CS2040S

Data Structures and Algorithms

(e-learning edition)

Augmented Trees!

Part 1

Last week…

Dictionaries

Binary search trees

Tries

Balanced search trees

- AVL trees
- Skip lists
- B-trees

Today: Dynamic Data Structures

1. Maintain a set of items

2. Modify the set of items

3. Answer queries.

Today: 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.

B-trees are at the heart of *every* database!

Dynamic Data Structures

- Operations that create a data structure
  - build (preprocess)
- Operations that modify the structure
  - insert
  - delete
- Query operations
  - search, select, etc.

“Why do we need to learn how an AVL tree works?”

Just use a Java TreeMap, right?

“Why do we need to learn how an AVL tree works?”

1. Learn how to think like a computer scientist.

“Why do we need to learn how an AVL tree works?”

1. Learn how to think like a computer scientist.
2. Learn to modify existing data structures to solve new problems.

Augmented Data Structures

Many problems require storing additional data in a standard data structure.

Augment more frequently than invent…

Today

Three examples of augmenting balanced BSTs

1. Order Statistics

2. Interval Queries

3. Orthogonal Range Searching

Augmenting data structures

Basic methodology:

1. Choose underlying data structure

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

Augmenting data structures

Basic methodology:

1. Choose underlying data structure

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

2. Determine additional info needed.

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.)

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.

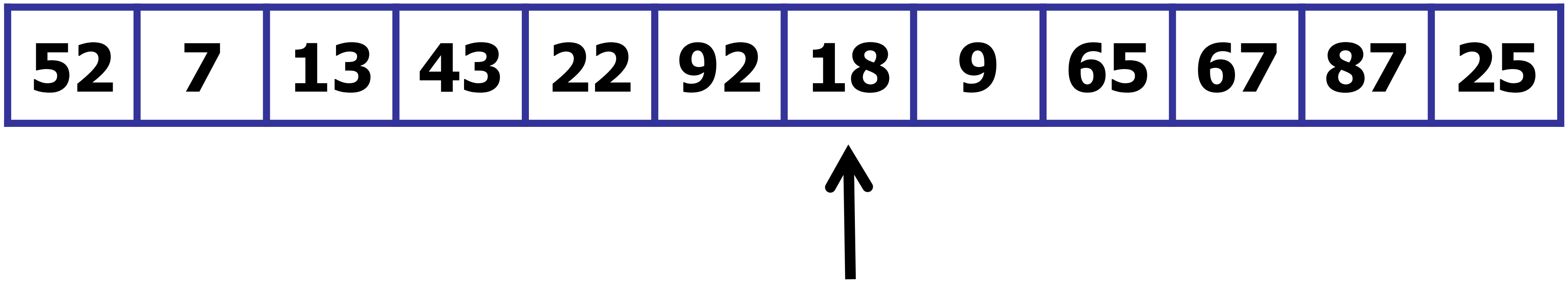
Order Statistics

Input

A set of integers.

Output: select(k)

The kth item in the set.



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

select(4)

select(1) returns:

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

1. 52

[✓ 2. 7]

3. 13

4. 43

5. 25

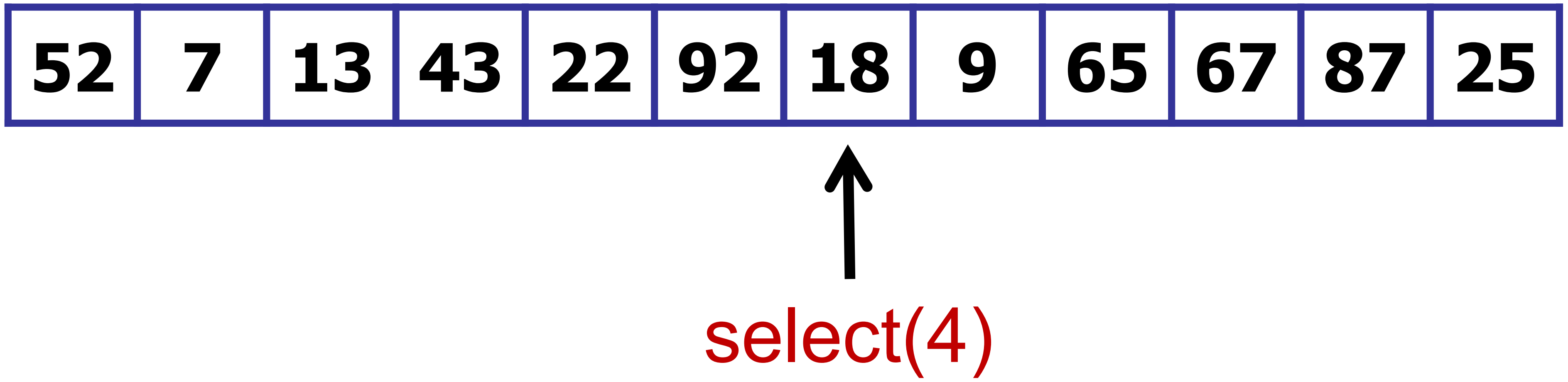
Order Statistics

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)

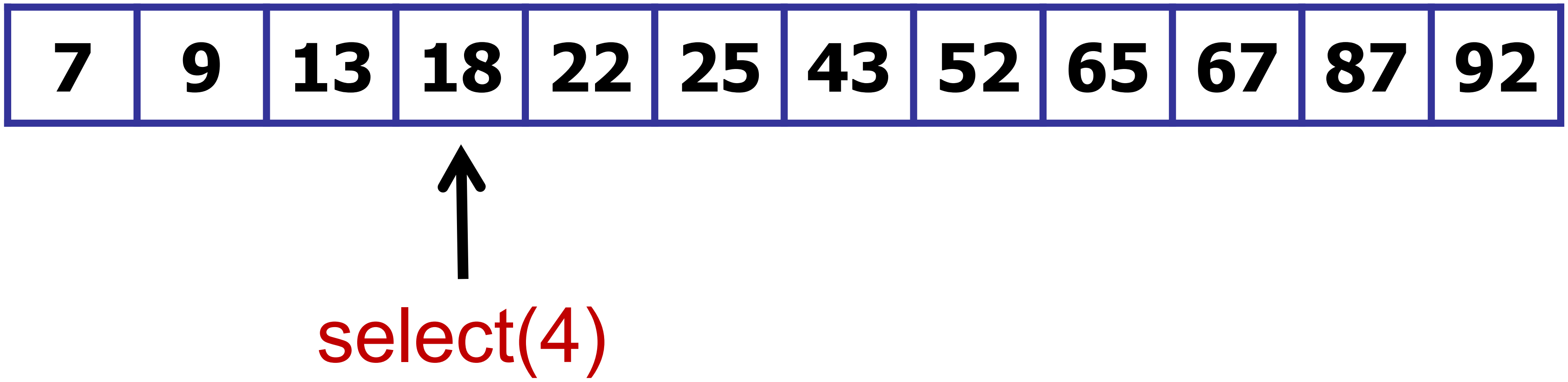
Order Statistics

Input

A set of integers.

Output: select(k)

The kth item in the set.



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

select(4)

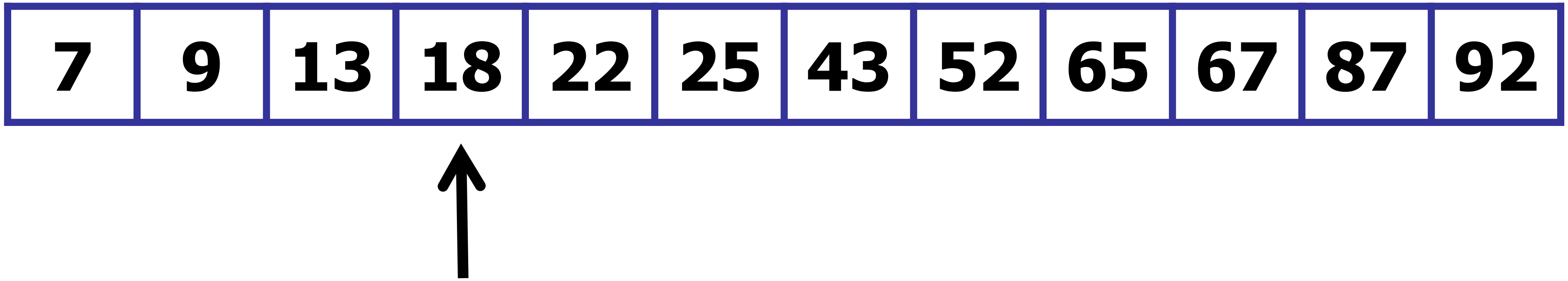
Order Statistics

Input

A set of integers.

Output: select(k) → Sort: O(n log n)

The kth item in the set.



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

select(4)

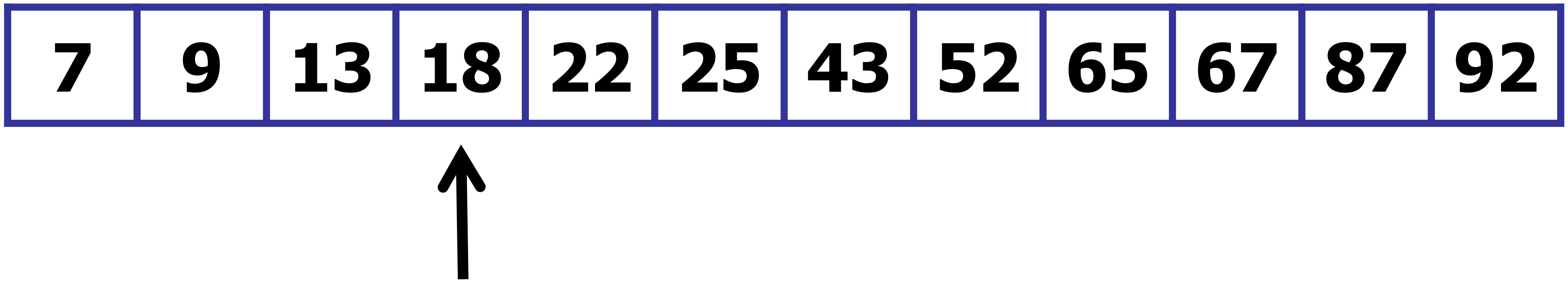
Order Statistics

Input

A set of integers.

Output: select(k) → QuickSelect: O(n)

The k^th item in the set.



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

select(4)

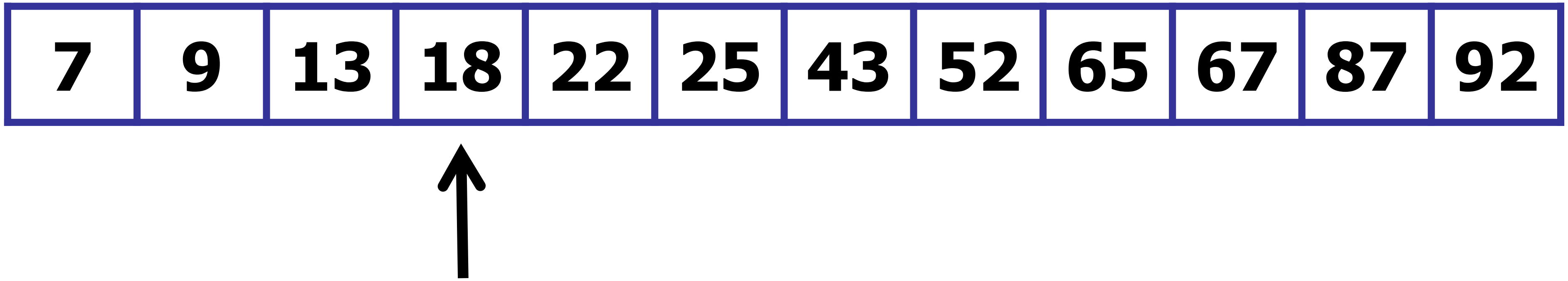
Order Statistics

Solution 1:

Sort: O(n log n)

Solution 2:

QuickSelect: O(n)



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

select(4)

Order Statistics

Solution 1:

Preprocess: sort --- O(n log n)

Select: O(1)

Solution 2:

Preprocess: nothing --- O(1)

QuickSelect: O(n)

Order Statistics

Solution 1:

Preprocess: sort --- O(n log n)

Select: O(1)

Solution 2:

Preprocess: nothing --- O(1)

QuickSelect: O(n)

Trade-off: how many items to select?

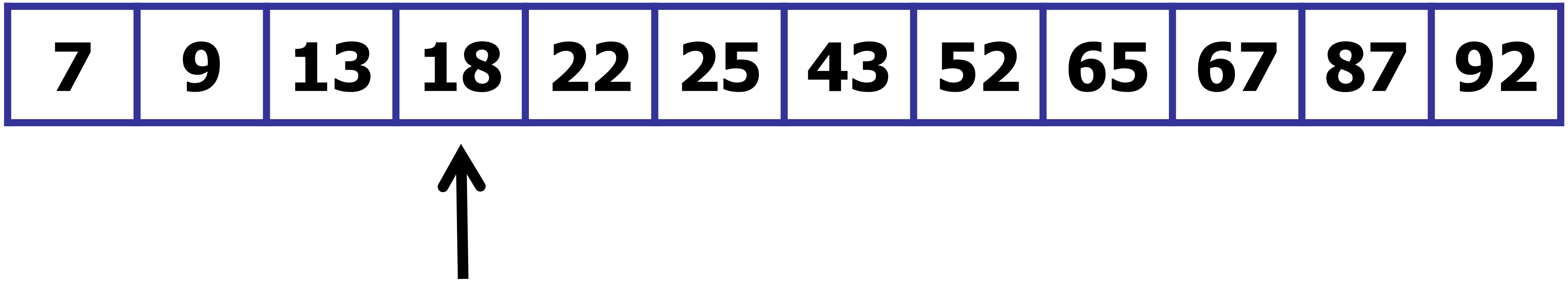
Dynamic Order Statistics

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

Solution 1:

Basic structure: sorted array A.

insert(int item): add item to sorted array A.

select(int k): return A[k]

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

Dynamic Order Statistics

Solution 2:

Basic structure: unsorted array A.

insert(int item): add item to end of array A.

select(int k): run QuickSelect(k)

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

When is it more efficient to maintain a sorted array (Solution 1)?

A. Always

B. When there are more inserts than selects.

[✓ C. When there are more selects than inserts.]

D. Never

E. I’m confused.

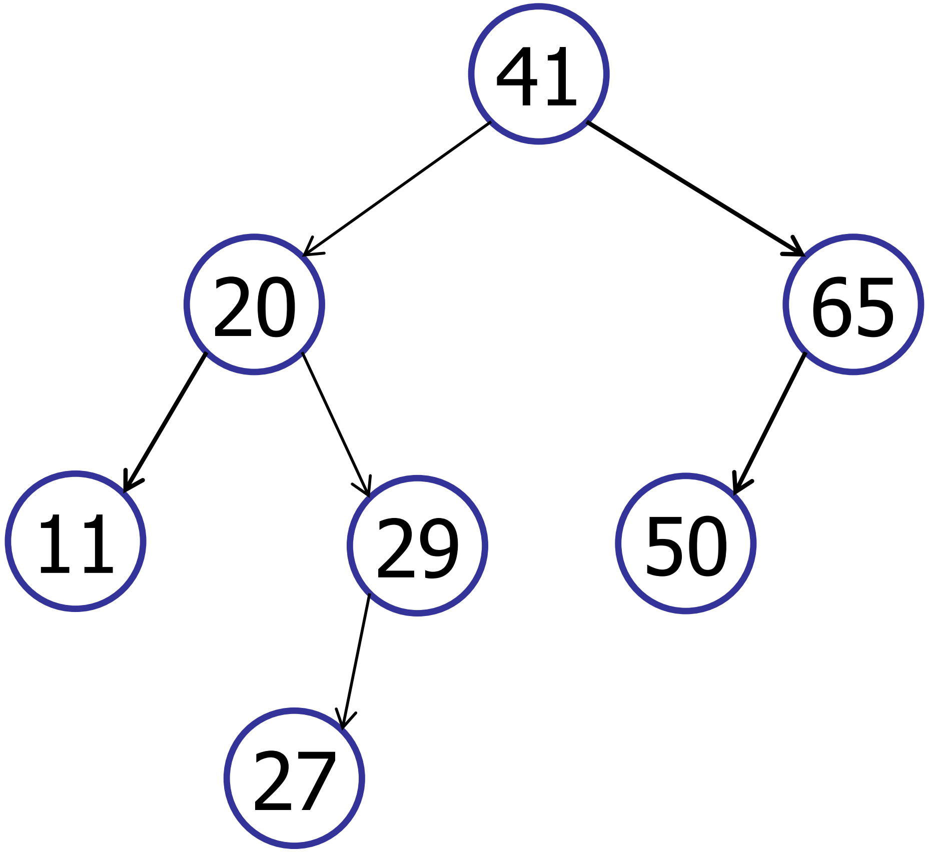
Dynamic Order Statistics

	Insert	Select
Solution 1: Sorted Array	O(n)	O(1)
Solution 2: Unsorted Array	O(1)	O(n)

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

Dynamic Order Statistics

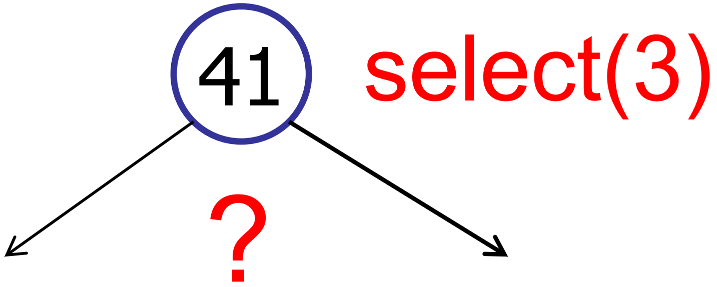
Today: use a (balanced) tree



11	**20**	**27**	**29**	**41**	**50**	**65**

Dynamic Order Statistics

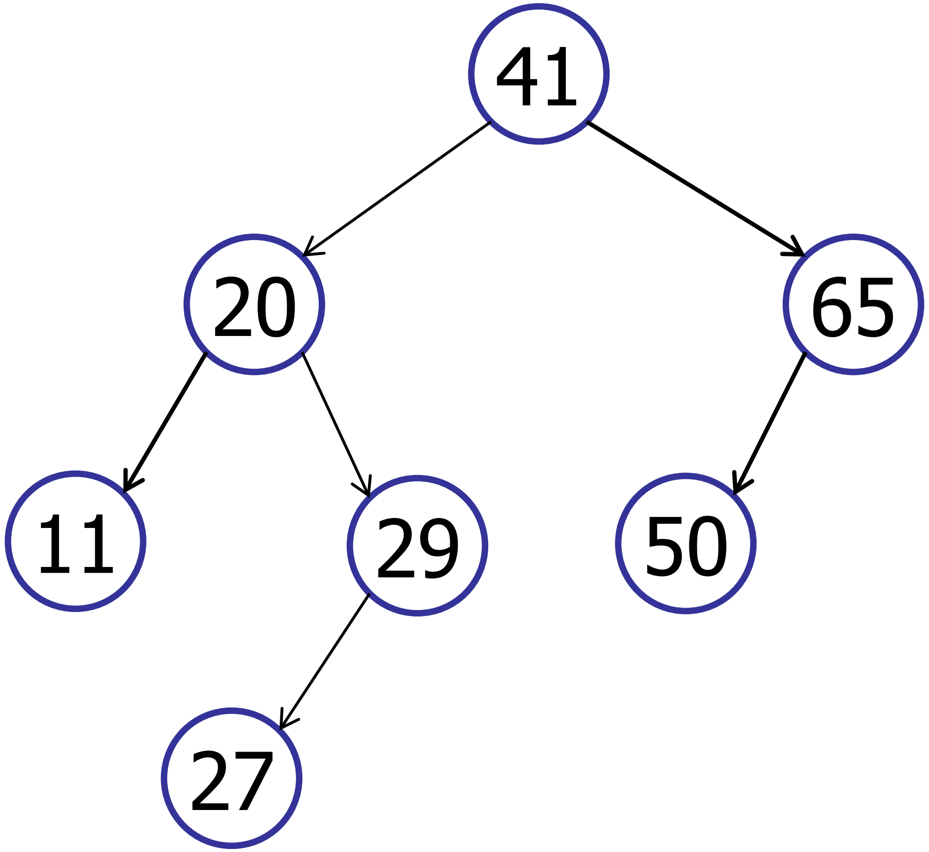
How to find the right item?



Dynamic Order Statistics

Simple solution: traversal

select(k): O(k)
in-order
traversal

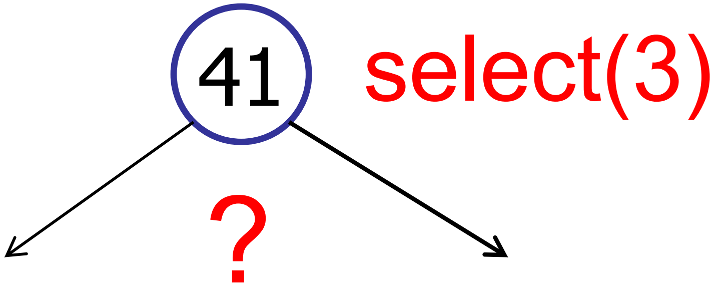


11	**20**	**27**	**29**	**41**	**50**	**65**

Dynamic Order Statistics

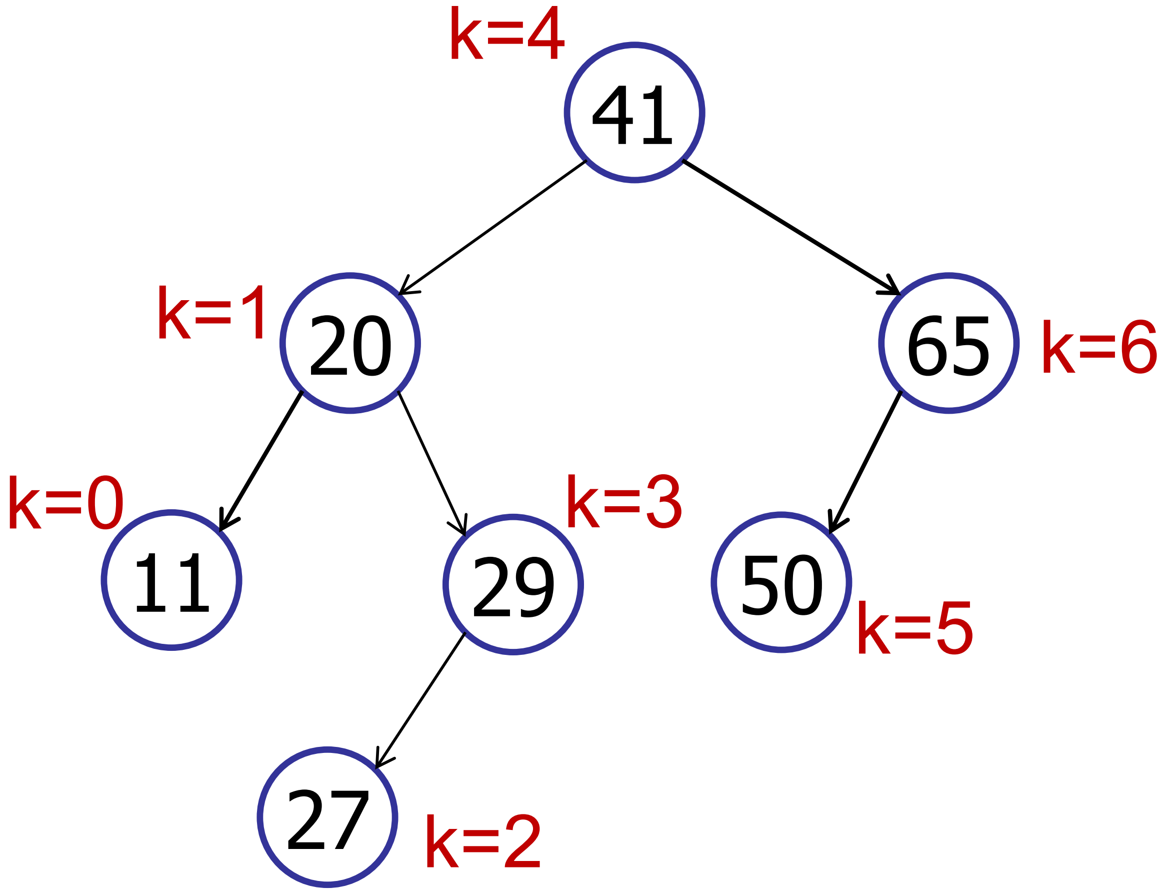
Augment!

What extra information would help?



Dynamic Order Statistics

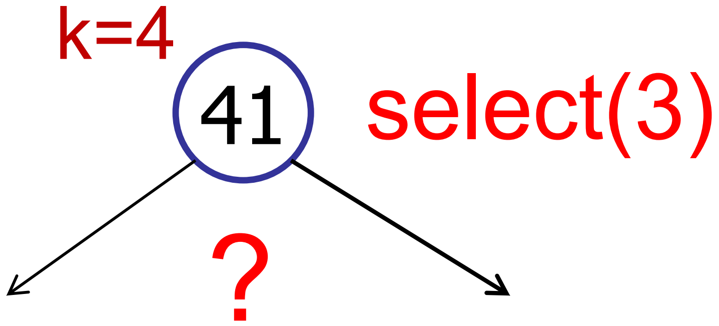
Idea: store rank in every node



11	**20**	**27**	**29**	**41**	**50**	**65**

Dynamic Order Statistics

Idea: store rank in every node



k=4

41

select(3)

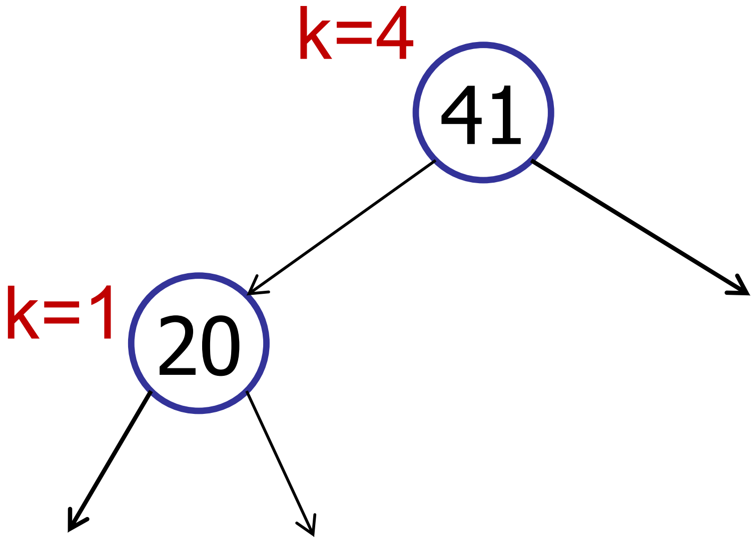
?

11	**20**	**27**	**29**	**41**	**50**	**65**

Dynamic Order Statistics

Idea: store rank in every node

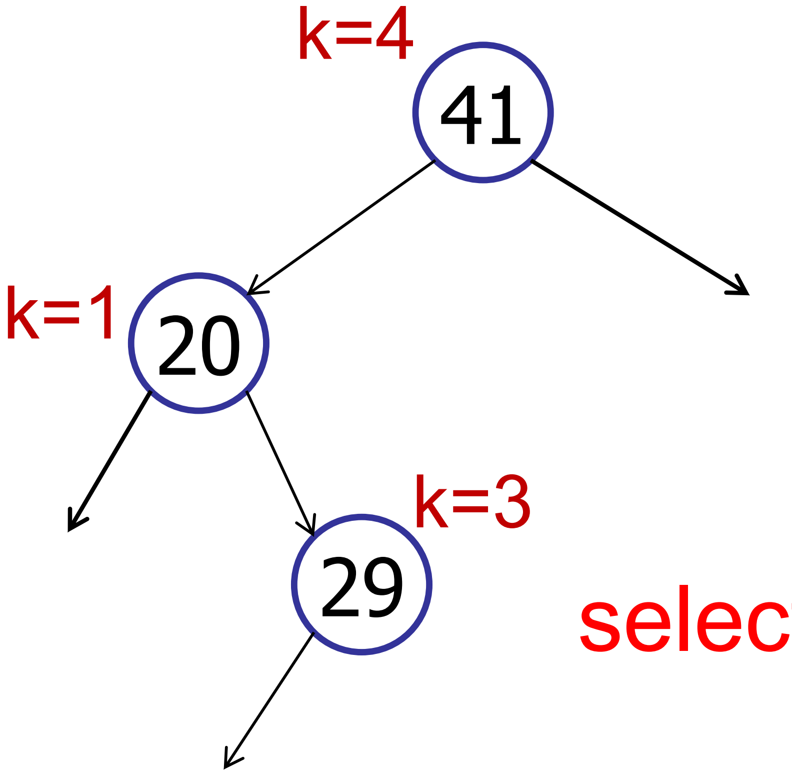
select(3)



11	**20**	**27**	**29**	**41**	**50**	**65**

Dynamic Order Statistics

Idea: store rank in every node

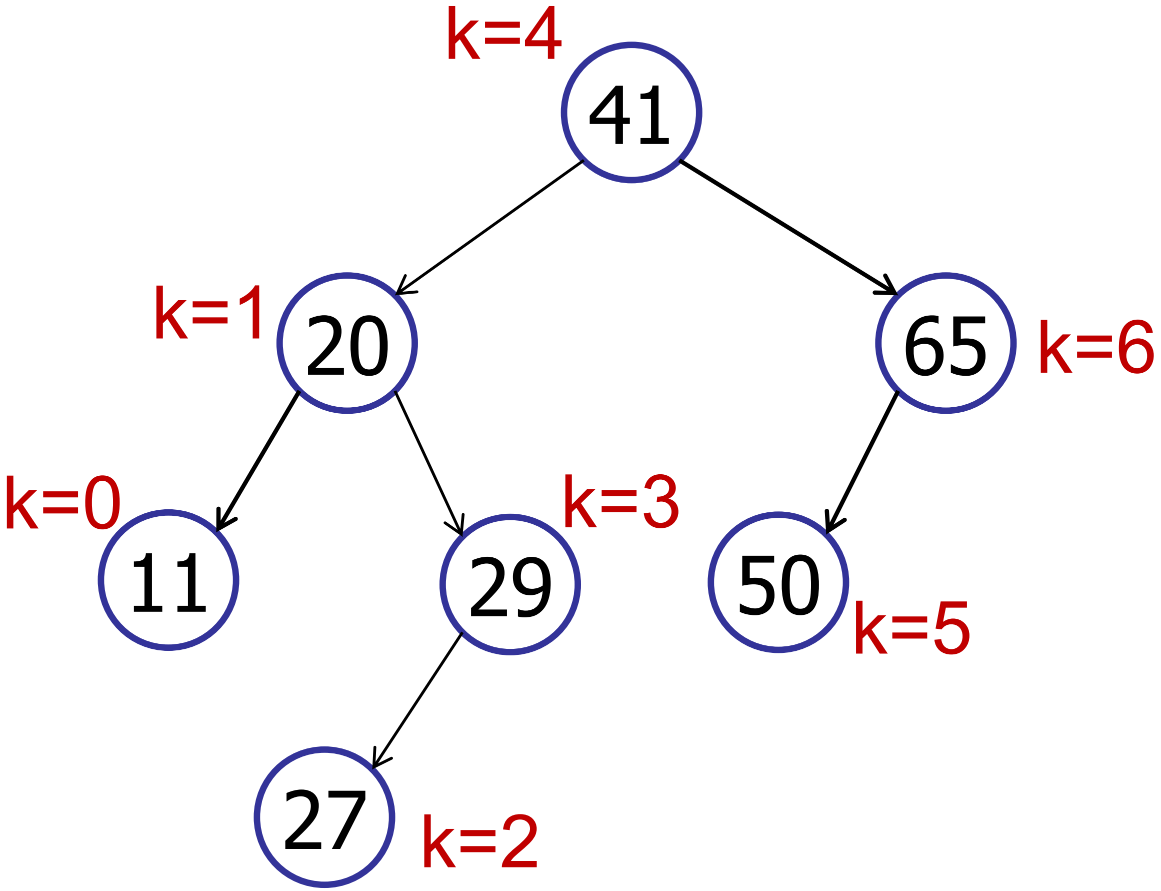


select(3)

11	**20**	**27**	**29**	**41**	**50**	**65**

Dynamic Order Statistics

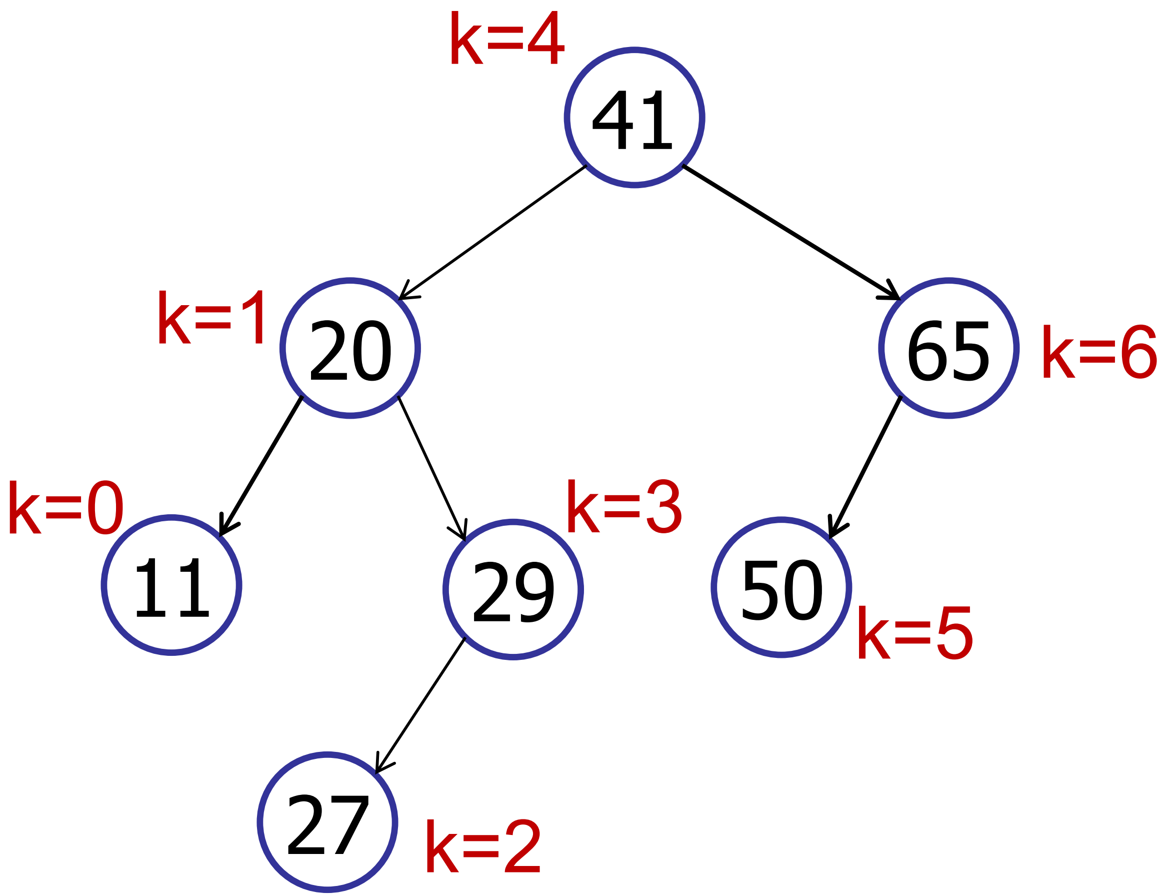
Idea: store rank in every node



11	**20**	**27**	**29**	**41**	**50**	**65**

Dynamic Order Statistics

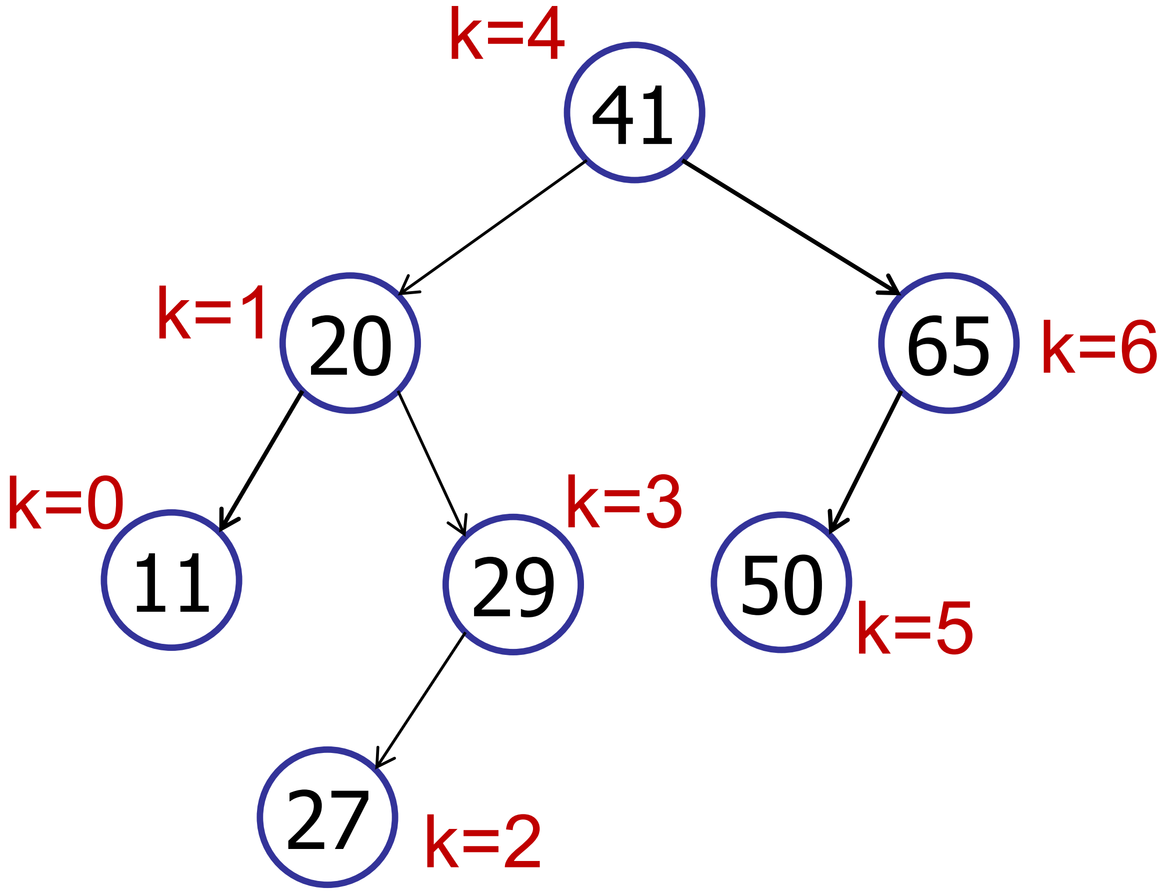
Idea: store rank in every node



Problem: insert(5)

Dynamic Order Statistics

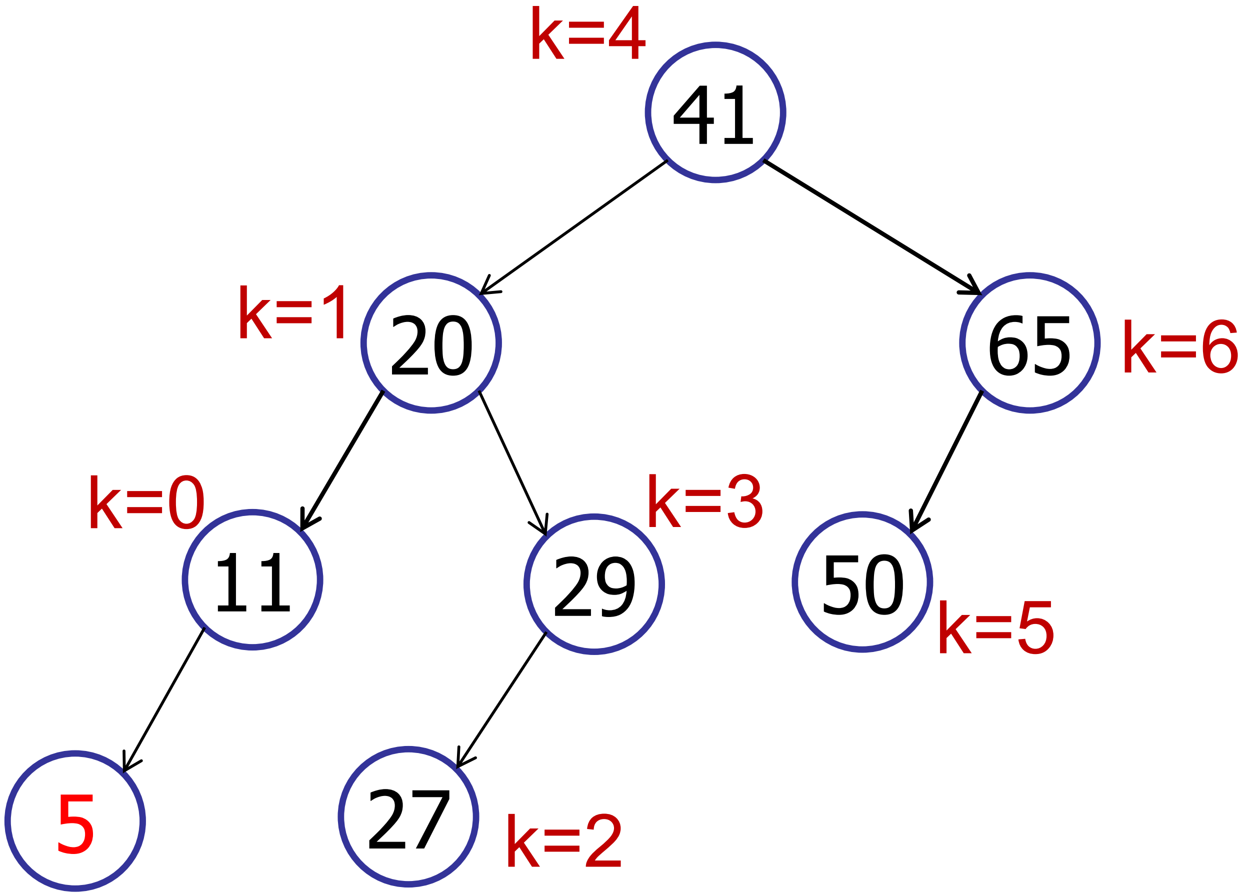
Idea: store rank in every node



Problem: insert(5) requires updating *all* the ranks!

Dynamic Order Statistics

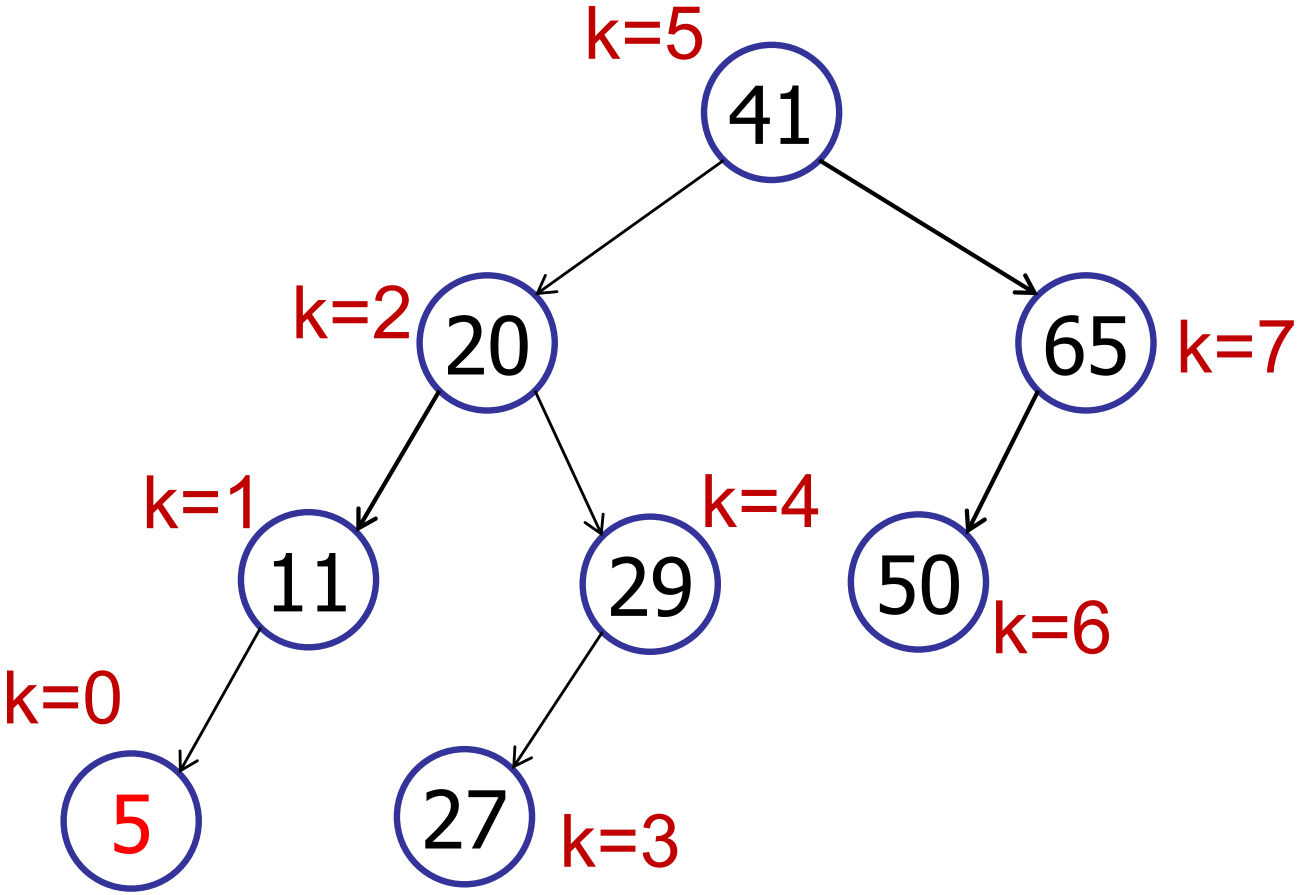
Idea: store rank in every node



5	**11**	**20**	**27**	**29**	**41**	**50**	**65**

Dynamic Order Statistics

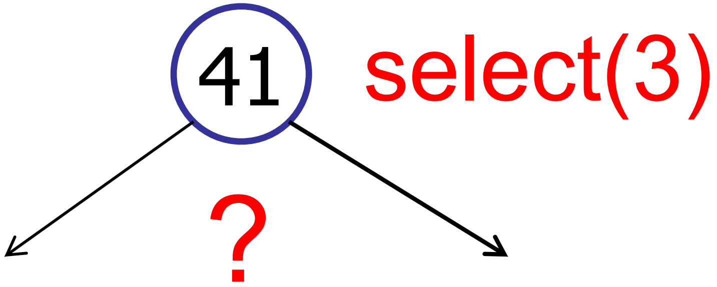
Conclusion: too expensive to store rank in every node!



5	**11**	**20**	**27**	**29**	**41**	**50**	**65**

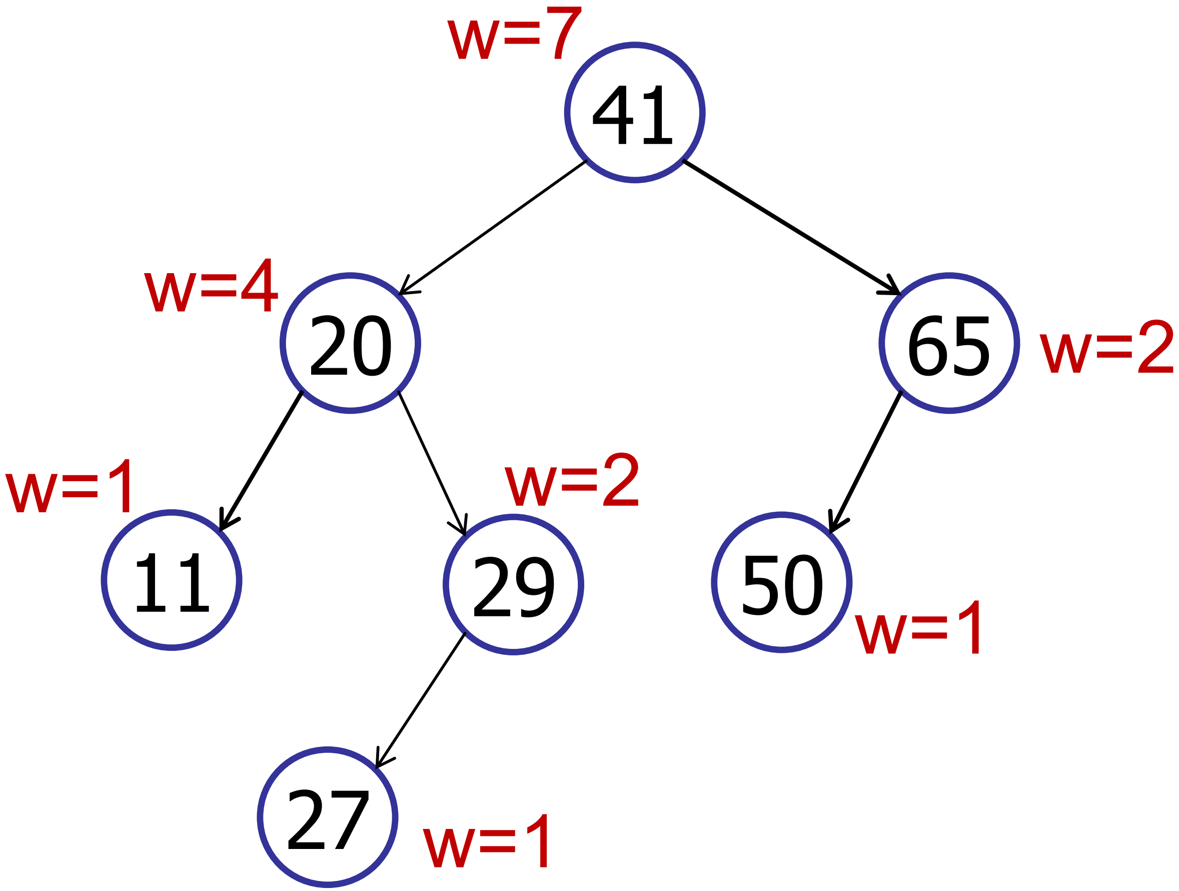
Dynamic Order Statistics

What should we store in each node?



Dynamic Order Statistics

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



Dynamic Order Statistics

Idea: store size of sub-tree in every node

The weight of a node is the size of the tree rooted at that node.

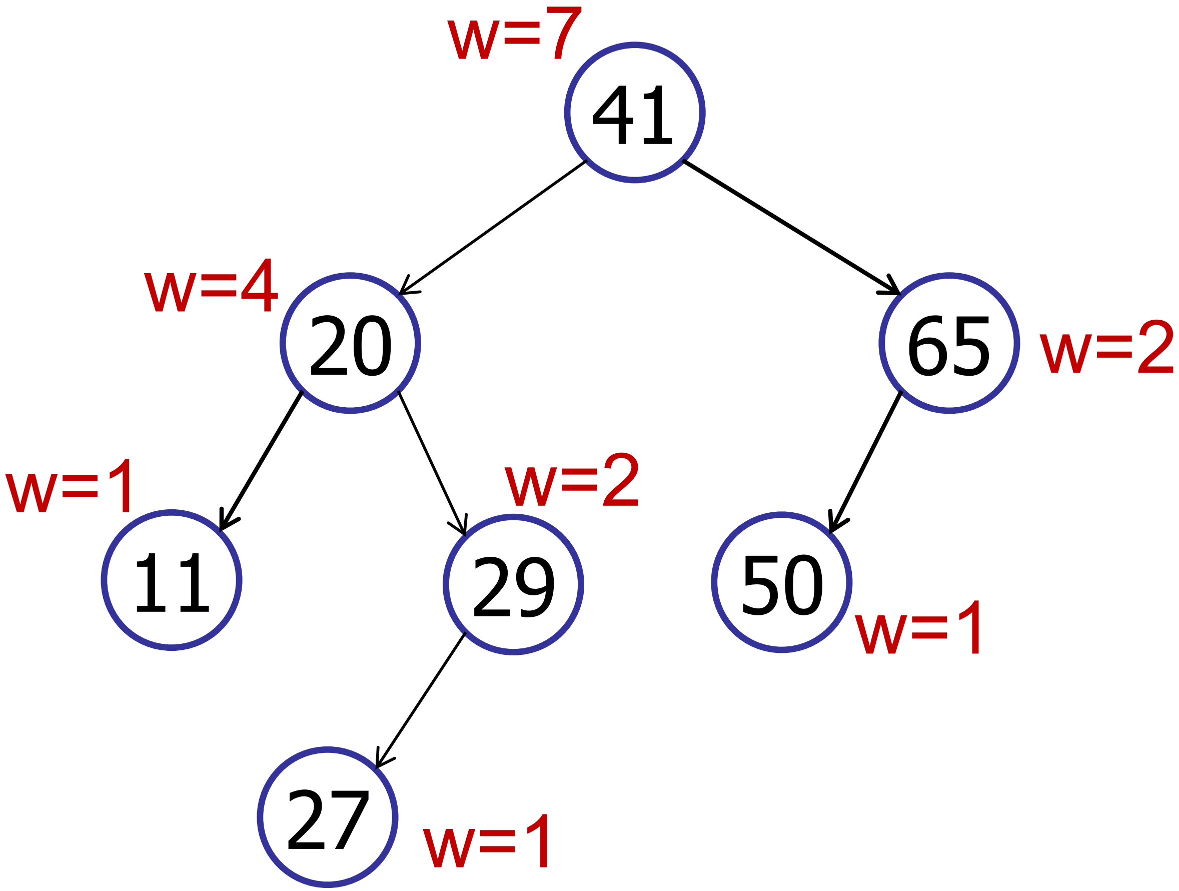
Define weight:

w(leaf) = 1

w(v) = w(v.left) + w(v.right) + 1

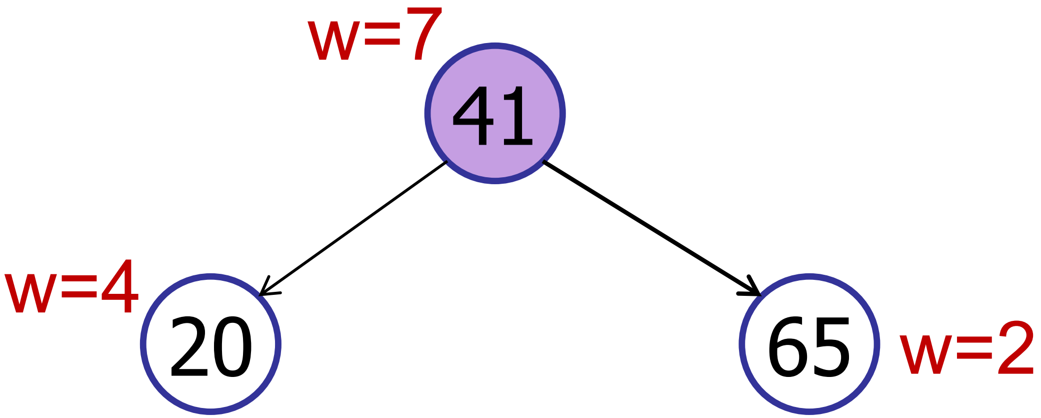
Dynamic Order Statistics

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



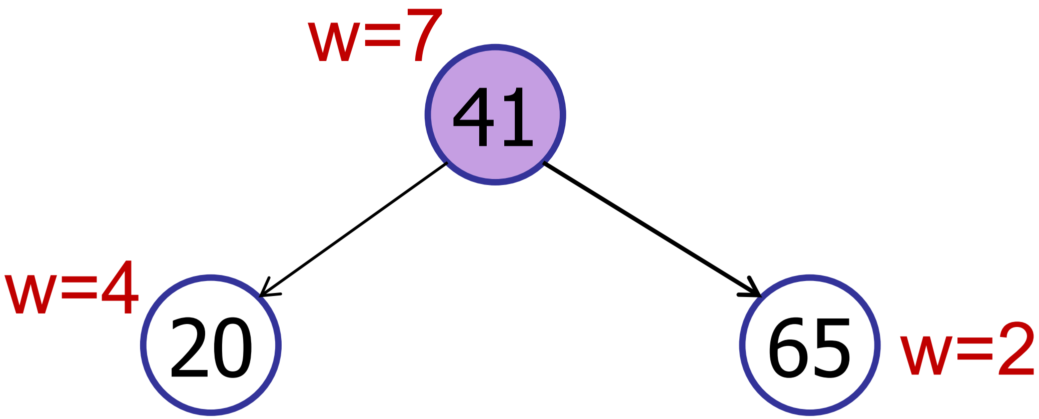
Dynamic Order Statistics

Example: select(3)



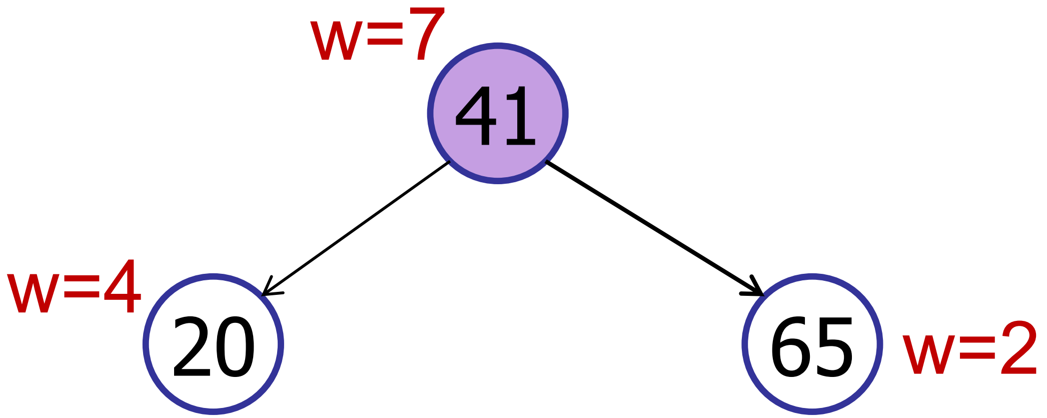
What is the rank of 41?

1. 1
2. 3
3. ✓ 5
4. 7
5. 9
6. Can’t tell.



Dynamic Order Statistics

Example: select(3)



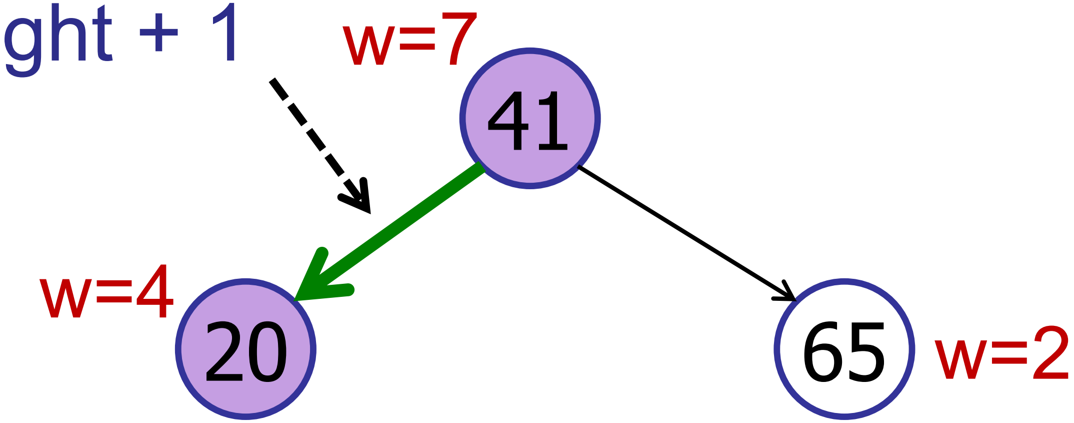
“rank in subtree” = left.weight + 1

Dynamic Order Statistics

Example: select(3)

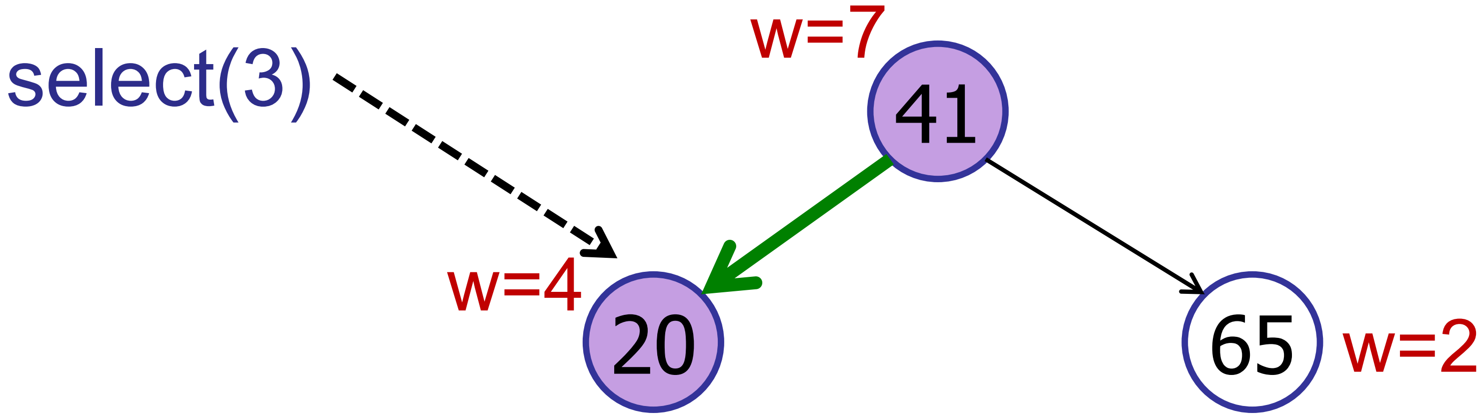
3 < left.weight + 1

Go left!



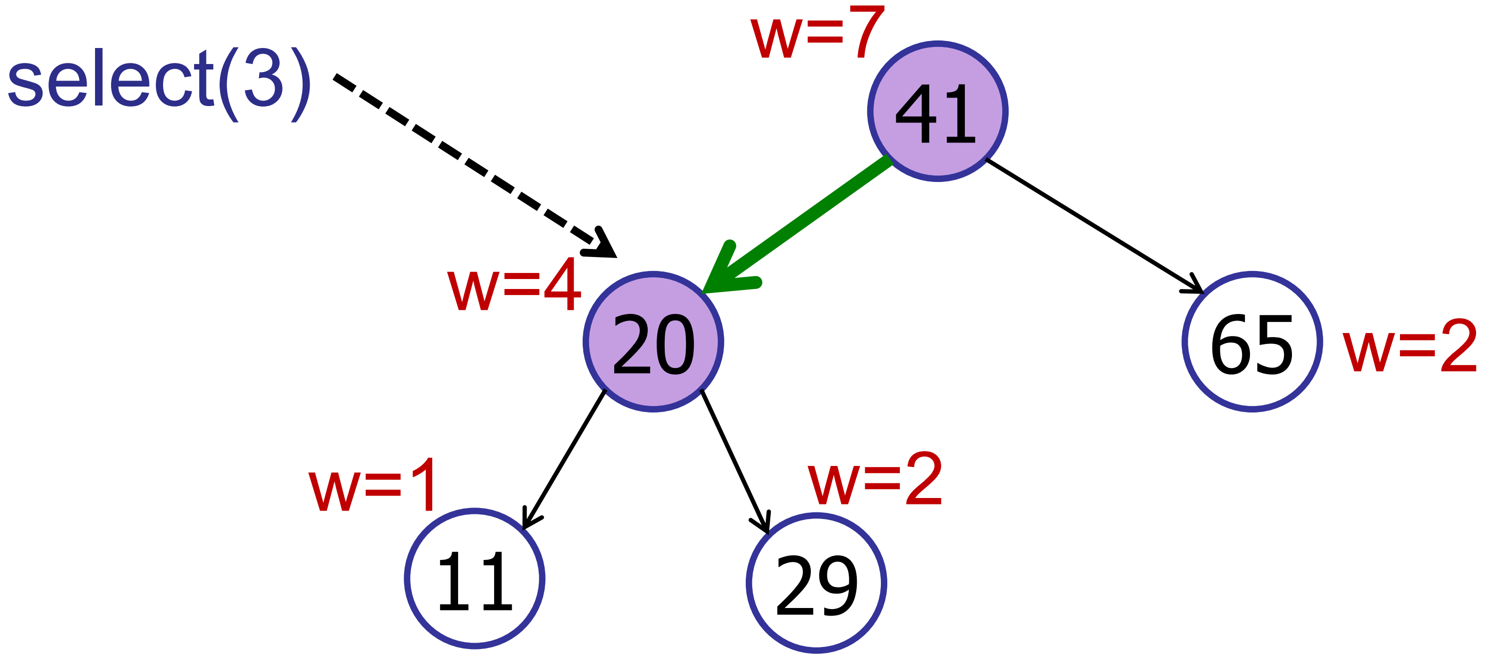
Dynamic Order Statistics

Example: select(3)



Dynamic Order Statistics

Example: select(3)

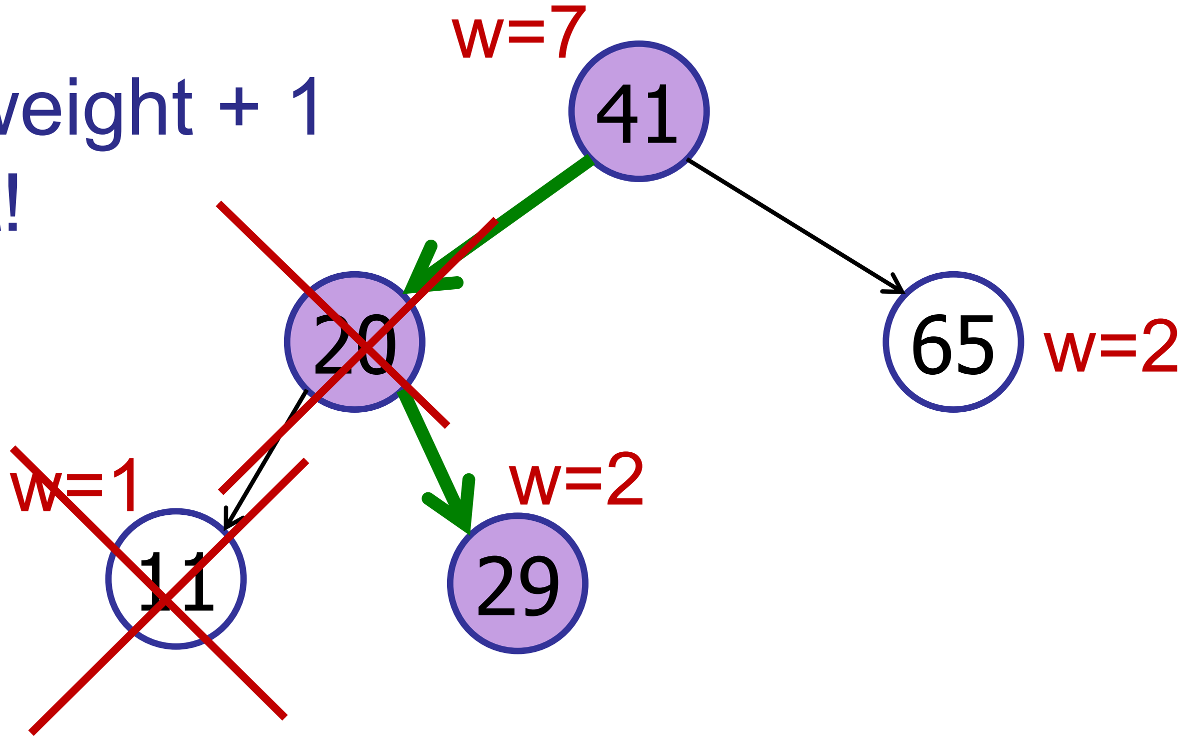


Dynamic Order Statistics

Example: select(3)

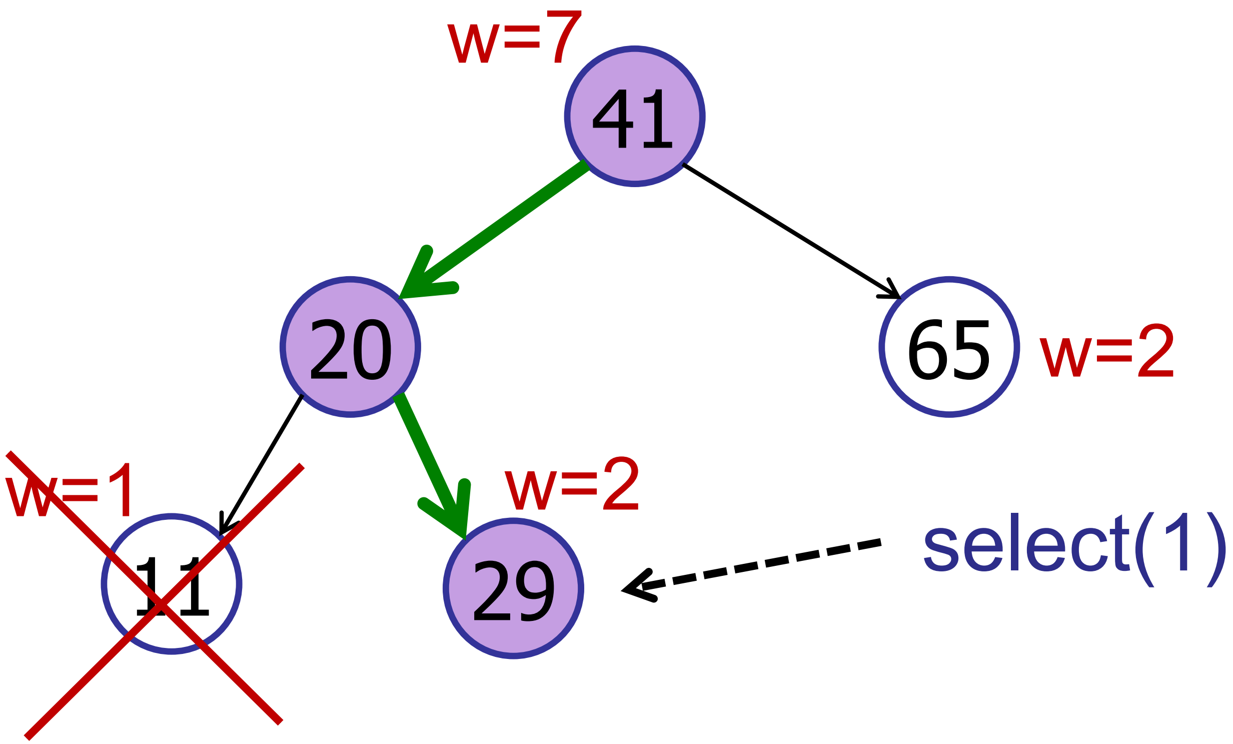
3 > left.weight + 1

Go right!



Dynamic Order Statistics

Example: select(3)



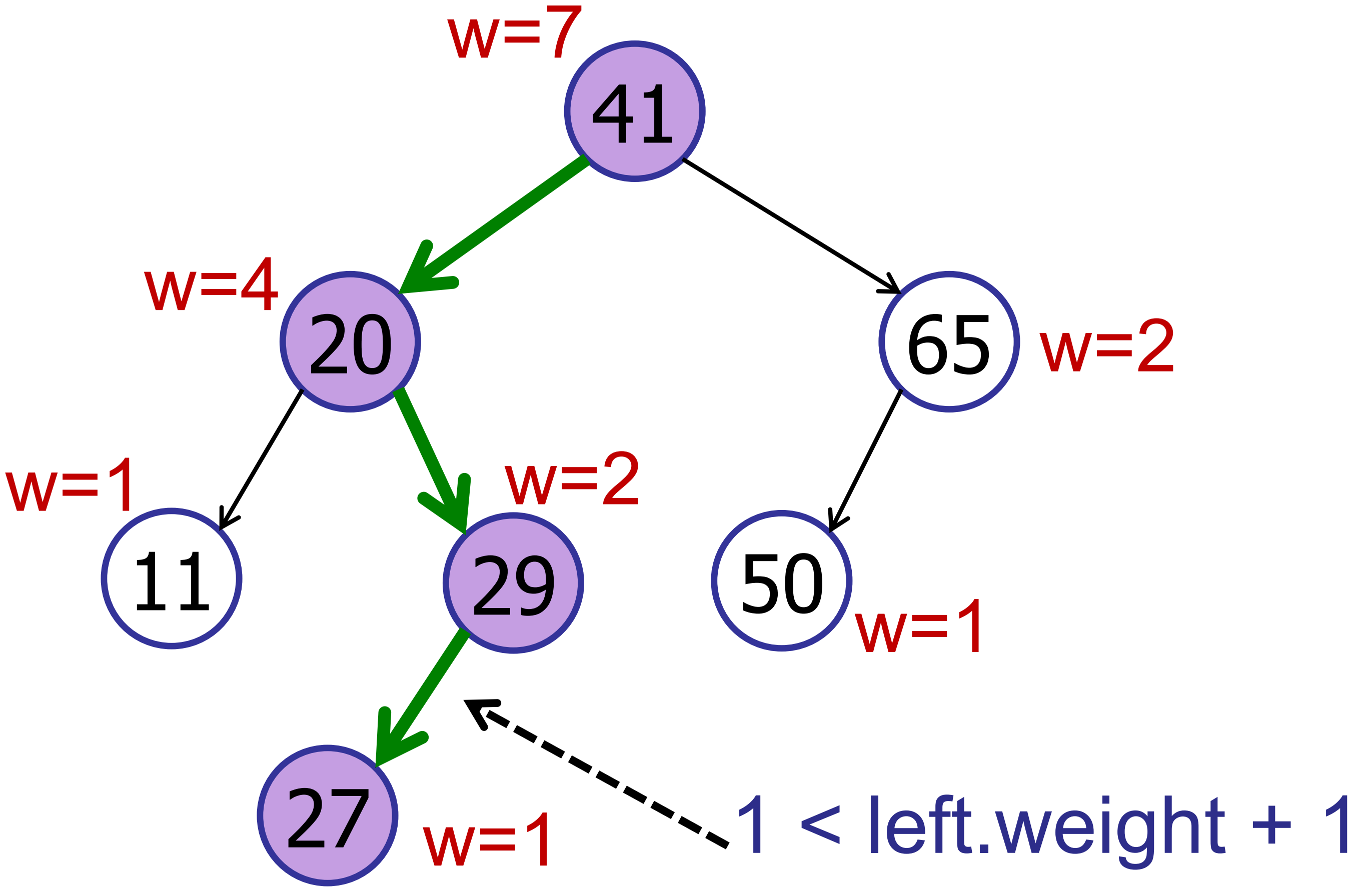
Item to select:

3 – (left.weight + 1) =

3 – (1 + 1) = 1

Dynamic Order Statistics

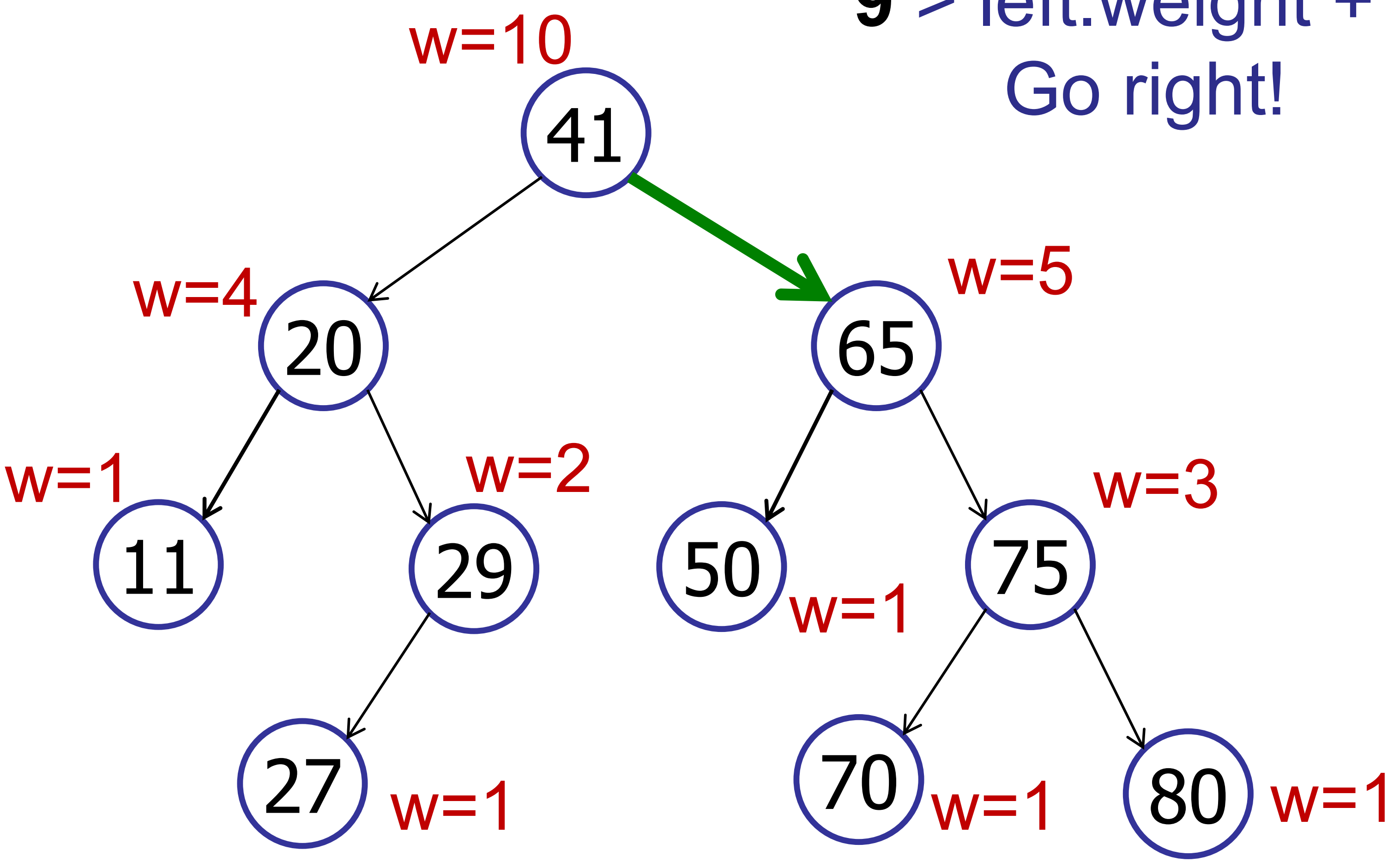
Example: select(3)



1 < left.weight + 1
Go left!

Dynamic Order Statistics

Example: select(9)

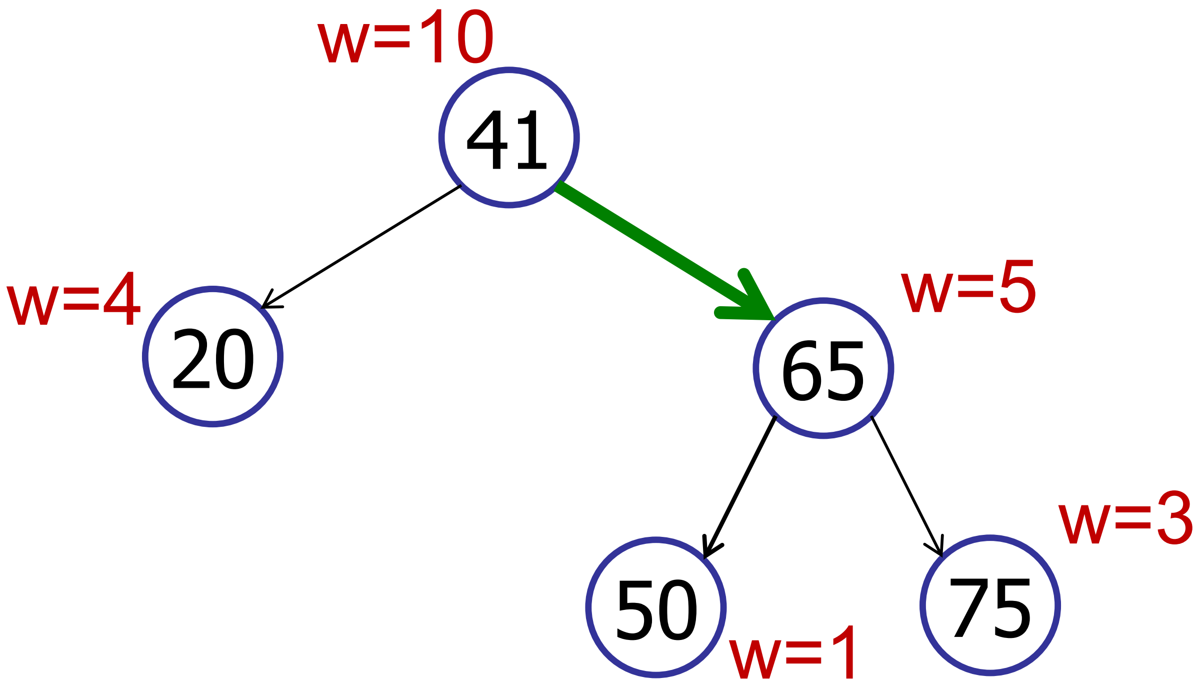


9 > left.weight + 1

Go right!

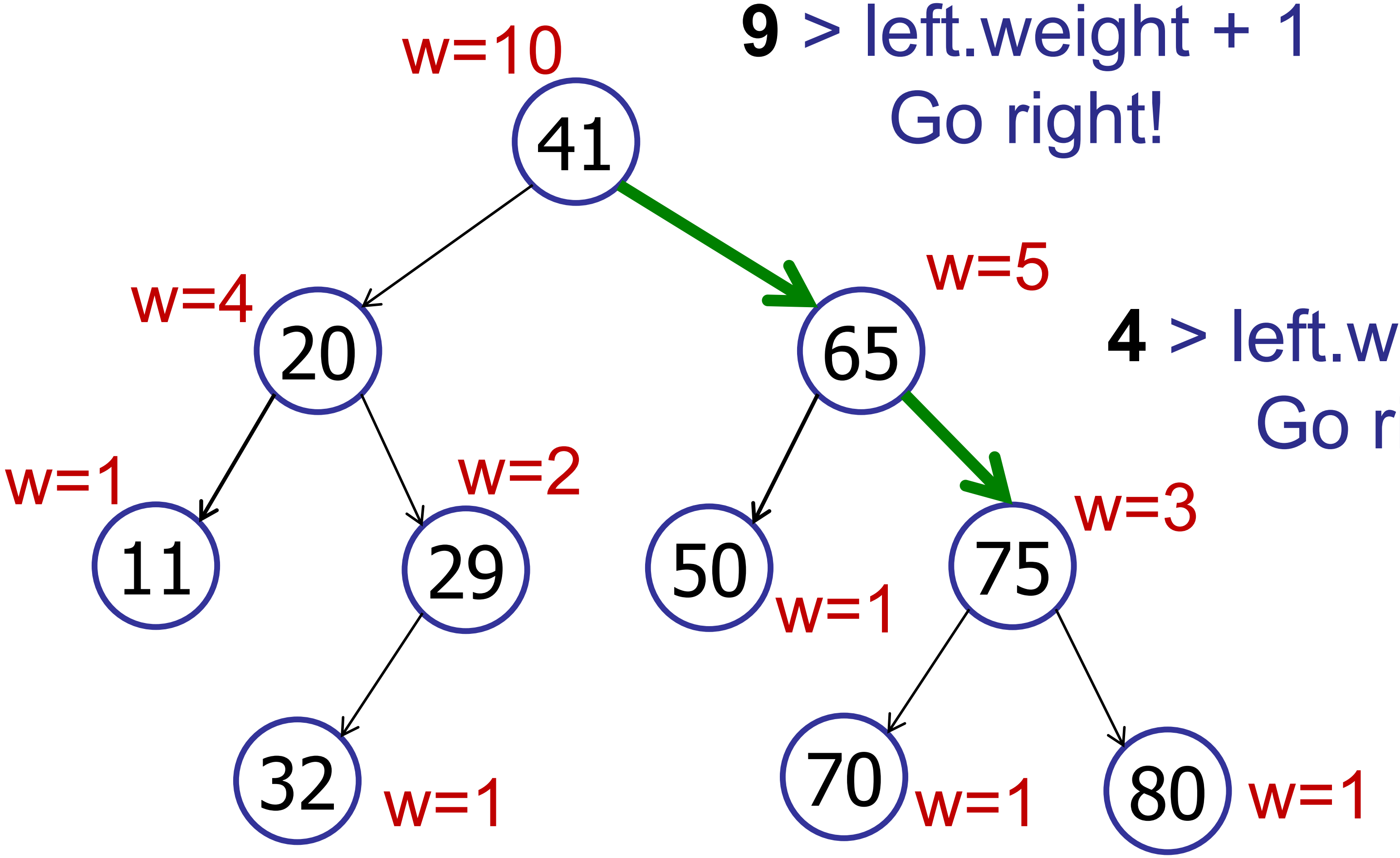
select(9)

1. Go left at 65
2. ✓ Go right at 65
3. Stop at 65
4. I’m confused



Dynamic Order Statistics

select(9)

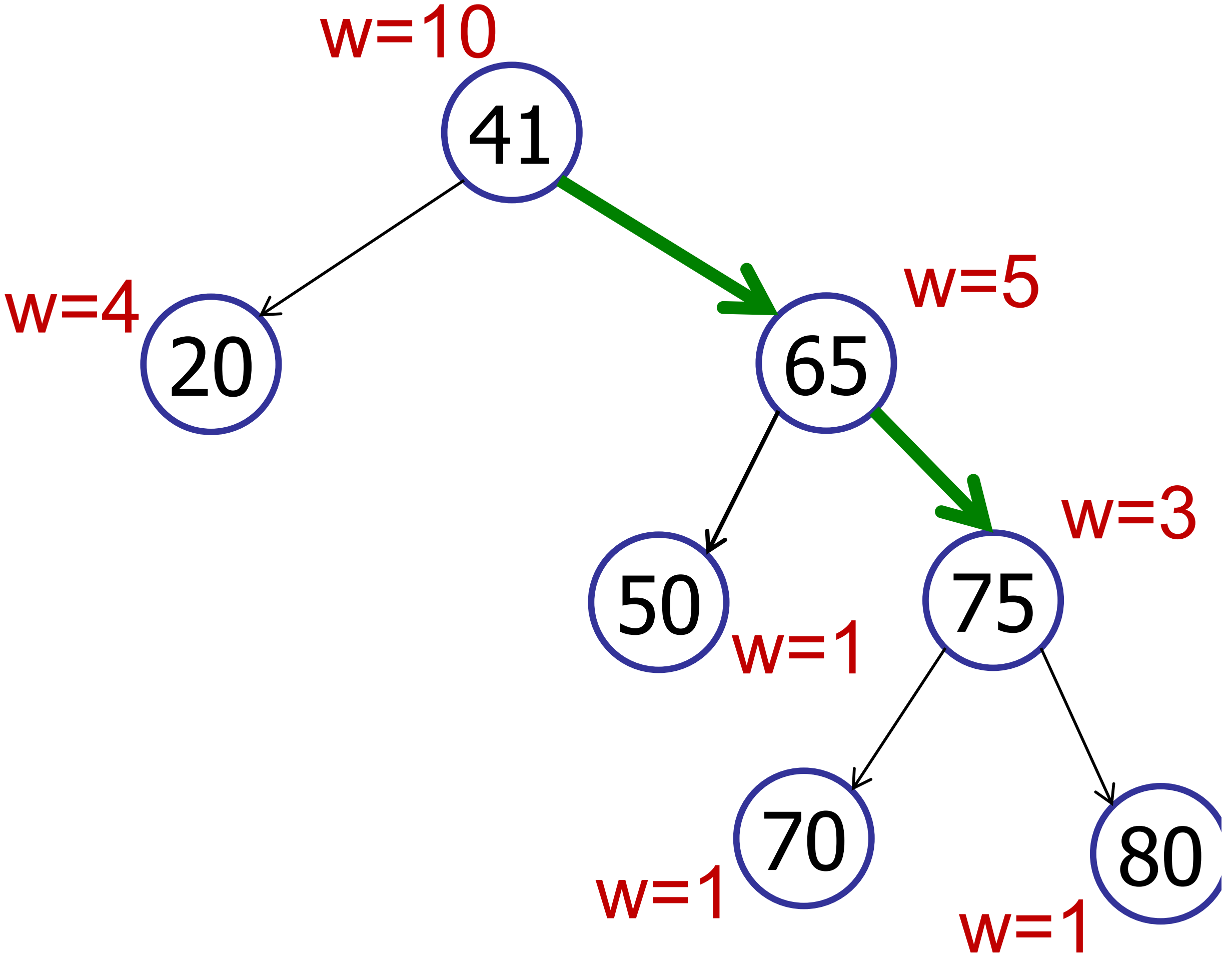


9 > left.weight + 1
Go right!

4 > left.weight + 1
Go right!

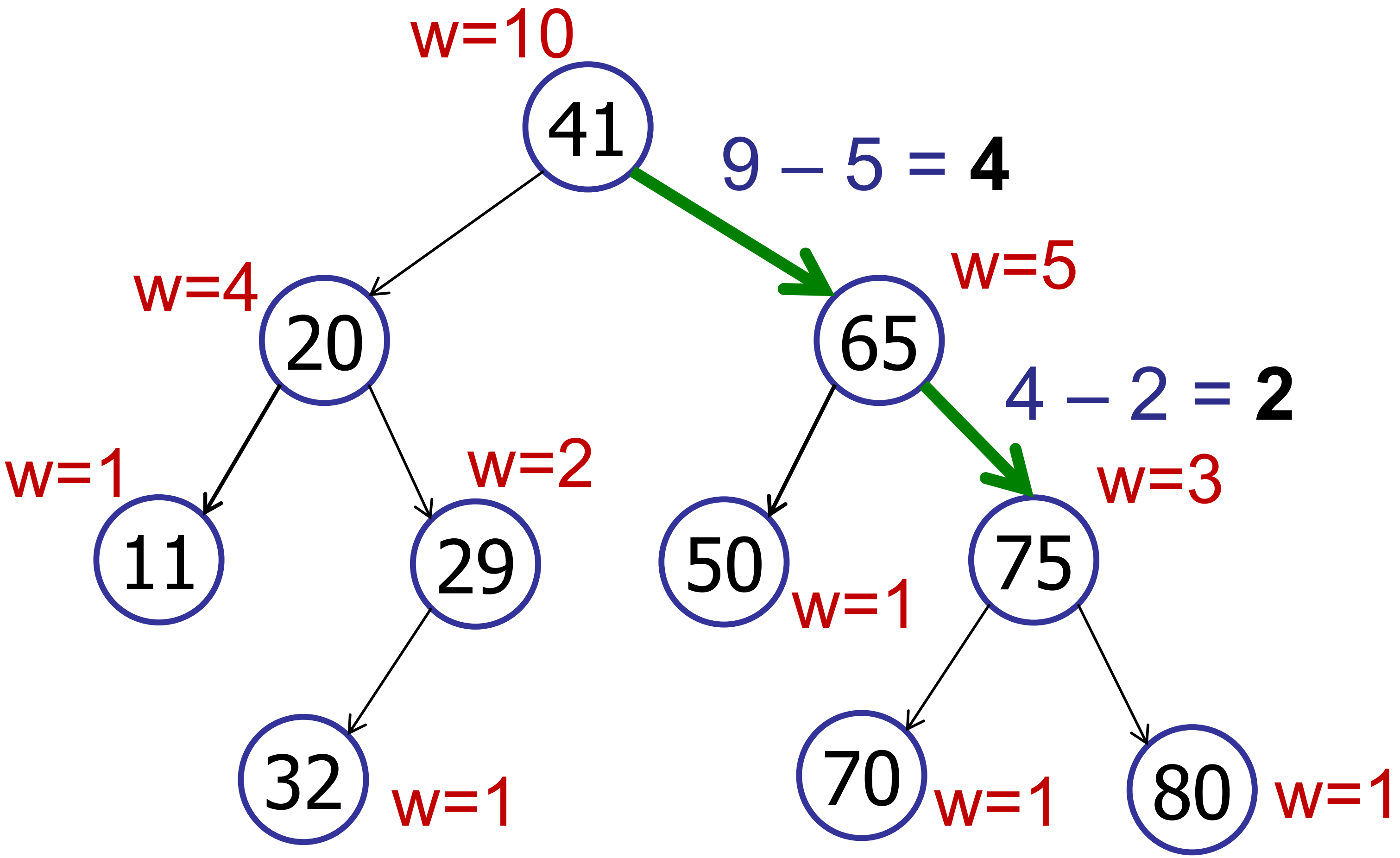
select(9)

1. Go left at 75
2. Go right at 75
3. ✓ Stop at 75
4. I’m confused



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

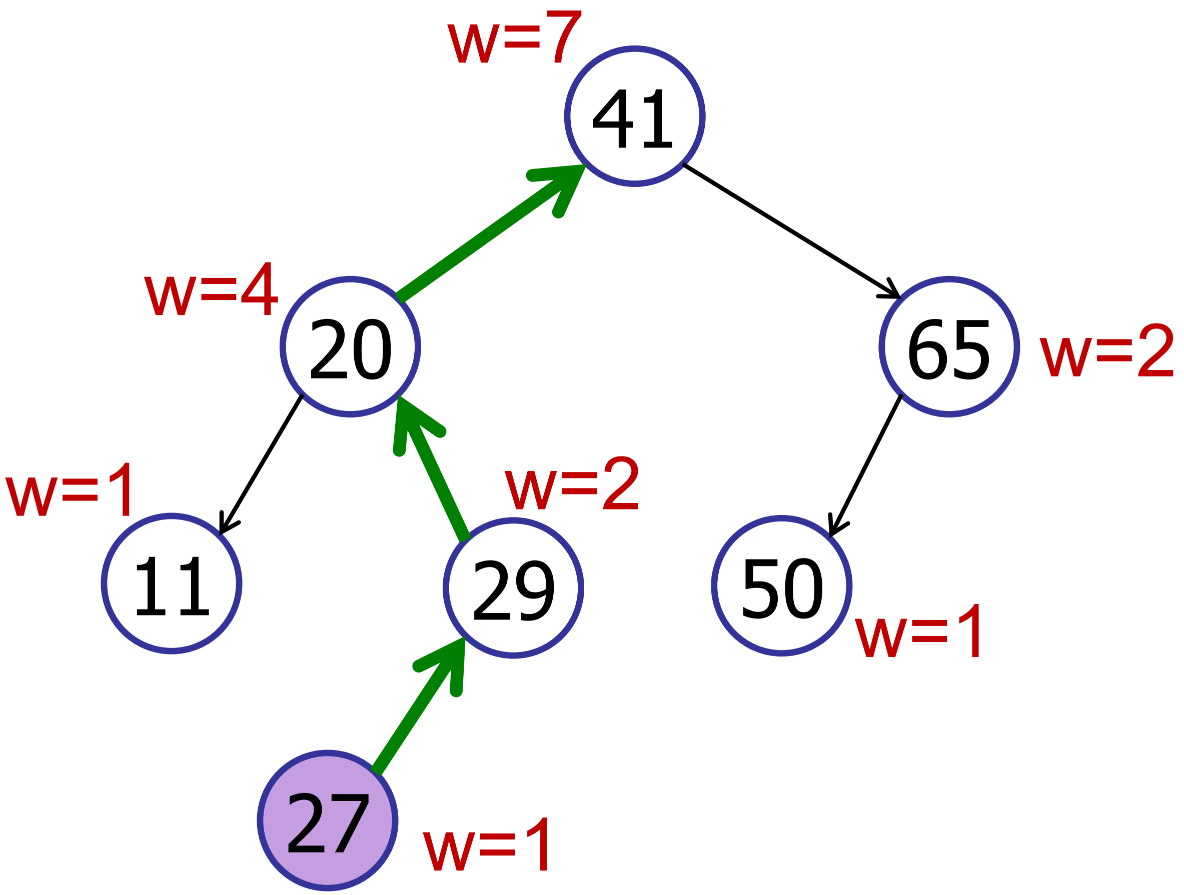
Example: find the 10th tallest student in the class.

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

Example: determine the percentile of Johnny’s height.
Is Johnny in the 10th percentile or the 90th percentile?

Dynamic Order Statistics

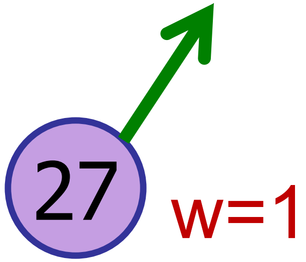
Example: rank(27)



rank = 1

Dynamic Order Statistics

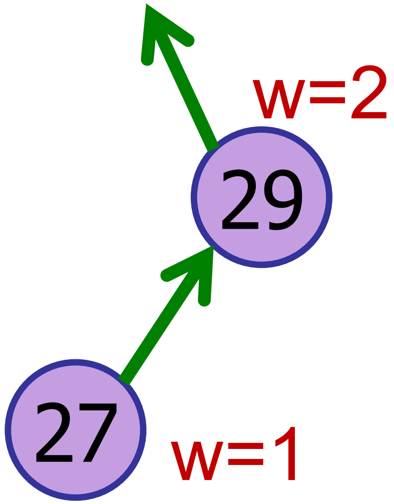
Example: rank(27)



rank = 1

Dynamic Order Statistics

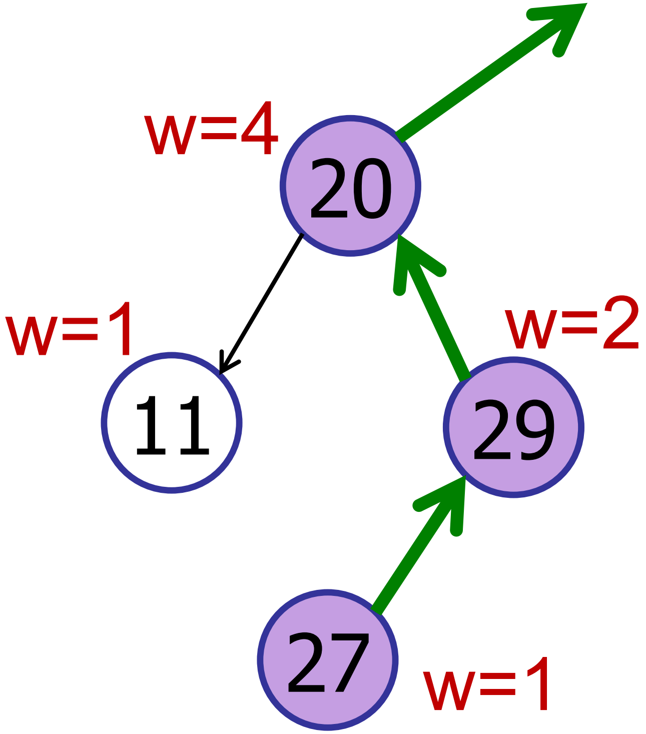
Example: rank(27)



rank = 1

Dynamic Order Statistics

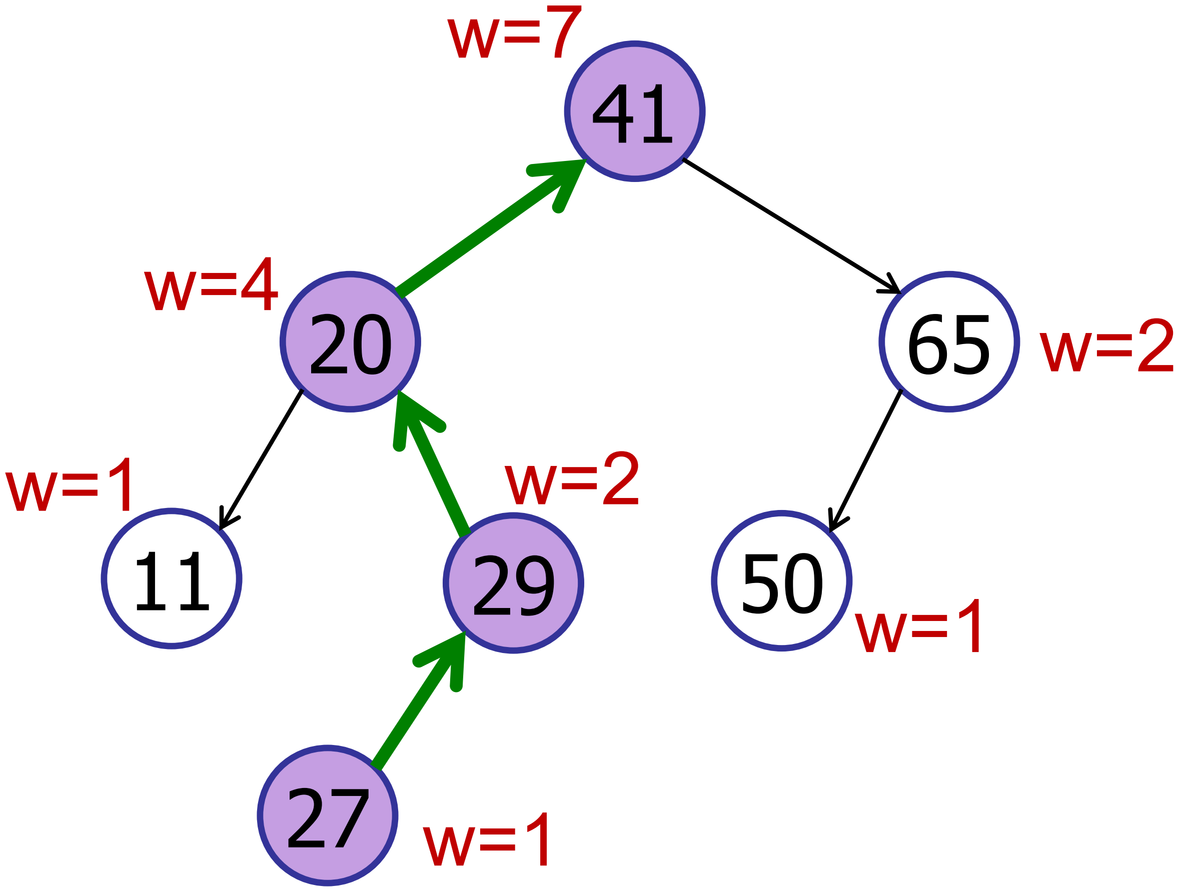
Example: rank(27)



rank = 1 + 2

Dynamic Order Statistics

Example: rank(27)



rank = 1 + 2 = 3

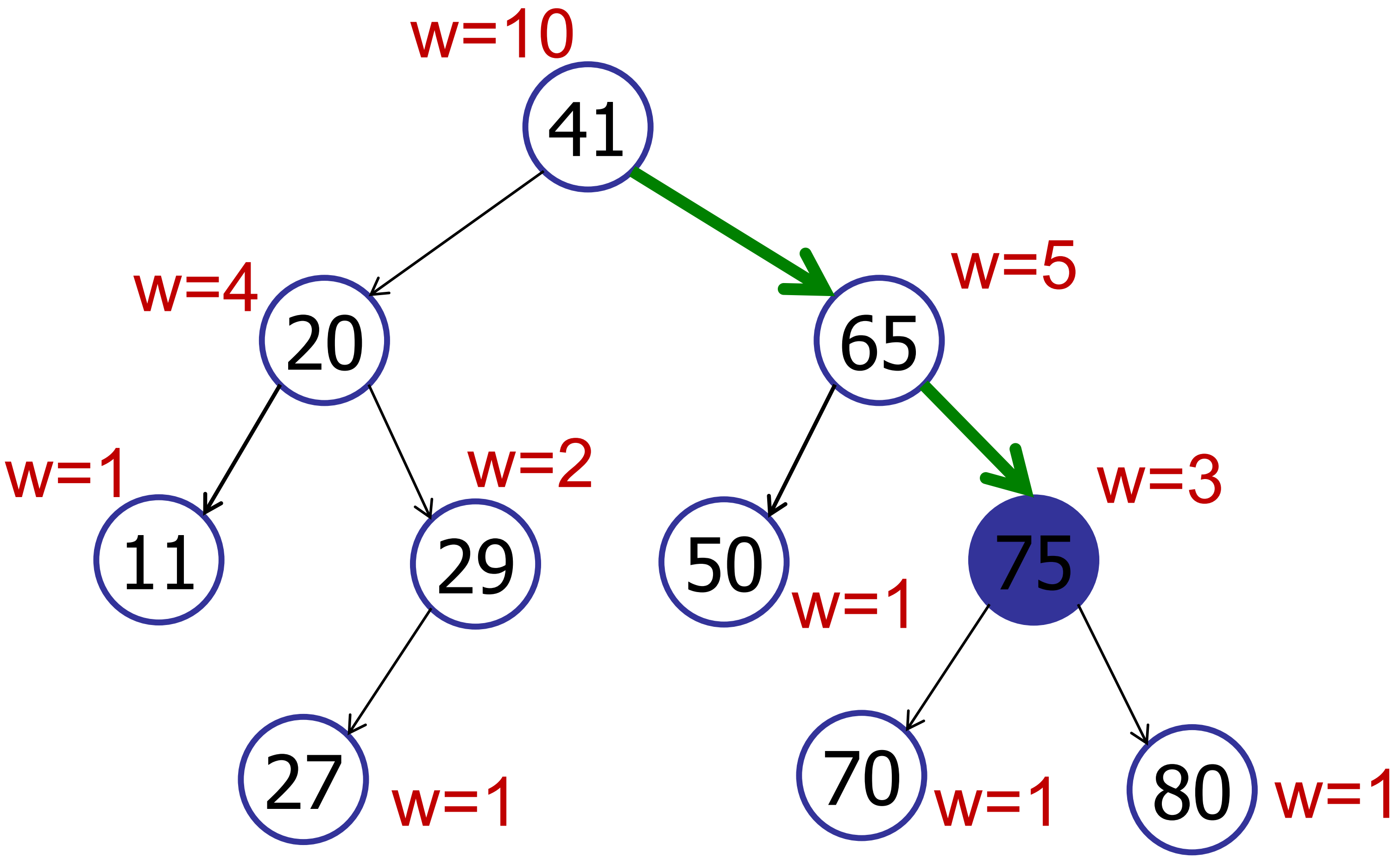
Dynamic Order Statistics

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

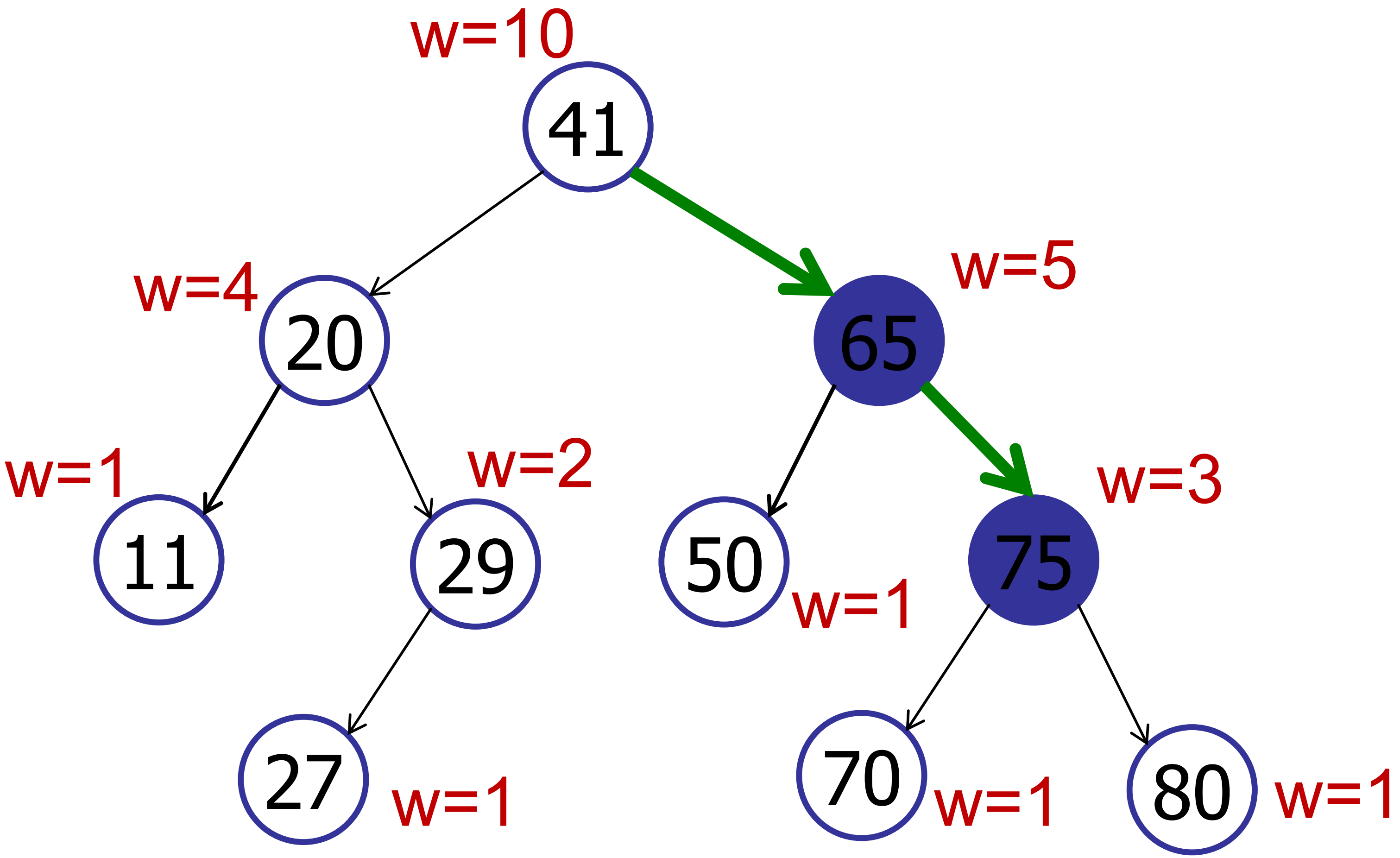
rank(75)



rank = 2

Dynamic Order Statistics

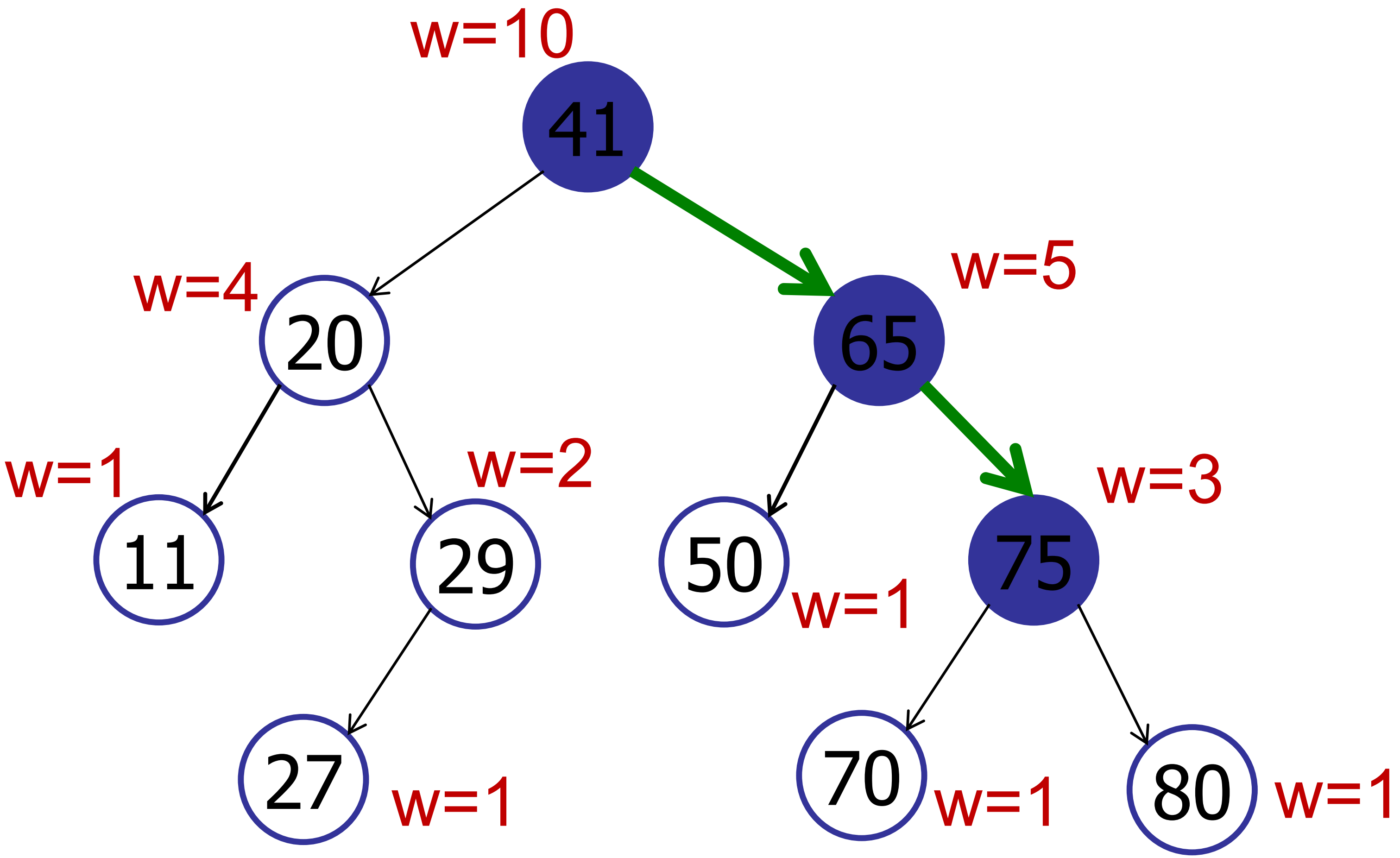
rank(75)



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.

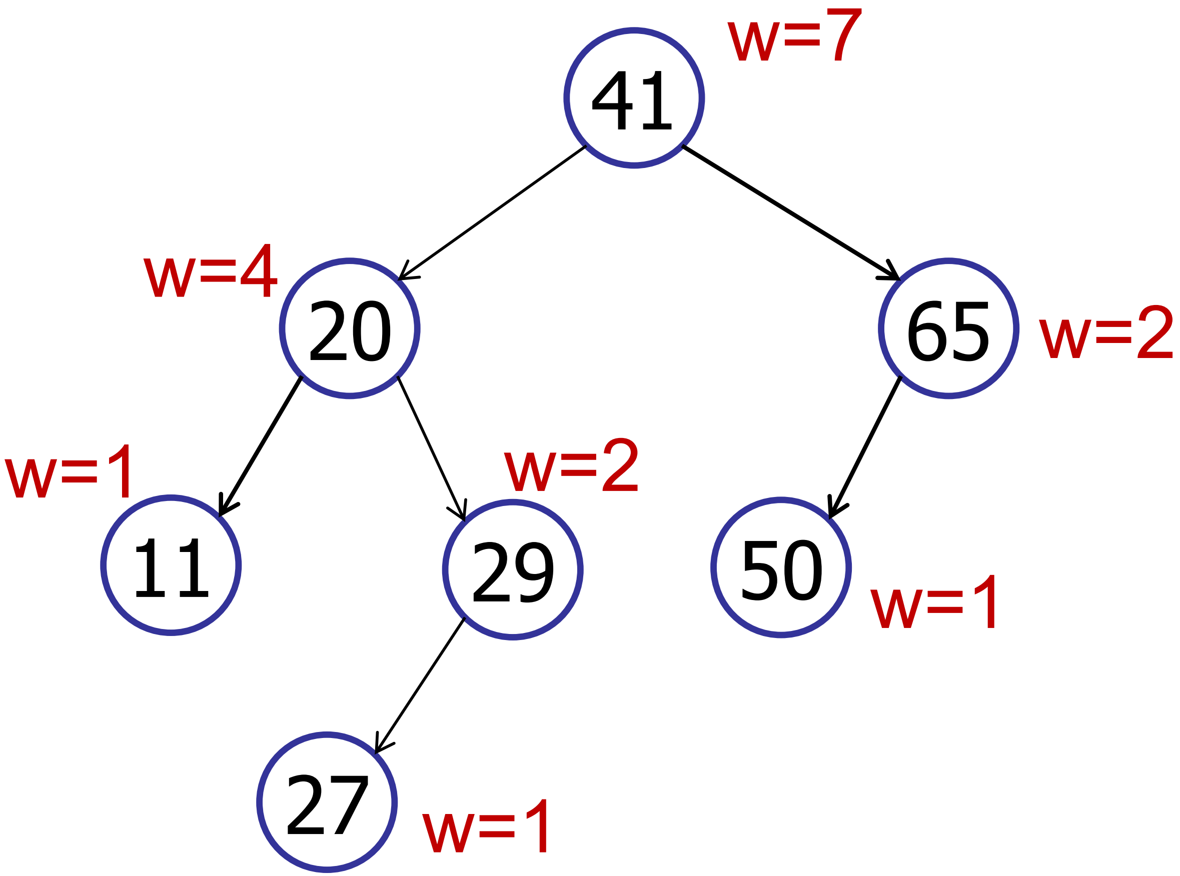
    Update weights as needed

4. Develop new operations using the new info.

    Select and Rank

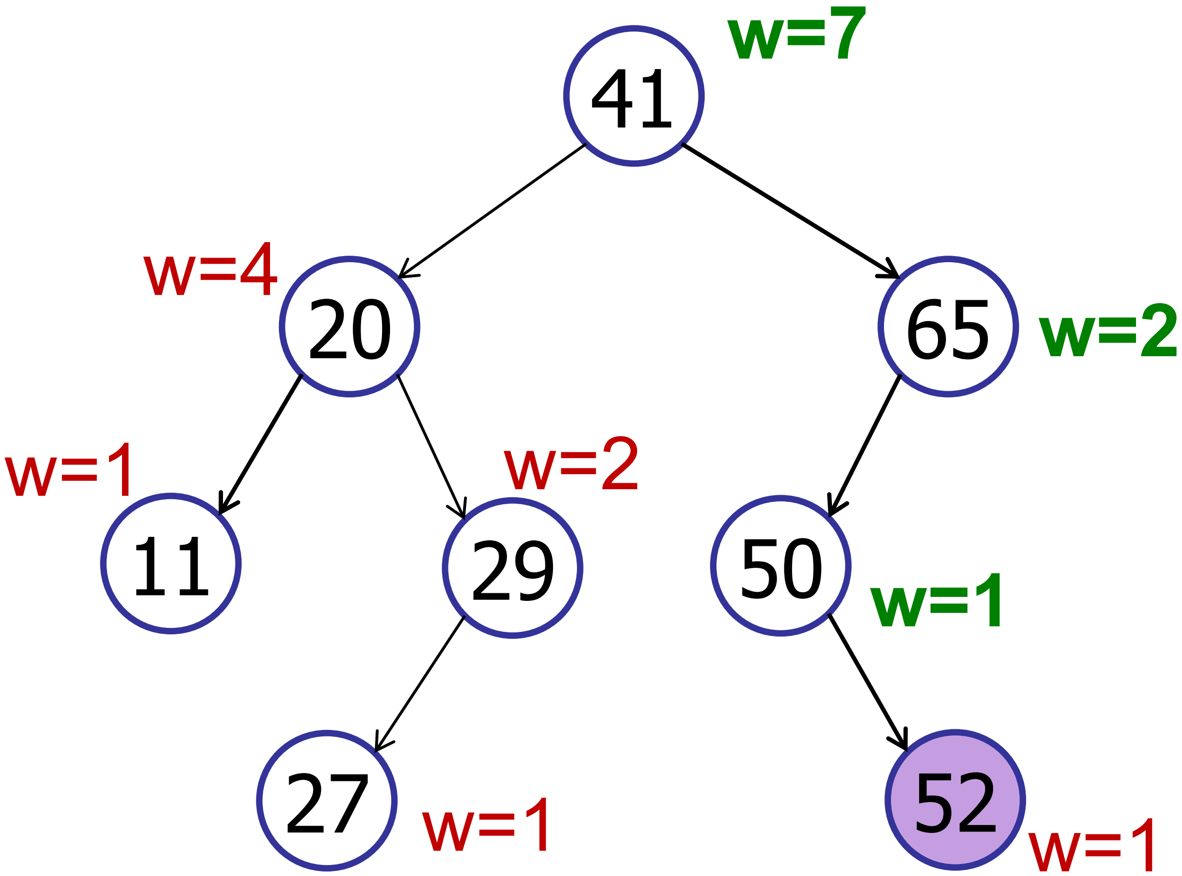
Augmented Trees

Maintain weight during insertions:



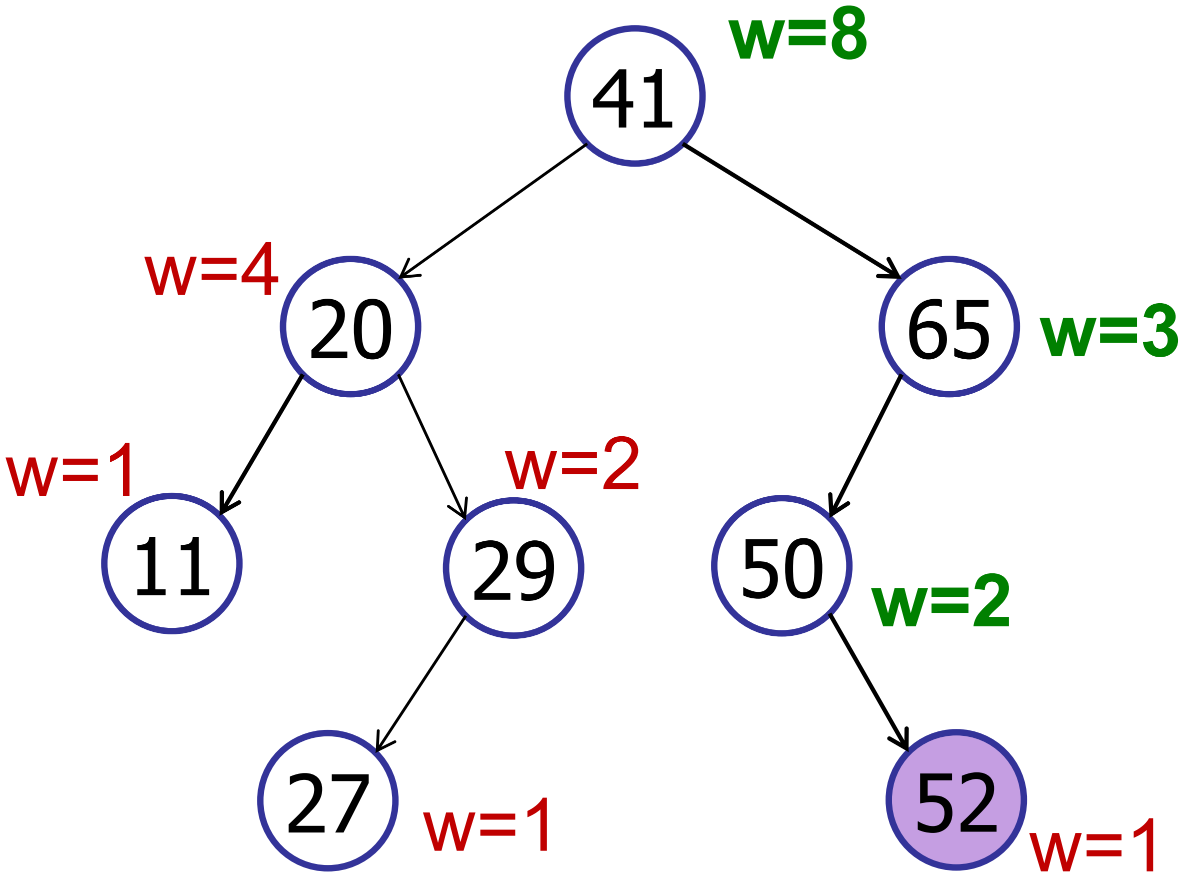
Augmented Trees

Maintain weight during insertions:



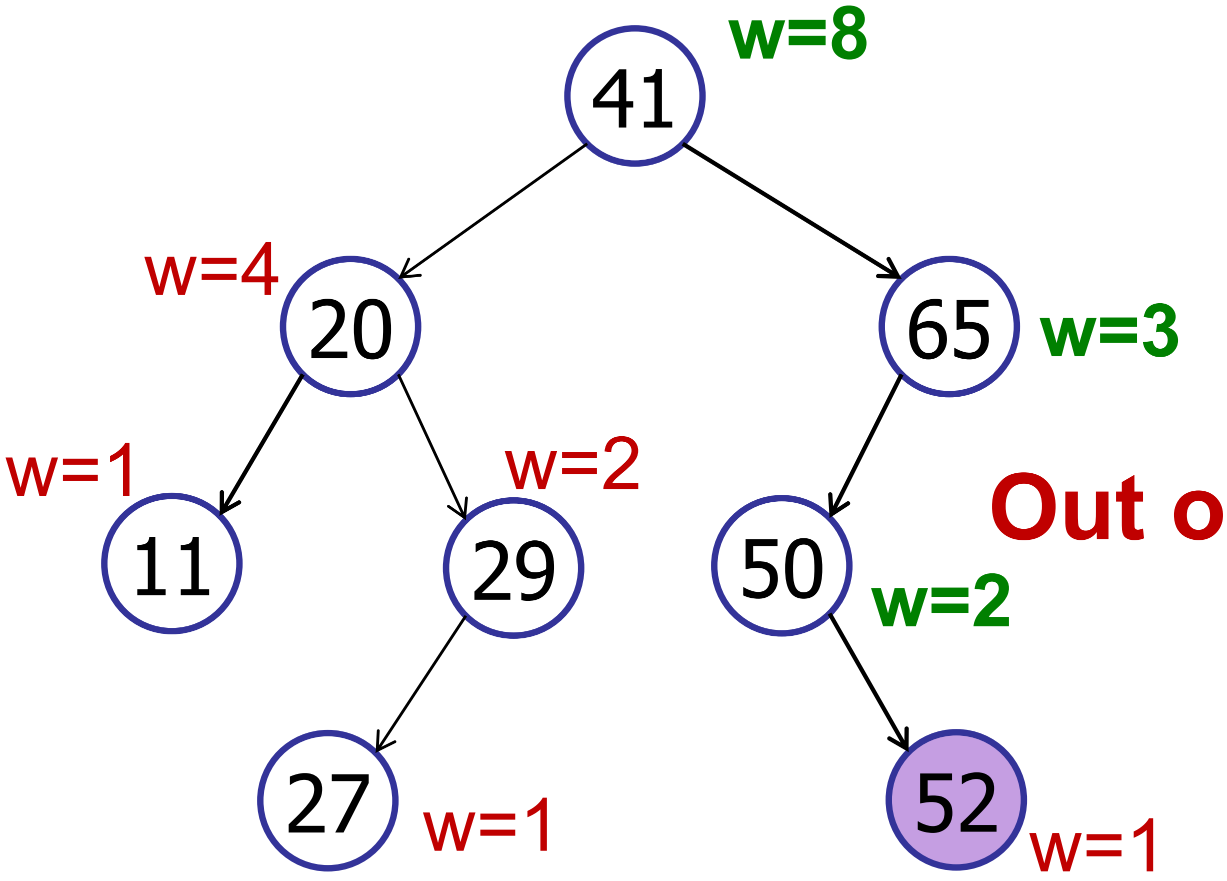
Augmented Trees

Maintain weight during insertions:



Augmented Trees

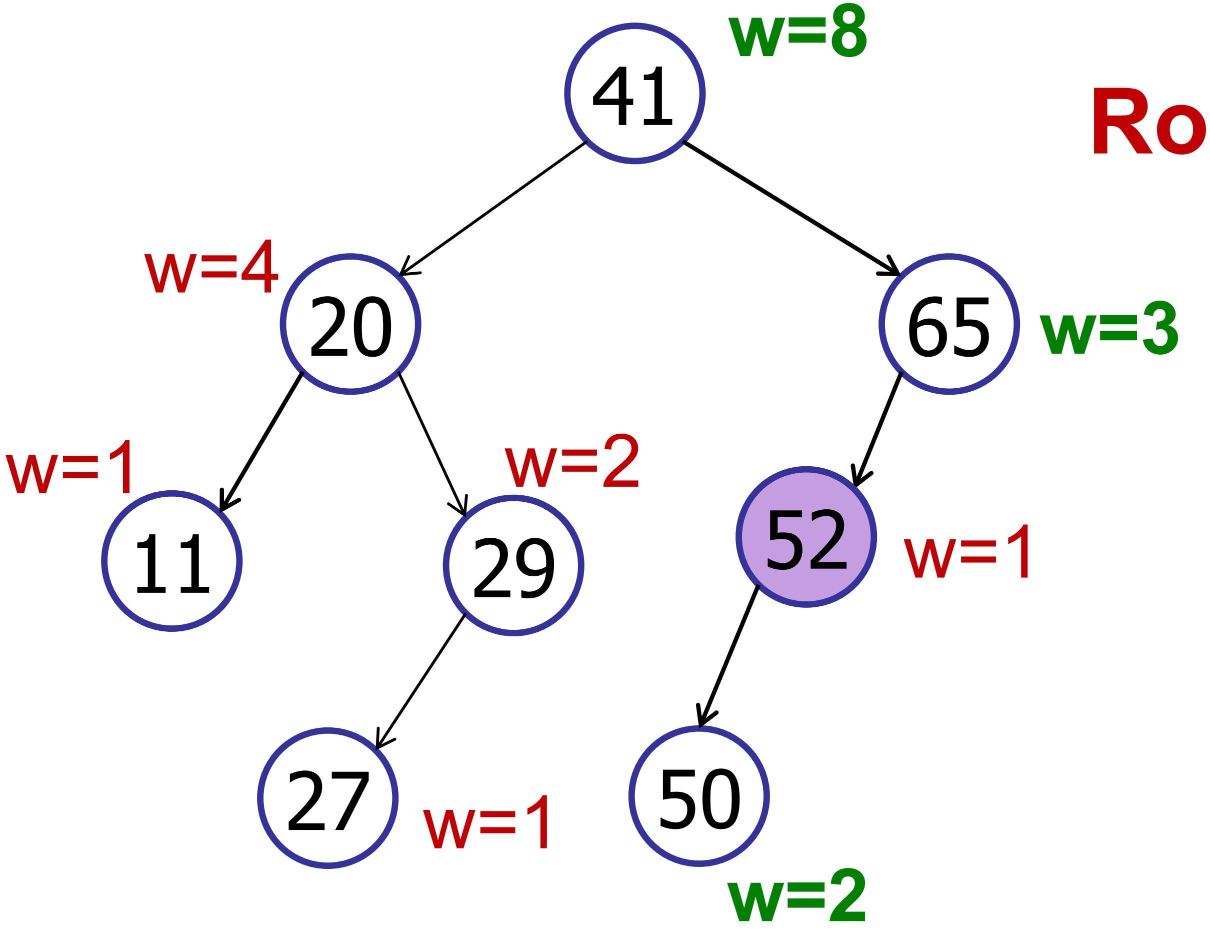
Maintain weight during insertions:



Out of balance!

Augmented Trees

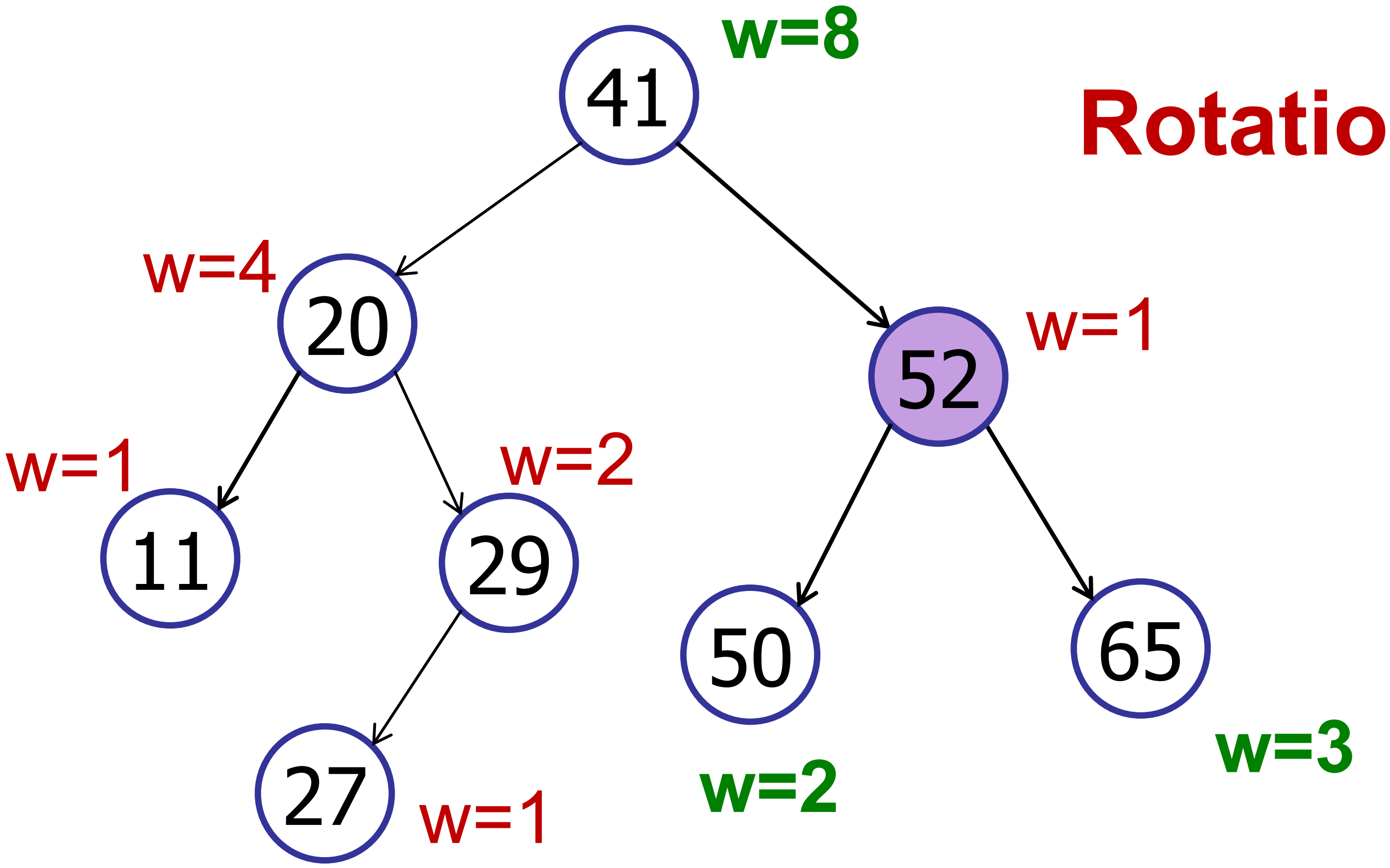
Maintain weight during insertions:



Rotation 1:

Augmented Trees

Maintain weight during insertions:

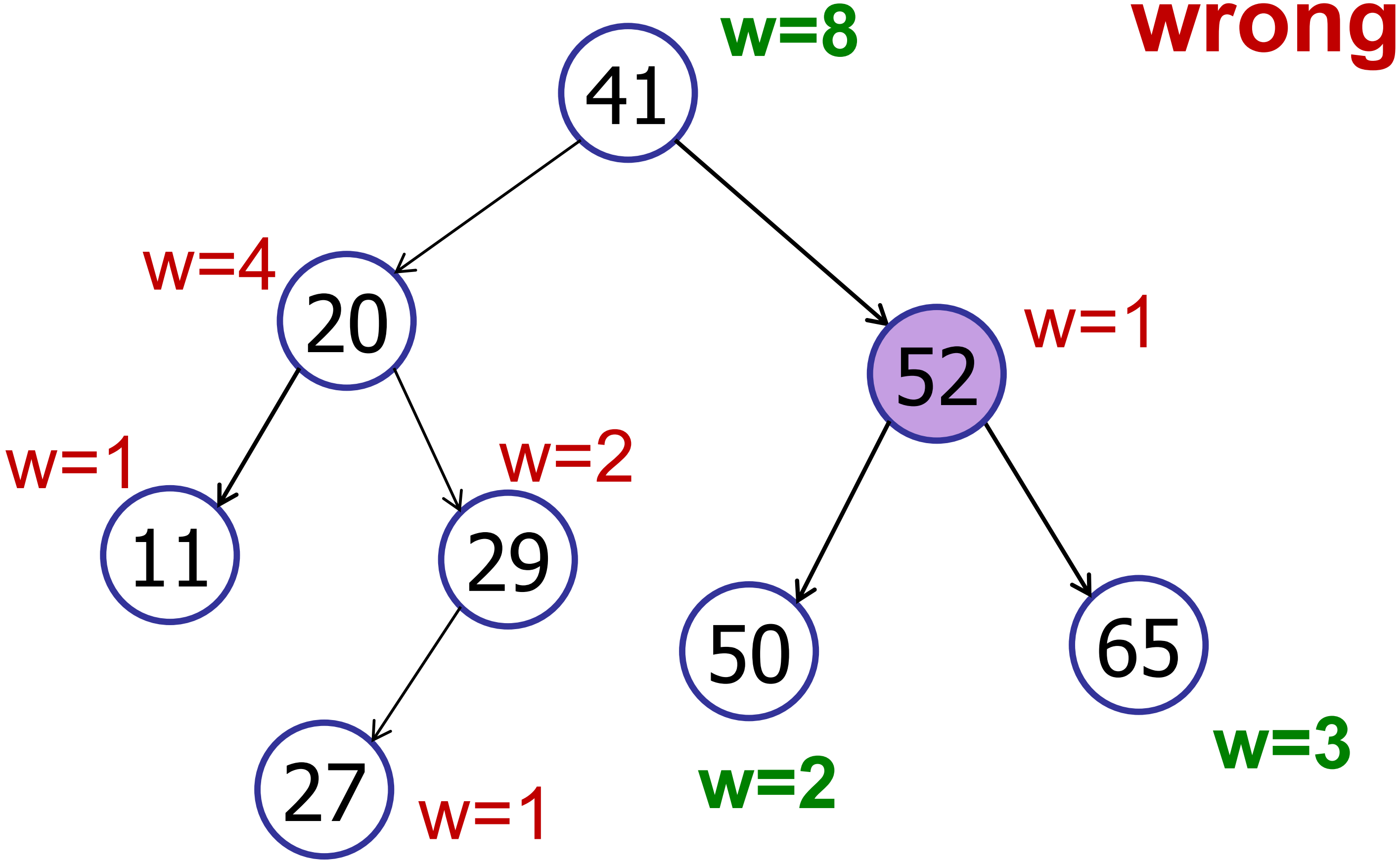


Rotation 2:

Augmented Trees

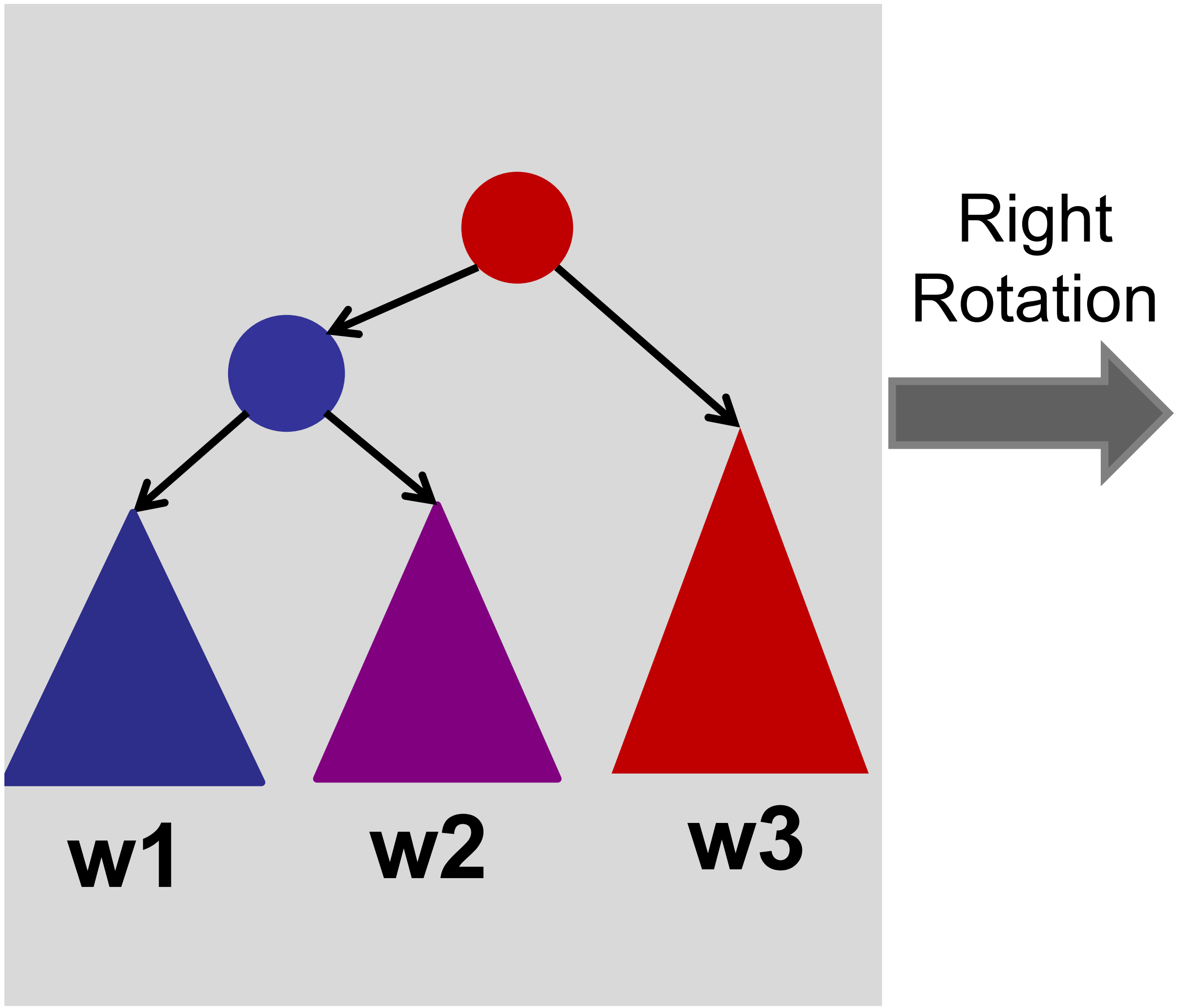
How to update weights on rotation?

Weights all wrong!



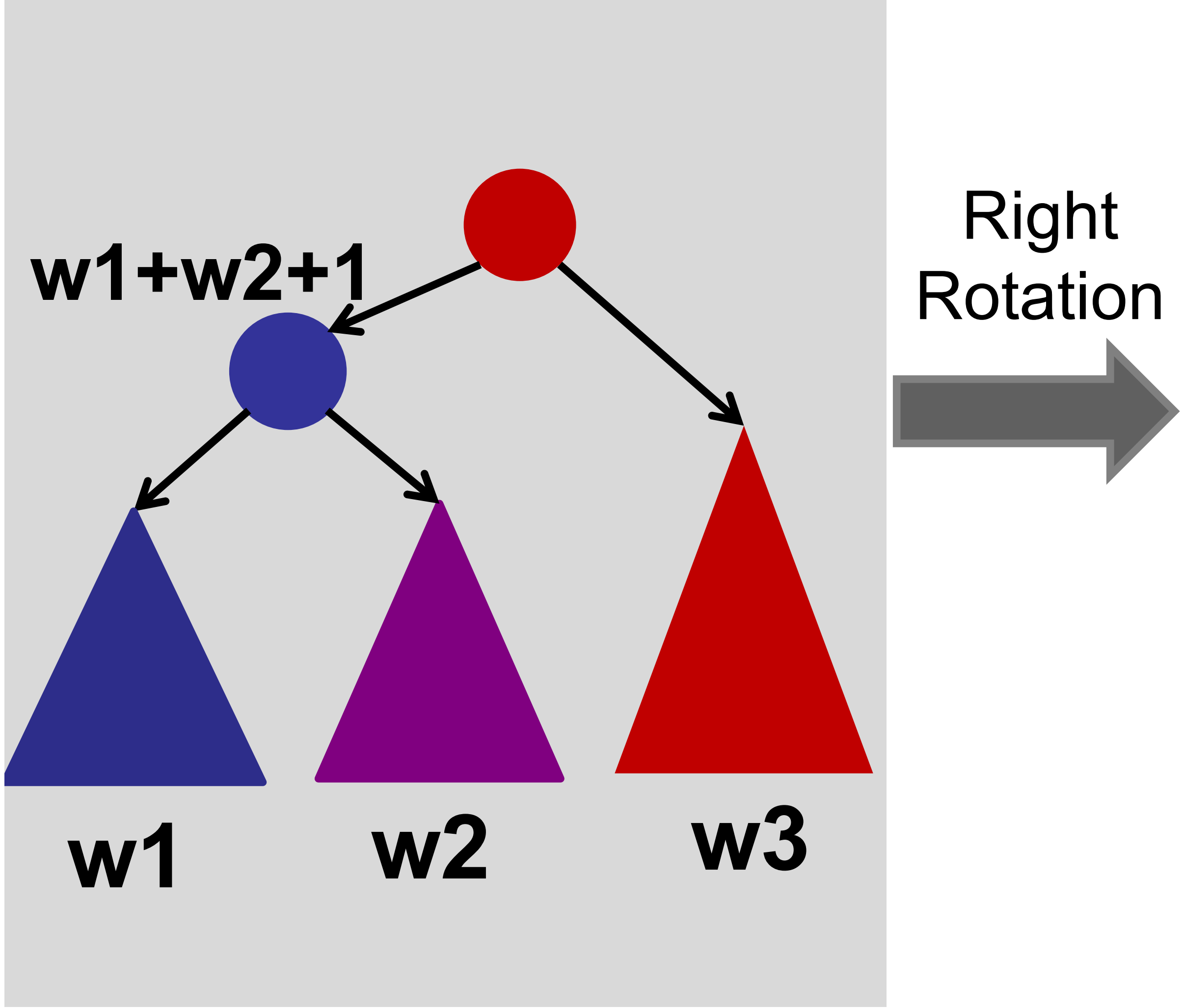
Augmented Trees

Maintain weight during rotations:



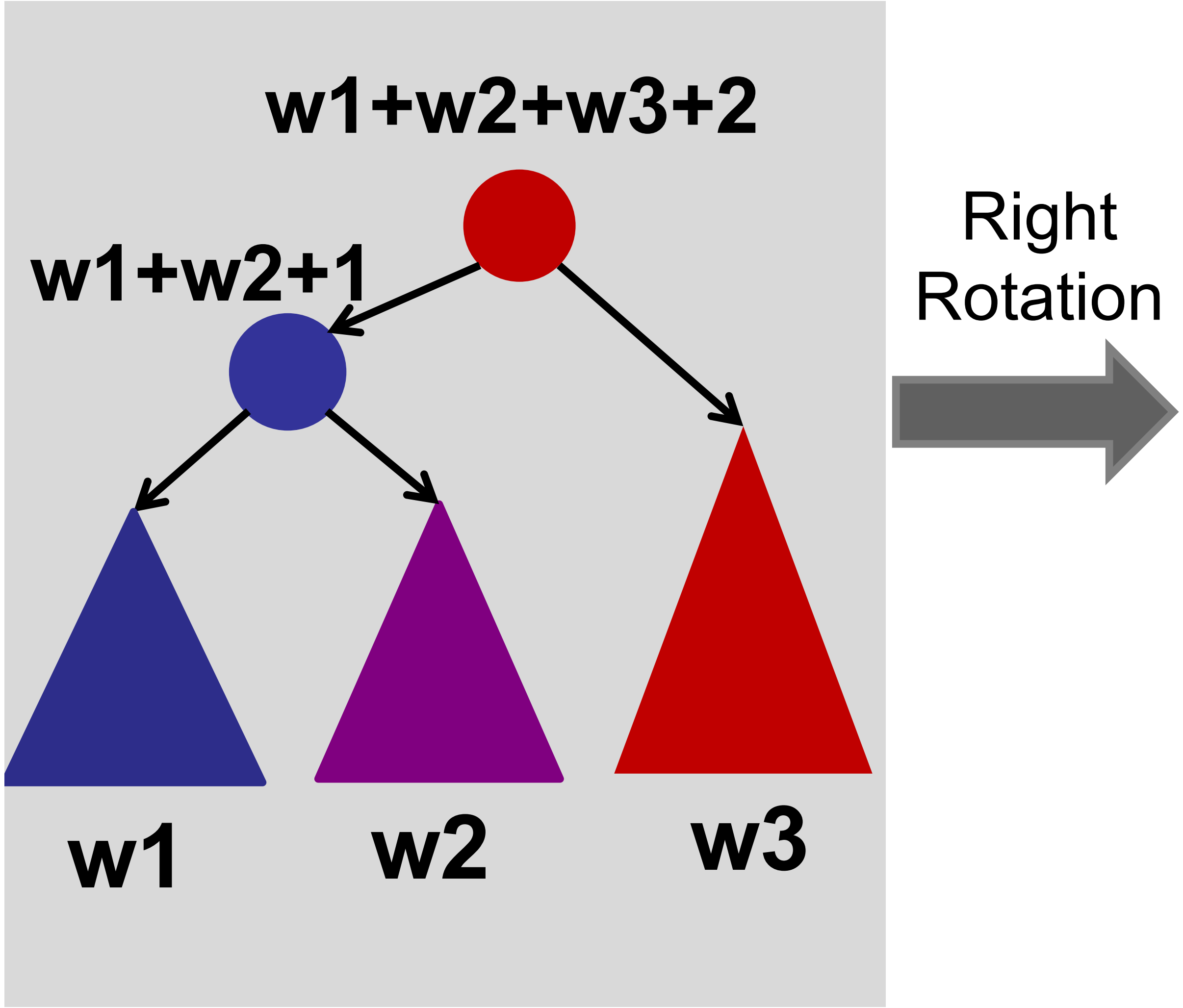
Augmented Trees

Maintain weight during rotations:



Augmented Trees

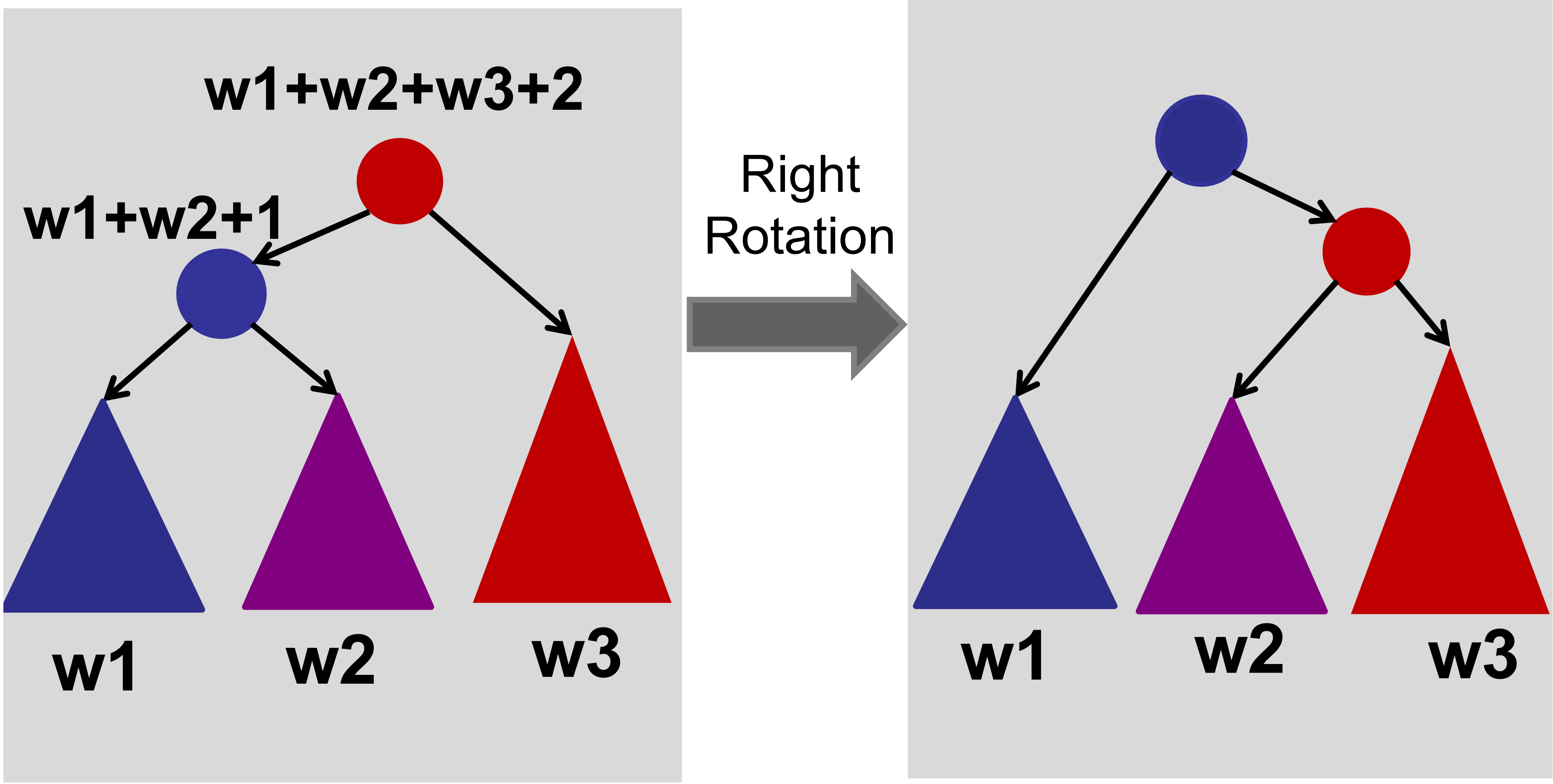
Maintain weight during rotations:



Right Rotation

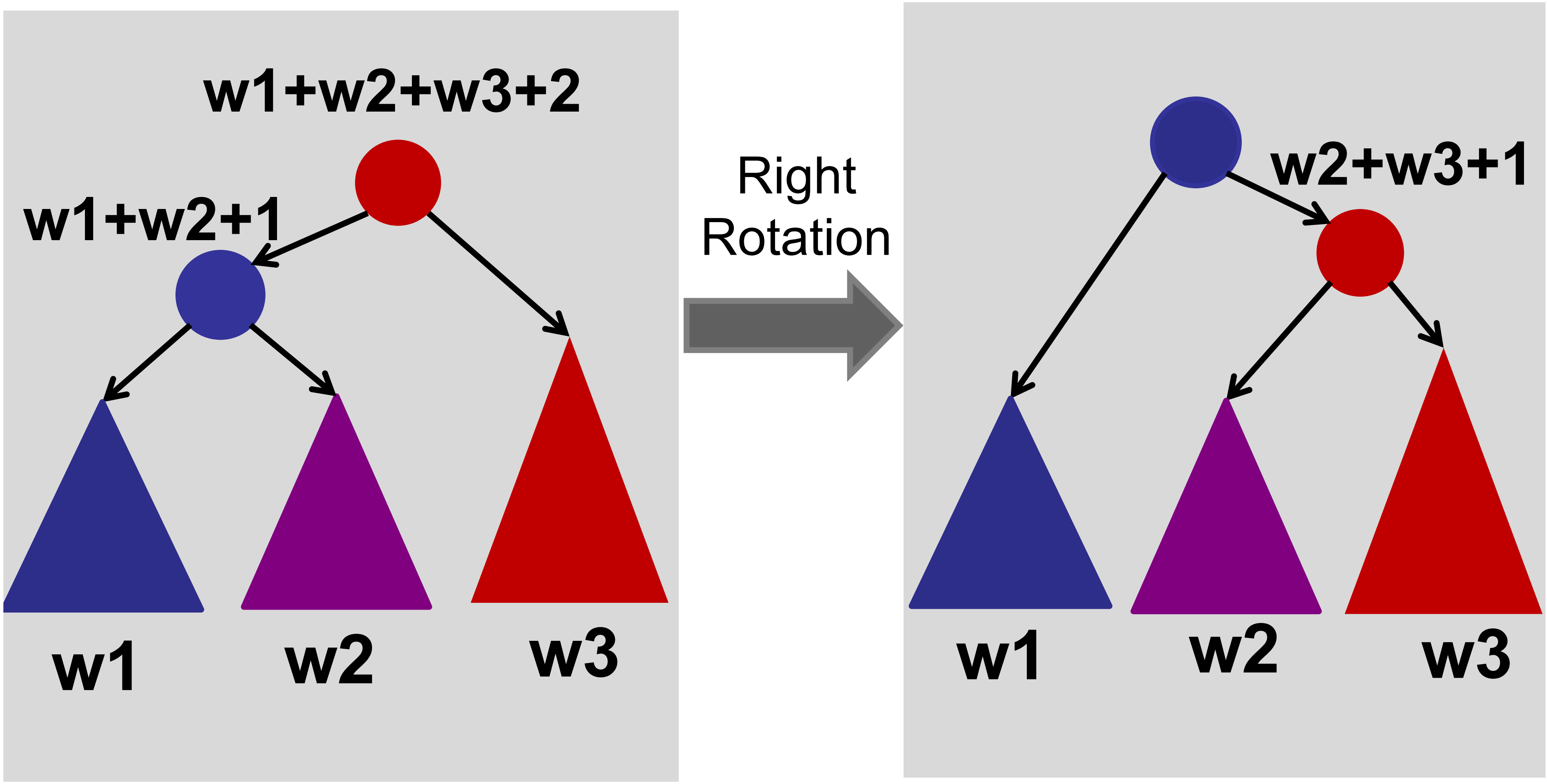
Augmented Trees

Maintain weight during rotations:



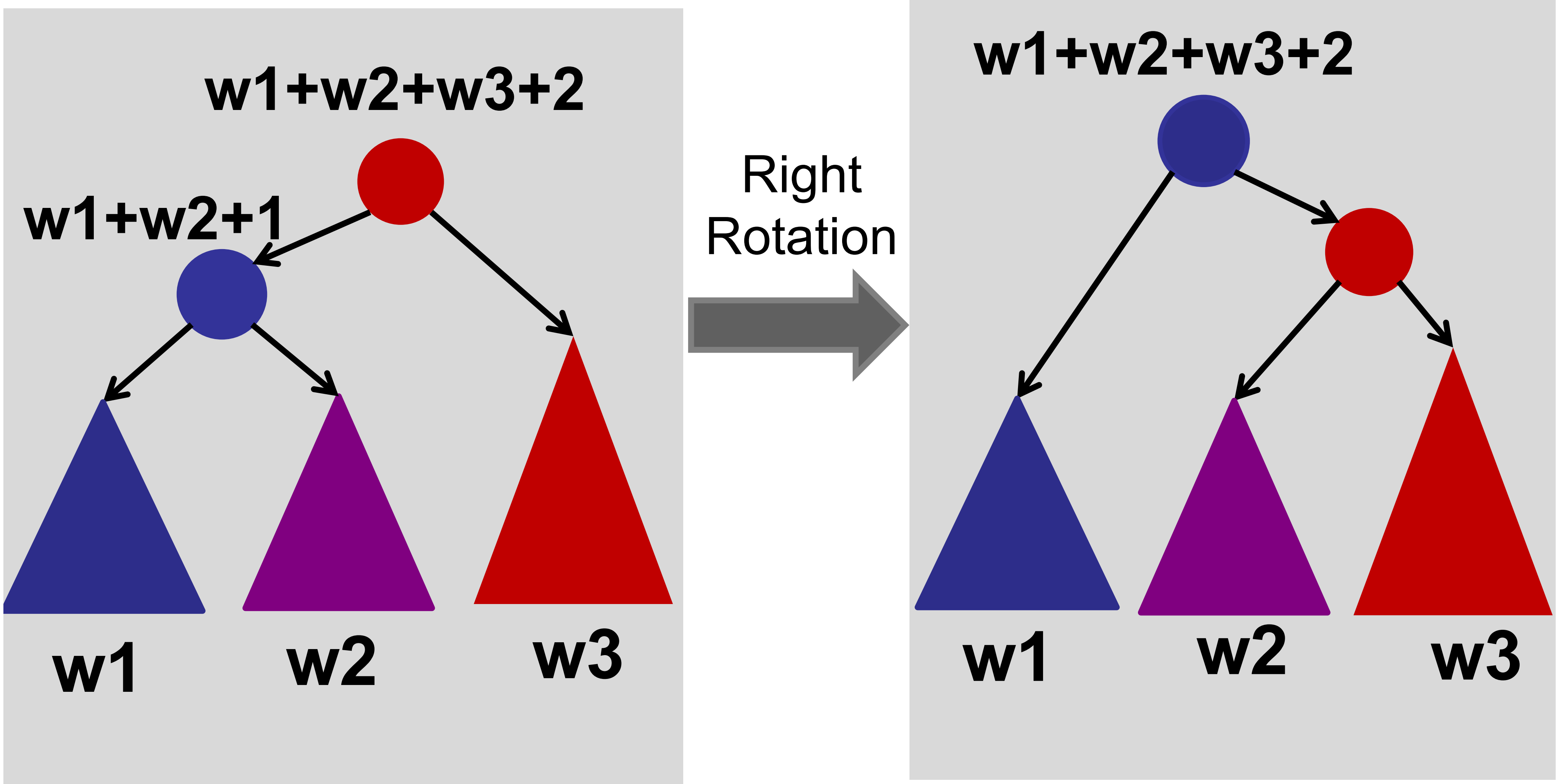
Augmented Trees

Maintain weight during rotations:



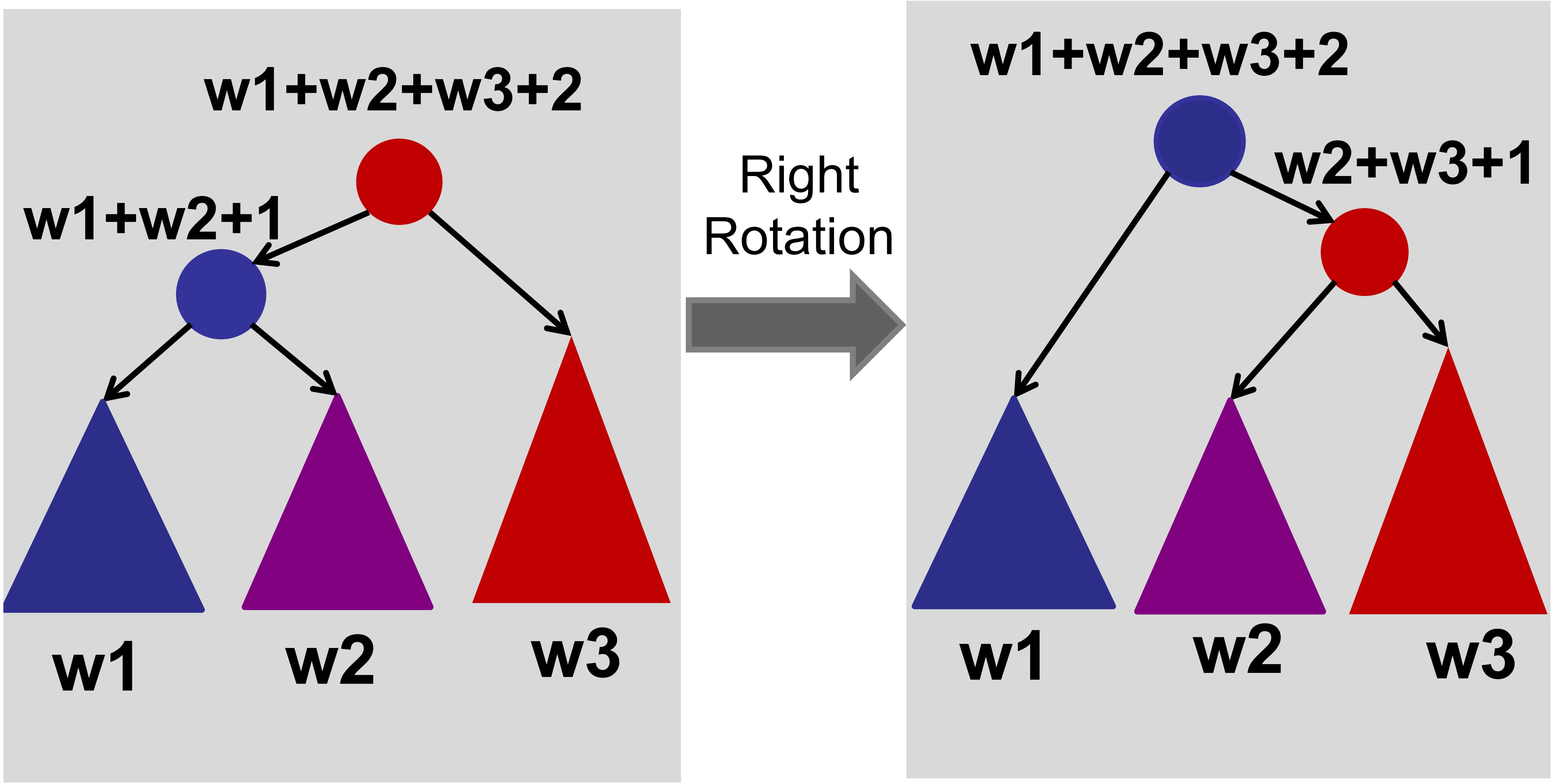
Augmented Trees

Maintain weight during rotations:



Augmented Trees

Maintain weight during rotations:

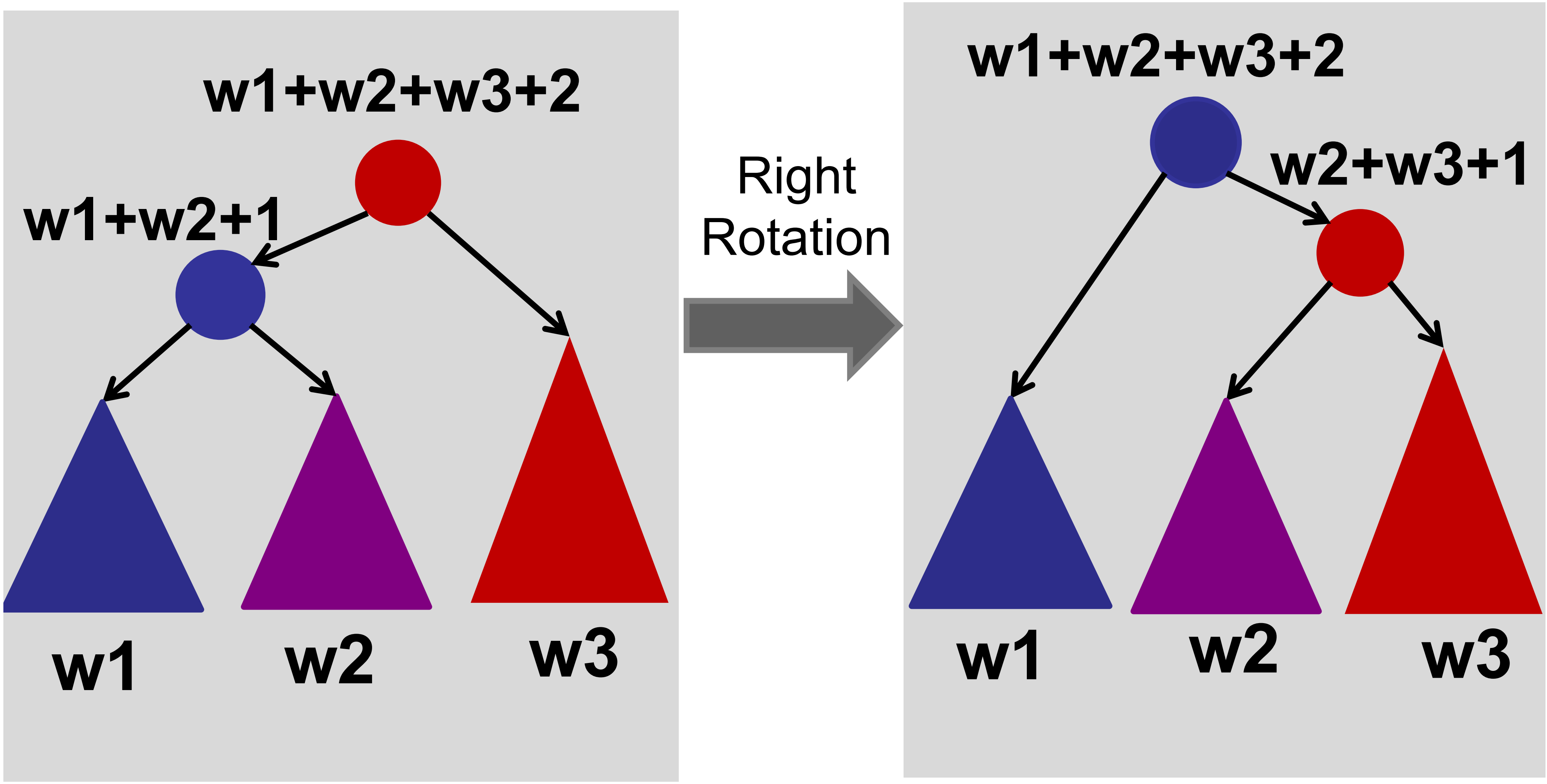


How long does it take to update the weights during a rotation?

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

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.

Today

Three examples of augmenting balanced BSTs

1. Order Statistics

2. Intervals

3. Orthogonal Range Searching