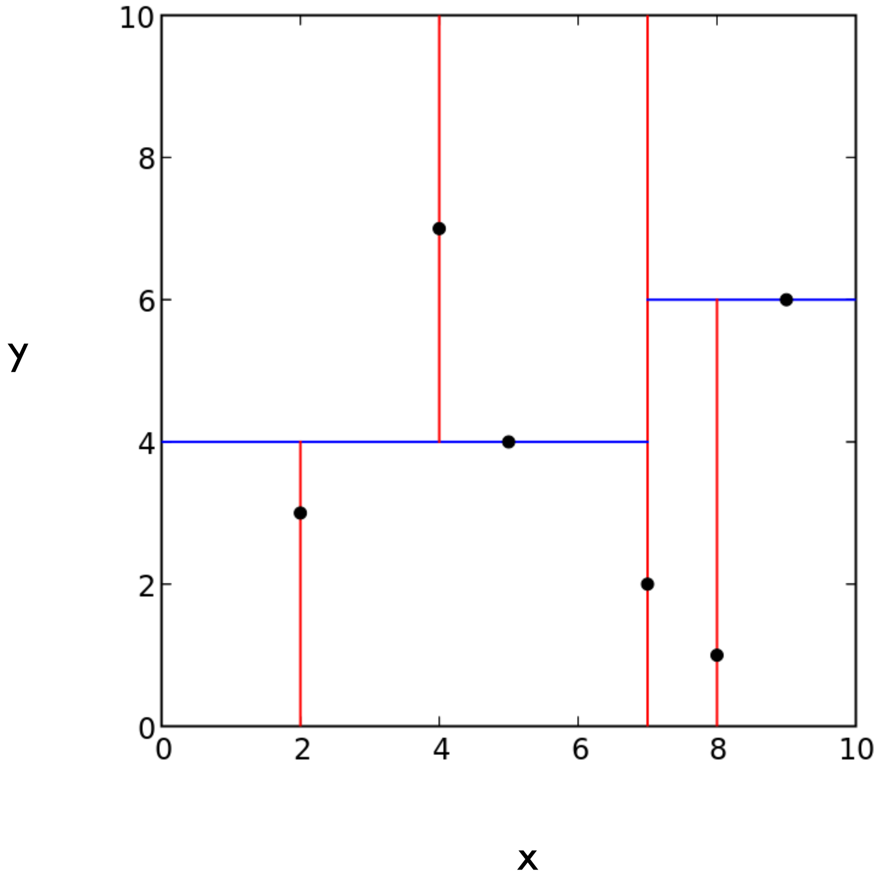
- Cycle dimensions as we walk down tree

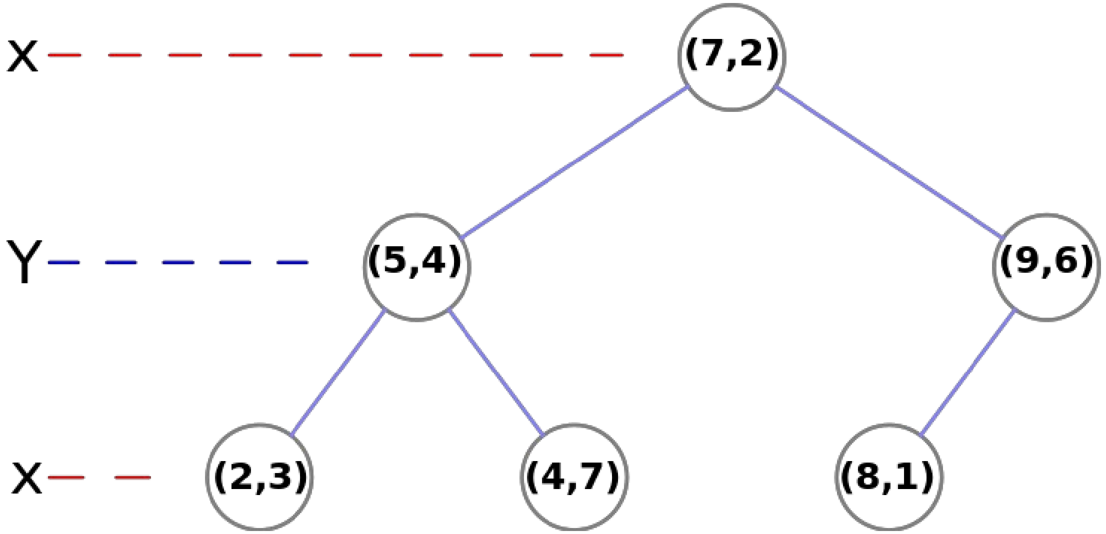
Each node contains a point $p = (x, y)$

At each level, only compare coordinate **in the cutting dimension.**



KD-TREE EXAMPLE





KD-TREE INSERTION

One-by-one insertion.

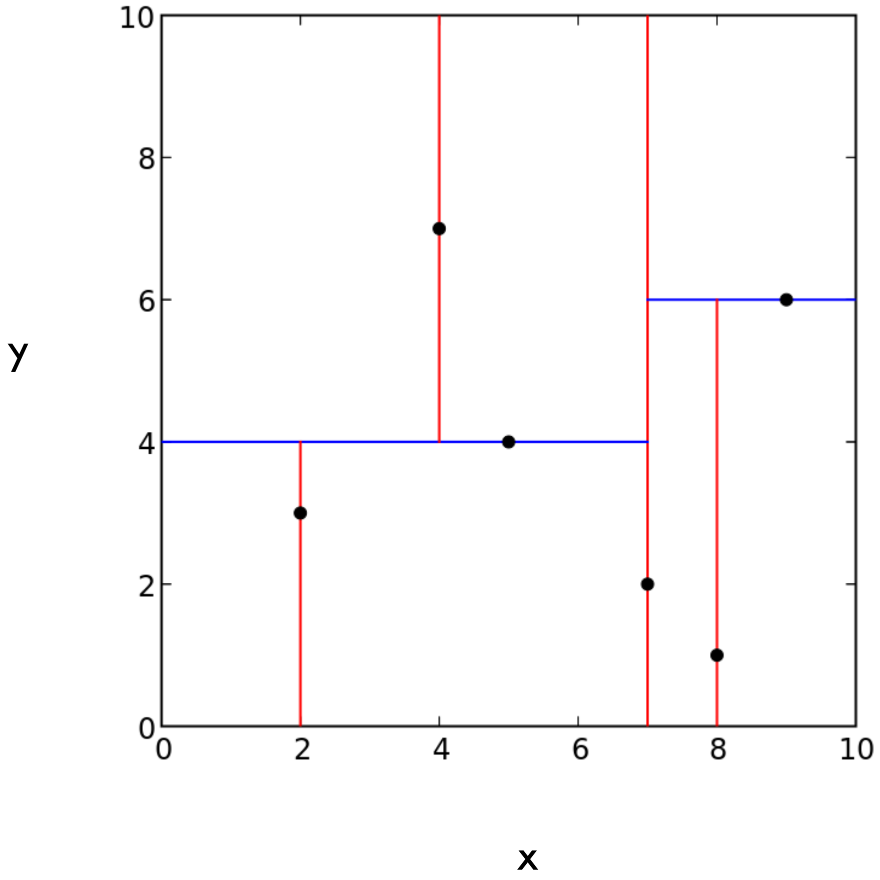
Idea: Go down the tree similar to BST insert.

Except: we check the appropriate dimension per level.

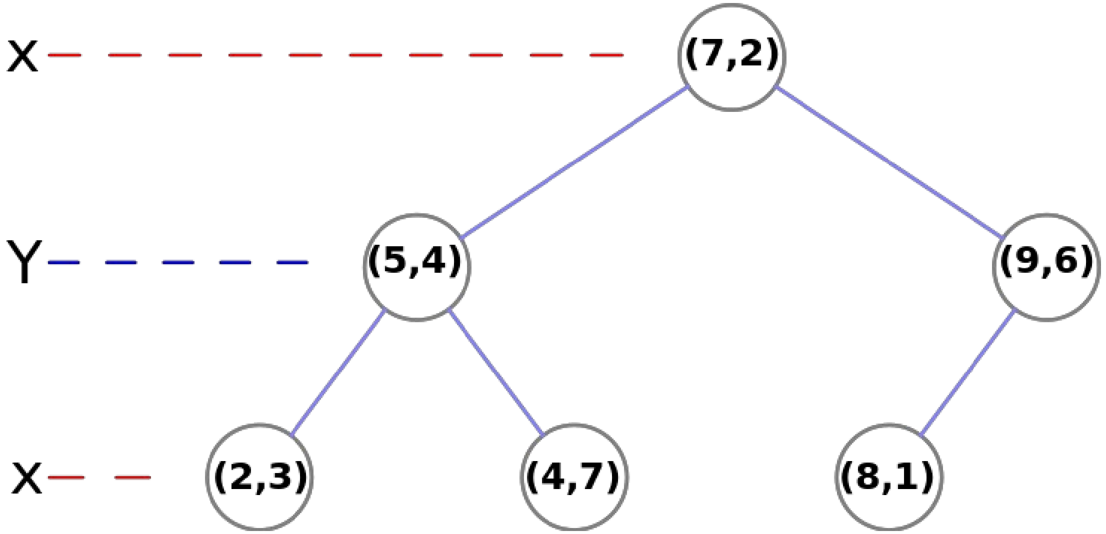
Point to insert Our Tree Current dimension

```
insert(Point x, KDNode t, int cd) {
    if t == null
        t = new KDNode(x)
    else if (x == t.data)
        // error! duplicate
    else if (x[cd] < t.data[cd])
        t.left = insert(x, t.left, (cd+1) % DIM)
    else
        t.right = insert(x, t.right, (cd+1) % DIM)
    return t
}
```

KD-TREE EXAMPLE

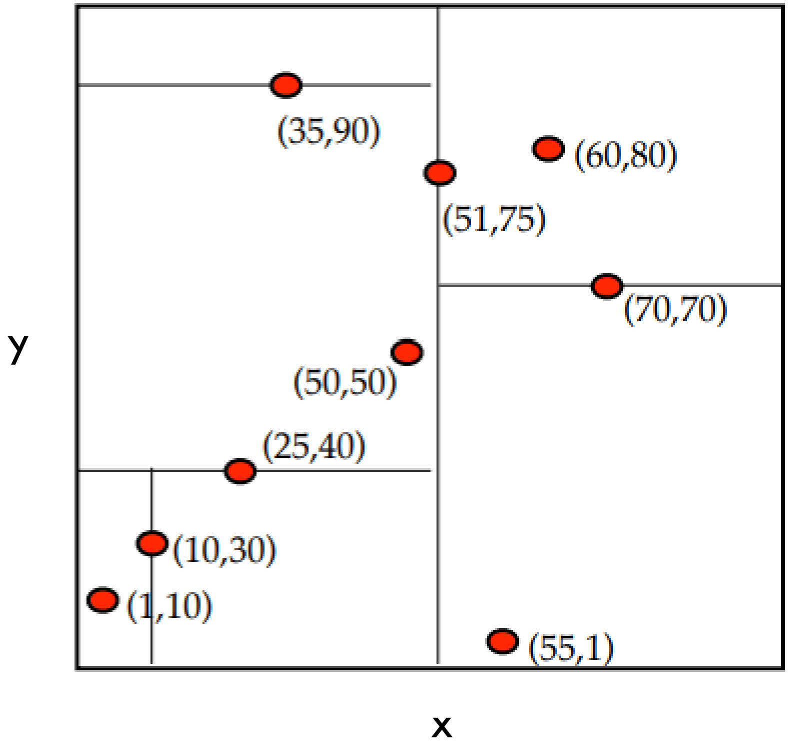


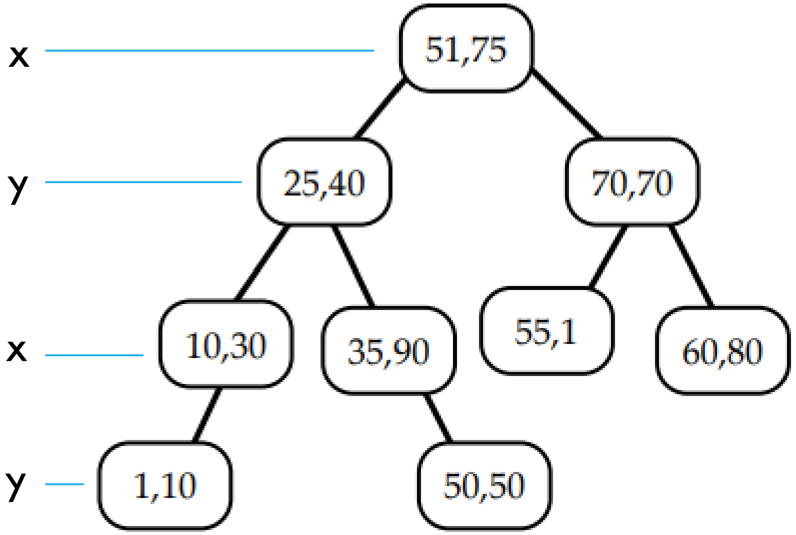
Try inserting (8,8)



NEAREST NEIGHBOR QUERY

Search for a *new* query point $q = (a, b)$





Idea: Search the tree like a BST and return the point in the final leaf node.

Does this idea work?

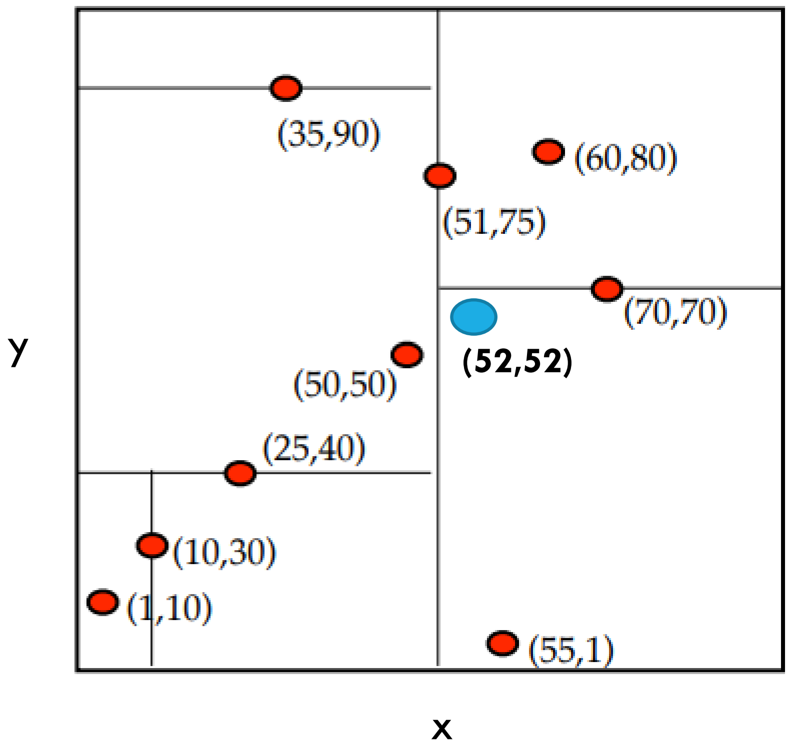
A. Yes! Betul!

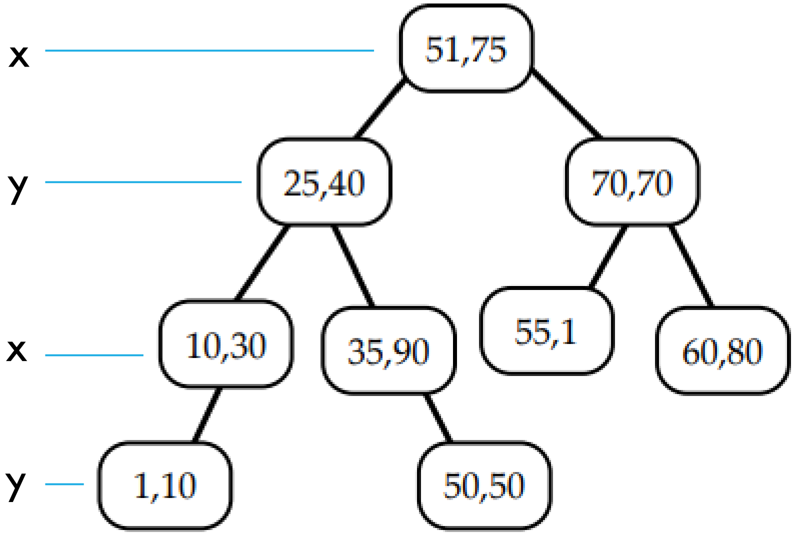
B. No! Salah!

https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf

NEAREST NEIGHBOR QUERY

Try: NearestNeighbor(52,52)





Idea: Search the tree like a BST and return the point in the final leaf node.

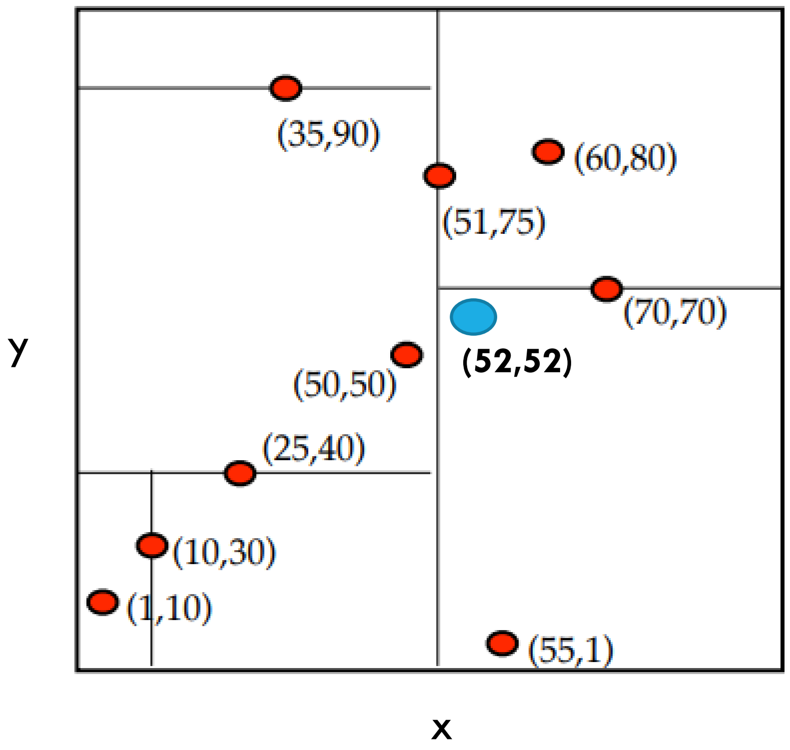
Does this idea work?

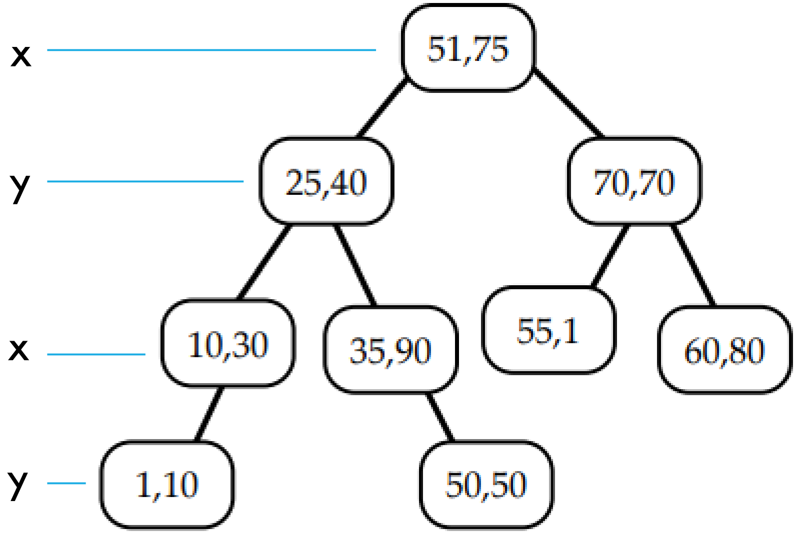
A. Yes! Betul!

B. No! Salah!

https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf

NEAREST NEIGHBOR QUERY





Idea 2: Search the tree like a BST and keep track of the closest point along the way. Return the point with the closest distance.

Does idea 2 work?

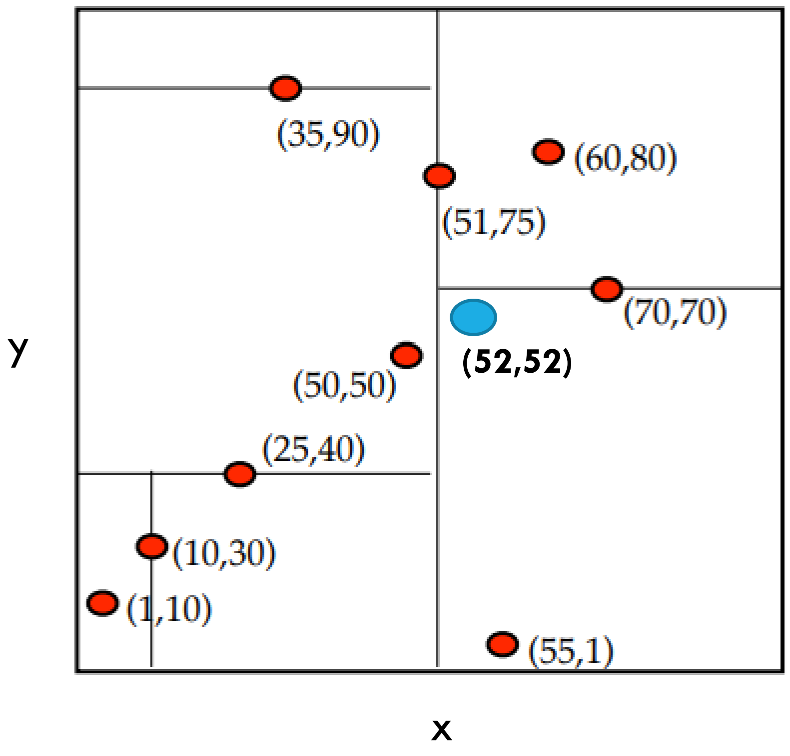
A. Yes! Betul!

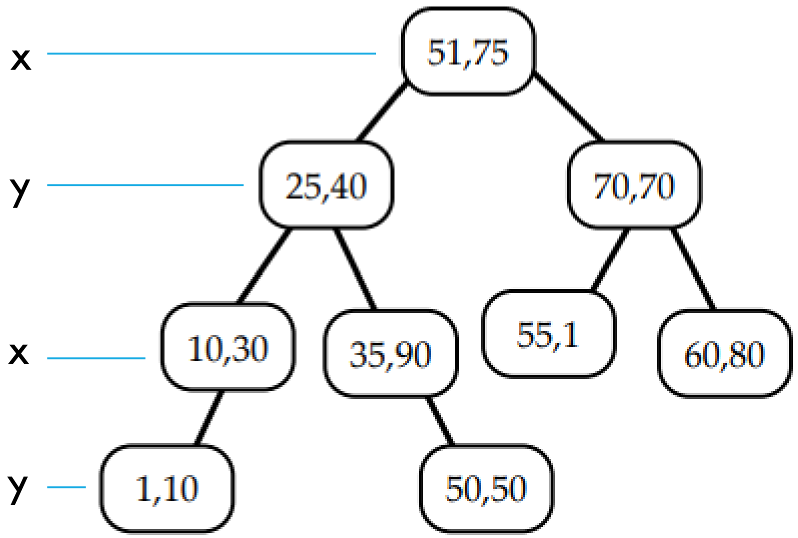
B. No! Salah!

https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf

NEAREST NEIGHBOR QUERY

The Nearest Point to q in space may be far from q in the tree





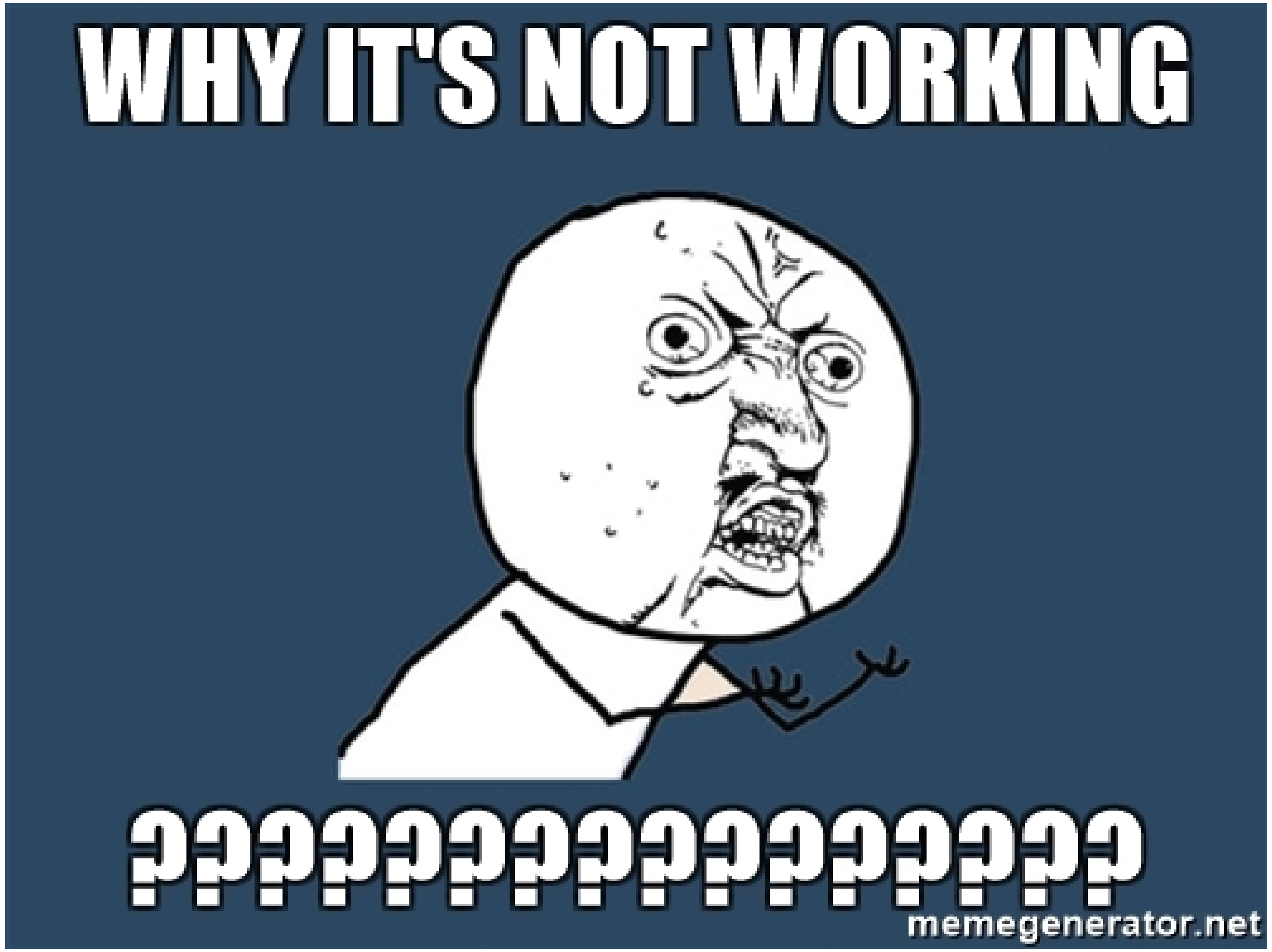
Idea 2: Search the tree like a BST and keep track of the closest point along the way. Return the point with the closest distance.

Does idea 2 work?

A. Yes! Betul!

B. No! Salah!

https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf

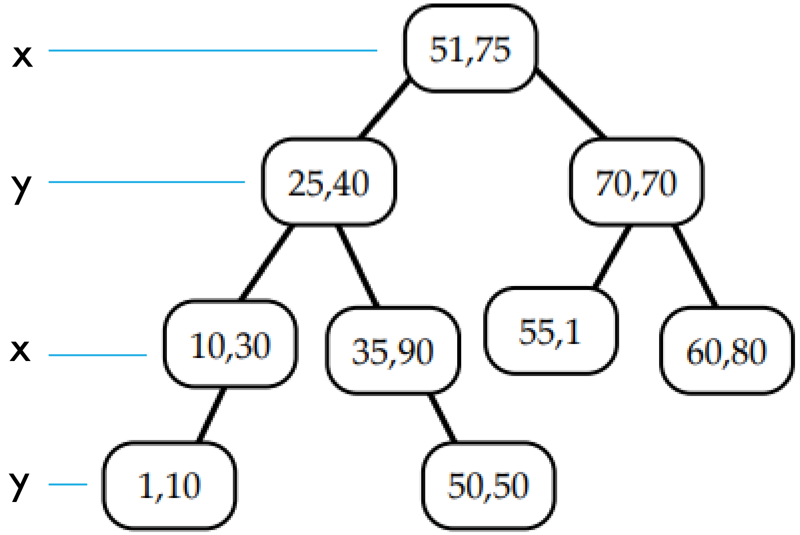

WHY IT'S NOT WORKING
????????????????????
memegenerator.net

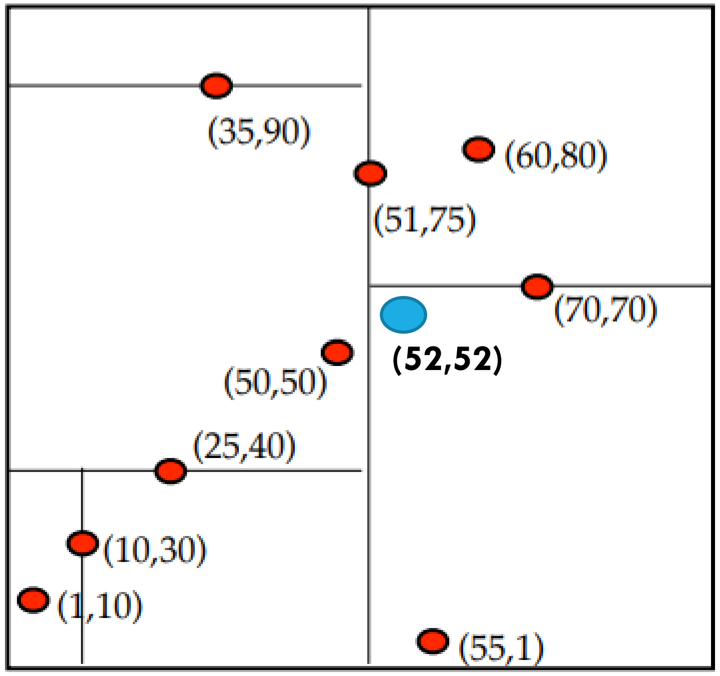
NEAREST NEIGHBOR QUERY

Idea: Traverse the whole tree but ***prune*** the search space.

How to prune?

- Keep variable of closest point C (candidate solution).
- Prune subtrees whose bounding boxes cannot contain any point closer than C.





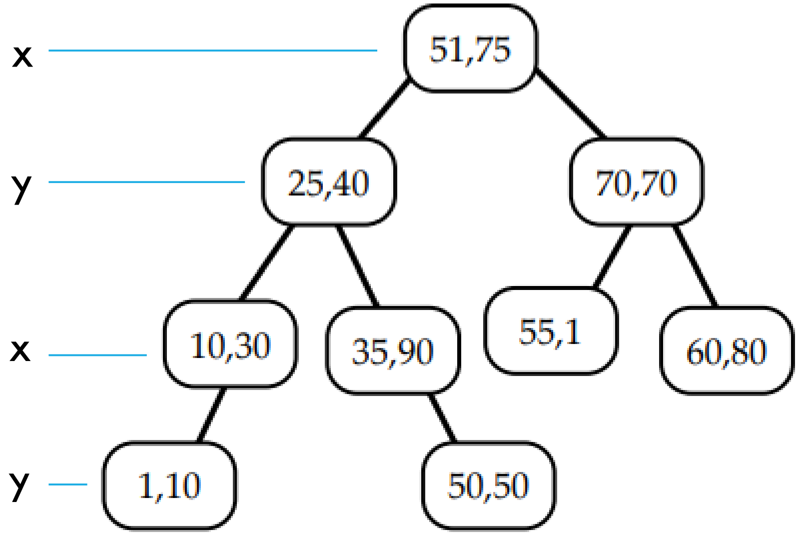
NEAREST NEIGHBOR QUERY

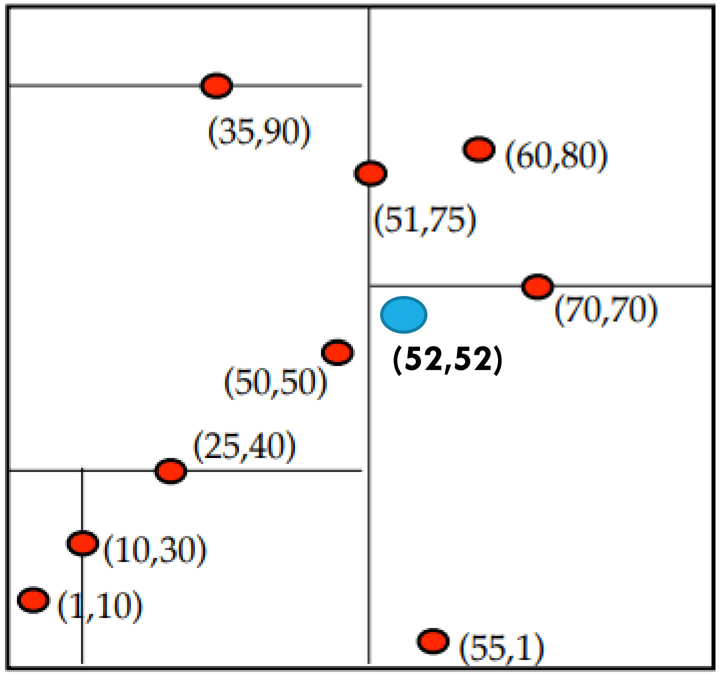
Phase 1:

Start with root node, move down the tree recursively similar to BST

> go left or right depending on whether the point is lesser than or greater than the current node **in the split dimension.**

At each step, check the node point and if the distance is closer, save node as C (the "current best“).





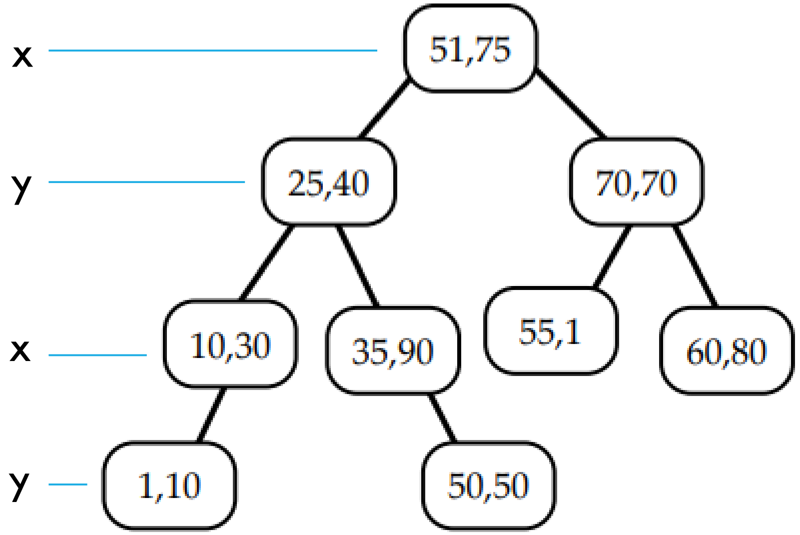
NEAREST NEIGHBOR QUERY

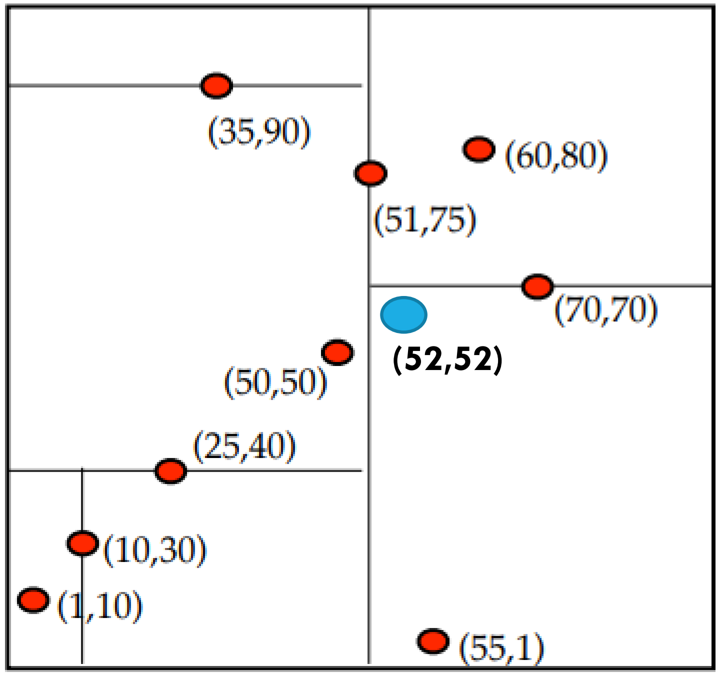
Phase 2:

Unwind the recursion. At each node:

- If current node has smaller distance, it becomes C
- Check if there are any points on the other side of the splitting dimension that could be closer than C
  - If **Yes**: go down the branch to the other side (recursively as before).
  - If **No**: can prune! Move up the tree.

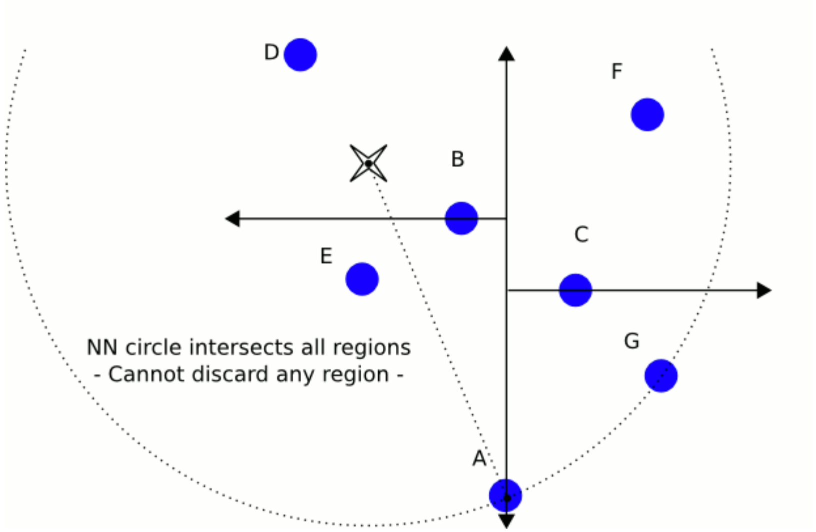
How to do this?

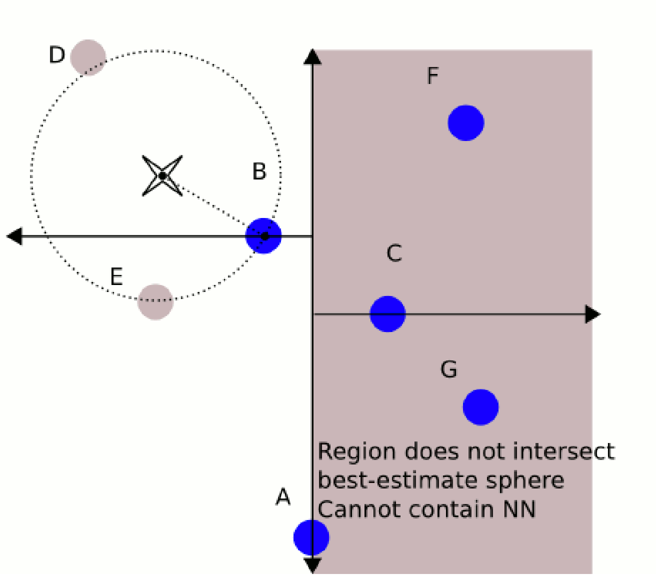




CLOSER POINTS ON THE OTHER SIDE?

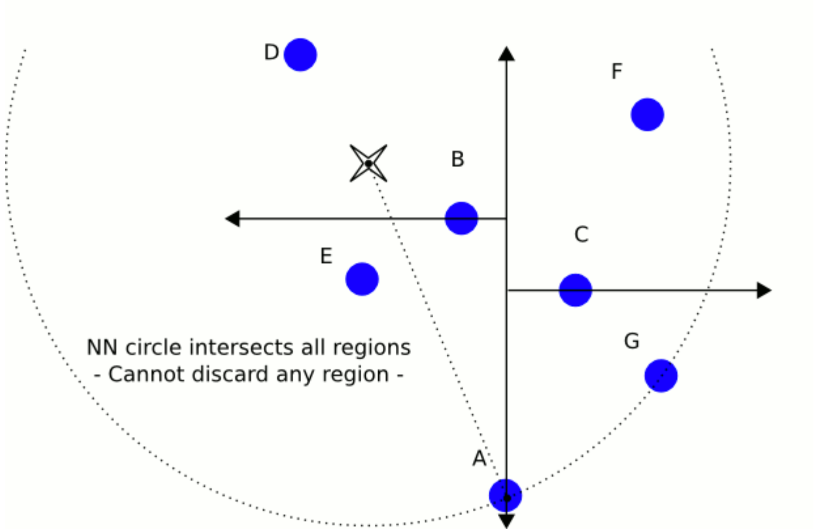
Idea: Check if (hyper)-sphere of radius r = dist(q,C) around point intersects with splitting hyperplane.

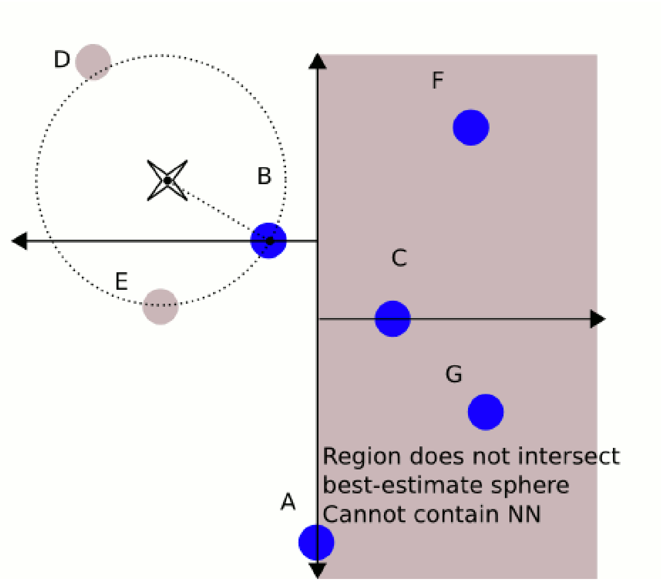




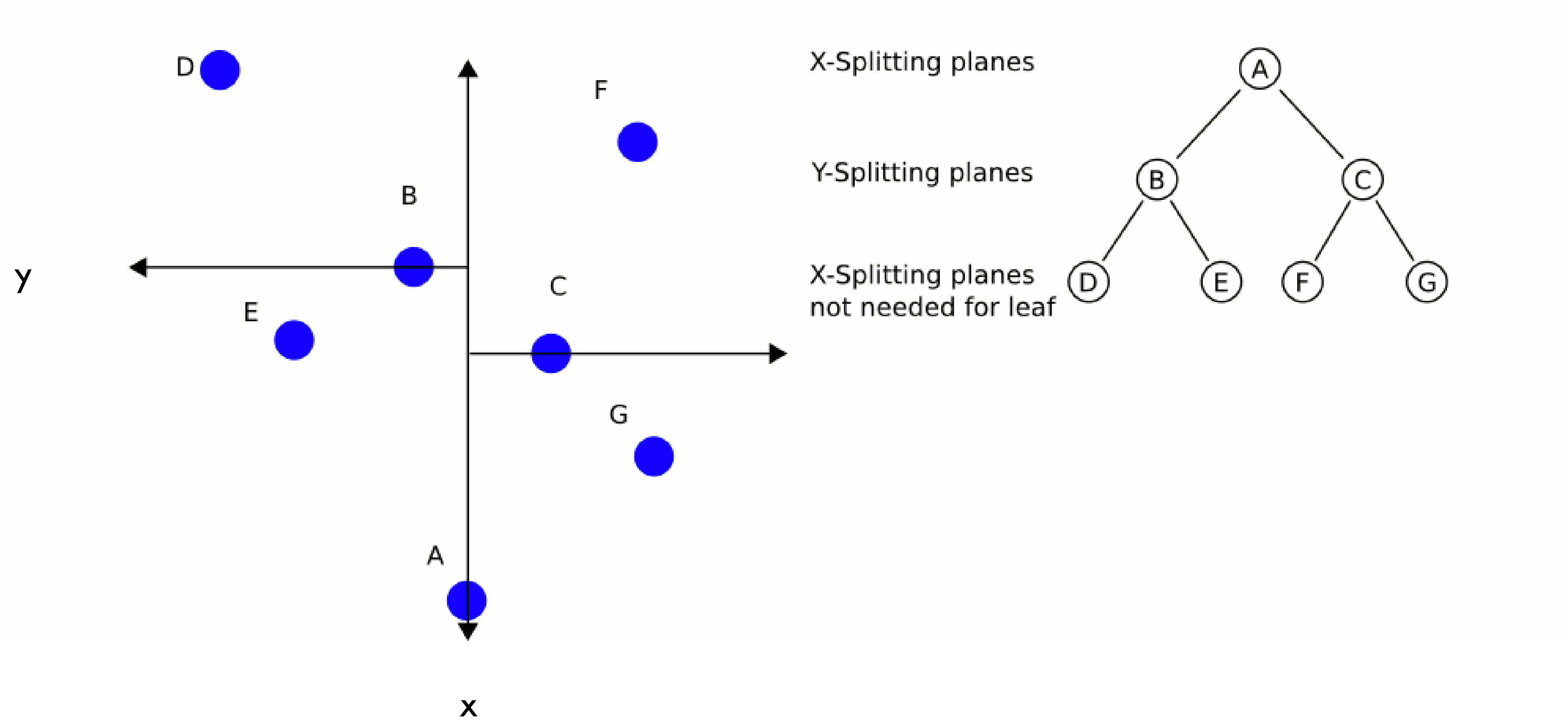
CLOSER POINTS ON THE OTHER SIDE?

Implementation: check that the distance between the **splitting coordinate** of the search point q and the current node is less than dist(q,C)

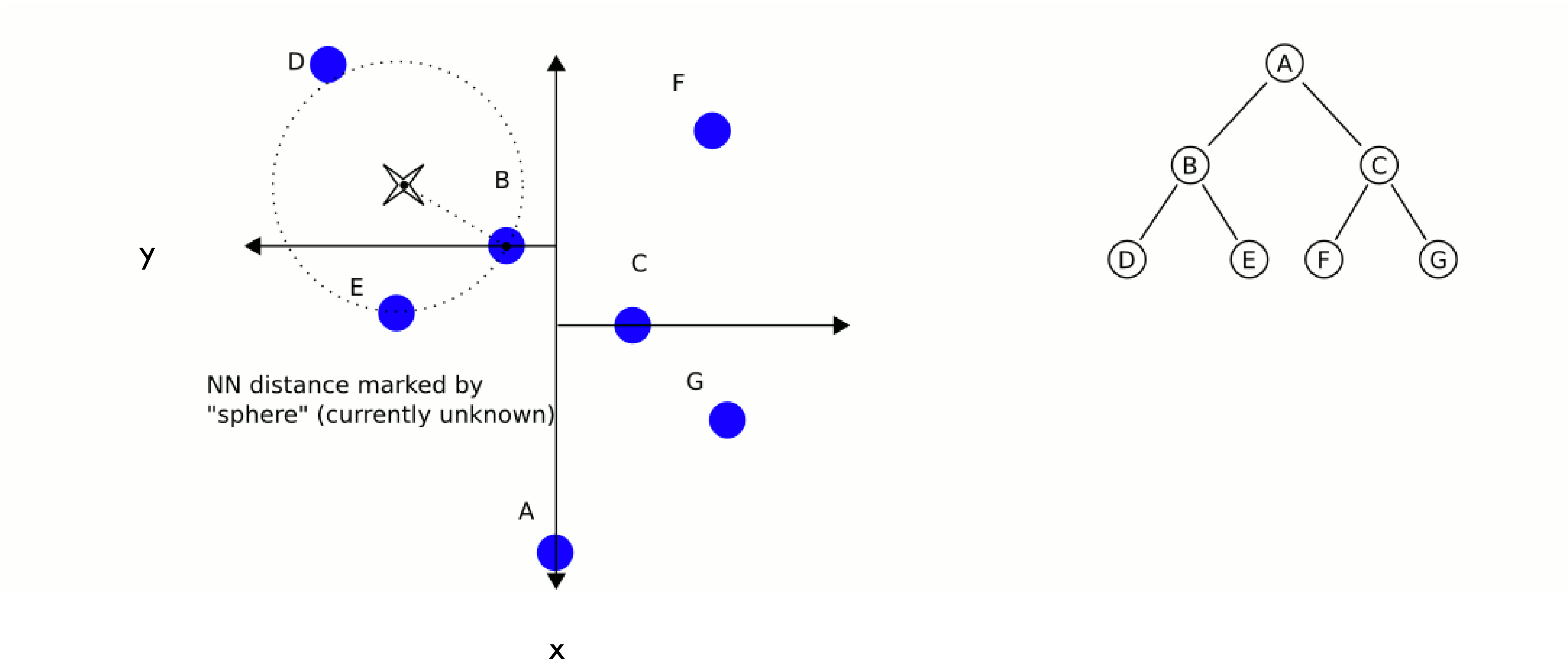




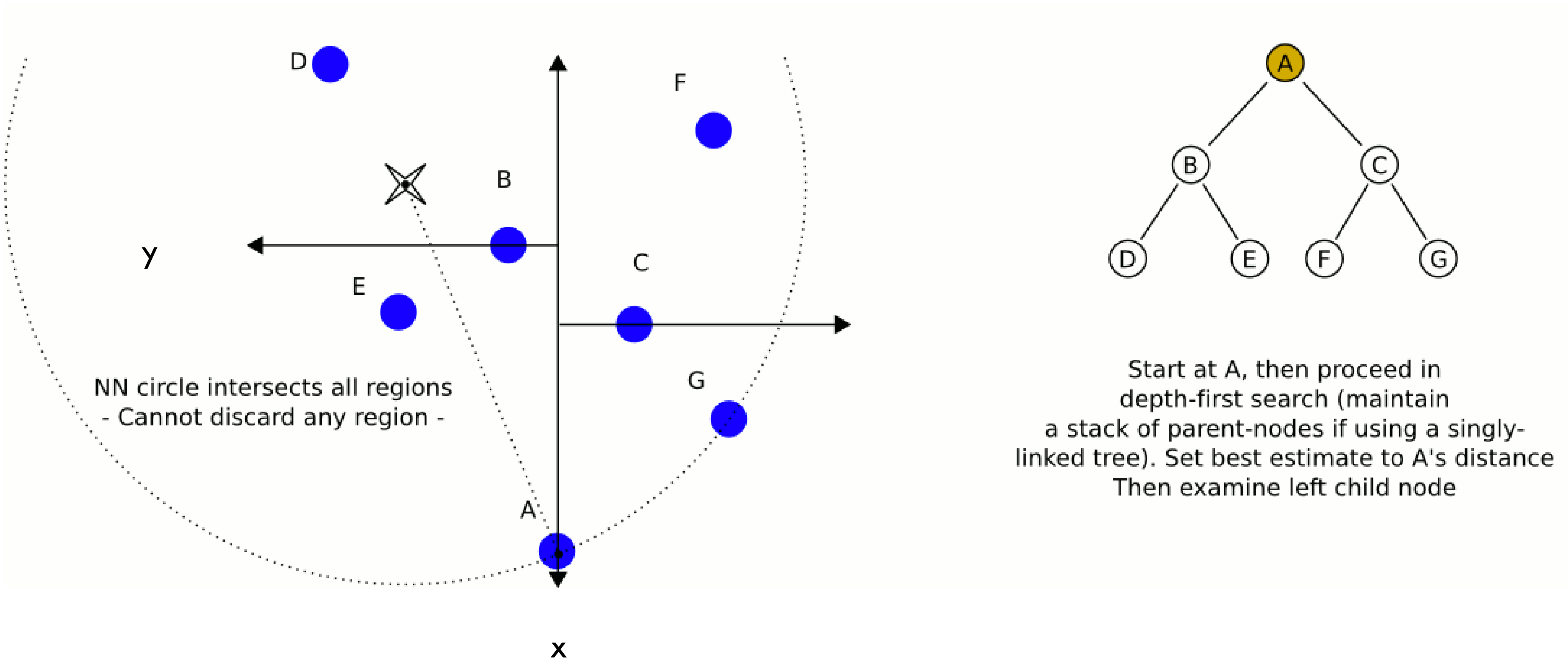
NEAREST NEIGHBOUR QUERY



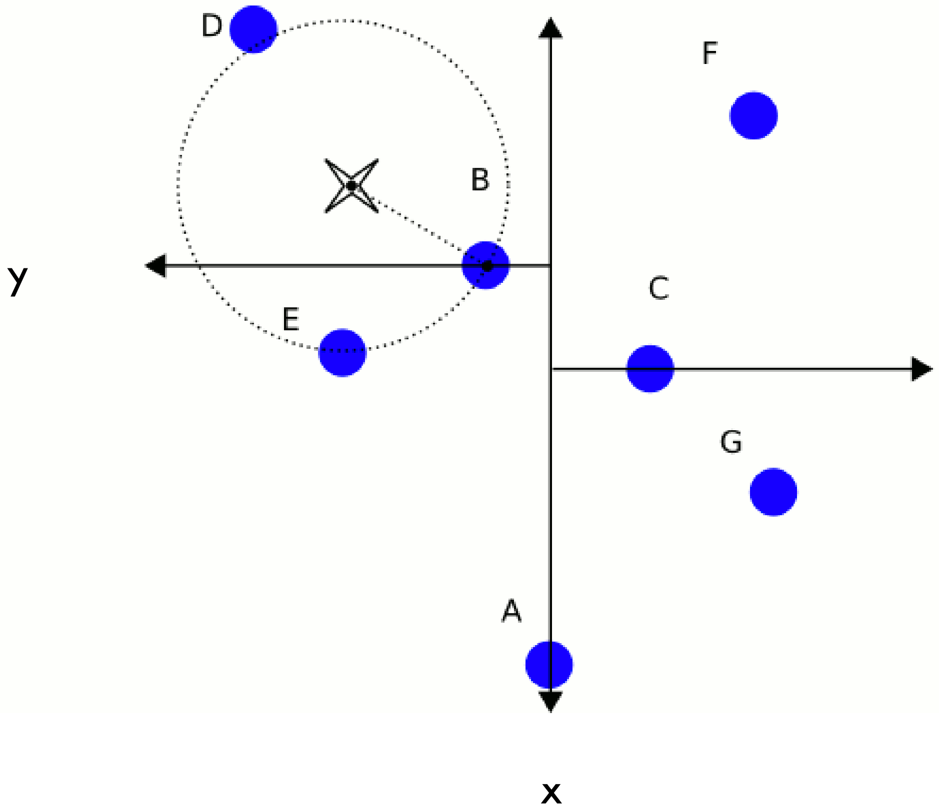
NEAREST NEIGHBOUR QUERY

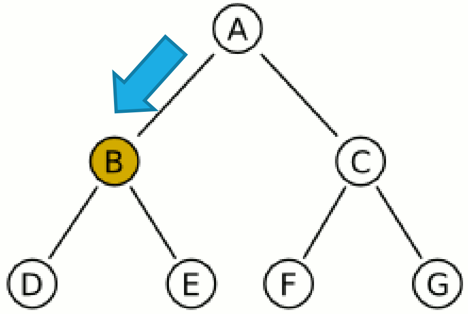


NEAREST NEIGHBOUR QUERY



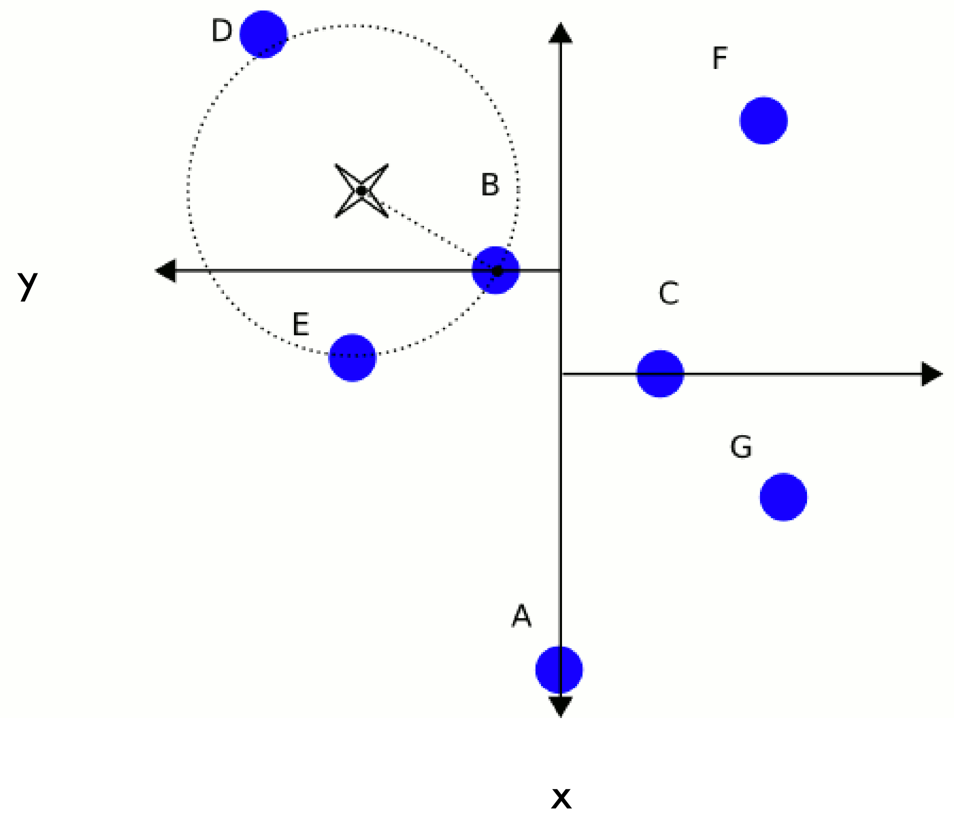
NEAREST NEIGHBOUR QUERY

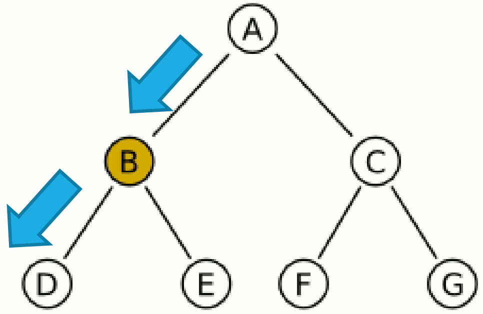




Calculate B's distance and
compare against best estimate
- It is smaller distance, so update
best estimate. Examine children (left then right)

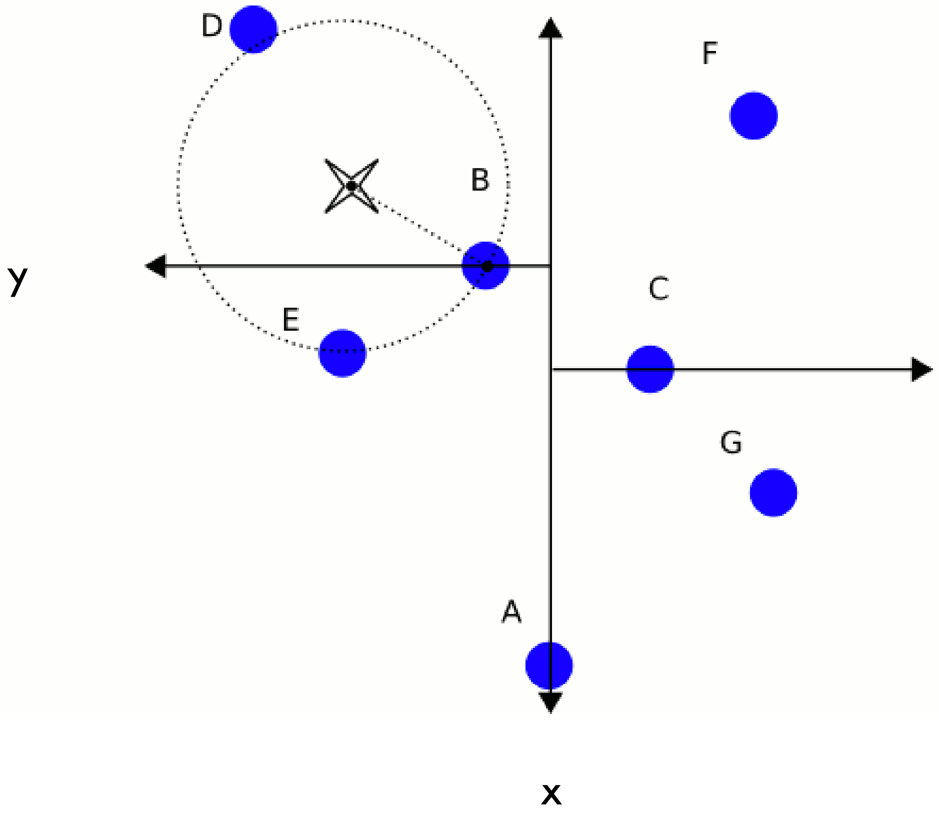
NEAREST NEIGHBOUR QUERY

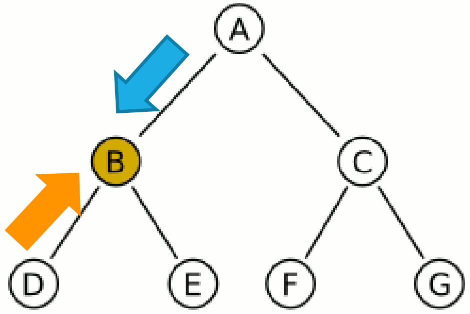




Calculate B's distance and
compare against best estimate
- It is smaller distance, so update
best estimate. Examine children (left then right)

NEAREST NEIGHBOUR QUERY



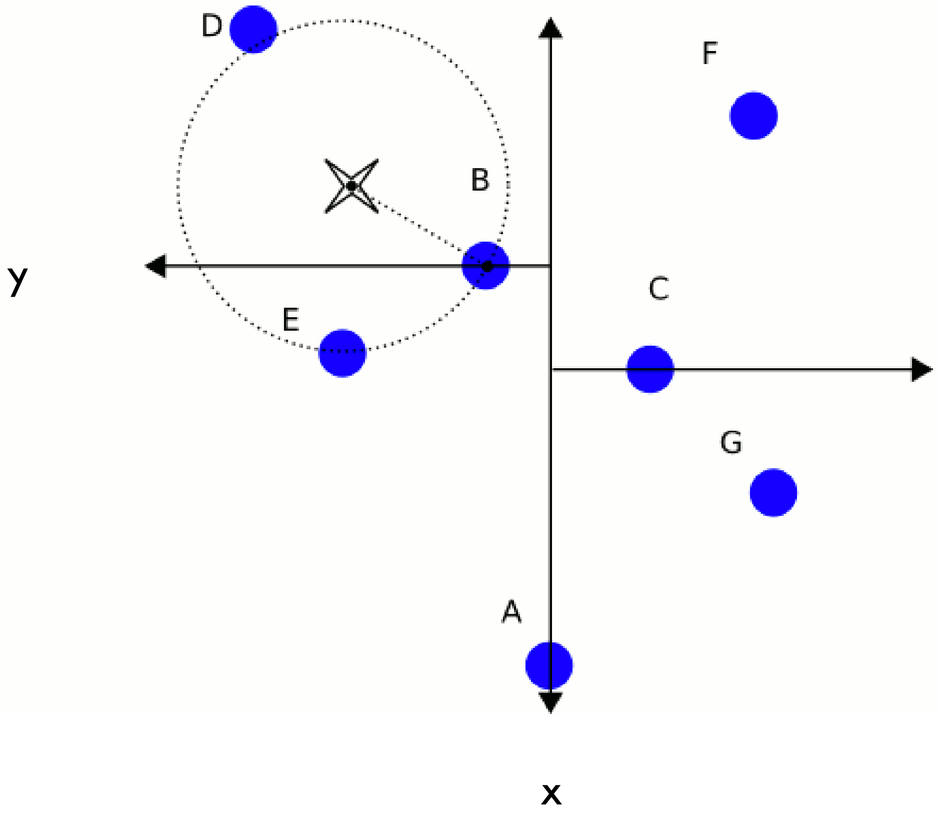


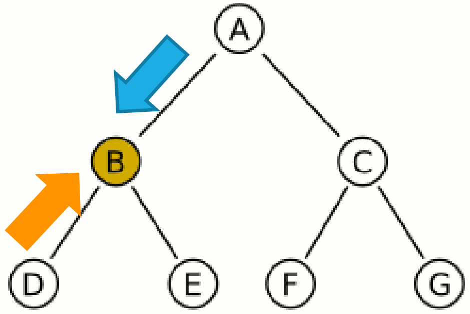
Calculate B's distance and
compare against best estimate
- It is smaller distance, so update
best estimate. Examine children (left then right)

NEAREST NEIGHBOUR QUERY

Is $|\ y_q - y_B\ | < dist(q, B)$?

Yes, have to check!



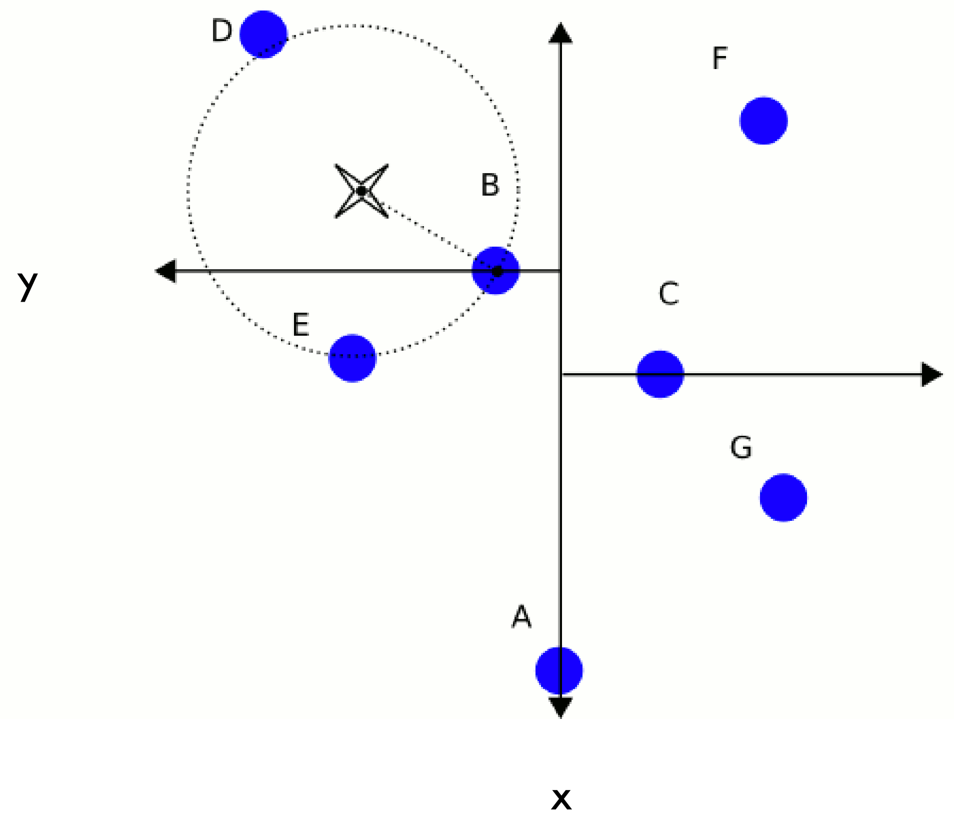


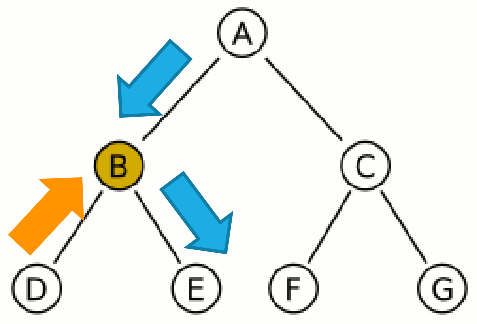
Calculate B's distance and
compare against best estimate
- It is smaller distance, so update
best estimate. Examine children (left then right)

NEAREST NEIGHBOUR QUERY

Is $| y_q - y_B | < dist(q, B)$?

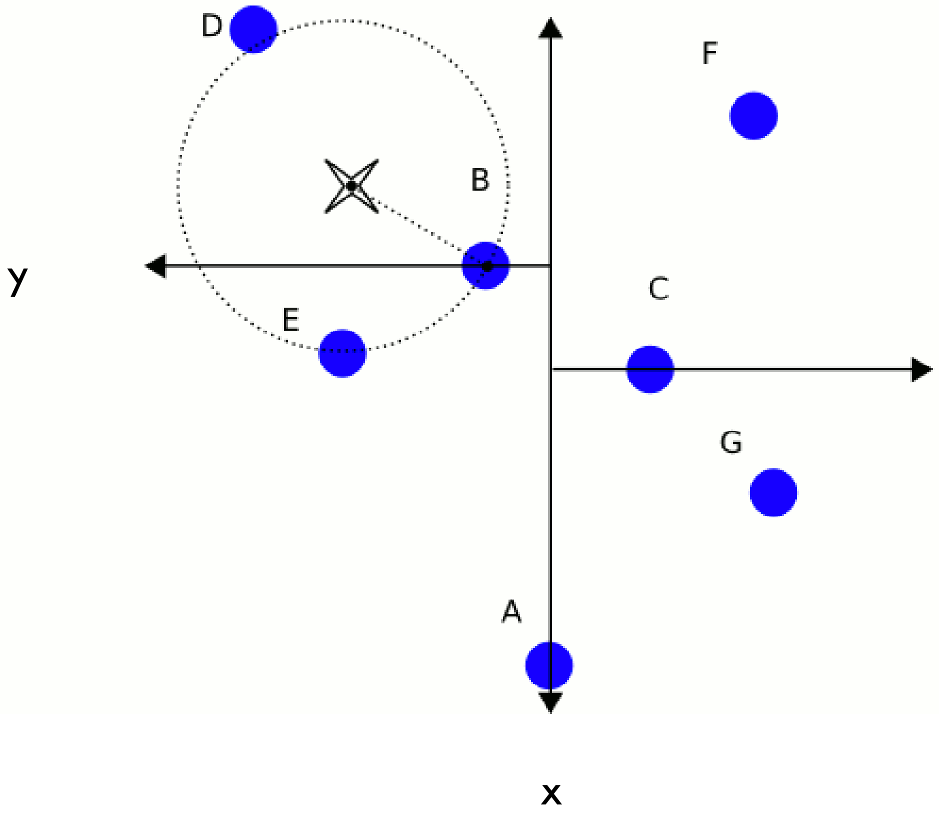
Yes, have to check!

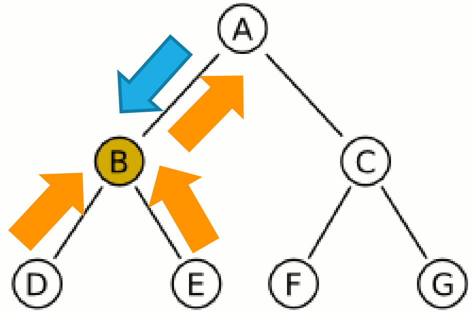




Calculate B's distance and
compare against best estimate
- It is smaller distance, so update
best estimate. Examine children (left then right)

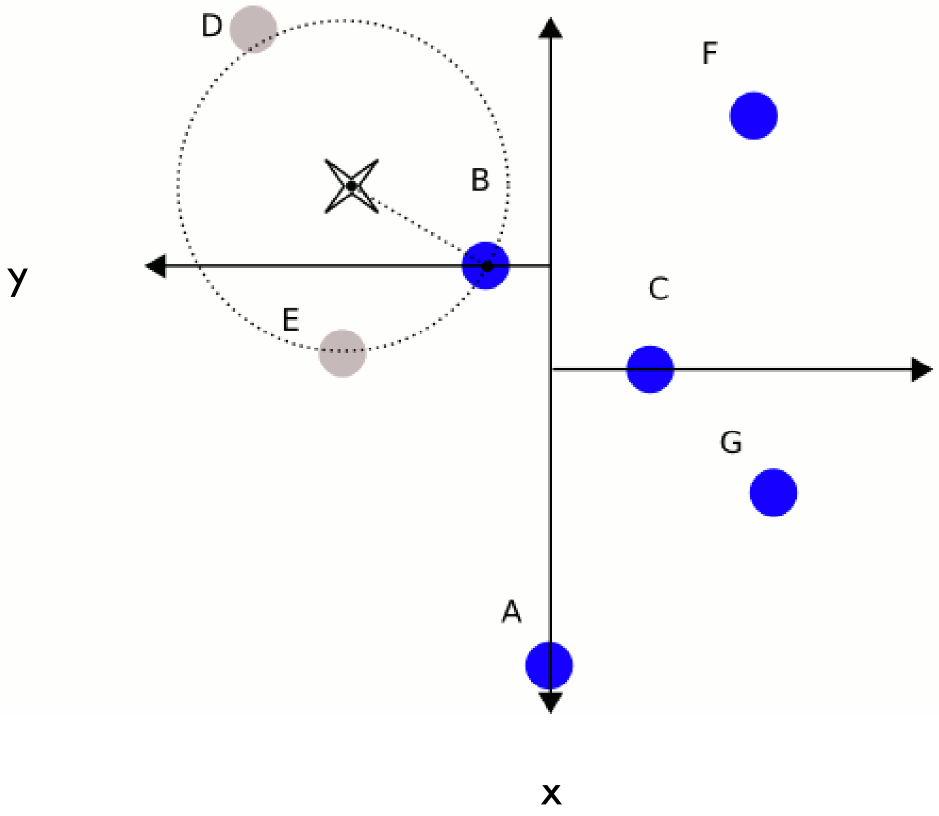
NEAREST NEIGHBOUR QUERY

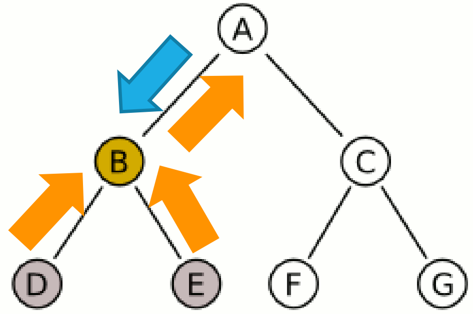




Calculate B's distance and
compare against best estimate
- It is smaller distance, so update
best estimate. Examine children (left then right)

NEAREST NEIGHBOUR QUERY



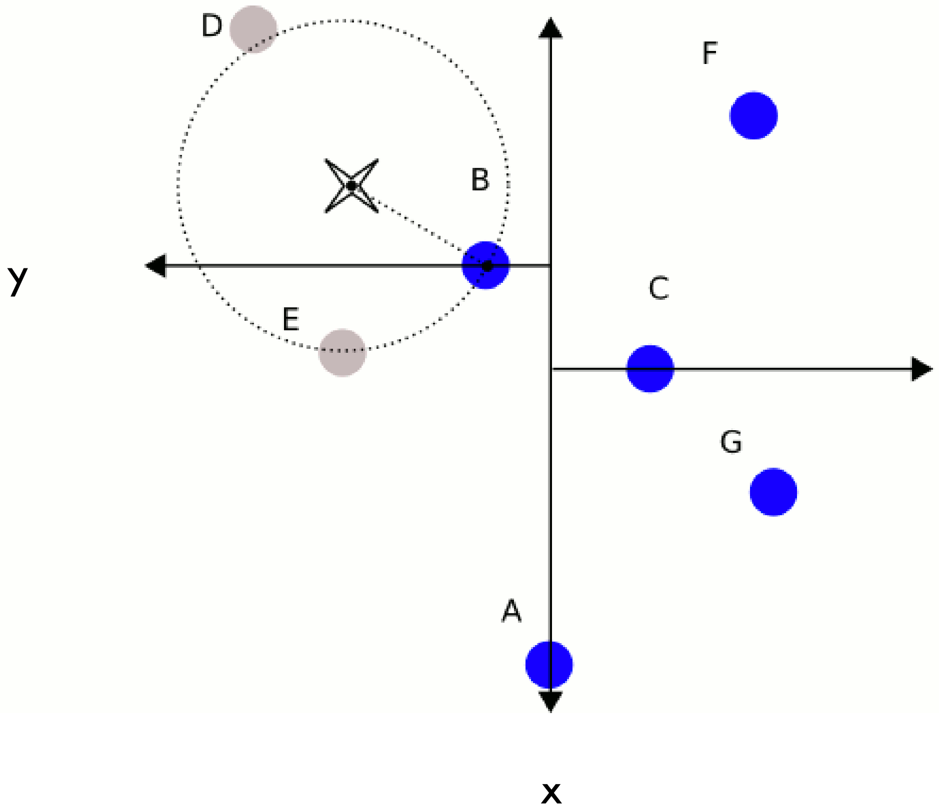


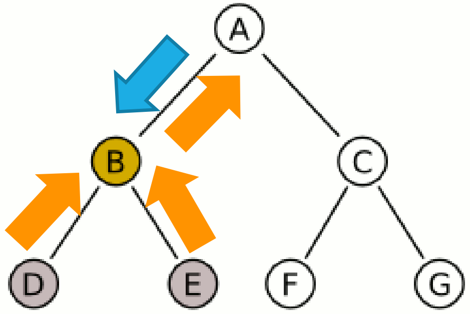
D & E Discarded as B
(already visited) is closer.
B is the best estimate for B's sub-branch
Proceed back to parent node

NEAREST NEIGHBOUR QUERY

Is $| x_q - x_A | < dist(q, B)$?

No! Can prune!



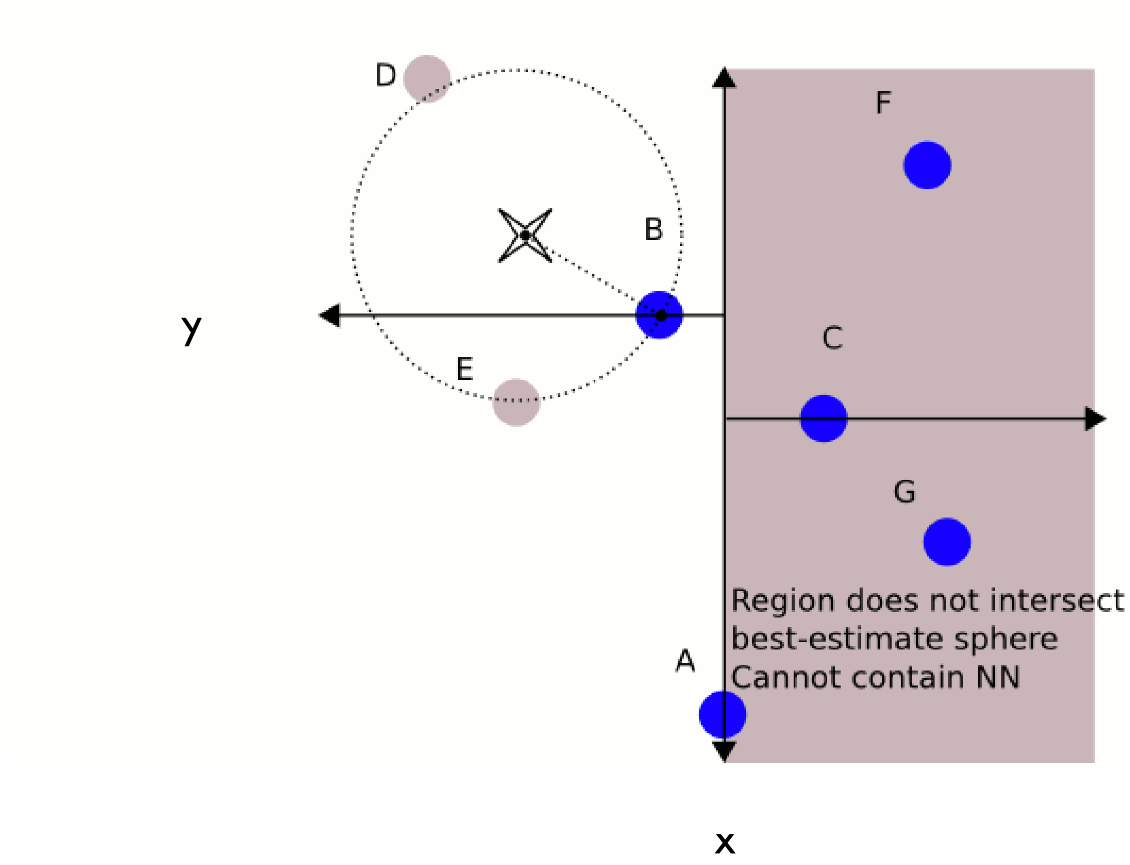


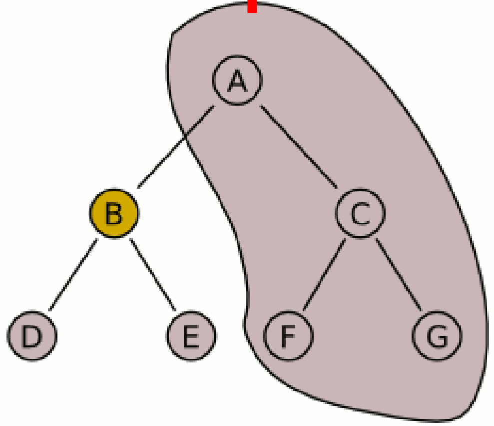
D & E Discarded as B
(already visited) is closer.
B is the best estimate for B's sub-branch
Proceed back to parent node

NEAREST NEIGHBOUR QUERY

Is $| x_q - x_A | < dist(q,B)$?

No! Can prune!



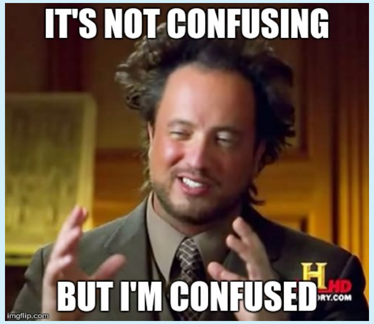


A's children have all been searched,
B is the best estimate for entire tree

NEAREST NEIGHBOUR QUERY: TIME COMPLEXITY

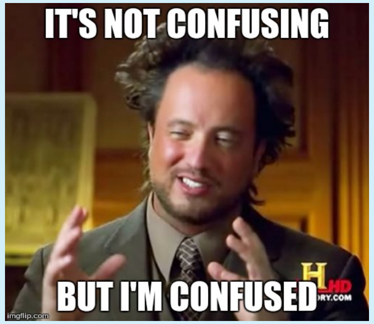
What is the worst case time complexity for a nearest neighbor query for fixed no. of dims?

A. O(1)
B. O(log n)
C. O(n)
D.



NEAREST NEIGHBOUR QUERY: TIME COMPLEXITY

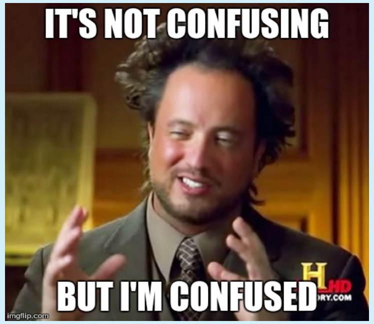
What is the worst case time complexity for a nearest neighbor query for fixed no. of dims?

A. O(1)
B. O(log n)
C. **O(n)**
D. 

NEAREST NEIGHBOUR QUERY: TIME COMPLEXITY

What is the **average** case time complexity for a nearest neighbor query for fixed no. of dims?

A. O(1)
B. O(log n)
C. O(n)
D.



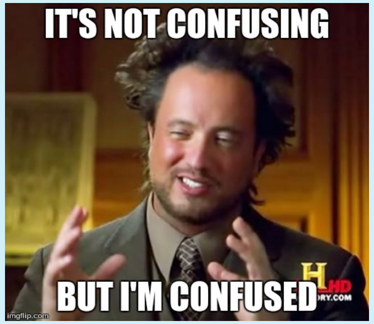
NEAREST NEIGHBOUR QUERY: TIME COMPLEXITY

What is the **average** case time complexity for a nearest neighbor query for fixed no. of dims?

A. O(1)

B. O(log n)

C. O(n)

D. 

Actually, $O(2^k + \log(n))$

KD-TREE CONSTRUCTION

Given a set of points $P = \{(x_i, y_i)\}$,

What's a simple way to construct a KD-Tree?

Insert the points one-by-one

What's the problem with this approach?

Tree might be imbalanced!

Try Insert: (1,10), (10,30), (51,75), (70,70), (72, 80)

KD-TREE CONSTRUCTION

Recursively (while cycling through axes):

- Pick median in the current splitting dimension
- Partition points into 2 groups (before and after median)
- Recursively call construct on both groups

- **Think about:**
  - How fast can we find the median? Using which algorithm?
  - Does the method lead to a balanced tree?

KD-TREE VARIANTS

kd-Tree works well in low-medium dimensions (not great for high dimensional data)

Many variants of kd-trees.

- Can store points at the leaf nodes only.
- Need not cycle dimensions or use median.

Can choose at each node:

- Which dimension to split on?
- What value to split on?
- How many points to store at the leaf nodes?

MANY APPLICATIONS

Nearest location searches

Ray-Tracing

2d range search

N-body simulation

Collision detection

Etc.

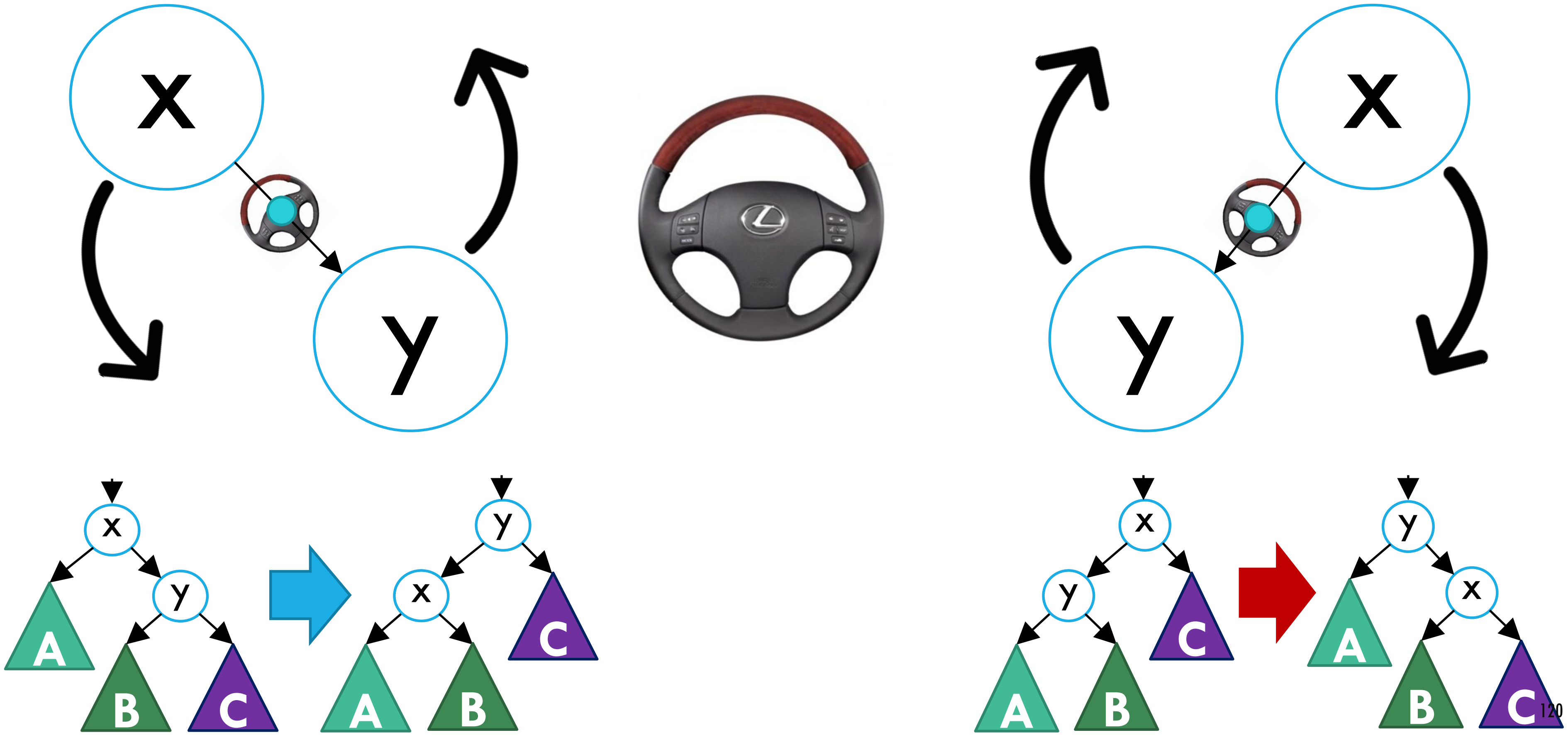
LEARNING OUTCOMES

By the end of this session, students should be able to:

- Derive how **height-balanced trees** ensure O(log n) operations
- Describe **how balance is maintained in an AVL tree.**
- **Explain rotations** and **how they are used to correct height imbalances.**

HOW I REMEMBER IT

left / anti-clockwise | **right / clockwise**

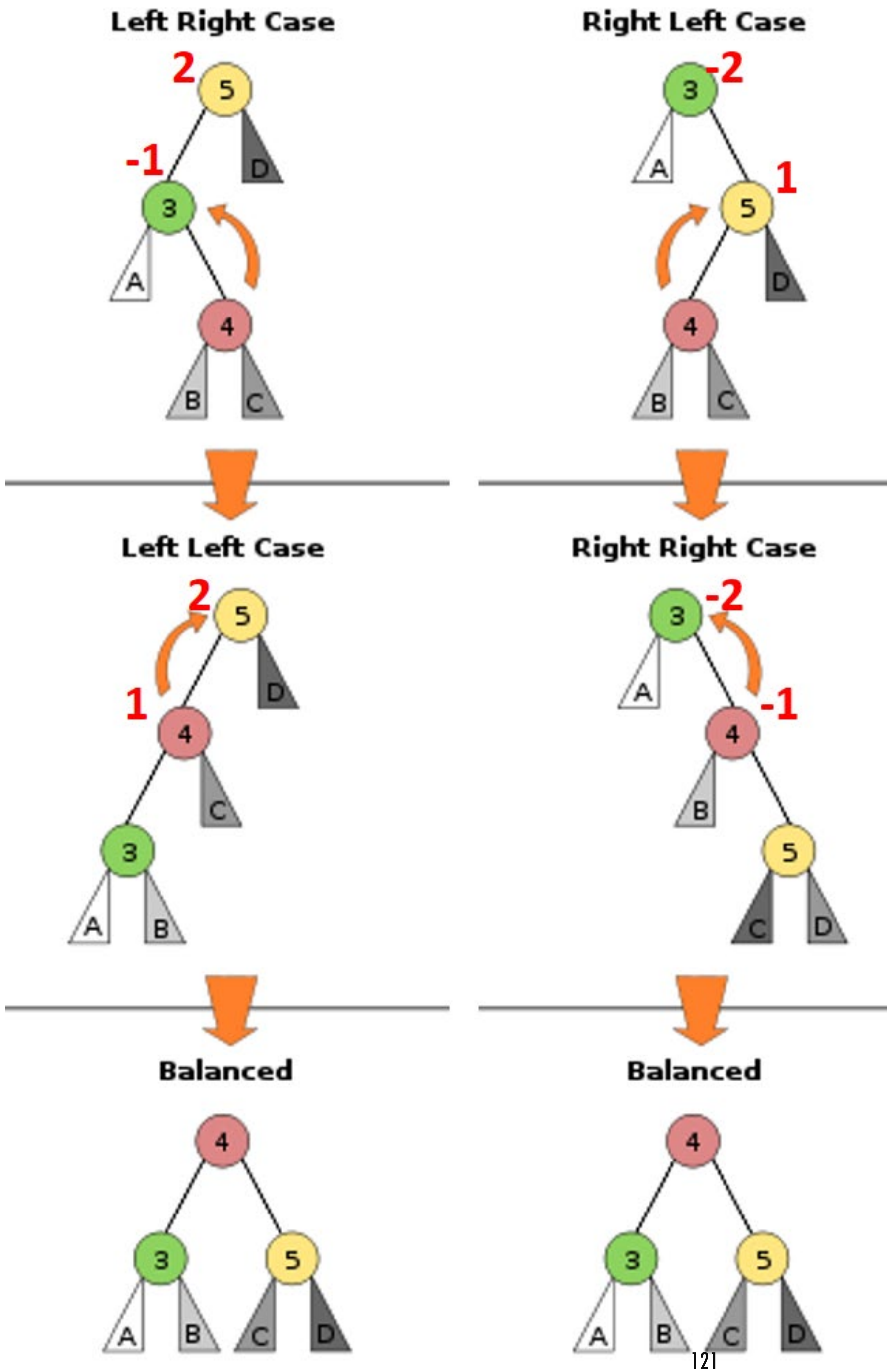


ALGORITHM TO FIX IMBALANCES

For both insertion and deletions:

- walk up the tree (to the root) from the inserted/deleted node & update balance factors.
- if we find a height-balance violation, fix it via a rotation:

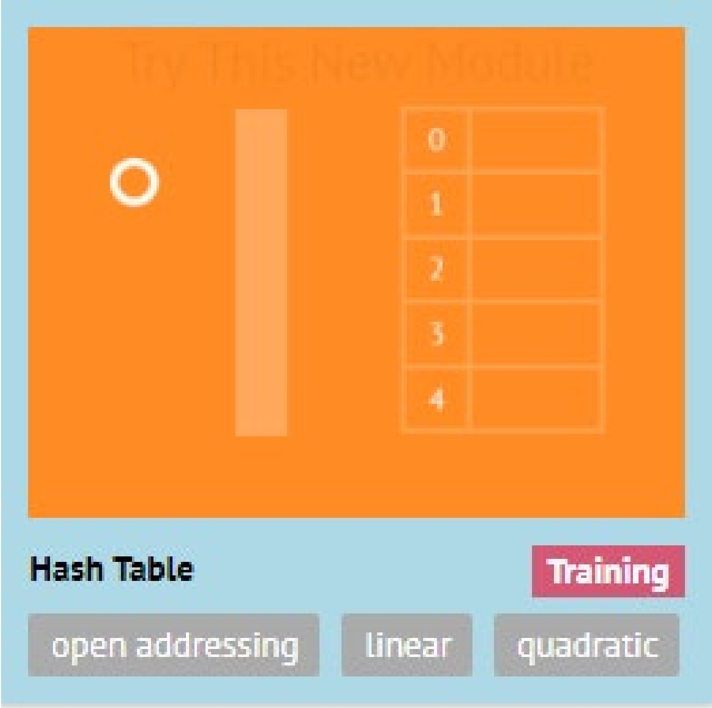
```
if tree is right heavy
      if tree's right subtree is left heavy
            left-right rotate
      else
            left rotate
else if tree is left heavy
      if tree's left subtree is right heavy
            right-left rotate
      else
            right rotate
```



BEFORE LECTURE NEXT WEEK

Go to Visualgo.net and do the Hash Table Module:

- https://visualgo.net/en/hashtable
- Review: 1-10 (Separate Chaining)
- Optional: 11



QUESTIONS?