Aula 13 · Menor Ancestral Comum, Decomposição Pesado-Leve e Menor no Maior

Desafios de Programação

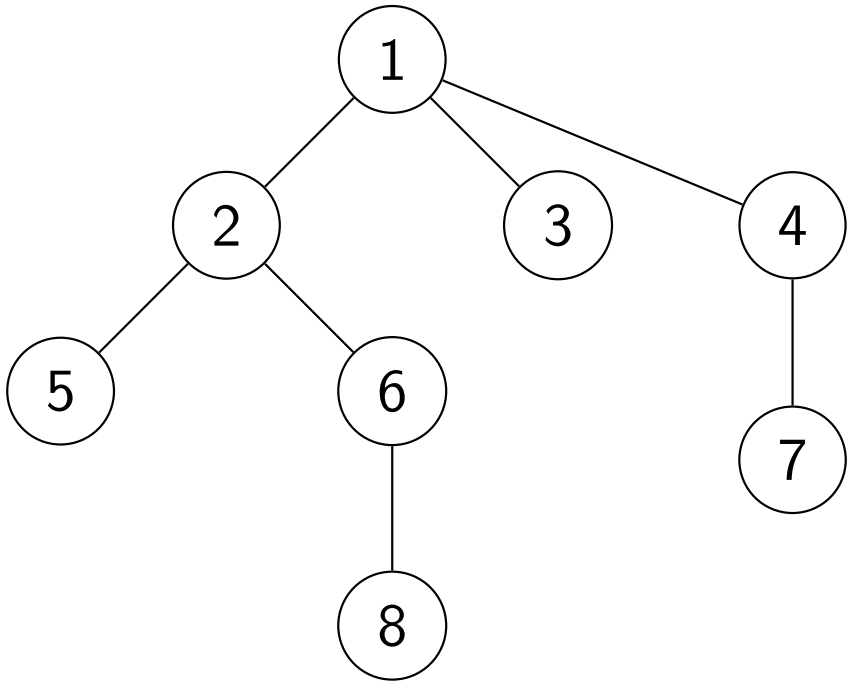
Fernando Kiotheka Vinicius Tikara Date

UFPR

25/11/2024

Menor Ancestral Comum

Menor Ancestral Comum



Subida binária

Árvore enraizada.

Código `bl.h`

```
const int L = log2(N);
vector<int> depth (N, 0);
vector<vector<int>> weiop (N, vector<int>(L+1));
vector<vector<int>> up (N, vector<int>(L+1));
void bl_euler_tour(int u, int p, int w) {
    up[u][0] = p; weiop[u][0] = w; depth[u] = depth[p] + 1;
    for (auto [v, w] : g[u]) if (v != p)
        bl_euler_tour(v, u, w);
}
void bl_init(int u, int n) {
    depth[u] = 0; bl_euler_tour(u, u, 0);
    for (int l = 0; l < L; l++)
    for (int u = 0; u < n; u++) {
        int a = up[u][l];
        up[u][l+1] = up[a][l];
        weiop[u][l+1] = OP(weiop[u][l], weiop[a][l]);
    }
}
```

Menor ancestral comum

Código `bllca.h`

```
int bl_lca(int a, int b) {
    if (!(depth[b] < depth[a])) { swap(a, b); }
    int diff = depth[a] - depth[b];
    for (int l = L; l >= 0; l--) if (diff & (1 << l))
        a = up[a][l];
    if (a == b) { return a; }
    for (int l = L; l >= 0; l--) if (up[a][l] != up[b][l])
        a = up[a][l], b = up[b][l];
    return up[a][0];
}
```

Operações no caminho

Código `blop.h`

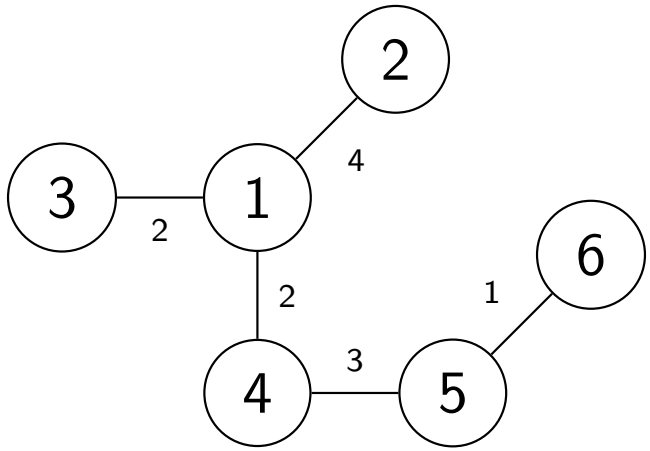
```
int bl_op(int a, int b) {
    if (!(depth[a] > depth[b])) { swap(a, b); }
    int res = NEUTRAL;
    int diff = depth[a] - depth[b];
    for (int l = L; l >= 0; l--) if (diff & (1 << l)) {
        res = OP(res, weiop[a][l]); a = up[a][l]; }
    if (a == b) { return res; }
    for (int l = L; l >= 0; l--)
        if (up[a][l] != up[b][l]) {
            res = OP(res, OP(weiop[a][l], weiop[b][l]));
            a = up[a][l], b = up[b][l];
        }
    return OP(res, OP(weiop[a][0], weiop[b][0]));
}
```

Maior peso entre dois vértices de uma árvore

Código `blmax.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ii = pair<int, int>;
const int N = 1e5+15;
vector<vector<ii>> g (N);
#define NEUTRAL 0
#define OP(X, Y) max(X, Y)
#include "bl.h"
#include "blop.h"
int main() {
    int n, m, q; cin >> n >> m >> q;
    while (m--) {
        int u, v, w; cin >> u >> v >> w; u--; v--;
        g[u].push_back(ii(v, w)); g[v].push_back(ii(u, w));
    }
    bl_init(0, n);
    while (q--) {
        int u, v; cin >> u >> v; u--; v--;
        cout << bl_op(u, v) << "\n";
    }
}
```

Exemplo de maior peso entre dois vértices de uma árvore



Entrada	Saída
6 5 5 3 1 2 1 4 2 2 1 4 4 5 3 5 6 1 1 6 2 6 3 2 3 4 5 6	3 4 4 2 1

Decomposição em Cadeias

Decomposição em cadeias

Decomposição em Cadeias (ou Decomposição Pesado-Leve) é uma técnica que resolve muitos problemas de consulta em árvore.

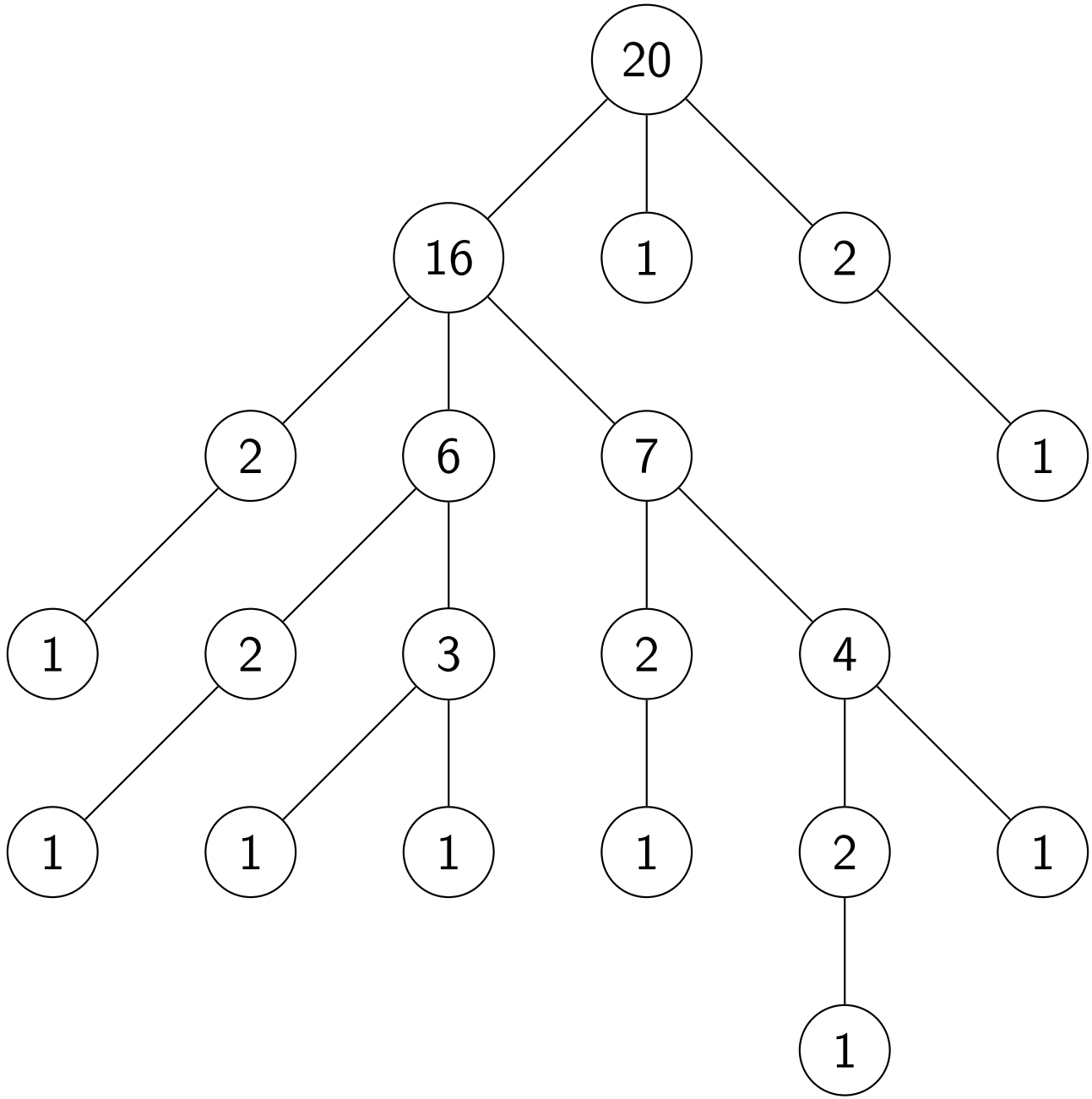
- Dividir a árvore inteira em várias cadeias disjuntas de maneira que qualquer nó pode chegar na raiz atravessando $\mathcal{O}(\lg N)$ cadeias.
- Consultas do tipo "soma no caminho de a até b" viram "soma de todas as cadeias no caminho de a até b".
- Que cadeias disjuntas são essas?

Arestas pesadas e leves

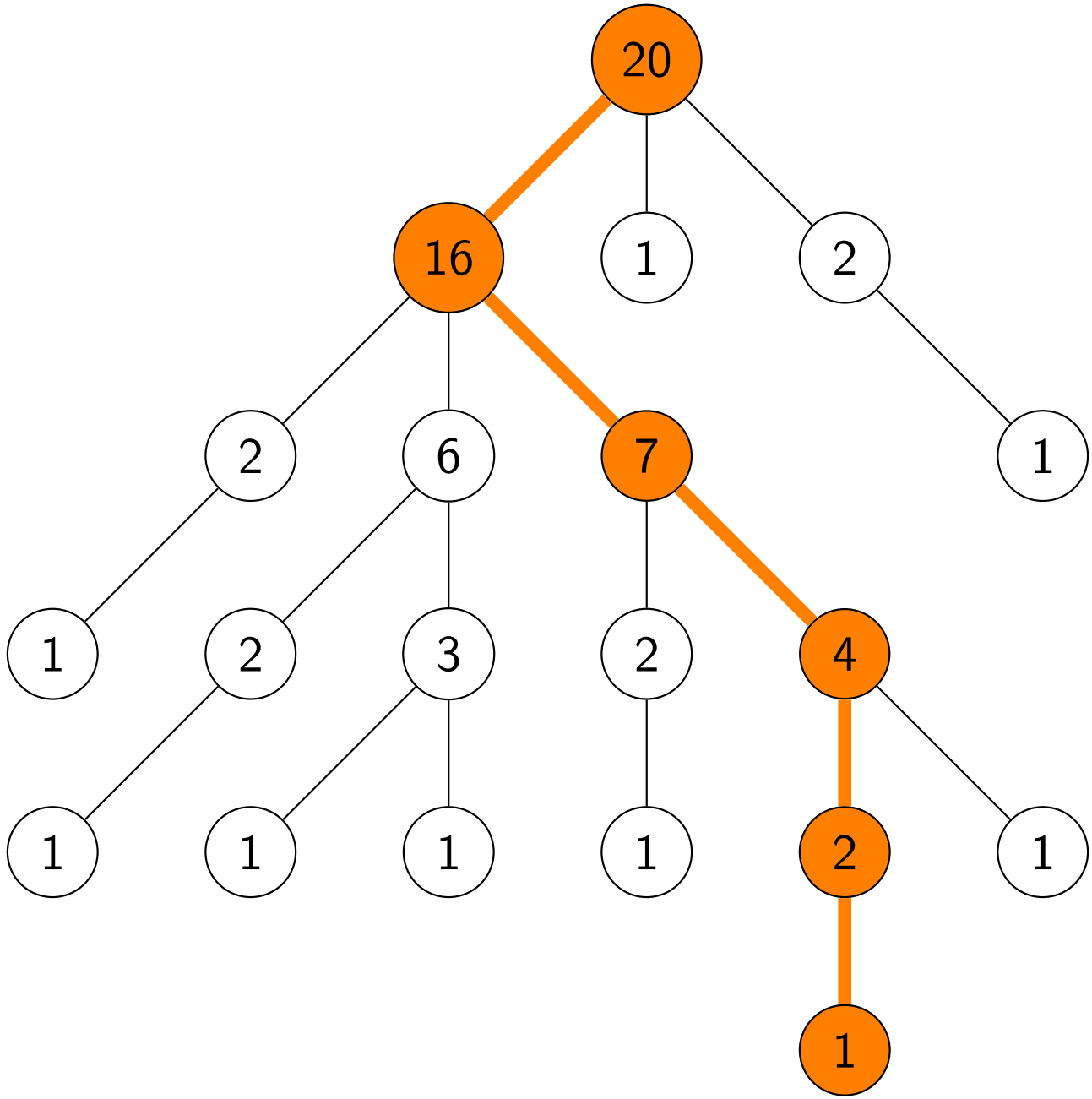
- Seja $S(u)$ a quantidade de vértices na subárvore do vértice u (incluindo u).
- Chamaremos a aresta (u, v) de **pesada** se $S(v)$ for maior que todas as outras arestas. Caso múltiplas satisfaçam esse requisito, escolha uma.
- Todas as outras arestas (u, v) serão ditas **leves**.
- Indo da raiz até qualquer vértice, passamos por no máximo $\mathcal{O}(\lg N)$ arestas leves.
- Isto porque andar para uma aresta leve reduz o tamanho da subárvore que vamos considerar **pela metade**:

$$S(v) < \frac{S(u)}{2}$$

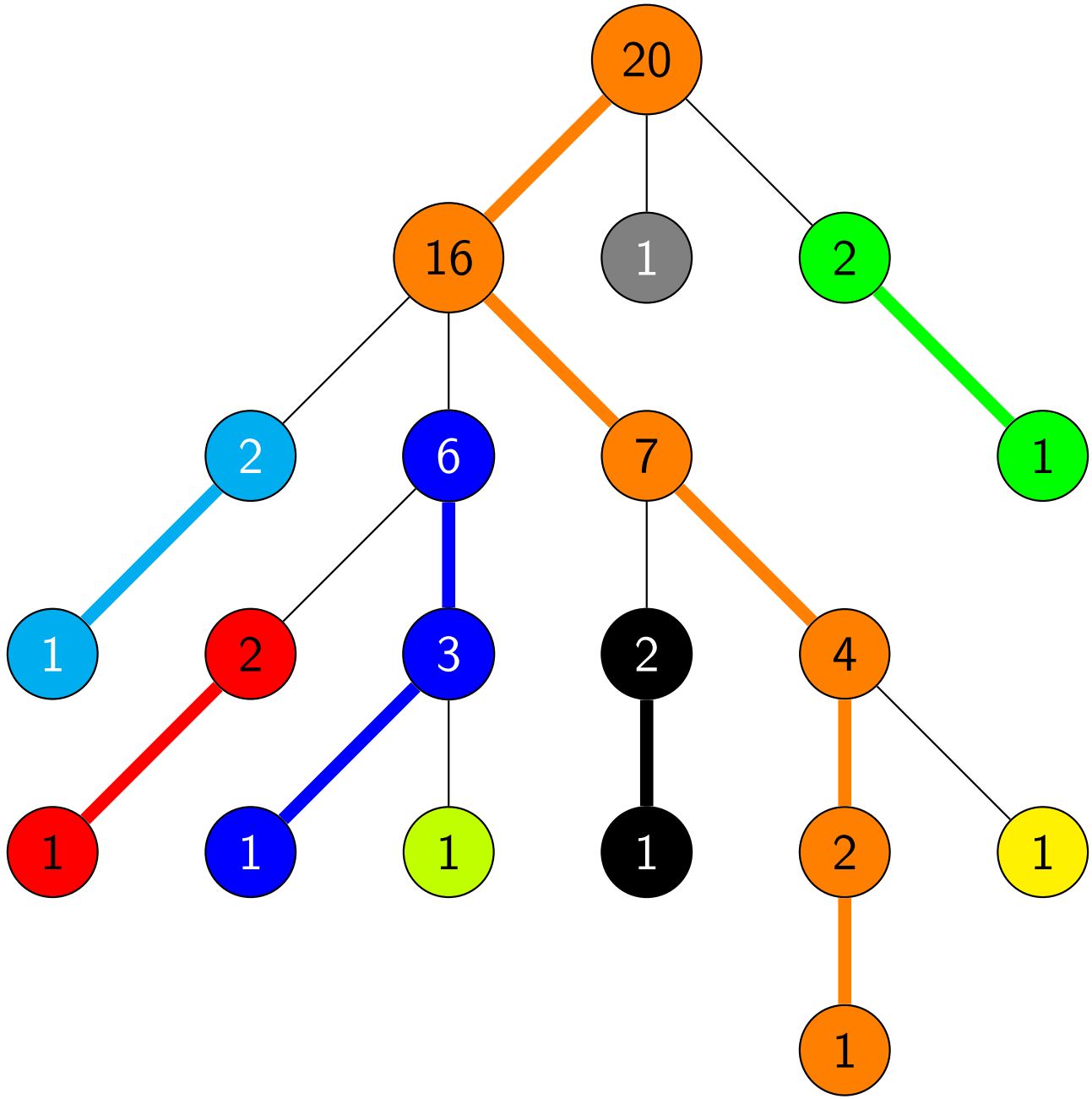
As cadeias pesadas



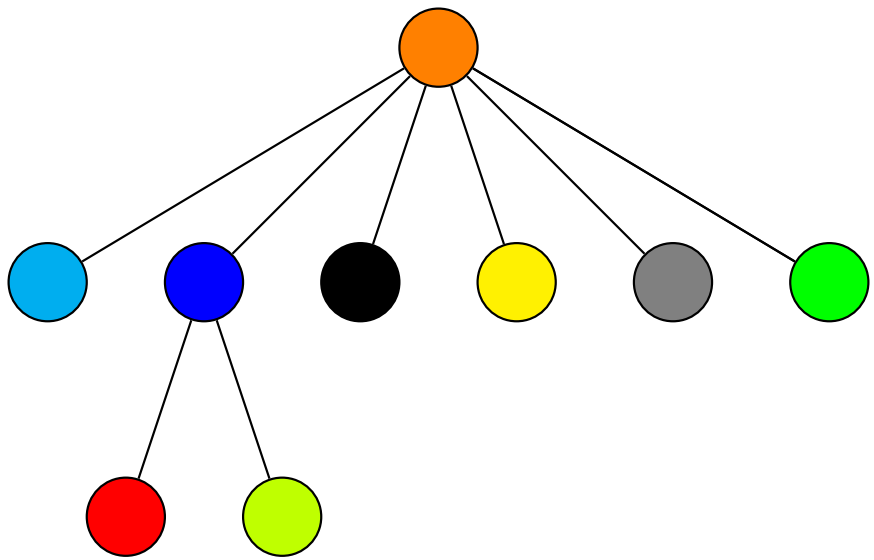
As cadeias pesadas



As cadeias pesadas



As cadeias pesadas (comprimidas)



Obtendo as cadeias pesadas

- Primeiro preencheremos o $S(u)$ de cada vértice. Faremos isso usando uma busca em profundidade.
- Nessa busca já podemos estabelecer para cada vértice a aresta que é pesada, então vamos guardar qual o filho que é pesado para cada u.
- Em seguida, faremos outra busca em profundidade, agora montando as cadeias. Manteremos um vetor que diz qual o elemento cabeça de cada cadeia, e continuaremos a cadeia, montando uma sequência de índices referente a aquela cadeia.
- Depois de terminarmos uma cadeia, chamaremos a função recursivamente para os filhos leves criarem suas próprias cadeias.

Preenchendo os filhos pesados

Código `hldfill.h`

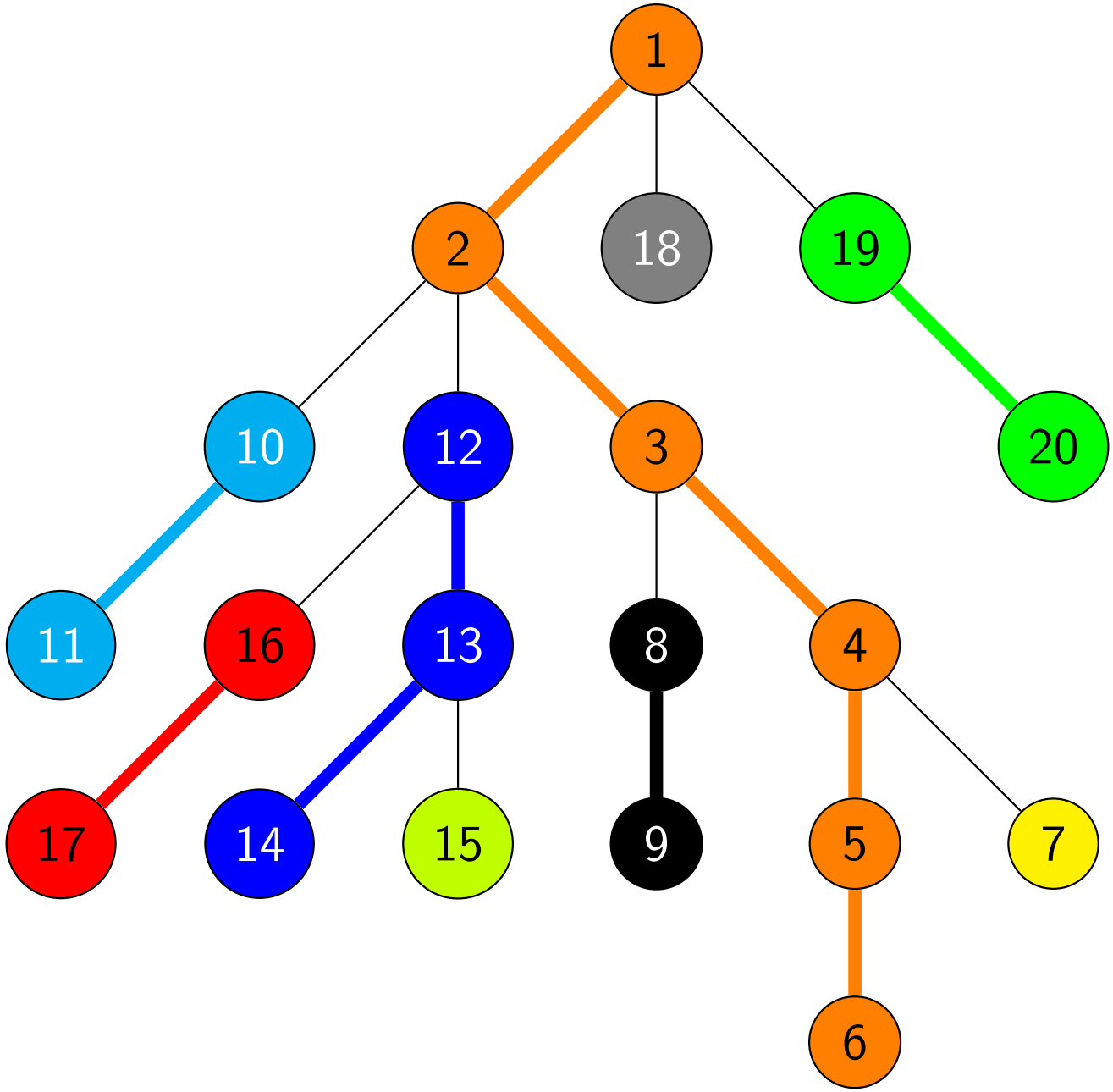
```
vector<int> heavy (N, -1), par (N, -1);
vector<int> depth (N), wei (N);
int hld_fill /* O(V + E) */ (int u, int w) {
    int s = 1, maxs = 0;
    wei[u] = w;
    for (auto [v, w] : g[u]) if (v != par[u]) {
        par[v] = u;
        depth[v] = depth[u] + 1;
        int cs = hld_fill(v, w);
        s += cs;
        if (cs > maxs) {
            maxs = cs;
            heavy[u] = v;
        }
    }
    return s;
}
```

Montando as cadeias

Código `hld.h`

```
vector<int> hds (N), ixs (N), origin (N);
int cix = 1;
void hld /* O(n lg n) */ (int u, int h) {
    hds[u] = h;
    origin[cix] = wei[u];
    ixs[u] = cix++;
    if (heavy[u] != -1)
        hld(heavy[u], h); // continue chain
    for (auto [v, w] : g[u])
        if (v != par[u] && v != heavy[u])
            hld(v, v); // new chain
}
```

Os índices das cadeias pesadas



E agora?

- Temos esse vetor colorido:

$$\left[1\,2\,3\,4\,5\,6\,7\,8\,9\,10\,11\,12\,13\,14\,15\,16\,17\,18\,19\,20\right]$$

Algumas propriedades:

- A subárvore de um vértice está em um intervalo contíguo.
- Caminhos na árvore são compostos de $\mathcal{O}(\lg N)$ intervalos.

Qual o próximo passo então? Usar uma estrutura de dados que trabalha bem com intervalos.

- Vamos usar a Árvore de Segmentos Preguiçosa.
- Então atualização e consulta em intervalos em $\mathcal{O}(\lg N)$.
- Operações em caminho acabam com complexidade $\mathcal{O}(\lg^2 N)$.
- Note que se você pré-computar a soma total dos intervalos e não atualizá-los, a complexidade pode ser de $\mathcal{O}(\lg N)$.

Fazendo operações nas cadeias

Código `hldop.h`

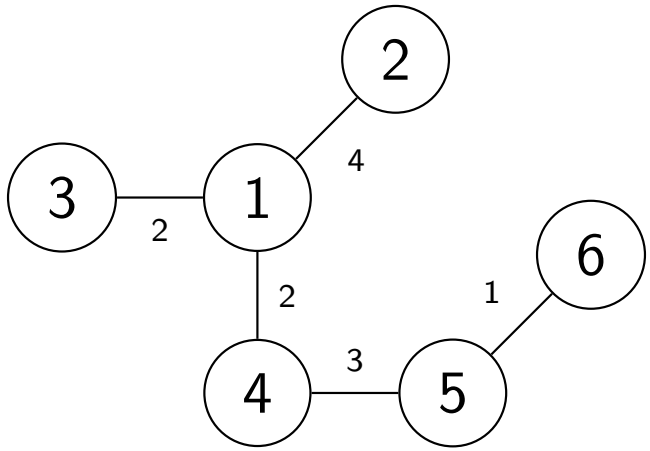
```
// O(lg~2 n)
ll hld_op(int u, int v) {
    ll ans = 0;
    for (; hds[u] != hds[v]; v = par[hds[v]]) {
        if (depth[hds[u]] > depth[hds[v]]) { swap(u, v); }
        ans = OP(ans,
            op_inclusive(ixs[hds[v]], ixs[v]));
    }
    if (depth[u] > depth[v]) { swap(u, v); }
    // Remove +1 if values are associated with vertices
    return OP(ans, op_inclusive(ixs[u] + 1, ixs[v]));
}
```

Maior peso entre dois vértices de uma árvore

Código `hldmax.cpp`

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long; using ii = pair<int, int>;
const int N = 1e5+15; vector<vector<ii>> g (N);
#define NEUTRAL 0
#define OP(X, Y) max(X, Y)
#define FACTOR 1
#include "lst.h"
#include "lstbuild.h"
#include "hldfill.h"
#include "hld.h"
#include "hldop.h"
int main() {
    int n, m, q; cin >> n >> m >> q;
    while (m--) {
        int u, v, w; cin >> u >> v >> w; u--; v--;
        g[u].push_back(ii(v, w)); g[v].push_back(ii(u, w)); }
    hld_fill(0, 0); hld(0, 0); build(origin);
    while (q--) {
        int u, v; cin >> u >> v; u--; v--;
        cout << hld_op(u, v) << "\n"; }
}
```

Exemplo de maior peso entre dois vértices de uma árvore



Entrada	Saída
6 5 5 3 1 2 1 4 2 2 1 4 4 5 3 5 6 1 1 6 2 6 3 2 3 4 5 6	3 4 4 2 1

Onde ir daqui

Algoritmo	Construção	Ancestral	Atualização	Memória
Subida Binária	$\mathcal{O}(n \lg n)$	$\mathcal{O}(\lg n)$	–	$\mathcal{O}(n \lg n)$
Tabela Esparsa	$\mathcal{O}(n \lg n)$	$\mathcal{O}(1)$	–	$\mathcal{O}(n \lg n)$
Árvore de Segmentos	$\mathcal{O}(n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(n)$
Decomp. Pesado-Leve	$\mathcal{O}(n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$	$\mathcal{O}(n)$

Menor no maior

Código `small-to-large.h`

```
vector<int> hvy (N), sz (N, 1);

int calc_size(int u, int p) {
  for (int v : g[u]) if (v != p)
    sz[u] += calc_size(v, u);
  return sz[u];
}

void dfs_add(int u, int p, int x) {
  // add u
  for (int v : g[u]) if (v != p && !hvy[v])
    dfs_add(v, u, x);
}

void small_large(int u, int p, bool keep) {
  int mx = -1, h = -1;
  for (int v : g[u]) if (v != p && sz[v] > mx) {
    mx = sz[v]; h = v;
  }
  for (int v : g[u]) if (v != p && v != h)
    small_large(v, u, 0);
  if (h != -1) { small_large(h, u, 1); hvy[h] = 1; }
  dfs_add(u, p, +1);
  // solve query for u
  if (h != -1) { hvy[h] = 0; }
  if (keep == 0) { dfs_add(u, p, -1); }
}
```

E isso é tudo!

- O conteúdo que vocês precisam para fazer a competição que logo começará está todo aqui.
- Bons estudos!