**Experiment No. 3**

**Title: Implementation of Fenwick Tree**

**Batch: B-4** **Roll No: 16010422234** **Name: Chandana Ramesh Galgali**

**Experiment No.: 3**

**Aim: Implementation of Fenwick Tree operations**

---

**Resources needed:** Text Editor, C/C++ IDE

---

**Theory:**

**Binary Indexed Tree or Fenwick Tree:**

Binary Indexed Tree also called Fenwick Tree provides a way to represent an array of numbers in an array, allowing prefix sums to be calculated efficiently. For example, an array is [2, 3, -1, 0, 6] the length 3 prefix [2, 3, -1] with sum 2 + 3 + -1 = 4). Calculating prefix sums efficiently is useful in various scenarios. Let's start with a simple problem.

We are given an array a[], and we want to be able to perform two types of operations on it.

1. Change the value stored at an index i. (This is called a point update operation)
2. Find the sum of a prefix of length k. (This is called a range sum query)

**Simple Solution is :**

```
int a[] = {2, 1, 4, 6, -1, 5, -32, 0, 1};
void update(int i, int v)   //assigns value v to a[i]
{
    a[i] = v;
}
int prefixsum(int k)    //calculate the sum of all a[i] such that 0 <= i < k
{
   int sum = 0;
   for(int i = 0; i < k; i++)
        sum += a[i];
   return sum;
}
```

But the time required to calculate a prefix sum is proportional to the length of the array, so this will usually time out when a large number of such intermingled operations are performed. One efficient solution is to use a segment tree that can perform both operations in O(logN) time. Using binary Indexed trees also, we can perform both the tasks in O(logN) time. But then why learn another data structure when segment trees can do the work for us. It's because binary indexed trees require less space and are very easy to implement during programming contests.

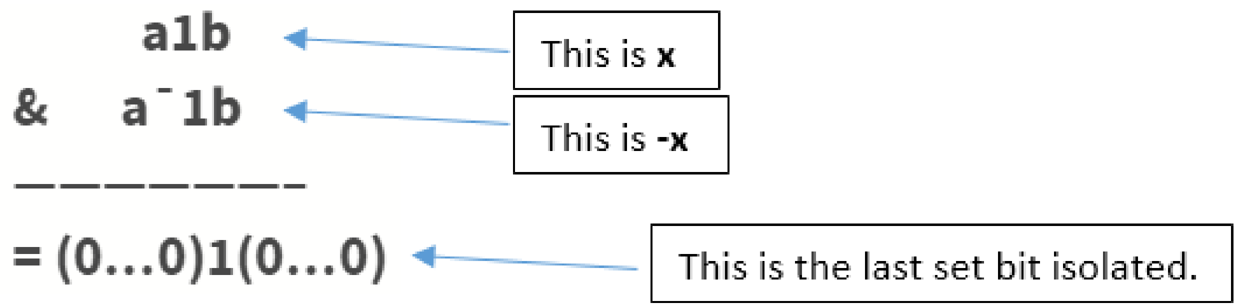
**Understanding Bit manipulation:**

This is the last set bit, and we need to isolate this.

Let's take an example, a number x = 1110(in binary),

| Binary digit | 1 | 1 | 1 | 0 |
|---|---|---|---|---|
| Index | 3 | 2 | 1 | 0 |

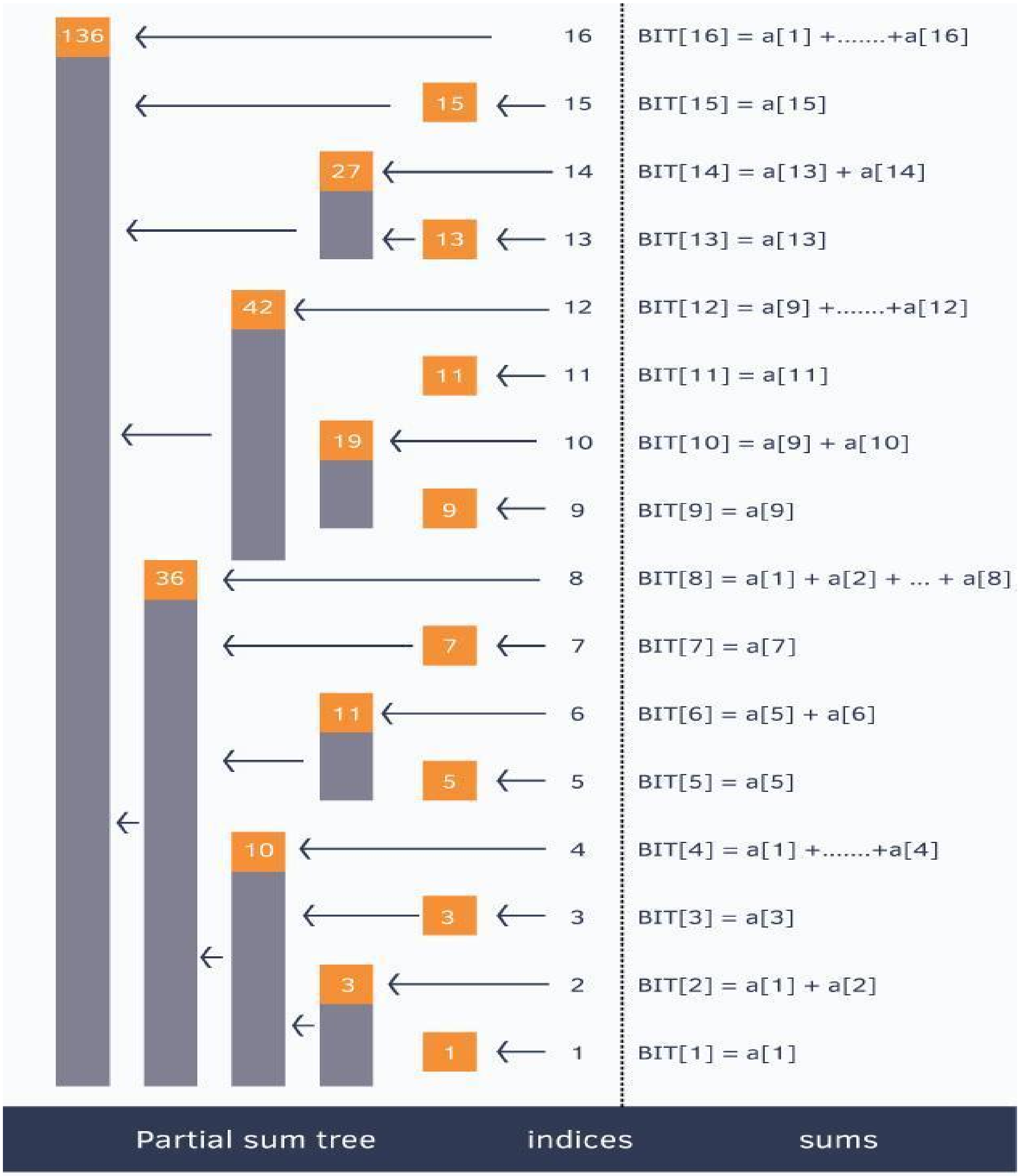
**x** = 2's complement of x = **(a1b)'** + 1 = **a'0b'** + 1 = **a'0(0….0)' + 1 = a'0(1...1) + 1 = a'1(0…0) = a'1b**




**Basic Idea of Binary Indexed Tree:**

We know the fact that each integer can be represented as a sum of powers of two. Similarly, for a given array of size N, we can maintain an array BIT[] such that, at any index we can store the sum of some numbers of the given array. This can also be called a partial sum tree.

Let Us Consider int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};




The value in the enclosed box represents BIT[index].

The above picture shows the binary indexed tree, each enclosed box of which denotes the value BIT[index] and each BIT[index] stores a partial sum of some numbers.

```
          {           a[x],                  if x is odd
BIT[x] =                   a[1] + ... + a[x],     if x is power of 2
              }
```

To generalize this every index i in the BIT[] array stores the cumulative sum from the index i to i - (1<<r) + 1 (both inclusive), where r represents the last set bit in the index i

Sum of first 12 numbers in array

a[] = BIT[12] + BIT[8] = (a[12] + … + a[9]) + (a[8] + … + a[1])

sum of first 6 elements = BIT[6] + BIT[4] = (a[6] + a[5]) + (a[4] + … + a[1])

Sum of first 8 elements = BIT[8] = a[8] + … + a[1]

we call update() operation for each element of a given array to construct the Binary Indexed Tree.

The update() operation is discussed below.

```
void update(int x, int delta)       //add "delta" at index "x"
{
for(; x <= n; x += x&-x)
 BIT[x] += delta;
}
```

Suppose we call update(13, 2).

Here we see from the above figure that indices 13, 14, 16 cover index 13 and thus we need to add 2 to them also.

Initially x is 13, we update BIT[13]

BIT[13] += 2;

Now isolate the last set bit of x = 13(1101) and add that to x , i.e. x += x&(-x)

Last bit is of x = 13(1101) is 1 which we add to x, then x = 13+1 = 14, we update BIT[14]

BIT[14] += 2;

Now 14 is 1110, isolate last bit and add to 14, x becomes 14+2 = 16(10000), we update BIT[16]

BIT[16] += 2;

How to **query** such structure for prefix sums?

```
int query(int x)      //returns the sum of first x elements in given array a[]
{
     int sum = 0;
     for(; x > 0; x -= x&-x)
         sum += BIT[x];
     return sum;
}
```

The above function query() returns the sum of first x elements in given array. Let's see how it works.

Suppose we call **query(14)**, initially **sum = 0**

x is 14(1110) we add BIT[14] to our sum variable, thus **sum = BIT[14]** = (a[14] + a[13])

now we isolate the last set bit from **x = 14(1110)** and subtract it from x

last set bit in **14(1110)** is 2(10), thus **x = 14 – 2 = 12**

we add BIT[12] to our sum variable, thus[

**sum = BIT[14] + BIT[12] = (a[14] + a[13]) + (a[12] + … + a[9])**

again we isolate last set bit from x = 12(1100) and subtract it from x

last set bit in 12(1100) is 4(100), **thus x = 12 – 4 = 8**

we add BIT[8] to our sum variable, thus

**sum = BIT[14] + BIT[12] + BIT[8]** = (a[14] + a[13]) + (a[12] + … + a[9]) + (a[8] + … + a[1])

once again we isolate last set bit from x = 8(1000) and subtract it from x

last set bit in 8(1000) is 8(1000), **thus x = 8 – 8 = 0**

since x = 0, the for loop breaks and we return the prefix sum.

**Space Complexity**: **O(N)** for declaring another array of size N

**Time Complexity**: **O(logN)** for each operation(update and query as well)

---

**Activity:**

Write a program to solve range-based query over an array for performing sum and update operation using Fenwick tree.

---

**Solution:**

```
class FenwickTree:
    def __init__(self, size):
        self.size = size
        self.tree = [0] * (size + 1)
```

```
    def _lsb(self, i):
        return i & -i
    def update(self, i, delta):
        i += 1
        while i <= self.size:
            self.tree[i] += delta
            i += self._lsb(i)
    def prefix_sum(self, i):
        i += 1
        result = 0
        while i > 0:
            result += self.tree[i]
            i -= self._lsb(i)
        return result
    def range_sum(self, left, right):
        return self.prefix_sum(right) - self.prefix_sum(left - 1)
if __name__ == "__main__":
    arr = list(map(int, input("Enter array elements separated by
space: ").split()))
    fenwick_tree = FenwickTree(len(arr))
    for i, val in enumerate(arr):
        fenwick_tree.update(i, val)
    q = int(input("Enter number of queries: "))
    for _ in range(q):
        query_type = input("Enter query type (sum/update): ").lower()
        if query_type == "update":
            idx, value = map(int, input("Enter index and value
separated by space: ").split())
            fenwick_tree.update(idx, value - arr[idx])
            arr[idx] = value
        elif query_type == "sum":
            left, right = map(int, input("Enter left and right indices
for sum query (separated by hyphen): ").split("-"))
            print(f"Sum of range [{left}, {right}]:
{fenwick_tree.range_sum(left, right)}")
```

```
        else:
            print("Invalid query type. Please enter 'sum' or 
'update'.")
```

**Output:**

```
PS C:\Users\chand\Downloads\IV SEM> & C:/Users/chand
/AppData/Local/Microsoft/WindowsApps/python3.11.exe
"c:/Users/chand/Downloads/IV SEM/CPL/exp3.py"
Enter array elements separated by space: 4 7 -6 3 1
5 2 1 10 3 2
Enter number of queries: 3
Enter query type (sum/update): sum
Enter left and right indices for sum query (separate
d by hyphen): 0-6
Sum of range [0, 6]: 16
Enter query type (sum/update): update
Enter index and value separated by space: 4 7
Enter query type (sum/update): sum
Enter left and right indices for sum query (separate
d by hyphen): 0-6
Sum of range [0, 6]: 22
PS C:\Users\chand\Downloads\IV SEM>
```

---

**Outcomes: Understand the fundamental concepts for managing the data using different data structures such as lists, queues, trees etc.**

---

**Conclusion: (Conclusion to be based on the objectives and outcomes achieved)**

The experiment successfully achieved its objective of implementing Fenwick Tree operations, providing a versatile data structure for efficient range-based queries and updates. This implementation can serve as a valuable tool for solving various problems requiring such operations, including tasks in computational geometry, data compression, and more.

---

**References:**

1. https://www.hackerearth.com/practice/data-structures/advanced-data-ructures/segment-trees/tutorial/
2. https://cp-algorithms.com/data_structures/segment_tree.html