Segment Trees

(Primary Author: Ahmad Barhamje, University of Central Florida)

Tree Terminology:

Depth of a Node N:

Length of the path from the root to N (number of edges from the root to the node)

Height of a Node N:

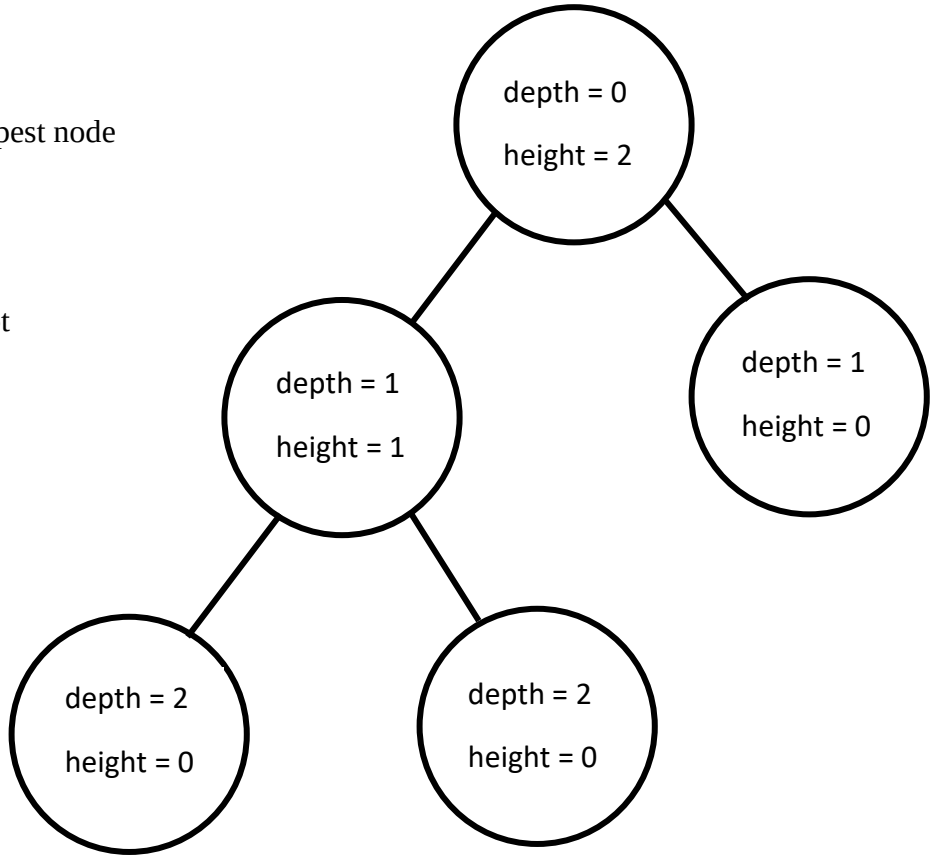
Length of the longest path from N to a leaf (number of edges from the node to the deepest leaf)

Depth of a Tree:

Depth of the deepest node

Height of a Tree:

Height of the root



Balanced Binary Tree:

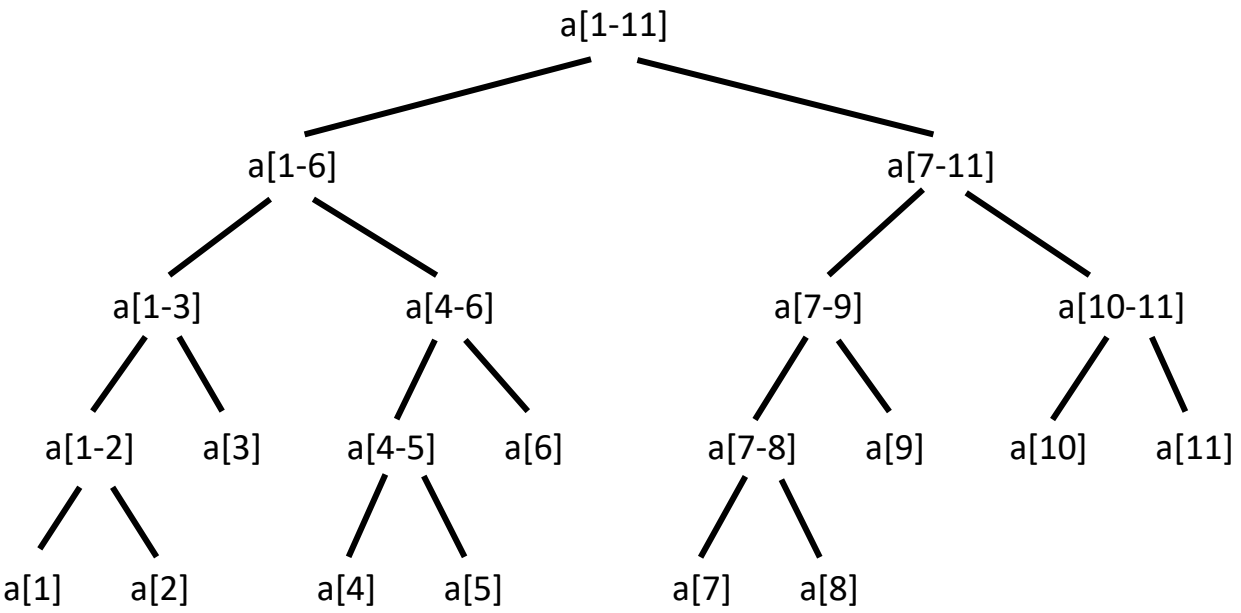
A binary tree is a tree in which every node has at most two children (generally called the left and right children). A binary tree is considered balanced when, for every node, the difference between the heights of its two children is ≤ 1.

An important property of a balanced binary tree consisting of ***n*** nodes is that its height will be no greater than $\lceil \log_2(\boldsymbol{n}) \rceil$.

Generating a Balanced Binary Tree from an Array of Values:

In Segment Trees, the array elements will all be at the leaf level. So, we need to generate a balanced tree where all the array values are at the leaf level. This requires including non-leaf nodes to make a balanced tree.

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]



We are now ready to move to Segment Trees!

Motivating Problem:

Given an array of integers, we need to answer two types of queries on this array:

1) What is the minimum number on the range [**L**, **R**] in the array?
2) Change the value at index ***p*** to some new value ***v***.

The naïve approach would be to have an *O(n)* runtime for every query of type 1 (just loop along the range and find the minimum value) and an *O(1)* runtime for type 2 queries. Of course, we can do better by taking advantage of a balanced binary tree.

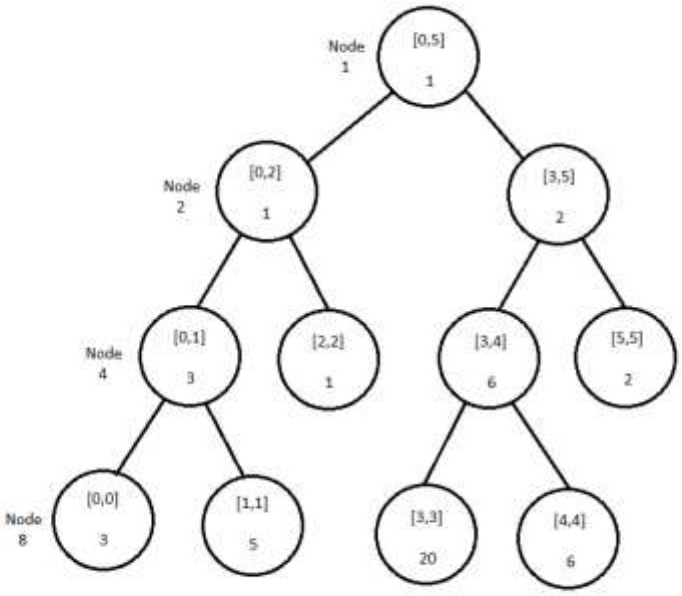
Segment Trees:

A segment tree is a balanced binary tree in which every node stores some information about a contiguous range of data (a segment of data). Every node in this binary tree is responsible for a portion of the information we care about and we can combine the information stored in these nodes to get information about any range of the array.

In the case of our motivating problem, what each of our nodes will store will be:

- The range of the original array that it covers (two integers L and R).
- The minimum value over the range it covers.

For array = [3, 5, 1, 20, 6, 2], the segment tree looks like:



There are a couple of things to notice about this tree. First, its height (which is 3) follows the claim we made earlier about balanced binary trees since $\log_2(6)$ is approximately 3.

The second thing to notice is about the ranges covered by a node and its children. If we look at Node 2 (whose range is [0, 2]) and the ranges of its children 4 and 5 ([0, 1] and [2, 2] respectively) we can notice that the union of the ranges covered by Nodes 4 and 5 is exactly equal to the range covered by Node 2. This holds for every node in the tree except for leaf nodes who cover a range of exactly 1 index.

Now we'll go over how the "minimum value" property for each node is calculated. Leaf nodes in this tree will always cover exactly one index so their minimum value is simple, it's just whatever value is at that index in the array. Next we'll look at Node 4 (since both of its children already have their values computed). Because the range covered by Node 4 completely contains the ranges covered by Nodes 8 and 9 and because the union of the ranges covered by Nodes 8 and 9 is also exactly equal to Node 4's range we can say that the minimal value covered by Node 4 is simlpy the minimum of the Nodes 8 and 9's minimal value (which is 3).

This process repeats all the way up to the root node where every node looks at the values computed by its children and determines its own values based on only those two nodes.

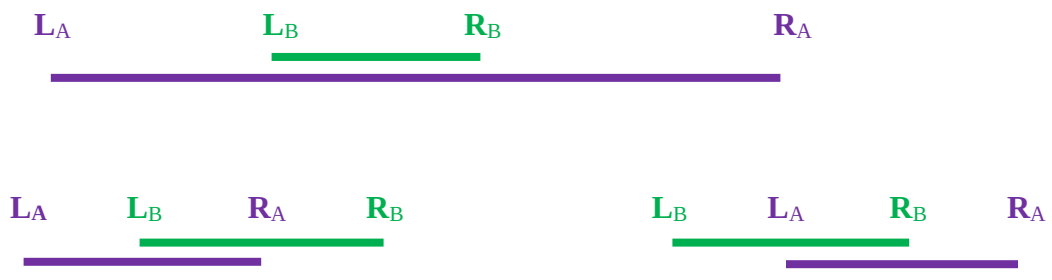
Querying a Range:

So, let's figure out how to calculate an answer for a type 1 query in *O(log(**n**))* time. First, let's define the term "complete covering" and "partial covering":

A range ***B*** is said to be completely covered by a range ***A*** if $\mathbf{L_A} \leq \mathbf{L_B} \leq \mathbf{R_B} \leq \mathbf{R_A}$

A range ***B*** is said to be partially covered by a range ***A*** if $\mathbf{L_A} \leq \mathbf{L_B} \leq \mathbf{R_A} \leq \mathbf{R_B}$ or if $\mathbf{L_B} \leq \mathbf{L_A} \leq \mathbf{R_B} \leq \mathbf{R_A}$

In simple terms, a range is partially covered by another if they share at least one but not all elements that they cover, and a range is completely covered if it shares all its elements with another (and is smaller).



Let's say the range for our minimum is called ***A*** and $\mathbf{L_A}$ = 1 and $\mathbf{R_A}$ = 4. We always start our queries at the root node 1 and repeat this recursive process for every node:

1. Look at the range covered by our current node (we'll call it ***B***).
2. If ***B*** is completely covered by ***A***, then we return the minimum value stored at this node.
3. If ***B*** is not completely or partially covered by ***A*** then we'll return some *null* value.
4. If ***B*** is partially covered by ***A***, then we call this process of the current node's left child and right child and return the minimum of the two values.

If we mark all nodes where the 2nd step was satisfied (i.e., all the nodes whose ranges were completely covered by the original range), we'll notice that only 3 nodes are marked (which is again approximately $\log_2(\boldsymbol{n})$).

A noticeable property of these marked nodes is that if a node is marked, no nodes in its subtree will be marked. Also, if a node is not marked it is guaranteed that both of its children will not be marked. From this, we can make the claim that for any query, no more than 2 nodes will be marked on the same depth level. Since we only have $\log_2(\boldsymbol{n})$ levels and there are at most 2 marked nodes per level we'll visit a total of 2 * $\log_2(\boldsymbol{n})$ levels for a total runtime of $O(\log_2(\boldsymbol{n}))$.

Modifying an Index:

To modify a single index, we'll follow a procedure similar to querying a range. The target index we're looking to change is called ***idx*** and the value we want to change it to is ***v***. The goal for this process is to find the leaf node that covers only ***idx*** and modify that note along with all its ancestors. The recursive process is as follows:

1. Look at the range covered by our current node (call it ***B***).
2. If ***idx*** is out of the range of ***B*** then we return.
3. If we're at a leaf node and our range is exactly equal to ***idx*** then we've found the correct leaf node and we can set its minimum value to ***v*** and return.
4. Otherwise we know ***idx*** is in one of our children and so we'll call this procedure on both of them (one should return immediately). After we return from those recursive calls, we know that we'll have modified ***idx*** and we need to recalculate our minimum value. Since our minimum depends only on our two children, we set our value to min(leftChild, rightChild) and return.

At the end of this process, all nodes on the way to the leaf containing ***idx*** will have been properly modified and we can notice that we only walk down a single path (which has length $\log_2(\boldsymbol{n})$).

Other Operations:

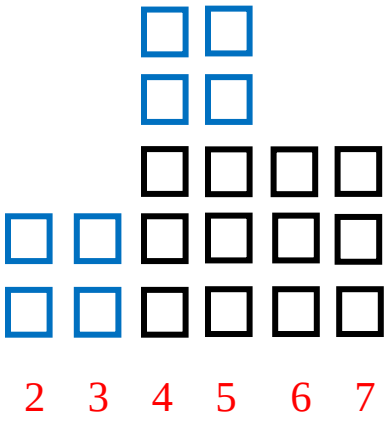
In our motivating problem, we really only looked at changing a specific value and finding the minimum across a range. But you can do many different operations on a range instead of just

minimum. For example: sum, gcd, and xor can all be done with what is basically the same code for finding minimum on a range.

By storing some extra information in your nodes you can even count the number of occurences of a pattern string ***P*** on the range of a separate string ***S***!

Example 1 – Chocolate Bars:

In the new game Chocolate Building, the game screen has a given width, in chocolate bar units, and an infinite height. On a single move, a player chooses a consecutive sequence of columns and adds some fixed number of unit chocolate bars to each of these columns. Naturally, the chocolate drops as far down to the bottom until it rests. For example, if we add 3 units of chocolate to columns 4 through 7 and then add 2 units of chocolate to columns 2 through 5, our building would look like this:



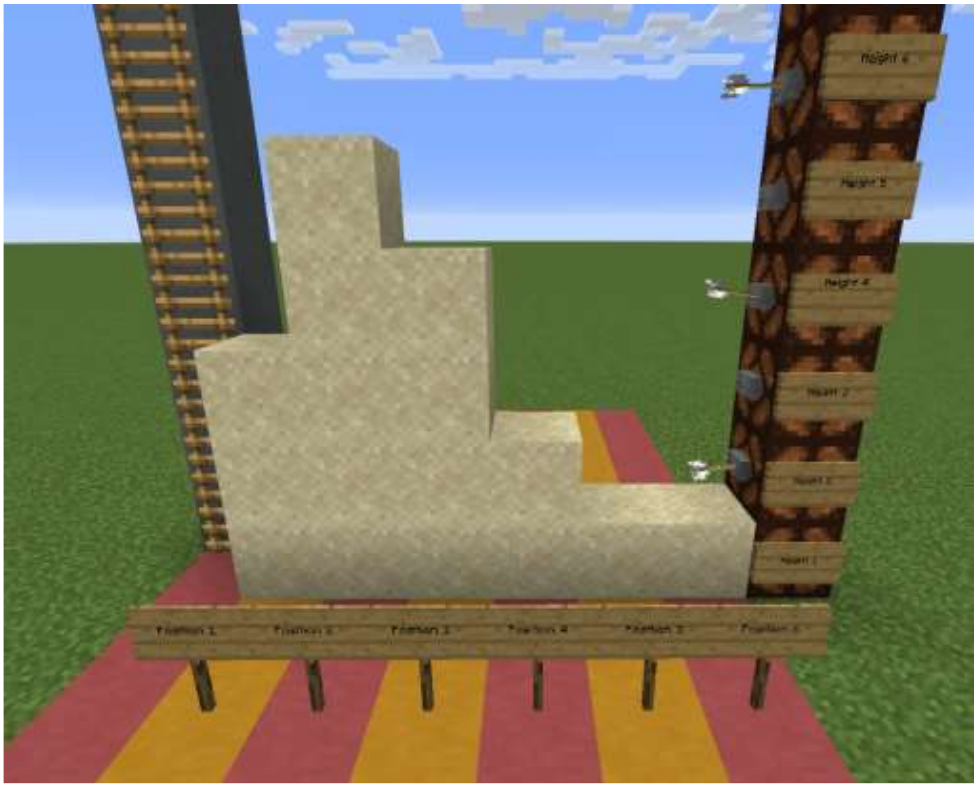
At any time, you are also curious how many unit chocolate bars are within a given consecutive sequence of columns. For example, we may want to know how many unit bars of chocolate there are in between columns 3 and 6, inclusive. For this query, the result is 2 + 5 + 5 + 3 = 15. Write a program that can process a sequence of moves and queries during the chocolate game.

This problem can be solved using a "sum segment tree".

Example 2 – Snake and Ladders:

You are a big fan of the video game character Solid Snake, so you are practicing your Minecraft PvP skills and ladder climbing in hopes of training to be as good as he is. Today you plan to shoot arrows at buttons at different heights. In your shooting range, there is a ladder on the left side and targets on the right, with n blocks separating them. q times, you will climb up or down the ladders until you are level with the target you wish to shoot, and then shoot an arrow straight across from left to right at the desired target.

One of your friends has just crafted an invisibility potion and, in an attempt to troll you, will drop a single piece of sand above each position from l_i to r_i in the shooting range before each shot you make. The sand will fall straight down from where it is dropped, taking up one block of space when it lands. You would like to know, for each shot, whether you will be able to hit the desired target, or whether you will instead hit a block of sand. Initially the range is flat and all blocks are height 0. The sand will land before you are able to shoot.



Given n times in which sand is dropped in the range l_i to r_i and the height h_i of the target you would like to hit, determine whether your shot will hit the target or a block of sand.

This problem can be solved using a "max segment tree".

Example 3 – Circular RMQ (from CodeForces):

https://codeforces.com/contest/52/problem/C

Lazy Propagation:

So far, all of our update operations have been on a single index but we can also apply updates to entire ranges of our array. The idea of lazy propagation is to only apply our updates when we need the information.

We'll go over the process of adding a number ***v*** over a range [**L, R**] on our array. To do this, every node will store another variable called its "lazy value." The lazy value acts like a flag so that once we revisit this node, we know that its values need to be updated.

Our update operations will now end up looking very similar to our query operations. Here's the basic process for adding a number ***v*** to a range ***A*** of an array (remember we always start at the root node of our segment tree):

1. Look at the current range of our node (***B***).
2. If ***B*** is completely covered by ***A***, we add ***v*** to our current node's lazy value.
3. If ***B*** isn't partially covered by ***A***, return.
4. If our current node has a non-null lazy value (i.e., some other query attempted to update this node) then we need to propagate (or push down) that value to its children.
5. Recurse on both left and right children. After those recursive calls, we need to reupdate the current node's minimum value on the range. This is done by taking our left child's value + our left child's lazy value and doing the same thing with our right child. Our value will be the minimum of the two.

The noticeable changes here are in steps 4 and 5. In step 4, we found a value that was previously placed there lazily and now that we need to deal with our children (who didn't previously receive that update), we need to make sure their lazy values get updated to include our current lazy value.

For step 5, we're now updating our minimum value by adding our children's lazy values. This is because our current minimum value needs to reflect all changes that were pushed through this node even if our children haven't completely updated theirs yet.

Now if we want to query the minimum value on a range ***A***, the recursive process will look something like this:

1. Look at the current range of our node (***B***) and if it isn't partially or completely covered by ***A***, return some null value.
2. Otherwise, if it's completely covered by ***A***, we return this node's minimum value + lazy value.
3. If it's partially covered by ***A***, we should check to see if we have a non-null lazy value. If we do, we push it down to our children and make the two recursive calls.
4. Now that we've pushed down our lazy value, we need to again make sure that the minimum value stored at this node accurately reflects the array based on our children's values. So we update our minimum with leftValue + leftLazy and rightValue + rightLazy.
5. We'll then return the minimum value that we got from our two recursive calls.

The key thing to understand from the lazy value is that it represents an update to the entire subtree of a node, but we won't bother communicating that to the rest of the subtree until we get a query that needs to go down there. And if some query's range completely covers a node that has a lazy value it has yet to push down, we need to make sure that the value we return uses the lazy value as well.

As we learned in the previous sections, these query and update operations will always hit $O(log_2(\boldsymbol{n}))$ nodes and so lazy propagation will also let us update a range in $O(log_2(\boldsymbol{n}))$ time.

Lazy Propagation with Other Operations:

We really only looked at lazy propagation when our update operation is to add a value to a range. Generally, lazy propagation is an idea that works easily with almost all operations. For example, if you wanted to assign an entire range to be equal to some value $\boldsymbol{v}$, you can lazily assign a node's value to have $\boldsymbol{v}$. Later if we ended up querying that node for the sum across a range, this node would return its lazy value ($\boldsymbol{v}$) * the number of indices it covers.

In general, it's unlikely that you will run into classic segment tree problems in a live contest but will instead need to maintain some more complicated information in the nodes of your segment tree or use the idea of a segment tree to speed up other ideas (it's common to see a DP problem require a segment tree to quickly calculate values). But if you understand the underlying ideas behind how and why a segment tree works, adapting it to non-classic ideas becomes significantly easier.