Segment Tree

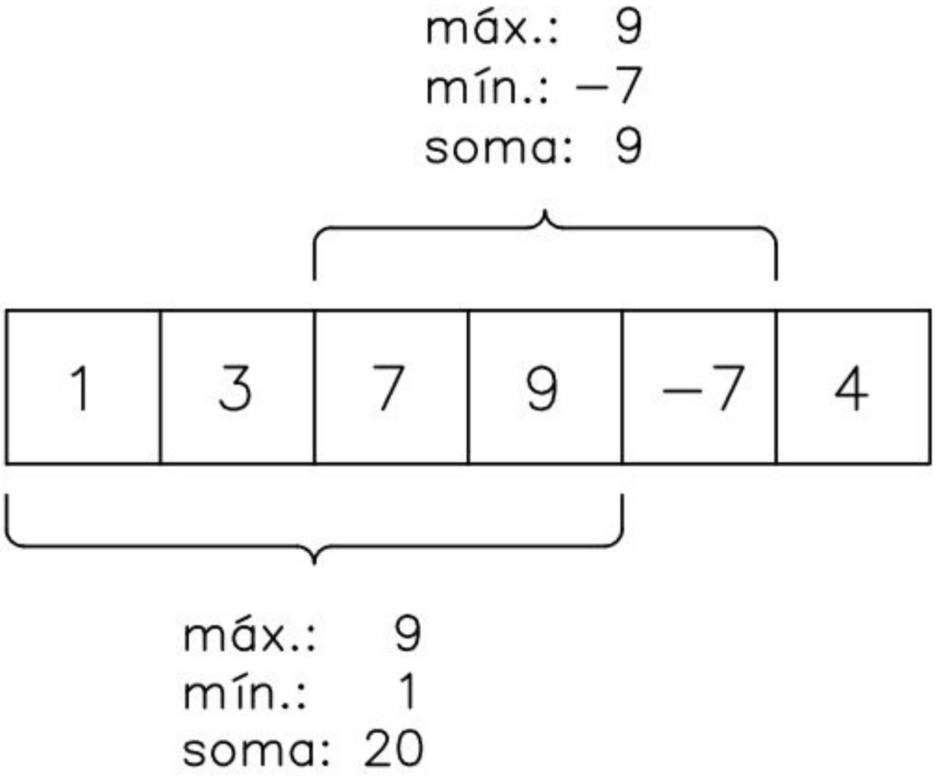
Laboratório de Programação Competitiva - 2020

Pedro Henrique Paiola
Giulia Moura Crusco
João Pedro Marin Comini

Operações em intervalos

- Diversos problemas exigem operações em intervalos, em especial, consultas em intervalos (*range queries*).
- Por força bruta, estas consultas normalmente terão complexidade O(n)
- Exemplos: Range Minimum/Maximum Query (RMQ), Range Sum Query (RSQ)

Operações em intervalos



Range Minimum Query

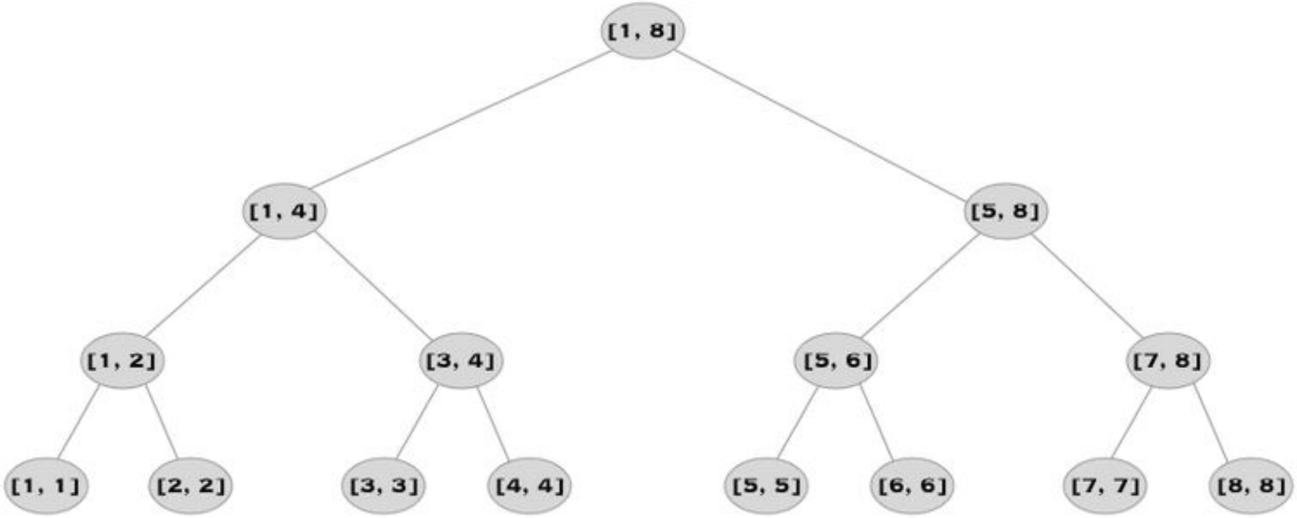
- Tomando como exemplo o problema RMQ.
- Vamos supor que temos um vetor de N elementos, em que podemos realizar uma das seguintes operações:
  - update(i, a): atualizar a posição **i** com o valor **a**
  - query(i, j): consultar o menor valor entre as posições **i** e **j**
- De forma ingênua, podemos realizar estas operações com as seguintes complexidades:
  - update: O(1)
  - query: **O(n)**

Segment Tree

- A *Segment Tree* (Árvore de Segmentos) é uma estrutura que permite fazer ambas as operações em O(log N).
- Uma SegTree é bastante versátil, e pode ser utilizada para resolver uma gama enorme de problemas envolvendo *range queries* usando-se a mesma estrutura básica.
- Porém, para cada caso teremos que fazer algumas alterações na sua implementação, por isso é importante entender exatamente como ela funciona.

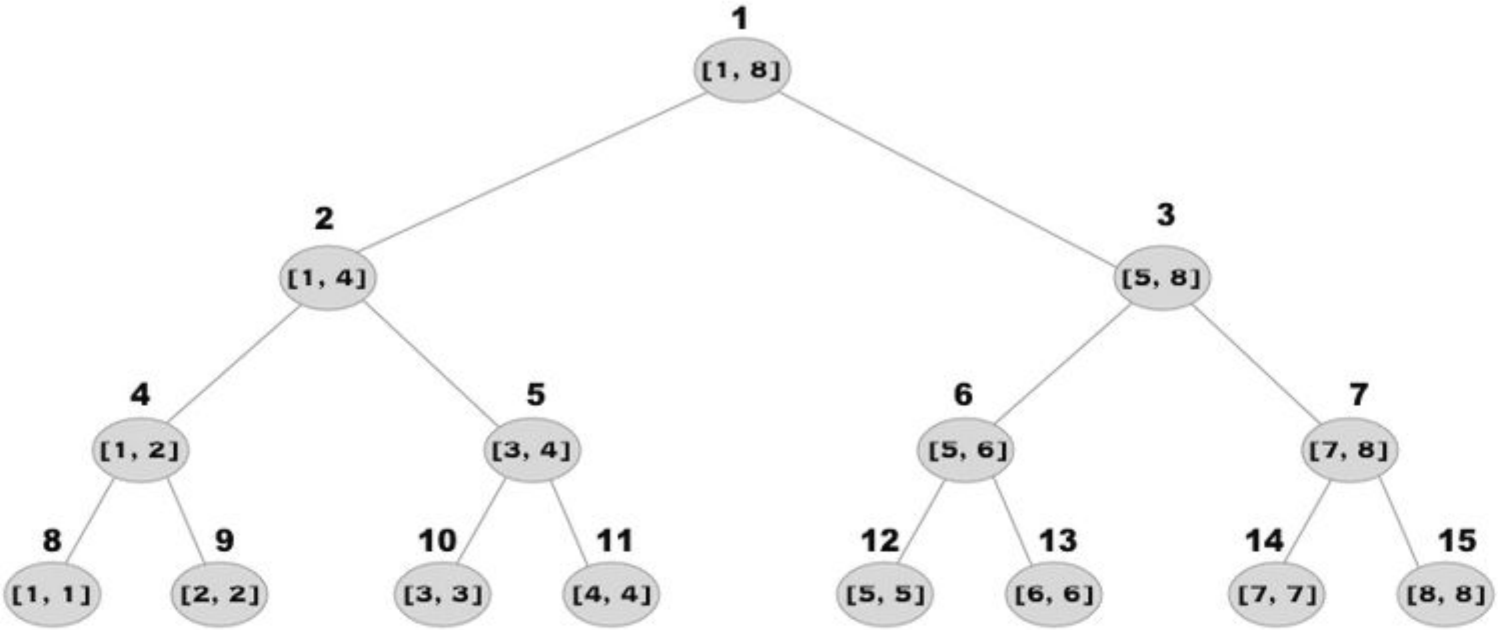
Segment Tree

- Árvore binária de consulta
- Cada nó representa um segmento de um vetor
- Os filhos de um nó que representa o segmento [i,j] serão os nós que representam os segmentos $[i, \left\lfloor \frac{i+j}{2} \right\rfloor]$ e $[\left\lfloor \frac{i+j}{2} \right\rfloor + 1, j]$



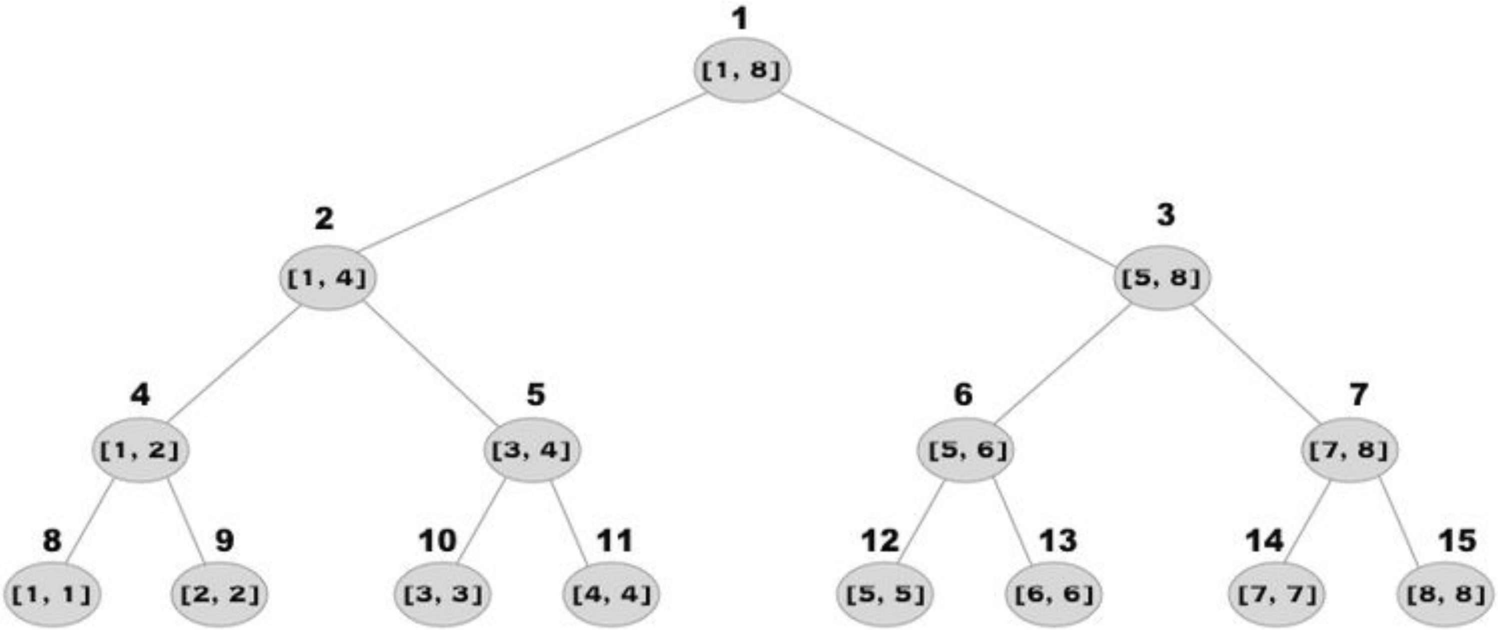
Segment Tree

- Podemos rotular cada um dos nós. Começamos rotulando a raiz como 1, e seguimos nível a nível, numerando da esquerda pra direita.



Segment Tree

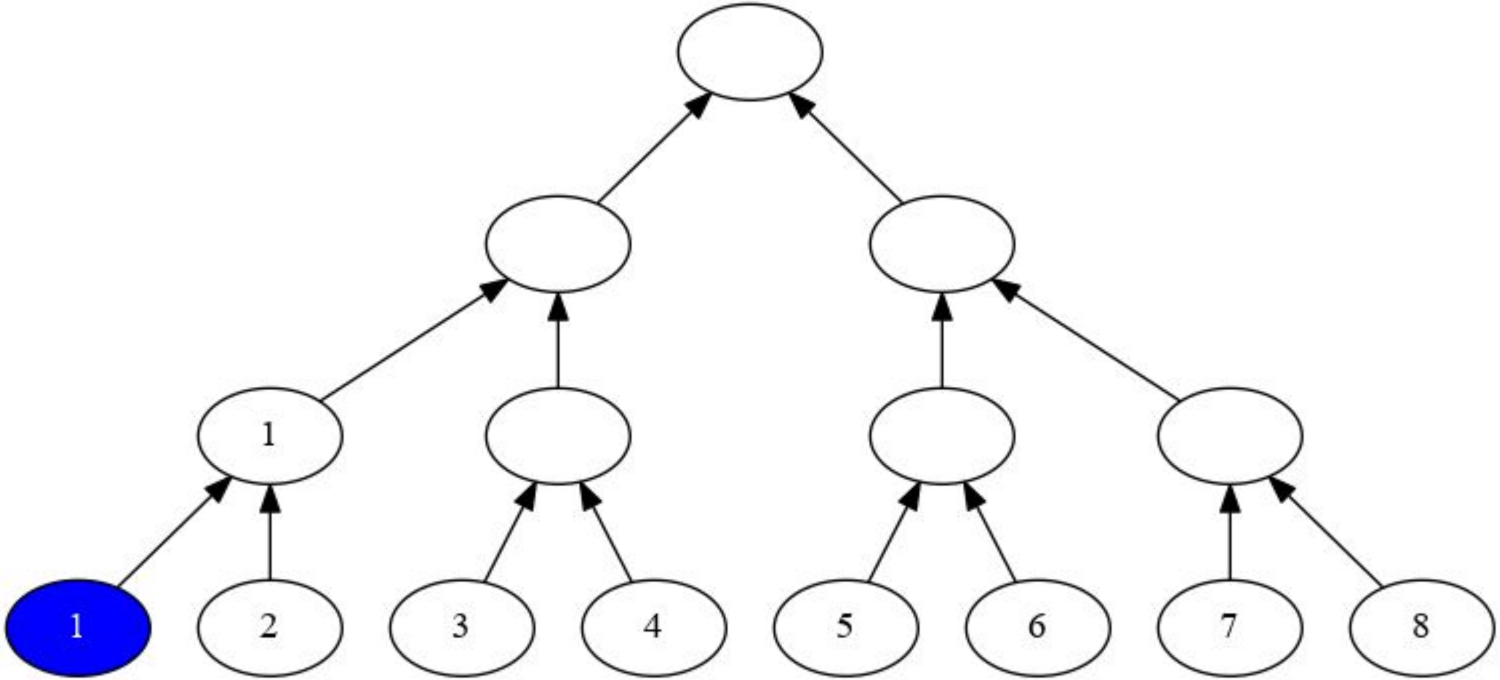
- Percebe-se que os filhos de um nó **x** são os nós **2x** e **2x + 1**



Segment Tree

- Funções básicas
  - build()
  - update()
  - query()
- Uma árvore de segmentos é bastante versátil, podemos alterar o seu uso com pequenas e intuitivas mudanças no código

Segment Tree



Representação

- Vamos considerar que temos um vetor de tamanho n chamado, criativamente, de vetor
- Para a nossa árvore de segmentos, vamos também considerar um vetor, onde cada uma posição i representa o nó i. Esse deve ter $2 * 2^{\lceil \log_2 n \rceil} - 1$ posições

```
vector<int> vetor;
vector<int> st;
int size;
```

Operação

- Como já dissemos, a SegTree é uma estrutura bastante versátil. Para tentarmos generalizar um pouco, vamos definir uma função **f** que define a informação que queremos saber a respeito dos elementos do vetor.
- Nesse caso, vamos supor uma SegTree que queira saber o mínimo de intervalos, mas poderia ser soma, máximo, produto, xor, gcd, mmc, or, and, ...

```
int f(int a, int b){
    return min(a,b);
}
```

Elemento neutro

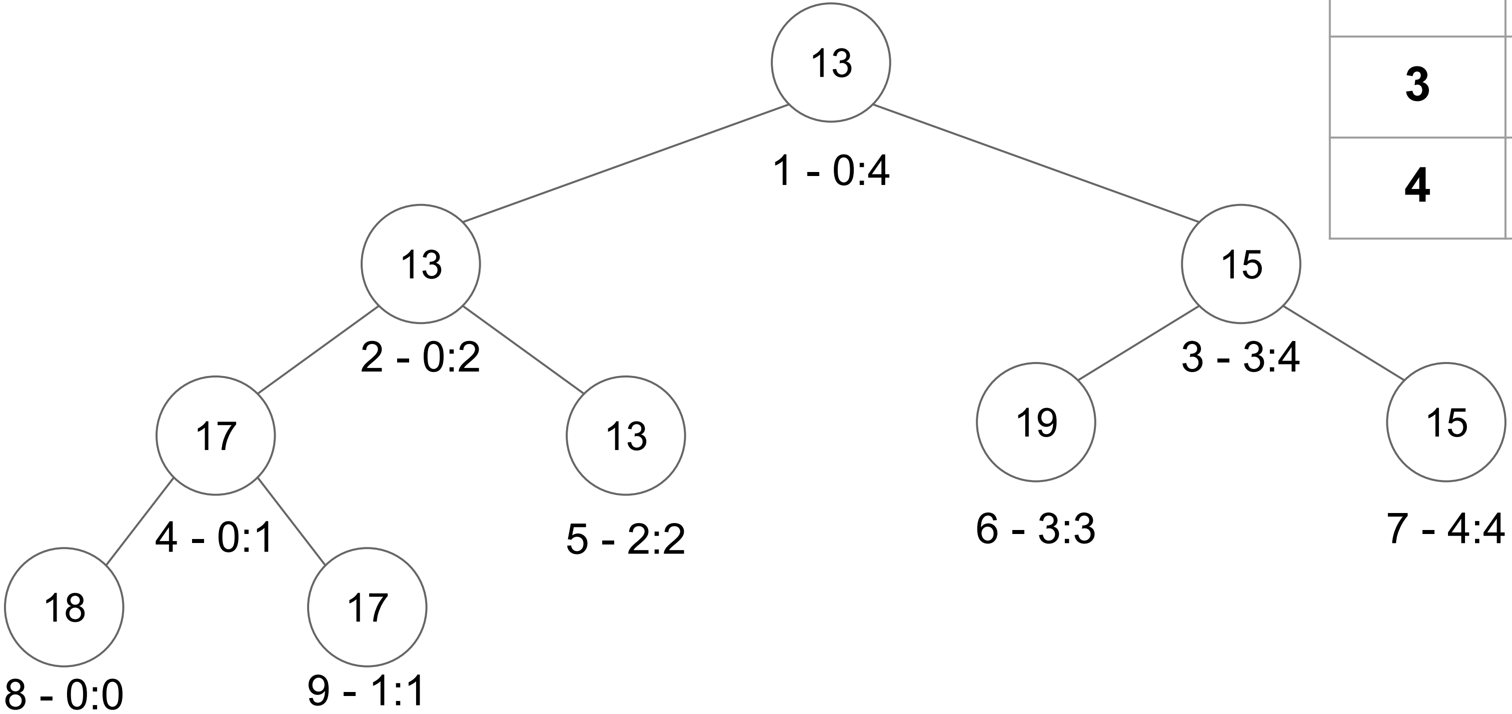
- O elemento neutro depende da operação. Como queremos saber os mínimos, o elemento neutro dessa operação seria um número muito grande.
- f(el_neutro, x) = x para todo x

```
int el_neutro = INT_MAX;
```

Atualização

- Atualizando uma posição do vetor
    - Alterando o valor de uma posição do vetor, temos que atualizar a árvore de segmentos.
    - Começaremos da raiz e iremos descendo ao longo da árvore, atualizando os vértices conforme for necessário

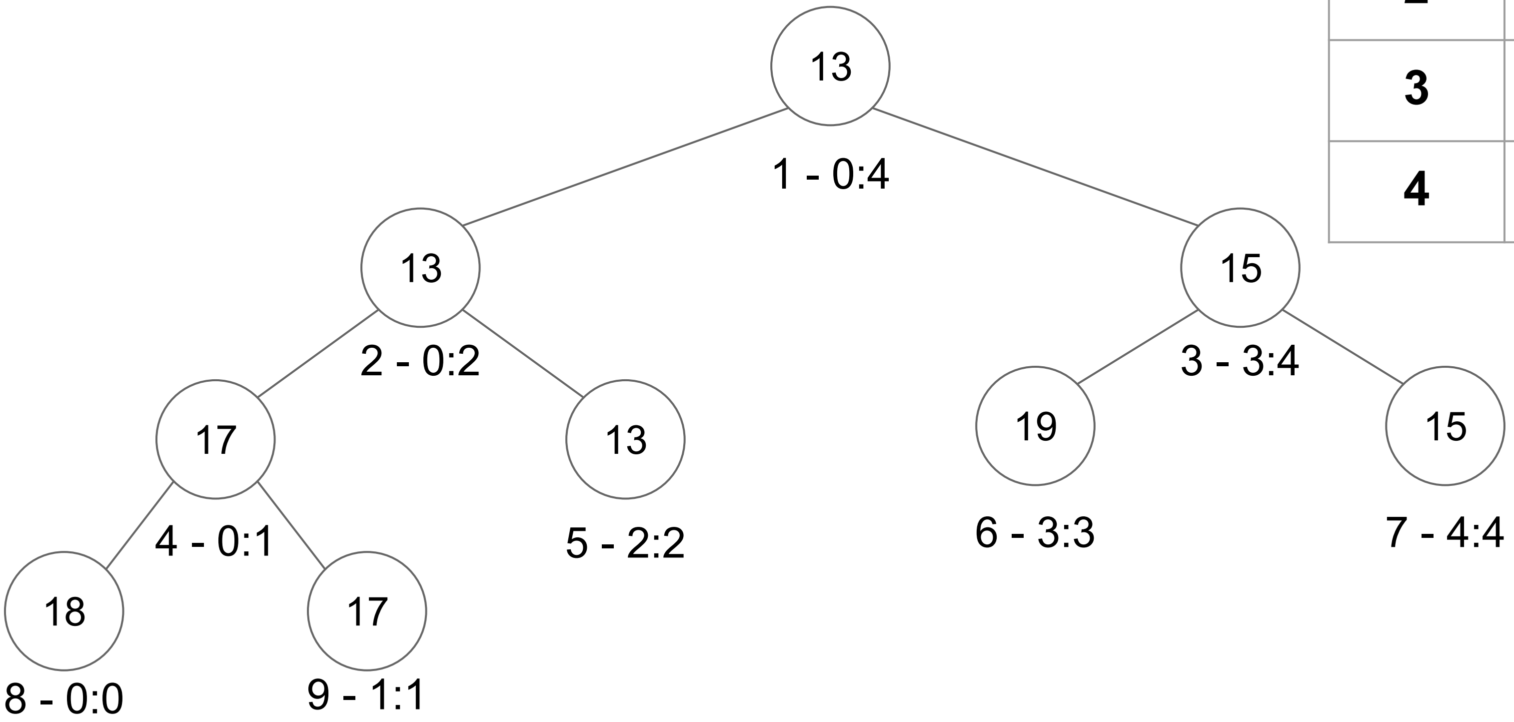
Atualização



0	18
1	17
2	13
3	19
4	15

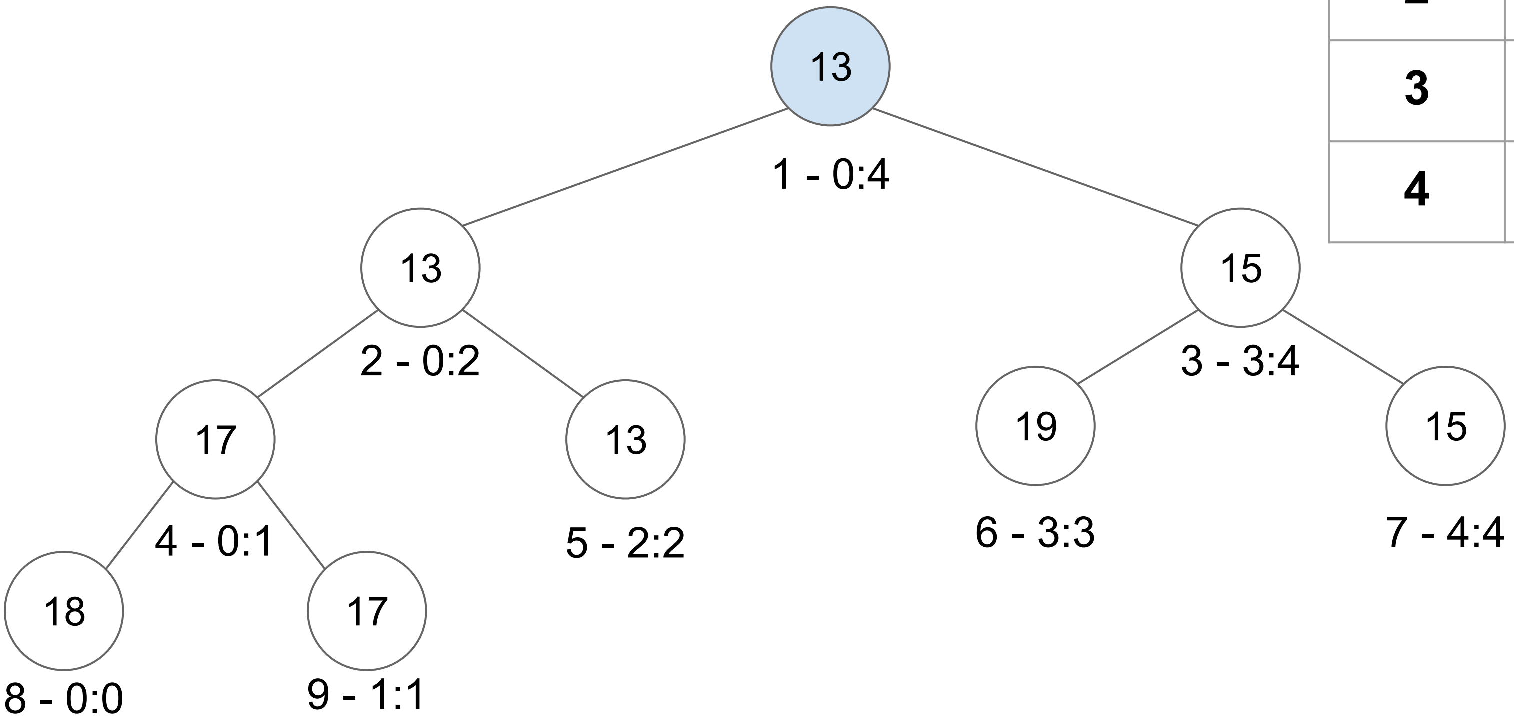
Atualização

0	18
1	12
2	13
3	19
4	15

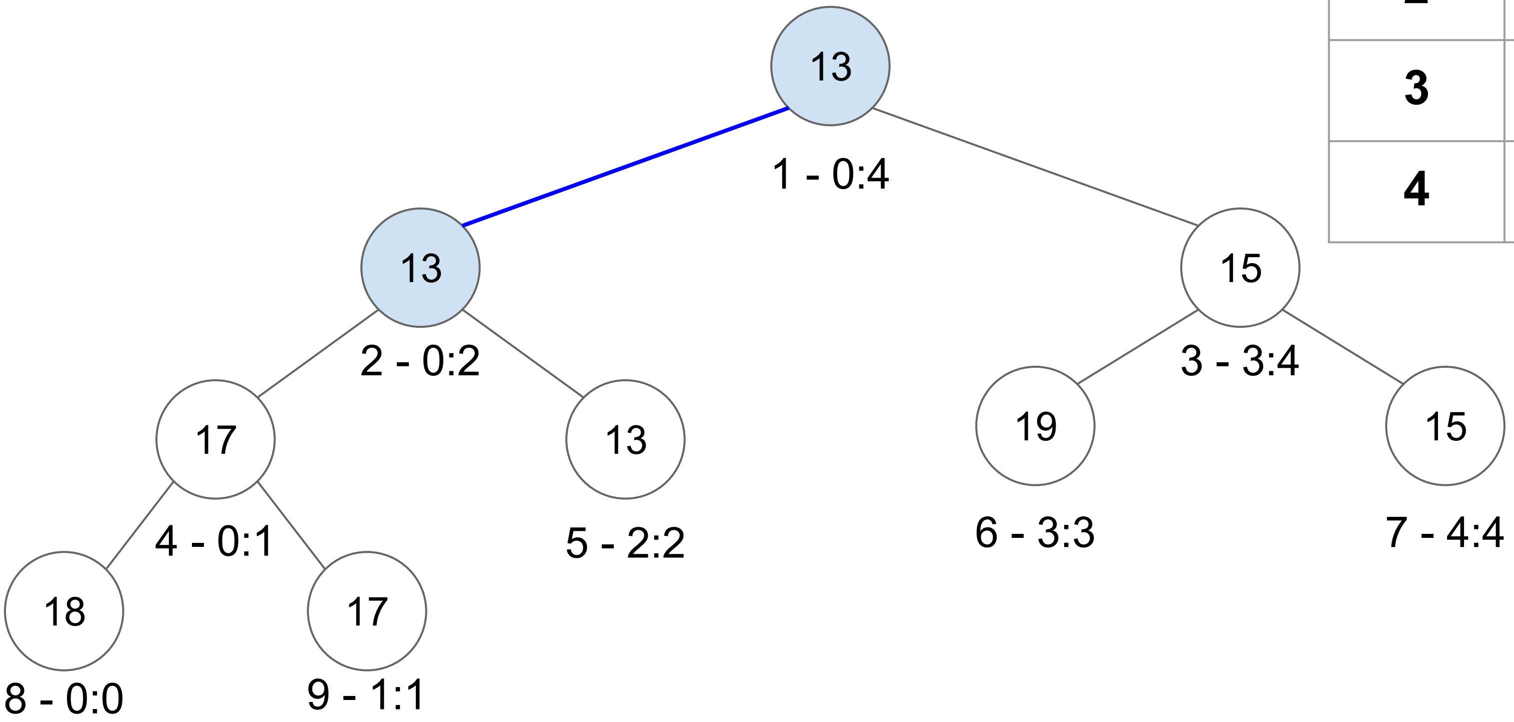


Atualização

0	18
1	12
2	13
3	19
4	15



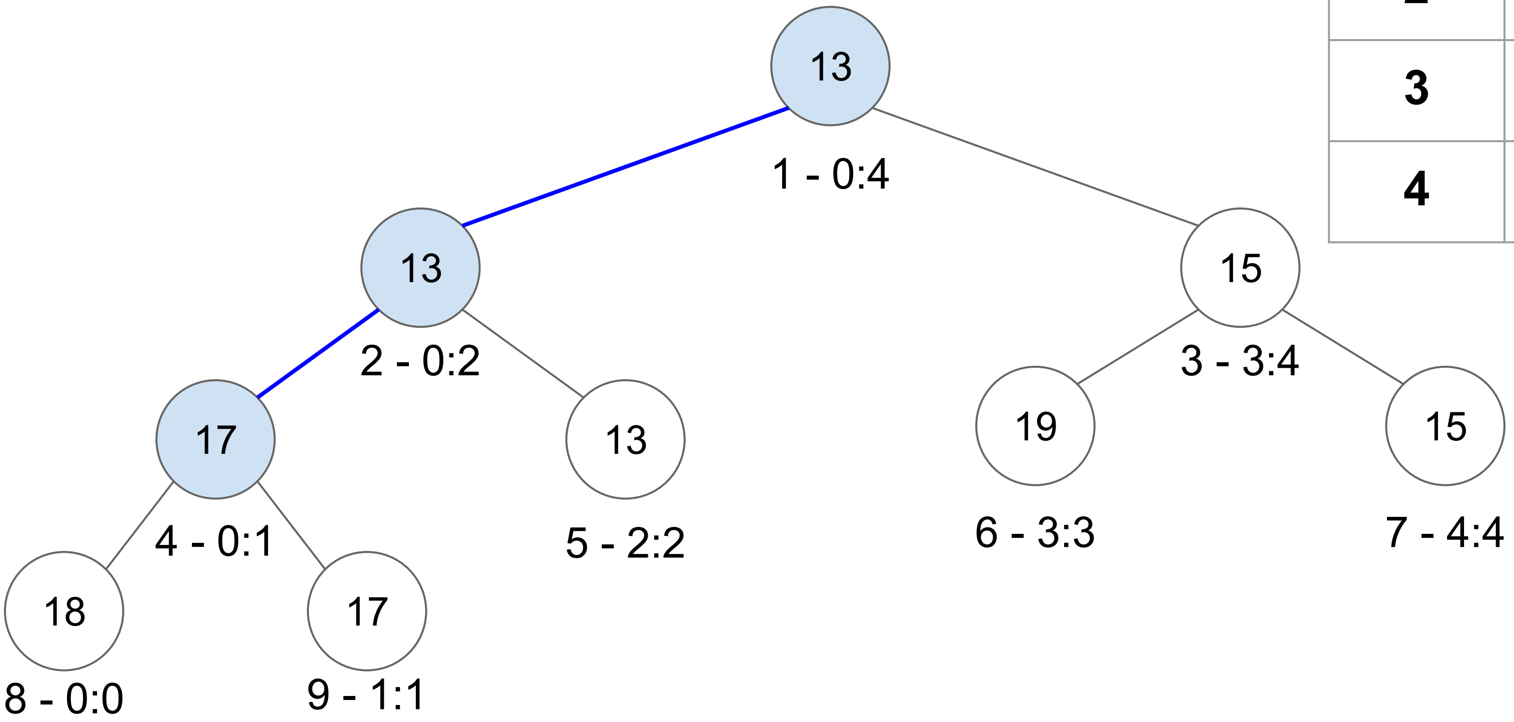
Atualização



0	18
1	12
2	13
3	19
4	15

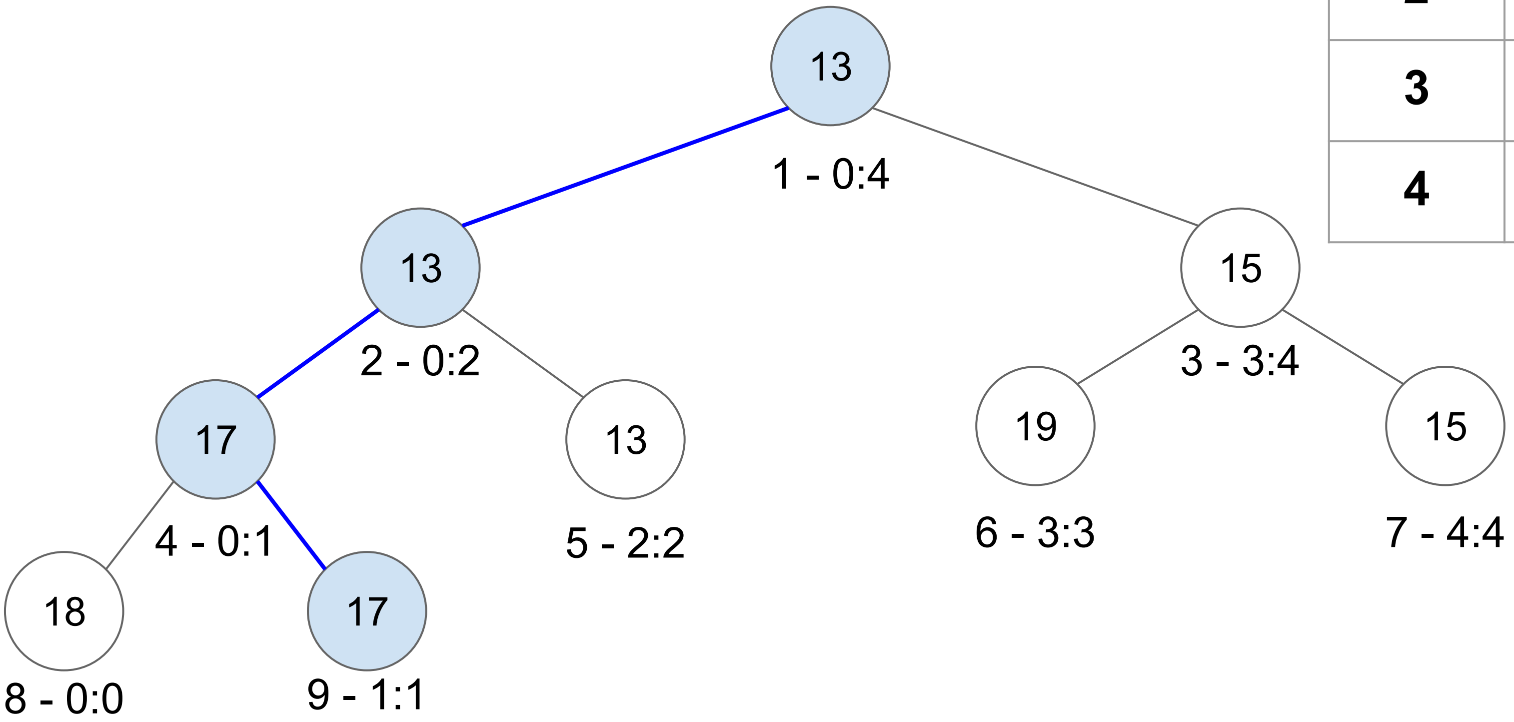
Atualização

0	18
1	12
2	13
3	19
4	15



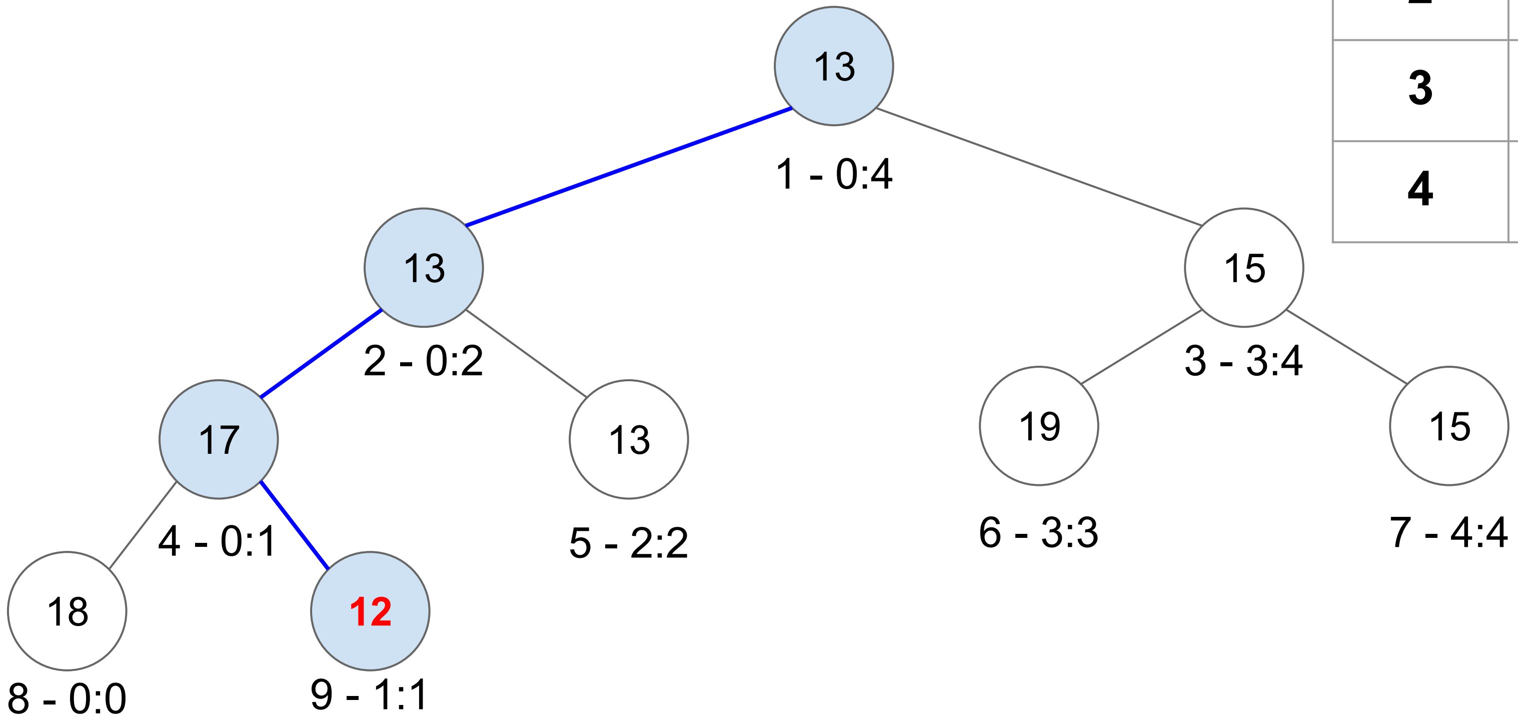
Atualização

0	18
1	12
2	13
3	19
4	15



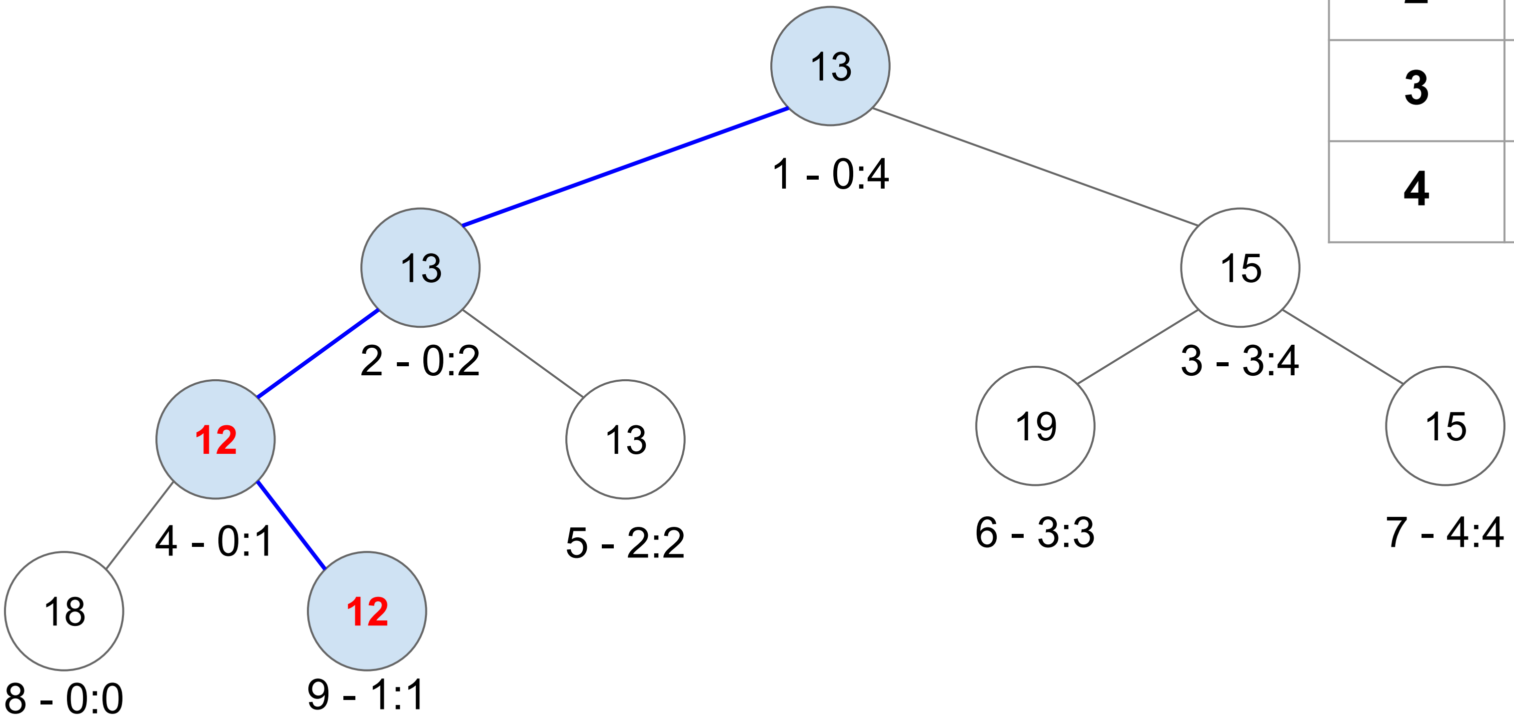
Atualização

0	18
1	12
2	13
3	19
4	15

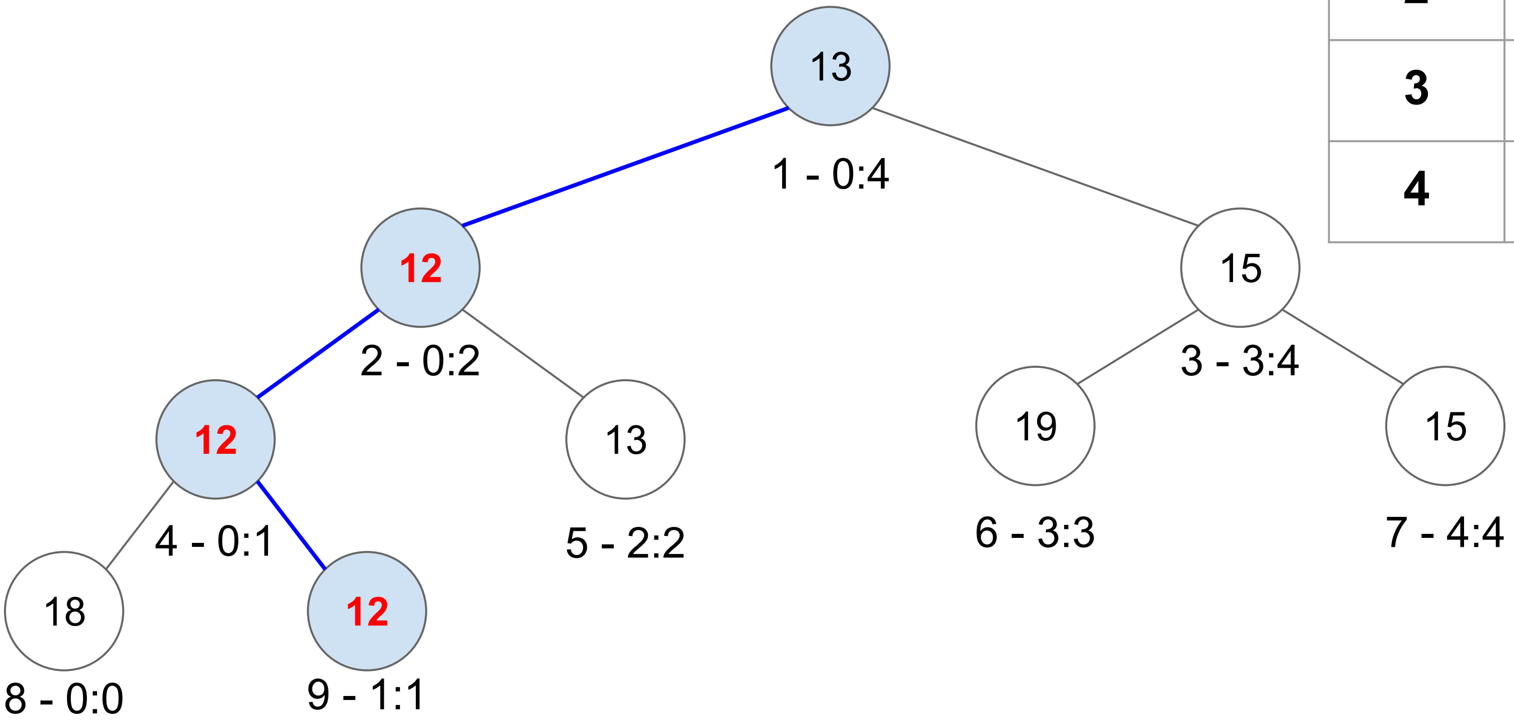


Atualização

0	18
1	12
2	13
3	19
4	15



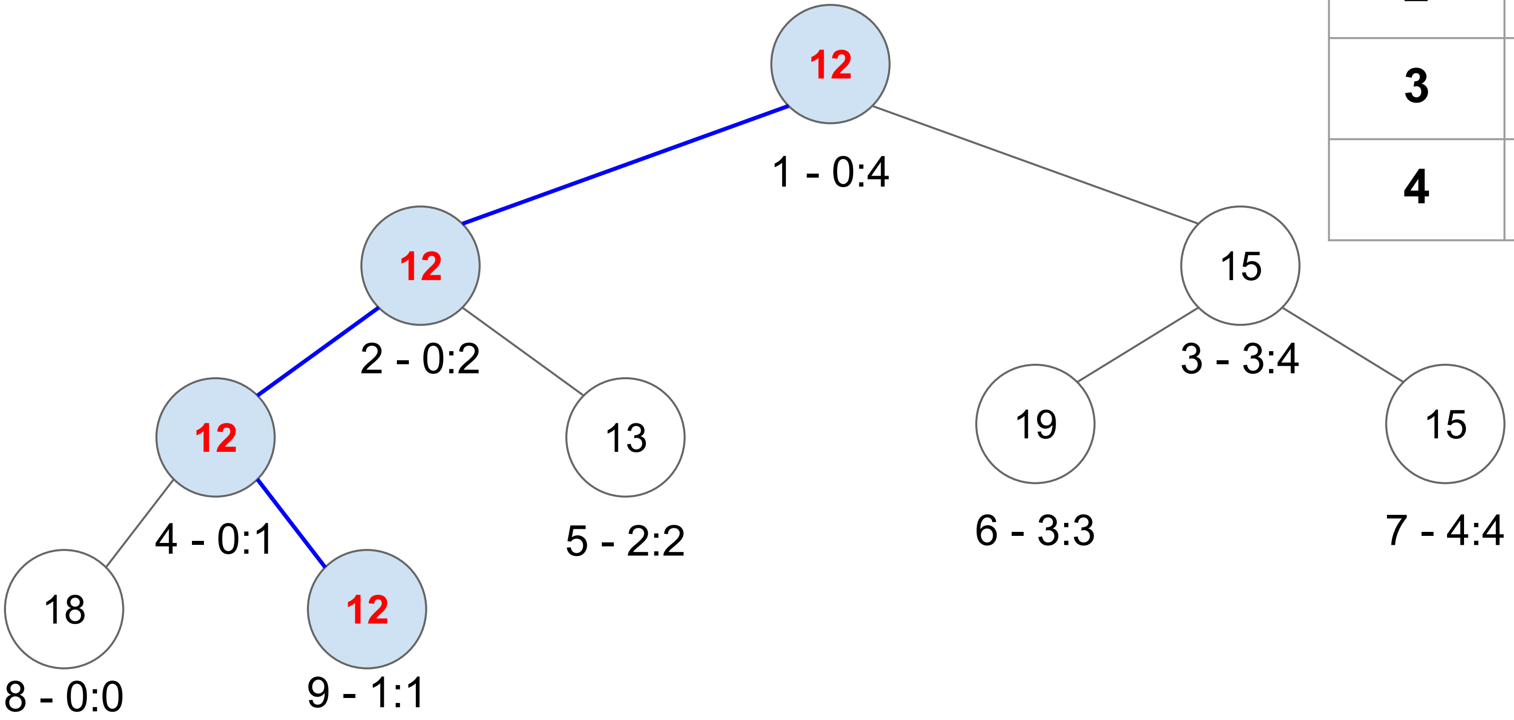
Atualização



0	18
1	12
2	13
3	19
4	15

Atualização

0	18
1	12
2	13
3	19
4	15



Atualização

```
void update(int no, int i, int j, int pos, int new_v)
{
    if(i == j)  //Se estamos em uma folha (i == j == pos)
    {
        vetor[pos] = new_v;
        st[no] = new_v;
        return;
    }

    if(i > pos || j < pos)
        return;    //O intervalo não contém o índice pos
```

Atualização

```
    //O intervalo contém o índice, mas temos que chegar no nó
específico, e voltar recursivamente atualizando os filhos

    int mid = (i + j)/2;
    //Percorrendo e atualizando os filhos
    update(no*2, i, mid, pos, new_v);
    update(no*2 + 1, mid + 1, j, pos, new_v);
    //Com os filhos atualizados, atualizar o próprio nó
    st[no] = f(st[no*2], st[no*2 + 1]);
}
```

Chamada:
update(1, 0, size-1, pos, new_v)

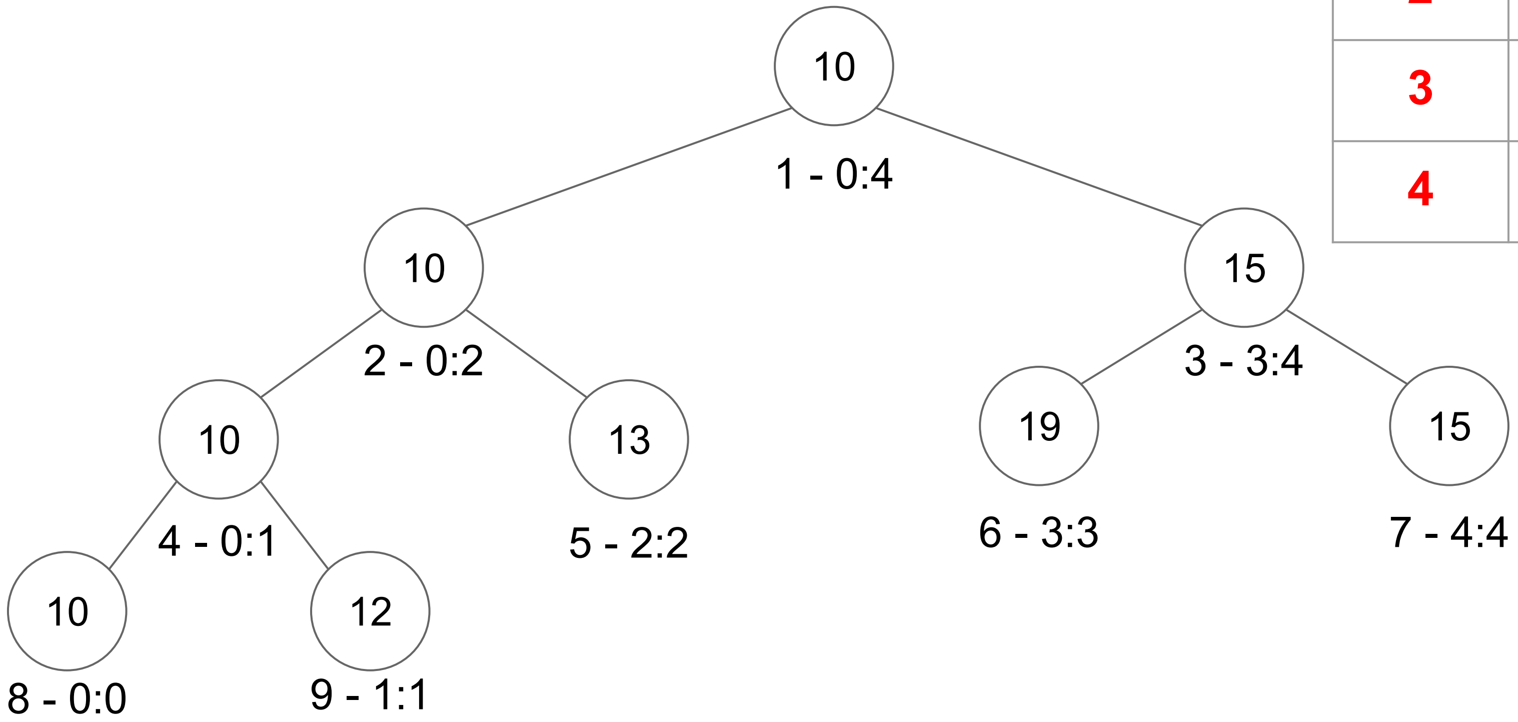
Consulta

- Consultando o menor valor entre A e B
  - Para retornar a posição com o menor valor entre A e B, iniciaremos a busca a partir do nó 1, com intervalo [0, N-1], e seguiremos o seguinte procedimento:

```
Se [i,j] estiver contido entre [A,B]     (A <= i <= j <= B)
        retorna st[no]
Se [i,j] e [A,B] forem disjuntos          (A > j ou i > B)
        retorna elemento neutro
Senão
        chamamos a função recursivamente para os filhos
```

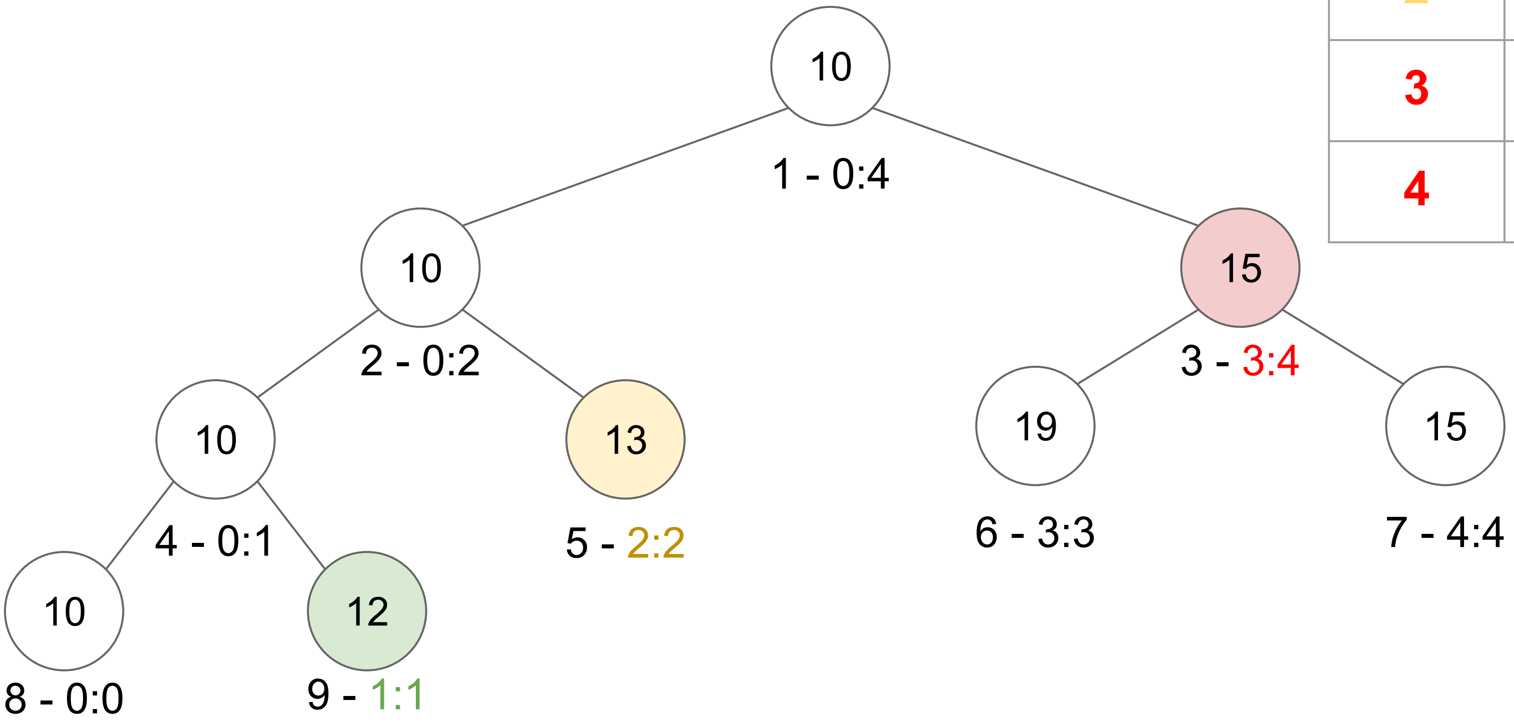
Consulta

0	10
1	12
2	13
3	19
4	15



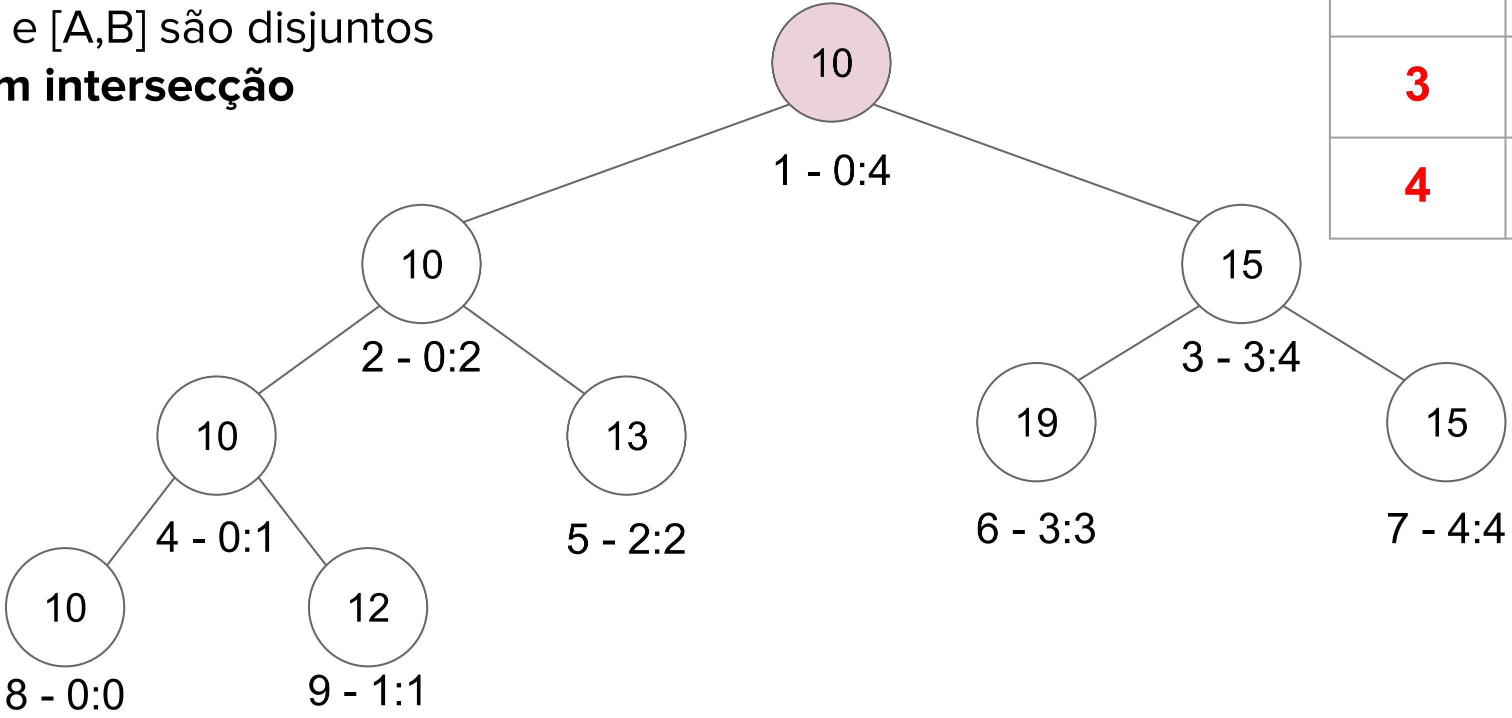
Consulta

0	10
1	12
2	13
3	19
4	15



Consulta

1. [i,j] está em em [A,B]
2. [i,j] e [A,B] são disjuntos
3. **Tem intersecção**

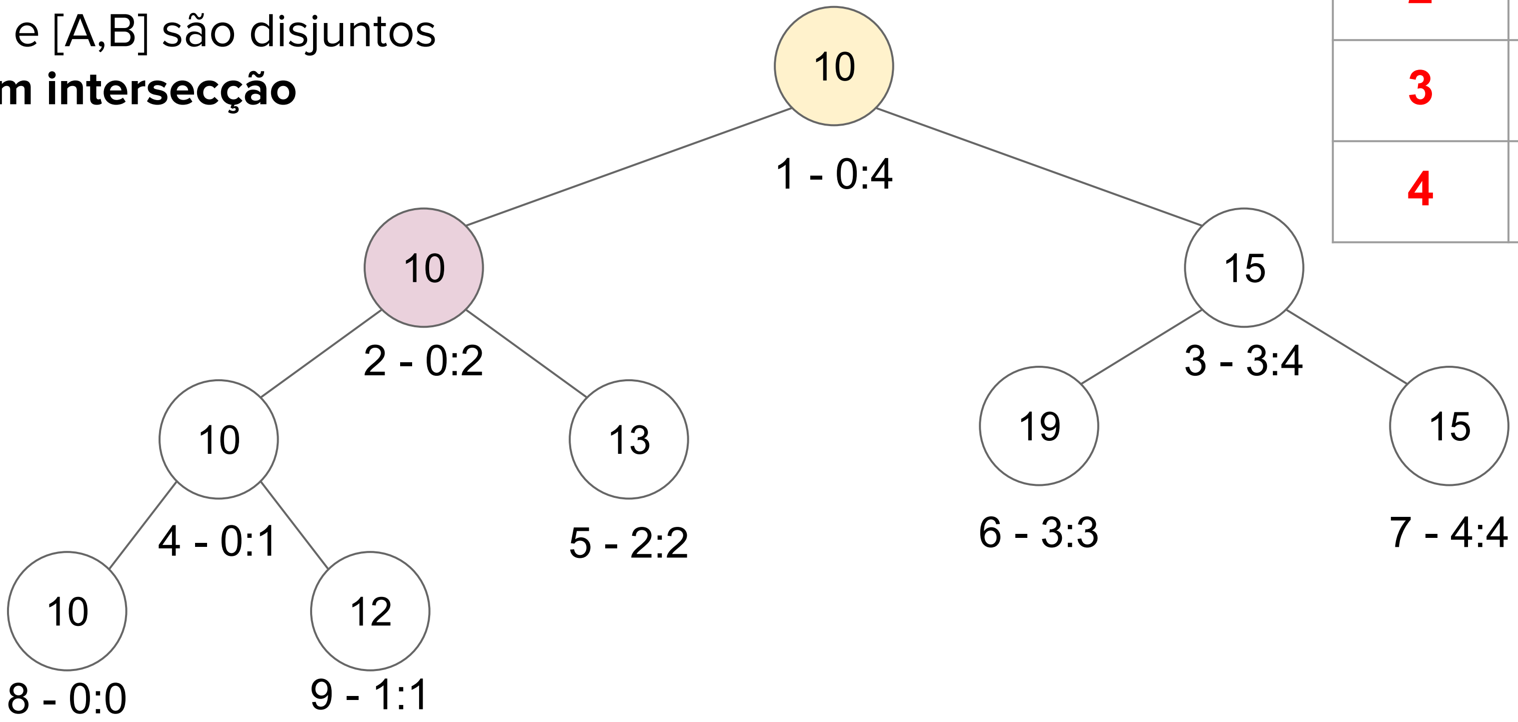


0	10
1	12
2	13
3	19
4	15

Consulta

1. [i,j] está em em [A,B]
2. [i,j] e [A,B] são disjuntos
3. **Tem intersecção**

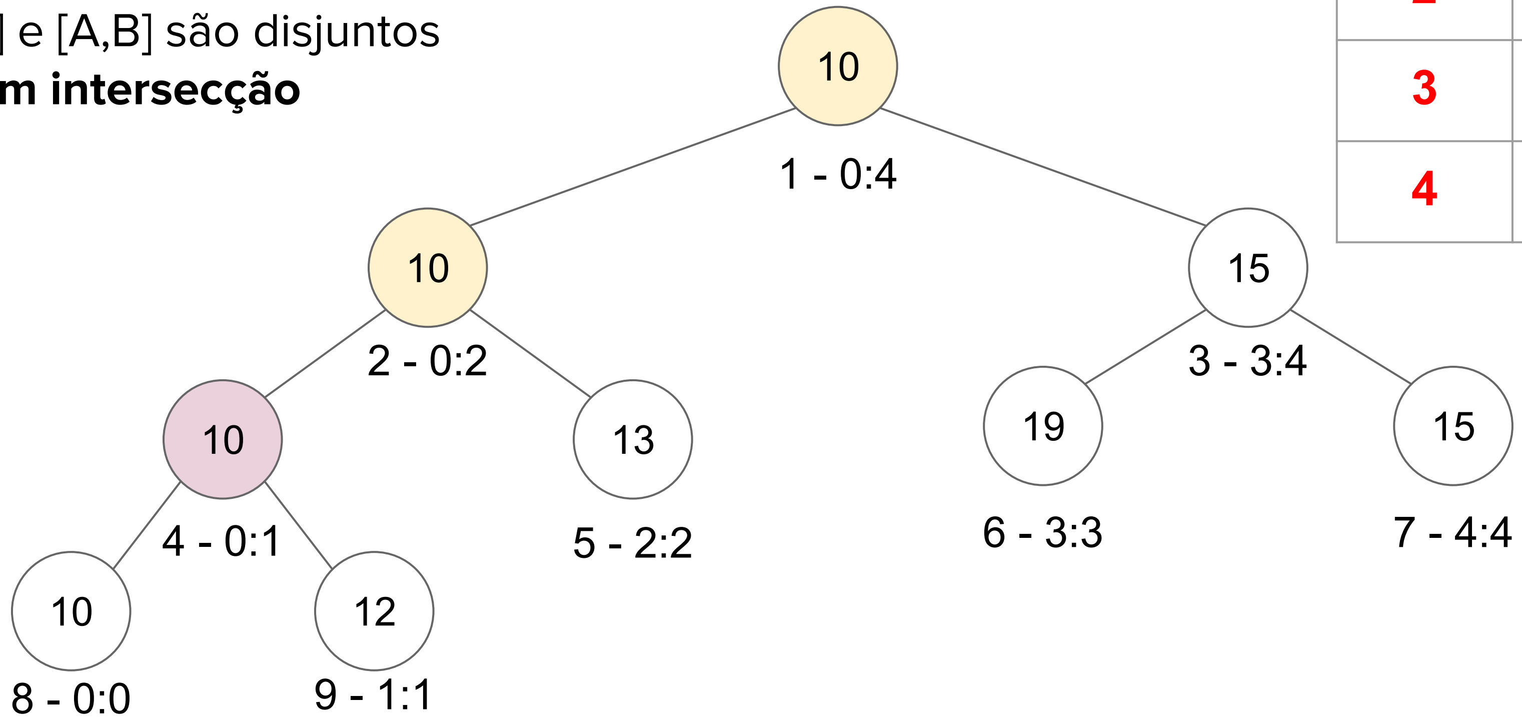
0	10
1	12
2	13
3	19
4	15



Consulta

1. [i,j] está em em [A,B]
2. [i,j] e [A,B] são disjuntos
3. **Tem intersecção**

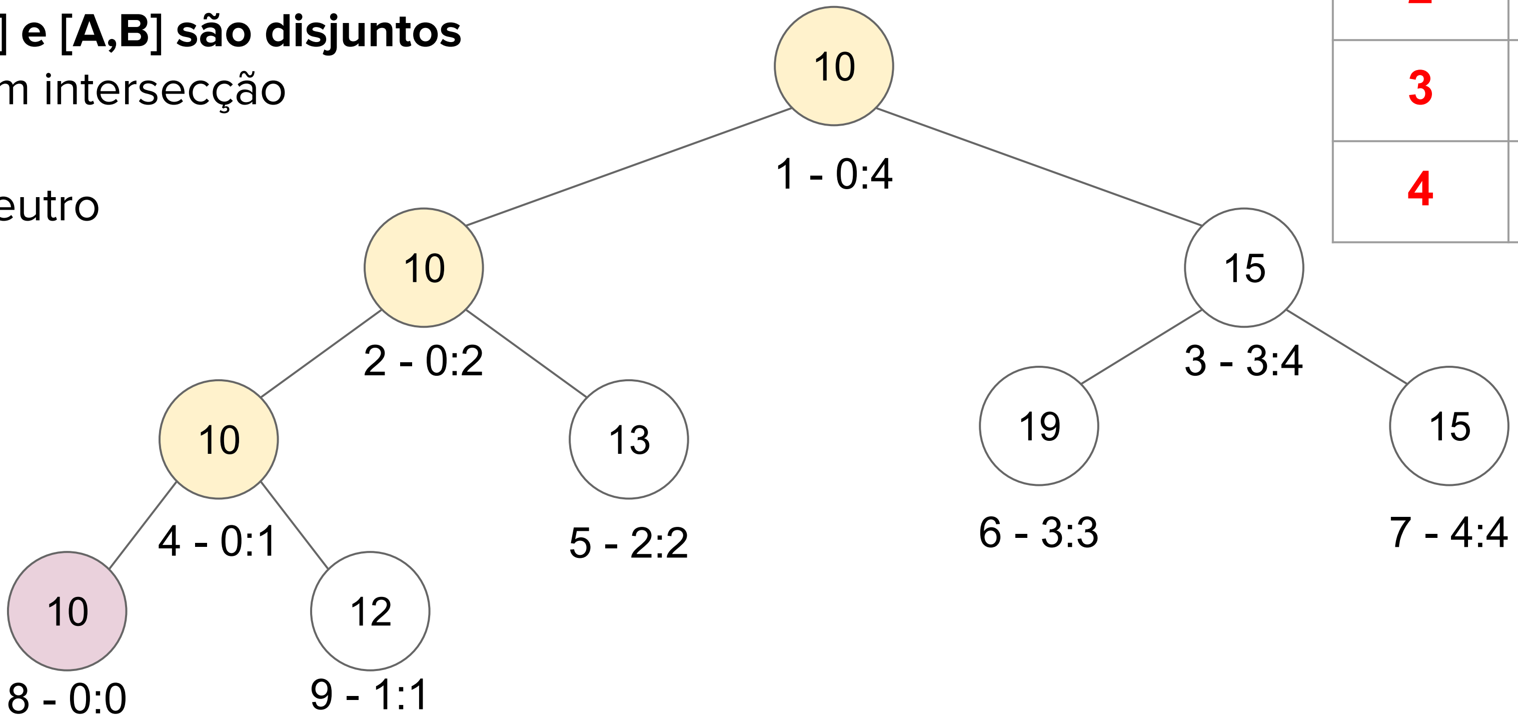
0	10
1	12
2	13
3	19
4	15



Consulta

1. [i,j] está em em [A,B]
2. **[i,j] e [A,B] são disjuntos**
3. Tem intersecção

8: el_neutro

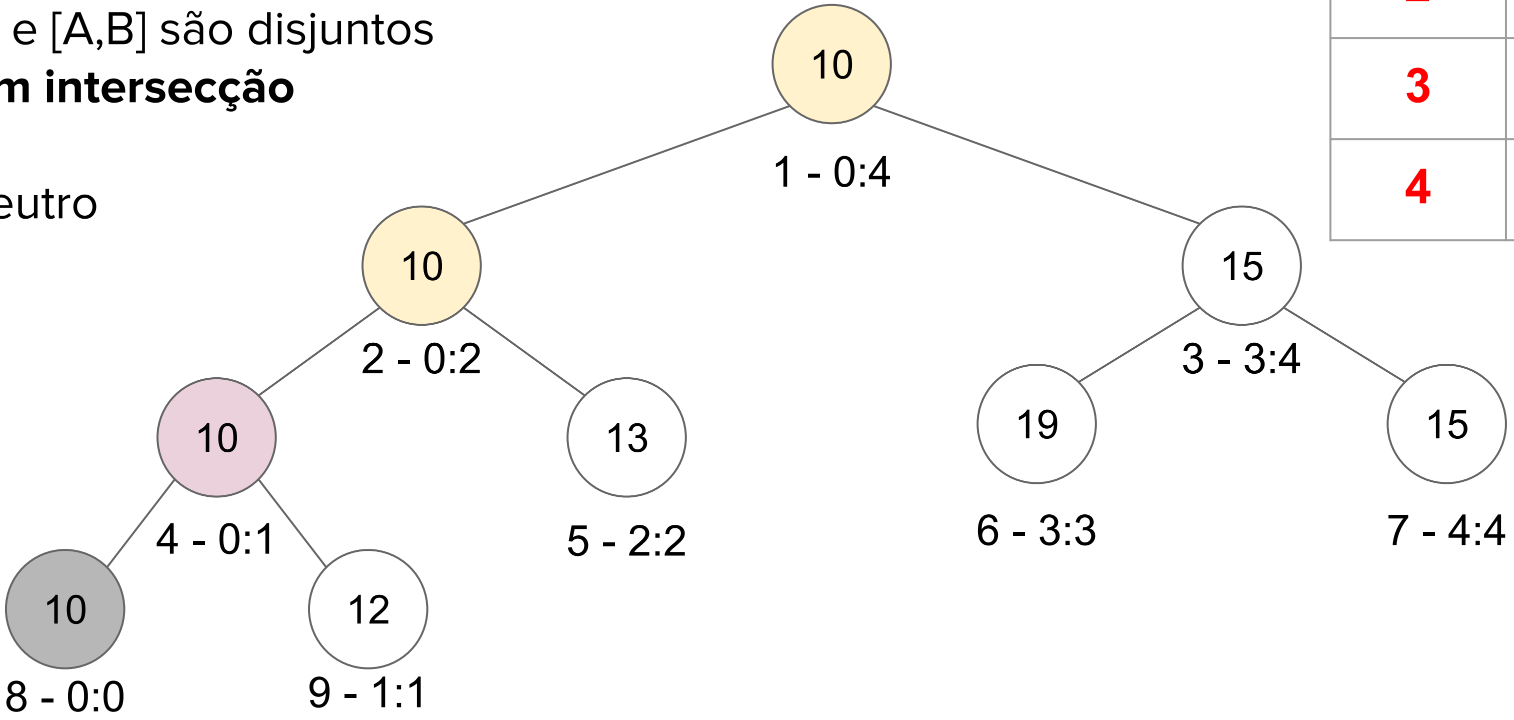


0	10
1	12
2	13
3	19
4	15

Consulta

1. [i,j] está em em [A,B]
2. [i,j] e [A,B] são disjuntos
3. **Tem intersecção**

8: el_neutro

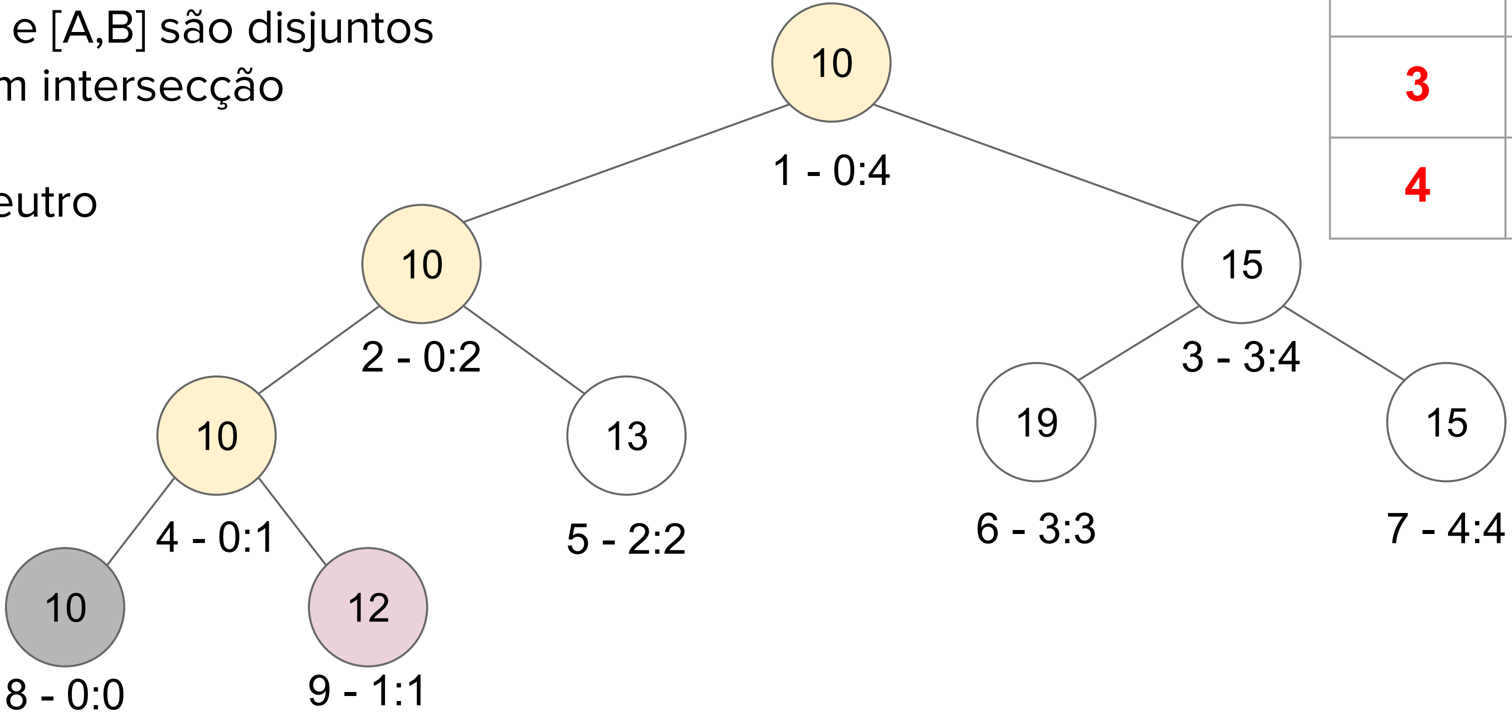


0	10
1	12
2	13
3	19
4	15

Consulta

1. **[i,j] está em em [A,B]**
2. [i,j] e [A,B] são disjuntos
3. Tem intersecção

8: el_neutro
9: 12

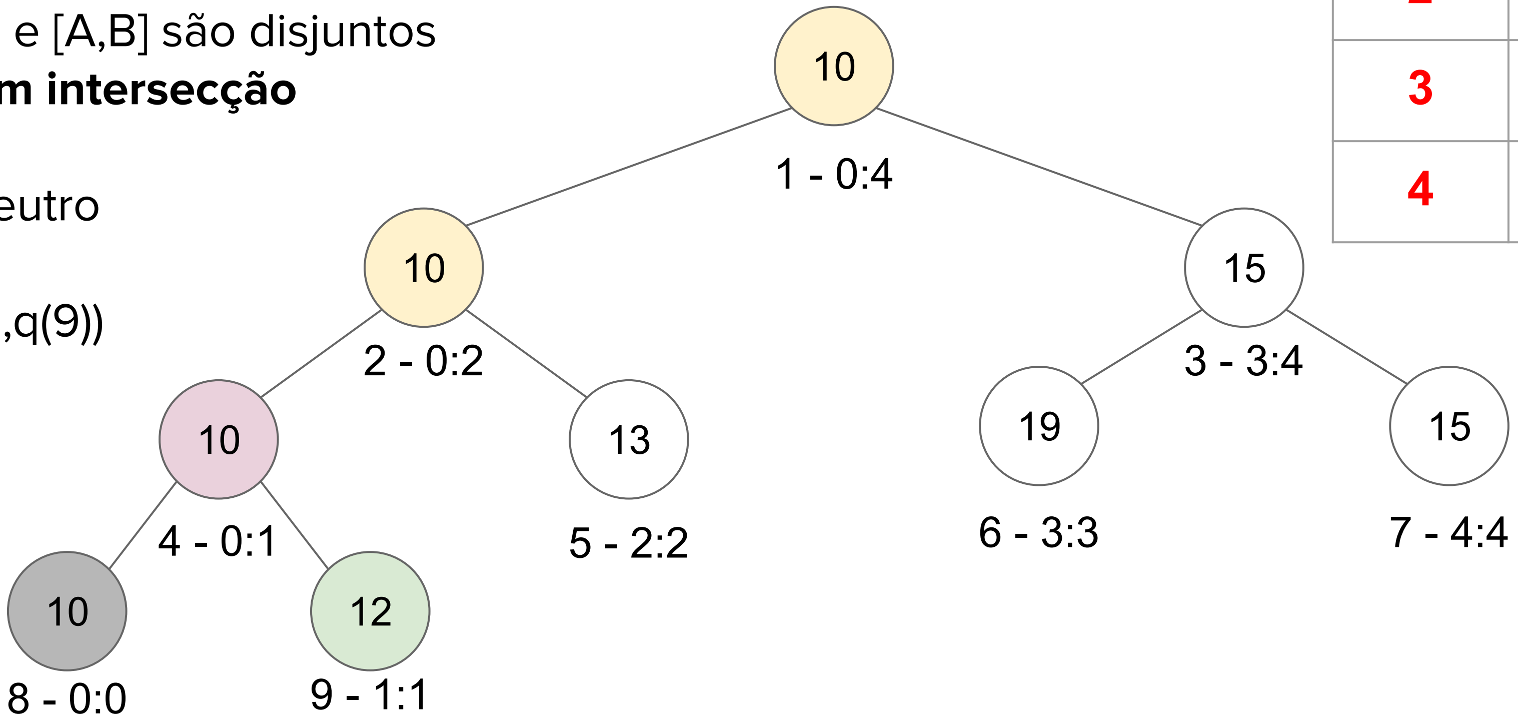


0	10
1	12
2	13
3	19
4	15

Consulta

1. [i,j] está em em [A,B]
2. [i,j] e [A,B] são disjuntos
3. **Tem intersecção**

8: el_neutro
9: 12
4: f(q(8),q(9))

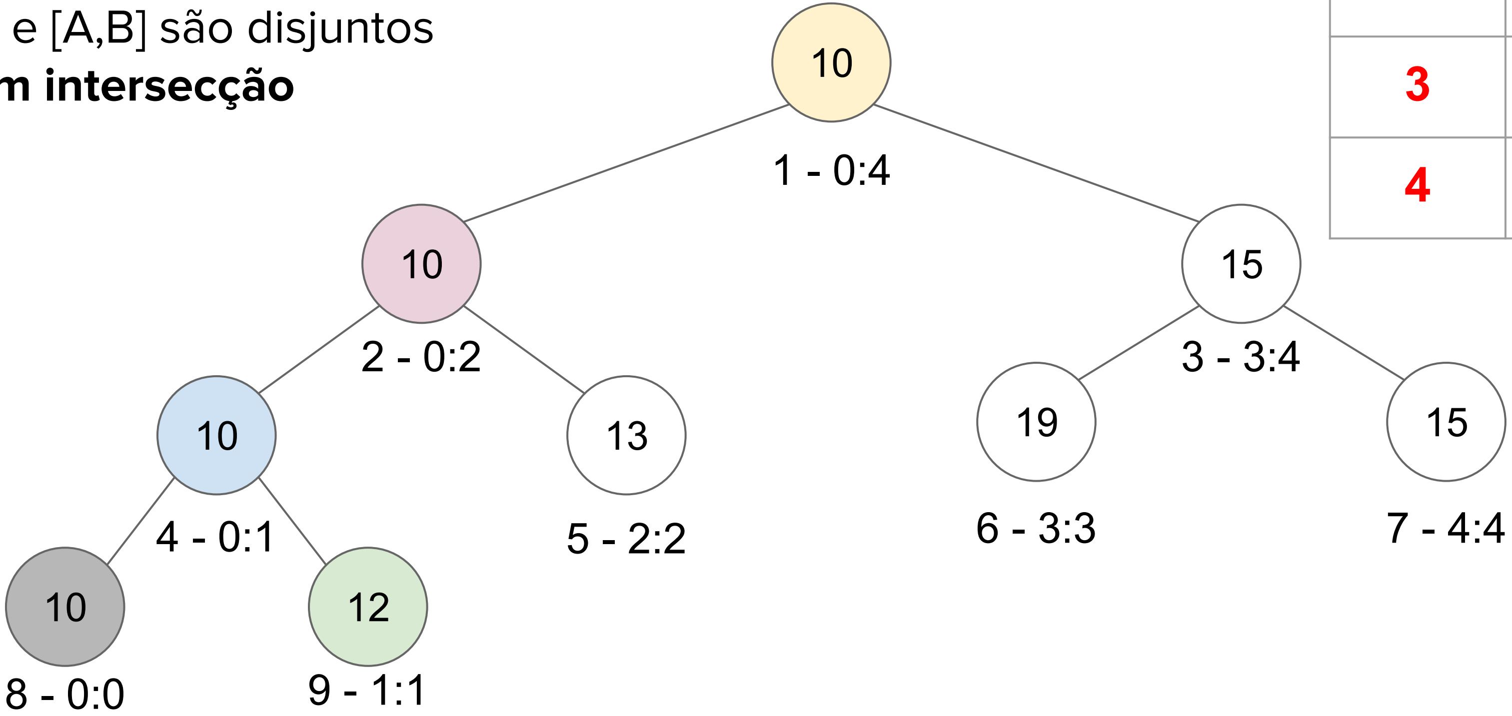


0	10
1	12
2	13
3	19
4	15

Consulta

1. [i,j] está em em [A,B]
2. [i,j] e [A,B] são disjuntos
3. **Tem intersecção**

4: 12

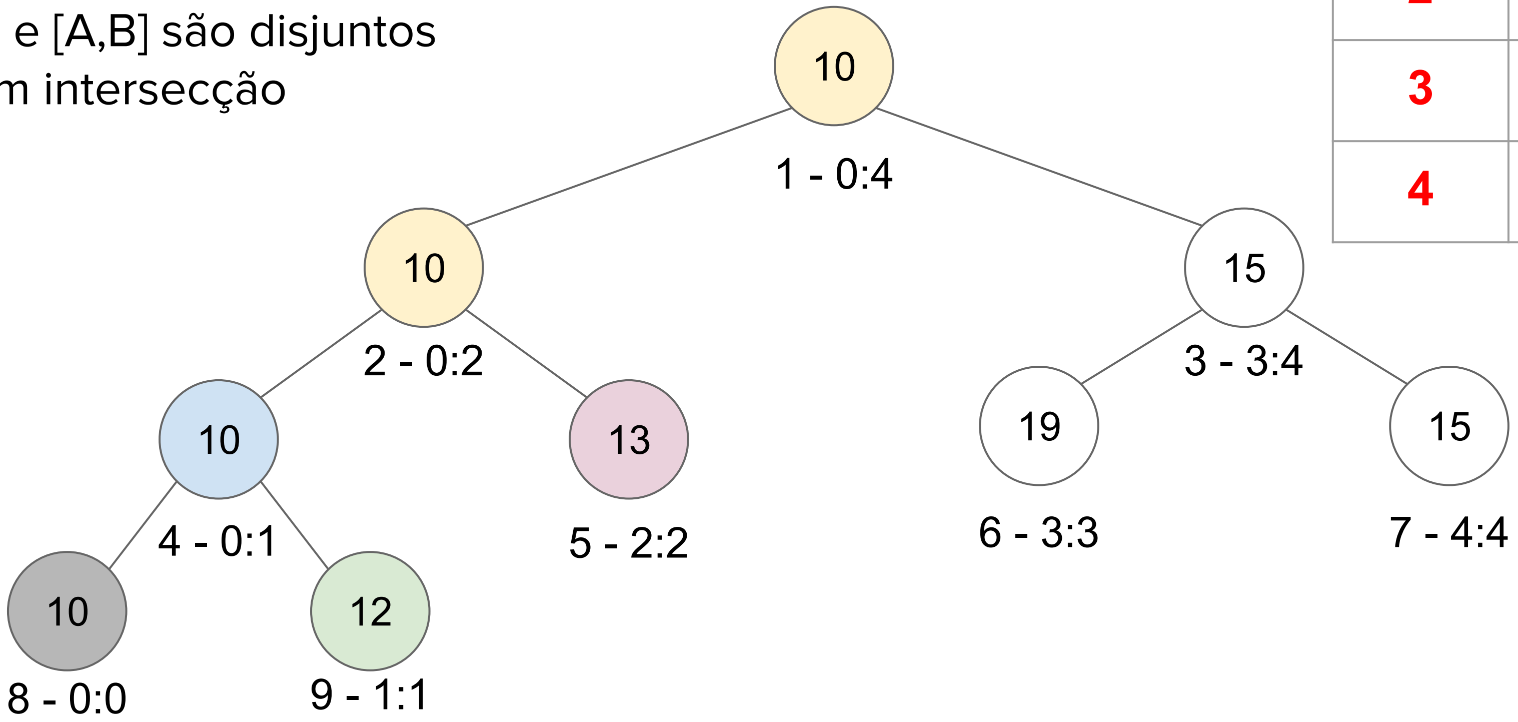


0	10
1	12
2	13
3	19
4	15

Consulta

1. **[i,j] está em em [A,B]**
2. [i,j] e [A,B] são disjuntos
3. Tem intersecção

4: 12
5: 13

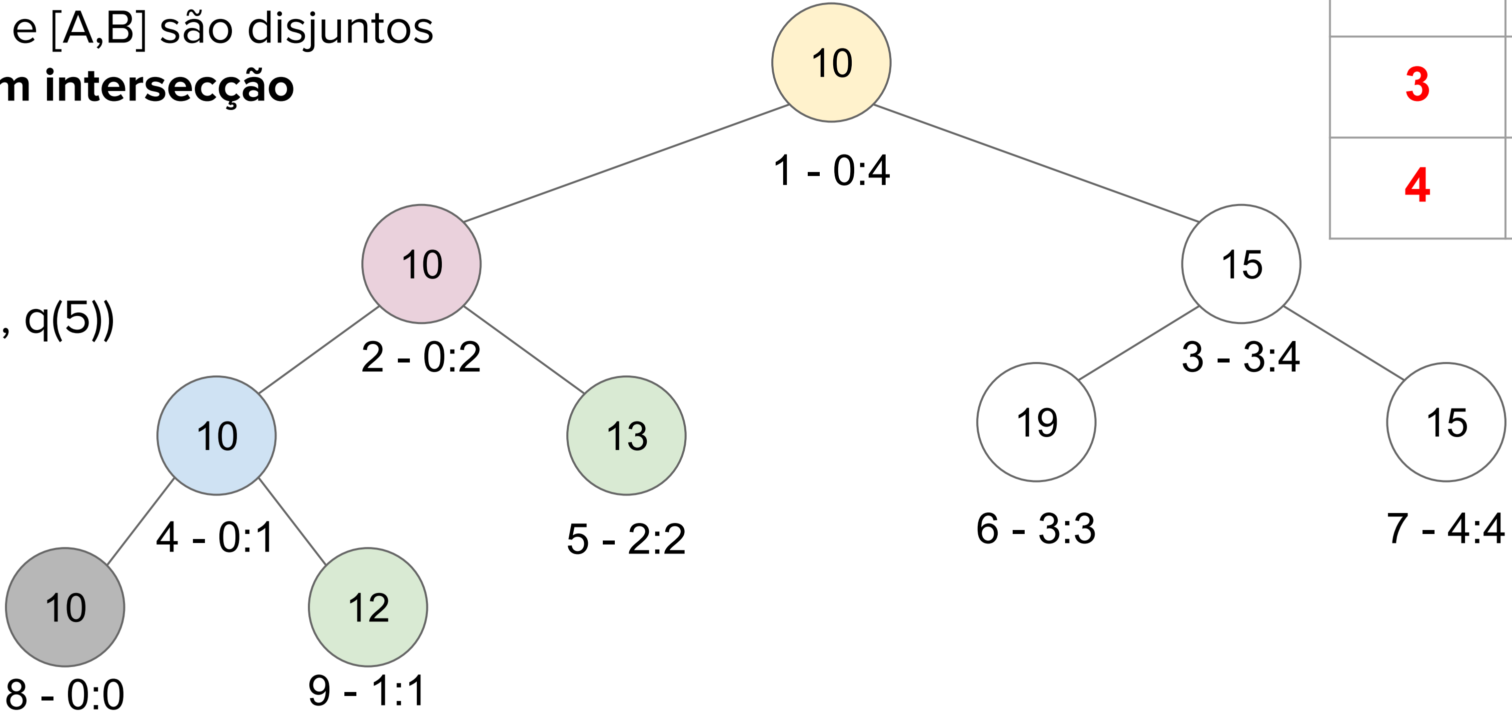


0	10
1	12
2	13
3	19
4	15

Consulta

1. [i,j] está em em [A,B]
2. [i,j] e [A,B] são disjuntos
3. **Tem intersecção**

4: 12
5: 13
2: f(q(4), q(5))

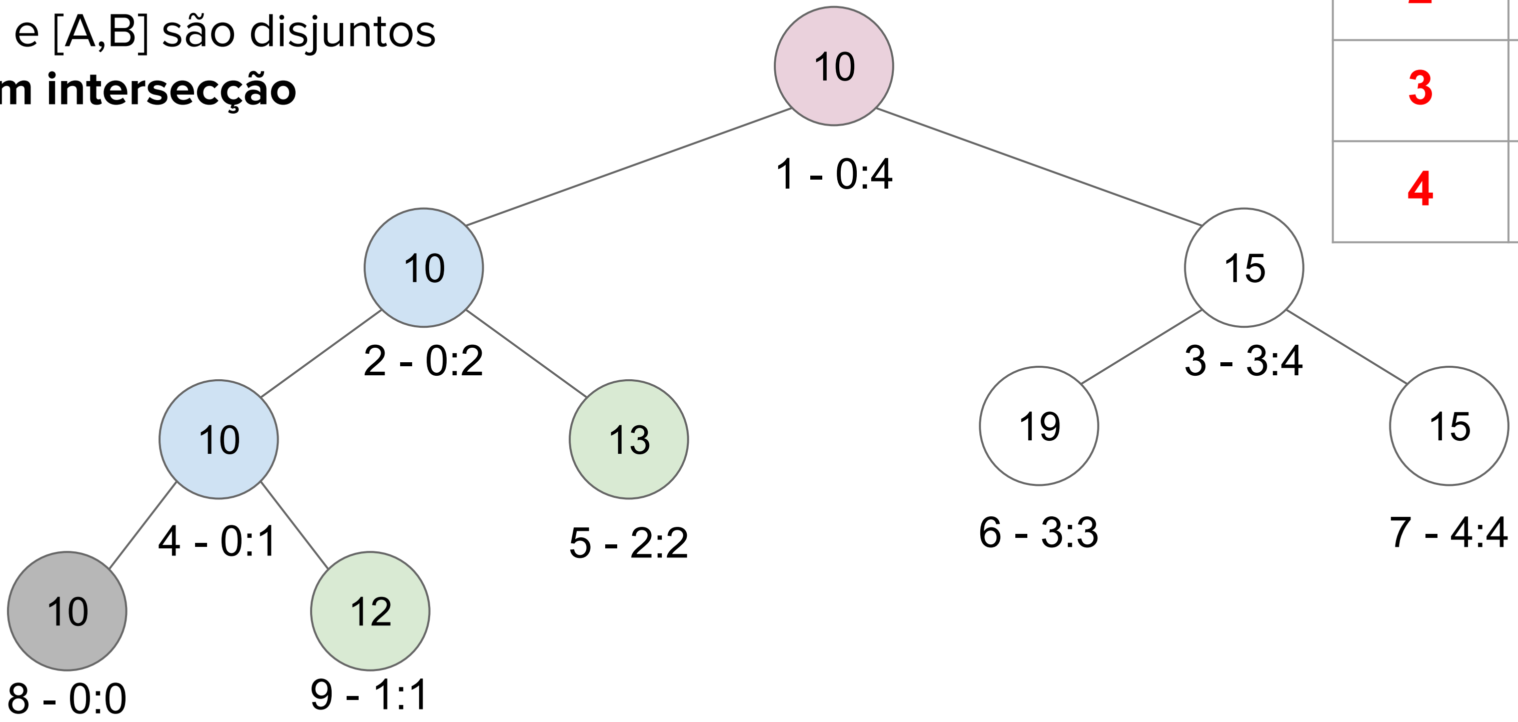


0	10
1	12
2	13
3	19
4	15

Consulta

1. [i,j] está em em [A,B]
2. [i,j] e [A,B] são disjuntos
3. **Tem intersecção**

2: 12



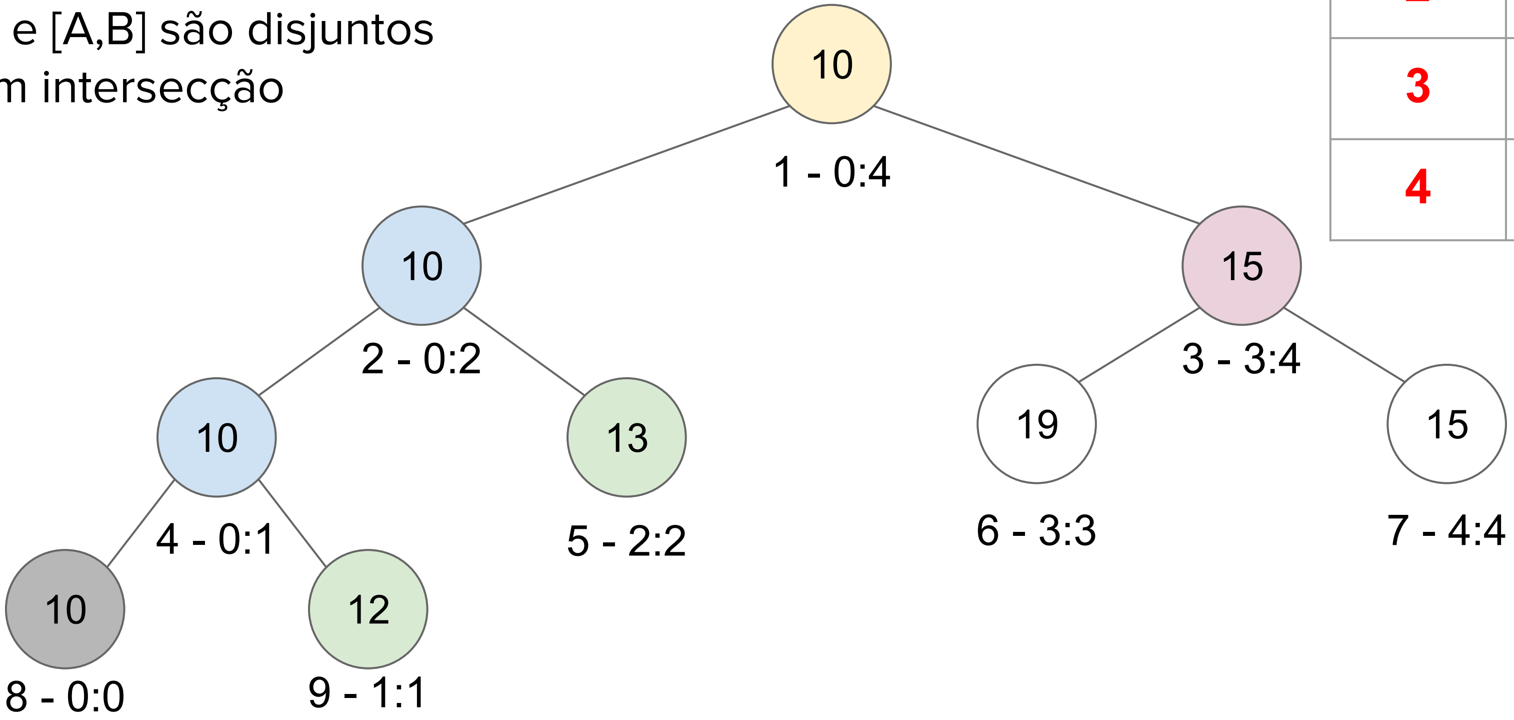
0	10
1	12
2	13
3	19
4	15

Consulta

1. **[i,j] está em em [A,B]**
2. [i,j] e [A,B] são disjuntos
3. Tem intersecção

2: 12
3: 15

0	10
1	12
2	13
3	19
4	15

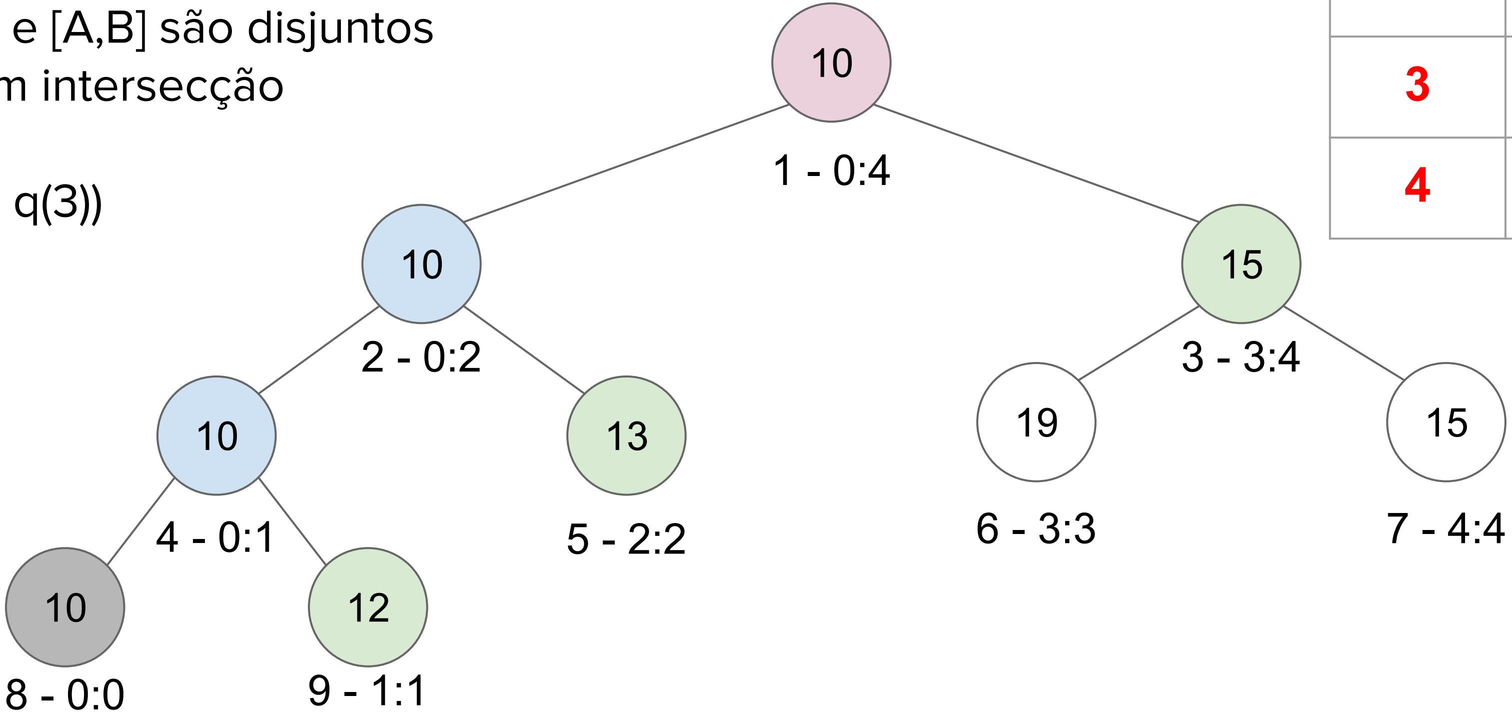


Consulta

0	10
1	12
2	13
3	19
4	15

1. **[i,j] está em em [A,B]**
2. [i,j] e [A,B] são disjuntos
3. Tem intersecção

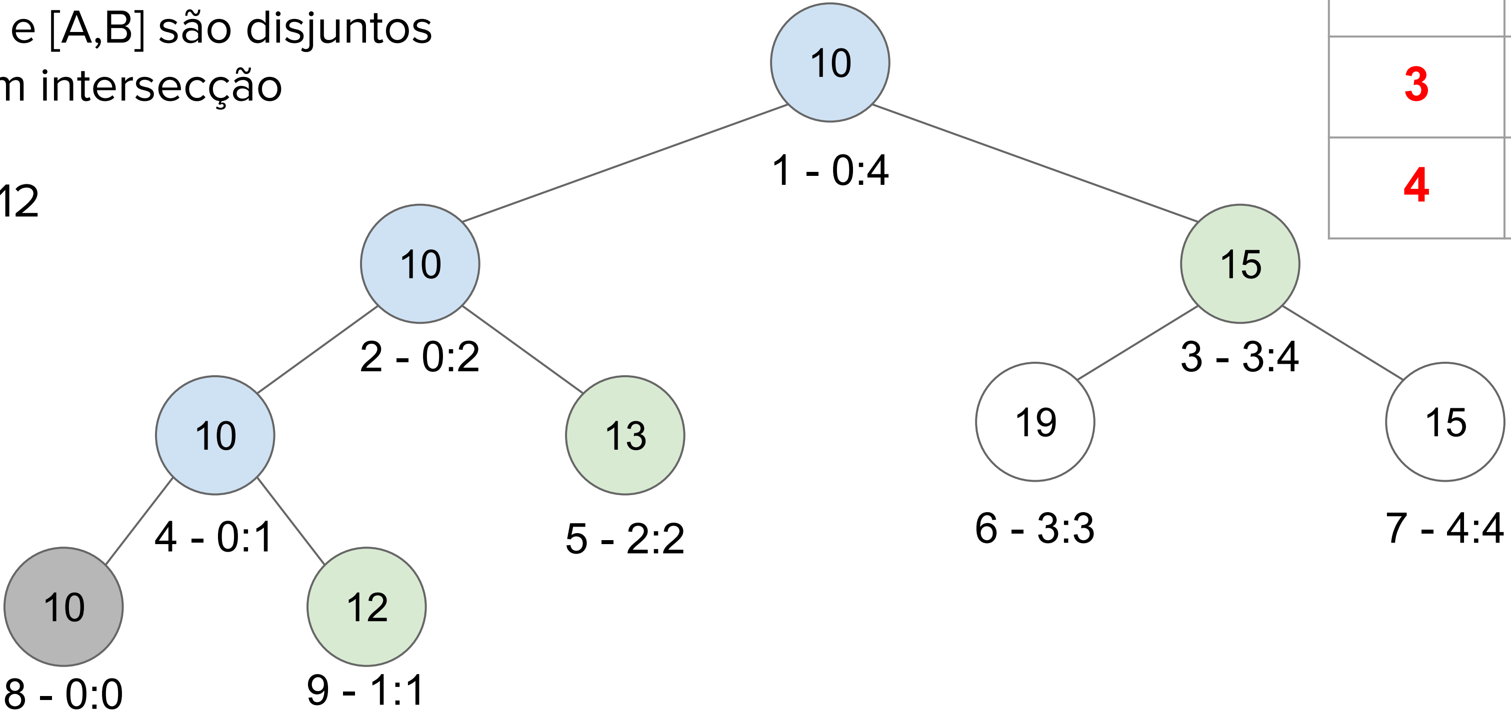
1: f(q(2), q(3))
2: 12
3: 15



Consulta

1. **[i,j] está em em [A,B]**
2. [i,j] e [A,B] são disjuntos
3. Tem intersecção

Resp = 12



0	10
1	12
2	13
3	19
4	15

Consulta

```
void query(int no, int i, int j, int A, int B)
{
    //Se o nó está fora do intervalo A:B
    if(j < A || B < i)
        return el_neutro; //retornamos o elemento neutro
    //Se o nó está completamente incluido no intervalo A:B
    if(i >= A && j <= B)
        return st[no];    //retornamos o valor deste nó
    //Caso contrário, o nó está parcialmente contido em A:B,
então temos que olhar para os filhos
    int mid = (i + j)/2;
    return f(query(no*2, i, mid, A, B),
                query(no*2 + 1, mid + 1, j, A, B));
}
```

Lazy Propagation

- Em certas situações, temos que atualizar todas as posições de um intervalo [A,B]. A partir das funções que já temos, teríamos que chamar a função update() para cada posição desse intervalo. Nesse caso, a Segment Tree não é muito eficiente, com complexidade O(n.logn) para atualizações em intervalo.

- Uma forma de resolver isso, é usar como Segment Tree com Lazy Propagation. A ideia consiste em atualizar um nó apenas quando a informação sobre aquele nó for necessária.

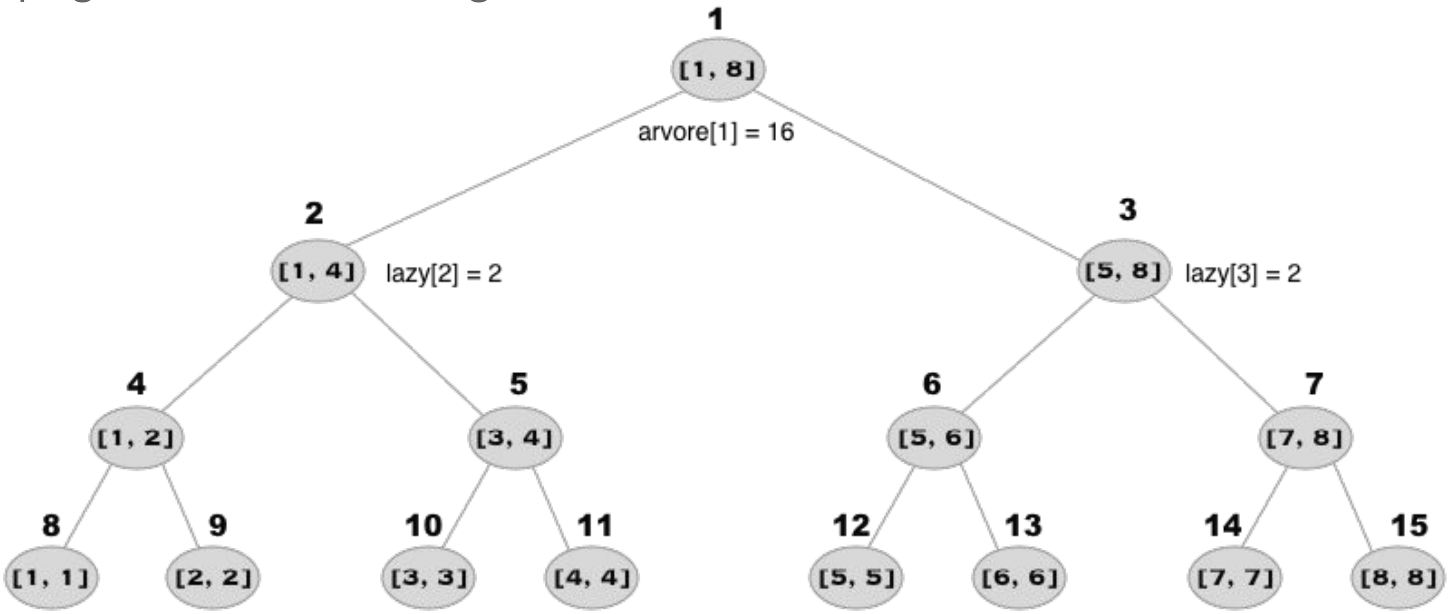
Lazy Propagation

- Vamos usar uma ideia parecida com a da consulta. Em vez de atualizar individualmente os elementos, podemos atualizar a resposta nos intervalos que os contém, e postergar a atualização dos filhos.
- Isso mudará um pouco nossa implementação. Para exemplificar, vamos considerar o problema da RSQ
- Vamos adicionar dois vetores na implementação

```
vector<int> lazy; //Indica o valor para qual o nó precisa
                   ser atualizado
vector<bool> has; //Indica se há uma atualização para ser
                  feita naquele nó
```

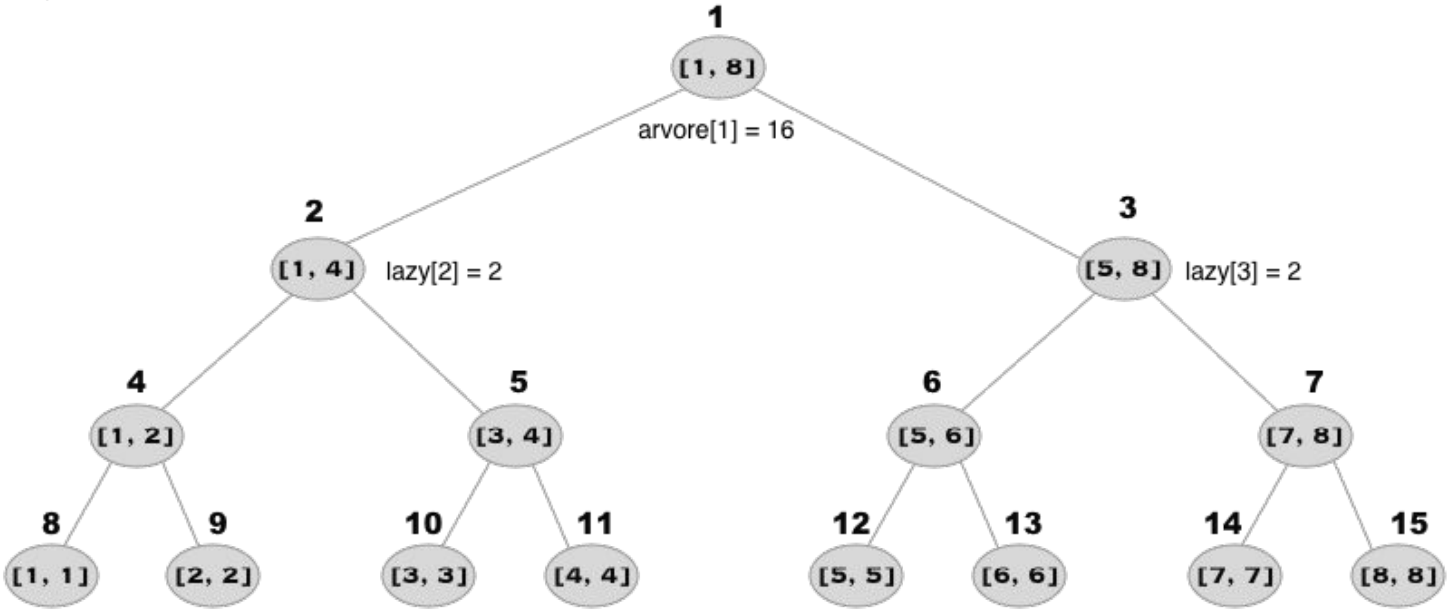
Lazy Propagation

- Imagine que temos um vetor de 8 posições, todas com o valor 0, e então somamos o valor 2 em todas as posições. Nossa Segment Tree com Lazy Propagation ficaria da seguinte forma:



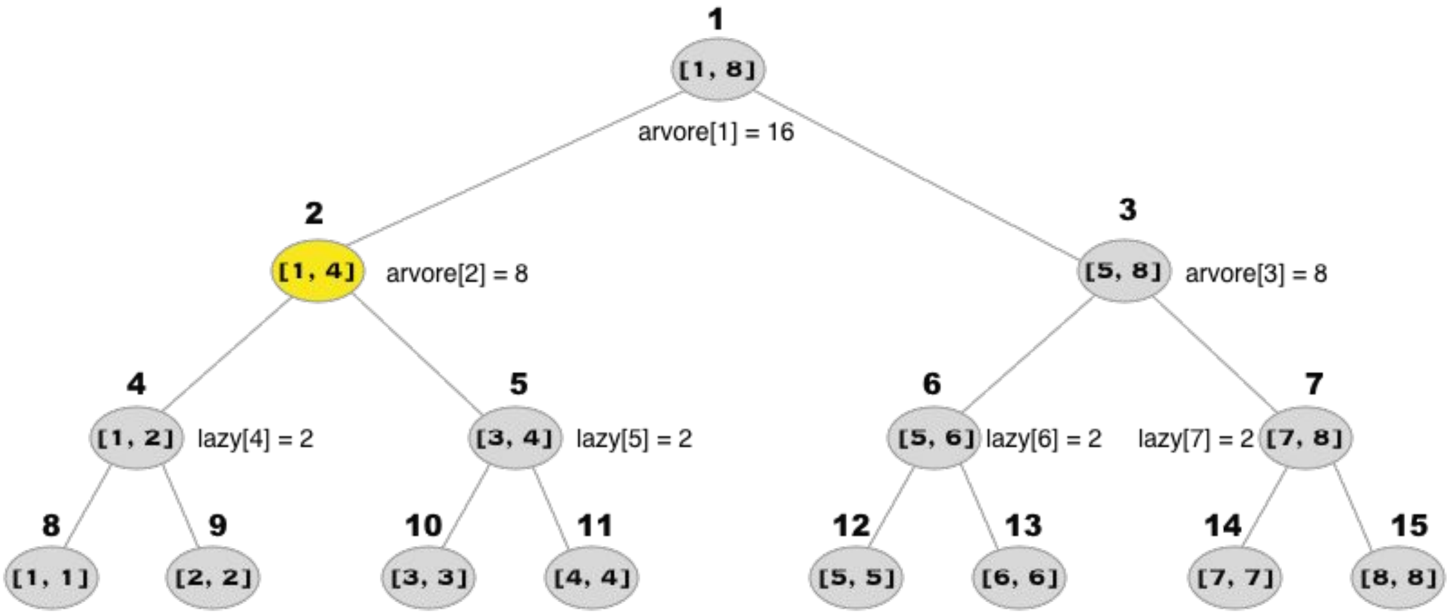
Lazy Propagation

- Perceba que apenas o nó 1 foi atualizado de fato, para seus filhos apenas indicamos (através do vetor ‘lazy’) que teremos que somar 2 em todas as posições dos intervalos correspondentes.



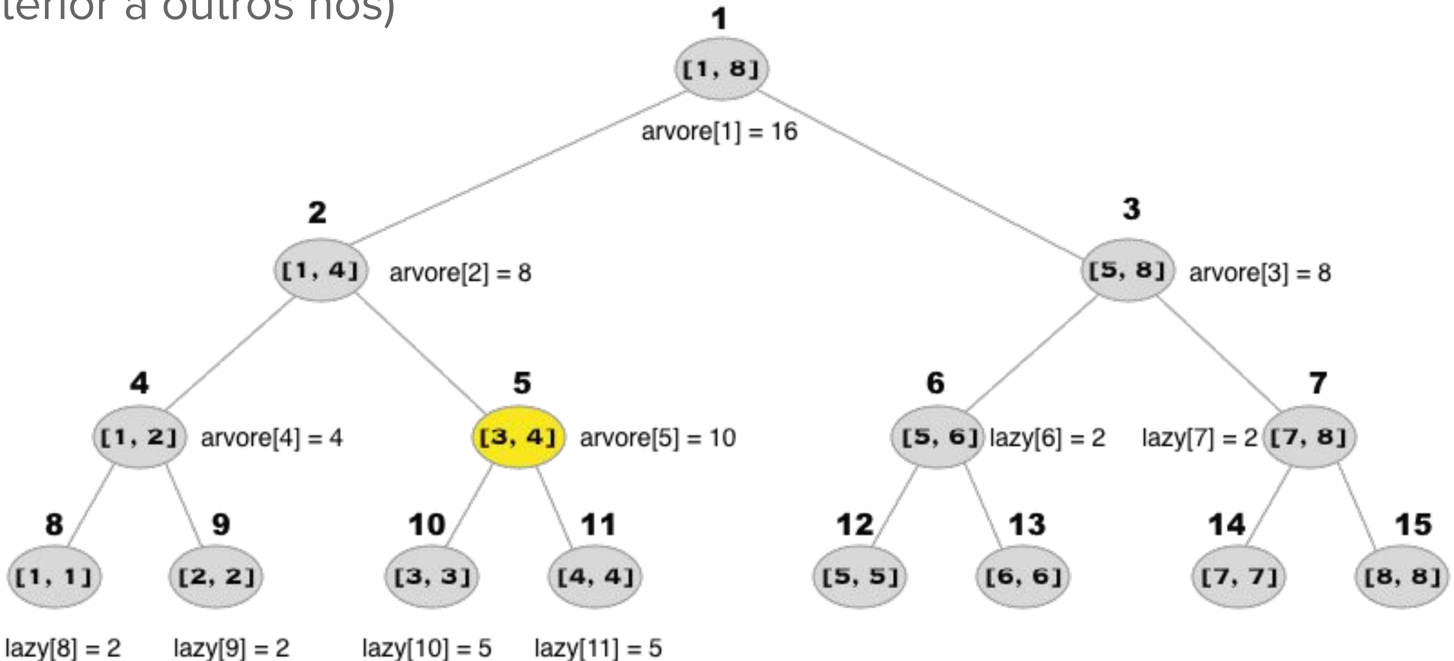
Lazy Propagation

- Agora, se somarmos 3 em todas as posições no intervalo [3,4], teremos um processo um pouco maior (e teremos que aplicar parcialmente a atualização anterior a outros nós)



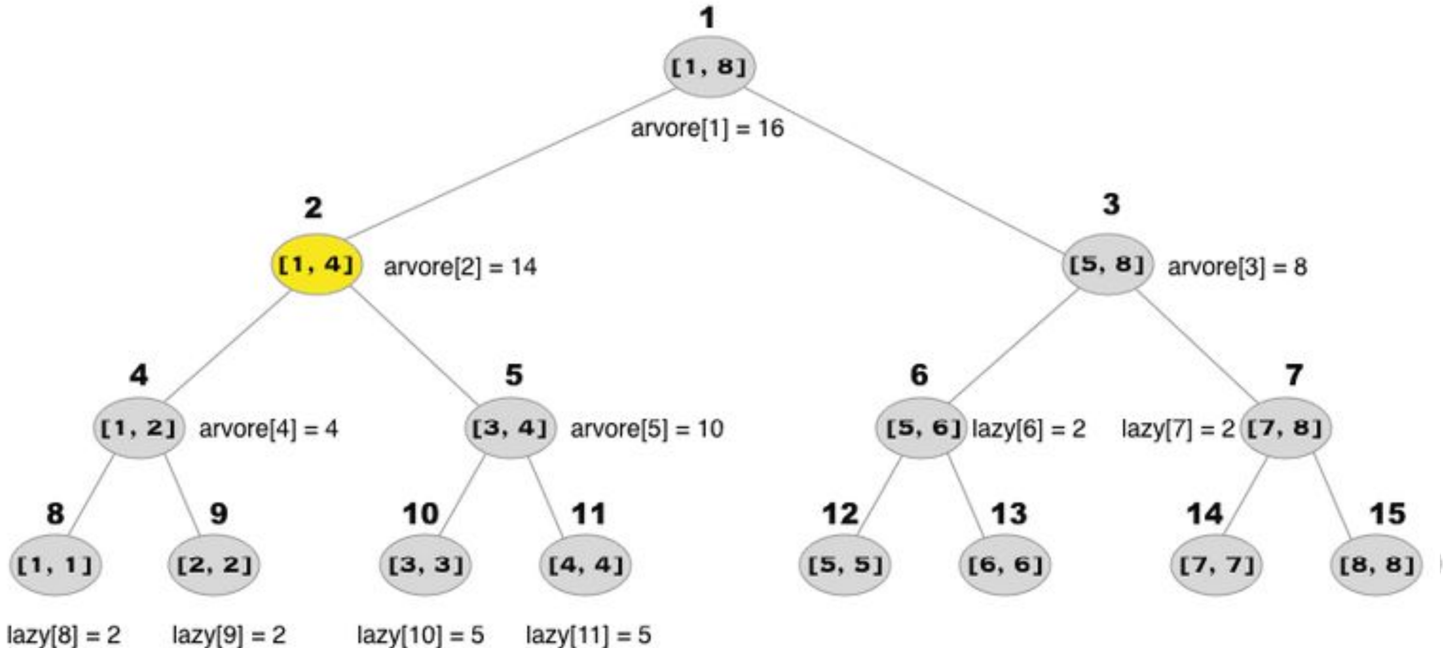
Lazy Propagation

- Agora, se somarmos 3 em todas as posições no intervalo [3,4], teremos um processo um pouco maior (e teremos que aplicar parcialmente a atualização anterior a outros nós)



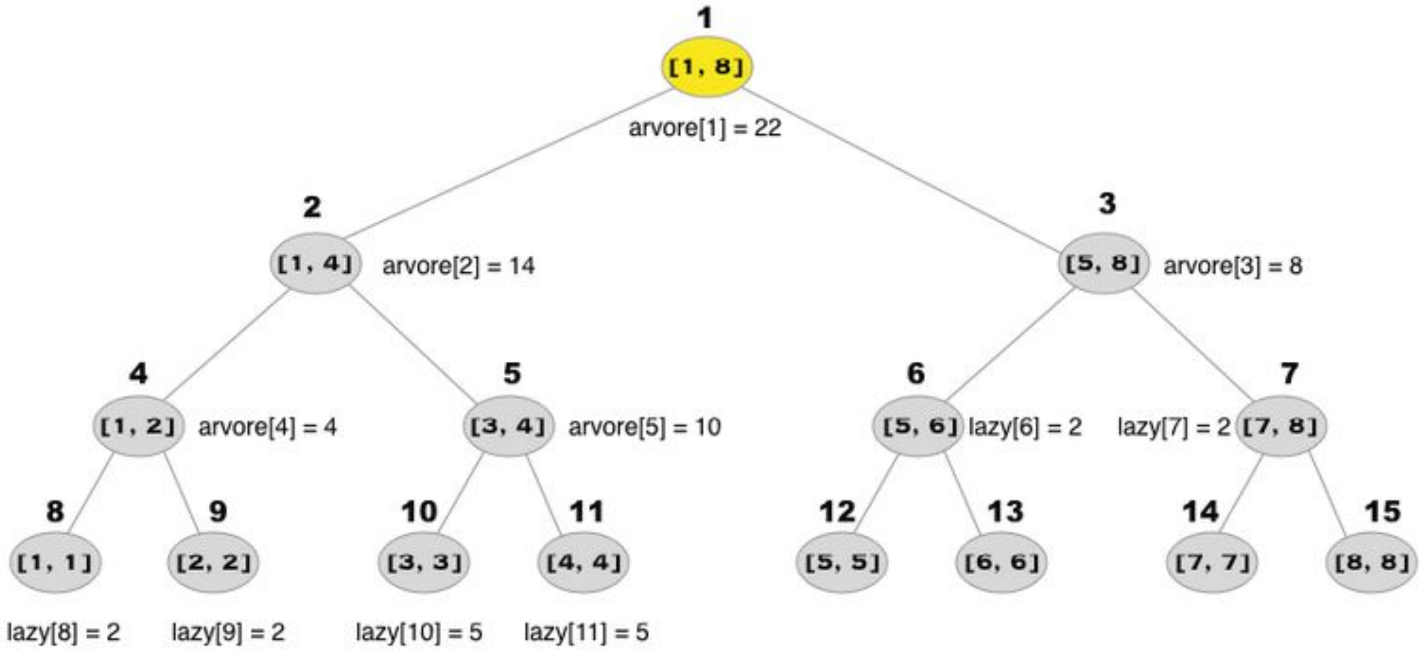
Lazy Propagation

- Depois de atualizar o intervalo desejado, voltamos na recursão, atualizando os pais como sendo a soma dos valores de seus filhos.



Lazy Propagation

- Depois de atualizar o intervalo desejado, voltamos na recursão, atualizando os pais como sendo a soma dos valores de seus filhos.



Lazy Propagation

- A chave da implementação da SegTree com Lazy Propagation é a função que realiza a propagação:

```
void propagate(int no, int i, int j){
    if (has[no]){
        st[no] += lazy[no] * (j - i + 1);
        if (i != j){
            lazy[no*2] = lazy[no*2 + 1] = lazy[no];
            has[no*2] = has[no*2 + 1] = true;
        }
        has[no] = false;
    }
}
```

Lazy Propagation

- Para demais detalhes da implementação, consulte: https://github.com/UnBalloon/programacao-competitiva/tree/master/Segment%20Trees%20(%C3%81rvores%20de%20segmento)

Referências

https://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/

https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/

http://www.codcad.com/lesson/53

http://www.codcad.com/lesson/60

https://github.com/UnBalloon/programacao-competitiva/tree/master/Segment%20Trees%20(%C3%81rvores%20de%20segmento)

https://github.com/icmcgema/gema/blob/master/11-Arvore_de_Segmentos.md

https://cp-algorithms.com/data_structures/segment_tree.html

https://linux.ime.usp.br/~matheusmso/mac0499/poster.pdf

https://neps.academy/lesson/266