



SEGMENT TREE

Basics

Topics

- Defn
- Simplest form of Segment Tree
- Structure
- Visual Representation
- No of vertices
- Construction

Defn

- A Segment Tree is a data structure that allows answering range queries over an array effectively, while still being flexible enough to allow modifying the array.
- This includes finding the sum of consecutive array elements $a[l \dots r]$, or finding the minimum element in a such a range in $O(\log n)$ time.
- Between answering such queries the Segment Tree allows modifying the array by replacing one element, or even change the elements of a whole subsegment (e.g. assigning all elements $a[l \dots r]$ to any value, or adding a value to all element in the subsegment).

Simplest Form of segment tree

- To start easy, we consider the simplest form of a Segment Tree.
- We want to answer sum queries efficiently. The formal definition of our task is: We have an array $a[0\dots n - 1]$, and the Segment Tree must be able to find the sum of elements between the indices l and r (i.e. computing the sum $\sum_{i=l}^r a[i]$, and also handle changing values of the elements in the array (i.e. perform assignments of the form $a[i] = x$).
- The Segment Tree should be able to process both queries in $O(\log n)$ time.

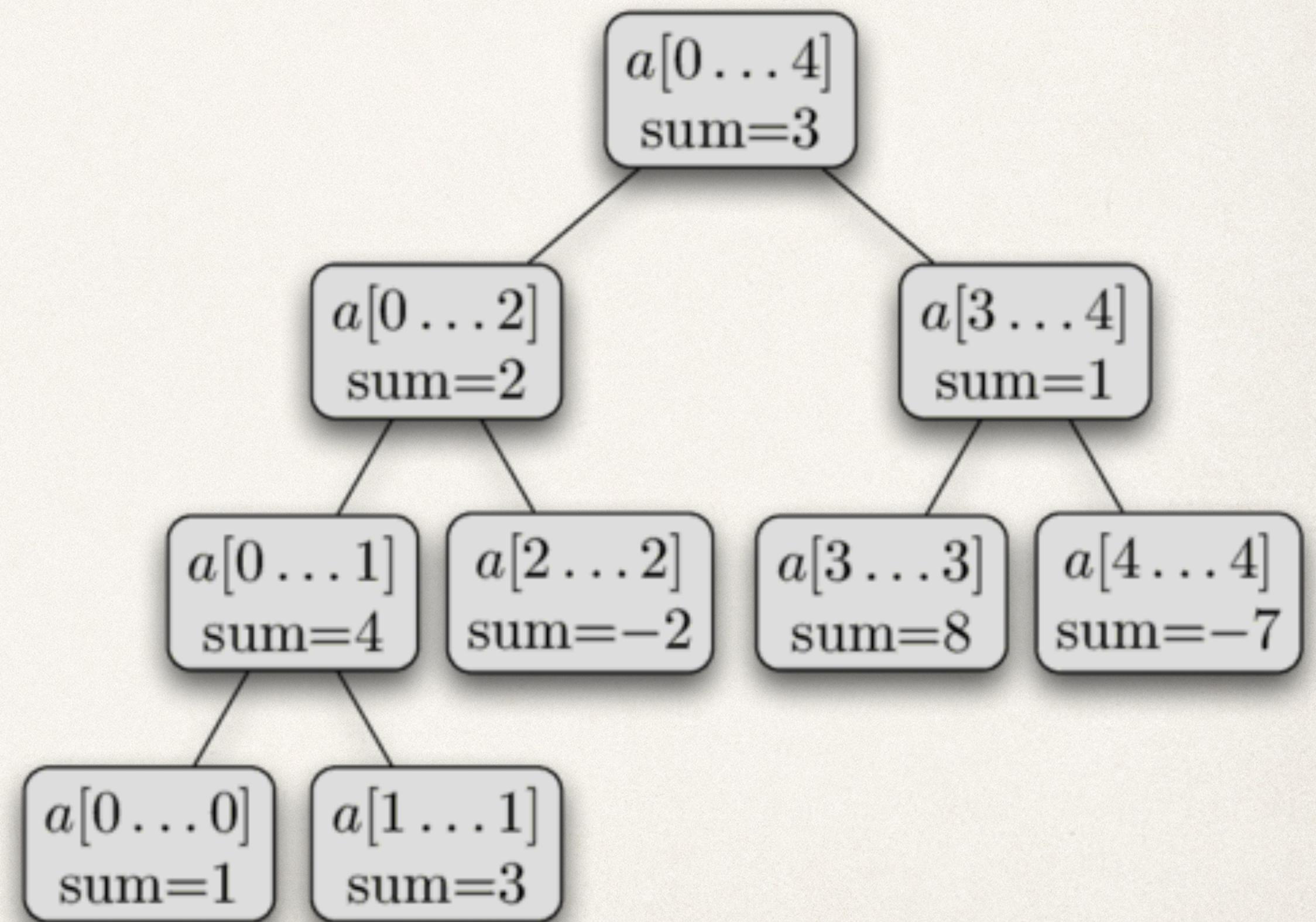
Structure

So, what is a Segment Tree?

- We compute and store the sum of the elements of the whole array, i.e. the sum of the segment $a[0...n - 1]$.
- We then split the array into two halves $a[0...n/2]$ and $a[n/2 + 1...n - 1]$ and compute the sum of each halve and store them.
- Each of these two halves in turn also split in half, their sums are computed and stored. And this process repeats until all segments reach size 1.
- In other words we start with the segment $a[0...n - 1]$, split the current segment in half (if it has not yet become a segment containing a single element), and then calling the same procedure for both halves.
- For each such segment we store the sum of the numbers on it.
- We can say, that these segments form a binary tree: the root of this tree is the segment $a[0 \dots n - 1]$, and each vertex (except leaf vertices) has exactly two child vertices.
- This is why the data structure is called “Segment Tree”, even though in most implementations the tree is not constructed explicitly (see [Implementation](#)).

Visual Representation

Here is a visual representation of such a Segment Tree over the array
 $a = [1, 3, -2, 8, -7]:$



No of vertices

From this short description of the data structure,

- we can already conclude that a Segment Tree only requires a linear number of vertices.
- The first level of the tree contains a single node (the root), the second level will contain two vertices, in the third it will contain four vertices, until the number of vertices reaches n .
- Thus the number of vertices in the worst case can be estimated by the sum
$$1 + 2 + 4 + \dots + 2^{\lceil \log_2 n \rceil} = 2^{\lceil \log_2 n \rceil + 1} < 4n .$$
- It is worth noting that whenever n is not a power of two, not all levels of the Segment Tree will be completely filled.
- We can see that behavior in the image. For now we can forget about this fact, but it will become important later during the implementation.
- The height of the Segment Tree is $O(\log n)$, because when going down from the root to the leaves the size of the segments decreases approximately by half.

Construction

Before constructing the segment tree, we need to decide:

- The value that gets stored at each node of the segment tree. For example, in a sum segment tree, a node would store the sum of the elements in its range $[l, r]$.
- The *merge* operation that merges two siblings in a segment tree. For example, in a sum segment tree, the two nodes corresponding to the ranges $a[l_1 \dots r_1]$ and $a[l_2 \dots r_2]$ would be merged into a node corresponding to the range $a[l_1 \dots r_2]$ by adding the values of the two nodes.

.

Note

Note that a vertex is a “leaf vertex”, if its corresponding segment covers only one value in the original array. It is present at the lowermost level of a segment tree. Its value would be equal to the (corresponding) element $a[i]$.

Now, for construction of the segment tree, we start at the bottom level (the leaf vertices) and assign them their respective values. On the basis of these values, we can compute the values of the previous level, using the merge function. And on the basis of those, we can compute the values of the previous, and repeat the procedure until we reach the root vertex.

It is convenient to describe this operation recursively in the other direction, i.e., from the root vertex to the leaf vertices. The construction procedure, if called on a non-leaf vertex, does the following:

1. recursively construct the values of the two child vertices
2. merge the computed values of these children.

We start the construction at the root vertex, and hence, we are able to compute the entire segment tree. The time complexity of this construction is $O(n)$, assuming that the merge operation is constant time (the merge operation gets called n times, which is equal to the number of internal nodes in the segment tree).

For now we are going to answer sum queries. As an input we receive two integers l and r , and we have to compute the sum of the segment $a[l \dots r]$ in $O(\log n)$ time.

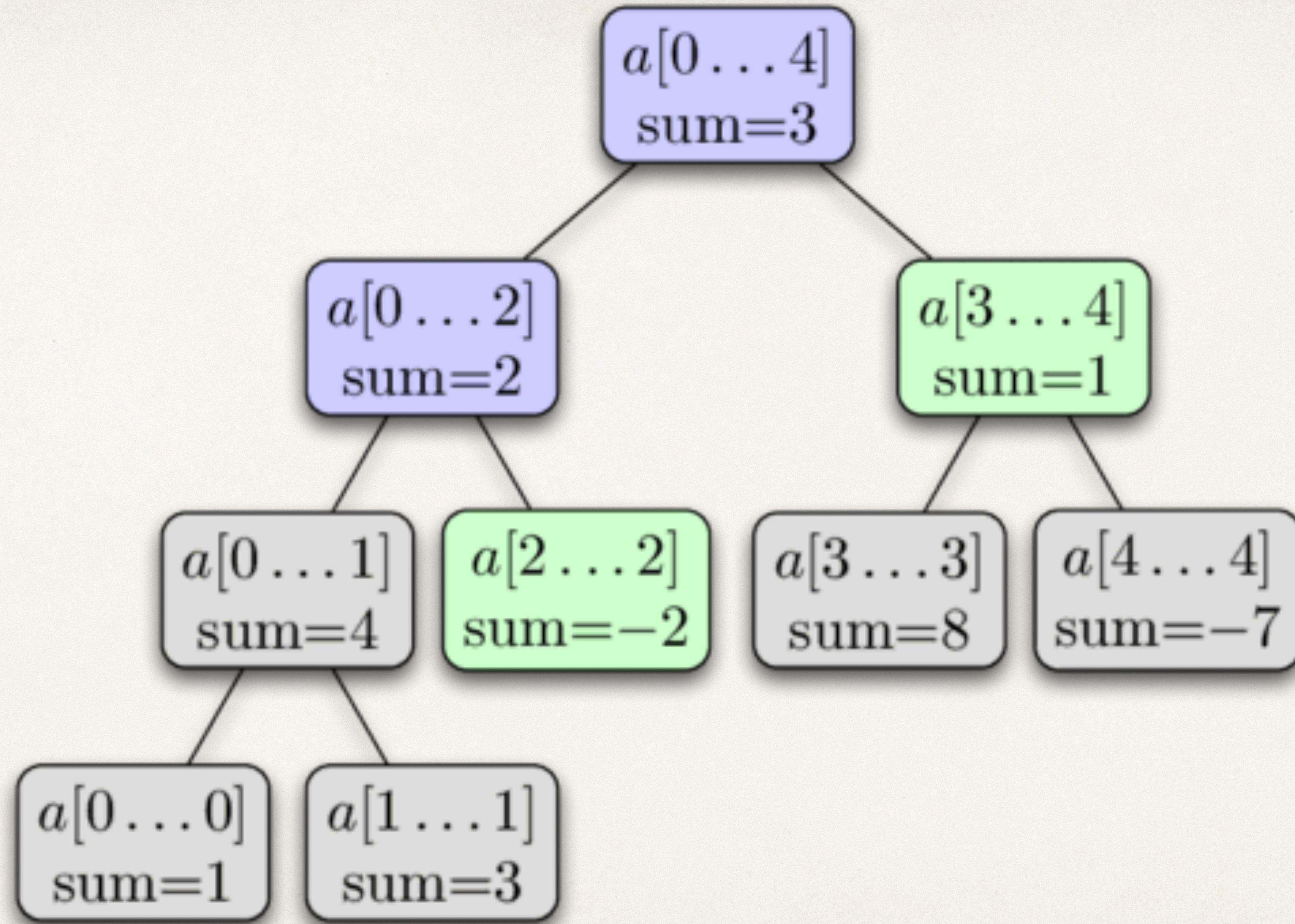
To do this, we will traverse the Segment Tree and use the precomputed sums of the segments. Let's assume that we are currently at the vertex that covers the segment $a[tl \dots tr]$. There are three possible cases.

The easiest case is when the segment $a[l \dots r]$ is equal to the corresponding segment of the current vertex (i.e. $a[l \dots r] = a[tl \dots tr]$), then we are finished and can return the precomputed sum that is stored in the vertex.

Alternatively the segment of the query can fall completely into the domain of either the left or the right child. Recall that the left child covers the segment $a[tl \dots tm]$ and the right vertex covers the segment $a[tm+1 \dots tr]$ with $tm = (tl + tr)/2$. In this case we can simply go to the child vertex, which corresponding segment covers the query segment, and execute the algorithm described here with that vertex.

And then there is the last case, the query segment intersects with both children. In this case we have no other option as to make two recursive calls, one for each child. First we go to the left child, compute a partial answer for this vertex (i.e. the sum of values of the intersection between the segment of the query and the segment of the left child), then go to the right child, compute the partial answer using that vertex, and then combine the answers by adding them. In other words, since the left child represents the segment $a[tl \dots tm]$ and the right child the segment $a[tm + 1 \dots tr]$, we compute the sum query $a[l \dots tm]$ using the left child, and the sum query $a[tm + 1 \dots r]$ using the right child.

- So processing a sum query is a function that recursively calls itself once with either the left or the right child (without changing the query boundaries), or twice, once for the left and once for the right child (by splitting the query into two subqueries).
- And the recursion ends, whenever the boundaries of the current query segment coincides with the boundaries of the segment of the current vertex.
- In that case the answer will be the precomputed value of the sum of this segment, which is stored in the tree.
- In other words, the calculation of the query is a traversal of the tree, which spreads through all necessary branches of the tree, and uses the precomputed sum values of the segments in the tree.
- Obviously we will start the traversal from the root vertex of the Segment Tree.
- The procedure is illustrated in the following image. Again the array $a = [1, 3, -2, 8, -7]$ is used, and here we want to compute the sum $\sum_{i=2}^4 a[i]$. The colored vertices will be visited, and we will use the precomputed values of the green vertices. This gives us the result $-2 + 1 = -1$.



Time Complexity

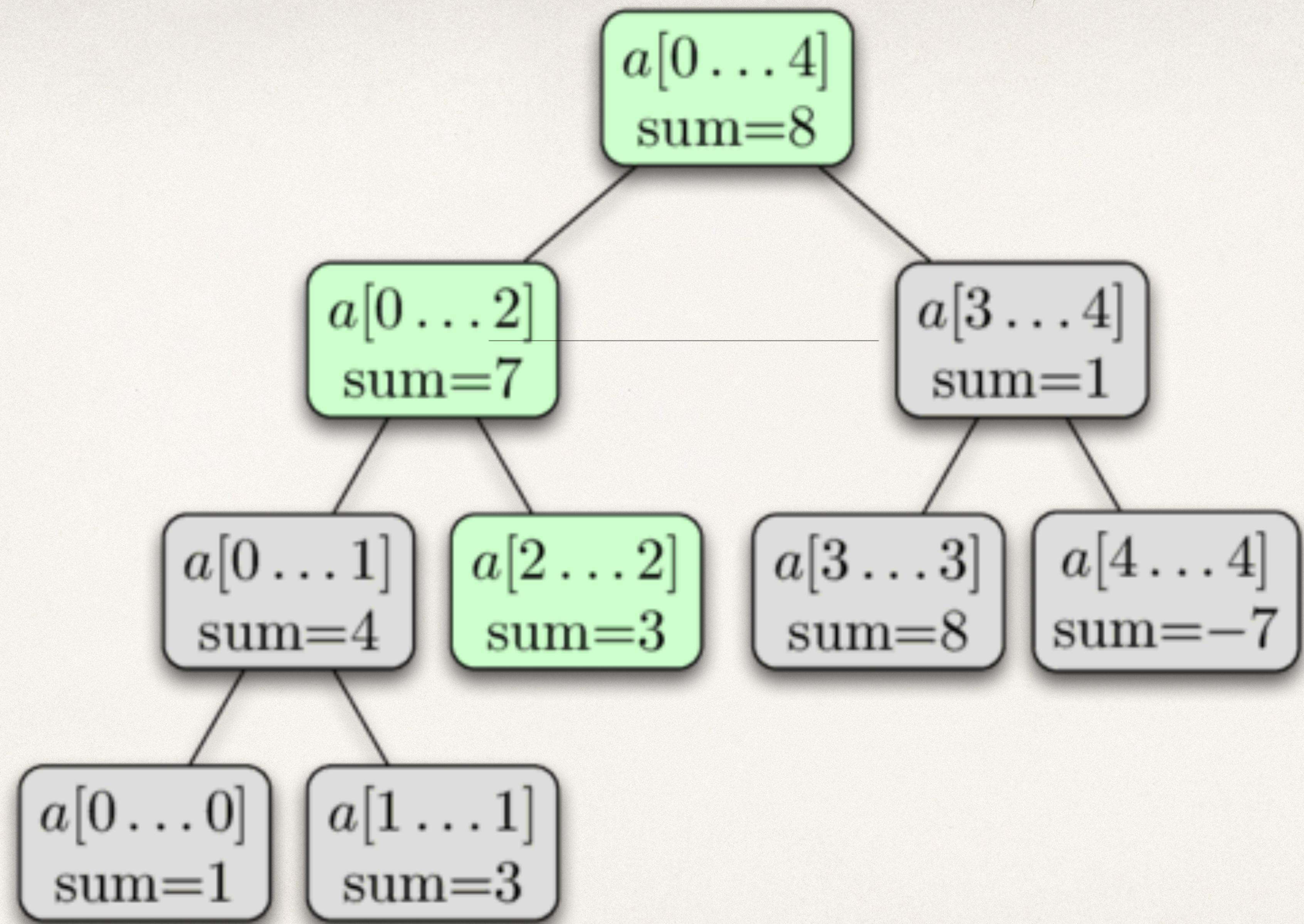
- Why is the complexity of this algorithm $O(\log n)$?
- To show this complexity we look at each level of the tree.
- It turns out, that for each level we only visit not more than four vertices. And since the height of the tree is $O(\log n)$, we receive the desired running time.
- We can show that this proposition (at most four vertices each level) is true by induction.
At the first level, we only visit one vertex, the root vertex, so here we visit less than four vertices. Now let's look at an arbitrary level. By induction hypothesis, we visit at most four vertices. If we only visit at most two vertices, the next level has at most four vertices. That trivial, because each vertex can
 -

Proof

- only cause at most two recursive calls.
- So let's assume that we visit three or four vertices in the current level. From those vertices, we will analyze the vertices in the middle more carefully. Since the sum query asks for the sum of a continuous subarray, we know that segments corresponding to the visited vertices in the middle will be completely covered by the segment of the sum query.
- Therefore these vertices will not make any recursive calls. So only the most left, and the most right vertex will have the potential to make recursive calls.
- And those will only create at most four recursive calls, so also the next level will satisfy the assertion.
- We can say that one branch approaches the left boundary of the query, and the second branch approaches the right one.
- Therefore we visit at most $4 \log n$ vertices in total, and that is equal to a running time of $O(\log n)$.
- In conclusion the query works by dividing the input segment into several sub-segments for which all the sums are already precomputed and stored in the tree. And if we stop partitioning whenever the query segment coincides with the vertex segment, then we only need $O(\log n)$ such segments, which gives the effectiveness of the Segment Tree.

Update Query

- Now we want to modify a specific element in the array, let's say we want to do the assignment $a[i] = x$. And we have to rebuild the Segment Tree, such that it correspond to the new, modified array.
- This query is easier than the sum query. Each level of a Segment Tree forms a partition of the array. Therefore an element $a[i]$ only contributes to one segment from each level. Thus only $O(\log n)$ vertices need to be updated.
- It is easy to see, that the update request can be implemented using a recursive function. The function gets passed the current tree vertex, and it recursively calls itself with one of the two child vertices (the one that contains $a[i]$ in its segment), and after that recomputes its sum value, similar how it is done in the build method (that is as the sum of its two children).
- Again here is a visualization using the same array. Here we perform the update $a[2] = 3$. The green vertices are the vertices that we visit and update.
.



Implementation

- The main consideration is how to store the Segment Tree.
- Of course we can define a Vertex struct and create objects, that store the boundaries of the segment, its sum and additionally also pointers to its child vertices.
- However this requires storing a lot of redundant information. We will use a simple trick, to make this a lot more efficient. We only store the sums in an array. The sum of the root vertex at index 1, the sums of its two child vertices at indices 2 and 3, the sums of the children of those two vertices at indices 4 to 7, and so on. It is easy to see, that the left child of a vertex at index i is stored at index $2i$, and the right one at index $2i + 1$.
- This simplifies the implementation a lot. We don't need to store the structure of the tree in memory. It is defined implicitly. We only need one array which contains the sums of all segments.
- As noted before, we need to store at most $4n$ vertices. It might be less, but for convenience we always allocate an array of size $4n$. There will be some elements in the sum array, that will not correspond to any vertices in the actual tree, but this doesn't complicate the implementation.
- So, we store the Segment Tree simply as an array $t[]$ with a size of four times the input size n :

So, we store the Segment Tree simply as an array $t[]$ with a size of four times the input size n :

```
int n, t[4*MAXN];
```

The procedure for constructing the Segment Tree from a given array $a[]$ looks like this: it is a recursive function with the parameters $a[]$ (the input array), v (the index of the current vertex), and the boundaries tl and tr of the current segment. In the main program this function will be called with the parameters
of the root vertex: $v=1, tl=0, and tr=n-1$.

