**CSE4303: DATA STRUCTURES**

# ASSIGNMENT ON
# BINARY INDEXED TREE

**NAME:** Chowdhury Ashfaq
**STUDENT ID:** 200042123
**PROGRAM:** BSc. In SWE
**DEPARTMENT:** CSE
**SUBMISSION DATE:** 20/12/2022

## What is a Binary Indexed Tree?

A Binary Indexed Tree is such a data structure which allows to store the prefix sum of an array efficiently in an array.
The Binary Indexed tree is used because it required less time to perform operations in it so it's more efficient. The data structure have been designed in such a way so that we can compute range sums efficiently.

## What are the operations we can perform on a Standard Binary Indexed Tree?

The operations that can be performed on a Binary Indexed Tree is performing range sum queries and update operation.

## Advantages of Binary Indexed Tree

Binary Indexed Tree is preferred over other data structures in case of finding out range sums. But Range sums can also be calculated through segment tree. So why'd we use Binary Indexed tree? The reasons are:

1. Requires less space than Segment Tree.
2. Requires less time for updating.
3. Very easy to implement.
4. Searching and execution is fast.
5. Hierarchical structure and easy to understand.

## Disadvantages of Binary Indexed Tree

No data structure comes without any disadvantage. There are a few disadvantage of Binary Indexed Tree which are given below:

1. Accessing is slower than in an array.
2. Deletion of a node is complex.
3. There are many useless null pointers.

## Sample Code of a basic implementation of Binary Indexed Tree

The implementation of Binary Indexed Tree in C++ have been given below:

```cpp
#include <iostream>
#include<vector>
#include<map>

using namespace std;

int getSum(int BITree[], int index)
{
    int sum = 0;
    index = index + 1;


    while (index > 0)
    {

        sum += BITree[index];
        index -= index & (-index);
    }
    return sum;
}

void updateBIT(int BITree[], int n, int index, int val)
{
    index = index + 1;

    while (index <= n)
    {
        BITree[index] += val;
        index += index & (-index);
    }
}

int* constructBITree(int arr[], int n)
{
    int* BITree = new int[n + 1];
    for (int i = 1; i <= n; i++)
        BITree[i] = 0;


    for (int i = 0; i < n; i++)
        updateBIT(BITree, n, i, arr[i]);

    return BITree;
}
```

## Complexity Analysis

We mainly have two operations that are performed on a Binary Indexed Tree.
The update operation takes time equal to the height of the tree at the worst case which is O(log n) where n is the number of elements in the array.

Again, performing range sum also takes O(log n) time which is efficient than Segment tree. Building a BIT takes O(n Log n) time because it uses the update operation to build the tree.

The space complexity of Binary Indexed Tree is O(n) which is lower than Segment tree.

## Use cases

Binary Indexed Tree has quite a number of use cases. Few of them are:

1. To implement the arithmetic coding algorithm Binary Indexed Tree is most primarily used.
2. To count inversions in an array Binary Indexed Tree is used.

A problem where Binary Indexed Tree can be used is:

There is an array b with m elements and an integer k. Omit has q query in type [L,R], for i-Th query print the length of the shortest segment $[x,y] \in [L_i,R_i]$, such that $k \mid \prod_{j=x}^{y} b_j$.