Algorithm Design and Analysis

Range Query Data Structures

Reminders

- **Midterm One is next week!! Tuesday at 7:00pm.**
- This week's homework is an **oral homework.** Make sure your team has signed up for a time slot.

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 $a_0, a_1, \dots, a_{n-1}$
- Given a range $[i, j)$, need to answer queries about $a_i, \dots, a_{j-1}$

Example (Range sum queries): Given an array $a_0, \dots, a_{n-1}$, 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
$O(n)$	$O(1)$

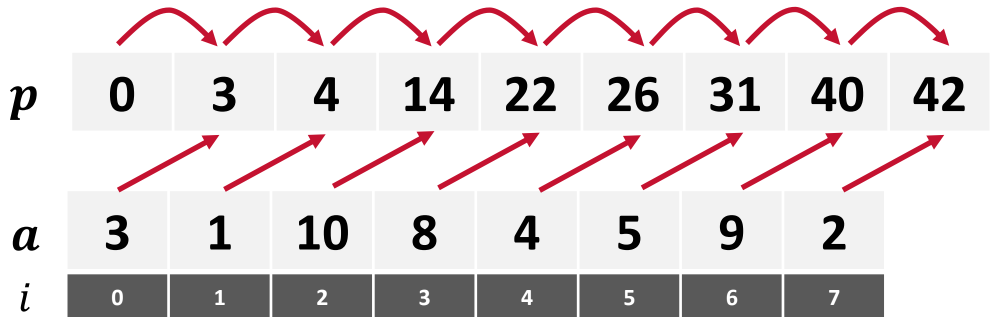
Let's make it more interesting

Updates: We also want to support update operations:

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

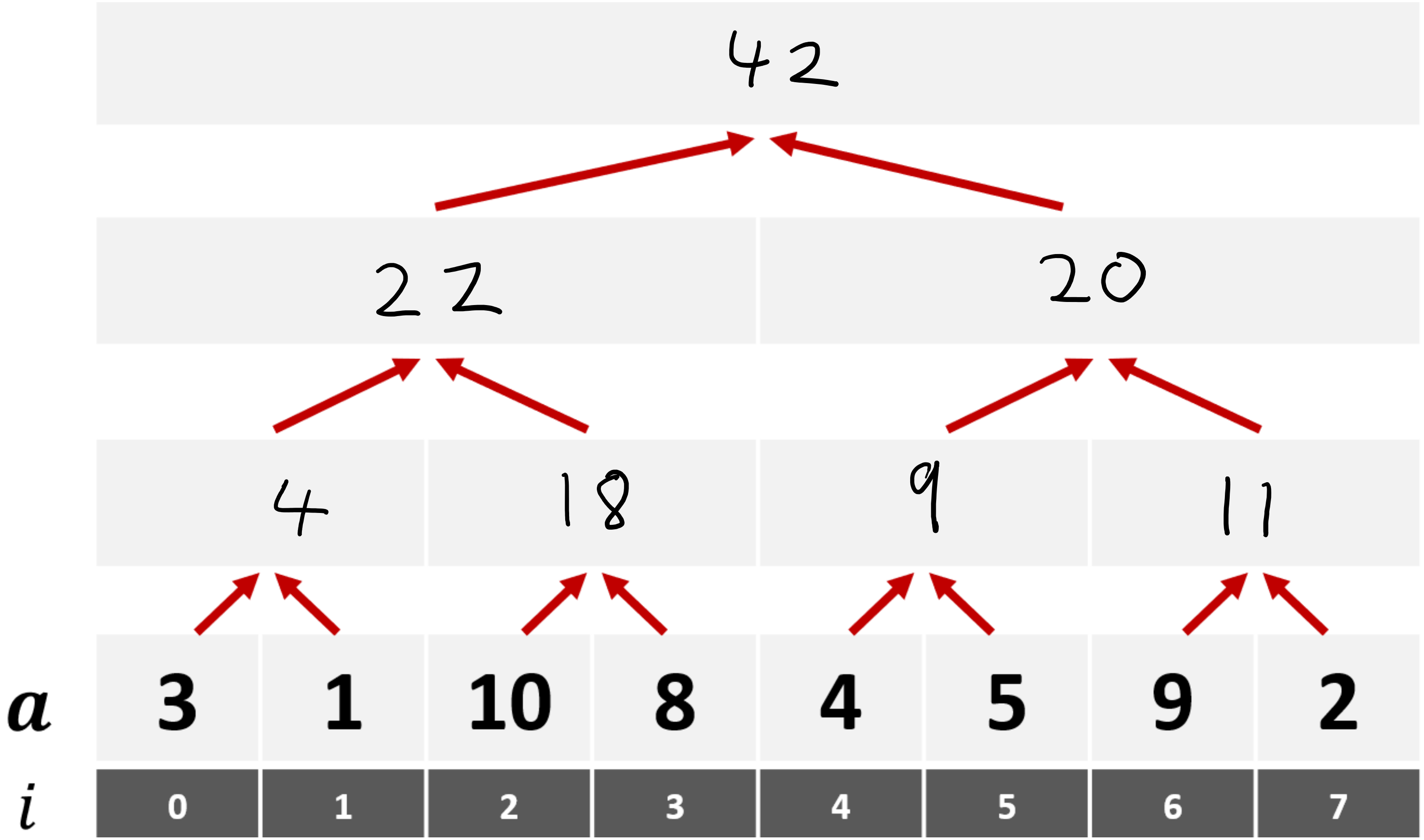
	Preprocessing time	Update time	Query time
Just do it	0	O(1)	O(n)
Prefix sums	O(n)	O(n)	O(1)

Why are the updates slow?



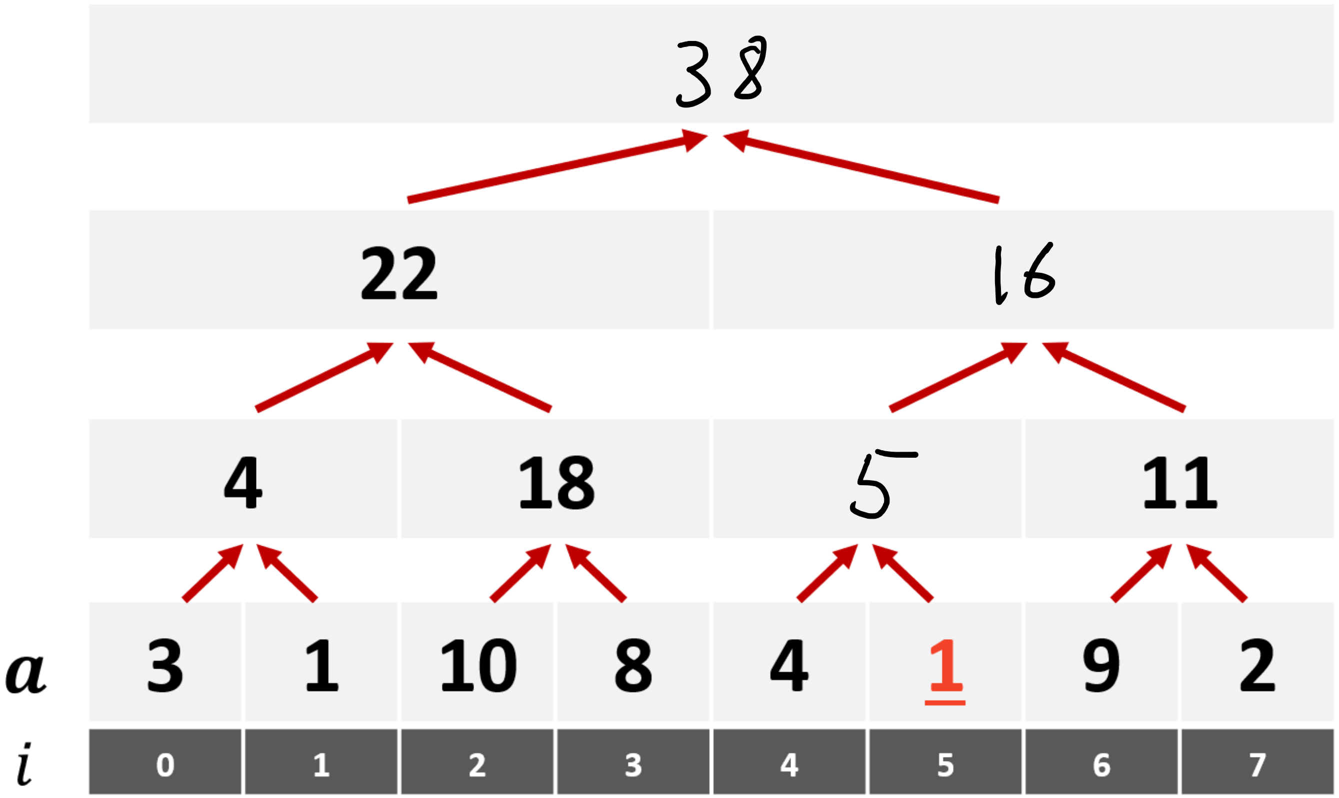
- Updating a single value might cause $O(n)$ ***dependent*** values to change
- **Big idea**: Can we compute the sums with ***fewer dependencies***
- If you've taken 15-210, this might remind you of something…

Divide-and-conquer summation

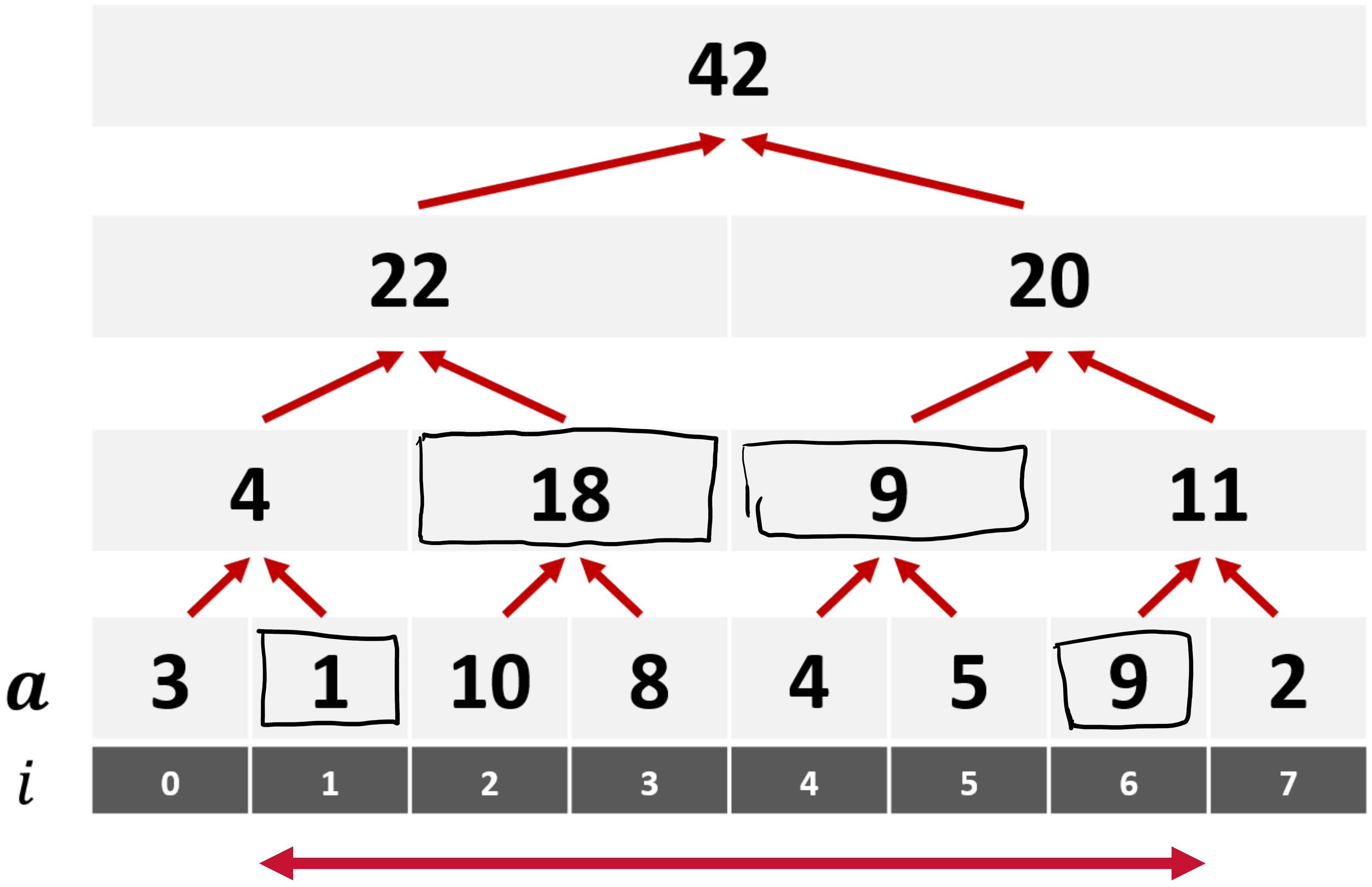


Updating a value (Assign)

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



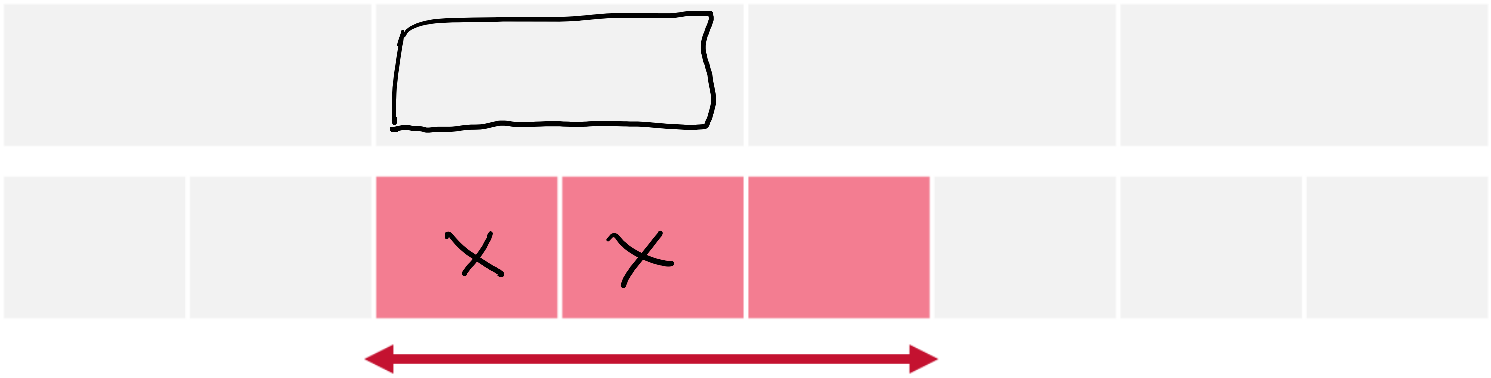
Queries (RangeSum)



Analysis

Lemma: Any interval $[i, j)$ can be broken into a set of intervals/blocks from the tree such that we use at most two intervals per level

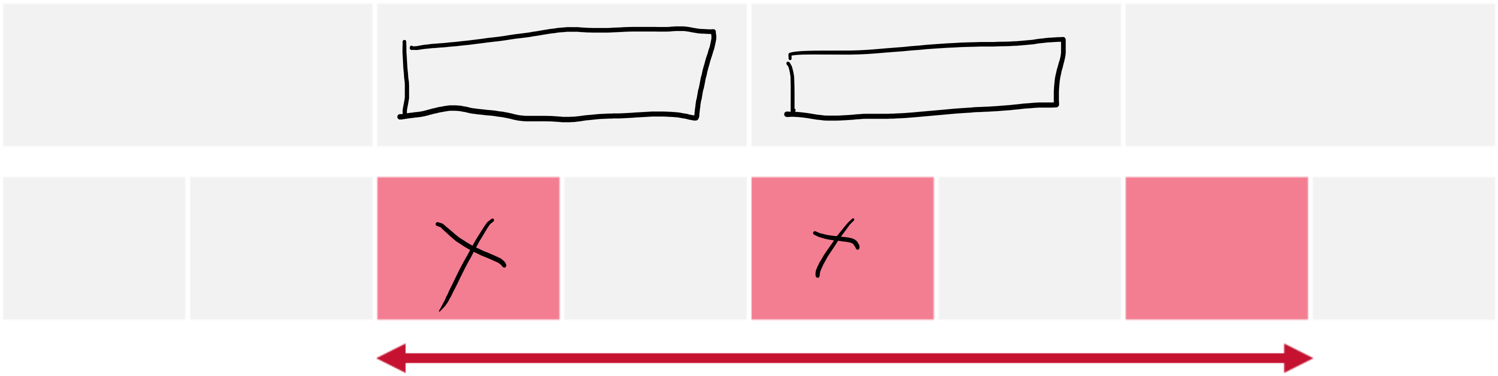
Case 1: Suppose there are three adjacent blocks on a level



Analysis

Lemma: Any interval $[i, j)$ can be broken into a set of intervals/blocks from the tree such that we use at most two intervals per level

Case 2: Three non-adjacent blocks. The leftmost block is a left child.

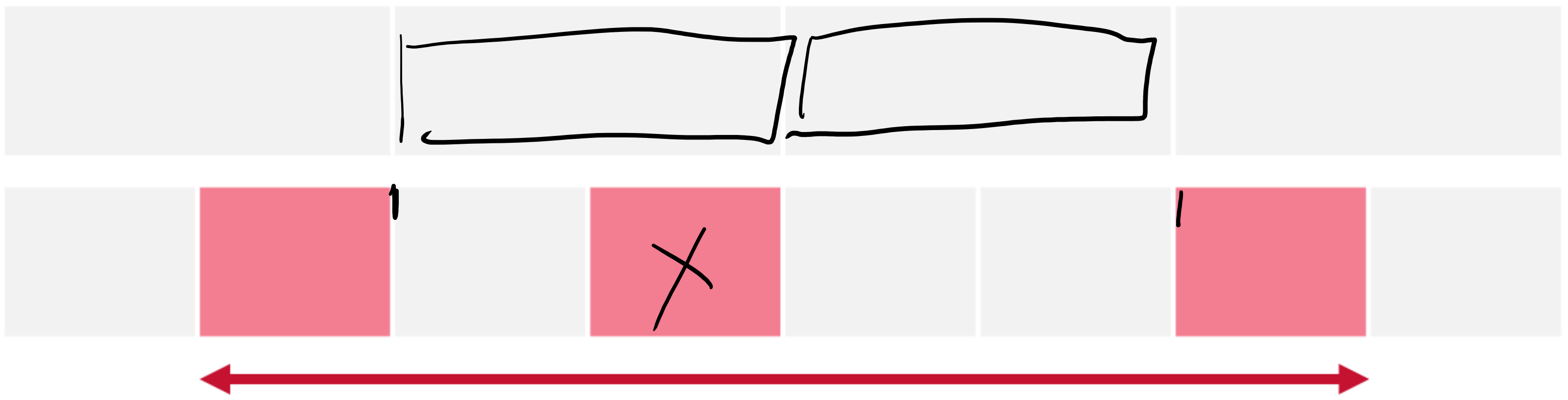


(Same in the symmetric case – the rightmost block is a right child)

Analysis

Lemma: Any interval $[i, j)$ can be broken into a set of intervals/blocks from the tree such that we use at most two intervals per level

Case 3: Three non-adjacent blocks. The leftmost block is a right child.



(Same in the symmetric case – the rightmost block is a left child)

Analysis

Corollary: Any interval $[i, j)$ can be broken into at most $2 \log n$ blocks

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

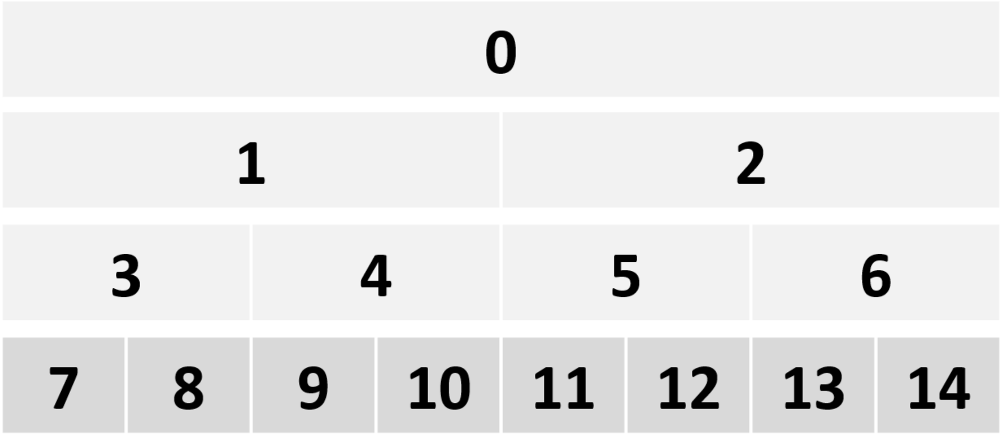
- *Proof*: We will implement it ☺

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 $2i + 2$



- 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

Building

```
struct SegTree { int n;  list<int> A; };

// Create a SegTree with values from a
SegTree::SegTree(list<int> a) {
    n := size(a);
    A := list<int>(2*n)
    A[n…2n] = a;     ←
    build(0, 0, n);  ←
}

int LeftChild(u) { return 2*u+1; }
int RightChild(u) { return 2*u+2; }
```

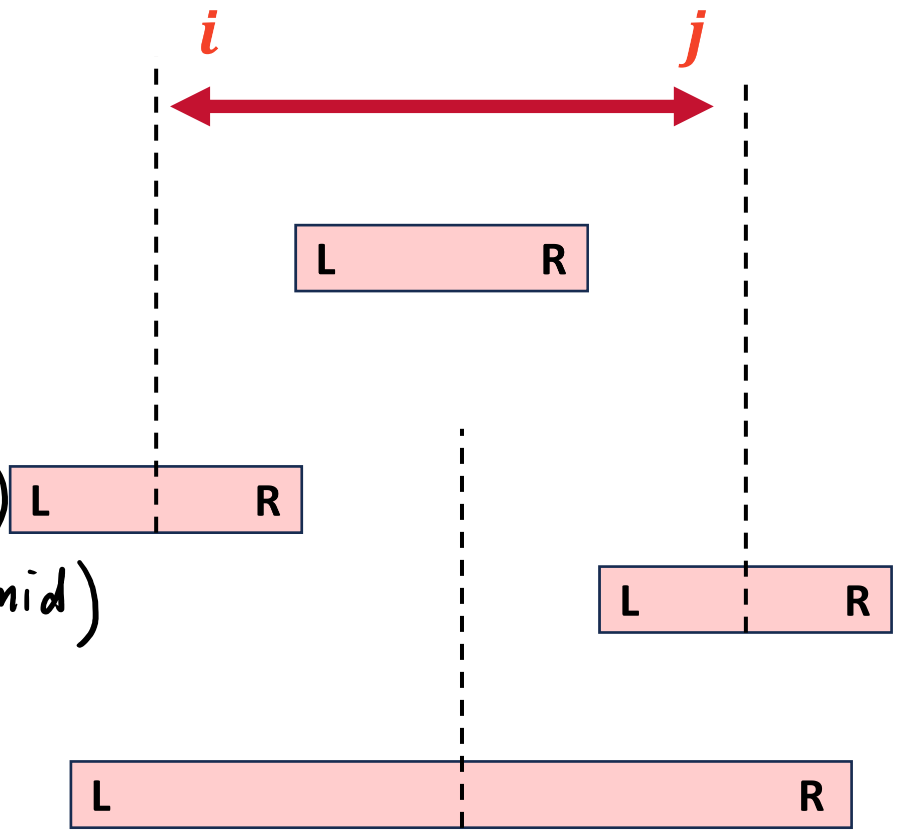
```
// Recursively build the SegTree values
SegTree::build(int node, int left, int right) {
    mid := (left + right) / 2
    if LeftChild(node) < n – 1 then
        build(LeftChild(node), left, mid)
    if RightChild(node) < n – 1 then
        build(RightChild(node), mid, right)
    A[node] = A[LeftChild(node)] +
              A[RC(node)]
}
```

Updating

```
SegTree::assign(int i, int x) {
    int node := i + n – 1;
    A[node] = x;
    while node > 0 do {
        node = Parent(node);
        A[node] = A[leftChild(node)] + A[RightChild(node)]

    }
}

int Parent(int u) { return (u-1)/2; }
```

Range Query

```
int sum(int node, int i, int j, int left, int right) {
    if (i <= left and right <= j) then  return A[node]
    else {
        int mid := (left + right) / 2;
        if (i >= mid) then return  sum(RightChild(node), i, j, mid, r)
        else if (j <= mid) then return sum (LC(node), i, j, left, mid)
        else {
            leftsum = sum (LC(node), i, j, left, mid)
            rightsum = sum (RC(node), i, j, mid, right)
            return  leftsum + rightsum

        }
    }
}
```




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

Question break

Speeding up algorithms

Problem (Inversion count): Given a permutation $p_0, p_1, \dots, p_{n-1}$, 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 each j:

count the number of $i < j$ such that $p_i > p_j$

That sounds like a range query!!!

Faster inversion counting

```
// Remember: input are integers between 0 and n-1
int n = size(p);
SegTree counts = SegTree(list<int>(n,0));        // Initialize with n zeros
int result = 0;
for j = 0 to n – 1  do {
    result += counts.RangeQuery(p[j], n)
    counts.Assign(p[j], 1)

}
return result
```

Extensions of SegTrees

- Reducing over other associative operations!
  - $+$ can be replaced with any binary associative operator, e.g., min, max
  - A fun example awaits you in recitation

- Extra operations
  - We supported the API: **Assign**(i, x) and **RangeSum**(i, j)
  - What if we want to read a single element?
  - What if we want to add to an element instead of assigning?

More Extensions of SegTrees

- What if we want ***range updates*** instead of range queries?
  - **RangeAdd**(i, j, x): Add x to every element $a_i, \dots, a_{j-1}$
  - **Get**(i): Return a_i

RangeAdd(i, j, x) : Add(i, x)
Add$(j, -x)$

Get(i) : RangeSum$(0, i+1)$