CS2020
Data Structures and Algorithms

Welcome!

Why Learn Binary Search Trees?

- For "ordered" applications:
  - Use Java built-in libraries!
  - Or, use a real database!
    - BerkeleyDB, MySQL, etc.
  - Faster and more efficient than my code…

- For "dictionary" applications:
  - Hash tables are faster.

Why Learn Binary Search Trees?

1. You have to understand the underlying data structure to use it efficiently.
   - When to use SkipList vs. B-tree vs. Hash table?
   - Which operations are expensive/slow?
2. Many problems require modifying the underlying data structures.
   - If you are limited to existing operations, it may be hard to efficiently solve your problem.
   - Sometimes: all new data structure…
   - More often: **augment** existing data structures.

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

Two examples of augmenting BSTs

1. Order Statistics
   - Rank
   - Select

2. Orthogonal Range Searching
   - Geometric search problem
   - 1-dimension / 2-dimension

Augmented Search Trees

Dynamic Order Statistics

Implement a binary search tree that supports:

- insert(int key)
- search(int key)

and also:

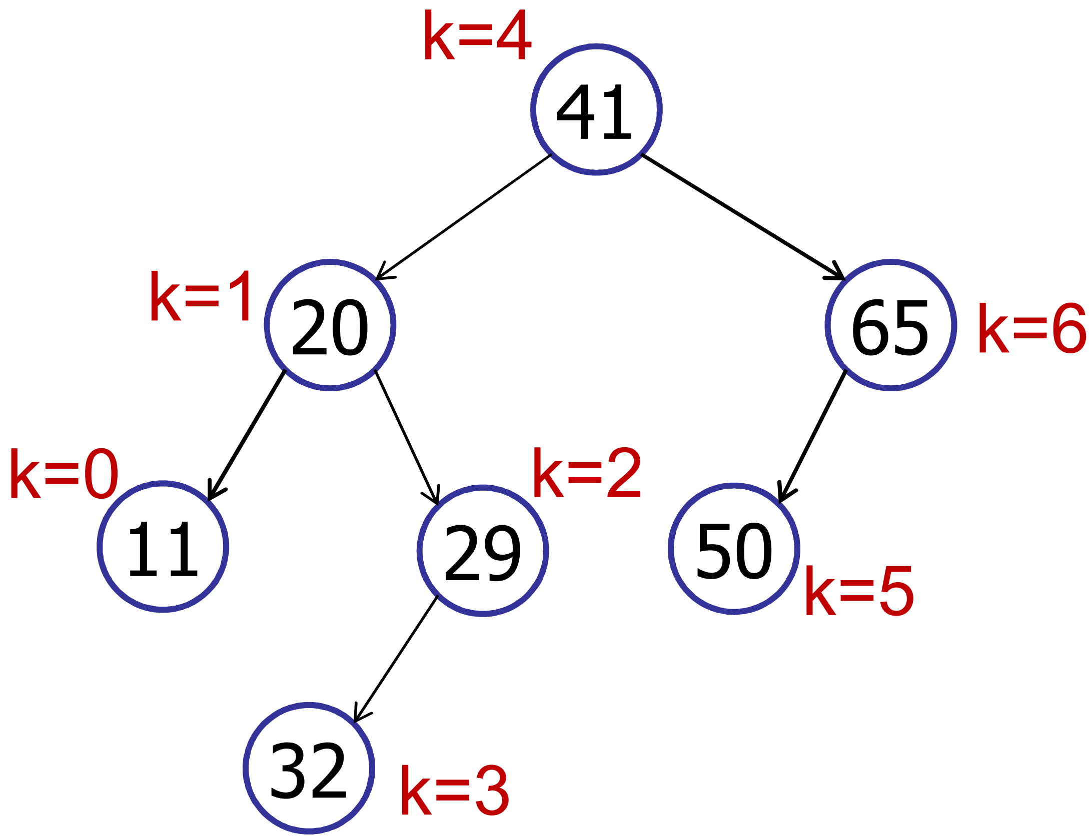
- select(int k)

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

select(4)

Dynamic Order Statistics

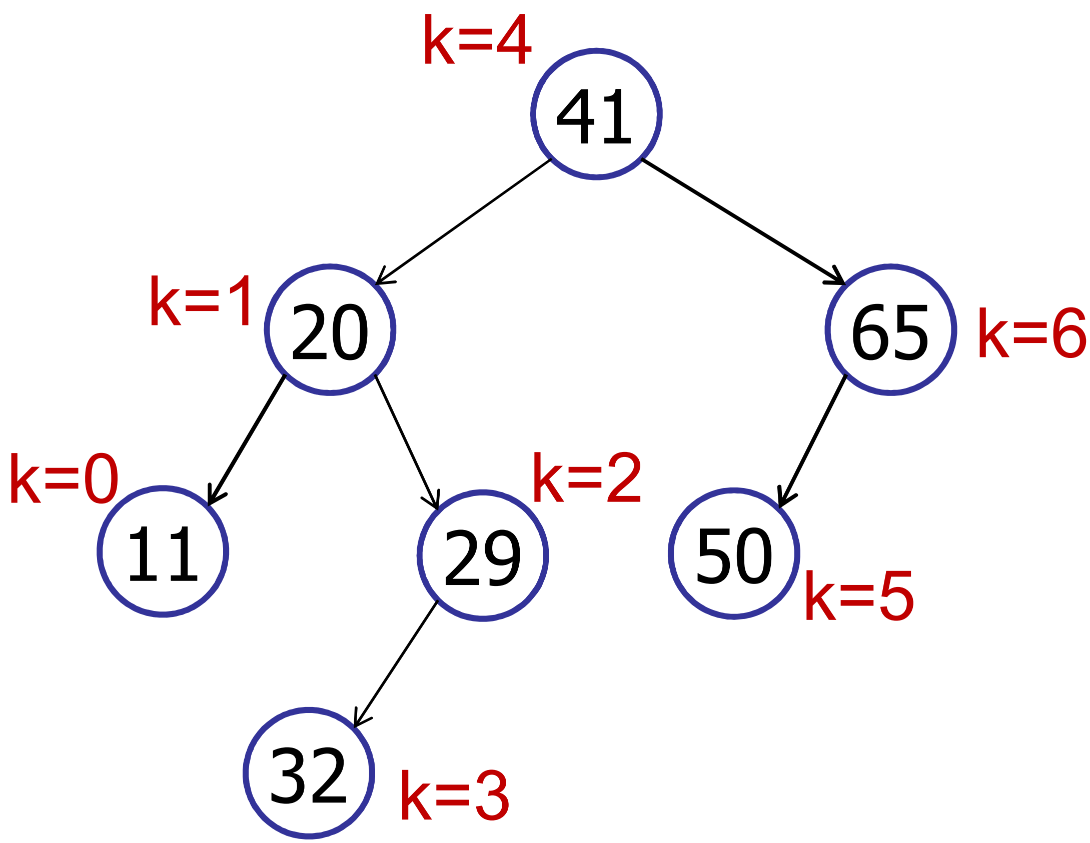
Option 1: store rank in every node



(Nota bene: k=rank, not height.)

Dynamic Order Statistics

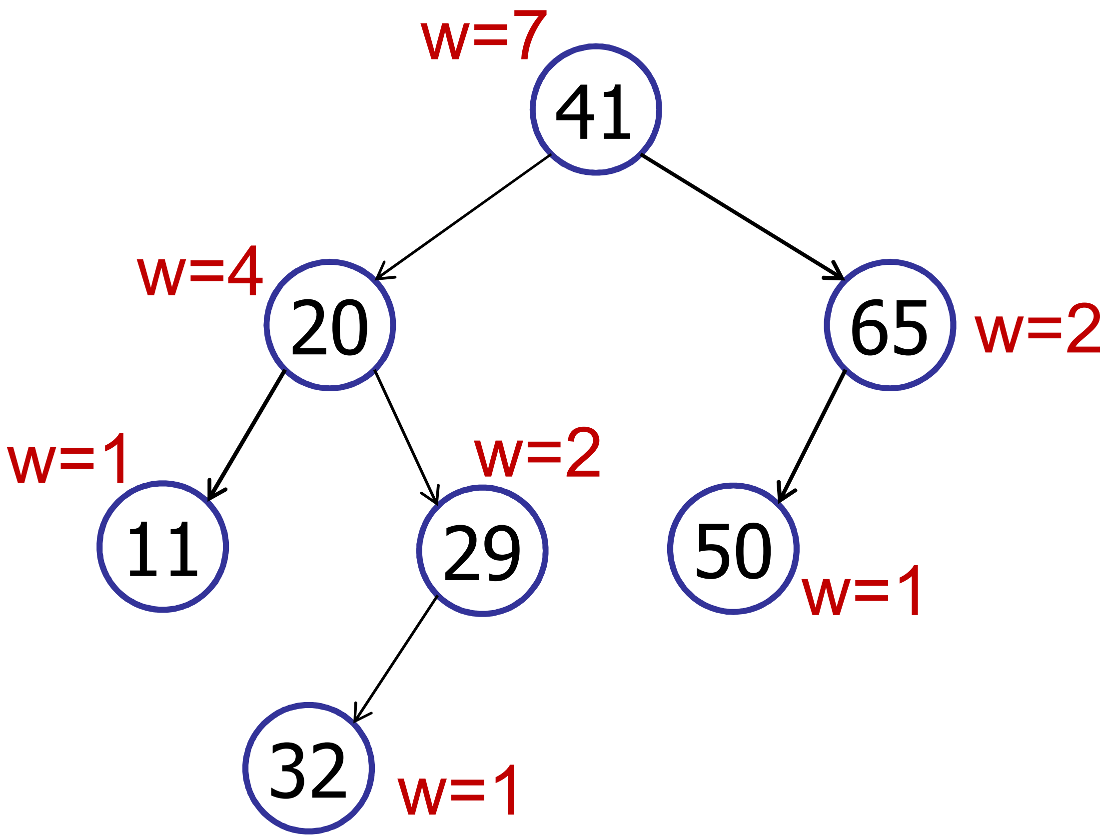
Option 1: store rank in every node



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

Dynamic Order Statistics

Option 2: store size of sub-tree in every node



Nota bene: w=weight, not height.

Dynamic Order Statistics

Option 2: store size of sub-tree in every node

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

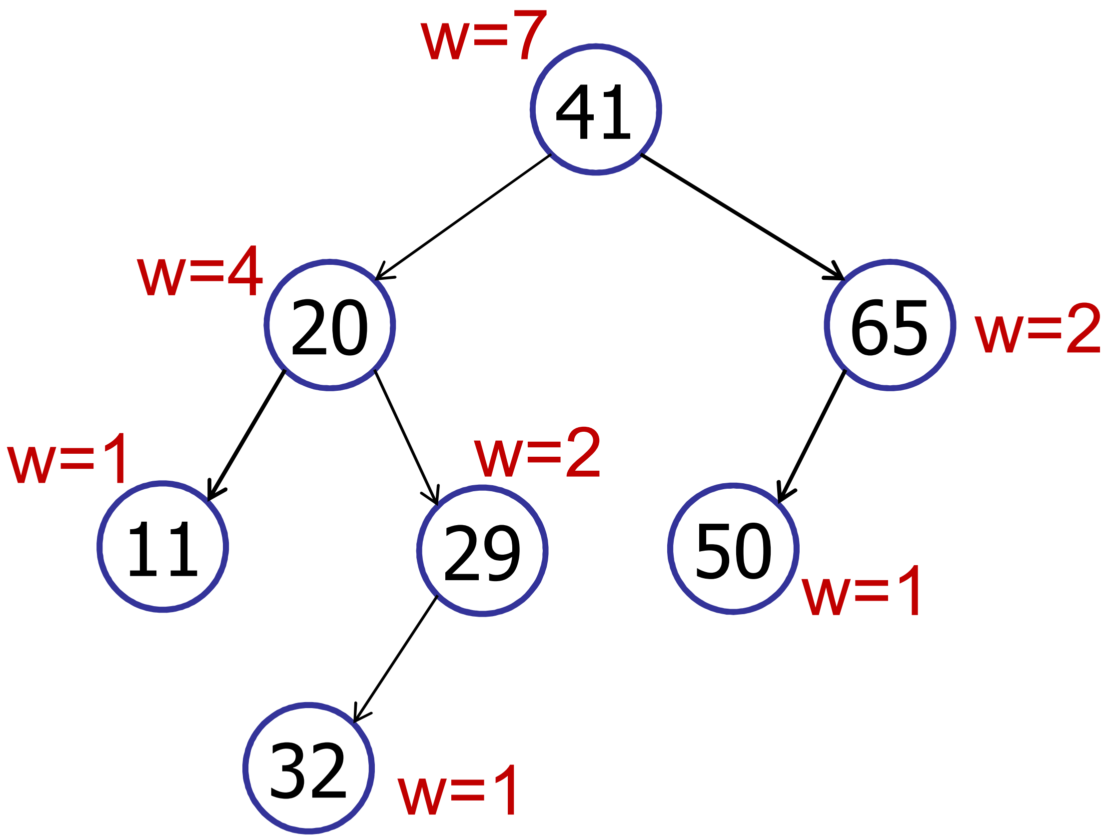
Define weight:

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

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

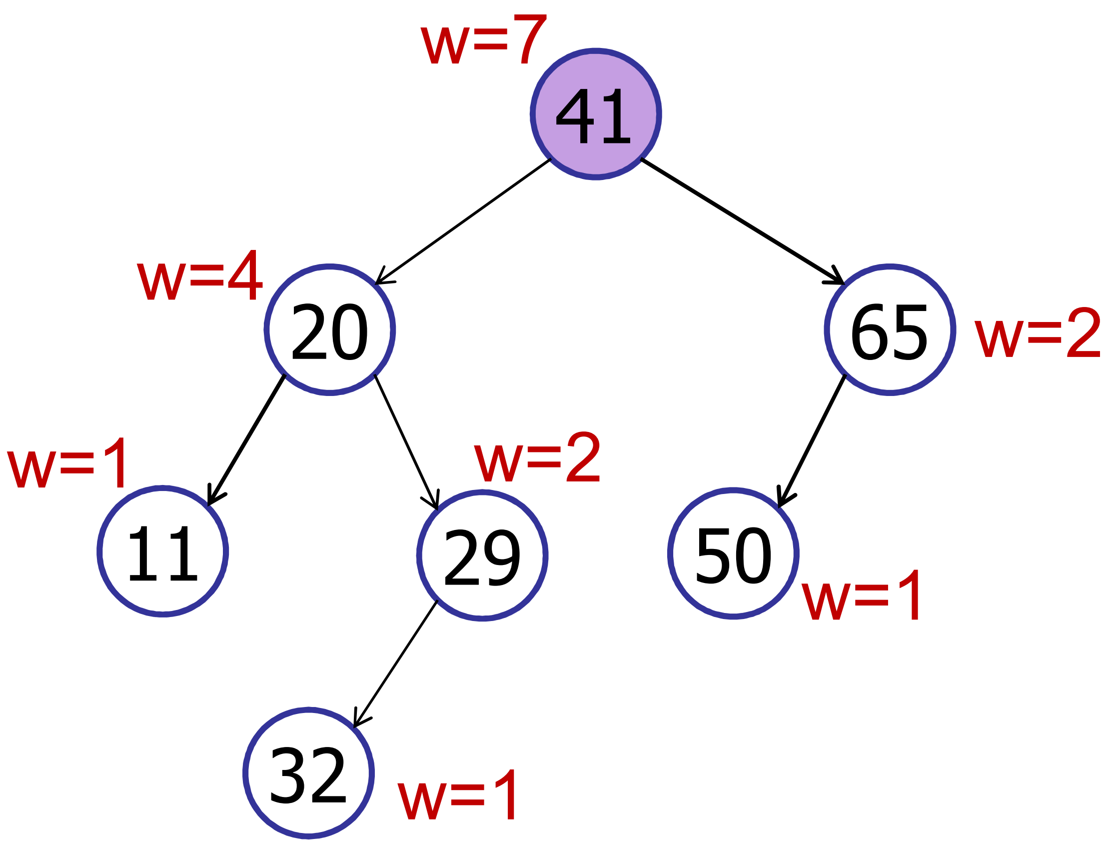
Dynamic Order Statistics

Option 2: store size of sub-tree in every node



Dynamic Order Statistics

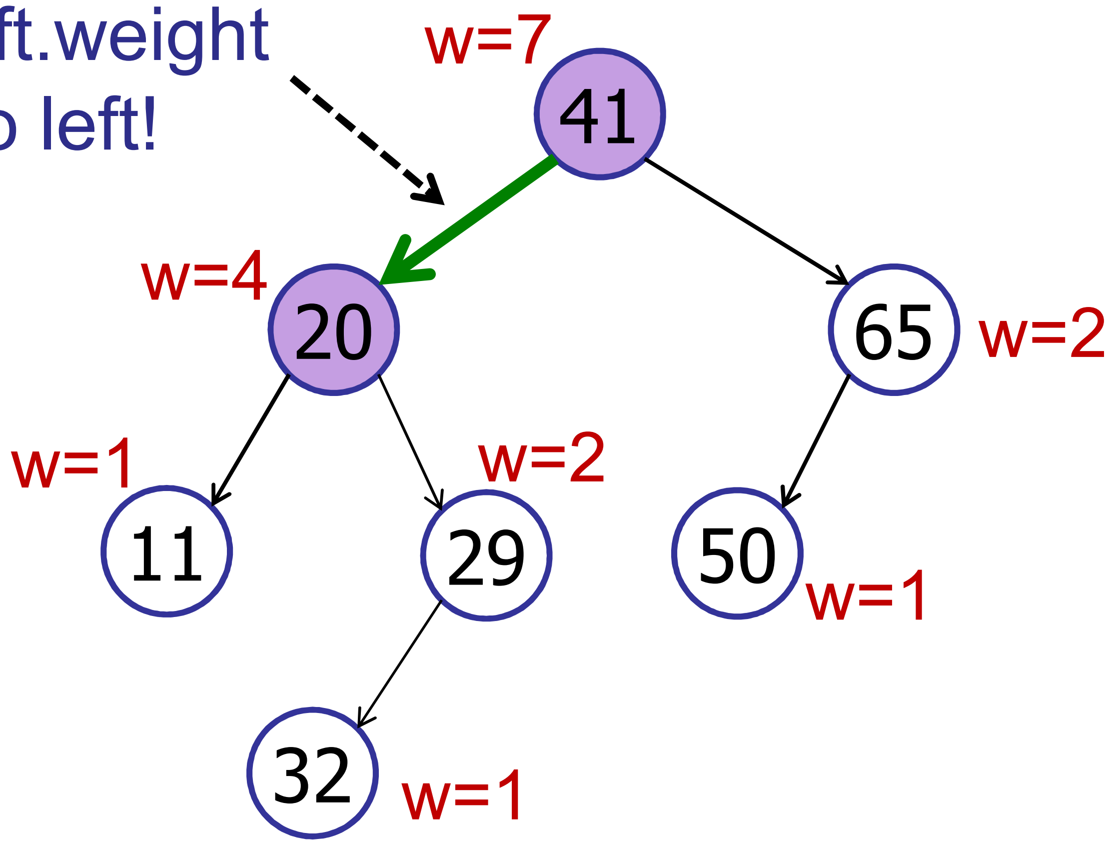
Example: select(3)



Dynamic Order Statistics

Example: select(3)

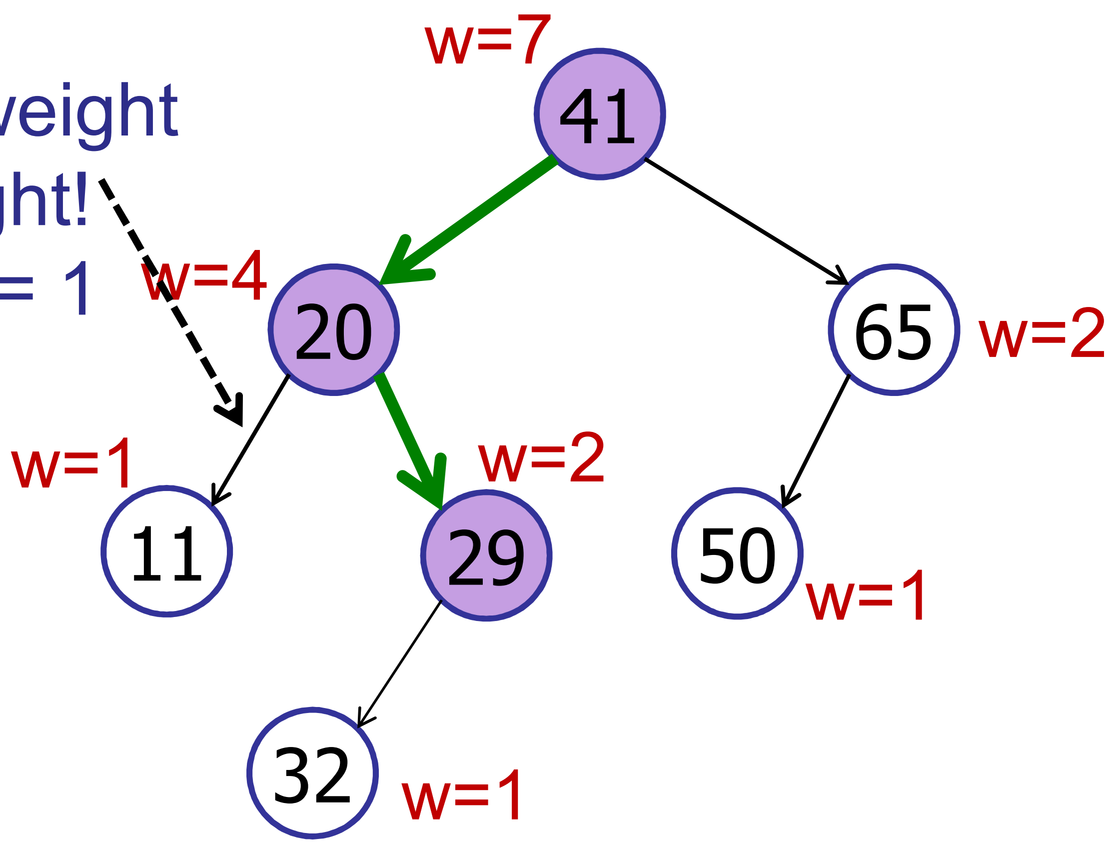
3 ≤ left.weight
Go left!



Dynamic Order Statistics

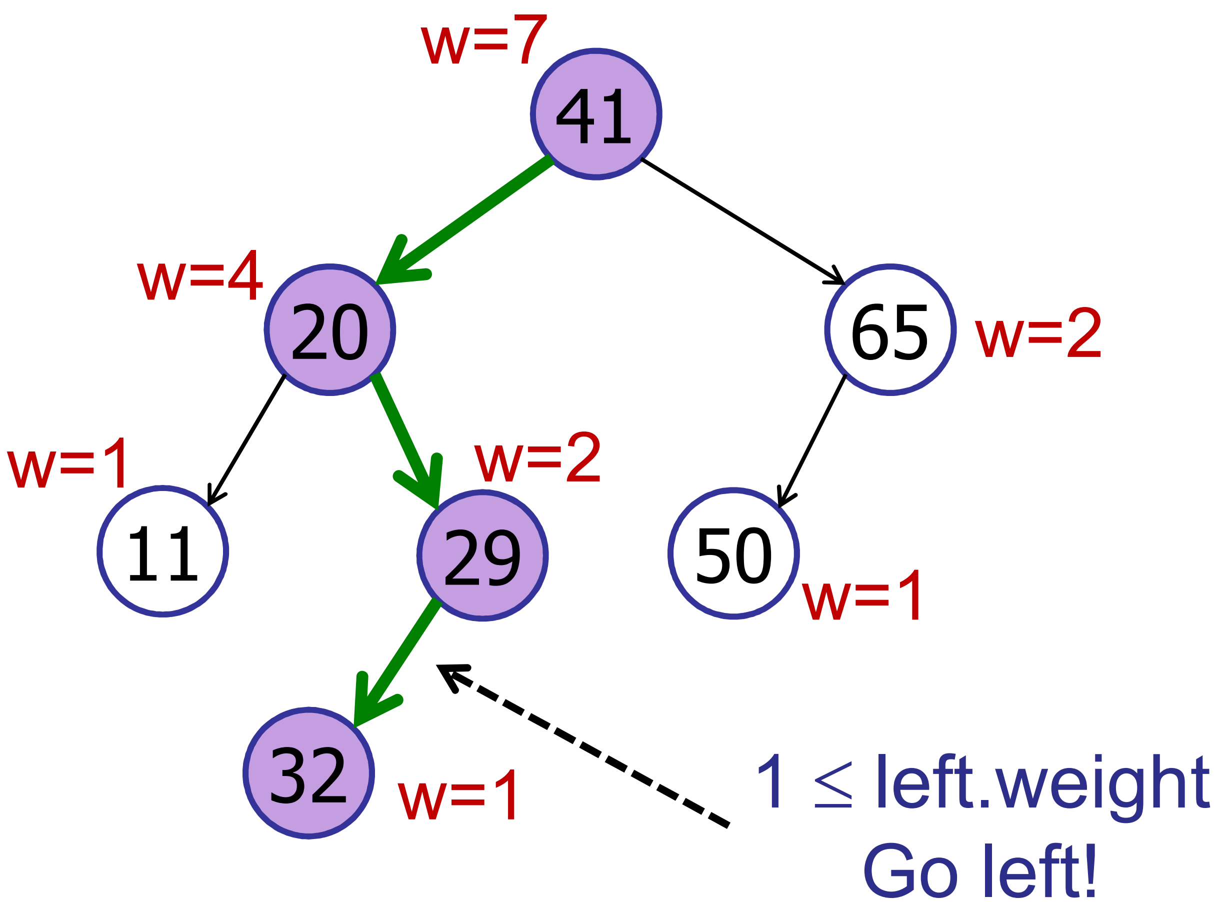
Example: select(3)

3 ≥ left.weight
Go right!
3-1-1 = 1



Dynamic Order Statistics

Example: select(3)



Dynamic Order Statistics

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

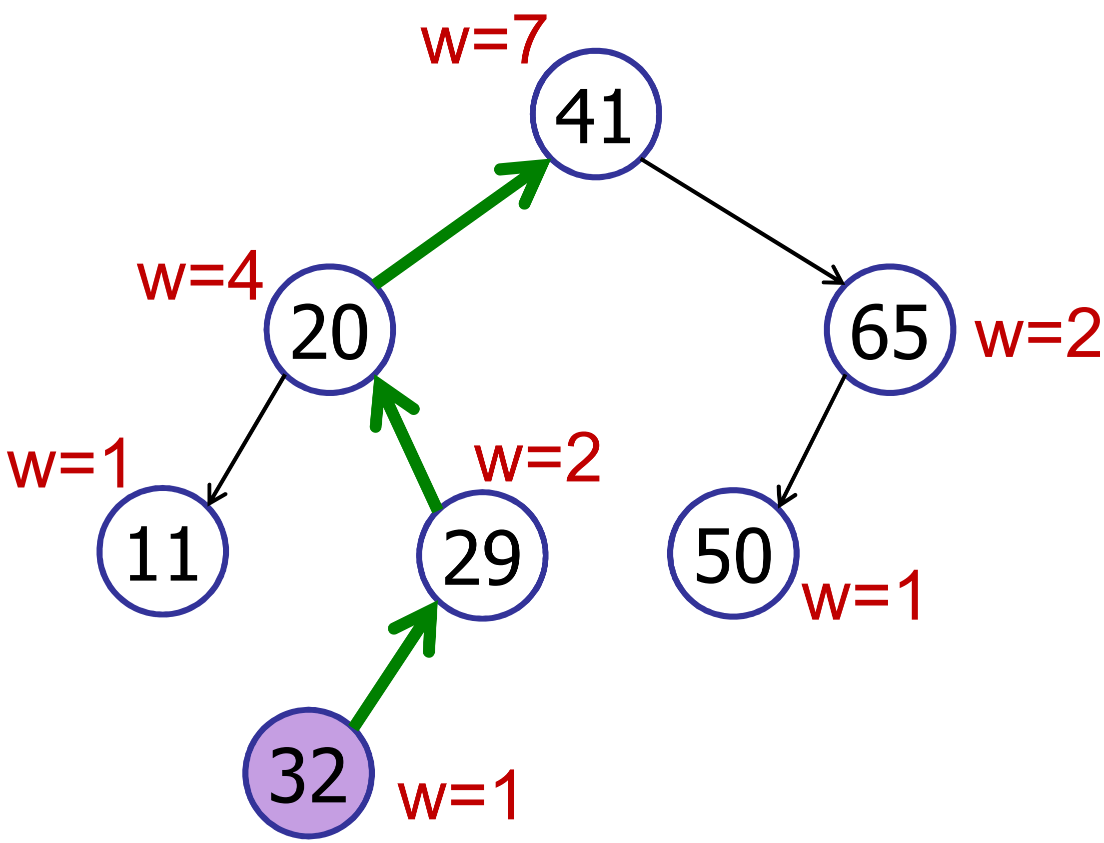
Dynamic Order Statistics

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

```
rank(v)
    r = v.left.weight + 1;
    while (v != root) do
        if v is right child then
            r += y.parent.left.weight + 1
        y = y.parent
    return r;
```

Dynamic Order Statistics

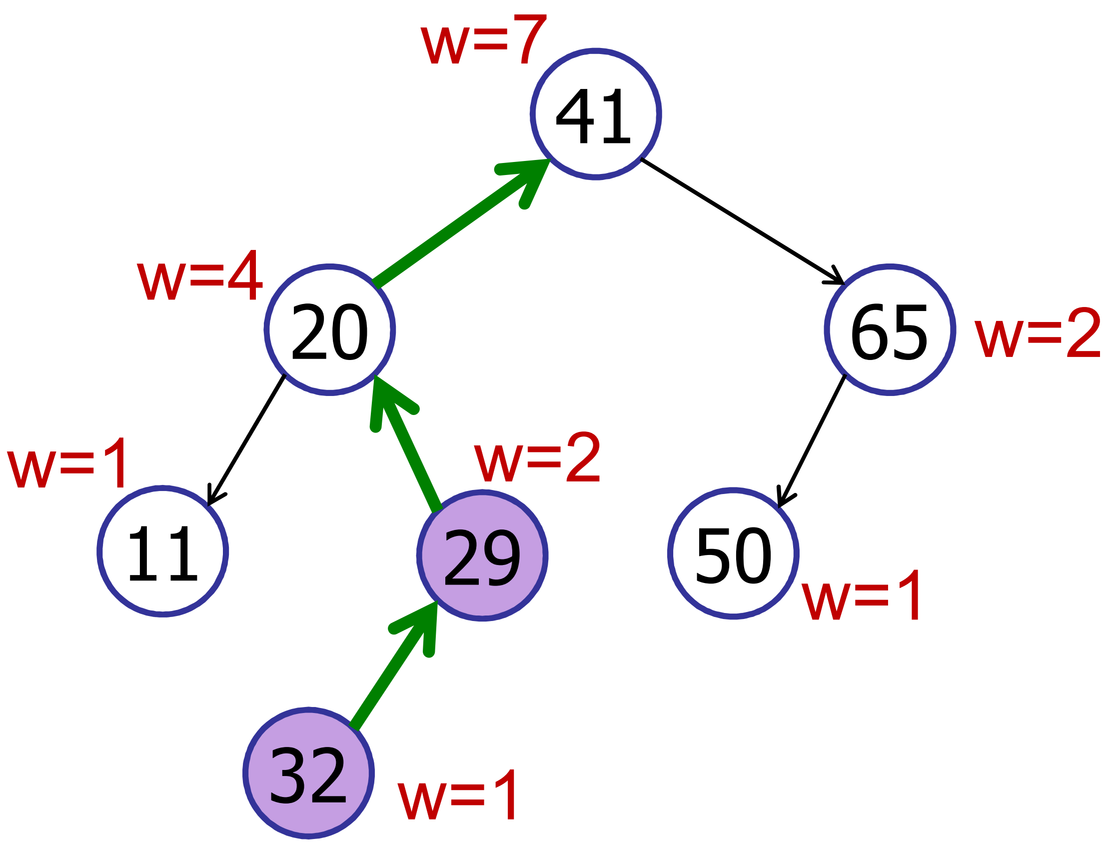
Example: rank(32)



rank = 1

Dynamic Order Statistics

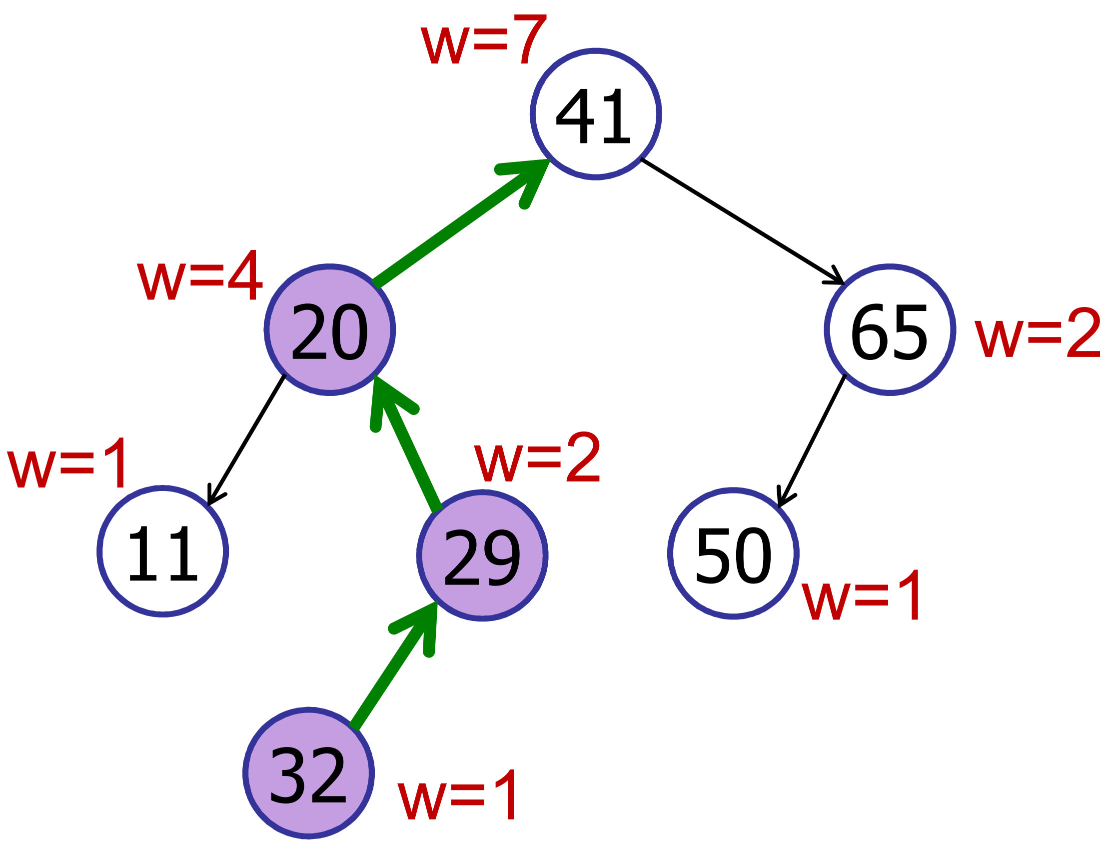
Example: rank(32)



rank = 1

Dynamic Order Statistics

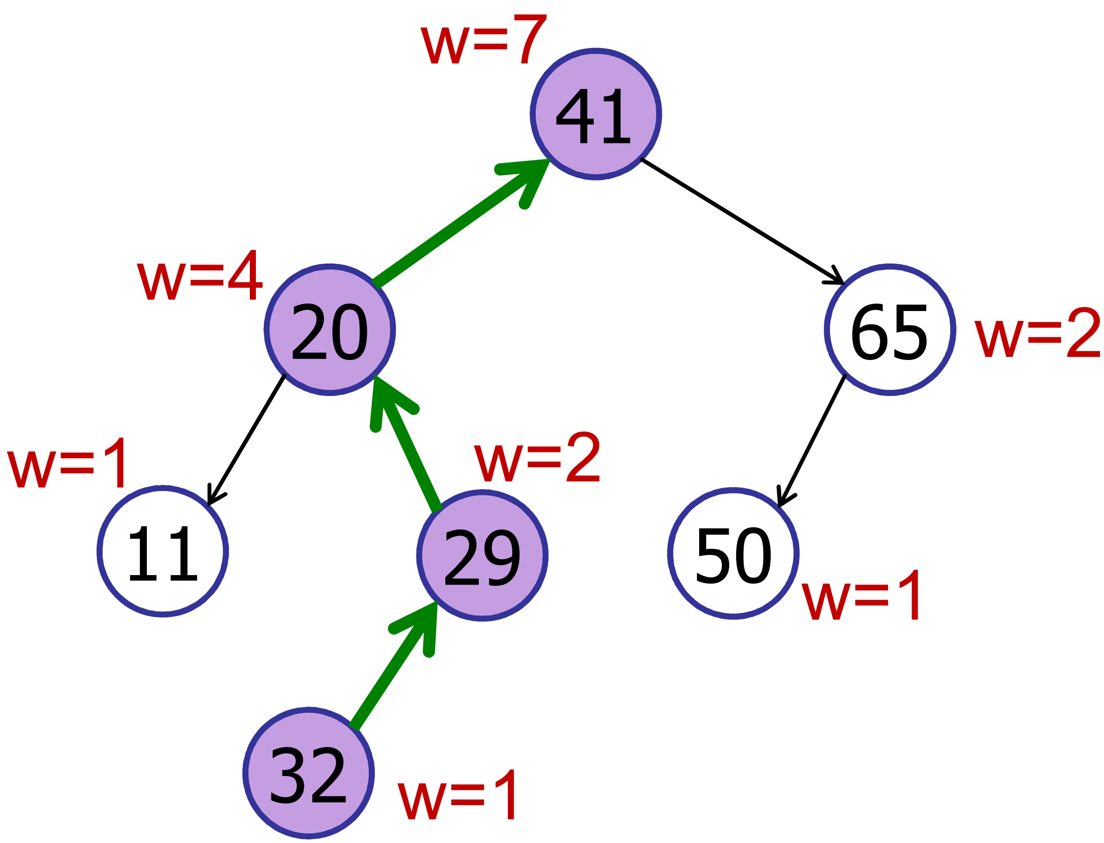
Example: rank(32)



rank = 1 + 2

Dynamic Order Statistics

Example: rank(32)

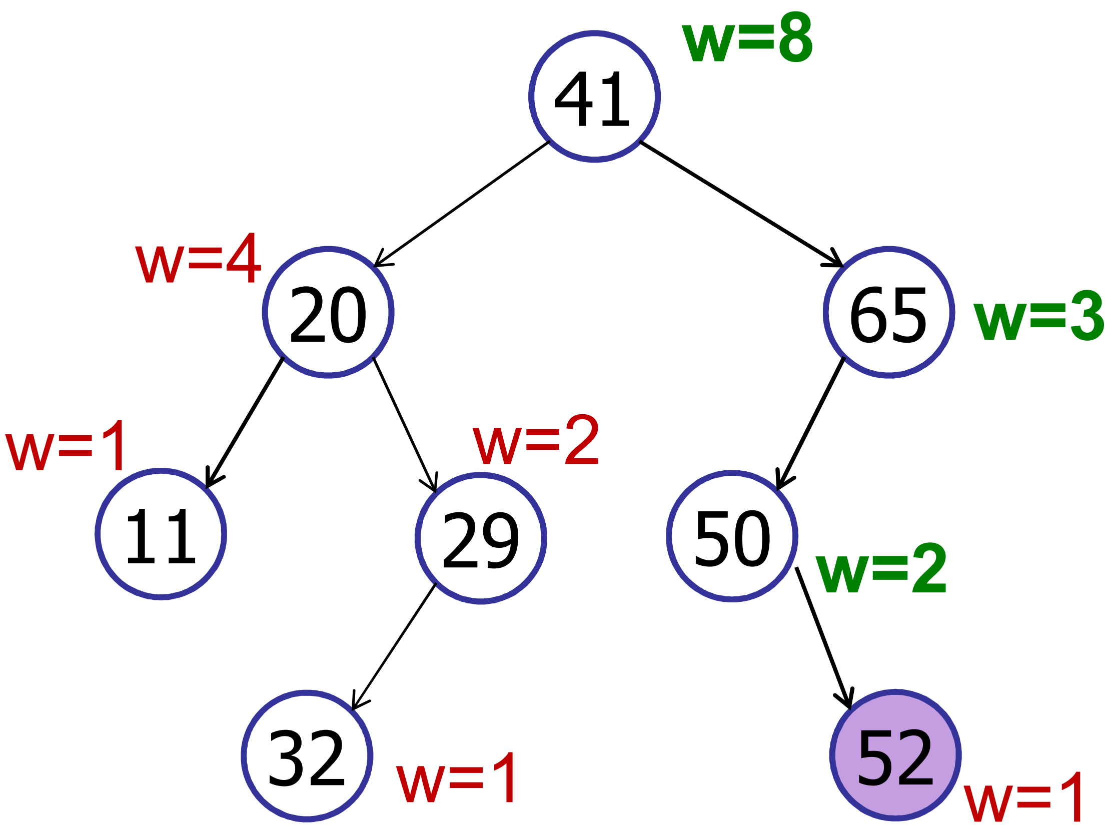


rank = 1 + 2 = 3

Augmented Trees

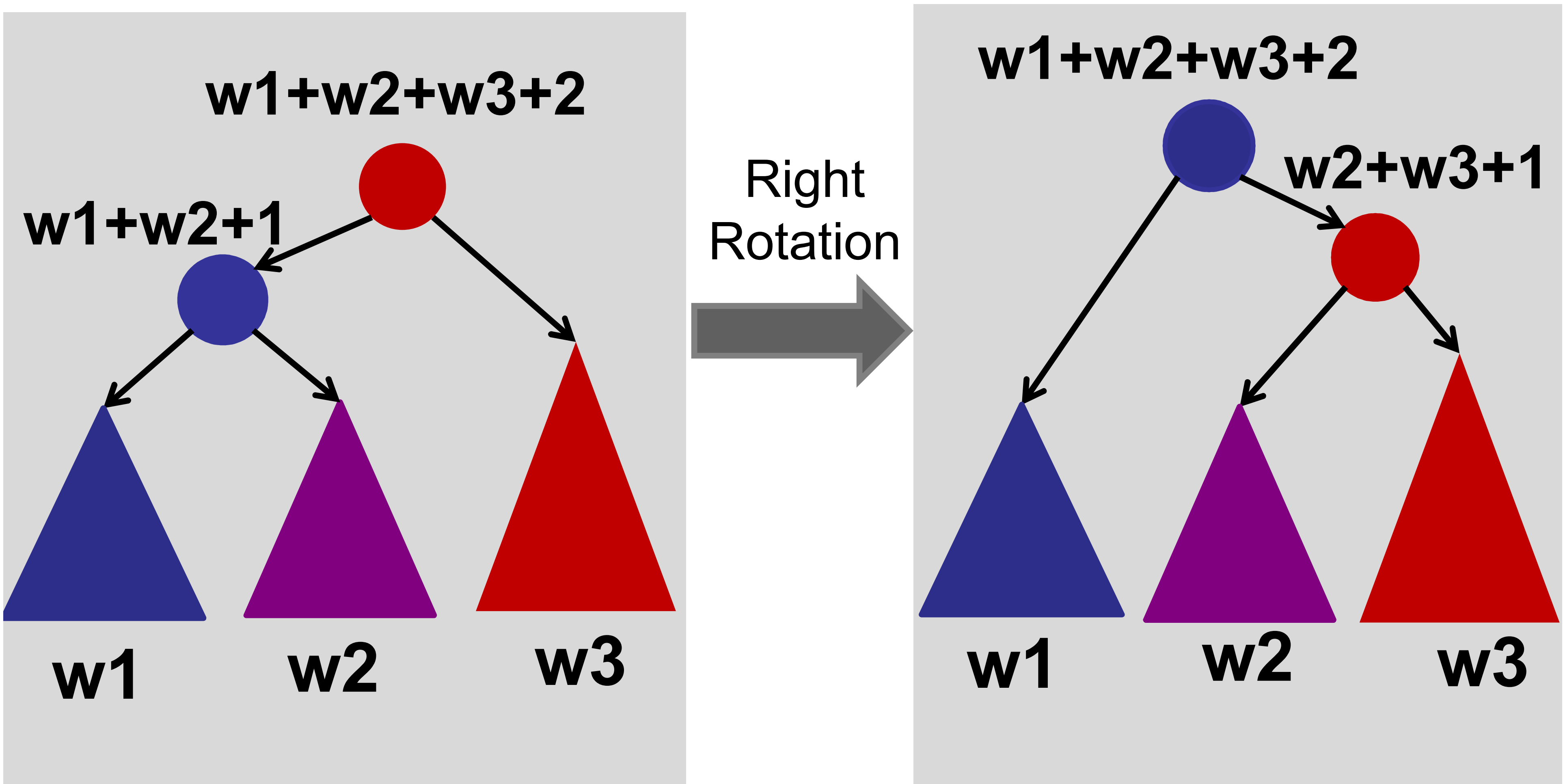
Maintain weight during insertions:

- Just like maintaining height…



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.

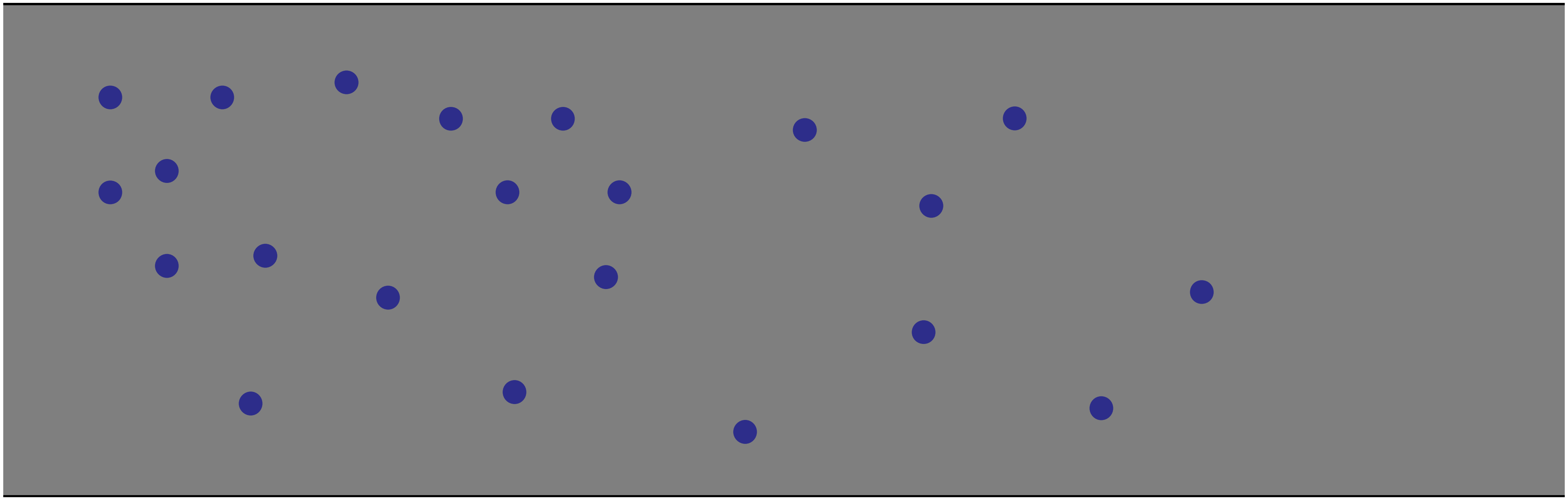
3. Verify that the additional info can be maintained as the data structure is modified.

   (subject to insert/delete/etc.)

4. Develop new operations using the new info.

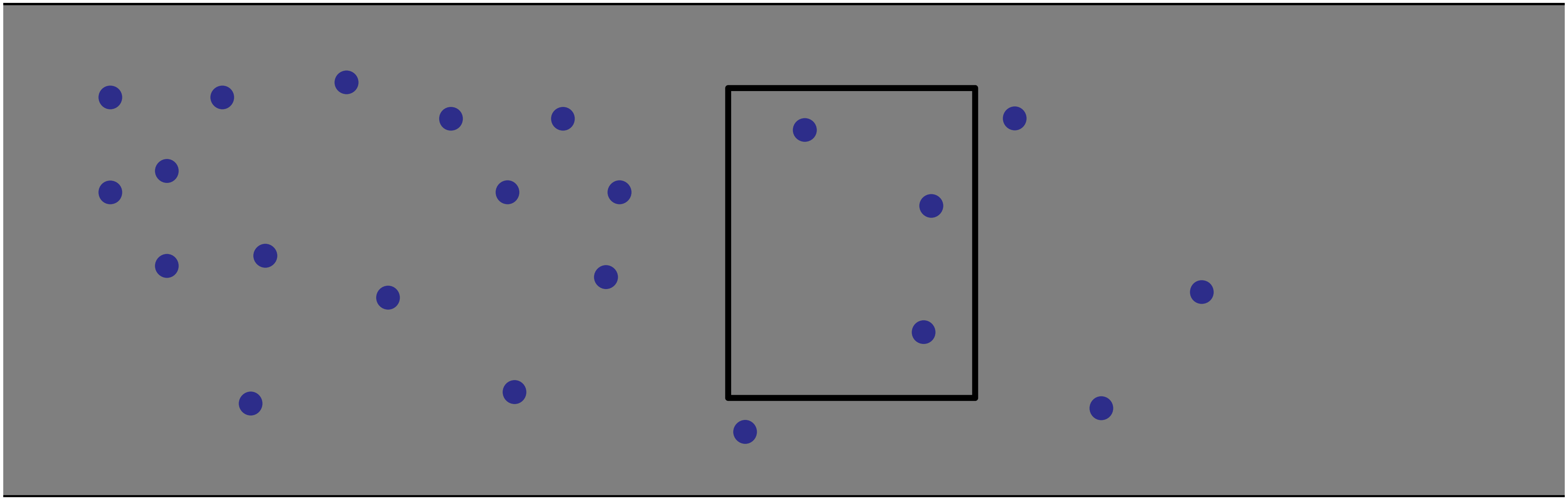
Orthogonal Range Searching

Input: n points in a 2d plane



Orthogonal Range Searching

Input: *n* points in a 2d plane

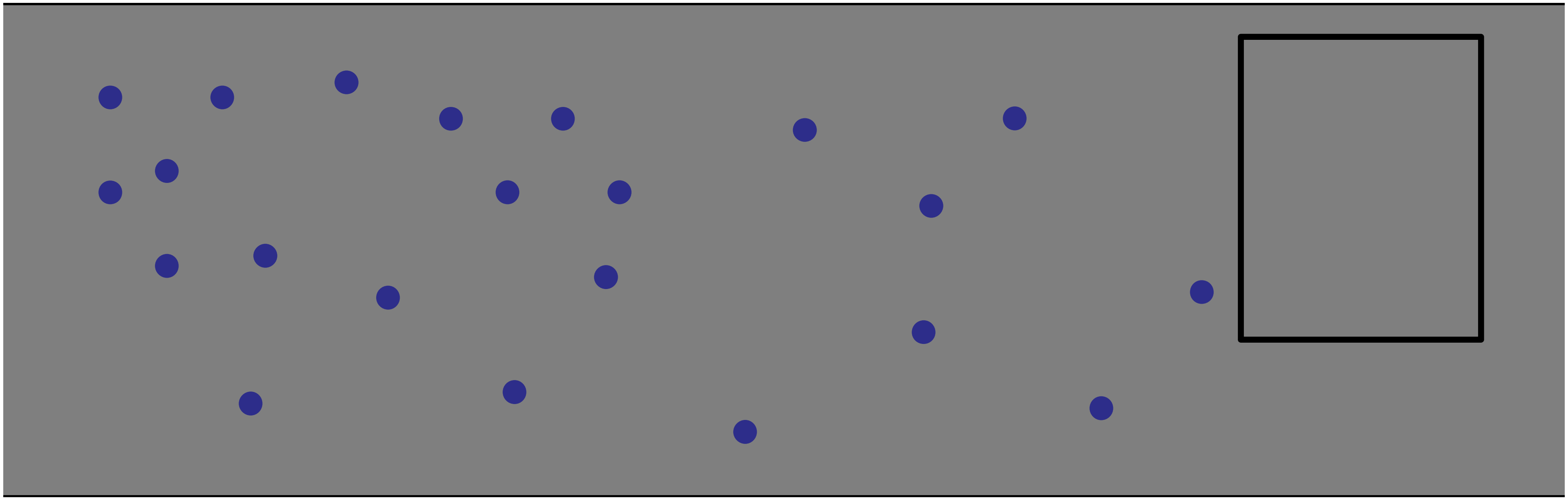


Query: Box

- Contains at least one point?
- How many?

Orthogonal Range Searching

Input: n points in a 2d plane

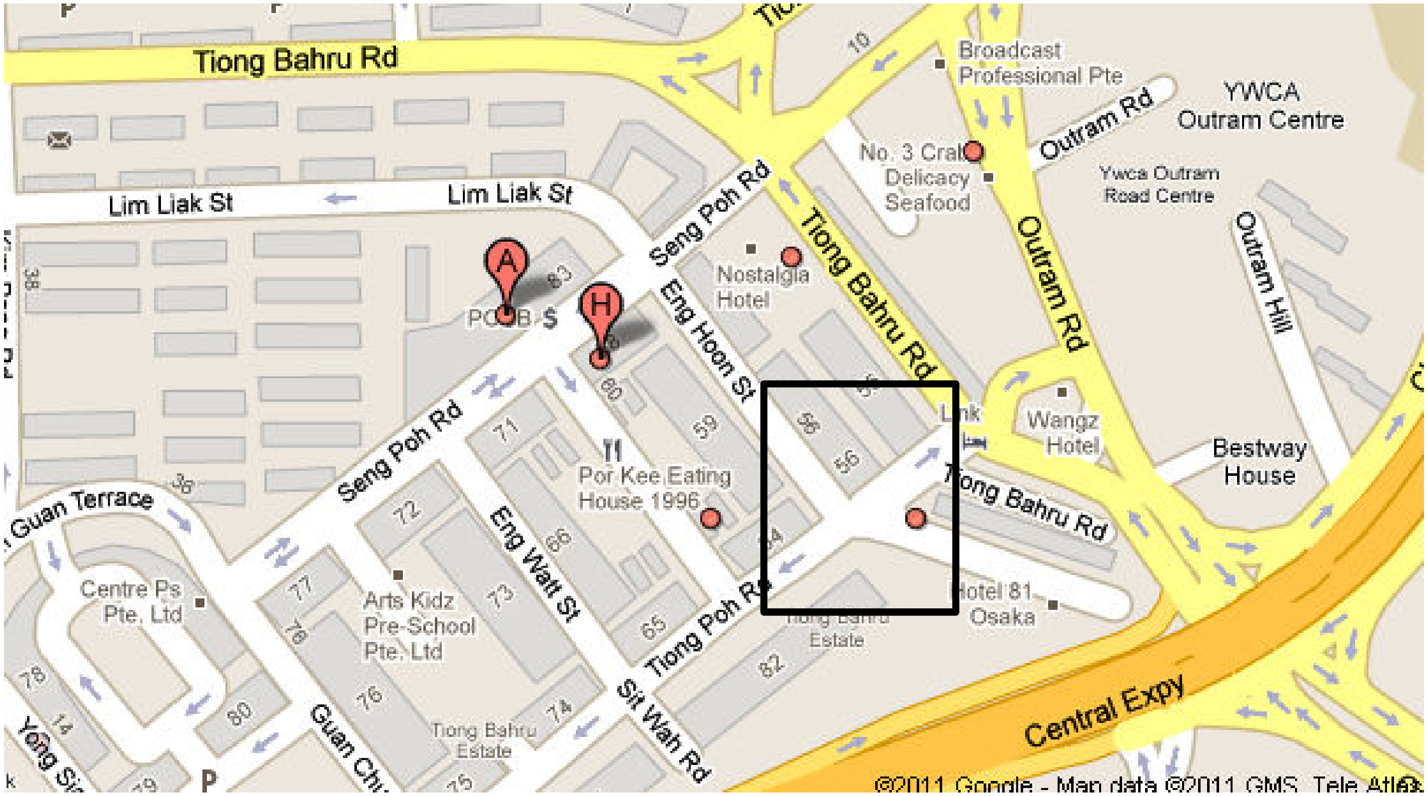


Query: Box

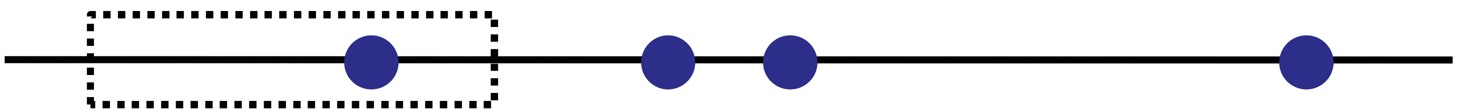
- Contains at least one point?
- How many?

Practical Example

Are there any good restaurants within one block of me?



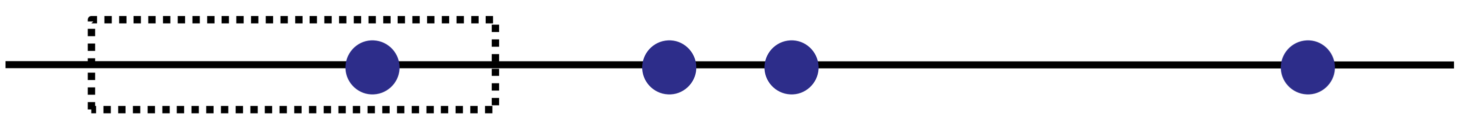
One Dimension



One Dimension

Range Queries

- Important in databases
- Data locality…
- “Find me everyone between ages 22 and 27.”



One Dimension

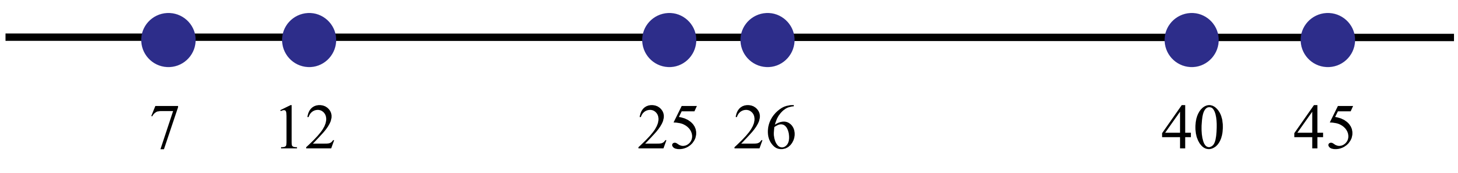
Strategy:

1. Use a binary search tree.
2. Store all points in the leaves of the tree.

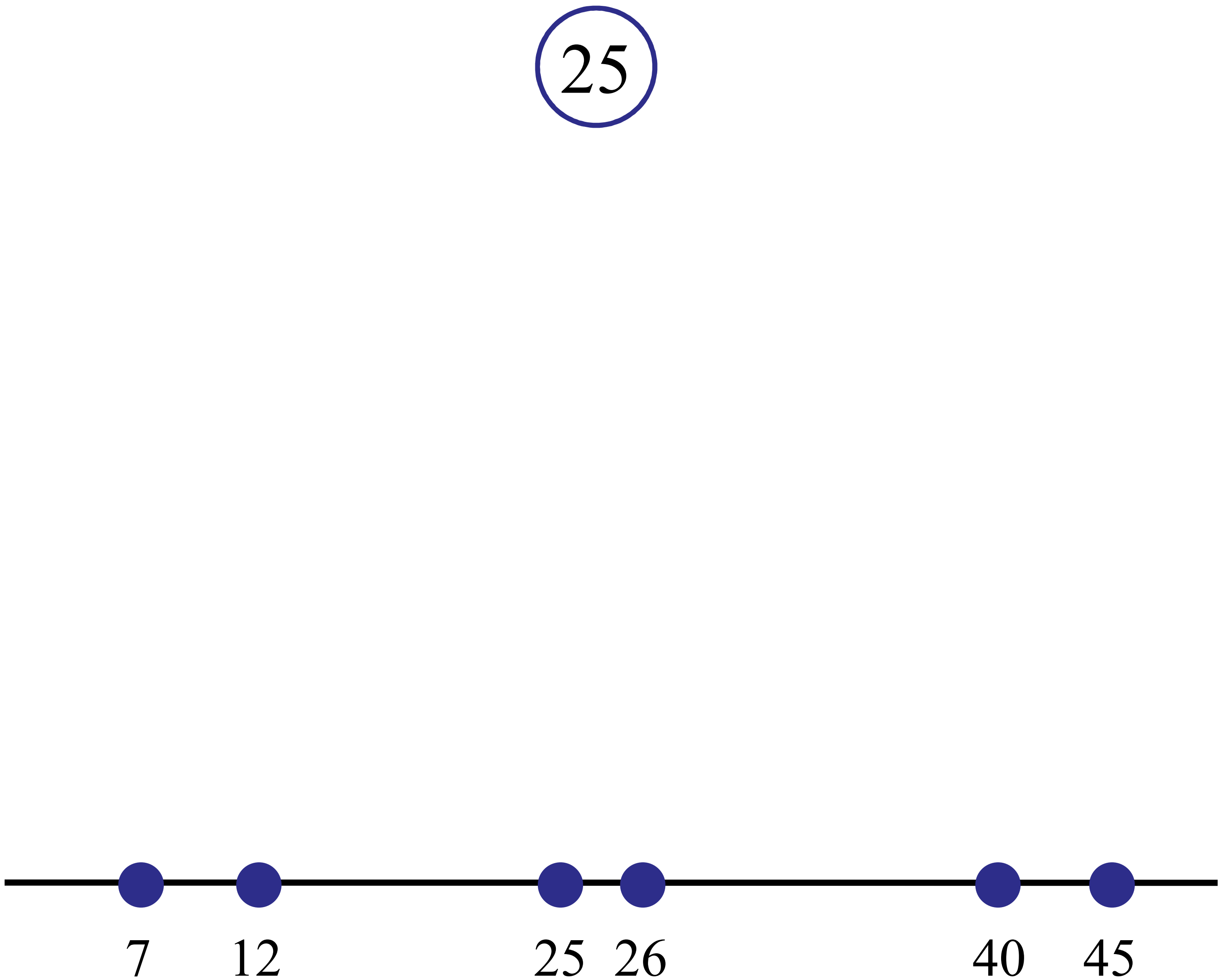
(Internal nodes store only copies.)

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

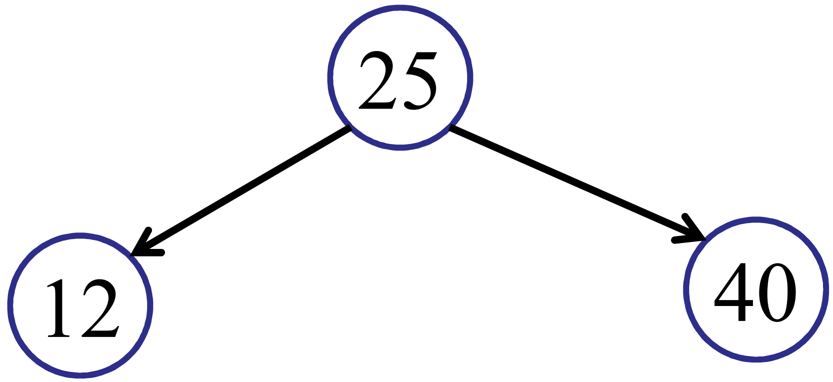
Example

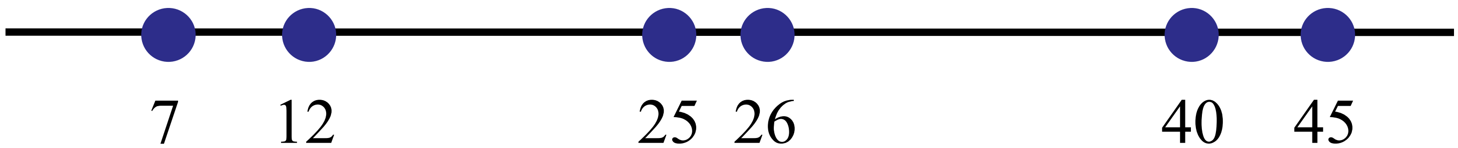


Example

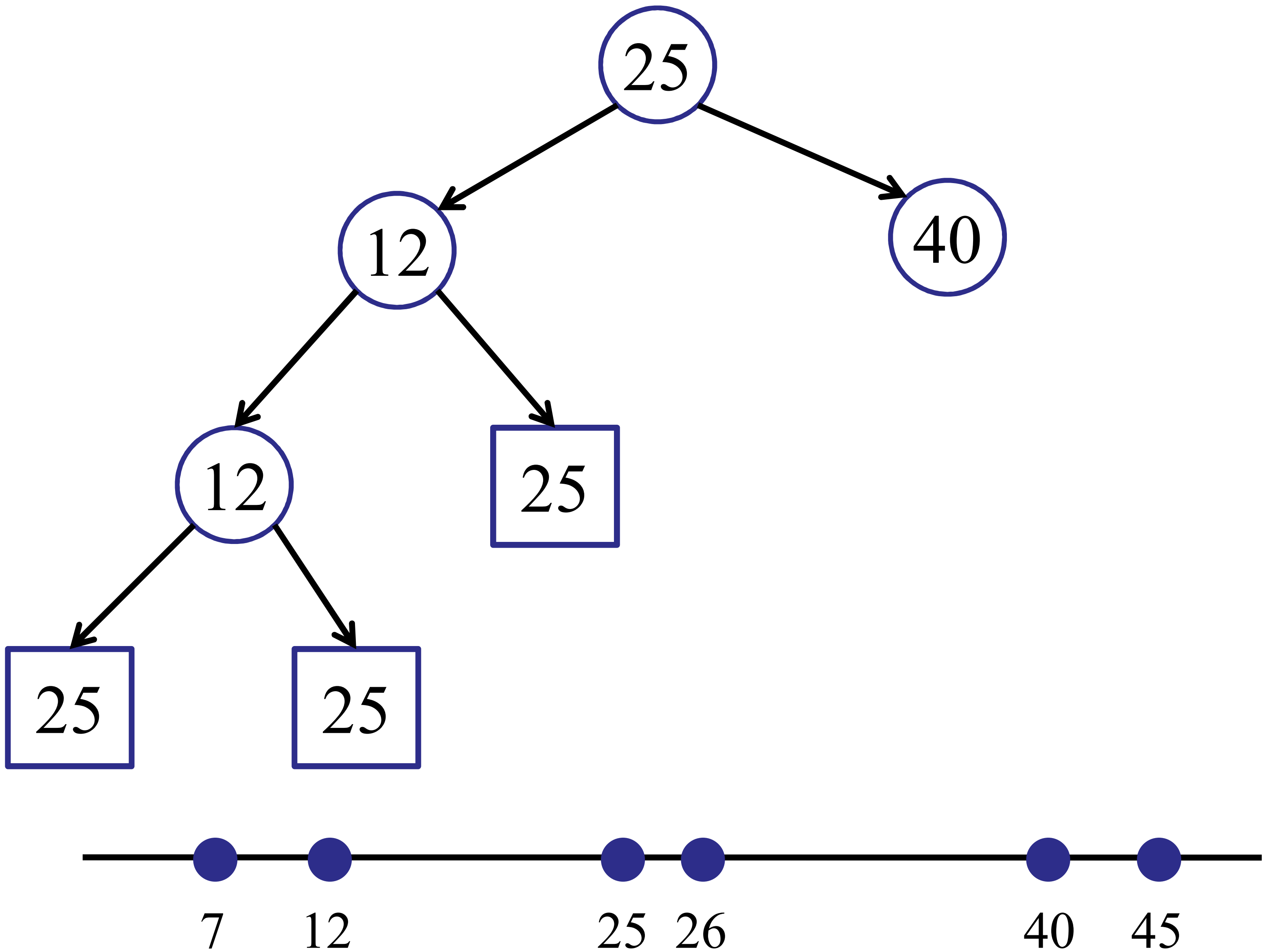


Example

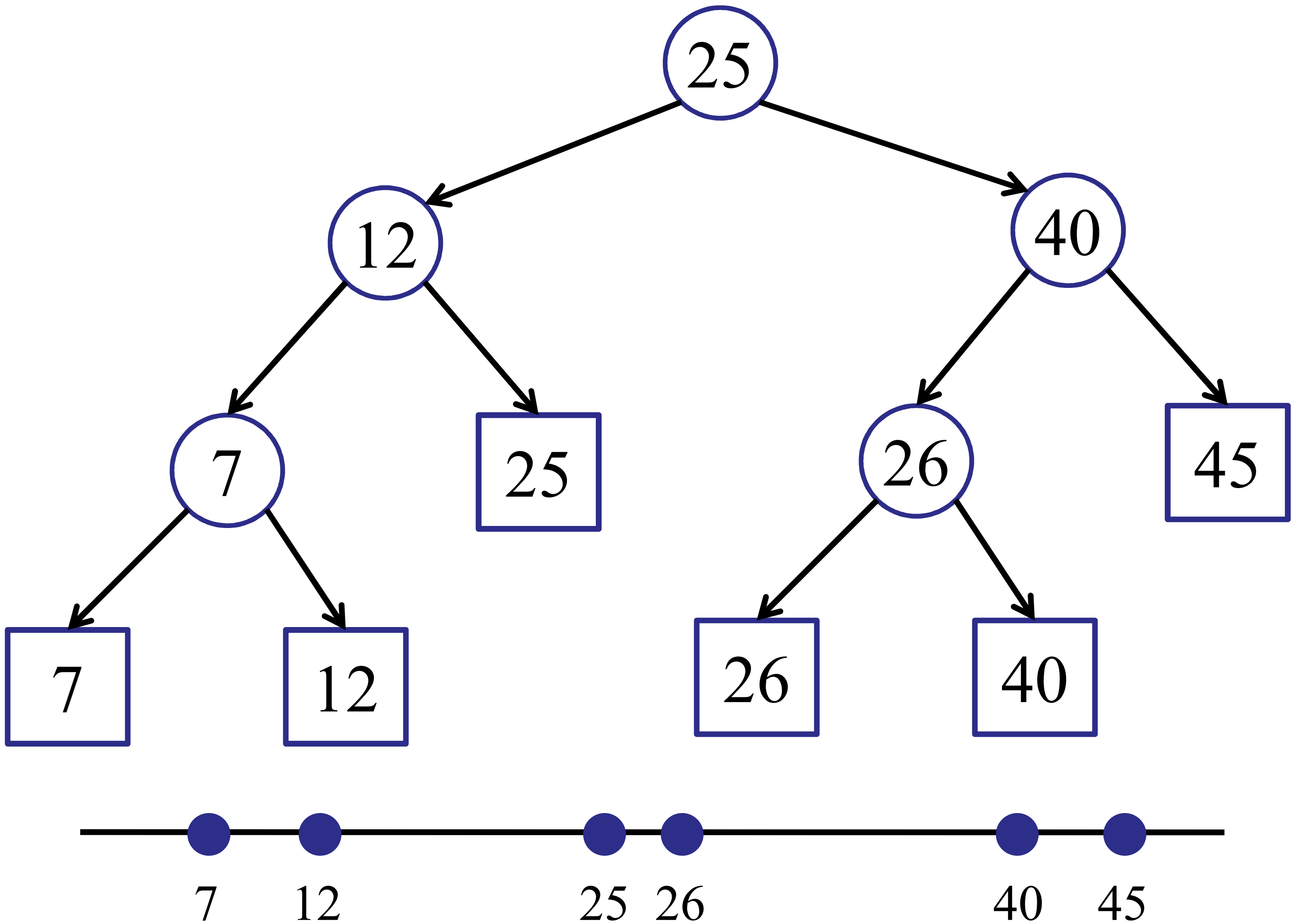




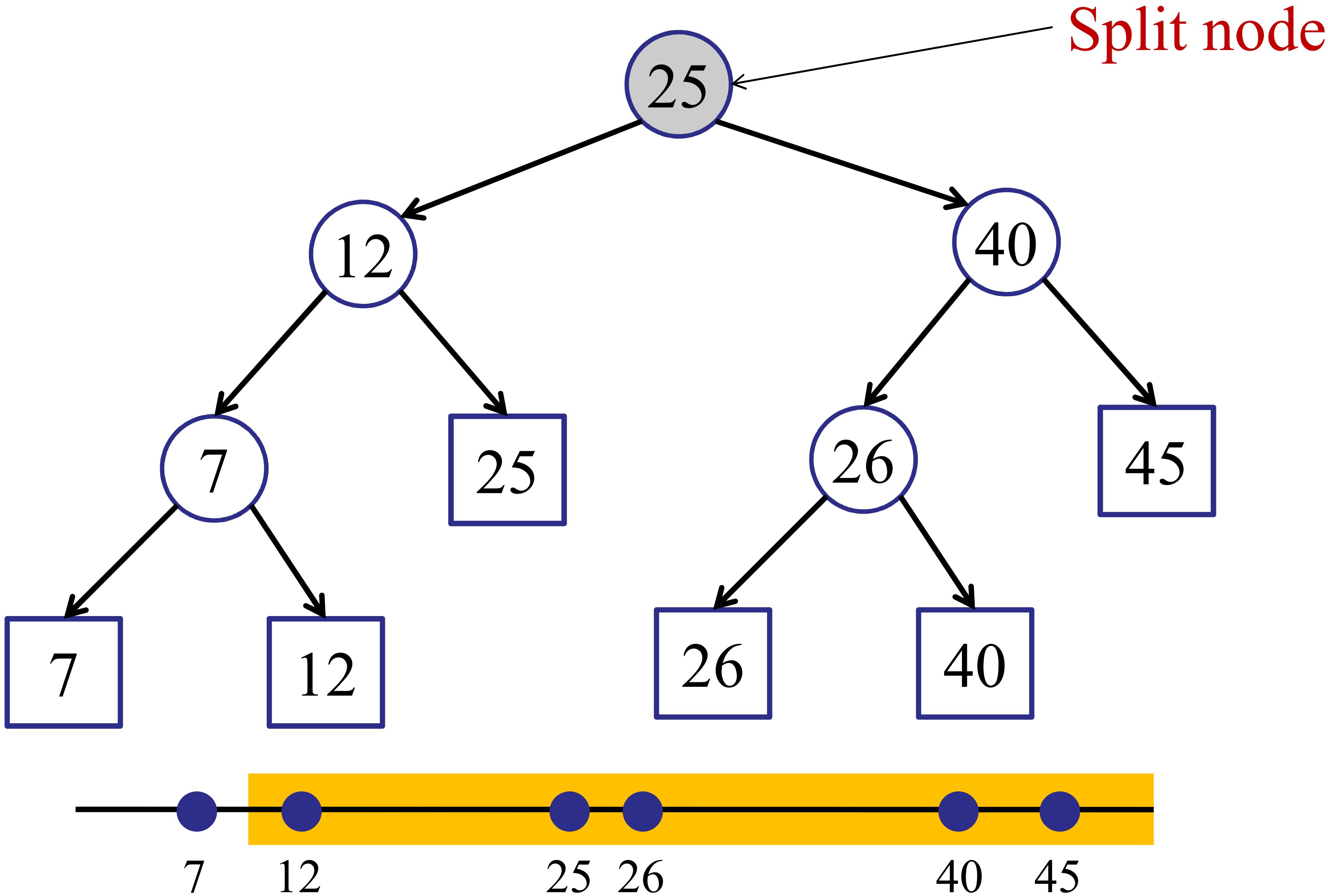
Example



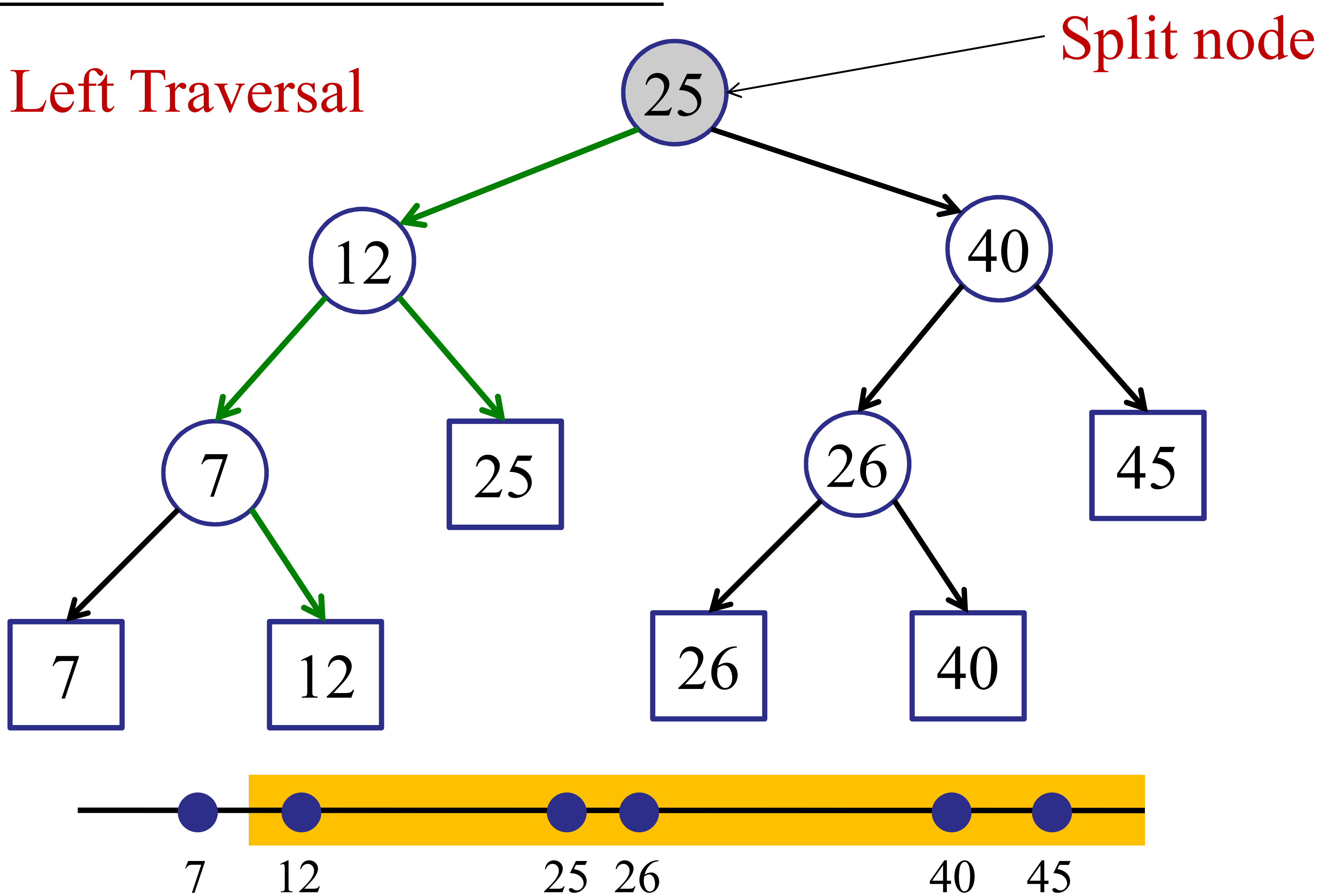
Note: BST Property



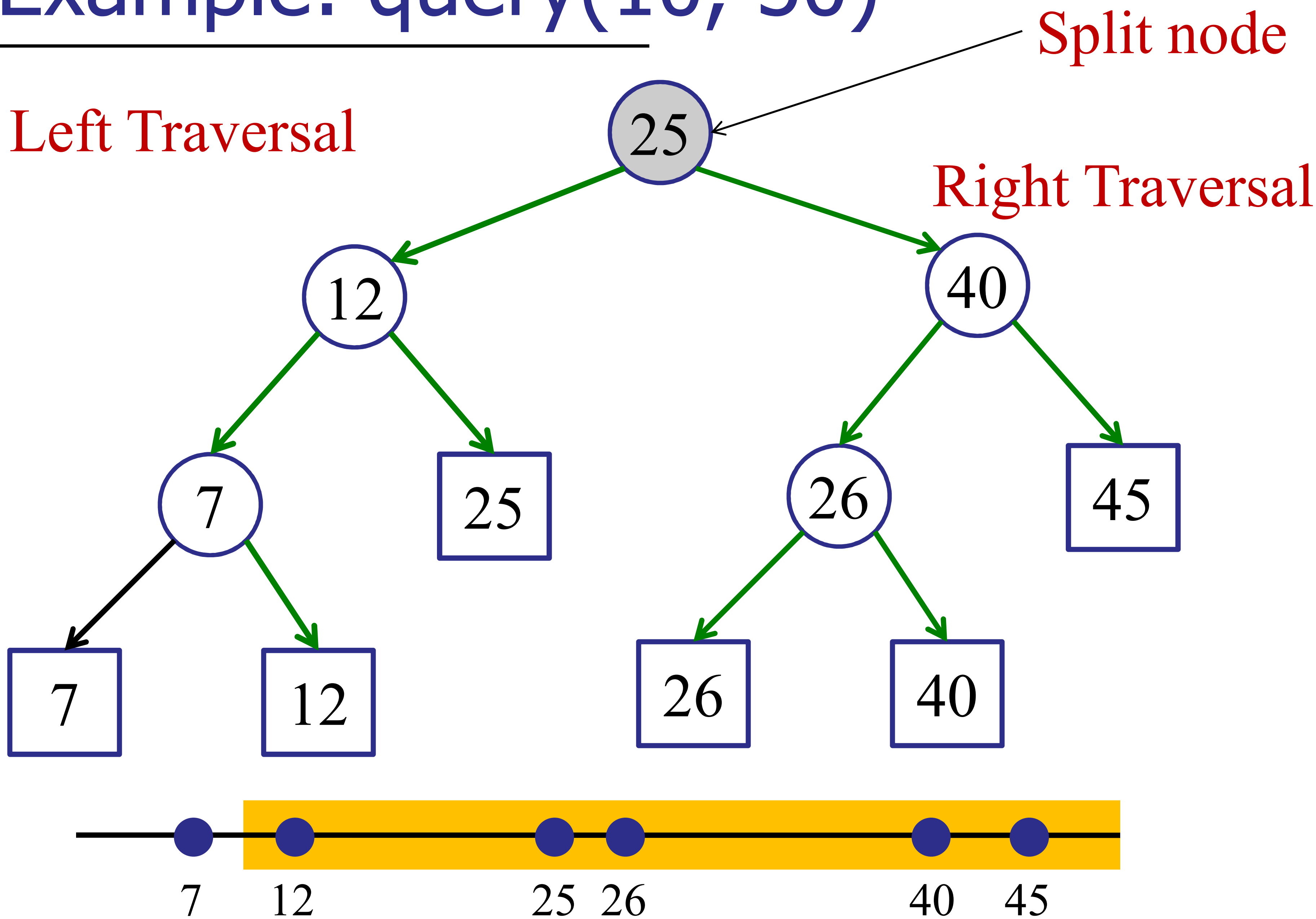
Example: query(10, 50)



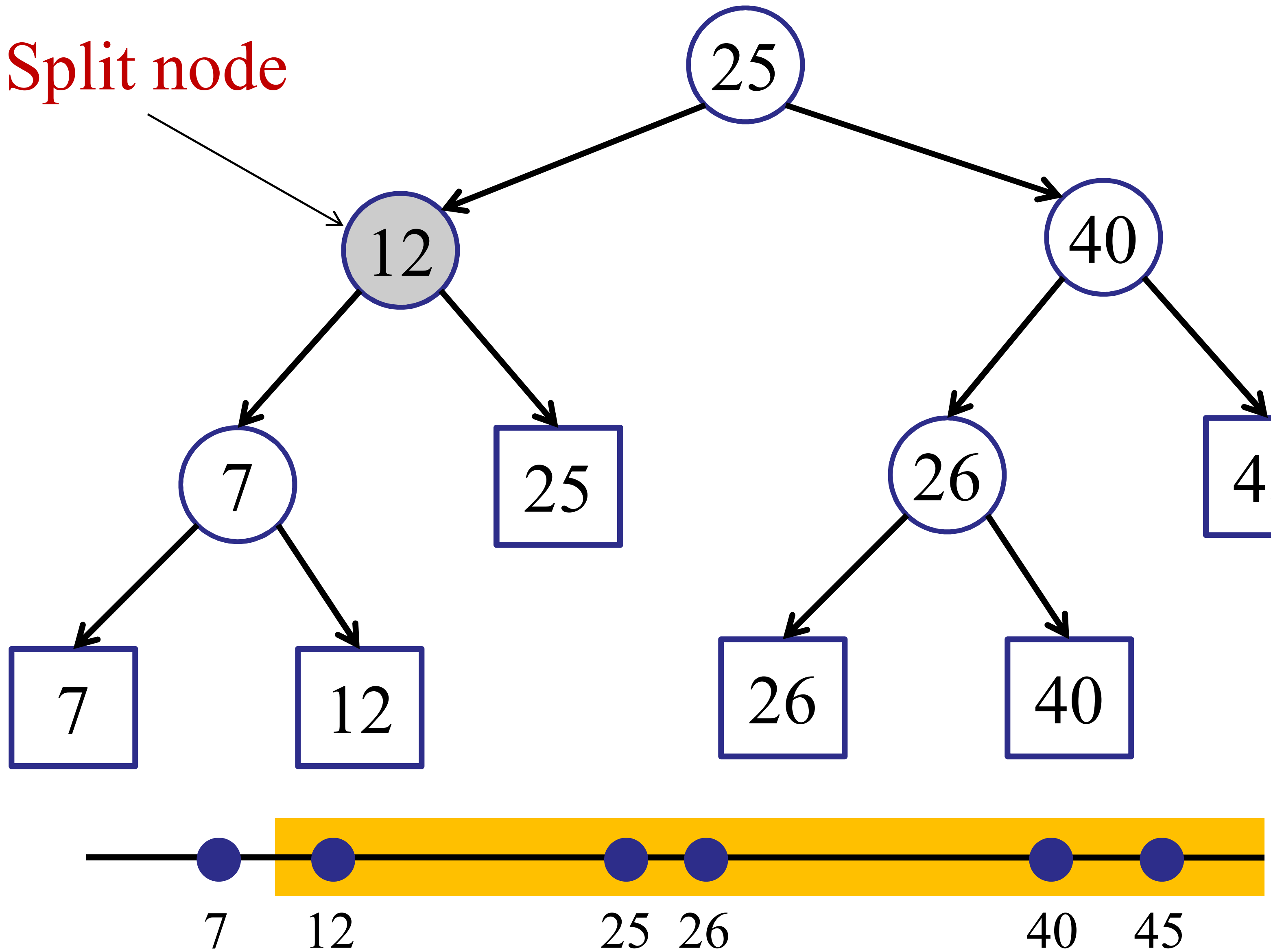
Example: query(10, 50)



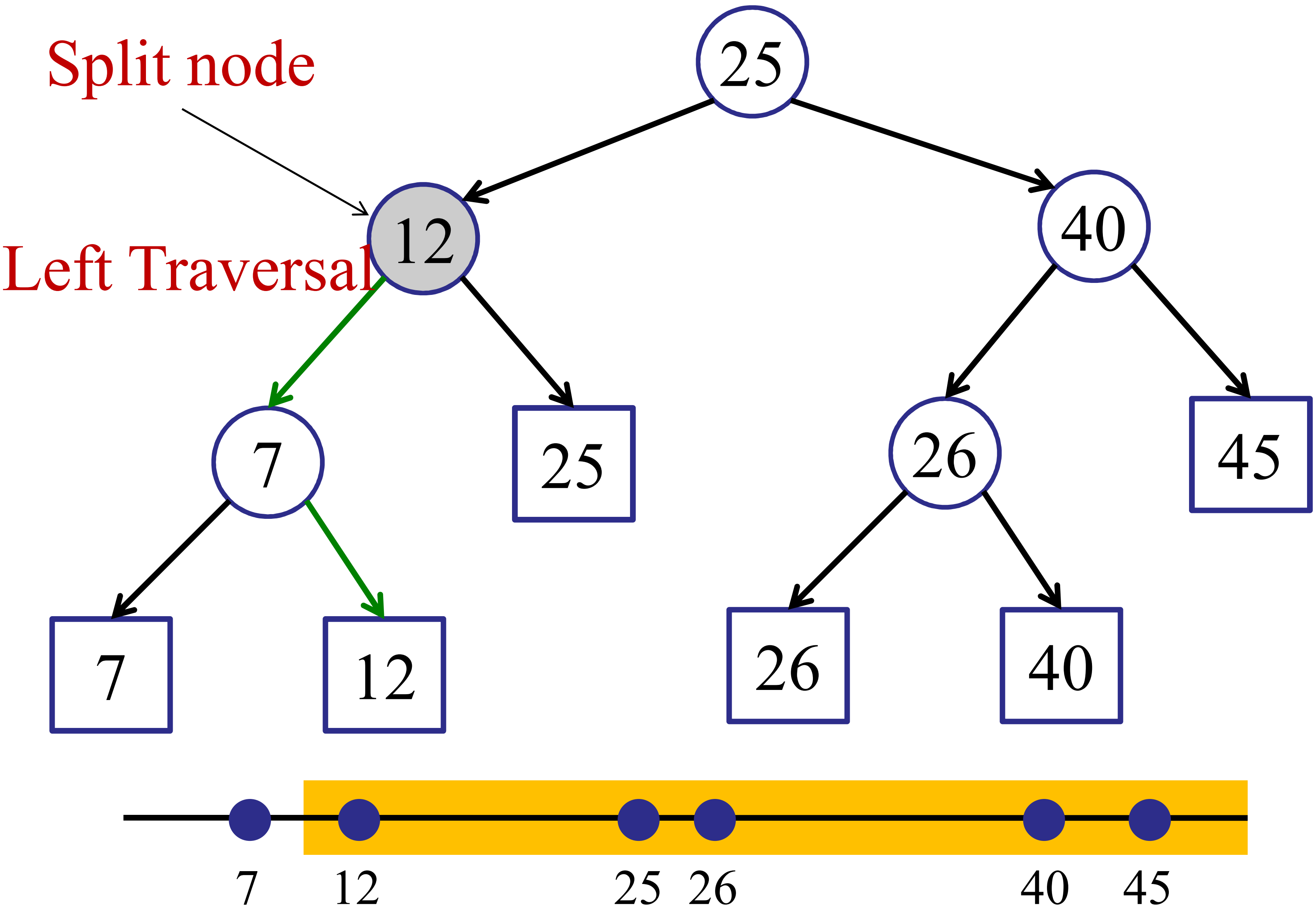
Example: query(10, 50)



Example: query(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;
```

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

One Dimensional Range Queries

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

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:

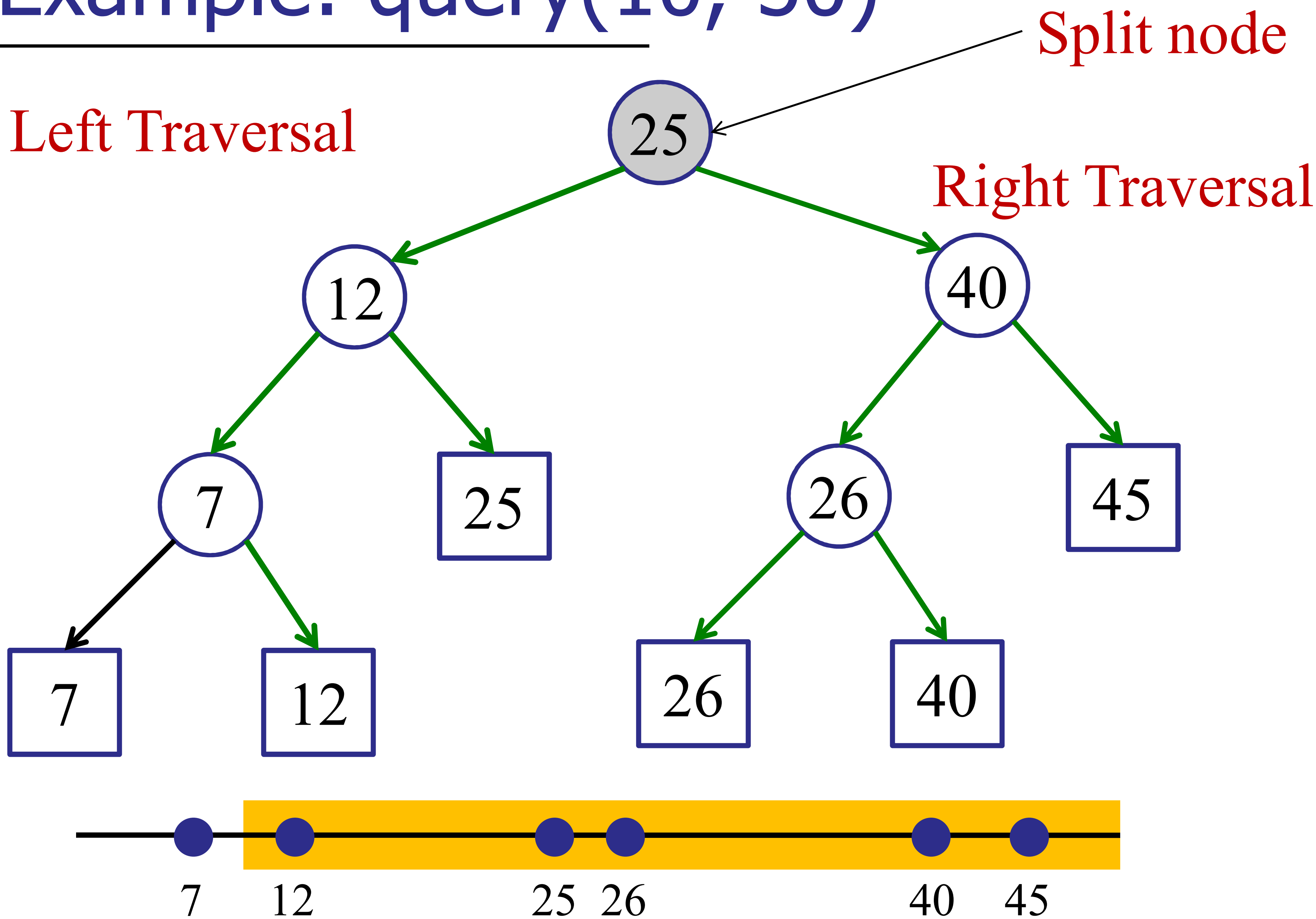
At every step, we either:

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

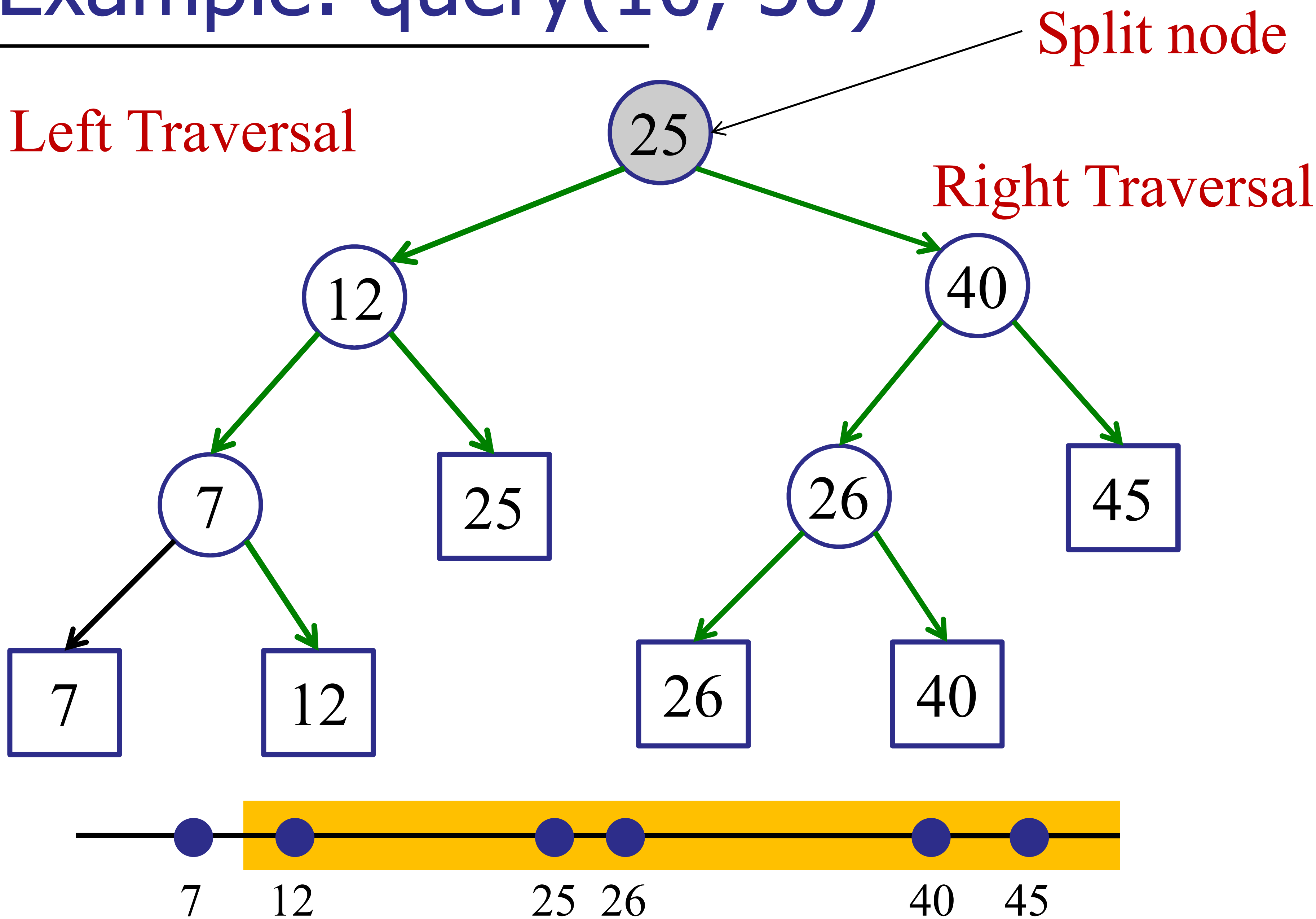
What if you just want to know *how many* points are in the range?

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 count.

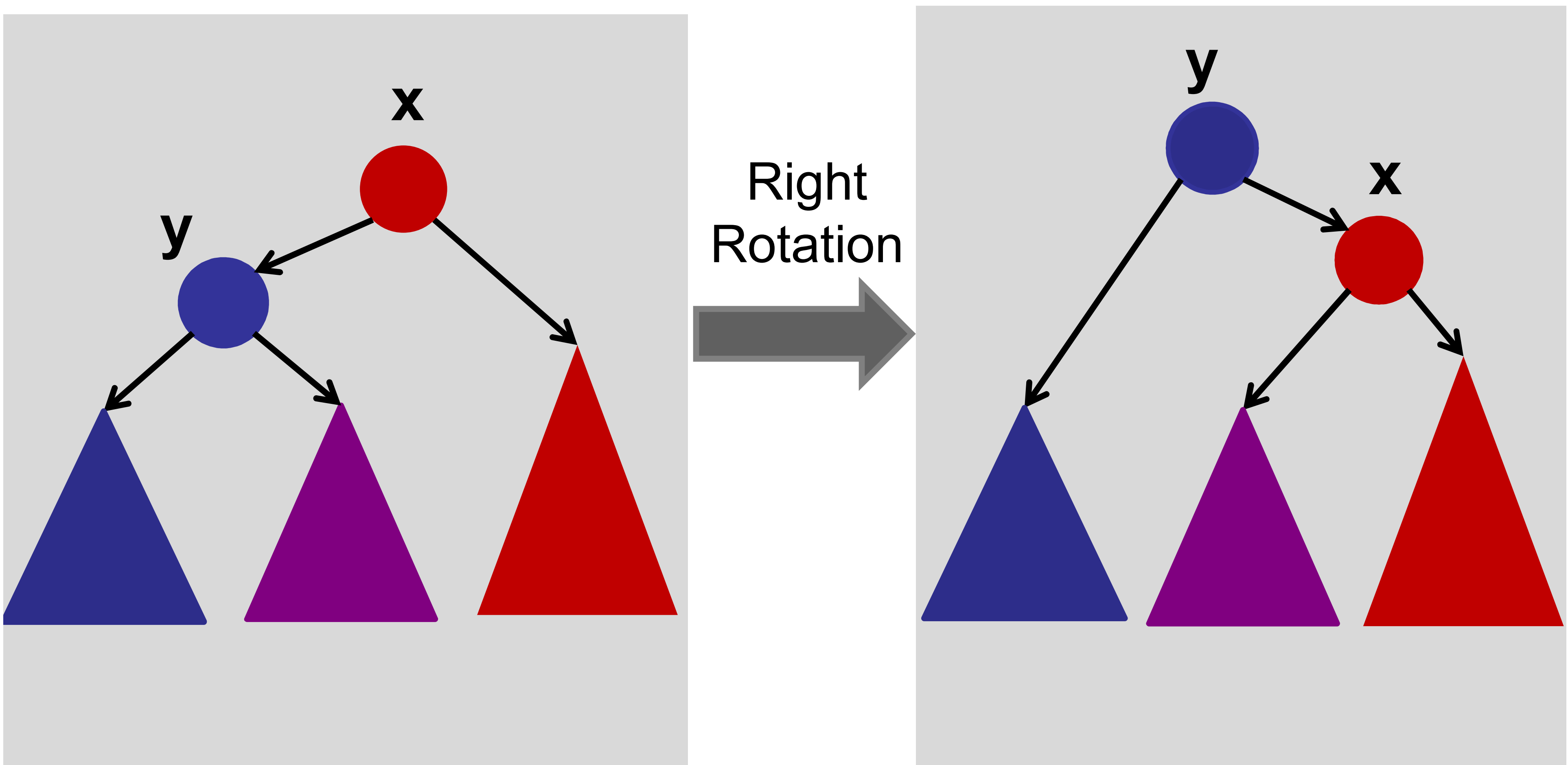
Example: query(10, 50)



One Dimensional Range Queries

What about dynamic updates?

– Need to verify rotations!

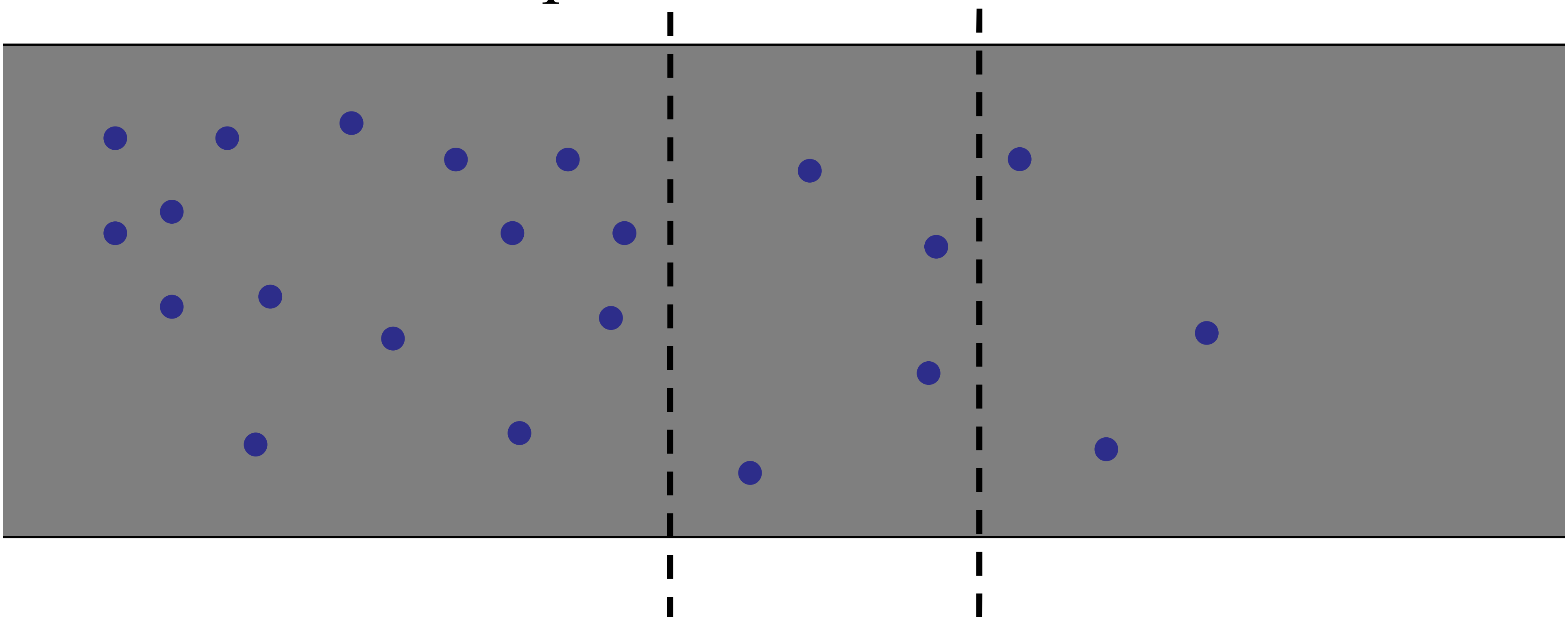


Two Dimensional Range Tree

Step 1:

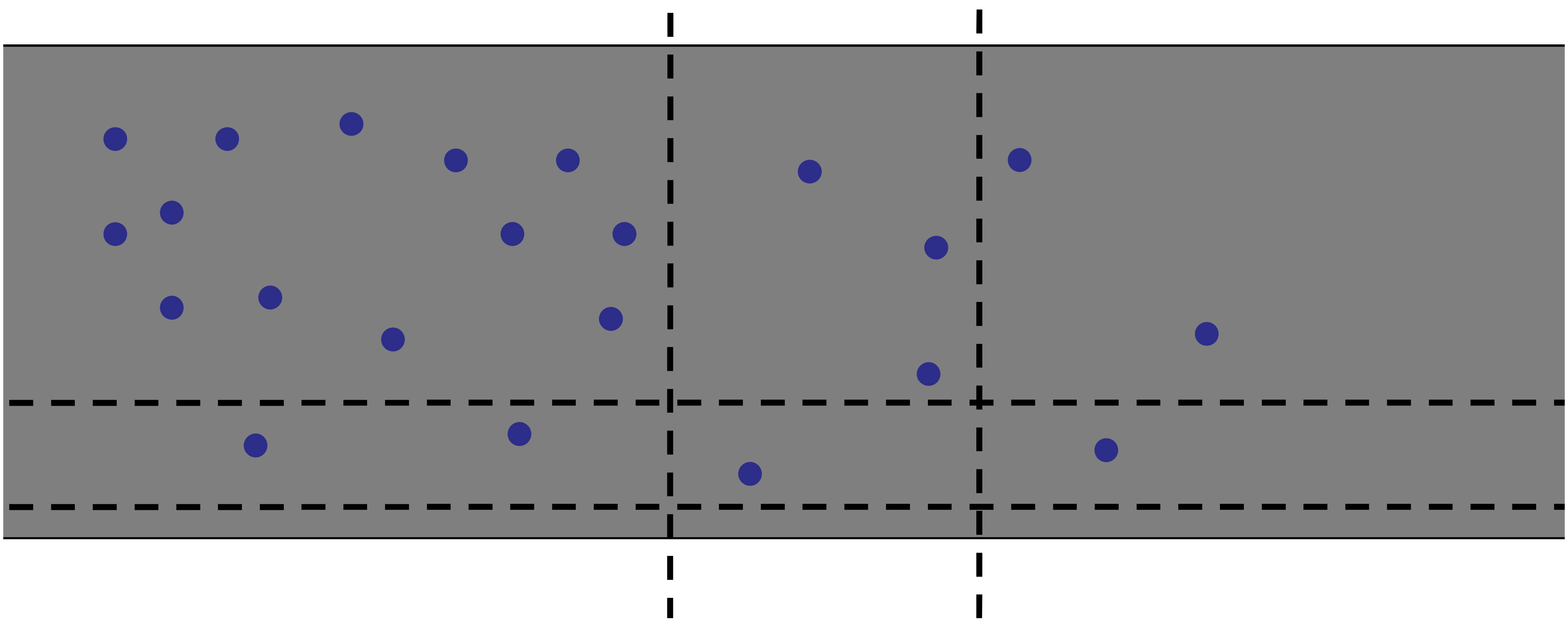
- Create a range-tree on the x-coords.

Ex: search for all points between dashed lines.



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.



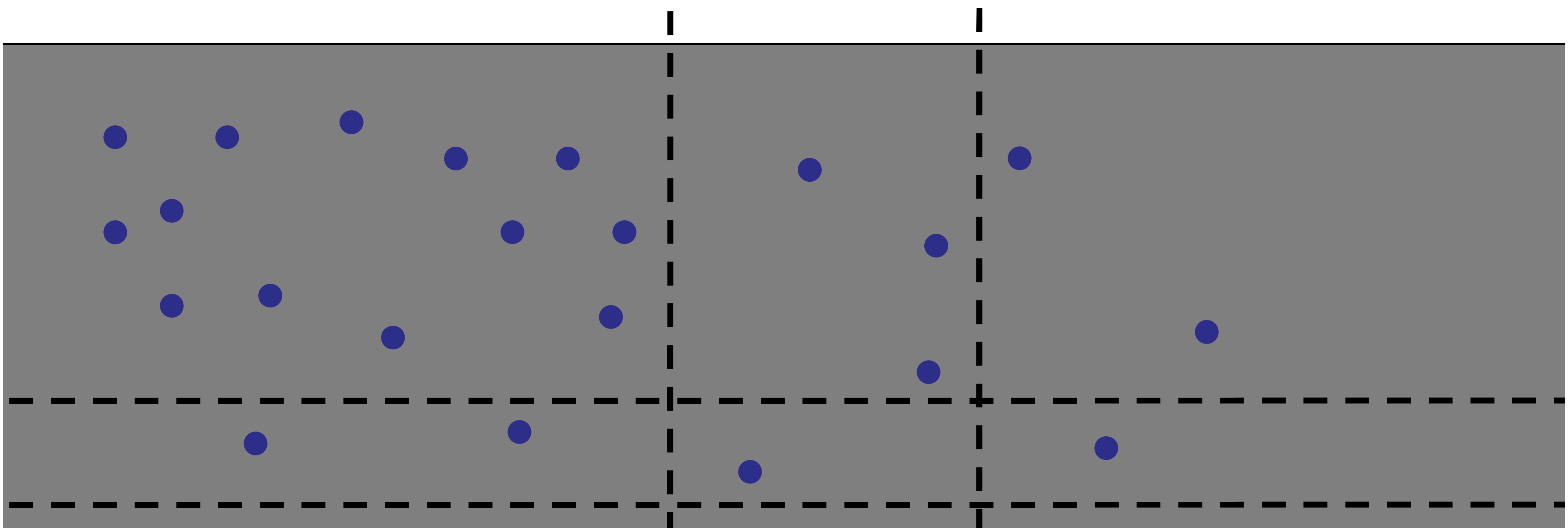
One Dimensional Range Queries

```
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!

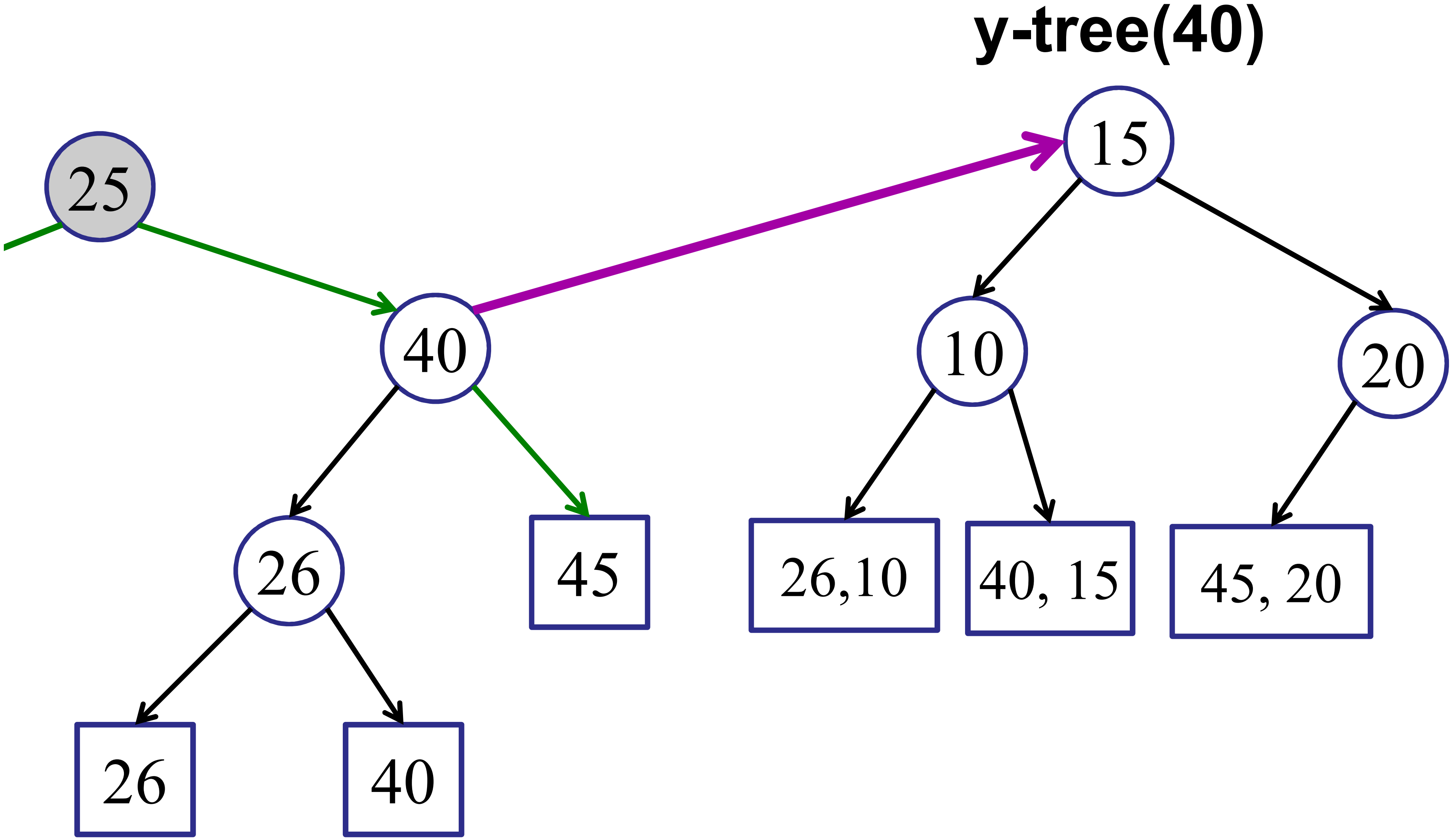
- Each node in the x-tree has a set of points in its sub-tree.
- Store a y-tree at each x-node containing all the points in the sub-tree.



One Dimensional Range Queries

```
LeftTraversal(v, low, high)
    if (v.key >= low) {
        ytree.search(low, high);
        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

Building the tree: O(n log n)

- Tricky…
- Left as a puzzle… ☺

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.

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)!

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.

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