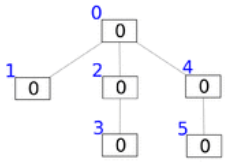
Fenwick tree

A **Fenwick tree** or **binary indexed tree** is a data structure that can efficiently update elements and calculate prefix sums in a table of numbers.

This structure was proposed by Boris Ryabko in 1989 [1] with a further modification published in 1992. [2] It has subsequently became known under the name Fenwick tree after Peter Fenwick who has described this structure in his 1994 paper.[3]

When compared with a flat array of numbers, the Fenwick tree achieves a much better balance between two operations: element update and prefix sum calculation. In a flat array of n numbers, you can either store the elements, or the prefix sums. In the first case, computing prefix sums requires linear time; in the second case, updating the array elements requires linear time (in both cases, the other operation can be performed in constant time). Fenwick trees allow both operations to be performed in $O(\log n)$ time. This is achieved by representing the numbers as a tree, where the value of each node is the sum of the numbers in that subtree. The tree structure allows operations to be performed using only $O(\log n)$ node accesses.



Create a binary indexed tree for the array [1, 2, 3, 4, 5] by inserting one by one.

Contents

Motivation

Given a table of elements, it is sometimes desirable to calculate the running total of values up to each index according to some associative binary operation (addition on integers being by far the most common). Fenwick trees provide a method to query the running total at any index, in addition to allowing changes to the underlying value table and having all further queries reflect those changes.

Fenwick trees are particularly designed to implement the arithmetic coding algorithm, which maintains counts of each symbol produced and needs to convert those to the cumulative probability of a symbol less than a given symbol. Development of operations it supports were primarily motivated by use in that case.

Using a Fenwick tree it requires only $O(\log n)$ operations to compute any desired cumulative sum, or more generally the sum of any range of values (not necessarily starting at zero).

Fenwick trees can be extended to update and query subarrays of multidimensional arrays. These operations can be performed with complexity $O(4^d \log^d n)$, where d is number of dimensions and n is the number of elements along each dimension.[4]

Description

Although Fenwick trees are trees in concept, in practice they are implemented as an implicit data structure using a flat array analogous to implementations of a binary heap. Given an index in the array representing a vertex, the index of a vertex's parent or child is calculated through bitwise operations on the binary representation of its index. Each element of the array contains the pre-calculated sum of a range of values, and by combining that sum with additional ranges encountered during an upward traversal to the root, the prefix sum is calculated. When a table value is modified, all range sums which contain the modified value are in turn modified during a similar traversal of the tree. The range sums are defined in such a way that both queries and modifications to the table are executed in asymptotically equivalent time ($O(\log n)$ in the worst case).

The initial process of building the Fenwick tree over a table of values runs in $O(n)$ time. Other efficient operations include locating the index of a value if all values are positive, or all indices with a given value if all values are non-negative. Also supported is the scaling of all values by a constant factor in $O(n)$ time.

A Fenwick tree is most easily understood by considering a one-based array. Each element whose index `i` is a power of 2 contains the sum of the first `i` elements. Elements whose indices are the sum of two (distinct) powers of 2 contain the sum of the elements since the preceding power of 2. In general, each element contains the sum of the values since its parent in the tree, and that parent is found by clearing the least-significant bit in the index.

To find the sum up to any given index, consider the binary expansion of the index, and add elements which correspond to each 1 bit in the binary form.

For example, say one wishes to find the sum of the first eleven values. Eleven is 1011_2 in binary. This contains three 1 bits, so three elements must be added: 1000_2, 1010_2, and 1011_2. These contain the sums of values 1–8, 9–10, and 11, respectively.

To modify the eleventh value, the elements which must be modified are 1011_2, 1100_2, 10000_2, and all higher powers of 2 up to the size of the array. These contain the sums of values 11, 9–12, and 1–16, respectively. The maximum number of elements which may need to be updated is limited by the number of bits in the size of the array.

Implementation

A simple C implementation follows.

```c
#define LSB(i) ((i) & -(i)) // zeroes all the bits except the least significant one

int A[SIZE];

int sum(int i) // Returns the sum from index 1 to i
{
    int sum = 0;
```

```
    while (i > 0)
        sum += A[i], i -= LSB(i);
    return sum;
}

void add(int i, int k) // Adds k to element with index i
{
    while (i < SIZE)
        A[i] += k, i += LSB(i);
}
```

Updating and Querying the Tree

The following table describes various ways in which Fenwick tree can be used. More importantly it states the right API to be called or used in order to achieve the desired result along with an example explaining the use case.

Binary Indexed Tree Operation Combination And Corresponding Algorithm						
Test Type Code	**Update Operation**	**Query Operation**	**Algorithm**	**Corresponding API's to execute**	**Comment**	**Example**
1	Point Update	Point Query (Frequency)	Update & Query on single BIT array	Update(BIT1, index, value) Query(BIT1, index) - Query(BIT1, index-1)	Alternative 1: Query(index) using common ancestor technique. Alternative 2: This query can be answered in O(1) time by trading off for O(n) space.	A = [1 2 3 4 5]; Update(2, 3) = [1 5 3 4 5] Query(2) = 5, Query(3) = 3
2	Point Update	Point Query (Cumulative Frequency)	Update & Query on single BIT array	Update(BIT1, index, value) Query(BIT1, index)		A = [1 2 3 4 5]; Update(2, 3) = [1 5 3 4 5] Query(2) = 6, Query(3) = 9

3	Point Update	Range Query (Frequency)	Update & Query on single BIT array
			Perform operation 1 on each index in the range
			Range = [L,R]
				Update(BIT1, index, value)
				for(index from L to R)
				{
				Query(BIT1, index) - Query(BIT1, index-1)
				}
					This condition is ideally not interesting. But has to be covered in order to cover all scenarios and to also give one concrete meaning to this scenario.
					Others can have their own definition. This query can be answered in O(k) time by trading off for O(n) space.
						A = [1 2 3 4 5];
						Update(2, 3) = [1 5 3 4 5]
						Query(2, 4) = [5 3 4]

4	Point Update	Range Query (Cumulative Frequency)	Update & Query on single BIT array Range = [L,R]	Update(BIT1, index, value) Query(BIT1, R) - Query(BIT1, L - 1)		A = [1 2 3 4 5]; Update(2, 3) = [1 5 3 4 5] Query(2, 4) = Query(4) - Query(1) = 12
5	Range Update	Point Query (Frequency)	Update & Query on two BIT arrays Range = [L,R]	Update(BIT1, L, value) Update(BIT1, R+1, -value) Update(BIT2, L, (L-1)*value) Update(BIT2, R+1, -value*R) Query(BIT1, BIT2, index) - Query(BIT1, BIT2, index-1)	Operation 1 techniques does not apply here. Query(BIT1, BIT2, index) = index*sum(BIT1,index) - sum(BIT2,index)	A = [1 2 3 4 5]; Update(2, 4, 3) = [1 5 6 7 5] Query(2) = 5, Query(3) = 6
6	Range Update	Point Query (Cumulative Frequency)	Update & Query on two BIT arrays Range = [L,R]	Update(BIT1, L, value) Update(BIT1, R+1, -value) Update(BIT2, L, (L-1)*value) Update(BIT2, R+1, -value*R) Query(BIT1, BIT2, index)	Query(BIT1, BIT2, index) = index*sum(BIT1,index) - sum(BIT2,index)	A = [1 2 3 4 5]; Update(2, 4, 3) = [1 5 6 7 5] Query(2) = 6, Query(3) = 12

7	Range Update	Range Query (Frequency)	Update & Query on two BIT arrays
			Perform operation 1 on each index in the range
			Range = [L,R]
				Update(BIT1, L, value)
				Update(BIT1, R+1, -value)
				Update(BIT2, L, (L-1)*value)
				Update(BIT2, R+1, -value*R)
				for(index from L to R)
				{
				Query(BIT1, BIT2, index) - Query(BIT1, BIT2, index-1)
				}
					Query(BIT1, BIT2, index) = index*sum(BIT1,index) - sum(BIT2,index)	A = [1 2 3 4 5];
						Update(2, 4, 3) = [1 5 6 7 5]
						Query(2, 4) = [5 6 7]

8	Range Update	Range Query	Update & Query on two BIT arrays	Update(BIT1, L, value)	Query(BIT1, BIT2, index) = index*sum(BIT1,index) -	A = [1 2 3 4 5];

		(Cumulative Frequency)	Range = [L,R]	Update(BIT1, R+1, -value) Update(BIT2, L, (L-1)*value) Update(BIT2, R+1, -value*R) Query(BIT1, BIT2, R) - Query(BIT1, BIT2, L-1)	sum(BIT2,index)	Update(2, 4, 3) = [1 5 6 7 5] Query(2, 4) = Query(4) - Query(1) = 18

See also

- Order statistic tree
- Prefix sums

References

1. Boris Ryabko (1989). "A fast on-line code" (http://boris.ryabko.net/dan1989.pdf) (PDF). *Soviet Math. Dokl*. **39** (3): 533–537.
2. Boris Ryabko (1992). "A fast on-line adaptive code" (http://boris.ryabko.net/ryabko1992.pdf) (PDF). *IEEE Trans.on Inform.Theory*. **28** (1): 1400–1404.
3. Peter M. Fenwick (1994). "A new data structure for cumulative frequency tables" (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.8917&rep=rep1&type=pdf). *Software: Practice and Experience*. **24** (3): 327–336. CiteSeerX 10.1.1.14.8917 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.8917). doi:10.1002/spe.4380240306 (https://doi.org/10.1002%2Fspe.4380240306).
4. Pushkar Mishra (2013). "A New Algorithm for Updating and Querying Sub-arrays of Multidimensional Arrays". arXiv:1311.6093 (https://arxiv.org/abs/1311.6093). doi:10.13140/RG.2.1.2394.2485 (https://doi.org/10.13140%2FRG.2.1.2394.2485).

External links

- A tutorial on Fenwick Trees on TopCoder (https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/)
- An article on Fenwick Trees on Algorithmist (http://www.algorithmist.com/index.php/Fenwick_tree)
- An entry on Fenwick Trees on Polymath wiki (http://michaelnielsen.org/polymath1/index.php?title=Updating_partial_sums_with_Fenwick_tree)
- stack exchange (https://cs.stackexchange.com/q/10538)