<https://leetcode.com/problems/range-sum-query-mutable/description/>

Given an integer array nums, handle multiple queries of the following types:

* **Update** the value of an element in nums.
* Calculate the **sum** of the elements of nums between indices left and right **inclusive** where left <= right.

Implement the NumArray class:

* NumArray(int[] nums) Initializes the object with the integer array nums.
* void update(int index, int val) **Updates** the value of nums[index] to be val.
* int sumRange(int left, int right) Returns the **sum** of the elements of nums between indices left and right **inclusive** (i.e. nums[left] + nums[left + 1] + ... + nums[right]).

**Example 1:**

Input

["NumArray", "sumRange", "update", "sumRange"]

[[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]

Output

[null, 9, null, 8]

Explanation

NumArray numArray = new NumArray([1, 3, 5]);

numArray.sumRange(0, 2); // return 1 + 3 + 5 = 9

numArray.update(1, 2); // nums = [1, 2, 5]

numArray.sumRange(0, 2); // return 1 + 2 + 5 = 8

**Constraints:**

* 1 <= nums.length <= 3 \* 104
* -100 <= nums[i] <= 100
* 0 <= index < nums.length
* -100 <= val <= 100
* 0 <= left <= right < nums.length
* At most 3 \* 104 calls will be made to update and sumRange.

**Attempt 1: 2023-04-19**

**Solution 1: Segment Tree (60 min)**

**Style 1: Iterative version of Segment Trees**

class NumArray {

int[] tree;

int n;

public NumArray(int[] nums) {

n = nums.length;

tree = new int[2 \* n];

buildTree(nums);

}

private void buildTree(int[] nums) {

for(int i = n, j = 0; i < 2 \* n; i++, j++) {

tree[i] = nums[j];

}

// Note: i > 0 is strictly > not >=, because the segment

// tree root should start only on i = 1, and its easy to

// prove in turn, since tree[i] = tree[2\*i] + tree[2\*i+1],

// which means parent node equal to sum up of left and

// right child node, but if i = 0, then tree[0] = tree[0]

// + tree[1], the parent node index equal to left child node

// index, its invalid scenario.

for(int i = n - 1; i > 0; i--) {

tree[i] = tree[i \* 2] + tree[i \* 2 + 1];

}

}

public void update(int index, int val) {

index += n;

tree[index] = val;

while(index > 0) {

int left = index;

int right = index;

if(index % 2 == 0) {

right = index + 1;

} else {

left = index - 1;

}

// parent is updated after child is updated

tree[index / 2] = tree[left] + tree[right];

index /= 2;

}

}

public int sumRange(int left, int right) {

int sum = 0;

// Get leaf with value 'l'

left += n;

// Get leaf with value 'r'

right += n;

while(left <= right) {

// Check if l is right child of its parent P

if(left % 2 == 1) {

// P P, [(l + 1) / 2]

// / \ ==> / \

// x l ... r x l ... r

// l is right child of P. Then P contains sum of range of l and

// another child x which is outside the range [l,r] and we don't

// need parent P sum. Add l to sum without its parent P and set

// l to point to the right of P on the upper level.

sum += tree[left];

left++;

left /= 2;

} else {

// P P <= [l / 2]

// / \ ==> / \

// l ... r l ... r

// l is not right child of P. Then parent P contains sum of range which

// lies in [l,r]. Add P to sum which implement by setting l to point to

// the parent of P

left /= 2;

}

// Check if r is left child of its parent P

if(right % 2 == 0) {

// P [(r - 1) / 2], P

// / \ ==> / \

// l ... r x l ... r

// r is left child of P. Then P contains sum of range of r and another

// child x which is outside the range [l,r] and we don't need parent P

// sum. Add r to sum without its parent P and set r to point to the

// left of P on the upper level.

sum += tree[right];

right--;

right /= 2;

} else {

// P [r / 2] => P

// / \ ==> / \

// l ... r l ... r

// r is not left child of P. Then parent P contains sum of range which

// lies in [l,r]. Add P to sum which implement by setting r to point to

// the parent of P

right /= 2;

}

}

return sum;

}

}

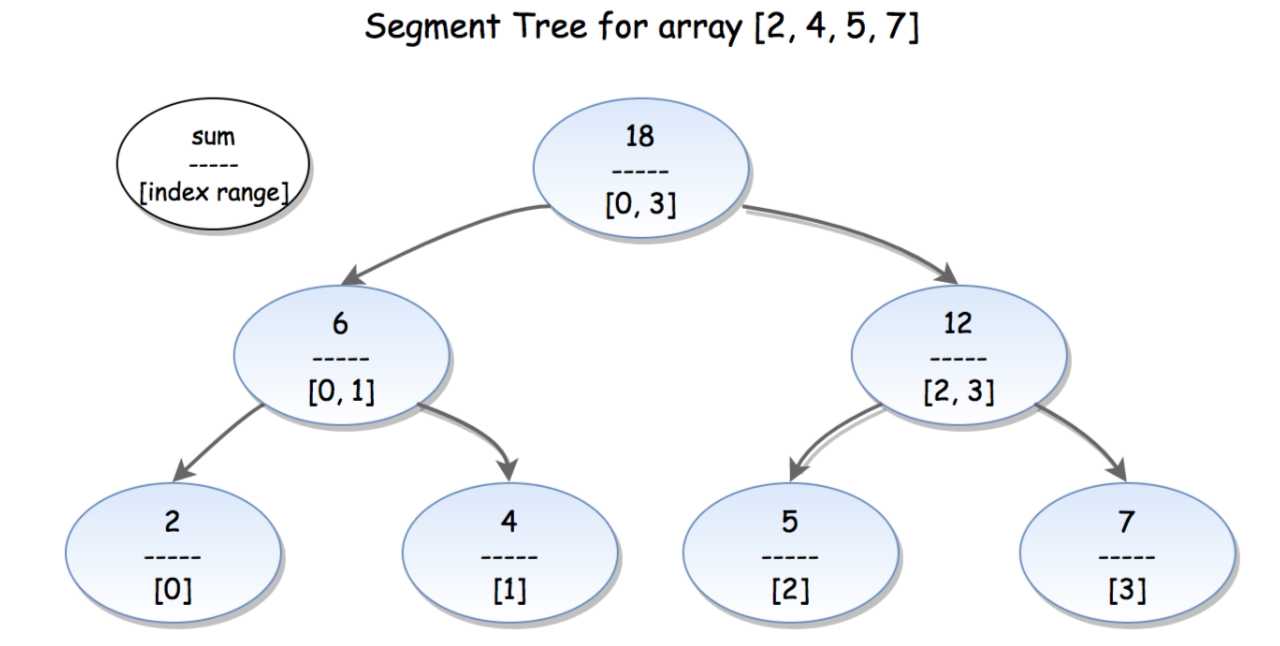
**Refer to**

<https://leetcode.com/problems/range-sum-query-mutable/editorial/>

#### **Approach 3: Iterative Segment Tree**

**Algorithm**

Segment tree is a very flexible data structure, because it is used to solve numerous range query problems like finding minimum, maximum, sum, greatest common divisor, least common denominator in array in logarithmic time.



*Figure 2. Illustration of Segment tree.*

The segment tree for array a[0,1,…,n−1] is a binary tree in which each node contains **aggregate** information (min, max, sum, etc.) for a subrange [i…j] of the array, as its left and right child hold information for range [i…(i+j)/2] and [(i+j)/2+1,j].

Segment tree could be implemented using either an array or a tree. For an array implementation, if the element at index

i is not a leaf, its left and right child are stored at index 2i and 2i+1 respectively.

In the example above (Figure 2), every leaf node contains the initial array elements {2,4,5,7}. The internal nodes contain the sum of the corresponding elements in range, *e.g.* (6) is the sum for the elements from index 0 to index 1. The root (18) is the sum of its children (6) and (12), which also holds the total sum of the entire array.

Segment Tree can be broken down to the three following steps:

1. Pre-processing step which builds the segment tree from a given array.
2. Update the segment tree when an element is modified.
3. Calculate the Range Sum Query using the segment tree.

##### **1. Build segment tree**

We will use a very effective bottom-up approach to build segment tree. We already know from the above that if some node p holds the sum of [i…j] range, its left and right children hold the sum for range [i…(i+j)/2] and [(i+j)/2+1,j] respectively.

Therefore to find the sum of node p, we need to calculate the sum of its right and left child in advance.

We begin from the leaves, initialize them with input array elements a[0,1,…,n−1]. Then we move upward to the higher level to calculate the parents' sum till we get to the root of the segment tree.

int[] tree;

int n;

public NumArray(int[] nums) {

if (nums.length > 0) {

n = nums.length;

tree = new int[n \* 2];

buildTree(nums);

}

}

private void buildTree(int[] nums) {

for (int i = n, j = 0; i < 2 \* n; i++, j++)

tree[i] = nums[j];

for (int i = n - 1; i > 0; --i)

tree[i] = tree[i \* 2] + tree[i \* 2 + 1];

}

**Complexity Analysis**

* Time complexity : O(n), because we calculate the sum of one node during each iteration of the for loop. There are approximately 2n nodes in a segment tree.
* This could be proved in the following way: Segmented tree for array with n elements has n leaves (the array elements itself). The number of nodes in each level is half the number in the level below.
* So if we sum the number by level we will get: n+n/2+n/4+n/8+…+1≈2n
* Space complexity : O(n). We used 2n extra space to store the segment tree.

##### **2. Update segment tree**

When we update the array at some index i we need to rebuild the segment tree, because there are tree nodes which contain the sum of the modified element. Again we will use a bottom-up approach. We update the leaf node that stores a[i]. From there we will follow the path up to the root updating the value of each parent as a sum of its children values.

void update(int pos, int val) {

pos += n;

tree[pos] = val;

while (pos > 0) {

int left = pos;

int right = pos;

if (pos % 2 == 0) {

right = pos + 1;

} else {

left = pos - 1;

}

// parent is updated after child is updated

tree[pos / 2] = tree[left] + tree[right];

pos /= 2;

}

}

**Complexity Analysis**

* Time complexity : O(log⁡n).
* Algorithm has O(log⁡n) time complexity, because there are a few tree nodes with range that include i th array element, one on each level. There are log⁡(n) levels.
* Space complexity : O(1).

##### **3. Range Sum Query**

We can find range sum query [L,R] using segment tree in the following way:

Algorithm hold loop invariant:

l≤r and sum of [L…l] and [r…R] has been calculated, where l and r are the left and right boundary of calculated sum. Initially we set l with left leaf L and r with right leaf R. Range [l,r] shrinks on each iteration till range borders meets after approximately log⁡n iterations of the algorithm

Loop till l≤r

Check if l is right child of its parent P

l is right child of P. Then P contains sum of range of l and another child which is outside the range [l,r] and we don't need parent P sum. Add l to sum without its parent P and set l to point to the right of P on the upper level.

l is not right child of P . Then parent P contains sum of range which lies in [l,r]. Add P to sum and set l to point to the parent of P

Check if r is left child of its parent P

r is left child of P. Then P contains sum of range of r and another child which is outside the range [l,r] and we don't need parent P sum. Add r to sum without its parent P and set r to point to the left of P on the upper level.

r is not left child of P. Then parent P contains sum of range which lies in [l,r]. Add P to sum and set r to point to the parent of P

public int sumRange(int l, int r) {

// get leaf with value 'l'

l += n;

// get leaf with value 'r'

r += n;

int sum = 0;

while (l <= r) {

if ((l % 2) == 1) {

sum += tree[l];

l++;

}

if ((r % 2) == 0) {

sum += tree[r];

r--;

}

l /= 2;

r /= 2;

}

return sum;

}

**Complexity Analysis**

* Time complexity : O(log⁡n)
* Time complexity is O(log⁡n) because on each iteration of the algorithm we move one level up, either to the parent of the current node or to the next sibling of parent to the left or right direction till the two boundaries meet. In the worst-case scenario this happens at the root after log⁡n iterations of the algorithm.
* Space complexity : O(1).

## **Further Thoughts**

The iterative version of Segment Trees was introduced in this article. A more intuitive, recursive version of Segment Trees to solve this problem is discussed [here](https://leetcode.com/articles/a-recursive-approach-to-segment-trees-range-sum-queries-lazy-propagation/). The concept of Lazy Propagation is also introduced there.

There is an alternative solution of the problem using Binary Indexed Tree. It is faster and simpler to code. You can find it [here](https://leetcode.com/problems/range-sum-query-mutable/discuss/75753/Java-using-Binary-Indexed-Tree-with-clear-explanation).

**Style 2: Recursive version of Segment Trees**

class NumArray {

int[] tree;

int n;

public NumArray(int[] nums) {

n = nums.length;

// Why not n \* 2 but n \* 4 ?

// https://stackoverflow.com/questions/28470692/how-is-the-memory-of-the-array-of-segment-tree-2-2-ceillogn-1

tree = new int[4 \* n];

// Call this method as buildTree(nums, 0, 0, n-1),

// Here nums[] is input array and n is its size

buildTree(nums, 0, 0, n - 1);

}

// Pending:

// 1. Modify to root.left = ... root.right = ... Divide and Conquer way which has return type

// 2. Start with index = 1, not index = 0, then child = 2 \* i & 2 \* i + 1

// --------------------------------------------------------------------------------

// The build tree process is quite different than iterative segment tree, which

// directly assign original array into tree as leaves, indexes between [n, 2n - 1],

// in recursive solution, the leaves node not directly locate between [n, 2n - 1],

// instead follow 2 \* treeIndex + 1 or 2 \* treeIndex + 2 rule level by level

// until the leaves level reached, that's also why instead only requires 2 \* n

// length for iterative segement tree, in recursive segement tree requires 4 \* n

private void buildTree(int[] nums, int treeIndex, int lo, int hi) {

if(lo == hi) {

tree[treeIndex] = nums[lo];

return;

}

int mid = lo + (hi - lo) / 2;

buildTree(nums, 2 \* treeIndex + 1, lo, mid);

buildTree(nums, 2 \* treeIndex + 2, mid + 1, hi);

tree[treeIndex] = tree[2 \* treeIndex + 1] + tree[2 \* treeIndex + 2];

}

public void update(int index, int val) {

// Call this method as updateHelper(0, 0, n-1, i, val),

// here you want to update the value at index with value val.

updateHelper(0, 0, n - 1, index, val);

}

private void updateHelper(int treeIndex, int lo, int hi, int index, int val) {

// Leaf node, update element

if(lo == hi) {

tree[treeIndex] = val;

return;

}

// Recurse deeper for appropriate child

int mid = lo + (hi - lo) / 2;

if(index > mid) {

updateHelper(2 \* treeIndex + 2, mid + 1, hi, index, val);

} else {

updateHelper(2 \* treeIndex + 1, lo, mid, index, val);

}

// Merge updates

tree[treeIndex] = tree[2 \* treeIndex + 1] + tree[2 \* treeIndex + 2];

}

public int sumRange(int left, int right) {

// Call this method as querySegTree(0, 0, n-1, left, right),

// here [left, right] is the range/interval you are querying.

// This method relies on "null" nodes being equivalent to storing zero.

return sumRangeHelper(0, 0, n - 1, left, right);

}

// Query for arr[left ... right]

// Note: the coordinate as [left, right] is the basement, which is part of

// segment tree, the variable keep on changing and should compare with the

// [left, right] range as boundary judgement is [lo, hi]

private int sumRangeHelper(int treeIndex, int lo, int hi, int left, int right) {

// Segment completely outside range, represents a null node

// Basement: --- left --- right --- OR --- left --- right ---

// Variable: ---------------------- lo ------ hi ----------------------

if(lo > right || hi < left) {

return 0;

}

// Segment completely inside range

// Basement: --- left ------------ right ---

// Variable: ---------------------- lo ------ hi ----------------------

if(lo >= left && hi <= right) {

return tree[treeIndex];

}

// Partial overlap of current segment and queried range. Recurse deeper

int mid = lo + (hi - lo) / 2;

if(left > mid) {

return sumRangeHelper(treeIndex \* 2 + 2, mid + 1, hi, left, right);

} else if(right <= mid) {

return sumRangeHelper(treeIndex \* 2 + 1, lo, mid, left, right);

} else {

int leftSum = sumRangeHelper(treeIndex \* 2 + 1, lo, mid, left, mid);

int rightSum = sumRangeHelper(treeIndex \* 2 + 2, mid + 1, hi, mid + 1, right);

return leftSum + rightSum;

}

}

}

**Refer to**

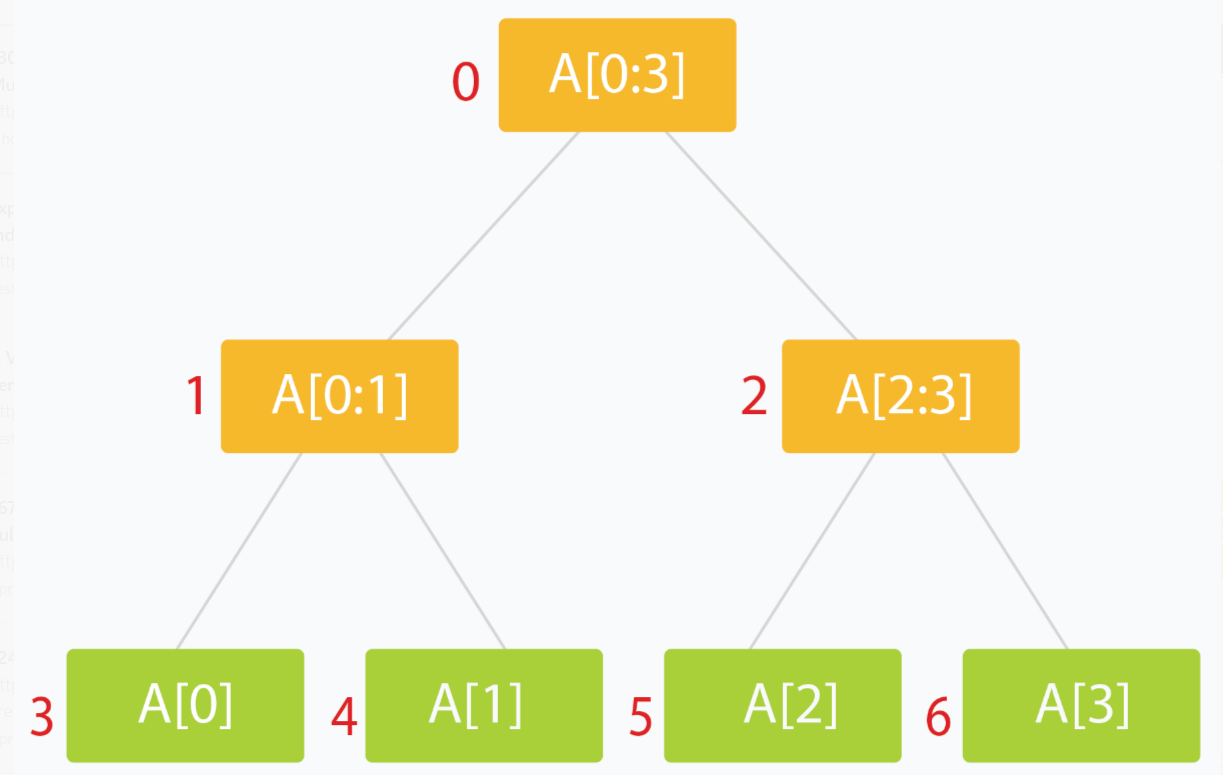
<https://leetcode.com/articles/a-recursive-approach-to-segment-trees-range-sum-queries-lazy-propagation/>

## Recursive Approach to Segment Trees

### Brief Introduction

#### What is a Segment Tree?

A segment tree is a binary tree where each node represents an interval. Generally a node would store one or more properties of an interval which can be queried later.

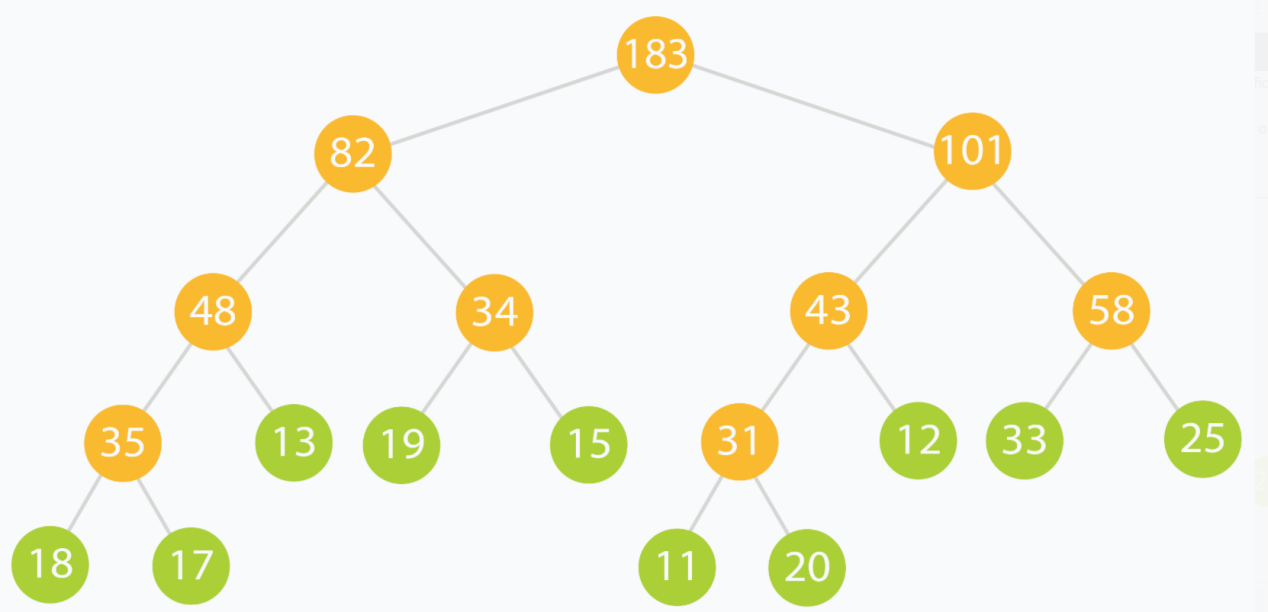


#### Why do we require it? (or What's the point of this?)

Many problems require that we give results based on query over a range or segment of available data. This can be a tedious and slow process, especially if the number of queries is large and repetitive. A segment tree let's us process such queries efficiently in logarithmic order of time.

Segment Trees have applications in areas of computational geometry and [geographic information systems.](https://en.wikipedia.org/wiki/Geographic_information_systems) For example, we may have a large number of points in space at certain distances from a central reference/origin point. Suppose we have to lookup the points which are in a certain range of distances from our origin. An ordinary lookup table would require a linear scan over all the possible points or all possible distances (think hash-maps). Segment Trees lets us achieve this in logarithmic time with much less space cost. Such a problem is called [Planar Range Searching](https://en.wikipedia.org/wiki/Range_searching). Solving such problems efficiently is critical, especially when dealing with dynamic data which changes fast and unpredictably (for example, a radar system for air traffic.)

We will solve the [Range Sum Query problem](https://leetcode.com/articles/a-recursive-approach-to-segment-trees-range-sum-queries-lazy-propagation/" \l "range-sum-queries) later in this editorial as an example of how Segment Trees help us save loads on runtime costs.



We will use the above tree as a practical example of what a Range Sum Query segment tree looks and behaves like.

#### How do we make one?

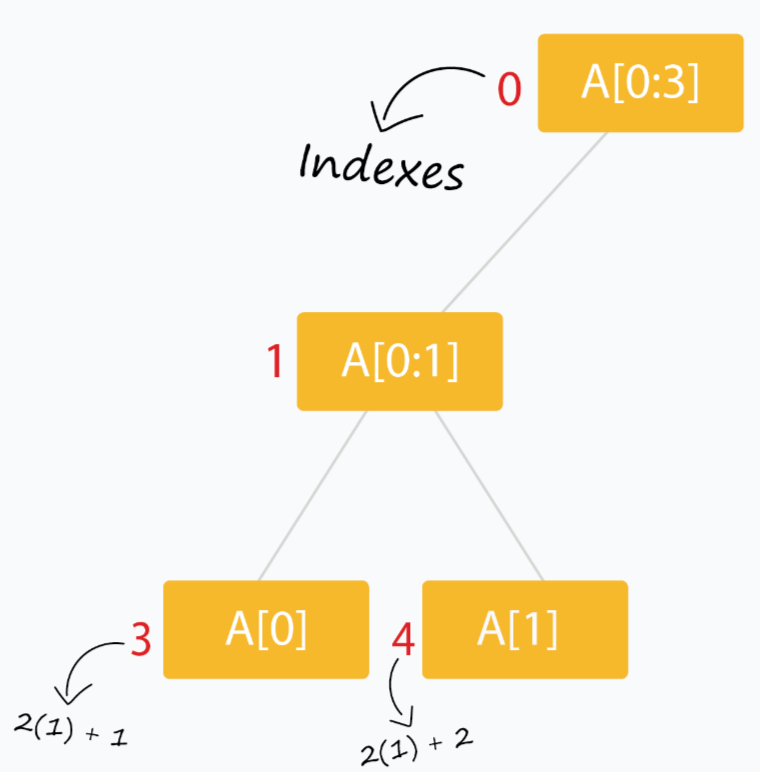
Let our data be in an array arr[] of size *n*.

1. The root of our segment tree typically represents the entire interval of data we are interested in. This would be arr[0:n-1].
2. Each leaf of the tree represents a range comprising of just a single element. Thus the leaves represent arr[0], arr[1]and so on till arr[n-1].
3. The internal nodes of the tree would represent the **merged** or **union** result of their children nodes.
4. Each of the children nodes could represent approximately half of the range represented by their parent.

A segment tree for an *n* element range can be comfortably represented using an array of size ≈ 4∗*n*. ([Stack Overflow](http://stackoverflow.com/q/28470692/2844164) has a good discussion as to why. If you are not convinced, fret not. We will discuss it later on.)

But how?

The idea is simple: A node at index *i* can have two children at indexes (2∗i +1) and (2∗i+2).



  Segment trees are very intuitive and easy to use when built recursively.

### Recursive methods for Segment Trees

We will use the array tree[]to store the nodes of our segment tree (initialized to all zeros). The following scheme (0- based indexing) is used:

* The node of the tree is at index 00. Thus tree[0]is the root of our tree.
* The children of tree[i] are stored at tree[2\*i+1]and tree[2\*i+2].
* We will pad our arr[] with extra 0or null values so that *n* = 2^*k* (where *n* is the final length of arr[] and *k* is a non negative integer.)

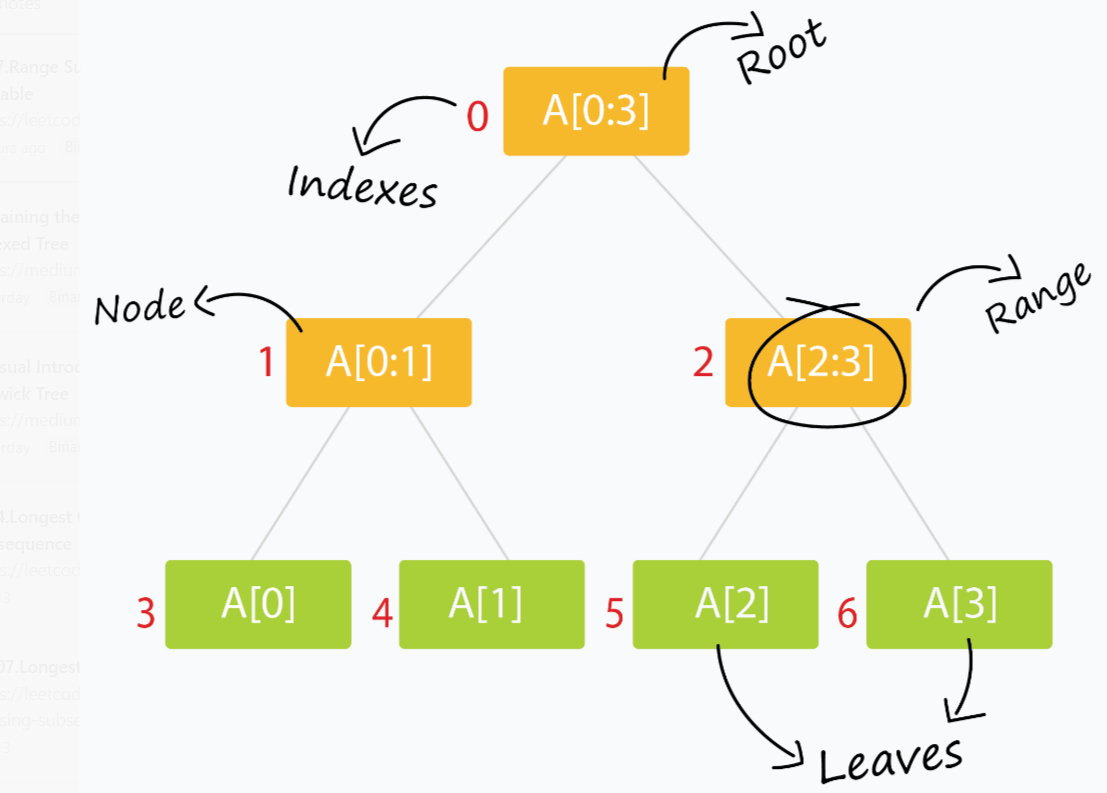
Do we actually need to pad `arr[]` with zeros?

No, not really. Just ensure that `tree[]` is large enough and always zero-initialized and you don't need to worry about extra leaf nodes not being processed.

* The leaves of the tree occur at indexes 2^*k*−1 to 2^(k + 1) - 2.

What if we started by storing the node of tree at index 1 \*instead of\* index 0 ? How would the positions of related nodes change?

For starters, the children for node at index *i* will \*\*now\*\* be located at indexes (2∗*i*) and (2∗*i*+1). Can you determine which indexes the leaves will be stored at?



We also require only three kinds of methods:

#### 1. Build the tree from the original data.

void buildSegTree(vector<int>& arr, int treeIndex, int lo, int hi)

{

if (lo == hi) { // leaf node. store value in node.

tree[treeIndex] = arr[lo];

return;

}

int mid = lo + (hi - lo) / 2; // recurse deeper for children.

buildSegTree(arr, 2 \* treeIndex + 1, lo, mid);

buildSegTree(arr, 2 \* treeIndex + 2, mid + 1, hi);

// merge build results

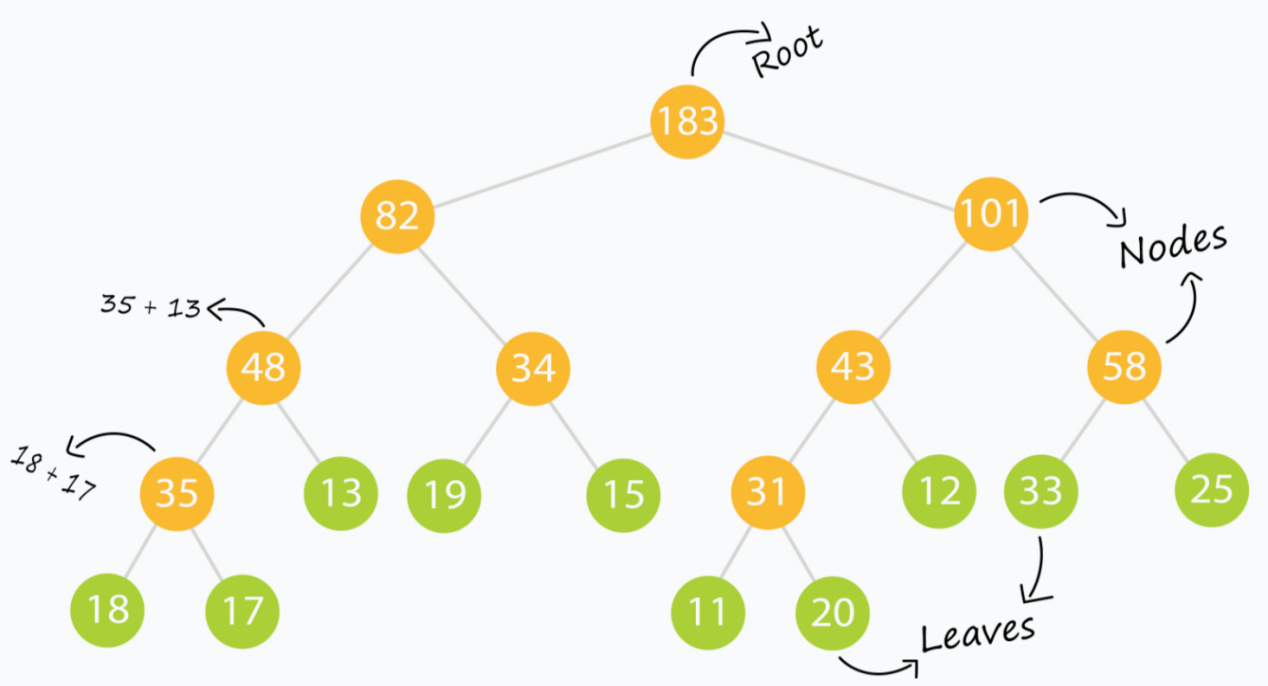
tree[treeIndex] = merge(tree[2 \* treeIndex + 1], tree[2 \* treeIndex + 2]);

}

// call this method as buildSegTree(arr, 0, 0, n-1);

// Here arr[] is input array and n is its size.

The method builds the entire `tree` in a bottom up fashion. When the condition *lo*=*hi* is satisfied, we are left with a range comprising of just a single element (which happens to be `arr[lo]`). This constitutes a leaf of the tree. The rest of the nodes are built by merging the results of their two children. `treeIndex` is the index of the current node of the segment tree which is being processed.



For example, the tree above is made from the input array: (which we will use throughout this tutorial)

arr[] = { 18, 17, 13, 19, 15, 11, 20, 12, 33, 25 };

Can you guess what the `merge` operation is in this example? After building the tree, the `tree[]` array looks like:

tree[] = { 183, 82, 101, 48, 34, 43, 58, 35, 13, 19, 15, 31, 12, 33, 25, 18, 17, 0, 0, 0, 0, 0, 0, 11, 20, 0, 0, 0, 0, 0, 0 };

Notice the the groups of zeros near the end of the `tree[]` array? Those are `null` values we used as padding to ensure a complete binary tree is formed (since we only had 10 leaf elements. Had we had, say, 16 leaf elements, we wouldn't need any `null` elements. Can you prove why?)

**NOTE:** The merge operation varies from problem to problem. You should closely think of what to store in a node of the segment tree and how two nodes will merge to provide a result before you even start building a segment tree.

#### 2. Read/Query on an interval or segment of the data.

int querySegTree(int treeIndex, int lo, int hi, int i, int j)

{

// query for arr[i..j]

if (lo > j || hi < i) // segment completely outside range

return 0; // represents a null node

if (i <= lo && j >= hi) // segment completely inside range

return tree[treeIndex];

int mid = lo + (hi - lo) / 2; // partial overlap of current segment and queried range. Recurse deeper.

if (i > mid)

return querySegTree(2 \* treeIndex + 2, mid + 1, hi, i, j);

else if (j <= mid)

return querySegTree(2 \* treeIndex + 1, lo, mid, i, j);

int leftQuery = querySegTree(2 \* treeIndex + 1, lo, mid, i, mid);

int rightQuery = querySegTree(2 \* treeIndex + 2, mid + 1, hi, mid + 1, j);

// merge query results

return merge(leftQuery, rightQuery);

}

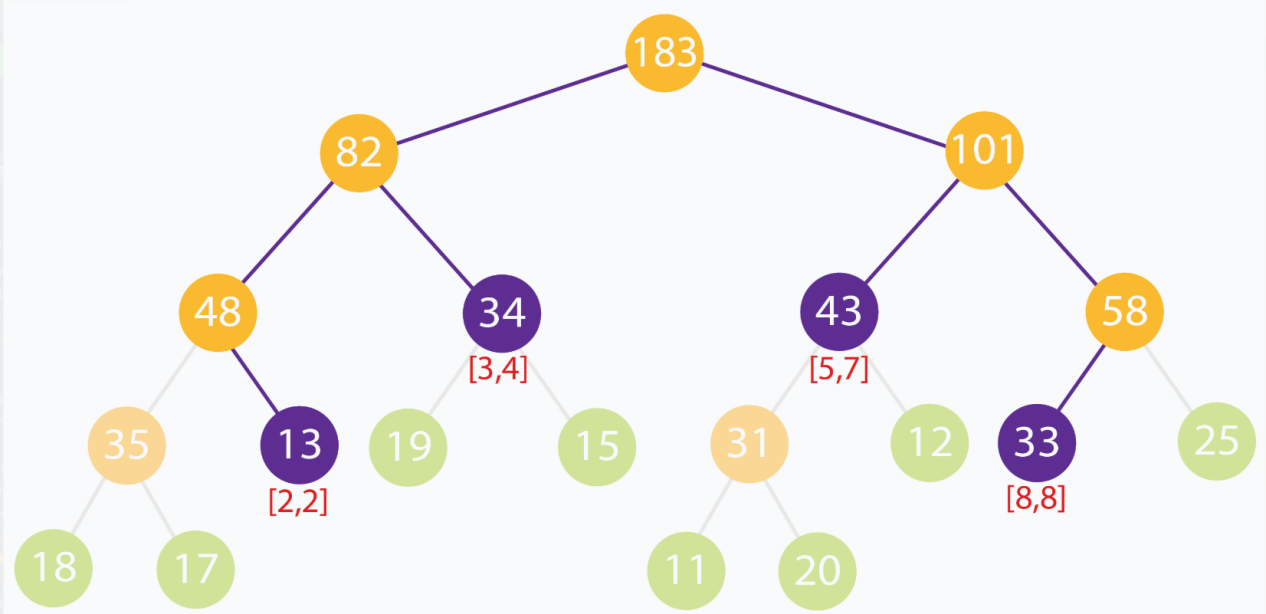
// call this method as querySegTree(0, 0, n-1, i, j);

// Here [i,j] is the range/interval you are querying.

// This method relies on "null" nodes being equivalent to storing zero.

The method returns a result when the queried range matches exactly with the range represented by a current node. Else it digs deeper into the tree to find nodes which match a portion of the node exactly.

This is where the beauty of the segment tree lies.



n the above example, we are trying to find the sum of the elements in the range [2,8]. No segment completely represents the range [2,8]. However we can see that [2,8] can be built up using the ranges [2,2],[3,4],[5,7] and [8,8]. As a quick verification, we can see that sum of input elements at indexes [2,8] is 13+19+15+11+20+12+33=123. The sum of node values for the nodes representing ranges [2,2],[3,4],[5,7] and [8,8] are 13+34+43+33=123.

#### 3. Update the value of an element.

void updateValSegTree(int treeIndex, int lo, int hi, int arrIndex, int val)

{

if (lo == hi) { // leaf node. update element.

tree[treeIndex] = val;

return;

}

int mid = lo + (hi - lo) / 2; // recurse deeper for appropriate child

if (arrIndex > mid)

updateValSegTree(2 \* treeIndex + 2, mid + 1, hi, arrIndex, val);

else if (arrIndex <= mid)

updateValSegTree(2 \* treeIndex + 1, lo, mid, arrIndex, val);

// merge updates

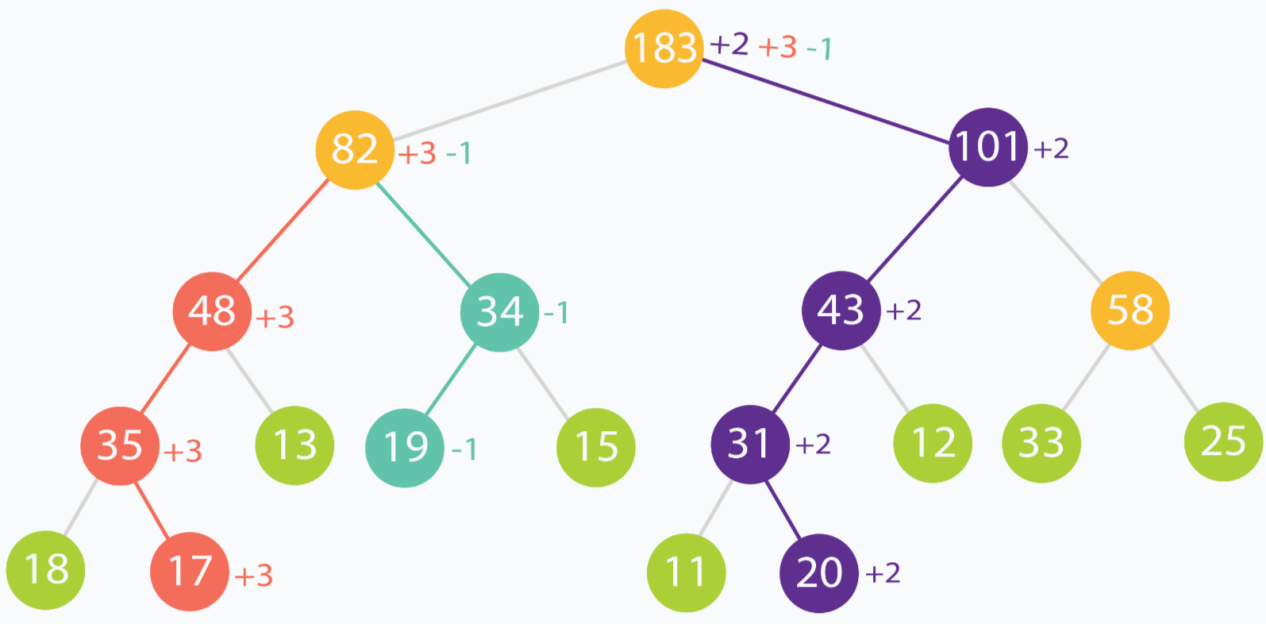
tree[treeIndex] = merge(tree[2 \* treeIndex + 1], tree[2 \* treeIndex + 2]);

}

// call this method as updateValSegTree(0, 0, n-1, i, val);

// Here you want to update the value at index i with value val.

This is similar to `buildSegTree`. We update the value of the leaf node of our tree which corresponds to the updated element. Later the changes are propagated through the upper levels of the tree straight to the root.



In this example, element at indexes (in original input data) 1,3 and 6 are incremented by +3,−1 and +2 respectively. You can see how the changes propagate up the tree, all through to the root.

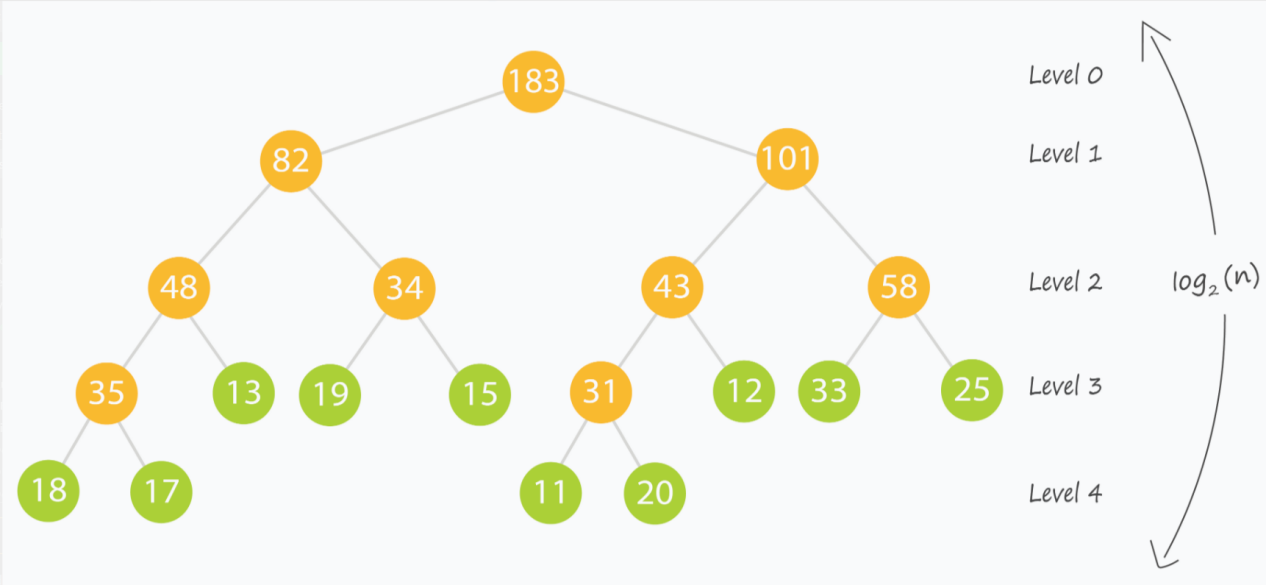
### Complexity Analysis

Let's take a look at the build process. We visit each leaf of the segment tree (corresponding to each element in our array

arr[]). That makes *n* leaves. Also there will be *n*−1 internal nodes. So we process about 2∗*n* nodes. This makes the build process run in*O*(*n*) linear complexity.

The update process discards half of the range for every level of recursion to reach the appropriate leaf in the tree. This is similar to binary search and takes logarithmic time. After the leaf is updated, its direct ancestors at each level of the tree are updated. This takes time linear to height of the tree.

The read/query process traverses depth-first through the tree looking for node(s) that match exactly with the queried range. At best, we query for the entire range and get our result from the root of the segment tree itself. At worst, we query for a interval/range of size 1(which corresponds to a single element), and we end up traversing through the height of the tree. This takes time linear to height of the tree.



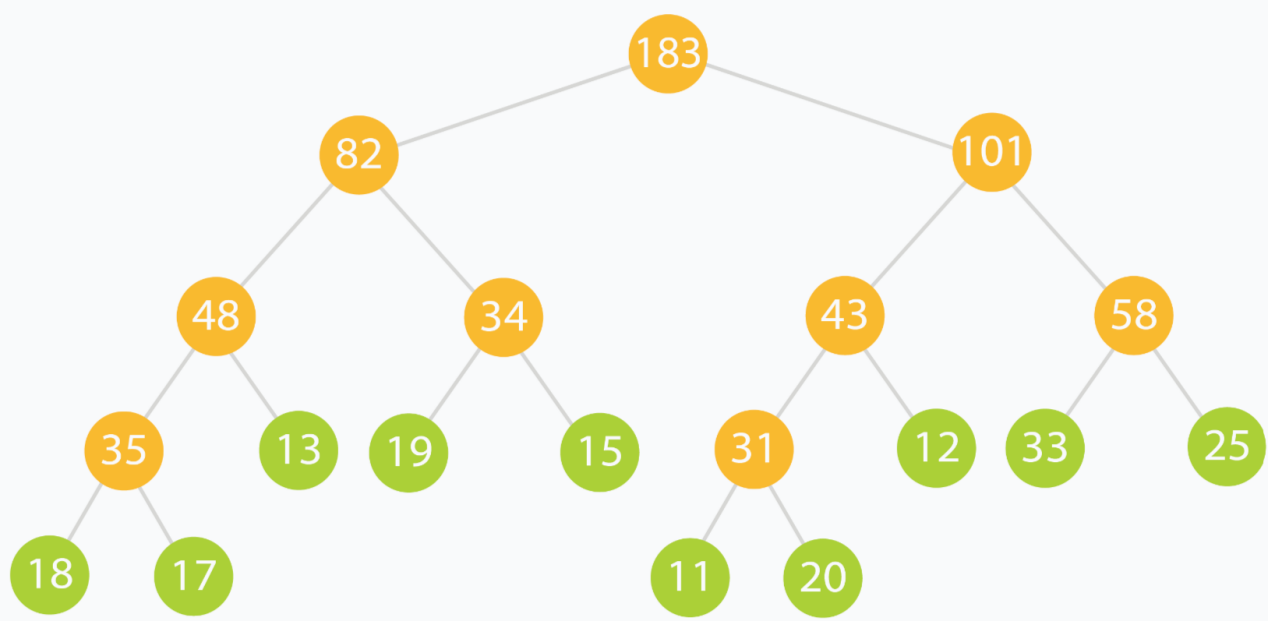
This is the time to revisit something said before:

A segment tree for an n element range can be comfortably represented using an array of size ≈ 4∗n

This ensures that we build our segment tree as a complete binary tree, which in turn ensures that the height of the tree is upper-bounded by the logarithm of the size of our input.

**Voila!** Both the read and update queries now take logarithmic *O*(*log*2(*n*))time, which is what we desired.

## Range Sum Queries



The **Range Sum Query** problem is a subset of the [Range Query](https://en.wikipedia.org/wiki/Range_query_(data_structures)) class of problems. Given an array or sequence of data elements, one is required to process read and update queries which consist of ranges of elements. Segment Trees (along with other Interval-based data structures like the **Binary Indexed Tree** (a.k.a. **Fenwick Tree**)) are used to solve this class of problems reasonably fast for practical usage.

The **Range Sum Query** problem specifically deals with the sum of elements in the queried range. Many variations of the problem exist, including for [immutable data](https://leetcode.com/problems/range-sum-query-immutable/), [mutable data](https://leetcode.com/problems/range-sum-query-mutable/), [multiple updates, single query](https://leetcode.com/problems/range-addition/) and [multiple updates, multiple queries (each being very costly in terms of computation).](https://leetcode.com/problems/range-sum-query-2d-mutable/)

A [sample solution](https://gist.github.com/babhishek21/7198fcaabc9d61d171d8f418139c2372) solves the [Range Sum Query problem for mutable arrays](https://leetcode.com/problems/range-sum-query-mutable/) *efficiently* through the use of a recursive segment tree (pretty much like the one we just discussed.) The merge operation in this case is simply taking the sum of the two nodes (since each node stores the sum of the range it represents.)

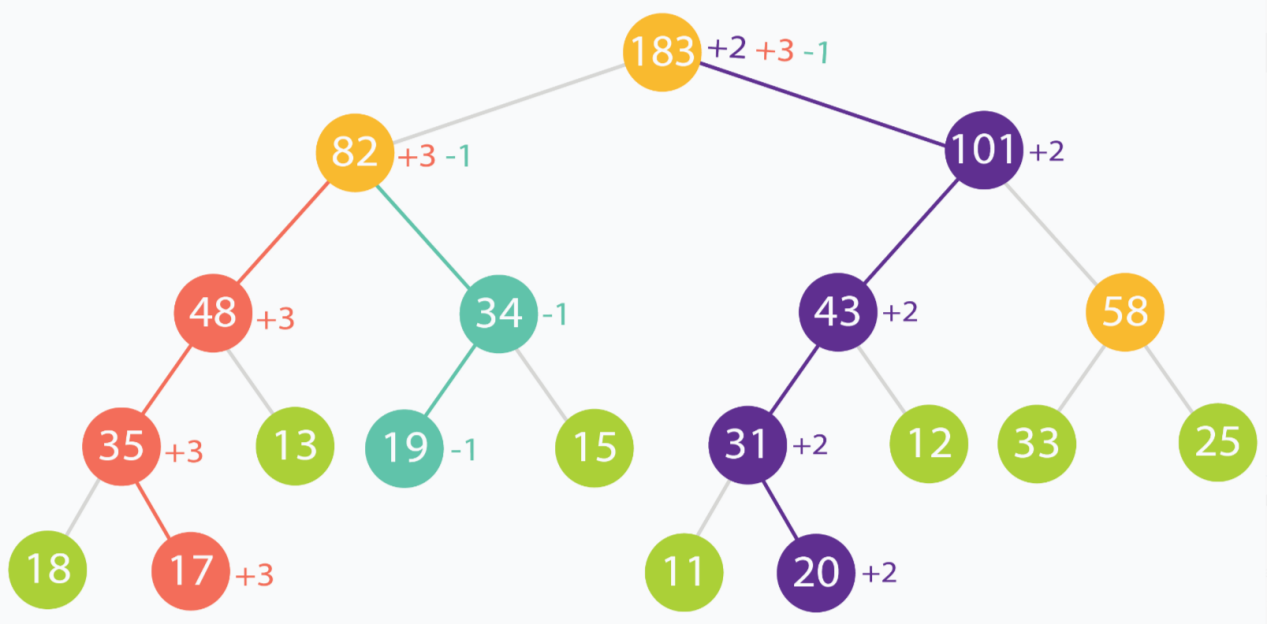
## Lazy Propagation

### Motivation

Till now we have been updating single elements only. That happens in logarithmic time and it's pretty efficient.

But what if we had to update a *range* of elements? By our current method, each of the elements would have to be updated independently, each incurring some run time cost.

The construction of a tree poses another issue called *ancestral locality*. Ancestors of adjacent leaves are guaranteed to be common at some levels of the tree. Updating each of these leaves individually would mean that we process their common ancestors multiple times. What if we could reduce this repetitive computation?



In the above example, the root is updated thrice and the node numbered 82 is updated twice. This is because, at some level of the tree, the changes propagated from different leaves will meet.

A third kind of problem is when queried ranges do not contain frequently updated elements. We might be wasting valuable time updating nodes which are rarely going to be accessed/read.

Using **Lazy Propagation** allows us to overcome all of these problems by reducing wasteful computations and processing nodes *on-demand.*

### How do we use it?

As the name suggests, we update nodes lazily. In short, we try to postpone updating descendants of a node, until the descendants themselves need to be accessed.

For the purpose of applying it to the [Range Sum Query problem](https://leetcode.com/problems/range-sum-query-mutable/), we assume that the update operation on a range, increments each element in the range by some amount val.

We use another array lazy[] which is the same size as our segment tree array tree[] to represent a lazy node. lazy[i] holds the amount by which the node tree[i] needs to be incremented, when that node is finally accessed or queried. When lazy[i] is zero, it means that node tree[i] is not lazy and has no pending updates.

#### 1. Updating a range lazily

This is a three step process:

1. Normalize the current node. This is done by removing laziness. We simple increment the current node by appropriate amount to remove it's laziness. Then we mark its children to be lazy as the descendants haven't been processed yet.
2. Apply the current update operation to the current node if the current segment lies inside the update range.
3. Recurse for the children as you would normally to find appropriate segments to update.

void updateLazySegTree(int treeIndex, int lo, int hi, int i, int j, int val)

{

if (lazy[treeIndex] != 0) { // this node is lazy

tree[treeIndex] += (hi - lo + 1) \* lazy[treeIndex]; // normalize current node by removing laziness

if (lo != hi) { // update lazy[] for children nodes

lazy[2 \* treeIndex + 1] += lazy[treeIndex];

lazy[2 \* treeIndex + 2] += lazy[treeIndex];

}

lazy[treeIndex] = 0; // current node processed. No longer lazy

}

if (lo > hi || lo > j || hi < i)

return; // out of range. escape.

if (i <= lo && hi <= j) { // segment is fully within update range

tree[treeIndex] += (hi - lo + 1) \* val; // update segment

if (lo != hi) { // update lazy[] for children

lazy[2 \* treeIndex + 1] += val;

lazy[2 \* treeIndex + 2] += val;

}

return;

}

int mid = lo + (hi - lo) / 2; // recurse deeper for appropriate child

updateLazySegTree(2 \* treeIndex + 1, lo, mid, i, j, val);

updateLazySegTree(2 \* treeIndex + 2, mid + 1, hi, i, j, val);

// merge updates

tree[treeIndex] = tree[2 \* treeIndex + 1] + tree[2 \* treeIndex + 2];

}

// call this method as updateLazySegTree(0, 0, n-1, i, j, val);

// Here you want to update the range [i, j] with value val.

#### 2. Querying a lazily propagated tree

This is a two step process:

* Normalize the current node by removing laziness. This step is the same as the update step.
* Recurse for the children as you would normally to find appropriate segments which fit in queried range.

int queryLazySegTree(int treeIndex, int lo, int hi, int i, int j)

{

// query for arr[i..j]

if (lo > j || hi < i) // segment completely outside range

return 0; // represents a null node

if (lazy[treeIndex] != 0) { // this node is lazy

tree[treeIndex] += (hi - lo + 1) \* lazy[treeIndex]; // normalize current node by removing laziness

if (lo != hi) { // update lazy[] for children nodes

lazy[2 \* treeIndex + 1] += lazy[treeIndex];

lazy[2 \* treeIndex + 2] += lazy[treeIndex];

}

lazy[treeIndex] = 0; // current node processed. No longer lazy

}

if (i <= lo && j >= hi) // segment completely inside range

return tree[treeIndex];

int mid = lo + (hi - lo) / 2; // partial overlap of current segment and queried range. Recurse deeper.

if (i > mid)

return queryLazySegTree(2 \* treeIndex + 2, mid + 1, hi, i, j);

else if (j <= mid)

return queryLazySegTree(2 \* treeIndex + 1, lo, mid, i, j);

int leftQuery = queryLazySegTree(2 \* treeIndex + 1, lo, mid, i, mid);

int rightQuery = queryLazySegTree(2 \* treeIndex + 2, mid + 1, hi, mid + 1, j);

// merge query results

return leftQuery + rightQuery;

}

// call this method as queryLazySegTree(0, 0, n-1, i, j);

// Here [i,j] is the range/interval you are querying.

// This method relies on "null" nodes being equivalent to storing zero.

**NOTE:** The following lines:

tree[treeIndex] += (hi - lo + 1) \* lazy[treeIndex]; // normalize current node by removing laziness

// and

tree[treeIndex] += (hi - lo + 1) \* val; // update segment

// and

tree[treeIndex] = tree[2 \* treeIndex + 1] + tree[2 \* treeIndex + 2]; // merge updates

are specific to the [Range Sum Query problem.](<https://leetcode.com/problems/range-sum-query-mutable/>) Different problems may have different updating and merging schemes. In this case, updates are increments of +*val* and nodes contain the sum of the elements of range/segment they represent.

## Bonus

* A quick cheat-sheet I recommend is located [here.](http://maratona.ic.unicamp.br/MaratonaVerao2016/material/segment_tree_lecture.pdf)
* [VisuAlgo](http://visualgo.net/) has an awesome visualizer for segment trees [here](http://visualgo.net/segmenttree). Put in the example data in this tutorial to see the segment tree operations in action!

**Refer to**

<https://leetcode.com/problems/range-sum-query-mutable/solutions/1281195/clean-solution-w-explanation-segment-tree-beats-100/>

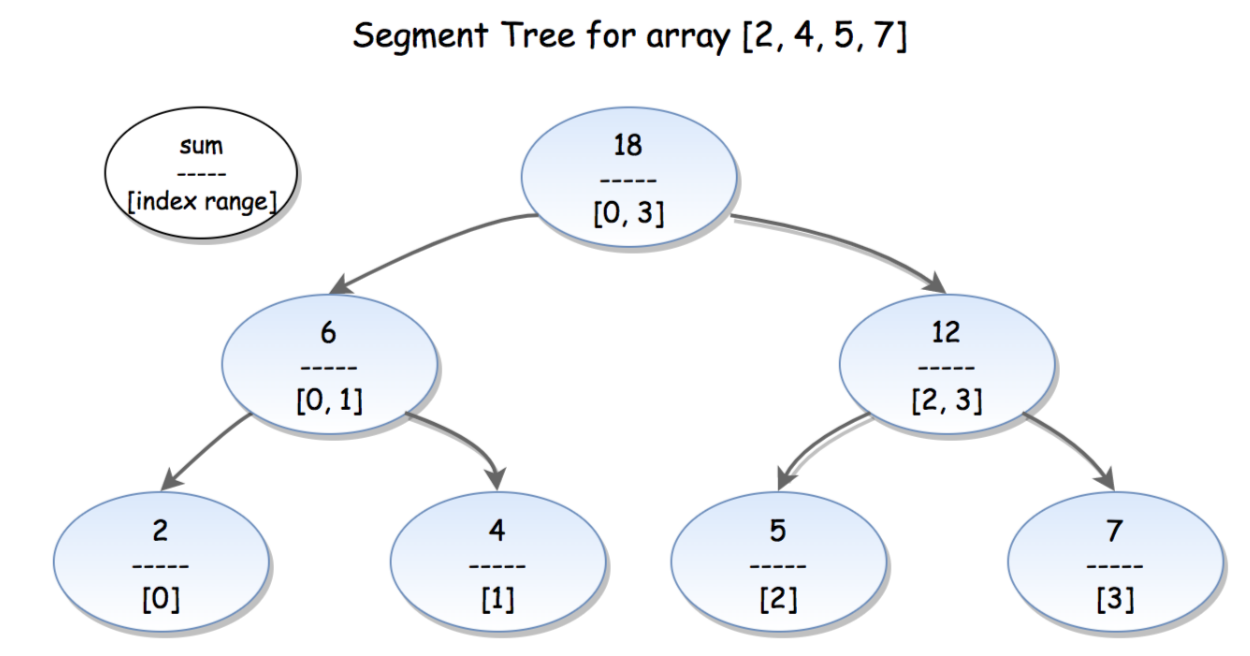
<https://leetcode.com/problems/range-sum-query-mutable/solutions/75724/17-ms-java-solution-with-segment-tree/>

The [303. Range Sum Query - Immutable](https://leetcode.com/problems/range-sum-query-immutable/) could be solved using simple prefix sum approach because the array was fixed. Here the array can be dynamically updated and thus using that approach, we can either make update or sumRange as O(1) and the other remains O(N). This will lead to TLE.

For efficient updates and sum queries, there are many approaches like square root decomposition, binary-index tree and segment trees. Here, I will explain the segment tree approach

✔️ ***Solution - I (Segment Trees using Array)***

Range Sum Query is a very common application for Segment Tree (referred henceforth as **ST**). ST consists of three functions, namely *build()*, *update()* and *query()* It allows us O(logn) update and query time and O(N) time to build the tree.



*Borrowing Segment Tree image from LC for illustration*

**build**:

Here, we will recursively build the ST for the given array.

*segTree[2\*i]* and *segTree[2\*i+1]* will denote the left child and right child respectively for the node rooted at ith index in *segTree*. By definition, *segTree[i] = segTree[2\*i] + segTree[2\*i + 1]*

The build function will be recursively called to split the given array into two halves (**Divide**). When we reach a segment consisting of single array element, it can't be split further and so we stop here and assign the value to corresponding segTree index. Finally, the parent nodes value will be constructed from from their child value (**Merge**).

**update**:

The update operation is simple. We traverse down to the node holding the segment [i] and update each node on our way up.

If the index i to be updated is less than **mid** of the current segment, then recurse the left child, otherwise recurse the right child.

When we reach the node holding segment [i], update it's value to new\_val.

Finally, on our way back up, we update all the parent node's value as sum of left + right child

**query**:

We are given a range [L, R] and we need to find the sum of this range.

Consider current node's segment to be [l, r]. There exists three cases:

[L, R] == [l, r]- We can directly return current node's value in this case

[L, R] falls in the left or right child's range ([l, mid] or [mid + 1, r])- We need to recurse for left or right child.

[L, R] falls partially in both left and right child's range- Here, we need to make two separate recursive calls to both the left and right child nodes and return their sum.

Instead of writing two separate cases for 2 and 3, I have combined it into single return statement in the below function.

class NumArray {

inline static int n;

int\* segTree;

public:

NumArray(vector<int>& nums) {

n = size(nums);

segTree = (int\*)malloc(4 \* n \* sizeof(int));

build(nums, 1, 0, n - 1);

}

void build(vector<int>& nums, int segIdx, int l, int r) {

if(l == r) segTree[segIdx] = nums[l]; // can't split further - assign corresponding index of segTree as nums[l]

else {

int mid = (l + r) / 2; // divide array into two halves (left and right child of current node of segment tree)

build(nums, 2\*segIdx, l, mid); // calculate result for left child

build(nums, 2\*segIdx + 1, mid + 1, r); // calculate result for right child

segTree[segIdx] = segTree[2\*segIdx] + segTree[2\*segIdx + 1]; // merge result into parent - parent val = sum of left & right child

}

}

void update(int index, int val, int segIdx = 1, int l = 0, int r = n - 1) {

if(l == r) segTree[segIdx] = val; // update segment tree index corresponding to the array index to be updated

else {

int mid = (l + r) / 2;

if (index <= mid) update(index, val, 2\*segIdx, l, mid); // if node holding [nums[index]] segment lies in left child, recurse for it

else update(index, val, 2\*segIdx + 1, mid + 1, r); // other wise recurse the right child

segTree[segIdx] = segTree[2\*segIdx] + segTree[2\*segIdx + 1]; // child nodes got updated - so parent need to be updated again as sum of left + right child nodes

}

}

int sumRange(int left, int right, int segIdx = 1, int l = 0, int r = n - 1) {

if(left > right) return 0;

if(l == left && r == right) return segTree[segIdx]; // case - 1

int mid = (l + r) / 2;

// case - 2 and 3

// Here, we are picking min(right, mid) & max(left, mid+1) to ensure that [left, right] is always subrange of [l, r] for recursive call

// otherwise base condition would never be reached in some cases

return sumRange(left, min(right, mid), 2\*segIdx, l, mid) +

sumRange(max(left, mid + 1), right, 2\*segIdx + 1, mid + 1, r);

}

};

***Time Complexity :***

Initial construction: **O(N)**, where N is the number of elements in *nums*.

update(): **O(logN)** for each call

sumRange(): **O(logN)** for each call

***Space Complexity :*** **O(N)**, required to maintain the *segTree* array

✔️ ***Solution - II (Segment Trees w/ actual Tree construction)***

Here, I am using custom *SegmentTree* class to actually construct a segment tree rather than simulating using the arrays. Most of the functions remains the same with only few changes in implementation of them.

class SegmentTree {

SegmentTree \*left, \*right;

int L, R, val;

public:

SegmentTree(vector<int>& nums, int l, int r): L(l), R(r), left(NULL), right(NULL) {

build(nums);

}

void build(vector<int>& nums) {

if(L == R) val = nums[L];

else {

int mid = (L + R) / 2;

left = new SegmentTree(nums, L, mid);

right = new SegmentTree(nums, mid + 1, R);

val = left -> val + right -> val;

}

}

void update(int i, int new\_val) {

if(L == R) val = new\_val;

else {

int mid = (L + R) / 2;

if(i <= mid) left -> update(i, new\_val);

else right -> update(i, new\_val);

val = left -> val + right -> val;

}

}

int sum(int l, int r) {

if(l > r) return 0;

if(l == L && r == R) return val;

int mid = (L + R) / 2;

return left -> sum(l, min(mid, r)) + right -> sum(max(l, mid + 1), r);

}

};

class NumArray {

SegmentTree \*tree;

public:

NumArray(vector<int>& nums) {

tree = new SegmentTree(nums, 0, size(nums) - 1);

}

void update(int index, int val) {

tree -> update(index, val);

}

int sumRange(int left, int right) {

return tree -> sum(left, right);

}

};

***Time and Space Complexity :*** same as above

**Solution 2: Binary Indexed Tree (60 min)**

class NumArray {

int[] nums;

int[] BIT;

int n;

public NumArray(int[] nums) {

this.nums = nums;

n = nums.length;

BIT = new int[n + 1];

// Each round will contribute to certain indexes in BIT array

for(int i = 0; i < n; i++) {

init(i, nums[i]);

}

}

private void init(int i, int val) {

// Shift 1 for match BIT array indexes as [1, n] rather than [0, n - 1]

i++;

while(i <= n) {

BIT[i] += val;

i += (i & -i);

}

}

public void update(int index, int val) {

int diff = val - nums[index];

nums[index] = val;

// We also have to update all involved range BIT array indexes

init(index, diff);

}

public int sumRange(int left, int right) {

return getSum(right) - getSum(left - 1);

}

private int getSum(int i) {

int sum = 0;

// Shift 1 for match BIT array indexes as [1, n] rather than [0, n - 1]

i++;

while(i > 0) {

sum += BIT[i];

// Just reverse the logic how BIT build up to find path from given

// node and trace back to top node

i -= (i & -i);

}

return sum;

}

}

/\*\*

\* Your NumArray object will be instantiated and called as such:

\* NumArray obj = new NumArray(nums);

\* obj.update(index,val);

\* int param\_2 = obj.sumRange(left,right);

\*/

Time Complexity:

Prefix sum calculation: O(logN) since we only need to traverse the height of the tree, instead of the whole array

Update: O(logN)

Space Complexity: O(N) since we need to initialize an array of size N+1 to hold the binary indexed tree

**Refer to Document: Explaining the Binary Indexed Tree**

<https://medium.com/@edison.cy.yang/explaining-the-binary-indexed-tree-34f27ad0a513>

**Based on this document, the most critical part is how to build up the Binary Indexed Tree with given nums array, below is the detail steps for the same input nums array shared on document**

======================================================================

Basic operation for i & -i

00000001 = 1

11111111 = -1

00000001 = 1 (1&-1)

00000010 = 2

11111110 = -2

00000010 = 2 (2&-2)

00000011 = 3

11111101 = -3

00000001 = 1 (3&-3)

00000100 = 4

11111100 = -4

00000100 = 4 (4&-4)

00000101 = 5

11111011 = -5

00000001 = 1 (5&-5)

00000110 = 6

11111010 = -6

00000010 = 2 (6&-6)

00000111 = 7

11111001 = -7

00000001 = 1 (7&-7)

etc...

======================================================================

Take example nums array to see how Binary Indexed Tree build up:

public NumArray(int[] nums) {

this.nums = nums;

n = nums.length;

BIT = new int[n + 1];

for (int i = 0; i < n; i++)

init(i, nums[i]);

}

public void init(int i, int val) {

i++;

while (i <= n) {

BIT[i] += val;

i += (i & -i);

}

}

nums = {3, 2, -4, 5, 7, 3, -2, 4, 7, 1, 3, 5, 1, 6, -2}

i must <= 15

----------------------------------------------------------------------

i=0 -> i++=1 -> 1+(1&-1)=2 -> 2+(2&-2)=4 -> 4+(4&-4)=8 -> 8+(8&-8)=16>15[discard]

which means the 1st round as i=0 will impact on BIT index as 1, 2, 4, 8 when building tree by adding nums[0]=3 to these indexes

Reverse thinking, after round i=0, BIT[1]=nums[0], BIT[2]=nums[0], BIT[4]=nums[0], BIT[8]=nums[0]

BIT=[0, 3, 3, 0, 3, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=1 -> i++=2 -> 2+(2&-2)=4 -> 4+(4&-4)=8 -> 8+(8&-8)=16>15[discard]

which means the 2nd round as i=1 will impact on BIT index as 2, 4, 8 when building tree by adding nums[1]=2 to these indexes

Reverse thinking, after round i=1, BIT[2]=nums[0]+nums[1], BIT[4]=nums[0]+nums[1], BIT[8]=nums[0]+nums[1]

BIT=[0, 3, 5, 0, 5, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=2 -> i++=3 -> 3+(3&-3)=4 -> 4+(4&-4)=8 -> 8+(8&-8)=16>15[discard]

which means the 3rd round as i=2 will impact on BIT index as 3, 4, 8 when building tree by adding nums[2]=-4 to these indexes

Reverse thinking, after round i=2, BIT[3]=nums[2], BIT[4]=nums[0]+nums[1]+nums[2], BIT[8]=nums[0]+nums[1]+nums[2]

BIT=[0, 3, 5, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=3 -> i++=4 -> 4+(4&-4)=8 -> 8+(8&-8)=16>15[discard]

which means the 4th round as i=3 will impact on BIT index as 4, 8 when building tree by adding nums[3]=5 to these indexes

Reverse thinking, after round i=3, BIT[4]=nums[0]+nums[1]+nums[2]+nums[3], BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]

BIT=[0, 3, 5, -4, 6, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=4 -> i++=5 -> 5+(5&-5)=6 -> 6+(6&-6)=8 -> 8+(8&-8)=16>15[discard]

which means the 5th round as i=4 will impact on BIT index as 5, 6, 8 when building tree by adding nums[4]=7 to these indexes

Reverse thinking, after round i=4, BIT[5]=nums[4], BIT[6]=nums[4], BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]+nums[4]

BIT=[0, 3, 5, -4, 6, 7, 7, 0, 13, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=5 -> i++=6 -> 6+(6&-6)=8 -> 8+(8&-8)=16>15[discard]

which means the 6th round as i=5 will impact on BIT index as 6, 8 when building tree by adding nums[5]=3 to these indexes

Reverse thinking, after round i=5, BIT[6]=nums[4]+nums[5], BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]+nums[4]+nums[5]

BIT=[0, 3, 5, -4, 6, 7, 10, 0, 16, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=6 -> i++=7 -> 7+(7&-7)=8 -> 8+(8&-8)=16>15[discard]

which means the 7th round as i=6 will impact on BIT index as 7, 8 when building tree by adding nums[6]=-2 to these indexes

Reverse thinking, after round i=6, BIT[7]=nums[6], BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]+nums[4]+nums[5]+nums[6]

BIT=[0, 3, 5, -4, 6, 7, 10, -2, 14, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=7 -> i++=8 -> 8+(8&-8)=16>15[discard]

which means the 8th round as i=7 will impact on BIT index as 8 when building tree by adding nums[7]=4 to these indexes

Reverse thinking, after round i=7, BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]+nums[4]+nums[5]+nums[6]+nums[7]

BIT=[0, 3, 5, -4, 6, 7, 10, -2, 18, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

etc...

======================================================================

After 8 rounds, we can see below relation build up between original input nums array and output BIT array

BIT[1]=nums[0]

BIT[2]=nums[0]+nums[1]

BIT[3]=nums[2]

BIT[4]=nums[0]+nums[1]+nums[2]+nums[3]

BIT[5]=nums[4]

BIT[6]=nums[4]+nums[5]

BIT[7]=nums[6]

BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]+nums[4]+nums[5]+nums[6]+nums[7]

======================================================================

Update with relation between BITs to see range coverage

BIT[1]=nums[0]

BIT[2]=BIT[1]+nums[1]

BIT[3]=nums[2]

BIT[4]=BIT[2]+BIT[3]+nums[3]

BIT[5]=nums[4]

BIT[6]=BIT[5]+nums[5]

BIT[7]=nums[6]

BIT[8]=BIT[4]+BIT[6]+BIT[7]+nums[7]

======================================================================

Coverage relation

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*

\_\_\_\_\_\_\*

\_\_\* \_\_\*

\* \* \* \*

BIT indexes: 0 1 2 3 4 5 6 7 8

arr indexes: 0 1 2 3 4 5 6 7

======================================================================

**Refer to**

<https://leetcode.com/problems/range-sum-query-mutable/solutions/75753/java-using-binary-indexed-tree-with-clear-explanation/>

Java using Binary Indexed Tree with clear explanation

This is to share the explanation of the BIT and the meaning of the bit operations.

public class NumArray {

/\*\*

\* Binary Indexed Trees (BIT or Fenwick tree):

\* https://www.topcoder.com/community/data-science/data-science-

\* tutorials/binary-indexed-trees/

\*

\* Example: given an array a[0]...a[7], we use a array BIT[9] to

\* represent a tree, where index [2] is the parent of [1] and [3], [6]

\* is the parent of [5] and [7], [4] is the parent of [2] and [6], and

\* [8] is the parent of [4]. I.e.,

\*

\* BIT[] as a binary tree:

\* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\*

\* \_\_\_\_\_\_\*

\* \_\_\* \_\_\*

\* \* \* \* \*

\* indices: 0 1 2 3 4 5 6 7 8

\*

\* BIT[i] = ([i] is a left child) ? the partial sum from its left most

\* descendant to itself : the partial sum from its parent (exclusive) to

\* itself. (check the range of "\_\_").

\*

\* Eg. BIT[1]=a[0], BIT[2]=a[1]+BIT[1]=a[1]+a[0], BIT[3]=a[2],

\* BIT[4]=a[3]+BIT[3]+BIT[2]=a[3]+a[2]+a[1]+a[0],

\* BIT[6]=a[5]+BIT[5]=a[5]+a[4],

\* BIT[8]=a[7]+BIT[7]+BIT[6]+BIT[4]=a[7]+a[6]+...+a[0], ...

\*

\* Thus, to update a[1]=BIT[2], we shall update BIT[2], BIT[4], BIT[8],

\* i.e., for current [i], the next update [j] is j=i+(i&-i) //double the

\* last 1-bit from [i].

\*

\* Similarly, to get the partial sum up to a[6]=BIT[7], we shall get the

\* sum of BIT[7], BIT[6], BIT[4], i.e., for current [i], the next

\* summand [j] is j=i-(i&-i) // delete the last 1-bit from [i].

\*

\* To obtain the original value of a[7] (corresponding to index [8] of

\* BIT), we have to subtract BIT[7], BIT[6], BIT[4] from BIT[8], i.e.,

\* starting from [idx-1], for current [i], the next subtrahend [j] is

\* j=i-(i&-i), up to j==idx-(idx&-idx) exclusive. (However, a quicker

\* way but using extra space is to store the original array.)

\*/

int[] nums;

int[] BIT;

int n;

public NumArray(int[] nums) {

this.nums = nums;

n = nums.length;

BIT = new int[n + 1];

for (int i = 0; i < n; i++)

init(i, nums[i]);

}

public void init(int i, int val) {

i++;

while (i <= n) {

BIT[i] += val;

i += (i & -i);

}

}

void update(int i, int val) {

int diff = val - nums[i];

nums[i] = val;

init(i, diff);

}

public int getSum(int i) {

int sum = 0;

i++;

while (i > 0) {

sum += BIT[i];

i -= (i & -i);

}

return sum;

}

public int sumRange(int i, int j) {

return getSum(j) - getSum(i - 1);

}

}

// Your NumArray object will be instantiated and called as such:

// NumArray numArray = new NumArray(nums);

// numArray.sumRange(0, 1);

// numArray.update(1, 10);

// numArray.sumRange(1, 2);