Mục lục

Link-cut tree Vs. Dynamic connectivity

1. Bài toán liên thông động (dynamic connectivity)

Cho đồ thị vô hướng với tập đỉnh cố định $V = \{1,2, \dots, n\}$, tập cạnh E có thể thay đổi (thêm cạnh, xóa cạnh). Cần trả lời theo thời gian các truy vấn dạng: hai đỉnh u, v có cùng thuộc một thành phần liên thông hay không.

Bài toán có ba mức độ khó

- Các cạnh chỉ được thêm vào E (incremental connectivity)
- Các cạnh chỉ được xóa khỏi E (decremental connectivity)
- Các cạnh có thể được thêm vào hay xóa đi khỏi E (fully dynamic connectivity)

1.1. Incremental connectivity

Quá trình giải bài toán hoàn toàn tương đương với việc xây dựng – quản lí cây/rừng khung của đồ thị.

Dễ dàng giải quyết bài toán bằng một cấu trúc dữ liệu quản lí các tập hợp rời (disjoint-set data structure). Sử dụng kĩ thuật nén lộ trình và hợp theo hạng, đạt được chi phí khấu trừ cho mỗi truy vấn là $\Theta(\alpha(n))$ với $\alpha(n)$ là hàm ngược của hàm Ackermann.

Cài đặt theo cách trên chính là cài đặt thuật toán Kruskal.

1.2. Decremental connectivity

Nếu có thể giải quyết offline, ta chỉ cần đảo chiều xảy ra các sự kiện xóa cạnh, bài toán trở thành dạng chỉ được thêm cạnh ở trên.

Giả thiết phải thực hiện xóa cạnh và trả lời truy vấn online. Ta cần duy trì một bảng cho biết mỗi đỉnh thuộc thành phần liên thông nào. Với bảng này, việc trả lời truy vấn tốn chi phí $O(1)$. Nhiệm vụ chính là cập nhật bảng trong mỗi sự kiện xóa.

Ta xét hai trường hợp: đồ thị không có chu trình (trong mọi thời điểm) và đồ thị vô hướng tổng quát.

1.2.1. Đồ thị không có chu trình (acyclic graph)

Với mỗi sự kiện xóa cạnh (u, v) thực hiện song song hai tiến trình: loang từ u và loang từ v.

Việc xóa cạnh trong cây chắc chắn tách cây thành hai cây mới, do đó phải có một tiến trình kết thúc trước. Cập nhật bảng chỉ số thành phần liên thông T cho mọi đỉnh được thăm của tiến trình này giúp ta hoàn thành xử lí xóa cạnh (u, v).

Rõ ràng ta luôn cập nhật bảng T cho cây nhỏ hơn trong hai cây mới, điều này chứng tỏ chi phí khấu trừ cho mỗi thao tác xóa cạnh là $O(\log n)$.

Chú ý rằng, loang song song trình bày ở trên chỉ cần đơn giản là thực hiện xen kẽ các bước loang của hai tiến trình (thực hiện một bước trong tiến trình A, rồi thực hiện một bước trong tiến trình B, rồi thực hiện một bước trong tiến trình A, và cứ như vậy …)

1.2.2. Đồ thị vô hướng tổng quát

So với trường hợp đồ thị không có chu trình ở trên, phát sinh khả năng: việc xóa cạnh (u, v) không làm tăng số thành phần liên thông.

Với mỗi sự kiện xóa cạnh (u, v), ta sẽ thực hiện song song/xen kẽ hai tiến trình A, B:

- Tiến trình A: phát hiện thành phần liên thông mới. Tiến trình này gồm hai tiến trình con song song/xen kẽ giống như với đồ thị không có chu trình: loang từ u, loang từ v. Nếu một trong hai tiến trình con kết thúc mà không gặp đỉnh thuộc tiến trình con còn lại), thì cập nhật bảng chỉ số thành phần liên thông T, kết thúc A và kết thúc B.
- Tiến trình B: phát hiện và cập nhật dữ liệu cho khả năng xóa cạnh (u, v) không làm tăng số thành phần liên thông.

Để hỗ trợ tiến trình B, cần xây dựng trước một cấu trúc dữ liệu lưu trữ thông tin BFS, ta sẽ gọi tắt là cấu trúc BFS. Tiến trình B sẽ cập nhật cấu trúc dữ liệu này.

Khởi tạo cấu trúc BFS (trước mọi sự kiện xóa cạnh) như sau

- Chọn một đỉnh r làm gốc, loang từ r, ghi nhận cấp của mỗi đỉnh là khoảng cách đến r (cấp của r bằng 0). Nếu đồ thị không liên thông thì lần lượt chọn v chưa thăm nào đó, ghi nhận một cạnh giả nối v đến r, gán cấp v là 1 rồi loang từ v, … Chú ý rằng cạnh giả (r, v) chỉ sử dụng trong tiến trình B, có thể coi cạnh giả như một cạnh đã xóa và ngược lại.
- Sau khi tất cả các đỉnh đã được thăm, phân loại các cạnh liên thuộc với mỗi đỉnh x (giả sử $lev(x) = k$) như sau:
    - Cạnh $backward$: nối đến y có $lev(y) = k - 1$, đưa vào $backward(x)$
    - Cạnh $forward$: nối đến y có $lev(y) = k + 1$, đưa vào $forward(x)$
    - Cạnh $local$: nối đến y có $lev(y) = k$, đưa vào $local(x)$

Do trước đó có cạnh (u, v) nên chênh lệch giữa $lev(u)$ và $lev(v)$ là không vượt quá 1, không mất tổng quát có thể giả thiết $lev(u) \leq lev(v)$. Tiến trình B thực thi như sau

- Nếu $lev(u) = lev(v)$: u, v đều có cạnh ngược lên gốc r, xóa cạnh (u, v) không làm tăng số thành phần liên thông. Cập nhật: xóa u khỏi $local(v)$, xóa v khỏi $local(u)$. Kết thúc B (và kết thúc A), T không thay đổi.
- Nếu $lev(u) = k - 1, lev(v) = k$, xóa u khỏi $backward(v)$, xóa v khỏi $forward(u)$, sau đó kiểm tra $backward(v)$
    - Nếu $backward(v) \neq \emptyset$: ngoài cạnh (u, v), v còn có cạnh backward khác, nghĩa là xóa cạnh (u, v) không làm u, v mất liên thông, kết thúc B (và kết thúc A), T không đổi
    - Nếu $backward(v) = \emptyset$ (cạnh (u, v) là cạnh backward duy nhất của v). Sau lệnh xóa $lev(v)$ trở thành $k + 1$, thực hiện việc loang từ v để cập nhật cấp và phân loại cạnh cho các đỉnh liên quan. Khi rơi vào trường hợp này, ta có ba tiến trình loang song song/xen kẽ và chúng hoặc kết thúc cùng thời điểm (trường hợp bảng T không thay đổi, do nguyên lí meet-in-the-middle) hoặc kết thúc do A phát hiện phát sinh thành phần liên thông mới.

Để cấu trúc dữ liệu BFS được cập nhật chính xác cho các lượt xóa cạnh sau, nếu tiến trình A phát hiện thành phần liên thông mới, thì cần bỏ qua các thay đổi đã thực hiện trong B, trong cấu trúc dữ liệu BFS trước đó ghi nhận (u, v) là cạnh giả.

Trong trường hợp xóa cạnh (u, v) không phát sinh thành phần liên thông mới, pha loang của B có thể dẫn đến trường hợp xấu nhất: cấp của một đỉnh có thể đạt $O(n)$. Điều này làm cho chi phí khấu trừ của mỗi lần xóa cạnh đạt $O(n)$.

1.3. Fully dynamic connectivity

Bài toán với đồ thị vô hướng tổng quát và chấp nhận cả thêm và xóa cạnh có thể giải quyết bằng Euler tour tree và một cấu trúc dữ liệu hỗ trợ: cấu trúc phân cấp đỉnh (Level structure), cấu trúc tập cạnh cắt (Cutset structure, đây là một cách lưu thông tin tập đỉnh bằng dãy bit và phép XOR rất thú vị, đáng học hỏi). Cài đặt bài toán này tương đối cồng kềnh, phức tạp, ít ứng dụng trong lập trình thi đấu nên tạm xin không bàn đến ở đây.

Ta đặt mối quan tâm chủ yếu vào trường hợp đồ thị không xuất hiện chu trình trong toàn bộ quá trình thêm/xóa cạnh.

Do không có chu trình, đồ thị là một rừng, hoàn toàn có thể coi mỗi cây trong đó là cây có gốc. Thao tác xóa cạnh làm phát sinh cây con mới. Thao tác thêm cạnh đòi hỏi phải hợp hai cây thành cây mới. Các vấn đề phát sinh là

- Hai nút u, v liên thông nếu chúng thuộc cùng một cây, để kiểm tra khả năng này cần nhanh chóng tìm được gốc của cây chứa u, v
- Thêm hay xóa cạnh ứng với nối hai cây với nhau, cắt một cây con khỏi một cây. Việc tìm ra cạnh/cây để nối hay cắt là không khó, thử thách chủ yếu là cách thức lưu trữ hỗ trợ quá trình tìm gốc của cây hay lưu thông tin tích lũy cho các truy vấn trên đường đi đơn hay trên cây con.

Hai cấu trúc dữ liệu phổ biến hỗ trợ được việc tìm kiếm gốc cây trong $O(\log n)$ đồng thời cho phép truy vấn các thông tin tích lũy là Euler tour tree và Link-cut tree.

Đối với Euler tour tree, thao tác tìm gốc được quy về tìm phần tử nhỏ nhất trong một đoạn, thao tác cắt hay nối quy về cắt và dán một đoạn từ vị trí này sang vị trí khác. Cài đặt quản lí dãy số hỗ trợ cắt, dán đoạn hết sức phức tạp.

Link-cut tree là cấu trúc dữ liệu phục vụ chính mục tiêu quản lí cây động được sáng tạo năm 1982 bởi Daniel Dominic Sleator và Robert Endre Tarjan. Concept ban đầu của giải pháp là phân rã cây thành các đường đi đơn gọi là đường ưu tiên (prefered path), quản lí mỗi đường ưu tiên bởi một binary search tree (BST). Cây trở thành cây trên các nút mà mỗi nút là một BST.

Việc lần ngược về gốc của cây nguyên bản được tăng tốc nhờ hai yếu tố:

- binary lifting mỗi khi đi vào một đường ưu tiên
- biến đổi cây trong mỗi thao tác thực hiện, các nút được truy cập nhiều được đẩy vào đường ưu tiên hoặc đẩy gần đến đường ưu tiên hơn.

Cũng chính hai tác giả Daniel Dominic Sleator và Robert Endre Tarjan đã sáng tạo splay tree sau đó 3 năm (1985). Từ thời điểm này splay tree được thay thế cho BST trong các cài đặt link-cut tree. Splay tree không quan tâm đến việc cân bằng cây, nó chỉ đơn giản luôn đẩy nút được truy cập mới nhất về gốc cây thông qua thao tác splay. Có lẽ không sai biệt lắm nếu nói rằng việc phát minh ra splay tree là dựa trên nền móng của link-cut tree.

Cũng có các thử nghiệm thay thế BST bởi các cấu trúc dữ liệu cây nhị phân tự cân bằng, chẳng hạn như treap, tuy nhiên không có cấu trúc nào đơn giản và hiệu quả hơn splay tree.

Bài viết này sau đây tập trung vào giới thiệu link-cut tree ở các phương diện

- Cài đặt đơn giản nhất, chấp nhận một số hạn chế trong thêm/xóa cạnh
- Cài đặt đầy đủ, cho phép thao tác thêm/xóa cạnh tùy ý
- Cách tổ chức dữ liệu bổ sung để giải quyết các các truy vấn thông tin tích lũy được trên cây: trọng số cạnh trên đường đi đơn trong đồ thị, trọng số đỉnh trên các cây con, …

2. Giới thiệu link-cut tree

Cấu trúc dữ liệu link-cut tree biểu diễn một rừng các cây có gốc và có thể thực hiện các thao tác sau trong thời gian khấu trừ $O(\log n)$ với n là số nút:

- $link(p, v)$: Nối một cây gốc v thành nút con của p (trước đó p, v không cùng cây).
- $cut(v)$: Tách cây con gốc v khỏi nút cha (của v).
- $find\_root(v)$: Tìm gốc của cây con chứa nút v.

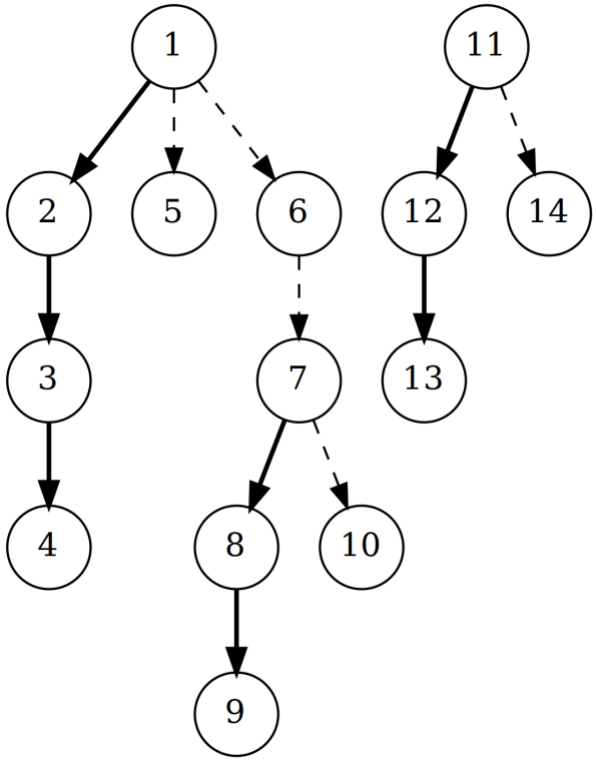
- $lca(u, v)$: Tìm tổ tiên chung gần nhất của hai nút u, v.

Rừng (biểu diễn bởi link-cut tree) được phân rã thành các **đường ưu tiên** thỏa mãn

- Mỗi đường ưu tiên bắt đầu từ một nút và hướng xuống một lá nào đó,
- Hai đường ưu tiên bất kì không có nút chung,
- Các cạnh ưu tiên nối mỗi nút với nút con mới được truy cập gần nhất, ràng buộc này nhằm lợi dụng giả thuyết: tồn tại 20% nút chiếm 80% số truy cập. Những nút được truy cập nhiều nhất nên có thời gian xử lí nhanh nhất. Đây cũng là cơ sở xây dựng nên cấu trúc dữ liệu splay tree.

Ta lưu trữ mỗi đường ưu tiên bằng một splay tree, khóa so sánh của mỗi nút là độ sâu của nút đó. Mỗi đường ưu tiên có một con trỏ **path-parent** trỏ đến nút cha (trong cây nguyên bản) của nút cao nhất (nếu có). Vì vậy, mỗi cạnh là một **cạnh ưu tiên** hoặc cạnh path-parent (sau đây sẽ gọi là cạnh bổ trợ).

Minh họa



Trong rừng ở hình trên, các cạnh đậm là cạnh ưu tiên, các cạnh nét đứt là cạnh bổ trợ. 1,11 là các nút gốc của hai cây trong rừng. các đường ưu tiên là: (1,2,3,4), (5), (6), (7, 8, 9), (10), (11, 12, 13), (14).

Một số chú ý:

- Ta có một rừng/cây nguyên bản, nó được biểu diễn bằng một tập các cây, để tránh nhập nhằng về tên, tạm gọi các cây đó là các cây bổ trợ (auxiliary tree),
- Mỗi cây bổ trợ gồm các nút là splay tree biểu diễn các đường ưu tiên, cạnh nối các splay tree là cạnh không thuộc đường ưu tiên (cạnh path-parent), tạm gọi là cạnh bổ trợ,
- Mỗi splay tree lưu trữ một đường ưu tiên, quan hệ cha-con trong splay tree cơ bản là không liên quan đến quan hệ cha-con trong rừng nguyên bản,
- Nút gốc trong splay tree có thể không phải là nút cao nhất trong đường ưu tiên,
- Nút cha trong splay tree khác với nút cha trong cây bổ trợ,
- Cạnh trong splay tree khác với cạnh bổ trợ.

2.1. Splay tree

Mỗi splay tree lưu thêm một cạnh bổ trợ (có thể không có cạnh này nếu nó là nút gốc trong cây bổ trợ). Tuy nhiên, để đơn giản, ta sẽ lưu trường cạnh bổ trợ vào mỗi nút trong splay tree vì trong quá trình hoạt động, mỗi nút trong splay tree đều có cơ hội được splay thành gốc, cạnh bổ trợ của một splay tree xác định chính xác từ nút gốc của splay tree đó. Nếu xét trong rừng nguyên bản, cạnh bổ trợ là cạnh nối nút cao nhất trong đường ưu tiên đến nút cha.

Source

```
template<typename T>
struct SplayNode{
  SplayNode(){}
  SplayNode(T value_arg): value{value_arg} {}
  T value{};                      // chỉ số của nút
  array<SplayNode *, 2> child{}; // con trái, phải (splay tree)
  SplayNode *parent{};           // nút cha trong splay tree
  Node* path_parent{};           // cạnh bổ trợ nối lên nút cha trong cây bổ trợ
  bool side() const {
    // xác định *this là con trái hay phải
    return parent->child[1] == &this;
  }
  void rotate() {
    const auto p = parent;
    const bool i = side();

    if (p->parent) {
      p->parent->attach(p->side(), &this);
    } else {
      parent = nullptr;
    }
    p->attach(i, child[!i]);
    attach(!i, p);
    this.path_parent = p->path_parent; // cạnh bổ trợ của splay tree
  }
  void splay() {
    // splay nút *this thành gốc splay tree
    for(;parent;rotate()) {
      if (parent->parent) {
        (side() == parent->side() ? parent: &this)->rotate();
      }
    }
  }
  array<SplayNode *, 2> split() {
    splay();
    // tách con phải của *this khỏi splay tree
    const auto right = child[1];
    if (right) {
      right->parent = nullptr;
    }
    this->right = nullptr;
    return {&this, right};
  }
  void attach(bool side, SplayNode *const new_) {
    // Nối *new_ vào con trái/phải của *this
    if (new_) {
      new_->parent = &this;
    }
    child[side] = new_;
  }
};
```

Cài đặt splay tree trên đây gom tất cả các trường hợp quay một nút (Zig, Zag, ZigZig, ZigZag, ZagZig, ZagZag) vào một thao tác $rotate()$, cách làm này chủ yếu là để rút ngắn độ dài chương trình, phục vụ cho lập trình thi đấu. Ta hoàn toàn có thể sử dụng các cài đặt splay tree truyền thống có thể tìm được trong các tài liệu thuật toán phổ biến.

2.2. Thao tác access(v)

Đây là thao tác cơ bản được thực hiện liên tục trong quá trình duy trì cây bổ trợ. Thao tác này bao gồm các công việc

- Trong splay tree chứa v, splay v về gốc
- Trên đường từ v về gốc của cây bổ trợ, giả sử p là cha của v
    - Chuyển cạnh ưu tiên đến con phải của p (trong splay tree chứa p) thành cạnh bổ trợ (hàm $detach\_child(v)$).
    - Chuyển cạnh bổ trợ của splay tree gốc v thành cạnh ưu tiên, tức là v trở thành con phải p trong splay tree.
    - Chú ý rằng tại mọi bước lặp, cần splay cả p, v để đảm bảo hiệu năng.

Souce

```
Node *make_tree() {
  // tạo cây mới
  return new Node{};
}
void detach_child(Node* node){
  // chuyển cạnh ưu tiên nối với con phải thành cạnh bổ trợ
  if(node->child[1]){
    node->child[1]->path_parent = node;
    node->child[1]->parent = nullptr;
  }
}
Node *access(Node *node) {
  // di chuyển từ nút node lên gốc cây bổ trợ (qua cạnh bổ trợ)
  // thay đổi tính chất các cạnh phù hợp
  // đường từ node đến gốc (cây nguyên bản) trở thành đường ưu tiên
  node->splay();
  detach_child(node);
  node->child[1] = nullptr;
  Node *par = node;
  while (node->path_parent) {
    par = node->path_parent;
    par->splay();
    detach_child(par);
    par->attach(1, node);
    node->path_parent = nullptr;
    node->splay();
  }
  return par;
}
```

2.3. Thao tác find_root(v)

- Thực hiện $access(v)$: các nút trên đường lên gốc cây bổ trợ (theo các cạnh bổ trợ) từ v lần lượt trở thành gốc các splay tree tương ứng
- Trong splay tree chứa nút gốc của cây bổ trợ, liên tiếp đi theo nhánh trái đến con cực trái vr, đây sẽ là nút cao nhất và là gốc (của cây nguyên bản) cần tìm
- Thực hiện $access(vr)$ một lần nữa để tăng tốc $find\_root()$ trong các lần sau

Source

```
Node *find_root(Node *node) {
  // tìm gốc của cây bổ trợ chứa nút node
  access(node);
  for (; node->child[0]; node = node->child[0]);
  access(node);
  return node;
}
```

2.4. Thao tác cut(v)

- Thực hiện $access(v)$,
- Gỡ nhánh con trái của v, đây là nút cha trực tiếp của v trong cây nguyên bản.

Source

```
void cut(Node *node) {
  // Tách cây con gốc node khỏi nút cha
  access(node);
  node->child[0]->parent = nullptr;
  node->child[0] = nullptr;
}
```

2.5. Thao tác link(p, v)

Mục tiêu: gắn cây bổ trợ gốc v thành con nút p

- $access(v)$
- $access(p)$
- Nối v vào con trái của p

Source

```
void link(Node *par, Node *child) {
  // gắn child thành nút con của par
  access(child);
  access(par);
  child->attach(0, par);
}
```

Chú ý rằng thao tác này được đơn giản hóa rất nhiều vì chỉ xét trường hợp v là gốc của một cây bổ trợ.

2.6. Truy vấn lca(u, v)

- Kiểm tra u, v có cùng thuộc một cây
- Nếu có, thực hiện $access(u)$ rồi $access(v)$, nút được trả về từ $access(v)$ chính là nút phân tách hai cây con chứa u, v.

Source

```
Node *lca(Node *u, Node *v) {
  // Tìm tổ tiên chung gần nhất của u và v
  if (find_root(u) != find_root(v)) {
    return nullptr;
  }
  access(u);
  return access(v);
}
```

2.7. Link-cut tree đầy đủ

Bản cài đặt link-cut tree ở trên mang ý nghĩa lí thuyết, nó chủ yếu được sử dụng để chứng minh độ phức tạp tính toán của cấu trúc dữ liệu. Ta cần cài đặt link-cut tree với $link(v, p)$ chấp nhận v là tùy ý, miễn là không cùng cây con với p.

Để làm được điều đó, ta cần khả năng lật đối xứng nhánh trái splay tree gốc v (sau khi $access(v)$), và thao tác lật này phải được lan truyền lười.

Dưới đây giới thiệu một cài đặt link-cut tree đầy đủ có bổ sung stress test để kiểm tra. Cài đặt này của thành viên Codeforces **bicsi** (https://codeforces.com/profile/bicsi), có một vài cách cài đặt ngắn hơn, nhưng cũng khó hiểu hơn và mang nặng tính thủ thuật, mẹo mực.

```
#include <bits/stdc++.h>
using namespace std;

struct LinkCut {
  struct Node {
    int p = 0, c[2] = {0, 0}, pp = 0;
    bool flip = 0;
    int val = 0, dp = 0;
  };
  vector<Node> T;

  LinkCut(int n) : T(n + 1) {}

  // SPLAY TREE OPERATIONS START

  int dir(int x, int y) { return T[x].c[1] == y; }

  void set(int x, int d, int y) {
    if (x) T[x].c[d] = y, pull(x);
    if (y) T[y].p = x;
  }

  void pull(int x) {
    if (!x) return;
    int &l = T[x].c[0], &r = T[x].c[1];
    T[x].dp = max({T[x].val, T[l].dp, T[r].dp});
  }

  void push(int x) {
    if (!x || !T[x].flip) return;
    int &l = T[x].c[0], &r = T[x].c[1];
    swap(l, r); T[l].flip ^= 1; T[r].flip ^= 1;
    T[x].flip = 0;
  }

  void rotate(int x, int d) {
    int y = T[x].p, z = T[y].p, w = T[x].c[d];
    swap(T[x].pp, T[y].pp);
    set(y, !d, w);
    set(x, d, y);
    set(z, dir(z, y), x);
  }

  void splay(int x) {
    for (push(x); T[x].p;) {
      int y = T[x].p, z = T[y].p;
      push(z); push(y); push(x);
      int dx = dir(y, x), dy = dir(z, y);
      if (!z)
        rotate(x, !dx);
```

```
      else if (dx == dy)
        rotate(y, !dx), rotate(x, !dx);
      else
        rotate(x, dy), rotate(x, dx);
    }
  }

  // SPLAY TREE OPERATIONS END

  void MakeRoot(int u) {
    Access(u);
    int l = T[u].c[0];
    T[l].flip ^= 1;
    swap(T[l].p, T[l].pp);
    set(u, 0, 0);
  }

  void Access(int _u) {
    for (int v = 0, u = _u; u; u = T[v = u].pp) {
      splay(u); splay(v);
      int r = T[u].c[1];
      T[v].pp = 0;
      swap(T[r].p, T[r].pp);
      set(u, 1, v);
    }
    splay(_u);
  }

  void Link(int u, int v) {
    assert(!Connected(u, v));
    MakeRoot(v);
    T[v].pp = u;
  }

  void Cut(int u, int v) {
    MakeRoot(u); Access(u); splay(v);
    assert(T[v].pp == u);
    T[v].pp = 0;
  }

  bool Connected(int u, int v) {
    if (u == v) return true;
    MakeRoot(u); Access(v); splay(u);
    return T[v].p == u || T[T[v].p].p == u;
  }

  int GetPath(int u, int v) {
    MakeRoot(u); Access(v); return v;
  }
};

void Test() {
  int N = 100, Q = 100, V = 1000;
  while (true) {
    int n = rand() % N + 1;
    int q = rand() % Q + 1;
    int p1 = rand() % 100, p2 = rand() % 100,
        p3 = rand() % 100, p4 = rand() % 100, p5 = rand() % 100;

    vector<pair<int, int>> edges;
    LinkCut lc(n);

    auto conn = [&](int a, int b) {
```

```
    vector<int> dp(n + 1, -1);
    dp[a] = lc.T[a].val;
    for (int ch = 1; ch >= 0; ch--) {
      for (auto p : edges) {
        for (int it = 0; it < 2; ++it) {
          if (dp[p.first] != -1 && dp[p.second] == -1) {
            dp[p.second] = max(dp[p.first], lc.T[p.second].val);
            ch = 1;
          }
          swap(p.first, p.second);
        }
      }
    }
    return dp[b];
  };


  auto sim_op = [&]() {
    while (true) {
      int t = rand() % (p1 + p2 + p3 + p4 + p5);

      if (t < p1) {
        int a = rand() % n + 1, b = rand() % n + 1;
        if (conn(a, b) == -1) {
          edges.emplace_back(a, b);
          lc.Link(a, b);
          return;
        }
        continue;
      }

      t -= p1;

      if (t < p2) {
        if (edges.empty()) continue;
        int pos = rand() % edges.size();
        swap(edges[pos], edges.back());
        // cerr << "CUT: " << edges.back().first << " " << edges.back().second << endl;
        lc.Cut(edges.back().first, edges.back().second);
        edges.pop_back();
        return;
      }

      t -= p2;

      if (t < p3) {
        int node = rand() % n + 1;
        lc.Access(node);
        lc.T[node].val = rand() % V + 1;
        lc.pull(node);
        // cerr << "UPDATE: " << node << endl;
        return;
      }

      t -= p3;

      if (t < p4) {
        int a = rand() % n + 1, b = rand() % n + 1;
        int expected = conn(a, b);
        if (expected != -1) {
          // cerr << "QUERY: " << a << " " << b << endl;
          int c = lc.GetPath(a, b);
          int actual = lc.T[c].dp;
```

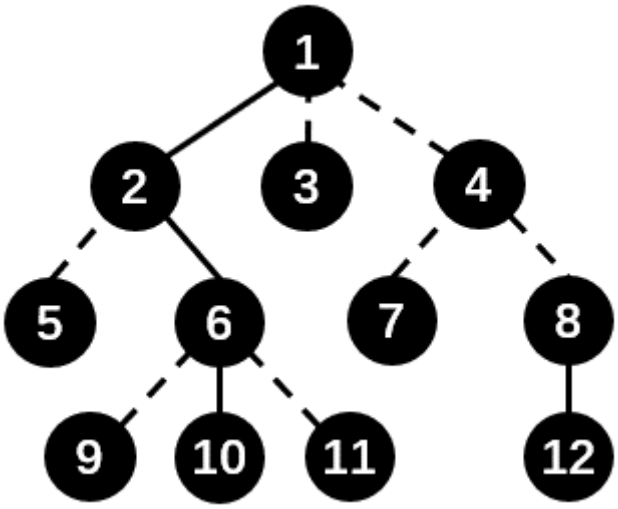
```
            assert(expected == actual);
            return;
          }
          continue;
        }

        t -= p4;

        if (t < p5) {
          int a = rand() % n + 1, b = rand() % n + 1;
          // cerr << "CONNECTED: " << a << " " << b << endl;
          int expected = (conn(a, b) != -1);
          // cerr << "EXP: " << expected << endl;
          int actual = lc.Connected(a, b);
          assert(expected == actual);
          return;
        }

        assert(false);
      }
    };

    for (int i = 0; i < q; ++i) {
      sim_op();
    }

    cerr << "OK N = " << n << " Q = " << q << endl;
  }
}

int main() {
  Test();
  return 0;
}
```

3. Quản lí các cây con

Khi cài đặt link-cut tree quản lí một rừng các cây có gốc, ngoài mục tiêu xác định tính liên thông hay tìm tổ tiên chung gần nhất, ta còn muốn thực hiện các truy vấn yêu cầu tính toán các thông tin tích lũy trên các đường đi đơn hay thông tin tích lũy tại gốc các cây con (trong rừng nguyên bản).

Sau đây giới thiệu cách thức lưu thông tin của các cây con: ghi nhận tại mỗi nút v tổng tích lũy của các nút u là hậu duệ *thực sự* của v (u là hậu duệ v nhưng u không thuộc đường ưu tiên chứa v, cũng có nghĩa u không phải là con phải v trong splay tree chứa v). Mỗi u như vậy gọi là một cây con ảo của v.

Xem xét hình minh họa



Ở đây (1,2,6,10) là một đường ưu tiên, chúng cùng nằm trong một splay tree, tương tự với (8,12). Nút 1 có hai cây con ảo gốc 3,4, nút 4 có hai cây con ảo gốc 7,8, nút 8 không có cây con ảo nào.

Ta cần cập nhật thông tin khi các cây con ảo thay đổi, điều này xảy ra trong các thao tác *access* và *link*. Giả thiết cần lưu trữ thông tin tổng kích thước (số nút) của các cây con ảo.

3.1. Thông tin cần ghi nhận

Nút cha (trong link-cut tree), hai con trái/phải (trong splay tree), kích thước cây con (bao gồm gốc), tổng kích thước các cây con ảo

```
struct Node
{
    int fa, ch[2], siz, vir;
} t[N];
```

Đây là các thông tin cần xử lí chính, tạm lược bỏ các yếu tố cần cho việc lan truyền thao tác đảo một đoạn trên đường ưu tiên.

3.2. Cập nhật trong thao tác *access*

Khi đi dọc theo các cạnh bổ trợ để lên gốc của link-cut tree, ở mỗi bước có một cạnh bổ trợ trở thành cạnh ưu tiên, một cạnh ưu tiên trở thành cạnh bổ trợ, gia giảm thông tin cây con ảo phù hợp với mỗi trường hợp.

Ngoài ra, cần tính lại kích thước cây con bằng tổng của: kích thước con trái, con phải (splay tree), tổng kích thước các cây con ảo, và 1 cho chính nút đó (hàm *pushup*).

```
void pushup(int x)
{
    t[x].siz = t[t[x].ch[0]].siz + t[t[x].ch[1]].siz + t[x].vir + 1;
}

void access(int x)
{
    for (int y = 0; x; x = t[y = x].fa)
    {
        Splay(x);
        t[x].vir -= t[y].siz;           // x-y trở thành cạnh ưu tiên
        t[x].vir += t[t[x].ch[1]].siz; // con phải cũ của x trở thành con bổ trợ
        t[x].ch[1] = y;
```

```
        pushup(x);                    // tính lại size(x)
    }
}
```

3.3. Cập nhật trong thao tác *link*

Sau khi nối x vào thành con y, x trở thành một cây con ảo của y.

```
void link(int x, int y)
{
    makeroot(x);
    access(y);
    Splay(y);
    t[x].fa = y;
    t[y].vir += t[x].siz;
}
```

3.4. Ứng dụng

3.4.1. Bài toán: Số đường đi

BJOI2014: https://www.luogu.org/problemnew/show/P4219

Cho đồ thị vô hướng gồm n đỉnh 1,2, … , n, ban đầu đồ thị chưa có cạnh nào. Quản lí đồ thị và trả lời các truy vấn

- $< A\ x\ y >$: thêm cạnh nối hai đỉnh x, y. Đảm bảo trong đồ thị không xuất hiện chu trình
- $< Q\ x\ y >$: xác định số lượng đường đi đơn (không lặp đỉnh) chứa cạnh (x, y). Đảm bảo cạnh (x, y) tồn tại trước đó.

Dữ liệu

- Dòng 1: n, m $(n, m \leq 10^5)$

Kết quả

- Dòng 1 …: kết quả trả lời các truy vấn $< Q\ x\ y >$

Thuật toán

Ta sử dụng một link-cut tree không có thao tác cut. Với mỗi truy vấn $< Q\ x\ y >$, mục tiêu là tính $size[x] \cdot size[y]$. Để có điều này cần thực hiện

- $access(x)$
- $make\_root(x)$: trong splay tree chứa x, đưa x về gốc, đảo chiều đường ưu tiên
- $access(y)$
- $splay(y)$
- Kết quả: $size(x) \cdot size(y) = (T(x).virtual + 1) \cdot (T(y).virtual + 1)$

Sources

[https://codeforces.com/profile/bicsi]

```
#include <cstdio>
#include <iostream>

using namespace std;

const int N = 100010;

struct Node
{
    int ch[2], fa, vir, siz;
    bool rev;
```

```
} t[N];

bool nroot(int x);
void rotate(int x);
void Splay(int x);
void access(int x);
void makeroot(int x);
void split(int x, int y);
void link(int x, int y);
void reverse(int x);
void pushup(int x);
void pushdown(int x);

int n, q, sta[N], top;

int main()
{
    int x, y;
    char op[10];

    scanf("%d%d", &n, &q);

    while (q--)
    {
        scanf("%s%d%d", op, &x, &y);
        if (op[0] == 'A') link(x, y);
        else
        {
            split(x, y);
            printf("%lld\n", 1ll * (t[x].vir + 1) * (t[y].vir + 1));
        }
    }

    return 0;
}

bool nroot(int x) { return x == t[t[x].fa].ch[0] || x == t[t[x].fa].ch[1]; }

void rotate(int x)
{
    int y = t[x].fa;
    int z = t[y].fa;
    int k = x == t[y].ch[1];
    if (nroot(y)) t[z].ch[y == t[z].ch[1]] = x;
    t[x].fa = z;
    t[y].ch[k] = t[x].ch[k ^ 1];
    t[t[x].ch[k ^ 1]].fa = y;
    t[x].ch[k ^ 1] = y;
    t[y].fa = x;
    pushup(y);
    pushup(x);
}

void Splay(int x)
{
    int u = x;
    sta[++top] = x;
    while (nroot(u)) sta[++top] = u = t[u].fa;
    while (top) pushdown(sta[top--]);
    while (nroot(x))
    {
        int y = t[x].fa;
        int z = t[y].fa;
```

```
        if (nroot(y)) (x == t[y].ch[1]) ^ (y == t[z].ch[1]) ? rotate(x) : rotate(y);
        rotate(x);
    }
}

void access(int x)
{
    for (int y = 0; x; x = t[y = x].fa)
    {
        Splay(x);
        t[x].vir += t[t[x].ch[1]].siz;
        t[x].ch[1] = y;
        t[x].vir -= t[t[x].ch[1]].siz;
        pushup(x);
    }
}

void makeroot(int x)
{
    access(x);
    Splay(x);
    reverse(x);
}

void split(int x, int y)
{
    makeroot(x);
    access(y);
    Splay(y);
}

void link(int x, int y)
{
    makeroot(x);
    access(y);
    Splay(y);
    t[x].fa = y;
    t[y].vir += t[x].siz;
}

void reverse(int x)
{
    swap(t[x].ch[0], t[x].ch[1]);
    t[x].rev ^= 1;
}

void pushup(int x)
{
    t[x].siz = t[t[x].ch[0]].siz + t[t[x].ch[1]].siz + t[x].vir + 1;
}

void pushdown(int x)
{
    if (t[x].rev)
    {
        reverse(t[x].ch[0]);
        reverse(t[x].ch[1]);
        t[x].rev = false;
    }
}
```

3.4.2. Bài toán: Cạnh quan trọng

UOJ207: http://uoj.ac/problem/207

Cho cây n nút, các nút đánh số 1,2, … , n. Duy trì một tập hợp S chứa các yêu cầu xác định đường đi giữa hai nút u, v nào đó. Ban đầu $S = \emptyset$. Quản lí cây và trả lời các truy vấn

- $< 1\ x\ y\ u\ v >$: xóa cạnh nối hai nút x, y, thêm cạnh nối hai nút u, v, dữ liệu đảm bảo đồ thị vẫn là cây sau truy vấn này.
- $< 2\ x\ y >$: thêm yêu cầu xác định đường đi giữa hai nút x, y vào S
- $< 3\ i >$: xóa khỏi S yêu cầu được thêm vào $(2\ x\ y)$ thứ i
- $< 4\ x\ y >$: kiểm tra mọi yêu cầu trong S có đều sử dụng cạnh nối hai nút x, y

Dữ liệu

- Dòng 1: n, m $(n \le 10^5; m \le 3 \cdot 10^5)$
- Dòng 2 … n: x, y thể hiện cạnh nối hai nút x, y trong cây ban đầu
- Dòng $n + 1$ … $n + m$: các truy vấn

Kết quả

- Dòng 1 …: trả lời của các truy vấn $4\ x\ y$.

Thuật toán

Mỗi nút u sẽ được gán một trọng số $w(u)$, quy ước trọng số một cây con T_u là $w(T_u)$ bằng tổng XOR của trọng số các nút trong T_u.

Khi thêm yêu cầu xác định đường đi giữa hai nút x, y (truy vấn $< 2\ x\ y >$), kí hiệu đường đi giữa hai nút x, y là $P(x, y)$), gán trọng số của $P(x, y)$ bằng một số ngẫu nhiên, trọng số x, y được XOR với $P(x, y)$. Như vậy, nếu x, y cùng thuộc một cây con T_u, tham gia của chúng trong $w(T_u)$ bằng 0.

Xét một truy vấn $< 4\ x\ y >$, lấy $w(T_x)$ XOR $w(T_y)$, nếu giá trị này bằng với tổng XOR của toàn bộ S thì kết quả truy vấn *hầu như* là khẳng định.

Với truy vấn xóa $P(x, y)$, cần phục hồi $w(x), w(y)$ và tính lại tổng XOR của S.

Cây link-cut tree được sử dụng trong bài này để quản lí các $w(T_u)$.

Source

[https://codeforces.com/profile/bicsi]

```cpp
#include <bits/stdc++.h>
using namespace std;

int read()
{
    int out = 0;
    char c;
    while (!isdigit(c = getchar()));
    for (; isdigit(c); c = getchar()) out = out * 10 + c - '0';
    return out;
}

typedef unsigned long long ull;

const int N = 100010;
const int M = 300010;

ull seed = time(0);
ull rd() { return seed = seed * 998244353 + 1000000007; }
```

```
struct Node
{
    int ch[2], fa;
    ull self, val, vir;
    bool rev;
} t[N];

bool nroot(int x);
void rotate(int x);
void Splay(int x);
void access(int x);
void makeroot(int x);
void link(int x, int y);
void cut(int x, int y);
void reverse(int x);
void pushup(int x);
void pushdown(int x);

int id, n, m, sta[N], top, sx[M], sy[M], stot;
ull xorp[M], xorsum;

int main()
{
    int i, x, y;

    id = read();
    n = read();
    m = read();

    for (i = 1; i < n; ++i) link(read(), read());

    while (m--)
    {
        switch (read())
        {
            case 1:
                cut(read(), read());
                link(read(), read());
                break;
            case 2:
                sx[++stot] = x = read();
                sy[stot] = y = read();
                xorp[stot] = rd();
                access(x);
                Splay(x);
                t[x].self ^= xorp[stot];
                access(y);
                Splay(y);
                t[y].self ^= xorp[stot];
                xorsum ^= xorp[stot];
                break;
            case 3:
                x = read();
                access(sx[x]);
                Splay(sx[x]);
                t[sx[x]].self ^= xorp[x];
                access(sy[x]);
                Splay(sy[x]);
                t[sy[x]].self ^= xorp[x];
                xorsum ^= xorp[x];
                break;
            case 4:
                x = read();
```

```
                y = read();
                makeroot(x);
                access(y);
                if ((t[y].vir ^ t[y].self) == xorsum) puts("YES");
                else puts("NO");
                break;
            }
        }

        return 0;
}

bool nroot(int x) { return x == t[t[x].fa].ch[0] || x == t[t[x].fa].ch[1]; }

void rotate(int x)
{
    int y = t[x].fa;
    int z = t[y].fa;
    int k = x == t[y].ch[1];
    if (nroot(y)) t[z].ch[y == t[z].ch[1]] = x;
    t[x].fa = z;
    t[y].ch[k] = t[x].ch[k ^ 1];
    t[t[x].ch[k ^ 1]].fa = y;
    t[x].ch[k ^ 1] = y;
    t[y].fa = x;
    pushup(y);
    pushup(x);
}

void Splay(int x)
{
    int u = x;
    sta[++top] = u;
    while (nroot(u)) sta[++top] = u = t[u].fa;
    while (top) pushdown(sta[top--]);
    while (nroot(x))
    {
        int y = t[x].fa;
        int z = t[y].fa;
        if (nroot(y)) (x == t[y].ch[1]) ^ (y == t[z].ch[1]) ? rotate(x) : rotate(y);
        rotate(x);
    }
}

void access(int x)
{
    for (int y = 0; x; x = t[y = x].fa)
    {
        Splay(x);
        t[x].vir ^= t[t[x].ch[1]].val;
        t[x].vir ^= t[t[x].ch[1] = y].val;
        pushup(x);
    }
}

void makeroot(int x)
{
    access(x);
    Splay(x);
    reverse(x);
}

void link(int x, int y)
```

```
{
    makeroot(x);
    access(y);
    Splay(y);
    t[x].fa = y;
    t[y].vir ^= t[x].val;
}

void cut(int x, int y)
{
    makeroot(x);
    access(y);
    Splay(y);
    t[x].fa = t[y].ch[0] = 0;
    pushup(y);
}

void reverse(int x)
{
    swap(t[x].ch[0], t[x].ch[1]);
    t[x].rev ^= 1;
}

void pushup(int x)
{
    t[x].val = t[x].self ^ t[x].vir ^ t[t[x].ch[0]].val ^ t[t[x].ch[1]].val;
}

void pushdown(int x)
{
    if (t[x].rev)
    {
        reverse(t[x].ch[0]);
        reverse(t[x].ch[1]);
        t[x].rev = false;
    }
}
```

3.4.3. Bài toán: Số màu phân biệt

CF1172E: https://codeforces.com/contest/1172/problem/E

Cho cây n nút, các nút đánh số 1,2, … , n. Ban đầu, nút i có màu c_i.

Trọng số mỗi đường đi đơn (không lặp đỉnh) được tính bằng số lượng màu phân biệt có trong đường đi đơn đó.

Đưa ra tổng trọng số của tất cả các đường đi đơn của cây ban đầu và sau mỗi lượt tô lại màu của một nút, các lượt tô màu thực hiện theo trình tự thời gian.

Quy ước: nếu $u \neq v$ thì $P(u,v) \neq P(v,u)$

Dữ liệu

- Dòng 1: n, m $(n, m \leq 4 \cdot 10^5)$
- Dòng 2: $c_1, c_2, \dots, c_n$ $(1 \leq c_i \leq n)$
- Dòng 3 … $m+2$: $vertex\ newcolor$

Kết quả

- Dòng 1: $m+1$ số nguyên kết quả

Thuật toán (tác giả: https://codeforces.com/profile/ODT)

Với mỗi màu c, ta tìm cách duy trì số lượng đường đi đơn không chứa nút nào có màu c.

Từ đó, với mỗi màu c, ta có thể xác định số lượng đường đi đơn có chứa nút màu c. Lấy tổng đối với tất cả các màu thu được kết quả.

Xét một màu c, giả sử xóa tất cả các nút màu có màu này, thu được các mảnh (cây con), số lượng đường đi đơn không chứa màu c sẽ là $\sum_T |T|^2$.

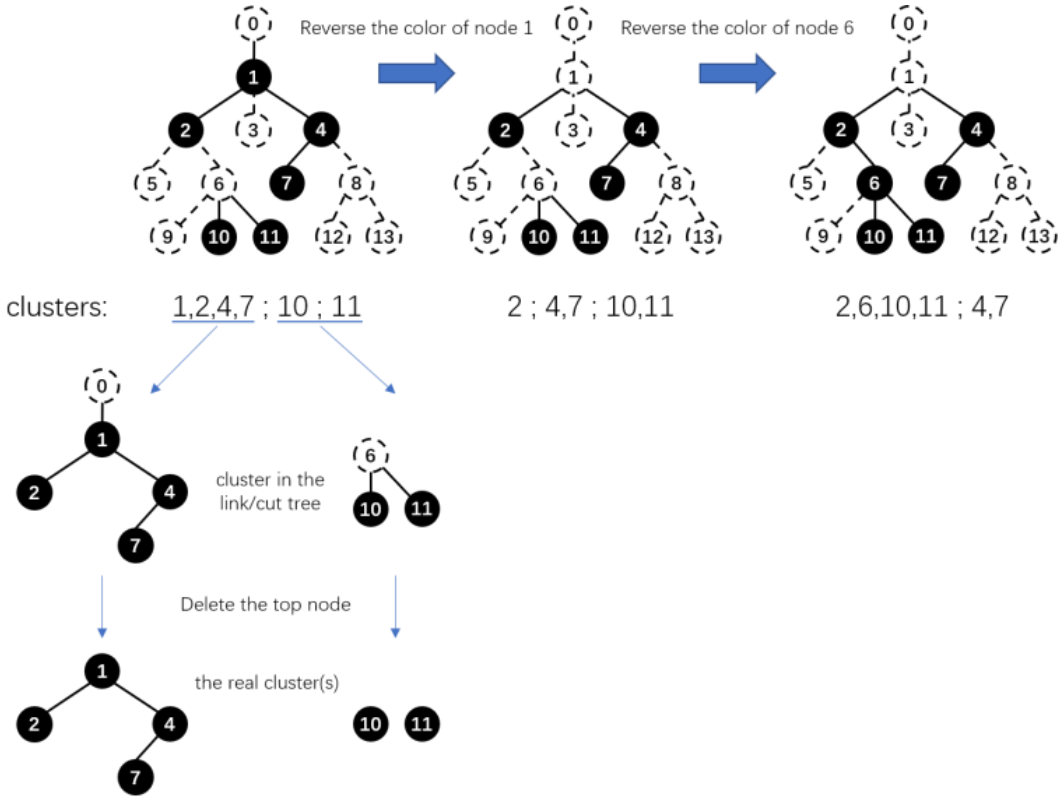
Với màu cụ thể c, có thể coi màu các nút là trắng hay đen ứng với khác c hay bằng c. Ta có bài toán với cây các nút màu trắng:

- Đảo màu một nút (trắng ↔ đen)
- Tính $\sum_T |T|^2$

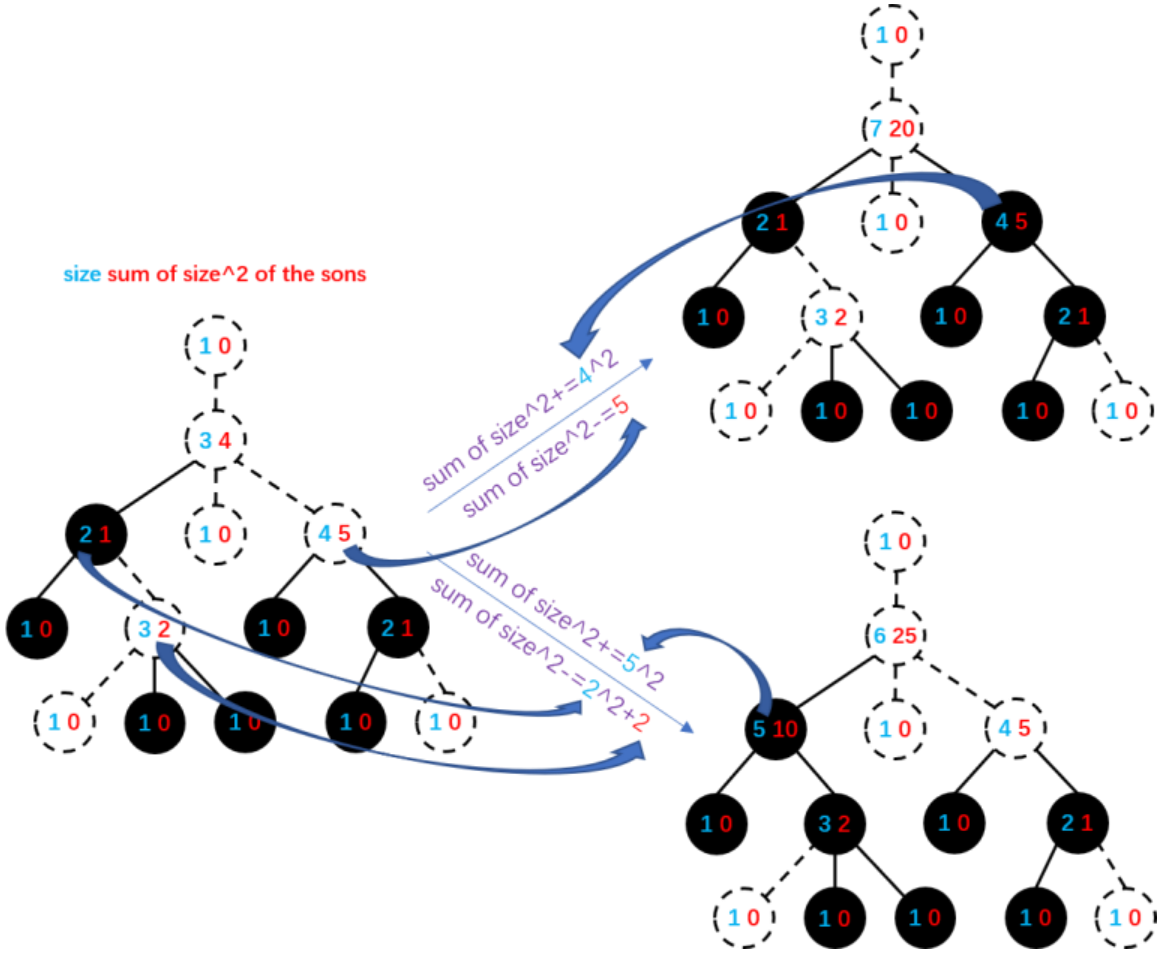
Quản lí các cây con bằng link-cut tree.

- Giữ siz là kích thước mỗi cây con, $siz2$ là tổng bình phương kích thước của mỗi nút con.
- Mỗi khi một nút đổi màu, link/cut nó với nút cha, cập nhật $\sum_T |T|^2$ trong khi link/cut.

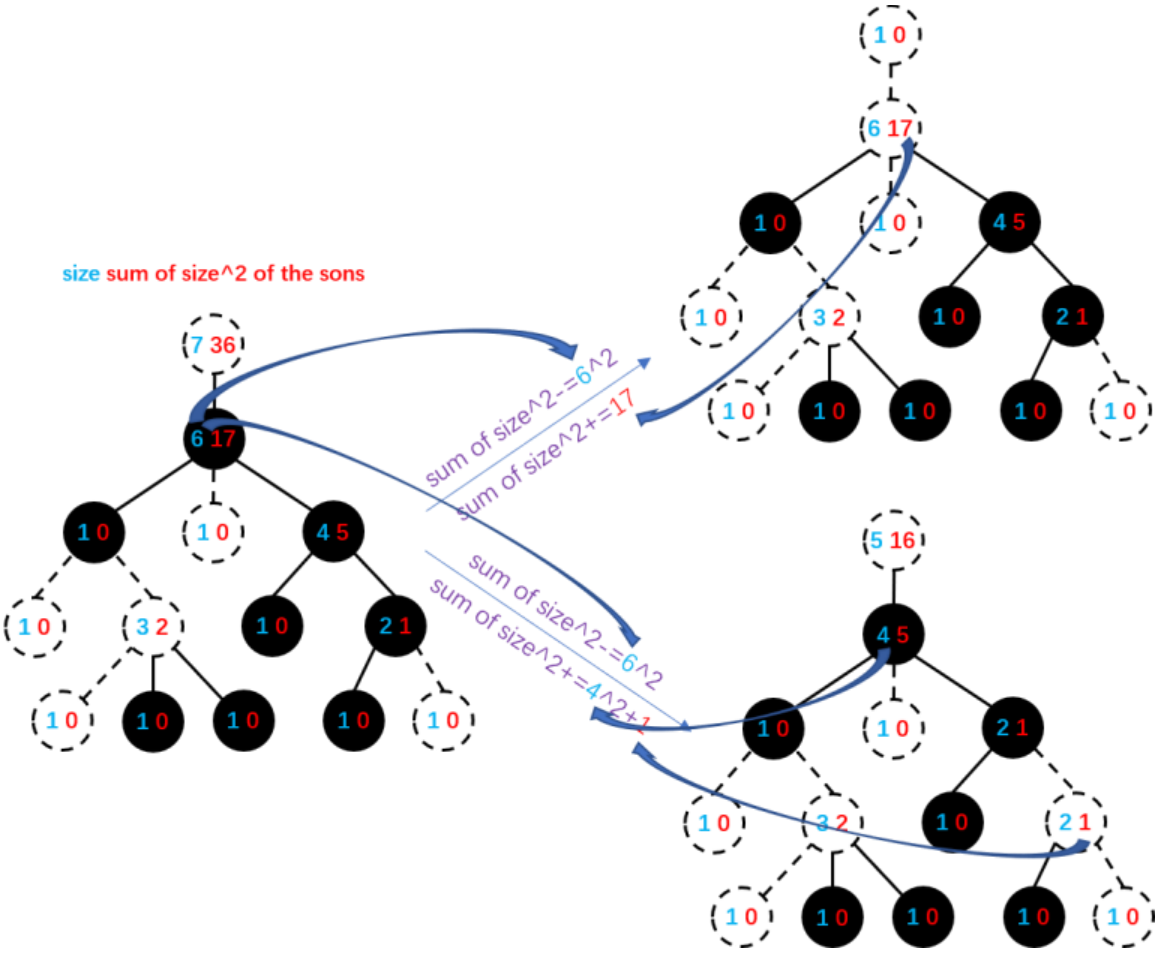
Hình dưới minh họa biến đổi cây khi đảo màu một nút



Minh họa thao tác *link*



Minh họa thao tác *cut*



Sources

[https://codeforces.com/profile/ODT]

```
#include <algorithm>
#include <cctype>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;

const int N = 400010;

struct Node
{
    int fa, ch[2], siz, sizi;
    ll siz2i;
    ll siz2() { return (ll) siz * siz; }
} t[N];

bool nroot(int x);
void rotate(int x);
void Splay(int x);
void access(int x);
int findroot(int x);
void link(int x);
void cut(int x);
void pushup(int x);

void add(int u, int v);
void dfs(int u);

int head[N], nxt[N << 1], to[N << 1], cnt;
int n, m, c[N], f[N];
ll ans, delta[N];
bool bw[N];
vector<int> mod[N][2];

int main()
{
    int i, j, u, v;
    ll last;

    scanf("%d%d", &n, &m);

    for (i = 1; i <= n; ++i)
    {
        scanf("%d", c + i);
        mod[c[i]][0].push_back(i);
        mod[c[i]][1].push_back(0);
    }

    for (i = 1; i <= n + 1; ++i) t[i].siz = 1;

    for (i = 1; i < n; ++i)
    {
        scanf("%d%d", &u, &v);
        add(u, v);
        add(v, u);
    }
```

```
    for (i = 1; i <= m; ++i)
    {
        scanf("%d%d", &u, &v);
        mod[c[u]][0].push_back(u);
        mod[c[u]][1].push_back(i);
        c[u] = v;
        mod[v][0].push_back(u);
        mod[v][1].push_back(i);
    }

    f[1] = n + 1;
    dfs(1);

    for (i = 1; i <= n; ++i) link(i);

    for (i = 1; i <= n; ++i)
    {
        if (!mod[i][0].size())
        {
            delta[0] += (ll)n * n;
            continue;
        }
        if (mod[i][1][0])
        {
            delta[0] += (ll)n * n;
            last = (ll)n * n;
        } else
            last = 0;
        for (j = 0; j < mod[i][0].size(); ++j)
        {
            u = mod[i][0][j];
            if (bw[u] ^= 1)
                cut(u);
            else
                link(u);
            if (j == mod[i][0].size() - 1 || mod[i][1][j + 1] != mod[i][1][j])
            {
                delta[mod[i][1][j]] += ans - last;
                last = ans;
            }
        }
        for (j = mod[i][0].size() - 1; ~j; --j)
        {
            u = mod[i][0][j];
            if (bw[u] ^= 1)
                cut(u);
            else
                link(u);
        }
    }

    ans = (ll) n * n * n;
    for (i = 0; i <= m; ++i)
    {
        ans -= delta[i];
        printf("%I64d ", ans);
    }

    return 0;
}

bool nroot(int x) { return x == t[t[x].fa].ch[0] || x == t[t[x].fa].ch[1]; }
```

```
void rotate(int x)
{
    int y = t[x].fa;
    int z = t[y].fa;
    int k = x == t[y].ch[1];
    if (nroot(y)) t[z].ch[y == t[z].ch[1]] = x;
    t[x].fa = z;
    t[y].ch[k] = t[x].ch[k ^ 1];
    t[t[x].ch[k ^ 1]].fa = y;
    t[x].ch[k ^ 1] = y;
    t[y].fa = x;
    pushup(y);
    pushup(x);
}

void Splay(int x)
{
    while (nroot(x))
    {
        int y = t[x].fa;
        int z = t[y].fa;
        if (nroot(y)) (x == t[y].ch[1]) ^ (y == t[z].ch[1]) ? rotate(x) : rotate(y);
        rotate(x);
    }
}

void access(int x)
{
    for (int y = 0; x; x = t[y = x].fa)
    {
        Splay(x);
        t[x].sizi += t[t[x].ch[1]].siz;
        t[x].sizi -= t[y].siz;
        t[x].siz2i += t[t[x].ch[1]].siz2();
        t[x].siz2i -= t[y].siz2();
        t[x].ch[1] = y;
        pushup(x);
    }
}

int findroot(int x)
{
    access(x);
    Splay(x);
    while (t[x].ch[0]) x = t[x].ch[0];
    Splay(x);
    return x;
}

void link(int x)
{
    int y = f[x];
    Splay(x);
    ans -= t[x].siz2i + t[t[x].ch[1]].siz2();
    int z = findroot(y);
    access(x);
    Splay(z);
    ans -= t[t[z].ch[1]].siz2();
    t[x].fa = y;
    Splay(y);
    t[y].sizi += t[x].siz;
    t[y].siz2i += t[x].siz2();
```

```
    pushup(y);
    access(x);
    Splay(z);
    ans += t[t[z].ch[1]].siz2();
}

void cut(int x)
{
    int y = f[x];
    access(x);
    ans += t[x].siz2i;
    int z = findroot(y);
    access(x);
    Splay(z);
    ans -= t[t[z].ch[1]].siz2();
    Splay(x);
    t[x].ch[0] = t[t[x].ch[0]].fa = 0;
    pushup(x);
    Splay(z);
    ans += t[t[z].ch[1]].siz2();
}

void pushup(int x)
{
    t[x].siz = t[t[x].ch[0]].siz + t[t[x].ch[1]].siz + t[x].sizi + 1;
}

void add(int u, int v)
{
    nxt[++cnt] = head[u];
    head[u] = cnt;
    to[cnt] = v;
}

void dfs(int u)
{
    int i, v;
    for (i = head[u]; i; i = nxt[i])
    {
        v = to[i];
        if (v != f[u])
        {
            f[v] = u;
            dfs(v);
        }
    }
}
```

4. Bài tập luyện tập

4.1. Liên thông động

SPOJ-DYNACON1: https://www.spoj.com/problems/DYNACON1/

Quản lí tính liên thông của đồ thị n đỉnh 1,2, … , n với m truy vấn tuần tự thuộc một trong ba loại

- $< add\ u\ v >$: thêm cạnh (u, v), đảm bảo trước đó chưa có cạnh này
- $< rem\ u\ v >$: xóa cạnh (u, v), đảm bảo trước đó đã có cạnh này
- $< cnn\ u\ v >$: hỏi u, v có liên thông.

Giới hạn: $n, m \leq 10^5$.

Nhận xét

Bài toán chỉ đòi hỏi các thao tác cơ bản của link-cut tree với bộ test tương đối yếu và time limit khá rộng, có thể sử dụng để thử nghiệm nhiều cấu trúc dữ liệu khác nhau với các cải tiến khác nhau.

4.2. Bẻ ghi

CF1344E: https://codeforces.com/problemset/problem/1344/E

Cho các ga và đường tàu lập thành một cấu trúc cây với nút gốc là nút 1. Các đường tàu là một chiều, tại các ga có thể bẻ ghi để thay đổi hướng đến của đoàn tàu ra khỏi ga này (hướng đi mặc định ban đầu của mỗi ga u là cạnh $(u, ?)$ xuất hiện sau cùng trong input).

Có m đoàn tàu, đoàn tàu i bắt đầu từ ga 1 ở thời điểm t_i, đích đến là ga s_i, các giá trị t_i phân biệt, tăng. Thời điểm bẻ ghi là thời điểm nguyên, mỗi lần bẻ ghi tại không quá một ga.

Ngay khi một đoàn tàu đi vào một hướng không thể đến đích, nó sẽ nổ tung. Cần xác định thời điểm này muộn nhất có thể và số lần bẻ ghi ít nhất đạt được điều đó.

Dữ liệu

- Dòng 1: n, m $(0 < n, m \leq 10^5)$
- Dòng 2 ... n: u, v, d $(0 < d \leq 10^9)$
- Dòng $n+1$ ... $n+1$: s_i, t_i $(0 < t_i \leq 10^9)$

Kết quả

- Dòng 1: T_{max}, C_{min}.

Thuật toán

Nhận xét: một tàu không thể vượt qua một tàu xuất phát trước nó. Do đó có thể xét từng tàu độc lập. Với một tàu i, lần theo đường từ 1 đến s_i, có thể phải bẻ ghi tại một số ga. Xét một lần bẻ ghi ở ga nào đó, quãng thời gian cho phép thực hiện hành động này là $(L; R]$ với R là thời điểm tàu i đến ga đó, L là thời điểm gần nhất ga đó được một tàu đi qua theo một hướng khác.

Giả sử ghi nhận được k sự kiện bẻ ghi với đoạn thời gian tương ứng. Quản lí các sự kiện này bằng một hàng đợi ưu tiên, luôn bẻ ghi ở thời điểm sớm nhất có thể. Lặp lại đến khi hết sự kiện hoặc bị quá hạn dẫn đến nổ tàu. Công đoạn này tốn chi phí $O(k \log n)$.

Nhận thấy việc bảo đảm di chuyển từ 1 đến mỗi s_i tương đương với một thao tác $access(s_i)$ trong link-cut tree. Link-cut tree có chi phí khấu trừ trung bình $O(\log n)$ cho mỗi thao tác, chứng tỏ số thao tác bẻ ghi $k = O(n + m \log n)$. Nghĩa là sau khi có danh sách sự kiện, việc giải bài toán tốn chi phí $O(n \log n + n \log^2 n)$.

Phần còn lại là xác định đoạn thời gian cho mỗi sự kiện. Ta sẽ làm điều này bằng link-cut tree, việc bẻ ghi từ (u, v_1) thành (u, v_2) chính xác là $cut(v_1)$ rồi $link(u, v_2)$.

Source

```
#include <bits/stdc++.h>
using namespace std;
#define FOR(i,a,b) for (int i=(a);i<=(b);i++)
typedef long long LL;
typedef pair <int, int> II;
const int N = 1 + 1e5;
int n, m;
vector <II> e[N];
LL dep[N];
```

```
void dfs(int x, LL d) {
    dep[x] = d;
    for(auto E : e[x])
        dfs(E.first, d + E.second);
}
vector <pair <LL, LL> > v;
namespace lct {
int fa[N], son[N][2];
LL val[N], tag[N];
int isroot(int x) {
    return son[fa[x]][0] != x && son[fa[x]][1] != x;
}
int wson(int x) {
    return son[fa[x]][1] == x;
}
void rotate(int x) {
    int y = fa[x], z = fa[y], L = wson(x), R = L ^ 1;
    if(!isroot(y))
        son[z][wson(y)] = x;
    fa[x] = z, fa[y] = x, fa[son[x][R]] = y;
    son[y][L] = son[x][R], son[x][R] = y;
}
void pushson(int x, LL v) {
    val[x] = max(val[x], v);
    tag[x] = max(tag[x], v);
}
void pushdown(int x) {
    if(tag[x]) {
        pushson(son[x][0], tag[x]);
        pushson(son[x][1], tag[x]);
        tag[x] = 0;
    }
}
void pushadd(int x) {
    if(!isroot(x))
        pushadd(fa[x]);
    pushdown(x);
}
void splay(int x) {
    pushadd(x);
    for(int y = fa[x]; !isroot(x); rotate(x), y = fa[x])
        if(!isroot(y))
            rotate(wson(x) == wson(y) ? y : x);
}
void access(int x, LL start_time) {
    int _x = x;
    int t = 0;
    while(x) {
        splay(x);
        if(x != _x) {
            v.push_back({val[x] == 0 ? 1 : val[x] + dep[x] + 1, start_time + dep[x]});
            son[x][1] = t;
        }
        t = x;
        x = fa[x];
    }
    x = _x;
    splay(x);
    if(son[x][0])
        pushson(son[x][0], start_time);
}
}
void answer(LL res) {
```

```
    cout << res << ' ';
    res == -1 ? res = 1e18 : res;
    int cnt = 0;
    for(auto i : v)
        cnt += i.second < res;
    cout << cnt << '\n';
    exit(0);
}
int main() {
    cin.tie(0)->sync_with_stdio(0);
    cin >> n >> m;
    FOR(i, 1, n - 1) {
        int x, y, z;
        cin >> x >> y >> z;
        e[x].push_back({y, z});
        lct::fa[y] = x, lct::son[x][1] = y;
    }
    dfs(1, 0);
    FOR(i, 1, m) {
        int s, t;
        cin >> s >> t;
        lct::access(s, t);
    }
    sort(v.begin(), v.end());
    priority_queue <LL, vector <LL>, greater <LL> > q;
    LL t = 1;
    for(auto p : v) {
        while(!q.empty() && t < p.first) {
            if(q.top() < t)
                answer(q.top());
            q.pop();
            t++;
        }
        t = max(t, p.first);
        q.push(p.second);
    }
    while(!q.empty()) {
        if(q.top() < t)
            answer(q.top());
        q.pop();
        t++;
    }
    answer(-1);
    return 0;
}
```

4.3. Một số bài tập khác

Code name	Link
GERALD07	https://www.codechef.com/problems/GERALD07
RGY	https://www.codechef.com/problems/RGY
QTREE6	https://www.codechef.com/problems/QTREE6 http://www.spoj.com/problems/QTREE6/
ARCRT	https://www.codechef.com/problems/ARCRT
MCO16406	https://www.codechef.com/problems/MCO16406
RIVER	https://www.codechef.com/problems/RIVER
DYNALCA	http://www.spoj.com/problems/DYNALCA/
Timus: 1553	http://acm.timus.ru/problem.aspx?space=1&num=1553
SPOJ: OTOCI	http://www.spoj.com/problems/OTOCI/
Codeforces: 117E	https://codeforces.com/problemset/problem/117/E

UVA: 11998	https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=66&page=show_problem&problem=3149