Contents

List of Algorithms

Chapter 6

Complex Ideas and Data Structures

Here we build on previous material to introduce more complex ideas that are useful for solving USACO Gold problems and beyond.

6.1 Dynamic Programming over Subsets ($n2^n$ DP)

We've already covered how dynamic programming can turn exponential solutions into polynomial solutions, but it can also help turn factorial solutions into exponential. Problems where the bound on n is 20, for example, signal that an exponential solution is the one required. Consider the following problem:

(USACO December 2014, guard) Farmer John and his herd are playing frisbee. Bessie throws the frisbee down the field, but it's going straight to Mark the field hand on the other team! Mark has height H ($1 \leq H \leq 1,000,000,000$), but there are N cows on Bessie's team gathered around Mark ($2 \leq N \leq 20$). They can only catch the frisbee if they can stack up to be at least as high as Mark. Each of the N cows has a height, weight, and strength. A cow's strength indicates the maximum amount of total weight of the cows that can be stacked above her.

Given these constraints, Bessie wants to know if it is possible for her team to build a tall enough stack to catch the frisbee, and if so, what is the maximum safety factor of such a stack. The safety factor of a stack is the amount of weight that can be added to the top of the stack without exceeding any cow's strength.

We can try the $O(N!)$ brute force, trying every permutation of cows possible. However, this is far too slow. $N \leq 20$ hints at an exponential solution, so we think of trying every possible subset of the cows. Given a subset S of cows, the height reached is the same, so perhaps we sort the subset by strength, and put the strongest cow on the bottom. We see that this greedy approach fails: suppose that the first cow has weight 1 and strength 3 and the second cow has weight 4 and strength 2. Greedy would tell us to put the first cow on the bottom, but this fails, while putting the second cow on the bottom succeeds.

When greedy fails, the next strategy we look at is dynamic programming. To decide whether S is stable, we have to find whether there exists a cow j in S that can support the weight of all the other cows in S. But how do we know whether the set $S \setminus \{j\}$ is stable? This is where dynamic programming comes in.

This leads to a $O(N2^N)$ solution. This seems like a pain to code iteratively, but there is a nice fact about subsets: there is a cute bijection from the subsets of $\{0, 1, 2, \ldots, N-1\}$ to the integers from 0 to $2^N - 1$. That is, the subset $\{0, 2, 5, 7\}$ maps to $2^0 + 2^2 + 2^5 + 2^7 = 165$ in the bijection. We call this technique *masking*. We require all the subsets of S to be processed before S is processed, but that property is also handled by our bijection, since subtracting a power of 2 from a number decreases it. With a little knowledge of bit operators, this can be handled easily.

for $i \leftarrow 0, 2^N - 1$ **do** ▷ i represents the subset S
  $dp(i) \leftarrow -1$
  for all $j \in S$ **do** ▷ $j \in S$ satisfy `i & (1 << j) != 0`
    $alt \leftarrow \min(dp(i - 2^j), strength(j) - \sum_{k \in S \setminus \{j\}} weight(k))$
    if $dp(i) < alt$ **then**
      $dp(i) \leftarrow alt$

`&` is the bitwise and function, while `<<` is the left shift operator.

6.2 $\sqrt{n}$ Bucketing

$\sqrt{n}$ bucketing is a relatively straightforward idea – given n elements $\{x_i\}_{i=1}^n$ in a sequence, we group them into $\sqrt{n}$ equal-sized buckets. The motivation for arranging elements like this is to support an operation called a *range query*.

Let's take a concrete example. Suppose we want to support two operations:

- $update(i, z)$ – increment the value of x_i by z
- $query(i, j)$ – return $\sum_{k=i}^{j} x_k$.

Suppose we simply stored the sequence in an array. *update* then becomes an $O(1)$ operation, but *query* is $O(n)$.

2	4	7	−5	3	6	−3	1	−2	−4	−6	2	8	6	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

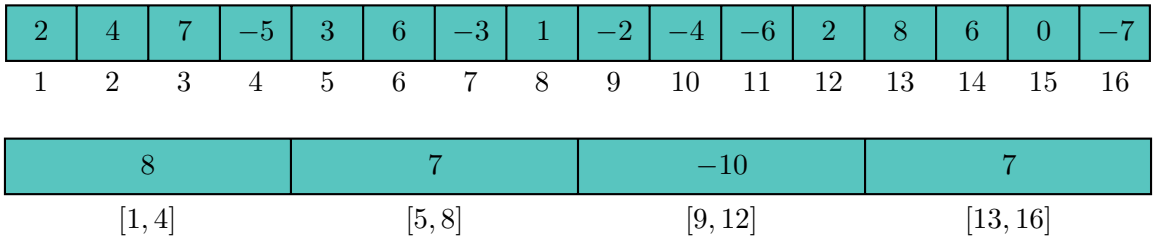
Another natural approach would be to store in a separate array the sum of the first i terms in the sequence for every index i, or store the *prefix sums*.

0	2	6	13	8	11	17	14	15	13	9	3	5	13	19	19	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

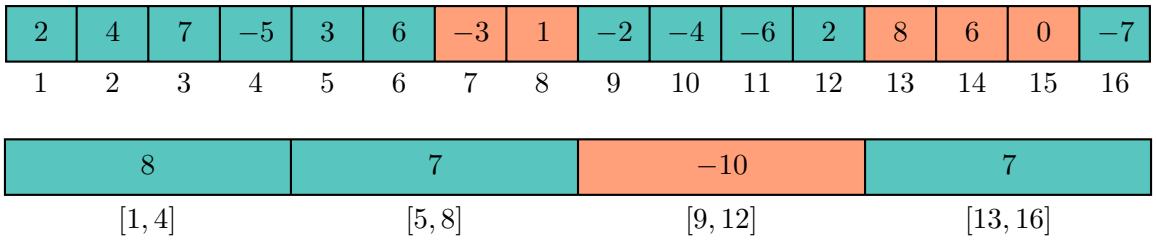
Now *query* becomes an $O(1)$ operation, as we can simply subtract two elements in the array to answer a query. Unfortunately, *update* becomes $O(n)$, as changing the value of an element in the beginning of the sequence forces us to change almost all the values in the prefix sum array.

We can still use this idea, though... what we are looking for is some way to group values into sums such that we only need to change a small number of the sums to *update* and only require a small number of them to *query*.

This leads us directly to a $\sqrt{n}$ bucketing solution. Let's group the 16 elements into 4 groups.



We'll keep track of the total sum of each group. Now, if we want to update a value, we need to change only two values – the value of that element in the original array and the total sum of the bucket it is in. When we query a range, we'll take advantage of the sum of the bucket when we can. Highlighted are the numbers we'll need for $query(7, 15)$.



Querying requires access to at most $\sqrt{n}$ bucket sums and $2(\sqrt{n}-1)$ individual values. Therefore we have $O(\sqrt{n})$ query and $O(1)$ update. We are able to improve $O(\sqrt{n})$ update to $O(1)$ because of nice properties of the $+$ operator. This is not always the case for range queries: suppose, for instance, we needed to find the minimum element on a range.

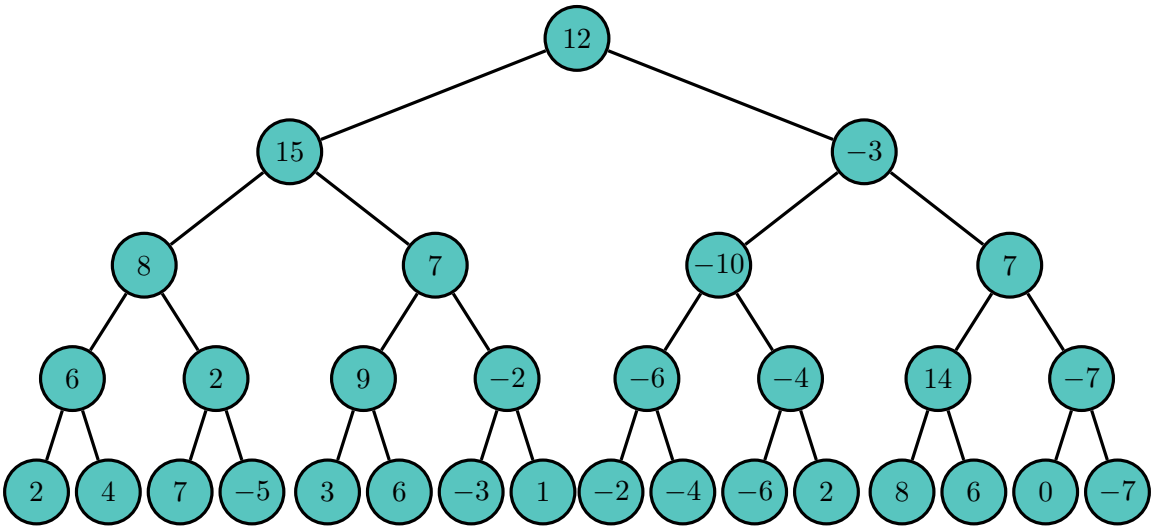
It is often the case that $O(\sqrt{n})$ bounds can be improved to $O(\log n)$ using more complex data structures like segment trees and more complex ideas like 2^n jump pointers, both of which are covered in this chapter. These are, however, more complicated to implement and as such are often comparable in runtime in the contest environment. Steven Hao is notorious for using crude $\sqrt{n}$ bucketing algorithms to solve problems that should have required tighter algorithm complexities. $\sqrt{n}$ bucketing is a crude yet powerful idea; always keep it in the back of your mind.

6.3 Segment Tree

For our range sum query problem, it turns out that we can do as well as $O(\log n)$ with a *segment tree*, also known as a *range tree* or *augmented static BBST*. The essential idea is still the same—we want to group elements in some way that allows us to update and query efficiently.

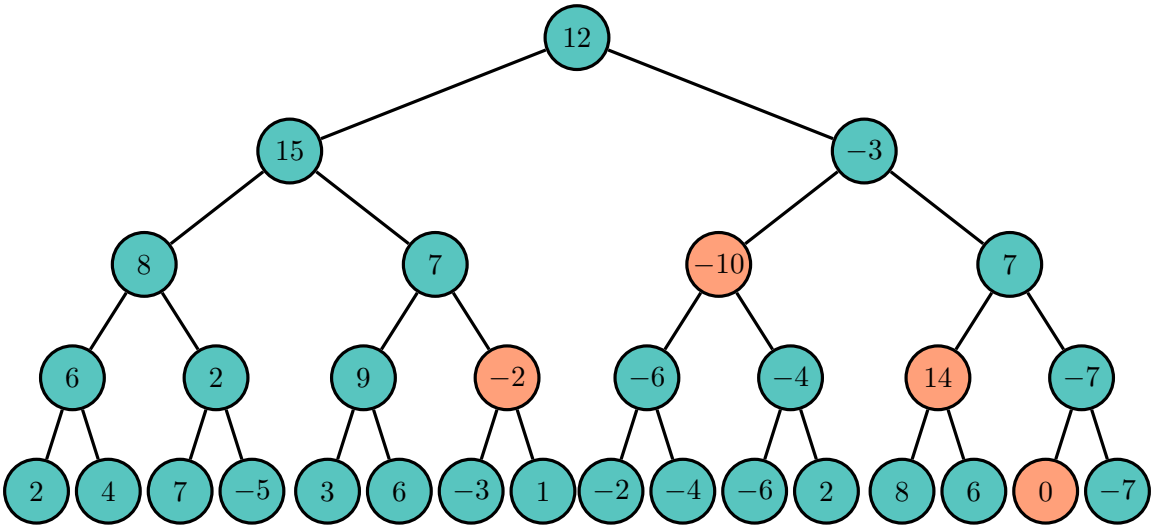
As the name "tree" suggests, we draw inspiration from a binary structure. Let's build a tree on top of the array, where each node keeps track of the sum of the numbers associated with its children. Every node keeps track of the *range* it is associated with and the *value* of the sum of all elements on that range. For example, the root of the tree is responsible for the sum of all elements in the range $[1, n]$, its left child is responsible for the range $\left[1, \left\lfloor \frac{1+n}{2} \right\rfloor\right]$ and its right child is responsible for $\left[\left\lfloor \frac{1+n}{2} \right\rfloor + 1, n\right]$.

In general, for the vertex responsible for the range $[l, r]$, its left child holds the sum for $\left[l, \left\lfloor \frac{l+r}{2} \right\rfloor\right]$ and its right child $\left[\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right]$. As we go down to the tree, eventually we'll have nodes with ranges $[l, l]$ that represent a single element in the original list. These, of course, will not have any children.



2	4	7	−5	3	6	−3	1	−2	−4	−6	2	8	6	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Highlighted are the nodes we'll need to access for $query(7, 15)$. Notice how the subtrees associated with each of these nodes neatly covers the entire range $[7, 15]$.

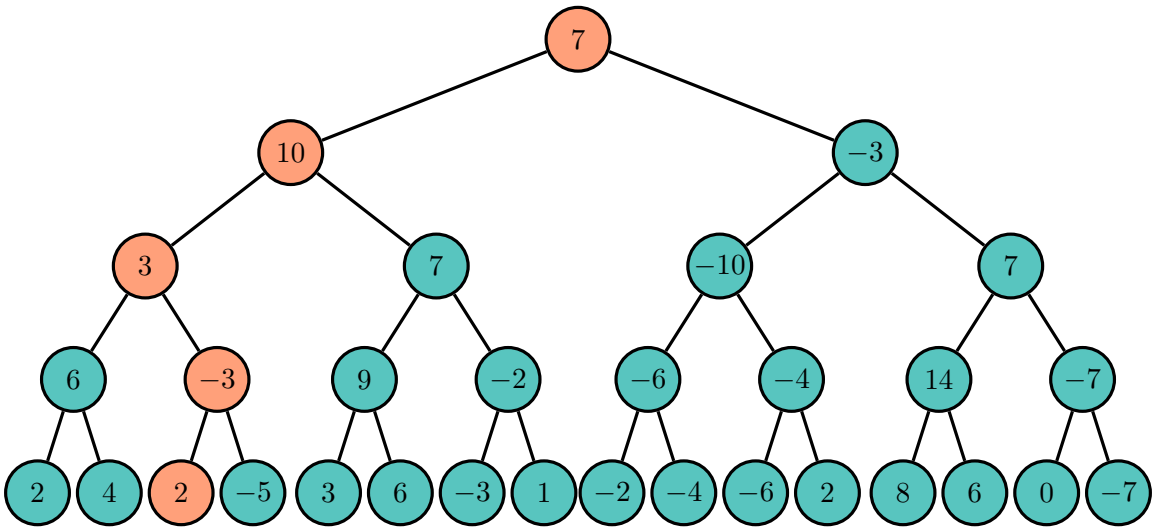


-2 represents the sum $x_7 + x_8$, -10 the sum $x_9 + x_{10} + x_{11} + x_{12}$, 14 the sum $x_{13} + x_{14}$, and 0 represents the single element x_{15}. It seems we always want to take the *largest* segments that stay within the range $[7, 15]$. But how do we know exactly which segments these are?

We handle queries using a recursive call, starting at the root of the tree. We then proceed as follows: If the current node's interval is completely disjoint from the queried interval, we return 0. If the current node's interval is completely contained within the queried interval, we return the sum associated with that node. Otherwise, we pass the query on to the node's two children. Note that this process is $O(\log n)$ because each level in the tree can have at most two highlighted nodes.

function QUERY(range $[l, r]$, range $[a, b]$) ▷ at node $[l, r]$, want $\sum_{i=a}^{b} x_i$
  if $r < a$ or $b < l$ **then** ▷ $[l, r] \cap [a, b] = \emptyset$
    return 0
  if $a \le l$ and $r \le b$ **then** ▷ $[l, r] \subseteq [a, b]$
    return $sum(l, r)$
  return QUERY($\left[l, \left\lfloor \frac{l+r}{2} \right\rfloor\right], [a, b]$) + QUERY($\left[\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right], [a, b]$)

Segment trees also handle modifications well. If we want to change the third element to 2, then we have to update the highlighted nodes in the following diagram. We can implement this the same way we implement queries. Starting from the root, we update each modified node's children before recomputing the value stored at that node. The complexity is $O(\log n)$; we change the value of one node in each level of the tree.



function UPDATE(range $[l, r]$, p, k) ▷ $x_p \leftarrow x_p + k$
  if $r < p$ or $p < l$ **then** ▷ $p \notin [l, r]$
    return
  if $l = r$ **then** ▷ leaf node
    $sum(l, r) \leftarrow k$
    return
  UPDATE($\left[l, \left\lfloor \frac{l+r}{2} \right\rfloor\right]$, p, k)
  UPDATE($\left[\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right]$, p, k)
  $sum(l, r) \leftarrow sum(l, \left\lfloor \frac{l+r}{2} \right\rfloor) + sum(\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r)$

I cheated with my example by using a nice power of two, $n = 16$, as the number of elements in the sequence. Of course, the size is not always this nice. However, if we want a tree of size n, we can always round up to the nearest power of 2, which is at most $2n$.

Now all we need is a way for us to easily store the sum values associated with each node. We can clearly use a structure with two child pointers, but we can also exploit the fact that the segment tree structure is a balanced binary search tree. We can store the tree like we store a heap. The root is labeled 1, and for each node with label i, the left child is labeled $2i$ and the right child $2i+1$. Here's some sample code.

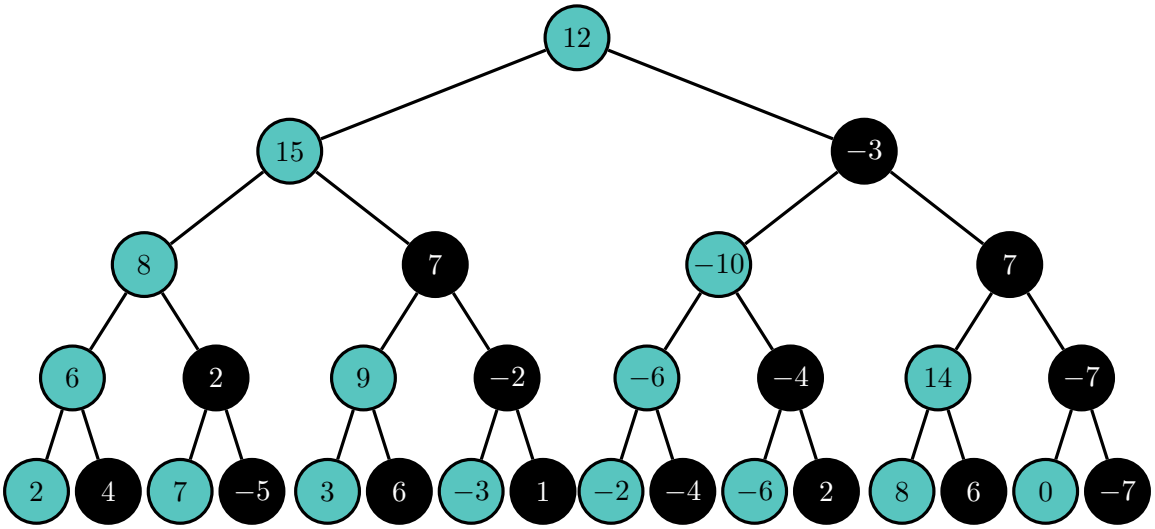
```
final int SZ = 1 << 17;
int[] sum = new int[2 * SZ]; // sum[1] contains sum for [1, SZ], and so on
int query(int i, int l, int r, int a, int b) {
    // i.e. query(1, 1, SZ, 7, 15)
    if(r < a || b < l) return 0;
    if(a <= l && r <= b) return tree[i];
    int m = (l + r) / 2;
    int ql = query(2 * i, l, m, a, b);
    int qr = query(2 * i + 1, m + 1, r, a, b);
    return ql + qr;
}
void update(int i, int l, int r, int p, int k) {
    // i.e. update(1, 1, SZ, 3, 2)
    if(r < p || p < l) return;
    if(l == r){
        sum[i] = k;
        return;
    }
    int m = (l + r) / 2;
    update(2 * i, l, m, p, k);
    update(2 * i + 1, m + 1, r, p, k);
    sum[i] = sum[2 * i] + sum[2 * i + 1];
}
```

6.3.1 Fenwick Tree

A *Fenwick tree*, or *binary indexed tree (BIT)*, is simply a faster and easier to code segment tree when the operator in question has an inverse. Unfortunately, it's not at all intuitive, so bear with me at first and let the magic of the Fenwick tree reveal itself later. In fact, it is so magical that Richard Peng hates it because it is too gimmicky. The key idea is to compress the data stored within a segment tree in a crazy way that ends up having a really slick implementation using some bit operation tricks.

As discussed earlier, the $+$ operator has an inverse, $-$. Therefore, there is an inherent redundancy, for example, in keeping track of the sum of the first $\frac{n}{2}$ elements, the sum of all n elements, and the sum of the last $\frac{n}{2}$ elements, as we do in the segment tree. If we are given only $\sum_{k=1}^{n/2} a_k$ and $\sum_{k=1}^{n} a_k$, we can find $\sum_{k=n/2+1}^{n} a_k$ easily using subtraction.

With this in mind, let's ignore every right child in the tree. We'll mark them as black in the diagram. After that, we'll write out the tree nodes in postfix traversal order, without writing anything whenever we encounter a black node.



2	4	7	−5	3	6	−3	1	−2	−4	−6	2	8	6	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

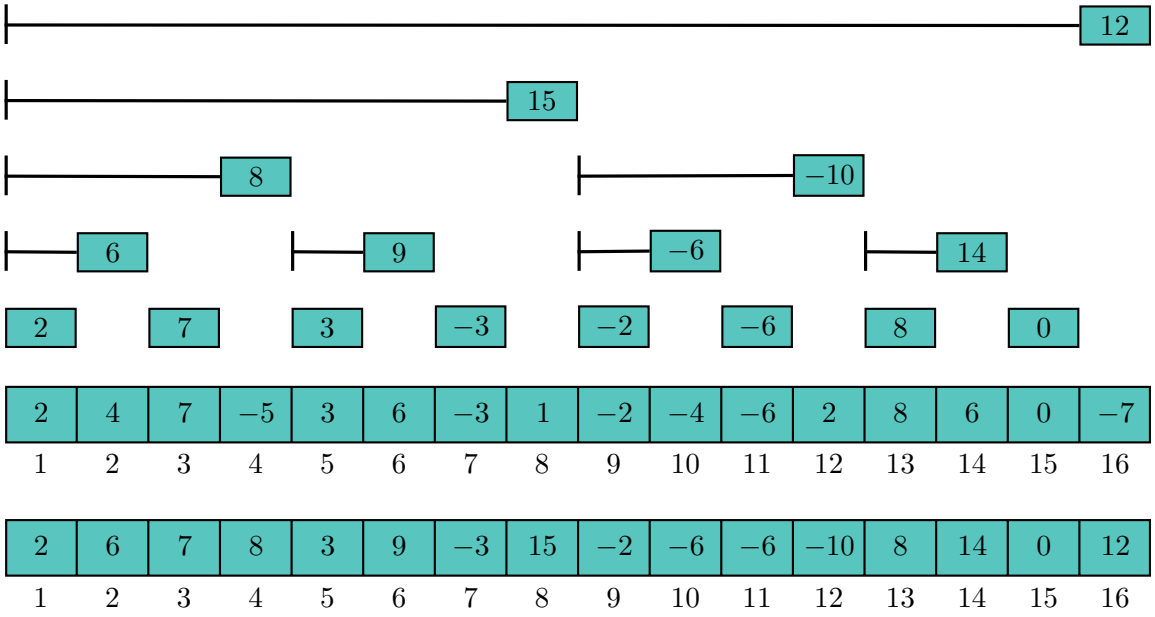
2	6	7	8	3	9	−3	15	−2	−6	−6	−10	8	14	0	12
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Our Fenwick tree is simply this last array. This should be quite confusing – it is not at all clear why this array resembles a tree, and the numbers in the array make no sense whatsoever right now.

Notice that the final position of every unblackened node is just the rightmost black child in its subtree. This leads to the fact that the ith element in the Fenwick tree array is the sum

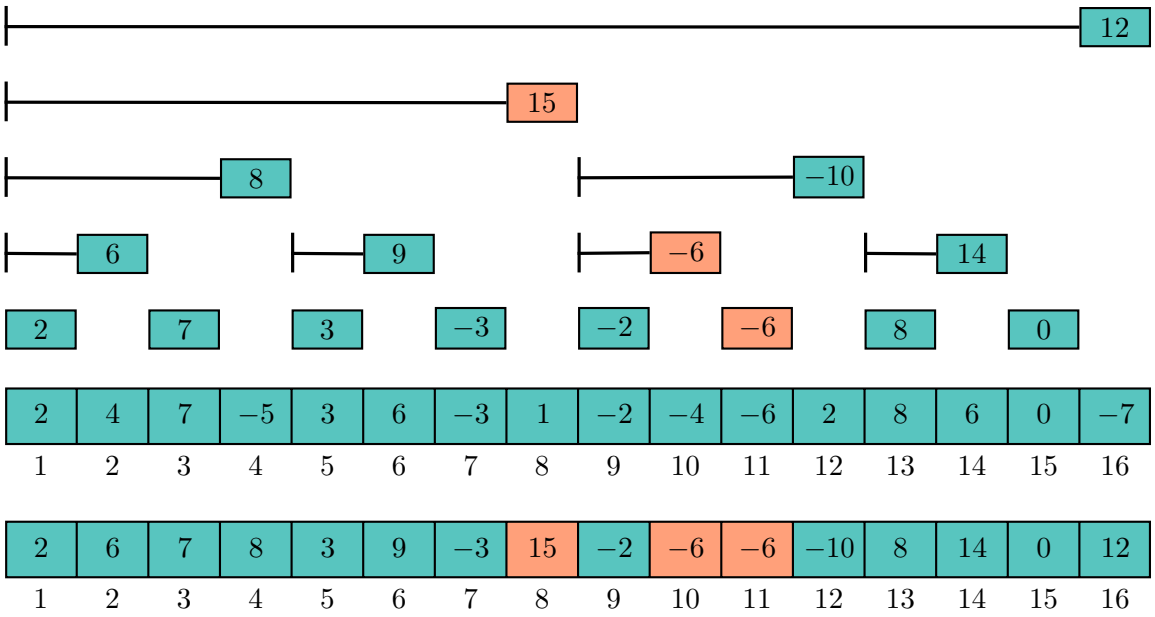
$$y_i = \sum_{k=i-2^{v_2(i)}+1}^{i} x_k,$$

where $2^{v_2(i)}$ is simply the greatest power of 2 that divides i. Let's look at a new diagram that hopefully will better illustrate this key property of the random array we just came up with.



All the framework is now in place. Now we need to find out how to query and update the Fenwick tree.

Suppose we wanted to find the sum $\sum_{k=1}^{11} x_k$. Let's take a look at the diagram to see which elements we need.

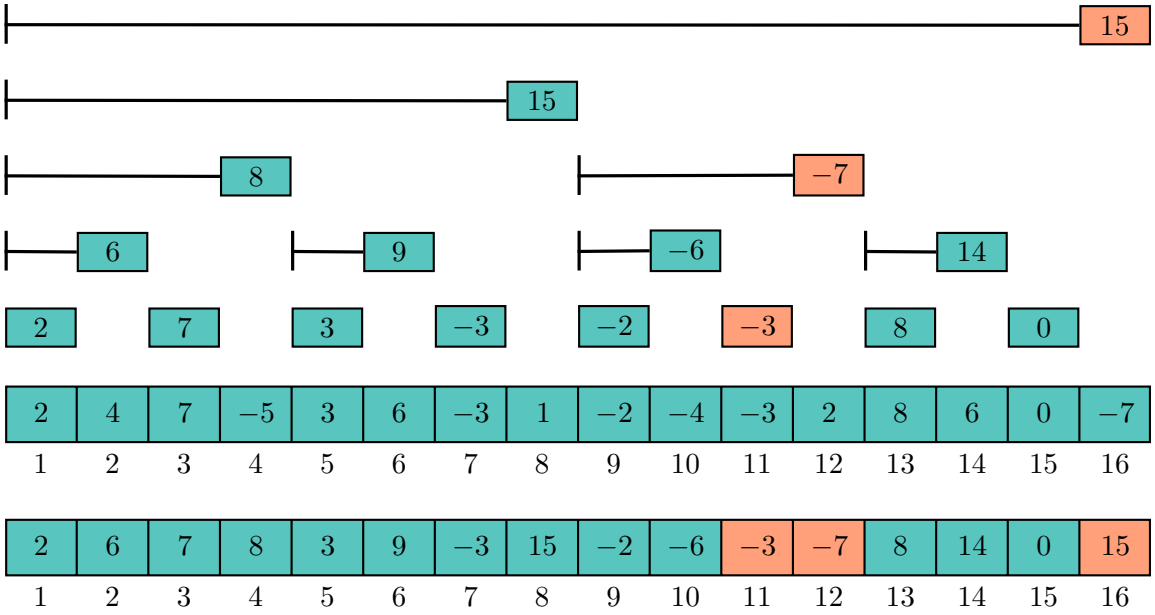


We see that the sum $\sum_{k=1}^{11} x_k = y_8 + y_{10} + y_{11}$. If we look at 11 in binary, we have $11 = 01011_2$. Let's see if we can find a pattern in these numbers in binary:

$$\begin{aligned} 11 &= 01011_2, \\ 10 = 11 - 2^{v_2(11)} &= 01010_2, \\ 8 = 10 - 2^{v_2(10)} &= 01000_2, \\ 0 = 8 - 2^{v_2(8)} &= 00000_2. \end{aligned}$$

So, we can simply subtract $11 - 2^{v_2(11)} = 10 = 01010_2$, find the sum of the first 10 elements, and add b_{11} to that sum to get the sum of the first 11 elements. We see that repeating this process takes off the last 1 in the binary representation of the number i, and since there are at most $\log n + 1$ 1s in the binary representation $\forall i \in [1, n]$, the query operation is $O(\log n)$.

And now for the update operation. Suppose we want to change the value of x_{11} from -6 to -3. Which numbers will we have to change?



We needed to increment the highlighted values, y_{11}, y_{12}, and y_{16}, by 3. Once again we'll look at 11, 12, and 16 in base 2.

$$\begin{aligned} 11 &= 01011_2, \\ 12 &= 01100_2 = 11 + 2^{v_2(11)}, \\ 16 &= 10000_2 = 12 + 2^{v_2(12)}. \end{aligned}$$

It appears that instead of subtracting the largest dividing power of 2, we are adding. Once again this is an $O(\log n)$ operation.

The real magic in the Fenwick tree is how quickly it can be coded. The only tricky part is finding exactly what $2^{v_2(i)}$ is. But it turns out, by the way bits are arranged in negative numbers, this is just `i & -i`. With this in mind, here's all the code that's necessary to code a Fenwick tree.

```
int[] y = new int[MAXN]; // Fenwick tree stored as array
void update(int i, int x) {
    for( ; i < MAXN; i += i & -i)
        y[i] += x;
}
int prefixSum(int i) {
    int sum = 0;
    for( ; i > 0; i -= i & -i)
        sum += y[i];
    return sum;
}
int query(int i, int j) {
    return prefixSum(j) - prefixSum(i - 1);
}
```

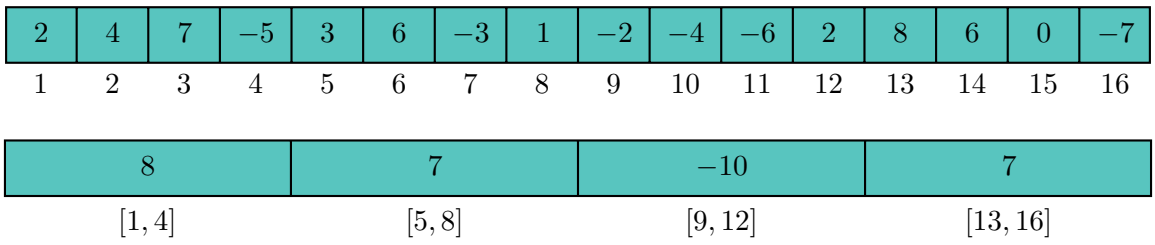
6.3.2 Lazy Propagation

It is often the case that in addition to performing range queries, we need to be able to perform *range updates*. (Before, we only had to implement point updates.) One extension of our sum problem would require the following two functions:

- $update(i, j, z)$ – increment the value of x_k by z for all $k \in [i, j]$
- $query(i, j)$ – return $\sum_{k=i}^{j} x_k$.

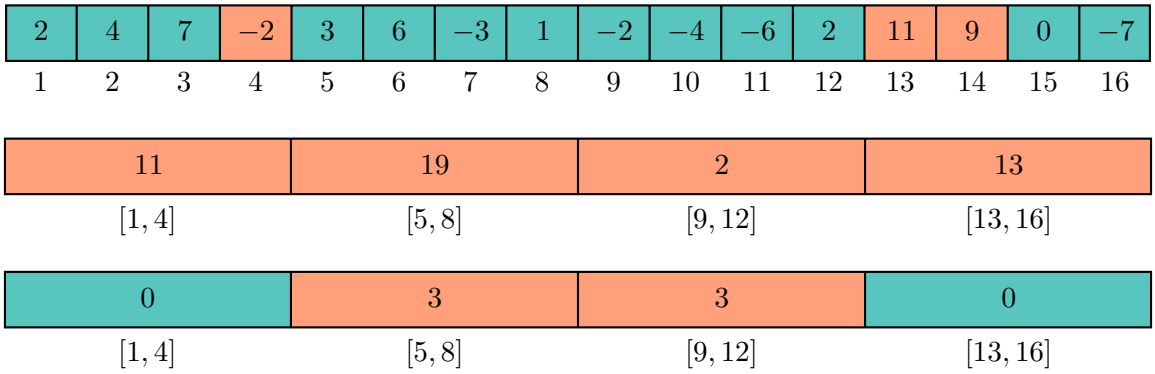
Some Motivation: $\sqrt{n}$ Blocking

Let's go back to our $\sqrt{n}$ blocking solution and see what changes we can make, and hopefully we can extend this idea back to our segment tree. If we're looking for an $O(\sqrt{n})$ implementation for *update*, we clearly can't perform point updates for all values in the range. The way we sped up *query* was by keeping track of an extra set of data, the sum of all the elements in a bucket, which we used when *the entire bucket was in the query range*.



What can we do with this idea for *update*? Let's see what we can do if an entire bucket were included in the update range. Again, we don't want to touch the original array a at all since that makes the operation linear. Let's try storing some information separately. This other information we're storing is the key idea behind lazy propagation.

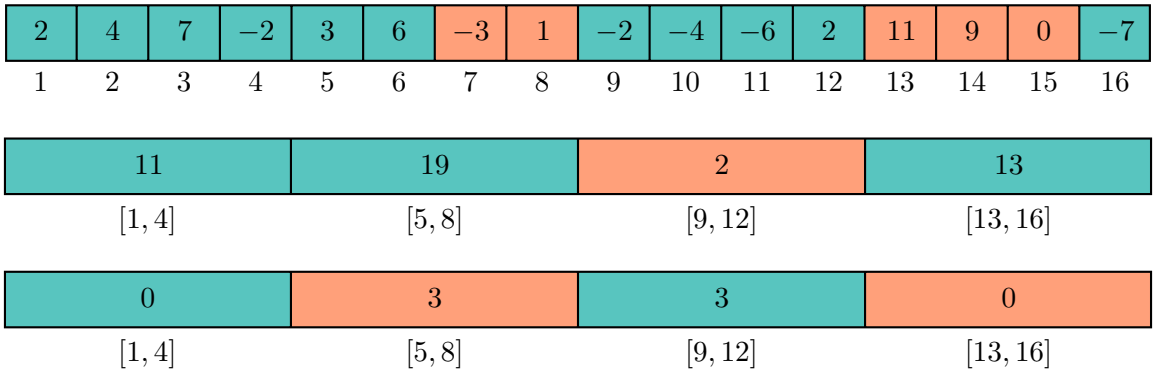
With this in mind, highlighted are the elements we'll need for $update(4, 14, 3)$.



Note that the sums associated with each bucket must adjust appropriately. In the example, there are four elements per bucket, so when an entire bucket needs to be incremented by 3, a single bucket can increment its sum by $4 \cdot 3 = 12$ easily.

To reiterate, we not storing the actual values of the elements where they were stored in solving the original formulation of the problem. Despite this fact, we are able to calculate what any single value is supposed to be. a_i is simply equal to the $\left\lceil \frac{i}{\sqrt{n}} \right\rceil$th value stored in the newest third array added to the ith value stored in the first array. Because of this, querying a range works in almost exactly the same way as it did in the original formulation.

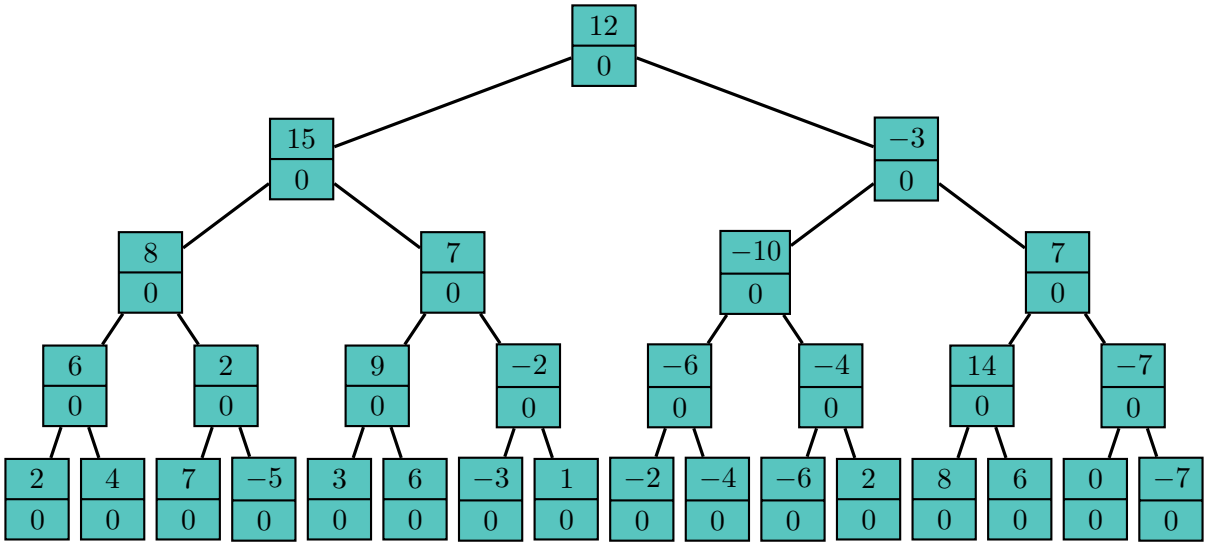
Highlighted are the values necessary for $query(7, 15)$.



Thus we have achieved $O(\sqrt{n})$ for both range updates and and range queries.

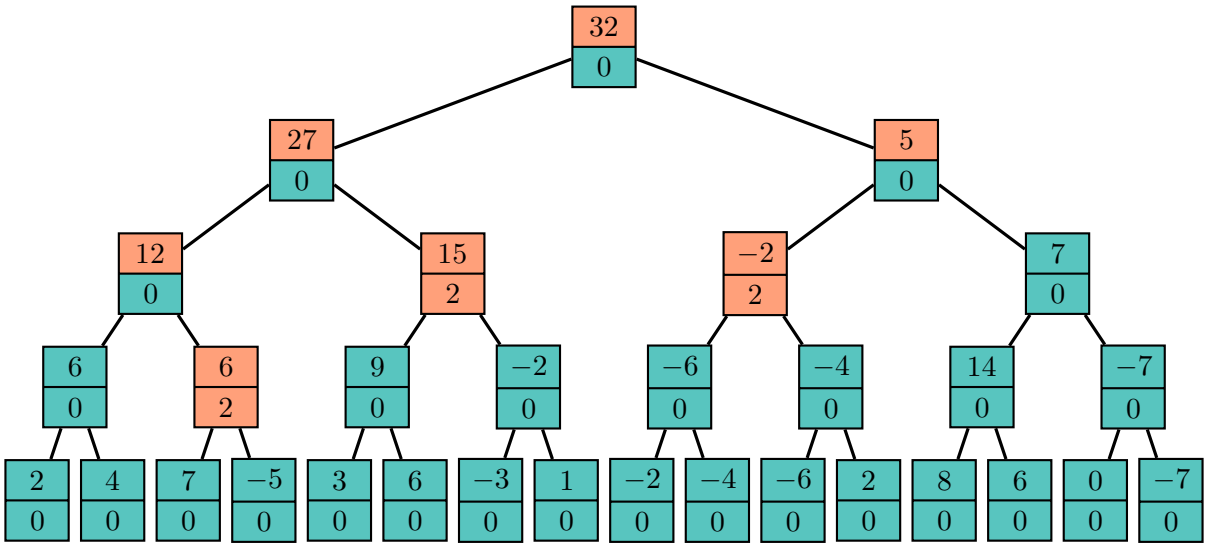
Lazy Propagation on a Segment Tree

Motivated by how we fixed our $O(\sqrt{n})$ solution, let's try adding a similar extra piece of information to our segment tree to try to get an $O(\log n)$ solution. Let's call this extra number the "lazy" number.



Once again, if the entire range associated with a node is contained within the update interval, we'll just make a note of it on that particular node and not update any of its children. We'll call such a node "lazy."

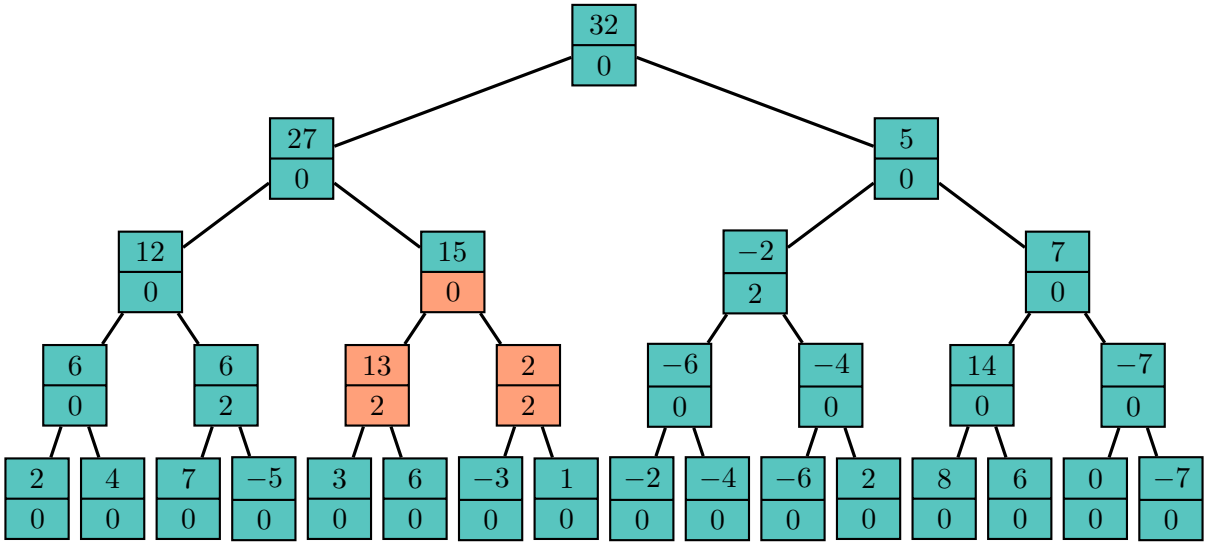
Here's the status of the tree after $update(3, 12, 2)$.



When a node is lazy, it indicates that the sum numbers of every other node in its subtree is no longer accurate. In particular, if a node is lazy, the sum number it keeps track of is not equal to the sum of the sum numbers of its children. This means whenever we need to access any node in that subtree, we'll need to update them then. Numbers in the tree therefore can change even after a query. Let's perform a query to illustrate that point.

$query(7, 13)$ requires access to the nodes for the ranges $[7, 8]$, $[9, 12]$, and $[13, 13]$. The nodes for $[9, 12]$ and $[13, 13]$ are up to date and store the correct sum, but the node for $[7, 8]$ does not, as it is a descendent of the node for $[5, 8]$, which has a nonzero lazy number. Highlighted are the nodes

we'll need to update as we perform the query. Notice how $[5, 8]$ simply passes its lazy number onto its children, and they update themselves as necessary.



Now, the nodes that we need are all up to date, and we can perform our query. This process is necessary whenever we are performing an operation on an interval that intersects but does not contain the range associated with a lazy node. Since this can only happen for two such nodes (one at either end of the interval in question), both updating and querying change values of at most four nodes per level, so both operations are $O(\log n)$.

6.4 Queue with Minimum Query

Suppose we wanted a list data structure with the following three operations:

- $add(x)$ – add x to the end of the list.
- $remove()$ – remove the first element in the list.
- $min()$ – return the minimum element in the list.

This is different from a heap since *remove* does not remove the minimum element. It's pretty easy to find a $O(\log n)$ solution using the data structures we already know. However, it is possible to build a data structure that can do any of these operations with complexity $O(1)$.

To solve this problem, we'll first solve an easier problem. Suppose instead of removing the first element of the list, we had remove the last element; in other words, we needed to build a stack with minimum query instead of a queue. This is a simple task; we'll just use a normal stack, but instead of storing single numbers, we'll store pairs. The pairs will each contain the number we're adding and the minimum element up to that point in the stack.

To build a queue given a stack with minimum query, we'll just have two stacks. When we add an element, we push it to the top of the first stack. When we remove an element, we take it off the

top of the second stack. The minimum element in the queue is the smaller element between the minima of either stack.

This seems like it obviously doesn't work – one of the stacks keeps growing, while the other can only shrink. This is not a problem, however; when the second stack runs out of elements, we'll just pop off every element of the first stack and push each onto the second stack. This amounts to one $O(n)$ operation for every n $O(1)$ operations, which averages out to constant time.

6.5 Balanced Binary Search Tree

Recall that a normal binary search tree is not guaranteed to be balanced. Many data structures have been invented to self-balance the binary search tree.

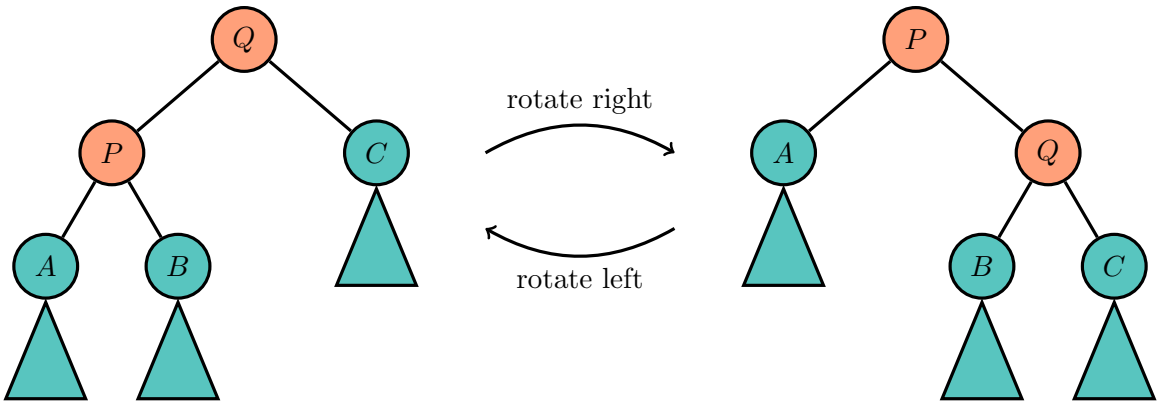
The *red-black tree* was an early development that represents the "just do it" approach, using casework to balance a tree. This results in a simple mathematical proof for the maximum depth of the tree. Unfortunately, coding these up are quite nasty do to the casework involved, so doing so is generally not preferred unless your name is Sreenath Are.

The *splay tree* is a data structure that guarantees amortized logarithmic performance; this means that any single operation is not necessarily linear, but over *any* sequence of length m for $m \geq n$, the performance for all m operations is $O(m \log n)$. Though the proof of the splay tree's complexity is necessarily more complicated than the red-black tree's complexity, as it requires amortized analysis, coding the splay tree is much easier than coding the red-black tree. In addition, the splay tree grants us more flexibility than the red-black tree provides, allowing us to build more complicated structures like link-cut trees using splay trees.

The *treap* is a probabilistic data structure that combines the BST with a heap to usually create a balanced tree. Just as quicksort is not guaranteed to run in $O(n \log n)$, treaps are not guaranteed to have logarithmic depth. In practice, however, the treap almost always has performance that is $O(\log n)$.

6.5.1 Tree Rotation

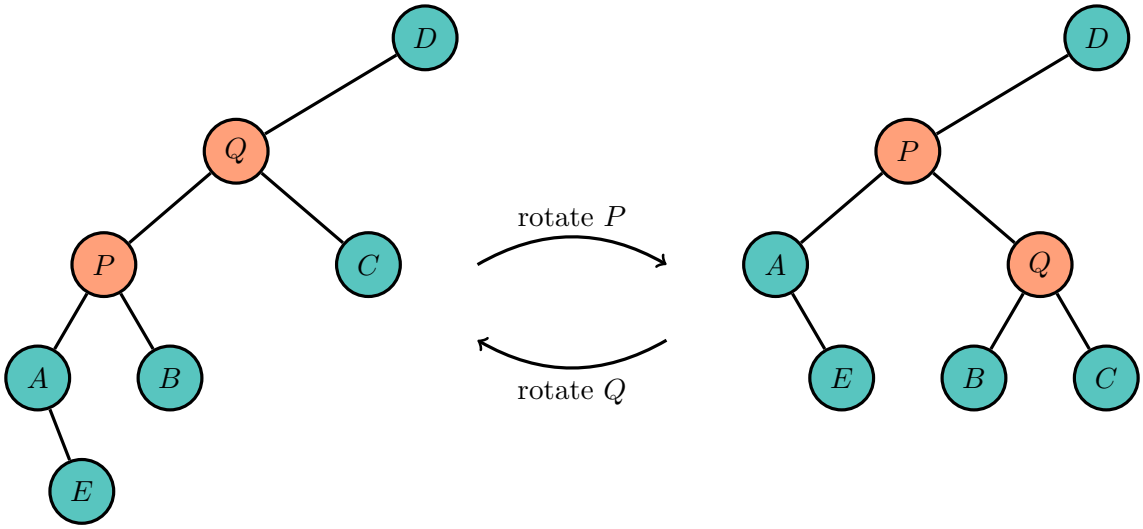
The key idea behind all of these data structures is the use of the *tree rotation*, which is used to change the structure of the tree but will not change the order of the elements (inorder). Maintaining the order of the elements is important, of course, if we wish our tree to remain a binary search tree.



Here, the triangles represent subtrees, as A, B, and C could very well have children of our own, but they are not shown in the diagram. Note that the inorder ordering of the nodes has not been changed.

When we rotate right, we literally take the edge connecting P and Q and rotate it clockwise. Then P becomes the parent of Q where before P was the child of Q. However, this definition of rotation is somewhat cumbersome, as we have different terms for the highly symmetrical rotating right and rotating left. The key characteristic a rotation is we move the lower node up one level. Thus, I prefer to think of tree rotation as whatever tree rotation, either left or right, will *rotate up* a node. In the diagram, rotating right is analogous to rotating P up, and rotating left is analogous to rotating Q up. Rotating a node up will change the tree such that its former parent is now its child.

The other notable change in the tree structure is the subtree associated with B passes between P and Q upon tree rotation. Finally, tree rotation can happen at any place in the tree, not just at the root. When we rotate P up, we must update the parent of Q to change its child to P.



Note that in this example, rotating up P decreases the total height of the tree. We want to somehow systematically rotate up nodes to accomplish this. The following data structures provide such a system.

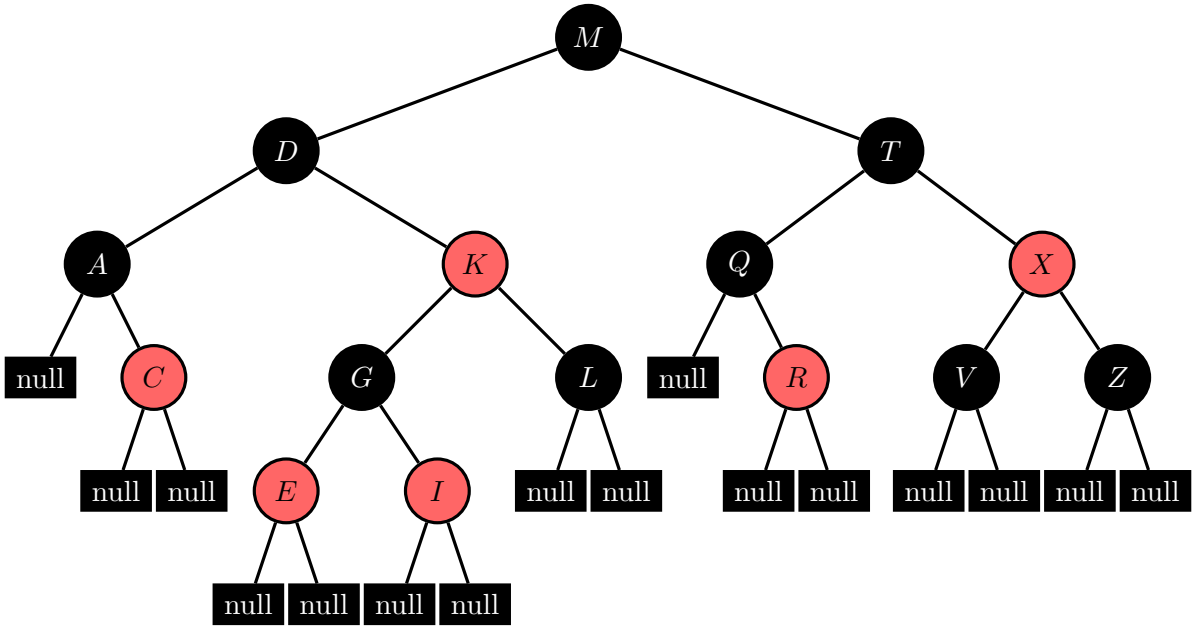
6.5.2 Red-Black Tree

As stated earlier, the red-black tree represents the "just do it" approach. We want to somehow bound the maximum length from the root to any leaf node by applying a constraint on the tree. In this case, our constraint is a coloring of the nodes (red or black) such certain properties are satisfied. For the red-black tree, the only nodes storing data are non-leaf nodes, and all leaf nodes store "null." The number of null nodes, then, is $n + 1$, where n is the number of non-null nodes. We do this so that any node storing useful information has two children.

We want our tree to satisfy the following properties:

1. The root is black.

2. All leaf nodes (null) are black.
3. Every red node has two black children. Consequently, a red node has a black parent.
4. Any path from the root to a null node contains the same number of black elements.



Note that every path from the root to a null node contains four black nodes.

The proof of $O(\log n)$ search follows immediately. The shortest possible path contains only black nodes, and the longest possible path contains black and red nodes alternating. Since the number of black nodes in both must be the same, any path is at most twice as long as any other path. As the number of nodes in the tree is $2n + 1$, the number of black nodes m in any path is then bounded below by $2^{2m} - 1 \geq 2n + 1$ and above by $2^m - 1 \leq 2n + 1$. Thus the height of the tree is on the order $O(\log n)$, and we are done.

Thus if our tree maintains its red-black coloring and satisfies the necessary properties, we can guarantee that our tree is balanced. We then consider the two ways we change the state of the tree, insertion and deletion. We can insert and delete in the normal way, but we might need to make changes to the tree after that to restore the red-black properties. We do this through a small number of color flips and tree rotations, which we can handle through casework.

Let's handle insertion first. When we insert a node, it takes the place of a black null leaf node. To maintain property 4, we must color the new node red, as it has two black null children. However, we may have violated some of the other properties, specifically 1 and 3.

We'll call the new node N, its parent P, its uncle (parent's sibling) U, and its grandparent G, if they exist.

We consider the following five cases.

1. N is the root node. That is, N is the first node added to the tree.

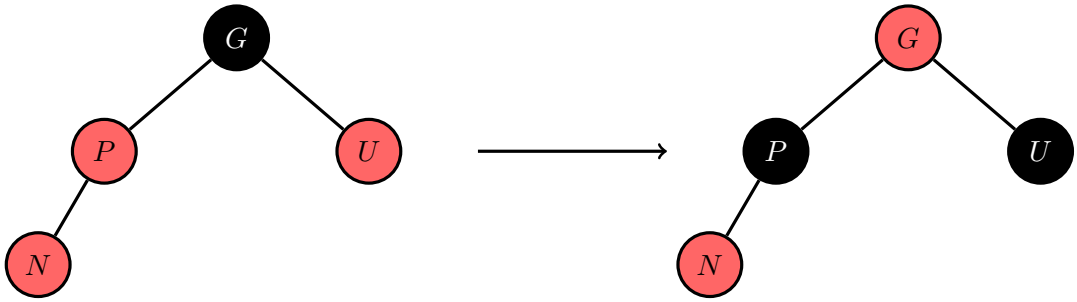
   It is easy to just change the color of N to red to restore property 1, and we are done.

2. P is black.

   Then property 3 is not violated, and we are done.

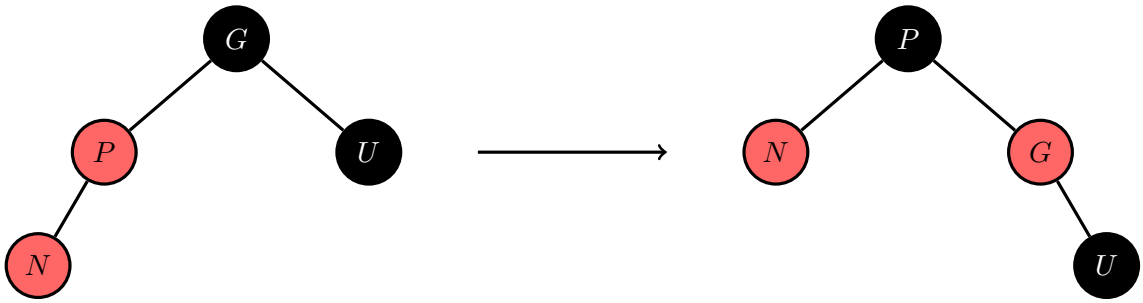
3. P is red, and U is red (G and U exist since P cannot be the root, as the root is black).

   As P and U are red, G is black. We simply color P and U black and G red to restore property 3.



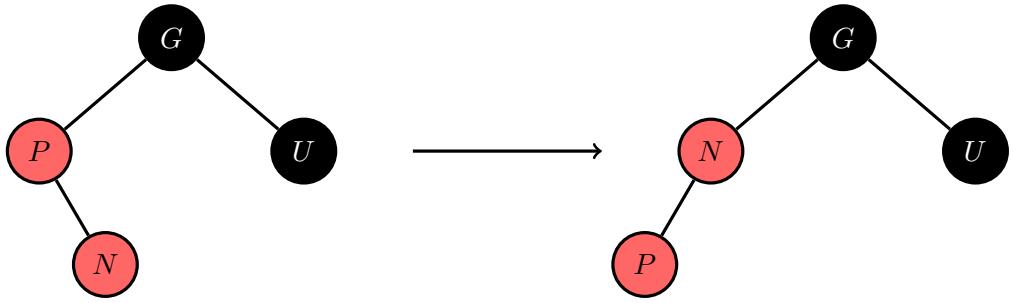
4. P is red, U is black, and N is on the same side of P as P is of G.

   G must be black. We rotate up P and color P black and G red.



5. P is red, U is black, and N is on the opposite side of P as P is of G.

   G must be black. We rotate up N, and this reduces to case 4.



Thus after we insert, we can always restructure the tree using a constant number of operations to restore the necessary red-black tree properties.

Now we need to work on deletion. Recall how deletion works on a normal binary search tree. If the node we need to replace has two children, we swap the node with the least element in its right subtree, which does not have two children. We then are able to remove that node more easily.

We can do the same thing with the red-black tree. If a node has two non-null children, we swap its value with the least non-null node in its right subtree and remove that node. Thus we reduce the deletion problem to the case where the node we need to remove has at least one null child.

Two cases are very easy.

1. The node we wish to remove is red.

   Then the node must have two null leaf nodes as children. Then we remove the node by replacing it and its children with a single null node.

2. The node is black and has a red child.

   We replace the node with its child and paint its child red.

The remaining case is the node black with two black null children. We'll first replace the node with a null node N. Then, all paths passing through N are one black node short compared to all other paths.

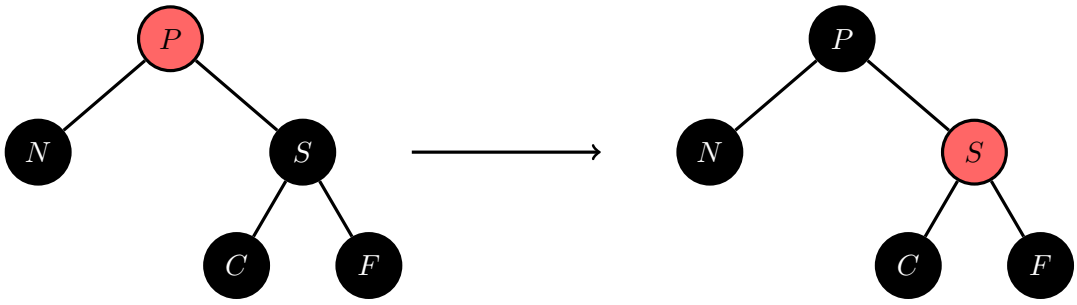
We denote the parent of N as P, its sibling S, and its sibling's children C and F, such that C is on the same side of S as N is of P, if they exist. That is, C is the "closer nephew" child, while F is the farther. We now describe a six-case balancing procedure on the black node N that fixes our problem.

1. N is the root.

   We are done, as every path possible must pass through N, so all paths are balanced.

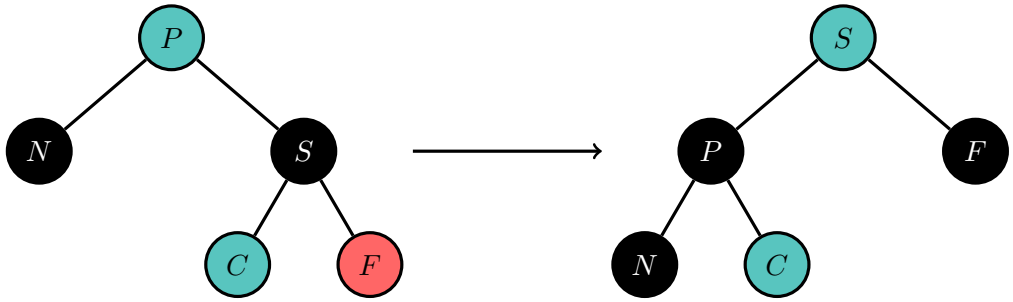
2. P is red and S, C, and F are black.

   Then we simply trade the colors of P and S. The number of black nodes for paths passing through S stays the same, but the number of black nodes for paths passing through N increases, so we are done.



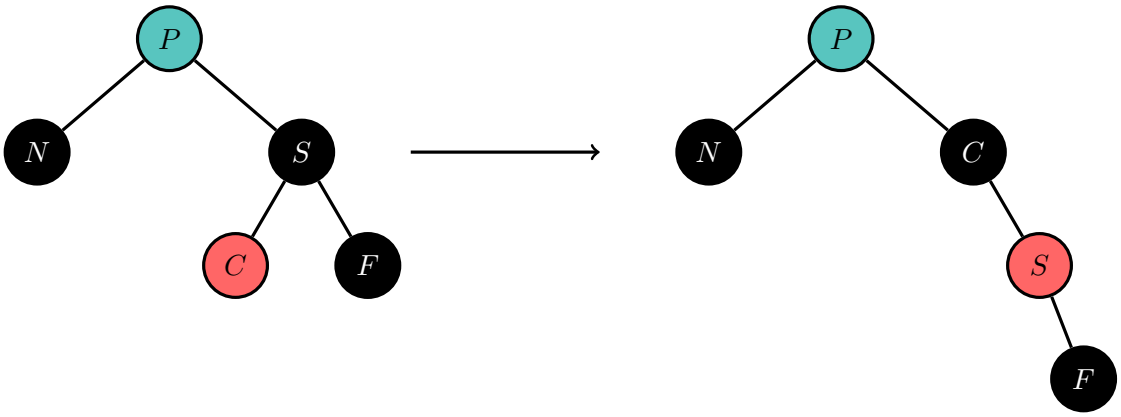
3. S is black and F is red.

   Then we color F black and rotate up S, before swapping the colors of P and S. Then paths passing through N gain a black vertex, while paths passing through C and F stay the same.



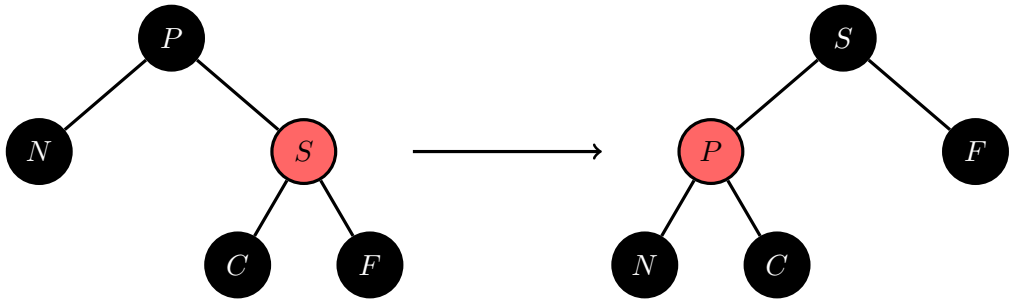
4. S is black, F is black, and C is red.

   Then we rotate up C and swap the colors of S and C. Then this reduces to case 3.



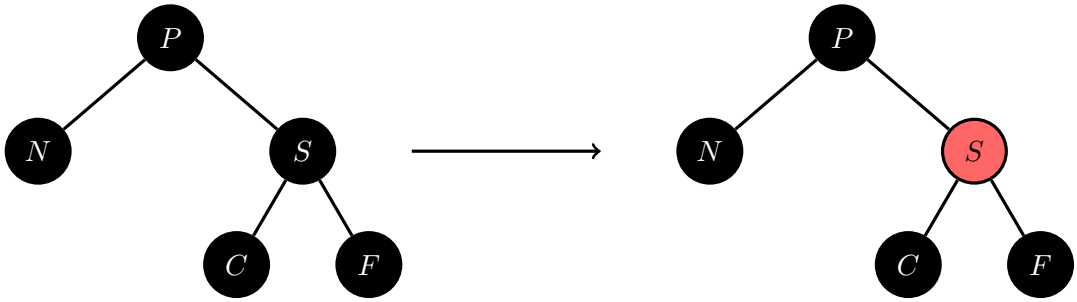
5. S is red.

   Then P, C, and F must be black. We rotate S up and swap the colors of P and S. C, the new sibling of N is black, so this then reduces to one of cases 2, 3, or 4.



6. P, S, C, and F are all black.

   Then we recolor S red. Now, all paths through P have the same number of black nodes, but each path through P has one less black node than each path not passing through P. Since P is black, and this procedure did not require that N be a leaf node, we can repeat this entire balancing procedure on P.



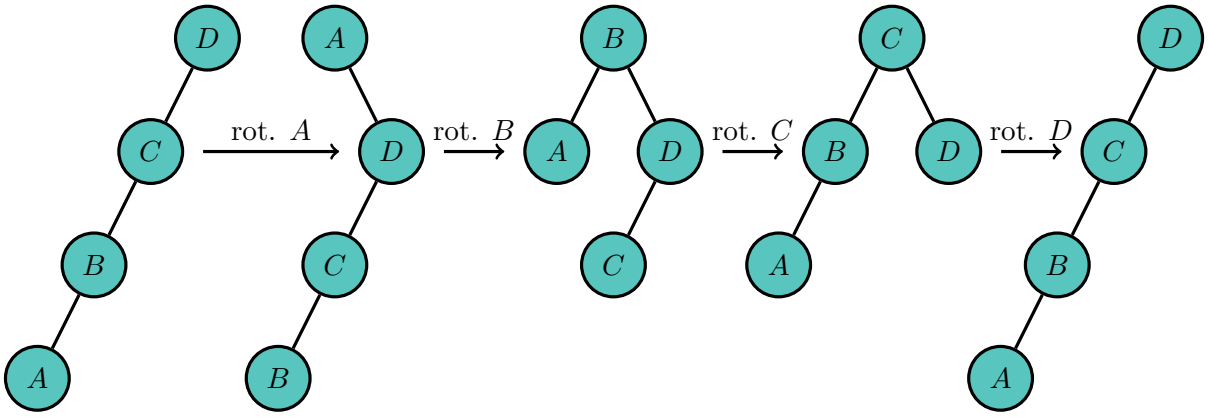
Unlike the balancing for insertions, the balancing for deletions has the potential to call itself on the parent node P. However, balancing following a deletion is still $O(\log n)$ worst case. It is possible prove that it is $O(1)$ amortized. Regardless, we now have a deletion algorithm that runs in $O(\log n)$ time.

Thus the red-black tree supports standard binary search tree operations, all in $O(\log n)$.

6.5.3 Splay Tree

Remember that rotating a node brings it one level closer to the root. The idea behind a splay tree is to apply a sequence of rotations that rotate a node all the way up to the root. The intuition for doing this is once we access a node, it is easy to perform other operations on it since it will simply be the root of the tree.

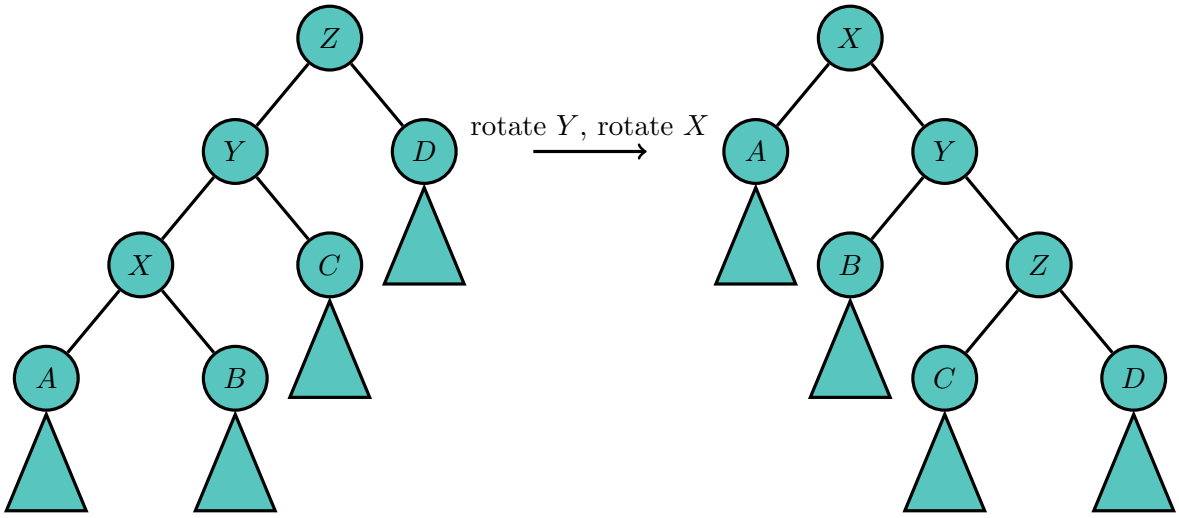
It is not at all clear that rotating nodes to the root will somehow improve the average performance of the tree. That's because rotating the normal way doesn't! To improve the performance of the tree, we expect that our balancing procedure tends to decrease the height of the tree. Unfortunately, simply repeatedly rotating a node up the tree doesn't exactly do that task. Consider a tree that looks liked a linked list, and rotating the nodes in the order A, B, C, D, as shown in the diagram. The height of the tree remains linear in magnitude, as at its smallest stage the tree is around $\frac{n}{2}$ in height, so accessing elements remains $O(n)$ on average.



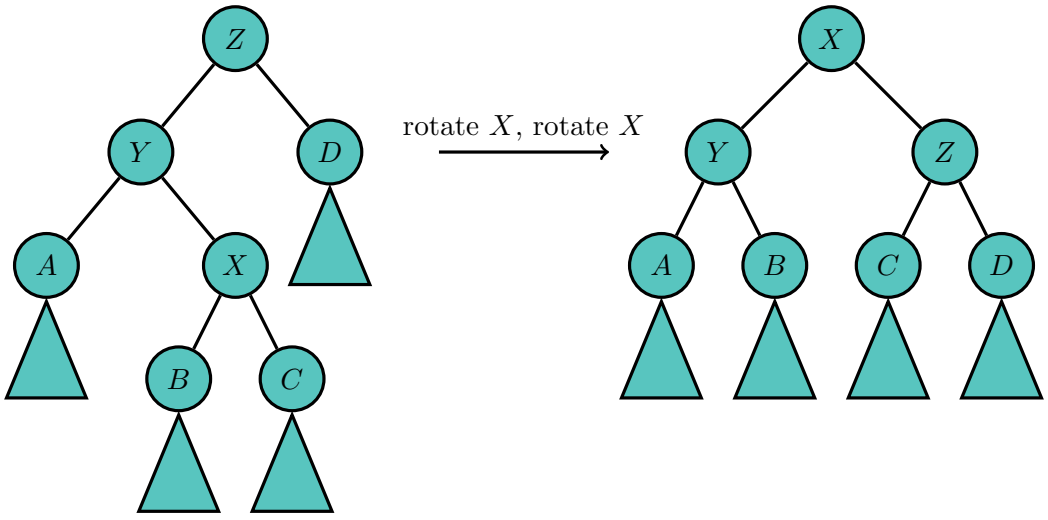
As discussed earlier, simply repeatedly applying the standard rotation clearly is not guaranteed to reduce the height of the tree on average. However, if we simply make one small change, magic happens. The trick of the splay tree is to rotate a node to the root in such a way that the tree has a

tendency to decrease in height. We'll use two compound rotations, and depending on the structure of the tree, we'll use a different one accordingly, to rotate a node to the root.

When a node is a left child and its parent is a left child, or the node is a right child and its parent is a right child, we first rotate up the parent, and then rotate up the node. This sequence of rotations is the only difference between splaying and rotating a node to the root using standard rotation.

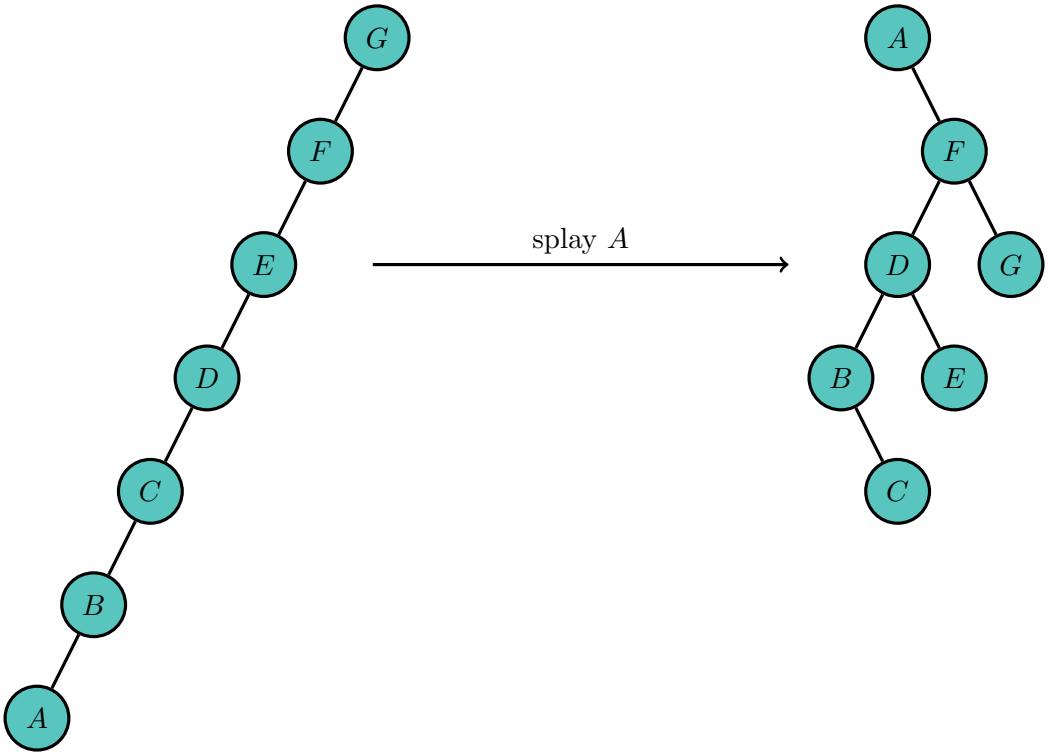


When a node is a left child and its parent is a right child, or the node is a right child and its parent is a left child, we rotate the node up twice, the normal way.



Finally, when a node's parent is the root, so it has no grandparent, we simply rotate up the normal way.

Rotating a node to the root in this way is called *splaying* the node. Thus, splaying a node brings it to the root of the tree. Splaying seems too simple to lead to an amortized $O(\log n)$ solution. To get a better idea of how splaying works, let's see what it does to a tree that looks like a linked list.



Splaying here cuts the height of the tree in half – a huge improvement. Splaying is not guaranteed to decrease the height of the tree, and it is possible for a sequence of splays to even result in a linked-list-like structure. However, given a number m at least equal to the maximum number of nodes n ever in the tree, *any* sequence of m splays runs in $O(m \log n)$. Splaying is then $O(\log n)$, amortized.[1]

Then, whenever we access a node, even in insertion and deletion, we splay that node to the top. This makes access, insert, and delete all $O(\log n)$ amortized.

Since splay trees need not satisfy a restrictive coloring, as red-black trees do, we have the freedom to completely change the structure of the tree at a whim. Recall how tedious the casework was to simply add or remove one node at a time in a red-black tree. Because the splay tree is simply a normal binary search tree that rotates nodes up upon access, we can detach an entire subtree from the tree and not worry about any properties of our tree no longer being satisfied.

For this reason, we define the *split* and *join* functions for splay trees.

Given two trees S and T, such that every element in S is less than every element in T, we join S and T into one tree by splaying the largest element in S, so that the root, which is now the largest element, has no right child. Then, we set its right child to the root of T, resulting in a single tree with elements from both S and T.

Given a tree and a value v, we split the tree in two by splaying the greatest element not greater than v to the root and detaching the right subtree from the rest. This results in two trees, one containing all elements at most v, and the other containing all elements greater than v.

[1] If you're curious to see the proof of this, a rather in depth explanation can be found here: `http://www.cs.cmu.edu/afs/cs/academic/class/15859-f05/www/documents/splay-trees.txt`

Both of these operations are $O(\log n)$ amortized. It is possible to implement insert and delete using split and join.

Since a binary search tree stores an ordered set, it is incredibly useful for storing a dynamic list of elements where the length can change and elements need to be added or removed from any point in the list, not just the beginning or the end. Splay trees are incredibly useful because of the split and join operations, as they allow us to remove consecutive elements from the list represented by our tree.

For example, to remove elements indexed in the range $[i, j]$, we split at index i and split at index $j + 1$ to get three splay trees, one representing $[1, i - 1]$, another $[i, j]$, and the last $[j + 1, n]$. We can then merge the first and third trees together, to get two trees, one representing $[i, j]$ and the other representing the original list with $[i, j]$ removed. A similar procedure can insert a list within another list as as contiguous block. Both of these operations can be completed in $O(\log n)$ amortized using a splay tree.

6.5.4 Treap

6.6 Fractional Cascading

6.7 Sweep Line