Segment Trees and Prefix Sum Arrays

Bloor CS Club 2020

What do I mean by Range Query Structures?

I will be talking about data structures useful for solving problems similar to the following one:

- You are given an array of integers

3	6	2	1	8	3	5	4

What do I mean by Range Query Structures?

I will be talking about data structures useful for solving problems similar to the following one:

- You are given an array of integers
- You will then be repeatedly asked to calculate something about a certain range of this array (e.g you might be asked for the sum of all elements between indices 1 and 5)

3	6	2	1	8	3	5	4

What do I mean by Range Query Structures?

I will be talking about data structures useful for solving problems similar to the following one:

- You are given an array of integers
- You will then be repeatedly asked to calculate something about a certain range of this array (e.g you might be asked for the sum of all elements between indices 1 and 5)
- In some cases, you might be required to update certain elements of the array (e.g adding 3 to all elements between indices 2 and 4)

3	6	2^{+4}	1^{+4}	8^{+4}	3	5	4

Prefix Sum Arrays

Prefix Sum Arrays

For now, we will be focusing on the version of the problem where you have to return the sum of a certain subarray of elements. We also won't worry about updates (yet).

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Prefix Sum Arrays

Every element of a prefix sum array is equal to the sum of all previous elements of the original array.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Queries

The nice thing about prefix sum arrays is that they perform queries really fast. In fact, they can perform them in **O(1)** time. For example, imagine that you are asked to calculate the sum of all elements of the array between index 2 and 6.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

Queries

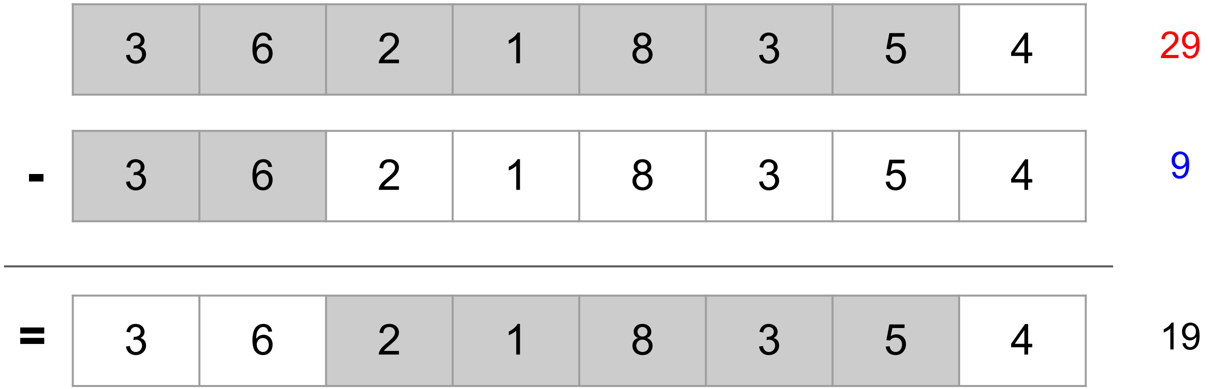
The nice thing about prefix sum arrays is that they perform queries really fast. In fact, they can perform them in **O(1)** time. For example, imagine that you are asked to calculate the sum of all elements of the array between index 2 and 6.

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	32

28 - 9 = 19

2+1+8+3+5 = 19

Why did this work?



Construction

Original array:

3	6	2	1	8	3	5	4

Prefix sum array:

Total: 0

Construction

Original array:

3	6	2	1	8	3	5	4

Prefix sum array:

3

Total: 3

Construction

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9						

Total: 9

Construction

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11					

Total: 11

Construction

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12				

Total: 12

Construction

Original array:

3	6	2	1	8	3	5	4

Prefix sum array:

3	9	11	12	20

Total: 20

Construction

Original array:

3	6	2	1	8	3	5	4

Prefix sum array:

3	9	11	12	20	23

Total: 23

Construction

Original array:	3	6	2	1	8	3	5	4
Prefix sum array:	3	9	11	12	20	23	28	

Total: 28

Construction

Original array:

3	6	2	1	8	3	5	4

Prefix sum array:

3	9	11	12	20	23	28	32

Total: 32

Implementation

Java

```java
int arrLength;
int[] arr;
long[] prefixArr;

void construct(){
    prefixArr = new long[arrLength];
    long total = 0;
    for (int i = 0; i < arrLength; i++) {
        total += arr[i];
        prefixArr[i] = total;
    }
}

long query(int l, int r){
    if(l == 0) return prefixArr[r];
    else return prefixArr[r] - prefixArr[l-1];
}
```

These slides were left out so that the lesson wouldn’t be too long

Higher Dimensions

Prefix sum arrays can also be used in more than one dimension. In 2D, each element of a prefix sum array represents the sum of all elements of the original array that have both indexes smaller than the element in the prefix sum array.

Original

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

Prefix

4	6	11	26
16	21	28	51
25	36	44	74
28	50	65	108

Higher Dimensions

Prefix sum arrays can also be used in more than one dimension. In 2D, each element of a prefix sum array represents the sum of all elements of the original array that have both indexes smaller than the element in the prefix sum array.

Original

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

Prefix

4	6	11	26
16	21	28	51
25	36	44	74
28	50	65	108

Higher Dimensions

Prefix sum arrays can also be used in more than one dimension. In 2D, each element of a prefix sum array represents the sum of all elements of the original array that have both indexes smaller than the element in the prefix sum array.

Original

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

Prefix

4	6	11	26
16	21	28	51
25	36	44	74
28	50	65	108

Higher Dimensions

Prefix sum arrays can also be used in more than one dimension. In 2D, each element of a prefix sum array represents the sum of all elements of the original array that have both indexes smaller than the element in the prefix sum array.

Original

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

Prefix

4	6	11	26
16	21	28	51
25	36	44	74
28	50	65	108

Queries

2D prefix sum arrays can compute the sum of any rectangular selection of elements in **O(1)** time. For example, we can calculate the sum of all elements between (1, 1) and (3, 2)

Original

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

Prefix

4	6	11	26
16	21	28	51
25	36	44	74
28	50	65	108

Queries

2D prefix sum arrays can compute the sum of any rectangular selection of elements in **O(1)** time. For example, we can calculate the sum of all elements between (1, 1) and (3, 2)

Original

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

Prefix

4	6	11	26
16	21	28	51
25	36	44	74
28	50	65	108

44 - 11 - 25 + 4 = 12

3+2+6+1 = 12

Why does this work?

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

-

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

-

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

+

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

=

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

Construction

Construction is done row by row.
The first row is constructed normally.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

Total : 0

Construction

Construction is done row by row.
The first row is constructed normally.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4			

Total : 4

Construction

Construction is done row by row.
The first row is constructed normally.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6		

Total : 6

Construction

Construction is done row by row.
The first row is constructed normally.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	

Total : 11

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26

Total : 26

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26

Total : 0

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16			

Total : 12

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21		

Total : 15

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	

Total : 17

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51

Total : 25

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that’s one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51

Total : 0

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51
25			

Total : 9

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51
25	36		

Total : 15

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51
25	36	44	

Total : 16

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51
25	36	44	74

Total : 23

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51
25	36	44	74

Total : 0

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51
25	36	44	74
28			

Total : 3

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51
25	36	44	74
28	50		

Total : 14

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51
25	36	44	74
28	50	65	

Total : 21

Construction

Construction is done row by row.
The first row is constructed normally.
For every other row, we still keep track of the total sum of elements in that row (just like in a 1D array) but we also add the value in the prefix array that's one above the current element.

4	2	5	15
12	3	2	8
9	6	1	7
3	11	7	13

4	6	11	26
16	21	28	51
25	36	44	74
28	50	65	108

Total : 34

Implementation

Java

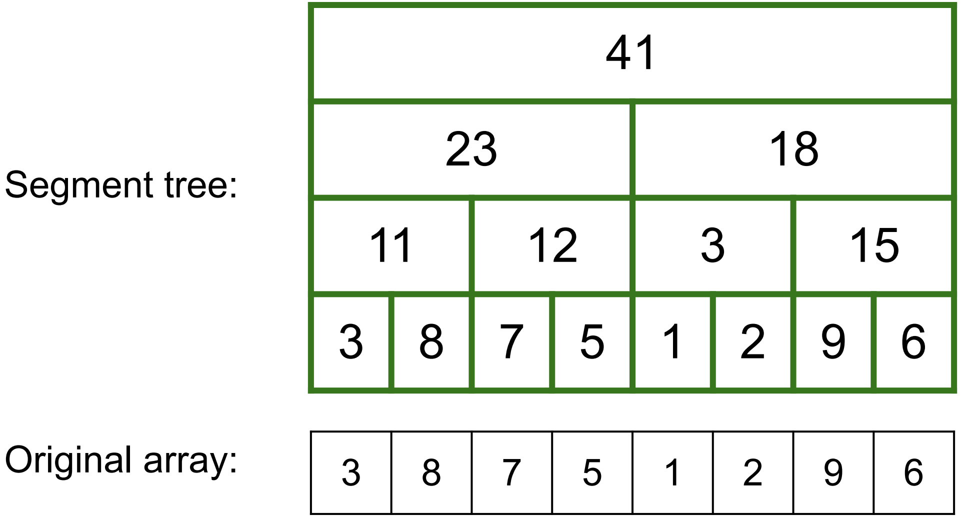
```java
int width;
int height;
int[][] arr;
long[][] prefixArr;

void construct(){
    prefixArr = new long[width][height];

    // Construct first row
    long total = 0;
    for (int x = 0; x < height; x++) {
        total += arr[x][0];
        prefixArr[x][0] = total;
    }

    // Construct the rest of the rows
    for (int y = 1; y < height; y++) {
        total = 0;
        for (int x = 0; x < width; x++) {
            total += arr[x][y];
            prefixArr[x][y] = total + prefixArr[x][y-1];
        }
    }
}
```

```java
long query(int x1, int x2, int y1, int y2){
    long result = prefixArr[x2][y2];

    if(x1 != 0) result -= prefixArr[x1-1][y2];
    if(y1 != 0) result -= prefixArr[x2][y1-1];
    if(x1 != 0 && y1 != 0) result += prefixArr[x1-1][y1-1];

    return result;
}
```

Segment Trees

Introduction

- A segment tree consists of segments, sometimes called nodes
- Each segment represents the sum of a certain subarray of elements
- Queries are done by adding together the sums of multiple segments

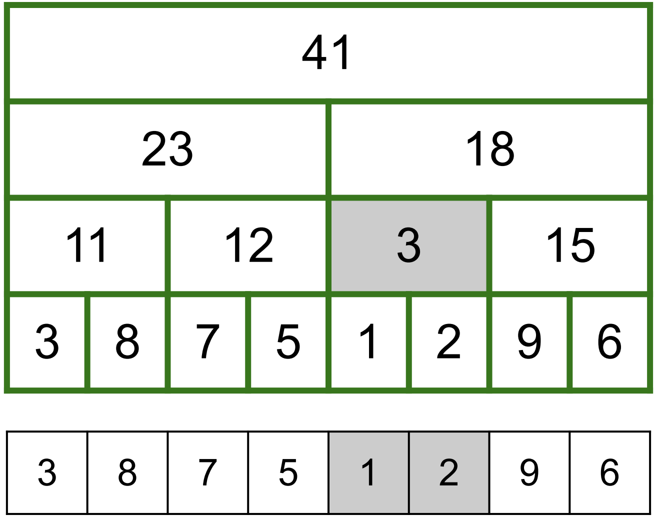


Introduction

- A segment tree consists of segments, sometimes called nodes
- Each segment represents the sum of a certain subarray of elements
- Queries are done by adding together the sums of multiple segments

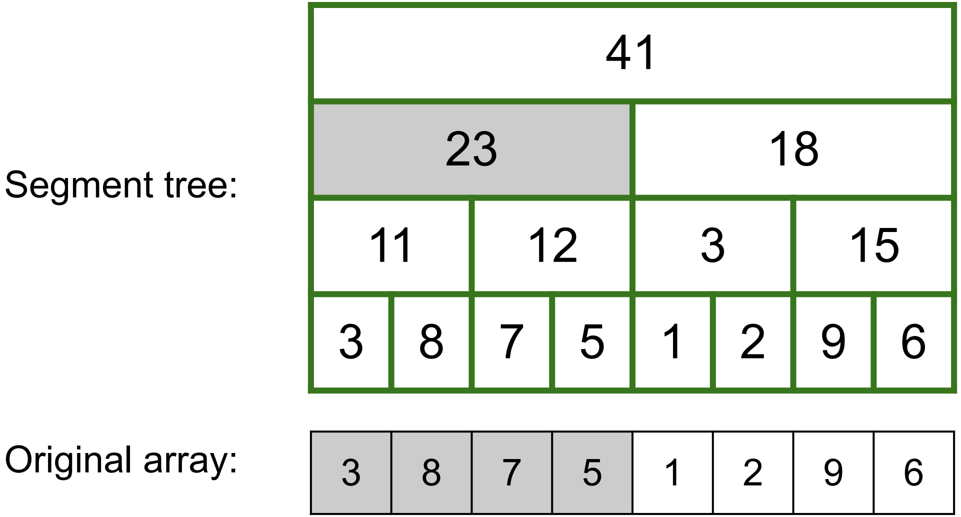
Segment tree:

Original array:



Introduction

- A segment tree consists of segments, sometimes called nodes
- Each segment represents the sum of a certain subarray of elements
- Queries are done by adding together the sums of multiple segments



Construction

Segment tree

109

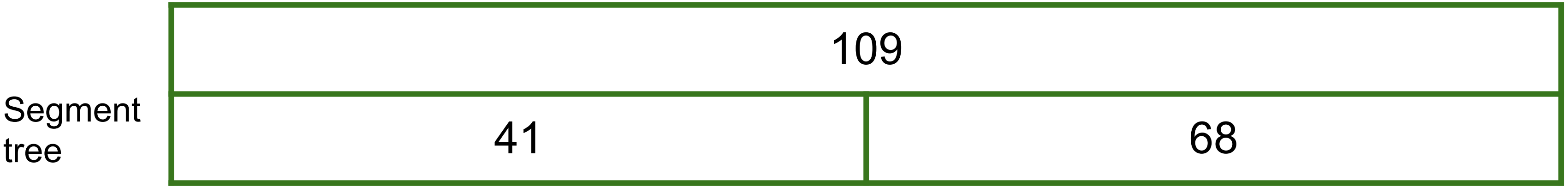
You start with the root segment, which is equal to the sum of the entire array.

Then you split it into 2 equal segments.

Original array

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

Construction



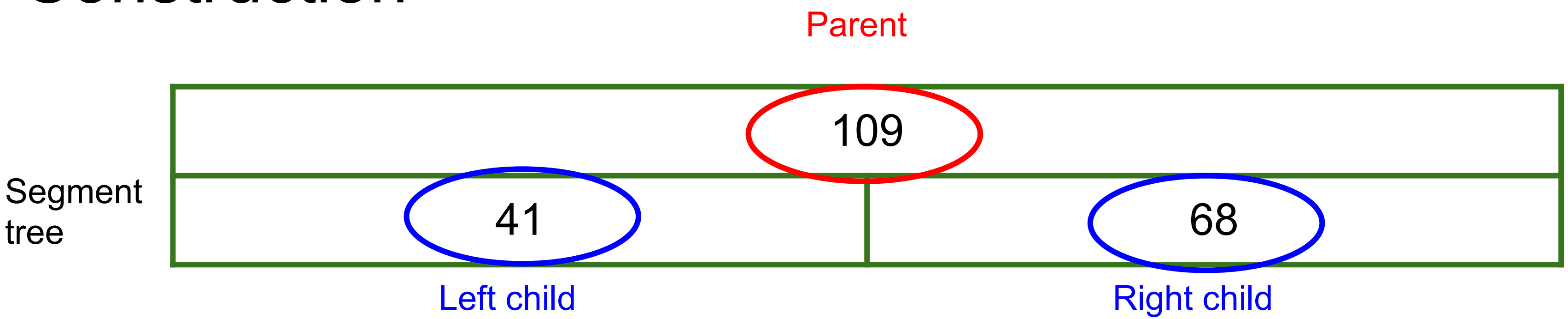
Then you split it into 2 equal segments.

These new segments are called the children of the original segment.

Original array

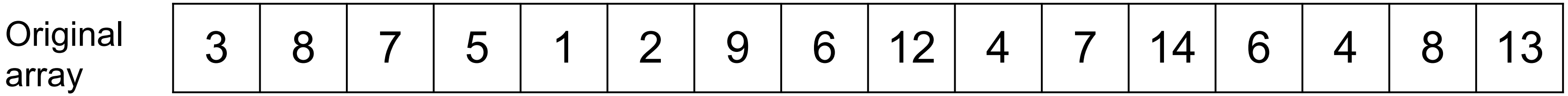
3	8	7	5	1	2	9	6	12	4	7	14	6	4	8	13

Construction



Then you split it into 2 equal segments.

These new segments are called the children of the original segment.



Construction

Segment tree

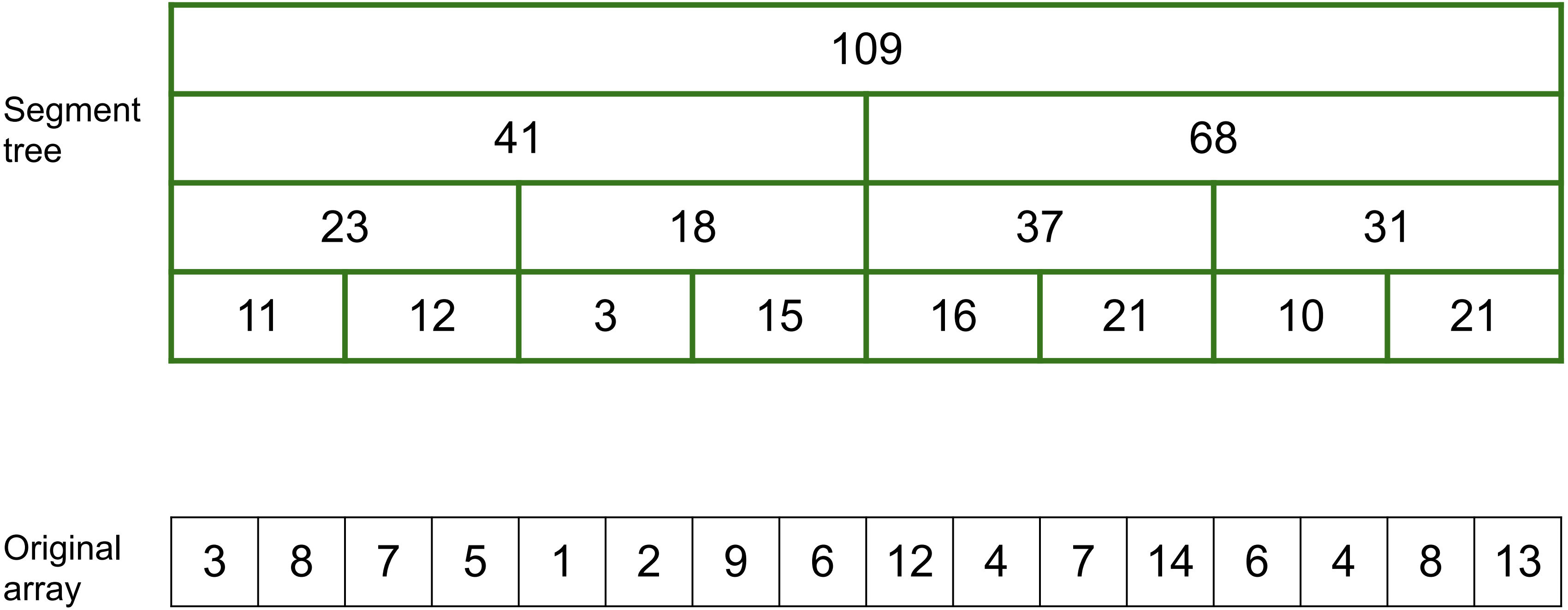
<table>
<tr><td colspan="4">109</td></tr>
<tr><td colspan="2">41</td><td colspan="2">68</td></tr>
<tr><td>23</td><td>18</td><td>37</td><td>31</td></tr>
</table>

Then you keep splitting each segment into 2 equal segments until you reach segments of length one.

Original array

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

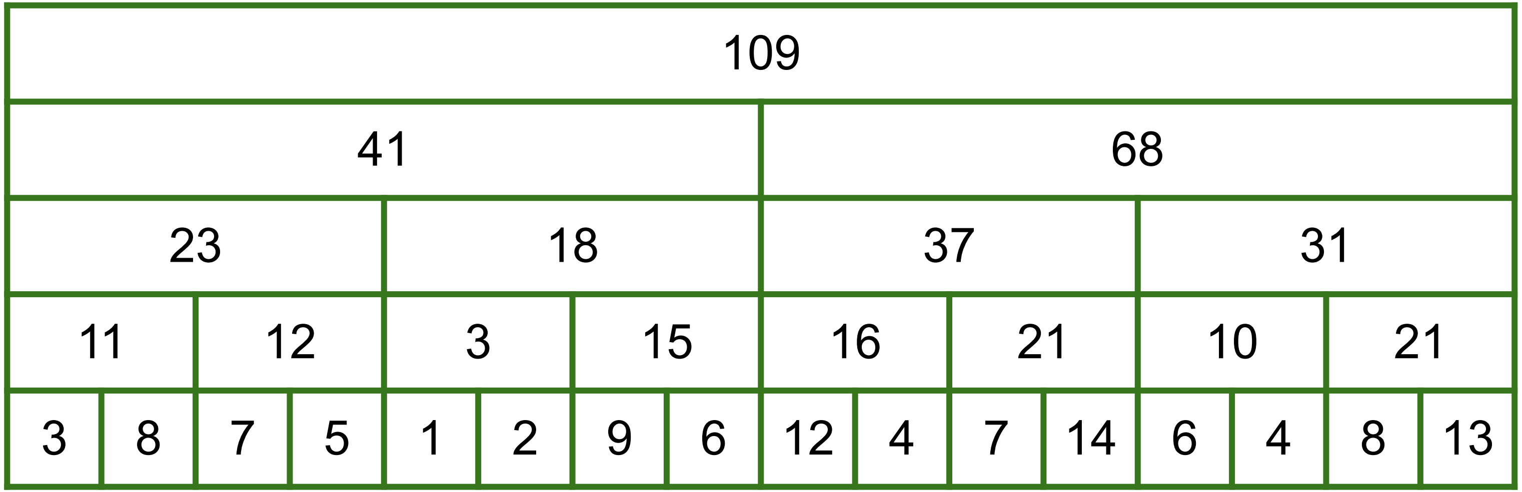
Construction



Construction

Height = $\log_2(N)$

Segment tree



Original array

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

Construction

If a segment has length 1, its value is equal to an element of the array.

Segment tree

<table>
<tr><td colspan="16">109</td></tr>
<tr><td colspan="8">41</td><td colspan="8">68</td></tr>
<tr><td colspan="4">23</td><td colspan="4">18</td><td colspan="4">37</td><td colspan="4">31</td></tr>
<tr><td colspan="2">11</td><td colspan="2">12</td><td colspan="2">3</td><td colspan="2">15</td><td colspan="2">16</td><td colspan="2">21</td><td colspan="2">10</td><td colspan="2">21</td></tr>
<tr><td>3</td><td>8</td><td>7</td><td>5</td><td>1</td><td>2</td><td>9</td><td>6</td><td>12</td><td>4</td><td>7</td><td>14</td><td>6</td><td>4</td><td>8</td><td>13</td></tr>
</table>

Original array

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

Construction

If a segment has length 1, its value is equal to an element of the array.
Otherwise, the value of a segment is equal to the sum of its two children.

Segment tree

<table>
<tr><td colspan="16">109</td></tr>
<tr><td colspan="8">41</td><td colspan="8">68</td></tr>
<tr><td colspan="4">23</td><td colspan="4">18</td><td colspan="4">37</td><td colspan="4">31</td></tr>
<tr><td colspan="2">11</td><td colspan="2">12</td><td colspan="2">3</td><td colspan="2">15</td><td colspan="2">16</td><td colspan="2">21</td><td colspan="2">10</td><td colspan="2">21</td></tr>
<tr><td>3</td><td>8</td><td>7</td><td>5</td><td>1</td><td>2</td><td>9</td><td>6</td><td>12</td><td>4</td><td>7</td><td>14</td><td>6</td><td>4</td><td>8</td><td>13</td></tr>
</table>

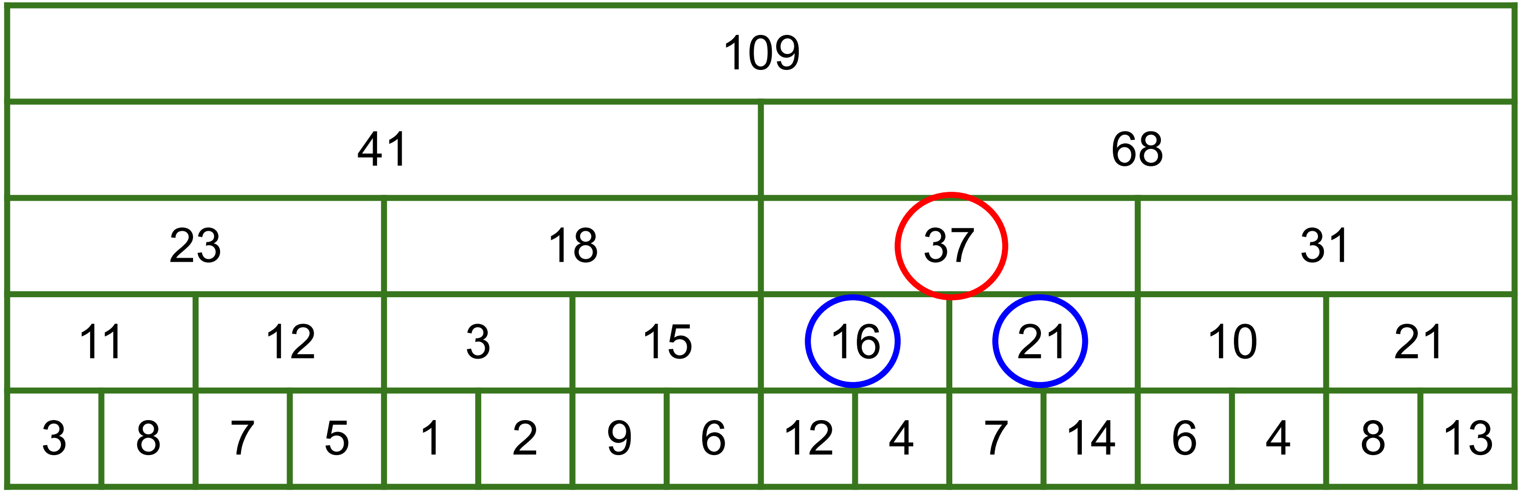
Original array

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

Construction

If a segment has length 1, its value is equal to an element of the array.
Otherwise, the value of a segment is equal to the sum of its two children.

Segment tree



Original array

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

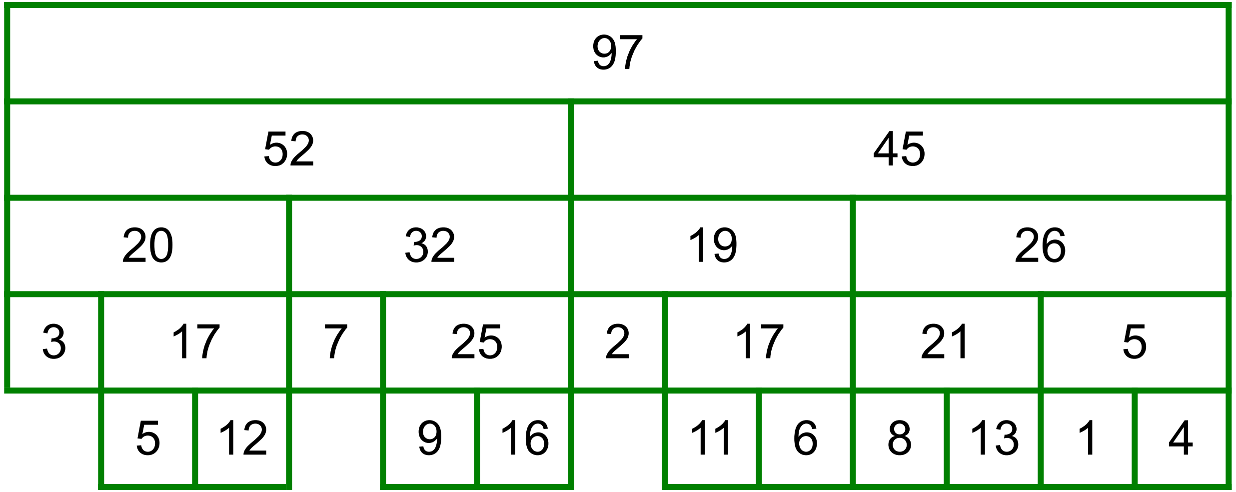
void construct (segment) {

If this segment has length of 1
    Set the value of this segment equal to the value of the corresponding element in the original array

Else
    Split this segment into two children of equal length
    Call the construct function on both of the children
    Set the value of this segment equal to the sum of the two children

What happens if you split a segment with an odd length?

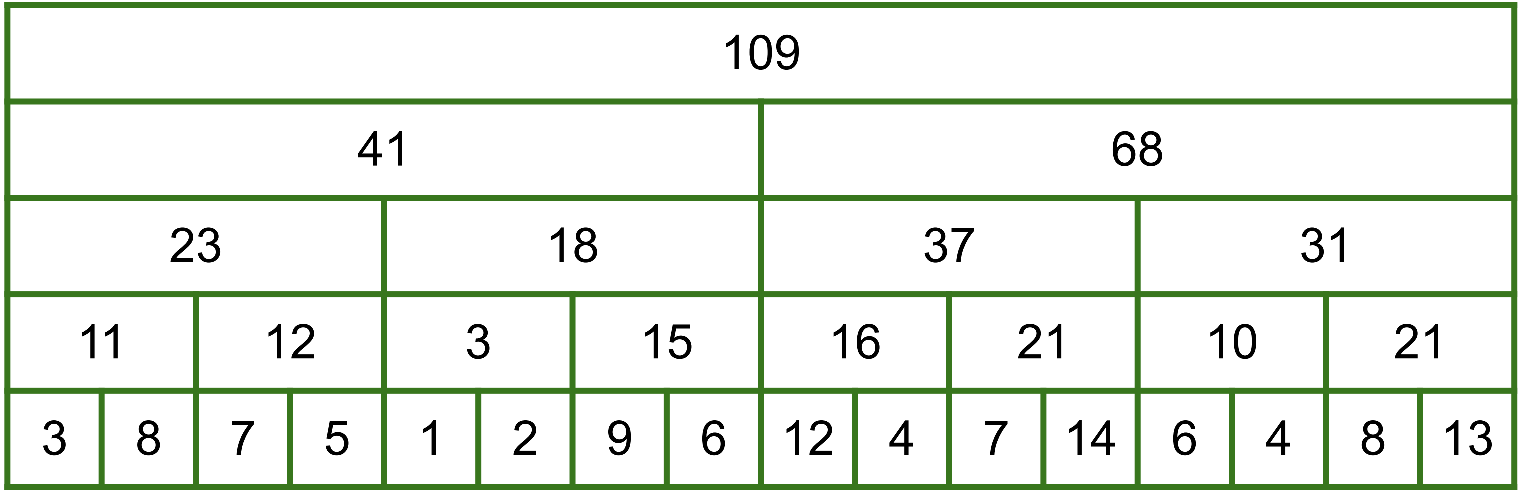
The segment tree won't look as nice, but it'll still work:



3	5	12	7	9	16	2	11	6	21	13	5	4

Queries

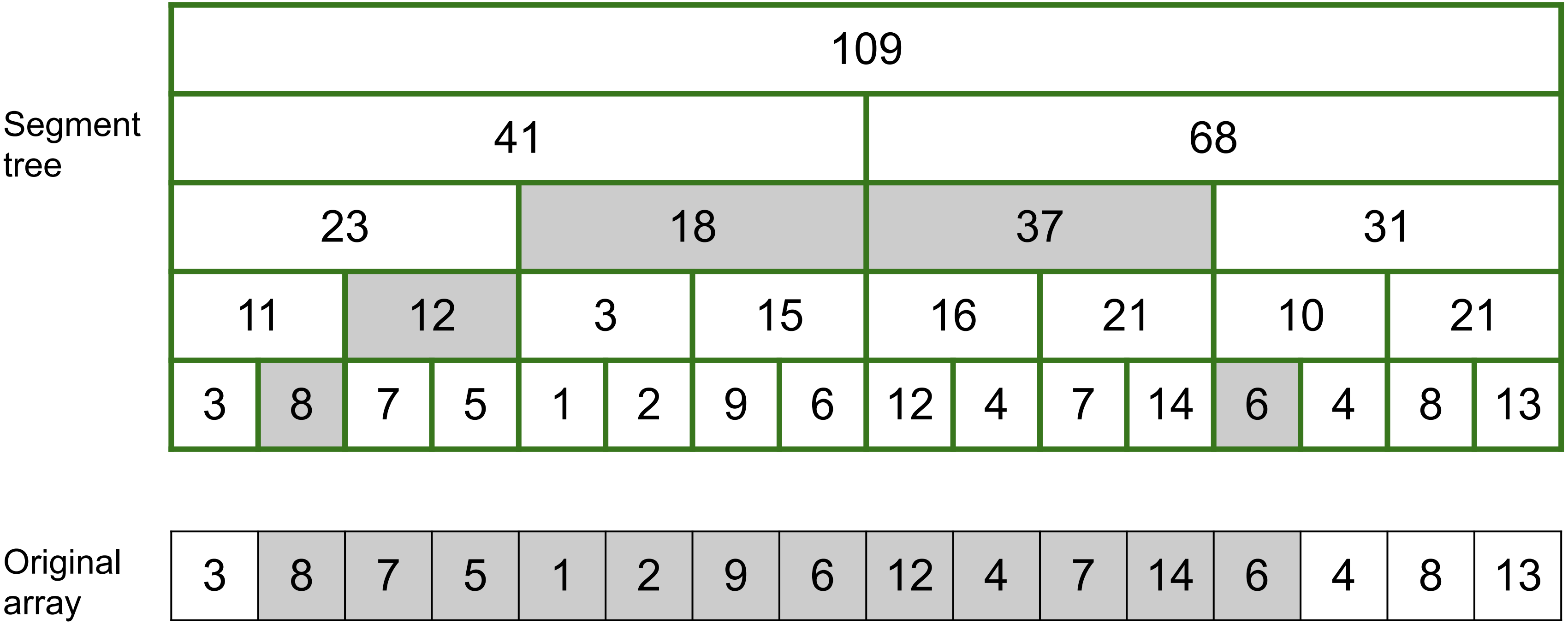
Segment tree



Original array

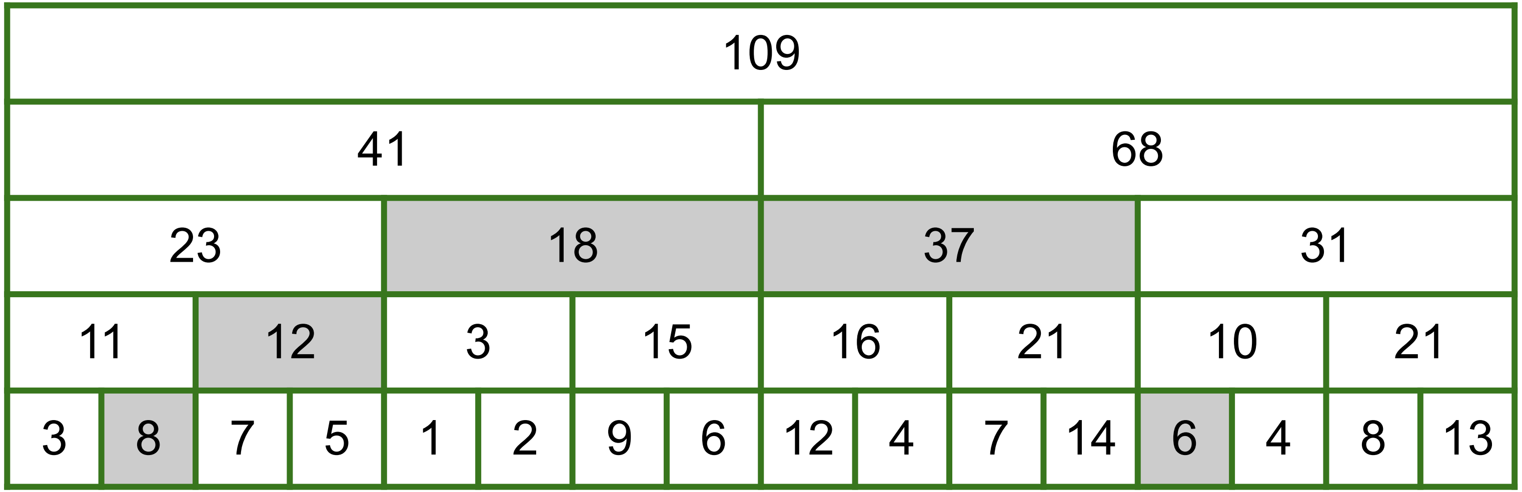
3	8	7	5	1	2	9	6	12	4	7	14	6	4	8	13

Queries


Segment tree
109
41
68
23
18
37
31
11
12
3
15
16
21
10
21
3 8 7 5 1 2 9 6 12 4 7 14 6 4 8 13
Original array
3 8 7 5 1 2 9 6 12 4 7 14 6 4 8 13

Queries

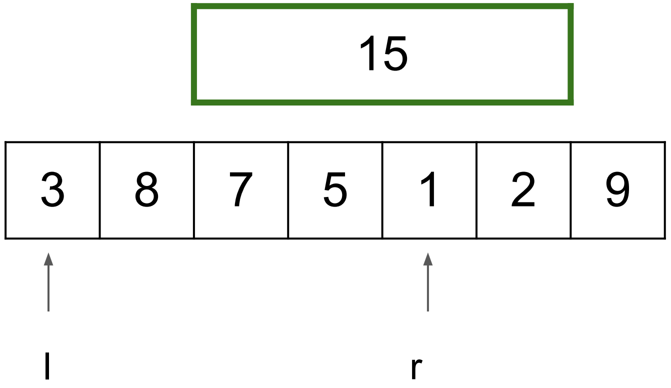
Segment tree



2 segments per layer * log(N) layers = **O(log(N))** time complexity for queries

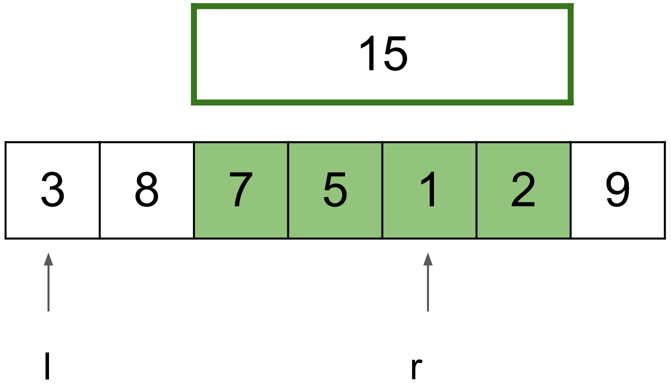
Queries

```
int query(segment, l, r) {
    // Returns the sum of the intersection of the segment and the range l..r
}
```



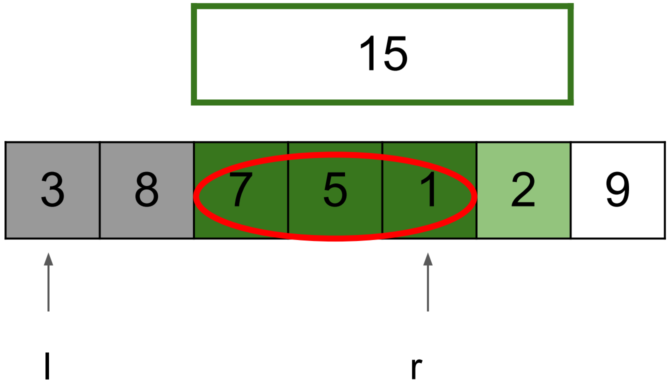
Queries

```
int query(segment, l, r) {
	// Returns the sum of the intersection of the segment and the range l..r
}
```



Queries

```
int query(segment, l, r) {
    // Returns the sum of the intersection of the segment and the range l..r
}
```

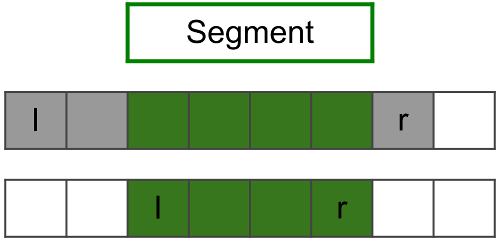


Why?

- Calling this function on the root segment has will return the sum of all elements between l and r
- Defining the query function in this way will help us later on

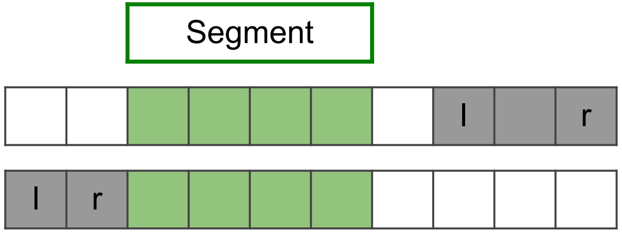
int query(segment, l, r) {

Segment falls entirely in range of l..r



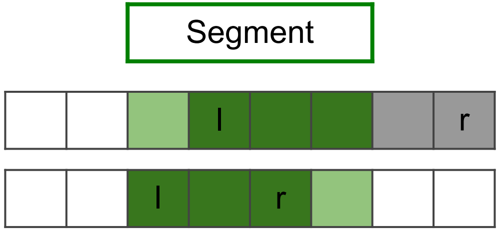
Return the value stored at the segment.

Segment falls entirely out of range of l..r



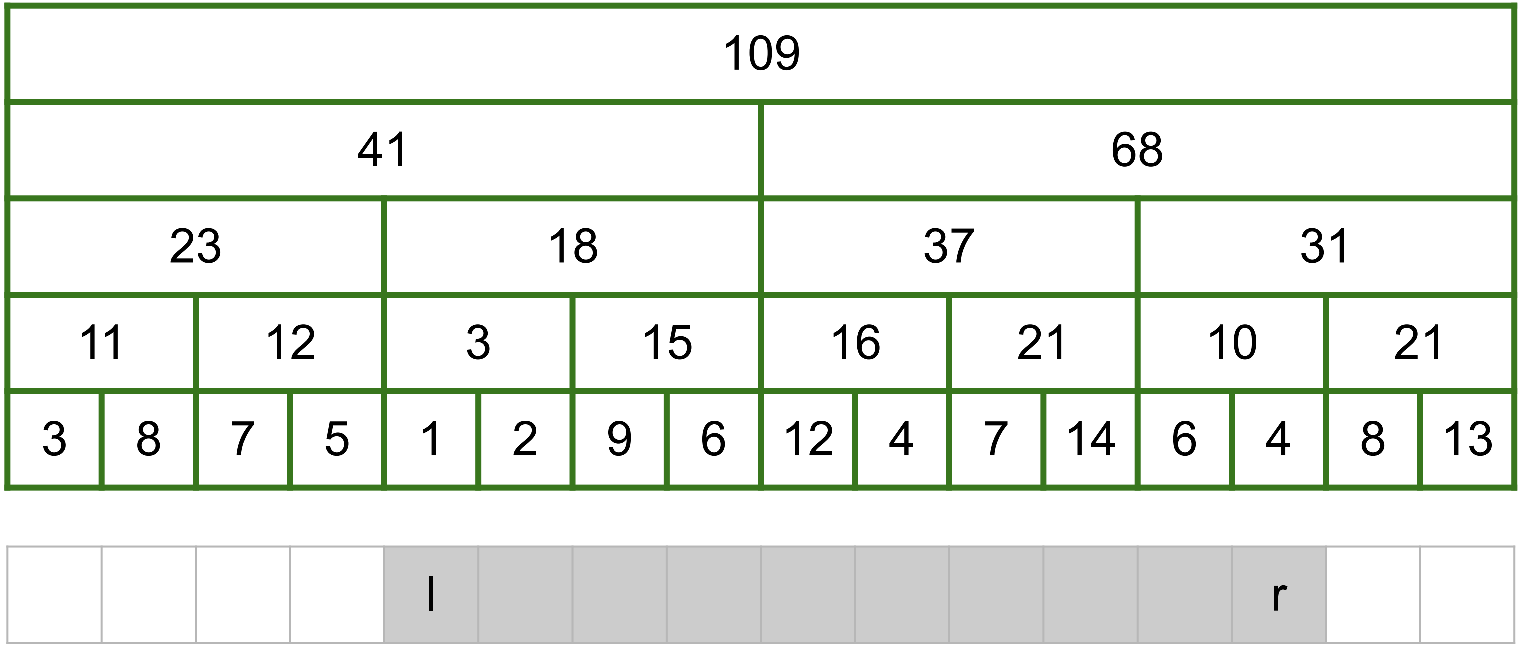
Return 0

Other

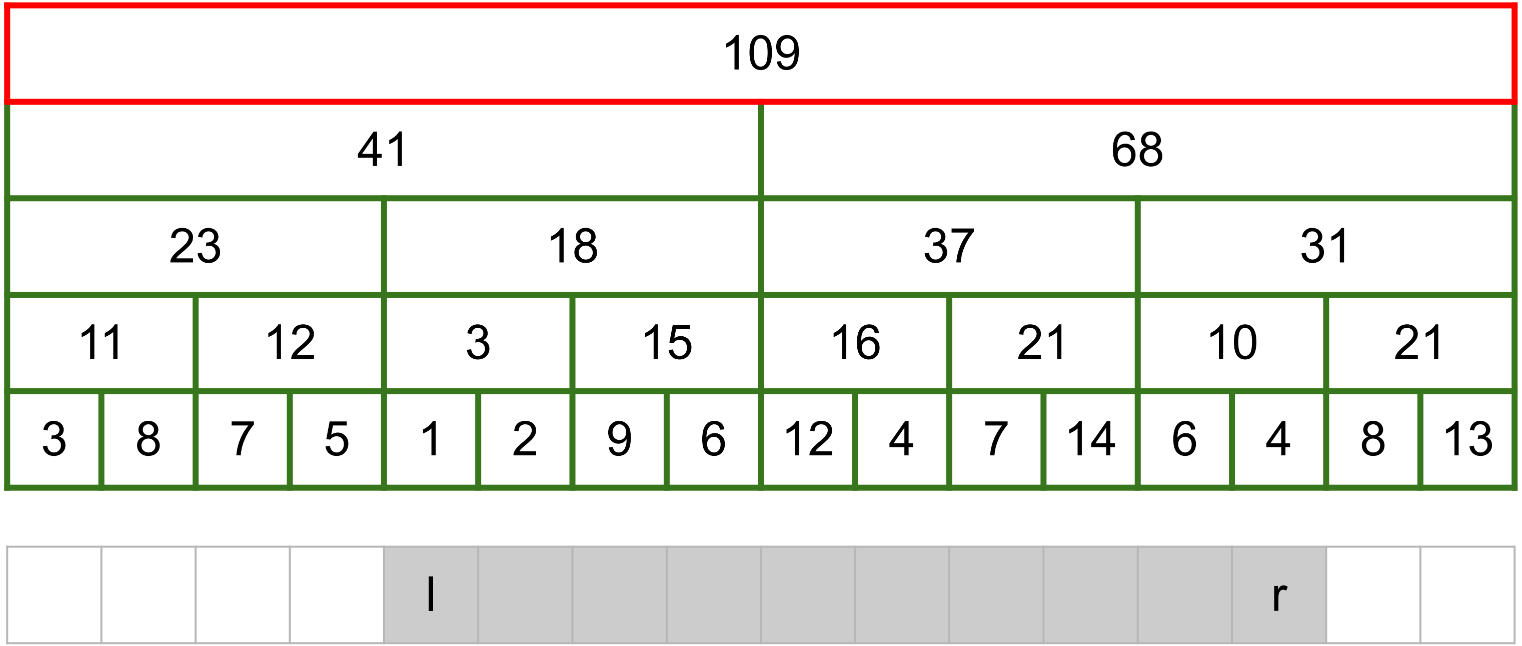


Return query(lChild, l, r) + query(rChild, l, r)

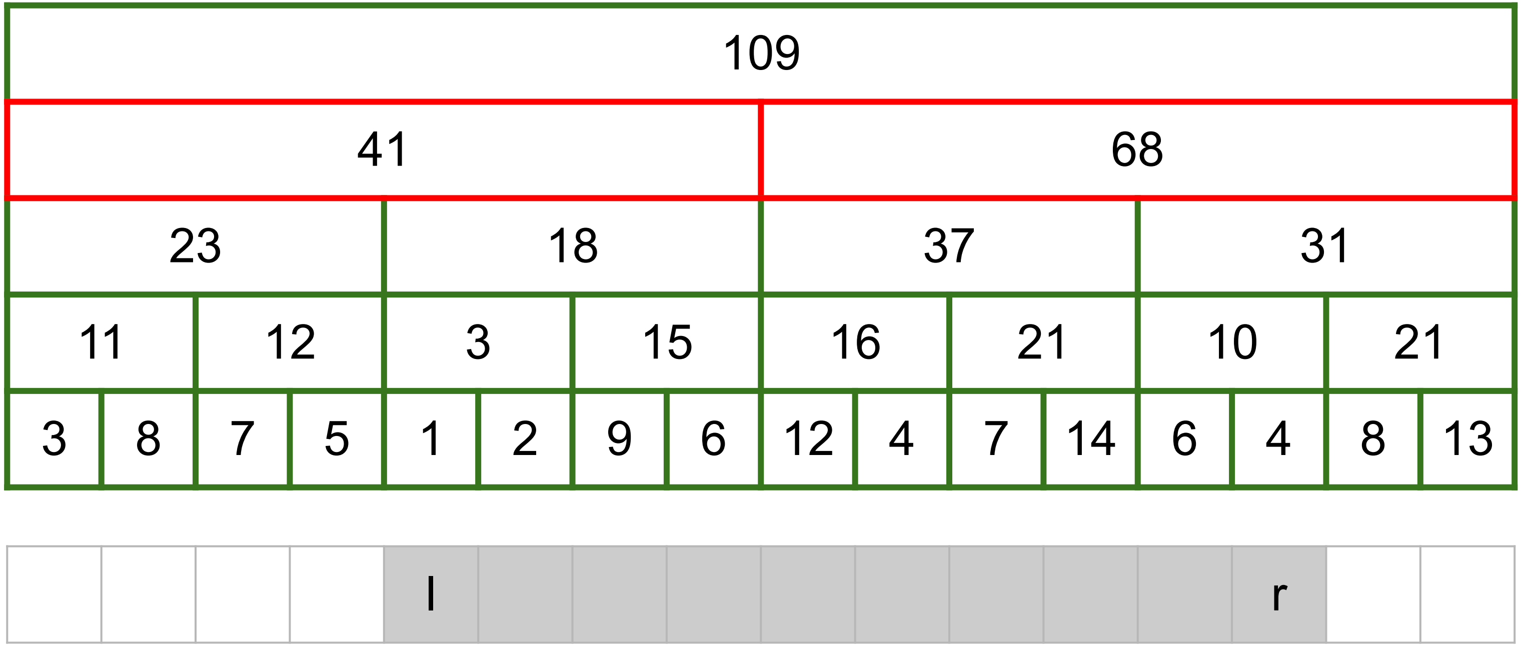
Example


109
41
68
23
18
37
31
11
12
3
15
16
21
10
21
3
8
7
5
1
2
9
6
12
4
7
14
6
4
8
13
l
r

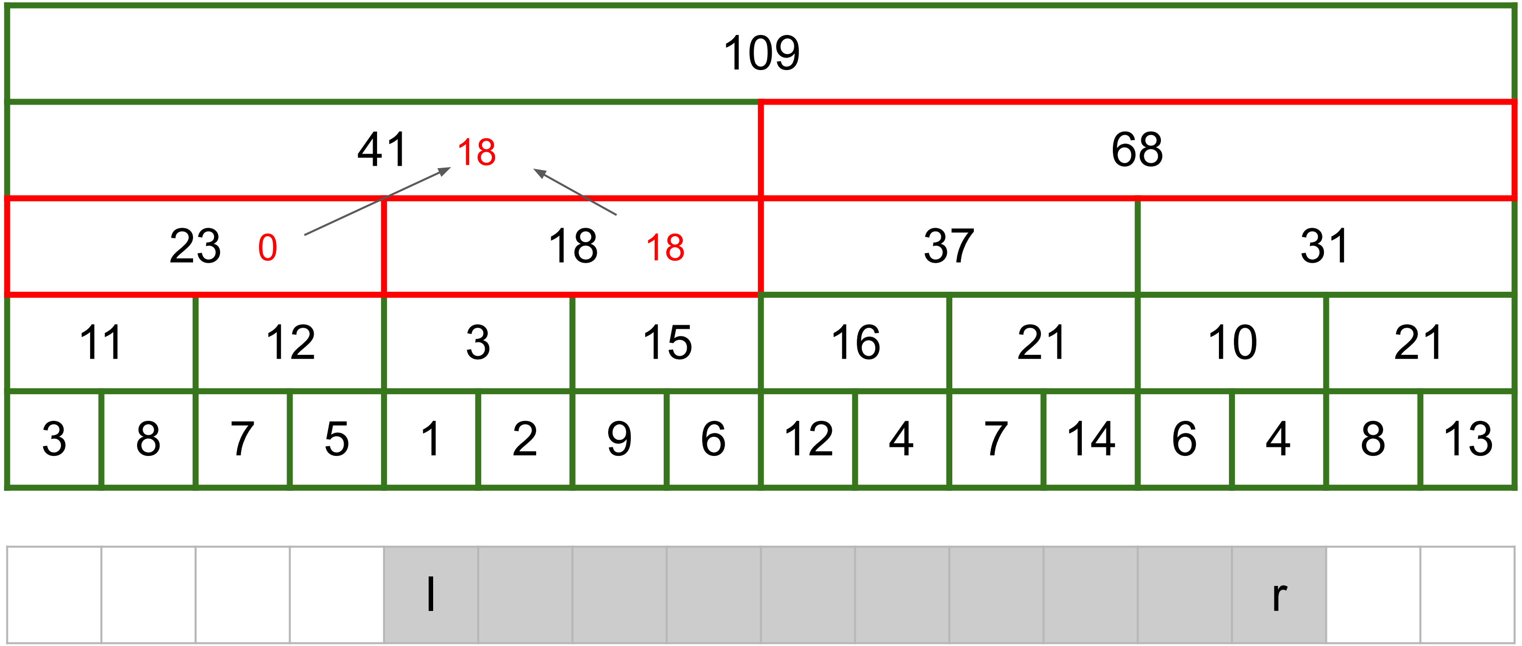
Example


109
41
68
23
18
37
31
11
12
3
15
16
21
10
21
3
8
7
5
1
2
9
6
12
4
7
14
6
4
8
13
l
r

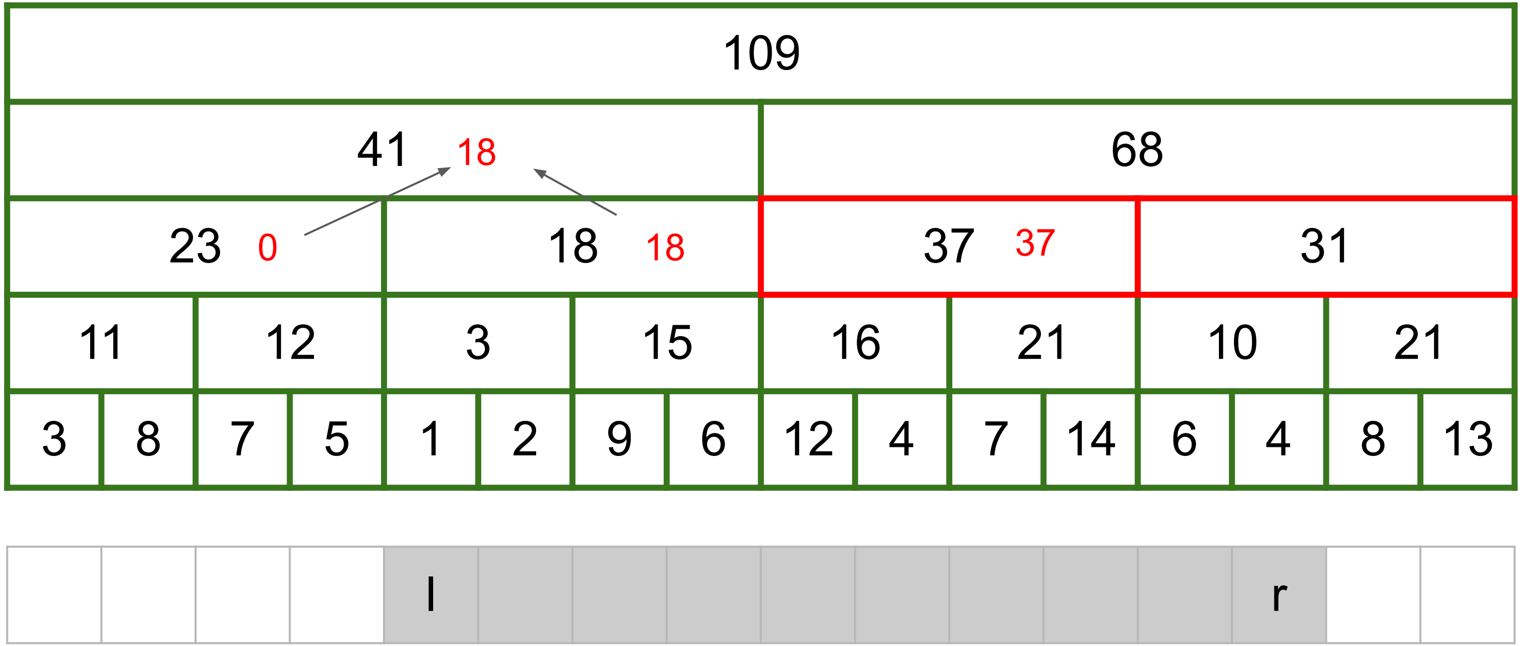
Example


109
41
68
23
18
37
31
11
12
3
15
16
21
10
21
3
8
7
5
1
2
9
6
12
4
7
14
6
4
8
13
l
r

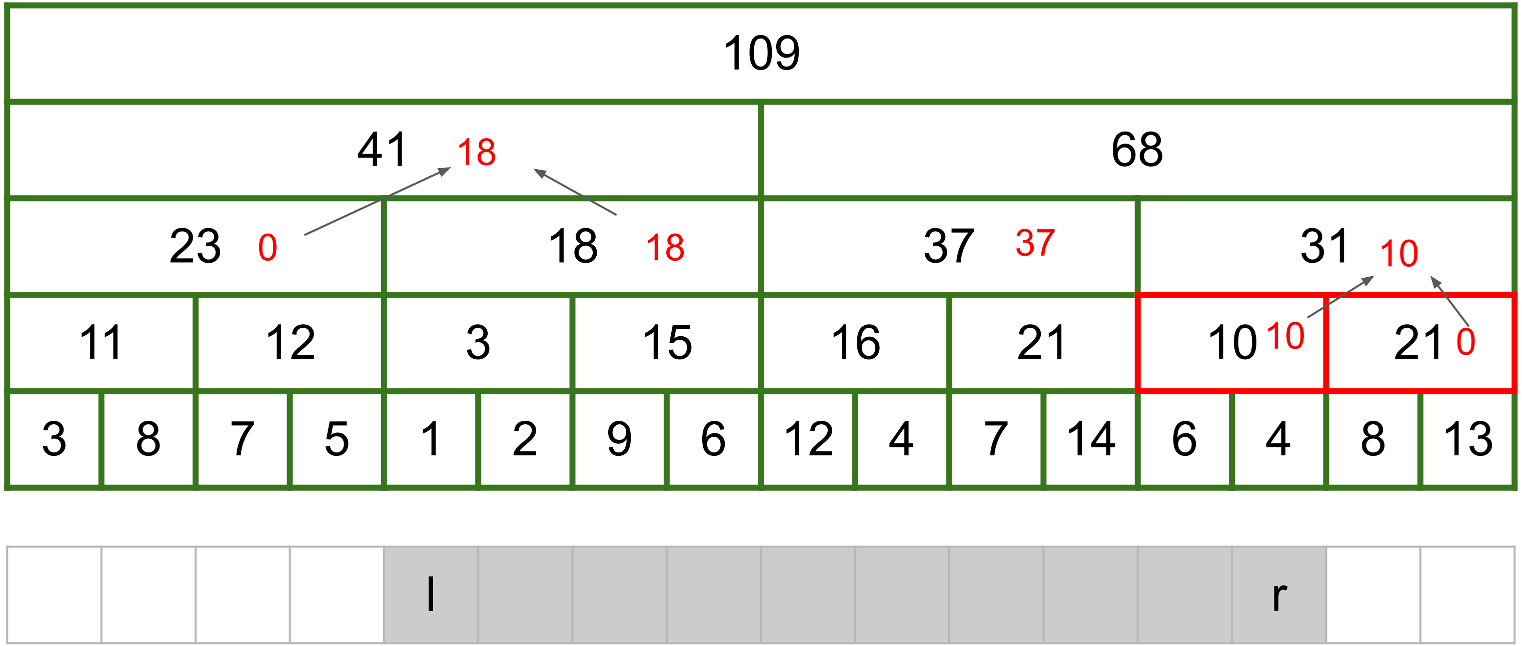
Example


109
41 18
68
23 0
18 18
37
31
11 12 3 15 16 21 10 21
3 8 7 5 1 2 9 6 12 4 7 14 6 4 8 13
l
r

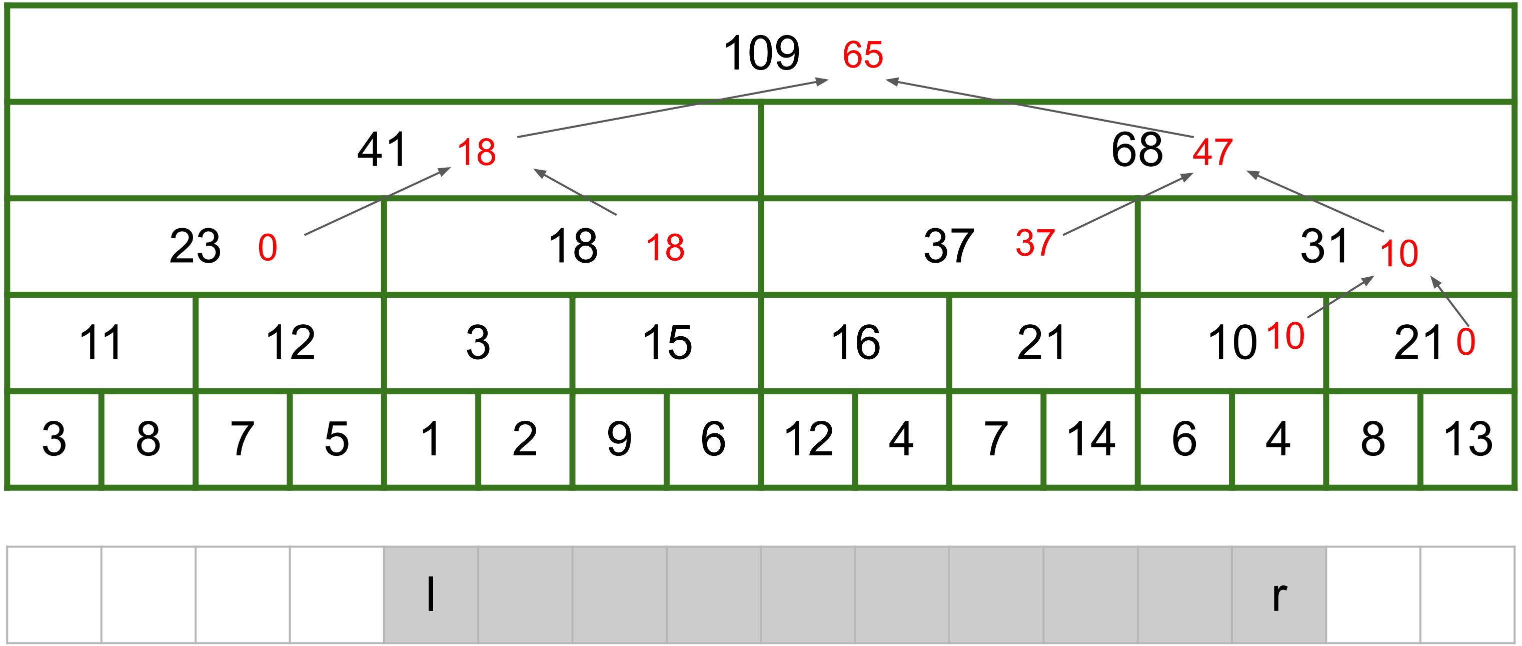
Example


109
41 18
68
23 0
18 18
37 37
31
11
12
3
15
16
21
10
21
3 8 7 5 1 2 9 6 12 4 7 14 6 4 8 13
l
r

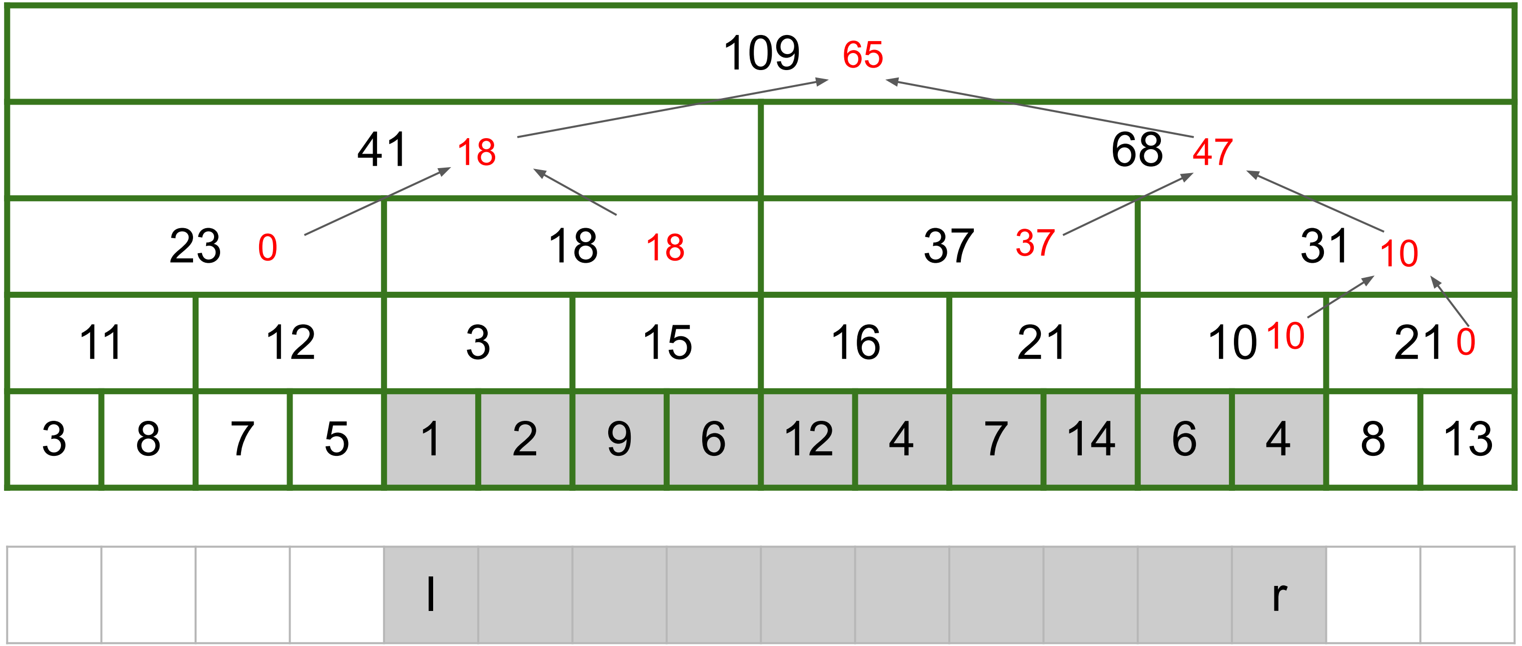
Example


109
41 18
68
23 0
18 18
37 37
31 10
11
12
3
15
16
21
10 10
21 0
3 8 7 5 1 2 9 6 12 4 7 14 6 4 8 13
l
r

Example

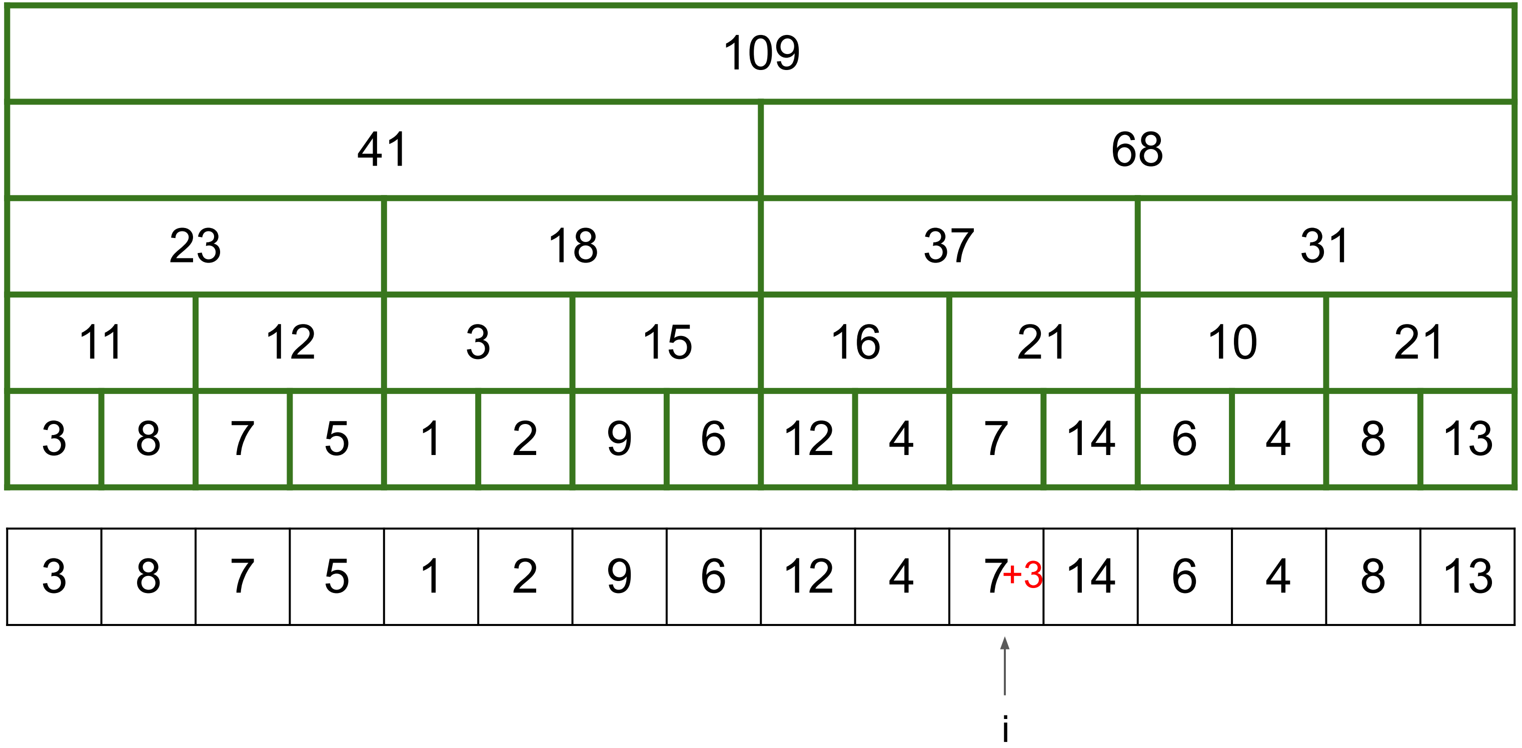

109 65
41 18
68 47
23 0
18 18
37 37
31 10
11
12
3
15
16
21
10 10
21 0
3 8 7 5 1 2 9 6 12 4 7 14 6 4 8 13
l
r

Example

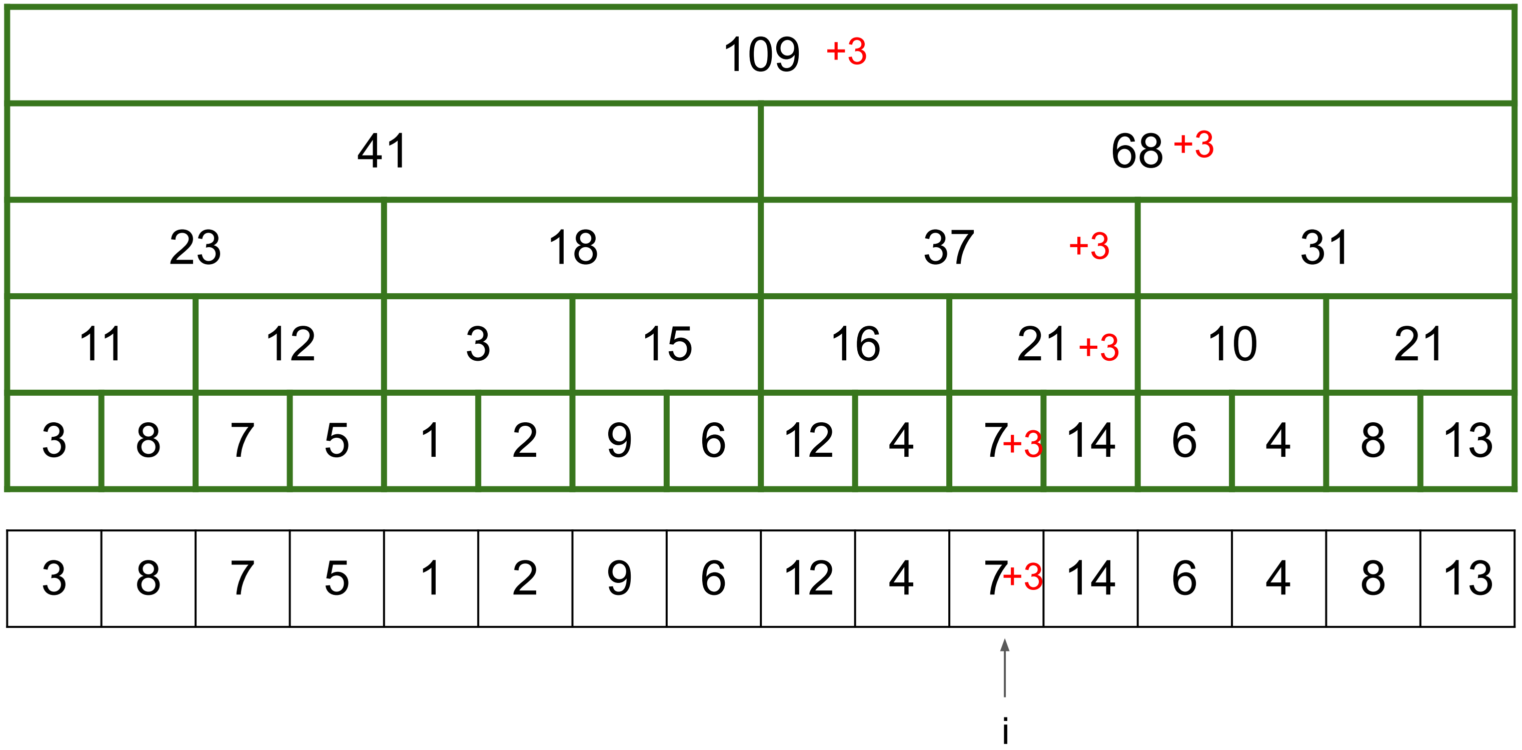


1+2+9+6+12+4+7+14+6+4 = 65

Updating an Element



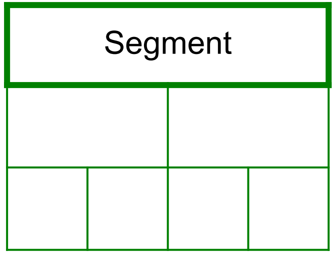
Updating an Element



O(log(N))

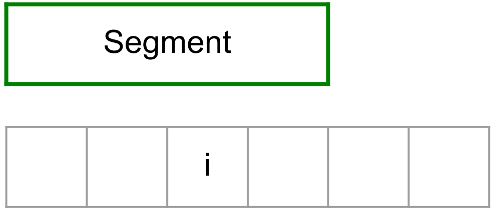
Updating an Element

```
void update(segment, i, amount) {
        // Updates the value of this segment and all of
        // the segments below it
}
```



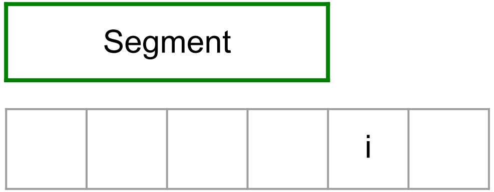
Updating an Element

The segment contains the element



Increase the value of segment by *amount*
update(lChild, i, amount)
update(rChild, i, amount)

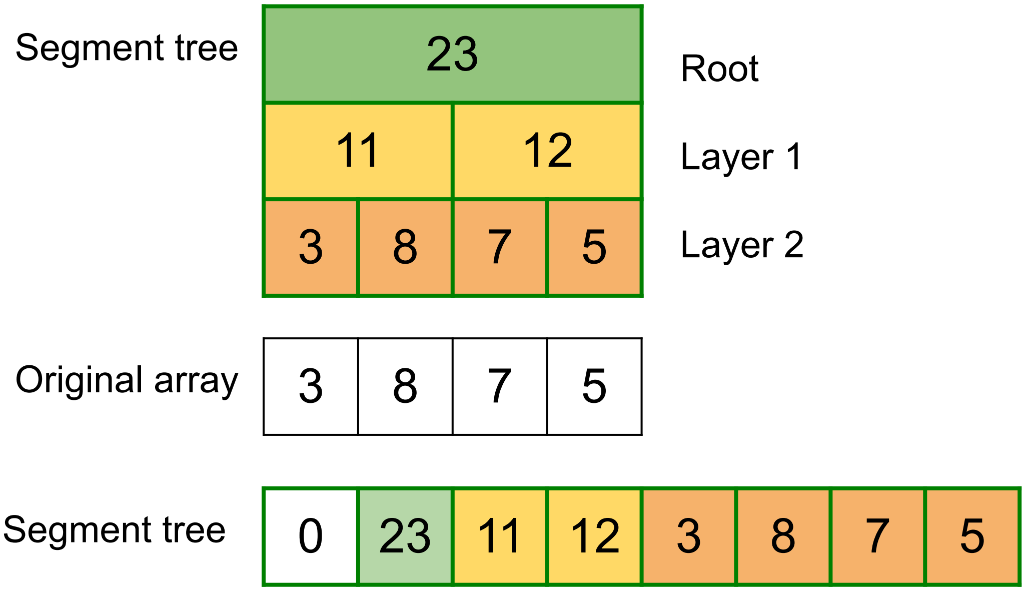
The segment does not contain the element



Do nothing

Implementation

Array Representation



The root is at index 1.

The children of the segment with index *i* are the segments with indices *i*2* and *i*2 + 1*

Implementation

Java

```
public class SegmentTree {
    int arrLength; // length of the original data
    int[] arr; // original data
    long[] tree; // array representation of this segment tree
```

Tells the function which segment to perform the operation on

```
int query(int i, int segBegin, int segEnd, int l, int r)
```

Index of the current segment in the array representation of this tree.

For example,
i = 1 means this is the root
i = 2 means this is the left child of the root
i = 3 means this is the right child of the root…

The value of the current segment is *tree[i]*

Start and end of this segment.
The value of this segment is equal to the sum of all elements in the original array whose indexes are between *segBegin* (inclusive) and *segEnd* (exclusive).

For example, if *segBegin* = 2 and *segEnd* = 5, then the value of this segment is equal to *arr[2]* + *arr[3]* + *arr[4].*

Construction

```
public SegmentTree(int[] arr){
    this.arrLength = arr.length;
    this.arr = arr;

    int height = 32 - Integer.numberOfLeadingZeros(arrLength-1); // Log base 2 of arrLength
    int size = 1 << (height + 1); // 2 to the power of height + 1
    tree = new long[size];

    construct(1, 0, arrLength);
}

// Calculates the value of the specified segment
// Also returns this newly calculated value
private long construct(int i, int segBegin, int segEnd){
    if(segBegin + 1 == segEnd) return tree[i] = arr[segBegin]; // Segment has length of 1
    else {
        int mid = (segBegin + segEnd)/2;
        return tree[i] = construct(i*2, segBegin, mid) + construct(i*2+1, mid, segEnd);
    }
}
```

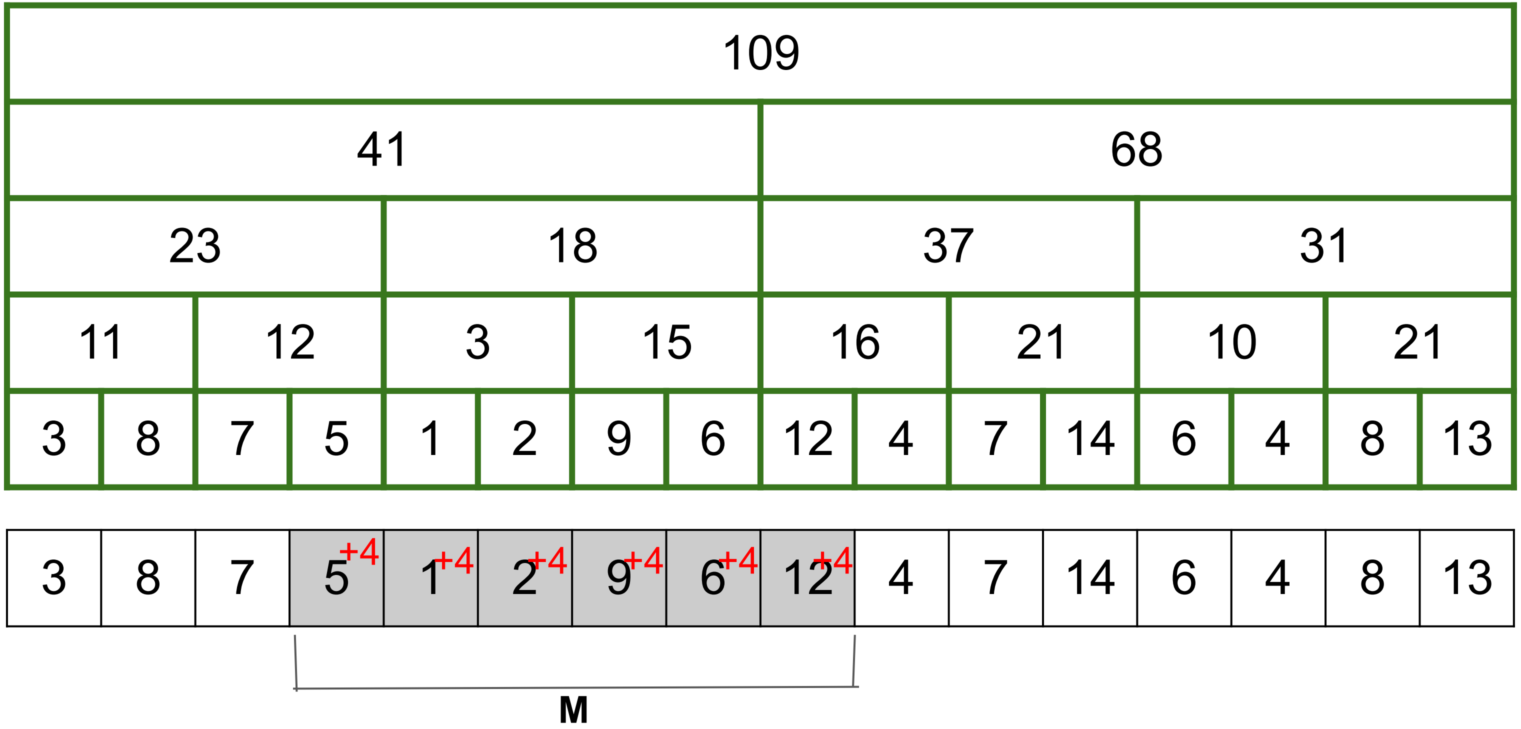
Queries

```
private long query(int i, int segBegin, int segEnd, int l, int r){
    if(l <= segBegin && r >= segEnd) return tree[i]; // Segment is fully contained in l..r
    if(r <= segBegin || l >= segEnd) return 0; // Segment is fully outside of l..r

    // Segment is partially in l..r
    int mid = (segBegin + segEnd)/2;
    return query(i*2, segBegin, mid, l, r) + query(i*2+1, mid, segEnd, l, r);
}

public long query(int l, int r){
    return query(1, 0, arrLength, l, r);
}
```

Updates

```
    private void update(int i, int segBegin, int segEnd, int pos, int amount){
        if(segBegin <= pos && segEnd > pos){
            tree[i] += amount;
            if(segBegin + 1 != segEnd) {
                int mid = (segBegin + segEnd)/2;
                update(i*2, segBegin, mid, pos, amount);
                update(i*2 + 1, mid, segEnd, pos, amount);
            }
        }
    }

    public void update(int pos, int amount){
        update(1, 0, arrLength, pos, amount);
    }
}
```

Range Updates



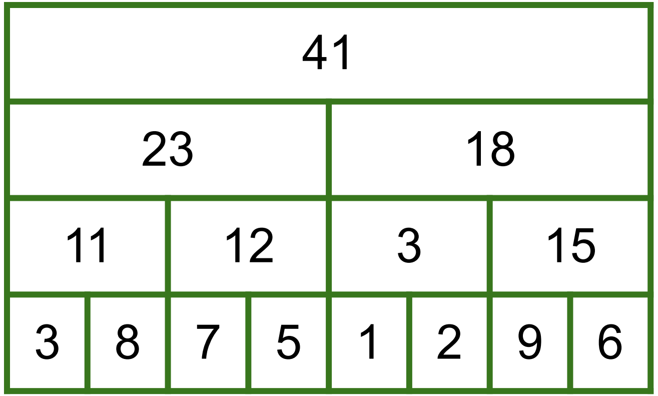
O(M * log(N))

This is too slow.

Lazy Propagation

Lazy Propagated Segment Trees

In a lazy propagated segment tree, every segment stores an additional value, called its lazy value.



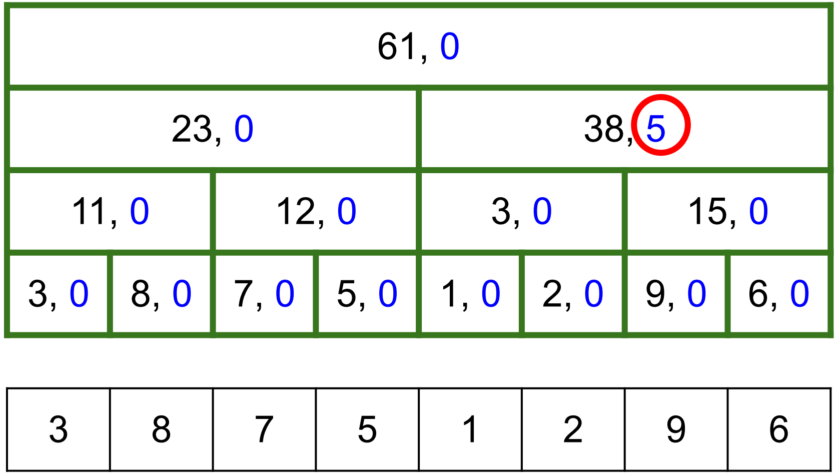
Lazy Propagated Segment Trees

In a lazy propagated segment tree, every segment stores an additional value, called its lazy value.

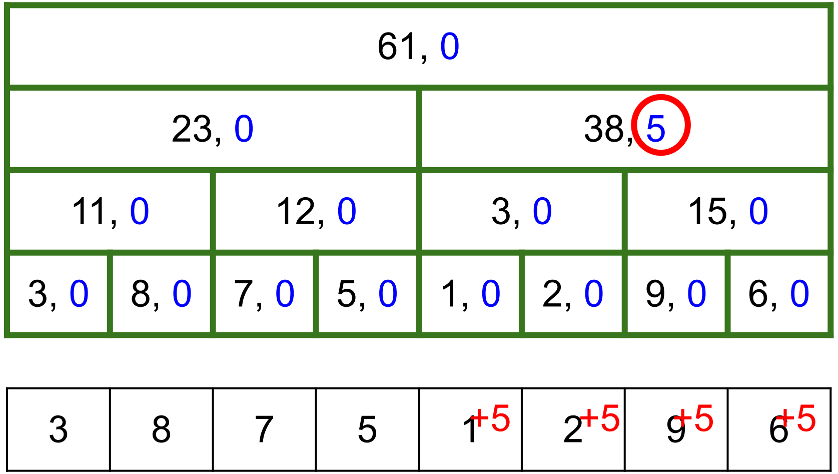
If a segment has a non-zero lazy value of *x*, then that means that every element in that segment was increased by *x*, but the values of all of all segments below this segment have not yet updated to reflect this change.

<table>
<tr><td colspan="8" align="center">41, 0</td></tr>
<tr><td colspan="4" align="center">23, 0</td><td colspan="4" align="center">18, 0</td></tr>
<tr><td colspan="2" align="center">11, 0</td><td colspan="2" align="center">12, 0</td><td colspan="2" align="center">3, 0</td><td colspan="2" align="center">15, 0</td></tr>
<tr><td>3, 0</td><td>8, 0</td><td>7, 0</td><td>5, 0</td><td>1, 0</td><td>2, 0</td><td>9, 0</td><td>6, 0</td></tr>
</table>

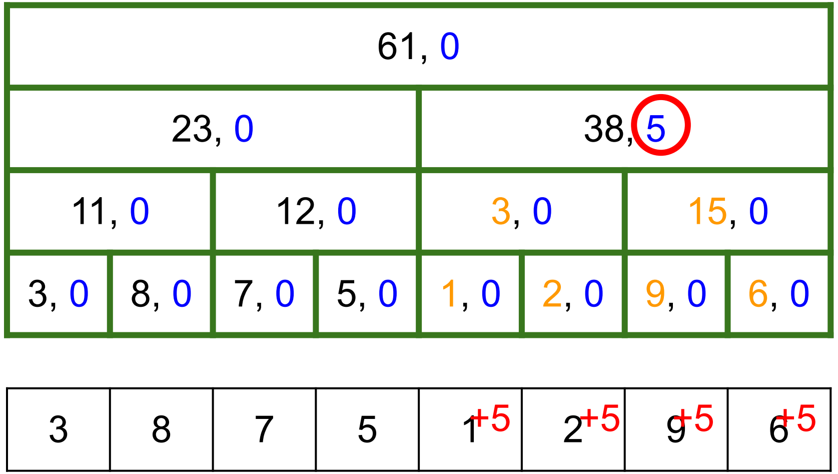
Lazy Propagated Segment Trees



Lazy Propagated Segment Trees

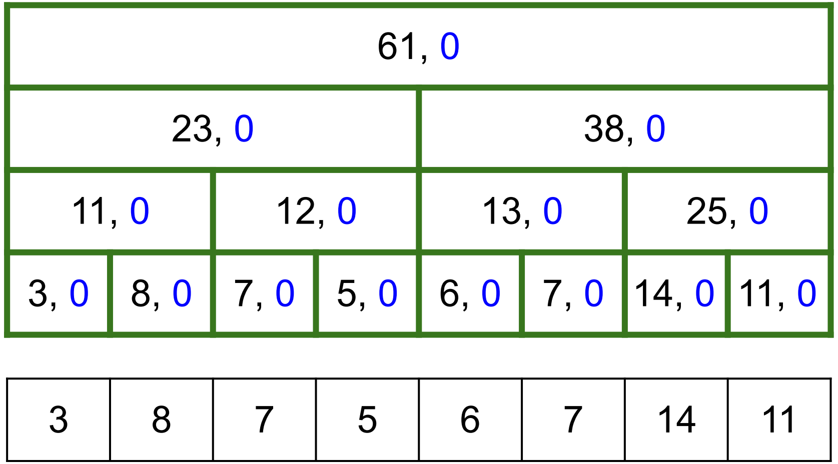


Lazy Propagated Segment Trees



=

After fully updating everything, the segment tree would look like this:



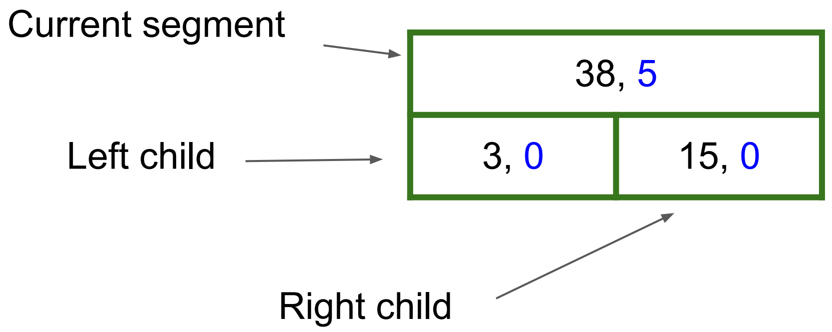
Update orange values later, when we actually need to.

How does this work?

Simple, before doing **any** operation with a segment, we check whether it has a lazy value. If it does, we update it (which takes **O(1)** time):

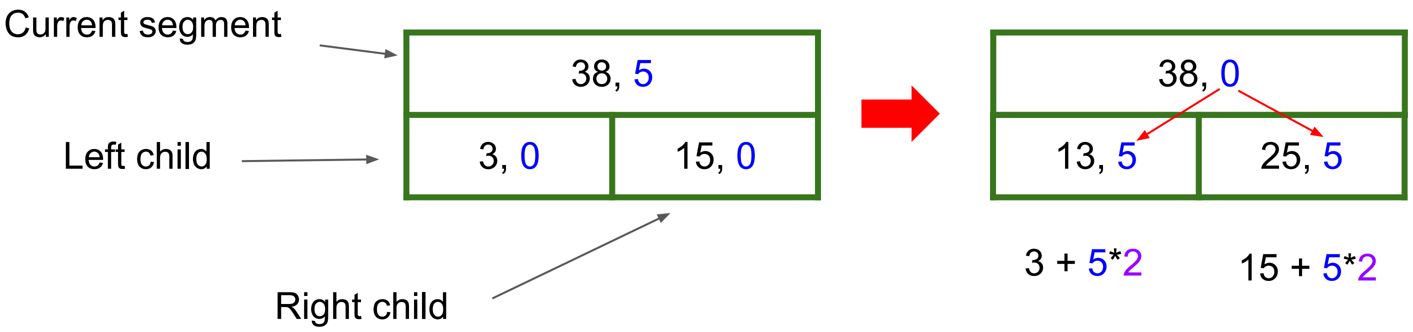
How does this work?

Simple, before doing **any** operation with a segment, we check whether it has a lazy value. If it does, we update it (which takes **O(1)** time):



How does this work?

Simple, before doing **any** operation with a segment, we check whether it has a lazy value. If it does, we update it (which takes **O(1)** time):

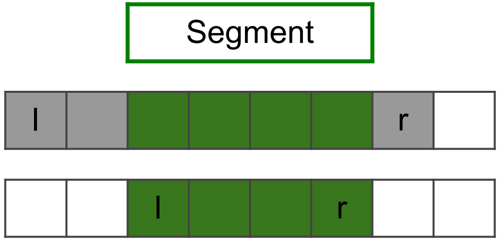


void updateLazy(segment) {

```
// This function makes sure that this segment has no lazy value.
Increase lazy value of left child by the lazy value of this segment.
Increase lazy value of right child by the lazy value of this segment.
Increase normal value of left child by lazy value of this segment * length of left child.
Increase normal value of right child by lazy value of this segment * length of right child.
Set the lazy value of this segment to 0.
```

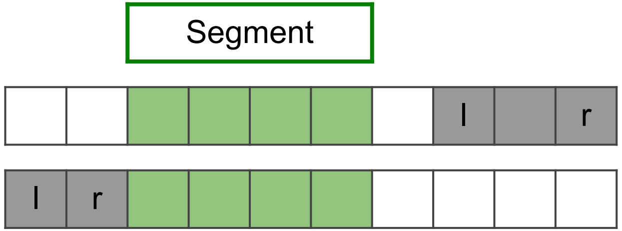
void update(segment, l, r, x) {

Segment falls entirely in range of l..r



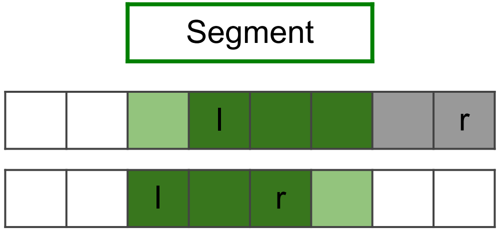
Increase the value of this segment by *x**segLength.
Increase the lazy value of this segment by *x*.

Segment falls entirely out of range of l..r



Do nothing.

Other



updateLazy(this)
update(lChild, l, r, x)
update(rChild, l, r, x)
Set the value of this segment to the sum of its two children.

Implementation

Java

```
public class LazySegmentTree {
    int[] arr; // original data
    int arrLength; // length of the original data
    long[] tree; // array representation of this segment tree
    long[] lazy; // lazy values of this tree
```

Construction

```
public LazySegmentTree(int[] arr){
    this.arrLength = arr.length;
    this.arr = arr;

    int height = 32 - Integer.numberOfLeadingZeros(arrLength-1); // Log base 2 of arrLength
    int size = 1 << (height + 1); // 2 to the power of height + 1
    tree = new long[size];
    lazy = new long[size];

    construct(1, 0, arrLength);
}

private long construct(int i, int segBegin, int segEnd){
    if(segBegin + 1 == segEnd) return tree[i] = arr[segBegin]; // Segment has length of 1
    else {
        int mid = (segBegin + segEnd)/2;
        return tree[i] = construct(i*2, segBegin, mid)
                + construct(i*2+1, mid, segEnd);
    }
}
```

Lazy updates

```
private void updateLazy(int i, int segBegin, int segEnd){
    long lzyVal = lazy[i];
    lazy[i] = 0;

    lazy[i*2] += lzyVal;
    lazy[i*2+1] += lzyVal;

    int mid = (segEnd + segBegin)/2;
    tree[i*2] += lzyVal * (mid - segBegin);
    tree[i*2+1] += lzyVal * (segEnd - mid);
}
```

Queries

```
private long query(int i, int segBegin, int segEnd, int l, int r){
    if(l <= segBegin && r >= segEnd) return tree[i];
    if(r <= segBegin || l >= segEnd) return 0;

    updateLazy(i, segBegin, segEnd);
    int mid = (segBegin + segEnd)/2;
    return query(i*2, segBegin, mid, l, r) + query(i*2+1, mid, segEnd, l, r);
}

public long query(int l, int r){
    return query(1, 0, arrLength, l, r);
}
```

Range Updates

```
    // Updates the segment
    // Returns the new value of the segment
    private long update(int i, int segBegin, int segEnd, int l, int r, int amount){
        if(r <= segBegin || l >= segEnd) return tree[i]; // Segment is completely outside l..r
        if(l <= segBegin && r >= segEnd){ // Segment is fully inside l..r
            lazy[i] += amount;
            tree[i] += amount * (segEnd - segBegin);
            return tree[i];
        }

        // Segment is partially in l..r
        updateLazy(i, segBegin, segEnd);
        int mid = (segBegin + segEnd)/2;
        return tree[i] = update(i*2, segBegin, mid, l, r, amount)
                + update(i*2+1, mid, segEnd, l, r, amount);
    }

    public void update(int l, int r, int amount){
        update(1, 0, arrLength, l, r, amount);
    }
}
```

Other functions

Segment trees are not limited to just sum queries, segment trees can support an extremely wide variety of operations (which is one of the reasons they come up so often). Examples include minimum queries, maximum queries and different types of updates like setting the value of a range.

Here are some properties your function must satisfy if you want to make a segment tree for it:

- You must be able to calculate the value of a segment in O(1) time based on the values of its two children

If it is a lazy propagated segment tree:

- You must also be able to find the value of a segment that's fully inside an update range in O(1) time

Note: for more complex questions, you may need to store more than one value per segment.

Other data structures

Sparse Tables:

- Can calculate minimums/maximums super quickly
- Can’t be updates quickly

Binary Indexed Trees:

- Kinda like a lightweight segment tree
- Can only update one element at a time
- Doesn’t support as many functions as a segment tree

Other data structures

We learned about today: Prefix sum array, Segment tree, Lazy propagated segment tree

You should consider learning about later: Binary indexed tree, Sparse table

Name	Space	Construction	Query	Point update	Range update	Supported functions	Implementation
Prefix sum array	N	N	O(1)	O(N)	O(N)	Sum	Very short
Segment tree	4N	4N	O(log(N))	O(log(N)	O(M*log(N))	Very many	Medium
Lazy propagated segment tree	8N	4N	O(log(N))	O(log(N))	O(log(N))	Many	Long
Binary indexed tree	N	N*logN	O(log(N))	O(log(N))	O(M*log(N))	Sum	Short
Sparse table	N*logN	N*logN	O(1)	O(N) ?	**O(MN)** ?	Min/Max	Medium