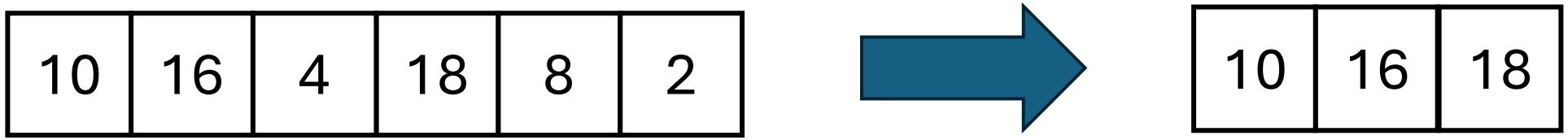
CSE 332 Summer 2026
Lecture 18: Parallel Prefix Sum, Parallel Runtime

Michael Whitmeyer

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

Pack/Filter

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



$f(x) = x > 9$

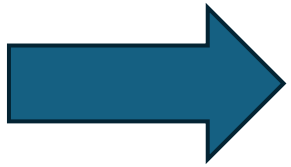
Question: Can we Pack/Filter in parallel?

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

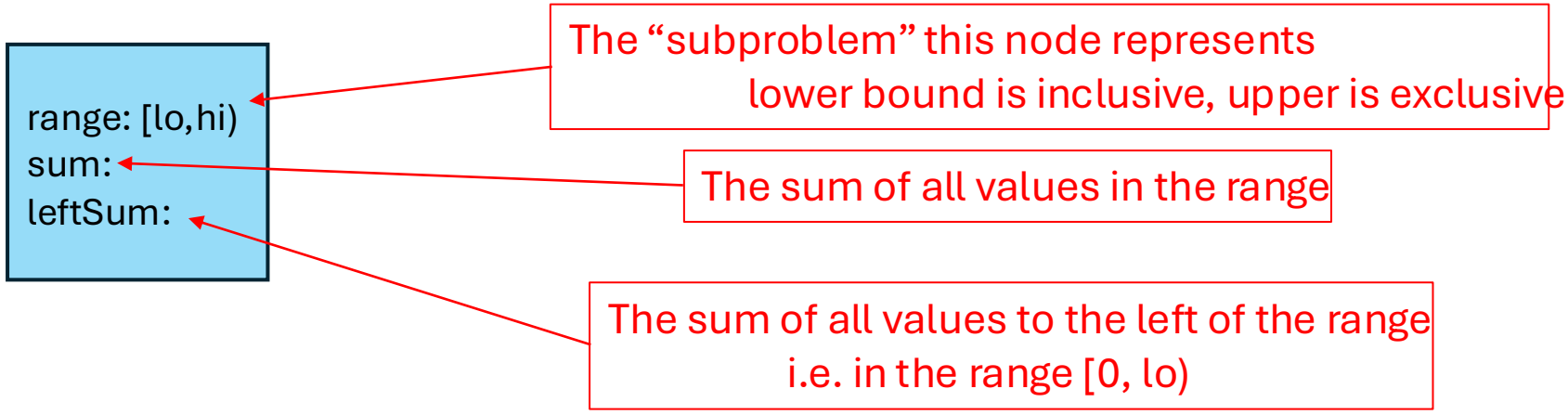
Later: will be useful for parallel packing/filtering!

Parallel Prefix Sum

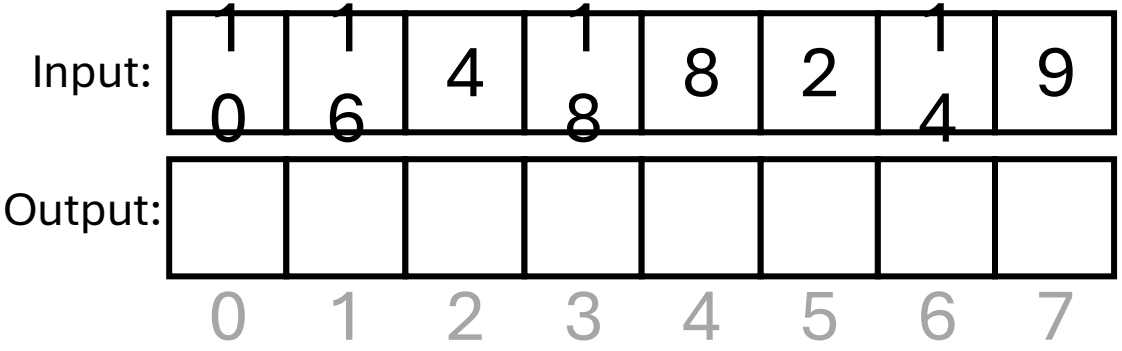
Richard Ladner
Allen School Faculty

- 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

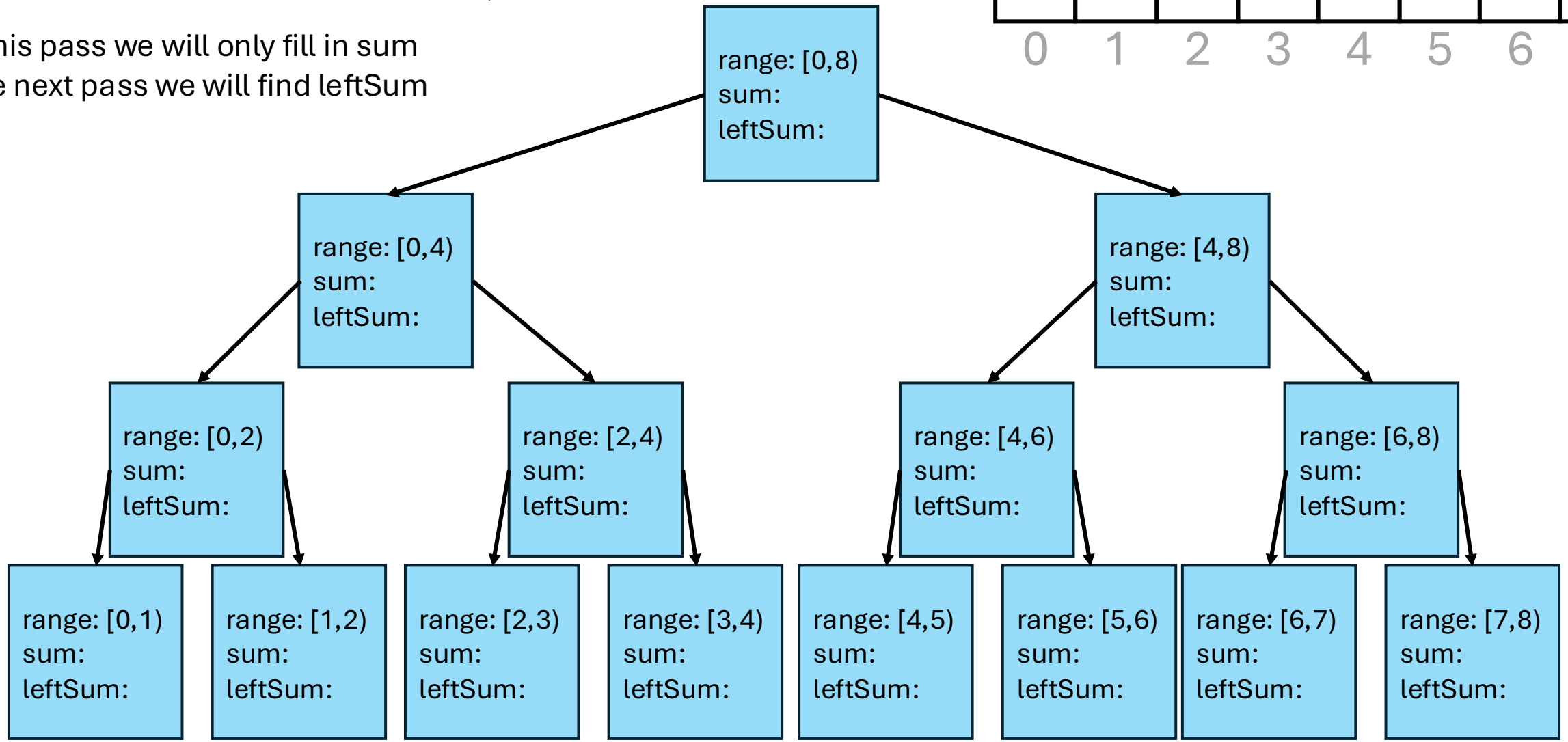
Tree Node:



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

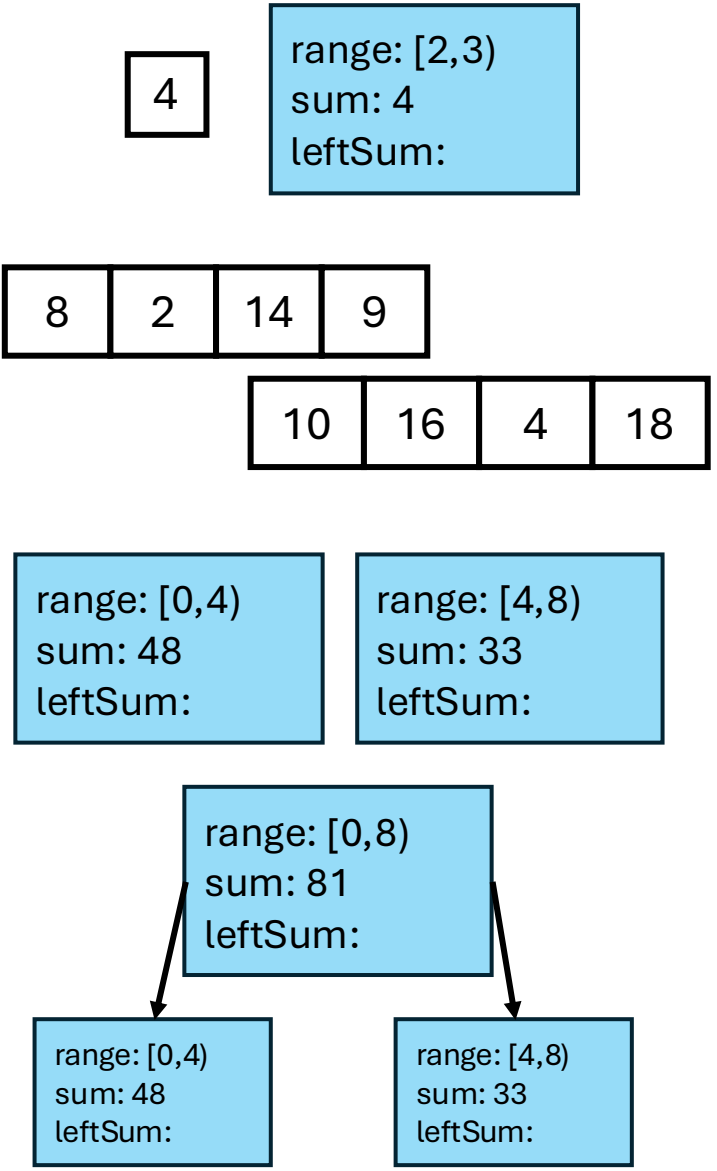


For this pass we will only fill in sum
In the next pass we will find leftSum



Step 1: Create a Tree, Fill in sum

10	16	4	18	8	2	14	9



- **Base Case**:
  - If the 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)

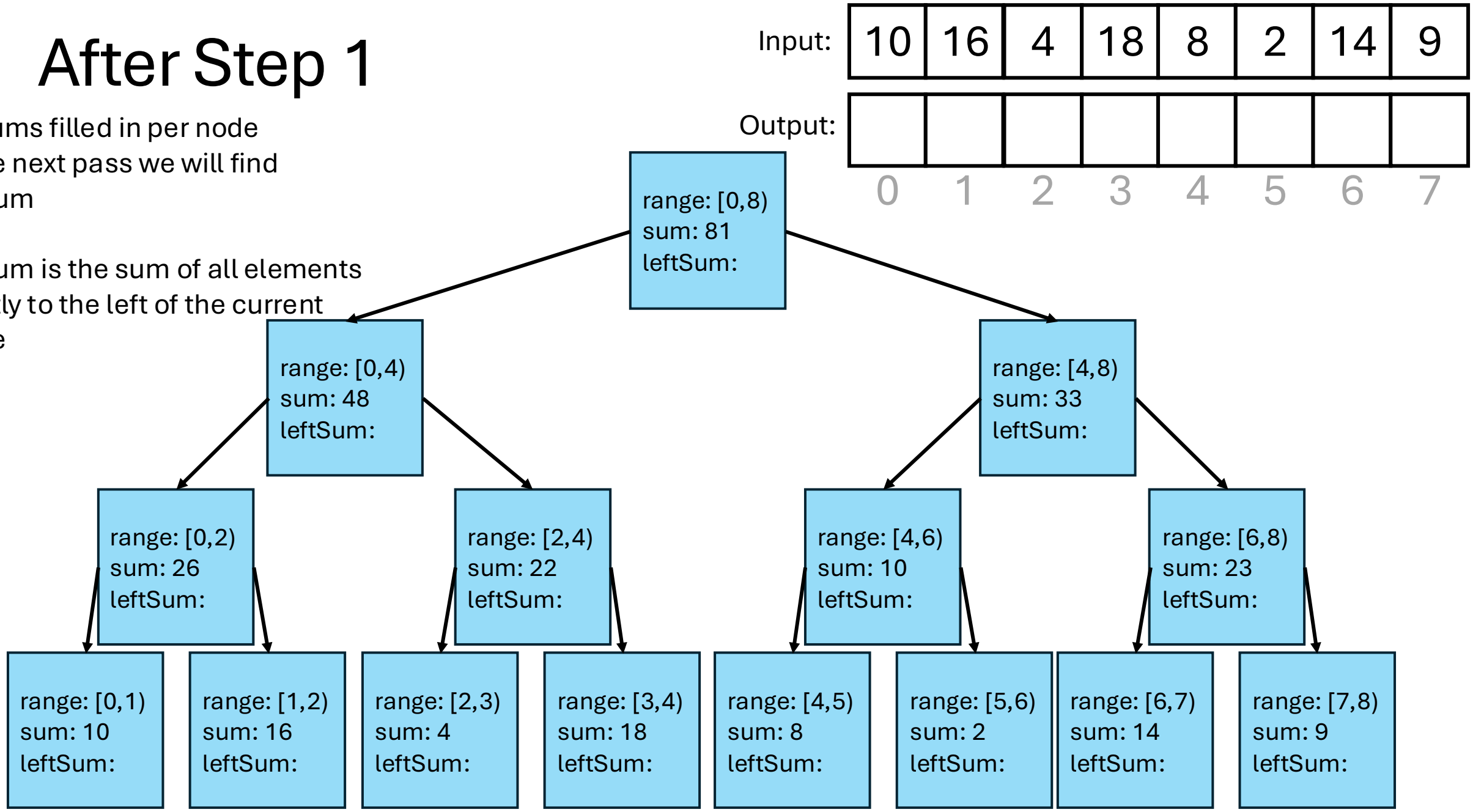
BuildTree(arr)

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. 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

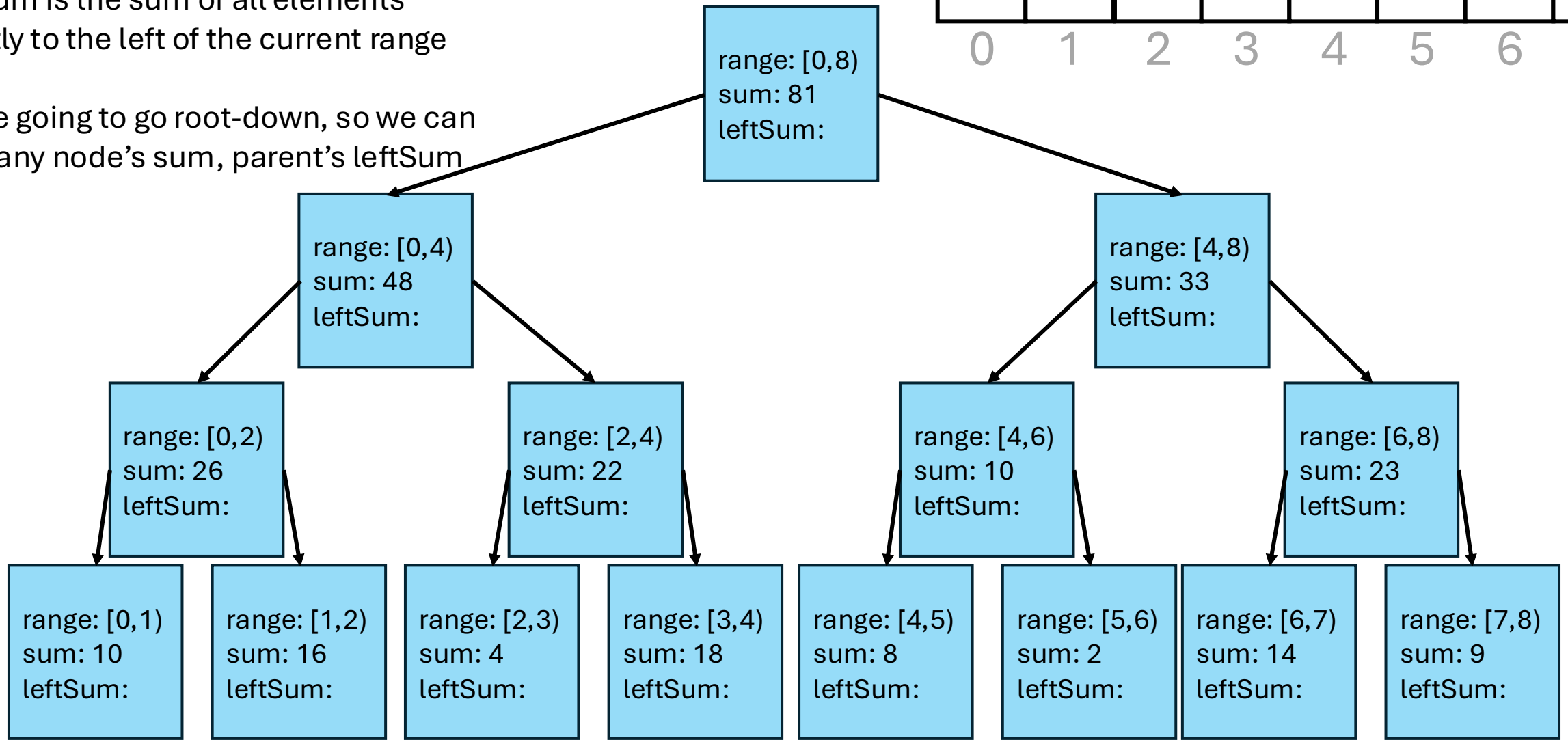
leftSum is the sum of all elements strictly to the left of the current range



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

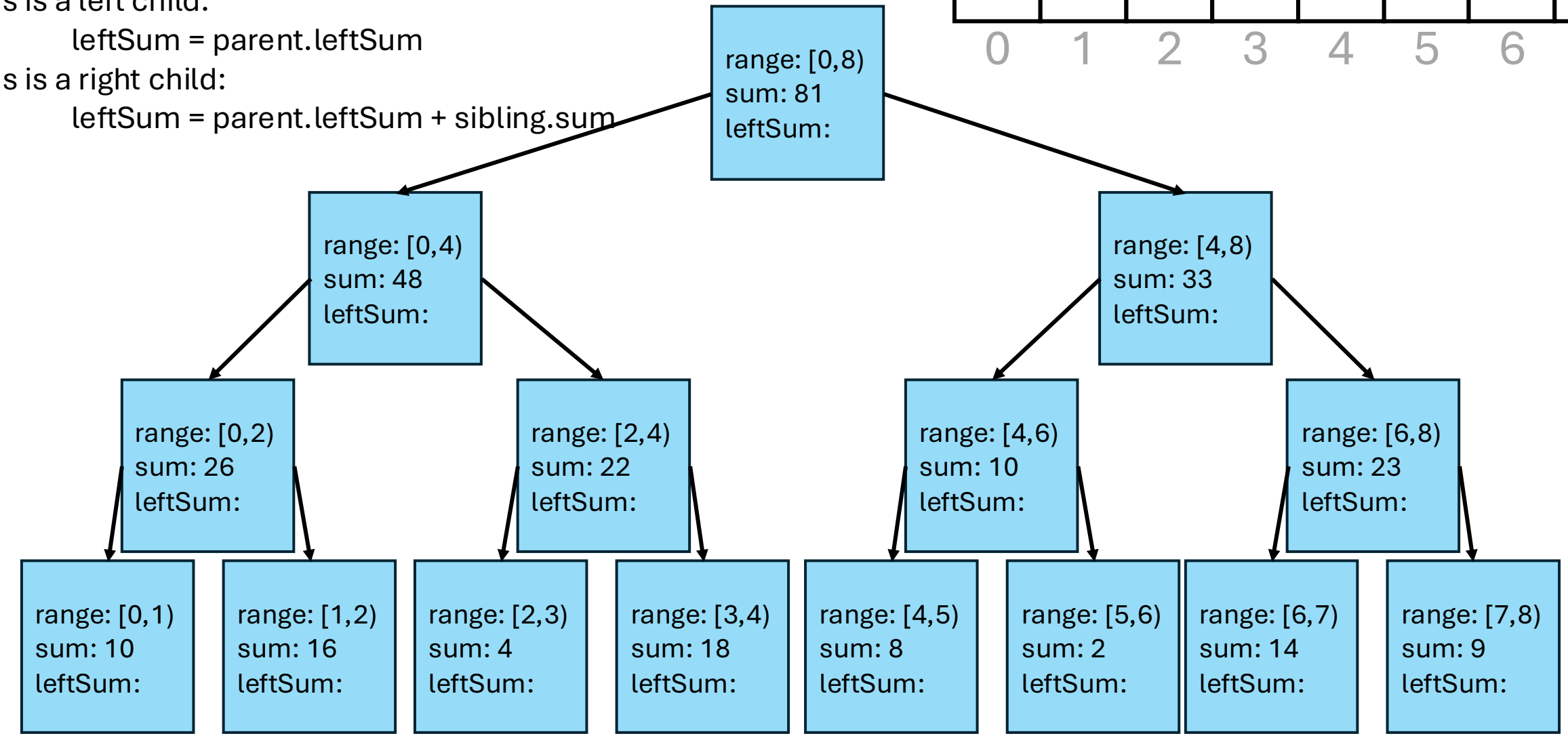
If this is a left child:

leftSum = parent.leftSum

If this is a right child:

leftSum = parent.leftSum + sibling.sum

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



Step 2: fill in leftSum and Output

If this is a left child:

leftSum = parent.leftSum

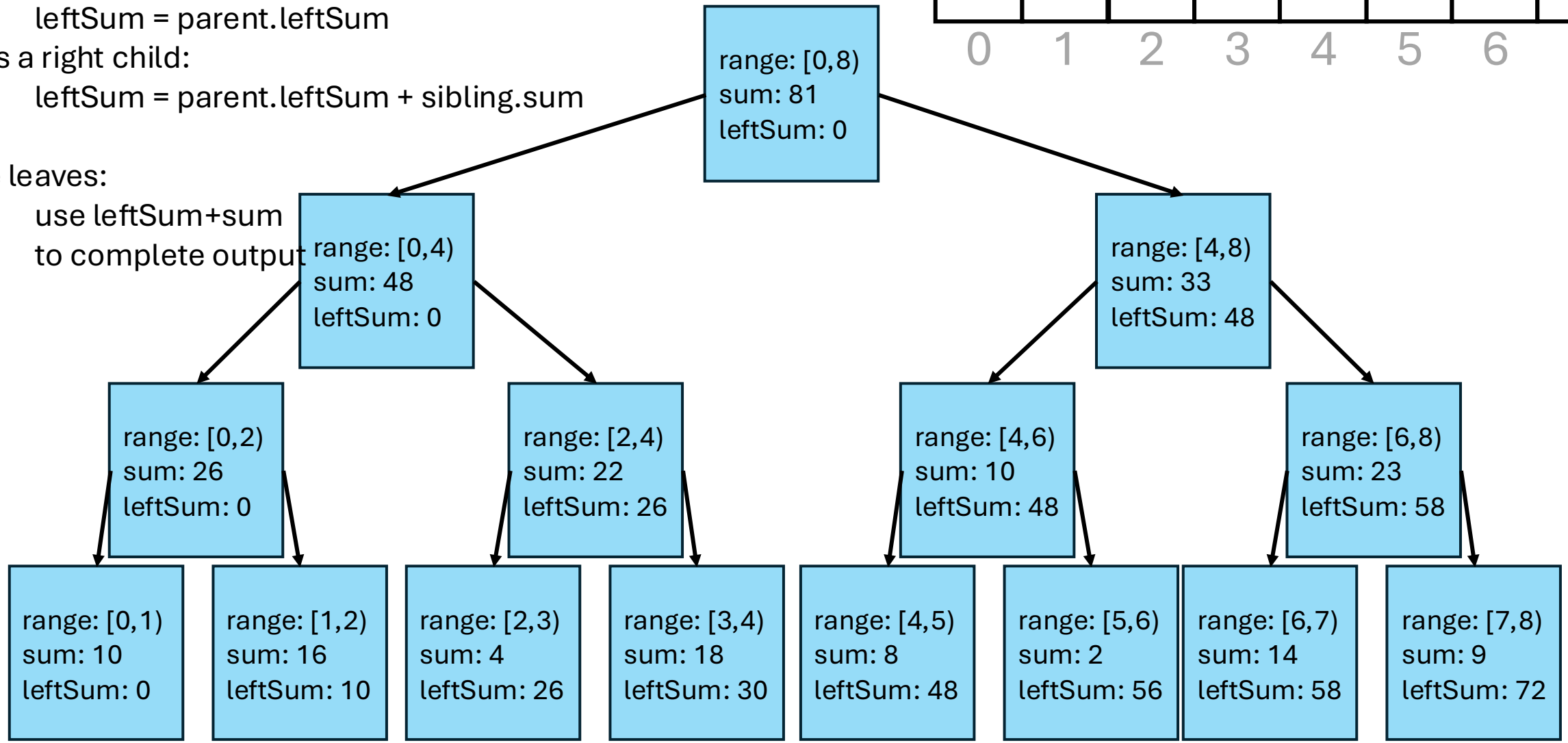
If this is a right child:

leftSum = parent.leftSum + sibling.sum

For the leaves:

use leftSum+sum to complete output

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



Step 2: fill in leftSum and Output

If this is a left child:

leftSum = parent.leftSum

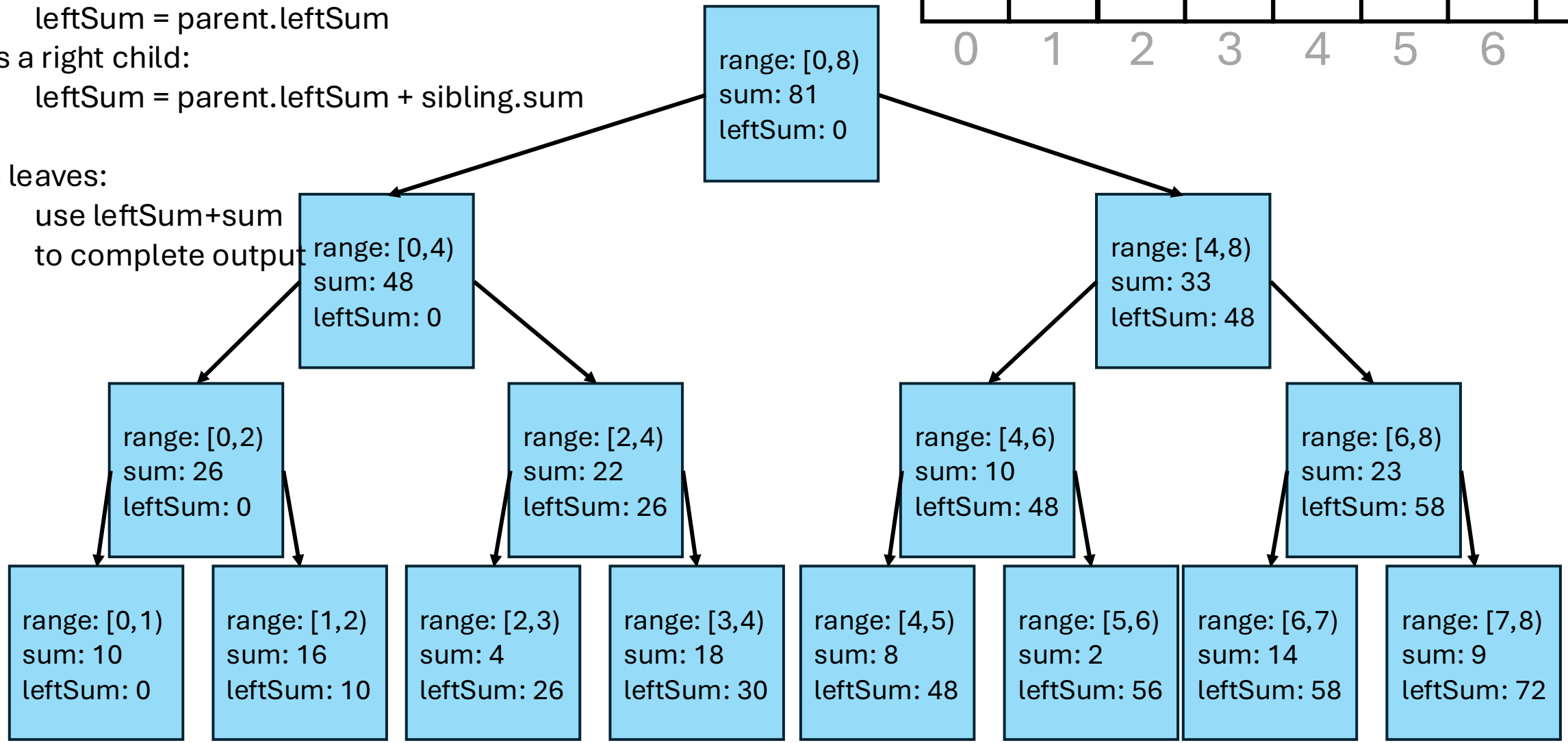
If this is a right child:

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)

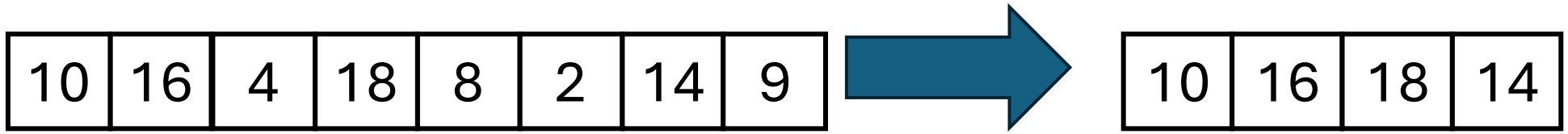
CompleteTree(Node curr)

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”



$f(x) = x > 9$

Input:

10	16	4	18	8	2	14	9

, $f(x) = x > 9$

Parallel Filter

Output:

10	16	18	14
0	1	2	3

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

1	1	0	1	0	0	1	0

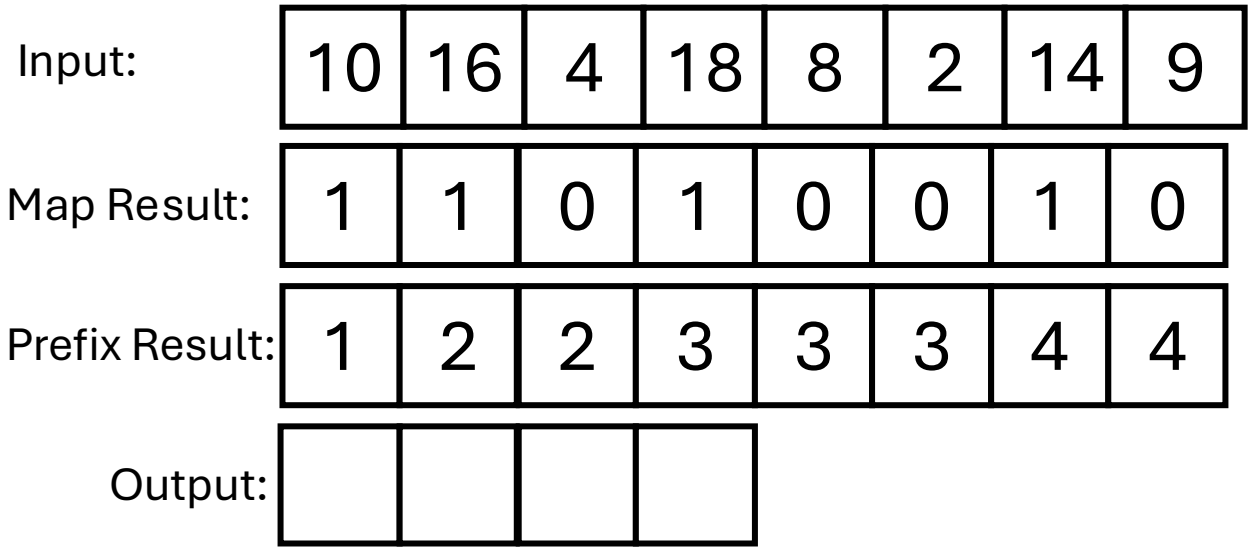
2. Do **prefix sum** on the result → count of true elements seen to the left of each position

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/Filter 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 **filter** on the array to remove all even values
  - Then do a **reduction** to multiply together that final result

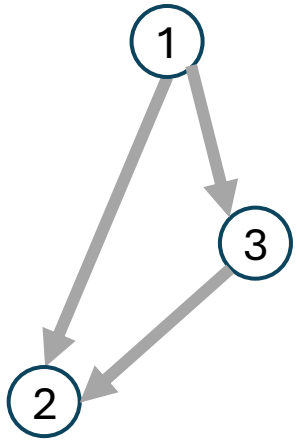
Runtime of Parallel Algorithms

Work and Span

- Let $T_P(n)$ be the running time if there are P processors available
- Two key measures of run time:
  - **Work**: How long it would take 1 processor, so $T_1(n)$
    - Just suppose all forks are done sequentially
    - Cumulative work all processors must complete
    - For array sum: $\Theta(n)$
  - **Span**: How long it would take an infinite number of processors, so $T_\infty(n)$
    - Theoretical ideal for parallelization
    - Longest "dependence chain" in the algorithm
    - Also called "critical path length" or "computation depth"
    - For array sum: $\Theta(\log n)$

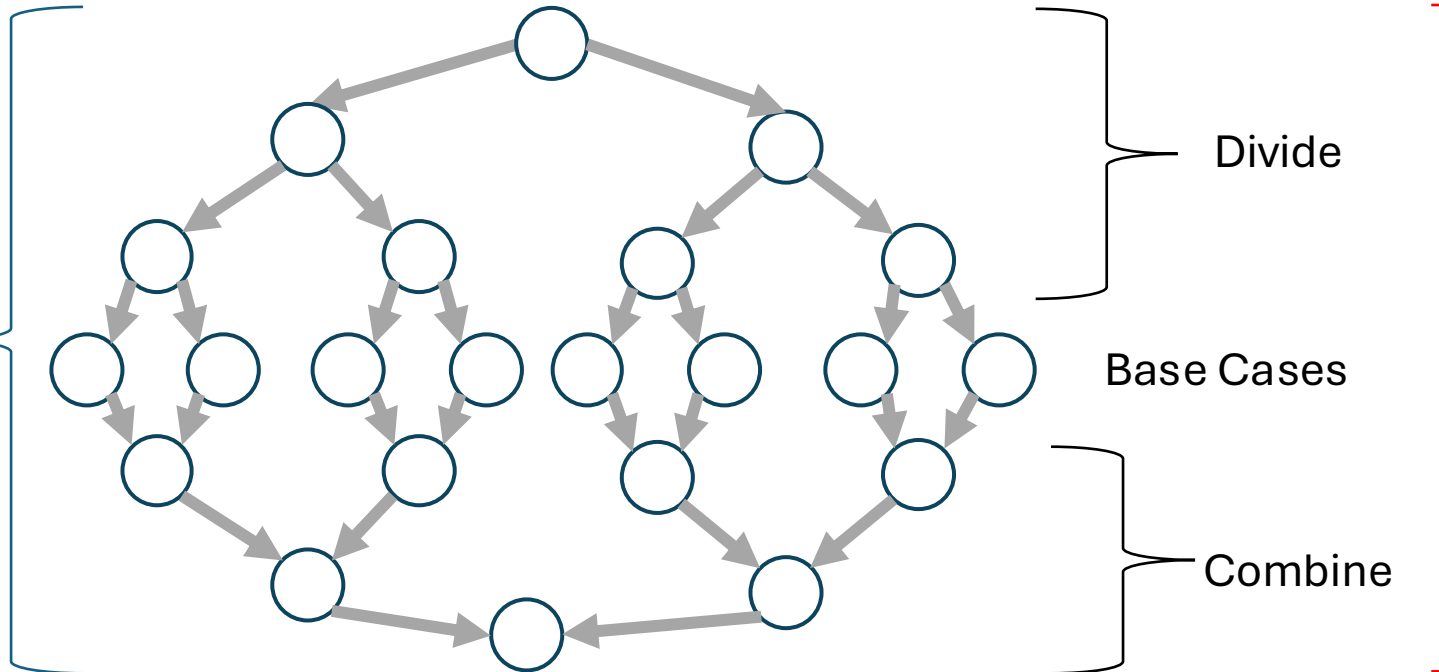
Directed Acyclic Graph (DAG)

- A directed graph that has no cycles
- Often used to depict dependencies
  - E.g. software dependencies, Java inheritance, dependencies among sequential tasks!



Task Dependency Graph

- "Sketches" what parts of the algorithm may be done in parallel vs. must be done in-order
  - Each node is a step of the algorithm that may depend on other steps (draw an edge)
  - This graph depicts a forkjoin algorithm. **Each node takes constant time**.
- Fork and Join each create a new node
  - When calling fork/compute
    - Algorithm creates two new threads, there is a dependency from the creating code to the code done by these threads
  - When calling join
    - There is a dependency from the code done by the other thread to the code after join



Work = sum of the running times of all nodes

Span = longest path from start node to end node.

Work Law

- States that $P \cdot T_P(n) \geq T_1(n)$
  - P processors can do at most P things in parallel
    - Work must match the sum of the operations done by all processors, so if this does not hold then the parallel algorithm somehow skipped steps that sequential version would have done.
  - If the "division of labor" across processors is uneven then it might be that $P \cdot T_P(n) > T_1(n)$

Asymptotically Optimal T_P

- T_P cannot be better than $\frac{T_1}{P}$
  - Because of the Work Law
- T_P cannot be better than T_∞
  - A finite number of processors can't outperform an infinite number ("Span Law")
- Considering both of these, we can characterize the best-case scenario for T_P
  - $T_P(n) \in \Omega\left(\frac{T_1(n)}{P} + T_\infty(n)\right)$
  - $T_1(n)/P$ dominates for small P, $T_\infty(n)$ dominates for large P
- ForkJoin Framework gives an expected time guarantee of asymptotically optimal!

More Vocab

- Speedup:
  - How much faster (than one processor) do we get for more processors
    - Identifies how well the algorithm scales as processors increases
    - May be different for different algorithms
  - $T_1(n)/T_P(n)$
  - If $\frac{T_1}{T_P} = P$ this is “perfect linear speedup” (best possible)
- Parallelism
  - Maximum possible speedup
  - T_1/T_∞
  - At some point more processors won’t be more helpful, when that point is depends on the span
- Writing parallel algorithms is about decreasing span without substantially increasing work

And now for some bad news…

- In practice it's common for your program to have:
  - Parts that parallelize well
    - Maps/reduces/filters over arrays and other data structures
    - Anything ForkJoin!
    - Tasks that don't need to access a shared data structure
  - Parts that don't parallelize at all
    - Reading a linked list, getting input, computations where each step needs the results of previous step, concurrency concerns forcing mutual exclusion
- These unparallelizable parts can turn out to be a big bottleneck

Amdahl's Law (the bad news)

- Suppose $T_1 = 1$
  - Work for the entire program is 1
- Let S be the proportion of the program that cannot be parallelized
  - $T_1 = S + (1 - S) = 1$
- Suppose we get perfect linear speedup on the parallel portion
  - $T_P = S + \frac{1-S}{P}$
- For the entire program, the speedup is:
  - $\frac{T_1}{T_P} = \frac{1}{S+\frac{1-S}{P}}$
- The parallelism (infinite processors) is:
  - $\frac{T_1}{T_\infty} = \frac{1}{S}$

Amdahl's Law Example

- Suppose 2/3 of your program is parallelizable, but 1/3 is not.
  - $S = \frac{1}{3}$
  - $T_1 = \frac{2}{3} + \frac{1}{3} = 1$
- $T_P = S + \frac{1-S}{P} = \frac{1}{3} + \frac{2/3}{P}$
- If T_1 is 100 seconds:
  - $T_P = 33 + \frac{67}{P}$
  - $T_2 = 33 + \frac{67}{2} \approx 67$
  - $T_3 = 33 + \frac{67}{3} \approx 55$
  - $T_6 = 33 + \frac{67}{6} \approx 44$
  - $T_{12} = 33 + \frac{67}{12} \approx 39$

Notice:

- We got a lot of speedup with the second processor (23%)
- Adding a third processor only gave half as much speedup (12%)
- After going up to 6 processors, we still only got 11% speedup
- No matter how many processors we add, we will never get more 11% additional improvement

Conclusion:

When a portion of our code is sequential, the improvement gained from more and more processors diminishes very quickly.

Conclusion

- Even with many *many* processors the sequential part of your program becomes a bottleneck
- Parallelizable code requires skill and insight from the developer to recognize where parallelism is possible, and how to do it well.

Concurrency

Memory Sharing With ForkJoin

- Structure of ForkJoin:
  - All threads are executing the same algorithm
  - Each task is responsible for **its own portion** of the input/output
  - If one task needs another's result, use join() to ensure it uses the final answer

- This does not help when:
  - Memory accessed by threads is overlapping or unpredictable
  - Threads are doing independent tasks using same resources (rather than implementing the same algorithm)

Example: Shared Queue

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Imagine two threads are both using the **same** linked list based queue.

What could go wrong?

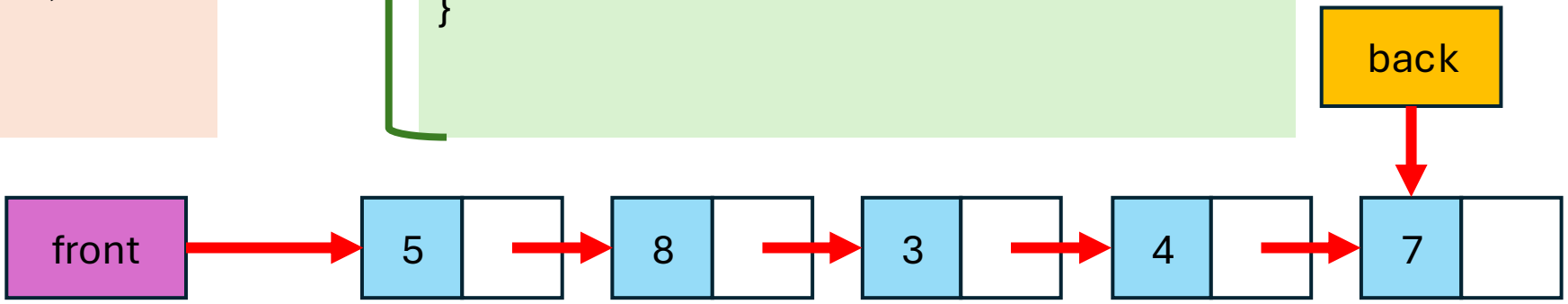
Shared Queue Interleaving

Thread

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new
Node(x);
                back = back.next;
        }
}
```

Thread

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```



Empty Shared Queue

Suppose the Queue is empty, and the threads execute in this order. Assume Thread 1 enqueues 1, and Thread 2 enqueues 2.

Thread

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new
Node(x);
                back = back.next;
        }
}
```

Thread

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

front

back

Thread 1 – first two lines

Thread 1 has decided the queue is empty

Thread 1

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Thread 2

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

front

back

Thread 2 – all lines

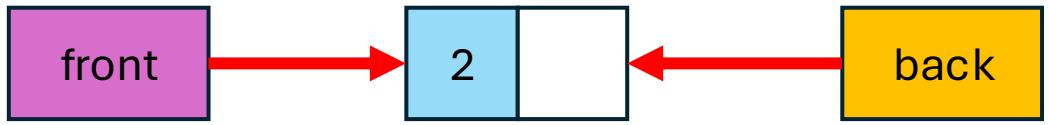
Thread 1 has decided the queue is empty
Meanwhile, thread 2 enqueues 2

Thread 1

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new
Node(x);
                back = back.next;
        }
}
```

Thread 2

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new
Node(x);
                back = back.next;
        }
}
```



Thread 1 – remaining lines

Thread 1 has decided the queue is empty
Meanwhile, thread 2 enqueues 2
Thread 1 continues as if the queue is still empty, overwriting Thread 2's work

Thread 1

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Thread 2

```
enqueue(x){
        if ( back == null ){
                back = new Node(x);
                front = back;
        }
        else {
                back.next = new Node(x);
                back = back.next;
        }
}
```

Interleaving

- Due to time slicing, a thread can be interrupted at any time
  - Between any two lines of code
  - Within a single line of code
- The sequence that operations occur across two threads is called an interleaving
- Without doing anything else, we have no control over how different threads might be interleaved

Concurrent Programming

- Concurrency:
  - Correctly and efficiently managing access to shared resources across multiple possibly-simultaneous tasks
- Requires synchronization to avoid incorrect simultaneous access
  - Use some way of “blocking” other tasks from using a resource when another modifies it or makes decisions based on its state
  - That blocking task will free up the resource when it’s done
- Warning:
  - Because we have no control over when threads are scheduled by the OS, even correct implementations are highly non-deterministic
  - Errors are hard to reproduce, which complicates debugging

Analogue Example – Data Race

1. Count to 10 aloud
2. Clap three times
3. Erase the contents of the box
4. Write your name in the box,
   pausing 2 seconds between letters
5. Wave your arms in the air

Analogue Example – With “Mutual Exclusion”

1. Count to 10 aloud
2. Clap three times
3. **Pick up the trident**
4. Erase the contents of the box
5. Write your name in the box
6. **Put down the trident**
7. Wave your arms in the air

There is only one trident, so no two threads (both executing this same code) can execute lines 4 and 5 at the same time.

All threads after the first must wait at line 3 for the trident to become available.

The trident causes any one thread to “lock out” (exclude) all other threads.

Any thread being between lines 3 and 6 excludes all other threads, so this is called “mutual exclusion”.

The trident itself we call a “**lock**”