



# ATHENA X – MỤC LỤC

<b>ĐØ THI</b>	6
<b>Duyệt theo chiều sâu và thời điểm vào/ra</b>	6
Bài toán 1	6
Chương trình	7
Bài toán 2	9
<b>Sắp xếp tó pô</b>	11
Bài toán	11
Giải thuật	11
Chương trình 1 - DFS	12
Chương trình 2 - BFS	14
<b>Chuyển đổi dạng biểu diễn cây</b>	15
Bài toán	16
Giải thuật:	17
Chương trình	18
<b>Kiểm tra tồn tại chu trình</b>	19
Bài toán	19
Giải thuật	20
Chương trình	20
<b>Thành phần liên thông</b>	22
Bài toán	22
Giải thuật	23
Chương trình	24
<b>Liên thông mạnh và cõ đặc đồ thị</b>	25
Giải thuật Kosaraju – Sharir	25
Định lý	26
Giải thuật	26
Chương trình	27
<b>Tìm kiếm cầu trên đồ thị vô hướng</b>	29
Giải thuật tìm cầu offline	29
Giải thuật tìm cầu online	31

<b>Tìm điểm khớp của đồ thị</b>	37
Giải thuật	37
Chương trình	38
<b>HÀNG ĐƠI ƯU TIÊN VÀ GIẢI THUẬT DIJKSTRA</b>	39
BÀI TOÁN	39
GIẢI THUẬT	40
KỸ THUẬT CÀI ĐẶT HIỆU QUẢ CAO VỚI ĐỒ THỊ MA TRẬN THUA	40
Chương trình sử dụng cấu trúc dữ liệu kiểu <i>Tập hợp</i>	43
<b>Giải thuật Floyd – Warshall</b>	45
Bài toán	45
Giải thuật	46
Chương trình	47
<b>Cây khung nhỏ nhất – Giải thuật Prim</b>	48
Bài toán	48
Giải thuật	49
Chương trình	50
Giải thuật độ phức tạp $O(m\log n)$	51
Chương trình	52
Tính chất cây khung nhỏ nhất và giải thuật	53
<b>Tìm cha chung gần nhất</b>	54
Bài toán	54
Giải thuật	55
Chương trình	58
Giải thuật Farach-Colton – Bender	59
Giải thuật	59
Chương trình	60
<b>CÂU TRÚC CÂY</b>	63
<b>Cây Fenwick</b>	63
Bài toán 1.	63
Giải thuật	64
<i>Xử lý offline</i>	64

<b>Chương trình</b>	65
<b>Xử lý online</b>	65
<b>Chương trình</b>	66
<b>Bài toán 2.</b>	66
<b>Chương trình</b>	67
<b>Mở rộng nhiều chiều</b>	69
<b>Cây quản lý đoạn</b>	71
<b>Cấu trúc cây</b>	71
<b>Tổ chức dữ liệu</b>	71
<b>Các phép xử lý</b>	72
<b>Khởi tạo cây</b>	72
<b>Cập nhật</b>	73
<b>Truy vấn kết quả trên đoạn [u, v]</b>	75
<b>Bài toán ứng dụng minh họa</b>	76
<b>Chương trình</b>	77
<b>TREAP</b>	79
<b>Các phép xử lý với Treap</b>	80
<b>Các phép xử lý cơ sở</b>	80
<b>Khai báo cấu trúc phần tử của Treap</b>	80
<b>Merge (Hợp nhất Treaps)</b>	80
<b>split (Tách cây)</b>	81
<b>Tìm kiếm trên Treap</b>	82
<b>Bổ sung phần tử mới</b>	82
<b>Xóa phần tử của cây</b>	82
<b>Tìm phần tử</b>	83
<b>Truy vấn theo nhóm phần tử</b>	84
<b>Xử lý truy vấn trên toàn bộ cây</b>	84
<b>Xử lý truy vấn trên đoạn</b>	86
<b>Cập nhật theo nhóm phần tử</b>	86
<b>Cập nhật trên toàn cây</b>	86
<b>Cập nhật trên đoạn</b>	88
<b>Phạm vi ứng dụng và hạn chế của cấu trúc Treap</b>	88

<i>Úng dụng</i>	88	
<i>Hạn chế</i>	89	
<b>Treap với khóa ẩn</b>	90	
<i>Ý tưởng chính</i>	90	
<i>Khóa X</i>	90	
<i>Đại lượng thay thế C</i>	90	
<i>Các phép xử lý cấu trúc</i>	91	
<i>Hàm split</i>	91	
<i>Hàm merge</i>	91	
<i>Đảm bảo tính nhất quán của c</i>	92	
<i>Úng dụng Treap với khóa ẩn</i>	92	
<i>Tổ chức Treap với khóa ẩn trên C++</i>	92	
<i>Bài tập ứng dụng</i>	94	
<b>VU04. HAI BẢN ĐỒ</b>	<i>Tên chương trình: TWOMAPS.CPP</i>	94
<i>Hệ thống các tập không giao nhau</i>	104	
<i>Tổ chức dữ liệu</i>	104	
<i>Giải pháp heuristic nén đường đi</i>	105	
<i>Giải pháp heuristic hợp nhất theo bậc của cây</i>	107	
<i>Hợp nhất các giải pháp heuristic – nén đường đi kết hợp với chọn bậc</i>	109	
<i>Ứng dụng của hệ thống các tập không giao nhau</i>	110	
<i>Quản lý thành phần liên thông của đồ thị</i>	110	
<i>Tìm thành phần liên thông trên ảnh</i>	110	
<i>Hỗ trợ lưu thông tin bổ sung với mỗi tập hợp</i>	111	
<i>Nén thông tin rời rạc trên đoạn thẳng</i>	111	
<i>Quản lý khoảng cách tới đỉnh đại diện</i>	112	
<i>Giải thuật tìm min trên một đoạn (<i>Range Minimum Query – RMQ</i>)</i>	115	
<i>Tìm cha chung nhỏ nhất trong cây (<i>Lowest Common Ancestor – LCA</i>)</i>	115	
<i>Hợp nhất các cấu trúc dữ liệu khác nhau</i>	118	
<i>Tìm cầu của đồ thị trong chế độ online với độ phức tạp O(<math>\alpha(n)</math>)</i>	119	
<i>Sơ lược về lịch sử DSU</i>	120	
<i>Bài tập</i>	121	
<b>VV07. BỘ TRÍ DỮ LIỆU</b>	<i>Tên chương trình: DATA.CPP</i>	121

<i>Rừng số tìm kiếm nhanh</i> .....	125
<i>Rừng cây</i> .....	125
<i>Ký hiệu</i> .....	125
<i>Các bước xây dựng</i> .....	125
<i>Ưu và nhược điểm của cấu trúc rừng</i> .....	126
<i>Rừng số</i> .....	126
<i>Rừng số tìm kiếm nhanh (x-fast-trie)</i> .....	127
<i>Rừng số tìm kiếm siêu nhanh (y-fast-trie)</i> .....	130
<i>Vụn đồng Fibonacci heap</i> .....	132
<i>Cấu trúc của Fibonacci heap</i> .....	132
<i>Các phép xử lý</i> .....	133
<i>Các phép biến đổi trong Fibonacci heap</i> .....	135
<i>Phạm vi sử dụng của Fibonacci heap</i> .....	140

# ĐÓ THỊ

## Duyệt theo chiều sâu và thời điểm vào/ra

Tìm kiếm theo chiều sâu (*depth-first search – dfs*) là tìm đường đi có thứ tự từ điển nhỏ nhất từ gốc qua tất cả các đỉnh của đồ thị, một đỉnh có thể gặp nhiều lần trên đường đi.

Với đồ thị  $n$  đỉnh và  $m$  cạnh, giải thuật duyệt theo chiều sâu có độ phức tạp  $O(n+m)$ .

Dfs là cơ sở để giải quyết các vấn đề:

- ✚ Tìm đường đi bất kỳ trên đồ thị,
- ✚ Tìm đường đi có thứ tự từ điển nhỏ nhất từ gốc tới một đỉnh,
- ✚ Kiểm tra với chi phí  $O(1)$  một đỉnh có phải là cha của đỉnh khác hay không,
- ✚ Tìm cha chung gần nhất (*Least common Ancestor – LCA*),
- ✚ Sắp xếp tô pô,
- ✚ Kiểm tra tồn tại chu trình và tìm chu trình,
- ✚ Tìm thành phần liên thông mạnh,
- ✚ Tìm cầu của đồ thị.

Tham số dẫn xuất quan trọng nhận được khi duyệt theo chiều sâu là thời điểm vào và thời điểm ra của mỗi đỉnh trong quá trình duyệt. Đây là các tham số cơ sở để giải quyết các vấn đề đã nêu.

*Thời điểm vào* của một đỉnh là *thời điểm lần đầu tiên tới thăm đỉnh* đó khi xuất phát từ gốc. *Thời điểm ra* là *thời điểm lần cuối cùng thăm đỉnh* trong quá trình duyệt.

Để tính thời điểm vào/ra ở mỗi đỉnh ta cần tổ chức một đồng hồ `dfs_timer` với giá trị đầu là 0. Đồ thị được lưu trữ dưới dạng danh sách đỉnh kè: vector `g[v]` chứa các đỉnh kè với đỉnh `v`. Ngoài ra, ta còn cần phải đánh dấu các đỉnh đã thăm để tránh lặp trong quá trình duyệt.

### Bài toán 1

Xét đồ thị  $n$  đỉnh, đánh số từ 1 đến  $n$ , gốc của đồ thị là đỉnh 1. Đồ thị có  $m$  cạnh, cạnh  $j$  nối trực tiếp 2 đỉnh  $x_j$  và  $y_j$ ,  $j = 1 \dots m$ . Không có cạnh nào nối một đỉnh với chính nó. Xuất phát từ thời điểm 0 ở đỉnh gốc người ta duyệt đồ thị theo chiều sâu. Từ một đỉnh, cứ sau mỗi đơn vị thời gian giải thuật lại chuyển sang xử lý đỉnh kè.

Hãy xác định thời điểm vào và ra của mỗi đỉnh.

**Dữ liệu:** Vào từ file văn bản G\_IN\_OUT.INP:

- ✚ Dòng đầu tiên chứa 2 số nguyên  $n$  và  $m$  ( $1 \leq n \leq 10^5$ ,  $1 \leq m \leq \min\{10^5, \frac{n \times (n-1)}{2}\}$ ),
- ✚ Dòng thứ  $j$  trong  $m$  dòng tiếp theo chứa 2 số nguyên  $x_j$ ,  $y_j$  ( $1 \leq x_j, y_j \leq n$ ,  $x_j \neq y_j$ ). Không có 2 cạnh nào trùng nhau.

**Kết quả:** Đưa ra file văn bản G\_IN\_OUT.OUT:

- Dòng đầu tiên chứa thông báo “G . . . in/out:”
- Dòng thứ 2 chứa n số nguyên, số thứ i xác định thời gian vào của đỉnh i,  $i = 1 \div n$ ,
- Dòng thứ 3 chứa n số nguyên, số thứ i xác định thời gian ra của đỉnh i,  $i = 1 \div n$ .

**Ví dụ:**

G_IN_OUT.INP	G_IN_OUT.OUT
5 8	G . . . in/out:
1 3	0 4 1 2 3
1 4	9 5 8 7 6
1 5	
5 2	
5 4	
3 4	
3 2	
2 4	

**Tổ chức dữ liệu:** (Chương trình phục vụ minh họa giải thuật, không đặt vấn đề tối ưu hóa việc sử dụng bộ nhớ)

- Các mảng động `vector <int>` g[N] lưu trữ danh sách đỉnh kề,
- Mảng `vector<bool>` used(N, `false`) đánh dấu đỉnh đã xử lý,
- Các mảng `vector<int>` time\_in(N), time\_out(N) lưu trữ kết quả.

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "G_in_out."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100000;

int n,m;
vector<bool> used(N, false);
vector<int> time_in(N), time_out(N);
vector <int> g[N];
int dfs_timer = 0;

void wtr()
{
    fo<<"Graph... in/out:"<<endl;
    for(int i=0;i<n;++i) fo<<time_in[i]<<' ' ; fo<<endl;
```

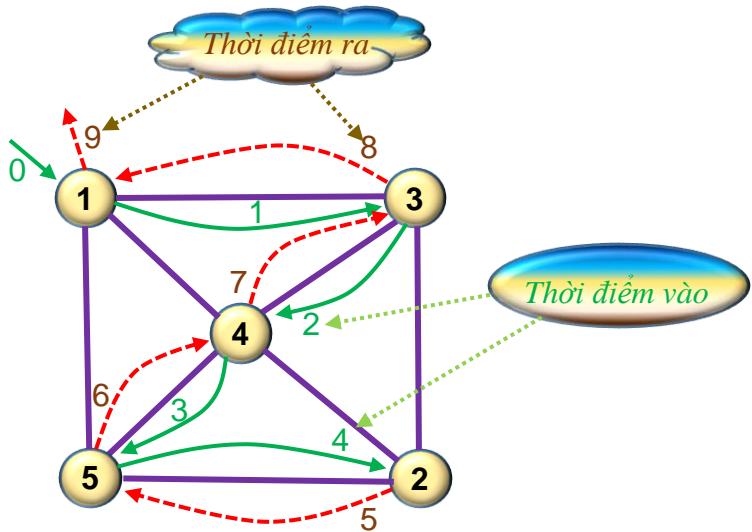
```

        for (int i=0; i<n; ++i) fo<<time_out[i]<<' '; fo<<endl;
    }

void dfs (int v)
{
    time_in[v] = dfs_timer++;
    used[v] = true;
    for (auto& i:g[v])
        if (!used[i]) dfs (i);
    used[v] = true;
    time_out[v] = dfs_timer++;
}

int main()
{
    fi>>n>>m;
    for (int i=0; i<m; ++i)
    {
        int x, y;
        fi>>x>>y;
        --x, --y;
        g[x].push_back(y);
        g[y].push_back(x);
    }
    dfs(0); wtr();
}

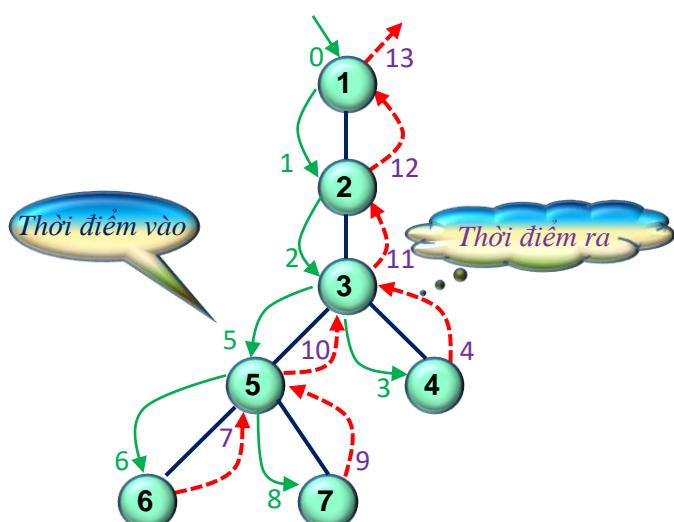
```



Kết quả thực hiện với bộ dữ liệu:

7 6  
1 2  
2 3  
3 4  
3 5  
5 6  
5 7

Graph... in/out:  
0 1 2 3 5 6 8  
13 12 11 4 10 7 9



Trong nhiều trường hợp, đặc biệt là các bài toán xử lý cây, thời điểm vào đóng vai trò quan trọng, còn thời điểm ra được tính theo việc kết thúc xử lý, ra khỏi quá trình xử lý đê quy đối với nút. Các trình tự vào/ra vẫn giữ nguyên, nhưng thời điểm ra có thể là chung đối với một số nút. Cách ghi nhận này cung cấp thêm thông tin về quá trình xử lý!

### Bài toán 2

Xét cây có  $n$  đỉnh. Các đỉnh được đánh số từ 1 đến  $n$ . Gốc của cây là đỉnh 1. Cấu trúc của cây được cho dưới dạng  $n-1$  cặp đỉnh kề cạnh  $(x_i, y_i)$ ,  $i = 1 \dots n-1$ . Thời gian xử lý thông tin ở mỗi đỉnh là 1, thời gian thoát khỏi đê quy là 0. Bắt đầu từ thời điểm 0 người ta duyệt cây theo chiều sâu.

Hãy xác định thời điểm vào và ra của mỗi đỉnh.

*Dữ liệu:* Vào từ file văn bản T\_IN\_OUT.INP:

- Dòng đầu tiên chứa số nguyên  $n$  ( $1 \leq n \leq 10^5$ ),
- Dòng thứ  $i$  trong  $n-1$  dòng tiếp theo chứa 2 số nguyên  $x_i, y_i$  ( $1 \leq x_j, y_j \leq n$ ,  $x_j \neq y_j$ ). Không có 2 cạnh nào trùng nhau.

*Kết quả:* Đưa ra file văn bản T\_IN\_OUT.OUT:

- Dòng đầu tiên chứa thông báo “**T . . . in/out:**”
- Dòng thứ 2 chứa  $n$  số nguyên, số thứ  $i$  xác định thời gian vào của đỉnh  $i$ ,  $i = 1 \dots n$ ,
- Dòng thứ 3 chứa  $n$  số nguyên, số thứ  $i$  xác định thời gian ra của đỉnh  $i$ ,  $i = 1 \dots n$ .

*Ví dụ:*

G_IN_OUT.INP
7
1 2
2 3
3 4
3 5
5 6
5 7

G_IN_OUT.OUT
<b>T... in/out:</b>
0 1 2 3 4 5 6
7 7 7 4 7 6 7

Với cây, việc xử lý sẽ đơn giản hơn đồ thị tổng quát. Thay đổi cách gọi đê quy một chút, ta không cần đánh dấu một đỉnh đã xét hay chưa.

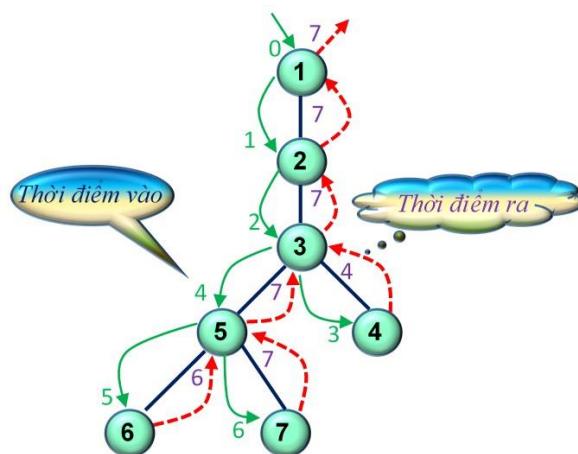
## Chương trình

```
#include <bits/stdc++.h>
#define NAME "T_In_Out."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100000;
vector<int> v[N];
int n;
vector<int> time_in(N), time_out(N);
int dfs_timer = 0;

void wtr()
{
    fo<<"T... in/out:"<<endl;
    for(int i=0;i<n;++i) fo<<time_in[i]<<' '; fo<<endl;
    for(int i=0;i<n;++i) fo<<time_out[i]<<' '; fo<<endl;
}

void dfs(int x, int pr)
{
    time_in[x] = dfs_timer++;
    for (int y : v[x])
    {
        if (y != pr) dfs(y, x);
    }
    time_out[x] = dfs_timer;
}

int main()
{
    fi>>n;
    for(int i=0;i<n-1;++i)
    {
        int x, y;
        fi>>x>>y;
        --x, --y;
        v[x].push_back(y);
        v[y].push_back(x);
    }
    dfs(0, 0); wtr();
}
```



# Sắp xếp tô pô

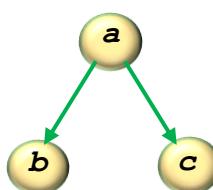
## Bài toán

Cho đồ thị có hướng  $n$  đỉnh và  $m$  cạnh. Yêu cầu **đánh số lại các đỉnh** của đồ thị sao cho mỗi cạnh của đồ thị đều dẫn từ **đỉnh có số nhỏ** tới **đỉnh có số lớn hơn**. Nói cách khác, phải tìm một hoán vị các đỉnh tương ứng với trình tự các cạnh đã chỉ.

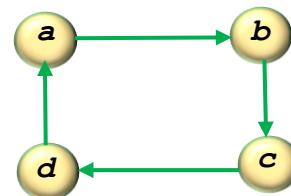
Việc đánh số lại đỉnh được gọi là **sắp xếp tô pô**.

Sắp xếp tô pô có thể cho kết quả không duy nhất, ví dụ đồ thị 3 đỉnh **a, b, c** với 2 cạnh **a → b, a → c**.

Sắp xếp tô pô có thể không tồn tại, ví dụ khi đồ thị chứa chu trình.



Không đơn trị



Không tồn tại

Dạng bài toán thường gặp: Có  $n$  biến, mỗi biến nhận một giá trị nào đó không được biết. Cho một số quan hệ nhỏ hơn giữa một số cặp biến. Hãy xác định các quan hệ đã cho có mâu thuẫn với nhau hay không. Trong trường hợp không mâu thuẫn hãy đưa ra một cách sắp xếp các biến theo thứ tự tăng dần của giá trị.

Giả thiết đồ thị không chứa chu trình. Trong trường hợp đó chắc chắn tồn tại cách đánh số các đỉnh để mỗi đỉnh chỉ có đỉnh con có số lớn hơn đỉnh cha.

## Giải thuật

Thời điểm ra của mỗi đỉnh bao giờ cũng lớn hơn thời điểm ra của các nút con của nó. Thời điểm ra của một đỉnh là thời điểm thoát khỏi xử lý đệ quy của đỉnh đó khi duyệt đồ thị theo chiều sâu. Chính vì vậy ta chỉ cần ghi lại trình tự ra các đỉnh trong quá trình dfs, đảo ngược trình tự đã ghi ta sẽ nhận được kết quả sắp xếp tô pô, tức là nếu đỉnh **v** trong cách đánh số ban đầu được ghi nhận ở vị trí **i** trong mảng **ans** thì giá trị của nó trong sắp xếp tô pô là **ans[i]**.

Độ phức tạp của giải thuật sẽ là  $O(n)$ .

Có thể áp dụng kỹ thuật loang theo chiều rộng để đảm bảo sự thay đổi ít nhất trong cách đánh số lại.

## Chương trình 1 - DFS

```
// Đồ thị có hướng n đỉnh, m cạnh gốc - đỉnh k
#include <bits/stdc++.h>
#define NAME "topo."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100000;

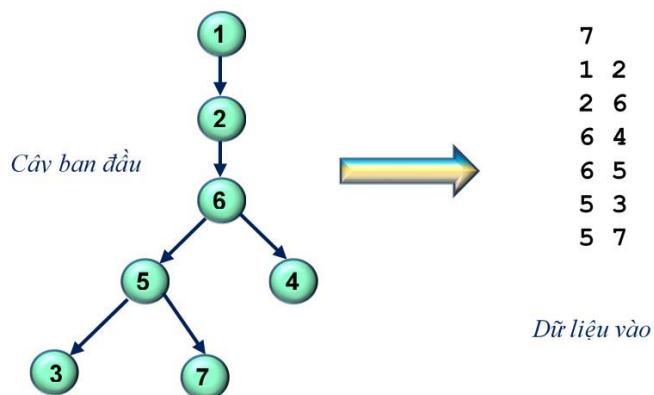
int n,m,k,x,y;
vector<bool> used(N,false);
vector <int> ans,g[N];

void dfs(int v)
{
    used[v]=true;
    for(int i:g[v])
        if(!used[i])dfs(i);
    ans.push_back(v);
}

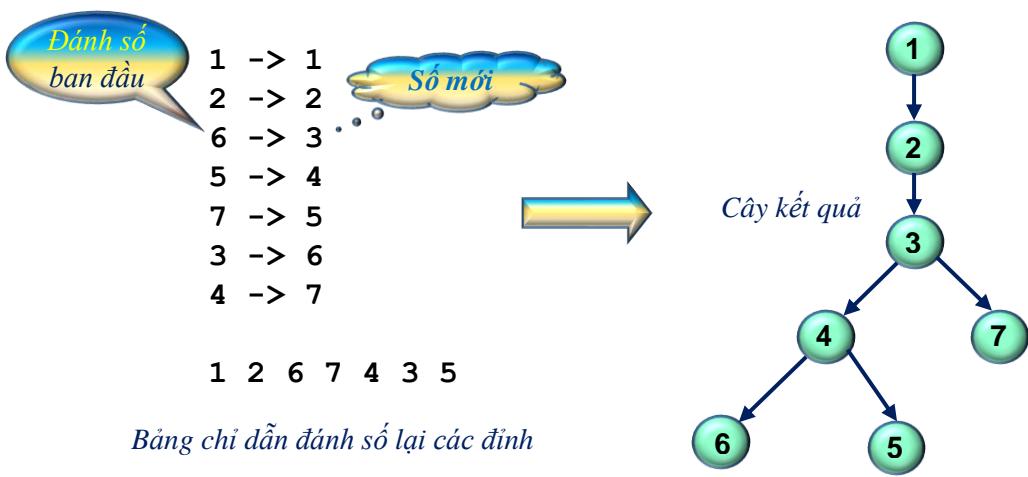
int main()
{
    fi>>n>>m>>k; --k;
    for(int i=0;i<m;++i)
    {
        fi>>x>>y;
        --x; --y; g[x].push_back(y);
    }
    dfs(k);
    for(int i=0;i<n;++i)
        if(!used[i])dfs(i);
    reverse(ans.begin(),ans.end());
}

vector<int>b(n);
for(int i=0;i<n;++i) b[ans[i]]=i;
for(int i=0;i<n;++i) fo<<ans[i]+1<<" -> "<

```



7  
1 2  
2 6  
6 4  
6 5  
5 3  
5 7



## Chương trình 2 – BFS

```
#include <bits/stdc++.h>
#define NAME "topo_G."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100000;

int n,m,k,x,y,v,depth=0;
vector<int> g[N];
vector<bool> used(N, false);
pair<int,int> c[N];
queue<int> q;

void bfs()
{
    while (!q.empty())
    {
        v=q.front(); ++depth;
        q.pop();
        if(c[v].first<depth) c[v].first=depth;
        if(!g[v].empty() && (!used[v])) {for(int &u:g[v]) q.push(u); used[v]=true;}
    }
}

int main()
{
    fi>>n>>m>>k;--k;
    for(int i=0;i<n;++i) c[i].first=0, c[i].second=i;
    for(int i=0;i<m;++i)
    {
        fi>>x>>y;
        --x; --y; g[x].push_back(y);
    }
    v=k; q.push(v); bfs();
    sort(c,c+n);

    vector<int> ans(n);
    for(int i=0;i<n;++i) ans[c[i].second]=i;
    for(int i=0;i<n;++i) fo<<i+1<<" -> "<< ans[i]+1<<'\\n';
    fo<<"\\nTime: "<<clock() / (double) 1000<<" sec";
}
```



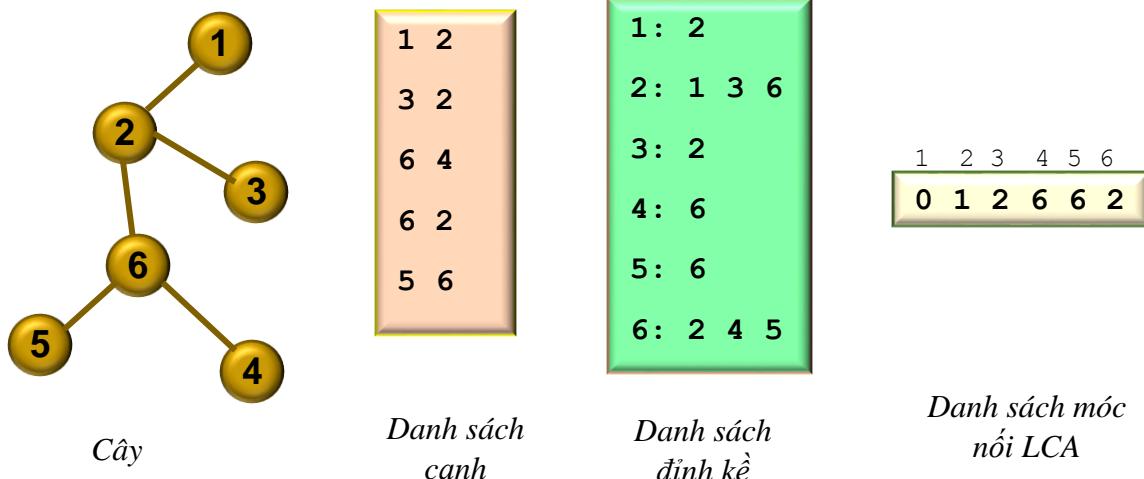
## Chuyển đổi dạng biểu diễn cây

Xét đồ thị vô hướng  $n$  đỉnh, các đỉnh được đánh số từ 1 đến  $n$ . Giữ 2 đỉnh bất kỳ của đồ thị luôn luôn tồn tại một và chỉ một đường đi nối 2 đỉnh đó. Đồ thị có tính chất như vậy được gọi là một cây.

Cây có thể được mô tả như một đồ thị tổng quát bằng danh sách các cặp đỉnh có cạnh nối trực tiếp hoặc danh sách đỉnh kề.

Tuy vậy, do đặc điểm của cây là trừ đỉnh gốc, mỗi đỉnh còn lại đều có đúng một nút cha trực tiếp ta có thể lưu toàn bộ cấu trúc cây bằng một danh sách mốc nối  $n$  phần tử, phần tử thứ  $i$  trong danh sách là con trỏ chỉ tới đỉnh cha trực tiếp của đỉnh  $i$ . Đỉnh gốc có mốc nối với giá trị **NULL** (ví dụ, -1 hoặc 0).

Ví dụ:



Dạng danh sách mốc nối LCA có độ nén thông tin rất cao và hỗ trợ một cách có hiệu quả cho việc giải nhiều lớp bài toán trên cây như tìm nút lá, tìm cha chung gần nhất, phân lớp đỉnh, giải quyết nhiều bài toán thống kê với hiệu quả tương đương cây Fenwick, ...

Danh sách mốc nối LCA có nội dung phụ thuộc vào việc chọn điểm gốc cho cây và đưa cây về dạng tương đương một đồ thị có hướng.

## Bài toán TỔ MỐI

Tổ mối là một công trình ngầm đồ sộ đáng kinh ngạc. Tên mặt đất người ta chỉ thấy một số ụ mối nhô lên. Trên một khu đất rộng người ta quan sát thấy  $n$  ụ mối đánh số từ 1 đến  $n$ . Kết quả khảo sát cho thấy các ụ mối này đều thuộc một tổ mối lớn, có  $n-1$  cặp ụ mối có đường thông nhau trực tiếp giữa 2 ụ mối bất kỳ đều có một và chỉ một đường ngầm nối trực tiếp với nhau hoặc qua các ụ mối trung gian khác.

Để diệt tổ mối này việc đầu tiên người ta cần có bản đồ số tổng thể về quan hệ các ụ mối. Bản đồ số là dãy số  $\mathbf{P} = (\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n)$ , trong đó  $\mathbf{p}_i$  cho biết có đường đi trực tiếp từ ụ mối  $i$  tới ụ mối  $p_i$ . Tồn tại một ụ mối trung tâm nơi có mối chúa sống và ụ mối này có  $\mathbf{p}_i = 0$ . Hiện tại người ta chưa biết đâu là ụ mối trung tâm, nhưng một khi đã có bản đồ số thì việc chỉnh lý là không quá khó.

Hãy đưa ra một bản đồ số phù hợp với các số liệu hiện có.

**Dữ liệu:** Vào từ file văn bản TERMITE.INP:

- + Dòng đầu tiên chứa một số nguyên  $n$  ( $1 \leq n \leq 10^5$ ),
- + Dòng thứ  $i$  trong  $n-1$  dòng tiếp theo chứa 2 số nguyên  $a_i$  và  $b_i$  xác định giữa 2 ụ mối  $a_i$  và  $b_i$  có đường nối ngầm trực tiếp ( $1 \leq a_i, b_i \leq n$ ,  $a_i \neq b_i$ ,  $i = 1 \div n$ ), không có cặp đường nối nào bị đưa ra trùng lặp.

**Kết quả:** Đưa ra file văn bản TERMITE.OUT trên một dòng  $n$  số nguyên  $\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n$ .

**Ví dụ:**

TERMITE.INP	TERMITE.OUT
<pre> 6 2 1 2 3 4 6 6 5 6 2 </pre>	<pre> 2 6 2 6 6 0 </pre>



## Giải thuật:

- + Cấu trúc cây có thể biểu diễn bằng vec tơ  $\mathbf{PA} = (\mathbf{pa}_1, \mathbf{pa}_2, \dots, \mathbf{pa}_n)$  cho biết nút  $i$  được nối với nút  $\mathbf{pa}_i$ ,  $\mathbf{pa}_i = 0$  nếu  $i$  là nút gốc,
- + Dạng biểu diễn này cho phép dễ dàng tìm nút cha chung gần nhất, đường kính của cây, tìm nút trung tâm, . . .

Tổ chức dữ liệu:

- vector<int> $\mathbf{a}[\mathbf{MX}]$  – véc tơ  $\mathbf{a}_i$  lưu trữ các đỉnh kề với đỉnh  $i$ ,
- pair<int, int> >  $\mathbf{na}[\mathbf{MX}]$  – cặp dữ liệu ( $s_i, i$ ) lưu trữ số lượng đỉnh kề với đỉnh  $i$  (kích thước của véc tơ  $a_i$ ),
- int  $\mathbf{pa}[\mathbf{MX}] = \{0\}$  – lưu trữ kết quả.

Xử lý:

- + Với mỗi cặp dữ liệu ( $u, v$ ) – nạp  $v$  vào  $\mathbf{a}_u$  và  $u$  – vào  $\mathbf{a}_v$ , tăng số lượng trong các phần tử tương ứng của  $\mathbf{na}$ ,
- + Sắp xếp  $\mathbf{na}$  theo thứ tự tăng dần,
- + Chọn đỉnh gốc: có thể chọn đỉnh bất kỳ, nhưng để đỉnh gốc là điểm trung tâm hay gần với trung tâm – nên chọn đỉnh ứng với phần tử  $\mathbf{na}$  cuối cùng,
- + Với mỗi đỉnh  $tg$  đã chọn:
  - ❖ Duyệt tất cả các đỉnh  $t$  kề với  $tg$ ,
  - ❖ Điều kiện ghi nhận vào  $\mathbf{PA}$ :
    - ❖ Đỉnh  $tg$  chưa được ghi nhận cạnh đi ra ( $\mathbf{pa}_{tg} = 0$ ),
    - ❖ Quan hệ  $t \rightarrow tg$  chưa được ghi nhận ( $\mathbf{pa}_t \neq tg$ ).

Độ phức tạp của giải thuật:  $O(nlnn)$ .

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "termite."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
typedef pair<int,int> pii;
const int MX=300002;
int n,u,v,w,t,tg,rt,pa[MX]={0};
vector<int>a[MX];
pii na[MX];

int main()
{clock_t aa=clock();
 fi>>n;
 for(int i=0;i<=n;++i){na[i].first=0;na[i].second=i;}
 for(int i=1;i<n;++i)
 {
     fi>>u>>v>>w;
     a[u].push_back(v);++na[u].first;
     a[v].push_back(u);++na[v].first;
 }
 sort(na+1,na+n+1);

 rt=na[n].second;
 for(int i=1;i<=n;++i)
 {
     tg=na[i].second;
     for(int j=1;j<=na[i].first;++j)
     {
         t=a[tg][j-1];
         if(pa[tg]==0 && pa[t]!=tg)pa[tg]=t;
     }
 }
 for(int i=1;i<=n;++i)fo<<pa[i]<<' ';
 clock_t bb=clock();
 fo<<"\nTime: "<<(double)(bb-aa)/1000<<" sec";
}
```



## Kiểm tra tồn tại chu trình

Xét đồ thị có hướng  $G$  không chứa các cạnh kép hoặc một đỉnh tự nối với chính nó. Cần xác định đồ thị có chứa chu trình hay không. Nếu tồn tại chu trình thì chỉ ra một chu trình tùy chọn.

### Bài toán

Xét đồ thị có hướng  $n$  đỉnh và  $m$  cạnh, cạnh thứ  $i$  được xác định bởi cặp số nguyên  $(a_i, b_i)$  cho biết tồn tại cạnh  $a_i \rightarrow b_i$ ,  $i = 1 \dots m$ .

Hãy xác định đồ thị chứa chu trình hay không và đưa ra thông báo “**Acylic**” nếu không có chu trình. Nếu tồn tại chu trình thì đưa ra thông báo “**Cyclic**” và ở dòng tiếp theo – dãy các đỉnh xác định một chu trình tùy chọn (xem ví dụ).

**Dữ liệu:** Vào từ file văn bản ACYCLIC.INP:

- ✚ Dòng đầu tiên chứa 2 số nguyên  $n$  và  $m$  ( $1 \leq n \leq 10^5$ ,  $0 \leq m \leq 2 \times 10^5$ ),
- ✚ Dòng thứ  $i$  trong  $m$  dòng sau chứa 2 số nguyên  $a_i, b_i$  ( $1 \leq a_i, b_i \leq n$ ,  $a_i \neq b_i$ ).

**Kết quả:** Đưa ra file văn bản ACYCLIC.OUT kết quả theo quy cách đã nêu.

**Ví dụ:**

ACYCLIC.INP
10 10
1 2
2 4
4 3
3 6
3 8
4 5
4 7
9 2
6 10
10 9

ACYCLIC.OUT
Cyclic
2 4 3 6 10 9 2

## Giải thuật

Thực hiện dãy các lần duyệt theo chiều sâu, mỗi lần bắt đầu từ đỉnh chưa tới thăm bởi các lần duyệt trước.

Khi tới một đỉnh, ta đánh số gán cho nó màu 1, khi ra khỏi đỉnh – gán màu 2.

Nếu trong quá trình duyệt gặp đỉnh màu 1 có nghĩa là tồn tại chu trình.

Cần lưu trữ danh sách các đỉnh đã đi qua trong mỗi lần duyệt để dẫn xuất chu trình nếu có.

Tổ chức dữ liệu:

- ▀ Mảng `vector < vector<int> > g(N)` lưu cấu trúc đồ thị,
- ▀ Mảng `vector<int> p` đánh dấu đỉnh đã thăm,
- ▀ Mảng `vector<char> cl` đánh dấu màu để nhận dạng chu trình,
- ▀ Mảng `vector<int> cycle` lưu các đỉnh trong chu trình.

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "acyclic."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N= 100000;
int n,m,x,y;
vector < vector<int> > g(N);
vector<char> cl;
vector<int> p;
int cycle_st, cycle_end;

bool dfs (int v)
{
    cl[v] = 1;
    for (auto &to: g[v])
    {
        if (cl[to] == 0)
        {
            p[to] = v;
            if (dfs (to)) return true;
        }
        else if (cl[to] == 1)
        {
            cycle_end = v;
            cycle_st = to;
            return true;
        }
    }
    cl[v] = 2;
    return false;
}

int main()
{
    fi>>n>>m;
    for(int i=0; i<m; ++i)
```

```

{
    fi>>x>>y;
    --x; --y; g[x].push_back(y);
p.assign(n, -1);
cl.assign(n, 0);
cycle_st = -1;
for (int i=0; i<n; ++i)
if (dfs (i)) break;
if (cycle_st == -1) fo<<"Acyclic\n";
else
{
    fo<<"Cyclic\n";
    vector<int> cycle;
    cycle.push_back (cycle_st);
    for (int v=cycle_end; v!=cycle_st; v=p[v]) cycle.push_back (v);
    cycle.push_back (cycle_st);
    reverse (cycle.begin(), cycle.end());
    for (size_t i=0; i<cycle.size(); ++i) fo<<cycle[i]+1<<' ';
    fo<<'\n';
}
fo<<"\nTime: "<<clock() / (double) 1000<<" sec";
}

```



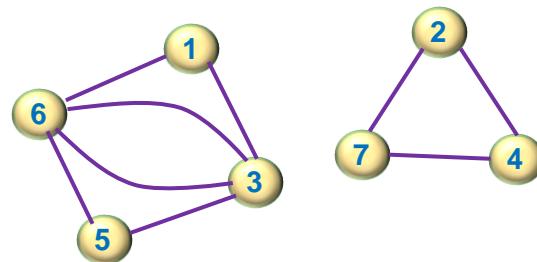
## Thành phần liên thông

Xét đồ thị vô hướng  $n$  đỉnh và  $m$  cạnh. Miền liên thông là một nhóm đỉnh mà giữa 2 đỉnh bất kỳ trong nhóm có đường đi tới nhau. Thành phần liên thông là miền liên thông mà giữa một đỉnh bất kỳ thuộc miền liên thông và một đỉnh khác ngoài miền liên thông không có đường đi tới nhau. Nói một cách khác thành phần liên thông là miền liên thông cực đại theo số đỉnh.

### Bài toán

Cho đồ thị vô hướng  $n$  đỉnh, các đỉnh được đánh số từ 1 đến  $n$ . Đồ thị có  $m$  cạnh, cạnh thứ  $j$  nối trực tiếp 2 đỉnh  $a_j$  và  $b_j$ ,  $j =$

$1 \div m$ . Hãy xác định các thành phần liên thông của đồ thị. Với thành phần liên thông thứ  $k$  xác định được, đưa ra một dòng thông báo dưới dạng “**Component k:**”, số lượng đỉnh trong thành phần và danh sách các đỉnh đó.



**Dữ liệu:** Vào từ file văn bản CONNECTED.INP:

- ✚ Dòng đầu tiên chứa 2 số nguyên  $n$  và  $m$  ( $1 \leq n \leq 10^5$ ,  $0 \leq m \leq 5 \times 10^5$ ),
- ✚ Dòng thứ  $j$  trong  $m$  dòng sau chứa 2 số nguyên  $a_j$  và  $b_j$  ( $1 \leq a_j, b_j \leq n$ ).

**Kết quả:** Đưa ra file văn bản CONNECTED.OUT thông báo về các miền liên thông xác định được theo quy cách đã nêu.

**Ví dụ:**

CONNECTED.INP
7 9
1 3
1 6
2 4
5 6
6 3
2 7
5 3
4 7
6 3

CONNECTED.OUT
Component 1 : 4 1 3 6 5
Component 2 : 3 2 4 7

## Giải thuật

Để tìm kiếm thành phần liên thông ta có thể thực hiện loang theo chiều rộng hoặc chiều sâu, hiệu quả giải thuật tương tự như nhau.

Bắt đầu từ đỉnh 1 ta tiến hành loang tìm miền liên thông cực đại (theo số đỉnh) chưa đỉnh 1, đánh dấu các đỉnh này và đưa các đỉnh đã tới thăm vào vector kết quả.

Sau khi xuất kết quả của thành phần liên thông tìm được, xóa vector kết quả, lặp lại việc tìm kiếm từ một đỉnh chưa được đánh dấu.

Quá trình xử lý kết thúc khi mọi đỉnh đều được đánh dấu.

Phụ thuộc vào cách đánh dấu và lưu trữ, ta có thể đưa ra các thông tin tổng kết như số lượng thành phần liên thông, thành phần liên thông nhiều phần tử nhất, thành phần liên thông ít phần tử nhất, . . . trước khi đưa ra thông tin cụ thể về các thành phần liên thông.

Độ phức tạp của giải thuật:  $O(n+m)$

Tổ chức dữ liệu:

- Các mảng **vector <int>** g [N] lưu danh sách đỉnh kè,
- Mảng **vector <int>** s lưu danh sách đỉnh của một thành phần liên thông,
- Mảng **vector<bool>** used (N, **false**) đánh dấu các đỉnh đã duyệt.

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "connected."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100000;

int n,m,k=0,x,y;
vector<bool> used(N,false);
vector <int> s,g[N];

void dfs(int v)
{
    used[v]=true; s.push_back(v);
    for(int i:g[v])
        if(!used[i])dfs(i);
}

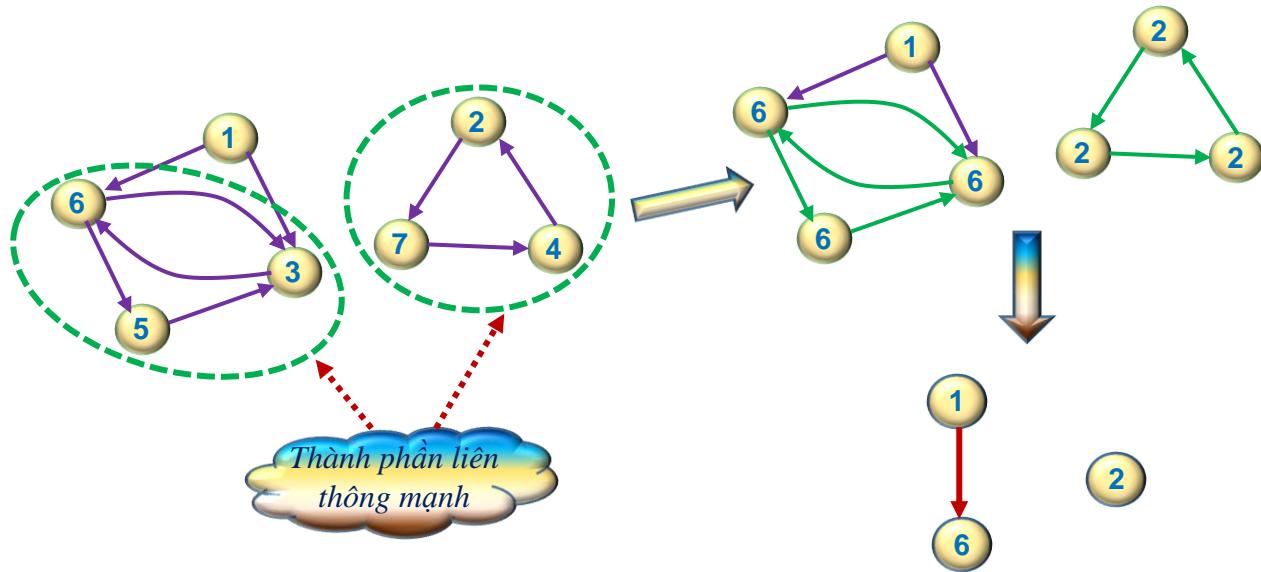
int main()
{
    fi>>n>>m;
    for(int i=0;i<m;++i)
    {
        fi>>x>>y;
        --x; --y; g[x].push_back(y); g[y].push_back(x);
    }
    for(int i=0;i<n;++i)
    {
        if(!used[i])
        {
            ++k; s.clear();
            dfs(i);
            fo<<"Component "<<k<<" : "<<s.size()<<' ';
            for(int j:s)fo<<j+1<<' '; fo<<'\'n';
        }
    }
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```



## Liên thông mạnh và cô đặc đồ thị

Xét đồ thị có hướng  $n$  đỉnh và  $m$  cạnh. Các đỉnh được đánh số từ 1 đến  $n$ . Cạnh thứ  $j$  của đồ thị xác định có đường đi trực tiếp từ đỉnh  $a_j$  tới đỉnh  $b_j$ ,  $j = 1 \dots n$ .

Thành phần liên thông mạnh (*Strongly Connected Component – SCC*) là một tập các đỉnh, trong đó nếu  $u$  và  $v$  thuộc tập thì tồn tại đường đi từ  $u$  tới  $v$  và đường đi từ  $v$  tới  $u$  (ký hiệu  $u \Leftrightarrow v$ ).



Đồ thị cô đặc

Rõ ràng rằng các thành phần liên thông mạnh không giao nhau và có thể dùng làm công cụ phân lớp các đỉnh của đồ thị. Nếu ta chọn một đỉnh của thành phần liên thông mạnh làm *đỉnh đại diện* và trong đồ thị ban đầu thay toàn bộ miền liên thông mạnh bằng đỉnh đại diện, ta có *đồ thị cô đặc*.

Tính chất quan trọng nhất của đồ thị cô đặc là *không chứa chu trình*. Điều này có thể dễ dàng chứng minh bằng phương pháp phản chứng.

### Giải thuật Kosaraju – Sharir

Độc lập với nhau, Kosaraju và Sharir công bố giải thuật tìm thành phần liên thông mạnh năm 1979.

Tiến hành duyệt đồ thị theo chiều sâu và lưu thời điểm ra ở mỗi đỉnh. Trên thực tế, ta không cần bắn thân thời điểm ra mà chỉ *cần trình tự ra* của đỉnh vì vậy chỉ cần *lưu các đỉnh theo trình tự tăng dần của thời điểm ra*.

Gọi  $C$  là một thành phần liên thông mạnh,  $t_{\text{out}}[v]$  là thời điểm ra của đỉnh  $v$ ,  $t_{\text{out}}[C] -$  thời điểm khỏi  $C$ ,  $t_{\text{out}}[C] = \max\{t_{\text{out}}[v], v \in C\}$ .

## Định lý

Nếu  $C$  và  $C'$  là 2 thành phần liên thông mạnh của đồ thị và trong đồ thị có đặc có cạnh  $C \rightarrow C'$ , thì  $t_{\text{out}}[C] > t_{\text{out}}[C']$ .

### Chứng minh

Có hai trường hợp xảy ra:

- ⊕ Trong quá trình duyệt theo chiều sâu ta gặp đỉnh  $v \in C$  và chưa gặp đỉnh nào trong số các đỉnh còn lại của  $C$ , cũng như chưa gặp đỉnh nào thuộc  $C'$ . Nhưng theo điều kiện của định lý tồn tại cạnh  $C \rightarrow C'$ , nên từ  $v$ , duyệt theo chiều sâu, có thể đến tất cả các đỉnh của  $C$  và của  $C'$ . Nói một cách khác mọi đỉnh còn lại của  $C$  và mọi đỉnh của  $C'$  đều là con của  $v$ , từ đó suy ra, với đỉnh  $u$  bất kỳ,  $u \in C \cup C'$  đều có  $t_{\text{out}}[v] > t_{\text{out}}[u]$ . Đó là điều phải chứng minh.
- ⊕ Trong quá trình duyệt theo chiều sâu ta gặp một đỉnh  $v$  của  $C'$  trước và chưa gặp đỉnh nào thuộc  $C$  cũng như các đỉnh còn lại của  $C'$ . Theo điều kiện của định lý, tồn tại cạnh  $C \rightarrow C'$  và tính chất của đồ thị có đặc là không có chu trình nên không có đường đi từ  $C'$  tới  $C$ . Vì vậy khi duyệt theo chiều sâu từ  $v$ , ta không thể tới các đỉnh của  $C$ . Điều này nói lên rằng các đỉnh còn lại của  $C'$  sẽ gặp muộn hơn trong quá trình duyệt tức là  $t_{\text{out}}[v] > t_{\text{out}}[u]$ . Đó là điều phải chứng minh.

## Giải thuật

Định lý này là cơ sở của giải thuật xác định thành phần liên thông mạnh. Quá trình xử lý bao gồm hai lần duyệt theo chiều sâu và vì vậy có độ phức tạp  $O(n+m)$ .

Bước 1: Duyệt đồ thị theo chiều sâu và xây dựng mảng  $ord$  ghi nhận các đỉnh theo thời điểm ra tăng dần,

Bước 2: Xây dựng đồ thị quan hệ chuyển vị, tức là đồ thị trong đó quan hệ  $u \rightarrow v$  được thay bằng  $v \rightarrow u$ , duyệt đồ thị này theo trình tự giảm dần thời điểm ra của các đỉnh. Các tập nhận được sẽ là thành phần liên thông mạnh cần tìm.

Nhận xét:

- ▣ Đồ thị quan hệ chuyển vị có thể xây dựng ngay trong quá trình nhập dữ liệu,
- ▣ Bản chất bước 1 là sắp xếp tố pô các đỉnh của đồ thị,
- ▣ Bước 2 là tìm thành phần liên thông theo trình tự tố pô đã xác định.

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "scc."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100000;

int n,m,k=0,x,y;
vector<bool> used(N,false);
vector <int> ord,component,g[N],gr[N],g_cond;

void dfs1(int v)
{
    used[v]=true;
    for(int i:g[v])
        if(!used[i])dfs1(i);
    ord.push_back(v);
}

void dfs2(int v)
{
    used[v]=true;
    component.push_back(v);
    for(int i:gr[v])
        if(!used[i])dfs2(i);
}

int main()
{
    fi>>n>>m;
    for(int i=0;i<m;++i)
    {
        fi>>x>>y;
        --x; --y;
        g[x].push_back(y);
        gr[y].push_back(x);
    }

    for(int i=0;i<n;++i)
        if(!used[i]) dfs1(i);
    used.assign(n,false);
    g_cond.reserve(n);
    for(int i=n-1;i>=0;--i)
    {
        int v=ord[i];
        if(!used[v])
        {
            ++k; dfs2(v);

            fo<<"Component "<<k<<" : "<<component.size()<<' ';
            for(int &j:component) fo<<j+1<<' '; fo<<endl;
            v=component[0];
            for(int &j:component) g_cond[j]=v;

            component.clear();
        }
    }
}
```

```

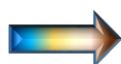
    }
}

fo<<"Condensing Graph: ";
for(int i=0;i<n;++i) fo<<g_cond[i]+1<<' ';
fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}

```

Kết quả với đồ thị đã nêu trong hình trên:

7	9
1	6
1	3
6	3
3	6
6	5
5	3
2	7
7	4
4	2



Component 1 : 3 2 4 7
Component 2 : 1 1
Component 3 : 3 6 3 5
Condensing Graph: 1 2 6 2 6 6 2
Time: 0.003 sec

*Input*

*Output*



## Tìm kiếm cầu trên đồ thị vô hướng

Xét đồ thị vô hướng  $G$  có  $n$  đỉnh và  $m$  cạnh. Giả thiết giữa 2 đỉnh của  $G$  có *không quá một cạnh* nối trực tiếp. Tập các đỉnh của  $G$  mà giữa 2 đỉnh bất kỳ của tập có đường đi tới nhau (trực tiếp hoặc qua các đỉnh khác) gọi là một thành phần liên thông.

Cầu của đồ thị  $G$  là cạnh mà nếu bỏ nó số thành phần của  $G$  sẽ tăng.

Có 2 loại giải thuật chính trong việc tìm cầu ứng với các trường hợp:

Trường hợp đồ thị đã cho trước và không thay đổi trong quá trình tìm kiếm: Giải thuật offline,

Đồ thị  $G$  thay đổi (thêm, bớt cạnh) trong quá trình tìm cầu: Giải thuật online.

### Giải thuật tìm cầu offline

Đồ thị  $G$  đã cho không thay đổi trong quá trình giải bài toán. Việc tìm kiếm có thể thực hiện bằng giải thuật loang theo chiều sâu với *độ phức tạp  $O(n+m)$* .

Gọi **root** là gốc của đồ thị. Xuất phát từ **root**, loang theo chiều sâu (*dfs*) ta tới đỉnh **v**. Nếu với **v** tồn tại cạnh (**v**, **to**) và từ đỉnh **to** hoặc từ các đỉnh con của **to** không có đường nối tới **v** hay tới đỉnh cha của **v** thì (**v**, **to**) là một cầu. Trong trường hợp ngược lại (**v**, **to**) không phải là cầu.

Phương pháp hiệu quả nhận biết điều kiện trên được thực hiện dựa trên thời điểm vào của mỗi đỉnh.

Gọi **tin[v]** – thời điểm vào đỉnh **v** khi duyệt theo chiều sâu.

Xây dựng mảng hỗ trợ **fup[v]**:

$$fup[v] = \min \begin{cases} tin[v], \\ tin[p] \text{ với mọi } p \text{ dẫn tới } v \text{ hoặc đỉnh cha của } v, \\ fup[to] \text{ với mọi } (v, to). \end{cases}$$

Ta thấy, từ **v** hoặc từ đỉnh con của **v** có đỉnh nối tới đỉnh cha của **v** khi và chỉ khi tồn tại đỉnh con **to** thỏa mãn điều kiện  $fup[to] \leq tin[v]$ . Điều kiện bằng xảy ra khi tồn tại đỉnh con dẫn tới đúng **v**.

Ngoài ra, còn cần mảng **used[to]** đánh dấu đỉnh khi duyệt:

**used[to] = false** – đỉnh chưa được duyệt,

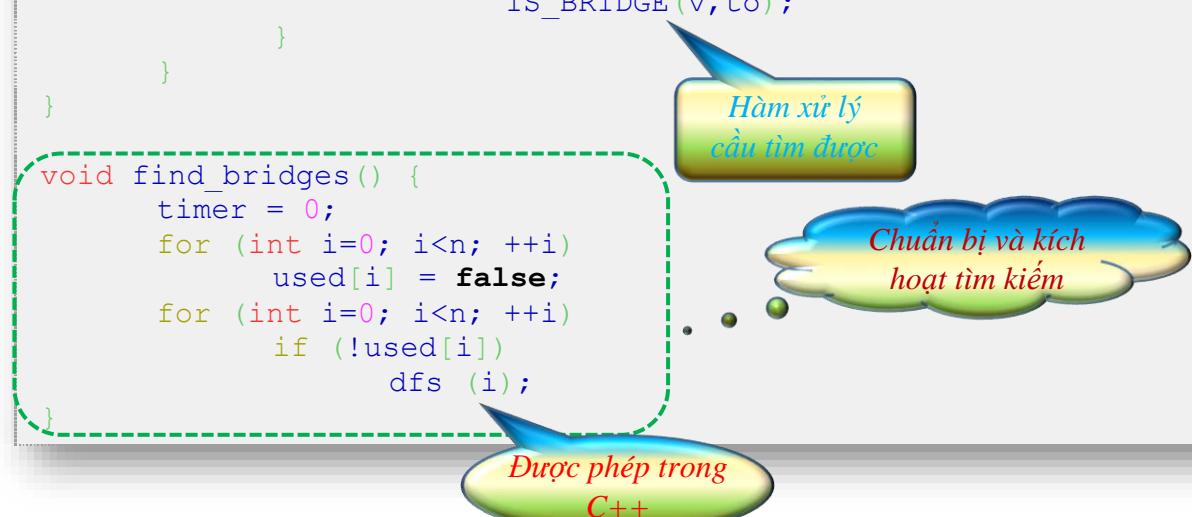
**used[*to*] = true** và **to ≠ parent** – điều kiện đi ngược lên trên,  
**to = parent** – điều kiện cạnh dẫn ngược đã xét.

*Giải thuật:*

```
const int MAXN = ...;
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = fup[v] = timer++;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (to == p) continue;
        if (used[to])
            fup[v] = min (fup[v], tin[to]);
        else {
            dfs (to, v);
            fup[v] = min (fup[v], fup[to]);
            if (fup[to] > tin[v])
                IS_BRIDGE(v,to);
        }
    }
}

void find_bridges() {
    timer = 0;
    for (int i=0; i<n; ++i)
        used[i] = false;
    for (int i=0; i<n; ++i)
        if (!used[i])
            dfs (i);
}
```



## Giải thuật tìm cầu online

Giả thiết đồ thị **G** n đỉnh, trong đó giữa 2 đỉnh bất kỳ có không quá một đường nối trực tiếp. Xét trường hợp **G** không biết trước, từng cạnh một sẽ được *bổ sung dần* trong quá trình xử lý.

Ban đầu **G** không chứa cạnh nào. Có **m** truy vấn, mỗi truy vấn là yêu cầu xác định các cầu của đồ thị sau khi bổ sung thêm cạnh (**a**, **b**).

Giải thuật đặt nền tảng trên cấu trúc dữ liệu của hệ thống các tập không giao nhau và có độ phức tạp  $O(n \log n + m)$ .

Các cầu chia đỉnh của đồ thị thành các nhóm song liên thông. Nếu ánh xạ mỗi nhóm thành một đỉnh và chỉ giữ lại các cạnh là cầu thì ta có một rừng, tức là nhiều cây rời rạc.

Ban đầu, khi đồ thị còn rỗng (không có cạnh nào) ta có **n** nhóm đỉnh song liên thông 2 chiều và các nhóm này không giao nhau.

Có 3 tình huống có thể xảy ra khi bổ sung thêm một cạnh (**a**, **b**):

- ⊕ Cả 2 đỉnh **a** và **b** đều thuộc một nhóm song liên thông, như vậy cạnh này không thể là cầu, cấu trúc của rừng không thay đổi và có thể bỏ qua, không xử lý cạnh,
- ⊕ Các đỉnh **a** và **b** thuộc 2 nhóm liên thông khác nhau, (**a**, **b**) là một cầu nối 2 cây, 2 nhóm liên thông này sẽ được hợp nhất thành một và giữ nguyên các cầu đã có, số lượng cầu tăng thêm 1,
- ⊕ Các đỉnh **a** và **b** thuộc một nhóm liên thông nhưng thuộc 2 nhóm song liên thông khác nhau. Trong trường hợp này cạnh mới sẽ tạo với một số cầu hiện có thành chu trình, những cầu đó bây giờ không phải là cầu, các nhóm song liên thông tham gia vào chu trình hình thành được hợp nhất thành một nhóm song liên thông mới. Số lượng cầu của đồ thị sẽ giảm.

Toàn bộ giải thuật xoáy quanh việc nhận biết và xử lý một cách hiệu quả 3 tình huống trên.

### Tổ chức dữ liệu lưu trữ rừng

Ta cần 2 hệ thống lưu trữ các tập không giao nhau, một để lưu trữ các *thành phần liên thông* và cấu trúc thứ 2 – lưu trữ các *thành phần song liên thông*.

Ngoài ra, còn cần thêm mảng **par[]** móc nối tới nút cha để lưu trữ cây trong các thành phần song liên thông.

### Các phép xử lý cần hiện thực hóa

- Kiểm tra 2 đỉnh có thuộc một nhóm liên thông/song liên thông hay không. Việc kiểm tra được thực hiện theo sơ đồ xử lý chuẩn hệ thống các tập không giao nhau.
- Hợp nhất 2 cây theo cạnh (**a**, **b**). Các đỉnh **a** và **b** có thể không phải là gốc của cây nào trong 2 cây cần hợp nhất, vì vậy treo một cây vào cây khác. Ví dụ, có thể treo một cây vào đỉnh **a**, sau đó biến đỉnh **a** thành con của **b** và gắn vào đó cây thứ 2.

Như vậy quá trình xử lý đòi hỏi phải có giải thuật hiệu quả việc treo cây gốc **r** vào đỉnh **v**. Để làm được việc đó cần tìm đường đi từ **v** tới **r**, đảo mốc nối **par[]** trên đường đi, đồng thời cập nhật mốc nối tương ứng trong dữ liệu về hệ thống các tập không giao nhau tương ứng với nhóm liên thông đang xử lý.

Như vậy độ phức tạp của việc treo cây là  $O(h)$ , trong đó **h** là độ cao của cây. Về toàn cục, có thể đánh giá độ phức tạp là  $O(\text{size})$ , trong đó **size** – số đỉnh trong cây.

Để đảm bảo hiệu quả cao cần treo cây có ít đỉnh hơn vào cây còn lại.

Gọi **T(n)** là số phép tính cần thực hiện để nhận được cây n đỉnh từ các phép hợp nhất và treo cây, ta có:

$$T(n) = \max_{k=1 \dots n-1} \{ T(k) + T(n-k) + O(n) \} = O(n \log n)$$

- Tìm chu trình hình thành bởi cạnh (**a**, **b**) được bô sung. Trên thực tế, đó là việc tìm nút cha chung nhỏ nhất (**LCA**) của 2 đỉnh **a** và **b**. Do cuối cùng ta sẽ thay tất cả các đỉnh trong nhóm bằng một đỉnh đại diện nên có thể sử dụng giải thuật bất kỳ tìm **LCA** với độ phức tạp tỷ lệ với độ dài cần duyệt.

Vì trong tay ta chỉ có **par[]** – thông tin mốc nối tới nút cha, nên con đường duy nhất tìm **LCA** ở đây là đánh dấu các đỉnh cha từ **a** và từ **b** cho đến khi gặp đỉnh đã được đánh dấu, đó sẽ là **LCA** cần tìm. Xác định lại đường từ nút này tới **a** và tới **b** ta sẽ có chu trình cần tìm.

Rõ ràng giả thuật tìm kiếm trên có độ phức tạp bằng độ dài đường đi.

- Nén chu trình hình thành bởi sự xuất hiện của cạnh mới (**a**, **b**). Ta phải tạo nhóm song liên thông mới mà không làm thay đổi cấu trúc cây, không thay đổi các mốc nối **par[]** và bảo toàn tính đúng đắn của các tập không giao nhau.

Phương pháp đơn giản nhất là nén các đỉnh trong chu trình mới tìm được vào **LCA** của chúng. Thật vậy, đỉnh **LCA** là cao nhất trong số các đỉnh được nén, vì vậy **par[]** của nó sẽ vẫn giữ nguyên như cũ. Thông tin về các đỉnh còn lại cũng không cần cập

nhật – chúng trở nên “*trong suốt*” trong các phép xử lý tiếp theo, còn trong hệ thống các tập không giao nhau các đỉnh này được trích dẫn tới đại diện là đỉnh **LCA**. Ta cũng không cần sắp xếp hay cập nhật trình tự đỉnh đại diện vì khi cần duyệt đã có **par[]** đảm bảo.

Độ phức tạp xử lý theo đúng sơ đồ lý thuyết về hệ thống các tập không giao nhau sẽ là  $O(\log n)$ . Tuy vậy, với đặc thù của bài toán đồ thị, ta có thể xử lý với *độ phức tạp  $O(1)$* : đơn thuần là gán cho **par[LCA]** giá trị **par** mới!

*Giải thuật:*

Giải thuật nêu dưới đây không lưu trữ thông tin về các cầu mà chỉ lưu trữ số lượng cầu trong biến **bridges**. Nếu cần thông tin về chính các cầu có thể tổ chức một tập hợp **s** để lưu trữ.

Hàm **init()** có nhiệm vụ khởi tạo hai hệ thống lưu trữ các tập không giao nhau cũng như giá trị đầu của các **par[]**.

```

const int MAXN = ...;
int n, bridges, par[MAXN], bl[MAXN], comp[MAXN], size[MAXN];

void init() {
    for (int i=0; i<n; ++i) {
        bl[i] = comp[i] = i;
        size[i] = 1;
        par[i] = -1;
    }
    bridges = 0;
}

int get (int v) {
    if (v===-1) return -1;
    return bl[v]==v ? v : bl[v]=get(bl[v]);
}

int get_comp (int v) {
    v = get(v);
    return comp[v]==v ? v : comp[v]=get_comp(comp[v]);
}

void make_root (int v) {
    v = get(v);
    int root = v,
        child = -1;
    while (v != -1) {
        int p = get(par[v]);
        par[v] = child;
        comp[v] = root;
        child=v; v=p;
    }
}

```

```

        size[root] = size[child];
    }

int cu, u[MAXN];

void merge_path (int a, int b) {
    ++cu;

    vector<int> va, vb;
    int lca = -1;
    for(;;) {
        if (a != -1) {
            a = get(a);
            va.pb(a);

            if (u[a] == cu) {
                lca = a;
                break;
            }
            u[a] = cu;
            a = par[a];
        }

        if (b != -1) {
            b = get(b);
            vb.pb(b);

            if (u[b] == cu) {
                lca = b;
                break;
            }
            u[b] = cu;
            b = par[b];
        }
    }

    for (size_t i=0; i<va.size(); ++i) {
        bl[va[i]] = lca;
        if (va[i] == lca) break;
        --bridges;
    }
    for (size_t i=0; i<vb.size(); ++i) {
        bl[vb[i]] = lca;
        if (vb[i] == lca) break;
        --bridges;
    }
}

void add_edge (int a, int b) {
    a = get(a);    b = get(b);
    if (a == b) return;

    int ca = get_comp(a),
        cb = get_comp(b);
    if (ca != cb) {

```

```

    ++bridges;
    if (size[ca] > size[cb]) {
        swap(a, b);
        swap(ca, cb);
    }
    make_root(a);
    par[a] = comp[a] = b;
    size[cb] += size[a];
}
else
    merge_path(a, b);
}

// Kết giải thuật

```

*Chú thích:*

Mảng **bl []** dùng để lưu trữ các tập không giao nhau áp dụng trên các nhóm đỉnh song liên thông,

Hàm **get(v)** trả về giá trị đỉnh đại diện cho nhóm đỉnh song liên thông, sau khi nén các đỉnh trong nhóm song liên thông trở nên “trong suốt”, mọi xử lý sẽ được thực hiện với đỉnh đại diện,

Mảng **comp []** – lưu các tập không giao nhau áp dụng trên các nhóm đỉnh liên thông,

Mảng **size []** – lưu kích thước tập xác định bởi comp,

Hàm **get\_comp(v)** trả về đỉnh đại diện của nhóm liên thông (trên thực tế đó là gốc của cây),

Hàm **make\_root(v)** có nhiệm vụ treo cây vào đỉnh **v**: di chuyển từ đỉnh **v** theo các nút cha về gốc, đảo mốc nối **par []** (cho chỉ xuống dưới, hướng về **v**), đồng thời cập nhật con trỏ **comp []** để chỉ về gốc mới, sau khi gắn cây vào gốc mới – cập nhật **size** của nhóm liên thông. Lưu ý là trong quá trình làm việc phải sử dụng hàm **get()** để lấy thông tin về đỉnh đại diện nhóm liên thông vì các đỉnh trong nhóm về mặt lô gic – đã bị xóa.

Hàm **merge\_path(a, b)** – tìm **LCA** của các đỉnh **a** và **b**. Việc dùng mảng đánh dấu sẽ tiết kiệm nhiều thời gian hơn dùng **set**. Các đường đã đi qua được lưu trữ trong **va** và **vb** để sau này duyệt lại, tìm tất cả các đỉnh thuộc chu trình. Tất cả các đỉnh này được nén lại, bằng cách nối chúng với **LCA**. Độ phức tạp của công đoạn này là O(logn). Số lượng các cạnh đã đi qua (số cầu cũ) được đếm đồng thời các xử lý khác và sau đó giảm giá trị số lượng cầu **bridges** một cách tương ứng.

Hàm xử lý truy vấn **`add_edge(a, b)`** xác định các thành phần liên thông chứa **a** và **b**. Nếu **a** và **b** thuộc các thành phần liên thông khác nhau thì treo cây nhỏ hơn vào gốc mới, sau đó hợp nhất kết quả với cây lớn hơn. Nếu **a** và **b** ở cùng một cây nhưng thuộc các nhóm song liên thông khác nhau thì gọi hàm **`merge_path(a, b)`** để tìm chu trình và sau đó – hợp nhất thành một nhóm song liên thông.



## Tìm điểm khớp của đồ thị

Xét đồ thị vô hướng liên thông. Điểm khớp của đồ thị (*Articulation Point*) là đỉnh mà nếu ta loại bỏ nó đồ thị không còn liên thông.

### Giải thuật

Chọn một đỉnh xuất phát làm gốc (**root**) ta tiến hành loang theo chiều sâu.

Dễ dàng nhận thấy tiêu chuẩn làm điểm khớp của một đỉnh là như sau:

Giả thiết trong quá trình duyệt ta tới thăm đỉnh  $v \neq \text{root}$ . Nếu cạnh ( $v, to$ ) thỏa mãn điều kiện từ  $to$  và từ các đỉnh con của nó theo quá trình duyệt không có cạnh nối tới  $v$  hoặc đỉnh cha của  $v$  thì  $v$  là điểm khớp. Trong trường hợp ngược lại –  $v$  không là điểm khớp,

Nếu  $v = \text{root}$ ,  $v$  là điểm khớp khi và chỉ khi nó có nhiều hơn một đỉnh con (theo quá trình duyệt).

Việc nhận dạng một đỉnh có thể làm điểm khớp hay không được thực hiện dựa vào thời điểm vào của đỉnh.

Gọi  $t\_in[v]$  là thời điểm vào của đỉnh  $v$ . Xác định  $fup[v]$  như sau:

$$fup[v] = \min \begin{cases} t\_in[v], \\ t\_in[p] \text{ với mọi cạnh ngược } (v, p), \\ fup[to] \text{ với mọi cạnh } (v, to) \text{ của đồ thị.} \end{cases}$$

Từ  $v$  hoặc đỉnh con của nó có cạnh dẫn ngược tới một đỉnh cha của  $v$  khi và chỉ khi có một đỉnh con  $to$  thỏa mãn điều kiện  $fup[to] < t\_in[v]$ .

Như vậy, với cạnh đang xét ( $v, to$ ) trên cây tìm kiếm, nếu có  $fup[to] \geq t\_in[v]$  thì  $v$  là một nút khớp.

Với đỉnh gốc **root**, cần tính số nút con trực tiếp (theo cây tìm kiếm). Nếu số nút con lớn hơn 1 thì **root** cũng là một điểm khớp.

Độ phức tạp của giải thuật:  $O(n+m)$ .

Tổ chức dữ liệu:

- ▀ Mảng **vector<int>** g [MAXN] cấu trúc đồ thị,
- ▀ Mảng **bool** used [MAXN] đánh dấu các đỉnh đã duyệt,
- ▀ Mảng **int** tin [MAXN] lưu thời điểm vào của mỗi đỉnh,
- ▀ Mảng **int** fup [MAXN] lưu giá trị nhận dạng,
- ▀ Mảng **vector<int>** cutpoint lưu điểm khớp của đồ thị.

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "cutpoint."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int MAXN = 100000;

int n,m,k,x,y;
vector<int> g[MAXN],cutpoint;
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

void dfs (int v, int p = -1)
{
    used[v] = true;
    tin[v] = fup[v] = timer++;
    int children = 0;
    for (size_t i=0; i<g[v].size(); ++i)
    {
        int to = g[v][i];
        if (to == p) continue;
        if (used[to])
            fup[v] = min (fup[v], tin[to]);
        else
        {
            dfs (to, v);
            fup[v] = min (fup[v], fup[to]);
            if (fup[to] >= tin[v] && p != -1)
                cutpoint.push_back(v);
            ++children;
        }
    }
    if (p == -1 && children > 1)
        cutpoint.push_back(v);
}

int main()
{
    fi>>n>>m;
    for(int i=0;i<m;++i)
    {
        fi>>x>>y;
        --x; --y;
        g[x].push_back(y);
        g[y].push_back(x);
    }
    timer = 0;
    for (int i=0; i<n; ++i)
        used[i] = false;
    dfs (0);
    fo<<cutpoint.size()<<'\n';
    for(int &i:cutpoint) fo<<i+1<<' ';
    fo<<"\nTime: "<<clock () / (double) 1000<<" sec";
}
```



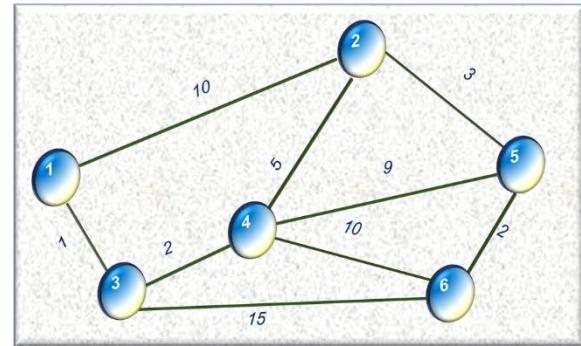
# HÀNG ĐƠI ƯU TIÊN VÀ GIẢI THUẬT DIJSKTRA

## BÀI TOÁN

Cho đồ thị  $n$  đỉnh và  $m$  cạnh. Đồ thị có thể có hướng hoặc không. Trọng số của mỗi cạnh là không âm. Hãy xác định đường đi ngắn nhất từ đỉnh  $s$  cho trước *tới mỗi đỉnh còn lại* và độ dài của đường đi đó.

**Dữ liệu:** Vào từ file văn bản DIJSKTRA.INP:

- Dòng đầu tiên chứa 2 số nguyên  $n$  và  $m$  ( $n > 0, m \geq 0$ ),
- Nếu  $m > 0$  thì mỗi dòng trong  $m$  dòng tiếp theo chứa 3 số nguyên  $a, b$  và  $r$  cho biết có cạnh nối từ  $a$  tới  $b$  với trọng số  $r$  ( $1 \leq a, b \leq n, a \neq b, 0 \leq r \leq 10^6$ ), không có hai dòng nào giống nhau,
- Dòng  $m+2$  chứa số nguyên  $s$  ( $1 \leq s \leq n$ ),
- Dòng cuối cùng chứa số nguyên  $k$  và sau đó là  $k$  số nguyên  $c_1, c_2, \dots, c_k$  cho biết phải dẫn xuất đường đi ngắn nhất từ  $s$  tới  $c_i, i = 1 \div k, 1 \leq c_i \leq n$ .



**Kết quả:** Đưa ra file văn bản DIJSKTRA.OUT:

- Dòng đầu tiên chứa  $n$  số nguyên, số thứ  $i$  là độ dài đường đi ngắn nhất từ  $s$  đến đỉnh  $i$ . Độ dài bằng -1 nếu không tồn tại đường đi từ  $s$  đến đỉnh đó,
- Dòng thứ  $i$  trong  $k$  dòng sau chứa thông tin về đường đi ngắn nhất (theo quy cách nêu trong ví dụ).

**Ví dụ:**

DIJSKTRA.INP	DIJSKTRA.OUT
6 9 1 2 10 1 3 1 2 4 5 3 4 2 2 5 3 3 6 15 4 6 10 4 5 9 5 6 2 2 2 1 6	8 0 7 5 3 5 Shortets route from 2 to 1: 2 4 3 1 Shortets route from 2 to 6: 2 5 6 Time: 0 sec

## GIẢI THUẬT

Tổ chức mảng  $D = (d_1, d_2, \dots, d_n)$ ,  $d_v$  lưu trữ độ dài đường đi ngắn nhất từ  $s$  tới  $v$ ,  $v = 1 \div n$ . Ban đầu  $d_s = 0$ ,  $d_v = \infty$ ,  $v = 1 \div n$ ,  $v \neq s$ . Ngoài ra, còn cần tới mảng lô gic  $U = (u_1, u_2, \dots, u_n)$  để đánh dấu, cho biết đỉnh  $v$  đã được xét hay chưa. Ban đầu,  $u_v = \text{false}$  với  $v = 1 \div n$ .

Bản thân giải thuật Dijkstra gồm  $n$  bước.

Ở mỗi bước cần chọn đỉnh  $v$  có  $d_v$  là nhỏ nhất trong số các đỉnh  $v$  chưa được đánh dấu, tức là

$$d_v = \min\{d_i \mid u_i = \text{false}, i = 1 \div n\}$$

Công việc tiếp theo trong bước này là điều chỉnh  $D$ : xét tất cả các cạnh  $(v, t)$ . Gọi  $1t$  là trọng số của cạnh  $(v, t)$ . Giá trị  $d_t$  được chỉnh lý theo công thức

$$d_t = \min\{d_t, d_v + 1t\}$$

Sau  $n$  bước, tất cả các đỉnh đều được đánh dấu và  $d_v$  sẽ là độ dài đường đi ngắn nhất từ  $s$  đến  $v$ . Nếu không tồn tại đường đi từ  $s$  đến  $v$  thì  $d_v$  vẫn nhận giá trị  $\infty$ .

Để khôi phục đường đi có độ dài ngắn nhất cần tổ chức mảng  $P = (p_1, p_2, \dots, p_n)$ , trong đó  $p_v$  lưu đỉnh cuối cùng trước đỉnh  $v$  trong đường đi ngắn nhất từ  $s$  đến  $v$ . Mỗi lần, khi  $d_t$  thay đổi giá trị thì đỉnh đạt min:  $p_t = v$ .

Tính đúng đắn của giải thuật được nêu trong nhiều tài liệu khác nhau và là điều không cần phải trình bày ở đây.

Điều quan trọng là đánh giá độ phức tạp của giải thuật và làm thế nào để giảm độ phức tạp đó. Giải thuật bao gồm  $n$  bước lặp, ở mỗi bước lặp cần duyệt tất cả các đỉnh và sau đó – chỉnh lý  $d_t$ . Như vậy giải thuật có độ phức tạp là  $O(n^2+m)$ .

## KỸ THUẬT CÀI ĐẶT HIỆU QUẢ CAO VỚI ĐỒ THỊ MA TRẠM THUA

Với đồ thị có số cạnh  $m$  nhỏ hơn nhiều so với  $n^2$  thì độ phức tạp của giải thuật có thể giảm xuống bằng việc cải tiến cách duyệt đỉnh ở mỗi bước lặp.

Mục tiêu này có thể đạt được thông qua việc sử dụng Cấu trúc vun đống Fibonacci (**Fibonacci Heap**), Cấu trúc tập hợp (**Set**) hoặc cấu trúc Hàng đợi ưu tiên (**Priority Queue**).

Cấu trúc vun đống Fibonacci cho phép giải bài toán tìm đường đi ngắn nhất với độ phức tạp  $O(n \log n + m)$ . Về mặt lý thuyết, đây là độ phức tạp *tối ưu* cho chương trình giải các bài toán dựa trên cơ sở giải thuật Dijkstra. Tuy nhiên việc cài đặt khác phức tạp vì thư viện chuẩn STL của các hệ thống lập trình dựa trên C++ chưa trực tiếp hỗ trợ Fibonacci Heap.

Các giải thuật dựa trên Set hoặc Priority\_Queue tuy không hiệu quả bằng sử dụng Fibonacci heap nhưng cũng cho độ phức tạp khá tốt, đủ chấp nhận được –  $O(m\log n)$ .

Với cấu trúc tập hợp (Set), mỗi đơn vị dữ liệu input cần được tổ chức dưới dạng cặp số nguyên (**pair<int, int>**), phần tử thứ nhất là trọng số và phần tử thứ hai là đỉnh của cạnh. Dữ liệu sẽ được tự động sắp xếp theo trọng số tăng dần – điều mà ta đang cần! Việc tổ chức mảng đánh dấu  $U$  cũng trở nên không cần thiết. Khi điều chỉnh  $D$ , mỗi khi có sự thay đổi, trước hết cần xóa cặp dữ liệu cũ, tính lại  $d_t$  và nạp lại cặp dữ liệu mới ứng với  $d_t$  vừa tính được.

Chương trình làm việc với hàng đợi ưu tiên hoạt động nhanh hơn một chút so với phương án sử dụng tập hợp. Tuy vậy, theo bản chất của cấu trúc dữ liệu, hệ thống không cung cấp dịch vụ xóa thông tin nếu nó không đứng ở đầu hàng đợi. Chính vì vậy phần lớn các giải thuật sử dụng hàng đợi (kể cả hàng đợi ưu tiên) đều phải giải quyết vấn đề **lọc dữ liệu thừa** trong quá trình xử lý. Trong giải thuật này, việc đó đơn thuần là so sánh giá trị lưu trữ ở đầu hàng đợi  $q$  với  $d_v$ . Khi chỉnh lý  $D$ , cặp giá trị ( $d_t, t$ ) được nạp vào hàng đợi. Cặp giá trị mới này sẽ đứng trước các cặp khác có cùng giá trị  $t$ .

Cần lưu ý là với khai báo **priority\_queue < pair<int, int> > q**; việc tổ chức ngầm định sẽ đặt giá trị lớn nhất lên đầu hàng đợi. Muốn có hàng đợi sắp xếp theo thứ tự tăng dần ta cần khai báo:

```
typedef p2 pair<int,int>;  
priority_queue <p2,vector<p2>,greater<p2> > q;
```

Trong giải thuật này các giá trị khóa đều không âm, vì vậy ta có thể dùng khai báo đơn giản theo kiểu ngầm định, nhưng nạp giá trị khóa âm. Kết quả là giá trị thực sự nhỏ nhất vẫn đứng ở đầu hàng đợi.

## Chương trình

```
// Sử dụng cấu trúc dữ liệu Hàng đợi ưu tiên:
#include <bits/stdc++.h>
using namespace std;
const int INF = 1000000000;
ifstream fi ("Dijkstra.inp");
ofstream fo ("Dijkstra.out");
pair<int,int>x;
int a,b,dd,k;
int main()
{
    int n,m;
    fi>>n>>m;
    vector < vector < pair<int,int> > > g (n+1);
    for(int i=1;i<=m;++i)
        {fi>>a>>b>>dd;x.first=a;x.second=dd;
        g[b].push_back(x); x.first=b;g[a].push_back(x);
        }
    int s ;
    fi>>s;
    vector<int> d (n+1, INF), p (n+1);
    d[s] = 0;
    priority_queue < pair<int,int> > q;
    q.push (make_pair (0, s));
    while (!q.empty()) {
        int v = q.top().second, cur_d = -q.top().first;
        q.pop();
        if (cur_d > d[v]) continue;

        for (size_t j=0; j<g[v].size(); ++j) {
            int to = g[v][j].first,
                len = g[v][j].second;
            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
                q.push (make_pair (-d[to], to));
            }
        }
    }
    for(int i=1;i<=n;++i) if(d[i]<INF) fo<<d[i]<<" "; else fo<<-1<<" ";
    fi>>k;
    for(int i=0;i<k;++i)
    {int t;
    fi>>t;
    vector<int> path;
    fo<<"\nShortets route from "<<s<<" to "<<t<<": ";
    for (int j=t; j!=s; j=p[j])
    path.push_back (j);
    path.push_back (s);
    reverse (path.begin(), path.end());
    for (size_t j=0; j<path.size(); ++j) fo<<path[j]<<" ";
    }

    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```

*Độ phức tạp của giải thuật: O(mlogn).*

### **Ghi chú:**

- ⊕ Với mục đích khảo sát và so sánh hiệu quả cài đặt, chương trình có đưa ra thời gian thực hiện với độ chính xác mili giây,
- ⊕ Thời gian thực hiện chương trình còn có thể giảm xuống nếu thay việc nhập dữ liệu theo kiểu stream bằng nhập theo quy cách và tổ chức vòng tránh sử dụng dữ liệu dạng cặp (Pair).

Kiểu cài đặt dùng cấu trúc dữ liệu Tập hợp tuy kém hiệu quả hơn đôi chút, nhưng cũng đáng để tham khảo.

#### **Chương trình sử dụng cấu trúc dữ liệu kiểu Tập hợp**

```
#include <bits/stdc++.h>
using namespace std;
const int INF = 1000000000;
ifstream fi ("Dijkstra.inp");
ofstream fo ("Dijkstra.out");
pair<int,int>x;
int a,b,dd,k;

int main()
{
    int n,m;
    fi>>n>>m;
    vector < vector < pair<int,int> > > g (n+1);
    for(int i=1;i<=m;++i)
        {fi>>a>>b>>dd;x.first=a;x.second=dd;
        g[b].push_back(x); x.first=b;g[a].push_back(x);
        }
    int s ;
    fi>>s;
    vector<int> d (n+1, INF), p (n+1);
    d[s] = 0;

    set < pair<int,int> > q;
    q.insert (make_pair (d[s], s));
    while (!q.empty())
    {
        int v = q.begin()->second;
        q.erase (q.begin());

        for (size_t j=0; j<g[v].size(); ++j) {
            int to = g[v][j].first,
                len = g[v][j].second;
            if (d[v] + len < d[to])
            {
                q.erase (make_pair (d[to], to));
                d[to] = d[v] + len;
                p[to] = v;
                q.insert (make_pair (d[to], to));
            }
        }
    }
}
```

```

for (int i=1; i<=n; ++i) if (d[i]<INF) fo<<d[i]<<" "; else fo<<-1<<" ";
fi>>k;
for (int i=0; i<k; ++i)
{int t;
fi>>t;
vector<int> path;
fo<<"\n Shortets route from "<<s<<" to "<<t<<": ";
for (int j=t; j!=s; j=p[j])
path.push_back (j);
path.push_back (s);
reverse (path.begin(), path.end());
for (size_t j=0; j<path.size(); ++j) fo<<path[j]<<" ";
}

fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}

```

Khác với PASCAL, cấu trúc dữ liệu Set trong C/C++ hoạt động khá hiệu quả bởi vì nó (và cả cấu trúc dữ liệu Map) được xây dựng dựa trên cơ sở cây tìm kiếm Đỏ – Đen. Tuy vậy, trong phạm vi báo cáo này, ta sẽ *không dùng lại kỹ* ở cơ sở lý thuyết này, mặc dù đó là mảng kiến thức quan trọng làm nền tảng cho việc xây dựng một loạt các giải thuật hiệu quả cao giải quyết các bài toán có mô hình đồ thị.



# Giải thuật Floyd – Warshall

## Bài toán

Xét đồ thị vô hướng không chứa chu trình âm  $n$  đỉnh và  $m$  cạnh. Cạnh  $j$  nối trực tiếp 2 đỉnh  $a_j$  và  $b_j$  với trọng số  $t_j$ ,  $j = 1 \dots m$ .

Cho  $q$  truy vấn, mỗi truy vấn yêu cầu tìm độ dài đường đi ngắn nhất từ  $x$  tới  $y$  ( $1 \leq x, y \leq n$ ) và chỉ ra đường đi trong trường hợp tồn tại. Nếu không có đường đi thì đưa ra số  $-1$ .

**Dữ liệu:** Vào từ file văn bản FLOYD.INP:

- ✚ Dòng đầu tiên chứa 2 số nguyên  $n$  và  $m$  ( $1 \leq n \leq 500$ ,  $0 \leq m \leq n \times (n-1)/2$ ),
- ✚ Dòng thứ  $j$  trong  $m$  dòng sau chứa 3 số nguyên  $a_j$ ,  $b_j$  và  $t_j$  ( $1 \leq a_j, b_j \leq n$ ,  $0 \leq t_j \leq 10^4$ ),
- ✚ Dòng tiếp theo chứa số nguyên  $q$  ( $1 \leq q \leq 10^4$ ),
- ✚ Mỗi dòng trong  $q$  dòng tiếp theo chứa 2 số nguyên  $x$  và  $y$ .

**Kết quả:** Đưa ra file văn bản FLOYD.OUT, với mỗi truy vấn đưa ra số  $-1$  nếu không có đường đi hoặc 2 dòng thông tin: dòng đầu tiên chứa một số nguyên – độ dài đường đi ngắn nhất, dòng thứ 2 chứa các số nguyên xác định đường đi từ  $x$  tới  $y$  (xem ví dụ).

**Ví dụ:**

FLOYD.INP
6 10
1 2 2
1 3 4
1 6 6
2 4 10
2 3 8
6 3 1
3 4 20
6 4 3
5 6 4
5 4 9
3
1 4
5 4
4 1

FLOYD.OUT
8
1 3 6 4
7
5 6 4
8
4 6 3 1

## Giải thuật

Kích thước đồ thị không quá lớn và số truy vấn có thể rất nhiều vì vậy có thể tìm đường đi ngắn nhất giữa 2 đỉnh bất kỳ, từ đó việc xử lý truy vấn đơn thuần chỉ là tra cứu và dẫn xuất độ dài cũng như đường đi ngắn nhất cần tìm.

Giả thiết đồ thị không có chu trình âm.

Giải thuật được R. Floyd và S. Warshall công bố năm 1962. Giải thuật hoạt động trên đồ thị có hướng cũng như vô hướng.

Quá trình tìm kiếm đường đi ngắn nhất được chia thành  $n+1$  giai đoạn.  $d_{i,j}$  lưu độ dài đường đi ngắn nhất từ đỉnh  $i$  tới đỉnh  $j$  ở mỗi giai đoạn.

Ở giai đoạn 0  $d_{i,j} = a_{i,j}$  – độ dài cạnh nối trực tiếp  $i$  với  $j$ .  $d_{i,j} = \infty$  nếu không có đường đi trực tiếp từ  $i$  tới  $j$ .  $d_{i,i} = 0$  với mọi  $i$ .

Ở giai đoạn  $k-1$  ( $k < n$ ),  $d_{i,j}$  là độ dài đường đi ngắn nhất từ  $i$  tới  $j$  có thể qua các đỉnh trung gian từ 1 đến  $k-1$  nếu cần.

Ở giai đoạn  $k$  ( $k \leq n$ ),  $d_{i,j}$  là độ dài đường đi ngắn nhất từ  $i$  tới  $j$  có thể qua các đỉnh trung gian từ 1 đến  $k$  nếu cần và ta có  $d_{i,j} = \min\{d_{i,j}, d_{i,k} + d_{k,j}\}$ .

Sau mỗi giai đoạn ta sẽ có đường đi mới từ  $i$  tới  $j$  tốt hơn đường cũ nếu có một đỉnh trung gian cho phép cải tiến đường đi.

Để khôi phục đường đi ngắn nhất ta phải giữ lại giá trị  $r_{i,j}$  – đỉnh trung gian lớn nhất cần qua trên đường đi từ  $i$  tới  $j$ .

Giải thuật có thể áp dụng cho *đồ thị vô hướng* và *có hướng* (phụ thuộc vào *cách ghi nhận dữ liệu* về cạnh và trọng số của nó).

Độ phức tạp của giải thuật:  $O(n^3)$ .

Số đường đi tối ưu được lưu trữ là  $n \times (n-1)/2$ . Chi phí trung bình cho việc tính một đường đi tối ưu là  $\approx O(n)$ . Vì vậy đây là một giải thuật thích hợp khi cần xét nhiều đường đi khác nhau với  $n$  không quá lớn.

## Chương trình

```
#include <bits/stdc++.h>
using namespace std;
const int INF = 1000000000;
const int N = 501;
ifstream fi ("floyd.inp");
ofstream fo ("floyd.out");
int n,m,x,y,t,q;
int d[N][N], r[N][N]={0};

void route(int u,int v)
{
    if(r[u][v]==0) fo<<v<<' ';
    else{route(u,r[u][v]); route(r[u][v],v);}
}

int main()
{
    fi>>n>>m;
    for(int i=0;i<=n;++i)
        for(int j=0;j<=n;++j)d[i][j]=INF;

    for(int i=0;i<m;++i)
    {
        fi>>x>>y>>t; d[x][y]=t; d[y][x]=t;
    }
    for(int i=1;i<= n;++i)d[i][i]=0;

    for(int k=1;k<=n;++k)
        for(int i=1;i<=n;++i)
            for(int j=1;j<=n;++j)
                if(d[i][j]>d[i][k]+d[k][j])
                    d[i][j]=d[i][k]+d[k][j], r[i][j]=k;

    fi>>q;
    for(int i=0;i<q;++i)
    {
        fi>>x>>y;
        if(d[x][y]==INF) fo<<"-1\n";
        else {fo<<d[x][y]<<' \n'; fo<<x<<' '; route(x,y); fo<<endl;}
    }

    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```

Đồ thị vô hướng

Để dẫn xuất đường đi



# Cây khung nhỏ nhất – Giải thuật Prim

## Bài toán

Xét đồ thị vô hướng có trọng số với  $n$  đỉnh và  $m$  cạnh. Cạnh  $j$  nối trực tiếp 2 đỉnh  $a_j$  và  $b_j$  với trọng số  $w_j$ ,  $j = 1 \dots m$ .

Hãy tìm một cây con của đồ thị thỏa mãn các tính chất:

Tổng trọng số các cạnh là nhỏ nhất,

Giữa 2 đỉnh bất kỳ của đồ thị có một và chỉ một đường đi trong cây con nối chúng.

Cây con thỏa mãn các tính chất trên được gọi là cây con nhỏ nhất.

**Dữ liệu:** Vào từ file văn bản PRIM.INP:

- ⊕ Dòng đầu tiên chứa 2 số nguyên  $n$  và  $m$  ( $1 \leq n \leq 500$ ,  $0 \leq m \leq n \times (n-1)/2$ ),
- ⊕ Dòng thứ  $j$  trong  $m$  dòng sau chứa 3 số nguyên  $a_j$ ,  $b_j$  và  $w_j$  ( $1 \leq a_j, b_j \leq n$ ,  $0 \leq w_j \leq 10^4$ )

**Kết quả:** Đưa ra file văn bản .OUT thông báo “**No MST**” (Không tồn tại cây khung nhỏ nhất) nếu không tồn tại cây cần tìm hoặc  $n-1$  cặp số mỗi cặp trên một dòng và xác định một cạnh của cây con tìm được.

**Ví dụ:**

PRIM.INP
6 10
1 2 2
1 3 4
1 6 6
2 4 10
2 3 8
6 3 1
3 4 20
6 4 3
5 6 4
5 4 9

PRIM.OUT
2 1
3 1
6 3
4 6
5 6

## Giải thuật

Chọn một đỉnh bất kỳ đưa vào cây con,

Chọn cạnh có trọng số nhỏ nhất xuất phát từ đỉnh đã chọn kết nạp vào cây con, bây giờ cây con đã có 2 đỉnh,

Chọn cạnh có trọng số nhỏ nhất, một đỉnh thuộc cây con, đỉnh kia – ngoài cây con, kết nạp cạnh tìm được vào cây con, lặp lại bước này cho đến khi không thể kết nạp thêm cạnh mới,

Nếu kết quả cây con chứa **n-1** cạnh – ta có kết quả cần tìm, trong trường hợp ngược lại – không tồn tại lời giải.

Tổ chức dữ liệu:

- Mảng `int` `g[N][N]` lưu trọng số các cạnh,
- Mảng `vector<bool>` `used(n)` đánh dấu các đỉnh được chọn,
- Mảng `vector<int>` `min_e(n, INF)` lưu trọng số nhỏ nhất có thể chọn từ một đỉnh,
- Mảng `vector<int>` `sel_e(n, -1)` cạnh của cây khung.

Dộ phức tạp của giải thuật:  $O(n^2)$ .

## Chương trình

```
#include <bits/stdc++.h>
using namespace std;
const int INF = 1000000000;
const int N = 501;
ifstream fi ("prim.inp");
ofstream fo ("prim.out");
int n,m,a,b,w;
int g[N][N];

int main()
{
    fi>>n>>m;
    for(int i=0; i<=n; ++i)
        for(int j=0; j<=n; ++j) g[i][j]=INF;
    for(int i=0; i<m; ++i)
    {
        fi>>a>>b>>w; --a; --b;
        g[a][b]=w; g[b][a]=w;
    }
    vector<bool> used (n);
    vector<int> min_e (n, INF), sel_e (n, -1);
    min_e[0] = 0;
    for (int i=0; i<n; ++i)
    {
        int v = -1;
        for (int j=0; j<n; ++j)
            if (!used[j] && (v == -1 || min_e[j] < min_e[v])) v = j;
        if (min_e[v] == INF)
        {
            cout << "No MST!";
            exit(0);
        }
        used[v] = true;
        if (sel_e[v] != -1)
            fo<< v+1 << " " << sel_e[v]+1 << endl;
        for (int to=0; to<n; ++to)
            if (g[v][to] < min_e[to])
            {
                min_e[to] = g[v][to];
                sel_e[to] = v;
            }
    }
}
```

## Giải thuật độ phức tạp O(mlogn)

Trong phần lớn các bài toán đồ thị cần xét khá thưa. Việc giữ đầy đủ ma trận kề sẽ làm tăng độ phức tạp của giải thuật và không khả thi khi n lớn.

Độ phức tạp của giải thuật đã xét phụ thuộc nhiều vào khâu tìm cạnh (và đỉnh) tiếp theo có thể nạp vào cây khung.

Để giảm độ phức tạp của giải thuật và hạn chế khối lượng bộ nhớ sử dụng trong phạm vi khả thi cần có các thay đổi sau trong tổ chức dữ liệu:

- + Dùng các mảng động `vector<vector<pair<int, int>>>` g(n) lưu cấu trúc đồ thị, trong đó `g[i][j]` lưu cặp dữ liệu (`v, w`), `w` là trọng số của cạnh `i → v`,
- + Tập `set<pair<int, int>>` q lưu thông tin về các đỉnh tiếp theo cần xét để kết nạp vào cây khung.

Các dữ liệu còn lại và cách xử lý: không thay đổi.

*Độ phức tạp của giải thuật: O(mlogn).*

## Chương trình

```
#include <bits/stdc++.h>
using namespace std;
const int INF = 1000000000;
ifstream fi ("prim.inp");
ofstream fo ("prim.out");

int main()
{
    int n,m,a,b,w;
    fi>>n>>m;
    vector < vector < pair<int,int> > > g(n);
    for(int i=0;i<m;++i)
    {
        fi>>a>>b>>w;--a;--b;
        g[a].push_back({b,w});
        g[b].push_back({a,w});
    }
    vector<bool> used (n, 0);
    vector<int> min_e (n, INF), sel_e (n, -1);
    min_e[0] = 0;
    set < pair<int,int> > q;
    q.insert ({0, 0});
    for (int i=0; i<n; ++i)
    {
        if (q.empty())
        {
            fo << "No MST!";
            exit(0);
        }
        int v = q.begin() ->second;
        used[v]=true;
        q.erase (q.begin());
        if (sel_e[v] != -1)
            fo << v+1 << " " << sel_e[v]+1 << endl;
        for (size_t j=0; j<g[v].size(); ++j)
        {
            int to = g[v][j].first,
            cost = g[v][j].second;
            if (cost < min_e[to] && !used[to])
            {
                q.erase ({min_e[to], to});
                min_e[to] = cost;
                sel_e[to] = v;
                q.insert ({min_e[to], to});
            }
        }
    }
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```

## Tính chất cây khung nhỏ nhất và giải thuật

Cây khung nhỏ nhất còn thường được gọi là cây khung cực tiểu,

Cây khung cực đại cũng có thể dễ dàng xác định bằng chính các chương trình trên: trong dữ liệu (hoặc trong chương trình) thay trọng số  $w$  bằng  $-w$ . Sự tồn tại chu trình âm không ảnh hưởng tới giải thuật,

Nếu *trọng số các cạnh khác nhau* thì cây khung tối thiểu là *duy nhất*.

Cây khung tối thiểu theo trọng số cũng là cây khung *tối thiểu theo tích các trọng số* nếu các trọng số đều lớn hơn 0, dễ dàng thấy được điều này nếu ta thay trọng số bằng logarit của nó và tìm cây khung với giá trị trọng số thực,

Cây khung nhỏ nhất cũng là cây khung *cực tiểu của trọng số cạnh lớn nhất*,

*Tiêu chuẩn nhận dạng cây khung cực tiểu*: Một cây khung là cực tiểu khi và chỉ khi với cạnh bất kỳ không thuộc cây khung, chu trình chứa cạnh đó hình thành khi xét nó cùng với cây khung không chứa cạnh có trọng số lớn hơn cạnh bất kỳ đang xét.



## Tìm cha chung gần nhất

Cho cây  $G$  và hai đỉnh  $v_1, v_2$ . Đường đi từ gốc tới  $v_1$  và từ gốc tới  $v_2$  sẽ có một hoặc một số đỉnh chung. Các đỉnh chung đó là cha đồng thời của  $v_1$  và  $v_2$ . Nút cha chung cuối cùng gặp trên 2 đường đi là nút cha chung gần nhất (*Least Common Ancestor – LCA*).

### Bài toán

Xét cây có  $n$  đỉnh được cho bởi danh sách cạnh  $(a_i, b_i)$ ,  $i = 1 \dots n-1$ . Cho  $m$  truy vấn, mỗi truy vấn chứa 2 số nguyên  $x$  và  $y$  xác định 2 đỉnh của cây. Với mỗi truy vấn hãy xác định cha chung gần nhất của 2 đỉnh đã cho.

**Dữ liệu:** Vào từ file văn bản LCA.INP:

- ⊕ Dòng đầu tiên chứa một số nguyên  $n$  ( $1 \leq n \leq 10^5$ ),
- ⊕ Mỗi dòng trong  $n-1$  dòng tiếp theo chứa 2 số nguyên  $a$  và  $b$  xác định một cạnh của cây ( $1 \leq a, b \leq n$ ,  $a \neq b$ ).
- ⊕ Dòng tiếp theo chứa số nguyên  $m$  ( $1 \leq m \leq 10^5$ ),
- ⊕ Mỗi dòng trong  $m$  dòng tiếp theo chứa 2 số nguyên  $x$  và  $y$  ( $1 \leq x, y \leq n$ ).

**Kết quả:** Đưa ra file văn bản LCA.OUT, kết quả mỗi truy vấn đưa ra trên một dòng dưới dạng số nguyên.

**Ví dụ:**

LCA.INP
8
1 2
2 4
4 3
6 3
8 3
5 4
7 4
3
6 4
6 8
8 7

LCA.OUT
4
3
4

## Giải thuật

Việc phân loại nút trên cây được thực hiện dựa trên 3 tham số:

- ✚ Thời điểm vào **tin [x]**,
- ✚ Thời điểm ra **tout [x]**,
- ✚ Độ sâu của đỉnh **depth [x]**.

Các phương pháp **tính** và **lưu trữ khác nhau** các đại lượng trên (hoặc **đại lượng dẫn xuất tương đương**) đưa đến việc hình thành những **giải thuật khác nhau** tìm LCA.

Ở đây ta sẽ xét phương pháp tìm LCA với các đặc trưng:

- ♣ Các **thời điểm ra** của các **đỉnh khác nhau** là **khác nhau**,
- ♣ **Độ sâu** của một đỉnh cho biết nó là **con thứ mấy của gốc**, để thuận tiện tìm nút cha ta cần biết và lưu trữ thông tin ngược lại – **cha thứ k của một đỉnh** là đỉnh nào.

Điều quyết định tính hiệu quả của giải thuật:

- ✓ Lưu trữ thông tin ở dưới dạng có thể nhanh chóng **tra cứu** kết quả cho từng trường hợp cụ thể,
- ✓ Đảm bảo tính đối xứng giải thuật trong lưu trữ: Cách tra cứu là như nhau với mọi đầu vào khác nhau trên cùng một tập dữ liệu được lưu trữ,
- ✓ Khối lượng thông tin cần lưu trữ là đủ nhỏ.

Điểm mấu chốt của giải thuật này là bảng **up** kích thước  $n \times p$ , trong đó **p** là số nguyên nhỏ nhất thỏa mãn  $n \leq 2^p$ .

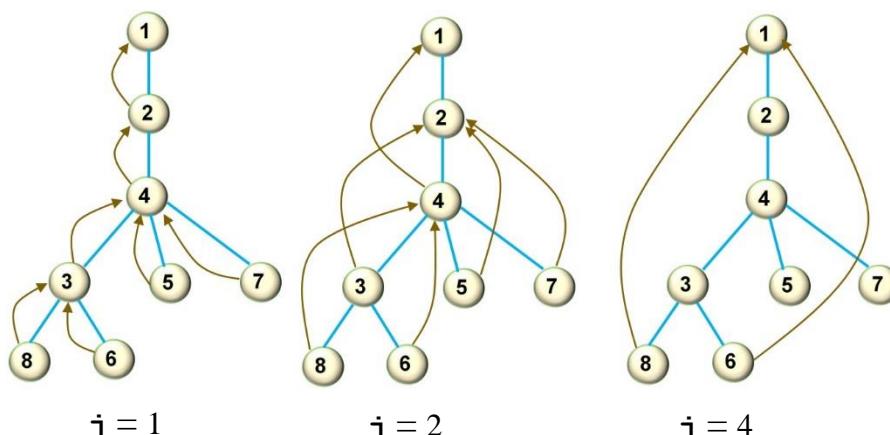
$up_{i,j}$  xác định cha thứ  $2^j$  của đỉnh **i**,  $j = 0, 1, 2, \dots$ . Với một đỉnh **i** có ít hơn  $2^j$  cha thì  $up_{i,j} = 0$  (quy về đỉnh gốc). Vì vậy  $up_{0,j} = 0$  với mọi **j**.

Nếu cây chứa cạnh **a → b** thì  $up_{b,0} = a$ .

Mảng **up** được xây dựng theo nguyên tắc quy hoạch động đơn giản:

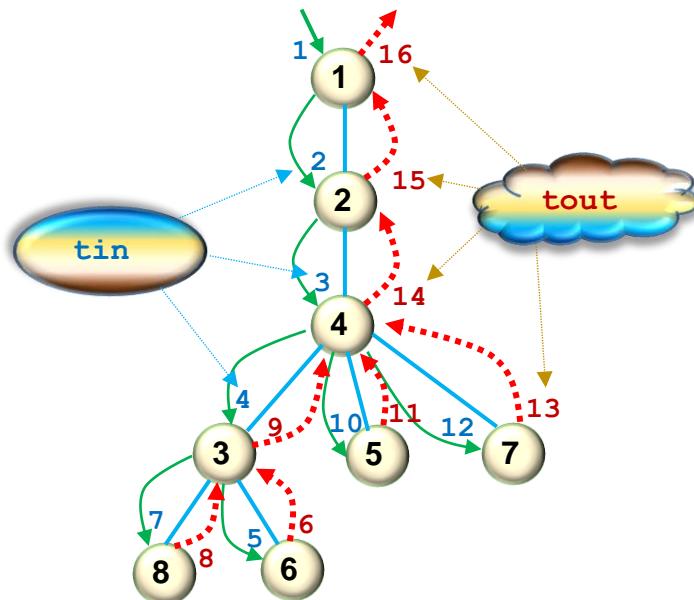
$$up_{b,0} = a \text{ nếu có } a \rightarrow b,$$

$$up_{i,j} = up_{up_{i,j-1}, j-1}$$



Tiêu chuẩn nhận dạng **a** là nút cha của **b**:

$$tin_a \leq tin_b \leq tout_b \leq tout_a$$



Tổ chức dữ liệu:

- ▀ Mảng `vector < vector<int> >` g lưu cấu trúc cây,
- ▀ Các `vector<int>` tin, tout mảng lưu thời điểm vào và ra,
- ▀ Mảng `vector < vector<int> >` up xác định nút cha theo mức  $2^j$  từ đỉnh con.

Xử lý:

Ghi nhận dữ liệu,

Xin cấp phát bộ nhớ

```
fi>>n;
g.resize(n);
for(int i=0; i<n-1; ++i)
{
    fi>>x>>y; --x; --y;
    g[x].push_back(y); g[y].push_back(x);
}
```

Xin cấp phát bộ nhớ cho các mảng:

```
tin.resize (n), tout.resize (n), up.resize (n);
l = 1;
while ((1<<l) < n) ++l;
for (int i=0; i<n; ++i) up[i].resize (l+1);
```

$$2^{l+1} \geq n$$

Chuẩn bị các tham số: lời gọi **dfs(0)** ;

```
void dfs (int v, int p = 0)
{
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i=1; i<=l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];
    for (auto &to:g[v]) if (to != p) dfs (to, v);
    tout[v] = ++timer;
}
```

*Tính cha thứ  $2^i$*

Xử lý một truy vấn:

```
int lca (int a, int b)
{
    if (upper (a, b)) return a;
    if (upper (b, a)) return b;
    for (int i=l; i>=0; --i)
        if (! upper (up[a][i], b)) a = up[a][i];
    return up[a][0];
}
```

*Cha chung không trùng với a và b*

Độ phức tạp của giải thuật:  $O(m\log n)$ .

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "lca."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100000;
int n,m,l,x,y, timer;
vector < vector<int> > g;
vector<int> tin, tout;
vector < vector<int> > up;

void dfs (int v, int p = 0)
{
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i=1; i<=l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];
    for (auto &to:g[v]) if (to != p) dfs (to, v);
    tout[v] = ++timer;
}
bool upper (int a, int b)
{
    return tin[a] <= tin[b] && tout[a] >= tout[b];
}
int lca (int a, int b)
{
    if (upper (a, b)) return a;
    if (upper (b, a)) return b;
    for (int i=l; i>=0; --i)
        if (! upper (up[a][i], b)) a = up[a][i];
    return up[a][0];
}

int main()
{
    fi>>n;
    g.resize(n);
    for(int i=0;i<n-1;++i)
    {
        fi>>x>>y; --x; --y;
        g[x].push_back(y); g[y].push_back(x);
    }
    tin.resize (n), tout.resize (n), up.resize (n);
    l = 1;
    while ((1<<l) <= n) ++l;
    for (int i=0; i<n; ++i) up[i].resize (l+1);
    dfs (0);

    fi>>m;
    for(int i=0;i<m;++i)
    {
        fi>>x>>y; --x; --y;
        int res = lca (x, y)+1;
        fo<<res<<'\n';
    }
    fo<<"\nTime: "<<clock() / (double) 1000<<" sec";
}
```

## Giải thuật Farach-Colton – Bender

Farach-Colton và Bender công bố giải thuật tìm LCA với chi phí thời gian chuẩn bị  $O(n)$  và thời gian xử lý mỗi truy vấn là  $O(1)$ .

Đây là *giải thuật hiệu quả nhất* và độ khó lập trình không cao.

### Giải thuật

Bài toán tìm LCA có thể dẫn xuất về bài toán tìm RMQ (Cực tiểu trên đoạn thẳng).

Bài toán RMQ nhận được rất đặc thù: hai phần tử kề nhau trong mảng luôn chỉ khác biệt một đơn vị (vì các phần tử trong mảng là độ sâu của đỉnh theo trình tự duyệt, nếu đi từ đỉnh cha xuống đỉnh con – độ sâu tăng thêm 1, ngược lại, di từ đỉnh con sang đỉnh cha – độ sâu giảm 1). Giải thuật đang xét đã tận dụng tính chất này để nâng cao hiệu quả xử lý.

Gọi A là mảng chứa thông tin phục vụ các truy vấn RMQ và N là kích thước của A.

Đầu tiên xét giải thuật xử lý RMQ với *chi phí chuẩn bị là  $O(N \log N)$*  và *thời gian xử lý mỗi truy vấn là  $O(1)$* . Tạo bảng thưa (Sparse Table)  $T[l][i]$ , trong đó  $T[l][i]$  là min của A trong khoảng  $[l, l+2^i]$ . Rõ ràng kích thước của bảng có bậc  $O(N \log N)$ . Bảng có thể được xây dựng với chi phí thời gian  $O(N \log N)$  vì

$$T[l][i] = \min\{T[l][i-1], T[l+2^{i-1}][i-1]\}$$

Với truy vấn  $(l, r)$  kết quả cần tìm sẽ là  $\min\{T[l][sz], T[r-2^{sz}+1][sz]\}$ , trong đó sz là lũy thừa lớn nhất của 2 không vượt quá  $r-l+1$ . Để thực sự có chi phí thời gian  $O(1)$  cần chuẩn bị sẵn bảng sz cho các giá trị từ 1 tới N.

Xét việc cải tiến giải thuật để nhận được chi phí  $O(N)$  cho công tác chuẩn bị.

Chia mảng A thành các khối kích thước  $K = 0.5 \log_2 N$ . Với mỗi khối – tìm phần tử min và vị trí của nó (vì với LCA quan trọng là vị trí). Gọi B là mảng kích thước  $N/K$  lưu các min tìm được trong mỗi khối. Xây dựng bảng thưa đối với nó theo cách đã nêu. Kích thước và chi phí thời gian xây dựng bảng sẽ là:

$$\begin{aligned} \frac{N}{K} \log \frac{N}{K} &= (2N / \log N) \log (2N / \log N) = \\ &= (2N / \log N) (1 + \log (N / \log N)) \leq 2N / \log N + 2N = O(N) \end{aligned}$$

Nếu truy vấn  $(l, r)$  bao gồm nhiều khối thì kết quả sẽ là min phần cuối khối tương ứng với l, lấy min tiếp với các khối nằm giữa l và r, cuối cùng – lấy min tiếp với phần đầu khối tương ứng với r. Bảng thưa T cho phép thực hiện các công việc đó với chi phí  $O(1)$ .

Xét trường hợp khi  $(l, r)$  nằm gọn trong một khối.

Lưu ý tính chất hơn kém nhau 1 đã nêu ở trên. Nếu lấy các phần tử trong mỗi khối trừ đi phần tử đầu của khối, ta được dãy số  $\pm 1$  độ dài  $K-1$ . Như vậy số lượng các khối khác nhau sẽ là

$$2^{K-1} = 2^{0.5 \log N - 1} = 0.5 \sqrt{N}$$

Như vậy, số lượng các khối khác nhau có bậc  $O(\sqrt{N})$ . Ta có thể tính trước kết quả trong khối ở tất cả các khối với chi phí  $O(\sqrt{N}K^2) = O(\sqrt{N}\log^2 N) = O(N)$ . Trên thực tế, mỗi khối được đặc trưng bởi dãy bít độ dài  $K-1$  và có thể lưu trữ như một số dạng int và các kết quả RMQ đã tính có thể lưu trong bảng  $R[\text{mask}][l][r]$  kích thước  $O(\sqrt{N}\log^2 N)$ .

Như vậy trong mọi trường hợp chi phí tìm kiếm chỉ là  $O(1)$ .

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "lca_Bender."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int MAXN = 100000;
const int MAXLIST = MAXN * 2;
const int LOG_MAXLIST = 18;
const int SQRT_MAXLIST = 447;
const int MAXBLOCKS = MAXLIST / ((LOG_MAXLIST+1)/2) + 1;
int n,m,x,y;
vector<int> g[MAXN];
int h[MAXN]; // vertex height
vector<int> a; // dfs list
int a_pos[MAXN]; // positions in dfs list
int block; // block size = 0.5 log A.size()
int bt[MAXBLOCKS][LOG_MAXLIST+1]; // sparse table on blocks (relative minimum
positions in blocks)
int bhash[MAXBLOCKS]; // block hashes
int brmq[SQRT_MAXLIST][LOG_MAXLIST/2][LOG_MAXLIST/2]; // rmq inside each
block, indexed by block hash
int log_2[2*MAXN]; // precalced logarithms (floored values)
// walk graph
void dfs (int v, int curh)
{
    h[v] = curh;
    a_pos[v] = (int)a.size();
    a.push_back (v);
    for (size_t i=0; i<g[v].size(); ++i)
        if (h[g[v][i]] == -1)
        {
            dfs (g[v][i], curh+1);
            a.push_back (v);
        }
}
int log (int n)
{
    int res = 1;
    while (1<<res < n) ++res;
    return res;
}
// compares two indices in a inline
int min_h (int i, int j)
{
    return h[a[i]] < h[a[j]] ? i : j;
}
// O(N) preprocessing
```

```

void build_lca()
{
    int sz = (int)a.size();
    block = (log(sz) + 1) / 2;
    int blocks = sz / block + (sz % block ? 1 : 0);
// precalc in each block and build sparse table
    memset(bt, 255, sizeof bt);
    for (int i=0, bl=0, j=0; i<sz; ++i, ++j)
    {
        if (j == block)
            j = 0, ++bl;
        if (bt[bl][0] == -1 || min_h(i, bt[bl][0]) == i) bt[bl][0] = i;
    }
    for (int j=1; j<=log(sz); ++j)
        for (int i=0; i<blocks; ++i)
        {
            int ni = i + (1<<(j-1));
            if (ni >= blocks) bt[i][j] = bt[i][j-1];
            else bt[i][j] = min_h(bt[i][j-1], bt[ni][j-1]);
        }
// calc hashes of blocks
    memset(bhash, 0, sizeof bhash);
    for (int i=0, bl=0, j=0; i<sz || j<block; ++i, ++j)
    {
        if (j == block) j = 0, ++bl;
        if (j > 0 && (i >= sz || min_h(i-1, i) == i-1)) bhash[bl] += 1<<(j-1);
    }
// precalc RMQ inside each unique block
    memset(brmq, 255, sizeof brmq);
    for (int i=0; i<blocks; ++i)
    {
        int id = bhash[i];
        if (brmq[id][0][0] != -1) continue;
        for (int l=0; l<block; ++l)
        {
            brmq[id][l][1] = l;
            for (int r=l+1; r<block; ++r)
            {
                brmq[id][l][r] = brmq[id][l][r-1];
                if (i*block+r < sz)
                    brmq[id][l][r] = min_h(i*block+brmq[id][l][r], i*block+r)
- i*block;
            }
        }
    }
// precalc logarithms
    for (int i=0, j=0; i<sz; ++i)
    {
        if (1<<(j+1) <= i) ++j;
        log_2[i] = j;
    }
}
// answers RMQ in block #bl [l;r] in O(1)
inline int lca_in_block (int bl, int l, int r)
{
    return brmq[bhash[bl]][l][r] + bl*block;
}
// answers LCA in O(1)

```

```

int lca (int v1, int v2)
{
    int l = a_pos[v1], r = a_pos[v2];
    if (l > r) swap (l, r);
    int bl = l/block, br = r/block;
    if (bl == br) return a[lca_in_block(bl,l%block,r%block)];
    int ans1 = lca_in_block(bl,l%block,block-1);
    int ans2 = lca_in_block(br,0,r%block);
    int ans = min_h (ans1, ans2);
    if (bl < br - 1)
    {
        int pw2 = log_2[br-bl-1];
        int ans3 = bt[bl+1][pw2];
        int ans4 = bt[br-(1<<pw2)][pw2];
        ans = min_h (ans, min_h (ans3, ans4));
    }
    return a[ans];
}

int main()
{
    fi>>n;
    for(int i=0;i<n-1;++i)
    {
        fi>>x>>y; --x; --y;
        g[x].push_back(y); g[y].push_back(x);
        h[x]=-1; h[y]=-1;
    }

    dfs (0,-1); build_lca();

    fi>>m;
    for(int i=0;i<m;++i)
    {
        fi>>x>>y; --x; --y;
        int res = lca (x, y)+1;
        fo<<res<<'\n';
    }
    fo<<"\nTime: "<<clock () / (double) 1000<<" sec";
}

```



# CẤU TRÚC CÂY

## Cây Fenwick

Cây Fenwick là cấu trúc dữ liệu quản lý tổng tiền tố (Prefix Sum).

Đặc điểm và khả năng của cây Fenwick:

- ✚ Chỉ sử dụng bộ nhớ đúng bằng bộ nhớ lưu trữ mảng cần xử lý,
- ✚ Các phép xử lý insert\_fw (Bổ sung), remove\_fw (Xóa), sum\_fw (Tính tổng trong khoảng) đều có độ phức tạp  $O(\log n)$ ,
- ✚ Dễ dàng thay đổi giá trị bất kỳ của mảng xử lý với độ phức tạp  $O(\log n)$ ,
- ✚ Làm việc cả trong chế độ xử lý offline và online,
- ✚ Dễ dàng mở rộng để xử lý mảng nhiều chiều.

Cây Fenwick lần đầu tiên được mô tả trong bài báo “*A new data structure for cumulative frequency tables*” của Peter M Fenwick 1994.

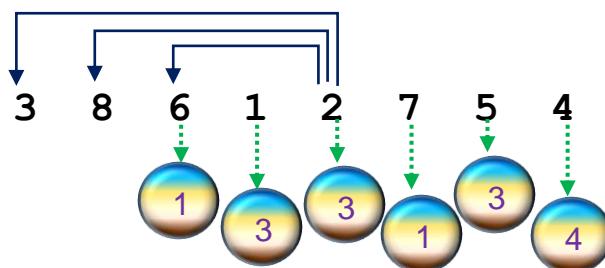
Chi tiết hơn nữa về cấu trúc dữ liệu này có thể xem ở tập I. Ở đó, các công thức biến đổi tham số điều kiện đã được trình bày và chứng minh. Tại đây ta chỉ xem xét việc triển khai các **hàm xử lý** và **ứng dụng**. Với mục đích tiết kiệm thời gian các hàm được nêu ngay trong ví dụ ứng dụng.

### Bài toán 1.

#### Tính số nghịch thế của hoán vị

Cho  $n$  và hoán vị  $\mathbf{A}$  các số từ 1 đến  $n$ ,  $\mathbf{A} = (a_1, a_2, \dots, a_n)$ . Hãy tính và đưa ra số lượng nghịch thế trong hoán vị  $\mathbf{A}$ . Nghịch thế là cặp giá trị  $a_i$  và  $a_j$  thỏa mãn các điều kiện  $a_i > a_j$  và  $i < j$ . Nói một cách khác, nghịch thế là cặp giá trị mà số lớn hơn đứng trước số bé trong dãy.

Ví dụ,  $n = 8$  và  $\mathbf{A} = (3, 8, 6, 1, 2, 7, 5, 4)$ ,  $a_5 = 2$  tạo thành 3 nghịch thế (với các số 3, 8 và 6)



**Dữ liệu:** Vào từ file văn bản INV\_NUM.INP:

- ✚ Dòng đầu tiên chứa một số nguyên  $n$  ( $1 \leq n \leq 10^5$ ),

- Dòng thứ 2 chứa  $n$  số nguyên  $a_1, a_2, \dots, a_n$  – xác định một hoán vị các số từ 1 đến  $n$ .

**Kết quả:** Đưa ra file văn bản INV\_NUM.OUT một số nguyên – số lượng nghịch thế trong hoán vị đã cho.

Ví dụ:

INV_NUM.INP	INV_NUM.OUT
8	15
3 8 6 1 2 7 5 4	

### Giải thuật

Có hai cách xử lý:

- Kiểu offline: đọc toàn bộ hoán vị, sau đó tính số nghịch thế,
- Kiểu online: lần lượt đọc các phần tử của hoán vị và cập nhật số nghịch thế.

Kiểu xử lý online cho phép ta không cần lưu bản thân hoán vị. Xử lý online được áp dụng trong các bài toán tương tác người – máy, các bài toán xử lý truy vấn, xử lý dữ liệu động (theo kích thước hoặc giá trị). Thông thường, xử lý online phức hơn phương thức offline. Tuy nhiên, trong trường hợp này, độ phức tạp (của giải thuật và của lập trình) là tương đương.

### Xử lý offline

Đọc và lưu toàn bộ giá trị và mảng  $\mathbf{A}$ ,

Nạp phần tử cuối cùng ( $a_n$ ) vào cây Fenwick,

Với mỗi phần tử  $a_i$  tiếp theo,  $i = n-2 \div 1$ : nạp vào cây và cập nhật số nghịch thế. Số nghịch thế của  $a_i$  sẽ là số lượng số nhỏ hơn  $a_i$  đã có trong cây. Theo sơ đồ chuẩn (tham khảo phần lý thuyết, Tập I) cây Fenwick tích lũy kết quả từ 1 trở đi, nên ta sẽ thống kê số phần tử không lớn hơn  $a_i$  và trừ 1 (loại bản thân số  $a_i$ ).

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "inv_num."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");

const int N = 100001;
int n, a[N], fw[N]={0};
int64_t res=0;
```

```
void insert_fw(int x)
{
    while (x<=n) {++fw[x]; x+=x&(-x);}
}
```

Công thức biến đổi  
tham số khi nạp vào cây

Sơ đồ lặp nhanh hơn  
để quy

```
int64_t sum_fw(int x)
{
    int r=0;
    while (x>0) {r+=fw[x]; x&=(x-1);}
    return r;
}
```

Sơ đồ lặp nhanh hơn  
để quy

```
int main()
{
    fi>>n;
    for(int i=1;i<=n;++i) fi>>a[i];
    insert_fw(a[n]);
    for(int i=n-1;i>0;--i)
    {
        insert_fw(a[i]);
        res+=sum_fw(a[i])-1;
    }
    fo<<res;
    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}
```

Công thức biến đổi  
tham số khi duyệt cây

## Xử lý online

Đọc từng phần tử hoán vị, xử lý và bỏ qua dữ liệu vào (không lưu lại),

Với dữ liệu thứ **i** đọc được, gọi hàm **sum\_fw()** ta sẽ có số lượng thuận thế, số lượng nghịch thế sẽ là phần bù với **i**.

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "inv_num_ol."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100001;
int n,a,fw[N]={0};
int64_t res=0;

void insert_fw(int x)
{
    while (x<=n) {++fw[x]; x+=x&(-x);}
}

int64_t sum_fw(int x)
{
    int r=0;
    while (x>0) {r+=fw[x]; x&=(x-1);}
    return r;
}

int main()
{
    fi>>n;
    for(int i=1;i<=n;++i)
    {
        fi>>a;
        insert_fw(a);
        res+=i-sum_fw(a);
    }
    fo<<res;
    fo<<"\nTime: "<<clock() / (double) 1000<<" sec";
}
```

### Bài toán 2.

#### Tính tổng trên đoạn với dãy số thay đổi

Cho  $n$ , mảng số nguyên  $a_1, a_2, \dots, a_n$  và  $m$  truy vấn, mỗi truy vấn có một trong 2 dạng sau:

- 1  $p$   $v$  – Thay giá trị  $a_p$  bằng  $v$  ( $1 \leq p \leq n, |v| \leq 10^9$ ),
- 2  $l$   $r$  – Yêu cầu tính tổng các  $a_i$  với  $i = l \div r$  ( $1 \leq l \leq r \leq n$ ),

Với mỗi truy vấn loại 2 hãy đưa ra tổng tìm được.

*Dữ liệu:* Vào từ file văn bản SUM\_FW.INP:

- ─ Dòng đầu tiên chứa một số nguyên  $n$  ( $1 \leq n \leq 10^5$ ),
- ─ Dòng thứ 2 chứa  $n$  số nguyên  $a_1, a_2, \dots, a_n$  ( $|a_i| \leq 10^9$ ,  $i = 1 \div n$ ),
- ─ Dòng thứ 3 chứa số nguyên  $m$  ( $1 \leq m \leq 10^5$ ),
- ─ Mỗi dòng trong  $m$  dòng sau chứa 3 số nguyên xác định một truy vấn.

**Kết quả:** Đưa ra file văn bản SUM\_FW.OUT với mỗi truy vấn loại 2 đưa ra một số nguyên – tổng tính được, mỗi kết quả đưa ra trên một dòng.

**Ví dụ:**

SUM_FW.INP	SUM_FW.OUT
8	15
3 8 6 1 2 7 5 4	4
3	
2 2 4	
1 2 -6	
2 1 4	

*Xử lý*

Tính tổng: lưu ý trường hợp đoạn cần tính bắt đầu từ phần tử đầu tiên,

Cập nhật dãy: Cập nhật vào mảng chứa tổng giá trị v-ap bắt đầu từ vị trí p và lưu giá trị mới v vào mảng ban đầu.

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "sum_fw."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100001;
int n,m,k,t,l,r;
int64_t b,a[N],fw[N]={0};
int64_t res=0;

void insert_fw(int x,int v)
{
    while (x<=n) { fw[x] += v; x+=x&(-x); }
}

int64_t sum_fw(int x)
{
    int r=0;
    while (x>0) { r+=fw[x]; x&=(x-1); }
    return r;
}

int main()
{
    fi>>n;
    for(int i=1;i<=n;++i)
    {
        fi>>a[i];
        insert_fw(i,a[i]);
    }
}
```

```

    }

fi>>m;
for(int i=0; i<m;++i)
{
    fi>>t;
    if(t==1)
    {
        fi>>l>>b;
        insert_fw(l,b-a[l]);
        a[l]=b;
    }
    else
    {
        fi>>l>>r;
        if(l==1)b=0; else b=sum_fw(l-1);
        res=sum_fw(r)-b;
        fo<<res<<'`n';
    }
}

fo<<"`nTime: "<<clock() / (double)1000<<" sec";
}

```

## Mở rộng nhiều chiều

Bộ nhớ dành cho cây vẫn đúng bằng bộ nhớ dành cho mảng dữ liệu vào. Mảng dữ liệu vào chỉ cần lưu trữ khi có các phép truy vấn cập nhật có liên quan tới dữ liệu cũ.

Thay vì một chu trình trong các phép xử lý bây giờ phải có chu trình với độ lồng đúng bằng số chiều.

Ví dụ minh họa:

Xét mảng 2 chiều **A** kích thước  $n \times m$  (**n** dòng, **m** cột), mỗi phần tử của mảng là một số nguyên có giá trị tuyệt đối không vượt quá  $10^7$ . Cho **k** truy vấn, mỗi truy vấn yêu cầu tính tổng các phần tử của mảng con xác định bởi hai vị trí đối  $(1, 1)$  và  $(p, q)$ .

**Dữ liệu:** Vào từ file văn bản FW\_2D.INP:

- ✚ Dòng đầu tiên chứa 2 số nguyên **n** và **m** ( $1 \leq n, m \leq 1000$ ),
- ✚ Mỗi dòng trong **n** dòng tiếp theo chứa **m** số nguyên xác định một dòng của mảng,
- ✚ Dòng tiếp theo chứa số nguyên **k** ( $1 \leq k \leq 1000$ ),
- ✚ Mỗi dòng trong **k** dòng tiếp theo chứa 2 số nguyên **p** và **q** xác định một truy vấn ( $1 \leq p \leq n, 1 \leq q \leq m$ ).

**Kết quả:** Đưa ra file văn bản FW\_2D.OUT, kết quả mỗi truy vấn đưa ra trên một dòng.

Ví dụ:

FW_2D.INP	FW_2D.OUT
4 5	
1 2 3 4 5	
2 3 4 5 6	
3 4 5 6 7	
4 5 6 7 8	
2	
2 3	
3 3	

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "fw_2d."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 1000;
int n,m,k,l,r;
int64_t a[1000][1000], fw[1000][1000];
int64_t t,res=0;
```

```

void insert_fw(int x,int y,int64_t v)
{
    for(int i=x;i<=n;i+=i&(-i))
        for(int j=y;j<=m;j+=j&(-j)) fw[i][j]+=v;
}

int64_t sum_fw(int x,int y)
{
    int r=0;
    for(int i=x;i>0;i&=(i-1))
        for(int j=y;j>0;j&=(j-1)) r+=fw[i][j];
    return r;
}

int main()
{
    fi>>n>>m;
    for(int i=1;i<=n;++i)
    {
        for(int j=1;j<=m;++j)
        {
            fi>>a[i][j];
            insert_fw(i,j,a[i][j]);
        }
    }

    fi>>k;
    for(int i=0;i<k;++i)
    {int p,q;
        fi>>p>>q;
        res=sum_fw(p,q);
        fo<<res<<'\\n';
    }

    fo<<"\\nTime: "<<clock() / (double)1000<<" sec";
}

```



## Cây quản lý đoạn

Cây quản lý đoạn là một cấu trúc dữ liệu đơn giản trong tổ chức, linh hoạt về chức năng, hiệu quả trong xử lý và không đòi hỏi nhiều bộ nhớ.

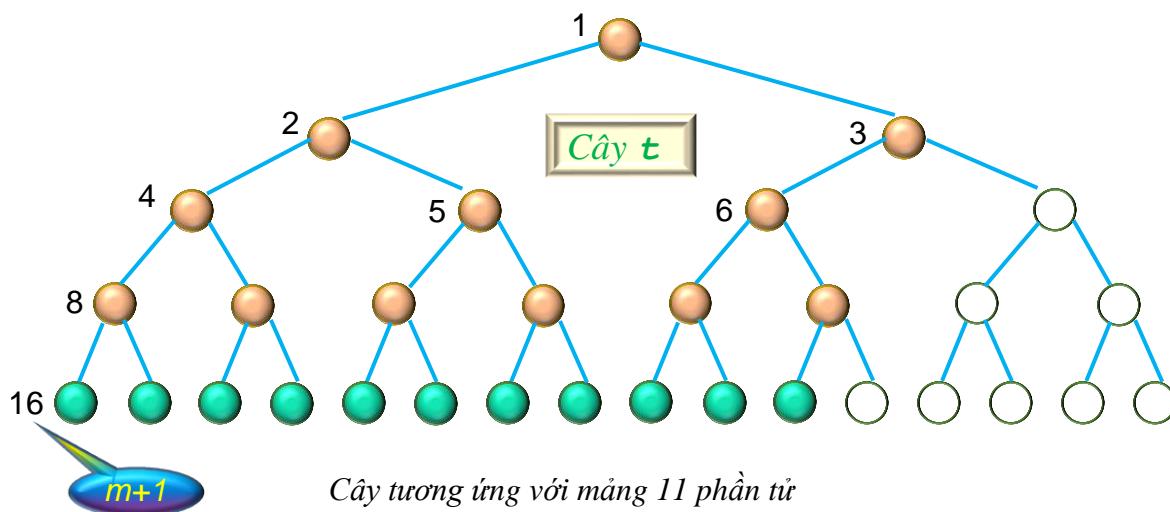
Cấu trúc cây cần  $4 \times n$  phần tử để lưu trữ và xử lý mảng  $n$  phần tử.

Cây cho phép tìm tổng các phần tử, tìm min/max trong đoạn  $[1 \dots r]$ , thay đổi giá trị một hoặc một số phần tử liên tiếp nhau của mảng, ... Về mặt lý thuyết, số các phép xử lý khác nhau có thể thực hiện là không hạn chế!

Phần lớn các phép xử lý đều có độ phức tạp  $O(\log n)$ .

### Cấu trúc cây

Các nút của cây được đánh số từ 1 trở đi, nút 1 là gốc của cây. Trừ các nút lá không có nút con, mỗi nút còn lại của cây (*nút trong*) có đúng 2 nút con.



Nút trong  $u$  có 2 nút con là  $2 \times u$  và  $2 \times u + 1$ . Trừ nút gốc, mỗi nút còn lại (kể cả nút lá) đều có nút cha. Nút  $u$  có cha là nút  $u/2$ .

Các nút lá chứa dữ liệu của dãy ban đầu.

Các nút trong chứa thông tin tổng kết từ hai nút con của nó theo nhiệm vụ quản lý của cây. Ví dụ cây quản lý tổng thì  $t[u] = t[2*u] + t[2*u+1]$ , với cây quản lý min ta có  $t[u] = \min\{t[2*u], t[2*u+1]\}$  v.v...

### Tổ chức dữ liệu

- Các mảng **int**  $t_{\min}[N4] = \{N\}$ ,  $t_{\max}[N4] = \{0\}$  lưu trữ các cây quản lý đoạn, mỗi cây phục vụ một tiêu chí quản lý, khai báo trên tổ chức 2 cây để quản lý *min* và *max*,  $N4 = 4 \times N = 4 \times (10^5 + 1)$
- Mảng **int**  $a[N4]$  lưu dữ liệu dãy ban đầu.

Trong nhiều trường hợp không cần lưu riêng mảng dữ liệu ban đầu mà đưa thẳng vào cây khi nhập.

Mỗi tiêu chí quản lý tương ứng với một cây hoặc có thể gộp chung trong một cây với kiểu dữ liệu phần tử là pair hoặc tuple. Việc gộp chung chỉ đẹp về mặt lý thuyết, không có hiệu ứng tiết kiệm bộ nhớ cũng như số câu lệnh xử lý.

## Các phép xử lý

### Khởi tạo cây

Gồm 3 công việc:

- + Gán giá trị đầu thích hợp cho tất cả các nút của cây, trong một trường hợp công việc này có thể được bỏ qua,
- + Gán giá trị cho các nút lá,
- + Tính giá trị cho các nút trong của cây.

Các nút của cây cần được gán giá trị đầu phù hợp với việc tính giá trị quản lý, ví dụ bằng 0 với việc tính tổng, số cực lớn cho việc quản lý min, số cực bé – quản lý max, v.v... Việc gán giá trị đầu có thể được thực hiện ngay trong khi khai báo biến hoặc dùng hàm assign, ví dụ **t.assign(m, 0)** – gán 0 cho m phần tử đầu.

Gán giá trị cho các nút lá: Điều máu chót là tìm được vị trí nút lá đầu tiên trong cây. Các nút của cây có thể phân thành các mức:

Mức 0 chứa nút gốc và có  $2^0 = 1$  nút,

Mức 1 chứa các nút con của gốc và có  $2^1 = 2$  nút ,

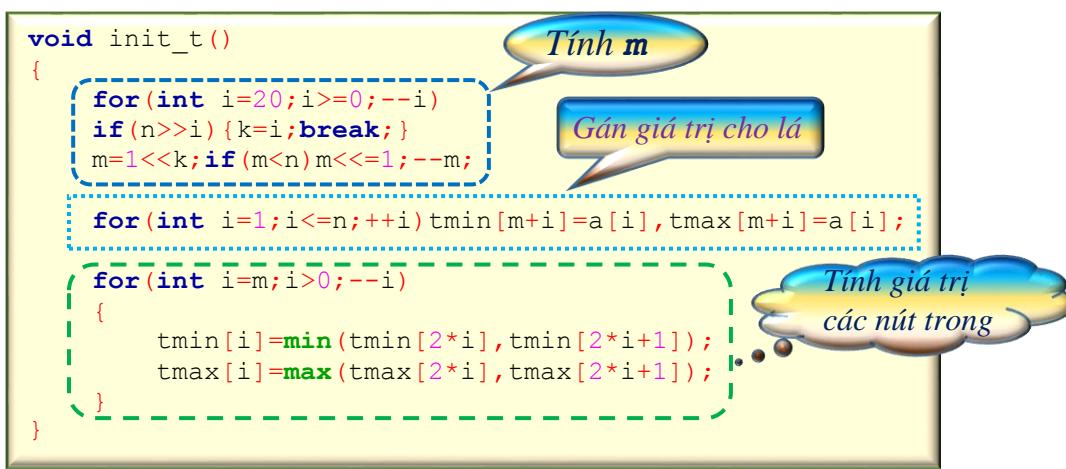
.....

Mức  $i$  chứa các nút con của các nút mức  $i-1$  và có  $2^i$  nút,  $i = 1, 2, 3, \dots$

Gọi  $k$  là mức chứa nút lá của cây quản lý mảng ( $a_1, a_2, \dots, a_n$ ).

$k$  phải thỏa mãn điều kiện  $2^{k-1} < n \leq 2^k$ .

Chỉ số lớn nhất của nút trong sẽ là  $m = 2^k - 1$ . Chỉ số các nút lá bắt đầu từ  $m+1$  và  $t[m+i] = a[i]$ .

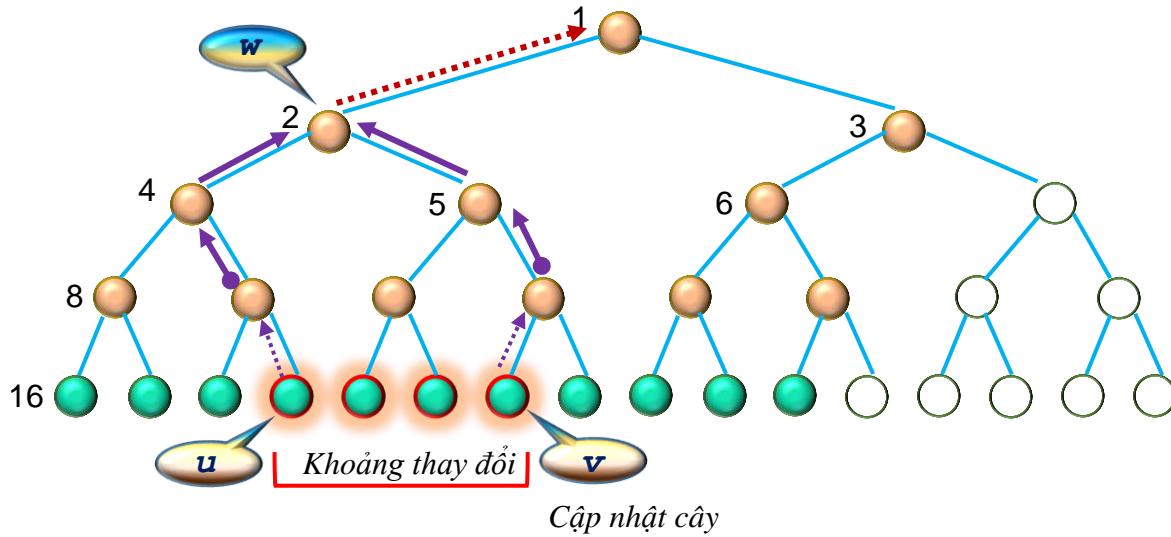


Khởi tạo 2 cây quản lý min và max

Có thể sử dụng sơ đồ đệ quy để xây dựng cây. Sơ đồ đệ quy đẹp về mặt lý thuyết nhưng kém hiệu quả trong hoạt động (tốn bộ nhớ và chậm hơn).

### Cập nhật

Ở đây ta chỉ xét việc cập nhật cây khi *giá trị một số nút lá thay đổi*, kích thước cây vẫn giữ nguyên.



Xét 2 trường hợp:

- ✚ Trường hợp chung: Nhiều nút lá trong đoạn  $[u, v]$  thay đổi giá trị,
- ✚ Trường hợp riêng: Chỉ có nút lá  $u$  và nút lá  $v$  thay đổi giá trị ( $u \leq v$ ).

Trường hợp riêng thường gặp trong nhiều bài toán ứng dụng, vì vậy ta cần xây dựng riêng một sơ đồ xử lý tối ưu.

Nguyên lý chung:

- ✚ Tính lại giá trị các nút của cây con chứa các nút lá  $u$  và  $v$  tới nút cha chung  $w$  gần nhất của  $u$  và  $v$ ,
- ✚ Cập nhật lại giá trị các nút có liên quan trên đường đi từ  $w$  về gốc.

Hàm xử lý trường hợp chung:

Áp dụng khi có nhiều nút lá trong đoạn  $[u, v]$  bị thay đổi.

```

Cập nhật cây con

void update_t (int p, int q)
{
    int u=p+m, v=q+m;
    while (u!=v)
    {
        u>>=1; v>>=1;
        for (int i=u; i<=v; ++i)
        {
            tmin[i]=min(tmin[2*i], tmin[2*i+1]);
            tmax[i]=max(tmax[2*i], tmax[2*i+1]);
        }
    }

    while (u!=1)
    {
        u/=2;
        tmin[u] =min(tmin[2*u], tmin[2*u+1]);
        tmax[u] =max(tmax[2*u], tmax[2*u+1]);
    }
}

```

*Cập nhật đường đi  
về gốc*

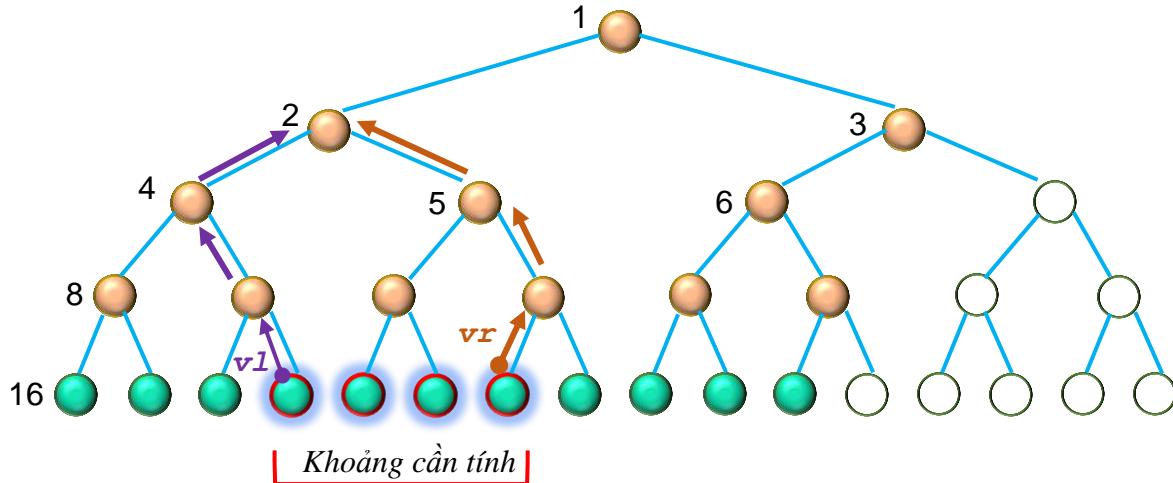
Trường hợp riêng: Khi chỉ có 2 nút lá thay đổi ta chỉ cần cập nhật các nút liên quan tới đường đi từ u và đường đi từ v về nút cha chung w, sau đó cập nhật đường đi từ w tới gốc. Chi phí cập nhật sẽ giảm đáng kể so với trường hợp chung khi u và v cách nhau đủ xa.

```

void change ()
{
    int u=p+m, v=q+m;
    t=tmin[u]; tmin[u]=tmin[v]; tmin[v]=t;
    t=tmax[u]; tmax[u]=tmax[v]; tmax[v]=t;
    while (u!=v)
    {
        u/=2; v/=2;
        tmin[u]=min(tmin[2*u], tmin[2*u+1]);
        tmax[u]=max(tmax[2*u], tmax[2*u+1]);
        tmin[v]=min(tmin[2*v], tmin[2*v+1]);
        tmax[v]=max(tmax[2*v], tmax[2*v+1]);
    }
    while (u!=1)
    {
        u/=2;
        tmin[u]=min(tmin[2*u], tmin[2*u+1]);
        tmax[u]=max(tmax[2*u], tmax[2*u+1]);
    }
}

```

## Truy vấn kết quả trên đoạn $[u, v]$



## Truy vấn kết quả trên đoạn $[u, v]$

Duyệt đồng thời 2 đường đi trái và phải xuất phát tương ứng từ  $u$  và  $v$  cho tới đỉnh con chung gần nhất. Việc tổng hợp kết quả được thực hiện khi *khoảng cách 2 nút* tương ứng trên 2 đường đi *bằng 1*.

```
void check_cv()
{
    int u=p+m, v=q+m, vln, vrn, vlx, vr;
    vln=tmin[u]; vlx=tmax[u];
    vrn=tmin[v]; vr=tmax[v];
    if(u==v) {vmin=tmin[u]; vmax=tmax[u]; return;}
    while(v-u>1)
    {
        if((u&1)==0) {vln=min(vln,tmin[u+1]); vlx=max(vlx,tmax[u+1]);}
        if(v&1) {vrn=min(vrn,tmin[v-1]); vr=max(vr,tmax[v-1]);}
        u>>=1; v>>=1;
    }
    vmin=min(vln,vrn); vmax=max(vlx,vr);
}

if(vmax-vmin==q-p) fo<<"Yes\n"; else fo<<"No\n";
```

Phụ thuộc yêu cầu bài toán

Tại mỗi bước, ở đường đi bên trái chỉ cập nhật kết quả khi xuất phát từ đỉnh có số chẵn, với đường đi bên phải – khi đỉnh xuất phát có số thứ tự lẻ.

## Bài toán ứng dụng minh họa

Xét dãy số nguyên dương  $a_1, a_2, \dots, a_n$ , các số nhận giá trị không vượt quá  $n$  và khác nhau từng đôi một.

Yêu cầu xử lý  $m$  truy vấn, mỗi truy vấn có một trong 2 dạng:

- ✓ **1 p q** – đổi chỗ 2 số trong dãy ở các vị trí  $p$  và  $q$  cho nhau,
- ✓ **2 p q** – kiểm tra các số của dãy ở các vị trí thuộc đoạn  $[p, q]$  có phủ kín các điểm tọa độ nguyên của một đoạn thẳng nào đó hay không, nói một cách khác, có tồn tại số nguyên  $r$  để các số  $r, r+1, \dots, r+q-p$  thuộc đoạn  $[p, q]$  của dãy và đưa ra kết quả “**Yes**” hoặc “**No**” tương ứng.

**Dữ liệu:** Vào từ file văn bản COVERED.INP:

- ⊕ Dòng đầu tiên chứa một số nguyên  $n$  ( $1 \leq n \leq 10^5$ ),
- ⊕ Dòng thứ 2 chứa  $n$  số nguyên  $a_1, a_2, \dots, a_n$ ,
- ⊕ Dòng thứ 3 chứa số nguyên  $m$  ( $1 \leq m \leq 10^5$ ),
- ⊕ Mỗi dòng trong  $m$  dòng sau chứa 3 số nguyên xác định một truy vấn.

**Kết quả:** Đưa ra file văn bản COVERED.OUT các kết quả tương ứng với truy vấn loại 2, mỗi kết quả trên một dòng.

**Ví dụ:**

COVERED.INP
12
3 1 2 6 9 8 7 4 5 12 10 11
10
2 4 7
1 8 9
2 4 8
2 4 9
2 5 11
1 8 9
1 4 8
1 12 10
2 5 11
2 5 12

COVERED.OUT
Yes
Yes
Yes
No
Yes
Yes

## Chương trình

```
#include <bits/stdc++.h>
#define NAME "covered."
using namespace std;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int N = 100001;
const int N4 = 400005;
int n,m,k,p,q,t,a[N],tmin[N4]={N},tmax[N4]={0};
int vmin,vmax;

void init_t()
{
    for(int i=20;i>=0;--i)
    if(n>>i) {k=i; break;}
    m=1<<k; if(m>n) m<<=1; --m;
    for(int i=1;i<=n;++i)tmin[m+i]=a[i],tmax[m+i]=a[i];
    for(int i=m;i>0;--i)
    {
        tmin[i]=min(tmin[2*i],tmin[2*i+1]);
        tmax[i]=max(tmax[2*i],tmax[2*i+1]);
    }
}

void change()
{
    int u=p+m, v=q+m;
    t=tmin[u]; tmin[u]=tmin[v]; tmin[v]=t;
    t=tmax[u]; tmax[u]=tmax[v]; tmax[v]=t;
    while(u!=v)
    {
        u/=2; v/=2;
        tmin[u]=min(tmin[2*u],tmin[2*u+1]);
        tmax[u]=max(tmax[2*u],tmax[2*u+1]);
        tmin[v]=min(tmin[2*v],tmin[2*v+1]);
        tmax[v]=max(tmax[2*v],tmax[2*v+1]);
    }
    while(u!=1)
    {
        u/=2;
        tmin[u]=min(tmin[2*u],tmin[2*u+1]);
        tmax[u]=max(tmax[2*u],tmax[2*u+1]);
    }
}

void check_cv()
{
    int u=p+m, v=q+m, vln, vrn, vlx, vrnx;
    vln=tmin[u]; vlx=tmax[u];
    vrn=tmin[v]; vrnx=tmax[v];
    if(u==v) {vmin=tmin[u]; vmax=tmax[u]; return;}
    while(v-u>1)
    {
        if((u&1)==0) {vln=min(vln,tmin[u+1]); vlx=max(vlx,tmax[u+1]);}
        if(v&1) {vrn=min(vrn,tmin[v-1]); vrnx=max(vrnx,tmax[v-1]);}
        u>>=1; v>>=1;
    }
    vmin=min(vln,vrn); vmax=max(vlx,vrnx);
}
```

```

if (vmax-vmin==q-p) fo<<"Yes\n";else fo<<"No\n";
}

int main()
{
    fi>>n;
    for(int i=1; i<=n; ++i) fi>>a[i];
    init_t();

    fi>>k;
    for(int i=0; i<k; ++i)
    {
        fi>>t>>p>>q;
        if (p>q) swap(p, q);
        if (t==1) change(); else check_cv();
    }

    fo<<"\nTime: "<<clock() / (double)1000<<" sec";
}

```



## TREAP

Treap là cấu trúc dữ liệu kết hợp giữa cây tìm kiếm nhị phân với vun đống nhị phân, vì vậy tên gọi là sự kết hợp tên của hai cấu trúc trên (*Treap = Tree + Heap*).

Cấu trúc dữ liệu này lưu trữ các cặp ( $x, y$ ) dưới dạng tạo thành cây tìm kiếm nhị phân theo  $x$  và là vun đống nhị phân theo  $y$ . Giả thiết tất cả các  $x$  và tất cả các  $y$  khác nhau. Khi đó nếu trong cấu trúc có nút lưu trữ cặp ( $x0, y0$ ) thì ở các nút của cây con trái của nút này đều có  $x$  nhỏ hơn  $x0$ , ở các nút của cây con phải của nút này đều có  $x$  lớn hơn  $x0$ , ngoài ra, ở cả cây con phải lẫn cây con trái giá trị  $y$  ở các nút đều nhỏ hơn  $y0$ .

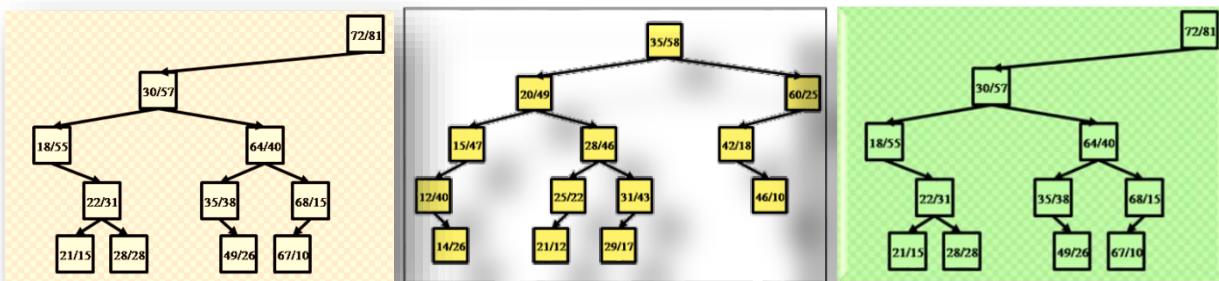
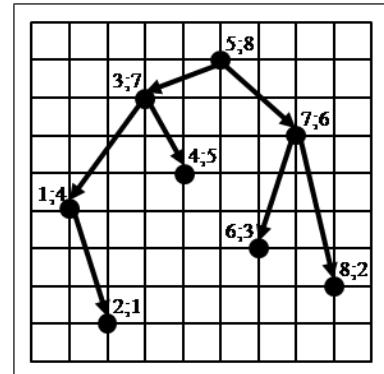
Treap do Siedel và Argon đề xuất năm 1996.

Trong phần lớn các ứng dụng Treap,  $x$  là khóa (*key*), đồng thời cũng là giá trị lưu trữ trong cấu trúc, còn  $y$  là độ ưu tiên (*priority*). Nếu như không có độ ưu tiên  $y$ , với bộ dữ liệu  $x$  cho trước ta sẽ có một tập các cây tìm kiếm nhị phân, trong số đó một số cây suy biến thành nhánh và làm chậm một cách đáng kể quá trình tìm kiếm.

Việc đưa mức độ ưu tiên vào sẽ cho phép xác định cây một cách đơn trị, không phụ thuộc vào trình tự bổ sung các nút. Ta có thể xây dựng cây nhị phân không suy biến, đảm bảo thời gian tìm kiếm trung bình là  $O(\log n)$ , bởi vì khi bổ sung nút với độ ưu tiên ngẫu nhiên xác xuất nhận được cây có độ cao lớn hơn  $[4\log_2 n]$  là vô cùng bé.

Cặp giá trị ( $x, y$ ) có thể xét như tọa độ một điểm trên mặt phẳng. Khi đó Treap là một cây đồ thị phẳng (các cạnh không giao nhau) nối các điểm trên mặt phẳng. Vì vậy Treap còn được gọi là Cây Đề các (*Carsian Tree*).

Ví dụ: Một số cấu trúc Treap:



## Các phép xử lý với Treap

Treap cho phép tổ chức xử lý như với cây tìm kiếm nhị phân bình thường, nhưng trong phần lớn các trường hợp thời gian thực hiện mỗi phép xử lý đều là  $O(\log n)$ :

- ♣ `void insert(T1 key, T2 value)` – bổ sung cặp giá trị (`key, value`) vào Treap,
- ♣ `void remove(T1 key)` – xóa các nút có khóa bằng `key` khỏi Treap,
- ♣ `T2 find(T1 key)` – tìm giá trị xác định bởi khóa `key`.

Cấu trúc Treap cũng cho phép dễ dàng thực hiện các phép chọn và cập nhật theo nhóm:

- ♣ `T2 query(T1 key1, T1 key2)` – xác định giá trị của một hàm (tổng, tích, max, . . .) với các phần tử trong khoảng `[key1, key2]`,
- ♣ `void modify(T1 key1, T1 key2)` – cập nhật giá trị (*tăng thêm một lượng, nhân với một số, xác lập một thuộc tính, . . .*) với tất cả các phần tử trong khoảng `[key1, key2]`.

Tồn tại nhiều phép xử lý phức tạp hơn như hợp nhất các Treap, phân chia một Treap thành vài Treaps riêng biệt, . . . đều được thực hiện dựa trên các phép xử lý cơ sở sẽ xét chi tiết dưới đây.

## Các phép xử lý cơ sở

*Khai báo cấu trúc phần tử của Treap*

```
struct TreapNode
{
    int k, p, v;
    TreapNode *l, *r;
    TreapNode(int key, int value)
    {
        k = key;
        v = value;
        p = rand();
        l = r = NULL;
    }
};
```

### Merge (Hợp nhất Treaps)

Xét Treap **A** có gốc là **a** và Treap **B** có gốc là **b**. Giả thiết các khóa của **A** đều nhỏ hơn khóa bất kỳ của **B** (hoặc không lớn hơn nếu tổ chức Treap với các phần tử có

thể có khóa giống nhau). Cần tạo Treap **T** mới từ các phần tử của **A** và **B**, trả về con trỏ chỉ tới gốc của **T**:

```
TreapNode *merge(TreapNode *a, TreapNode *b)
{
    if (!a || !b)
        return a ? a : b;
    if (a->p > b->p)
    {
        a->r = merge(a->r, b);
        return a;
    } else
    {
        b->l = merge(a, b->l);
        return b;
    }
}
```

Giả thiết khóa của các phần tử 2 cây đều khác nhau. Khi đó gốc của **T** sẽ là phần tử của **A** hoặc **B** có độ ưu tiên lớn nhất. Để dàng thấy rằng phần tử đó là **a** hoặc **b**. Nếu **a->p > b->p** thì **a** là gốc của cây hợp nhất. Các phần tử của **B** đều có khóa lớn hơn **a->k** vì vậy chúng sẽ được gắn vào cây con **a->r**. Như vậy bài toán ban đầu được dẫn về bài toán hợp nhất 2 cây **a->r** với **B**. Không khó để nhận thấy là quy trình xử lý trên có thể thể hiện bằng sơ đồ đệ quy. Đệ quy kết thúc khi một trong 2 cây là rỗng. Trường hợp gốc là **b** được xử lý tương tự.

#### *split (Tách cây)*

Cho số **k** và cây **T** với gốc là **t**. Yêu cầu tạo 2 cây: cây **A** gồm các phần tử của **T** có khóa nhỏ hơn **k** và cây **B** – chứa các phần tử còn lại của **T**. Hàm trả về giá trị 2 con trỏ chỉ tới gốc các cây.

Nếu **t->k < k** thì **t** thuộc **A**, trong trường hợp ngược lại – **t** thuộc **B**. Giả thiết **t** thuộc **A**, khi đó **t** sẽ là gốc của **A**. Khi đó các phần tử của cây con **t->l** sẽ là một phần của **A** vì có khóa nhỏ hơn **t->k** và do đó – nhỏ hơn **k**. Với cây con **t->r** tình hình phức tạp hơn: cây con có thể chứa các phần tử có khóa nhỏ hơn **k** lẫn các phần tử có khóa không nhỏ hơn **k**. Ta có thể xử lý theo sơ đồ đệ quy: tách cây con **t->r** theo **k**, nhận được 2 cây **A'** và **B'**. Treap **A'** được đặt thế chỗ cho **t->r**, khi đó **T** sẽ chỉ chứa các phần tử có khóa nhỏ hơn **k**. **B'** chứa các phần tử có khóa không nhỏ hơn **k**. Như vậy kết quả tách cây sẽ là **T** và **B'**. Trường hợp **t** thuộc **B** – xử lý tương tự. Đệ quy kết thúc khi **T** trở thành rỗng.

```

void split(TreapNode *t, int k, TreapNode *&a, TreapNode *&b)
{
    if (!t)
        a = b = NULL;
    else if (t->k < k)
    {
        split(t->r, k, t->r, b);
        a = t;
    } else
    {
        split(t->l, k, a, t->l);
        b = t;
    }
}

```

### Tìm kiếm trên Treap

Việc tìm kiếm trên Treap được thực hiện theo đúng sơ đồ tìm kiếm trên cây nhị phân. Dễ dàng thấy rằng độ phức tạp của quá trình tìm kiếm không vượt quá  $O(\log n)$ . Như vậy Treap có thể coi như cây tìm kiếm nhị phân cân bằng (*tuy bản thân Treap có cấu trúc phức tạp hơn*).

Luôn luôn giả thiết rằng con trỏ `TreapNode *t` chỉ tới gốc của Treap.

### Bổ sung phần tử mới

Xét việc bổ sung phần tử với khóa **key** và giá trị **value** vào cây **T**. Phần tử này có thể xem như một Treap **Tn** hoàn chỉnh bao gồm một nút. Như vậy có thể tiến hành hợp nhất hai cây. Tuy vậy, đầu tiên phải xử lý sơ bộ, chia **T** thành 2 phần: **T1** chứa các phần tử có khóa nhỏ hơn key và **T2** – chứa các phần tử còn lại, sau đó thực hiện hai phép hợp nhất và hoàn thành việc bổ sung phần tử mới vào cây.

```

void insert(int key, int value)
{
    TreapNode *tn = new TreapNode(key, value), *t1, *t2;
    split(t, key, t1, t2);
    t = merge(t1, tn);
    t = merge(t, t2);
}

```

### Xóa phần tử của cây

Giả thiết phải loại bỏ khỏi **T** tất cả các phần tử có khóa bằng **key**. Tách **T** thành 2 cây **T1** và **T2** theo khóa **key**, **T1** chứa các phần tử có khóa nhỏ hơn **key**, **T2** chứa

các phần tử còn lại. Tách **T2** theo khóa **key+1**, nhánh phải (**T3**) của cây nhận được chứa các phần tử có khóa lớn hơn key, còn trong **T2** chỉ có các phần tử có khóa bằng **key**. Hợp nhất **T1** và **T3** ta có kết quả cần tìm.

Nếu cần giải phóng bộ nhớ của các phần tử bị xóa ta cần gọi hàm **dispose**:

```
void dispose(TreapNode *n)
{
    if (n == NULL)
        return;
    dispose(n->l);
    dispose(n->r);
    delete n;
}
```

```
void remove(int key)
{
    TreapNode *t1, *t2, *t3;
    split(t, key, t1, t2);
    split(t2, key + 1, t2, t3);
    t = merge(t1, t3);
    dispose(t2);
}
```

### *Tìm phần tử*

Việc tìm kiếm một phần tử với khóa cho trước trên Treap có thể thực hiện theo đúng sơ đồ tìm kiếm trên cây nhị phân bình thường.

Mặt khác có thể sử dụng các công cụ **merge()** và **split()** đã xây dựng ở trên để tìm kiếm. Phần của cây có thể chứa phần tử cần tìm với khóa cho trước có thể xác định bằng hai lát cắt. Sau khi trích hoặc xử lý kết quả cần khôi phục lại cây (bằng hai phép ghép). Cách xử lý này đặc biệt có hiệu quả với các truy vấn xử lý nhóm phần tử. Nếu phần tử cần tìm không tồn tại kết quả sẽ là 0.

```
int find(int key)
{
    int res = 0;
    TreapNode *t1, *t2, *t3;
    split(t, key, t1, t2);
    split(t2, key + 1, t2, t3);
    if (t2 != NULL)
        res = t2->v;
    t1 = merge(t1, t2);
    t = merge(t1, t3);
    return res;
}
```

## Truy vấn theo nhóm phần tử

Các truy vấn đòi hỏi phải làm việc với nhiều phần tử như tính tổng hoặc tích, đếm số lượng, tìm giá trị max hoặc min, . . . có thể dễ dàng thực hiện trên Treap.

### Xử lý truy vấn trên toàn bộ cây

Giả thiết cần tính tổng các phần tử, đồng thời xác định số lượng phần tử.

Trong cấu trúc phần tử cần thay đổi: bổ sung thêm 2 trường **cnt** và **sum** để lưu số lượng và tổng của các phần tử thuộc cây con có gốc là nút đang xét. Trong khai báo cấu trúc các trường này cần được khởi tạo là 1 và giá trị của phần tử.

```
struct TreapNode
{
    int k, p, v;
    int sum, cnt;
    TreapNode *l, *r;
    TreapNode(int key, int value)
    {
        k = key;
        v = value;
        p = rand();
        l = r = NULL;
        sum = value;
        cnt = 1;
    }
};
```

Ngoài ra, cần có thêm các hàm **get()**, **sum()** và **update()** phục vụ cập nhật các trường này trong quá trình xây dựng cây.

```
int getSum(TreapNode *n)
{
    return n != NULL ? n->sum : 0;
}
```

```
int getCnt(TreapNode *n)
{
    return n != NULL ? n->cnt : 0;
}
```

```

void update(TreapNode *n)
{
    if (n == NULL) return;
    n->sum = v + getSum(n->l) + getSum(n->r);
    n->cnt = 1 + getCnt(n->l) + getCnt(n->r);
}

```

Các phép **merge ()** và **split ()** cũng cần được thay đổi thích hợp để phục vụ cho các trường mới đưa vào. Với mỗi nhánh cây nhận được cần gọi hàm **update ()** để chỉnh lý giá trị các trường **cnt** và **sum**.

```

TreapNode *merge(TreapNode *a, TreapNode *b)
{
    if (!a || !b)
        return a ? a : b;
    if (a->p > b->p) {
        a->r = merge(a->r, b);
        update(a);
        return a;
    } else {
        b->l = merge(a, b->l);
        update(b);
        return b;
    }
}

```

```

void split(TreapNode *t, int k,
           TreapNode *&a, TreapNode *&b)
{
    if (!t)
        a = b = NULL;
    else if (t->k < k) {
        split(t->r, k, t->r, b);
        a = t;
    } else {
        split(t->l, k, a, t->l);
        b = t;
    }
    update(a); update(b);
}

```

Nếu cần tìm số lượng hay tổng các phần tử chỉ cần truy nhập tới trường tương ứng của phần tử gốc.

### Xử lý truy vấn trên đoạn

Giả thiết cần đếm số phần tử hoặc tổng các phần tử có khóa nằm trên đoạn  $[k_1, k_2]$ . Bằng hai lát cắt (tương tự như khi tìm hoặc xóa phần tử) ta có thể xác định nhánh cây chứa các phần tử cần tìm. Việc xác định giá trị quan tâm được thực hiện như đã nêu ở mục trên.

```
int rangeSum(int k1, int k2)
{
    TreapNode *t1, *t2, *t3;
    split(t, k1, t1, t2);
    split(t2, k2 + 1, t2, t3);
    int s = getSum(t2);
    t1 = merge(t1, t2);
    t = merge(t1, t3);
    return s;
}
```

### Cập nhật theo nhóm phần tử

Các phép cập nhật theo nhóm phần tử có thể là thay các giá trị cũ bằng một giá trị mới, nhân một số với các giá trị ở nút, bổ sung thêm một thuộc tính nào đó, ...

Dưới đây ta sẽ xét việc cộng thêm một số vào các giá trị của nút trên toàn cây hoặc trên một đoạn. Các loại cập nhật khác được thực hiện theo kiểu tương tự.

### Cập nhật trên toàn cây

Xét việc cộng thêm một số vào các giá trị ở tất cả các nút của cây.

Cần bổ sung vào cấu trúc **TreapNode** trường **add** lưu giá trị cần tăng. Trong khai báo cần gán giá trị đầu là 0 cho trường mới này.

```
struct TreapNode
{
    int k, p, v;
    int sum, cnt, add;
    TreapNode *l, *r;
    TreapNode(int key, int value)
    {
        k = key;
        v = value;
        p = rand();
        l = r = NULL;
        sum = value;
        cnt = 1;
        add = 0;
    }
};
```

Khi đó giá trị thực lưu ở nút gốc **t** của cây **T**:

Không đơn thuần là  $t \rightarrow v$  mà sẽ là  $t \rightarrow v + t \rightarrow add$ .

Giá trị trường **sum**: thay  $t \rightarrow sum$  bằng  $t \rightarrow sum + t \rightarrow add * t \rightarrow cnt$ .

```
int getSum(TreapNode *n)
{
    return n != NULL ? n->sum + n->add * n->cnt : 0;
}
```

Để các hàm **merge()** và **split()** hoạt động đúng cần đảm bảo việc chuyển giao giá trị thực cho các tham số. Sự tồn tại của trường add có thể phá vỡ tính đúng đắn. Ví dụ  $t \rightarrow add$  bằng 10, còn  $t \rightarrow l \rightarrow add$  bằng 0, thì sau khi ngắt  $t \rightarrow l$  khỏi  $t$  thông tin xác định là giá trị các phần tử của cây phải được cộng thêm 10 sẽ bị mất. Vì vậy trước khi thực hiện các phép **merge()** và **split()** cần chuyển giao thông tin này cho các nhánh sẽ nhận được bằng hàm **down()**.

```
void down(TreapNode *n)
{
    if (n == NULL)
        return;
    n->v += n->add;
    if (n->l != NULL)
        n->l->add += n->add;
    if (n->r != NULL)
        n->r->add += n->add;
    n->add = 0;
}
```

Các hàm **merge()** và **split()** sẽ có dạng:

```
TreapNode *merge(TreapNode *a, TreapNode *b)
{
    if (!a || !b)
        return a ? a : b;
    if (a->p > b->p)
    {
        down(a);
        a->r = merge(a->r, b);
        update(a);
        return a;
    } else
    {
        down(b);
        b->l = merge(a, b->l);
        update(b);
        return b;
    }
}
```

```

void split(TreapNode *t, int k,
           TreapNode *&a, TreapNode *&b)
{
    down(t);
    if (!t)
        a = b = NULL;
    else if (t->k < k)
    {
        split(t->r, k, t->r, b);
        a = t;
    } else
    {
        split(t->l, k, a, t->l);
        b = t;
    }
    update(a);
    update(b);
}

```

Với Treap được tổ chức như trên, nếu muốn cộng thêm một giá trị nào đó vào tất cả các phần tử của cây thì cần gán giá trị này cho trường **add** của phần tử gốc.

### Cập nhật trên đoạn

Nếu như đã thiết kế Treap với khả năng cập nhật toàn bộ cây thì việc cập nhật trên đoạn **[k1, k2]** cũng sẽ dễ dàng thực hiện, tương tự như việc xử lý truy vấn trên đoạn.

```

void rangeAdd(int k1, int k2, int addVal)
{
    TreapNode *t1, *t2, *t3;
    split(t, k1, t1, t2);
    split(t2, k2 + 1, t2, t3);
    if (t2 != NULL)
        t2->add += addVal;
    t1 = merge(t1, t2);
    t = merge(t1, t3);
}

```

## Phạm vi ứng dụng và hạn chế của cấu trúc Treap

### Ứng dụng

- ✚ Cấu trúc Treap tương đối dễ lập trình, vì vậy nó là công cụ hữu hiệu giải nhiều loại bài toán đòi hỏi xử lý truy vấn ở các dạng:
- ✚ Các bài toán đòi hỏi tổ chức cây đủ cân bằng để tìm kiếm và cập nhật (ví dụ, tổ chức tập hợp, tổ chức từ điển),

- ✚ Các loại bài toán xử lý truy vấn trên một phạm vi nào đó, các bài toán cập nhật trên đoạn. Khác với cây quản lý đoạn, Treap cho phép xử lý truy vấn trên các tập thay đổi,
- ✚ Tính toán các số liệu thống kê với độ phức tạp  $O(\log n)$ ,
- ✚ Phục vụ xây dựng một cấu trúc dữ liệu mạnh hơn – Treap với khóa ẩn.

### ***Hạn chế***

- ✓ Tốn bộ nhớ: mỗi nút của cây phải lưu trữ khá nhiều trường dữ liệu trung gian cho các loại truy vấn khác nhau,
- ✓ Việc thay đổi cấu trúc nút sẽ kéo theo sự thay đổi trong các phép xử lý cơ bản, điều này làm tăng độ phức tạp lập trình.

## Treap với khóa ẩn

### Ý tưởng chính

Cấu trúc dữ liệu vector trong C/C++ cho phép tổ chức mảng động với khả năng bổ sung phần tử mới vào cuối mảng hoặc xóa phần tử cuối trong danh sách. Cấu trúc này cũng cho phép xác định hoặc cập nhật giá trị của phần tử theo vị trí của nó.

Giả thiết ta cần một cấu trúc dữ liệu như vậy cùng với các khả năng mới để có thể bổ sung phần tử vào vị trí bất kỳ hoặc xóa một phần tử cùng với việc đánh số tự động lại các phần tử của mảng.

Cấu trúc dữ liệu với các tính chất đã nêu có thể xây dựng dựa trên cấu trúc Treap và được gọi là *Treap với khóa ẩn* (*Treap with implicit key*).

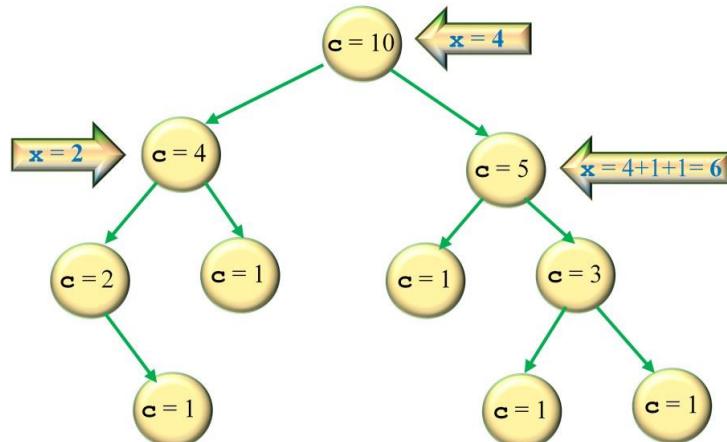
### Khóa X

Như đã nói ở trên, Treap là cấu trúc dữ liệu kết hợp giữa cây nhị phân và vun đong nhị phân. Trong Treap khóa **x** phục vụ tổ chức cây nhị phân được lưu trữ tường minh. Nhưng người ta cũng có thể tránh việc lưu trữ khóa tường minh bằng cách lấy *số lượng phần tử ở nhánh trái* của phần tử đang xét làm *khóa* tổ chức cây nhị phân. Như vậy, trong mỗi phần tử chỉ lưu trữ tường minh độ ưu tiên **y**, còn khóa sẽ là *số thứ tự* của phần tử trong cây *trù 1*.

Vấn đề phải giải quyết ở đây là khi bổ sung hay loại bỏ phần tử, số thứ tự các phần tử trong cây có thể bị thay đổi. Nếu không có cách tiếp cận hợp lý, độ phức tạp của các phép bổ sung, loại bỏ sẽ có độ phức tạp  $O(n)$  và làm mất ưu việt của cây tìm kiếm nhị phân.

### Đại lượng thay thế C

Lối thoát khỏi tình huống rắc rối trên là khá đơn giản. Thay vì lưu trữ tường minh **x** người ta lưu trữ **c** – *số lượng nút của cây con* có gốc là phần tử đang xét (bao gồm cả nút gốc). Đây là một đại lượng có miền xác định đủ nhỏ và dễ dàng dẫn xuất khi cần thiết. Lưu ý là các phép xử lý trong Treap đều thực hiện theo chiều từ trên xuống dưới. Nếu trong quá trình duyệt, đi từ gốc tới một đỉnh nào đó ta cộng số lượng nút và tăng thêm 1 của các nhánh trái bị bỏ qua thì khi tới nút cần xét sẽ có trong tay khóa của phần tử đó.



## Các phép xử lý cấu trúc

Hai phép xử lý cấu trúc cây:

- ✚ **merge()** – hợp nhất 2 cây, trong đó ở một cây các khóa **x** có giá trị nhỏ hơn giá trị khóa ở cây kia,
- ✚ **split()** – tách một cây thành 2 cây, trong đó ở một cây các khóa **x** có giá trị nhỏ hơn giá trị khóa ở cây kia.

Trong Treap với khóa ẩn tham số cho hàm merge là gốc của 2 cây bất kỳ cần hợp nhất: **merge(root1, root)**. Lời gọi hàm **split** sẽ là **split(root, t)** xác định việc tách cây được thực hiện sao cho nhánh trái có **t** nút.

### Hàm split

Xuất phát từ nút gốc, xét yêu cầu cắt **k** nút của một cây. Giả thiết nhánh trái của cây có **1** nút và nhánh phải – **r** nút. Có 2 trường hợp xảy ra:

- ♣ **1 ≥ k**: khi đó cần gọi đệ quy split từ nút con trái của gốc với cùng tham số **k**, đồng thời biến phần phải của kết quả thành con trái của gốc, còn gốc của cây đang xử lý sẽ là phần phải của kết quả,
- ♣ **1 < k**: xử lý tương tự như trên nếu đổi vai trò của trái và phải, split được gọi đệ quy từ nút con phải với tham số **k-1-1**, phần trái của kết quả là nút con trái phải, còn gốc – thuộc phần trái kết quả.

Để dễ hiểu tư tưởng thuật toán, các bước xử lý được trình bày ở đoạn mã giả sau (giải thuật xử lý chính xác trên C++ với đầy đủ các hàm sẽ được xét ở phần cuối).

```
<Treap, Treap> split(Treap t, int k)
{
    int l = t.left.size
    if l >= k
        <t1, t2> = split(t.left, k)
        t.left = t2
        update(v)
        r = v
    return <t1, t2>
else
    <t1, t2> = split(t.right, k - l - 1)
    t.right = t1
    update(v)
    l = v
return <t1, t2>
```

### Hàm merge

Trong quá trình hợp nhất không có sử dụng giá trị khóa x vì vậy hàm **merge** được xử lý như ở Treap bình thường.

## Đảm bảo tính nhất quán của c

Sau mỗi thao tác xử lý đỉnh con cần ghi vào trường lưu trữ c tổng giá trị tương ứng của các nút con cộng thêm 1.

```
void update(Treap t)
t.size = 1 + t.left.size + t.right.size
```

## Ứng dụng Treap với khóa ẩn

- ✚ Bổ sung phần tử mới vào bất kỳ vị trí nào trong cây đang xét đồng thời duy trì mọi tính chất của cây ban đầu,
- ✚ Di chuyển đoạn các phần tử của mảng sang nơi mới bất kỳ,
- ✚ Thực hiện các phép xử lý đổi với nhóm phần tử, kích thước nhóm thay đổi và được xác định ở mỗi lần xử lý, điều mà cấu trúc quản lý đoạn không đáp ứng được,
- ✚ Có thể thực hiện việc đổi chỗ các phần tử ở vị trí chẵn sang vị trí lẻ và ngược lại,
- ✚ Thực hiện các phép xử lý nêu trên một cách hiệu quả đối với xâu (*cấu trúc Rope*).

## Tổ chức Treap với khóa ẩn trên C++

```
struct TN {
    int v, p, c, maxV;
    TN *l, *r;
    TN(int V) : v(V), p(rand()), c(1), maxV(V), l(0), r(0) {}
};

class ITreap {
    TN *root;
    int getMaxV(TN *n) {
        return n ? n->maxV : -(1 << 30);
    }
    int getC(TN *n) {
        return n ? n->c : 0;
    }
    void update(TN *n) {
        if (!n)
            return;
        n->c = 1 + getC(n->l) + getC(n->r);
        n->maxV = max(n->v, max(getMaxV(n->l), getMaxV(n->r)));
    }
    TN *merge(TN *a, TN *b) {
        if (!a || !b)
            return a ? a : b;
        if (a->p > b->p) {
            a->r = merge(a->r, b);
            update(a);
            return a;
        }
        else {
            b->l = merge(a, b->l);
            update(b);
            return b;
        }
    }
};
```

```

    } else {
        b->l = merge(a, b->l);
        update(b);
        return b;
    }
}
void split(TN *t, int k, TN *&a, TN *&b) {
    if (!t) {
        a = b = 0;
        return;
    }
    if (getC(t->l) < k) {
        split(t->r, k - getC(t->l) - 1, t->r, b);
        a = t;
    } else {
        split(t->l, k, a, t->l);
        b = t;
    }
    update(a);
    update(b);
}
public:
    ITreap() : root(0) {}
    int size() {
        return getC(root);
    }
    void insert(int pos, int v) {
        TN *a, *b;
        split(root, pos, a, b);
        root = merge(a, merge(new TN(v), b));
    }
    int getMax(int l, int r) {
        TN *a, *b, *c;
        split(root, l, a, b);
        split(b, r - l + 1, b, c);
        int res = getMaxV(b);
        root = merge(a, merge(b, c));
        return res;
    }
};

```

## Bài tập ứng dụng VU04. HAI BẢN ĐỒ

Tên chương trình: TWOMAPS.CPP

Vết đứt gãy lớn trên vỏ trái đất *Banda Detachment* nằm ở phía tây Thái bình dương, độ sâu đáy biển ở đó đạt tới 7 km. Nhiều bức ảnh địa hình đã được chụp.

Để khảo sát vết đứt gãy lớn này người ta dùng một máy thăm dò tự động. Một trong số các khe nứt tạo thành một đường thẳng và là vùng thuộc kế hoạch thăm dò. Theo dự kiến máy thăm dò sẽ tiến hành đo đặc khảo cứu đoạn có độ dài  $s$ . Bộ nhớ của thiết bị thăm dò chỉ có thể chứa 2 bản đồ cho tổng độ dài là  $s$ . Nếu một phần nào đó có cả 2 bản đồ thì chỉ tính là một. Bộ phận chuẩn bị có 2 thao tác cơ bản:

- Thao tác **A** có dạng **1 1  $x$**  – xin cung cấp bản đồ của khe nứt đoạn từ điểm 1 đến điểm  $x$ ,  $1 < x$ ,
- Thao tác **B** có dạng **2  $k$**  – trả về bản đồ đã xin ở thao tác thứ k. Đảm bảo là thao tác thứ k thuộc loại A và không có việc một bản đồ bị trả về 2 lần.

Các thao tác được đánh số từ 1, có thể tồn tại nhiều bản đồ cùng chụp một đoạn (có 1 và  $x$  giống nhau). Có  $n$  thao tác được thực hiện. Sau mỗi thao tác hãy xác định số cách khác nhau chọn 2 bản đồ khác nhau cho tổng độ dài đúng bằng  $s$ . Hai cách chọn gọi là khác nhau nếu một bản đồ (xác định theo trình tự xin cung cấp) có ở cách chọn thứ nhất và không có ở cách chọn thứ 2. Ban đầu bộ phận chuẩn bị chưa có bản đồ nào trong tay.

**Dữ liệu:** Vào từ file văn bản TWOMAPS.INP:

- ✚ Dòng đầu tiên chứa 2 số nguyên  $s$  và  $n$  ( $1 \leq s \leq 10^9$ ,  $1 \leq n \leq 10^5$ ),
- ✚ Mỗi dòng trong  $n$  dòng tiếp theo chứa 2 hoặc 3 số nguyên xác định thao tác dạng **A** hoặc **B**, trong đó **1,  $x$**  có giá trị tuyệt đối không vượt quá  $5 \times 10^8$ .

**Kết quả:** Đưa ra file văn bản TWOMAPS.OUT số cách chọn tính được sau mỗi thao tác, các kết quả đưa ra dưới dạng số nguyên, mỗi số trên một dòng.

**Ví dụ:**

TWOMAPS.INP	TWOMAPS.OUT
10 6	0
1 0 8	1
1 7 10	2
1 5 15	0
2 2	2
1 12 14	0
2 5	



VU04 Io20161105 J

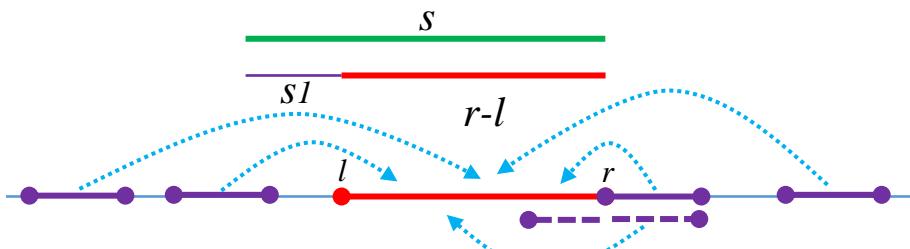
## *Giải thuật: Phương án A: Không sử dụng cấu trúc dữ liệu Treap.*

Nhận xét:

Xét việc bổ sung bản đồ  $[l, r]$ ,

Đặt  $s1 = s - (r - l)$ , có 3 trường hợp xảy ra:

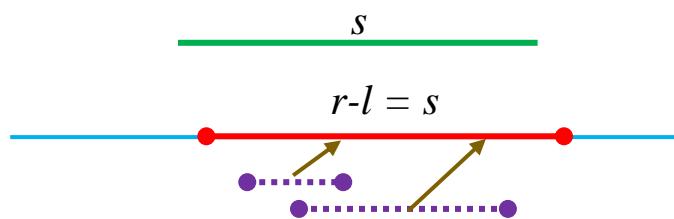
- ✚  $s1 < 0$  : không có thêm lựa chọn mới,
- ✚  $s1 = 0$  : Số lựa chọn bổ sung bằng số đoạn nằm gọn trong  $[l, r]$ ,
- ✚  $s1 > 0$  : Số lựa chọn bổ sung bằng tổng số đoạn có độ dài  $s1$  nằm ngoài đoạn  $[l, r]$ , cộng với số lượng đoạn có giao khác rỗng với  $[l, r]$  và tổng độ dài ngoài phần chung bằng  $s1$ .



Để quản lý các đoạn phục vụ tìm kiếm cần lưu trữ:

- ─ Điểm đầu của đoạn và số lượng điểm đầu trùng với nó *theo khóa điểm cuối*,
- ─ Điểm cuối của đoạn và số lượng điểm cuối trùng với nó *theo khóa điểm đầu*,
- ─ Điểm đầu của đoạn và số lượng điểm đầu trùng với nó *theo khóa độ dài đoạn*,

Khi  $r - l$  bằng  $s$  các thông tin lưu trữ nêu trên mới cho phép thống kê các đoạn cần tìm có điểm đầu hoặc điểm cuối trùng với đoạn đang xét.



Cần lưu trữ *tổng tiền tố* số lượng điểm đầu của các đoạn có độ dài không vượt quá  $s - 2$ , tức là các đoạn có thể nằm gọn trong đoạn độ dài  $s$  và không có chung điểm đầu và điểm cuối với đoạn độ dài  $s$ .

Tổ chức dữ liệu:

Để thuận tiện lập trình và dễ hiểu hơn khi chuyển sang phương án dùng Treap ta khai báo một *cấu trúc dữ liệu hướng đối tượng* (*OOP*), gắn các phép xử lý với dữ liệu. Điều này cho phép dù dữ liệu được lồng vào cấu trúc lưu trữ chuẩn nào, dạng lời gọi tới phép xử lý vẫn giữ nguyên.

```

class MSet
{
    vector < pair < int , int > > a;
public:
    void add(int p, int v)
    {
        for (auto & q : a)
            if (q.first == p)
            {
                q.second += v;
                return;
            }
        a.emplace_back(p, v);
    }
    int get(int p)
    {
        int s = 0;
        for (auto q : a)
            if (q.first < p)
                s += q.second;
        return s;
    }
    int get(int l, int r)
    {
        int s = 0;
        for (auto q : a)
            if (l <= q.first && q.first <= r)
                s += q.second;
        return s;
    }
};

```

Tạo cặp dữ liệu mới  
và bổ sung vào cuối

Hai kiểu truy xuất  
kết quả

Khai báo dữ liệu:

Lưu điểm cuối và số lượng  
(khóa: điểm đầu)

```
map < int , MSet > fs, fl, fr;
```

Lưu điểm đầu và số lượng  
(khóa: độ dài đoạn)

Lưu điểm đầu và số lượng  
(khóa: điểm cuối)

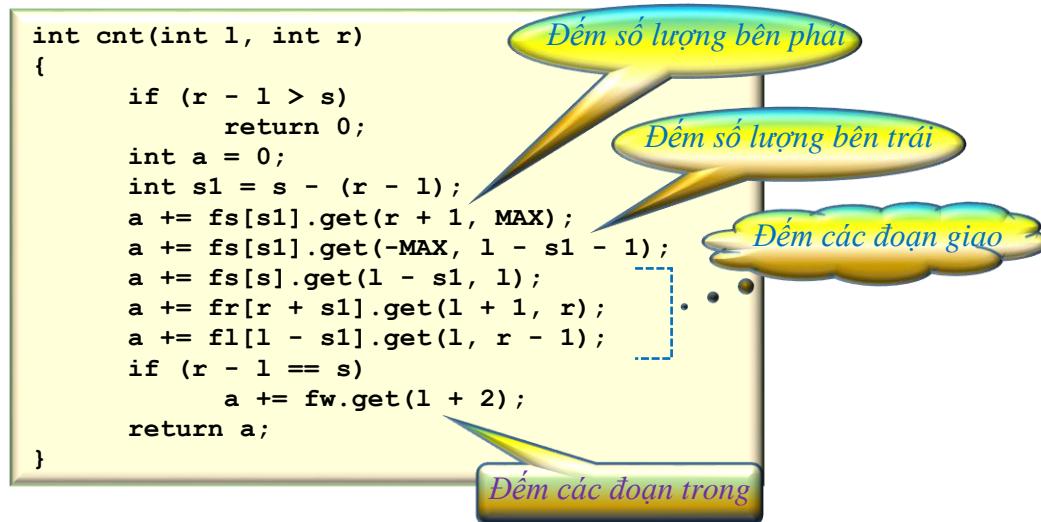
Tích lũy tổng tiền tố  
(khóa: điểm đầu, cuối)

```
MSet fw;
```

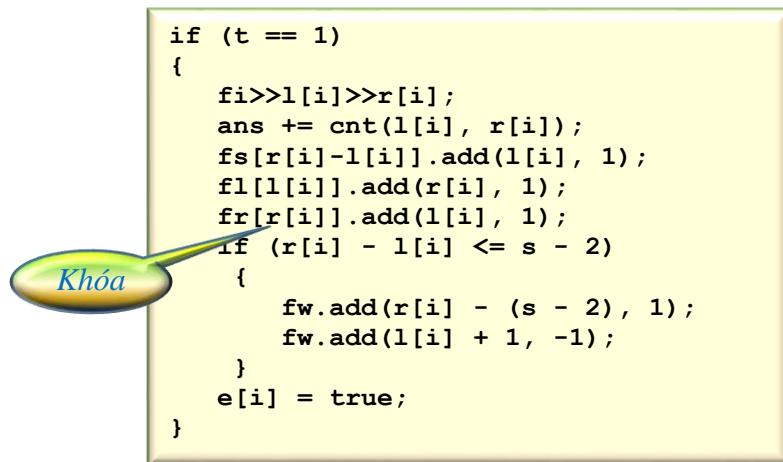
Xử lý:

Thêm bản đồ mới  $[l_i, r_i]$ :

- Cập nhật kết quả: **ans+=cnt(l[i], r[i]);**



- Cập nhật các mảng ghi nhận dữ liệu:



Vì việc loại bỏ bản đồ: được thực hiện tương tự, nhưng cập nhật thông tin trước, sau đó tính lại kết quả.

*Dộ phức tạp của giải thuật:  $O(n^2)$ .*

## Chương trình: Giải thuật A (Không sử dụng Treap)

```
#include <bits/stdc++.h>
#define NAME "twomaps."
using namespace std;
typedef long long ll;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int MAX = 5e8;
const int N = 100002;

class MSet
{
    vector< pair< int , int > > a;
public:
    void add(int p, int v)
    {
        for (auto & q : a)
            if (q.first == p)
            {
                q.second += v;
                return;
            }
        a.emplace_back(p, v);
    }
    int get(int p)
    {
        int s = 0;
        for (auto q : a)
            if (q.first < p)
                s += q.second;
        return s;
    }
    int get(int l, int r)
    {
        int s = 0;
        for (auto q : a)
            if (l <= q.first && q.first <= r)
                s += q.second;
        return s;
    }
};

map< int , MSet > fs, fl, fr;
MSet fw;

int n, s, l[N], r[N];
ll ans = 0;
bool e[N];

int cnt(int l, int r)
{
    if (r - l > s)
        return 0;
    int a = 0;
    int sl = s - (r - l);
    a += fs[sl].get(r + 1, MAX);
    a += fs[sl].get(-MAX, l - sl - 1);
```

```

    a += fs[s].get(l - s1, l);
    a += fr[r + s1].get(l + 1, r);
    a += fl[l - s1].get(l, r - 1);
    if (r - l == s)
        a += fw.get(l + 2);
    return a;
}

int main()
{
    fi>>s>>n;
    for (int i = 0; i < n; ++i)
    {
        int t;
        fi>>t;
        if (t == 1)
        {
            fi>>l[i]>>r[i];
            ans += cnt(l[i], r[i]);
            fs[r[i] - l[i]].add(l[i], 1);
            fl[l[i]].add(r[i], 1);
            fr[r[i]].add(l[i], 1);
            if (r[i] - l[i] <= s - 2)
            {
                fw.add(r[i] - (s - 2), 1);
                fw.add(l[i] + 1, -1);
            }
            e[i] = true;
        } else {
            int k;
            fi>>k;
            --k;
            e[k] = false;
            fs[r[k] - l[k]].add(l[k], -1);
            fl[l[k]].add(r[k], -1);
            fr[r[k]].add(l[k], -1);
            if (r[k] - l[k] <= s - 2)
            {
                fw.add(r[k] - (s - 2), -1);
                fw.add(l[k] + 1, 1);
            }
            ans -= cnt(l[k], r[k]);
        }
        fo<<ans<<'\\n';
    }

    fo<<"\\nTime: "<<clock() / (double)1000<<" sec";
    return 0;
}

```

## *Giải thuật: Phương án B: Sử dụng cấu trúc dữ liệu Treap.*

Nhận xét:

Độ phức tạp của phương án A cao vì không thể chèn dữ liệu mới vào chỗ cần thiết trong các vectors để đảm bảo các dữ liệu luôn luôn được sắp xếp, từ đó có thể áp dụng các phương pháp tìm kiếm nhanh,

Giải pháp:

- ✚ Tổ chức Treap với trường **sum** phục vụ tính tổng tiền tố,
- ✚ Gắn các phép xử lý Treap với cấu trúc.
- ✚ Cấu trúc Treap và các phép xử lý: theo đúng sơ đồ lý thuyết đã nêu.

Lớp dữ liệu Mset được xác định như sau:

```
class MSet
{
    Treap * t;
public:
    MSet() : t(Treap::null) {}
    void add(int p, int v)
    {
        if (!tadd(t, p, v))
        {
            Treap * t1, * t2;
            split(t, p, t1, t2);
            t = merge(merge(t1, new Treap(p, v)), t2);
        }
    }
    int get(int p)
    {
        Treap * t1, * t2;
        split(t, p, t1, t2);
        int w = t1->sum;
        t = merge(t1, t2);
        return w;
    }
    int get(int l, int r)
    {
        Treap * t1, * t2, * t3;
        split(t, l, t1, t2);
        split(t2, r + 1, t2, t3);
        int w = t2->sum;
        t = merge(t1, merge(t2, t3));
        return w;
    }
};
```

Khai báo dữ liệu trong chương trình và sơ đồ xử lý: giữ nguyên như ở phương án A.

*Độ phức tạp của giải thuật: O(nlnn).*

## Chương trình: Giải thuật B – Sử dụng cấu trúc dữ liệu Treap.

```
#include <bits/stdc++.h>
#define NAME "twomaps."
using namespace std;
typedef long long ll;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");
const int MAX = 5e8;
const int N = 100179;

struct Treap
{
    static Treap * null;
    int x, y, val, sum;
    Treap * l, * r;
    Treap(const char *) : x(0), y(0), val(0), sum(0), l(this), r(this) {}
    Treap(int nx,int nval=0):x(nx),y(rand()),val(nval),sum(nval),l(null),r(null) {}

    void update()
    {sum = l->sum + val + r->sum; }
};

Treap * Treap::null = new Treap("...");

Treap * merge(Treap * l, Treap * r)
{
    if (l == Treap::null) return r;
    if (r == Treap::null) return l;
    if (l->y > r->y)
    {
        l->r = merge(l->r, r);
        l->update();
        return l;
    } else {
        r->l = merge(l, r->l);
        r->update();
        return r;
    }
}

void split(Treap * t, int x, Treap *& l, Treap *& r)
{
    if (t == Treap::null)
    { l = r = t;
        return;
    }
    if (t->x < x)
    { l = t;
        split(t->r, x, t->r, r);
    } else {r = t;
        split(t->l, x, l, t->l);
    }
    t->update();
}

bool tadd(Treap * t, int x, int v)
{
    if (t == Treap::null)
```

```

        return false;
    if (t->x == x)
    {
        t->val += v;
        t->sum += v;
        return true;
    } else if (x < t->x)
    {
        if (tadd(t->l, x, v))
        {
            t->sum += v;
            return true;
        }
        return false;
    } else {
        if (tadd(t->r, x, v))
        {
            t->sum += v;
            return true;
        }
        return false;
    }
}
class MSet
{
    Treap * t;
public:
    MSet() : t(Treap::null) {}
    void add(int p, int v)
    {
        if (!tadd(t, p, v))
        {
            Treap * t1, * t2;
            split(t, p, t1, t2);
            t = merge(merge(t1, new Treap(p, v)), t2);
        }
    }
    int get(int p)
    {
        Treap * t1, * t2;
        split(t, p, t1, t2);
        int w = t1->sum;
        t = merge(t1, t2);
        return w;
    }
    int get(int l, int r)
    {
        Treap * t1, * t2, * t3;
        split(t, l, t1, t2);
        split(t2, r + 1, t2, t3);
        int w = t2->sum;
        t = merge(t1, merge(t2, t3));
        return w;
    }
};

map < int , MSet > fs, fl, fr;
MSet fw;
int n, s, l[N], r[N];
ll ans = 0;

```

```

bool e[N];
int cnt(int l, int r)
{
    if (r - l > s)
        return 0;
    int a = 0;
    int s1 = s - (r - l);
    a += fs[s1].get(r + 1, MAX);
    a += fs[s1].get(-MAX, l - s1 - 1);
    a += fs[s].get(l - s1, l);
    a += fr[r + s1].get(l + 1, r);
    a += fl[l - s1].get(l, r - 1);
    if (r - l == s)
        a += fw.get(l + 2);
    return a;
}
int main()
{
    fi>>s>>n;
    for (int i = 0; i < n; ++i)
    {
        int t;
        fi>>t;
        if (t == 1)
        {
            fi>>l[i]>>r[i];
            ans += cnt(l[i], r[i]);
            fs[r[i] - l[i]].add(l[i], 1);
            fl[l[i]].add(r[i], 1);
            fr[r[i]].add(l[i], 1);
            if (r[i] - l[i] <= s - 2)
            {
                fw.add(r[i] - (s - 2), 1);
                fw.add(l[i] + 1, -1);
            }
            e[i] = true;
        } else {
            int k;
            fi>>k; --k;
            e[k] = false;
            fs[r[k] - l[k]].add(l[k], -1);
            fl[l[k]].add(r[k], -1);
            fr[r[k]].add(l[k], -1);
            if (r[k] - l[k] <= s - 2)
            {
                fw.add(r[k] - (s - 2), -1);
                fw.add(l[k] + 1, 1);
            }
            ans -= cnt(l[k], r[k]);
        }
        fo<<ans<<'\\n';
    }
    fo<<"\\nTime: "<<clock() / (double)1000<<" sec";
    return 0;
}

```



## Hệ thống các tập không giao nhau

Hệ thống các tập không giao nhau (*disjoint-set-union – DSU*) là một cách tổ chức dữ liệu đơn giản nhưng rất hiệu quả để giải quyết nhiều loại bài toán khác nhau.

Ban đầu mỗi phần tử của dữ liệu thuộc một tập riêng. Các phép xử lý cơ sở trên cấu trúc là:

- ⊕ **union\_sets(x, y)** – Hợp nhất hai tập: tập chứa phần tử **x** với tập chứa phần tử **y**,
- ⊕ **find\_set(x)** – Xác định tập chứa phần tử **x**, chính xác hơn – trả về một phần tử (được gọi là *phần tử đại diện*) của tập,
- ⊕ **make\_set(x)** – Tạo tập riêng chứa phần tử **x** mới.

Nếu gọi hàm **find\_set()** với hai phần tử khác nhau và nhận được cùng một kết quả thì có nghĩa là 2 phần tử đó thuộc cùng một tập hợp. Trong trường hợp kết quả khác nhau – hai phần tử thuộc 2 tập khác nhau.

Cấu trúc dữ liệu mô tả dưới đây cho phép thực hiện các phép xử lý trên với độ phức tạp xấp xỉ O(1).

### Tổ chức dữ liệu

Các phần tử của mỗi tập hợp được lưu trữ dưới dạng cây và gốc của cây là phần tử đại diện cho tập.

Với tất cả các cây cần có *một mảng parent* lưu *móc nối tới phần tử cha* trong cây. Với *phần tử đại diện*, móc nối này chỉ tới chính nó.

Cách tổ chức giản đơn.

Đầu tiên ta sẽ xét cách tổ chức tầm thường, tuy hoạt động không thật hiệu quả nhưng tạo tiền đề để cải tiến và nâng cấp.

Để tạo tập mới (phép **make\_set(v)**) ta cho **parent[v]** chỉ tới chính nó, tạo cây mới chỉ chứa một phần tử.

Để hợp nhất 2 phần tử (phép **union\_sets(a, b)**) ta tìm các đại diện của tập chứa **a** và tập chứa **b**. Nếu 2 đại diện này trùng nhau – không phải làm gì! Cả 2 phần tử đều thuộc một tập. Nếu 2 đại diện khác nhau: cho móc nối cha của **b** chỉ tới **a** (hoặc ngược lại) và kết quả là 2 cây được hợp nhất thành một.

Việc tìm đại diện (phép **find\_set(v)**) hoàn toàn đơn giản: duyệt cây từ đỉnh **v** cho đến khi gặp phần tử chỉ tới chính nó. Việc duyệt có thể thực hiện theo sơ đồ đệ quy.

```

void make_set (int v) {
    parent[v] = v;
}

int find_set (int v) {
    if (v == parent[v])
        return v;
    return find_set (parent[v]);
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b)
        parent[b] = a;
}

```

Cách tổ chức trên đơn giản nhưng không hiệu quả. Sau nhiều lần hợp nhất cây có thể suy biến thành một chuỗi mốc nối dài và độ phức tạp của phép `find_set()` sẽ tiến dần tới  $O(n)$ .

Có hai kỹ thuật tối ưu hóa cho phép tăng đáng kể hiệu quả tìm kiếm (ngay cả khi áp dụng một cách riêng rẽ các kỹ thuật này).

### **Giải pháp heuristic nén đường đi**

Giải pháp này được áp dụng để nâng cao hiệu quả hoạt động của hàm tìm kiếm `find_set(v)`.

Khi gọi hàm này ta phải đi qua một con đường dài dẫn tới phần tử đại diện **p**. Có thể sê đơn giản hơn nếu `parent[]` của tất cả các phần tử tập hợp đều chỉ trực tiếp tới **p**? Như vậy vai trò của `parent[]` có thay đổi đôi chút. Mảng mốc nối chỉ tới nút cha đã bị nén để chỉ xa hơn. Tuy vậy cũng dễ dàng thấy rằng không thể tổ chức để `parent[]` luôn chỉ tới phần tử đại diện. Khi đó ta sẽ gặp khó khăn với phép hợp nhất `union_sets()`: khi hợp nhất ta phải cập nhật mốc nối tới đại diện ở  $O(n)$  phần tử tập hợp. Như vậy việc nén chỉ nên thực hiện bộ phận.

Hàm `find_set()` sẽ có dạng mới như sau:

```

int find_set (int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set (parent[v]);
}

```

Sự cải tiến đơn giản này đã mang lại hiệu quả mà ta đang chờ đợi. Đầu tiên, bằng cách gọi đệ quy ta *tìm được phần tử đại diện*, sau đó quá trình làm rỗng stack sẽ *gán vị trí phần tử này cho các mốc nối parent* ở các phần tử đã đi qua.

Giải thuật này có thể thực hiện theo sơ đồ lặp, nhưng khi đó ta phải *duyệt đường đi 2 lần*: một lần để tìm đại diện, lần thứ 2 – để cập nhật mốc nối, hiệu quả xử lý gần như không thay đổi.

### **Đánh giá hiệu quả của việc ứng dụng giải pháp heuristic nén đường đi**

Ta sẽ chứng minh việc nén đường đi theo giải pháp heuristic mang lại hiệu quả trung bình  $O(\log n)$  cho mỗi truy vấn.

Lưu ý rằng phép hợp nhất **`union_sets()`** đòi hỏi thực hiện **`find_set()`** hai lần và chi phí  $O(1)$  để cập nhật mốc nối, vì vậy ta chỉ phải tập trung đánh giá độ phức tạp  $O(m)$  của phép **`find_set()`**.

Gọi số đỉnh con của **v** (kể cả chính nó) là trọng số của **v** và ký hiệu là **w[v]**. Rõ ràng, trọng số các đỉnh chỉ có thể tăng trong quá trình xử lý truy vấn.

Gọi độ bao trùm của cạnh (**a**, **b**) là giá trị tuyệt đối hiệu các trọng số:  $|w[a] - w[b]|$ . Với cạnh (**a**, **b**) cố định độ bao trùm chỉ có thể tăng trong quá trình xử lý.

Các cạnh được phân loại thành các lớp. Một cạnh thuộc lớp **k** khi độ bao trùm của nó nằm trong khoảng  $[2^k, 2^{k+1}-1]$ . Như vậy lớp của cạnh là một số nằm trong phạm vi từ 0 đến  $\lceil \log n \rceil$ .

Chọn một đỉnh **x** cố định và xét sự thay đổi cạnh từ nó tới đỉnh cha. Ban đầu cạnh không tồn tại (chứng nào **x** còn là đỉnh đại diện). Sau đó **x** được nối tới một đỉnh nào đó (do kết quả hợp nhất tập) và rồi mốc nối lại thay đổi bởi hiệu ứng nén khi hàm **`find_set()`** được gọi.

Xét hoạt động của một lần gọi hàm **`find_set()`**: cây được duyệt theo một đường đi nào đó, mốc nối parent của các đỉnh trên đường đi bị thay bằng mốc nối chỉ tới đỉnh đại diện. Xet đường đi này và loại bỏ cạnh cuối cùng ở mỗi lớp (tức là không quá 1 cạnh ở mỗi lớp trong số các lớp từ 0 đến  $\lceil \log n \rceil$ ). Như vậy có  $O(\log n)$  cạnh bị loại từ mỗi truy vấn.

Xét các cạnh còn lại của đường đi này. Nếu một cạnh có lớp là **k** thì có nghĩa là trên đường đi còn có ít nhất một cạnh lớp **k** nữa (vì nếu nó là duy nhất ở lớp **k** thì đã bị xóa). Do đó, sau khi nén đường đi cạnh này được thay thế bởi cạnh có lớp tối thiểu

là  $k+1$ . Trọng số của cạnh thể giảm nên với mỗi đỉnh được duyệt qua bởi **find\_set()** cạnh từ nó tới đỉnh cha hoặc bị xóa hoặc chuyển sang lớp cao hơn.

Tổng hợp các đánh giá trên ta thấy độ phức tạp xử lý  $m$  truy vấn sẽ là  $O((n+m)\log n)$ . Khi  $m \geq n$  độ phức tạp trung bình xử lý một truy vấn có bậc  $O(\log n)$ .

## Giải pháp heuristic hợp nhất theo bậc của cây

Một giải pháp khác nâng cao hiệu quả giải thuật là cải tiến cách hợp nhất hai tập. Giải pháp này kết hợp với nén đường đi sẽ cho phép xử lý mỗi truy vấn với độ phức tạp gần như là một hằng.

Phép hợp nhất tập **union\_sets()** được thực hiện theo 2 bước:

- Lựa chọn cây đưa đi kết nối dựa vào một trong 2 tiêu chí đánh giá bậc của cây:
  - ♣ Xác định bậc của cây theo số lượng đỉnh,
  - ♣ Xác định bậc của cây theo độ sâu, hay chính xác hơn – giới hạn trên của độ sâu,
- Cây có bậc thấp hơn sẽ được gắn và cây có bậc cao.

Giải thuật hợp nhất tập *dựa trên số đỉnh* của cây sẽ có dạng như sau:

```
void make_set (int v) {  
    parent[v] = v;  
    size[v] = 1;  
}  
  
void union_sets (int a, int b) {  
    a = find_set (a);  
    b = find_set (b);  
    if (a != b) {  
        if (size[a] < size[b])  
            swap (a, b);  
        parent[b] = a;  
        size[a] += size[b];  
    }  
}
```

Giải thuật hợp nhất tập *dựa theo độ sâu* của cây sẽ có dạng như sau:

```
void make_set (int v) {  
    parent[v] = v;  
    rank[v] = 0;  
}
```

```

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            ++rank[a];
    }
}

```

Cả hai phương án là tương đương trên quan điểm hiệu quả, vì vậy khi triển khai ứng dụng có thể lựa chọn phương pháp bất kỳ.

### *Dánh giá hiệu quả của giải pháp heuristic hợp nhất theo bậc*

Ta sẽ chứng minh độ phức tạp xử lý hợp nhất tập với sự *lựa chọn heuristic theo bậc và không áp dụng kỹ thuật nén đường đi* là  $O(\log n)$ .

Không phụ thuộc vào cách lựa chọn kiểu xác định bậc ta sẽ chứng minh độ sâu của mỗi cây sẽ có bậc  $O(\log n)$ , điều này tự động dẫn đến độ phức tạp xử lý `find_set()` có bậc lô ga rit và kéo theo độ phức tạp xử lý `union_sets()` có sẽ có bậc lô ga rit.

Đầu tiên xét việc *dánh giá bậc theo độ sâu* của cây.

Bằng phương pháp quy nạp ta sẽ chứng minh, nếu bậc của cây bằng  $k$  thì cây phải có tối thiểu  $2^k$  đỉnh:

- Với  $k = 0$  – đó là điều hiển nhiên,
- Giả thiết điều cần chứng minh là đúng với  $k-1$ . Khi nén đường đi độ sâu của cây chỉ có thể giảm. Độ sâu của cây sẽ tăng từ  $k-1$  lên  $k$  khi gắn vào nó cây bậc  $k-1$ . Khi gắn 2 cây độ sâu  $k-1$  ta được cây độ sâu  $k$  và tổng số đỉnh, từ giả thiết quy nạp – sẽ không ít hơn  $2^k$ , đó là điều phải chứng minh.

Xét việc *dánh giá bậc theo số đỉnh* của cây.

Nếu cây có số lượng đỉnh là  $k$  thì độ sâu của nó không vượt quá  $\lfloor \log k \rfloor$ . Điều này có thể chứng minh bằng quy nạp:

- Với  $k = 1$  – kết luận trên là hiển nhiên,

💡 Xét việc hợp nhất 2 cây bậc  $\mathbf{k}_1$  và  $\mathbf{k}_2$ . Theo giả thiết quy nạp, độ sâu của cây tương ứng không vượt quá  $\lfloor \log k_1 \rfloor$  và  $\lfloor \log k_2 \rfloor$ . Không mất tính chất tổng quát, có thể coi  $\mathbf{k}_1 \geq \mathbf{k}_2$ . Sau khi hợp nhất, độ sâu của cây có  $\mathbf{k}_1 + \mathbf{k}_2$  đỉnh sẽ là

$$h = \max(\lfloor \log k_1 \rfloor, 1 + \lfloor \log k_2 \rfloor).$$

$$\begin{aligned} h &\stackrel{?}{\leq} \lfloor \log(k_1 + k_2) \rfloor, \\ 2^h &= \max(2^{\lfloor \log k_1 \rfloor}, 2^{\lfloor \log 2k_2 \rfloor}) \stackrel{?}{\leq} 2^{\lfloor \log(k_1 + k_2) \rfloor}, \end{aligned}$$

Vấn đề còn lại chỉ là phải chứng minh:

Nhưng đây là điều khá hiển nhiên bởi vì  $\mathbf{k}_1 \leq \mathbf{k}_1 + \mathbf{k}_2$  và  $2\mathbf{k}_2 \leq \mathbf{k}_1 + \mathbf{k}_2$ .

## Hợp nhất các giải pháp heuristic – nén đường đi kết hợp với chọn bậc

Việc kết hợp các giải pháp heuristic ở 2 hàm xử lý sẽ làm cho độ phức tạp xử lý mỗi truy vấn trở thành một hằng! Chứng minh điều này khá phức tạp và đã được nêu trong nhiều tài liệu khác nhau, ví dụ của tác giả Tarjan năm 1975, Kurt Mehlhorn và Peter Sanders năm 2008.

Người ta đã chứng minh được rằng, khi áp dụng kết hợp các giải pháp heuristic độ phức tạp xử lý mỗi truy vấn sẽ là  $O(\alpha(n))$ , trong đó  $\alpha(n)$  là hàm nghịch đảo Akkerman có độ tăng rất chậm, đến mức trong phạm vi  $n \leq 10^{600}$   $\alpha(n)$  có giá trị không quá 4.

Vì vậy có thể nói hệ thống các tập không giao nhau hoạt động với độ phức tạp gần như là một hằng.

Giải thuật sau hiện thực hóa việc kết hợp nén đường đi với xác định bậc theo độ sâu:

```
void make_set (int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find_set (int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set (parent[v]);
}
```

```

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            ++rank[a];
    }
}

```

## Ứng dụng của hệ thống các tập không giao nhau

Hệ thống các tập không giao nhau có thể áp dụng trực tiếp hoặc với một số cải biến nhỏ để giải một cách hiệu quả nhiều bài toán tin học khác nhau.

### Quản lý thành phần liên thông của đồ thị

Đây là ứng dụng hiển nhiên của hệ thống các tập không giao nhau và là động lực đưa tới sự hình thành lý thuyết về hệ thống các tập không giao nhau.

Bài toán được phát biểu một cách hình thức như sau: Xuất phát từ đồ thị rỗng ban đầu người ta lần lượt bổ sung các đỉnh và cạnh vô hướng đan xen với truy vấn (**a**, **b**) – “các đỉnh **a** và **b** có cùng thuộc một nhóm liên thông hay không?”

Áp dụng các giải thuật đã nêu, ta có thời gian xử lý mỗi truy vấn gần như một hằng. Bài toán này cũng có thể giải quyết theo thuật toán Kruscal, nhưng ở đây ta có công cụ hiệu quả hơn nhiều, có độ phức tạp tỷ lệ tuyến tính với số truy vấn.

Đôi khi trong thực tế ta gặp bài toán nghịch đảo: Ban đầu có một đồ thị với số lượng đỉnh và cạnh nào đó. Cần xử lý các truy vấn xóa bớt cạnh.

Nếu bài toán cho ở chế độ offline, tức là ta biết được mọi truy vấn trước khi giải bài toán thì có thể đảo ngược quá trình xử lý: xuất phát từ đồ thị rỗng, duyệt từ truy vấn cuối cùng về đầu, thay việc xóa cạnh bằng bổ sung dần cạnh. Như vậy, có thể trực tiếp áp dụng giải thuật đã nêu, không cần phải bổ sung, sửa đổi.

### Tìm thành phần liên thông trên ảnh

*Bài toán:* Xét ảnh hình chữ nhật hình thành từ  $n \times m$  pixels. Ban đầu tất cả các pixels có màu trắng. Dần dần xuất hiện các pixels màu đen. Hãy xác định kích thước các miền liên thông màu trắng.

Để giải bài toán này ta duyệt tất cả các ô màu trắng, với mỗi ô – xét 4 ô kè cạnh, nếu là ô trắng thì gọi hàm **`union_sets()`** liên kết các ô này. Như vậy ta có DSU với  $n \times m$  đỉnh. Số lượng cây trong DSU sẽ là kết quả cần tìm.

Bài toán này có thể giải bằng phương pháp loang theo chiều sâu hoặc chiều rộng, nhưng nếu ứng dụng DSU ta chỉ cần xử lý theo dòng hoặc cột với yêu cầu bộ nhớ chỉ là  $O(\min(n,m))$ .

### Hỗ trợ lưu thông tin bổ sung với mỗi tập hợp

Các thông tin bổ sung có thể coi như *thuộc tính của đỉnh đại diện* và có thể lưu trữ độc lập với sơ đồ xử lý. Ví dụ, ta có thể lưu kích thước các miền liên thông.

Việc cập nhật thông tin bổ sung không làm thay đổi sơ đồ xử lý chung.

Như vậy cấu trúc dữ liệu này có tính mở và phục vụ được cho nhiều yêu cầu cụ thể khác nhau.

### Nén thông tin rời rạc trên đoạn thẳng

Nếu ta có một tập các phần tử và từ mỗi phần tử có một cạnh đi ra thì với DSU ta có thể nhanh chóng xác định được điểm cuối nếu bắt đầu chuyển từ điểm ban đầu cho trước.

Xét bài toán tô màu các đoạn con: Xét đoạn thẳng độ dài **L**. Ban đầu mọi đoạn con độ dài đơn vị kể từ đầu đoạn thẳng đều có màu số 0. Cho một số truy vấn dạng (**l, r, c**) – tô đoạn thẳng **[l, r]** bằng màu **c**. Hãy xác định màu của mỗi đoạn đơn vị. Các truy vấn được cho trước, tức là bài toán thuộc loại offline.

Để giải bài toán ta cần xây dựng cấu trúc DSU ở mỗi đoạn đơn vị (gọi ngắn gọn là ô) lưu mốc nối tới đoạn đơn vị bên phải gần nhất chưa được tô. Ban đầu mốc nối chỉ tới chính nó. Sau khi tô màu lần đầu tiên ô trước đoạn được tô chỉ tới ô sau đoạn đã tô.

Để giải bài toán ta xét các truy vấn *ngược từ cuối về đầu*. Để thực hiện truy vấn cần tìm ô trái nhất chưa được tô trong đoạn thẳng và chuyển mốc nối của nó sang cho ô tiếp theo bên phải.

Như vậy ta có một DSU với giải pháp nén đường đi nhưng không phân cấp theo bậc (vì ta rất cần biết ô nào trở thành đại diện sau khi hợp nhất). Chi phí xử lý có bậc  $O(\log n)$ .

Các hàm xử lý:

```
void init() {
    for (int i=0; i<L; ++i)
        make_set (i);
```

```

}

void process_query (int l, int r, int c) {
    for (int v=l; ; ) {
        v = find_set (v);
        if (v >= r) break;
        answer[v] = c;
        parent[v] = v+1;
    }
}

```

Cũng có thể áp dụng kỹ thuật phân bậc ở đây: dùng mảng **end[]** lưu điểm cuối (điểm phải nhất) của mỗi tập. Khi đó việc hợp nhất hai tập có thể thực hiện theo bậc của tập thông qua việc cập nhật giới hạn phải mới. Độ phức tạp của giải thuật sẽ là  $O(\alpha(n))$ .

### Quản lý khoảng cách tới đỉnh đại diện

Trong một trường hợp ta cần phải tính khoảng cách theo cạnh từ một đỉnh tới gốc. Nếu không có nén đường đi thì mọi việc đều đơn giản: khoảng cách bằng số lần gọi đệ quy trong hàm **find\_set()**.

Nhưng việc nén đã biến đường đi trên vài cạnh thành một cạnh. Như vậy ở mỗi đỉnh cần lưu thêm thông tin phụ: độ dài của cạnh này tính từ đỉnh đó tới đỉnh cha mà nó chỉ tới. Khi đó mảng **parent[]** lưu không phải một số nguyên mà là cặp giá trị: *đỉnh cha và khoảng cách tới đó*.

Chương trình sẽ có dạng:

```

void make_set (int v) {
    parent[v] = make_pair (v, 0);
    rank[v] = 0;
}

pair<int,int> find_set (int v) {
    if (v != parent[v].first) {
        int len = parent[v].second;
        parent[v] = find_set (parent[v].first);
        parent[v].second += len;
    }
    return parent[v];
}

void union_sets (int a, int b) {
    a = find_set (a).first;
    b = find_set (b).first;
    if (a != b) {
        if (rank[a] < rank[b])

```

```

        swap (a, b);
parent[b] = make_pair (a, 1);
if (rank[a] == rank[b])
    ++rank[a];
}
}

```

Quản lý tính chẵn lẻ của độ dài đường đi và kiểm tra khả năng tồn tại đồ thị hai phía  
Nếu tính được độ dài đường đi thì tại sao lại phải xét riêng việc đánh tính chẵn – lẻ  
của độ dài?

Vấn đề ở chỗ là tính chẵn lẻ của độ dài đường đi là cần thiết để kiểm tra, sau khi bổ  
sung cạnh thành phần liên thông có trở thành đồ thị hai phía hay không?

Để giải quyết vấn đề này ta tạo DSU quản lý các thành phần liên thông và lưu ở mỗi  
đỉnh tính chẵn lẻ của độ dài đường đi. Khi đó, mỗi lần bổ sung thêm cạnh ta dễ dàng  
biết được việc bổ sung này có phá vỡ tính hai phía của đồ thị hay không. Nếu các  
đỉnh của cạnh đề thuộc một thành phần liên thông và có độ chẵn lẻ của độ dài đường  
đi tới đỉnh giống nhau thì cạnh bổ sung sẽ tạo ra một chu trình độ dài lẻ và phá vỡ  
tính hai phía của đồ thị.

Điều rắc rối chủ yếu ở đây là ta phải cẩn thận lưu ý tính chẵn lẻ khi hợp nhất hai tập  
trong hàm **`union_sets()`**.

Khi bổ sung cạnh (**a**, **b**) nối 2 thành phần liên thông thành một, ta phải gắn một cây  
vào cây khác sao cho **a** và **b** có độ chẵn lẻ khác nhau.

Ta sẽ dẫn xuất công thức tính độ chẵn lẻ cho một đại diện khi gắn nó tới đại diện  
của tập kia.

Ký hiệu **x** là độ chẵn lẻ của đường đi từ **a** tới đại diện của nó và **y** – độ chẵn lẻ  
của đường đi từ **b** tới đại diện của mình. Gọi **t** là độ chẵn lẻ cần tìm cho đại diện  
được gắn vào tập mới. Nếu **a** được gắn vào hợp như một cây con thì sau khi hợp  
nhất độ chẵn lẻ của **b** không thay đổi và vẫn bằng **y**, còn độ chẵn lẻ của **a** sẽ là **x⊕t**,  
trong đó  $\oplus$  là phép tính **XOR**. Điều ta cần là độ chẵn lẻ ở **a** và ở **b** phải khác nhau,  
tức là **x⊕t⊕y = 1**, từ đây suy ra **t = x⊕y⊕1**.

Như vậy, không quan trọng việc cây nào sẽ được mang đi gắn như một cây con, công  
thức trên phải được sử dụng để tính độ chẵn lẻ từ một đại diện tới đại diện kia.

Trong chương trình, cũng giống như ở mục trên, với mỗi đỉnh cần lưu cặp giá trị trong **parent** []. Ngoài ra, cần có mảng **bipartite** [] đánh dấu tập có phải là đồ thị hai phía hay không.

```
void make_set (int v) {
    parent[v] = make_pair (v, 0);
    rank[v] = 0;
    bipartite[v] = true;
}

pair<int,int> find_set (int v) {
    if (v != parent[v].first) {
        int parity = parent[v].second;
        parent[v] = find_set (parent[v].first);
        parent[v].second ^= parity;
    }
    return parent[v];
}

void add_edge (int a, int b) {
    pair<int,int> pa = find_set (a);
    a = pa.first;
    int x = pa.second;

    pair<int,int> pb = find_set (b);
    b = pb.first;
    int y = pb.second;

    if (a == b) {
        if (x == y)
            bipartite[a] = false;
    }
    else {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = make_pair (a, x ^ y ^ 1);
        bipartite[a] &= bipartite[b];
        if (rank[a] == rank[b])
            ++rank[a];
    }
}

bool is_bipartite (int v) {
    return bipartite[ find_set(v) .first ];
}
```

## Giải thuật tìm min trên một đoạn (*Range Minimum Query – RMQ*)

Xét bài toán tìm RMQ với độ phức tạp  $O(\alpha(n))$  cho bài toán truy vấn offline. Bài toán có thể phát biểu một cách hình thức như sau: Phải tổ chức dữ liệu xử lý hai loại truy vấn: **insert(i)**,  $i = 1, 2, \dots, n$  – bổ sung một số vào dãy, mỗi số chỉ bổ sung một lần và **extract\_min()** cạnh nó.

Giả thiết dãy các truy vấn đã biết trước, tức là ta có bài toán offline.

Đường lối giải bài toán là như sau. Thay vì lần lượt trả lời các truy vấn theo trình tự đã cho ta xét  $i = 1, 2, \dots, n$  và xác định nó là câu trả lời cho truy vấn nào. Để làm được điều đó ta phải tìm truy vấn đầu tiên không trả lời được đúng sau phép **insert(i)** của số đó. Dễ dàng thấy rằng đó chính là truy vấn có câu trả lời là  $i$ .

Như vậy ta có sơ đồ xử lý giống như ở bài toán tô màu đoạn thẳng đã xét ở trên.

Độ phức tạp trung bình cho việc xử lý một truy vấn là  $O(\log n)$  nếu không áp dụng giải pháp heuristic theo bậc và với mỗi phần tử chỉ lưu mốc nối phải tới kết quả của truy vấn **extract\_min()** và có sử dụng kỹ thuật nén đường đi đối với các mốc nối này khi hợp nhất.

Độ phức tạp xử lý sẽ là  $O(\alpha(n))$  nếu áp dụng giải pháp heuristic theo bậc và với mỗi tập – lưu số của vị trí kết thúc của tập (ở phương án giải trước thông tin này nhận được một cách tự động vì mốc nối chỉ luôn luôn hướng sang phải, còn ở phương án giả này – phải lưu trữ một cách tường minh).

## Tìm cha chung nhỏ nhất trong cây (*Lowest Common Ancestor – LCA*)

Cho cây **G** với  $n$  đỉnh và  $m$  truy vấn dạng  $(a_i, b_i)$ . Với mỗi truy vấn cần tìm cha chung nhỏ nhất của 2 đỉnh  $a_i$  và  $b_i$ , tức là tìm đỉnh  $c_i$  xa gốc nhất và là cha chung của các đỉnh  $a_i$  và  $b_i$ .

Bài toán được xét trong chế độ offline, tức là đã biết trước mọi truy vấn. Tarjan đã đề xuất giải thuật trả lời tất cả các truy vấn với chi phí thời gian bậc  $O(n+m)$ , tức là  $O(1)$  với mỗi truy vấn.

## Giải thuật Tarjan

### Sơ đồ xử lý

Bản chất của giải thuật là tiến hành loang theo chiều sâu, bắt đầu từ gốc của cây và lần lượt tìm ra câu trả lời cho các truy vấn. Kết quả truy vấn  $(v, u)$  được xác định khi loang tới đỉnh  $u$  và đã thăm đỉnh  $v$  hoặc ngược lại.

Trong quá trình loang theo chiều sâu sẽ có lúc ta tới đỉnh **v** và đã thăm các đỉnh con của nó. Giả thiết tồn tại truy vấn (**v**, **u**), trong đó **u** đã được thăm.

Khi đó **LCA(v, u)** sẽ là chính đỉnh **v** hoặc một trong số các đỉnh cha của **v**. Như vậy phải tìm đỉnh cha thấp nhất của **v** (kể cả chính nó) nhận **u** là đỉnh con.

Với một **v** cố định, theo dấu hiệu “*đỉnh cha thấp nhất của v và cũng là cha của một đỉnh nào đó khác*” các đỉnh của cây bị phân rã thành các lớp không giao nhau. Với mỗi đỉnh cha **p ≠ v** sẽ có một lớp chứa **p** và các cây con có đỉnh là con của **p** đồng thời nằm “bên trái” trên đường tới **v** (tức là đã được thăm trước khi tới **v**).

Các lớp này có thể được quản lý một cách hiệu quả bằng cấu trúc hệ thống tập không giao nhau, mỗi lớp tương ứng với một tập với **p** là đỉnh đại diện và được lưu trong mảng **ancestor[]**.

Giả thiết trong khi loang theo chiều sâu ta tới đỉnh **v**. Đặt **v** vào một tập không giao nhau riêng, **ancestor[v] = v**. Việc loang theo chiều sâu đòi hỏi duyệt tất cả các cạnh (**v, to**) đi ra từ **v**. Với mỗi đỉnh **to** trong số trên đầu tiên ta phải loang theo chiều sâu từ nó, sau đó bằng phép **union\_sets()** bổ sung đỉnh này cùng tất cả các đỉnh của cây con vào lớp chứa đỉnh **v** và cập nhật lại giá trị **ancestor[]** của đại diện thành **v** (vì trong quá trình hợp nhất đại diện có thể bị thay đổi). Sau khi xử lý xong các cạnh ta duyệt mọi truy vấn dạng (**v, u**), nếu **u** đã được thăm trong quá trình loang theo chiều sâu thì ghi nhận kết quả truy vấn **LCA(v, u) = ancestor[find\_set(u)]**.

Không khó để nhận thấy là với mỗi truy vấn sự kiện một đỉnh là đang xét và đỉnh kia đã được thăm chỉ xảy ra một lần.

### **Đánh giá độ phức tạp**

Độ phức tạp của việc loang theo chiều sâu là  $O(n)$ ,

Độ phức tạp của việc hợp nhất các tập với mọi lần hợp nhất trong khi duyệt là  $O(n)$ ,

Với mỗi truy vấn: 2 lần kiểm tra điều kiện xử lý với chi phí  $O(1)$ .

Tổng hợp, ta có độ phức tạp của giải thuật là  $O(n+m)$ , với  $m$  đủ lớn (tức là  $n = O(m)$ ) ta có chi phí xử lý mỗi truy vấn là  $O(1)$ .

## Chương trình

```
const int MAXN = максимальное число вершин в графе;
vector<int> g[MAXN], q[MAXN]; // lưu cây và truy vấn
int dsu[MAXN], ancestor[MAXN];
bool u[MAXN];

int dsu_get (int v) {
    return v == dsu[v] ? v : dsu[v] = dsu_get (dsu[v]);
}

void dsu_unite (int a, int b, int new_ancestor) {
    a = dsu_get (a), b = dsu_get (b);
    if (rand() & 1) swap (a, b);
    dsu[a] = b, ancestor[b] = new_ancestor;
}

void dfs (int v) {
    dsu[v] = v, ancestor[v] = v;
    u[v] = true;
    for (size_t i=0; i<g[v].size(); ++i)
        if (!u[g[v][i]]) {
            dfs (g[v][i]);
            dsu_unite (v, g[v][i], v);
        }
    for (size_t i=0; i<q[v].size(); ++i)
        if (u[q[v][i]]) {
            printf ("%d %d -> %d\n", v+1, q[v][i]+1,
                   ancestor[dsu_get (q[v][i]) ]+1);
        }
}

int main() {
    ... nhập dữ liệu cây ...

    // đọc và xử lý lưu các truy vấn
    for (;;) {
        int a, b = ...; // thông tin về một truy vấn
        --a, --b;
        q[a].push_back (b);
        q[b].push_back (a);
    }

    // loang theo chiều sâu và xử lý truy vấn
    dfs (0);
}
```

## Hợp nhất các cấu trúc dữ liệu khác nhau

Một trong các phương pháp lưu trữ DSU là dùng *danh sách tương minh* lưu trữ các phần tử của mỗi tập. Mỗi phần tử được lưu trữ cùng với mốc nối tới đại diện của tập chứa nó.

Thoạt nhìn, có vẻ cấu trúc dữ liệu này kém hiệu quả: khi hợp nhất hai tập ta phải bổ sung một danh sách vào danh sách khác và cập nhật lại mốc nối tới đại diện ở các phần tử của một danh sách.

Tuy nhiên nếu áp dụng giải pháp heuristic theo bậc ta luôn ghép tập nhỏ hơn vào tập lớn với chi phí thời gian tỷ lệ kích thước tập nhỏ, còn việc tìm phần tử đại diện – luôn luôn có chi phí  $O(1)$ .

*Dánh giá độ phức tạp:* Giả thiết cần thực hiện  $m$  truy vấn. Có định một phần tử  $\mathbf{x}$  và xét các tác động của **union\_set()** lên nó. Khi lần đầu tiên tác động lên nó,  $\mathbf{x}$  sẽ rơi vào tập mới có ít nhất 2 phần tử, khi tác động lần thứ 2, tập mới chứa nó có ít nhất 4 phần tử, . . . . Như vậy tối đa số lần tác động lên  $\mathbf{x}$  là  $\lceil \log n \rceil$ . Như vậy với toàn bộ tập, số lần tác động là  $O(n \log n)$ . Với mỗi truy vấn ta cần  $O(1)$  để xử lý.

Tổng cộng, độ phức tạp của giải thuật là  $O(m+n \log n)$ .

Các hàm xử lý:

```
vector<int> lst[MAXN];
int parent[MAXN];

void make_set (int v) {
    lst[v] = vector<int> (1, v);
    parent[v] = v;
}

int find_set (int v) {
    return parent[v];
}

void union_sets (int a, int b) {
    a = find_set (a);
    b = find_set (b);
    if (a != b) {
        if (lst[a].size() < lst[b].size())
            swap (a, b);
        while (!lst[b].empty()) {
            int v = lst[b].back();
            lst[b].pop_back();
            parent[v] = a;
        }
    }
}
```

```

        parent[v] = a;
        lst[a].push_back(v);
    }
}

```

Tư tưởng kết nối với heuristic theo bậc có thể triển khai hiệu quả ở các bài toán khác.

Ví dụ, xét bài toán: cho cây, trên mỗi lá của cây có ghi một số (có thể giống nhau ở các lá khác nhau). Yêu cầu xác định với mỗi đỉnh của cây số lượng các số khác nhau trong cây con của đỉnh.

Theo tư tưởng đã xét, sơ đồ xử lý sẽ bao gồm các bước sau. Tiến hành loang theo chiều sâu trên cây và trả về con trỏ tới tập **set** – danh sách các số ở cây con. Để dẫn xuất kết quả ở mỗi đỉnh (không phải là lá) cần loang theo chiều sâu của các cây con thuộc đỉnh đang xét, hợp nhất tất cả các **set** nhận được. Kích thước của tập kết quả là nghiệm cần tìm.

Độ phức tạp của giải thuật là  $O(n \log^2 n)$  vì việc bổ sung phần tử vào set có độ phức tạp  $O(\log n)$ .

### **Tìm cầu của đồ thị trong chế độ online với độ phức tạp $O(\alpha(n))$**

Một trong những mặt mạnh của DSU là cho phép *lưu cấu trúc của cây* ở cả dưới dạng *nén* và *không nén*.

Dạng nén dùng để hợp nhất nhanh 2 cây và kiểm tra hai đỉnh có thuộc một cây hay không, dạng không nén – để tìm đường đi giữa hai đỉnh hoặc các phép xử lý khác khi duyệt cây.

Như vậy ngoài mảng **parent**[] phục vụ thông tin nén có thể còn cần thêm mảng **real\_parent**[] lưu thông tin không nén. Sự tồn tại của mảng thứ 2 không ảnh hưởng đến độ phức tạp của giải thuật vì sự thay đổi trong đó chỉ xảy ra ở phép hợp nhất cây và cũng chỉ thay đổi ở một phần tử.

Nhiều bài toán thực tế đòi hỏi kết nối 2 cây bằng cạnh cho trước không nhất thiết xuất phát từ gốc. Như vậy buộc phải kết nối cây vào một trong 2 đỉnh đã chỉ định bằng cách biến gốc của cây mang đi kết nối thành đỉnh con của đỉnh kia trong cạnh chỉ định kết nối.

Có vẻ như phép kết nối phức tạp và rất tốn thời gian vì khi gắn một cây vào đỉnh v ta phải cập nhật tất cả các mốc nối trong **parent**[] và **real\_parent**[] khi đi từ **v** tới gốc của cây.

Trên thực tế, tình hình không đến nỗi tồi tệ như vậy vì ta sẽ gắn cây có bậc nhỏ hơn vào cây lớn và độ phức tạp trung bình của việc kết nối là  $O(\log n)$ .

## Sơ lược về lịch sử DSU

DSU ra đời cách đây khá lâu. Cấu trúc dữ liệu này được dùng để mô tả rừng cây và được Galler, Fisher mô tả năm 1964 trong bài báo "*An Improved Equivalence Algorithm*". Tuy vậy việc phân tích, đánh giá độ phức tạp được thực hiện muộn hơn rất nhiều.

Giải pháp heuristic nén đường đi và hợp nhất theo bậc được đề xuất bởi McIlroy cùng Morris và độc lập với họ – Tritter.

Trong một thời gian dài độ phức tạp của DSU được biết đến như  $O(\log^* n)$  do Hopcroft và Ullman đưa ra năm 1973. Đó là một hàm tăng chậm, nhưng vẫn còn nhanh hơn rất nhiều so với hàm Akkerman  $\alpha(n)$ .

Đánh giá  $O(\alpha(n))$  được Tarjan đưa ra năm 1975 trong công trình "*Efficiency of a Good But Not Linear Set Union Algorithm*". Muộn hơn, vào năm 1985 Tarjan cùng với Leeuwen đã công bố một số kết quả đánh giá độ phức tạp của các giải pháp heuristic áp dụng với DSU trong bài báo "*Worst-Case Analysis of Set Union Algorithms*".

Cuối cùng, vào năm 1989 trong công trình "*The cell probe complexity of dynamic data structures*" Fredman và Saks đã chứng minh được rằng giải thuật bất kỳ dựa cơ sở trên DSU có độ phức tạp trung bình không vượt quá  $O(\alpha(n))$ .

Tuy nhiên, cũng có một số tác giả như Zhang trong "*The Union-Find Problem Is Linear*", Wu, Otoo trong "*A Simpler Proof of the Average Case Complexity of Union-Find with Path Compression*" không đồng tình với đánh giá trên và cho rằng DSU cá áp dụng các giải pháp heuristic làm việc với độ phức tạp  $O(1)$ .

## Bài tập

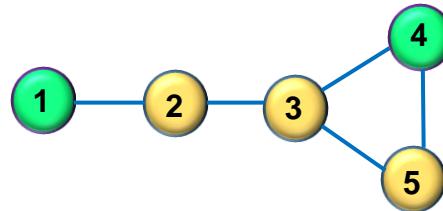
### VV07. BỘ TRÍ RŨ LIỆU

Tên chương trình: DATA.CPP

Mạng lưới truyền thông của một hãng thông tin lớn có  $n$  servers, đánh số từ 1 đến  $n$ . Có  $m$  đường cáp nối trực tiếp 2 servers với nhau, đường cáp thứ  $i$  nối 2 servers  $a_i$  và  $b_i$ ,  $a_i \neq b_i$ ,  $i = 1 \dots m$ . Hệ thống đường cáp được bố trí đảm bảo từ một server có thể truyền thông tin tới server khác, trực tiếp hoặc qua các servers trung gian. Không có đường nối một server với chính nó.

Tập servers **A** được gọi là ổn định cao nếu trong trường hợp một đường cáp bị sự cố thì server **x** bất kỳ ngoài tập **A** vẫn có thể nhận thông tin từ một trong số các servers thuộc **A**.

Với mạng nối các servers như ở hình bên tập các servers (1, 2) là tập ổn định cao.



Các dữ liệu quan trọng được lưu lặp lại như nhau trong các servers thuộc tập **A**. Để giảm chi phí lưu và bảo trì người ta muốn có tập **A** càng nhỏ càng tốt. Ngoài ra, để đánh giá độ linh hoạt của toàn hệ thống người ta cũng muốn biết có tất cả bao nhiêu cách chọn tập **A** với kích thước tối thiểu. Hai cách chọn gọi là khác nhau nếu tồn tại ít nhất một server có trong các chọn thứ nhất và không có trong các chọn thứ 2.

**Dữ liệu:** Vào từ file văn bản DATA.INP:

- ✚ Dòng đầu tiên chứa 2 số nguyên  $n$  và  $m$  ( $2 \leq n \leq 2 \times 10^5$ ,  $1 \leq m \leq 2 \times 10^5$ ),
- ✚ Dòng thứ  $i$  trong  $m$  dòng sau chứa 2 số nguyên  $a_i$  và  $b_i$  ( $1 \leq a_i, b_i \leq n$ ,  $a_i \neq b_i$ ).

**Kết quả:** Đưa ra file văn bản DATA.OUT đưa ra trên một dòng 2 số nguyên – kích thước tối thiểu của tập A và số cách chọn theo mô đun  $10^9 + 7$ .

**Ví dụ:**

DATA.INP	DATA.OUT
5 5 1 2 2 3 3 4 3 5 4 5	2 3



## *Giải thuật: Tìm cầu, chu trình và miền song liên thông của đồ thị.*

Nhận xét:

Các servers nối với nhau tạo thành một đồ thị liên thông, trong đó đỉnh là các servers,

Gọi B là tập các đỉnh tạo thành chu trình, nếu ta bỏ một cạnh nào đó trong chu trình thì giữa hai đỉnh bất kỳ thuộc B vẫn có đường đi, như vậy mọi đỉnh trong chu trình có kết nối như nhau không phụ thuộc vào việc có đường truyền nào bị sự cố,

Giả thiết x và y là 2 đỉnh có đường nối trực tiếp, cạnh (x, y) được gọi là *cầu* của đồ thị nếu việc *loại bỏ cạnh này sẽ làm tăng số phần liên thông* của đồ thị,

Như vậy, sự tồn tại cầu sẽ tác động lên cách xây dựng tập A,

Tìm cầu là bài toán chuẩn trong lý thuyết đồ thị,

Ta có thể tìm tất cả các cạnh là cầu của đồ thị, loại bỏ chúng, đồ thị sẽ phân rã thành các miền liên thông riêng biệt,

Ở mỗi miền liên thông chọn một đỉnh bất kỳ làm đại diện và thay toàn bộ miền liên thông bằng đỉnh đại diện,

Khôi phục lại các cầu đã loại bỏ ta được một cây và các nút lá là những nút cần chọn để xây dựng A,

Như vậy số lượng nút lá là lực lượng của tập A và là một trong số các kết quả cần xác định,

Nếu nút lá là nút đại diện cho miền liên thông chứa p đỉnh, thì với nút lá đó ta có p các chọn đỉnh tham gia vào A,

Số cách xây dựng tập A sẽ là tích các khả năng chọn ở mỗi lá.

*Tổ chức dữ liệu:* sử dụng các mảng như trong sơ đồ lý thuyết chung xác định cầu của đồ thị.

*Độ phức tạp của giải thuật:*  $O(n+m)$ .

## Chương trình:

```
#include <bits/stdc++.h>
#define NAME "data."
using namespace std;
typedef vector<vector<int>> vvi;
ifstream fi (NAME"inp");
ofstream fo (NAME"out");

const int64_t MOD = (int64_t)1e9 + 7;

void dfs(const vvi &graph,
         vector<bool> &used,
         set< pair<int, int>> &bridges,
         vector<int> &tin,
         vector<int> &fup,
         int &timer, int v, int p)
{
    used[v] = true;
    tin[v] = fup[v] = timer++;
    for (int u : graph[v])
    {
        if (u == p) continue;
        if (!used[u])
        {
            dfs(graph, used, bridges, tin, fup, timer, u, v);
            fup[v] = min(fup[v], fup[u]);
            if (fup[u] > tin[v]) bridges.insert({ min(v, u), max(v, u) });
        } else fup[v] = min(fup[v], tin[u]);
    }
}

void findComponent(const vvi &graph,
                   vector<bool> &used,
                   const set< pair<int, int>> &bridges,
                   vector<int> &getComponent,
                   int component, int &vs, int v)
{
    used[v] = true;
    getComponent[v] = component;
    vs++;
    for (int u : graph[v])
    {
        pair<int, int> p = { min(v, u), max(v, u) };
        if (bridges.find(p) != bridges.end()) continue;
        if (!used[u]) findComponent(graph, used, bridges, getComponent, component, vs, u);
    }
}

int main()
{
    int n, m;
```

```

fi>>n>>m;
vvi graph(n);
for (int i = 0; i < m; i++)
{
    int v, u;
    fi>>v>>u;
    --v, --u;
    graph[v].push_back(u);
    graph[u].push_back(v);
}
set< pair<int, int>> bridges;
vector<bool> used(n, 0);
vector<int> tin(n, 0);
vector<int> fup(n, 0);
int timer = 0;
for (int i = 0; i < n; i++)
    if (!used[i]) dfs(graph, used, bridges, tin, fup, timer, i, -1);
fill(used.begin(), used.end(), false);
vector<int> componentSize(n, 0);
vector<int> getComponent(n, 0);
int component = 0;
for (int i = 0; i < n; i++)
{
    if (!used[i])
    {
        int vs = 0;
        findComponent(graph, used, bridges, getComponent, component, vs, i);
        componentSize[component++] = vs;
    }
}
vector<int> deg(component, 0);
for (const auto &bridge : bridges)
{
    deg[getComponent[bridge.first]]++;
    deg[getComponent[bridge.second]]++;
}
int ans = 0;
int cnt = 1;
for (int i = 0; i < component; i++)
    if (deg[i] <= 1)
    {
        ans++;
        cnt = (cnt * 1LL * componentSize[i]) % MOD;
    }

fo<<ans<<' '<<cnt;
fo<<"\nTime: "<<clock() / (double)1000<<" sec";
return 0;
}

```

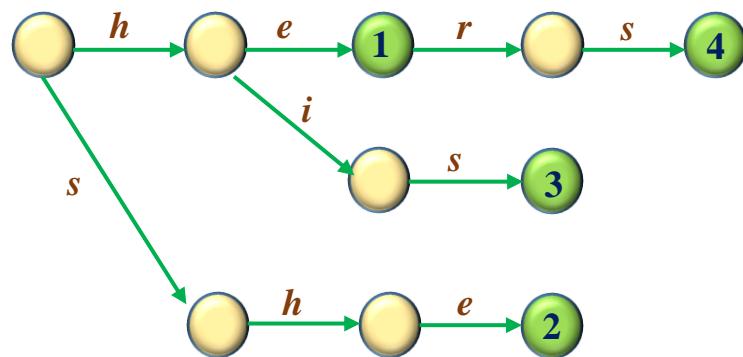


## Rừng số tìm kiếm nhanh

### Rừng cây

Rừng cây, hay ngắn gọn – *rừng* (*Trie*) là cấu trúc dữ liệu dạng cây liên kết nhiều cây độc lập. Thông thường rừng dùng để lưu trữ và xử lý các xâu. Giá trị của *cạnh* là một ký tự, đỉnh gồm 2 loại *trung gian* và *kết thúc*. Đường đi từ gốc tới đỉnh kết thúc xác định một xâu trong tập các xâu lưu trữ. Kích thước của rừng phụ thuộc tuyến tính vào tổng độ dài các xâu. Thời gian tìm kiếm tỷ lệ với độ dài mẫu cần tìm.

Ví dụ: xét rừng chứa các từ {*she, he, his, hers*}



### Xây dựng rừng

#### Ký hiệu

Các ký hiệu sau đây sẽ được sử dụng:

$\Sigma$  – bảng chữ cái,

$\mathbf{P} = \{\mathbf{P}_1, \mathbf{P}_2, \dots, \mathbf{P}_k\}$  – từ điển gồm các xâu xây dựng trên bảng chữ cái  $\Sigma$ ,

$\mathbf{N} = \sum_{i=1}^k |\mathbf{P}_i|$  – tổng độ dài các xâu,

Rừng được lưu trữ như tập các đỉnh, mỗi đỉnh có dấu hiệu đánh dấu đó là đỉnh kết thúc hay không và mốc nối tới đỉnh tiếp theo (hoặc NULL nếu là đỉnh lá của nhánh).

#### Các bước xây dựng

**S1.** Tạo cây từ một đỉnh (tạo đỉnh gốc),

**S2.** Bổ sung các phần tử vào cây:

- ✚ Duyệt từ gốc trở đi, lần lượt bổ sung các ký tự của  $\mathbf{P}_i$  vào cây, ghi nhận cạnh theo ký tự,
- ✚ Nếu  $\mathbf{P}_i$  kết thúc ở đỉnh  $v$ : ghi nhận dấu hiệu nhận dạng  $\mathbf{P}_i$  (ví dụ, lưu giá trị  $i$ ) và đánh dấu  $v$  như một đỉnh kết thúc,

- ✚ Nếu cạnh tương ứng với ký tự tiếp theo của  $P_i$  không có: tạo cạnh và đỉnh mới.

Thời gian xây dựng cây sẽ là  $O(|P_1| + |P_2| + \dots + |P_k|)$  vì thời gian tìm ký tự để chuyển tới là  $O(1)$ .

Mỗi đỉnh cần tối đa  $O(|\Sigma|)$  bộ nhớ, nên tổng bộ nhớ cần thiết là không quá  $O(n|\Sigma|)$ .

### ***Ưu và nhược điểm của cấu trúc rừng***

Rừng cho phép phát huy nhiều điểm ưu việt của cây tìm kiếm nhị phân và hàm băm, thực hiện một số phép xử lý mà riêng biệt cây tìm kiếm nhị phân hoặc hàm băm không thực hiện được. Rừng có thể được sử dụng như một công cụ tổ chức mảng kết hợp (*Associative Array*).

	<i>Rừng</i>	<i>Cây tìm kiếm nhị phân</i>	<i>Bảng Hàm băm</i>
<i>Bổ sung phần tử</i>	$O( S )$	$O( S  \log k)$	$O( S )$
<i>Xác định tất cả các khóa theo một trật tự cho trước</i>	$O(k)$	$O(k)$	$O(k \log k)$

Tuy vậy nó cũng có một số nhược điểm:

- ✚ Rừng được định hướng lưu trữ xâu hoặc ký tự, vì vậy giá trị khóa sẽ bị ràng buộc bởi kiểu đối tượng. Để khắc phục điều này cần dẫn xuất kiểu bất kỳ về dạng xâu, khi đó có thể dùng dữ liệu dạng bất kỳ làm khóa,
- ✚ Việc tổ chức mảng kết hợp dưới dạng rừng với khóa là xâu sẽ tốn quá nhiều bộ nhớ (ví dụ khi không có tiền tố chung bộ nhớ cần dùng sẽ là  $O(n|\Sigma|)$ ).

### ***Rừng số***

Xét việc lưu trữ và xử lý các số nguyên độ dài  $w$  bit. Các phép xử lý đối với những số được lưu trữ là các phép tính số học và lô gic, phép đầy bít, xác định địa chỉ lưu trữ, ..., mỗi phép xử lý cần thực hiện với chi phí thời gian  $O(1)$ . Cây Van Emde Boa (*Van Emde Boas tree*, *vEB tree*) cũng đảm bảo được nhiều trong số các yêu cầu đã nêu, nhưng ở đây ta sẽ xét một cấu trúc đơn giản và mạnh hơn – *rừng số*.

Rừng số là rừng trong đó dạng biểu diễn nhị phân của số (kể cả các số 0 không có nghĩa bên trái) đóng vai trò xâu trong cấu trúc rừng. Như vậy rừng sẽ có độ sâu  $w$ .

Các phép xử lý trên rừng số là ***insert***, ***remove***, ***find***, ***succ*** và ***pred***.

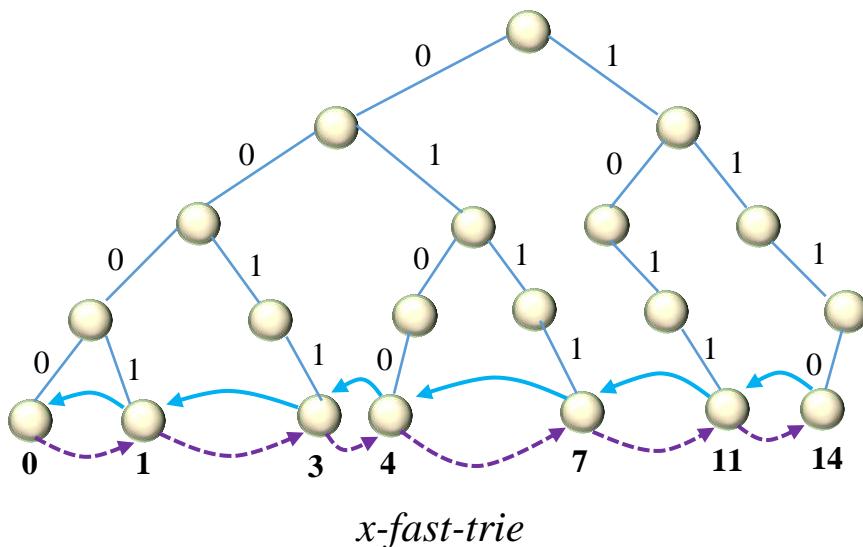
Bỏ sung đinh được thực hiện như ở rừng bình thường. Việc loại bỏ có thể thực hiện một cách đơn giản là xóa trường đánh dấu ở nút. Cũng có thể lưu số lượng nút được đánh dấu trong cây con và xóa cả cây con (xóa gốc cây con) khi số lượng này bằng 0.

### **Phép succ – Tìm phần tử tiếp theo**

Việc tìm phần tử tiếp theo được thực hiện bằng cách duyệt rừng từ gốc đến đỉnh không thể đi tiếp theo hướng cần tìm. Nếu không thể đi sang trái (theo cạnh 0) thì kết quả sẽ là phần tử min ở cây con bên phải. Nếu không thể đi sang phải thì lùi ngược lên chừng nào đỉnh còn là nút con phải. đỉnh trở thành nút con trái thì lùi tiếp cho đến khi đỉnh không có nút con phải (nếu không gặp tình huống này thì có nghĩa truy vấn được thực hiện với phần tử có giá trị lớn nhất trong rừng). Kết quả cần tìm sẽ là min ở cây con bên phải.

Ưu điểm của cấu trúc là *dễ lập trình*, chỉ cần  $O(n \times w)$  bộ nhớ và các phép xử lý đều thực hiện với *độ phức tạp*  $O(w)$ , tuy thua kém cây Van Emde Boas ở mặt tốc độ nhưng hơn ở tất cả các mặt còn lại, đặc biệt là về bộ nhớ!

### **Rừng số tìm kiếm nhanh (*x-fast-trie*)**



Lợi dụng tính chất tất cả các đường đi từ gốc tới lá đều cùng độ dài ta có thể đưa vào một số cải tiến để các phép tìm kiếm chỉ cần chi phí thời gian  $O(\log w)$ .

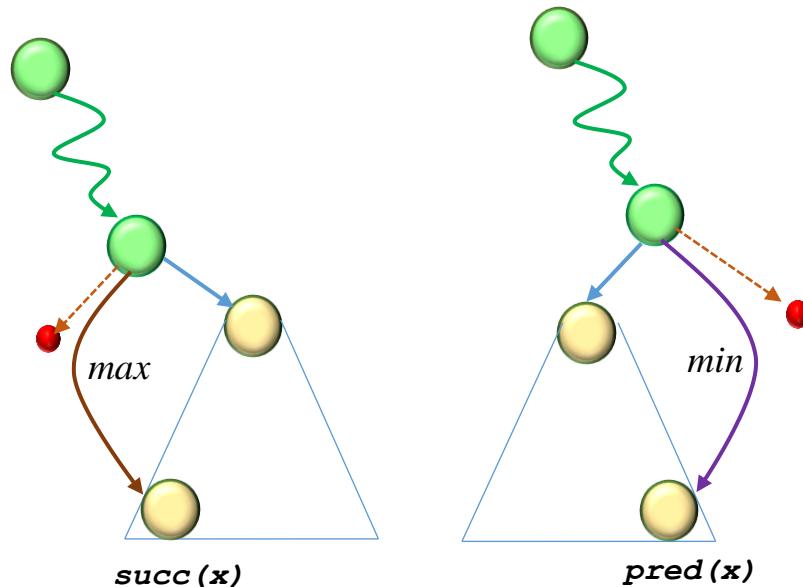
### **Phép succOrPred**

Lưu các phần tử vào danh sách mốc nối hai chiều theo trình tự xuất hiện trong rừng. Phép tìm kiếm succOrPred trả về phần tử trước hoặc sau tùy thuộc việc phần tử nào dễ dẫn xuất hơn trong tình huống cụ thể.

Bắt đầu duyệt từ gốc, ta sẽ đi hết phần tiền tố chung, sau đó tiếp tục, ta sẽ tới min ở cây con bên phải hoặc max ở cây con bên trái. Dựa vào danh sách mốc nối hai chiều, chỉ việc so sánh hai giá trị kề có thể xác định giá trị cần tìm.

Để tránh việc phải duyệt tiếp khi đi hết phần tiền tố chung ta bổ sung thêm các mốc nối chỉ trực tiếp tới phần tử min hoặc max ở những nơi tiềm năng có thể hết tiền tố chung, tức là ở các nút chỉ có một đường đi tiếp.

Với khóa tìm kiếm  $x$ , ở nút không có đường đi tiếp sang 0 mốc nối bổ sung sẽ chỉ tới phần tử lớn nhất trước  $x$ , tức là  $\text{pred}(x)$ , còn nếu không có đường đi sang 1 – mốc nối bổ sung chỉ tới phần tử nhỏ nhất sau  $x$ , tức là  $\text{succ}(x)$ .



### Phép insert

Với phép succOrPred và danh sách mốc nối hai chiều ta dễ dàng tìm được cặp phần tử trước và sau phần tử cần bổ sung, nơi bắt đầu đường đi mới cần đưa vào. Trừ nút lá, mỗi nút ở đường đi mới chỉ có một nút con. Mốc nối sẽ được thiết lập khi quay lui trong đệ quy.

```
// prefixes — HashMap tất cả các tiền tố của rừng
// Các nút của danh sách mốc nối và của cây được lưu theo cùng một kiểu: mốc nối phải và mốc nối trái và giá trị
// là số nguyên
// các số cần quản lý được lưu trong danh sách, ở rừng nút là chia giá trị 1, các nút còn lại — giá trị 0.
function insert(x: N):
    if x in prefixes
        return
    Node left = pred(x), right = succ(x), node = Node(x)
    // insert node giữa left và right trong danh sách mốc nối hai chiều các nút lá
    // Cho mốc nối chỉ tới phần tử trong danh sách để tìm kiếm nhanh khi chỉ có một nút con
    root = insertNode(root, w, node)
    prefixes.addAll(allPrefixes(x))
```

```

N insertNode(vertex: N, depth: unsigned int, node: N):
    if vertex == ∅
        vertex = Node(left = ∅, right = ∅, terminal = depth == 0)
    if depth == 0
        return vertex
    if bit(node.value, depth) == 0 // bit tương ứng với độ sâu đang xét
        vertex.left = insertNode(vertex.left, depth - 1, node)
    else
        vertex.right = insertNode(vertex.right, depth - 1, node)
    if vertex.left == ∅
        vertex.mark = HASNOLEFTSON
        vertex.left = node
    else if vertex.right == ∅
        vertex.mark = HASNORIGHTSON
        vertex.right = node
    else
        vertex.mark = HASALLSONS

```

## Phép `delete`

Việc xóa được thực hiện tương tự như bổ sung. Tại mỗi đỉnh của rừng cần lưu số lượng đỉnh ở các cây con. Nếu số lượng bằng 0 – xóa đỉnh đó, trong trường hợp ngược lại, khi một nút con bị xóa cần cập nhật các mốc nối tới *min/max*. Mốc nối tới *min/max* ở cây con bên *phải/trái* sẽ là *successor/predecessor* của đỉnh bị xóa.

Các phép insert và delete được thực hiện với độ phức tạp  $O(w)$ .

## Phép `binarySearch`

Cho đến đây ta chưa có thu hoạch gì đặc biệt: độ phức tạp của các phép xử lý vẫn là  $O(w)$ . Cần có giải pháp tác động lên khâu yếu nhất của các phép xử lý đã xét: tìm tiền tố chung lớn nhất.

Việc tìm kiếm này có thể thực hiện giải thuật tìm kiếm nhị phân.

Tạo mảng kết hợp (*Associative Array*) **HashMap** chứa mốc nối tới đỉnh của rừng theo tiền tố. Để bảo tồn vai trò các số 0 không có nghĩa cần dùng mặt nạ dạng **00...0...11...1** để tách và nhận dạng tiền tố, tiến hành tìm kiếm nhị phân theo độ dài tiền tố chung lớn nhất. Khi tìm được tiền tố lớn nhất – chuyển tới đỉnh tương ứng trong rừng. Đỉnh này chỉ có thể có một đỉnh con (nếu không tiền tố chung chưa dừng ở đây). Với chi phí  $O(1)$  ta tìm được min hoặc max và với chi phí  $O(1)$  – chuyển theo danh sách mốc nối nếu cần thiết.

Như vậy các phép **find**, **succ** và **pred** được thực hiện với chi phí  $O(\log w)$ .

## Rừng số tìm kiếm siêu nhanh (*y-fast-trie*)

Bây giờ ta sẽ cải tiến *x-fast-trie* thành rừng số tìm kiếm siêu nhanh *y-fast-trie* với bộ nhớ cần để lưu trữ rừng mới là  $O(n)$  và mọi phép xử lý có độ phức tạp  $O(\log w)$ , riêng các phép làm thay đổi rừng có độ phức tạp lớn hơn đôi chút.

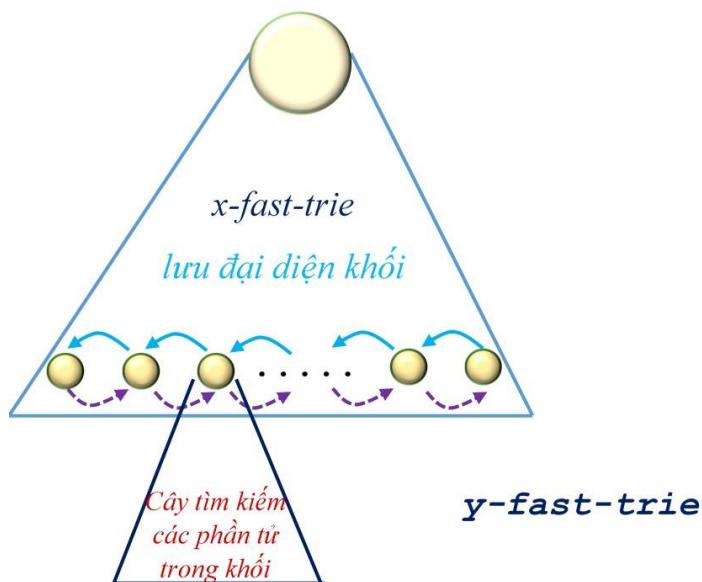
### Giảm kích thước bộ nhớ cần dùng

Giả thiết cần lưu trữ các số  $a_1 < a_2 < a_3 < \dots < a_n$ . Chọn một số nguyên  $k$  (vai trò của  $k$  và cách chọn sẽ được làm rõ trong quá trình xem xét giải thuật). Phân các số cần lưu trữ thành  $s$  khối  $\mathbf{B}$  kích thước từ  $k/2$  đến  $2 \times k$ :

$$\overbrace{a_1 < a_2 < a_3}^{B_1} \quad \overbrace{a_4 < \dots}^{B_2} \quad \dots < \overbrace{a_{n-1} < a_n}^{B_s}$$

Chọn ở mỗi khối một đại diện và nạp các đại diện này vào *x-fast-trie*. Rừng này sẽ có  $O(\frac{2 \cdot n \cdot w}{k})$  phần tử. Vì vậy nếu chọn  $k$  là bội của  $w$  thì bộ nhớ cần dùng sẽ là  $O(n)$ . Mỗi lá của *x-fast-trie* là đại diện của một khối, các phần tử trong khối (kể cả phần tử đại diện) được gắn vào rừng như một cây tìm kiếm nhị phân cân bằng, mỗi cây lưu trữ từ  $w/2$  đến  $2 \times w$  phần tử vì vậy có độ cao  $O(\log w)$ .

Tất cả các cây chiếm  $O(n)$  bộ nhớ, bản thân *x-fast-trie* cũng chiếm  $O(n)$  bộ nhớ, vì vậy *y-fast-trie* cũng chiếm  $O(n)$  bộ nhớ.



### Phép *find*

Tính **succ(x)** trong số các đại diện ở *x-fast-trie*, sau đó tìm tiếp kết quả ở cây gắn với nút lá  $x$  và ở cây gắn với nút lá **pred(x)** trong *x-fast-trie*. Đại diện của cây

không nhất thiết phải là phần tử nhỏ nhất hay lớn nhất, vì vậy phải tìm kết quả ở cả 2 cây (phần tử cần tìm nhất thiết phải nằm giữa phần tử trước và sau nó).

Chi phí tìm kiếm trong x-fast-trie là  $O(\log w)$ , chi phí tìm kiếm trong cây nhị phân cân bằng cũng là  $O(\log w)$ , vì vậy chi phí tìm kiếm chung sẽ là  $O(\log w)$ .

Các phép **succ** và **pred** được thực hiện tương tự.

### *Phép insert*

Tìm **succ (x)** và bổ sung nó vào lá của cây. Tuy nhiên có thể xuất hiện tình huống cây sẽ có  $2 \times w + 1$  nút, khi đó cần xóa cây khỏi x-fast-trie, chia các phần tử trong cây bị xóa thành 2 cây với số lượng nút tương ứng là  $w$  và  $w + 1$ , gắn lại hai cây này vào x-fast-trie.

### *Phép delete*

Thực hiện tương tự, nhưng nếu gặp trường hợp cây còn  $w/2 - 1$  phần tử thì hợp nhất cây này với một trong 2 cây cạnh nó. Nếu kết quả hợp nhất cho cây với số nút lớn hơn  $2 \times w$  thì xử lý theo cách đã nêu ở trên.

Các cây cân bằng hoặc cây đỏ - đen cho phép hợp nhất hay phân chia với chi phí thời gian tuyến tính. Vì vậy các phép bổ sung hay loại bỏ được thực hiện với chi phí thời gian  $O(w)$ .

### *Đánh giá độ phức tạp*

Các phép xử lý không làm thay đổi rurgeon trên các cây nhị phân như **insert**, **succ**, **pred** được thực hiện với độ phức tạp  $O(\log w)$ .

Các phép xử lý làm thay đổi rurgeon bên trên phức tạp hơn, nhưng chúng ít khi xảy ra.

Ta sẽ áp dụng kỹ thuật phân tích khâu hao để đánh giá độ phức tạp của các phép xử lý này. Việc hợp nhất hay tách mảng được thực hiện với độ phức tạp  $O(w)$ . Với việc phân chia, trường hợp xấu nhất xảy ra khi tiếp theo ta cần bổ sung phần tử, với kích thước hiện có  $w/2 - 1$  và  $2 \times w$ , hợp nhất lại ta có cây với số lượng phần tử lớn hơn  $2 \times w$ , phân chia chúng ta có 2 cây với số phần tử  $5 \times w/4$  ở mỗi cây. Với việc hợp nhất, tình huống xấu nhất xảy ra khi ta có  $w$  phần tử (xảy ra khi tách cây với  $2 \times w + 1$  phần tử). Nhưng điều này phần lớn các trường hợp không xảy ra ngay trên biên nên ta có dự trữ các phép xử lý với độ phức tạp thấp là  $O(\log w)$  đủ để bù trừ cho trường hợp xấu nhất –  $O(w)$ .

Cây Van Emde Boa cũng có các chỉ số đánh giá tương tự, nhưng ở đó cần tới  $O(n)$  bộ nhớ.

## Vùng đống Fibonacci heap

Đống Fibonacci được phát triển bởi Michael L.Fredman và Robert E.Tarjan vào năm 1984 và lần đầu tiên được công bố trong một tạp chí khoa học vào năm 1987. Tên của Fibonacci heap xuất phát từ dãy số Fibonacci được sử dụng trong phân tích thời gian chạy.

Đống Fibonacci là một cấu trúc dữ liệu đống bao gồm một bộ sưu tập các cây. Vì vậy, nó cũng được ghép vào cấu trúc cây. Thời gian thực hiện giải thuật của nó hơn rất nhiều của một đống nhị thức .

Hoạt động tìm kiếm ít nhất là trong thời gian  $O(1)$ , còn hoạt động chèn, giảm giá trị khóa, và kết nối trong thời gian tối thiểu là  $O(\log n)$ . Điều này có nghĩa là từ một cấu trúc dữ liệu nào, bất kỳ chuỗi các hoạt động  $a$  từ nhóm đầu tiên và các hoạt động  $b$  từ nhóm thứ hai sẽ mất  $O(a+b\log n)$  thời gian. Trong một đống nhị thức các hoạt động sẽ mất  $O((a+b)\log (n))$  thời gian, tốt hơn so với một đống nhị thức khi  $b$ .

Sử dụng Fibonacci đống cho hàng đợi ưu tiên làm cải thiện thời gian xử lý của thuật toán.

### Cấu trúc của Fibonacci heap

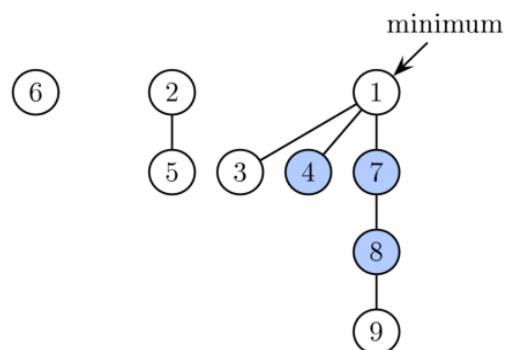
Một đống Fibonacci là một tập hợp các cây đáp ứng các điều kiện tối thiểu đống. Điều này ngụ ý rằng khóa của đống luôn luôn là gốc của một trong những cây trong đống. So với một đống nhị thức, cấu trúc của một đống Fibonacci là linh hoạt hơn. Cây không có một hình dạng theo quy định và trong trường hợp cực đoan đống có thể có tất cả các yếu tố trong một cây riêng biệt. Sự linh hoạt này cho phép một số hoạt động sẽ được thực hiện và làm trì hoãn công việc cho các hoạt động sau đó. Ví dụ đống sáp nhập được thực hiện đơn giản bằng cách ghép hai danh sách cây, và hoạt động giảm giá trị khóa đôi khi cắt một nút từ cha mẹ của nó và tạo thành một cây mới.

Đặc biệt, mức độ các nút được lưu giữ khá thấp, mỗi nút có bậc nhiều nhất là  $O(\log n)$  và kích thước của một cây con bắt nguồn từ một nút của mức độ k ít nhất là  $F_{k+2}$  trong đó  $F_k$  là thứ k số Fibonacci . Điều này đạt được bởi các quy tắc mà chúng ta có thể cắt giảm nhiều nhất là một con của mỗi nút không phải root. Khi đứa con thứ hai được cắt, nút chính nó cần phải được cắt giảm từ cha mẹ của nó và trở thành gốc rễ của một cây. Số cây được giảm trong hoạt động tối thiểu xóa, nơi cây được liên kết với nhau.

Là kết quả của một cấu trúc, một số hoạt động có thể mất một thời gian dài trong khi những hoạt động khác được thực hiện rất nhanh chóng. Giả sử rằng các hoạt động xảy ra rất nhanh và mất một ít thời gian. Thêm thời gian này sau đó kết hợp và trừ vào thời gian vận hành thực tế của hoạt động chậm. Số lượng thời gian lưu để sử dụng sau được đo tại bất kỳ thời điểm nào bằng một chức năng tiềm năng. Tiềm năng của một đống Fibonacci được cho bởi

$Tiềm\ năng = t + 2m$   
Trong đó : t là số cây trong đống Fibonacci  
m là số lượng các nút được đánh dấu.

Một nút được đánh dấu nếu ít nhất một trong các con của nó đã bị cắt từ nút này đã được thực hiện một đứa con của một nút khác (tất cả các rẽ là đánh dấu). Thời gian thực hiện cho một hoạt động được đưa ra bởi tổng thời gian thực tế và c lần sự khác biệt trong tiềm năng, trong đó c là một hằng số (lựa chọn để phù hợp với các yếu tố liên tục trong các ký hiệu O cho thời gian thực tế).



Trong ví dụ ta thấy nó có ba cây của độ 0, 1 và 3. Ba đỉnh được thể hiện trong màu xanh. Do đó, tiềm năng của heap là 9 (3 cây + 2 \* 3 đỉnh).

Như vậy, gốc rẽ của từng cây trong một đống có một đơn vị thời gian lưu trữ. Sau này đơn vị thời gian này có thể được sử dụng để liên kết cây này với các cây khác vào thời gian thực hiện 0. Ngoài ra, mỗi nút được đánh dấu có hai đơn vị thời gian lưu trữ. Người ta có thể được sử dụng để cắt các nút từ cha của nó. Nếu điều này xảy ra, các nút sẽ trở thành một nút khóa và đơn vị thứ hai của thời gian sẽ vẫn lưu giữ trong nó như trong bất kỳ thư mục gốc khác.

### Các phép xử lý

Các phép xử lý trên Fibonacci heap bao gồm :

- ✚  $Q.initialize()$ : khởi tạo một hàng đợi rỗng  $Q$ .
- ✚  $Q.isEmpty()$ : trả về giá trị đúng.
- ✚  $Q.insert(e)$ : chèn  $e$  vào hàng đợi  $Q$  và trả về nút có chứa  $e$ .
- ✚  $Q.deleteMin()$ : trả về phần tử tối thiểu của  $Q$  và xóa nó.
- ✚  $Q.min()$ : trả về phần tử tối thiểu của  $Q$ .
- ✚  $Q.decreaseKey(v, k)$ : giảm giá trị của  $v$  thành giá trị  $k$ .

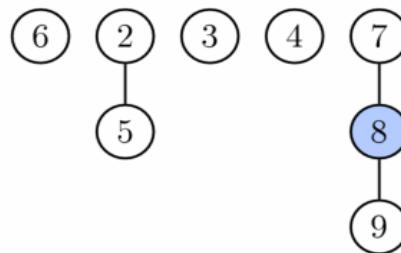
Vun đống Fibonacci cho phép xóa nhanh và nối các nút khóa của tất cả các cây được liên kết bằng cách sử dụng một vòng tròn, danh sách liên kết kép. Các nhánh con của mỗi nút cũng được liên kết bằng cách sử dụng một danh sách như vậy. Đối với

mỗi nút, ta duy trì số lượng các con và các nút được đánh dấu. Hơn nữa ta duy trì một con trỏ ở thư mục gốc.

Hoạt động tìm *min* bây giờ là đơn giản bởi vì ta giữ cho con trỏ đến nút chứa nó. Nó không thay đổi tiềm năng của heap, do đó cả chi phí thực tế được phân bổ là không đổi. Như đã đề cập ở trên, phép kết nối được thực hiện đơn giản bằng cách ghép danh sách các nút chính của hai đống. Điều này có thể được thực hiện trong thời gian liên tục và khả năng không thay đổi, dẫn đến độ phức tạp của bài toán thay đổi.

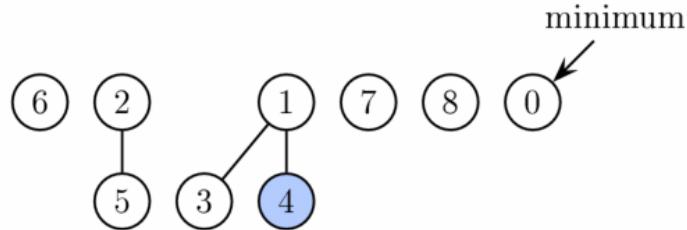
Hoạt động chèn bằng cách tạo ra một đống mới với một phần tử và làm hợp nhất. Điều này cần có thời gian liên tục, và tăng tiềm năng, bởi vì số lượng của cây tăng lên. Độ phức tạp của giải thuật là không đổi.

Hoạt động xử lý nút cơ sở (tương tự như xóa) hoạt động trong ba giai đoạn. Trước tiên, ta tới tận nút có chứa nút con đó và loại bỏ nó. Con của nó sẽ trở thành nút cha của những cây mới. Nếu có số nút con là  $d$ , phải mất thời gian  $O(d)$  để xử lý tất cả các rẽ mới và tăng tiềm năng bằng cách  $d - 1$ . Do đó, phân bổ thời gian hoạt động của giai đoạn này là  $O(d)=O(\log n)$ . Ở giai đoạn đầu của hoạt động xử lý ví dụ trên sẽ được thể hiện như sau:

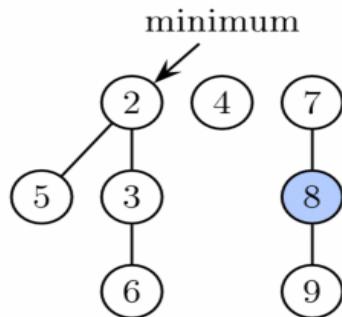


Tuy nhiên để hoàn thiện các hoạt động, ta cần phải cập nhật con trỏ vào thư mục gốc với chính nút cơ sở. Có thể lên đến  $n$  nút ta cần phải kiểm tra. Trong giai đoạn hai, ta giảm số lượng nút của cây và gắn các nút kết với nhau sao cho cùng một mức độ. Khi hai nhân  $u$  và  $v$  có cùng một mức độ, ta thực hiện một trong số họ là con của cây khác để một với phím nhỏ hơn vẫn là nút khóa. Mức độ của nó sẽ tăng một. Điều này được lặp đi lặp lại cho đến khi mọi nút khóa có một mức độ khác nhau. Để tìm cây của cùng một mức độ hiệu quả, ta sử dụng một mảng chiều dài  $O(\log n)$ , trong đó, ta giữ một con trỏ đến một thư mục gốc của mỗi mức độ. Khi một nút thứ hai được tìm thấy trong cùng một mức độ thì liên kết và mảng được cập nhật. Thời gian chạy thực tế là  $O(\log n + m)$  trong đó  $m$  là số khóa vào đầu giai đoạn thứ hai. Cuối cùng, ta sẽ có ít nhất  $O(\log n)$  nút khóa. Do đó sự khác biệt trong chức năng tiềm năng từ trước khi giai đoạn này để sau đó là:  $O(\log n) \cdot m$ , độ phức tạp của bài toán nhiều nhất là  $O(\log n + m) + c(O(\log n) \cdot m)$ . Với một sự lựa chọn đủ lớn của  $c$ , điều này giúp đơn giản hóa đến  $O(\log n)$ .

Ta sẽ thấy, ví dụ trên sau khi giảm nút khóa chính của nút 9-0. Nút này cũng như hai nút khóa chính được đánh dấu và cắt từ một cây có bắt nguồn từ 1 và đặt như nút khóa mới.



Trong giai đoạn thứ ba, ta kiểm tra từng nút khóa chính còn lại. Giai đoạn này thực hiện trong thời gian  $O(\log n)$  và khả năng không thay đổi.



Đây là kết quả đạt được sau khi hoàn thiện xử lý các nút cơ sở.

Hoạt động giảm khóa sẽ mất đi các nút, và nếu khóa mới là nhỏ hơn so với cha của nó (cha không phải là một gốc) nó được đánh dấu. Nếu nó đã được đánh dấu, nó được cắt là tốt nhất. Ta tiếp tục trở lên cho đến khi đạt được một trong hai nút chính hoặc một nút chính được đánh dấu. Mỗi cây mới, ngoại trừ cây đầu tiên được đánh dấu ban đầu như vậy một nút khóa chính sẽ trở nên không rõ ràng. Do đó, số lượng các nút được đánh dấu thay đổi bởi  $-(k-1)+1=-k+2$ . Kết hợp 2 thay đổi, những thay đổi tiềm năng bằng  $2(-k+2)+k=-k+4$ . Thời gian thực tế để thực hiện việc cắt là  $O(k)$ , do đó với một sự lựa chọn đủ lớn  $c$  thời gian thực hiện giải thuật là không đổi.

Hoạt động xóa có thể được thực hiện đơn giản bằng cách giảm giá trị của các nút khóa đó, do đó biến nó thành nút cơ sở của toàn bộ đồng. Thời gian xử lý bài toán của hoạt động này là  $O(\log n)$ .

Các phép xử lý khác trong Fibonacci heap gồm:

$Q.delete(v)$ : xóa  $v$  từ  $Q$  mà không thực hiện tìm kiếm  $v$

$Q.meld(Q')$ : kết hợp  $Q$  và  $Q'$ .

$Q.search(k)$ : tìm kiếm các phần tử  $k$  quan trọng trong  $Q$ .

### Các phép biến đổi trong Fibonacci heap

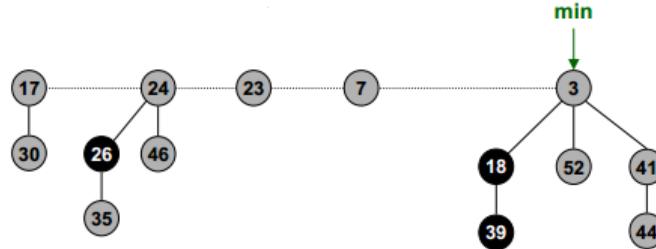
#### a. Phép chèn (Insert)

Phép chèn trong cây Fibonacci được thực hiện như sau:

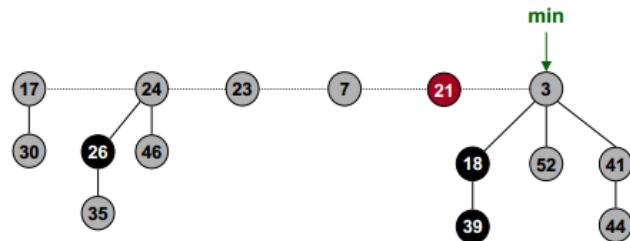
- Tạo ra một cây đơn mới.

- Thêm vào bên trái của con trỏ nhỏ nhất.
- Cập nhật lại con trỏ.

Ví dụ : Chèn phần tử 21 vào cây sau :



Thực hiện phép chèn như đã nêu trên ta được cây mới như sau :

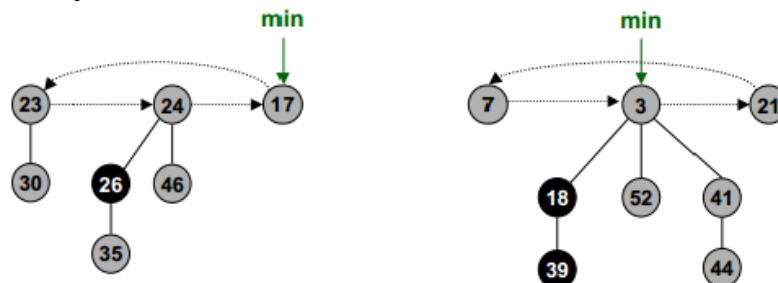


### b. Phép kết hợp(Union)

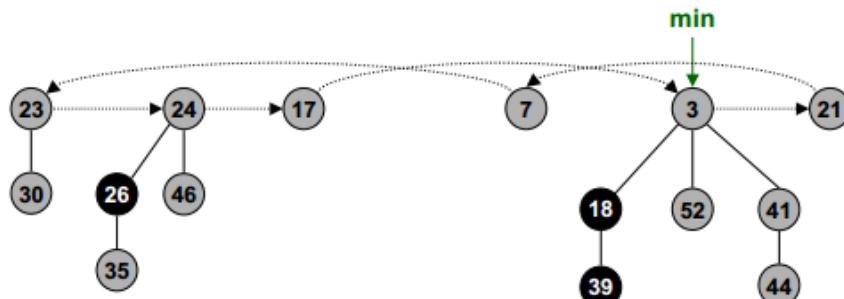
Phép kết hợp dùng để nối hai đống Fibonacci.

Danh sách gốc trước khi nối là hình tròn, danh sách sau khi kết hợp là kép.

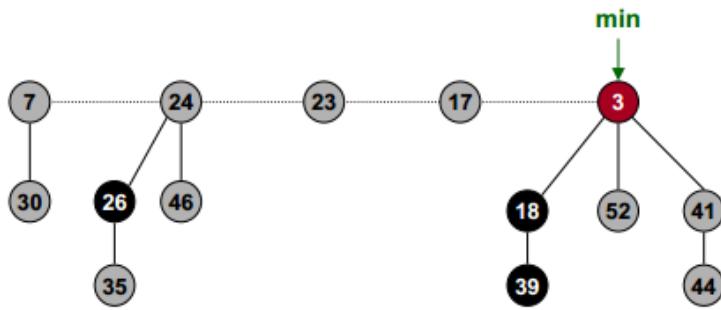
Ví dụ: kết hợp hai cây Fibonacci sau



Ta sẽ thực hiện như sau :



Ta được kết quả như sau :



### c. Phép xóa (Delete)

#### Delete min

- Phép delete min thực hiện xóa phần tử nhỏ nhất và nối các con của nó vào cây gốc.
- Củng cố lại cây để cây không tồn tại hai rễ có cùng một cấp độ.

```

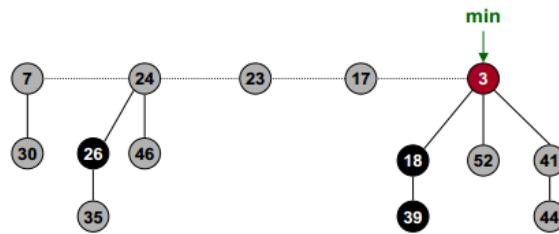
 $A = Q.\text{min}();$ 
if  $Q.\text{size}() > 0$ 
    then  $\text{remove}Q.\text{min}()$  from  $Q.\text{rootlist};$ 
        add  $Q.\text{min}.\text{childlist}$  to  $Q.\text{rootlist};$ 
     $Q.\text{consolidate}();$ 
return  $A;$ 
```

Với A là một mảng có độ dài  $2\log n$  ứng với nút trong đống Fibonacci

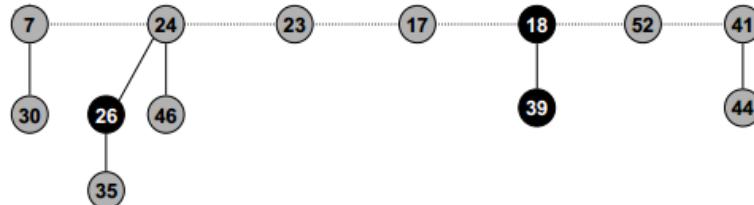
```

for  $i = 0$  to  $2 \log n$  do  $A[i] = \text{null};$ 
    while  $Q.\text{rootlist} \neq \emptyset$  do
         $B = Q.\text{delete-first}();$ 
        while  $A[\text{rank}(B)]$  is not null do
             $B' = A[\text{rank}(B)];$ 
             $A[\text{rank}(B)] = \text{null};$ 
             $B = \text{link}(B, B');$ 
    end while
     $A[\text{rang}(B)] = B;$ 
end while
determine  $Q.\text{min};$ 
```

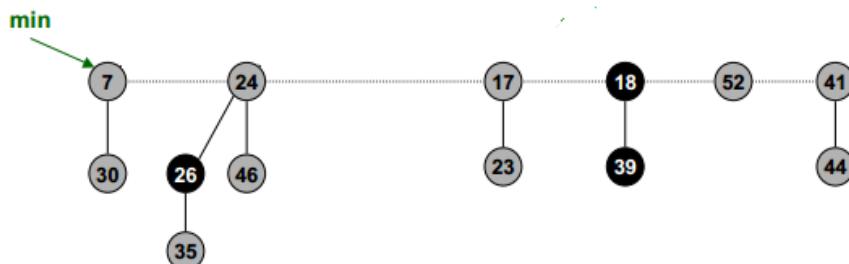
Ví dụ :



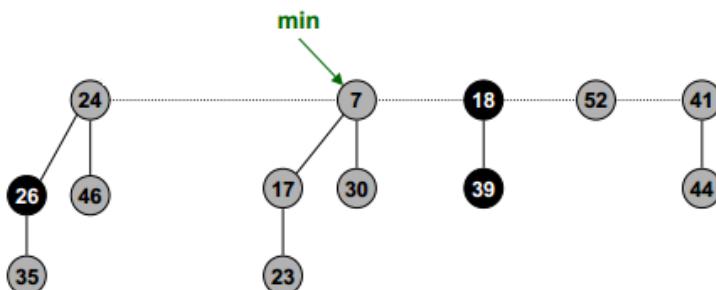
Sau khi xóa phần tử *min*, cây được cập nhật như sau:



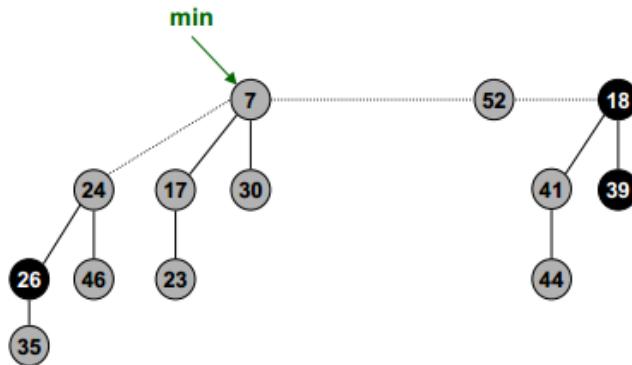
Xét thấy 23 và 17 cùng ở độ sâu 0 nên ghép 23 và 17 thành 1 cây như sau:



Tương tự, 7 và 17 cùng ở độ sâu 0 nên ghép chúng thành 1 cây.



Cứ như vậy ta sẽ được cây hoàn chỉnh sau khi xét tất cả các nút như sau :



d. Giảm giá trị khóa (Decrease key).

Khi thực hiện giảm khóa của một yếu tố nào đó từ giá trị  $u$  xuống giá trị  $v$  ta cần xét các trường hợp sau:

- Trường hợp 1:  $min$  của đống không vi phạm.
  - Giảm khóa từ giá trị  $u$  xuống giá trị  $v$ .
  - Thay đổi con trỏ  $min$  của đống nếu cần thiết.
- Trường hợp 2 : nút cha của  $u$  không được đánh dấu.
  - Giảm khóa từ giá trị  $u$  xuống giá trị  $v$ .
  - Cắt đứt mối quan hệ của  $u$  và cha của nó.
  - Đánh dấu cha của  $u$ .
  - Thêm cây có gốc là  $u$  vào danh sách đống, cập nhật giá trị  $min$  của đống.
- Trường hợp 3 : nút cha của  $u$  được đánh dấu.
  - Giảm khóa từ giá trị  $u$  xuống giá trị  $v$ .
  - Cắt đứt mối quan hệ giữa  $u$  và cha  $p$  của nó, thêm  $u$  vào danh sách gốc.
  - Cắt đứt mối quan hệ giữa  $p$  và nút cha của  $p$  để nhỏ tận gốc.  
Nếu cha của  $p$  không được đánh dấu : làm theo trường hợp 2.  
Nếu cha của  $p$  được đánh dấu : làm như trường hợp 3 và cứ lặp lại như vậy.

Đoạn mã lệnh được thể hiện như sau

```

if  $k > v.key$  then return
   $v.key = k;$ 
  update  $Q.min$ ;
    if  $v \in Q.rootlist$  or  $k \geq v.parent.key$ 
    then return;
do
  parent =  $v.parent$ ;
   $Q.cut(v)$ ;
   $v = parent$ ;

```

```

while  $v.mark$  and  $v \notin Q.rootlist$ 
if  $v \notin Q.rootlist$  then  $v.mark = true;$ 

```

Mã lệnh  $Q.cut(v)$

```

if  $v \notin Q.rootlist$ 
then  $rank(v.parent) = rank(v.parent) - 1;$ 
 $v.parent = null;$ 
remove  $v$  from  $v.parent.childlist$ 
add  $v$  to  $Q.rootlist$ 

```

### **Phạm vi sử dụng của Fibonacci heap**

Việc xử lý bài toán của một đống Fibonacci phụ thuộc vào mức độ (số lượng con) của bất kỳ gốc cây con  $O(\log n)$ , trong đó  $n$  là kích thước của heap. Ở đây kích thước bắt nguồn từ cây tại bất kỳ nút  $x$  của mức độ  $d$  trong đống phải có kích thước ít nhất là  $F_{d+2}$ , trong đó  $F_k$  là thứ  $k$  số Fibonacci. Mức độ ràng buộc sau từ này và thực tế mà  $F_{d+2} \geq \varphi_d$  cho tất cả các số nguyên  $d \geq 0$ ,  $\varphi = (1 + \sqrt{5})/2 = 1,618$ .

Xét bất kỳ nút  $x$  ở trong đống ( $x$  cần không phải là gốc của một trong những cây chính). Xác định kích thước  $x$  là kích thước của cây có gốc là  $x$  (số lượng con của  $x$ ). Chúng ta chứng minh bằng cảm ứng trên chiều cao của  $x$  (chiều dài của một con đường đơn giản nhất từ  $x$  đến một lá hậu duệ), kích thước  $x \geq F_{d+2}$ , trong đó  $d$  là độ  $x$ .

Trường hợp cơ sở: Nếu  $x$  có chiều cao 0,  $d = 0$  và kích thước  $x = 1 = F_2$ .

Trường hợp quy nạp: Giả sử  $x$  có chiều cao và mức độ tích cực  $d > 0$ . Cho  $y_1, y_2, \dots$  là con của  $x$ , lập chỉ mục theo thứ tự thời gian họ gần đây nhất được thực hiện con của  $x$  ( $y_1$  là sớm nhất và  $y_d$  là muộn nhất) và để cho  $c_1, c_2, \dots$  là độ tương ứng. Chúng tôi cho rằng  $c_i \geq i - 2$  (với  $2 \leq i \leq d$ ). Kể từ đỉnh cao của tất cả các  $y_i$  ít hơn so với  $x$ , chúng ta có thể áp dụng giả thuyết quy nạp cho họ để có được kích thước  $y_i \geq F_{c_i+2} \geq F_{(i-2)+2} = F_i$ . Các nút  $x$  và  $y_1$  đóng góp ít nhất 1 kích thước  $x$ , và vì vậy chúng ta có :

$$size(x_i) \geq 2 + \sum_{i=2}^d size(y_i) \geq 2 + \sum_{i=2}^d F_i = 1 + \sum_{i=2}^d F_i$$

Ta cũng chứng minh được rằng  $1 + \sum_{i=0}^d F_i = F_{d+2}$  cho bất kỳ  $d \geq 0$  mang đến cho các mong muốn thấp hơn ràng buộc về kích thước  $x$ .

