CS2020
Data Structures and Algorithms

Welcome!

Coding Quiz

- Date: March 10 / 11 / 13
  - Administered during your Discussion Group
  - Location: iCube 3-45/3-47
  - Do not skip Discussion Group next week.

- Practice problems:
  - See posted sample problems
  - See last year's Coding Quiz
  - Talk to your tutor.
  - If you want more practice problems, ask.

Administrative

Advice:

- Coding under time pressure is hard.
  - Don’t rush: read the problem carefully.
  - Don’t rush: plan before you code.
  - Document your code as you go.
  - Don’t get stuck if something doesn’t work.

- Use your time wisely.

Administrative

Advice:

- Test your solution
  - Working code is important.
  - Test “corner-cases.”

- Several possible solutions
  - First, ignore efficiency.
  - Develop a solution that works.
  - Test it. Test it. Test it.
  - Then, improve the efficiency.

Administrative

Advice:

- Use good coding style
  - Deductions for code that is badly formatted

- Explain your solution
  - Credit for well-documented code.

Administrative

Advice:

- Don't submit code that does not even compile!

  If you can not solve the problem correctly, then

  submit simple code that solves the problem simply.

Today: Data Structures

Last time...

Binary search trees

Dictionaries (Abstract Data Type)

Balanced search trees

AVL trees

Dynamic Data Structures

1. Maintain a set of items

2. Modify the set of items

3. Answer queries.

Dynamic Data Structures

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

Example: QuestionTree

- Operations that create a data structure
  - buildTree(Objects[])
- Operations that modify the structure
  - insert
  - delete
- Query operations
  - findQuery

What are trees good for?

- Symbol tables
  - insert, delete, search
- Dictionaries
  - insert, delete, search, successor, predecessor
- Bags, Heaps, etc.

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.)

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.

Order Statistics

Input

A set of integers.

Output: select(k)

The kth item in the set.

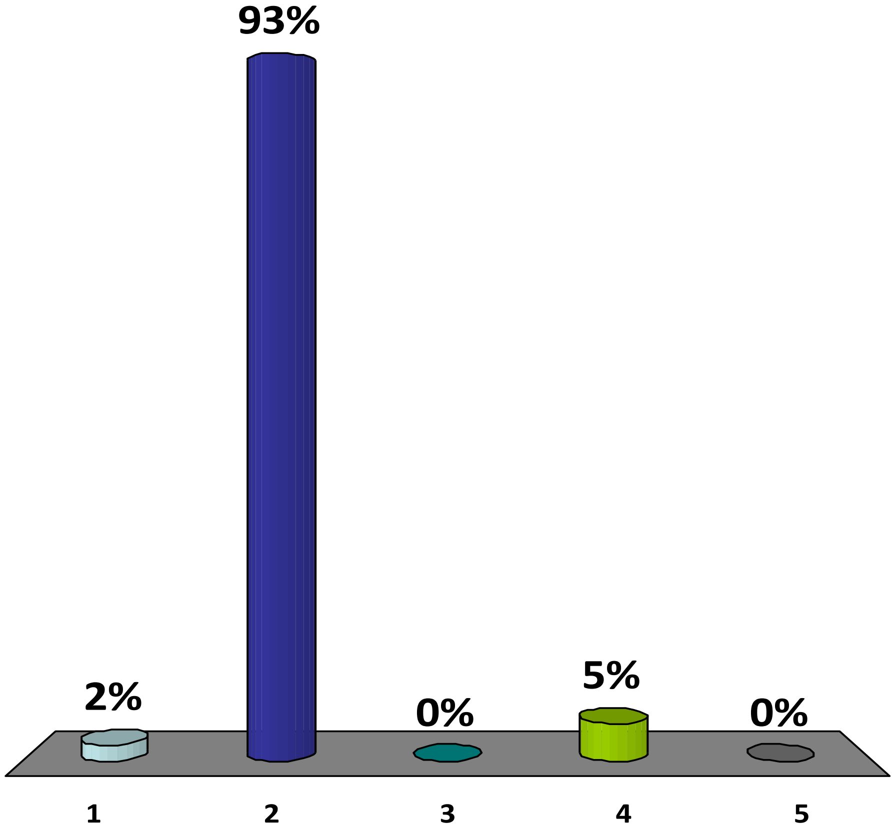
52	**7**	**13**	**43**	**22**	**92**	**18**	**9**	**65**	**67**	**87**	**25**

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 k^{th} 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 k^{th} 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)

Dynamic Data Structures

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

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 1:

Basic structure: sorted array A.

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

- O(n) time

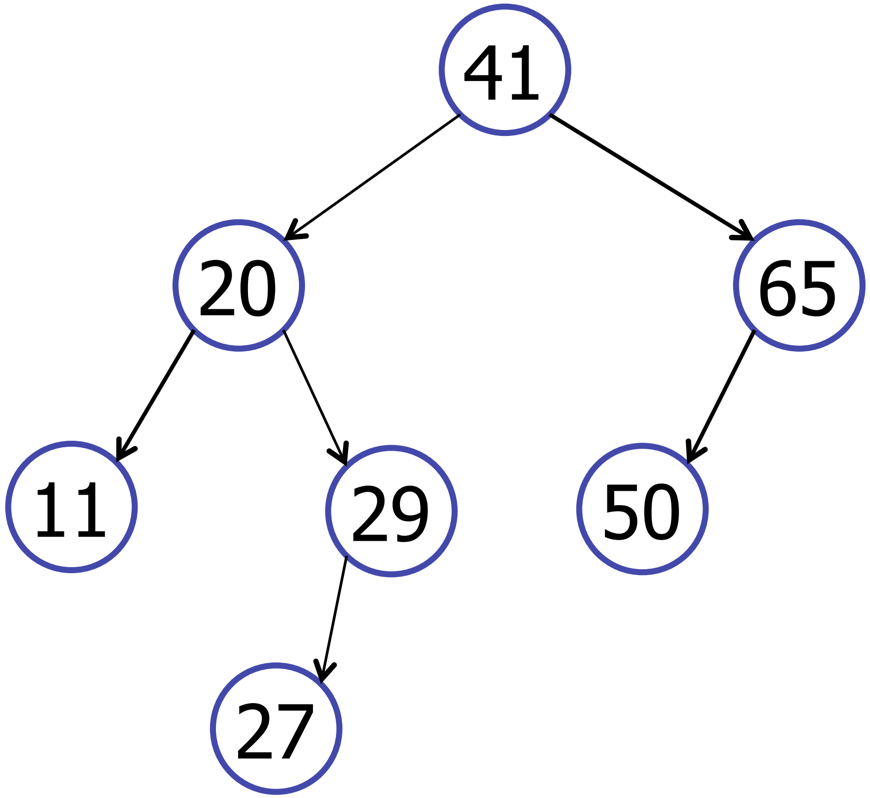
select(int k): return A[k]

- O(1) time

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

Dynamic Order Statistics

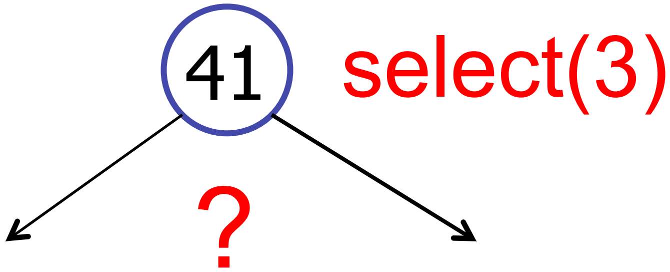
Solution 2: use a (balanced) tree



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

Dynamic Order Statistics

Solution 2: use a tree

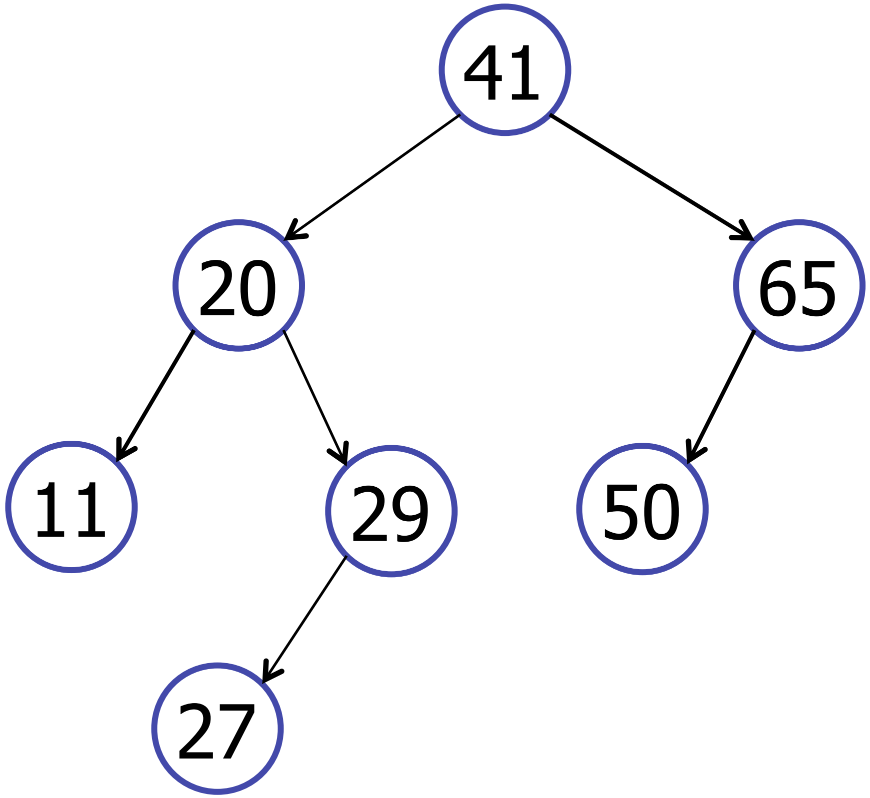


11	**20**	**27**	**29**	**41**	**50**	**65**

Dynamic Order Statistics

Solution 2: use a tree

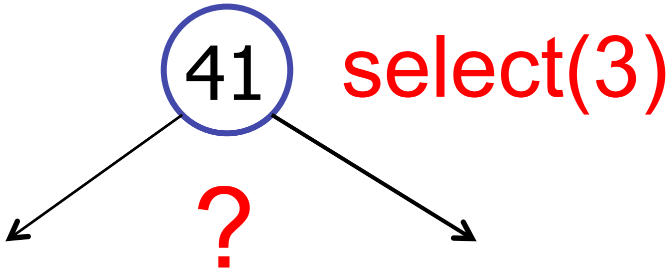
select(k): O(k)
in-order iterator



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

Dynamic Order Statistics

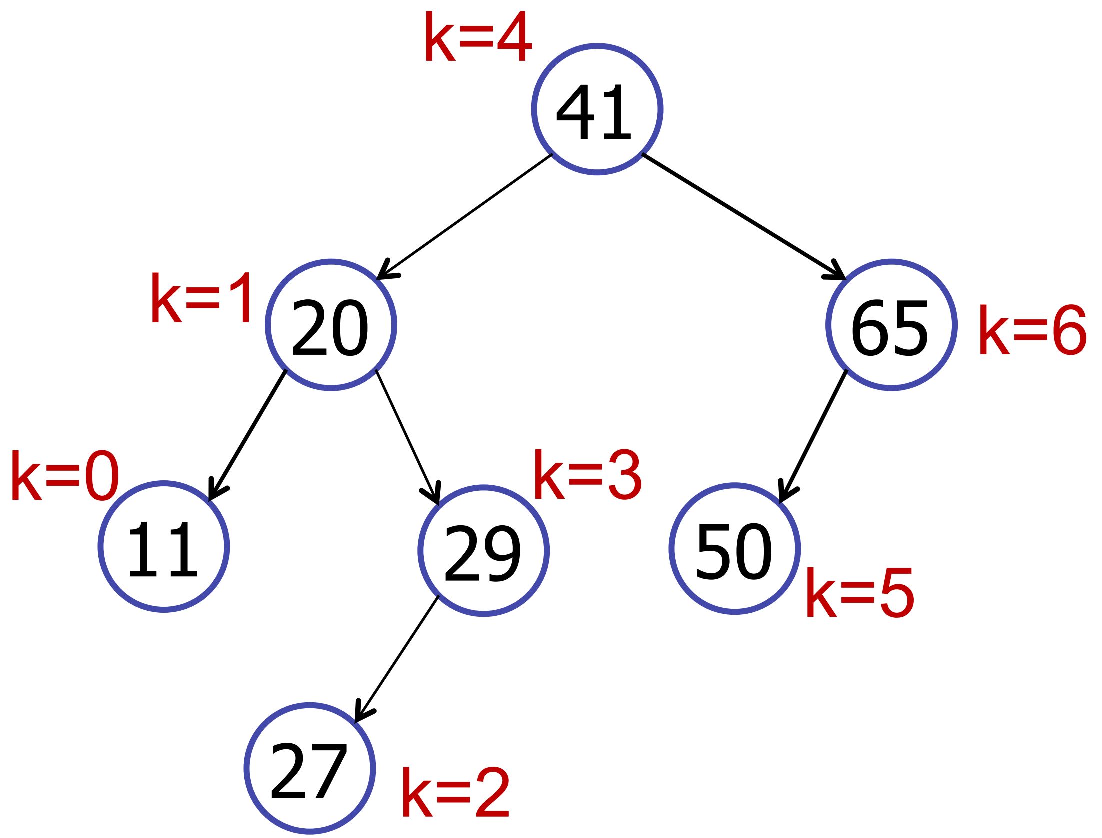
Solution 2: Augment! What to store in each node?



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

Dynamic Order Statistics

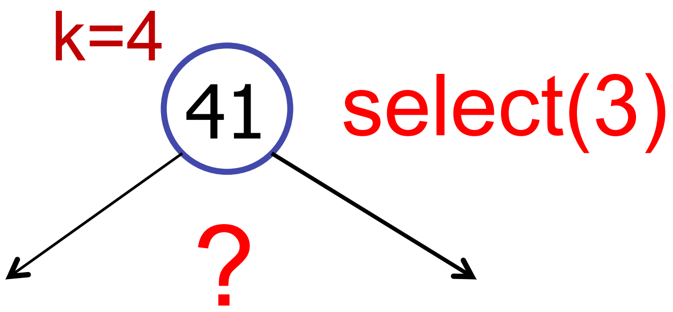
Solution 2: use a tree, store rank in every node



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

Dynamic Order Statistics

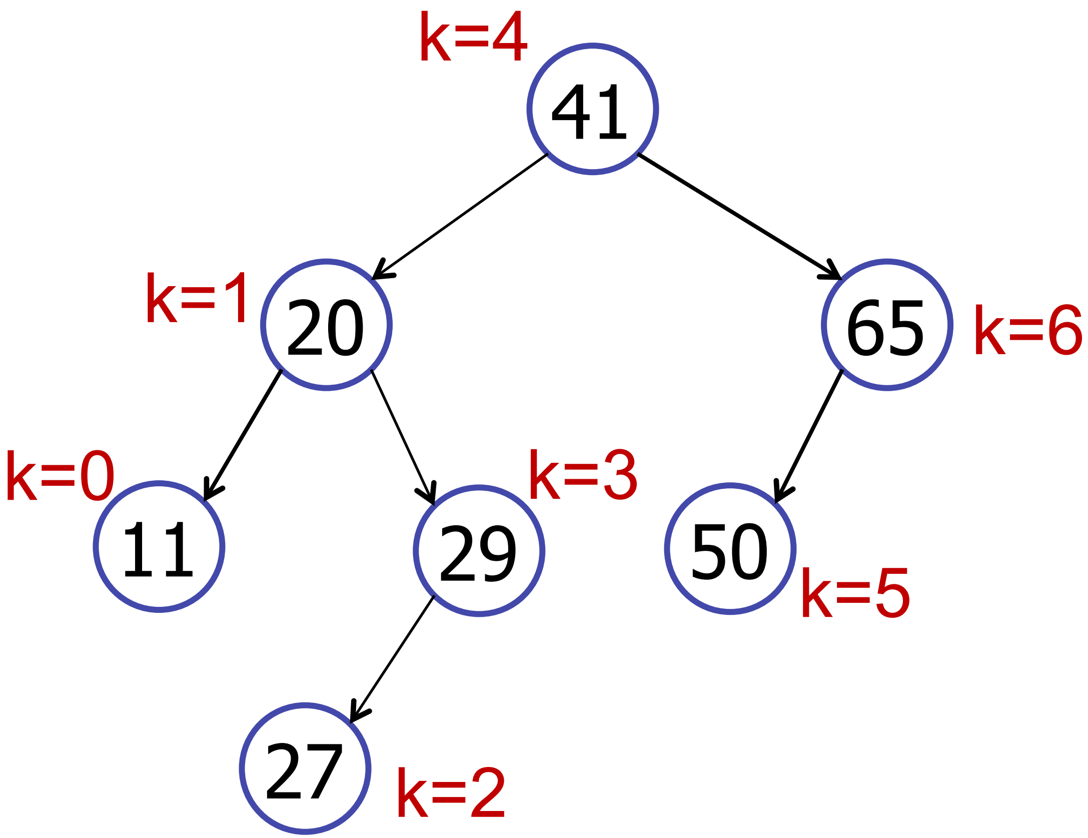
Solution 2: use a tree, store rank in every node



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

Dynamic Order Statistics

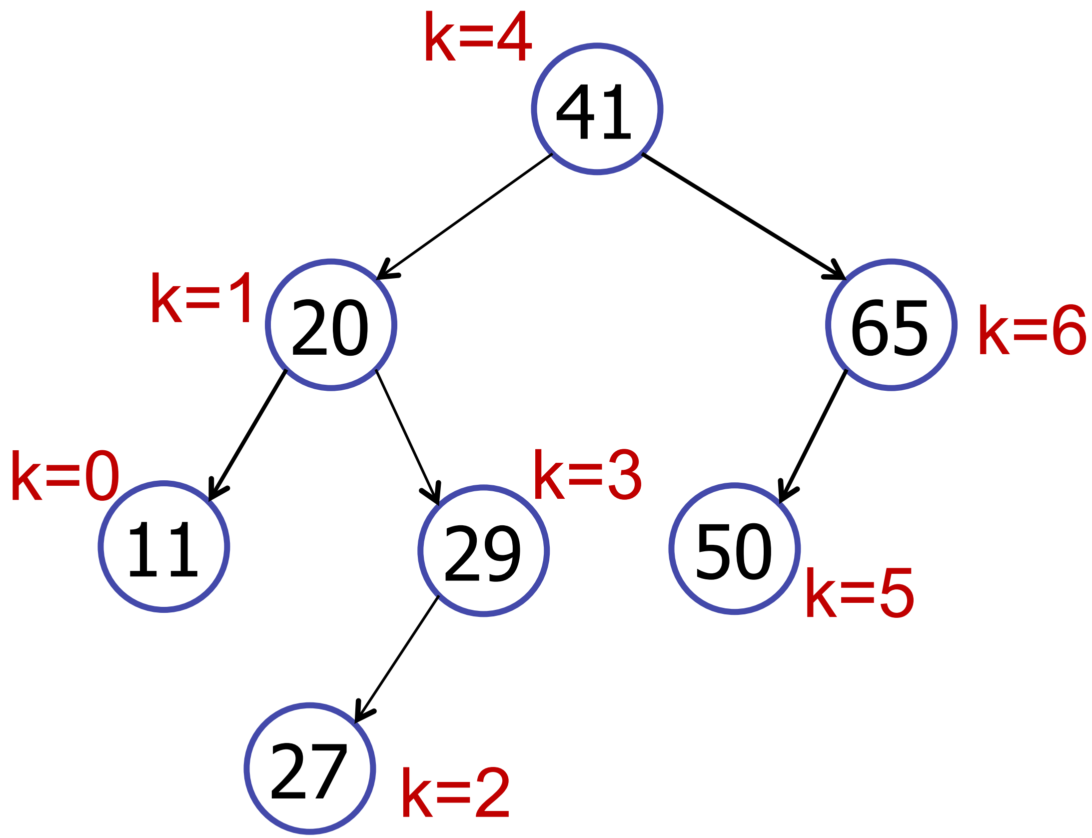
Solution 2: use a tree, store rank in every node



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

Dynamic Order Statistics

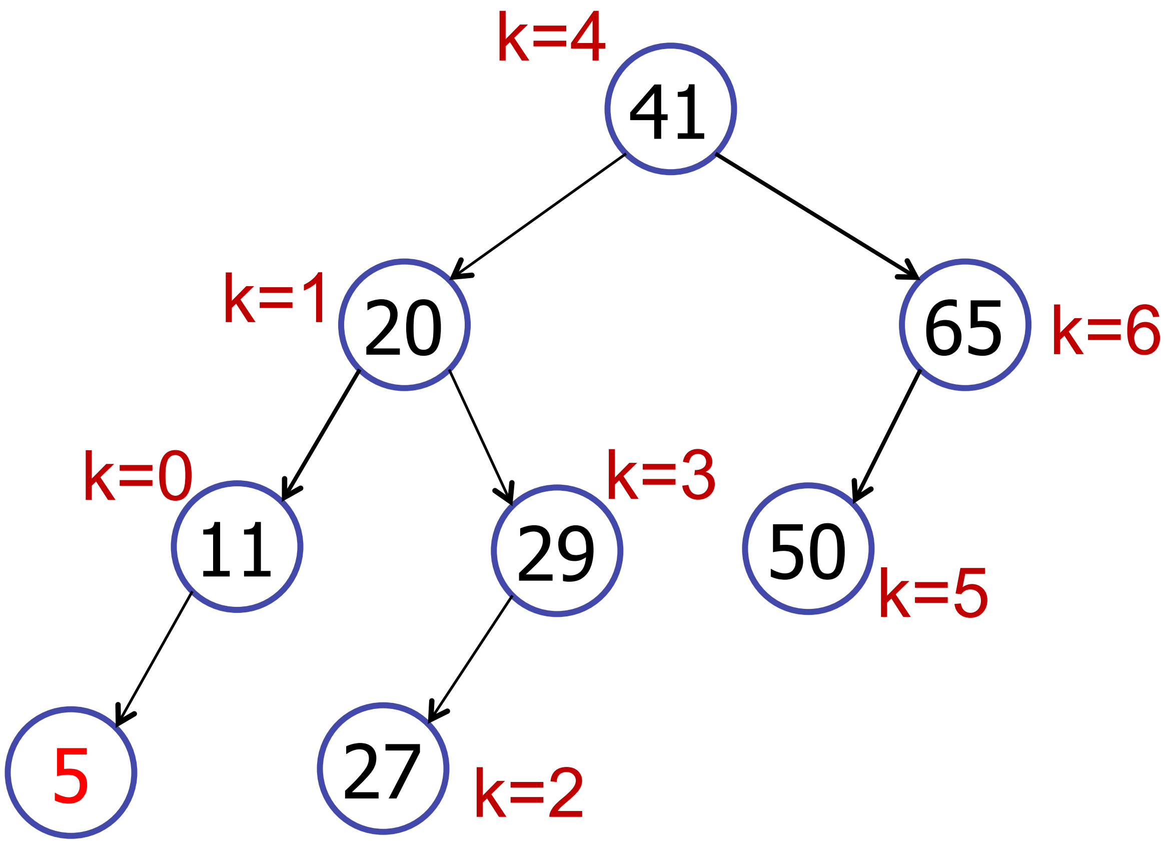
Solution 2: use a tree, store rank in every node



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

Dynamic Order Statistics

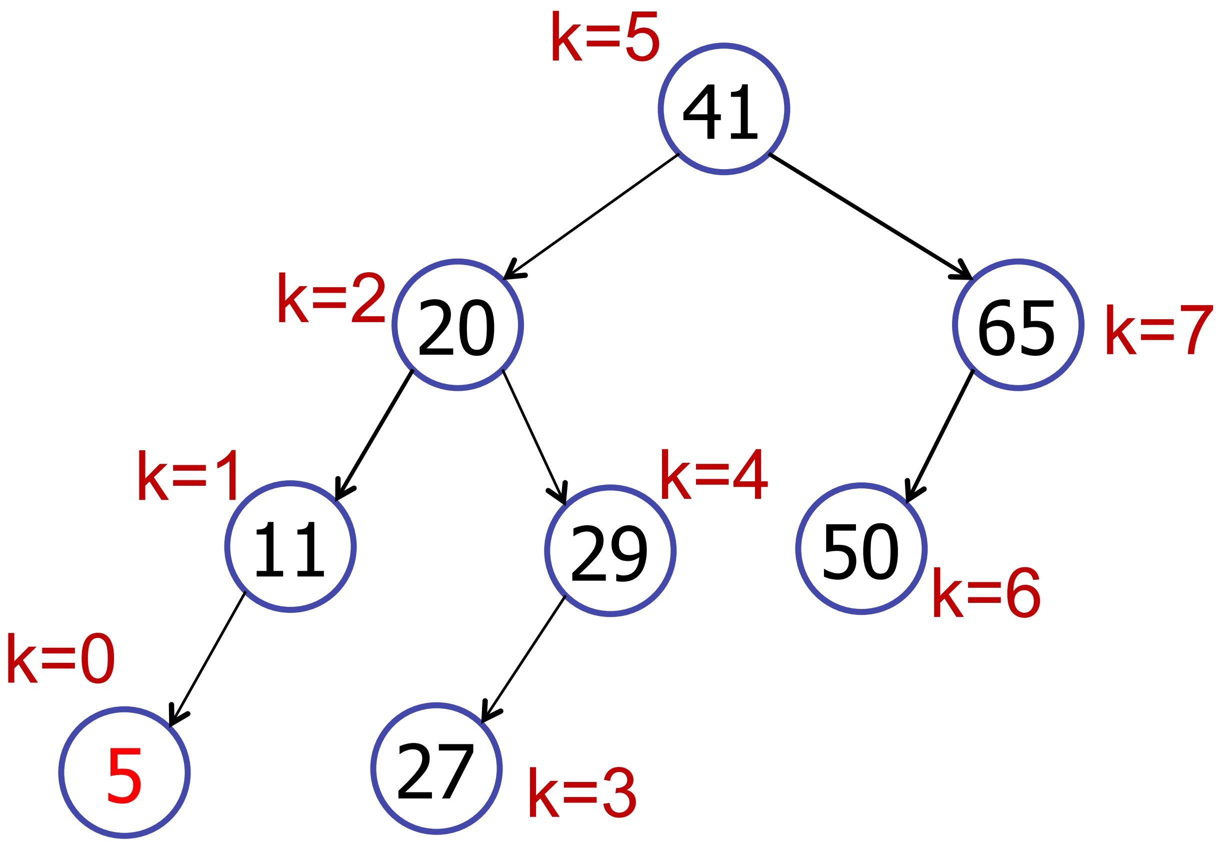
Solution 2: store rank in every node



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

Dynamic Order Statistics

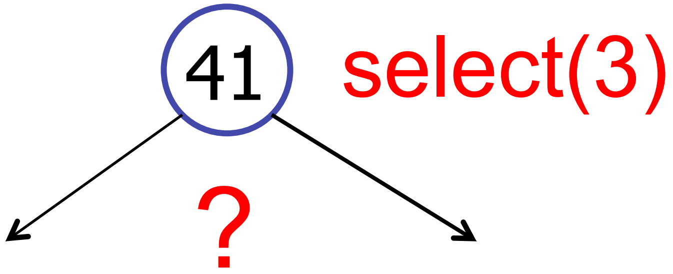
Solution 2: store rank in every node



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

Dynamic Order Statistics

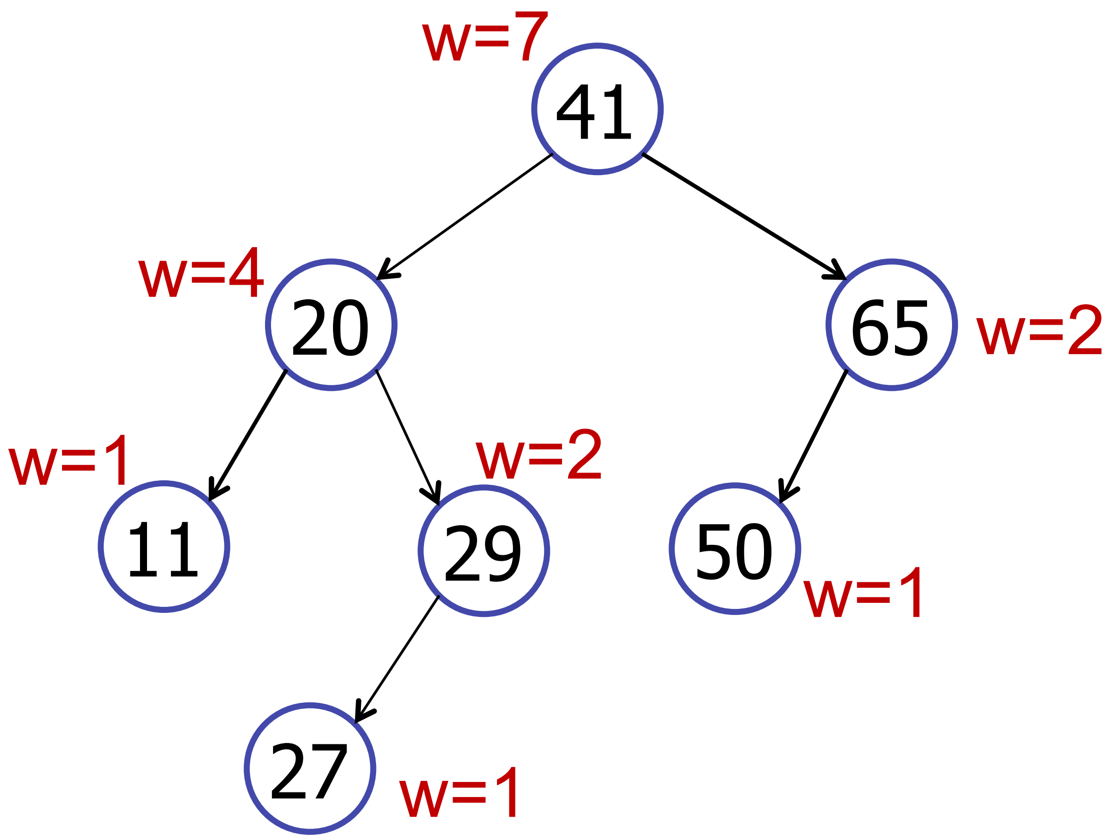
What should we store in each node?



11	20	27	29	41	50	65

Dynamic Order Statistics

Solution 3: store size of sub-tree in every node



Dynamic Order Statistics

Solution 3: store size of sub-tree in every node

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

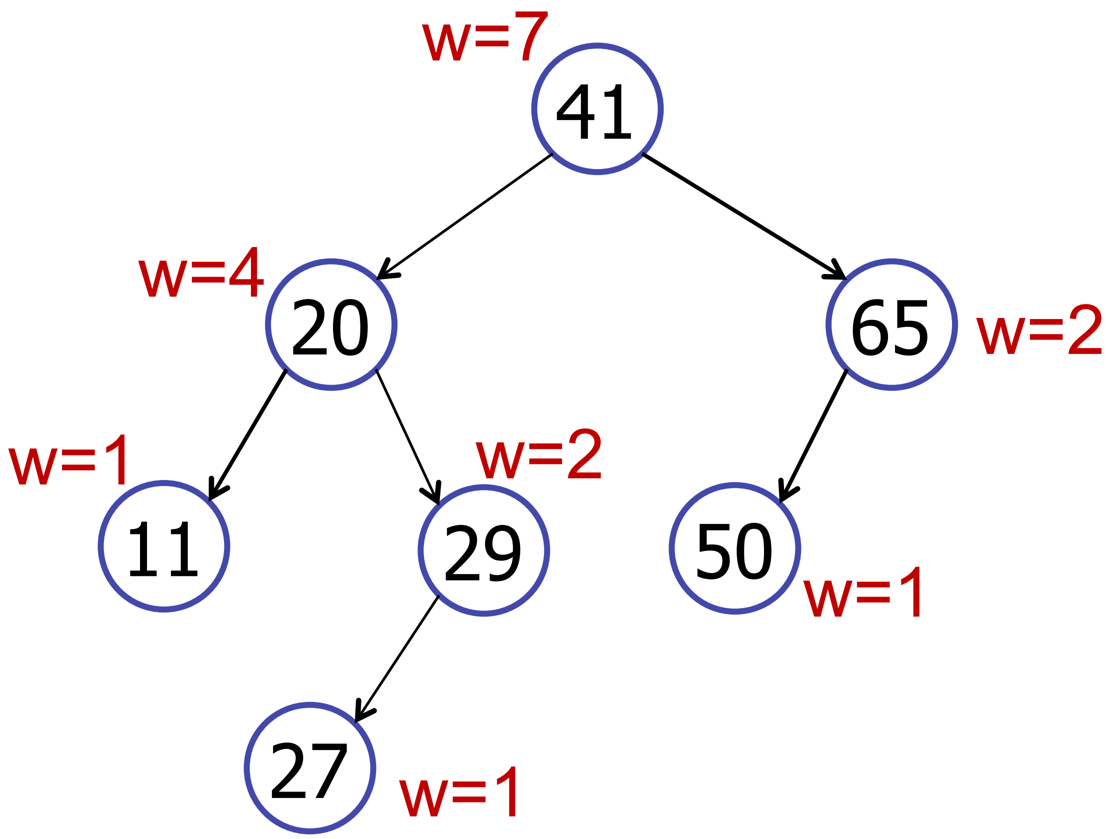
Define weight:

$w(\text{leaf}) = 1$

$w(v) = w(v.\text{left}) + w(v.\text{right}) + 1$

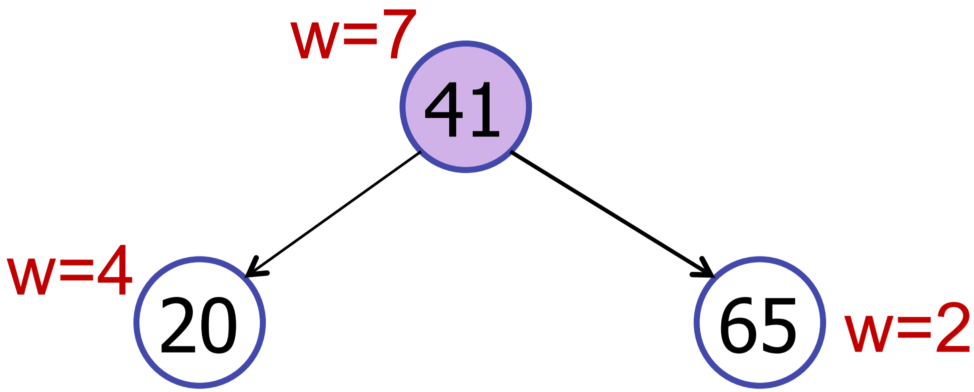
Dynamic Order Statistics

Solution 3: store size of sub-tree in every node



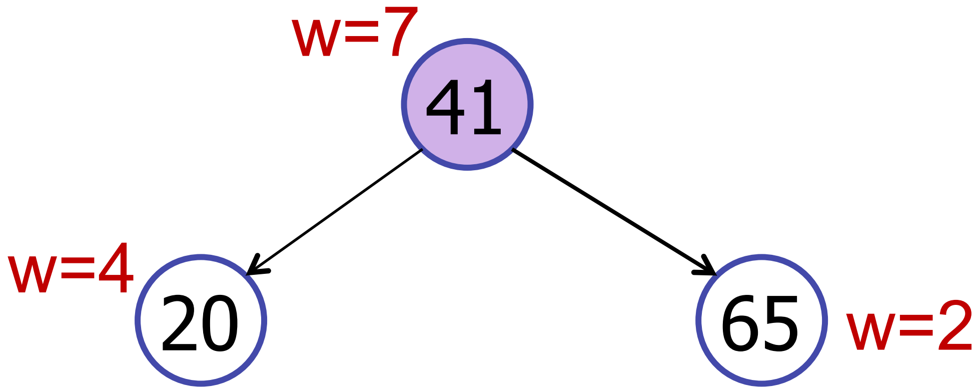
Dynamic Order Statistics

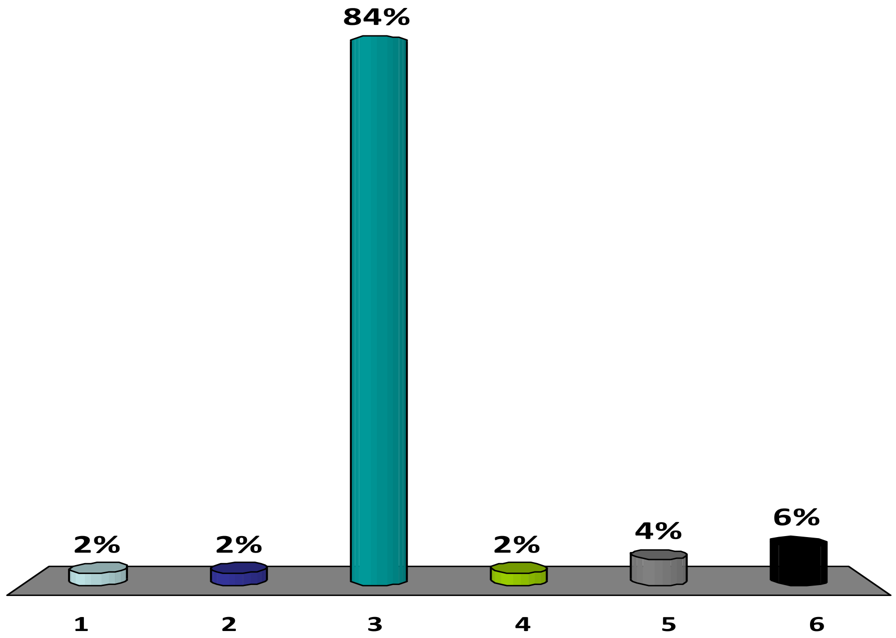
Example: select(3)



What is the rank of 41?

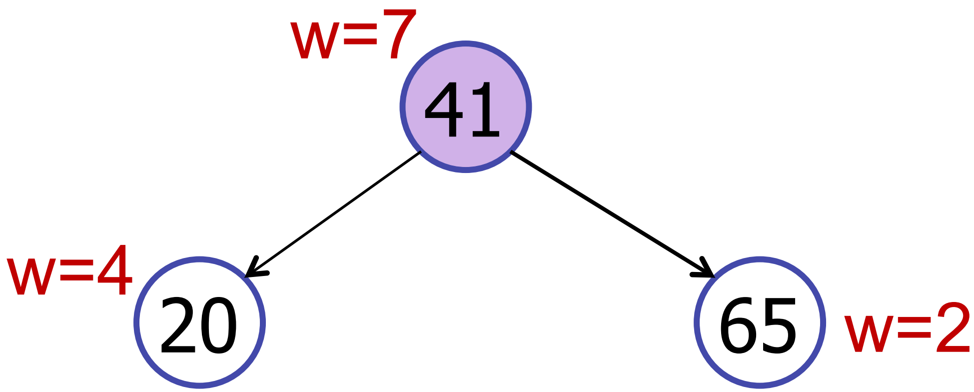
1. 1
2. 3
3. 5
4. 7
5. 9
6. Can’t tell.





Dynamic Order Statistics

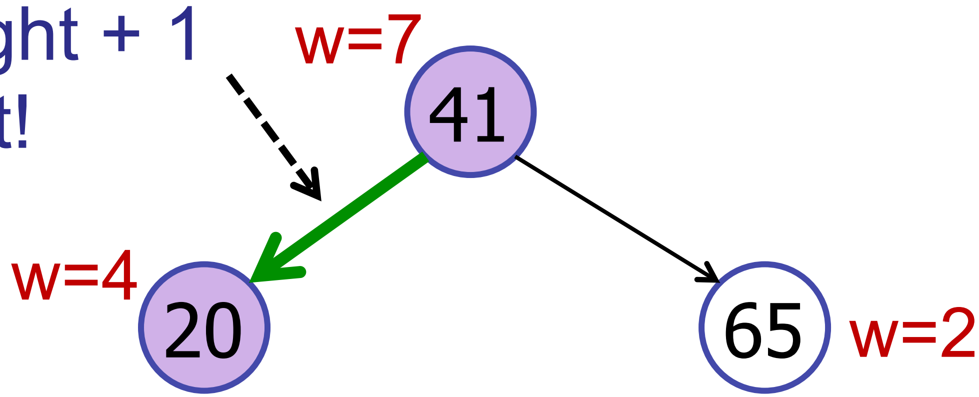
Example: select(3)



Dynamic Order Statistics

Example: select(3)

3 < left.weight + 1
Go left!



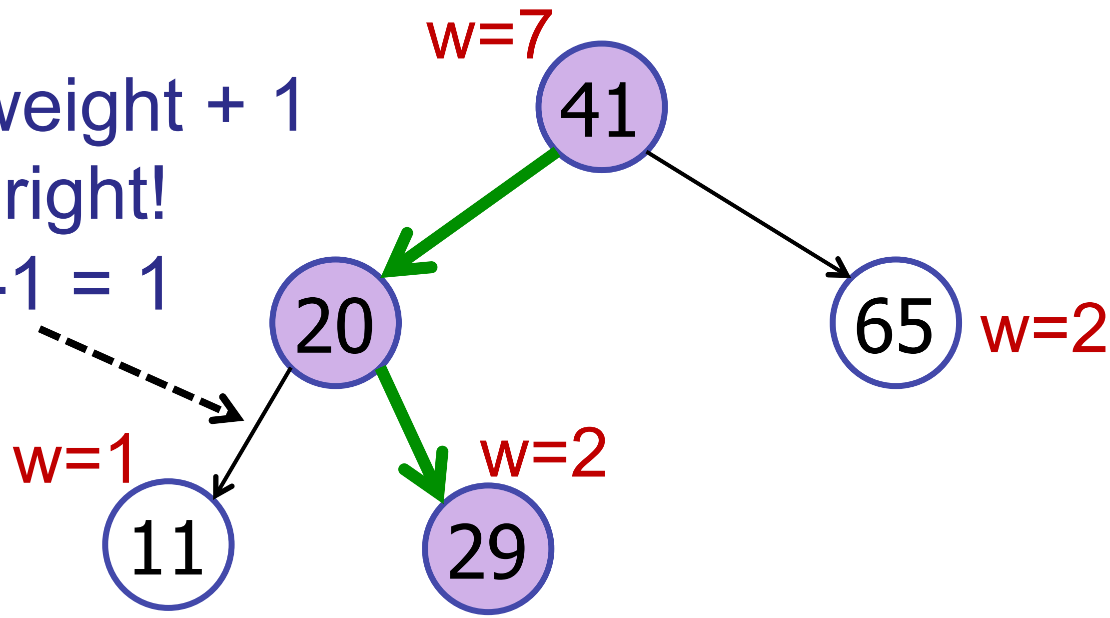
Dynamic Order Statistics

Example: select(3)

3 > left.weight + 1

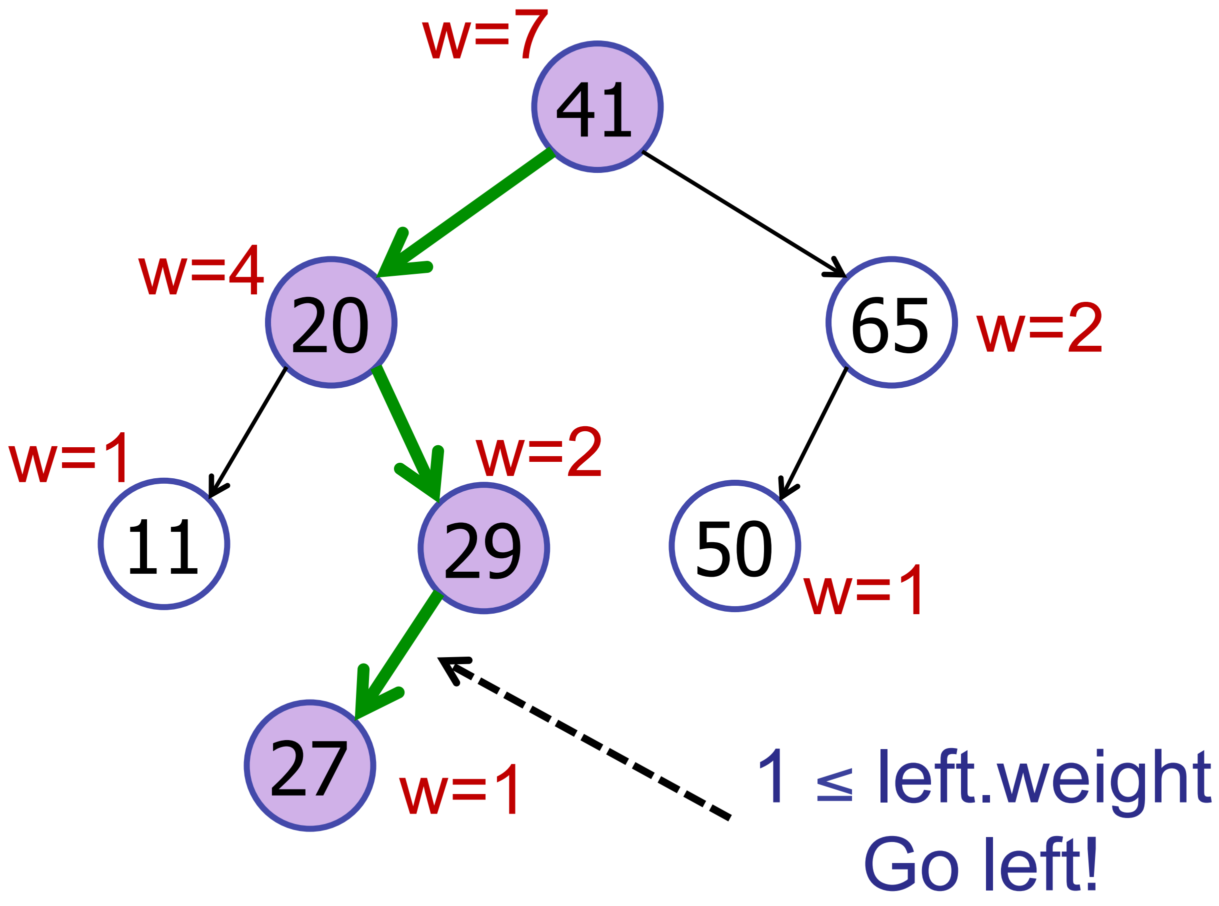
Go right!

3-1-1 = 1



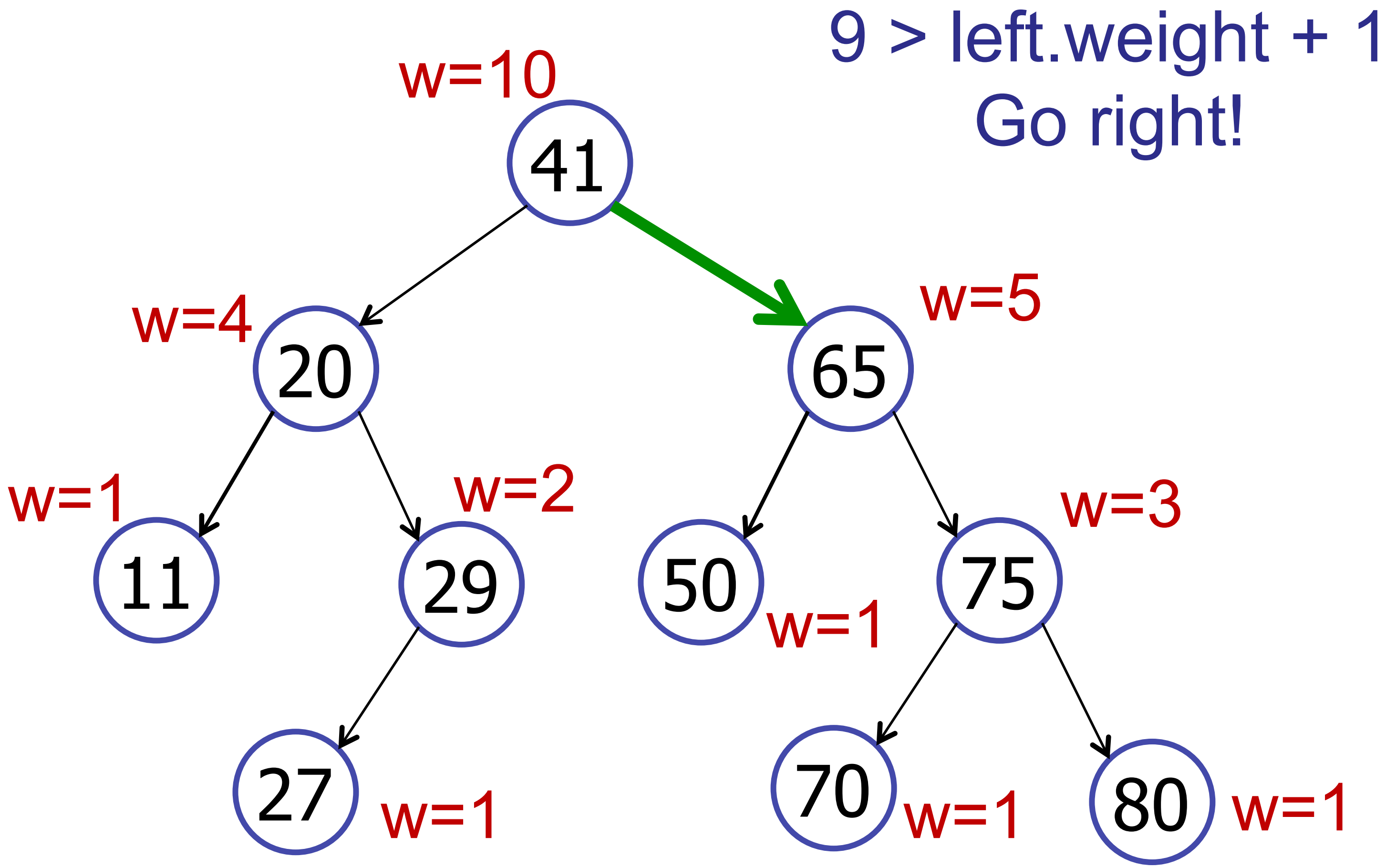
Dynamic Order Statistics

Example: select(3)

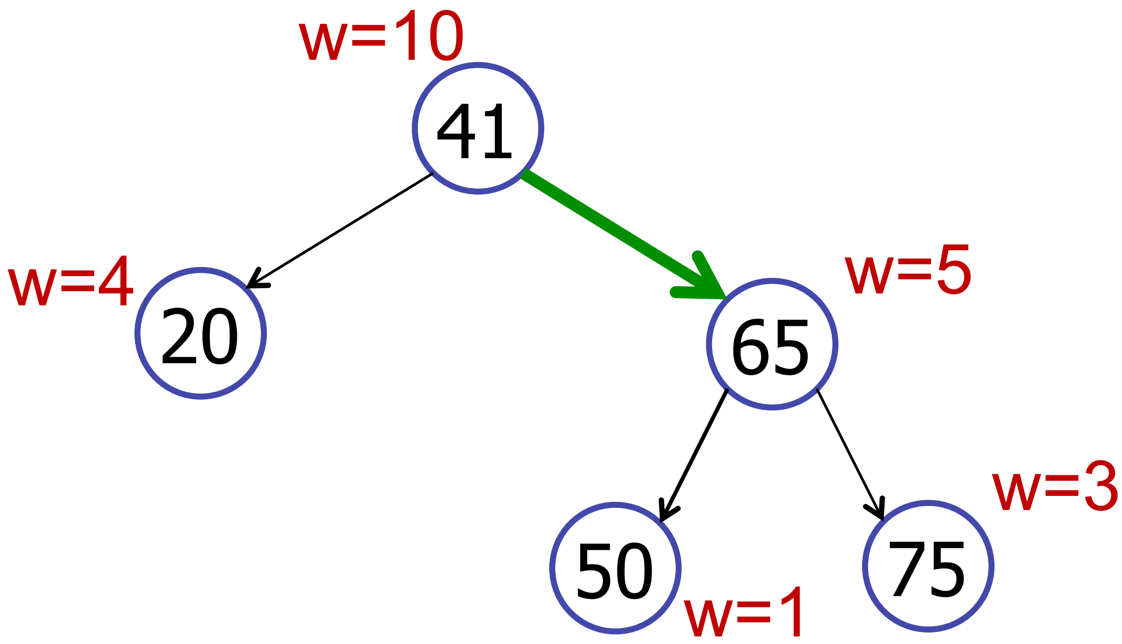


Dynamic Order Statistics

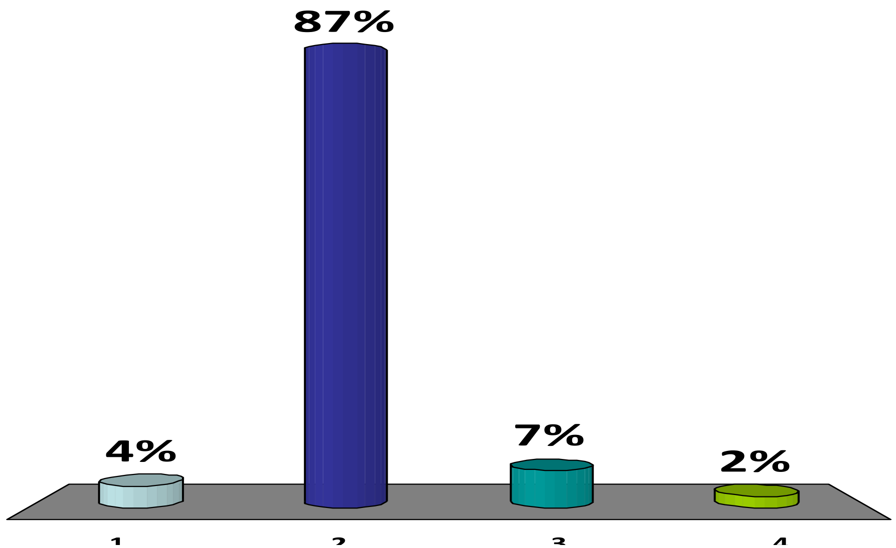
Example: select(9)



select(9)

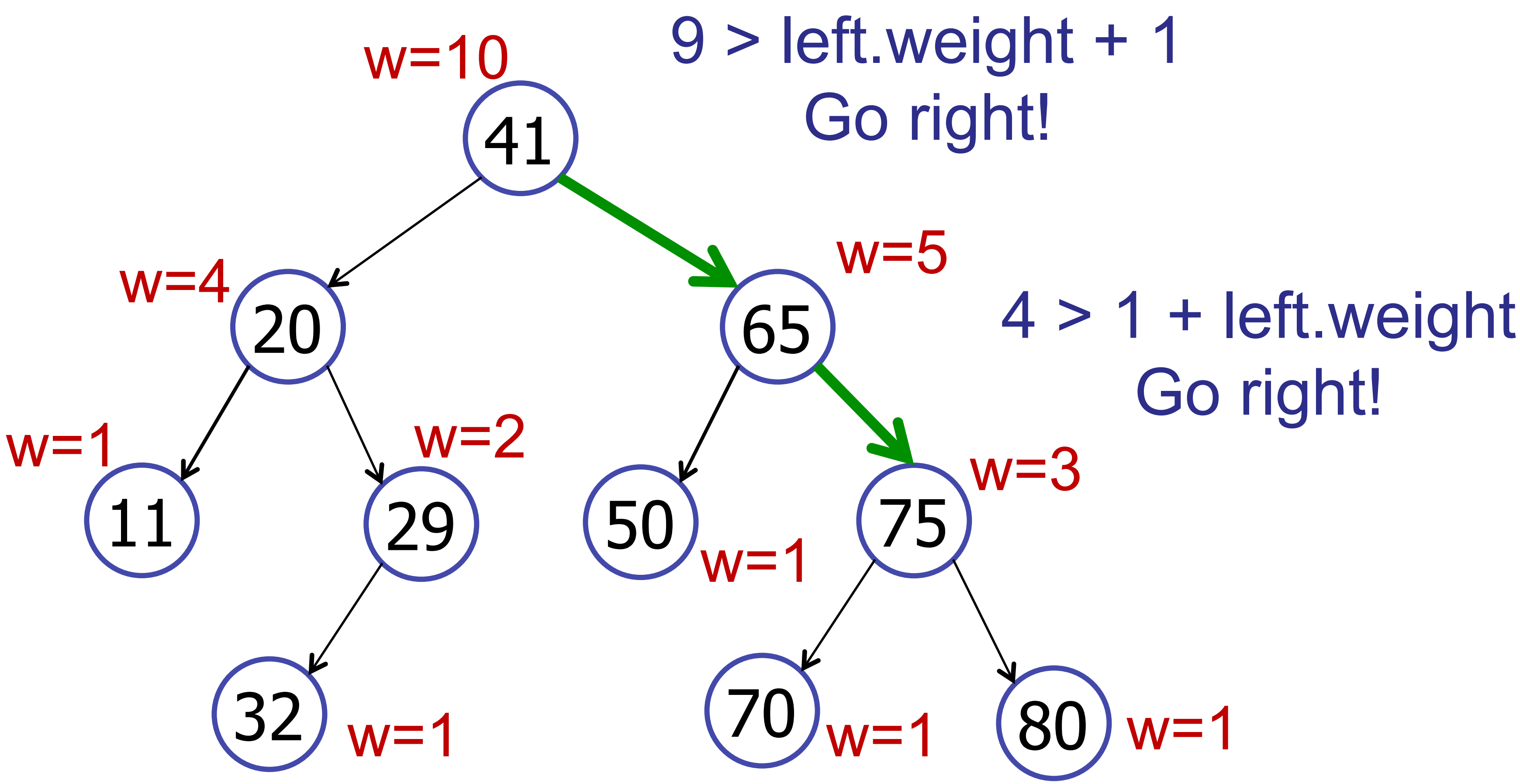


1. Go left at 65
2. Go right at 65
3. Stop at 65
4. I’m confused



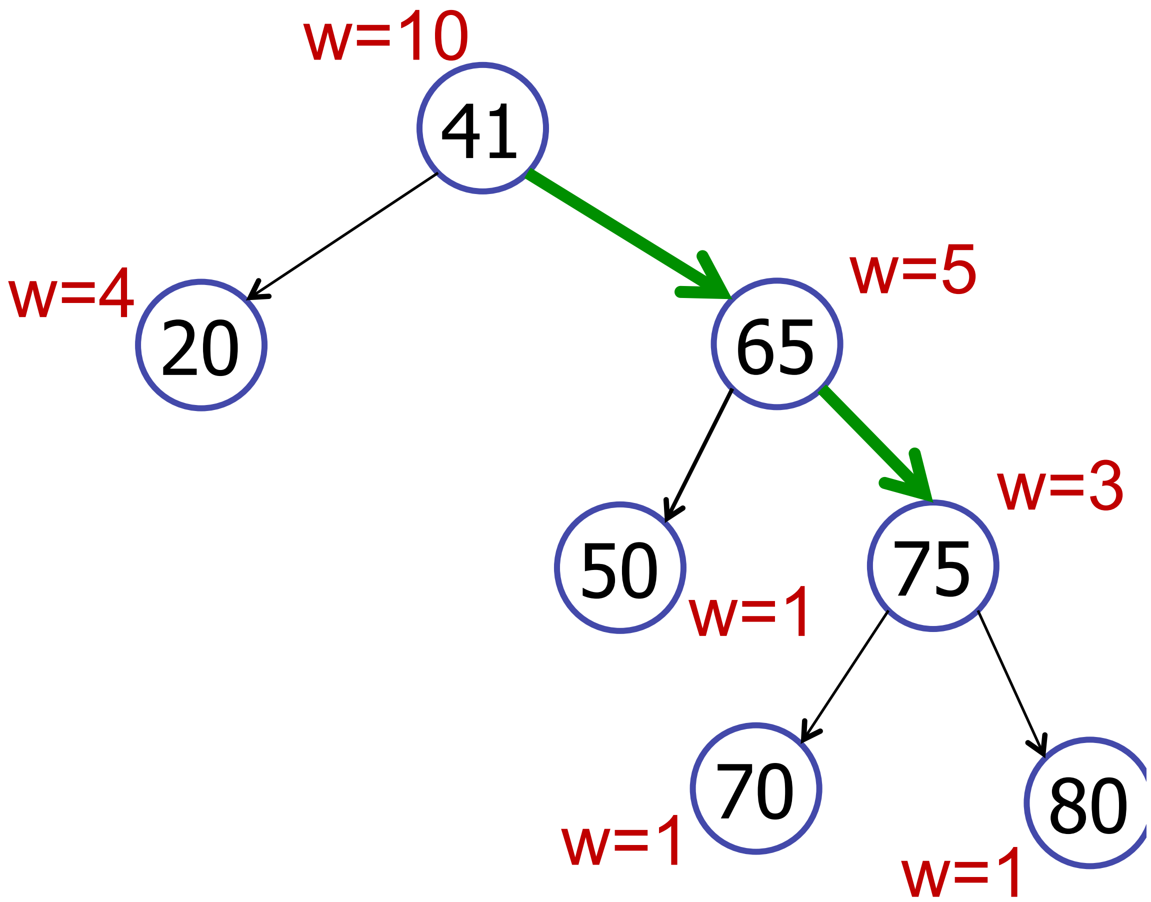
Dynamic Order Statistics

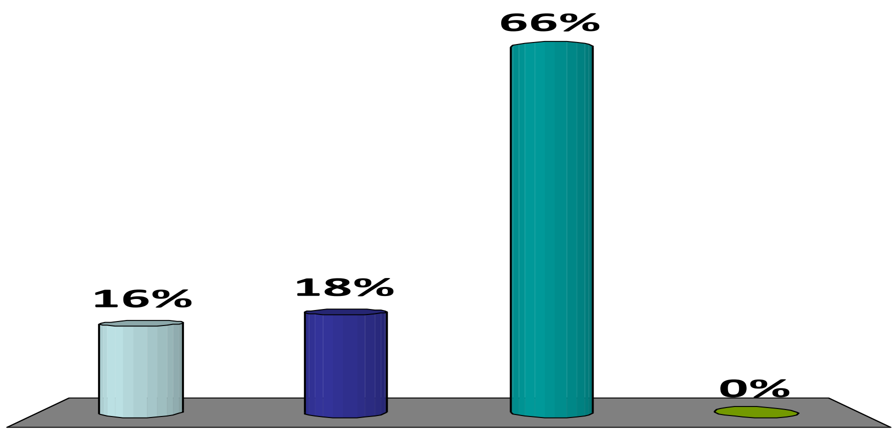
select(9)



select(9)

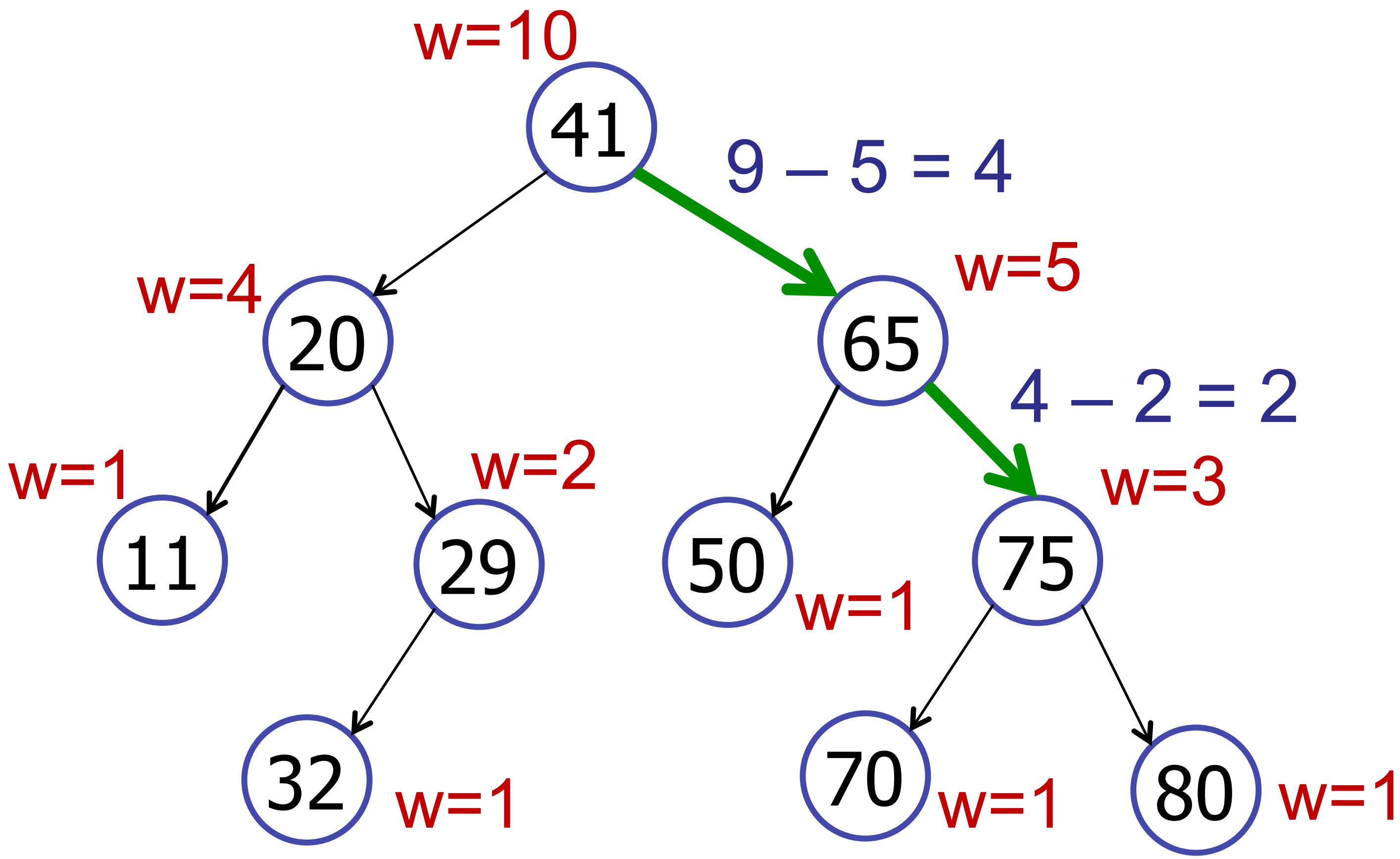
1. Go left at 75
2. Go right at 75
3. Stop at 75
4. I'm confused





Dynamic Order Statistics

select(9)



Dynamic Order Statistics

select(k)

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

Dynamic Order Statistics

select(k) : finds the node with rank k

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

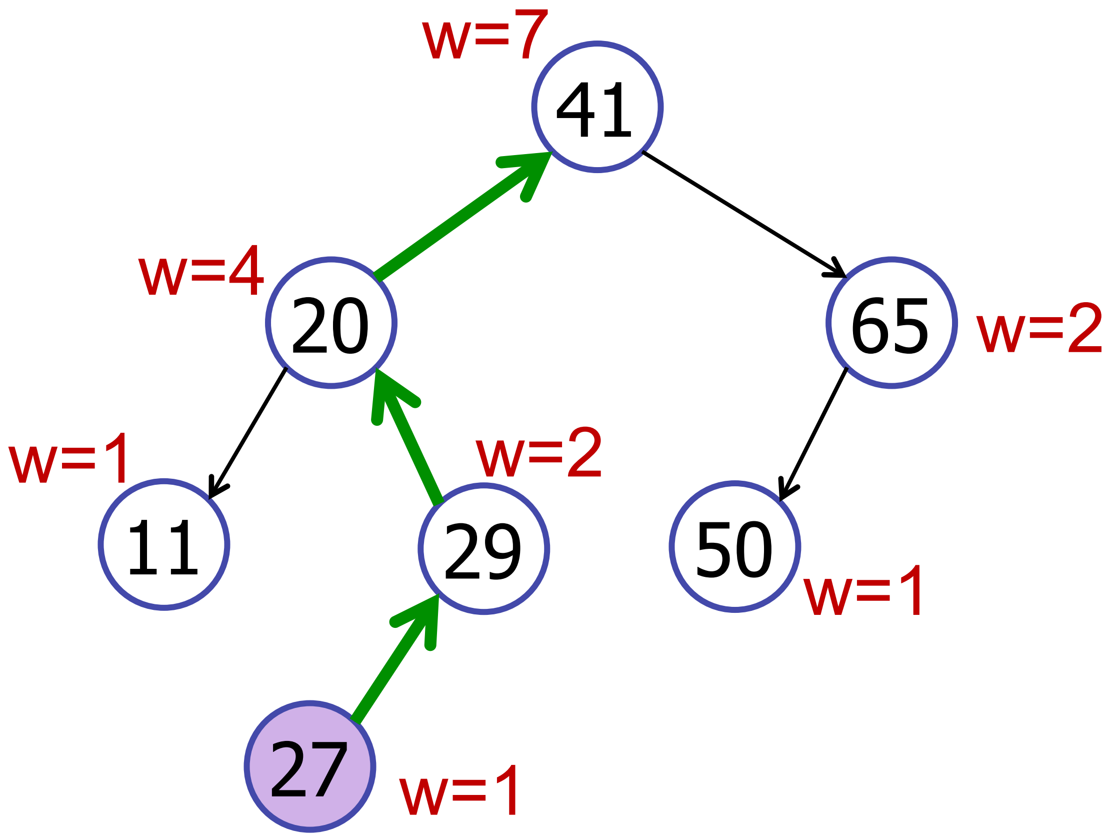
Dynamic Order Statistics

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

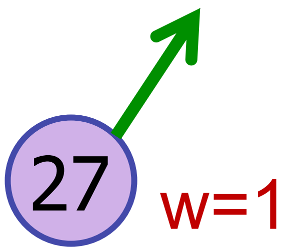
Example: rank(27)



rank = 1

Dynamic Order Statistics

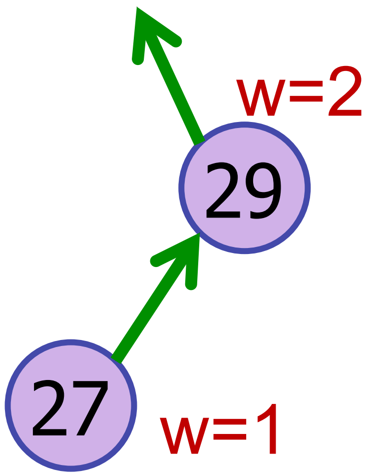
Example: rank(27)



rank = 1

Dynamic Order Statistics

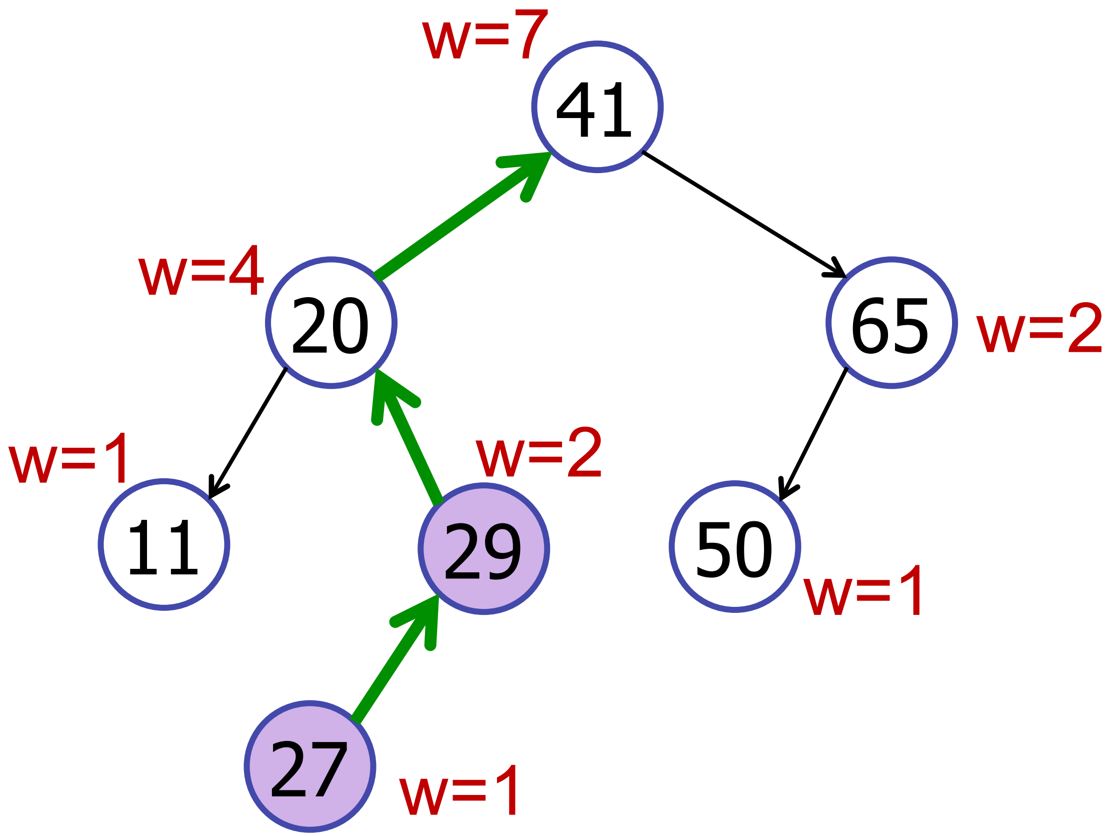
Example: rank(27)



rank = 1

Dynamic Order Statistics

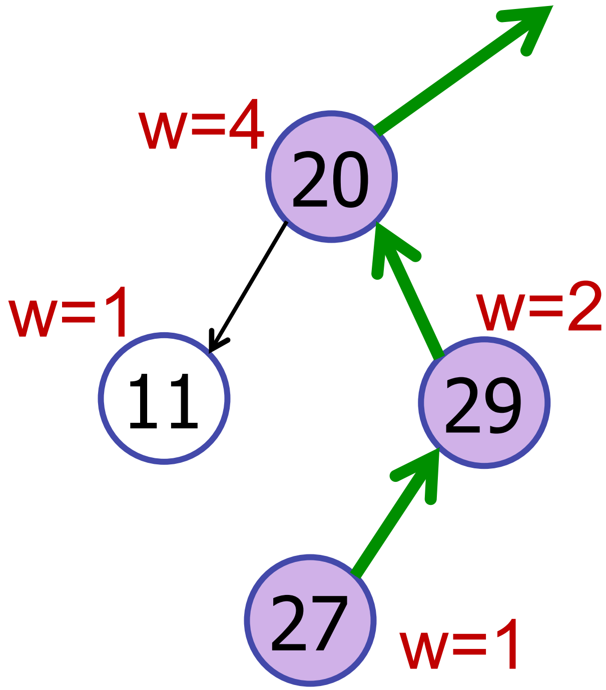
Example: rank(27)



rank = 1

Dynamic Order Statistics

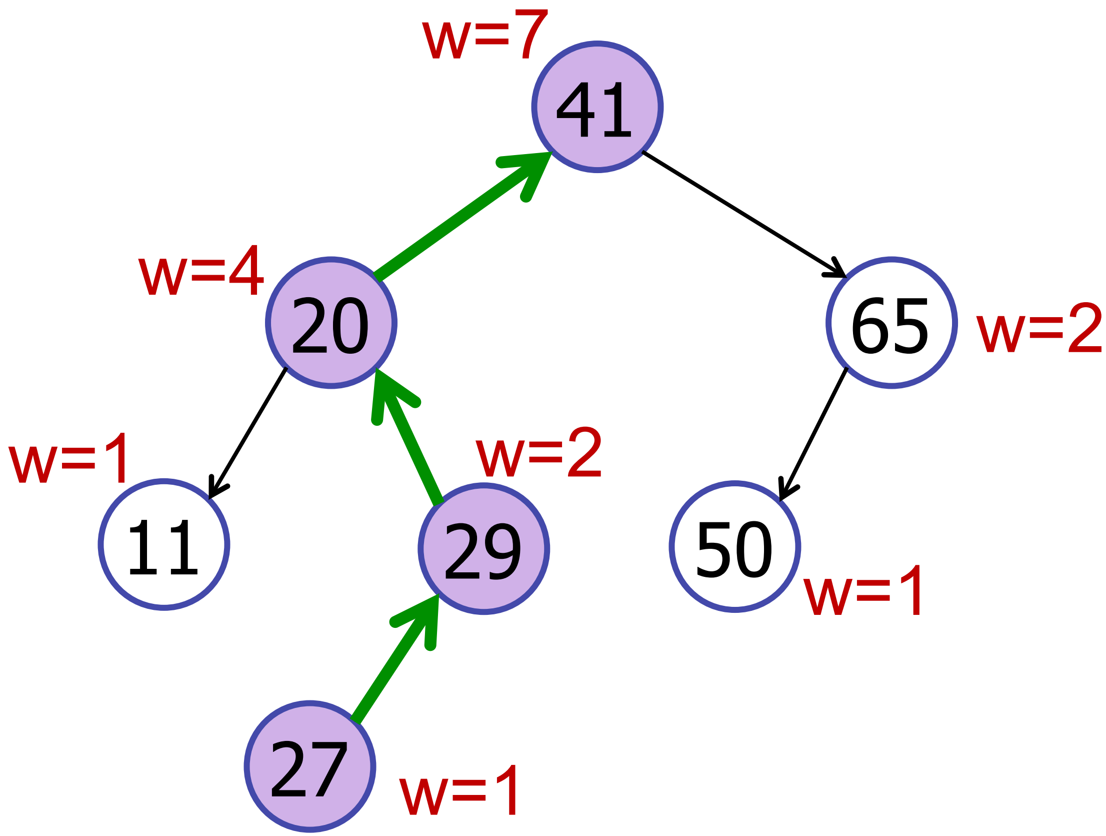
Example: rank(27)



rank = 1 + 2

Dynamic Order Statistics

Example: rank(27)



rank = 1 + 2 = 3

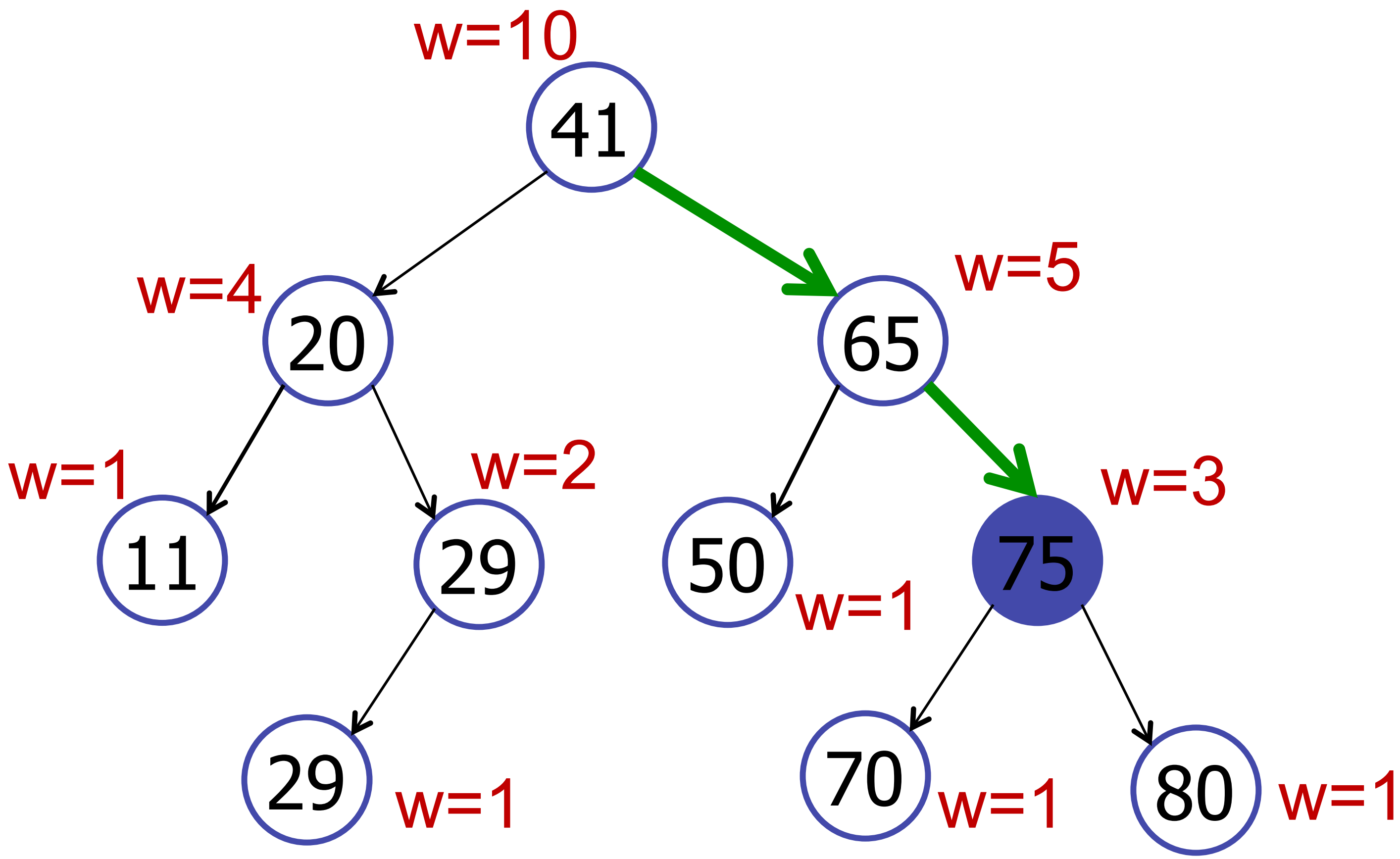
Dynamic Order Statistics

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

```
rank()
    r = left.weight + 1;
    node = this;
    while (node != null) do
        if node is right child then
            r += node.parent.left.weight + 1;
        node = node.parent;
    return r;
```

Dynamic Order Statistics

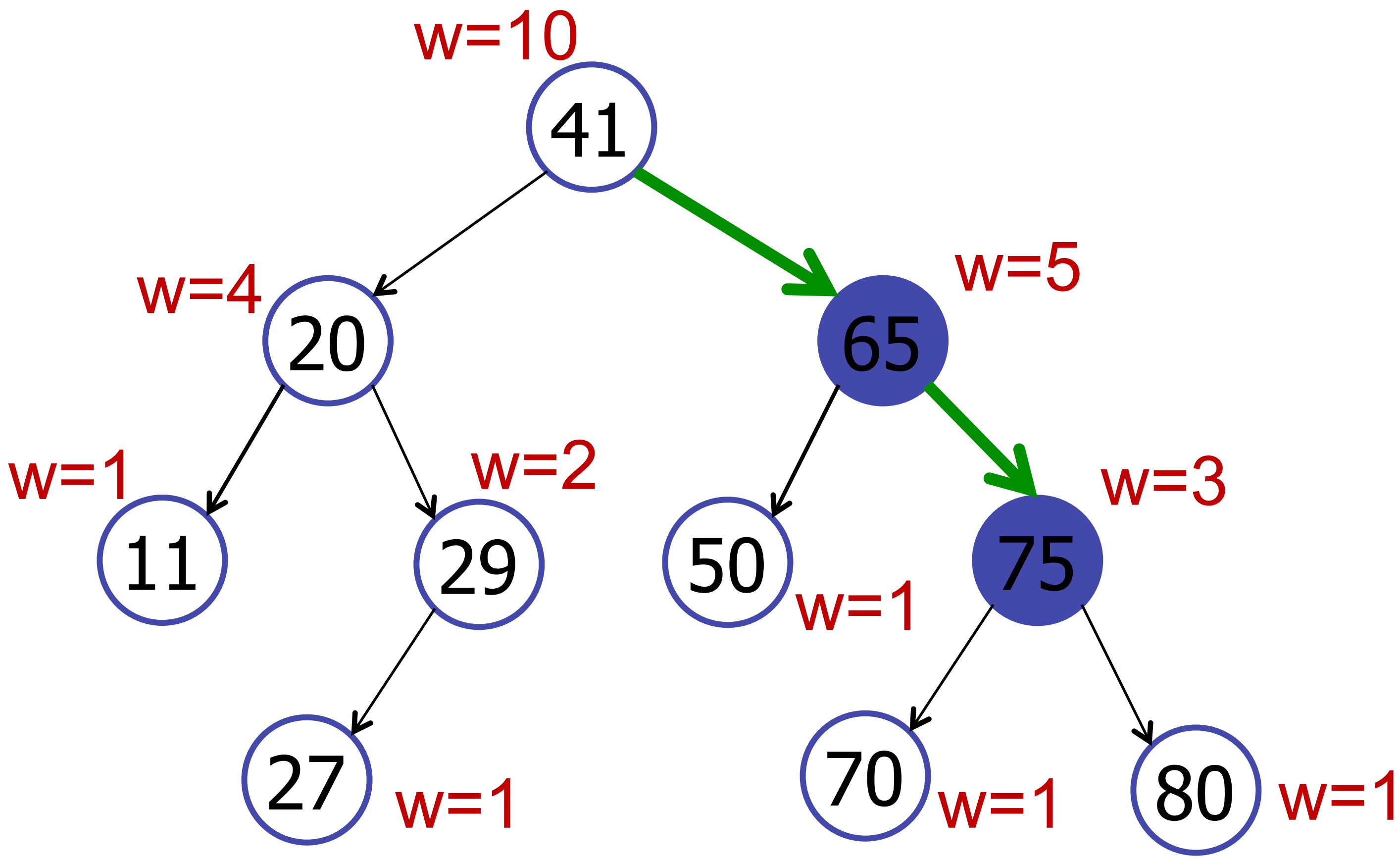
rank(75)



rk = 2

Dynamic Order Statistics

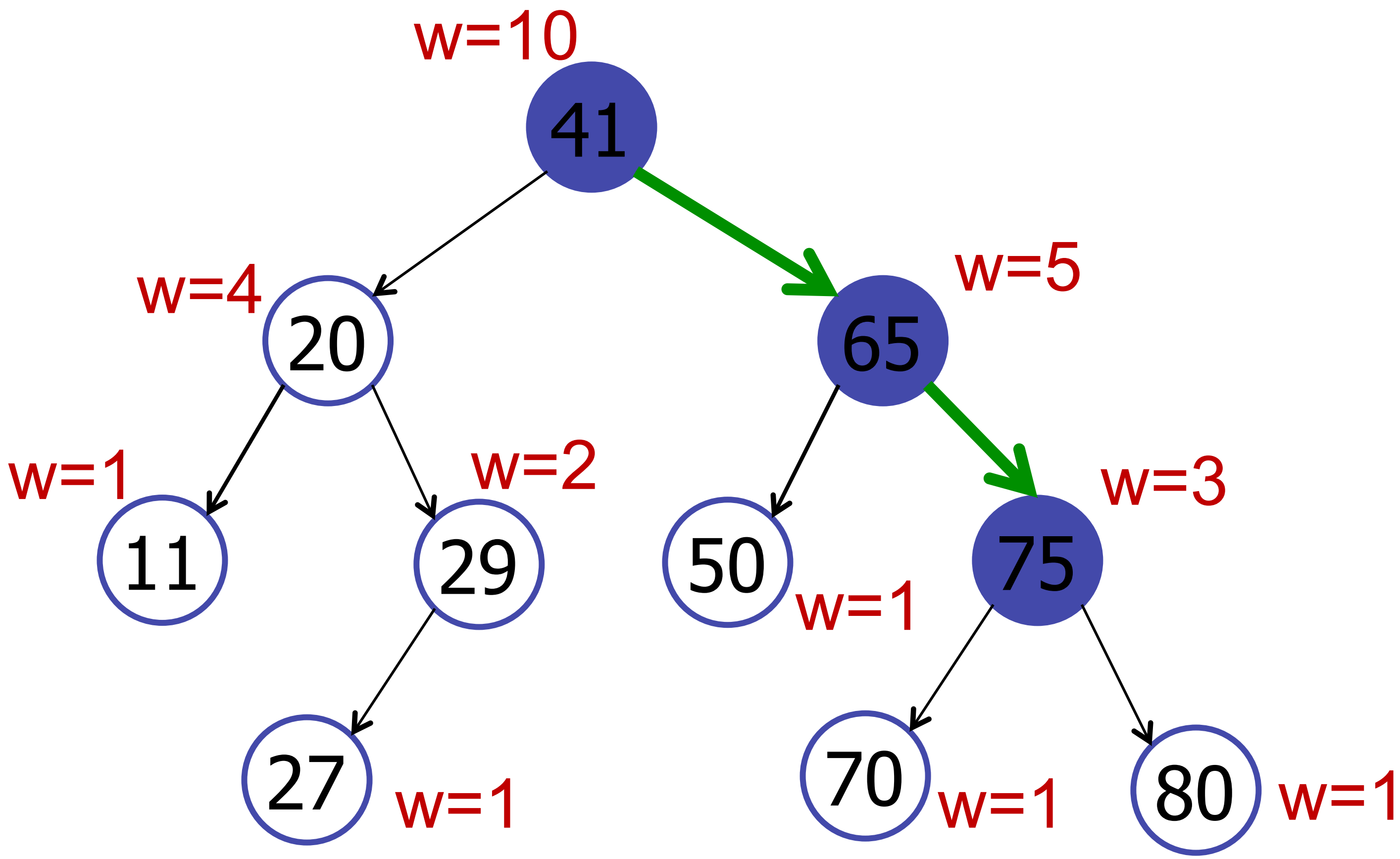
rank(75)



rk = 2 + 2

Dynamic Order Statistics

rank(75)



rk = 2 + 2 + 5 = 9

Dynamic Order Statistics

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

```
rank()
    r = left.weight + 1;
    node = this;
    while (node != null) do
        if node is right child then
            r += node.parent.left.weight + 1;
        node = node.parent;
    return r;
```

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.

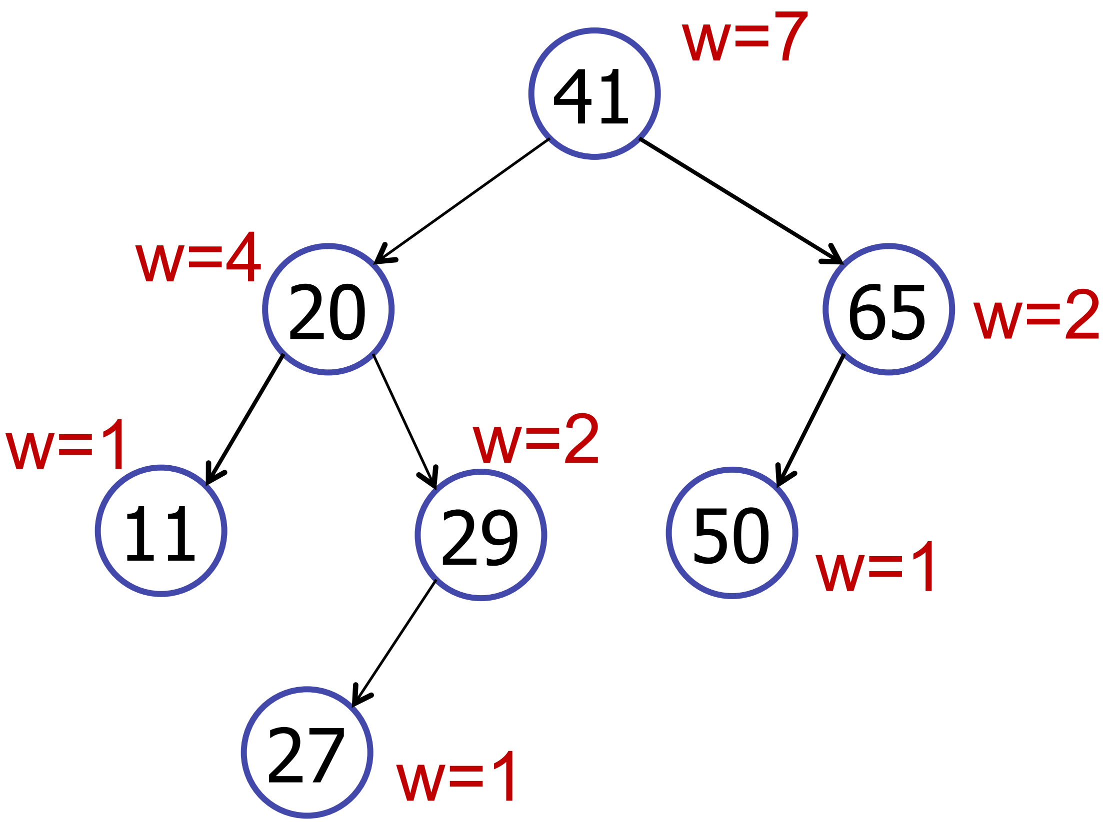
   Update weights as needed

4. Develop new operations using the new info.

   Select and Rank

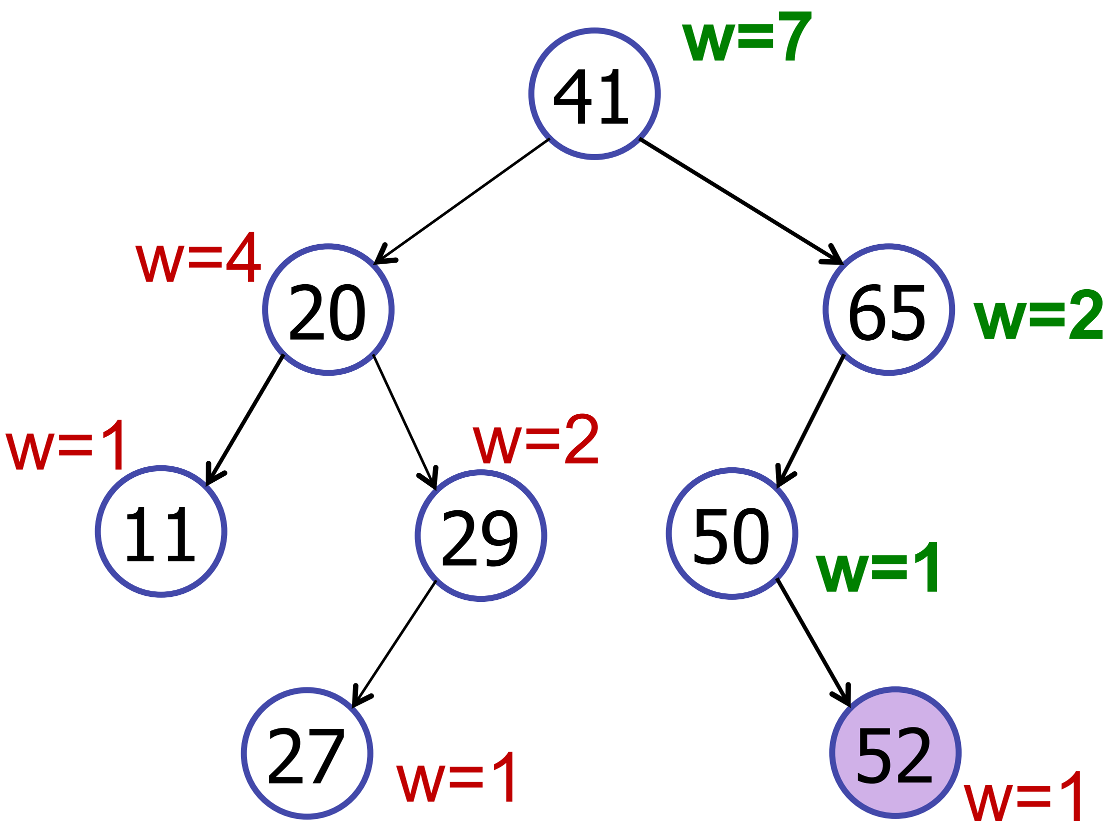
Augmented Trees

Maintain weight during insertions:



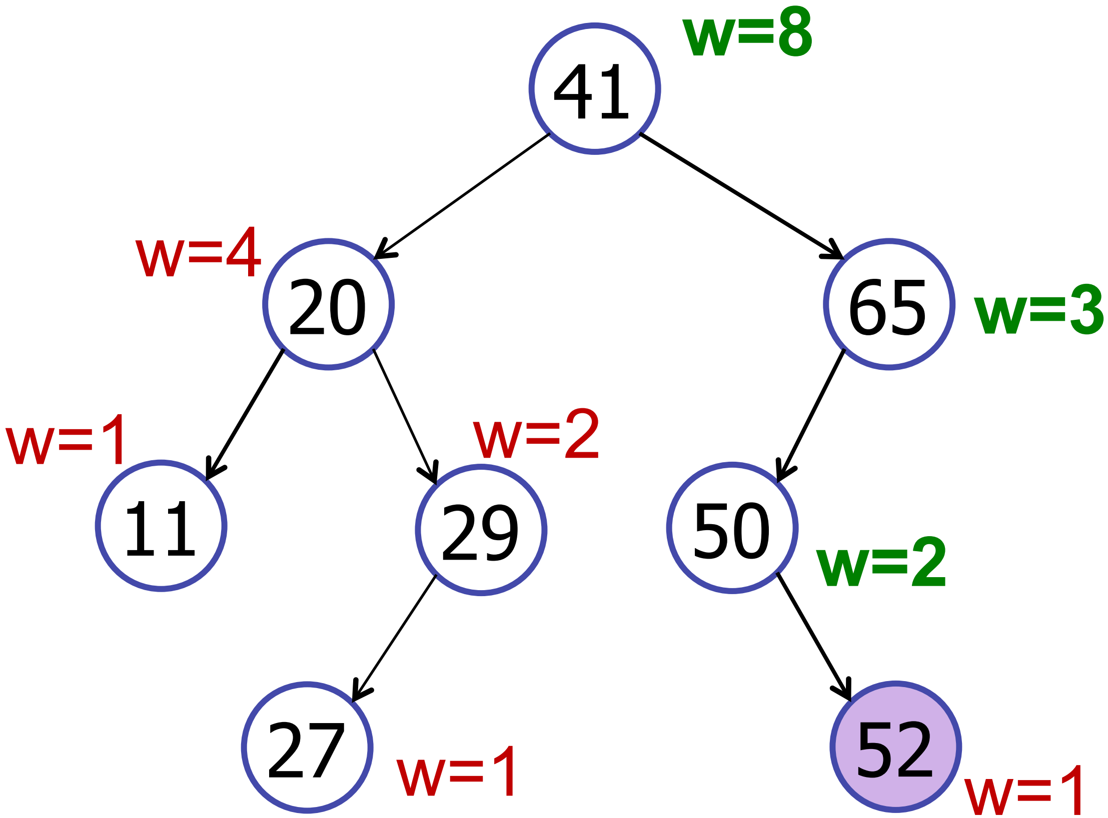
Augmented Trees

Maintain weight during insertions:



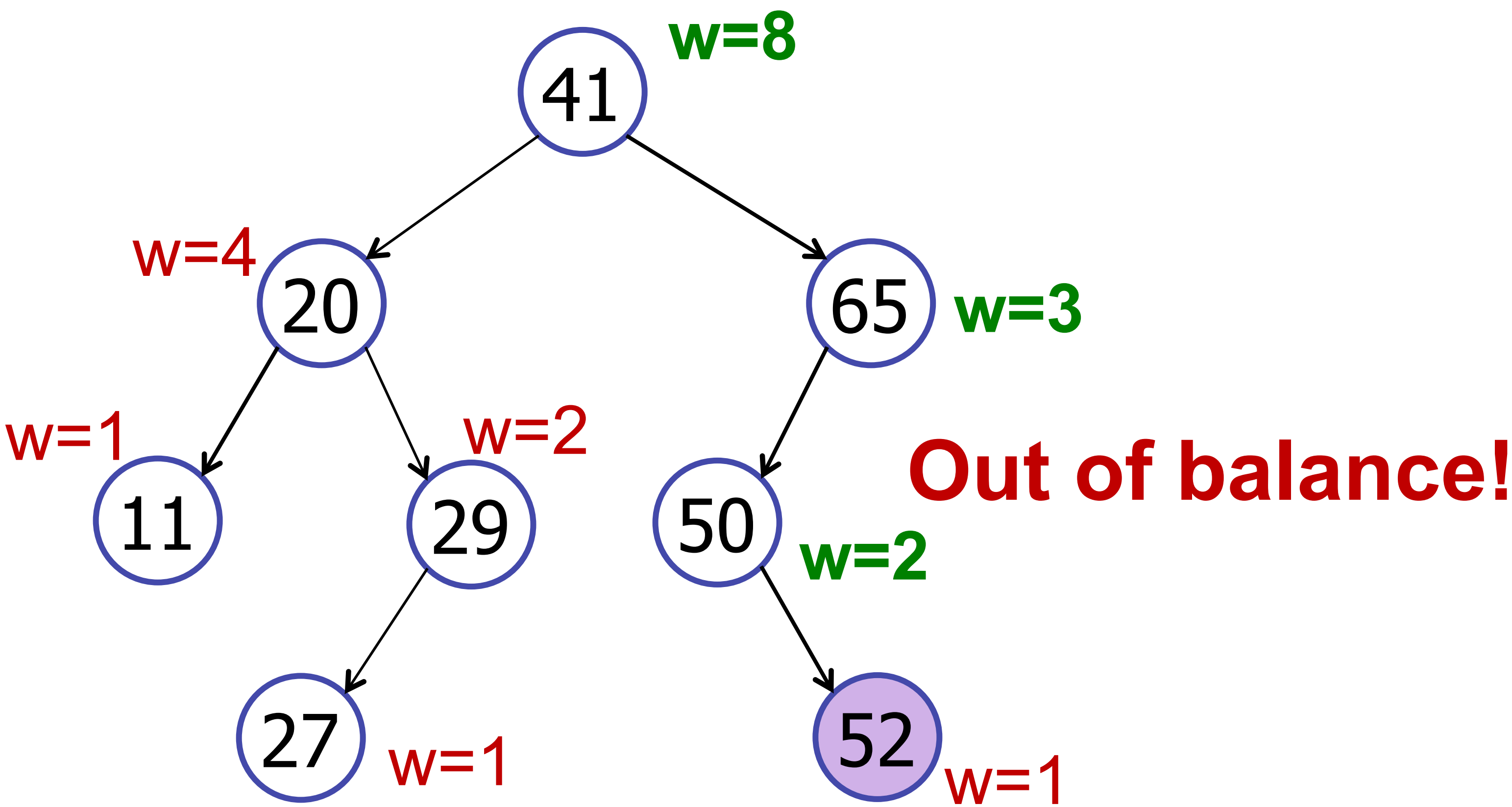
Augmented Trees

Maintain weight during insertions:



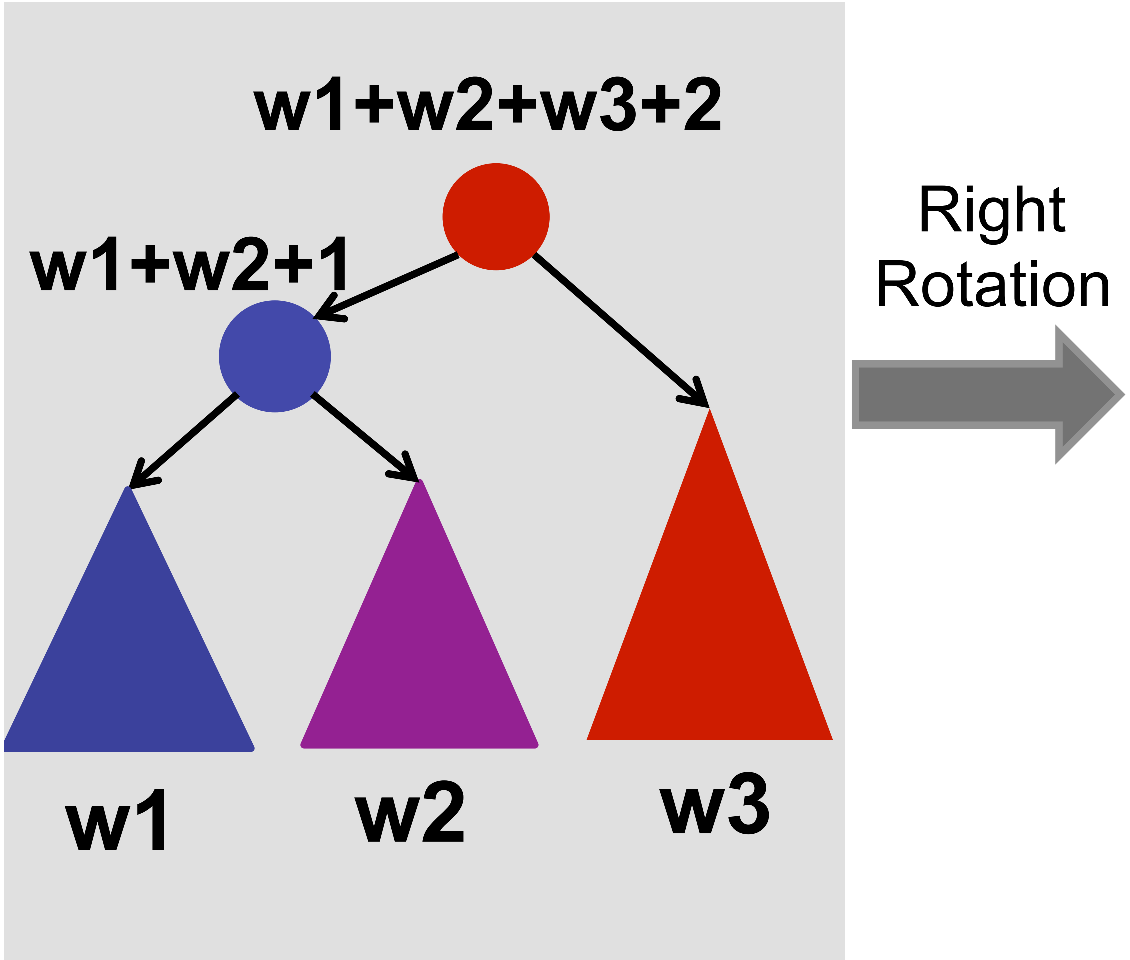
Augmented Trees

Maintain weight during insertions:



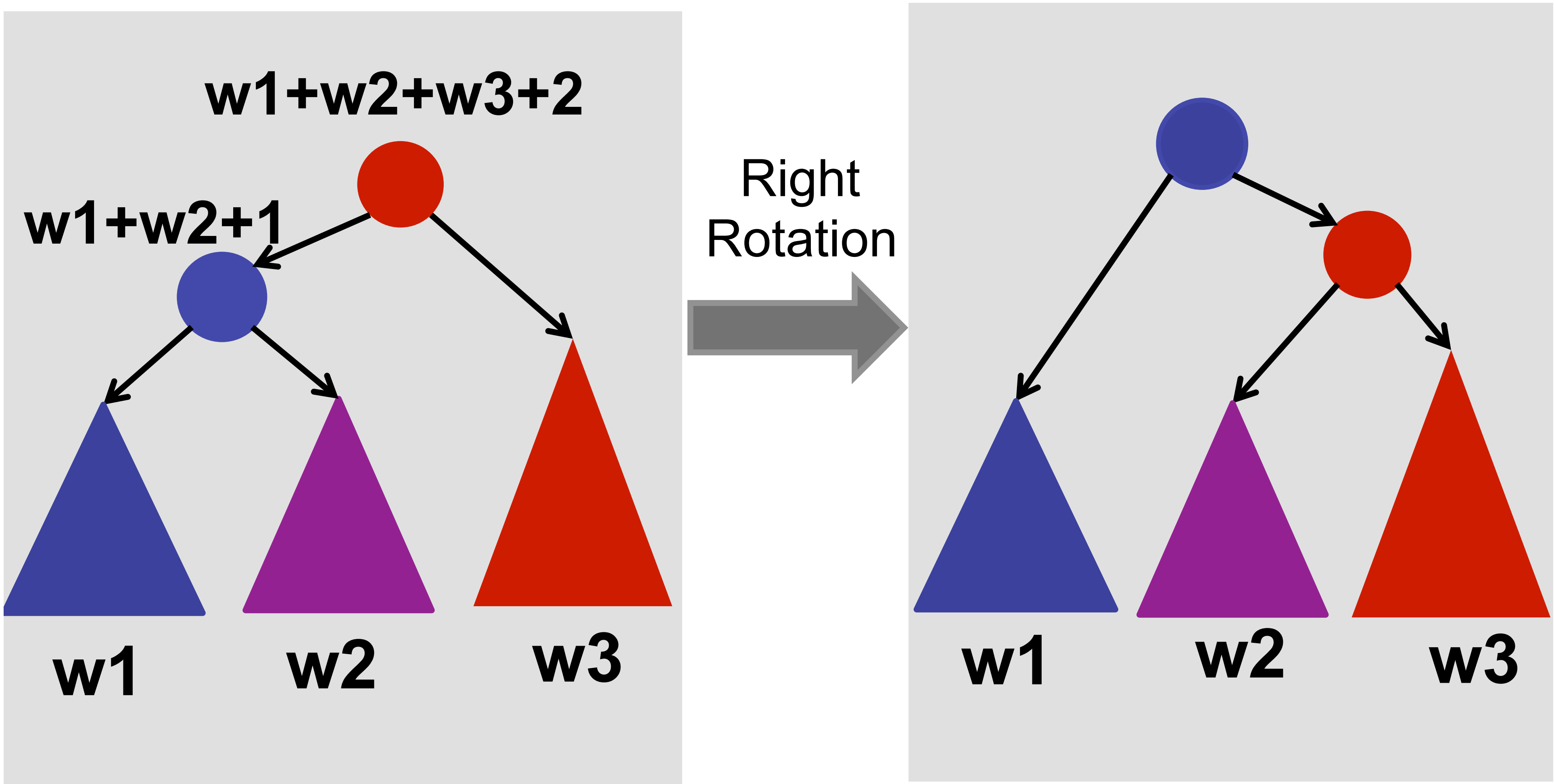
Augmented Trees

Maintain weight during rotations:



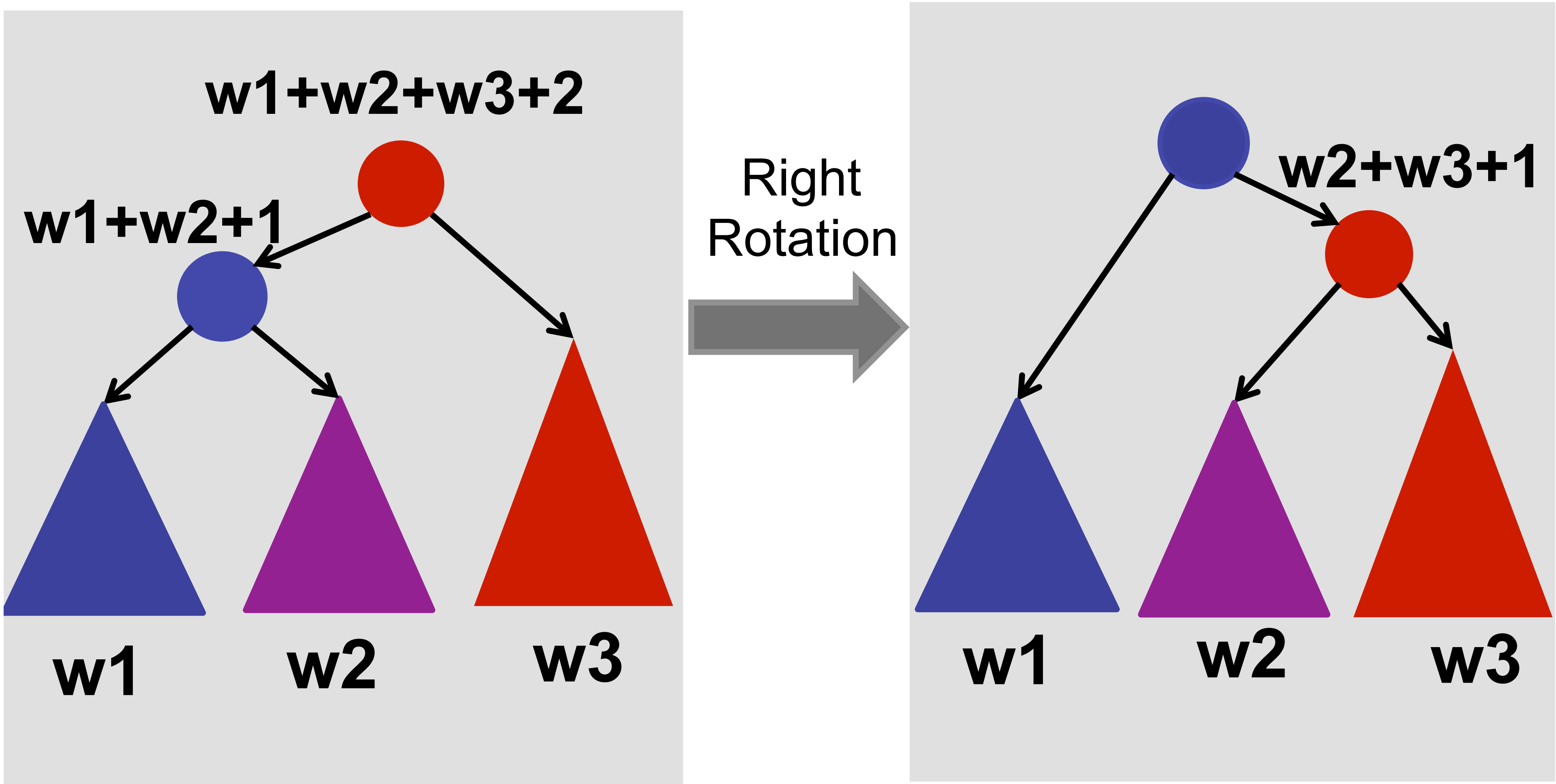
Augmented Trees

Maintain weight during rotations:



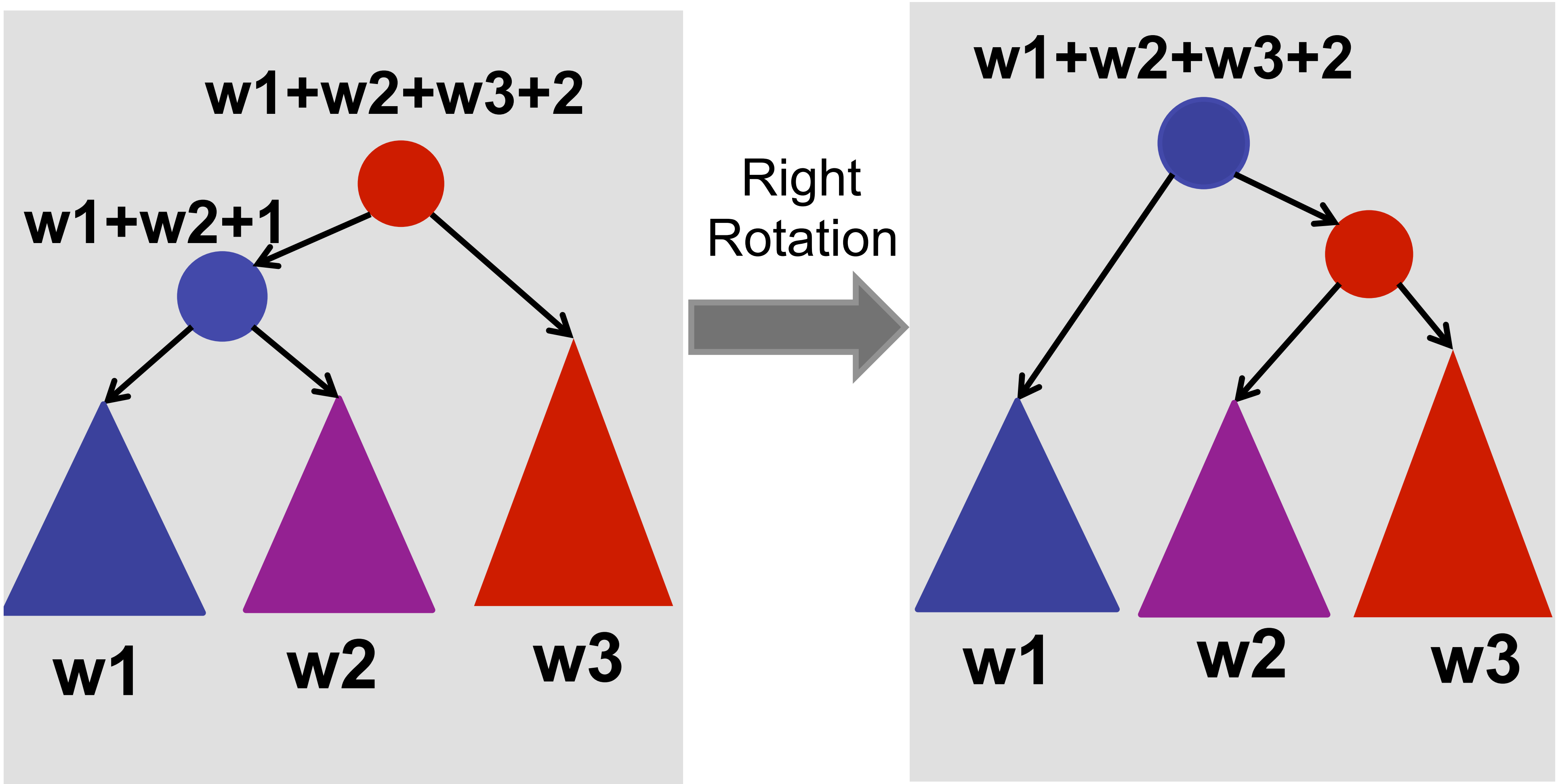
Augmented Trees

Maintain weight during rotations:



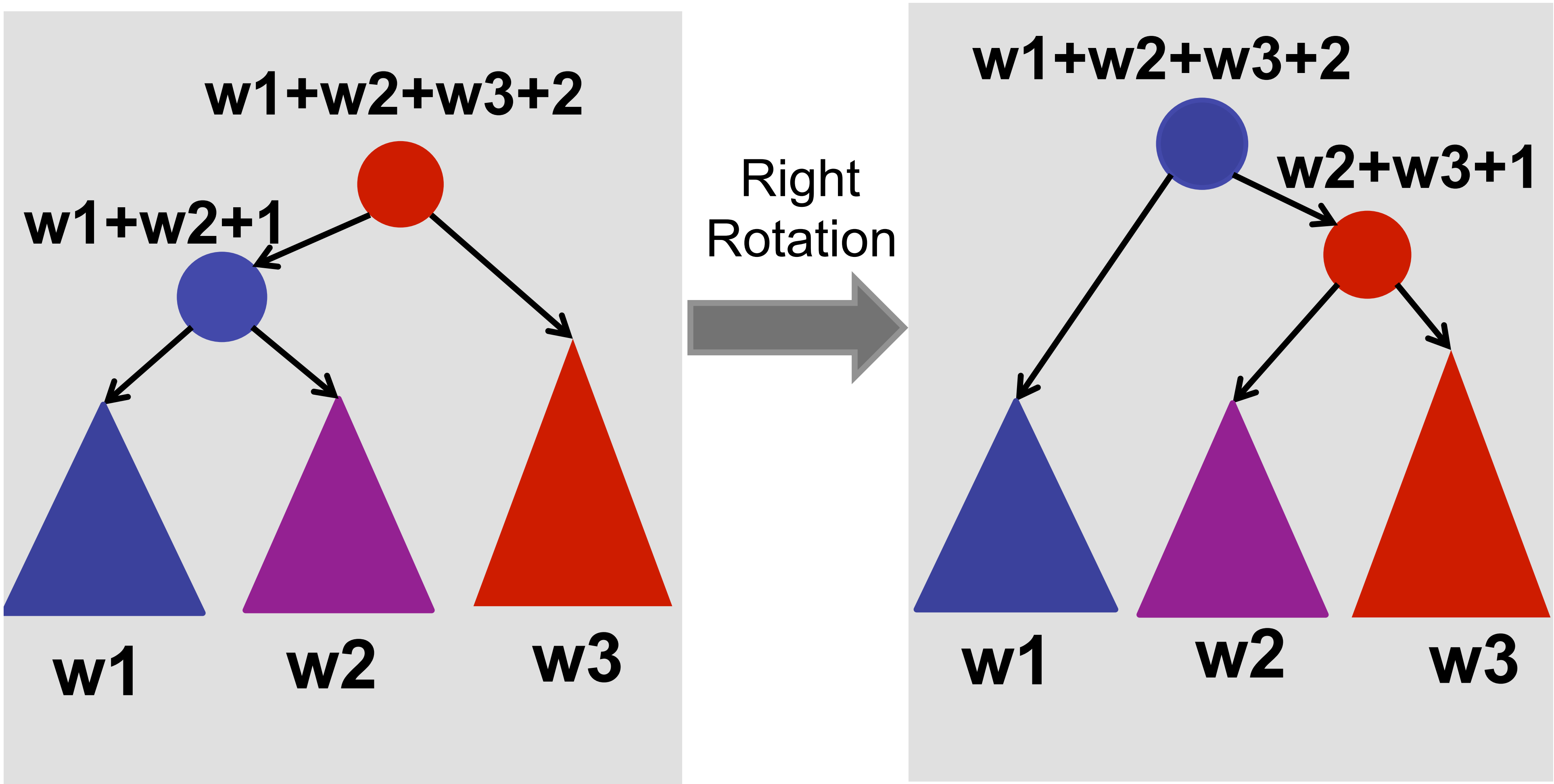
Augmented Trees

Maintain weight during rotations:



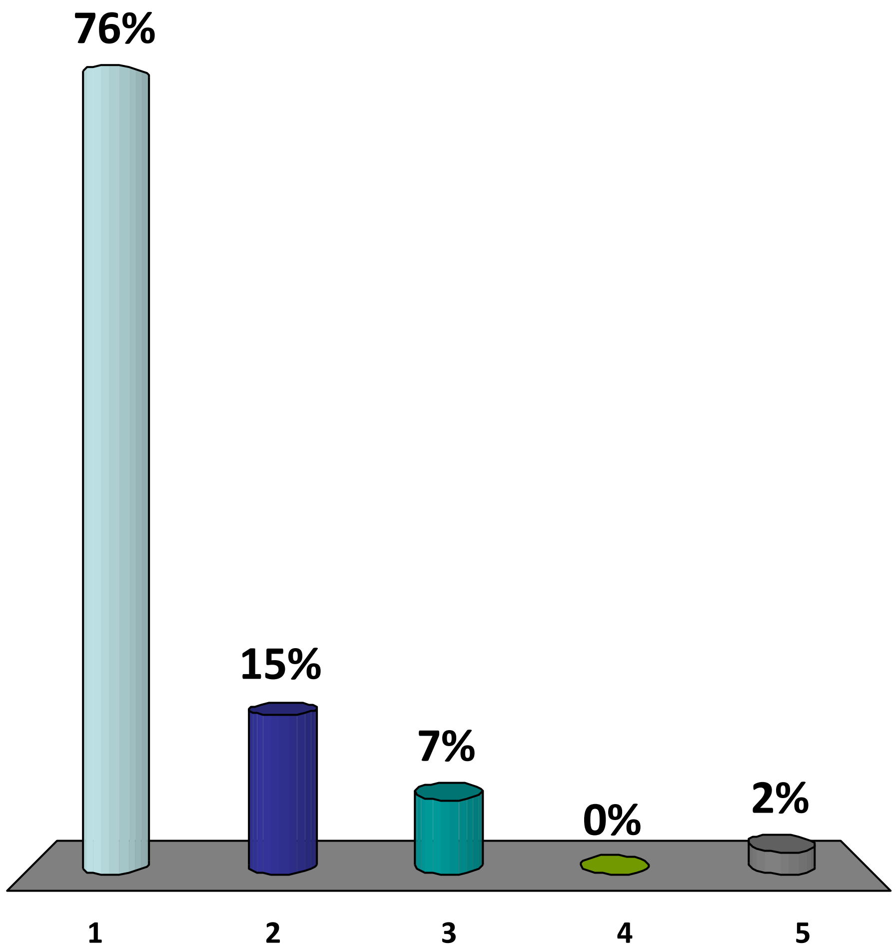
Augmented Trees

Maintain weight during rotations:



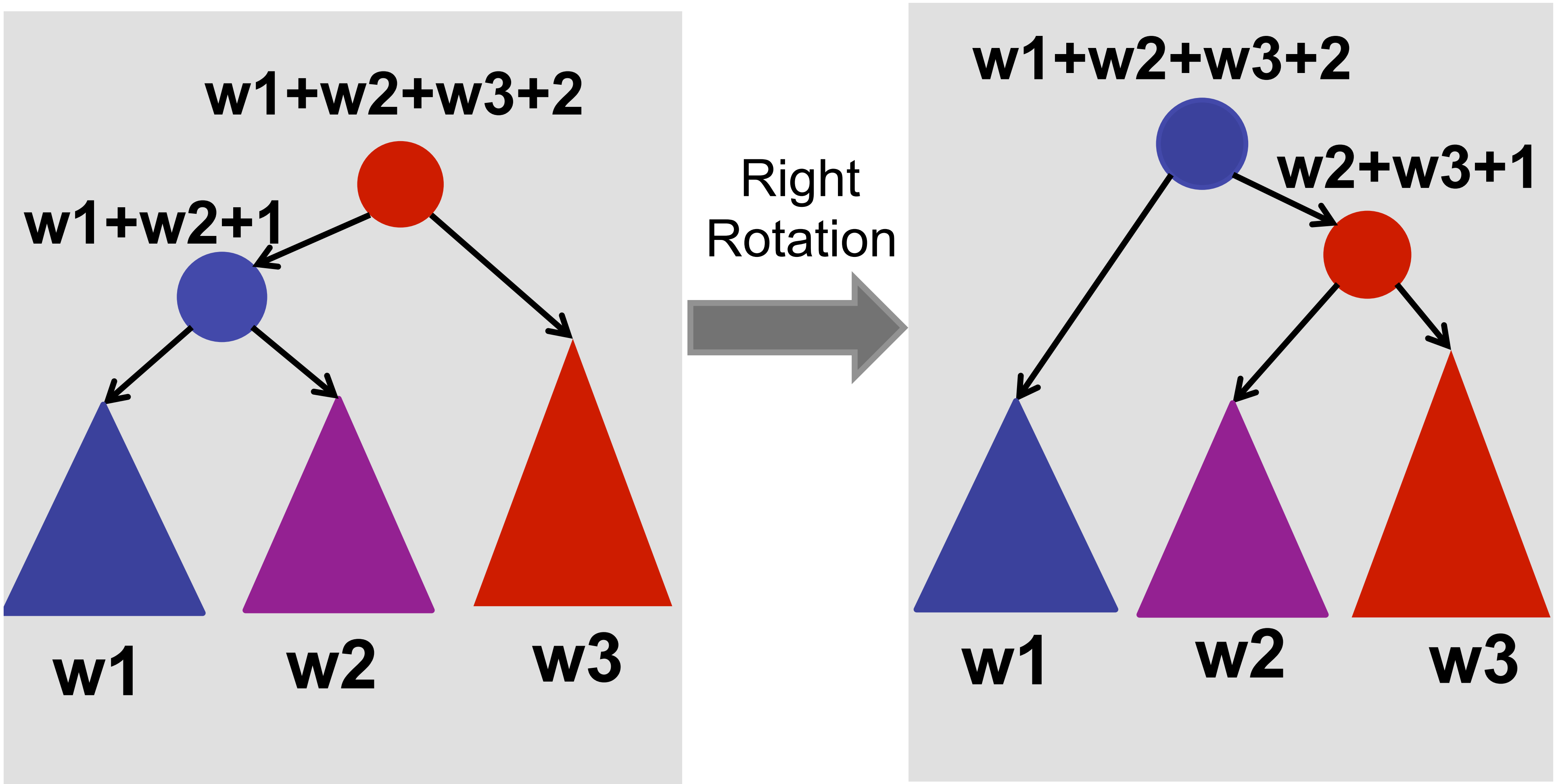
How long does it take to update the weights during a rotation?

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



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

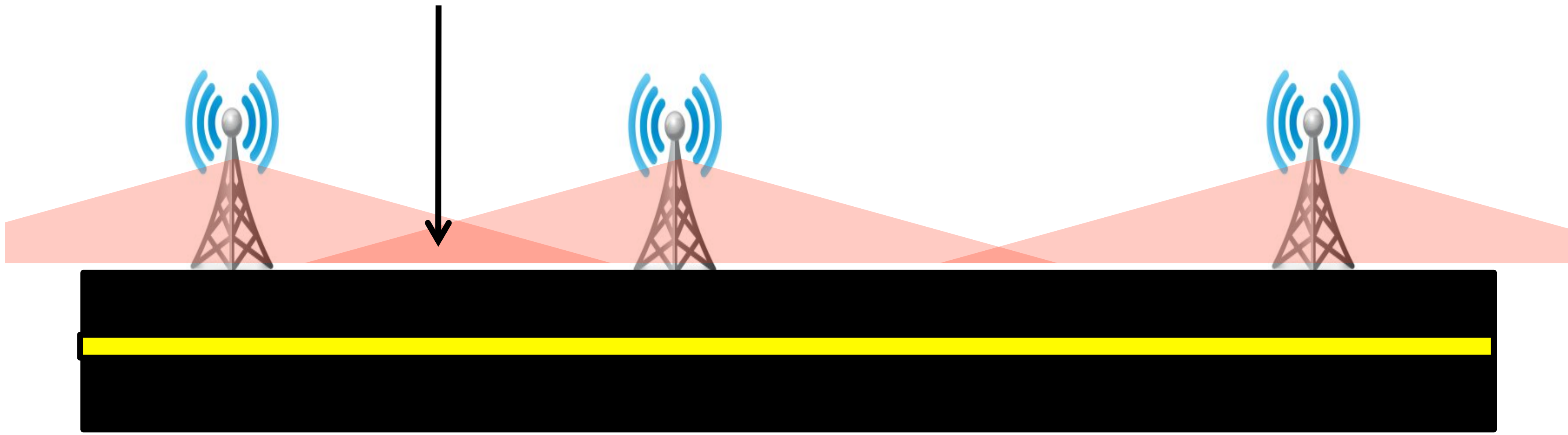
1. Order Statistics

2. Intervals

3. Orthogonal Range Searching

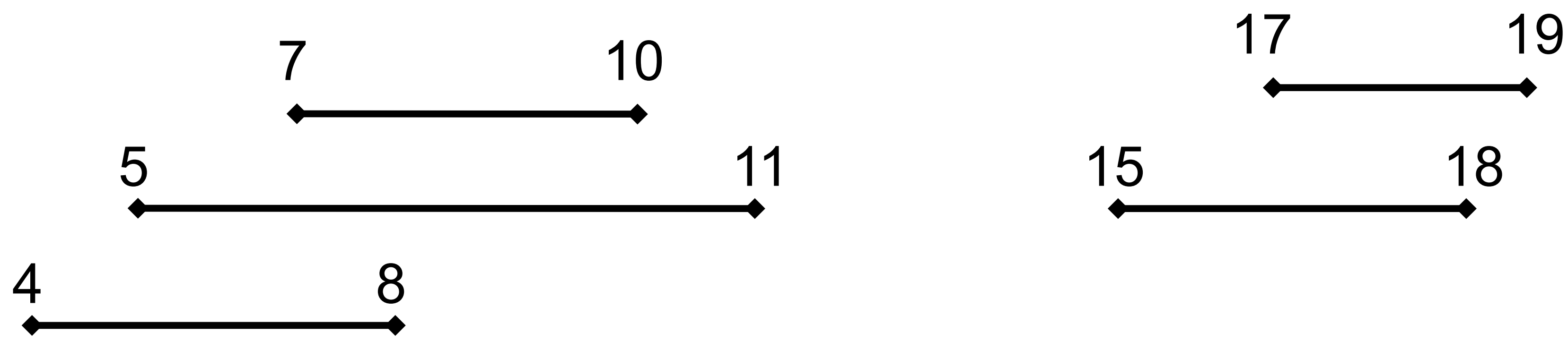
Cell Tower Coverage

Find a tower that covers my location.



Cell Tower Coverage

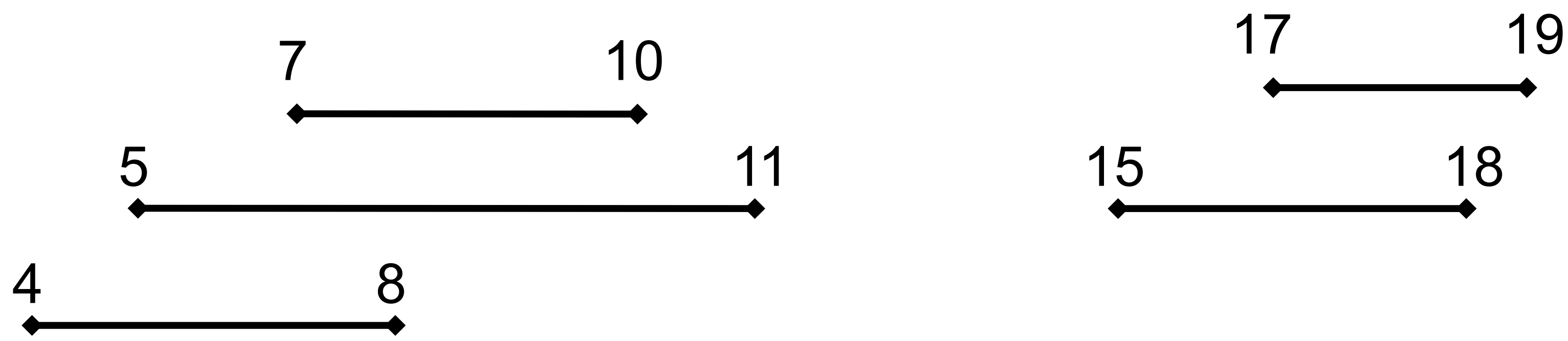
Find a tower that covers my location.



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

Cell Tower Coverage

Find a tower that covers my location.

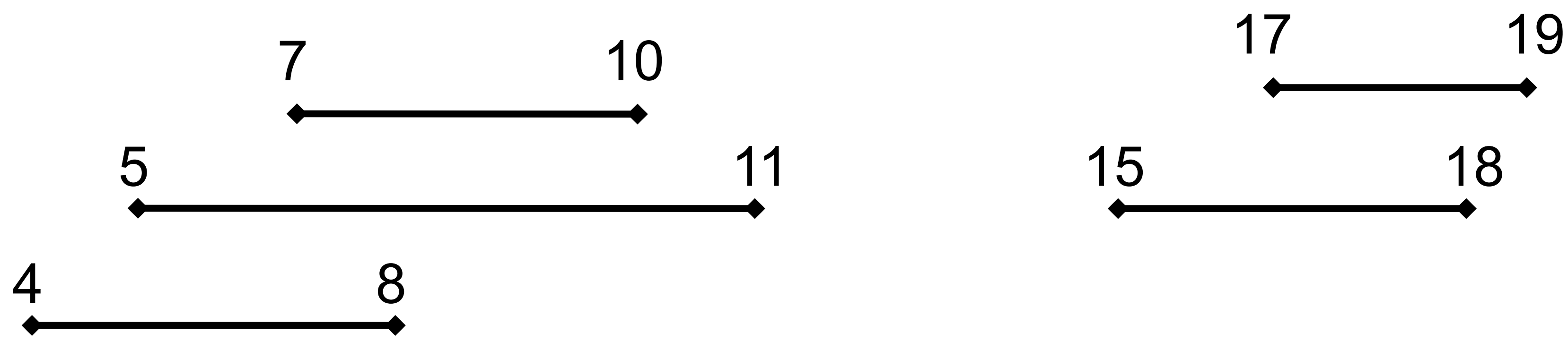


insert(begin, end)
delete(begin, end)

Query: find an interval that overlaps x.

Cell Tower Coverage

Find a tower that covers my location.



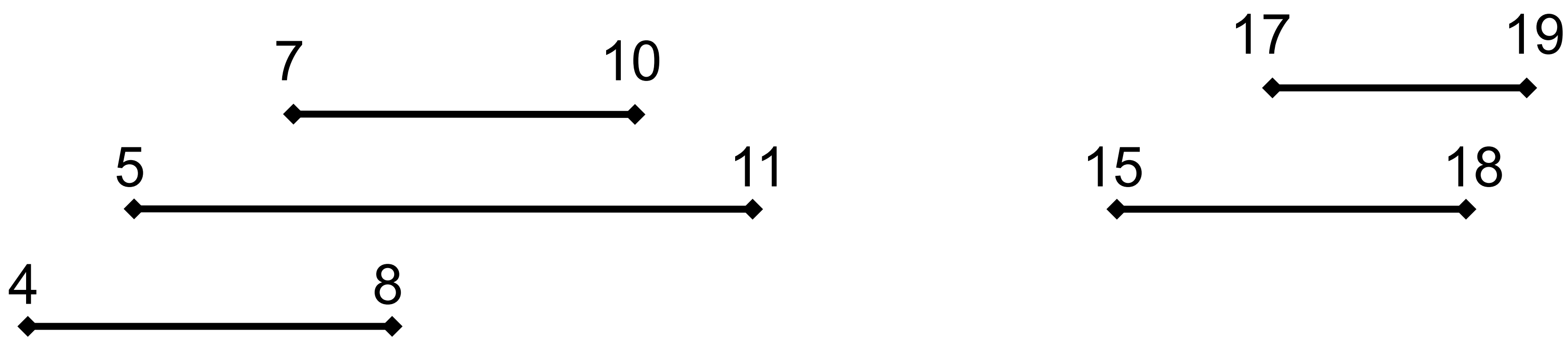
Solution 1: Keep intervals in a list.

Query: scan entire list.

Does sorting help?

Cell Tower Coverage

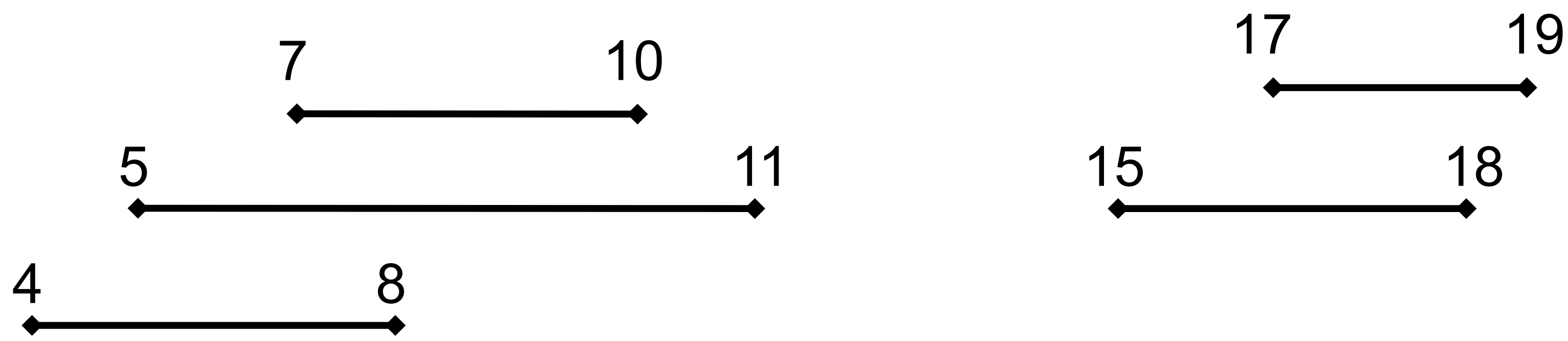
Find a tower that covers my location.



Solution 2: O(1) queries??

Cell Tower Coverage

Find a tower that covers my location.



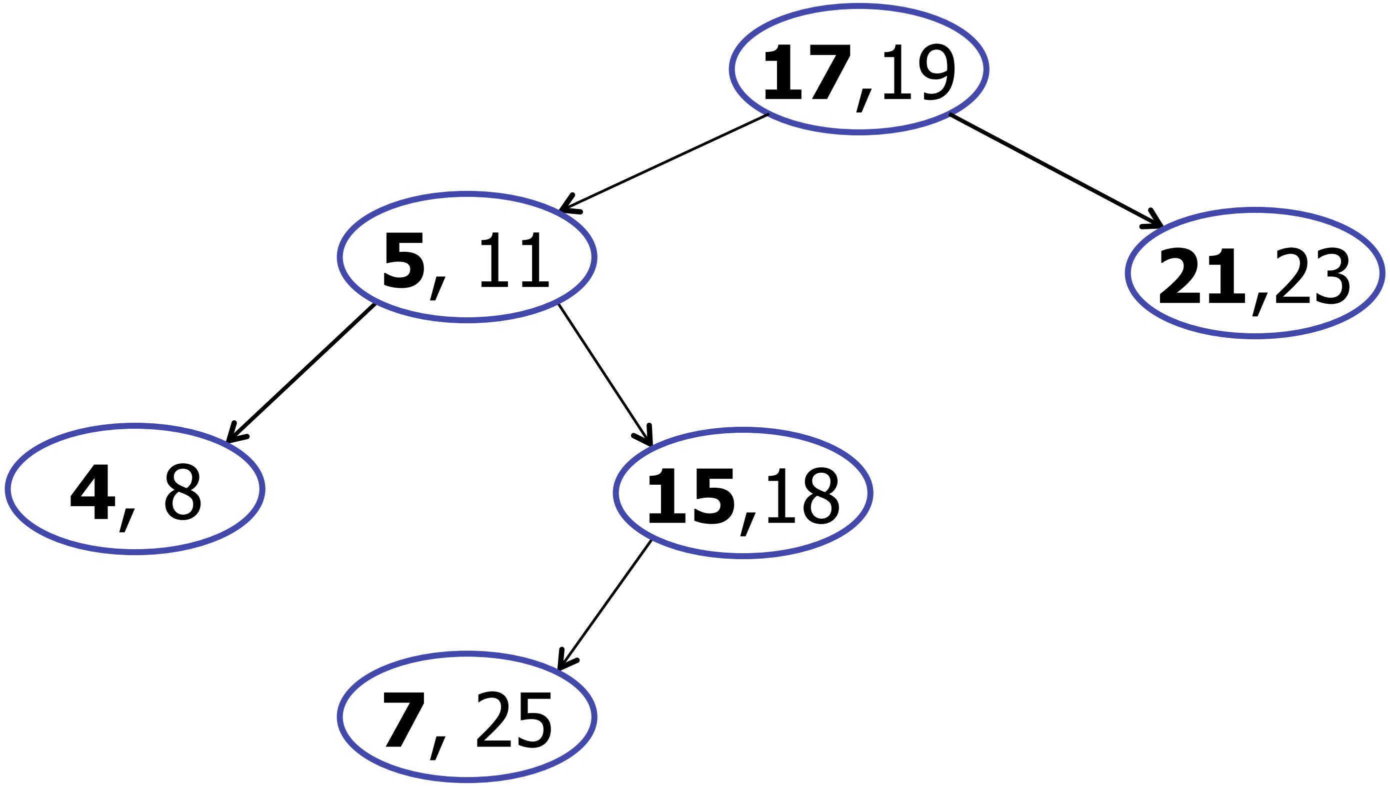
Solution 2: O(1) queries??

			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

Problems??

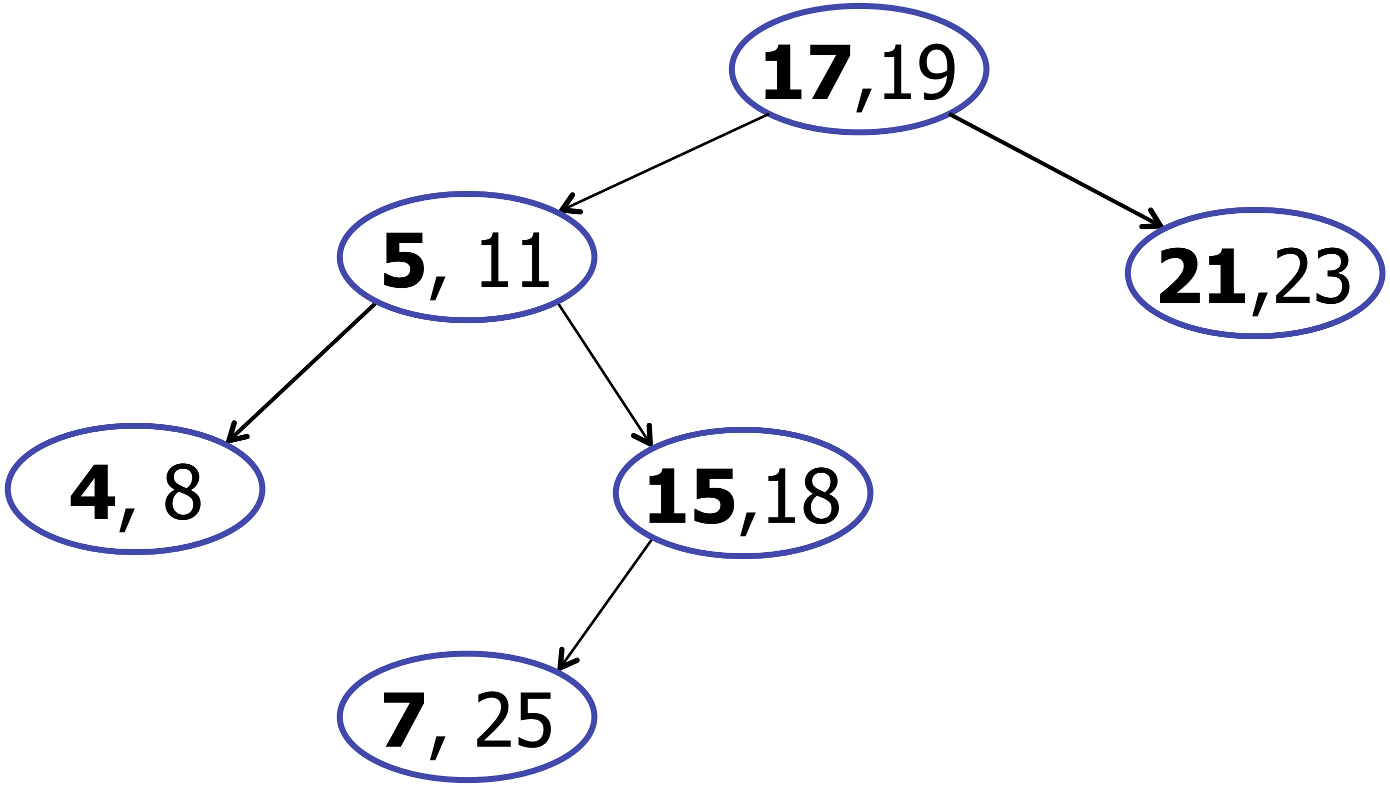
Interval Trees

Sorted by left endpoint



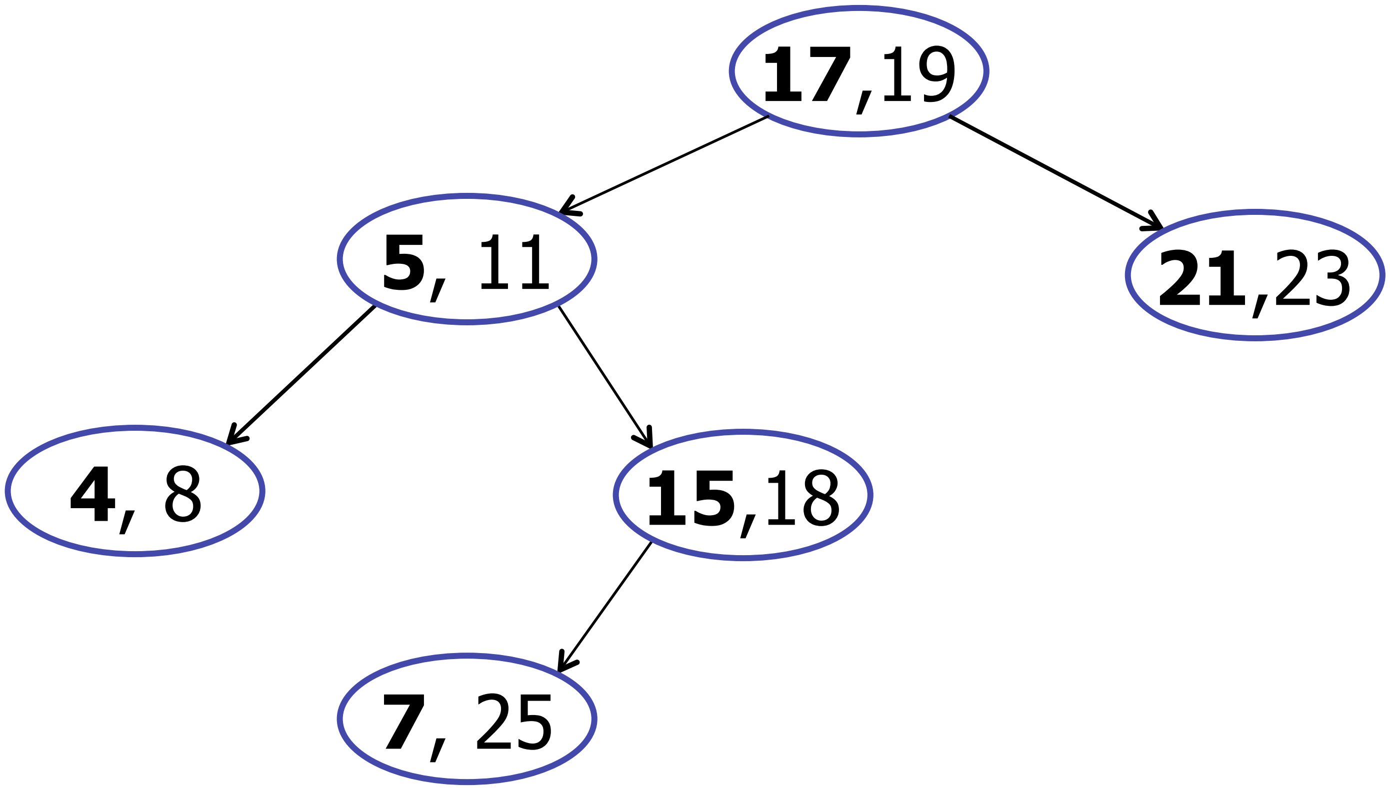
Interval Trees

search-interval(25) = ?



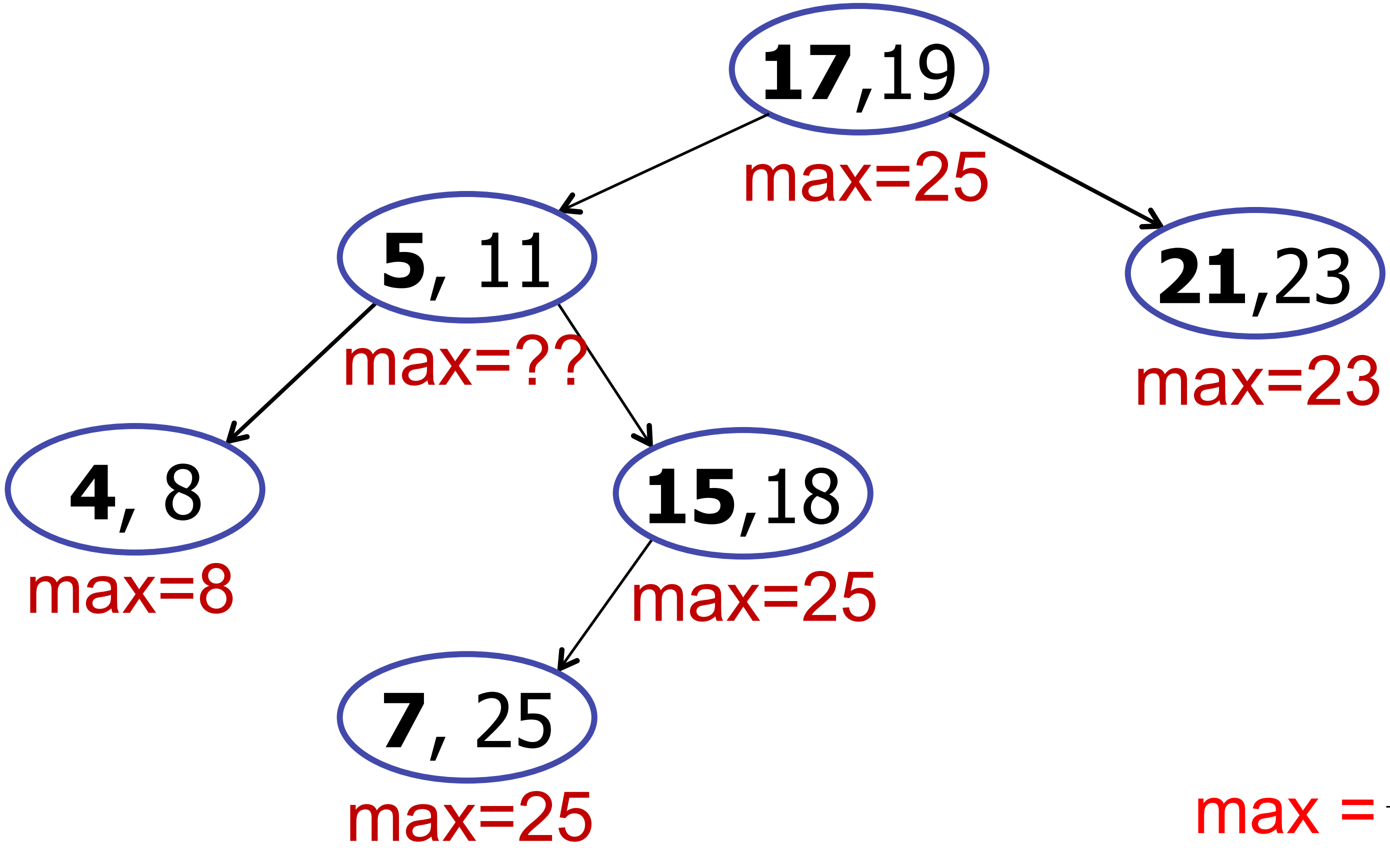
Interval Trees

Augment: ??



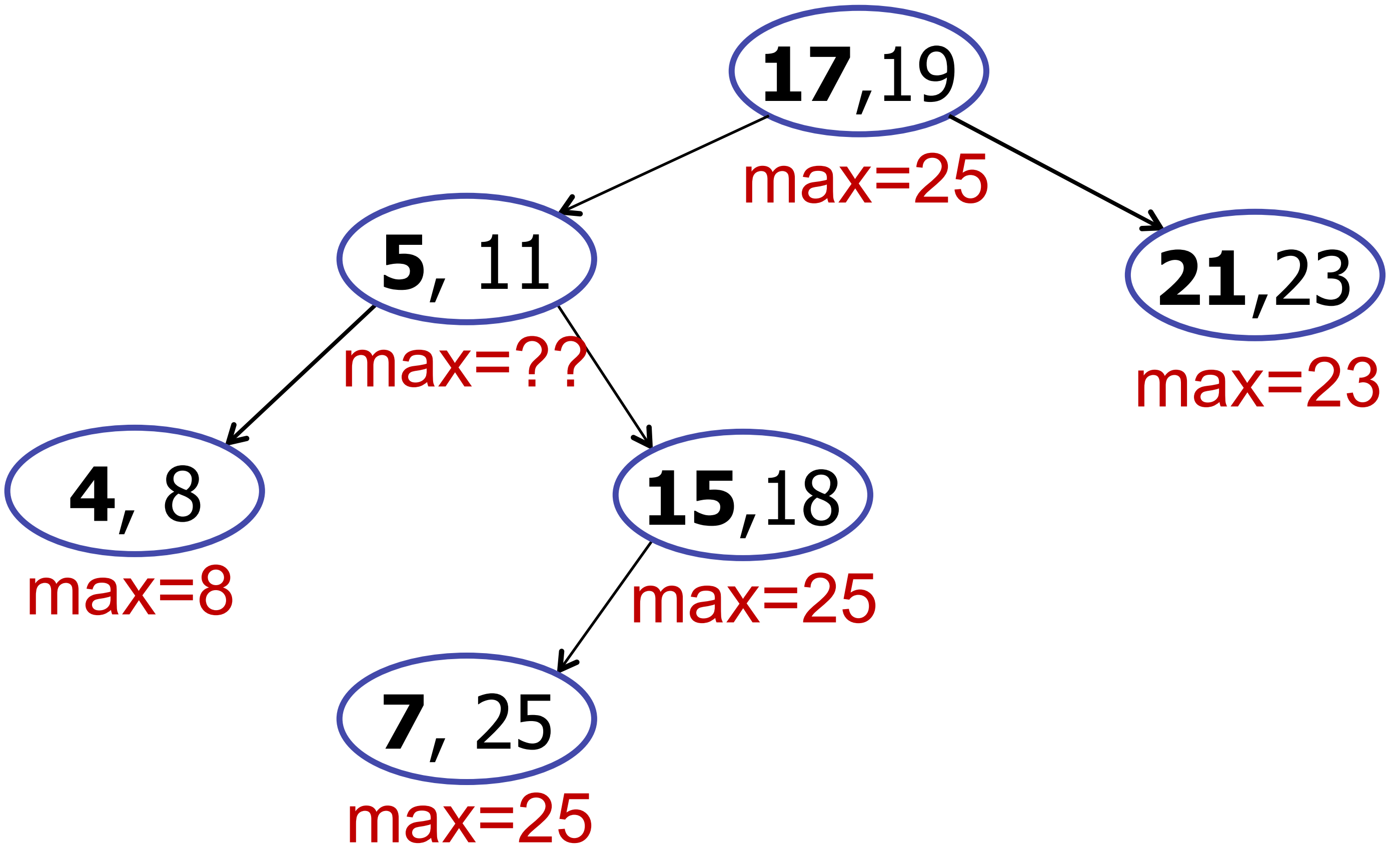
Interval Trees

Augment: maximum endpoint in subtree

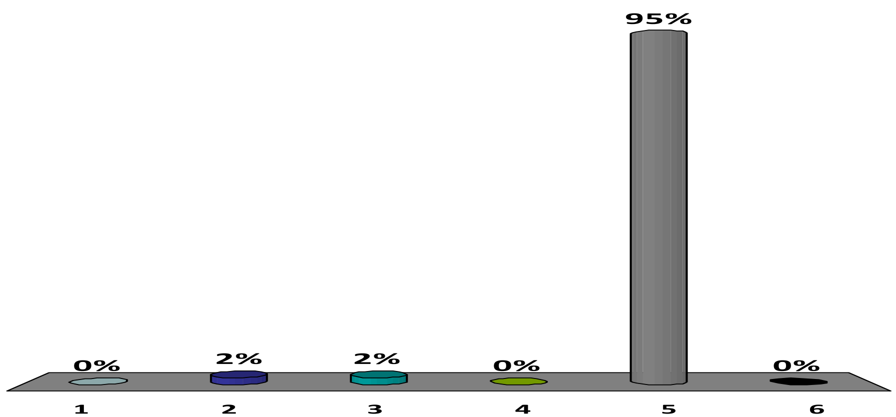


max = high / left.max / right.max

max=??

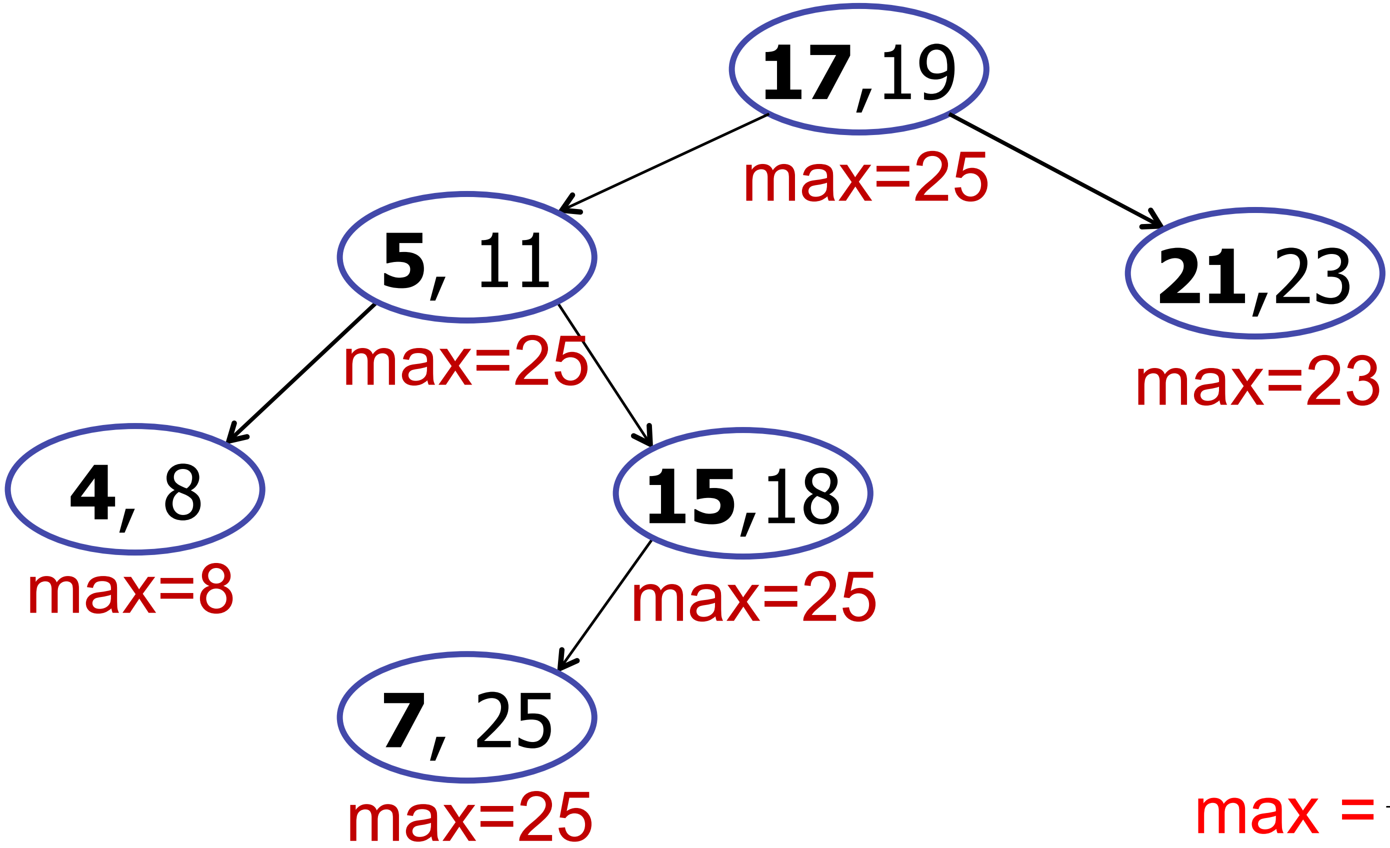


1. 5
2. 8
3. 11
4. 18
5. 25
6. 19



Interval Trees

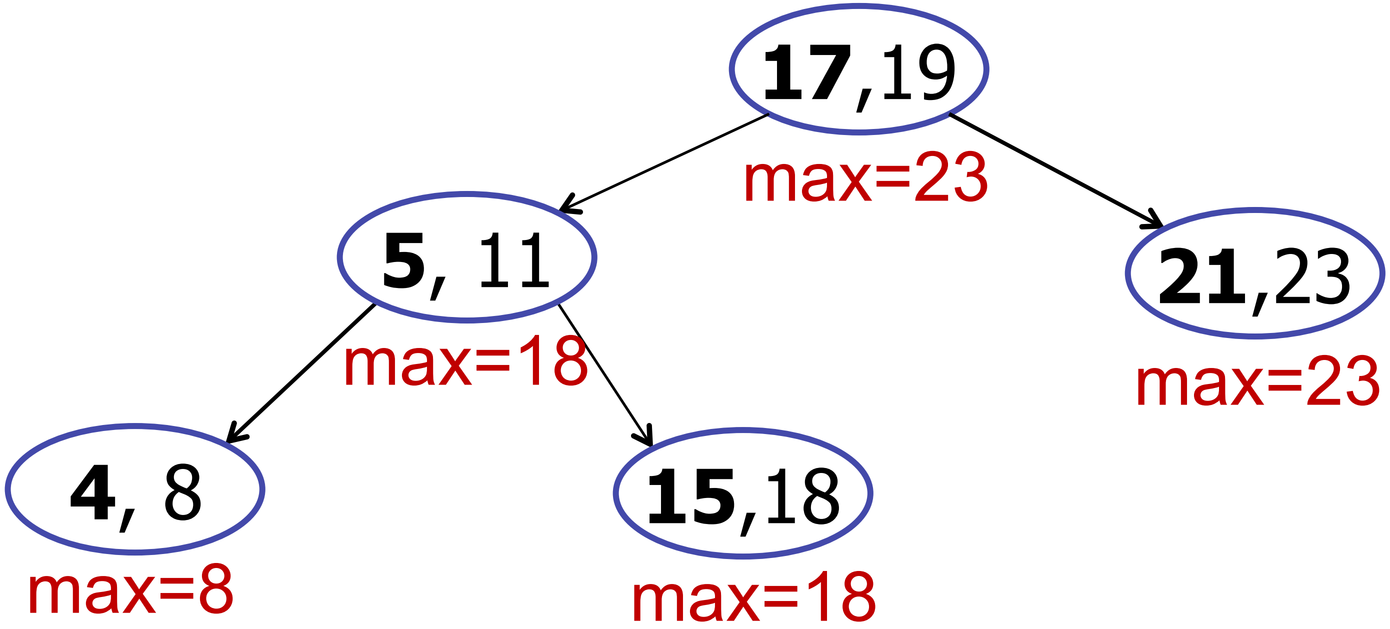
Augment: maximum endpoint in subtree



max = {high, left.max, right.max}

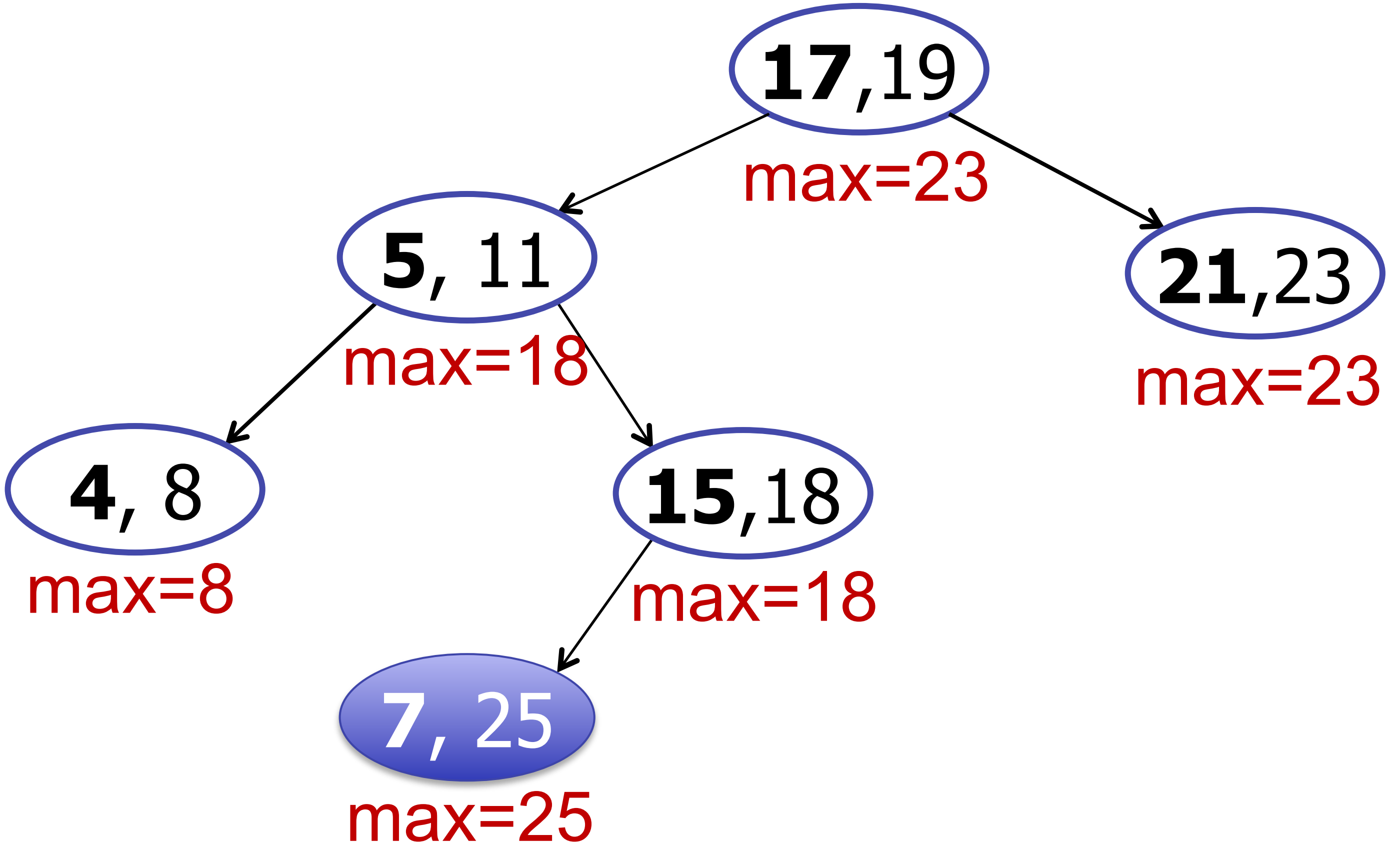
Interval Trees

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



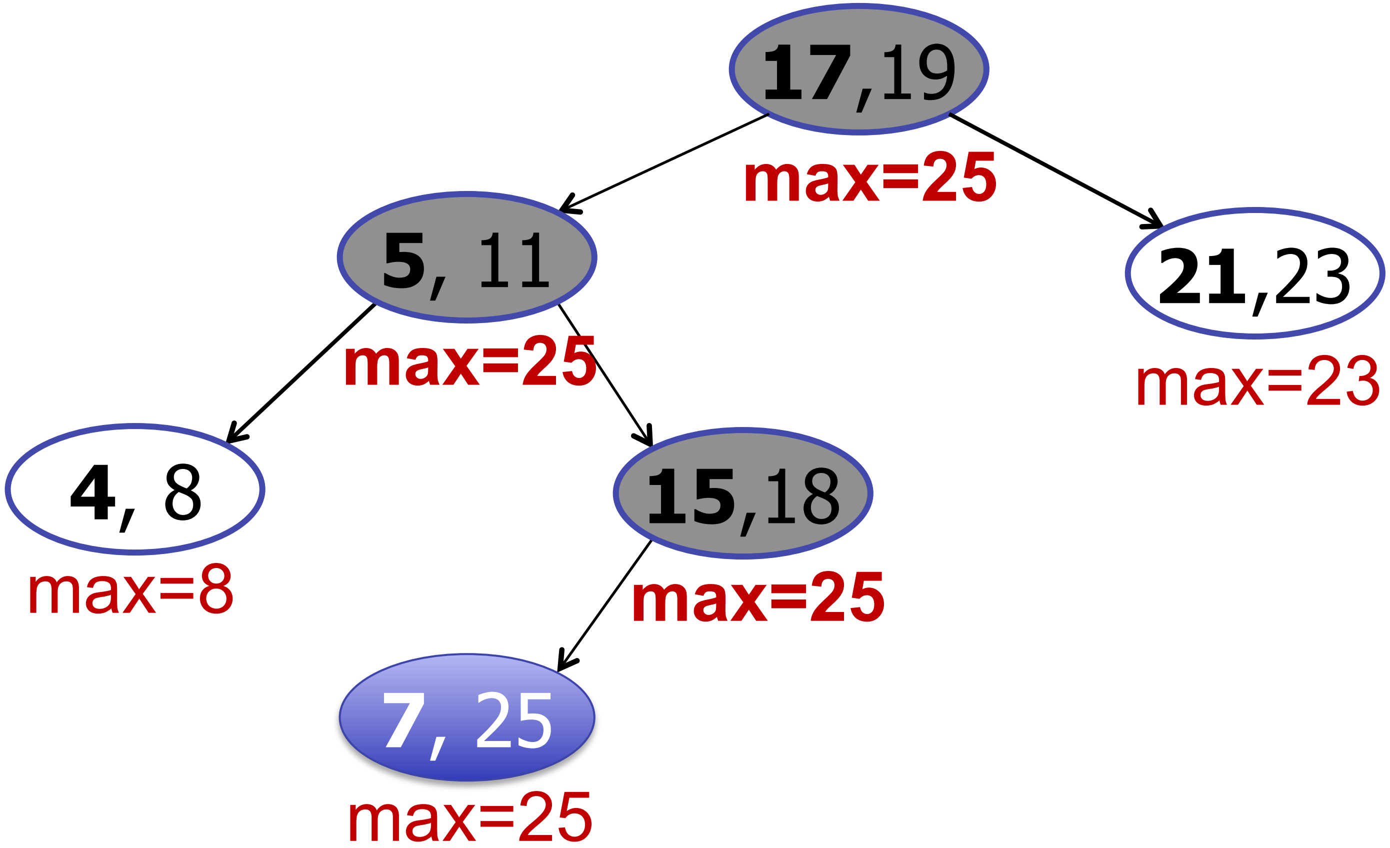
Interval Trees

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



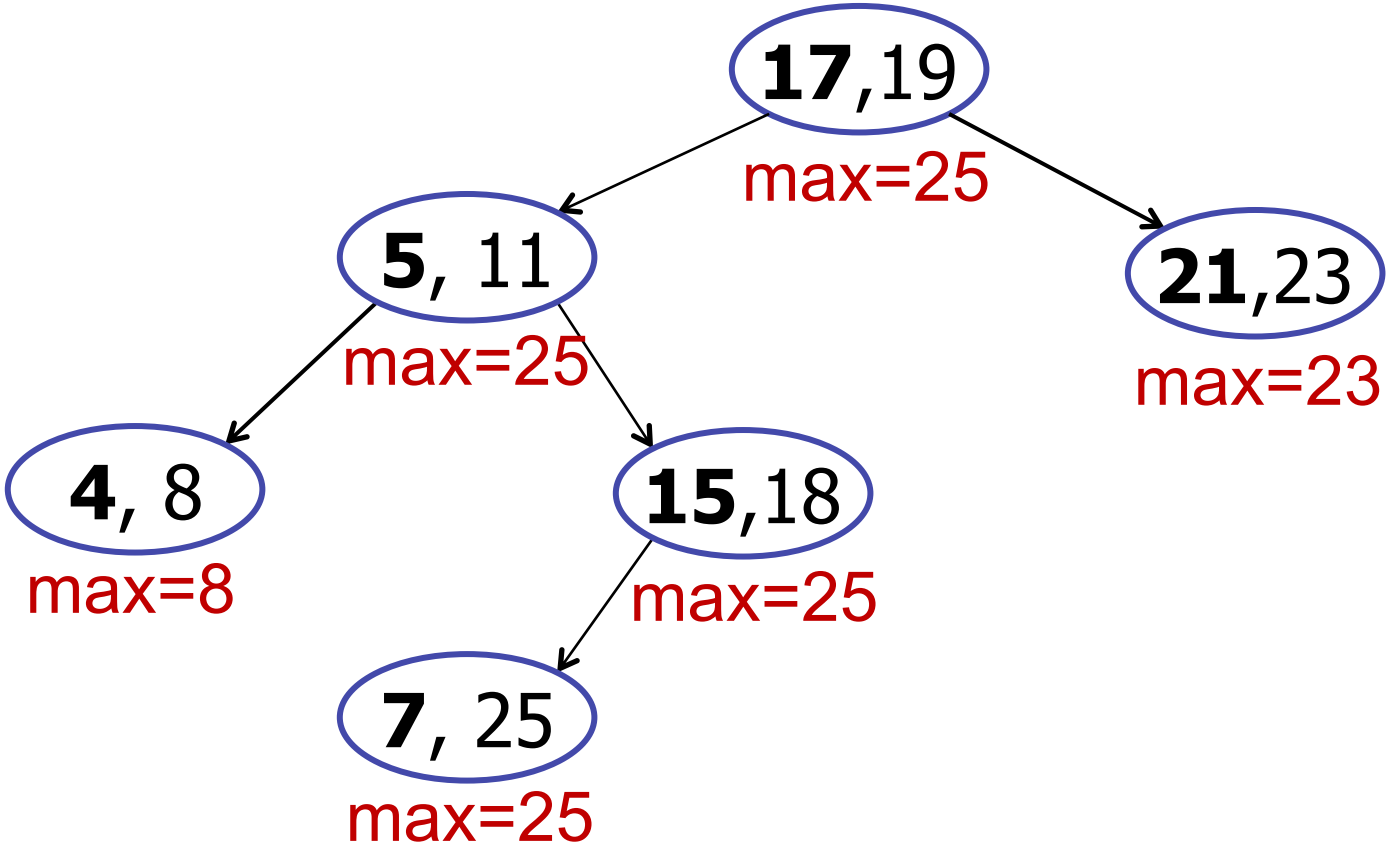
Interval Trees

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



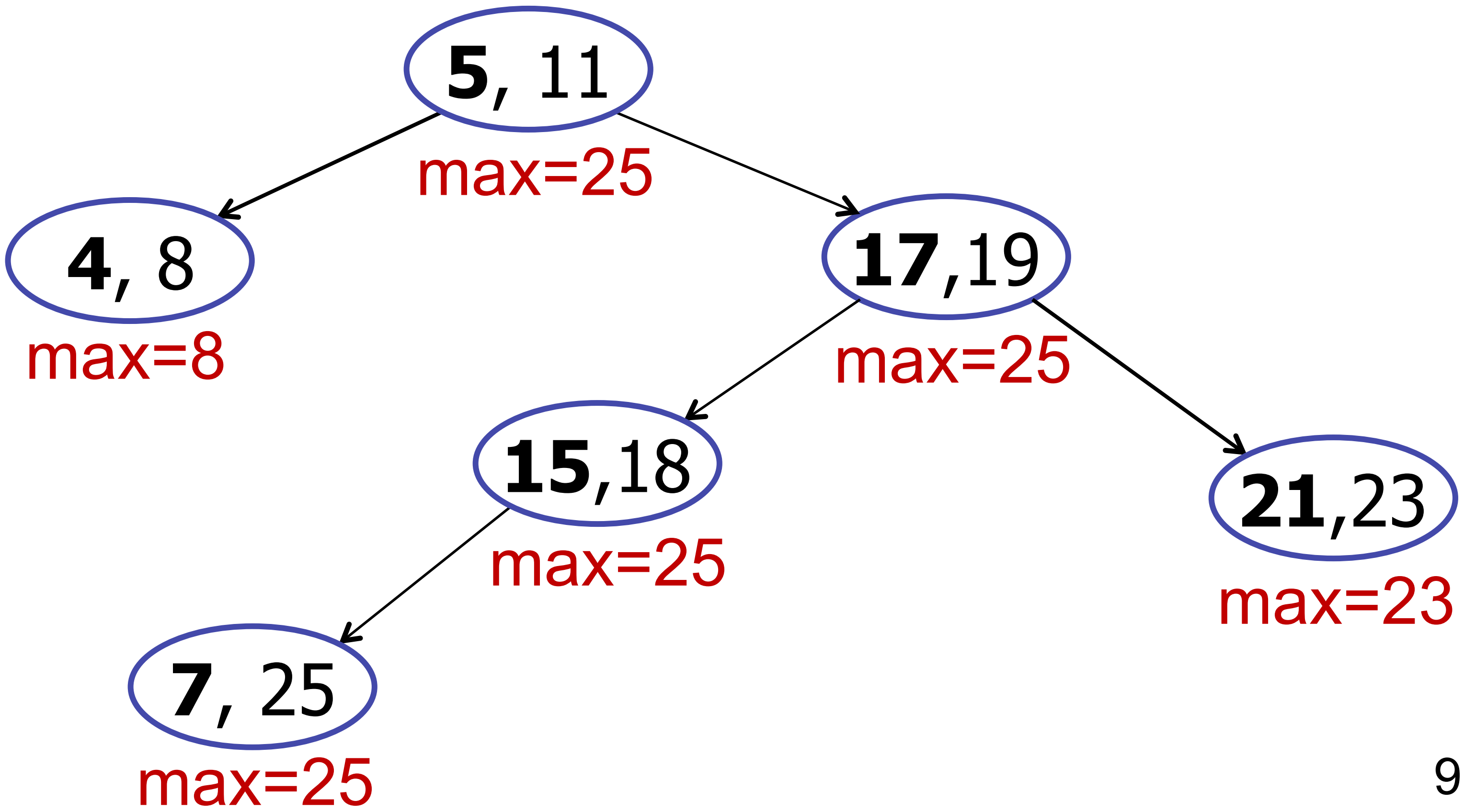
Interval Trees

Insertion: *out-of-balance*



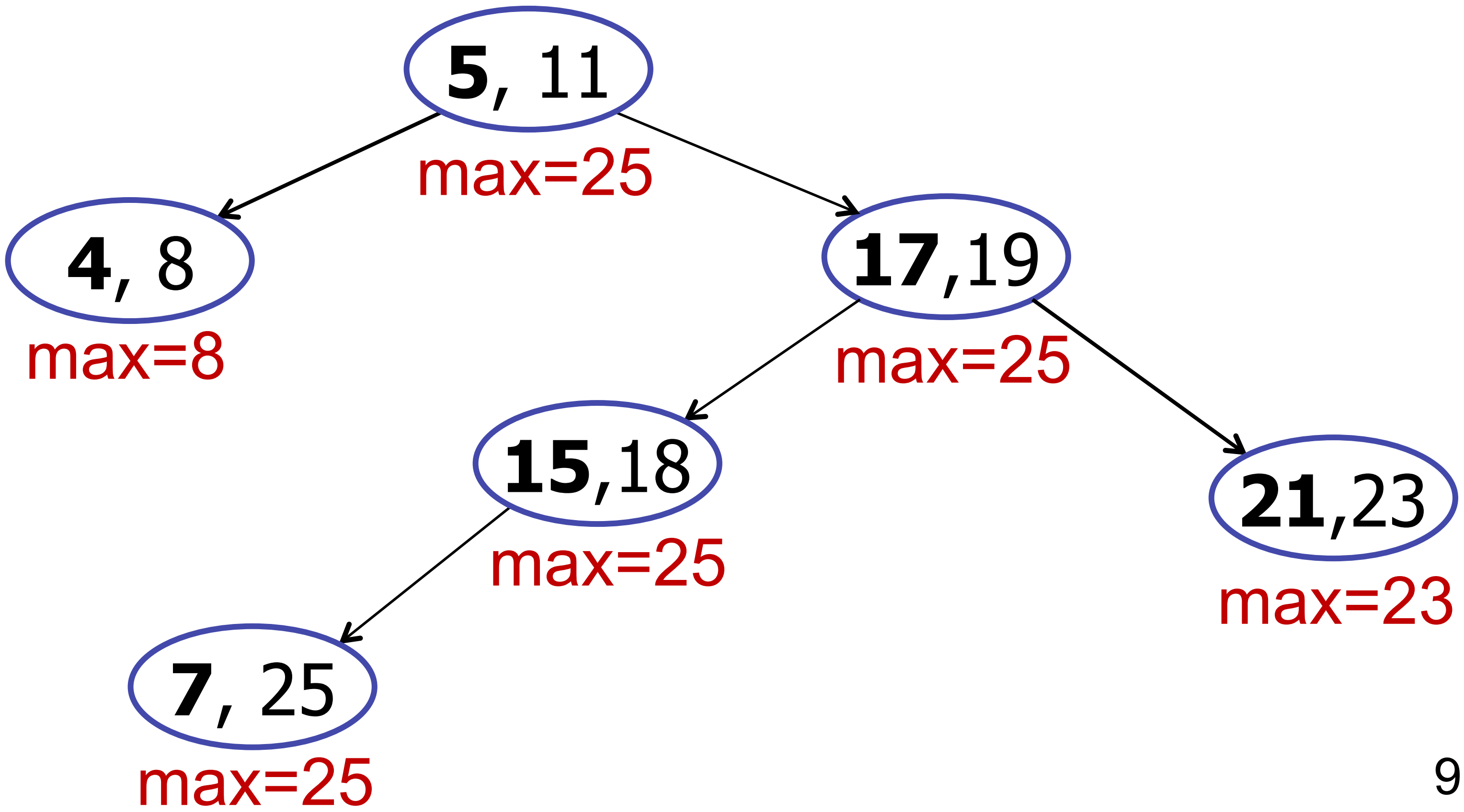
Interval Trees

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



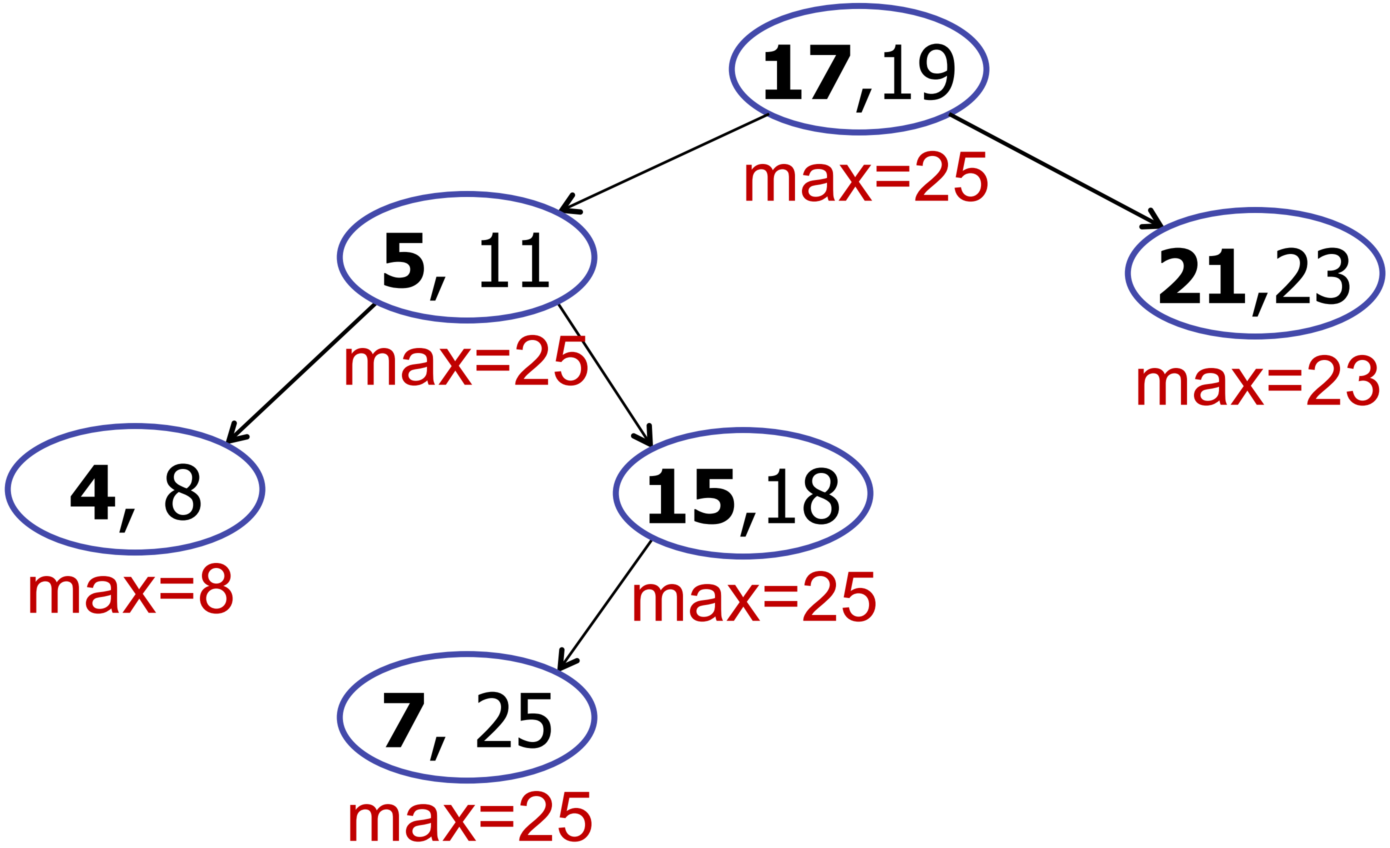
Interval Trees

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



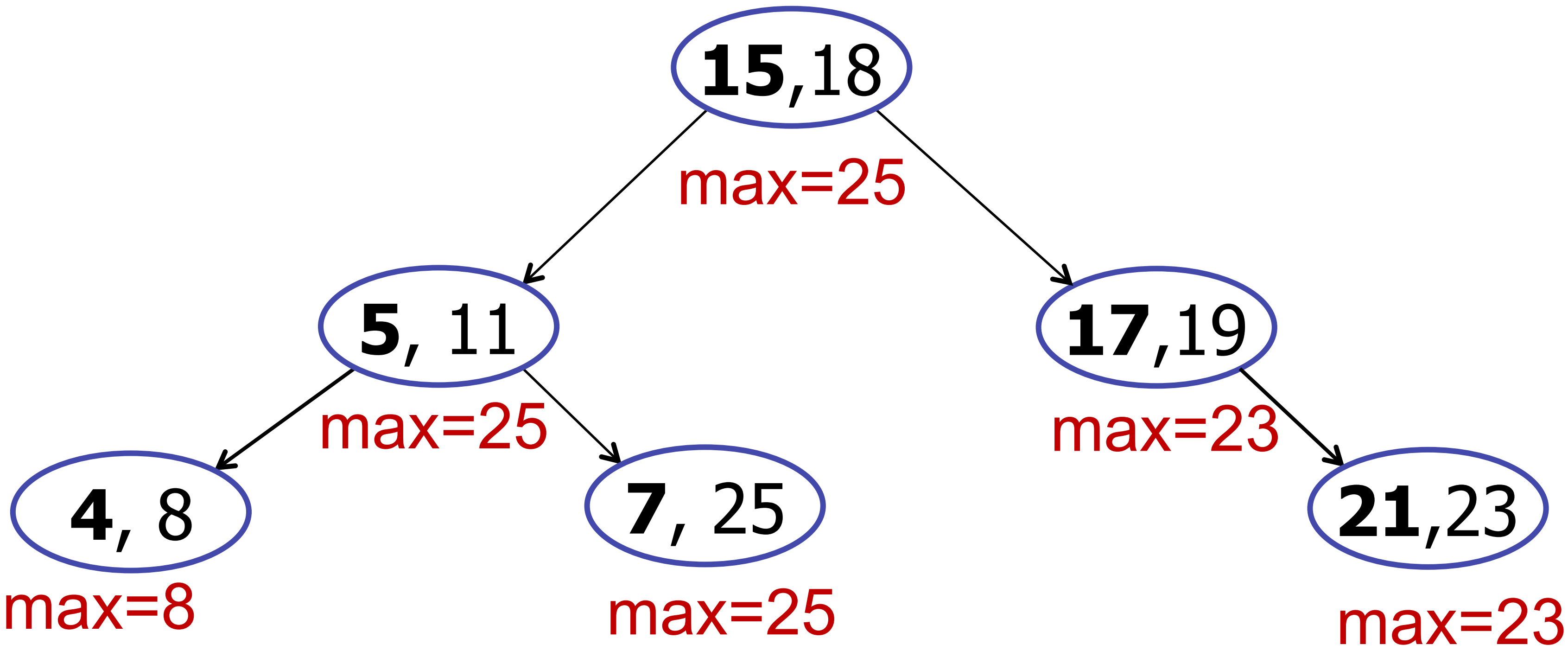
Interval Trees

Insertion: *out-of-balance*



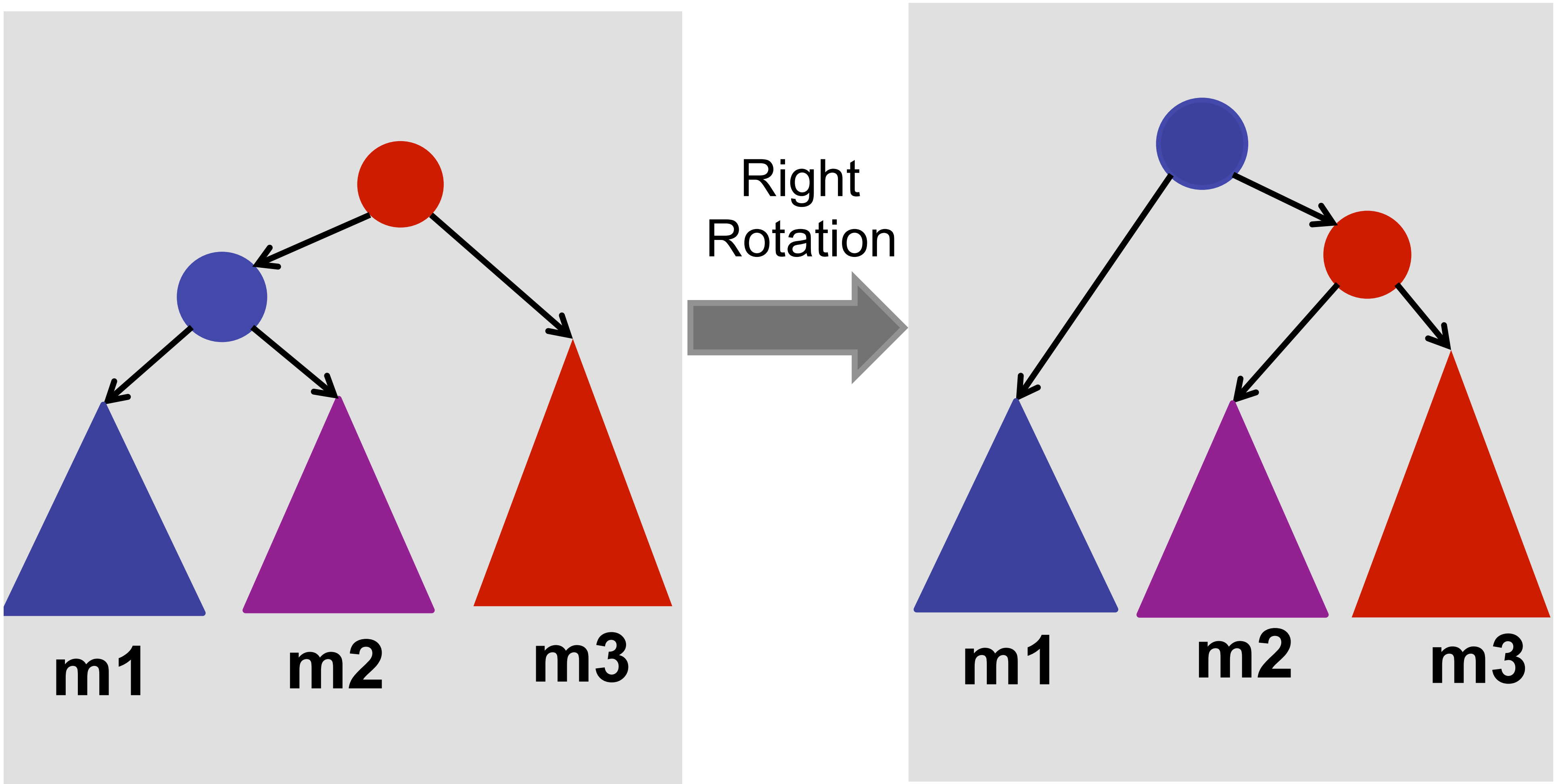
Interval Trees

Insertion: *left-rotate, right-rotate*



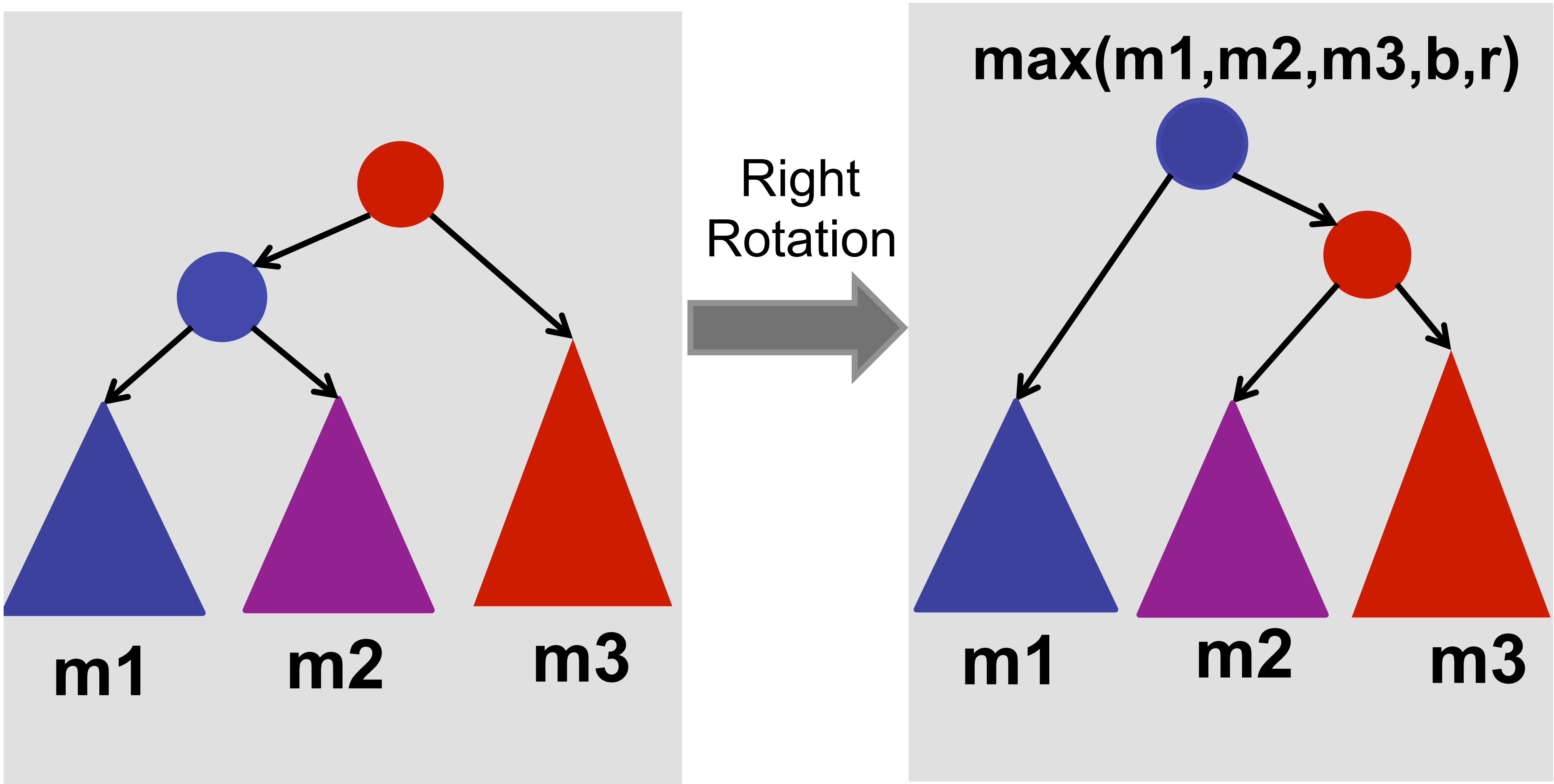
Interval Trees

Maintain MAX during rotations:



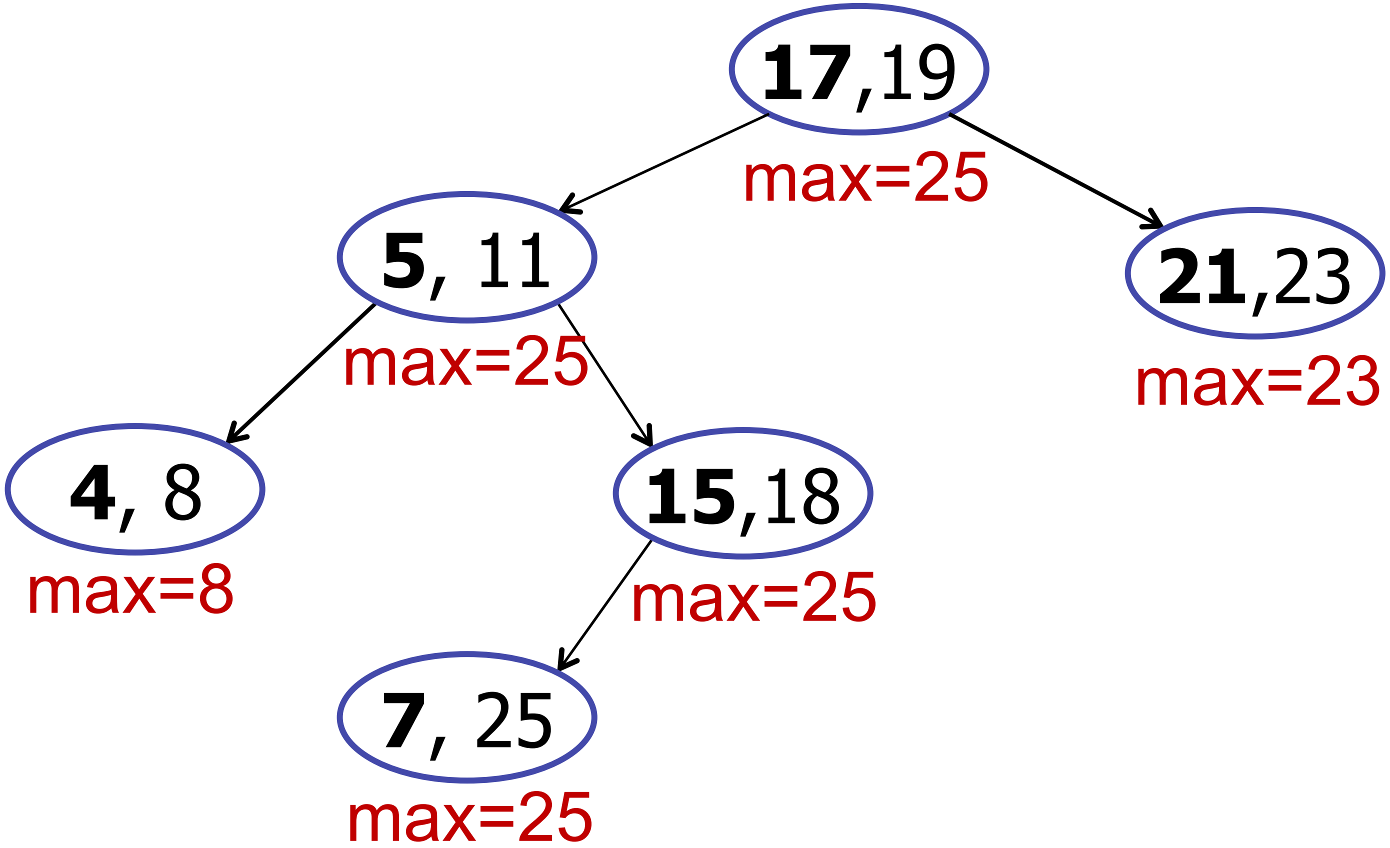
Interval Trees

Maintain MAX during rotations:



Interval Trees

Searching: *interval-search(22)*



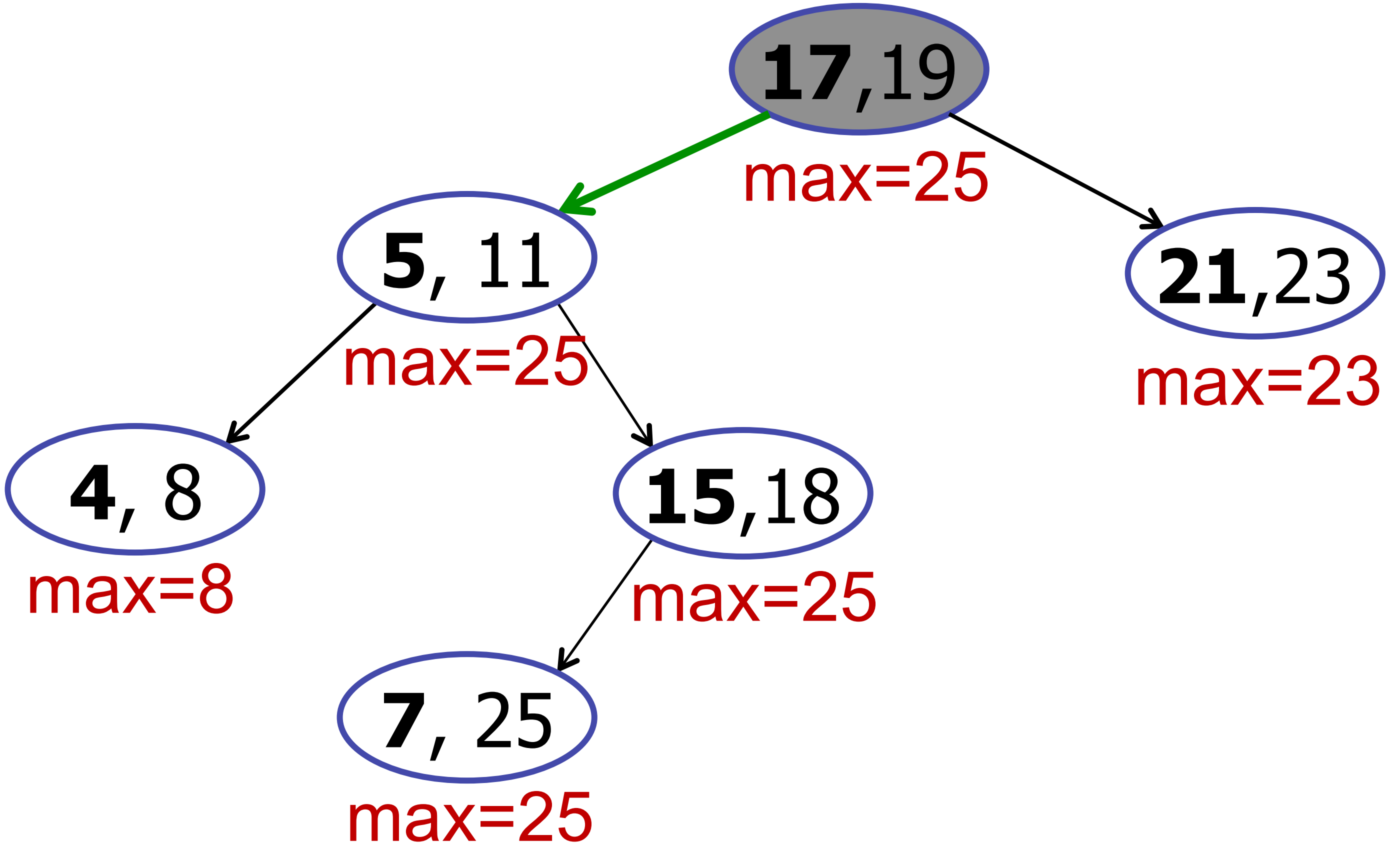
Dynamic Order Statistics

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;
```

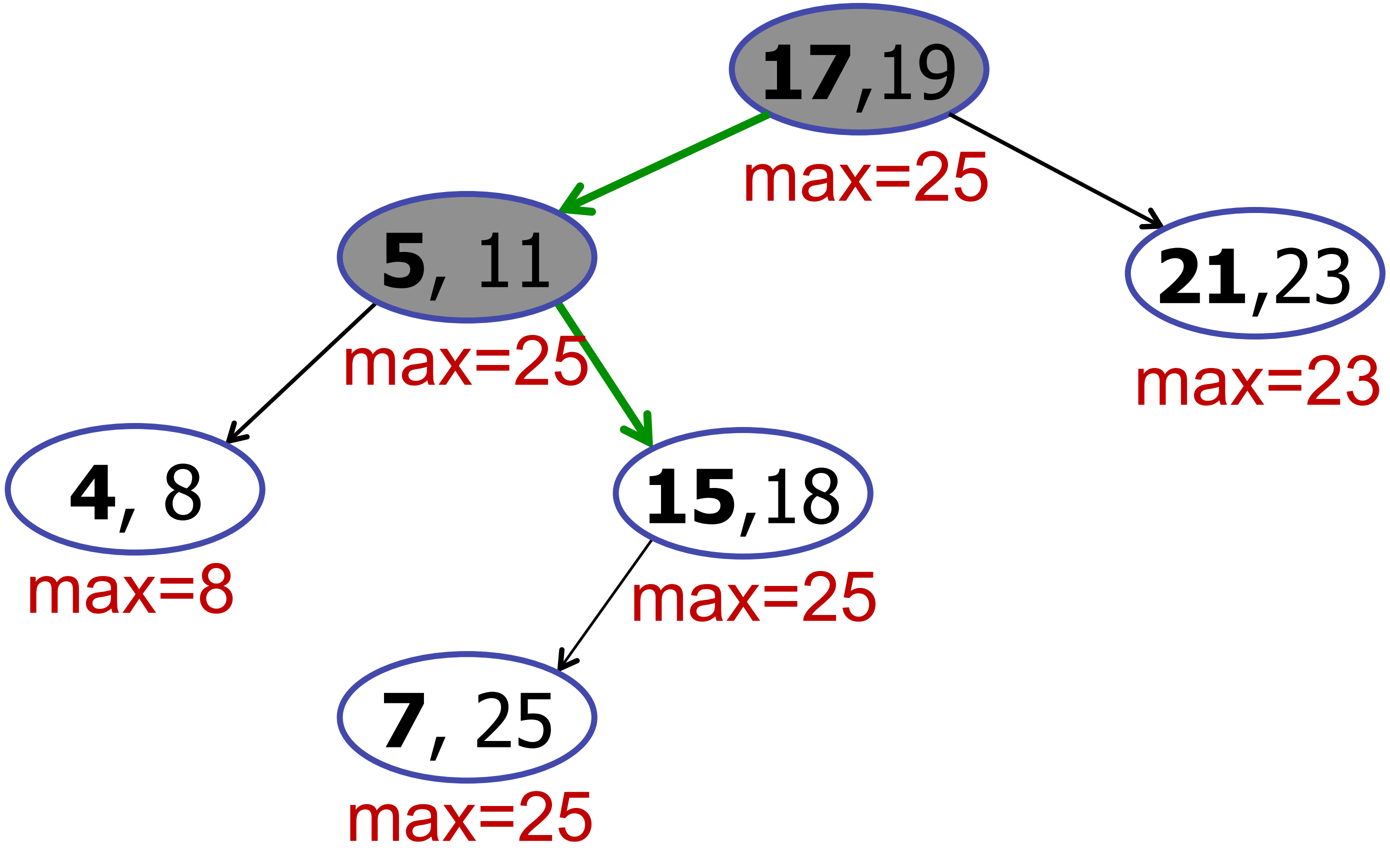
Interval Trees

Searching: *interval-search(22)*



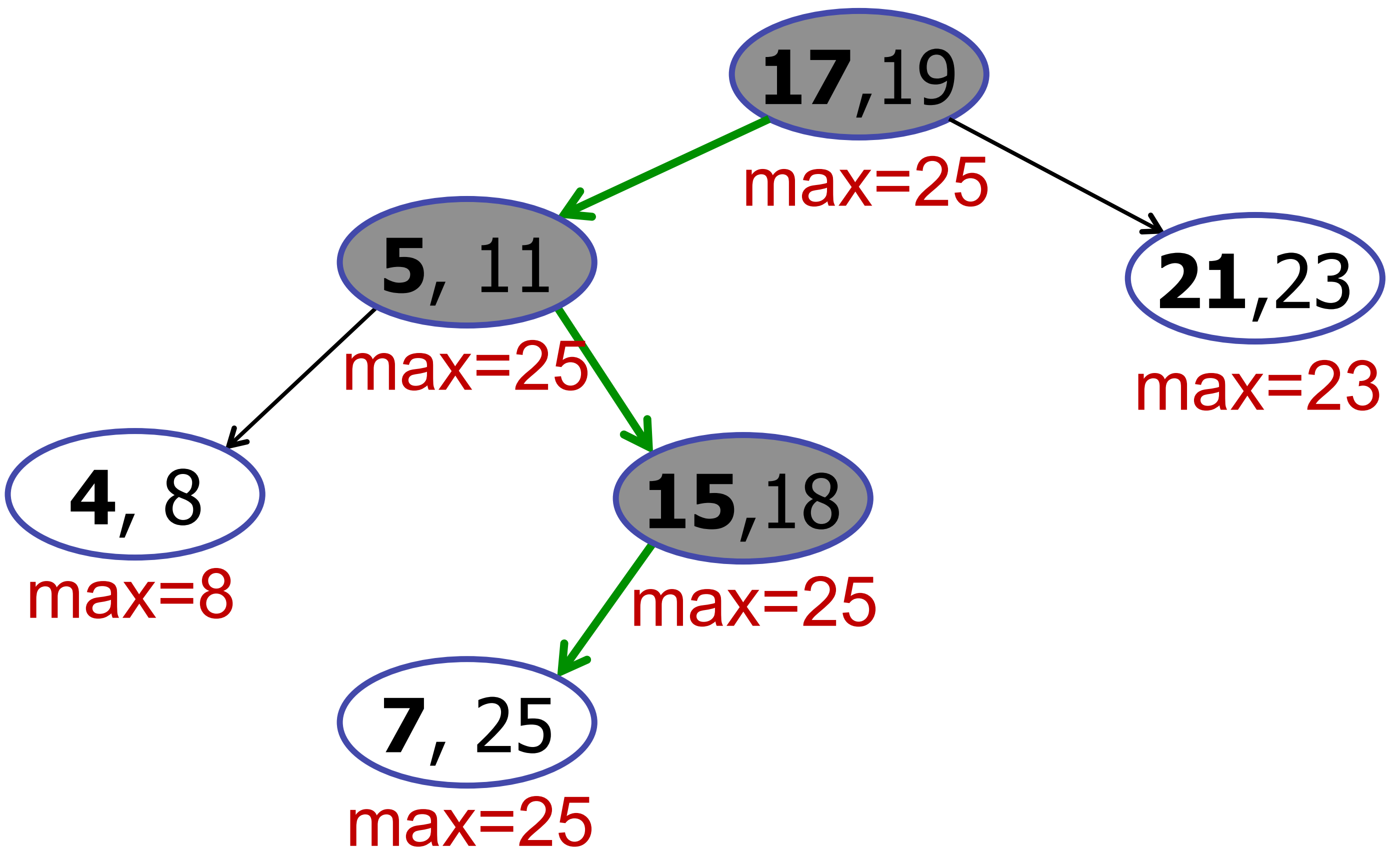
Interval Trees

Searching: *interval-search(22)*



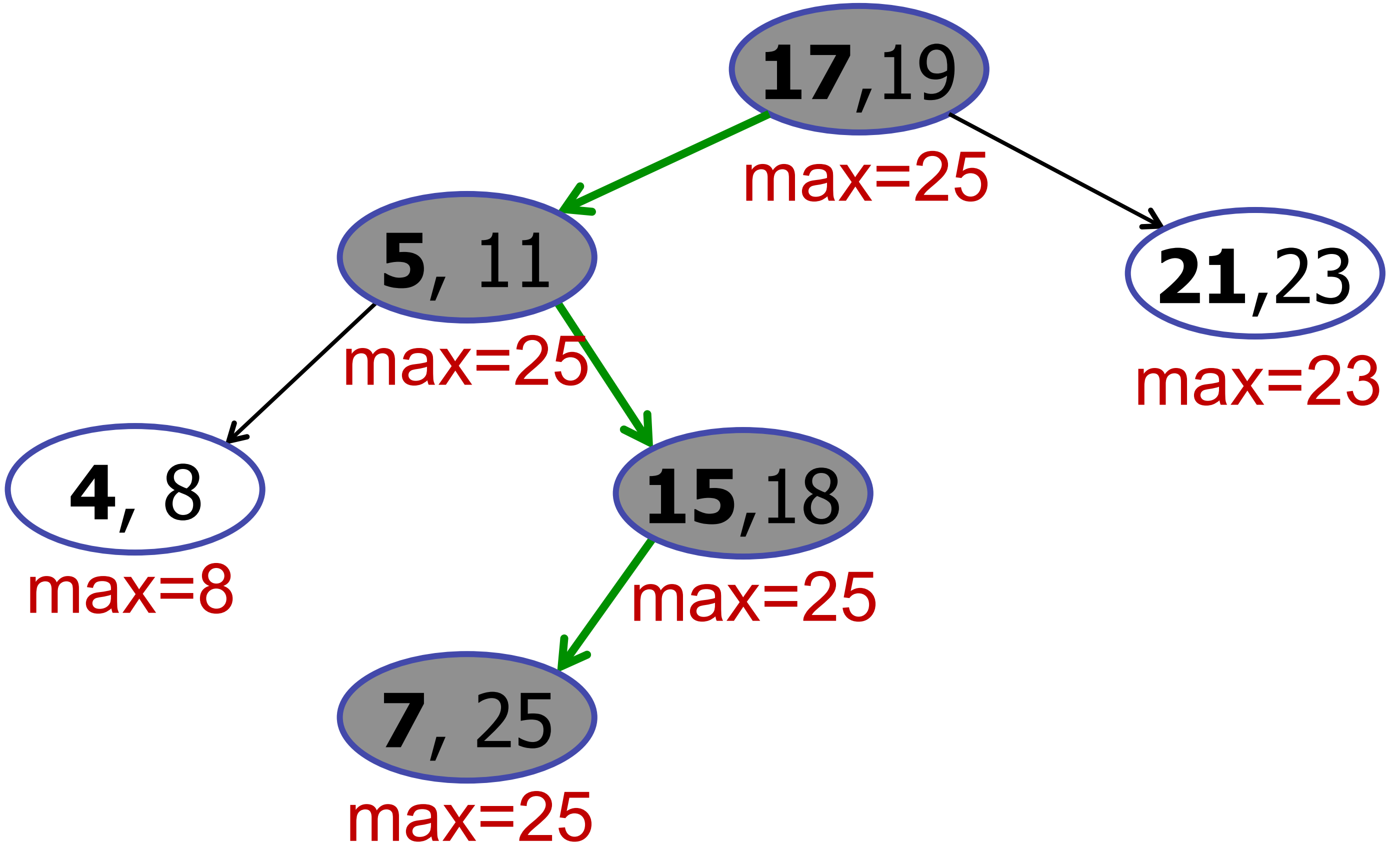
Interval Trees

Searching: *interval-search(22)*



Interval Trees

Searching: *interval-search(22)*



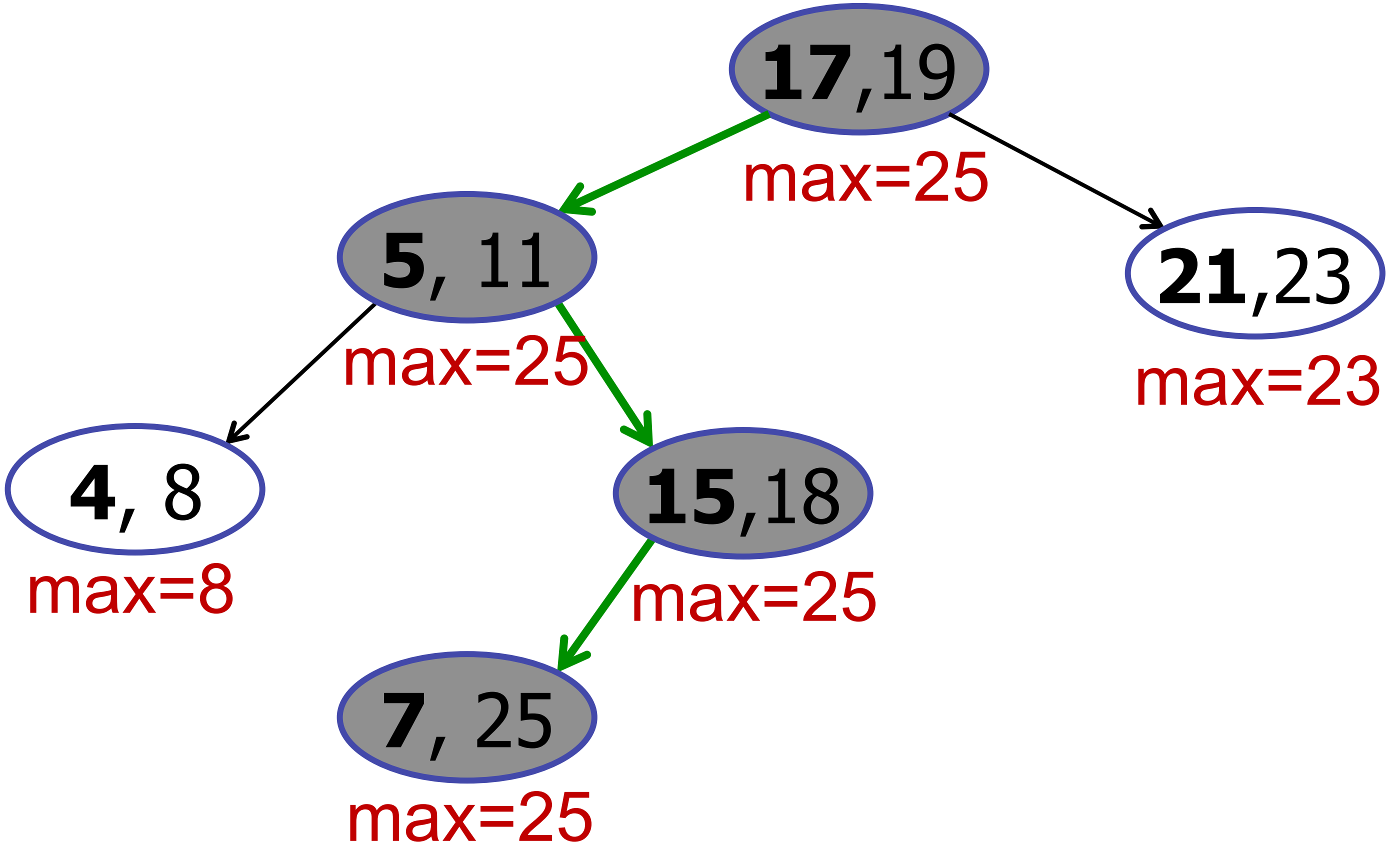
Dynamic Order Statistics

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;
```

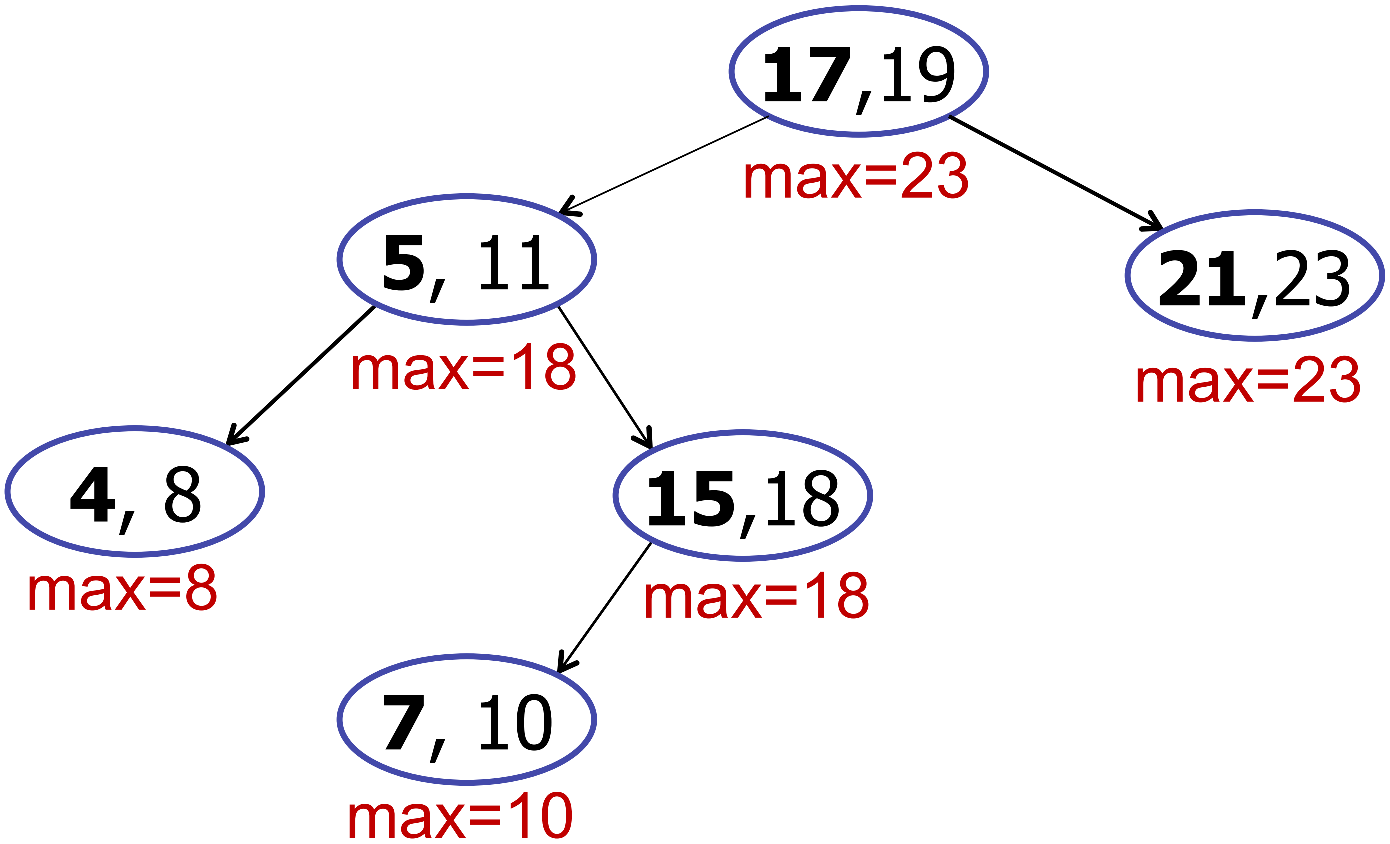
Interval Trees

Will any search find (21, 23)?



Interval Trees

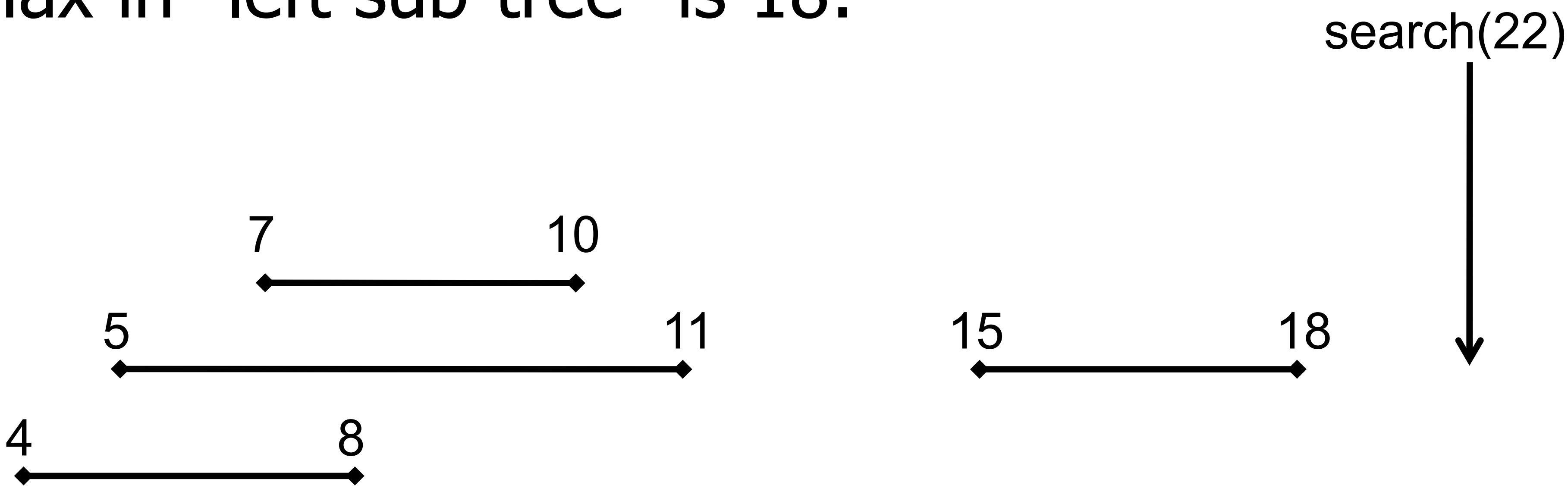
Searching: *interval-search(22)*



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

Interval Trees

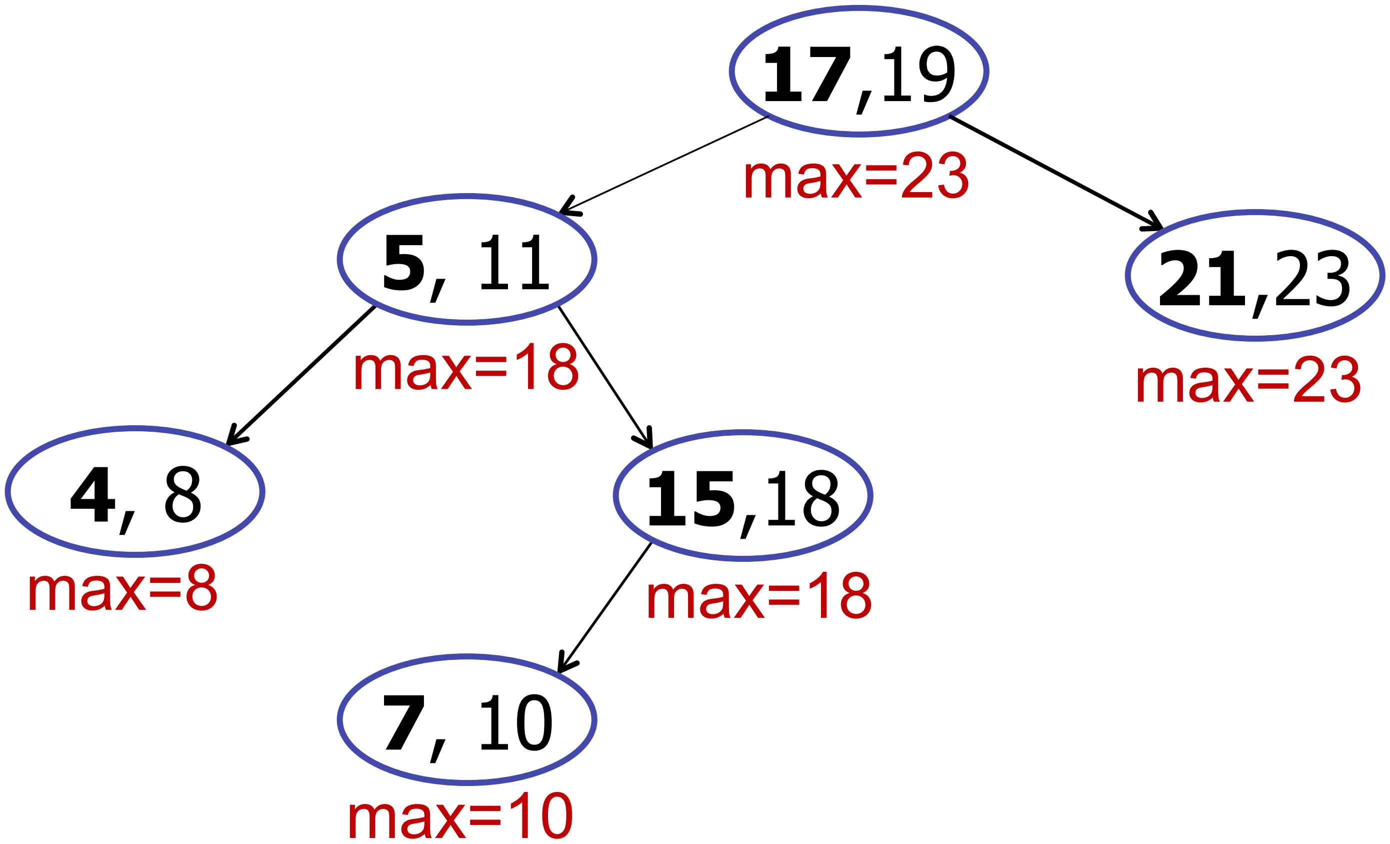
Max in "left sub-tree" is 18:



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

Interval Trees

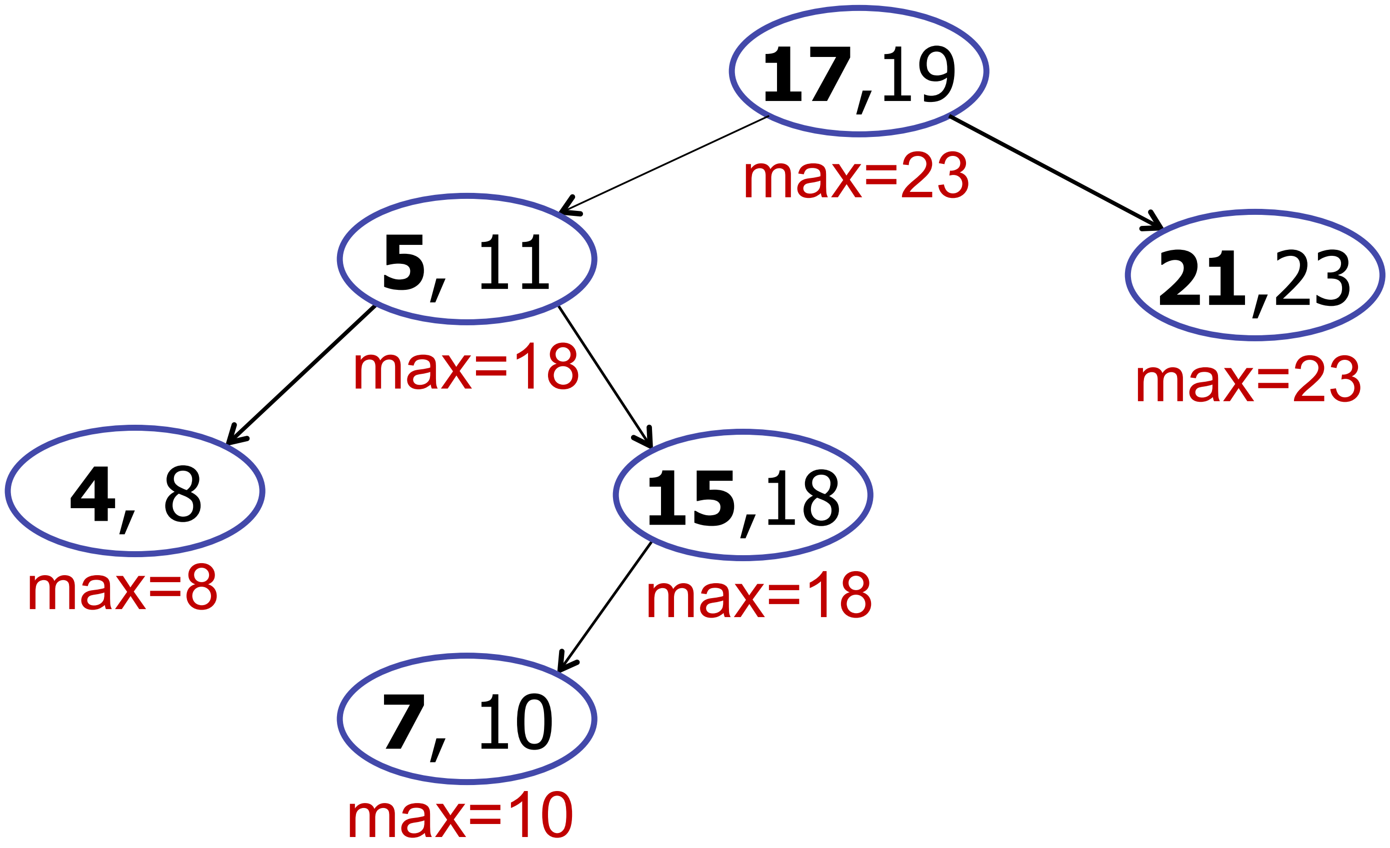
Searching: *interval-search(13)*



Claim: if search goes left and there is no overlap in the left subtree...

Interval Trees

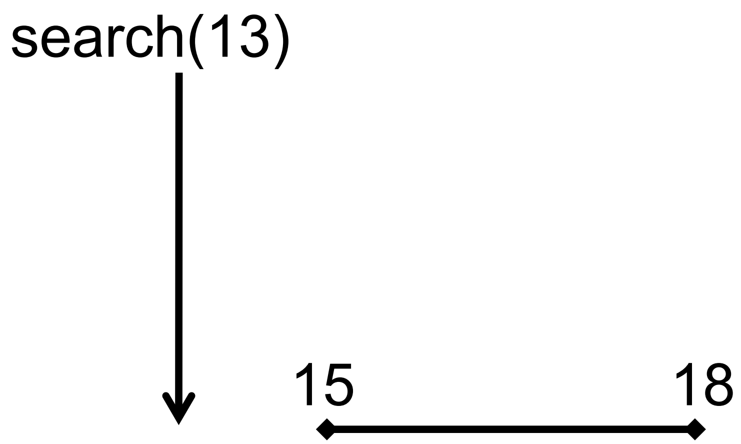
Searching: *interval-search(13)*



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

Interval Trees

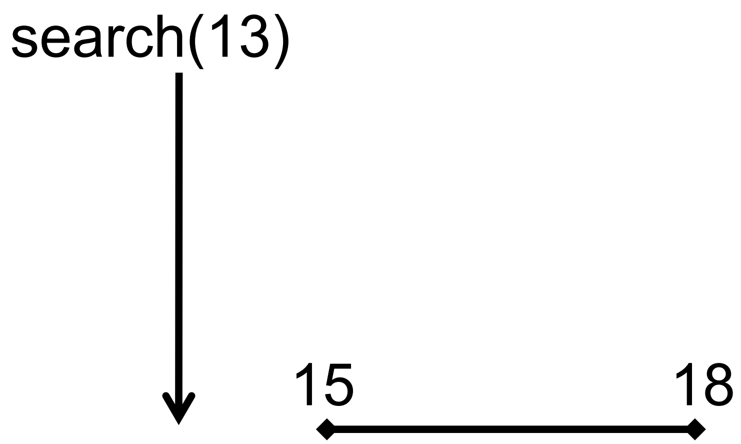
Max in "left sub-tree" is 18:



Go left: search(13) < 18

Interval Trees

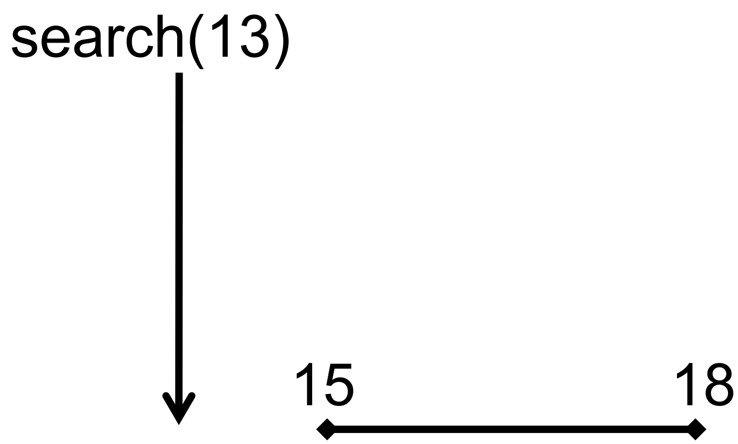
Max in “left sub-tree” is 18:



Go left: search(13) < 15 < 18

Interval Trees

Max in "left sub-tree" is 18:

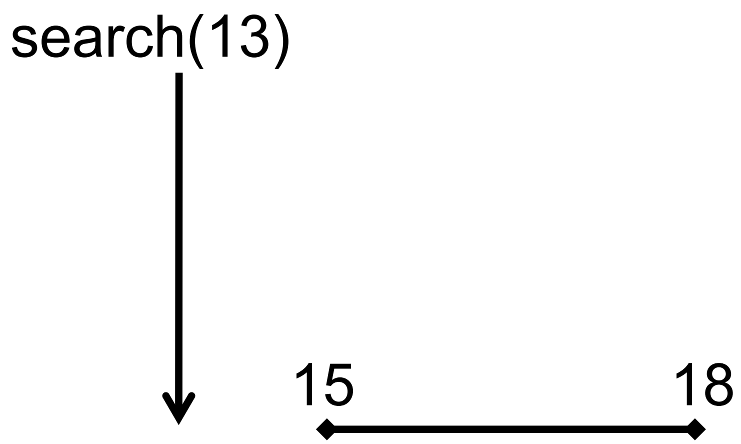


Go left: search(13) < 15 < 18

Tree sorted by left endpoint.

Interval Trees

Max in "left sub-tree" is 18:



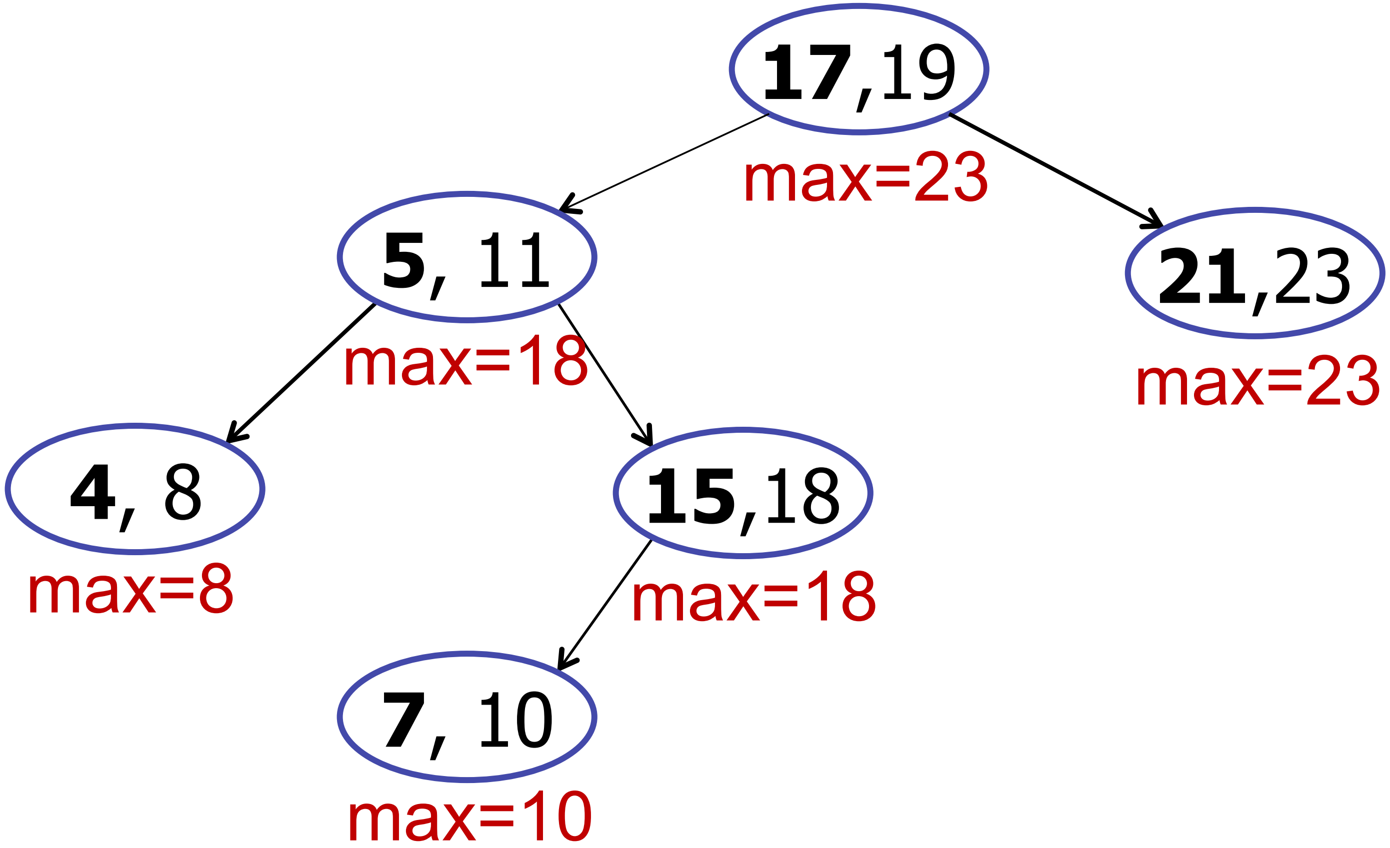
Go left: search(13) < 15 < 18

Tree sorted by left endpoint.

search(13) < every interval in right subtree

Interval Trees

Searching: *interval-search(13)*



Claim: if search goes left and no overlap, then search < every interval in right sub-tree.

Interval Trees

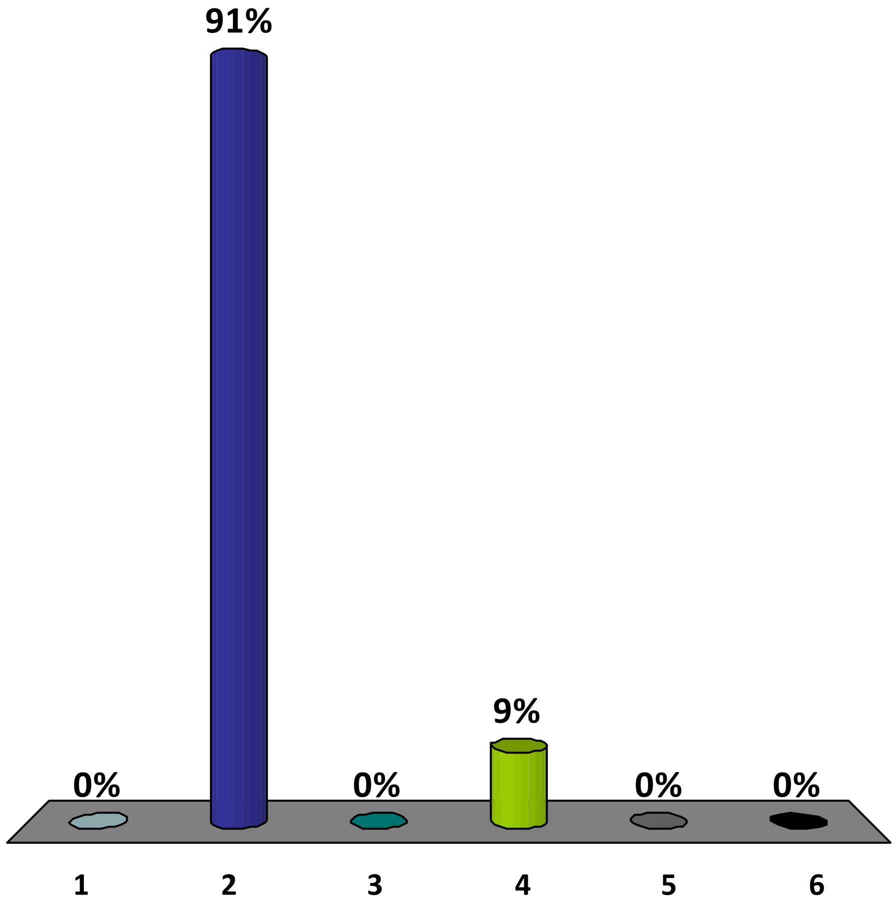
If search goes right, then no overlap in left subtree.

If search goes left, and if there is no overlap in left subtree, then there is no overlap in right subtree either.

Conclusion: search finds an overlapping interval.

The running time of interval-search is:

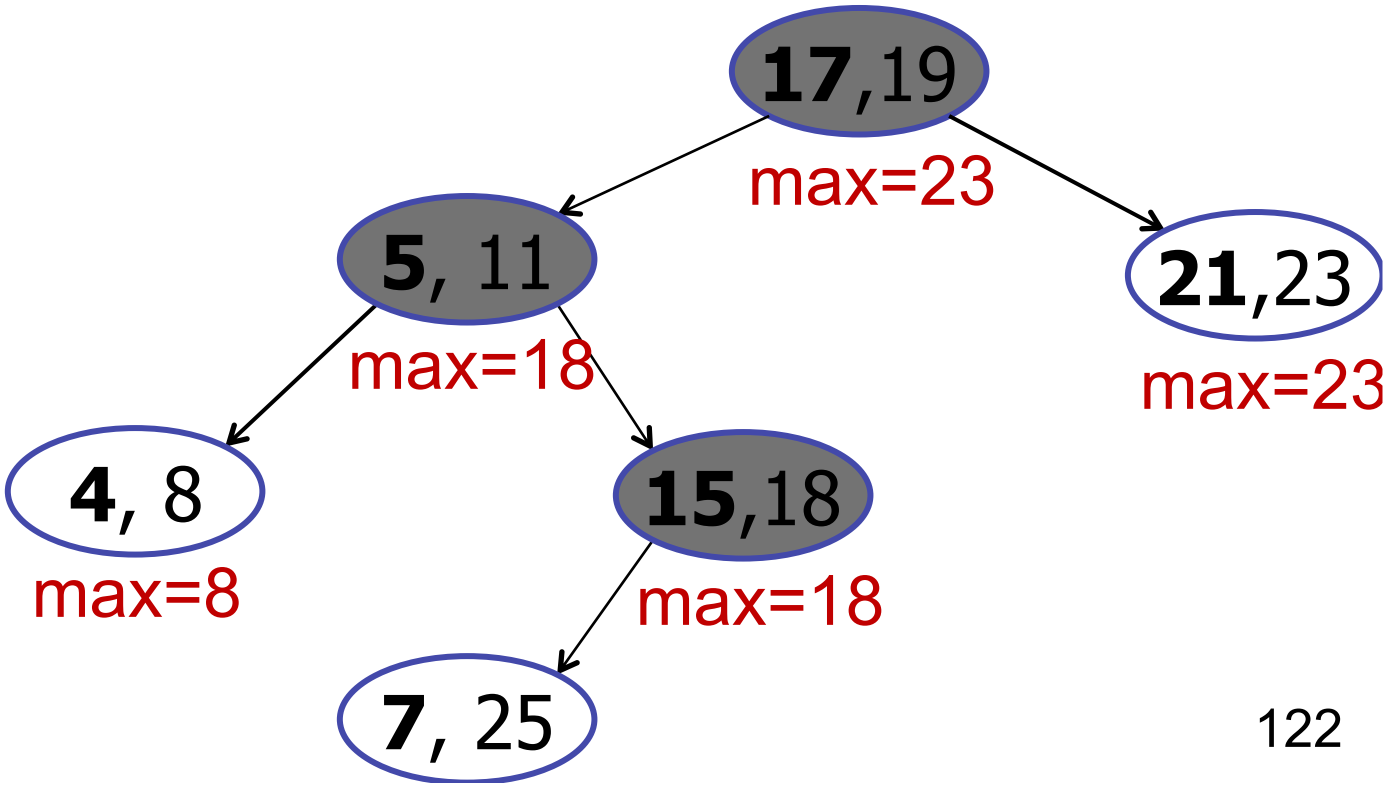
1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)
5. O(n^2)
6. Can’t say.



Interval Trees

Extensions

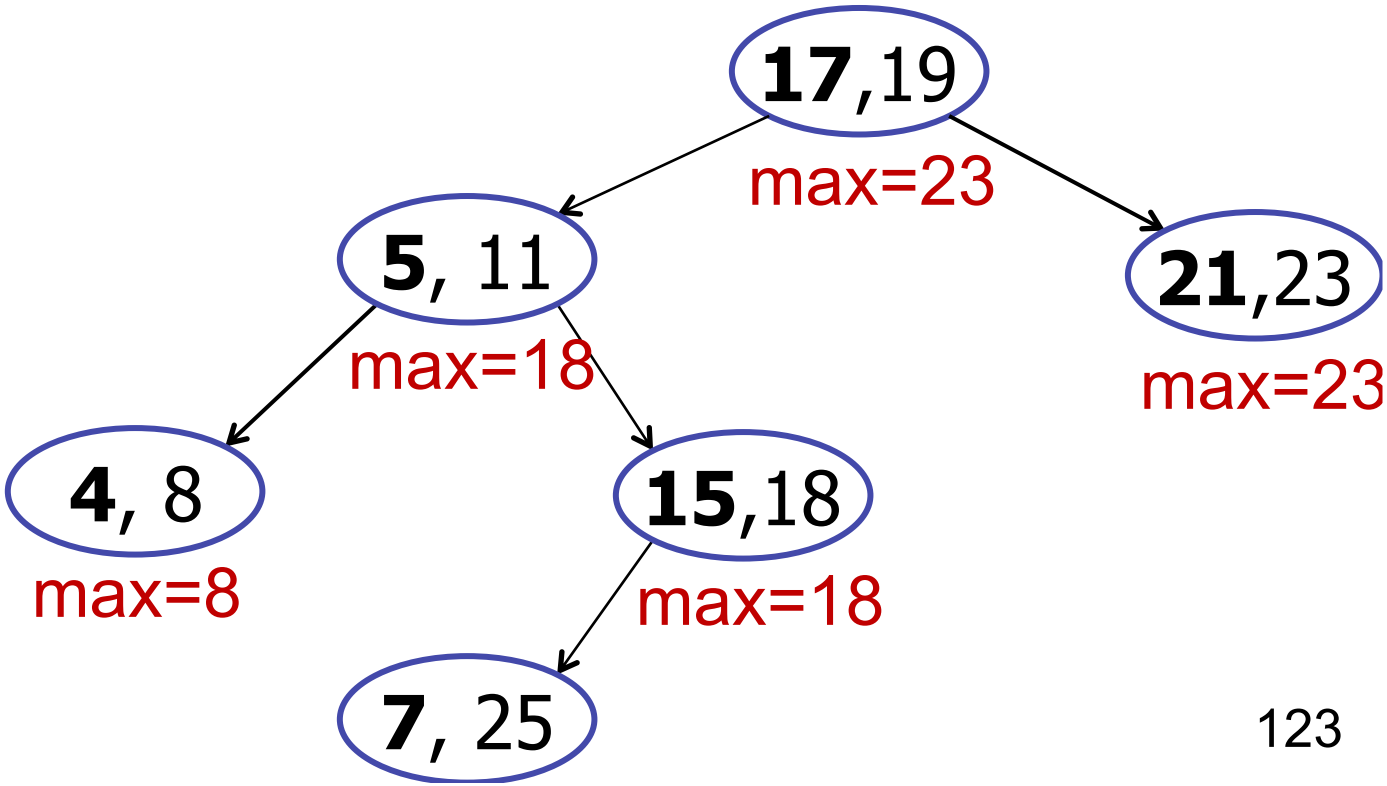
- What if you want to search for two intervals that overlap?
- Eg: **search(14,16)**



Interval Trees

Extensions

- Cost for listing all intervals that overlap with point?
- E.g.: search(22) returns:
  - (7,25)
  - (21,23)



Interval Trees

Extensions

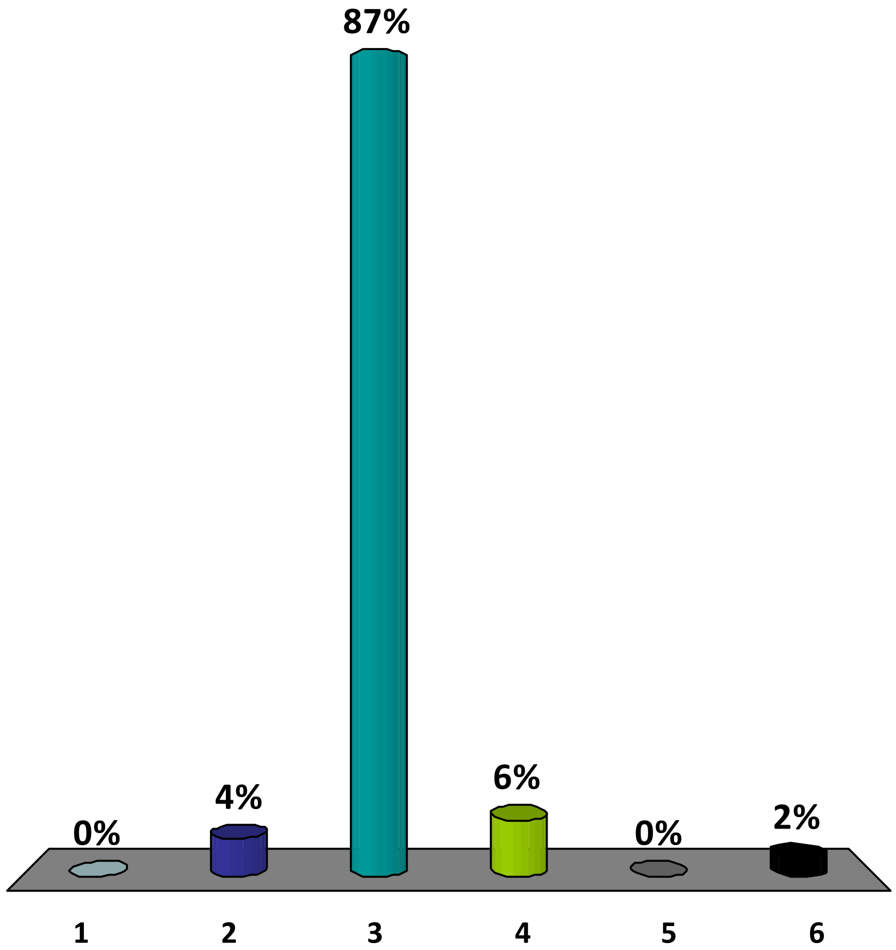
- Cost for listing all intervals that overlap with point?
- All-Overlaps Algorithm:

  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)



Interval Trees

Extensions

- Cost for listing all intervals that overlap 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

Three examples of augmenting BSTs

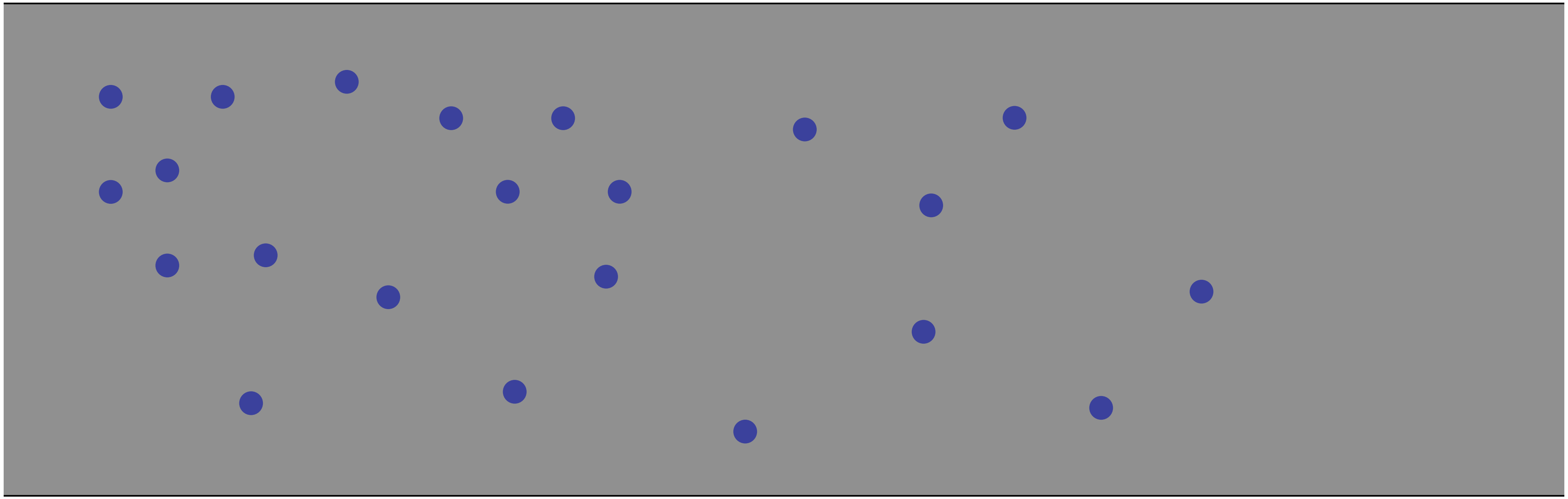
1. Order Statistics

2. Intervals

3. Orthogonal Range Searching

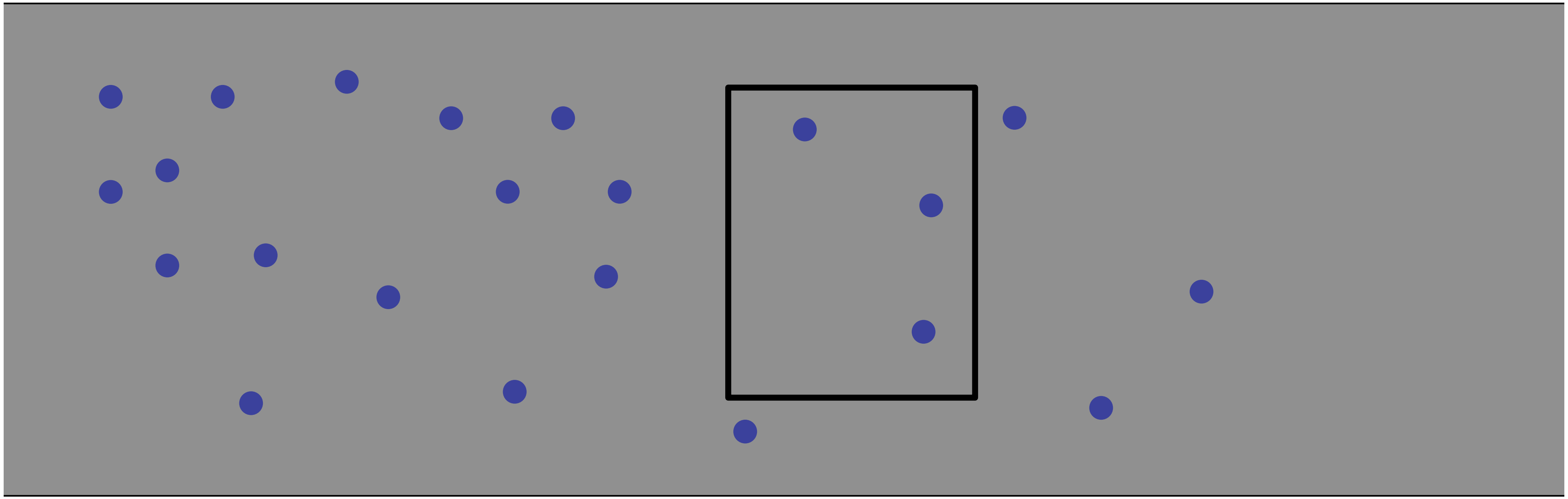
Orthogonal Range Searching

Input: n points in a 2d plane



Orthogonal Range Searching

Input: n points in a 2d plane

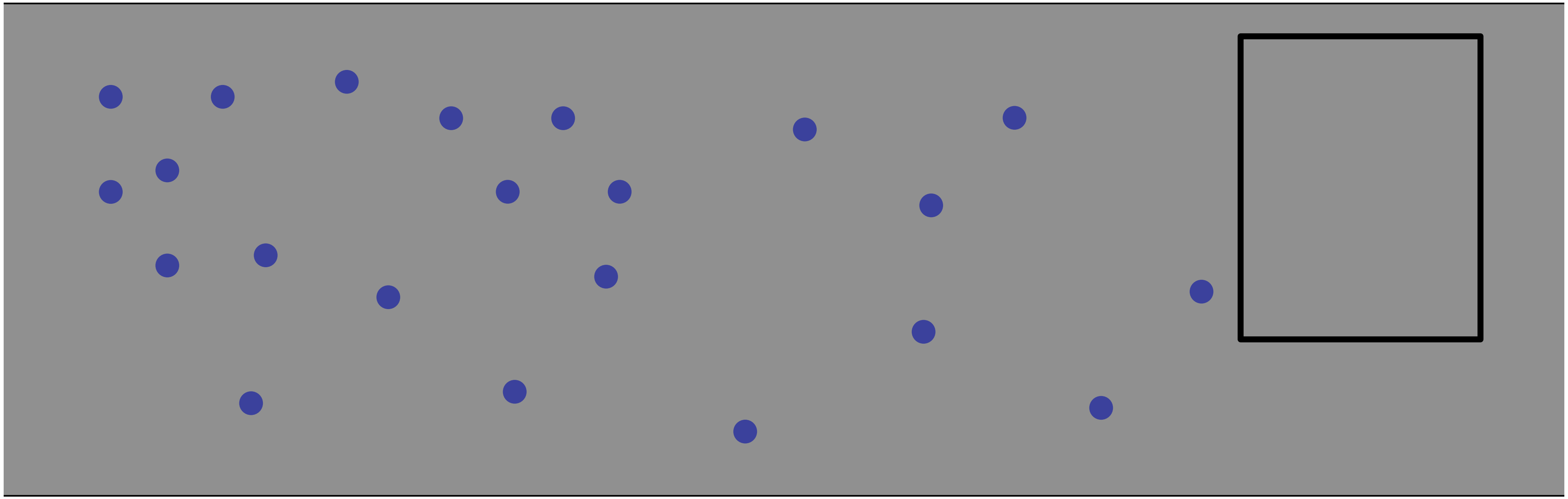


Query: Box

- Contains at least one point?
- How many?

Orthogonal Range Searching

Input: n points in a 2d plane

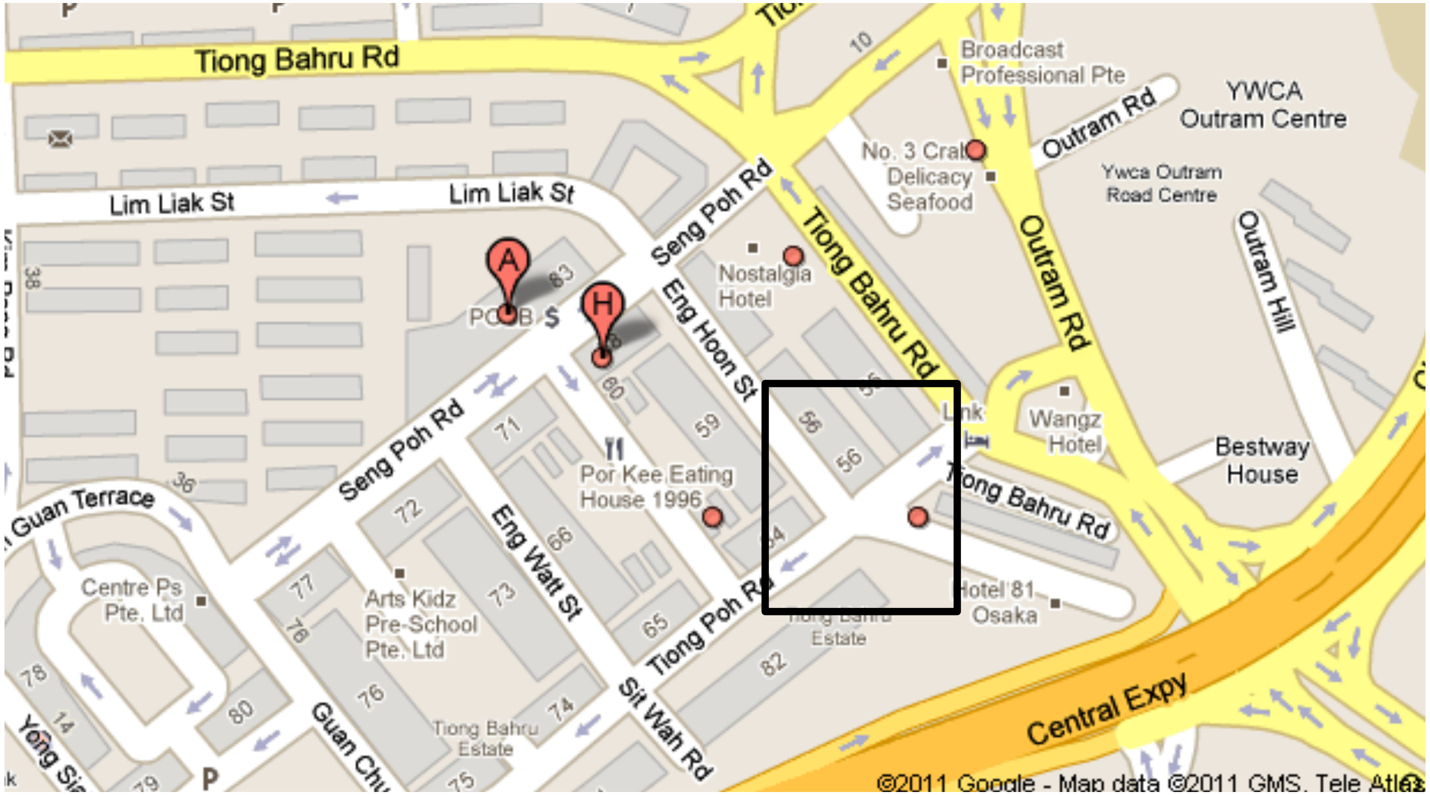


Query: Box

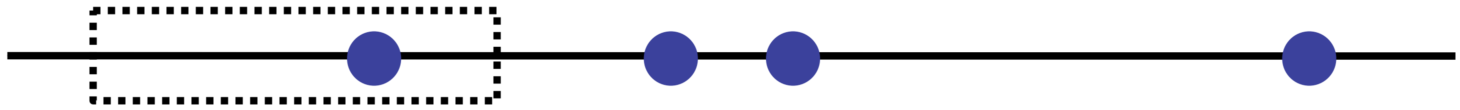
- Contains at least one point?
- How many?

Practical Example

Are there any good restaurants within one block of me?



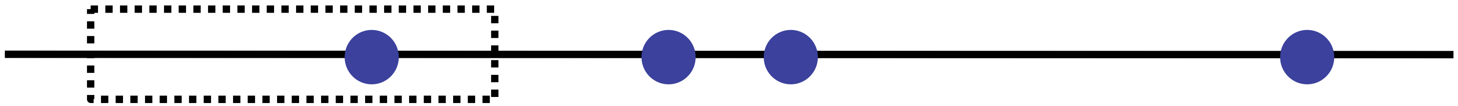
One Dimension



One Dimension

Range Queries

- Important in databases
- "Find me everyone between ages 22 and 27."



One Dimension

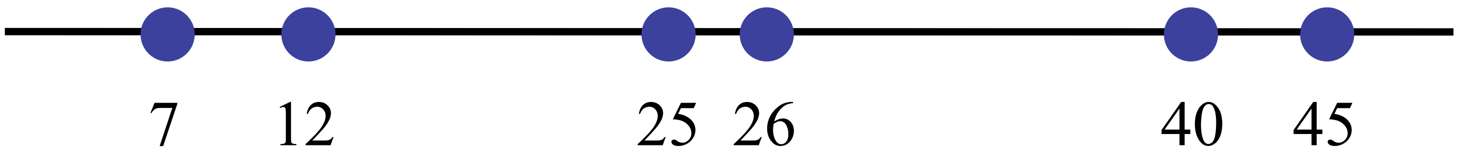
Strategy:

1. Use a binary search tree.

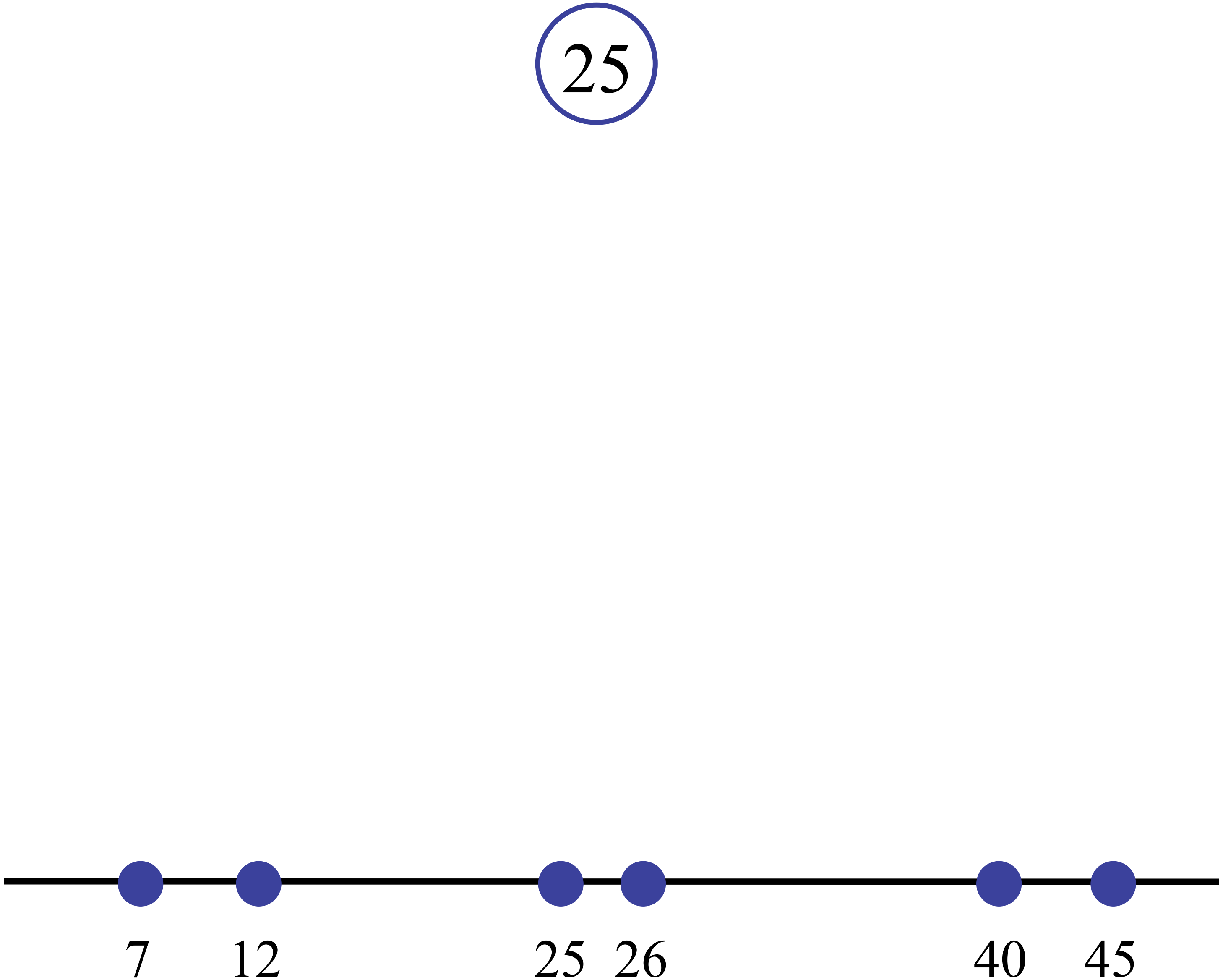
2. Store all points in the leaves of the tree.
   (Internal nodes store only copies.)

3. Each internal node *v* stores the MAX of any leaf in the left sub-tree.

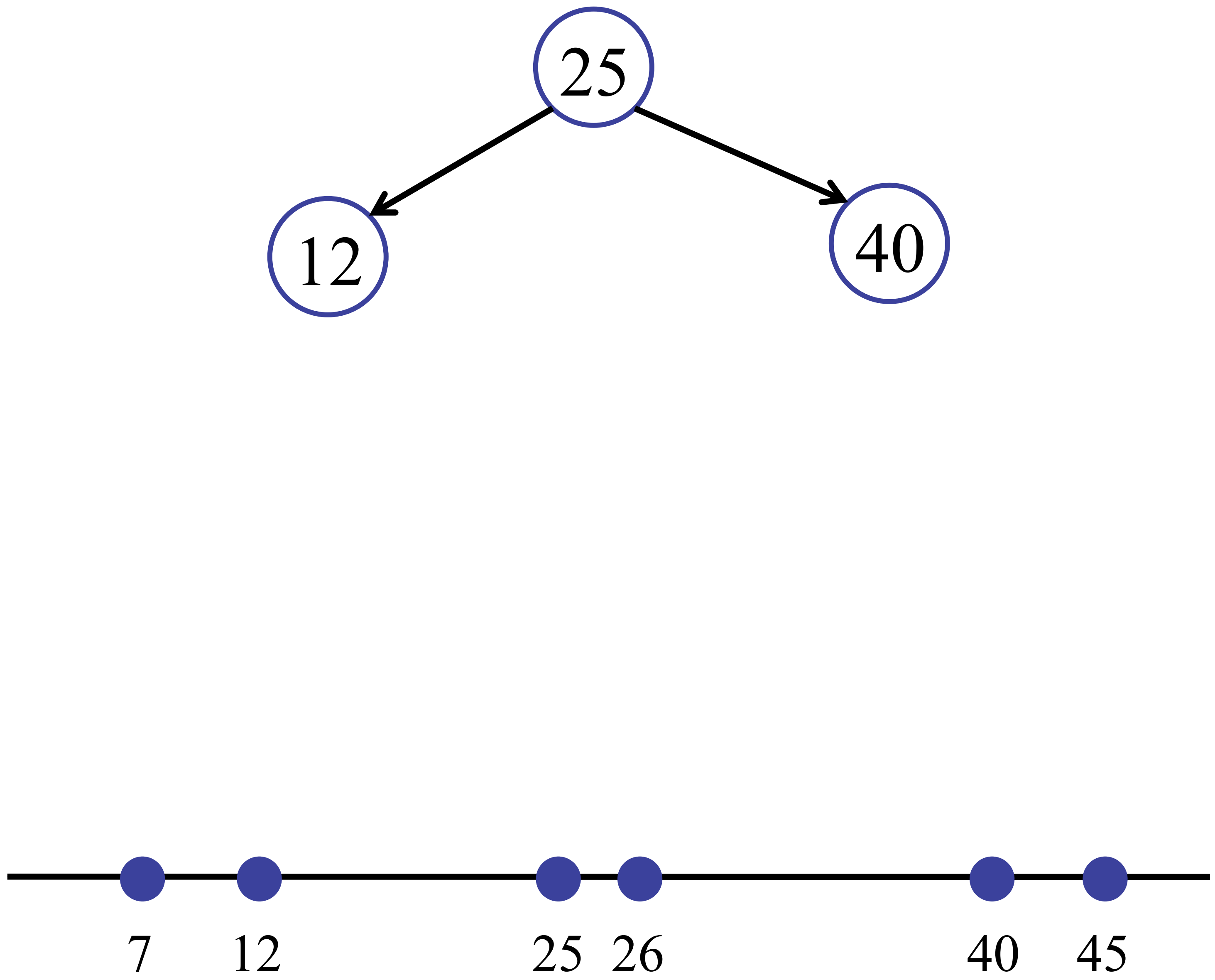
Example



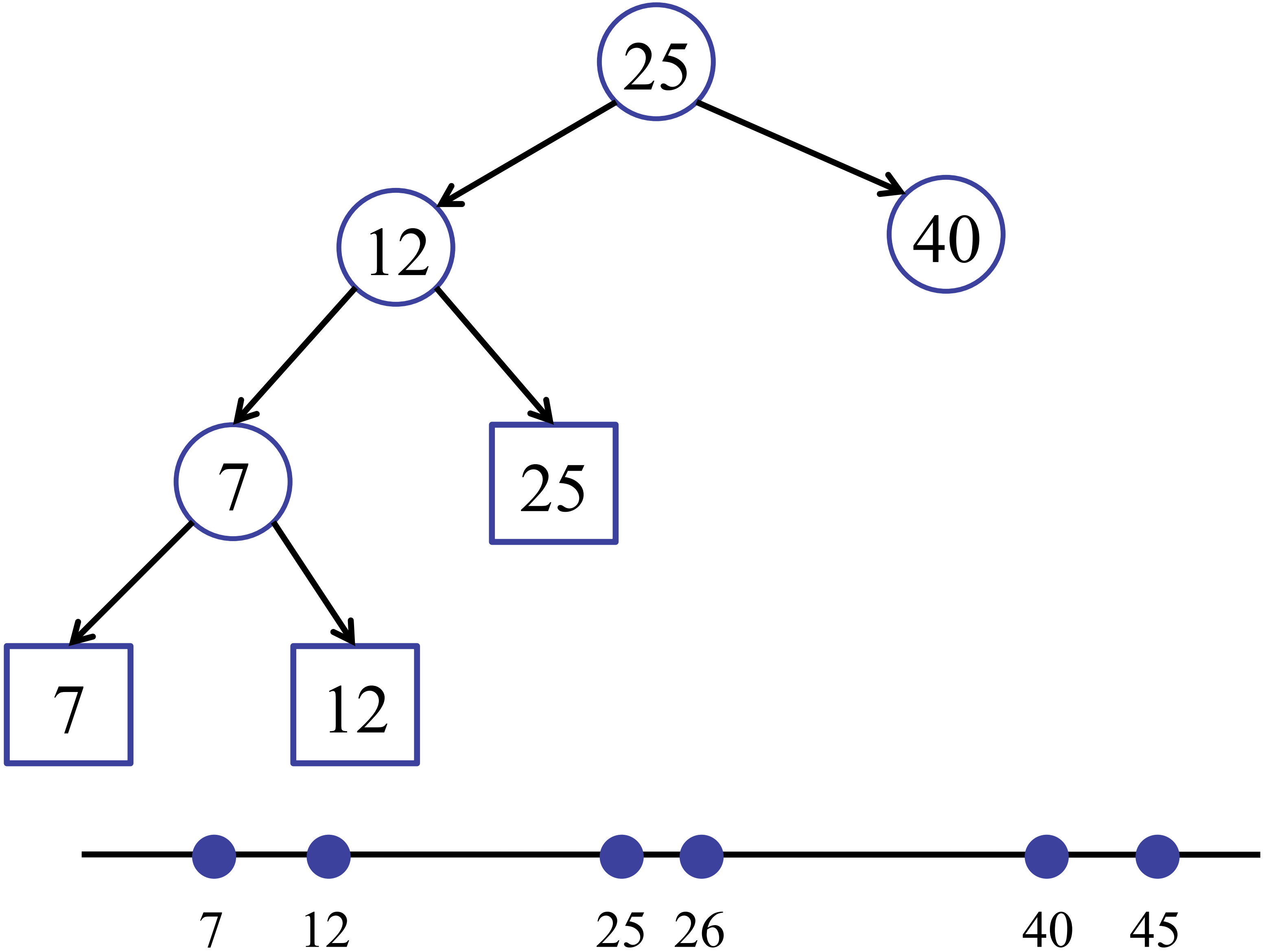
Example



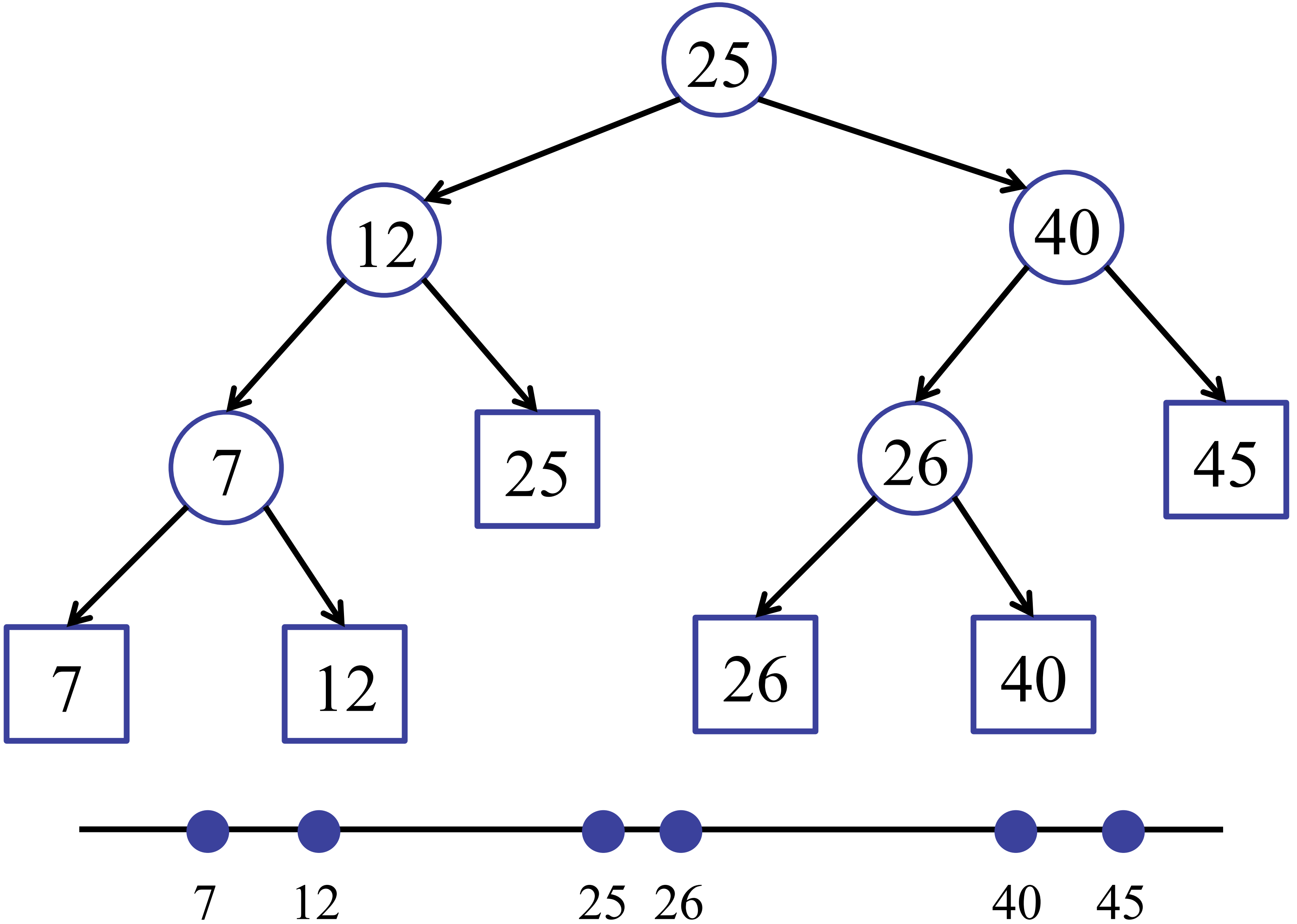
Example



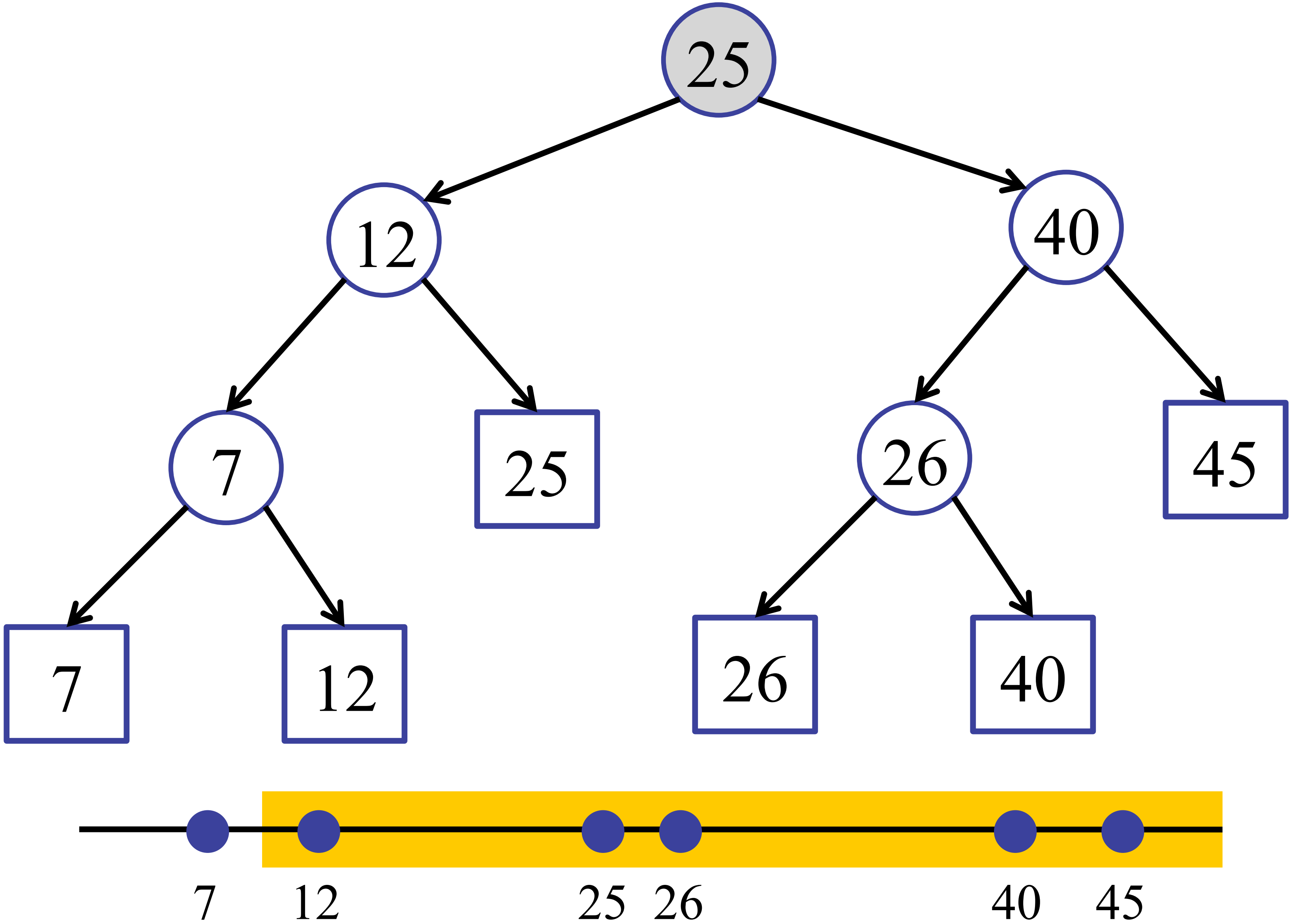
Example



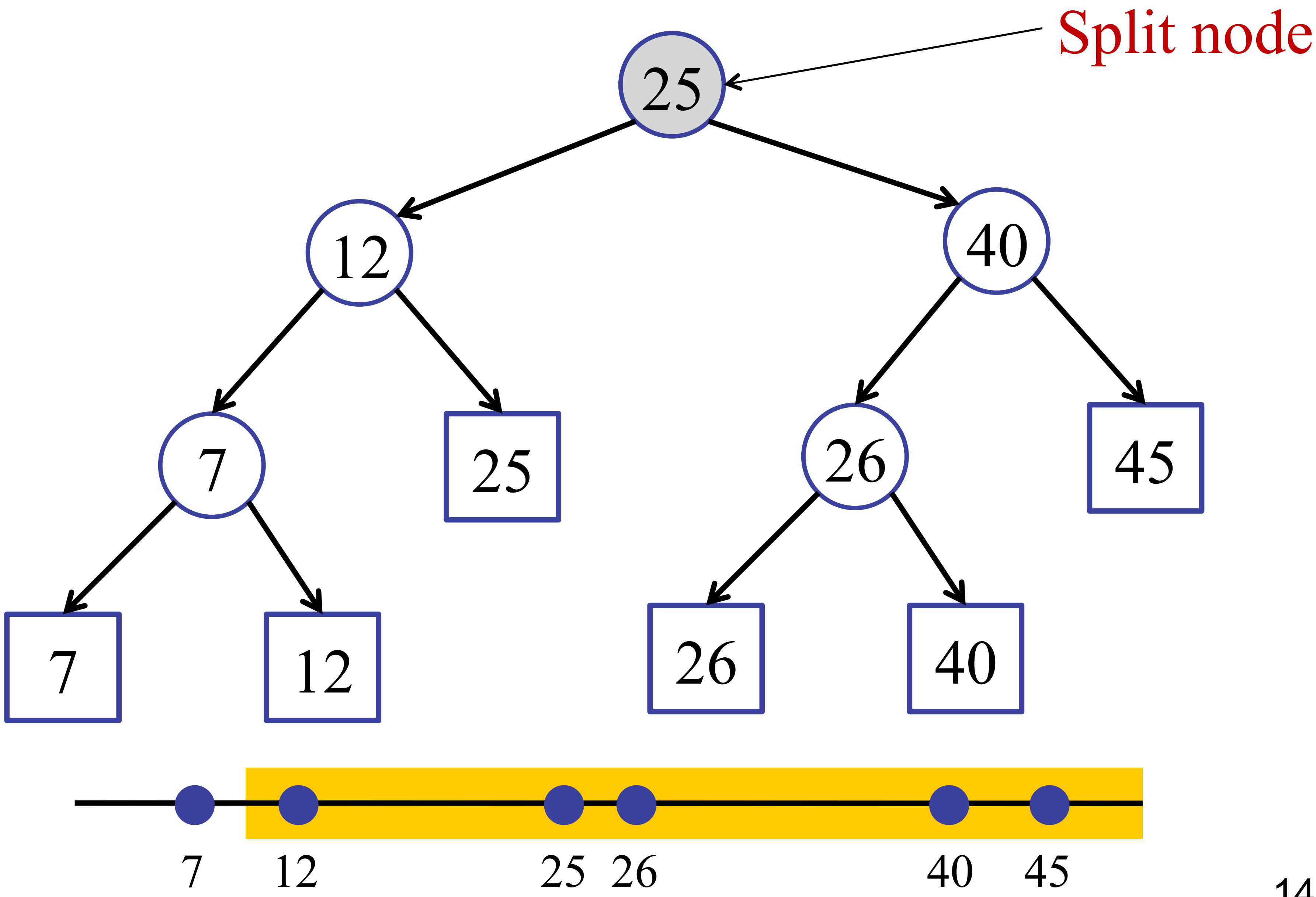
Note: BST Property



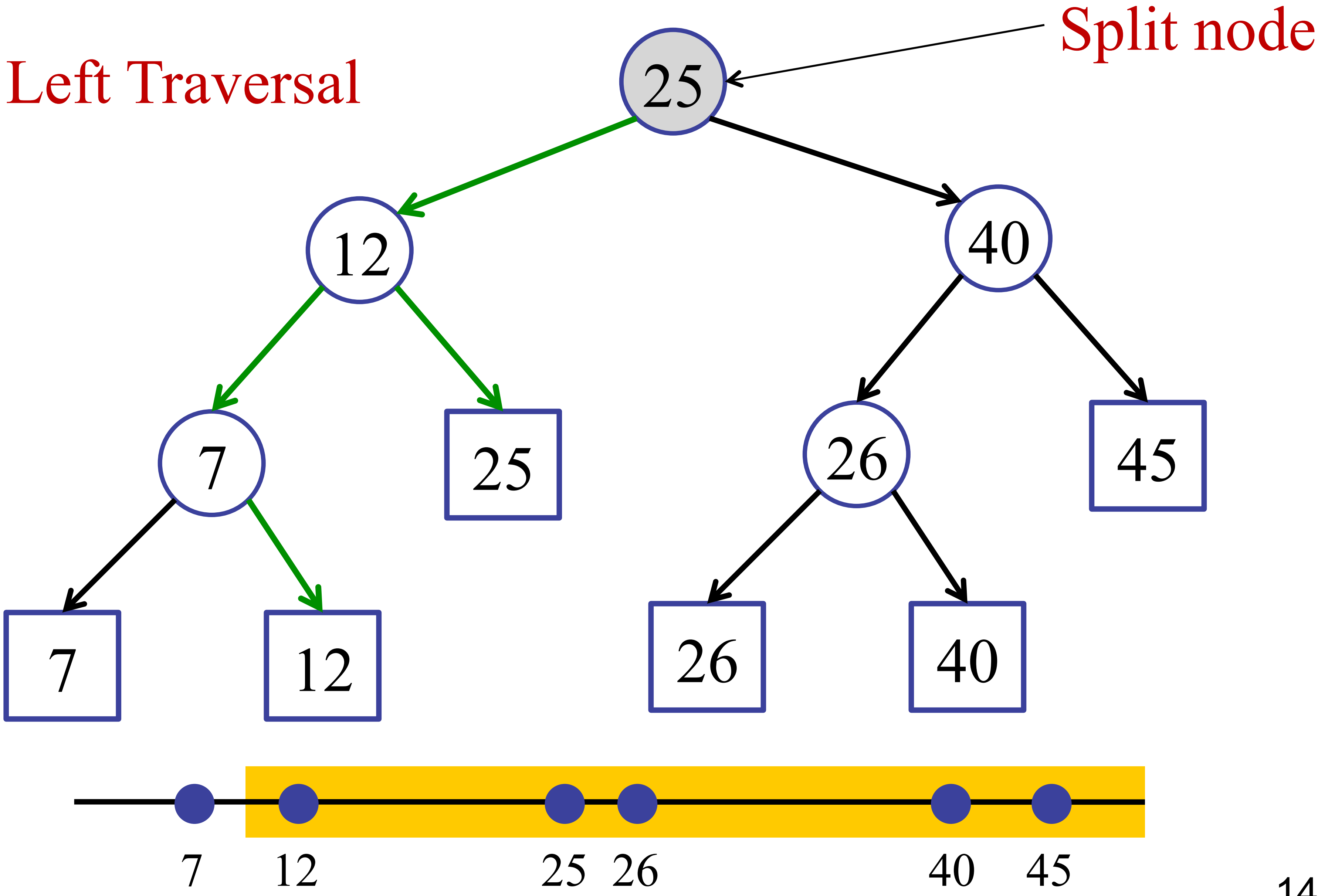
Example: query(10, 50)



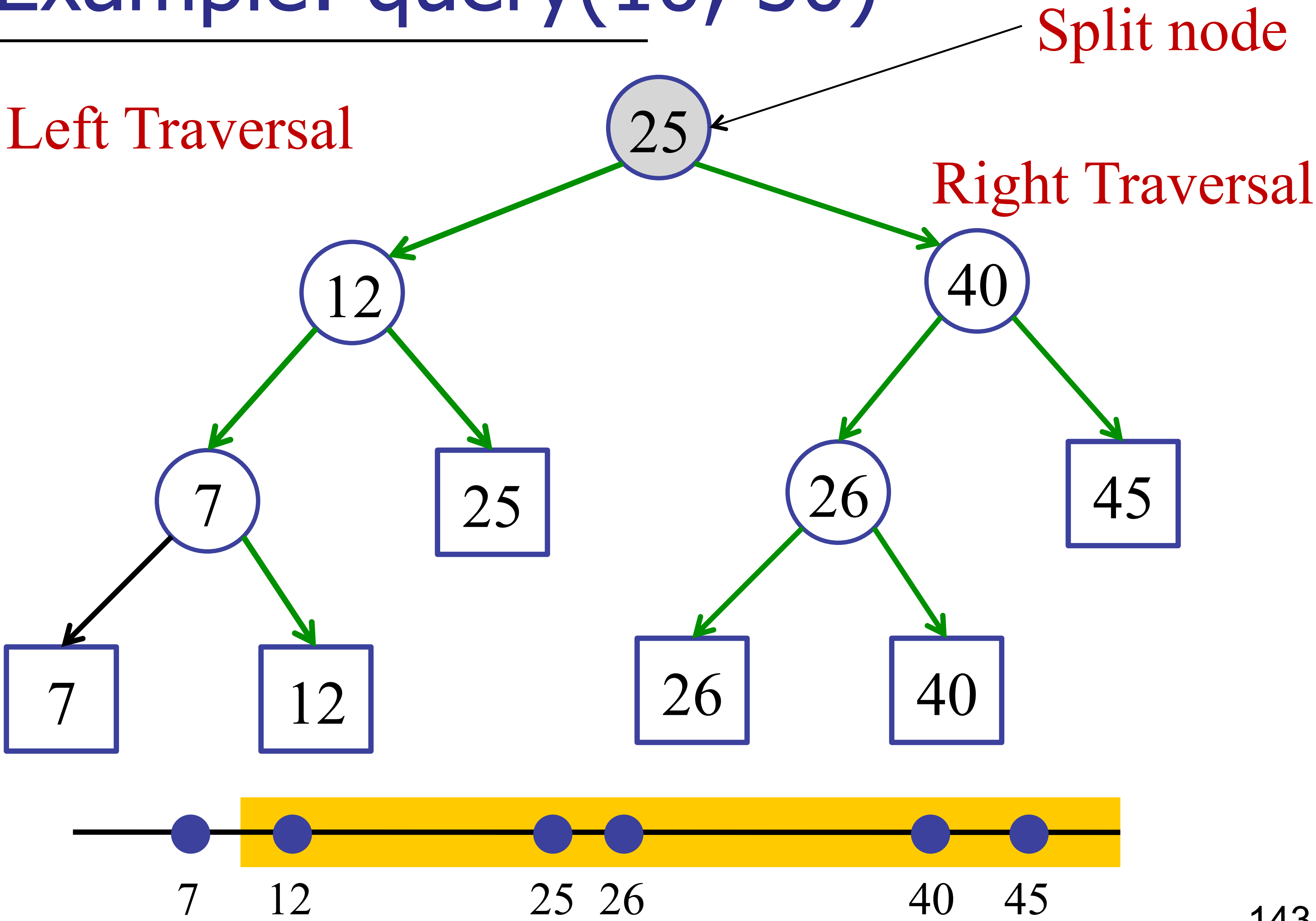
Example: query(10, 50)



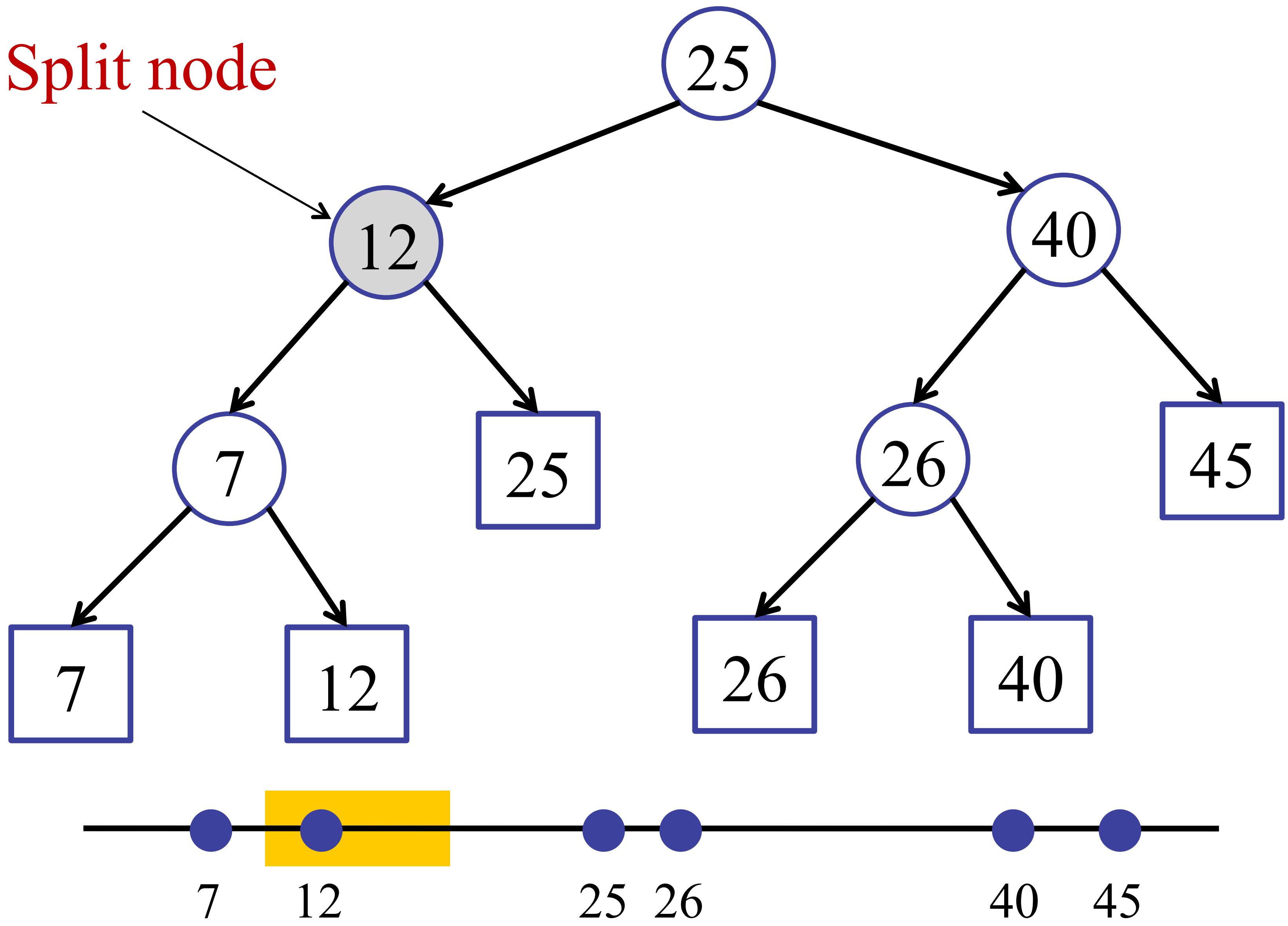
Example: query(10, 50)



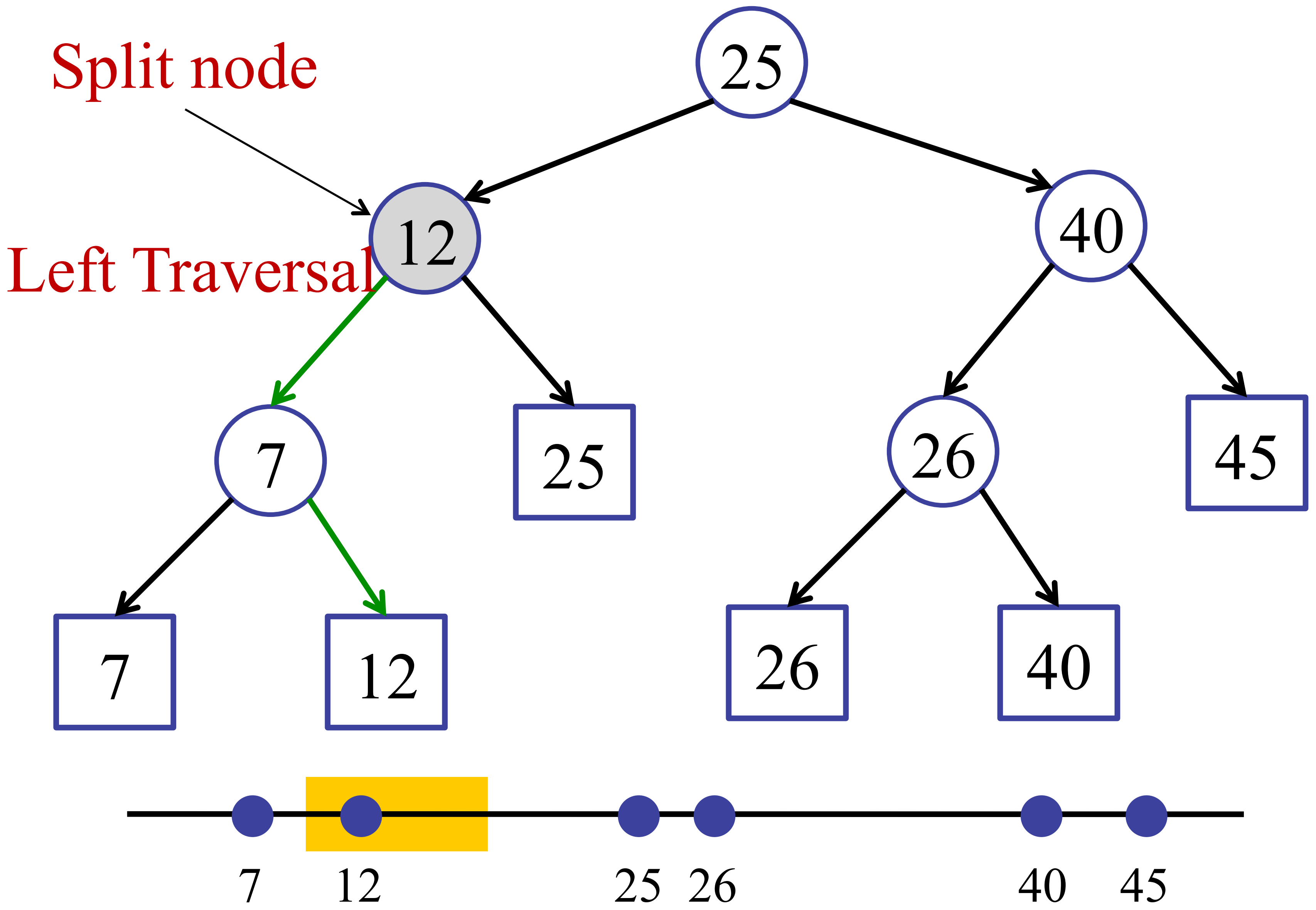
Example: query(10, 50)



Example: query(8, 20)



Example: query(8, 20)



One Dimensional Range Queries

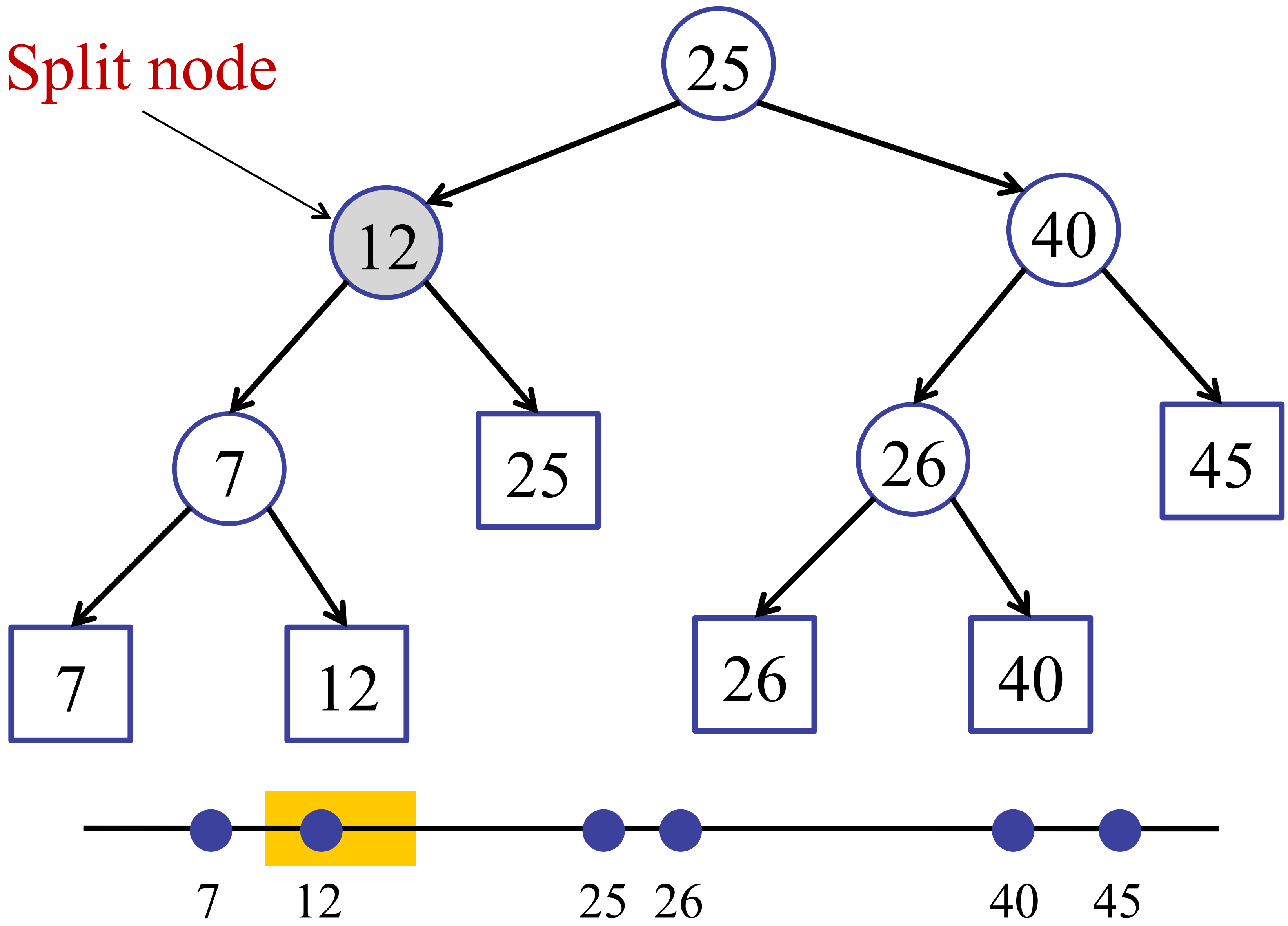
Algorithm:

- Find “split” node.
- Do left traversal.
- Do right traversal.

One Dimensional Range Queries

```
FindSplit(low, high)
        v = root;
        done = false;
        while !done {
                if (high <= v.key) then v=v.left;
                else if (low > v.key) then v=v.right;
                else (done = true);
        }
        return v;
```

Example: query(8, 20)



One Dimensional Range Queries

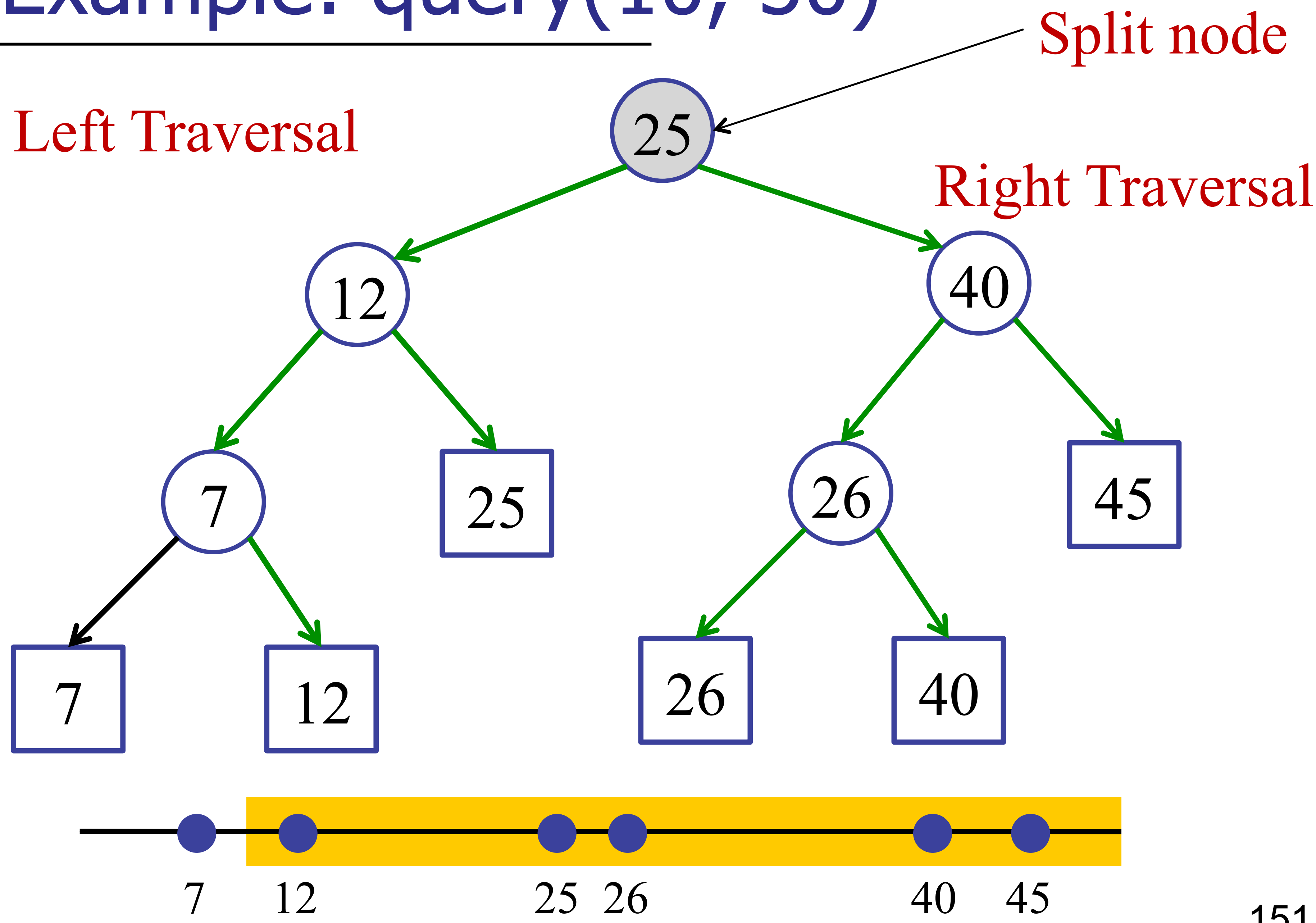
Algorithm:

- v = FindSplit(low, high);
- LeftTraversal(v, low, high);
- RightTraversal(v, low, high);

One Dimensional Range Queries

```
LeftTraversal(v, low, high)
    if (low <= v.key) {
        all-leaf-traversal(v.right);
        LeftTraversal(v.left, low, high);
    }
    else {
        LeftTraversal(v.right, low, high);
    }
}
```

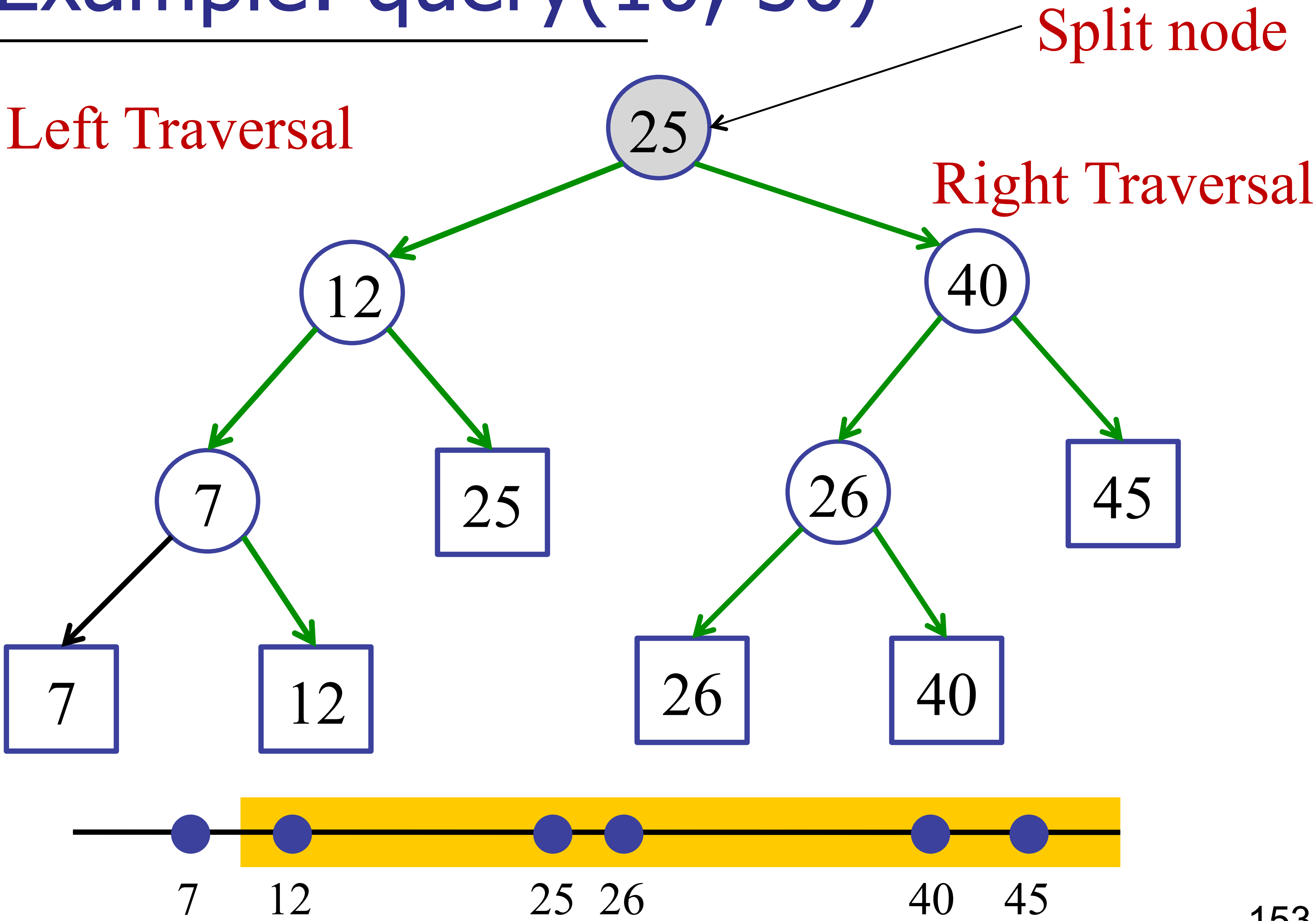
Example: query(10, 50)



One Dimensional Range Queries

```
RightTraversal(v, low, high)
    if (v.key <= high) {
        all-leaf-traverasal(v.left);
        RightTraversal(v.right, low, high);
    }
    else {
        RightTraversal(v.left, low, high);
    }
}
```

Example: query(10, 50)



Analysis

Query time:

- Finding split node: O(log n)
- Left Traversal:

  At every step, we either:

  1. Output all right sub-tree and recurse left.
  2. Recurse right.
- Right Traversal:

  At every step, we either:

  1. Output all left sub-tree and recurse right.
  2. Recurse left.

Analysis

– Left Traversal:

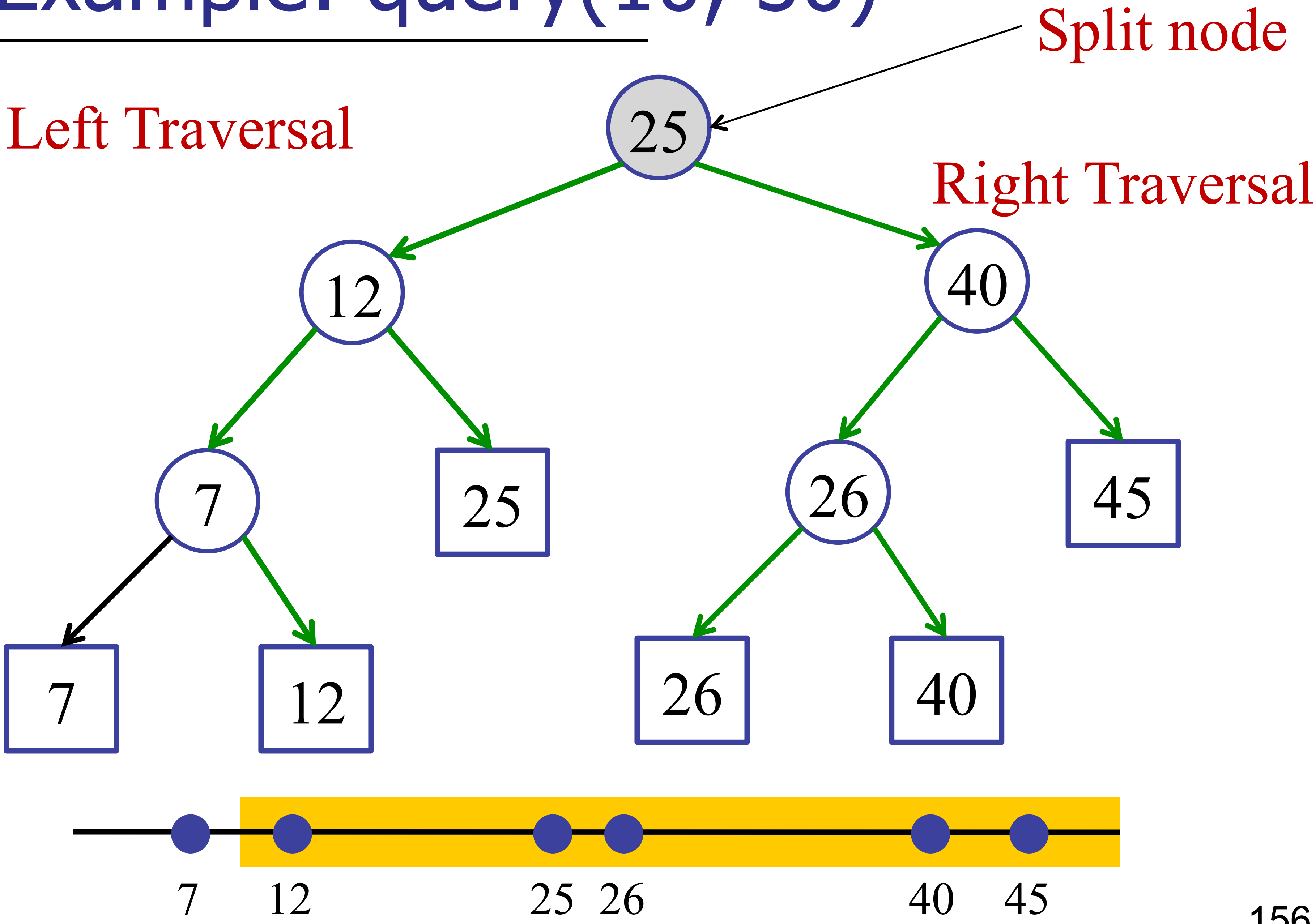
At every step, we either:

1. Output all right sub-tree and recurse left.
2. Recurse right.

– Counting:

1. Recurse at most O(log n) times.
2. How expensive is “output all sub-tree”?

Example: query(10, 50)



Analysis

- Left Traversal:

  At every step, we either:

  1. Output all right sub-tree and recurse left.
  2. Recurse right.

- Counting:

  1. Recurse at most O(log n) times.
  2. "Output all sub-tree" costs O(k).

Analysis

Query time complexity:

$$O(k + \log n)$$

where k is the number of points output.

Preprocessing (buildtree) time complexity:

$$O(n \log n)$$

Total space complexity:

$$O(n)$$

One Dimensional Range Queries

What if you just want to know *how many* points are in the range?

One Dimensional Range Queries

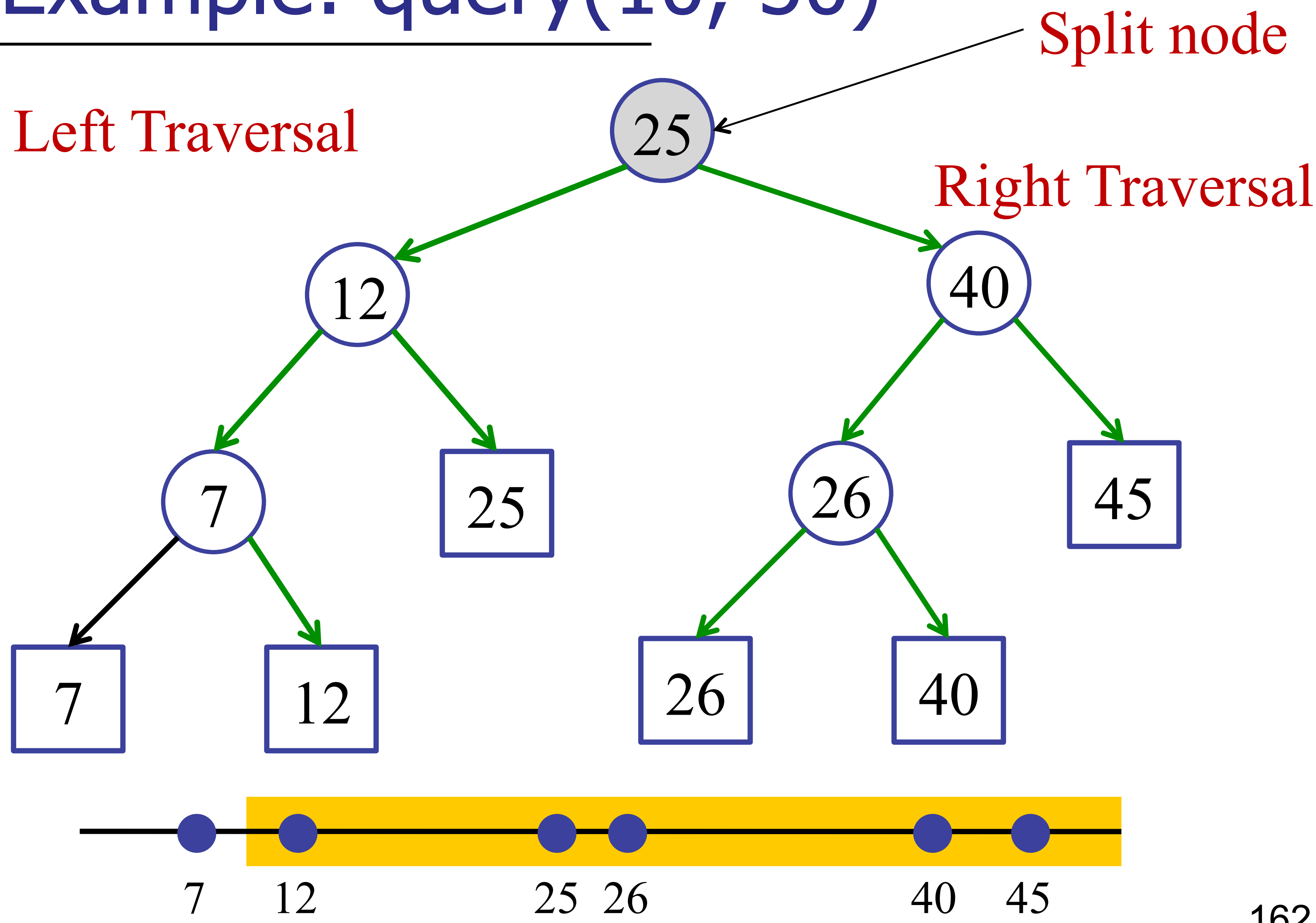
What if you just want to know *how many* points are in the range?

- Augment the tree!
- Keep a count of the number of nodes in each sub-tree.
- Instead of walking entire sub-tree, just remember the count.

One Dimensional Range Queries

```
LeftTraversal(v, low, high)
    if (low <= v.key) {
        ~~all-leaf-traversal(v.right);~~
        total += v.right.count;
        LeftTraversal(v.left, low, high);
    }
    else {
        LeftTraversal(v.right, low, high);
    }
}
```

Example: query(10, 50)



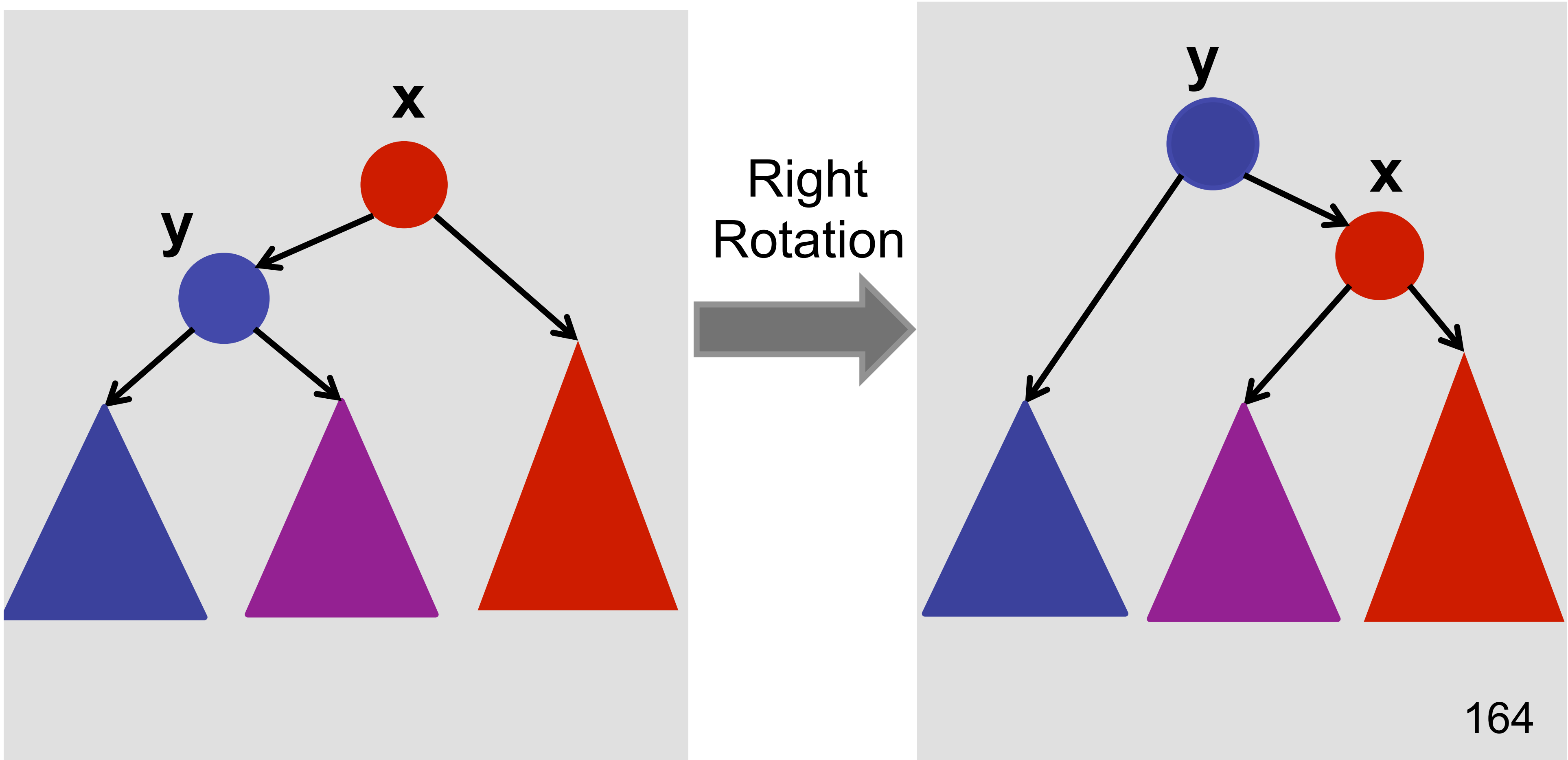
1D Range Tree

Done??

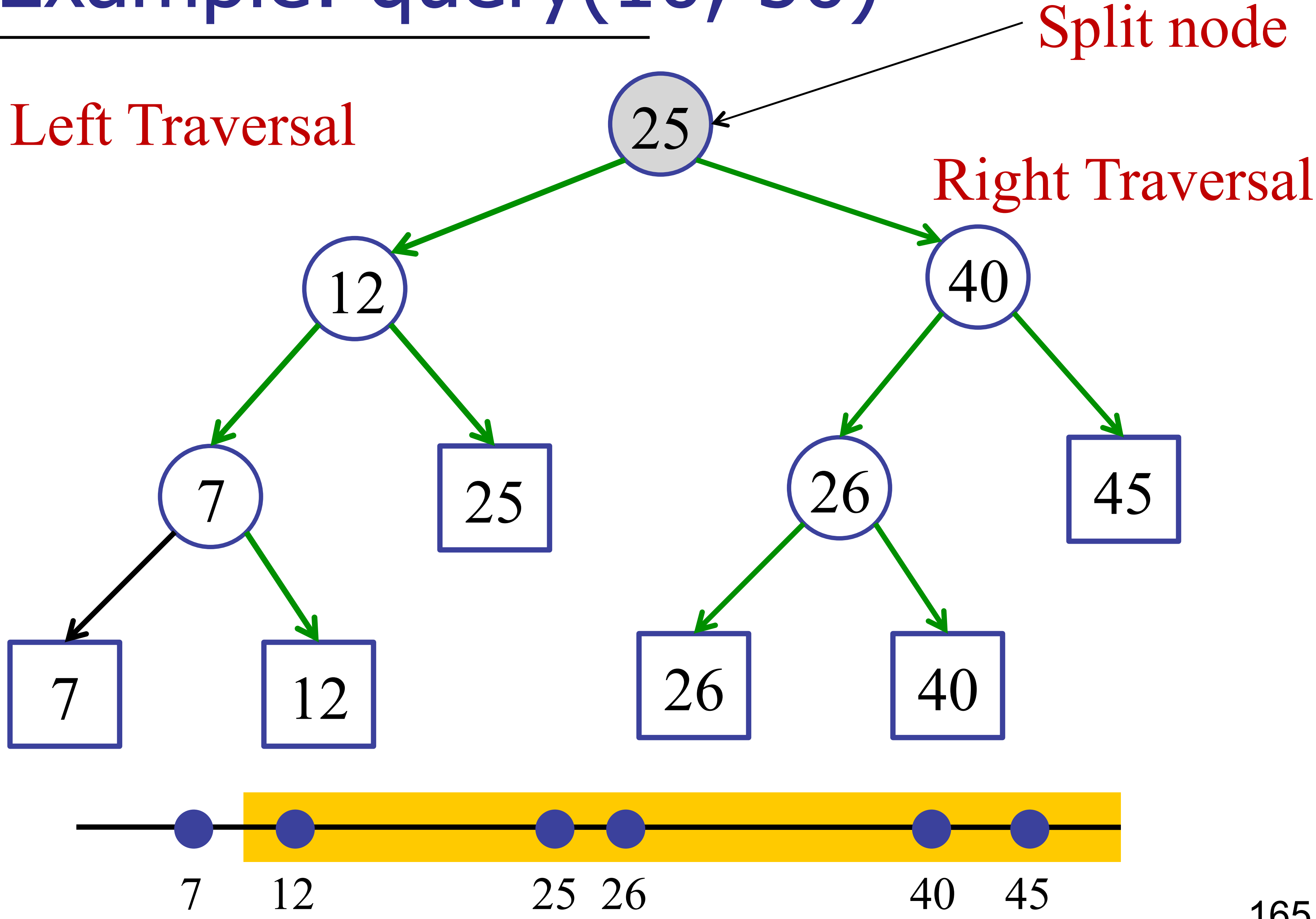
One Dimensional Range Queries

What about dynamic updates?

- Need to verify rotations!

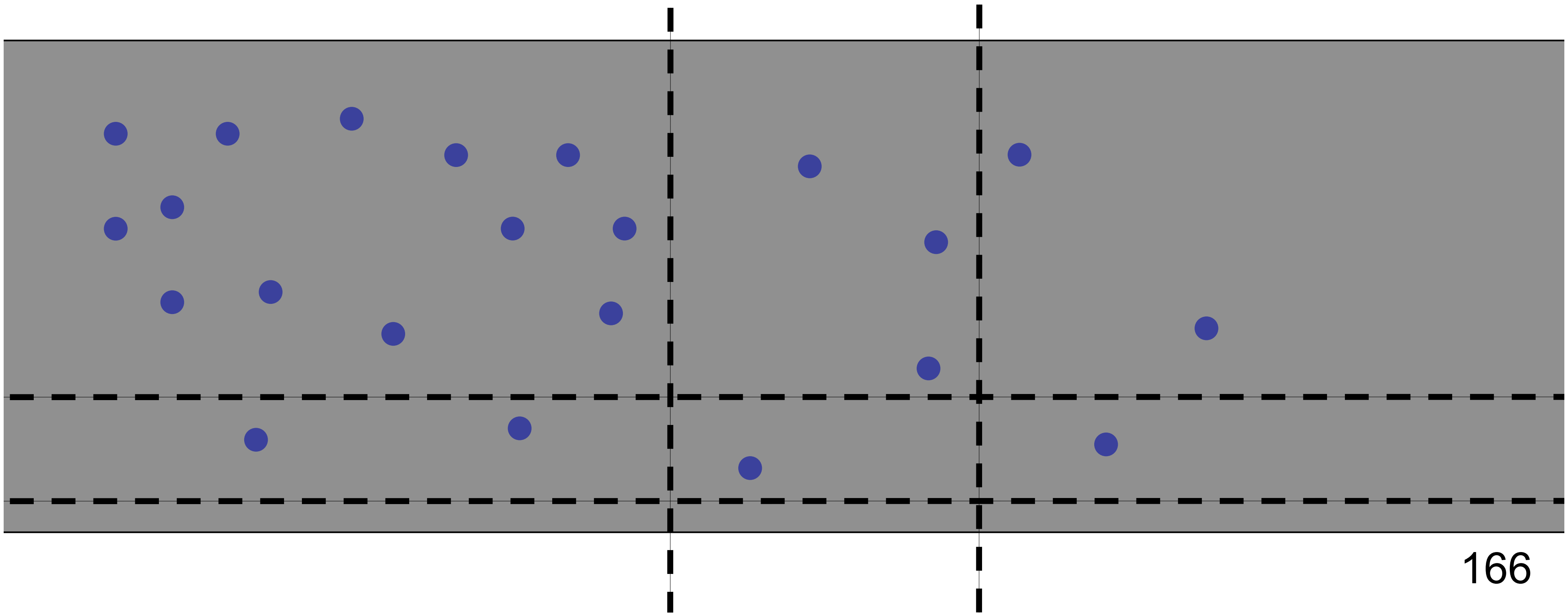


Example: query(10, 50)



Two Dimensional Range Tree

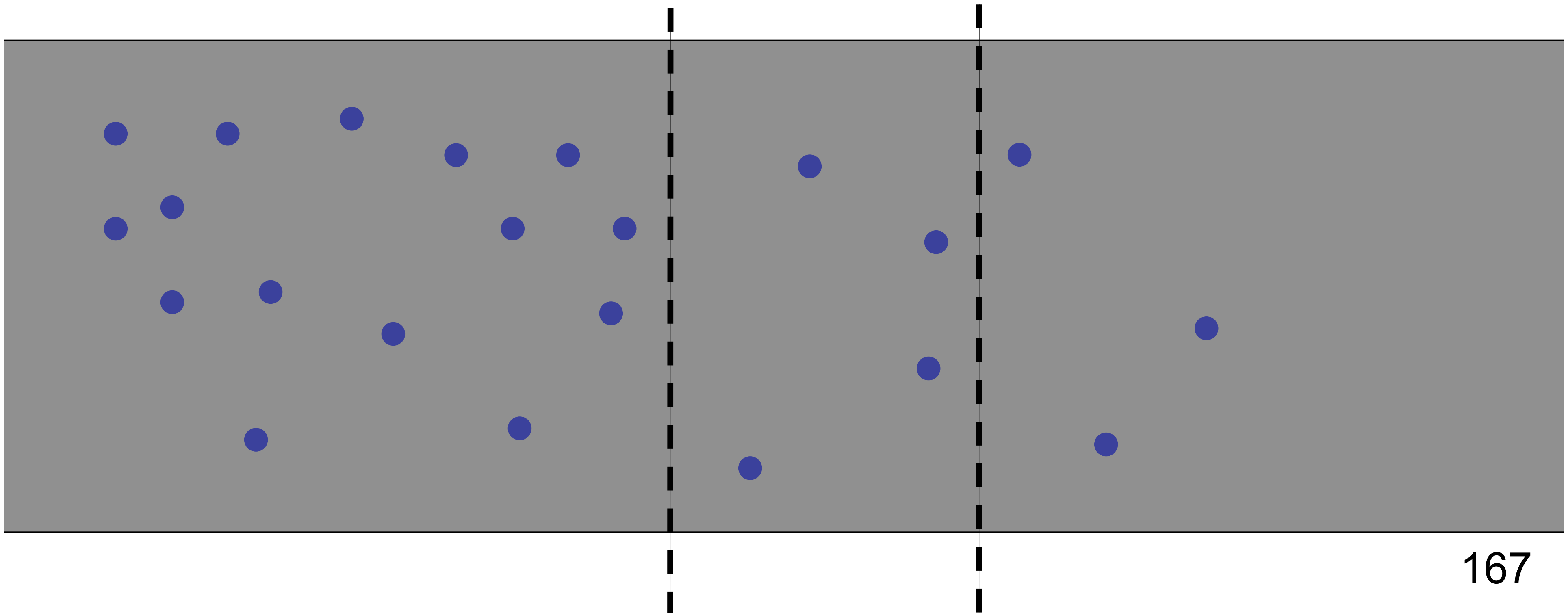
Ex: search for all points between dashed lines.



Two Dimensional Range Tree

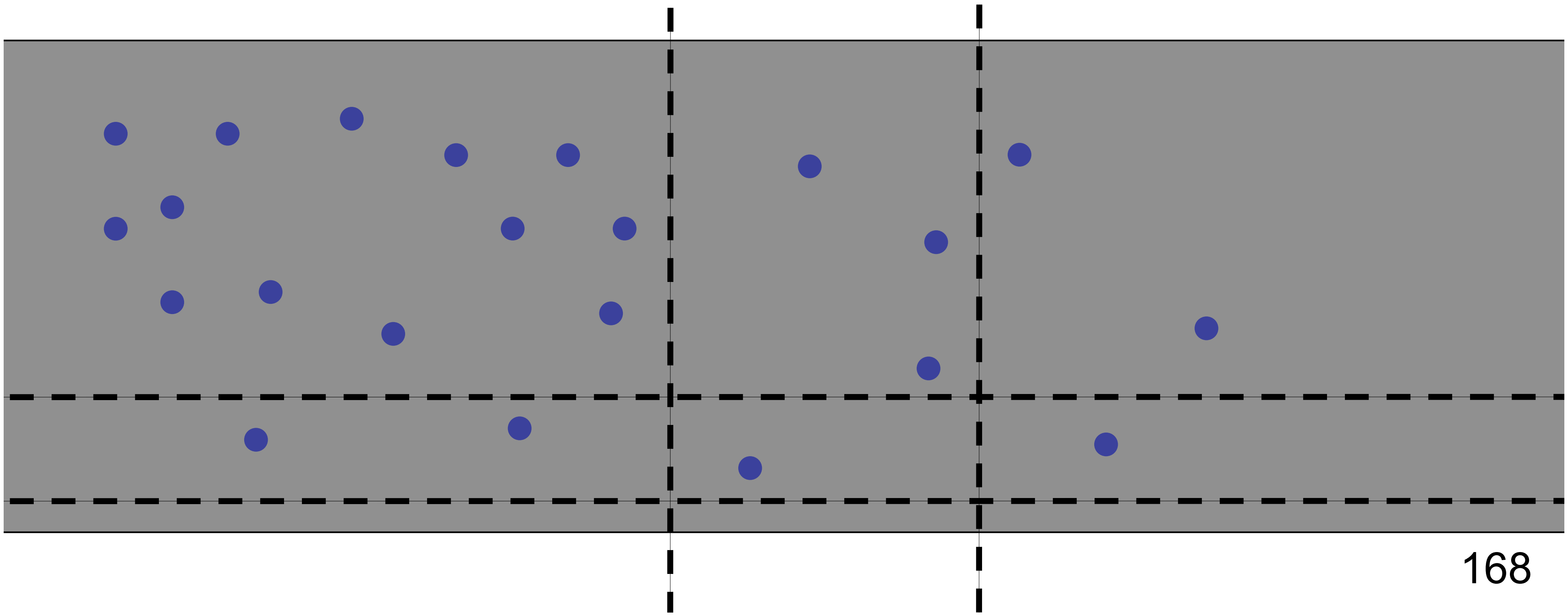
Step 1:

- Create a range-tree on the x-coords.



Two Dimensional Range Tree

Problem: can't enumerate entire sub-trees, since there may be too many nodes that don't satisfy the y-restriction.



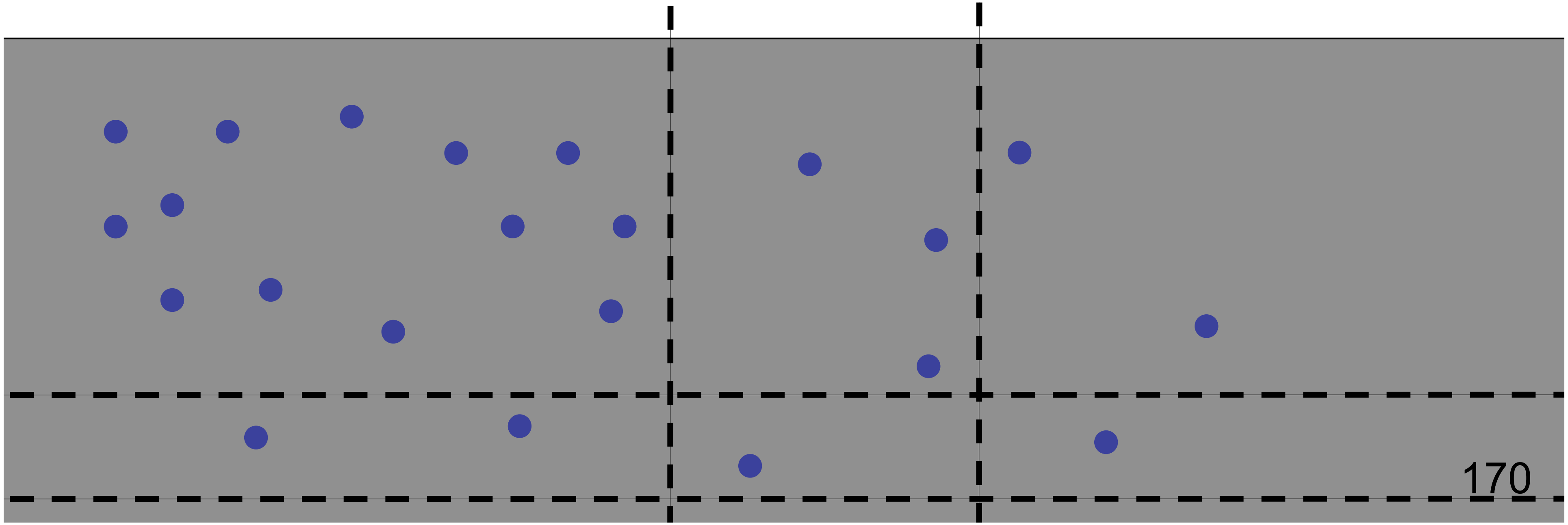
One Dimensional Range Queries

```
LeftTraversal(v, low, high)
    if (v.key >= low) {
            all-leaf-traversal(v.right);
            LeftTraversal(v.left, low, high);
    }
    else {
            LeftTraversal(v.right, low, high);
    }
}
```

Two Dimensional Range Tree

Solution: Augment!

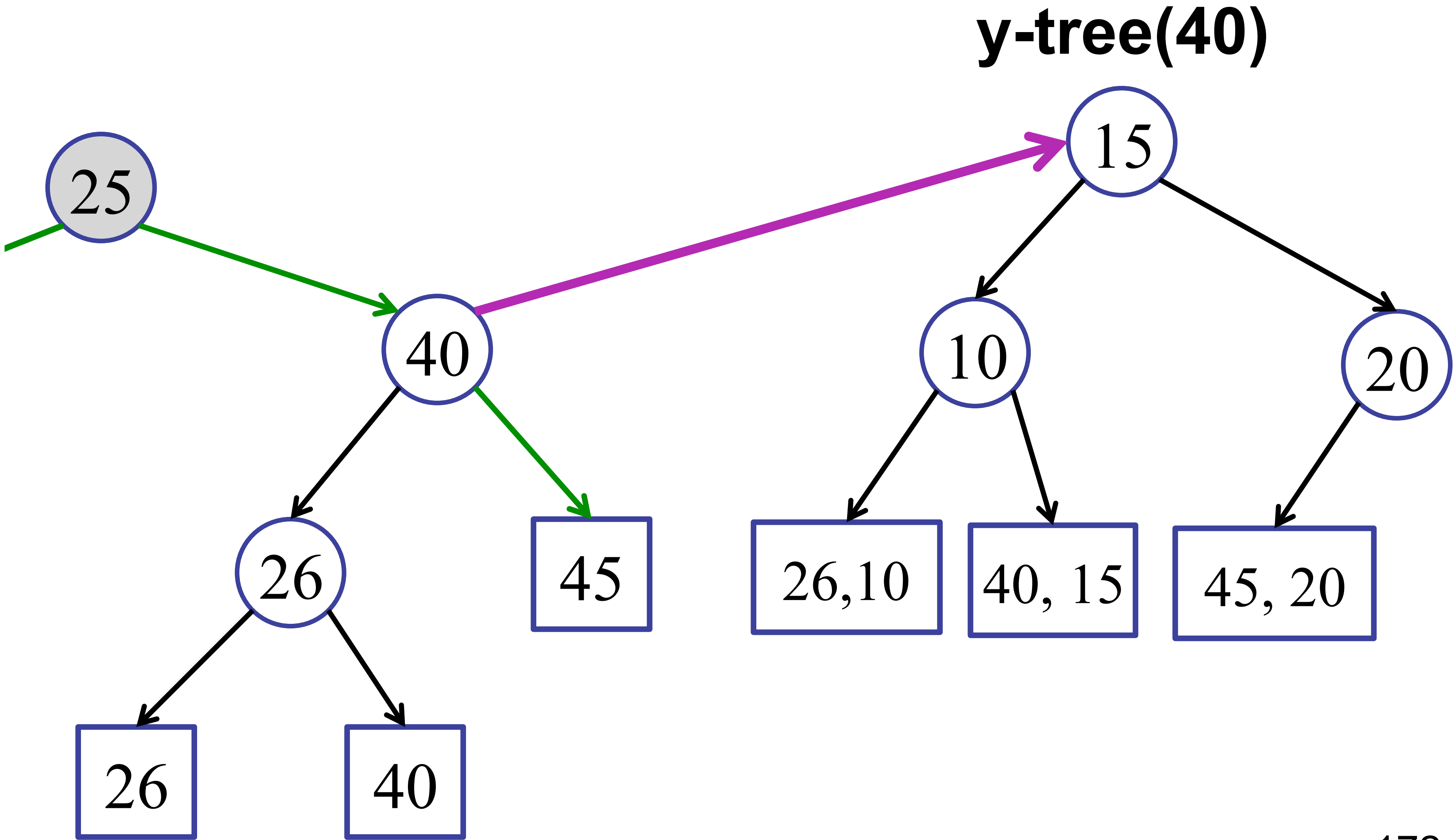
- Each node in the x-tree has a set of points in its sub-tree.
- Store a y-tree at each x-node containing all the points in the sub-tree.



One Dimensional Range Queries

```
LeftTraversal(v, low, high)
    if (v.key.x >= low.x) {
            ytree.search(low.y, highy);
            LeftTraversal(v.left, low, high);
    }
    else {
            LeftTraversal(v.right, low, high);
    }
}
```

Example:



Analysis

Query time: $O(\log^2 n + k)$

- $O(\log n)$ to find split node.
- $O(\log n)$ recurse steps
- $O(\log n)$ y-tree-searches of cost $O(\log n)$
- $O(k)$ enumerating output

Analysis

Space complexity: O(n log n)

- Each point appears in at most one y-tree per level.
- There are at O(log n) levels.
- The rest of the x-tree takes O(n) space.

Analysis

Building the tree: O(n log n)

- Tricky…
- Left as a puzzle… ☺

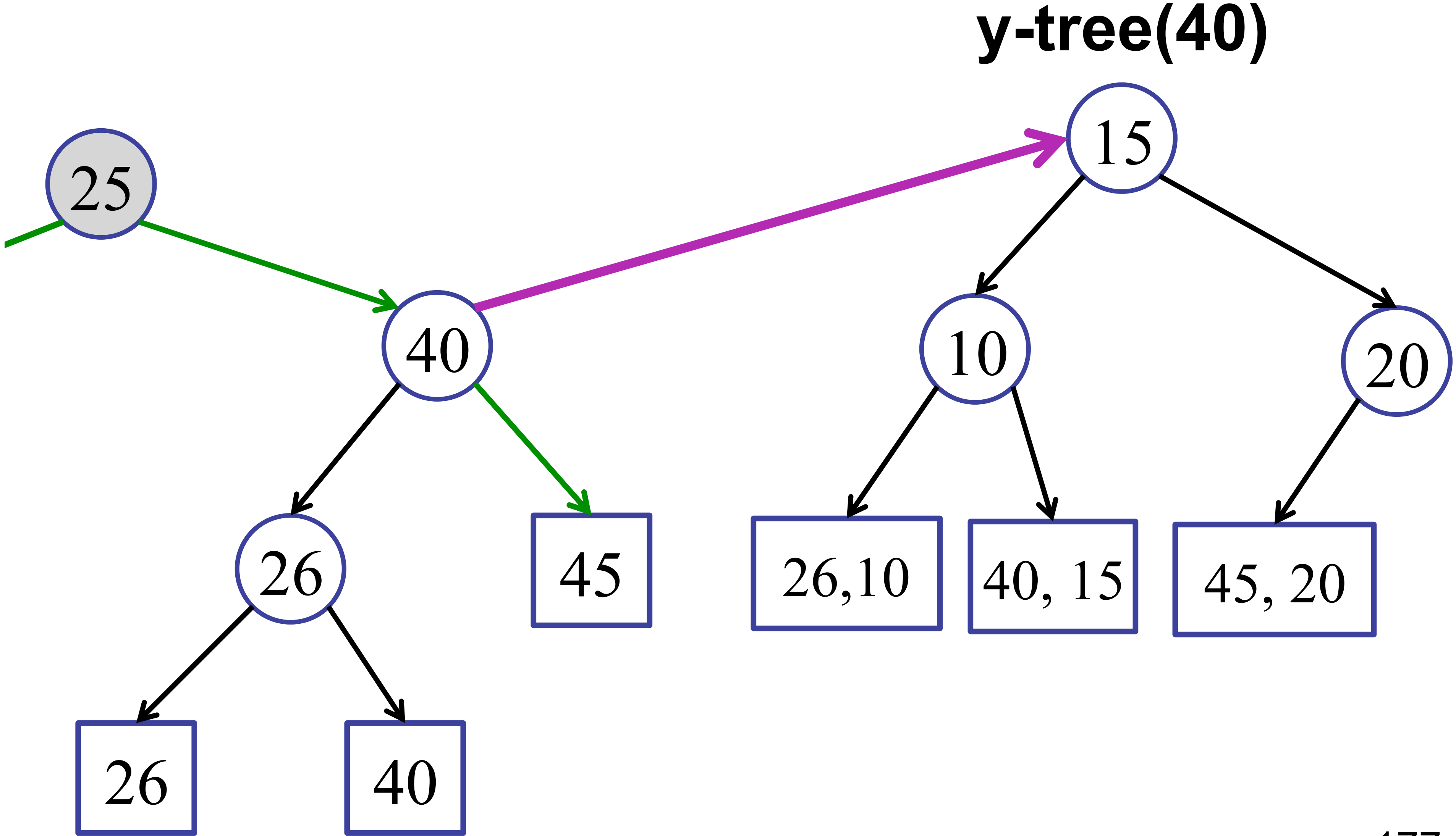
NB Challenge of the Day

Dynamic Trees

What about inserting/deleting nodes?

- Hard!
- How do you do rotations?
- Every rotation you may have to entirely rebuild the y-trees for the rotated nodes.
- Cost of rotate: O(n)   !!!!

Example:



d-dimensional

What if you want high-dimensional range queries?

- Query cost: $O(\log^d n + k)$
- buildTree cost: $O(n \log^{d-1} n)$
- Space: $O(n \log^{d-1} n)$

Idea:

- Store d–1 dimensional range-tree in each node of a 1D range-tree.
- Construct the d–1-dimeionsal range-tree recursively.

Curse of Dimensionality

What if you want high-dimensional range queries?

- Query cost: $O(\log^d n + k)$
- buildTree cost: $O(n \log^{d-1} n)$
- Space: $O(n \log^{d-1} n)$

Idea:

- Store d–1 dimensional range-tree in each node of a 1D range-tree.
- Construct the d–1-dimeionsal range-tree recursively.

Real World (aside)

kd-Trees

- Alternate levels in the tree:
  - vertical
  - horizontal
  - vertical
  - horizontal
- Each level divides the points in the plane in half.

Real World (aside)

kd-Trees

- Alternate levels in the tree
- Each level divides the points in the plane in half.
- Query cost: $O(\sqrt{n})$ worst-case
  - Sometimes works better in practice for many queries.
  - Easier to update dynamically.
  - Good for other types of queries: e.g., nearest-neighbor

Today

Three examples of augmenting BSTs

1. Order Statistics

2. Intervals

3. Orthogonal Range Searching