木に対する一般的なテク達

@okuraofvegetabl

目次

第 I 章

はじめに

こんにちは、69 回生の okuraofvegetable です。競技プログラミングが大好きで、普段は競プロの問題ばかり解いています。npca の部誌は昨年に引き続き 2 回目です。昨年の部誌では、グラフの基本的な知識や問題について書きました。私はグラフ理論の問題が好きです。見かけはグラフに見えなくてもグラフの問題に落としこめる問題などもとても好きです。今回はグラフの中でも、更に的を絞って"木"に対する問題のテクニックなどを紹介したいと思います。中でも、木に対するたくさんのクエリに高速に答える問題において威力を発揮するであろうテクニックを紹介します。木の根や形が変わるような動的木については扱いません。そちらについて詳しく知りたい方は 2 年前の部誌の catupper さんの記事に詳しく載っていますのでそちらを参照してください。昨年の部誌と同様に、文章中のソースコードではすべて C++ を用いています。また、ソース中の整数型でほとんど int 型を用いていますが、実際に問題を解く際は制約に注意して適宜 long long int 型などを用いてください。ソースコードをコピペして使って頂いても構いませんが、何があっても自己責任でお願いいたします。

第 II 章

知識編

1　木の用語

グラフの基本的な用語や知識 (辺、頂点などの用語、DFS などのアルゴリズム) については割愛させていただきます。もしわからない場合は調べるか、去年の部誌の私の記事を参照してください。

木

木というのは下の図のような特殊なグラフのことで N 個の頂点と $N-1$ 本の辺がある連結なグラフのことです。実はこの条件だけで、木には閉路が含まれないことがわかります。背理法で証明してみましょう。$M(0 < M < N)$ 頂点の閉路があると仮定します。閉路の中には M 本の辺が含まれています。残り $N-M$ 頂点と閉路を連結にするためには最低 $N-M$ 本の辺が必要です。閉路に含まれる辺とあわせて最低 N 本の辺が必要です。これは辺が $N-1$ 本であることに矛盾します。よって木の中に閉路は存在しません。木においては任意の頂点から他の頂点までの行き方 (パス) はただひと通りに定まります (もちろん往復など無駄な動きをしない場合)。図を見ればなぜこれが木と呼ばれるのかわかると思います。この記事のソースコード中で、木は普通の無向グラフと同様に std::vector を用いた隣接リストで表現します。

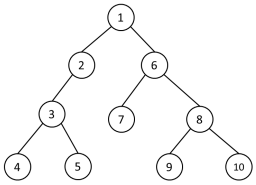

図 II.1　木

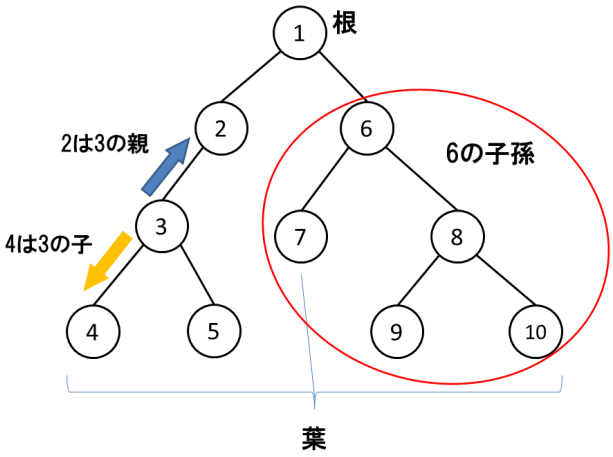

図 II.2　根、葉、子孫、先祖

根、葉

ある頂点を図の一番上に固定して考えると色々と便利なことがあります。その際、一番上に固定した頂点を根と呼びます。また、根を固定した木を根付き木と呼びます。また、子のいない頂点を葉と呼びます。

深さ

頂点 v の深さは、根から v までの辺の本数です。木の深さは全ての頂点の深さの最大値です。

子、親、子孫、先祖

根付き木において、根以外のある頂点 v に直接繋がっている頂点のうち根に近い頂点を v の親、それ以外を v の子と呼びます。根以外の任意の頂点はただ一つの親を持ちます。そして、頂点 v の子、子の子、子の子の子、…をすべてあわせて v の子孫と呼びます。[1]。同様に、頂点 v の親、親の親、親の親の親、…をすべてあわせて v の先祖と呼びます 各頂点の親や深さは dfs することで $O(N)$ で求まります。以下のソースでは、木を辺の集合として入力し DFS で頂点 0 を根とした時の各頂点の深さと親の頂点番号を求めています。[2]

[1] 図では v 自身も v の子孫に含んでいます。以後問題文中などでも v の子孫といった場合、v 自身も含める事にします

[2] 頂点番号は $0 \sim N-1$ と仮定します。

```cpp
#include <cstdio>
#include <vector>
using namespace std;
#define MAX_N 100100
#define pb push_back
vector<int> g[MAX_N];//tree
int N,root=0,parent[MAX_N],depth[MAX_N];
void dfs(int v,int p,int d)
//v...current vertex , p...parent of v ,d...depth of v
{
    depth[v]=d;
    for(int i=0;i<g[v].size();i++)
    {
        if(g[v][i]==p)continue;
        dfs(g[v][i],v,d+1);
    }
}
void add_edge(int u,int v)
{
    g[u].pb(v);
    g[v].pb(u);
}
int main()
{
    scanf("%d",&N);//number of vertexes
    for(int i=0;i<N-1;i++)// edges
    {
        int u,v;
        scanf("%d %d",&u,&v);
        add_edge(u,v);
    }
    dfs(root,-1,0);
}
```

2　木の問題について

そもそも木に対するクエリに高速に答える問題というのはどのようなものなのでしょうか?

クエリとは

クエリというのは言わば質問のようなもので、「〜な頂点は何個ありますか?」や「〜と…の間の距離はいくらですか?」のようなものが多いです。

問題の難しさ

実際に競技プログラミングのコンテストで出題されるような問題では、木の頂点数が 10^5 個、クエリの数が 10^5 個程度のものが多いです。あらかじめ、任意のクエリに対する解を前計算し、クエリを読み込むたびに $O(1)$ で答えられる問題もありますが、そうでない場合 Time Limit が 1,2 秒であることを考慮すると、1 クエリあたり 100〜500 回の計算で答えを出さなくてはなりません。頂点数に対して圧倒的に小さい回数の計算で答えを出すためには様々な前計算やテクニックが必要なのです。先に言ってしまうと、今後 $O(logN)$,$O((logN)^2)$[3]といった計算量がたくさん出てくると思いますが定数倍も考慮すると 1 クエリあたりこれくらいの計算量で解かなければなりません。

3　データ構造

一度、木から離れて考えてみます。少ない計算量で答えを出すためには効率良くデータを管理して効率良くデータの変更、読み出しを行わなくてはなりません。ここでは Segment Tree と BIT いうデータ構造を紹介します。これらは、数値の列に対するデータ構造です。木を扱わなければならないのに列のデータ構造なんて役に立つのか?と思われる方もいらっしゃるかもしれませんが後々紹介するテクを知ればなんでこれが木を扱うのに利用できるかわかると思います。

[3] 普通底の 2 は省略されます

Segment Tree

Segment Tree は応用範囲が広く、色々な機能を実現させられますがその中でも有名なのが RMQ(Range Minimum Query) です。RMQ では、N 要素の数列 A がある時に、

- 更新クエリ　　A_x の値を変更する.
- 最小値クエリ　区間 $[l, r)$ の最小値、すなわち $min\{A_i \mid l \leq i < r\}$ を求める.

という操作をそれぞれ $O(logN)$ で実現できます。$[l, r)$ は半開区間といい、$l, l+1..., r-1$ を表します RMQ を少し変更するだけで区間の max をとったり、区間の和をとったりもできます。Segment Tree は、下図に示すように完全二分木で、それぞれの頂点がある区間に対応する値を持っています。図のように各頂点を番号付けすると、葉以外の頂点 k の左の子は頂点 $2k+1$, 右の子は頂点 $2k+2$ となります。頂点 k から親にアクセスしたいときは頂点 $(k-1)/2$ を見ればいいです。[4]このように番号付けすると配列で管理でき、実装が楽です。[5]図を見る限り、2 の冪乗個の要素数の数列しか扱えないようですが、そうでないときは適宜クエリの答えに影響しないような値をつけ足して要素数を 2 の冪乗個にすればよいです。initialize は、N 回値の変更をすればよいです。[6]図から明らかなように数列の要素数が 2^N 個の時、Segment Tree の頂点数は $2^{N+1}-1$ 個になります。また、Segment Tree の深さは N です (根の深さを 0 とする)。すなわち要素数に対し、木の深さはおおよそ $logN$ なのです。RMQ では、次のページの図のようにいくつかの区間の最小値が計算されている

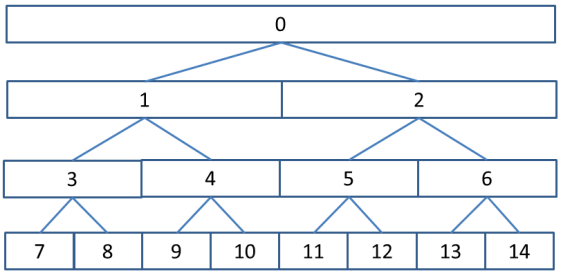

図 II.3　Segment Tree

状態を維持します。これらから、与えられた区間の最小値を区間全体を見ずに求めるので

[4] 整数型どうしの割り算では切り捨てのため
[5] これは個人の感想で、ポインタを用いてちゃんと木構造っぽく書くこともできます
[6] $O(NlogN)$ なので問題ない

す。それぞれのクエリについて順に見て行きましょう

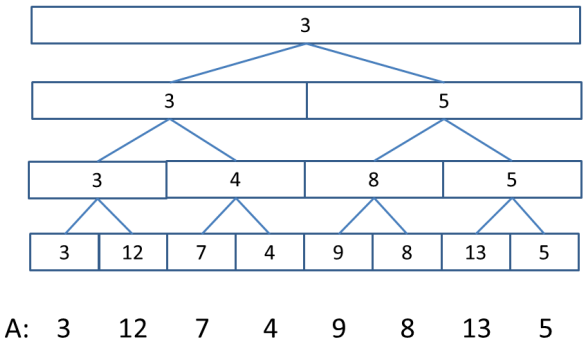

図 II.4　RMQ

更新クエリ

図のように更新したい頂点から親にたどっていき、更新していきます。数列の要素数を $SIZE$ とすると、数列の x 番目 (0-indexed) は Segment Tree の頂点 $x + SIZE - 1$ に対応します。対応する頂点の値を変更し、そこから親をたどっていきながら更新していけばよいです。計算量は Segment Tree の深さ分なので $O(logN)$ です。

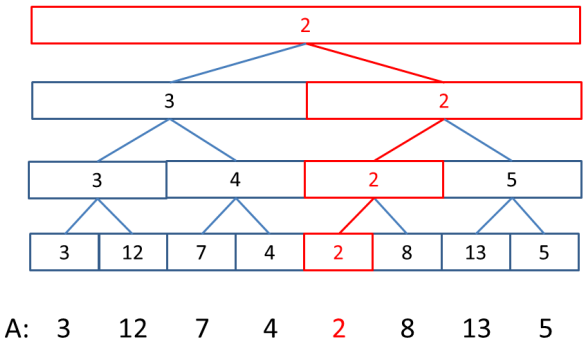

図 II.5　RMQ の更新

最小値クエリ

図のように根 (図で一番上の頂点、すなわち頂点 0) から順にたどって行きましょう。クエリが $[a, b)$, 今見ている頂点に対応する数列の区間を $[l, r)$ とすると、3 つの場合が考えられます。

- 2 つの区間が交わらない時、すなわち $r \leq a$ かつ $b \leq l$ の時
- $[l, r)$ が $[a, b)$ に完全に包含されている時
- それ以外の時

最初の場合は、答えに影響しない数 (今回の場合とても大きな数 INF) を返します。2 つ目の場合、答えが今見ている頂点の値より大きくなることはないのでそれを返します。最後の場合は、左の子と右の子について再帰的に計算してやって、それらの小さいほう (同じならその値) を返します。これで最小値クエリに答えることができます。さて、計算量は各深さにおいて見る頂点が定数個なのでこちらも $O(logN)$ です (詳しい評価は省きます)。最小

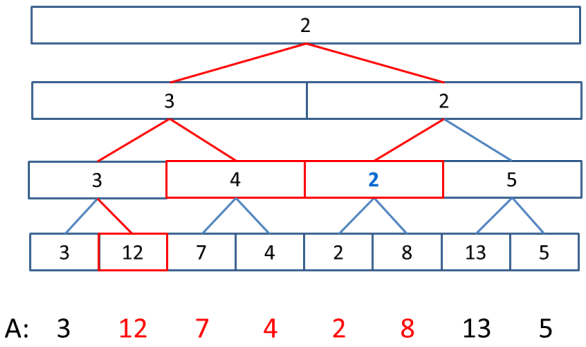

図 II.6　最小値クエリ

値クエリの実装では、今見ている頂点に対応する区間のインデックスも一緒に引数として渡してやると楽かもしれません。これで、基本的な Segment Tree が使えるようになりました。以下は RMQ の実装例です。

```cpp
#include <algorithm>
using namespace std;
#define INF 2000000000
const int SIZE = 1<<20;
struct RMQ
{
    int seg[SIZE*2];
    void update(int k,int x)
    //k...index of sequence
    //x...value
    {
        k += SIZE-1;
        seg[k]=x;
        while(k)
        {
            k = (k-1)/2;
            seg[k]=min(seg[k*2+1],seg[k*2+2]);
        }
    }
    int query(int a,int b,int k,int l,int r)
    //a,b...indexes of sequence corresponding query
    //k...index of Segment Tree
    //l,r...indexes of sequence corresponding k-th vertex of Segment Tree
    {
        if(r<=a||b<=l)return INF;
        else if(a<=l&&r<=b)return seg[k];
        else return min(query(a,b,k*2+1,l,(l+r)/2),query(a,b,k*2+2,(l+r)/2,r));
    }
};
```

ここで突然ですが次のような問題を考えて見ましょう。

要素数 N の数列 A がある。次の 2 種類の Q 個のクエリに答えよ

- $[l, r)$ に x を足す
- $min\{A_i \mid l \leq i < r\}$ を求める

$N \leq 10^5, Q \leq 10^5$

最小値クエリは前と同じですが更新クエリが前と少し異なっています。一見 $[l, r)$ に対して一つずつ x を足してやればいいように見えますが、これでは 1 回のクエリでの計算量が最悪 $O(NlogN)$ となり本末転倒です。実は、この問題も Segment Tree に少し工夫をするだけで各クエリ $O(logN)$ で解けてしまいます。次ではその具体的な方法について見ていくことにします。

遅延評価

さきほどの Segment Tree では予め特定の範囲の最小値が計算された状態を維持することで、少ない回数の計算で様々な区間の最小値を求めることができました。今回は、値が区間に対して足されるのでそれをいくつかの区間に分けてまとめて管理したいのです。なので、先ほどまで、Segment Tree の各頂点はそれぞれの頂点に対応する区間の最小値を持っていましたがさらに"その対応する区間に一様に足された値"を持たせてみます。(正確には、一様に足された値のうち、まだ反映されていない値です。) 以降、各頂点が持っている対応する区間の最小値を minimum, 区間に一様に足された値を lazy と言う事にします。上の図のようにさきほどと同様な初期状態の Segment Tree があるとします。それぞれのクエリにつ

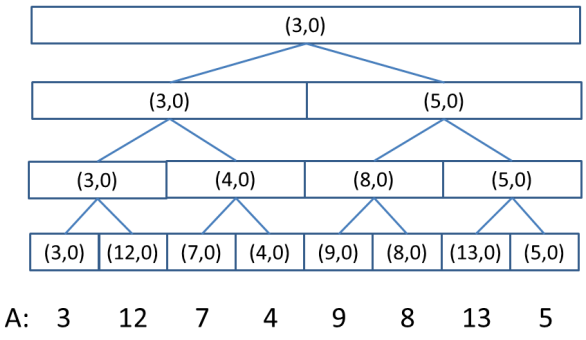

図 II.7　(minimum,lazy)

いて見ていきましょう。

更新クエリ

さきほどの RMQ の最小値クエリと同様の場合わけをします。1 つ目の場合はなにもせず終了。2 つ目の場合は今見ている頂点の lazy に値を加算。3 つ目の場合は**今見ている頂点の lazy を子に伝搬**し、して左右の子に対して再帰的に計算する。そして自分の子をもとに、自分の値を更新。これでうまく行きます。下図で、実際にどのような動きをしているのかを見ればなぜこれで良いのかわかると思います。先ほど太字で示したところが遅延評価と呼ばれるものです。

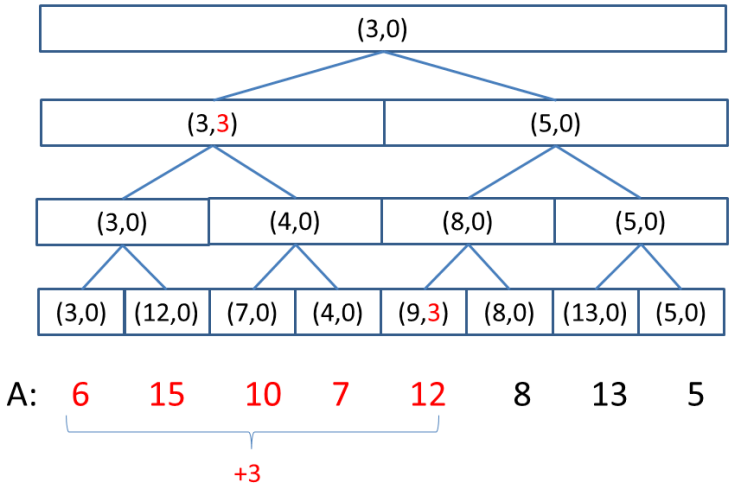

図 II.8　遅延評価 SegmentTree の更新 1

最小値クエリ

こちらは一つ前の Segment Tree とほとんど同じです。

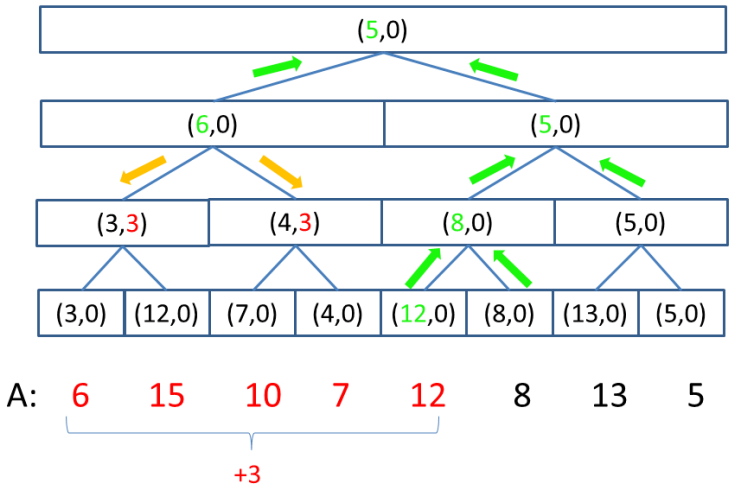

図 II.9　遅延評価 SegmentTree の更新 2

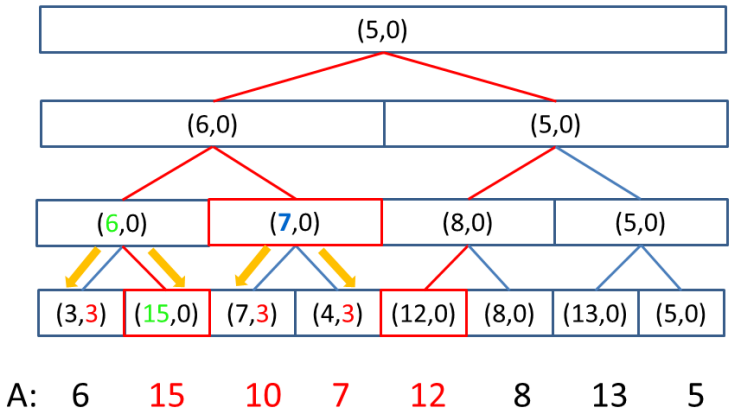

図 II.10　最小値クエリ

```
#include <algorithm>
using namespace std;
const int SIZE = 1<<18;
struct segtree
```

```
{
    int minimum[SIZE*2],lazy[SIZE*2];
    void lazy_evaluate(int k)
    {
        minimum[k]+=lazy[k];
        if(k<SIZE-1)// k isn't leaf of SegmentTree
        {
            lazy[2*k+1]+=lazy[k];
            lazy[2*k+2]+=lazy[k];
        }
        lazy[k]=0;
    }
    void update(int a,int b,int k,int l,int r,int x)
    {
        if(r<=a||b<=l)return;
        if(a<=l&&r<=b)
        {
            lazy[k]+=x;
            lazy_evaluate(k);
        }
        else
        {
            lazy_evaluate(k);
            update(a,b,k*2+1,l,(l+r)/2,x);
            update(a,b,k*2+2,(l+r)/2,r,x);
            minimum[k] = min(minimum[k*2+1],minimum[k*2+2]);
            return;
        }
    }
    int query(int a,int b,int k,int l,int r)
    {
        lazy_evaluate(k);
        if(r<=a||b<=l)return 0;
        if(a<=l&&r<=b)return minimum[k];
        else
        {
            int lch = query(a,b,k*2+1,l,(l+r)/2);
```

```
            int rch = query(a,b,k*2+2,(l+r)/2,r);
            return min(lch,rch);
        }
    }
};
```

Binary Indexed Tree

BIT(Binary Indexed Tree) は

- A_x の値を変更する
- $A_0 + A_1 + \cdots + A_x$ を求める

が $O(logN)$ でできるデータ構造です。おわかりの通り、Segment Tree で同じ機能を実現できます。なので細かいことは省いて実装だけ載せておきます。なぜこれでうまく動くのか知りたい方はぜひ調べてみてください。実装はとても軽いです。ただ、SegmentTree は 0-indexed でしたが BIT は 1-indexed なので注意してください (内部で 1 ずらしてやれば見かけ上 0-indexed として利用できます)。

```
#define MAX_N 100100
struct BIT
{
    int bit[MAX_N+1];
    void add(int i,int x)
    {
        while(i<=MAX_N)
        {
            bit[i]+=x;
            i+=i&-i;
        }
        return;
    }
    int sum(int i)
    {
        int res=0;
        while(i>0)
        {
```

```
                res+=bit[i];
                i-=i&-i;
            }
            return res;
        }
    };
```

いよいよ、木に対する様々なアプローチ、テクニックを紹介していこうと思います。

4　木に対するテクニック

LCA(Lowest Common Ancestor)

根付き木において、ある 2 頂点 u,v の両方の先祖である頂点のうち最も深さの大きい頂点を *LCA(Lowest Common Ancestor)* と言います。LCA を求めるためのアルゴリズムには

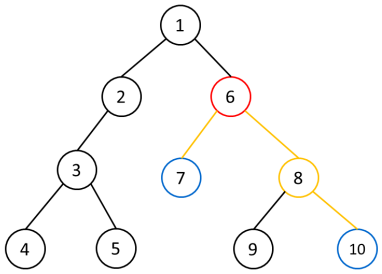

図 II.11　頂点 7 と頂点 10 の LCA は頂点 6

いろいろありますが、今回はそのうちの一つのダブリングを紹介します。ダブリングというのは、LCA 以外でも用いられる汎用的なアルゴリズムです。まず、前処理として次のようなテーブルを計算しておきます

$parent[v][i] :=$ 頂点 v から 2^i回親をたどった頂点 (なければ -1)

ここで $parent[v][i+1] = parent[parent[v][i]][i]$ より
$par[\sim][i]$ が計算できれば $par[\sim][i+1]$ が計算でき、i は log(木の深さ)[7]しか必要ないので時間計算量も空間計算量 (メモリ使用量) も $O(NlogN)$ です。このテーブルが求まれば、頂点 v から x 回親をたどった頂点というのは、x を 2 進数表記した時に i 番目のビットが立っていたら 2^i 回親をたどるということをして $O(logx)$ で求まります。これが求まれば後は簡単で、まず u,v の高さを揃えるために深いほうの頂点から $|depth[u] - depth[v]|$ 回親をたどります。$LCA(u,v)$ が u,v から x 回親をたどった頂点だとすると、u,v それぞれから x+1 回以上たどった頂点もすべて一致します。なので二分探索をすることで、その境目である LCA を求める事ができます。

[7] 一般の木の場合、木の深さは最悪 N です。

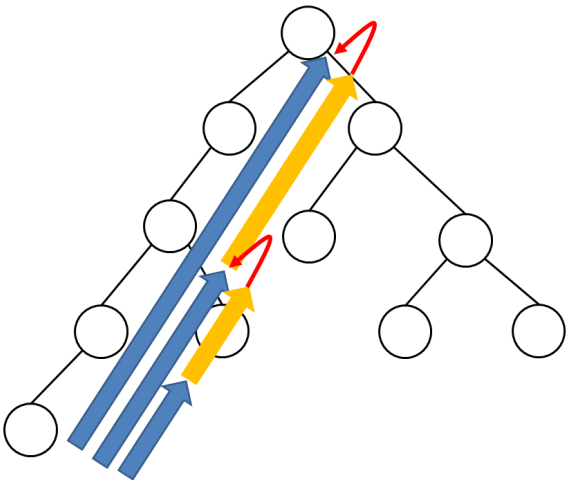

図 II.12　ダブリングのイメージ

```
#include <algorithm>
#include <vector>
using namespace std;
#define MAX_N 100100
#define MAX_LOG_N 20
int N,root;
vector<int> g[MAX_N];
int depth[MAX_N];
int par[MAX_N][MAX_LOG_N];
void dfs(int v,int p,int d)
{
    par[v][0]=p;
    depth[v]=d;
    for(int i=0;i<g[v].size();i++)
    {
        if(g[v][i]==p)continue;
        dfs(g[v][i],v,d+1);
    }
}
void fill_table()
{
    for(int i=0;i<19;i++)
```

```
    {
        for(int j=0;j<N;j++)
        {
            if(par[j][i]==-1)par[j][i+1]=-1;
            else par[j][i+1]=par[par[j][i]][i];
        }
    }
}
int lca(int u,int v)
{
    if(depth[u]>depth[v])swap(u,v);
    for(int i=19;i>=0;i--)
    {
        if(((depth[v]-depth[u])>>i)&1)v=par[v][i];
    }
    if(u==v)return u;
    for(int i=19;i>=0;i--)
    {
        if(par[u][i]!=par[v][i])
        {
            u = par[u][i];
            v = par[v][i];
        }
    }
    return par[u][0];
}
int main()
{
    dfs(root,-1,0);
    fill_table();
    return 0;
}
```

Euler Tour

木を扱うにあたって最重要といっても過言ではないテクニックがオイラーツアー (*Euler Tour*) です。オイラーツアーというのは, 図のように木を根から DFS する時に通る頂点を一直線に並べたものです。 各頂点番号が最初に登場するインデックスと最後に登場するイ

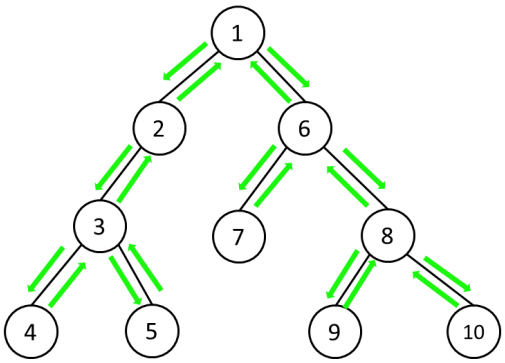

図 II.13　オイラーツアー

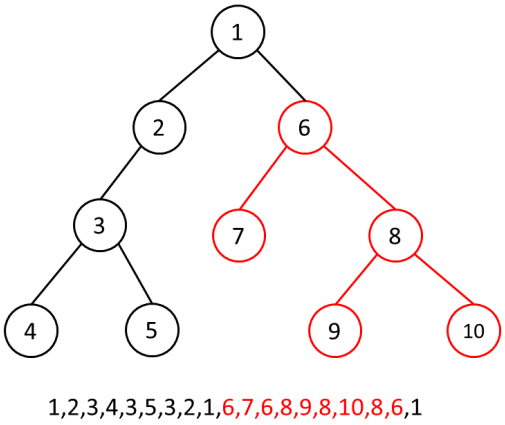

図 II.14　部分木が列に対応

ンデックスの間の区間はちょうど元の木の v の部分木に対応します。ついに木を列として扱うことできるようになりました。オイラーツアーに対して前述したデータ構造が使えるのです。例えば頂点が値を持っていて、

- 頂点 v 以下全ての頂点の値に x 加算する

- ある頂点の値を求める

というクエリがたくさんくるという問題なら、オイラーツアーを考えることで列の問題に変換でき、これは遅延評価 SegmentTree を用いて簡単に実現することができます。[8]オイラーツアーの実装は非常に簡単で、DFS で通る順にならべるので、実際に DFS するだけで O(N) でできます。オイラーツアーの要素数は、各辺を 2 回ずつ通るので植木算的に $2N-1$ だとわかります。

以下の実装で $begin[v], end[v]$ はそれぞれ v がオイラーツアー上で最初と最後に現れるインデックスです。(正確には $end[v]$ は最後に現れるインデックスの次) すなわち、$[begin[v], end[v])$ が v を根とする部分木に対応しているということです。

[8] 実は累積和を考えると遅延評価 SegmentTree ではなく BIT で解けます。基本的に、区間に同じ値を足す、区間の和を求める、というタイプならば BIT を用いればよいので遅延評価は必要ないです。

```cpp
#include <vector>
using namespace std;
#define pb push_back
#define MAX_N 100100
vector<int> g[MAX_N];
vector<int> euler_tour;
int begin[2*MAX_N],end[2*MAX_N];
int k=0,root=0;
void dfs(int v,int p)
{
    begin[v]=k;
    euler_tour.pb(v);
    k++;
    for(int i=0;i<g[v].size();i++)
    {
        if(g[v][i]!=p)
        {
            dfs(g[v][i],v);
            euler_tour.pb(v);
            k++;
        }
    }
    end[v]=k;
}
int main()
{
    dfs(root,-1);
}
```

パスに対するクエリ

オイラーツアーは部分木だけでなくパスに対するクエリもうまくデータをもつことで処理できます。木の辺にコストがついている時に、

- 辺のコストを変える
- u から v へのパスのコストを求める

という 2 つのクエリに答えたいとします。u, v 間のパスは、$u, lca(u, v)$ 間のパスと $lca(u, v), v$ 間のパスに分割できます。$lca(u, v)$ は u, v 両方の先祖なので、深さが単調に増加するパスのコストさえ高速に求められればこの問題を解くことができます。なので、ここでは u は v の先祖であると仮定します。まず図のオイラーツアーを眺めて見ると、部

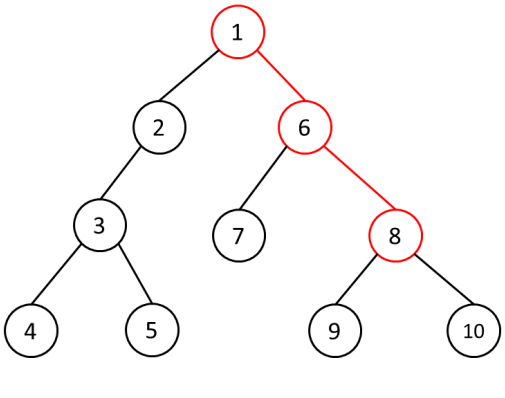

図 II.15　無駄な往復

分木の無駄な往復が含まれているので SegmentTree(または BIT) 上で $[begin[u], begin[v]]$ の和を求めてもパスのコストはわかりません。しかし、同じ辺でも根から離れる方向に移動するときはコスト正、戻る時は負にしてやると、部分木のコストは打ち消されてパスのコストが求まります。このようにコストに符号をつけて相殺するテクを知っていれば対応できる問題が増えると思います。

第 III 章

実践編

ここからは実践編と題して、問題の解説を書きたいと思います (あまり詳しい解説ではないのでわからなかった方は、各問題の公式の Editorial を参照してください)。今回は以下の問題を扱います。せっかくなら自分で解きたいという方は是非チャレンジしてみてください。おまけ以外はほぼこの記事で扱ったテクニックで解けます。各問題の詳しい概要、制約は下のリンクからご確認ください。

- Propagating tree (Codeforces Round #225 Div1 C)
  http://codeforces.com/contest/383/problem/C
- On Changing Tree (Codeforces Round #232 Div1 C)
  http://codeforces.com/contest/396/problem/C
- Running Away from the Barn (USACO 2012 December Gold)
  http://www.usaco.org/index.php?page=viewproblem2&cpid=213
- A and B and Lecture Rooms (Codeforces Round #294 Div2 E)
  http://codeforces.com/problemset/problem/519/E
- Xenia and Tree (Codeforces Round #199 Div2 E)(おまけ)
  http://codeforces.com/contest/342/problem/E

1 Propagating tree (Codeforces Round #225 Div1 C)

問題概要

各頂点が値をもった N 頂点の根付き木が与えられる。

- 頂点 x に val を加算し、x の子に $-val$ を加算、x の子の子に val を加算…
- 頂点 x の値を求める

という M 個のクエリに答えよ

加算のクエリが少し変ですね。+-交互にしないといけないので、一気にまとめて扱うのは難しそうです。ですが、x の子孫で、x からの距離が偶数のものには全て val 加算, 奇数のものには $-val$ 加算です。この性質を用いてうまくまとめて処理することを考えてみます。オイラーツアーを考えてみると、オイラーツアーで隣あう頂点の深さの差は 1 なので、深さの偶奇が同じ頂点はひとつ飛ばしに現れます。そこで、Segment Tree を 2 本用意して偶奇に分けて考えてみると、それぞれの x 以下の部分木に対応する区間にそれぞれ val,$-val$ を加算すればよく、値を知りたいときは適切なほうの SegmentTree の値を見ればわかります。なので遅延評価 SegmentTree を 2 本用意して、偶奇にわけて処理してやるとうまくいきます[1]。計算量は各クエリ $O(logN)$ なので全体で $O(MlogN)$ です

ソースコード中のオイラーツアーを生成する関数 dfs は省略します。

```
#include <cstdio>
#include <vector>
using namespace std;
#define pb push_back
const int SIZE=1<<19;
const int MAX_N = 200100;
int n,m,k=0;
int a[MAX_N];
vector<int> g[MAX_N];
int begin[MAX_N*2],end[MAX_N*2];
struct segtree
{
    int sum[SIZE*2],lazy[SIZE*2];
    void lazy_evaluate(int k,int l,int r)
```

[1] もちろん先ほどと同様に累積和をかんがえることで BIT で解けます。

```
{
    sum[k]+=lazy[k]*(r-l);
    if(k<SIZE-1)
    {
        lazy[2*k+1]+=lazy[k];
        lazy[2*k+2]+=lazy[k];
    }
    lazy[k]=0;
}
void update(int a,int b,int k,int l,int r,int x)
{
    if(r<=a||b<=l)return;
    if(a<=l&&r<=b)
    {
        lazy[k]+=x;
        lazy_evaluate(k,l,r);
    }
    else
    {
        lazy_evaluate(k,l,r);
        update(a,b,k*2+1,l,(l+r)/2,x);
        update(a,b,k*2+2,(l+r)/2,r,x);
        sum[k] = sum[k*2+1]+sum[k*2+2];
        return;
    }
}
int query(int a,int b,int k,int l,int r)
{
    lazy_evaluate(k,l,r);
    if(r<=a||b<=l)return 0;
    if(a<=l&&r<=b)return sum[k];
    else
    {
        int lch = query(a,b,k*2+1,l,(l+r)/2);
        int rch = query(a,b,k*2+2,(l+r)/2,r);
        return lch+rch;
    }
```

```
        }
        void update(int a,int b,int x){update(a,b,0,0,SIZE,x);}
        int query(int a,int b){return query(a,b,0,0,SIZE);}
    };
    segtree odd,even;
    int main()
    {
        scanf("%d %d",&n,&m);
        for(int i=0;i<n;i++)scanf("%d",&a[i]);
        for(int i=0;i<n-1;i++)
        {
            int u,v;
            scanf("%d %d",&u,&v);
            u--;v--;
            g[u].pb(v);
            g[v].pb(u);
        }
        dfs(0,-1);
        for(int i=0;i<n;i++)
        {
            if(begin[i]%2==0)even.update(begin[i],begin[i]+1,a[i]);
            else odd.update(begin[i],begin[i]+1,a[i]);
        }
        for(int i=0;i<m;i++)
        {
            int type;
            scanf("%d",&type);
            if(type==1)
            {
                int x,val;
                scanf("%d %d",&x,&val);
                x--;
                if(begin[x]%2==0)
                {
                    even.update(begin[x],end[x],val);
                    odd.update(begin[x],end[x],-val);
                }
```

```
            else
            {
                odd.update(begin[x],end[x],val);
                even.update(begin[x],end[x],-val);
            }
        }
        else
        {
            int x;
            scanf("%d",&x);
            x--;
            if(begin[x]%2==0)printf("%d\n",even.query(begin[x],begin[x]+1));
            else printf("%d\n",odd.query(begin[x],begin[x]+1));
        }
    }
    return 0;
}
```

2 On Changing Tree (Codeforces Round #232 Div1 C)

問題概要

各頂点が値をもった N 頂点の根付き木が与えられる。

- 頂点 v に x を加算し、v の子に $x-k$ を加算、v の子の子に $x-2k$ を加算…
- 頂点 x の値を 10^9+7 で割った余りを求める

という Q 個のクエリに答えよ

この問題も加算のクエリが少し変なタイプの問題です。この問題では深さの違う頂点に足される値は全て異なります。先ほどと同じアイデアで、深さごとにまとめて処理しようとしても木の深さは最悪頂点数と同じになるなので 1 つのクエリあたり $O(N)$ 以上かかってしまい間に合いません。足される値の性質に注目してみると深さごとに足される値が等差ですが、この性質をどう使えるのかはイマイチ見えてこないと思います。そこで足される値をもう少し正確に定式化してみましょう。v の子孫 u に足される値は $x-k*(depth[u]-depth[v])$ です。これを展開してみると $x+k*depth[v]-k*depth[u]$ となります。そこで 2 つに分けて、$x+k*depth[v]$ を v 以下の頂点すべてに足し、足しすぎた $k*depth[u]$ を引くことを考えてみます。$depth[u]$ は定数なので足し過ぎた k の合計値がわかれば足し過ぎた値を求められます。なので加算クエリが来た時に v の子孫 u について $k(=$ 足し過ぎた値$/depth[u])$ を加算をしたいということになります。これは、オイラーツアーの区間に値を足すだけですね。

まとめると、今回も 2 つの SegmenTree(または BIT) を 2 本用意して、片方には頂点 v の部分木に対応する区間に $x+k*depth[v]$ を、もう一方には k を加算することで加算クエリを $O(logN)$ で処理できます。値を求める時は、上記の計算をするだけなのでこちらも $O(logN)$ でできます。

BIT のほうが定数倍が軽いので、BIT で解けるときは BIT を使うことをおすすめします。$SegmentTree$ と dfs は省略しています。

```
#include <cstdio>
#include <vector>
using namespace std;
typedef long long ll;
#define pb push_back
#define MOD 1000000007
const int SIZE=1<<20;
const int MAX_N = 300100;
```

```
int n,q,K=0;
vector<int> g[MAX_N];
int depth[MAX_N];
int begin[MAX_N*2],end[MAX_N*2];
segtree sum,over;
int main()
{
    scanf("%d",&n);
    for(int i=1;i<n;i++)
    {
        int v;
        scanf("%d",&v);
        v--;
        g[i].pb(v);
        g[v].pb(i);
    }
    dfs(0,-1,0);
    scanf("%d",&q);
    for(int i=0;i<q;i++)
    {
        int type;
        scanf("%d",&type);
        if(type==1)
        {
            int v,x,k;
            scanf("%d %d %d",&v,&x,&k);
            v--;
            sum.update(begin[v],end[v],((ll)x+(ll)k*depth[v])%MOD);
            over.update(begin[v],end[v],k);
        }
        else
        {
            int v;
            scanf("%d",&v);
            v--;
            ll ans = sum.query(begin[v],begin[v]+1)
                     -(ll)depth[v]*over.query(begin[v],begin[v]+1);
```

```
                ans = ((ans%MOD)+MOD)%MOD;
                printf("%d\n",(int)ans);
            }
        }
        return 0;
}
```

3 Running Away from the Barn (USACO 2012 December Gold)

問題概要

各辺に正のコストがある N 頂点の根付き木が与えられる。各頂点について、子孫のうち距離 L 以下の頂点の数を求めよ。

これはクエリ形式の問題ではないですが N 頂点全てに対する答えを求めないといけないので、それぞれ独立に計算するとすれば一頂点あたり $O(logN)$ 以下で求めなければ間にあいません。

まず、各頂点について子孫のうち L 以下になるものを探すのは大変そうなので自分を距離 L 以下の子孫とする頂点たちのカウンタを増やす、という方向で考えていきましょう。頂点 v を子孫とするような頂点は根から頂点 v のパス上の頂点のみです。また、辺のコストは正なのであるところを境に、L 以下かそうでないかがわかれます。なので、パス上を二分探索することで $O(logN)$ で境目をみつけることができます (DFS で各頂点の根からの距離を求めておけば、距離は O(1) で求められる)。あとはパス上に 1 を足すだけです。ある頂点 x の値をオイラーツアーの x の部分木にあたる列の和に対応させると、$u, v(depth[u] < depth[v])$ 間に 1 を足したい時、BIT の $begin[u]$ に 1 を、$begin[v]+1$ から 1 を引くことで実現できます。全体で計算量は $O(N(logN)^2)$ になります。[2]

```
#include <cstdio>
#include <vector>
using namespace std;
typedef long long ll;
#define MAX_N 200100
#define pb push_back
BIT bit;
```

[2] 実装を工夫すれば $O(NlogN)$ にできます

```
struct edge
{
    int to;
    ll cost;
    edge(int to,ll cost):to(to),cost(cost){}
};
int N;
ll L;
vector<edge> g[MAX_N];
int par[MAX_N][20];
ll dist[MAX_N][20];
int depth[MAX_N];
int begin[MAX_N],end[MAX_N];
int K=1;
void dfs(int v,int p,ll d,int dep)
{
    begin[v]=K++;
    depth[v]=dep;
    par[v][0]=p;
    dist[v][0]=d;
    for(int i=0;i<g[v].size();i++)
    {
        edge e = g[v][i];
        if(e.to == p)continue;
        dfs(e.to,v,e.cost,dep+1);
        K++;
    }
    end[v]=K;
}
ll search_dist(int v,int x)
{
    ll res = 0ll;
    for(int i=19;i>=0;i--)
    {
        if((x>>i)&1)
        {
            res += dist[v][i];
```

```
                v = par[v][i];
            }
        }
        return res;
    }
    int search_parent(int v,int x)
    {
        for(int i=19;i>=0;i--)
        {
            if((x>>i)&1)v = par[v][i];
        }
        return v;
    }
    int main()
    {
        scanf("%d %lld",&N,&L);
        for(int i=1;i<N;i++)
        {
            int p;
            ll l;
            scanf("%d %lld",&p,&l);
            p--;
            g[i].pb(edge(p,l));
            g[p].pb(edge(i,l));
        }
        dfs(0,-1,-1,0);
        for(int i=0;i<19;i++)
        {
            for(int j=0;j<N;j++)
            {
                if(par[j][i]==-1)
                {
                    par[j][i+1]=-1;
                    dist[j][i+1]=-1;
                }
                else
                {
```

```
                par[j][i+1]=par[par[j][i]][i];
                dist[j][i+1]=dist[j][i]+dist[par[j][i]][i];
                if(par[j][i+1]==-1)dist[j][i+1]=-1;
            }
        }
    }
    for(int i=0;i<N;i++)
    {
        int l = 0, r = depth[i]+1;
        while(r-l>1)
        {
            int mid = (l+r)/2;
            if(search_dist(i,mid)<=L)l = mid;
            else r = mid;
        }
        int p = search_parent(i,l);
        bit.add(begin[p],1);
        bit.add(begin[i]+1,-1);
    }
    for(int i=0;i<N;i++)printf("%d\n",bit.sum(begin[i])-bit.sum(end[i]));
    return 0;
}
```

4 A and B and Lecture Rooms (Codeforces Round #294 Div2 E)

問題概要

辺のコストが全て 1 の N 頂点の木が与えられる。

- 頂点 u,v からの距離が等しい頂点の数を求める

という M 個のクエリに答えよ

まず、u, v 間のパスの長さが奇数の時、感覚でわかると思いますが、答えは明らかに 0 です。念の為証明してみましょう。木では隣あう頂点に移動した時に必ず深さは 1 ずつ変化します。なので u, v のパスの長さが奇数の時、u, v の深さの偶奇は異なります。ゆえに u, v から同じ距離の頂点の深さの偶奇が一致しないので、そのような頂点がないことがわ

かります。次に偶数の時を考えてみましょう。u, v 間のパスの真ん中の頂点 x は u, v からの距離が等しいです、また x の子のうち,u, v の先祖でない頂点を根とする部分木の頂点全ても u, v からの距離が等しいです。これだけで大丈夫なように見えますが頂点 x が丁度 u, v の LCA と一致するときは x の先祖も u, v からの距離が等しくなります。また、いやらしいコーナーケースですが u, v が同じ頂点の時は u, v 自身も u, v からの距離が等しくなります。[3]

```
#include <cstdio>
#include <vector>
using namespace std;
#define pb push_back
#define MAX_N 100100
int n,m;
vector<int> g[MAX_N];
int depth[MAX_N],child[MAX_N];
int par[MAX_N][20];
int dfs(int v,int p,int d)
{
    par[v][0]=p;
    depth[v]=d;
    child[v]=1;
    for(int i=0;i<g[v].size();i++)
    {
        if(g[v][i]==p)continue;
        child[v]+=dfs(g[v][i],v,d+1);
    }
    return child[v];
}
int search(int v,int x)
{
    for(int i=19;i>=0;i--)if((x>>i)&1)v=par[v][i];
    return v;
}
int main()
{
```

[3] 要するに全頂点への距離が等しいということです

```
scanf("%d",&n);
for(int i=0;i<n-1;i++)
{
    int a,b;
    scanf("%d %d",&a,&b);
    a--;b--;
    g[a].pb(b);
    g[b].pb(a);
}
dfs(0,-1,0);
fill_table();
scanf("%d",&m);
for(int i=0;i<m;i++)
{
    int a,b;
    scanf("%d %d",&a,&b);
    a--;b--;
    if(a==b)
    {
        printf("%d\n",n);
        continue;
    }
    if(depth[a]<depth[b])swap(a,b);
    int c = lca(a,b);
    int dist = depth[a]+depth[b]-depth[c]*2;
    if(dist%2!=0)
    {
        printf("0\n");
        continue;
    }
    dist/=2;
    int u = search(a,dist);
    int v = search(a,dist-1);
    if(dist==depth[a]-depth[c])
    {
        int w = search(b,dist-1);
        printf("%d\n",n-child[v]-child[w]);
```

```
            }
            else printf("%d\n",child[u]-child[v]);
        }
        return 0;
    }
```

5 Xenia and Tree (Codeforces Round #199 Div2 E)

問題概要

辺のコストが全て 1 の N 頂点の木が与えられる。初め、根は赤く、その他は青く塗られている

- 頂点 v を赤く塗る
- 頂点 v からもっとも近い赤い頂点までの距離を求める

という M 個のクエリに答えよ

おまけです。この問題は、おそらくいままで紹介した手法では解けないと思います。ですが、紹介した手法も使います。まず、ナイーブな方解法を考えてみましょう。最初に思いつくのは、更新が起こるたびに BFS をして $O(N)$ で最短距離を更新する、という解法だと思います。また、更新クエリが連続して来る時、それらはまとめて $O(N)$ で処理することができます。[4]この性質をうまく使ってやることは出来ないでしょうか。更新クエリをある数 (B とおく) ずつに分割して、更新クエリは各まとまりごとに処理することにします。そうすると、前から順番にクエリを処理していったときに、距離を答えるクエリの前に来る更新クエリのうち、最短距離に反映されていない更新クエリの個数はたかだか B-1 個しかないという状態になります。また、まとめて処理したものからの最小距離と、反映されていない更新クエリの頂点との距離の最小値のうちの小さいほうが答えとなります。2 頂点間の距離は LCA を用いて $O(logN)$ で求められるので、計算量は $O(N*M/B+M*B*logN)$ になり、B を $\sqrt{N}$ 程度にしてやると、時間内に解くことができます。このような手法はクエリの平方分割と呼ばれたりします。[5]以下のソースコードでは $dfs, fill\_table, lca$ を省略しています。

```
#include <cstdio>
#include <algorithm>
#include <vector>
```

[4] 最初にキューに入れる始点を増やすだけです。
[5] クエリの個数 N のとき、B を $\sqrt{N}$ 程度にするとうまく行く事が多いため。バケット法とも言います。

```
#include <queue>
using namespace std;
#define pb push_back
const int SQR = 334;
int N,M;
vector<int> g[100100];
int depth[100100];
int par[100100][20];
int d[100100];
int dist(int u,int v)
{
    int l = lca(u,v);
    return depth[u]+depth[v]-depth[l]*2;
}
vector<int> lazy;
void culc()
{
    queue<int> q;
    for(int i=0;i<lazy.size();i++)
    {
        d[lazy[i]]=0;
        q.push(lazy[i]);
    }
    while(!q.empty())
    {
        int v = q.front();
        q.pop();
        for(int i=0;i<g[v].size();i++)
        {
            if(d[g[v][i]]>d[v]+1)
            {
                d[g[v][i]]=d[v]+1;
                q.push(g[v][i]);
            }
        }
    }
    lazy.clear();
```

```
}
int main()
{
    scanf("%d %d",&N,&M);
    for(int i=0;i<N-1;i++)
    {
        int a,b;
        scanf("%d %d",&a,&b);
        a--;b--;
        g[a].pb(b);
        g[b].pb(a);
    }
    dfs(0,-1,0);
    fill_table();
    for(int i=0;i<M;i++)
    {
        int t,v;
        scanf("%d %d",&t,&v);
        v--;
        if(t==1)
        {
            lazy.pb(v);
            if(lazy.size()>=SQR)culc();
        }
        else
        {
            int ans = d[v];
            for(int i=0;i<lazy.size();i++)ans = min(ans,dist(v,lazy[i]));
            printf("%d\n",ans);
        }
    }
    return 0;
}
```

第 IV 章

おわりに

私の記事を最後まで読んでいただきありがとうございます。本当は Heavy-Light Decomposition などもう少しむずかしめのテクニックまで扱う予定だったのですが締め切りに追われてしまい結局書くことができませんでした。また、実践編の解説では自分の日本語力が足りないために非常にわかりにくい説明になってしまい申し訳ありません。ですが、この記事を通して、「木って奥深いなぁ」とか「グラフ理論おもしろそう」と感じて頂いたり、「競技プログラミングを始めてみたい」と思っていただけたなら幸いです。ここで紹介できなかった問題もいろいろな Online Judge にありますので、是非挑戦してみてください。もし間違ってる箇所の指摘などがありましたら@okuraofvegetabl まで連絡していただけると嬉しいです。

参考文献

[1] 秋葉拓哉, 岩田陽一, 北川宣稔『プログラミングコンテストチャレンジブック 第 2 版』(マイナビ,2010)