Introduction:

This is an article primarily to myself, to clarify my understanding of the Binary Indexed Tree (BIT). Maybe some competitve programmers that want insight into how BITs actually work may also benefit from it. But I would really appreciate feedback, because this is something that really makes me uncomfortable. This article presents the BIT scheme that I do prove runtimes for, but the main goal is to provide intuition behind and motivation for the BIT scheme.

The BIT is something I have felt shaky about and cowered in fear when seeing it on LeetCode. So in this article, I'll talk myself into reinventing the BIT scheme to placate future me.

Problem Statement:

I have a list of size N. I want to support 2 operations on the list: Range(X) and Update(X, V). Range(X) returns the sum of the first X values in the list: List[1] + List[2] + … + List[X]. And Update(X, V) updates the Xth entry of the list to the value V. Sitenote: Range(X) supports a Range(I, J) operation that returns List[I]+List[I+1]+…+List[J] since: Range(I, J) = Range(J) - Range(I-1)

Premise:

Let's work in a "universe" of a list of size 15, 1-indexed. So N = 15. All numbers, then, will be from 1-15 and can be expressed using 4 bits from 0b0001 to 0b1111. And Log will mean Log base 2. So given I'm using 4 bits, I have the bound: Log(X) <= Log(N) < Log(N+1) = Log(16) = 4.

2 Naive Approaches and Buckets

There are 2 approaches on opposite ends of runtime spectrums: vanilla list and prefix sum.

Vanilla list has O(N) Range and O(1) Update whereas prefix sum has O(1) Range and O(N) Update. So depending on the ratio of Range:Update, one scheme may be better than the other if it's skewed enough to amortize the linear operation. But, say there are equal, 50-50 split of Range and Update operations. Then both these operations on average will be O(N).

Vanilla list is just the list itself. Update(X, V) is simply setting List[X] = V and done. And Range(X) adds together the first X elements, so that is O(N).

Prefix sum is building an auxilary list, prefixsum, where the Ith element is Range[I]. Creating this list is done in O(N), single pass from left to right, sweep. Set prefixsum[1] = List[1] and then, from I spanning from left to right from 2 to N, prefixsum[I] = List[I] + prefixsum[I-1]. This is the most elementary example of dynamic programming. So clearly prefix sum supports an O(1) Range approach because Range(X) = prefixsum[X]. But the tradeoff is now Update is O(N) because now to implement Update(X, V) first compute dV = V - List[X] where dV is the change in the Xth value. And then increment the O(N) elements prefixsum[X], prefixsum[X+1],…,prefixsum[N] all by dV.

The goal, then, is to break these expensive Range and Update tradeoffs both approaches have by compromising and having a new approach, the BIT scheme, perform both operations in O(Log(N)). Here, I would like to introduce the word and the idea of "bucket".

A bucket stores the sum of certain elements in the original, input List. And these "certain elements" are completely specified by their incides. So to me, I view buckets as a set of indicies. And for brevity, for the rest of this article I may refer to List[I] simply as I when talking about buckets. For example a bucket containing I and J means that bucket stores the value that is List[I] + List[J]. Conceptually, simply imagine the list I'm working with as: [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16], so List[I] = I. (Yes, updates be damned). Continuing the spirit of being imprecise, let me define the size of a bucket as how many indices the bucket contains, or more precisely, how many elements it sums. Of course the actual bucket contains a single integer that is the sum.

Ok, so both prefix sum and vanilla list approaches have N buckets. I identify 2 guiding questions: what is the size of the bucket and for a given index, which buckets does it land into?

The prefix sum approach has these buckets be lined up in the auxilary, prefixsum, list. The Ith bucket stores the sum of the first I elements so the size, then, of buckets in the prefix sum approach is is O(N). And for a given index I, I is a member of O(N) buckets that need to be updated when List[I] is Update'd. These buckets are prefixsum[I], prefixsum[I+1],...,prefixsum[N].

As for the vanilla list approach, will the input List itself has all the N buckets as elements of the list. These are singleton buckets where the Ith bucket is simply contains List[I]. So the size of the Ith bucket is O(1) and literally 1 as the only member is I (or List[I]). And conversely, a given index I only lands into 1 bucket, the Ith bucket. So also O(1) for the second guiding question.

The two bucket-guiding questions give intuition for the runtimes of Range and Update, respectively. The larger the bucket size, the less buckets need to be summed up to compute Range(X). Since every bucket a given index X is a member of needs to be updated during Update(X, V), it is not ideal for indices to contribute to too many buckets.

Preliminary Thoughts

Two more related guiding questions: 1) For computing Range(X), which buckets do we use to compute this? 2) For computing Update(X, V), for a given index X, which buckets do we place it in? And for both we want to invole O(Log(N)) buckets, so that the Range and Update operations will be O(Log(N)).

Range LogN

Consider 1) the buckets themselves. I want to introduce 2 concepts: prefixes and branching and then hint (well, blatantly, heavy-handedly) at the BIT scheme satisfying these.

Say we partition X into buckets {1-A} | {A+1 - B}| {B+1 - X}. To compute Range(X) we sum the values of these 3 buckets, where bucket(I - J) holds the sum of [I + (I+1) + ... + J] (again this is shorthand for sum of [List[I], List[I+1],...,List[J]]. For a given X, we want O(Log(N)) buckets

Compare this to the vanilla list approach to compute Range(X): both use prefix sums. In this Log(N) approach, to get Range(X) I do: bucket {B+1 - X} + Range(B) where, again, bucket{B+1 - X} stores the value of the sum of [B+1, B+2,...,X]. And Range(B) in turn I compute as bucket{A+1 - B} + Range(A) and Range(A) in turn is bucket(1-A}. So there is a prefix nature of these Range(I) subproblems: Range(A) being a prefix to Range(B) being a prefix to Range(X). I view bucket {A+1 - B} as a "child" of bucket{1-A} that extends the prefix that is Range(A) to the prefix that is Range(B). And lastly, bucket {B+1 - X} as a "child" of bucket {A+1 - B} that in turn extends the prefix that is Range(B) to Range(X). So these buckets build upon each other. Now consider the vanilla list O(N) approach to computing Range(X): Range(X) = bucket{X} + Range(X-1) where bucket{I} is the singleton bucket {I - I}, that is it simply stores the value I. Now Range(X-1) in turn I compute as bucket(X-1) + Range(X-2) and so I have X + (X-1) + Range(X-2) and keep unrolling the Ranges till I get 1 + 2 + ... + X. This has the same prefix nature: Range(X-2) is the prefix extending to X-2 and then the singleton bucket{X-1} extends that to get Range(X-1) and finally bucket{X} extends Range(X-1) to get Range(X). Bucket{X} is child of Bucket{X-1} is child of Bucket{X-2} and so forth.

So both this vanilla list and this Log(N) approach essentially operate under the same prefix extending mechanism. Main difference seeming to be vanilla list is O(N) and uses X terms/buckets to compute Range(X) whereas a proposed Log(N) approach uses only Log(N) buckets. I will now contemplate potential implications of a Log(N) scheme and how it would likely be different from the vanilla case: namely, the key idea of branching as opposed to the linear, non-branching approach

that is the vanilla list. Upon careful examination of this scheme, I believe it is natural that "branching" happen (TODO think about this, REQUEST FOR FEEDBACK) First point is that under the Log(N) approach, not all of the buckets can be singleton buckets like the vanilla list case. Because the Log(N) buckets need to cover all X numbers, 1-X. Let's say the second bucket {A+1 - B} is non-singleton and contains some J != B, that is, A+1 <= J < B. Important: Consider Range(J). I believe it is "natural" to reuse the first bucket {1-A} to get the prefix of Range(A) and then take some other path of buckets to get to Range(J). Note that we can't use the seond bucket of {A+1 - B} becuase that would overshoot J. So the idea is that if a child of a bucket B is non-empty, it cannot be an only child and bucket B branches to more than one child where, again, all the children get to reuse the contents of bucket B which in turn reusues of its parents and so forth creating a happy prefix chain (of hopefully length <=Log(N)).

Actually, just as I write this, I realize this suggests why a binary recursive scheme may be natural. This may be getting a bit ahead of myself, and next paragraph I will give a tiny preview of the BIT scheme, but these prefix and branching notions gave me an idea. TODO review this as it's stream of conciousness, evaluate content/correctness and clarity/notation/try being consistent. Say we follow some Prefix of buckets in a chain that covers Range(A) and then we reach the bucket {A+1 - B} that is non-singleton. So it's parent bucket, P, ends in A and is the last node on the prefix bucket chain that, together, covers Range(A). From last paragraph discussion on branching to cover all elements, I believe this forces branching. Taking this bucket will jump to B, but intermediate ranges [A+1 - J] for A+1 <= J < B must be accounted for. Say there are K such J's (K = B - 1 - A) Then say we have some strategy, S, to build a subtree that is also a child of the parent bucket P (so this entire subtree is a sibling to the subtree rooted at bucket {A+1 - B}) And its job is to cover all these K range queries: Range{A+1}, Range{A+2},...,Range{B-1}. Now consider the descendants of the bucket {A+1 - B}, or the subtree rooted at that bucket. This bucket itself answers Range(B) when extending its prefix chain and its descendants would answer range queries Range(J) for J>B for some, but not necessarily all, J. A thought experiment is what if this subtree rooted at the bucket {A+1 - B} would answer K more range queries: Range{B+1}, Range{B+2}, ... , Range{B+K}. If I assume the S is an "optimal" strategy for creating a subtree to answer K range queries, it feels natural to me to reuse this same S to create the subtree rooted at {A+1 - B}. Essentially, this suggests that a recursive approach at least makes sense to me. And a binary one at that because the parent bucket, P, has 2 children: a left subtree (say with a dummy root) that handles K Range queries and a right subtree that is rooted at bucket {A+1 - B} which handles K+1 Range queries (K for the descendants and +1 for Range(B) that bucket{A+1 - B} the root handles).

So for the BIT scheme for N=15, we have a bucket {1-8} so Range(J) for 1 <= J <= 7 have some strategy and this same strategy applies for Range(K) for 9<=K<=15. Basically, recursive nature of this arrangement. I'm definitely getting ahead of myself, but I can almost declare completion here itself. Range(X) is Log(N) because 3 cases: simplest is X = 8 so Range(8) = bucket{1-8} and done, 1 <= X < 8 case recurse the left subtree so instead of universe [1,15] the search is now in universe [1,7] and we have eliminated half the buckets so given there are O(N) buckets total, this will be O(Log(N)) if the recursion keeps eliminating half the buckets. Which is does because the last case is is 8 < X <= 15 so universe has halved to [9, 15] (after incorporating bucket{1-8} into the sum what is left is [9,15]) and Range(X) is computed in the same manner as would Range(X-8) except there is an offset of bucket{1-8} and recurse on the right subtree rooted at the bucket {1-8}. So the recursion on the right subtree for Range(X) is mechanistically the same as recursion for Range(X-8) on the left subtree except I have do O(1) extra work that is adding bucket{1-8} to the answer. Thus Range(X) in this potential binary bucket tree approach will be O(Log(N)). Quick note: the crucial idea here is a symmetry where bucket {1-8} has a sibling subtree that follows the same strategy as the descendants of bucket {1-8}, meaning by symmetry, the number of buckets in the [1,7] "left universe" and the

same as the number of buckets in the “right universe”, so at most there are Log(N) steps given that each step halves the number of buckets that can be considered on the bucket chain that computes Range(X).

Let me restate to explain the binary search, halving, logarithmic nature. Range(X) is computed by traversing a chain of buckets and while in the middle of the chain, there is some prefix already computed upto Range(P) and what’s left is Range(P+1, X) or the remaining range. The elements the Range sums for Range(S, E) is E - S + 1 and let me call this number the range “width”. And until Range(X) is completely formed, there is at least 1 more bucket to take. and taking a bucket means adding its value to the sum and recursing right as this bucket is at the root of the right subtree. Buckets are designed such that they cover half(1 more than half) the indices and their descendant buckets cover the remaining half. Then each iteration after adding a bucket to the answer, the remaining “width” contracts by half.

I’m done with 1) but I just want to throw in a preview of the BIT scheme that respects a potential Log(N) scheme that uses Log(N) buckets to partition X. We want to use Log(N) buckets. Presumably larger Xs will require more buckets. Largest number is 15, so hypothetically being “inspired” by binary representation of 15 as 0b1111 and to demo the BIT scheme, I’ll use the buckets {1-8} | {9-12} | {13-14} | {15} to compute Range(15) which is exactly what BIT does. Where each 1 in the binary representation has a corresponding bucket with size commensurate with the place of that 1. So take the number 11 or 0b1011. Range(11) will use buckets {1-8} + {9-10} + {11}. Since these are just preliminary thoughts, the take away is that whatever scheme I use, I need at most Log(N) buckets which this binary scheme complies with. Why? Well X<=N has log(X) bits and at most all of them will be “1″s and in this scheme, the number of buckets Range(X) uses are the number of 1 bits in the binary represenation of X. And note that both Range(15) and Range(11) use the same {1-8} bucket and extend it (bucket {1-8} is a parent to bucket {9-12} and {9-10} in the BIT scheme), so this conveniently agrees with aforementioned concepts of reusing prefixes and branching. And also note that going down the chain of buckets, the size of the buckets at least halves, as does the remining range width, again consistent with a proposed binary, Log(N) scheme.

Update LogN

Whew, that finished O(Log(N)) Range. Now consider 2): which buckets I place the index/value X in for the other O(Log(N)) operation, Update. So every bucket I place X in, I will have to adjust its value if I were to execute an Update(X) value. So at most I can place X in 4 buckets as log(16) = 4 > log(X) so the Update(X) operation will have to touch at most 4 buckets.

Contrast this placing X in Log(N) buckets approach with the naive prefix sum approach in where insteand of Log(N) buckets, X is placed in O(N) buckets: bucket X, bucket X+1,…, bucket N.

Say X falls in multiple buckets. Now when I compute Range(K) for K>=X, I cannot double count X. So whatever chain of (Log(N)) buckets Range(K) uses, it must use exactly 1 of these buckets. I also observe the following related constraint: to avoid double counting any indices, whenever an index X is a member of a bucket, it may not be a member of any descendant buckets. Buckets only extend prefixes of the parent chain of buckets leading to a given bucket, extension with new indices only, no double counting already covered indices.

Now if I contemplate the aforementioned binary recursive partitioning scheme that I showed as being Log(N) Range, this constraint of having no descendants of a bucket contain any of the indices that bucket contains also enforces Update to be Log(N) as well in a very similar manner. Starting from the top of the binary bucket tree at dummy node that is an empty range prefix, there are decisions to make at each node before going down a level, and there are Log(N) levels. So at each node (and the fact that we reached a given node means X is greater than the prefix covered thus far.

Let me say that so far, the amount A has been covered), there is a left subtree of handling K range queries and a right subtree rooted at a bucket{A+1-B} with size K+1 and whose descendants cover an additional K queries.

If X > B, then bucket{A+1-B} and the entire left subtree will be ruled out so over half the candidate buckets are precluded from containing X, because X is too large and must then be in the descendants of bucket{A+1-B}. If X = B, then X is by definition in bucket{A-B} and like the previous case, it cannot be in the left subtree because the maximum that can cover is up to Range(B-1). And X cannot be in any of the descendants of bucket(A+1-B} to avoid double counting it. So this is infact the terminal case, the last bucket X will be a part of. Finally, if X < B it will show up in bucket{A-B} because, following this making-a-decision-at-each-node-and-descending-down-the-tree procedure, X is greater than the prefix so far, meaning X > A so X lies in bucket{A+1-B} again by definition and it may also show up multiple times in the left sub tree. However, the key insight again is this double counting constraint and since X lies in bucket{A+1 - B}, this precludes it from being in any of its descendants, so the rest of the right subtree is ruled out, +1 inclusion count of X in bucket{A+1 - B} and then, boom, recuse this downward procedure but on the left subtree again where the size of the left subtree is matches the size of an entire subtree sans its bucket{A+1 - B} root that was ruled out So level by level down we go at least ruling out half the candidate buckets X could lie in each time and at most placing X in a single bucket per step and by virtue of halving, the cap of steps is Log(N).

Just like past Range section where I gave a preview of the Range computation for the actual BIT scheme where I used the extreme, highest, number 15, I'll do the same for the Update operation just like how the BIT scheme does it. But this time, the number that appears in the maximum amount of buckets is the extreme, smallest, number that is 1. Which is intuitive because smaller numbers appear in Range sums of all larger numbers so the smallest number will be used the most for all other numbers. The number 1 appears in bucket{1}, bucket{1-2}, bucket{1-4}, bucket{1-8}. So again, there's this power of 2 deal going on where 1 appears in 4 buckets staying within LogN confines. And when I do Range(X) pursuant to the BIT scheme, I must either get the 1 from bucket{1-8} and potentially continue down that bucket path for all X >= 8 or get the 1 from the remaining buckets. This means X < 8 so it's basically binary search, next decision node is if X >= 4 (but < 8) use bucket{1-4}, else X < 4 and so forth.

A Binary Tree Scheme

This is the section that essentially should spell everything out exactly, at least for the 4 bit, N=2^4=16 case and hopefully for all powers of 2 N.

Guiding picture paste here TODO

Actual BIT vs this guiding picture 2 main differences. BIT is technically a compressed version of my one. Basically remove all +0 nodes to be left with non-empty buckets. So technically by removing all 0s BIT is technically a forest and not a tree as the root 0 must also be removed in the spirit of fairness right? Second difference is that this picture is mores a recipe to compute the sequence of +Ks so to compute the indices a particular bucket contains, you have to do all the prefix addition from root till that bucket. The following picture makes this process of addition to enumerate index contents of every bucket explicit: Draw out below this picture a same tree except like instead of +0 or +K it'll have {} for empty buckets and like the index ranges like {1-8} {9-12} so forth. TODO draw and insert this picture

Functionally this complete binary tree (link/reference to main guiding picture) to me is how I reason about BIT scheme and makes reasoning extremely easy. Recursion and symmetry are readily

evident. Understand this and you can get a BIT scheme for free again simply a compressed version of this which I'll briefly discuss in final subsection.

Range operation

This tree essentially shows how binary representation works. Note how the leaves are labeled from 0-15. Track the root->leaf path for any leaf and you get the binary representation of that leaf of value X: this is you get a sum of up to Log(N) powers of 2 that together add up to, or cover, all of X. For example take previous example X=11 = 0b1011 = RLRR. 1st R is +8 so is bucket covering {1-8}, then L that covers nothing, so a dummy bucket, then 2nd R is +4 but I already am up to prefix sum or range of 8 so this '+' in +4 means offset so this 2nd R is second bucket{9-10} with size 2 and finally 3rd R is {11} and done. 11 = 8+0+2+1 or range {1-11} = {1-8}||{9-10}||{11} so natural decomposition of Range(X) to all the 1s in the binary representation of X. Then all buckets in this binary scheme correspond to R's, the +K nodes. And the L's, the +0 nodes, are dummy nodes.

Before going up update I would like to reinforce some of the preliminary thoughts intuition prior section. This tree is like a decision tree with 4 levels each level from MSB to LSB. First level decision is +0 or +8. Second level decision is +0 or +4. Third is +0 or +2. Fourth is +0 or +1. Regardless of the particular subtree you are in, whatever prefix of Rs and Ls you took to get to the root of a particular subtree, all subtrees at that given depth have symmetric mechanisms, symmetric scale of decision to make. Take Range(11). The top decision is +0 or +8 and 11 > 8 so I add bucket{1-8} to the sum and recurse right. Range(1,8) is done and what's left is Range(9-11). 11-8 is 3 so this is mechanistically equalvent to Range(11-8) = Range(3). Basically 11's path again is RLRR and 3's path is LLRR. The suffix LRR is the same so every R taken rules out its sibling Ls entire subtree thereby at least cutting the remaining buckets in half. So for 11, the first R taken is +8. This eliminates half the tree as now I am in the +8XXX subtree or the RXXX or the 1XXX subtree with 8 leaves spanning 8-15 vs the left subtree of +0XXX or LXXX or 0XXX subtree with 8 leaves spanning 0-7 which was just eliminated. Note how the highest value in the left subtree eliminated is exactly 1 less than 8, the size of the +8 bucket. This is simply how binary works and agrees with my preliminary intuition of having a bucket of size K + 1 and with left subtree size K and descendants size K as well. When I say how binary works 7 = 0b0111 and 8 = 0b1000. Adding 1 to 7 causes domino effect leading to 8. (Add link to my sums of powers of 2). In general when at prefix node and have to make a decision to go left or right, again going left is taking a 0 at that level/height in the binary representation and going right is taking a 1. So if you go left then the highest value you can create is by then going right so in binary we have the prefix P + the offset of 0b0{11…1}, that is an offset of in binary 1 zero followed by K ones. Versus right bucket which itself encodes the binary digit 0b1{00..0} or 1 followed by K zeroes. And 1 and K zeroes is exactly 1 more than 0 and K ones. Thus the size of the bucket allows it to 'dominate' and rule out the entire sibling left subtree. So again binary representation of number X has some 1 bits corresponding to R buckets selected and everytime I take the R path this eliminates half of the remaining descendants, half the buckets, to use to form X. So Log(N) steps given each step at worst if take R cuts half the available buckets to use to create X. So back to 11 vs 3, 11 eliminates the first left subtree ranging from 0-7 leaves, halves the search space and then recurses on a smaller, at least half as smaller, problem of covering {9-11} given that {1-8} was just covered. And by symmetry with the left subtree, this problem of covering {9-11} is exactly the same is if I wanted to compute Range(3) to begin with. This decision tree process with Range(3) goes like 3 < +8 so L, then 3 < +4 so L again, then 3 > 2 so R and 3 = 2+1 so R and thus Range 3 uses the buckets {1-2} and {3} which mechanistically are equivalent to the buckets {9-10} and {11} used to cover Range(11) once half the search problem was cut by taking the right R bucket covering {1-8}.

Update operation

Update well basically the zeroes in binary representation to the left of the LSB 1. In the tree finding X in this tree or rather +X. At each node you can go left +0 or go right till you reach X. Going left again is a 0 at the bit corresponding to that level of the binary rep of X. So if you had gone right at that node instead than that right bucket contains X (and thus none of its descendants do). Why must this right bucket contain X? I already stated it in previous section using binary representations but I love reiterating. For starters, you could look at the image and convince yourself of this. For example, +8 -> +4 bucket stores [9-12] and the leaves of the subtree of +8 -> +0 range from 8-11 and 8 is covered by the prefix +8 and 9-11 is covered in the {9-12} bucket as 12 > 11. Note that X > 8 or else the +8 bucket that is [1-8] would have already reached X and stop no need to explore further. OK so another way is by considering binary as I've already talked about in previous subsection. Taking the left branch means you have 0 at a given index the level corresponds to versus going right meaning you have 1 at that same index. Now 0 followed by any K binary digits, at most K 1's will still be less than the right bucket which is 2^K or 1 followed by K 0's (see note on sum of powers of 2 provide link). So whenever I see 0s on the path to the LSB of X from the root, that is at a branching I go left instead of right, I fill the right 1 child with X. Another way to think about is is classic DFS, where explore node entails 1: explore node.left 2: explore node.right 3: return to caller so after DFS-exploring returns from exploring X (so trailing zeroes have already been popped off the call stack) and as the recursion unwinds, right calls pop off the stack but left calls being reached means explore(node.left) finished and it's time to explore node.right and node.right covers 1 more than the max of the left subtree (again, see note on sum of powers of 2 provide link) so node.right bucket must have X (again the fact that node.left recursed means X wasn't reached so X is greater than the parent node's highest value so X >= node.right's lowest index but X < node.right's highest index again as node.right covers 1 more than max of the left subtree that contains X, so X fits inside node.right bucket, and again the upshot is this precludes all descendants from having X. So all the zeros before the LSB 1 each when replaced from 0 -> 1 and then all zeroes point to a bucket that X must be placed in, again log(N) such buckets as Log(N) digits. Each level precludes a particular power of 2 - 1 number of buckets from containing X. Like in the extreme case of 1 being in the most buckets, placing 1 in bucket {1-8} cuts off buckets 9-15 from having 1 so 7 these 7 = 2^3-1 buckets cannot have 1, then placing 1 in bucket {1-4} cuts off buckets 5-7 from having 1 so these 3 = 2^2-1 buckets from having 1, and finally placing 1 in bucket {1-2} cuts off bucket 3 from having 1 so this 1 = 2^1 - 1 bucket is blocked. These 3 blocks correspond to the 3 leading zeroes in the binary representation of 1: 1=0b0001.

But that's all BEFORE LSB, before reaching X. At some point during this binary representation tree traversal process, I'll turn right and find a bucket with +K such that X gets reached. This final right turn is the LSB 1 in the binary representation. So this bucket contains X by definition. And then zero or more +0 left turns meaning possible trailing zeroes. These zeroes do not need to be analogously updated to the zeroes in front on the LSB 1 because well, this LSB 1 bucket that contains X contains X for all its descendants as well.

Time to go back to binary approach. X can be represented in binary as 1s and 0s. Now to get ALL numbers strictly > X, here is how: for every 0 in the binary representation of X, set it to 1 and then the remaining bits to the right set them to all combinations which is a nice power of 2. Now for zeroes before the LSB 1, setting such a 0 to 1 and then zeroing out all bits yields the base bucket at that particular level of the tree that contains X and all other combinations are descendants of this base bucket, the upshot being for this level that the place/position of 0 that became a 1 corresponded to, only a single sibling bucket needed to take on X and all it's descendants then get X for free by the prefix-extending nature of this approach.

Final task is to show with a binary representation of X that when finding Range(K) where K > X show exactly 1 bucket contains X which was something I mentioned but did not justify in my Preliminary Thoughts Update subsection. I mean could just say binary representation of K and done as the binary representation of K has specifies disjoint buckets by virtue of prefix extension and they cover all K. So somewhere in this bucket chain exactly 1 bucket must contain X. Fair enough. But to elaborate, there may be a possibly empty prefix from the left/MSB where K matches X in binary but since K > X at some index, again possibly the very first one, K will have a 1 and X will have a 0. And again 2 familiar classes. First this 0 in X could be before the LSB 1 which means right there the sibling bucket with that 1 as it's LSB will contain X and as range(K) traverses down the tree all future buckets in the chain will be descendants of this bucket with X and thus will not have X and so exactly 1 and all good. Second class is when the 0 is after the LSB 1 in X and again that case but bucket associated with the LSB of X itself, that bucket with X as its highest index, would have X and then same concept everything else in K is a descendant of this bucket so again, see X exactly once.

Comparison With Canon BIT

Actual binary index tree connection. Already have described a perfectly serviceable scheme that supports Range and Update! N=4 bits there are 31 nodes 16 zeroes and 15 ones. How convenient given list size is 15, spanning indices 1-15. Essentially the BIT is my binary tree approach except with half + 1 dummy zero nodes being discarded as who cares about empty buckets? My approach still uses O(N) space like O(2N+1) = O(N). See 2nd Figure where the {} buckets can be ignored. Given that there are N=15 non-empty buckets and given each bucket itself has a unique path leading to it from the root that, describes a unique binary number between 1 and N, and let me name this the binary representation of the bucket. (Why is this binary represendation number unique and between 1 and N? You take unique path to reach a given bucket from the root which corresponds to unique prefix and then only go left all the way to leaf meaning add trailing zeroes thereby forming the binary representation for that bucket). So there are N non empty buckets each with a unique binary representation between 1 and N so it is natural to order the buckets by assigning every bucket an index that is the bucket's binary representation, the index again being specified by the path down the binary tree till taking the final R to hit a given non-empty R bucket. This index is the highest index the bucket contains, which is another, equivalent, view of the index of a bucket. Ex. the +8 -> +4 bucket has index 12 where 12=8+4 and the path to get that bucket is RR, two rights, which means 2^3 for first R and 2^2 for second R, and 12 = 2^3 + 2^2. So now that I have N buckets with N indices compactly representing them So it behooves a practical implementor to use a 1 indexed array of size N to compactly order and store the buckets, which is exactly what the BIT schemes mechanism uses: a size N auxiliary array to implicitly encode the BIT. I won't cover actual BIT implementation, won't discuss the fancy LSB bit trick but suffice to say, this array makes the tree operations quite cute as the operands, the X in Range(X) and Update(X, V), have a binary representation than can cleverly be extracted and processed to point to various cells in the auxilliary array. I don't care too much for the details and flashy bit tricks but more so the understanding, so I personally prefer using my guiding (link/ref) tree picture when reasoning about BIT concepts.

Final sidenote: again main difference from my approach to BIT is removing empty +0 bucket nodes so to be consistent and to not treat the root +0 specially, in the spirit of equality and fairness, for justice for all +0s, the root monarch +0 must be treated no different, equalized all the same, then technically its not so much a Binary Indexed Tree as it as a Binary Indexed Forest, like a forest of disjoint subtrees of increasing powers of 2 sizes with roots at: {1}, {1-2}, {1-4}, {1-8} and so forth. Ok I've written enough, thanks for making it so far I know I probably didn't. I'm so tired and instead of waxing eloquent, I'm waxing delirious.

Final final note/TODO: for fun copy paste picture from Wikipedia Fenwick (give credit to the independent author, maybe even ask permission? worst case I'll recreate it or maybe actually even copy the picture from Dr. Fenciks article on said Fenwick/BIT tree) three but color different subtrees and with same coloring deface my guiding picture as well (see text with green circles to nemesis but can also next like red circles for smaller nested subtrees as well). Apologies for defacing two wonderful pictures but for education and instruction, I must do so.