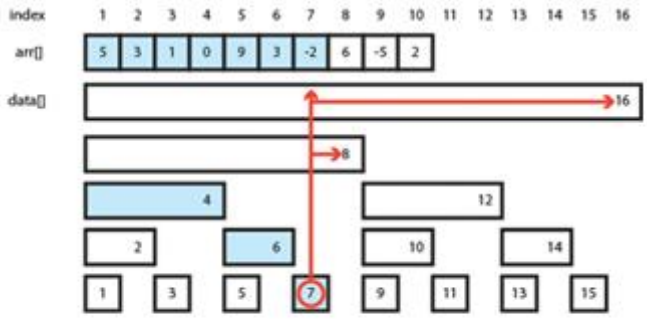
BỘ GIÁO DỤC & ĐÀO TẠO
TRƯỜNG ĐẠI HỌC KIẾN TRÚC ĐÀ NẴNG
KHOA: CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN

Đề tài: Tìm hiểu, xây dựng cây BIT (binary indexed tree) và xây dựng ứng dụng minh hoạ



Giảng viên hướng dẫn: ĐỖ PHÚC HẢO
Sinh viên thực hiện : NGUYỄN CHÍNH NGHĨA
NGÔ QUỐC VIỆT
Lớp : 19CT3

Tháng 5 năm 2021

MỤC LỤC

* Lý thuyết về cây Binary Indexed Tree (Fenwick tree):

Một **Fenwick tree** hoặc **cây được lập chỉ mục nhị phân (BIT)** là một cấu trúc dữ liệu mà hiệu quả có thể cập nhật các yếu tố và tính toán ***khoản tiền tiền tố*** trong một bảng số.

Khi so sánh với một dãy số phẳng, cây BIT đạt được sự cân bằng tốt hơn nhiều giữa hai hoạt động: cập nhật phần tử và tính tổng tiền tố. Trong một mảng phẳng của số, bạn có thể lưu trữ các phần tử hoặc tổng tiền tố:

- Trong trường hợp đầu tiên, tính toán tổng tiền tố yêu cầu thời gian tuyến tính;
- Trong trường hợp thứ hai, việc cập nhật các phần tử của mảng yêu cầu thời gian tuyến tính (trong cả hai trường hợp, hoạt động khác có thể được thực hiện trong thời gian không đổi).

- Cây BIT cho phép cả hai hoạt động được thực hiện trongthời gian. Điều này đạt được bằng cách biểu diễn các số dưới dạng cây , trong đó giá trị của mỗi nút là tổng các số trong cây con đó. Cấu trúc cây cho phép các hoạt động được thực hiện chỉ bằng cách sử dụng truy cập nút.

Cấu trúc này được Boris Ryabko đề xuất vào năm 1989 với một sửa đổi bổ sung được xuất bản vào năm 1992. Sau đó nó được biết đến với cái tên cây Fenwick theo tên "Peter Fenwick", người đã mô tả cấu trúc này trong bài báo năm 1994 của ông.

* Thuật toán Binary Indexed Tree:

1.Giới thiệu thuật toán:

Với một bảng các phần tử, đôi khi người ta mong muốn tính toán tổng số giá trị đang chạy lên đến mỗi chỉ mục theo một số phép toán nhị phân kết hợp (phép cộng trên số nguyên là phổ biến nhất cho đến nay). Cây BIT cung cấp một phương pháp để truy vấn tổng số đang chạy ở bất kỳ chỉ mục nào, ngoài việc cho phép thay đổi bảng giá trị cơ bản và có tất cả các truy vấn khác phản ánh những thay đổi đó.

Mỗi số nguyên có thể được biểu diễn như là tổng các lũy thừa của 2, hay biểu diễn được trong hệ cơ số 2. Theo cách này, tần số tích lũy có thể được biểu diễn như là tổng của các tập của các tần số con. Trong bài toán này, mỗi tập sẽ chứa một số liên tiếp các tần số.

Cây BIT được thiết kế đặc biệt để thực hiện thuật toán mã hóa số học , thuật toán này duy trì số lượng của mỗi ký hiệu được tạo ra và cần chuyển đổi chúng thành xác suất tích lũy của một ký hiệu nhỏ hơn một ký hiệu nhất định. Việc phát triển các hoạt động mà nó hỗ trợ chủ yếu được thúc đẩy bởi việc sử dụng trong trường hợp đó.

Sử dụng một cây BIT, nó chỉ yêu cầu các phép toán để tính bất kỳ tổng tích lũy mong muốn nào, hoặc nói chung hơn là tổng của bất kỳ dải giá trị nào (không nhất thiết phải bắt đầu từ 0).

2.Mô tả thuật toán:

Mặc dù trong khái niệm cây Fenwick là cây , nhưng trên thực tế, chúng được triển khai như một cấu trúc dữ liệu ngầm sử dụng một mảng phẳng tương tự như việc triển khai một đống nhị phân . Cho một chỉ số trong mảng đại diện cho một đỉnh, chỉ số của đỉnh cha hoặc con của đỉnh được tính thông qua các phép toán bit trên hệ nhị phânđại diện của chỉ số của nó. Mỗi phần tử của mảng chứa tổng được tính trước của một dải giá trị và bằng cách kết hợp tổng đó với các dải bổ sung gặp phải trong quá trình truyền tải lên đến gốc, tổng tiền tố sẽ được tính. Khi một giá trị bảng được sửa đổi, tất cả các tổng phạm vi có chứa giá trị đã sửa đổi lần lượt được sửa đổi trong quá trình truyền tải tương tự của cây. Tổng phạm vi được xác định theo cách mà cả truy vấn và sửa đổi đối với bảng đều được thực thi trong thời gian tiệm cận tương đương (trong trường hợp xấu nhất).

Quá trình ban đầu của việc xây dựng cây Fenwick trên một bảng giá trị chạy trong thời gian. Các hoạt động hiệu quả khác bao gồm định vị chỉ mục của một giá trị nếu tất cả các giá trị đều dương hoặc tất cả các chỉ số với một giá trị nhất định nếu tất cả các giá trị đều không âm. Cũng được hỗ trợ là tỷ lệ của tất cả các giá trị theo một hệ số không đổi trong thời gian.

Cây Fenwick dễ hiểu nhất bằng cách xem xét một mảng dựa trên một . Mỗi phần tử có chỉ số "i" là lũy thừa của 2 chứa tổng các "i" phần tử đầu tiên . Các phần tử có chỉ số là tổng của hai lũy thừa (riêng biệt) của 2 chứa tổng các phần tử kể từ lũy thừa đứng trước là 2. Nói chung, mỗi phần tử chứa tổng các giá trị kể từ gốc của nó trong cây và phần tử đó được tìm thấy bằng cách xóa bit ít quan trọng nhất trong chỉ mục.

Để tìm tổng cho bất kỳ chỉ mục nhất định nào, hãy xem xét sự mở rộng nhị phân của chỉ mục và thêm các phần tử tương ứng với mỗi 1 bit ở dạng nhị phân.

3. Ưu nhược điểm:

*** Ưu điểm:**

Bộ nhớ thấp, cài đặt đơn giản, có thể giải được nhiều bài toán về dãy số

Thời gian chạy: O(logn) với N là độ dài dãy cần quản lý

*** Nhược điểm**:

Không tổng quát bằng Segment Tree (Một cấu trúc dữ liệu giải thuật khác). Tất cả những bài giải được bằng Fenwick tree đều có thể giải được bằng Segment Tree. Nhưng chiều ngược lại thì không đúng. Khó hiểu.

4.Trình bày thuật toán:

4.1. Build cây từ 1 mảng INPUT:

VD: Cho 1 mảng a[N] phần tử với N nhập từ bàn phím Tạo cây BIT[] với N node bằng 0.

A[]	1	4	8	7	5	3	6	12	19	20	11	9	2	40	33

Đặt m = 2k.p (với p là số lẻ). Hay nói cách khác, k là vị trí của bit 1 bên phải nhất của m. Trong Fenwick-Tree, nút có số hiệu m sẽ là nút gốc của một cây con gồm 2k nút có số hiệu từ m- 2k+1 đến m.

BIT1 = a1 (BIT1 lưu tổng phần tử thứ nhất của mảng a)

BIT2 = a1 + a2 (BIT2 lưu tổng phần tử thứ 1 và thứ 2 của mảng a)

BIT3 = a3

BIT4 = a [1-4]

BIT5 = a5

BIT6 = a[5-6]

BIT7 = a7

BIT8 = a[1-8]

BIT9 = a9

BIT10 = a[9-10]

BIT11 = a11

BIT12 = a[9-12]

BIT13 = a13

BIT14 = a[13,14]

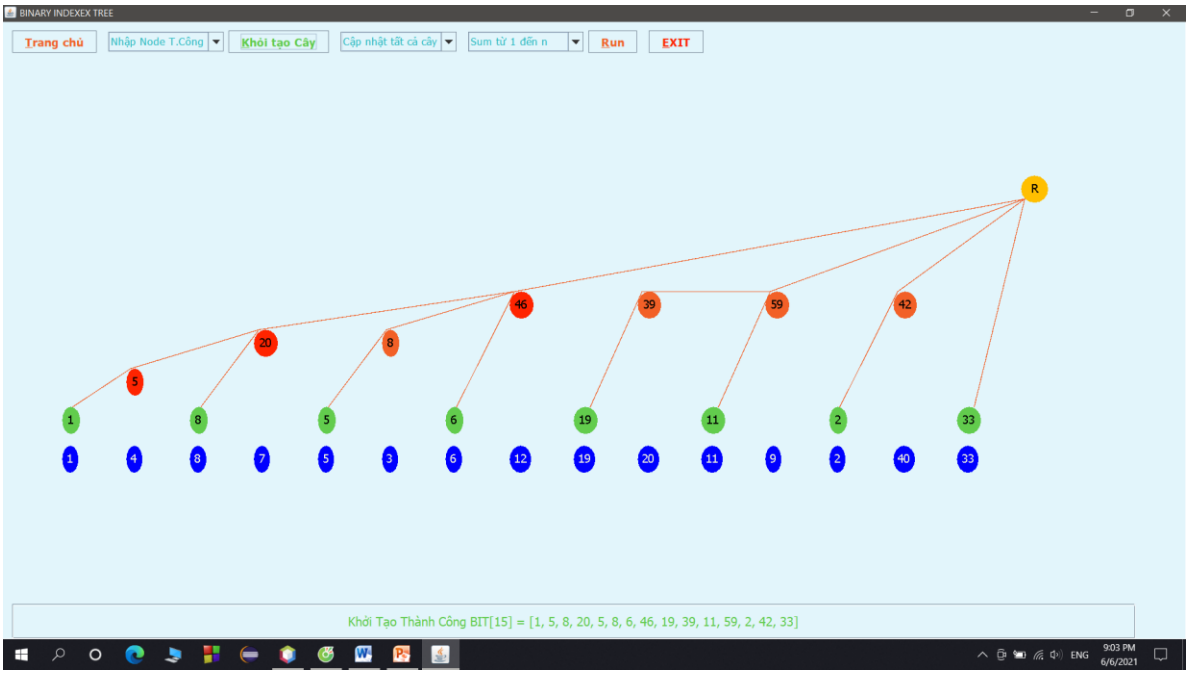
BIT15 = a15

Suy ra BIT[]

BIT[]	1	5	8	20	5	8	6	46	19	39	11	59	2	42	33

Code:

```
public static void create_tree() {
      for (int i = 1; i <= N; i++)
            BITree[i] = i;
         for (int i = 1; i <= N; i++){
            Sum_A[i] = a[i] + Sum_A[i - 1];
            BITree[i] =  Sum_A[i] - Sum_A[(i - (lowbit(i)) + 1) - 1];
         }
      }
```



4.2. Creat INPUT

4.2.1. Nhập node thủ công:

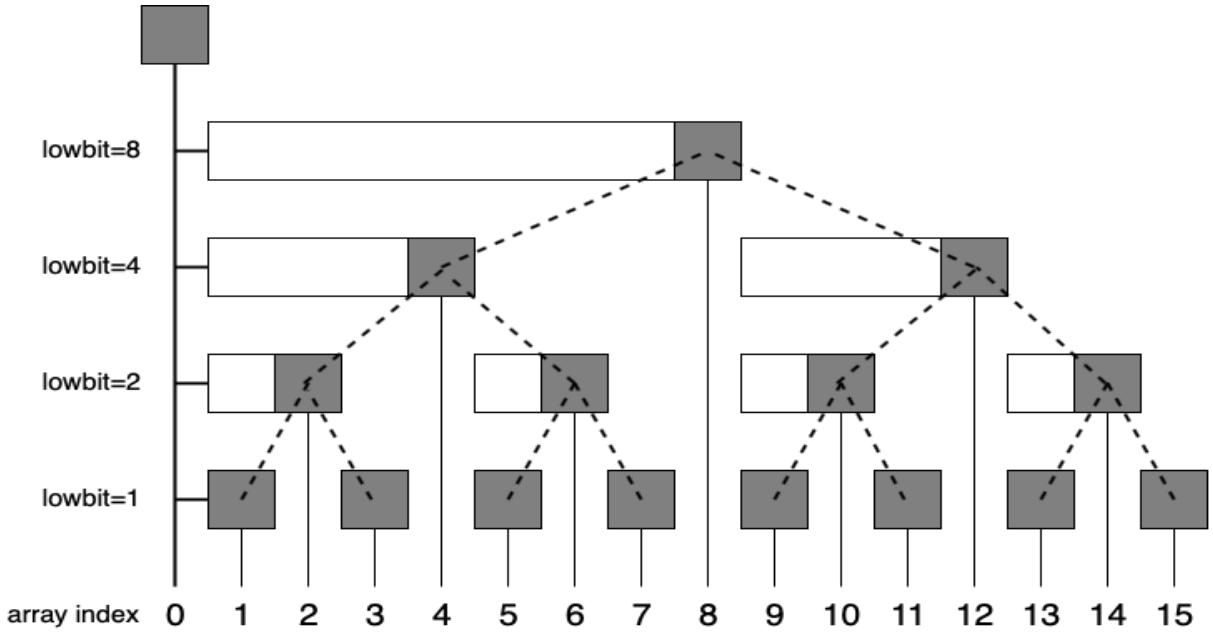
```
System.out.print("Create Node: ");
String s1 = sc.nextLine();
s1 = s1.trim();
String c[] = s1.split(" ");
N = c.length;
for (int i = 0, j = 1; i < N; i++, j++){
    a[j] = Integer.parseInt(c[i]);
}
```

4.2.2. Nhập N node:

```
System.out.print("Create N node: ");
N = sc.nextInt();
System.out.println("Create BITree " + N + " Node");
for (int i = 1; i <= N; i++)
     a[i] = i;
```

4.2.3. Random N node và Random giá trị node

```
N = rd.nextInt(25); //khoảng random có thể thay đổi
System.out.println("Create random N: " + N);
System.out.print("Create BITree " + N + " Node: ");
for (int i = 1; i <= N; i++)
     a[i] = rd.nextInt(50);
```



4.3 Update cây:

4.3.1. Update tại Node:

Giả sử, đề bài yêu cầu cập nhật node n + value (n, value nhập từ bàn phím)

B1: Tìm vị trí bit của n để update.

B2: Cộng giá trị value vào node cần update.

B3: Cập nhật lại mảng a[] và BIT[].

VD: update tại vị trí a[7] += 10 => update tại vị trí BIT[0111]

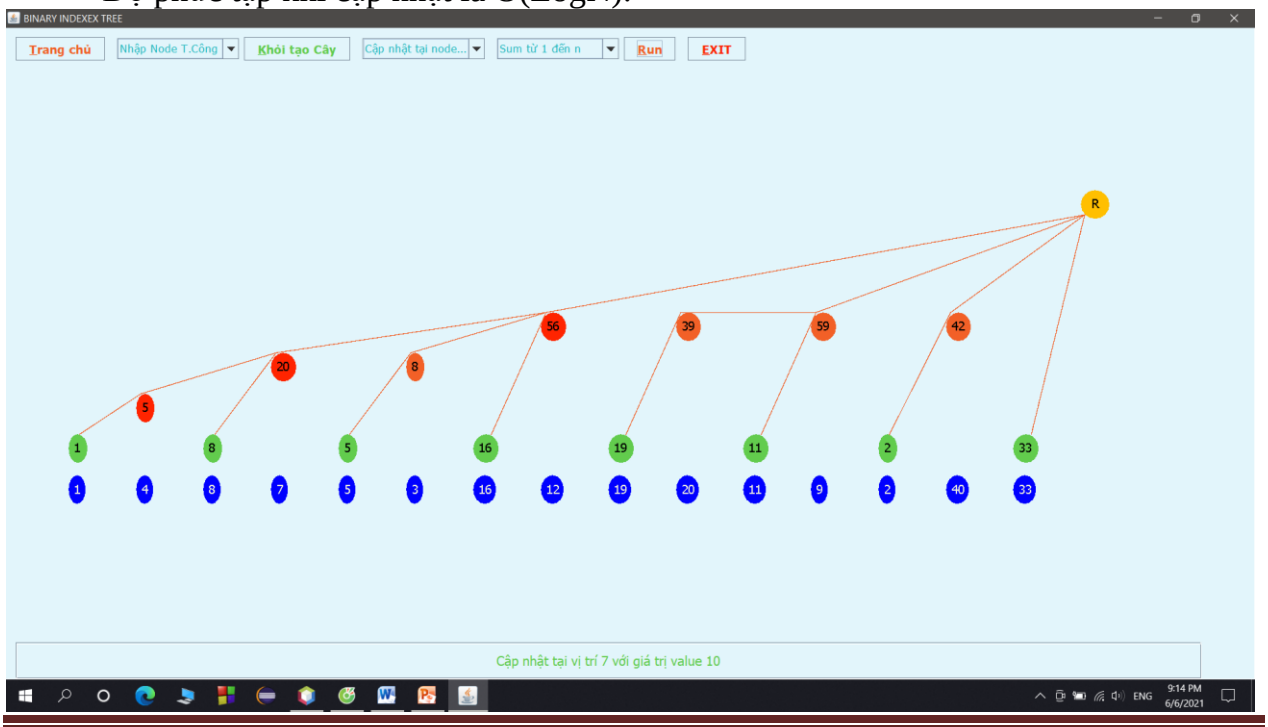
A[]	1	4	8	7	5	3	6	12	19	20	11	9	2	40	33
BIT[]	1	5	8	20	5	8	6	46	19	39	11	59	2	42	33

A[]	1	4	8	7	5	3	16	12	19	20	11	9	2	40	33
BIT[]	1	5	8	20	5	8	16	46	19	39	11	59	2	42	33

Code:

```
public static void update (int i, int val) {
     int index = i;
        while(index <= N){
            BITree[index] += val;
            index += index & (-index);
        }
        a[i] += val;
    }
```

Độ phức tạp khi cập nhật là O(LogN).



4.3.2. Update từ Node n đến Node m:

Giả sử, đề bài yêu cầu cập nhật node từ [n, m]+value (n, m, value nhập từ bàn phím)

B1: Duyệt qua tất cả các node trong khoảng [n, m] của mảng a[]

B2: Cộng lần lượt giá trị value vào node đã duyệt

B3: Cập nhật lại mảng a[] và BIT[]

VD: update từ a[8] đến a[13] += 5 => update từ BIT[1000] đến BIT[1011]

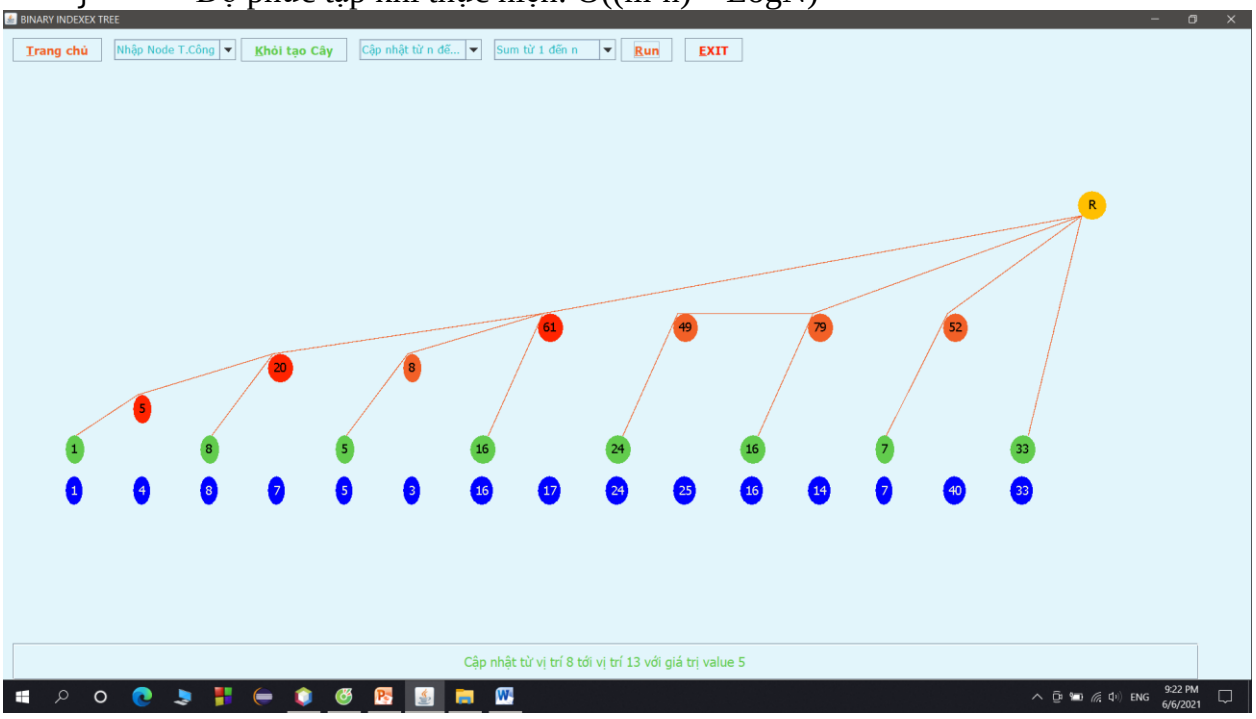
A[]	1	4	8	7	5	3	6	12	19	20	11	9	2	40	33
BIT[]	1	5	8	20	5	8	6	46	19	39	11	59	2	42	33

A[]	1	4	8	7	5	3	16	17	24	25	16	14	7	40	33
BIT[]	1	5	8	20	5	8	16	61	24	49	16	79	7	42	33

Code:

```
public static void update (int n, int m, int val) {
     for (int i = n; i <= m; i++){
           for (int j = i; j <= N; j += lowbit(j)){
               BITree[j] += val;
           }
       }
       for (int i = n; i <= m; i++){
           a[i] += val;
       }
   }
```

Độ phức tạp khi thực hiện: O((m-n) * LogN)



4.3.3. Update toàn bộ cây:

Giả sử, đề bài yêu cầu cập nhật tất cả node + value (value nhập từ bàn phím)

B1: Cộng lần lượt giá trị value vào tất cả các node

B2: Cập nhật lại mảng a[] và BIT[]

VD: update mảng += 7.

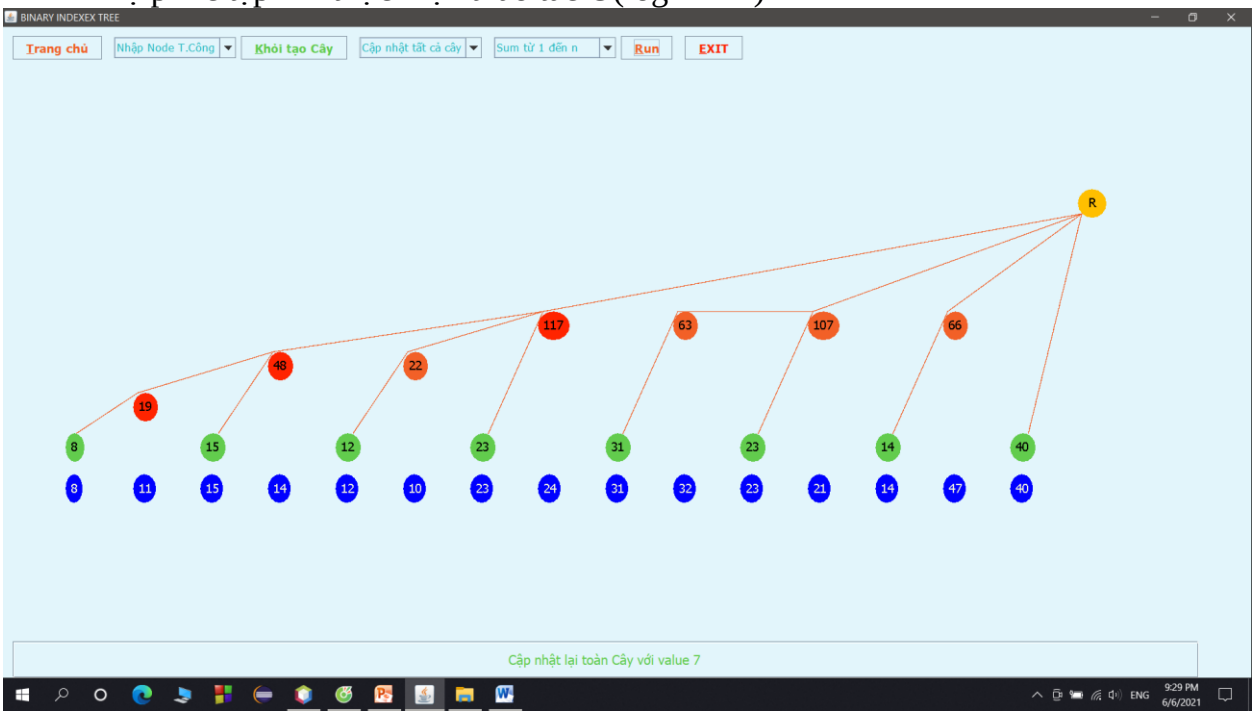
A[]	1	4	8	7	5	3	16	17	24	25	16	14	7	40	33
BIT[]	1	5	8	20	5	8	16	61	24	49	16	79	7	42	33

A[]	8	11	15	14	12	10	23	24	31	32	23	21	14	47	40
BIT[]	8	19	15	48	12	22	23	117	31	63	23	107	14	66	40

Code:

```
public static void update (int val) {
        for (int i = 1; i <= N; i++){
            for (int j = i; j <= N; j += lowbit(j)){
                BITree[j] += val;
            }
        }
        for (int i = 1; i <= N; i++){
            a[i] += val;
        }
}
```

Độ phức tạp khi thực hiện thao tác O(logN * N)



4.4. Sum

4.4.1. Sum từ 1 đến n:

Giả sử, đề bài yêu cầu tính tổng từ 1 đến n (n nhập từ bàn phím).

B1:Khởi tạo biến SUM = 0

B2: Cộng giá trị tại bit n.

B3: Chuyển bit 1 cuối cùng bên phải, gán vị trí đó = n rồi lặp lại B2.

B4: Trả về giá trị SUM.

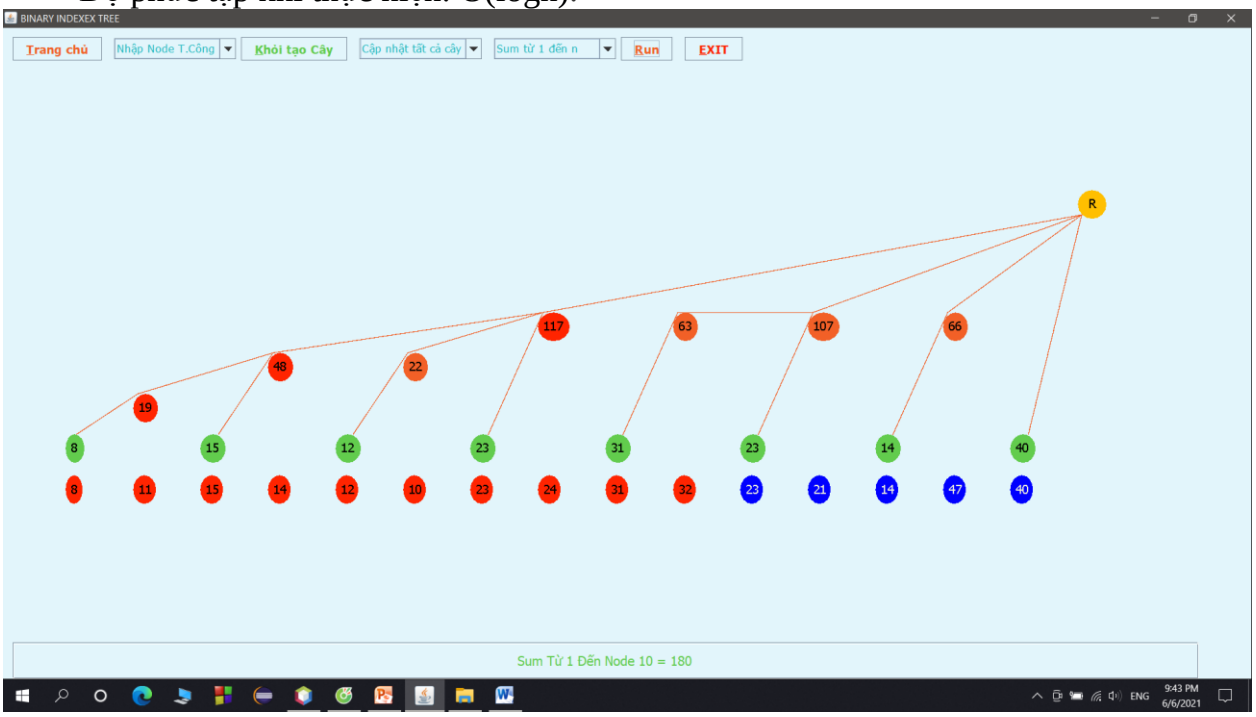
VD: Sum từ 1 đến 10. => a[10] = BIT[1010] +BIT[1000] = 180

A[]	8	11	15	14	12	10	23	24	31	32	23	21	14	47	40
BIT[]	8	19	15	48	12	22	23	117	31	63	23	107	14	66	40

Code:

```
public static int sum (int n) {
        int ans = 0;
        for (int i = n; i >= 1; i -= lowbit(i))
           ans += BITree[i];
        return ans;
    }
```

Độ phức tạp khi thực hiện: O(logn).



4.4.2. Sum từ n đến m:

Giả sử, đề bài yêu cầu tính tổng từ n đến m (n, m nhập từ bàn phím).

B1:Khởi tạo biến SUM = 0

B2: Cộng giá trị tại bit m.

B3: Chuyển bit 1 cuối cùng bên phải, gán vị trí đó = m rồi lặp lại B2.

B4: Trả về giá trị SUM.

B5:Sau khi B4 hoàn tất, trừ SUM cho giá trị SUM(n-1).

VD: Sum từ 4 đến 11

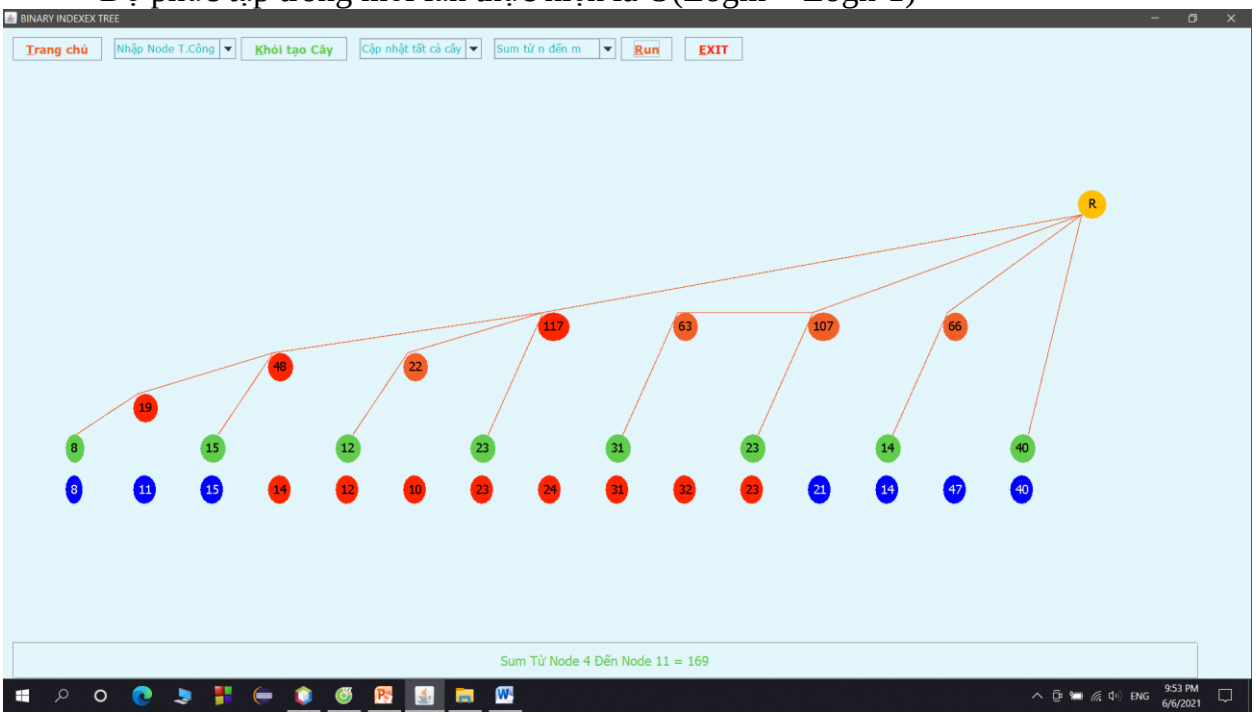
=> (BIT[1011] + BIT[1010]+BIT[1000]) – (BIT[0011]+BIT[0010]) = 169

A[]	8	11	15	14	12	10	23	24	31	32	23	21	14	47	40
BIT[]	8	19	15	48	12	22	23	117	31	63	23	107	14	66	40

Code:

```
public static int sum (int n, int m) {
          return sum (m) - sum (n-1);
}
```

Độ phức tạp trong mỗi lần thực hiện là O(Logm + Logn-1)



5. Bài toán áp dung và tài liệu tham khảo:

5.1. Tài liệu tham khảo:

Dưới đây là 1 số trang web tham khảo về cây BINARY INDEXED TREE mà Team cho rằng là dễ hiểu nhất.

https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/

https://viblo.asia/p/cau-truc-du-lieu-bit-binary-indexed-tree-fenwick-tree-Az45bWvNKxY

https://www.youtube.com/watch?v=v_wj_mOAlig&list=LL&index=15

http://community.topcoder.com

5.2. Source code Console:

https://paste.ubuntu.com/p/M6MJCyhFH9/

5.3. Bài toán áp dụng:

Ví dụ về cây BIT áp dụng trong bài toán **"Range Sum Query"**

Có n cái hộp được đánh số từ 1 đến n ($1 \le n \le 100.000$), ban đầu tất cả các hộp này đều rỗng. Có m ($1 \le m \le 100.000$) truy vấn, mỗi truy vấn có 1 trong 2 dạng sau:

- "+ i v": Thêm v viên bi vào hộp i ($1 \le i \le n$, $0 \le v \le 100.000$).
- "? i j": Tính tổng số lượng các viên bi nằm trong các hộp từ i đến j ($1 \le i \le j \le n$).

Dữ liệu: Dòng đầu tiên chứa 2 số nguyên n và m. Tiếp theo có m dòng, mỗi dòng chứa một phép một truy vấn như mô tả ở trên. Các số trên cùng một dòng ngăn cách nhau bởi một dấu cách.

Kết quả: Đưa ra lần lượt các câu trả lời cho mỗi truy vấn dạng thứ hai. Mỗi câu trả lời ghi trên một dòng.

Ví dụ:

input	Output
6 5 + 1 8 + 2 13 ? 1 3 + 4 5 ? 1 6	21 26

Code:

```
#include <bits/stdc++.h>
#define ll long long int;
using namespace std;

int n, m;
int tree[100001];

void update(int idx, int val) {
    while (idx <= n) {
        tree[idx] += val;
        idx += (idx & -idx);
    }
}

int read(int idx) {
    ll sum = 0;
    while (idx > 0) {
        sum += tree[idx];
          idx -= (idx & -idx);
    }
    return sum;
}

int main () {
        ios::sync_with_stdio(false);
cin.tie(0); cout.tie(0);
    cin >> n >> m;
    memset(tree, 0, sizeof(tree));
    for ( ; m > 0; m--) {
        char c;
        int i, j;
        cin >> c >> i >> j;
        if (c == '+')
            update(i, j);
        else
            cout << read(j) - read(i-1) << endl;
    }
    return 0;
}
```