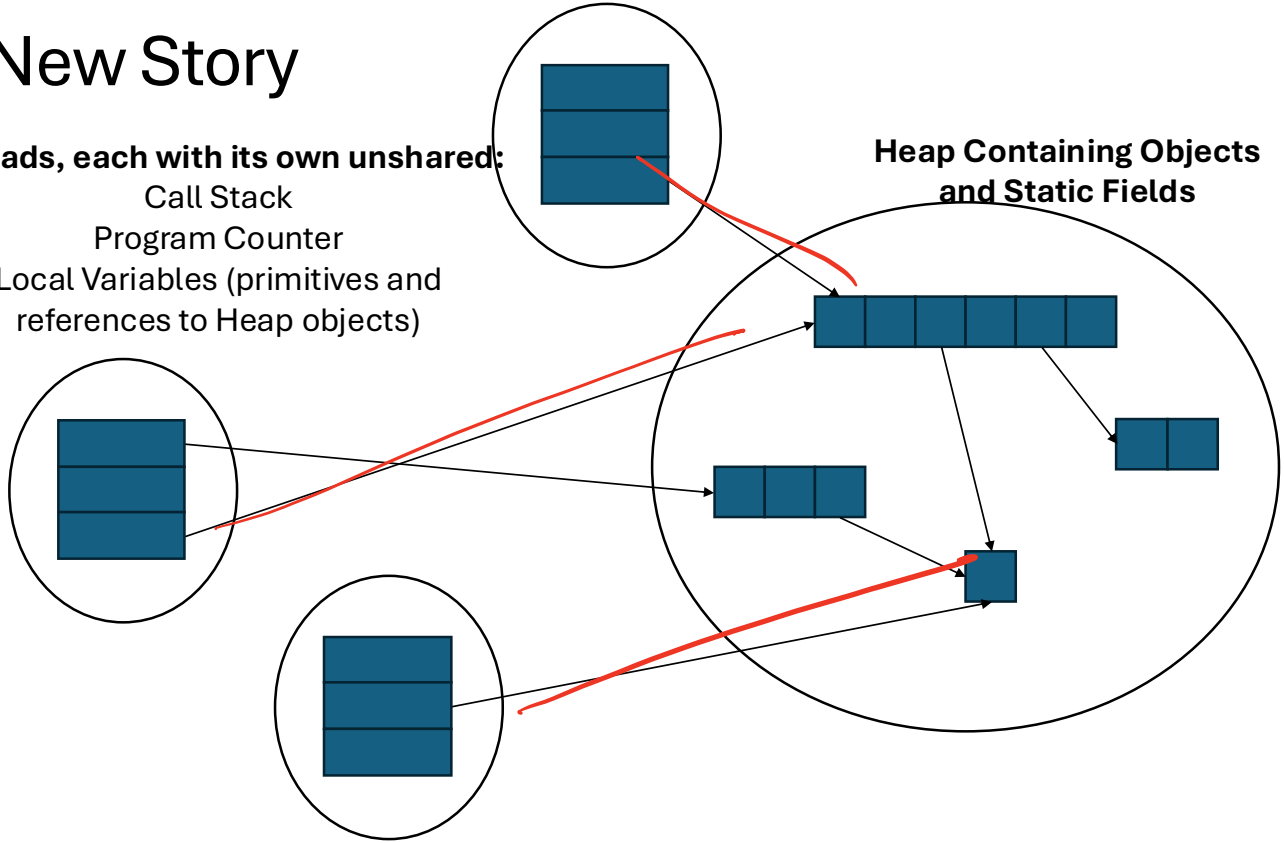
CSE 332 Summer 2026 Lecture 17: Forkjoin

Michael Whitmeyer

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

New Story

Threads, each with its own unshared:
Call Stack
Program Counter
Local Variables (primitives and references to Heap objects)



Accomplishing this in Java (**Java.lang.Thread**)

1. Create class C extending **java.lang.Thread**
   1. C must have a **run** method (acts as **main**)
2. Call C's **start** method to
   1. create a new thread (i.e. parallel task, not Thread object in java)
   2. execute **run** in that separate, parallel thread

- Calling "**run**" directly causes the program to execute "**run**" sequentially

Parallel Divide and Conquer Pseudocode

$\ell \approx$ 1000's

RecursiveSum(arr)

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

Divide and Conquer with Threads (optimized)

```
class SumThread extends java.lang.Thread {
        public void run(){ // override
                if(hi – lo < SEQUENTIAL_CUTOFF) // “base case”
                        for(int i=lo; i < hi; i++) ans += arr[i];
                else {
                        SumThread left = new SumThread(arr,lo,(hi+lo)/2); // divide
                        SumThread right= new SumThread(arr,(hi+lo)/2,hi); // divide
                        left.start(); // conquer in parallel in new thread
                        right.run(); // conquer in this thread
                        left.join(); // don’t move this up a line – why?
                        //right.join();
                        ans = left.ans + right.ans; // combine
                }
        }
}
int sum(int[] arr){ // just make one thread!
        SumThread t = new SumThread(arr,0,arr.length);
        t.run();
        return t.ans; }
```

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**

Divide and Conquer with ForkJoin

```
class SumTask extends RecursiveTask<Integer> {
        int lo; int hi; int[] arr; // fields to know what to do
        SumTask(int[] a, int l, int h) { ... } // constructor
        protected Integer compute(){ // return answer
                if(hi – lo < SEQUENTIAL_CUTOFF) {  // base case
                        int ans = 0; // local var, not a field
                        for(int i=lo; i < hi; i++) {
                                ans += arr[i]; return ans; }
                else {
                        SumTask left = new SumTask(arr,lo,(hi+lo)/2); // divide
                        SumTask right= new SumTask(arr,(hi+lo)/2,hi); // divide
                        left.fork()  // conquer in parallel
                        int rightAns = right.compute(); // conquer in this thread
                        int leftAns = left.join(); // wait
                        return leftAns + rightAns; // combine
                }
        }
}
```

Divide and Conquer with ForkJoin (continued)

```
static final ForkJoinPool POOL = new ForkJoinPool();
static int parallelSum(int[] arr){
        SumTask task = new SumTask(arr,0,arr.length)
        return POOL.invoke(task); // invoke returns the value
compute returns
}
```

ForkJoin Pseudocode for Summing

GenericTask(arr)

1. **If** len(arr) < ℓ: return sum 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)

fork()

compute()

join()

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

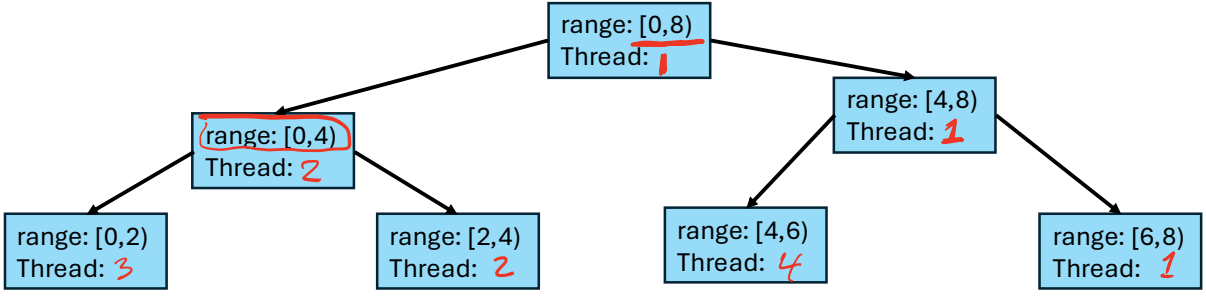
fork()

Save

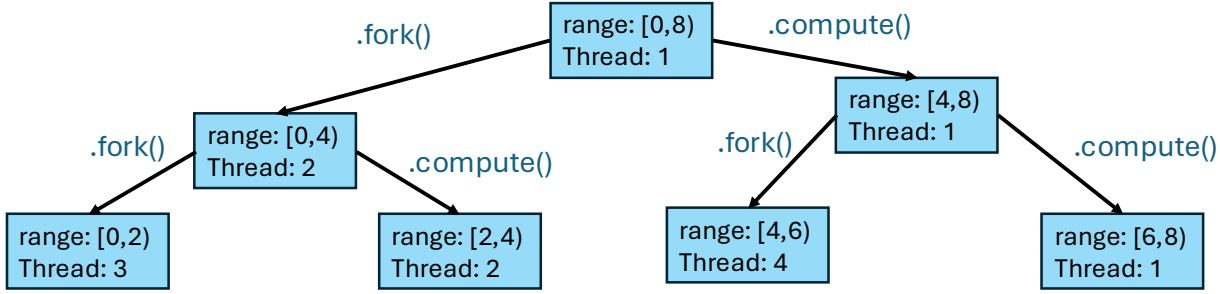
compute()

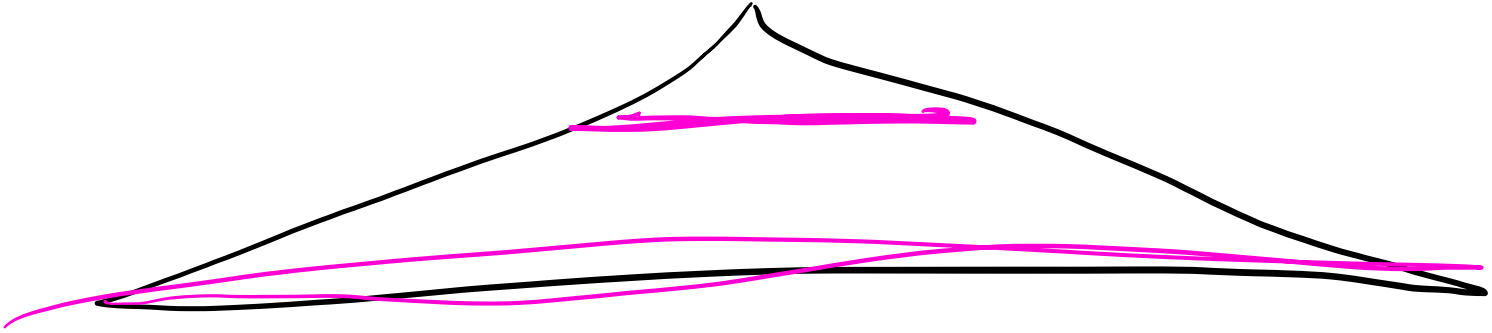
join()

ForkJoin Picture

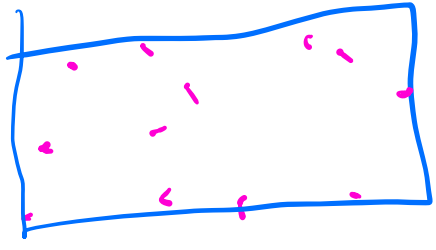


ForkJoin Picture





Other Problems that can be solved similarly



- Element Search
  - Is the value 17 in the array?
- Counting items with a certain property
  - How many elements of the array are divisible by 5?
- Checking if the array is sorted
- Find the smallest rectangle that covers all points in the array

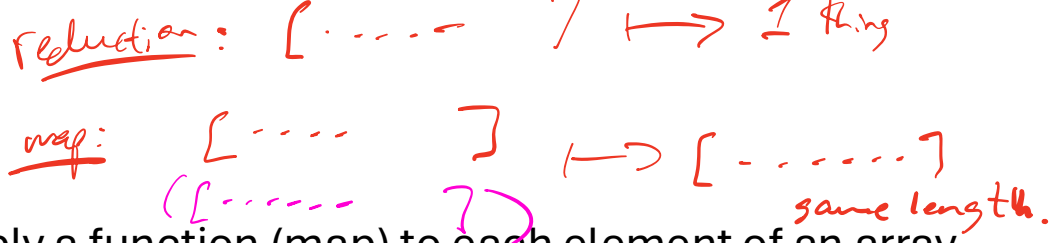
$[(x_1, y_1), (x_2, y_2)]$

- Find the first thing that satisfies a property
  - What is the leftmost item that is divisible by 20?

Reductions/Folds

- All examples of a category of computation called a reduction
  - We “reduce” all elements in an array to a single item
  - Requires operation done among elements is **associative**
    - $(x + y) + z = x + (y + z)$
    - min(min(x,y), z) = min(x, min(y, z))
  - The “single item” can itself be complex
    - E.g. create a histogram of results from an array of trials

Map

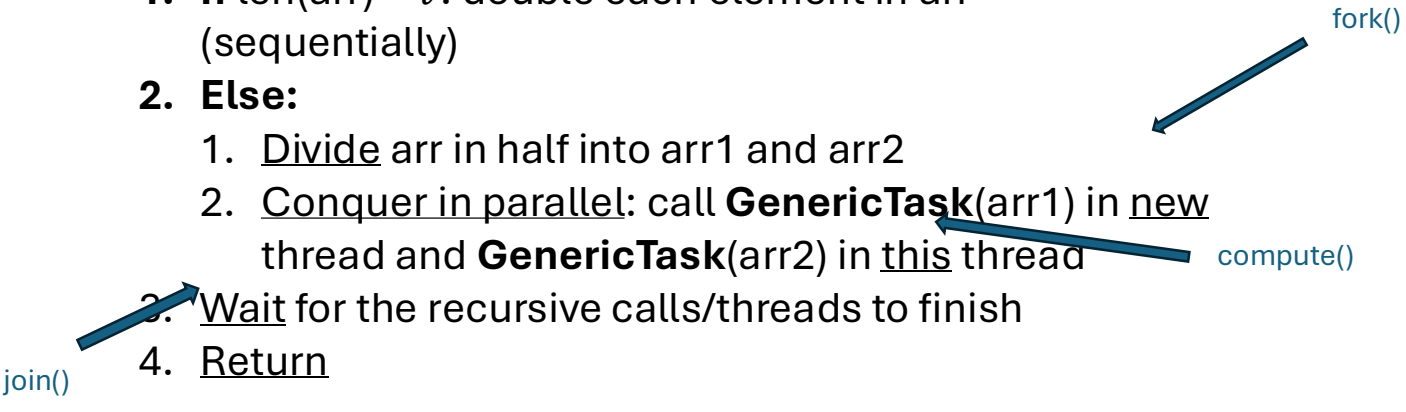


- **New task**: Apply a function (map) to each element of an array
- Examples:
  - Vector addition:
    - sum[i] = arr1[i] + arr2[i]
  - Function application:
    - out[i] = f(arr[i]);
- **Observation**: maps also parallelizable in the same way!
  - No need for combination step

ForkJoin Pseudocode for Doubling each element

GenericTask(arr)

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



ForkJoin Pseudocode for ~~Doubling each element~~ Applying map f

GenericTask(arr)

1. **If** len(arr) < ℓ: Apply f to ~~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()

Map with ForkJoin

```
class AddVecs extends RecursiveAction {
        int lo; int hi; int[] arr; // fields to know what to do
        AddVecs(int[] a, int[] b, int[] sum, int l, int h) { … }
        protected void compute(){// return answer
                if(hi – lo < SEQUENTIAL_CUTOFF) {  // base case
                        for(int i=lo; i < hi; i++) {
                                sum[i] = a[i] + b[i];}
                else {
                        AddTask left = new AddVecs(a,b,sum,lo,(hi+lo)/2); // divide
                        AddTask right= new AddVecs(a,b,sum,(hi+lo)/2,hi); // divide
                        left.fork(); // fork a thread and calls compute (conquer)
                        right.compute(); //call compute directly (conquer)
                        left.join(); // wait for thread to finish
                        return; // combine
                }
        }
}
```

Map with ForkJoin (continued)

```
static final ForkJoinPool POOL = new ForkJoinPool();
int[] add(int[] a, int[] b){
        int[] ans = new int[a.length];
        AddVecs task = new AddVecs(a, b, ans, 0, a.length)
        POOL.invoke(task);
        return ans;
}
```

Maps and Reductions

- “Workhorse” constructs in parallel programming
- Many problems can be written in terms of maps and reductions
- With practice, writing them will become second nature
  - Like how over time for loops and if statements have gotten easier

Map/Reduction Example

- **Task:** Multiply together the lengths of all the odd-length strings in array

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

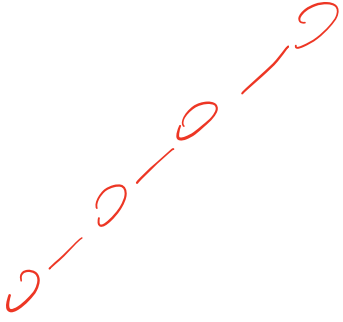
Which Data Structures are “Suitable” for Parallelism?

- 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 (✓)

if you are performing a task on each element of linked list that requires $O(m)$ time,

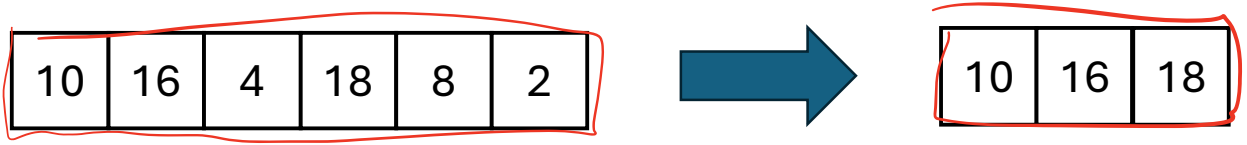
then sequential: $O(n \cdot m)$

parallel: $O(n+m)$



Pack/Filter

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



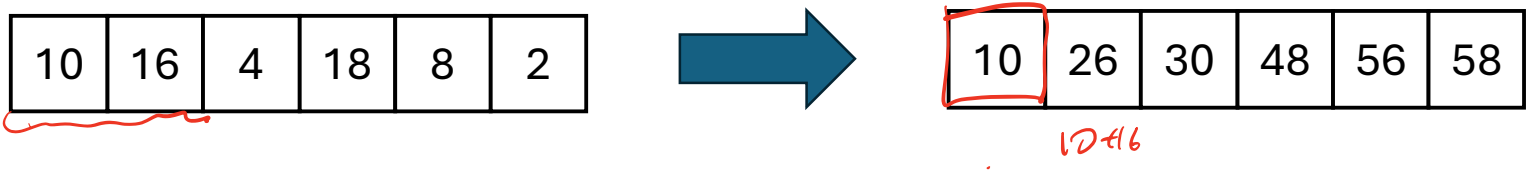
$f(x) = x > 9$

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



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

1st

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

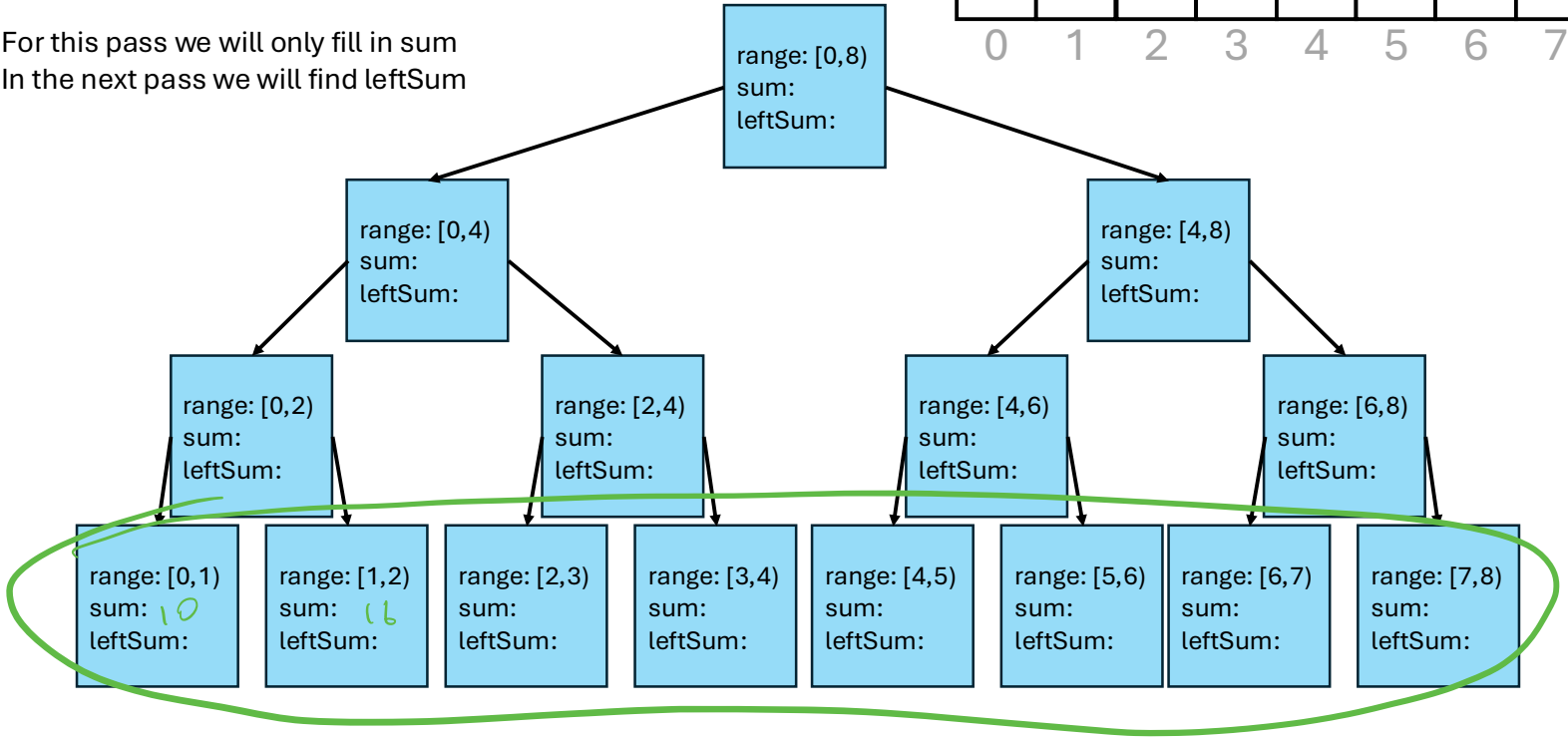
The sum of all values to the left of the range
i.e. in the range [0, lo)

Step 1: Using D&C

Create a Tree, Fill in sum

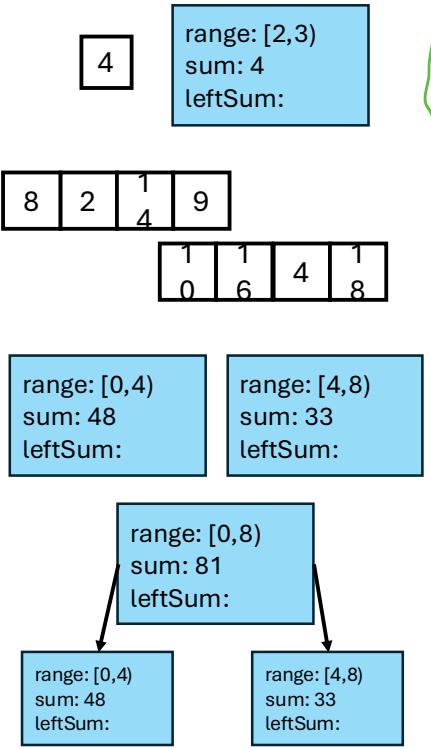
Input:	10	16	4	18	8	2	14	9
Output:								
	0	1	2	3	4	5	6	7

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



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)

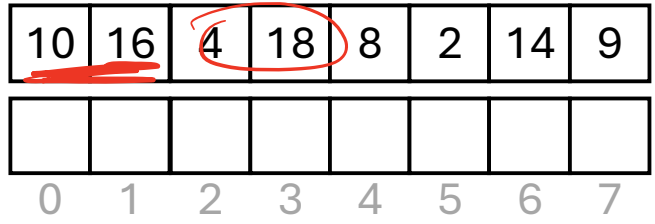
BuildTree(arr)

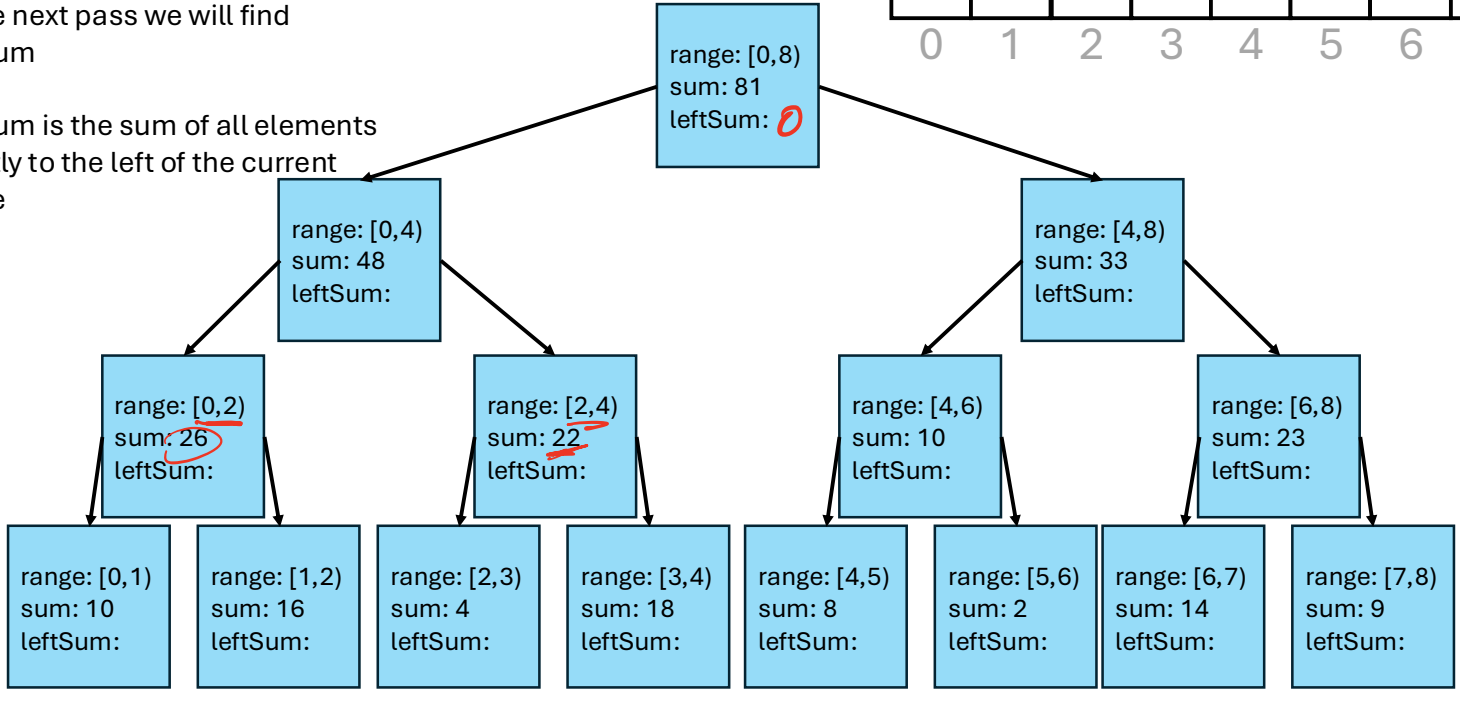
1. **If** len(arr) < ℓ:
    1. Create new TreeNode Leaf
    2. Set Leaf.sum = sum of values in arr and return Leaf
2. **Else:**
    1. Divide arr in half into arr1 and arr2
    2. Conquer: TreeNode leftChild = **BuildTree**(arr1) in new thread, TreeNode rightChild = **BuildTree**(arr2) in this thread
    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

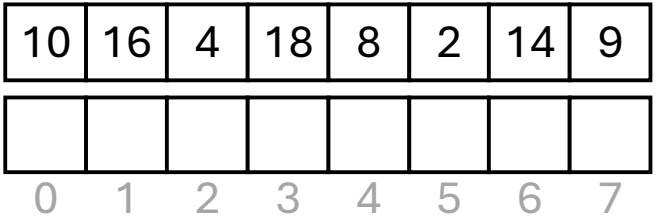


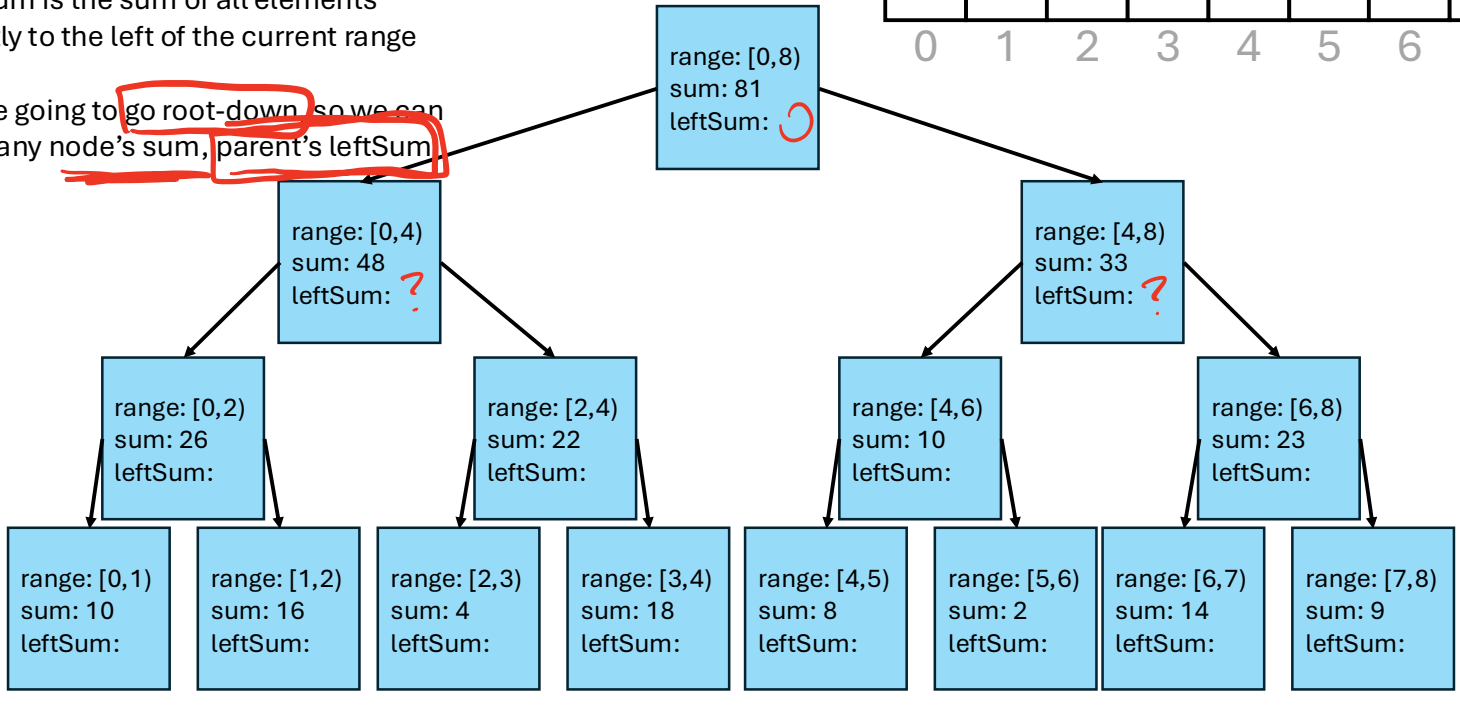


Step 2: fill in leftSum and Output

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

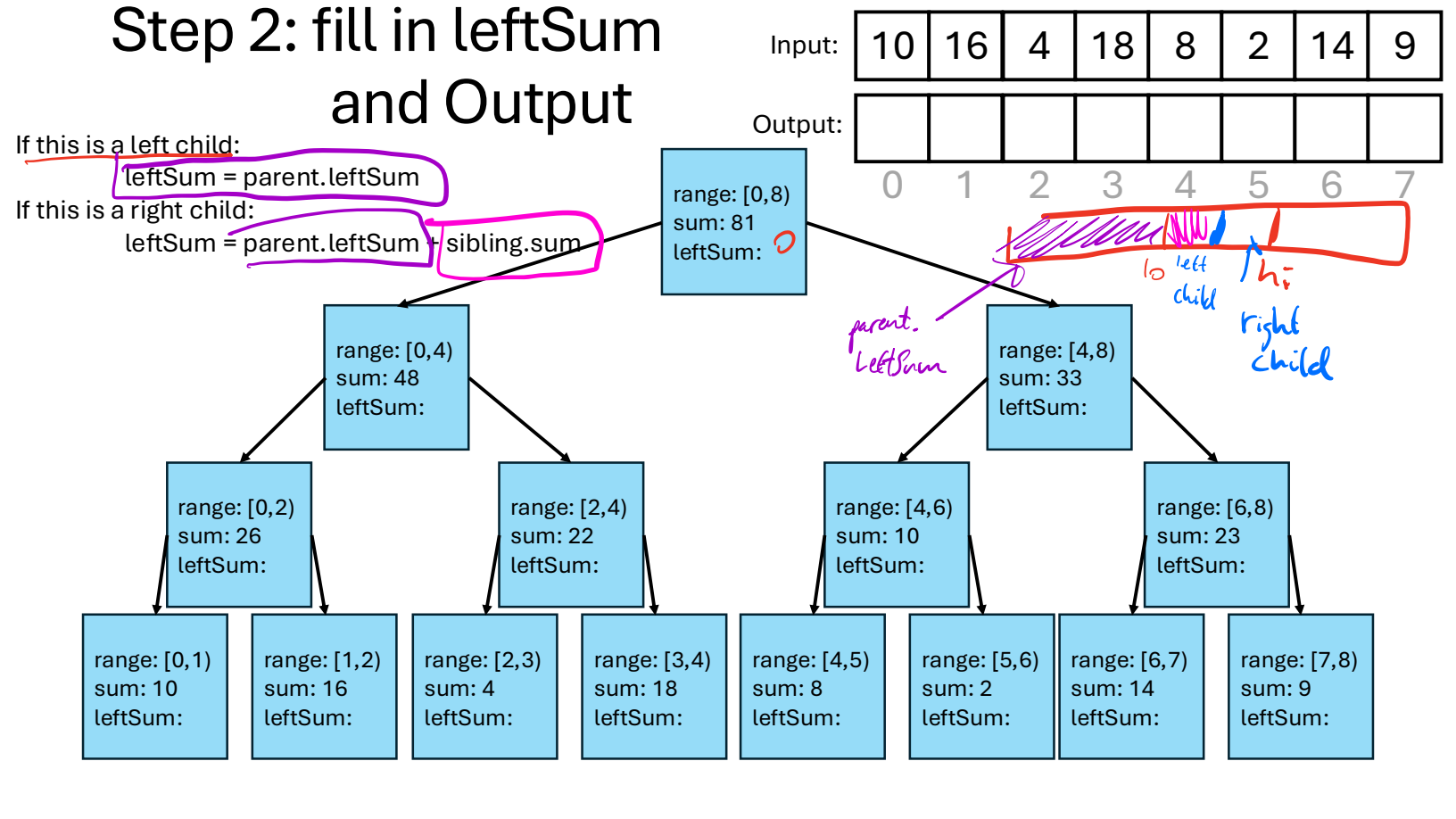
We're going to go root-down, so we can use: any node's sum, parent's leftSum





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

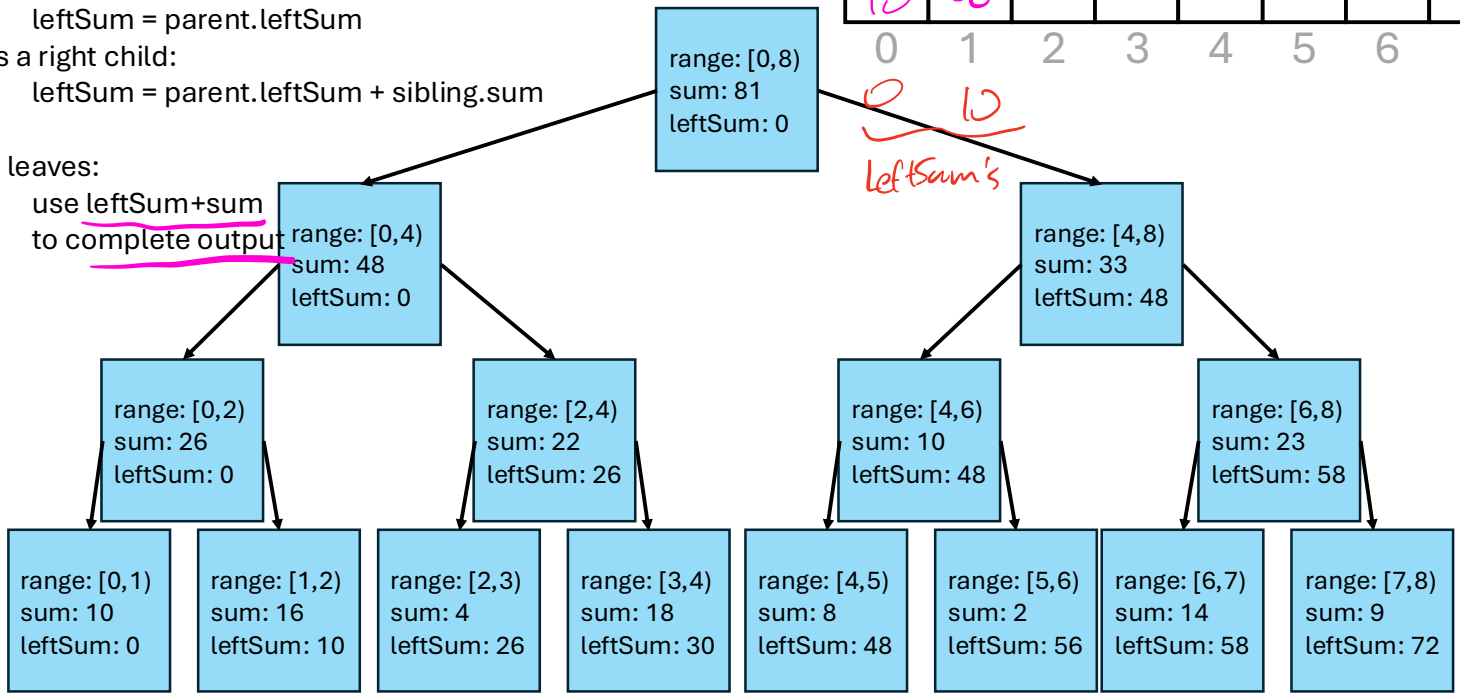


Step 2: fill in leftSum and Output

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						
	0	1	2	3	4	5	6	7

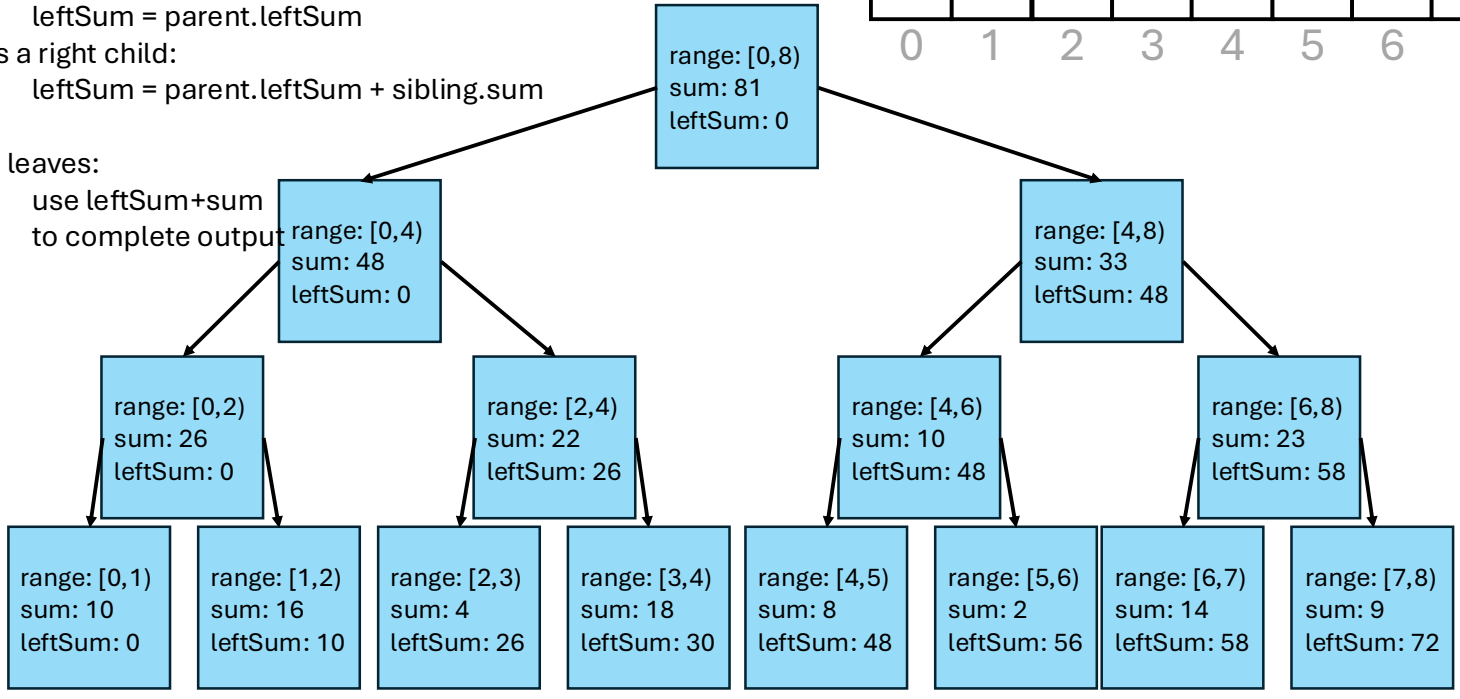


Step 2: fill in leftSum and Output

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

naive sequential alg.