Data Structure (IV)

Luo Tsz Fung {pepper1208}

2025-08-01

Segment Tree

Segment Tree is an **extremely common** data structure being widely used in Olympiad in Informatics. It is used to maintain information of **intervals**.

There are ample number of variations of segment tree and its application.

Point Update, Range Query

Given N integers a_1, a_2, a_3, …, a_N. Implement a data structure to support the following operations:

1. Given i and k. Add k to a_i.
2. Given L and R. Output the sum of a_L, a_{L+1}, a_{L+2}, …, a_R.

Solution: Directly maintain an array to support the operations above.

However, a major drawback arises as the time complexity can be O(NQ) for Q queries in total.

Point Update, Range Query

Given N integers a_1, a_2, a_3, …, a_N. Implement a data structure to support the following operations:

1. Given i and k. Add k to a_i.
2. Given L and R. Output the sum of a_L, a_{L+1}, a_{L+2}, …, a_R.

Better Solution: Maintain a **segment tree**!

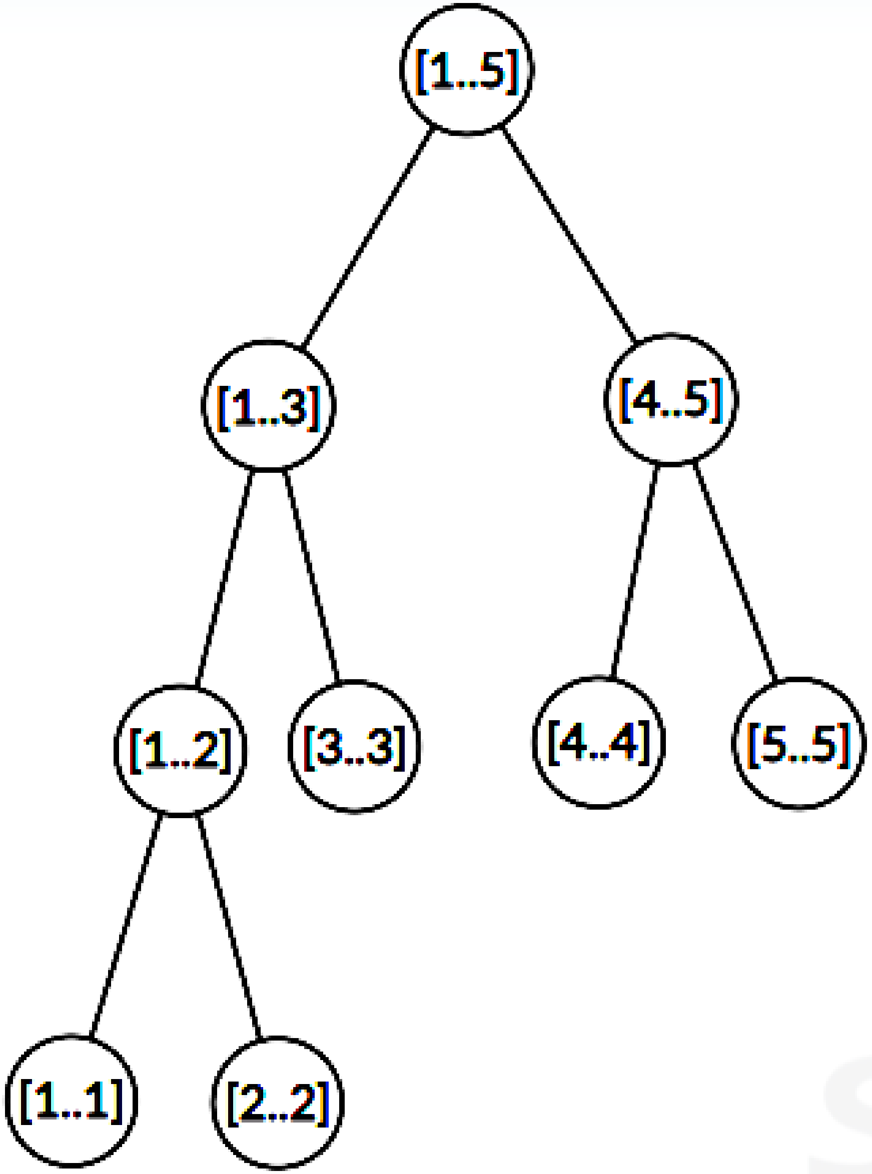
Segment Tree

Segment tree is a binary tree. Each node in the segment tree represents an "**interval**".

Assume the segment tree would like to maintain the interval $a[1..N]$.

The root node will represent $a[1..N]$. Then, for each node representing $a[L..R]$ with $L < R$, its left child will represent $a[L..M]$ and its right child will represent $a[M+1..R]$, where $M = (L + R) / 2$.

Example: $N = 5$



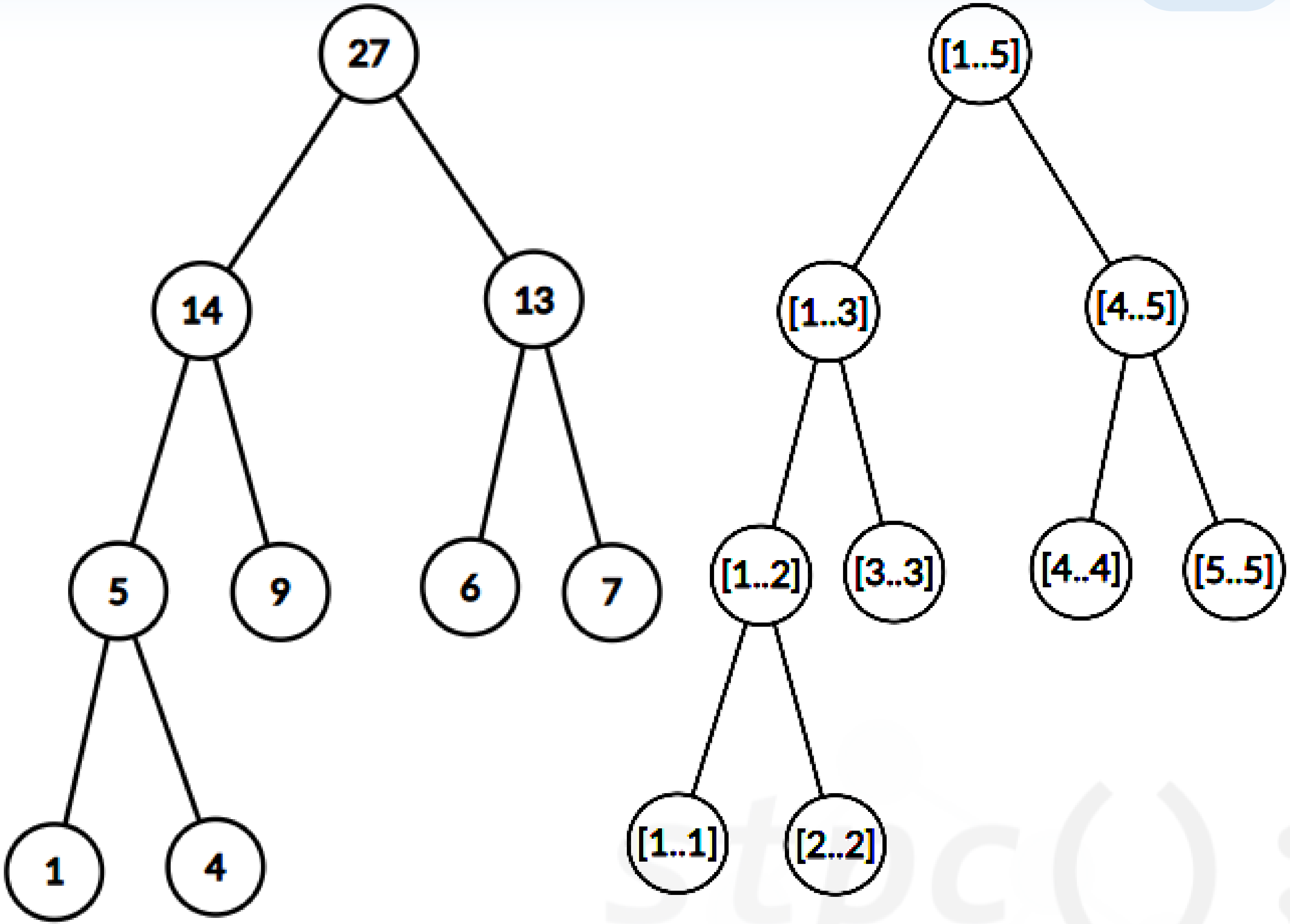
Segment Tree

Each node in the segment tree will store the "desired value(s)" of its interval.

In this case, the node representing $[L..R]$ will store the sum of a_L, a_{L+1}, a_{L+2}, …, a_R.

Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$



Segment Tree

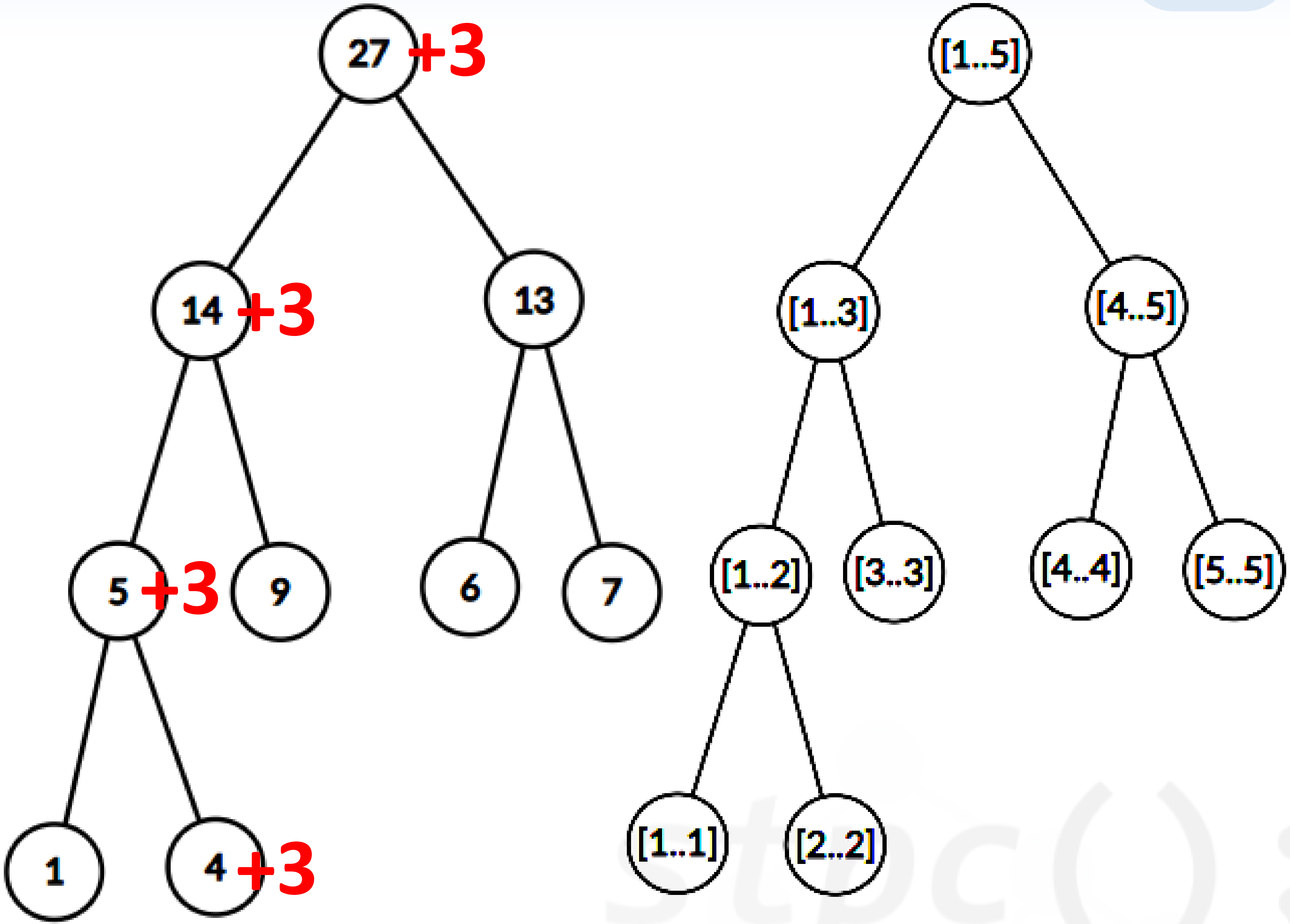
Operation 1: Given i and k. Add k to a_i.

We will process the query from the root of the segment tree and add k to the root. Then, find the interval from its child whether i lies in. Then, recursively update the child node and find the smaller interval from the node.

Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

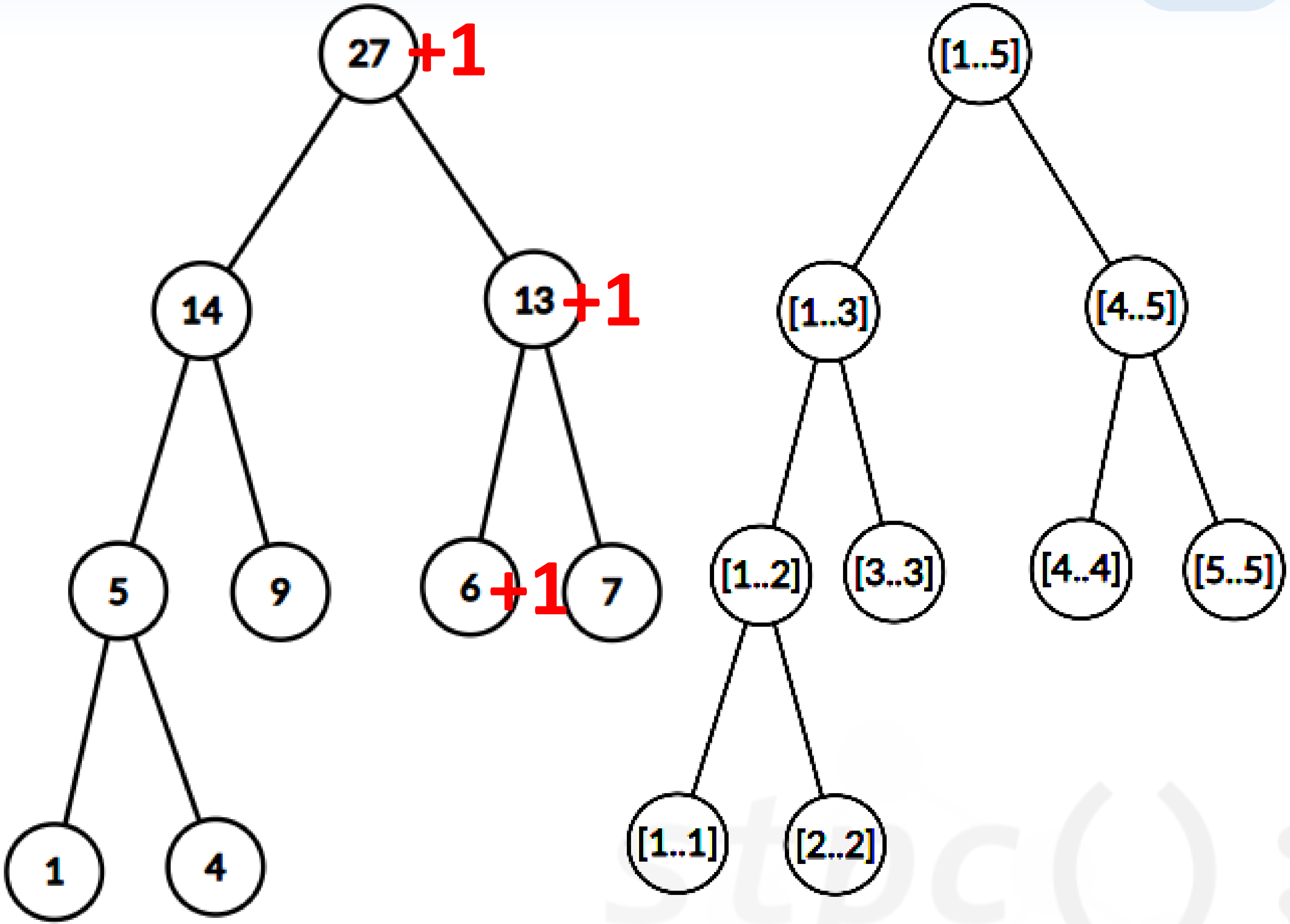
Add 3 to a_2.



Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

Add 1 to a_4.



Segment Tree

Operation 2: Given L and R. Output the sum of $a_L, a_{L+1}, a_{L+2}, \ldots, a_R$.

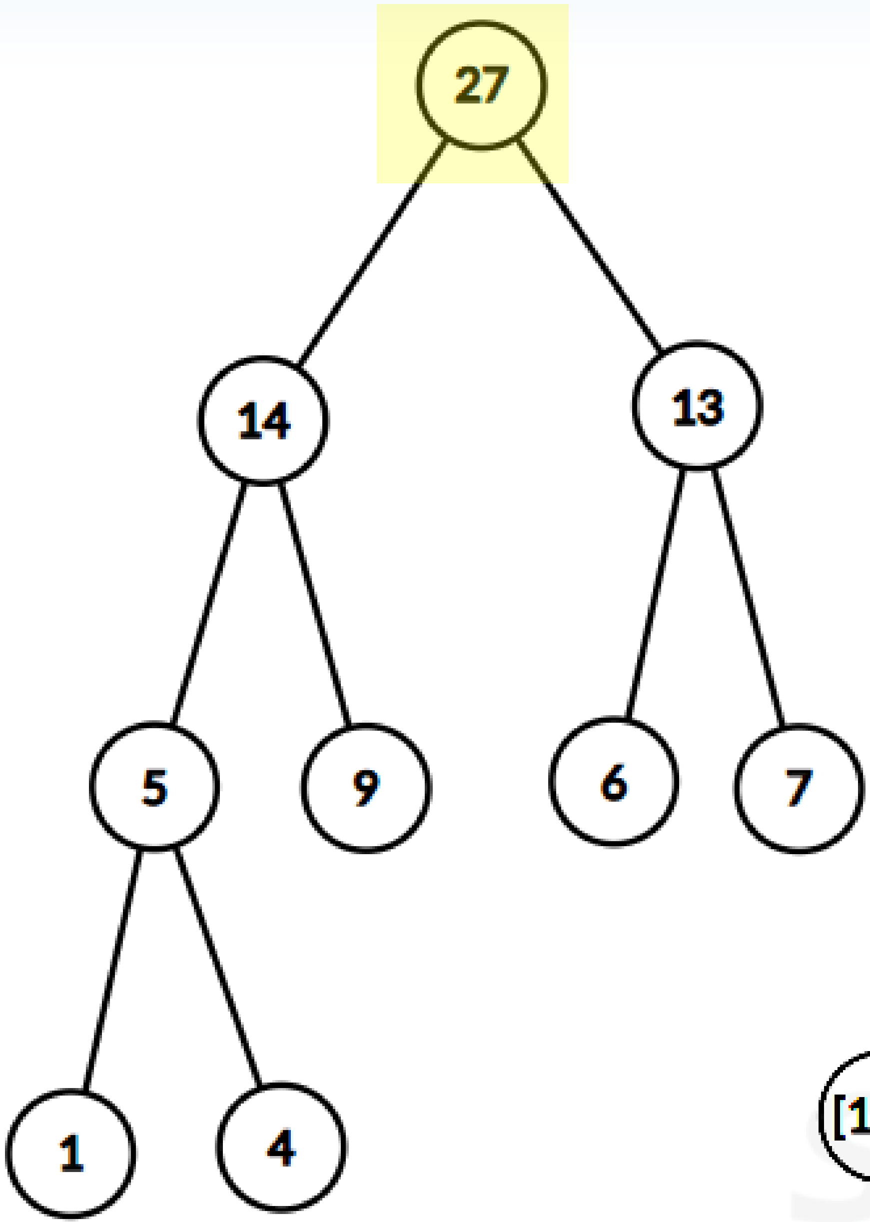
We are going to find the desired interval from the segment tree. Starting from the root node, search recursively:

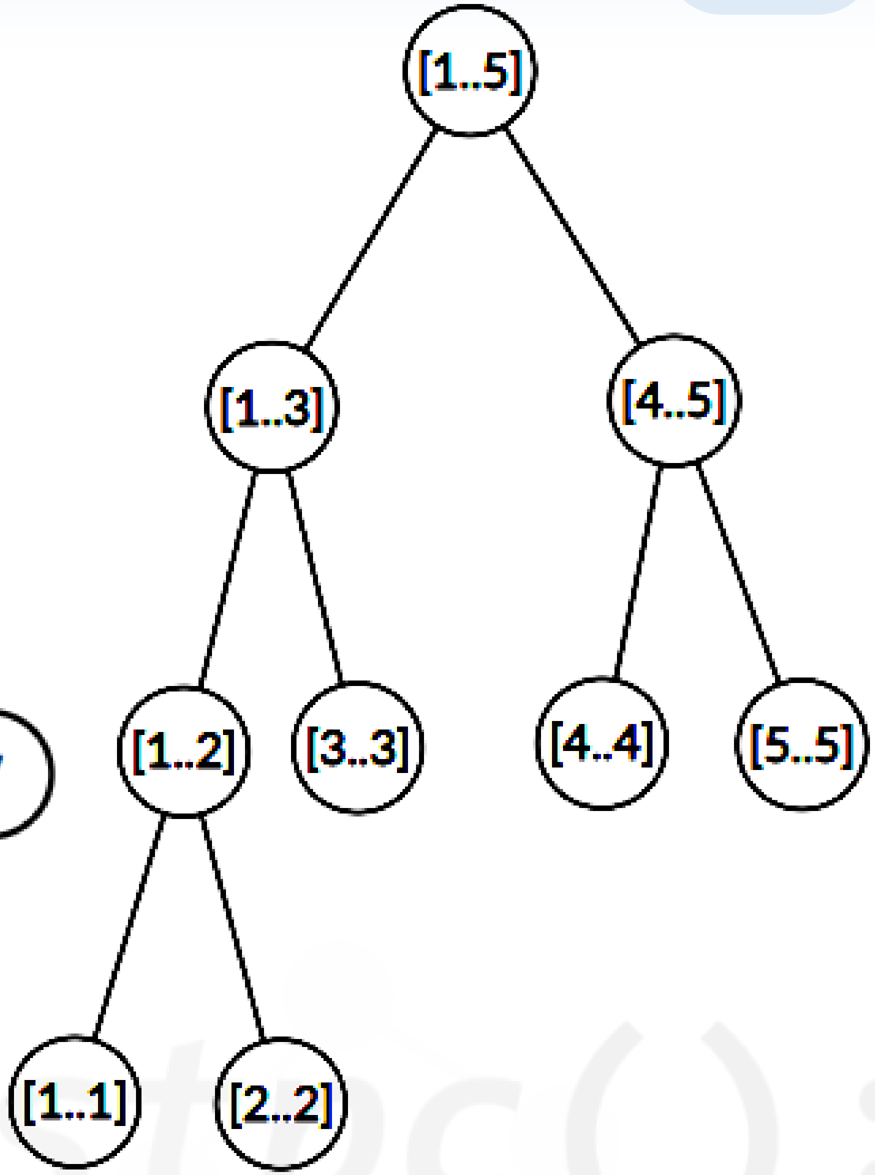
- If the interval lies in the desired interval, stop searching and return the value of the interval.
- If the interval is disjoint from the desired interval, stop searching and return nothing.
- Otherwise, search its children and return the result.

Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

Find sum of $a[2..5]$.

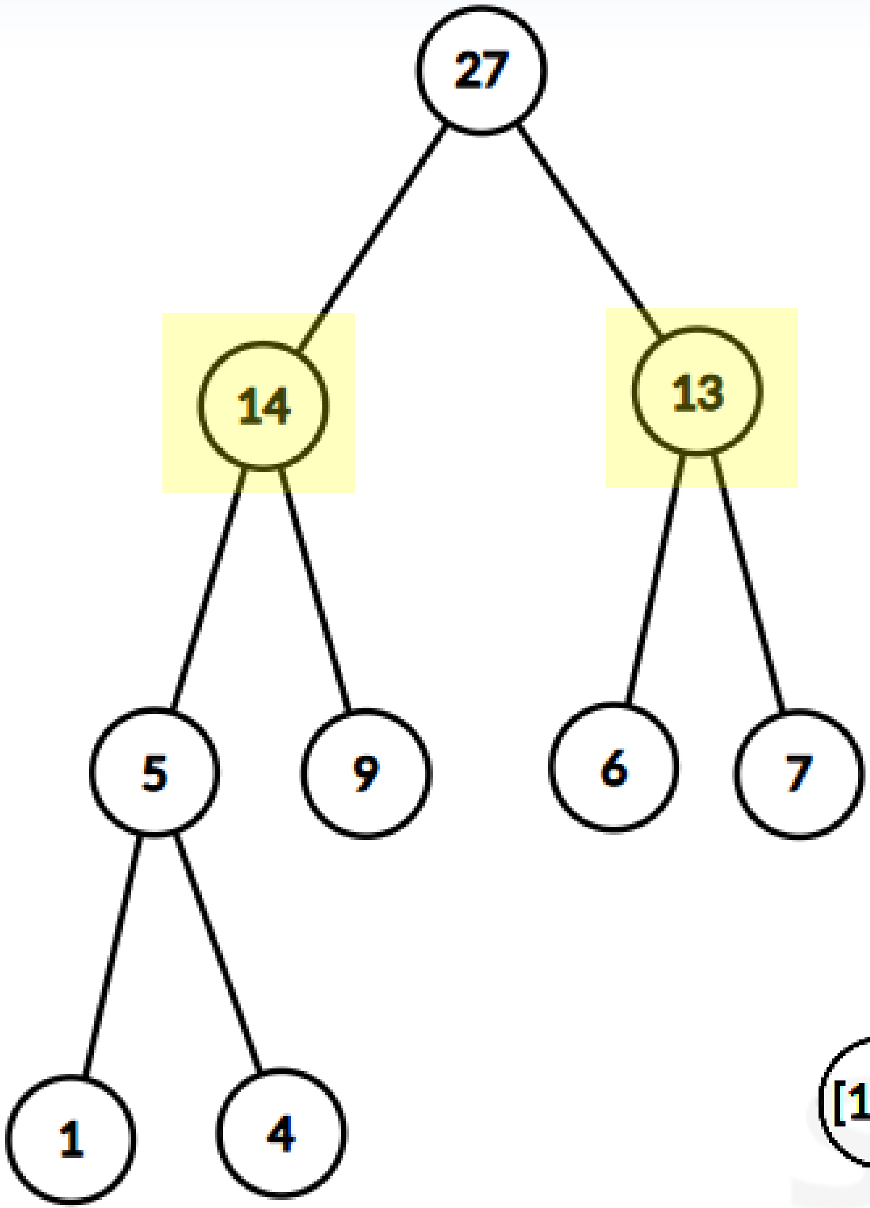


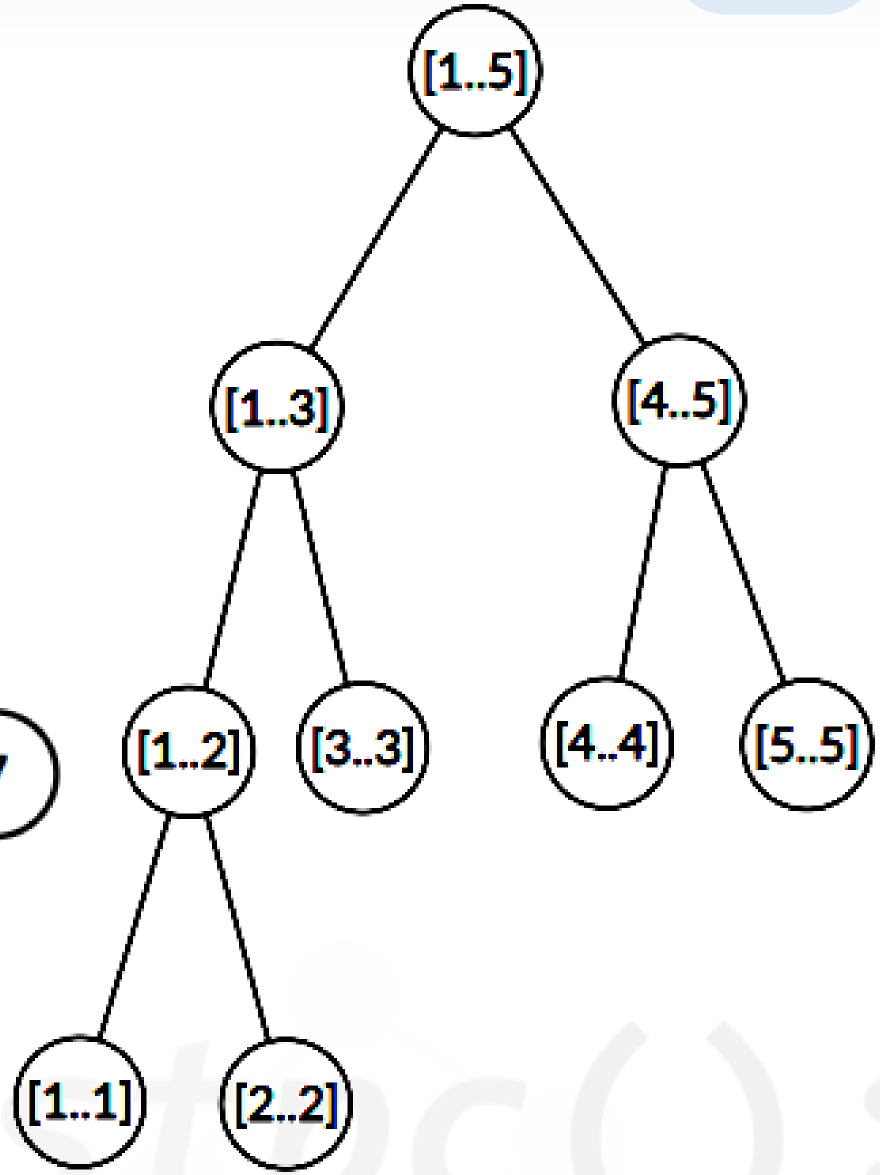


Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

Find sum of $a[2..5]$.

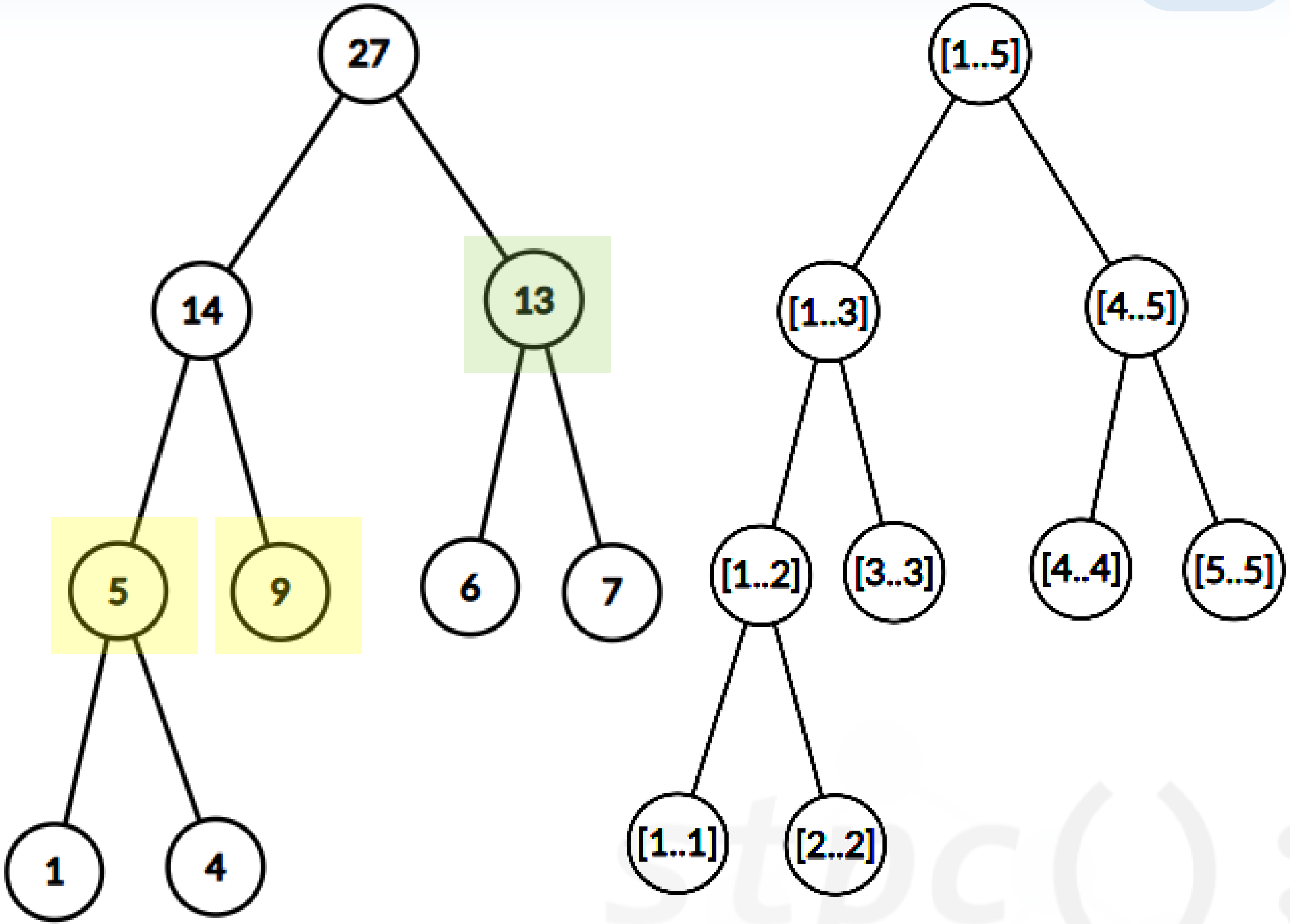




Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

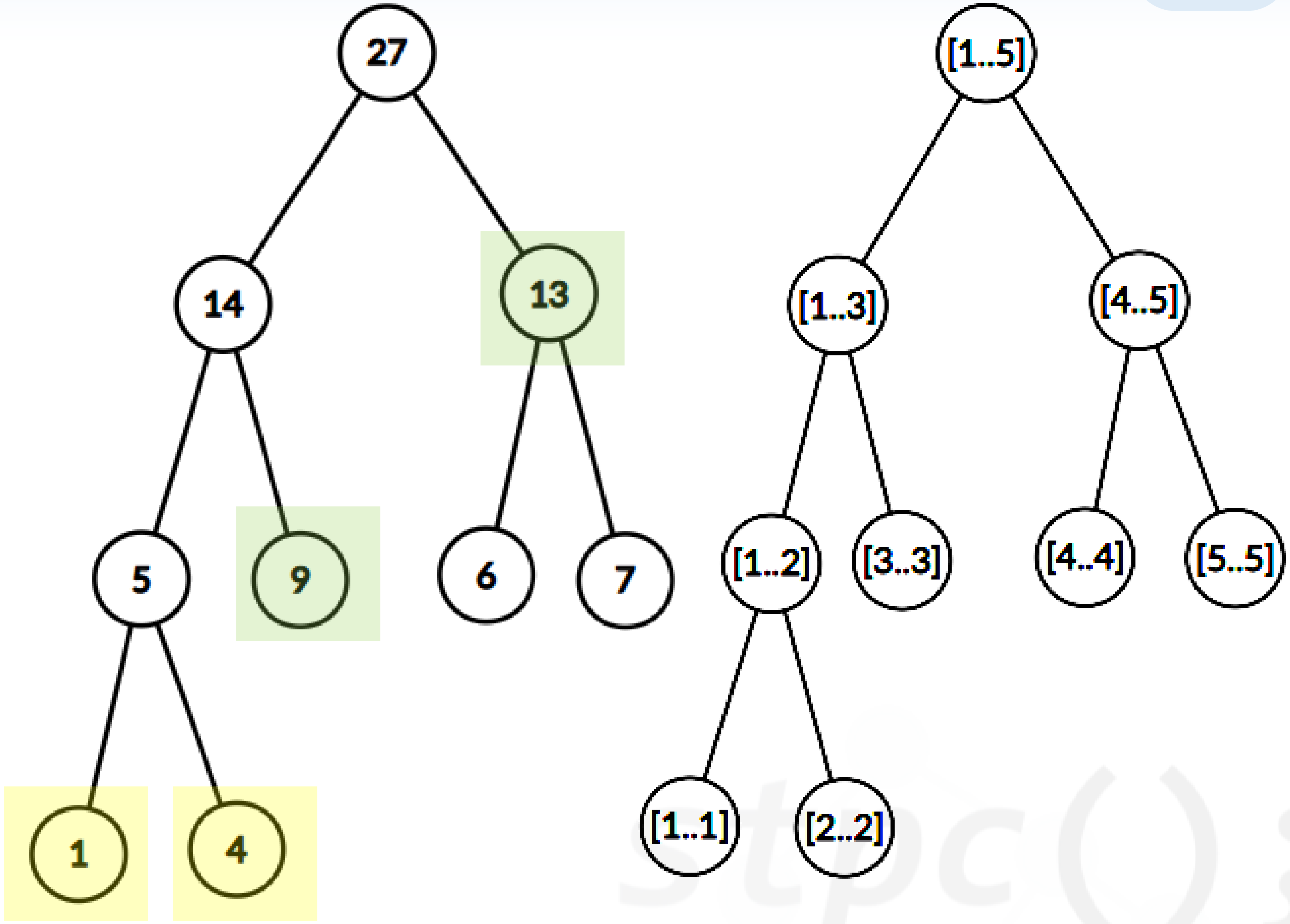
Find sum of $a[2..5]$.



Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

Find sum of $a[2..5]$.

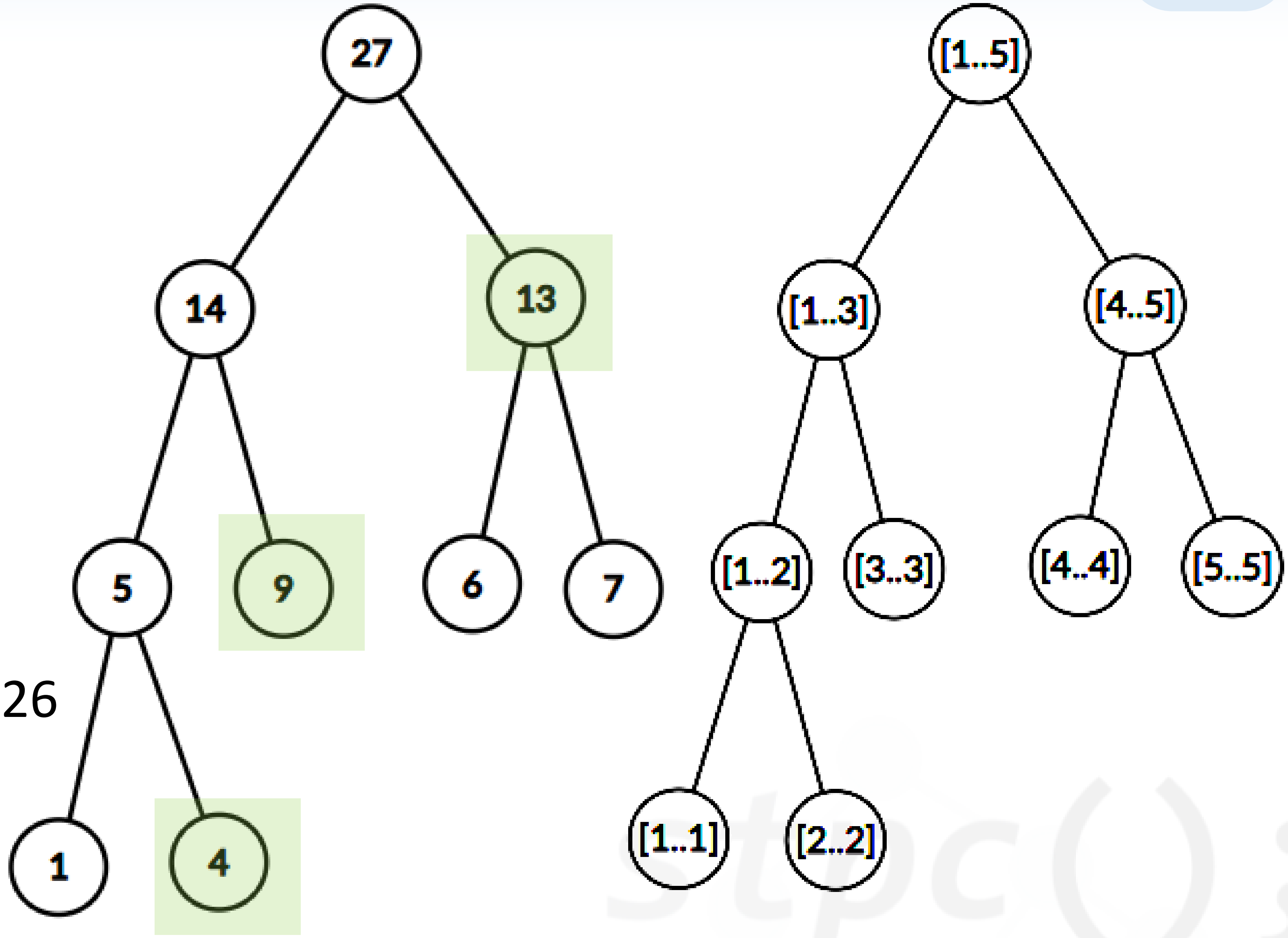


Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

Find sum of $a[2..5]$.

Result = 4 + 9 + 13 = 26



Segment Tree

Note that the number of nodes to maintain an interval with length N will not exceed O(N). Therefore, the height of the segment tree must be O(log N).

Operation 1: **O(log N)**

Operation 2: **O(log N)** not O(N)! Exercise is left as exercise.

For Q queries, the time complexity is O(Q log N) with space complexity O(N).

Implementation

```
build(id, L, R)
  if L = R
    Node[id] ← A[L]
    return
  mid ← (L + R) / 2
  build(id * 2, L, mid)
  build(id * 2 + 1, mid + 1, R)
  Node[id] ← Node[id * 2] + Node[id * 2 + 1]

build(1, 1, N)
```

Implementation

```
query(id, L, R, QL, QR) // range [L, R], query range [QL, QR]
  if QR < L or R < QL // no intersection between [L, R] & [QL, QR]
    return 0
  if QL ≤ L and R ≤ QR // [L, R] is fully inside [QL, QR]
    return Node[id]
  mid ← (L + R) / 2
  return query(id * 2, L, mid, QL, QR) + query(id * 2 + 1, mid + 1, R, QL,
QR)

query(1, 1, N, QL, QR)
```

Implementation

```
update(id, L, R, x, val)
  if L = R
    Node[id] ← val
    return
  mid ← (L + R) / 2
  if x ≤ mid
    update(id * 2, L, mid, x, val)
  else
    update(id * 2 + 1, mid + 1, R, x, val)
  Node[id] ← Node[id * 2] + Node[id * 2 + 1]

update(1, 1, N, x, val)
```

Segment Tree

You can store prefix min / max, hash sum, dp table, etc. as long as the operation satisfies some properties of monoid.

Range Update, Range Query

Given N integers a_1, a_2, a_3, …, a_N. Implement a data structure to support the following operations:

1. Given L, R and k. Add k to a_L, a_{L+1}, a_{L+2}, …, a_R.
2. Given L and R. Output the sum of a_L, a_{L+1}, a_{L+2}, …, a_R.

Solution: No problem! Maintain a segment tree.

However, the time complexity of operation 1 is O(N log N)! The overall time complexity can be O(Q N log N) for Q queries!

Lazy propagation

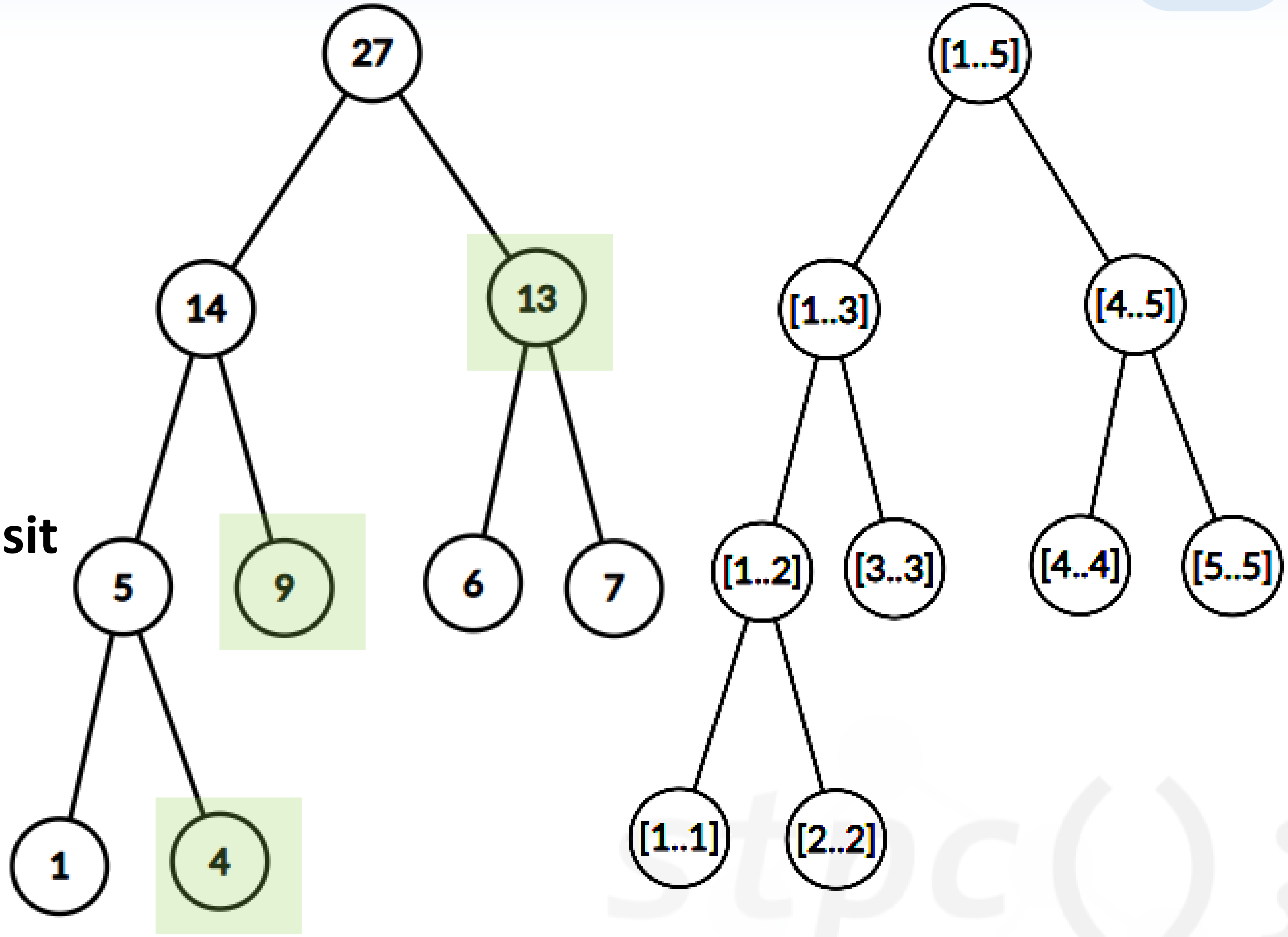
Let’s revisit the process for operation 2.

Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

Find sum of $a[2..5]$.

Actually, we wont visit [4..4] and [5..5].



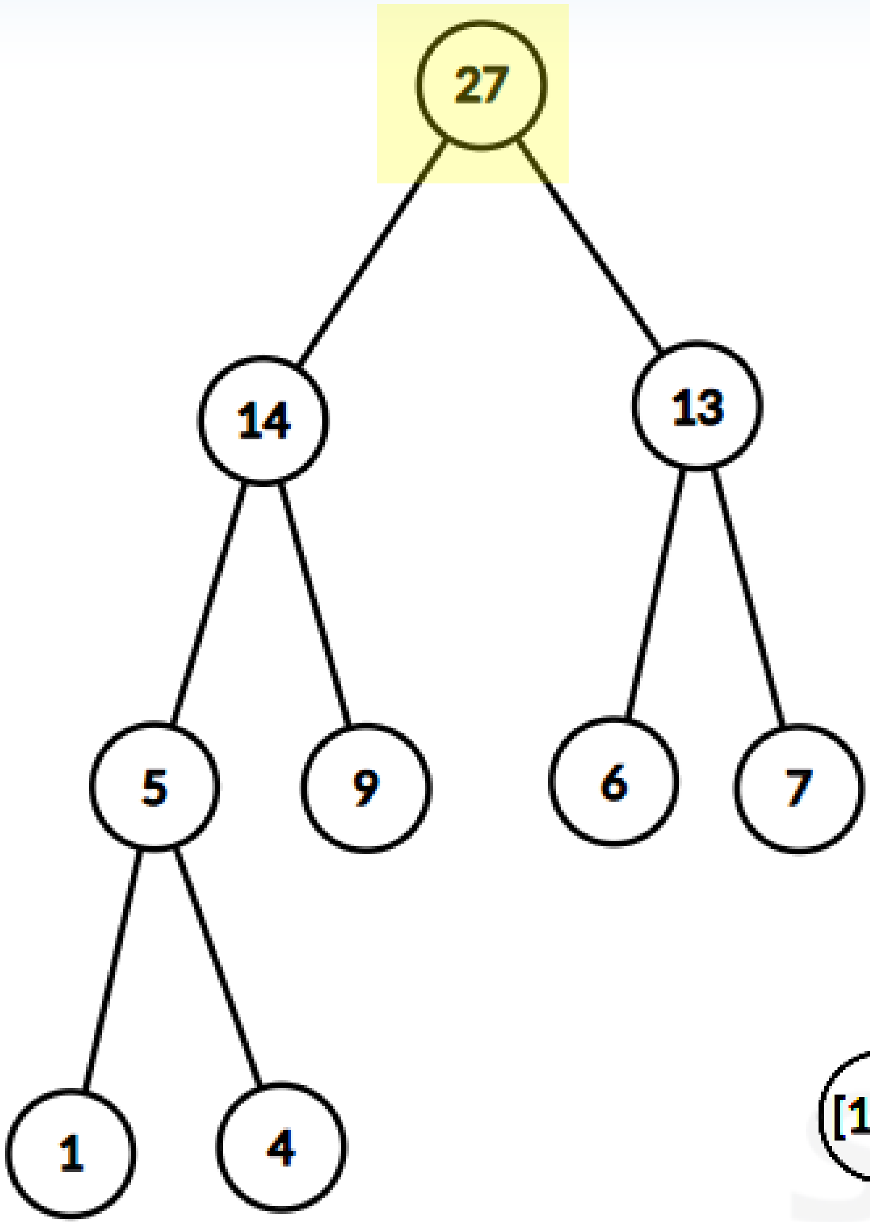
Lazy propagation

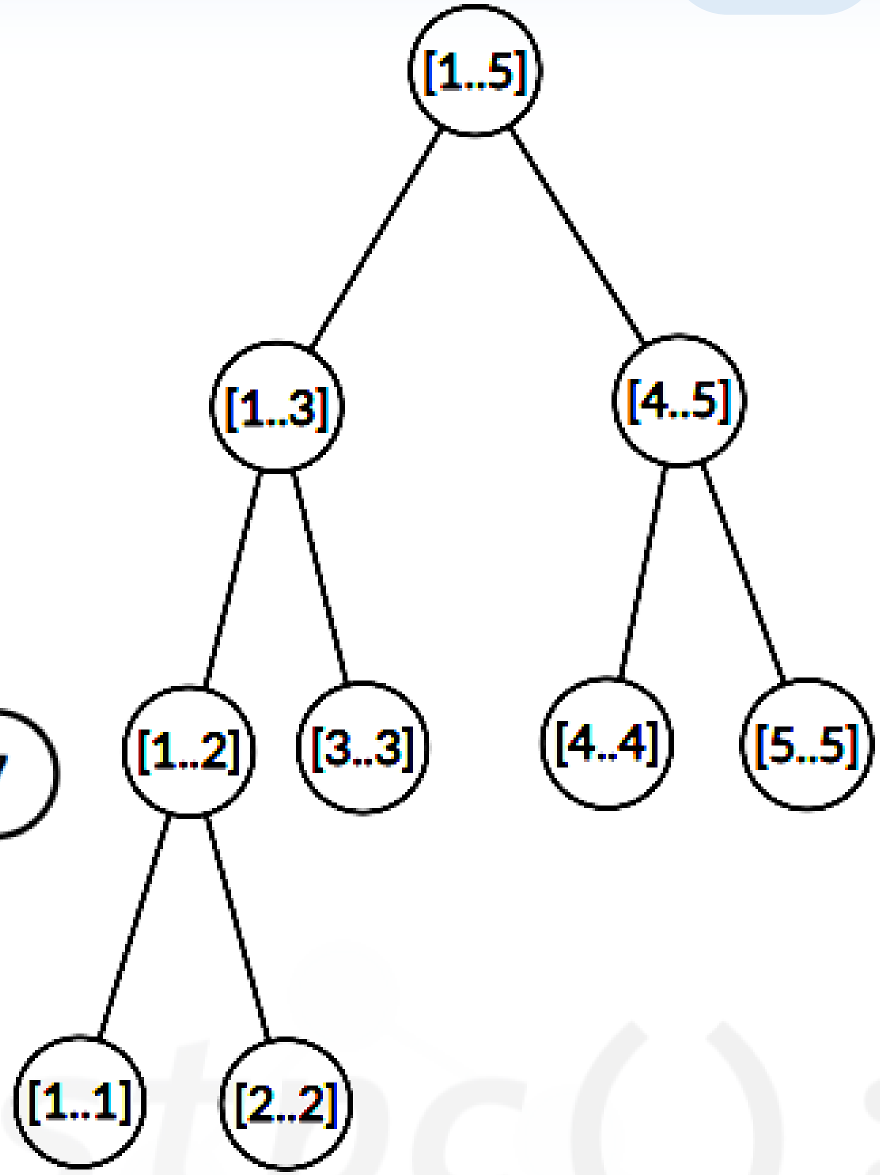
We can use the "early return" idea to implement a better version of operation 1 with the idea of **lazy tag**.

Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

Add 3 to $a[2..5]$.

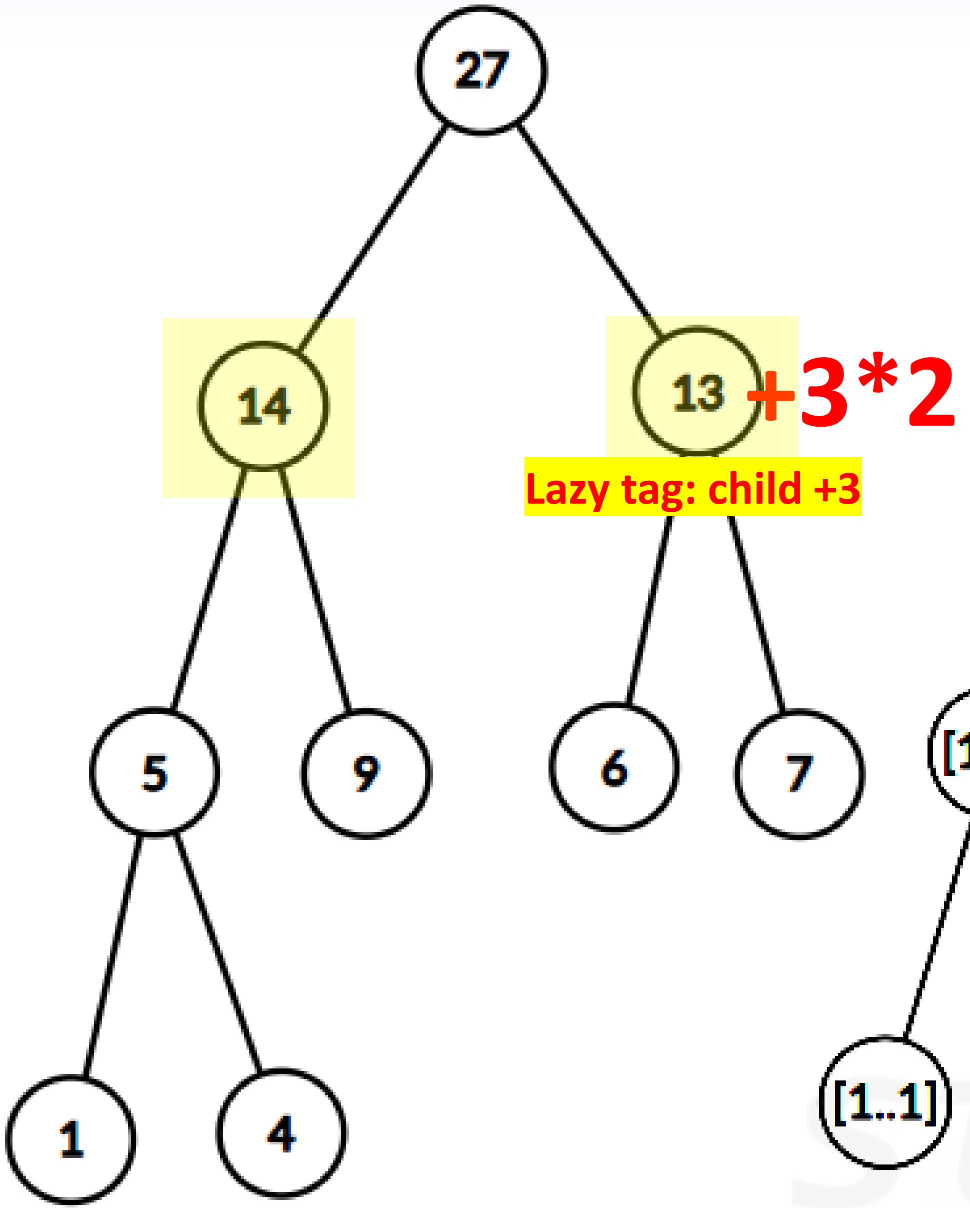


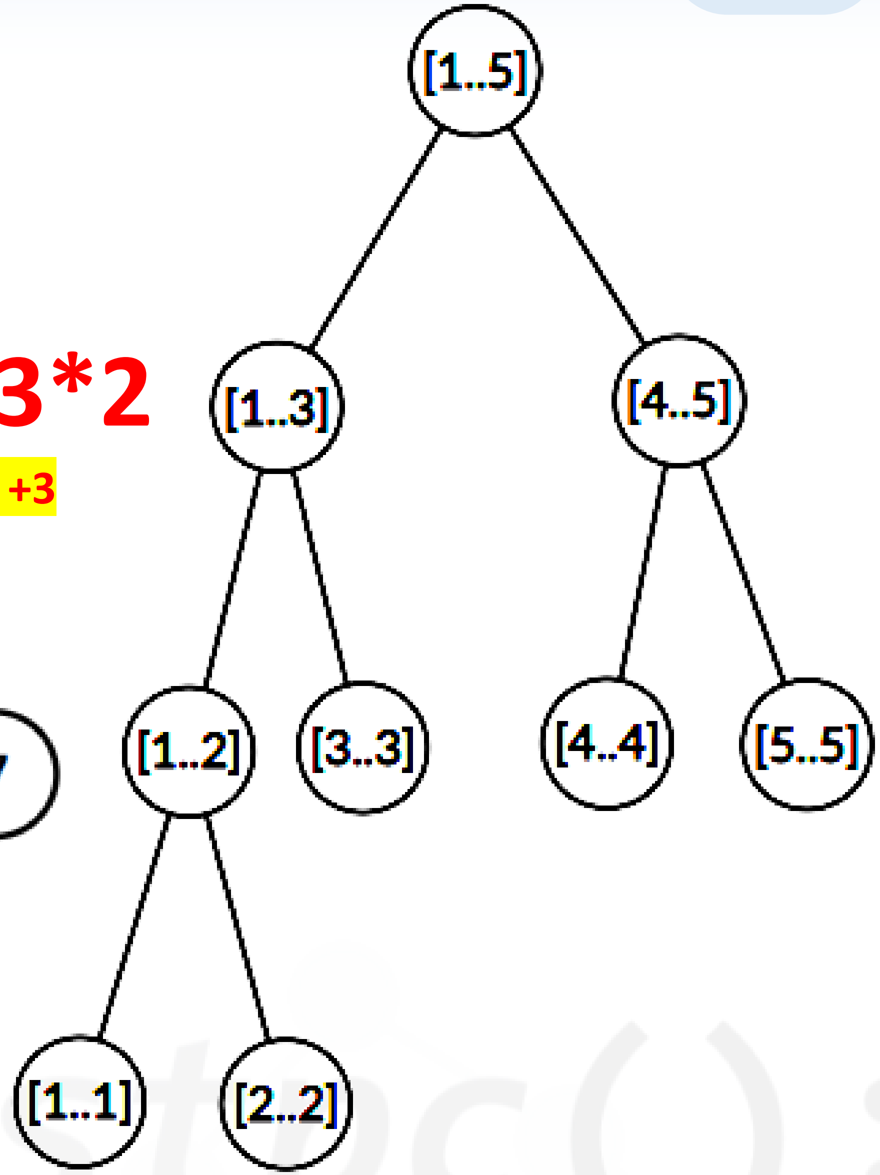


Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

Add 3 to $a[2..5]$.

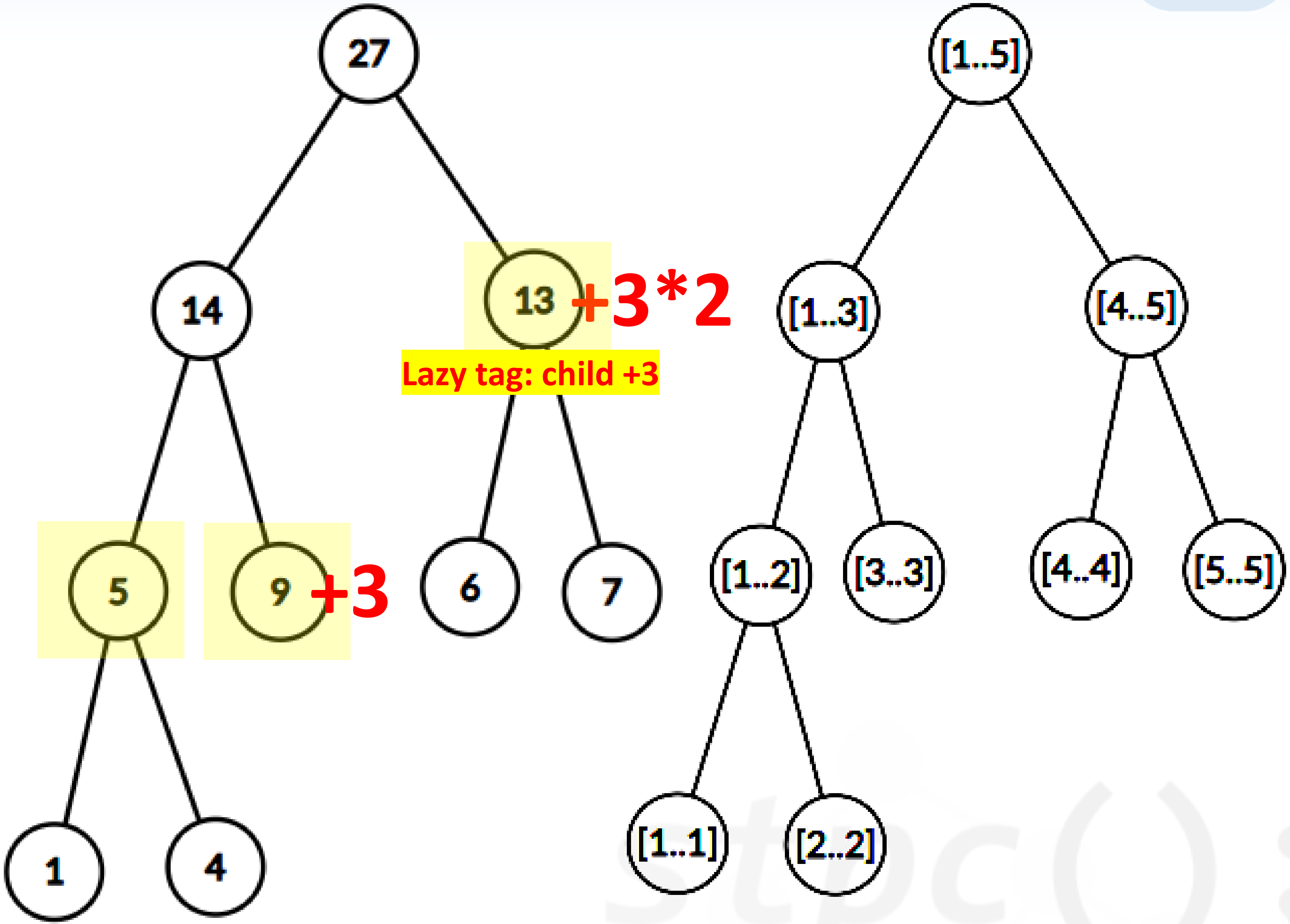




Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

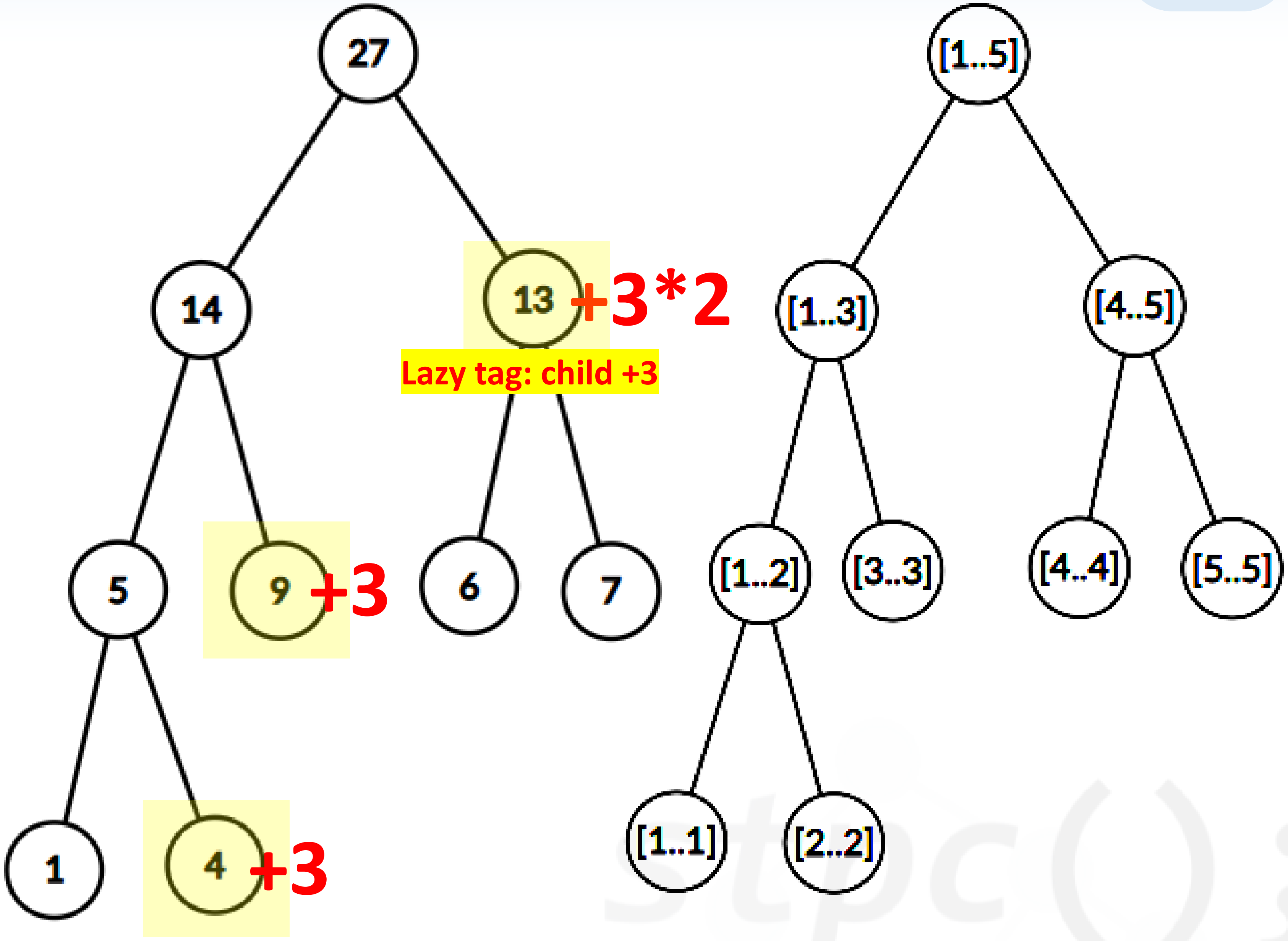
Add 3 to $a[2..5]$.



Example: $N = 5$

$a = \{1, 4, 9, 6, 7\}$

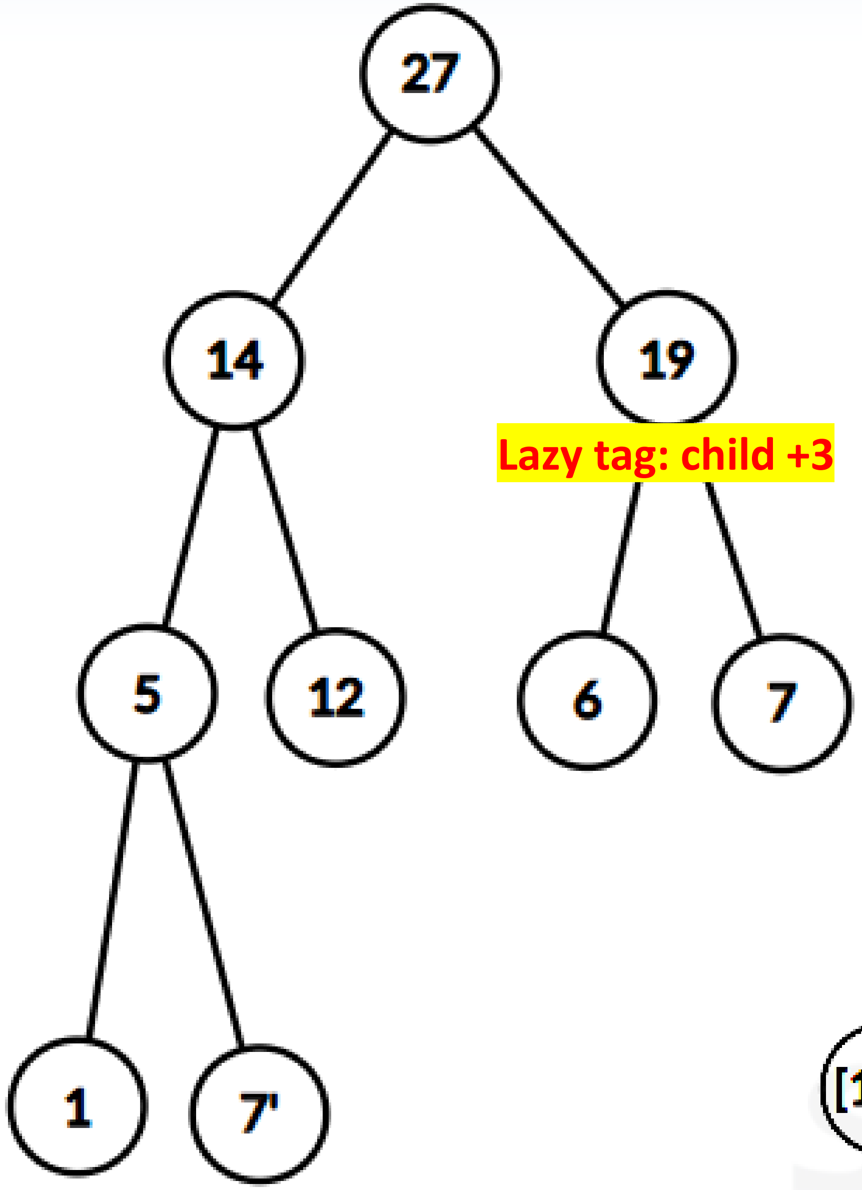
Add 3 to $a[2..5]$.

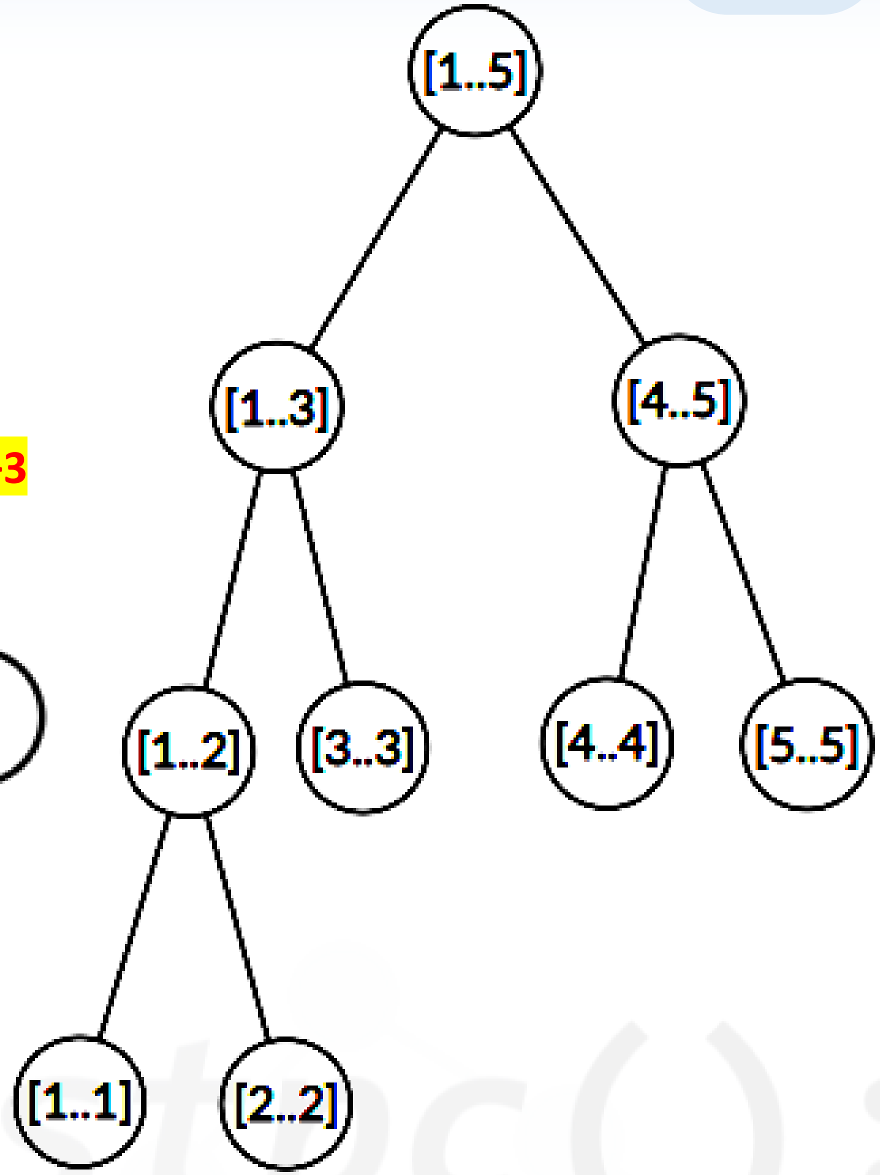


Example: $N = 5$

$a = \{1, 7, 12, 9, 10\}$

Add 3 to $a[2..5]$.





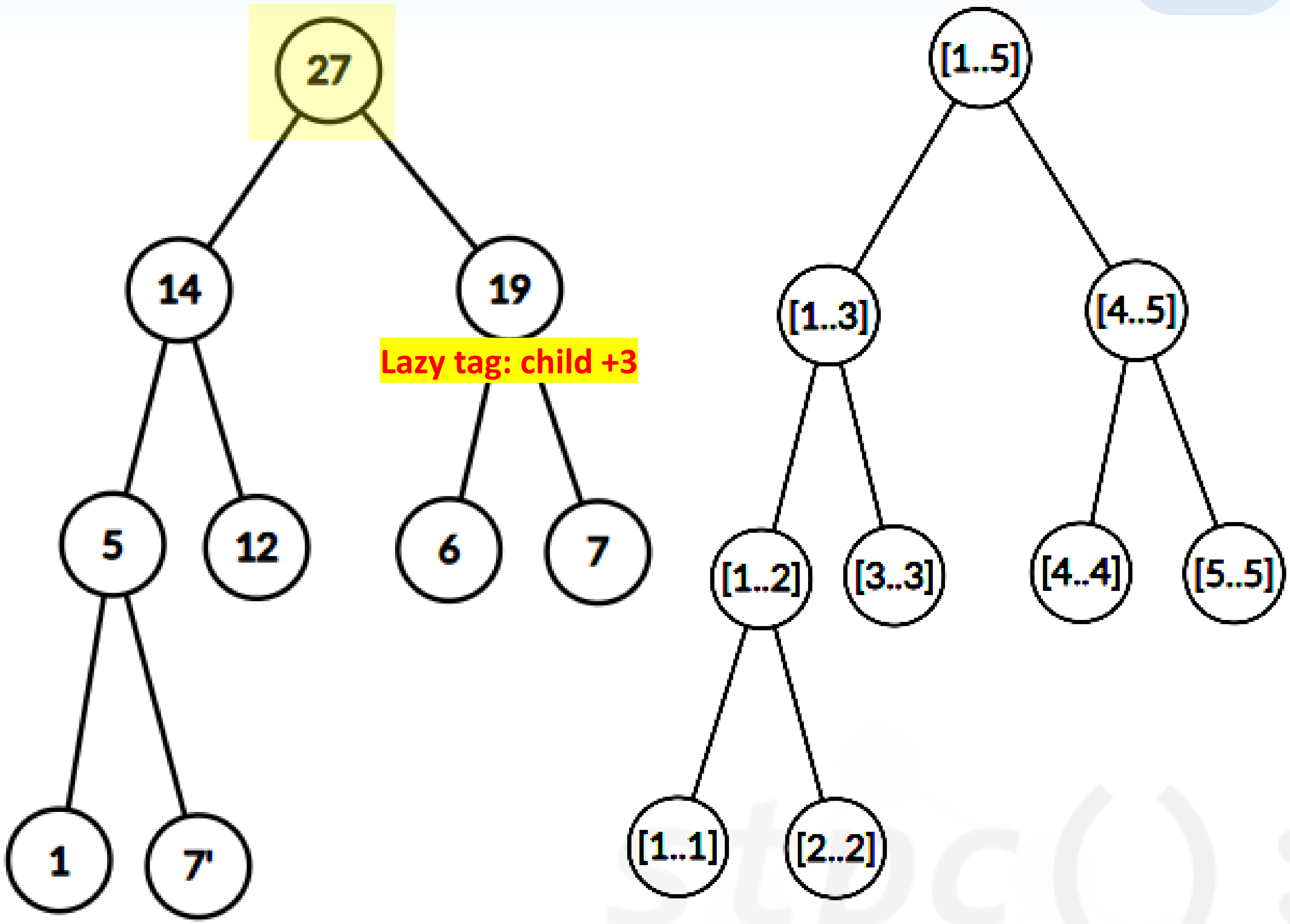
Lazy propagation

For any operations afterwards, if the node with lazy tag is visited, the lazy tag will be executed.

Example: $N = 5$

$a = \{1, 7, 12, 9, 10\}$

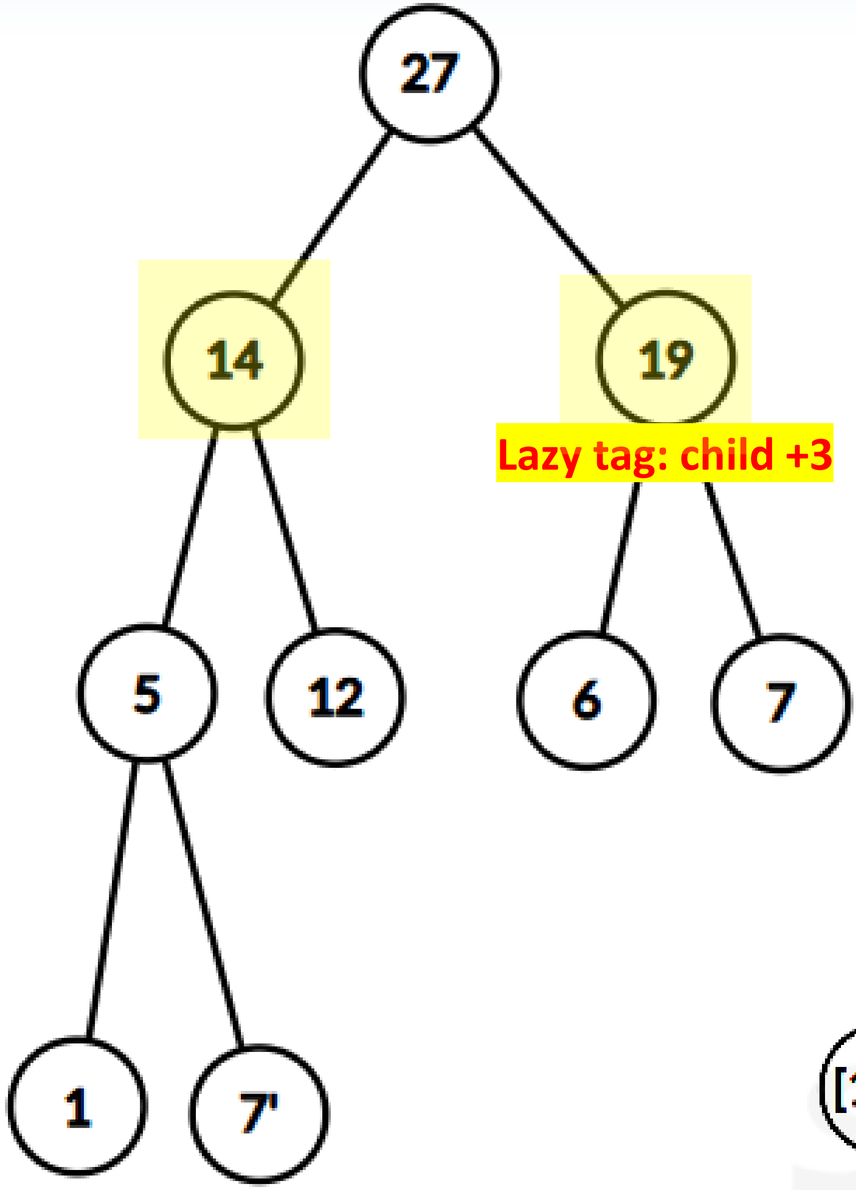
Find sum of $a[3..4]$.

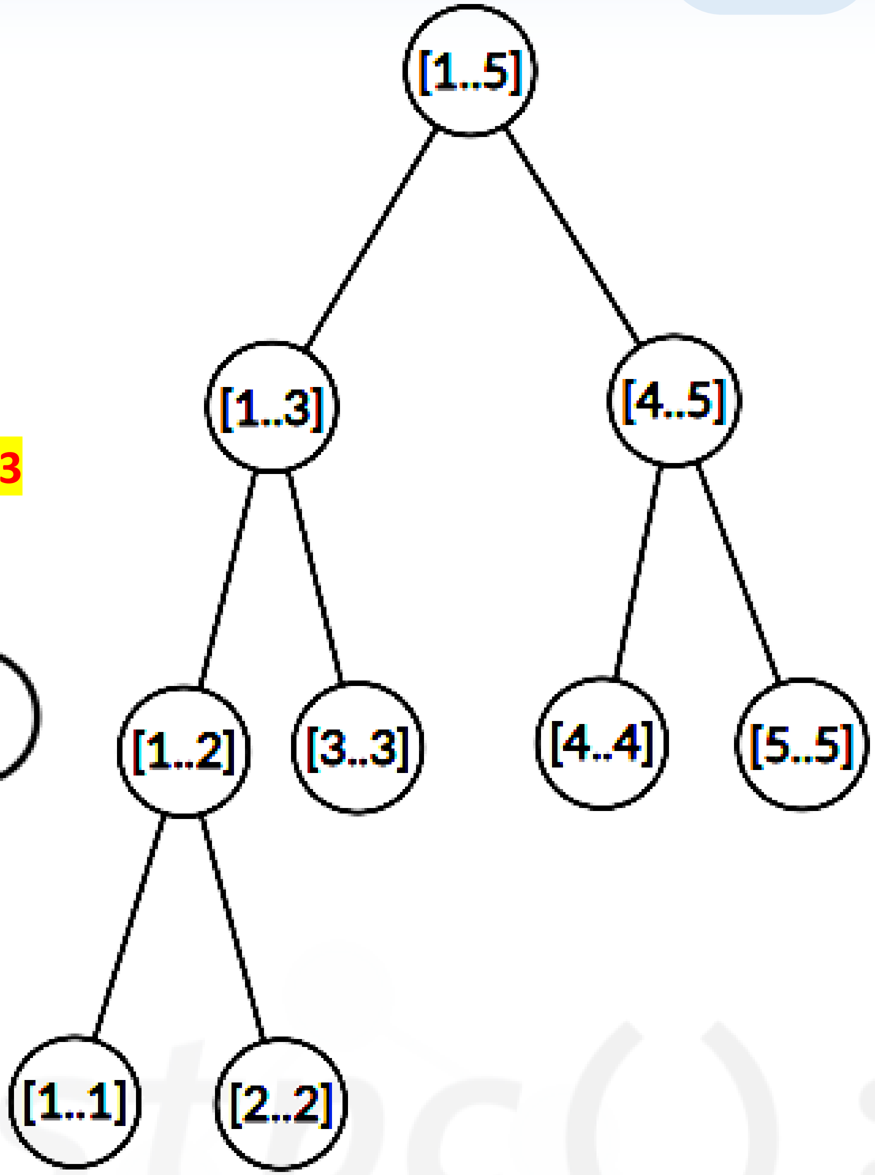


Example: N = 5

a = {1, 7, 12, 9, 10}

Find sum of a[3..4].

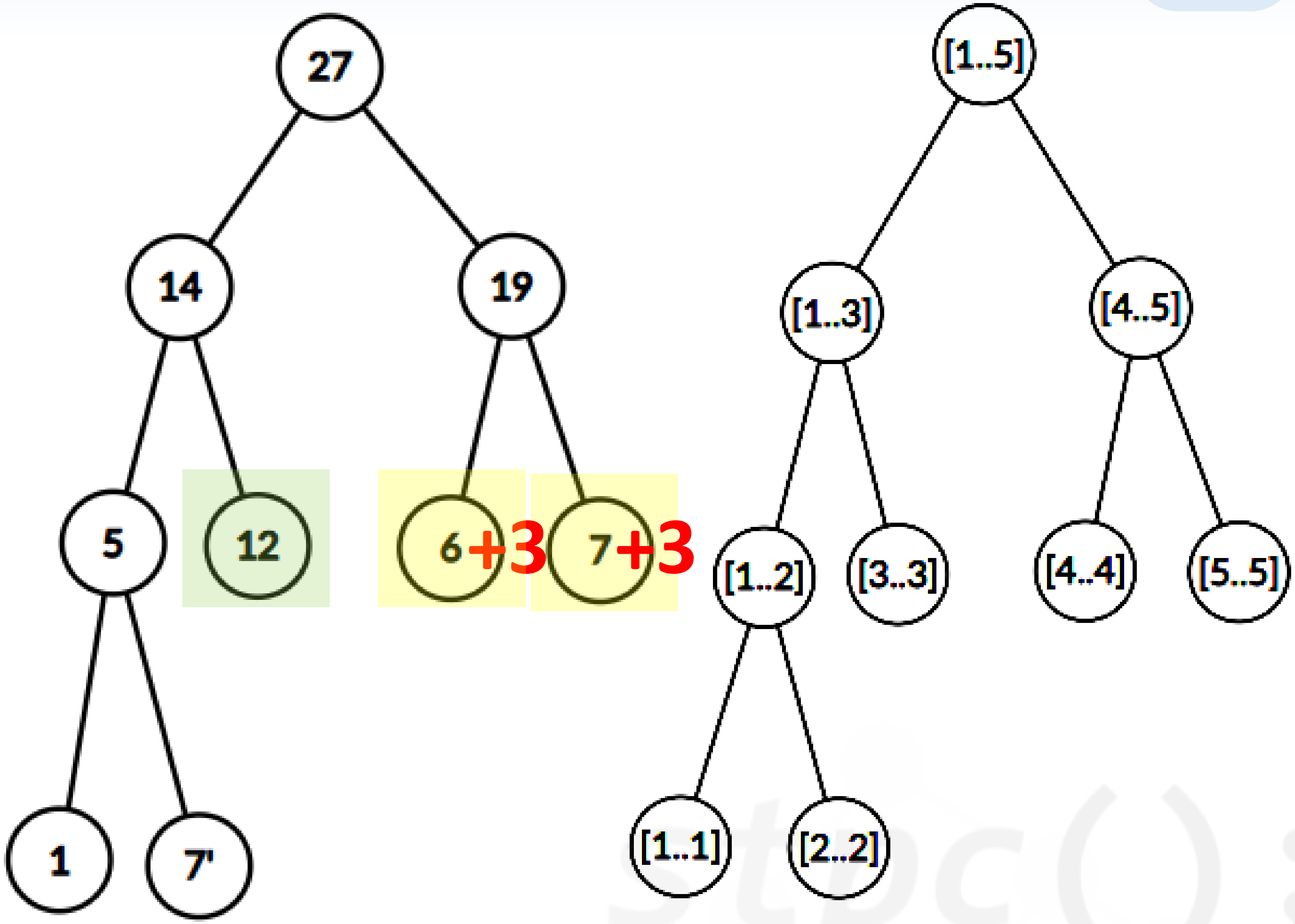




Example: $N = 5$

$a = \{1, 7, 12, 9, 10\}$

Find sum of $a[3..4]$.

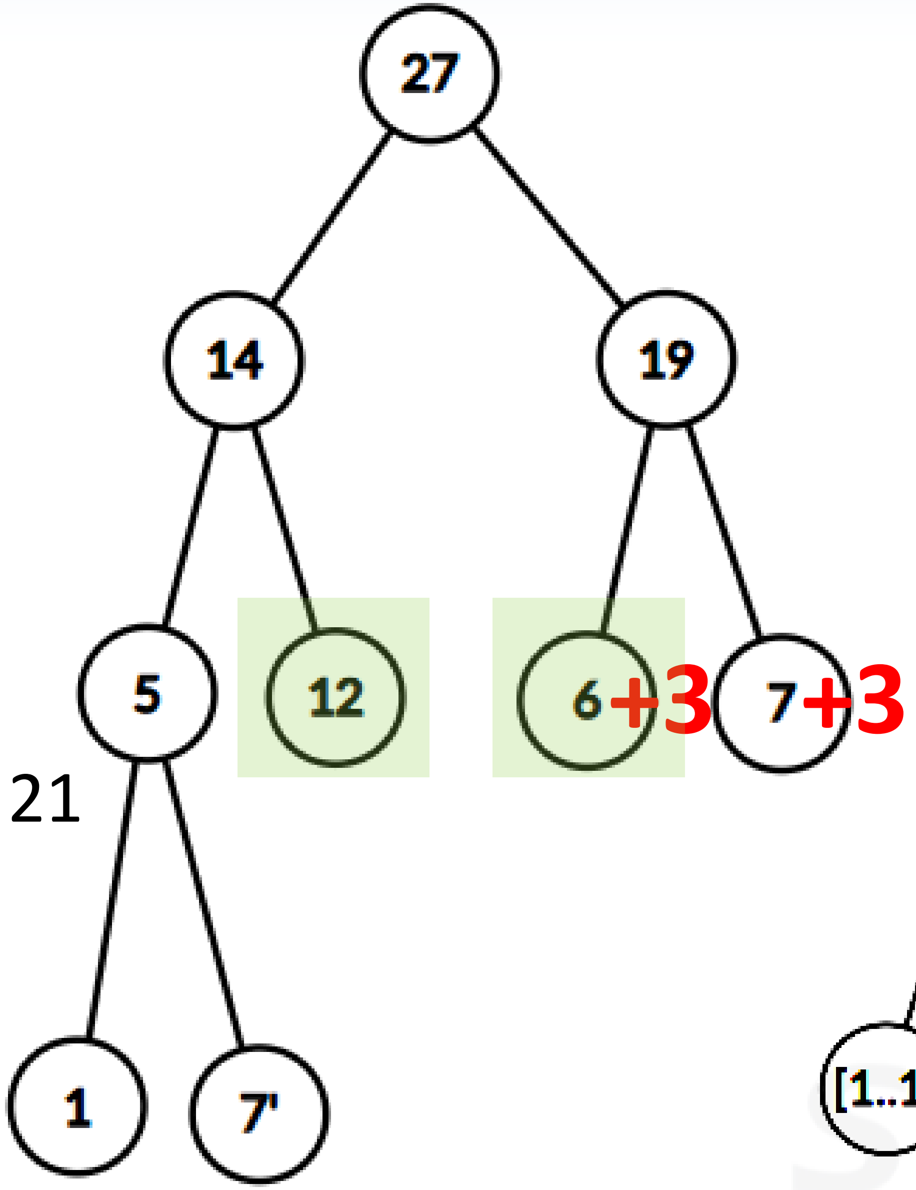


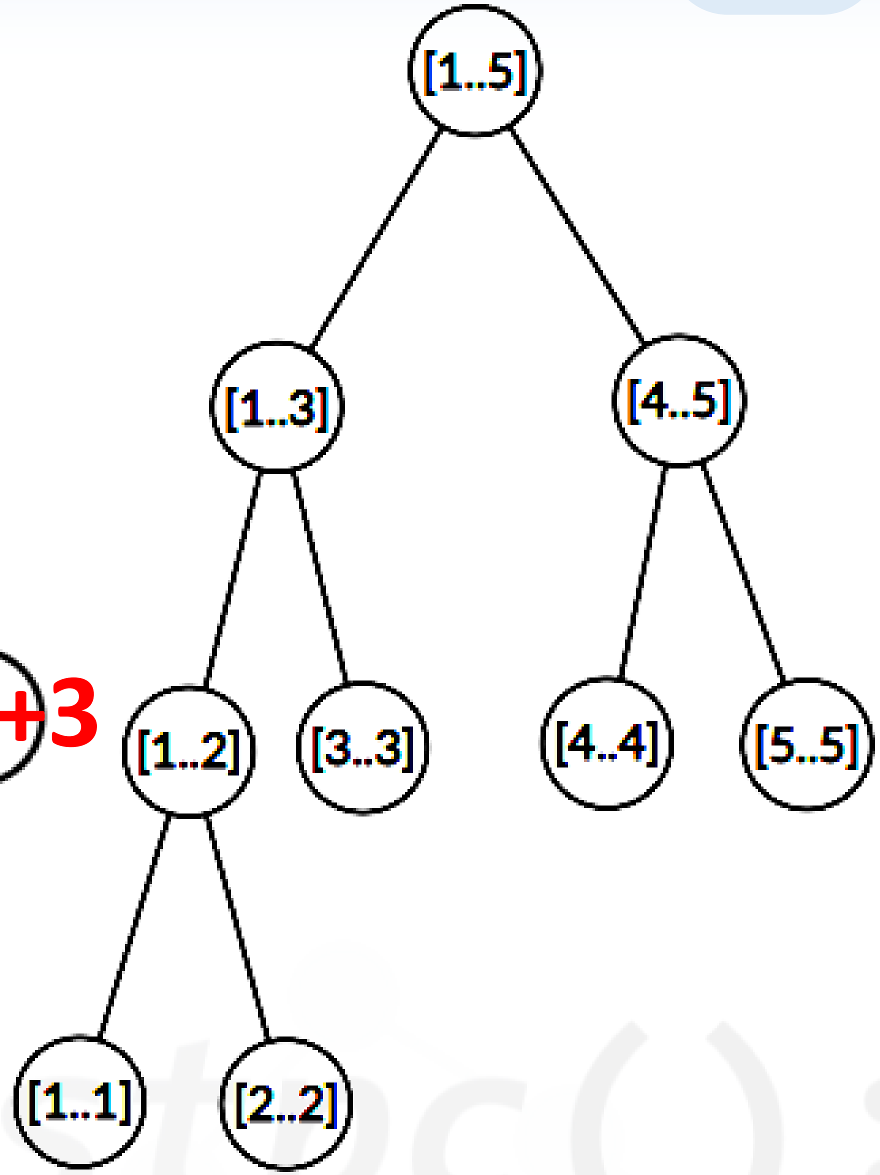
Example: $N = 5$

$a = \{1, 7, 12, 9, 10\}$

Find sum of $a[3..4]$.

Result = 12 + (6+3) = 21





Lazy propagation

As the pushing down of the lazy tag to the child node is O(1), therefore the new implementation of operation 1 will become O(log N).

The overall time complexity is O(Q log N) for Q queries.

The implementation is quite easy: maintain an additional array to store lazy tags.

Practice Problems

- Range Update, Range Query
- [TJOI 2009] 開關
- 等差數列
- 序列修改

Extensive Reading

- Memory efficient implementation of segment tree
- Discretization on segment tree
- Special operation on segment tree
- Merging and splitting segment trees
- Generalization of segment tree to higher dimensions
- Persistent segment tree