Sparse Table and LCA

Reynaldo Wijaya Hendry - Universitas Indonesia

Motivation

Classic Range Minimum Query Problem

Motivation

Diberikan N elemen, ada Q query yang menanyakan dari rentang L sampai R berapa nilai minimum ? **(GA ADA UPDATENYA)**

1 <= N < = 100.000

1 <= Q <= 100.000

Motivation

Diberikan N elemen, ada Q query yang menanyakan dari rentang L sampai R berapa nilai minimumnya ? **(GA ADA UPDATENYA)**

Kalo Constrainnya jadi gini?

1 <= N < = 100.000

1 <= Q <= 10.000.000

Sparse Table

Apa itu sparse table?

Sparse Table

Table yang tidak dense

Dense Table

\	**1**	2	3	4
1	5	3	3	1
2	3	3	3	1
3	3	3	4	1
4	1	1	1	1

Sparse Table

j\i	0	1
1	3	1
2	3	-1
3	1	-1
4	-1	-1

Sparse Table

Tabel[i][j] = menandakan minimum dari rentang i sampai i + (1 << j)

\	0	1
1	3	1
2	3	-1
3	3	-1
4	-1	-1

Sparse Table

Misal kita ingin query dari rentang 1 sampai 3

\	0	1
1	3	1
2	3	-1
3	3	-1
4	-1	-1

Sparse Table

Misal kita ingin query dari rentang 1 sampai 3
Tinggal min(tabel[1][0], tabel[2][0])

\	0	1
1	3	1
2	3	-1
3	3	-1
4	-1	-1

Construction

Cara Konstruksinya gimana?

Gampang

Construction

Base Case

```
memset(tabel, -1, sizeof tabel);
tabel[i][0] = min(val[i], val[i+1]);
```

Case Setelahnya

if(i + (1 << (j-1)) <= N)

```
tabel[i][j] = min(tabel[i][j - 1], tabel[i + (1 << (j -1))][j - 1] );
```

Query

Untuk Range dari L sampai R kita ingin mendapat 2 indeks dimana

Indeks L sampai L + (1 << i) untuk suatu i dan j + (1 << k) sampai R

Query

Untuk Range dari L sampai R kita ingin mendapat 2 indeks dimana

Indeks L sampai L + (1 << i) untuk suatu i dan j + (1 << k) sampai R

Val	4	3	1	6	7	8	9

Indeks	1	2	3	4	5	6	7

Query

Untuk Range dari L sampai R kita ingin mendapat 2 indeks dimana

Indeks L sampai L + (1 << i) untuk suatu i dan i + (1 << k) sampai R

Misal kita mau nyari dari indeks ke 2 - 7

Val	4	3	1	6	7	8	9

Indeks	1	2	3	4	5	6	7

Query

Untuk Range dari L sampai R kita ingin mendapat 2 indeks dimana

Indeks L sampai L + (1 << i) untuk suatu i dan i + (1 << k) sampai R

Misal kita mau nyari dari indeks ke 2 - 7

2 sampai 2 + (1 << 2) dan 3 sampai 3 + (1 << 2) artinya min(range(2, 6) + range(3, 7))

Val	4	3	1	6	7	8	9

Val	4	3	1	6	7	8	9

Indeks	1	2	3	4	5	6	7

Query

Untuk Range dari L sampai R kita ingin mendapat 2 indeks dimana

Indeks L sampai L + (1 << i) untuk suatu i dan i + (1 << k) sampai R

Misal kita mau nyari dari indeks ke 2 - 7

2 sampai 2 + (1 << 2) dan 3 sampai 3 + (1 << 2) artinya min(range(2, 6) + range(3, 7))

Query di tabelnya jadi `min(tabel[2][2], tabel[3][2]);`

Val	4	3	1	6	7	8	9
Indeks	1	2	3	4	5	6	7

Query

Kita harus mencari i terbesar yang masih memenuhi L + (1 << i) <= R, artinya kita mencari i terbesar yang memenuhi (1 << i) <= R

Kita bisa menggunakan fungsi log sehingga kita bisa mendapatkan

```
int logaritma = log(R - L) / log(2.0);
ans = min(tabel[L][logaritma], tabel[R - (1 << logaritma)][logaritma]);
return ans;
```

Query

SALAH!!!

Query

Kita harus mencari i terbesar yang masih memenuhi L + (1 << i) <= R, artinya kita mencari i terbesar yang memenuhi (1 << i) <= R

Kita bisa menggunakan fungsi log sehingga kita bisa mendapatkan

Cari salahnya

```
int logaritma = log(R - L) / log(2.0);
ans = min(tabel[L][logaritma], tabel[R - (1 << logaritma)][logaritma]);
return ans;
```

Query

Kita harus mencari i terbesar yang masih memenuhi L + (1 << i) <= R, artinya kita mencari i terbesar yang memenuhi (1 << i) <= R

Kita bisa menggunakan fungsi log sehingga kita bisa mendapatkan

Apabila R dan L sama maka logaritma akan menghitung log(0) dimana log (0) tidak terdefinisi

```
int logaritma = log(R - L) / log(2.0);
ans = min(tabel[L][logaritma], tabel[R - (1 << logaritma)][logaritma]);
return ans;
```

Query

Kita harus mencari i terbesar yang masih memenuhi L + (1 << i) <= R, artinya kita mencari i terbesar yang memenuhi (1 << i) <= R

Kita bisa menggunakan fungsi log sehingga kita bisa mendapatkan

```
if(R - L == 0) return val[R];
int logaritma = log(R - L) / log(2.0);
ans = min(tabel[L][logaritma], tabel[R - (1 << logaritma)][logaritma]);
return ans;
```

Kompleksitas

O(N log N)

Buat Tabel N log N

Query O(1)

Back To Problem

Diberikan N elemen, ada Q query yang menanyakan dari rentang L sampai R berapa **jumlahannya**? Pake Sparse Table

1 <= N < = 100.000

1 <= Q <= 100.000

Back To Problem

Bisa dan Tidak Bisa

Karena Sum tidak memiliki properti overlap, dimana apabila sum terdapat overlap maka hasilnya berbeda, tetapi kita masih dapat mencarinya dengan kompleksitas query menjadi log N.

Query 2.0

Caranya gimana?

Query 2.0

Caranya gimana?

Mirip seperti tadi kita manfaatkan MSB tapi karena tidak bisa overlap maka kita harus perkecil MSB nya perlahan-lahan

Query 2.0

Caranya gimana?

Mirip seperti tadi kita manfaatkan MSB tapi karena tidak bisa overlap maka kita harus perkecil MSB nya perlahan-lahan

Val	4	3	1	6	7	8	9

Indeks	1	2	3	4	5	6	7

Query 2.0

Caranya gimana?

Mirip seperti tadi kita manfaatkan MSB tapi karena tidak bisa overlap maka kita harus perkecil MSB nya perlahan-lahan

```
int L = 1;
ans += tabel[L][2]
L += (1 << i) + 1
```

Val	4	3	1	6	7	8	9

Indeks	1	2	3	4	5	6	7

Query 2.0

Caranya gimana?

Mirip seperti tadi kita manfaatkan MSB tapi karena tidak bisa overlap maka kita harus perkecil MSB nya perlahan-lahan

```
// L = 6
ans += tabel[L][0]
```

Val	4	3	1	6	7	8	9

Indeks	1	2	3	4	5	6	7

Query 2.0

Generalisir Kasus

```
for(int i = MAXLOG; i >= 0; i--) {
    if(L + (1 << i) <= R) {
        ans = ans + tabel[L][i];
        L = L + (1 << i) + 1;
    }
}
```

Val	4	3	1	6	7	8	9

Indeks	1	2	3	4	5	6	7

Any Question for Sparse Table?

Motivation

Diberikan sebuah tree ada N node, tiap node punya value, ada Query hitung jumlahan value semua node yang terletak pada jalan dari L menuju R.

Constraint

1 <= N <= 100.000

1 <= Q <= 100.000

Motivation

Kalo Treenya datar gampang. Tinggal Prefix Sum doang

Motivation

Kalo Treenya datar gampang. Tinggal **Prefix Sum** doang.

Kita bisa gunakan juga teknik prefix sum ini di tree.

Motivation

Kalo Treenya datar gampang. Tinggal **Prefix Sum** doang.

Kita bisa gunakan juga teknik prefix sum ini di tree.

Kita asumsikan pref[i] menyimpan jumlahan node i ke root

Motivation

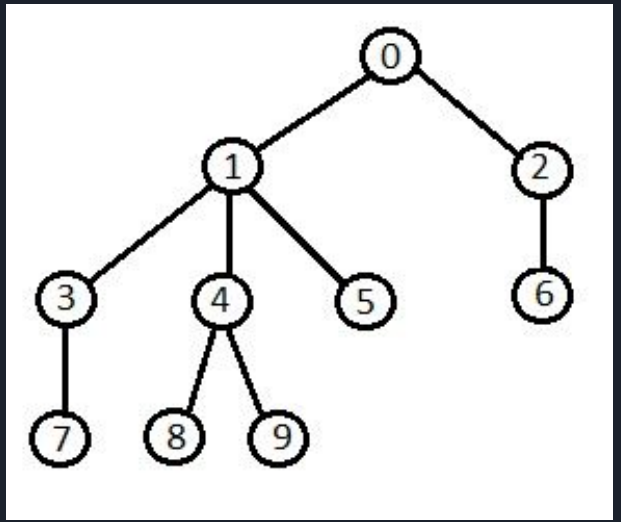
Kalo Treenya datar gampang. Tinggal **Prefix Sum** doang.

Kita bisa gunakan juga teknik prefix sum ini di tree.

Kita asumsikan pref[i] menyimpan jumlahan node i ke root

Maka untuk mencari dari L sampai R kita tinggal menghitung

pref[L] + pref[R]. Tetapi masalahnya ada yang overlap

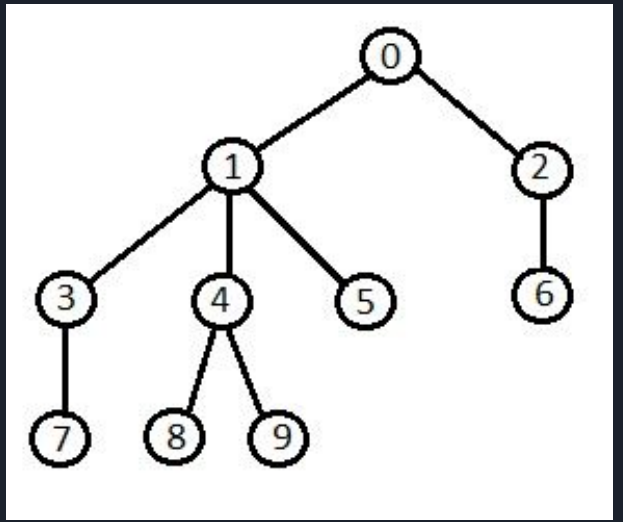


Motivation

Misal Querynya 7 dan 5 maka

Pref[7] = x[7] + x[3] + x[1] + x[0]

Pref[5] = x[5] + x[1] + x[0]



Motivation

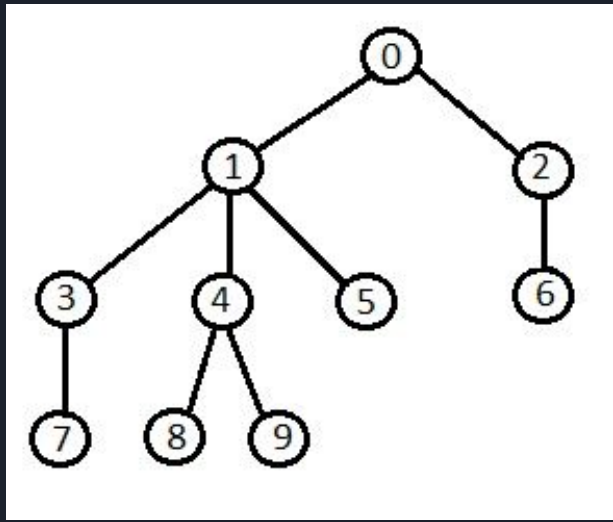
Misal Querynya 7 dan 5 maka

Pref[7] = x[7] + x[3] + **x[1] + x[0]**

Pref[5] = x[5] + **x[1] + x[0]**

Ada yang sama, kalo misalkan kita dapatkan 1 maka kita dapat

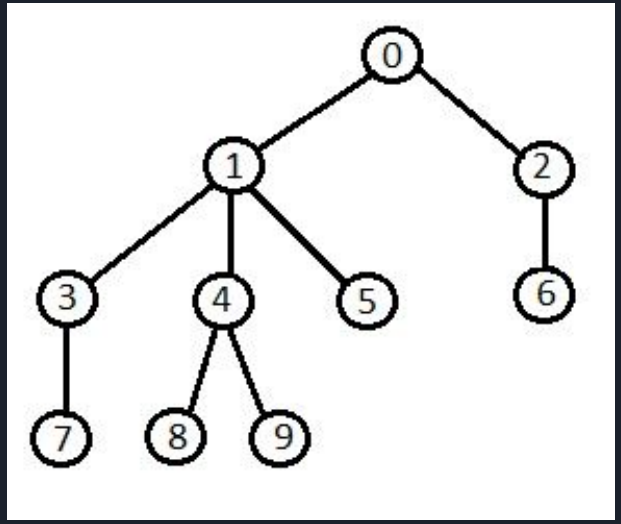
menghitung dengan cara pref[7] + pref[5] - pref[1] - pref[0]



Motivation

Masalahnya adalah bagaimana cara kita mendapatkan angka 1 ini

Angka 1 ini adalah yang kita sebut sebagai LCA atau kepanjangannya *Lowest Common Ancestor*



Definisi

Lowest = Terkecil

Common = Sama

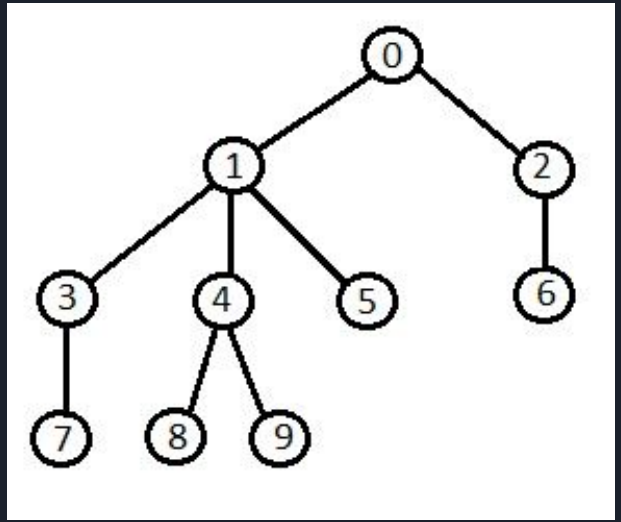
Ancestor

Ancestor Terkecil yang sama

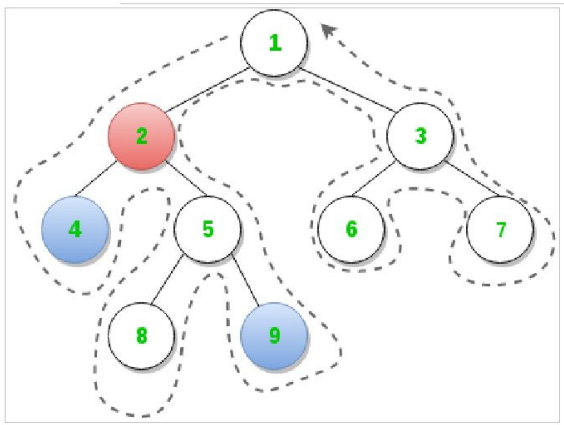
LCA(7, 5) = 1

LCA(8, 9) = 4

LCA(3, 6) = 0



Euler Tour



Euler Tour

An euler tour of the tree starting from node 1 will yield:

1	2	4	2	5	8	5	9	5	2	1	3	6	3	7	3	1

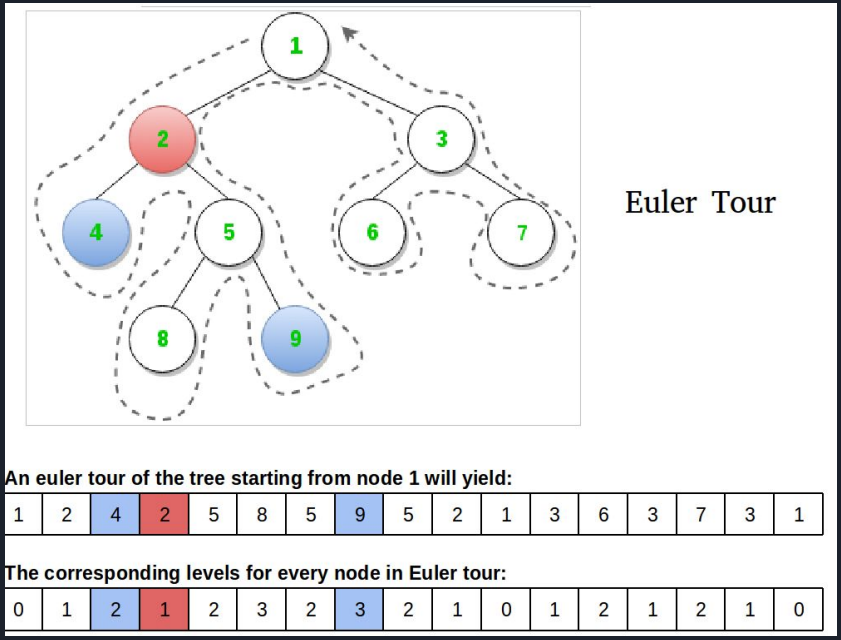
The corresponding levels for every node in Euler tour:

0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0

LCA

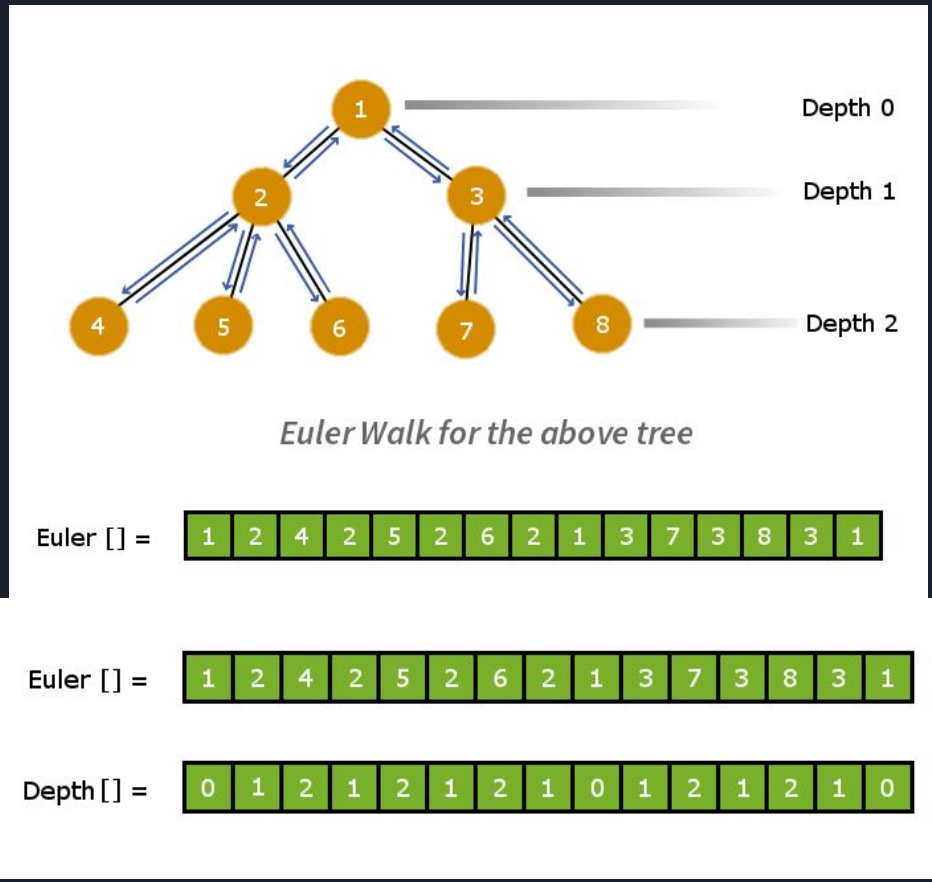
Menggunakan Euler tour

Mencari Depth Minimum menggunakan euler tour



LCA

4 - 6


1
2
3
4
5
6
7
8
Depth 0
Depth 1
Depth 2
Euler Walk for the above tree
Euler [] =
1 2 4 2 5 2 6 2 1 3 7 3 8 3 1
Euler [] =
1 2 4 2 5 2 6 2 1 3 7 3 8 3 1
Depth [] =
0 1 2 1 2 1 2 1 0 1 2 1 2 1 0

LCA

Generalisir Kasus

Misalkan E adalah Array Euler Tour

Misalkan D adalah Array Depth dari node yang ada di Euler Tour

Misalkan U adalah Array Ubah yang menerima indeks node dan mengubahnya menjadi indeks node tersebut di Array E

Maka LCA dari A ke B adalah

```
LCA(A, B) = E[RMQ(U[A], U[B])]
```

Dimana RMQ akan dilakukan di Array D dan mengreturn **Indeks** yang merupakan jawaban dari RMQ terkecil

Kompleksitas

Generate Table O(N log N)

Query O(1)

Kompleksitas Total O(N log N)

LCA v2.0

Motivasi : Lebih pendek, Lebih intuitif(?)

Kekurangan : Setiap Query jadi Log N

Nama lain : Binary Lifting

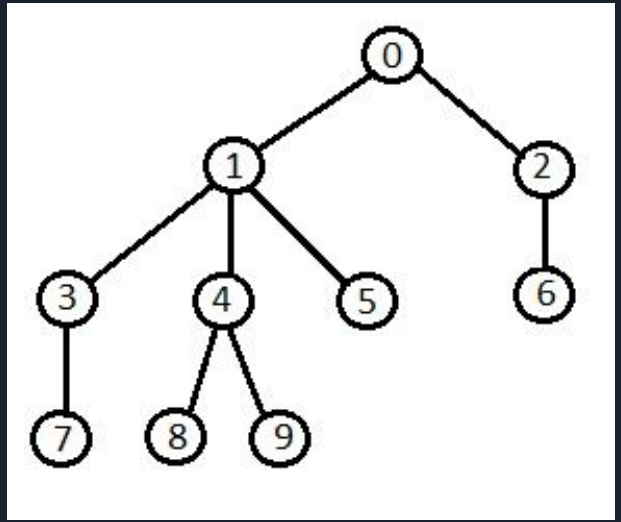
LCA v2.0

Idenya sama seperti tadi yaitu menggunakan sparse table, tapi kalo ini tabel yang dibutuhkan hanyalah sparse tabel parent dan tabel depth.

LCA v2.0

\	0	1	2
0	-1	-1	-1
1	0	-1	-1
2	0	-1	-1
3	1	0	-1
4	1	0	-1
5	1	0	-1
6	2	0	-1
7	3	1	-1
8	4	1	-1
9	4	1	-1

Tabel Depth Ga usah ya



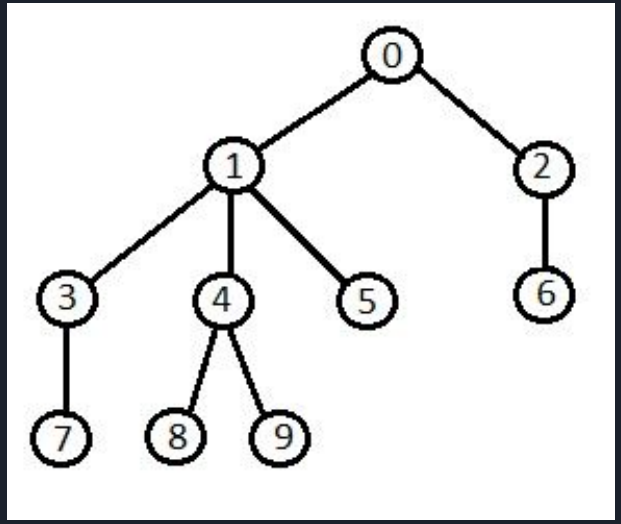
LCA v2.0

\	0	1	2
0	-1	-1	-1
1	0	-1	-1
2	0	-1	-1
3	1	0	-1
4	1	0	-1
5	1	0	-1
6	2	0	-1
7	3	1	-1
8	4	1	-1
9	4	1	-1

Misalkan LCA(7, 6)

Hal yang pertama kita harus lakukan adalah sesuaikan depthnya.

Menjadi LCA(3, 6)



LCA v2.0

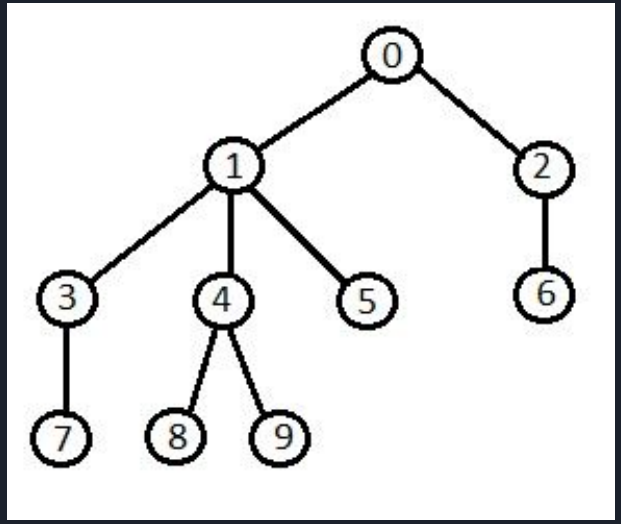
\	0	1	2
0	-1	-1	-1
1	0	-1	-1
2	0	-1	-1
3	1	0	-1
4	1	0	-1
5	1	0	-1
6	2	0	-1
7	3	1	-1
8	4	1	-1
9	4	1	-1

Misalkan LCA(7, 6)

Hal yang pertama kita harus lakukan adalah sesuaikan depthnya.

Menjadi LCA(7, 3)

Idenya adalah kita ingin meloncat hingga suatu titik dimana loncatan mereka berbeda



LCA v2.0

Untuk kodingan, pertama harus hitung tabel depth dan tabel parent tepat diatasnya dengan dfs biasa, untuk cara membuat table parent sama dengan cara membuat sparse table

```
memset(pa, -1, sizeof pa);
for(int i = 0; i < LOGN; i++) {
    // Menggunakan first indexing
    for(int j = 1; j <= N; j++) {
        if(i == 0) pa[i][j] = parent[j];
        else if(pa[i-1][j] != -1) {
            pa[i][j] = pa[i-1][pa[i-1][j]];
        }
    }
}
```

LCA v2.0

```
int LCA(int u, int v) {
    //Menyamakan tinggi
    if(depth[u] < depth[v]) swap(u, v);
    int diff = depth[u] - depth[v];
    for(int i = LOGN; i >= 0; i--) {
        if(diff & (1 << i)) u = pa[i][u];
    }
    // kalo mereka sudah sama
    if(u == v) return u;
    // menaikkan hingga satu sebelum LCAnya
    for(int i=LOGN; i >= 0; i--) {
        if(pa[i][u] != pa[i][v]) {
            u = pa[i][u]; v = pa[i][v];
        }
    }
    // yang direturn parentnya
    return pa[0][u];
}
```

Latihan Soal

Diberikan sebuah tree ada N node, tiap node punya value, ada Query hitung node dengan value **maksimum** pada semua node yang terletak pada jalan dari L menuju R.

Constraint

1 <= N <= 100.000

1 <= Q <= 100.000

Solusi

Binary Lifting

LCA

Solusi

Kita buat tabel baru yang merupakan sparse table baru maks yang menyerupai sparse table parent. Lakukan query LCA dari u ke v. Sparse table maks juga ikut naik mengikuti sparse table parent.

Soal Lagi

Ada tree N nodes, ada M orang yang terletak pada tree, bisa aja ada 2 atau lebih oran g pada satu node (punya value).

Ada Q query, ditanya dari path L ke R print K orang terkecil

(Coba Pikirin dulu deh 10 menit)

1 <= N, M, Q , X_i<= 100.000

1 <= L, R <= N

1 <= K <= 10

Solusi

LCA Sparse Table

Solusi

Bikin vector of sparse table yang bakal menyimpan paling banyak 10 nilai minimum.

Untuk build sparse tablenya tinggal bandingkan 2 sparse table (bisa menggunakan 2 pointer)

Untuk quernya tinggal menyimpan 10 nilai terkecil

Any Question?

Thank You