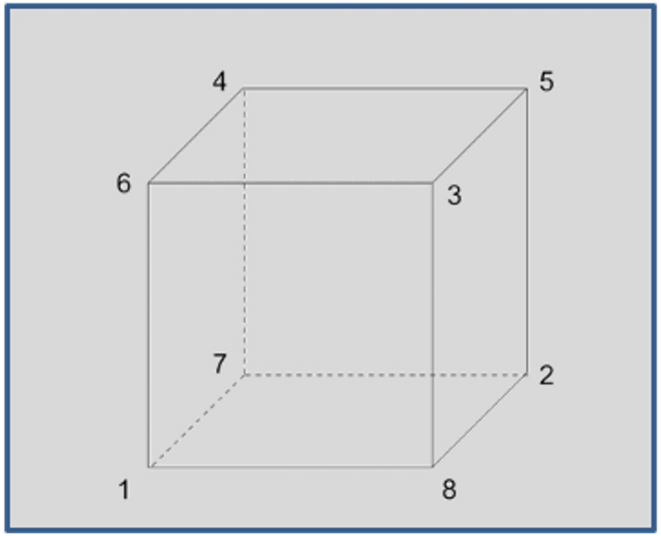
CS2040S
Data Structures and Algorithms

Heaps of trees and trees of heaps…

Example: all FaceSums = 18



Puzzle of the Week: Prime Cubes

A cube has eight vertices.

Assign a prime number to each vertex.

A FaceSum is the sum of the four vertices on a face.

Can you find an assignment of prime numbers so that all six FaceSums are equal?

Plan for the Week

Part I: More Augmented Trees

- Range trees
- 2d Range trees

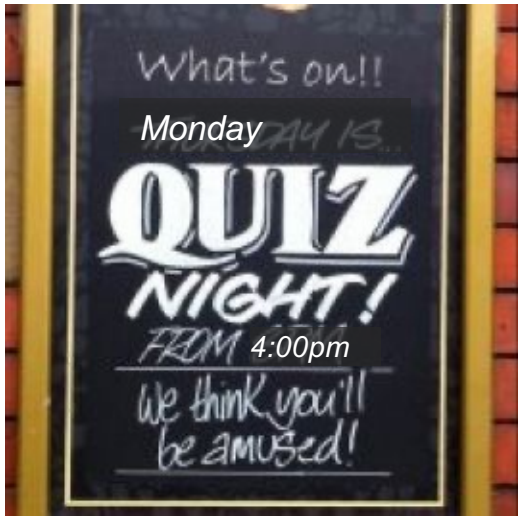
Part II: Priority Queues

- Binary Heaps
- HeapSort

Part III: Disjoint Set

- Problem: Dynamic Connectivity
- Algorithm: Union-Find

Midterm



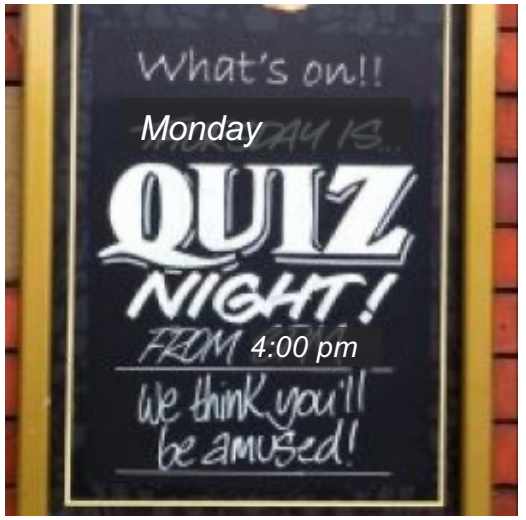
Monday March 6: 4:00pm

- Location: MPSH

Bring to quiz:

- One double-sided sheet of paper with any notes you like.
- Pens/pencils.
- You may not use anything else. (No calculators, no phones, etc.)

Midterm

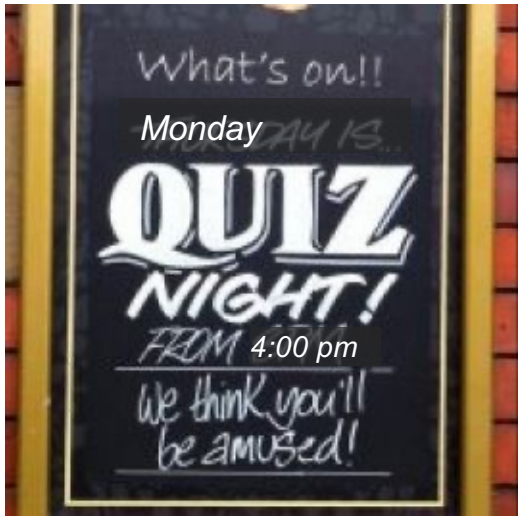


Midterm covers everything through today's lectures and this week's tutorials.

Practice Material:

- Posted on Coursemology
- Beware previous years were not necessarily the same as this year.

Midterm

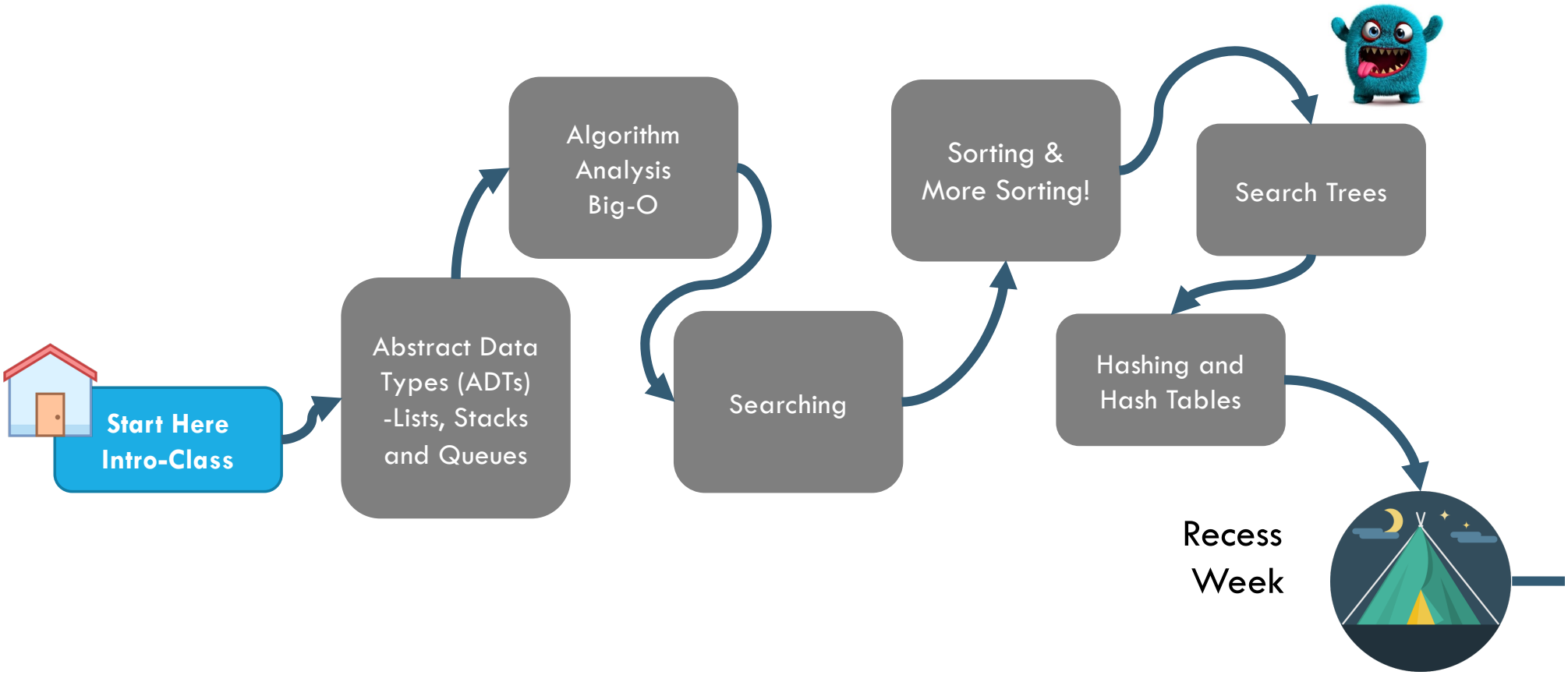


What to do if you are sick?

1. University policy: excused only with MC.
   - Even for Covid, need doctor's certification.
   - Self-testing no longer sufficient.

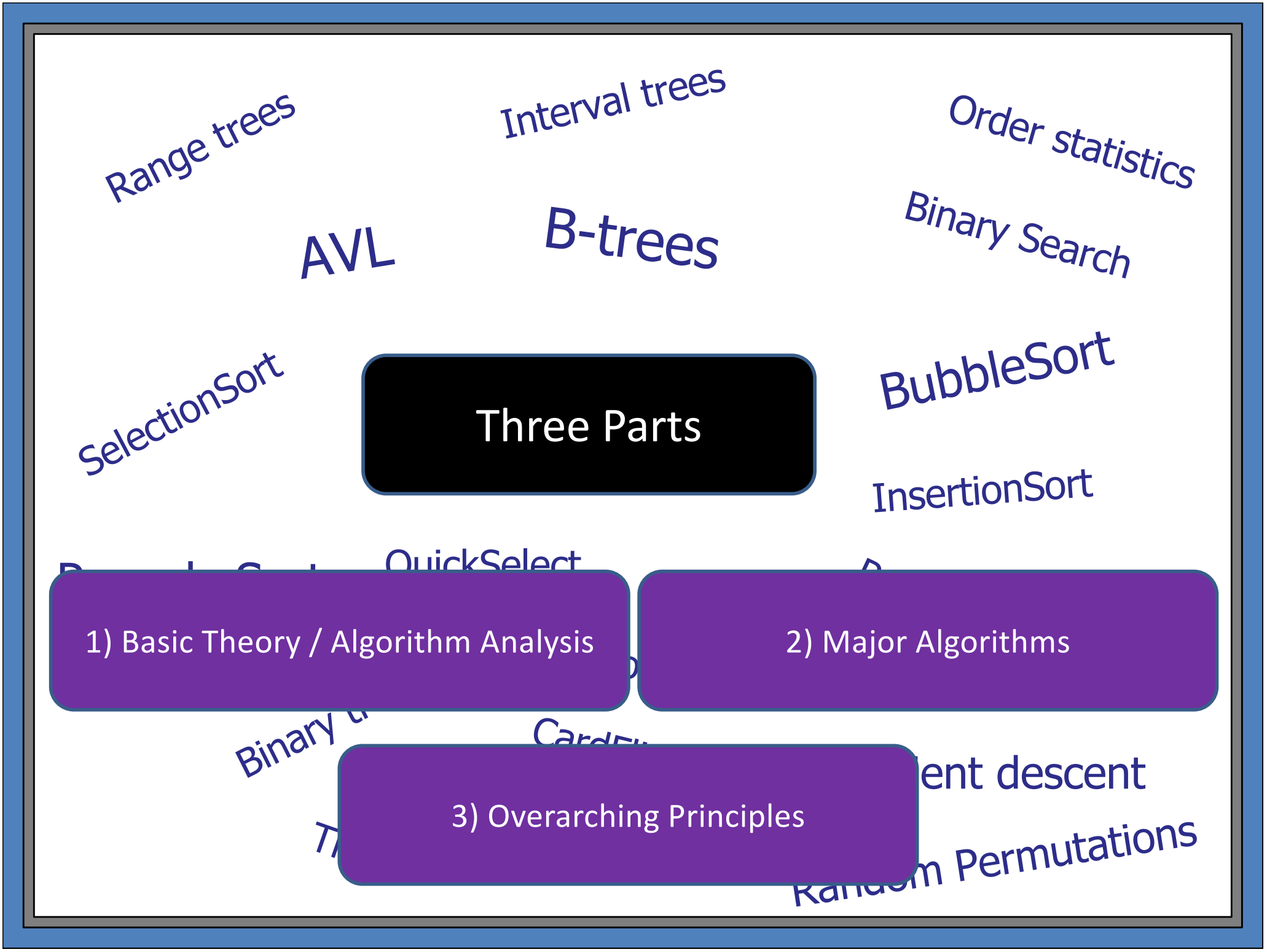
2. PLEASE do not attend the midterm if you are sick.
   - Courtesy to all of us: do not spread illness!

What have we done so far?



Part I: Organizing your Data


Range trees
Interval trees
Order statistics
AVL
B-trees
Binary Search
SelectionSort
So much material?
BubbleSort
InsertionSort
PancakeSort
QuickSelect
QuickSort
ReversalSort
BogoSort
MergeSort
Binary tree
CardFlipTrees
Hash table
Tries
Random Permutations


Range trees
Interval trees
Order statistics
AVL
B-trees
Binary Search
SelectionSort
Three Parts
BubbleSort
InsertionSort
QuickSelect
1) Basic Theory / Algorithm Analysis
2) Major Algorithms
3) Overarching Principles
ent descent
Random Permutations

Basic Theory: Key Topics

- Asymptotic notation (big-O, etc.)
- Simple recurrences
- Asymptotic analysis
- Basic probability (e.g., CS1231 review)

Key Algorithms

- Binary Search
- Sorting
- Balanced binary trees
- Augmented trees
- Heaps (today)

Key Ideas

- Basic strategies for solving problems
- Invariants
- Trade-offs: how to choose which data structure
- Augmenting data structures

Basic strategies for solving problems

Try something simple

E.g., naïve search

Reduce-and-conquer

E.g., binary search

Divide-and-conquer

E.g., MergeSort

Maintaining an invariant

E.g., AVL trees

E.g., keep your data sorted

Augment an existing data structure

E.g., AVL trees

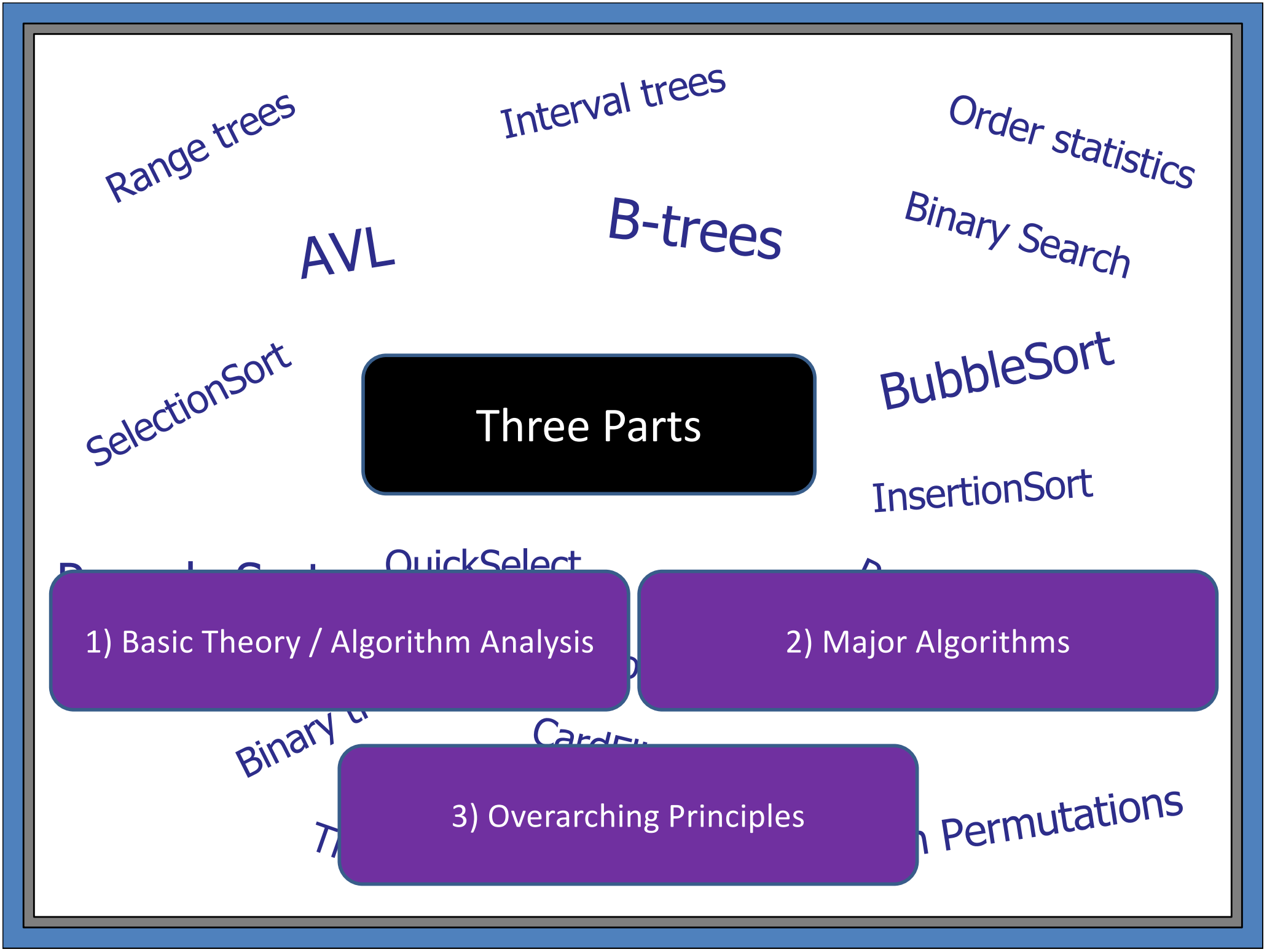
Two important questions:

How do you understand what an algorithm is doing?

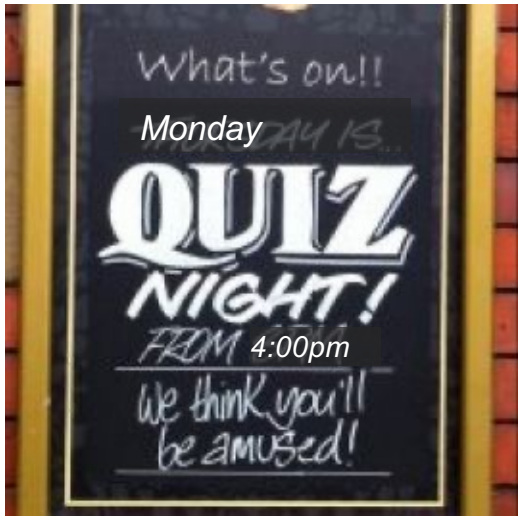
How do you show that an algorithm is correct?

To understand algorithms:

For every algorithm in the class, identify all of its important invariants.


Range trees
Interval trees
Order statistics
AVL
B-trees
Binary Search
SelectionSort
Three Parts
BubbleSort
InsertionSort
QuickSelect
1) Basic Theory / Algorithm Analysis
2) Major Algorithms
3) Overarching Principles
Permutations

Midterm



Monday March 6: 4:00pm

- Location: MPSH

Bring to quiz:

- One double-sided sheet of paper with any notes you like.
- Pens/pencils.
- You may not use anything else. (No calculators, no phones, etc.)

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.

Dynamic Data Structures

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

Augmented Data Structures

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

Augment more frequently than invent...

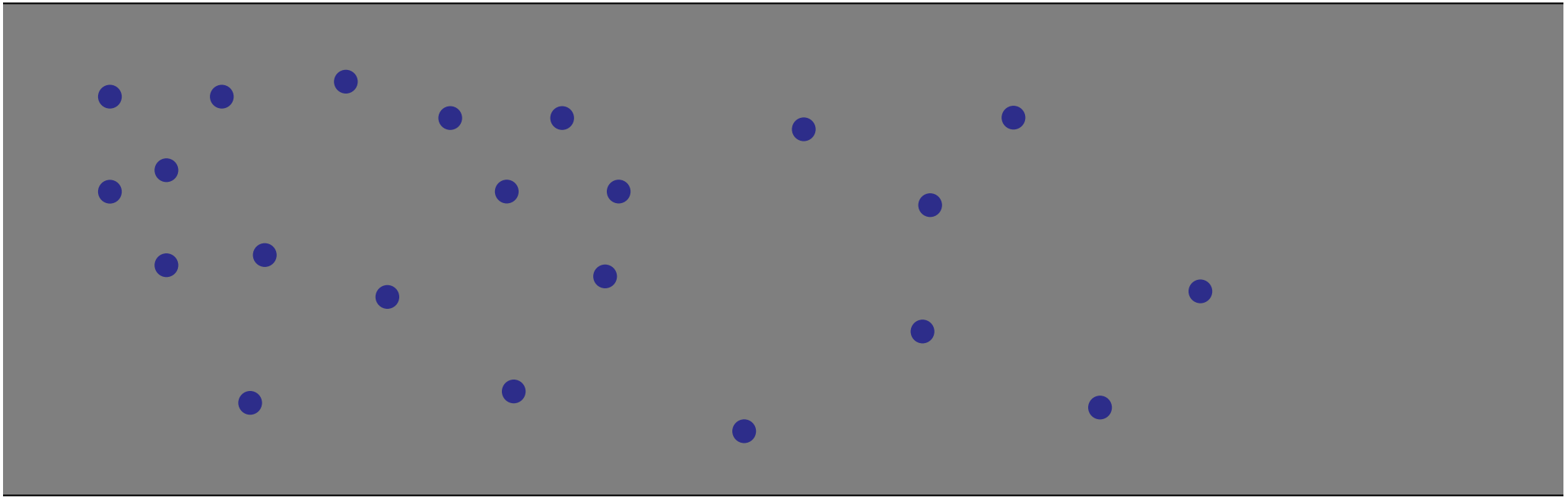
Plan

Three examples of augmenting balanced BSTs

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

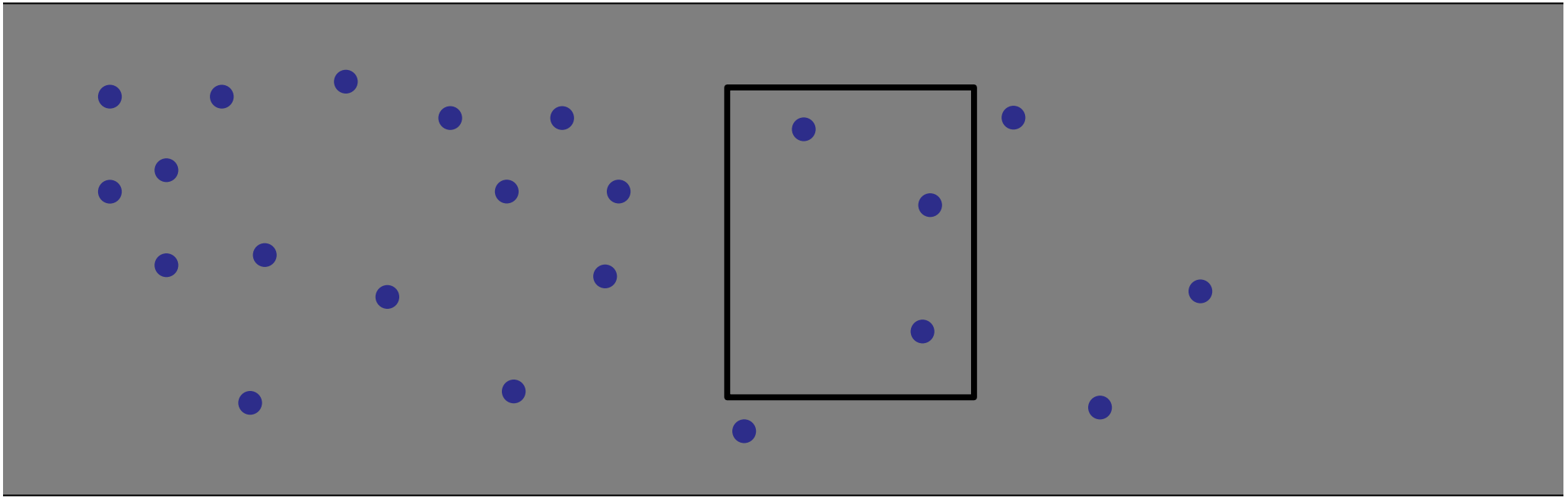
Orthogonal Range Searching

Input: n points in a 2d plane



Orthogonal Range Searching

Input: n points in a 2d plane

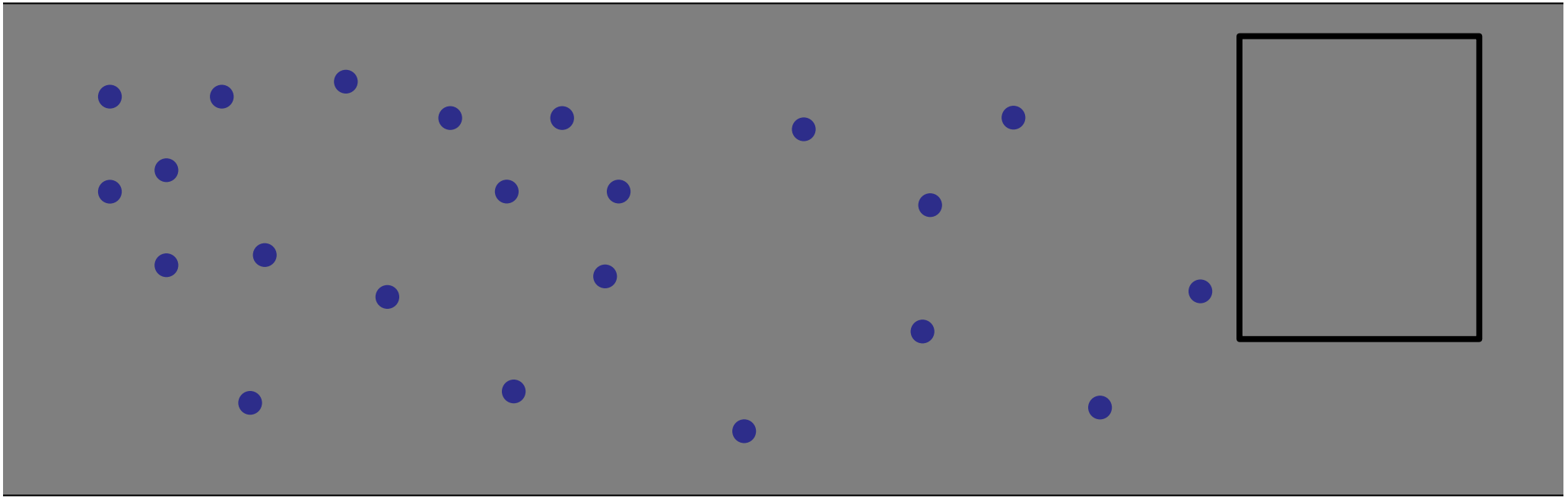


Query: Box

- Contains at least one point?
- How many?

Orthogonal Range Searching

Input: n points in a 2d plane

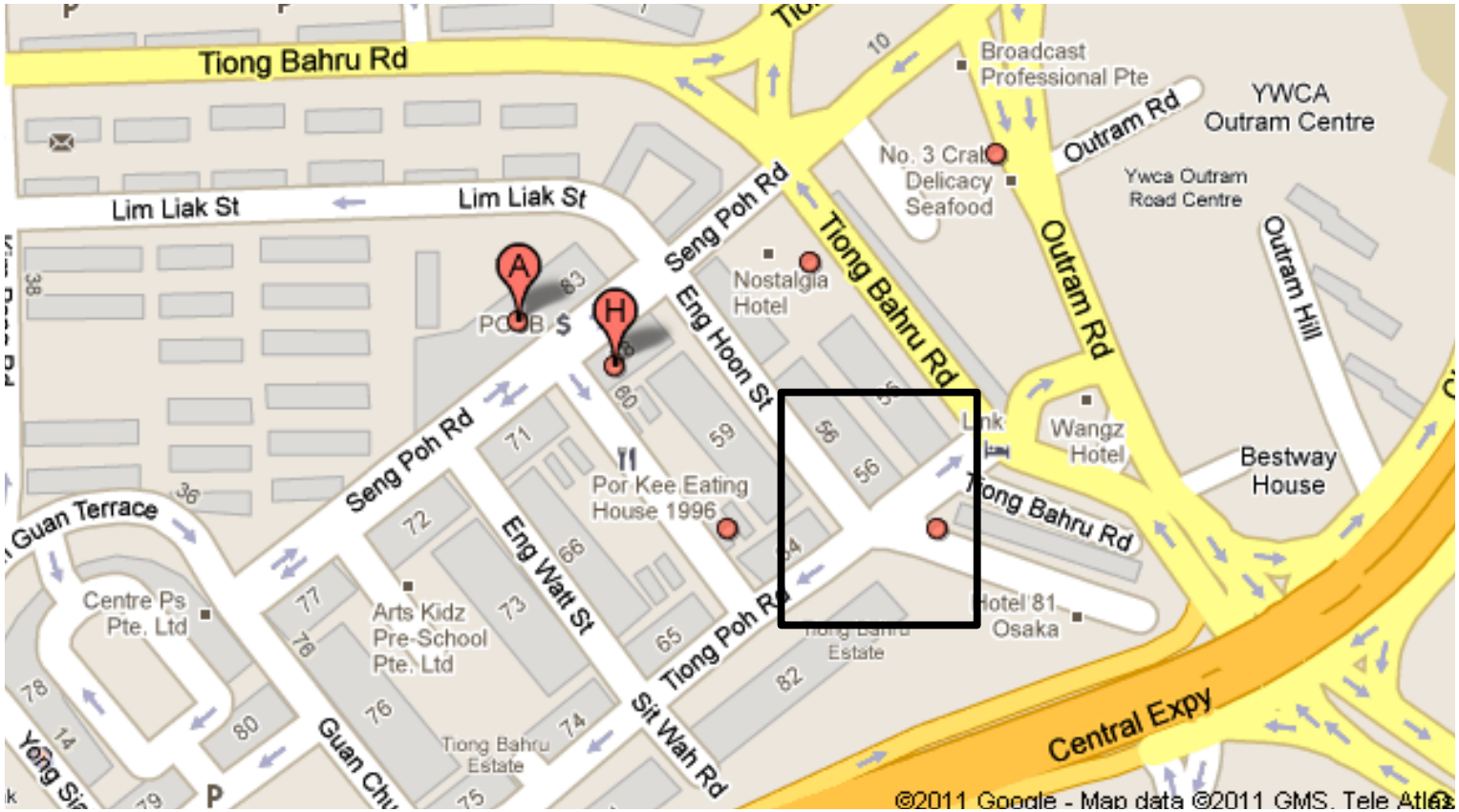


Query: Box

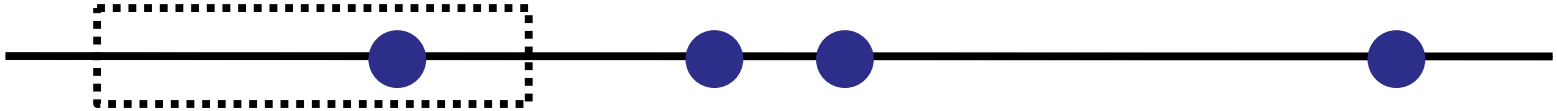
- Contains at least one point?
- How many?

Practical Example

Are there any good restaurants within one block of me?



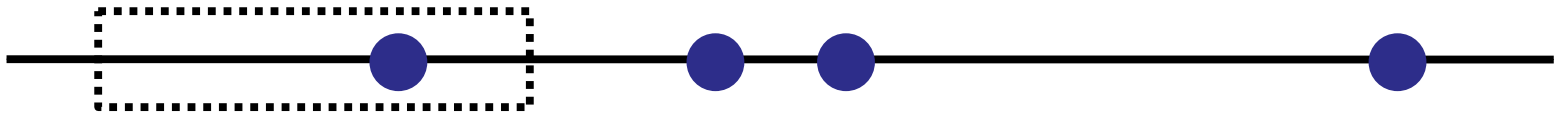
One Dimension



One Dimension

Range Queries

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



One Dimension

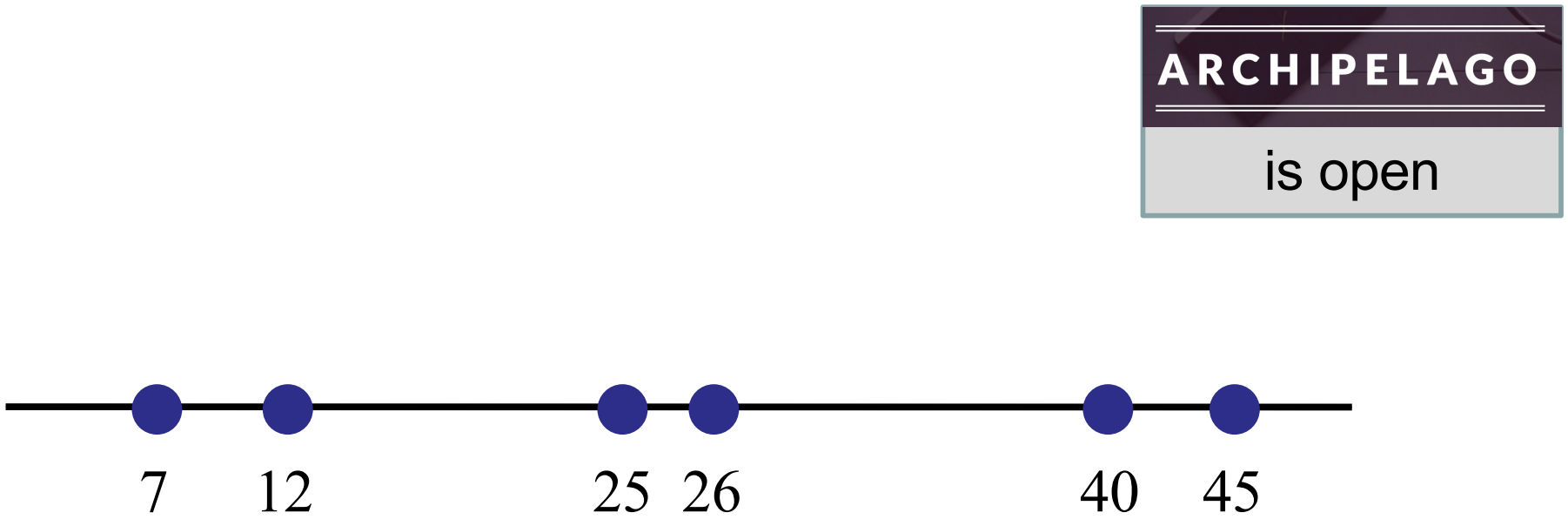
Strategy:

1. Use a binary search tree.

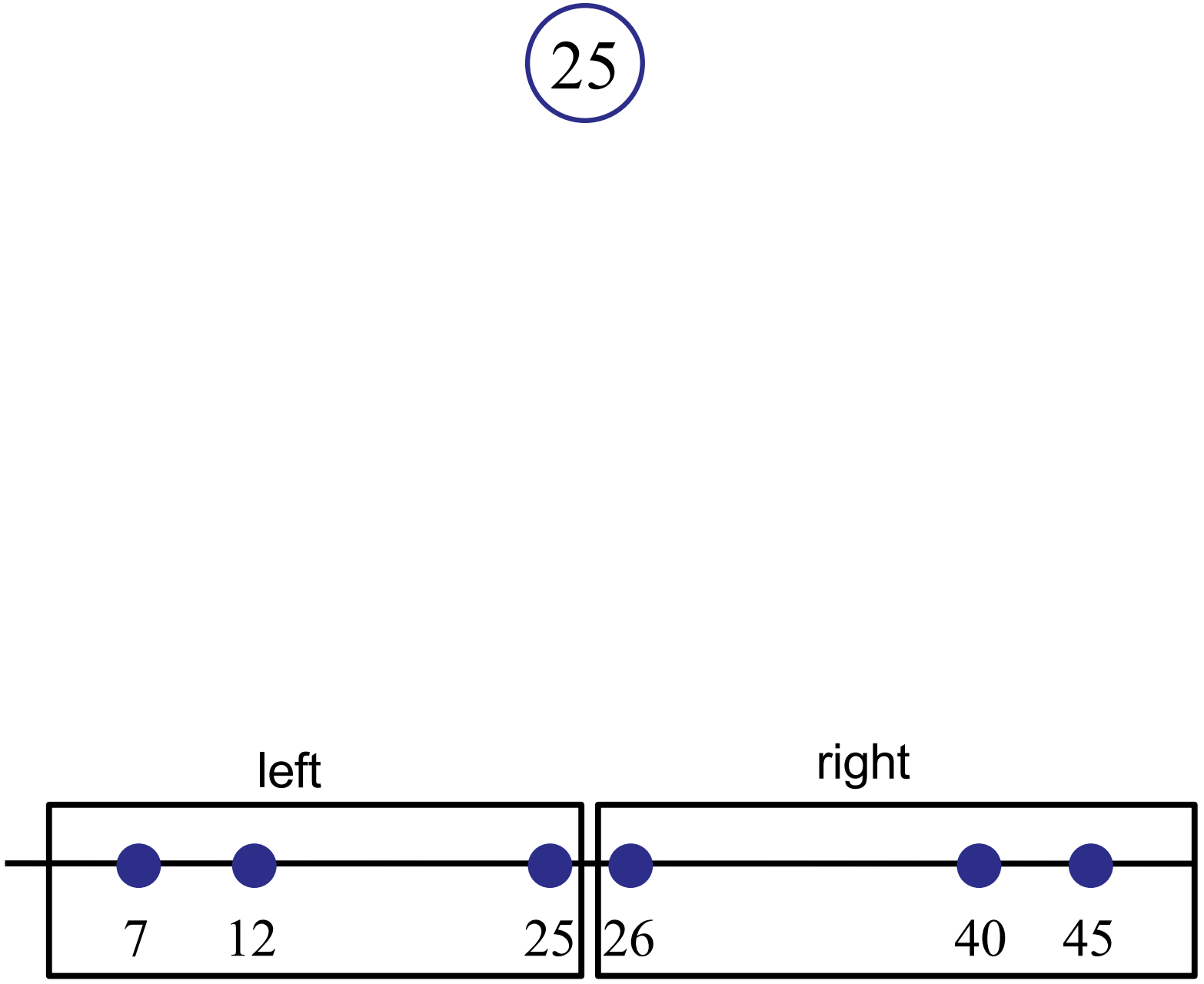
2. Store all points in the leaves of the tree.
   (Internal nodes store only copies.)

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

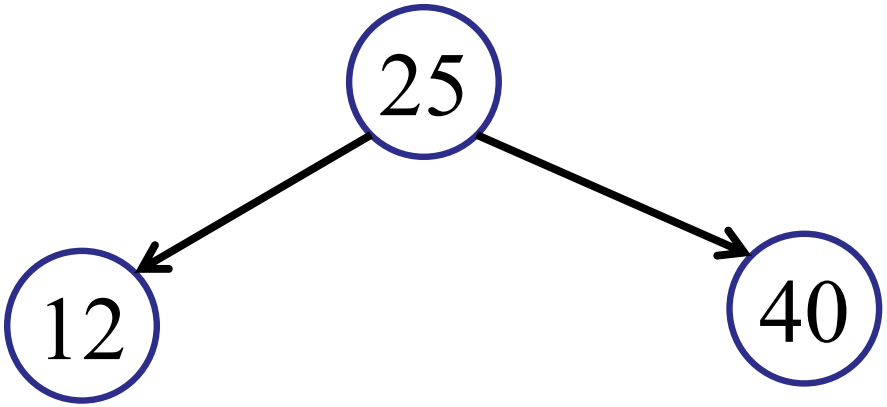
Example: what is the root?

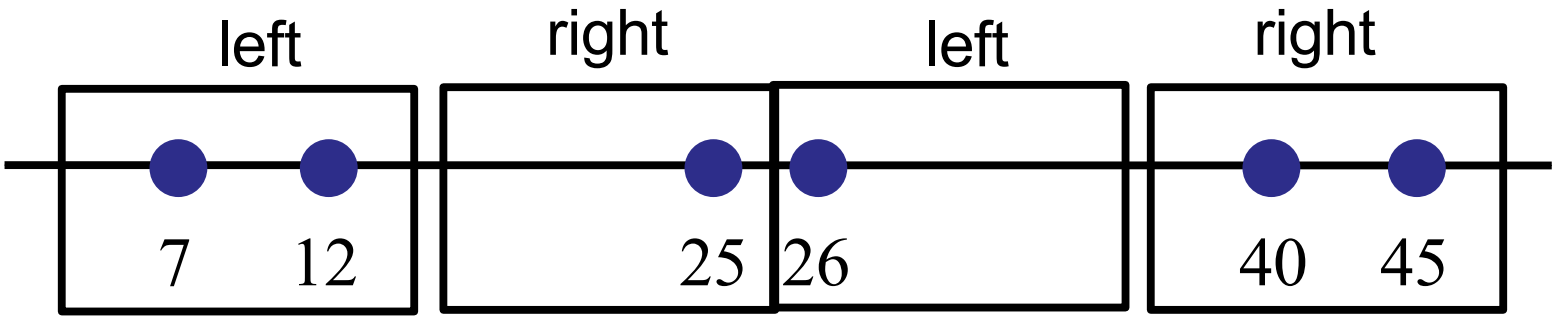


Example

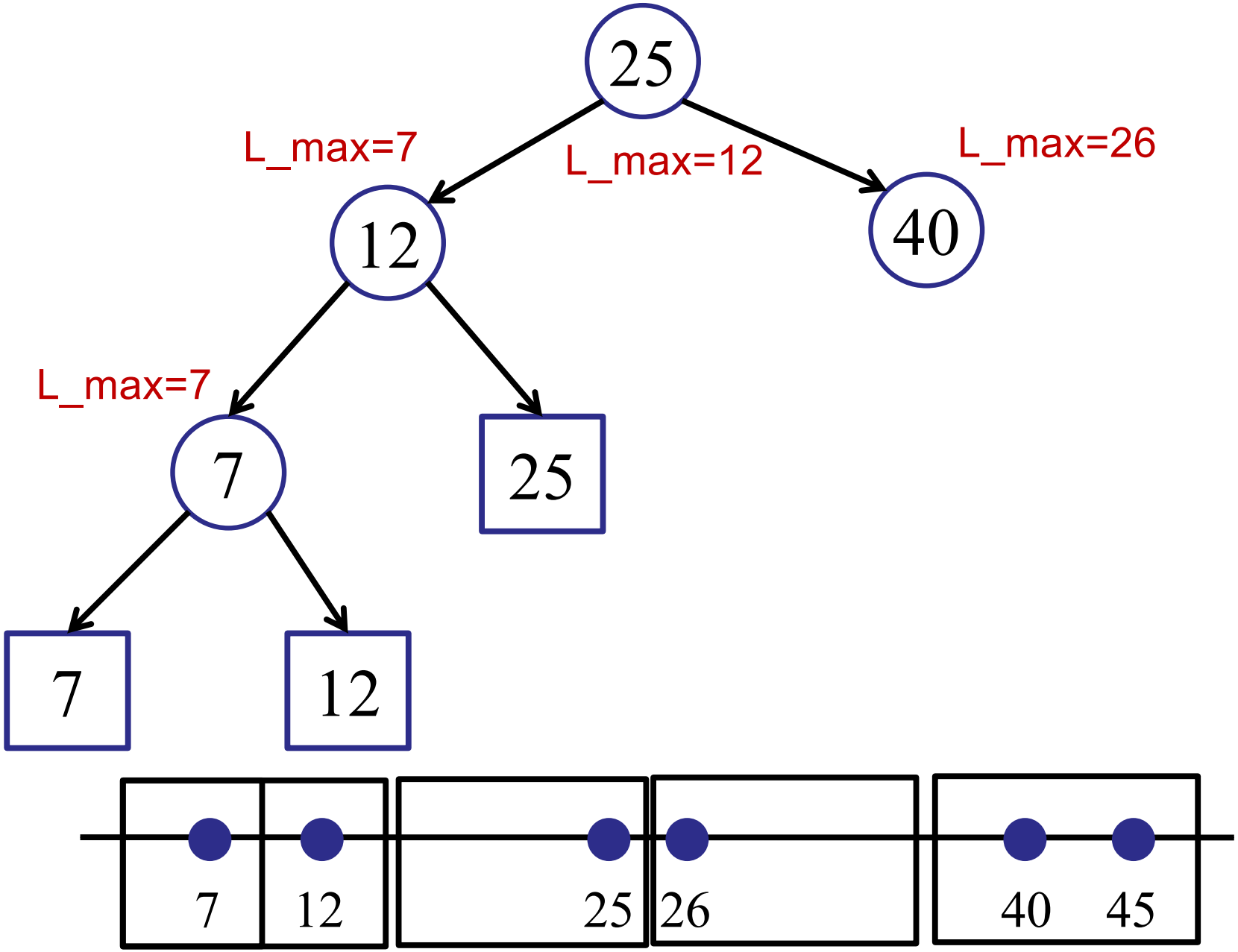


Example





Example

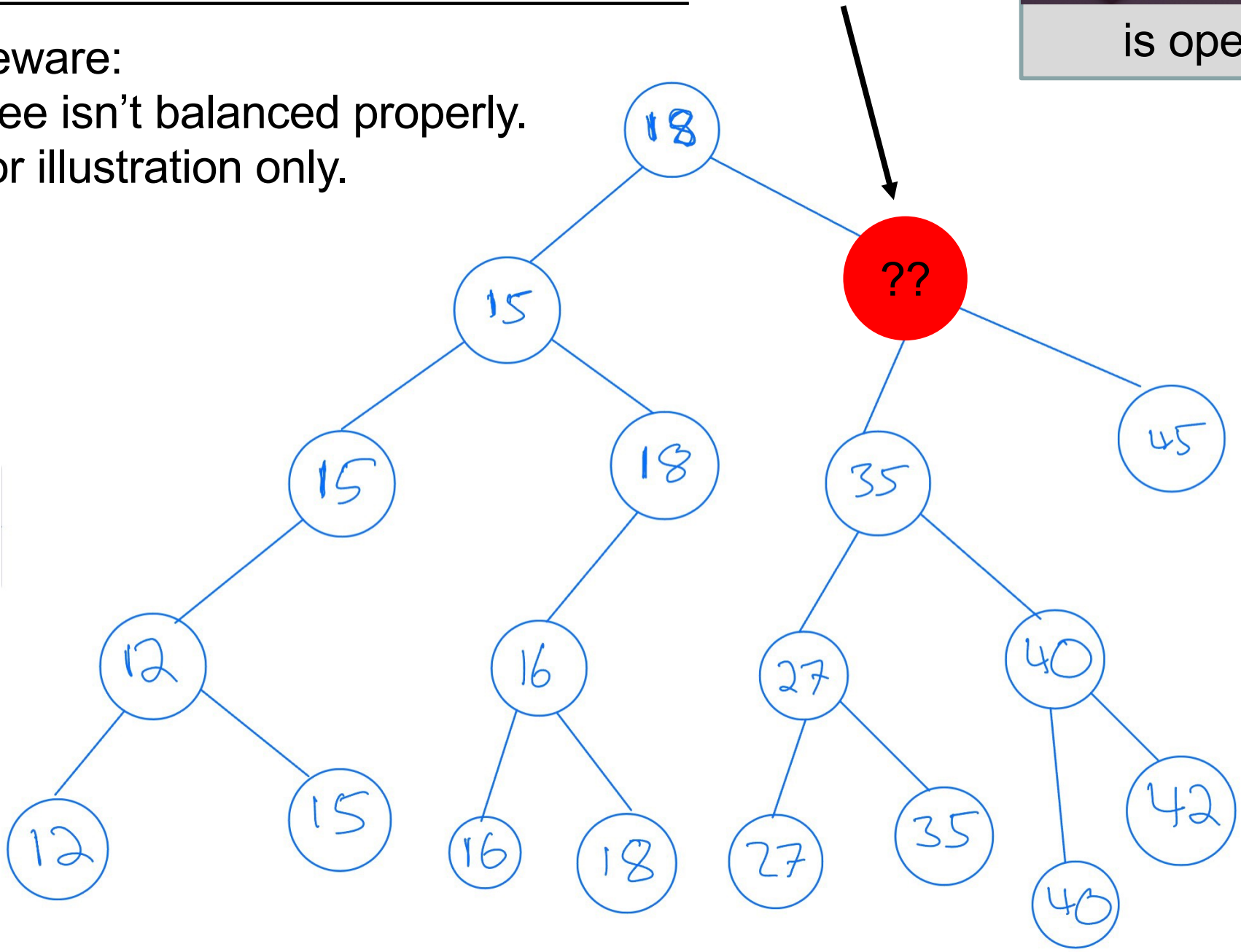


What value goes here?

ARCHIPELAGO

is open

Beware:
Tree isn't balanced properly.
For illustration only.

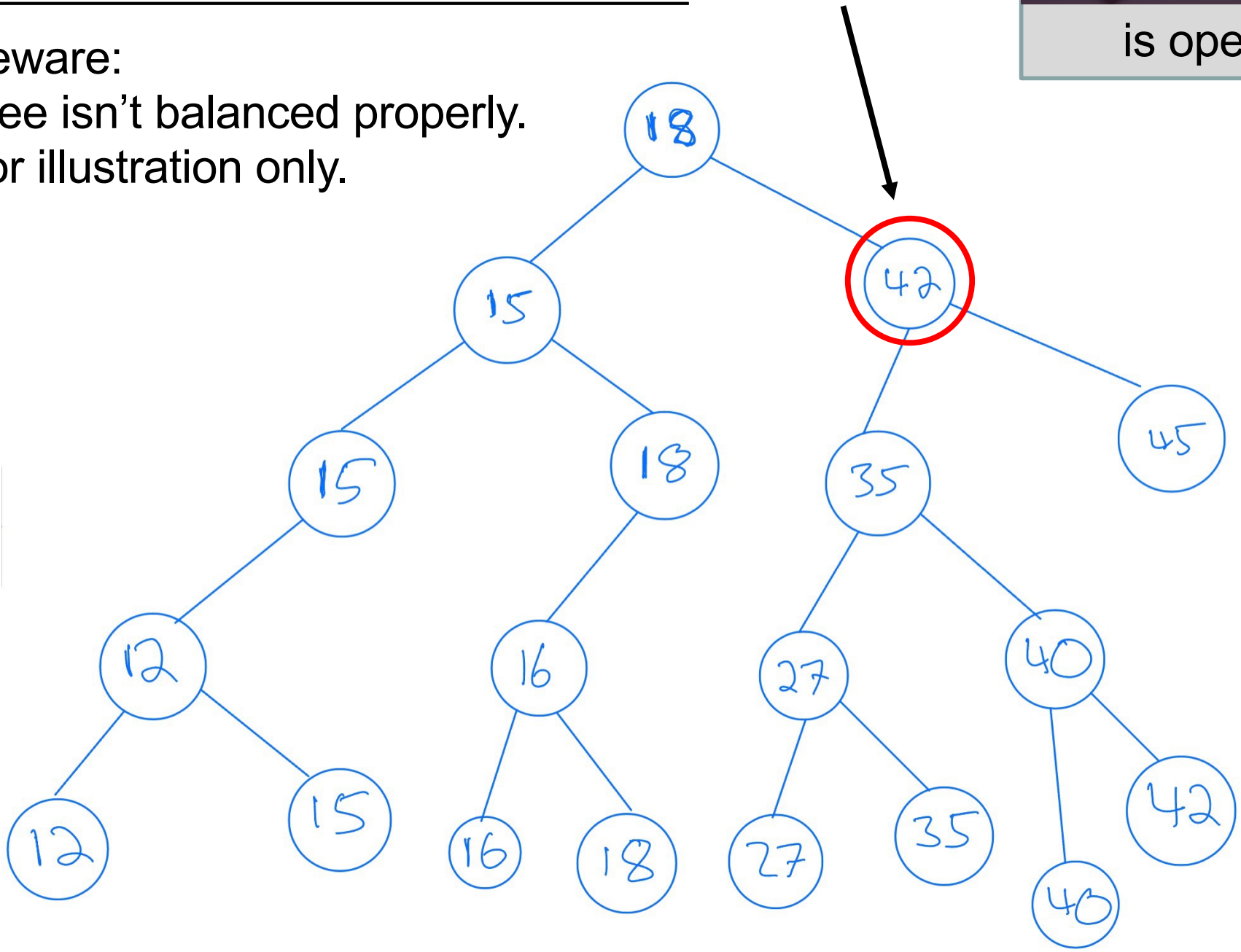


What value goes here?

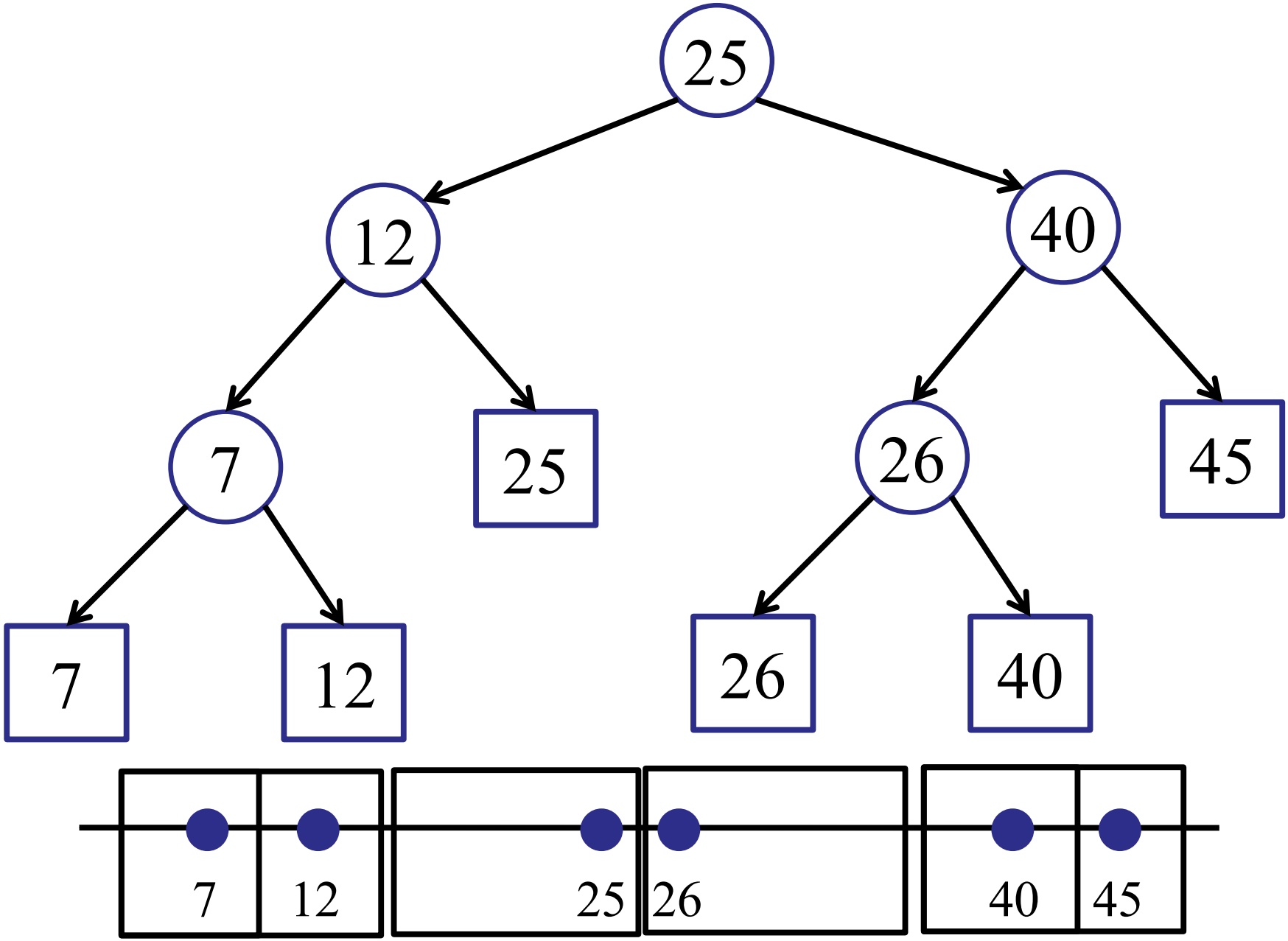
ARCHIPELAGO

is open

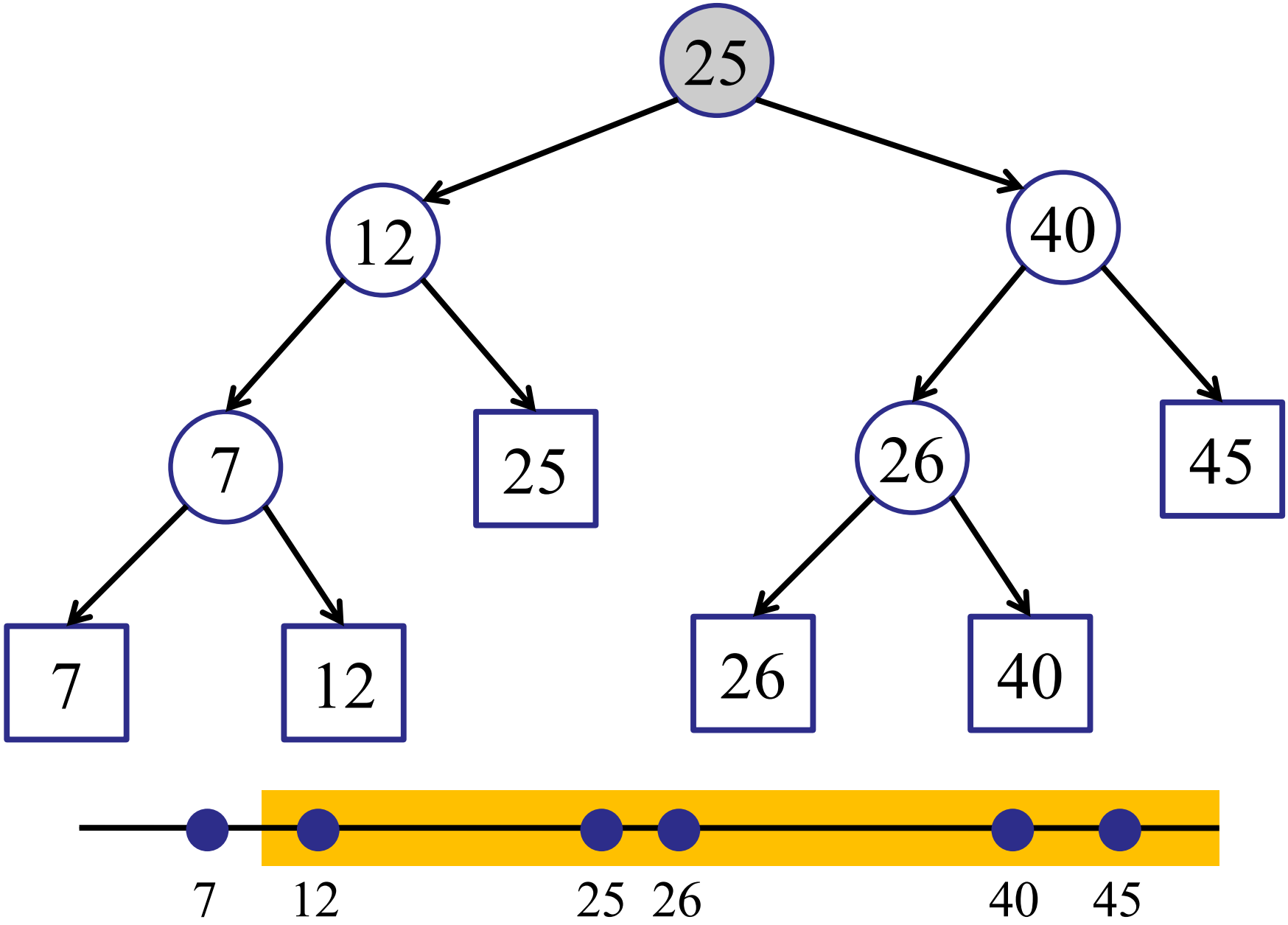
Beware:
Tree isn't balanced properly.
For illustration only.



Note: BST Property

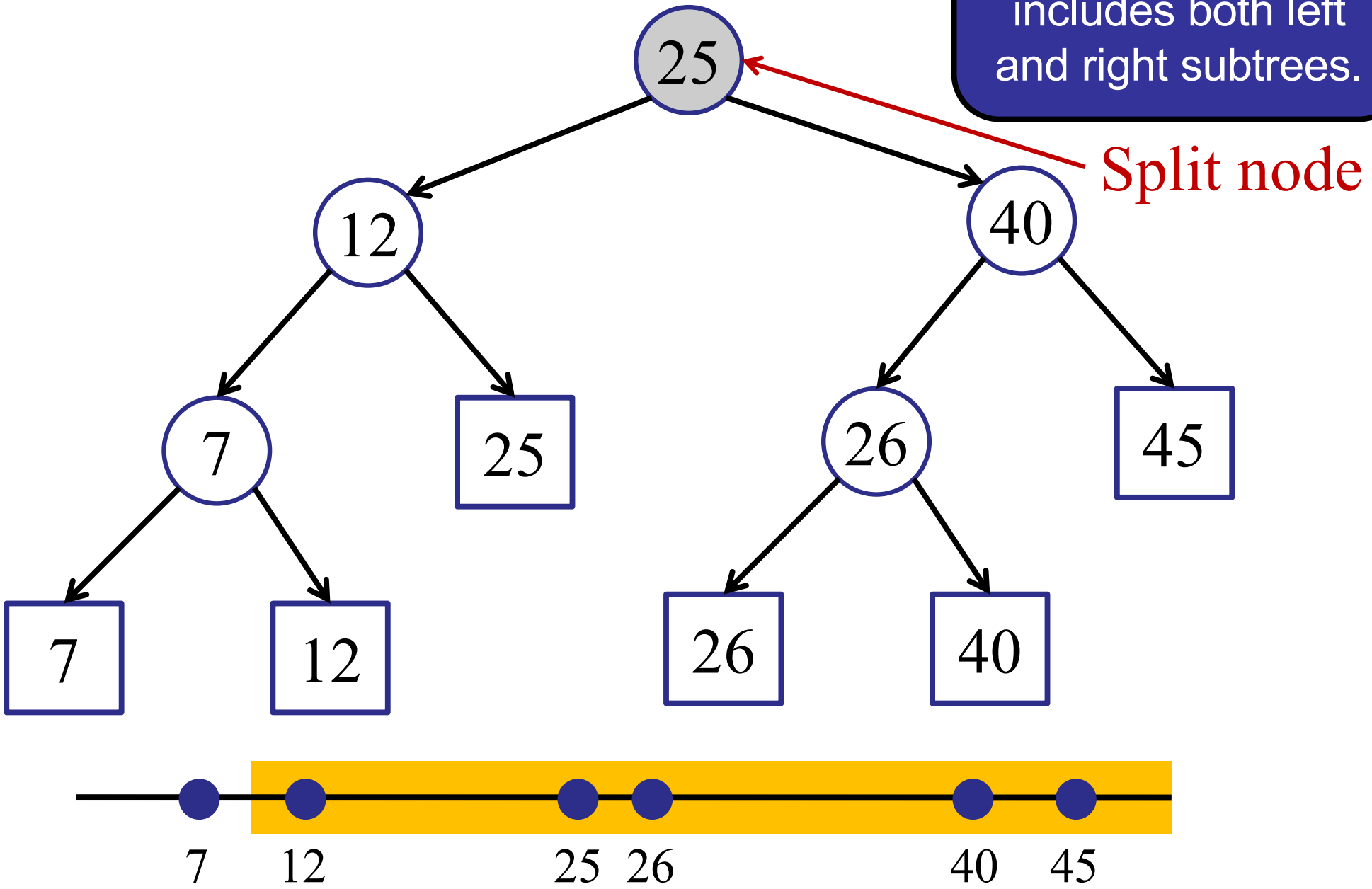


Example: query(10, 50)

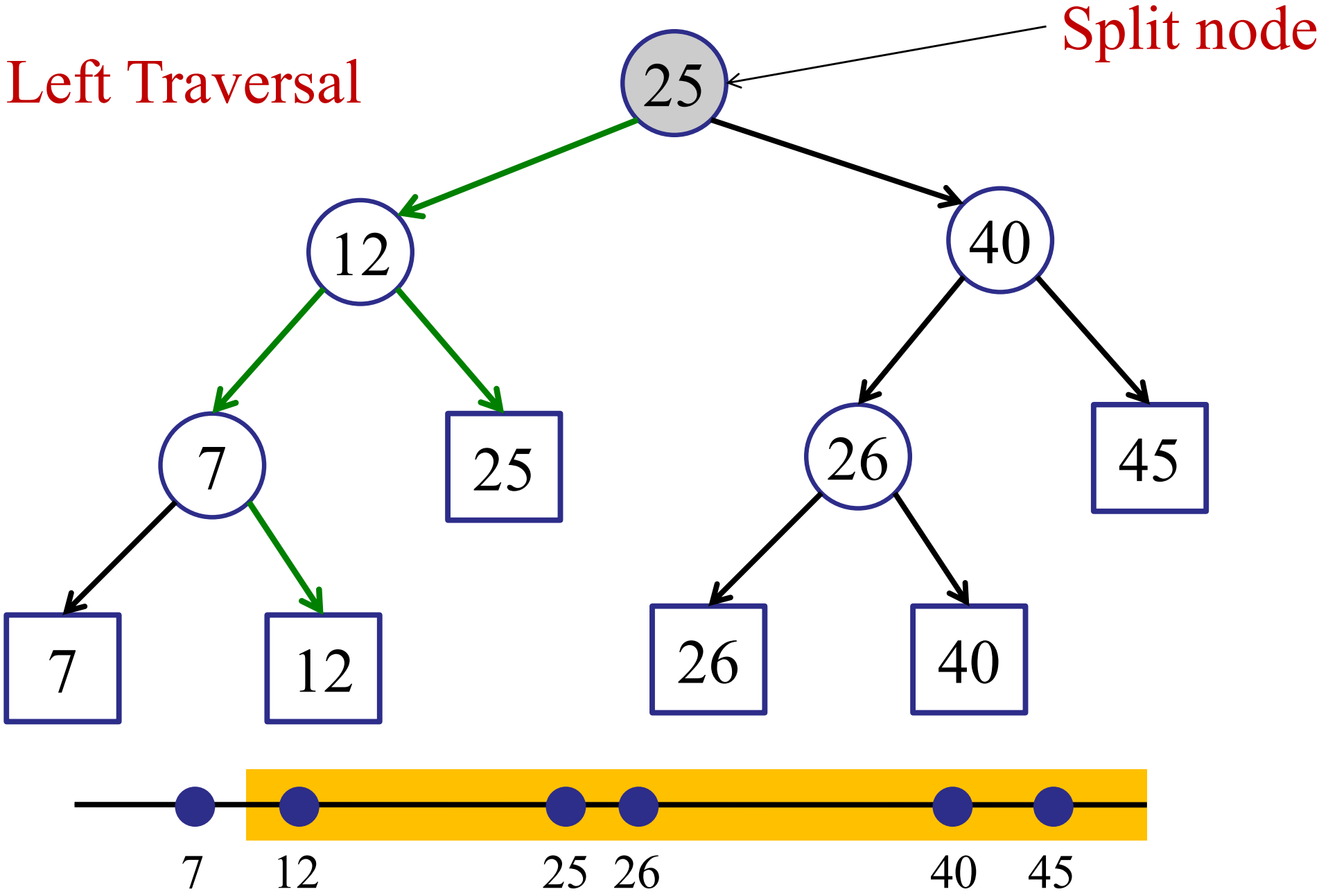


Example: query(10, 50)

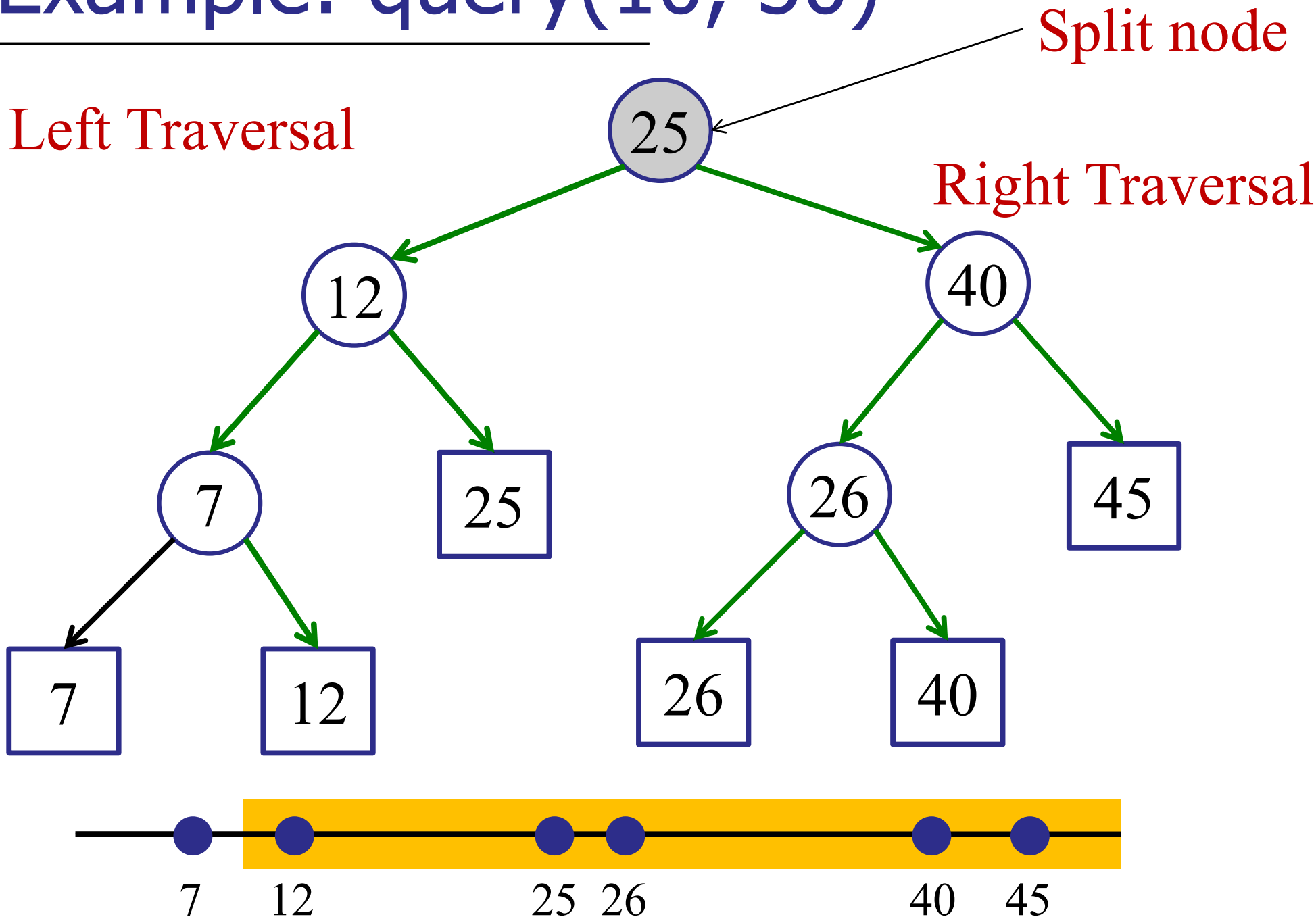
Highest node where search includes both left and right subtrees.



Example: query(10, 50)

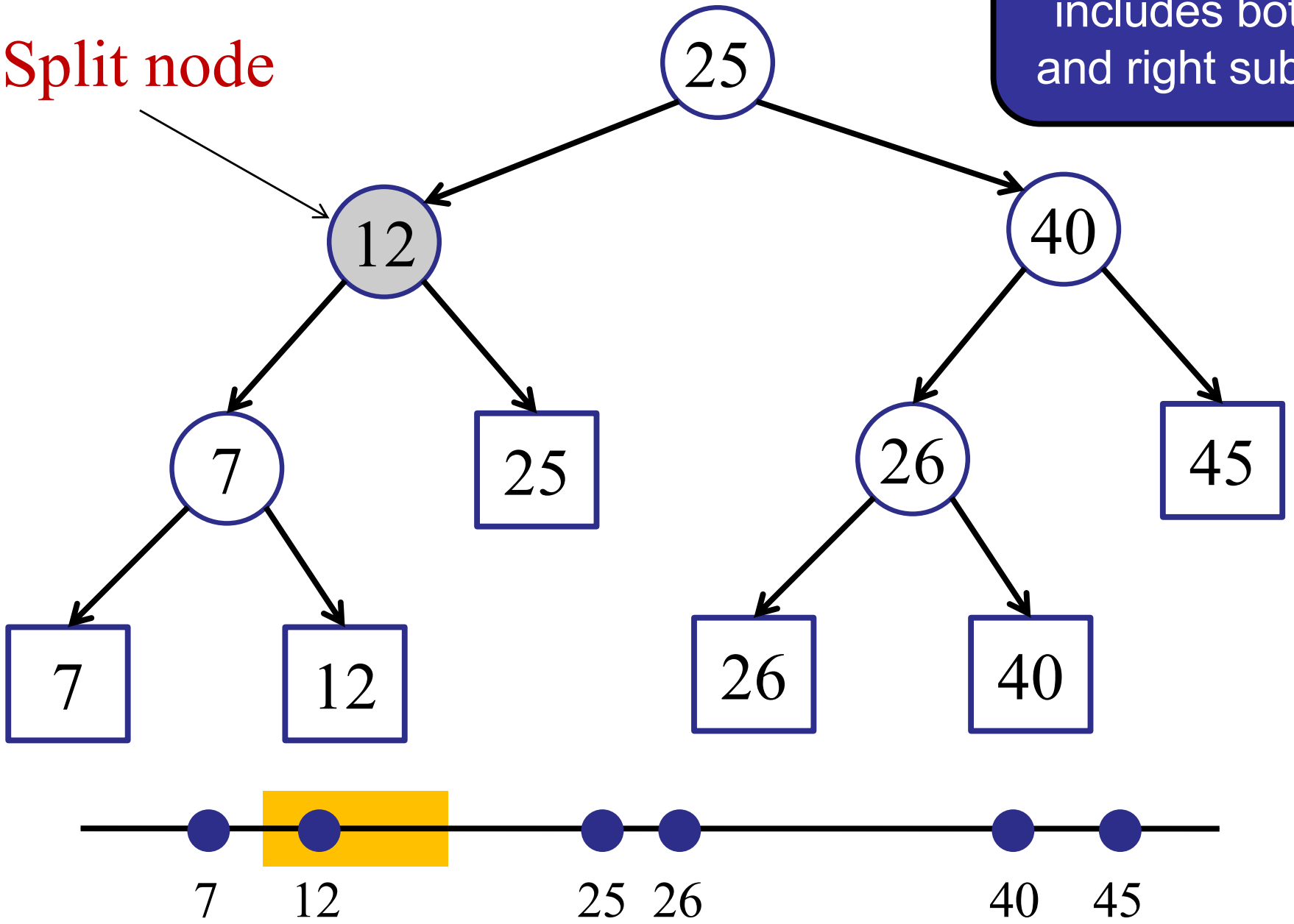


Example: query(10, 50)

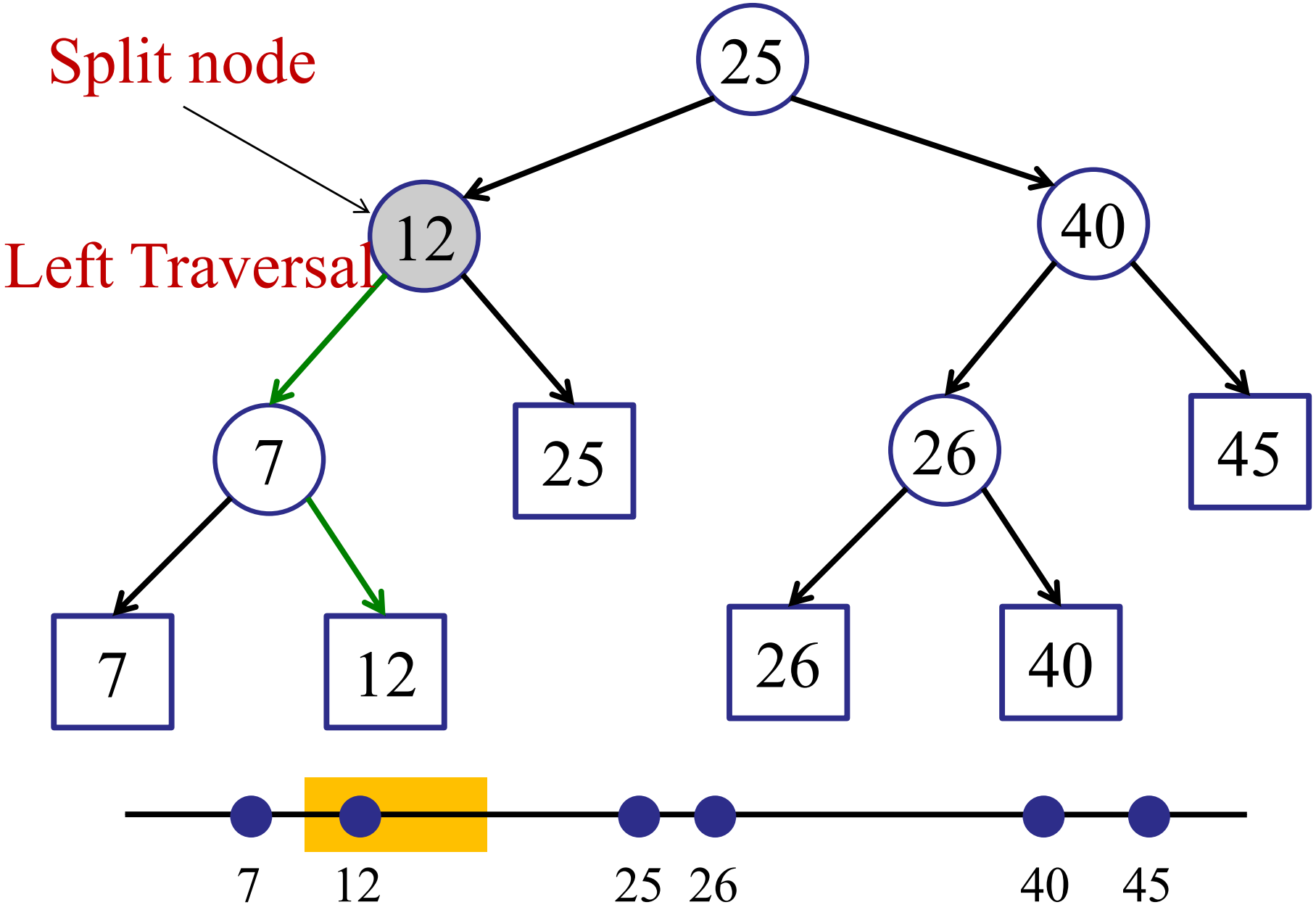


Example: query(8, 20)

Highest node where search includes both left and right subtrees.



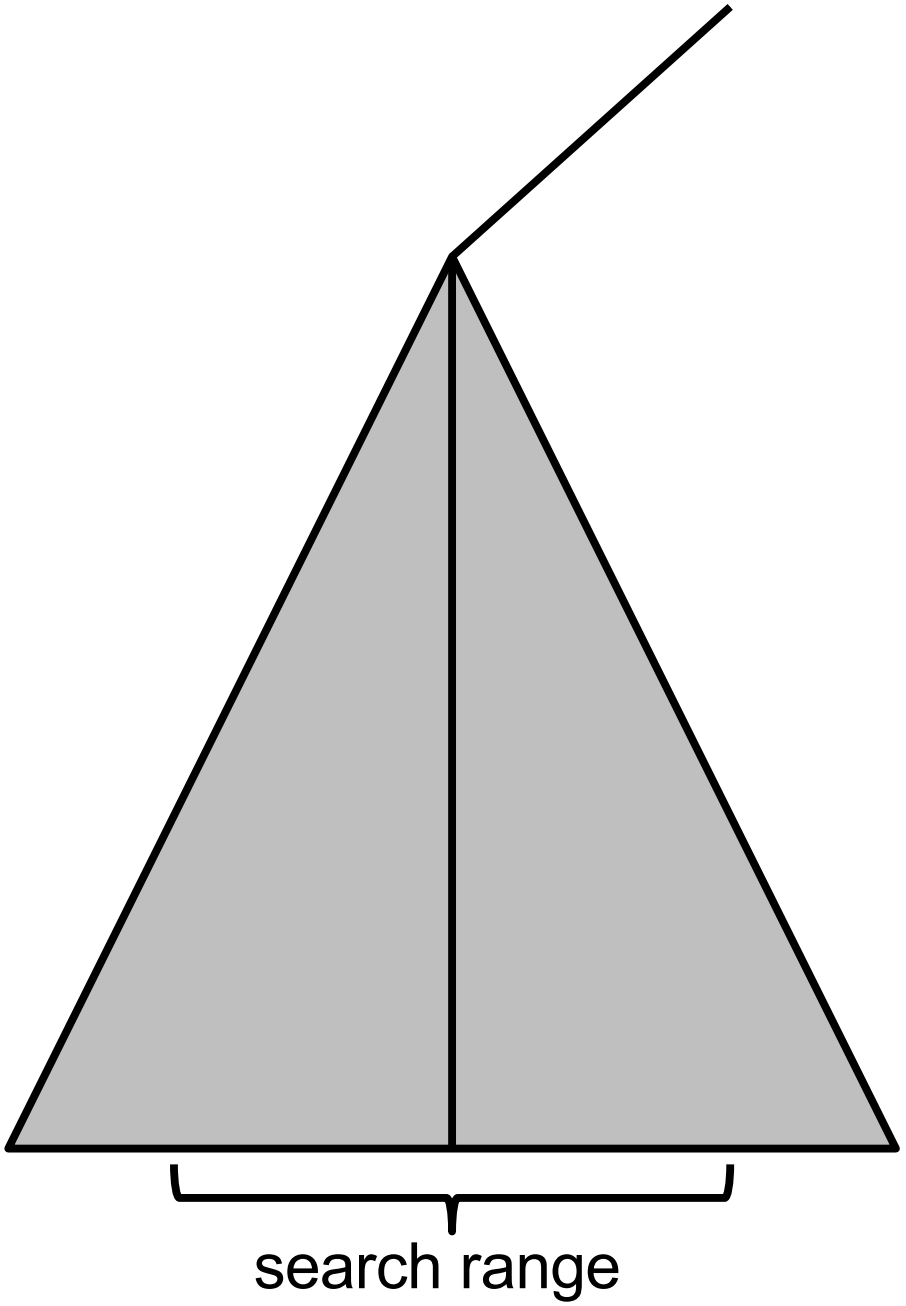
Example: query(8, 20)



One Dimensional Range Queries

Algorithm:

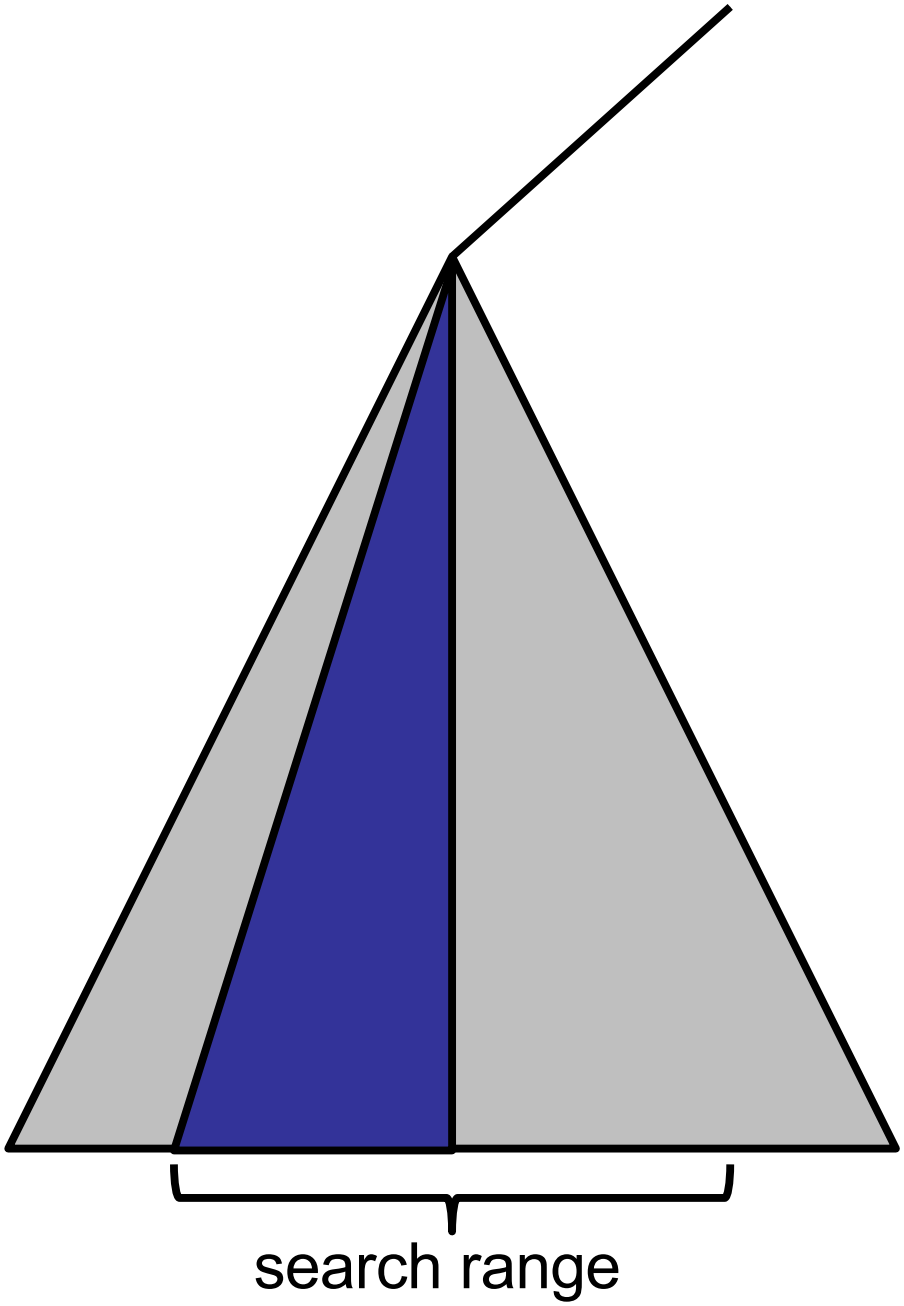
- Find "split" node.
- Do left traversal.
- Do right traversal.



One Dimensional Range Queries

Algorithm:

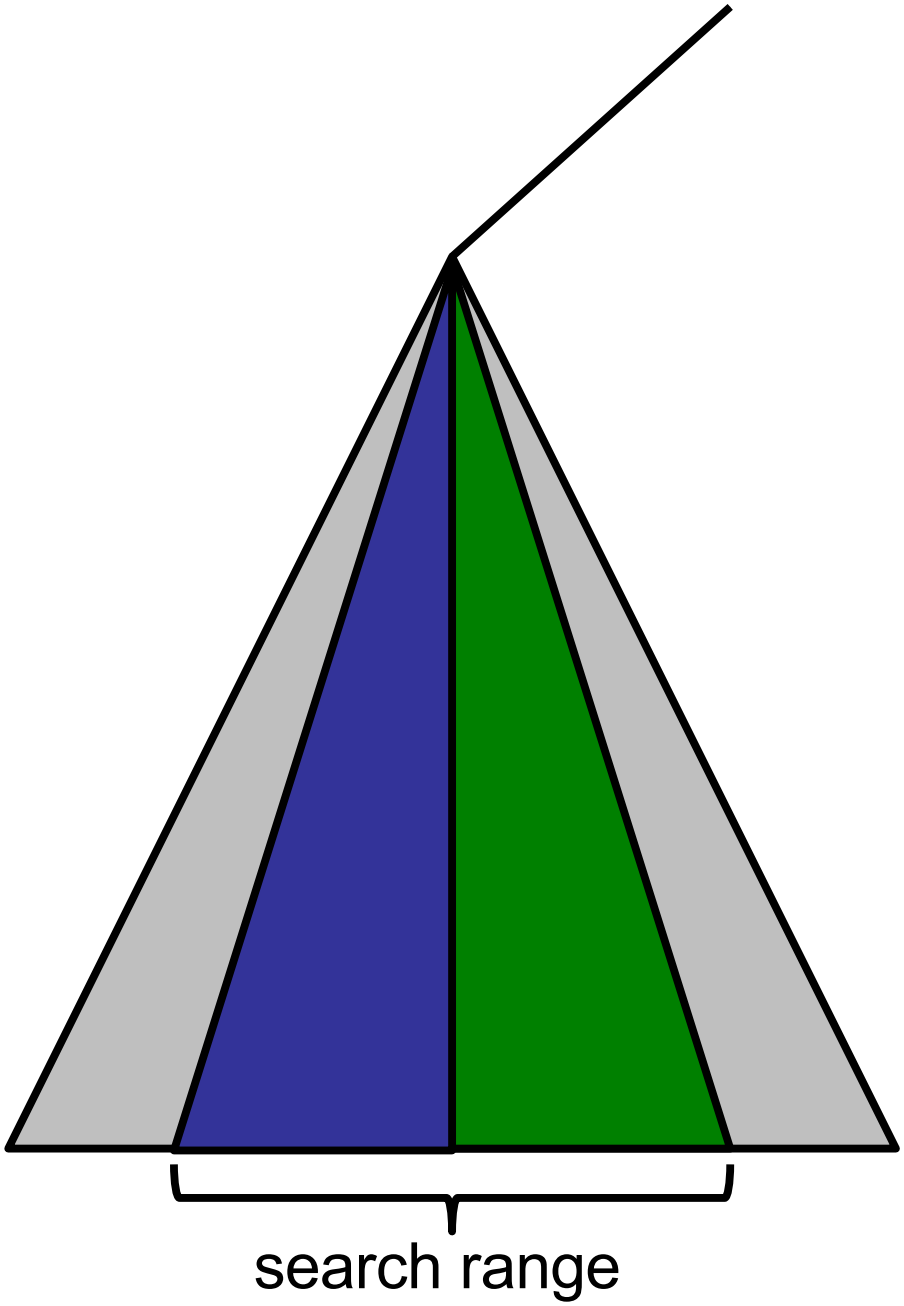
- Find "split" node.
- Do left traversal.
- Do right traversal.



One Dimensional Range Queries

Algorithm:

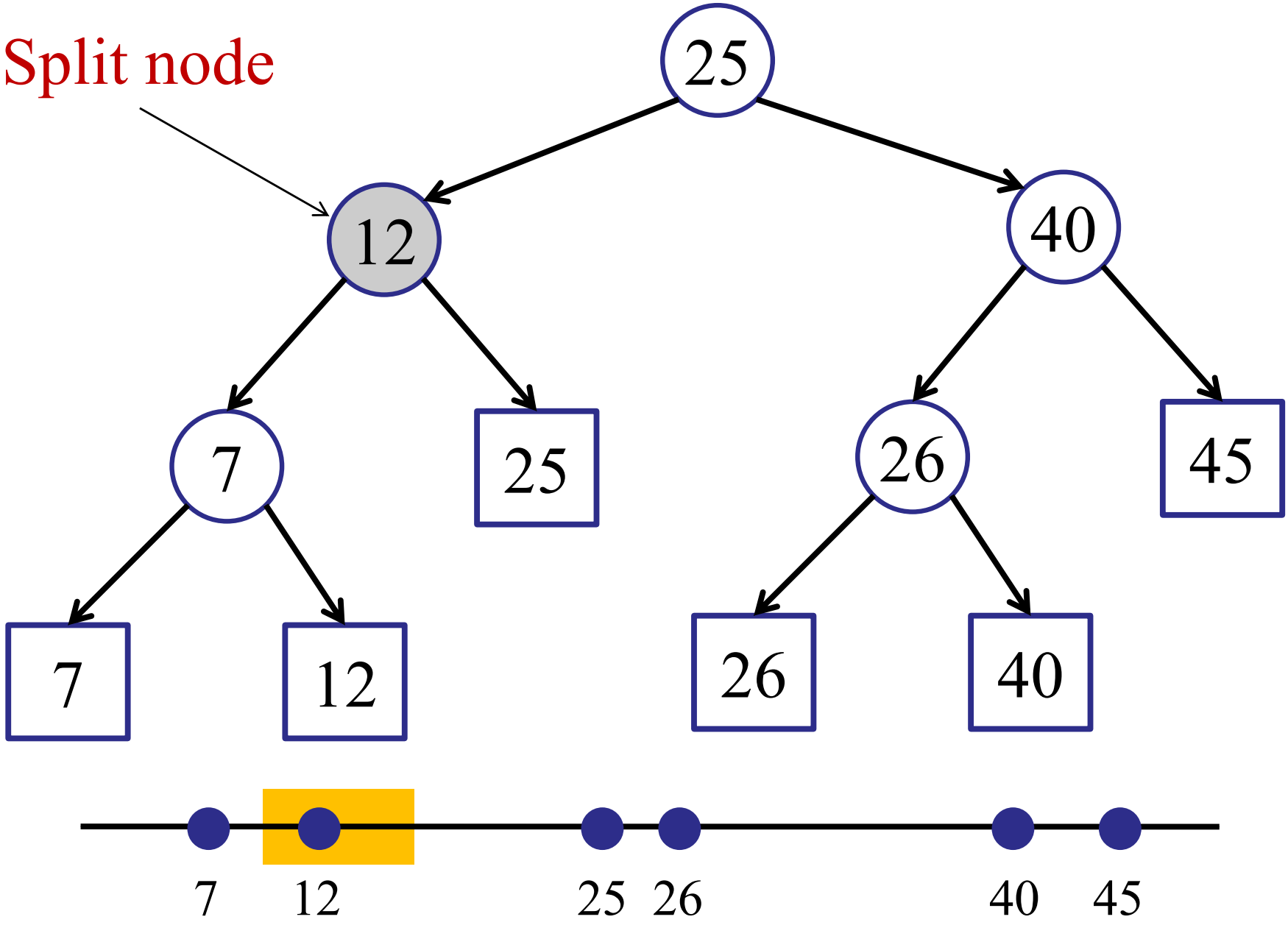
- Find "split" node.
- Do left traversal.
- Do right traversal.



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)



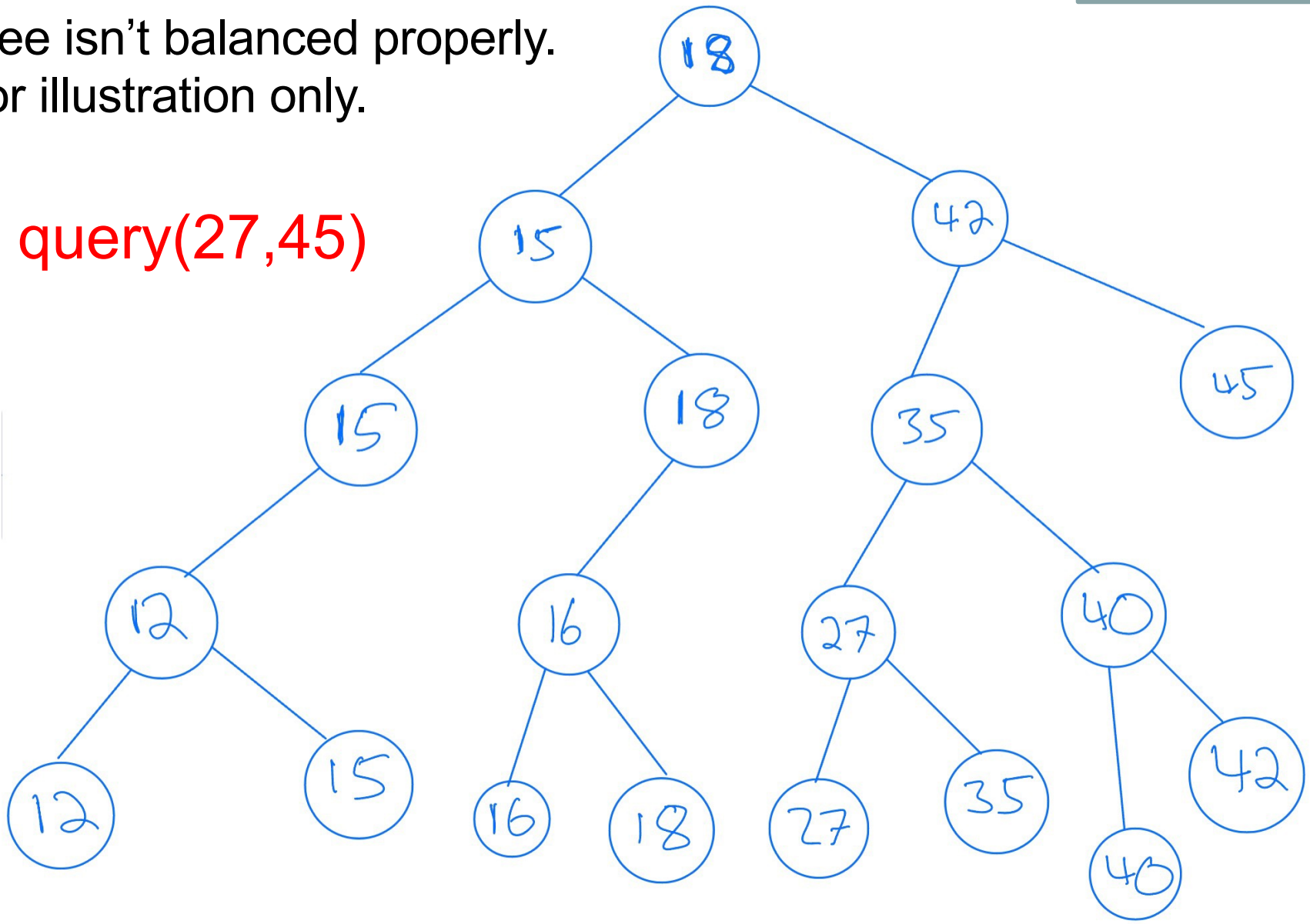
What is the split node?

ARCHIPELAGO

is open

Beware:
Tree isn't balanced properly.
For illustration only.

query(27,45)



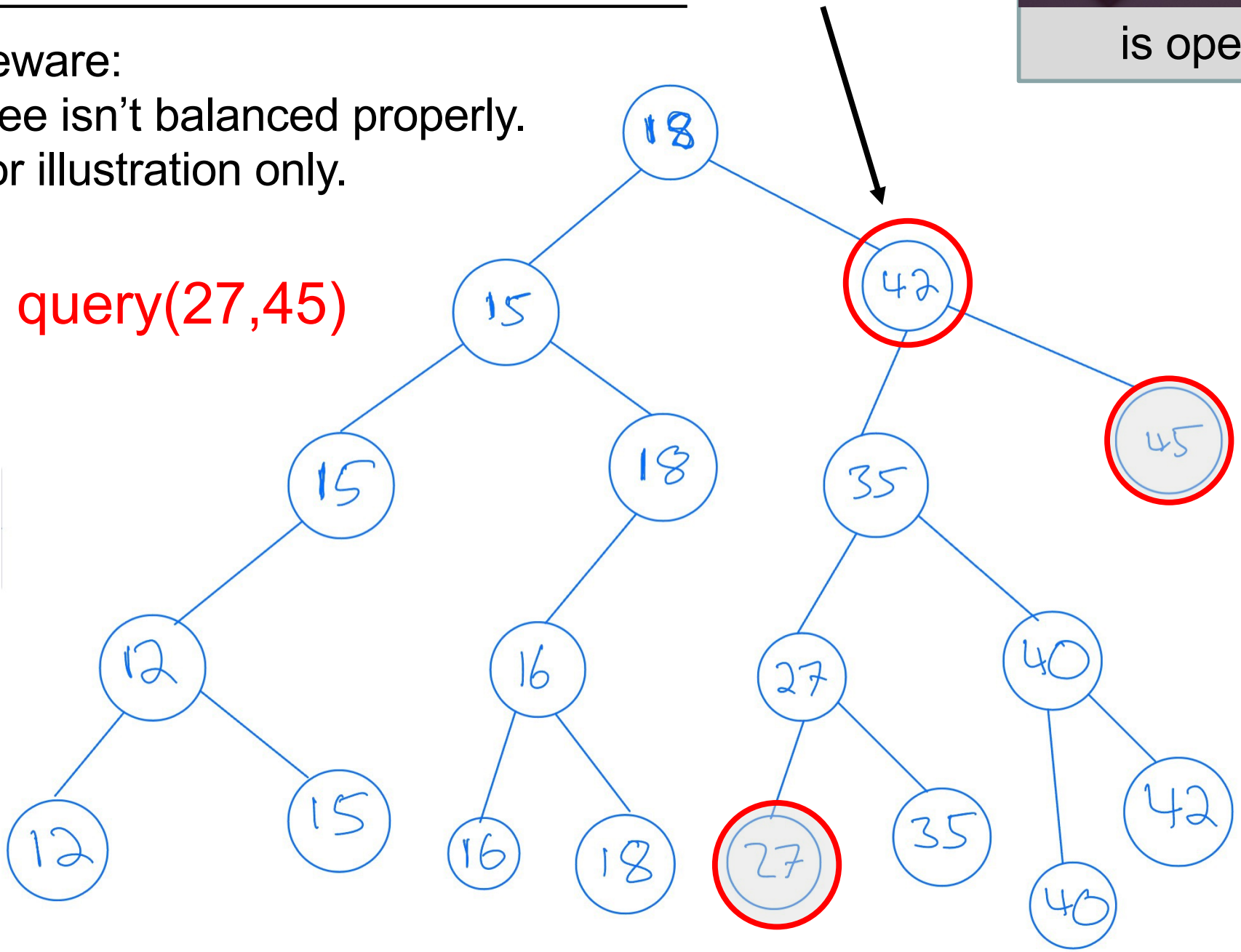
What is the split node?

ARCHIPELAGO

is open

Beware:
Tree isn't balanced properly.
For illustration only.

query(27,45)



One Dimensional Range Queries

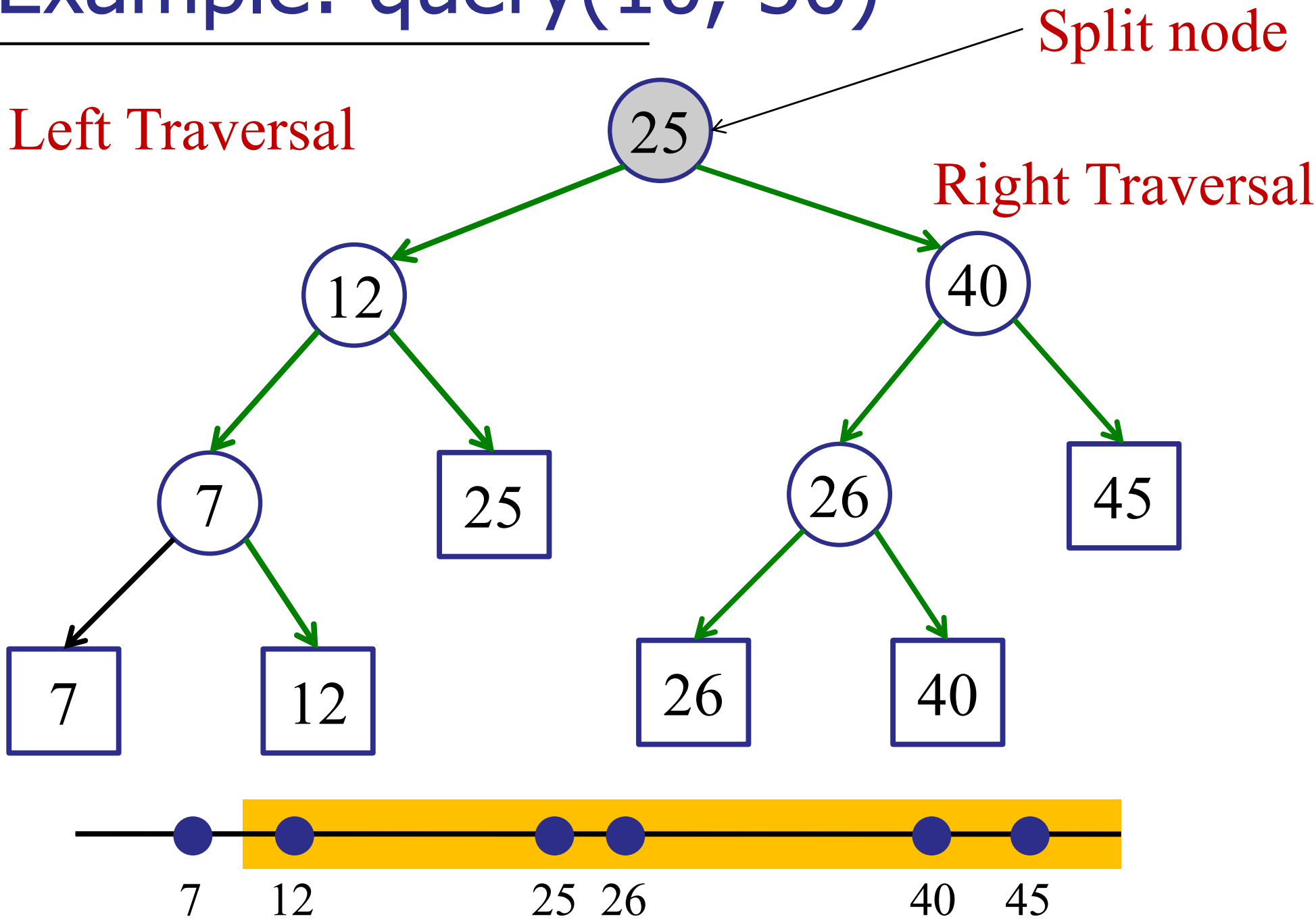
Algorithm:

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

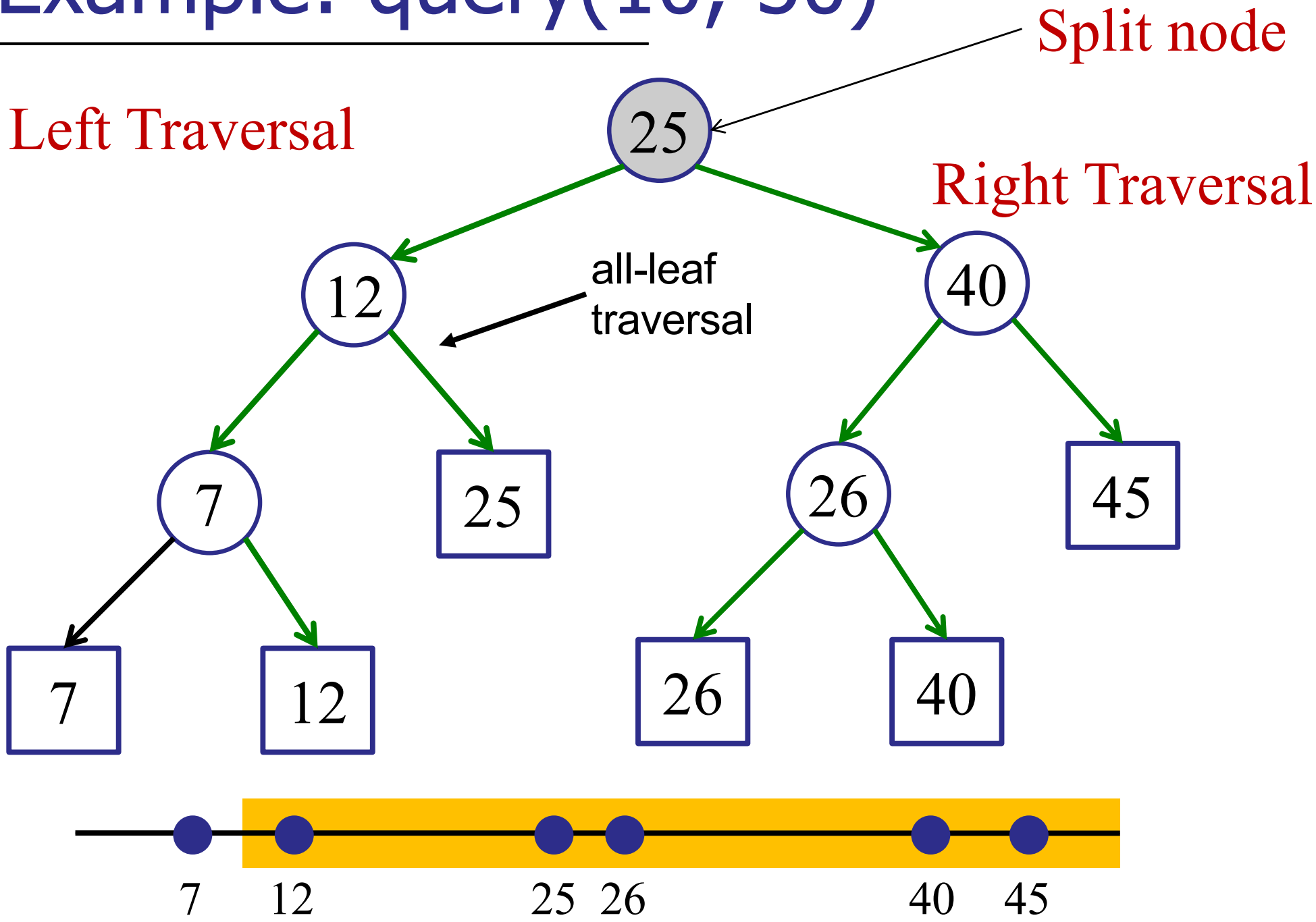
One Dimensional Range Queries

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

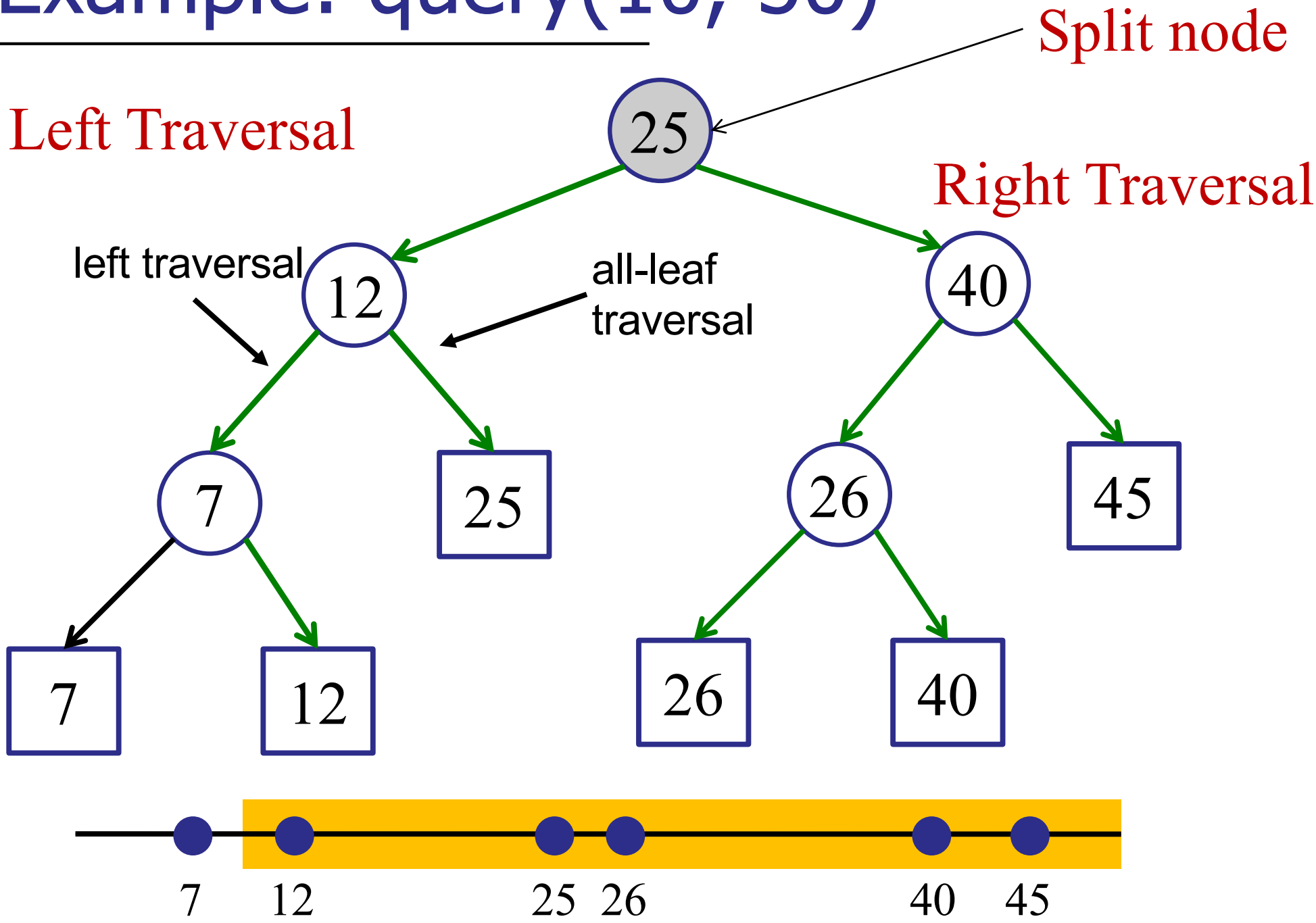
Example: query(10, 50)



Example: query(10, 50)



Example: query(10, 50)

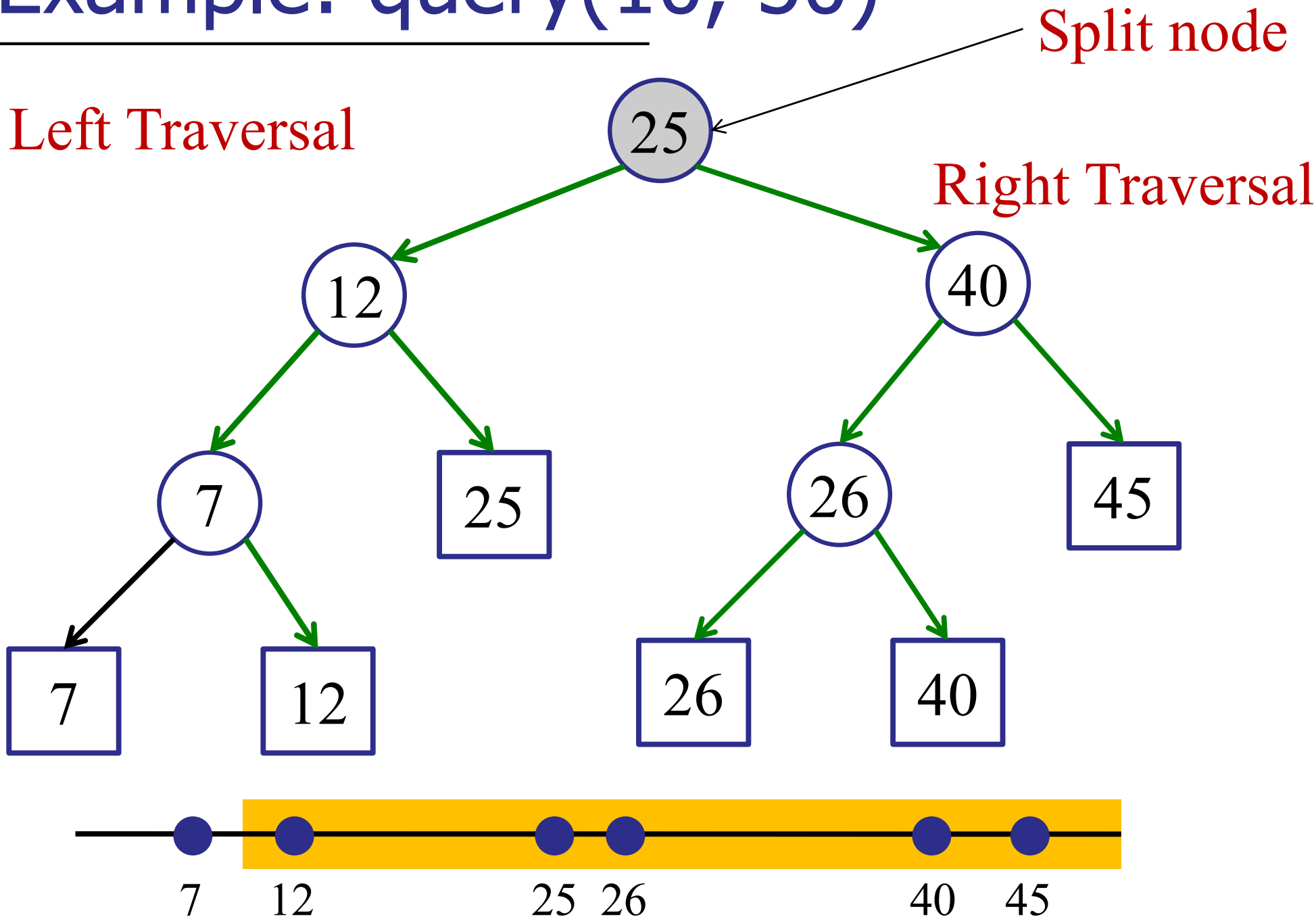


One Dimensional Range Queries

Invariant:

The search interval for a left-traversal at node v includes the maximum item in the subtree rooted at v.

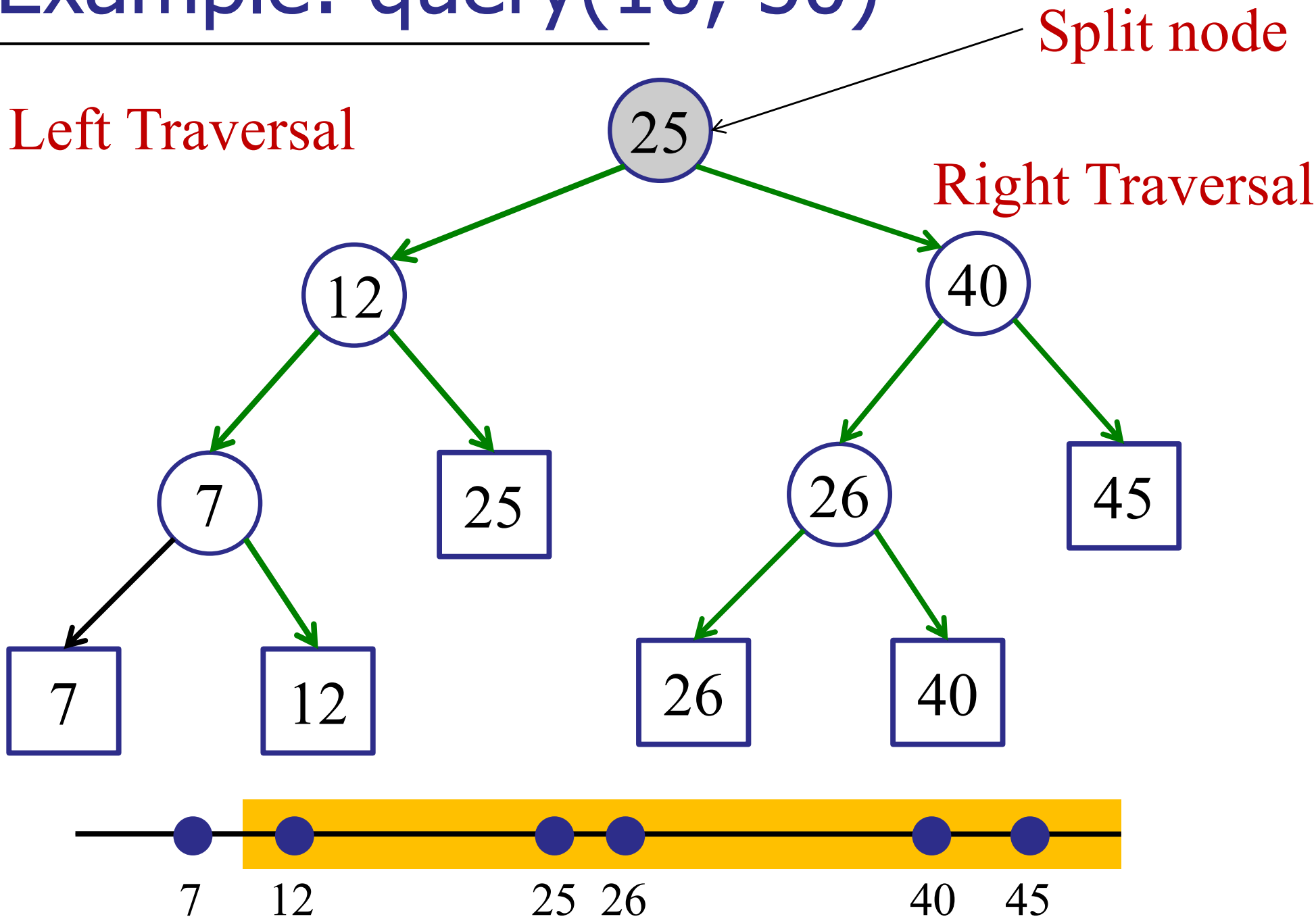
Example: query(10, 50)



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);
        }
}
```

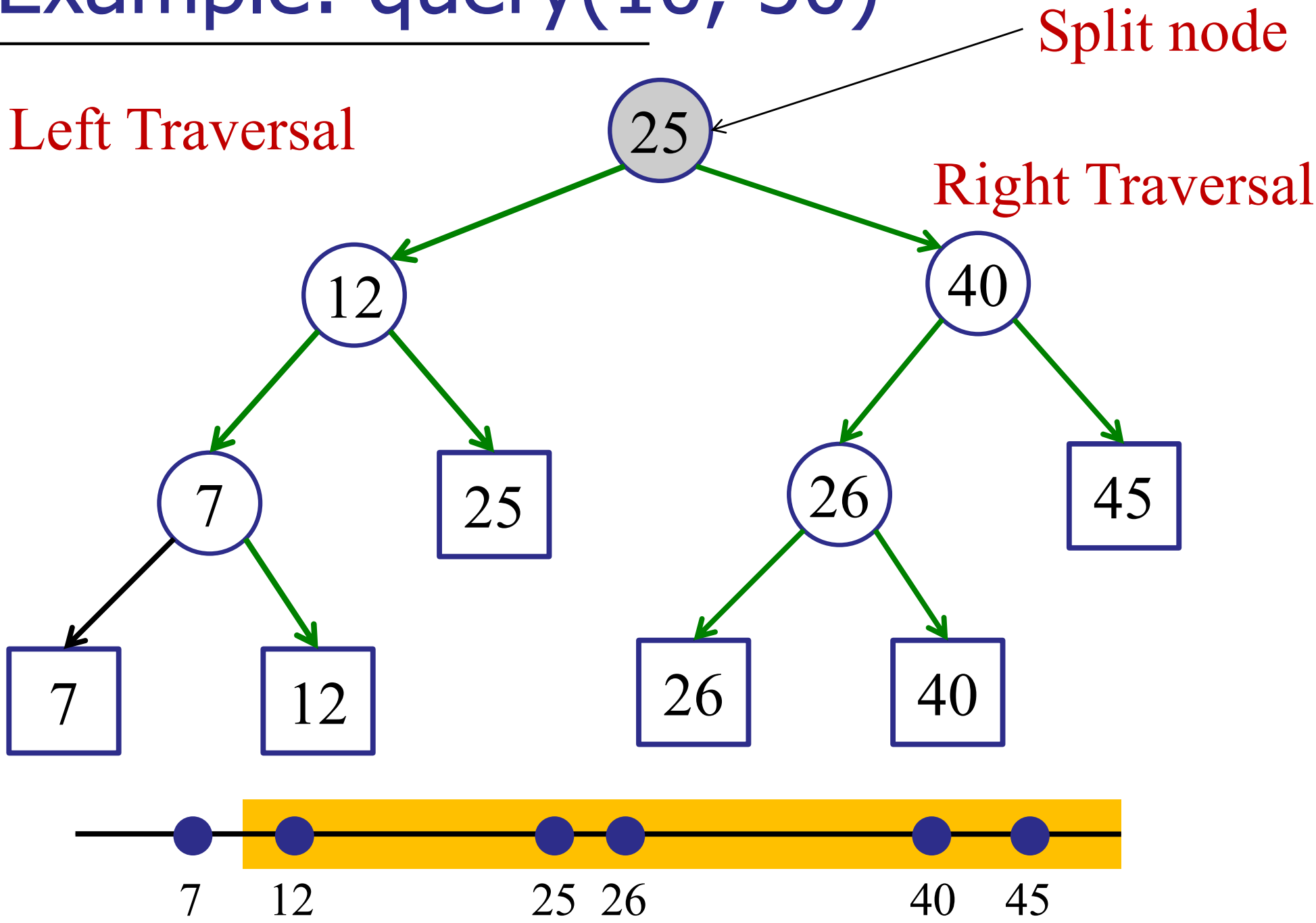
Example: query(10, 50)



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);
        }
}
```

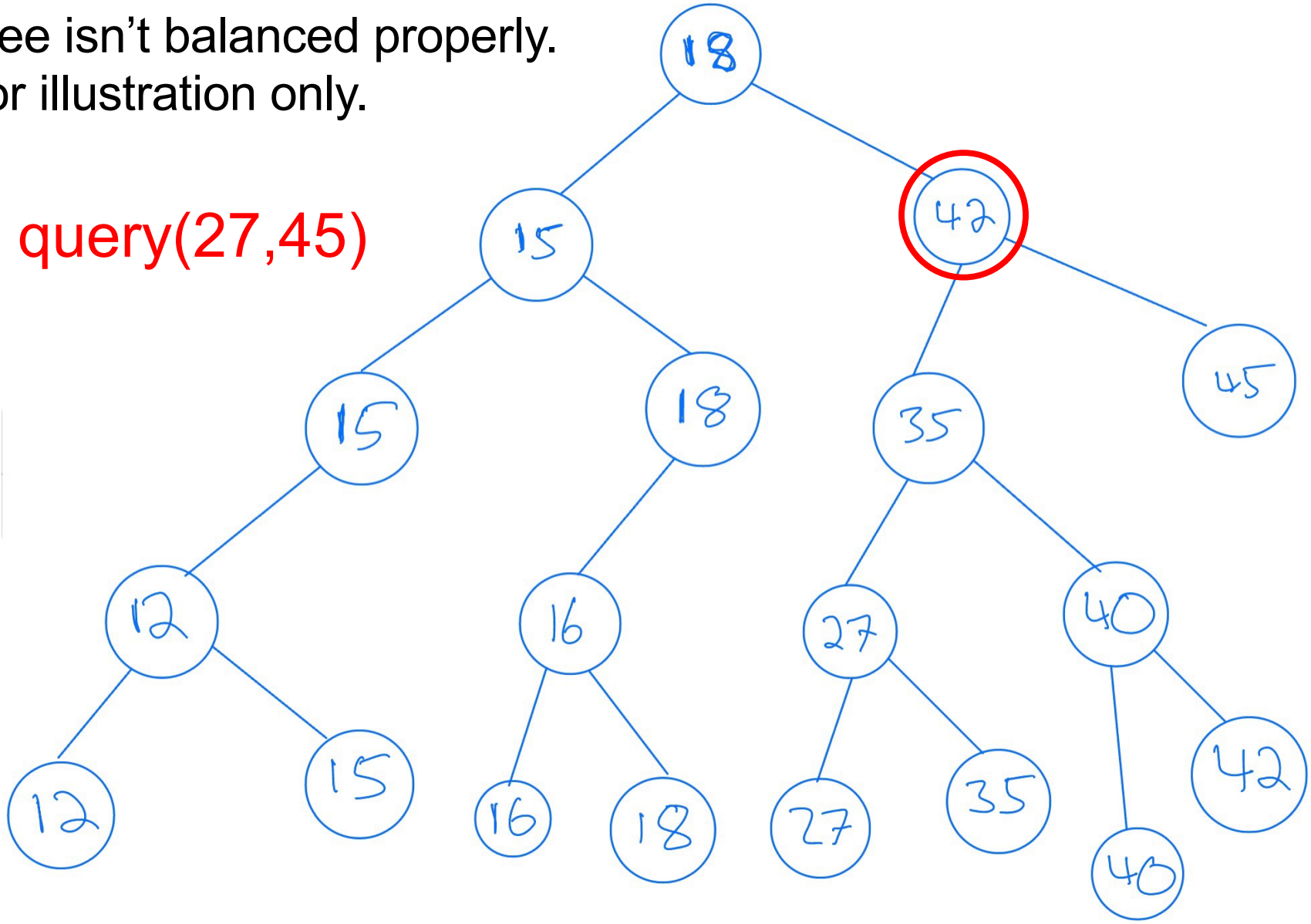
Example: query(10, 50)



Another example

Beware:
Tree isn't balanced properly.
For illustration only.

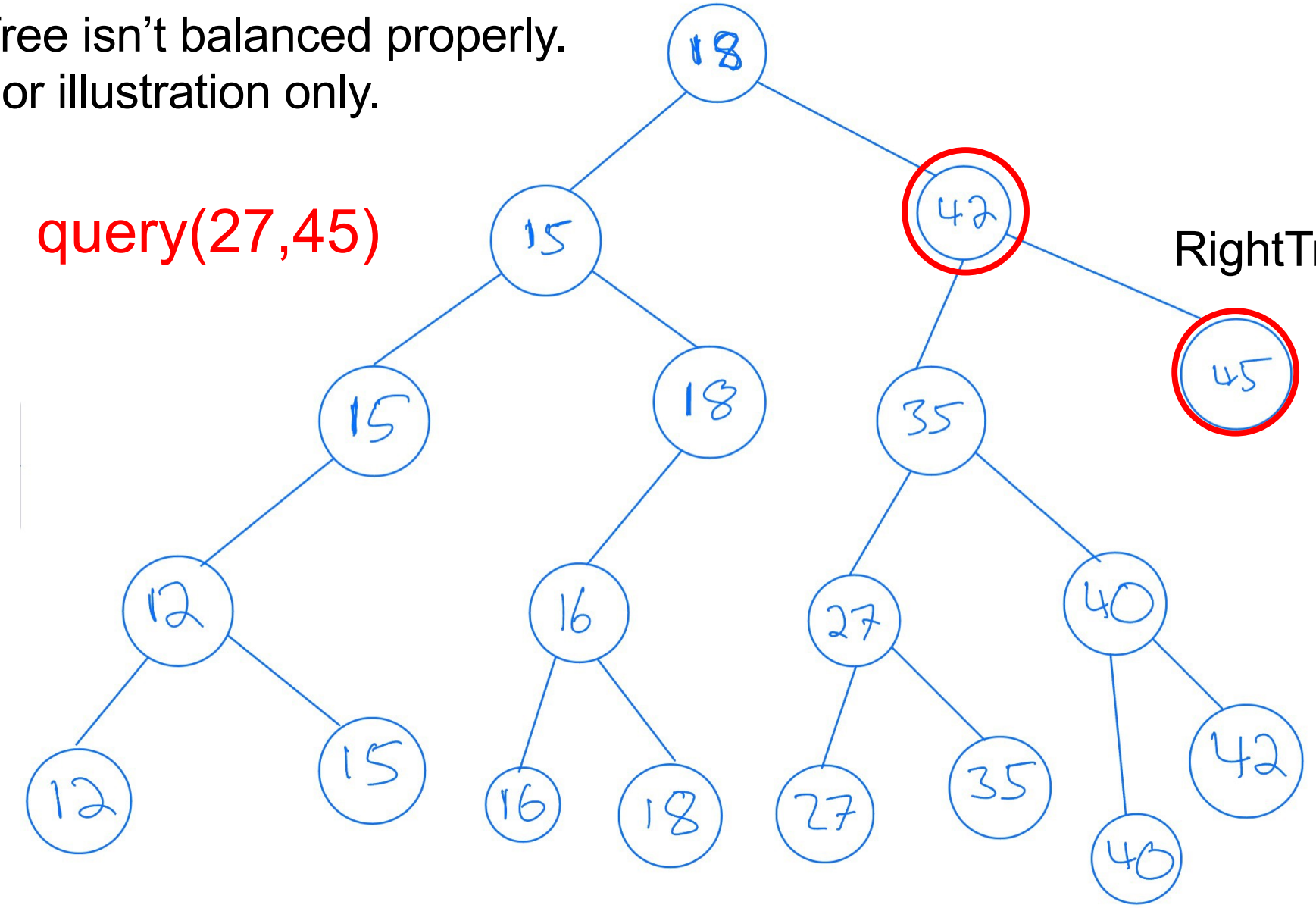
query(27,45)



Another example

Beware:
Tree isn't balanced properly.
For illustration only.

query(27,45)

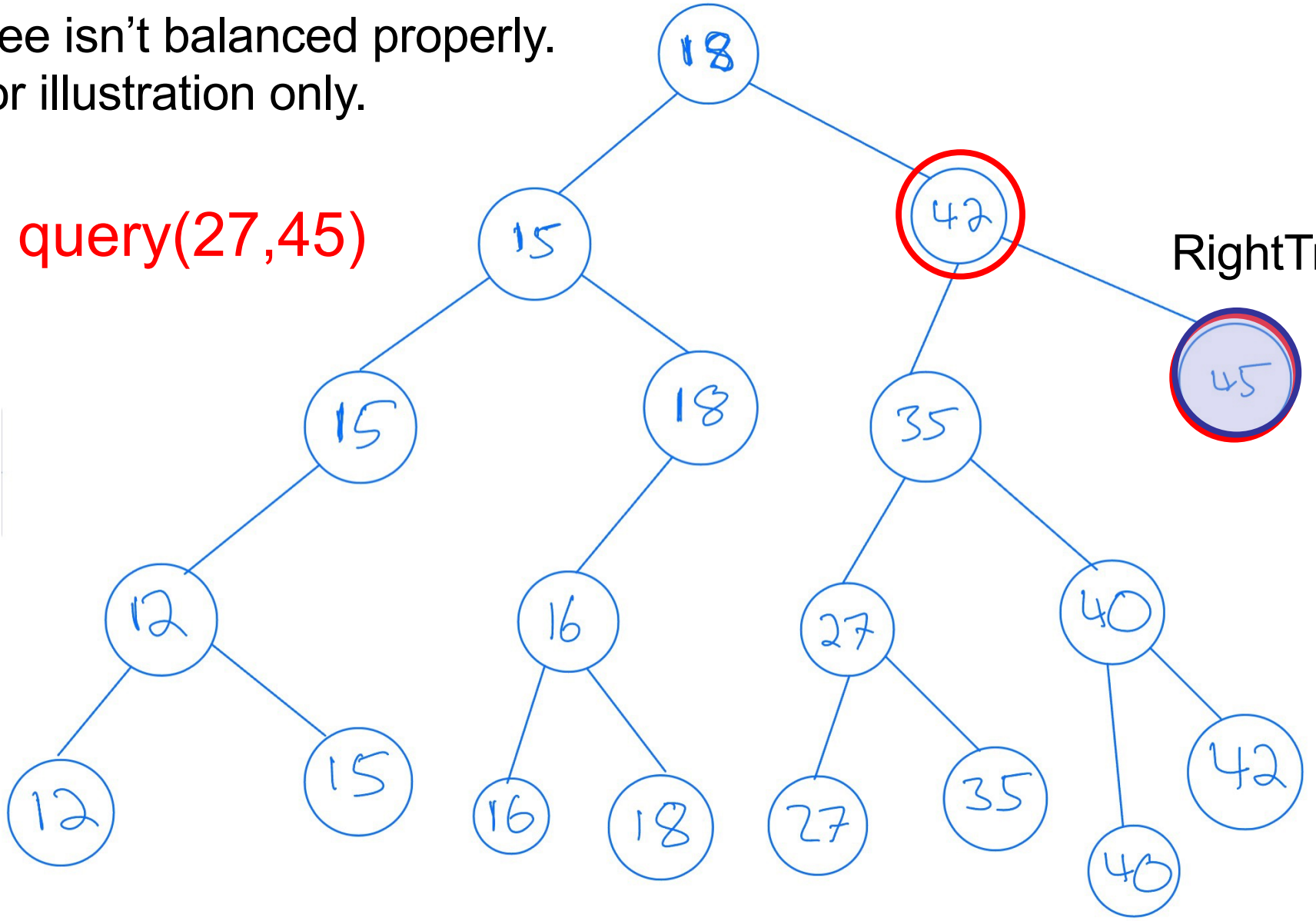


RightTraversal

Another example

Beware:
Tree isn't balanced properly.
For illustration only.

query(27,45)

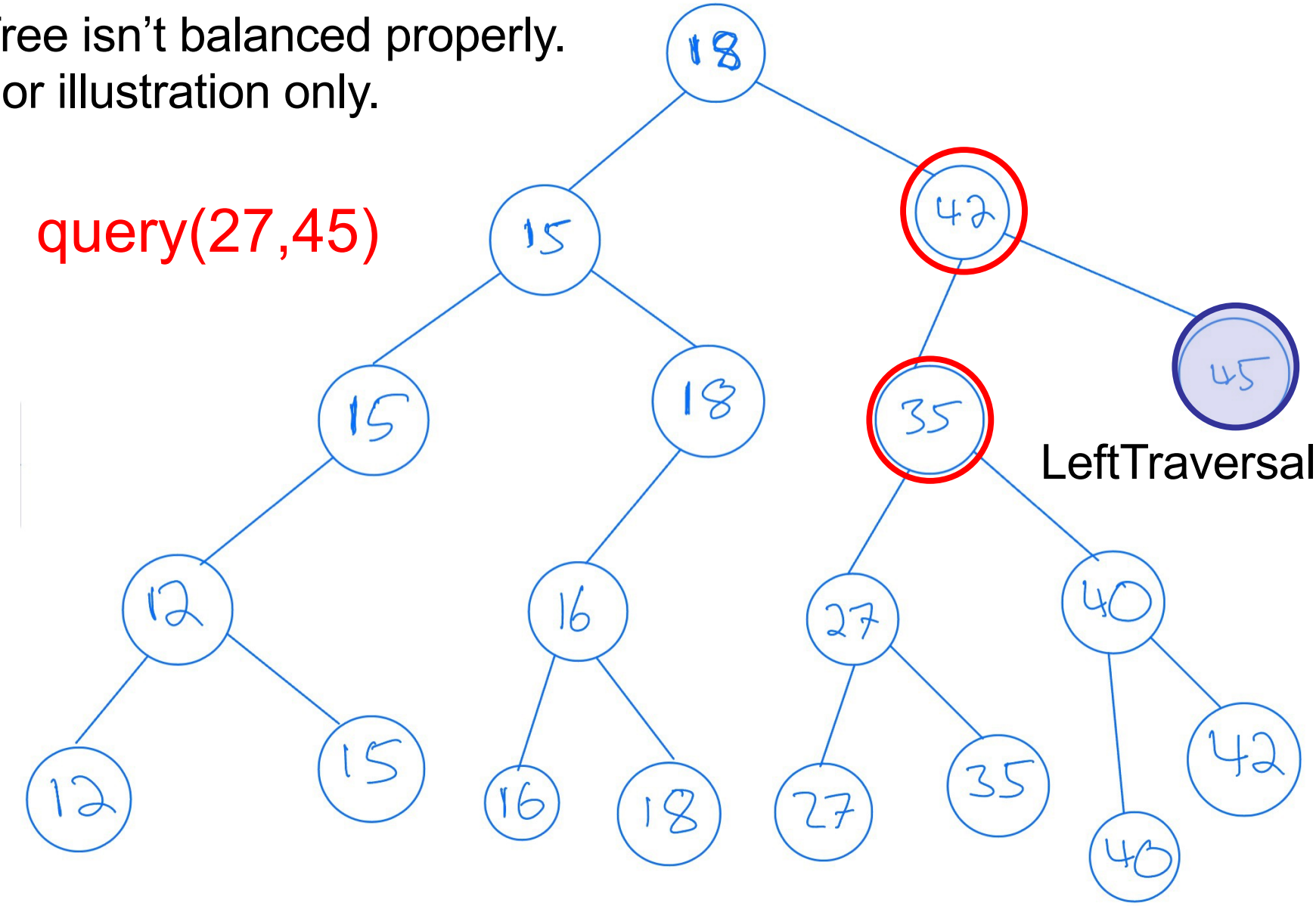


RightTraversal

Another example

Beware:
Tree isn't balanced properly.
For illustration only.

query(27,45)

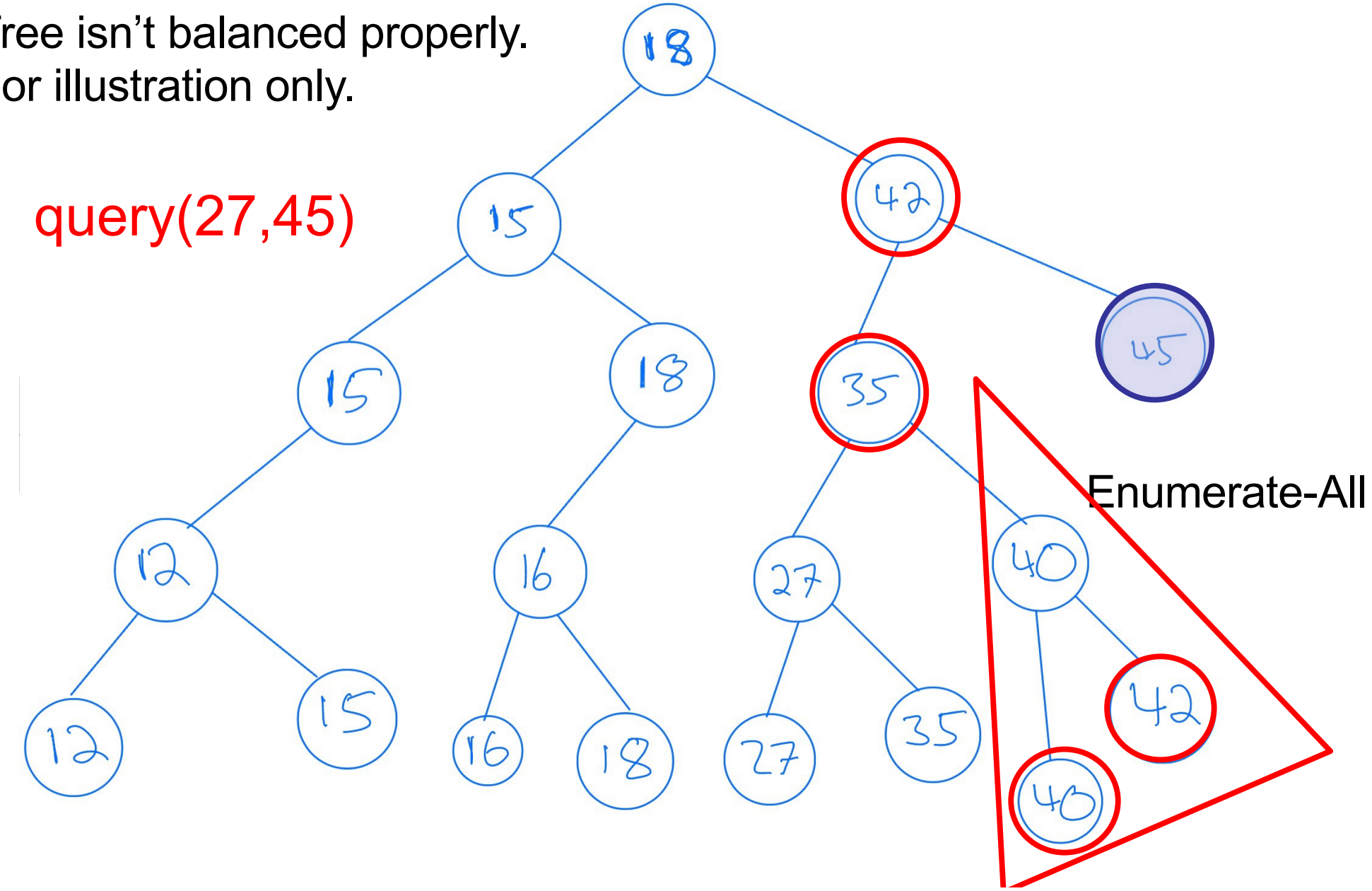


LeftTraversal

Another example

Beware:
Tree isn't balanced properly.
For illustration only.

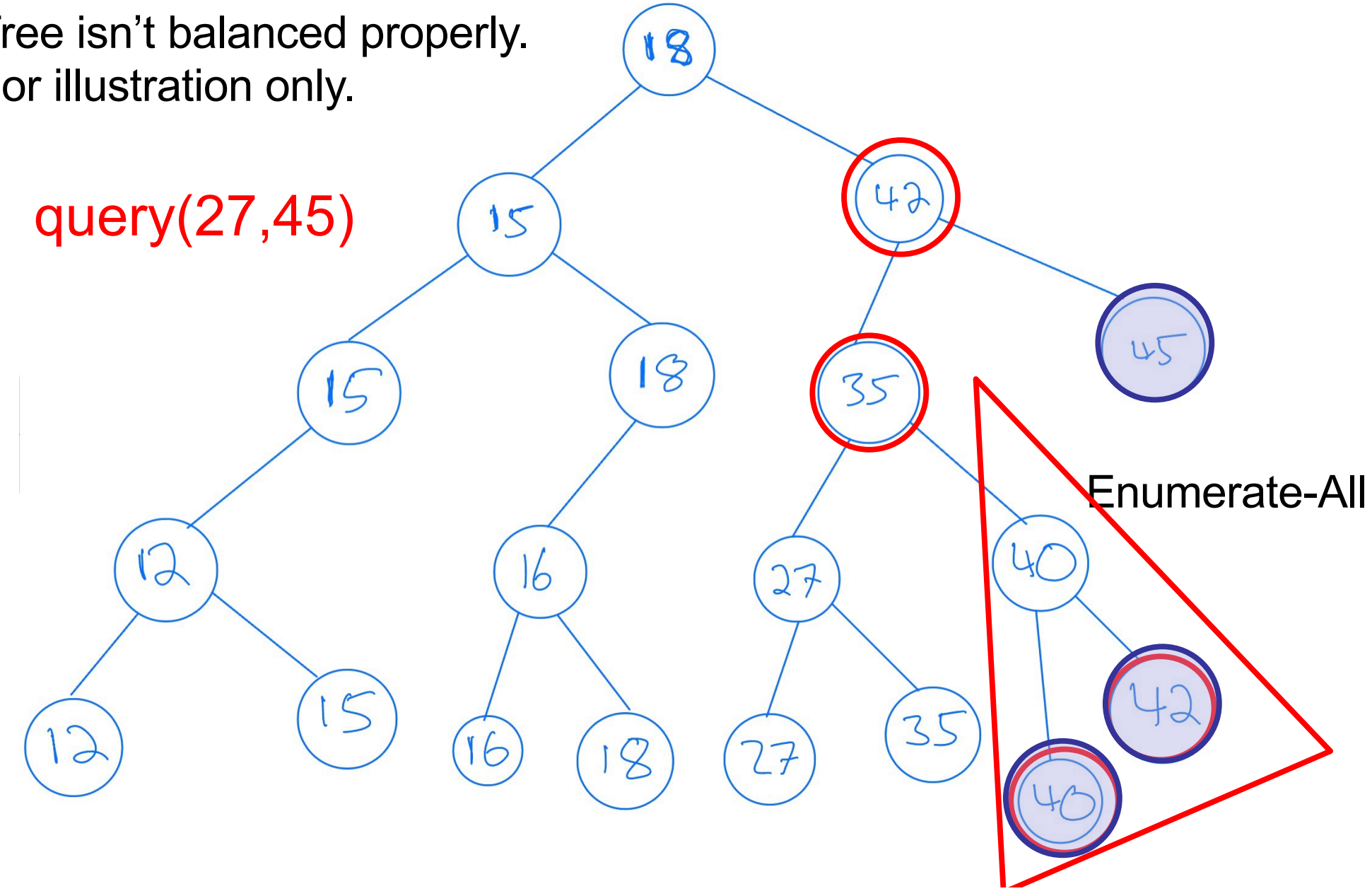
query(27,45)



Another example

Beware:
Tree isn't balanced properly.
For illustration only.

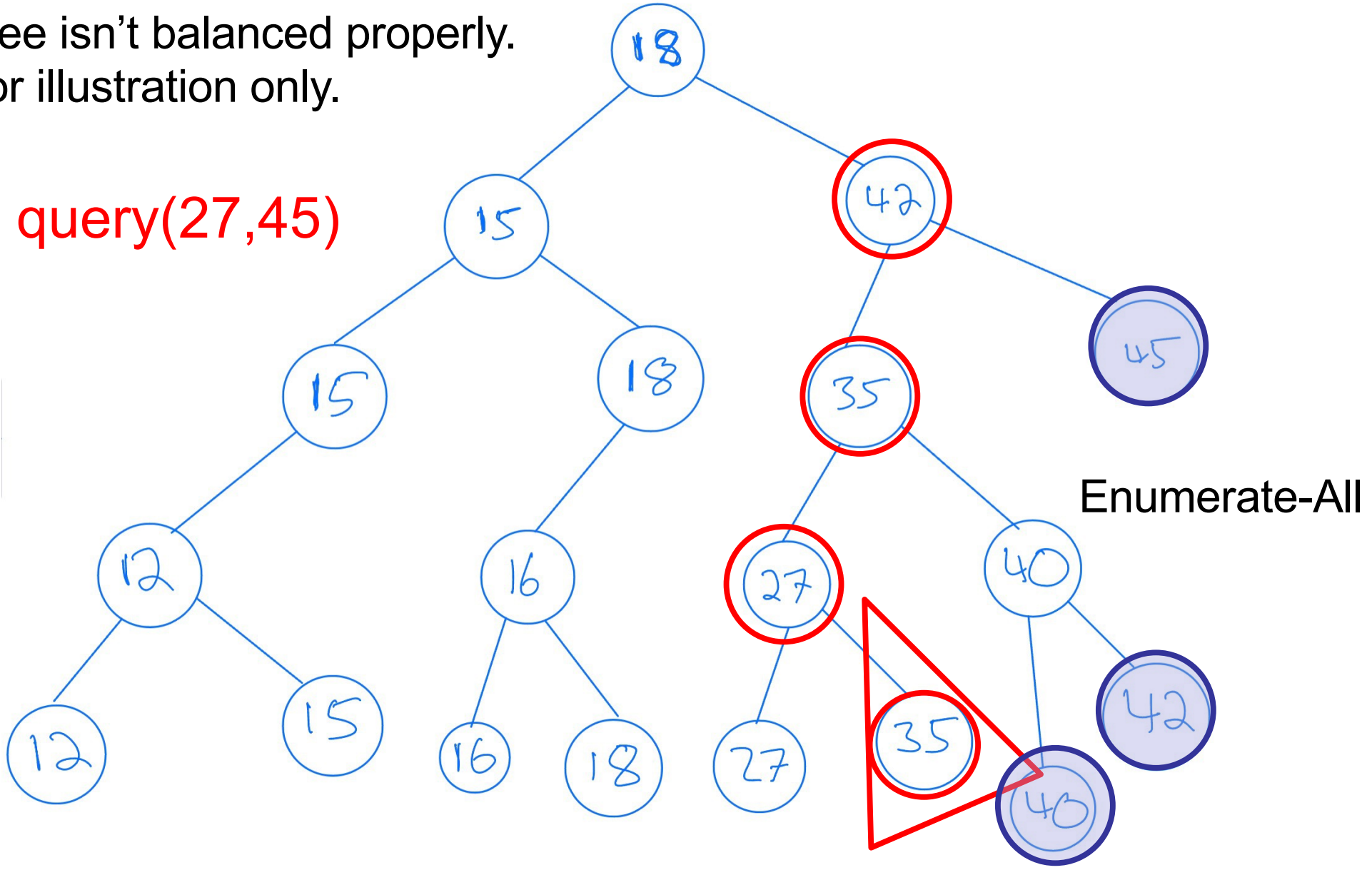
query(27,45)



Another example

Beware:
Tree isn't balanced properly.
For illustration only.

query(27,45)

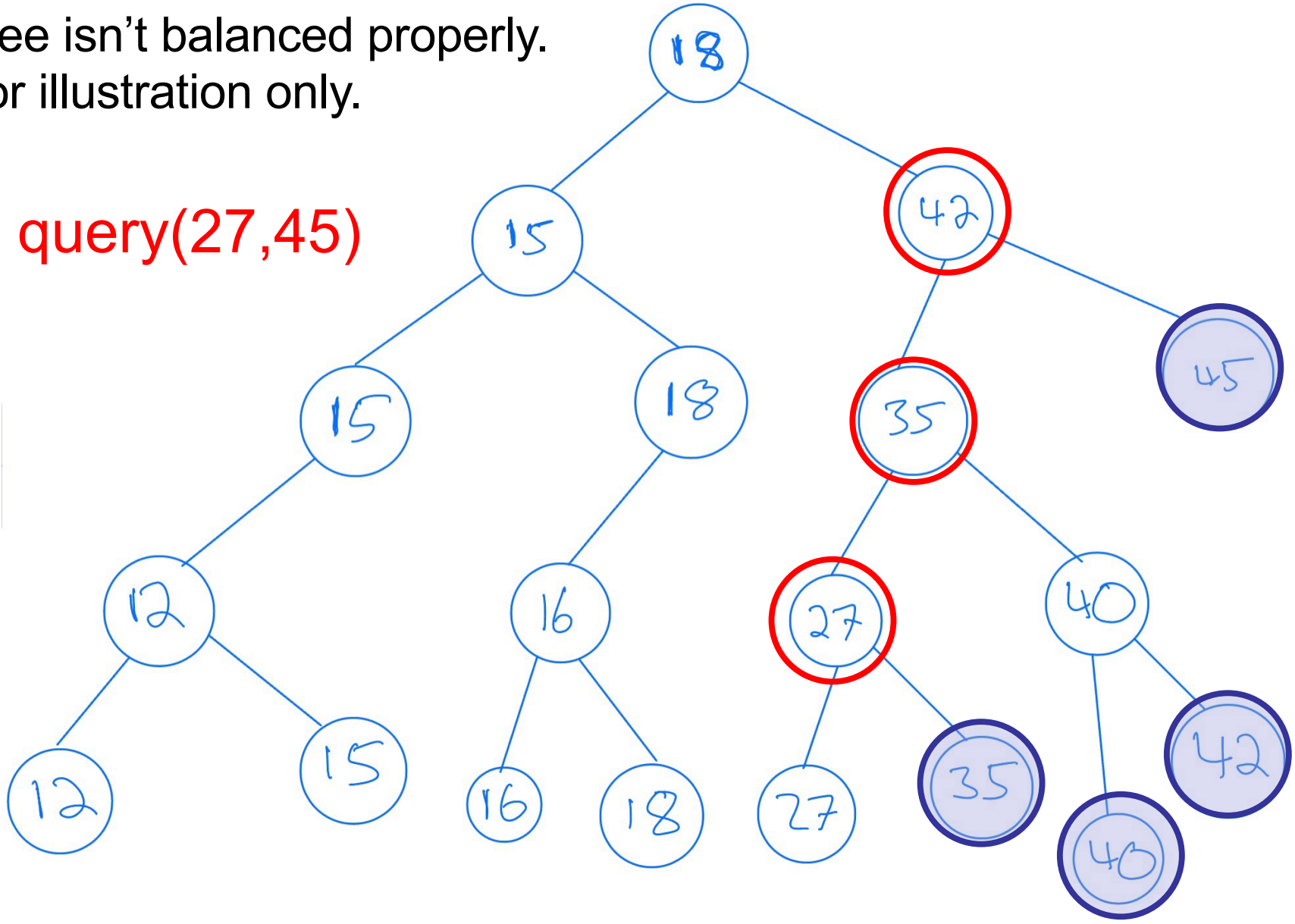


Enumerate-All

Another example

Beware:
Tree isn't balanced properly.
For illustration only.

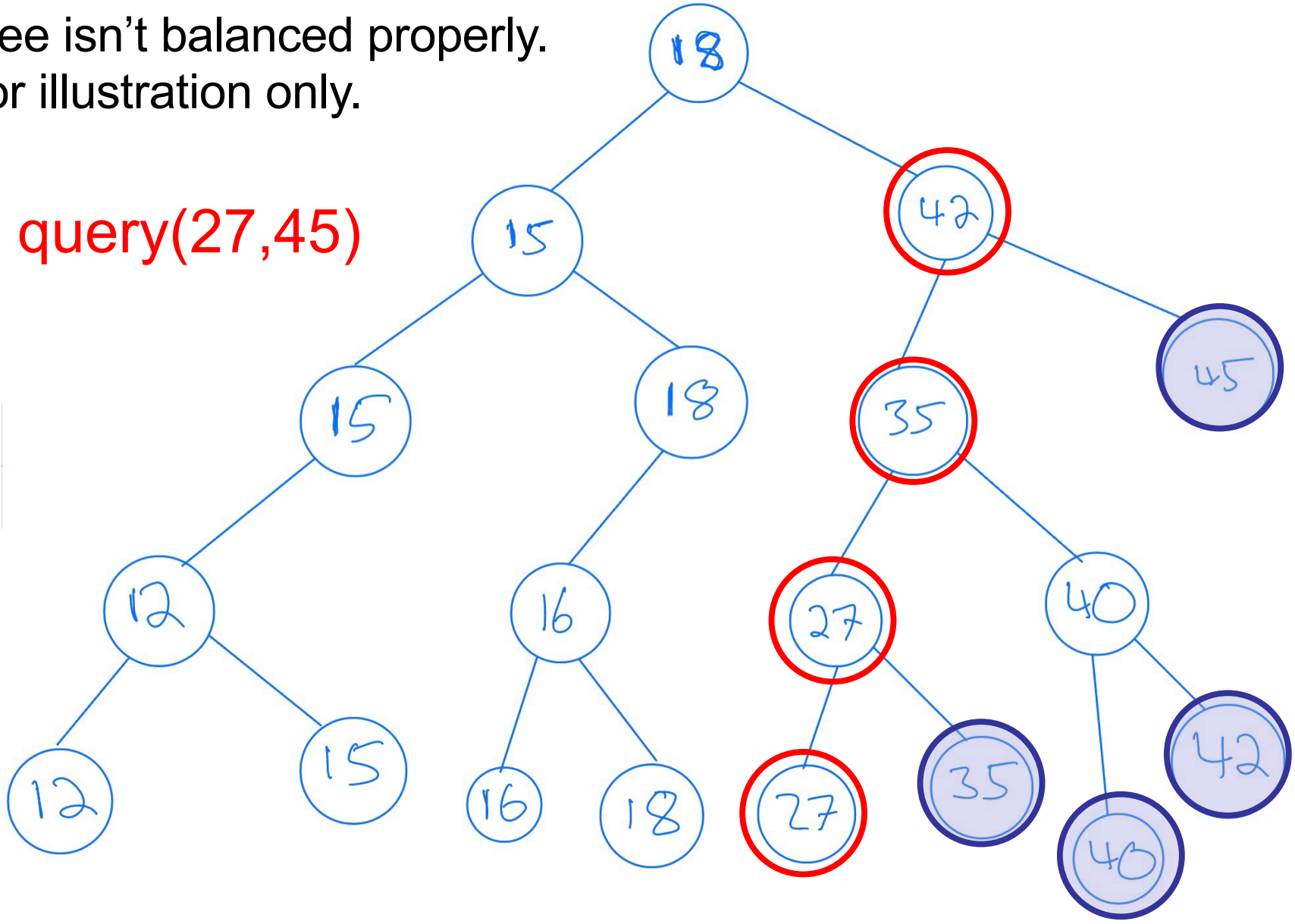
query(27,45)



Another example

Beware:
Tree isn't balanced properly.
For illustration only.

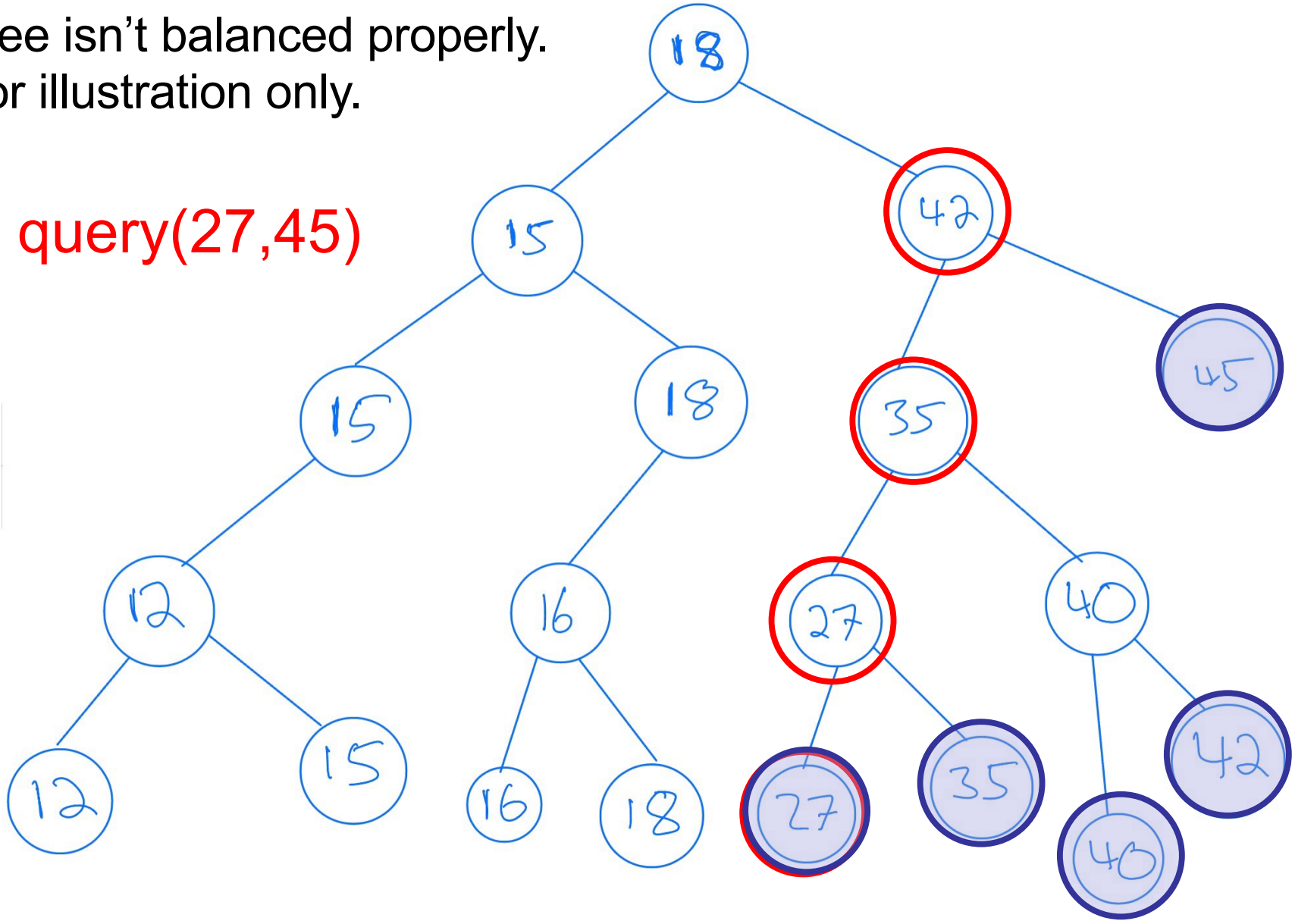
query(27,45)



Another example

Beware:
Tree isn't balanced properly.
For illustration only.

query(27,45)



One Dimensional Range Queries

Algorithm:

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

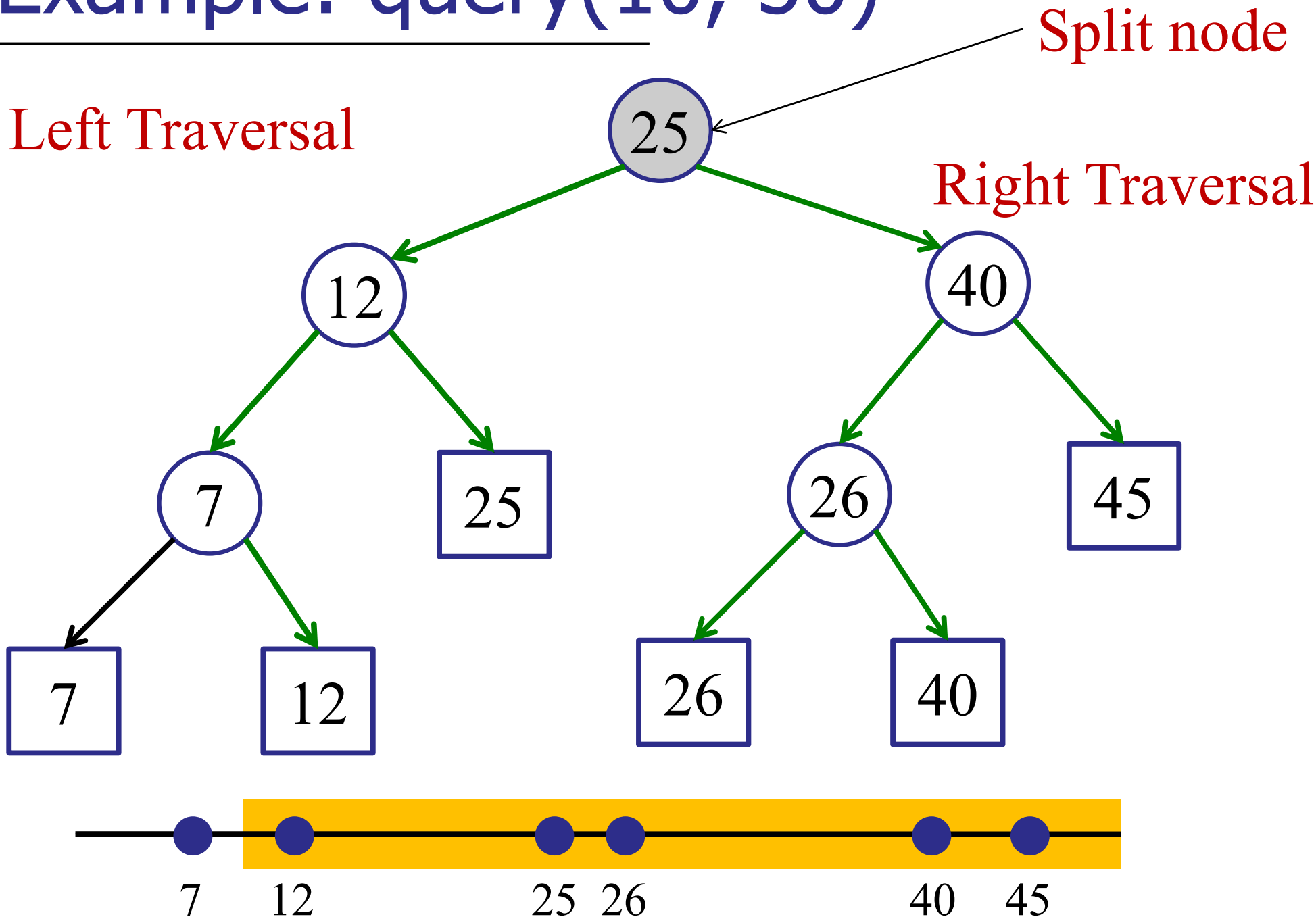
At every step, we either:

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

Counting:

1. Recurse at most O(log n) times (i.e., option 2).
2. How expensive is "output all sub-tree" (i.e., option 1)?

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 (i.e., option 2).
2. How expensive is "output all sub-tree" (i.e., option 1)?

➔ O(k), where k is number of items found.

Analysis

Query time complexity:

$O(k + \log n)$

where k is the number of points found.

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

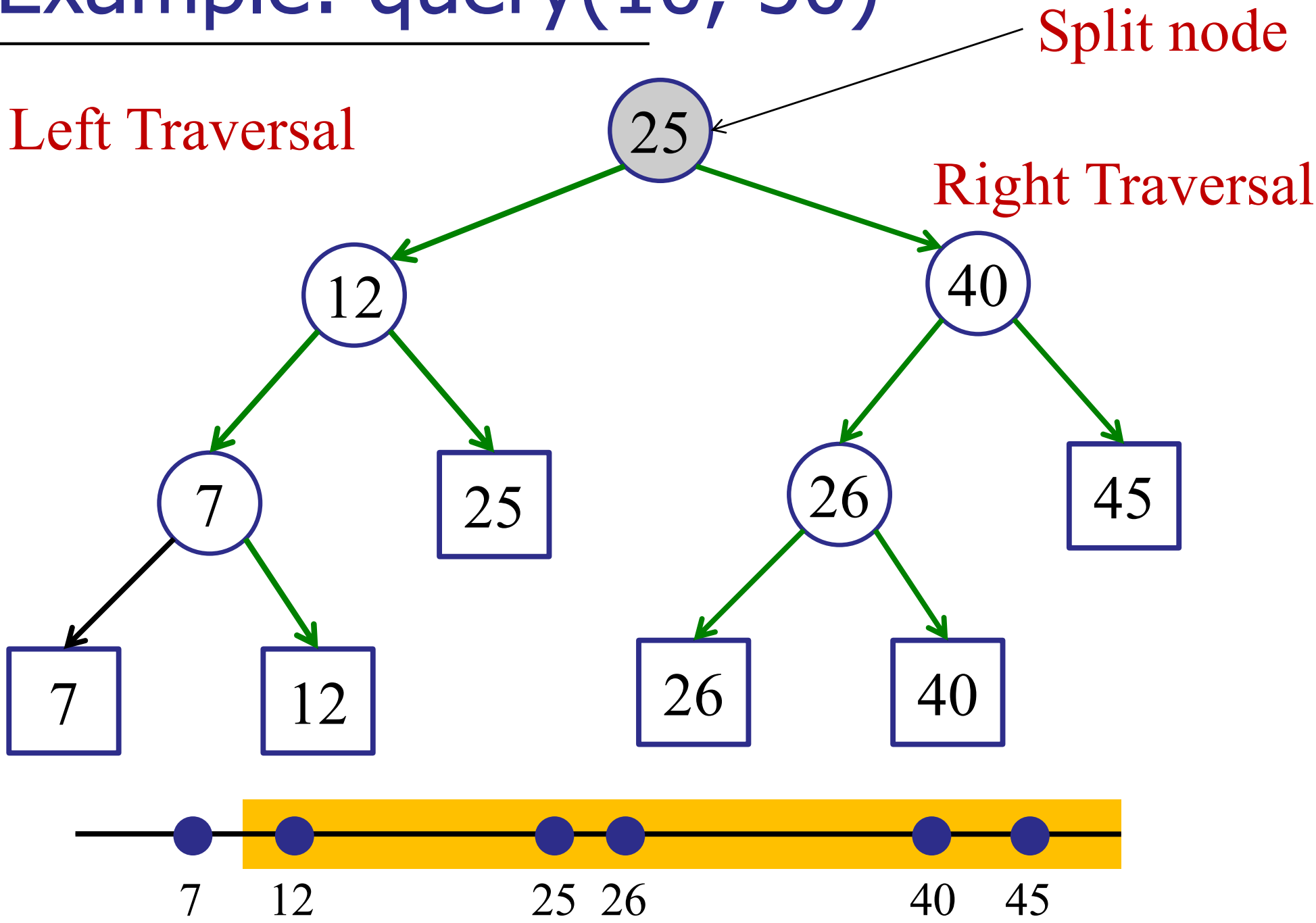
What if you just want to know *how many* points are in the range?

- 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)



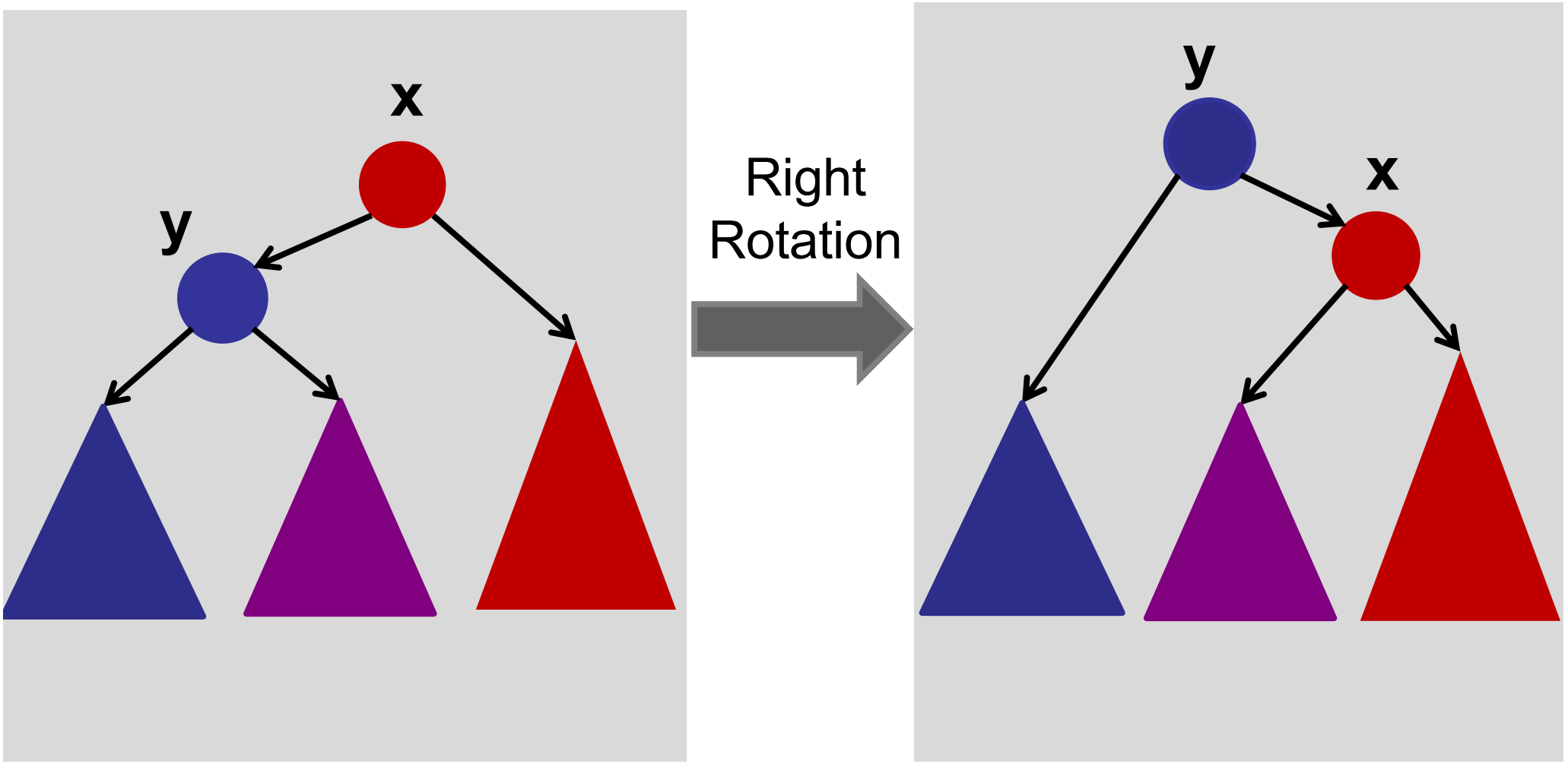
1D Range Tree

Done??

One Dimensional Range Queries

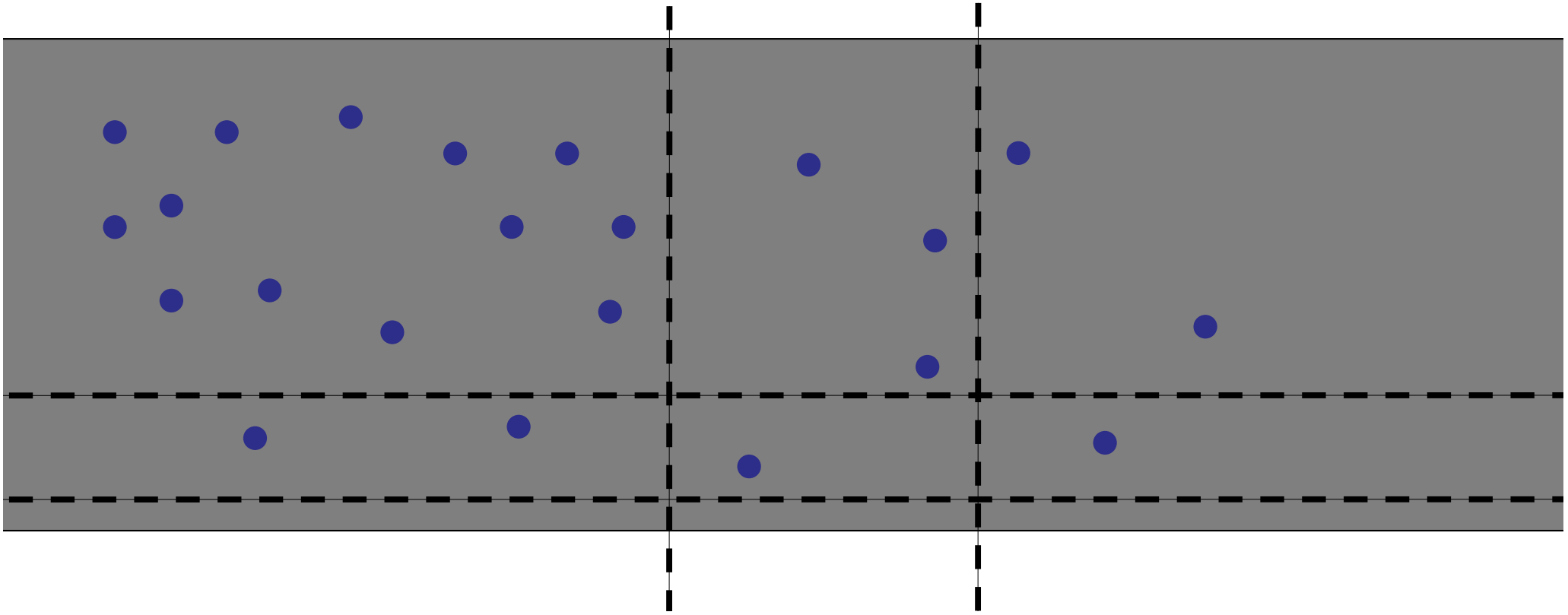
What about dynamic updates?

- Need to fix rotations! Any changes needed?



Two Dimensional Range Tree

Ex: search for all points between dashed lines.



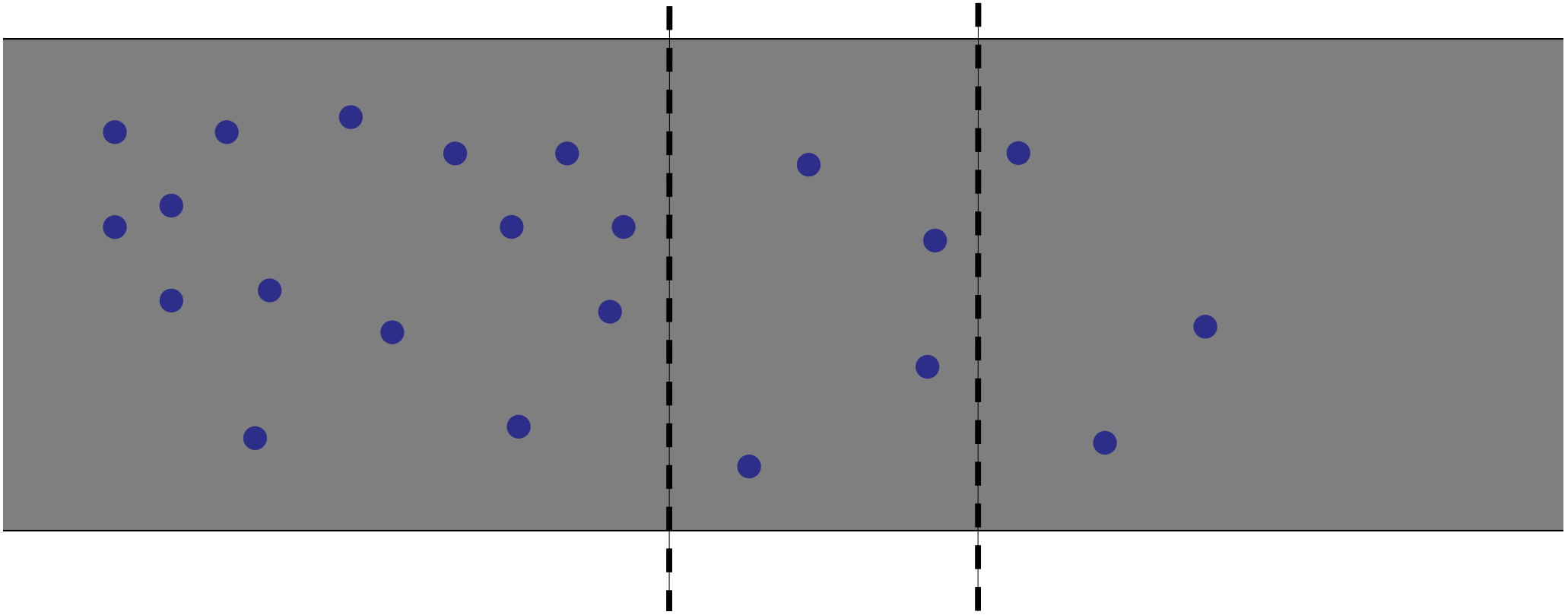
Two Dimensional Range Tree

Idea:

Step 1: Create a 1d-range-tree on the x-coords.

Step 2: Enumerate all points in range, sort.

Step 3: Return only points in the y-range.



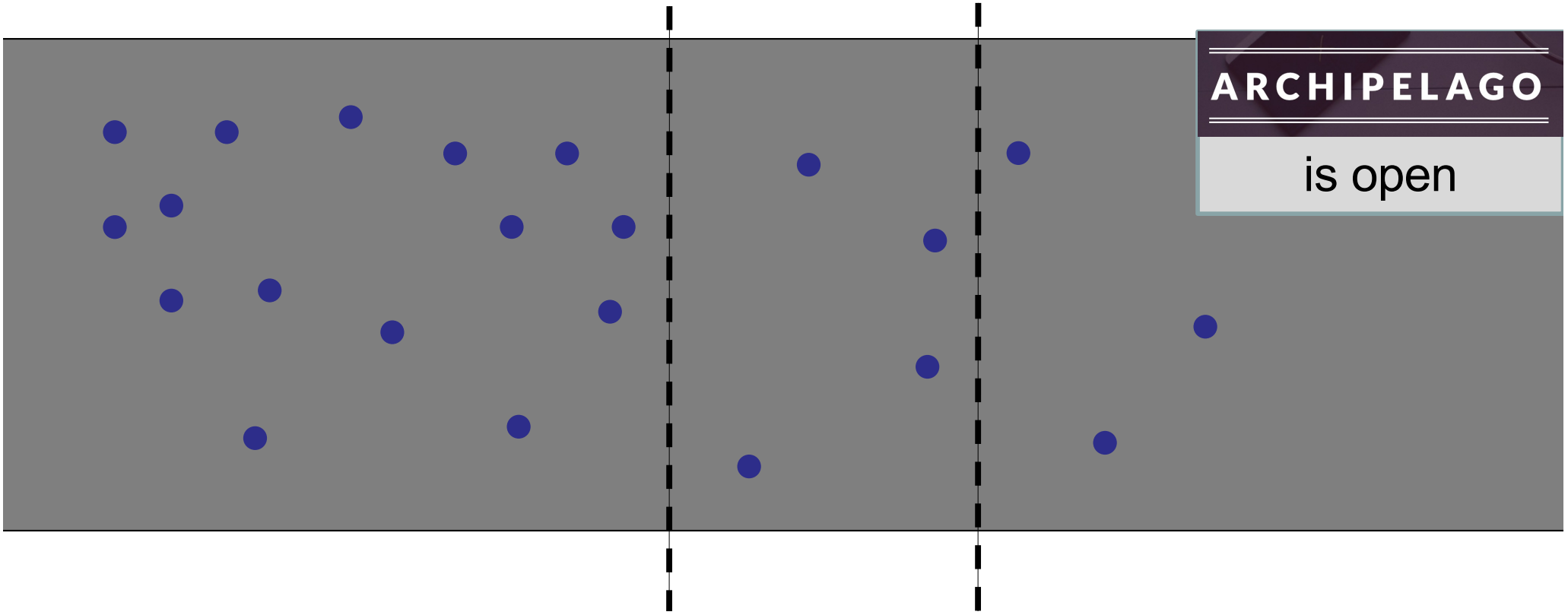
Two Dimensional Range Tree

Idea:

Step 1: Create a 1d-range-tree on the x-coords.

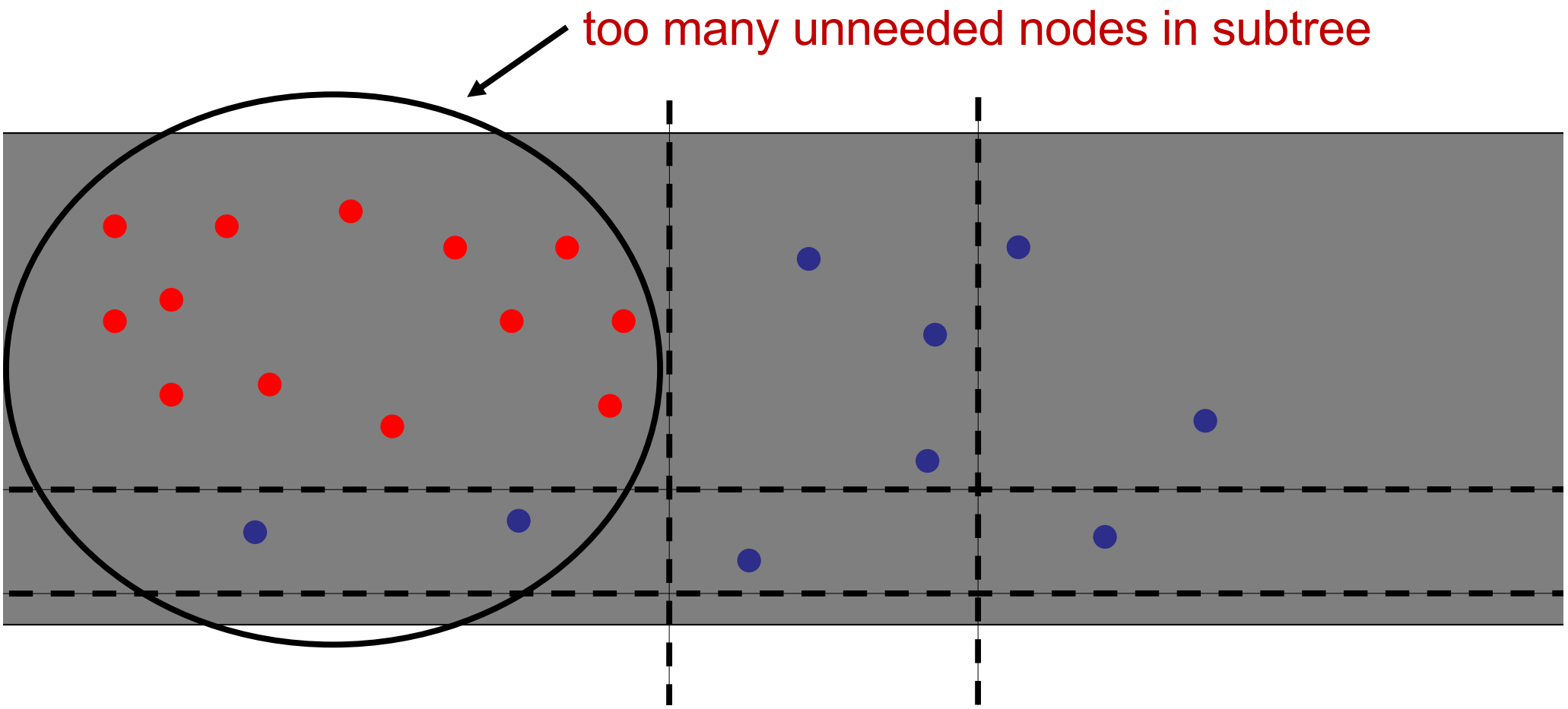
Step 2: Enumerate all points in range, sort.

Step 3: Return only points in the y-range.



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.



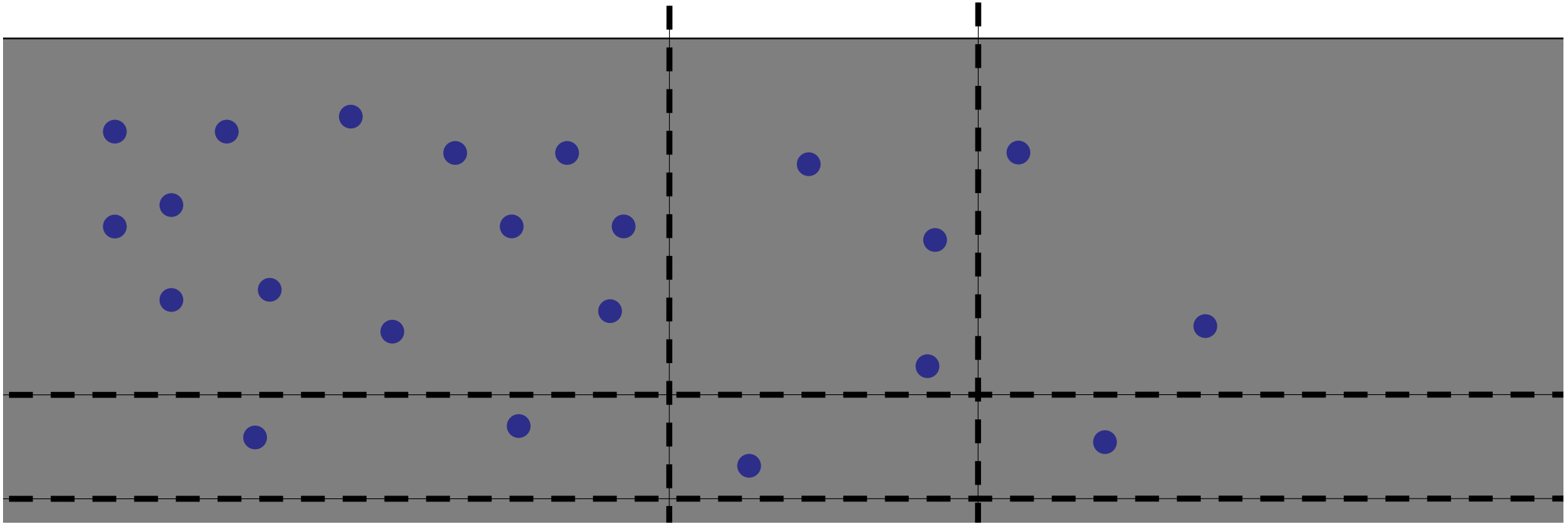
One Dimensional Range Queries

```
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!

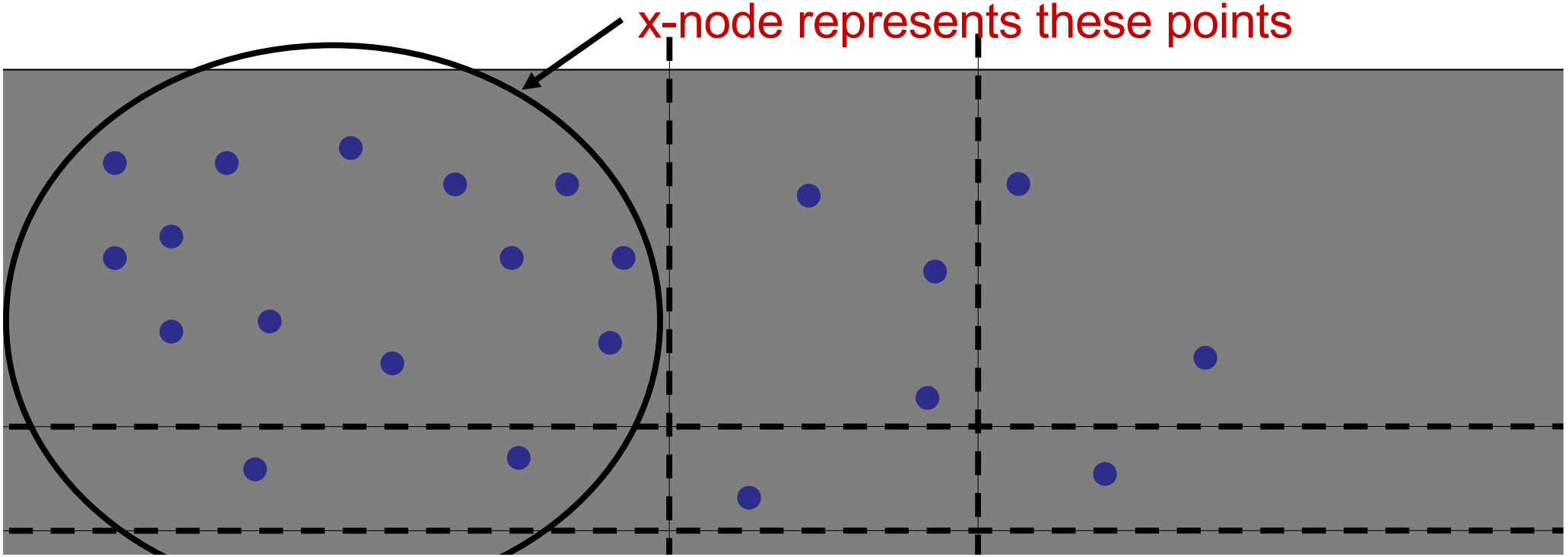
- Each node in the x-tree has a set of points in its sub-tree.
- Store a y-tree at each x-node containing all the points in the sub-tree.



Two Dimensional Range Tree

Solution: Augment!

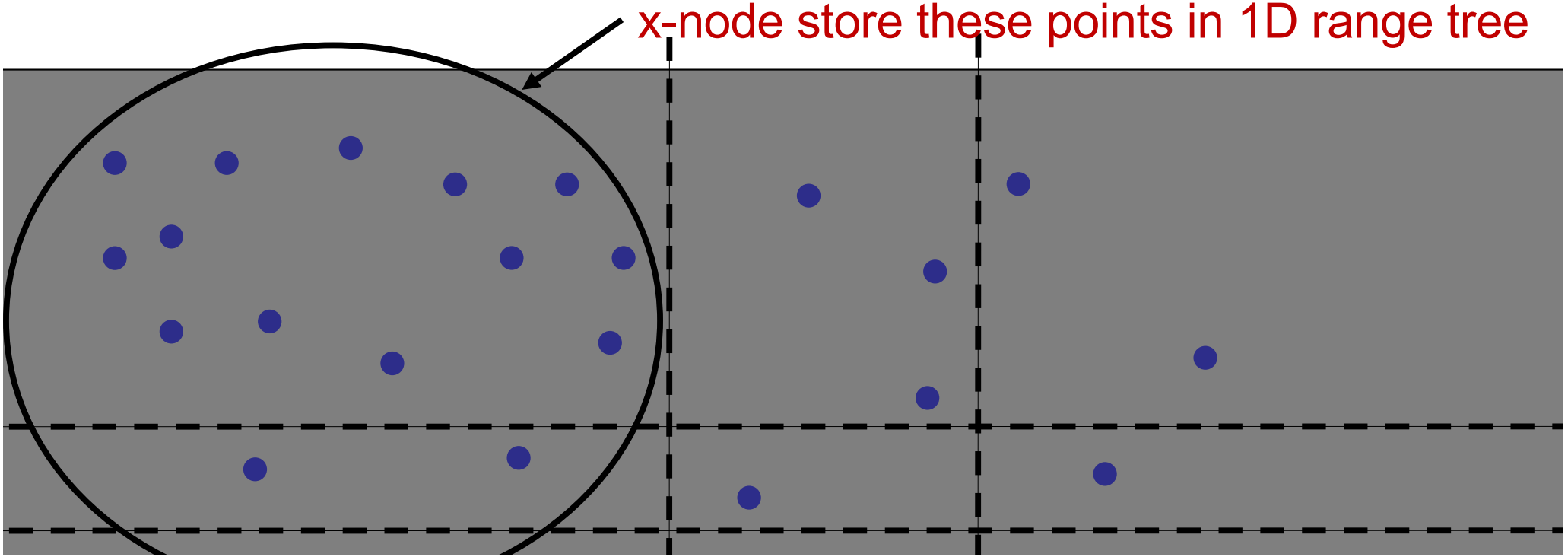
- Each node in the x-tree has a set of points in its sub-tree.
- Store a y-tree at each x-node containing all the points in the sub-tree.



Two Dimensional Range Tree

Solution: Augment!

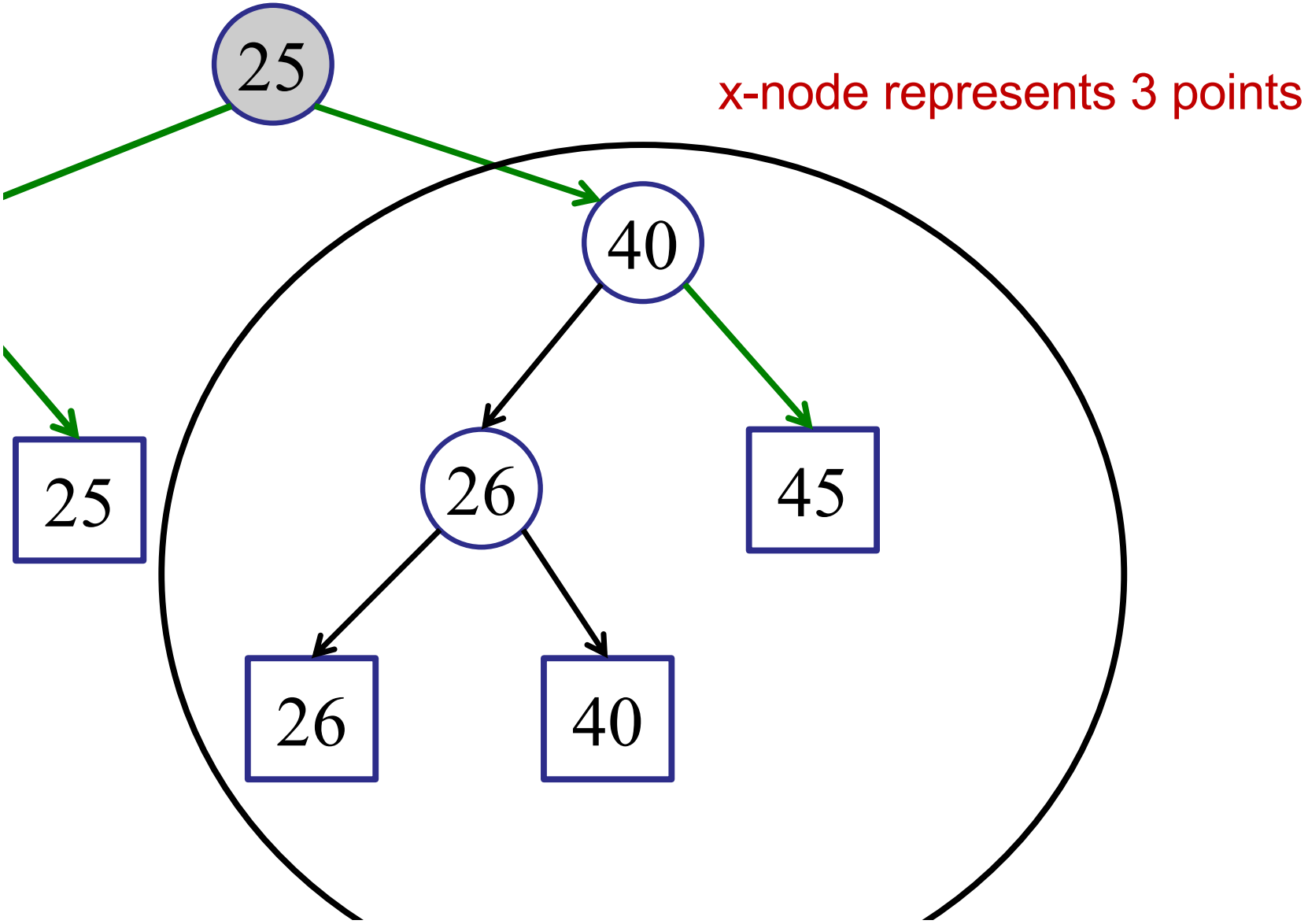
- Each node in the x-tree has a set of points in its sub-tree.
- Store a y-tree at each x-node containing all the points in the sub-tree.



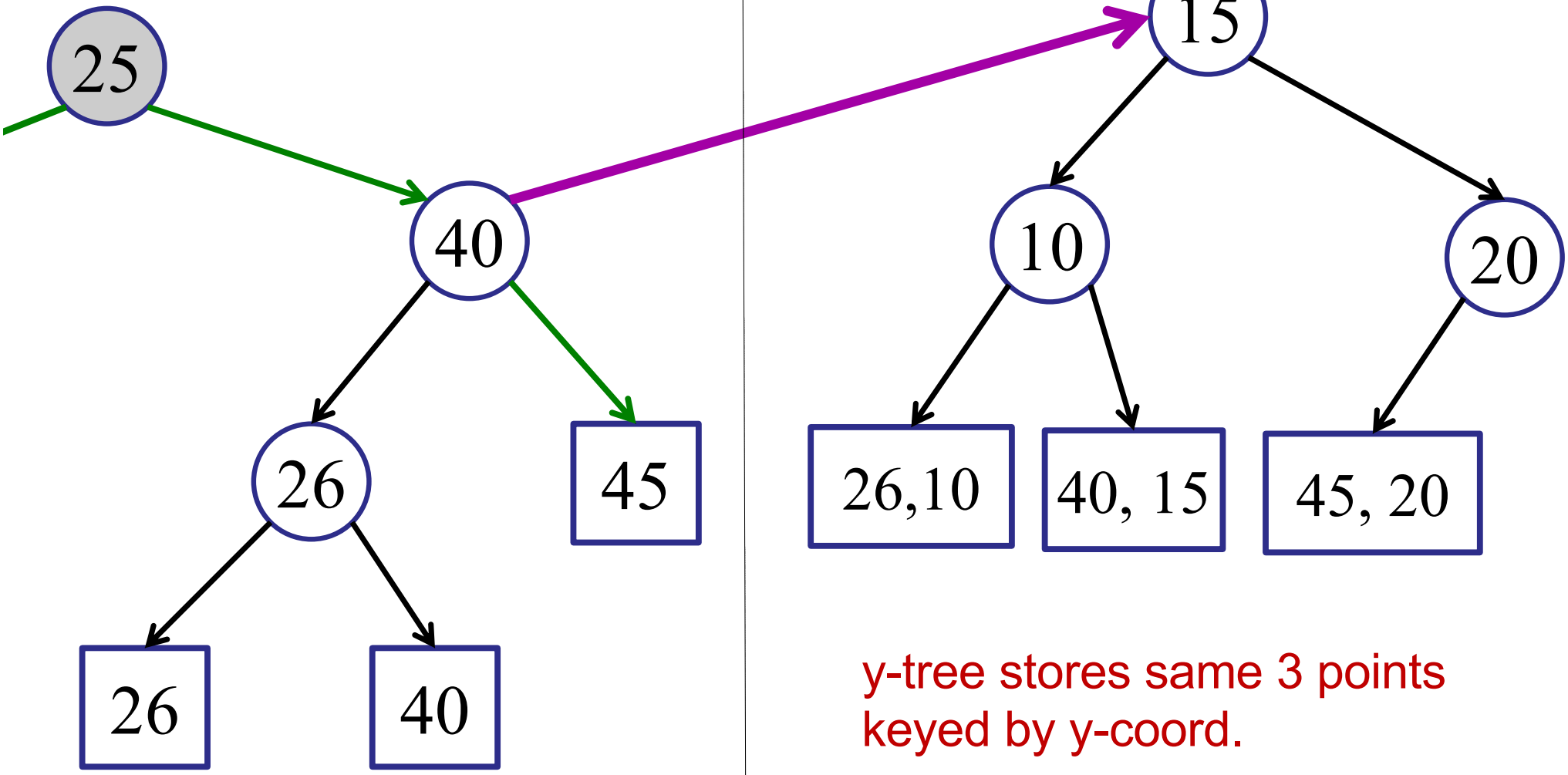
One Dimensional Range Queries

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

Example:



Example:

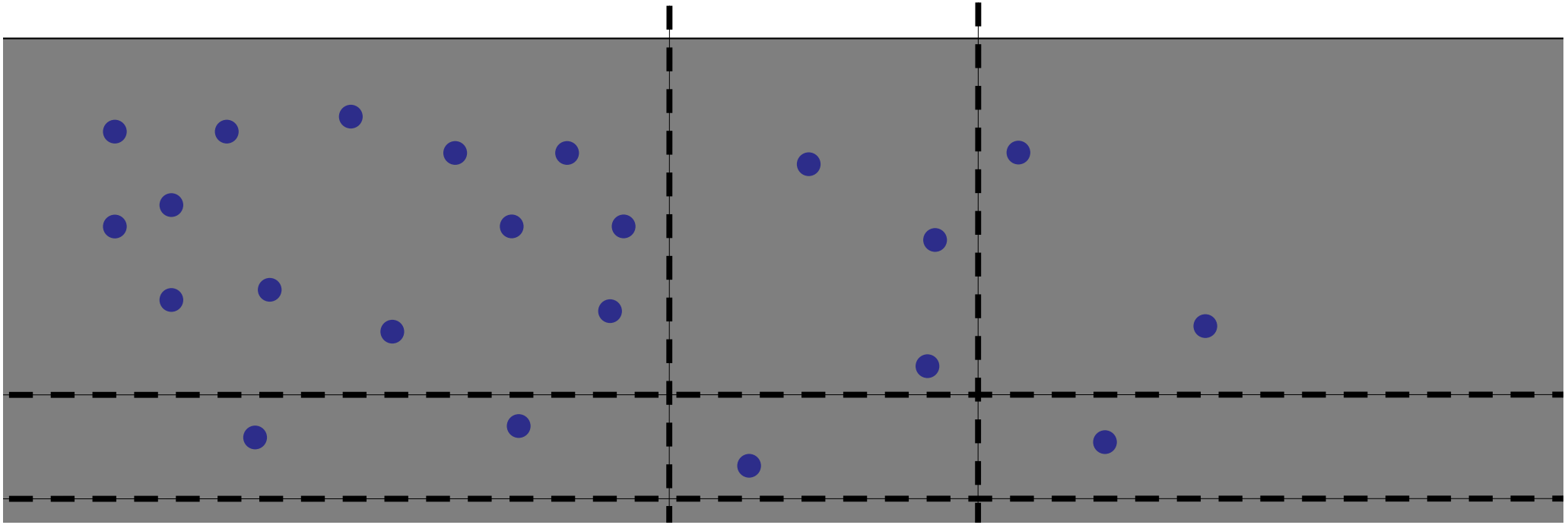


y-tree stores same 3 points keyed by y-coord.

Two Dimensional Range Tree

Idea:

- Build an x-tree using only x-coordinates.
- For every node in the x-tree, build a y-tree out of nodes in subtree using only y-coordinates.



Analysis

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

- $O(\log n)$ to find split node.
- $O(\log n)$ recursing 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 O(log n) levels.

➔ Each node appears in at most O(log n) y-trees.

- The rest of the x-tree takes O(n) space.

Analysis

Building the tree: O(n log n)

- Tricky…
- Left as a puzzle… ☺

Challenge of the Day…

Dynamic Trees

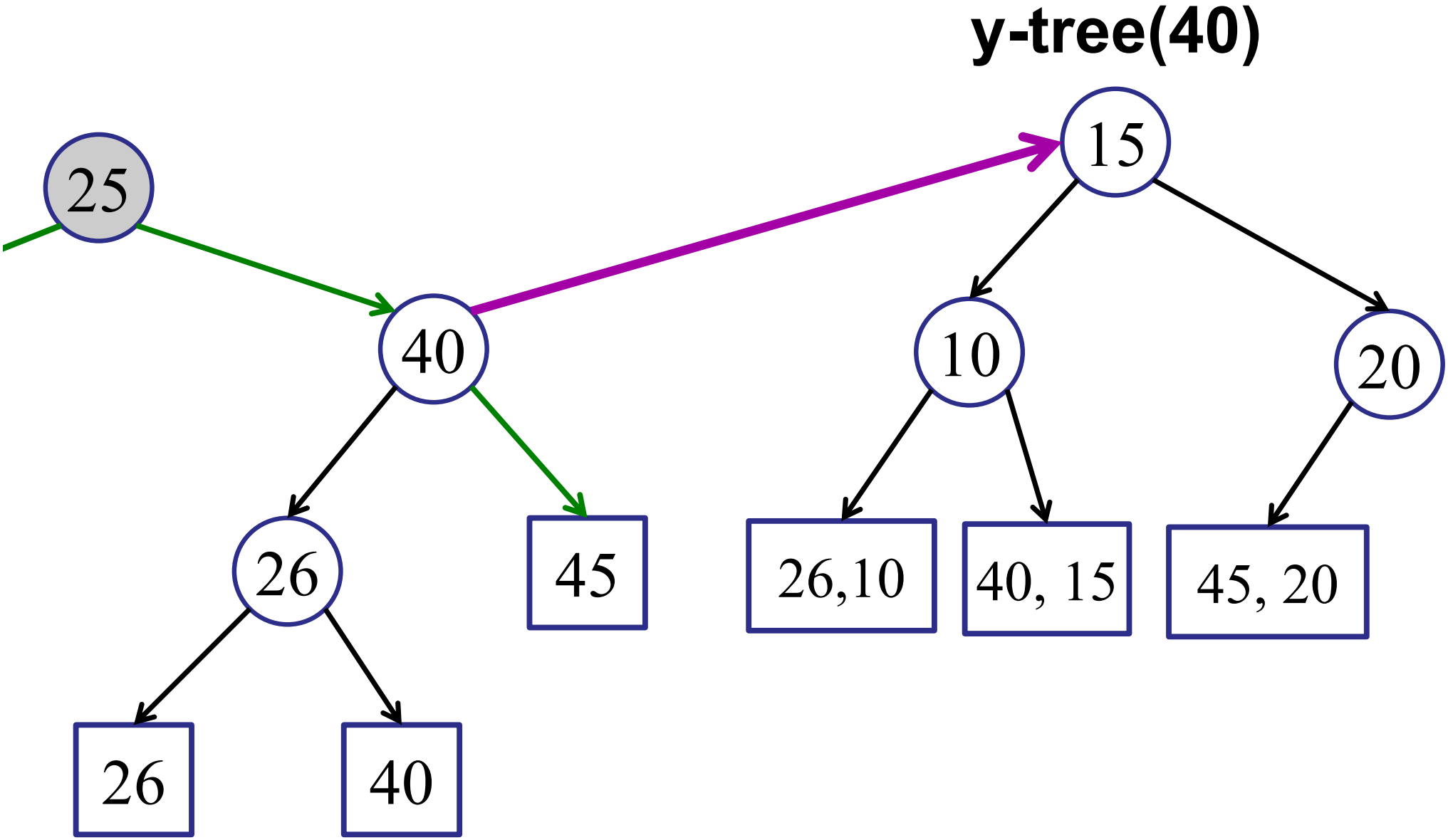
What about inserting/deleting nodes?

Dynamic Trees

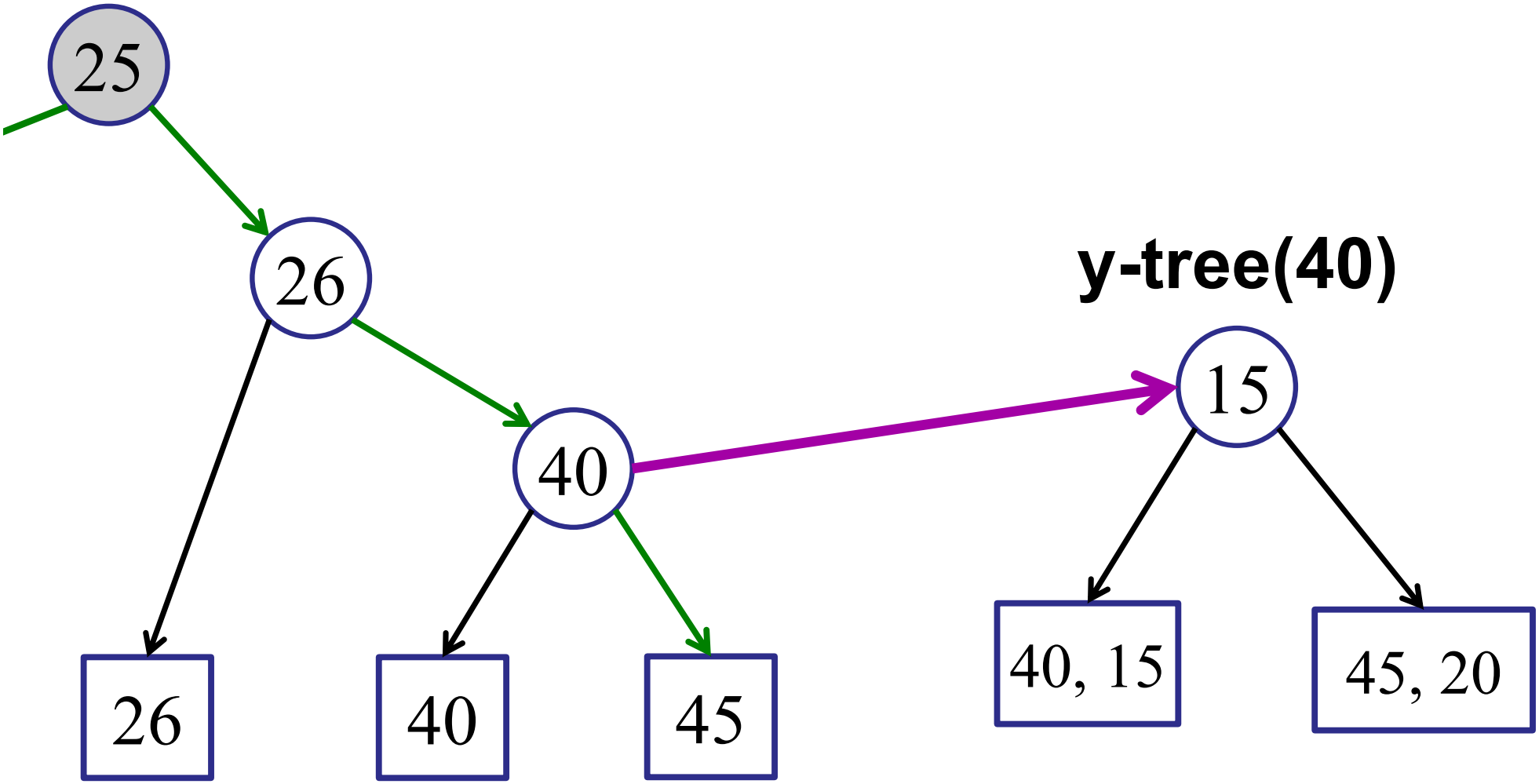
What about inserting/deleting nodes?

- 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:



Example:



Dynamic Trees

Moral:

- Static 2d-range trees support efficient operations.
- We do not support insert/delete operations in 2d-range trees because rotations would be too expensive.

Augmenting data structures has to be done carefully!

Many useful augmentations cannot be supported efficiently!

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-dimensionsal 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-dimensionsal 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.
- Supports more efficient updates
- Often better in practice.
- Good for a variety of queries (e.g., nearest neighbor).

Augmented BSTs

Three examples of augmenting BSTs

1. Order Statistics

2. Intervals

3. Orthogonal Range Searching

Priority Queue

Maintain a set of prioritized objects:

- insert: add a new object with a specified priority
- extractMin: remove and return the object with minimum valued priority

Ex: Scheduling

- Find next task to do
- Earliest deadline first

Task	Due date
CS4234 PS8	March 31
Study for Exam	April 4
Wash clothes	April 6
See friends	May 12

Abstract Data Type

Priority Queue

interface	**IPriorityQueue<Key, Priority>**	
void	insert(Key k, Priority p)	*insert k with priority p*
Data	extractMin()	*remove key with minimum priority*
void	decreaseKey(Key k, Priority p)	*reduce the priority of key k to priority p*
boolean	contains(Key k)	*does the priority queue contain key k?*
boolean	isEmpty()	*is the priority queue empty?*

Notes:

Assume data items are unique.

Abstract Data Type

Max Priority Queue

interface	**IMaxPriorityQueue<Key, Priority>**	
void	insert(Key k, Priority p)	*insert k with priority p*
Data	extractMax()	*remove key with maximum priority*
void	increaseKey(Key k, Priority p)	*increase the priority of key k to priority p*
boolean	contains(Key k)	*does the priority queue contain key k?*
boolean	isEmpty()	*is the priority queue empty?*

Notes:

Assume data items are unique.

Priority Queue

Sorted array

- insert: O(n)
  - Find insertion location in array.
  - Move everything over.
- extractMax: O(1)
  - Return largest element in array

object	G	C	Y	Z	B	D	F	J	L
priority	2	7	9	13	22	26	29	31	45

Priority Queue

Unsorted array

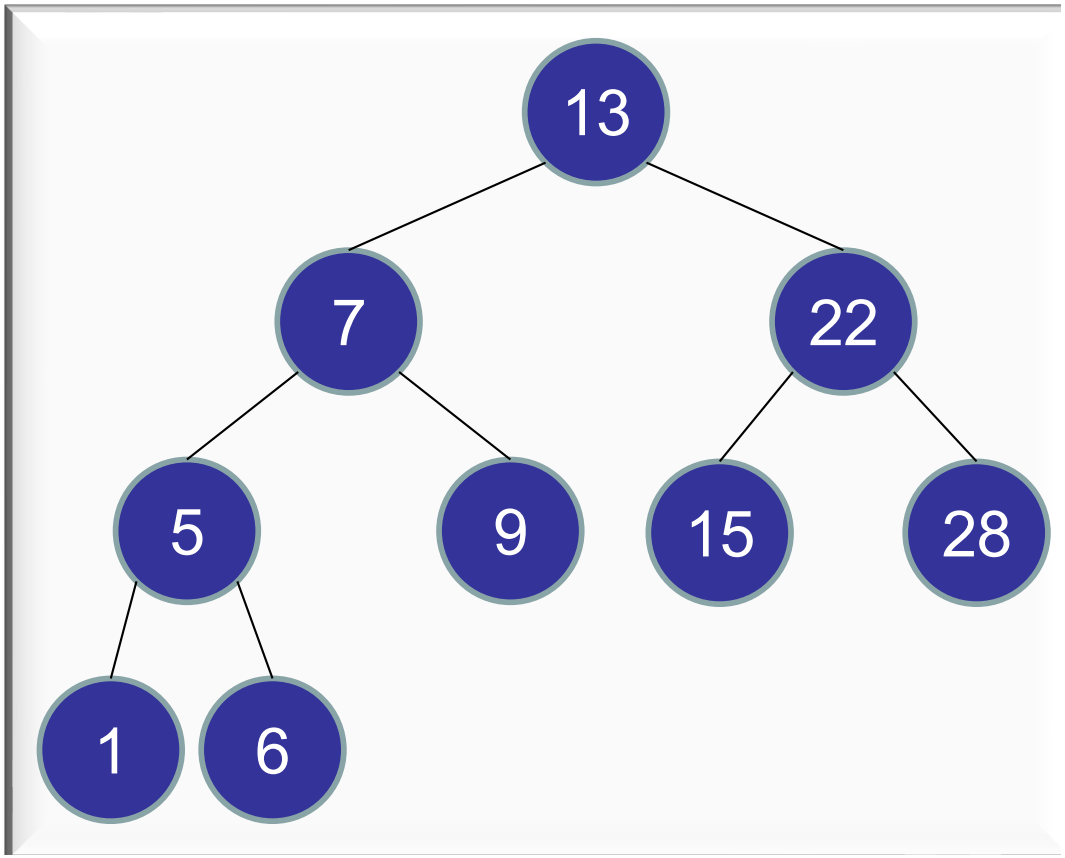
- insert: O(1)
  - Add object to end of list
- extractMax: O(n)
  - Search for largest element in array.
  - Remove and move everything over.

object	G	L	D	Z	B	J	F	C	Y
priority	2	45	26	13	22	31	29	7	9

Priority Queue

AVL Tree (indexed by priority)

- insert: O(log n)
  - Insert object in tree
- extractMax: O(log n)
  - Find maximum item.
  - Delete it from tree.



Priority Queue

Other operations:

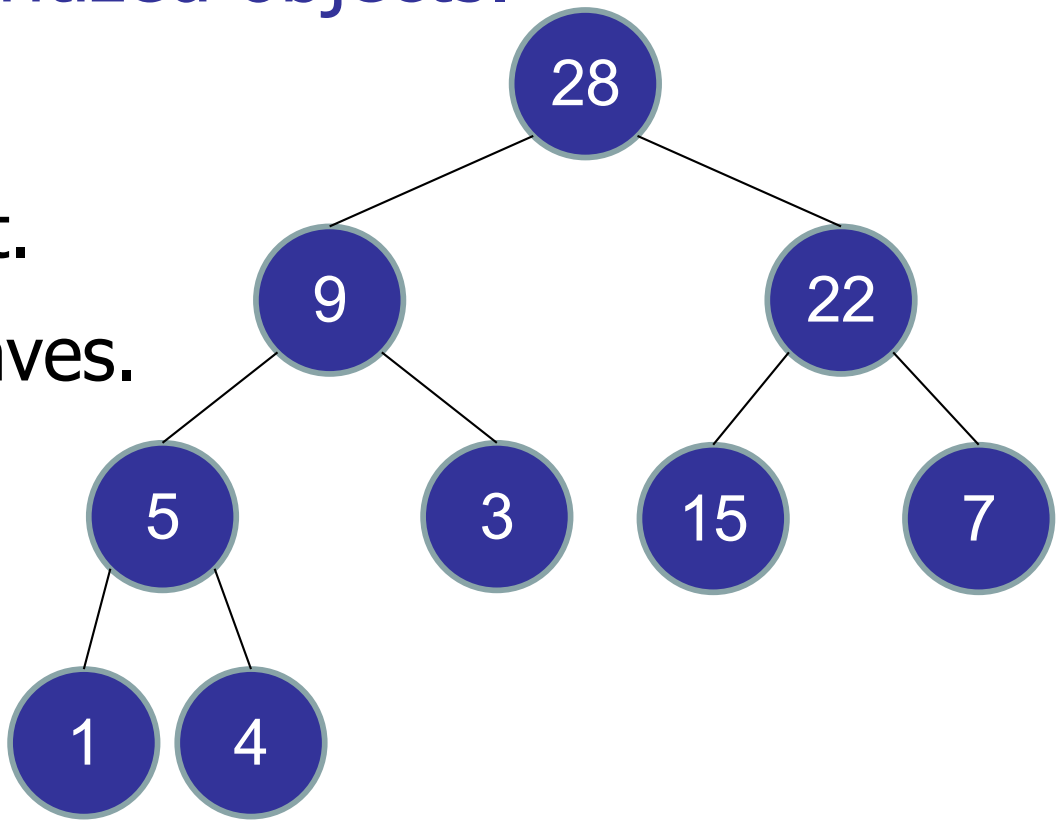
Use second dictionary indexed by key

- contains:
  - Look up key in dictionary.
- decreaseKey:
  - Look up key in dictionary.
  - Remove object from AVL tree.
  - Re-insert object into AVL tree.

Heap

(aka Binary Heap or MaxHeap)

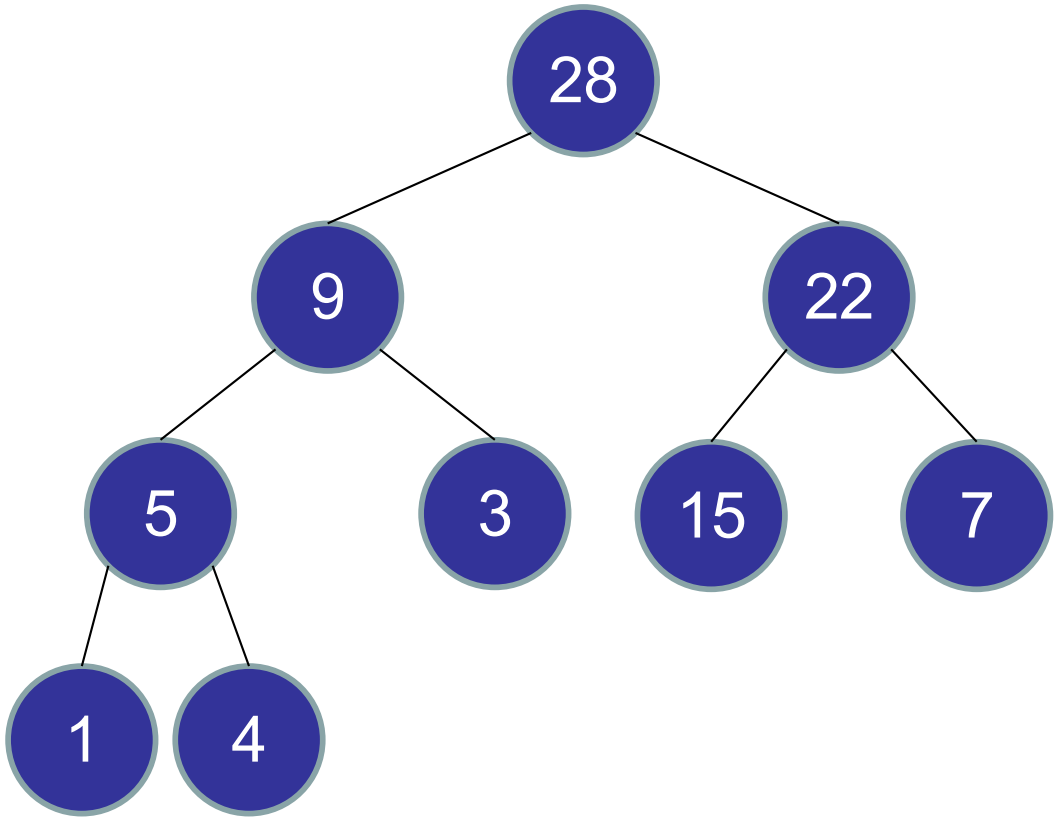
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.



Two Properties of a Heap

1. Heap Ordering

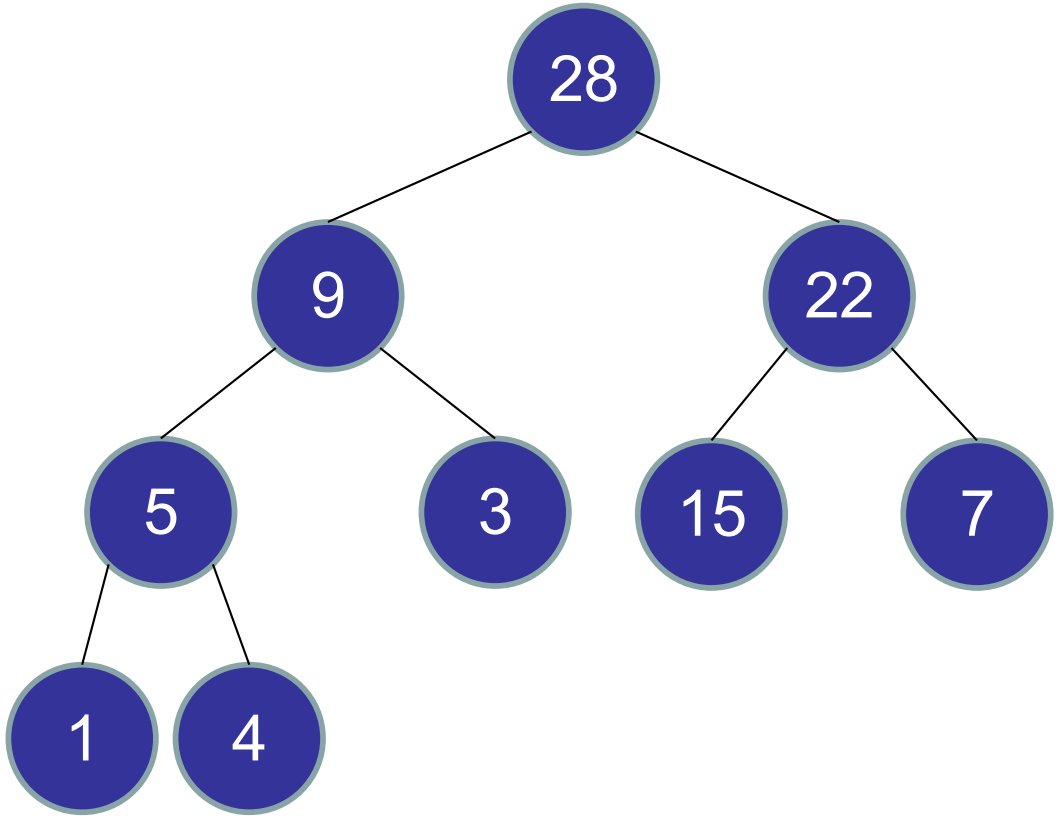
```
priority[parent] >= priority[child]
```



Note: not a binary search tree.

Two Properties of a Heap

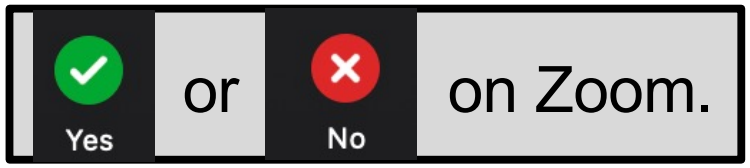
2. Complete binary tree
   - Every level is full, except possibly the last.
   - All nodes are as far left as possible.

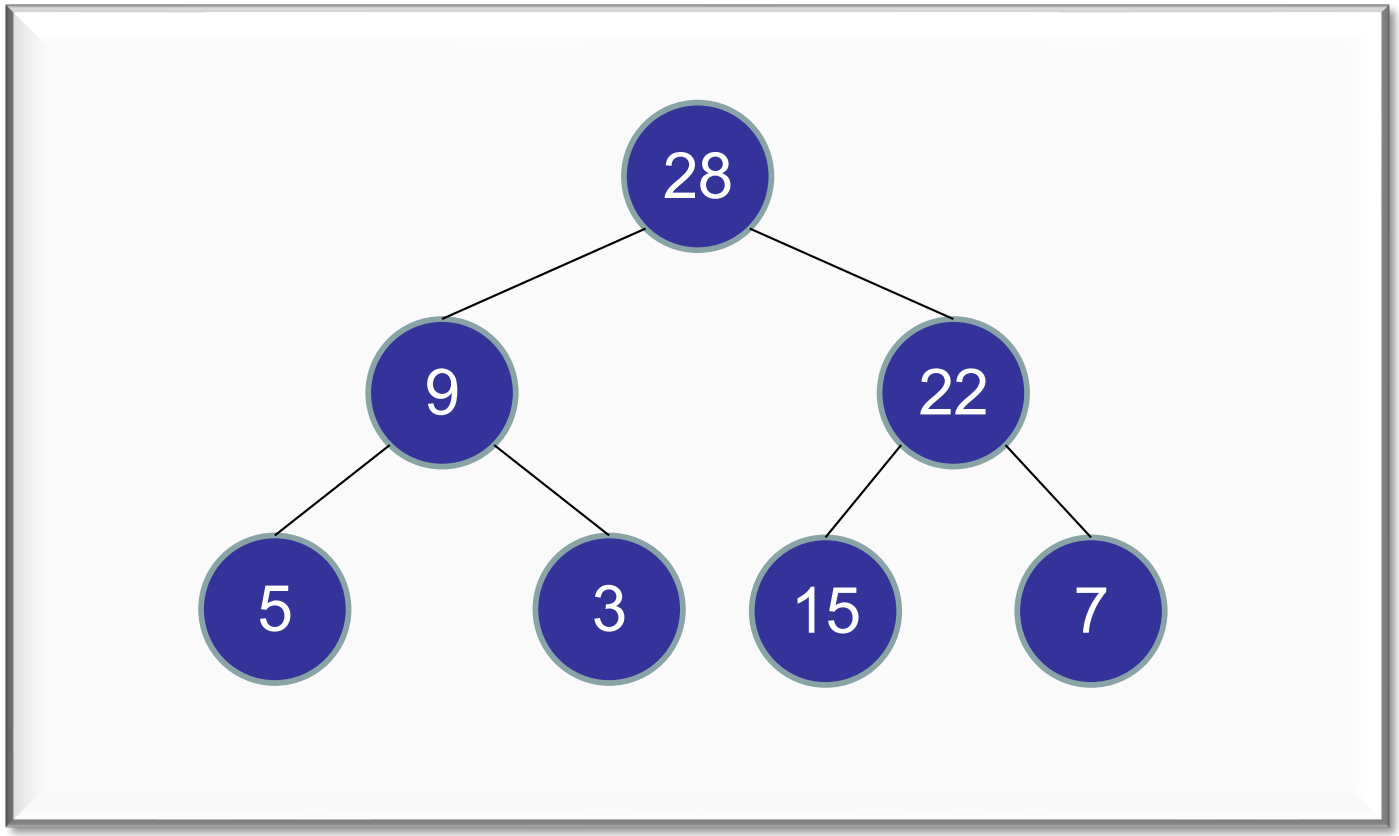


Is it a heap?

✓1. Yes

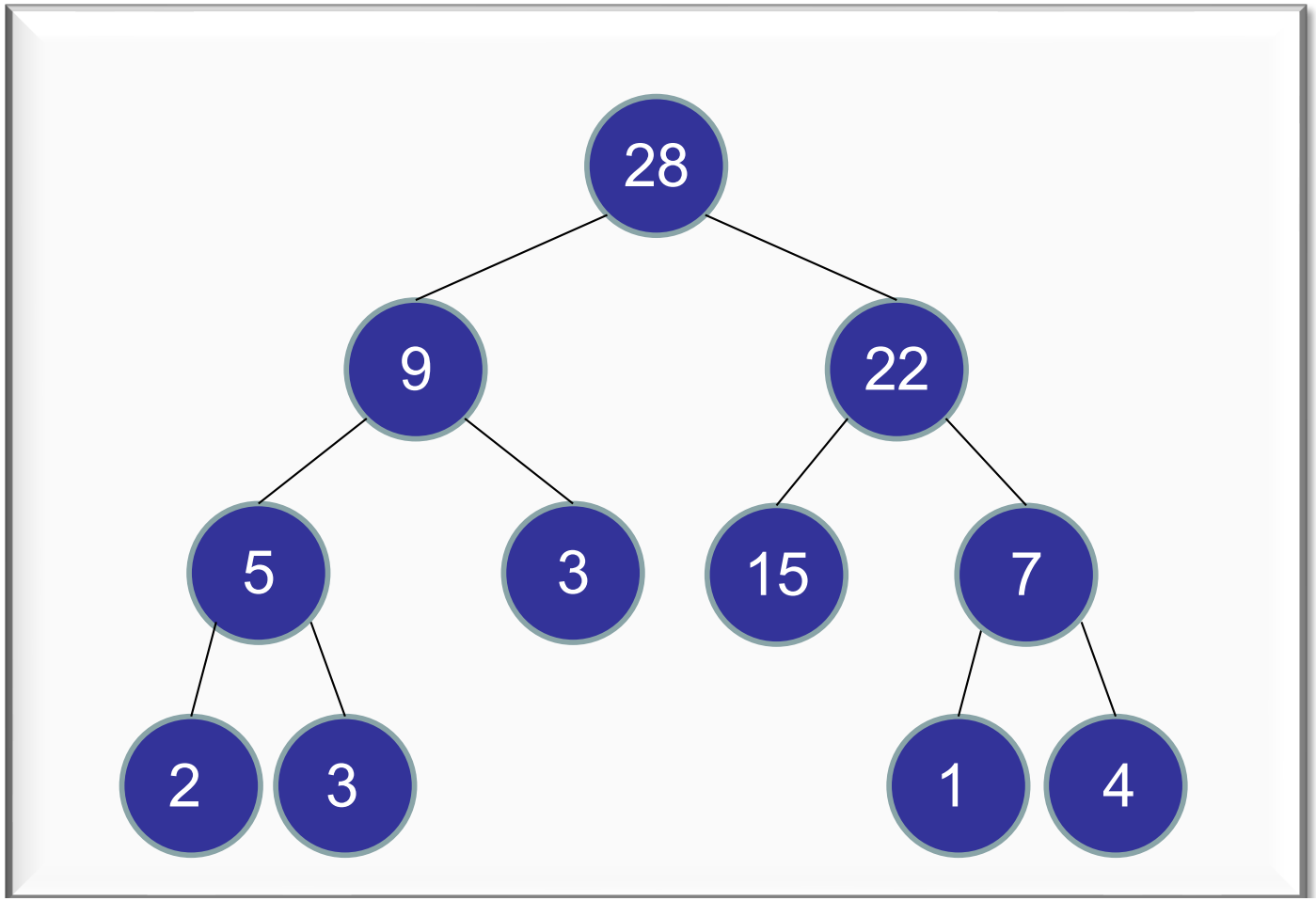
2. No.





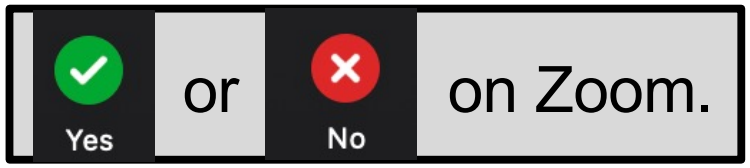
Is it a heap?

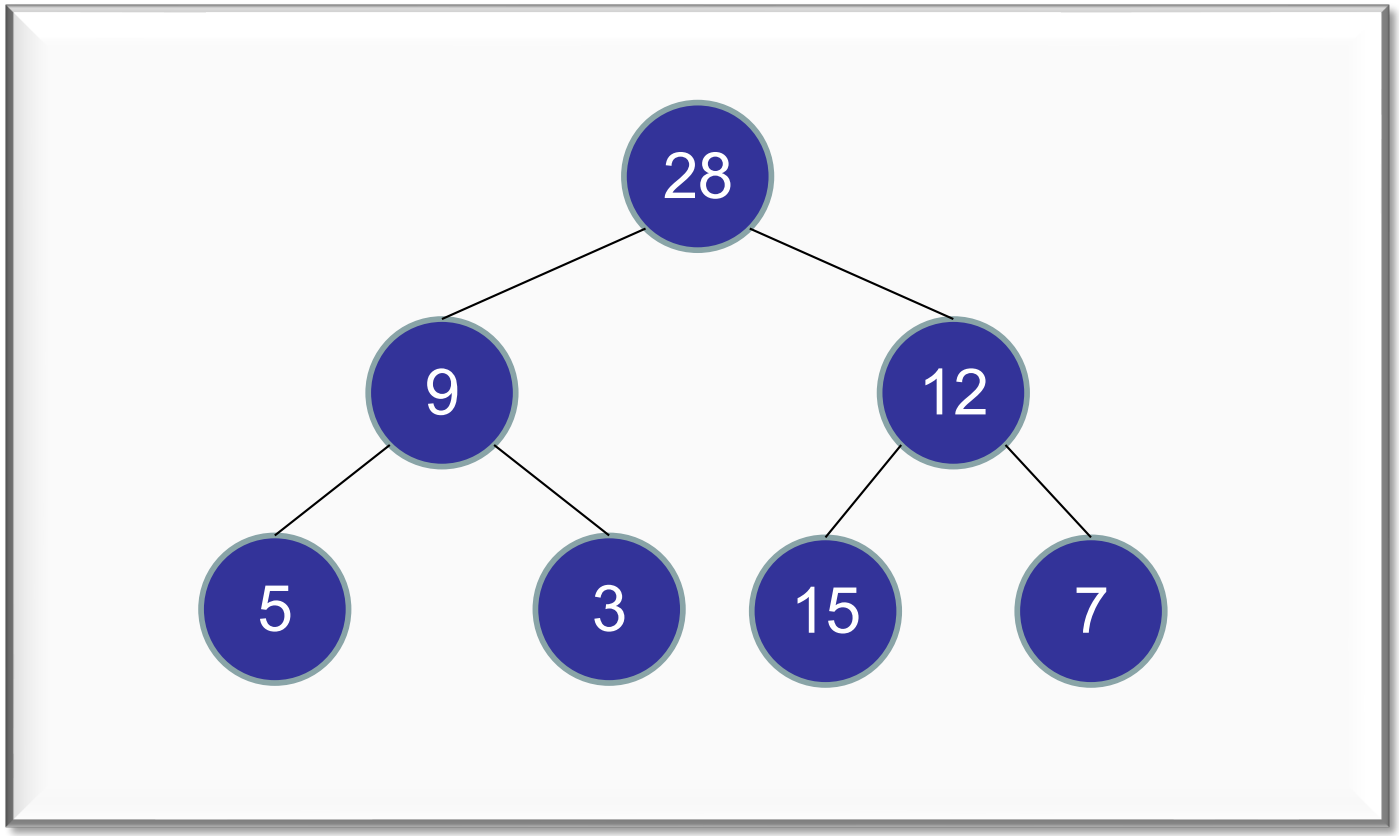
1. Yes
2. ✓ No.



Is it a heap?

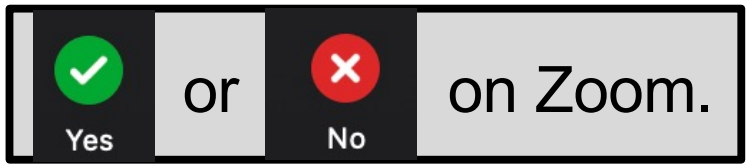
1. Yes
2. ✓ No.

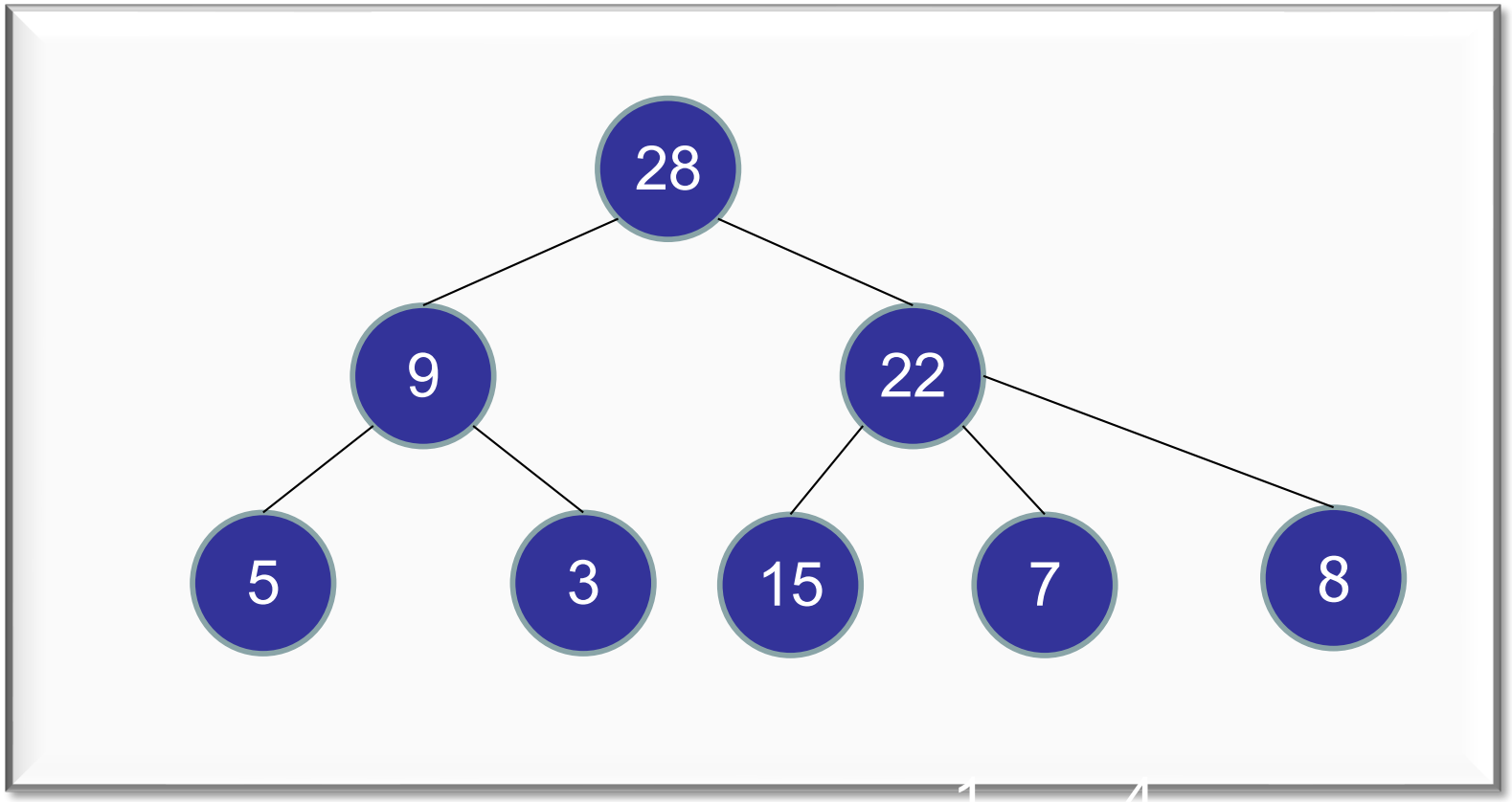




Is it a heap?

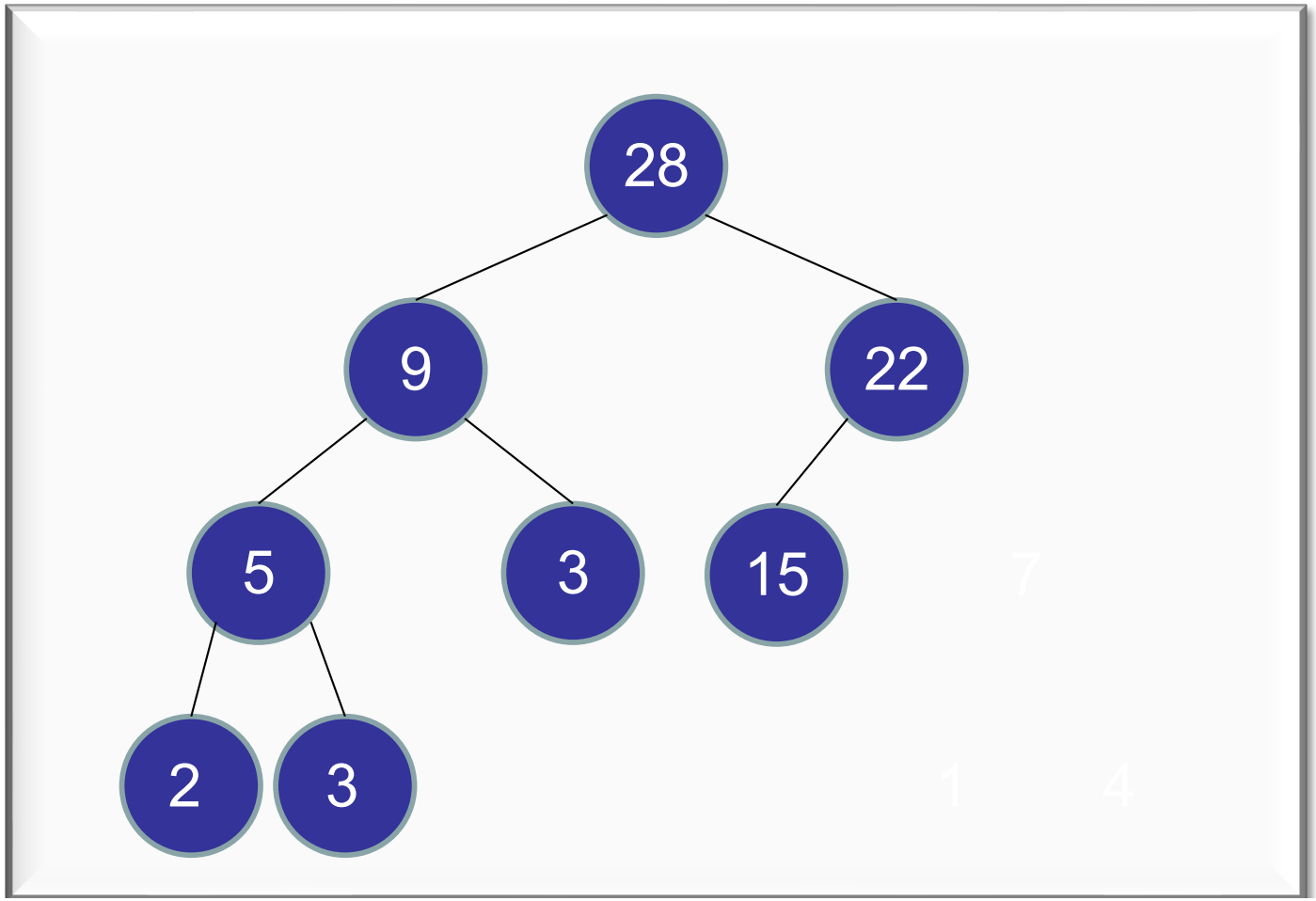
1. Yes
2. ✓ No.





Is it a heap?

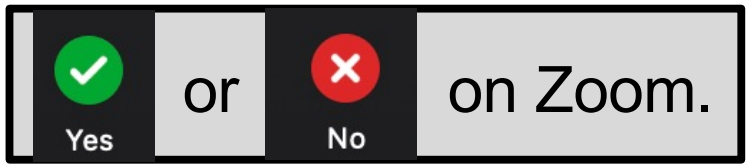
1. Yes
2. ✓ No.

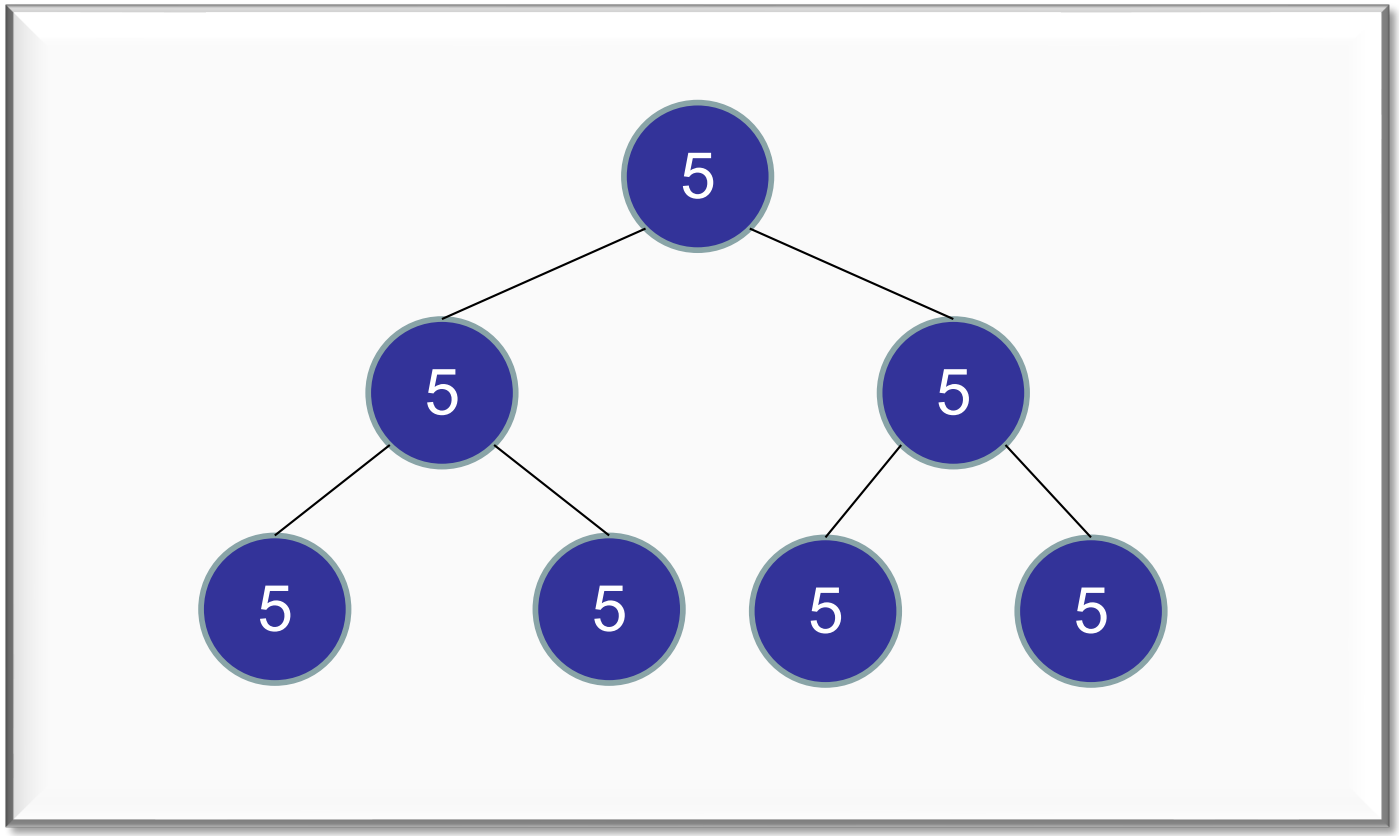


Is it a heap?

✓1. Yes

2. No.

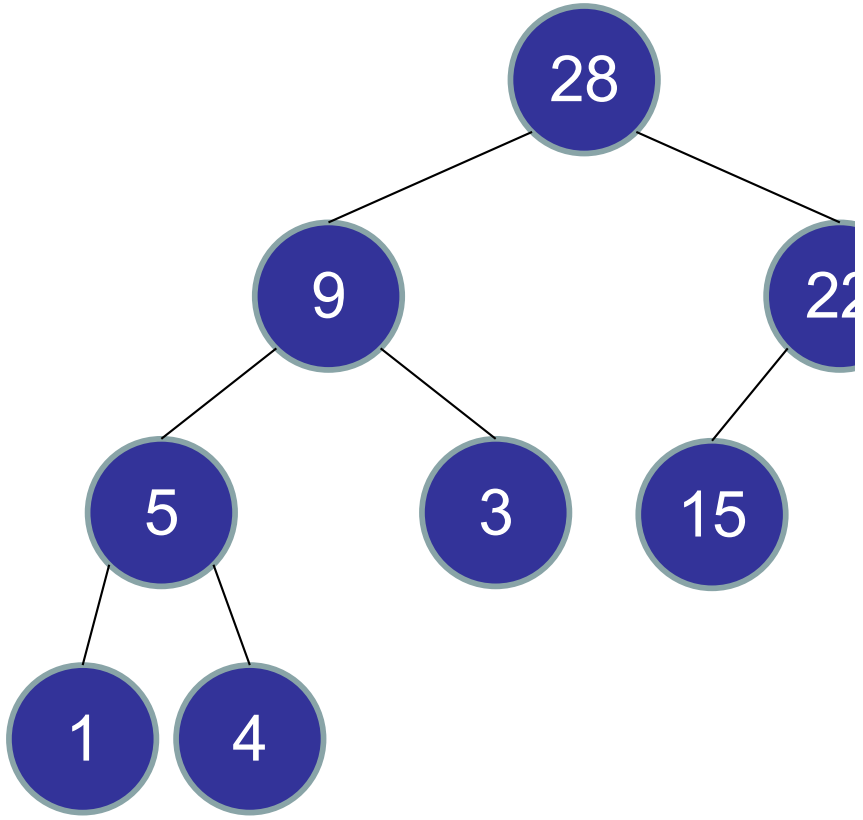




Heap

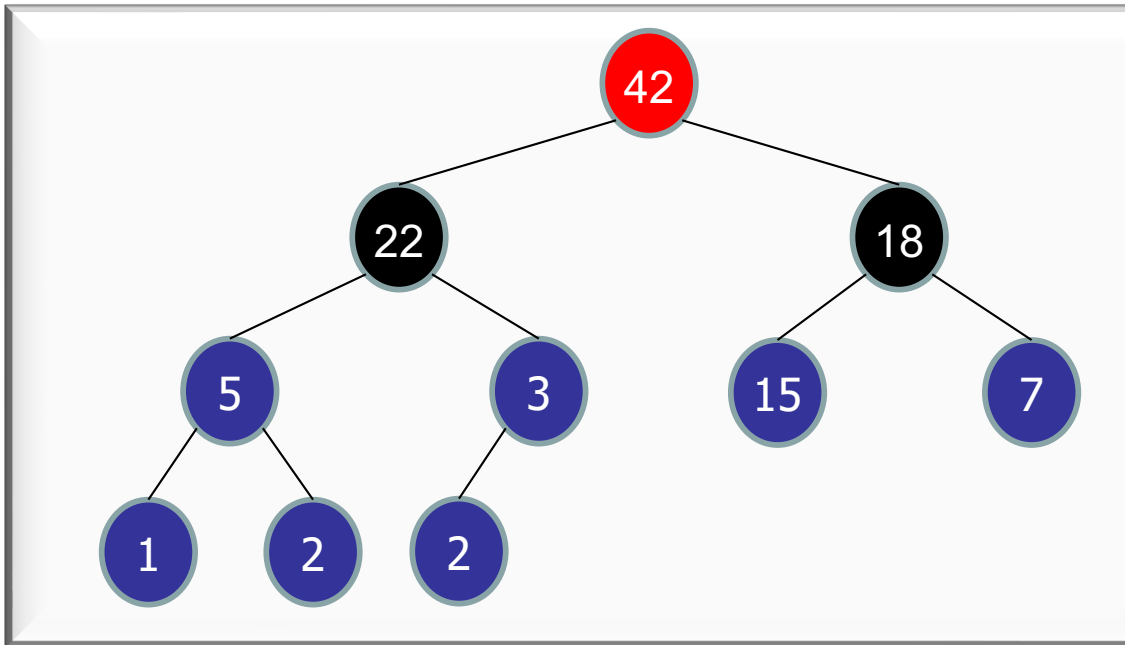
(aka Binary Heap or MaxHeap)

- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.
- Two properties:
  1. Heap Ordering
  2. Complete Binary Tree



What is the maximum height of a heap with n elements?

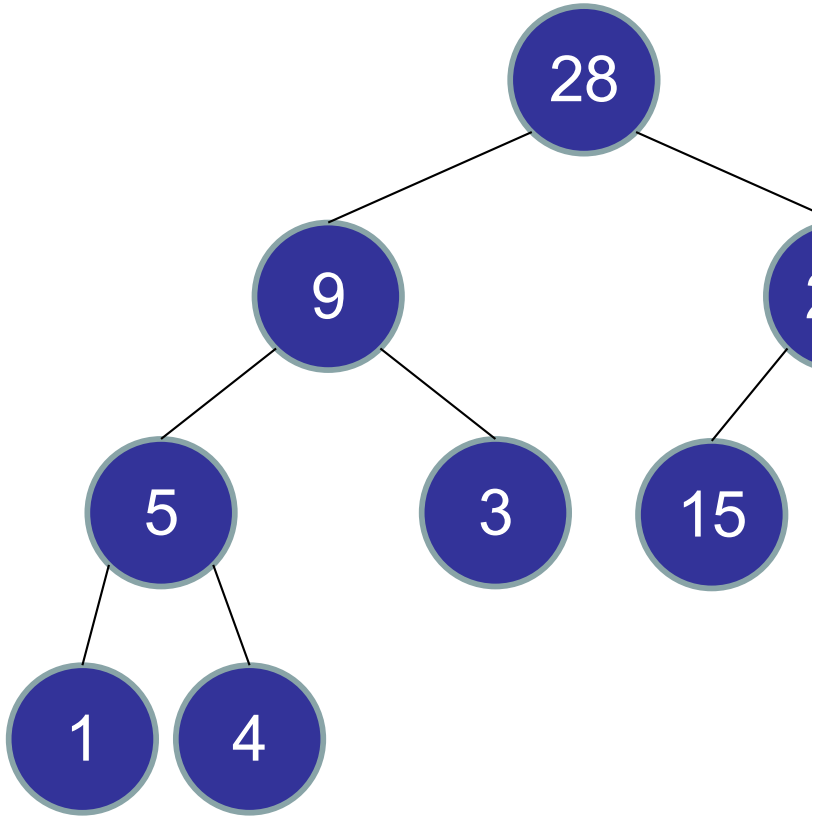
1. floor(log(n-1))
2. log(n)
3. ✓ floor(log n)
4. ceiling(log n)
5. ceiling(log(n+1))



Heap

(aka Binary Heap or MaxHeap)

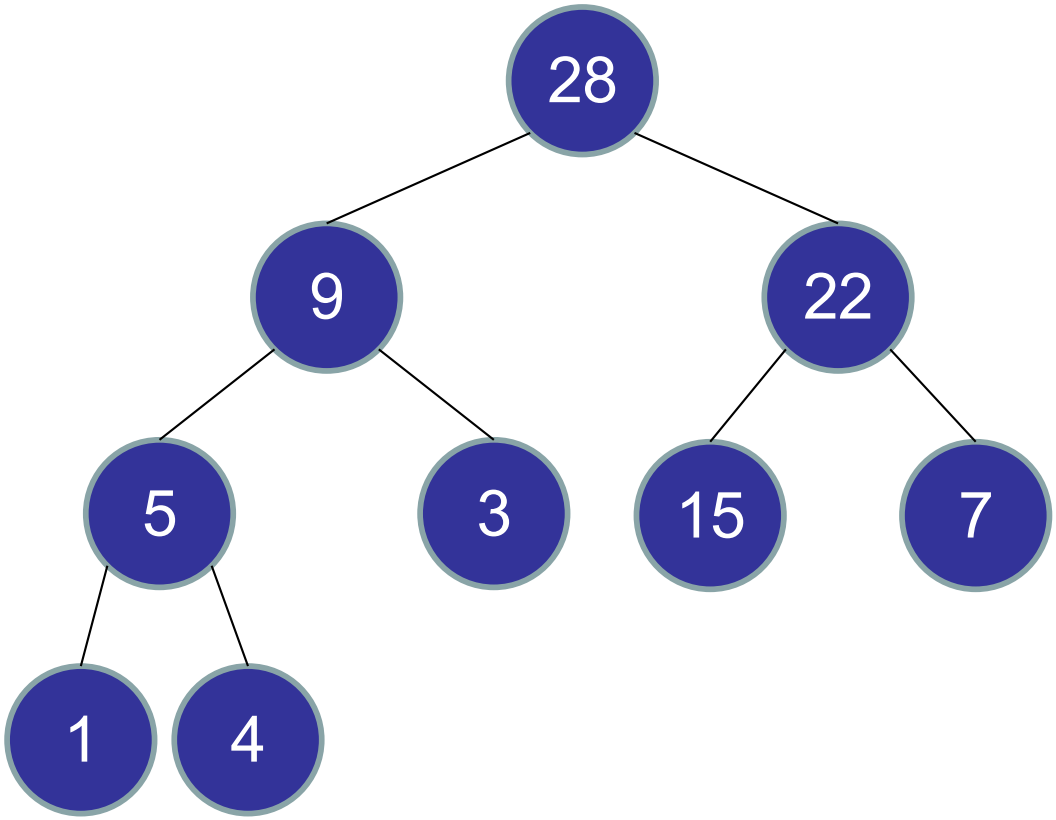
- Implements a Max Priority Queue
- Maintain a set of prioritized objects.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.
- Two properties:
  1. Heap Ordering
  2. Complete Binary Tree
- Height: O(log n)



Heap

Priority Queue Operations

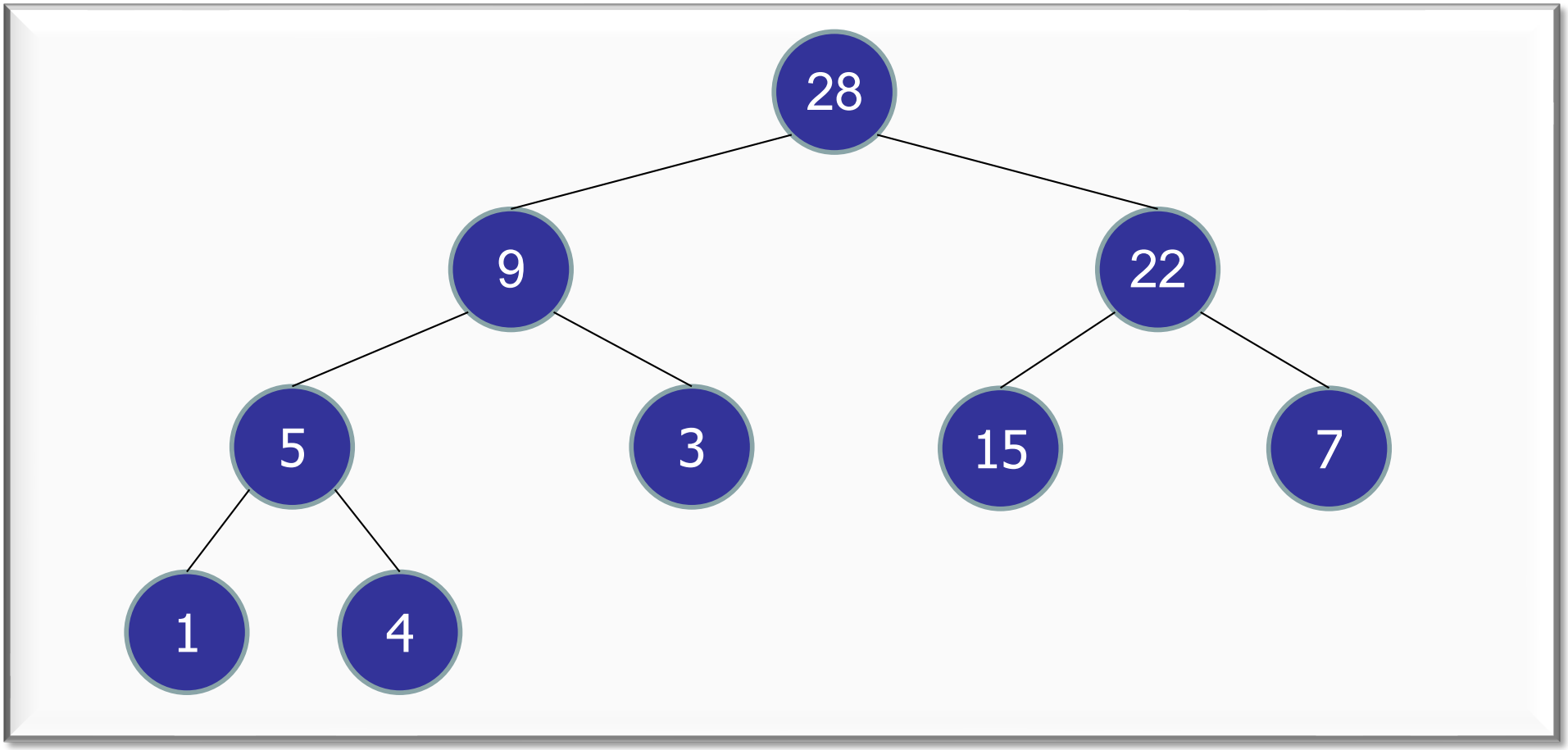
- insert
- extractMax
- increaseKey
- decreaseKey
- delete



Heap Operations

`insert(25)`:

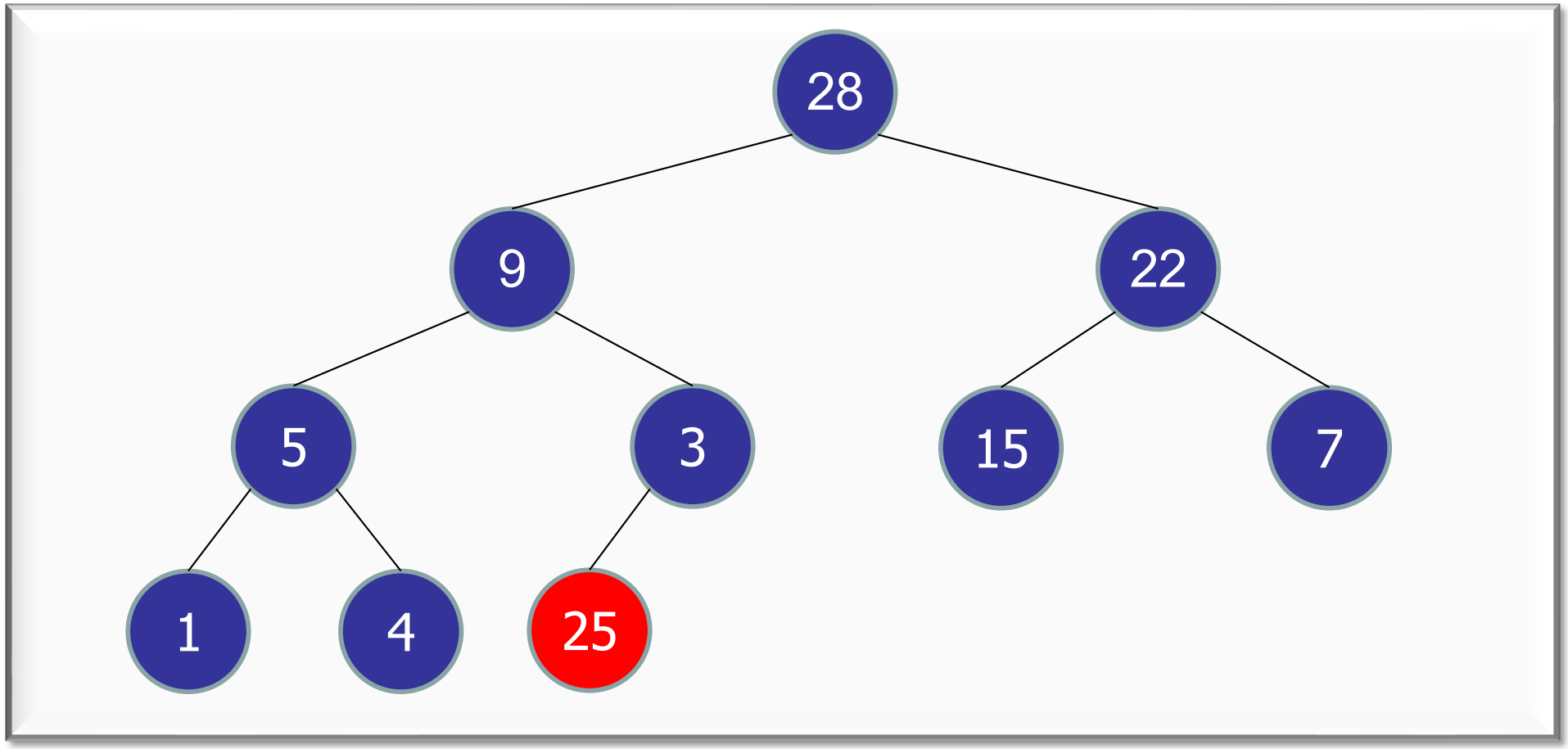
- Step one: add a new leaf with priority 25.



Heap Operations

`insert(25)`:

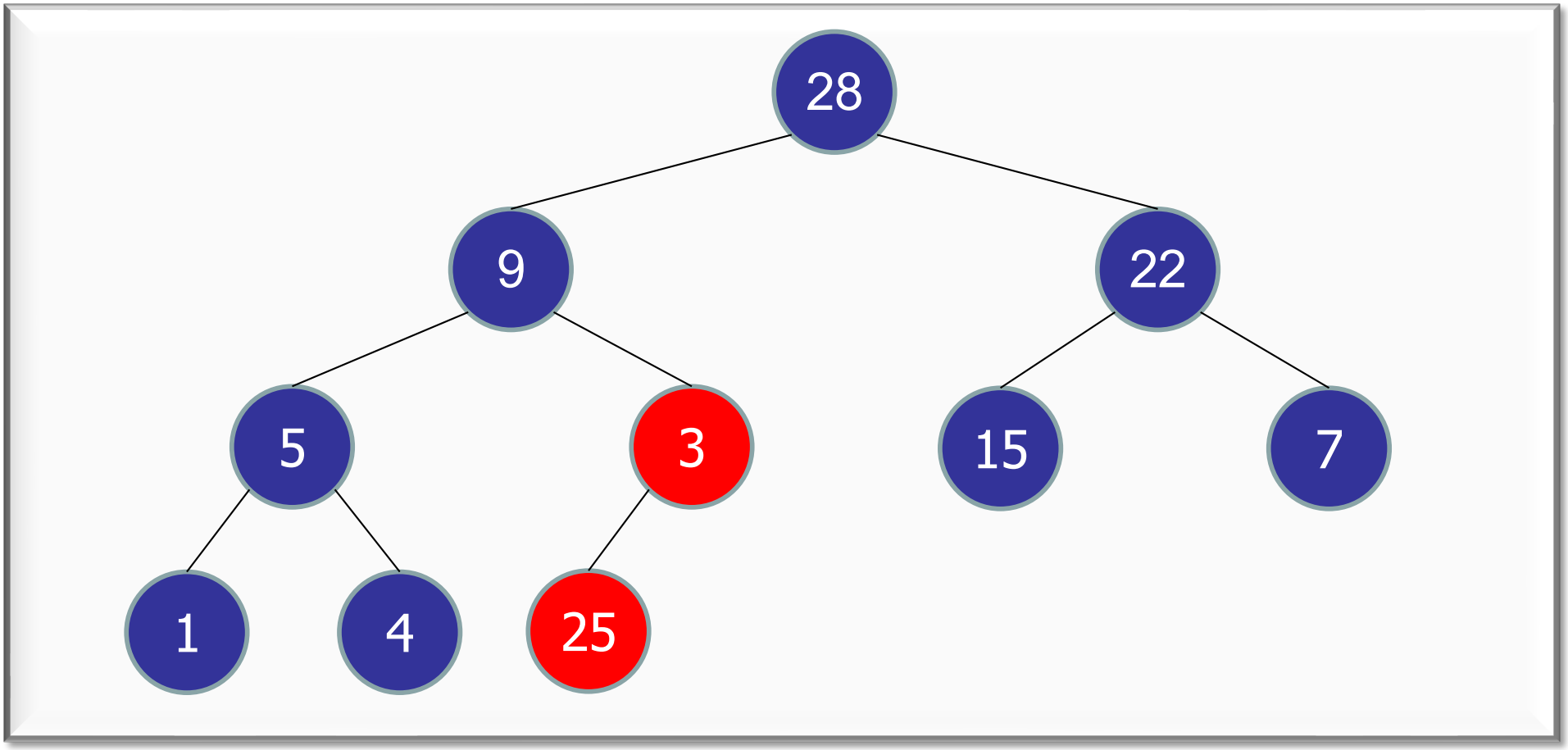
- Step one: add a new leaf with priority 25.



Heap Operations

`insert(25)`:

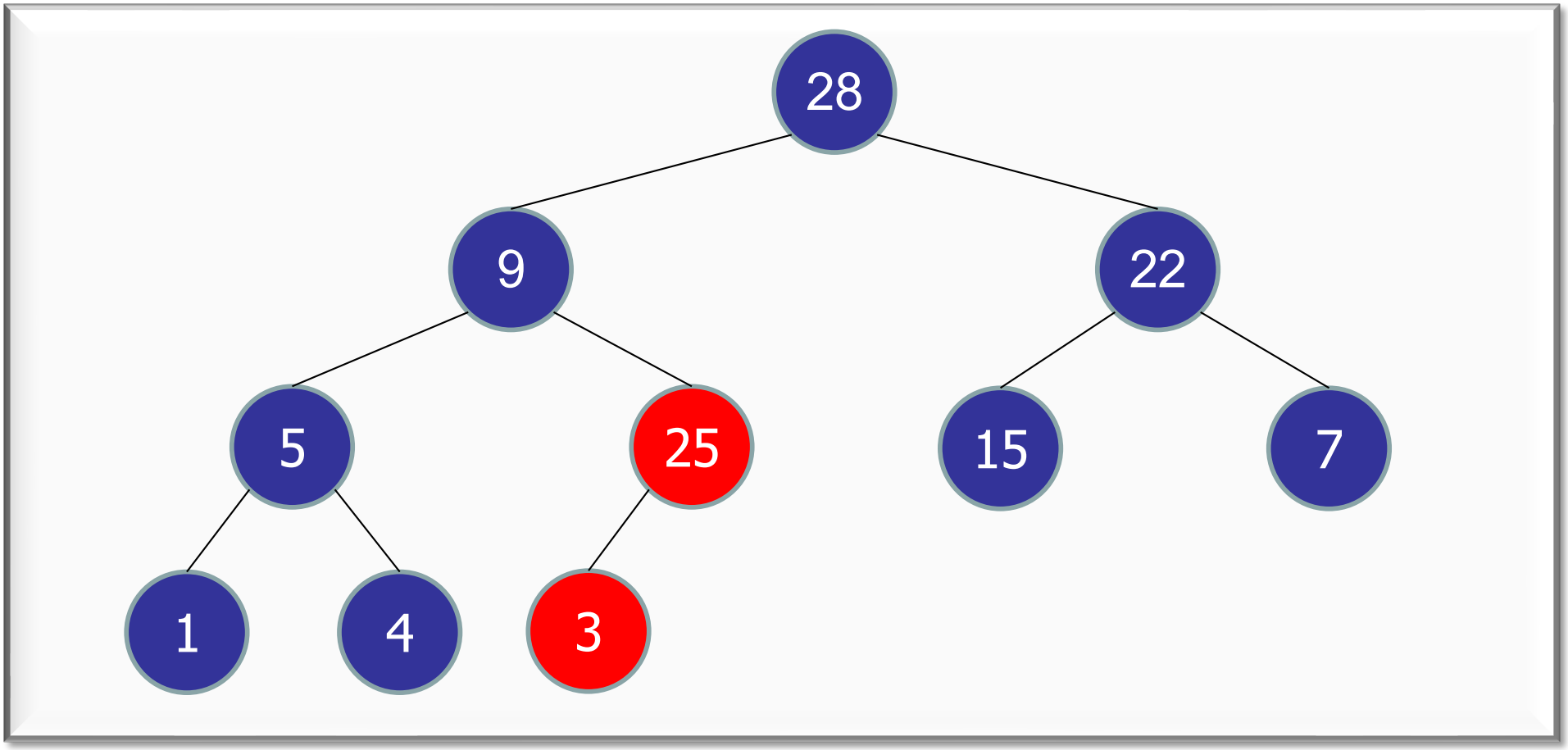
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

`insert(25)`:

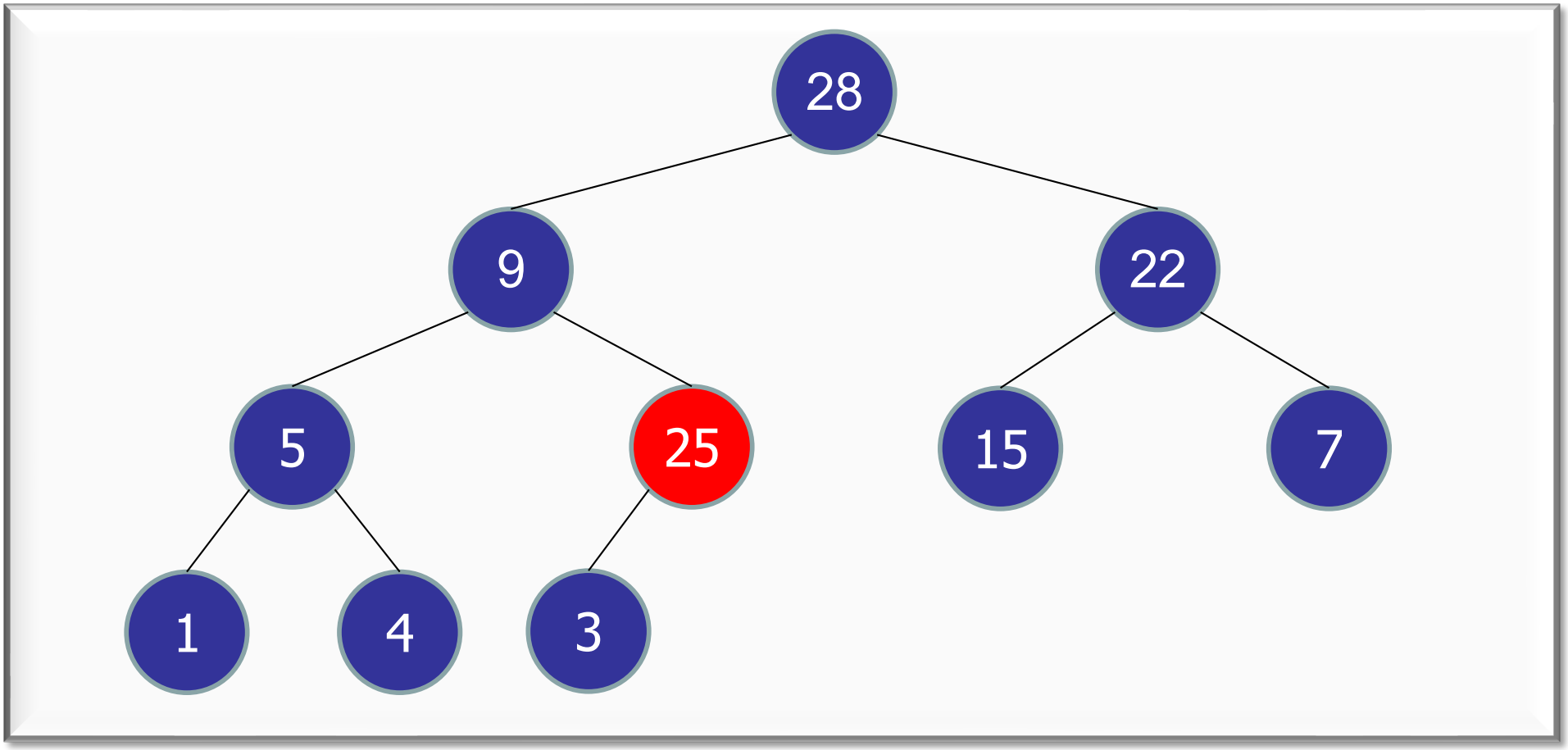
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

`insert(25)`:

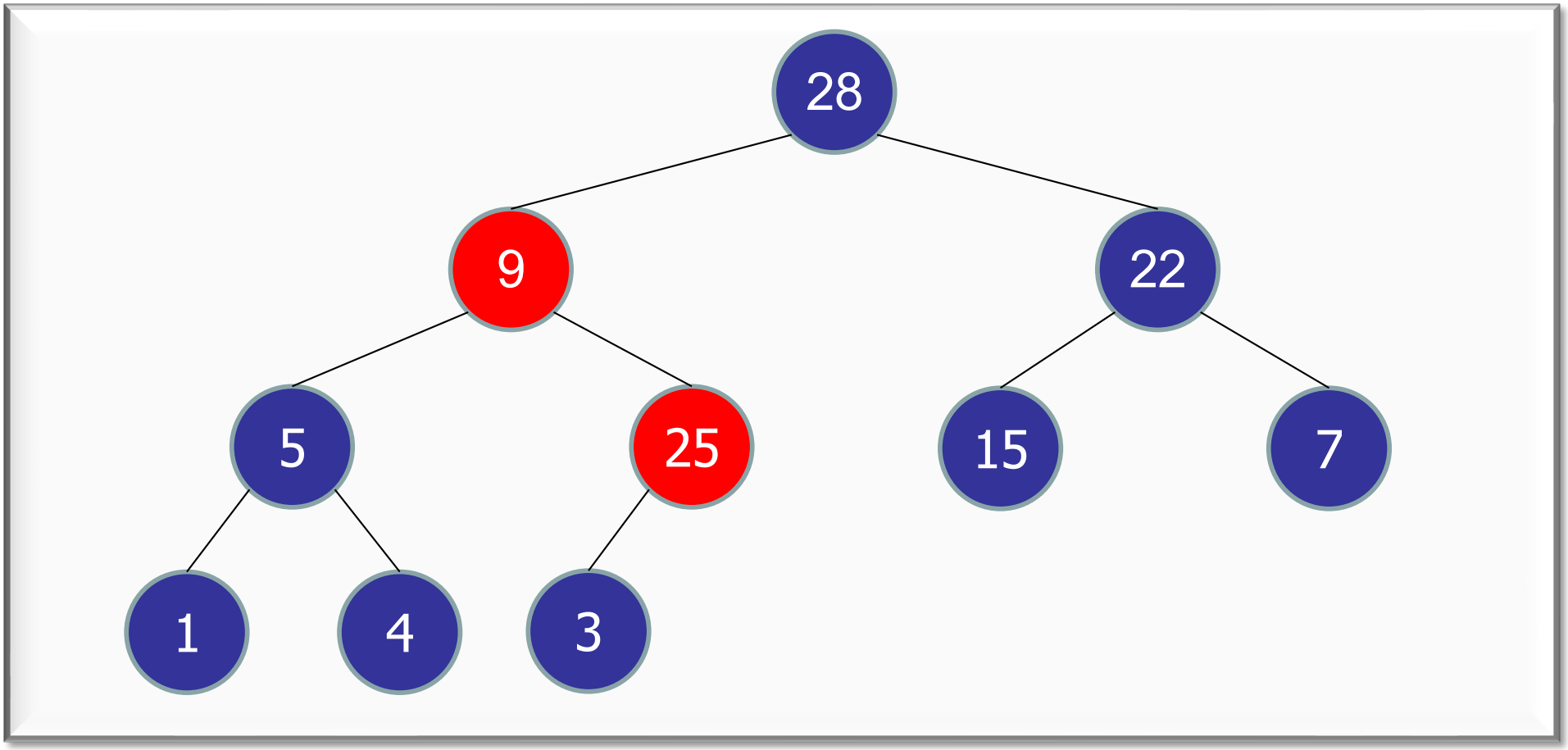
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

`insert(25)`:

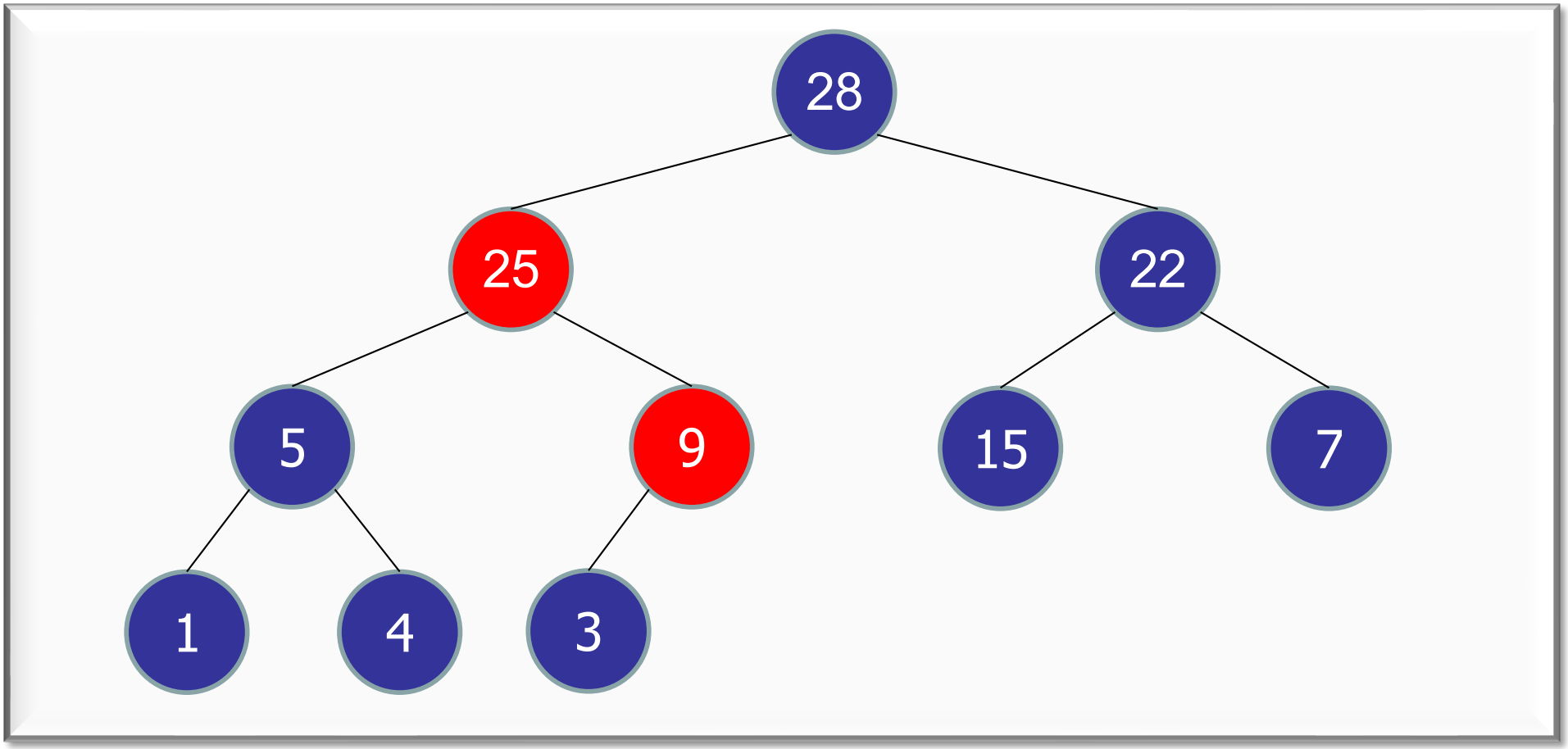
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

`insert(25)`:

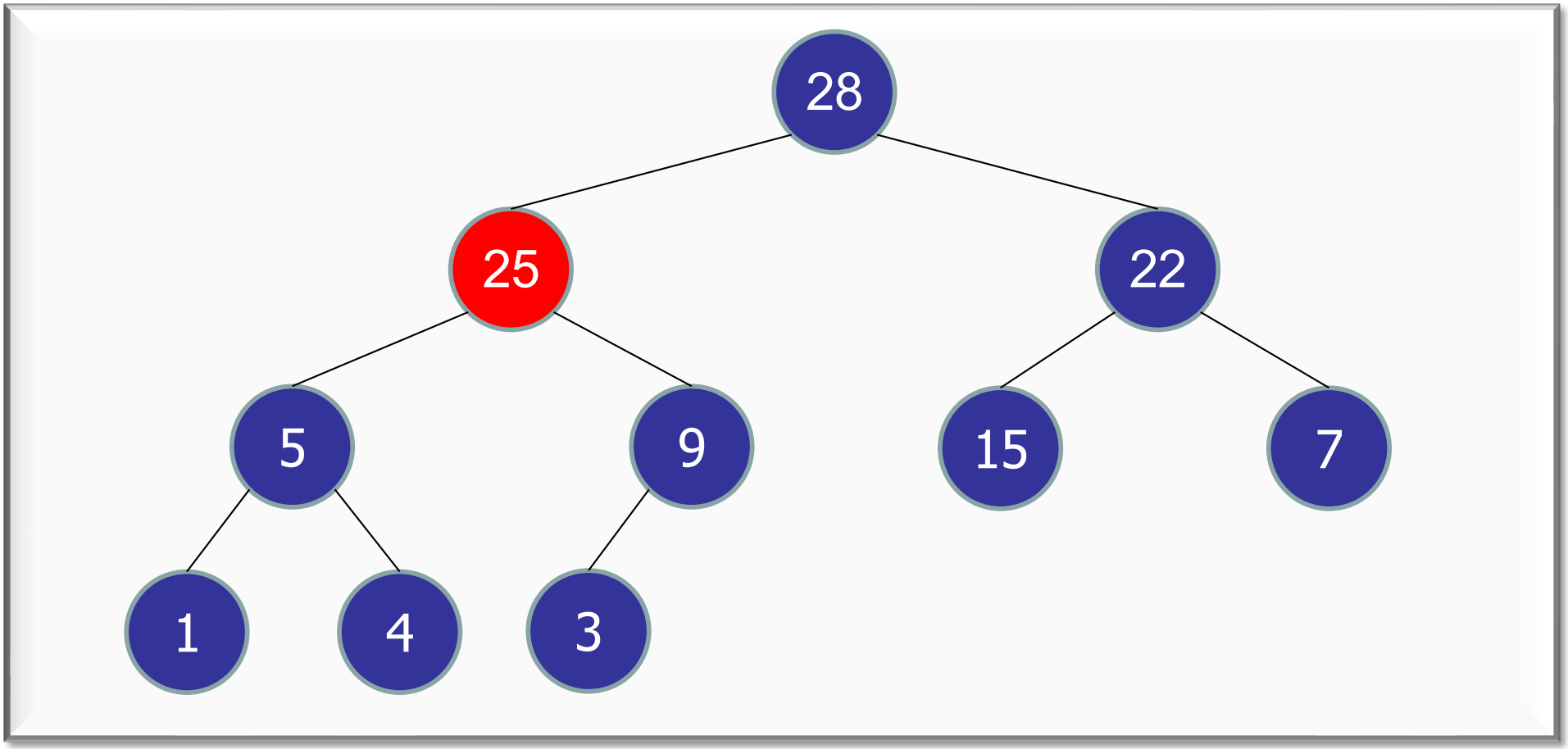
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

`insert(25)`:

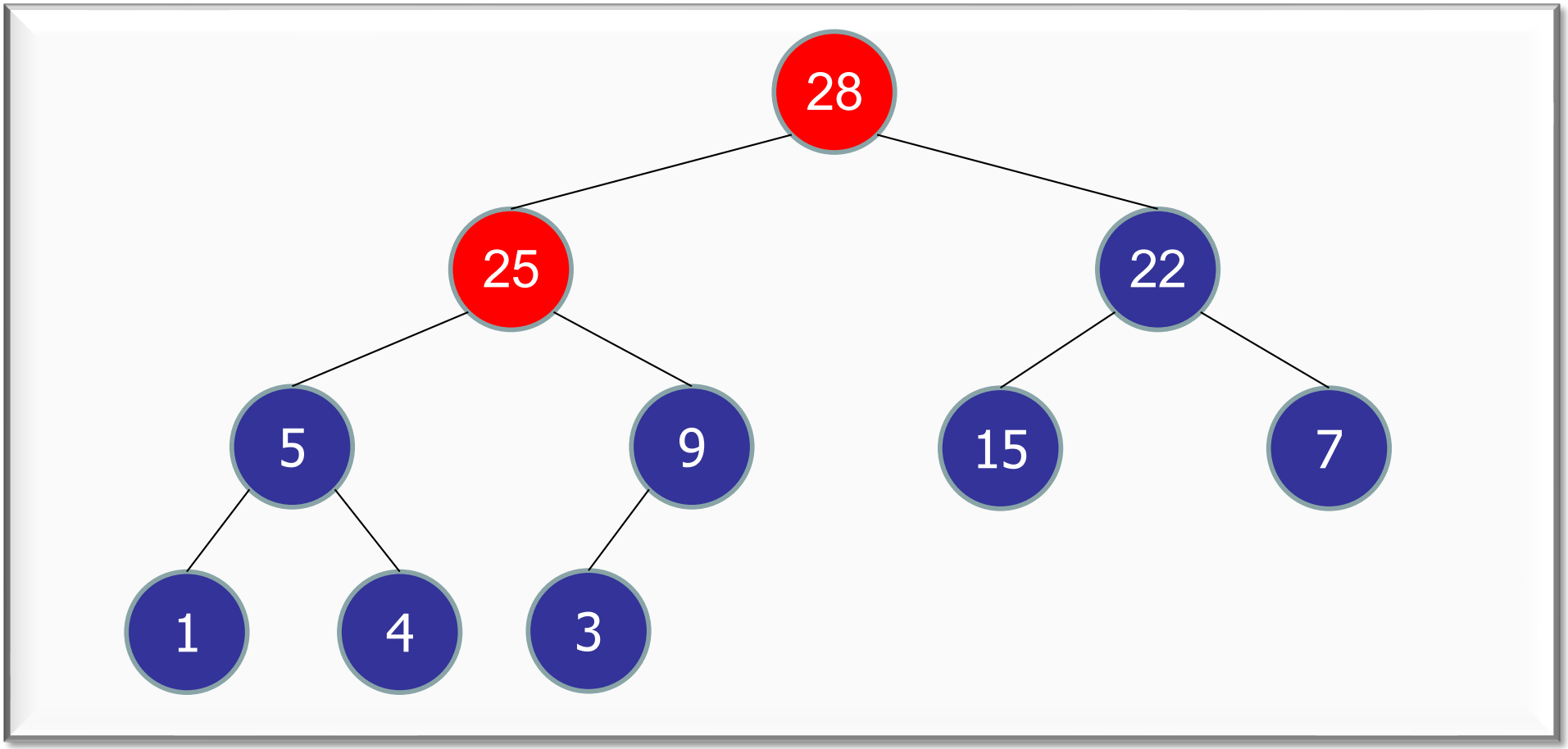
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

`insert(25)`:

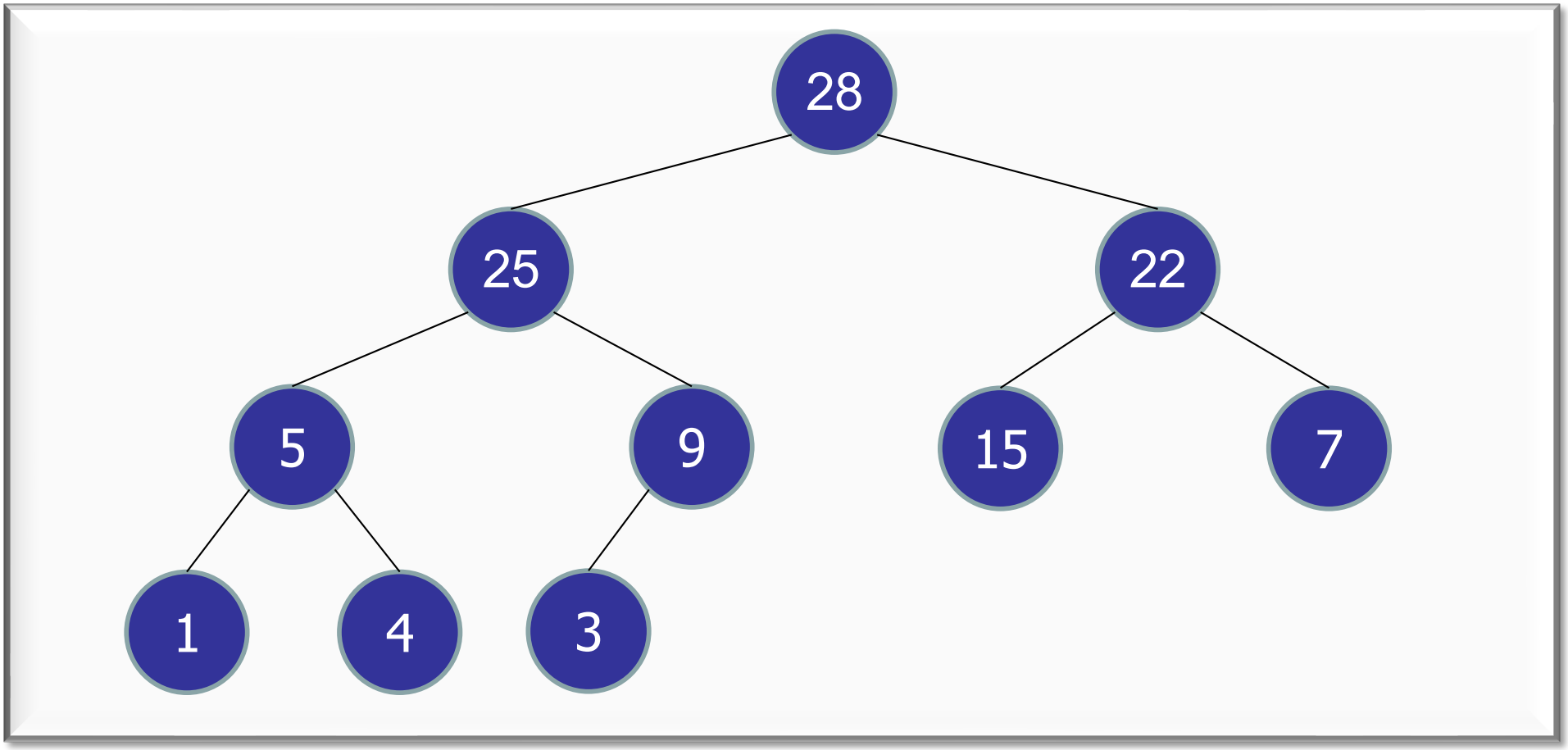
- Step one: add a new leaf with priority 25.
- Step two: bubble up



Heap Operations

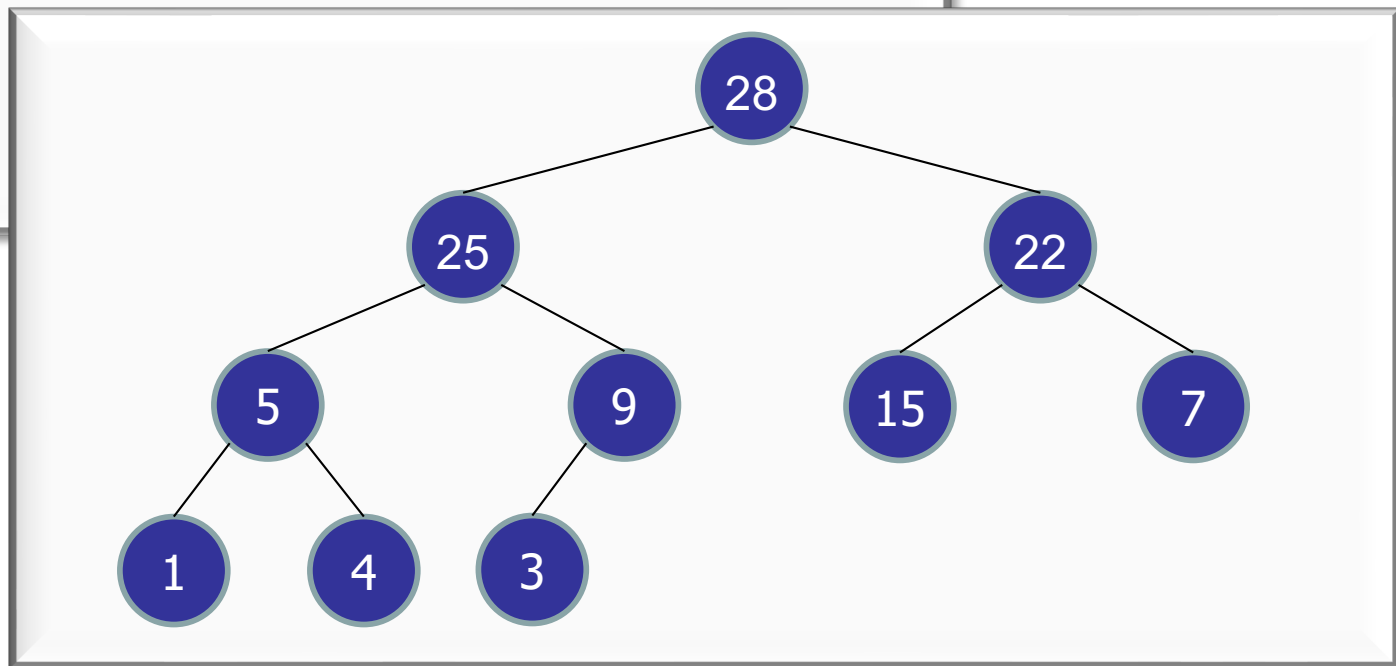
`insert(25)`:

- Step one: add a new leaf with priority 25.
- Step two: bubble up



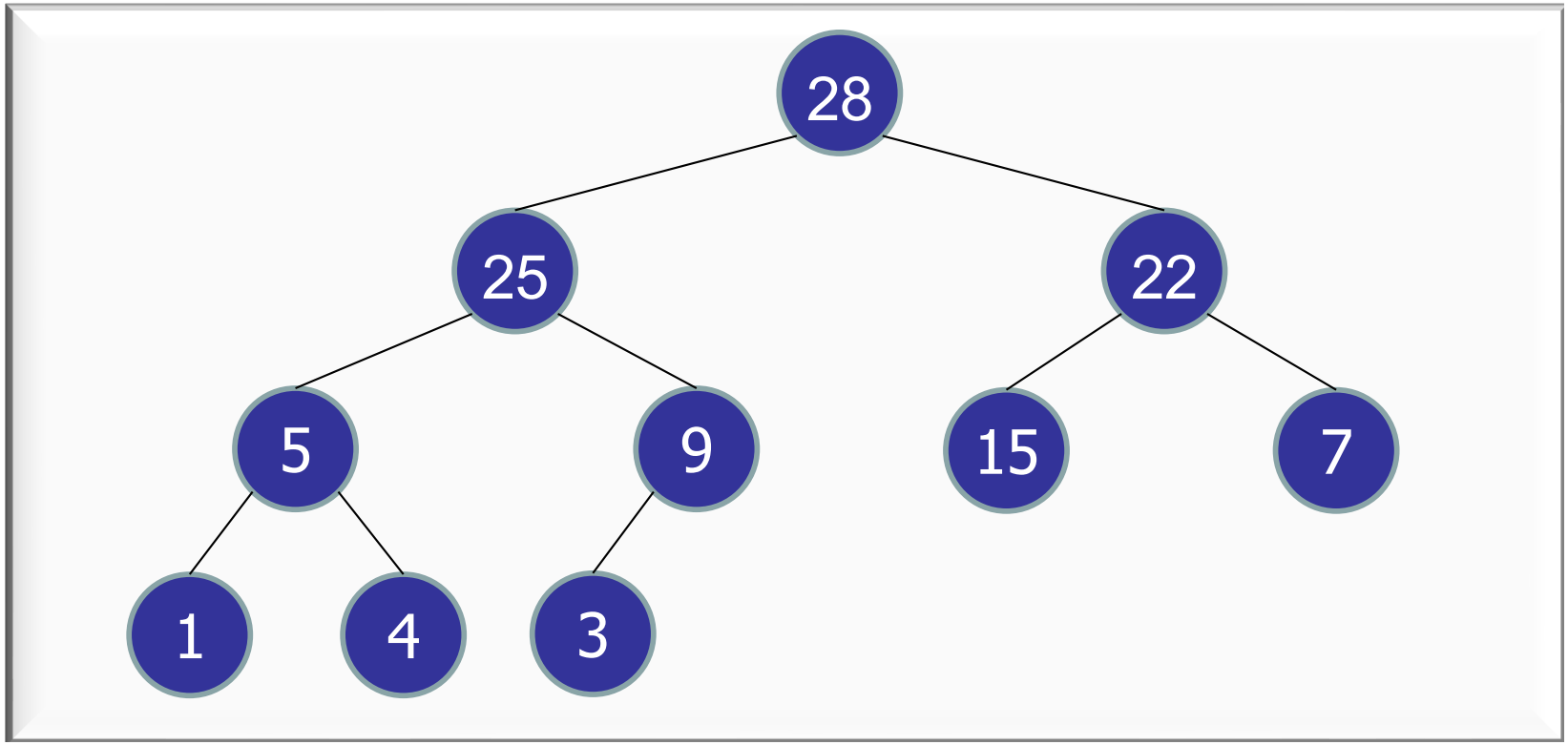
Heap Operations

```
bubbleUp(Node v) {
   while (v != null) {
      if (priority(v) > priority(parent(v)))
            swap(v, parent(v));
      else return;
      v = parent(v);
   }
}
```



Heap Operations

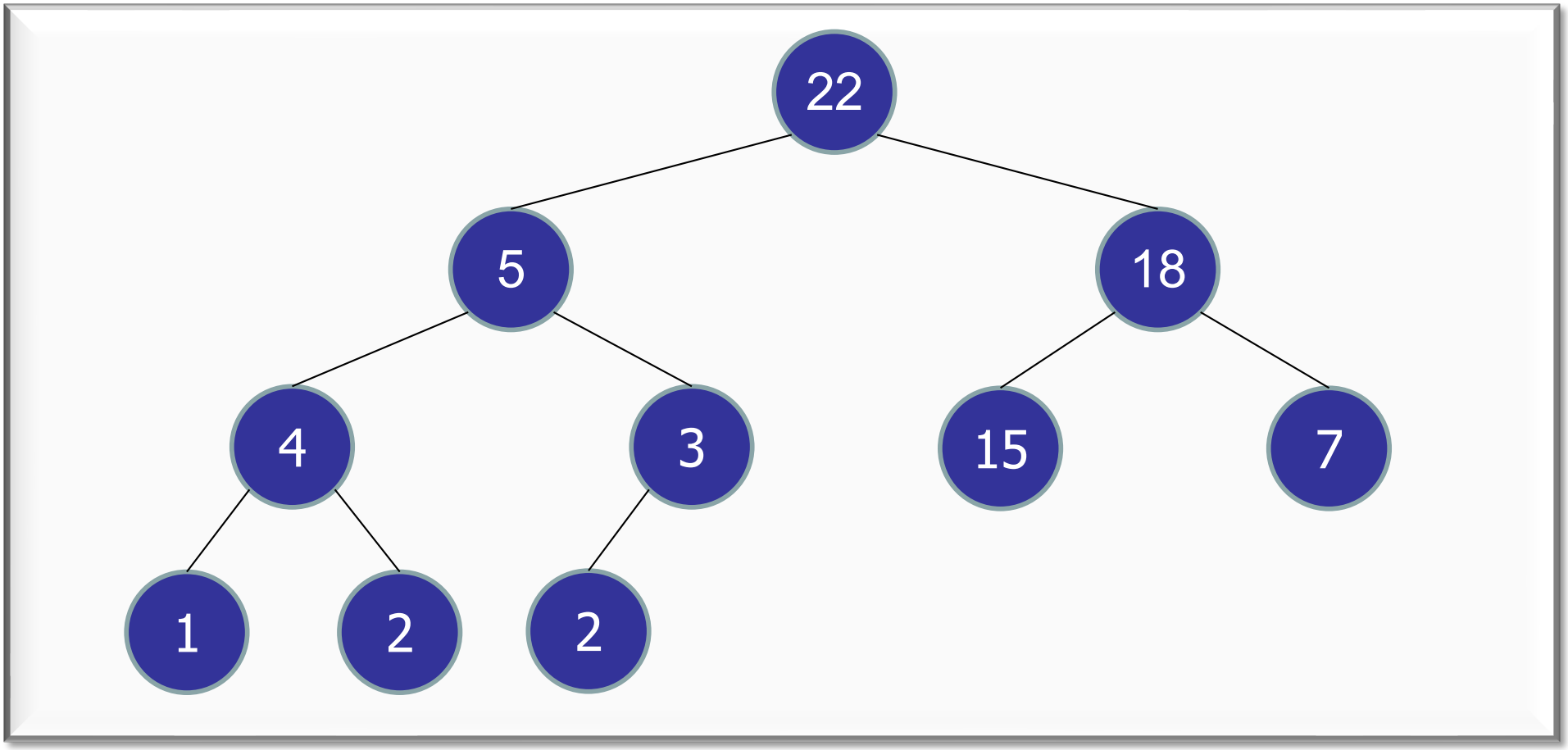
```
insert(Priority p, Key k) {
   Node v = completeTree.insert(p,k);
   bubbleUp(v);
}
```



Heap Operations

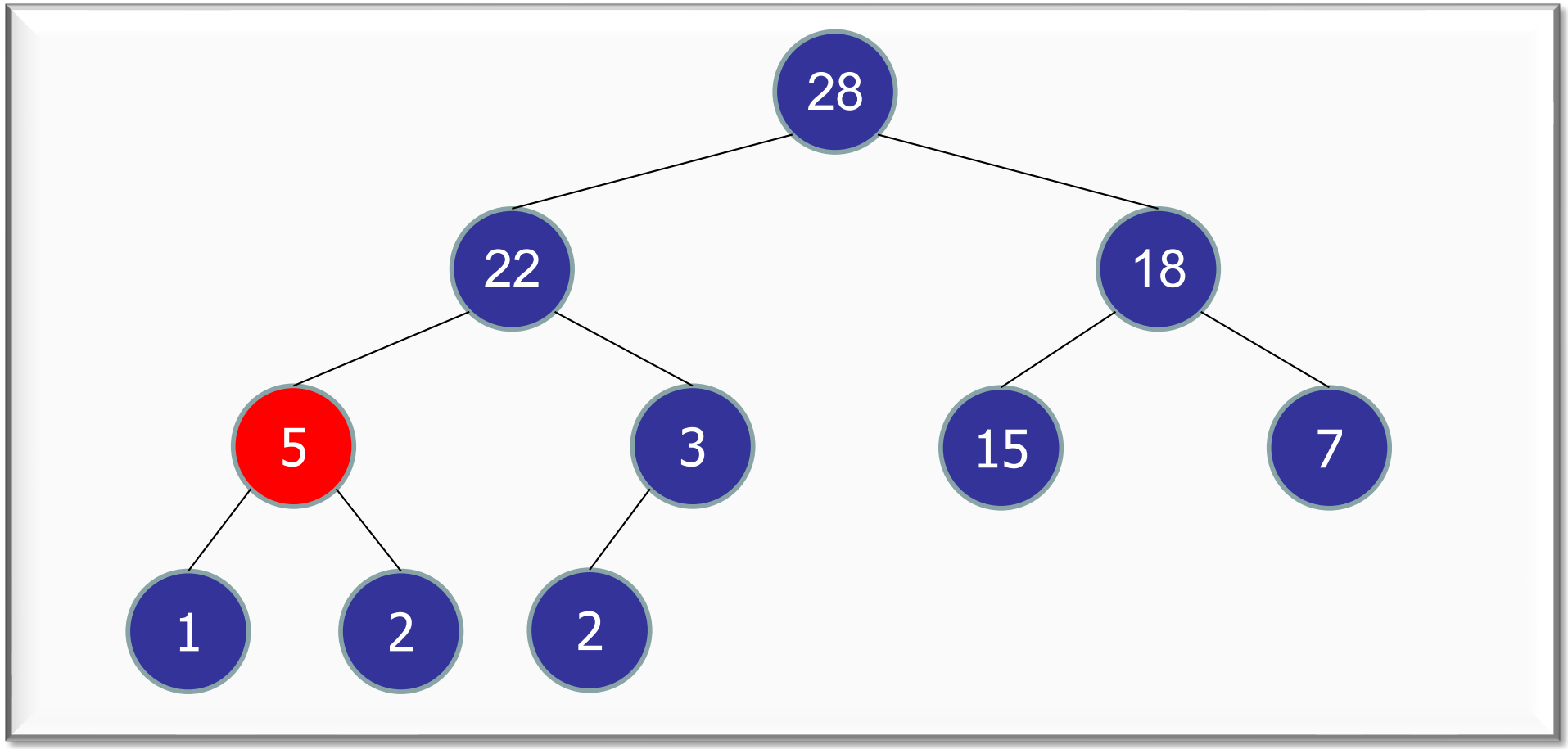
`insert(...)` :

- On completion, heap order is restored.
- Complete binary tree.



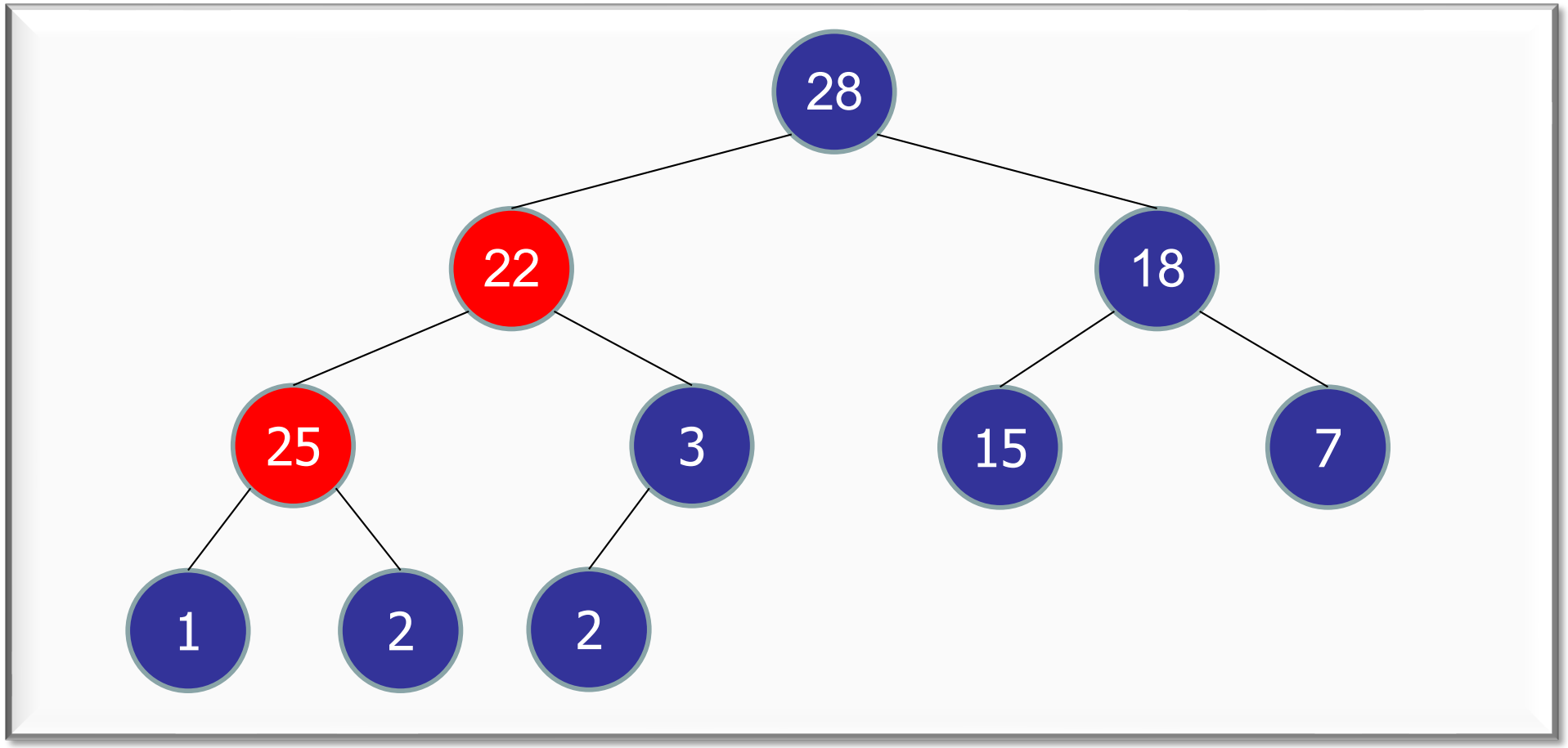
Heap Operations

`increaseKey(5 → 25):`



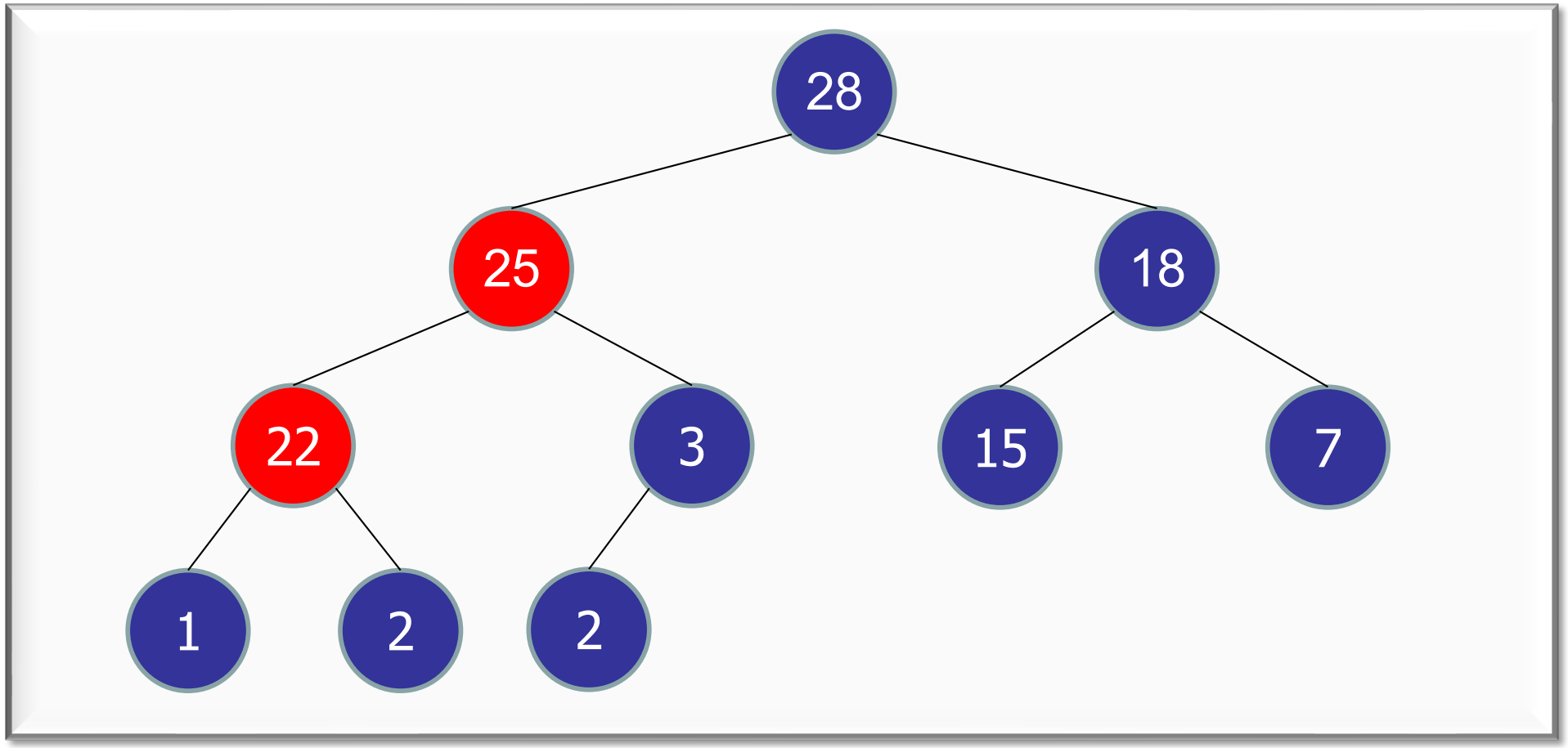
Heap Operations

`increaseKey(5 → 25)`: bubbleUp(25)



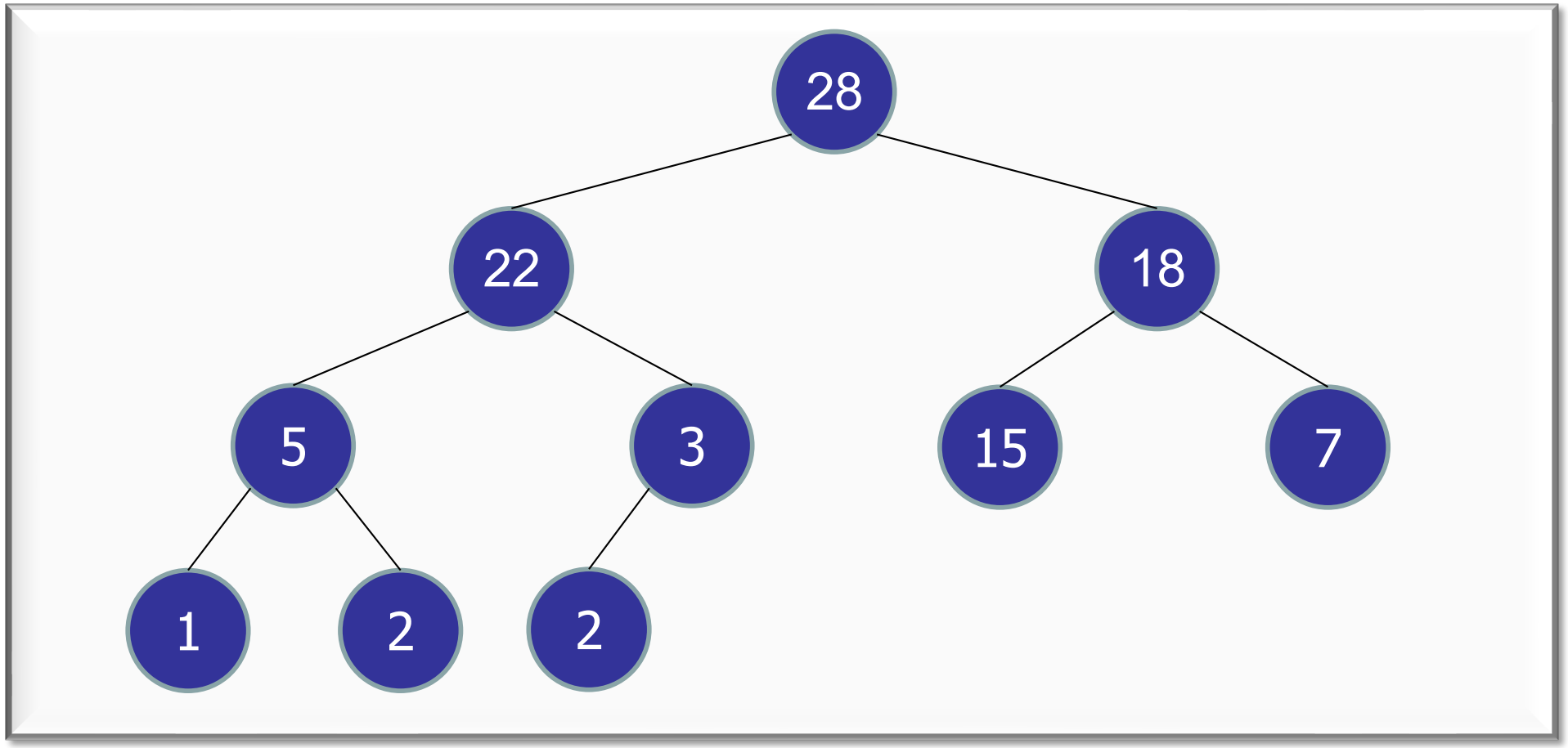
Heap Operations

`increaseKey(5 → 25)`: bubbleUp(25)



Heap Operations

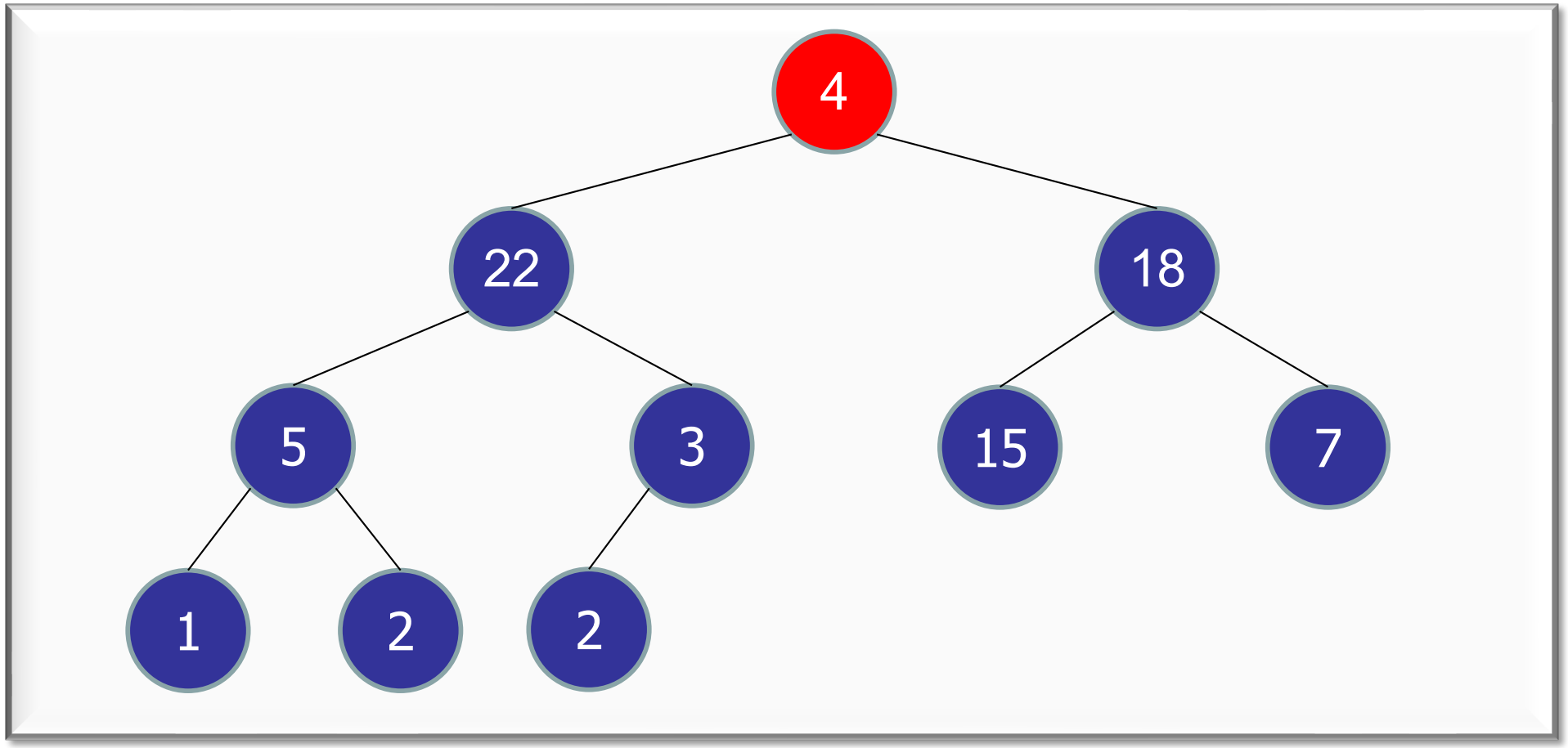
`decreaseKey(28 → 4):`



Heap Operations

`decreaseKey(28 → 4)`:

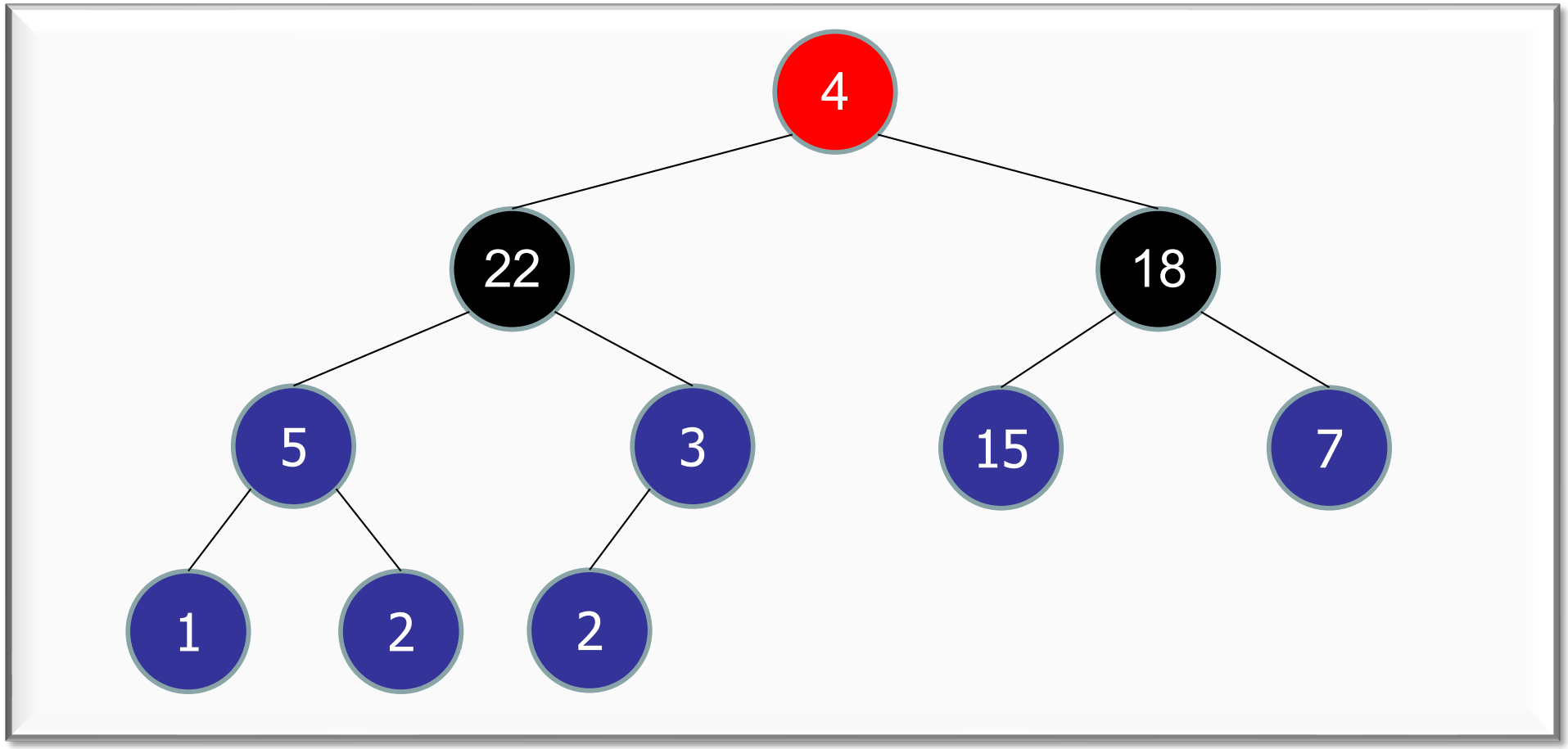
- Step 1: Update the priority



Heap Operations

`decreaseKey(28 → 4):`

- Step 1: Update the priority
- Step 2: bubbleDown(4)

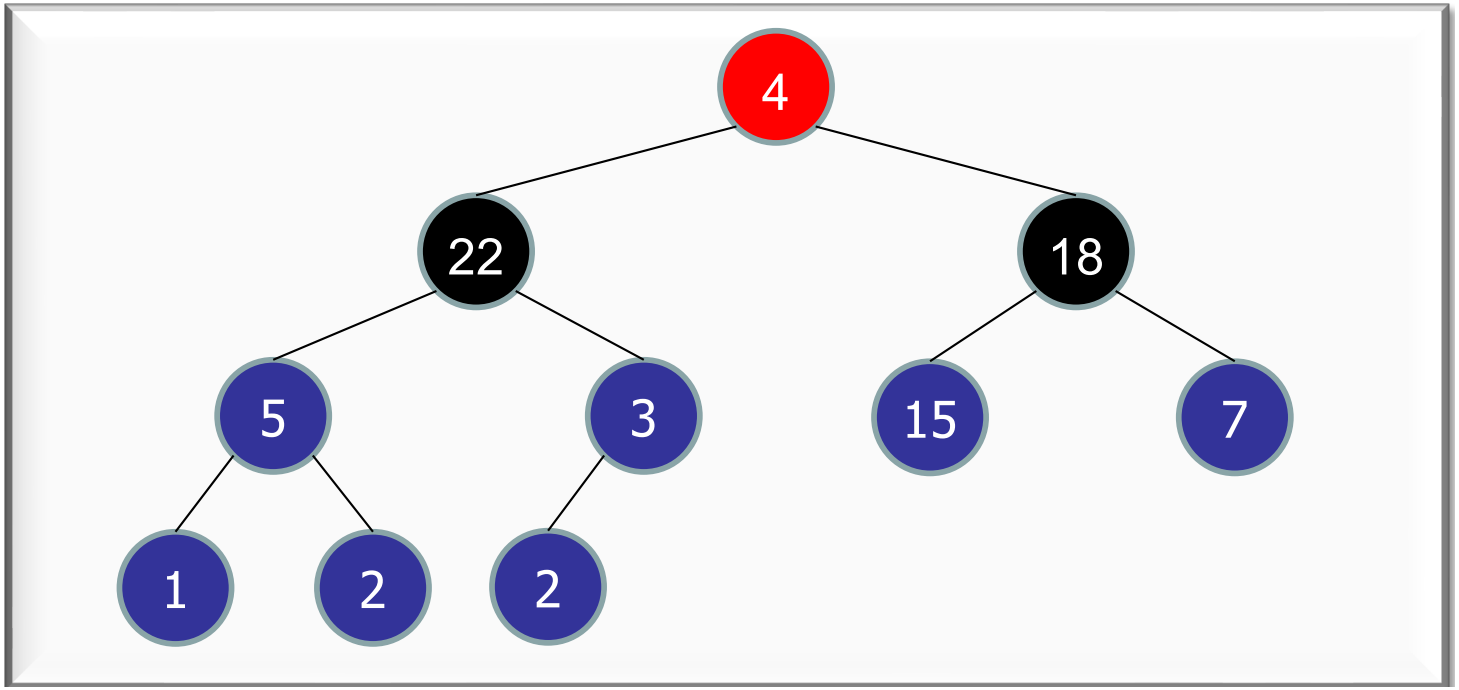


Which way to bubbleDown?

✓1. Left
2. Right
3. Does not matter

ARCHIPELAGO

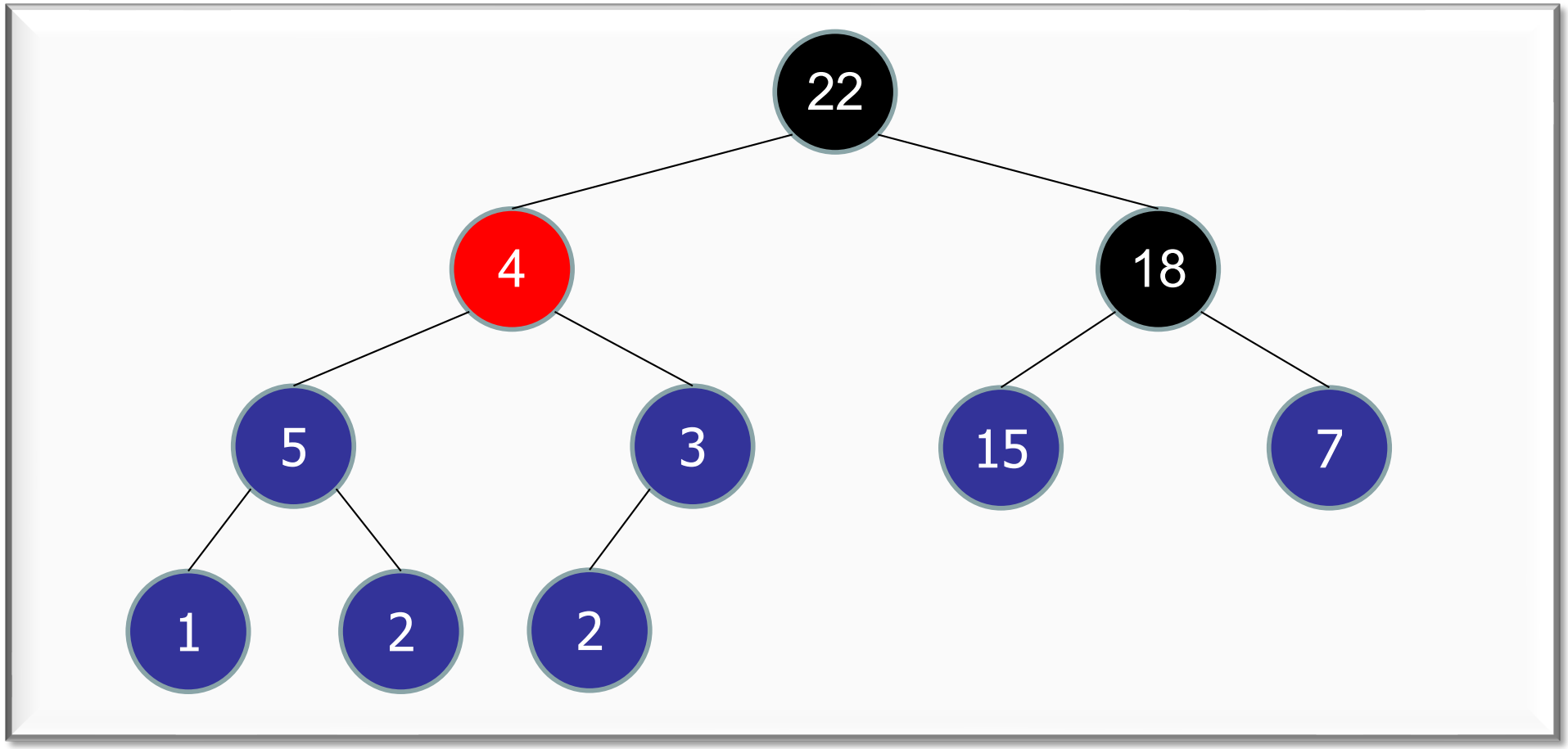
is open



Heap Operations

`decreaseKey(28 → 4)` :

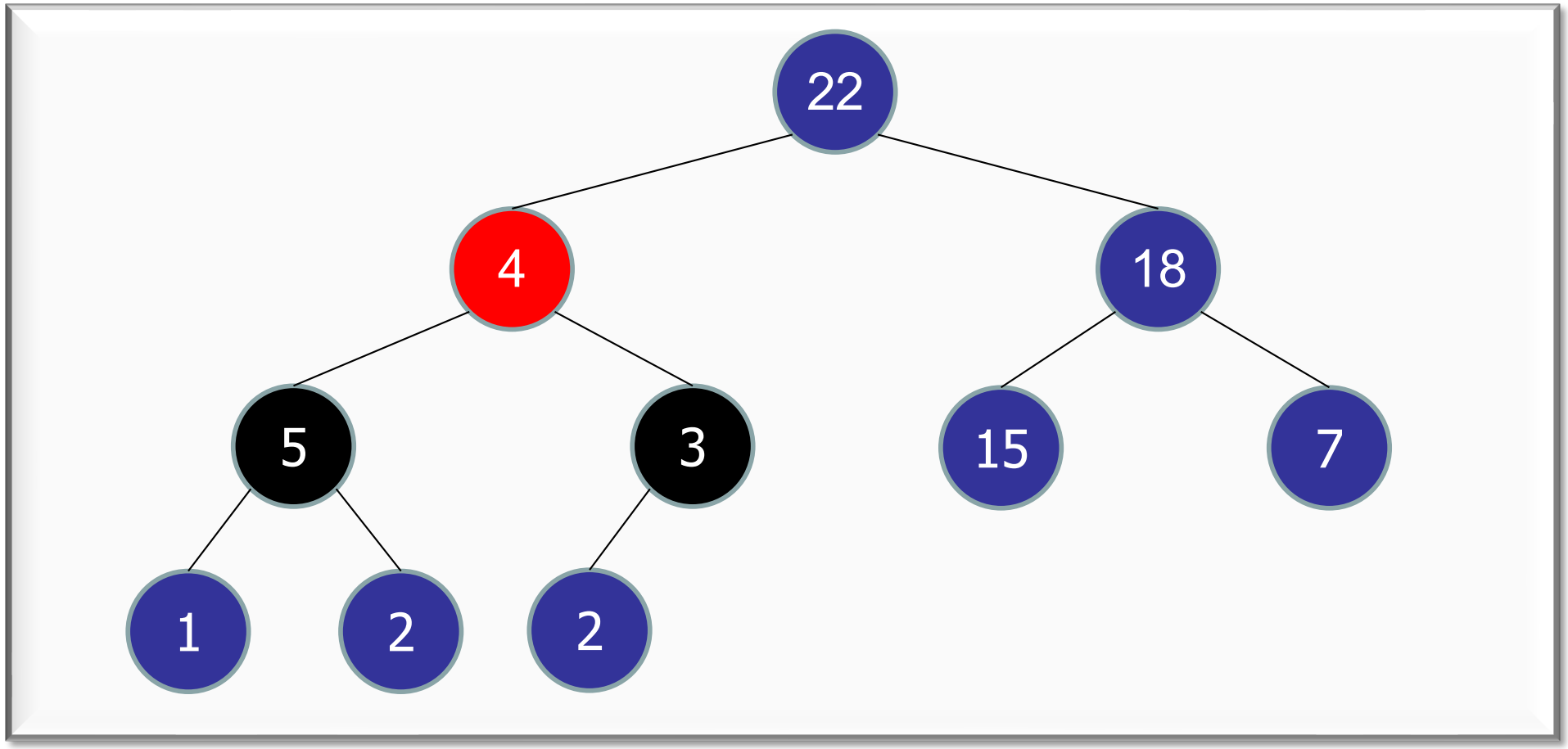
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Heap Operations

`decreaseKey(28 → 4)` :

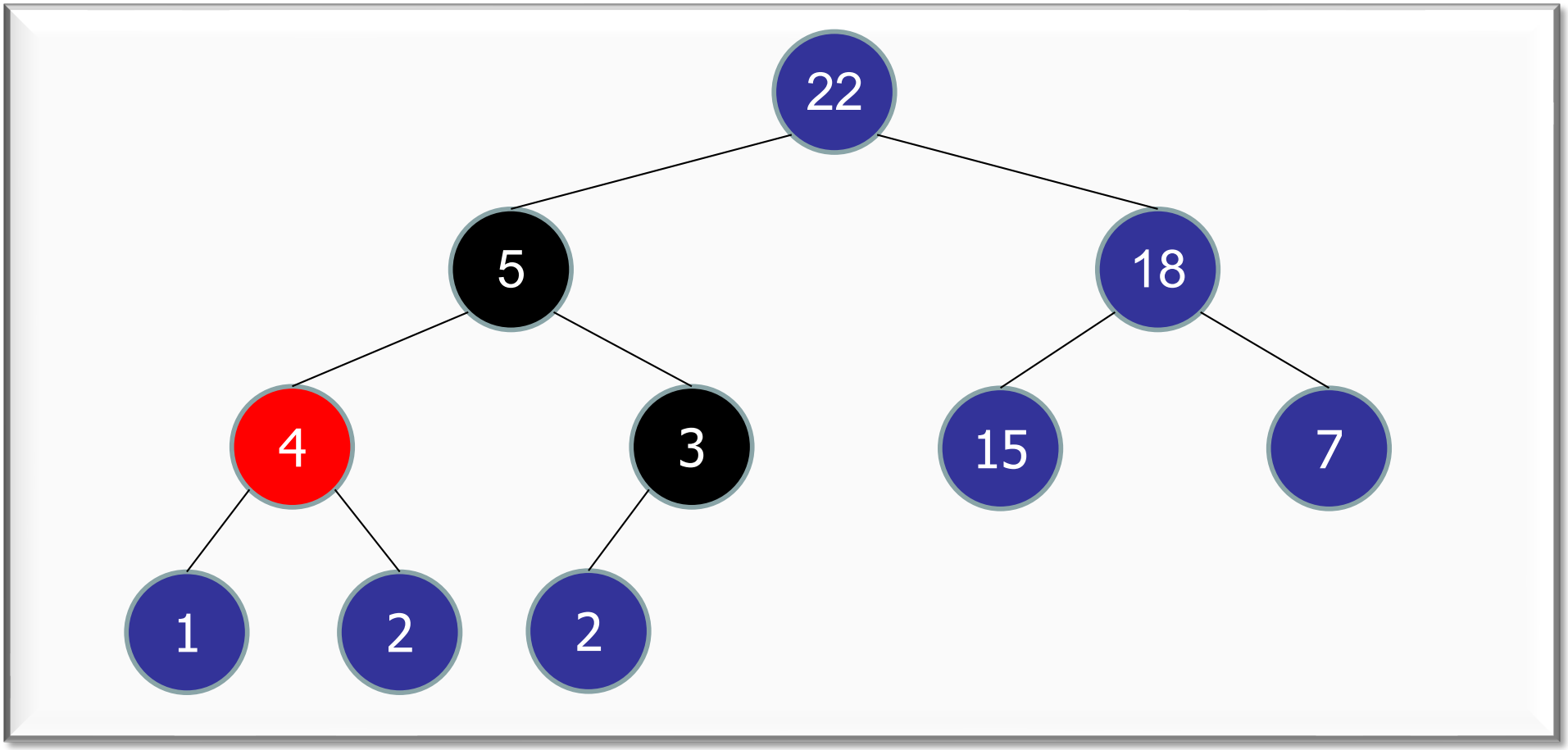
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Heap Operations

`decreaseKey(28 → 4)` :

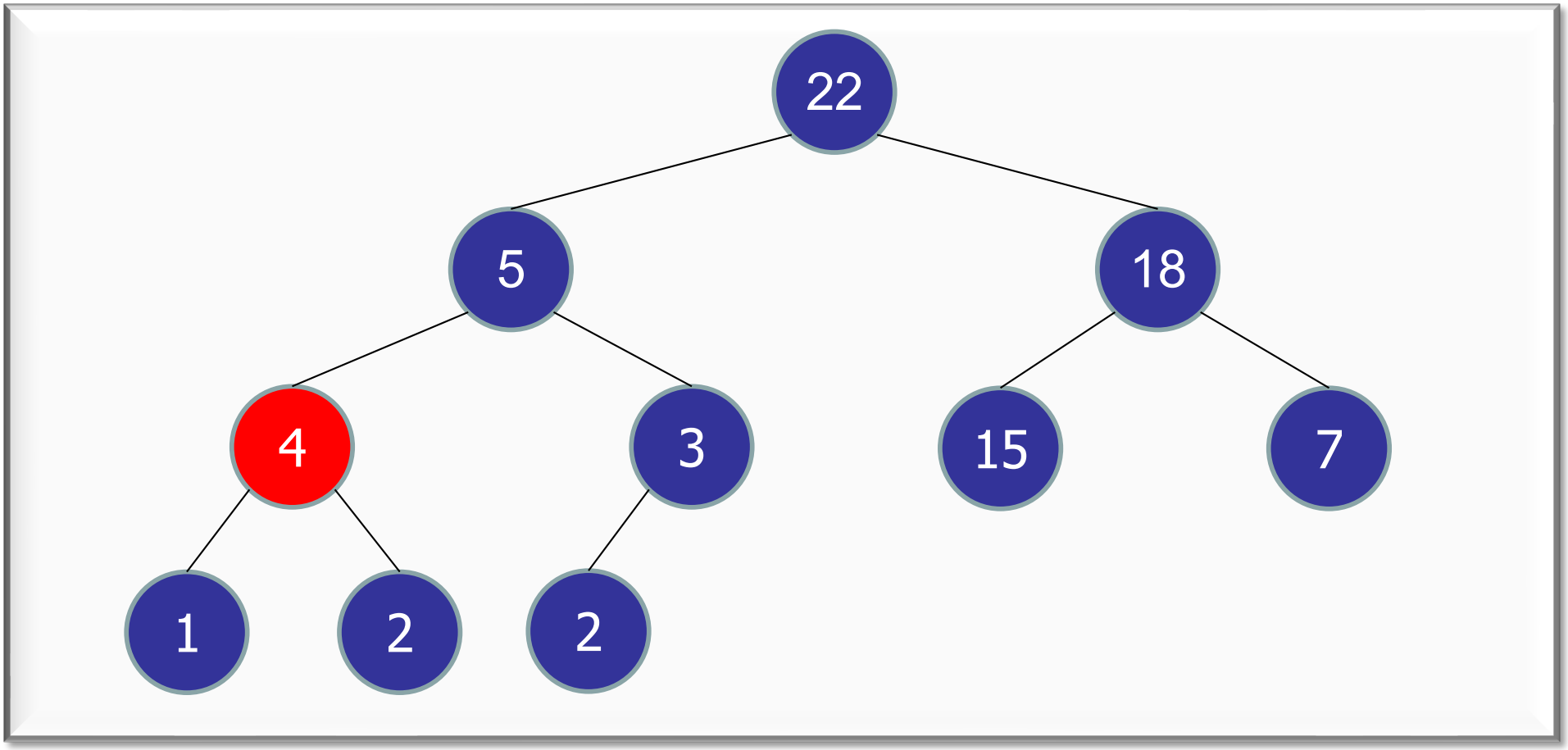
- Step 1: Update the priority
- Step 2: bubbleDown(4)



Heap Operations

`decreaseKey(28 → 4)` :

- Step 1: Update the priority
- Step 2: bubbleDown(4)



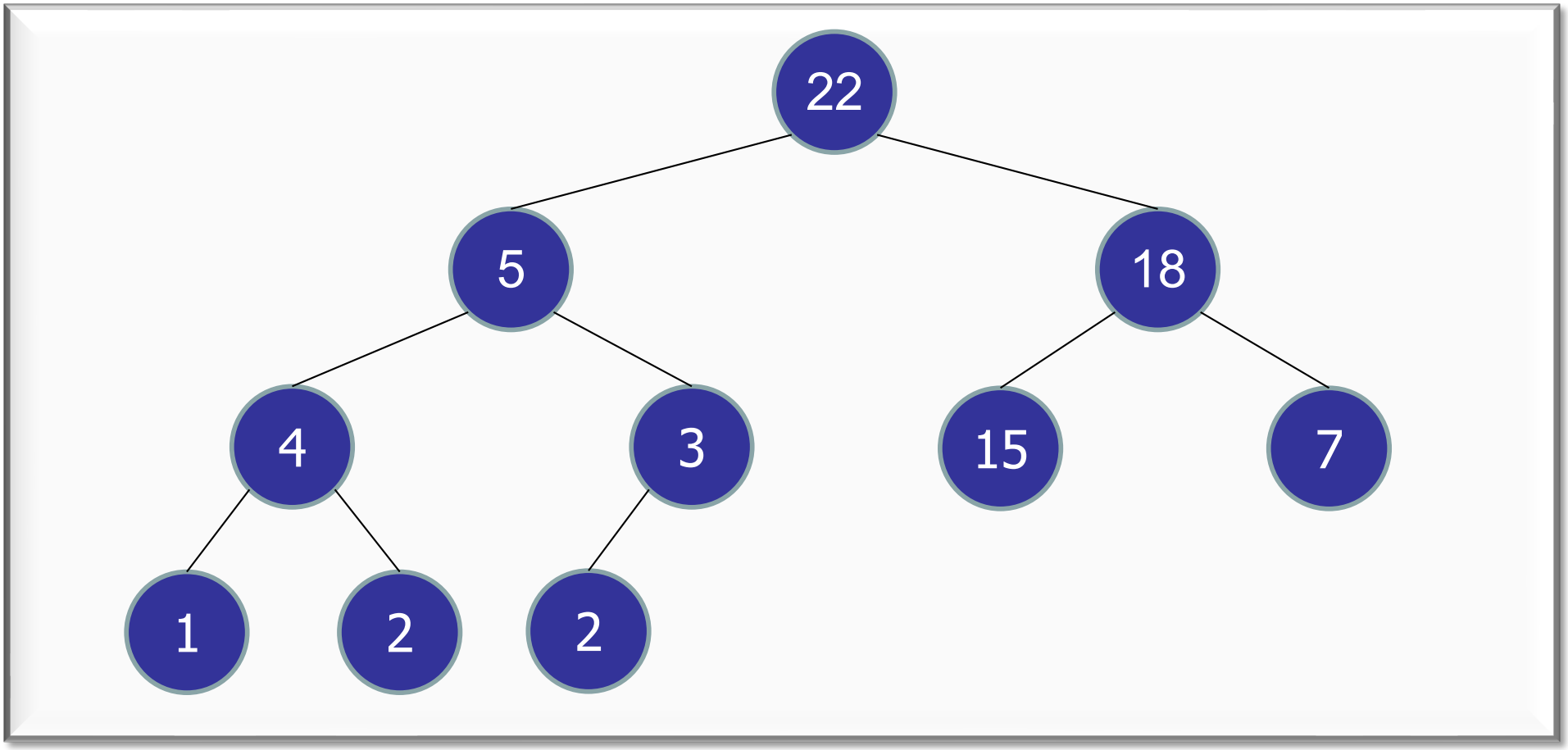
Heap Operations

```
bubbleDown(Node v)
  while (!leaf(v)) {
     leftP = priority(left(v));
     rightP = priority(right(v));
     maxP = max(leftP, rightP, priority(v));
     if (leftP == max) {
           swap(v, left(v));
           v = left(v); }
     else if (rightP == max) {
           swap(v, right(v));
           v = right(v); }
     else return;
  }
```

Heap Operations

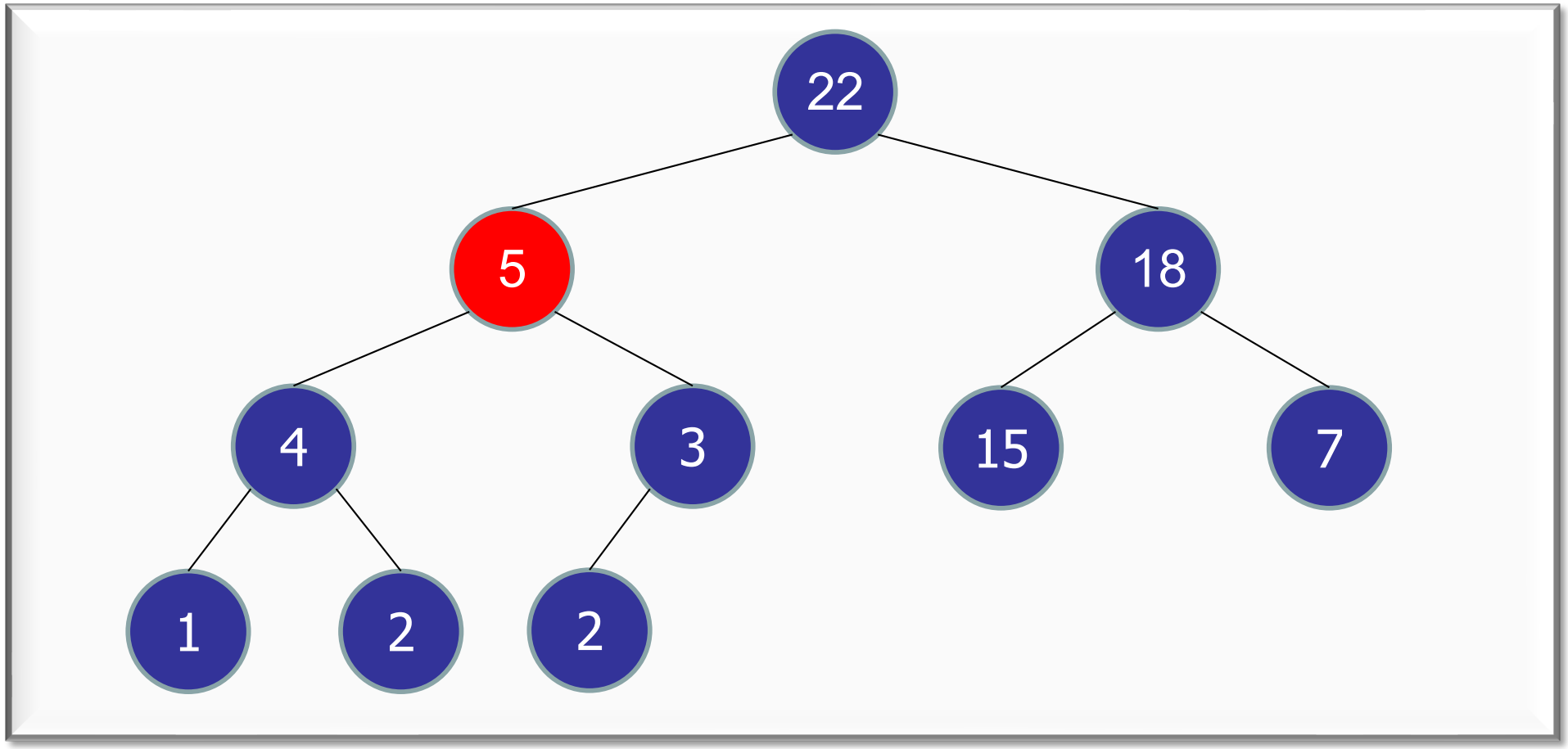
`decreaseKey(. . .)`:

- On completion, heap order is restored.
- Complete binary tree.



Heap Operations

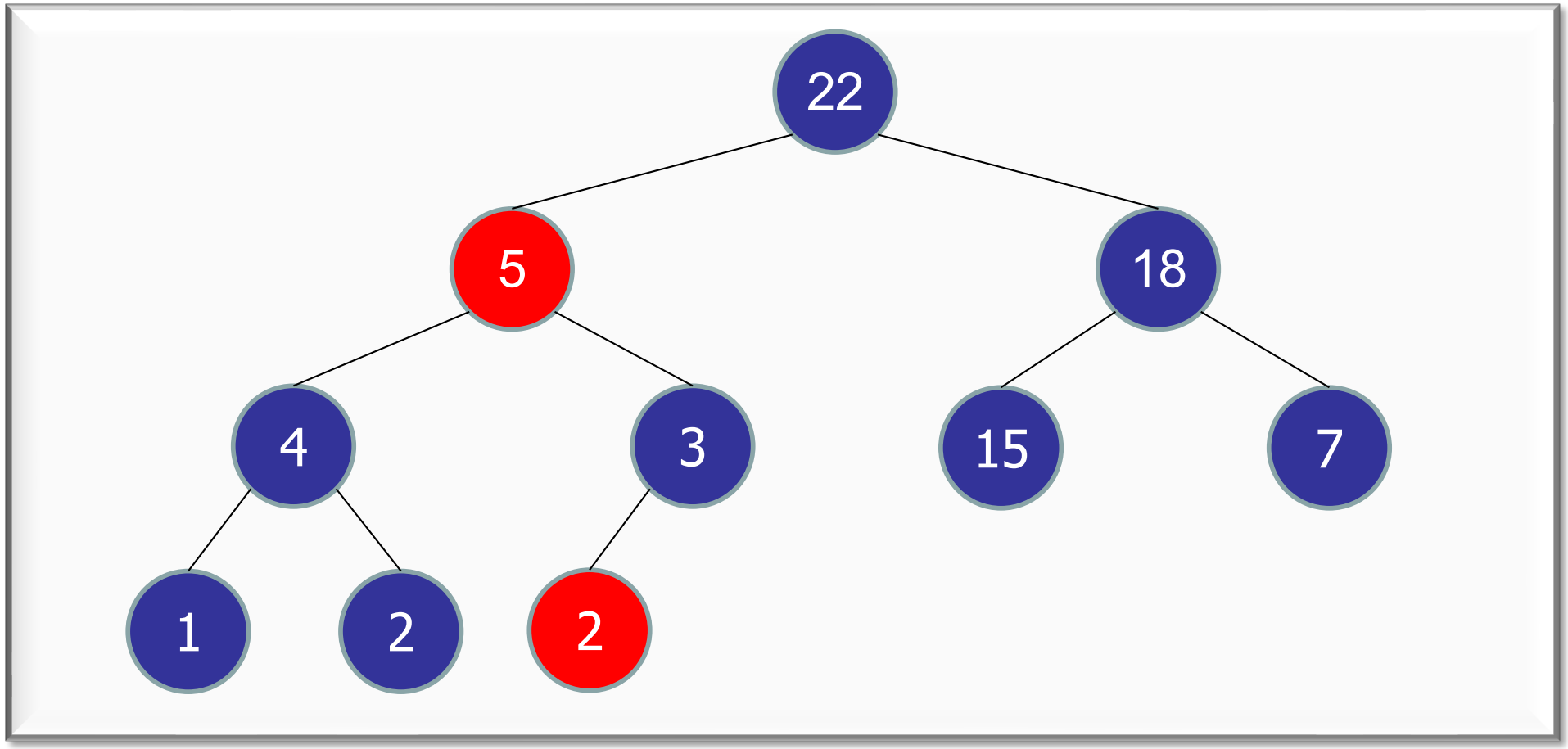
`delete(5)` :



Heap Operations

`delete(5)` :

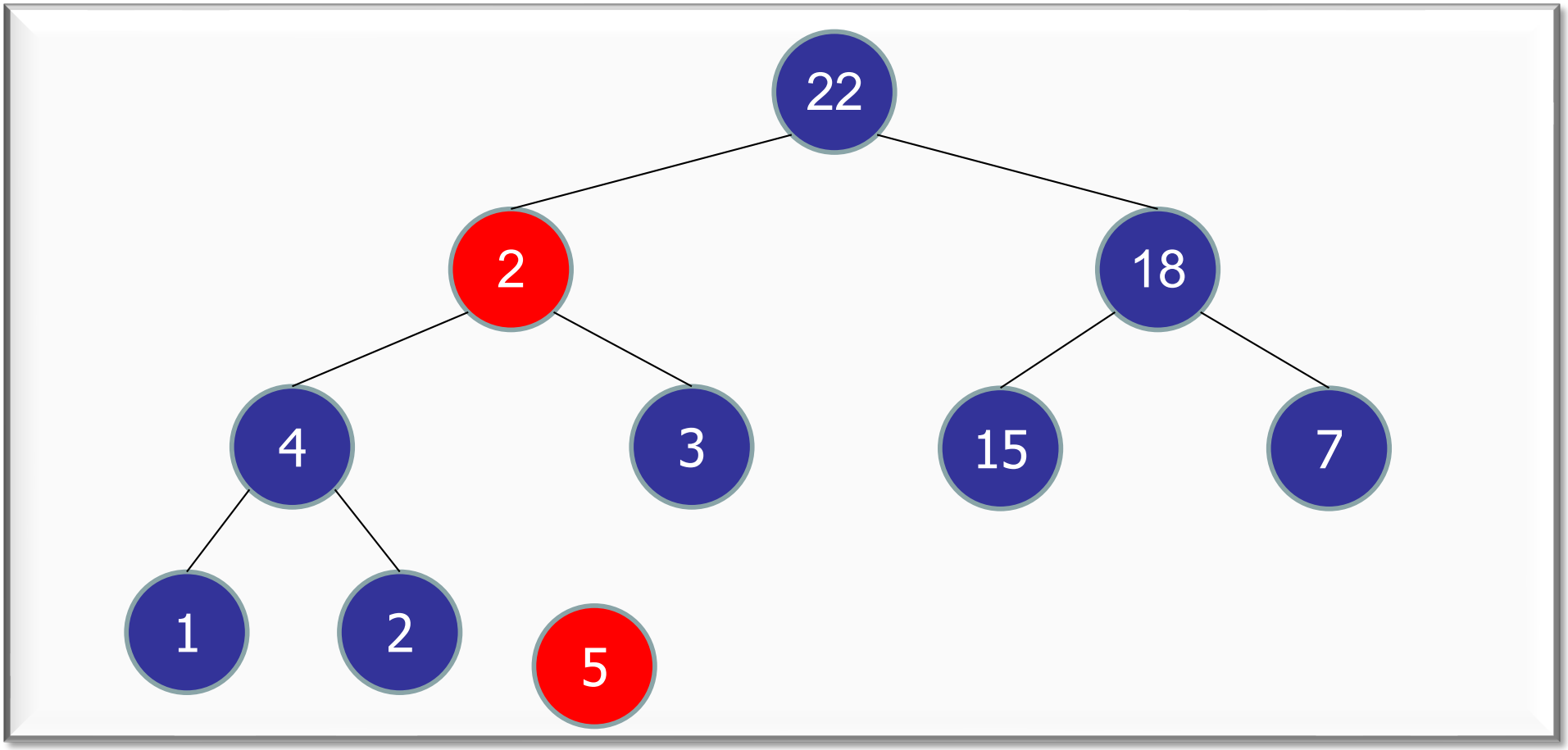
- swap(5, last())



Heap Operations

`delete(5)` :

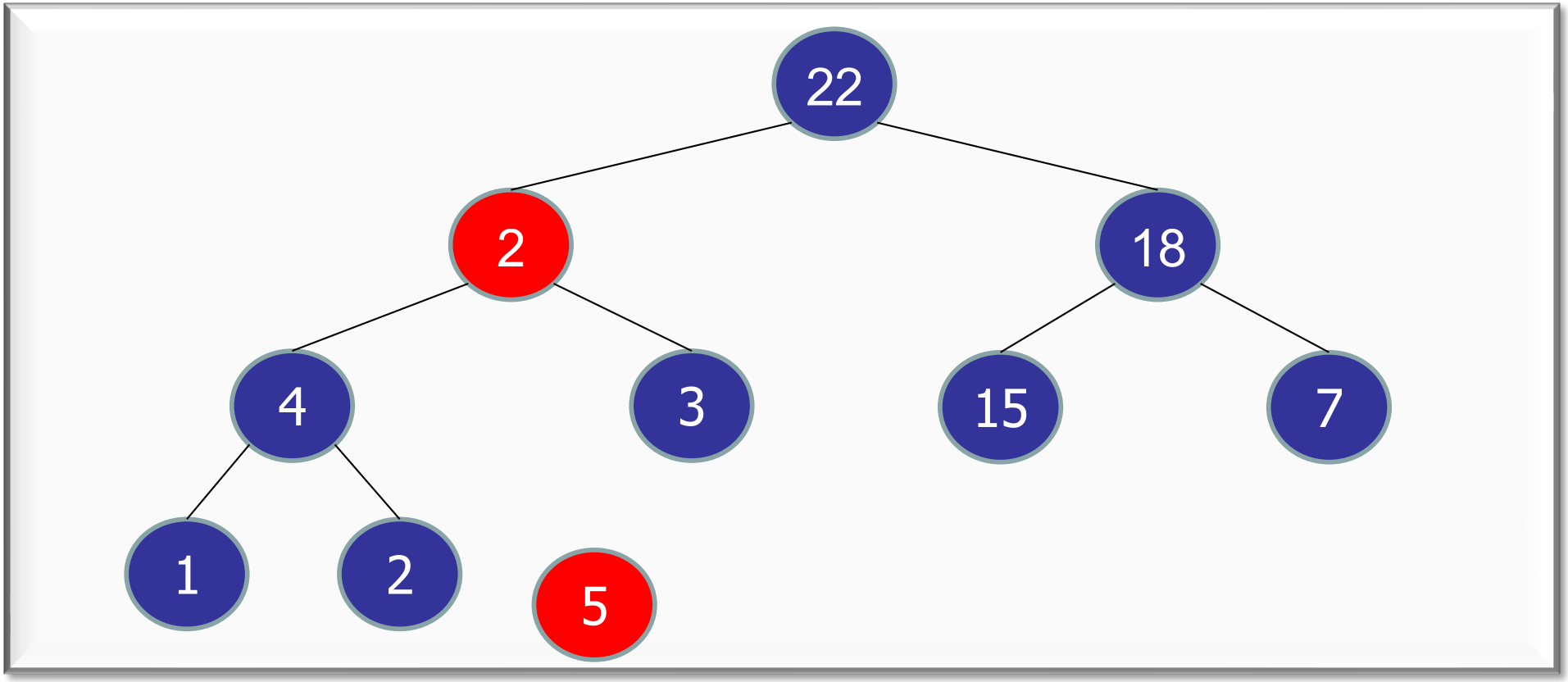
- swap(5, last())
- remove(last())



Heap Operations

`delete(5)` :

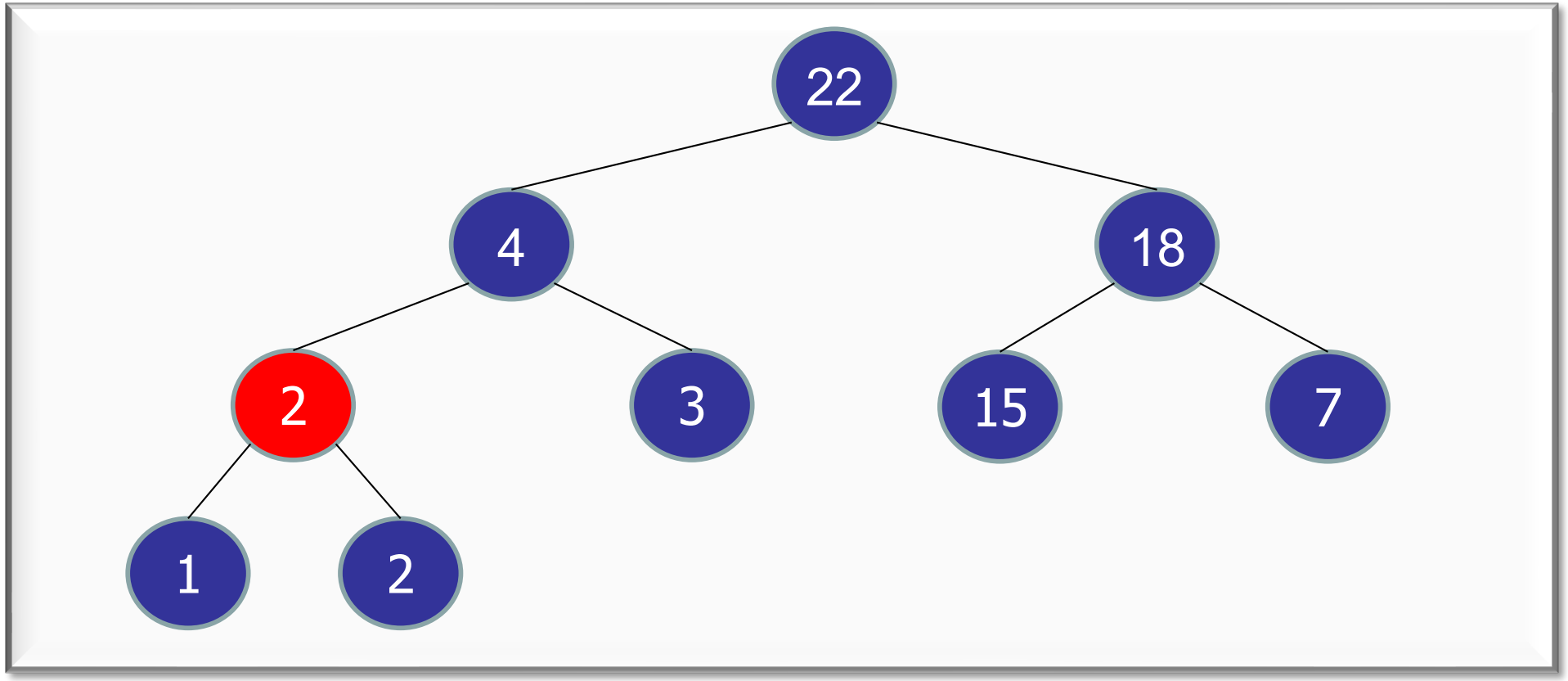
- swap(5, last())
- remove(last())
- bubbleDown(2)



Heap Operations

`delete(5)` :

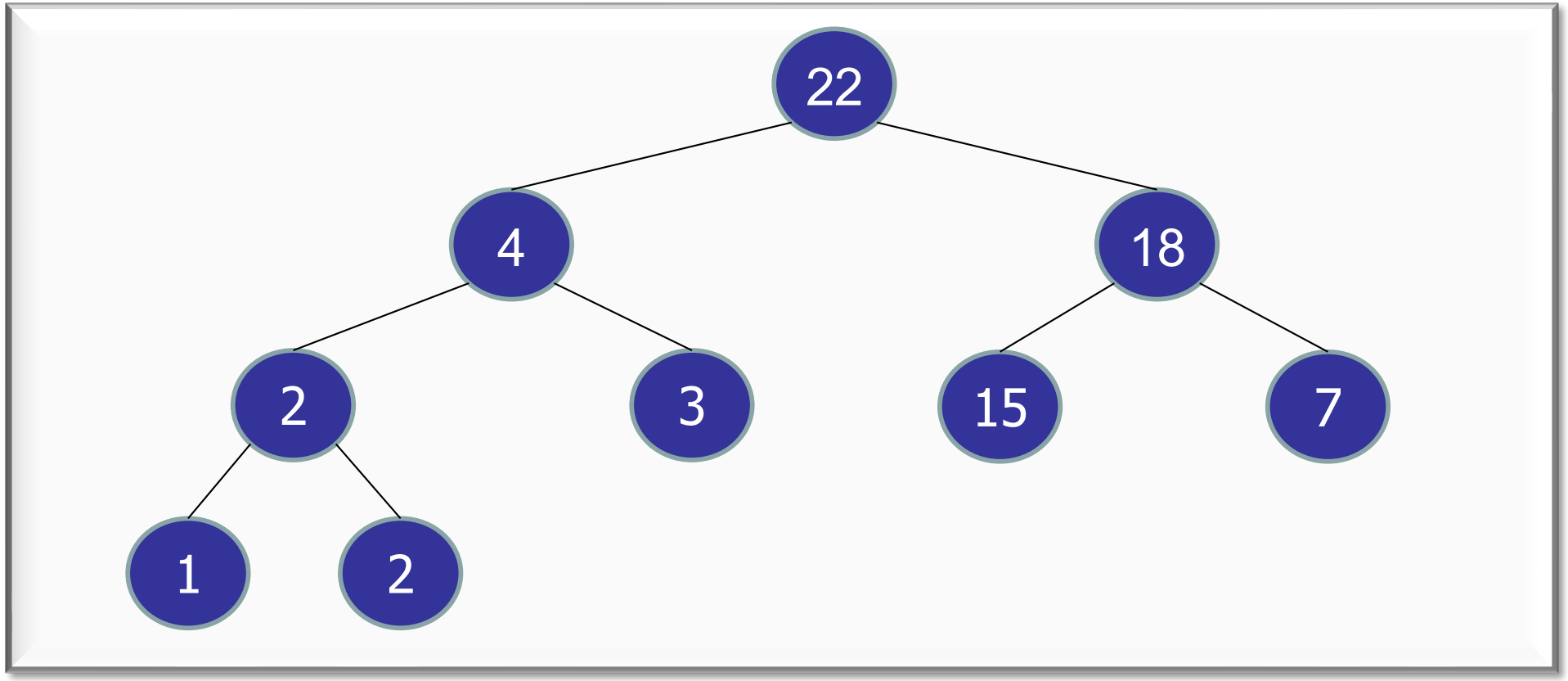
- swap(5, last())
- remove(last())
- bubbleDown(2)



Heap Operations

`delete(5)` :

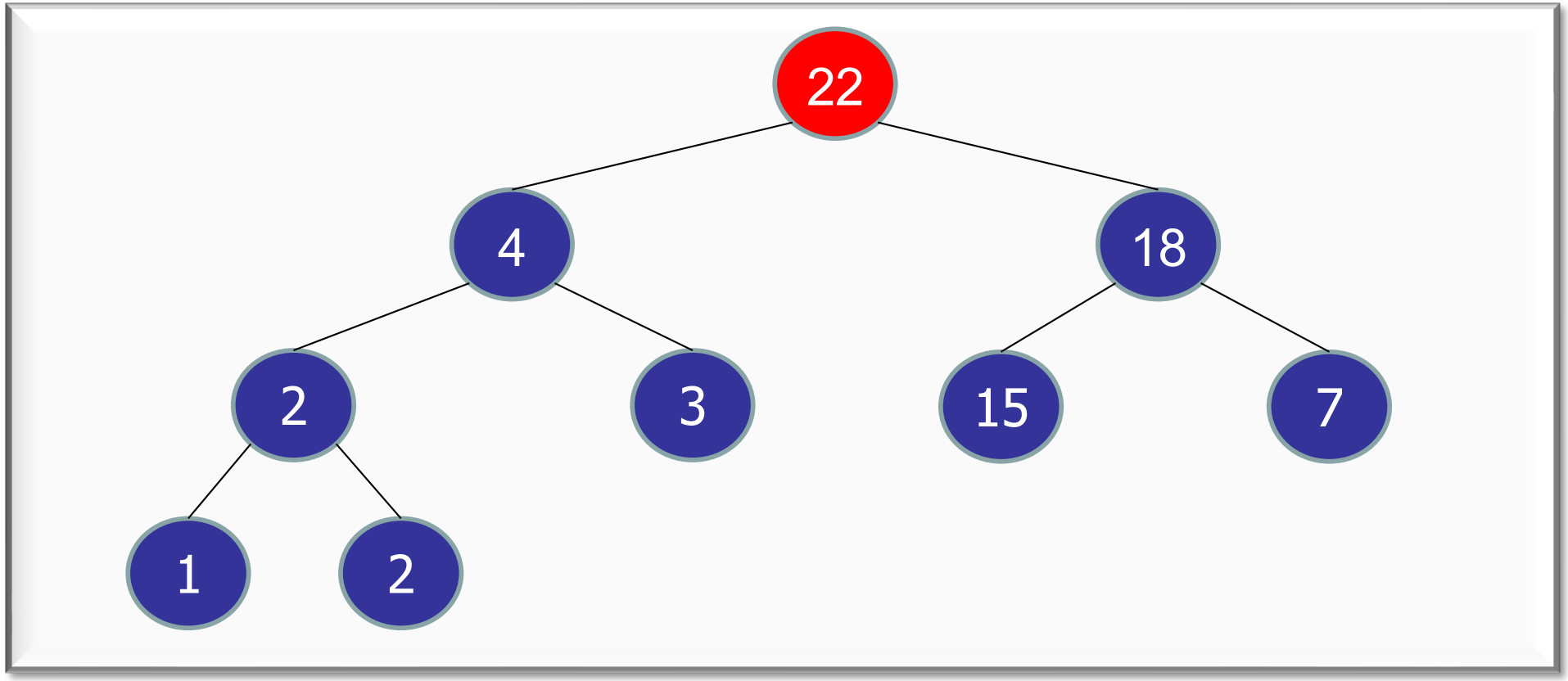
- swap(5, last())
- remove(last())
- bubbleDown(2)



Heap Operations

`extractMax()` :

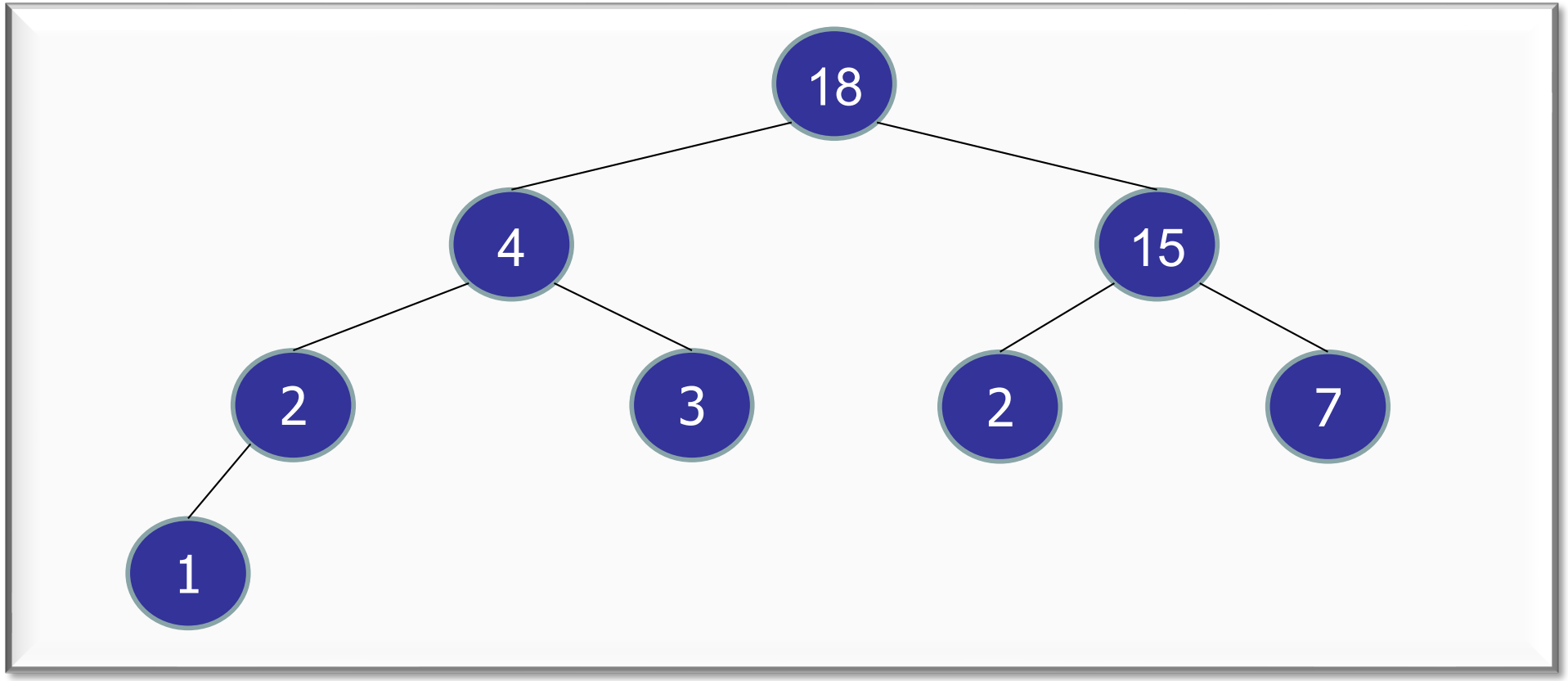
- Node v = root;
- delete(root);



Heap Operations

`extractMax()` :

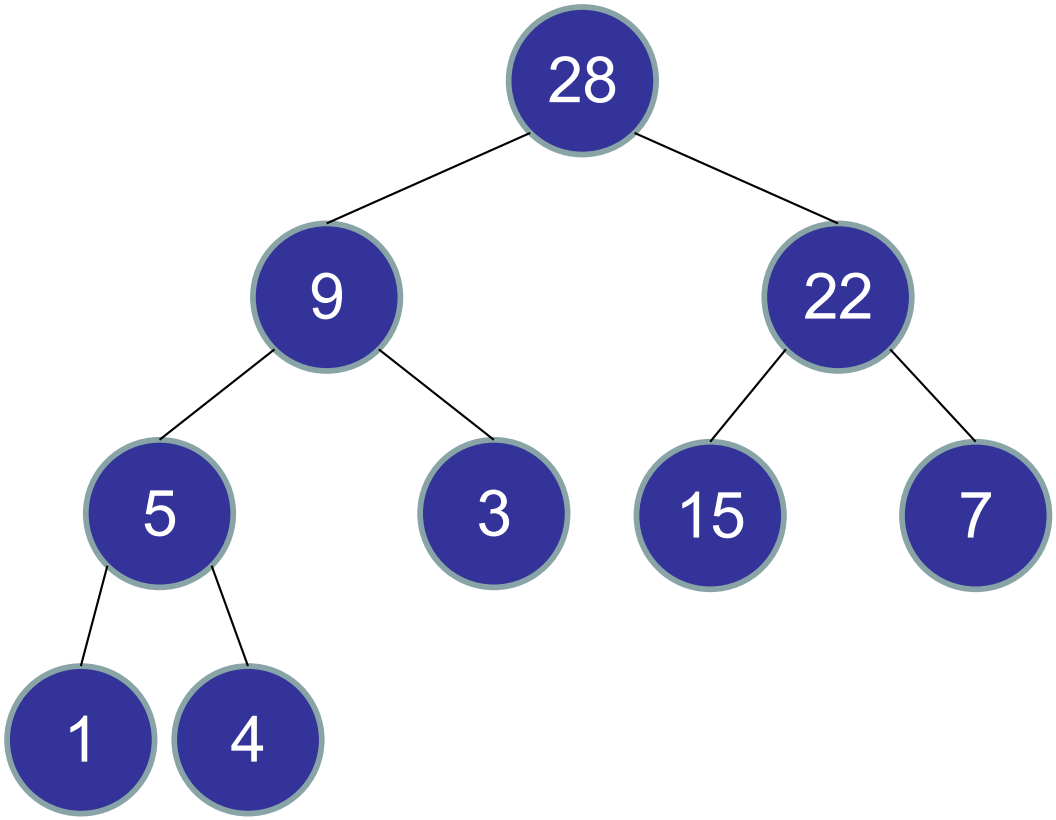
- Node v = root;
- delete(root);



(Max) Priority Queue

Heap Operations: O(log n)

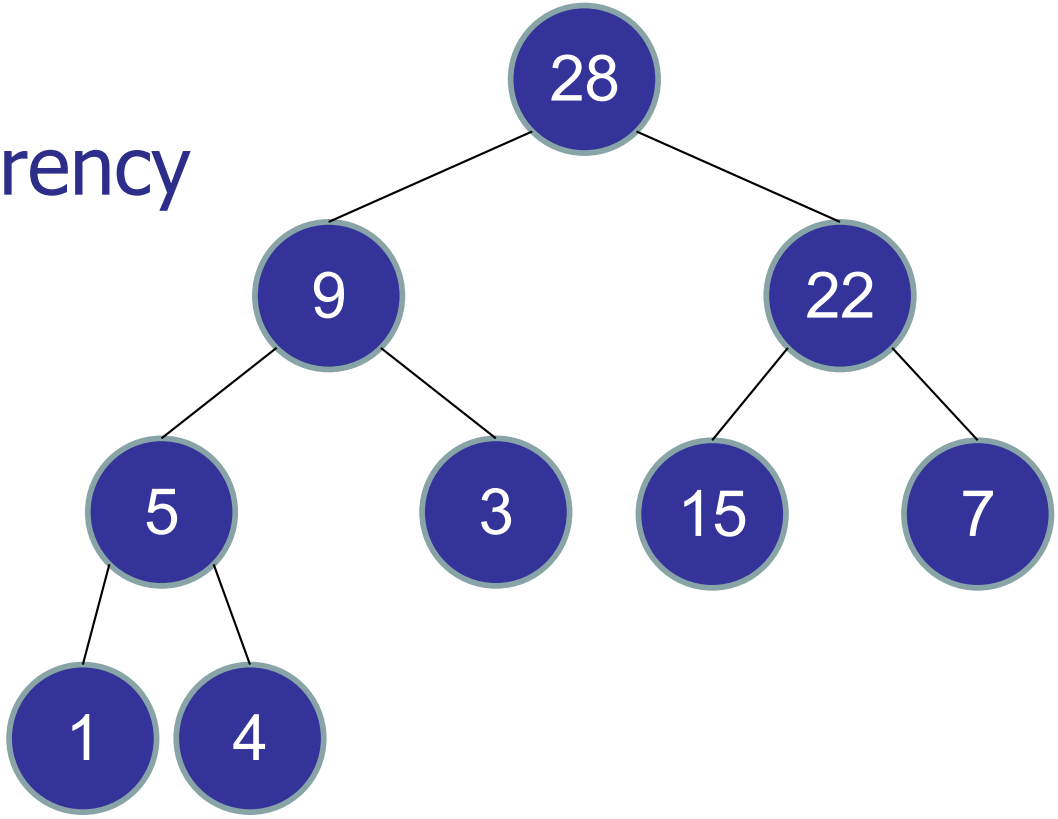
- insert
- extractMax
- increaseKey
- decreaseKey
- delete



(Max) Priority Queue

Heap vs. AVL Tree

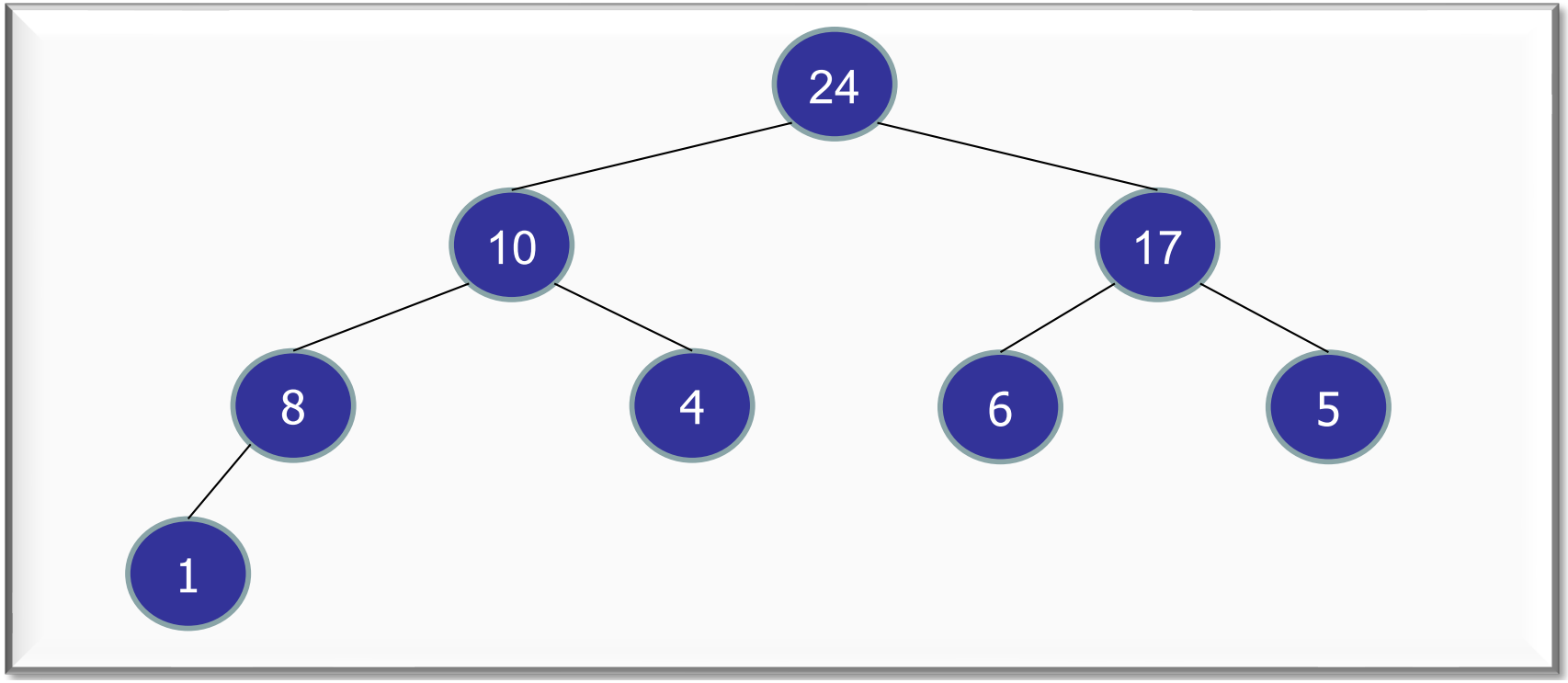
- Same asymptotic cost for operations
- Faster real cost (no constant factors!)
- Simpler: no rotations
- Slightly better concurrency



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

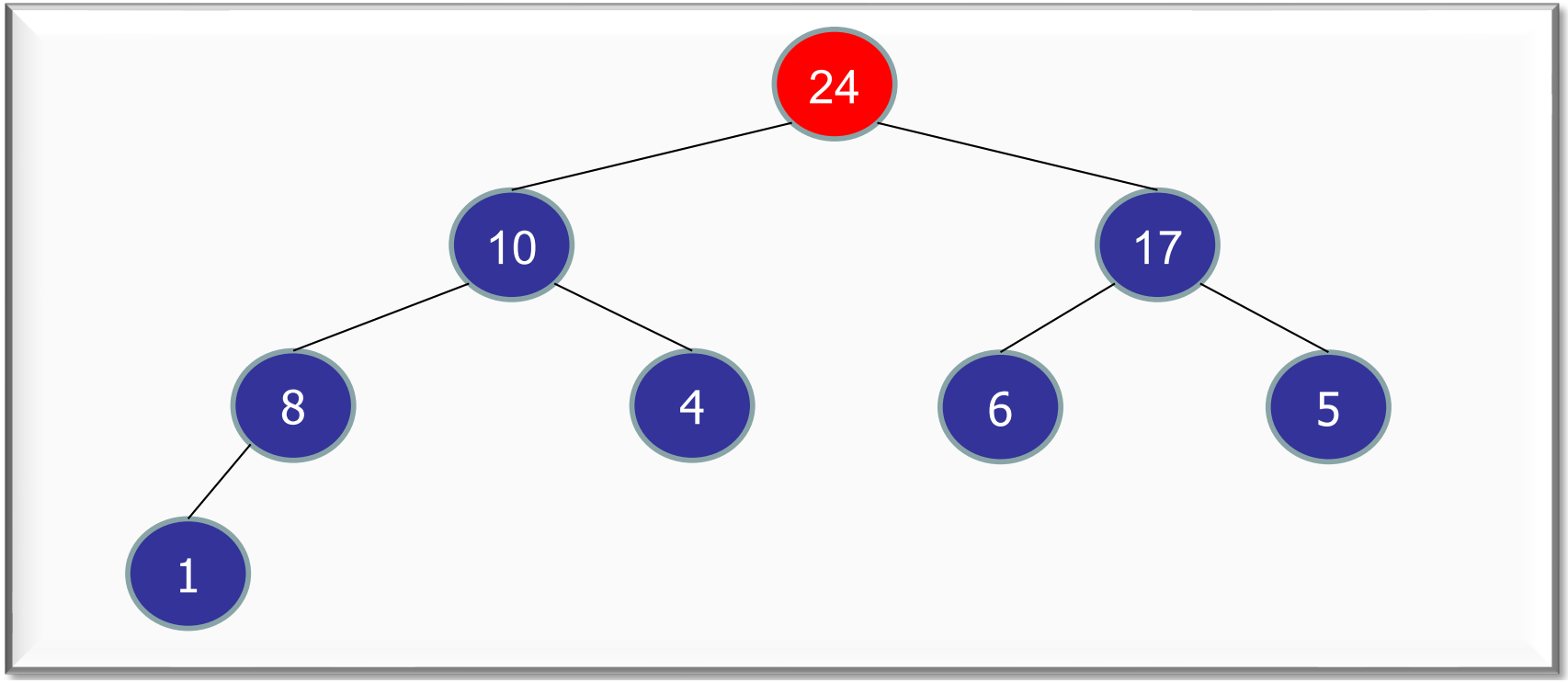
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

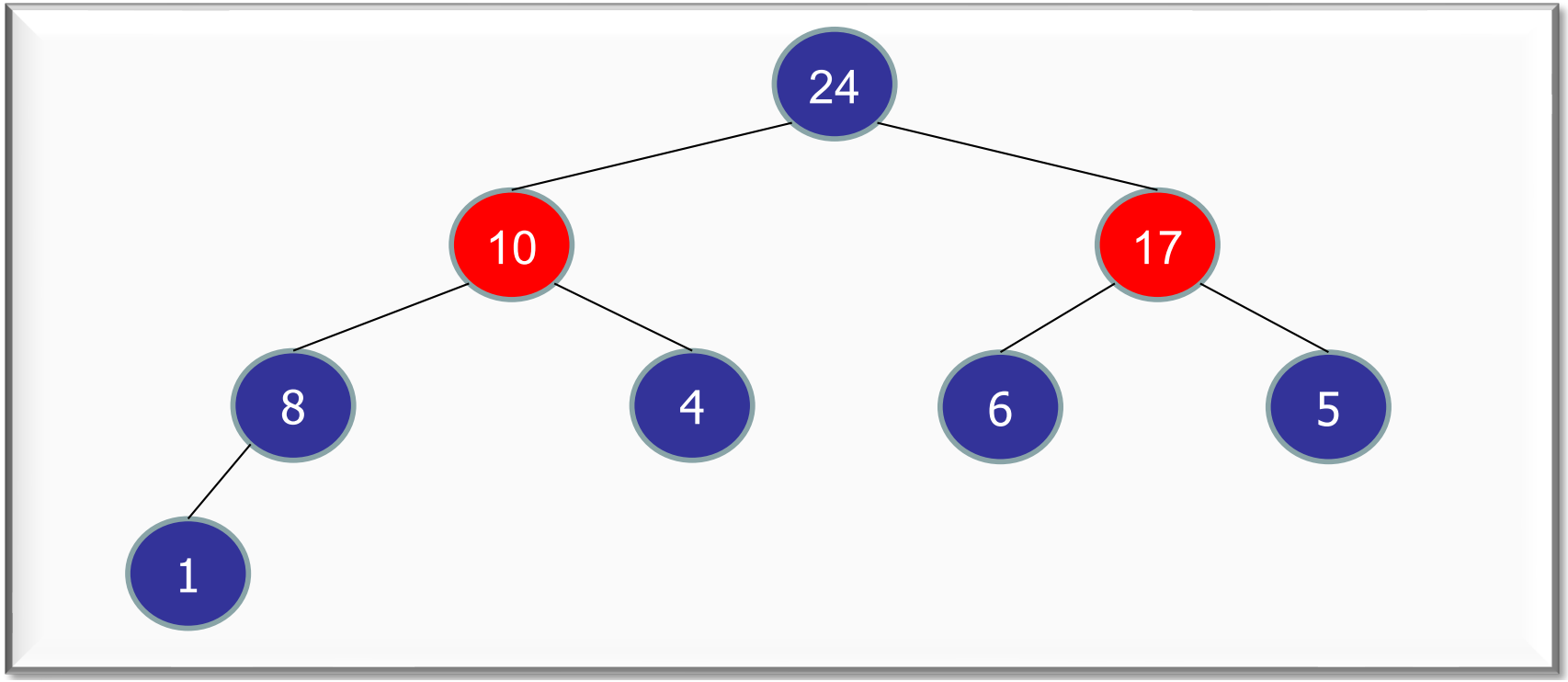
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

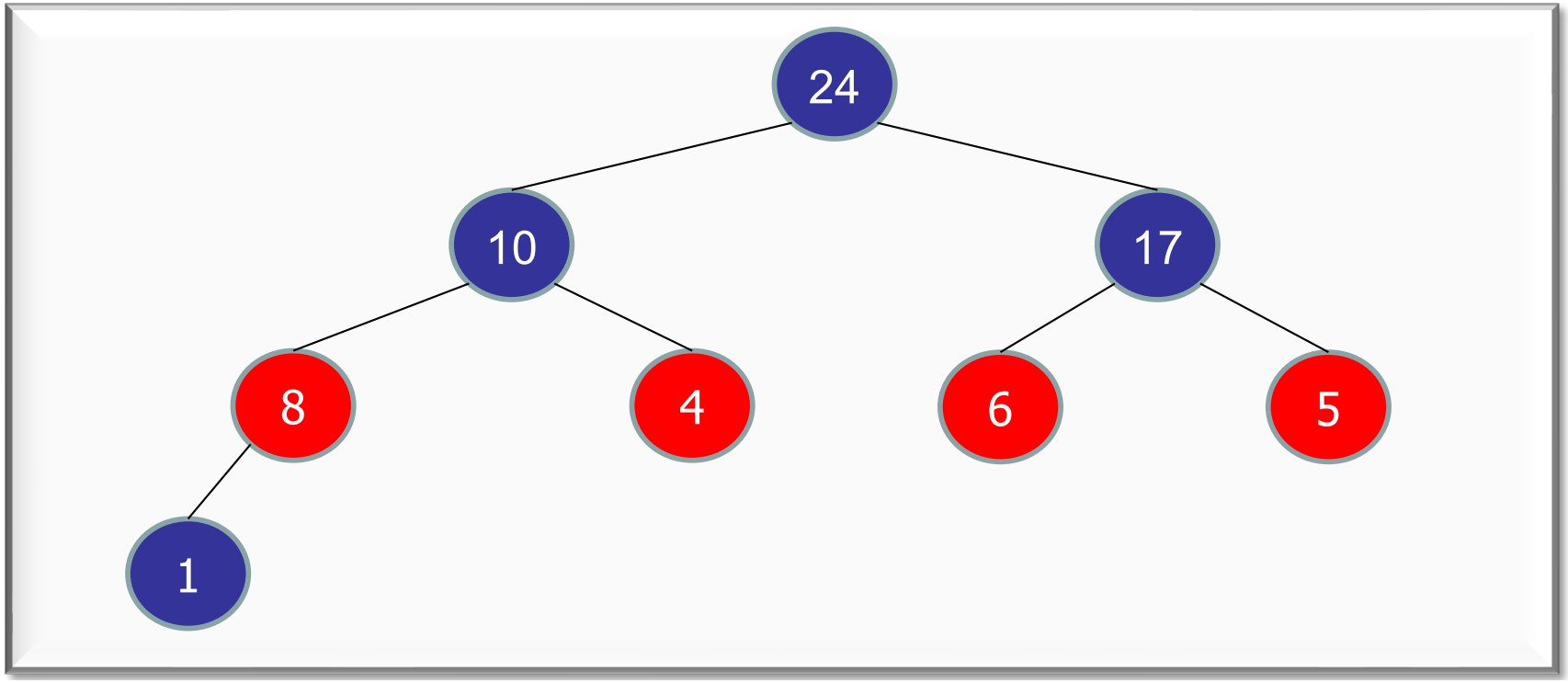
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	7	1	



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

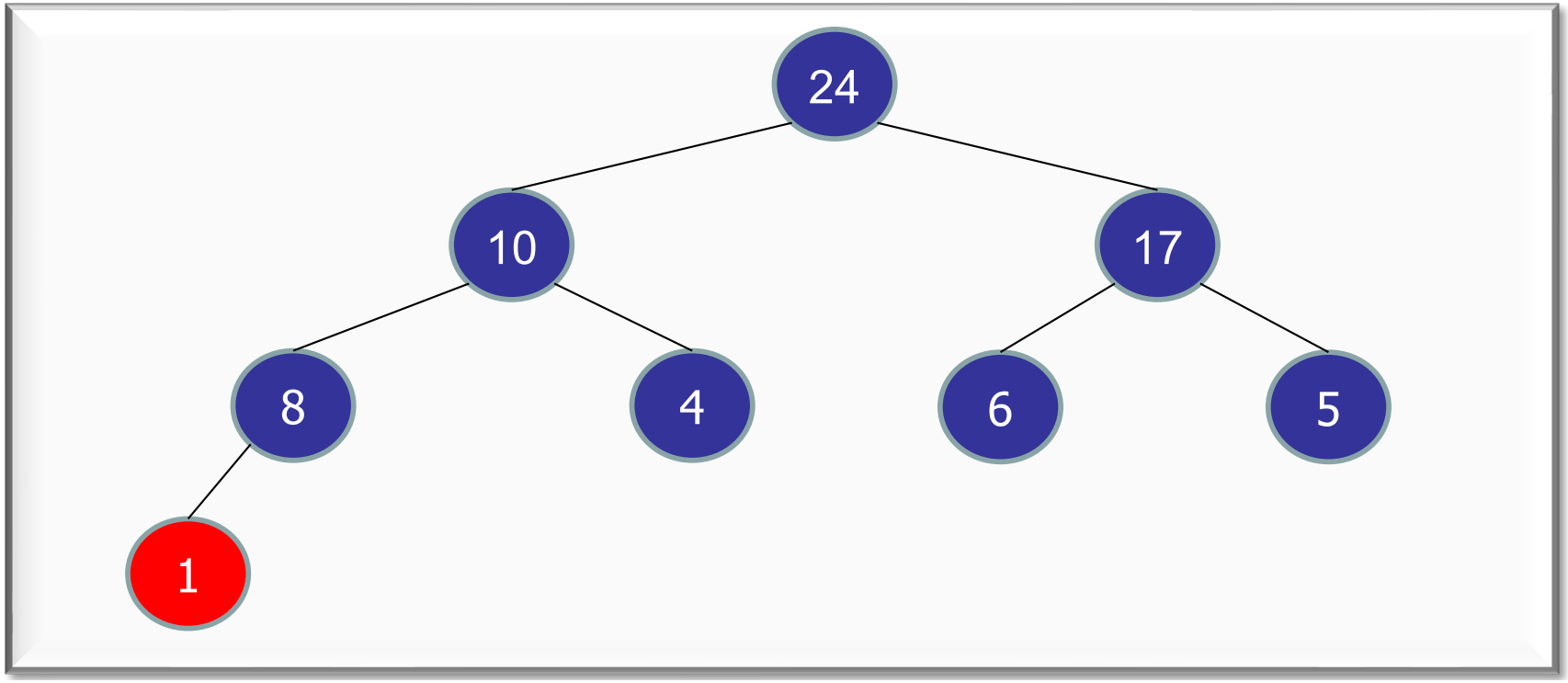
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

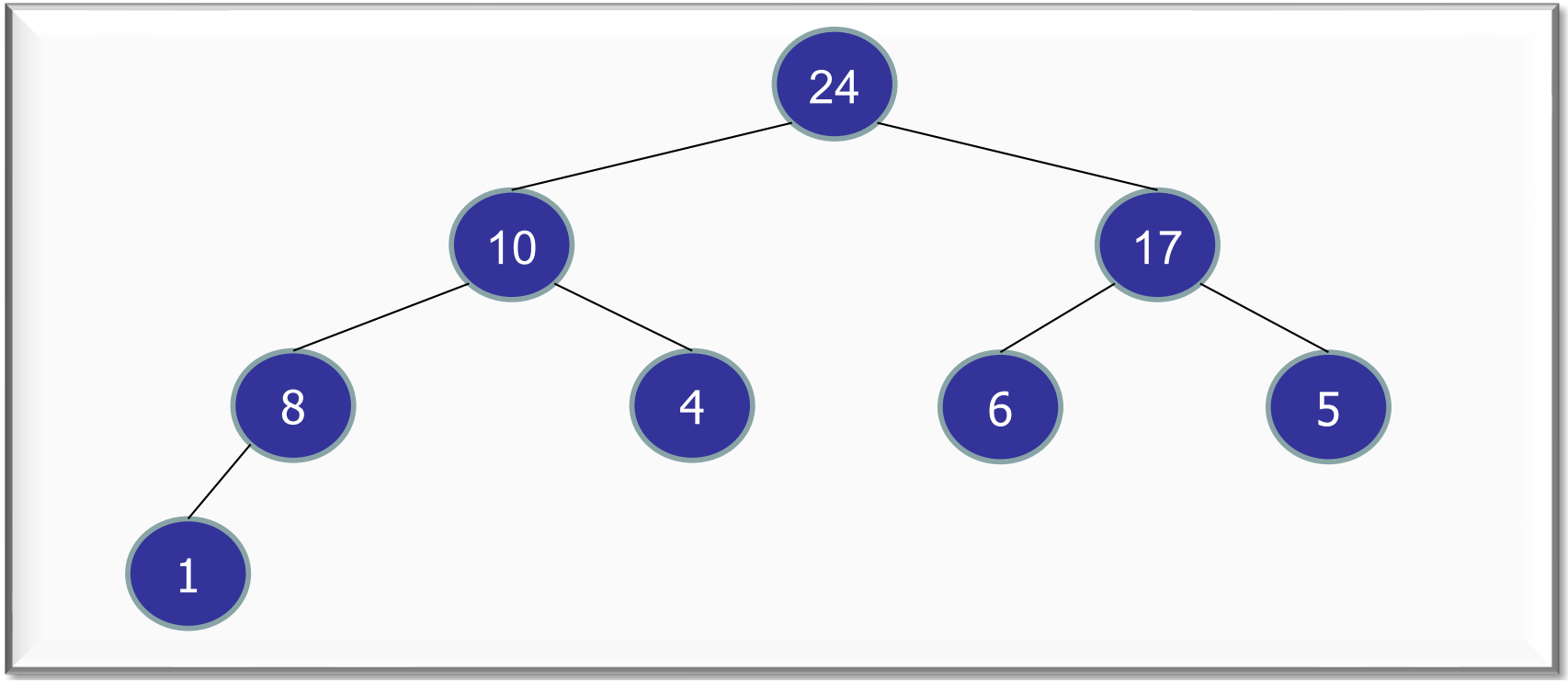
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

```
insert(15) :
```

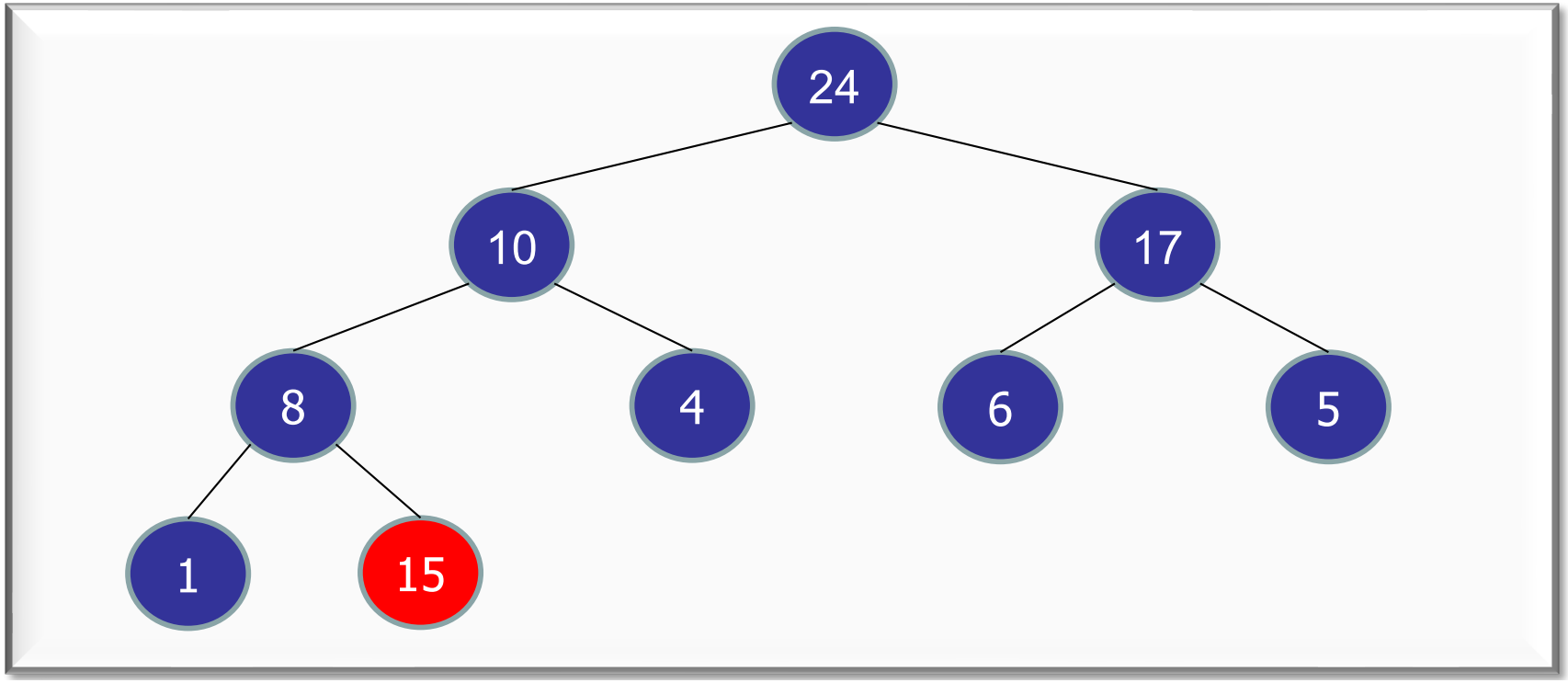
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



Store Tree in an Array

`insert(15) :`

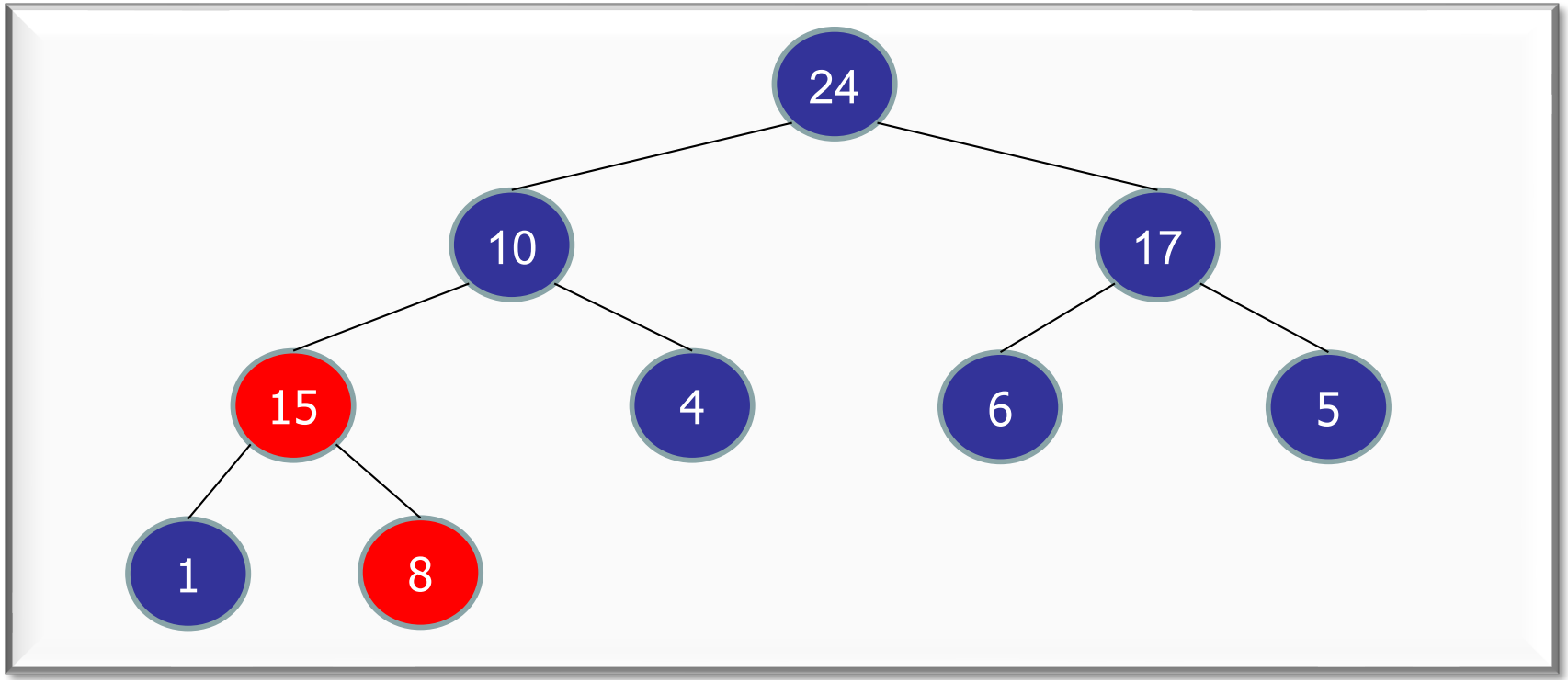
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	15



Store Tree in an Array

`insert(15) :`

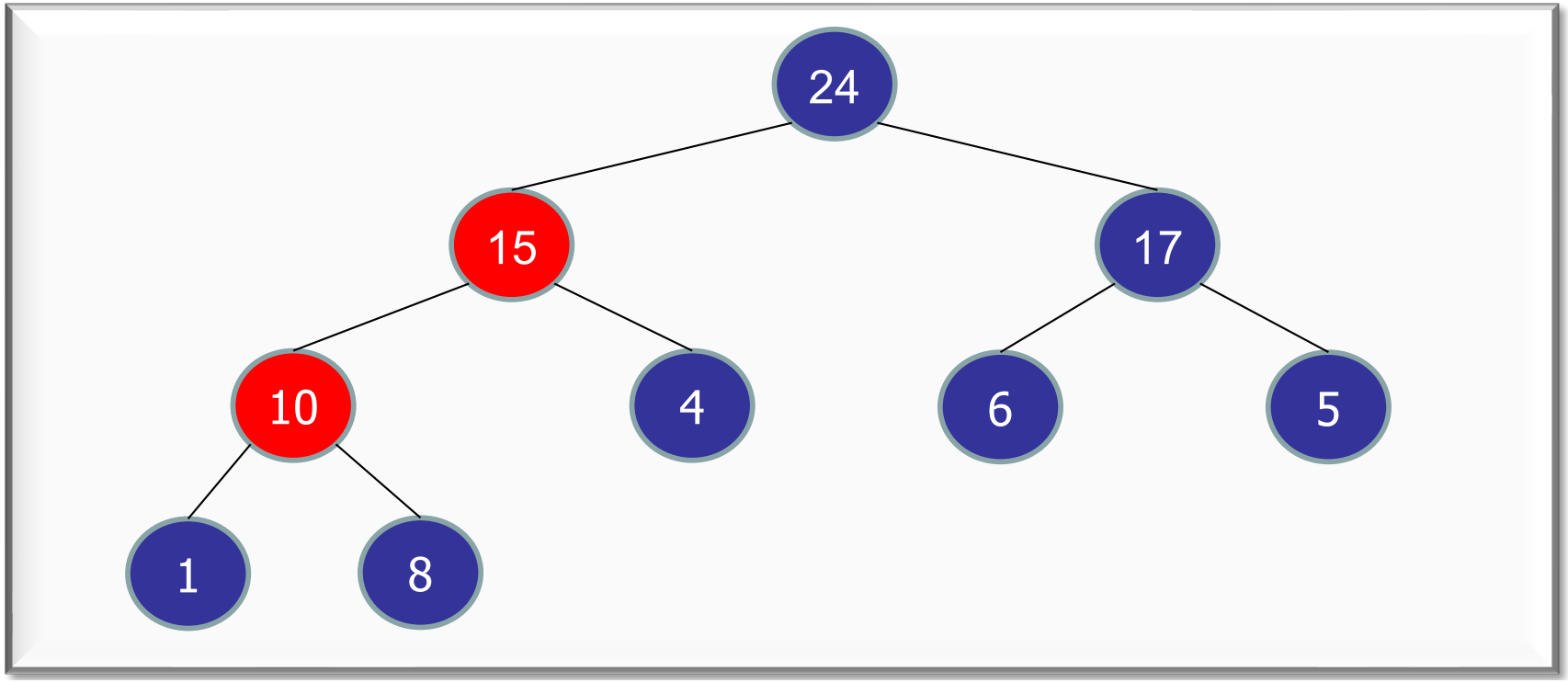
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	15	4	6	5	1	8



Store Tree in an Array

`insert(15) :`

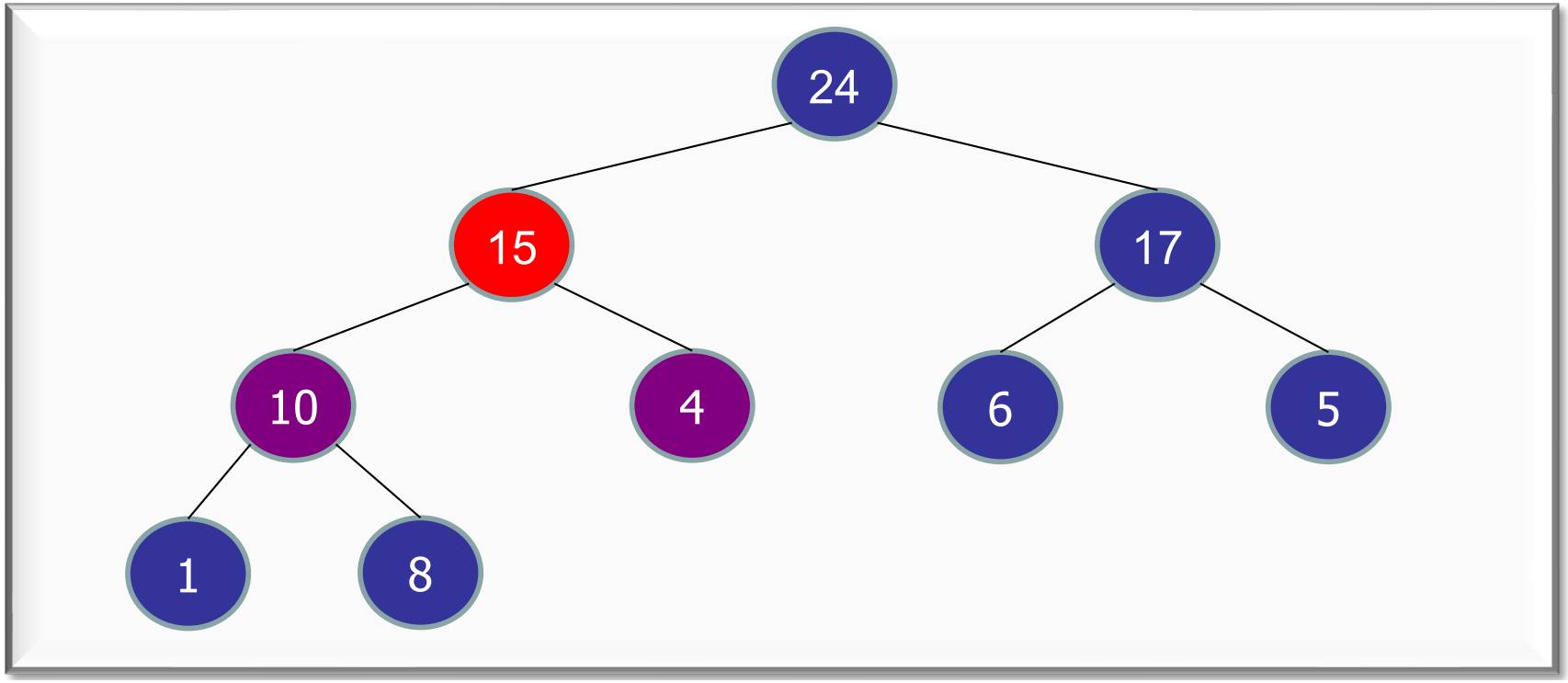
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



Store Tree in an Array

```
left(x)  = ??
right(x) = ??
```

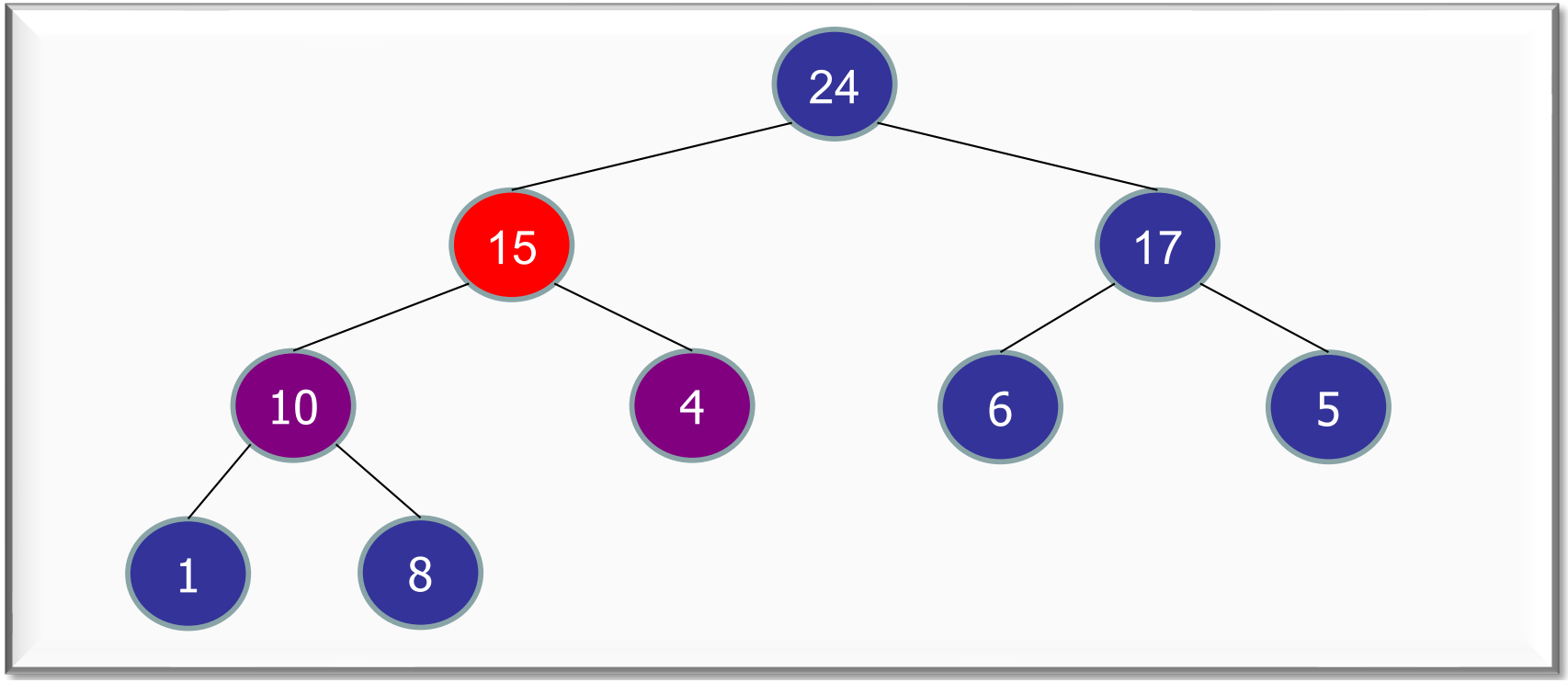
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



Store Tree in an Array

```
left(x)  = 2x+1
right(x) = 2x+2
```

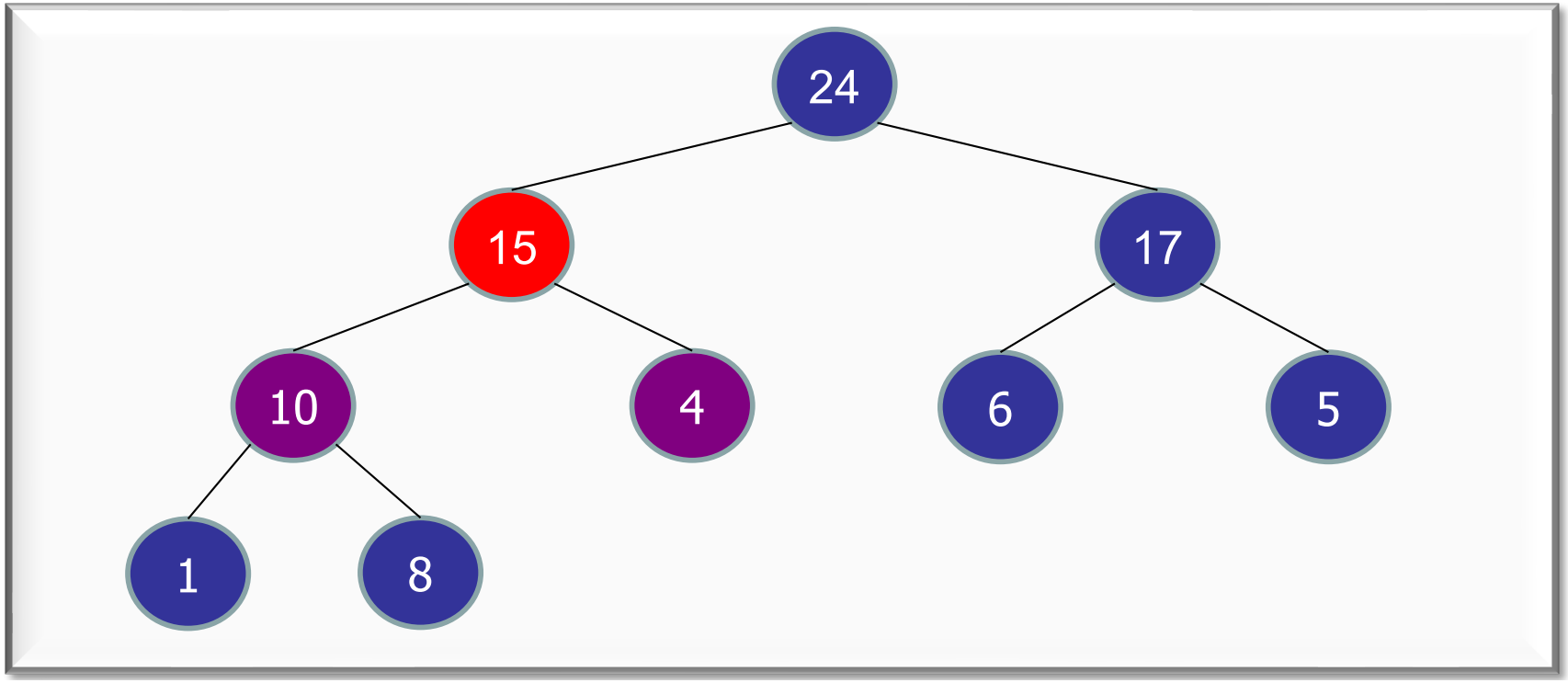
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



Store Tree in an Array

```
parent(x)  = floor((x-1)/ 2)
```

array slot	0	1	2	3	4	5	6	7	8
priority	24	15	17	10	4	6	5	1	8



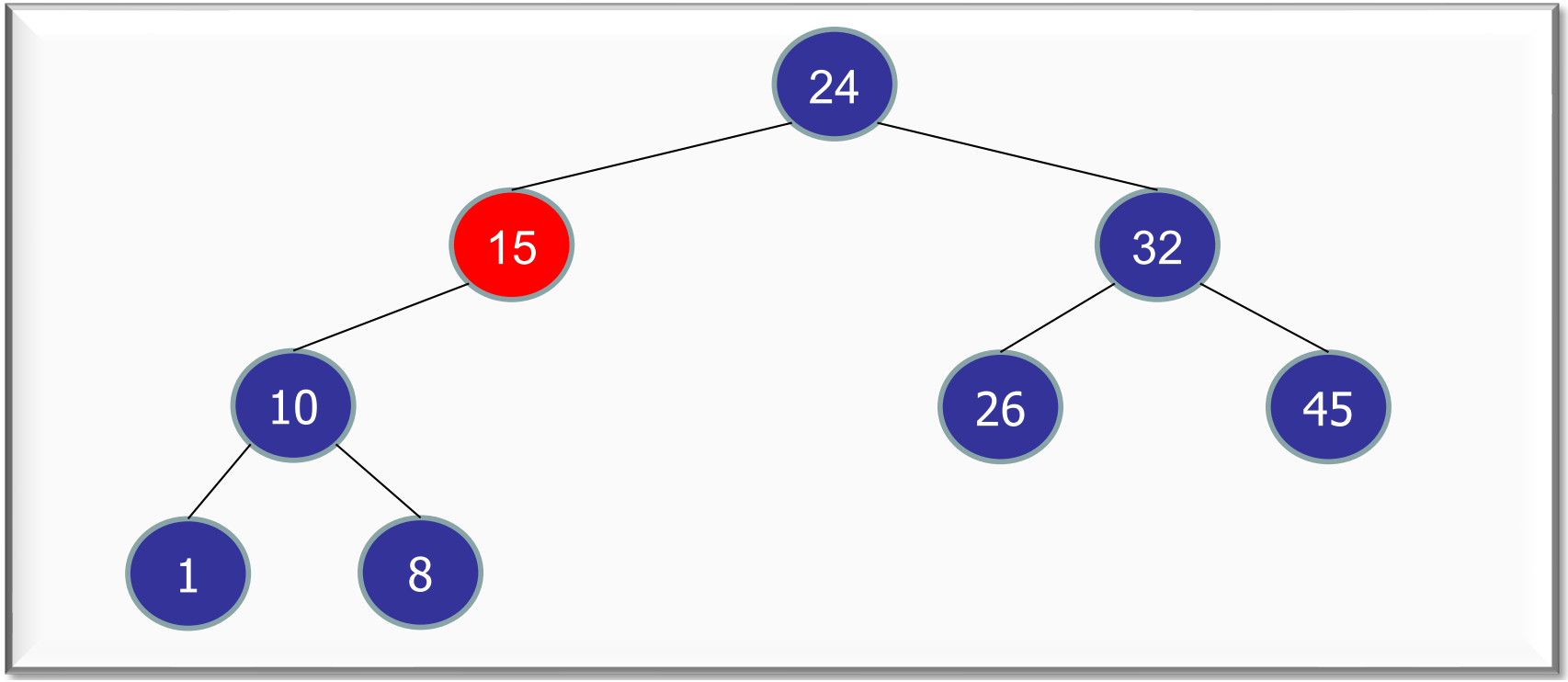
Can we store an AVL tree in an array?

If so, how? If not, why not?

Store Tree in an Array

```
right-rotate(15)
```

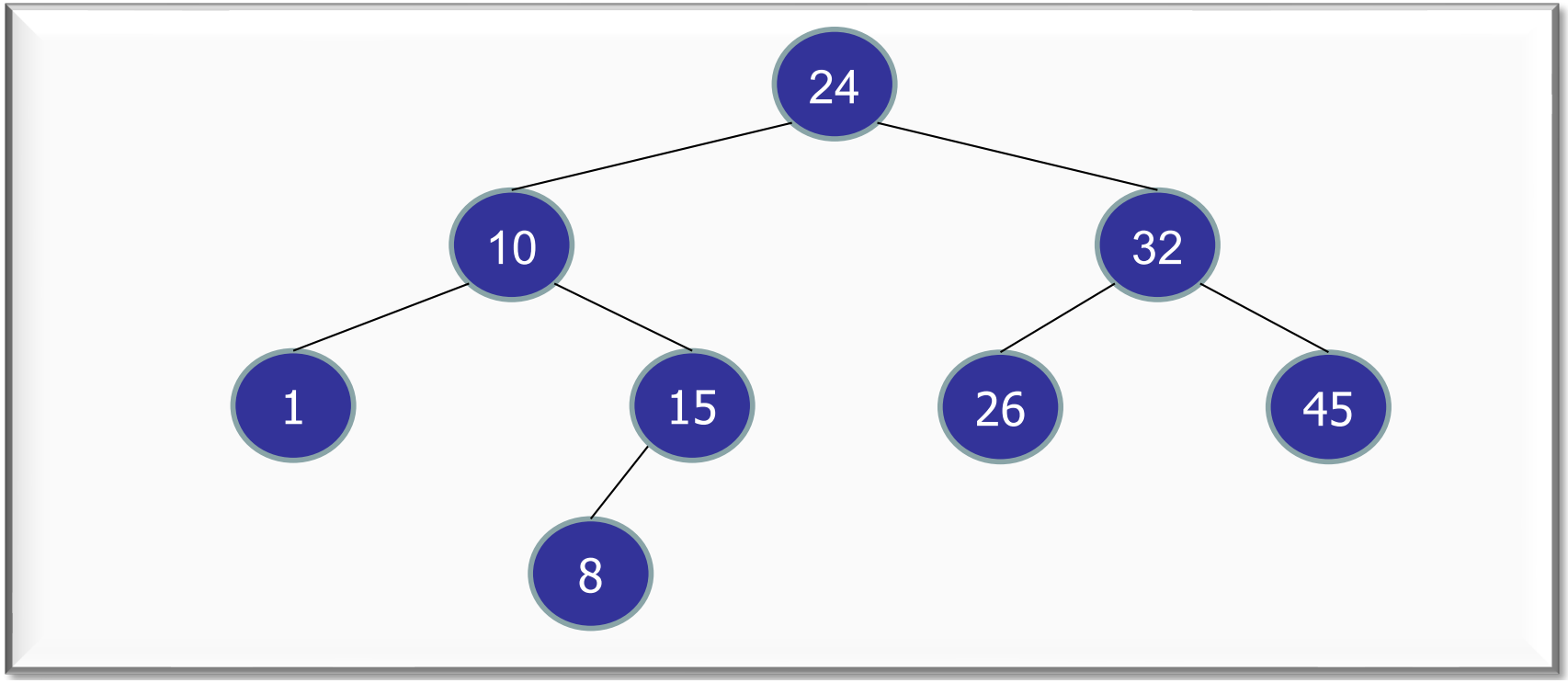
array slot	0	1	2	3	4	5	6	7	8
priority	24	15	32	10		26	45	1	8



Store Tree in an Array

`right-rotate(15)` : **not an O(1) operation!**

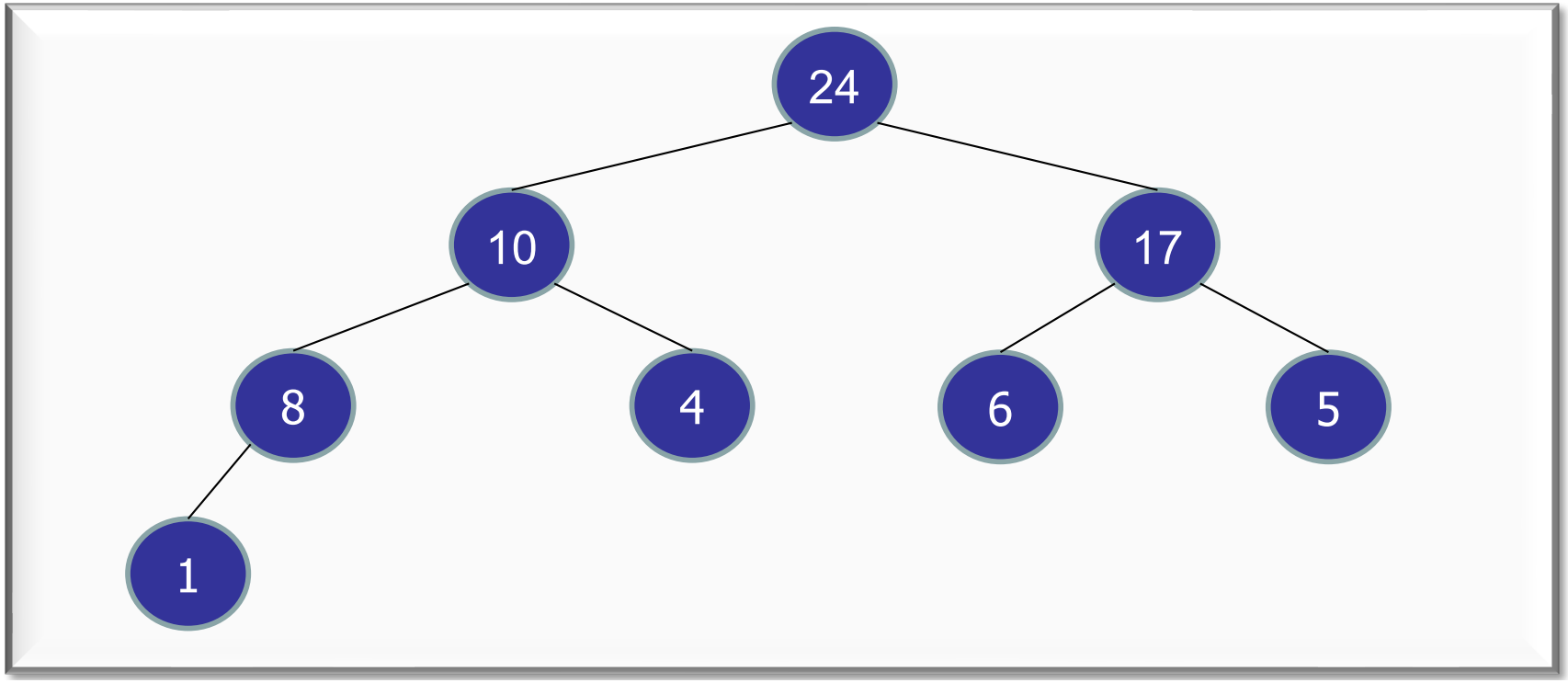
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	32	1	15	26	45	8	



Store Tree in an Array

Map each node in complete binary tree into a slot in an array.

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	



HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

Step 1. Unsorted list ➔ Heap

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3

HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

Step 1. Unsorted list ➔ Heap

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3

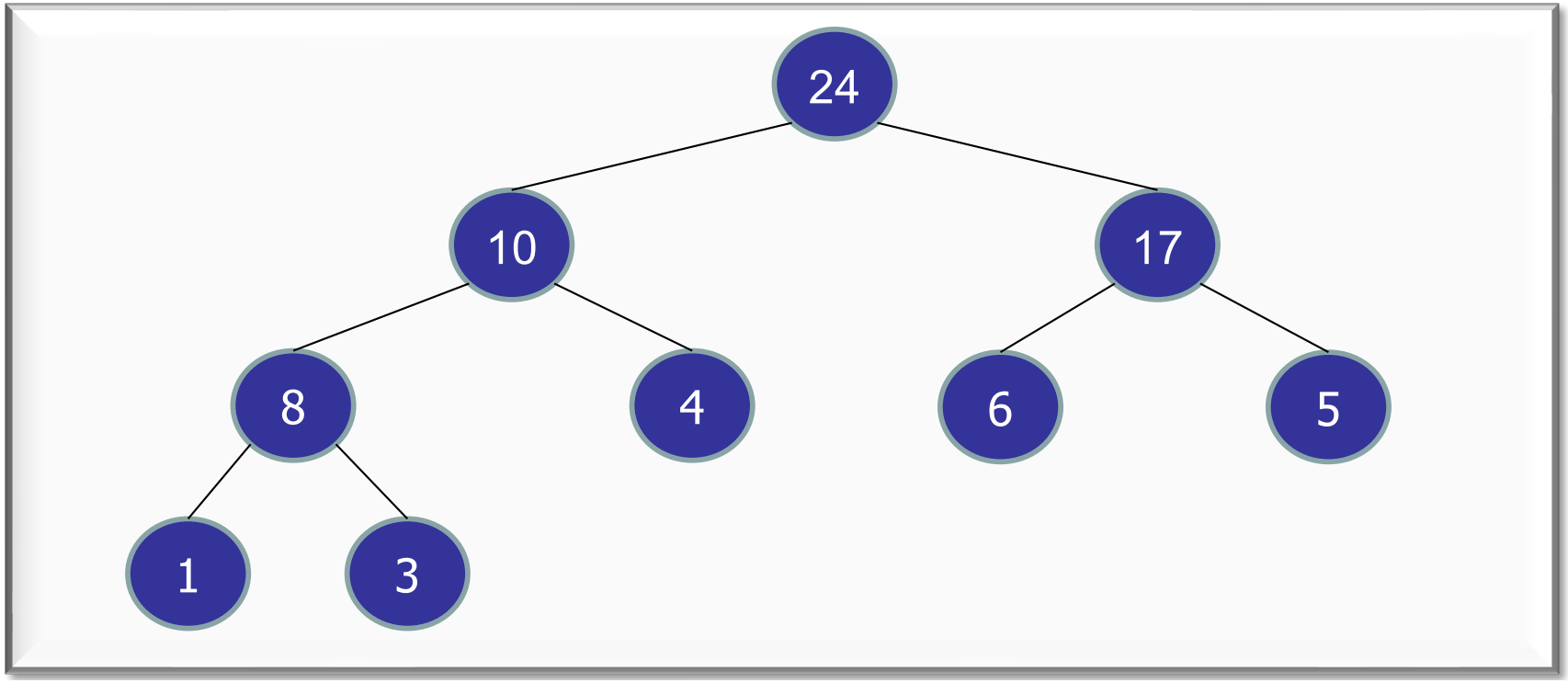
Step 2. Heap ➔ Sorted list:

array slot	0	1	2	3	4	5	6	7	8
key	1	3	4	5	6	8	10	17	24

HeapSort

Step 2. Heap ➔ Sorted list:

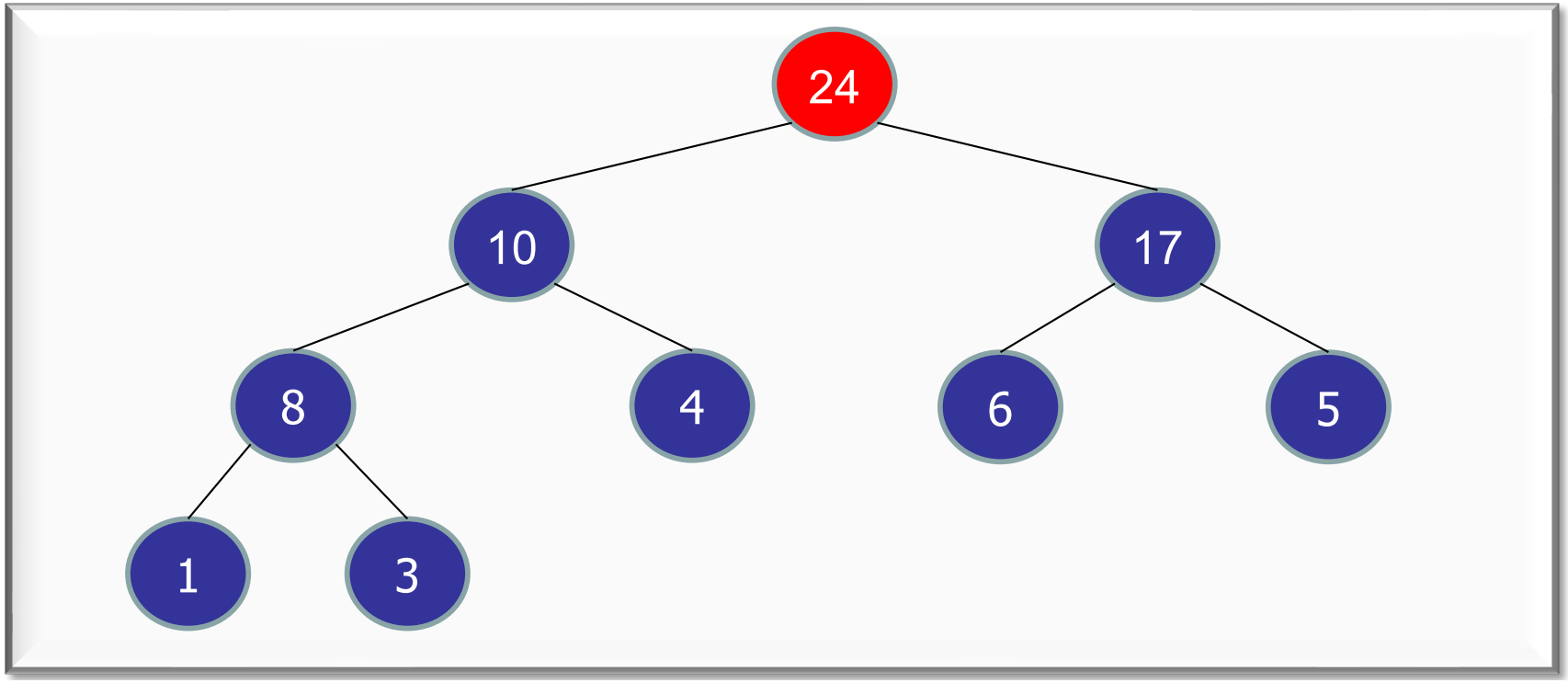
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3



HeapSort

```
value = extractMax();
```

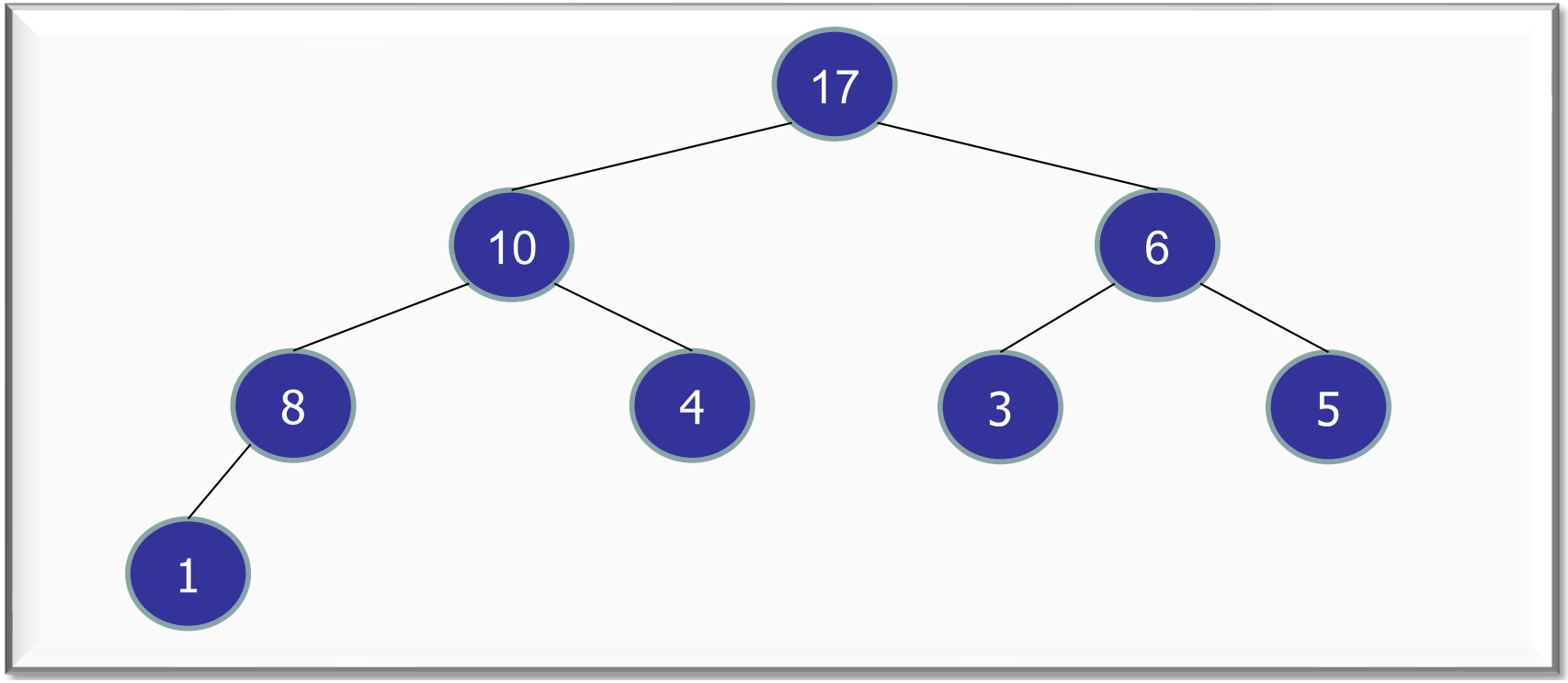
array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3



HeapSort

```
value = extractMax();
```

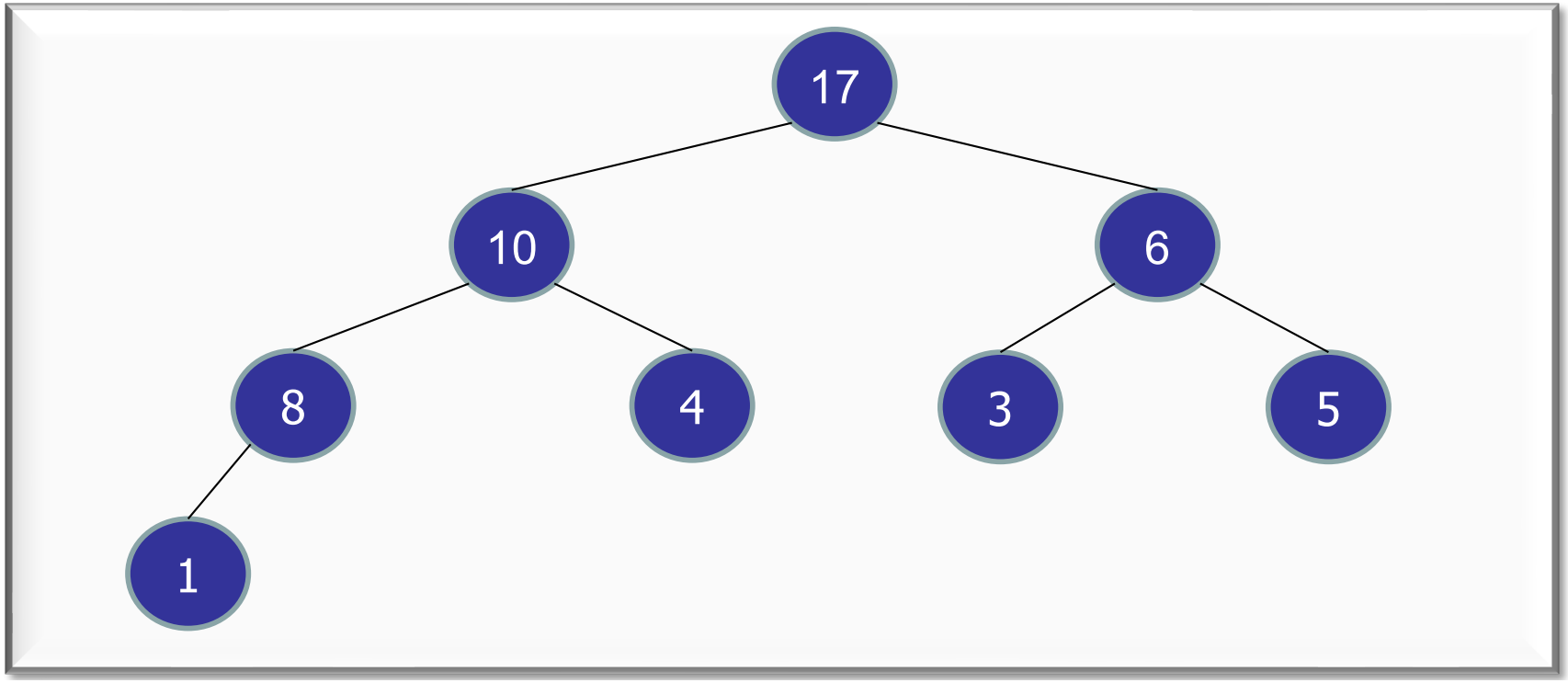
array slot	0	1	2	3	4	5	6	7	8
priority	17	10	6	8	4	3	5	1	



HeapSort

```
value = extractMax();
A[8] = value;
```

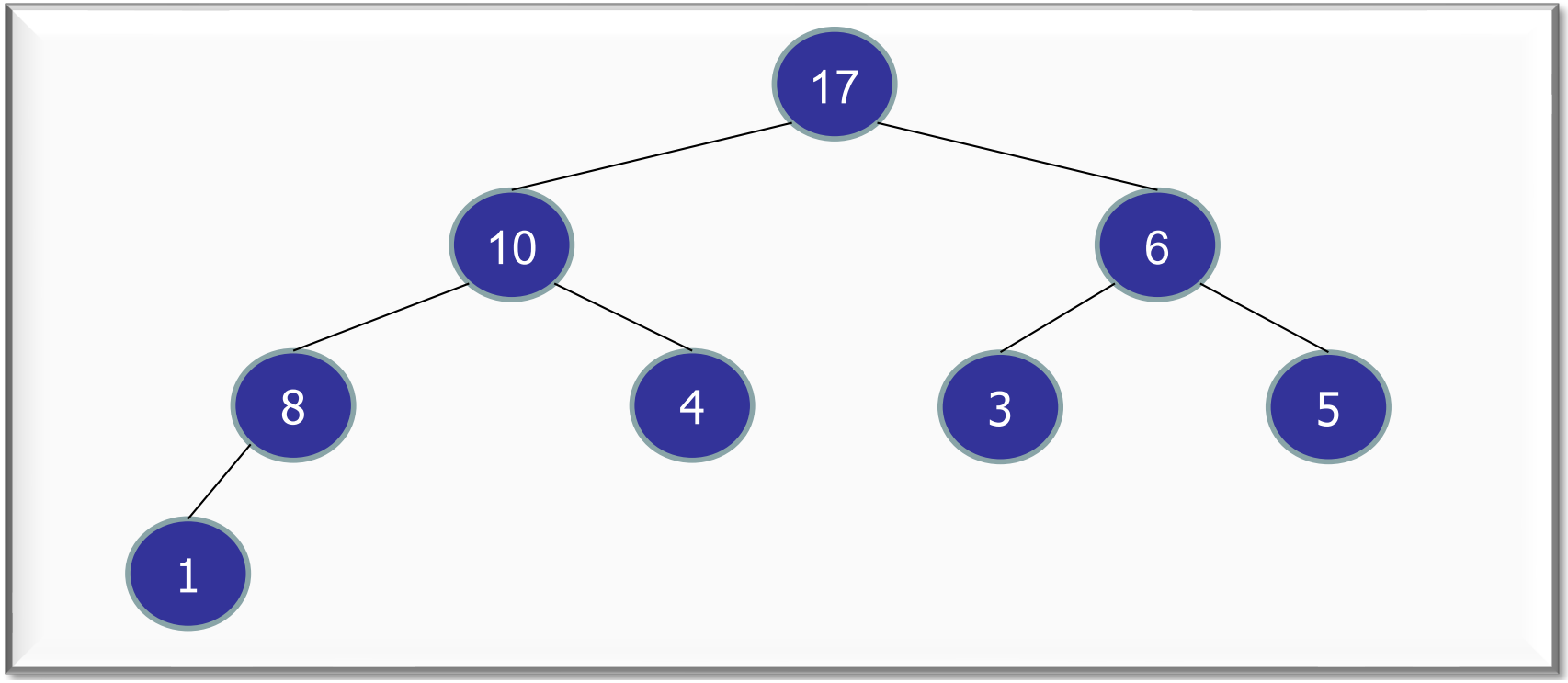
array slot	0	1	2	3	4	5	6	7	8
priority	17	10	6	8	4	3	5	1	24



HeapSort

```
value = extractMax();
```

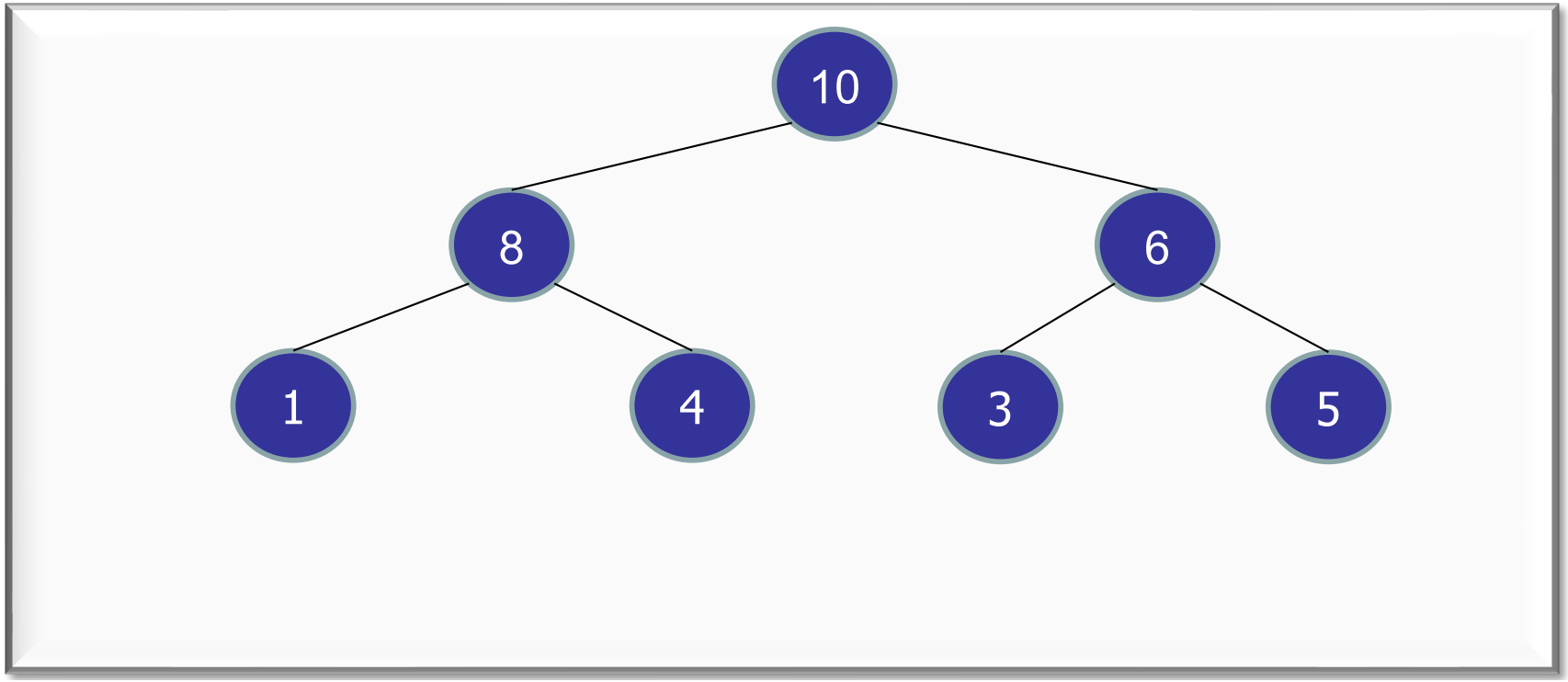
array slot	0	1	2	3	4	5	6	7	8
priority	17	10	6	8	4	3	5	1	24



HeapSort

```
value = extractMax();
A[7] = value;
```

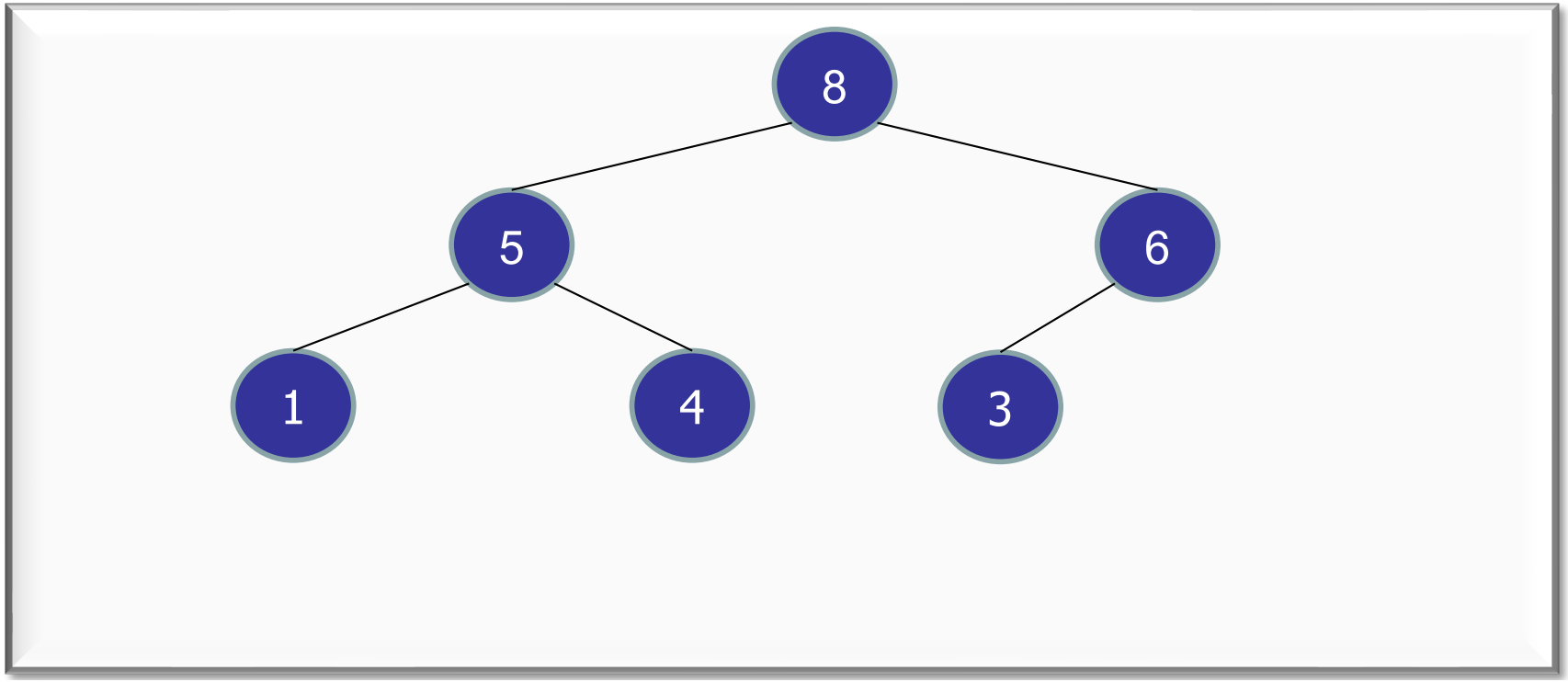
array slot	0	1	2	3	4	5	6	7	8
priority	10	8	6	1	4	3	5	17	24



HeapSort

```
value = extractMax();
A[6] = value;
```

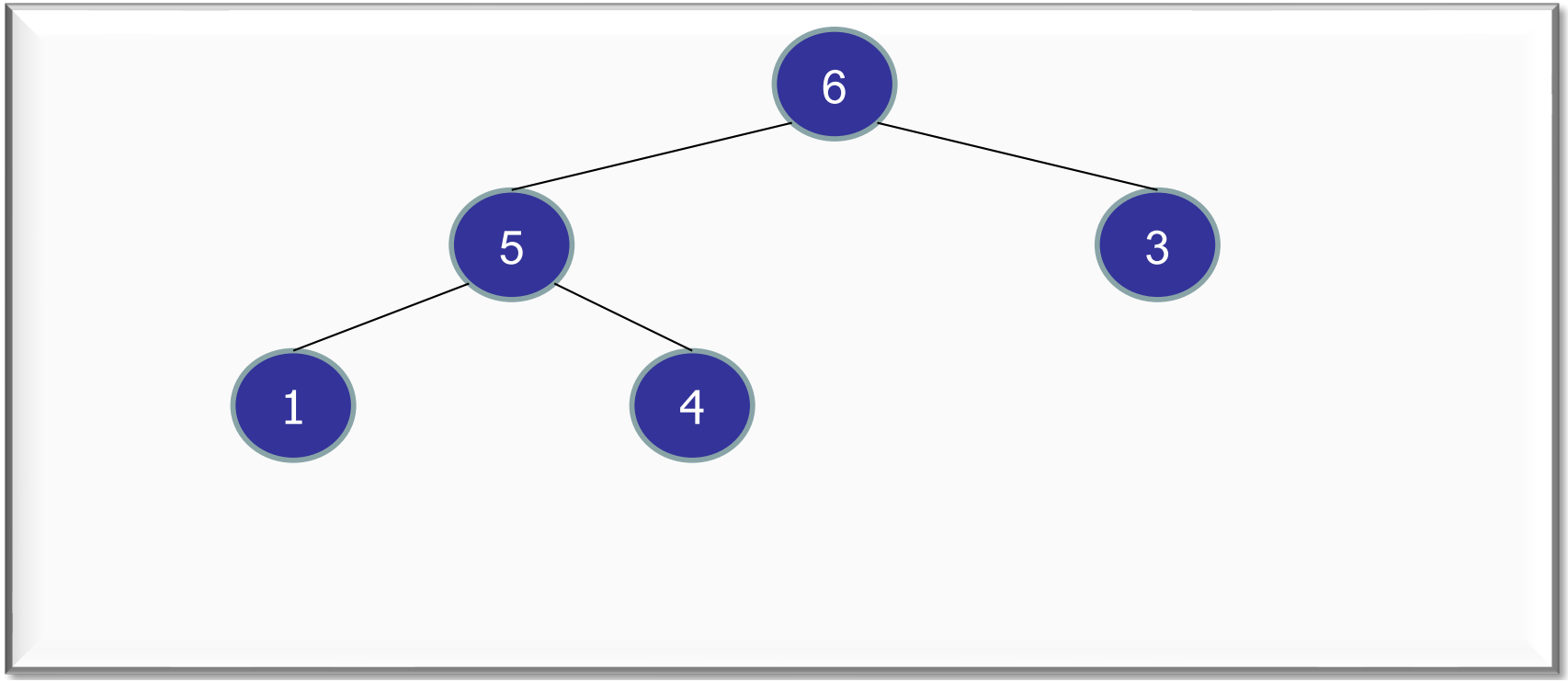
array slot	0	1	2	3	4	5	6	7	8
priority	8	5	6	1	4	3	10	17	24



HeapSort

```
value = extractMax();
A[5] = value;
```

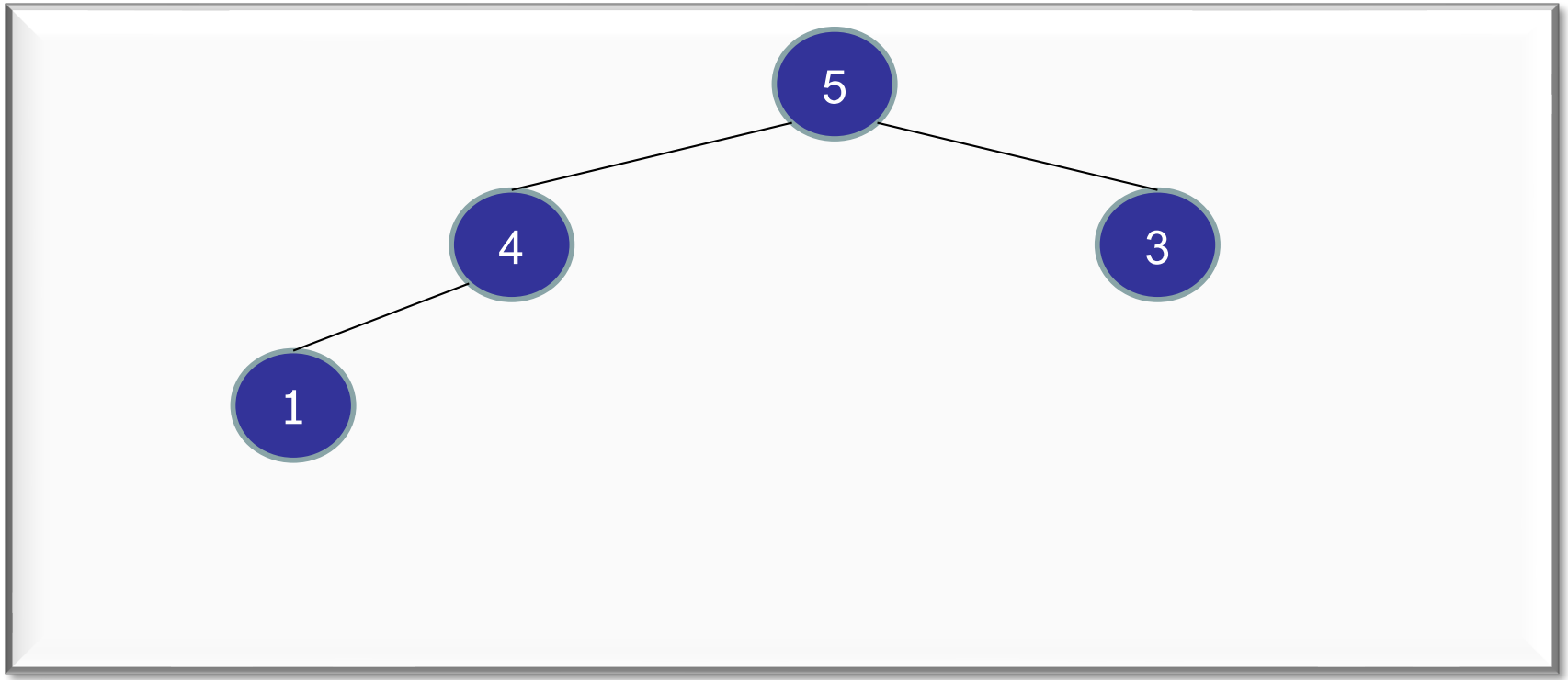
array slot	0	1	2	3	4	5	6	7	8
priority	6	5	3	1	4	8	10	17	24



HeapSort

```
value = extractMax();
A[4] = value;
```

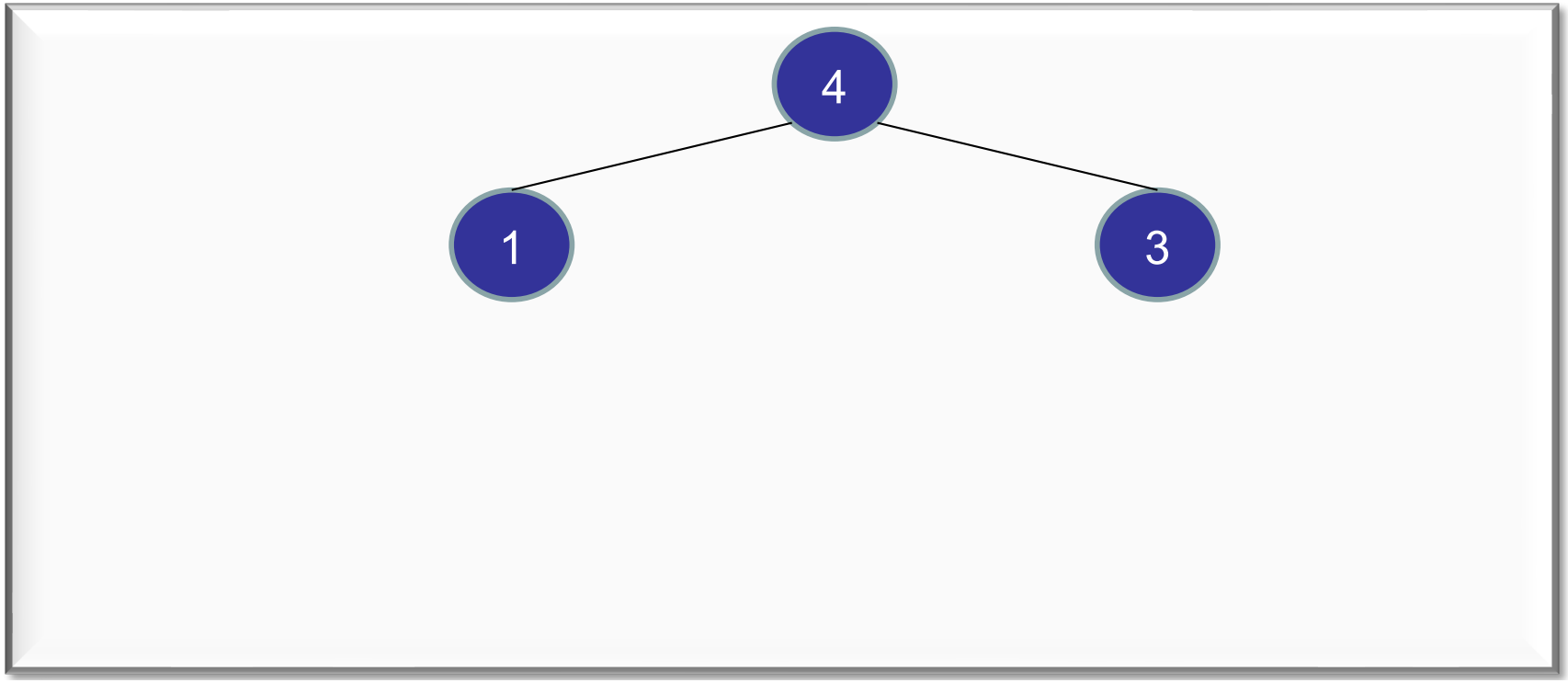
array slot	0	1	2	3	4	5	6	7	8
priority	5	4	3	1	6	8	10	17	24



HeapSort

```
value = extractMax();
A[3] = value;
```

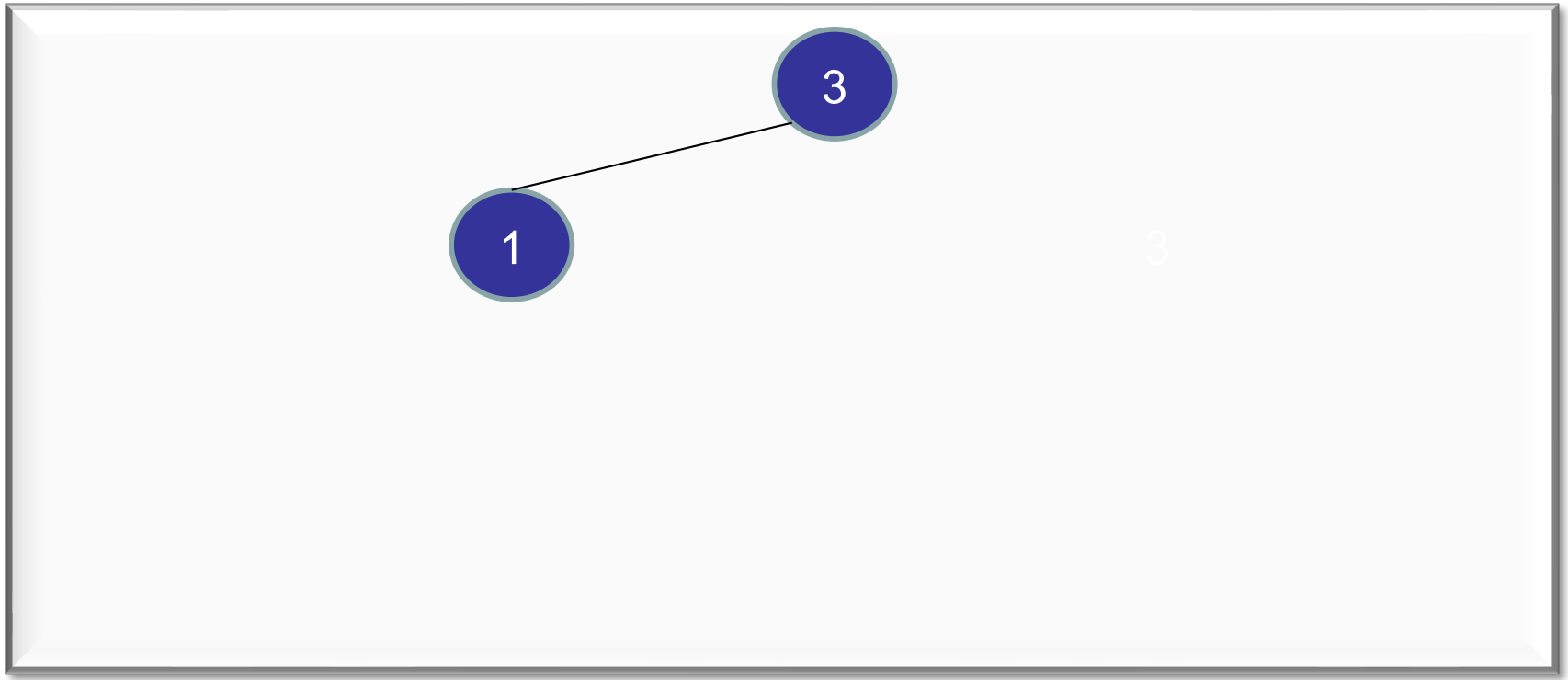
array slot	0	1	2	3	4	5	6	7	8
priority	4	1	3	5	6	8	10	17	24



HeapSort

```
value = extractMax();
A[2] = value;
```

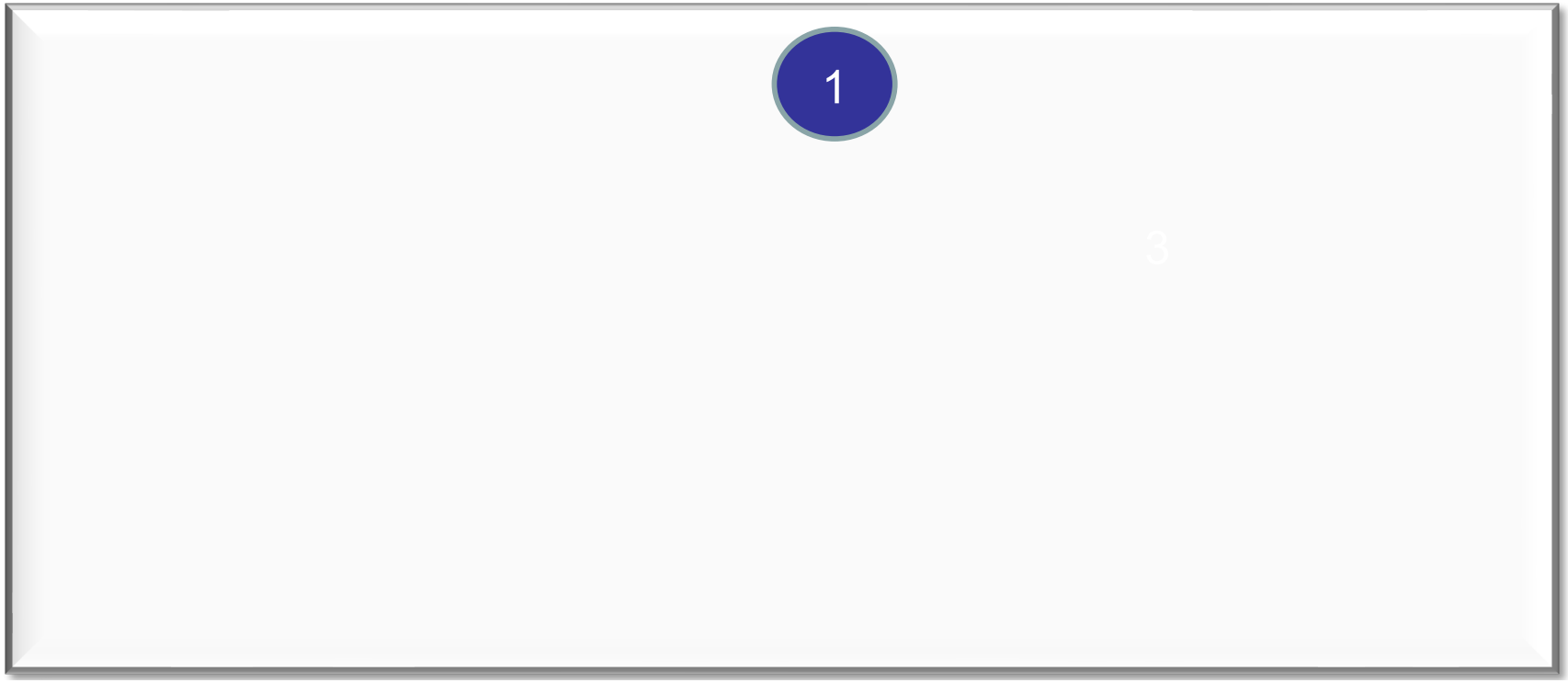
array slot	0	1	2	3	4	5	6	7	8
priority	3	1	4	5	6	8	10	17	24



HeapSort

```
value = extractMax();
A[1] = value;
```

array slot	0	1	2	3	4	5	6	7	8
priority	1	3	4	5	6	8	10	17	24



HeapSort

```
value = extractMax();
A[0] = value;
```

array slot	0	1	2	3	4	5	6	7	8
priority	1	3	4	5	6	8	10	17	24

HeapSort

Heap array ➔ Sorted list:

array slot	0	1	2	3	4	5	6	7	8
priority	1	3	4	5	6	8	10	17	24

```
// int[] A = array stored as a heap
for (int i=(n-1); i>=0; i--) {
        int value = extractMax(A);
        A[i] = value;
}
```

What is the running time for converting a heap into a sorted array?

1. O(log n)
2. O(n)
3. ✓ O(n log n)
4. $O(n^2)$
5. I have no idea.

HeapSort

Heap array ➔ Sorted list: O(n log n)

array slot	0	1	2	3	4	5	6	7	8
priority	1	3	4	5	6	8	10	17	24

```
// int[] A = array stored as a heap
for (int i=(n-1); i>=0; i--) {
      int value = extractMax(A); // O(log n)
      A[i] = value;
}
```

HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

Step 1. Unsorted list ➔ Heap

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3

HeapSort

Heapify: Unsorted list ➔ Heap:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

```
// int[] A = array of unsorted integers
for (int i=0; i<n; i++) {
        int value = A[i];
        A[i] = EMPTY:
        heapInsert(value, A, 0, i);
}
```

HeapSort

Heapify: Unsorted list ➔ Heap: O(n log n)

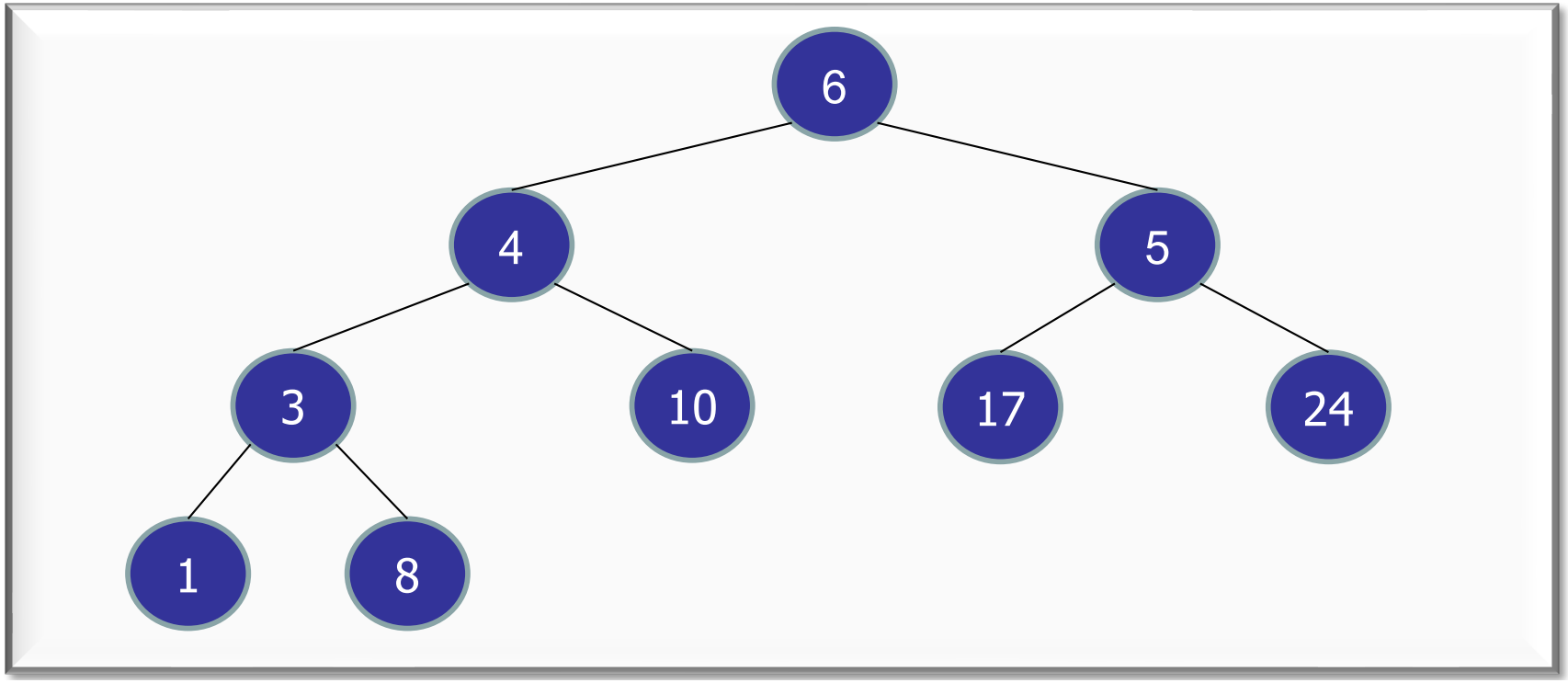
array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

```
// int[] A = array of unsorted integers
for (int i=0; i<n; i++) {
        int value = A[i];
        A[i] = EMPTY:
        heapInsert(value, A, 0, i); // O(log n)
}
```

HeapSort

Heapify v.2: Unsorted list ➔ Heap

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

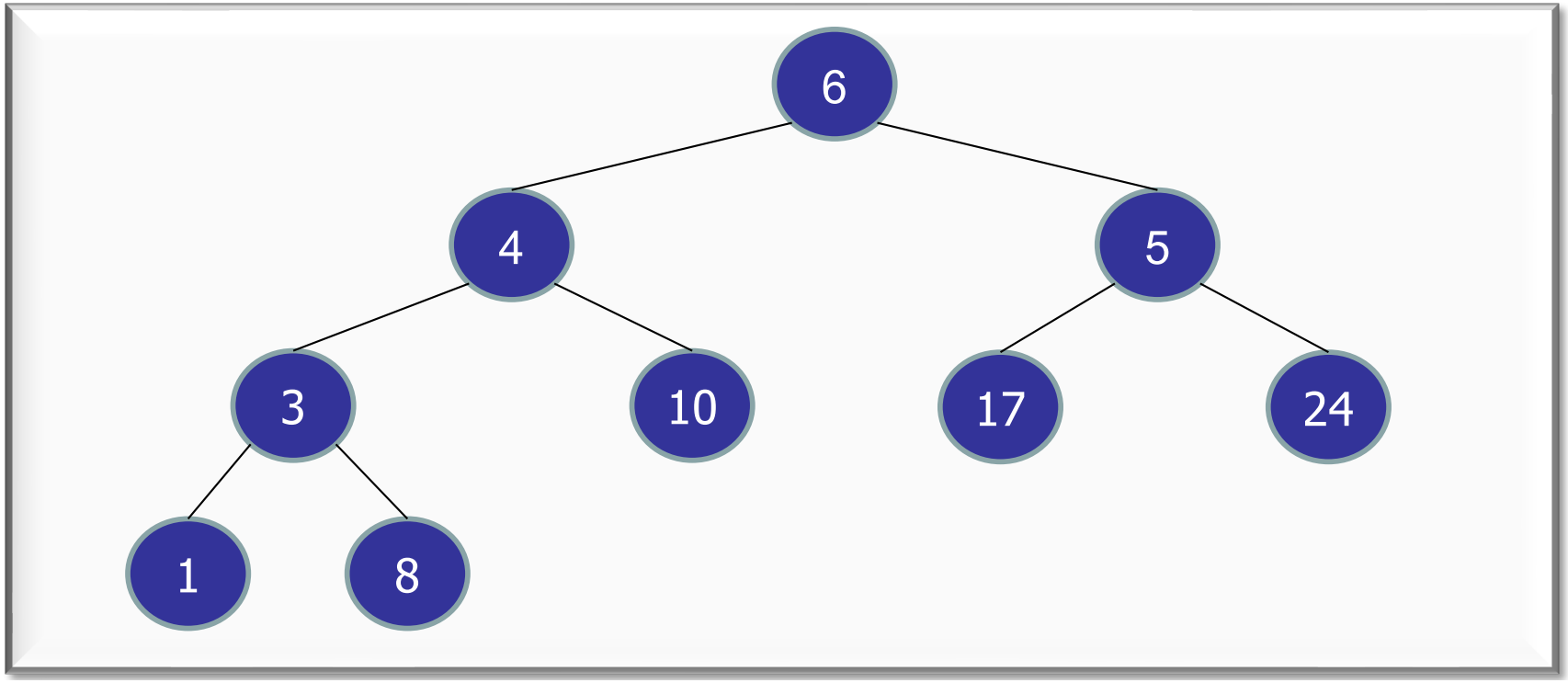


HeapSort

Idea: Recursion

Initially : Start with a complete tree.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

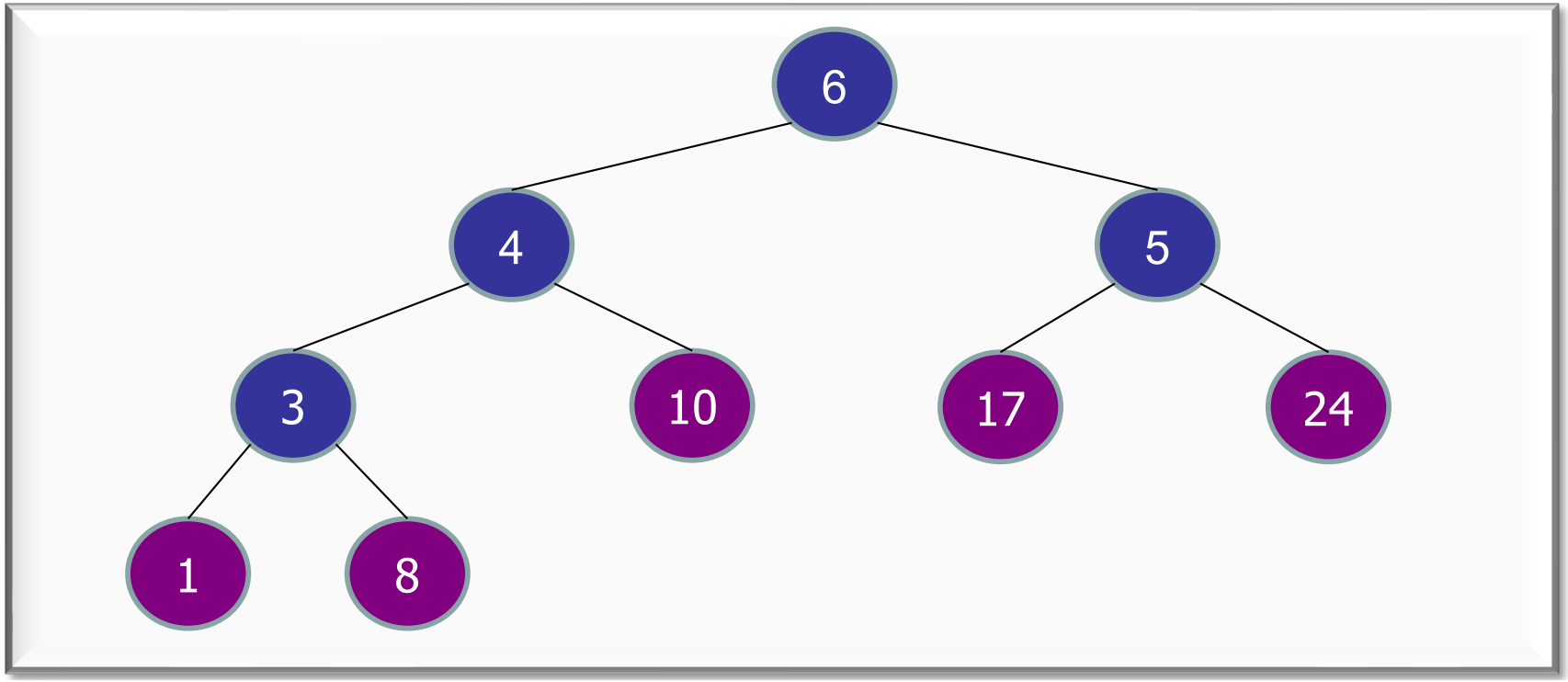


HeapSort

Idea: Recursion

Base case: each leaf is a heap.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

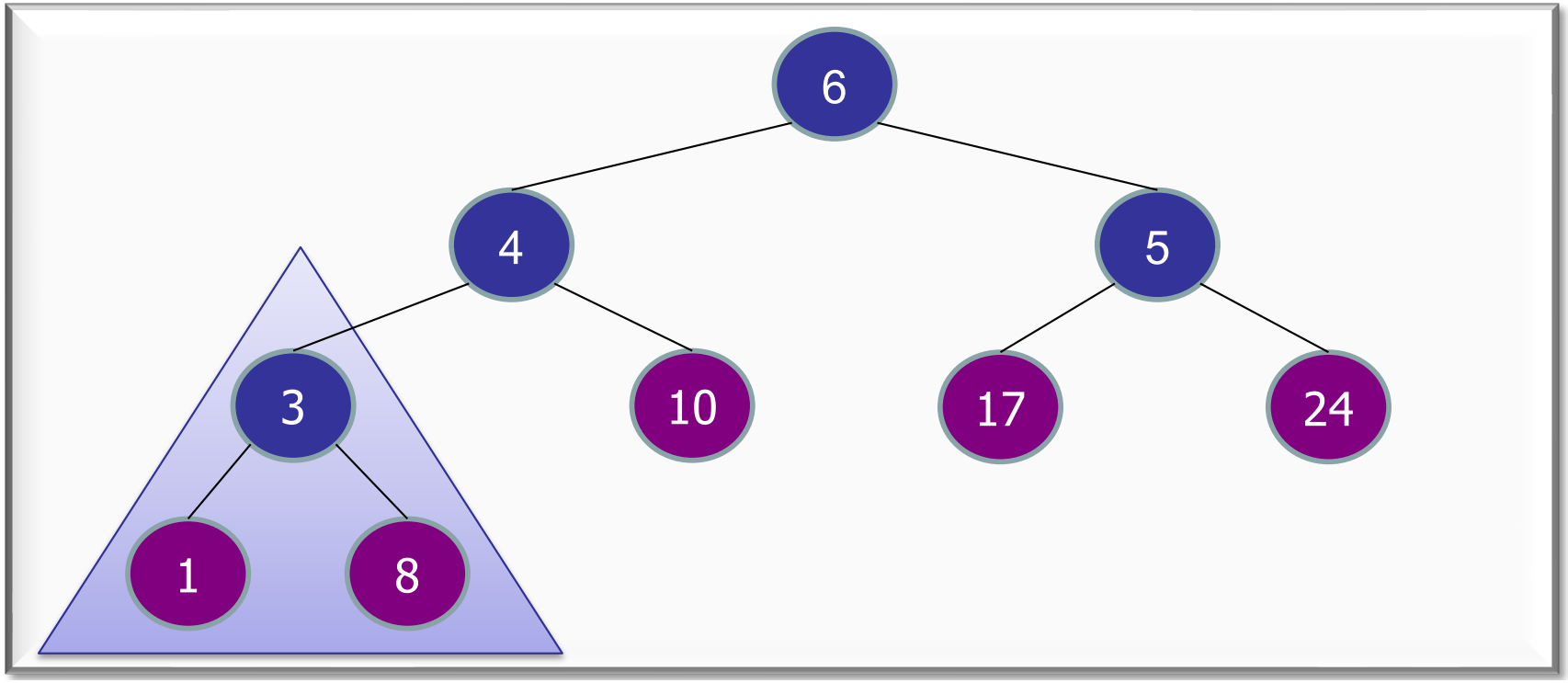


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

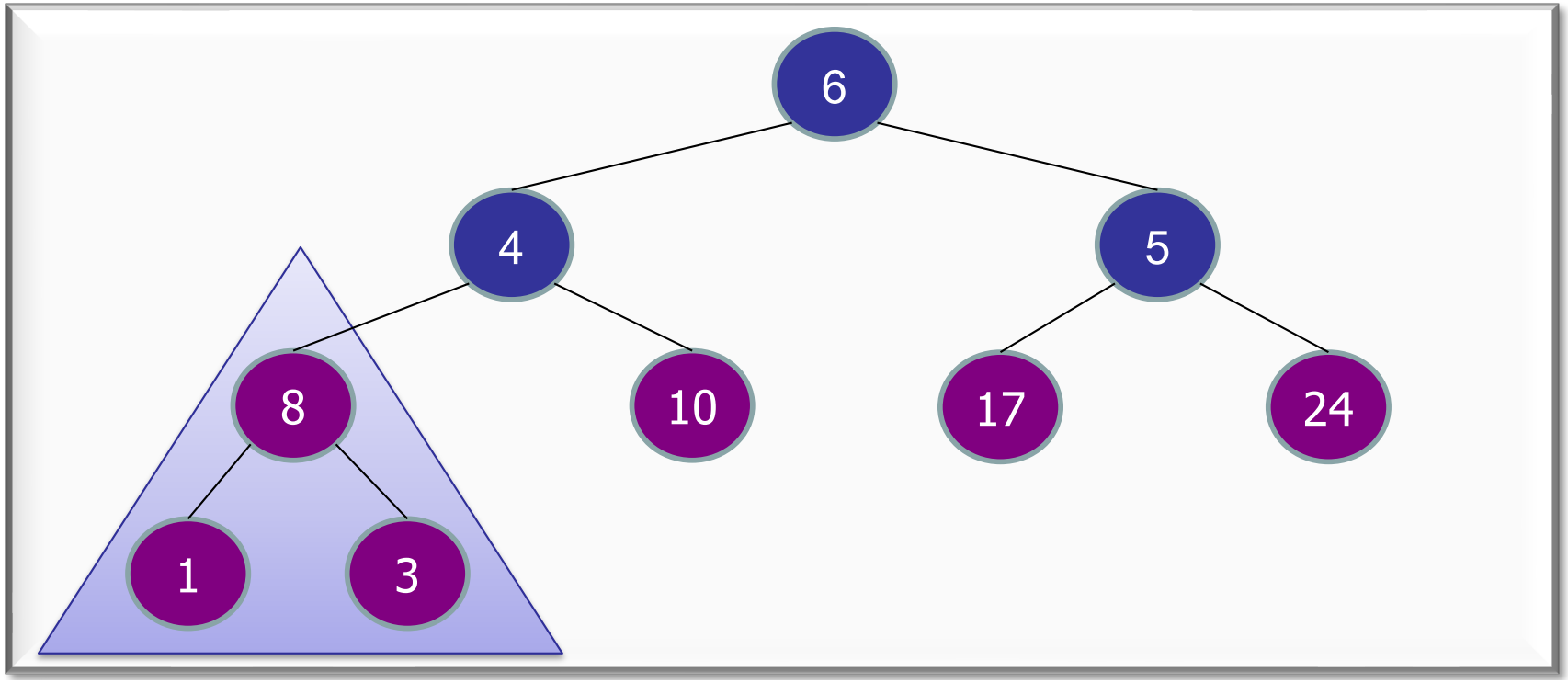


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	8	10	17	24	1	3

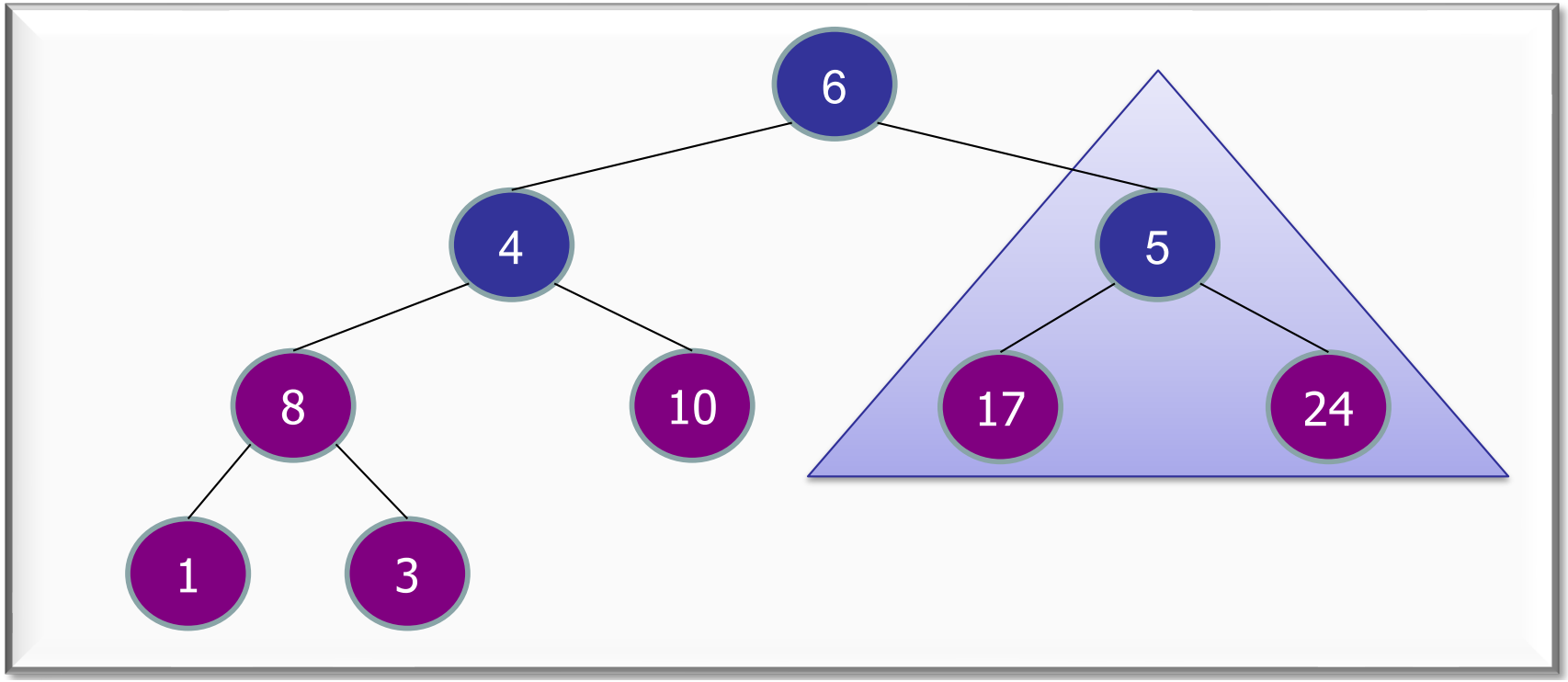


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	8	10	17	24	1	3

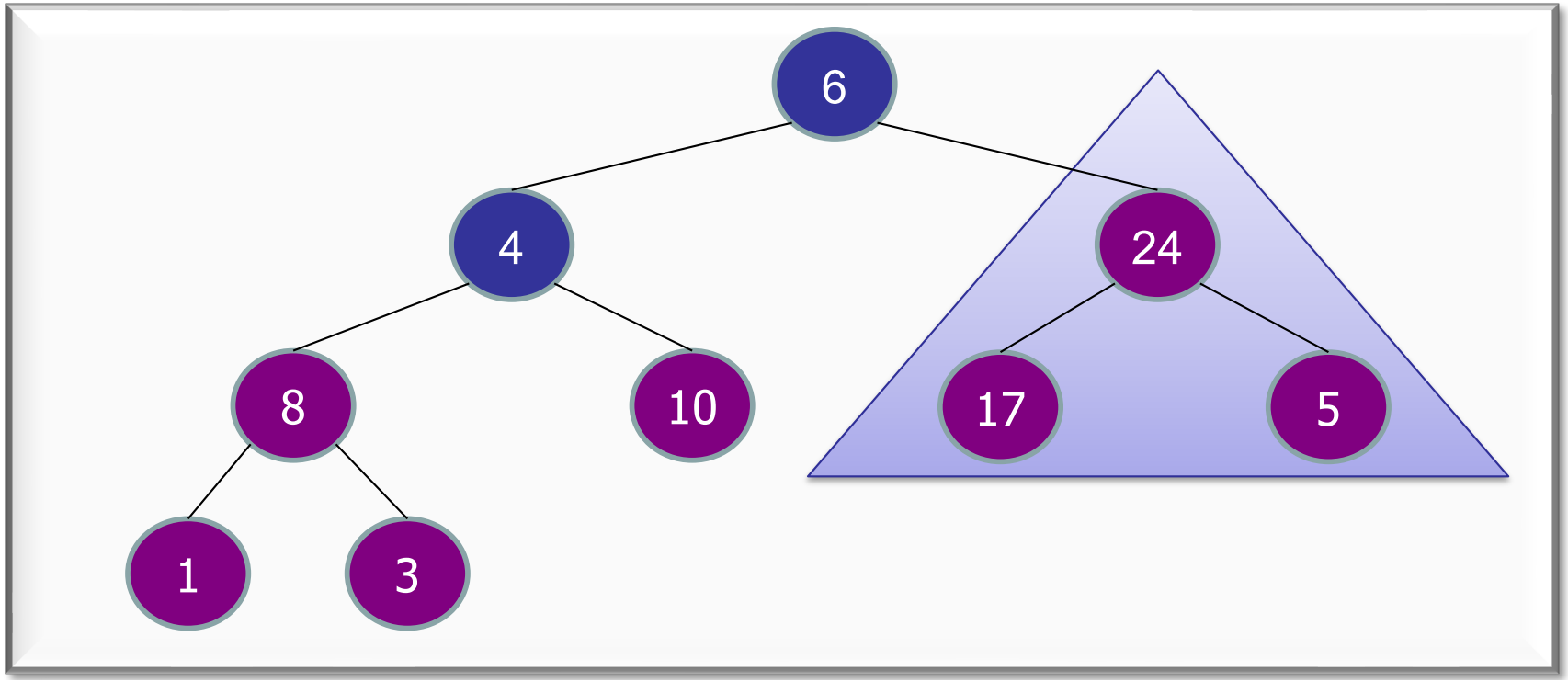


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3

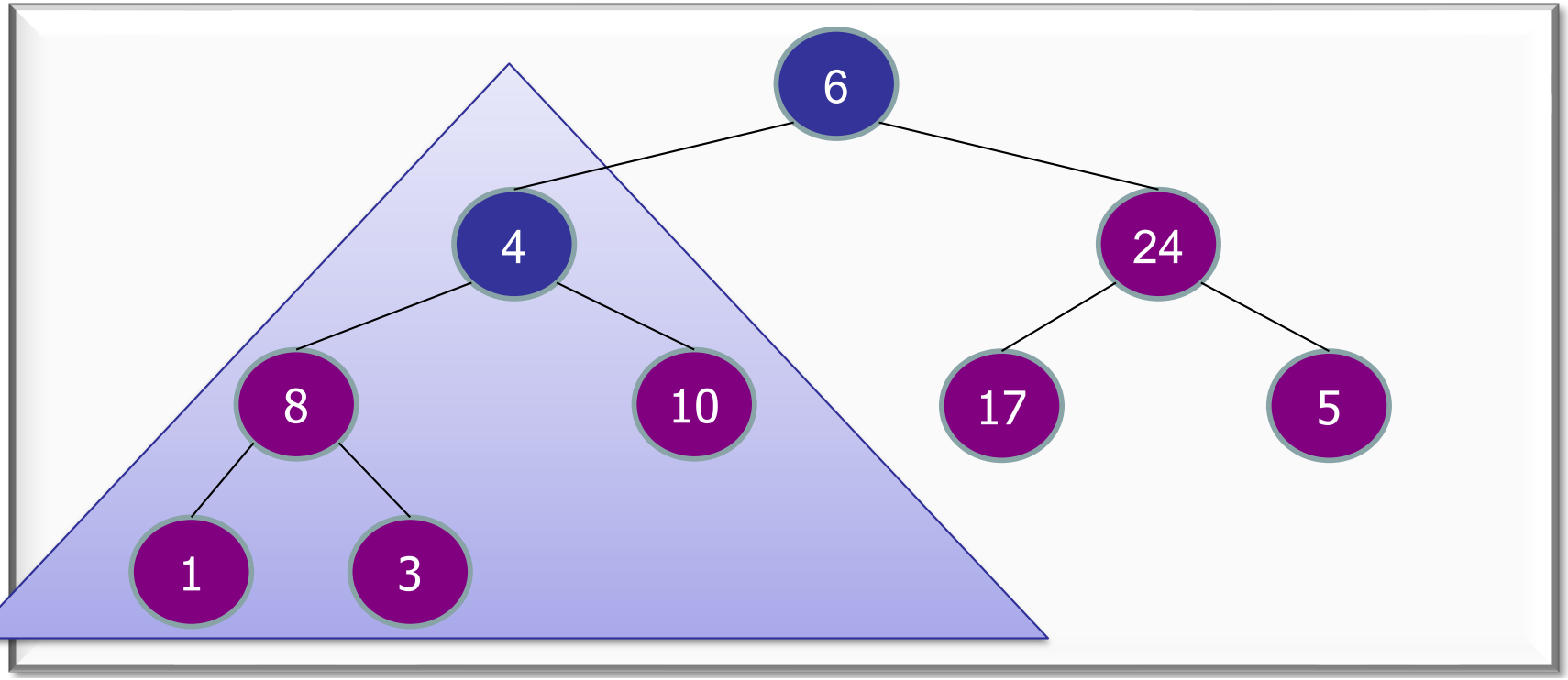


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3

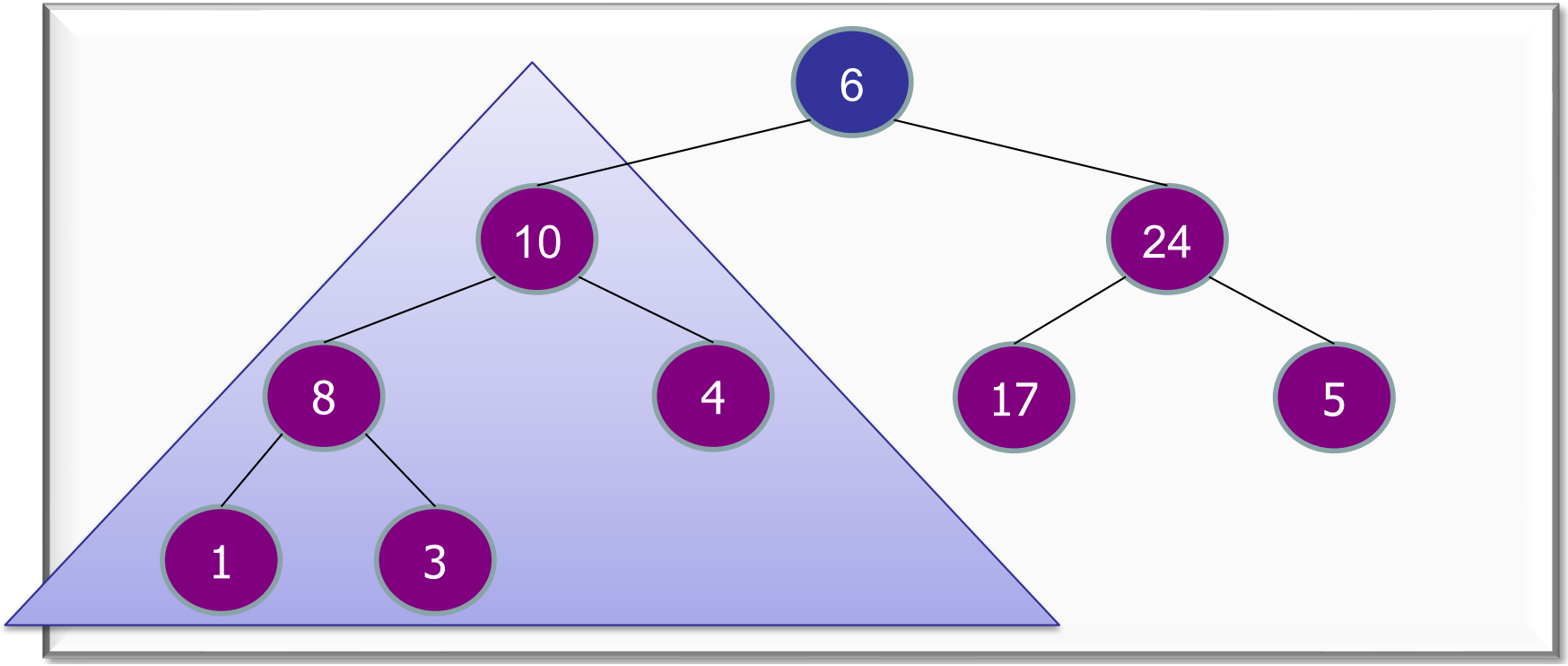


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	10	24	8	4	17	5	1	3

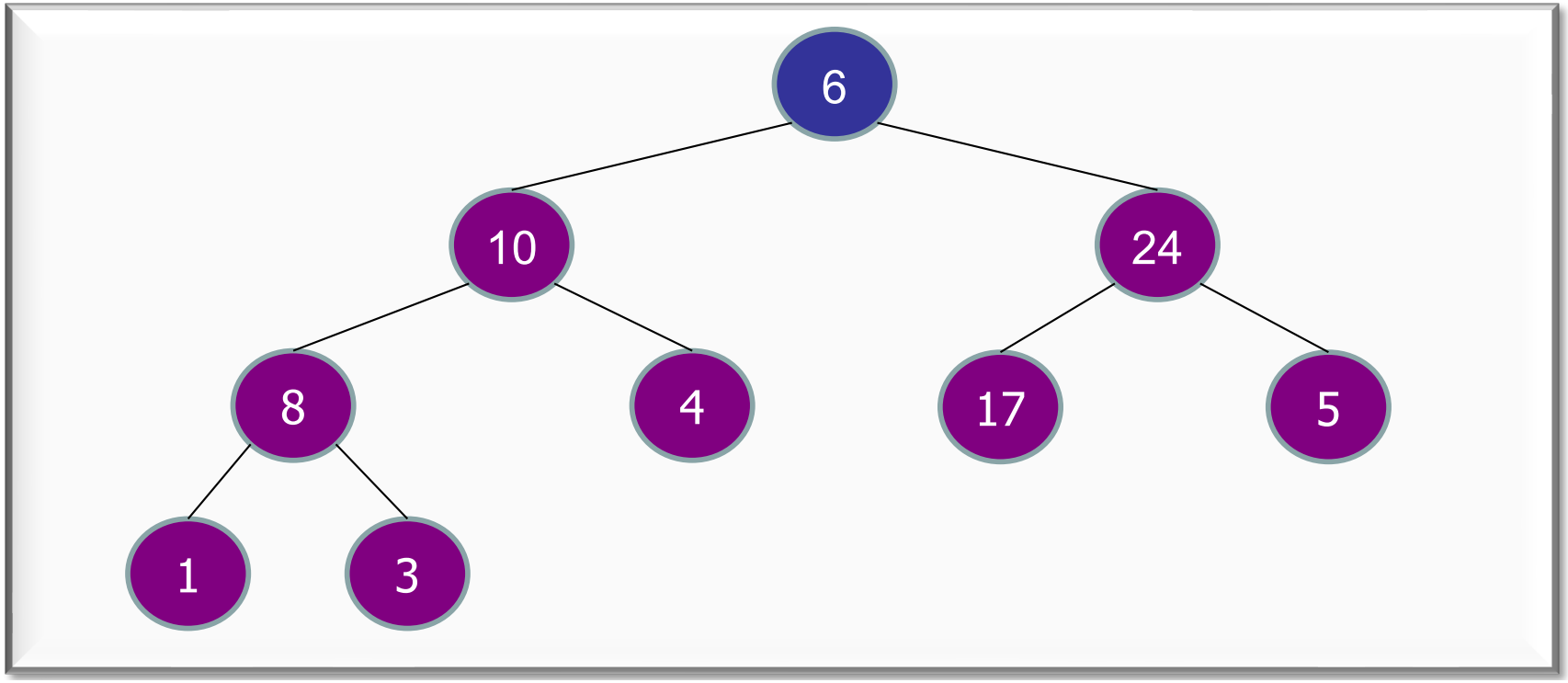


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	6	10	24	8	4	17	5	1	3

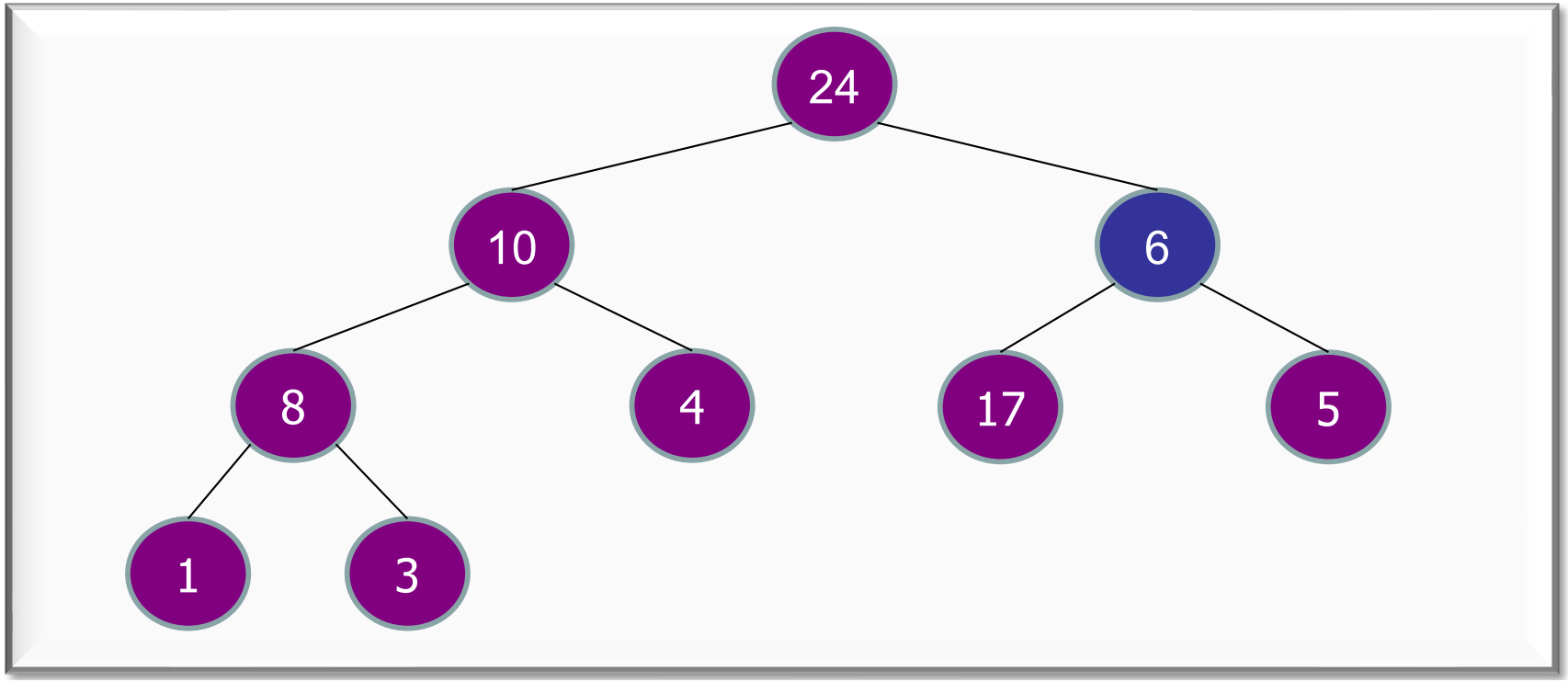


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	24	10	6	8	4	17	5	1	3

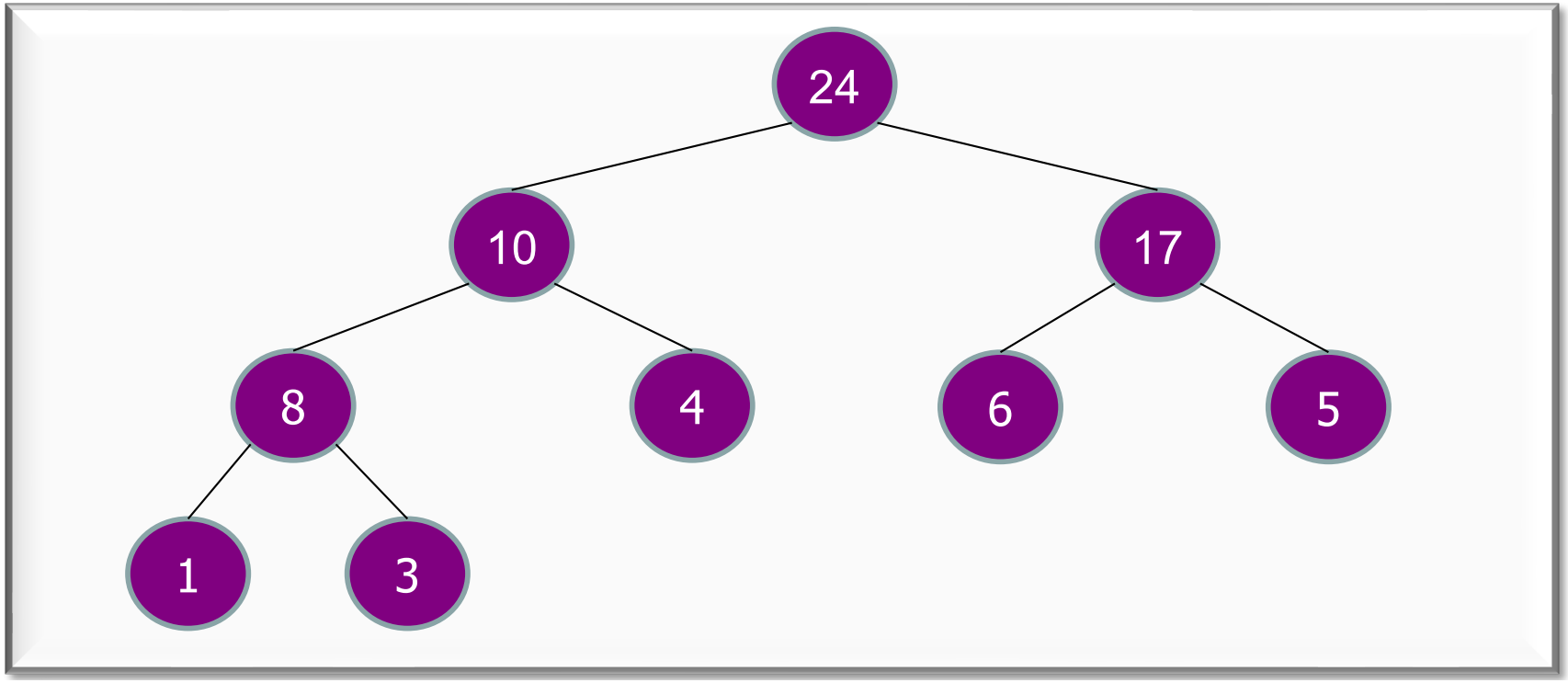


HeapSort

Idea: Recursion

Recursion: left + right are heaps.

array slot	0	1	2	3	4	5	6	7	8
key	24	10	17	8	4	6	5	1	3



HeapSort

Heapify v.2: Unsorted list ➔ Heap

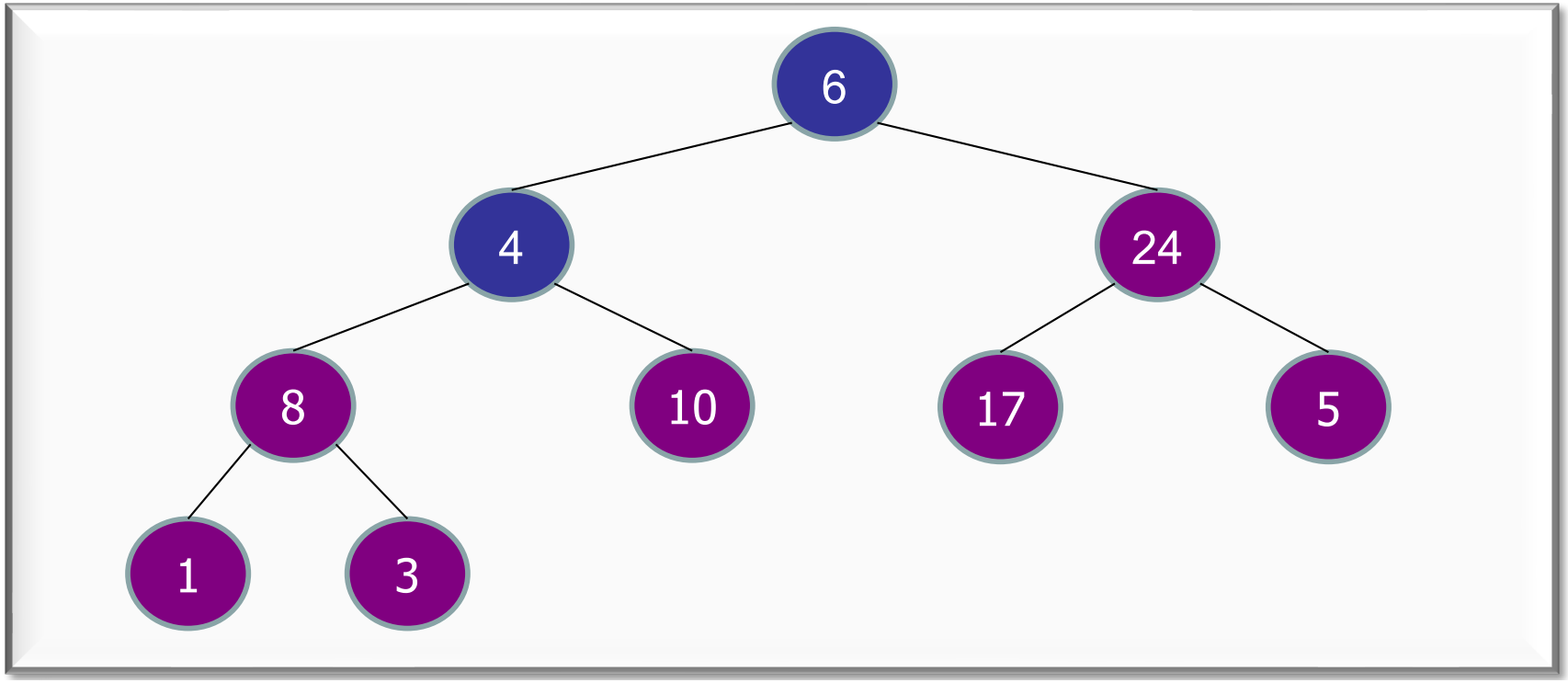
array slot	0	1	2	3	4	5	6	7	8
key	24	10	17	8	4	6	5	1	3

```
// int[] A = array of unsorted integers
for (int i=(n-1); i>=0; i--) {
      bubbleDown(i, A); // O(log n)
}
```

HeapSort

Observation: cost(bubbleDown) = height

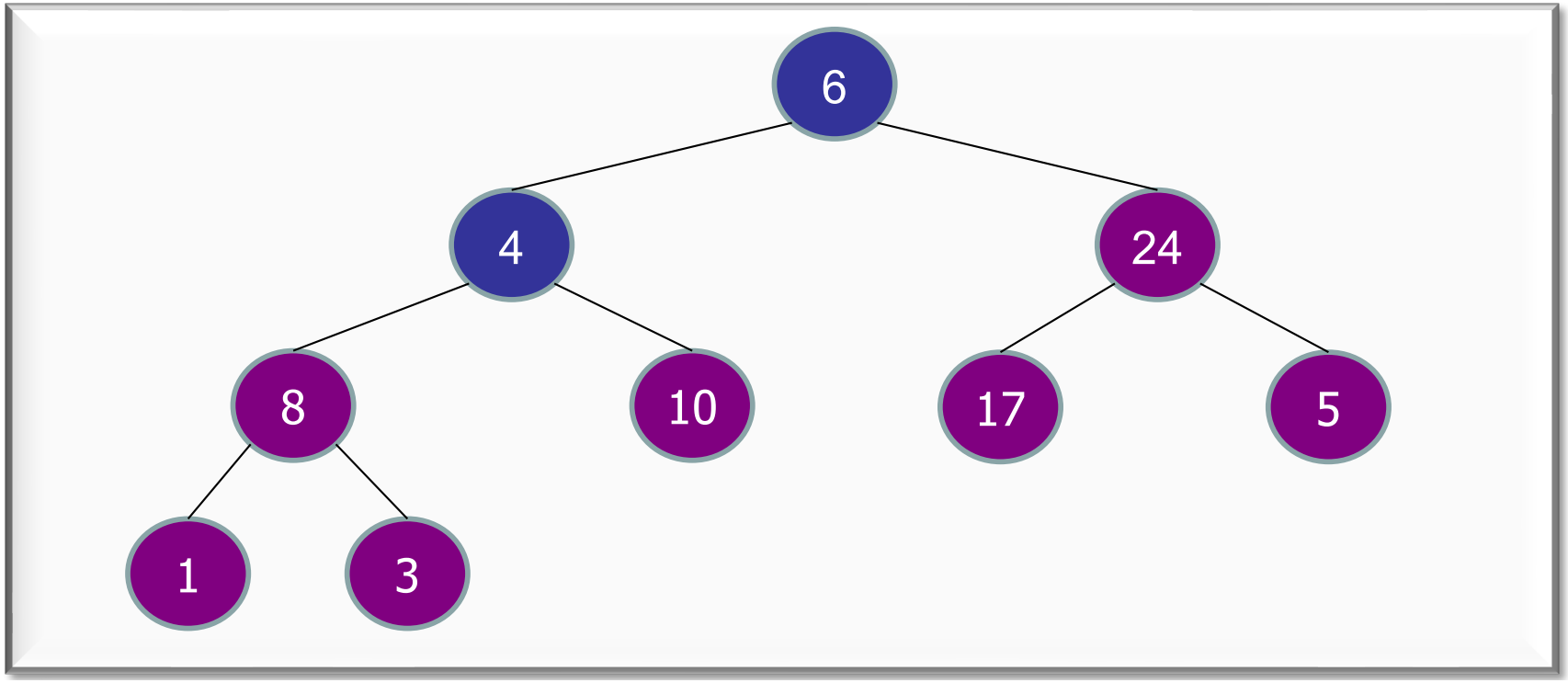
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

Observation: > n/2 nodes are leaves (height=0)

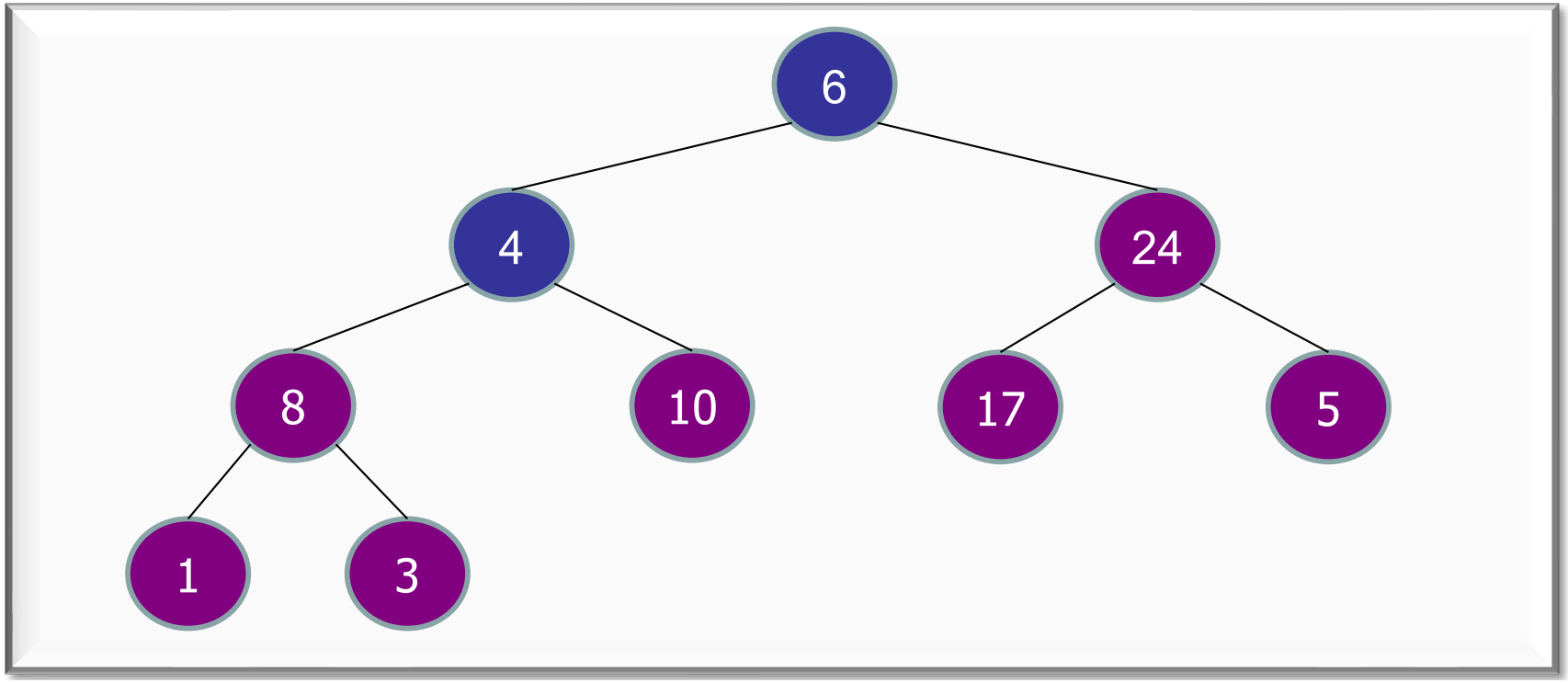
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

Observation: most nodes have small height!

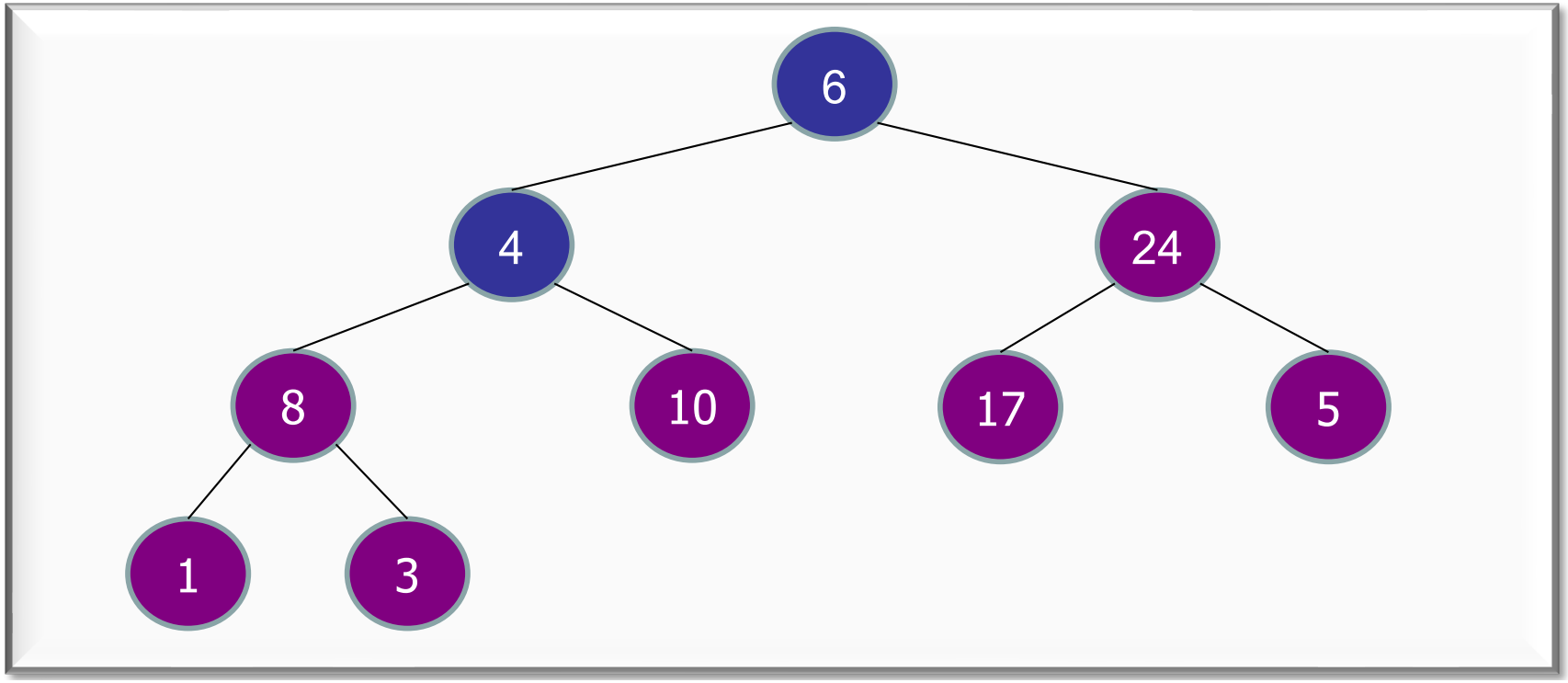
array slot	0	1	2	3	4	5	6	7	8
key	6	4	24	8	10	17	5	1	3



HeapSort

Cost of building a heap:

Height	0	1	2	3	...	⌊ log(n) ⌋
Number	⌈n/2⌉	⌈n/4⌉	⌈n/8⌉	⌈n/16⌉	...	1



HeapSort

Cost of building a heap:

Height	0	1	2	3	...	⌊ log(n) ⌋
Number	⌈n/2⌉	⌈n/4⌉	⌈n/8⌉	⌈n/16⌉	...	1

$$\sum_{h=0}^{h=\log(n)} \frac{n}{2^h} O(h)$$

cost for bubbling a node at level h

upper bound on number of nodes at level h

HeapSort

Cost of building a heap:

Height	0	1	2	3	...	⌊ log(n) ⌋
Number	⌈n/2⌉	⌈n/4⌉	⌈n/8⌉	⌈n/16⌉	...	1

$$\sum_{h=0}^{h=\log(n)} \frac{n}{2^h} O(h) \le cn\left(\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \ldots\right)$$

$$\le cn\left(\frac{1/2}{(1-1/2)^2}\right) \le 2 \cdot O(n)$$

HeapSort

Heapify v.2: Unsorted list ➔ Heap: O(n)

array slot	0	1	2	3	4	5	6	7	8
key	24	10	17	8	4	6	5	1	3

```
// int[] A = array of unsorted integers
for (int i=(n-1); i>=0; i--) {
      bubbleDown(i, A); // O(height)
}
```

HeapSort

Unsorted list:

array slot	0	1	2	3	4	5	6	7	8
key	6	4	5	3	10	17	24	1	8

Step 1. Unsorted list ➔ Heap: O(n)

array slot	0	1	2	3	4	5	6	7	8
priority	24	10	17	8	4	6	5	1	3

Step 2. Heap array ➔ Sorted list: O(n log n)

array slot	0	1	2	3	4	5	6	7	8
key	1	3	4	5	6	8	10	17	24

HeapSort

Summary

- O(n log n) time *worst-case*
- In-place: only need n space!
- Fast:
  - Faster than MergeSort
  - A little slower than QuickSort.
- Deterministic: always completes in O(n log n)
- Unstable (*Come up with an example*!)
- Ternary (3-way) HeapSort is a little faster.