Intuitively, you can think of a binary indexed tree as a compressed representation of a binary tree that is itself an optimization of a standard array representation. This answer goes into one possible derivation.

Let's suppose, for example, that you want to store cumulative frequencies for a total of 7 different elements. You could start off by writing out seven buckets into which the numbers will be distributed:

```
[   ] [   ] [   ] [   ] [   ] [   ] [   ]
  1     2     3     4     5     6     7
```

Now, let's suppose that the cumulative frequencies look something like this:

```
[ 5 ] [ 6 ] [14 ] [25 ] [77 ] [105] [105]
  1     2     3     4     5     6     7
```

Using this version of the array, you can increment the cumulative frequency of any element by increasing the value of the number stored at that spot, then incrementing the frequencies of everything that come afterwards. For example, to increase the cumulative frequency of 3 by 7, we could add 7 to each element in the array at or after position 3, as shown here:

```
[ 5 ] [ 6 ] [21 ] [32 ] [84 ] [112] [112]
  1     2     3     4     5     6     7
```

The problem with this is that it takes O(n) time to do this, which is pretty slow if n is large.

One way that we can think about improving this operation would be to change what we store in the buckets. Rather than storing the cumulative frequency up to the given point, you can instead think of just storing the amount that the current frequency has increased relative to the previous bucket. For example, in our case, we would rewrite the above buckets as follows:

```
Before:
[ 5 ] [ 6 ] [21 ] [32 ] [84 ] [112] [112]
  1     2     3     4     5     6     7

After:
[ +5] [ +1] [+15] [+11] [+52] [+28] [ +0]
  1     2     3     4     5     6     7
```

Now, we can increment the frequency within a bucket in time O(1) by just adding the appropriate amount to that bucket. However, the total cost of doing a lookup now becomes O(n), since we have to recompute the total in the bucket by summing up the values in all smaller buckets.

The first major insight we need to get from here to a binary indexed tree is the following: rather than continuously recomputing the sum of the array elements that precede a particular element, what if we were to precompute the total sum of all the elements before specific points in the sequence? If we could do that, then we could figure out the cumulative sum at a point by just summing up the right combination of these precomputed sums.

One way to do this is to change the representation from being an array of buckets to being a binary tree of nodes. Each node will be annotated with a value that represents the cumulative sum of all the nodes to the left of that given node. For example, suppose we construct the following binary tree from these nodes:

```
             4
          /     \
         2       6
        / \     / \
       1   3   5   7
```

Now, we can augment each node by storing the cumulative sum of all the values including that node and its left subtree. For example, given our values, we would store the following:

```
Before:
[ +5] [ +1] [+15] [+11] [+52] [+28] [ +0]
  1     2     3     4     5     6     7

After:
                 4
               [+32]
              /     \
           2           6
         [ +6]       [+80]
         /   \       /   \
        1     3     5     7
      [ +5] [+15] [+52] [ +0]
```

Given this tree structure, it's easy to determine the cumulative sum up to a point. The idea is the following: we maintain a counter, initially 0, then do a normal binary search up until we find the node in question. As we do so, we also the following: any time that we move right, we also add in the current value to the counter.

For example, suppose we want to look up the sum for 3. To do so, we do the following:

- Start at the root (4). Counter is 0.
- Go left to node (2). Counter is 0.
- Go right to node (3). Counter is 0 + 6 = 6.
- Find node (3). Counter is 6 + 15 = 21.

You could imagine also running this process in reverse: starting at a given node, initialize the counter to that node's value, then walk up the tree to the root. Any time you follow a right child link upward, add in the value at the node you arrive at. For example, to find the frequency for 3, we could do the following:

- Start at node (3). Counter is 15.
- Go upward to node (2). Counter is 15 + 6 = 21.
- Go upward to node (1). Counter is 21.

To increment the frequency of a node (and, implicitly, the frequencies of all nodes that come after it), we need to update the set of nodes in the tree that include that node in its left subtree. To do this, we do the following: increment the frequency for that node, then start walking up to the root of the tree. Any time you follow a link that takes you up as a left child, increment the frequency of the node you encounter by adding in the current value.

For example, to increment the frequency of node 1 by five, we would do the following:

```
                 4
               [+32]
              /     \
         2           6
       [ +6]       [+80]
       /   \       /   \
    > 1     3     5     7
    [ +5] [+15] [+52] [ +0]
```

Starting at node 1, increment its frequency by 5 to get

```
                 4
               [+32]
              /     \
         2           6
       [ +6]       [+80]
       /   \       /   \
    > 1     3     5     7
    [+10] [+15] [+52] [ +0]
```

Now, go to its parent:

```
                 4
               [+32]
              /     \
       > 2           6
       [ +6]       [+80]
       /   \       /   \
      1     3     5     7
    [+10] [+15] [+52] [ +0]
```

We followed a left child link upward, so we increment this node's frequency as well:

```
                 4
               [+32]
              /     \
       > 2           6
       [+11]       [+80]
       /   \       /   \
      1     3     5     7
    [+10] [+15] [+52] [ +0]
```

We now go to its parent:

```
               > 4
               [+32]
```

```
             /     \
          2           6
        [+11]       [+80]
        /   \       /   \
       1     3     5     7
     [+10] [+15] [+52] [ +0]
```
That was a left child link, so we increment this node as well:

```
                4
              [+37]
             /     \
          2           6
        [+11]       [+80]
        /   \       /   \
       1     3     5     7
     [+10] [+15] [+52] [ +0]
```
And now we're done!

The final step is to convert from this to a binary indexed tree, and this is where we get to do some fun things with binary numbers. Let's rewrite each bucket index in this tree in binary:

```
               100
              [+37]
             /     \
         010         110
        [+11]       [+80]
        /   \       /   \
      001   011   101   111
     [+10] [+15] [+52] [ +0]
```
Here, we can make a very, very cool observation. Take any of these binary numbers and find the very last 1 that was set in the number, then drop that bit off, along with all the bits that come after it. You are now left with the following:

```
             (empty)
              [+37]
             /     \
          0           1
        [+11]       [+80]
        /   \       /   \
       00   01     10   11
     [+10] [+15] [+52] [ +0]
```
Here is a really, really cool observation: if you treat 0 to mean "left" and 1 to mean "right," the remaining bits on each number spell out exactly how to start at the root and then walk down to that number. For example, node 5 has binary pattern 101. The last 1 is the final bit, so we drop that to get 10. Indeed, if you start at the root, go right (1), then go left (0), you end up at node 5!

The reason that this is significant is that our lookup and update operations depend on the access path from the node back up to the root and whether we're following left or right child links. For example, during a lookup, we just care about the left links we follow. During an update, we just care about the right links we follow. This binary indexed tree does all of this super efficiently by just using the bits in the index.

The key trick is the following property of this perfect binary tree:

Given node n, the next node on the access path back up to the root in which we go right is given by taking the binary representation of n and removing the last 1.

For example, take a look at the access path for node 7, which is 111. The nodes on the access path to the root that we take that involve following a right pointer upward is

- Node 7: 111
- Node 6: 110
- Node 4: 100

All of these are right links. If we take the access path for node 3, which is 011, and look at the nodes where we go right, we get

- Node 3: 011
- Node 2: 010
- *(Node 4: 100, which follows a left link)*

This means that we can very, very efficiently compute the cumulative sum up to a node as follows:

- Write out node n in binary.
- Set the counter to 0.
- Repeat the following while n ≠ 0:
    - Add in the value at node n.
    - Remove the leftmost 1 bit from n.

Similarly, let's think about how we would do an update step. To do this, we would want to follow the access path back up to the root, updating all nodes where we followed a left link upward. We can do this by essentially doing the above algorithm, but switching all 1's to 0's and 0's to 1's.

The final step in the binary indexed tree is to note that because of this bitwise trickery, we don't even need to have the tree stored explicitly anymore. We can just store all the nodes in an array of length n, then use the bitwise twiddling techniques to navigate the tree implicitly. In fact, that's exactly what the bitwise indexed tree does - it stores the nodes in an array, then uses these bitwise tricks to efficiently simulate walking upward in this tree.

Hope this helps!