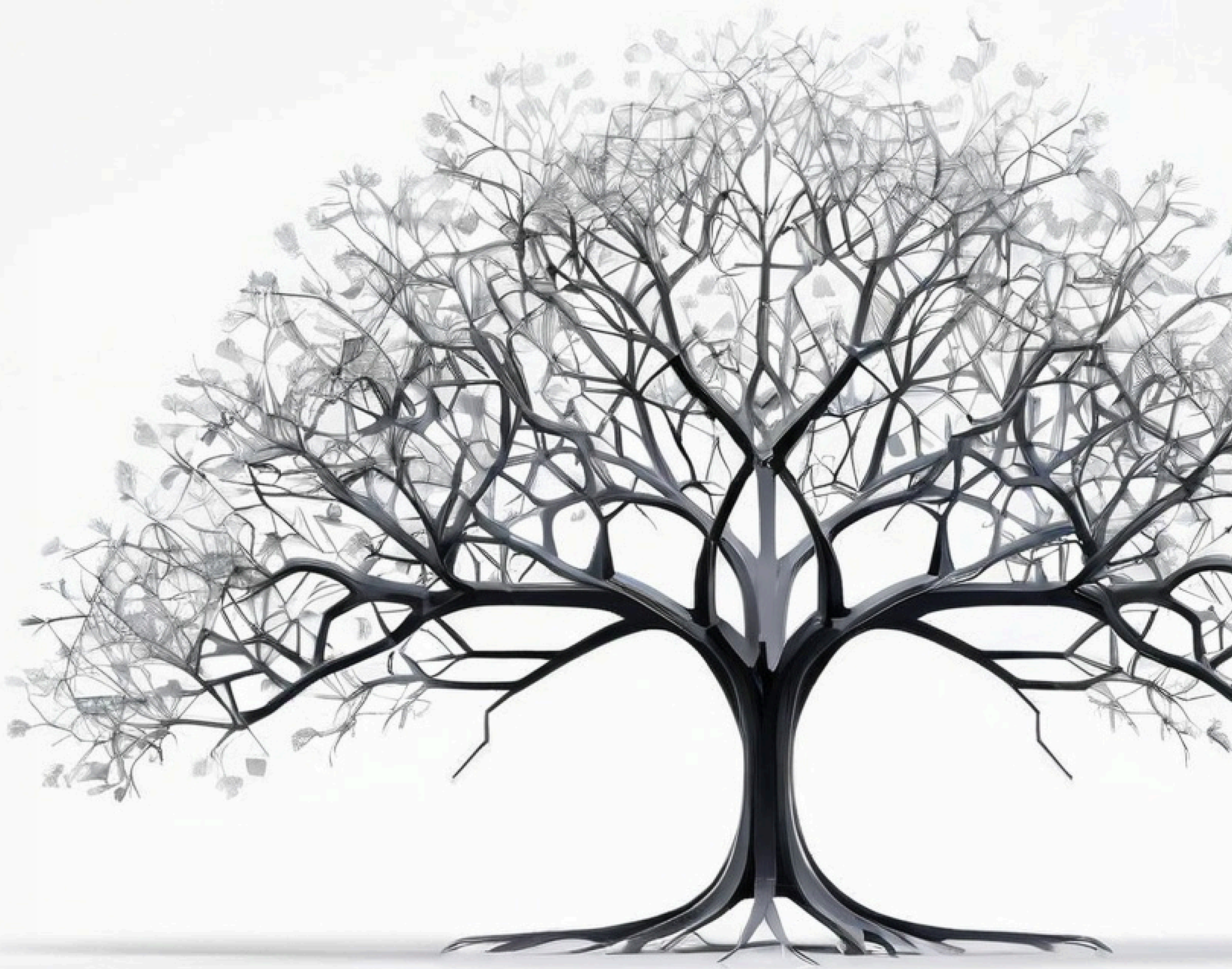
# Segment Tree



# Lecture Flow

- Motivation Problem
- Definition
- Construction
- Operations
- Implementation
- Variants
- Applications
- Alternatives
- Practice Problems
- Resources

# Part I

# Motivation Problem

We have an array **arr[0 . . . n-1]**. We should be able to

- Find the sum of elements from index **l** to **r** where **0 <= l <= r <= n-1**.
- Change the value of a specified element of the array to a new value **x**. We need to do **arr[i] = x** where **0 <= i <= n-1**.

# Definition

A Segment Tree is a data structure that stores information about array **intervals** as a tree. This allows answering range **queries** over an array efficiently, while still being flexible enough to allow quick **modification** of the array.

# Construction

We start at the bottom level (leaf nodes), and construct the tree upwards.

- A node corresponds to a range in the original array.
- A **leaf node** corresponds to the single element range, **a[i]**.

# Construction

- To form a higher level, from the leaf nodes, we will start merging two consecutive nodes at a time, until all the lower level nodes are covered.
- The two nodes corresponding to the ranges **a[l1...r1]** and **a[l2...r2]** would be merged into a node corresponding to the range **a[l1...r2]**.

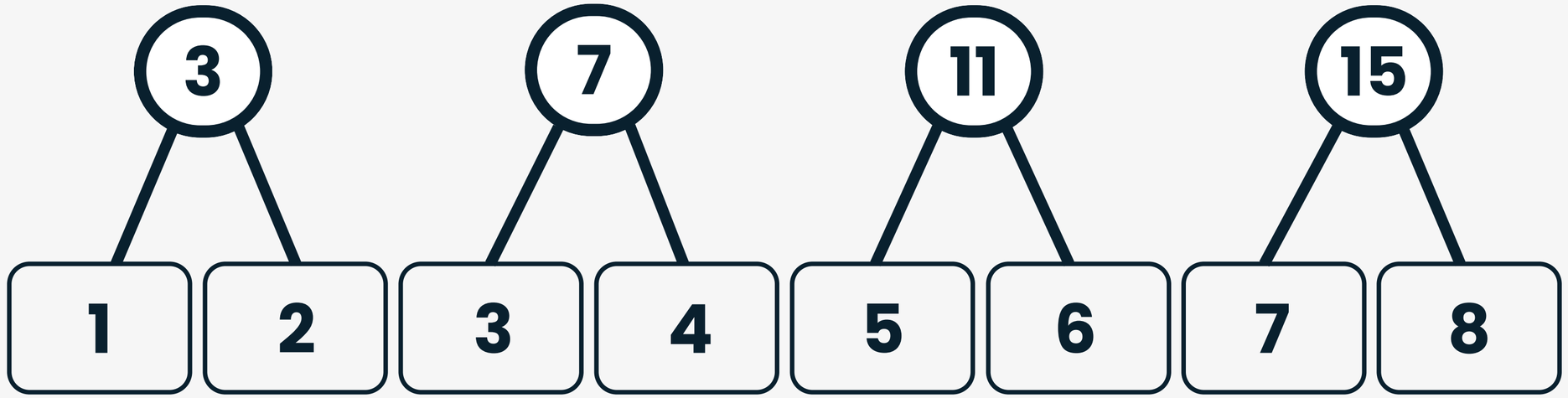
# Construction

- We repeat the previous procedure until we only have a single node (**the root**) that corresponds to the range **a[0...n-1]**.

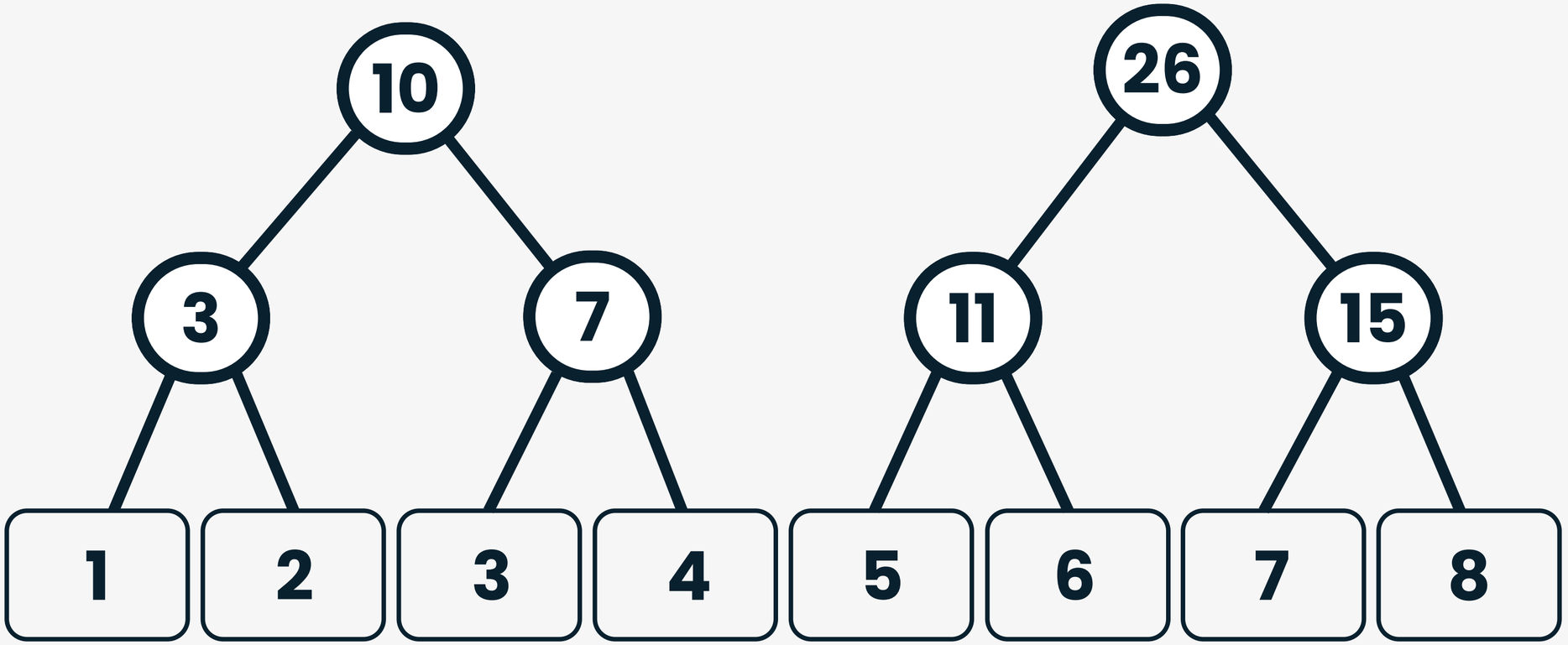
# Construction

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|

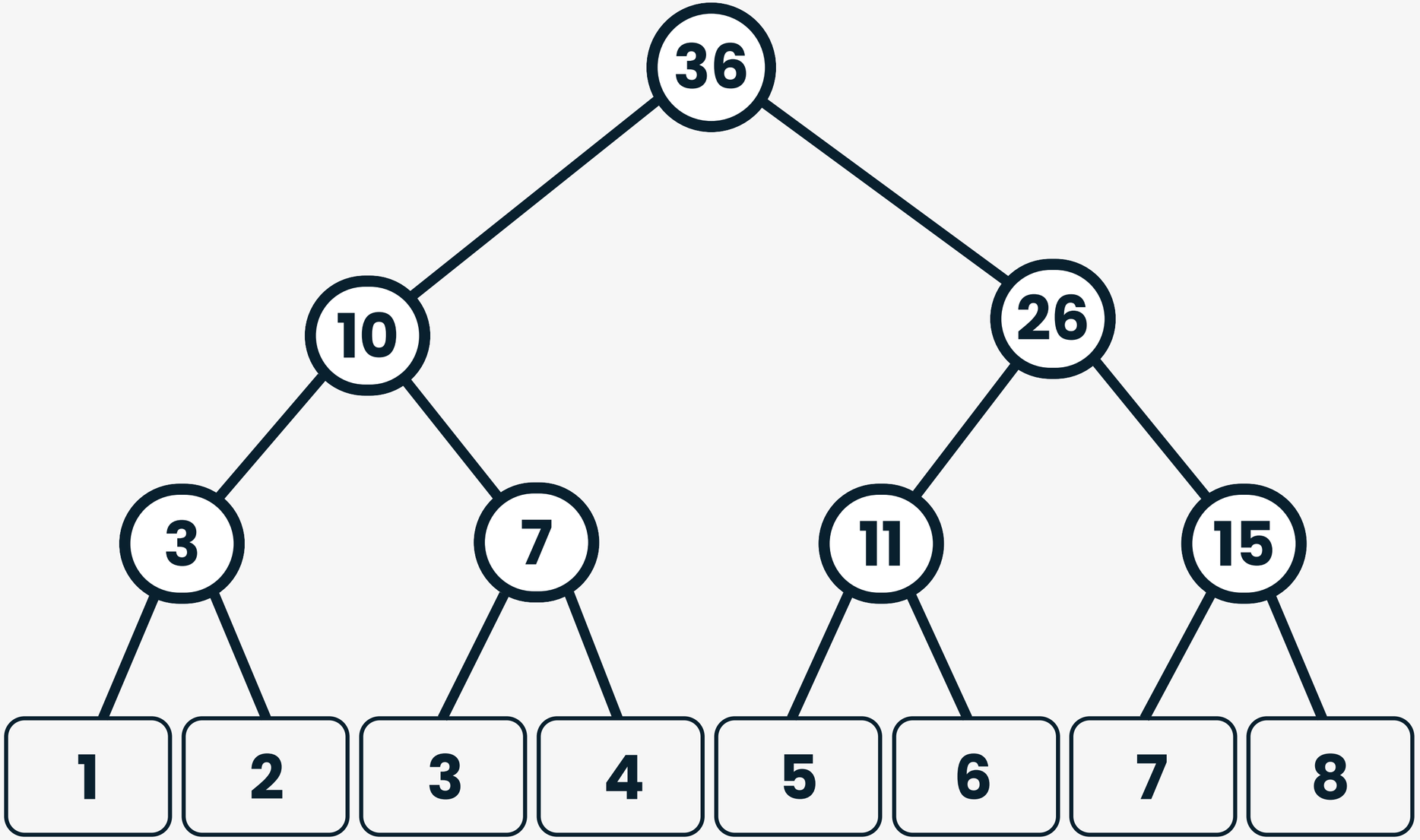
# Construction



# Construction



# Construction



# Question

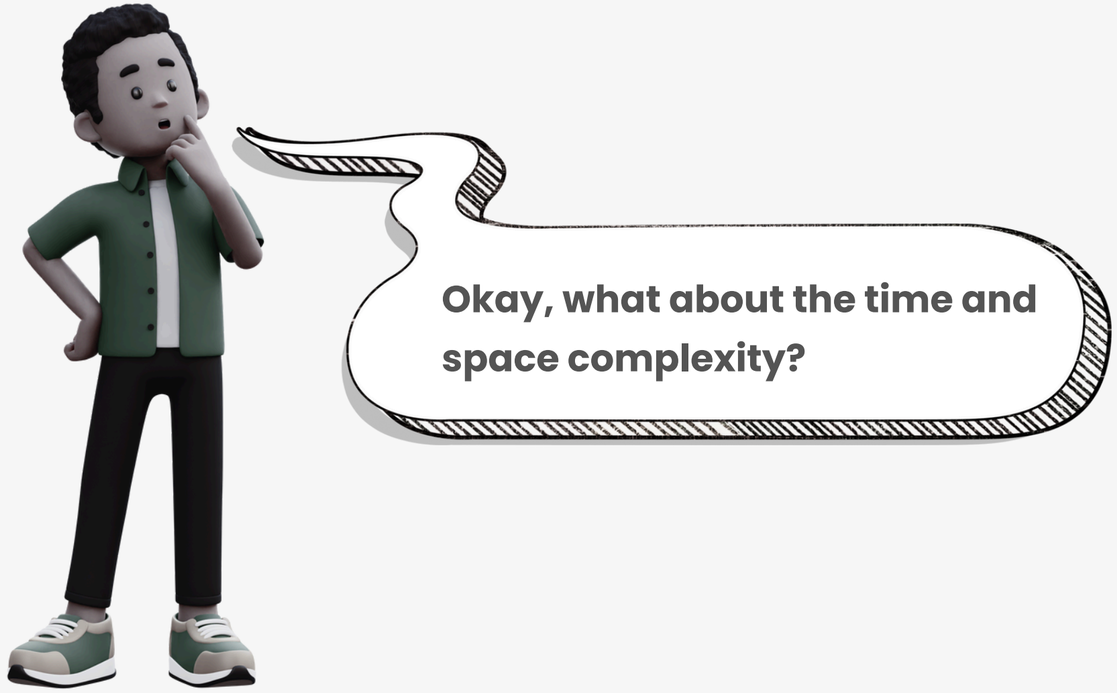
What if the original array’s size is not a power of two?

# Question

We can add a padding of zeros.

**What if the original array's size is not a power of two?**

# Question



# Question

Time: **O(n logn)**

Space: **O(n)**

**Okay, what about the time and space complexity?**

# Operations

Segment tree allows modification of the base array while giving an up to date information. These operations can be applied on a point or a range:

- Update
- Query

# Operations

Let’s go back to the **motivation problem**:

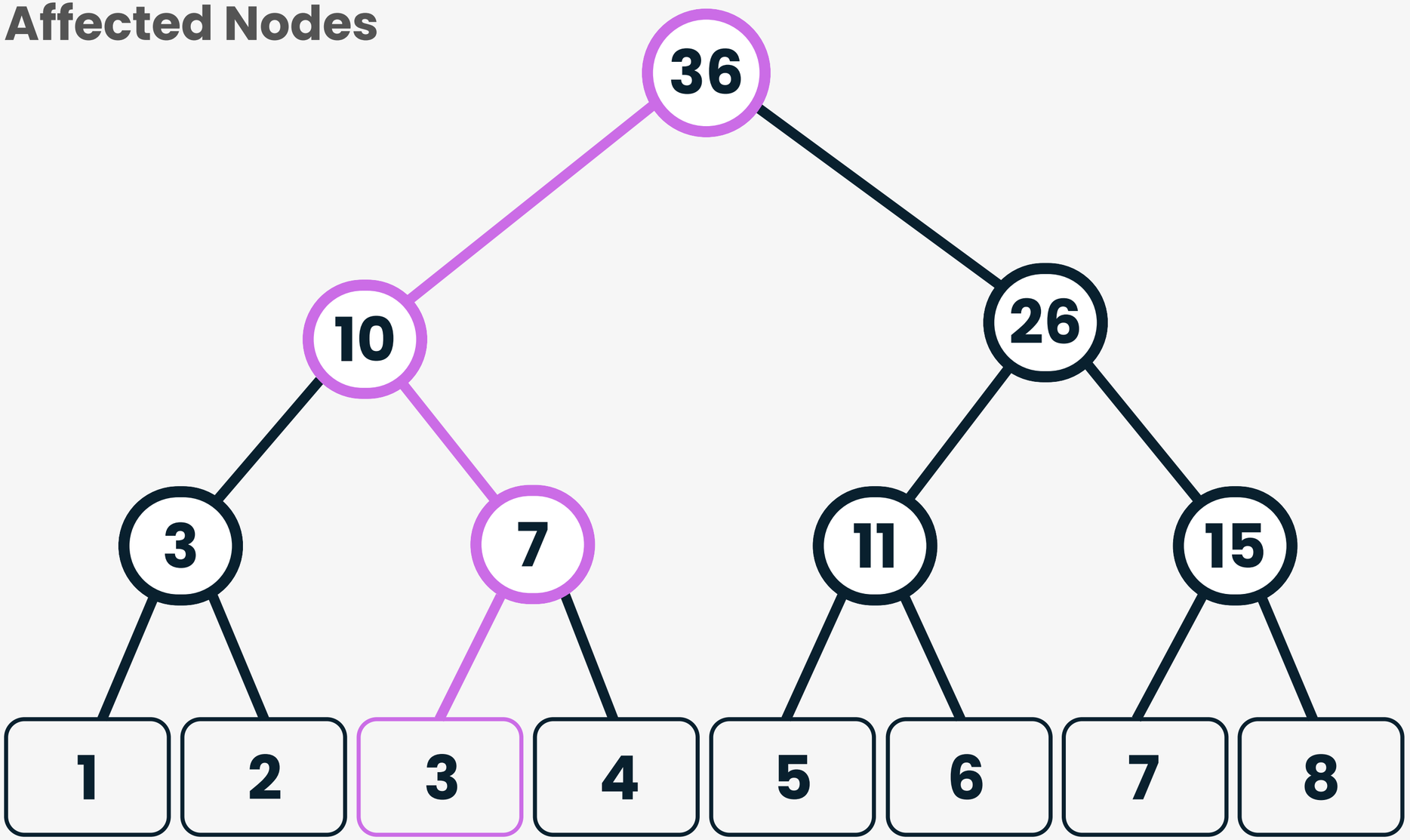
- We need to **update** the value at a given index **i**.
- We need to answer a sum **query** over a range **[l, r]**.

# Update

**Operation**: Make the value at **index 2** equal to **5**.

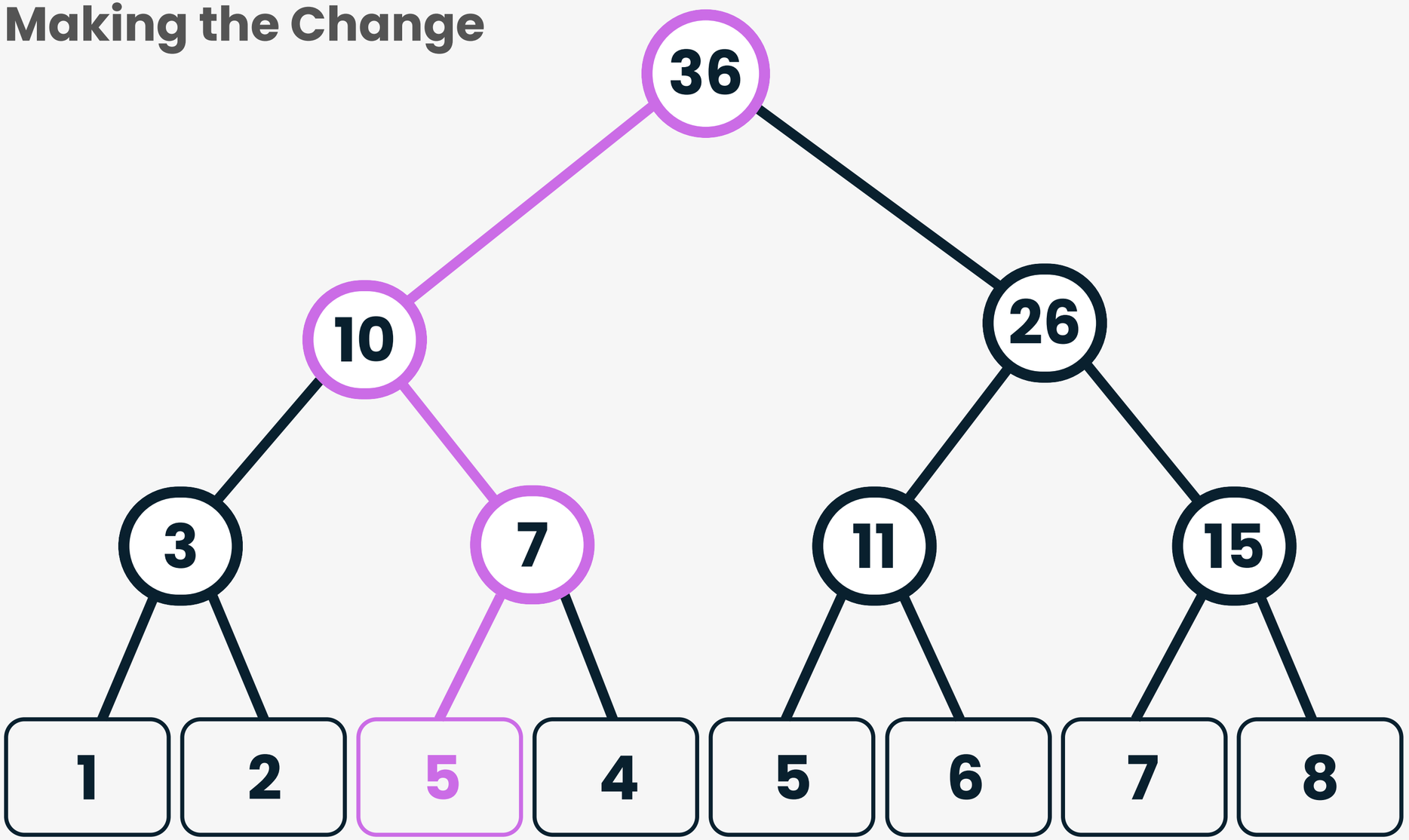
# Update

## Affected Nodes



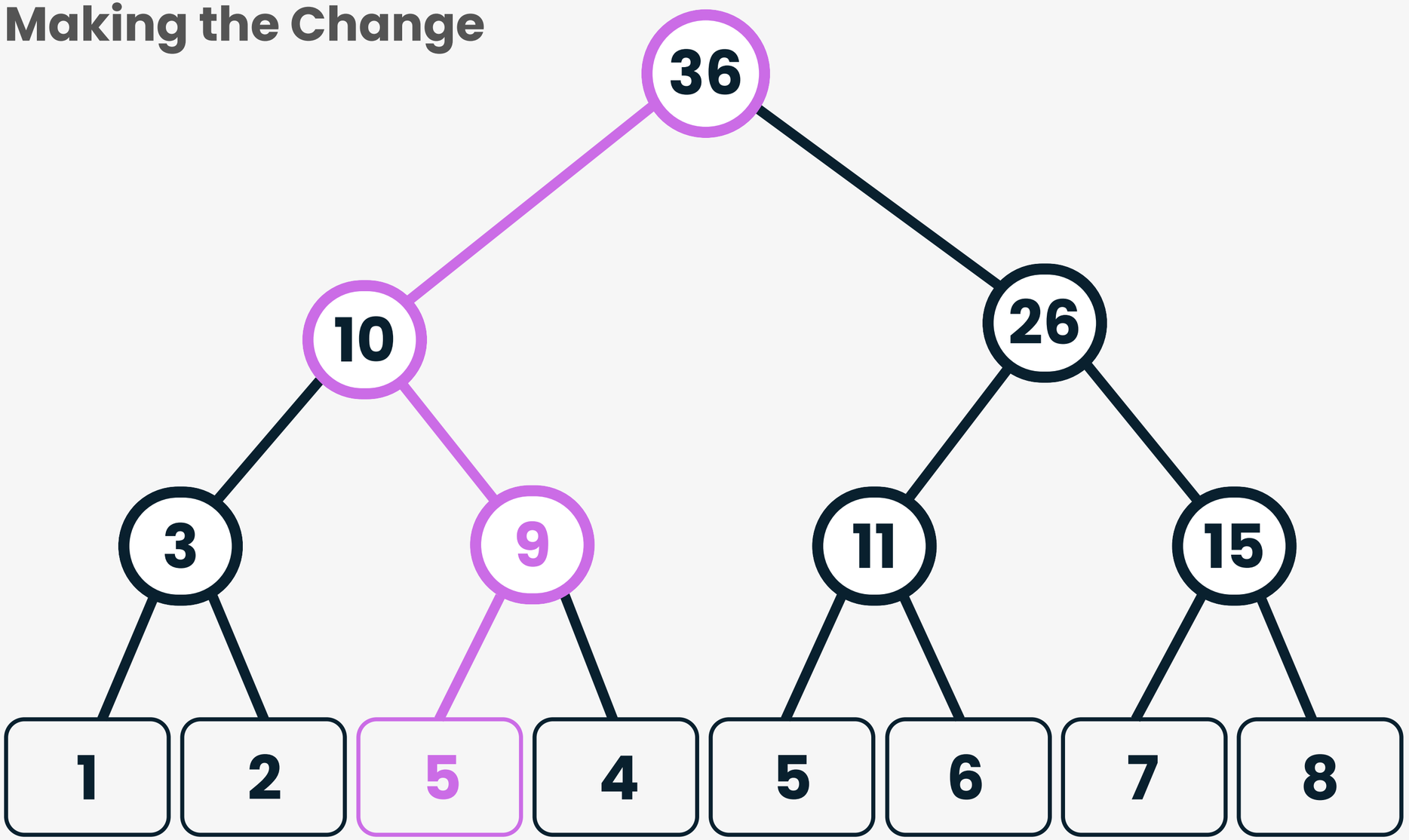
# Update

## Making the Change



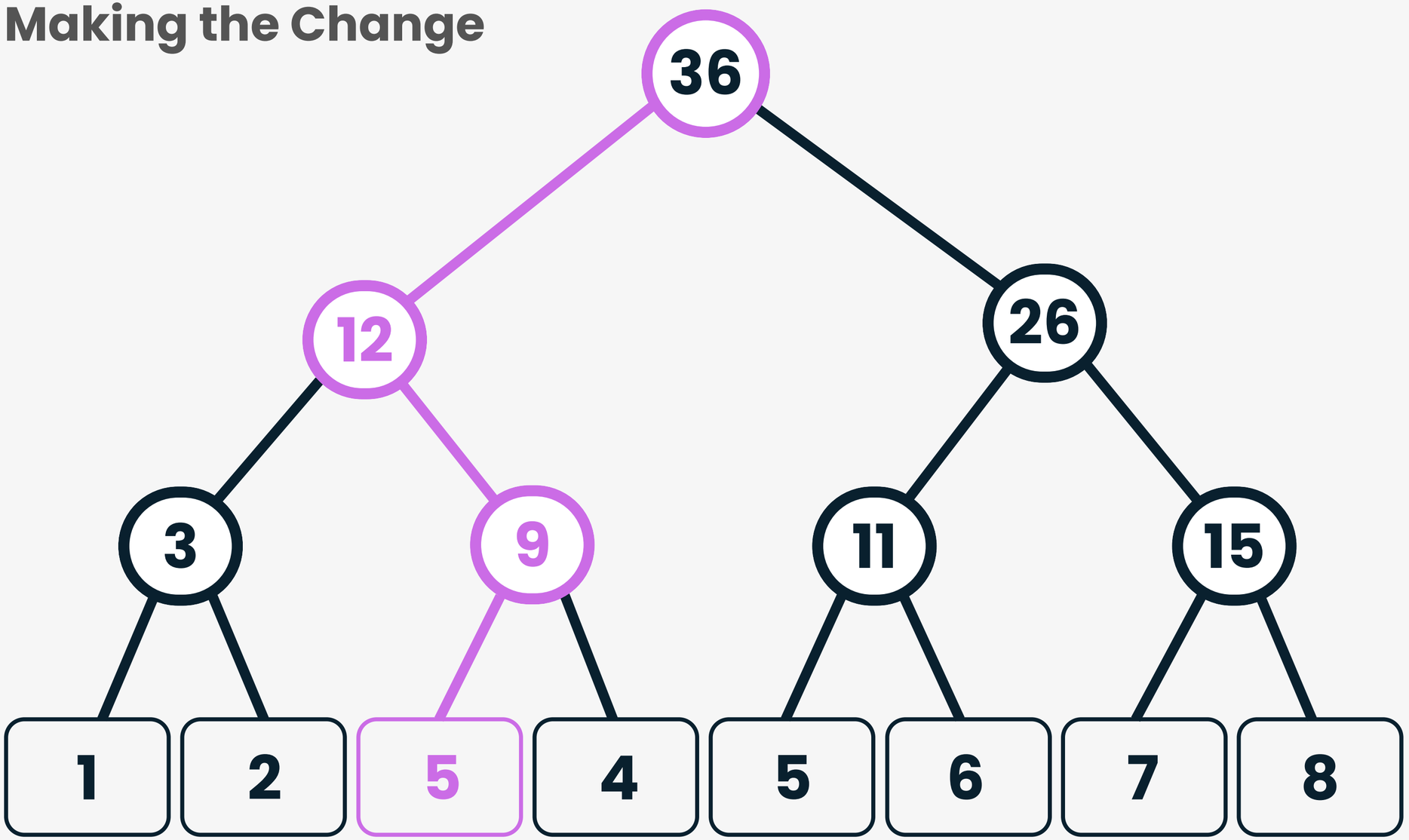
# Update

## Making the Change



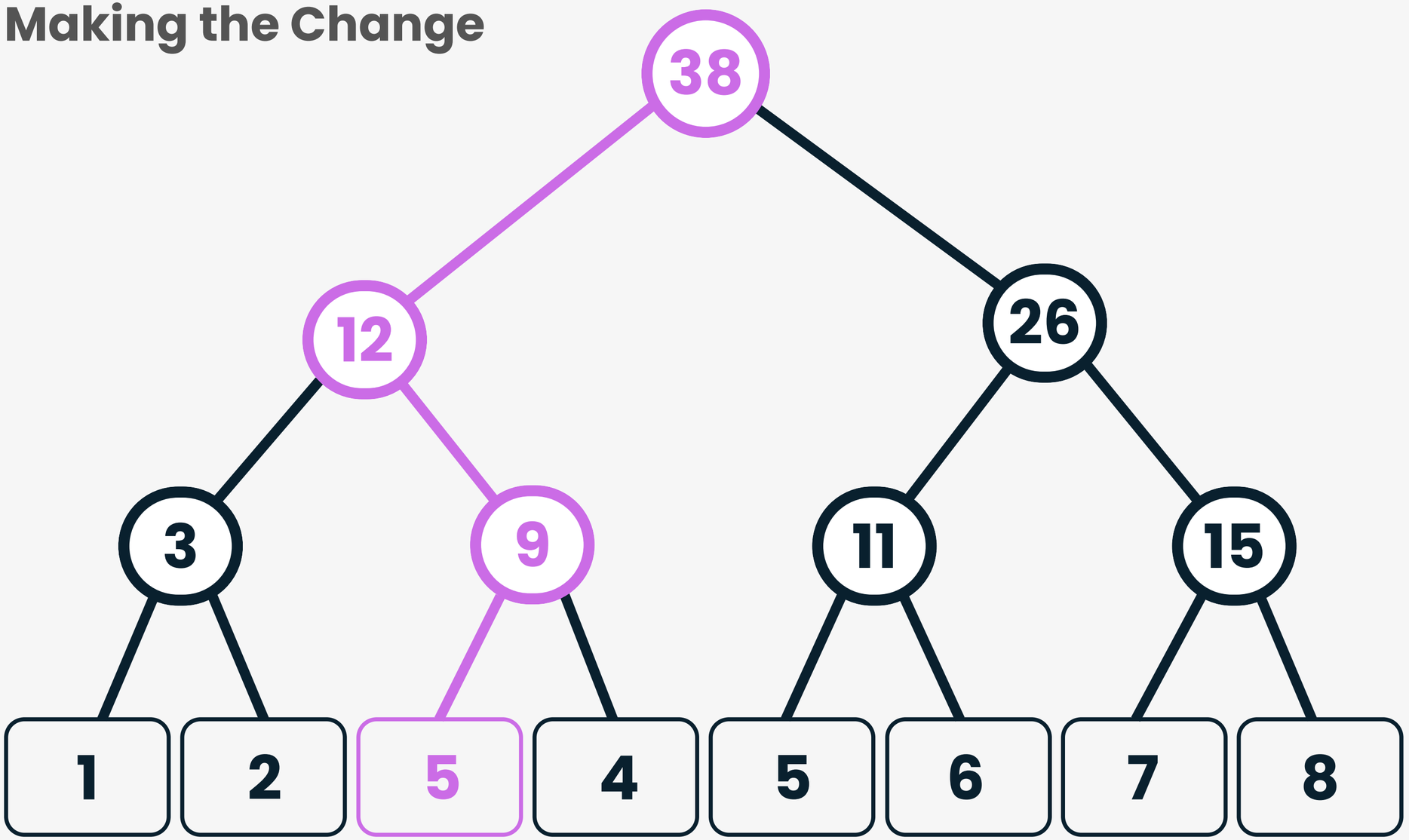
# Update

## Making the Change



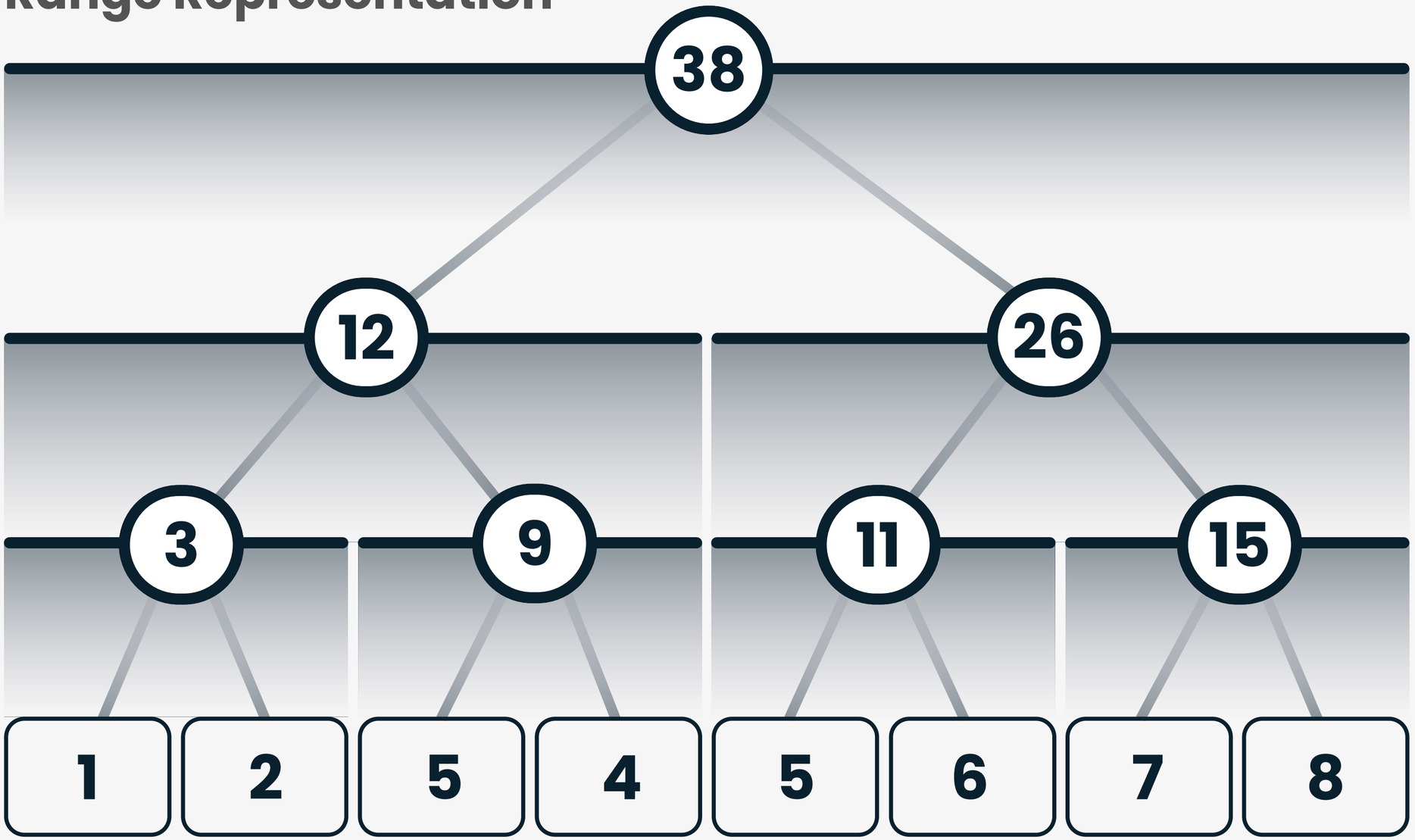
# Update

## Making the Change



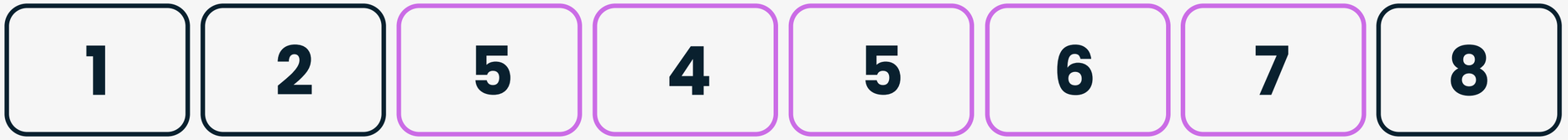
# Query

## Range Representation



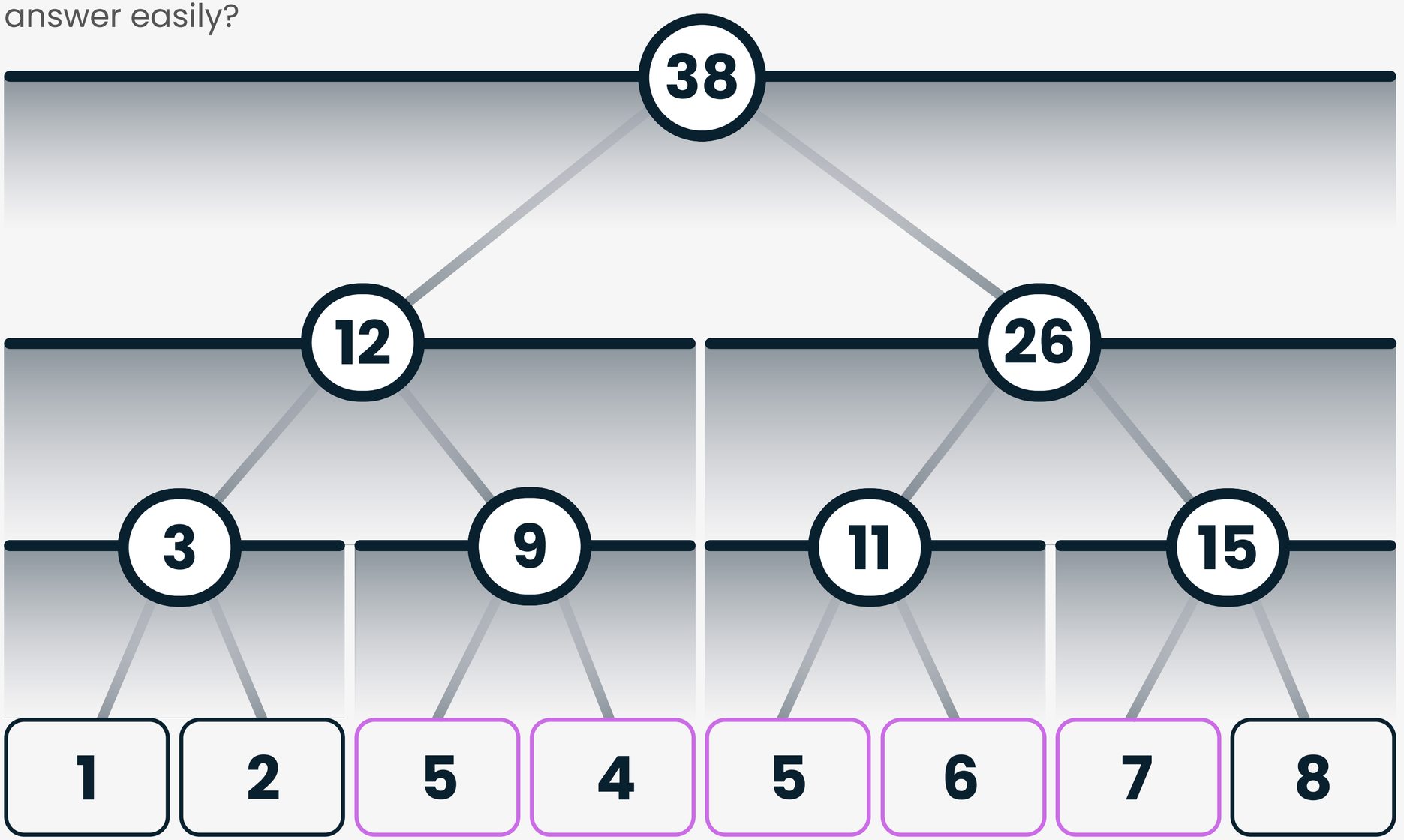
# Query

**Operation**: Find the sum of values at indices **2** to **6** inclusive.



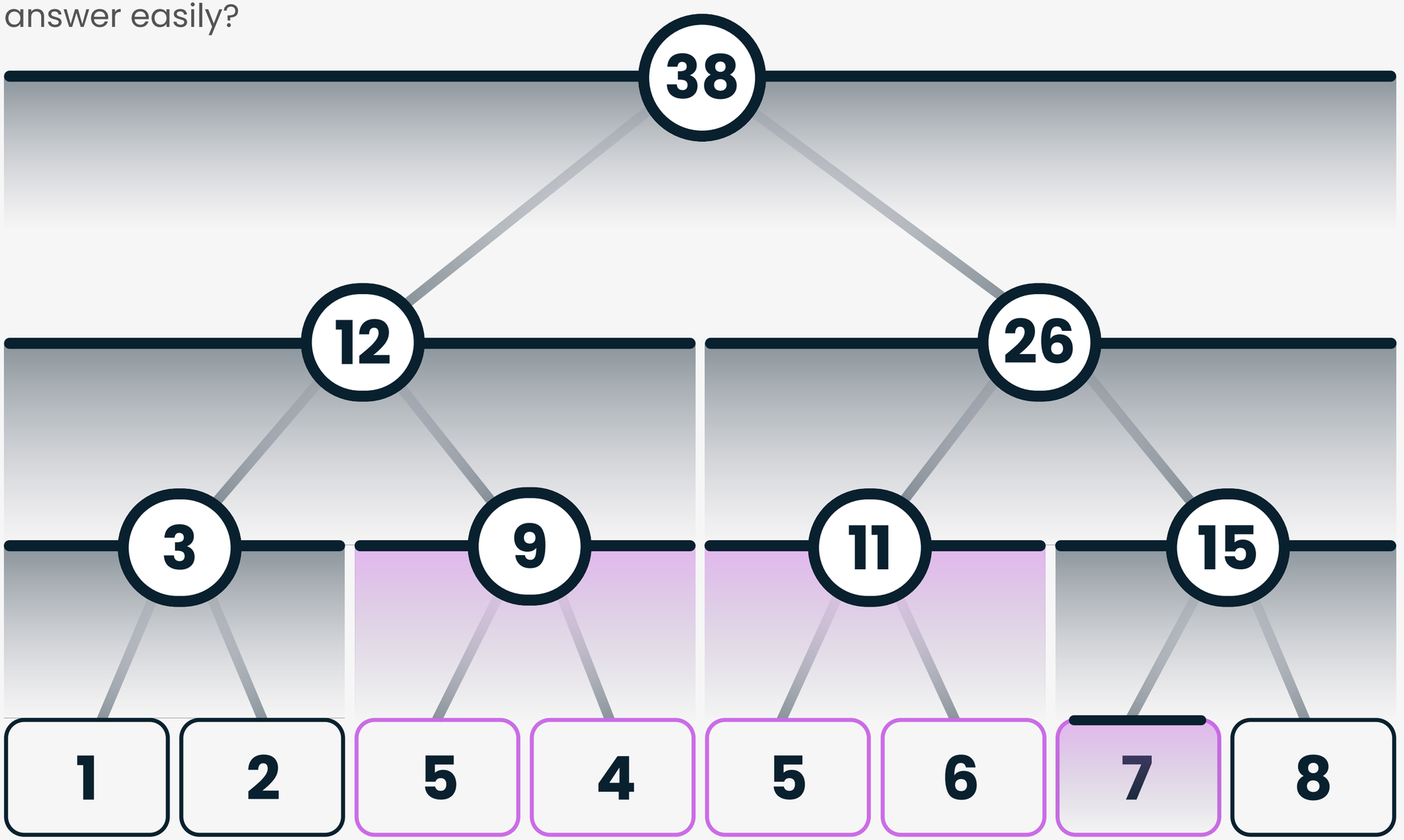
# Query

Which nodes help us get the answer easily?



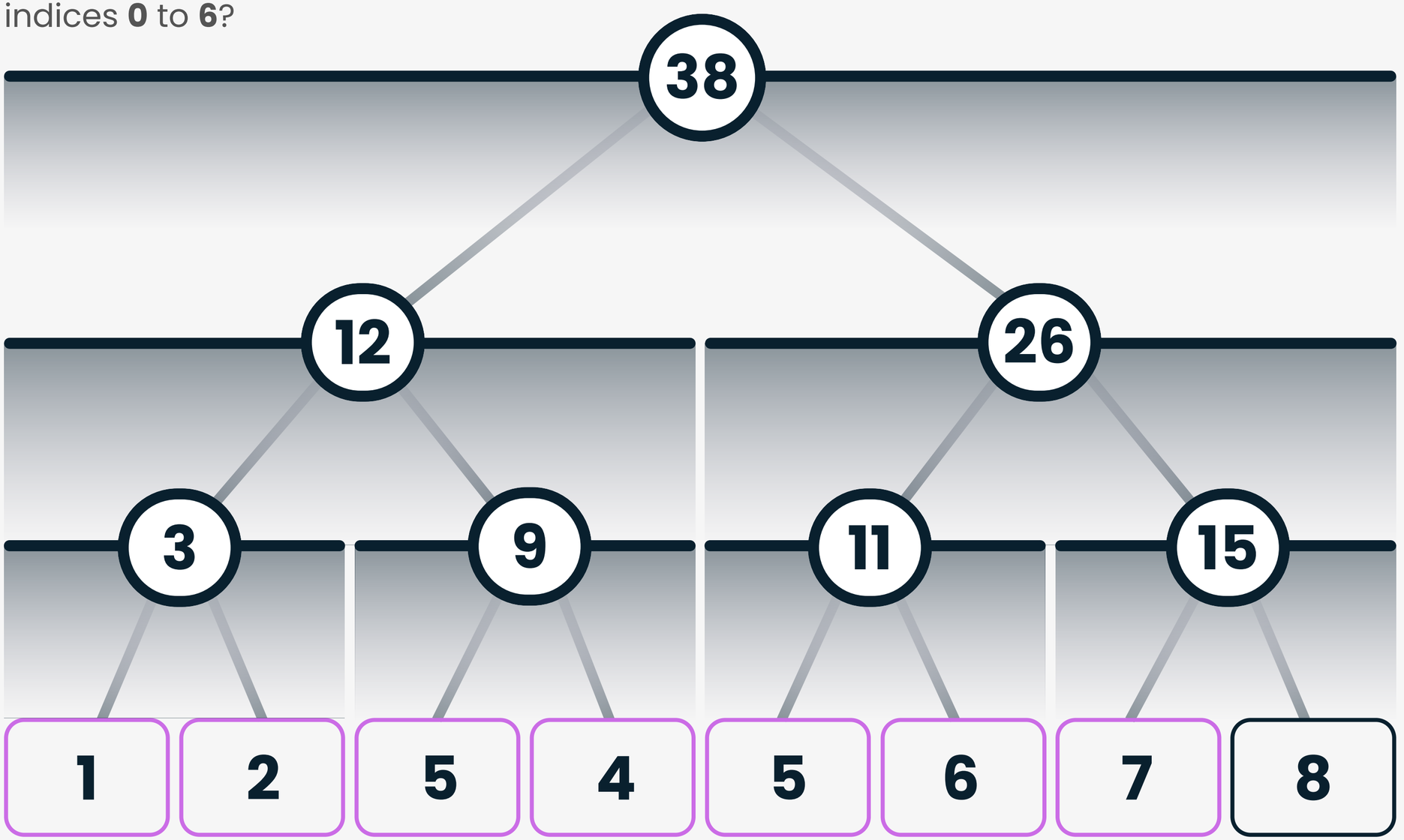
# Query

Which nodes help us get the answer easily?



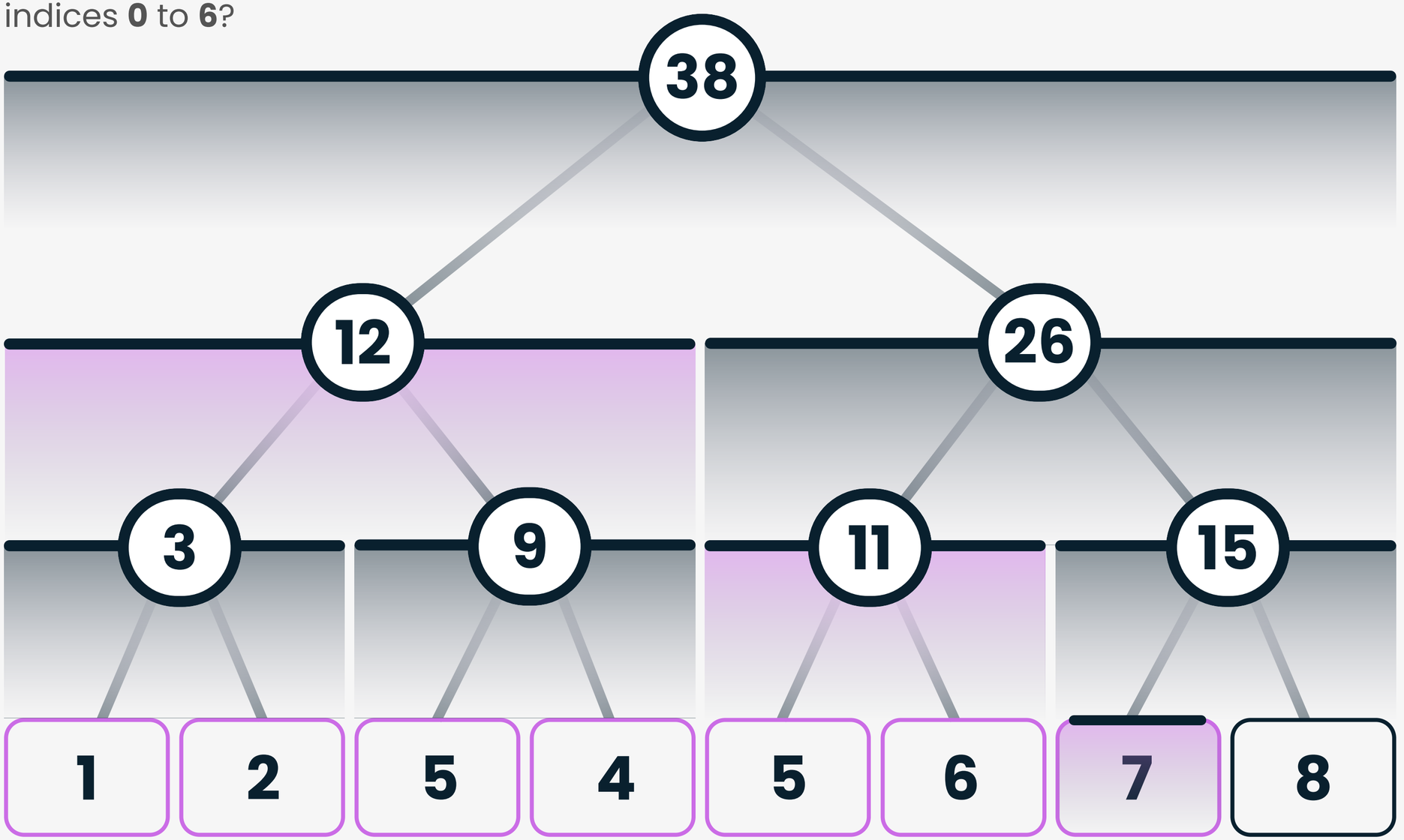
# Query

What is the sum of values at indices **0** to **6**?



# Query

What is the sum of values at indices **0** to **6**?



# Question

So, what’s the time complexity of these operations?

# Question

We can do both in **O(logn)** time.

So, what's the time complexity of these operations?

# Implementation

We could explicitly define a node class and construct the segment tree as a collection of individual nodes. However, a more optimized approach takes advantage of the fact that each node has exactly two children.

By utilizing an array, we can represent the tree **implicitly**, much like the structure of a **binary heap**.

# Simple Implementation

To store the node values in an array, we need to handle the case where **n** is not a **power of 2**, and have enough room in the array.

- **4 * n** space is sufficient. Proof

# Simple Implementation

By following the binary heap structure:

- The **root** will be placed at **index 0**.
- Then, for a non-leaf node stored at  index **i**:
  - The left child is at **2 * i + 1**.
  - The right child is at **2 * i + 2**.

# Simple Implementation

## The Constructor

```python
class SegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.arr = arr
        self.tree = [0] * (4 * self.n)
        self.build(0, 0, self.n - 1)
```

# Simple Implementation

## Getting the Children of a Node

```
def getChildren(self, node):
    left_child = 2*node + 1
    right_child = 2*node + 2
    return left_child, right_child
```

# Simple Implementation

## Tree Construction

```
def build(self, node, nLeft, nRight):
    """
    nLeft  : left boundary of the range that the node represents
    nRight : right boundary of the range that the node represents
    """

    if nLeft == nRight:
        self.tree[node] = self.arr[nLeft]
        return

    nMid = (nLeft + nRight)//2
    left_child, right_child = self.getChildren(node)

    self.build(left_child, nLeft, nMid)
    self.build(right_child, nMid + 1, nRight)

    self.tree[node] = self.tree[left_child] + self.tree[right_child]
```

# Simple Implementation

```
def update(self, node, nLeft, nRight, index, value):
    """
    nLeft  : left boundary of the range that the node represents
    nRight : right boundary of the range that the node represents
    """

    if nLeft == nRight:
        self.arr[index] = value
        self.tree[node] = value
        return

    nMid = (nLeft + nRight)//2
    left_child, right_child = self.getChildren(node)

    if index <= nMid:
        self.update(left_child, nLeft, nMid, index, value)
    else:
        self.update(right_child, nMid + 1, nRight, index, value)

    #reconstruction
    self.tree[node] = self.tree[left_child] + self.tree[right_child]
```

# Simple Implementation

```python
def query(self, node, nLeft, nRight, qLeft, qRight):
    """
    nLeft  : left boundary of the range that the node represents
    nRight : right boundary of the range that the node represents
    qLeft  : left boundary of the query
    qRight : right boundary of the query
    """

    if qLeft > qRight:
        return 0

    if qLeft == nLeft and qRight == nRight:
        return self.tree[node]

    nMid = (nLeft + nRight)//2
    left_child, right_child = self.getChildren(node)

    left_sum = self.query(left_child, nLeft, nMid, qLeft, min(nMid, qRight))
    right_sum = self.query(right_child, nMid + 1, nRight, max(nMid + 1, qLeft), qRight)

    return left_sum + right_sum
```

# Simple Implementation

## Interfaces for Easy Use

```
def setIndex(self, index, value):
    self.update(0, 0, self.n - 1, index, value)

def getRangeSum(self, left, right):
    return self.query(0, 0, self.n - 1, left, right)
```

# Practice

LeetCode | Range Sum Query Mutable

# Memory-Efficient Implementation

A segment tree of an array of **n** elements requires only **2 * n - 1** nodes.

- We can follow a **pre-order traversal** to avoid the unnecessary padding.

# Memory-Efficient Implementation

Let’s take the node **i** and let it be responsible for the segment **[l, r]**.

- The left child will have the index **i + 1**.
- The left child is responsible for the segment **[l, mid]**.
    - In total, there will be **2 * (mid – l + 1) – 1** nodes in the left subtree.
- Thus, The index of the right child will be **i + 2 * (mid – l + 1)**

# Memory-Efficient Implementation

## The Constructor

```
def __init__(self, arr):
    self.n = len(arr)
    self.arr = arr
    self.nodes = 2*self.n - 1
    self.tree = [0]*self.nodes
    self.build(0, 0, self.n - 1)
```

# Memory-Efficient Implementation

## Getting the Children

```python
def getChildren(self, node):
    left_child = 2*node + 1
    right_child = 2*node + 2
    return left_child, right_child
```



```python
def getChildren(self, node, nLeft, nRight):
    nMid = (nLeft + nRight)//2

    # number of nodes in the left subtree
    left_subtree = 2*(nMid - nLeft + 1) - 1
    left_child = node + 1
    right_child = node + left_subtree + 1
    return left_child, right_child
```

# Practice

LeetCode | Peaks in Array

# Part II

# Iterative Implementation

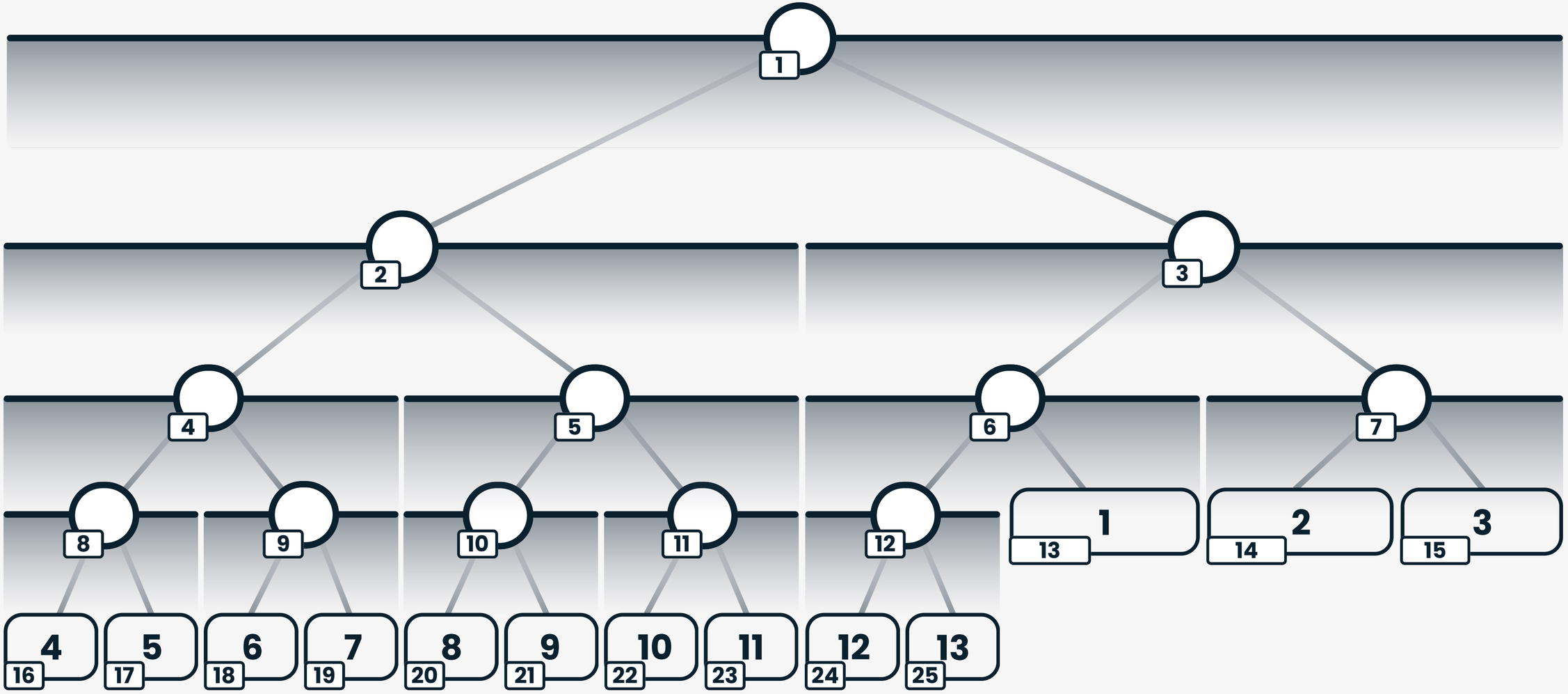
This is based on the idea that **leaves** are stored in **continuous** nodes with indices starting with **n**, element with index **i** corresponds to a node with index **i + n**. (The tree is **1-indexed** while the array is **0-indexed**).

- The ancestors of the leaves fill up the first **n** places of the tree array.
- The **root** takes the index **1**.

# Construction

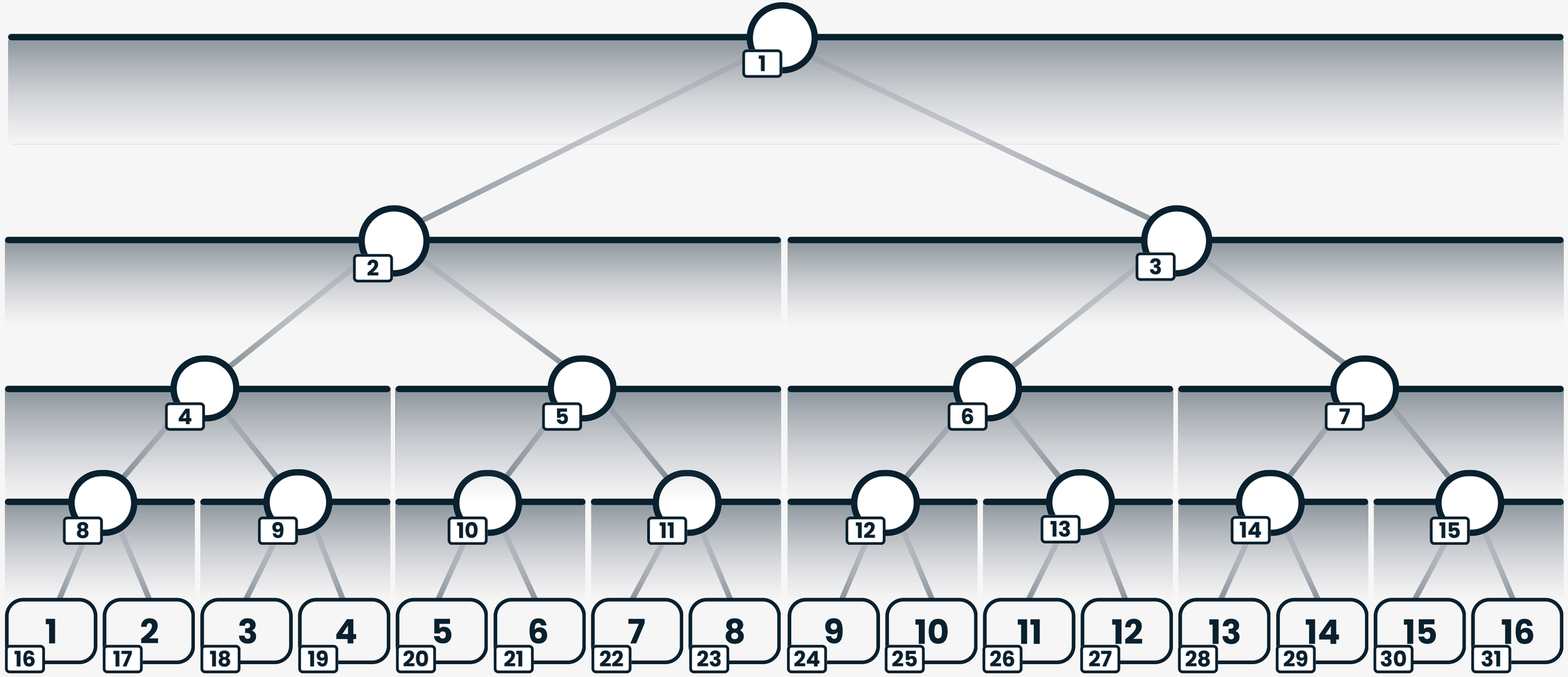
Value
index

Consider an example where **n = 13**.
The tree would look like as follows.



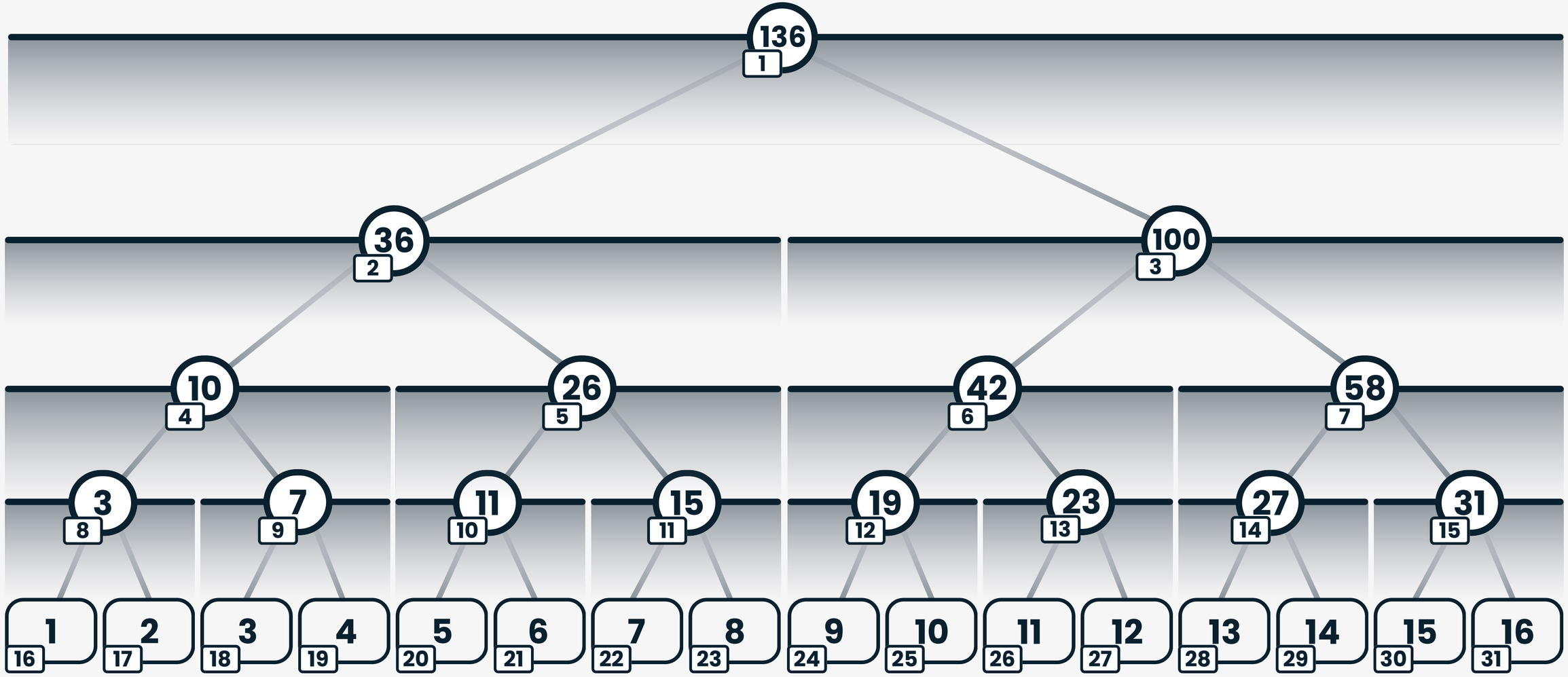
# Construction


For simplicity, let's take this example where **n = 16**.
We first populate the leave nodes starting from node **n**.



# Construction


For the remaining nodes, from **n - 1** to **1** inclusive, we'll compute **tree[i] = tree[2*i] + tree[2*i + 1]**.



# Iterative Implementation

## The Constructor

```
class SegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.arr = arr
        self.tree = [0] * (2*self.n)
        self.build()
```

# Iterative Implementation

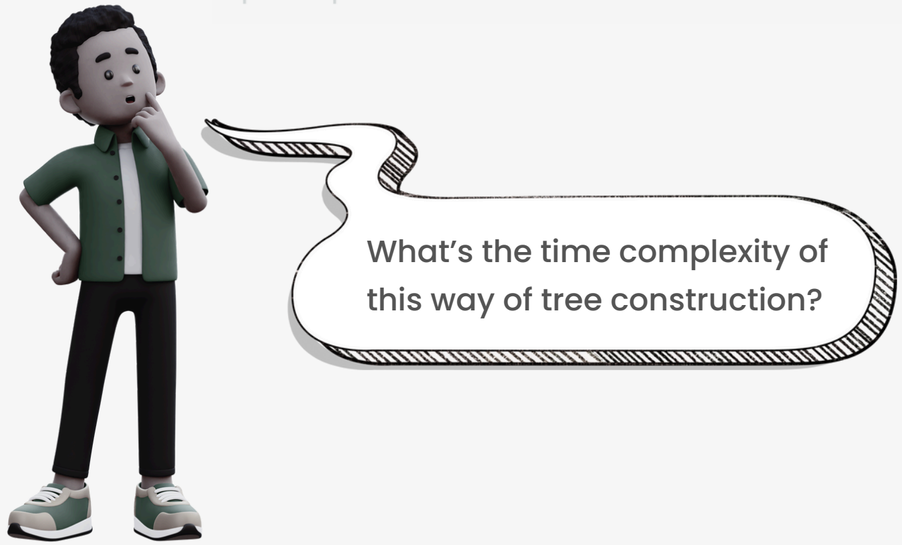
## Tree Construction

```python
def build(self):
    for index in range(self.n):
        self.tree[index + self.n] = self.arr[index]

    for index in range(self.n - 1, 0, -1):
        self.tree[index] = self.tree[index << 1] + self.tree[index << 1 | 1]
```

# Iterative Implementation

## Tree Construction

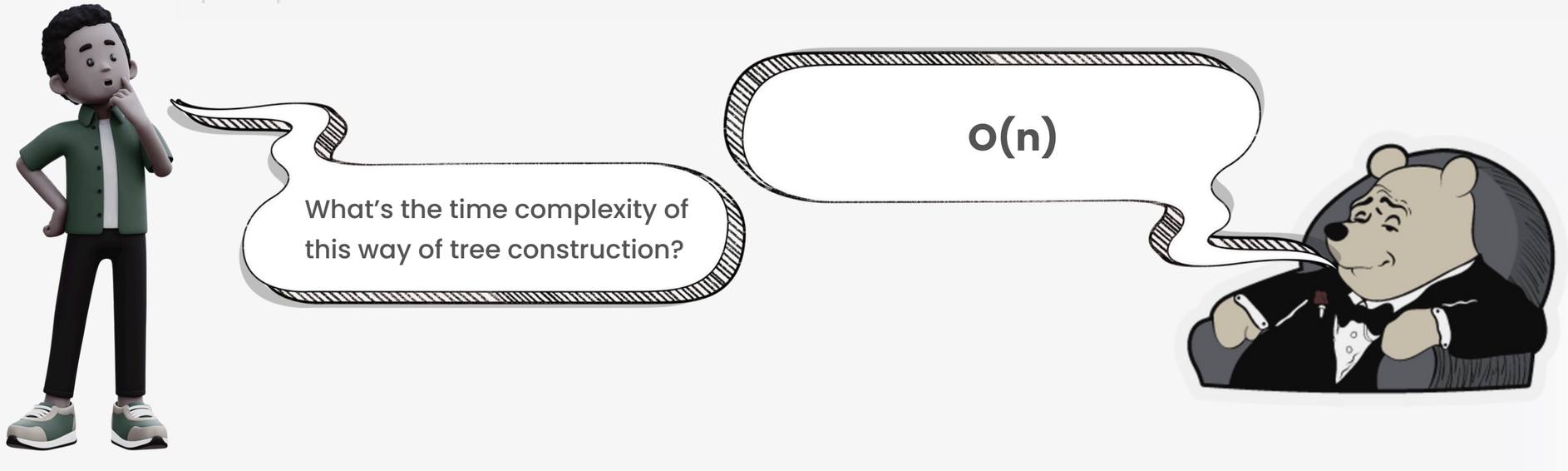
```python
def build(self):
    for index in range(self.n):
        self.tree[index + self.n] = self.arr[index]

    for index in range(self.n - 1, 0, -1):
        self.tree[index] = self.tree[index << 1] + self.tree[index << 1 | 1]
```




# Iterative Implementation

## Tree Construction

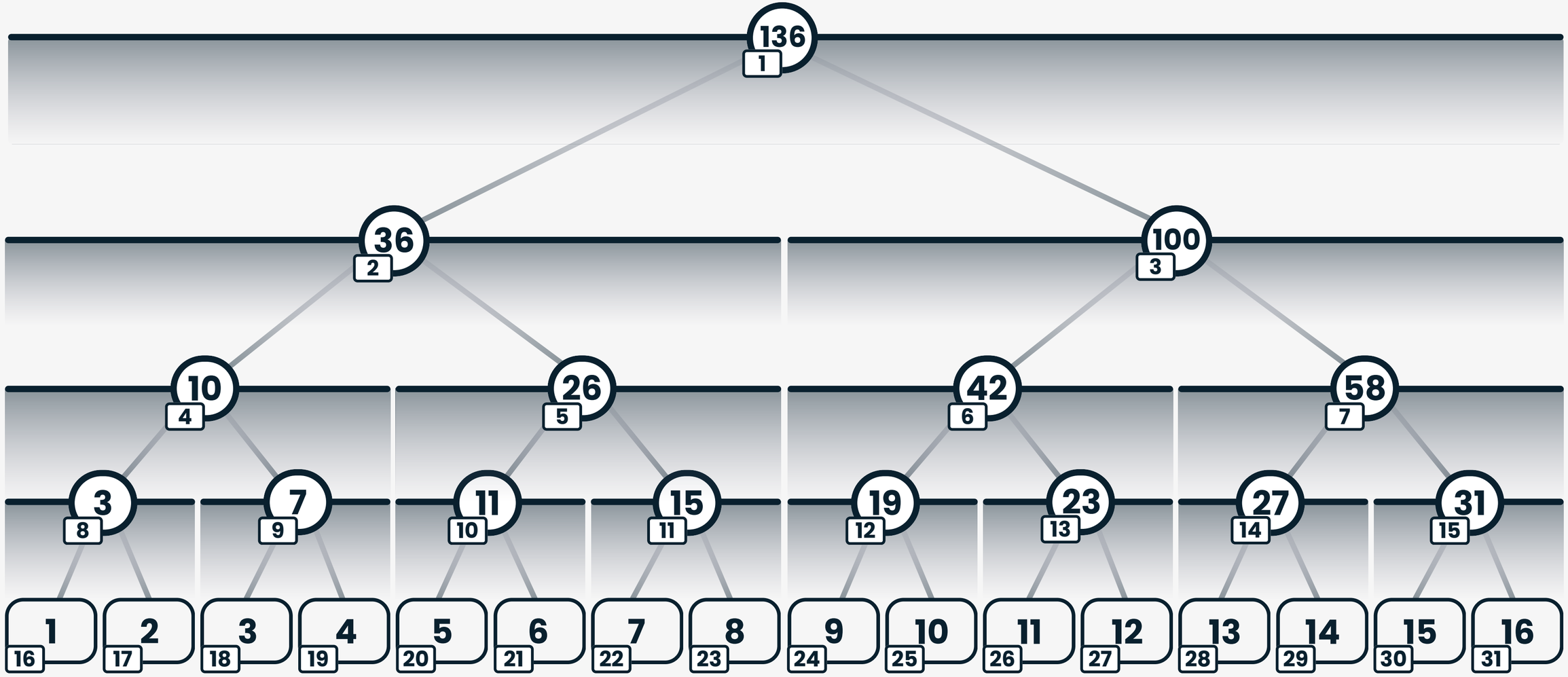
```python
def build(self):
    for index in range(self.n):
        self.tree[index + self.n] = self.arr[index]

    for index in range(self.n - 1, 0, -1):
        self.tree[index] = self.tree[index << 1] + self.tree[index << 1 | 1]
```



# Update

Value
index

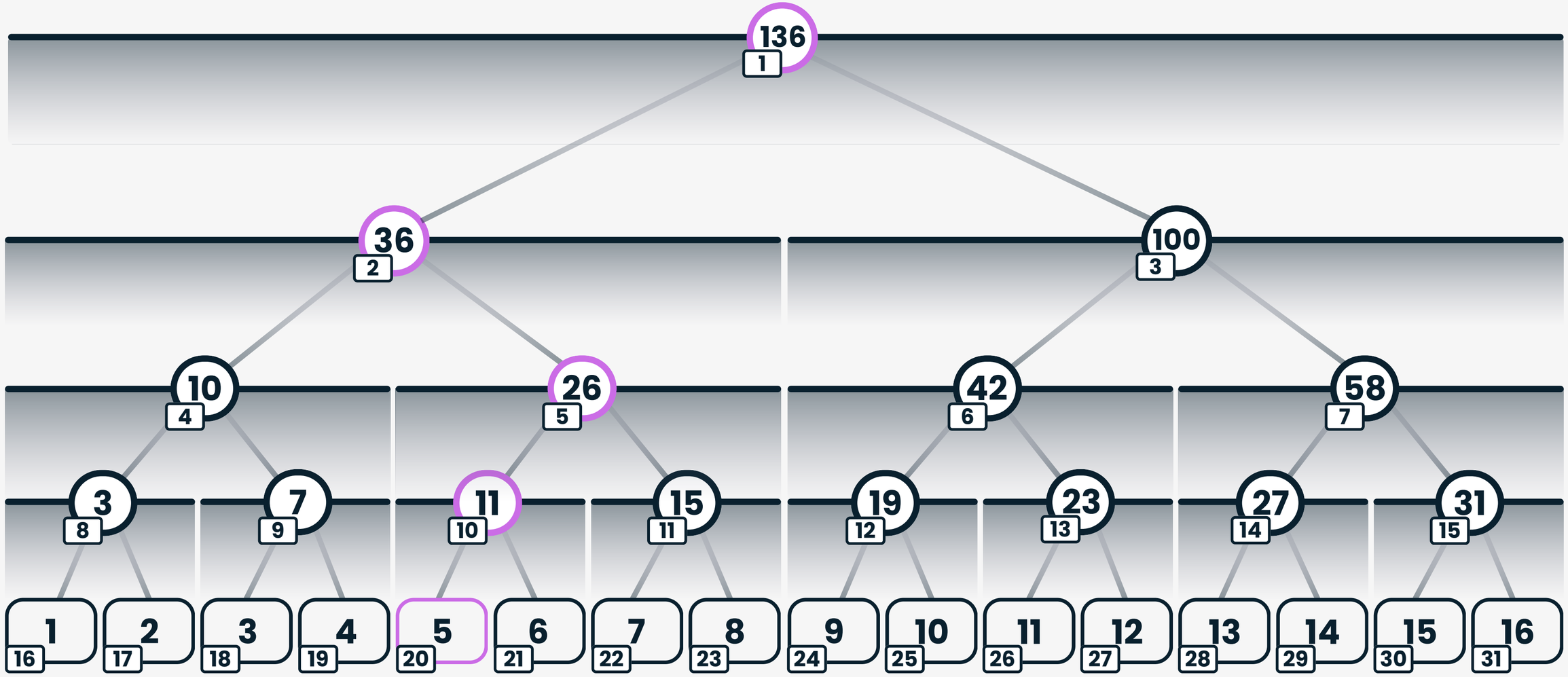
Let's take this example where **n = 16**.



# Update

Value
index

Updating index **i** in the original array, affects nodes in the path **n + i, (n + i)//2, ..., 1.**



# Iterative Implementation

```
def update(self, index, value):

    self.arr[index] = value
    index += self.n
    self.tree[index] = value

    while index > 1:
        self.tree[index >> 1] = self.tree[index] + self.tree[index ^ 1]
        index >>= 1
```

# Query

The idea behind the iterative query function is whether we should include an element in the sum or whether we should include its parent. Consider that **L** is the left border of an interval and **R** is the right border of the interval **[L,R)**.

- If **L** is **odd**, it means that it is the **right child** of its parent and our interval includes **only L** and **not** the parent.

# Query

The idea behind the iterative query function is whether we should include an element in the sum or whether we should include its parent. Consider that **L** is the left border of an interval and **R** is the right border of the interval **[L,R)**.

- So we will **include** this node to sum and move to the parent of its **next node** by doing **L** = **(L+1)/2**.
- If it was the left child, we would directly skip to the parent.

# Query

The idea behind the iterative query function is whether we should include an element in the sum or whether we should include its parent. Consider that **L** is the left border of an interval and **R** is the right border of the interval **[L,R)**.

From the right end of the interval,

- If **R** is **odd** (right child), then **R - 1** (left child) should be included in the sum.

# Query

The idea behind the iterative query function is whether we should include an element in the sum or whether we should include its parent. Consider that **L** is the left border of an interval and **R** is the right border of the interval **[L,R)**.

From the right end of the interval,

- So we add node **R – 1**, and move to the parent **(R – 1)//2.**
- If **R** was even, we would directly skip to the parent.

# Query

The idea behind the iterative query function is whether we should include an element in the sum or whether we should include its parent. Consider that **L** is the left border of an interval and **R** is the right border of the interval **[L,R)**.

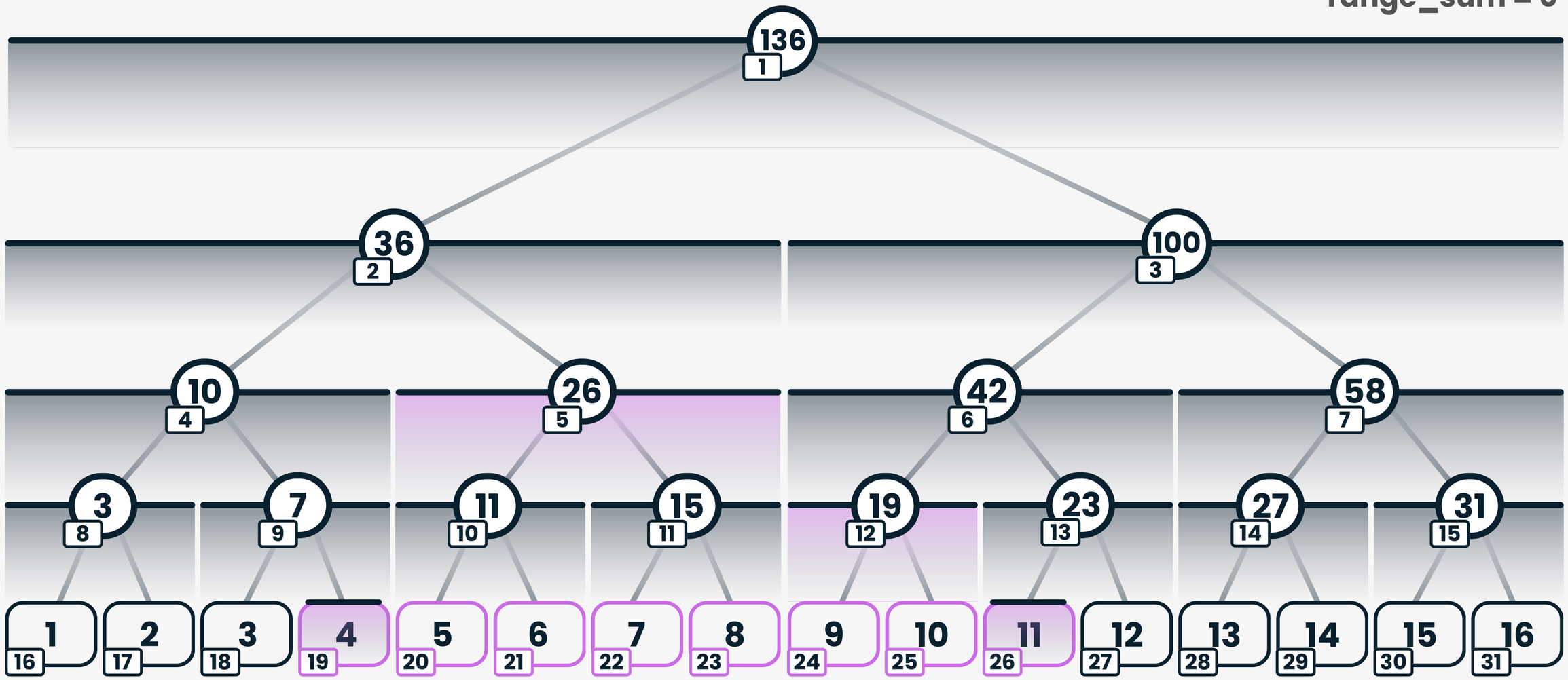
- We keep doing this as long as **L < R**.

# Query

Value
index

Let's find the sum in the range **[3, 11)**.
Now, **L** = 3 + n = 3 + 16 = **19**,
and **R** = 11 + n = 11 + 16 = **27**.

**range_sum = 0**

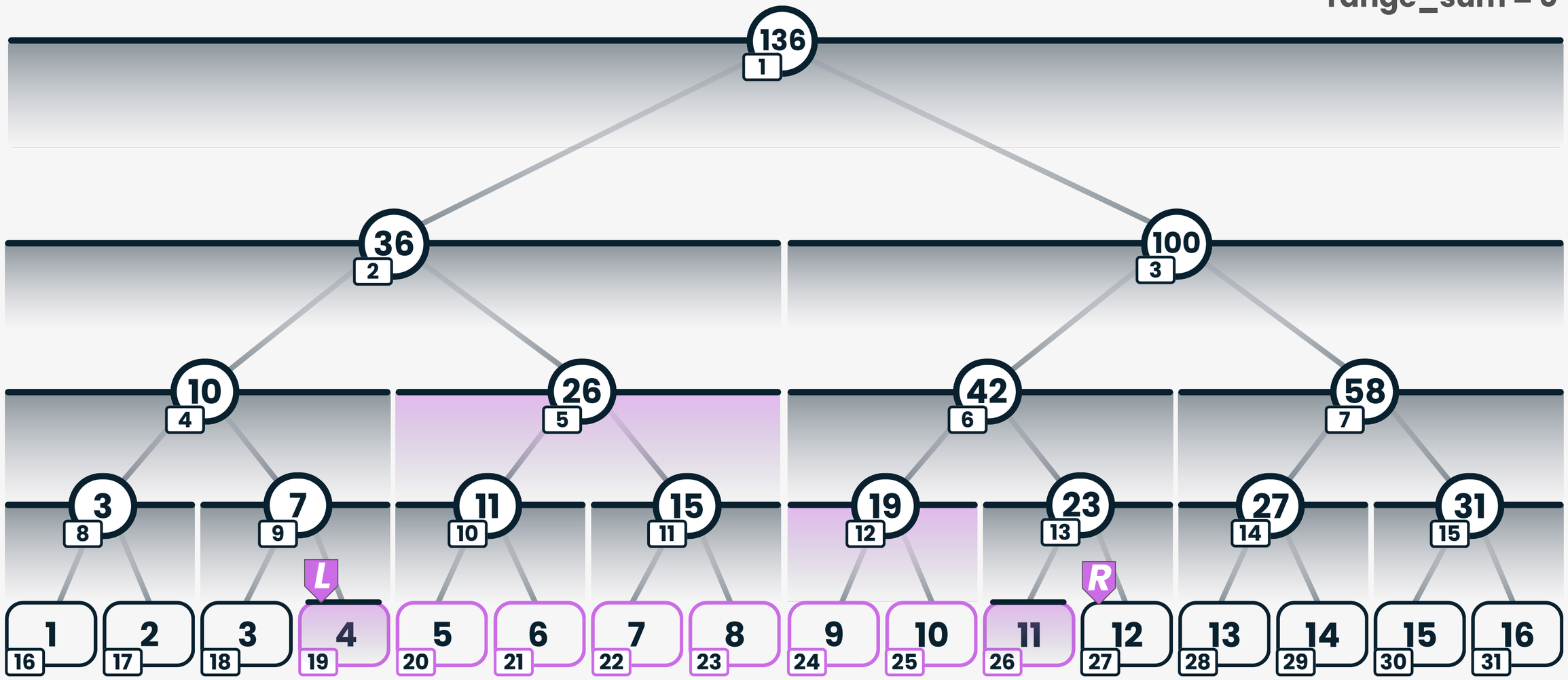


# Query

Value
index

Let's find the sum in the range **[3, 11)**.
Now, **L** = 3 + n = 3 + 16 = **19**,
and **R** = 11 + n = 11 + 16 = **27**.

**range_sum = 0**

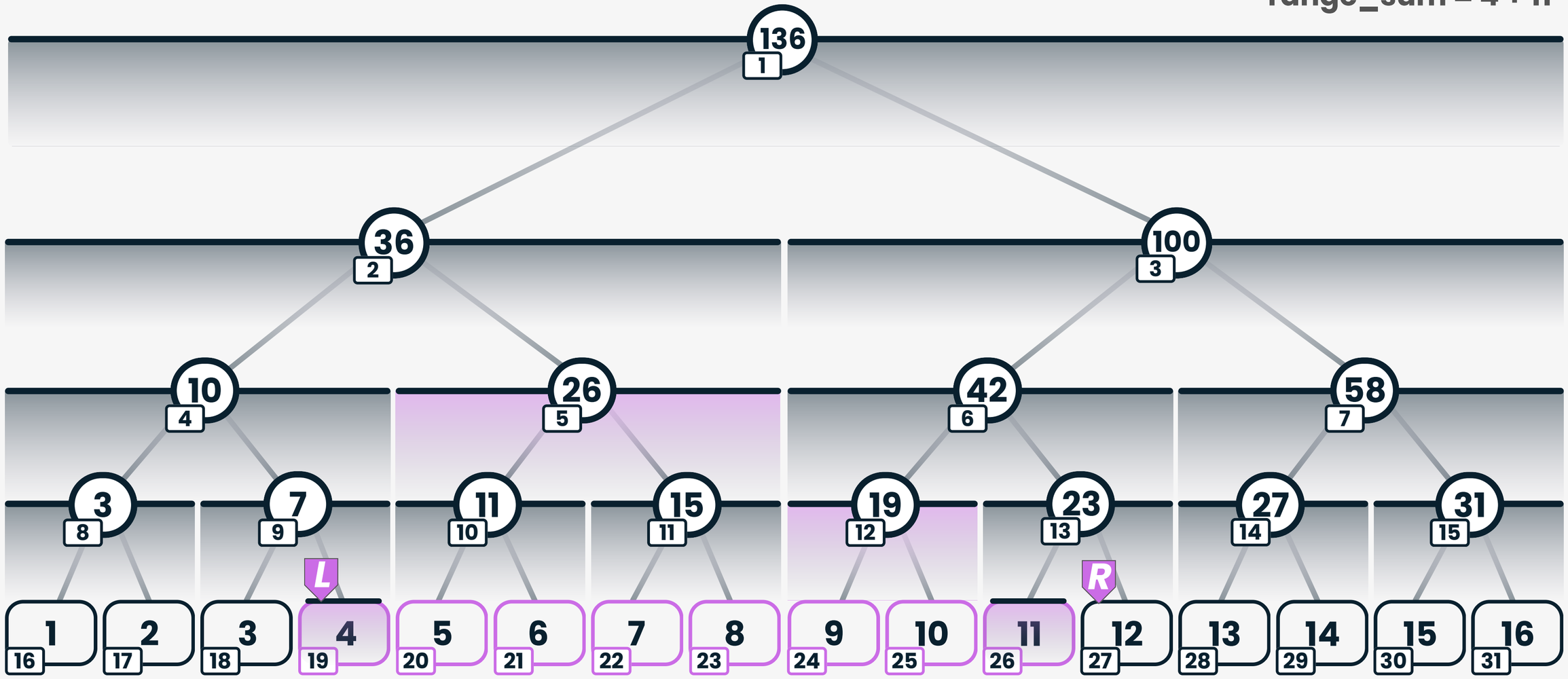


# Query

Value
index

**L** is odd, therefore it should be included in the sum.
**R** is odd, **R - 1** should be included in the sum.

**range_sum = 4 + 11**

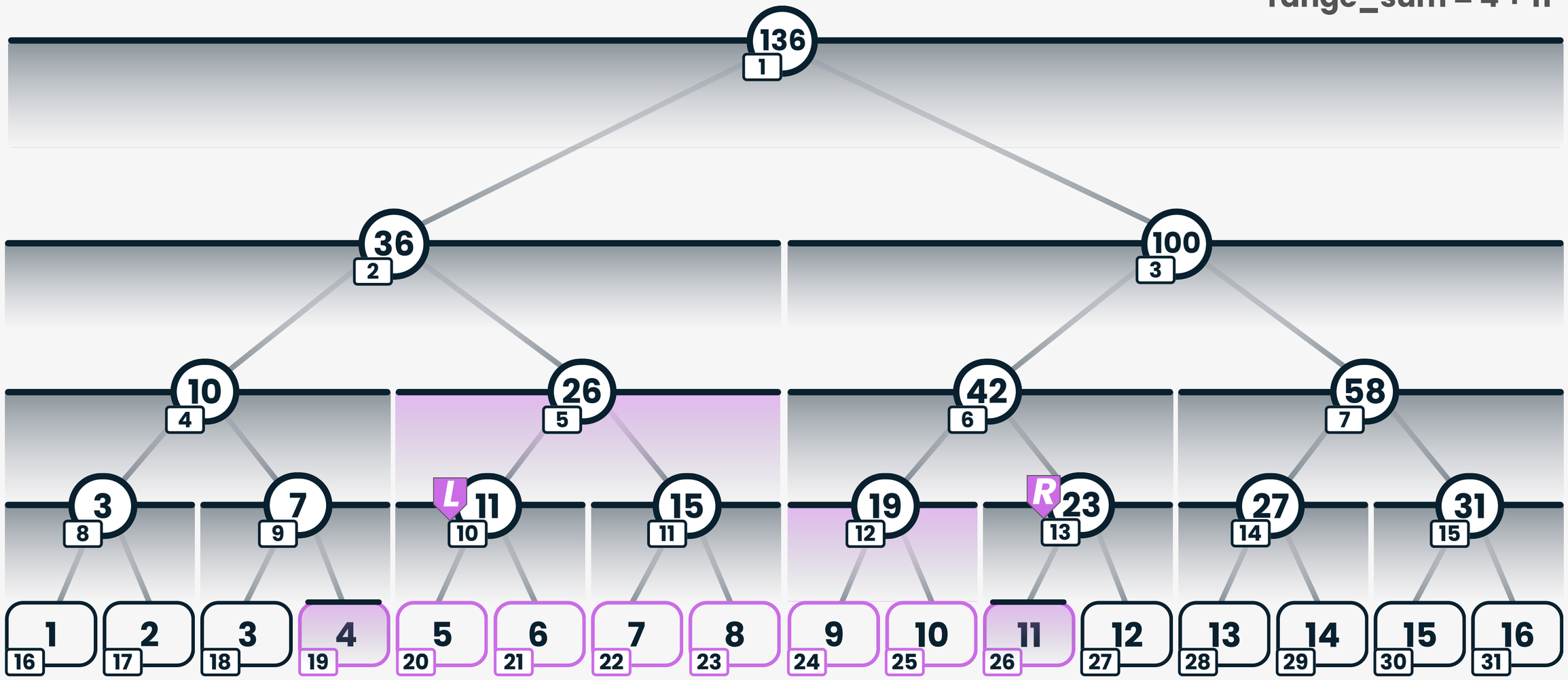


# Query

Value
index

**L** changes to **(L + 1)//2**.
**R** changes to **(R-1)//2**.

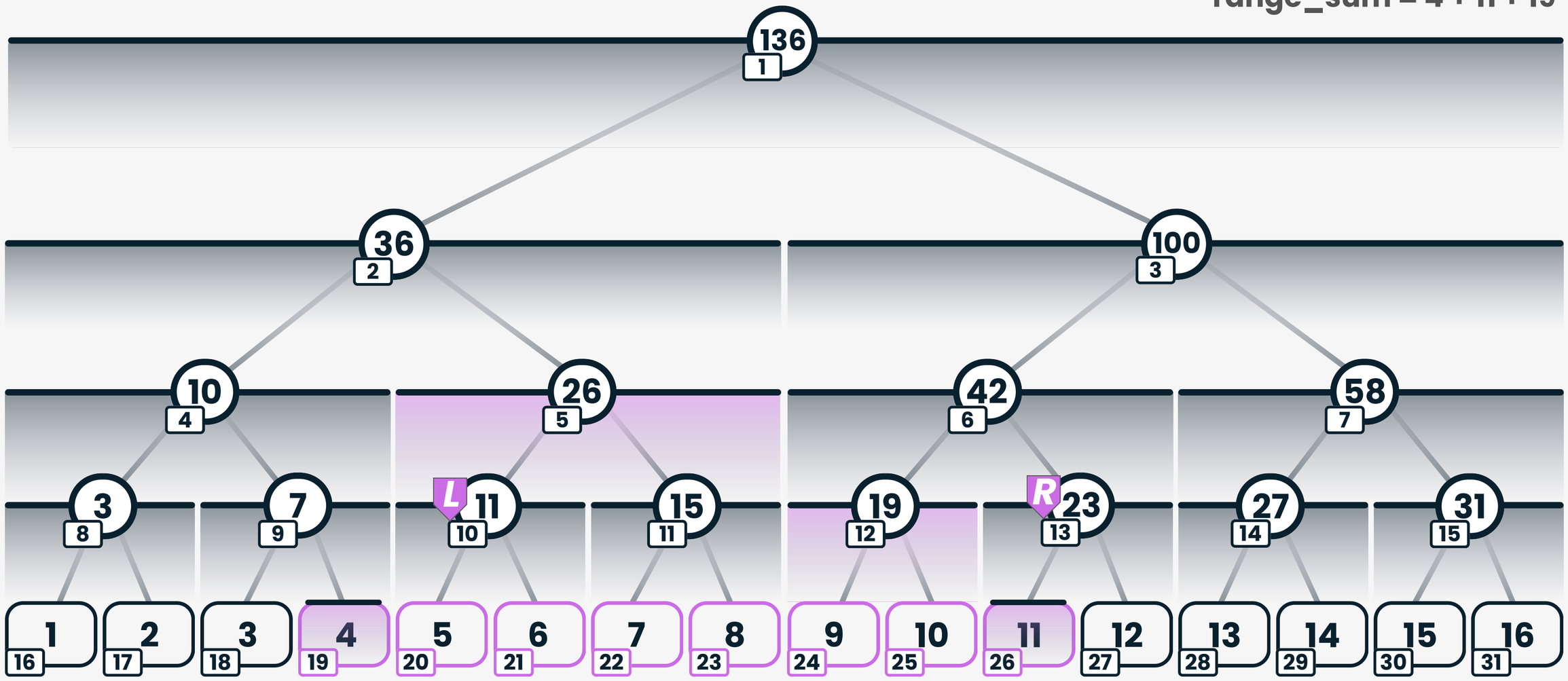
**range_sum = 4 + 11**



# Query

**L** is even, do nothing.
**R** is odd, **R - 1** should be included in the sum.

**range_sum = 4 + 11 + 19**

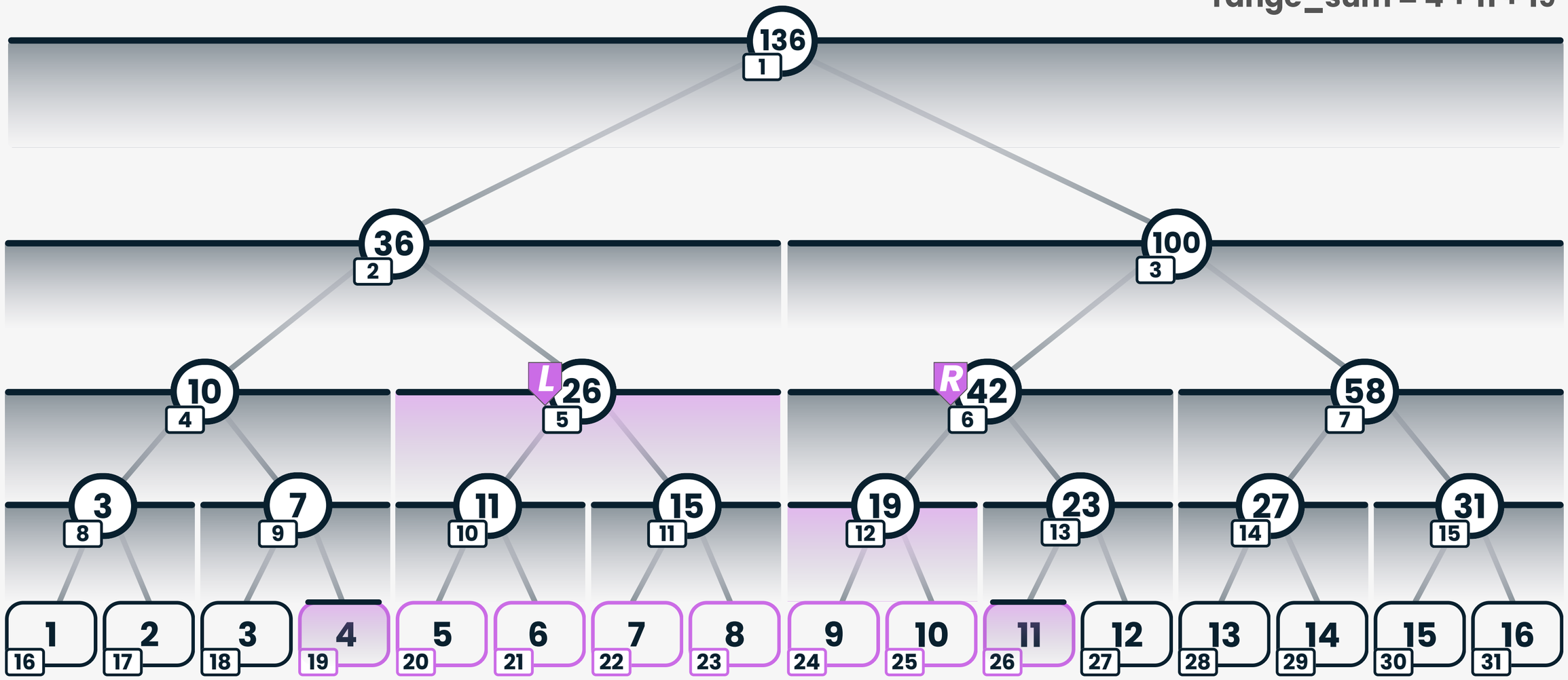


# Query

Value
index

**L** changes to its parent, **L//2**.
**R** changes to **(R - 1)//2**.

**range_sum = 4 + 11 + 19**

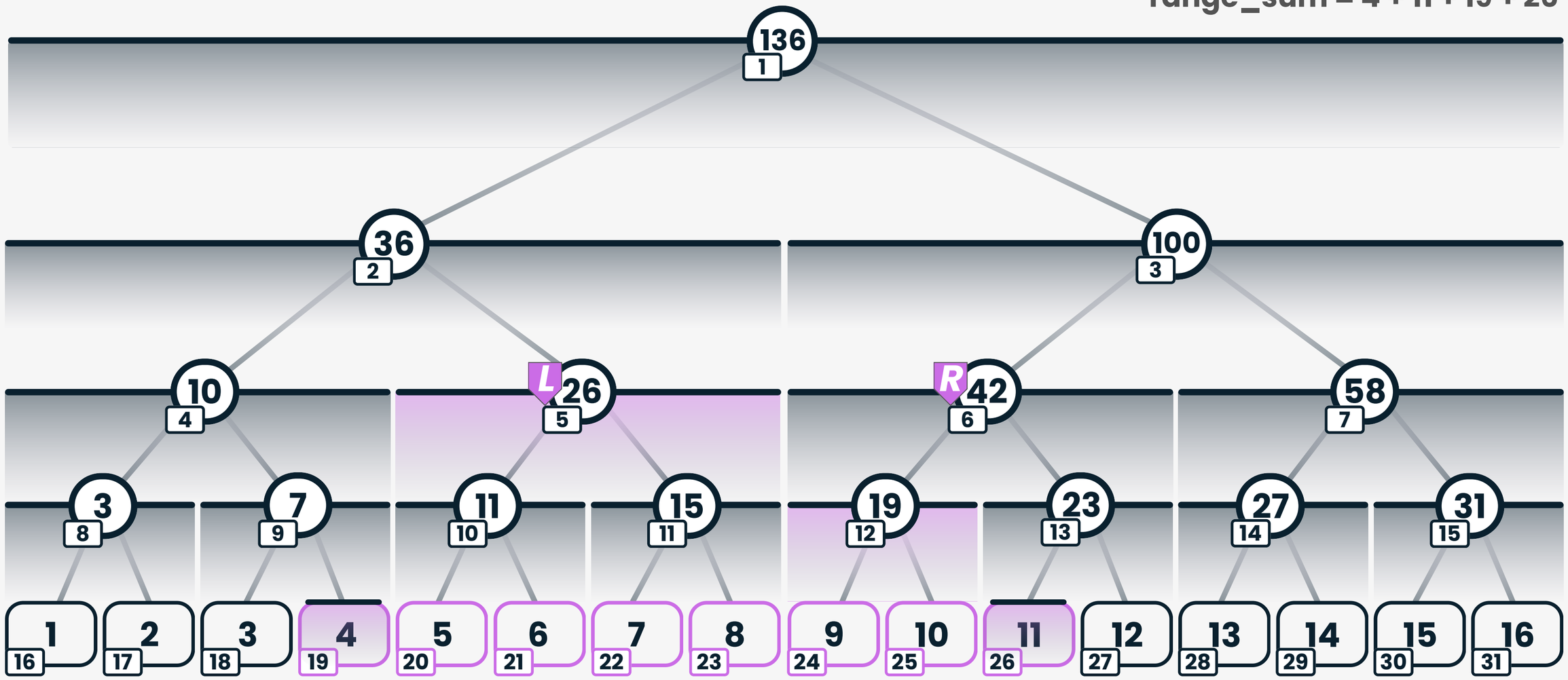


# Query

Value
index

**L** is odd, it should be included in the sum.
**R** is even, do nothing.

range_sum = 4 + 11 + 19 + 26

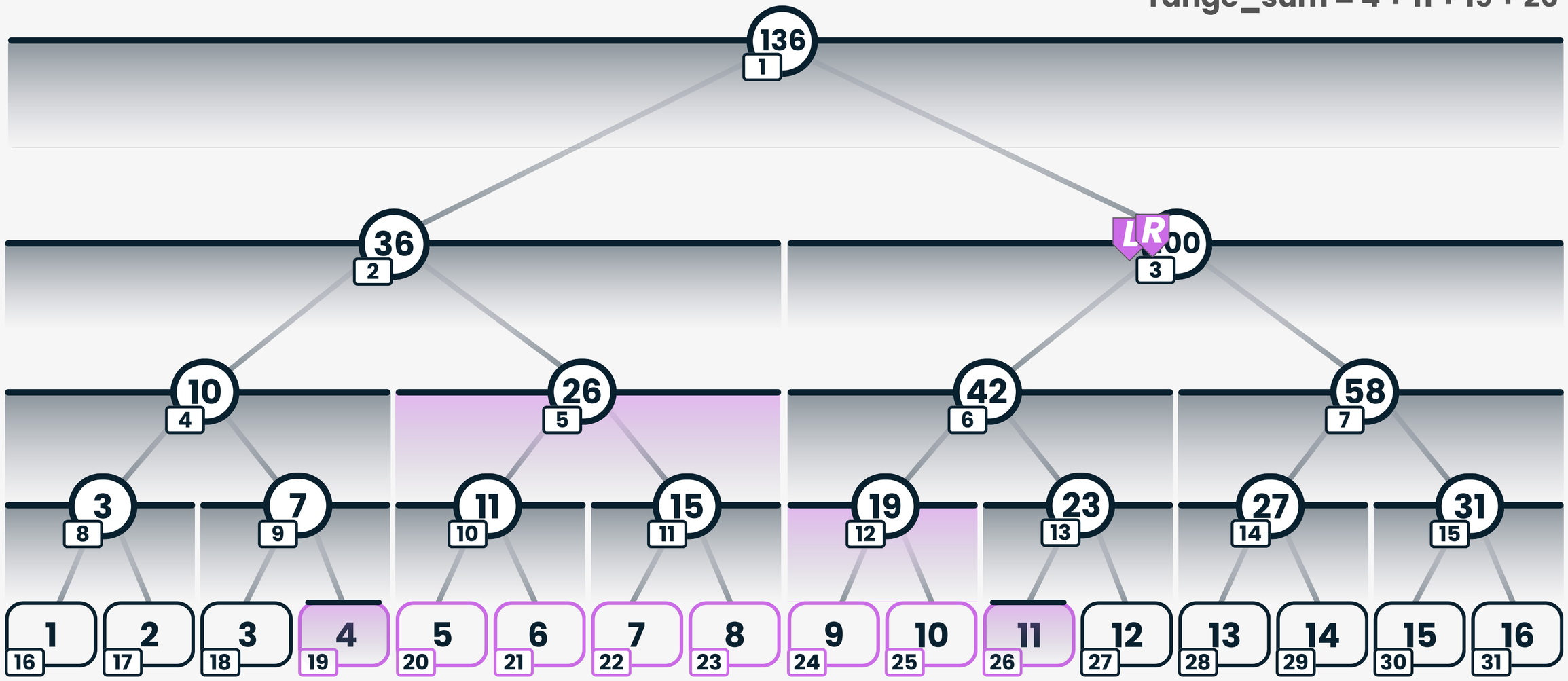


# Query

Value
index

**L** changes **(L + 1)//2**.
**R** changes to its parent, **R//2**.

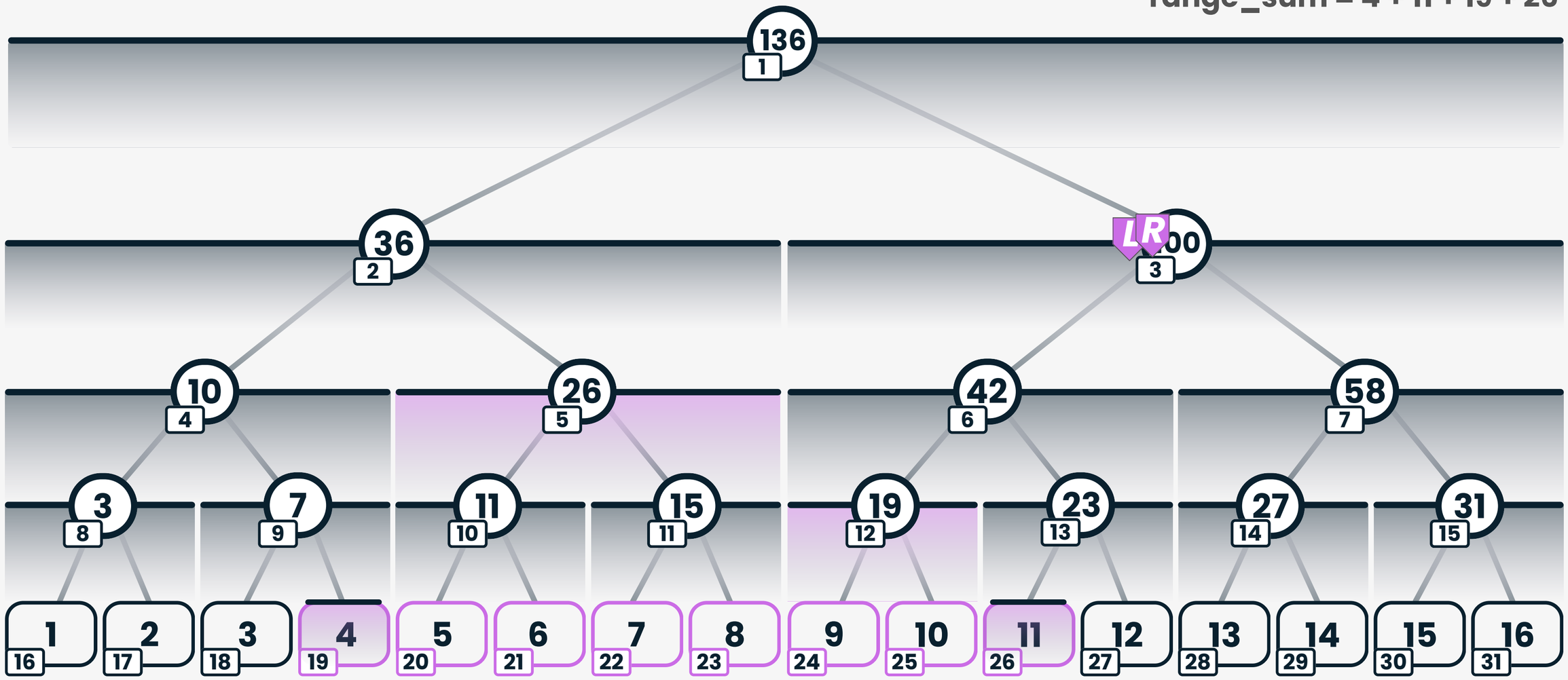
range_sum = 4 + 11 + 19 + 26



# Query

Now, **L is no longer less than R,** We **stop**.
The variable **range_sum** now holds the values of the target nodes.

**range_sum = 4 + 11 + 19 + 26**



# Iterative Implementation

```
def query(self, left, right):
    # The answer is left inclusive and right exclusive.

    left += self.n
    right += self.n
    range_sum = 0

    while left < right:
        if left & 1:
            range_sum += self.tree[left]
            left += 1

        if right & 1:
            right -= 1
            range_sum += self.tree[right]

        left >>= 1
        right >>= 1

    return range_sum
```

# Variants

Segment trees have several variants depending on the types of operations they support.

# Point Update - Range Query Standard Segment Tree

This is the most common type of segment tree, where:

- **Update**: You can update a **single** element in the array.
- **Query**: You can query for information over a **range** of elements.

# Point Update - Range Query Standard Segment Tree

**Practice Problems:**

- Codeforces | Segment Tree for the Minimum
- LeetCode | Longest Increasing Subsequence II

# Range Update - Point Query (Simple Version)

This variant allows you to update a range of elements at once, while queries are performed on single elements (points).

- The simple version is usually applied in operations that allow commutativity and associativity.
  - E.g. Increment Operations
- **Practice Problem**: Addition to Segment

# Range Update - Point Query (Advanced Version)

The advanced version makes it possible to apply a modification (like setting a value or adding a constant) to all elements in a specified range.

- A **lazy propagation** mechanism is used to delay the update until it's needed.

# Range Update - Range Query

This is a powerful variant where you can update an entire range and also query a range of elements.

- A **lazy propagation** mechanism is intensively used here.

# Applications

Segment trees are widely used in scenarios requiring efficient range queries and updates.

# Operations on Segment Trees

Segment trees can handle various operations. For instance,

- Min/Max Operations
- Bitwise Operations (AND/OR/XOR)
- GCD/LCM Queries

# Complex Queries

**Kth Zero with Modifications**:

- A segment tree can store the number of zeros in each segment, enabling efficient updates and queries to locate the K-th zero.

# Complex Queries

**Bisect Operation with Modifications**:

- You may want to search for a position where a condition first holds in a modified array. Segment trees allow for efficient binary search within a range, combined with modification queries.

# More Applications

- Multidimensional Data
- Persistent Data Structures

# More Applications

**Real-life Applications**:

- Range-based Statistics
- Interval Scheduling

# Common Pitfalls

- **Off-by-One Errors**: It's easy to get confused with inclusive and exclusive bounds, when working with ranges.
    - Forgetting the **right** pointer is **exclusive** on the iterative implementation.
- **Improper Size Allocation**: The required memory size can vary depending on the implementation strategy, which can lead to confusion about how much memory should be allocated.

# Alternatives

While segment trees are powerful for range queries and dynamic updates, there are several alternatives with distinct advantages depending on the problem requirements

# Sparse Table

A data structure used for answering range queries (e.g., range minimum or maximum queries) efficiently in **static arrays**.

- **Pros**: Queries are answered in **O(1)** time after **O(n logn)** preprocessing.
- **Cons**: No dynamic updates.

# Fenwick Tree (Binary Indexed Tree)

A data structure that efficiently supports **prefix sum queries** and **point updates**.

- **Pros**: Space-efficient and easier to implement than a segment tree.
- **Cons**:
  - Range updates are more complex and typically require modifications or using two Fenwick Trees.
  - More limited in functionality compared to segment trees.

# Sqrt Decomposition

A technique that divides an array into **blocks** of size **√n** and processes queries or updates by working within blocks.

- **Pros**: Simpler to implement compared to segment trees.
- **Cons**: Both updates and queries take **O(√n)** time. Not ideal for scenarios requiring **very frequent** updates or queries.

# Treap

A randomized binary search tree that maintains both the properties of a **binary search tree (BST)** and a **heap**.

- **Pros**: Supports range queries and updates similar to a segment tree and can handle data **insertion** and **deletion** efficiently.
- **Cons**: Implementation is complex. **Probabilistically** efficient.

# Practice Problems

- Number of Longest Increasing Subsequence
- My Calendar I
- My Calendar II
- Codeforces Edu - Segment Tree Part I
  - Practice Set 1
  - Practice Set 2

# Resources

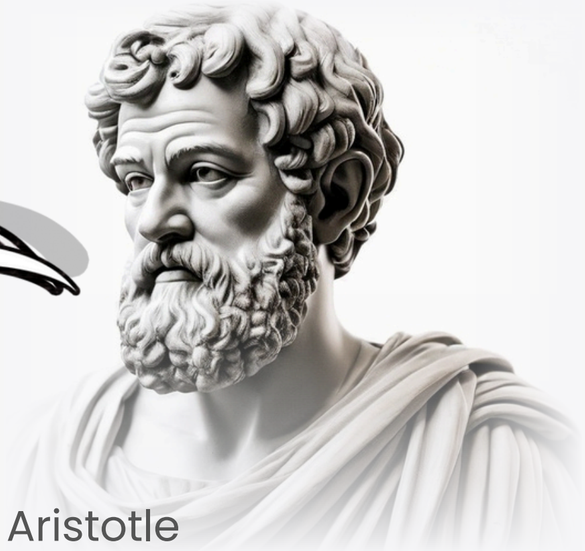
- YouTube: SecondThread
- YouTube: Algorithms Live

# Resources

- Codeforces Edu | Segment Tree Part I
- Codeforces Edu | Segment Tree Part II
- CP Algorithms | Segment Tree
- LeetCode | Segment Tree Article
- Geeksforgeeks | Segment Tree

# Quote of the Day

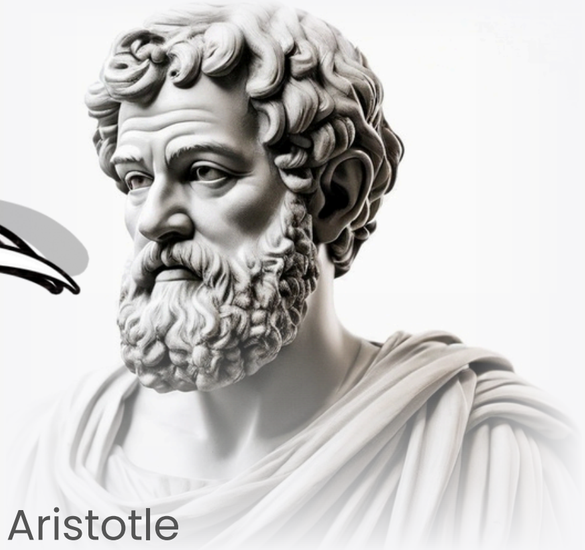
The whole is greater than the sum of its parts.



Aristotle

# Quote of the Day

The whole is greater than the sum of its parts.



Aristotle

# Thank You

2024