Binary Indexed Tree

Or

Fenwick Tree

Tutorial

Tutorial By

Vikas Awadhiya

Invented by,

Boris Raybko

Invented and popularized by,

Peter Fenwick

Tutorial By
Vikas Awadhiya
LinkedIn profile: https://in.linkedin.com/in/awadhiya-vikas

The Problem requires a Fenwick tree-like structure

Let's consider an array of sixteen elements as follows,

Array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]

Two types of operations are required to be performed on this array are as follows,

1.   Range Sum
2.   Element Update

There are few approaches to solve this problem.

Naïve approach

To update an element of array requires O(1) constant time complexity but finding the sub-array sum/range sum or prefix sum requires O(n) linear time complexity. This is not a practical approach with large size array and the number of range sum or prefix sum operations is much higher than the number of element update operations.

Prefix sum array

In this approach, an array of prefix sum is pre-calculated, where element at index contains the sum of array elements inclusively from 0 to that index as show below,

Array	=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Prefix Sum Array	=	1	3	6	10	15	21	28	36	45	55	66	78	91	105	120	136

Here finding the range sum or prefix sum is of O(1) constant time complexity but updating an element requires to re-evaluate the prefix sum array and have O(n) linear time complexity. This is also not a practical approach with large size array and the number of element update operations is much higher than the number of operations of range sum or prefix sum.

Binary Indexed Tree / Fenwick Tree

Binary indexed tree (BIT) or Fenwick tree approach performs range sum / prefix sum and element update in O(log n) logarithmic time complexity and has O(n) linear space complexity. Fenwick tree introduces binary indexed based tree data structure, which enable fast range sum and element update operations. Fenwick tree is an implicit tree, it means, Fenwick tree can be constructed using array and explicit nodes are not required. It maintains parent-child relationship among the elements through binary indexing. There is another data structure called Segment Tree, it also offers range query and update operation in logarithmic time complexity, but it requires more space than Fenwick tree.

Fenwick Tree Representation

How the Fenwick tree is constructed or represented in an auxiliary array is essential to understand in order to understand how it perform range sum query and element update, like finding the sum of range [3, 7] of array in O(log n) time complexity.

Fenwick tree can also be used with different type of queries like minimum or maximum of a range, but this tutorial uses range sum query to explain Fenwick tree and because of it the elements of Fenwick tree represent the range sums of an array.

Note: Binary indexing logic requires a 1-based index and doesn't work with a 0-based index. For explaining simplicity, we assume that the 0th index of array doesn't represent any value, as highlighted by blue color filled cell below in fig 1.0,

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

fig 1.0

These continuous values from 1 to 16 are taken as an array because these values also represent their respective indexes. If the array contains N elements, then an auxiliary array of size N + 1 is required to representing Fenwick tree because of the 1-based index requirement.

Each element of Fenwick tree represents sum of a particular range of array and that range is found as follows,

Let's consider index of Fenwick tree is 6 and its binary representation is 0110_b,

Index = 6 = 0110_b

And the least significant set bit is highlighted by bold and underline as shown below,

6 = $01\underline{\mathbf{1}}0_b$

This least significant set bit needs to be extracted. In other words, except for least significant set bit, all the set bits are unset.

The extraction of the least significant set bit of an index can be represented by lssb(index). The lssb is natural abbreviate of **_least significant set bit_**. Here LSSB is not used because **_lssb of index_** is written as a function having index as a parameter and function names don't start with a capital letter.

lssb(6) = lssb(0110_b) = 0010_b

lssb() turn the number 6 into number 2, then it is subtracted with index and one is added to get lower bound of the inclusive range as follows,

index – lssb(index) = 6 – lssb(6) = 0110_b – 0010_b = 6 – 2 = 4

Lower bound of range = (index – lssb(index)) + 1 = 4 + 1 = 5

So FenwickTree[6] contains the sum of inclusive range [5, 6] of Array.

FenwickTree[index] contains the sum of a range of array which has inclusive upper bound as index itself and inclusive lower bound can we found as explained above. As a formula, we can write as follows,

FenwickTree[index] = Sum of range [(index - lssb(index)) + 1, index], of array.

In programming, least significant set bit can be obtained by bitwise AND of index with its negative value as follows,

lssb(index) = index & (-index)

Now let's construct Binary Indexed Tree or Fenwick Tree and starts at index 1,

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

Fenwick Tree =		1	3	3	10												

fig 2.0
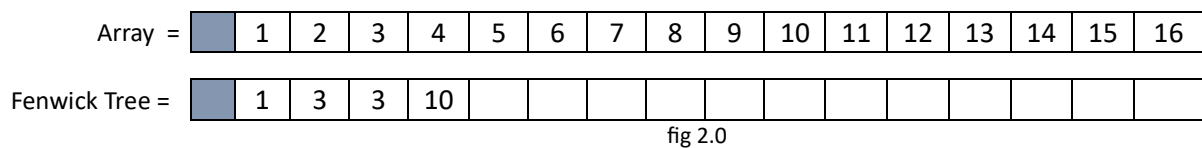
- index = 1 = 0001_b
  lssb(1) = lssb(0001_b) = 0001_b = 1, result of lssb(1) remains one because 1 has only one set bit. So the FenwickTree[1] will contains the sum of following range of Array,
  FenwickTree[1] = [(1 - lssb(1)) + 1, 1] = [(1 − 1) + 1, 1] = [1, 1],

  The FenwickTree[1] contains sum of range of single element of Array which starts at Array[1] and also end on Array[1]. As shown above in fig 2.0, FenwickTree[1] has value 1.

- index = 2 = 0010_b
  lssb(2) = lssb(0010_b) = 0010_b = 2, again number 2 contains only one set bit that's why the lssb(2) is equal to 2. So the FenwickTree[2] will contains the sum of following range of Array,
  FenwickTree[2] = [(2 − lssb(2)) + 1, 2] = [(2 − 2) + 1, 2] = [1, 2],

  The FenwickTree[2] contains sum of inclusive range of two elements of Array and range starts at Array[1] and inclusively end at Array[2] and the sum is equal to 3 as shown above in fig 2.0.

- Index = 3 = 0011_b
  lssb(3) = lssb(0011_b) = 0001_b = 1
  FenwickTree[3] = [(3 − lssb(3)) + 1, 3] = [(3 − 1) + 1, 3] = [3, 3]
  The FenwickTree[3] contains value of Array[3] which is 3 as shown above in fig 2.0.

- Index = 4 = 0100_b

  lssb(4) = lssb(0100_b) = 0100_b = 4

  FenwickTree[4] = [(4 − lssb(4)) + 1, 4] = [(4 − 4) + 1, 4] = [1, 4]

  The FenwickTree[4] contains sum of Array[1], Array[2], Array[3] and Array[4], which is 10 as shown above in fig 2.0.

Similarly, the values of remaining indexes can be calculated, and the fully constructed Fenwick tree is shown as below in fig 3.0,

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

Fenwick Tree =		1	3	3	10	5	11	7	36	9	19	11	42	13	27	15	136

fig 3.0

A similarity can be observed here between Fenwick Tree and Prefix Sum Array and that is, both the data-structures at a particular index contains the sum of range of array having index as inclusive upper bound but differ by inclusive lower bound. In Prefix Sum Array, the inclusive lower bound is always 1 but in case of Fenwick Tree inclusive lower bound varies and depends on index.

Now obvious question arises, what is the logic behind the way the inclusive lower bound of a range is calculate or in other words, how Fenwick tree divides array into different ranges and how collectively all these ranges can answer any possible range sum query and how update in array affect these ranges?

The logic behind how the Fenwick tree divides array into different ranges is explained later in this document but it is first required to understand how the sum query and update operation work to understand the range logic.

Range Sum Query

Let's see how the Fenwick tree find sum of a range of array.

Now consider a sum query to find the sum of range [2, 5] of Array,

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

Fenwick Tree =		1	3	3	10	5	11	7	36	9	19	11	42	13	27	15	136

fig 4.0

Range [2, 5] as highlighted above in the fig 4.0 by gray color filled cells. The sum of the range [2, 5] is 14, now let's see how Fenwick tree will find it,

Sum query is evaluated in following manner,

Sum = [lower bound, upper bound]

[1, upper bound] − [1, (lower bound − 1)]

To find the sum of the range [2, 5], first the sum of the range [1, 5] is calculated and then sum of the range [1, 1] is calculate and then subtracting these sums gives the final answer.

Sum of range [2, 5] = sum of range [1, 5] − sum of range [1, 1]

Sum of range [1, 5] is calculated as follows,

As we know, the FenwickTree[index] at each index contains the sum of inclusive range [(index − lssb(index)) + 1, index] of array where inclusive upper bound of range is index.

So the FenwickTree[5] must contains the sum of a range which may or may not be equal to the range [1, 5] but it definitely includes the value of Array[5] and that's why sum calculation always begin with upper bound as an index.

Sum of range [1, 5] = FenwickTree[5] = 5   (initially)

Now let's see what is the range whose sum is represented by the FenwickTree[5]

lssb(5) =   lssb(0101_b) = 0001_b = 1

FenwickTree[5] =   [(5 − lssb(5)) + 1, 5]
[(5 − 1) + 1, 5]
[5, 5]

The Fenwick tree at index 5 contains the sum of the range [5, 5] which is not equal to [1, 5], so it must continue until the entire range [1, 5] is covered.

Fenwick tree is an implicit tree data structure which doesn't have nodes, but its elements still have parent-child relationship among them. Suddenly the parent-child relationship came into the picture because the current element FenwickTree[5] contains the sum of the range [5, 5] which is not equal to the range [1, 5] and only partially covers it.

When an element in Fenwick tree doesn't completely provide the sum of the required range, calculation jumps to its parent element. In this case the parent element would be the element which either provides the complete sum of remaining part of the range or some other continuous fractional part of it.

Parent element/index of an element/index can be found in similar way the inclusive lower bound of a range at index is calculated with one difference.

As discussed at FenwickTree[index] = sum of range [(index – lssb(index)) + 1, index], and

    Inclusive lower bound = (index – lssb(index)) + 1, and

Parent Index =	index – lssb(index)
lssb(5) =	lssb(0101_b) = 0001_b = 1
Parent index =	Index – lssb(index) = 5 – lssb(5) = 0101_b – lssb(0101_b)
	0101_b – 0001_b = 0100_b
	4

The only difference between parent index and lower bound is that, in the lower bound one is added. It is obvious that parent represents sum of a range having upper bound one less than the lower bound of a range represents by child. It means the sums at parent and child indexes can be combined to find the range span across multiple levels in the Fenwick tree.

All the above theory is presented to explain why the index is subtracted by its lssb(index) value until it remains greater than zero (because 0th position is a dummy parent in the Fenwick tree and doesn't contribute to sum). So finally let's see sum query evaluation,

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

Fenwick Tree =		1	3	3	10	5	11	7	36	9	19	11	42	13	27	15	136

fig 5.0

Sum =	[2, 5]
	[1, 5] – [1, (2 – 1)]
	[1, 5] – [1, 1]
Sum of [1, 5]	
Index =	5 it represent sum of range [(5 – lssb(5)) + 1, 5] = [5, 5]
Sum$_{[1, 5]}$ =	FenwickTree[5] = 5 (see the value of FenwickTree[5] in fig 5.0 above)
Index =	Index - lssb(index) = 5 – lssb(5) = 0101_b – lssb(0101_b) = 0101_b – 0001_b = 0100_b
	4
	index 4 represents sum of range [(4 – lssb(4)) + 1, 4] = [1, 4]

Sum$_{[1, 5]}$ **=**	Sum$_{[1, 5]}$ + FenwickTree[4]	(see the value of FenwickTree[4] in fig 5.0 above)
	5 + 10	
	15	
Index **=**	Index − lssb(index) = 4 − lssb(4) = 0100$_b$ − lssb(0100$_b$) = 0100$_b$ − 0100$_b$ = 0000$_b$	
	0	
	No further evaluation required because index no longer greater than zero.	
Sum of [1, 1]		
Index **=**	1 It represent sum of range [(1 − lssb(1)) + 1, 1] = [1, 1]	
Sum$_{[1, 1]}$ **=**	FenwickTree[1] (see the value of FenwickTree[1] in fig 5.0 above)	
	1	
Index **=**	Index − lssb(index) = 1 − lssb(1) = 0001$_b$ − lssb(0001$_b$) = 0001$_b$ − 0001$_b$ = 0000$_b$	
	0	
	No further evaluation required because index no longer greater than zero.	

Finally, the sum of query range [2, 5] is,

Sum$_{[2, 5]}$ = Sum$_{[1, 5]}$ − Sum$_{[1, 1]}$ = 15 − 1 = 14

So, the Fenwick tree correctly calculated the sum of query range [2, 5] and this is how it works.

There is one edge case where sum query has range [1, 1] then,

Sum = [1, upper bound] − [1, (lower bound − 1)] = [1, 1] − [1, (1 − 1)]

        [1, 1] − [1, 0]

The range [1, 0] is an invalid range but Fenwick tree can still handle it because sum calculation of a range begins at upper bound as an index and in range [1, 0] upper bound is zero and because of it, sum evaluation process ends before it can begin and as a result the sum of range [1, 0] remains zero but this may not work with other type of range queries for example min or max range query and must be handled explicitly. One possible way to handle it is, if the range is one element long like [index, index] then directly return the value of Array[index] as a sum. The query range must be valid and cannot have upper bound index less then lower bound index.

Element Update

An element update of Array requires the update of sum of all the ranges those includes the value of this element in their sum and causes multiple elements update in Fenwick tree.

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

Fenwick Tree =		1	3	3	10	5	11	7	36	9	19	11	42	13	27	15	136

fig 6.0

Now let's consider updating value of fifth element of array to 9. Fifth element is highlighted by gray color filled cell above in fig 6.0.

Update of Array[5] requires the update of sum of all the ranges those includes value of fifth element of Array, represented in Fenwick tree.

It is known from the time of Fenwick tree construction that FenwickTree[index] represents the sum of a range having index as inclusive upper bound, for example FenwickTree[6] represents sum of a range [?, 6] which has some inclusive lower bound but 6 as inclusive upper bound. It means all the values of FenwickTree[index] where index is less than 5 do not represent sum of any range which includes Array[5], because inclusive upper bounds of those ranges are less than 5.

So, the update operation begins with element update index, in this case it is 5th index.

The value of Array[5] is updated to 9 which is greater than previous value 5 and the difference introduced is 4. So sum of all the ranges those includes the value of Array[5] must be increased by 4.

Update begins at index 5 by adding 4 to FenwickTree[5],

FenwickTree[5] = FenwickTree[5] + 4 = 5 + 4 = 9

After updating FenwickTree[5], index moves to next index in FenwickTree which represents sum of a range that also includes the value of Array[5]. Following is the way to find next index,

In sum query, value of lssb(index) was subtracted to index to find the parent index but in update operation value of lssb(index) will be added to index to find the next index. Here index is updated by adding value of lssb(5),

$$lssb(5) = lssb(0101_b)$$
$$0001_b$$
$$1$$
$$index = Index + lssb(5)$$
$$5 + 1$$
$$6$$

Adding the value of lssb(index) continues until the index remains smaller or equal to max index. Here max index equals to the number of elements in array.

So, let's see how update operation works,

Array =

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

Fenwick Tree =

	1	3	3	10	5	11	7	36	9	19	11	42	13	27	15	136

fig 7.0

Array[5] =	5
Update Array[5] =	9
diff =	Array[5] – previous value of Array[5] = 9 – 5 = 4
max index =	16 (equals to the number of elements in array)
index =	**5** (begins with the updated element's index of Array)
FenwickTree[index] =	FenwickTree[5] + diff
	5 + 4 = 9 (see the value of FenwickTree[5] above in fig 7.0)
lssb(index) =	lssb(5) = lssb(0101_b) = 0001_b = 1
Index =	Index + lssb(index) = 5 + 1 = **6** (6 < 16, so the process continues)
	Fenwick tree at index 6 represent sum of a range [5, 6] which includes the value of Array[5] in sum that's why update is required.
	How to find the range whose sum is represented at index is explained in Fenwick tree representation section.
FenwickTree[index] =	FenwickTree[6] + diff
	11 + 4 = 15 (see the value of FenwickTree[6] above in fig 7.0 above)
lssb(index) =	lssb(6) = lssb(0110_b) = 0010_b = 2
Index =	Index + lssb(index) = 6 + lssb(6) = 6 + 2 = **8** (8 < 16, so the process continues)
	Fenwick tree at index 8 represent sum of range [1, 8], which includes the value of Array[5] in sum that's why update is required.
FenwickTree[index] =	FenwickTree[8] + diff
	36 + 4 = 40 (see the value of FenwickTree[8] above in fig 7.0 above)
lssb(index) =	lssb(8) = lssb(1000_b) = 1000_b = 8
Index =	Index + lssb(index) = 8 + lssb(8) = 8 + 8 = **16** (16 == max index, so the process continues)

	Fenwick tree at index 16 represent sum of range [1, 16], which includes the value of Array[5] in sum that's why update is required.
FenwickTree[index] =	FenwickTree[16] + diff
	136 + 4 = 140 (see the value of FenwickTree[16] above in fig 7.0 above)
lssb(index) =	lssb(16) = lssb(10000_b) = 10000_b = 16
Index =	Index + lssb(index) = 16 + lssb(16) = 16 + 16 = **32**
	Now index is greater than max index 16, so the process ends here.

The updated elements are highlighted by gray color filled cells as follows,

Array =		1	2	3	4	9	6	7	8	9	10	11	12	13	14	15	16

Fenwick Tree =		1	3	3	10	9	15	7	40	9	19	11	42	13	27	15	140

fig 7.0

This is how Fenwick tree handles an element update of array by updating the sum of ranges accordingly.

Sum query and update operation are explained in detailed and now it's clear how these operations work. It is also clear how the Fenwick tree is constructed or represented in an auxiliary array. All the presented information up to now is enough to work with Fenwick tree but it is not enough to fully understand the Fenwick tree and to construct it in linear time complexity.

To understand why the sum query and update options are performed in manners as explained above and what is the logic behind it? It is first required to understand how Fenwick tree divides the array in ranges or in other words, why Fenwick tree at an index represent sum of an inclusive range [(index – lssb(index)) + 1, index]?

Next section of the document explains range logic in detail.

For explaining purpose, it was assumed that the 0th index of Array doesn't represent any value, but it doesn't happen in implementation and a 0-based index of array is required to be translated to the 1-based index for Fenwick tree.

Ranges

This section explains how the Fenwick tree divides the array in ranges and how the following range formula emerged?

Range represented at index = [(index – lssb(index)) + 1, index]

If we list the range of first four indexes and then other four indexes as follows,

[1, 1]	[1, 2]	[3, 3]	[1, 4]
[5, 5]	[5, 6]	[7, 7]	[1, 8]

Here inclusive upper bound also represents index.

As you can see, the similarity between these two sets of quadruplets, both first and third ranges are one element long and both second ranges are two elements long ([1, 2] and [5, 6]) and the last ranges include all the elements before it, as range at index 4 includes all elements from 1 to 4 and range at index 8 includes all the elements from 1 to 8.

If we continue and list two more quadruplets, similarity persists,

[9, 9]	[9, 10]	[11, 11]	[9, 12]
[13, 13]	[13, 14]	[15, 15]	[1, 16]

Here one difference is in fourth range of first quadruplet (range of index 12) that only include the elements from 9th index which is not like the other fourth range of quadruplet which include element from index 1 (from beginning). This range includes the index from the beginning of its quadruplet which is index 9.

We can list few more quadruplets to see the similarity and differences as follows,

[17, 17]	[17, 18]	[19, 19]	[17, 20]
[21, 21]	[21, 22]	[23, 23]	[17, 24]

By looking these ranges, one thing is obvious that there is some pattern in these ranges but still it is not clear what the pattern is? Now the Fenwick tree is drawn to understand the logic behind these ranges.

As discussed Fenwick Tree requires 1-based index that's why FenwickTree[0] is dummy element and doesn't represent any information. The 0th index is treated as root (dummy parent) in Fenwick tree. To draw the tree, it is required to know the parent child relationship between the elements and as discussed in the sum query section the parent of the index can be found as follows,

Parent = index – lssb(index)

index = 1	parent	=	1 – lssb(1)	=	0001_b – lssb(0001_b)	=	1 – 1	=	**0**
index = 2	parent	=	2 – lssb(2)	=	0010_b – lssb(0010_b)	=	2 – 2	=	**0**
index = 3	parent	=	3 – lssb(3)	=	0011_b – lssb(0011_b)	=	3 – 1	=	2
index = 4	parent	=	4 – lssb(4)	=	0100_b – lssb(0100_b)	=	4 – 4	=	**0**
index = 5	parent	=	5 – lssb(5)	=	0101_b – lssb(0101_b)	=	5 – 1	=	4
index = 6	parent	=	6 – lssb(6)	=	0110_b – lssb(0110_b)	=	6 – 2	=	4
index = 7	parent	=	7 – lssb(7)	=	0111_b – lssb(0111_b)	=	7 – 1	=	6
index = 8	parent	=	8 – lssb(8)	=	1000_b – lssb(1000_b)	=	8 – 8	=	**0**

As you can observer, zero is parent of all the indexes those are the power of two (2^p) like 1, 2, 4 and 8. This is obvious because binary representation of numbers those are 2^p have only one set bit and result of lssb(2^p) gives the same number and only subtracting a number with itself gives zero.

This is very import observation, and it means array at top level is divided at the indexes of 2^p. There may be more division/ranges at next level, but these next level ranges must be residing between the first level divisions introduced by indexes of 2^p. Below fig 8.0 depicts the first level of Fenwick tree. Here numbers in the boxes represent indexes to show the parent-child relationship.
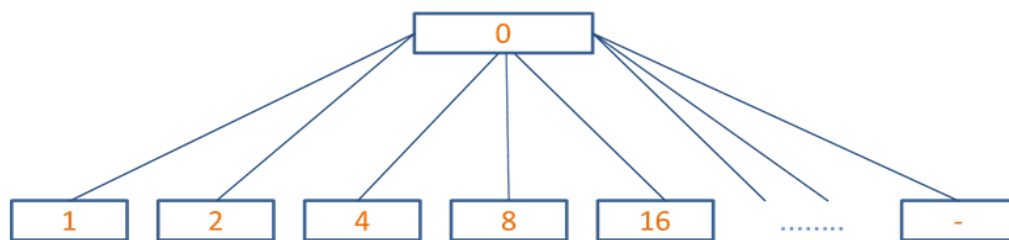

fig 8.0

To verify the above statement let's consider two indexes of 2^p, 4 and 8 and see do the next level ranges defines between them includes any element outside of indexes 4 and 8 or all next level ranges stay in the division introduced by indexes 4(2^2) and 8(2^3) at top level.

range at index =	[(index – lssb(index)) + 1, index]
range at index 5 =	[(5 – lssb(5)) + 1, 5] = [(5 – 1) + 1, 5] = [5, 5]
range at index 6 =	[(6 – lssb(6)) + 1, 6] = [(6 – 2) + 1, 6] = [5, 6]
range at index 7 =	[(7 – lssb(7)) + 1, 7] = [(7 – 1) + 1, 7] = [7, 7]

So as shown above, ranges defined at indexes 5, 6 and 7 don't includes any element outside of sentinel indexes 4 and 8. So let's draw these ranges and the parent of index 5 is index 4 (parent = 5 – lssb(5)), parent of index 6 is also index 4 and parent of index 7 is index 6. See the second level ranges in fig 9.0 below,
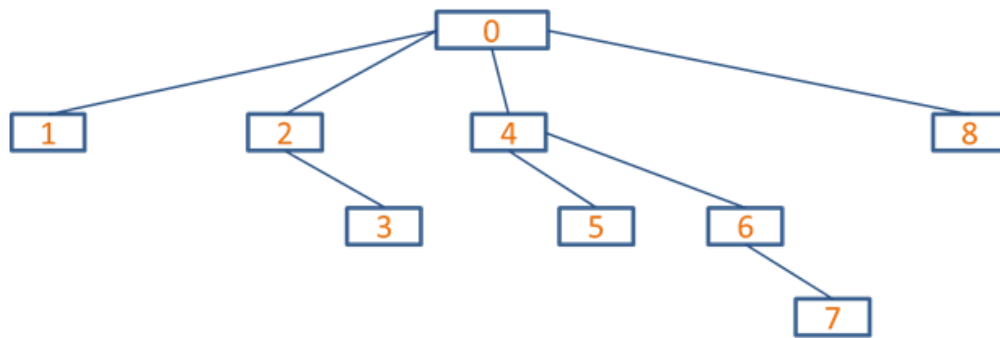
fig 9.0

Another important observation about indexes of 2^p is that, these indexes represent ranges always starts from first index that is $[1, 2^p]$, because 2^p have only one set bit and lssb(2^p) = 2^p, so always have lower bound = 1 (lower bound = (index − lssb(index)) + 1). For example, indexes, 2, 4, 8 and 16 represent ranges [1, 2], [1, 4], [1, 8] and [1, 16] respectively.

First level divisions/ranges are of higher importance and next or sub level division / sub ranges between these ranges are not typically important and don't introduce complexity overhead (explained in next section).

Due to the first level division introduced by indexes of 2^p, in sum query index move up to the parent level until sum of range is accumulated or reaches to the first level index. Similarly in the element update, index move to next index that represents the range includes the value of updated element in its sum, until the next first level index reaches and after that index only move to the first level indexes.

Odd indexes have their right most bit (2^0) set and lssb(odd index) = 1 and due to this odd indexes represent ranges of only one element long (range = [(index − lssb(index)) + 1, index]). Below in fig 10.0, odd indexes in Fenwick tree are highlighted by gray color filled cells and their values are equal to respective Array elements because they represent only 1 element long range.

Array =		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Fenwick Tree =		1	3	3	10	5	11	7	36	9	19	11	42	13	27	15	136

fig 10.0

And the even indexes represent range of length equals to the value of their lease significant set bit. Let's see few even indexes as follows,

Index = 10 = 1010$_b$	lssb(1010$_b$) = 2^1 = 2	Range = [9, 10], two elements long range.
Index = 12 = 1100$_b$	lssb(1100$_b$) = 2^2 = 4	Range = [9, 12], four elements long range.
Index = 40 = 101000$_b$	lssb(101000$_b$) = 2^3 = 8	Range = [33, 40], eight elements long range.
Index = 64 = 1000000$_b$	lssb(101000$_b$) = 2^6 = 64	Range =[1, 64], sixty-four elements long range.

Time Complexity

Let's revisit the sum query and update operation with the time complexity perspective.

In the sum query section range [1, 5] was one of the ranges whose sum was evaluated. So if we find its sum again then we start with upper bound as index = 5 which represent range [5, 5], so we add value of FenwickTree[5] to sum. 5 is not an index of 2^p, so we find the parent of it which is index 4, so we add value FenwickTree[4] to sum. 4 is an index of 2^p which is a sentinel of first level division includes the elements from first element. So, the sum is completely evaluated, and the process ends here.

It means if a sum query range [1, 2^p] has upper bound as 2^p then it is immediately calculated because upper bound is a sentinel index of first level division but if the upper bound of sum query is not equal to 2^p then process starts from upper bound as index and moves up to the previous index which is first level index of 2^p to evaluate the sum.

Let's consider a sum query range [1, 1913], upper bound is not 2^p and process begins with upper bound as index = 1913

Index = 1913 = | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |, add FenwickTree[1913] to sum and unset lssb

Index = 1912 = | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |, add FenwickTree[1912] to sum and unset lssb

Index = 1904 = | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |, add FenwickTree[1904] to sum and unset lssb

Index = 1888 = | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |, add FenwickTree[1888] to sum and unset lssb

Index = 1856 = | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | , add FenwickTree[1856] to sum and unset lssb

Index = 1792 = | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |, add FenwickTree[1792] to sum and unset lssb

Index = 1536 = | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |, add FenwickTree[1536] to sum and unset lssb

Index = 1024 = | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |, add FenwickTree[1024] to sum and unset lssb

Index = 0   = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |, process ends here.

Here un-setting the lssb (least significant set bit) highlighted by gray color filed cells is equivalent of moving up to the parent level (parent = index – lssb(index)). The upper bound index 1913 resides between the sentinel 2^{10} (1024) and 2^{11}(2048) and process moves up to the sentinel 1024 and technically it is the last index contributes to sum then index zero only indicates end of the process. As you can see sum from index 1 to index 1913 is calculated only in 8 steps and stops at index zero.

So, the sum query only requires number of steps equals to the number of set bits of upper bound of query range. Un-setting the lssb decrease the index by $2^{bit\ pos}$ and due to this it has logarithmic time complexity $\log_2(1913) \approx 10.9$ and requires only 9 steps including the 0th index step.

Now let's see the update operation again of Array[5] element with one difference from element update section, this time array size is 1913. The update process starts at index 5,

Index = 5   = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |, update FenwickTree[5] and add lssb to index

Index = 6   = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |, update FenwickTree[6] and add lssb to index

Index = 8   = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |, update FenwickTree[8] and add lssb to index

Index = 16 = | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |, update FenwickTree[16] and add lssb to index

Index = 32 = | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |, update FenwickTree[32] and add lssb to index

Index = 64 = | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |, update FenwickTree[64] and add lssb to index

Index = 128 = | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |, update FenwickTree[128] and add lssb to index

Index = 256 = | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |, update FenwickTree[256] and add lssb to index

Index = 512 = | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |, update FenwickTree[512] and add lssb to index

Index = 1024 = | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |, update FenwickTree[1024] and add lssb to index

Index=2048= | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |, process ends here.

Here adding lssb (least significant set bit) highlighted by gray color filed cells, to index moves the index to the next index represents a range also includes the value of Array[5] element. Updating FenwickTree[index] means adding difference introduce in Array[5] element to the FenwickTree[index] value.

Update starts at index = 5 but it is not an index of 2^p, so index moves toward the next sentinel index of first level division that is 8 (2^3). So after updating FenwickTree[5] index moves to the next index 6 and update FenwickTree[6] which represent range [5, 6] after this index reaches to next index 8 and that is first level sentinel (2^p). Once the index reaches at the index 8 after it, index only moves to the first level sentinel indexes like 16, 32, 64 and so on, because once the first level sentinel reaches no other range can include 5th element (as explained in ranges section) and only the sentinel indexes get updated because they represent ranges which include element from first element.

Process ends at index = 2048 but technically index = 1024 is the last index which gets updated because it is last valid first level sentinel/index and index 2084 is greater than array's size.

Number of steps require depends on the value of maximum index of 2^p which is less than or equals to the size of array. So, the maximum number of steps required is p + 1. For example, in this case, 1024 (2^{10}) is value of maximum index of 2^p which is less than 1913 because of it 10 + 1 = 11 steps were required by element update operation and has logarithmic time complexity $\log_2(1913) \approx 10.9$.

Construction in Linear Time Complexity

Fenwick tree can be constructed in O(n) linear time complexity. Let's consider the FenwickTree[12], it represent sum of array range [9, 12] and the elements Array[9], Array[10], Array[11] and Array[12] shall be added iteratively. Let's see how it acquires its sum step by step,

index = 9	Represents range [9, 9], so FenwickTree[9] has proper sum then add its value to FenwickTree[otherIndex], where otherIndex = 9 + lssb(9) = 10 because otherIndex represents a range that includes the value of Array[9].
index = 10	Represents range [9, 10] and FenwickTree[10] also has proper sum because of previous step (value of FenwickTree[9] was added) then add its value to FenwickTree[otherIndex], where otherIndex = 10 + lssb(10) = 12.
Index = 11	Represents range [11, 11], so FenwickTree[11] has proper sum then add its value to FenwickTree[otherIndex], where otherIndex = 11 + lssb(11) = 12.
Index = 12	Represents range [9, 12] and because of previous three steps the FenwickTree[12] accumulated the proper sum.

It means, at each index, it is only required to add the value of FenwickTree[index] to FenwickTree[otherIndex], where otherIndex = index + lssb(index), greater than index (see element update section) and represents a range which includes few of the same elements of Array as included by the range represent by index. The process begins at index = 1 and goes up to index = Array's size and construct the Fenwick tree in O(n) linear time complexity.

The implementation requires few modifications. In above example FenwickTree[9] was assumed to have proper sum but that is not the case in implementation and it is first required to add the value of Array[index − 1] to FenwickTree[index] before modifying the FenwickTree[otherIndex] in each step. It is required at each steps because auxiliary array which represents Fenwick tree can't be copy initialized by Array at the time of allocation due to 1-based index of Fenwick tree where FenwickTree[0] doesn't represent a valid value and auxiliary array is initialized with zero is provided as initialization value for all of its elements. For example in one of the step above at index = 9, value of Array[8] has to be added to the FenwickTree[9] before modifying FenwickTree[otherIndex] where otherIndex = 10.

Another modification is, index has to iterate from 2^p to 2^{p+1} (exclusive) and can't iterate from 1 to Array's size because let's say array's size is 12 and at index = 8, process tries to modify invalid otherIndex which is otherIndex = 8 + lssb(8) = 16. To avoid validation at each step, index must iterate from 2^p to 2^{p+1}(exclusive). In this case only one validation is required of 2^{p+1} and if 2^{p+1} is valid then for all the index from 2^p to 2^{p+1}(exclusive), otherIndex will never be greater than 2^{p+1}(as explained in range section). But if 2^{p+1} is greater than Array's size then index iterate from 2^p to Array's size with validation at each step in last iteration. Like in this case index iterates from 1 to 2 then from 2 to 4 after it from 4 to 8 but it can't iterate from 8 to 16 because size of array is 12. In last iteration index iterates from 8 to 12 and validation of otherIndex is required in each step.

Implementation

A C++ implementation of Fenwick tree is providing under MIT License. It also has a main.cpp file to demonstrate how to use it.

Code available under MIT License at:

https://github.com/vikasawadhiya/Binary-Indexed-Tree-Or-Fenwick-Tree

This implementation of Fenwick Tree can be used for range sum queries. If you want a more generic Fenwick Tree where Fenwick tree can be used for other types of range queries like minimum or maximum of a range then, you can implement it yourself.

This code demonstrates how Fenwick Tree can be implemented for the range sum queries in simplest manner and purposely not providing the generic implementation which can be used with other type of range queries because it would introduce code complexity and make the code harder to understand.

You can use arithmetic progression (AP) series initially to see the Fenwick tree in action, like this document used natural numbers from 1 to 16. The benefit of using AP series initially as examples is that, you can work with large size array and can easily verify the result given by Fenwick tree because you can find the sum of any range in AP series by its sum formula and in case of element update you can still verify because you know what should be the difference between the sum calculated by the sum formula and sum given by Fenwick tree of a range after element update.

December 2024, India