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)

max

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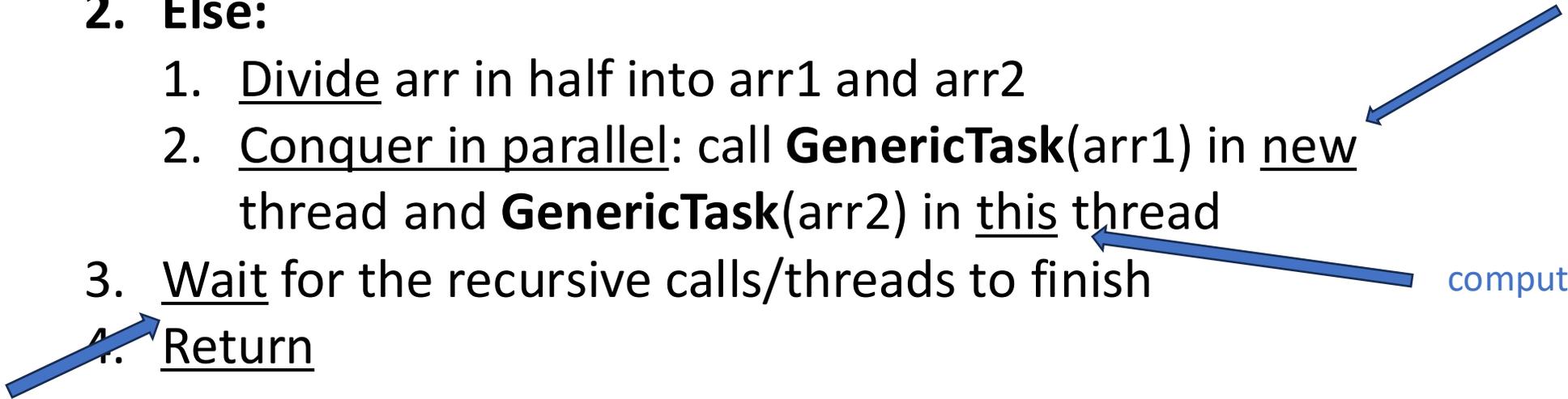
GenericTask(arr) Apply f to
1. **If** len(arr) < ℓ: ~~double~~ each element in arr (sequentially)
2. **Else:**
   1. <u>Divide</u> arr in half into arr1 and arr2
   2. <u>Conquer in parallel</u>: call **GenericTask**(arr1) in <u>new</u> thread and **GenericTask**(arr2) in <u>this</u> thread
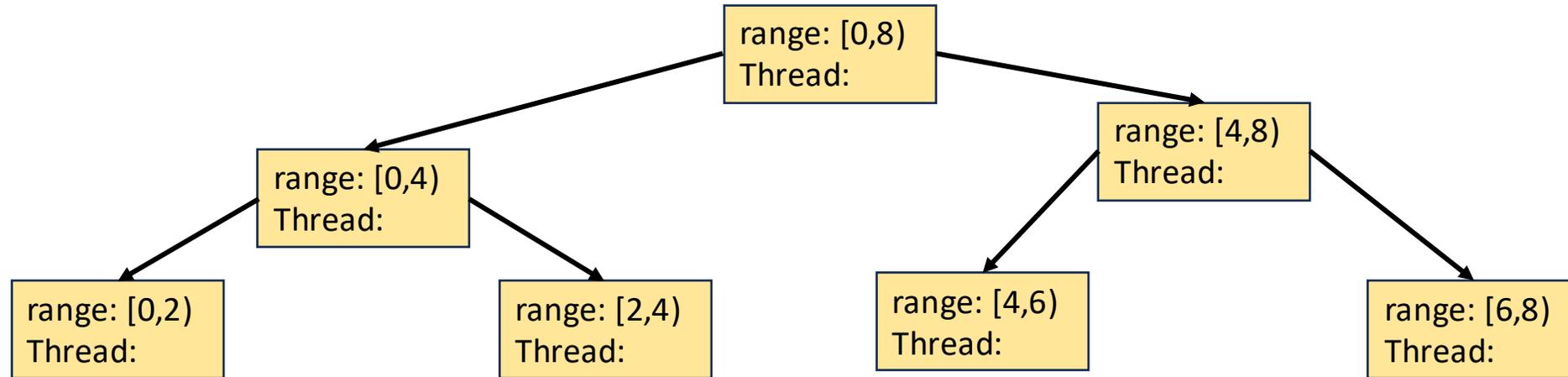3. <u>Wait</u> for the recursive calls/threads to finish
4. <u>Return</u>

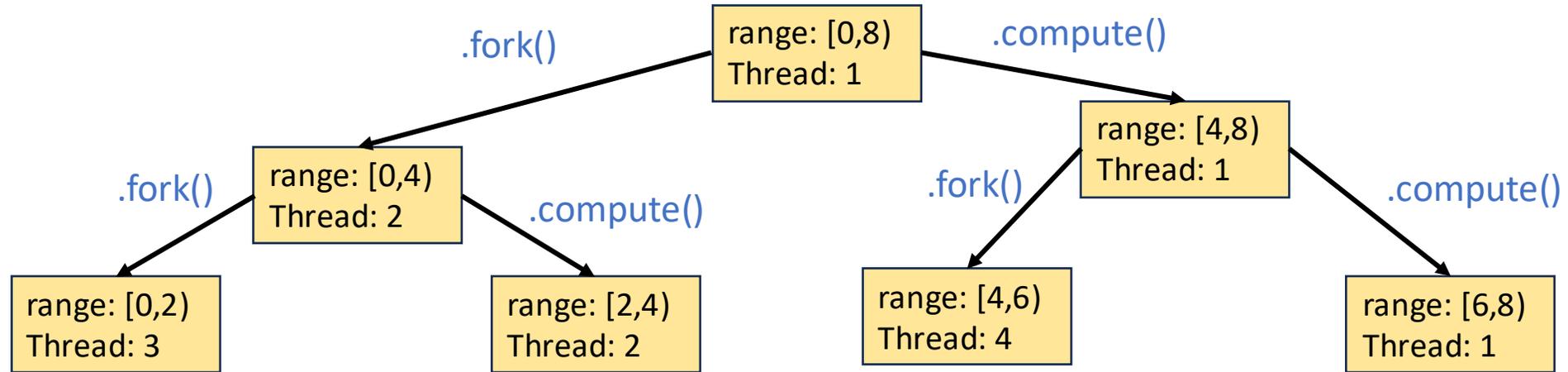fork()

compute()

join()

ForkJoin Picture

ForkJoin Picture

Map/Reduction Example

- **Task:** Multiply together the lengths of all the odd-length strings in array
  1. Apply <u>map</u> 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

→

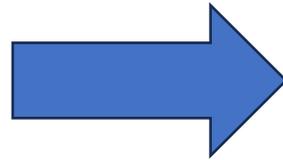| 10	16	18
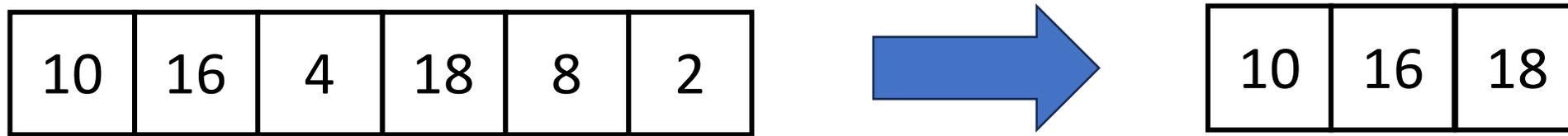
$$f(x) = x > 9$$

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

- 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

Richard Ladner
Allen School Faculty

Tree Node:

range: [lo,hi)
sum:
leftSum:
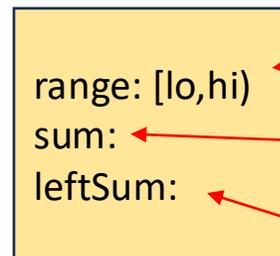
The "subproblem" this node represents
lower bound is inclusive, upper is exclusive

The sum of all values in the range

The sum of all values to the left of the range
i.e. in the range [0, lo)

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

range: [0,8)
sum:
leftSum:

range: [0,4)
sum:
leftSum:

range: [4,8)
sum:
leftSum:

range: [0,2)
sum:
leftSum:

range: [2,4)
sum:
leftSum:

range: [4,6)
sum:
leftSum:

range: [6,8)
sum:
leftSum:

range: [0,1)
sum:
leftSum:

range: [1,2)
sum:
leftSum:

range: [2,3)
sum:
leftSum:

range: [3,4)
sum:
leftSum:

range: [4,5)
sum:
leftSum:

range: [5,6)
sum:
leftSum:

range: [6,7)
sum:
leftSum:

range: [7,8)
sum:
leftSum:
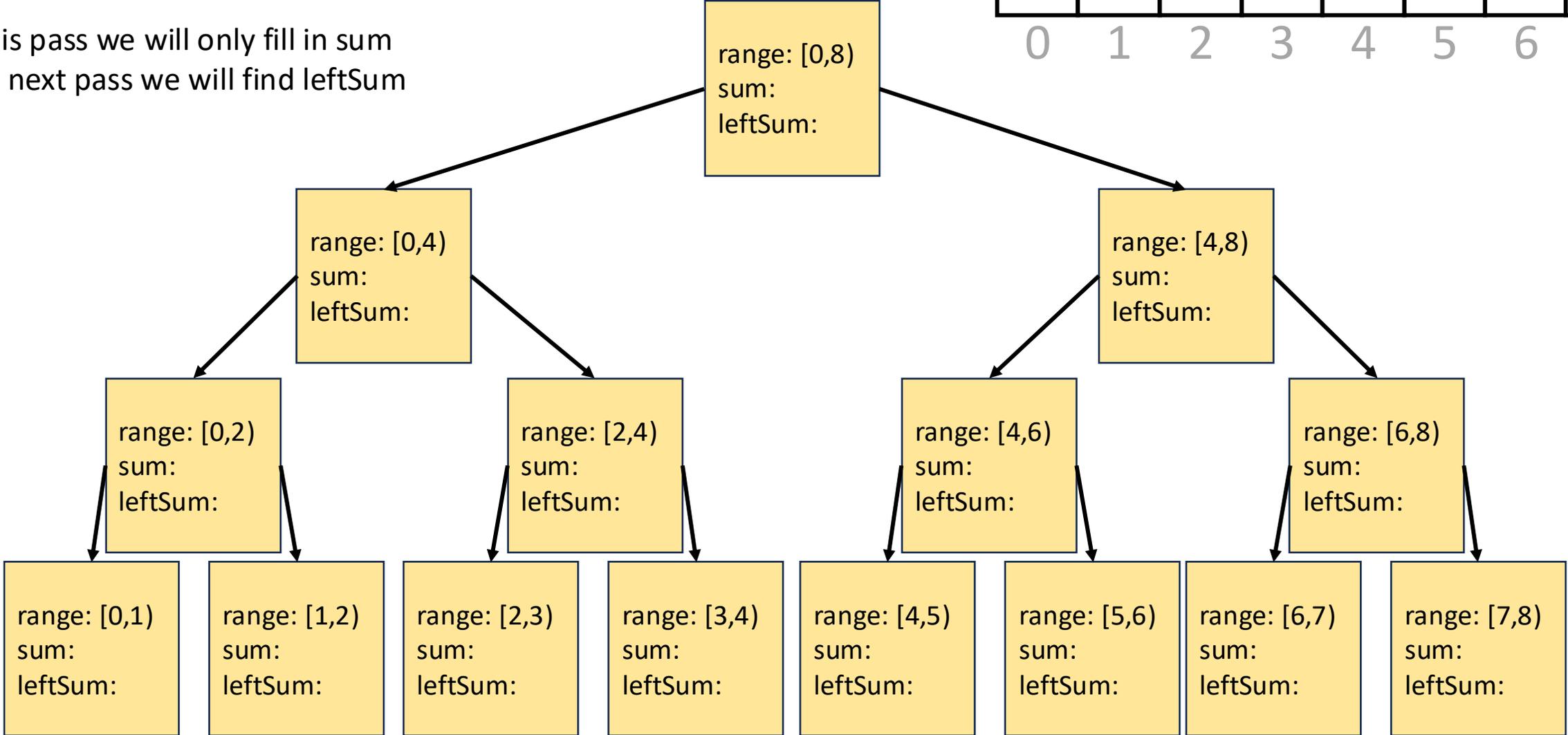
Step 1: Create a Tree, Fill in sum

| 10 | 16 | 4 | 18 | 8 | 2 | 14 | 9 |

| 4 |

range: [2,3)
sum: 4
leftSum:

| 8 | 2 | 14 | 9 |

| 10 | 16 | 4 | 18 |

range: [0,4)
sum: 48
leftSum:

range: [4,8)
sum: 33
leftSum:

range: [0,8)
sum: 81
leftSum:

range: [0,4)
sum: 48
leftSum:

range: [4,8)
sum: 33
leftSum:

- **Base Case**:
  - If the range 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**:
  - Compute the left and right subtrees in parallel

- **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. <u>Divide</u> arr in half into arr1 and arr2
   2. <u>Conquer:</u> TreeNode leftChild = **BuildTree**(arr1) in <u>new</u> thread, TreeNode rightChild = **BuildTree**(arr2) in <u>this</u> thread
   3. <u>Wait</u> for parallel computations to finish
   4. <u>Combine:</u> Create TreeNode parent, set parent.sum = leftChild.sum + rightChild.sum, parent.left = leftChild, parent.right = rightChild
   5. <u>Return</u> parent

After Step 1

Input:

10	16	4	18	8	2	14	9

Output:

0	1	2	3	4	5	6	7

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


```
range: [0,8)
sum: 81
leftSum:
```

```
range: [0,4)
sum: 48
leftSum:
```

```
range: [4,8)
sum: 33
leftSum:
```

```
range: [0,2)
sum: 26
leftSum:
```

```
range: [2,4)
sum: 22
leftSum:
```

```
range: [4,6)
sum: 10
leftSum:
```

```
range: [6,8)
sum: 23
leftSum:
```

```
range: [0,1)
sum: 10
leftSum:
```

```
range: [1,2)
sum: 16
leftSum:
```

```
range: [2,3)
sum: 4
leftSum:
```

```
range: [3,4)
sum: 18
leftSum:
```

```
range: [4,5)
sum: 8
leftSum:
```

```
range: [5,6)
sum: 2
leftSum:
```

```
range: [6,7)
sum: 14
leftSum:
```

```
range: [7,8)
sum: 9
leftSum:
```
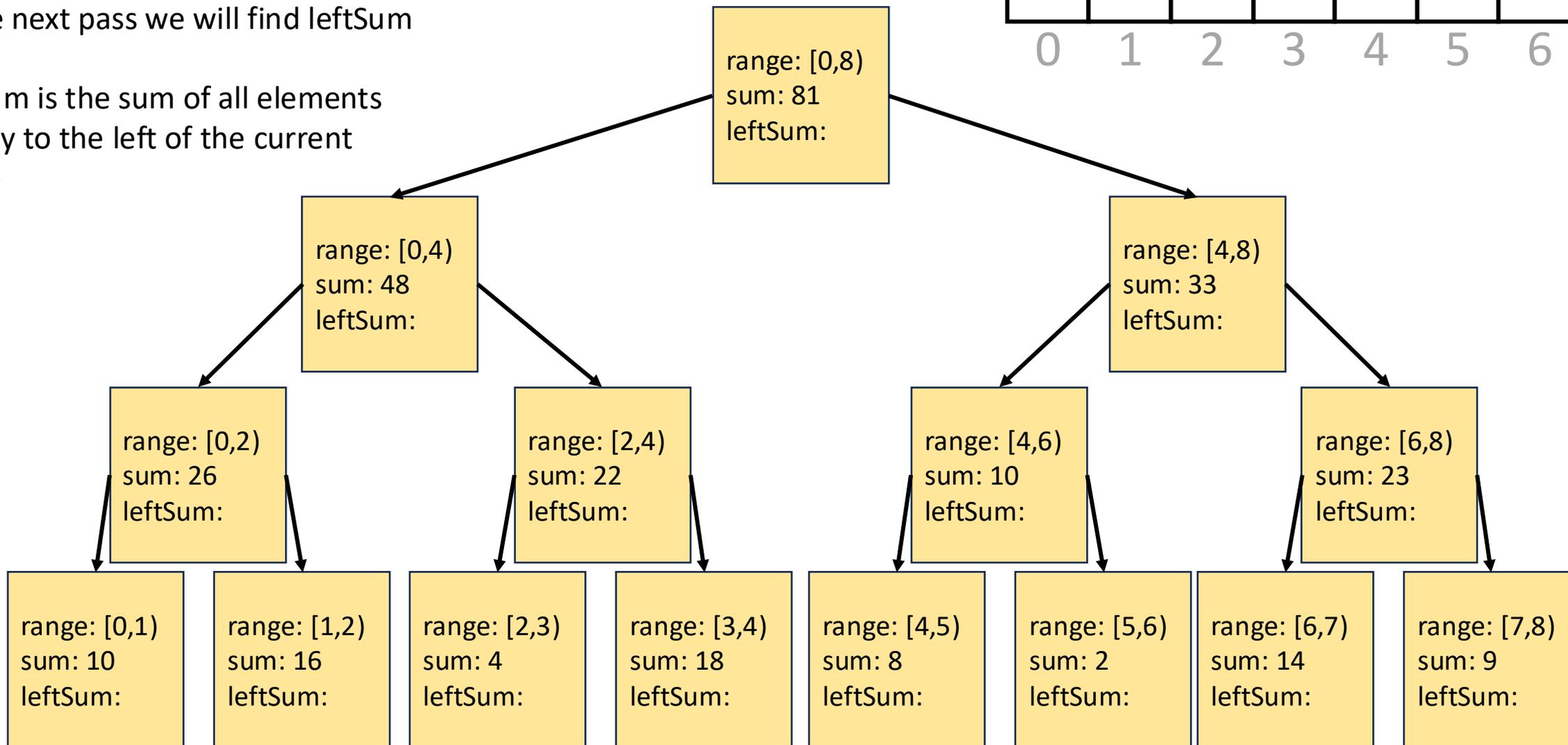
Step 2: fill in leftSum and Output

Input:
10	16	4	18	8	2	14	9

Output:

0   1   2   3   4   5   6   7

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

range: [0,8)
sum: 81
leftSum:

range: [0,4)
sum: 48
leftSum:

range: [4,8)
sum: 33
leftSum:

range: [0,2)
sum: 26
leftSum:

range: [2,4)
sum: 22
leftSum:

range: [4,6)
sum: 10
leftSum:

range: [6,8)
sum: 23
leftSum:

range: [0,1)
sum: 10
leftSum:

range: [1,2)
sum: 16
leftSum:

range: [2,3)
sum: 4
leftSum:

range: [3,4)
sum: 18
leftSum:

range: [4,5)
sum: 8
leftSum:

range: [5,6)
sum: 2
leftSum:

range: [6,7)
sum: 14
leftSum:

range: [7,8)
sum: 9
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

Input:

10	16	4	18	8	2	14	9

Output:

0	1	2	3	4	5	6	7

range: [0,8)
sum: 81
leftSum:

range: [0,4)
sum: 48
leftSum:

range: [4,8)
sum: 33
leftSum:

range: [0,2)
sum: 26
leftSum:

range: [2,4)
sum: 22
leftSum:

range: [4,6)
sum: 10
leftSum:

range: [6,8)
sum: 23
leftSum:

range: [0,1)
sum: 10
leftSum:

range: [1,2)
sum: 16
leftSum:

range: [2,3)
sum: 4
leftSum:

range: [3,4)
sum: 18
leftSum:

range: [4,5)
sum: 8
leftSum:

range: [5,6)
sum: 2
leftSum:

range: [6,7)
sum: 14
leftSum:

range: [7,8)
sum: 9
leftSum:

Step 2: fill in leftSum and Output

Input:

10	16	4	18	8	2	14	9

Output:

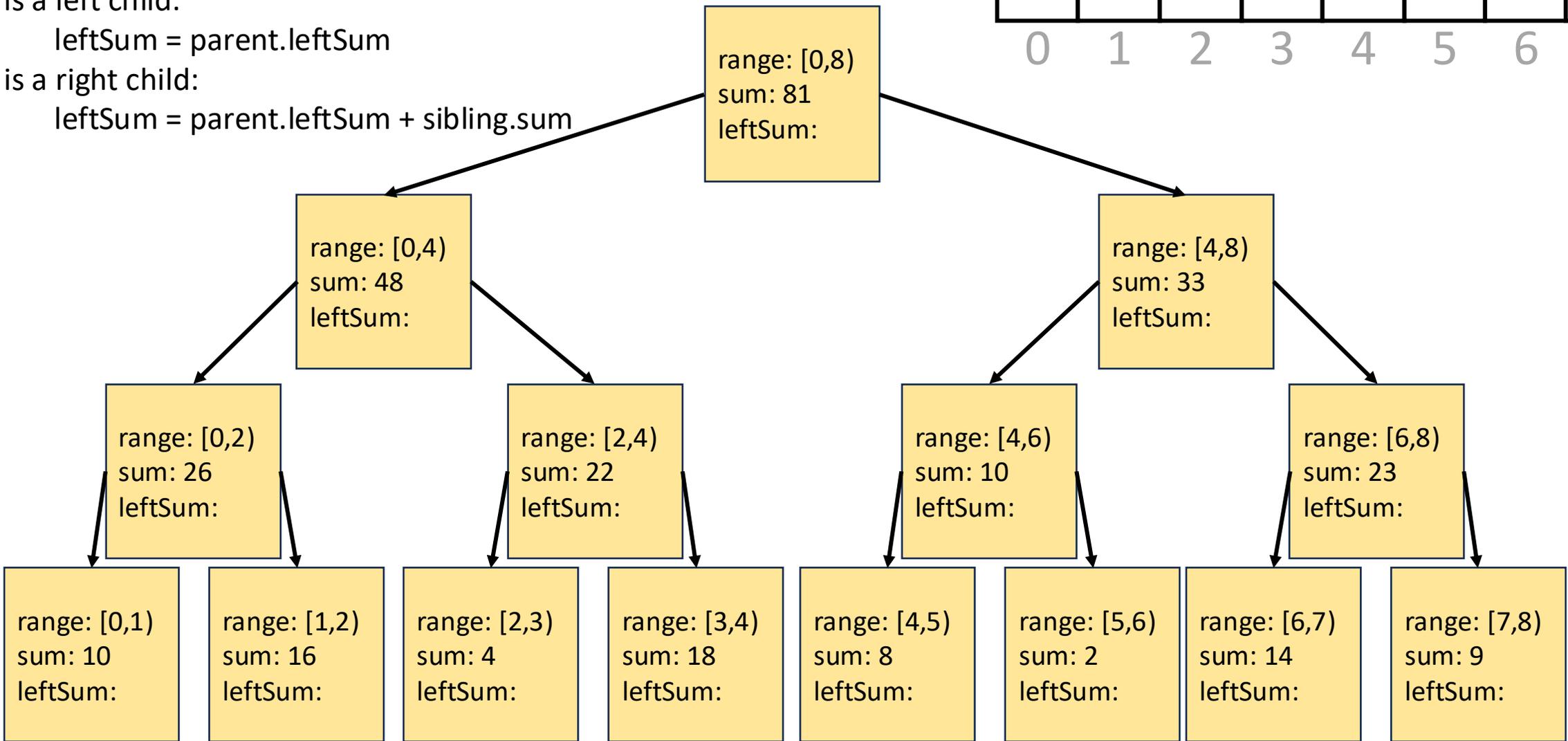|							

0   1   2   3   4   5   6   7

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

range: [0,8)
sum: 81
leftSum: 0

range: [0,4)
sum: 48
leftSum: 0

range: [4,8)
sum: 33
leftSum: 48

range: [0,2)
sum: 26
leftSum: 0

range: [2,4)
sum: 22
leftSum: 26

range: [4,6)
sum: 10
leftSum: 48

range: [6,8)
sum: 23
leftSum: 58

range: [0,1)
sum: 10
leftSum: 0

range: [1,2)
sum: 16
leftSum: 10

range: [2,3)
sum: 4
leftSum: 26

range: [3,4)
sum: 18
leftSum: 30

range: [4,5)
sum: 8
leftSum: 48

range: [5,6)
sum: 2
leftSum: 56

range: [6,7)
sum: 14
leftSum: 58

range: [7,8)
sum: 9
leftSum: 72
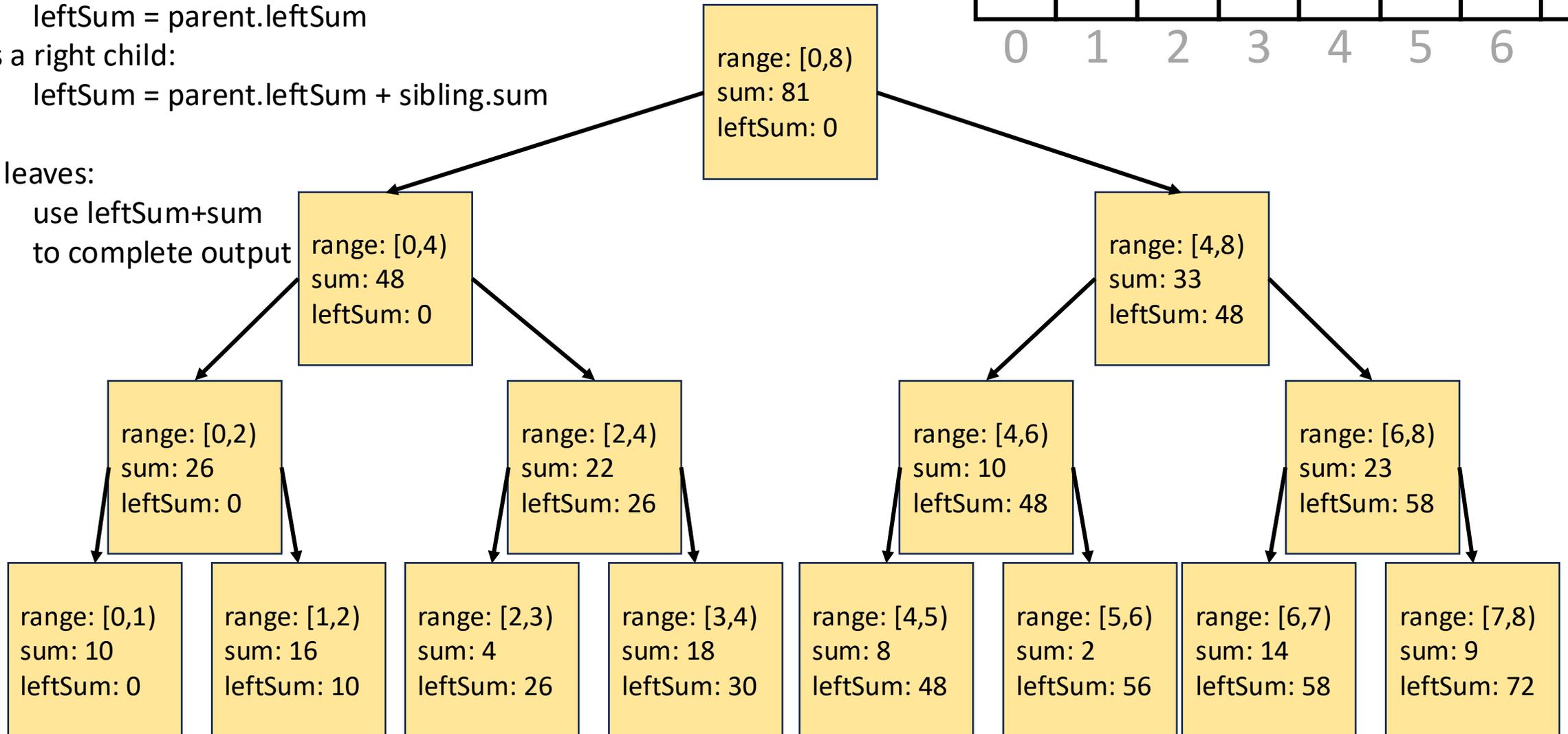
Step 2: fill in leftSum and Output

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

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

range: [0,8)
sum: 81
leftSum: 0

range: [0,4)
sum: 48
leftSum: 0

range: [4,8)
sum: 33
leftSum: 48

range: [0,2)
sum: 26
leftSum: 0

range: [2,4)
sum: 22
leftSum: 26

range: [4,6)
sum: 10
leftSum: 48

range: [6,8)
sum: 23
leftSum: 58

range: [0,1)
sum: 10
leftSum: 0

range: [1,2)
sum: 16
leftSum: 10

range: [2,3)
sum: 4
leftSum: 26

range: [3,4)
sum: 18
leftSum: 30

range: [4,5)
sum: 8
leftSum: 48

range: [5,6)
sum: 2
leftSum: 56

range: [6,7)
sum: 14
leftSum: 58

range: [7,8)
sum: 9
leftSum: 72
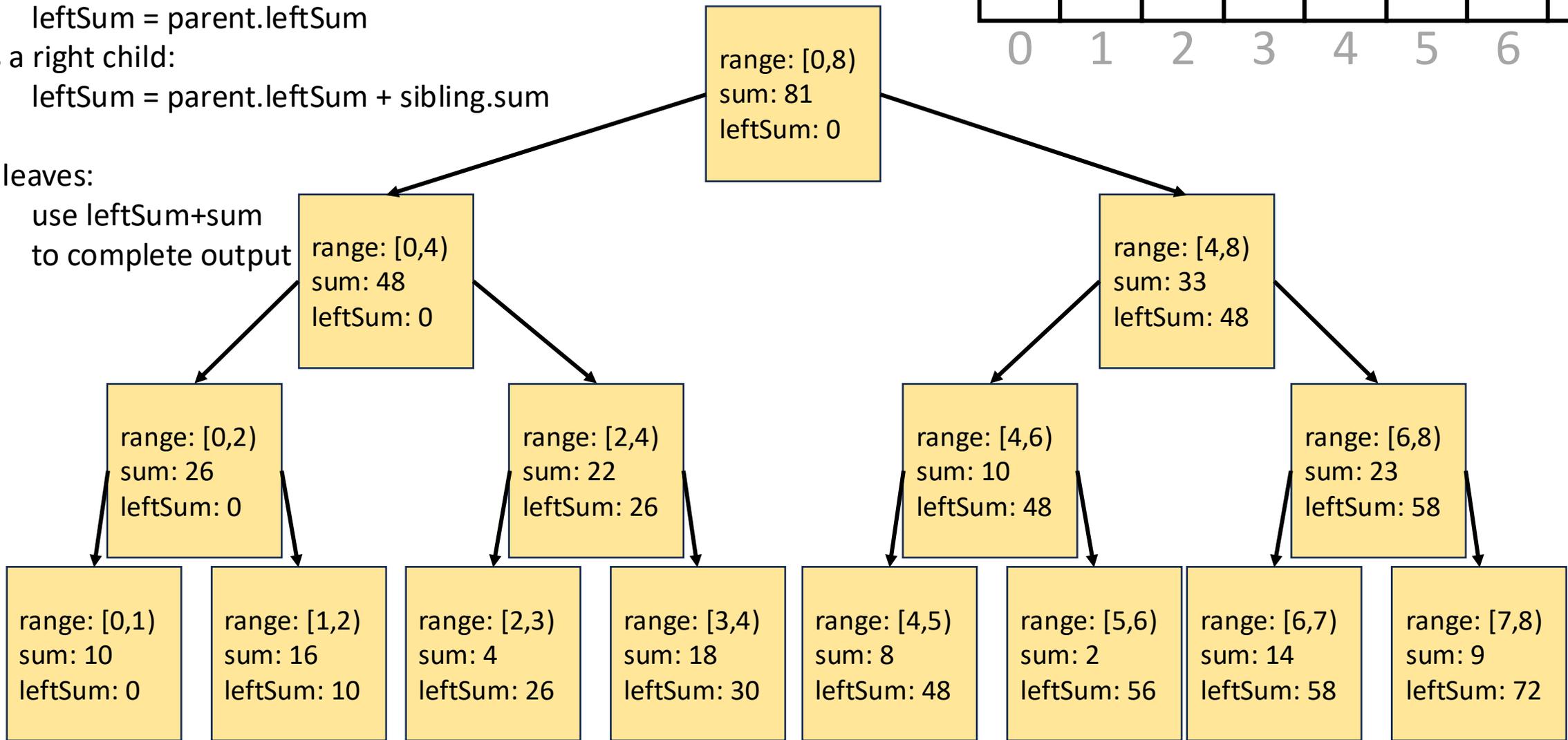
Step 2 pseudocode (Compute Leftsum)

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. <u>Divide and conquer:</u> call **CompleteTree**(curr.left) and **CompleteTree**(curr.right) in parallel threads
   2. <u>Wait</u> 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"

10	16	4	18	8	2	14	9
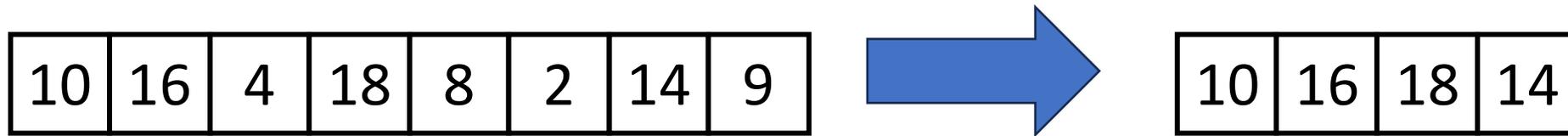
10	16	18	14

$$f(x) = x > 9$$

Input: | 10 | 16 | 4 | 18 | 8 | 2 | 14 | 9 | , $f(x) = x > 9$

Parallel Pack

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:

Input:

10	16	4	18	8	2	14	9

Map Result:

1	1	0	1	0	0	1	0

Prefix Result:

1	2	2	3	3	3	4	4

Output:

- 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