香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Data Structures (IV)

Isaac Wong {WongChun1234}
2026-03-07

Agenda

- Trie
- Lazy Propagation on Segment Tree
- Sweep Line Trick
- Binary Search on Segment Tree / Segment Tree Descent
- Persistent Segment Tree

Prerequisite

- Segment Tree
- Will be taught in Data Structure (III)

Task List

Trie:

- IOI08 Type Printer

Lazy Propagation:

- T192 Colourful Strips
- T213 Game Developer
- T253 Peaceful Pirate Pairs
- CF446C DZY Loves Fibonacci Numbers

Sweep Line

- NOI23 方格染色
- APIO18 New Home

Persistent Segment Tree:

- M1842 Another RMQ
- APIO17 Land of the Rainbow Gold
- NOI18 歸程
- P4735 最大異或和 (Persistent trie!)
- P2617 Dynamic Rankings

Trie

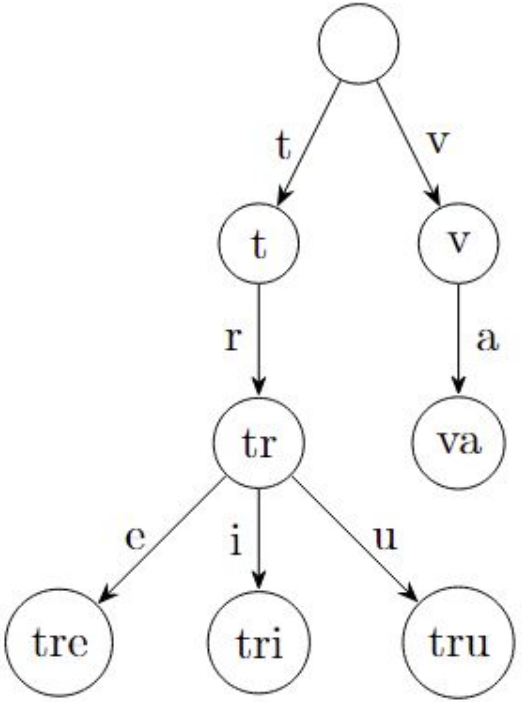
Not a formal word

- come from the word re**trie**val
- Pronounce as “try”

Tree data structure

Dictionary of a set of strings

- support quick insert and lookup
- searching/counting strings/prefixes

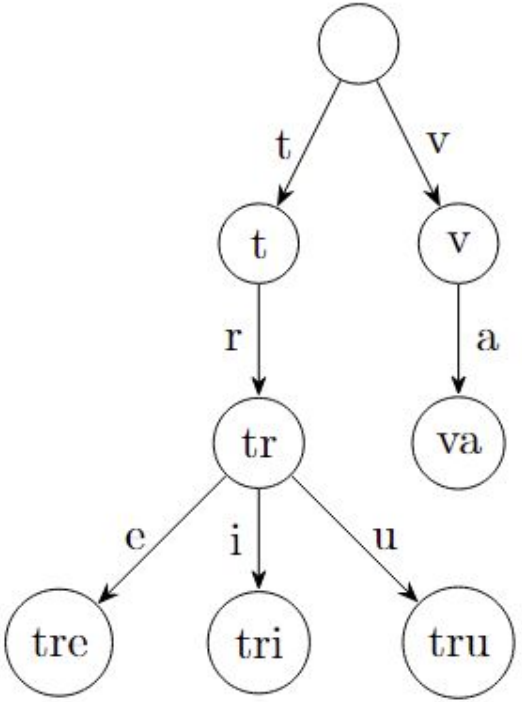


Trie

Edge: addition of a character

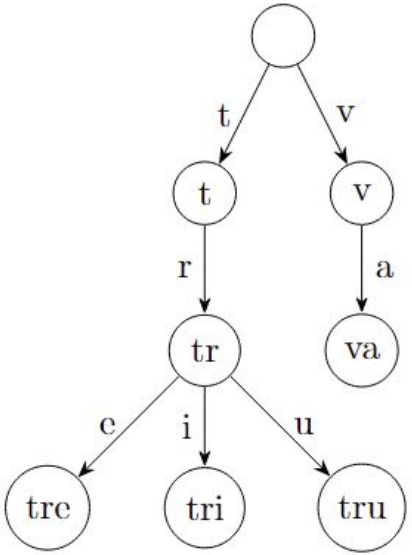
Node: a string

- obtained by concatenating characters from root to this node



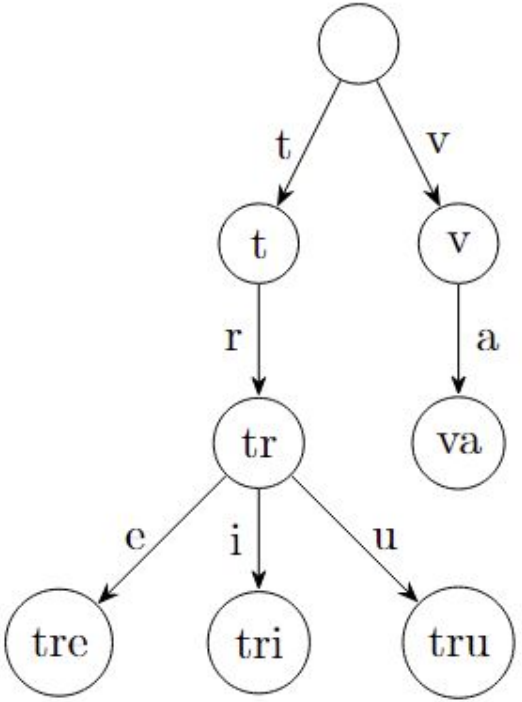
Trie

- A tree data structure (usually) used for string processing
- Supports O(|S|) insertion and prefix matching
- Each node represents a string
- Each edge represents the addition of a character at the end



Trie: Node Structure

- Each node S will have at most ALPHABET_SIZE ∑ out-edges, with edge t pointing to the string S+t (concatenate)
- We will also store some extra information depending on the problem
- Useful information to store in a node:
  - Is this the end of a word?
  - How many strings inserted in the trie has this string as a prefix?

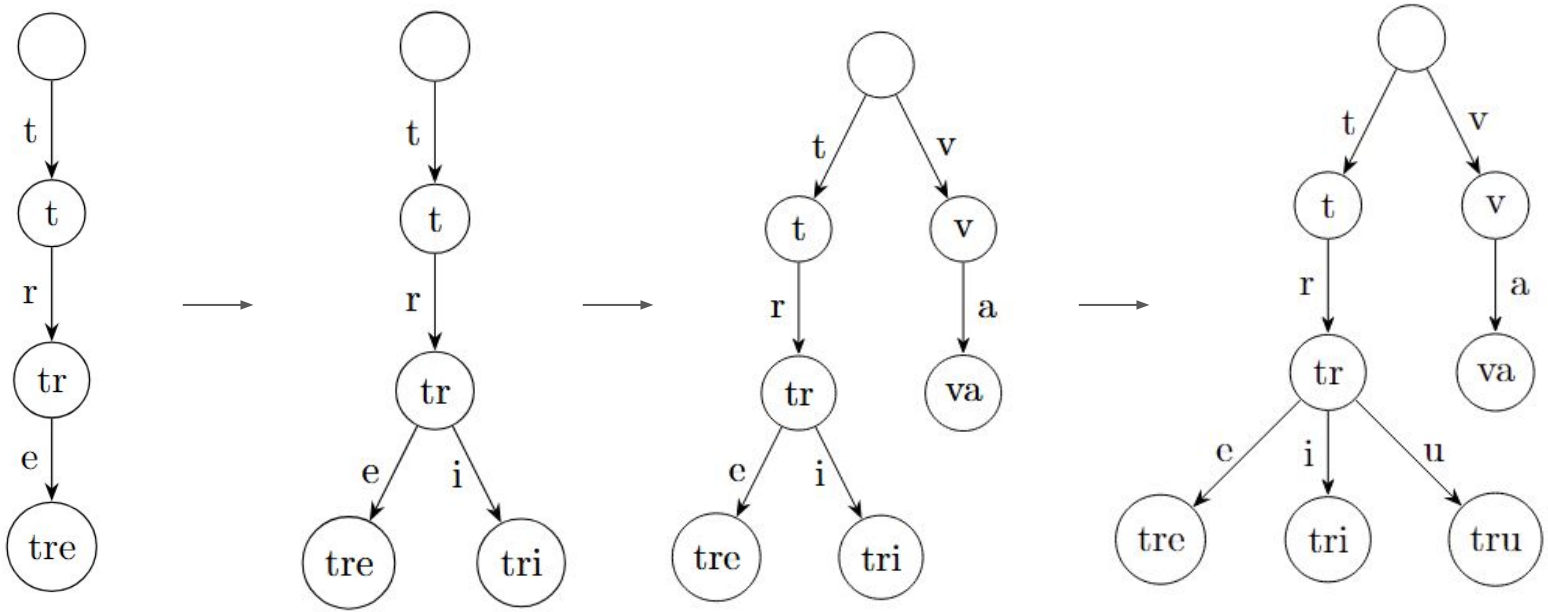


Trie: Insertion

- To insert a string into the trie, we go through the trie according to its prefixes
- Let us be on the node representing a string S and the next character is t
- If the node S+t does not exist, we create a new node and add an edge from S to S+t
- Otherwise, we can just travel to the node S+t
- Remember to update all relevant values during the insert operation!

Trie: Insertion

insert("tre") insert("tri") insert("va") insert("tru")



Trie: Insertion Implementation

Sample code

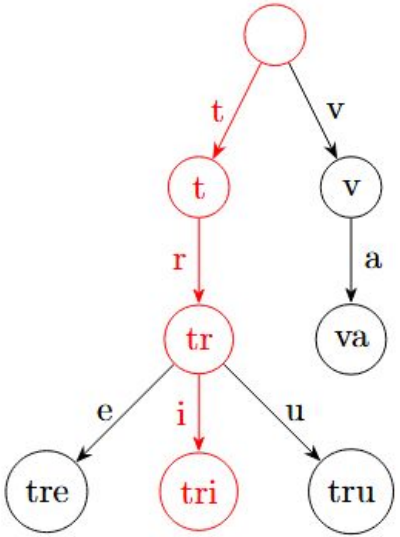
```
insert(s):
    curr = 0
    for (i in s):
        if (!nxt[curr][i - 'a']):
            nxt[curr][i - 'a'] = ++pt
        curr = nxt[curr][i - 'a']
    end[curr] = true
```

Trie: Prefix Query

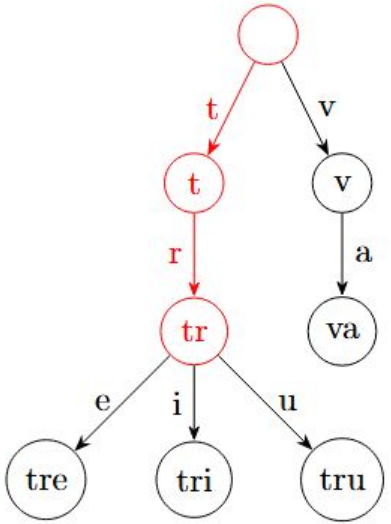
- Similarly, we traverse the trie according to the query string using DFS
- Let us be on the node representing a string S and the next character is t
- If the node S+t exists, we continue on through the trie
- Otherwise, we know that S is the longest common prefix of the query string with all strings

Trie: Prefix Query

query("tri")



query("tra")



Trie: Prefix Query

Sample code

```
query(s):
    curr = 0
    for (i in s):
        if (!nxt[curr][i - 'a']):
            return false
        curr = nxt[curr][i - 'a']
```

```
return end[curr]
//check if s is in dictionary
```

or

```
return true
//check if s is prefix of
any word in dictionary
```

Trie: Application

- Prefix matching is useful in questions involving strings, but can we use trie outside of string algorithm tasks?
- Example: A100 Trie: Maximum Xor
  - Given an array of N integers A_1, A_2, ... A_N, find the maximum value of A_l xor A_r for all pairs of values $l < r$

Trie: Maximum Xor

- Recall how do we maximize xor sum: we prioritize having the more significant bits being different
  - The contribution of a bit will be larger than the sum of all bits that are less significant than it
- For each index r, we can try and match an index $0 \le i < r$ such that A_i and $\sim A_r$ (the inverse of A_r) have the longest common prefix when represented in binary
- Naively implementing will take $O(N^2)$, which will TLE
- We have **insert** and **prefix query** operations – we can use a trie to speed it up

Trie: Maximum Xor

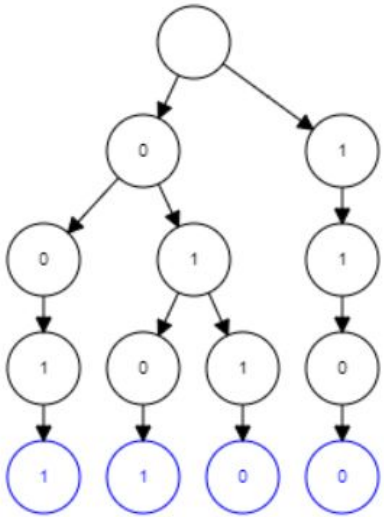
- For each index r, we can perform a prefix query of $\sim A_r$ and insert A_r into the trie afterwards
- The final answer will be the maximum across all indices $1 \le r \le N$
- This will yield a time complexity of $O(N \log \text{Max}(A_i))$
- You can try to implement it in A100 Trie: Maximum Xor

Trie: Maximum Xor

Insert all integers in **A** as a binary string from the most significant bit

- Right figure:
  Trie for **A** = {0101_2, 1100_2, 0011_2, 0110_2}

- Query:
  Idea is to greedily traverse in the direction of
  $\sim A_r$

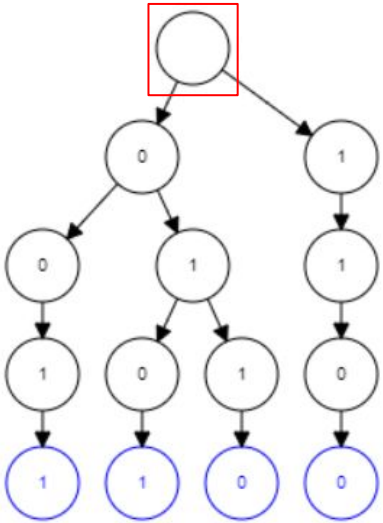


Trie: Maximum Xor

Query(~1011_2)

Current node (in **red**) : ε

Traverse to 0

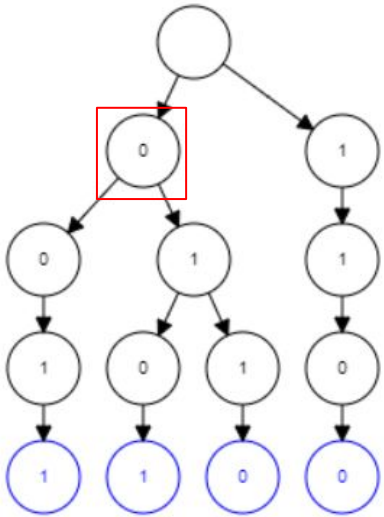


Trie: Maximum Xor

Query(~1011_2)

Current node (in **red**) : 0

Traverse to 01

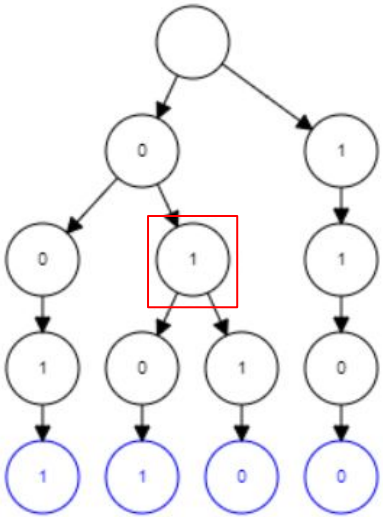


Trie: Maximum Xor

Query(~1011_2)

Current node (in **red**) : 01

Traverse to 010

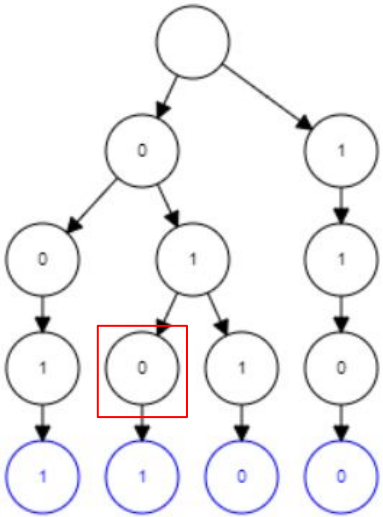


Trie: Maximum Xor

Query(~1011_2)

Current node (in **red**) : 010

Edge 0 does not exist, forced to traverse to 0101



Trie: Maximum Xor

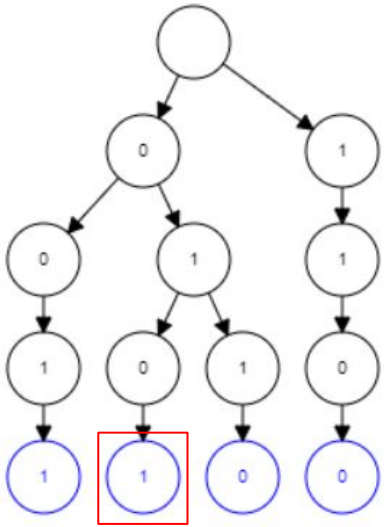
Query(~1011_2)

Current node (in **red**) : 0101

Reached end, max-xor = 1011_2 xor 0101_2 = 1110_2

Many bitwise problems can be done similarly

⇒ **01-trie**: trie where the alphabet ∑ = {0, 1}



Trie: Practice Tasks

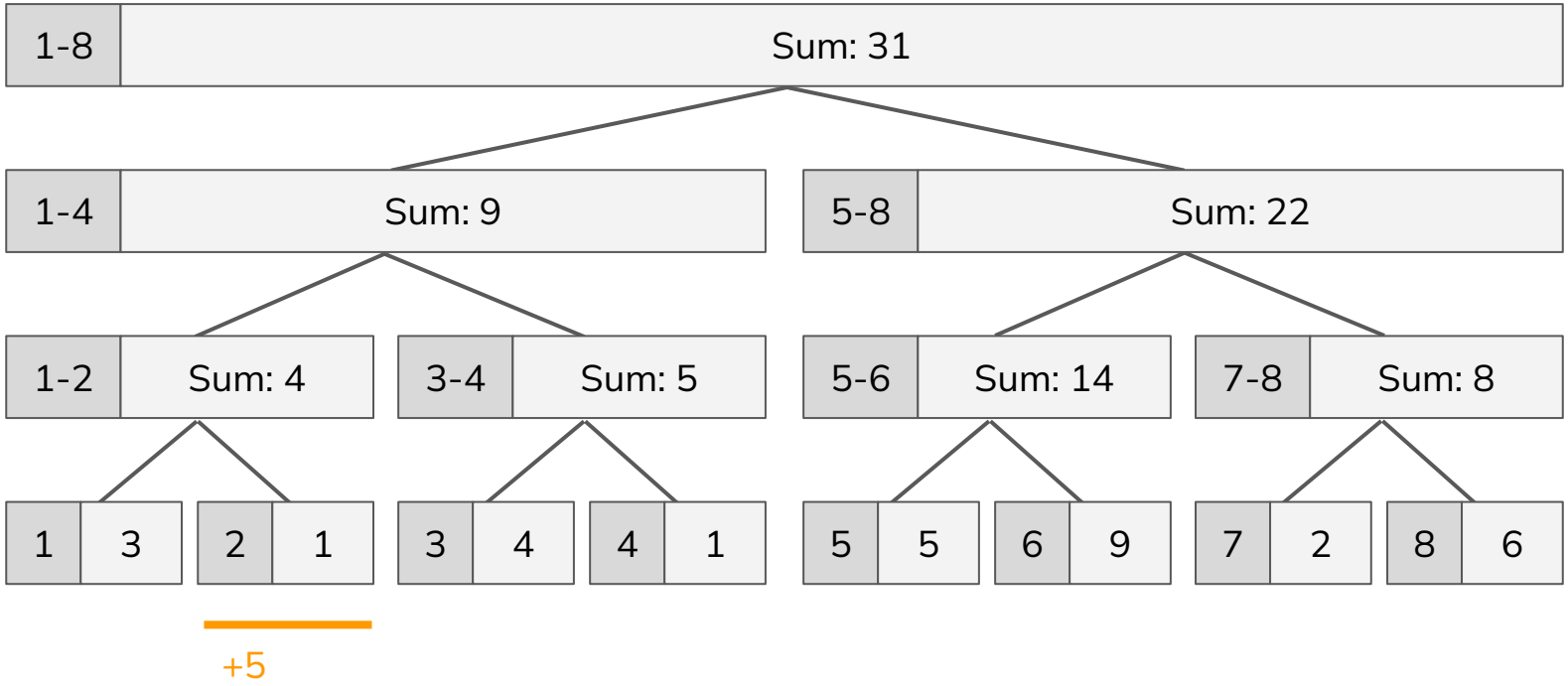
- A100 Trie: Maximum Xor
- IOI08 Type Printer

Before we continue…

- Let's do some revision on range query structures!
- Range query structures are data structures that allows us to query the min/max/sum/... of values in a range efficiently (usually in O(1)/O(log N) time)
- Some range query structures such as **Segment Tree** and **Fenwick Tree** also supports point updates

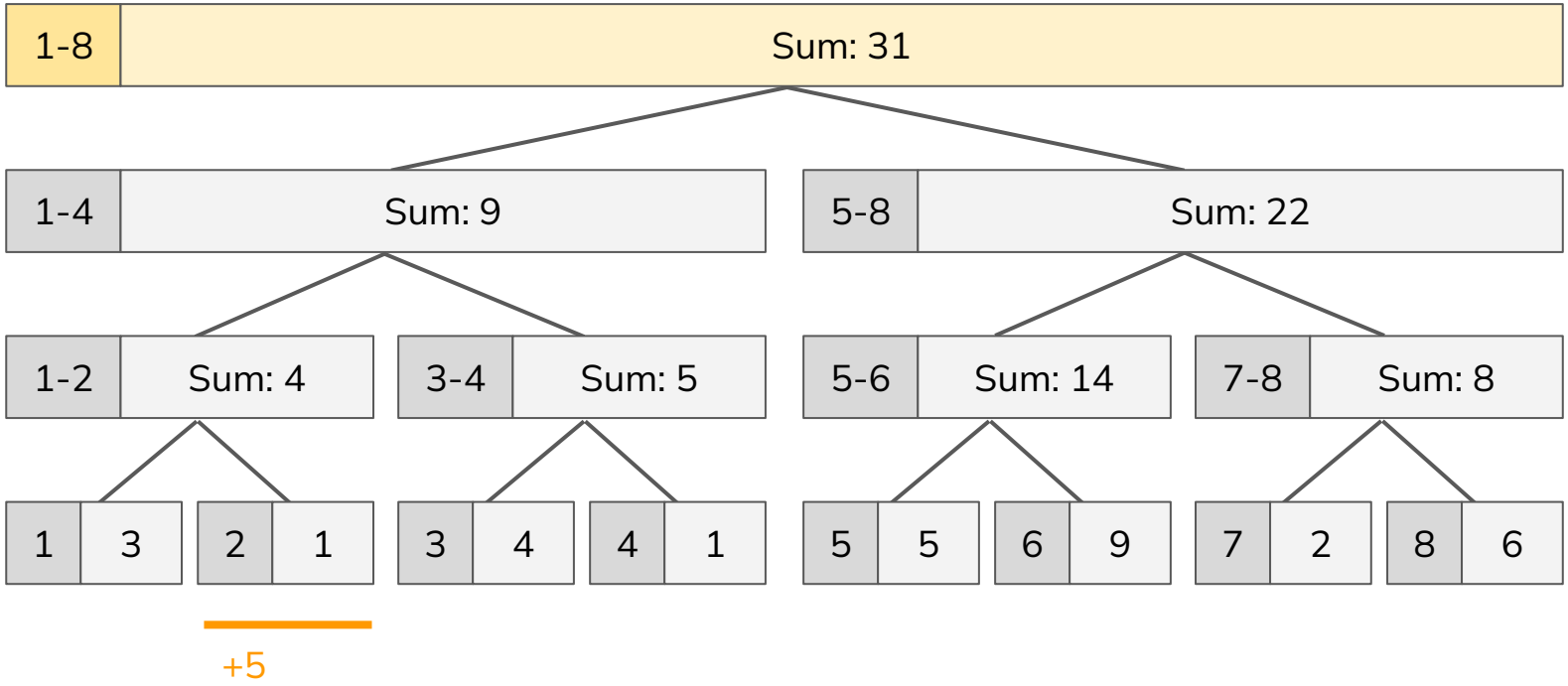
- Let's do a point update on segment tree as warm up

Point Update

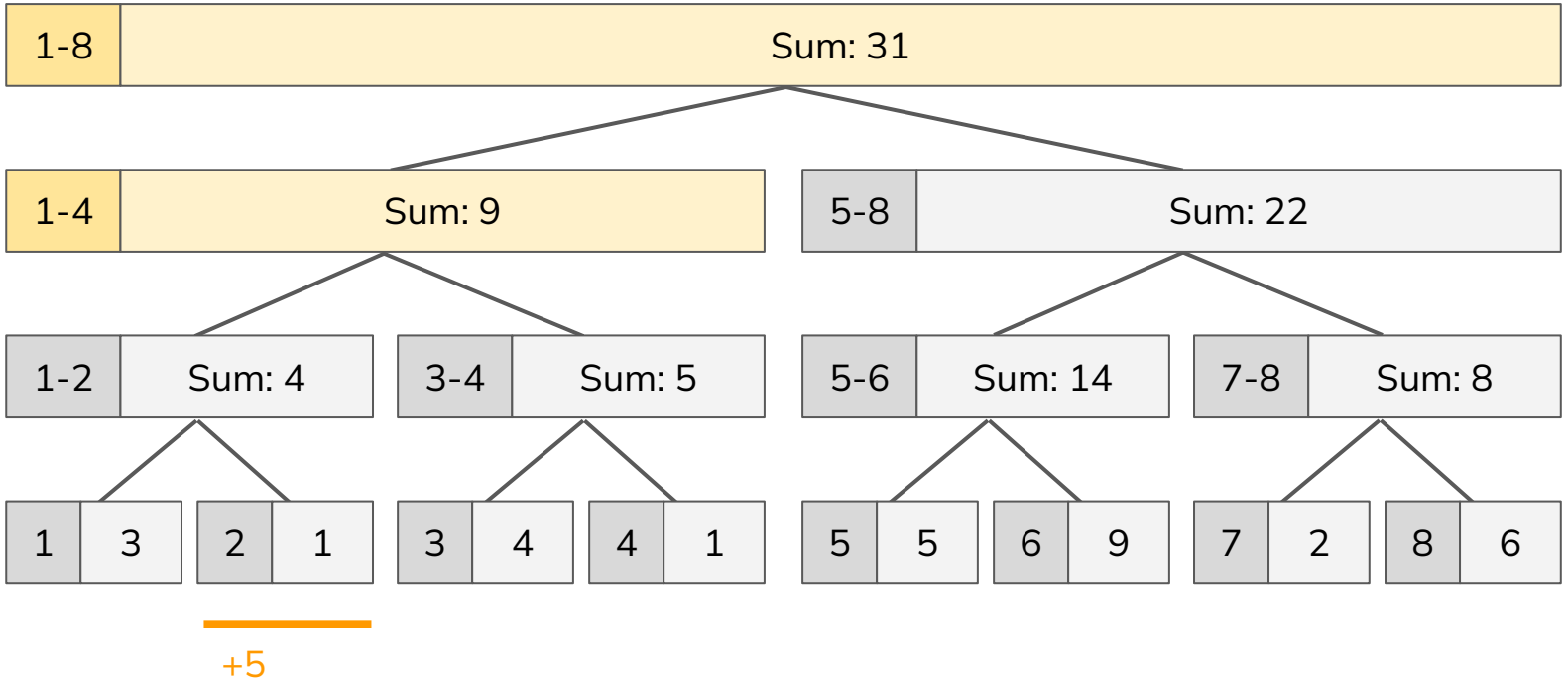


Point Update

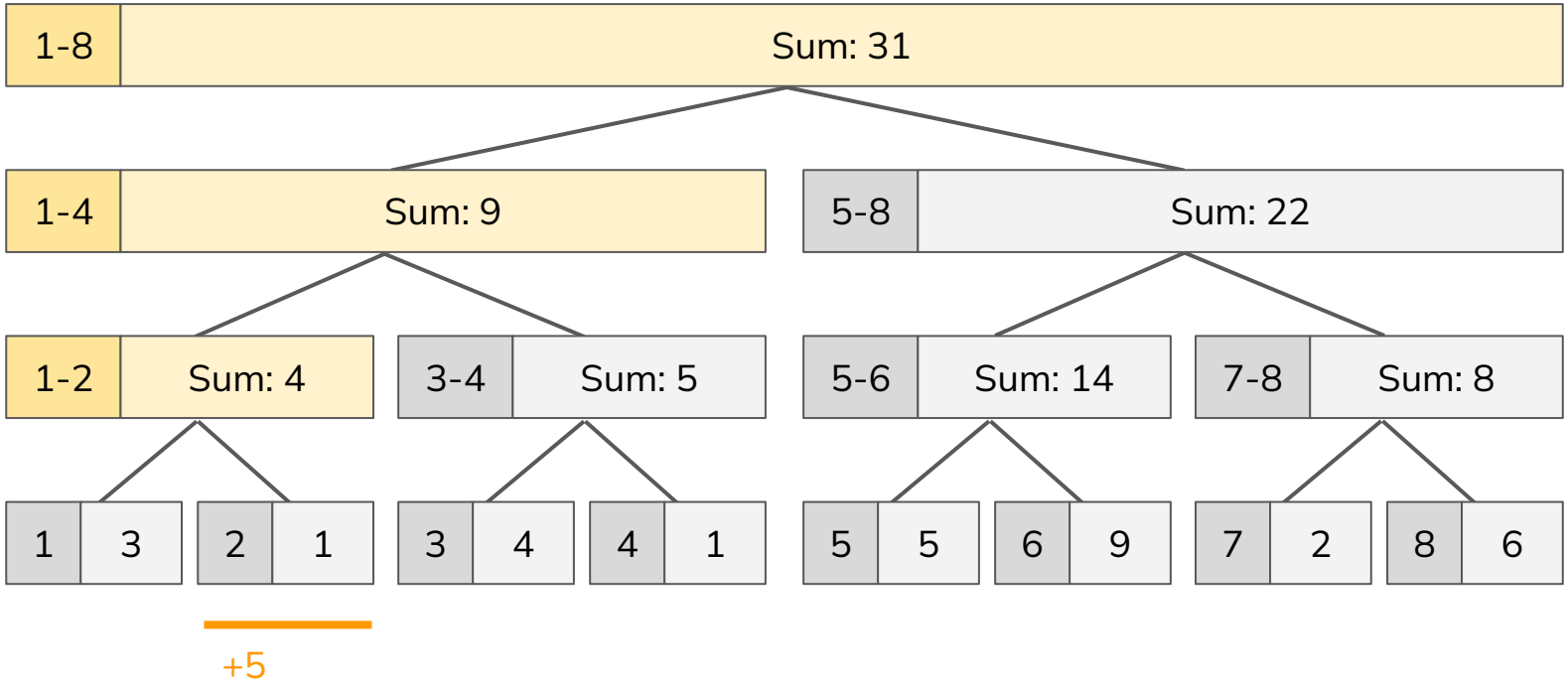
Yellow nodes: all nodes we modified



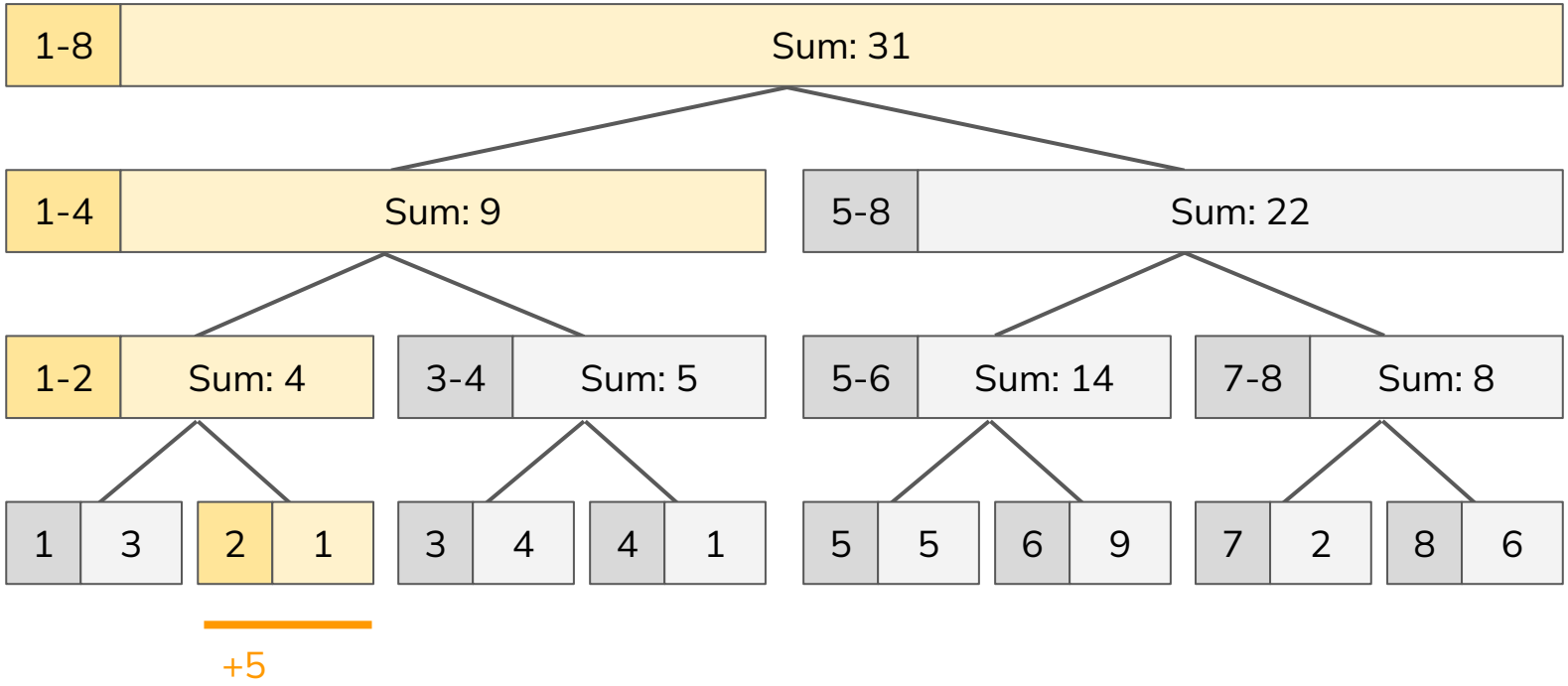
Point Update



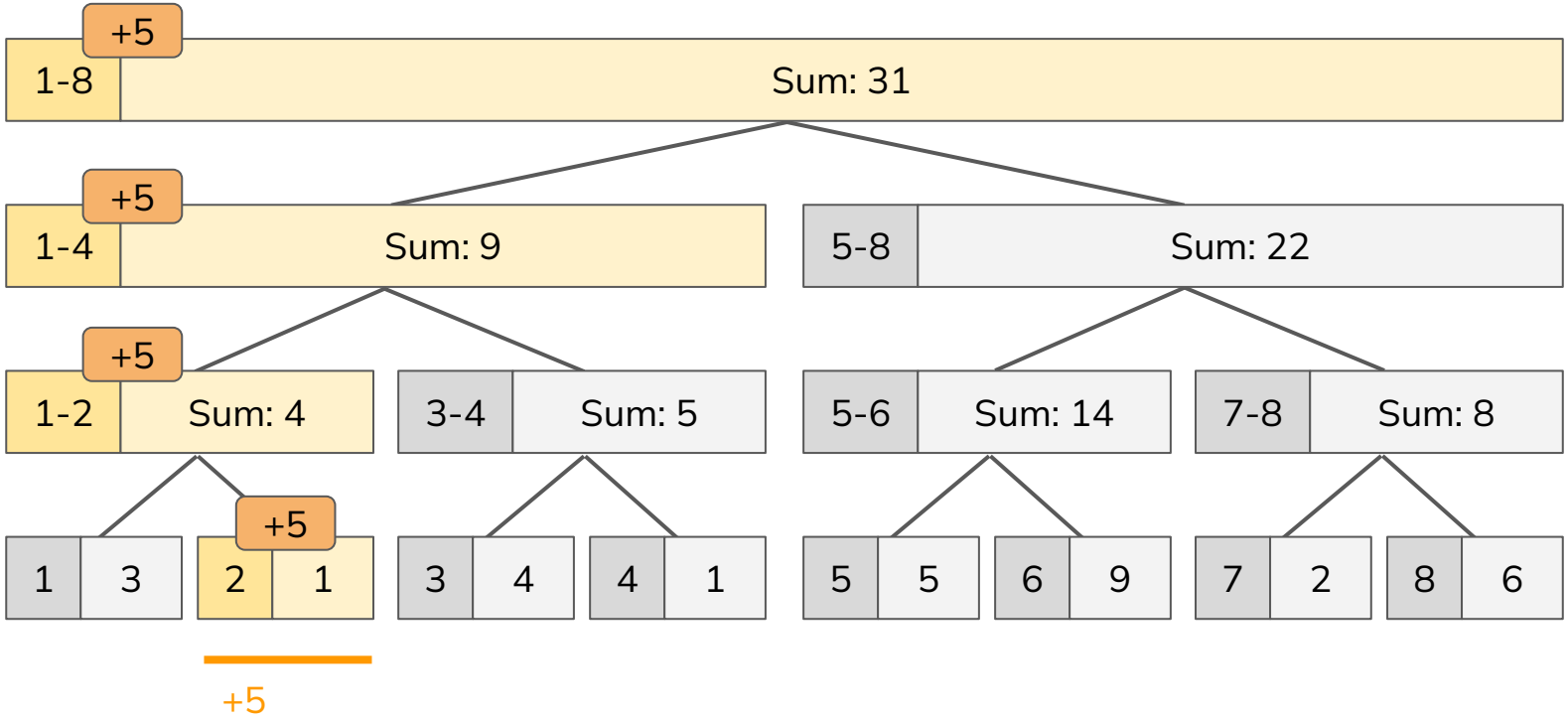
Point Update



Point Update

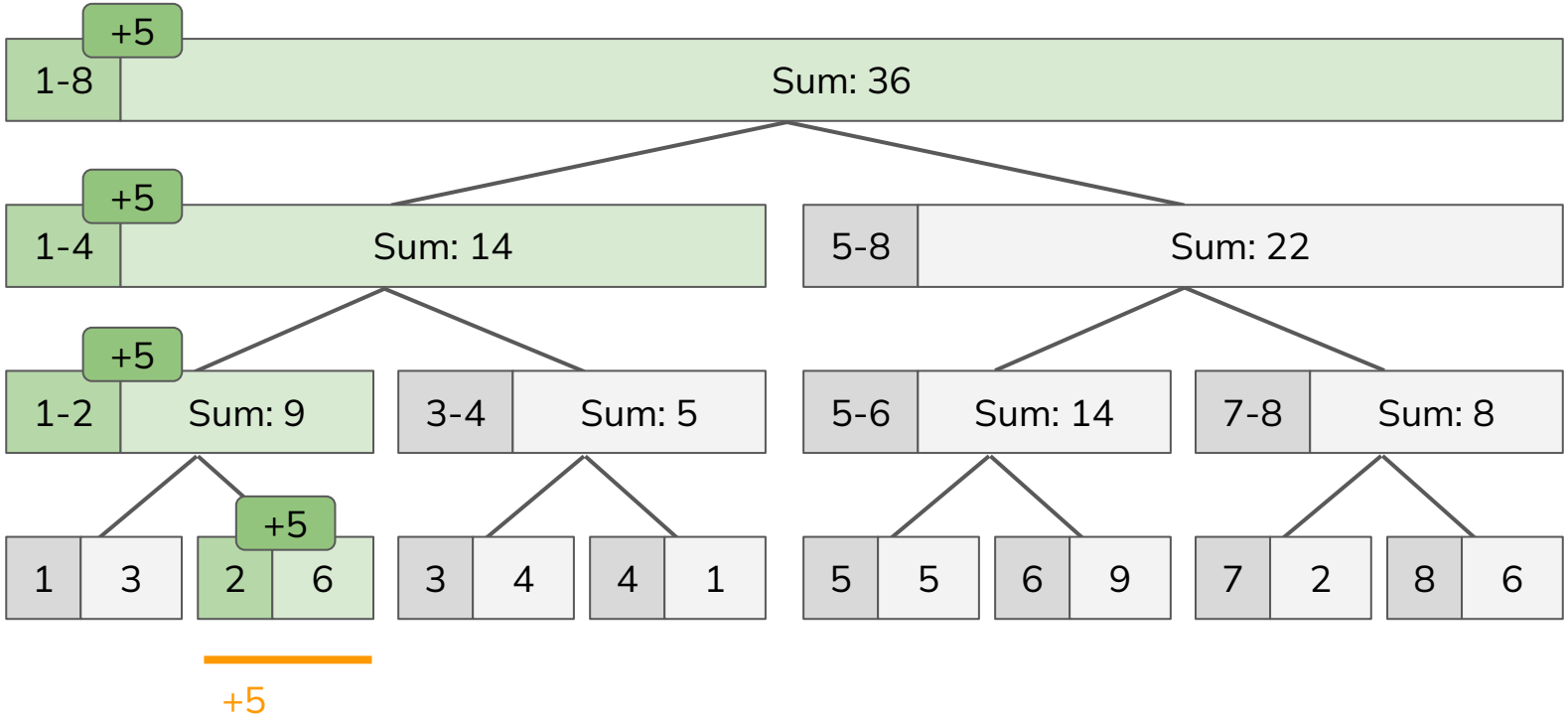


Point Update



Point Update

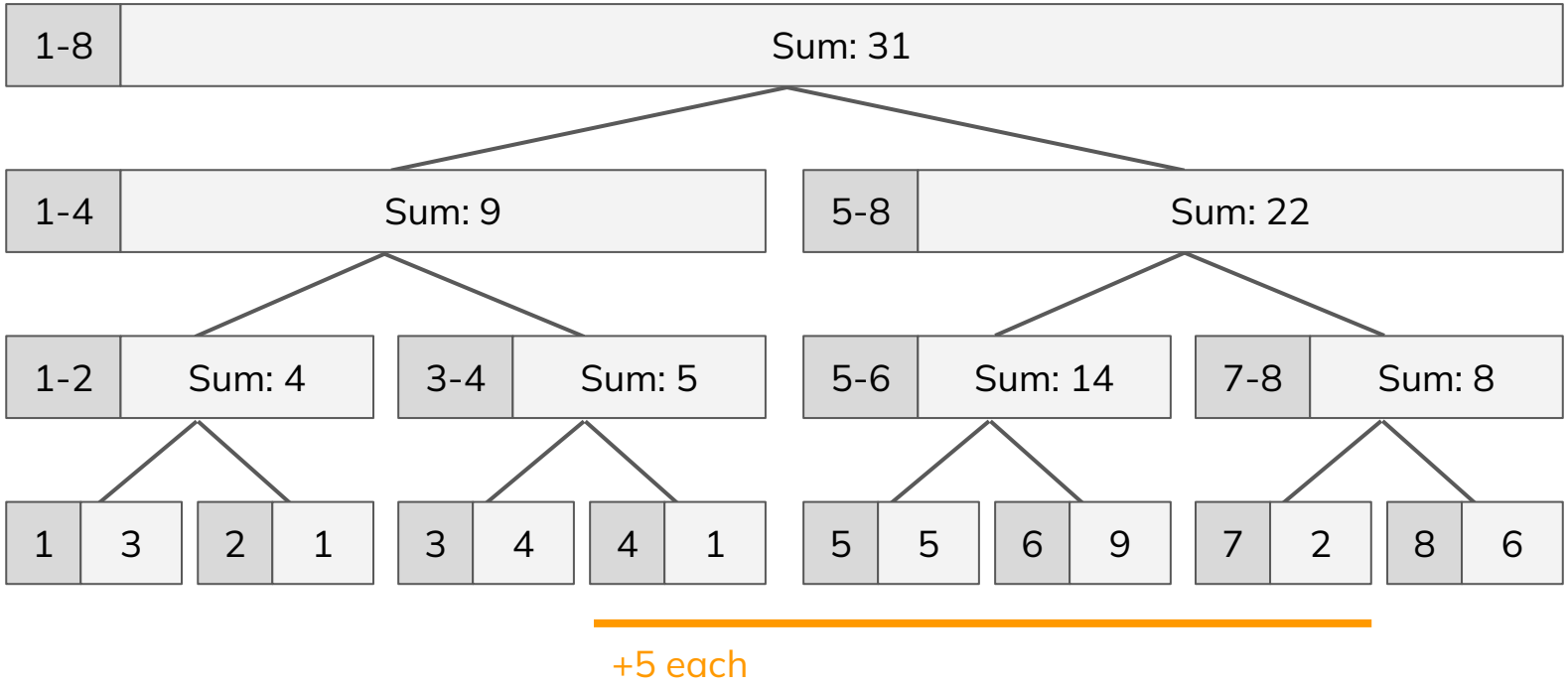
Green nodes: all nodes/ranges we accessed



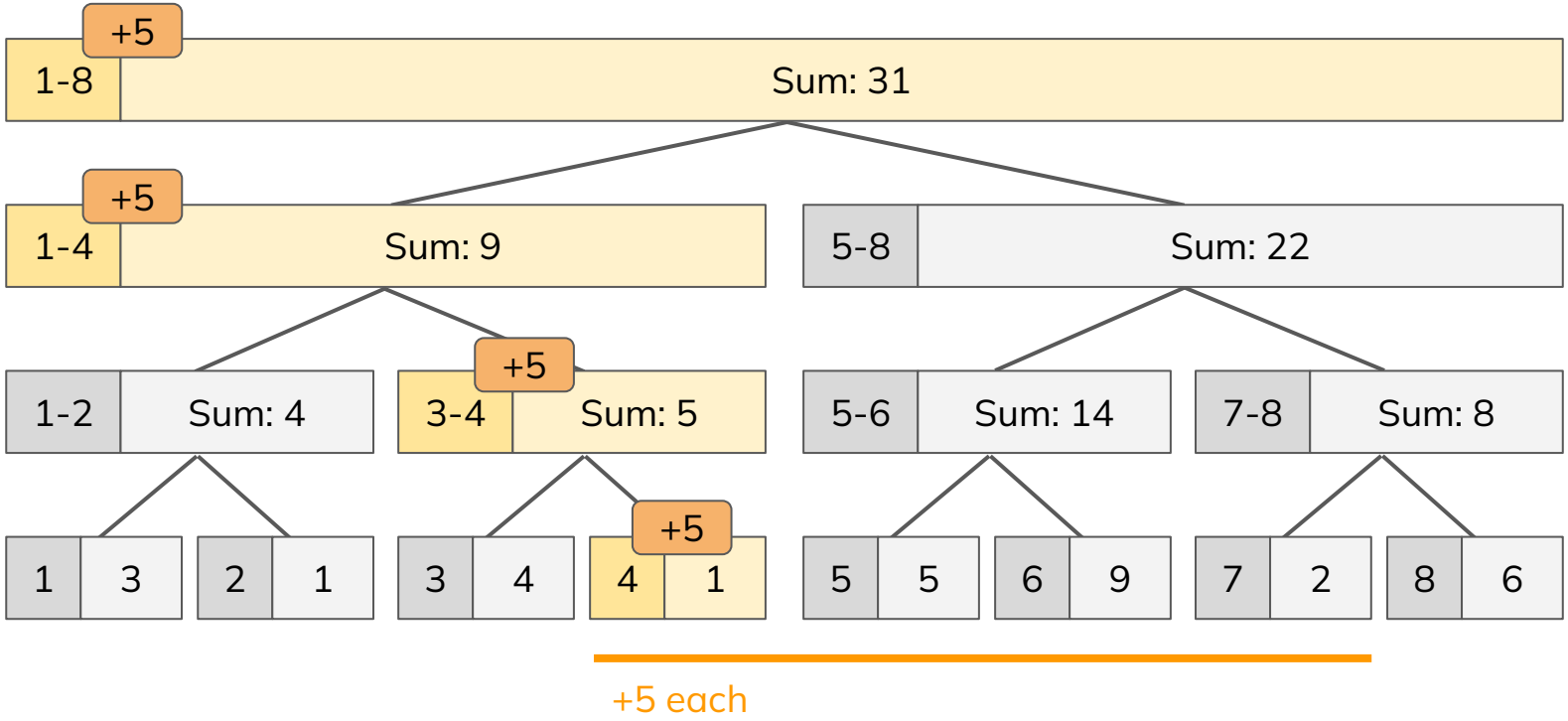
Range Update: Naive Solution

- We have used segment trees to solve point update, range query problems
- What if we want to update values in a range?
- The most intuitive solution is to perform point updates for each index in the range

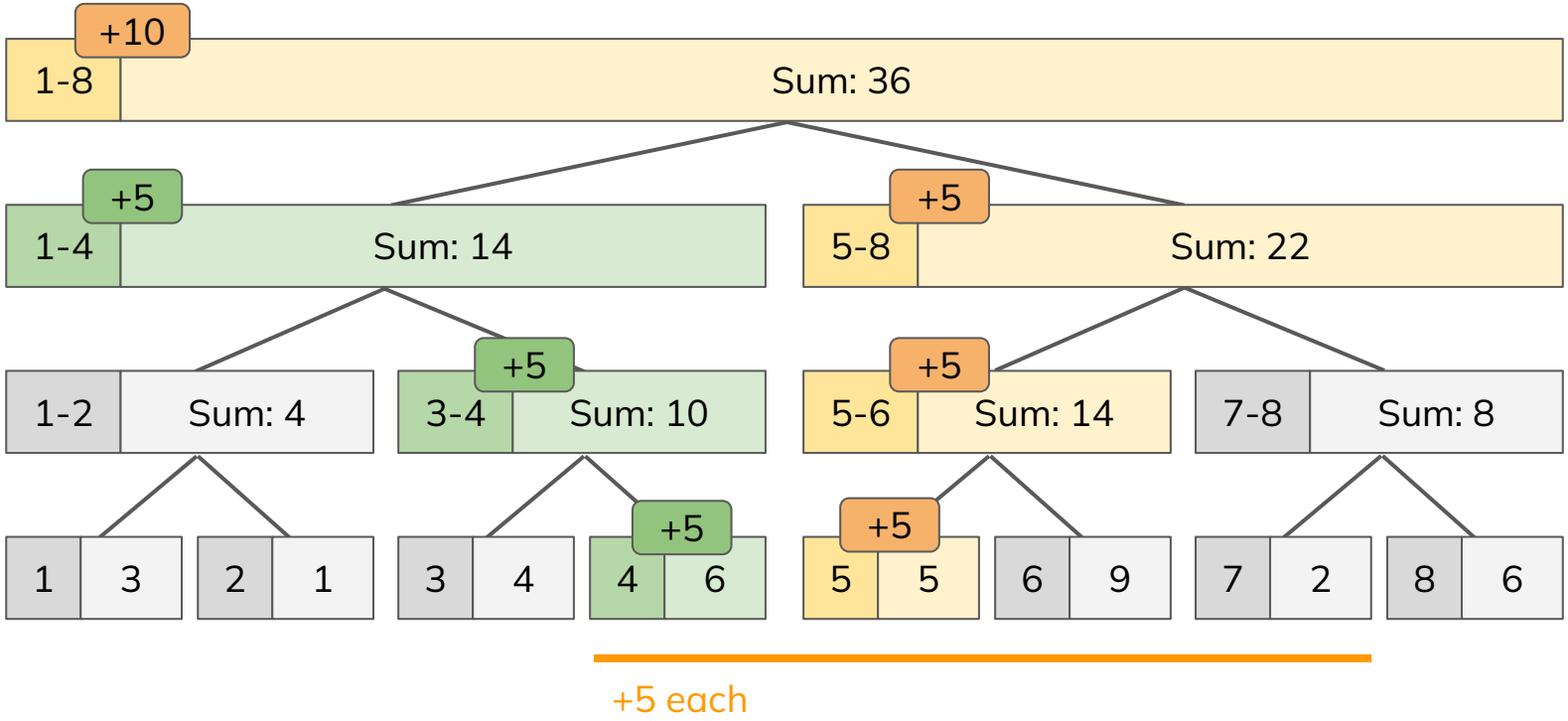
Range Update: Naive Solution



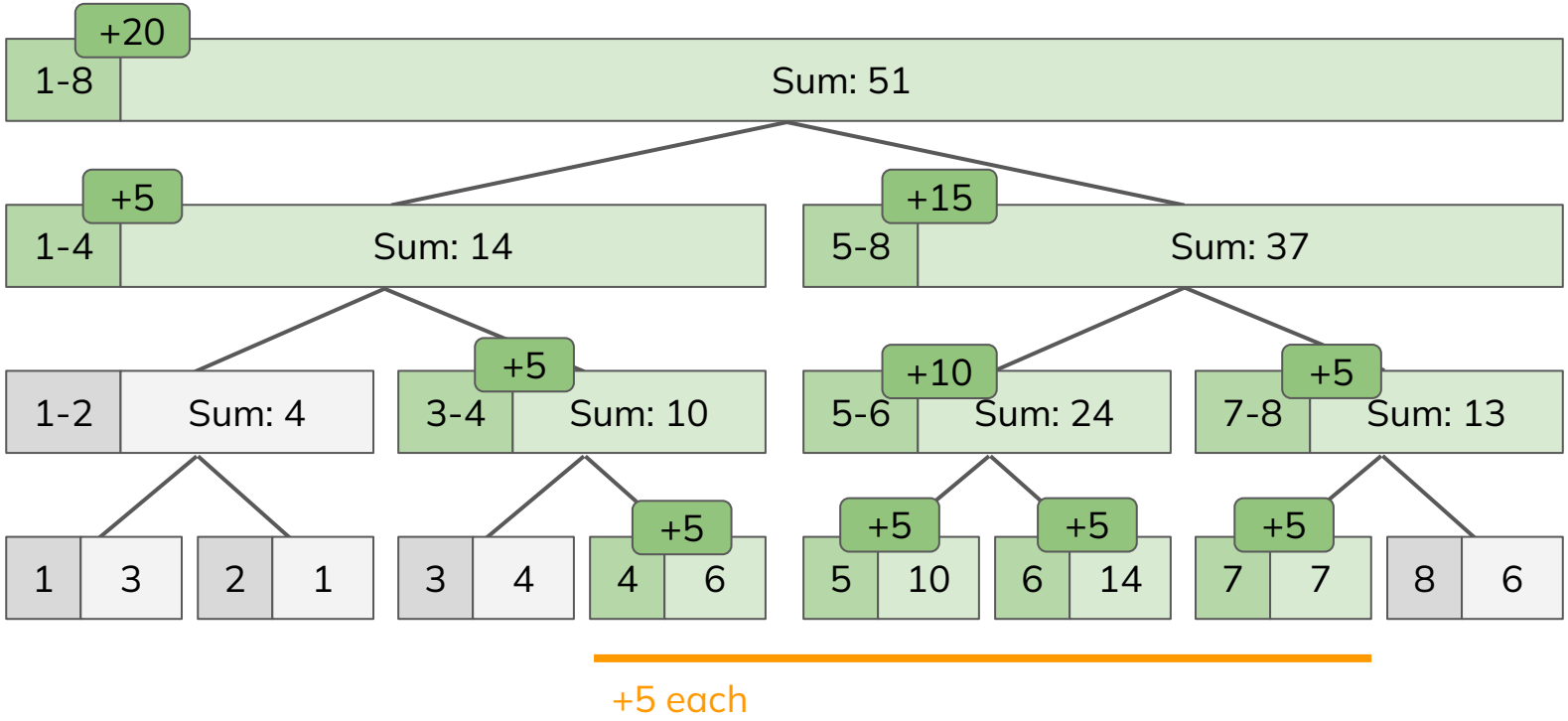
Range Update: Naive Solution



Range Update: Naive Solution



Range Update: Naive Solution

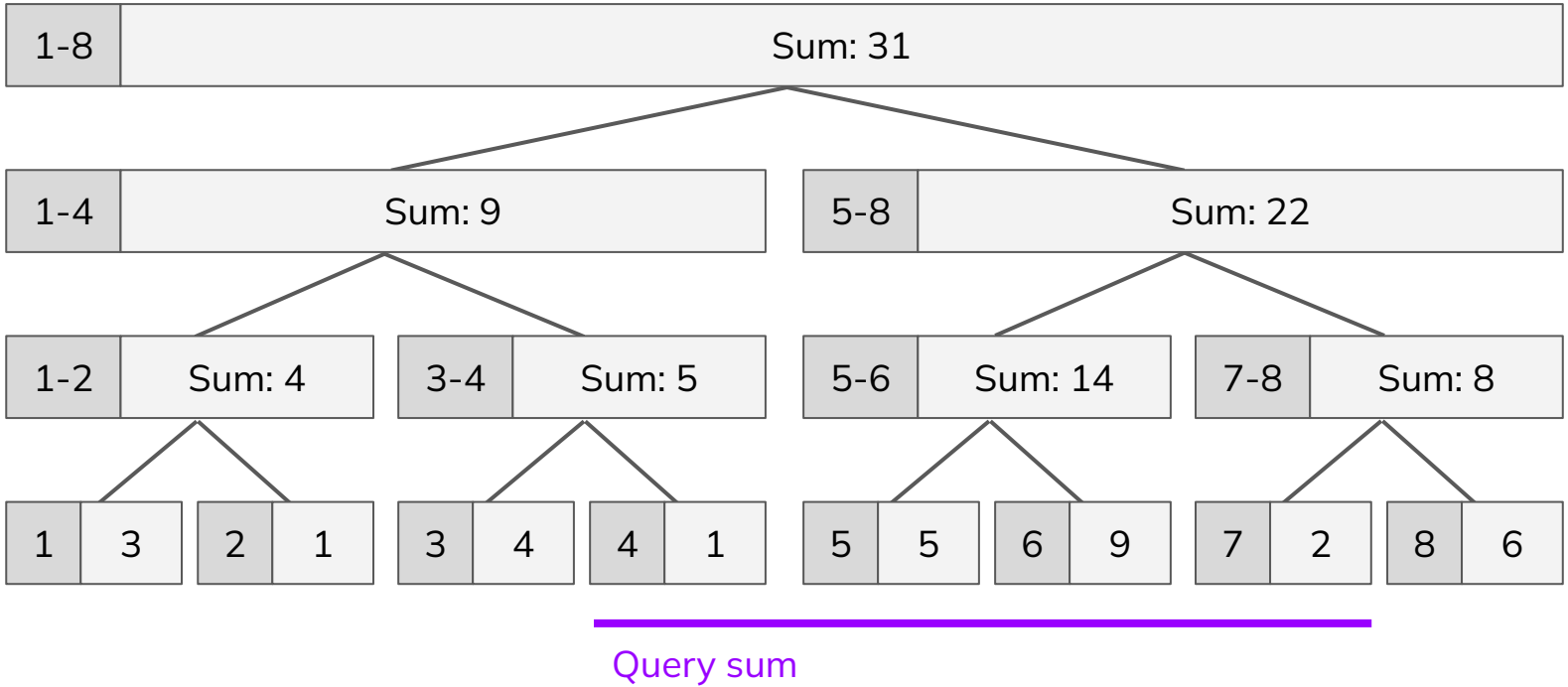


Range Update: Naive Solution

- This will, in worst case, perform on O(N log N) per update
  - O(N) nodes and O(log N) for each update
- Might work on smaller values of N
- Can we do better?

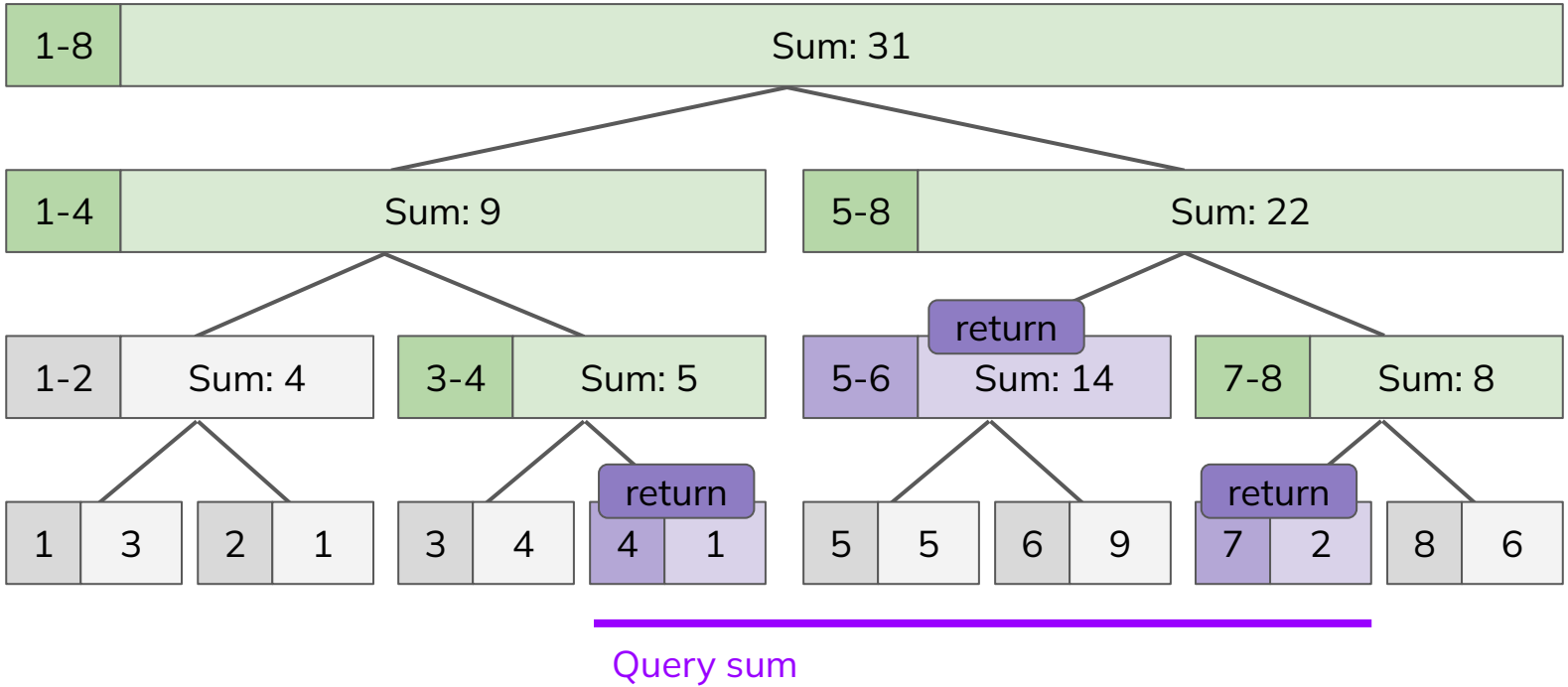
- Let's go back to range query instead of range update

Range… query?

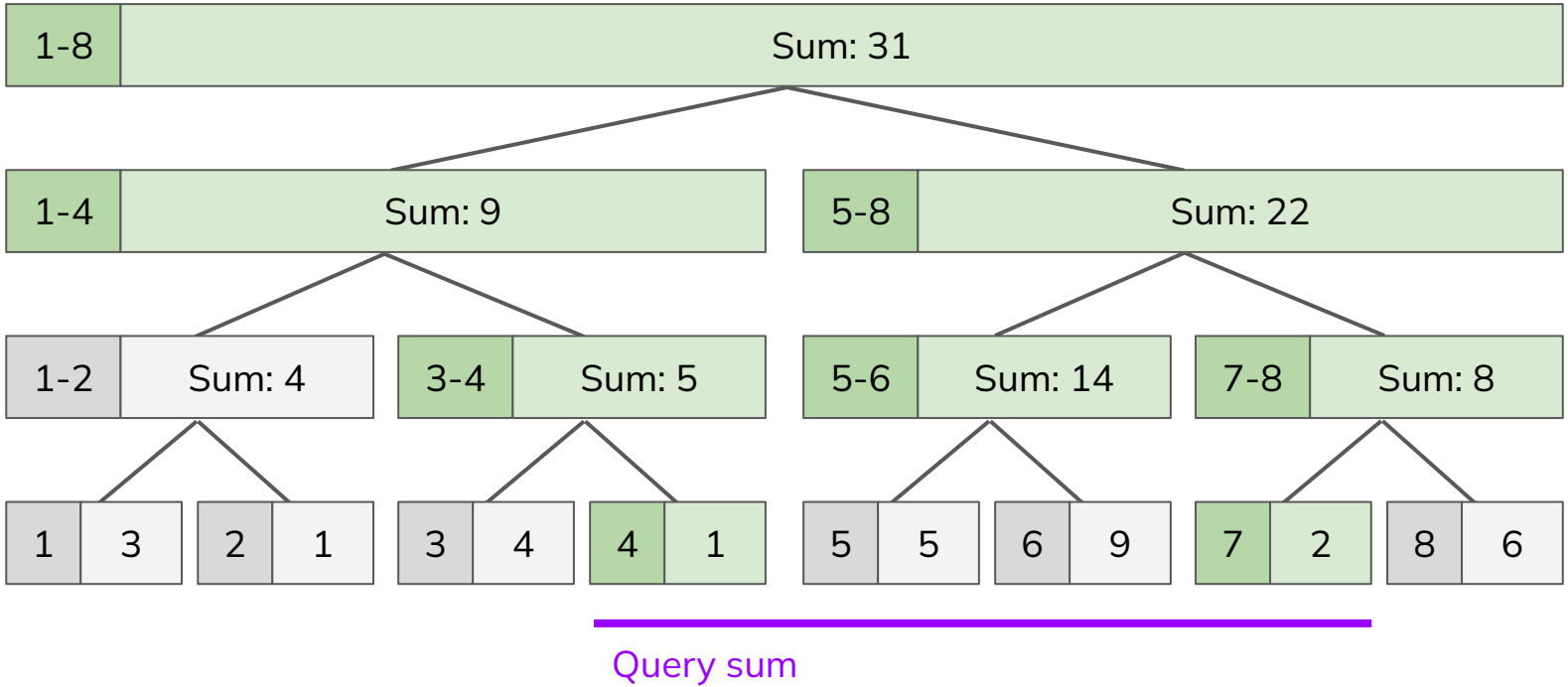

1-8
Sum: 31
1-4
Sum: 9
5-8
Sum: 22
1-2
Sum: 4
3-4
Sum: 5
5-6
Sum: 14
7-8
Sum: 8
1
3
2
1
3
4
4
1
5
5
6
9
7
2
8
6
Query sum

Range… query?

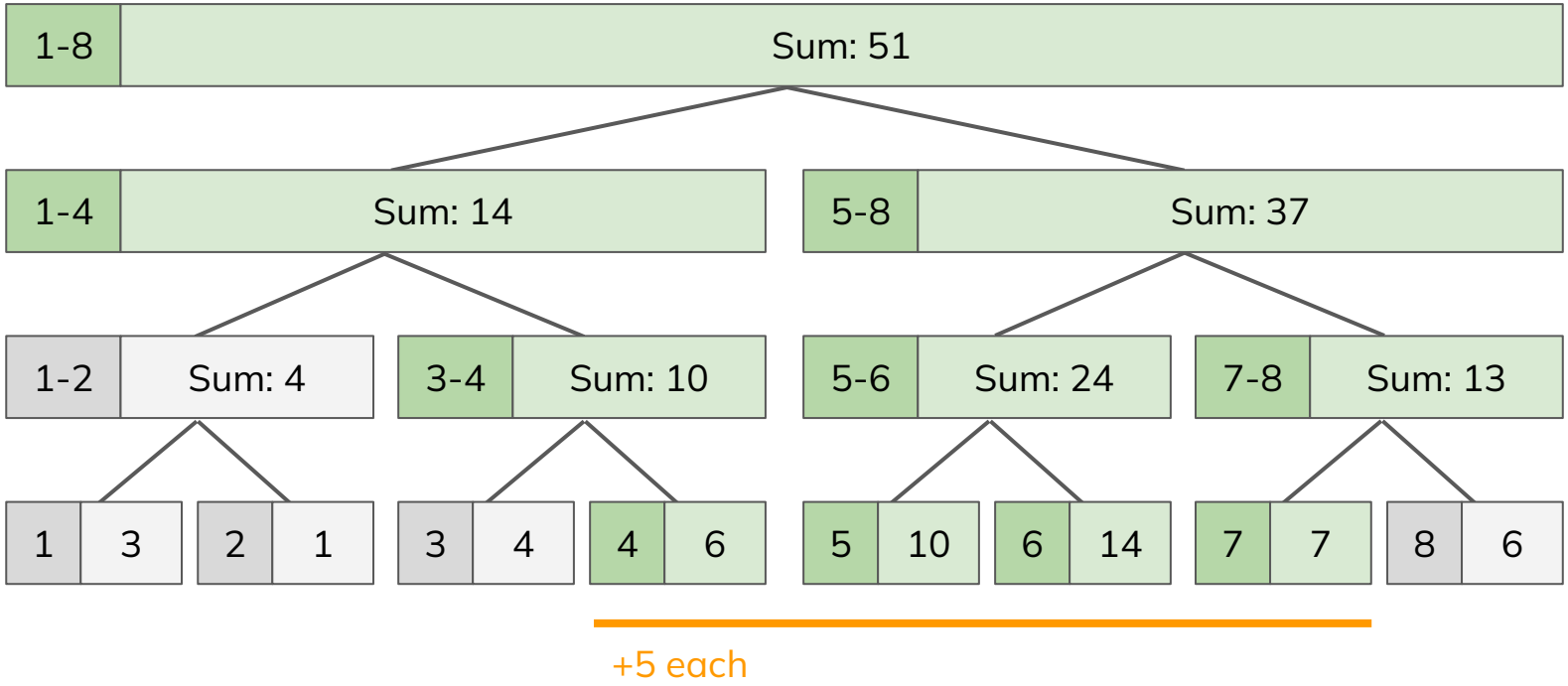
Purple nodes: all nodes we return the sum



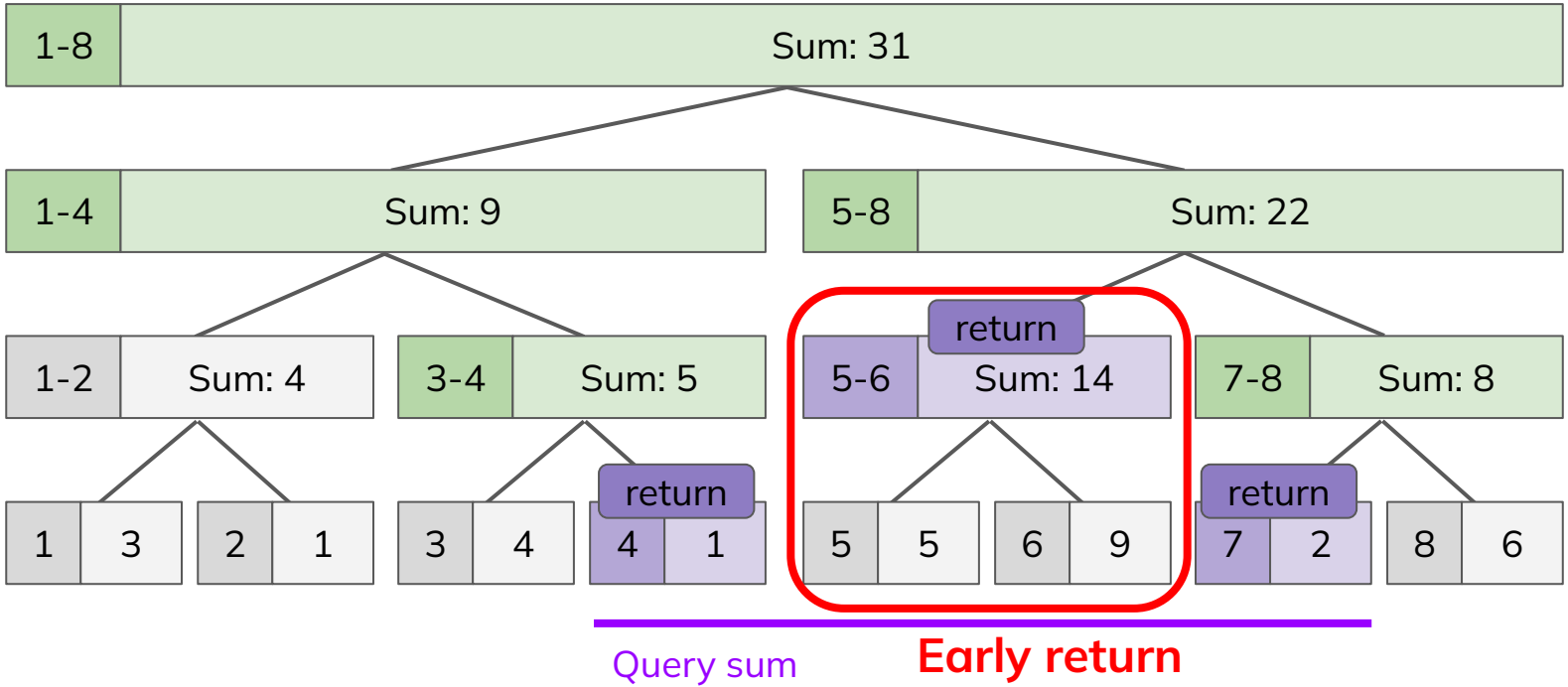
Range… query?


1-8
Sum: 31
1-4
Sum: 9
5-8
Sum: 22
1-2
Sum: 4
3-4
Sum: 5
5-6
Sum: 14
7-8
Sum: 8
1
3
2
1
3
4
4
1
5
5
6
9
7
2
8
6
Query sum

Noticed any difference?



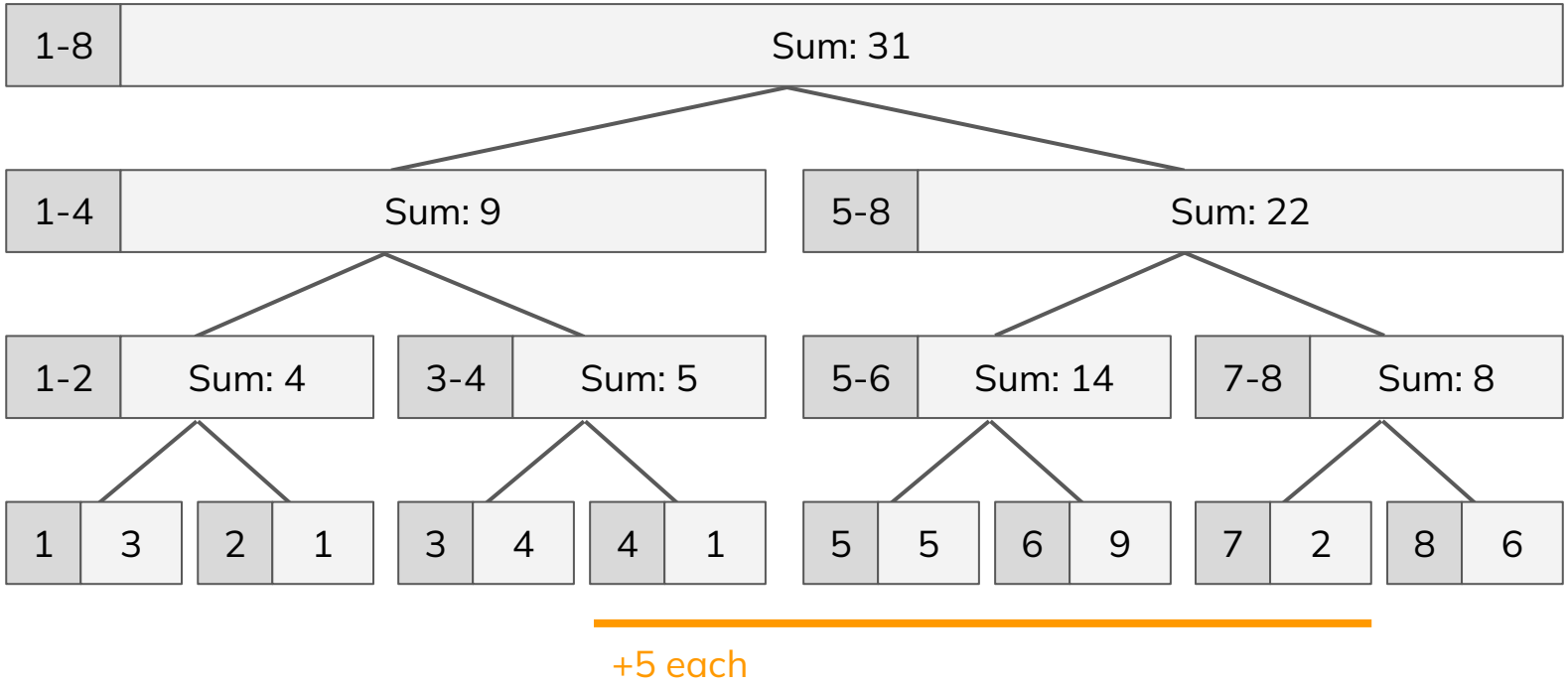
Noticed any difference?



Range Update: Early Return

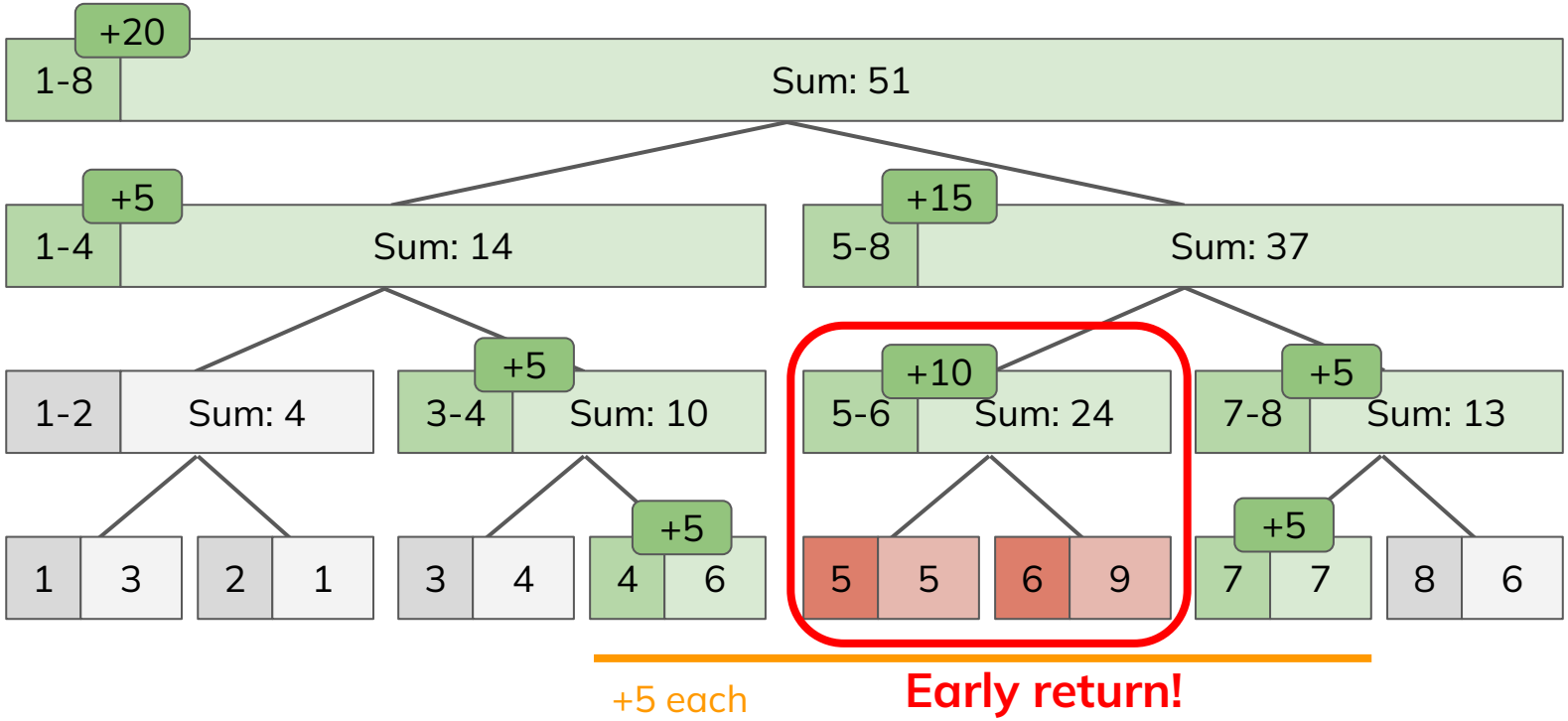
- Let's copy the same technique to range update to make it run faster!
- We will return directly when the whole range is covered by the update
- What would happen?
- What could possibly go wrong?

Range Update: Early Return

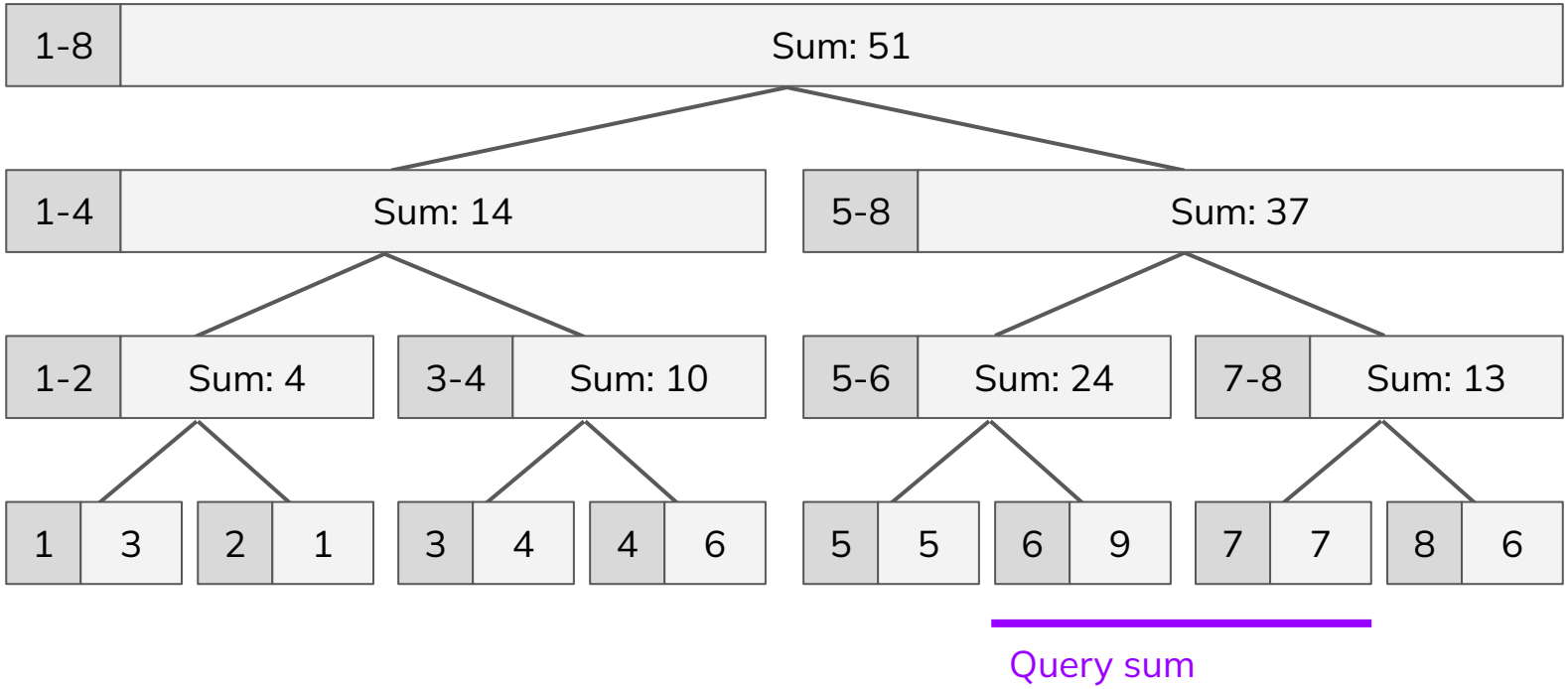


Range Update: Early Return

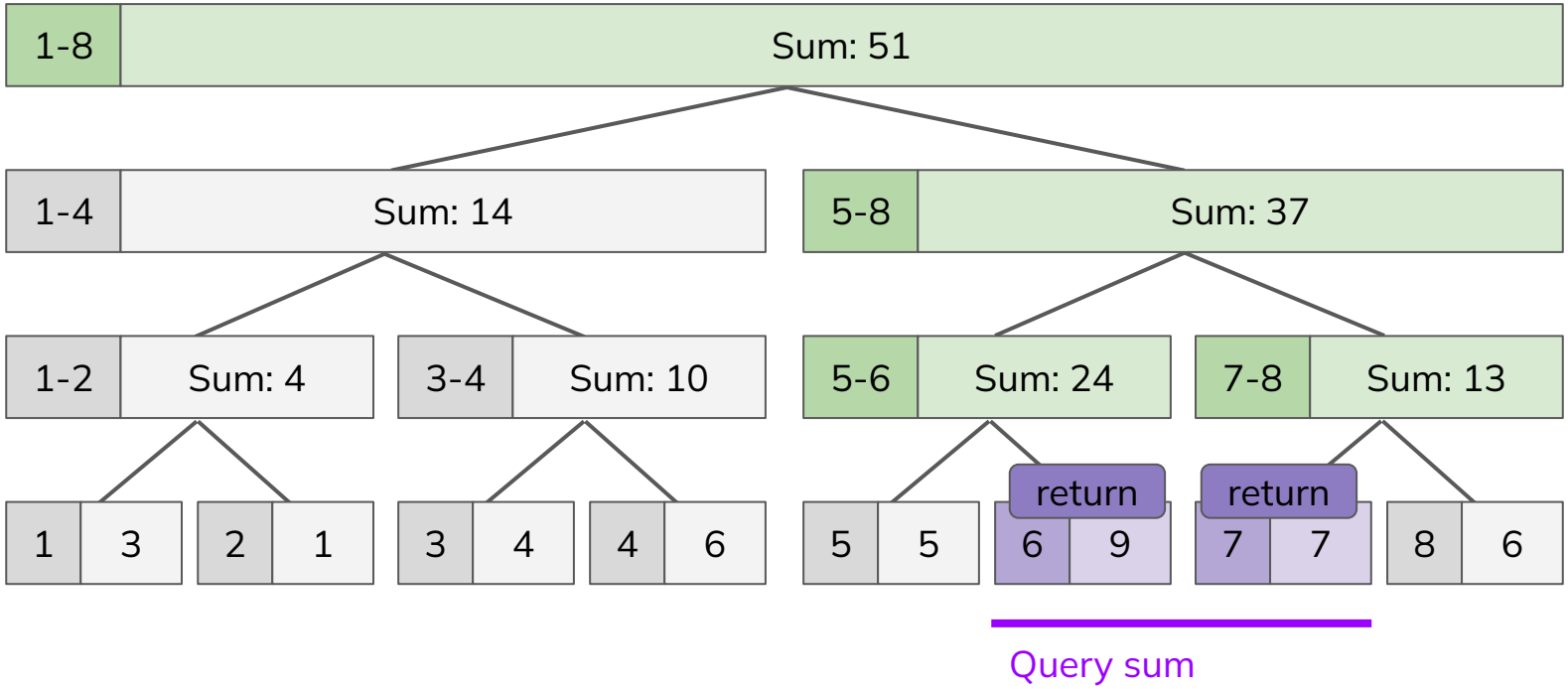
Red nodes: all nodes we skipped



Range Update: Early Return

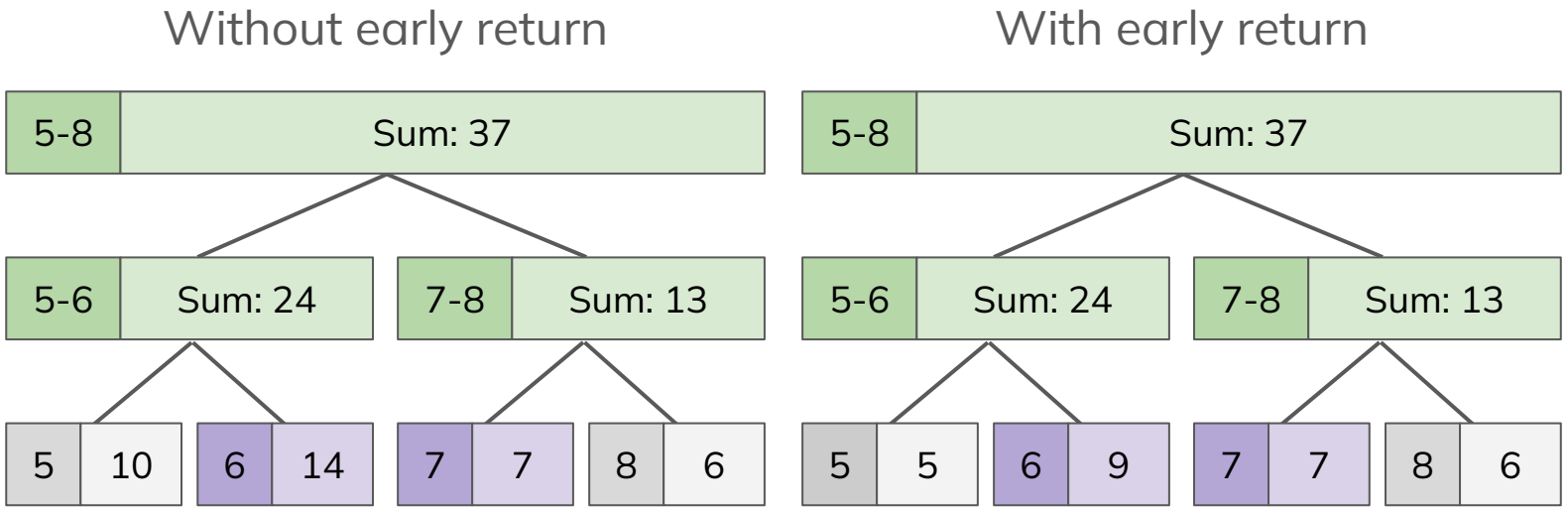


Range Update: Early Return



Range Update: Early Return

- Something's not right...
- Not updating every node led to consequences...

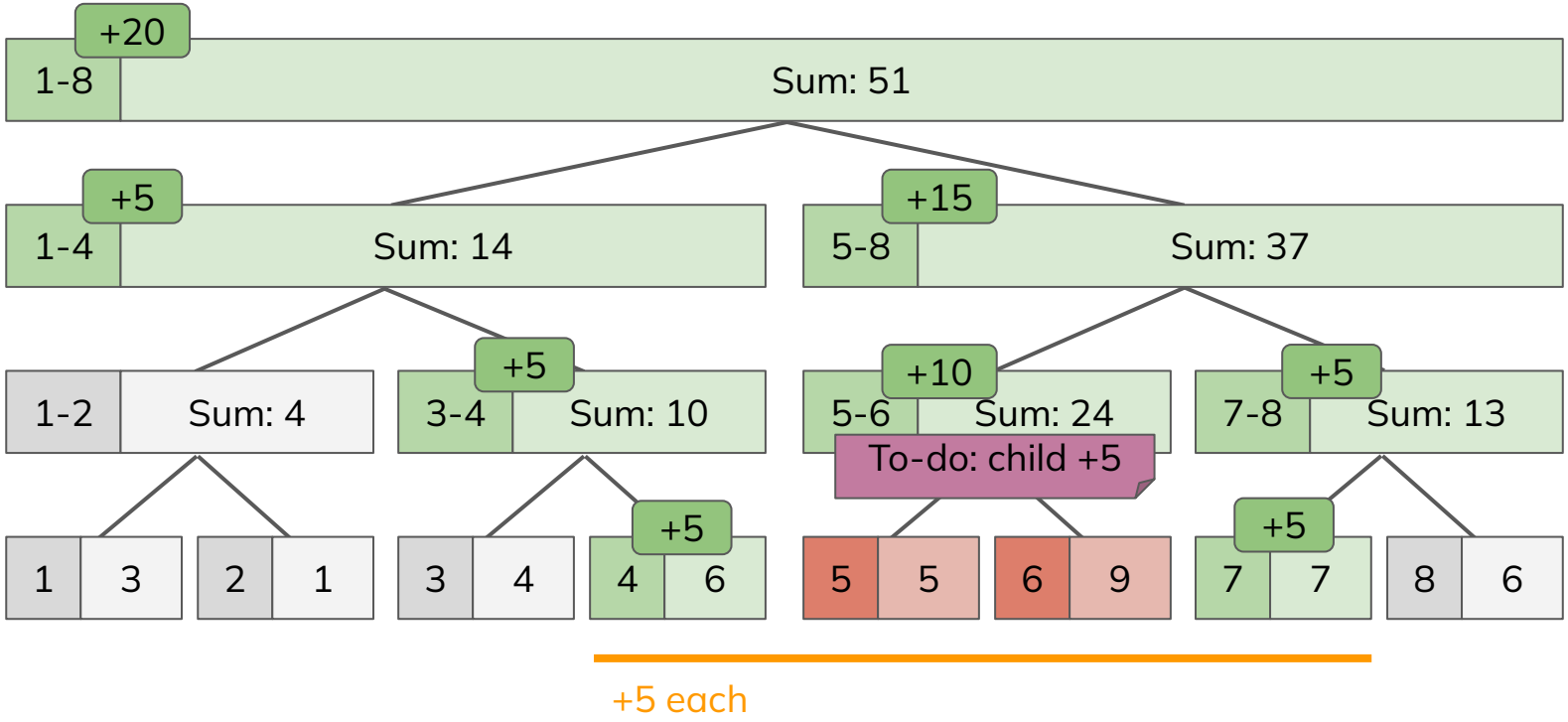


Range Update: Early Return

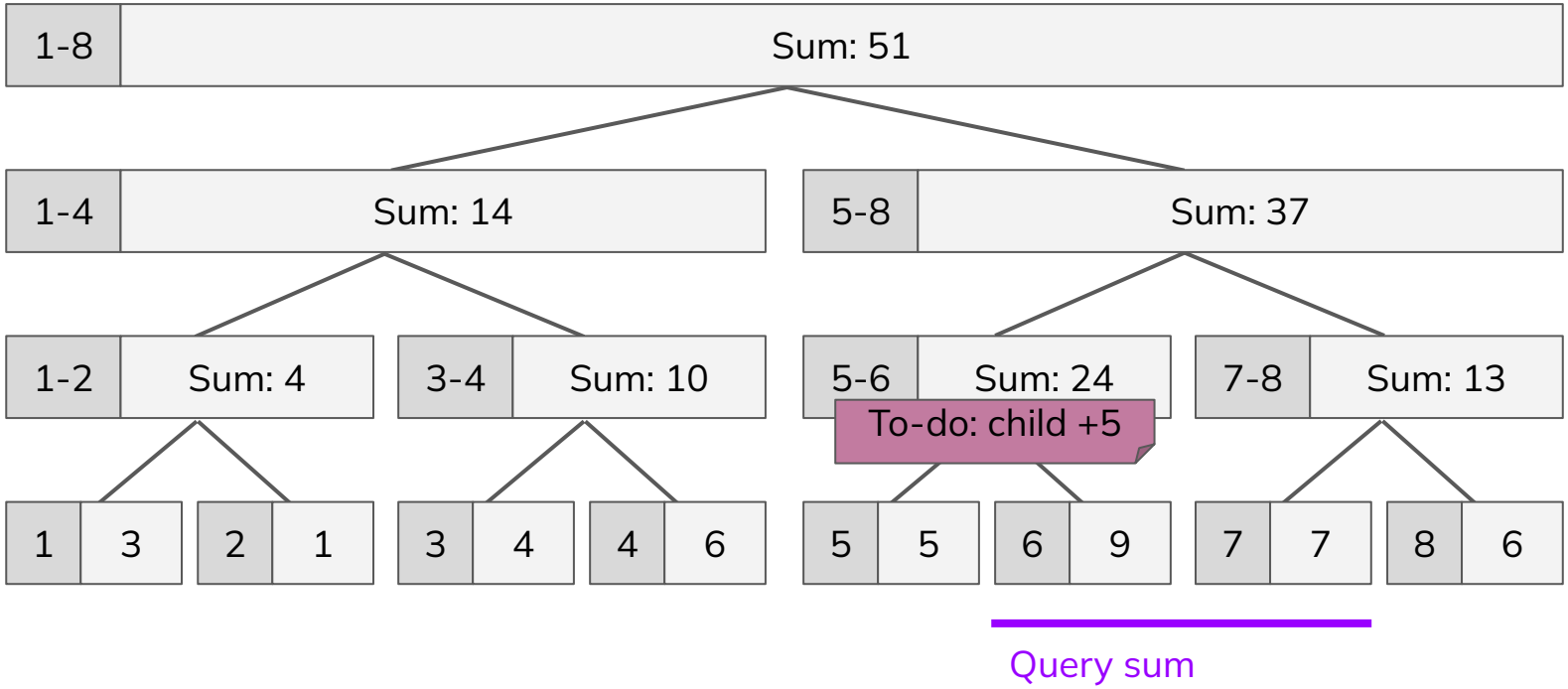
Why did it not work?

- We updated the sum of range [5, 6]
- But we didn't update [5] and [6] individually
- Let's just keep "updating [5] and [6] individually" in a to-do list

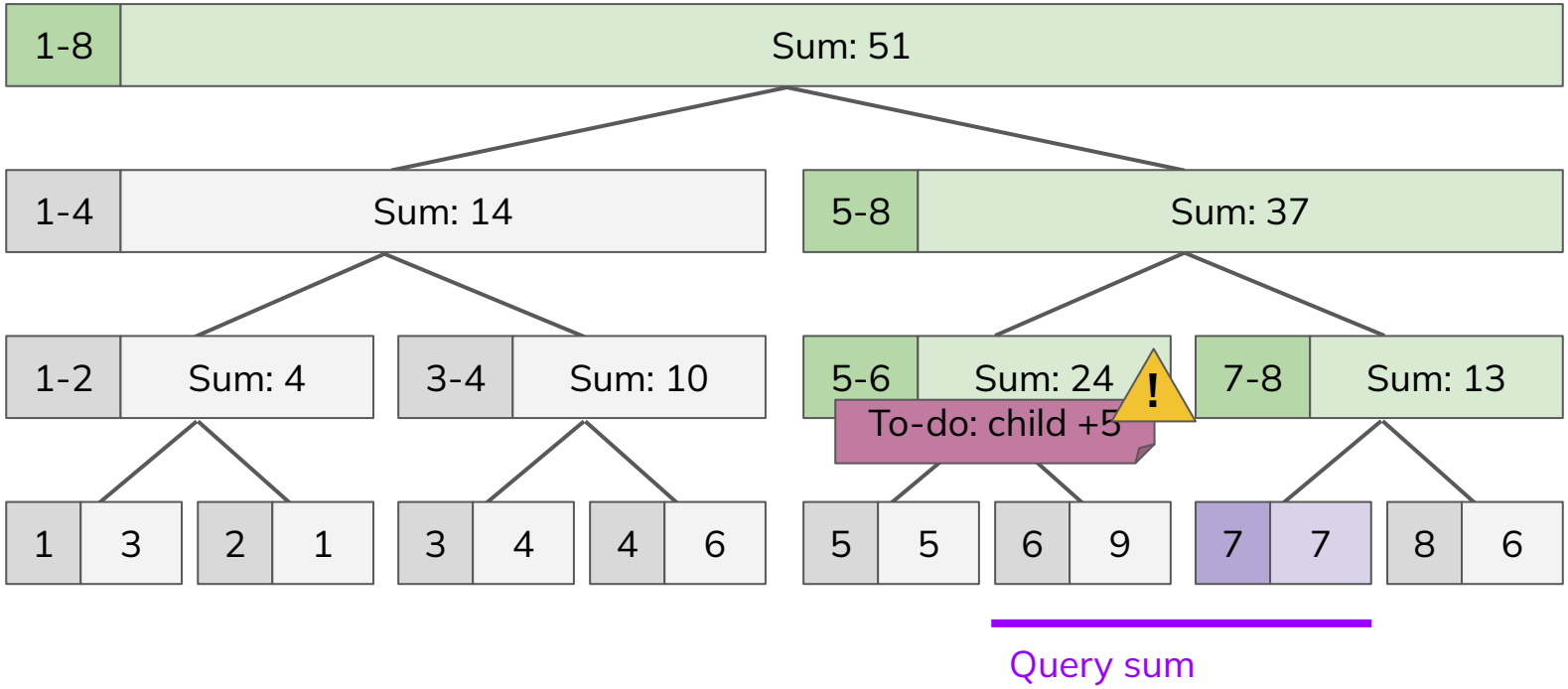
Range Update: To-do List



Range Update: To-do List

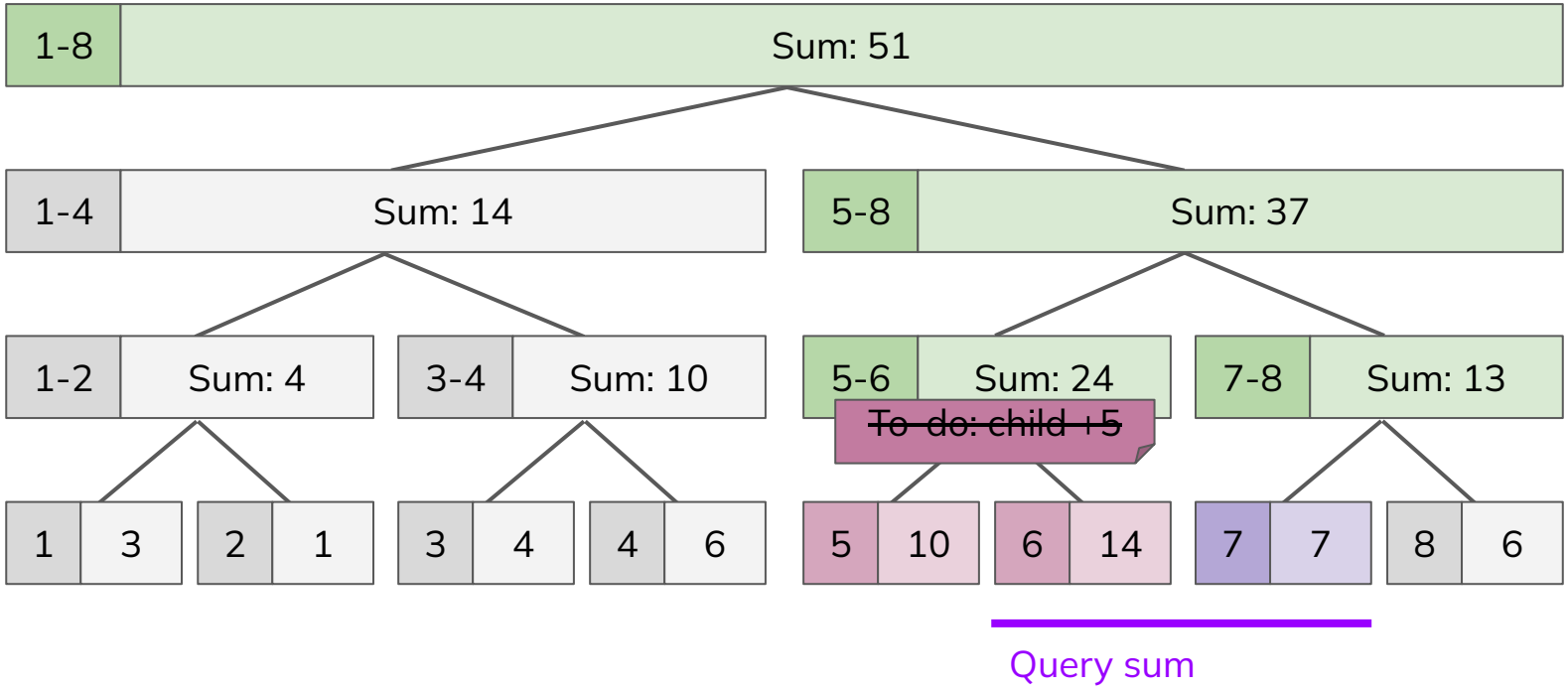


Range Update: To-do List

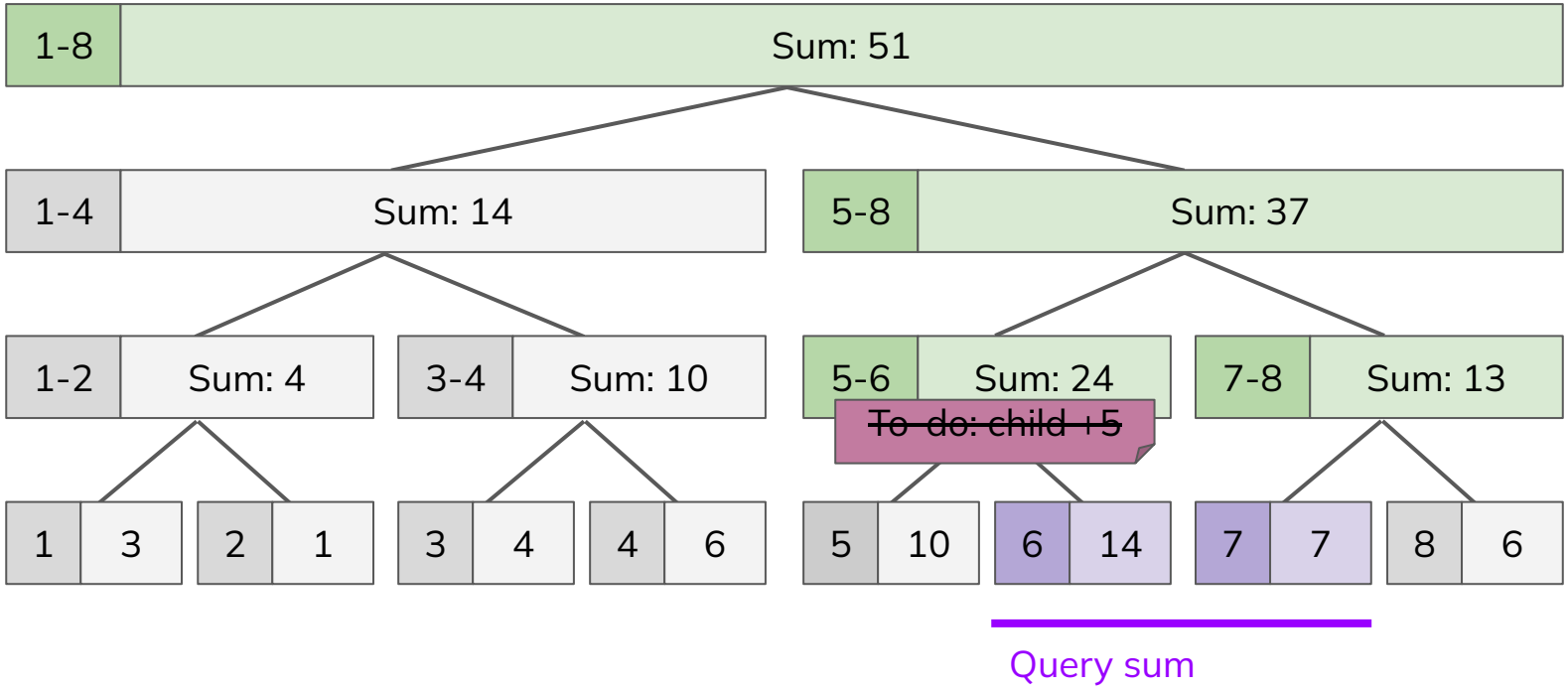


Range Update: To-do List

Magenta nodes: all nodes in to-do list



Range Update: To-do List



Range Update: To-do List

How is it faster?

- Our range update is now O(log n) as it is same as (old) range query
- New range query is still O(log n)! (Why?)
  - Updating value of both child node is O(1)
  - Pass down the to-do list to both child is also O(1)
    - We didn't do it in the example since the child is already a leaf
  - We will execute it at most O(log n) times

This is known as **lazy propagation**

Lazy Propagation: Update Implementation

Sample code

```
update(id, l, r, ql, qr, qv):
    if r < ql or l > qr: return
    if l >= ql and r <= qr:
        sum[id] += (r - l + 1) * qv
        lazy[id] += qv
        return
    m = (l + r) / 2
    lazy[id * 2] += lazy[id]
    sum[id * 2] += (m - l + 1) * lazy[id]
    lazy[id * 2 + 1] += lazy[id]
    sum[id * 2 + 1] += (r - m) * lazy[id]
    lazy[id] = 0
    ...
```

Lazy Propagation: Query Implementation

Sample code

```
query(id, l, r, ql, qr):
    if r < ql or l > qr: return 0
    if l >= ql and r <= qr: return sum[id]
    m = (l + r) / 2
    lazy[id * 2] += lazy[id]
    sum[id * 2] += (m - l + 1) * lazy[id]
    lazy[id * 2 + 1] += lazy[id]
    sum[id * 2 + 1] += (r - m) * lazy[id]
    lazy[id] = 0
    ...
```

Lazy Propagation: Better$_{(My)}$ Implementation

Sample code

```
pushdown(id, l, r, qv):
    sum[id] += (r - l + 1) * qv
    lazy[id] += qv
```

```
update(id, l, r, ql, qr, qv):
    if r < ql or l > qr: return
    if l >= ql and r <= qr:
        pushdown(id, l, r, qv)
        return
    m = (l + r) / 2
    pushdown(id * 2, l, m, lazy[id])
    pushdown(id * 2 + 1, m + 1, r, lazy[id])
    lazy[id] = 0
    ...
```

```
query(id, l, r, ql, qr):
    if r < ql or l > qr:
        return 0
    if l >= ql and r <= qr:
        return sum[id]
    m = (l + r) / 2
    pushdown(id * 2, l, m, lazy[id])
    pushdown(id * 2 + 1, m + 1, r, lazy[id])
    lazy[id] = 0
    ...
```

Lazy Propagation

- Instead of updating every index for each range update, we can store the update in intervals (i.e. the non-leaf nodes of the segment tree)
- For each node, we only relay the update information when one of its child nodes is used in a query
- Only O(log N) nodes will be updated for each query and each update (Why?)

Lazy Propagation: Implementation

- We can maintain an extra “lazy tag” to store the pending updates for its children for each node of the subtree
- When we update a node, if its range is fully within the update range, we store the update in the lazy tag and not update its children
- When we query a node, before querying its left and right children, we “push” the updates stored in the lazy tag of this node to its children
- Try to implement this in A101 Lazy Propagation

Lazy Propagation: Practice Tasks

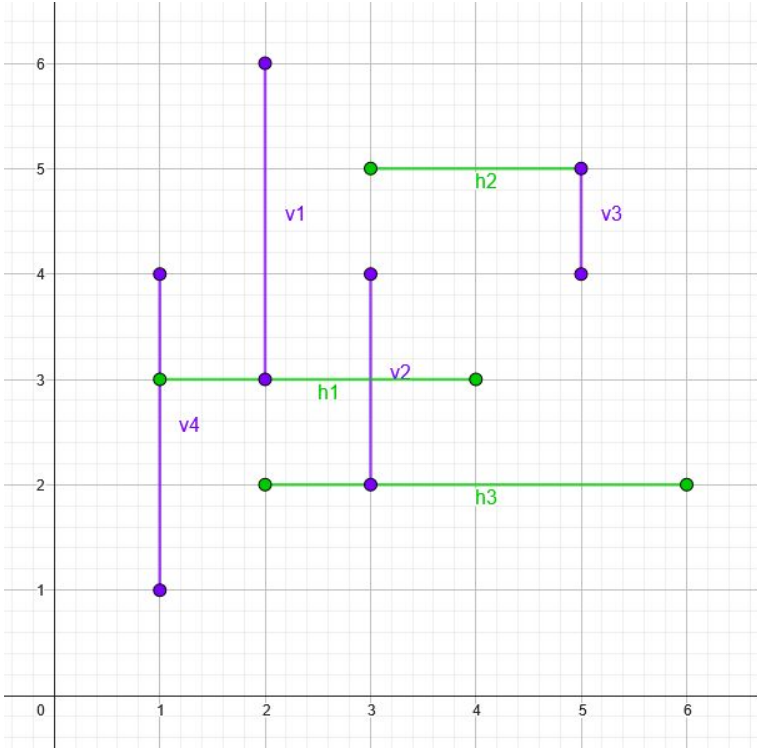
- A101 Lazy Propagation
- T192 Colourful Strips
- T213 Game Developer
- T253 Peaceful Pirate Pairs
- CF446C DZY Loves Fibonacci Numbers

break;

Relax a bit, or try to solve the practice tasks!

Intersection Problem

- You have N horizontal line segments and M vertical line segments on a 2D Cartesian plane
- Find the number of intersections between all (N+M) line segments
- No parallel line segments overlap (for easier implementation)



Intersection Problem: Naive Solution

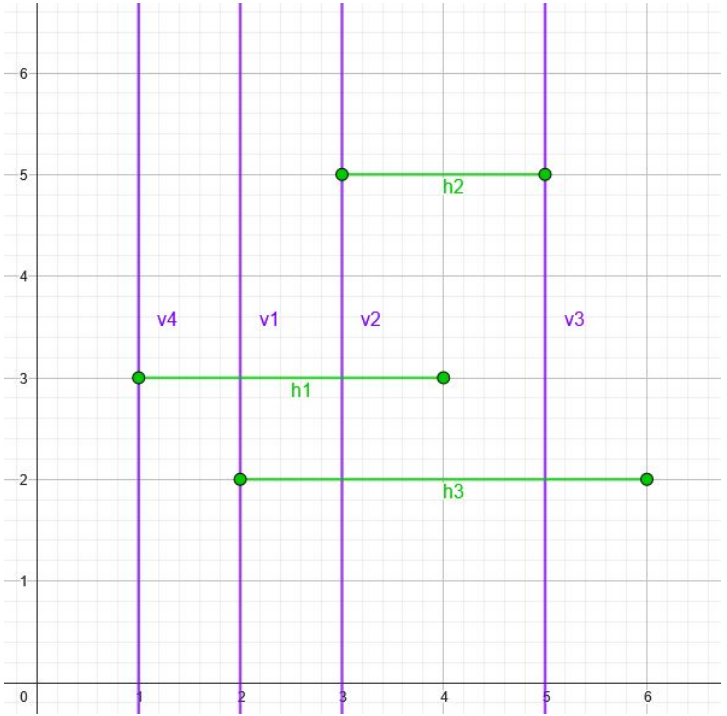
- For every horizontal segment, we count the number of vertical segments that intersect with it
- For each pair of horizontal segment and vertical segment, we can check for intersection in O(1)
- This will give a total time complexity of O(NM)
- Can we speed it up?

Intersection Problem: Easy Version

New constraint: Every vertical segment goes from y=-∞ to y=+∞
(i.e. it is a line instead of line segment)

- How can we solve this?

- Use partial sum
- +1 for all x coordinates of vertical lines
- Query sum from x_1 to x_2 for all horizontal lines



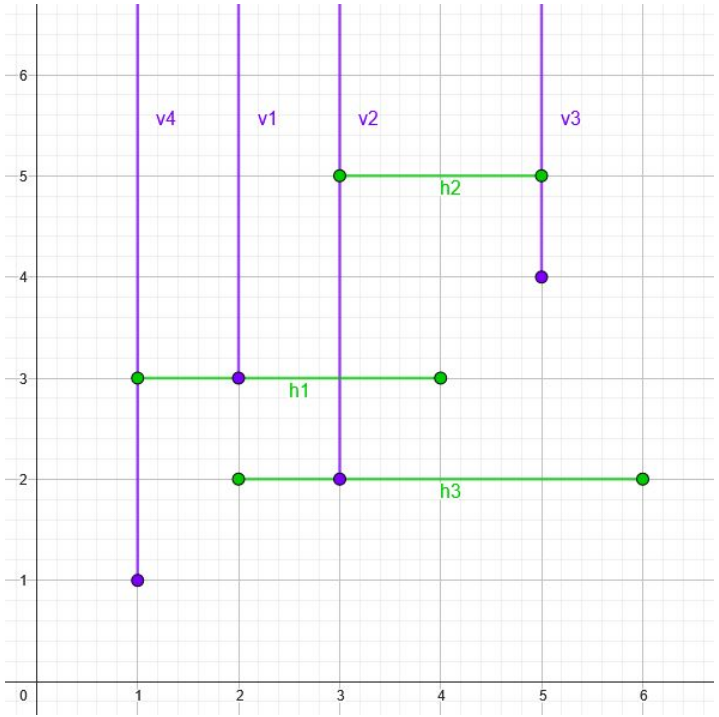
Intersection Problem: Not So Easy Version

Modified constraint: Every vertical segment goes from $y=y_1$ to $y=+\infty$
(i.e. it is a ray instead of line segment)

- How can we solve this?

- Use a segment tree instead
- Sort the horizontal segments by y
- Only +1 when the line segment starts

- Basically the full solution!



Intersection Problem: Sweep Line

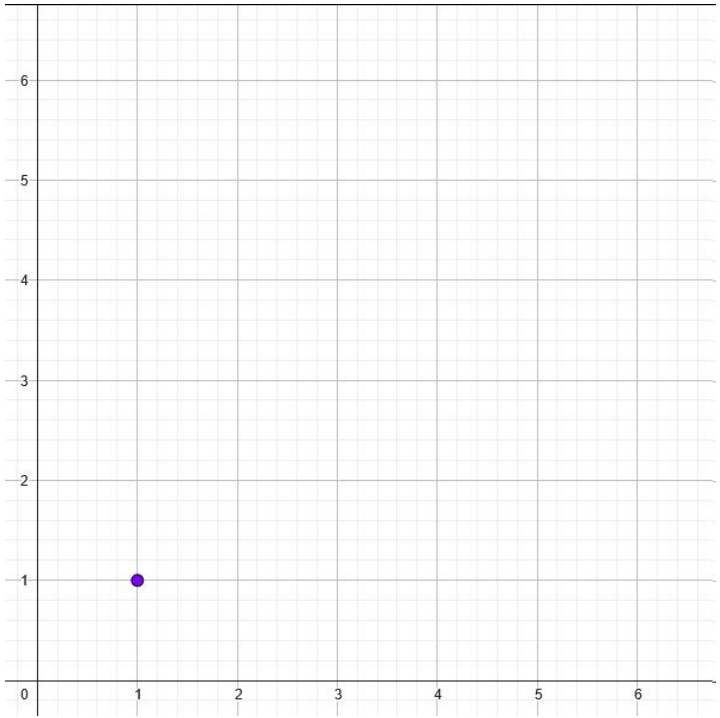
- Sort the horizontal lines by y
- We can view the endpoints of a vertical segment as a “toggle”
- Whenever we have a toggle at (X, Y), wetoggle whether there is a vertical segment at x-coordinate X
  - We toggle on before query operations and toggle off after query operations
- Whenever you have a horizontal line segment (X1, Y) - (X2, Y), you query the number of “active” vertical segments in x = [X1, X2]
  - This can be done using a **Segment Tree** or a **Fenwick Tree**
- This technique is called **Sweep Line**, and it is used to transform 2D problems into 1D range query problems
- You can try to implement it in A102 Sweep Line: Count Intersections

Intersection Problem: Example

y = 1

- toggleOn(1)

x	1	2	3	4	5	6
line[x]	1	0	0	0	0	0

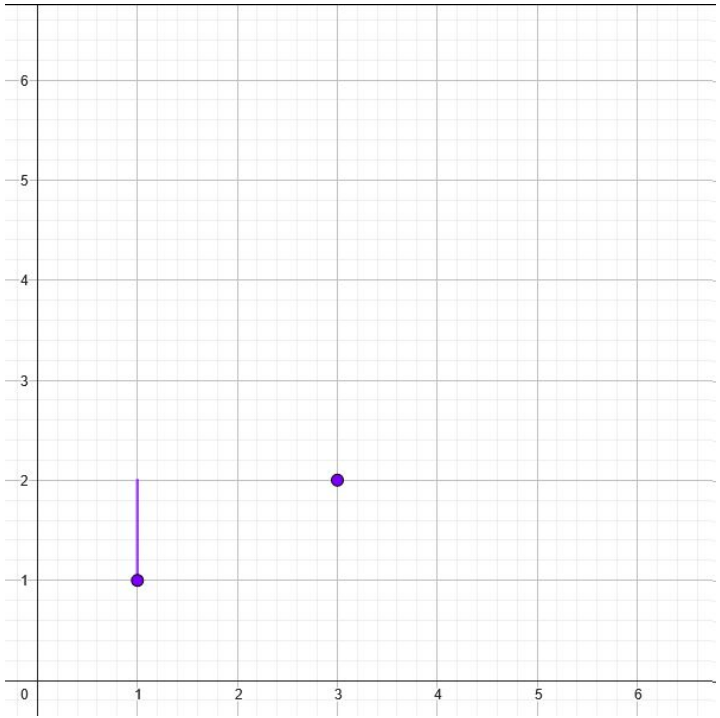


Intersection Problem: Example

y = 2

- toggleOn(3)

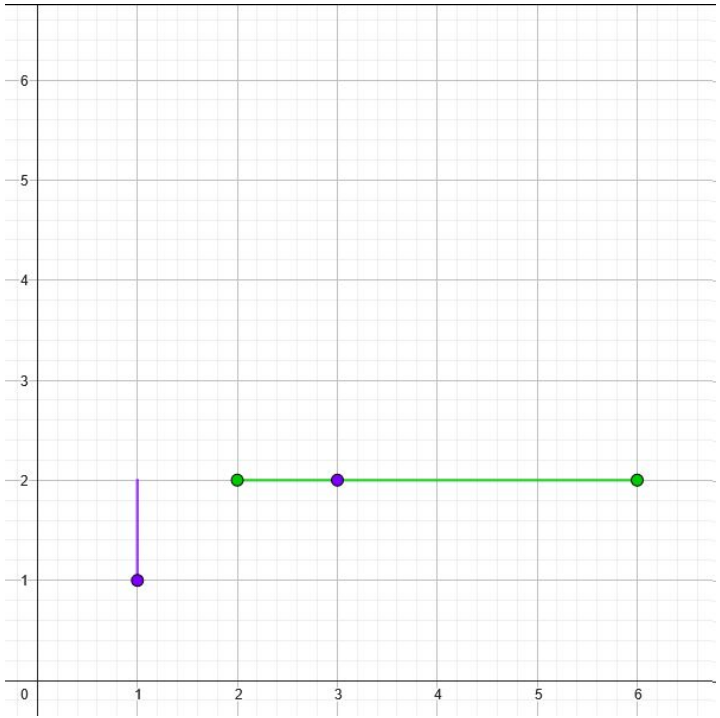
x	1	2	3	4	5	6
line[x]	1	0	1	0	0	0



Intersection Problem: Example

y = 2

- toggleOn(3)
- query(2, 6) = 1

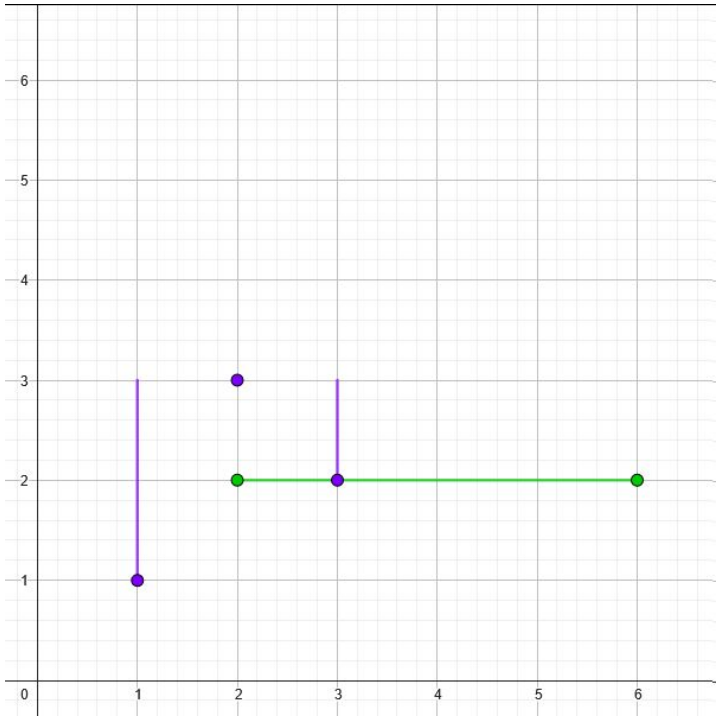


x	1	2	3	4	5	6
line[x]	1	0	1	0	0	0

Intersection Problem: Example

y = 3

- toggleOn(2)

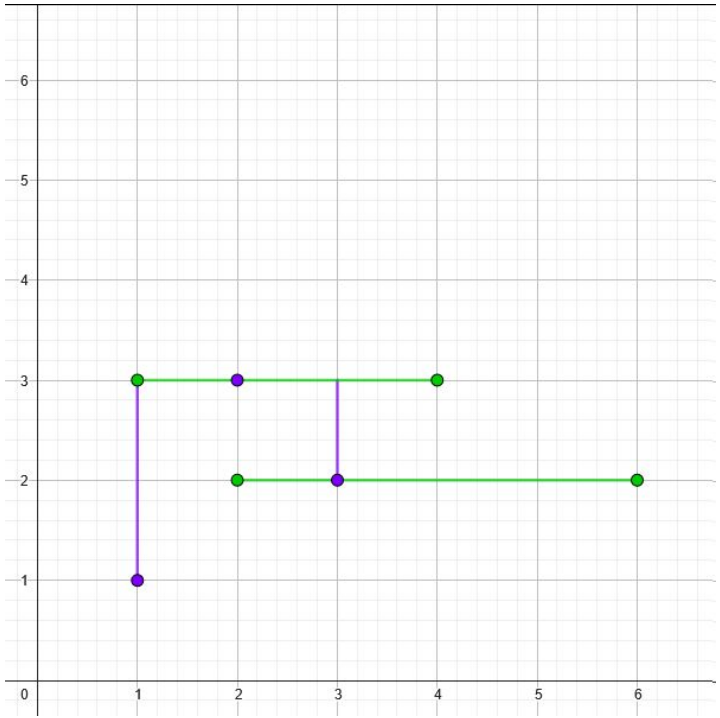


x	1	2	3	4	5	6
line[x]	1	1	1	0	0	0

Intersection Problem: Example

y = 3

- toggleOn(2)
- query(1, 4) = 3

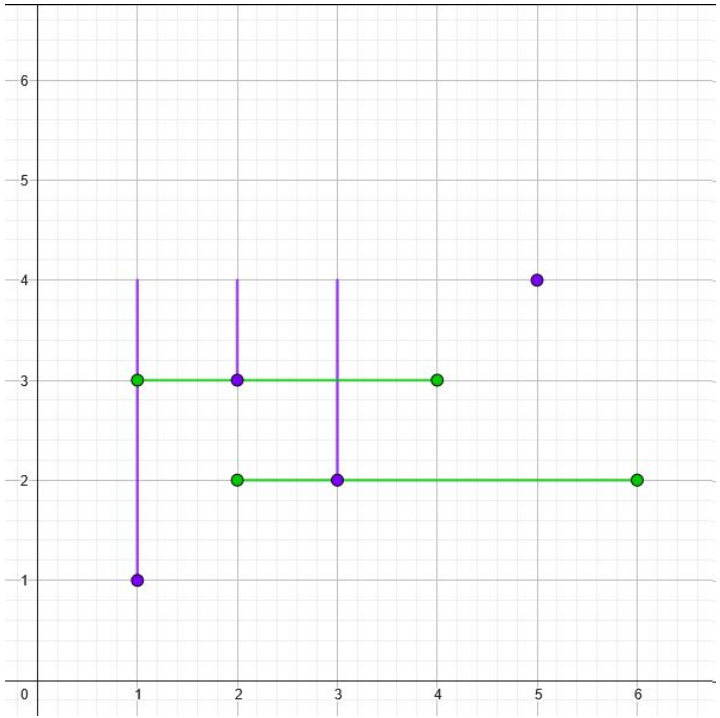


x	1	2	3	4	5	6
line[x]	1	1	1	0	0	0

Intersection Problem: Example

y = 4

- toggleOn(5)

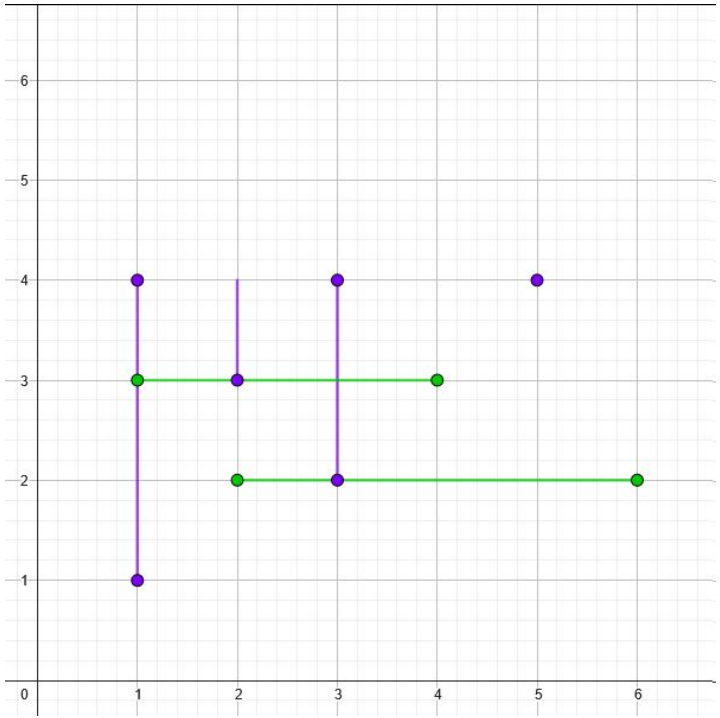


x	1	2	3	4	5	6
line[x]	1	1	1	0	1	0

Intersection Problem: Example

y = 4

- toggleOn(5)
- toggleOff(1)
- toggleOff(3)

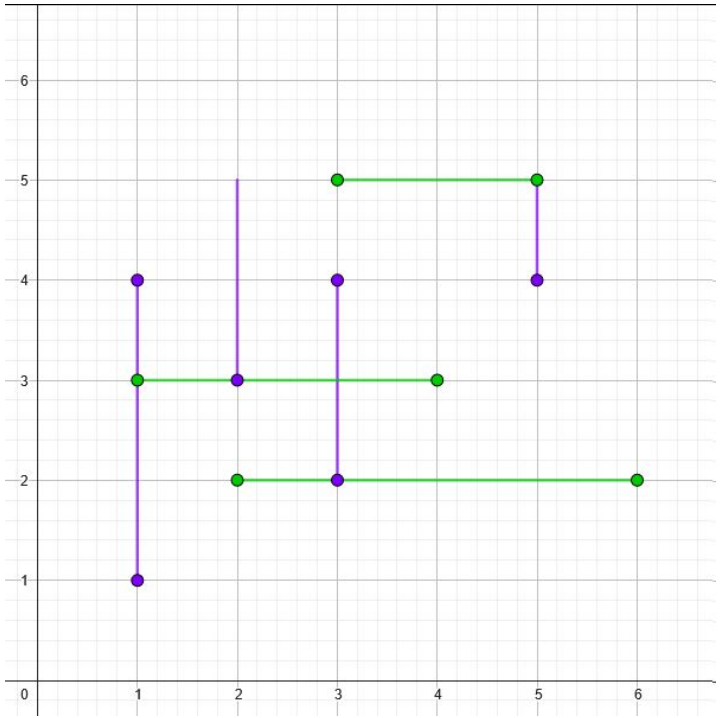


x	1	2	3	4	5	6
line[x]	0	1	0	0	1	0

Intersection Problem: Example

y = 5

- query(3, 5) = 1

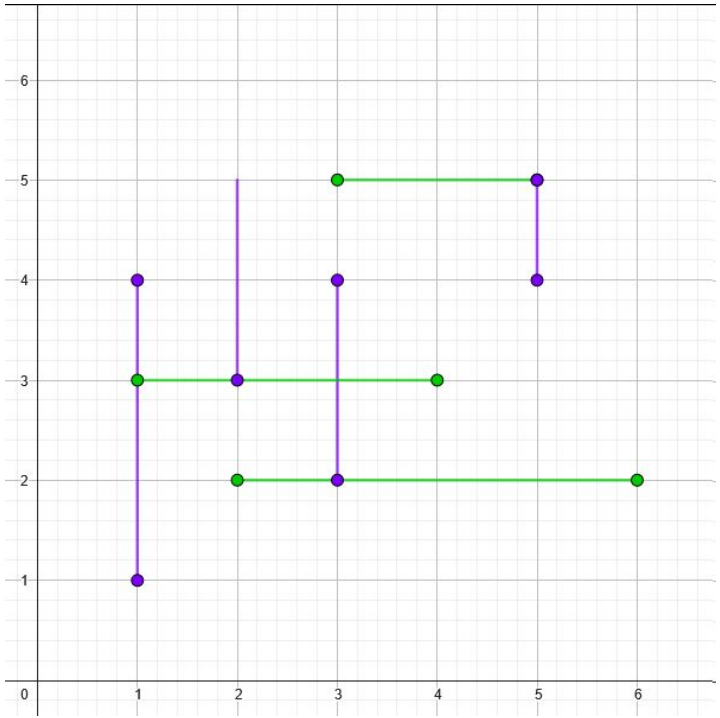


x	1	2	3	4	5	6
line[x]	0	1	0	0	1	0

Intersection Problem: Example

y = 5

- query(3, 5) = 1
- toggleOff(5)

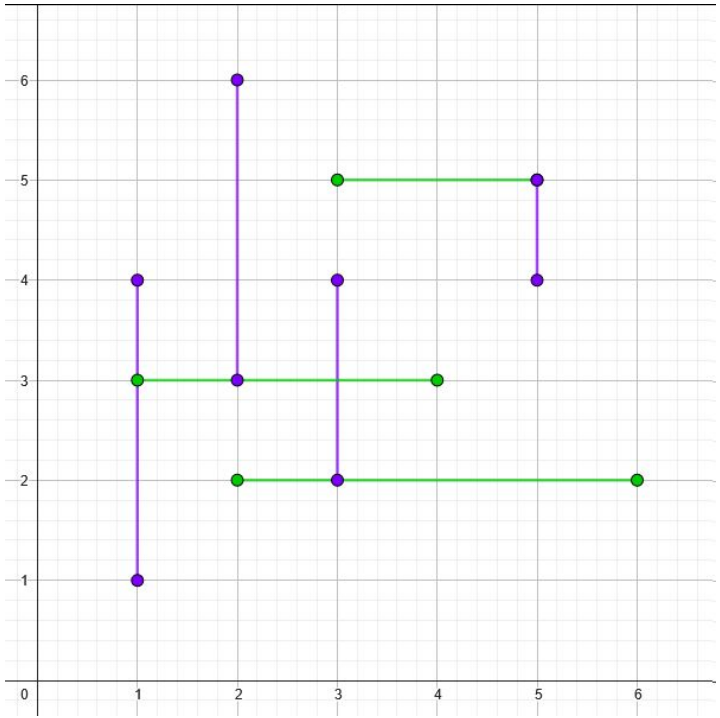


x	1	2	3	4	5	6
line[x]	0	1	0	0	0	0

Intersection Problem: Example

y = 6

- toggleOff(2)



x	1	2	3	4	5	6
line[x]	0	0	0	0	0	0

Sweep Line: Practice Tasks

- A102 Sweep Line: Count Intersections
- NOI23 方格染色
- APIO18 New Home

More Range Query Problems

- Range min query problems are easy[citation needed]
- But what if, instead of query about the min, each query we would like to know the K-th min in the range [L, R]?
- Can we still solve that in reasonable time?

K-th Min Query: Easy Version

- Let's first consider a (considerably) easier problem: what is the K-th min in the entire array A? (i.e. L = 1, R = N for all queries)
- Easiest way is to sort it and take the K-th element every time!

K-th Min Query: Not so Easy Version

- Now we consider the ranges to be a prefix of the array instead of the whole array (i.e. L = 1 for all queries)
- Updating a sorted array for different R is not easy...
- Let's think of something easier to update
- We can try using frequency array!

K-th Min Query: Easy Version Again

- ~~Easiest way is to sort it and take the K-th element every time!~~
- Another way to solve the problem is to build a cumulative frequency array of A
- Binary search on it

A[]	1	4	1	4	2	1	3	5	6

i	1	2	3	4	5	6
freq[]	3	1	1	2	1	1
cfreq[]	3	4	5	7	8	9

Smallest element = 1 ↑ 6th-smallest element = 4 ↑ ↑ 8th-smallest element = 5

K-th Min Query: Not so Easy Version Again

Now, what can we do if we want to add a new element in A (i.e. increase R by 1)?

- Segment tree!
- By querying the sum of [1, i] we can get $cfreq_i$
- We can “offline” the queries using the sweep line trick on R.
- Query takes $O(\log^2 N)$ now
  - Binary search takes O(log N) iterations
  - Query $cfreq_i$ takes O(log N)
- But can we still query the K-th min in O(log N) time?
- Yes!
- We can perform a **binary search on segment tree/segment tree descent**

Binary Search on Segment Tree

- Recall how a binary search works on a sorted array:
  - If (target value < median value) search the first half of the range
  - Otherwise search the second half of the range
- Also recall how the children of a node represents the two halves of the range of the node in a segment tree
- We can substitute it in and get the following binary search algorithm:
  - If (target value < left child value) search the left child
  - Otherwise search the right child
- Note that we might have to modify the search value if we descent to the right child
  - In our case, when we descent to the right child we need to subtract search value by the value of left child

Binary Search on Segment Tree

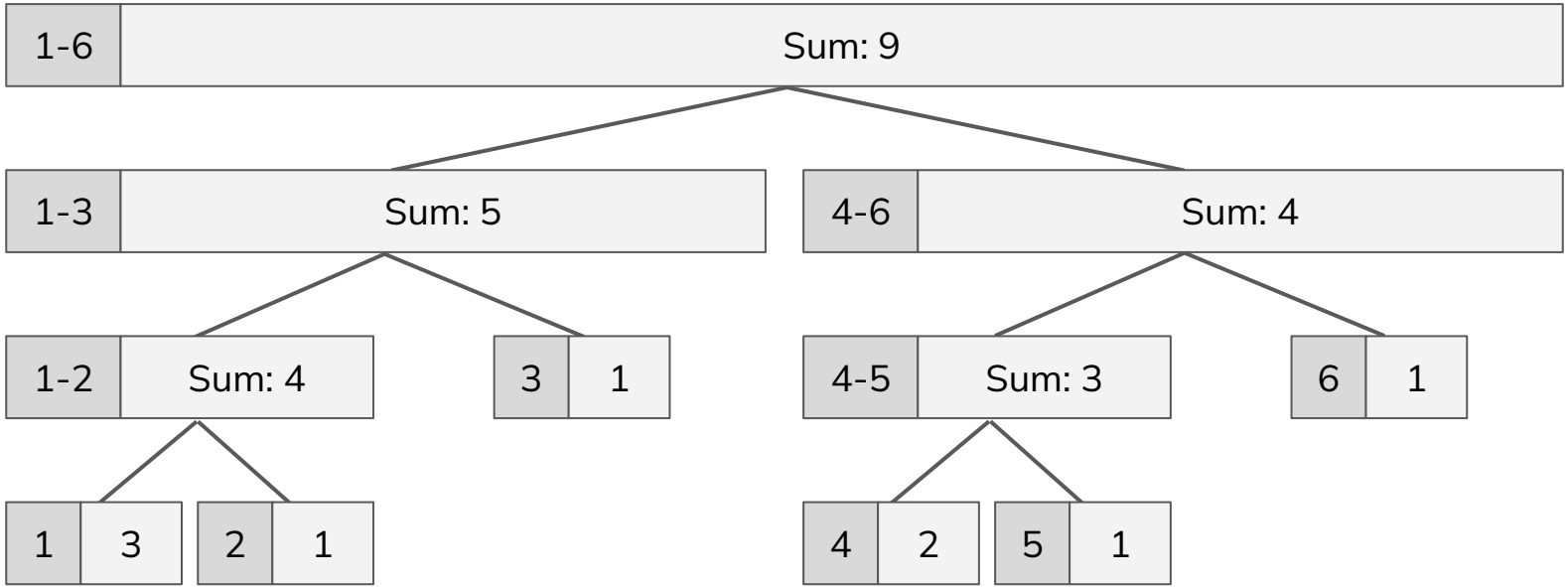
Let's try it on the previous example

A[]	1	4	1	4	2	1	3	5	6

i	1	2	3	4	5	6
freq[]	3	1	1	2	1	1
cfreq[]	3	4	5	7	8	9

↑ 8th-smallest element = 5

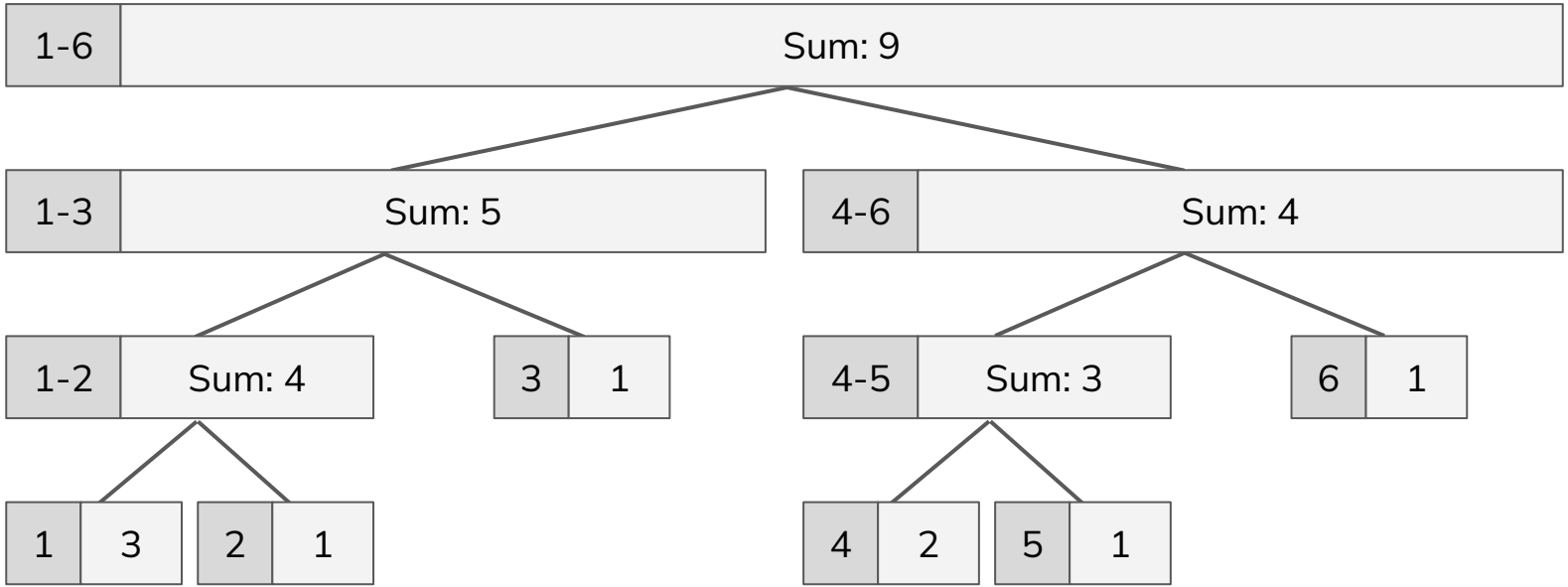
Binary Search on Segment Tree



freq[]	3	1	1	2	1	1

Binary Search on Segment Tree

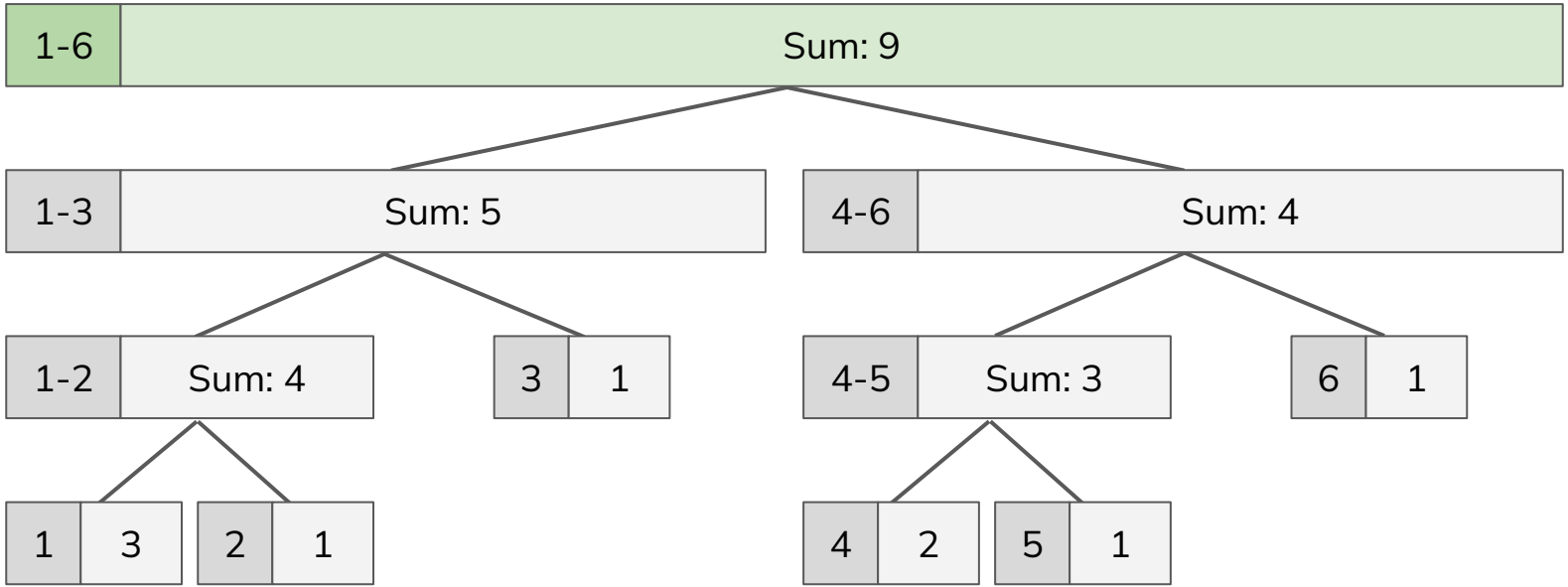
Target cfreq value: 8



freq[]	3	1	1	2	1	1

Binary Search on Segment Tree

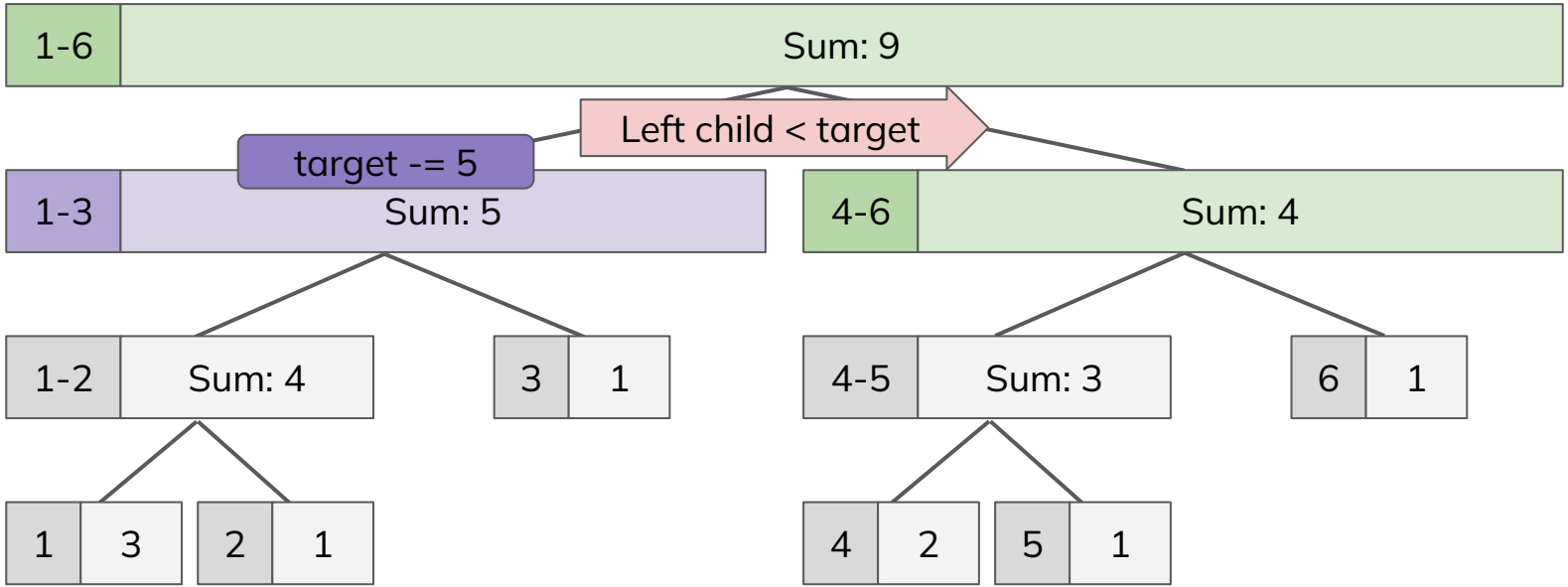
Target cfreq value: 8



freq[]	3	1	1	2	1	1

Binary Search on Segment Tree

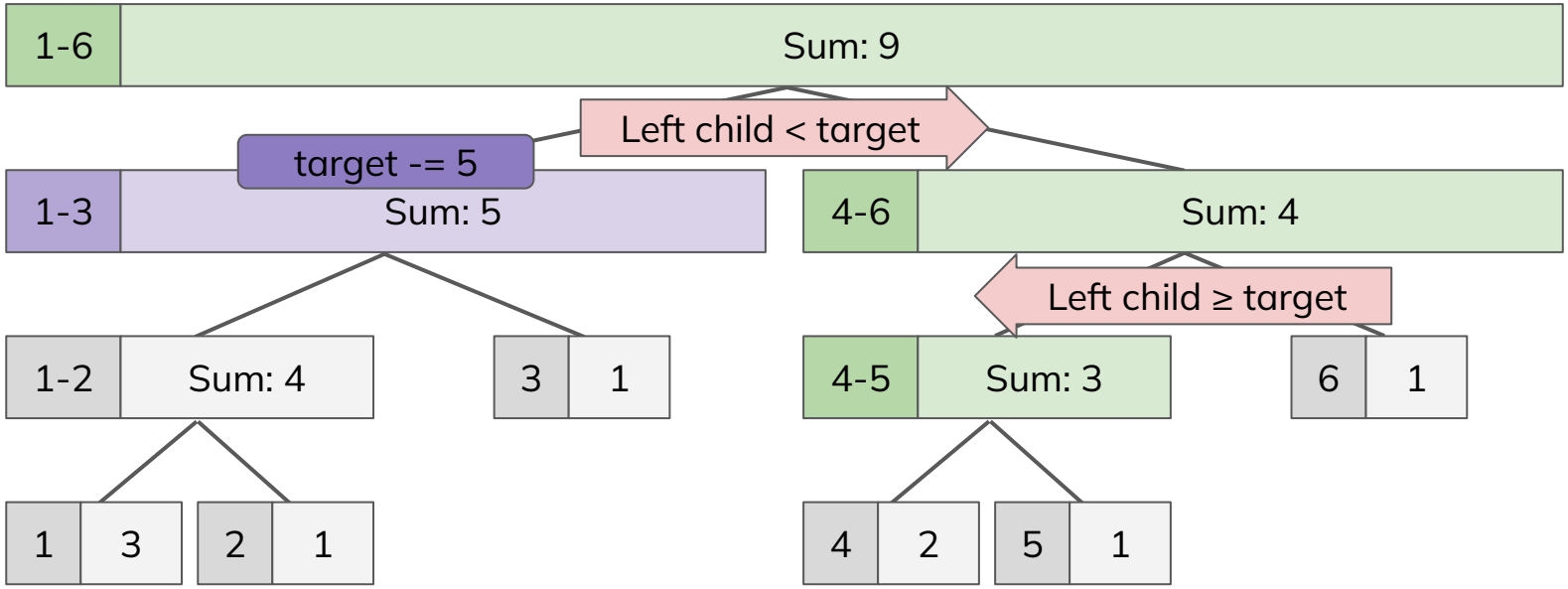
Target cfreq value: 8 - 5 = 3



freq[]	3	1	1	2	1	1

Binary Search on Segment Tree

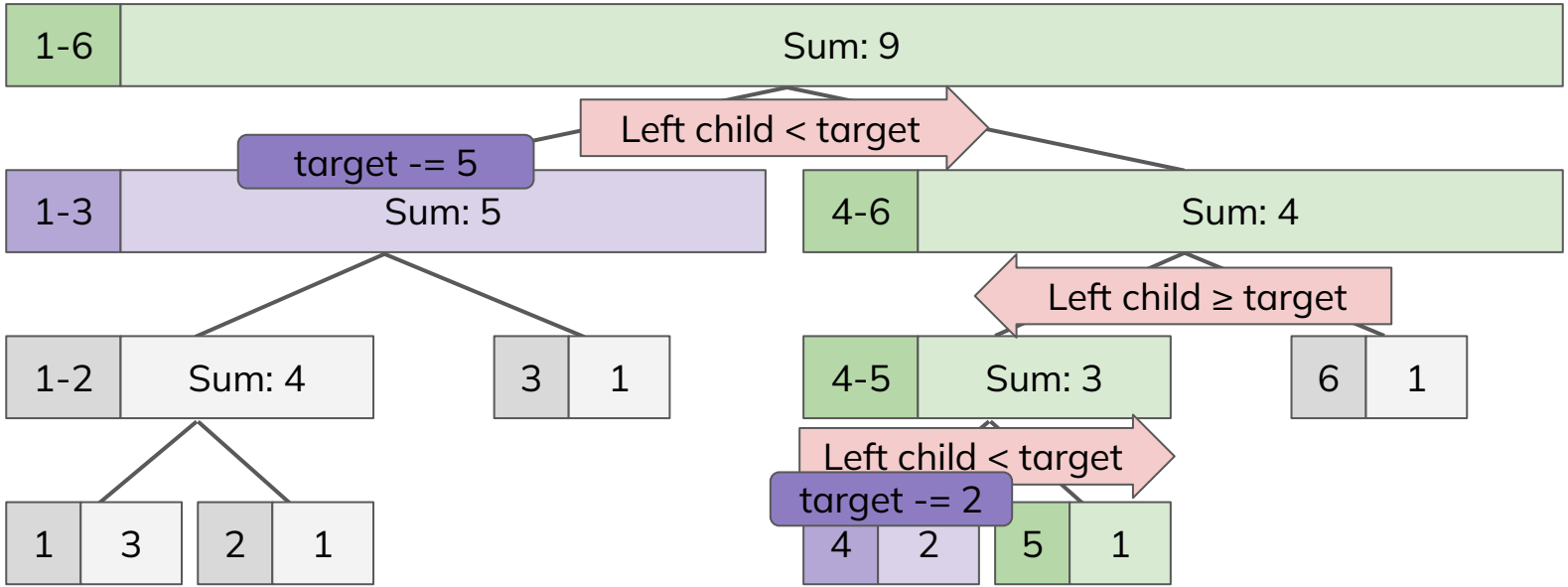
Target cfreq value: 3



freq[]	3	1	1	2	1	1

Binary Search on Segment Tree

Target cfreq value: 3 - 2 = 1



↑ 8th-smallest element = 5

freq[]	3	1	1	2	1	1

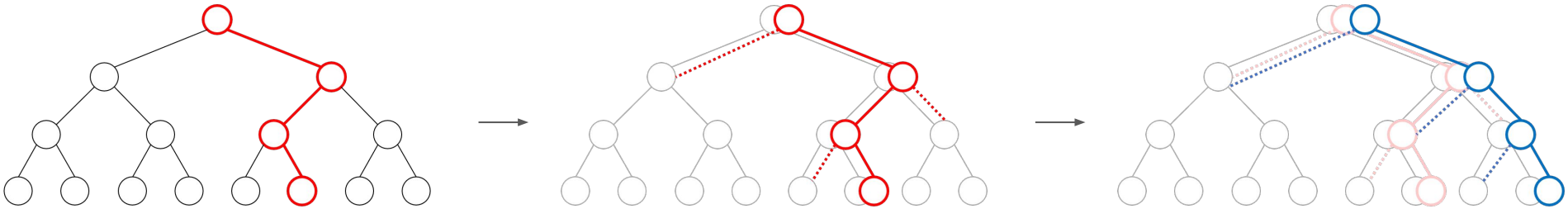
K-th Min Query: Not so Easy Version Again (and Again)

- Can we do this “online” without sorting the queries?
- We need to somehow maintain all N segment trees for each R...

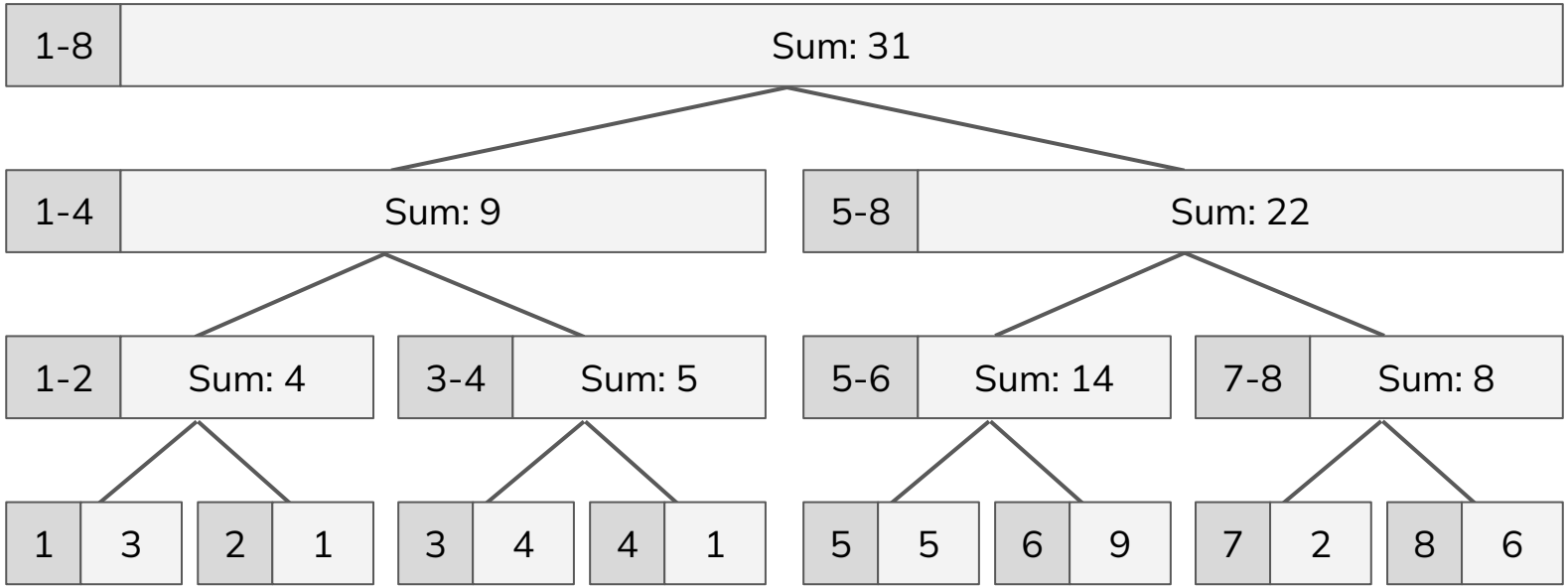
- Notice that every adjacent segment trees are “similar”
- They only differ by O(log N) nodes from one point update
- Can we reuse the other unchanged nodes?

Persistent Segment Tree

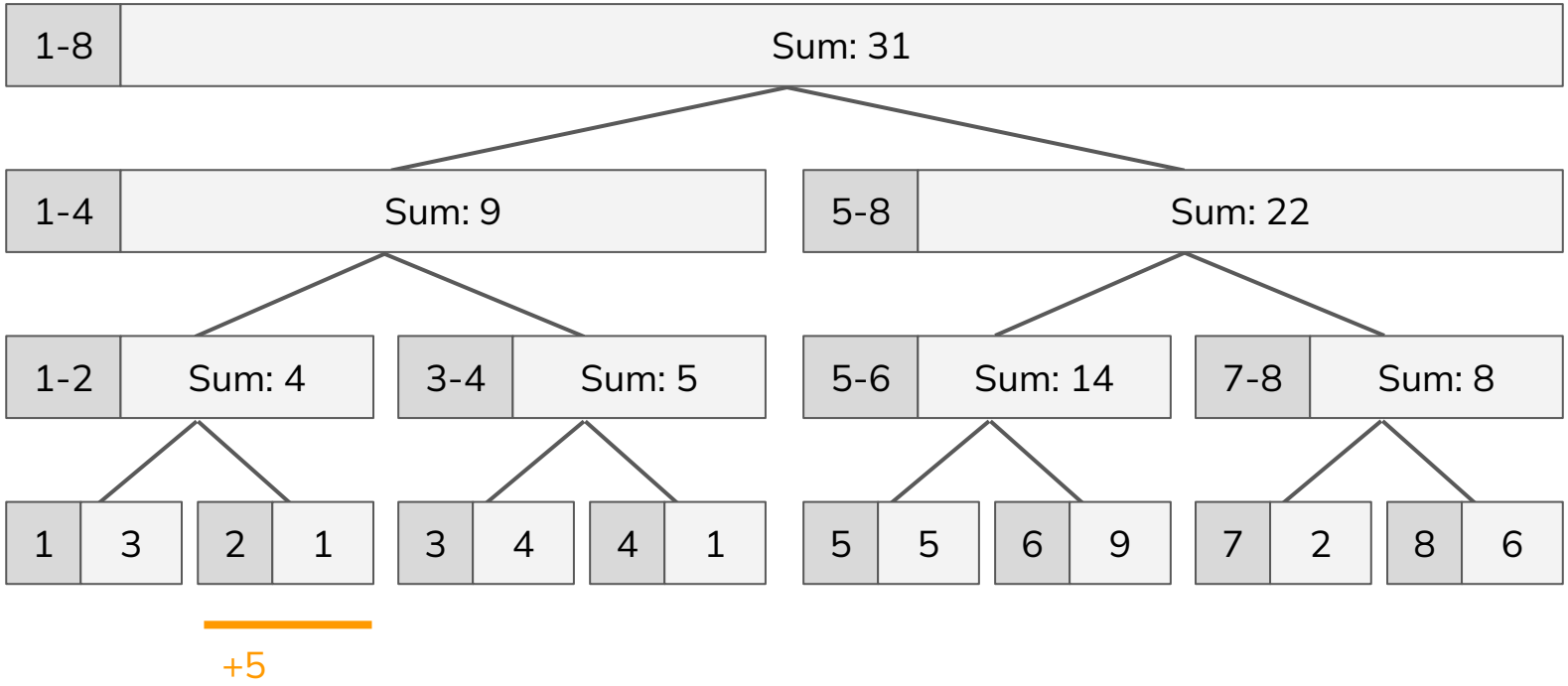
- Persistent segment tree builds new nodes for each update instead of updating existing nodes
- This preserves the data before updates, allowing us to “time travel” and view past versions of the segment tree



Persistent Segment Tree: Point Update

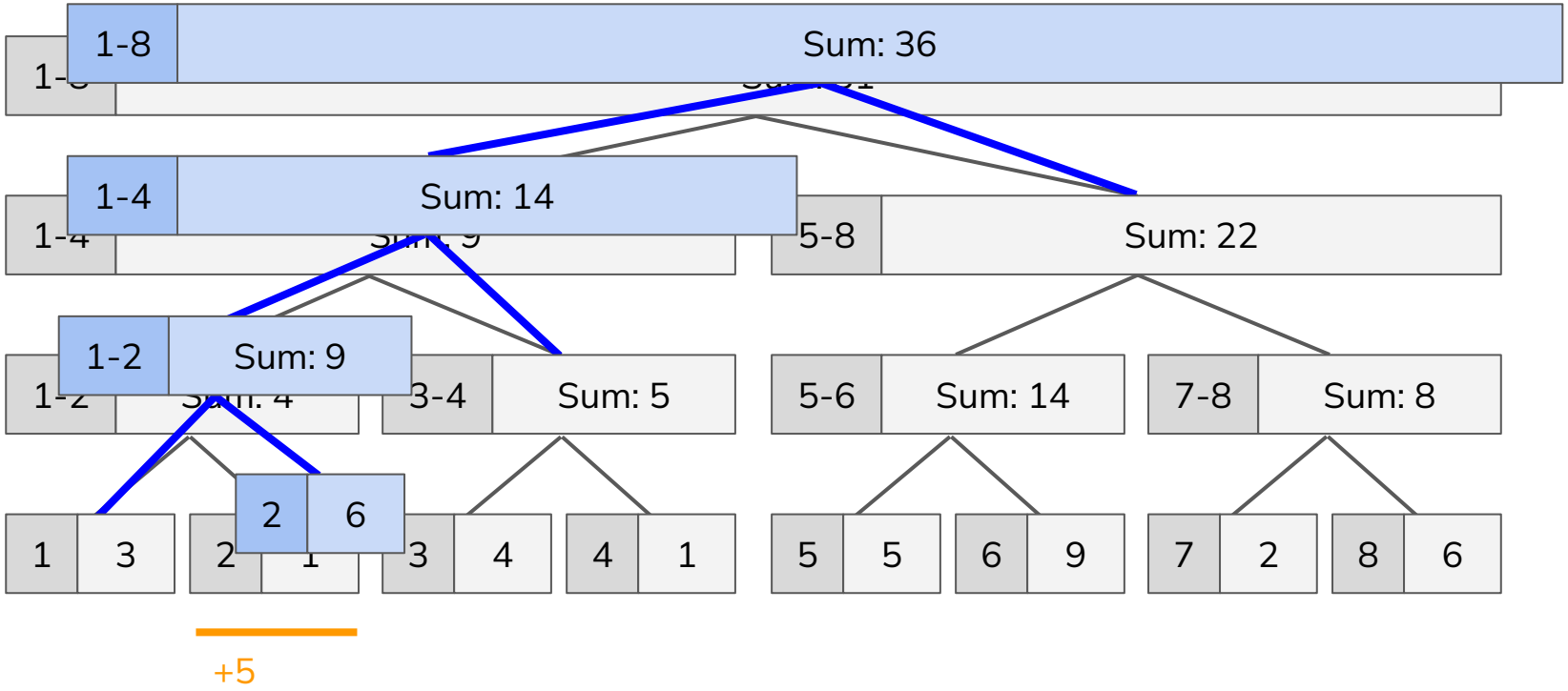


Persistent Segment Tree: Point Update

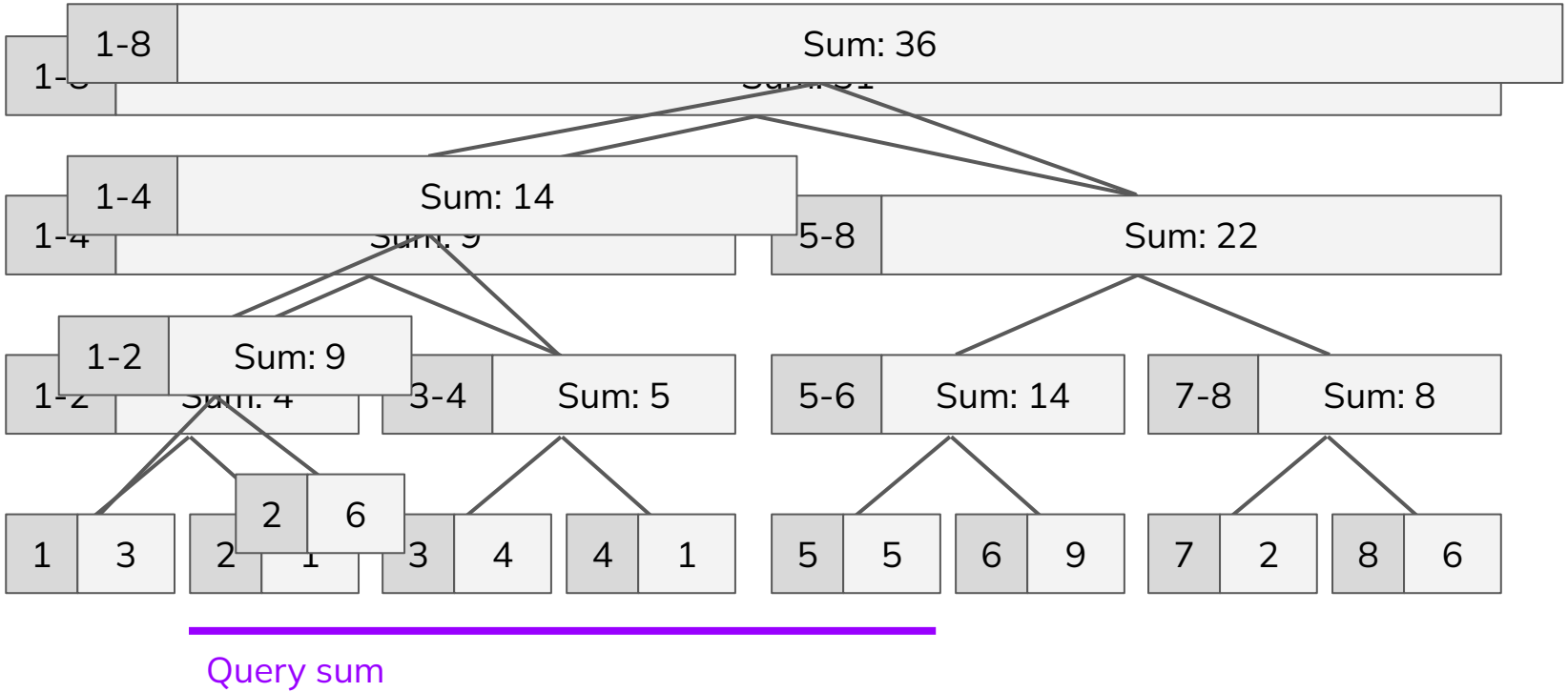


Persistent Segment Tree: Point Update

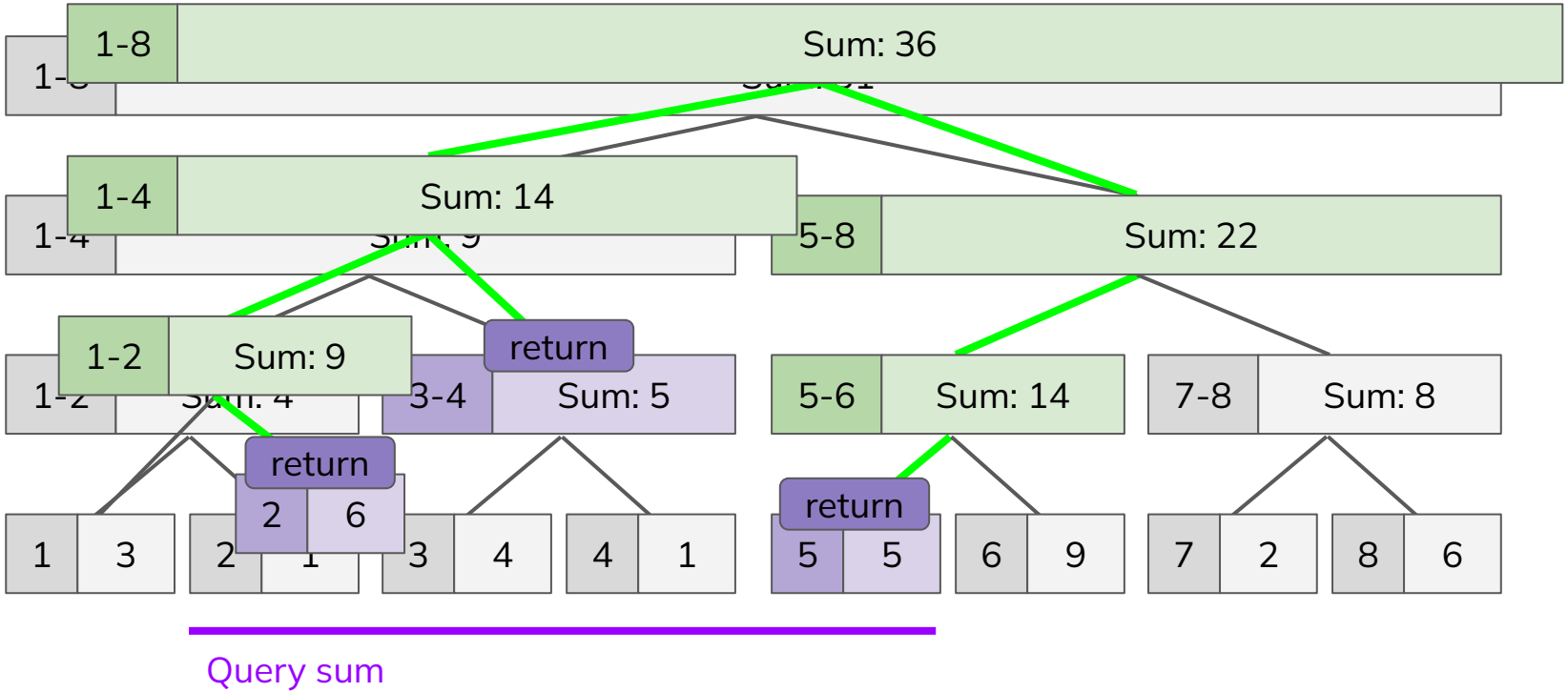
Blue nodes: new nodes



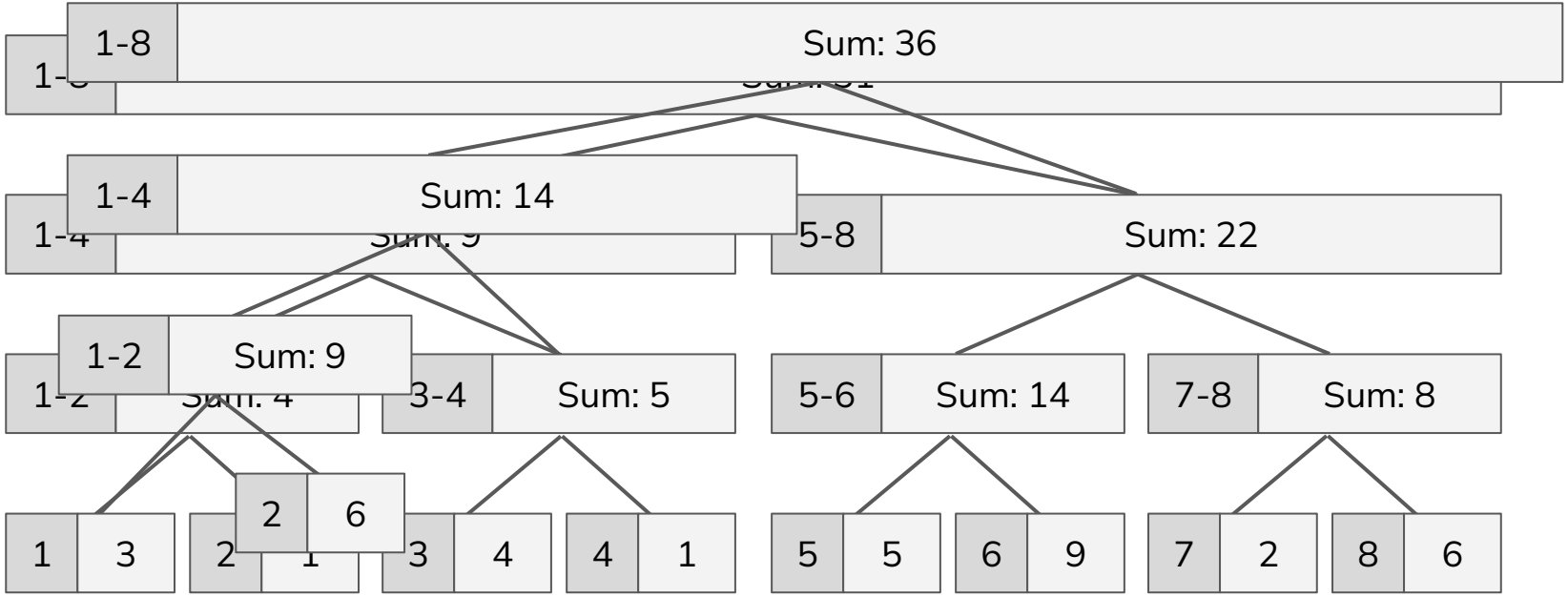
Persistent Segment Tree: Point Update



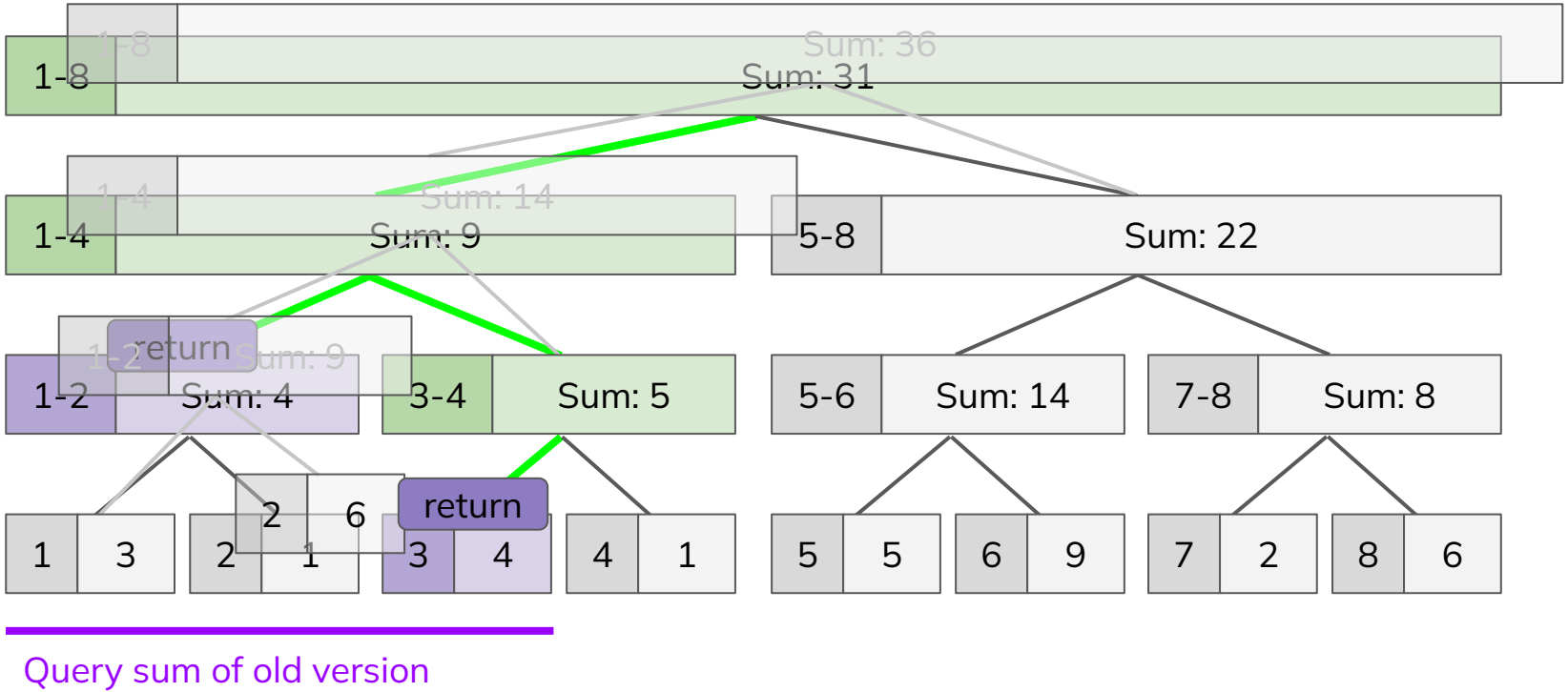
Persistent Segment Tree: Point Update



Persistent Segment Tree: Point Update



Persistent Segment Tree: Point Update



Persistent Segment Tree: Implementation

- For updates, we dynamically create new nodes when we traverse through the segment tree
- Notice that because we are creating new nodes, the children of node i will not always be node 2i and node (2i+1)
  - We have to additionally store the indices/pointer of the children for each node
- For queries, we can choose the root of the version of the segment tree we care about as the starting point
  - Notice that the root is always used in an update. Hence there will be a new root after each update

K-th Min Query: Hard Version

- With **persistent segment tree** and **binary search on segment tree**, we can solve the prefix version
- What if we also lift the L = 1 restriction? Is it still solvable in reasonable time?
- If only we can somehow obtain the cumulative frequency of [L, R]...

K-th Min Query: Hard Version

- Notice that the frequency of an element in [L, R] is just the frequency of it in [1, R] subtracted by the frequency of it in [1, L-1]
- The same goes for cumulative frequency
- Instead of binary searching on the segment tree after the R-th insertion, we can binary search on the **difference** between the R-th segment tree and the (L-1)-th segment tree
- You can try to implement it in A103 Range K-th Minimum Query

K-th Min Query: Even Harder Version

- New operation: set value of a_c to v
- Chronologically process updates and k-th min queries
- Left as exercise :)

Persistent Segment Tree: Practice Tasks

- A103 Range K-th Minimum Query
- M1842 Another RMQ
- APIO17 Land of the Rainbow Gold
- NOI18 歸程
- P4735 最大異或和 (Persistent trie!)
- P2617 Dynamic Rankings (Even harder version of k-th min query)

Acknowledgement

- Lecture slides referenced:
  - String Algorithms 2024 Lecture Notes
  - Data Structures (III) 2024 Lecture Notes
  - Data Structures (IV) 2024 Lecture Notes
  - Data Structures (IV) 2025 Lecture Notes
- Thanks Dickson Wong {hywong1} for preparing the slides