Heavy-Light Decomposition

Laboratório de Programação Competitiva - 2020

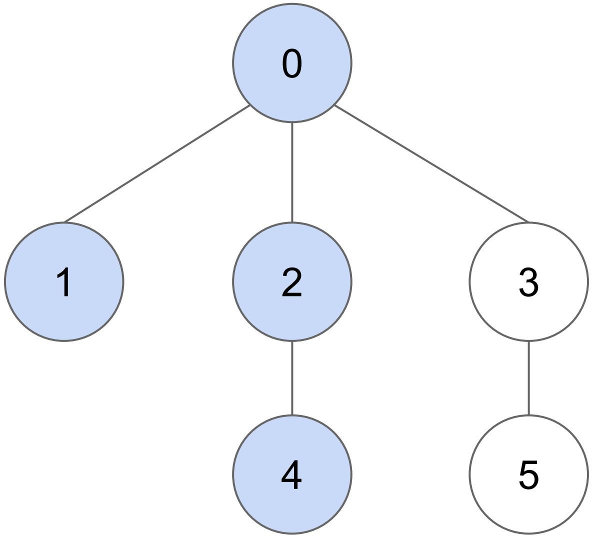
Pedro Henrique Paiola

Motivação

- Anteriormente, vimos uma série de estruturas de dados e técnicas para realizar *range queries* em *arrays*:
    - Segment Tree
    - Vetor de prefixos
    - BIT
    - Sparse Table
- A partir disso, será que podemos generalizar essas estruturas para lidar com árvores?

Motivação

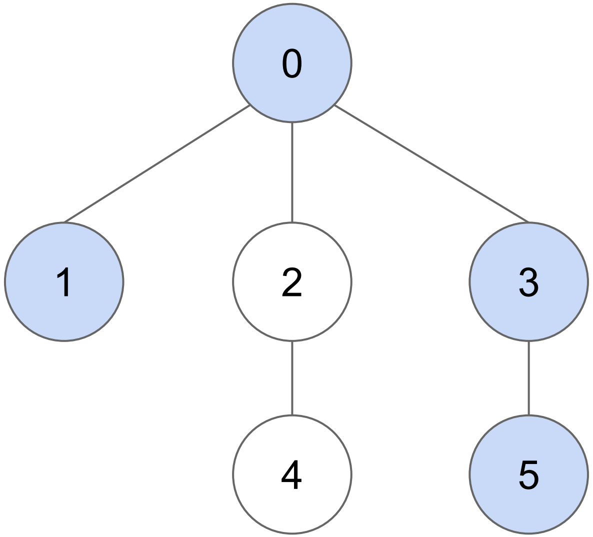
- O problema é tratar as ramificações. Elas nos impedem de (ou dificultam) tratar todos os possíveis segmentos (que representam caminhos de um vértice a outro) linearizados em um único vetor.



1	0	2	4

Motivação

- O problema é tratar as ramificações. Elas nos impedem de (ou dificultam) tratar todos os possíveis segmentos (que representam caminhos de um vértice a outro) linearizados em um único vetor.

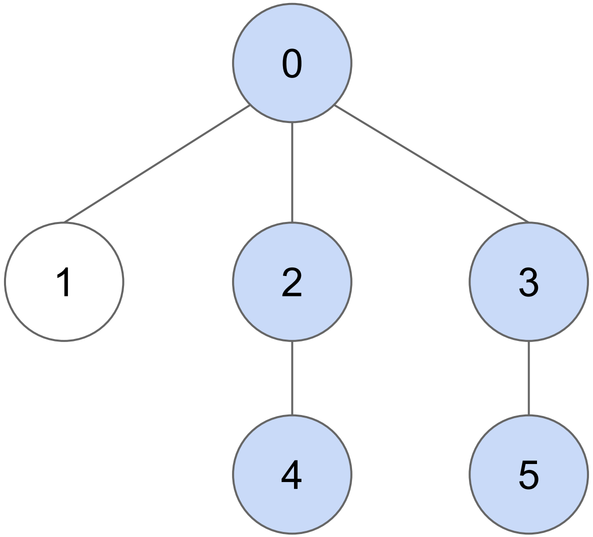


1	0	2	4

1	0	3	5

Motivação

- O problema é tratar as ramificações. Elas nos impedem de (ou dificultam) tratar todos os possíveis segmentos (que representam caminhos de um vértice a outro) linearizados em um único vetor.



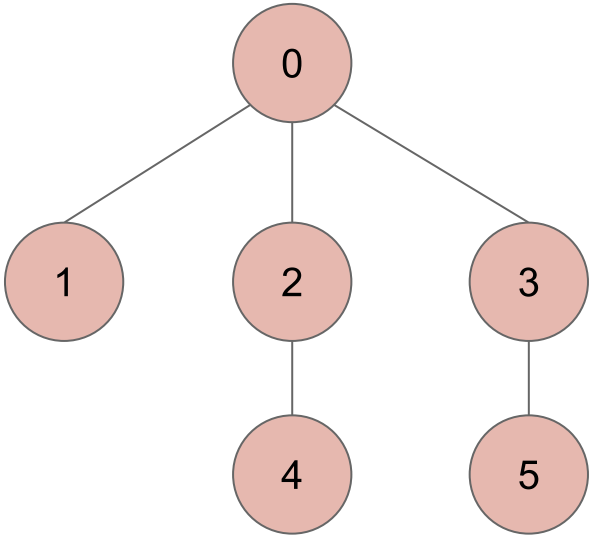
1	0	2	4

1	0	3	5

4	2	0	3	5

Motivação

- O problema é tratar as ramificações. Elas nos impedem de (ou dificultam) tratar todos os possíveis segmentos (que representam caminhos de um vértice a outro) linearizados em um único vetor.



1	0	2	4

1	0	3	5

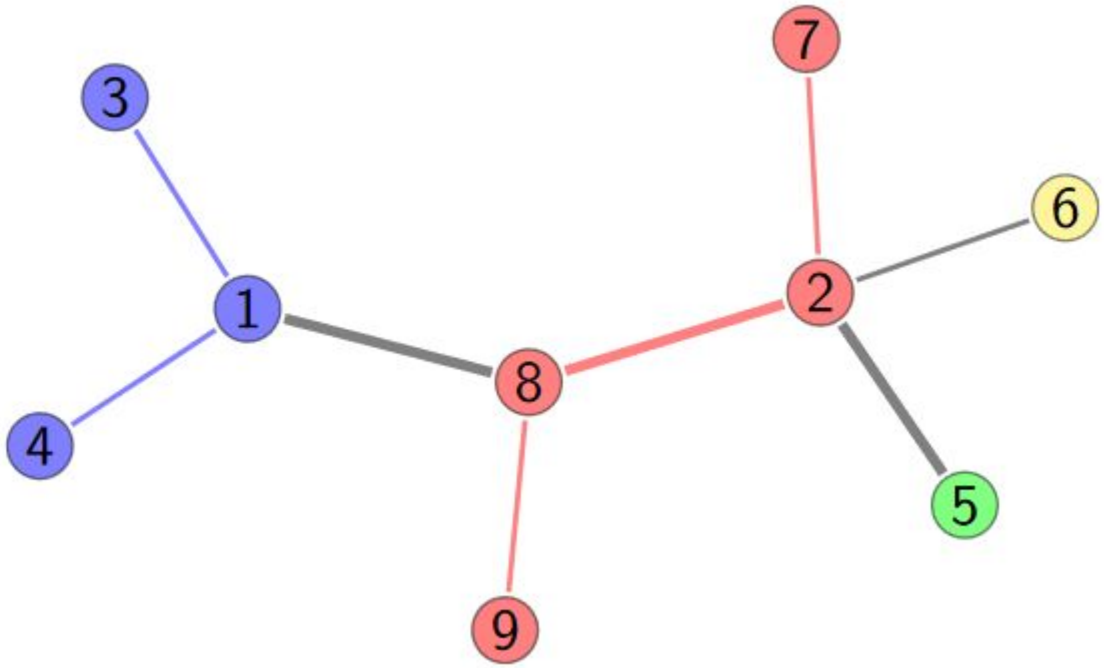
4	2	0	3	5

?	?	?	?	?	?

Motivação

- Porém, como ficou claro pelo exemplo anterior, sabemos tratar caminhos (usando SegTree, por exemplo).
- Sabemos combinar respostas parciais.
- Sendo assim, podemos decompor nossa árvore em caminhos (*chains*).
- Para cada *chain*, sua intersecção com o caminho da *query* vai ser um intervalo contíguo.
- Podemos realizar uma *query* em uma SegTree para cada *chain* e combinar as respostas.

Motivação



`query(1,5) = f(qblue(1,1), qred(8,2), qgreen(5,5))`

Heavy-Light Decomposition

- **Objetivo**: decompor uma árvore em vários caminhos disjuntos para que possamos alcançar o vértice raiz de qualquer **v** percorrendo no máximo **log n** caminhos.
- Com isso, uma *range query* única da forma “calcular algo no caminho de **a** até **b**” será reduzida para várias consultas do tipo “calcular algo no segmento **[l; r]** do **k**$^{\text{ésimo}}$ caminho”, sendo **[l;r]** $\subset$ **[a;b]**.

Heavy-Light Decomposition

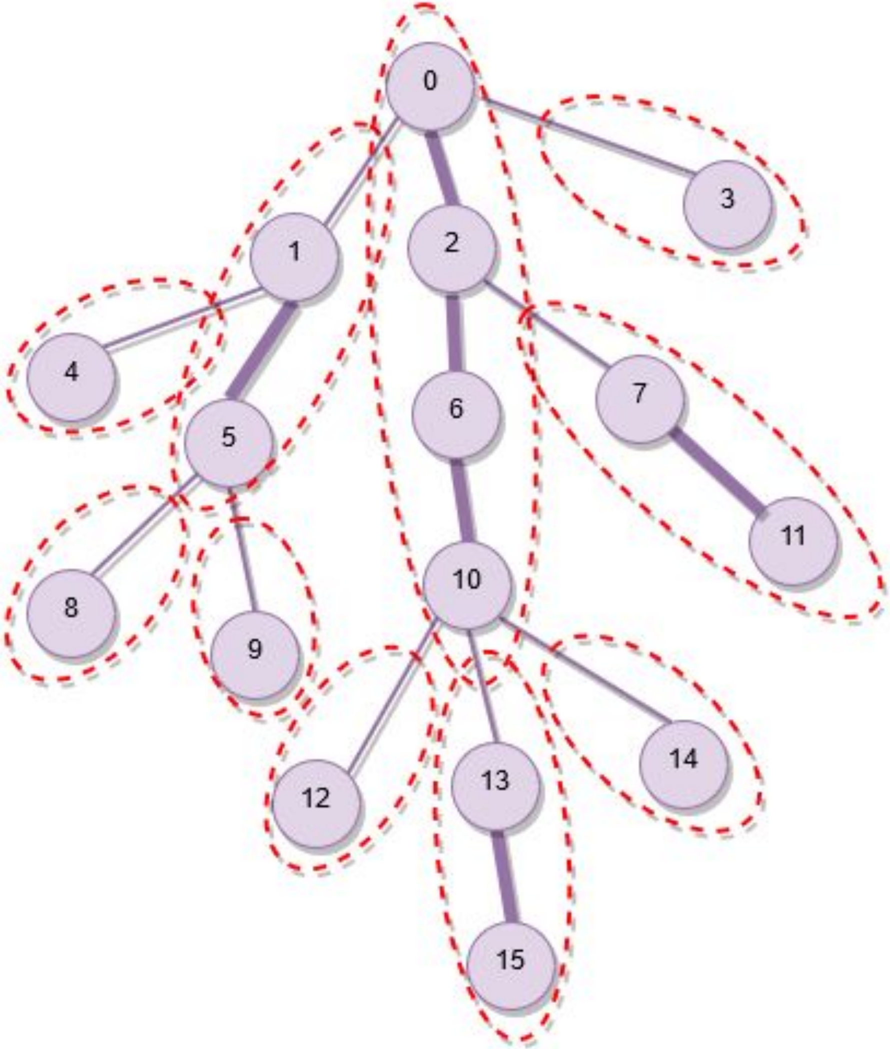
- Considere **s(v)** como sendo o tamanho da subárvore do vértice **v**, ou seja, o número de vértices na subárvore de **v** incluindo ele mesmo.
- **Definição**: dizemos que uma aresta que parte de **v** é **“heavy”** se levar a um vértice **c** de modo que:

$$s(c) \geq \frac{s(v)}{2} \iff \text{aresta } (v, c) \iff \text{heavy}$$

- Todas as outras arestas são definidas como **“light”**.

Heavy-Light Decomposition

- Perceba que cada vértice pode ter **no máximo 1** aresta heavy, pois caso contrário o vértice **v** teria pelo menos dois filhos de tamanho maior ou igual a **s(v)/2,** de forma que **s(v) >= 1 + 2*s(v)/2 > s(v)**, o que é um absurdo.
- Agora vamos decompor a árvore em caminhos disjuntos. Considere todos os vértices que não possuem nenhum filho ligado por uma aresta **heavy**. Subiremos de cada um desses vértices apenas utilizando arestas **heavy**.
- Os caminhos encontrados são chamados de **heavy paths**, e são os caminhos que desejamos.


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Heavy-Light Decomposition

- Para descer da raiz da árvore para um vértice arbitrário, o caminho encontrado está disposto em **até log n** *heavy paths*.
- Só saímos de um *heavy path* para outro por uma aresta *light*, caso contrário, se fosse uma aresta *heavy*, estaríamos no mesmo *heavy path*.
- Descer por uma aresta *light* implica em reduzir o tamanho da subárvore atual pelo menos na metade.
- Sendo assim, podemos no **máximo** percorrer **log n** arestas *light* antes que o tamanho da subárvore se reduza a um.

Exemplo: valor mínimo entre vértices

- **Problema:** dado dois vértices **a** e **b**, determinar o valor mínimo no caminho entre os vértices **a** e **b**.
- Construímos antecipadamente uma *heavy-light decomposition* da árvore.
- Sobre cada *heavy path* construiremos uma segment tree, que permitirá procurar um vértice com valor mínimo atribuído no segmento especificado pelo *heavy path* em O(log n).
- Embora o número de *heavy paths* seja O(n), a soma dos **tamanhos** de cada *heavy path* é exatamente **n** (todos os vértices estão distribuídos pelos *heavy paths*), sendo assim, a soma dos tamanhos de cada SegTree é O(n).

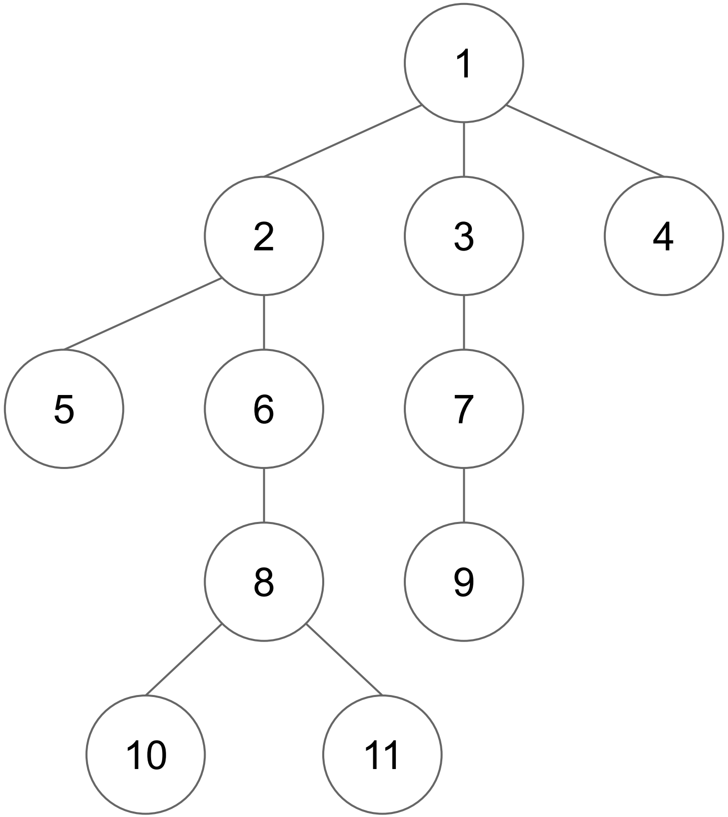
Exemplo: valor mínimo entre vértices

- Para responder a uma consulta **(a, b)**, encontramos o Menor Ancestral Comum de **a** e **b**, definido como **x**. Agora a tarefa foi reduzida para duas consultas: **(a, x)** e **(b, x)**.

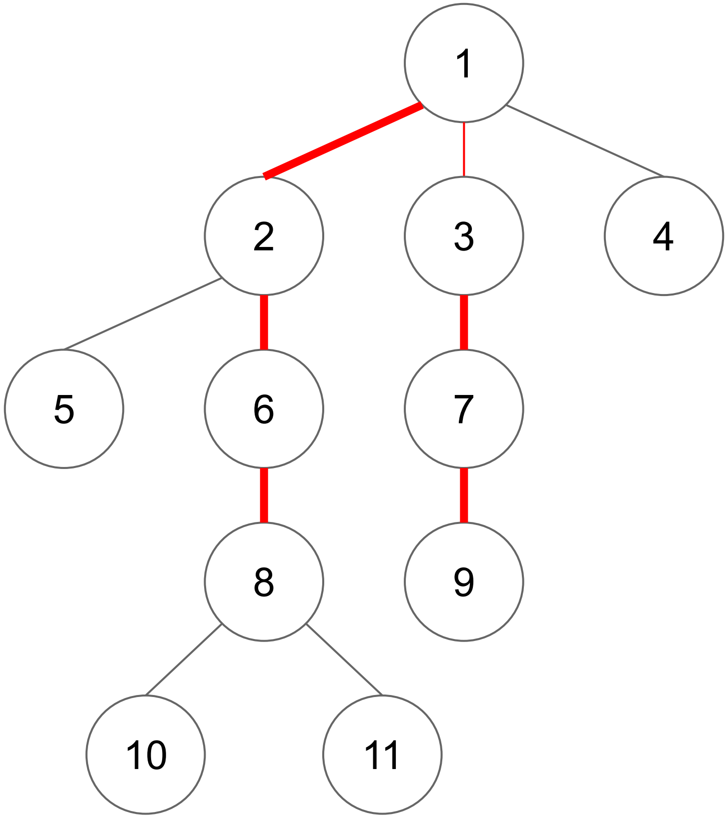
  Algoritmo:

  a. x = lca(a,b)
  b. resp = ∞
  c. Para cada uma das consultas **v** ∈ **{a, b}**:
     i. Enquanto chain(v) != chain(x)
        1. resp = min(resp , $\text{queryST}_{\text{chain(v)}}$(v, head(chain(v))))
        2. v = pai(head(chain(v)))
     ii. resp = min(resp , $\text{queryST}_{\text{chain(v)}}$(v, x))

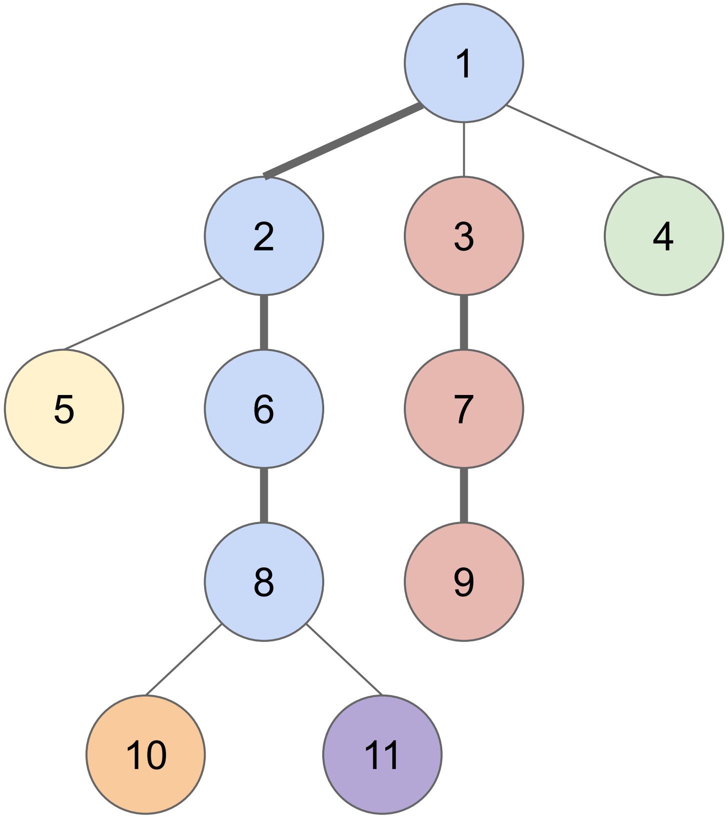
Exemplo: valor mínimo entre vértices



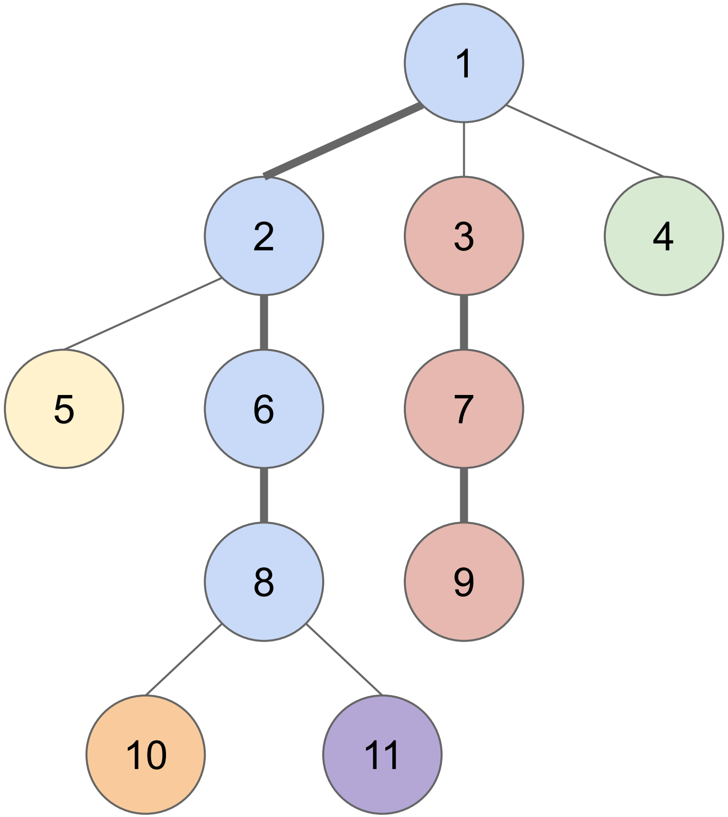
Exemplo: valor mínimo entre vértices



Exemplo: valor mínimo entre vértices



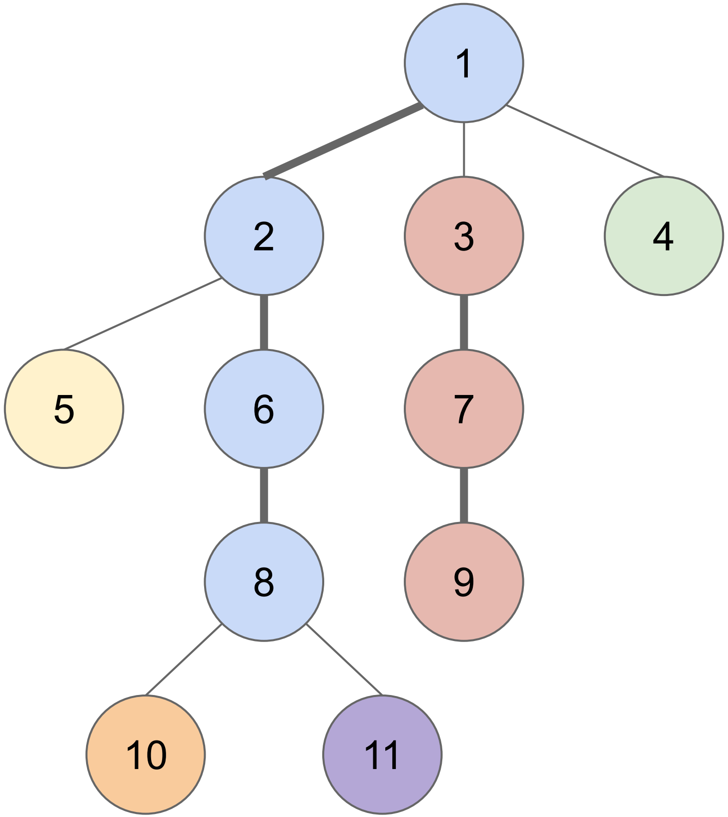
Exemplo: valor mínimo entre vértices



Poderíamos ter um vetor **value** que mapeia um valor para cada vértice, e queremos obter em uma consulta **(a, b),** o menor **value[v]** sendo **v** qualquer vértice no caminho de **a** até **b**.

Para simplificar o exemplo, consideraremos o valor de cada nó como sendo seu próprio índice.

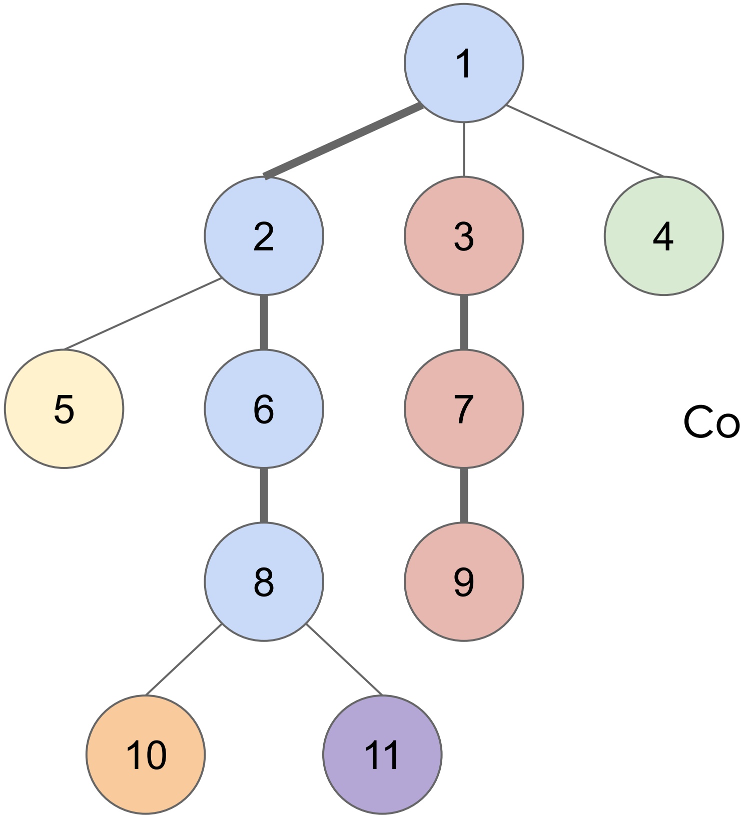
Exemplo: valor mínimo entre vértices



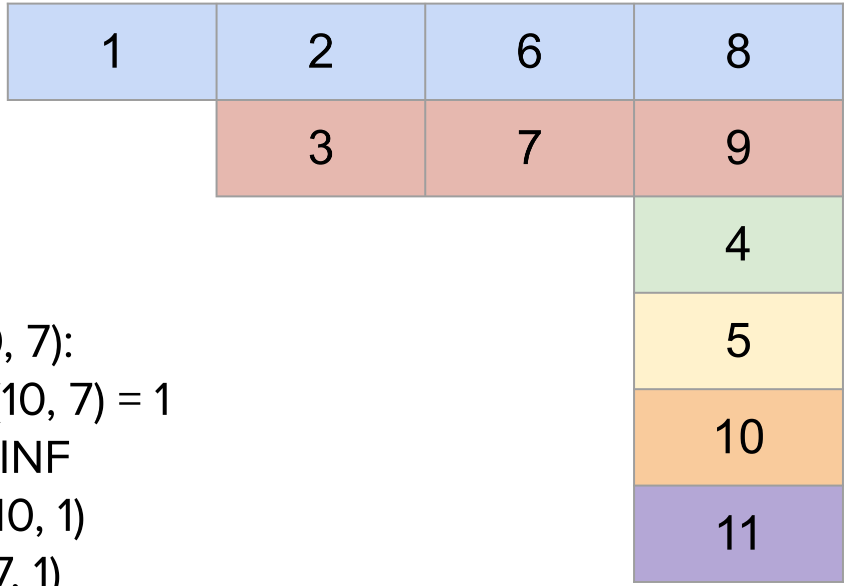
Segment Trees:

1	2	6	8
	3	7	9
			4
			5
			10
			11

Exemplo: valor mínimo entre vértices



Segment Trees para os “arrays”:



Consulta(10, 7):

x = lca(10, 7) = 1

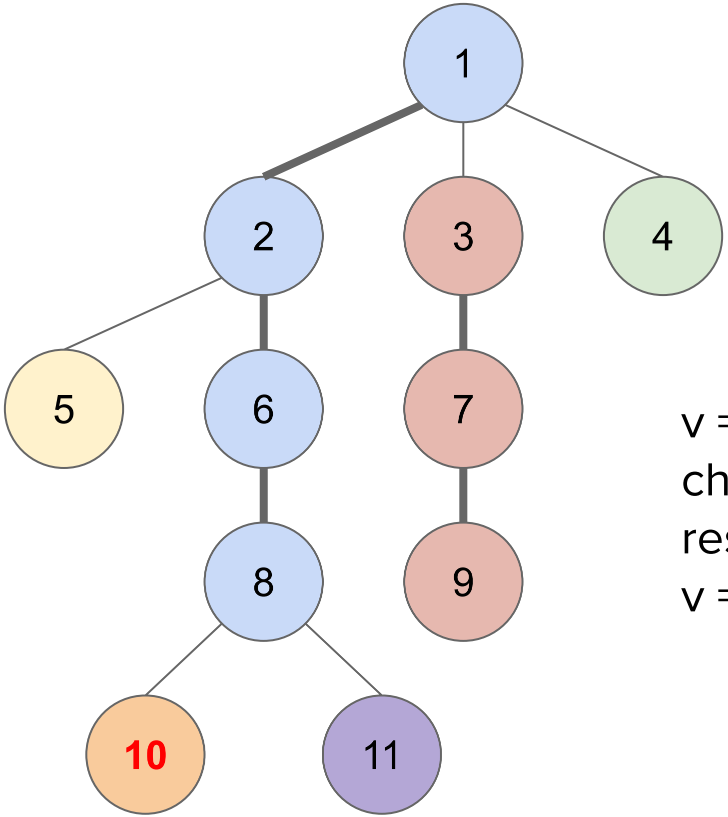
resp = INF

query(10, 1)

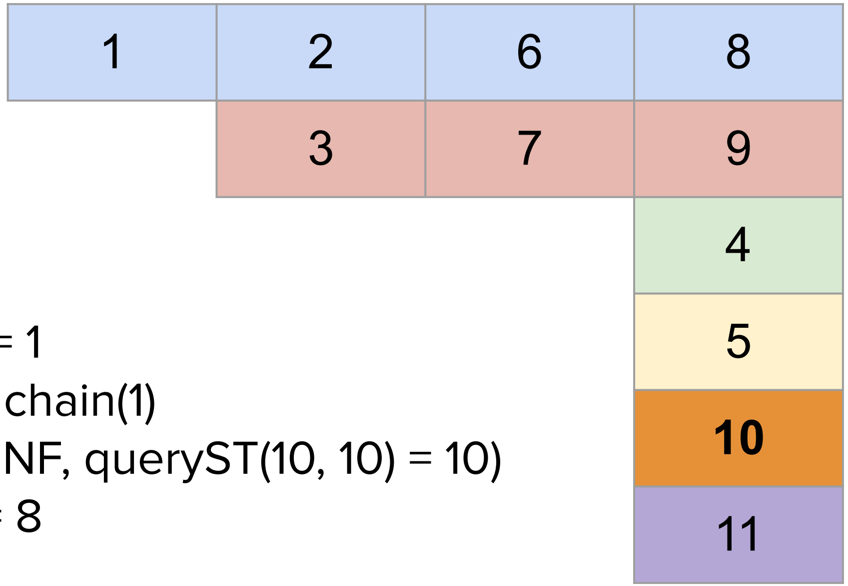
query(7, 1)

resp = INF

Exemplo: valor mínimo entre vértices



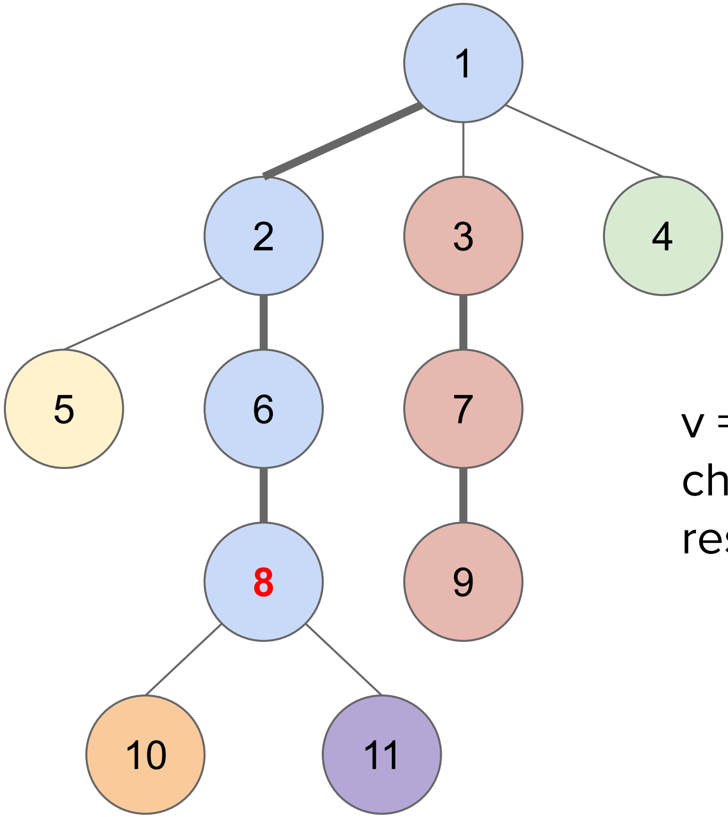
Segment Trees para os “arrays”:



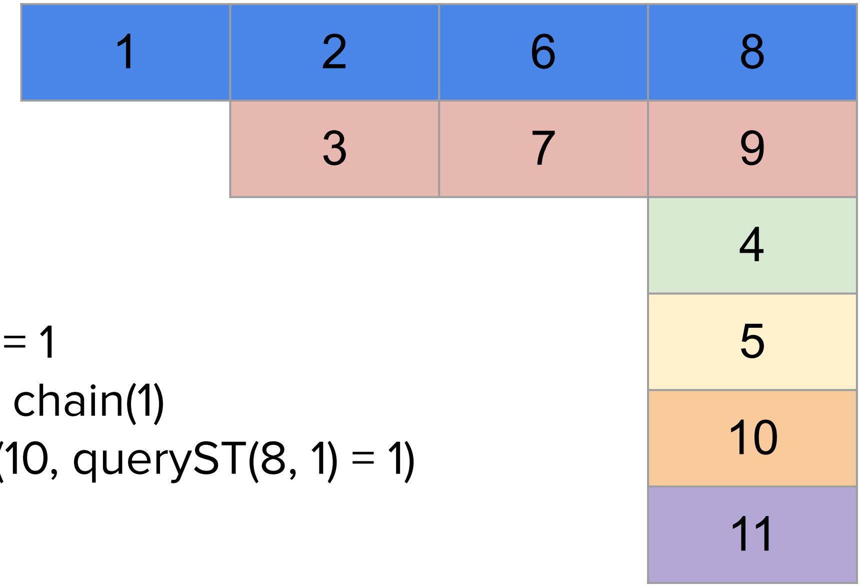
v = 10   x = 1
chain(10) != chain(1)
resp = min(INF, queryST(10, 10) = 10)
v = pai[10] = 8

resp = 10

Exemplo: valor mínimo entre vértices



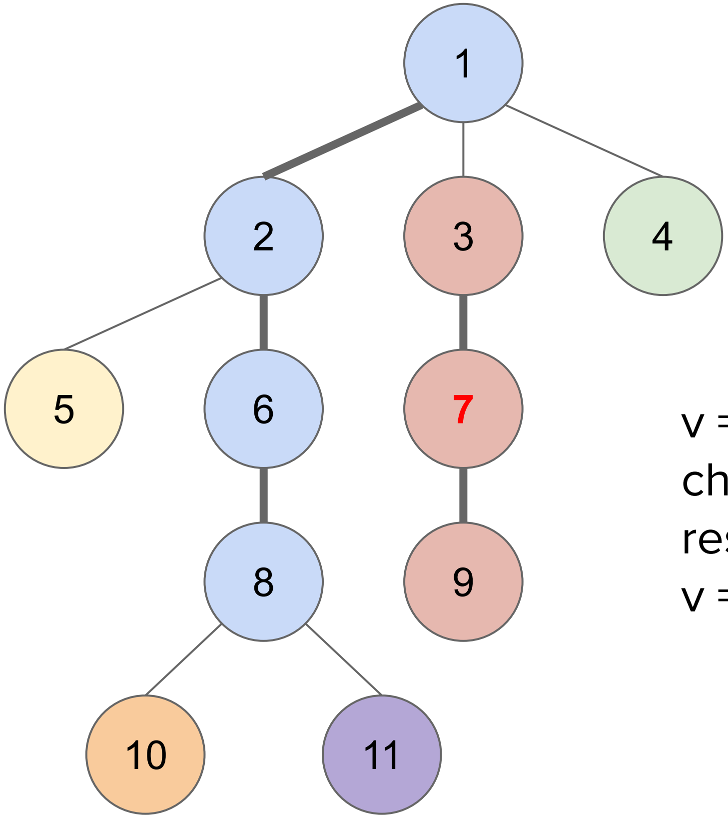
Segment Trees para os "arrays":



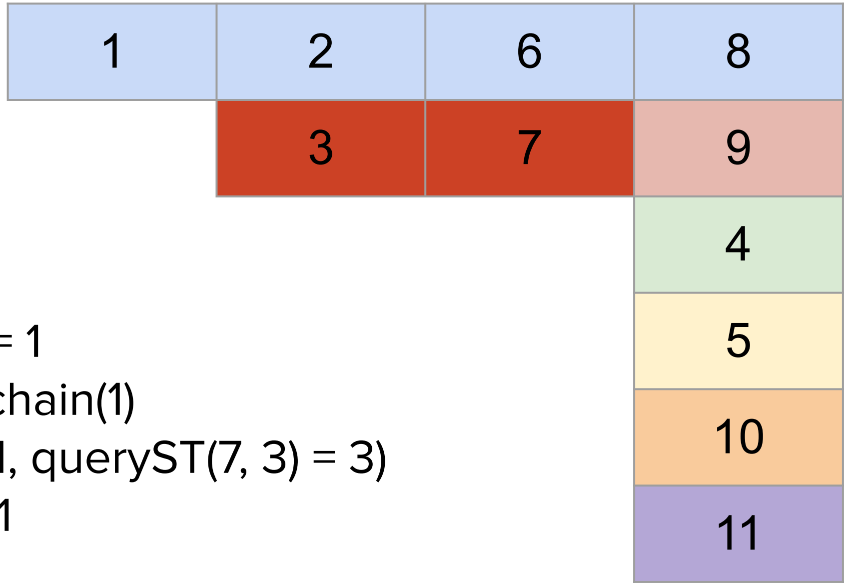
v = 8   x = 1
chain(8) == chain(1)
resp = min(10, queryST(8, 1) = 1)

resp = 1

Exemplo: valor mínimo entre vértices



Segment Trees para os “arrays”:



v = 7   x = 1

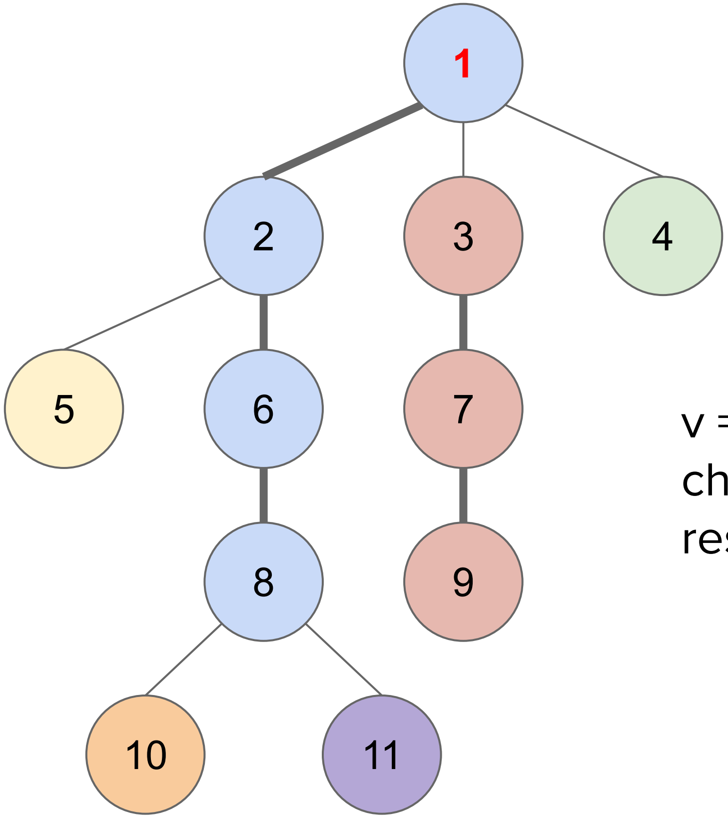
chain(7) != chain(1)

resp = min(1, queryST(7, 3) = 3)

v = pai[3] = 1

resp = 1

Exemplo: valor mínimo entre vértices



Segment Trees para os “arrays”:

1	2	6	8
	3	7	9
			4
			5
			10
			11

v = 1   x = 1

chain(1) == chain(1)

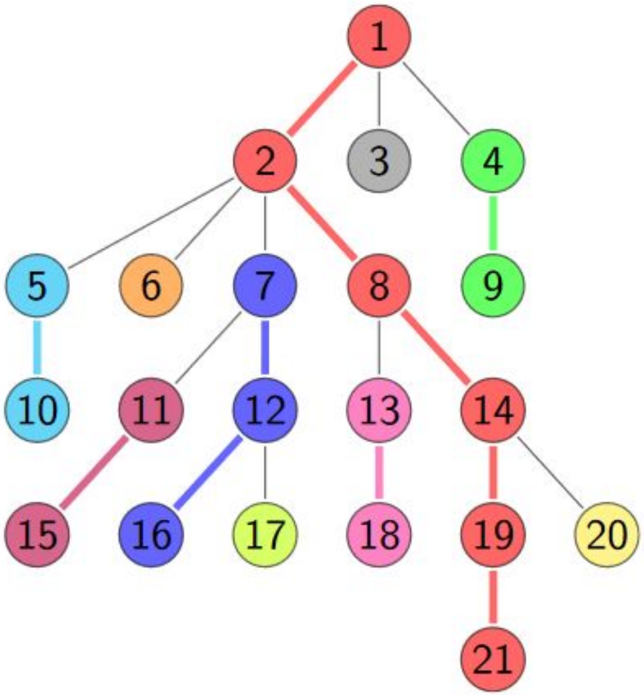
resp = min(1, queryST(1, 1) = 1)

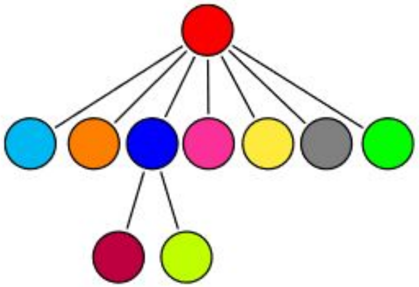
resp = 1

Simplificações para implementação

- Certas partes da abordagem discutida podem ser modificadas para facilitar a implementação sem perder a eficiência:
  - Definição de **aresta heavy** pode ser alterada para a **aresta que leva a maior subárvore.** Nesse caso, algumas arestas *lights*, pela definição anterior, podem ser convertidas em *heavies*.
  - Ao invés de construir uma SegTree para cada *heavy path*, **uma única SegTree** pode ser usada com segmentos separados para cada *heavy path*.
  - O cálculo do LCA pode ser feito durante a própria consulta.
    - De forma semelhante a *binary lift*, subimos por cada *heavy light* em busca do ancestral comum.

Simplificações para implementação





1 2 8 14 19 21 20 13 18 5 10 6 7 12 16 17 11 15 3 4 9

Implementação

```
vector<int> parent, depth, heavy, head, pos;
int cur_pos;
//Para cada nó determina seu nível e o filho com maior subárvore
int dfs(int v, vector<vector<int>> const& adj) {
    int size = 1, max_c_size = 0;
    for (int c : adj[v]) {
        if (c != parent[v]) {
            parent[c] = v, depth[c] = depth[v] + 1;
            int c_size = dfs(c, adj);
            size += c_size;
            if (c_size > max_c_size)
                max_c_size = c_size, heavy[v] = c;
        }
    }
    return size;
}
```

Implementação

```
int decompose(int v, int h, vector<vector<int>> const& adj) {
    head[v] = h, pos[v] = cur_pos++;
    if (heavy[v] != -1)
        decompose(heavy[v], h, adj);
    for (int c : adj[v]) {
        if (c != parent[v] && c != heavy[v])
            decompose(c, c, adj);
    }
}
```

Implementação

```
void init(vector<vector<int>> const& adj) {
    int n = adj.size();
    parent = vector<int>(n);
    depth = vector<int>(n);
    heavy = vector<int>(n, -1);
    head = vector<int>(n);
    pos = vector<int>(n);
    cur_pos = 0;

    dfs(0, adj);
    decompose(0, 0, adj);
}
```

Implementação

```
int query(int a, int b) {
    int res = INF;
    for (; head[a] != head[b]; b = parent[head[b]]) {
        if (depth[head[a]] > depth[head[b]])
            swap(a, b);
        int cur_min = segment_tree_query(pos[head[b]], pos[b]);
        res = min(res, cur_min);
    }
    if (depth[a] > depth[b])
        swap(a, b);
    int last_min = segment_tree_query(pos[a], pos[b]);
    res = min(res, last_min);
    return res;
}
```

Referências

https://cp-algorithms-brasil.com/grafos/heavylight.html

https://www.geeksforgeeks.org/heavy-light-decomposition-set-1-introduction/

https://homepages.dcc.ufmg.br/~monteirobruno/Slides_HLD