Binary Indexed Tree

Mirza Mohammad Azwad (Student ID: 200042121)

CSE 4303 Data Structures
BScEngg in Software Engineering(SWE), Department Computer Science and Engineering, Islamic University of Technology(IUT)
Board Bazar, Gazipur, Dhaka, Bangladesh
E-mail: mirzaazwad@iut-dhaka.edu

11th December, 2022

I Topics

- What is a Binary Indexed Tree?
- What are the operations we can perform on a Standard Binary Indexed Tree?
- Advantages and Disadvantages
- Sample Code of a basic implementation of Binary Indexed Tree
- Complexity Analysis
- Usecases (Describe a problem where we can use Binary Indexed Tree)

II What is a Binary Indexed Tree?

Binary Indexed Tree(BIT) or Fenwick Tree is a similar concept to sparse table and segment tree discussed in class. But unlike a sparse table, it is more memory efficient and unlike a segment tree, it's easier to code or implement. But unlike a segment tree, the underlying concept is slightly more complicated but in a sense similar to a sparse table. The key idea is that we use the binary representation of indices to create segments upon which queries can be performed. The indexing is done using the concept of binary numbers. It would be further discussed as to how we build a tree, perform a query or update the tree soon. There are also multiple different implementations, although in this document I focus on BIT for 1D ranged queries BIT can also be performed for 2D range queries.

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

1 Build Tree

Let's say we are given an array **a** as **[1,3,4,11,7,4,6,9]** where 0¡=i¡=8 and a_i contains integers. To form a BIT, we use an array to store the results. This array tends to have a size of n+1 for an input array of size n which in this case is 8+1=9 as BIT tends to store values using one-based indexing for the tree array. The first thing we have to consider is how BIT determines its parent node. To determine the parent node in BIT we basically invert the rightmost set bit or the rightmost 1. Secondly how do we build the tree? We follow these steps:

- We essentially take the binary combination of the index as per one-based indexing. Then for each number the lowest power of 2 indicates the number of positions to be moved from the starting position inclusive. The rest, that being except the lowest power of 2 indicates the starting position.
- If there is only one power of 2 we take that as the number of positions to move from start, and we take 0 as the starting index.

Using the aforementioned information, we can build a Binary Indexed Tree. Another important thing to note is that a fundamental property of binary numbers is that any decimal number can be represented in binary using at most $\lceil logn \rceil$ bits where n indicates the number itself. By this property, we can conclude that the height in this case would be at worst $\lceil logn \rceil$ as it would take the flipping of $\lceil logn \rceil$ bits to reach the root or 0.

2 Perform Query

First and foremost, let's equate the move to parent by inverting the rightmost bit. It can be given by:

$parent = child - child\Lambda(\neg child)$

This expression is the key to BIT operations. Because for each child node, the idea is to calculate the query for a given range, we move up from that node to the root node that being 0. This is the essential idea of performing queries to perform a query for a mentioned range say sum of index 0 to 6, we can check the 7th index for BIT as 6+1 and then move up to the root and sum all the values encountered along the path. If you want to get the sum for 5 to 6, find the sum for 0 to 6 using the method above and then 0 to 4 and then subtract the first sum from the second.

3 Update Tree

$index = index + index\Lambda(\neg index)$

This expression is the key expression to update the BIT array. This expression helps us identify which indices need to be updated when BIT needs updation. Let's say we want to update index 2. First we find the equivalent BI tree index that is 2+1=3. After finding that we find all the indices that needs updating by placing the index in the expression above and obtaining the next index in the Left Hand Side of the expression. We keep obtaining indices and updating the found positions until the index exceeds the bounds of the tree array, that's when we stop. Why does this work? Essentially in a BIT we split the array based on segments determined by the powers of 2. So the idea is to update only the nodes corresponding to the set bit operations so as to get an efficient means of updating the tree. Say we want to update index 3. In which indices are the value in index 3 involved is the key question. To answer that we first take the actual index and then increment it so that it corresponds to the BI Tree index. And then the indices it is involved with is the indices after 3 which for the tree is 4. To find that we can just add a 1 to the rightmost position and keep adding to get the next indices to be updated. Eventually we cross the length and that's when we stop.

IV Advantages and Disadvantages

1 Advantages

The biggest advantage of BIT is that it is easy to code. This easy-to-code helps greatly, especially in competitive programming where time is key. It is quick to implement and thus helps solve some complex problems faster.

2 Disadvantages

The biggest disadvantage is that understanding the underlying concepts appears to be pretty complicated and a solid understanding of binary numbers and bitwise operations is key to grasping how the tree is actually working. For update operations, segment tree with lazy propagation often works faster.

V Sample Code of a basic implementation of Binary Indexed Tree

1 Update Tree

```
void update(int BIT[], int len,
int index, int value)
{
    index++;
    while (index <= len)
    {
    BIT[index] += value;
    index += index & (-index);
    }
}
```

2 Build Tree

```
int *build(int input[], int len)
{
    int *BIT = new int[len+1];
    for (int i=1; i<=len; i++)
    {
        BIT[i] = 0;
    }

    for (int i=0; i<len; i++)
    {
        update(BIT, len, i, input[i]);
    }
    return BIT;
}
```

3 Perform Query

```
int query(int BIT[], int index)
{
    int result = 0;
    index = index + 1;
    while (index>0)
    {
        sum += BIT[index];
        index -= index & (-index);
    }
    return sum;
}
```

VI Complexity Analysis

1 Time Complexity

The time complexity for the update is O(nlogn), for query, it is O(logn), and building the tree itself can be done in O(n). Upon observing this we can say that the complexity is similar to a segment tree or sparse table. But a sparse table appears to be more efficient for performing overlapping queries as it can perform overlapping queries in O(1). But BIT would still require O(logn).

2 Space Complexity

BIT appears to be more space efficient than both sparse table and segment trees. A sparse table requires O(nlogn) space which makes it less efficient in terms of memory use compared to segment trees in most cases can be computed in O(n) space. But for segment trees we need an array about 3 times the size to store the tree values but here we can do it with an array of size n+1 which makes it relatively more efficient.

VII Usecases

BIT is very efficient to implement for certain problems that involve range queries such range minimum query(RMQ), finding the gcd for a given range, finding sum over a given range etc. An example problem that seems to be helpful to demonstrate how BIT is useful would be

1 Problem

Curious Robin Hood from LightOJ.

2 Solution

A Code

```
#include <iostream>
#include<vector>
using namespace std;

template <typename T>
class BinaryIndexedTree
{
private:
  vector<T> BIT;
  int length;

public:
  BinaryIndexedTree(int n)
  {
    length = n + 1;
    BIT.assign(n + 1, 0);
  }

  void update(int index, T value)
  {

    while (index <= length-1)
    {
      BIT[index] = BIT[index] + value;
      index = index + (index & (-index));
    }
  }

  T query(int index)
  {
```

```
    T ans;
    ans = 0;

    while (index > 0)
    {
      ans = ans + BIT[index];
      index = index - (index & (-index));
    }

    return ans;
  }

  void clear(){
    BIT.clear();
  }
};

namespace Problem
{
  class Program
  {
  private:
    int x;
    int y;
    int w;
    int choice;
    long long ans;
    int q;
    int ch;
    long long n;
  public:
    Program(int t = 1)
    {
      cout << "Case ";
      for (int i = 1; i <= t; i++)
      {
        cout << i<<": "<<endl;
        takeInput();
        solve();
        clearSpace();
      }
    }

    void takeInput()
    {
    }

    void clearSpace()
    {

    }
    void solve()
    {

        cin>>n>>q;
        BinaryIndexedTree<long long> *BIT=
        new BinaryIndexedTree<long long>(n);

        for (int i = 1; i <= n; i++)
        {
          cin>>x;
          BIT->update(i,x);
        }

        for (int i = 0; i < q; i++)
        {
          cin>>ch;
          switch (ch)
          {
          case 1:
            cin>>x;
            x++;
            ans = BIT->query(x) -
            BIT->query(x - 1);
            BIT->update(x, -ans);
            cout<<ans<<endl;
            break;
          case 2:
            cin>>x;
            x++;
            cin>>w;
            BIT->update(x, w);
            break;

          case 3:
            cin>>x>>y;
            x++;
            y++;
            ans = BIT->query(y) -
            BIT->query(x - 1);
            cout<<ans<<endl;
            break;
          }
        }
      }
  };
}
signed main()
{
  int tc;
  cin >> tc;
  Problem::Program program =
  Problem::Program(tc);
  return 0;
}
```

B Explanantion

The 3 cases can be represented with some form of range query, here we are using a BIT for the purpose of simplicity. To represent the first case, its simple we just find all the money of a particular sack, say i. For this we query the current value of i and the value for i-1. The second case involves updating as we need to update all the values after a particular index. In my opinion, using segment tree with lazy propagation would be more efficient here but still regardless, using a BIT we can then carry out an update operation by first inserting the value at i and then updating all the indices after that. The third case is just a range query where we are given a range we need to see if its a valid range and if it is we need to output the result for the range.

VIII References

- Codeforces Blog on BIT Blog 619
- Geeks For Geeks GFG BIT Analysis
- CP Algorithms BIT explanation BIT detailed explanation and practice problems

- Hackerearth BIT Tutorial BIT Simple tutorial
- Shafayet's Blog tutorial BIT tutorial