Algorithm Design and Analysis

Range Query Data Structures

Roadmap for today

- Understand the ***range query*** problem
- Learn about the **SegTree™** data structure for range queries
- See how to apply range queries to ***speed up other algorithms***

The range query problem

Given: An array $a_0, a_1, \ldots, a_{n-1}$
Queries: Given an interval $[i, j)$, need to answer queries about $a_i, \ldots, a_{j-1}$

Example (Range sum queries): Given an array $a_0, \ldots, a_{n-1}$, need to answer queries for the sum of a range $[i, j)$, i.e,

$$\sum_{i \le k < j} a_k$$

Algorithms

Algorithm 1 (Just do it): Do no precomputation, given a query, just compute the sum by looping over the range.

Preprocessing time	Query time
0	$O(n)$

Algorithm 2 (Prefix sums): Compute *prefix sums* $p_j = \sum_{i<j} a_i$. A query $[i, j)$ is answered by returning $p_j - p_i$

Preprocessing time	Query time
$O(n)$	$O(1)$

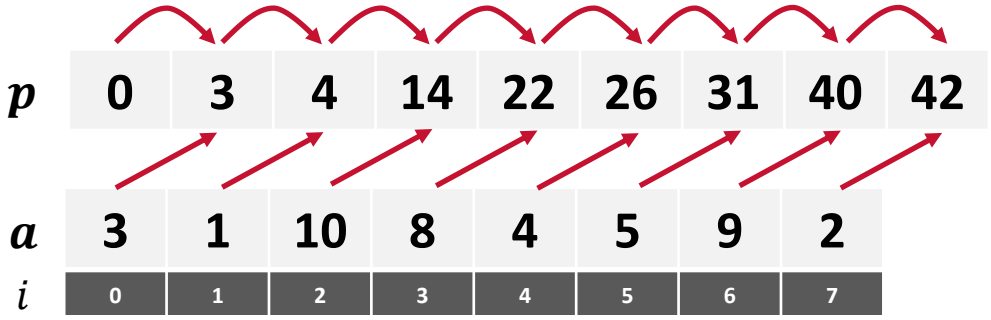
Let's make it more interesting

Updates: We also want to support update operations:

- **Assign**(i, x): Set $a_i \leftarrow x$
- **RangeSum**(i, j): Return $\sum_{i \le k < j} a_k$

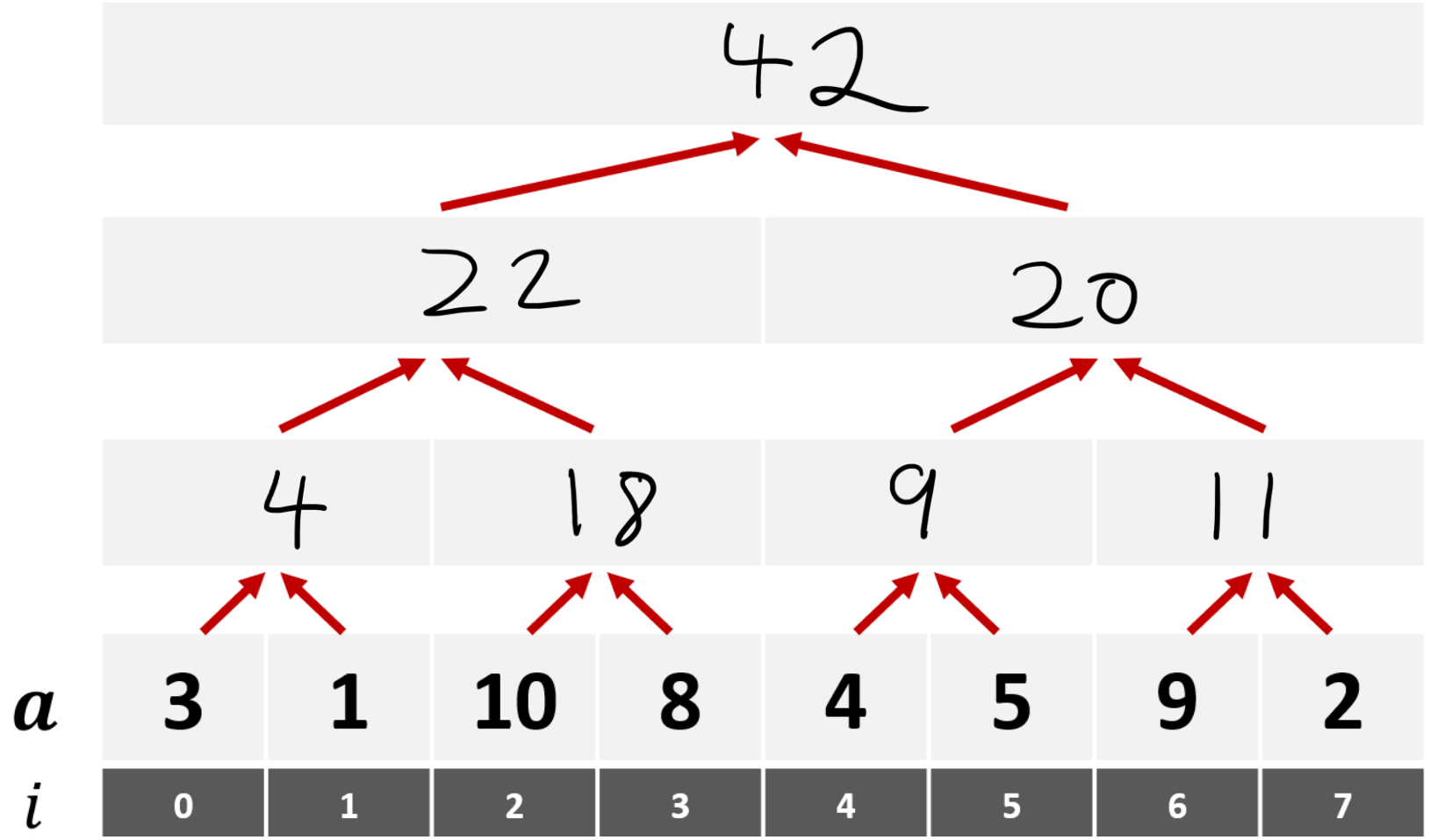
	Preprocessing time	Update time	Query time
Just do it	O	$O(1)$	$O(n)$
Prefix sums	$O(n)$	$O(n)$	$O(1)$

Why are the updates slow?



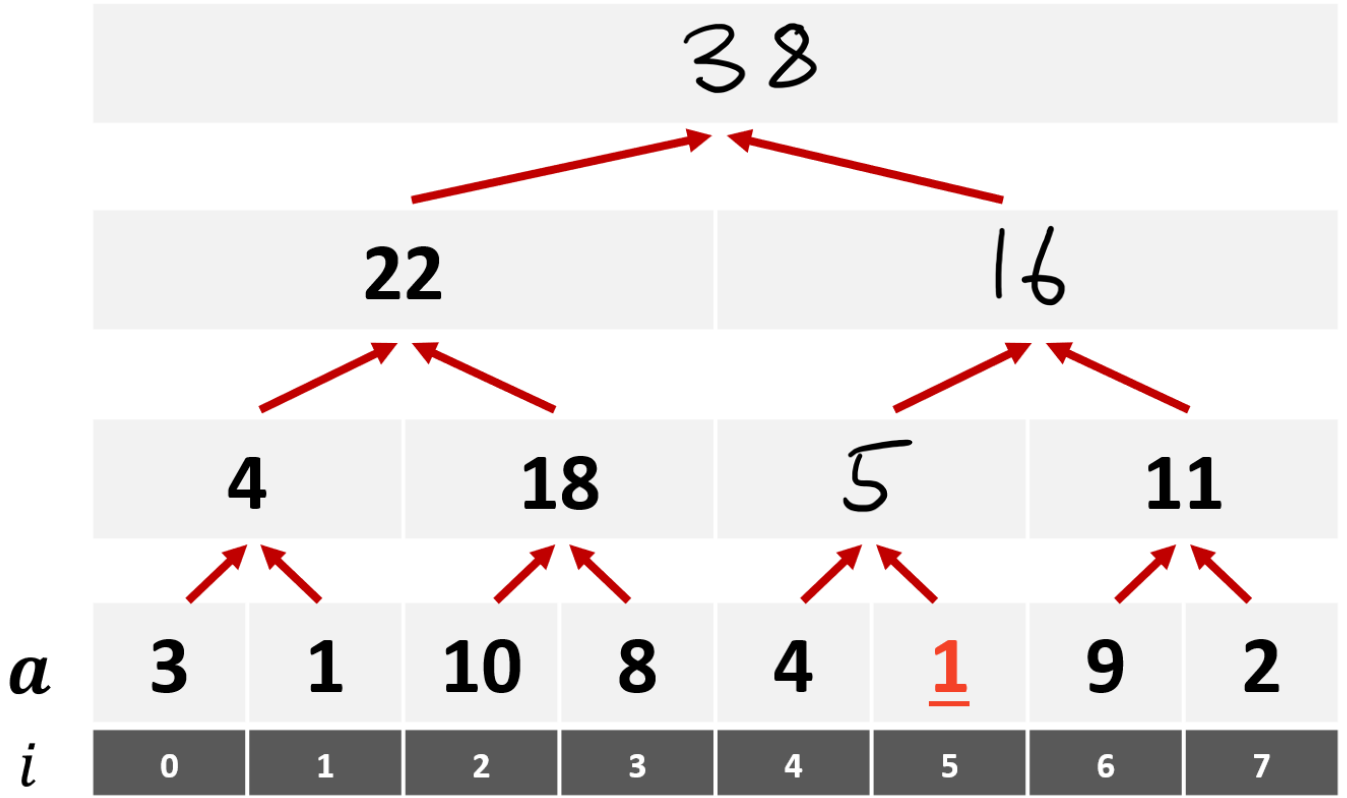
- Updating a single value might cause $O(n)$ ***dependent*** values to change
- **Big idea**: Can we compute the sums with ***fewer dependencies***
- If you've taken 15-210, this might remind you of something…

Divide-and-conquer summation

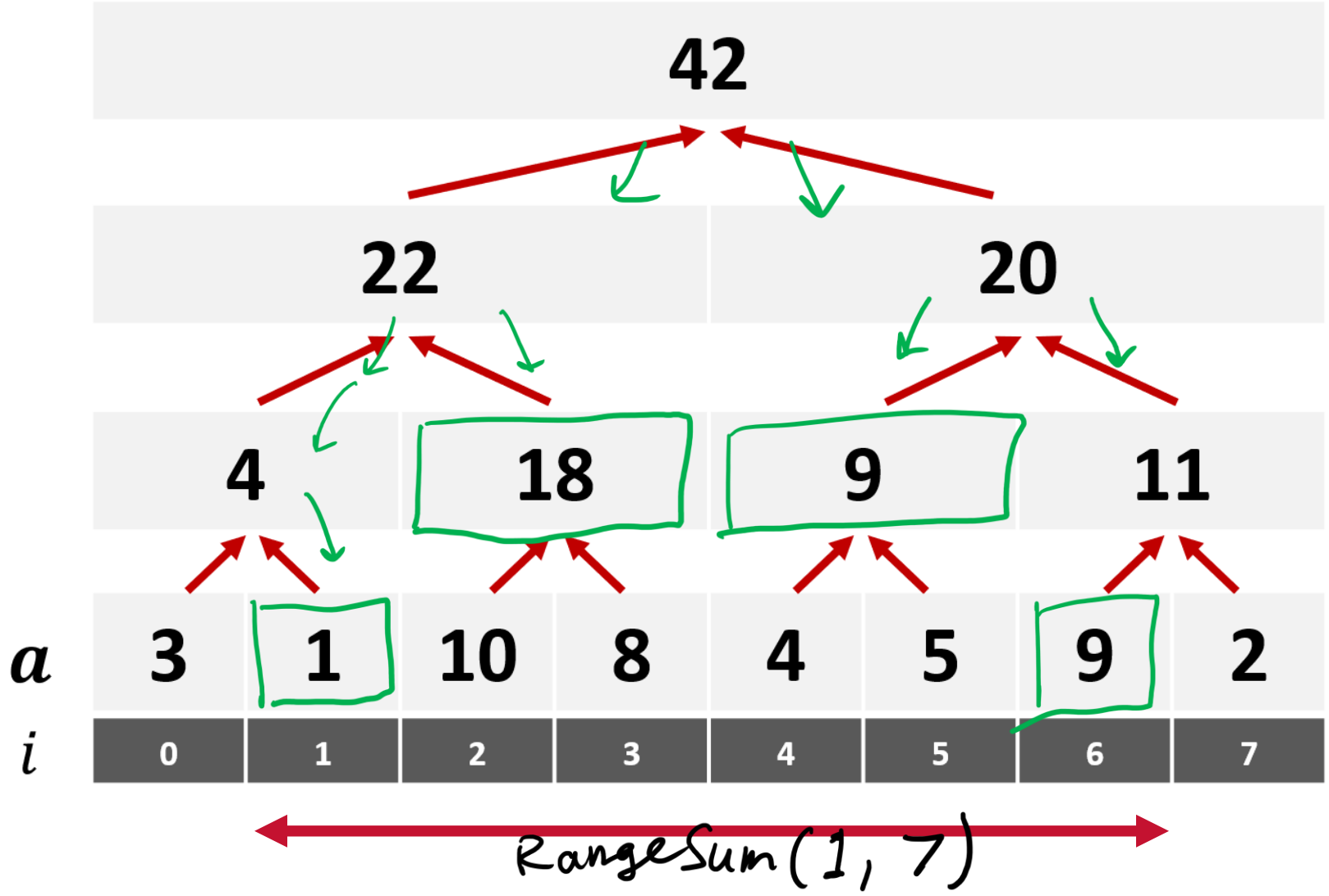


Analysis of Assign

Theorem: Assign(i, x) can be implemented in $O(\log n)$ time



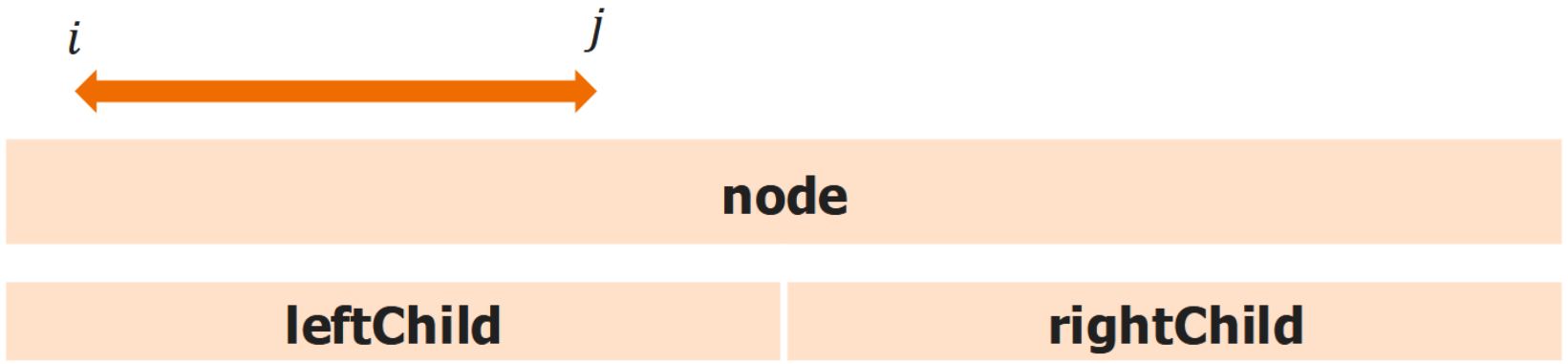
Queries (RangeSum)



Analysis of RangeSum

Theorem: RangeSum(i, j) can be implemented in $O(\log n)$

Case 1: The interval we're looking for is entirely contained in one half of the current node

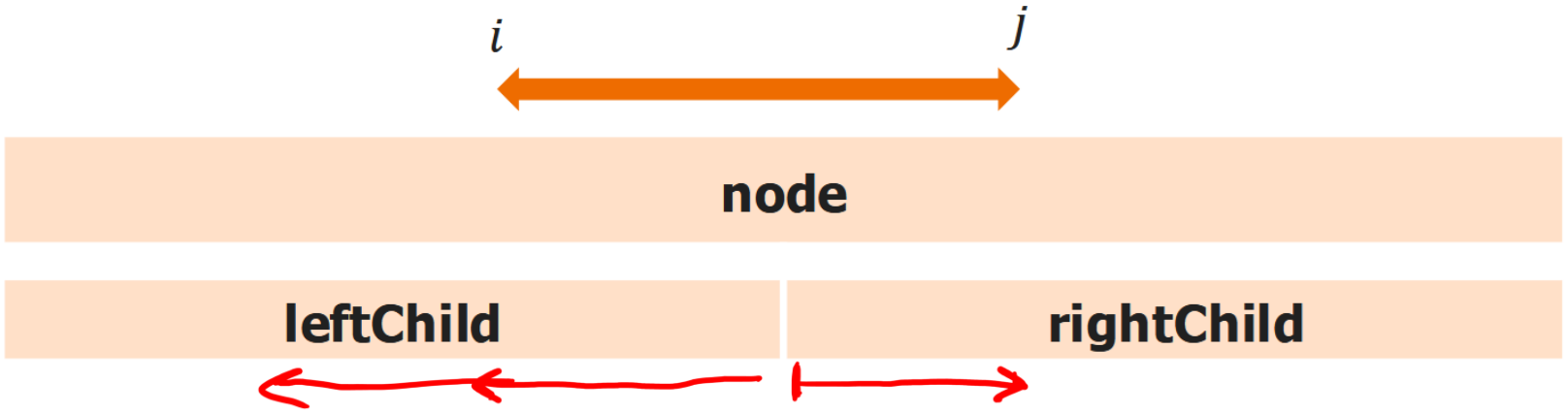


Only one recursive call!

Analysis of RangeSum

Theorem: RangeSum(i, j) can be implemented in $O(\log n)$

Case 2: The interval we're looking for is split across both halves



Two recursive calls…

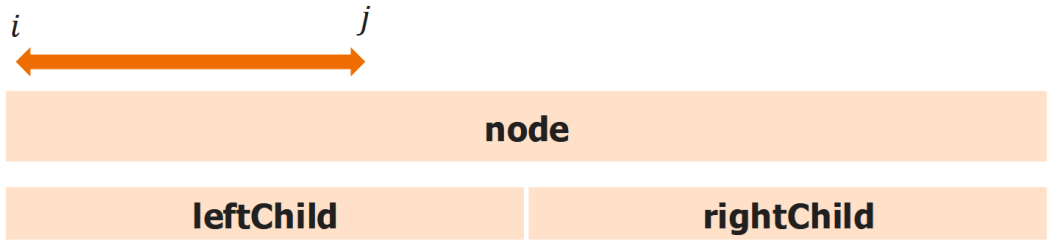
Claim: We only need two recursive calls once!

Analysis of RangeSum

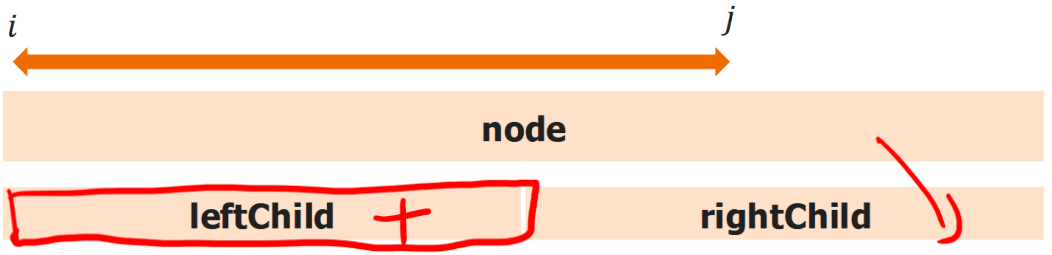
Theorem: RangeSum(i, j) can be implemented in $O(\log n)$

Claim: We only need two recursive calls once!

Each recursive call wants a prefix or suffix of the node



- The prefix is entirely in the left half
- Only one recursive call!

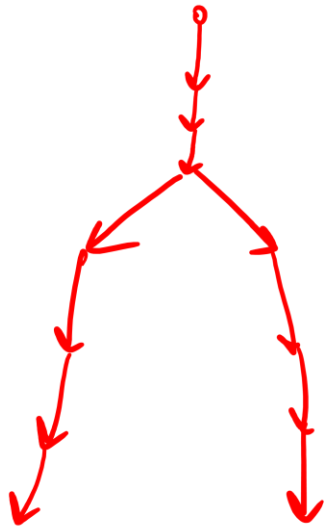


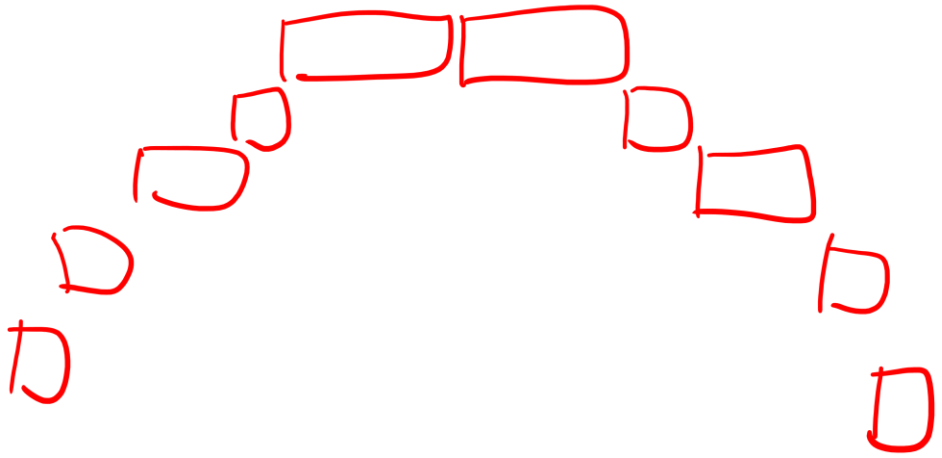
- The prefix stretches into the right half
- The sum is the entire left half + a recursive call on the right half
- Only one recursive call!

Analysis of RangeSum

Theorem: RangeSum(i, j) can be implemented in $O(\log n)$

Proof: We just argued that the recursion tree looks like…





Applications

Speeding up algorithms

1 2 5 3 4

Problem (Inversion count): Given a permutation $p_0, p_1, \ldots, p_{n-1}$ (of the integers $0 \ldots n-1$), an inversion is a pair p_i, p_j such that $\boldsymbol{i < j}$ but $\boldsymbol{p_i > p_j}$. The problem is to count the number of inversions in a sequence.

Slow Algorithm:

```
for i in [0, …, n-1]:
  for j in [i+1, …, n-1]:
    if P[i] > P[j]: count++
```

```
for i in [0, …, n-1]:
  count how many P[j]'s in the
  interval [i+1, n-1] are < P[i]
```

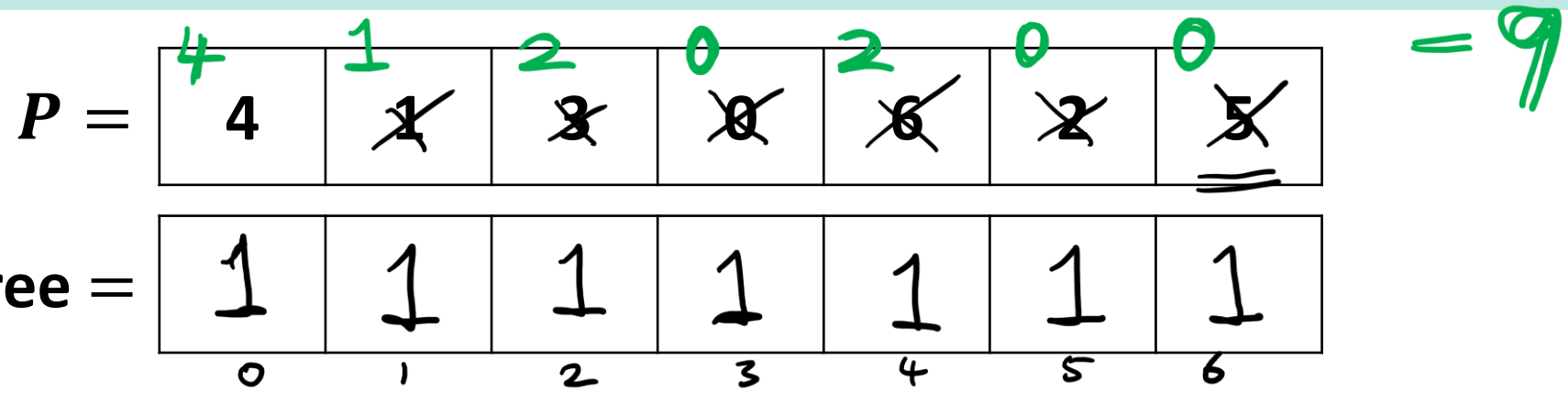
These sound like range queries???

Faster Inversion Count

- How can we use **SegTrees** to speed up our algorithm?

Idea: Make this into a range query:

count how many P[j]'s are < P[i]



*The SegTree will store **indicator variables**, 1 means we have seen that element before, 0 means we have not. RangeSum = count*

Faster Inversion Count: Code

```
function inversionCount(A : int list) {
    counts := SegTree([0] * len(A))
    invCount := 0
    for i in [n-1, n-2, ..., 0] {
        invCount += counts.RangeSum(0, A[i])
        counts.Assign(A[i], 1)
    }
    return invCount
}
```

$O(\log n)$

$O(\log n)$

$\longrightarrow O(n \log n)$

Implementation

Data structure implementation

- Remember binary heaps? Their structure is super useful!

Quick refresher:

- The root node is 0
- The left child of i is $2i + 1$
- The right child of i is $2i + 2$

0							
1				2			
3		4		5		6	
7	8	9	10	11	12	13	14

- Let's also use this for range queries – This will be called a **SegTree™**
- We will assume that n is a power of two for simplicity

Construction

```
class SegTree {
  nodes : Node list
  n : int }


class Node {
  val : int
  leftIdx : int
  rightIdx : int }

function left(u) {
  return 2*u+1; }


function right (u) {
  return 2*u+2; }
```

```
constructor (A : int list) {
  n = A.length
  nodes = [None] * (2*n - 1)

  # Fill in the leaves
  for i in [0, …, n-1]
    nodes[i+(n-1)] = Node(A[i], i, i+1)

  # Fill in the rest of the tree
  for i in [n-2, n-3, …, 0] {
    leftNode = nodes[left(i)]
    rightNode = nodes[right(i)]
    nodes[i] = Node(leftNode.val +
                    rightNode.val,
                    leftNode.leftIdx
                    rightNode.rightIdx)
}}
```

Assumes n is pow of 2

exclusive

Assign

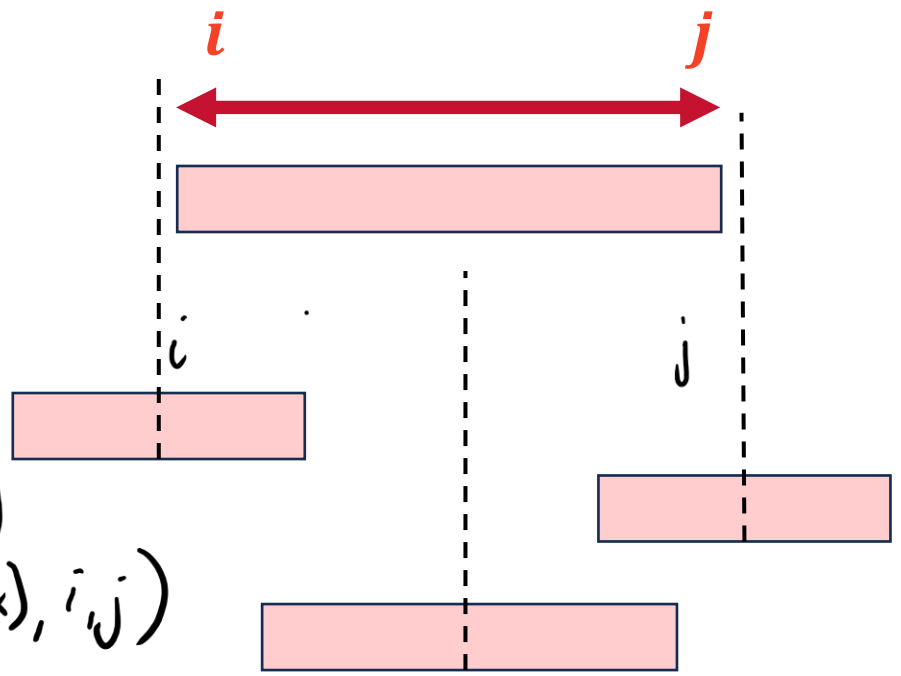
```
# Assign value x to position i
function assign(i, x) {
    nodeIdx = i + n - 1
    nodes[nodeIdx].val = x
    while nodeIdx > 0 {
        nodeIdx = parent(nodeIdx)
        node = nodes[nodeIdx]
        node.val = nodes[left(nodeIdx)].val
                 + nodes[right(nodeIdx)].val
    }
}
```

```
class SegTree {
  nodes : Node list
  n : int }

class Node {
  val : int
  leftIdx : int
  rightIdx : int }

function parent(u) {
  return (u - 1) // 2 }
```

Range Query

```
function sum(nodeIdx : int, i : int, j : int) {
  node = nodes[nodeIdx]
  if (i == node.leftIdx and node.rightIdx == j)
    return node.val
  else {
      mid := (L + R) / 2;
      if (i >= mid) return sum(right(nodeIdx), i, j)
      else if (j <= mid) return sum(left(nodeIdx), i, j)
      else {
          return sum(left(nodeIdx), i, mid)
                 + sum(right(nodeIdx), mid, j)
      }}}
```

(node.leftIdx + node.rightIdx)//2




```
function RangeSum(i, j) {
  return sum(0, i, j);
}
```

Extensions of SegTrees

Reducing over other associative operations!

- $+$ can be replaced with any binary associative operator, e.g., min, max
- Examples await you in recitation

- Just change the code in **`constructor`**, **`Assign`**, and **`sum`**
- Replace the + with your favourite associative operator!

```
return sum(left(nodeIdx), i, mid) +
       sum(right(nodeIdx), mid, j)
```

```
return max(sum(left(nodeIdx), i, mid),
           sum(right(nodeIdx), mid, j))
```

Summary

- **SegTrees** allow us to implement **dynamic range queries** in $O(\log n)$
  - Assign/Update in $O(\log n)$
  - RangeSum in $O(\log n)$
- SegTrees are useful for **speeding up algorithms** that require summing over a range of elements
  - E.g., speeding up inversion counting from $O(n^2)$ to $O(n \log n)$
- We can implement a SegTree with any associative operation
  - Plus/RangeSum, min/max, even fancier operations!
  - See recitation!