Algorithm Design and Analysis

Range Query Data Structures

Roadmap for Today

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

The Range Query Problem

A Motivating Example

- Let’s say I have some sensor data along a pipe
  - Different sensors may update their readings at different times
- I want to quickly be able to get information about the sensors in some range (eg. sum, max, min)

The Range Query Problem

Given: An array A
Queries: For an interval $[i, j)$, answer queries (e.g. sum, min, max) on that interval

This lecture: We focus on **range sums**

- Given an interval, $[i, j)$ return the sum of that interval, i.e.,

$$\sum_{i \le k < j} A[k]$$

Our Range Query Data Structure

Algorithm Brainstorming

What's the simplest algorithm you can think of?

Just do it

Loop through the range & add them up

How could we speed this up with pre-processing?

Prefix sums P[i] = prefix sum upto + not including i

sum([i, j)) = P[j] - P[i]

Algorithms

Algorithm 1 (Just do it): Loop over the range and compute the sum

Preprocessing Time	Query Time
O(1)	O(n)

Algorithm 2 (Prefix Sums): Precompute prefix sums P. A query on interval $[i, j)$ returns $P[j] - P[i]$.

Preprocessing Time	Query Time
O(n)	O(1)

Supporting Updates

Now we have 2 operations:

- **`RangeSum(i,j)`**: returns the sum of the range $[i, j)$
- **`Assign(i,x)`**: sets $A[i] = x$

Algorithm	Preprocessing Time	Query Time	Update Time
Just do it	O(1)	O(n)	O(1)
Prefix Sums	O(n)	O(1)	O(n)
Goal	O(n)	O(log n)	O(log n)

The Problem with Prefix Sums

i	0	1	2	3	4	5	6	7
A =	**3**	**1**	**10**	**8**	**4**	**5**	**9**	**2**

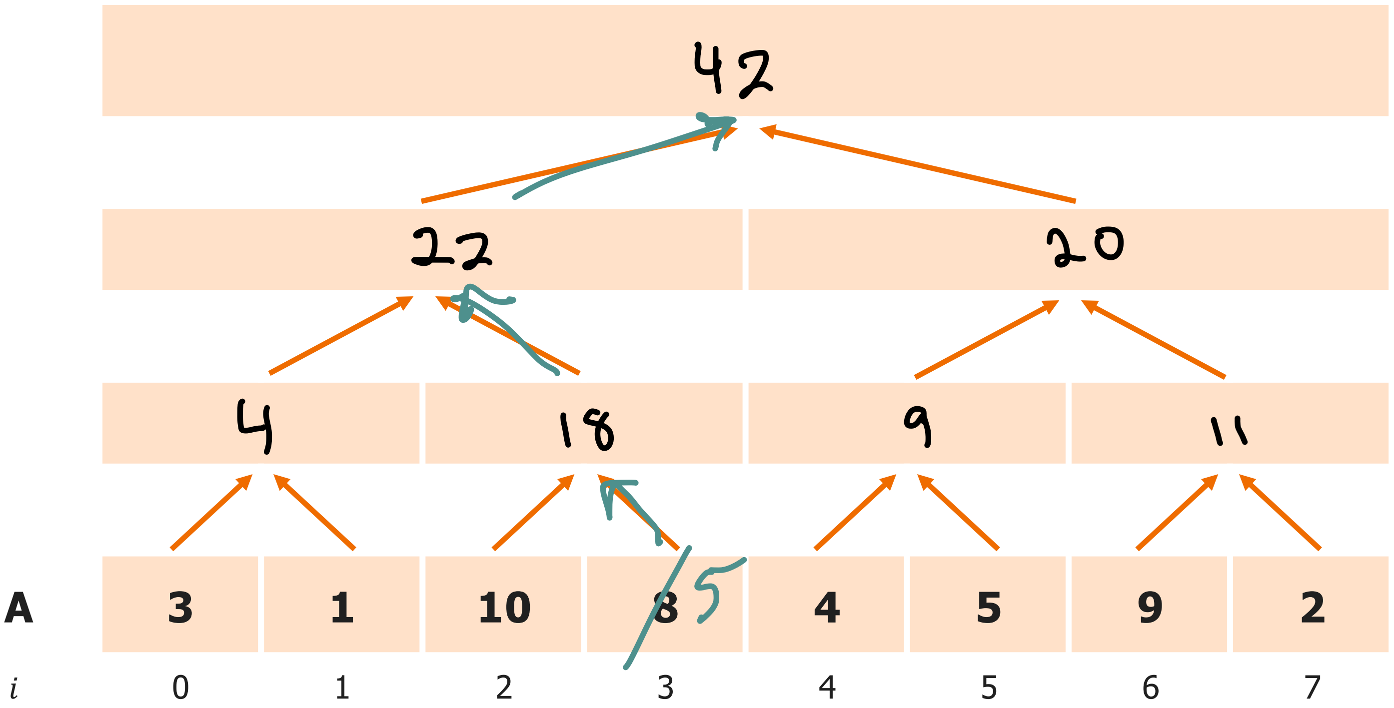
Prefixes =	**0**	**3**	**4**	**14**	**22**	**26**	**31**	**40**	**42**

Problem: If we update the first value in the list, we have to update $O(n)$ prefix sums

Big Idea: We want **fewer dependencies**

- This may remind us of parallel algorithms
- Is there a way to compute sums with less dependencies?

Divide and Conquer Summation



Goals

Show `construct(A)` is $O(N)$

$O(n)$ nodes $(2n-1)$

adding is $O(1)$

overall: $O(n)$

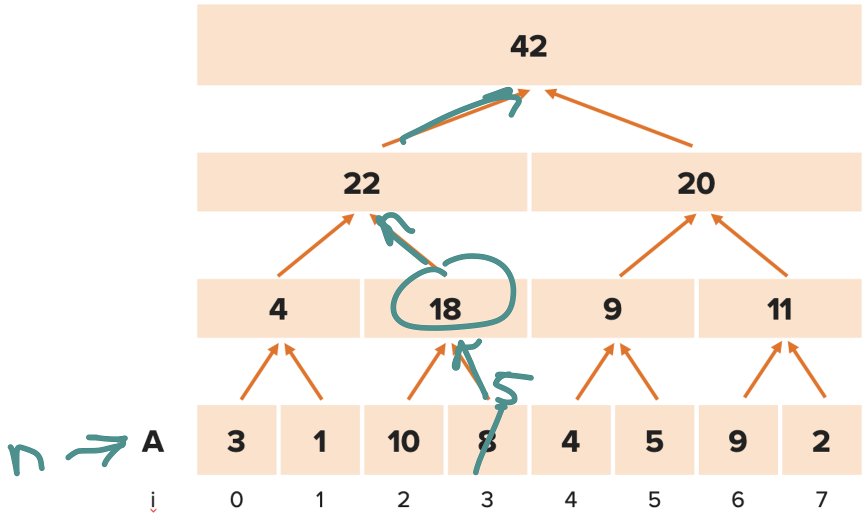
Show `Assign(i, x)` is $O(logN)$

$O(\log n)$ nodes to update

$O(1)$ to recalculate the val

$O(\log n)$ in total

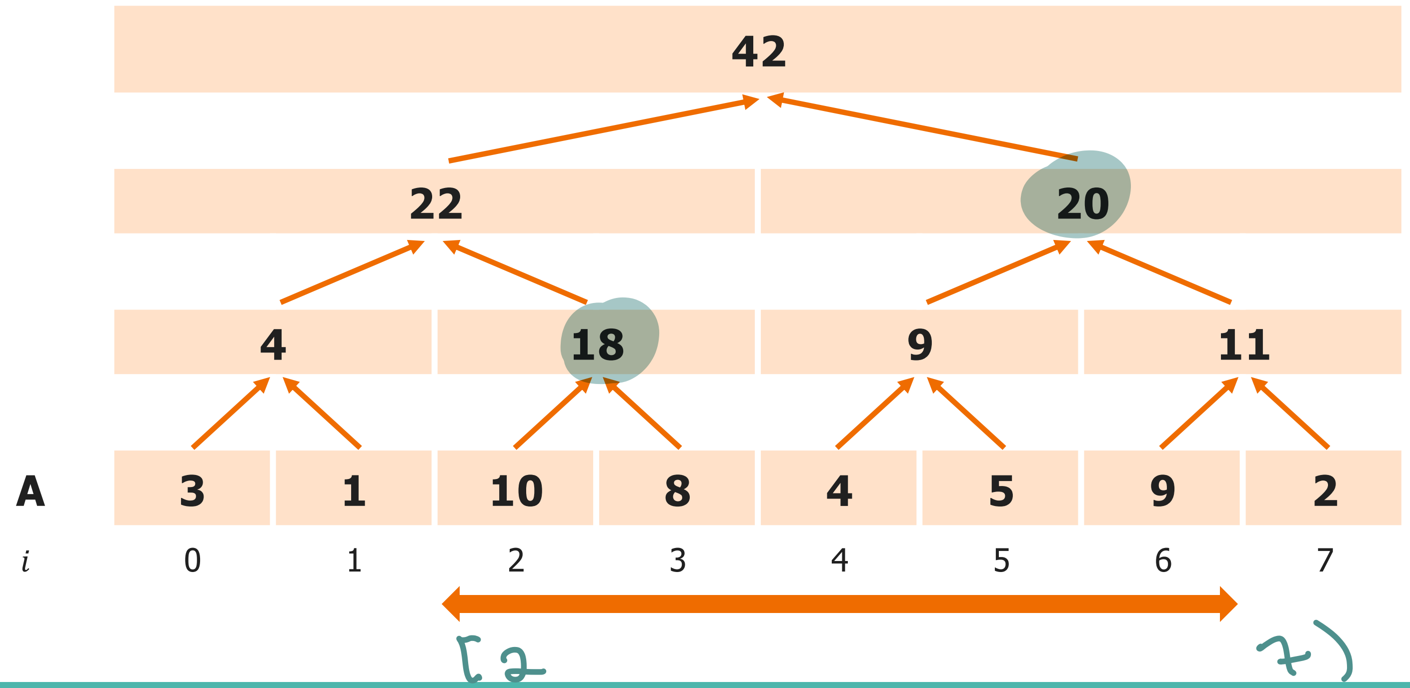
Show `RangeSum(i, j)` is $O(logN)$



Building Intuition

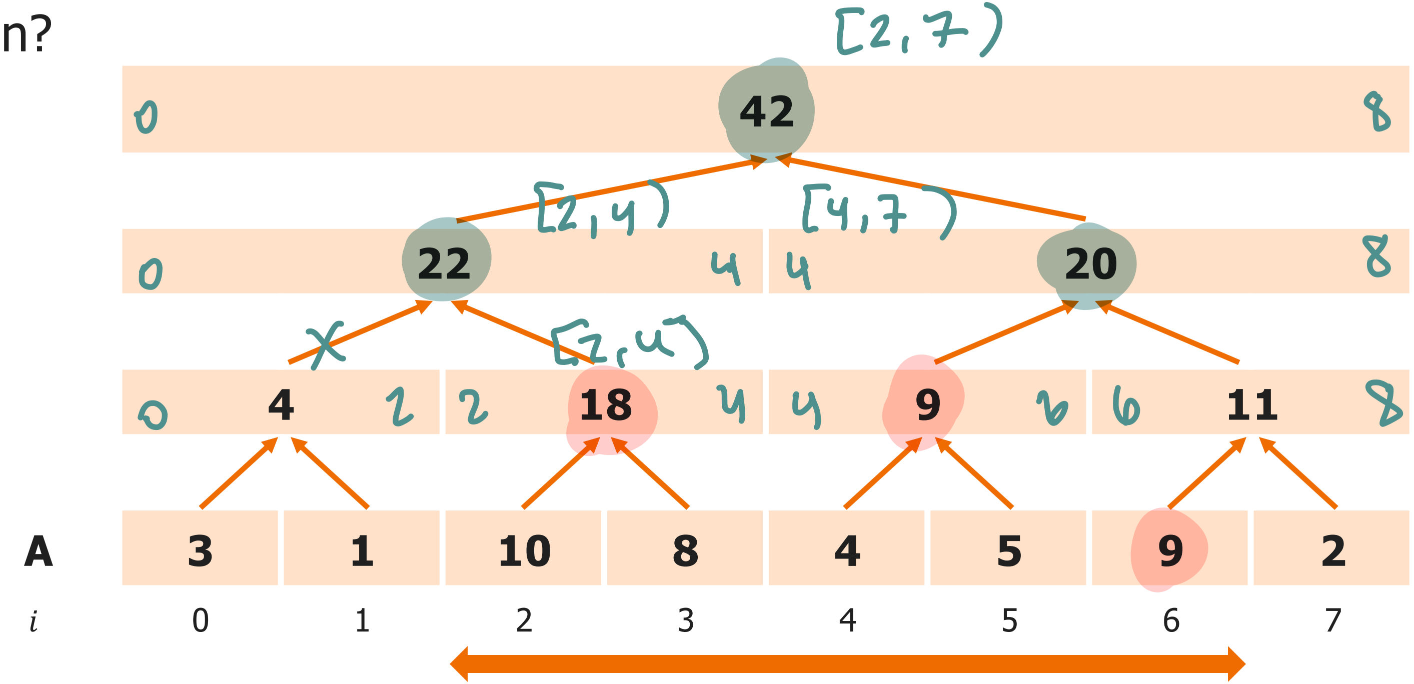
[2,7)

How would we go about finding the sum of this interval?



Our algorithm

Let's start at the top of the tree. What sums are we looking for in the children?

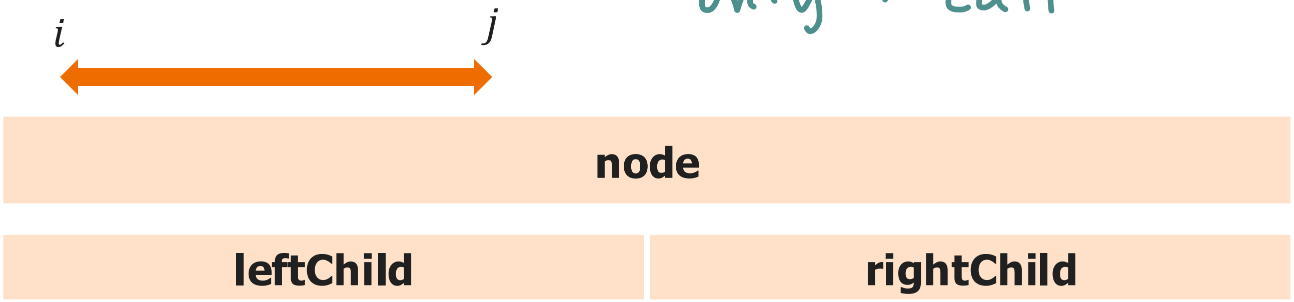


Proof: RangeSum is $O(\log(n))$

We start at the root, looking for the sum of $[i, j)$. We recursively look for the sum from the left/right children. We want $O(\log(n))$ recursive calls.

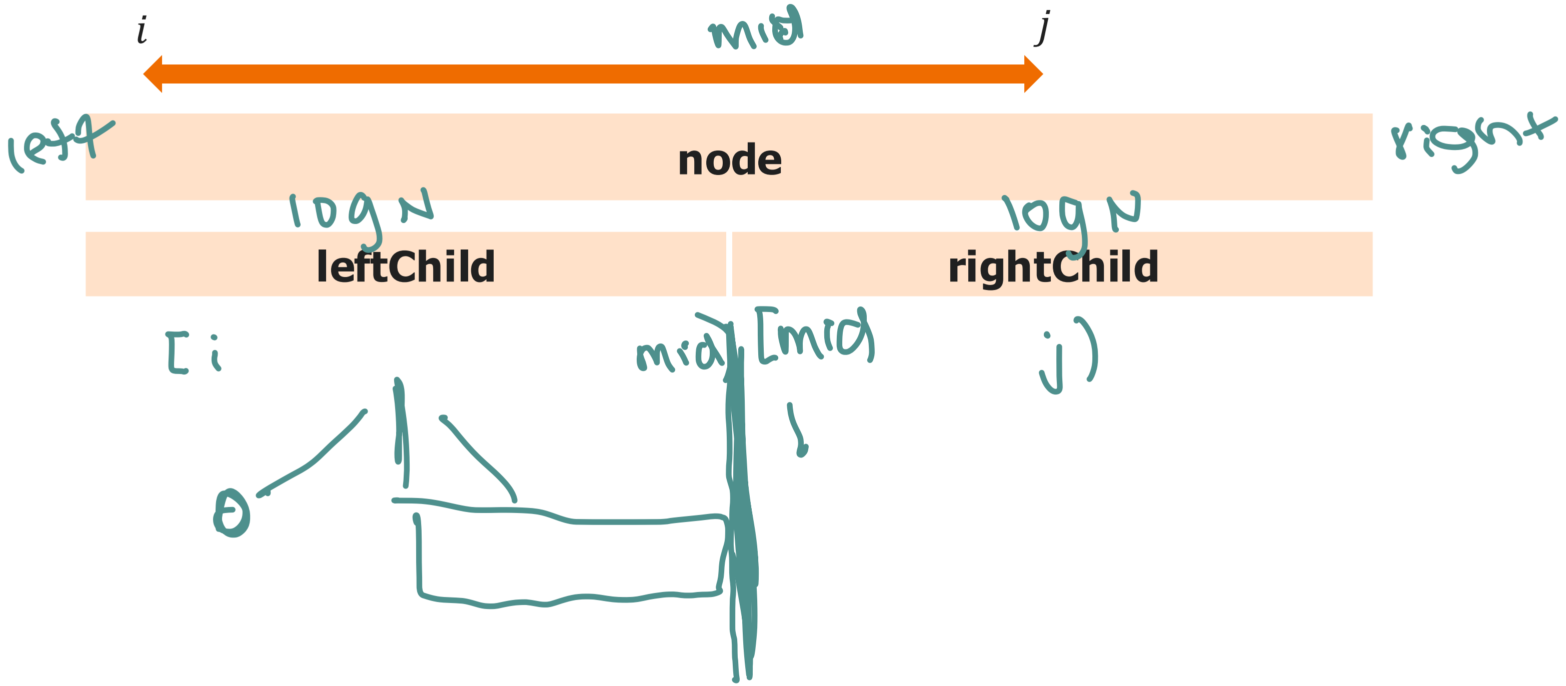
Case 1: The interval we're looking for is entirely contained in one half of the current node

only 1 call



Proof: RangeSum is $O(\log(n))$

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

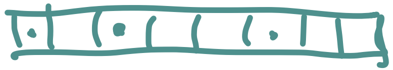


Applications

The Interface

- `Construct(A):` Takes an array `A`, and returns a segTree of `A`
  - $O(N)$
- `RangeSum(i,j):` Returns the sum of the elements in the interval $[i, j)$
  - $O(logN)$
- `Assign(i,x):` Sets `A[i] = x`
  - $O(logN)$

When using SegTrees, treat them as arrays. We don't care about the implementation!



Back to the Motivating Example

We have sensor data in a line and want the sum of sensor readings in a range

- Simply store the sensor data in a SegTree!
- To get the sum of readings in $[i, j)$, call `RangeSum(i, j)`

So should we always use SegTrees when we want queries in a range?

- What if we had stock data that updated with the current price every hour?
  - We want to answer queries about the sum of prices in different time intervals (we'll use that to get average prices)

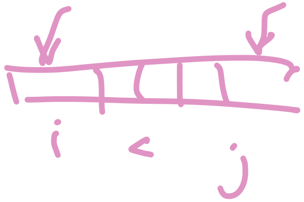
Warmup Problem

- **Given:** An array A of length n containing integers in $\{0, \ldots, 2n-1\}$
- Support **updates** of an element in A in $O(logN)$ time
- Answer **queries** in $O(logN)$ to return the number of values in A in the range ~~$[i, j)$~~ $[v_i, v_j)$

i	0	1	2	3
A =	**5**	**2**	**3**	**2**

query([2, 4)) $\rightarrow$ 3

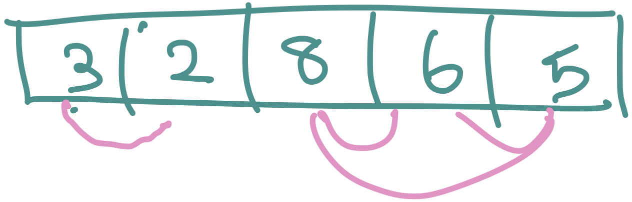
segTree =	0	0	2	1	0	1	0	0
	0	1	2	3	4	5	6	7



Speeding up Algorithms

Problem (Inversion Count):
- **Given:** An array A of length n containing integers in $\{0, \dots, 2n-1\}$
- **Return:** The number of inversions in A
  - An inversion is a pair of elements such that $i < j$ but $A[i] > A[j]$

Naive algorithm: $O(n^2)$




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

inversions = 4

Faster Inversion Count

How can we use **segTrees** to speed up our algorithm?

Where are we querying a **range**?

for i in [0, ..., n-1)
count # values less than A[i] that occur after i

1 0 2 1

A =

3	2	8	6	5

segTree =

	0	0	1	0	0	1	1	0	1	0
i	0	1	2	3	4	5	6	7	8	9

4 inversions

Faster Inversion Count: Pseudocode

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

RangeSum(A[i], A[i]+1)

Runtime:

O(n log n)

SegTree Implementation

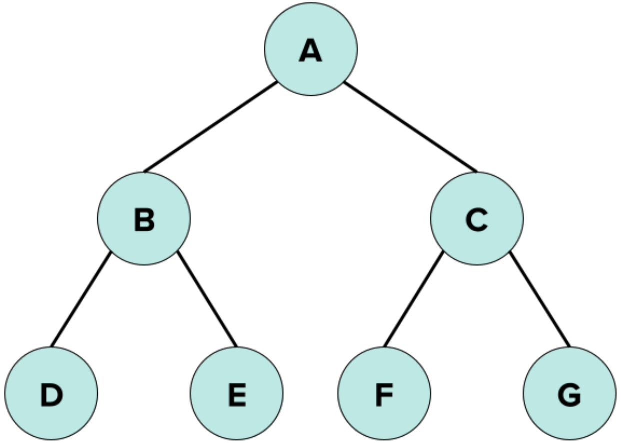
SegTree Implementation

Recall: Binary heaps

- Root is at index 0
- Left child of i is at index $2i + 1$
- Right child of i is at index $2i + 2$

$2*2+1$
$= 5$

$2*2+2$
$= 6$



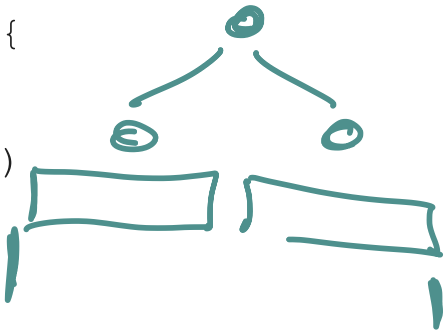
For simplicity: Assume n is a power of 2

heap =	A	B	C	D	E	F	G	
i	0	1	2	3	4	5	6	

Implementation: Construct

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

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

fun lChild(nodeIdx : int)
{ return 2*nodeIdx +1 }

fun rChild(nodeIdx : int)
{ return 2*nodeIdx + 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 from bottom to top
  for i in [n-2, n-3, …, 0] {
      leftNode = nodes[lchild(i)]
      rightNode = nodes[rchild(i)]
      nodes[i] = Node(leftNode.val + rightNode.val,
                 leftNode.leftIdx,
                 rightNode.rightIdx
}}
```

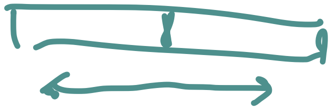


Implementation: Assign

```
fun assign(i, x) {
  nodeIdx = i + n - 1
  nodes[nodeIdx].val = x
  while nodeIdx > 0 {
      nodeIdx = parent(nodeIdx)
      node = nodes[nodeIdx]
      leftN = nodes[lchild(nodeIdx)]
      rightN = nodes[rchild(nodeIdx)]
      node.val = (leftN.val + rightN.val)
  }
}
```

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

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

fun parent(nodeIdx) {
  return (nodeIdx - 1) // 2 }
```

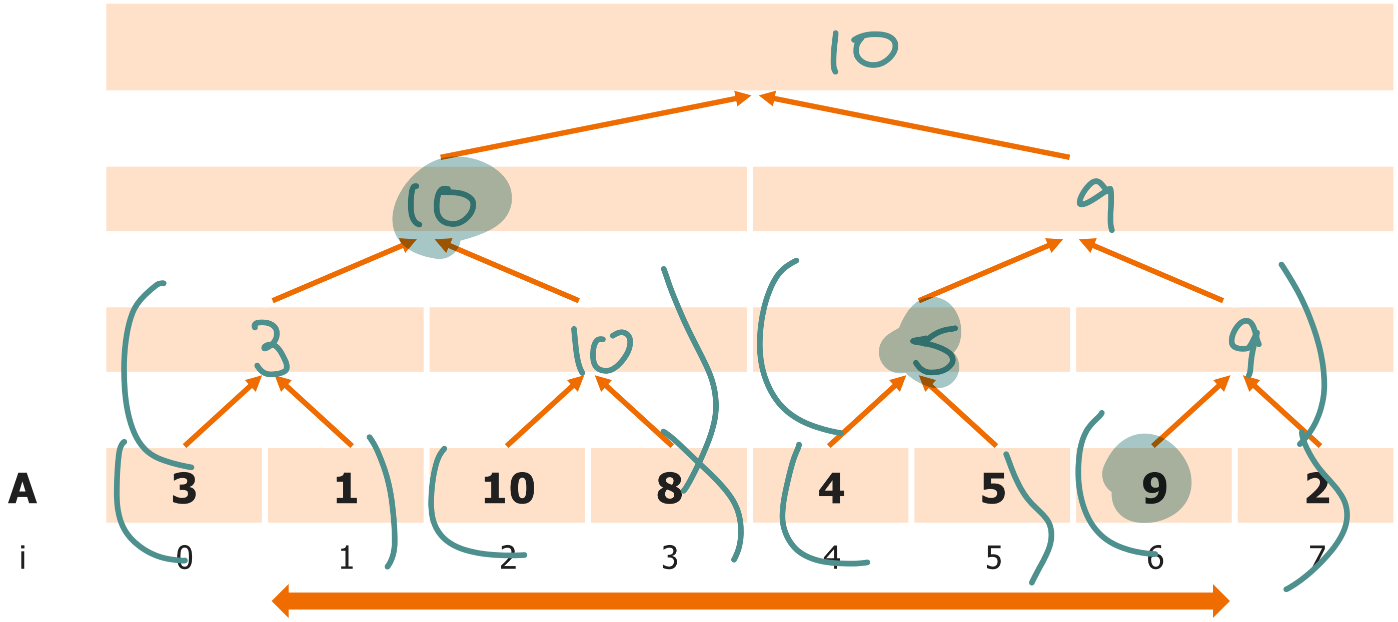
Implementation: RangeSum



```
fun sum(nodeIdx : int, i : int, j : int) {
  node = nodes[nodeIdx]
  if (i == node.leftIdx and node.rightIdx == j) { return node.val }
  else {
    mid = (node.leftIdx + node.rightIdx) / 2
    if (i >= mid) { return sum(rchild(nodeIdx), i, j) }
    else if (j <= mid) { return sum(lchild(nodeIdx), i, j) }
    else {
      return sum(lchild(nodeIdx), i, mid) +
             sum(rchild(nodeIdx), mid, j)
}}}
```

```
fun rangeSum(i, j) {
  return sum(0, i, j)
}
```

Did we have to sum?



Take-Home Messages

- SegTrees are useful for **speeding up algos** that involve queries over a range of elements
  - Remember that you may store information in a different order in the segTree than in your input array
  - E.g., speeding up inversion counting from $O(n^2)$ to $O(nlogn)$

- When you use SegTrees in your own algos, **don't think about them as trees**! They're basically fancy lists.

- We can implement a SegTree with any **associative operation** (even a custom one!)
  - You will see an example of this in recitation