

Disjoint Set Union

Disjoint Set Union

Người viết:

- Ngô Nhật Quang - HUS High School for Gifted Students

Reviewer:

- Trần Quang Lộc - ITMO University
- Hoàng Xuân Nhật - VNUHCM-University of Science
- Hồ Ngọc Vĩnh Phát - VNUHCM-University of Science

Giới thiệu

Disjoint Set Union, hay **DSU**, là một cấu trúc dữ liệu hữu dụng và thường xuyên được sử dụng trong các kì thi CP. DSU, đúng như tên gọi của nó, là một cấu trúc dữ liệu có thể quản lí một cách hiệu quả một tập hợp của các tập hợp.

Bài toán

Cho một đồ thị có n đỉnh, ban đầu không có cạnh nào. Chúng ta phải xử lí các truy vấn như sau:

- Thêm một cạnh giữa đỉnh x và đỉnh y trong đồ thị.
- In ra **YES** nếu như đỉnh x và đỉnh y nằm trong cùng một thành phần liên thông. In ra **NO** nếu ngược lại.

Một thành phần liên thông trong đồ thị là một đồ thị con trong đó giữa bất kì hai đỉnh nào đều có đường đi đến nhau, và không thể nhận thêm bất kì một đỉnh nào mà vẫn duy trì tính chất trên.

Cấu trúc dữ liệu Disjoint Set Union

Nếu ta coi mỗi đỉnh trong đồ thị là một phần tử và mỗi thành phần liên thông trong đồ thị là một tập hợp, truy vấn thứ nhất sẽ trở thành gộp hai tập hợp lần lượt chứa phần tử x và y thành một tập hợp mới và truy vấn thứ hai trở thành hỏi hai phần tử x và y có nằm trong cùng một tập hợp hay không.

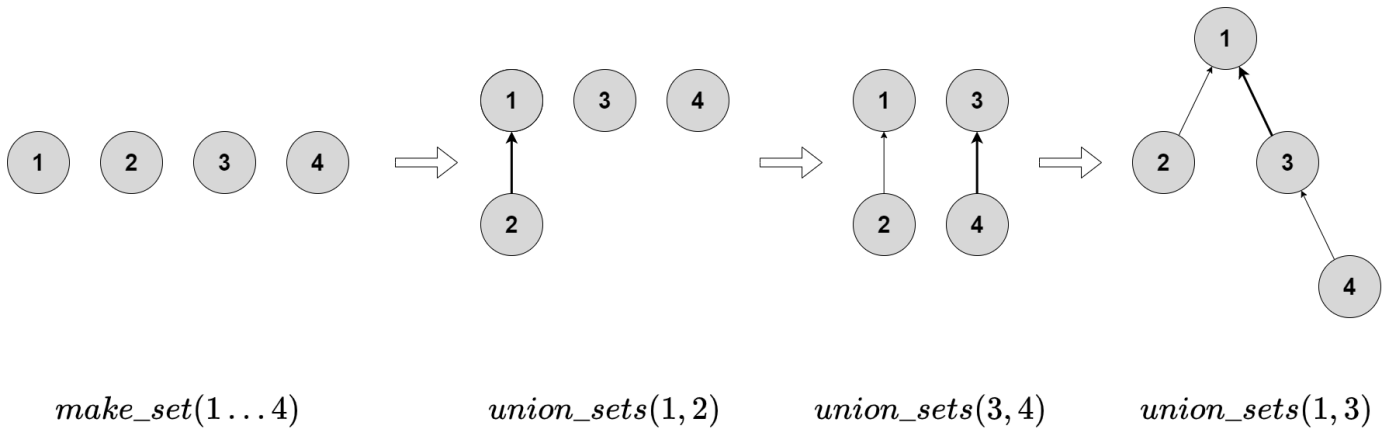
Để giải bài toán này, ta sẽ xây dựng một cấu trúc dữ liệu có ba thao tác như sau:

- `make_set(v)` - tạo ra một tập hợp mới chỉ chứa phần tử v .

- `union_sets(a, b)` - gộp tập hợp chứa phần tử `a` và tập hợp chứa phần tử `b` thành một.
- `find_set(v)` - cho biết **đại diện** của tập hợp có chứa phần tử `v`. Đại diện này sẽ là một phần tử của tập hợp đó và có thể thay đổi sau mỗi lần gọi thao tác `union_sets`. Ta có thể sử dụng đại diện đó để kiểm tra hai phần tử có nằm trong cùng một tập hợp hay không. `a` và `b` nằm trong cùng một tập hợp nếu như đại diện của hai tập chứa chúng là giống nhau và không nằm trong cùng một tập hợp nếu ngược lại.

Ta có thể xử lý các thao tác một cách hiệu quả này với các tập hợp được biểu diễn dưới dạng các cây, mỗi phần tử là một đỉnh và mỗi cây tương ứng với một tập hợp. Gốc của mỗi cây sẽ là đại diện của tập hợp đó.

Cấu trúc của cây được thể hiện qua ví dụ sau đây:



Ban đầu, mỗi phần tử thuộc một tập hợp riêng biệt, vậy mỗi đỉnh là một cây riêng biệt. Bước tiếp theo, ta gộp hai tập hợp chứa phần tử 1 và 2. Sau đó, ta gộp hai tập hợp chứa phần tử 3 và 4. Cuối cùng, ta gộp hai tập hợp chứa phần tử 1 và 3.

Với cách cài đặt này, ta sẽ lưu một mảng `parent` với `parent[v]` là cha của phần tử `v`.

Cài đặt "ngây thơ"

Để tạo một tập hợp mới gồm phần tử `v` (hay `make_set(v)`), ta chỉ cần tạo một cây có gốc là `v`, với `parent[v] = v`.

Để gộp hai tập hợp lần lượt chứa phần tử `a` và phần tử `b` (hay `union_sets(a, b)`), ta sẽ tìm gốc của cây có chứa phần tử `a` và gốc của cây có chứa phần tử `b`. Nếu hai giá trị này giống nhau, ta sẽ không làm gì do hai phần tử này đã nằm trong cùng một tập hợp. Còn nếu không, ta sẽ đặt gốc cây này là cha của gốc cây còn lại. Dễ thấy điều này sẽ gộp hai cây lại thành một.

Để tìm kí hiệu của một tập hợp có chứa phần tử `v` (hay `find_set(v)`), ta đơn giản nhảy lên các tổ tiên của đỉnh `v` cho đến khi ta đến gốc của cây. Thao tác này có thể dễ dàng được cài đặt bằng đệ quy.

```
void make_set(int v) {
    parent[v] = v; // Tạo ra cây mới có gốc là đỉnh v
}

int find_set(int v) {
```

```

7 |     if (v == parent[v]) return v; // Trả về đỉnh v nếu như đỉnh v là gốc của c;
8 |     return find_set(parent[v]); // Đệ quy lên cha của đỉnh v
9 | }
10 |
11 | void union_sets(int a, int b) {
12 |     a = find_set(a); // Tìm gốc của cây có chứa đỉnh a
13 |     b = find_set(b); // Tìm gốc của cây có chứa đỉnh b
14 |     if (a != b) parent[b] = a; // Gộp hai cây nếu như hai phần tử ở hai cây kh;
    }

```

Như đã nói, đây là cách cài đặt ngây thơ, ta có thể dễ dàng tạo ra một ví dụ sao cho khi sử dụng cách cài đặt này, cây sẽ trở thành một đoạn thẳng gồm n phần tử. Trong trường hợp này, độ phức tạp của thao tác `find_set` sẽ là $\mathcal{O}(n)$.

Tối ưu 1 - Gộp theo kích cỡ / độ cao

Phương pháp tối ưu này sẽ thay đổi thao tác `union_sets`. Chính xác hơn, ta sẽ thay đổi cách xét trong hai cây đang gộp, gốc của cây nào sẽ là cha của gốc của cây còn lại.

Có khá nhiều cách để xét điều này, nhưng hai cách được sử dụng nhiều nhất chính là gộp theo kích cỡ và gộp theo độ cao của cây.

Giả dụ mỗi cây có một giá trị. Ở cách thứ nhất, giá trị đó là kích cỡ của cây, và ở cách thứ hai, giá trị đó là độ cao của cây. Ở cả hai cách này, ta sẽ luôn đặt gốc của cây có giá trị cao hơn là cha của gốc của cây có giá trị thấp hơn.

Thao tác `union_sets` được tối ưu gộp theo kích cỡ:

```

1 | void make_set(int v) {
2 |     parent[v] = v;
3 |     sz[v] = 1; // Ban đầu tập hợp chứa v có kích cỡ là 1
4 | }
5 |
6 | void union_sets(int a, int b) {
7 |     a = find_set(a);
8 |     b = find_set(b);
9 |     if (a != b) {
10 |         if (sz[a] < sz[b]) swap(a, b); // Đặt biến a là gốc của cây có kích cỡ
11 |         parent[b] = a;
12 |         sz[a] += sz[b]; // Cập nhật kích cỡ của cây mới gộp lại
13 |     }
14 | }

```

Thao tác `union_sets` được tối ưu gộp theo độ cao:

```

1  void make_set(int v) {
2      parent[v] = v;
3      rank[v] = 0; // Gốc của cây có độ cao là 0
4  }
5
6  void union_sets(int a, int b) {
7      a = find_set(a);
8      b = find_set(b);
9      if (a != b) {
10         if (rank[a] < rank[b]) swap(a, b); // Đặt biến a là gốc của cây có độ cao lớn hơn
11         parent[b] = a;
12         if (rank[a] == rank[b]) rank[a]++; // Nếu như hai cây có cùng một độ cao thì tăng độ cao của gốc a lên 1
13     }
14 }

```

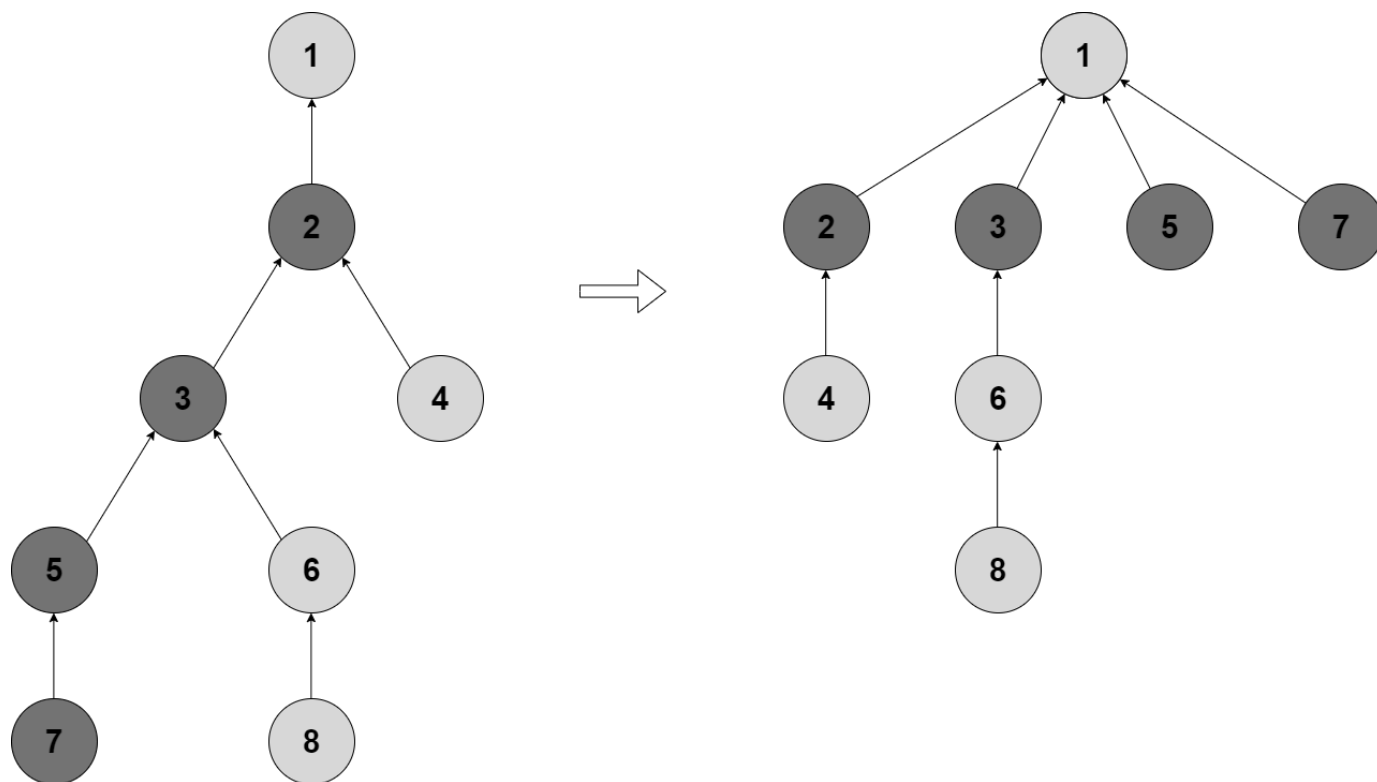
Chỉ cần sử dụng phương pháp tối ưu này, độ phức tạp của thao tác `find_set` sẽ trở thành $\mathcal{O}(\log n)$. Tuy nhiên, ta vẫn còn có thể làm tốt hơn thế khi kết hợp với phương pháp tối ưu thứ hai.

Tối ưu 2 - Nén đường đi

Phương pháp tối ưu này nhằm tăng tốc thao tác `find_set`.

Giả dụ ta gọi `find_set(v)` với một đỉnh `v` bất kì, chúng ta tìm được `p` là gốc của cây, đồng thời cũng là giá trị của mọi hàm `find_set(u)` với `u` là một đỉnh nằm trên đường đi từ `u` đến `p`. Cách tối ưu ở đây chính là làm cho đường đi đến gốc của các đỉnh `u` ngắn đi bằng cách gán trực tiếp cha của các đỉnh `u` này thành `p`.

Có thể thấy sau khi thực hiện một thao tác như vậy, cấu trúc cả cây có thể thay đổi. Ta có thể thấy điều này trong ví dụ sau đây:



Bên trái là cây ban đầu và bên phải là cây bị nén sau khi ta sử dụng thao tác `find_set(7)`, nén đường đi tới gốc của các đỉnh 7, 5, 3, 2.

Thao tác `find_set` mới này được cài đặt như sau:

```

1 | int find_set(int v) {
2 |     if (v == parent[v]) return v; // Trả về đỉnh v nếu như đỉnh v là gốc của c
3 |     int p = find_set(parent[v]); // Đệ quy lên cha của đỉnh v
4 |     parent[v] = p; // Nén đoạn từ v lên gốc của cây
5 |     return p;
6 | }

```

Một cách cài đặt khác của thao tác `find_set` mà thường được sử dụng nhiều trong CP do tính ngắn gọn của nó:

```

1 | int find_set(int v) {
2 |     return v == parent[v] ? v : parent[v] = find_set(parent[v]);
3 | }

```

Độ phức tạp thời gian

Khi kết hợp cả hai phương pháp tối ưu bên trên lại, ta sẽ đạt được độ phức tạp trung bình cho thao tác `find_set` là $\mathcal{O}(\alpha(n))$ với $\alpha(n)$ là hàm Ackermann nghịch đảo. Tuy nhiên hàm này **tăng rất chậm** (với $\alpha(n) \leq 3$ với $n \leq 61$ và $\alpha(n) \leq 4$ với xấp xỉ $n < 10^{600}$).

Độ phức tạp trung bình, đúng như tên gọi của nó, là tổng độ phức tạp của các thao tác, chia cho số lượng thao tác. Điều này có nghĩa ở một số thao tác, độ phức tạp có thể lên tới $\mathcal{O}(\log n)$ nhưng ta thực hiện m thao tác như vậy, độ phức tạp sẽ trở thành $\mathcal{O}(m \cdot \alpha(n))$ ($\approx \mathcal{O}(m)$ với m đủ lớn).

Một điều đáng lưu ý là nếu như chúng ta chỉ sử dụng phương pháp tối ưu 2, độ phức tạp trung bình của thao tác `find_set` cũng chỉ là $\mathcal{O}(\log n)$.

Chứng minh độ phức tạp thời gian

Tối ưu gộp set theo kích cỡ: Gọi a là độ lớn của cây con có gốc là đỉnh v , b là độ lớn của cây con có gốc là p (cha của đỉnh v). Dễ thấy rằng $b \geq 2 \times a$ do số lượng đỉnh trong cây con gốc p mà không thuộc cây con gốc v lớn hơn hoặc bằng a . Do vậy độ sâu tối đa của cây sẽ là $\log_2 n$.

Tối ưu gộp set theo độ cao: Ta sẽ chứng minh một cây có độ cao là k có ít nhất 2^k đỉnh. Có thể thấy rằng số cây có độ cao là 0 có chính xác 1 đỉnh. Một cây chỉ có thể có độ cao là k nếu như trước đó độ cao của nó là $k - 1$ và nó được gộp với một cây khác có độ cao là $k - 1$. Vì vậy, số đỉnh trong cây có độ cao là k sẽ lớn hơn hoặc bằng hai lần số đỉnh trong cây có độ cao là $k - 1$. Do đó độ cao lớn nhất có thể của cây sẽ là $\log_2 n$.

Kết hợp hai phương pháp tối ưu: Phần chứng minh này khá dài dòng và khó hiểu, bạn đọc có thể tìm hiểu tại [đây](#) hoặc [đây](#).

Một cách cài đặt khác

Ở trong một số tài liệu như Giải thuật và lập trình (thầy Lê Minh Hoàng) hay thư viện Atcoder, thay vì cài đặt cấu trúc dữ liệu DSU bằng hai mảng `parent` và `sz`, chỉ một mảng `lab` được sử dụng.

Nếu như `lab[v]` âm thì u là gốc của một cây và `-lab[v]` là số lượng đỉnh của cây đó. Còn nếu `lab[v]` dương thì `lab[v]` là cha của đỉnh u .

```

1  void make_set(int v) {
2      lab[u] = -1;
3  }
4
5  int find_set(int v) {
6      return lab[v] < 0 ? v : lab[v] = find_set(lab[v]);
7  }
8
9  void union_sets(int a, int b) {
10     a = find_set(a);
11     b = find_set(b);
12
13     if (a != b) {
14         if (lab[a] > lab[b]) swap(a, b);
15         lab[a] += lab[b];
16         lab[b] = a;
17     }
18 }

```

Một số ứng dụng của DSU

Lưu thêm thông tin khác cho mỗi tập hợp

Ngoài việc lưu các thông tin về cấu trúc cây, ta có thể lưu các hàm có tính chất giao hoán và kết hợp của từng tập hợp. Ví dụ, ta có thể lưu tổng các phần tử/ giá trị phần tử bé nhất của từng tập hợp. Lúc này, các thao tác của dsu sẽ được cài đặt như sau:

```

1  void make_set(int v) {
2      parent[v] = v;
3      sz[v] = 1;
4      mn[v] = value[v];
5      sum[v] = value[v];
6      // value[v] là giá trị của phần tử thứ v
7  }
8
9  int find_set(int v) {
10     return v == parent[v] ? v : parent[v] = find_set(parent[v]);
11 }
12
13 void union_sets(int a, int b) {
14     a = find_set(a);
15     b = find_set(b);
16     if (a != b) {
17         if (sz[a] < sz[b]) swap(a, b);
18         parent[b] = a;
19         sz[a] += sz[b];
20         sum[a] += sum[b];
21         mn[a] = min(mn[a], mn[b]);
22     }
23 }

```

Có thể thấy rằng, tương tự như thông tin về độ lớn của cây (**sz**) hay độ cao của cây (**rank**), ta sẽ lưu các hàm này tại gốc của từng cây.

```

1  int find_sum(int v) { // Trả về tổng của các phần tử trong tập hợp chứa v
2      v = find_set(v);
3      return sum[v];
4  }
5
6  int find_min(int v) { // Trả về giá trị bé nhất của các phần tử trong tập hợp
7      v = find_set(v);
8      return mn[v];
9  }

```

Bài toán xếp hàng

Bài toán

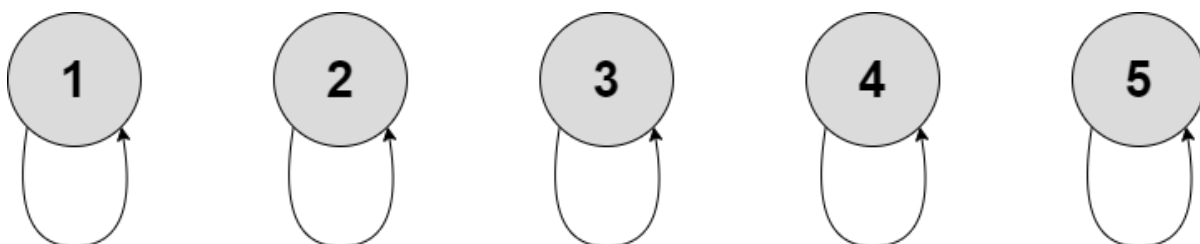
Cho n người đang xếp hàng ở các vị trí từ 1 đến n . Viết chương trình xử lý các truy vấn:

- ▶ Người đứng ở vị trí thứ i rời khỏi hàng.
- ▶ Tìm người gần nhất về bên phải vị trí p mà chưa rời khỏi hàng.

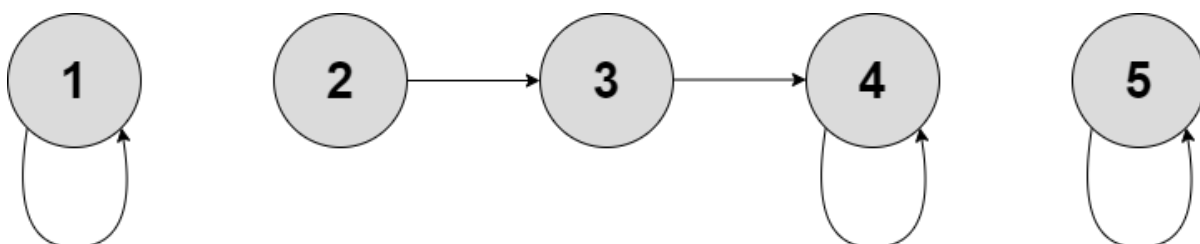
Lời giải

Với mỗi vị trí, ta sẽ có một con trỏ. Nếu người đứng ở vị trí này vẫn đang đứng trong hàng, con trỏ trỏ vào vị trí đó, nếu không thì con trỏ này sẽ trỏ vào vị trí ngay bên phải.

Xét ví dụ sau với $n = 5$, ban đầu ta có:



Giả dụ người đứng ở vị trí 2 và 3 rời khỏi hàng:



Dễ thấy để tìm người gần nhất bên phải mà chưa rời khỏi hàng, ta đi dần dần sang phải cho đến khi gặp một vị trí có con trỏ trỏ đến chính nó.

Chúng ta có thể sử dụng cấu trúc dữ liệu DSU để lưu trữ các thông tin trên và sử dụng phương pháp tối ưu nén đoạn để đạt được độ phức tạp trung bình $\mathcal{O}(\log n)$ với mỗi truy vấn.

Để ý kĩ hơn, ta thấy vị trí ta cần tìm chính là vị trí có thứ tự lớn nhất trong tập hợp. Ta có thể lưu phần tử lớn nhất trong một tập hợp như đã nói ở phần trên, qua đó đạt được độ phức tạp trung bình $\mathcal{O}(\alpha(n))$ với mỗi truy vấn.

Code mẫu

```

1  void make_set(int v) {
2      parent[v] = v;
3      sz[v] = 1;
4      mx[v] = v;
5  }
6
7  int find_set(int v) {
8      if (parent[v] == v) return v;
9      return parent[v] = find_set(parent[v]);
10 }
```



```

9      return v == parent[v] ? v : parent[v] = find_set(parent[v]);
10   }
11
12   void union_sets(int a, int b) {
13       a = find_set(a);
14       b = find_set(b);
15       if (a != b) {
16           if (sz[a] < sz[b]) swap(a, b);
17           parent[b] = a;
18           sz[a] += sz[b];
19           mx[a] = max(mx[a], mx[b]);
20       }
21   }
22
23   void leave(int v) { // Người thứ v rời khỏi hàng
24       union_sets(v, v + 1);
25   }
26
27   int find_next(int p) { // Trả về thứ tự của người gần nhất về bên phải
28                           // vị trí p mà chưa rời khỏi hàng
29       p = find_set(p);
30       return mx[p];
31   }

```

Tối ưu thuật toán tìm cây khung nhỏ nhất trong đồ thị

Sử dụng DSU, ta có thể tối ưu độ phức tạp của thuật toán tìm cây khung nhỏ nhất của đồ thị từ $\mathcal{O}(m \log n + n^2)$ xuống $\mathcal{O}(m \log n)$.

Bạn đọc có thể tìm hiểu kĩ hơn ở [blog](#) tìm cây khung nhỏ nhất trong đồ thị.

Đảo ngược truy vấn

Do tính chất một chiều của cấu trúc dữ liệu DSU (chỉ thêm chứ không xóa được), ở một số bài ta phải đảo ngược thứ tự của các truy vấn trong bài để giải.

Bài toán

[Codeforces 722C - Destroying Array](#) [🔗](#)

Cho mảng gồm n số nguyên không âm a_1, a_2, \dots, a_n và một hoán vị các số từ 1 đến n .

Các phần tử sẽ lần lượt bị phá hủy theo thứ tự hoán vị trên. Sau mỗi lần một phần tử bị phá hủy, hãy in ra dãy con liên tiếp có tổng lớn nhất mà không có phần tử nào đã bị phá hủy. Tổng của một đoạn con rỗng là 0.

Giới hạn: $1 \leq n \leq 10^5, 0 \leq a_i \leq 10^9$.

Lời giải

Do các phần tử là các số nguyên không âm, ta có thể thấy rằng nếu sau khi một số phần tử bị phá hủy, dãy bị chia thành k đoạn con liên tiếp thì đáp án sẽ là một trong k đoạn con này.

Đảo ngược thứ tự của các truy vấn, ta có thể thấy bài toán trở nên dễ dàng rất nhiều: Hồi sinh một số bị phá hủy trở về ban đầu và in ra đoạn con có tổng lớn nhất. Đến đây ta nghĩ tới cấu trúc dữ liệu DSU để xử lý các đoạn con liên tiếp.

Khi một số được hồi sinh, ta sẽ kiểm tra bên trái số đó, nếu có số nào đã được hồi sinh từ trước thì ta sẽ thêm cạnh giữa số đó và số bên trái số đó. Tương tự với số bên phải. Dễ thấy rằng mọi lúc các thành phần liên thông trong DSU sẽ thể hiện cho một đoạn con liên tiếp. Việc lưu trữ tổng của một thành phần liên thông đã được nhắc đến ở phần trước.

Code mẫu

```

1  #include <bits/stdc++.h>
2
3  #define int long long
4
5  using namespace std;
6
7  const int N = 1e5 + 5;
8  int n, ans;
9  int a[N], p[N], res[N];
10 bool flag[N];
11
12 struct DSU{
13     vector<int> parent, sz, sum;
14
15     DSU(int n) : parent(n), sz(n), sum(n) {};
16
17     void make_set(int v) {
18         parent[v] = v;
19         sz[v] = 1;
20         sum[v] = a[v];
21     }
22
23     int find_set(int v) {
24         return v == parent[v] ? v : parent[v] = find_set(parent[v]);
25     }
26
27     void join_sets(int a, int b) {
28         a = find_set(a);
29         b = find_set(b);
30         if (a != b) {
31             if (sz[a] < sz[b]) swap(a, b);
32             parent[b] = a;
33             sz[a] += sz[b];
34             sum[a] += sum[b];
35         }
36     }
37 };
38
39 signed main() {
40

```

```

40
41     ios_base::sync_with_stdio(false); cin.tie(NULL);
42
43     cin >> n;
44
45     for (int i = 1; i <= n; i++) cin >> a[i];
46     for (int i = 1; i <= n; i++) cin >> p[i];
47
48     DSU g(n + 5);
49     for (int i = 1; i <= n; i++) g.make_set(i);
50     for (int i = n; i >= 1; i--) {
51         flag[p[i]] = true;
52
53         if (p[i] > 1 && flag[p[i] - 1]) g.join_sets(p[i], p[i] - 1);
54         if (p[i] < n && flag[p[i] + 1]) g.join_sets(p[i], p[i] + 1);
55
56         ans = max(ans, g.sum[g.find_set(p[i])]);
57         res[i - 1] = ans;
58     }
59
60     for (int i = 1; i <= n; i++) cout << res[i] << "\n";
61
62 }

```

Kiểm tra tính chất hai phía của đồ thị online

Bài toán

Cho một đồ thị có n đỉnh, ban đầu không có cạnh nào. Xử lý các truy vấn thêm cạnh vào đồ thị. Hỏi sau truy vấn nào thì đồ thị không còn là đồ thị hai phía?

Lời giải

Dựa vào tính chất của đồ thị hai phía, dễ thấy rằng với mọi cặp đỉnh thuộc cùng một phía sẽ có đường đi bất kì giữa chúng có độ dài chẵn. Nói cách khác, nếu ta chọn một đỉnh l trong một thành phần liên thông, hai đỉnh a và b sẽ nằm cùng một phía nếu như khoảng cách của hai đỉnh này tới đỉnh l có cùng tính chẵn lẻ.

Chúng ta hoàn toàn có thể sử dụng cấu trúc dữ liệu DSU để lưu trữ thông tin này bằng cách lưu tính chẵn lẻ của đường đi từ mọi đỉnh tới gốc của cây. Lúc này, hàm `find_set` của ta sẽ trả về một cặp **{gốc của cây, tính chẵn lẻ của độ dài đường đi đến gốc của cây}** và được cài đặt như sau:

```

1 pair<int, int> find_set(int v) {
2     if (v == parent[v]) return {v, 0};
3     pair<int, int> val = get(parent[v]);
4     parent[v] = val.first;
5     dist[v] = (dist[v] + val.second) % 2;
6     // độ dài từ đỉnh đến cha mới
7     // = độ dài đến đỉnh cha cũ
8     // + độ dài từ cha cũ tới cha mới (gốc của cây)
9
10 }

```

```

    return {p[a], dist[a]};
}

```

Hàm `union_sets`, tương tự, cũng cần phải được thay đổi và được cài đặt như sau:

```

1 void union_sets(int a, int b) {
2     pair<int, int> valA = find_set(a),
3       valB = find_set(b);
4     a = valA.first; b = valB.first;
5
6     if (a == b) {
7         if (valA.second != valB.second) {
8             // Đồ thị không còn là đồ thị hai phía do có
9             // cạnh nối giữa hai đỉnh thuộc hai phía khác nhau.
10        }
11    }
12
13    else {
14        if (sz[a] < sz[b]) swap(a, b);
15        parent[b] = a;
16        sz[a] += sz[b];
17        dist[b] = (valA.second + valB.second + 1) % 2;
18        // Độ dài từ đỉnh b tới gốc cây
19        // = Độ dài từ đỉnh a tới gốc cây ban đầu chứa a
20        // + Độ dài từ đỉnh b tới gốc cây ban đầu chứa b
21        // + 1 (Khoảng cách giữa hai đỉnh a và b)
22    }
23 }

```

Một số kĩ thuật sử dụng tính chất của DSU

Ngoài ra tính chất của DSU còn được sử dụng trong một số kĩ thuật khá phổ biến.

Kĩ thuật gộp set (Small-to-Large Merging)

Giả sử ta cần lưu trực tiếp các phần tử của một tập hợp bằng một cấu trúc dữ liệu như set/map, thì liệu có cách nào đủ hiệu quả để thực hiện thao tác `union_sets` hay không? Câu trả lời là có và kĩ thuật này được gọi là gộp set (small-to-large merging).

Bài toán

VNOJ - colquery [🔗](#)

Cho một đồ thị vô hướng có n đỉnh, đỉnh thứ i có màu a_i . Ban đầu đồ thị chưa có cạnh nào.

Cho q truy vấn, mỗi truy vấn thuộc một trong hai dạng sau:

- 1 u v : Thêm một cạnh nối giữa u và v .

- $2u$ c : Tính số đỉnh có màu c trong thành phần liên thông chứa u .

Lời giải

Chúng ta vẫn sẽ sử dụng cấu trúc dữ liệu DSU trong bài này, và lưu thêm một map chứa số lượng từng màu tại gốc của từng cây.

Trong thao tác `union_sets`, ta sẽ chuyển lần lượt các phần tử trong map của tập hợp bé hơn vào map của tập hợp lớn hơn. Thoạt nhìn ban đầu, việc làm này có độ phức tạp tổng là $\mathcal{O}(n^2)$, nhưng thực chất nó chỉ là $\mathcal{O}(n \log n)$. Ta sẽ chứng minh tại sao.

Gọi số phần tử nằm trong hai dãy số lớn hơn và bé hơn lần lượt là a và b . Dễ thấy được rằng $a + b \geq 2 \cdot b$, nên mỗi lần một phần tử bị di chuyển, nó sẽ bị di chuyển tới một dãy số có kích thước lớn hơn ít nhất hai lần kích thước dãy số ban đầu nó nằm trong. Vì vậy mà ta thấy rằng một phần tử chỉ bị di chuyển tối đa $\log_2 n$ lần, qua đó mà đạt được độ phức tạp $\mathcal{O}(n \log n)$.

Code mẫu

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  const int N = 1e5 + 5;
6  int n, q;
7  int a[N];
8
9  struct DSU{
10     vector<map<int, int>> color;
11     vector<int> parent, sz;
12
13     DSU(int n) : color(n), parent(n), sz(n) {};
14
15     void make_set(int v) {
16         color[v][a[v]] = 1;
17         parent[v] = v;
18         sz[v] = 1;
19     }
20
21     int find_set(int v) {
22         if (v == parent[v]) return v;
23         int p = find_set(parent[v]);
24         parent[v] = p;
25         return p;
26     }
27
28     void union_sets(int a, int b) {
29         a = find_set(a);
30         b = find_set(b);
31         if (a != b) {
32             if (sz[a] < sz[b]) swap(a, b);
33             parent[b] = a;
34         }
35     }
36 }
```

```

35         sz[a] += sz[b];
36
37         for (auto p : color[b]) color[a][p.first] += p.second;
38         color[b].clear();
39     }
40 }
41
42 int query(int v, int c) {
43     v = find_set(v);
44     return color[v].find(c) != color[v].end() ? color[v][c] : 0;
45 }
46 };
47
48 signed main() {
49
50     ios_base::sync_with_stdio(false); cin.tie(NULL);
51
52     cin >> n >> q;
53     for (int i = 1; i <= n; i++) cin >> a[i];
54
55     DSU g(n + 5);
56     for (int i = 1; i <= n; i++) g.make_set(i);
57     while (q--) {
58         int op, x, y;
59         cin >> op >> x >> y;
60         if (op == 1) g.union_sets(x, y);
61         else cout << g.query(x, y) << "\n";
62     }
63 }

```

Độ phức tạp thuật toán: $\mathcal{O}(n \log^2 n)$, có thêm một log do ta phải lưu giữ thông tin bằng cấu trúc dữ liệu map.

Kĩ thuật DSU trên cây (Sack)

Đây là một thuật toán sử dụng ý tưởng gộp set ở phần trước để giải quyết một số bài toán truy vấn trên cây một cách hiệu quả.

Bài toán

Cho một cây có n đỉnh với gốc là đỉnh 1, đỉnh thứ i được tô màu c_i . Cho q truy vấn có dạng $v\ c$, với mỗi truy vấn in ra số lượng đỉnh có màu c trong cây con gốc v .

Lời giải

Sử dụng ý tưởng gộp set ở phần trước, ta có thể dễ dàng đạt được độ phức tạp $\mathcal{O}(n \log^2 n + q \log n)$. Tuy nhiên, ta còn có thể làm tốt hơn với kĩ thuật DSU trên cây

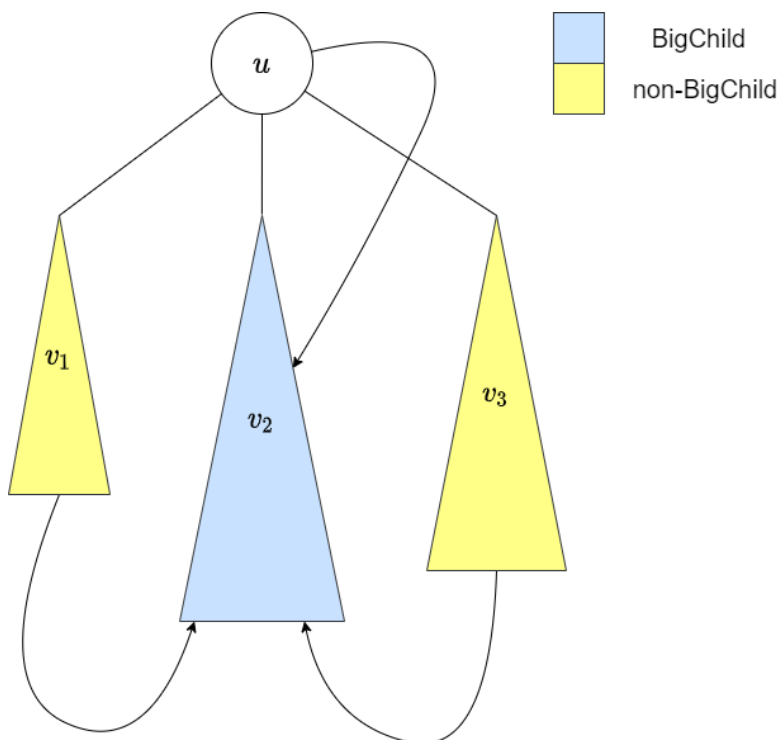
Bằng cách thay đổi cách dfs, ta có thể loại bỏ một log của cấu trúc dữ liệu map trong độ phức tạp, qua đó mà đạt được độ phức tạp $\mathcal{O}(n \log n + q)$. Ta sẽ tham khảo đoạn code dưới đây để hiểu hơn kĩ thuật này:

```

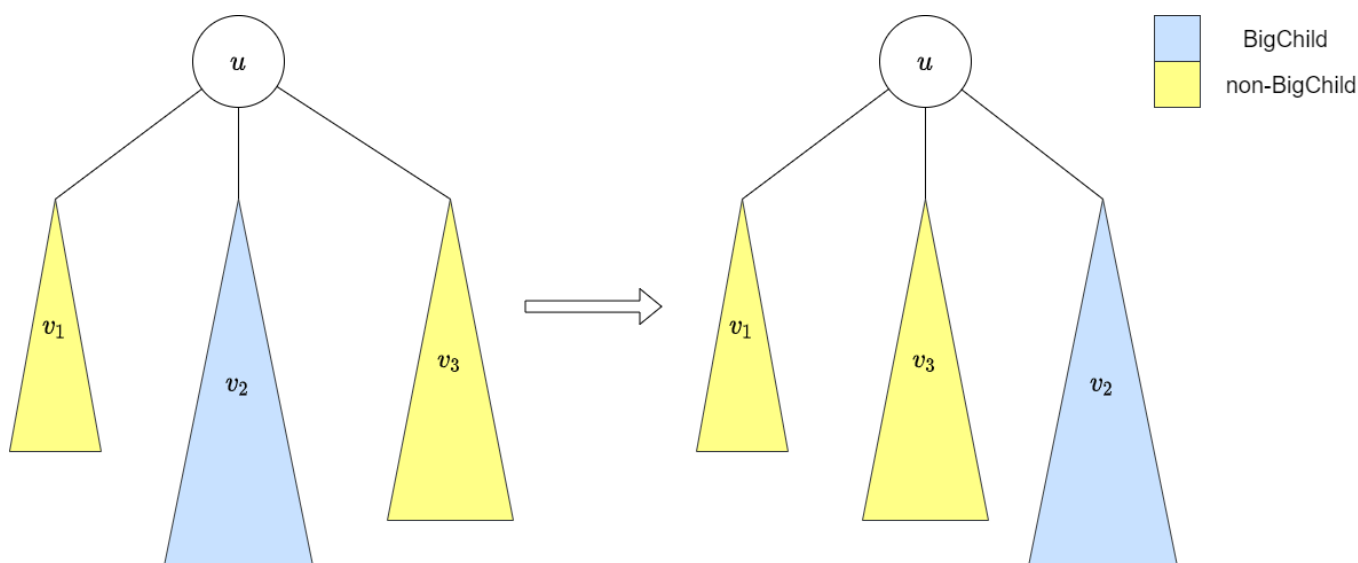
1  int sz[N];
2  int cnt[N];
3
4  void pre_dfs(int u, int p) { // Một hàm dfs chạy trước
5                               // để tính được độ lớn của từng cây con
6      sz[u] = 1;
7      for (auto v : g[u]) if (v != p) {
8          pre_dfs(v, u);
9          sz[u] += sz[v];
10     }
11 }
12
13 void update(int u, int p, int delta) {
14     cnt[color[u]] += delta;
15     for (auto v : g[u]) if (v != p) update(v, u, delta);
16 }
17
18 void dfs(int u, int p) {
19     int bigChild = -1;
20     for (auto v : g[u]) if (v != p) {
21         if (bigChild == -1 || sz[v] > sz[bigChild]) bigChild = v;
22     } // Tìm cây con lớn nhất trong
23         // các con trực tiếp của đỉnh u
24
25     for (auto v : g[u]) if (v != p && v != bigChild) {
26         dfs(v, u);
27         update(v, u, -1);
28     }
29     if (bigChild != -1) dfs(bigChild, u);
30
31     for (auto v : g[u]) if (v != p && v != bigChild) update(v, u, 1);
32     cnt[color[u]]++;
33
34     // Trả lời các truy vấn tại đỉnh u, với cnt[c]
35     // là số lượng đỉnh có màu c trong cây con gốc u
36 }

```

Với cây con gốc u đang xét, ta sẽ dfs xuống giải bài toán với đỉnh v là con trực tiếp của đỉnh u . Nếu giải như bài toán colquery, ở mỗi đỉnh ta sẽ lưu một cấu trúc dữ liệu map để lưu số lượng từng màu trong cây con đó. Sau đó ta sẽ gộp chúng lại để có được map chứa số lượng từng màu trong cây con gốc u (Gộp các map của cây con không phải cây con lớn nhất vào map của cây con lớn nhất).



Tuy nhiên, sự tối ưu của kỹ thuật này chính là ta có thể đảo thứ tự dfs và trả lời các truy vấn offline.



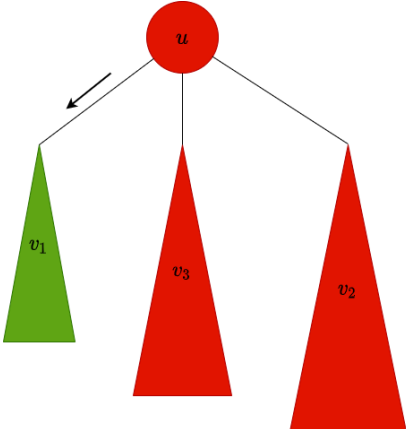
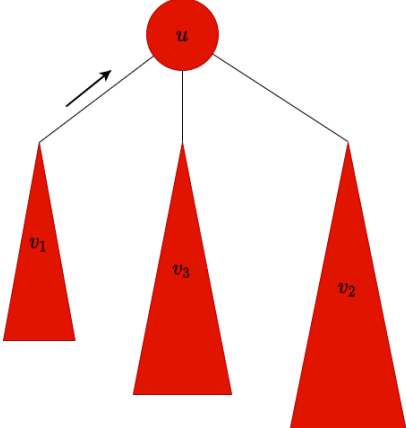
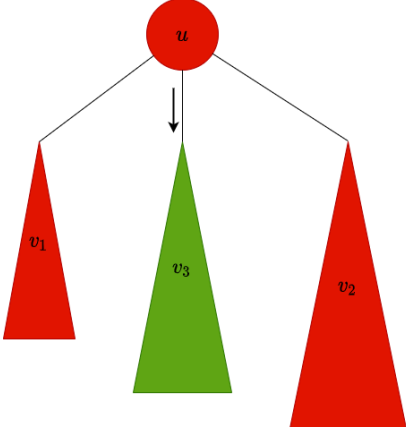
Ta sẽ sử dụng duy nhất một mảng để đếm số lượng từng màu trong một cây con. Bằng cách nào đó, đến cuối hàm dfs của cây con gốc u , ta có sẽ có mảng cnt với $cnt[c]$ là số lượng đỉnh có màu c trong cây con gốc u .

Do dùng chung một mảng chứ không phải lưu riêng từng cây con trong map riêng biệt, nên trước khi ta dfs xuống để giải bài toán cho các con tiếp theo, ta phải cập nhật "xóa" đi các màu của các cây con đã dfs từ trước khỏi mảng để tránh ghi đè lên đáp án. Do đó ta thấy chỉ có màu trong cây con được dfs cuối cùng là không nhất thiết phải xóa đi ngay lập tức.

Gọi đỉnh v có cây con lớn nhất là **bigChild**. Với ý tưởng gộp màu trong các cây con khác vào cây con **bigChild**, hay nói cách khác ta không được phép di chuyển các màu trong cây con **bigChild**, ta sẽ đảo thứ tự dfs của **bigChild** xuống cuối cùng và giữ lại các màu trong mảng mà không xóa đi. Tiếp đó ta sẽ dfs xuống các cây không phải **bigChild** chỉ để thêm lại các màu vào mảng, qua đó mà tìm được số

lượng từng màu trong cây con gốc u . Dễ thấy bằng cách này ta đã loại bỏ hoàn toàn việc sử dụng cấu trúc dữ liệu map, qua đó mà giảm được một log trong độ phức tạp thời gian.

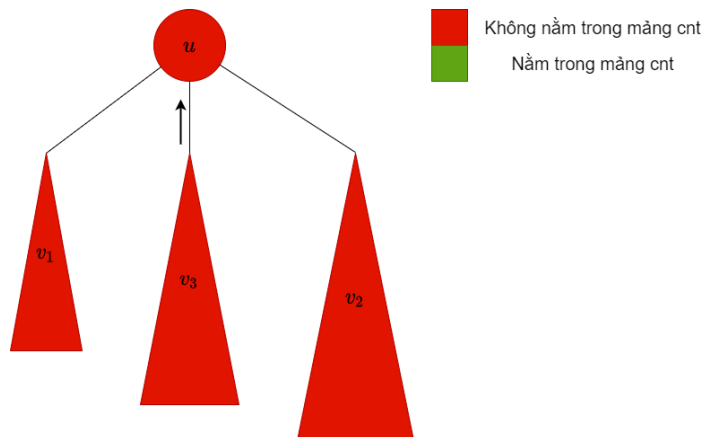
Ta có thể thấy rõ hơn thông tin mà mảng *cnt* lưu trữ trong quá trình sau đây:

Chú thích	Minh họa
dfs xuống cây con v_1 , lúc này trong mảng <i>cnt</i> chứa các màu trong cây con này	 <div><div></div> Không nằm trong mảng cnt</div> <div><div></div> Nằm trong mảng cnt</div>
v_1 không phải <i>bigChild</i> , do đó ta xóa các màu trong cây con này ra khỏi mảng <i>cnt</i>	 <div><div></div> Không nằm trong mảng cnt</div> <div><div></div> Nằm trong mảng cnt</div>
dfs xuống cây con v_3 , lúc này trong mảng <i>cnt</i> chứa các màu trong cây con này	 <div><div></div> Không nằm trong mảng cnt</div> <div><div></div> Nằm trong mảng cnt</div>

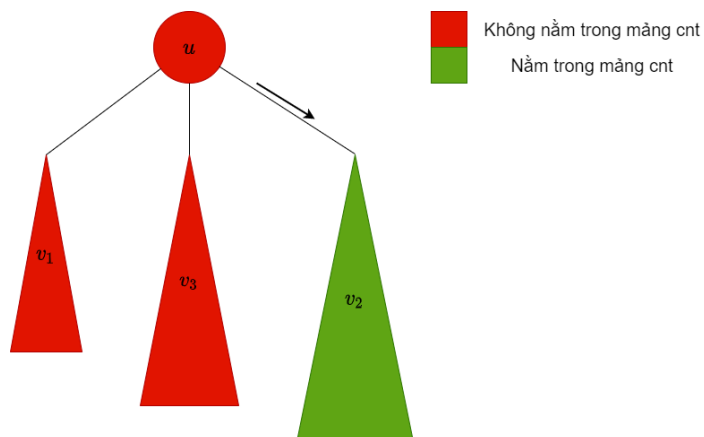
Chú thích

Minh họa

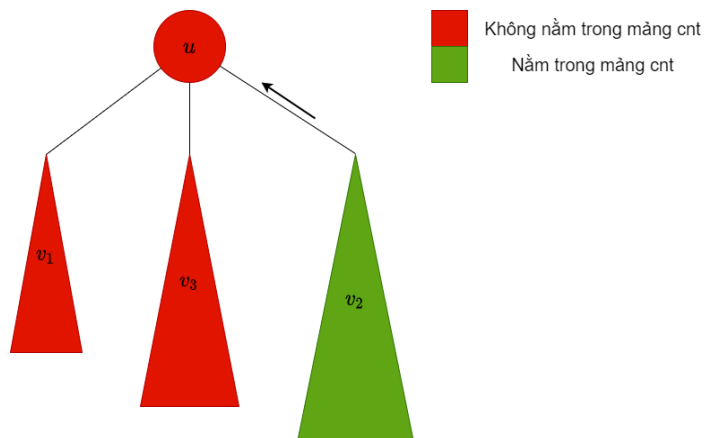
v_3 không phải **bigChild**, do đó ta xóa các màu trong cây con này ra khỏi mảng *cnt*

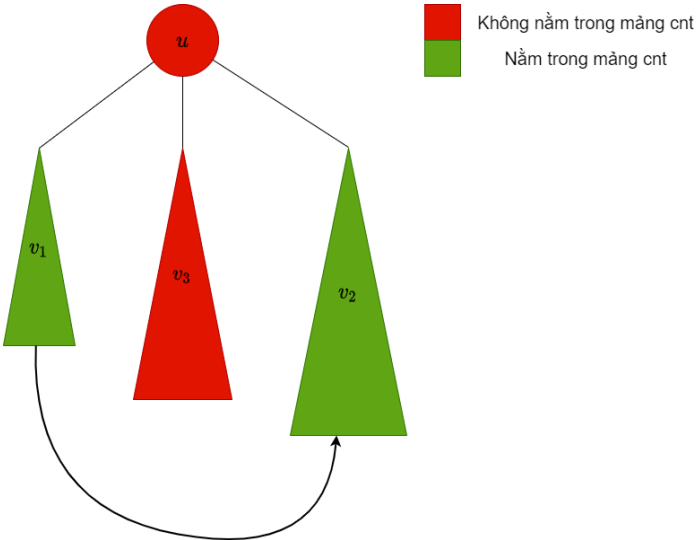
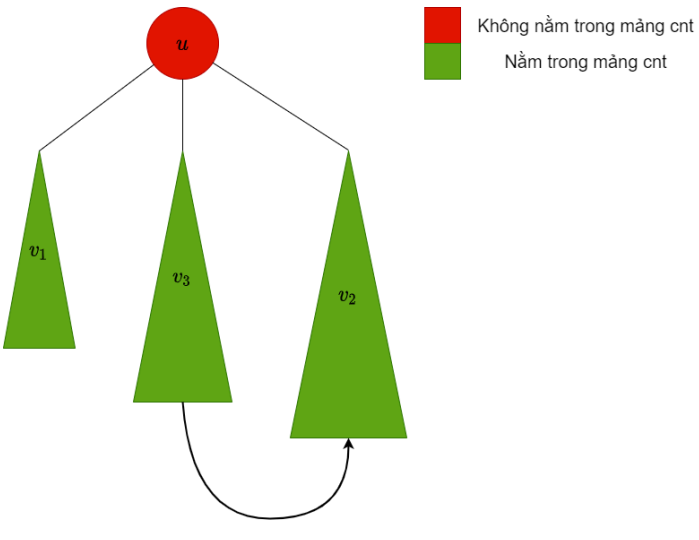
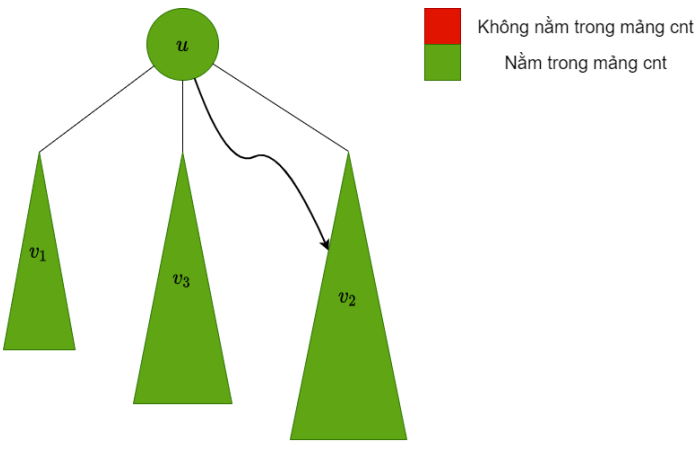


dfs xuống cây con v_2 , lúc này trong mảng *cnt* chứa các màu trong cây con này



v_2 là **bigChild**, do đó ta giữ nguyên các màu trong cây con này trong mảng *cnt*



Chú thích	Minh họa
Thêm các màu trong cây con v_1 vào mảng cnt	
Thêm các màu trong cây con v_3 vào mảng cnt	
Thêm đỉnh u vào mảng cnt	

Lúc này mảng cnt đã có đủ các màu trong cây con gốc u và ta có thể trả lời các truy vấn của đỉnh u .

Code mẫu

```




1 | #include <bits/stdc++.h>
2 |
3 |

```

```
4 using namespace std;
5
6 const int N = 5e5 + 5;
7 int n, q, color[N];
8 int sz[N];
9 int cnt[N];
10 int res[N]; // res[i] là đáp án của truy vấn thứ i
11 vector<pair<int, int>> queries[N];
12     // cặp (a, b) trong queries[v] có nghĩa là ở đỉnh v
13     // có truy vấn hỏi có bao nhiêu đỉnh trong cây con có màu a
14     // và số thứ tự của truy vấn là b
15 vector<int> g[N];
16
17 void pre_dfs(int u, int p) {
18     sz[u] = 1;
19     for (auto v : g[u]) if (v != p) {
20         pre_dfs(v, u);
21         sz[u] += sz[v];
22     }
23 }
24
25 void update(int u, int p, int delta) {
26     cnt[color[u]] += delta;
27     for (auto v : g[u]) if (v != p) update(v, u, delta);
28 }
29
30 void dfs(int u, int p) {
31     int bigChild = -1;
32     for (auto v : g[u]) if (v != p) {
33         if (bigChild == -1 || sz[v] > sz[bigChild]) bigChild = v;
34     }
35
36     for (auto v : g[u]) if (v != p && v != bigChild) {
37         dfs(v, u);
38         update(v, u, -1);
39     }
40     if (bigChild != -1) dfs(bigChild, u);
41
42     for (auto v : g[u]) if (v != p && v != bigChild) update(v, u, 1);
43     cnt[color[u]]++;
44
45     for (auto p : queries[u]) res[p.second] = cnt[p.first];
46 }
47
48 signed main() {
49
50     ios_base::sync_with_stdio(false); cin.tie(NULL);
51
52     cin >> n >> q;
53
54     for (int i = 1; i <= n; i++) cin >> color[i];
```






```
55  
56     for (int i = 1; i < n; i++) {  
57         int a, b;  
58         cin >> a >> b;  
59         g[a].push_back(b);  
60         g[b].push_back(a);  
61     }  
62  
63     for (int i = 1; i <= q; i++) {  
64         int v, c;  
65         cin >> v >> c;  
66         queries[v].push_back({c, i});  
67     }  
68  
69     pre_dfs(1, 0);  
70     dfs(1, 0);  
71  
72     for (int i = 1; i <= q; i++) cout << res[i] << "\n";  
73  
}
```

Tham khảo



- [CP-Algorithms](#) 
- [Codeforces ITMO Academy: pilot course](#) 
- [Codeforces blog: DSU trên cây \(Sack\)](#) 

Bài tập

Disjoint Set Union

- [Codeforces ITMO Academy: pilot course](#) 
- [Codeforces Problemset](#) 
- [VNOJ ILSBIN](#) 
- [VOI 2011 Bài 6](#) 
- [VOI 2020 Bài 2](#) 

Kỹ thuật Gộp set

- [Codeforces 1380E](#) 
- [SGU 507](#) 
- Các bài trong [phần DSU trên cây](#)

Kỹ thuật DSU trên cây

- [Codeforces 208E](#) 
- [Codeforces 246E](#) 
- [Codeforces 600E](#) 
- [Codeforces 570D](#) 
- [IOI 2011 - Race](#) 

Được cung cấp bởi [Wiki.js](#)