CSE 332 Winter 2026
Lecture 18: Parallel Prefix

Michael Whitmeyer

http://www.cs.uw.edu/332

Which Data Structures are “Suitable” for Parallelism?

- For each data structure, can we write a parallel algorithm to sum all of its values that’s *more efficient* than a sequential one?
  - Array
  - Linked List
  - Binary Tree

ForkJoin Framework

- This strategy is common enough that Java (and C++, and C#, and…) provides a library to do it for you!

What you would do in Threads	What to instead in ForkJoin
Subclass **Thread**	Subclass **RecursiveTask<V>**
Override **run**	Override **compute**
Store the answer in a field	Return a V from compute
Call **start**	Call **fork**
join synchronizes only	**join** synchronizes and returns the answer
Call **run** to execute sequentially	Call **compute** to execute sequentially
Have a topmost thread and call **run**	Create a pool and call **invoke**

Reduction/Fold

- **Input**: array
- **Output:** single object (sum, max, min, parity, histogram, etc.)
- We “reduce” all elements in an array to a single item
- **Requirement:** operation done among elements is associative
  - $(x + y) + z = x + (y + z)$
  - min(min(x,y),z) = min(x, min(y,z))
  - parity(parity(x,y), z) = parity(x, parity(y,z))
- The “single item” can itself be complex
  - E.g. create a histogram of results from an array of trials

ForkJoin Pseudocode for ~~Summing~~

Finding Max

GenericTask(arr)

1. **If** len(arr) < ℓ: return ~~sum~~ max of elements in arr
2. **Else:**
    1. Divide arr in half into arr1 and arr2
    2. Conquer in parallel: call **GenericTask**(arr1) in new thread and **GenericTask**(arr2) in this thread
3. Wait for the recursive calls/threads to finish
4. Combine ~~the sum of the two~~ recursive calls (and return)

Take the max of the two

fork()

compute()

join()

Map

- **Input:** array(s)
- **Output:** array (of same size)
- **Requirement:** apply some function to individual array elements.
- Examples:
  - Vector addition:
    - sum[i] = arr1[i] + arr2[i]
  - Function application:
    - out[i] = f(arr[i]);

ForkJoin Pseudocode for ~~Doubling each element~~

Applying map f

GenericTask(arr) Apply f to

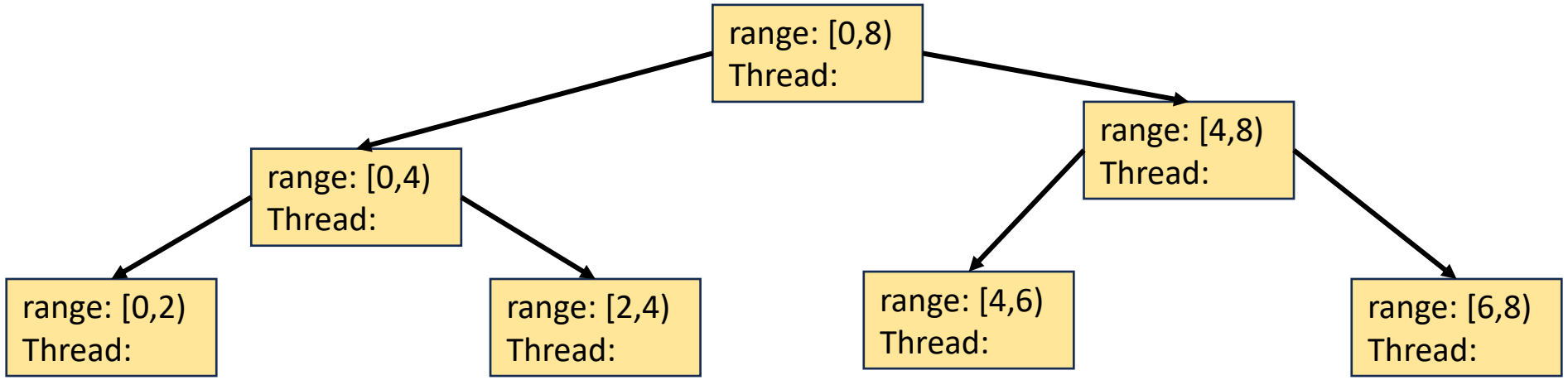
1. **If** len(arr) < ℓ: ~~double~~ each element in arr (sequentially)
2. **Else:**
    1. Divide arr in half into arr1 and arr2
    2. Conquer in parallel: call **GenericTask**(arr1) in new thread and **GenericTask**(arr2) in this thread
3. Wait for the recursive calls/threads to finish
4. Return

fork()

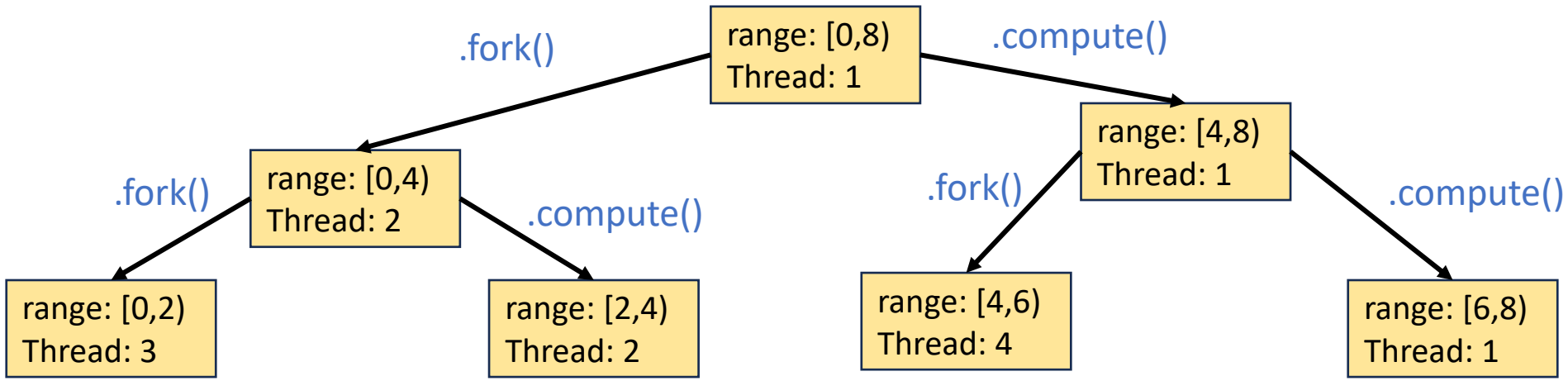
compute()

join()

ForkJoin Picture



ForkJoin Picture



Map/Reduction Example

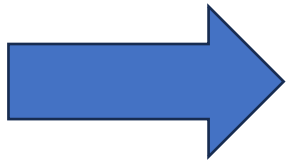
- **Task:** Multiply together the lengths of all the odd-length strings in array
  1. Apply map to convert the array of strings into an array of their lengths
  2. Then do a map on that array so each value maps to 1 if it's even and itself if it's odd
  3. Then do a reduction to multiply together that final result

- Note: You could do this in a single ForkJoin RecursiveTask
  - but "deconstructing" useful since some languages designed specifically for parallelism have Map/Reduce built in.
  - Map and Reduce are two from **a trio, with Pack/Filter** being the third

Pack/Filter

- Given an array of values and a Boolean function, return a new array which contains only elements that were “true”

10	16	4	18	8	2



10	16	18

$f(x) = x > 9$

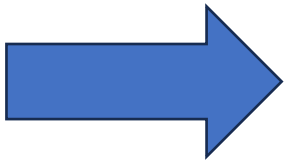
Question: Can we Pack/Filter in parallel?

Answer: yes, with help of parallel **Prefix Sum**

Prefix Sum

- Given an array, compute a new array where each index i is the sum of all values up to i

10	16	4	18	8	2



10	26	30	48	56	58

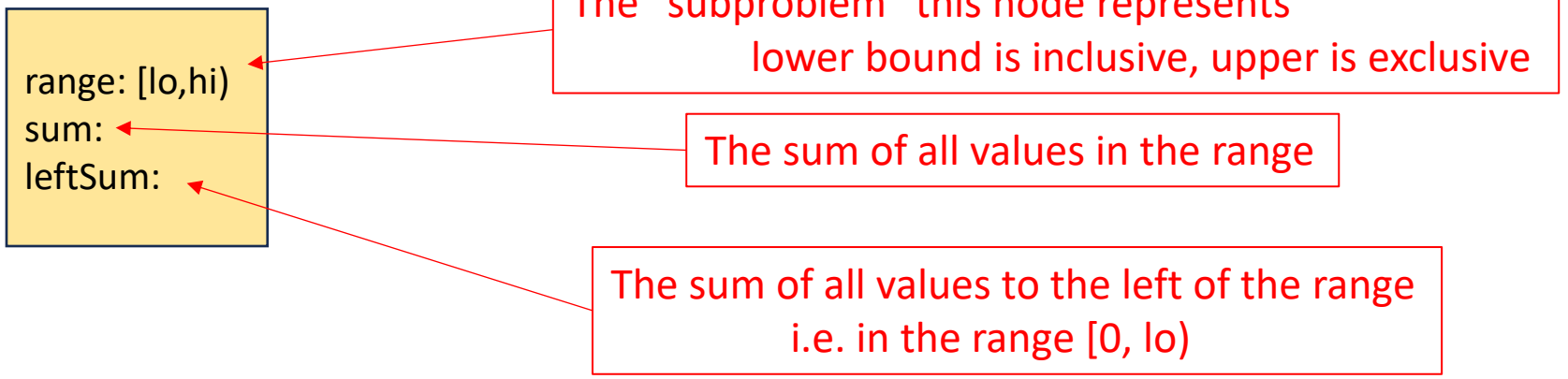
Later: will be useful for parallel packing/filtering!

Parallel Prefix Sum

Richard Ladner
Allen School Faculty

- Algorithm will have two major parallel steps
  - Called a “two pass” parallel algorithm
- First step:
  - Create a tree data structure
- Second Step:
  - Use the tree to fill in the output array

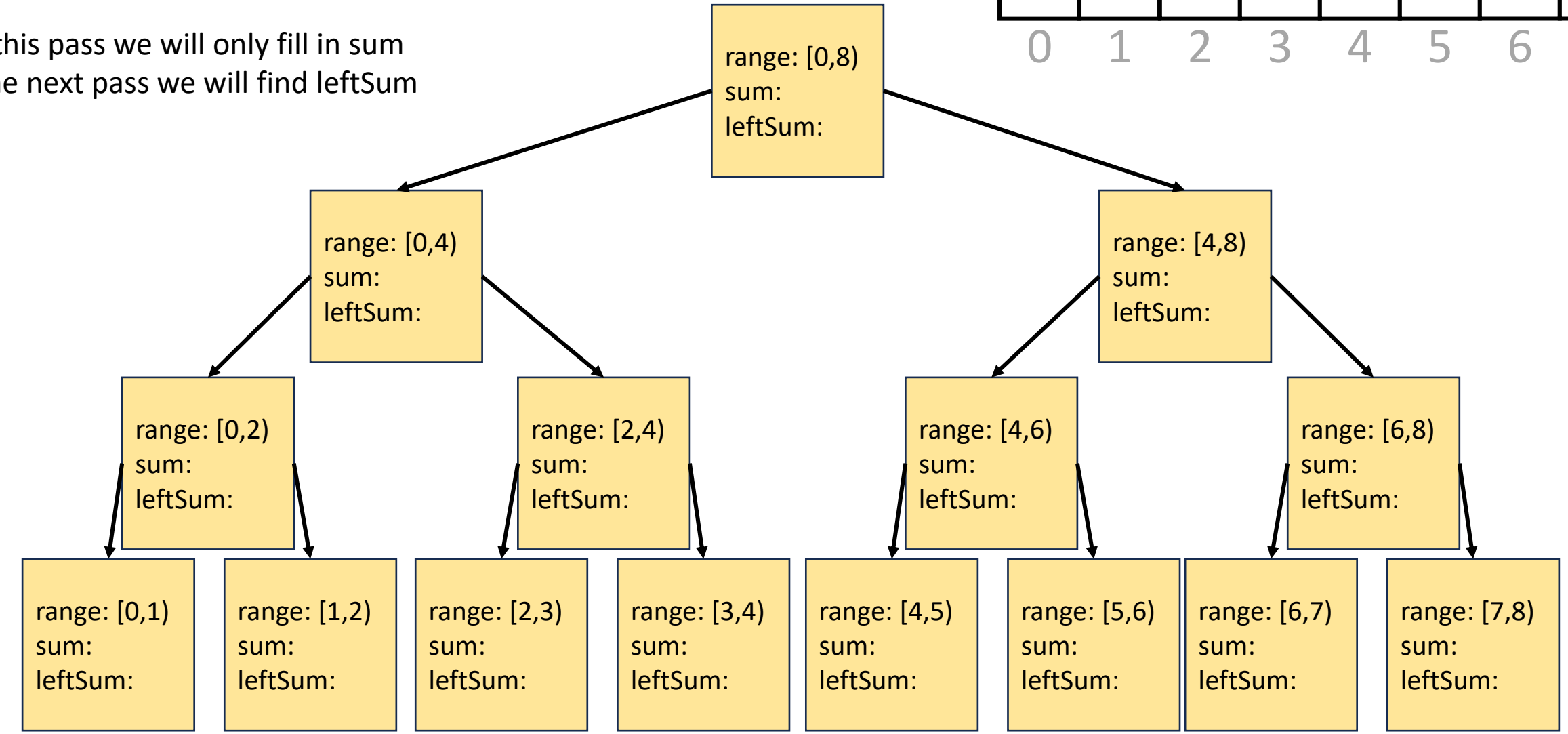
Tree Node:



Step 1: Using D&C
Create a Tree, Fill in sum

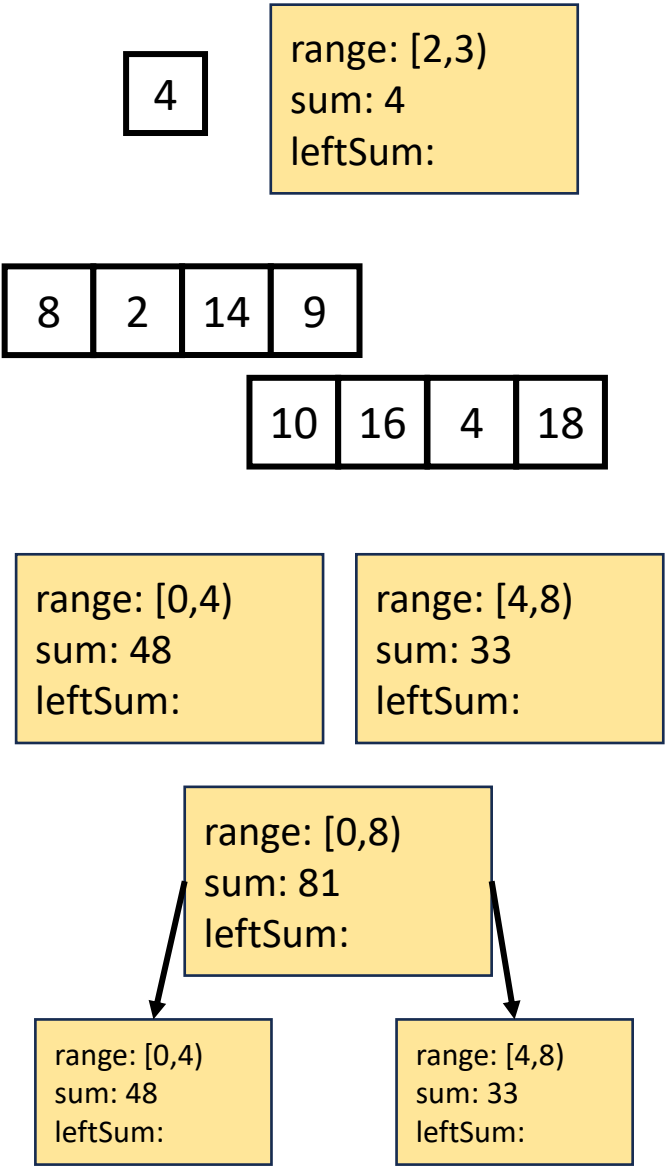
Input:	10	16	4	18	8	2	14	9
Output:								
	0	1	2	3	4	5	6	7

For this pass we will only fill in sum
In the next pass we will find leftSum



Step 1: Create a Tree, Fill in sum

10	16	4	18	8	2	14	9



- **Base Case:**
  - If the rand is smaller than the Sequential Cutoff, create a node for that range and find the sum sequentially
- **Divide:**
  - Split the list into two "sublists" of (roughly) equal length, create a thread for each sublist.
- **Conquer:**
  - Call **start()** for each thread to compute the left and right subtrees
- **Combine:**
  - Create parent node, connect to children, fill in sum

Step 1 pseudocode (create tree, compute sum)

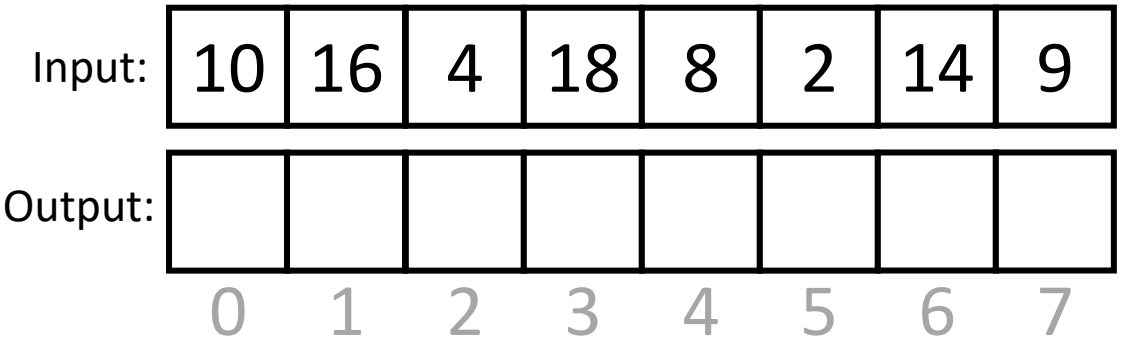
BuildTree(arr)

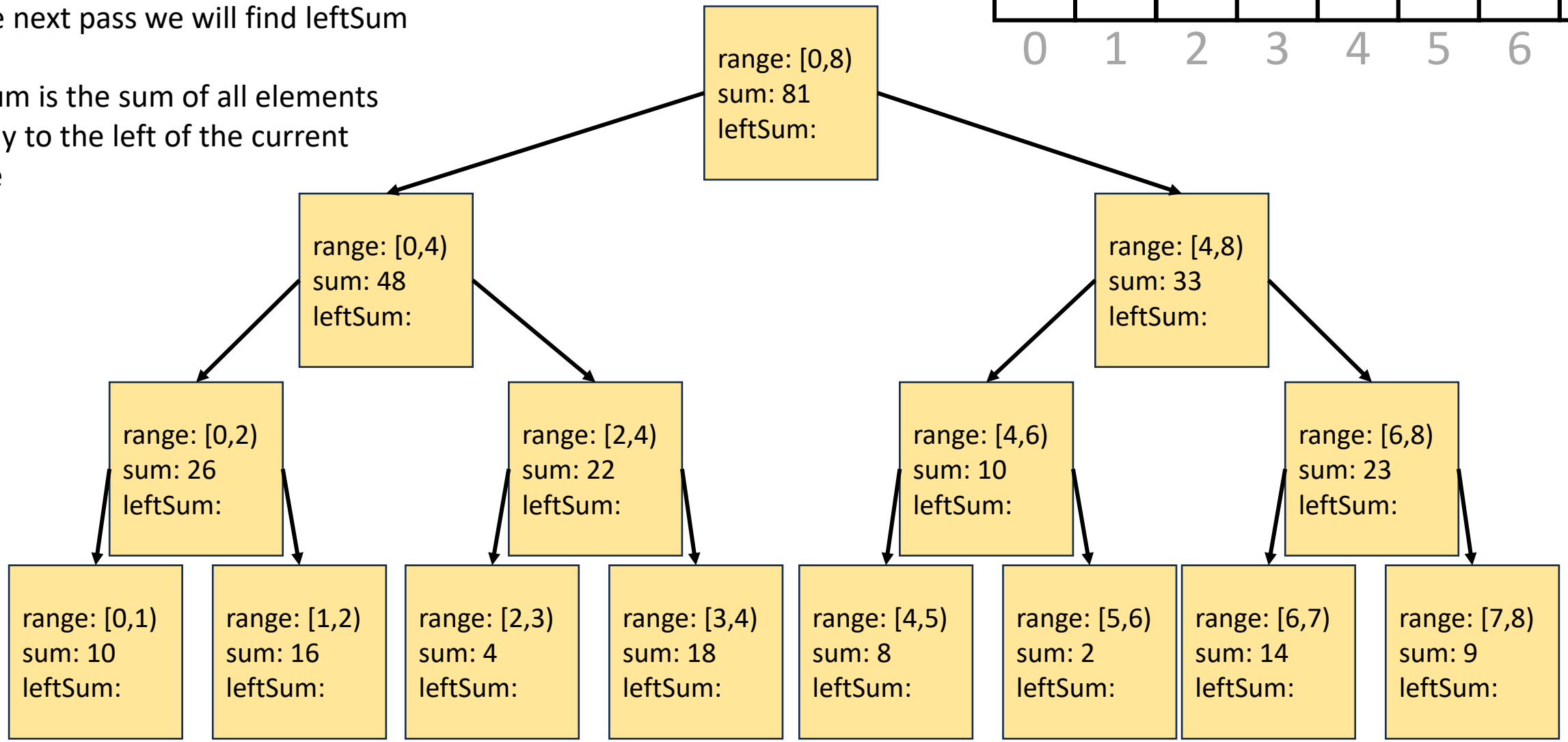
1. **If** len(arr) < ℓ:
    1. Create new TreeNode Leaf
    2. Set Leaf.sum = sum of values in arr and return Leaf
2. **Else:**
    1. Divide arr in half into arr1 and arr2
    2. Conquer: TreeNode leftChild = **BuildTree**(arr1) in new thread, TreeNode rightChild = **BuildTree**(arr2) in this thread
    3. Wait for parallel computations to finish
    4. Combine: Create TreeNode parent, set parent.sum = leftChild.sum + rightChild.sum, parent.left = leftChild, parent.right = rightChild
    5. Return parent

After Step 1

All sums filled in per node
In the next pass we will find leftSum

leftSum is the sum of all elements strictly to the left of the current range

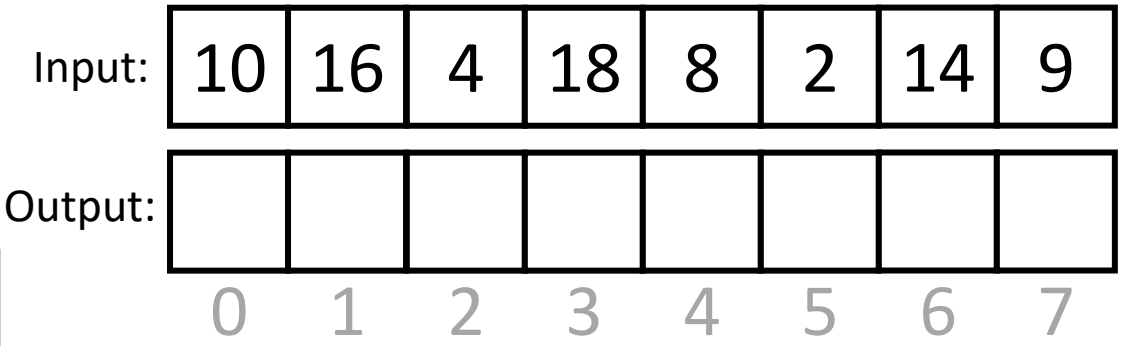


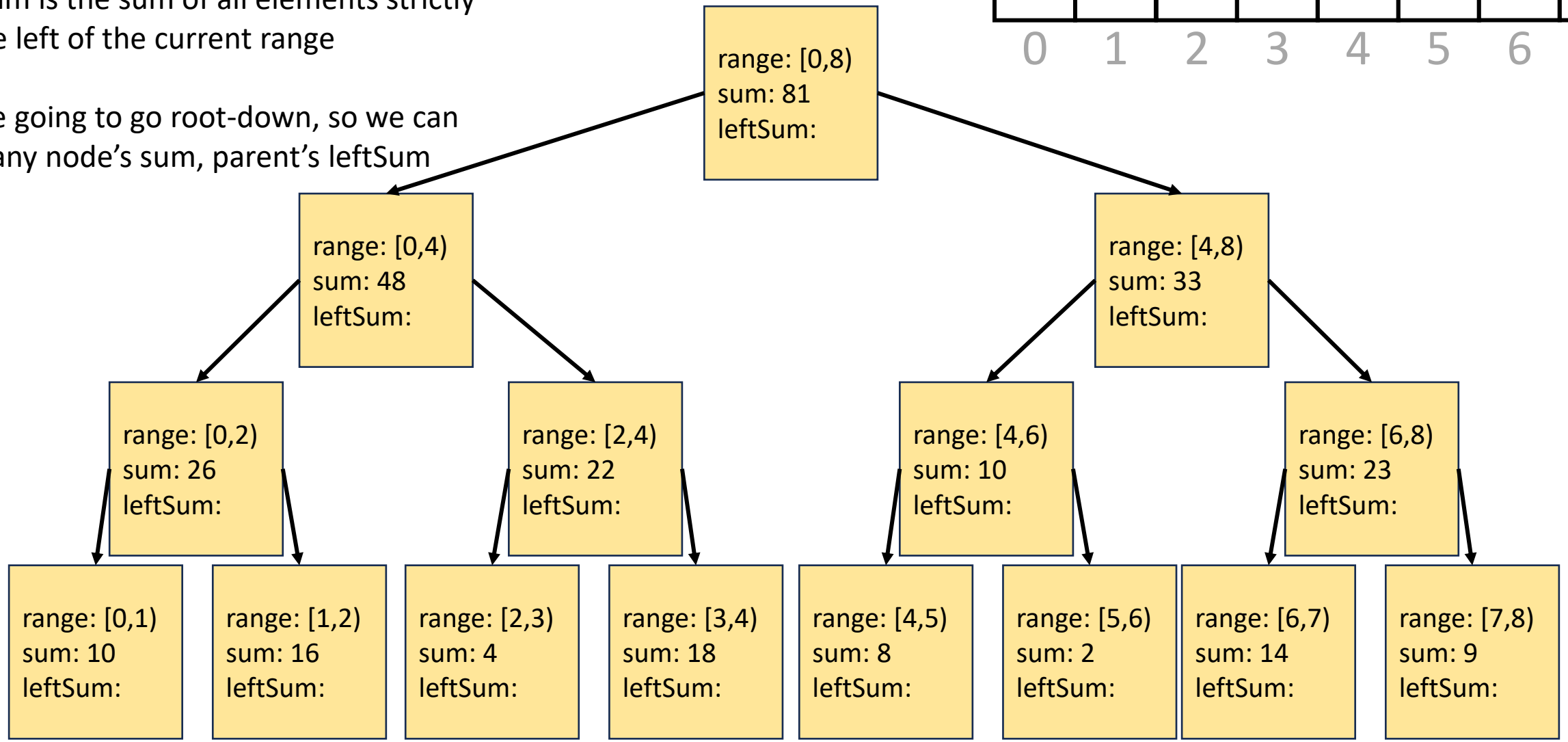


Step 2: fill in leftSum and Output

leftSum is the sum of all elements strictly to the left of the current range

We're going to go root-down, so we can use: any node's sum, parent's leftSum





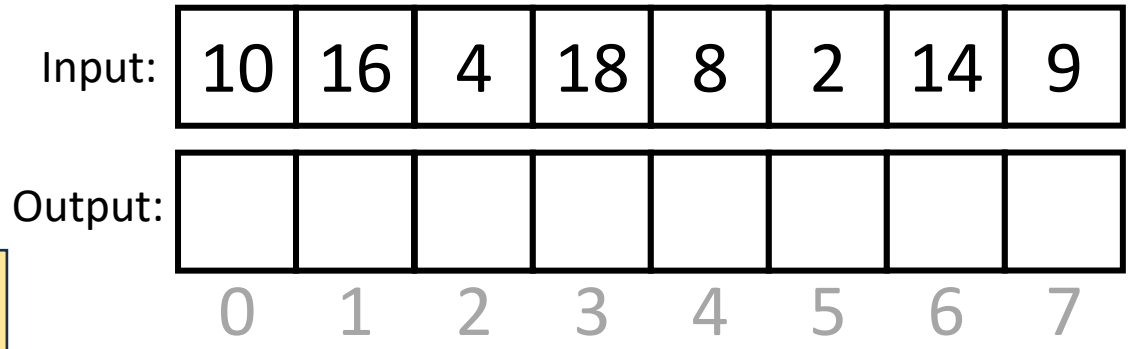
Step 2: fill in leftSum and Output

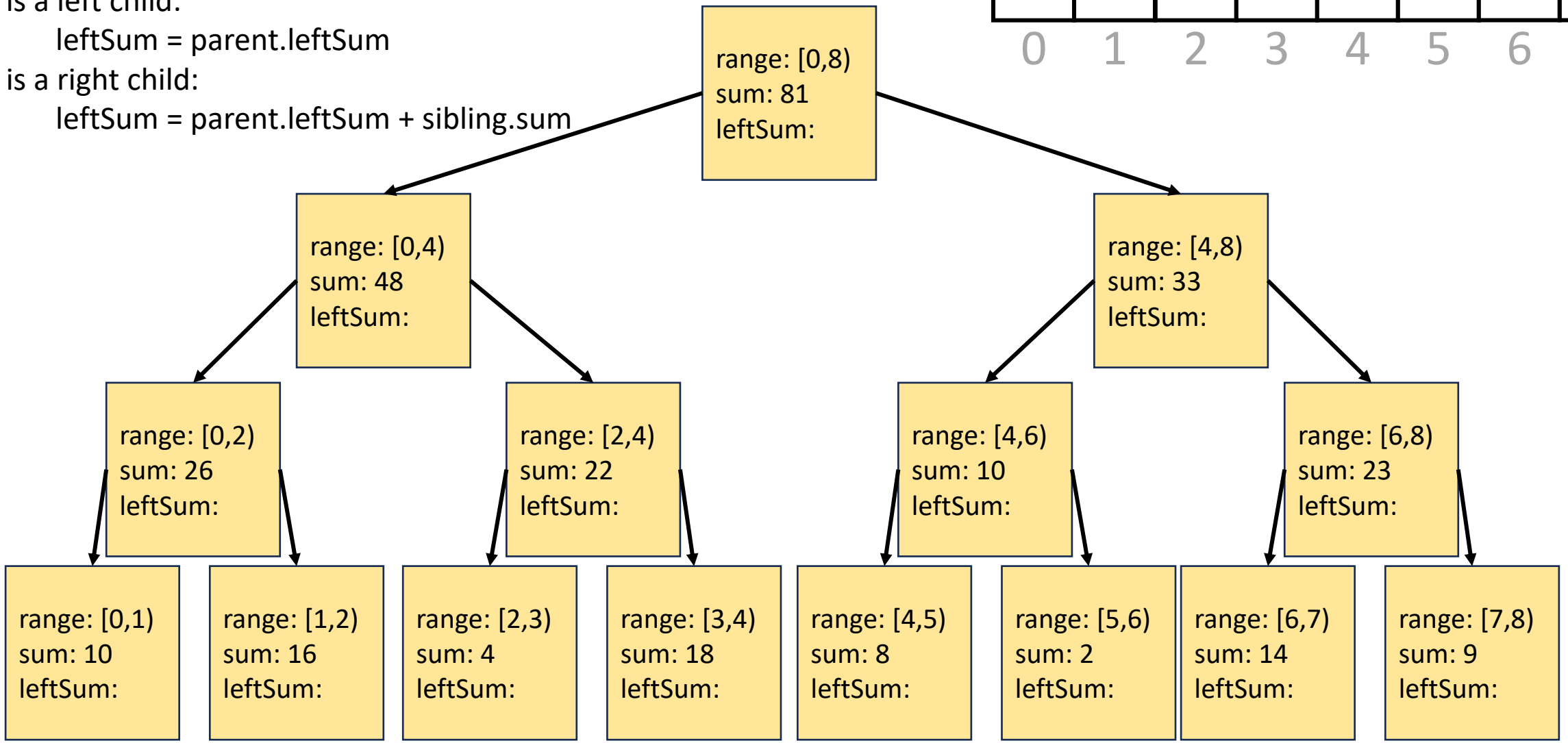
If this is a left child:

leftSum = parent.leftSum

If this is a right child:

leftSum = parent.leftSum + sibling.sum

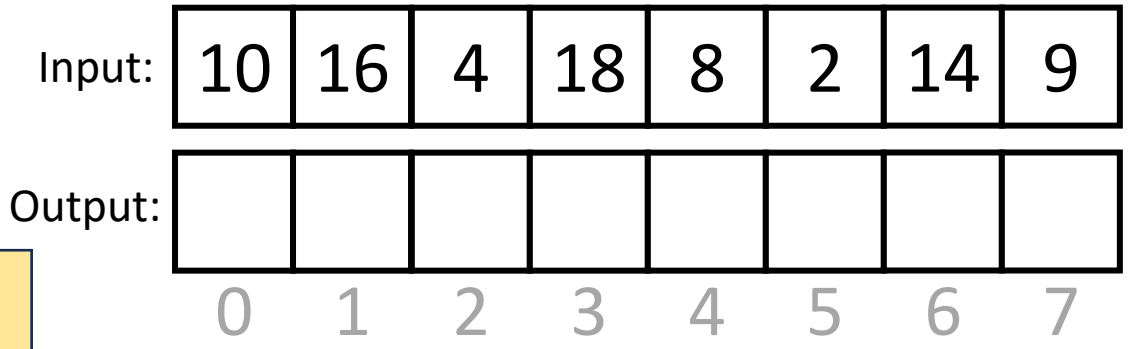


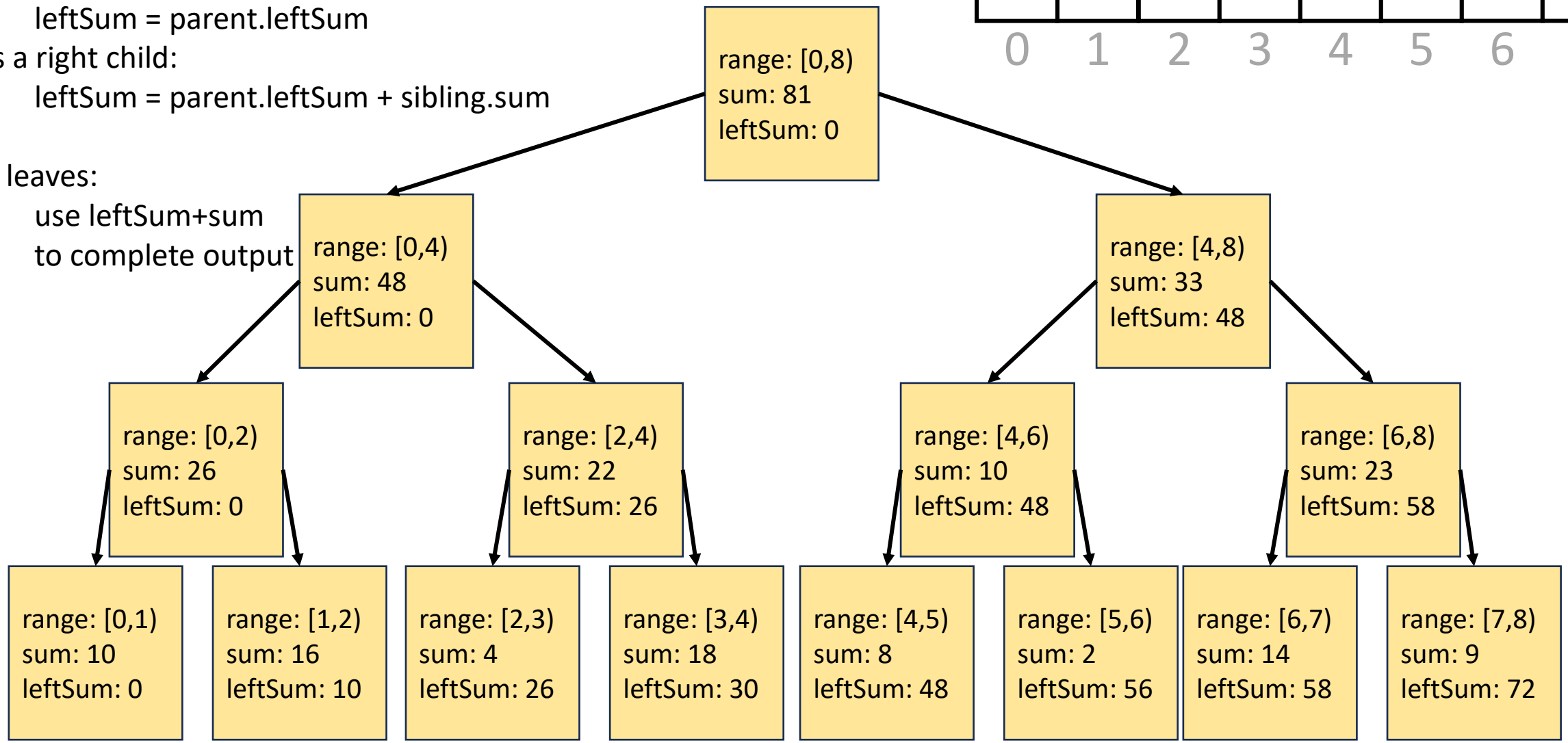


Step 2: fill in leftSum and Output

If this is a left child:
leftSum = parent.leftSum
If this is a right child:
leftSum = parent.leftSum + sibling.sum

For the leaves:
use leftSum+sum
to complete output



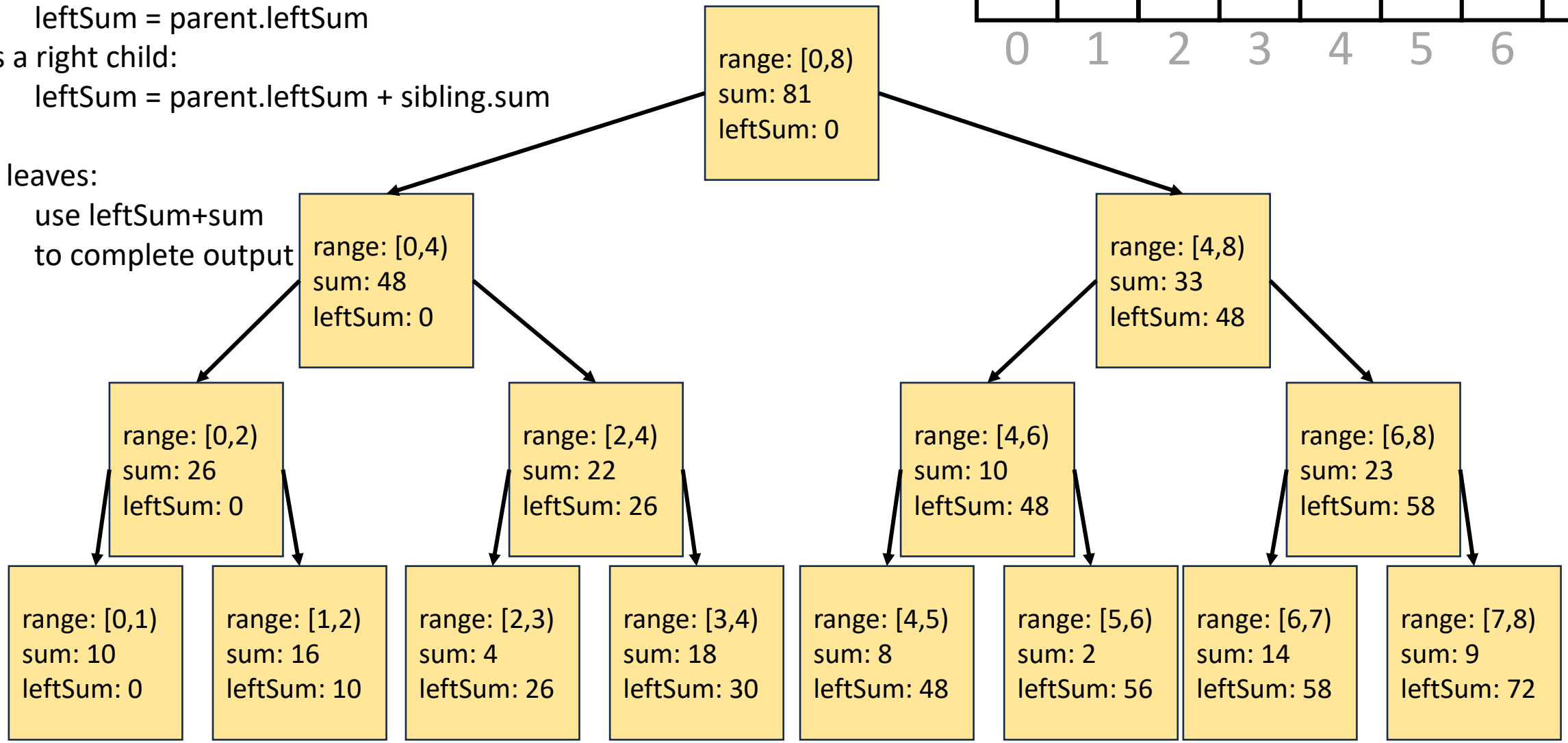


Step 2: fill in leftSum and Output

If this is a left child:
leftSum = parent.leftSum
If this is a right child:
leftSum = parent.leftSum + sibling.sum

For the leaves:
use leftSum+sum
to complete output

Input:	10	16	4	18	8	2	14	9
Output:	10	26	30	48	56	58	72	81
	0	1	2	3	4	5	6	7



Step 2 pseudocode (Compute Leftsum)

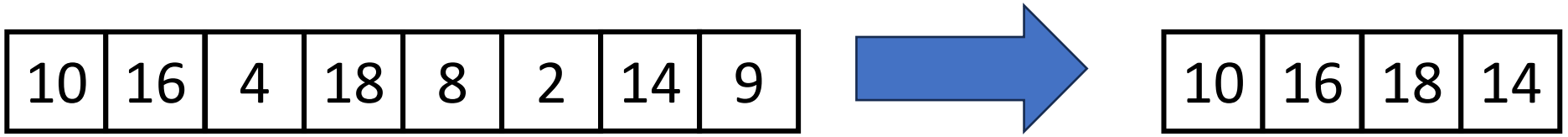
CompleteTree(Node curr)

1. **If** curr is a left child of curr.parent:
    1. Set curr.LeftSum = curr.parent.LeftSum
2. **Else:**
    1. Set curr.LeftSum = curr.parent.LeftSum + curr.sibling.Sum
3. **If** curr is not a leaf:
    1. Divide and conquer: call **CompleteTree**(curr.left) and **CompleteTree**(curr.right) in parallel threads
    2. Wait for parallel computations to finish
4. **Else (curr is a leaf)**:
    1. Set output[curr.lo] = curr.LeftSum + input[curr.lo]
    2. for i = curr.lo + 1 to curr.hi:
        1. Set output[i] = output[i-1] + input[i]

Base Case

Whew! Back to Pack/Filter

- Given an array of values and a Boolean function, return a new array which contains only elements that were “true”



$f(x) = x > 9$

Parallel Pack

Input:

10	16	4	18	8	2	14	9

, $f(x) = x > 9$

Output:

10	16	18	14

0 1 2 3 4 5 6 7

1. Do a **map** to identify the true elements

1	1	0	1	0	0	1	0

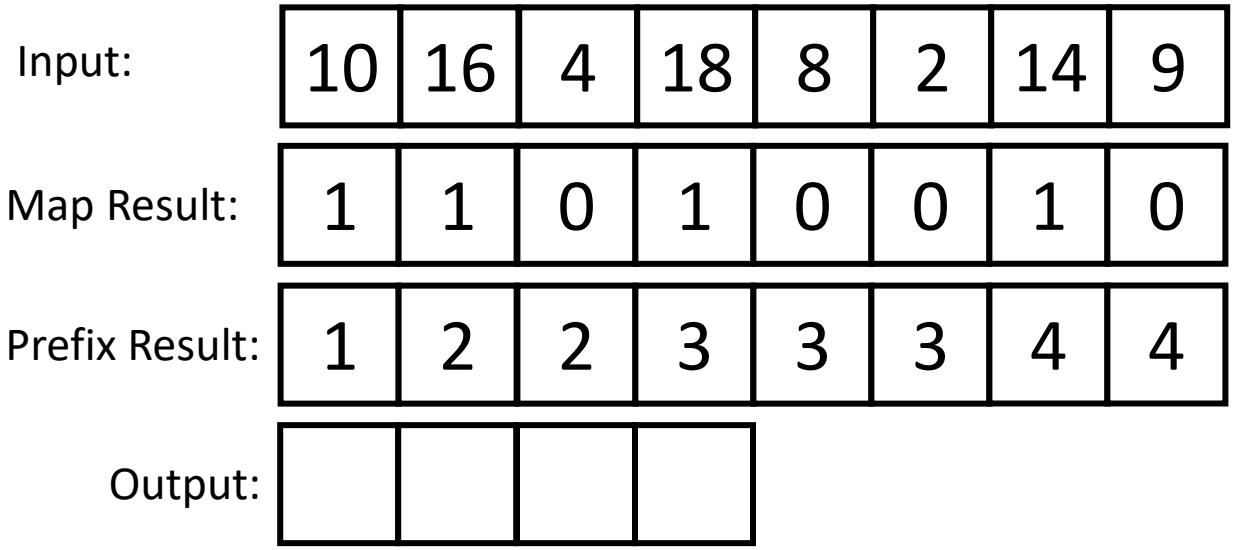
2. Do **prefix sum** on the result → count of true elements seen to the left of each position

1	2	2	3	3	3	4	4

3. Fill in the output in parallel:

10	16	18	14

3. Fill in the output in parallel:



- Because the last value in the prefix result is 4, the length of the output is 4
- Each time there is a 1 in the map result, we want to include that element in the output
- If element i should be included, its position matches prefixResult[i]-1

In parallel: if mapResult[i] == 1:
output[prefixResult[i]-1] = input[i]

Map/Reduction/Pack Example

- Multiply together the lengths of all of the odd-length strings in a given array
  - First, do a map to covert the array of strings into an array of their lengths
  - Then do a map on that array so each value maps to 1 if it's even and itself if it's odd
    - Alternatively, do a pack on the array to remove all even values
  - Then do a reduction to multiply together that final result