Binary Indexed Tree or Fenwick tree

Let us consider the following problem to understand Binary Indexed Tree.

We have an array arr[0 . . . n-1]. We should be able to
1 Find the sum of first i elements.
2 Change value of a specified element of the array arr[i] = x where 0 <= i <= n-1.

A **simple solution** is to run a loop from 0 to i-1 and calculate sum of elements. To update a value, simply do arr[i] = x. The first operation takes O(n) time and second operation takes O(1) time. Another simple solution is to create another array and store sum from start to i at the i'th index in this array. Sum of a given range can now be calculated in O(1) time, but update operation takes O(n) time now. This works well if the number of query operations are large and very few updates.

Can we perform both the operations in O(log n) time once given the array?
One Efficient Solution is to use Segment Tree that does both operations in O(Logn) time.

Using Binary Indexed Tree, we can do both tasks in O(Logn) time. The advantages of Binary Indexed Tree over Segment are, requires less space and very easy to implement..

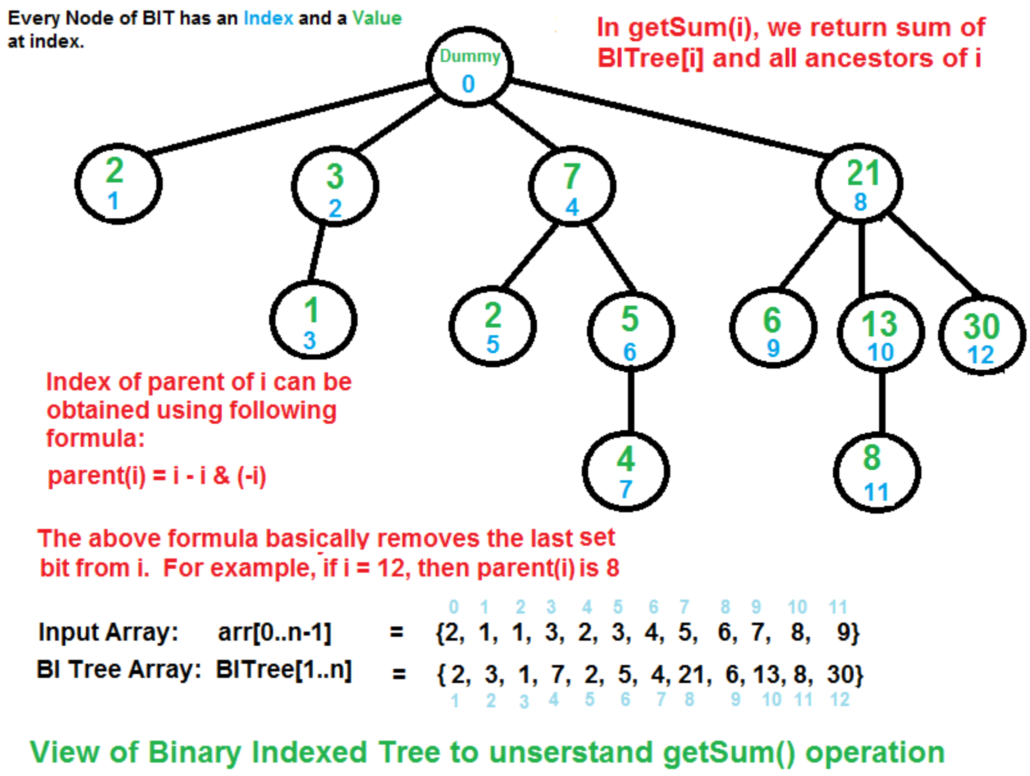
Representation
Binary Indexed Tree is represented as an array. Let the array be BITree[]. Each node of Binary Indexed Tree stores sum of some elements of given array. Size of Binary Indexed Tree is equal to n where n is size of input array. In the below code, we have used size as n+1 for ease of implementation.

Construction
We construct the Binary Indexed Tree by first initializing all values in BITree[] as 0. Then we call update() operation for all indexes to store actual sums, update is discussed below.

Operations

```
getSum(index): Returns sum of arr[0..index]
// Returns sum of arr[0..index] using BITree[0..n].  It assumes that
// BITree[] is constructed for given array arr[0..n-1]
1) Initialize sum as 0 and index as index+1.
```

```
2) Do following while index is greater than 0.
...a) Add BITree[index] to sum
...b) Go to parent of BITree[index].  Parent can be obtained by removing
     the last set bit from index, i.e., index = index - (index & (-index))
3) Return sum.
```



The above diagram demonstrates working of getSum(). Following are some important observations.

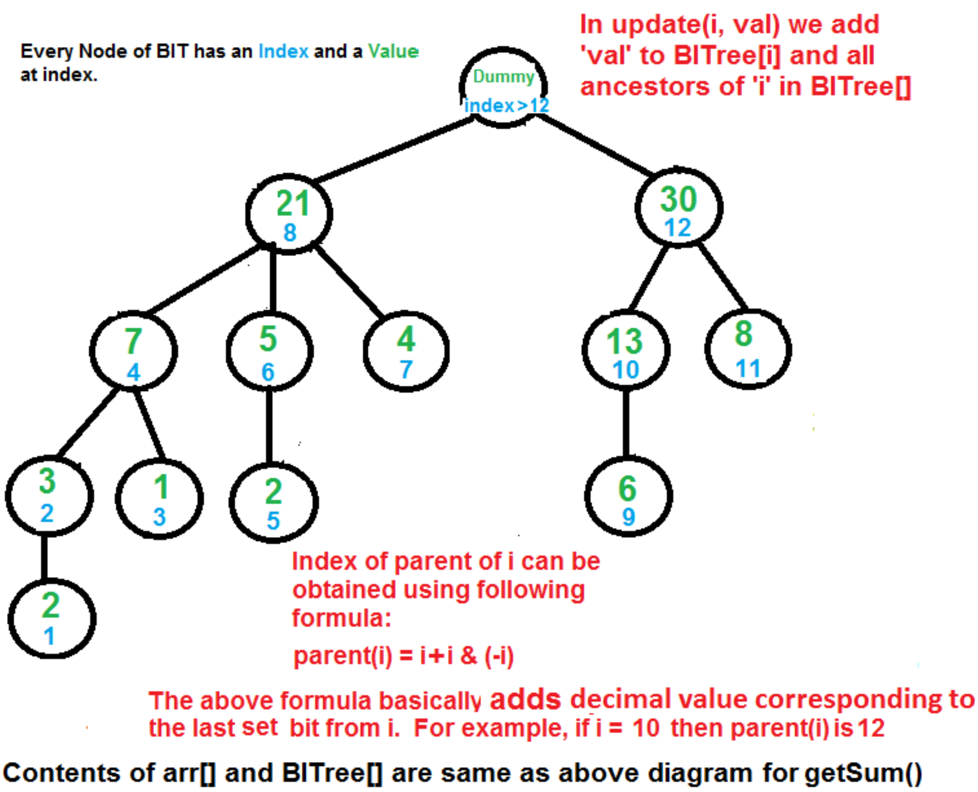
Node at index 0 is a dummy node.

A node at index y is parent of a node at index x, iff y can be obtained by removing last set bit from binary representation of x.

A child x of a node y stores sum of elements from of y(exclusive y) and of x(inclusive x).

```
update(index, val): Updates BIT for operation arr[index] += val
// Note that arr[] is not changed here.  It changes
// only BI Tree for the already made change in arr[].
1) Initialize index as index+1.
2) Do following while index is smaller than or equal to n.
```

```
...a) Add value to BITree[index]
...b) Go to parent of BITree[index].  Parent can be obtained by removing
     the last set bit from index, i.e., index = index - (index & (-index))
```



View of Binary Indexed Tree to understand update() operation

The update process needs to make sure that all BITree nodes that have arr[i] as part of the section they cover must be updated. We get all such nodes of BITree by repeatedly adding the decimal number corresponding to the last set bit.

How does Binary Indexed Tree work?
The idea is based on the fact that all positive integers can be represented as sum of powers of 2. For example 19 can be represented as 16 + 2 + 1. Every node of BI Tree stores sum of n elements where n is a power of 2. For example, in the above first diagram for getSum(), sum of first 12 elements can be obtained by sum of last 4 elements (from 9 to 12) plus sum of 8 elements (from 1 to 8). The number of set bits in binary representation of a number n is O(Logn). Therefore, we traverse at-most O(Logn) nodes in both getSum() and update() operations. Time complexity of construction is O(nLogn) as it calls update() for all n elements.

Implementation:
Following is C++ implementation of Binary Indexed Tree.

```
// C++ code to demonstrate operations of Binary Index Tr
#include <iostream>
using namespace std;

/*           n  --> No. of elements present in input ar
    BITree[0..n] --> Array that represents Binary Indexe
    arr[0..n-1]  --> Input array for whic prefix sum is

// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[].
int getSum(int BITree[], int n, int index)
{
    int sum = 0; // Iniialize result

    // index in BITree[] is 1 more than the index in arr
    index = index + 1;

    // Traverse ancestors of BITree[index]
    while (index>0)
    {
        // Add current element of BITree to sum
        sum += BITree[index];

        // Move index to parent node
        index -= index & (-index);
    }
    return sum;
}

// Updates a node in Binary Index Tree (BITree) at given
// in BITree.  The given value 'val' is added to BITree[
// all of its ancestors in tree.
void updateBIT(int *BITree, int n, int index, int val)
{
    // index in BITree[] is 1 more than the index in arr
    index = index + 1;

    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
       // Add 'val' to current node of BI Tree
       BITree[index] += val;

       // Update index to that of parent
       index += index & (-index);
    }
}

// Constructs and returns a Binary Indexed Tree for give
// array of size n.
```

```
int *constructBITree(int arr[], int n)
{
    // Create and initialize BITree[] as 0
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    // Store the actual values in BITree[] using update(
    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);

    // Uncomment below lines to see contents of BITree[]
    //for (int i=1; i<=n; i++)
    //      cout << BITree[i] << " ";

    return BITree;
}


// Driver program to test above functions
int main()
{
    int freq[] = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};
    int n = sizeof(freq)/sizeof(freq[0]);
    int *BITree = constructBITree(freq, n);
    cout << "Sum of elements in arr[0..5] is "
         << getSum(BITree, n, 5);

    // Let use test the update operation
    freq[3] += 6;
    updateBIT(BITree, n, 3, 6); //Update BIT for above cl

    cout << "\nSum of elements in arr[0..5] after update
         << getSum(BITree, n, 5);

    return 0;
}
```

Output:

```
Sum of elements in arr[0..5] is 12
Sum of elements in arr[0..5] after update is 18
```

Can we extend the Binary Indexed Tree for range Sum in Logn time?
This is simple to answer. The rangeSum(l, r) can be obtained as getSum(r) – getSum(l-1).

Applications:

Used to implement the arithmetic coding algorithm. Development of operations it supports were primarily motivated by use in that case. See this for more details.

References:

http://en.wikipedia.org/wiki/Fenwick_tree
http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees