CS2040S
Data Structures and Algorithms

Augmented Trees!

This Week

On the importance of being balanced

- Height-balanced binary search trees
- AVL trees
- Rotations

Tries

- How to handle text?

Data structure design

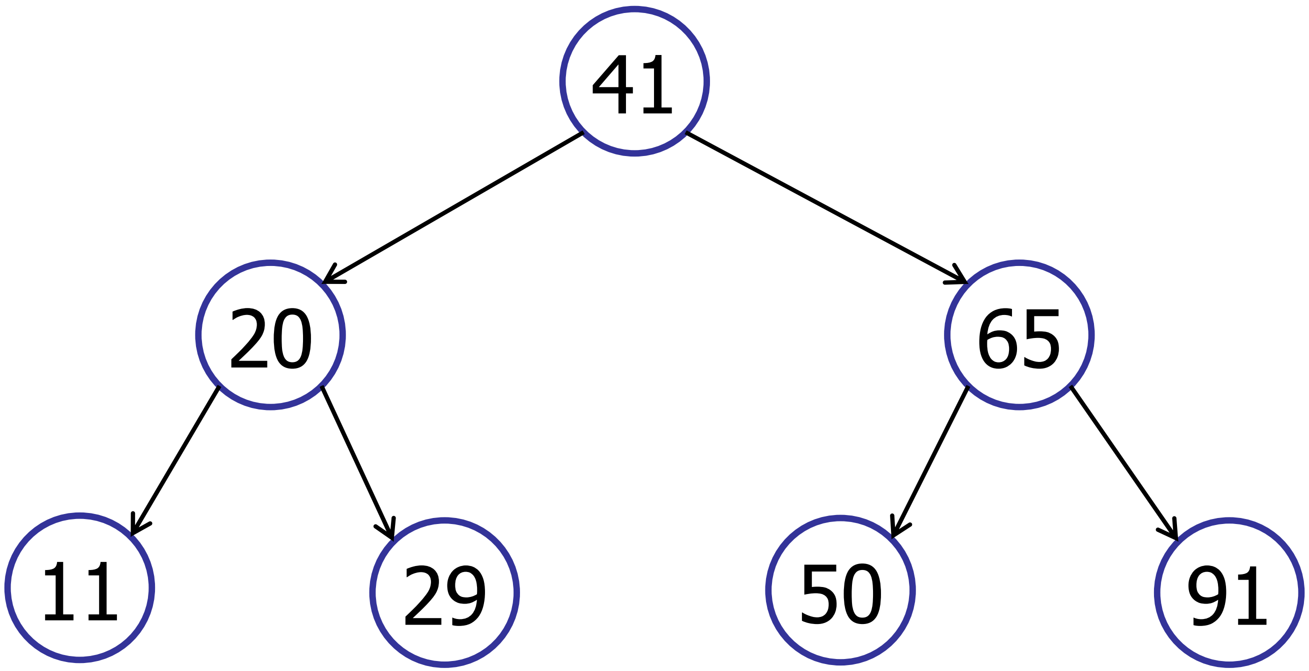
- How to build new structures on existing ideas?

Recap: Dictionary Interface

A collection of (key, value) pairs:

interface	**IDictionary**	
void	insert(Key k, Value v)	*insert (k,v) into table*
Value	search(Key k)	*get value paired with k*
Key	successor(Key k)	*find next key > k*
Key	predecessor(Key k)	*find next key < k*
void	delete(Key k)	*remove key k (and value)*
boolean	contains(Key k)	*is there a value for k?*
int	size()	*number of (k,v) pairs*

Recap: Binary Search Trees



- Two children: v.left, v.right
- Key: v.key
- **BST Property**: all in left sub-tree < key < all in right sub-right

The Importance of Being Balanced

How to get a balanced tree:

- Define a good property of a tree.
- Show that if the good property holds, then the tree is balanced.
- After every insert/delete, make sure the good property still holds. If not, fix it.

Invariant

AVL Trees [Adelson-Velskii & Landis 1962]

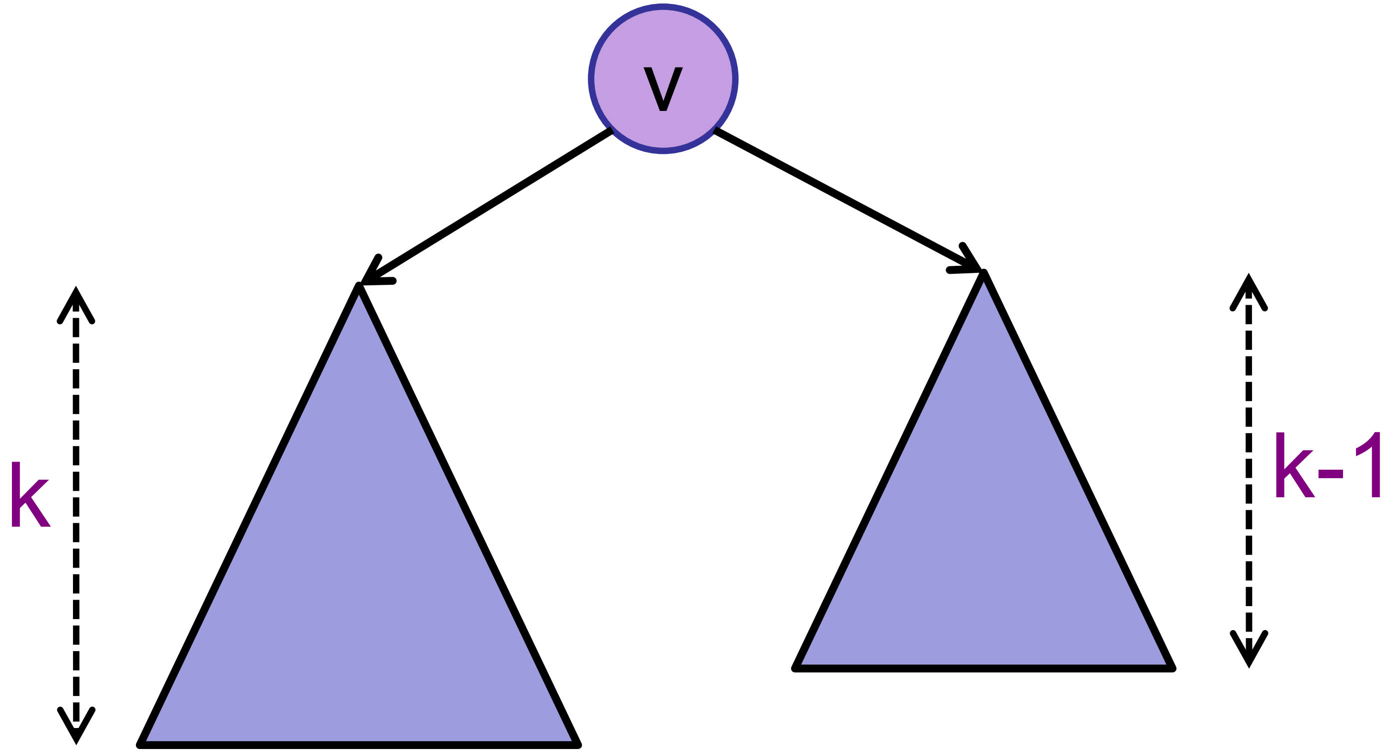
AVL Trees [Adelson-Velskii & Landis 1962]

Key definition

Step 1: Define Invariant

- A node v is **height-balanced** if:

$$|v.left.height - v.right.height| \leq 1$$



Height-Balanced Trees

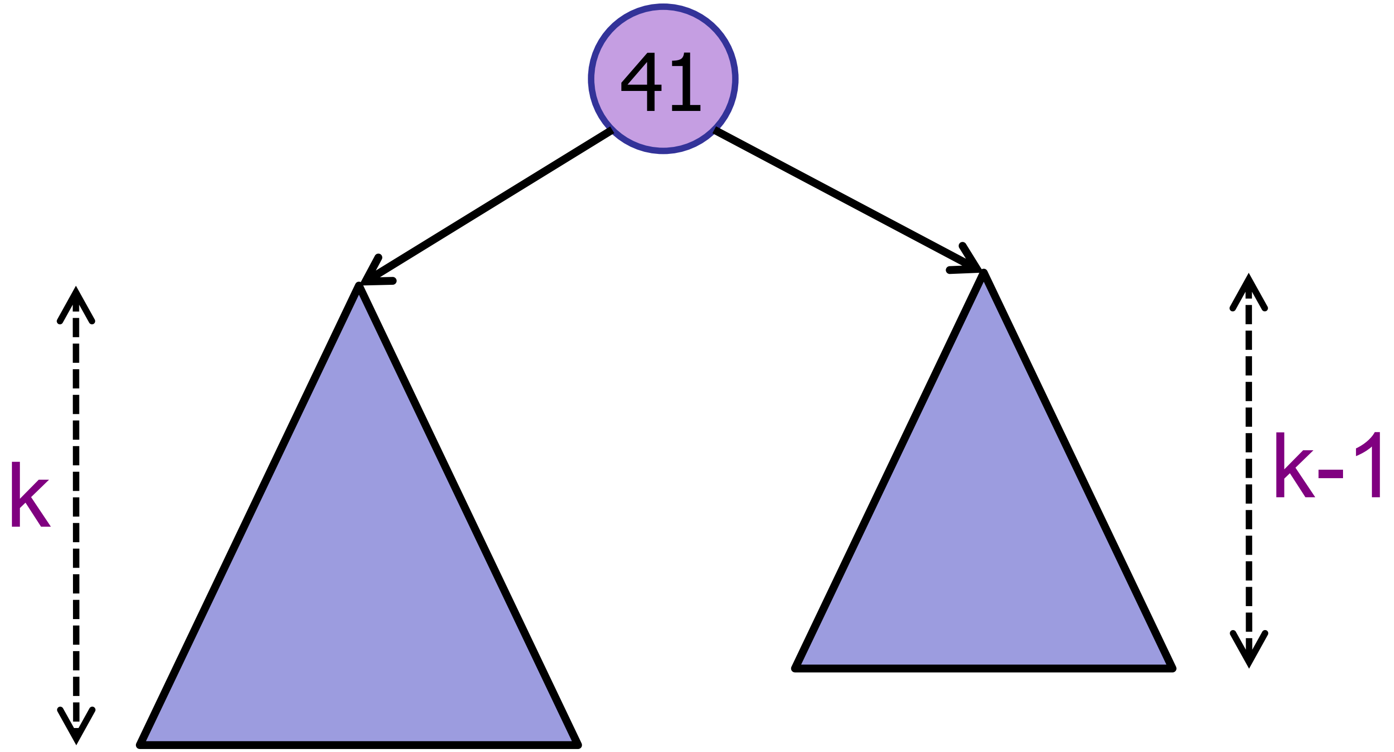
Theorem:

A height-balanced tree with n nodes has **at most** height $h < 2\log(n)$.

➔ A height-balanced tree is balanced.

AVL Trees [Adelson-Velskii & Landis 1962]

Step 2: Show how to maintain height-balance



Insert in AVL Tree

Summary:

- Insert key in BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance and return.

Key observation:

- Only need to fix *lowest* out-of-balance node.
- Only need at most two rotations to fix.

Delete in AVL Tree

Summary:

- Delete key from BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance.
  - Continue to root.

Key observation:

- It is *not* sufficient to only fix lowest out-of-balance node in tree.

Dynamic Data Structures

1. Maintain a set of items

2. Modify the set of items

3. Answer queries.

Big picture idea:

Trees are a good way to store, summarize, and search dynamic data.

Augmenting data structures

Basic methodology:

1. Choose underlying data structure

   (tree, hash table, linked list, stack, etc.)

2. Determine additional info needed.

3. Modify data structure to *maintain* additional info when the structure changes.

   (subject to insert/delete/etc.)

4. Develop new operations.

Plan

Two (or three?) examples of augmenting balanced BSTs

1. Order Statistics
2. Interval Queries
3. Orthogonal Range Searching

Order Statistics

Input

A set of integers.

Output: select(k)

The k^{th} item in the set.

52	**7**	**13**	**43**	**22**	**92**	**18**	**9**	**65**	**67**	**87**	**25**

select(4)

Dynamic Order Statistics

Implement a data structure that supports:

- insert(int key)
- delete(int key)

and also:

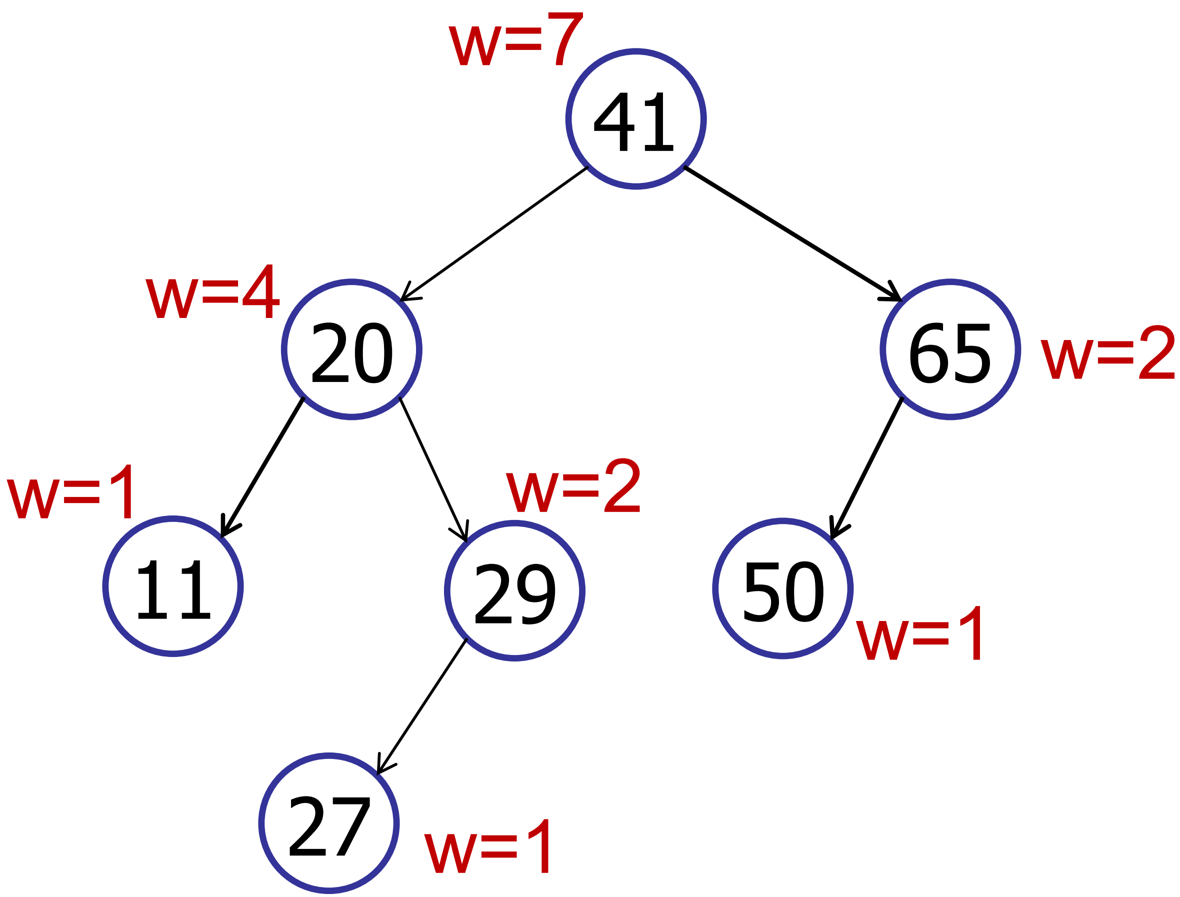
- select(int k)

7	**9**	**13**	**18**	**22**	**25**	**43**	**52**	**65**	**67**	**87**	**92**

select(4)

Dynamic Order Statistics

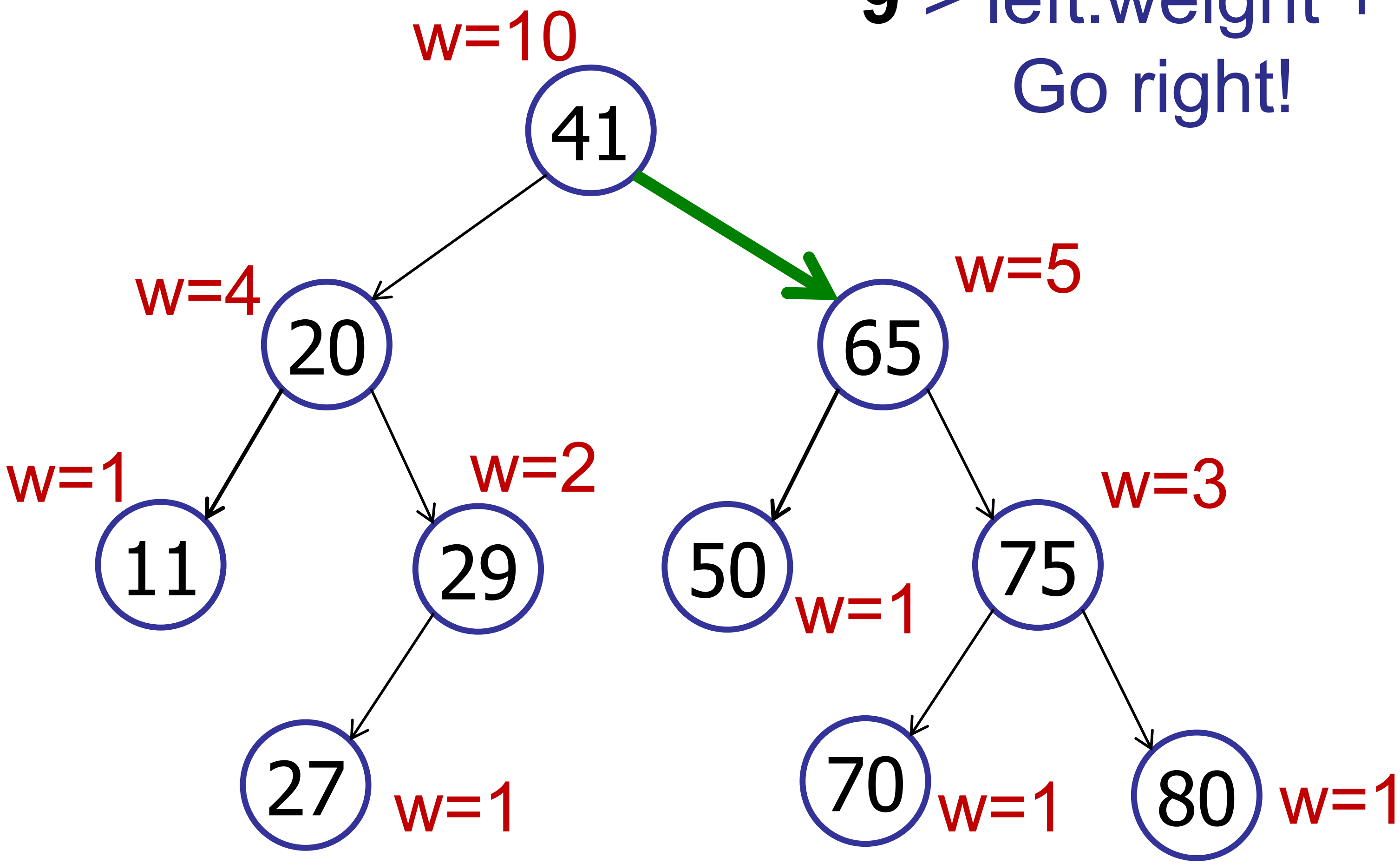
Idea: store *size* of sub-tree in every node



Dynamic Order Statistics

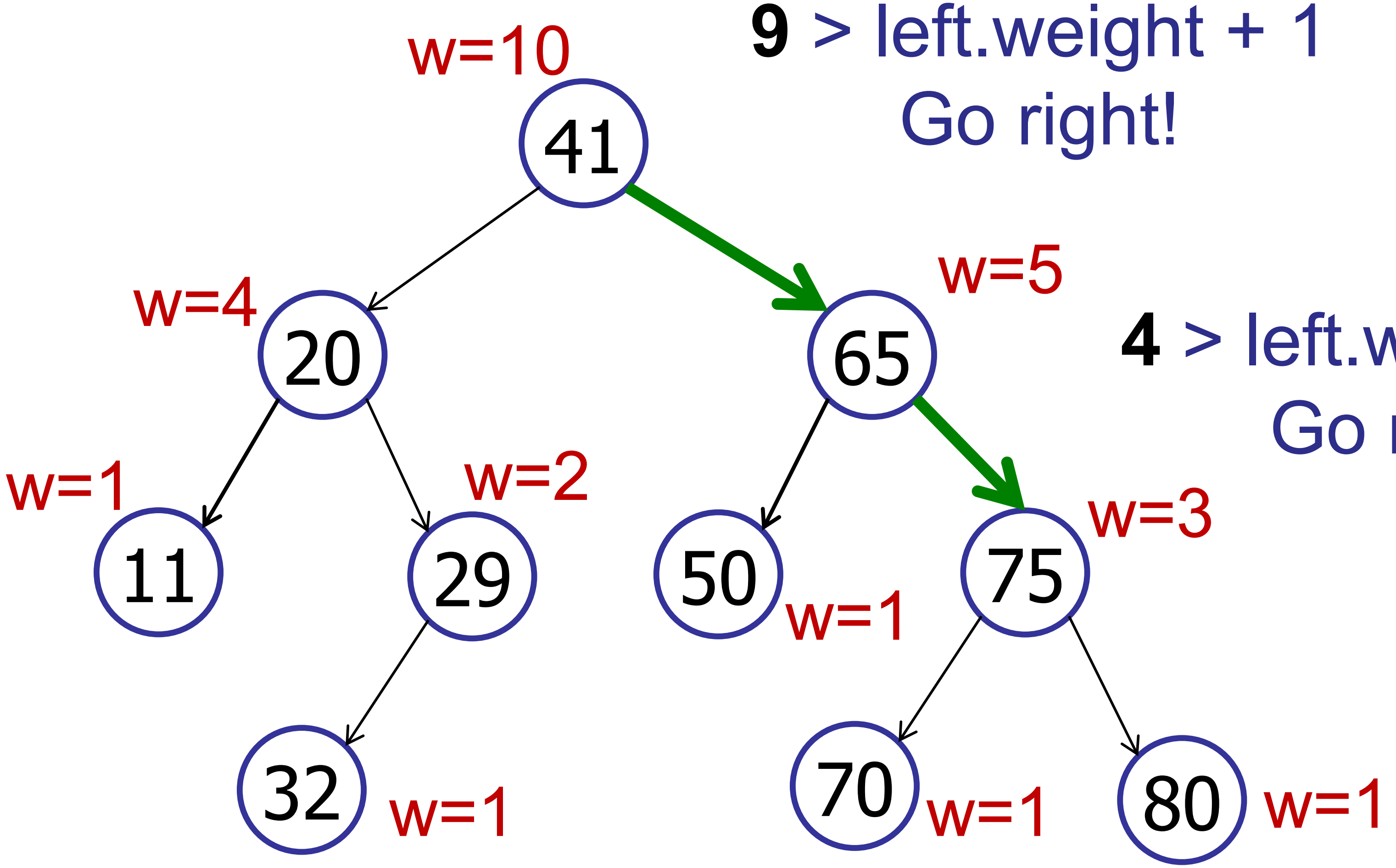
Example: select(9)

9 > left.weight + 1
Go right!



Dynamic Order Statistics

select(9)



9 > left.weight + 1

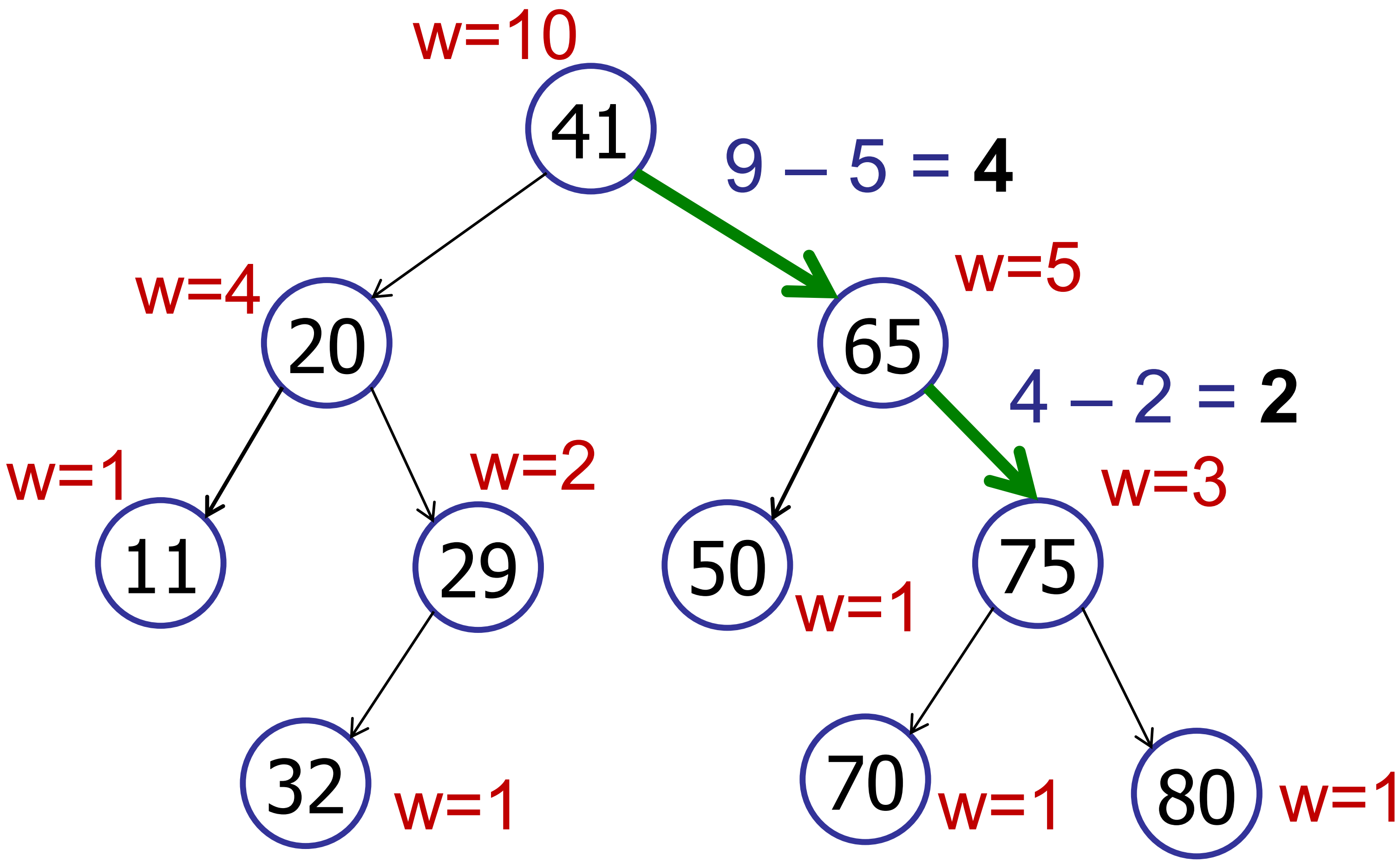
Go right!

4 > left.weight + 1

Go right!

Dynamic Order Statistics

select(9)



Dynamic Order Statistics

select(k)

```
rank = m_left.weight + 1;
if (k == rank) then
    return v;
else if (k < rank) then
    return m_left.select(k);
else if (k > rank) then
    return m_right.select(k–rank);
```

Dynamic Order Statistics

select(k) : finds the node with rank k

Example: find the 10th tallest student in the class.

Dynamic Order Statistics

select(k) : finds the node with rank k

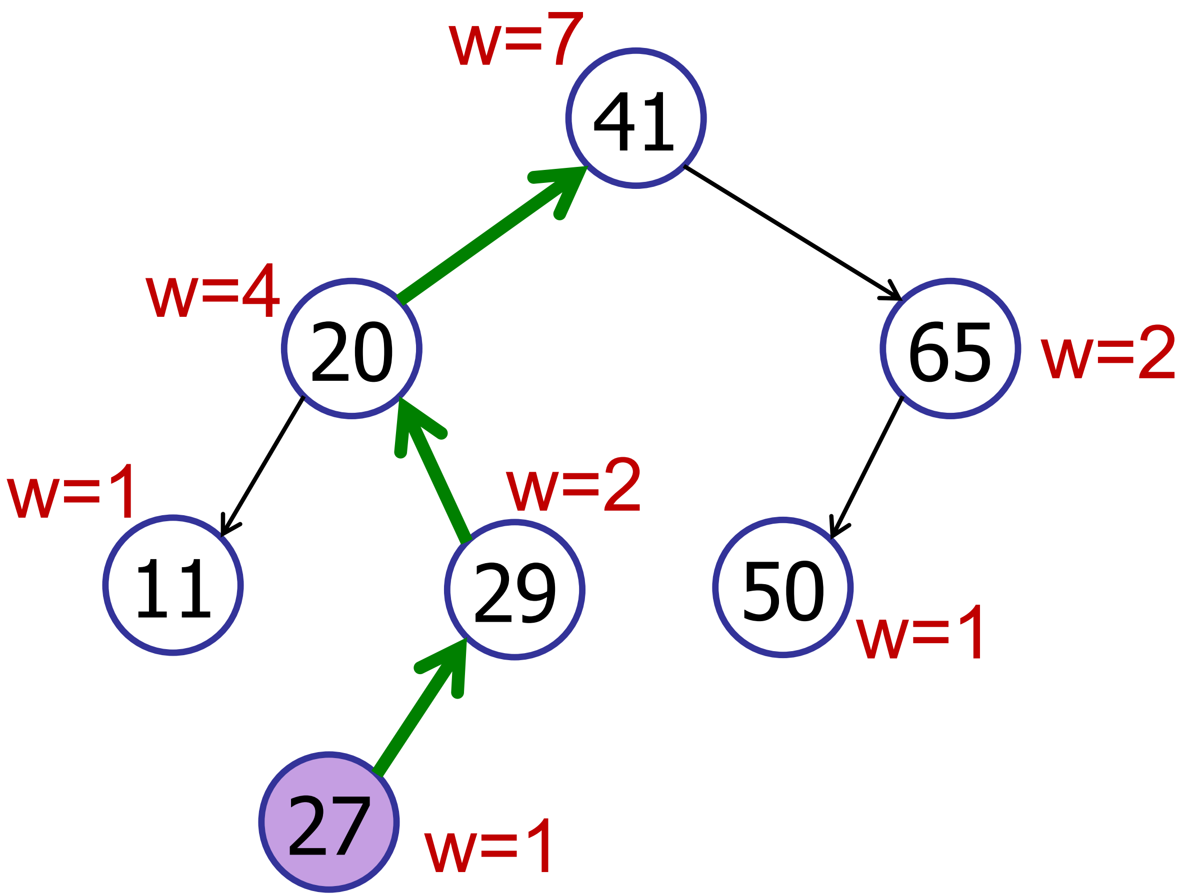
Example: find the 10th tallest student in the class.

rank(v) : computes the rank of a node v

Example: determine the percentile of Johnny's height.
Is Johnny in the 10^{th} percentile or the 90^{th} percentile?

Dynamic Order Statistics

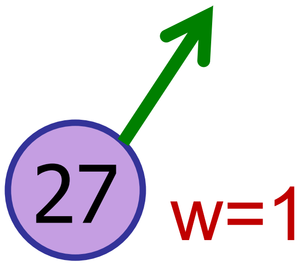
Example: rank(27)



rank = 1

Dynamic Order Statistics

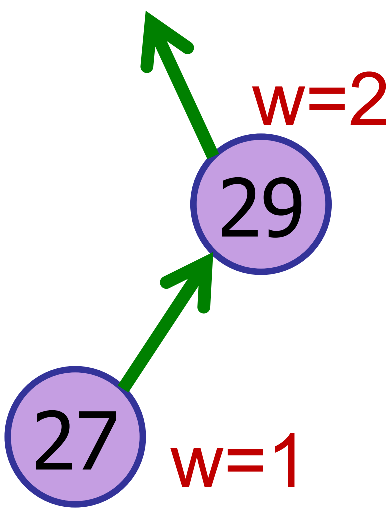
Example: rank(27)



rank = 1

Dynamic Order Statistics

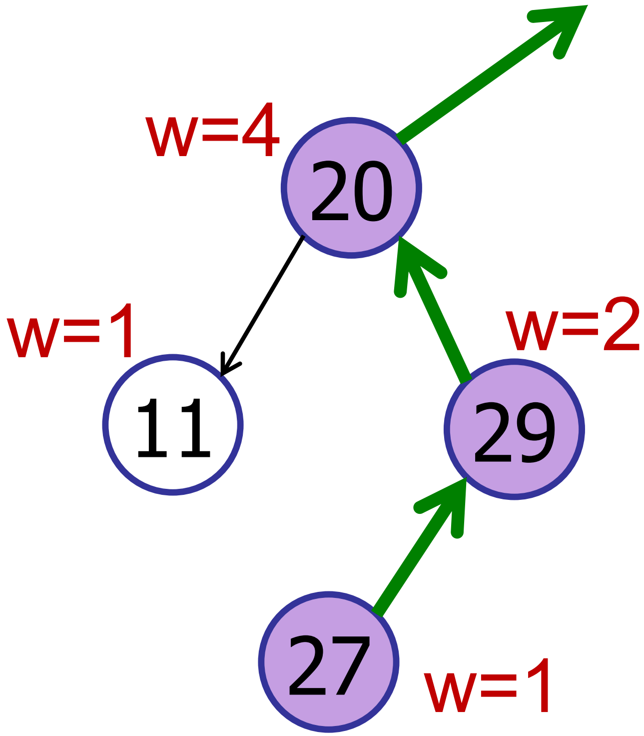
Example: rank(27)



rank = 1

Dynamic Order Statistics

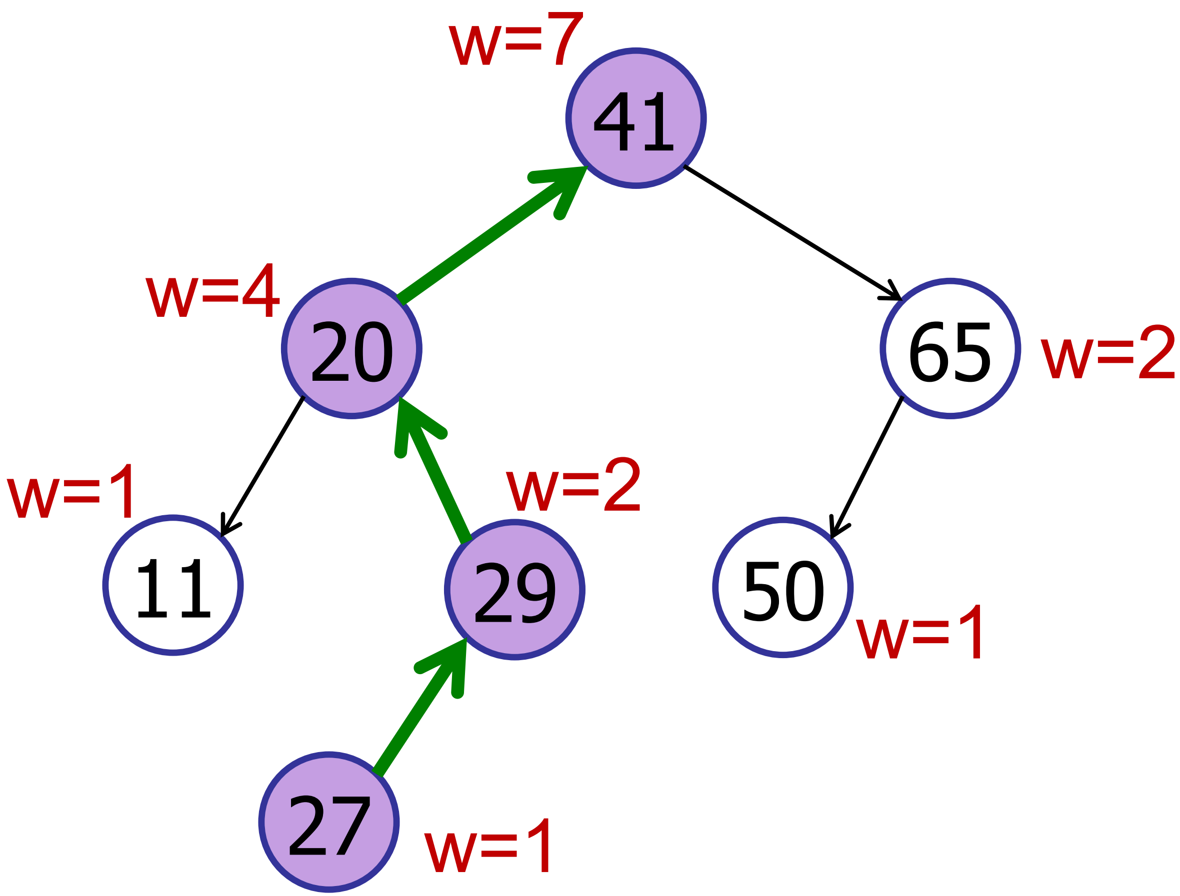
Example: rank(27)



rank = 1 + 2

Dynamic Order Statistics

Example: rank(27)



rank = 1 + 2 = 3

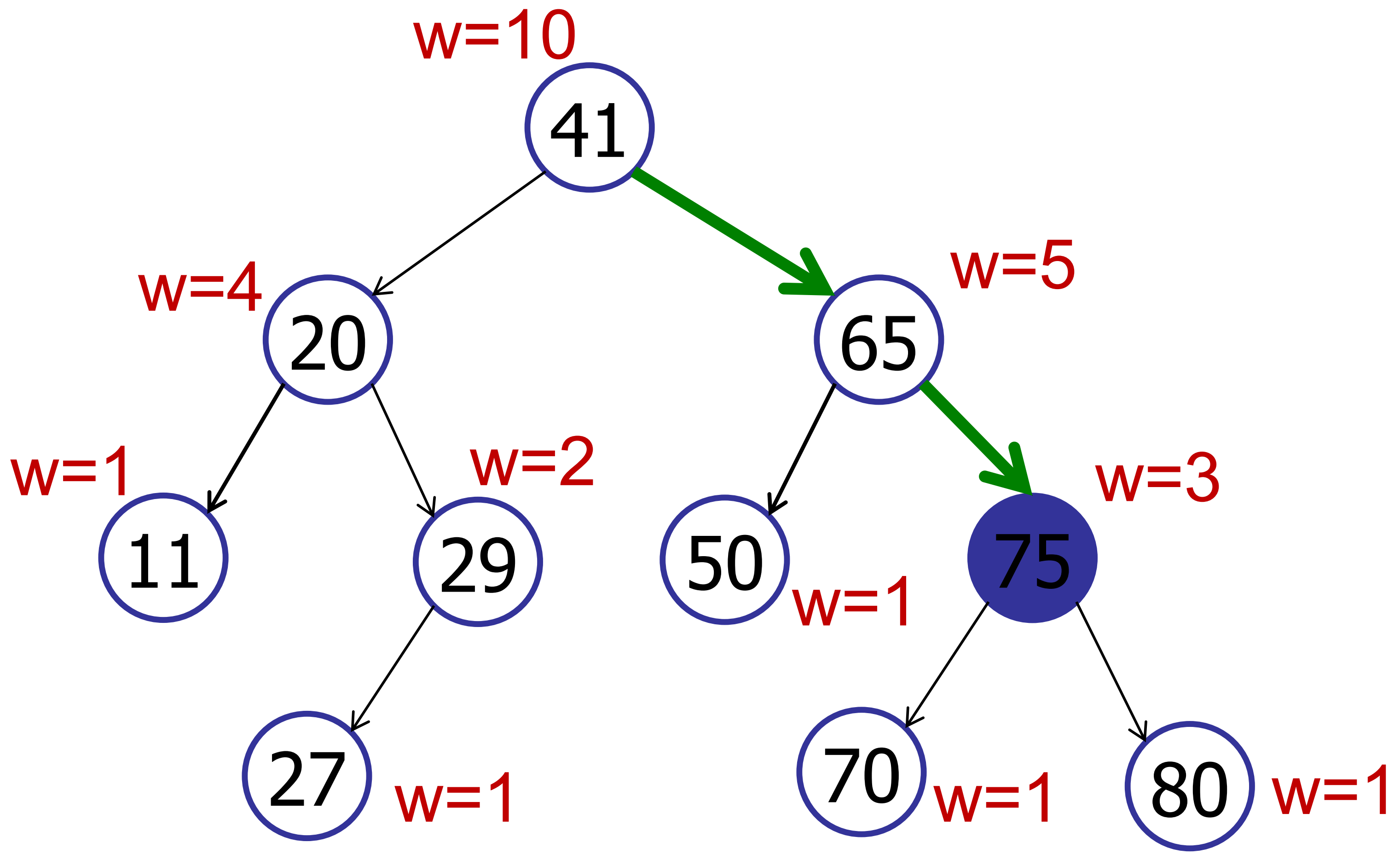
Dynamic Order Statistics

Rank(v) : computes the rank of a node v

```
rank(node)
    rank = node.left.weight + 1;
    while (node != null) do
        if node is left child then
            do nothing
        else if node is right child then
            rank += node.parent.left.weight + 1;
        node = node.parent;
    return rank;
```

Dynamic Order Statistics

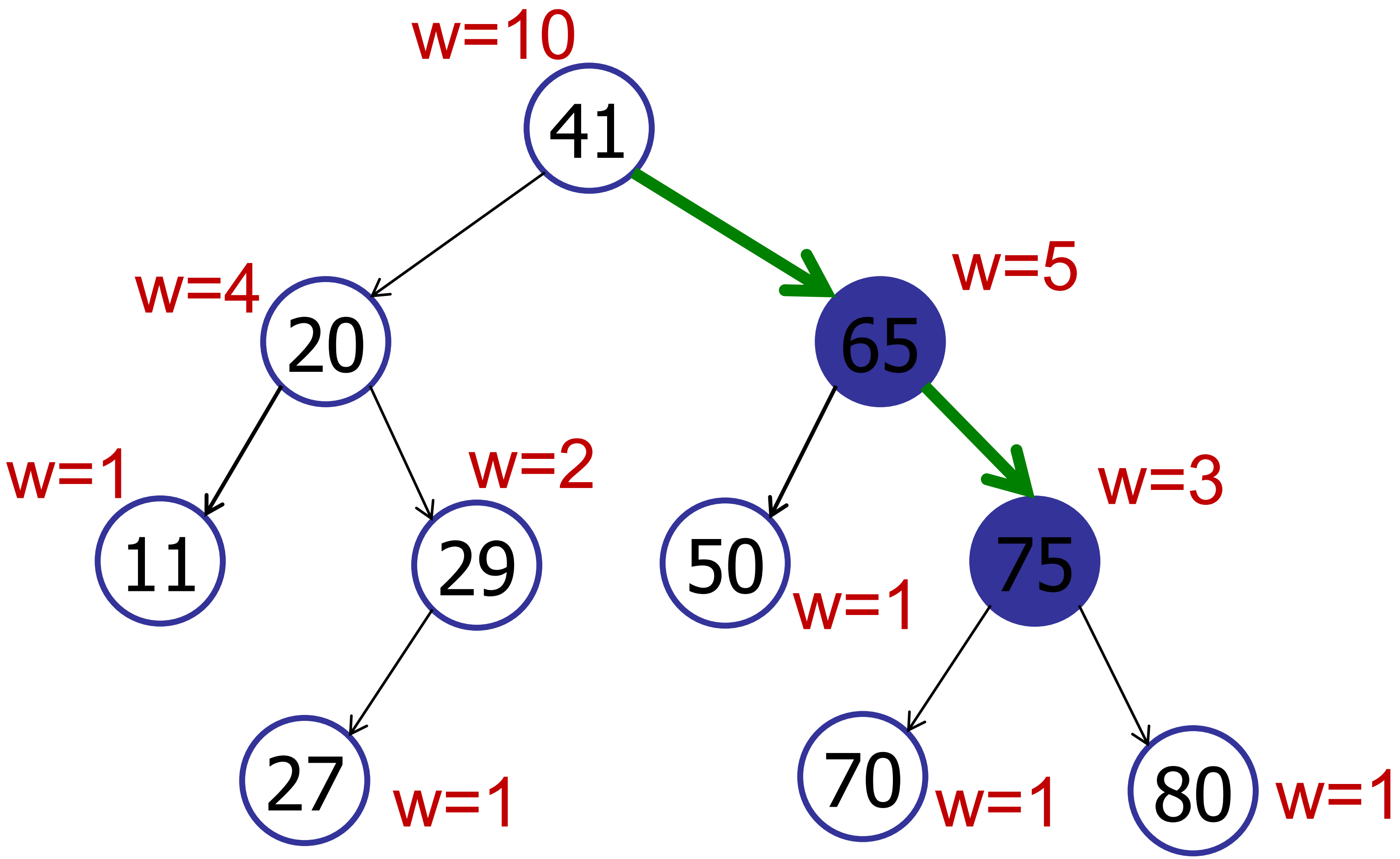
rank(75)



rank = 2

Dynamic Order Statistics

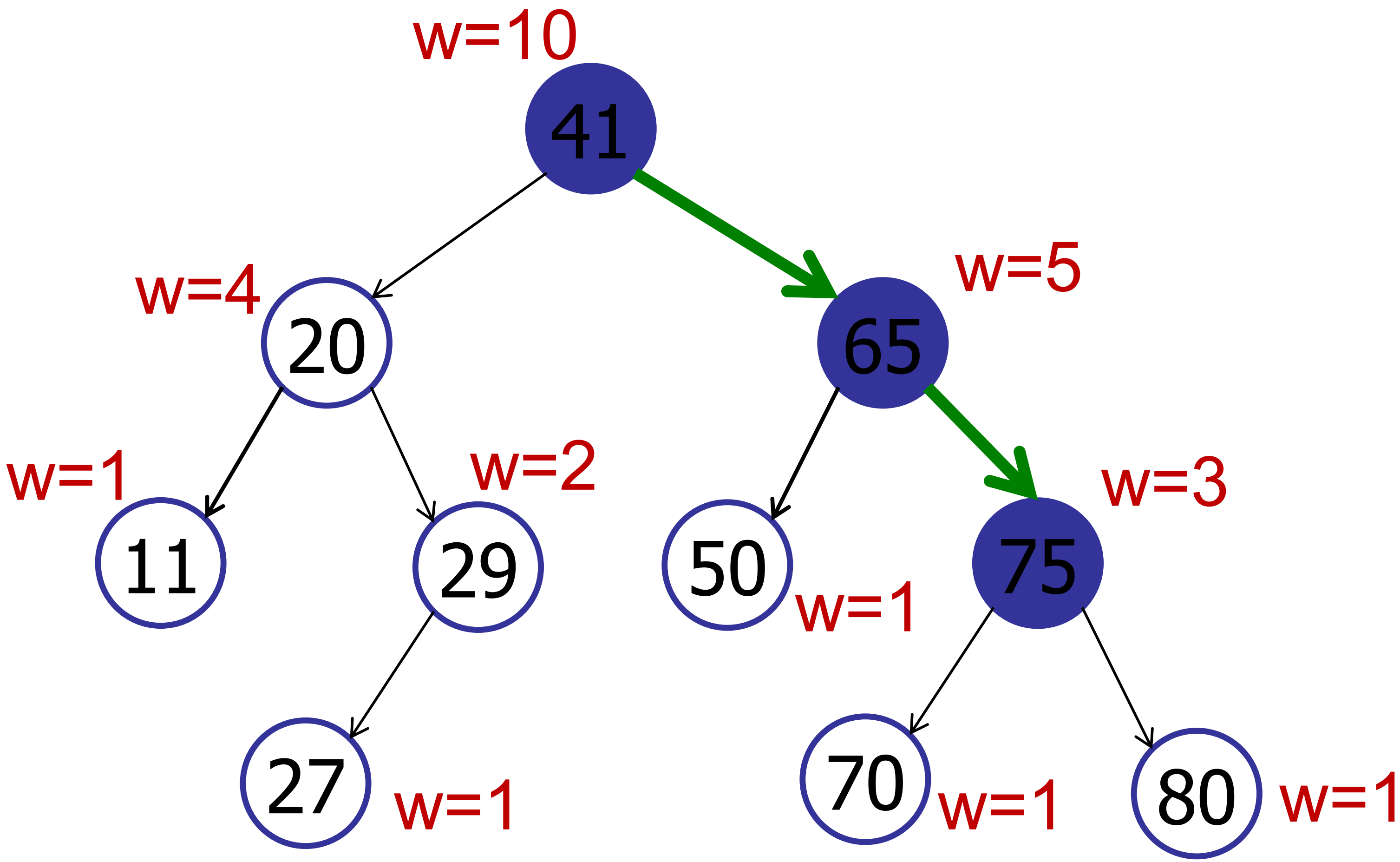
rank(75)



rank = 2 + 2

Dynamic Order Statistics

rank(75)



rank = 2 + 2 + 5 = 9

Dynamic Order Statistics

Rank(v) : computes the rank of a node v

```
rank(node)
    rank = node.left.weight + 1;
    while (node != null) do
        if node is left child then
            do nothing
        else if node is right child then
            rank += node.parent.left.weight + 1;
        node = node.parent;
    return rank;
```

Augmenting data structures

Basic methodology:

1. Choose underlying data structure:

    AVL tree

2. Determine additional info needed:

    Weight of each node

3. Maintained info as data structure is modified.

    Update weights as needed

4. Develop new operations using the new info.

    Select and Rank

Augmenting data structures

Basic methodology:

1. Choose underlying data structure:

    AVL tree

2. Determine additional info needed:

    Weight of each node

3. Maintained info as data structure is modified.

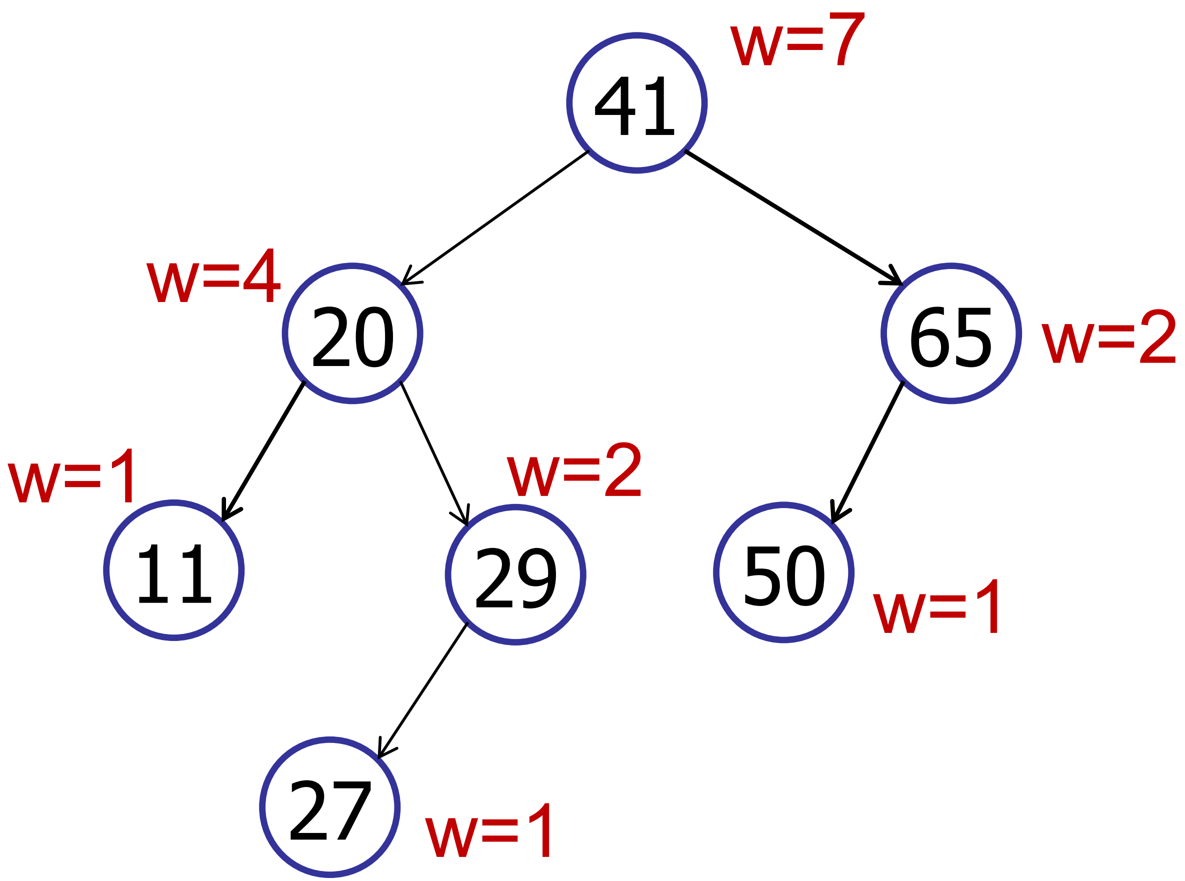
    Update weights as needed

4. Develop new operations using the new info.

    Select and Rank

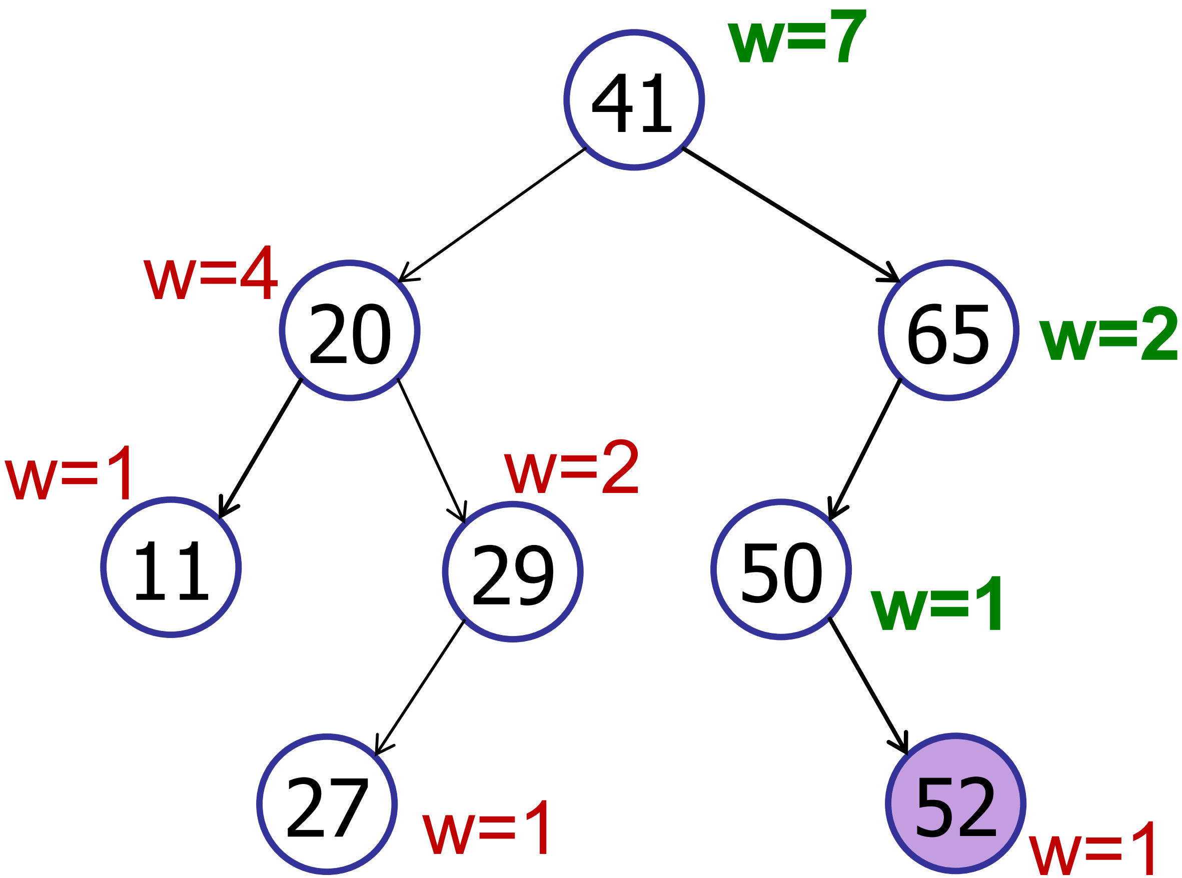
Augmented Trees

Maintain weight during insertions:



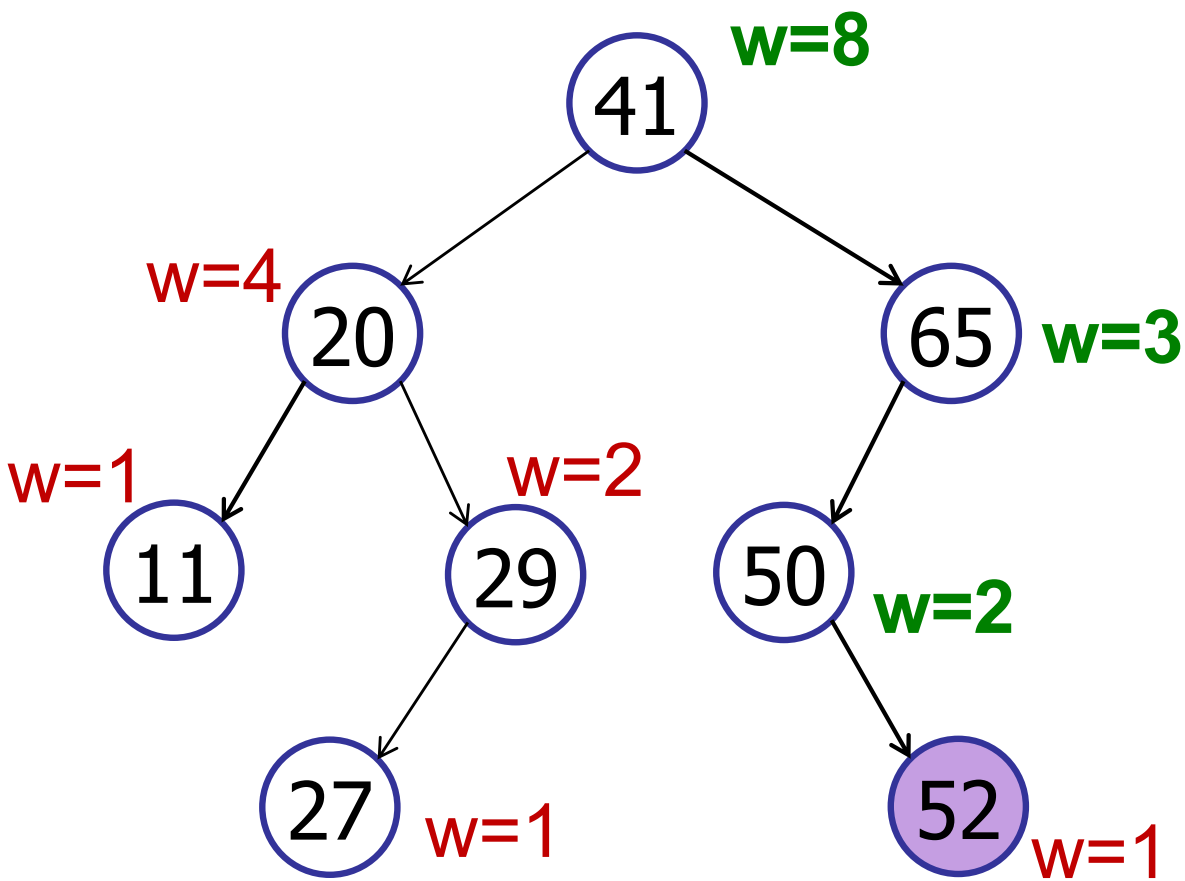
Augmented Trees

Maintain weight during insertions:



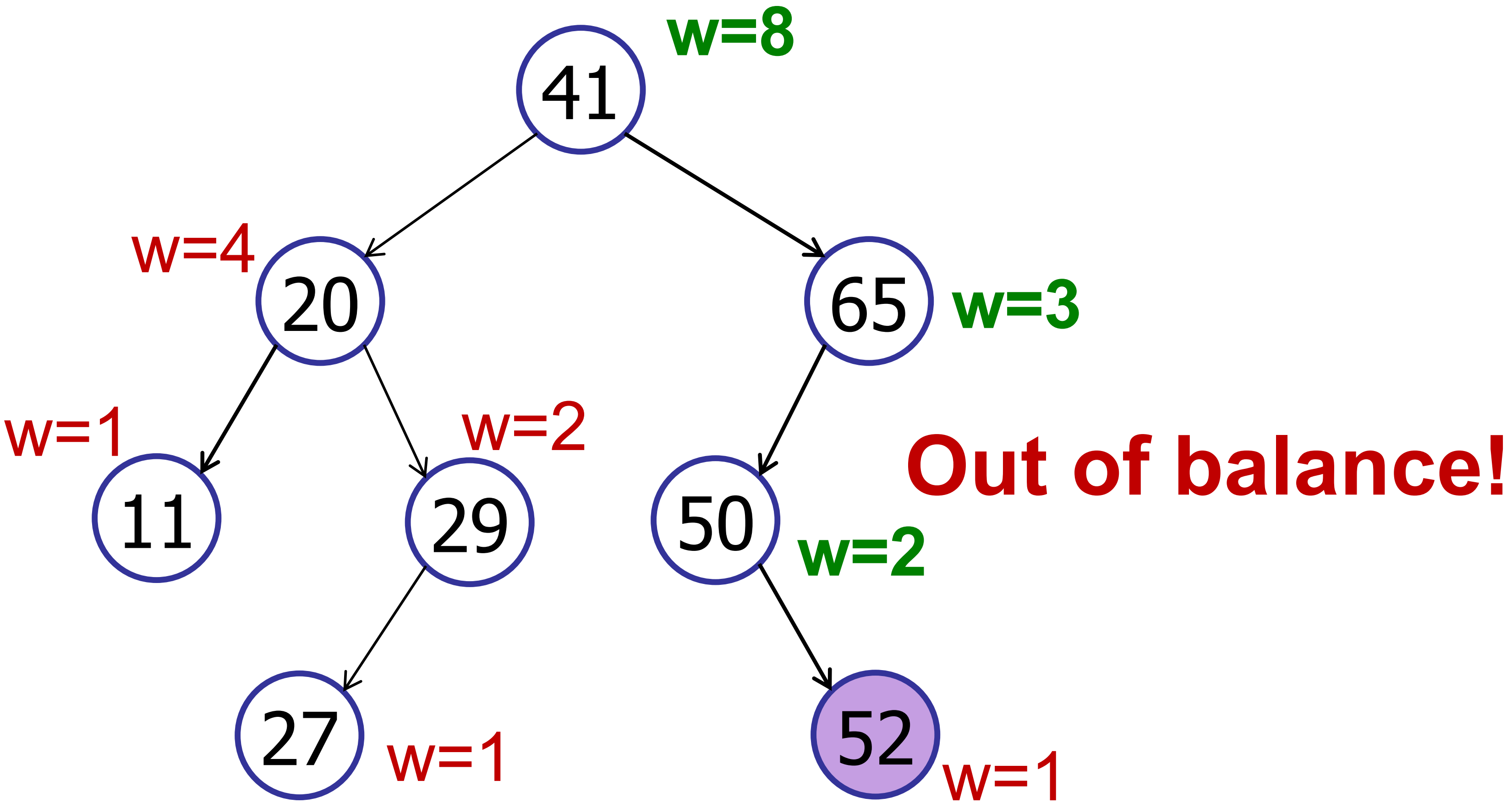
Augmented Trees

Maintain weight during insertions:



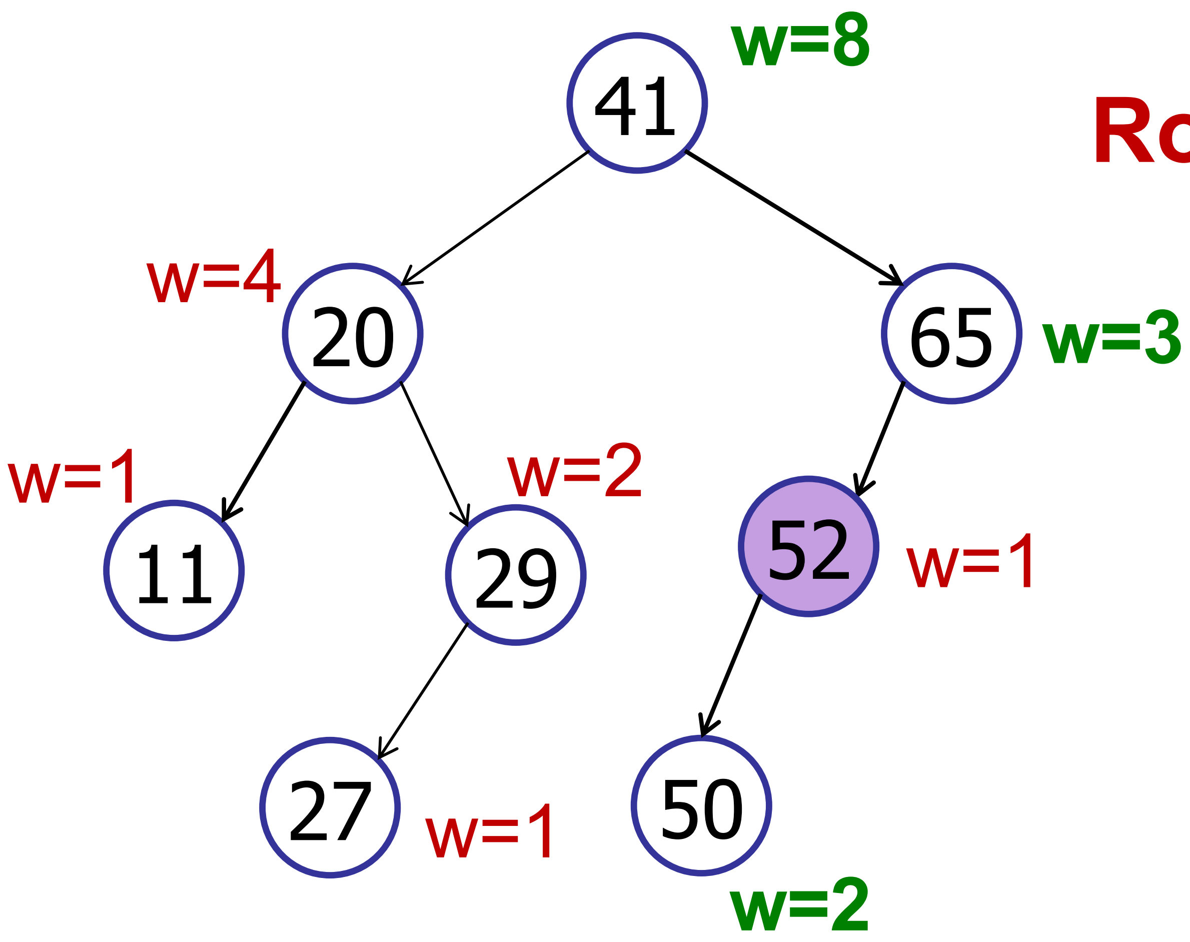
Augmented Trees

Maintain weight during insertions:



Augmented Trees

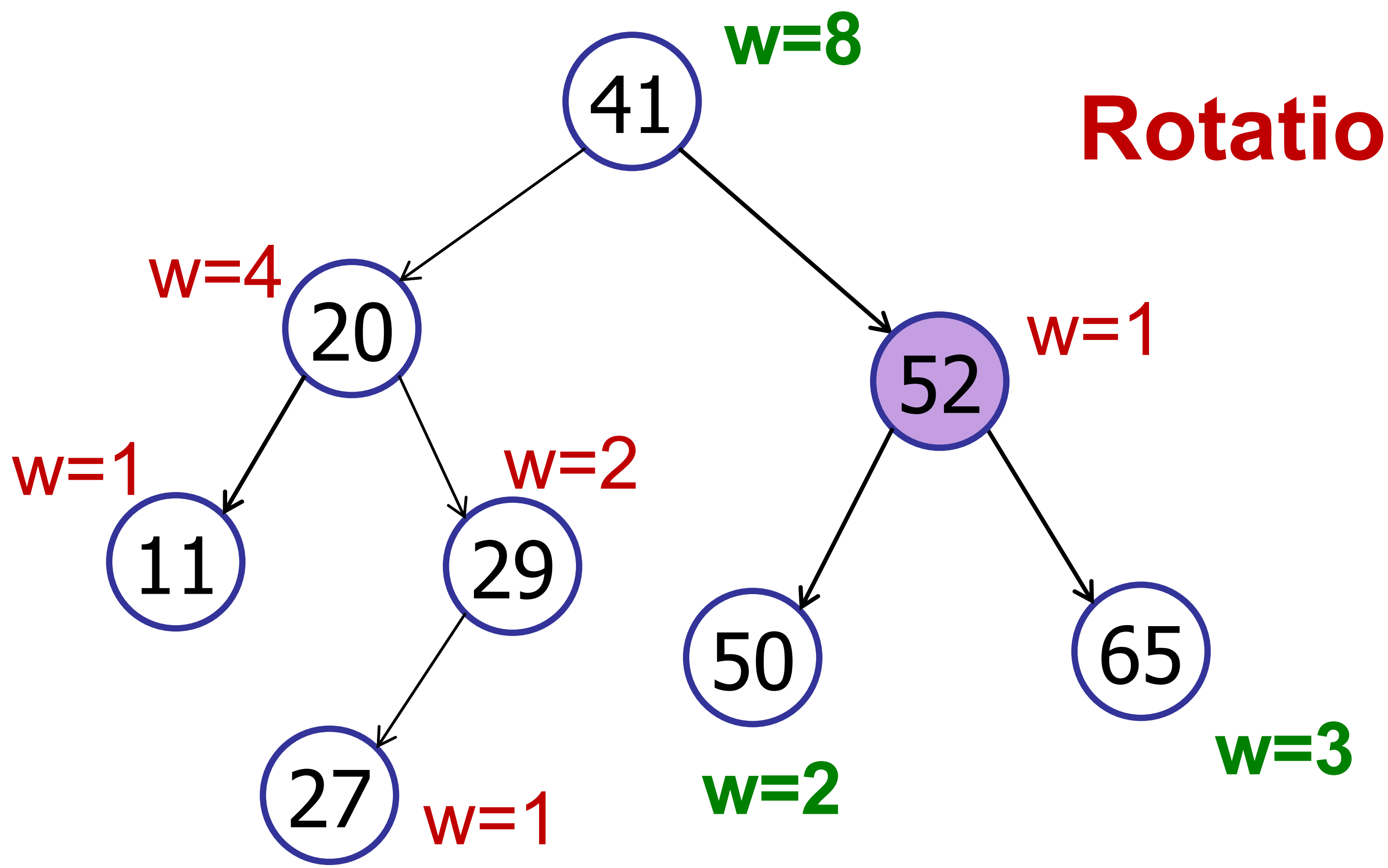
Maintain weight during insertions:



Rotation 1:

Augmented Trees

Maintain weight during insertions:

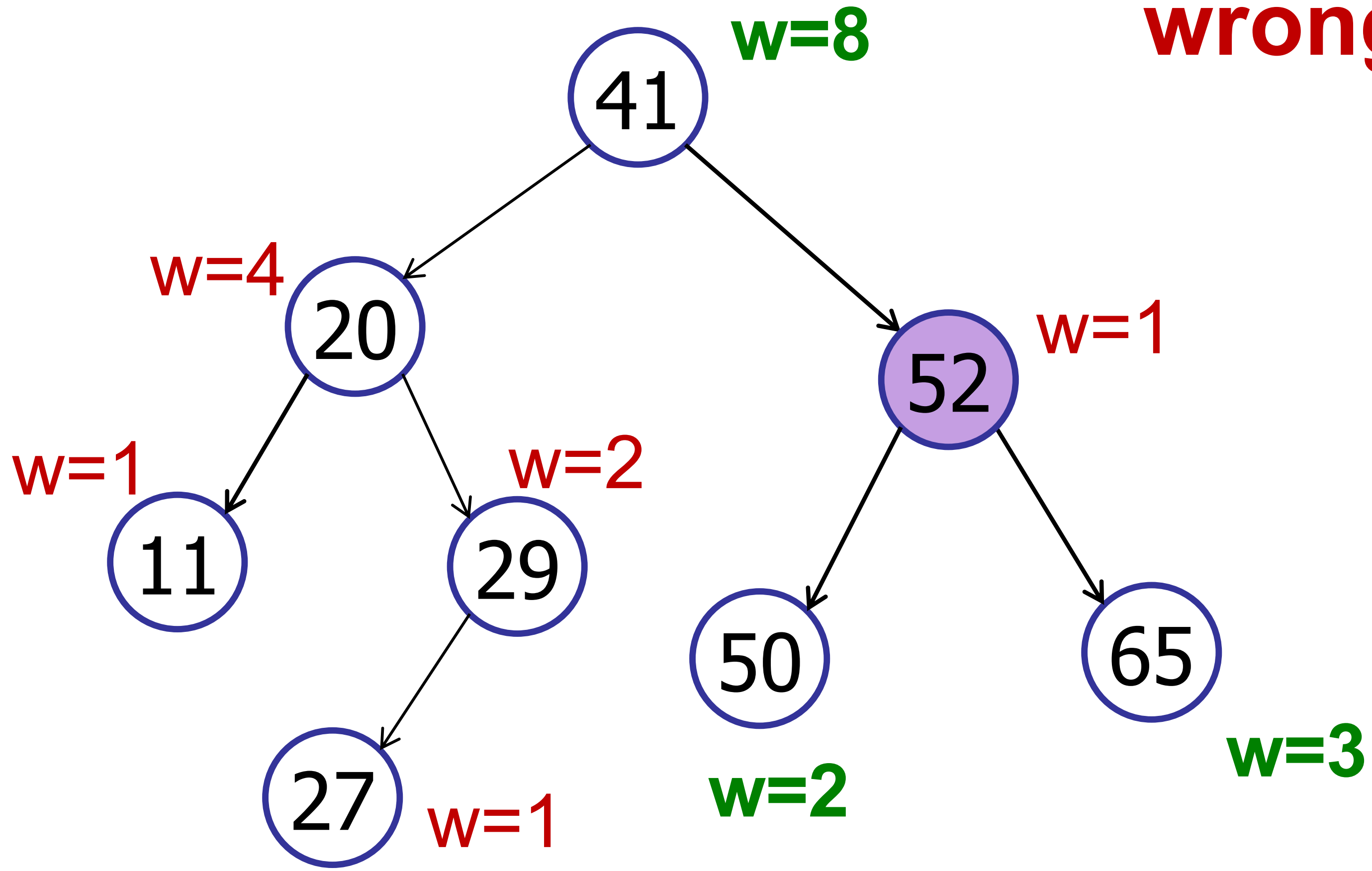


Rotation 2:

Augmented Trees

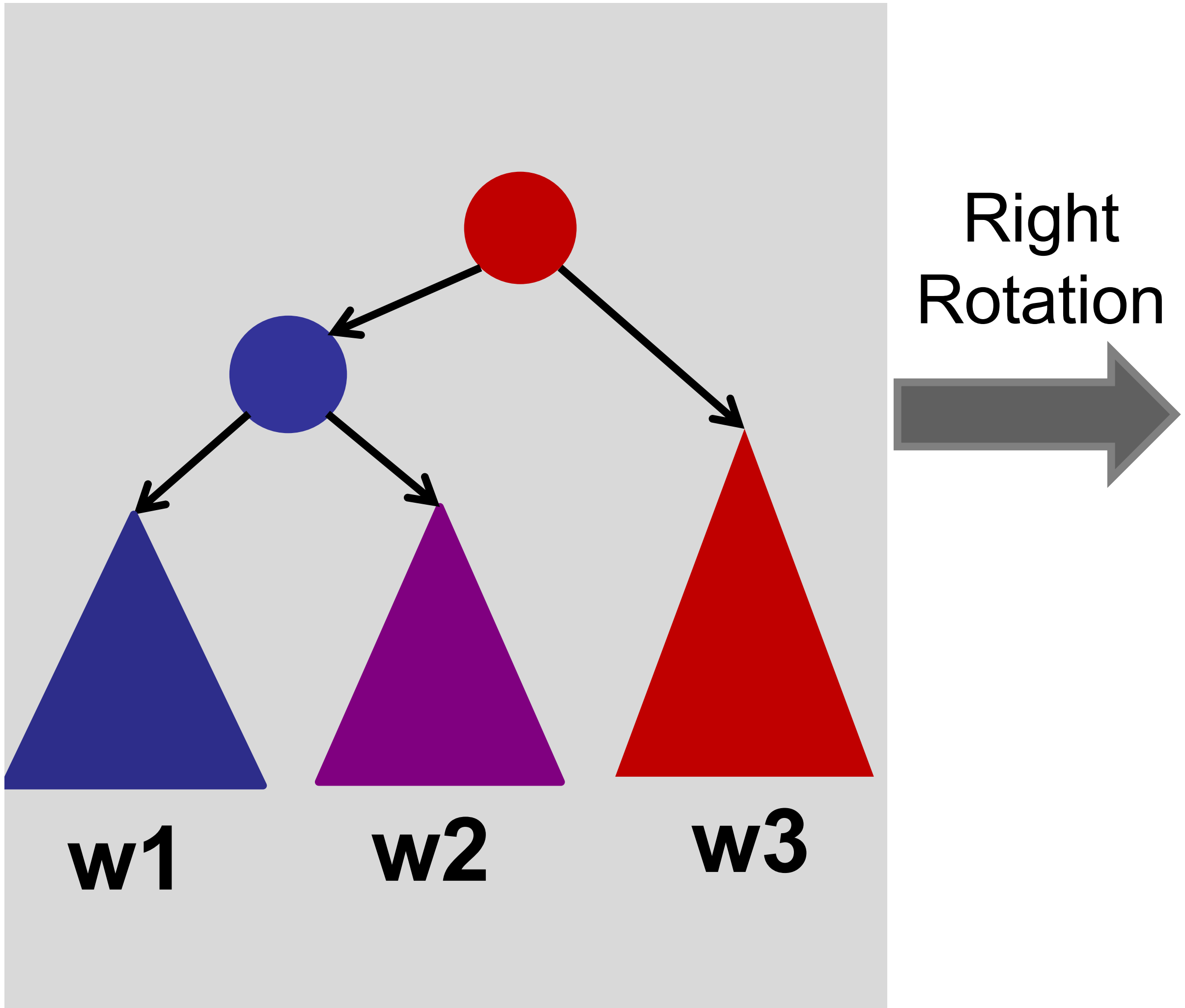
How to update weights on rotation?

Weights all wrong!



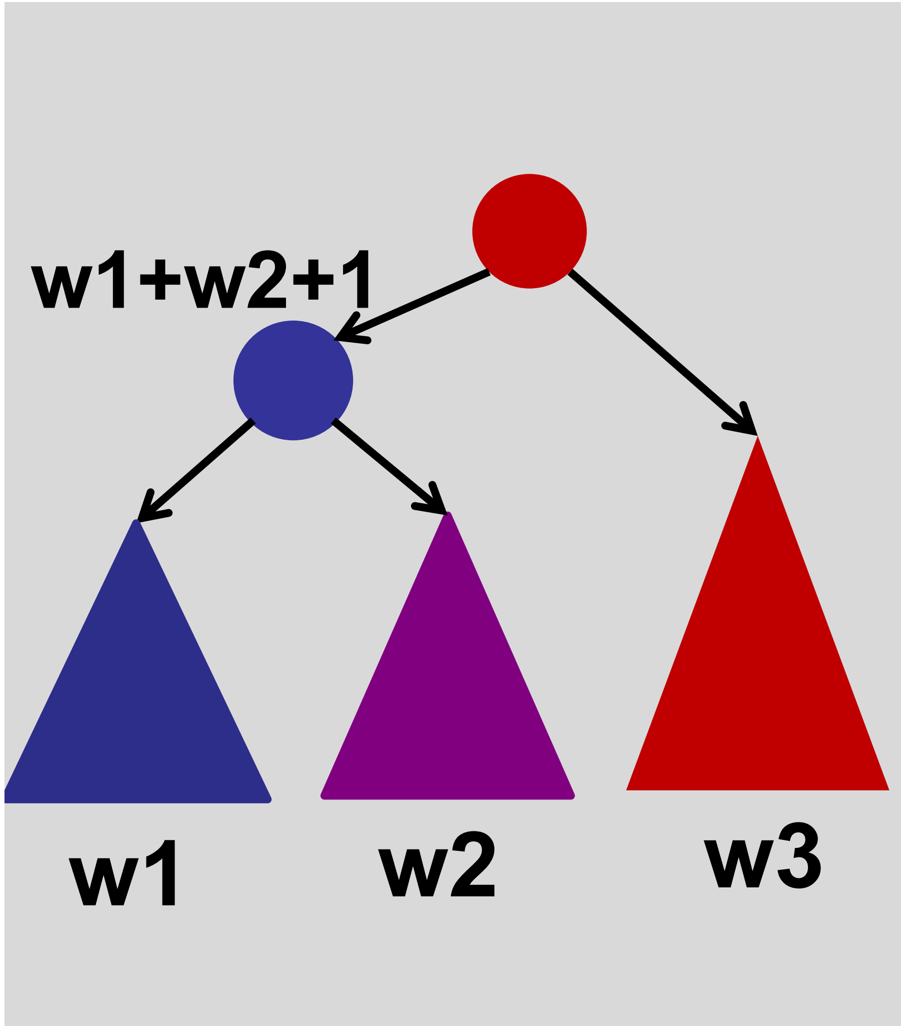
Augmented Trees

Maintain weight during rotations:



Augmented Trees

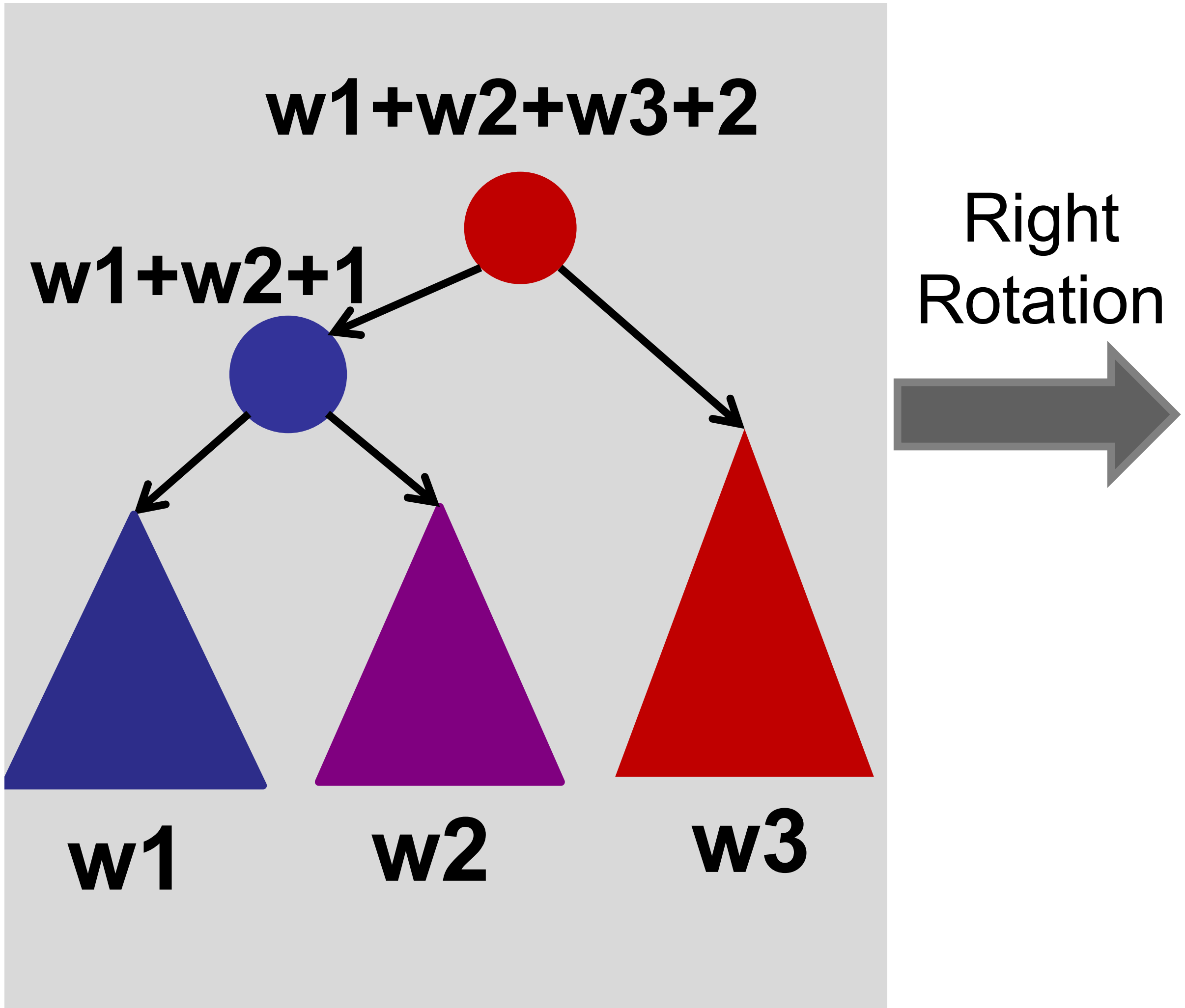
Maintain weight during rotations:



Right
Rotation

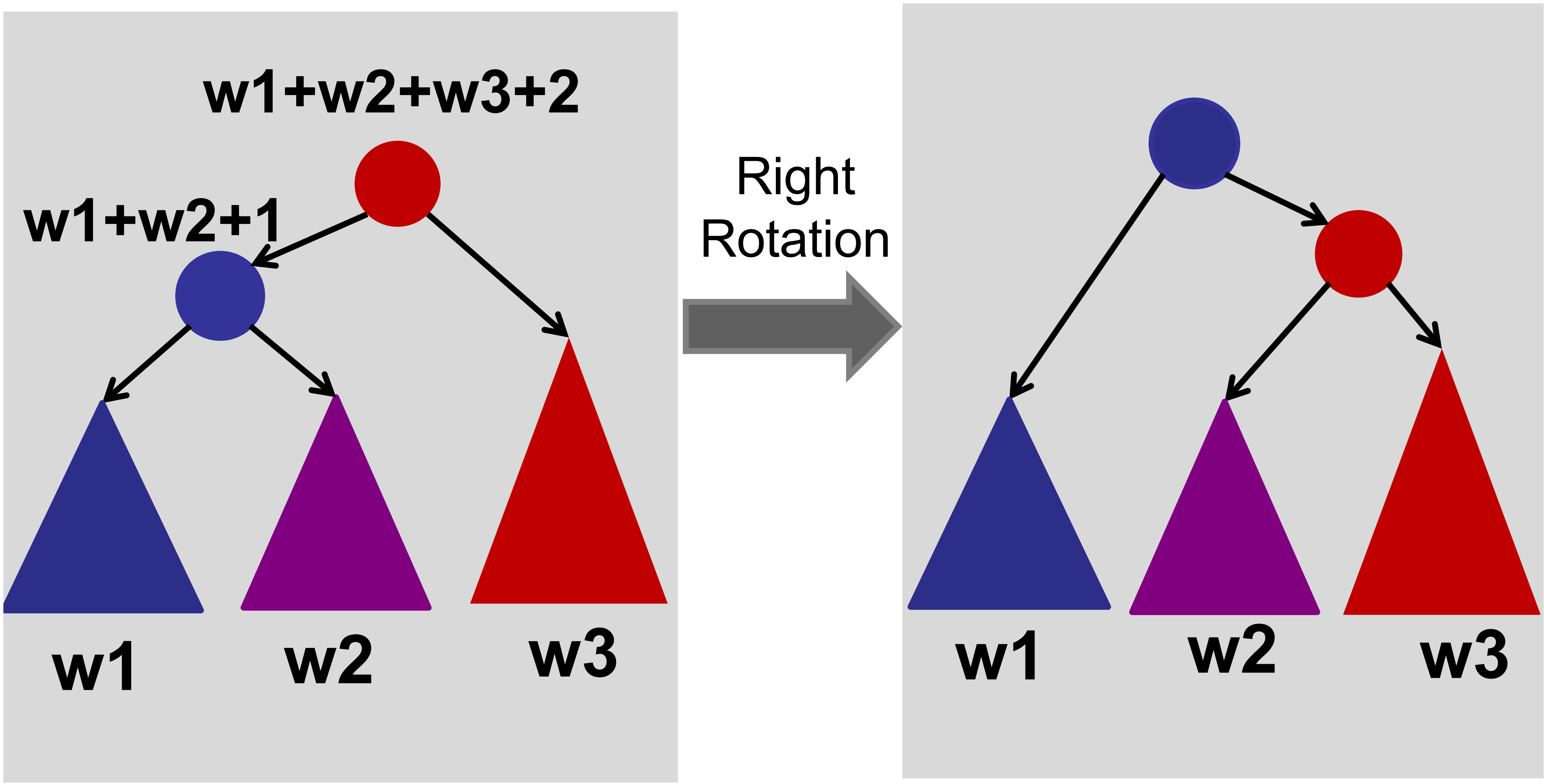
Augmented Trees

Maintain weight during rotations:



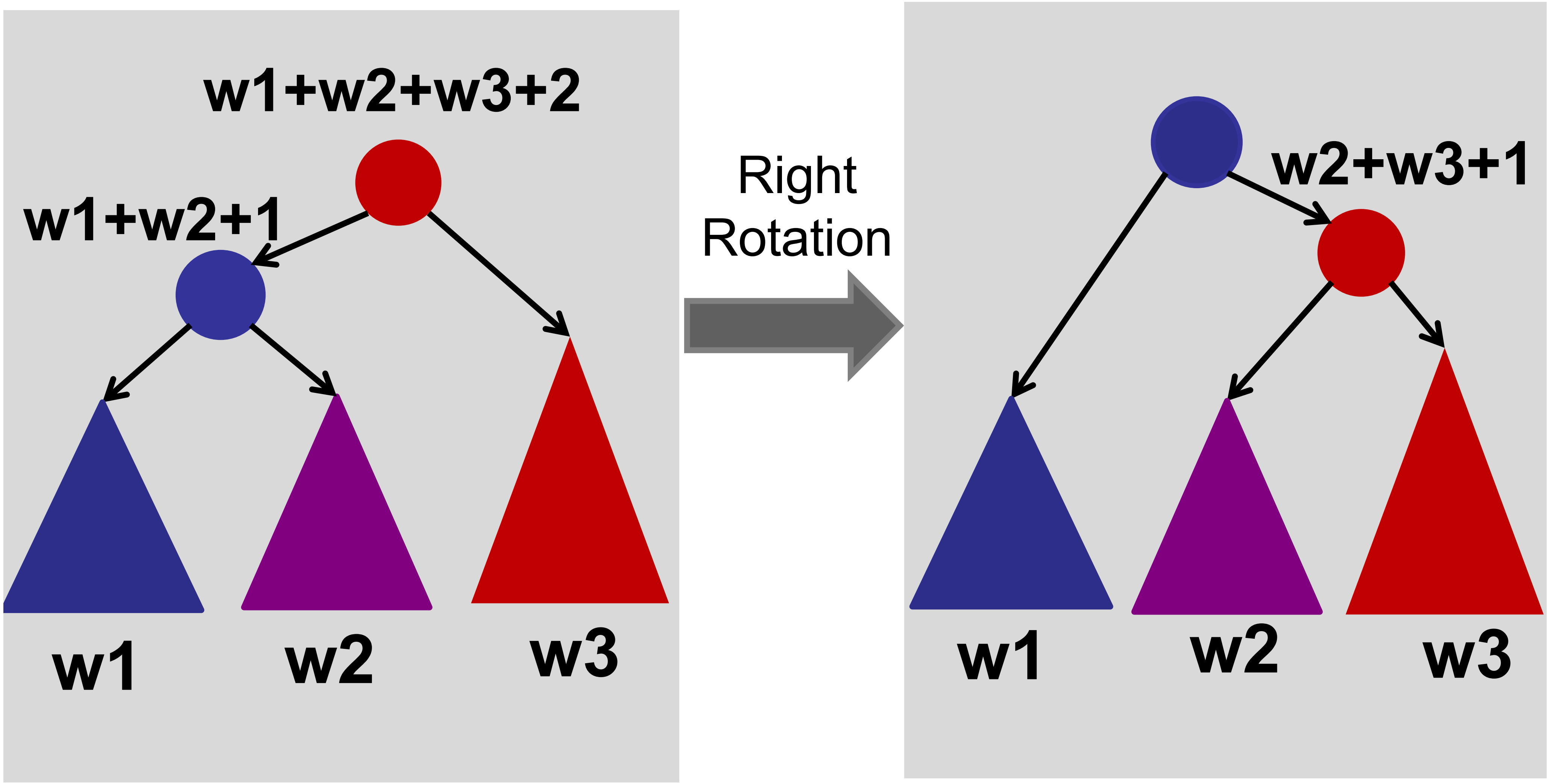
Augmented Trees

Maintain weight during rotations:



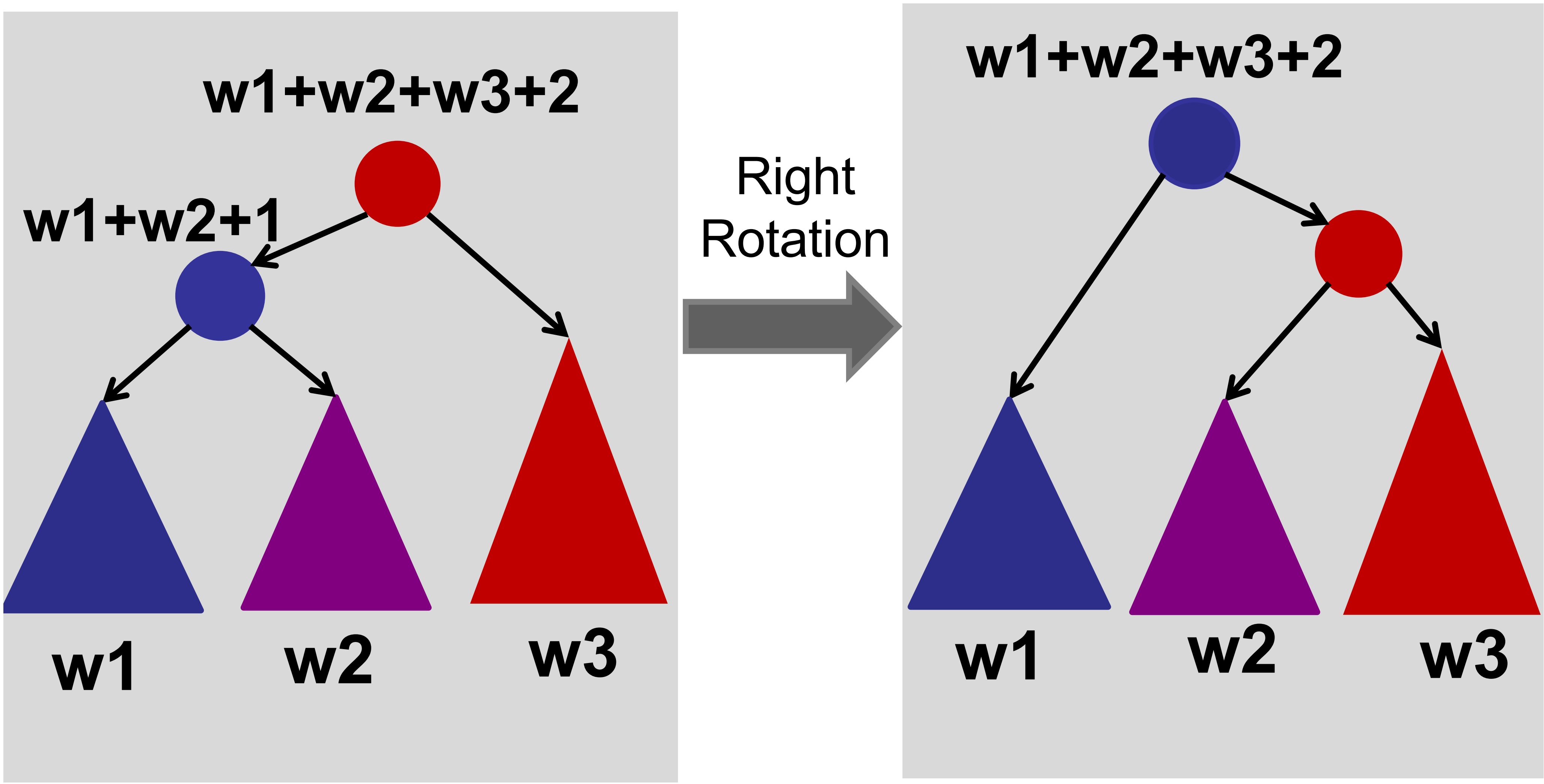
Augmented Trees

Maintain weight during rotations:



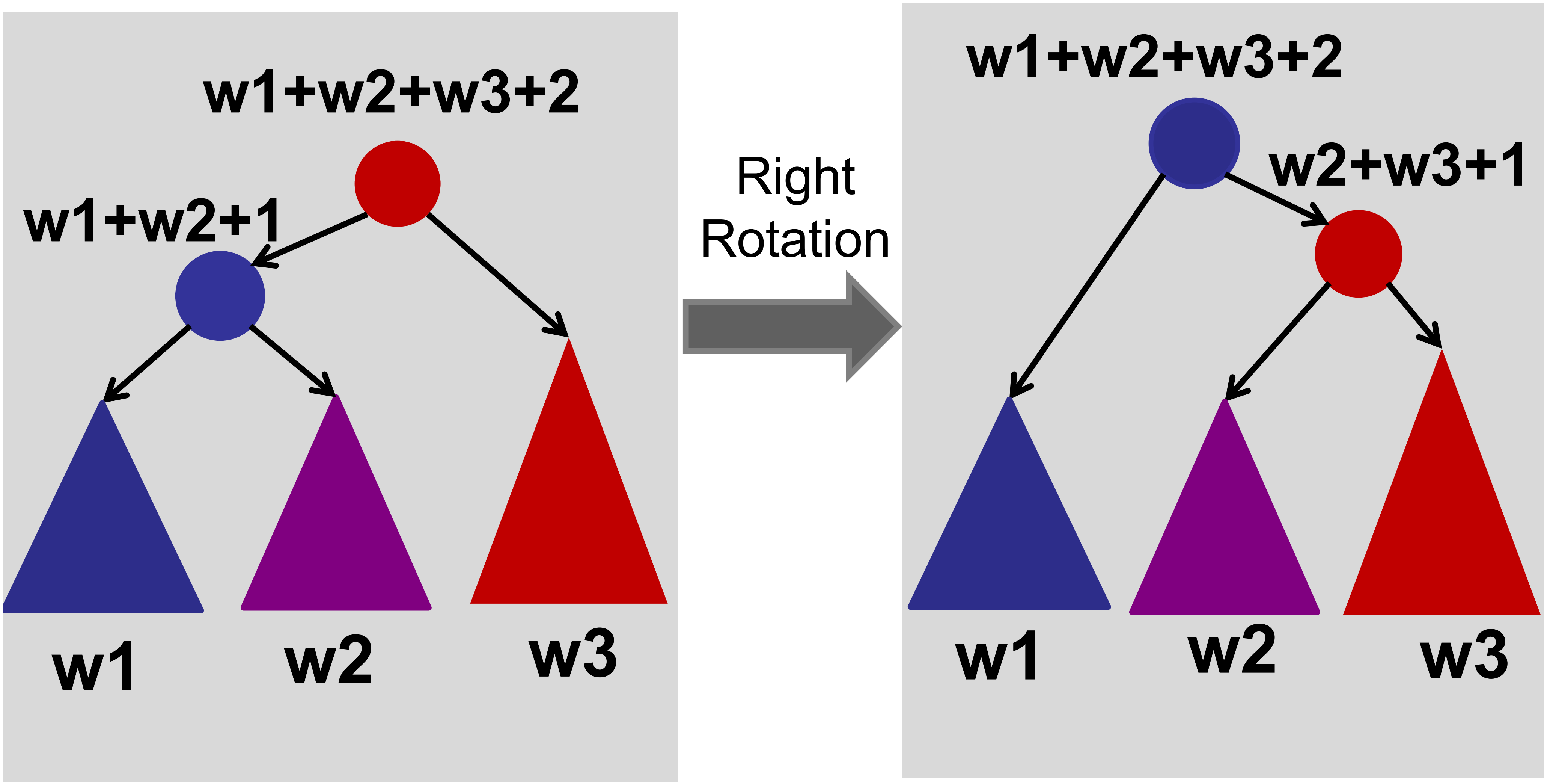
Augmented Trees

Maintain weight during rotations:



Augmented Trees

Maintain weight during rotations:



How long does it take to update the weights during a rotation?

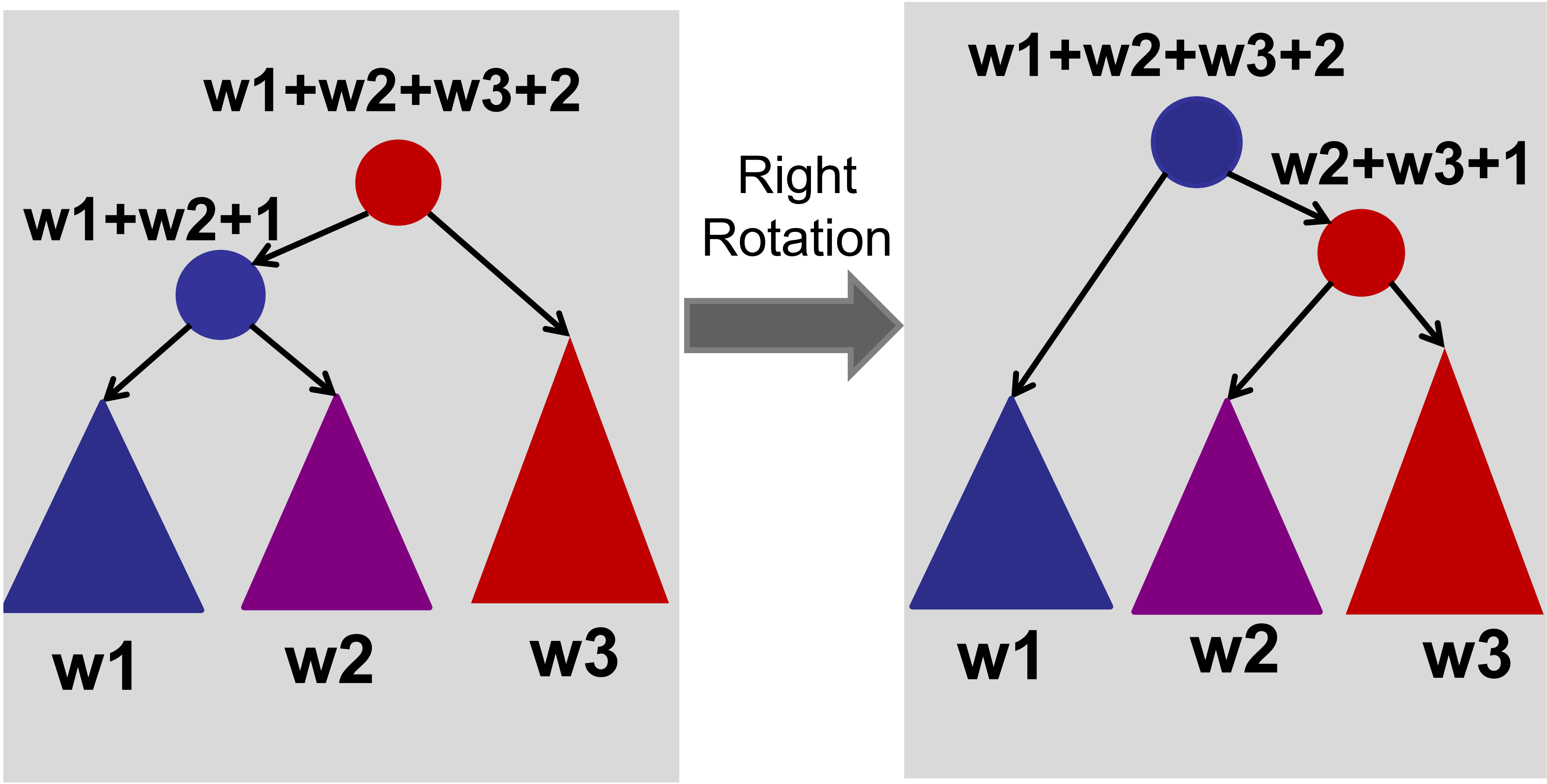
1. O(1)
2. O(log n)
3. O(n)
4. O(n^2)
5. What is a rotation?

ARCHIPELAGO

is open

Augmented Trees

Maintain weight during rotations:



Augmenting data structures

Basic methodology:

1. Choose underlying data structure

   (tree, hash table, linked list, stack, etc.)

2. Determine additional info needed.

3. Verify that the additional info can be maintained as the data structure is modified.

   (subject to insert/delete/etc.)

4. Develop new operations using the new info.

Next few lectures…

Three examples of augmenting balanced BSTs

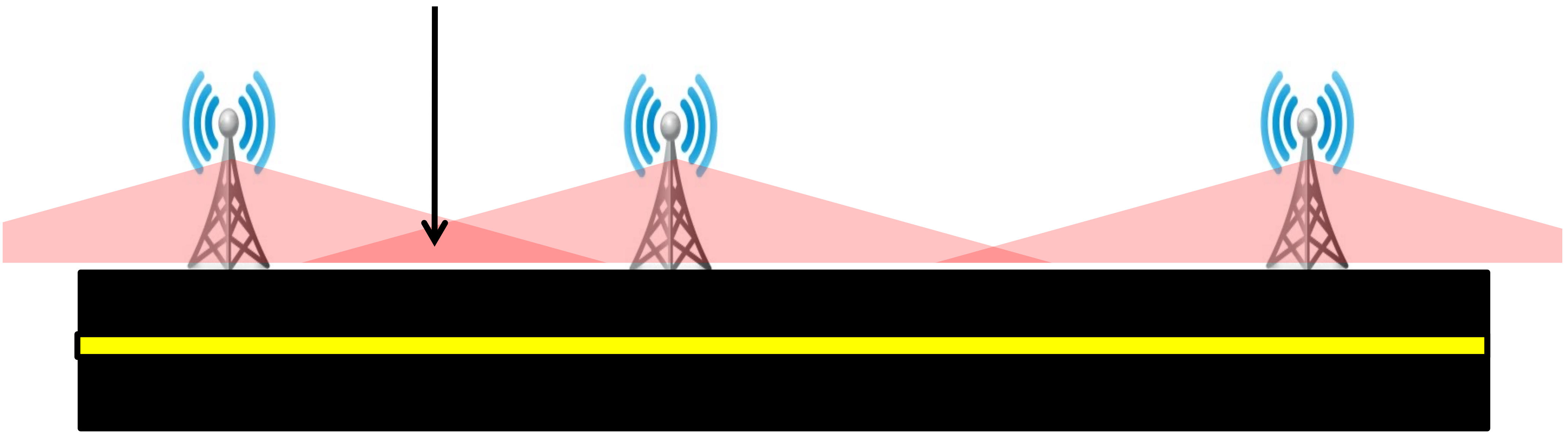
1. Order Statistics

2. Interval Queries

3. Orthogonal Range Searching

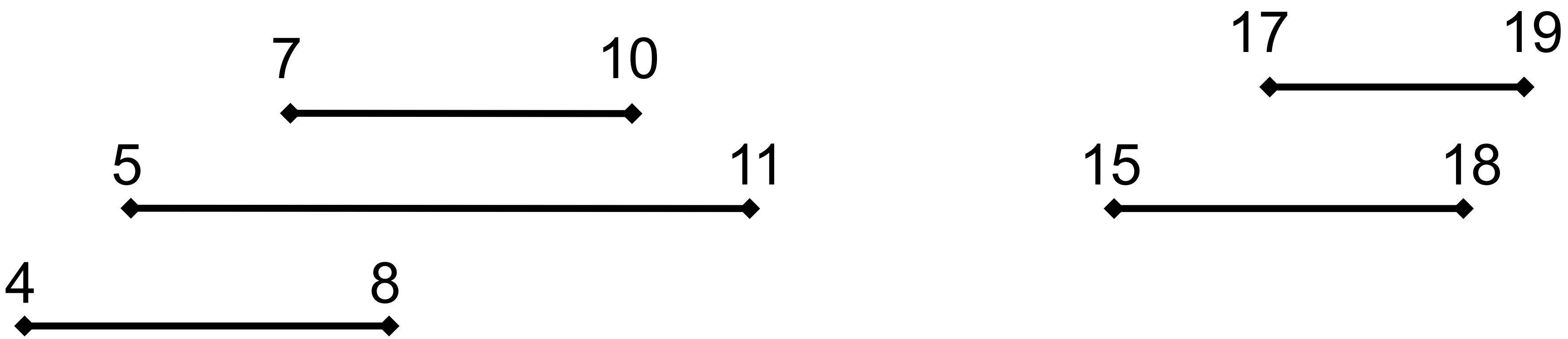
Cell Tower Coverage

Find a tower that covers my location.



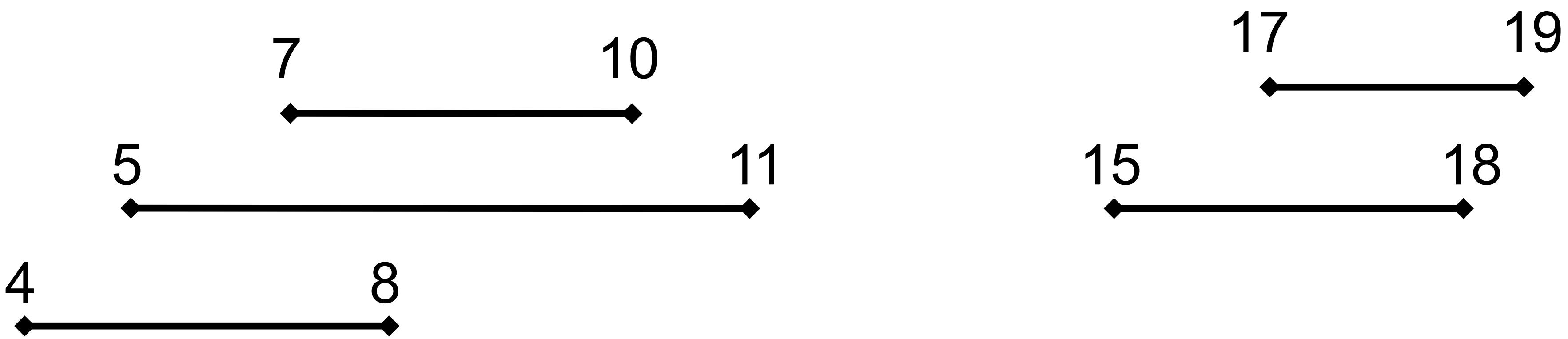
Cell Tower Coverage

Find a tower that covers my location.



Cell Tower Coverage

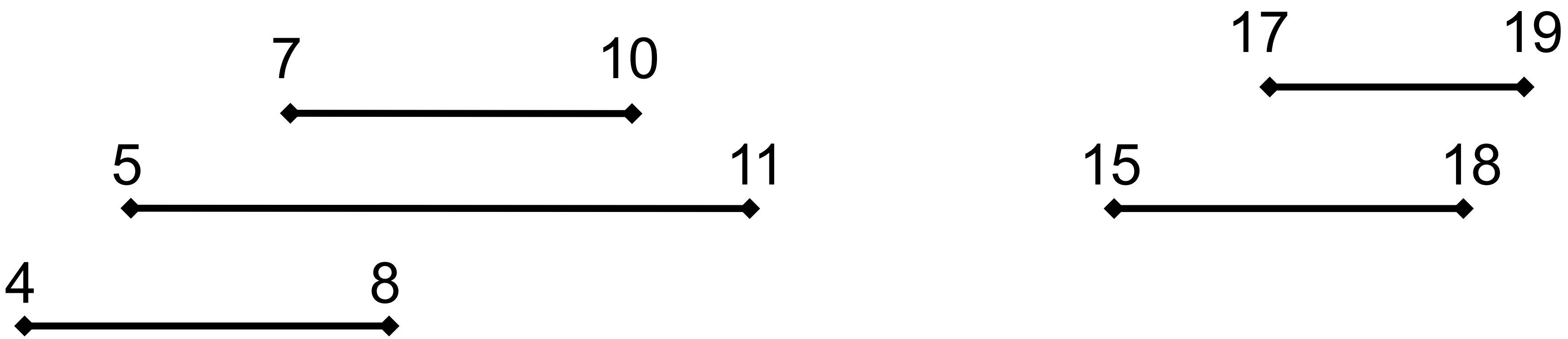
Dynamic data structure: supports new towers.



insert(begin, end)
delete(begin, end)

Cell Tower Coverage

Find a tower that covers my location.

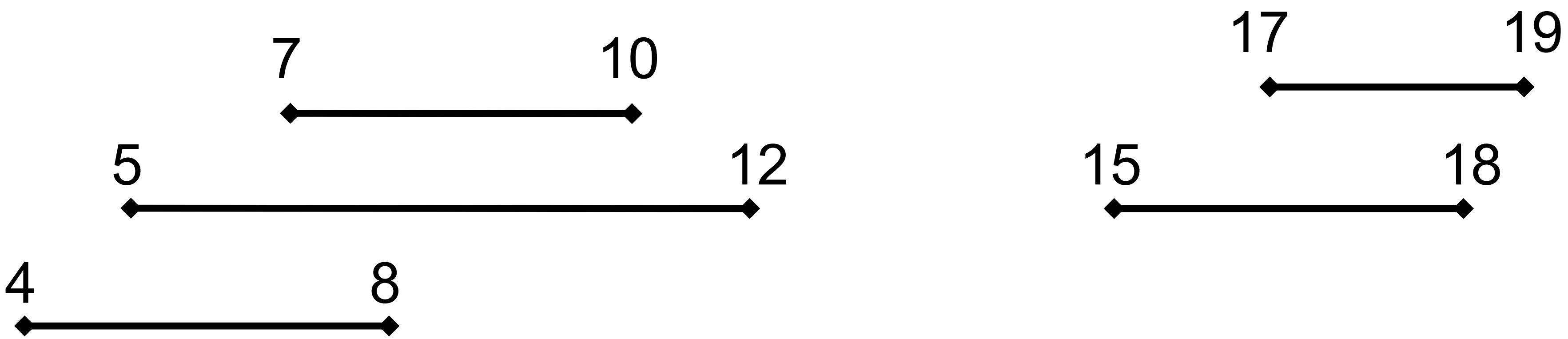


insert(begin, end)
delete(begin, end)

query(x): find an interval that overlaps x.

Cell Tower Coverage

Find a tower that covers my location.



Idea 1: Keep intervals in a list.

Sort by minimum value in interval.

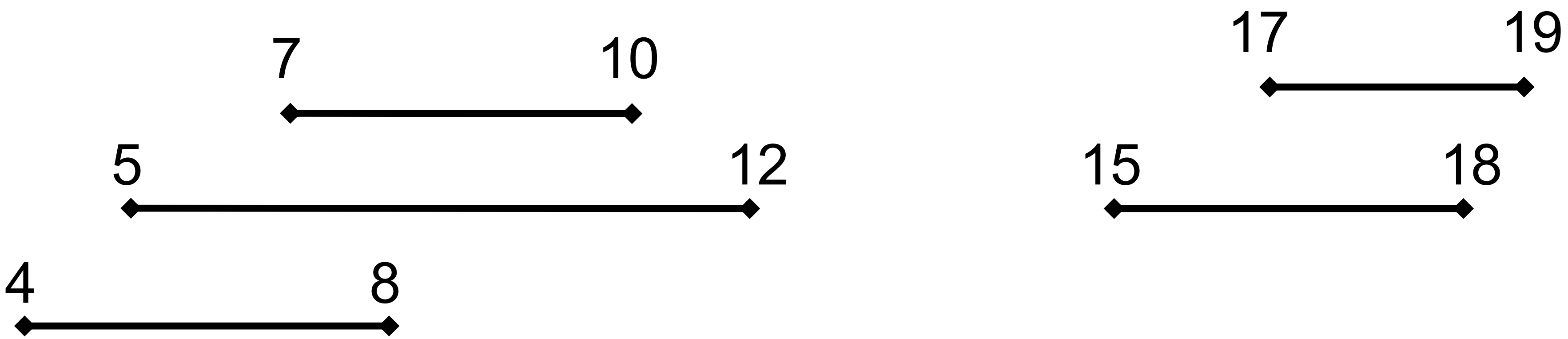
Query: scan entire list.

Does sorting help? Can we binary search?

Cell Tower Coverage

Find a tower that covers my location.

example: query(11)



Idea 1: Keep intervals in a list.

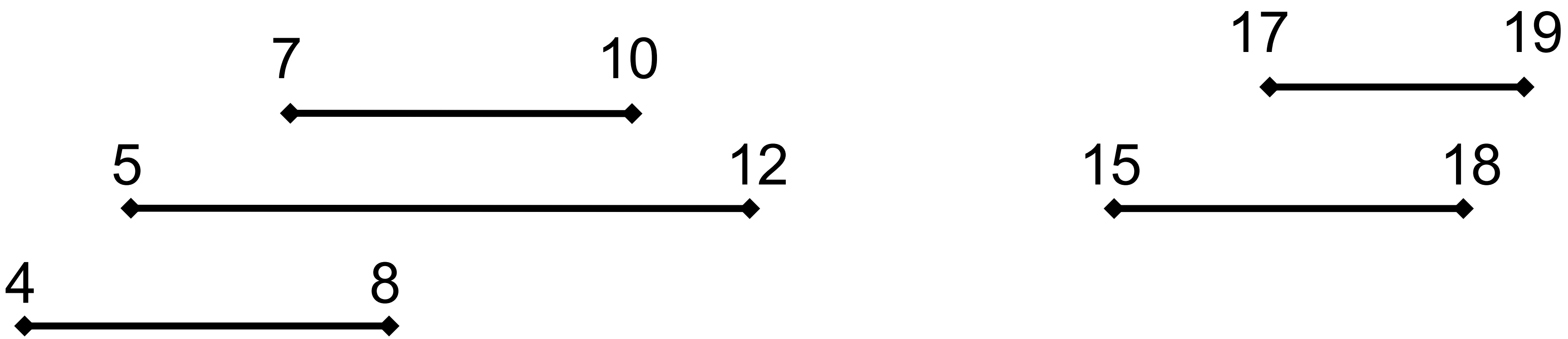
Sort by minimum value in interval.

Query: scan entire list.

Does sorting help? Can we binary search?

Cell Tower Coverage

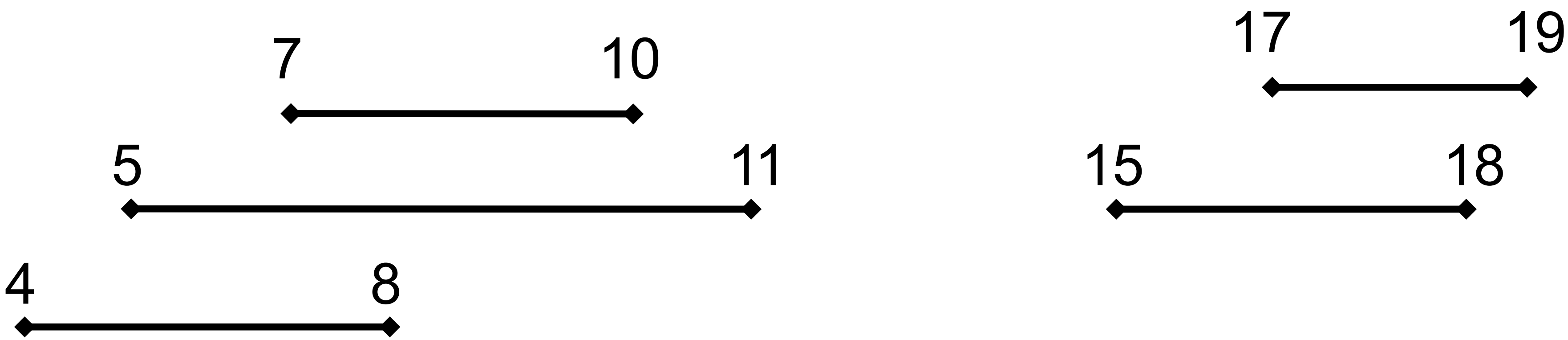
Find a tower that covers my location.



Idea 2: O(1) queries??

Cell Tower Coverage

Find a tower that covers my location.



Idea 2: O(1) queries

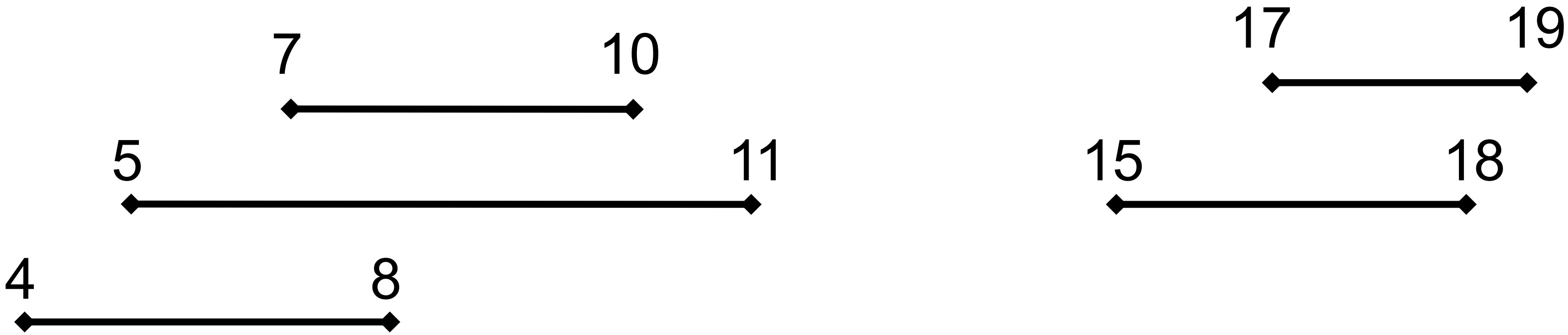
ARCHIPELAGO
is open

			A	**A**	**A**	**A**	**A**	**B**	**B**	**C**				**D**	**D**	**D**	**D**	**E**	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Problems??

Cell Tower Coverage

Find a tower that covers my location.



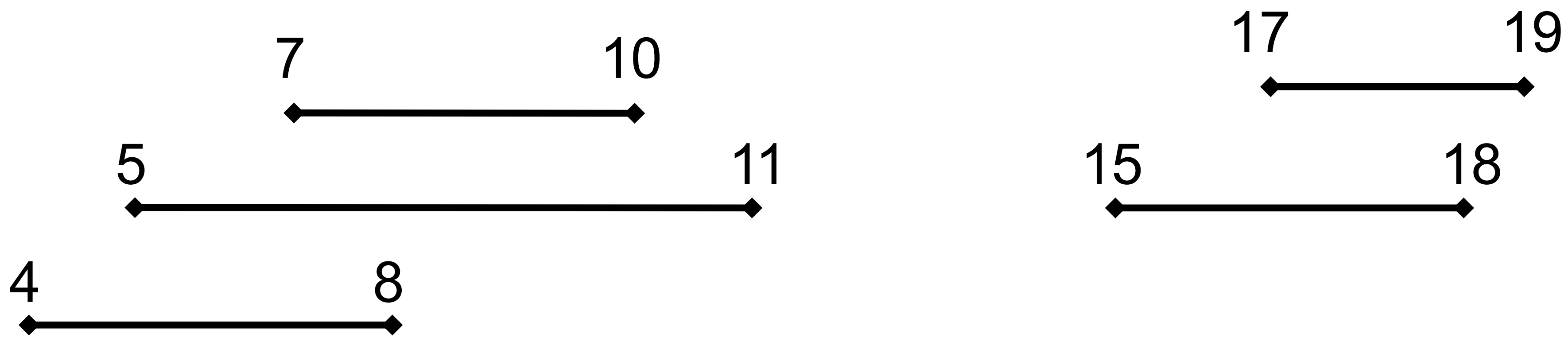
Idea 2: O(1) queries

			A	**A**	**A**	**A**	**A**	**B**	**B**	**C**				**D**	**D**	**D**	**D**	**E**	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Space usage, requires discrete integers, potentially expensive to update.

Cell Tower Coverage

Find a tower that covers my location.

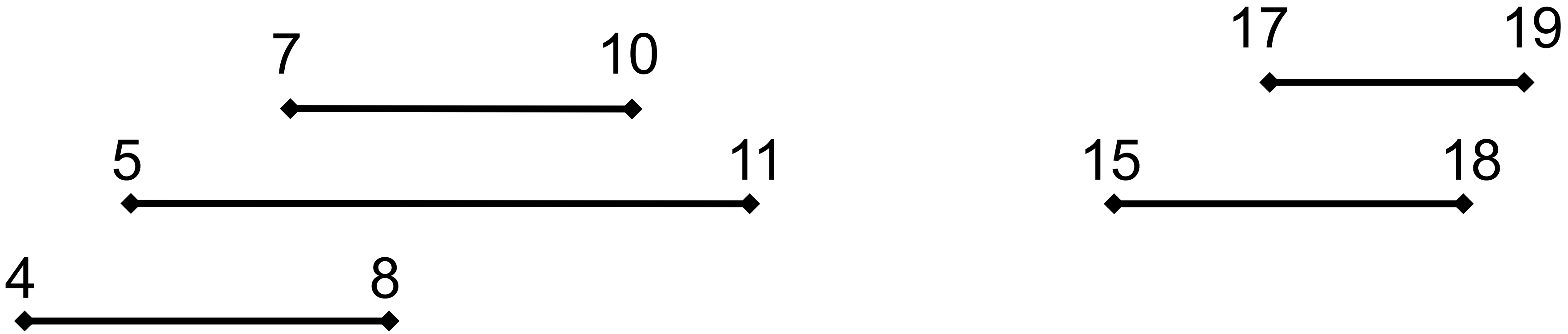


Not ideal solution:

- Space depends on the values stored.
- Time depends on the values stored.

Cell Tower Coverage

Find a tower that covers my location.



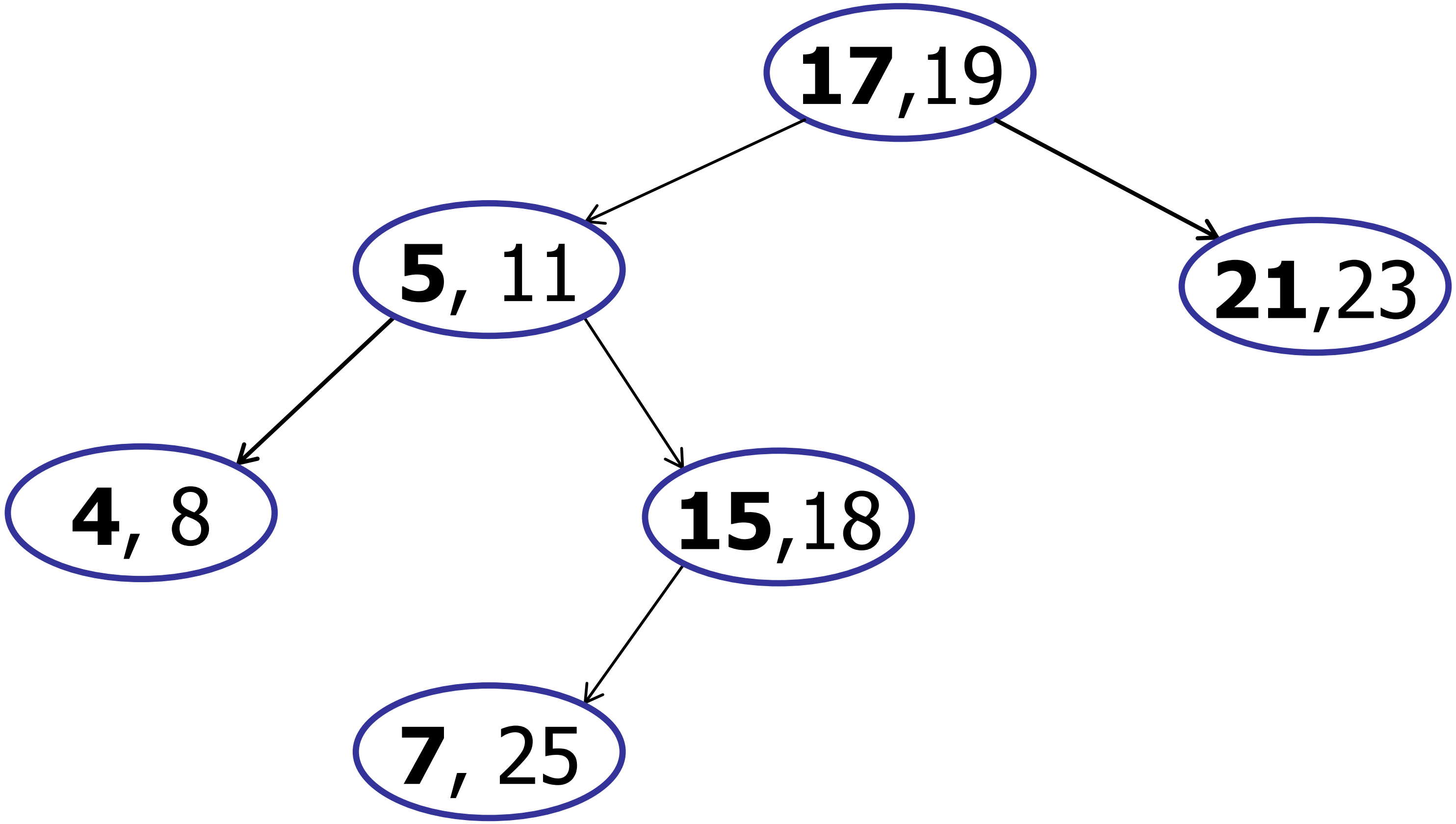
Goal:

- Solutions where space is linear (or near linear) in the number of things stored (i.e., intervals).
- Operations are logarithmic in # of things stored.

Idea 3: Interval Trees

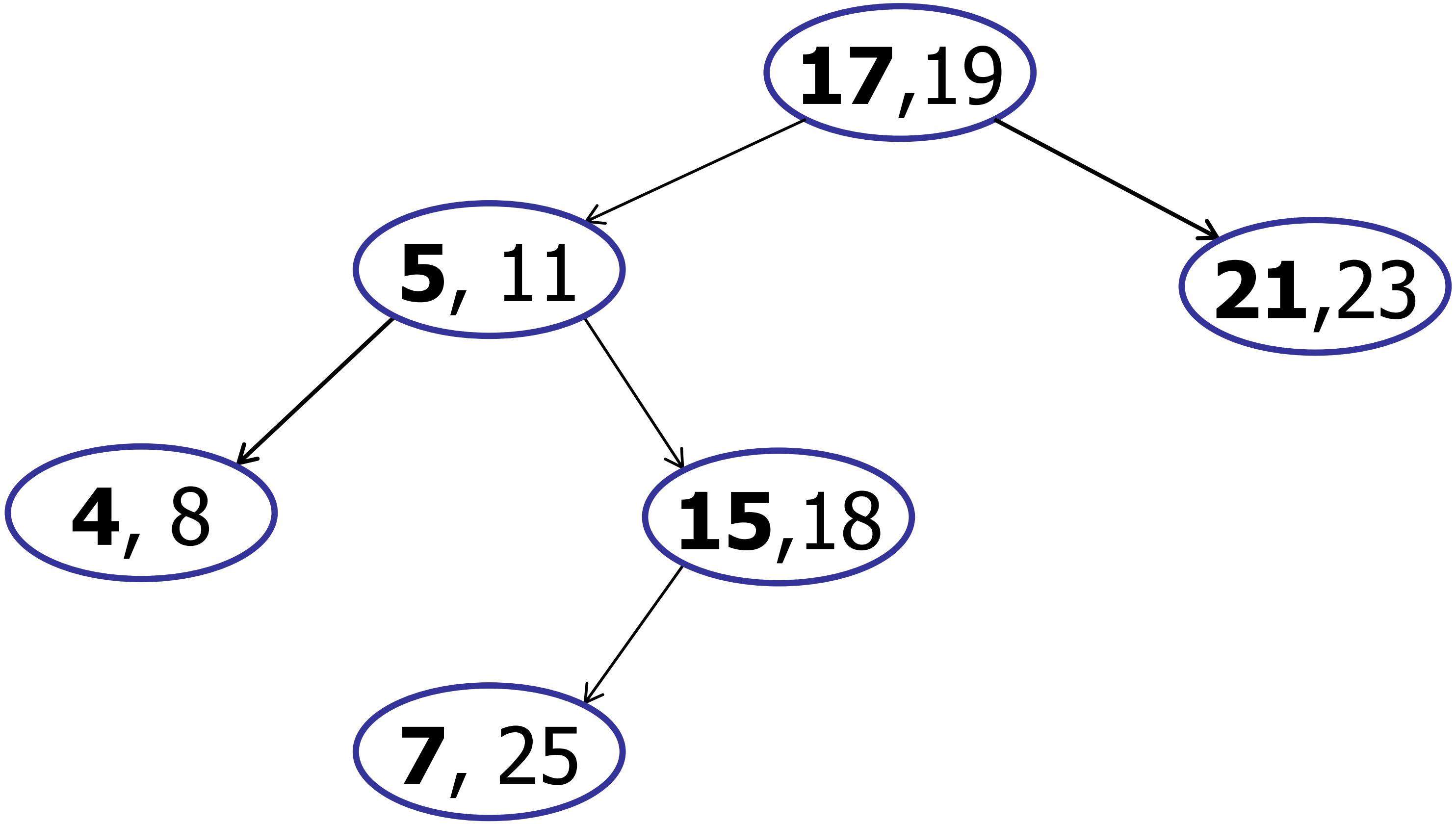
Interval Trees

Each node is an interval



Interval Trees

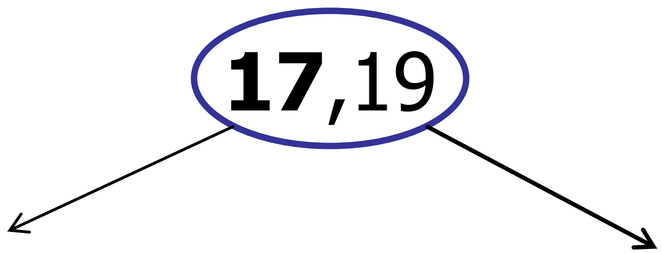
Sorted by left endpoint



Important: always specify what your tree is sorted by!

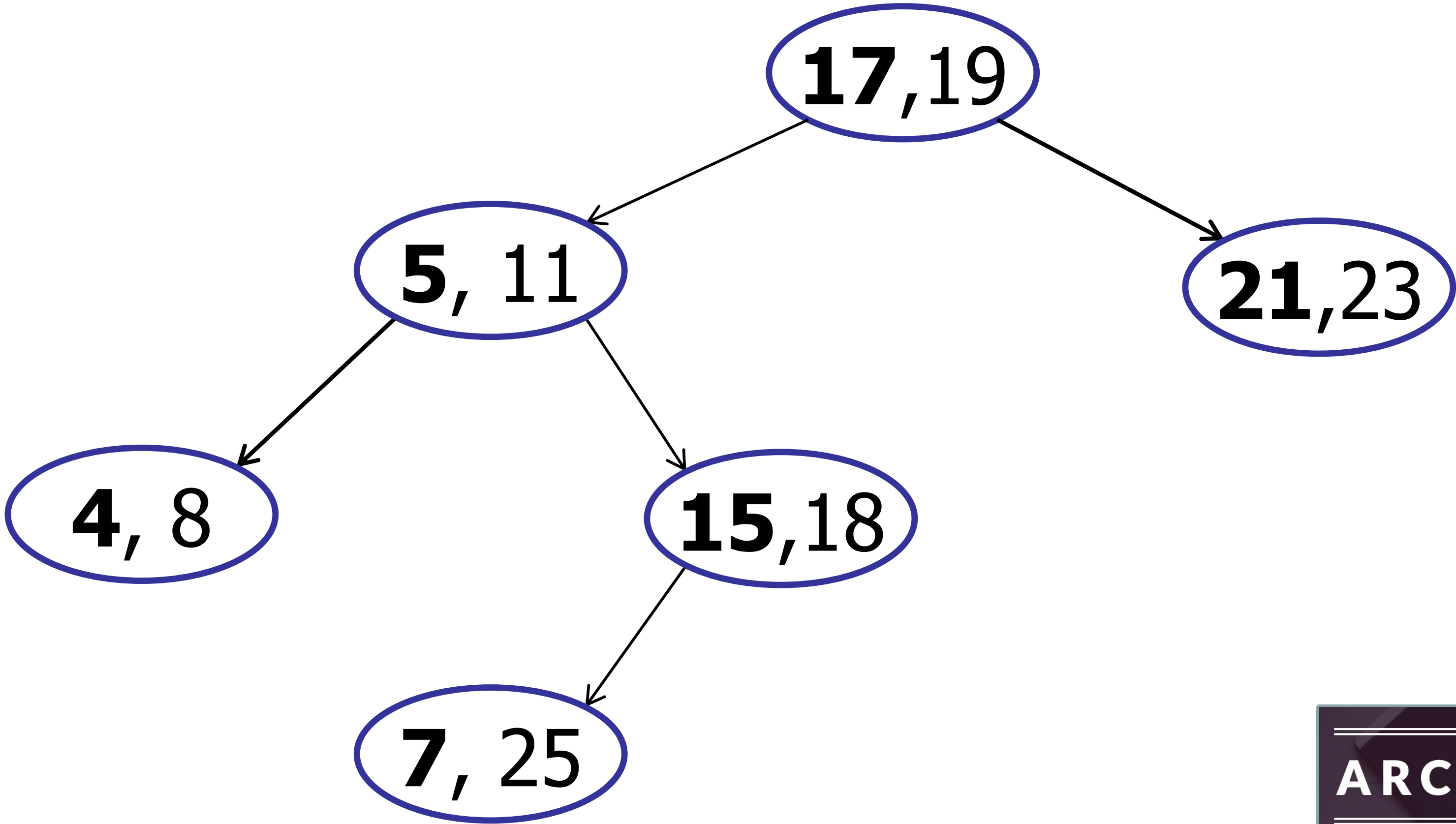
Interval Trees

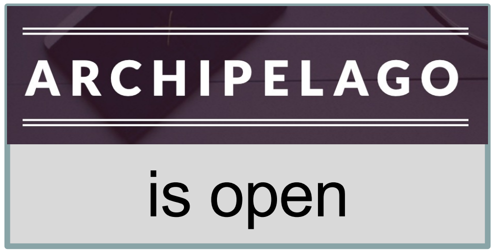
search-interval(25) = ?



Interval Trees

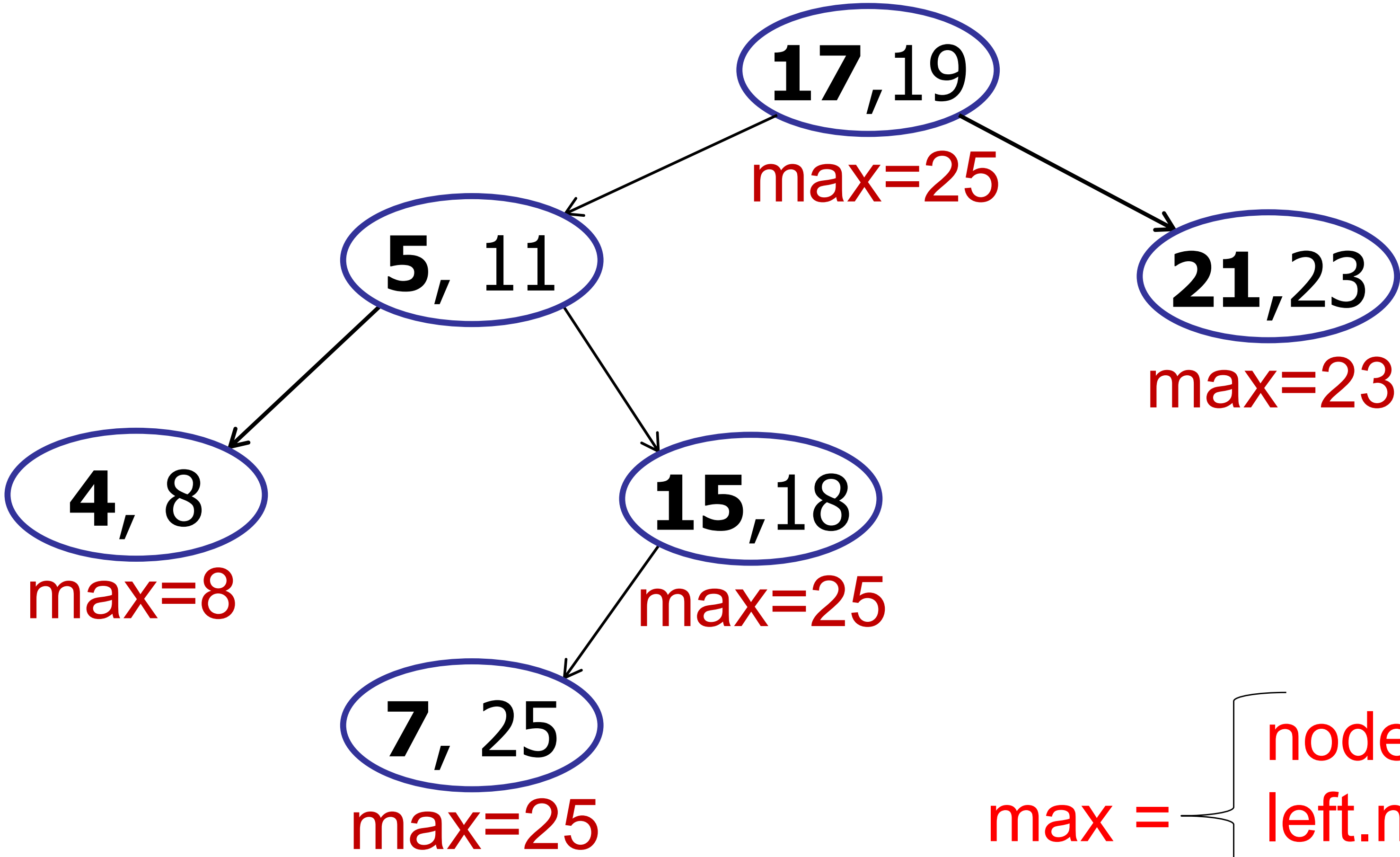
Augment: ??





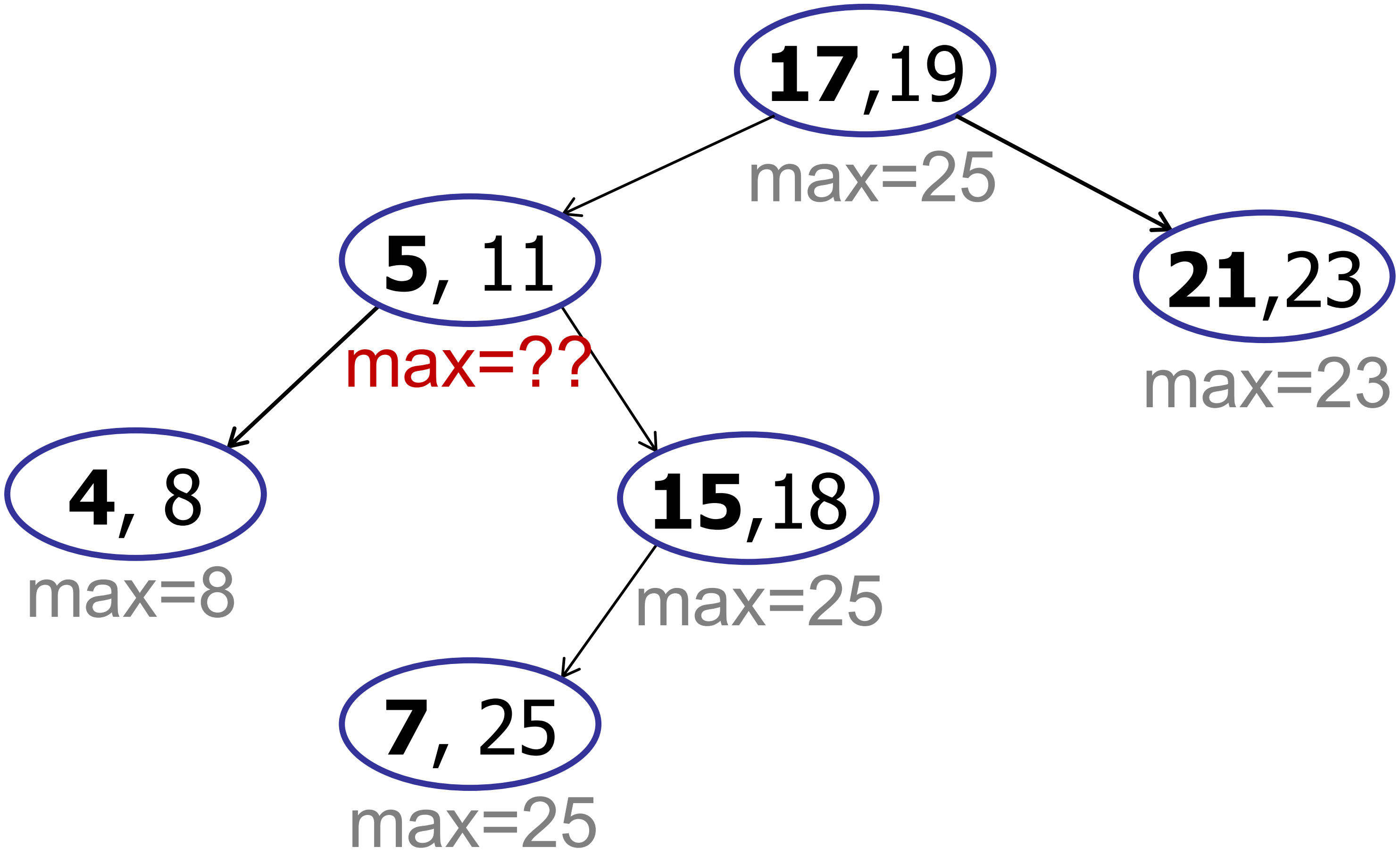
Interval Trees

Augment: maximum endpoint in subtree



$$\text{max} = \begin{cases} \text{node.right} \\ \text{left.max} \\ \text{right.max} \end{cases}$$

max=??

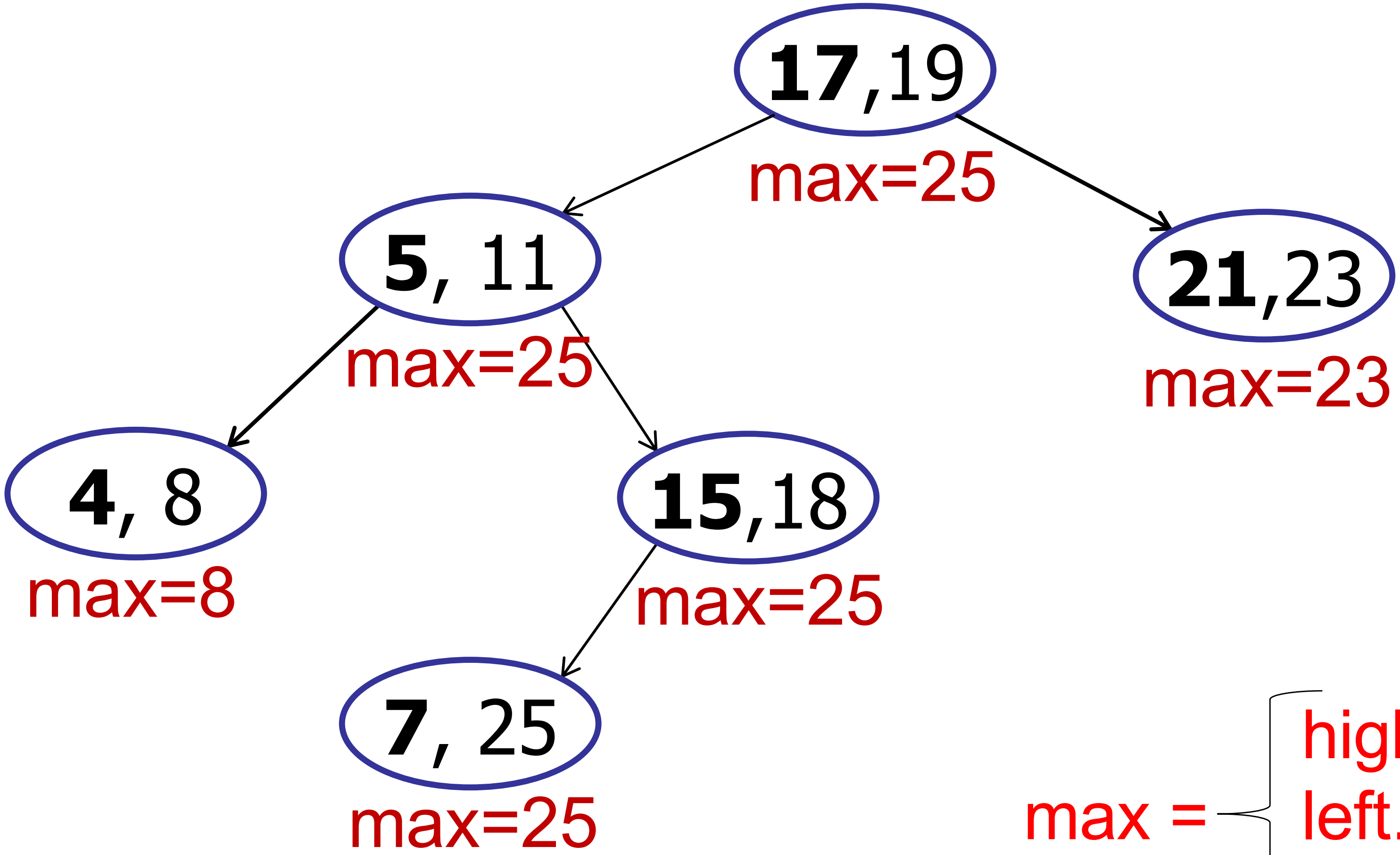


1. 5
2. 8
3. 11
4. 18
5. 25
6. 19

ARCHIPELAGO
is open

Interval Trees

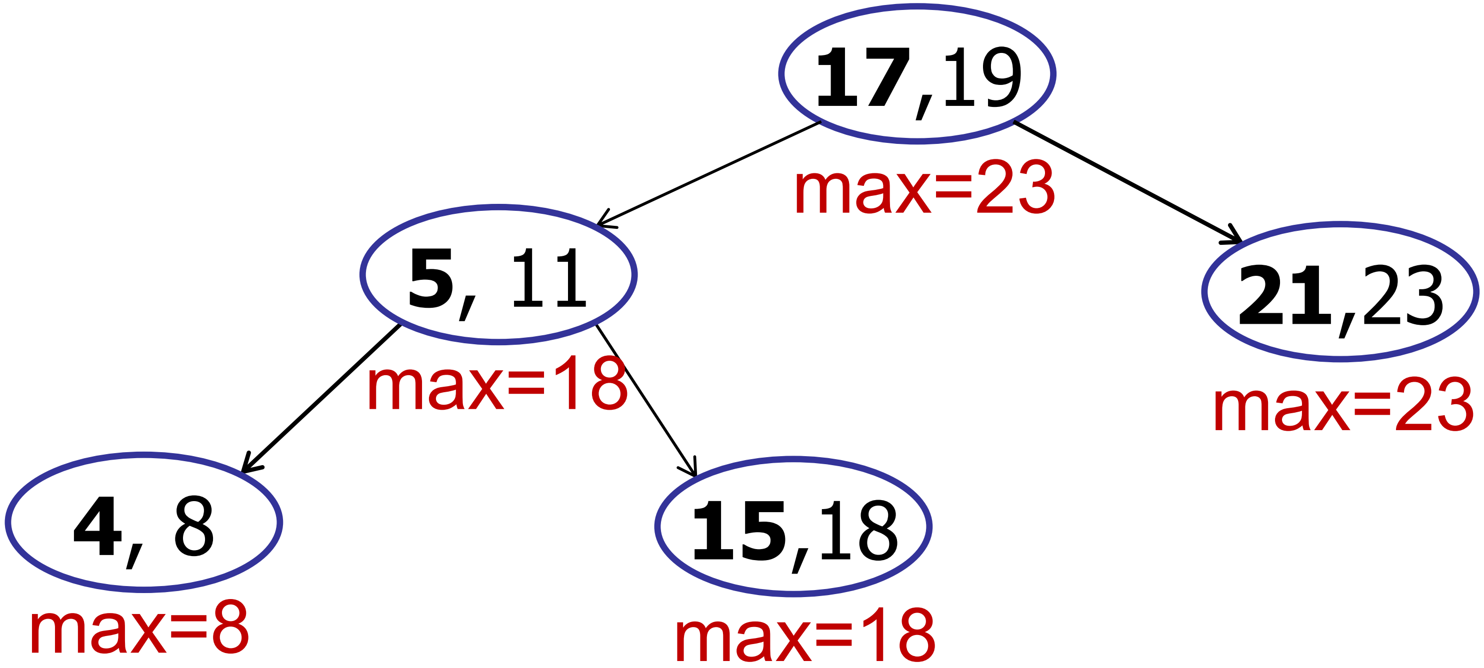
Augment: maximum endpoint in subtree



$$\text{max} = \begin{cases} \text{high} \\ \text{left.max} \\ \text{right.max} \end{cases}$$

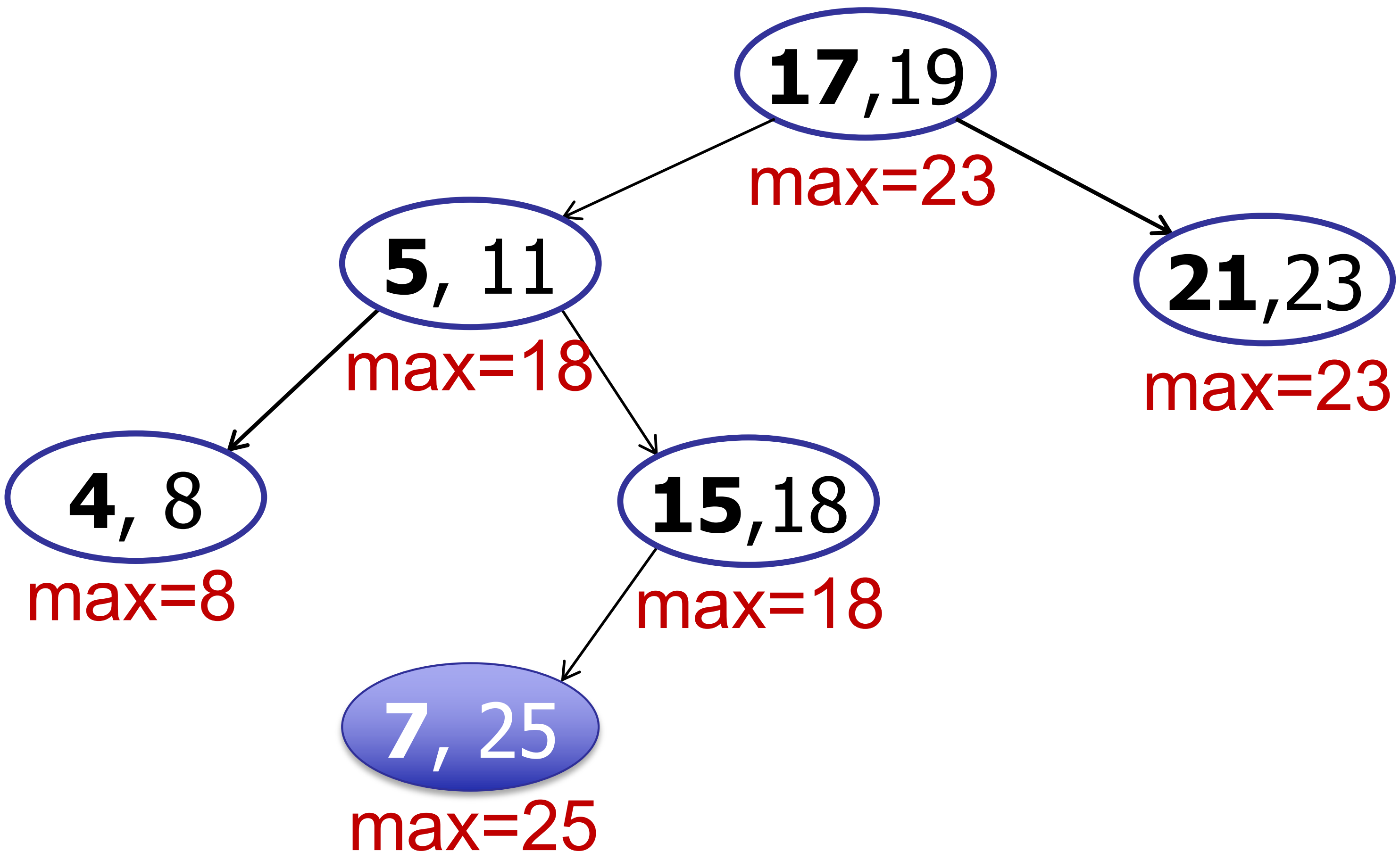
Interval Trees

Insertion: *example* – **insert(7, 25)**



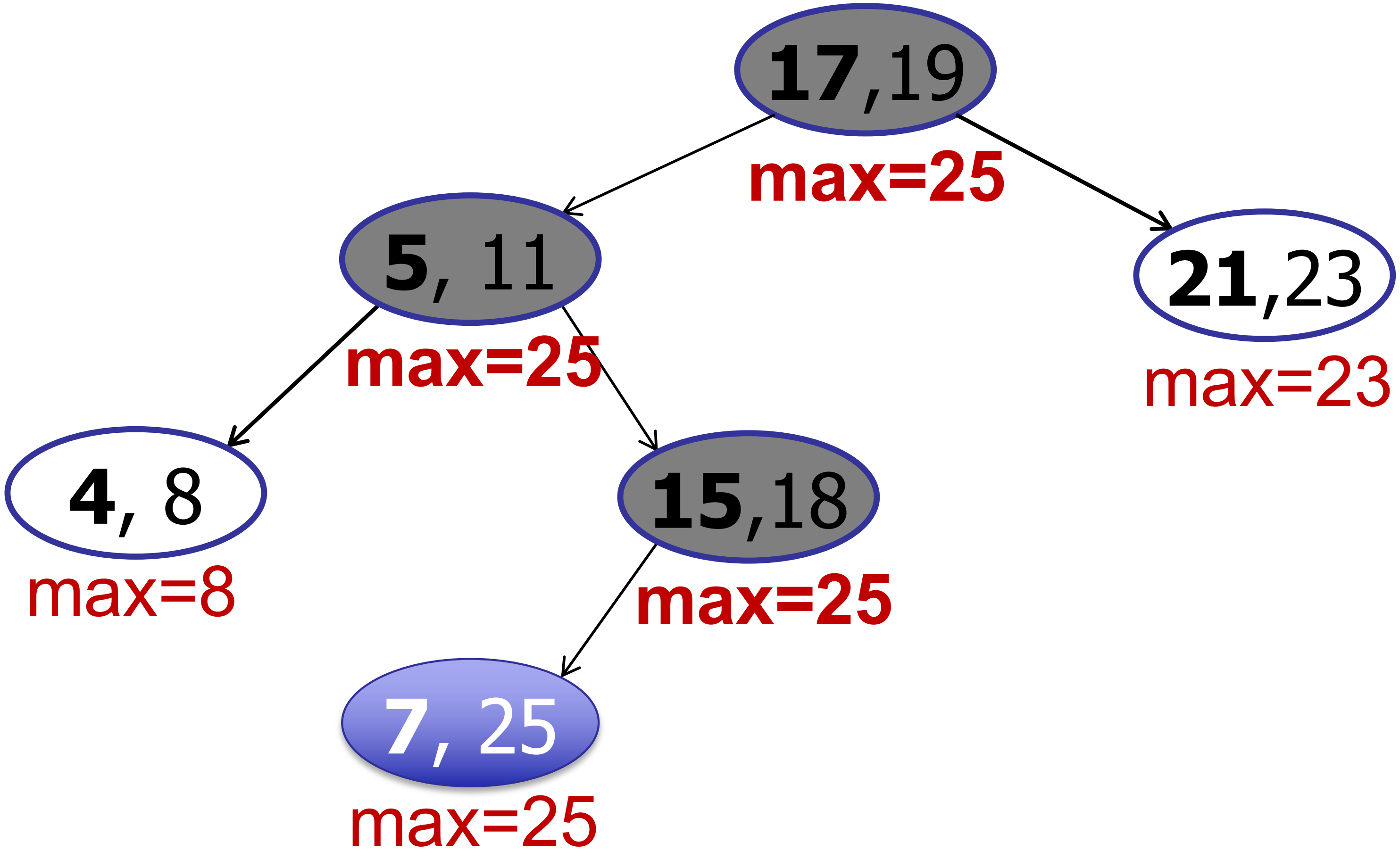
Interval Trees

Insertion: *example* – **insert(7, 25)**



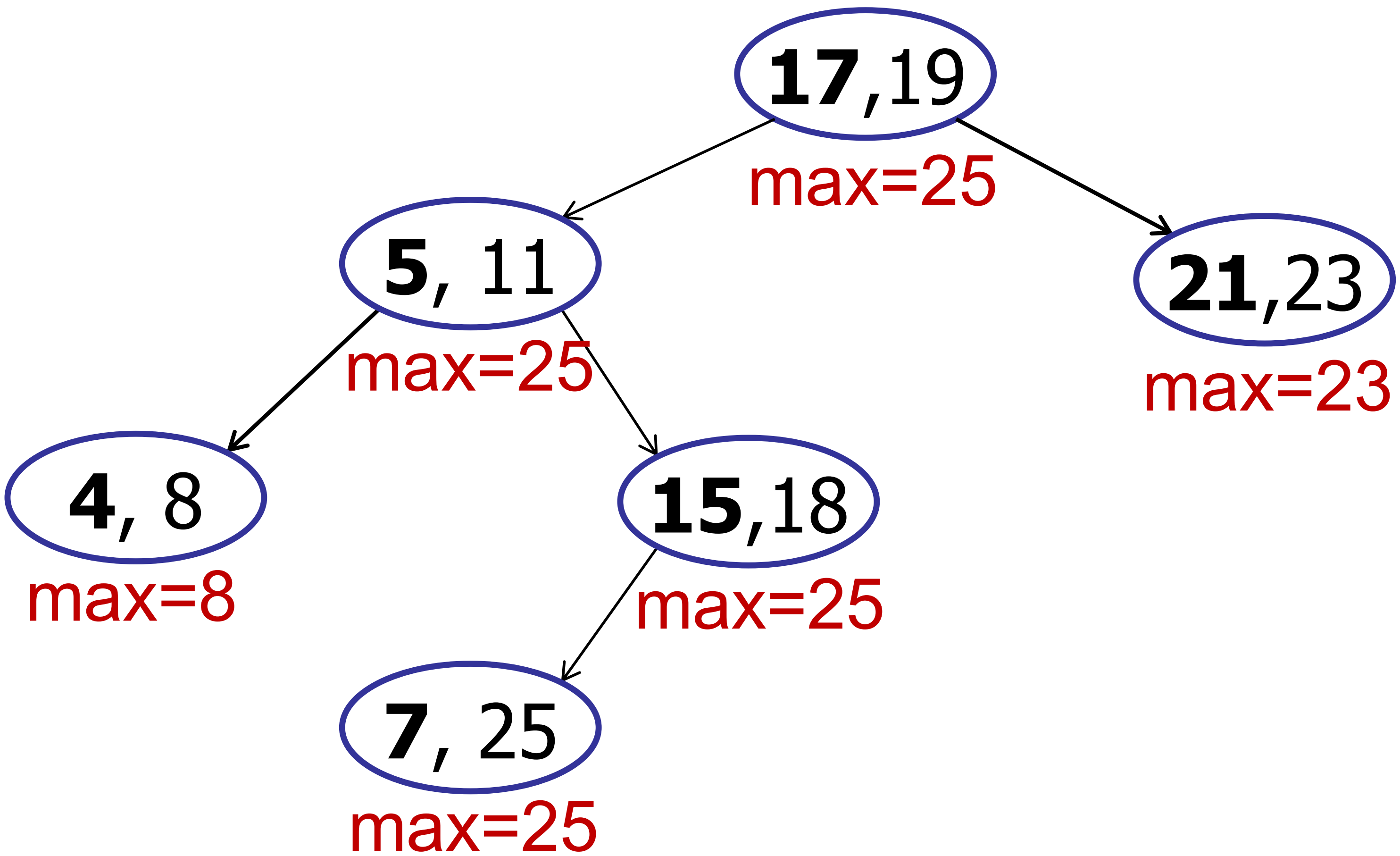
Interval Trees

Insertion: *example* – **insert(7, 25)**



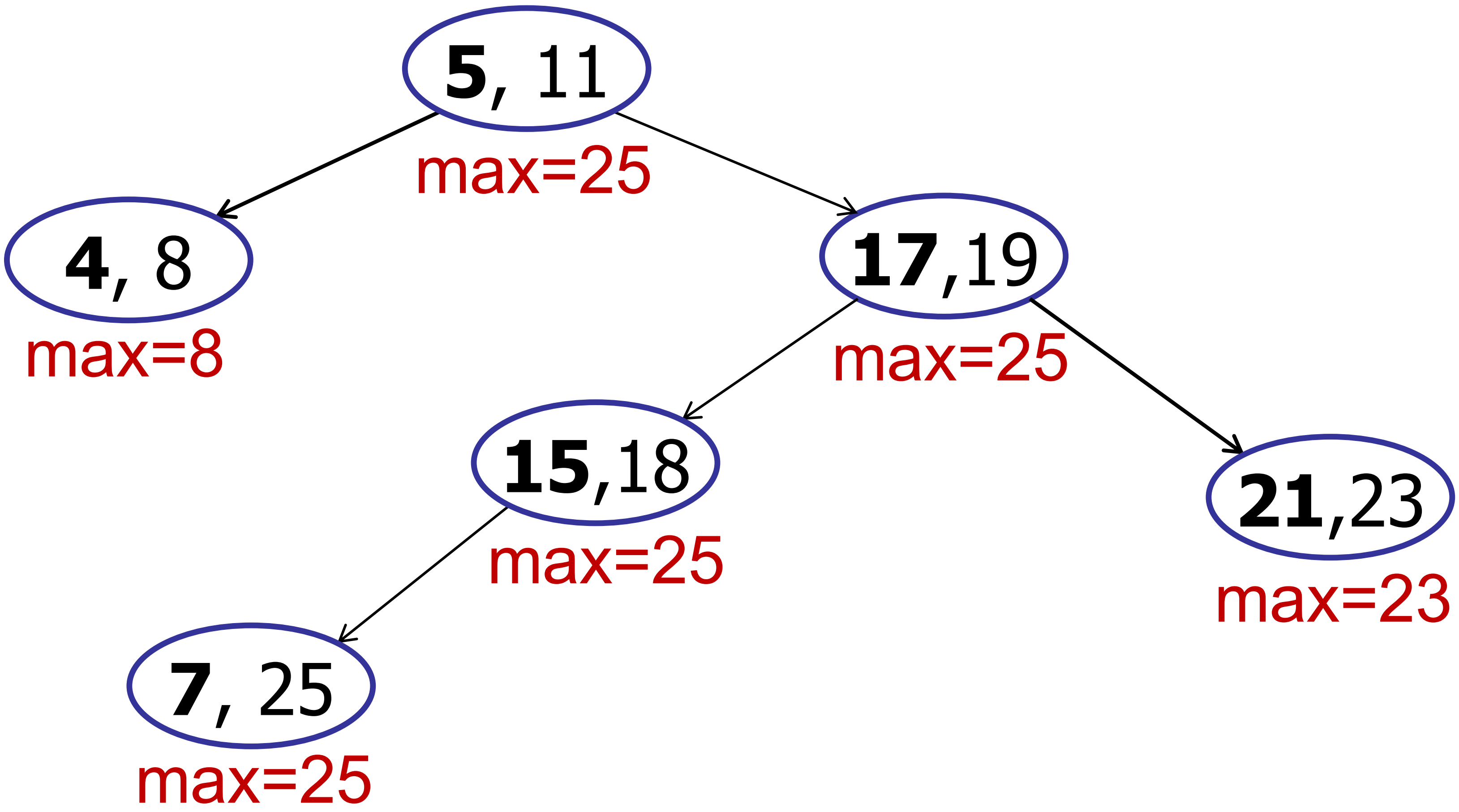
Interval Trees

Insertion: *out-of-balance*



Interval Trees

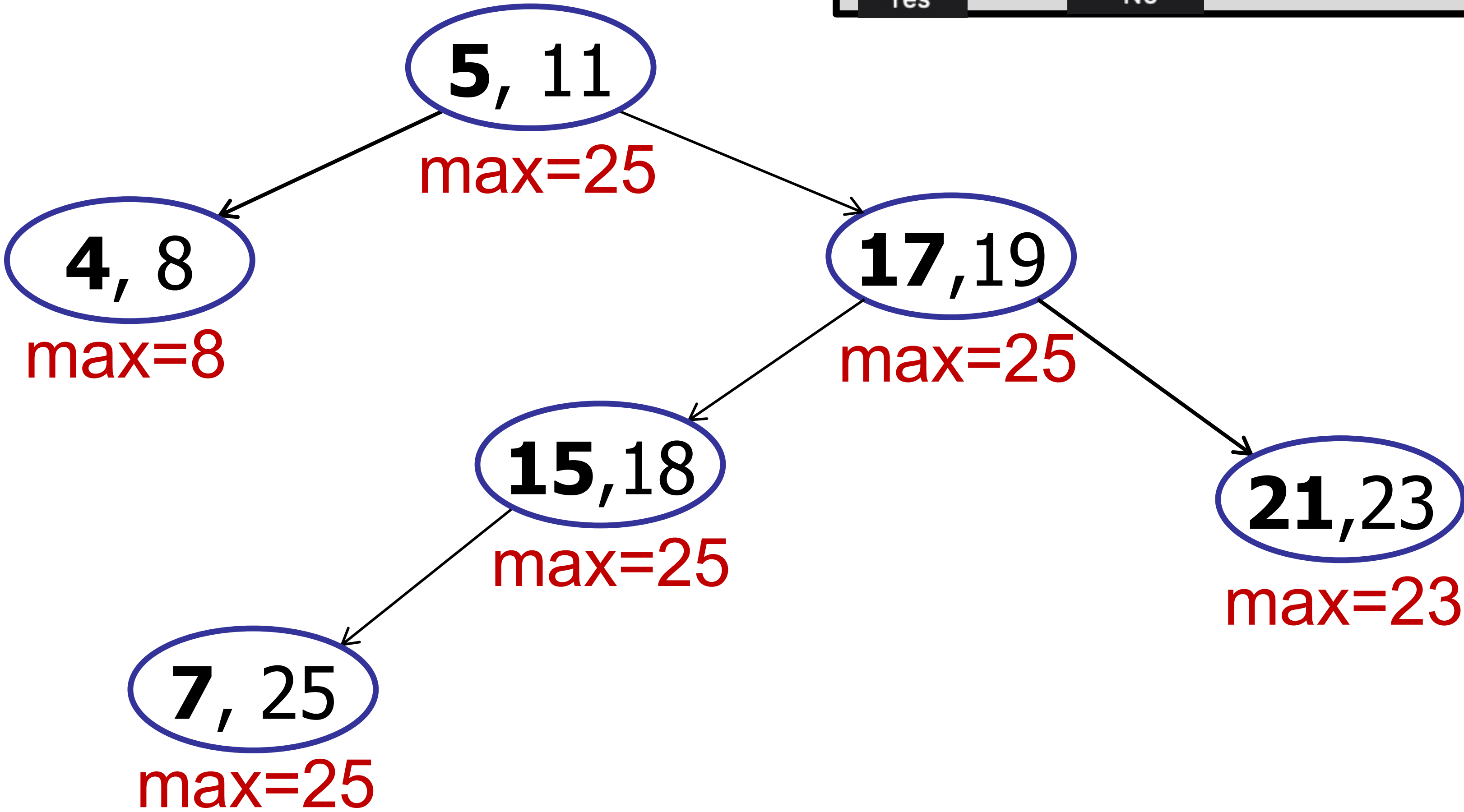
Insertion: right-rotate (17, 19)



Interval Trees

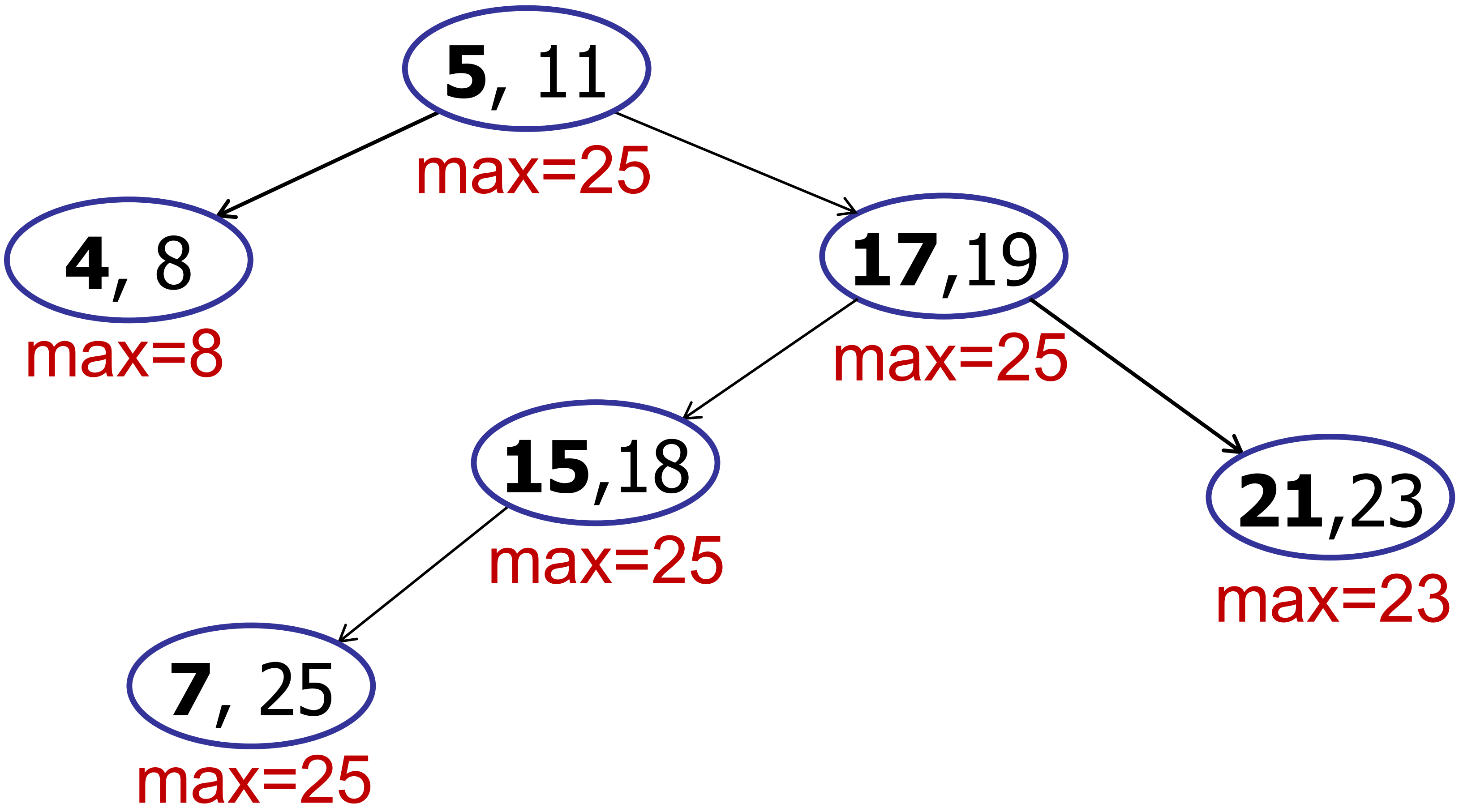
Insertion: right-rotate (17, 19)

Is the tree now balanced?



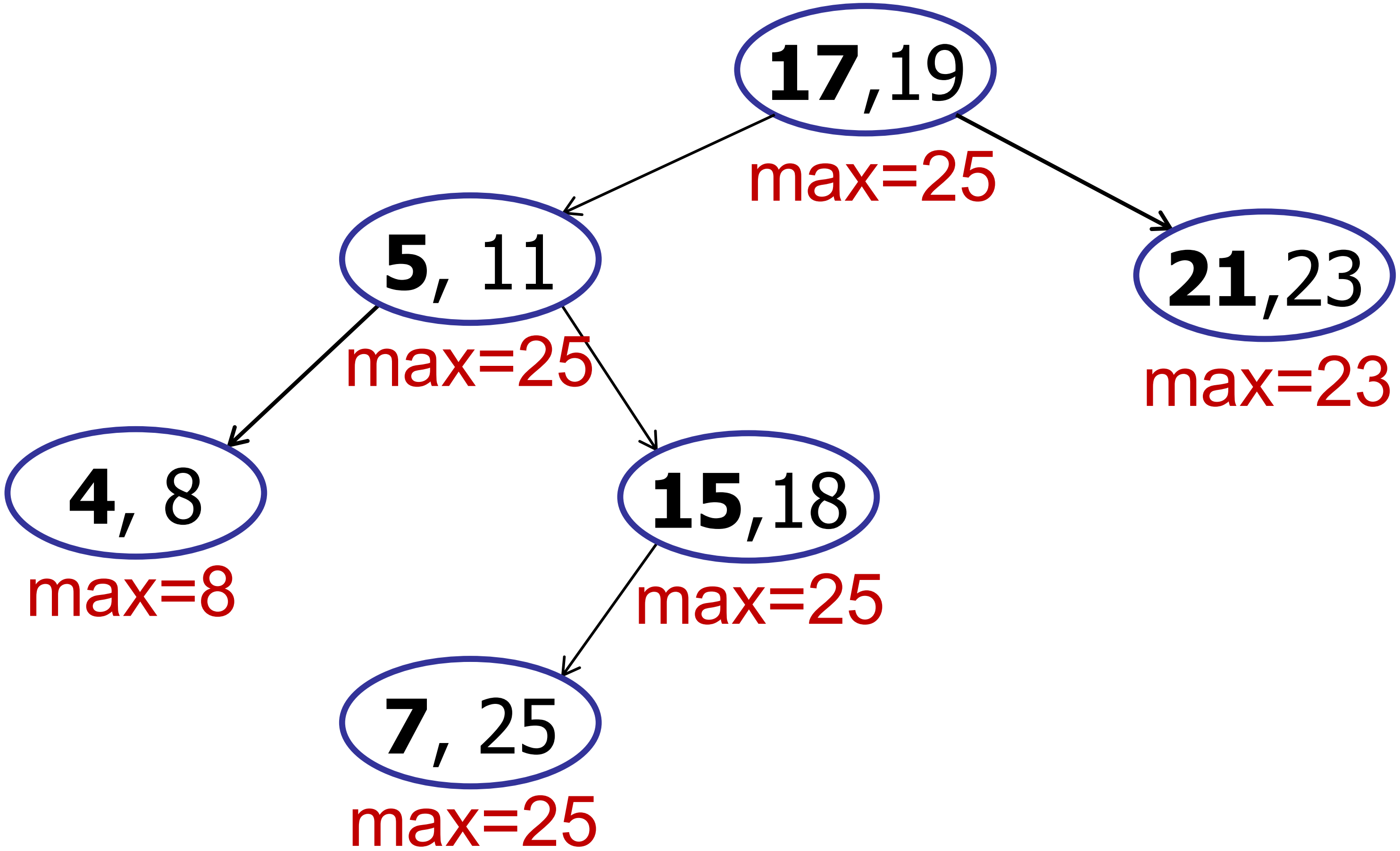
Interval Trees

Insertion: *right-rotate (17, 19),* OOPS!



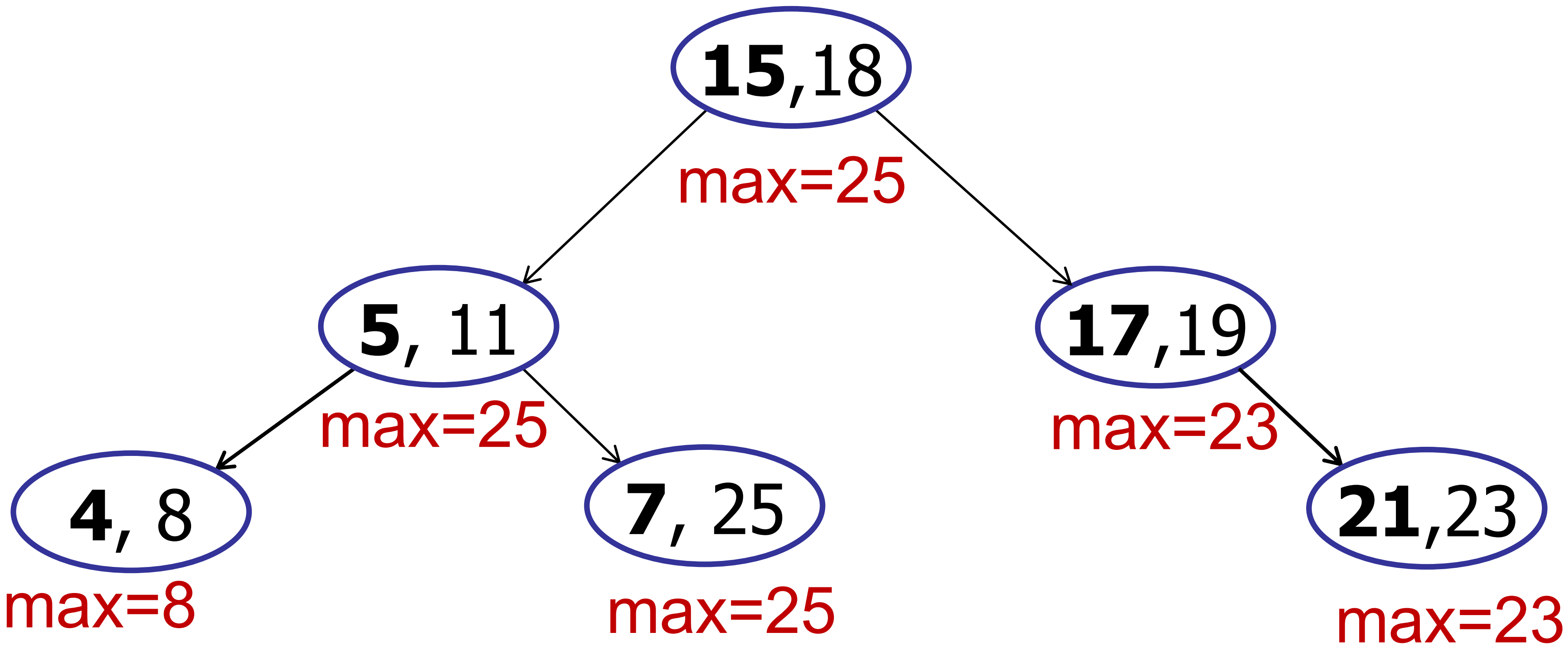
Interval Trees

Insertion: *out-of-balance*



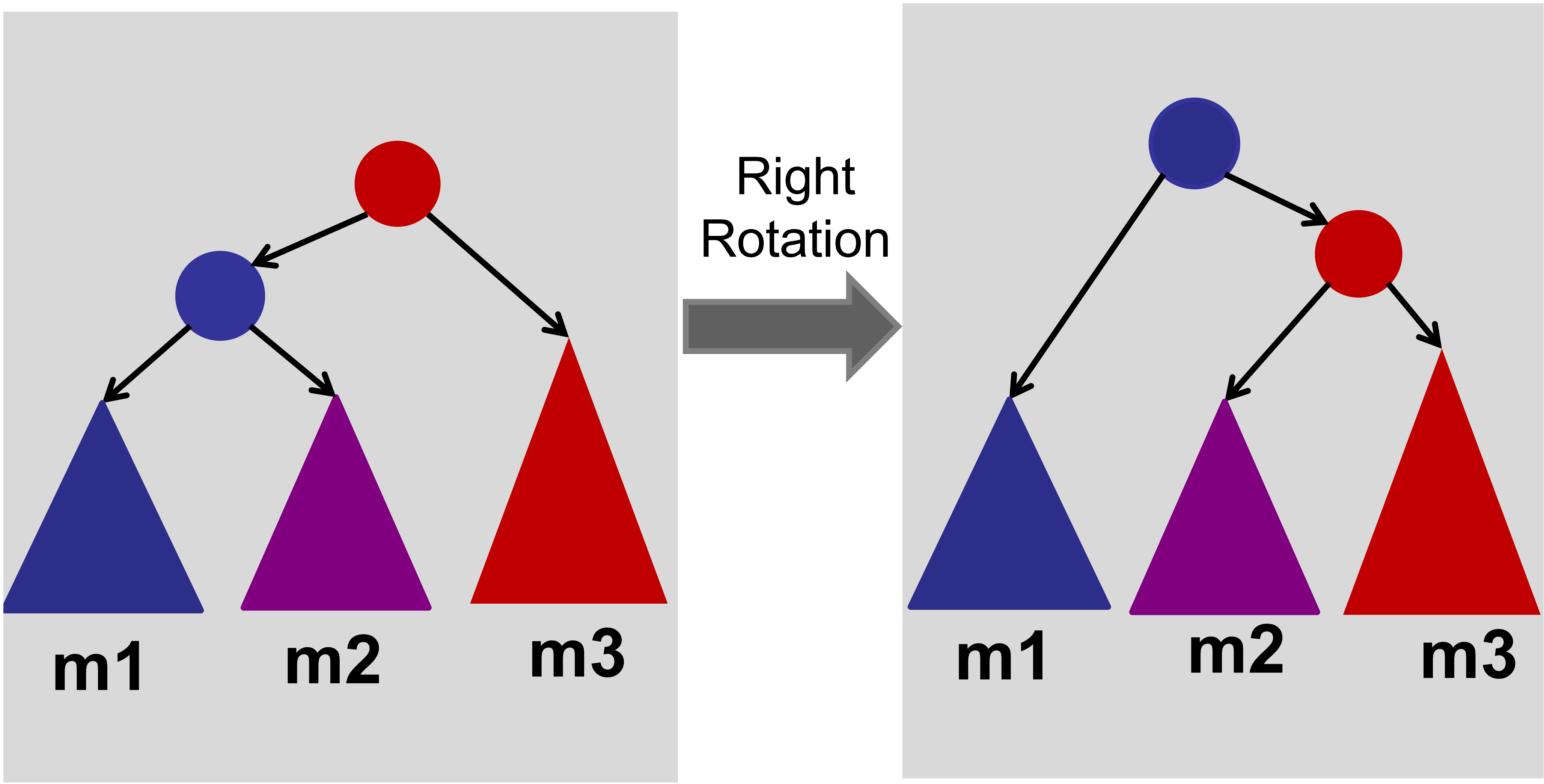
Interval Trees

Insertion: left-rotate, right-rotate



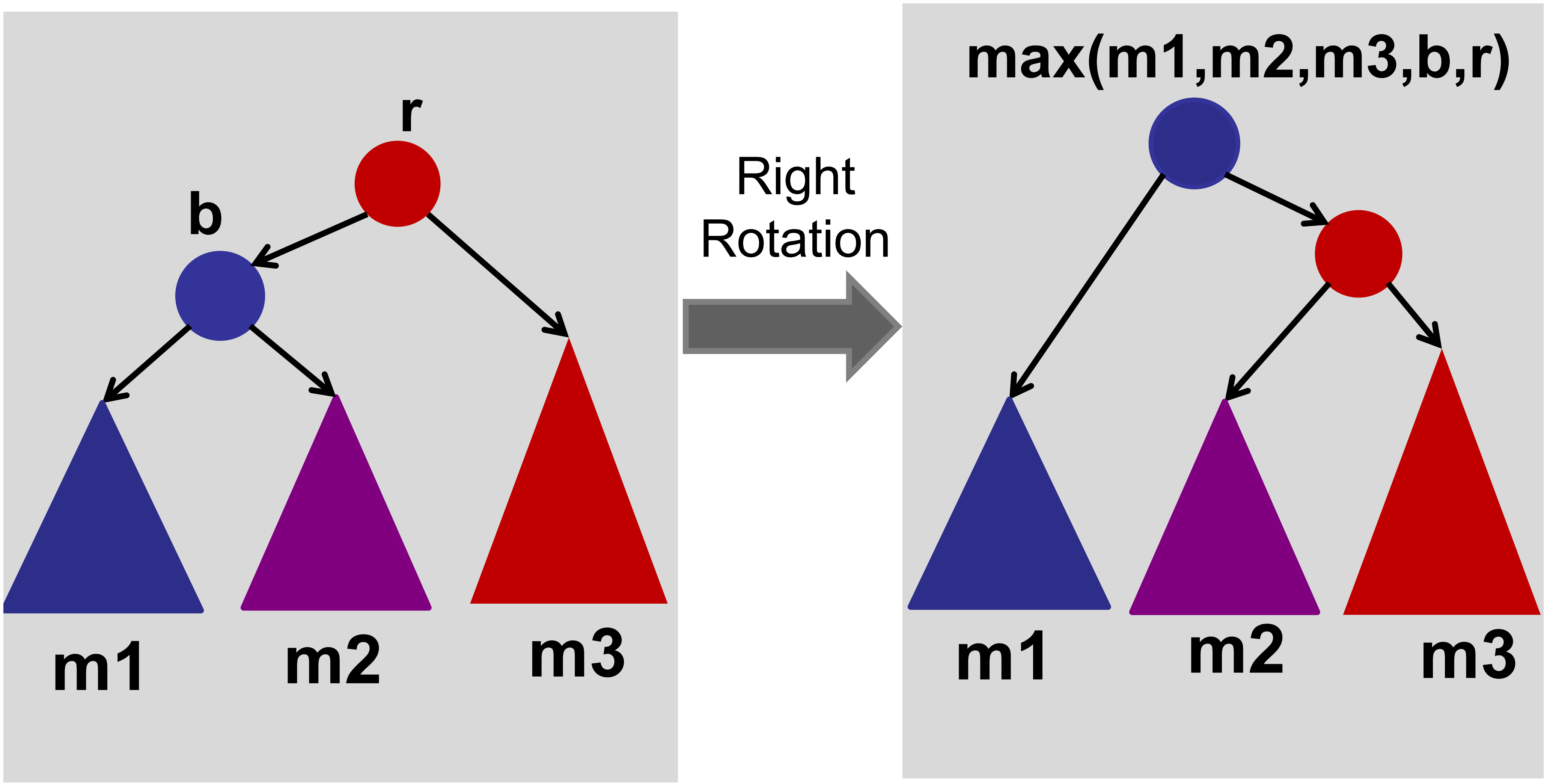
Interval Trees

Maintain MAX during rotations:



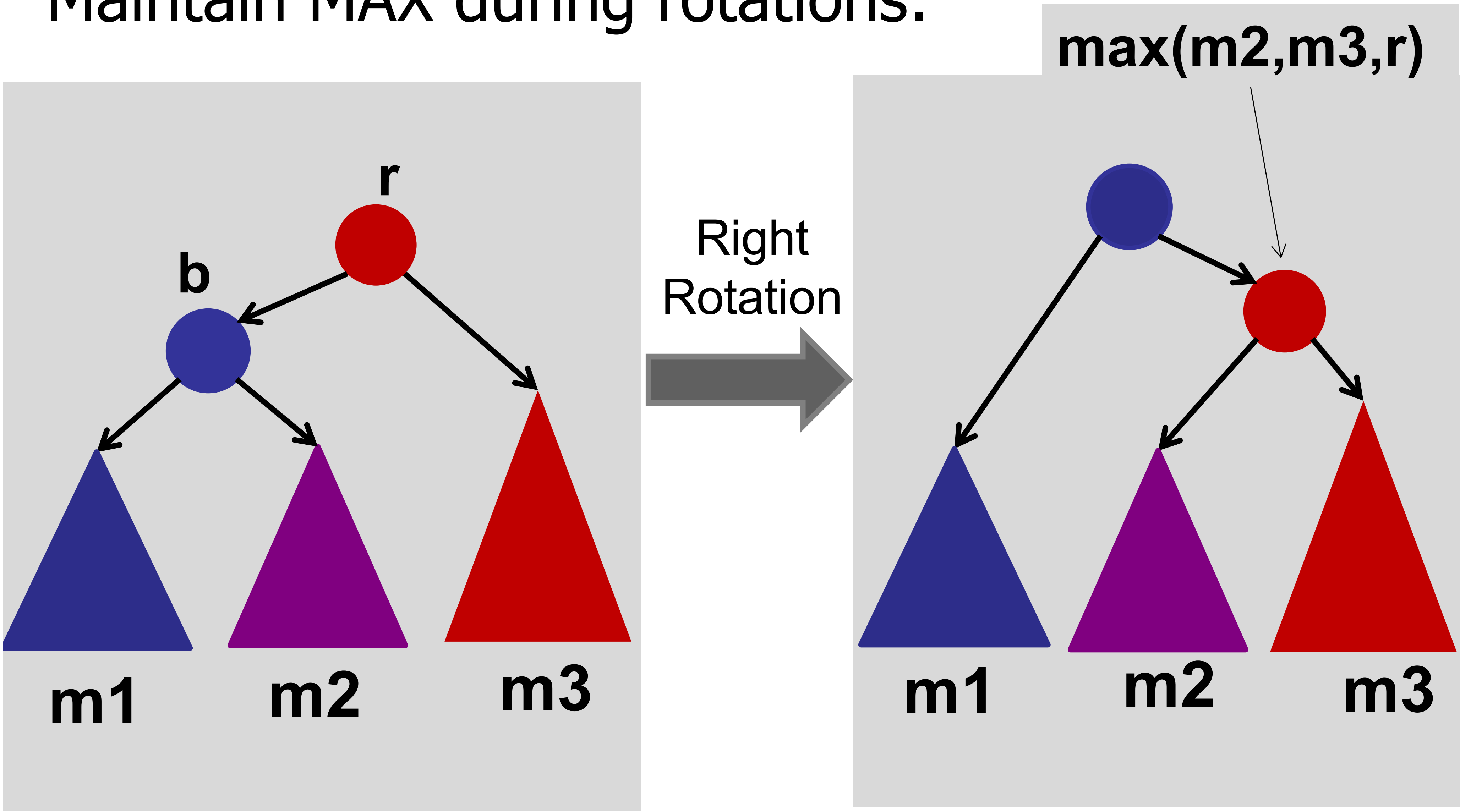
Interval Trees

Maintain MAX during rotations:



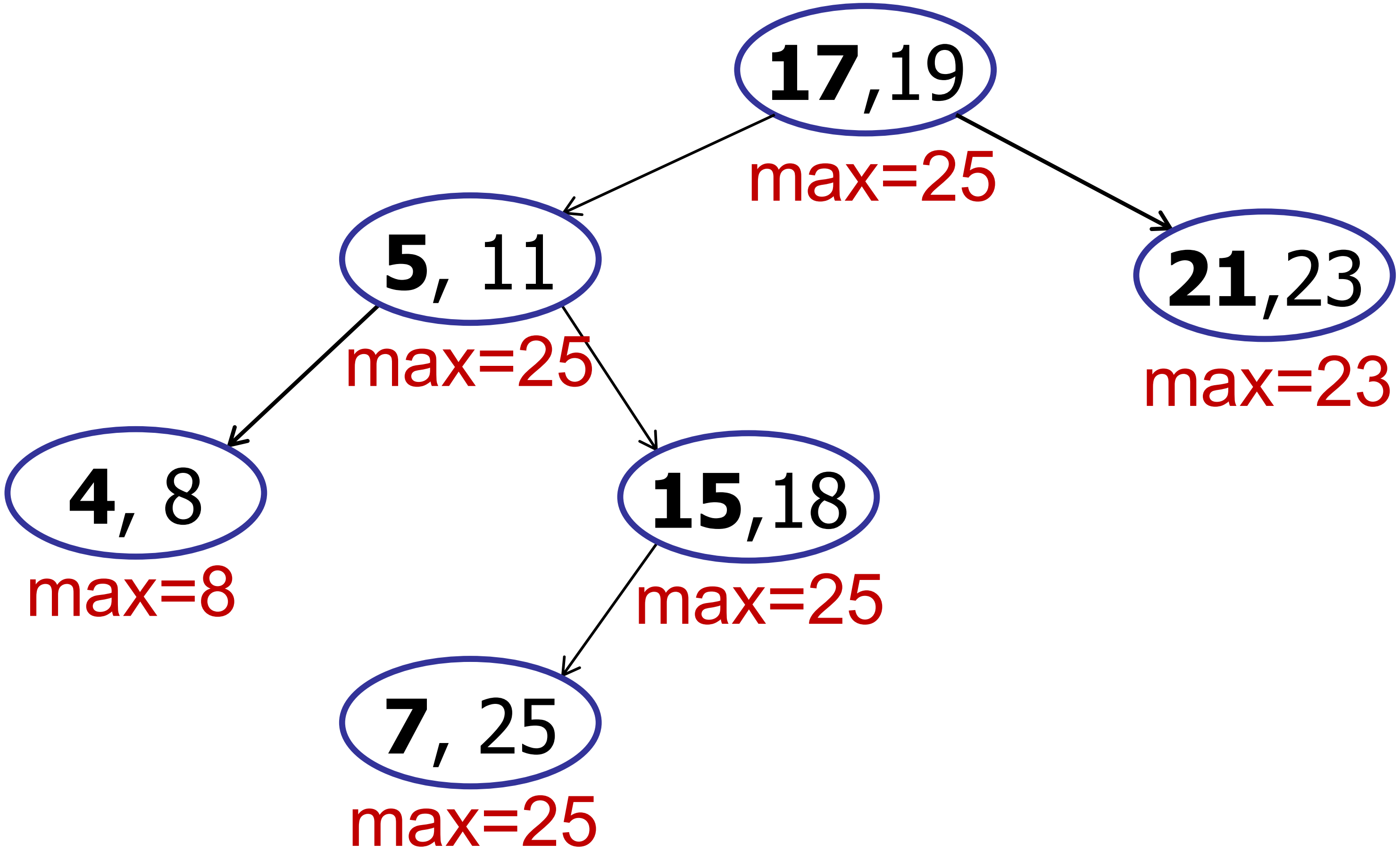
Interval Trees

Maintain MAX during rotations:



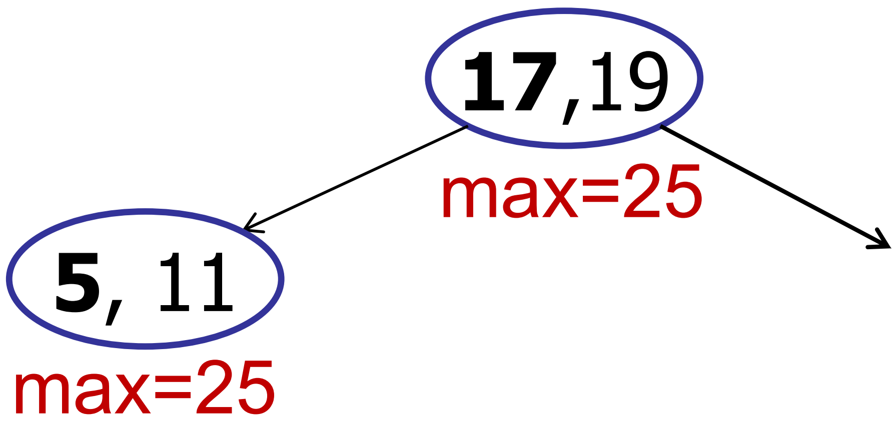
Interval Trees

Searching: interval-search(22)



Interval Trees

Searching: interval-search(22)



It is possible that 22 is covered in the left subtree.

Do we know *for sure* that going left will work?

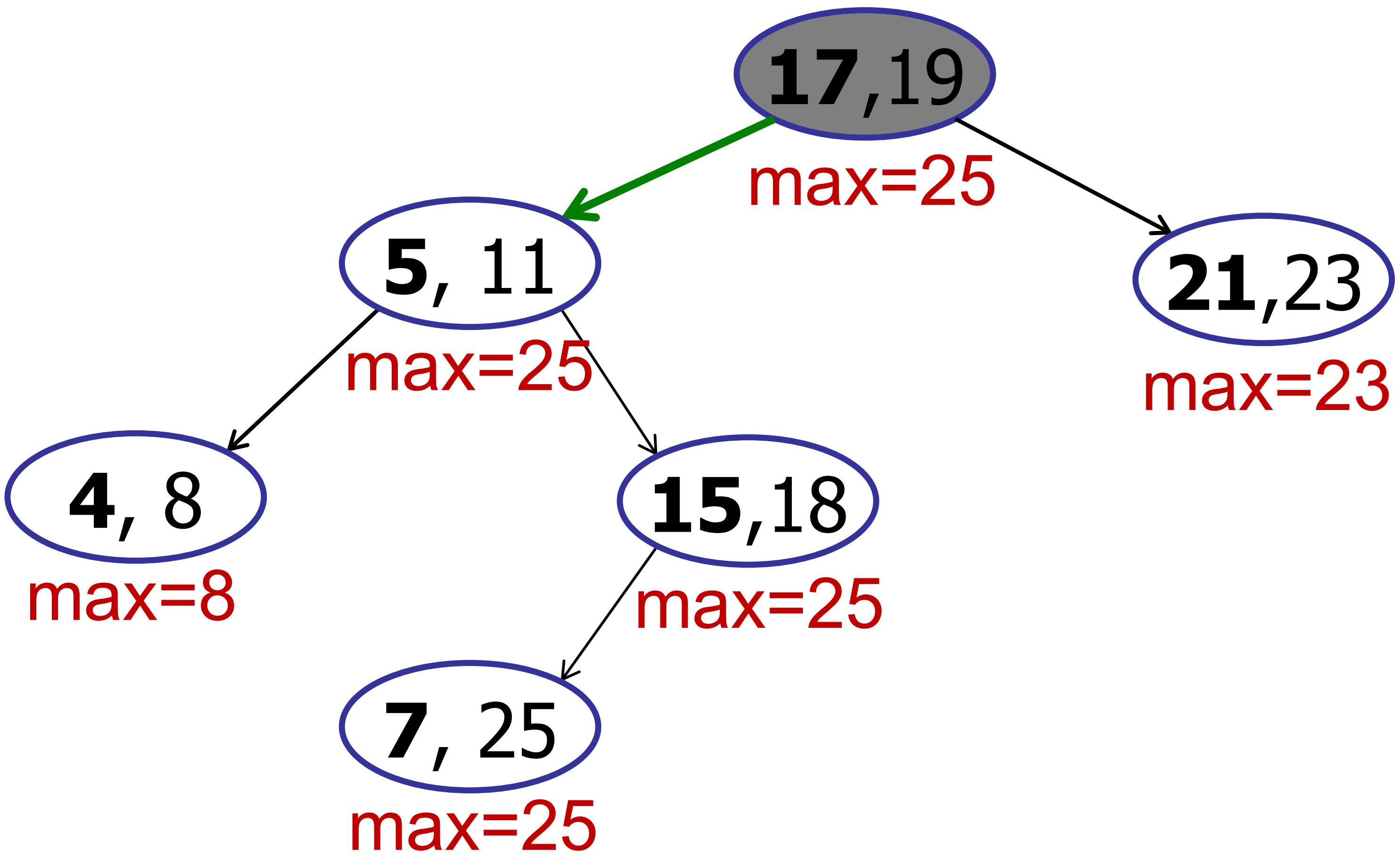
Interval Trees

interval-search(x) : find interval containing x

```
interval-search(x)
    c = root;
    while (c != null and x is not in c.interval) do
        if (c.left == null) then
            c = c.right;
        else if  (x > c.left.max)  then
            c = c.right;
        else c = c.left;
    return c.interval;
```

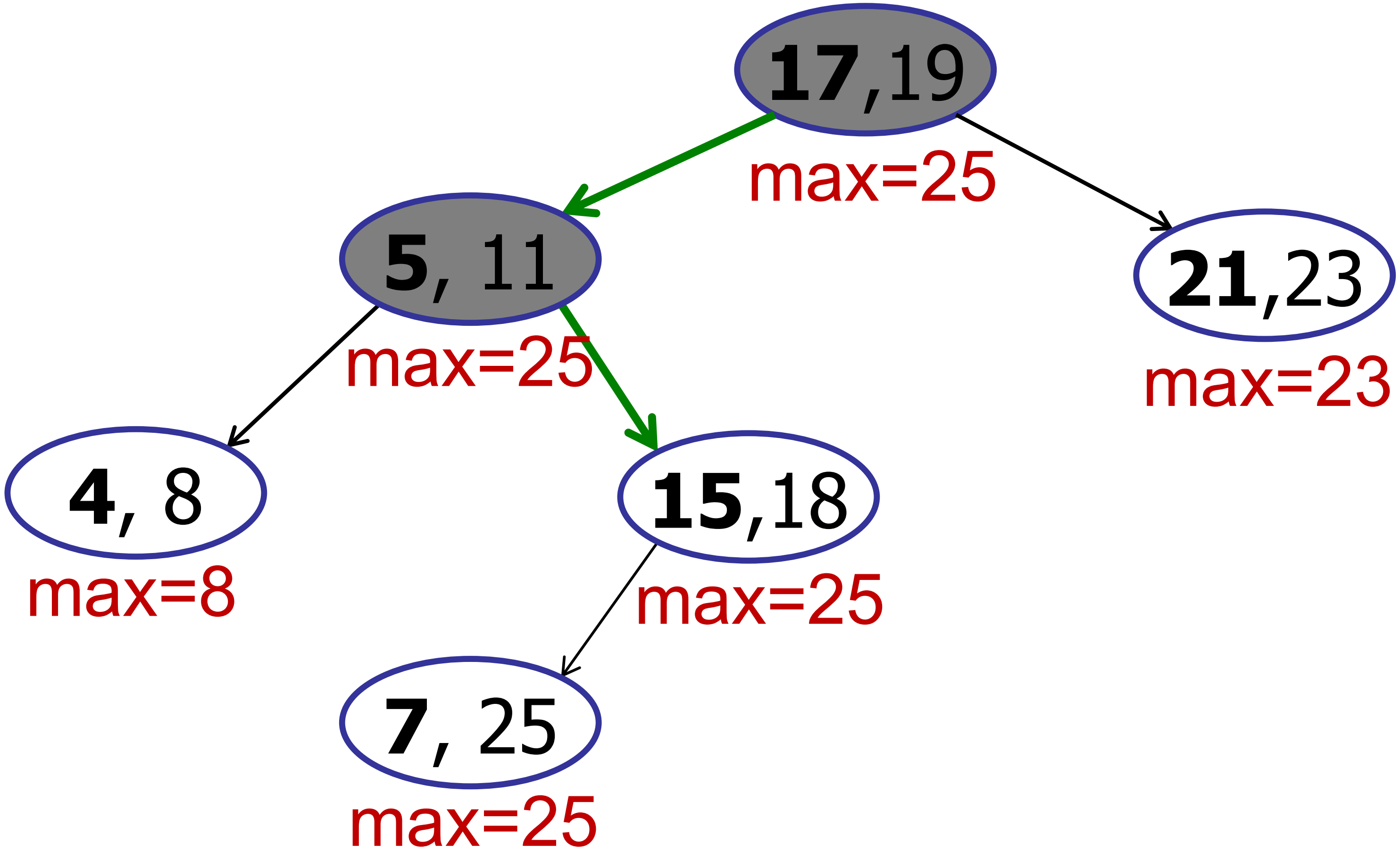
Interval Trees

Searching: interval-search(22)



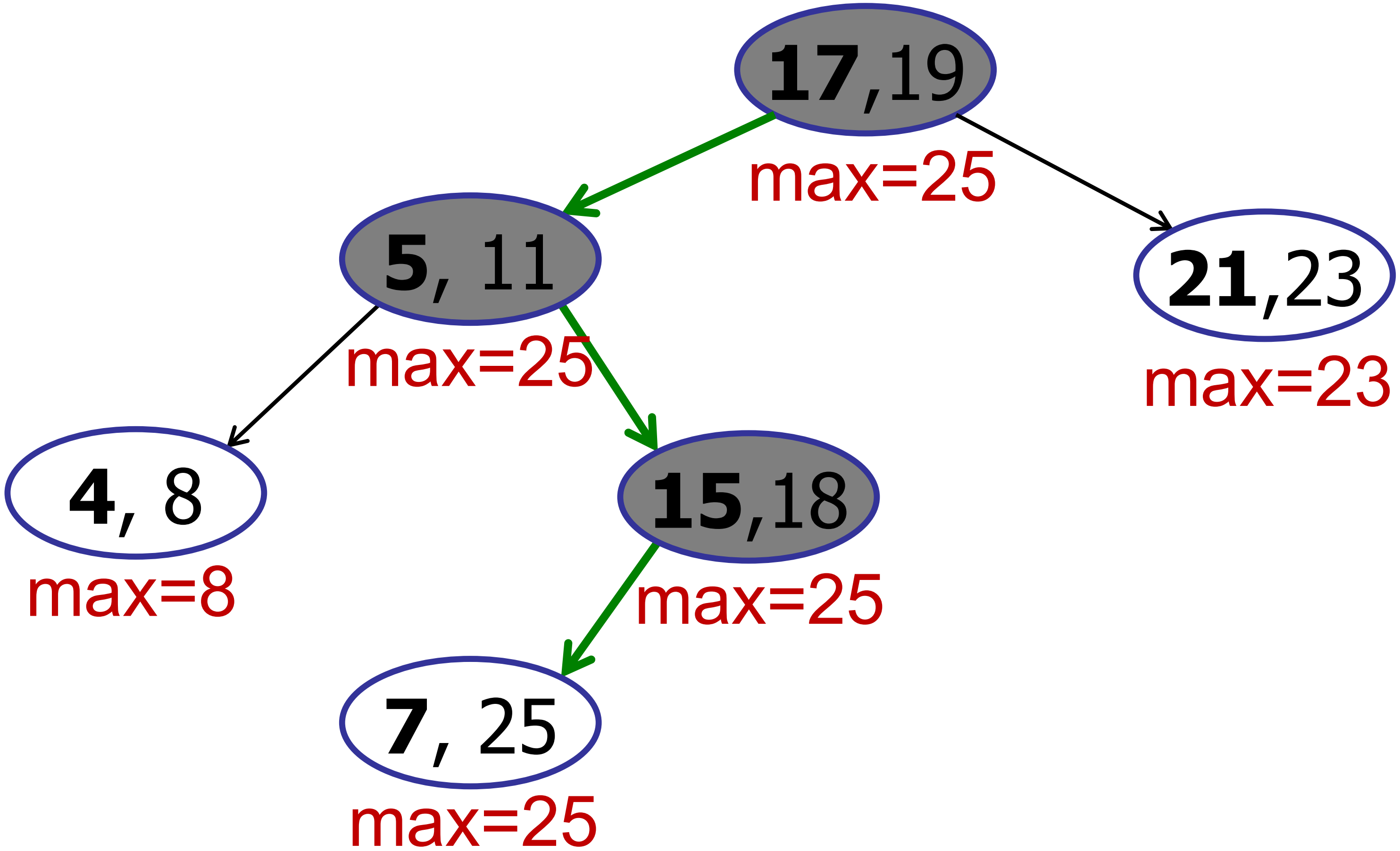
Interval Trees

Searching: interval-search(22)



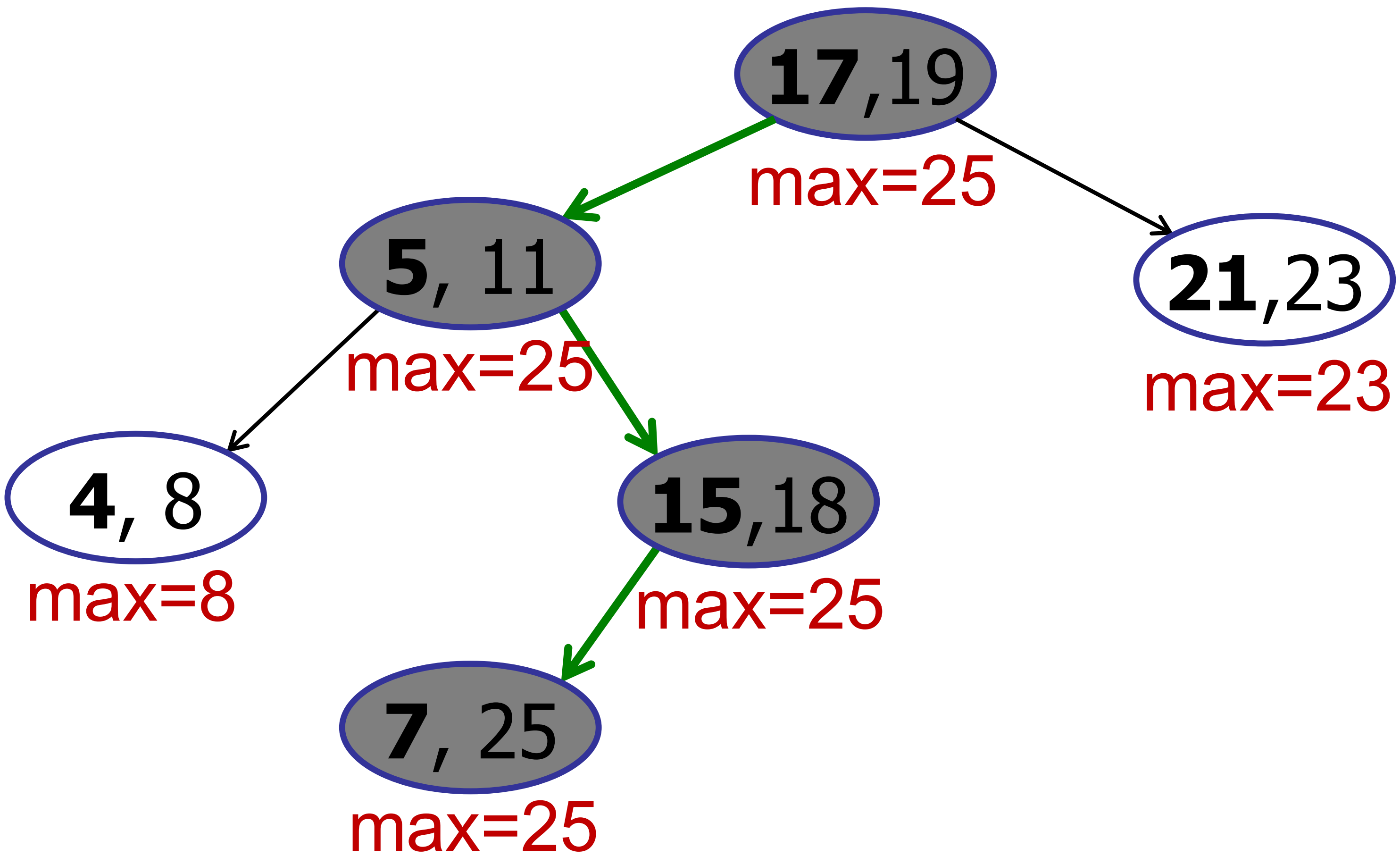
Interval Trees

Searching: interval-search(22)



Interval Trees

Searching: interval-search(22)



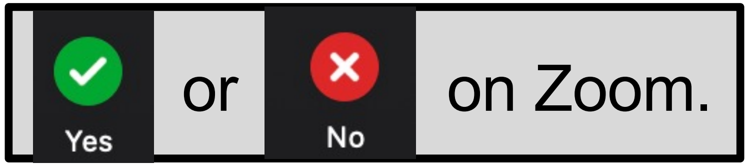
Interval Trees

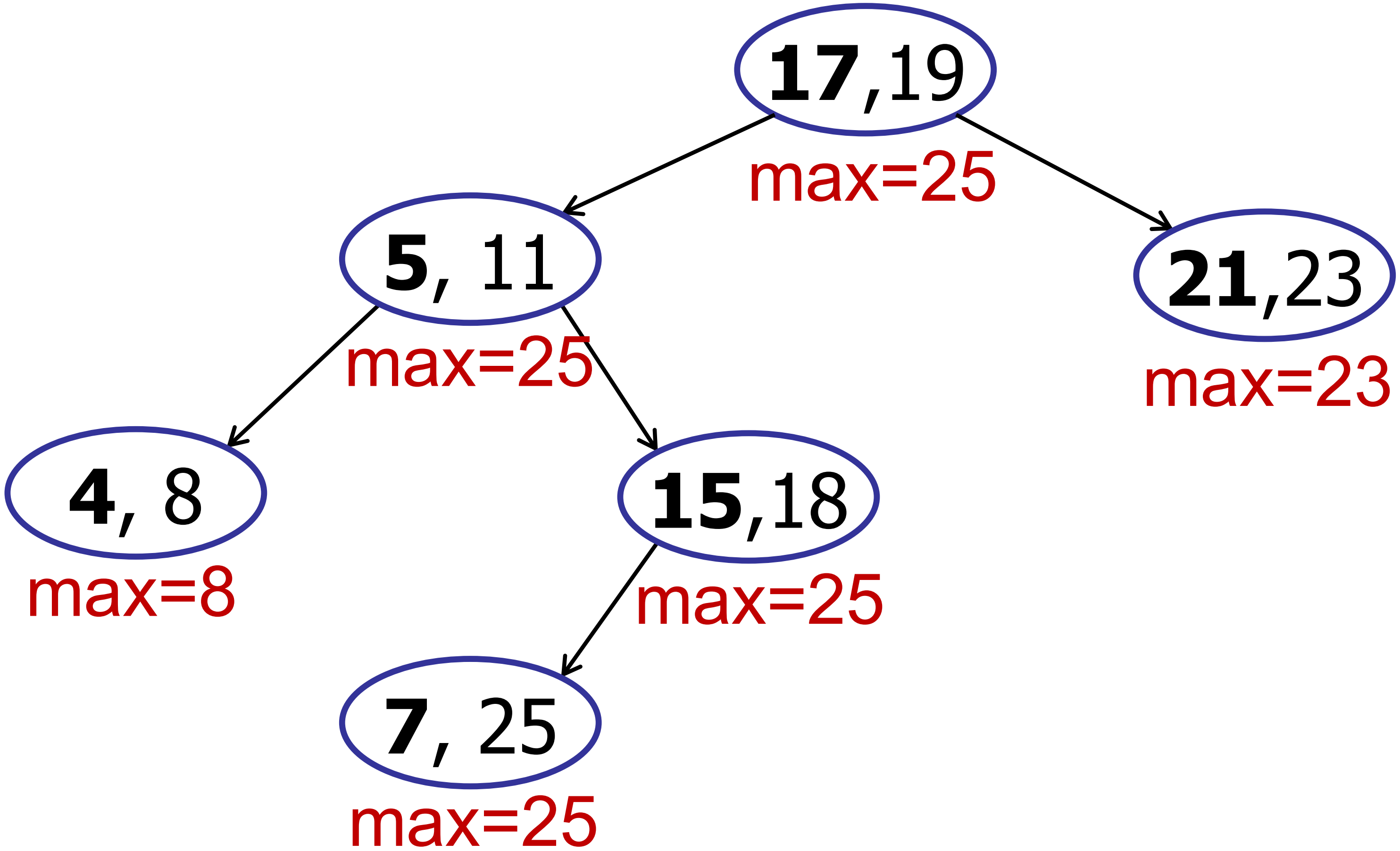
interval-search(x) : find interval containing x

```
interval-search(x)
    c = root;
    while (c != null and x is not in c.interval) do
        if (c.left == null) then
            c = c.right;
        else if  (x > c.left.max)  then
            c = c.right;
        else c = c.left;
    return c.interval;
```

Interval Trees

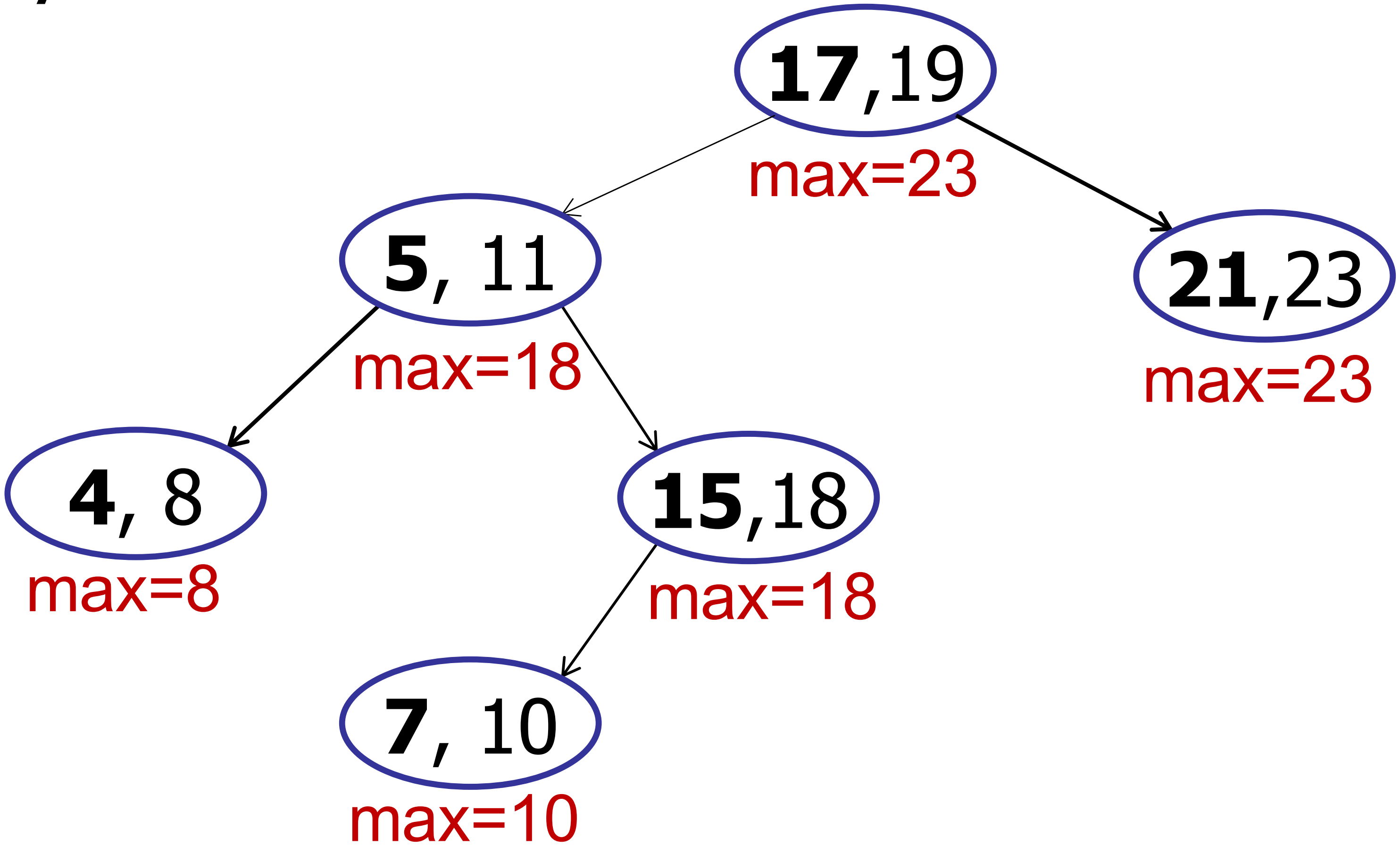
Will any search find (21, 23)?





Interval Trees

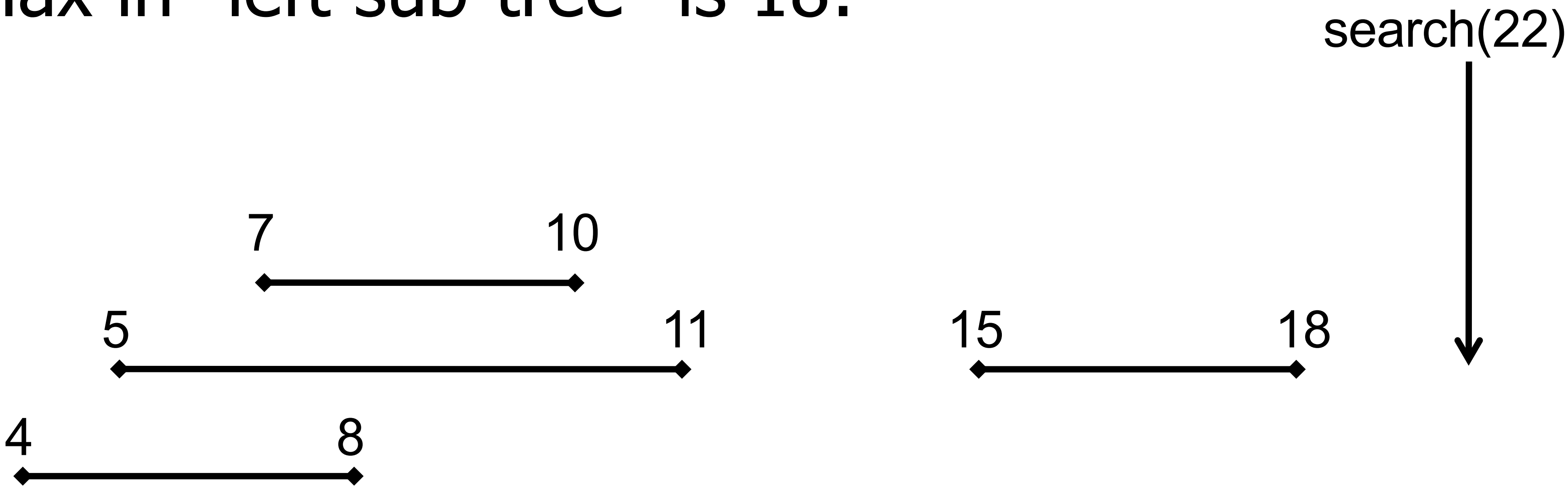
Why does it work?



Claim: If search goes right, then no overlap in left subtree.

Interval Trees

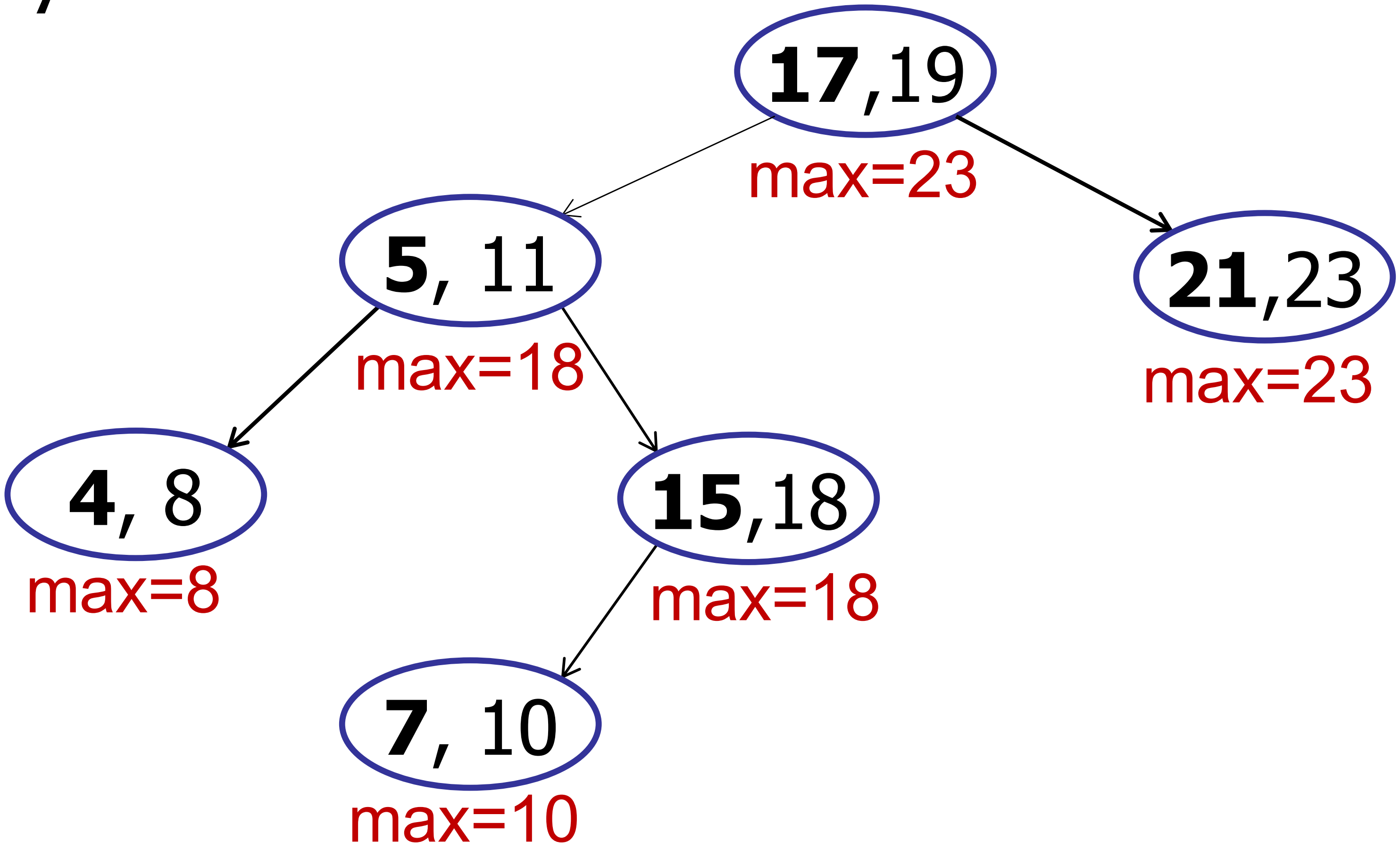
Max in “left sub-tree” is 18:



Safe to go right: 22 is not in the left sub-tree.

Interval Trees

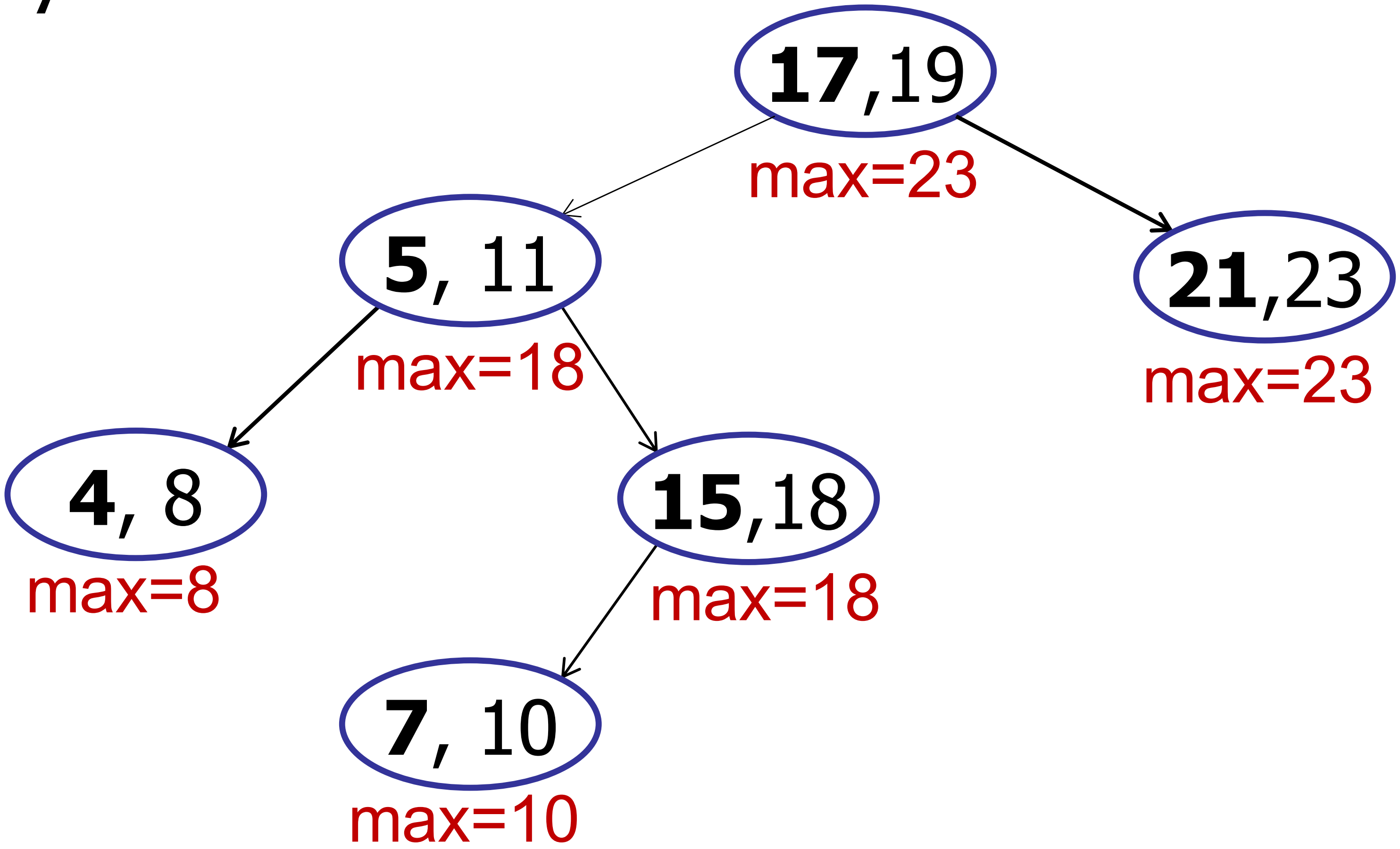
Why does it work?



Claim: If search goes left and there is no overlap in the left subtree…

Interval Trees

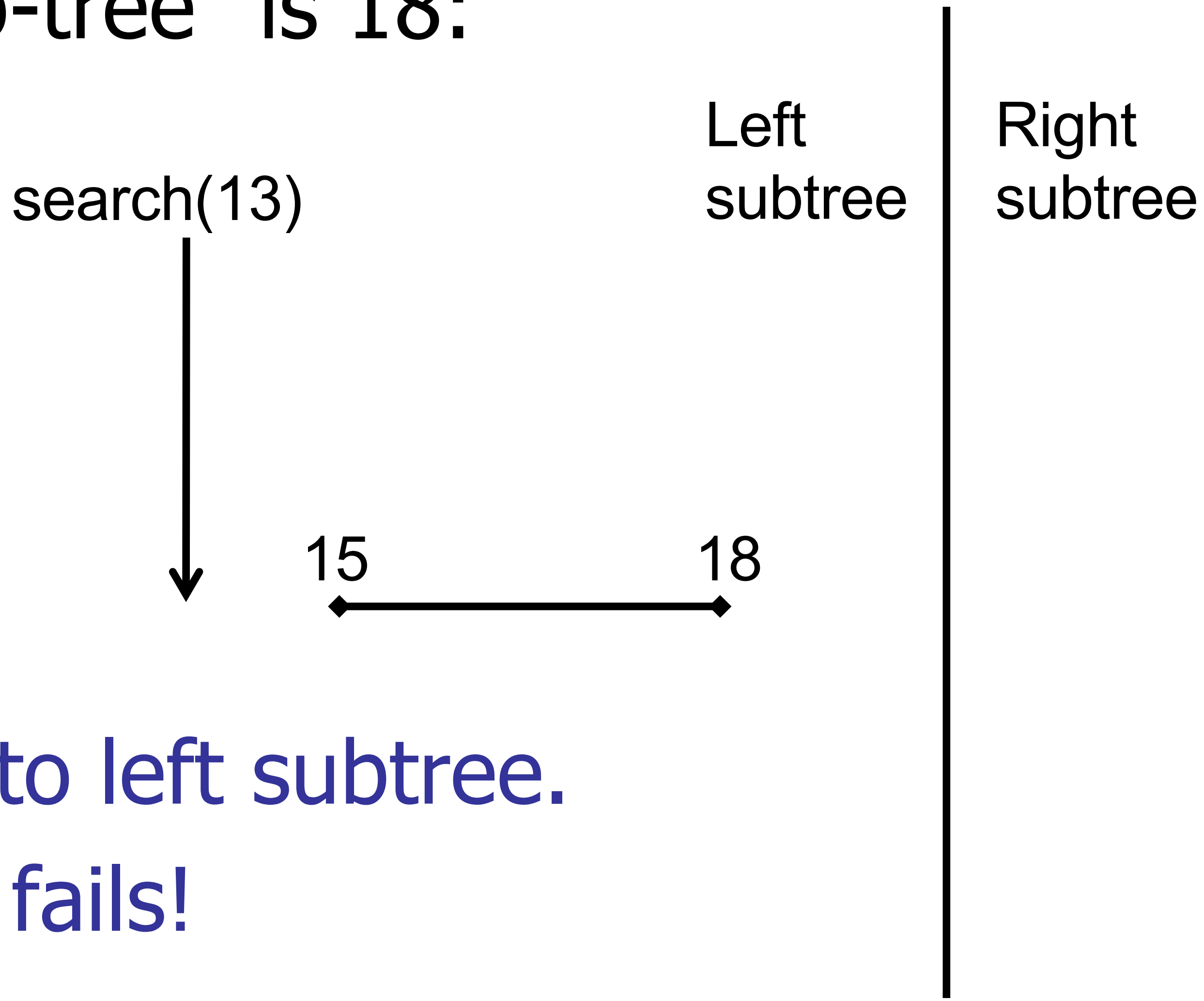
Why does it work?



Claim: If search goes left, then safe to go left.

Interval Trees

Max in “left sub-tree” is 18:

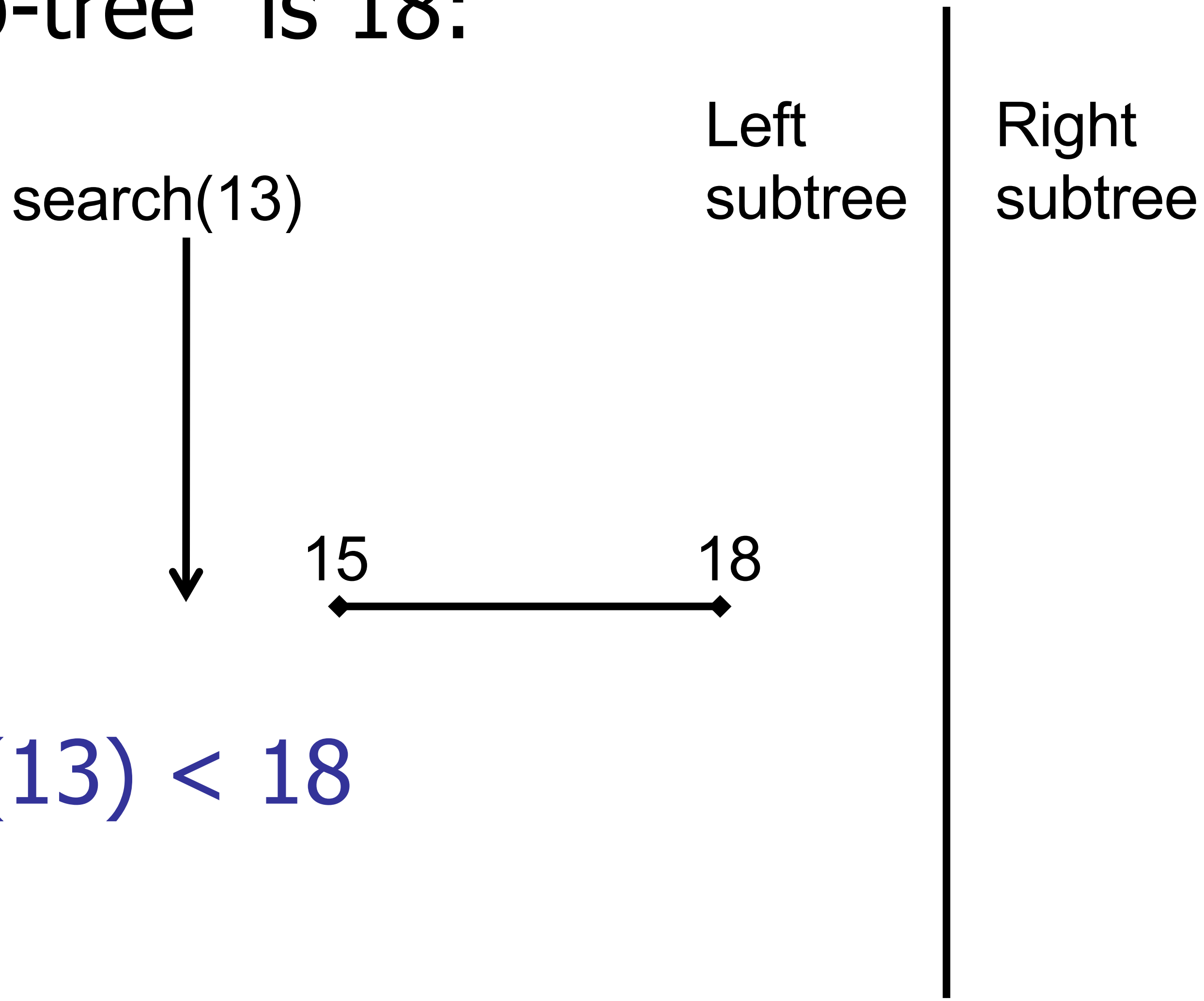


Assume we go to left subtree.

Assume search fails!

Interval Trees

Max in "left sub-tree" is 18:



Go left: search(13) < 18

Interval Trees

Max in “left sub-tree” is 18:

Left subtree

Right subtree

search(13)

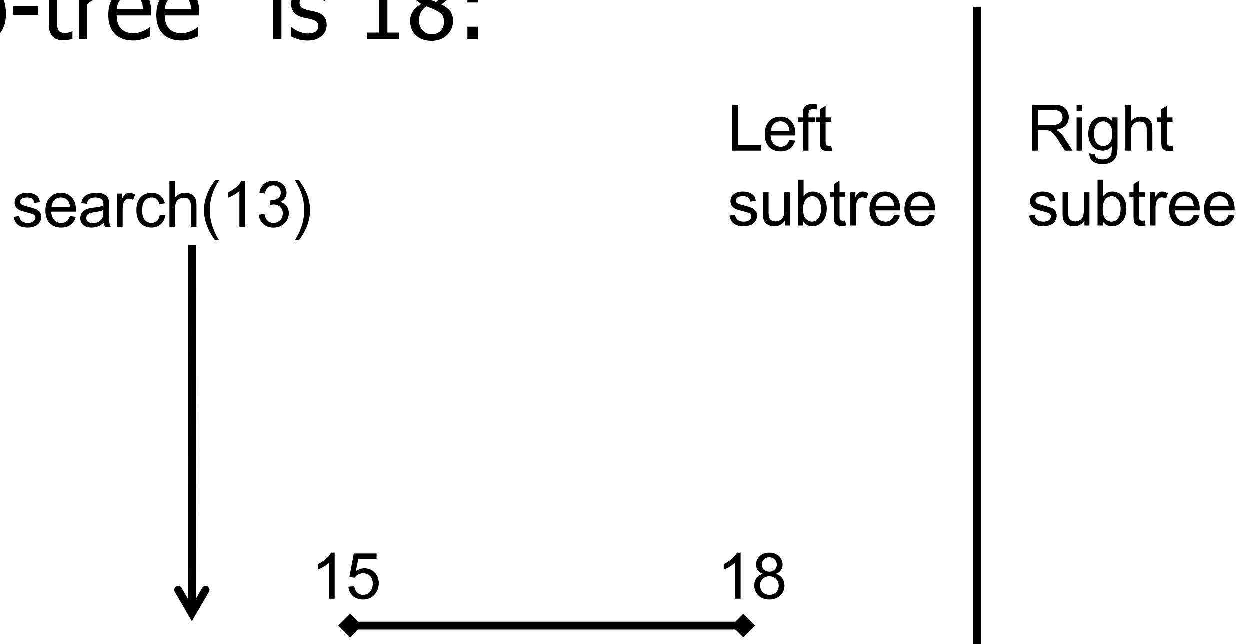
15

18

Go left: search(13) < 15 < 18

Interval Trees

Max in “left sub-tree” is 18:

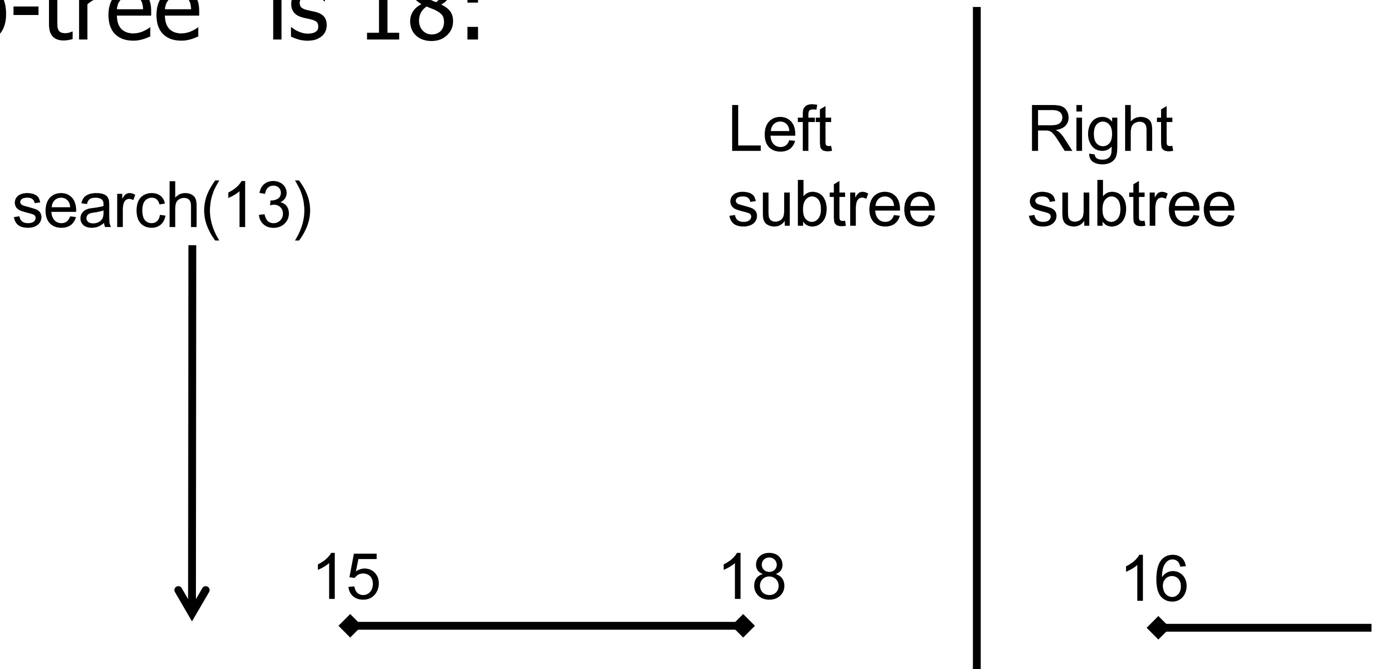


Go left: search(13) < 15 < 18

Tree sorted by left endpoint.

Interval Trees

Max in “left sub-tree” is 18:



Go left: search(13) < 15 < 18

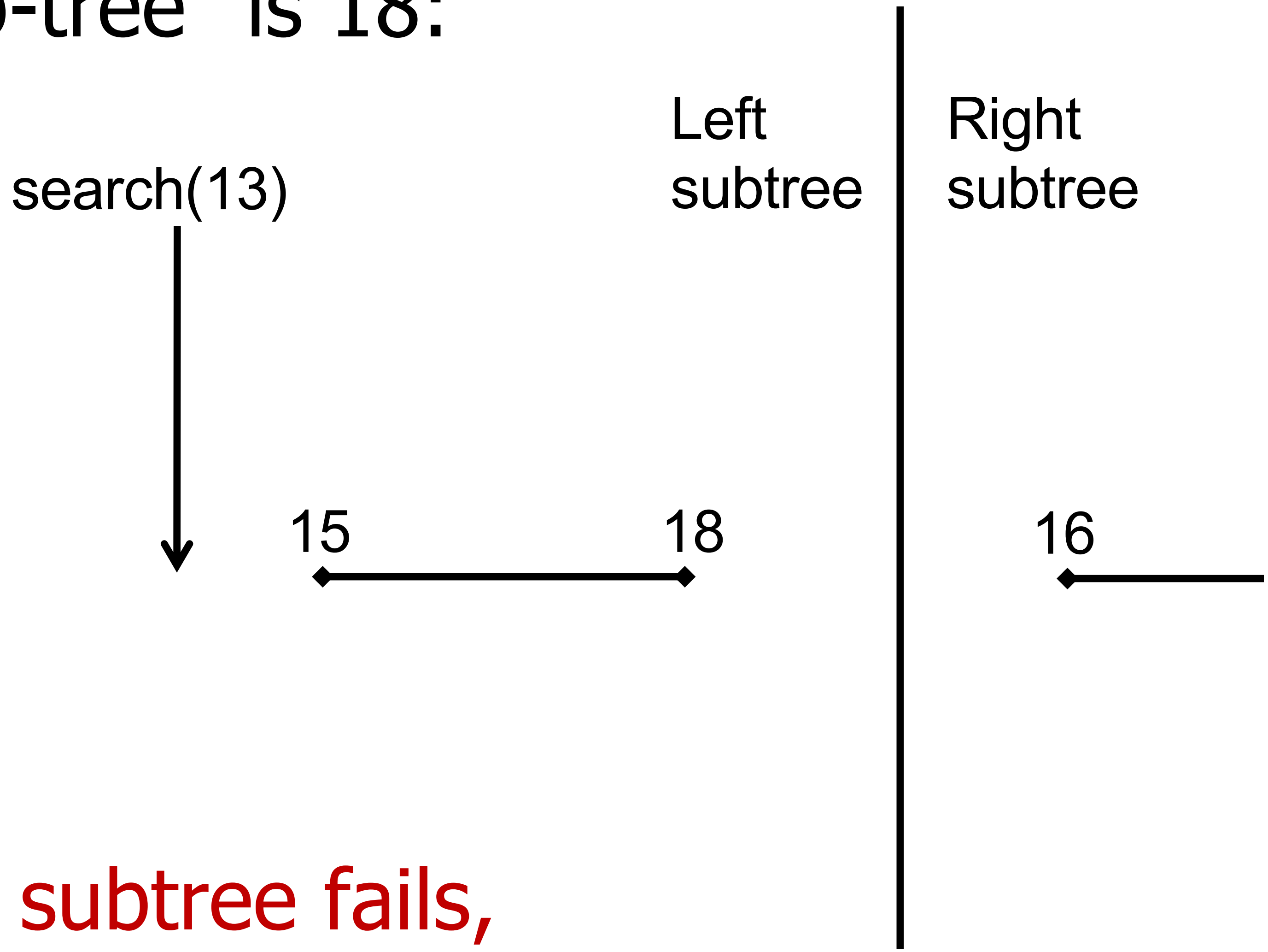
Tree sorted by left endpoint.

13 < every interval in right subtree

➔ Search also would fail in right subtree

Interval Trees

Max in "left sub-tree" is 18:

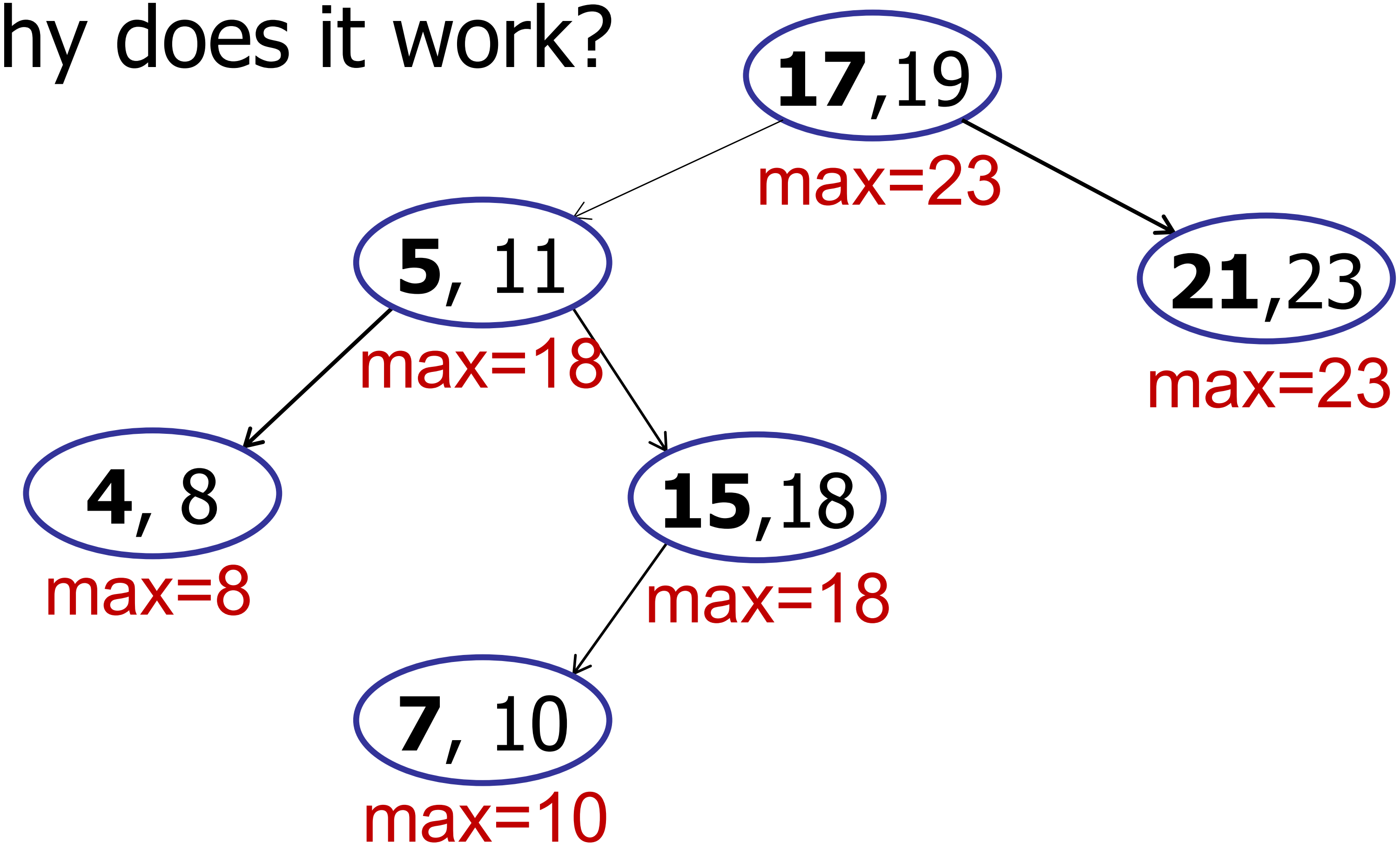


If search in left subtree fails,

Then search also would fail in right subtree!

Interval Trees

Why does it work?



Claim: If search goes left and fails, then key < every interval in right sub-tree.

Interval Trees

If search goes right: then no interval in left subtree.

➔ Either search finds key in right subtree or it is not in the tree.

If search goes left: if there is no interval in left subtree, then there is no interval in right subtree either.

➔ Either search finds key in left subtree or it is not in the tree.

Conclusion: search finds an overlapping interval, if it exists.

The running time of interval-search is:

1. $O(1)$
2. $\boxed{O(\log n)}$
3. $O(n)$
4. $O(n \log n)$
5. $O(n^2)$
6. Can’t say.

ARCHIPELAGO

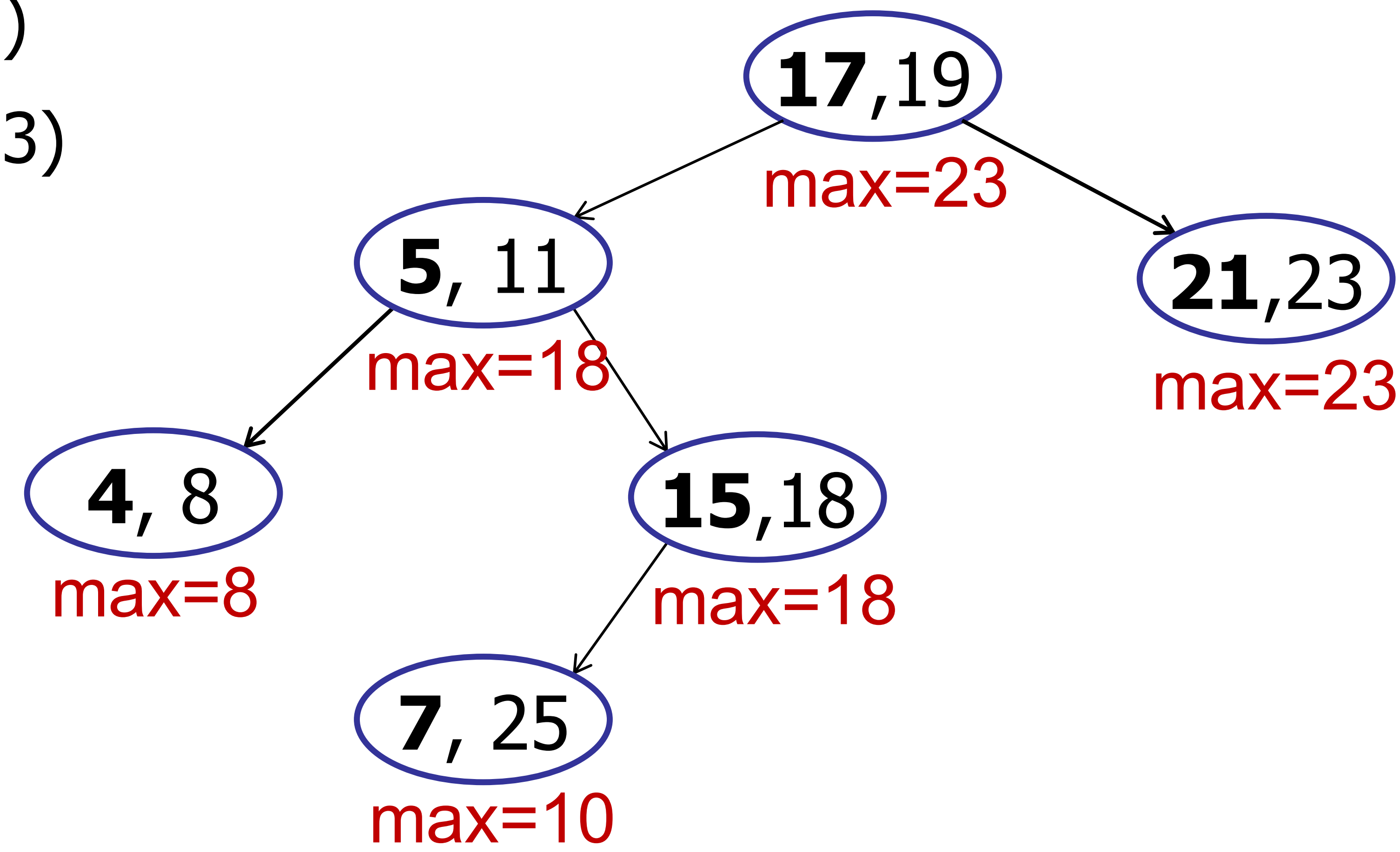
is open

Interval Trees

Extension: List all intervals that overlap with point?

E.g.: search(22) returns:

- (7,25)
- (21,23)



Interval Trees

Extension: List all intervals that overlap with point?

All-Overlaps Algorithm:

Repeat until no more intervals:

- Search for interval.
- Add to list.
- Delete interval.

Repeat for all intervals on list:

- Add interval back to tree.

The running time of All-Overlaps, if there are k overlapping intervals?

1. O(1)
2. O(k)
3. $\boxed{\text{O(k log n)}}$
4. O(k + log n)
5. O(kn)
6. O(kn log n)

ARCHIPELAGO

is open

Interval Trees

Extension: List all intervals that overlap with point?

All-Overlaps Algorithm: O(k log n)

Repeat until no more intervals:

- Search for interval.
- Add to list.
- Delete interval.

Repeat for all intervals on list:

- Add interval back to tree.

Best known solution: O(k + log n)

Today

Two examples of augmenting BSTs

1. Order Statistics

2. Intervals