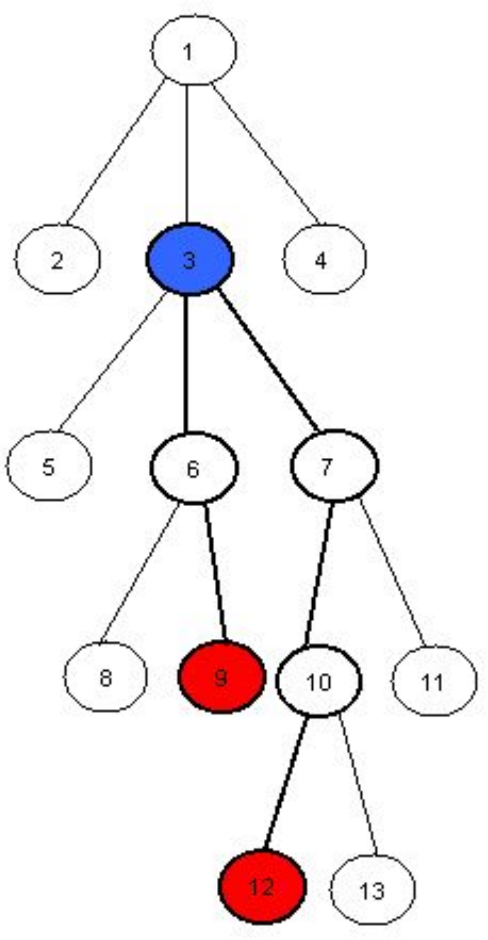
Lowest Common Ancestor (LCA)

Laboratório de Programação Competitiva - 2020

Pedro Henrique Paiola

Menor Ancestral Comum

- O problema do Menor Ancestral Comum (LCA) consiste em, dados dois nós **u** e **v**, determinar qual o nó mais baixo (relativo a raiz) que é ancestral de **u** e **v**.
- Existem diversos algoritmos e técnicas para resolver este problema, com diferentes vantagens e desvantagens. Algumas só podem ser aplicadas com restrições específicas.
- Hoje veremos algumas das técnicas mais genéricas.



$LCA_T(9, 12) = 3$

Aplicações

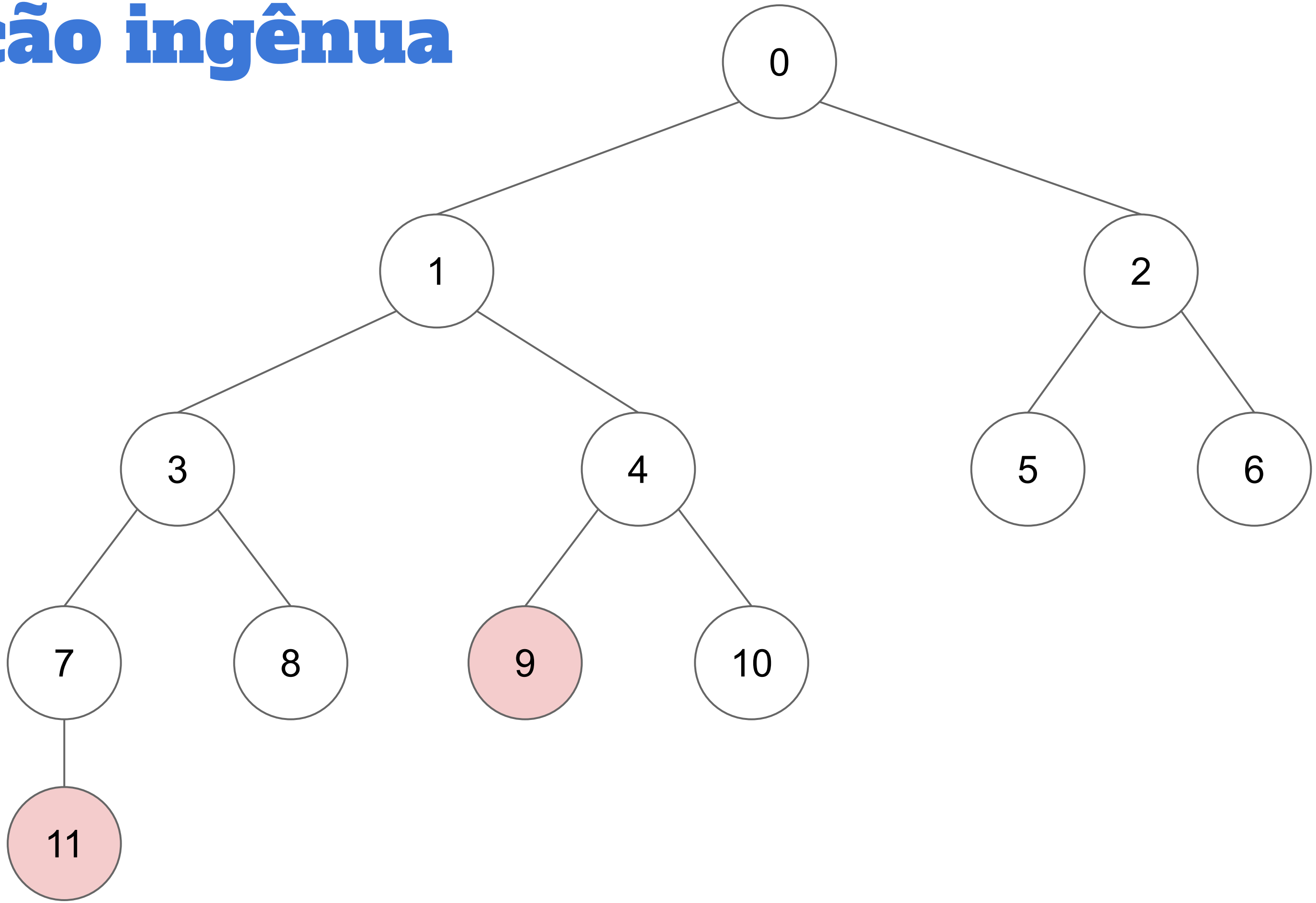
- O LCA possui diversas aplicações, e muitas vezes precisa sofrer algumas alterações para se adequar a elas. Mas este não será nosso foco nesta aula.
- Um exemplo de aplicação simples e imediata:
  - Descobrir a distância entre dois nós **u** e **v** da árvore:
    - depth[u] + depth[v] - 2*depth[lca(u,v)]
- Aplicações com Suffix Tree
  - Encontrar todos os palíndromos maximais/repetições encadeadas de um texto.
  - Buscas de padrões em textos admitindo erros (Approximate string matching).

Solução ingênua

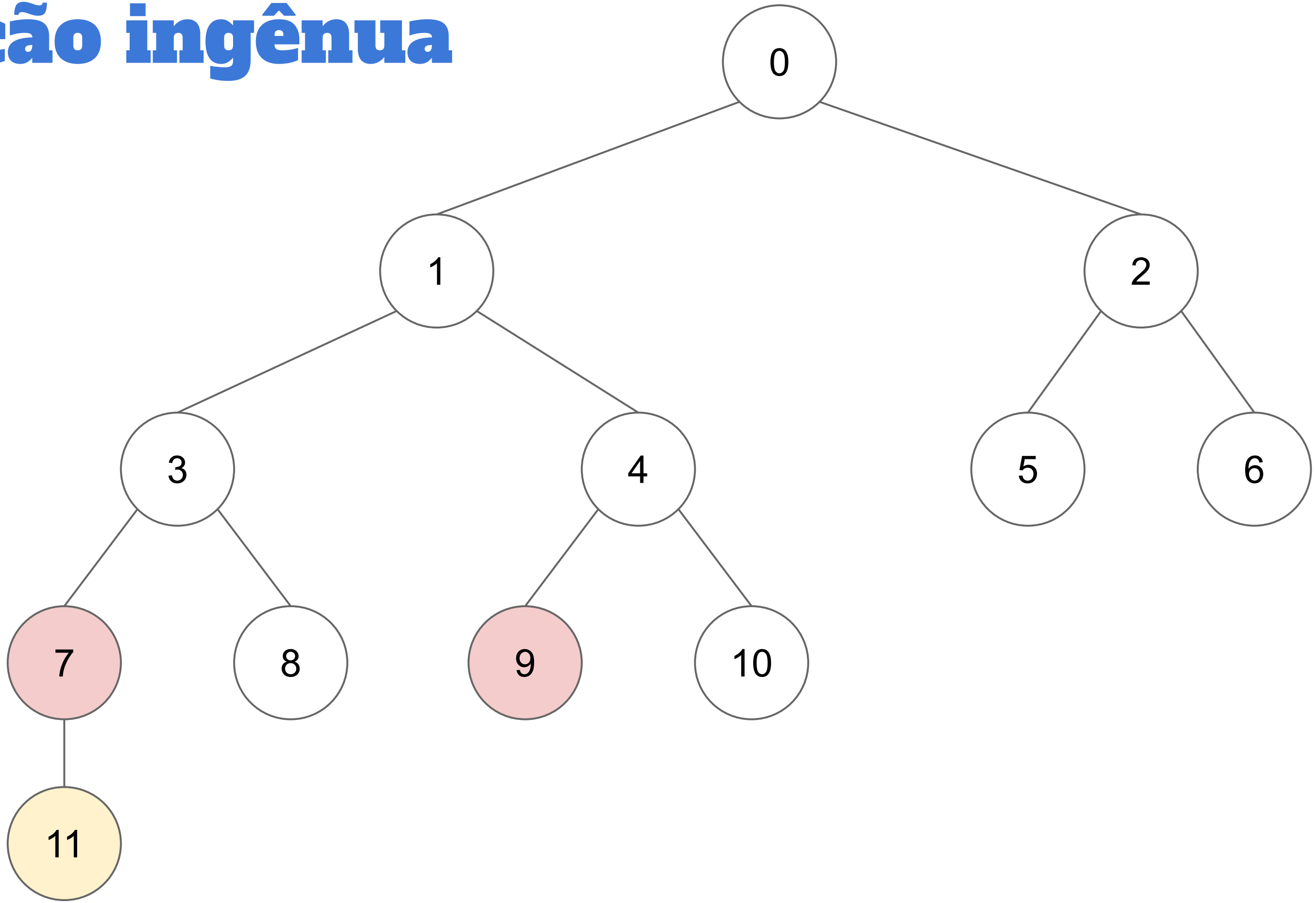
- A primeira solução que podemos pensar é ir subindo pela árvore a partir dos dois nós, até que eles se encontrem em um nó comum.
- Pré-processamento: O(n)
- Consulta: O(n)

```
while(depth[u] > depth[v])
    u = pai[u];
while(u != v){
    u = pai[u];
    v = pai[v];
}
```

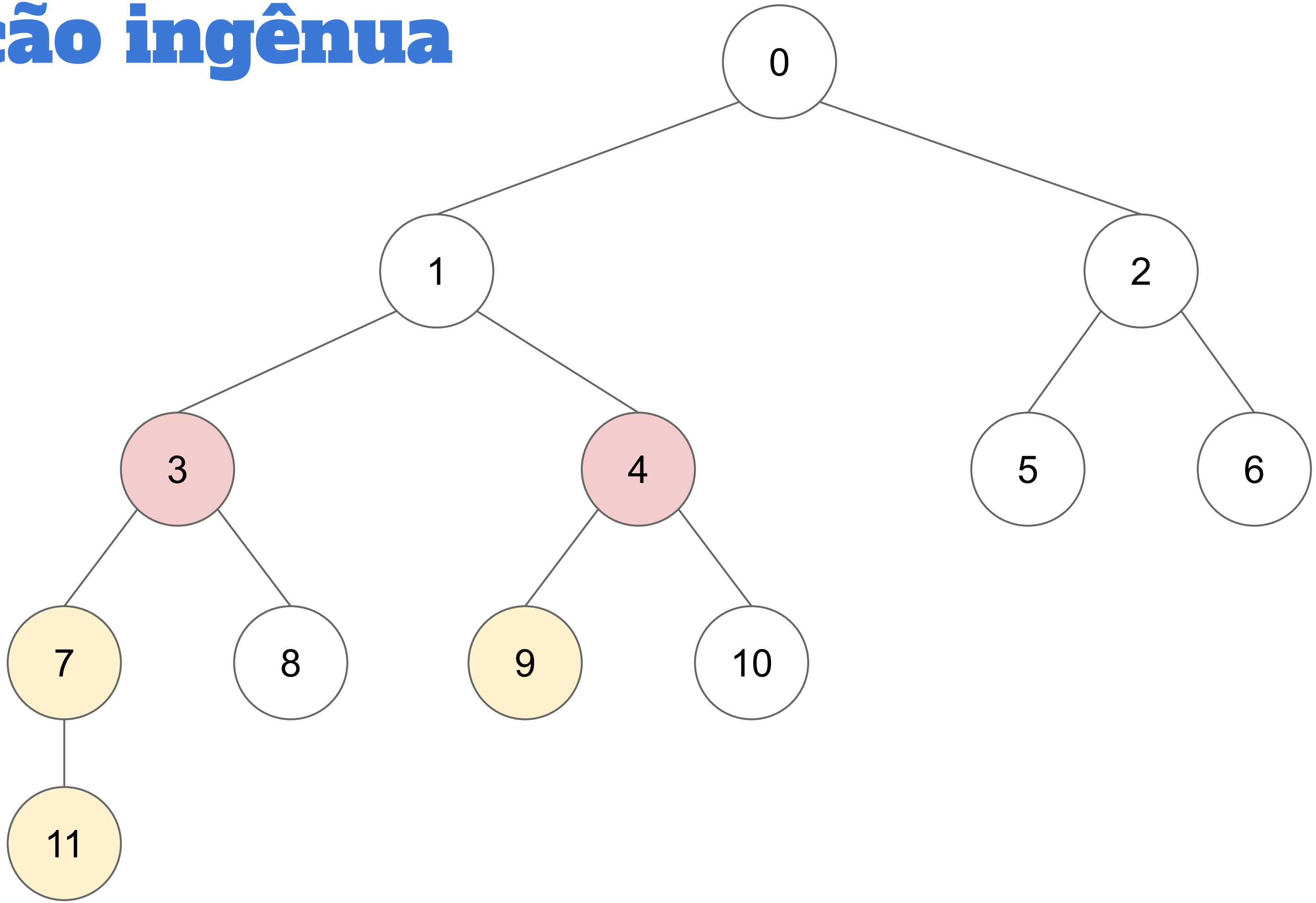
Solução ingênua



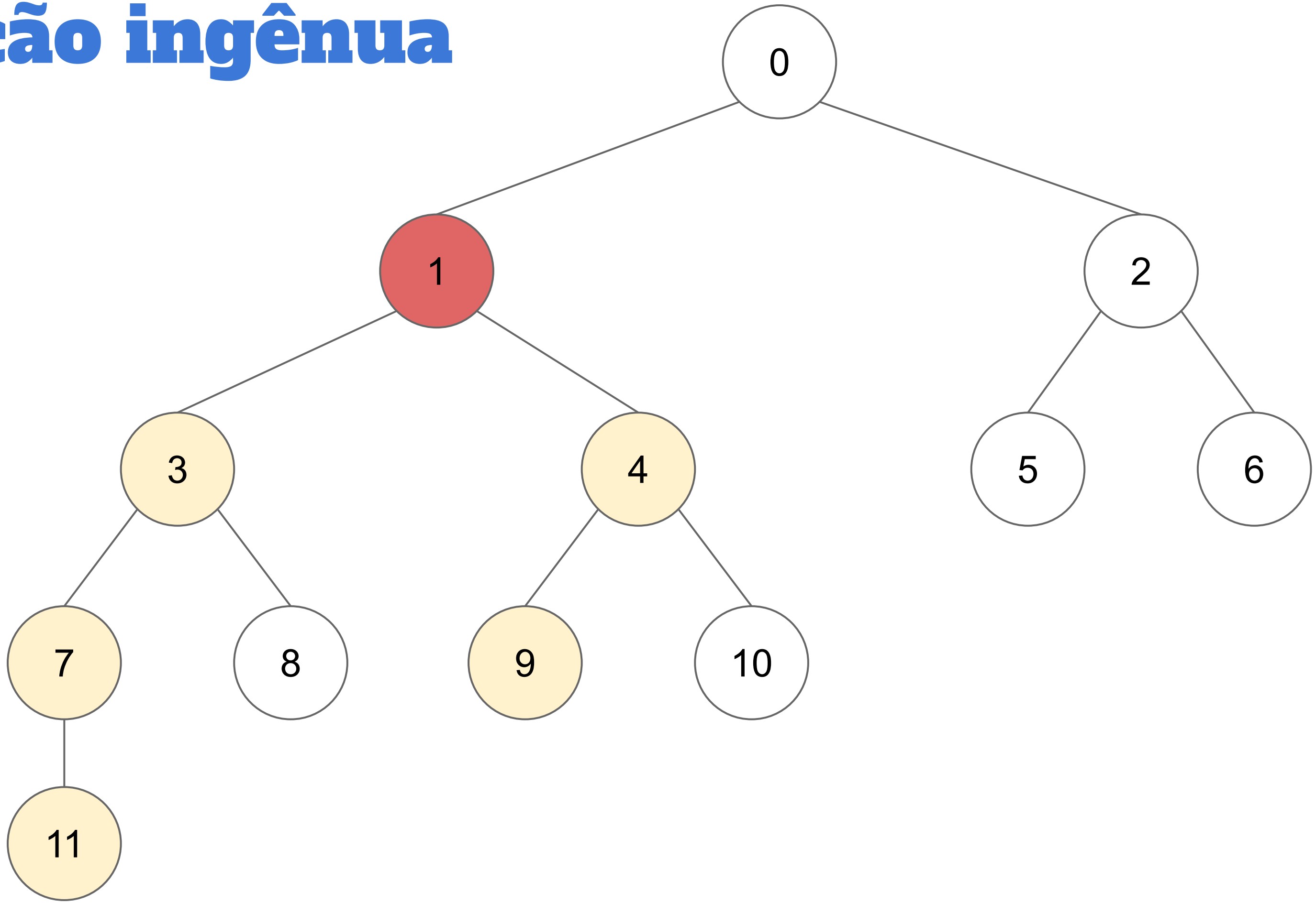
Solução ingênua



Solução ingênua



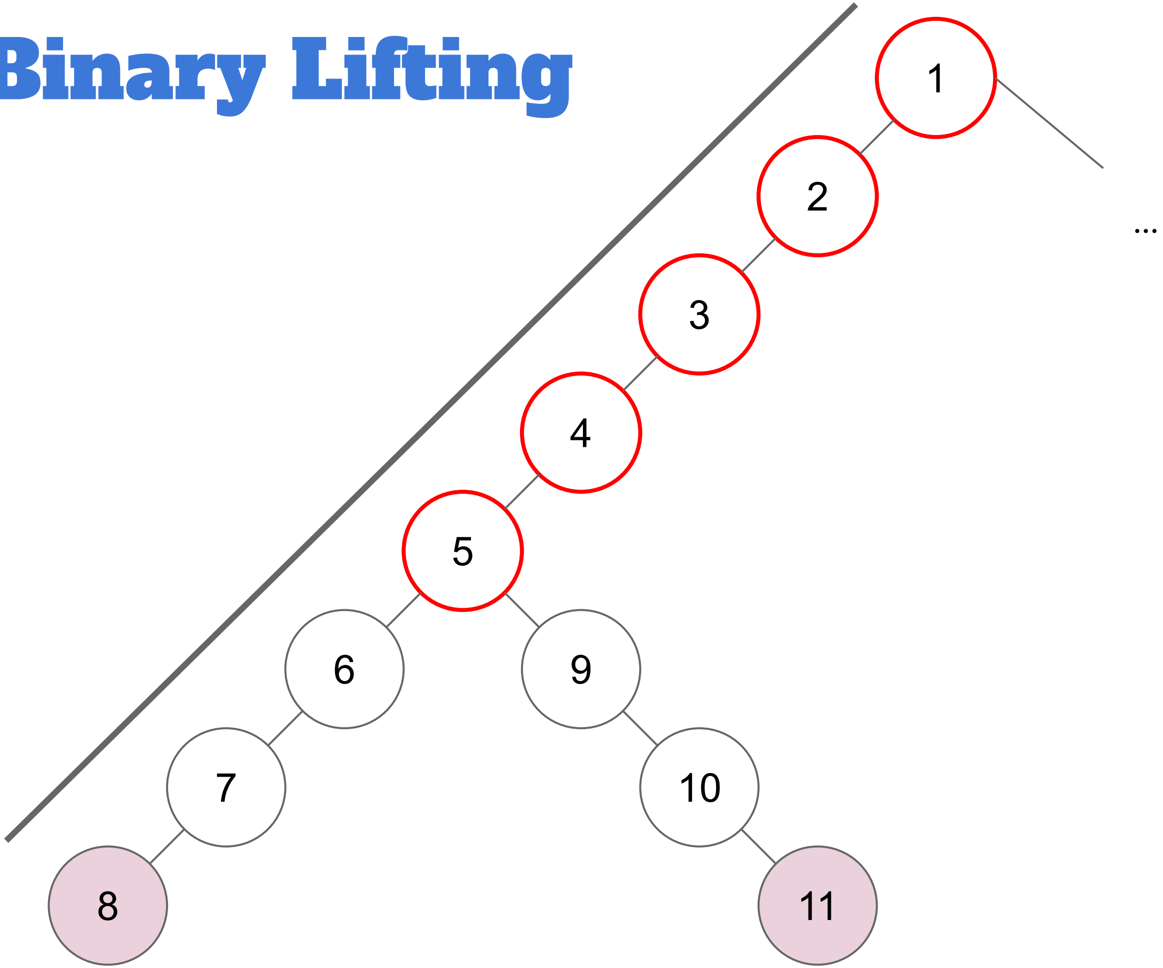
Solução ingênua



Binary Lifting

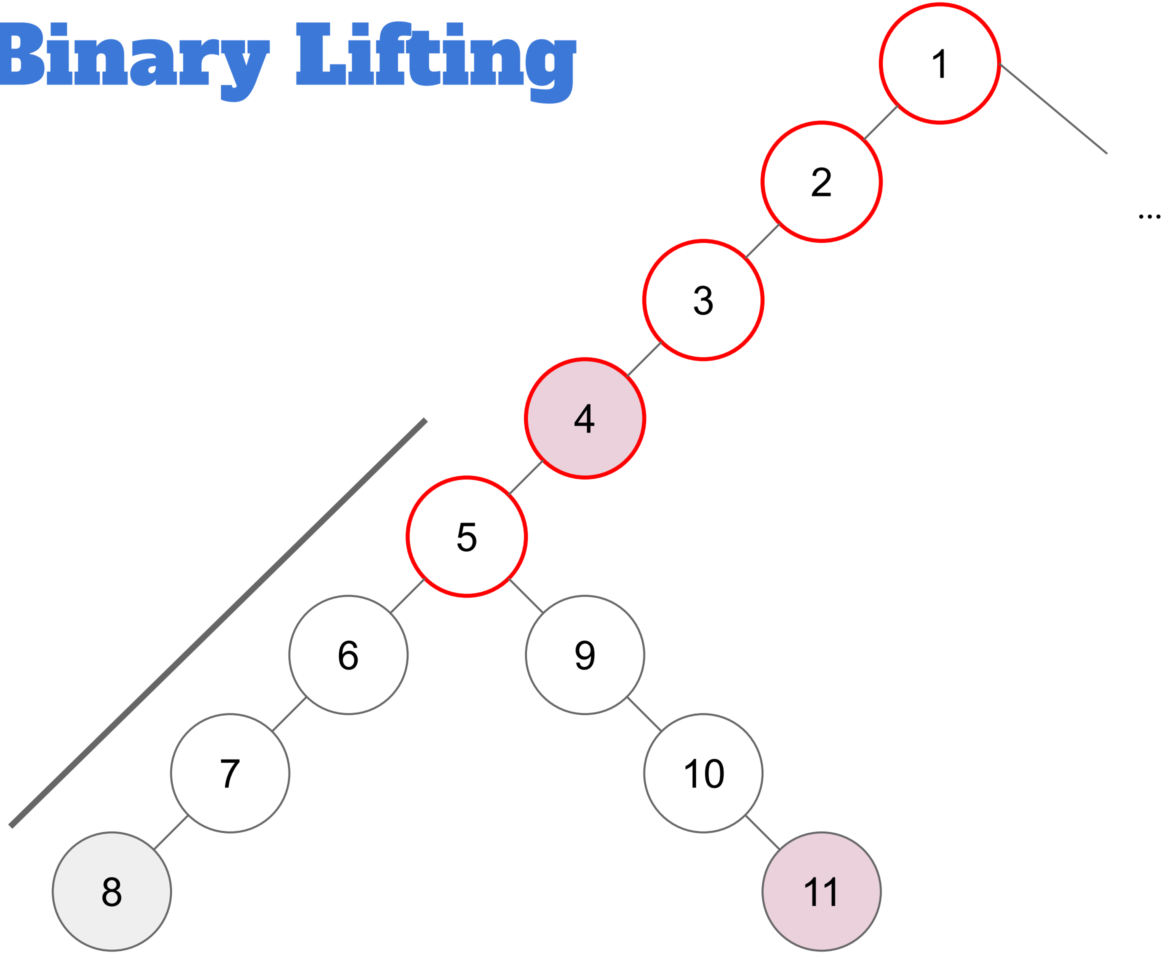
- Na solução ingênua, é como se estivessemos fazendo uma busca linear pelo primeiro ancestral comum, subindo sempre para o próximo ancestral, para o pai do nó.
- Imagine que tivéssemos uma função “mágica” climb(u, k), que nos retornasse o **k**-ésimo pai, ou **k**-ésimo ancestral, de **u**. Poderíamos aplicar uma busca binária para descobrir o ancestral que nos interessa.
- Vamos supor que temos essa função (na prática veremos que não é bem assim), e então podemos fazer a busca pelo último ancestral de **u** que não é ancestral de **v** (logo, o pai desse ancestral é o primeiro ancestral comum).

Binary Lifting



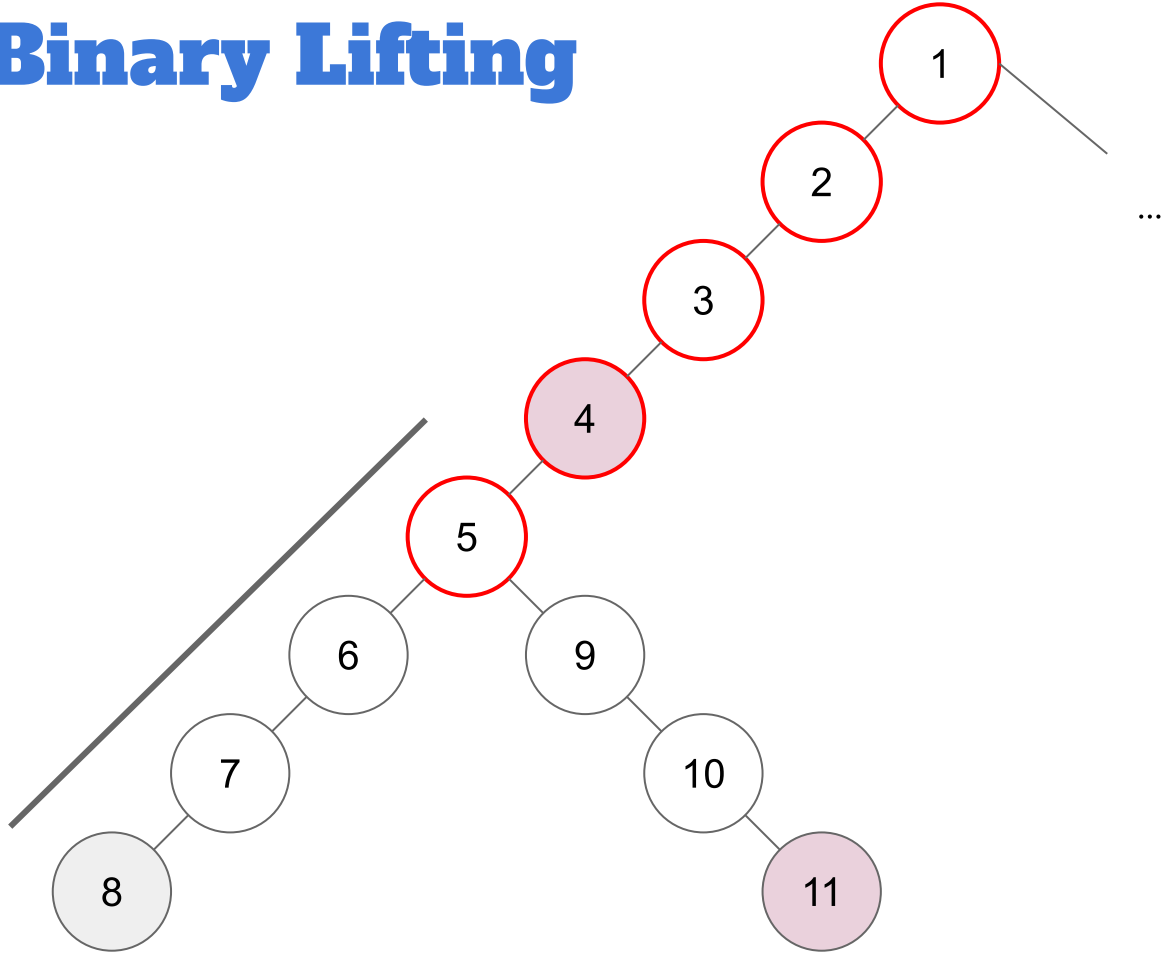
esq = 1
dir = 8
meio = 4
resp =

Binary Lifting



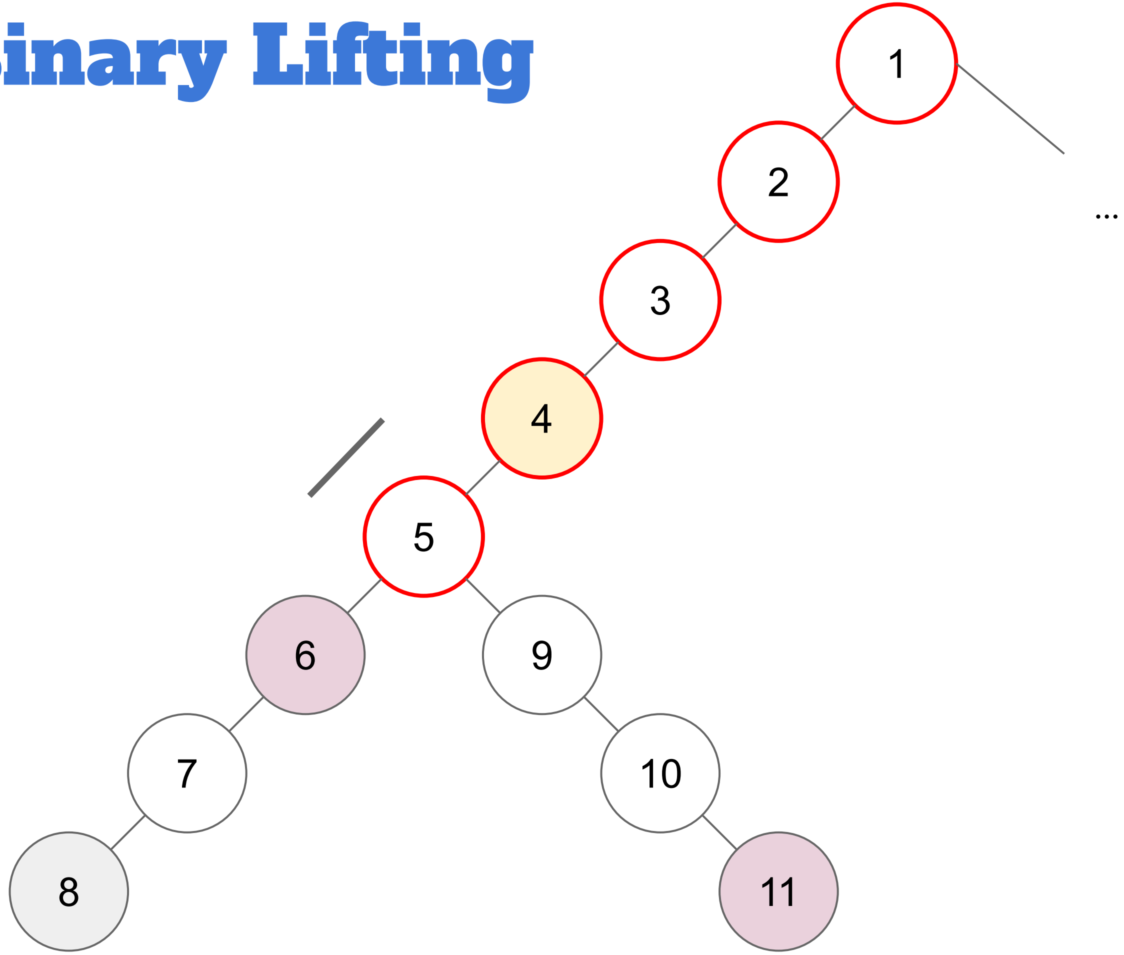
esq = 1
dir = 8
meio = 4
resp =

Binary Lifting



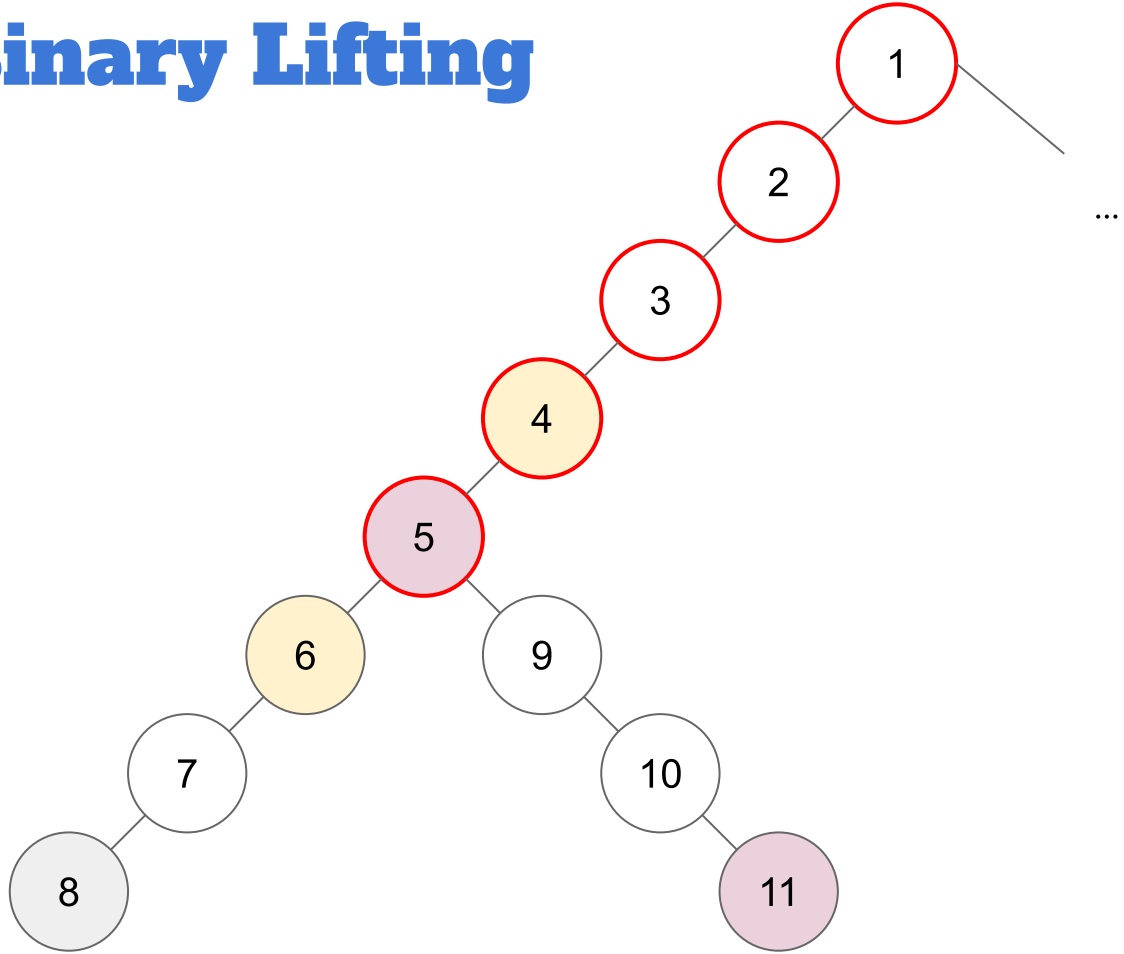
esq = 5
dir = 8
meio = 6
resp =

Binary Lifting



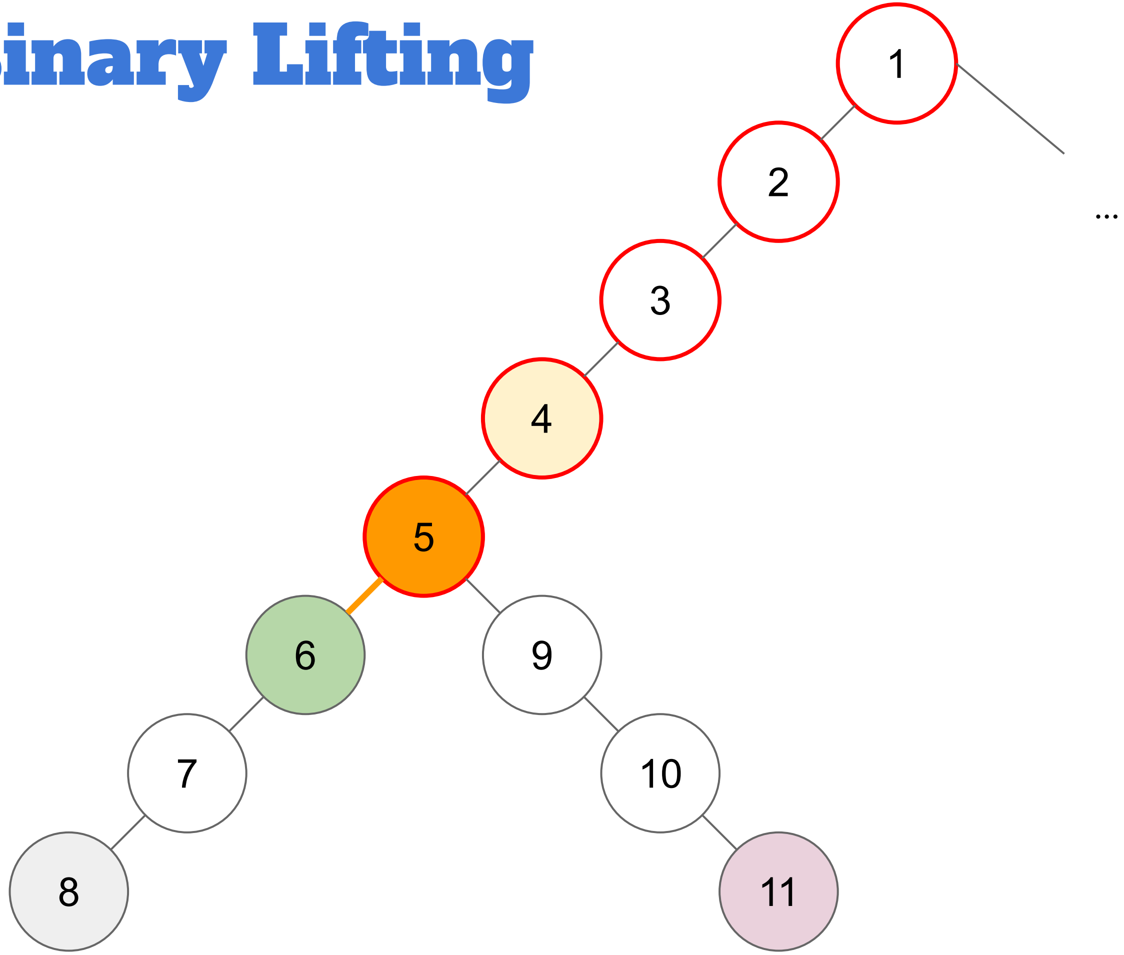
esq = 5
dir = 8
meio = 6
resp = 6

Binary Lifting



esq = 5
dir = 5
meio = 5
resp = 6

Binary Lifting



resp = 6
lca = pai[resp] = 5

Binary Lifting

- Esta é a ideia base, mas na prática, para obtermos essa função "mágica" climb(u, k) teríamos uma complexidade no pré-processamento muito alta, tanto em termos de memória como tempo: $O(n^2)$
- Então, criaremos uma matriz que, em uma certa posição **pai[i][j]**, armazenaremos o $\mathbf{2^j}$-ésimo pai do nó **i**.
- A partir desta estrutura, utilizaremos uma técnica chamada **binary lifting** (escalada binária), que é tem função bastante similar a busca binária.

Binary Lifting

- Para pré-processar a matriz **pai** usamos programação dinâmica, baseada na seguinte relação de recorrência

```
pai(u,0) = p[u]
pai(u,k) = pai(pai(u,k-1),k-1)
```

O 2^k pai de u é o 2^{k-1} pai do 2^{k-1} pai de u

$2^k = 2^{k-1} + 2^{k-1} = 2.2^{k-1} = 2^k$

Binary Lifting

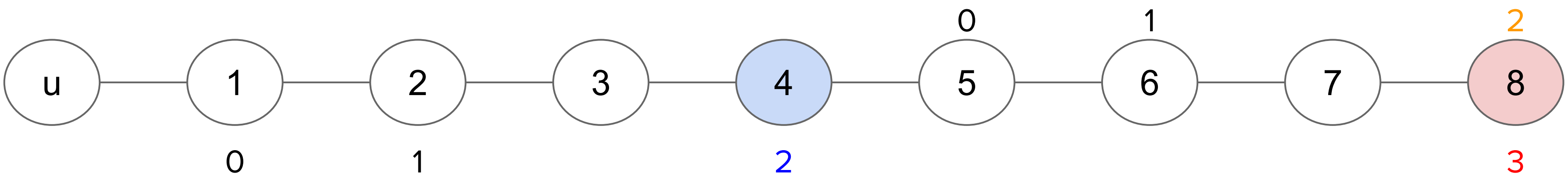
- Para pré-processar a matriz **pai** usamos programação dinâmica, baseada na seguinte relação de recorrência

```
pai(u,0) = p[u]
pai(u,k) = pai(pai(u,k-1),k-1)
```

O 2^k pai de u é o 2^{k-1} pai do 2^{k-1} pai de u

$2^k = 2^{k-1} + 2^{k-1} = 2.2^{k-1} = 2^k$

Exemplo: O 8º pai de u é o 4º pai do 4º pai de u



Binary Lifting

```
void preprocess(int u, int p){
    pai[u][0] = p;
    for(int i = 1; i <= LOGMAXN; i++){
        pai[u][i] = pai[pai[u][i-1]][i-1];
    }
    for(auto adj: tree[u]){
        depth[adj] = depth[u] + 1;
        preprocess(adj, u);
    }
}
//Complexidade: O(n.logn)
```

Binary Lifting

- Agora para consultar, vamos fazer um processo muito semelhante ao algoritmo ingênuo, mas utilizando a técnica de escalada binária para aumentar a eficiência.
- A ideia básica é, começando k com o maior número possível (log2(N)), verificamos se o 2^k-ésimo ancestral de **u** é igual o 2^k-ésimo ancestral de **v**. Se **não** for, subimos u e v para estes ancestrais.
- Com isso, no final iremos encontrar os últimos nós que não são ancestrais comuns, ou seja, pai[u][0] = pai[v][0] = lca.

Binary Lifting

```
int lca(int u, int v){
    if (depth[u] < depth[v])
        swap(u, v);

    for (int i = LOGMAXN; i >= 0; i--)
        if ((depth[u] - (1 << i)) >= depth[v])
            u = pai[u][i];

    if (u == v)
        return u;
```

Binary Lifting

```
    for (int i = LOGMAXN; i >= 0; i--){
        if (pai[u][i] != pai[v][i]){
            u = pai[u][i];
            v = pai[v][i];
        }
    }

    return pai[u][0];
}
//Complexidade: O(log n)
```

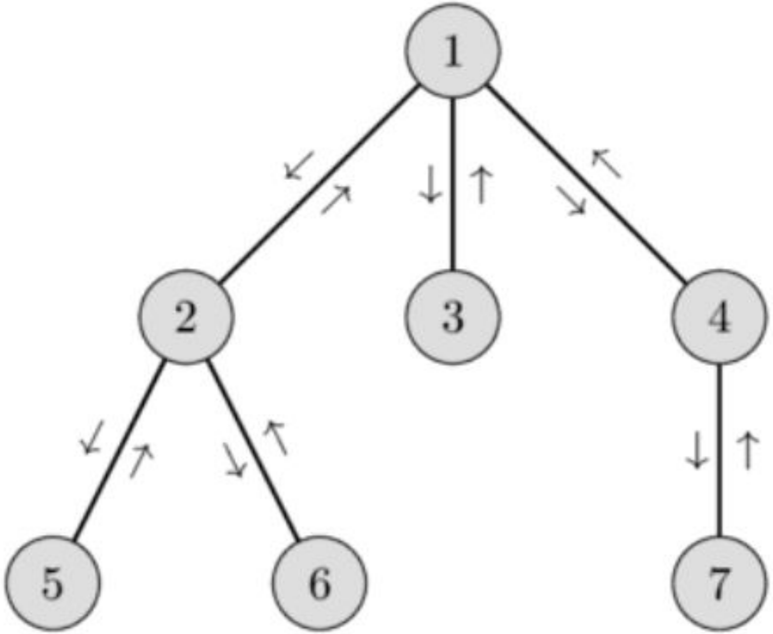
Redução para RMQ

- Outra técnica possível é reduzir o LCA para o problema da RMQ (*Range Minimum Query*).
- A grande vantagem é que conhecemos uma técnica de resolução para este problema que realiza consultas em O(1).
    - Sparse Table
- Porém, perdemos um pouco de versatilidade. Com a Binary Lifting podemos utilizar a DP para armazenar mais informações além de qual o 2^k-ésimo pai, por exemplo: a aresta mínima(ou máxima) do caminho, a soma dos custos das arestas, máximo divisor comum, ...

Redução para RMQ

- Para isso, vamos pré-processar nossa árvore fazendo um tour de Euler.
- No tour de Euler visitamos toda a árvore por uma DFS e adicionamos os vértices em uma lista de visitação quando visitados o nó pela primeira vez E quando passamos por ele no retorno da travessia
- Para cada nó vamos armazenar também sua altura/nível e marcar a primeira vez por qual passamos por ele

Redução para RMQ



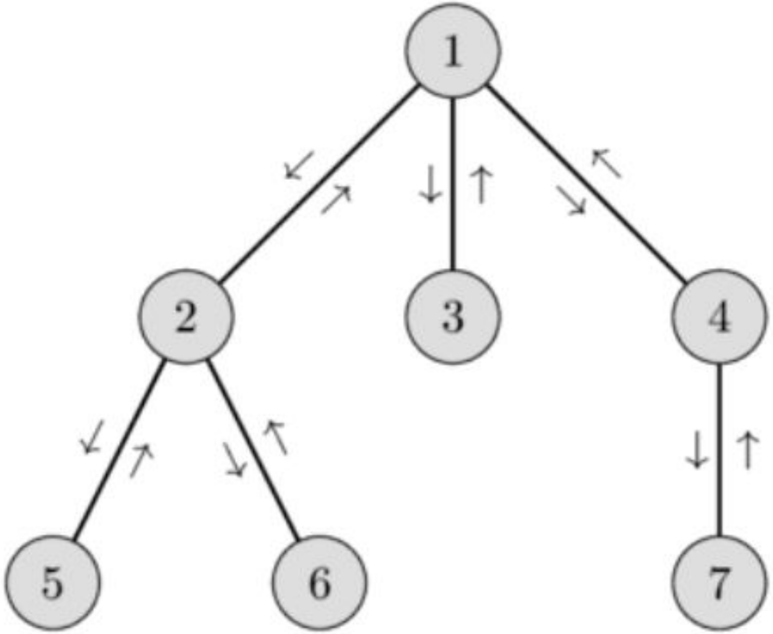
	1	2	3	4	5	6	7
dep	0	1	1	1	2	2	2
first	0	1	7	9	2	4	10

	0	1	2	3	4	5	6	7	8	9	10	11	12
euler	1	2	5	2	6	2	1	3	1	4	7	4	1

Redução para RMQ

- Desta forma forma, para dois nós **u** e **v**, o intervalo dado por [first(u), first(v)] contém todo o caminho de u até v.
- Se olharmos para as alturas dos nós desse caminho, o nó de menor altura representa o menor ancestral comum.

Redução para RMQ



	1	2	3	4	5	6	7
dep	0	1	1	1	2	2	2
first	0	1	7	9	2	4	10

	0	1	2	3	4	5	6	7	8	9	10	11	12
euler	1	2	5	2	6	2	1	3	1	4	7	4	1

Redução para RMQ

- Sendo assim, tendo feito o pré-processamento para calcular os vetores:
  - euler_tour
  - height/depth
  - first
- Basta aplicarmos uma estrutura capaz de trabalhar com range queries em euler_tour para a seguinte função:

```
int f(int x, int y){
    if (depth[x] < depth[y])
        return x;
    return y;
}
```

Redução para RMQ

- Pré-processamento:

```
void preprocess(int u, int d){
    visited[u] = 1;
    depth[u] = d;
    first[u] = sz;
    euler_tour[sz++] = u;
    for(auto adj:tree[u]){
        if(!visited[adj]){
            preprocess(adj, d+1);
            euler_tour[sz++] = u;
        }
    }
} //Complexidade: O(n) (e O(n.logn) para a Sparse Table)
```

Redução para RMQ

- Consulta (com sparse table já construída):

```
int lca(int u, int v){
    int L = first[u];
    int R = first[v];
    if (R < L)
        swap(L, R);
    return query(L, R);
}
//Complexidade: O(1) com Sparse Table
```

Referências

https://github.com/UnBalloon/programacao-competitiva/tree/master/LCA

https://cp-algorithms-brasil.com/grafos/lca4.html

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

https://www.topcoder.com/community/competitive-programming/tutorials/range-minimum-query-and-lowest-common-ancestor/#A%20O(N),%20O(sqrt(N))%20solution

https://neps.academy/lesson/199

https://cp-algorithms.com/graph/lca_binary_lifting.html

https://www.geeksforgeeks.org/lca-in-a-tree-using-binary-lifting-technique/

https://bcc.ime.usp.br/tccs/2005/daniel/poster.pdf