Árvore de Segmentos

Definição e Implementação

Prof. Edson Alves - UnB/FGA

2020

Sumário

Definição

Árvore de Segmentos

Definição

Uma **árvore de segmentos** (*segment tree*) é uma estrutura de dados que tem suporte para duas operações sobre um vetor xs de N elementos: realizar uma consulta sobre um subintervalo de índices $[i, j]$ (`range_query(i, j)`) e atualizar o valor de $xs[i]$ (`update(i, value)`), ambas com complexidade $O(\log N)$.

Características da árvore de segmentos

- Uma árvore de segmentos é uma árvore binária completa cujos nós intermediários armazenam os resultados da operação subjacente sobre um subintervalos de índices $[i, j]$, e cujas folhas são os elementos do vetor xs
- Se um nó intermediário armazena o resultado da operação para $[i, j]$, seu filho à esquerda armazena os resultados para $[i, i+m)$, e seu filho à direita armazena os resultados para $[i+m, j]$, onde $m = \lfloor (j-i+1)/2 \rfloor$
- As árvores de segmentos são estruturas mais flexíveis do que as árvores de Fenwick
- Por outro lado, elas são mais difíceis de implementar e precisam de mais memória

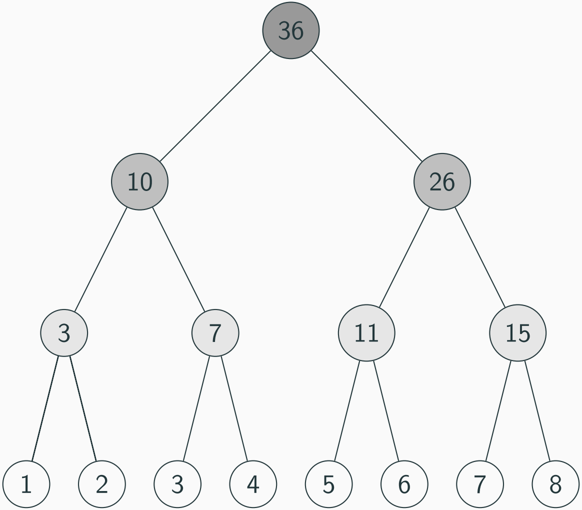
Operação subjacente

- Cada árvore de segmentos tem uma operação binária subjacente $\odot$
- Esta operação tem a propriedade de que, se $[a, c] = [a, b) \cup [b, c]$, $\odot([a, c])$ pode ser computado diretamente a partir dos valores de $\odot([a, b))$ e $\odot([b, c])$, isto é,

$$\odot([a, c]) = f(\odot([a, b)), \odot([b, c]))$$

- As operações subjacentes mais comuns são:
    (a) soma dos elementos do intervalo
    (b) elemento mínimo do intervalo
    (c) elemento máximo do intervalo
    (d) ou exclusivo dos elementos do intervalo
- Outras operações também podem ser implementadas, desde que possuam a propriedade supracitada

Visualização de uma árvore de segmentos para soma dos elementos



Implementação *bottom-up*

Número total de nós

- Suponha que $N = 2^k$, para algum k positivo
- Assim, o nível i da árvore terá 2^i nós que representam um intervalo de tamanho $N/2^i$
- A altura h da árvore é igual a $h = \log N = \log 2^k = k$
- Logo, o total de nós da árvore será igual a

$$\sum_{i=0}^{k} 2^i = 1 + 2 + \ldots + 2^k = 2^{k+1} - 1 < 2^{k+1} = 2N$$

- Assim, a árvore de segmentos deve reservar espaço para $2N$ nós

Construtor

- Em um implementação *bottom-up*, os nós da árvore de segmentos são armazenados em um vetor ns de $2N$ elementos, do mesmo tipo dos elementos de xs
- A posição 0 (zero) não é utilizada, sendo a raiz armazenada no índice 1 (um)
- Seja u o nó que ocupa o índice i de ns
- O filho a esquerda de u ocupará o índice $2i$, e o filho à direita o índice $2i + 1$
- O pai de u ocupará o índice $\lfloor i/2 \rfloor$
- Os elementos de xs ocuparam os índices de N até $2N - 1$
- Das folhas até a raiz, um nível por vez, serão preenchidos os nós internos, usando a operação subjacente

Visualização do construtor da árvore de segmentos

Operação subjacente: soma dos elementos

$xs =$

7	−3	−5	2

$ns =$

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Visualização do construtor da árvore de segmentos

Copia dos elementos de xs para as folhas

$$xs = \begin{array}{|c|c|c|c|} \hline 7 & -3 & -5 & 2 \\ \hline \end{array}$$

$$ns = \begin{array}{c|c|c|c|c|c|c|c} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} & \mathbf{7} \\ \hline & & & & 7 & -3 & -5 & 2 \\ \hline \end{array}$$

Visualização do construtor da árvore de segmentos

Preenchimento do nível intermediário

$xs =$

7	-3	-5	2

$ns =$

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
			-3	7	-3	-5	2

Visualização do construtor da árvore de segmentos

Preenchimento do nível intermediário

$xs =$

7	-3	-5	2

$ns =$

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
		4	-3	7	-3	-5	2

Visualização do construtor da árvore de segmentos

Preenchimento da raiz

$xs =$

7	-3	-5	2

$ns =$

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	1	4	-3	7	-3	-5	2

Implementação do construtor da árvore de segmentos

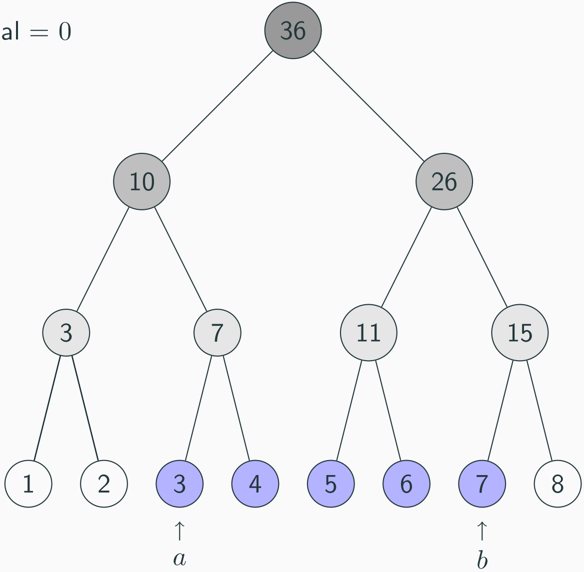
```cpp
#ifndef SEGMENT_TREE_H
#define SEGMENT_TREE_H

#include <vector>
#include <algorithm>

template<typename T>
class SegmentTree
{
    int N;
    std::vector<T> ns;

public:
    SegmentTree(const std::vector<T>& xs) : N(xs.size()), ns(2*N, -1)
    {
        std::copy(xs.begin(), xs.end(), ns.begin() + N);

        for (int i = N - 1; i > 0; --i)
            ns[i] = ns[2*i] + ns[2*i + 1];
    }

```

Range query

- Uma vez inicializada a árvore de segmentos, é possível determinar o resultado da operação subjacente para um intervalo $[a, b]$ arbitrário
- Para isto, três observações devem ser feitas:
    1. Se a é ímpar, ele é o filho à direita, logo ele deve ser processado separadamente
    2. O mesmo acontece se b é par
    3. Nos outros casos, os valores de a e b já foram processados por seus pais, e o processamento deve seguir para estes pais
- Assim, como a altura da árvore é igual a $\log N$, esta rotina tem complexidade $O(\log N)$
- Se a operação subjacente é a soma dos elementos, esta operação recebe o nome de *range sum query* (RSQ)

Visualização de RSQ(2, 6)

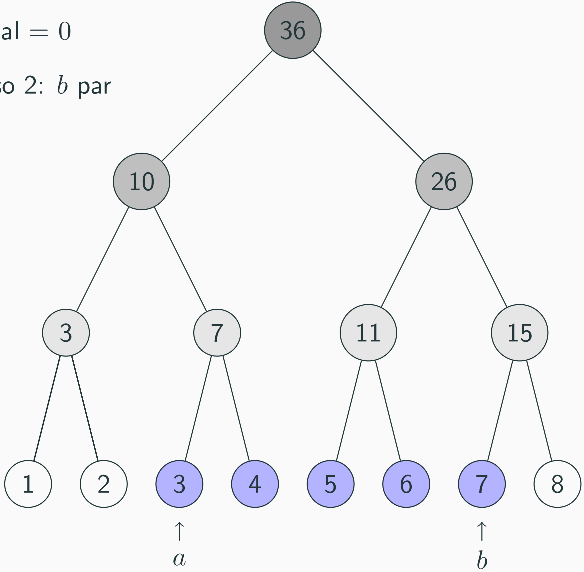
Total $= 0$



Visualização de RSQ(2, 6)

Total $= 0$

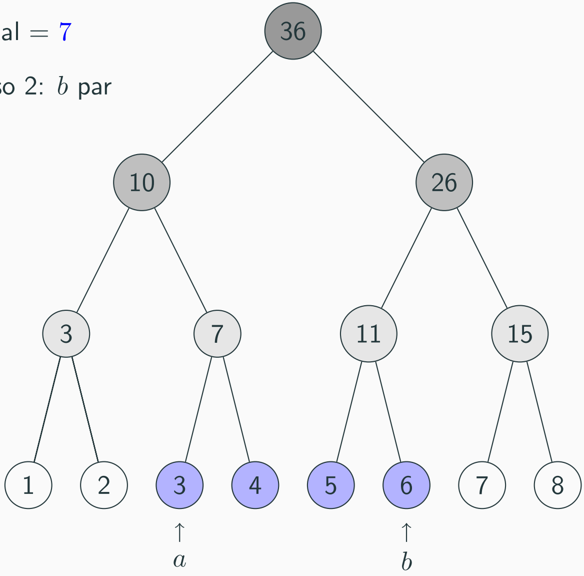
Caso 2: b par



Visualização de RSQ(2, 6)

Total = $\color{blue}{7}$

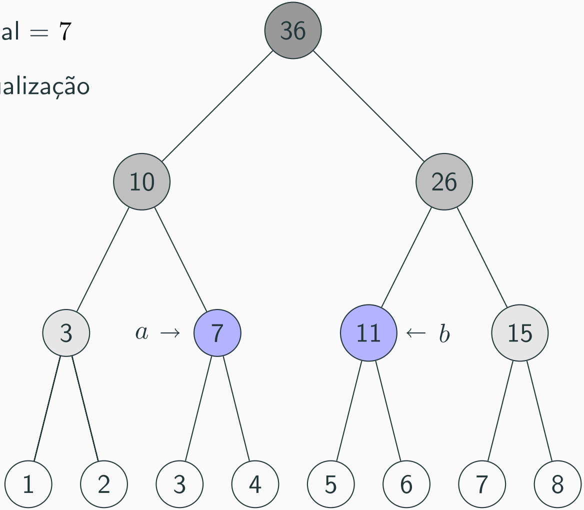
Caso 2: b par



Visualização de RSQ(2, 6)

Total $= \mathbf{7}$

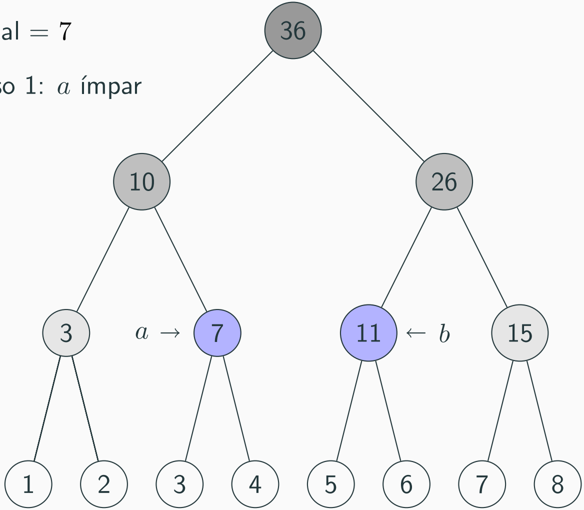
Atualização



Visualização de RSQ(2, 6)

Total $= \mathbf{7}$

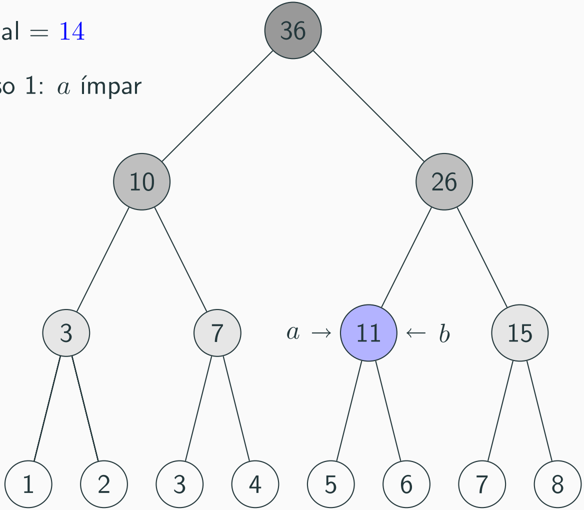
Caso 1: a ímpar



Visualização de RSQ(2, 6)

Total $= 14$

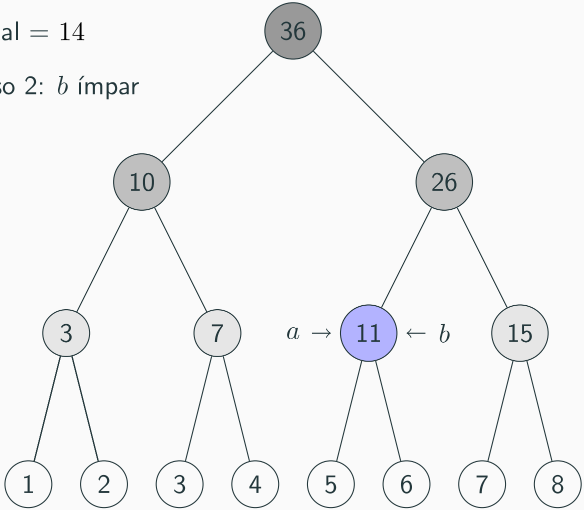
Caso 1: a ímpar



Visualização de RSQ(2, 6)

Total $= 14$

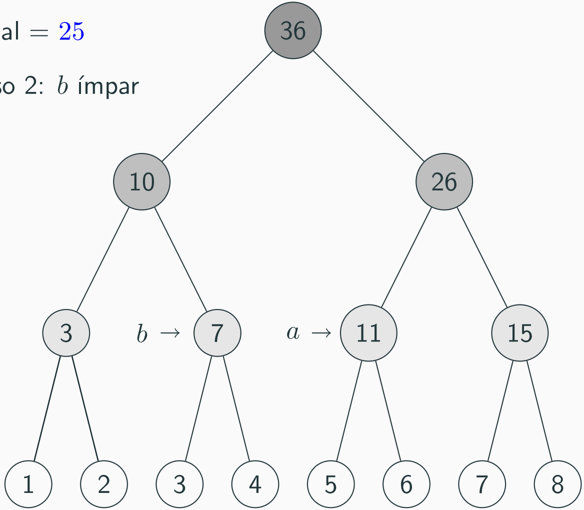
Caso 2: b ímpar



Visualização de RSQ(2, 6)

Total $= \color{blue}{25}$

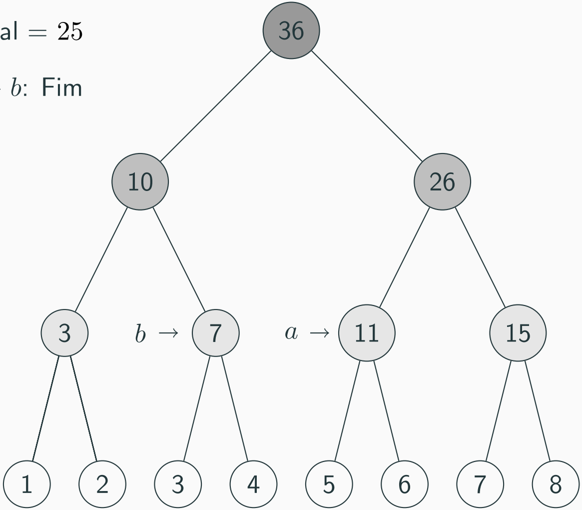
Caso 2: b ímpar



Visualização de RSQ(2, 6)

Total $= 25$

$a > b$: Fim



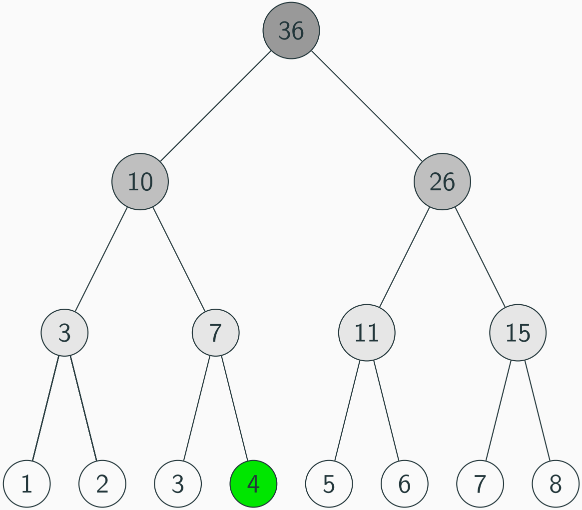
Implementação da *range query*

```
    T RSQ(int i, int j)
    {
        // As folhas estão na segunda metade de ns
        int a = i + N, b = j + N;
        T s = 0;

        while (a <= b)
        {
            if (a & 1)
                s += ns[a++];

            if (not (b & 1))
                s += ns[b--];

            a /= 2;
            b /= 2;
        }

        return s;
    }

```

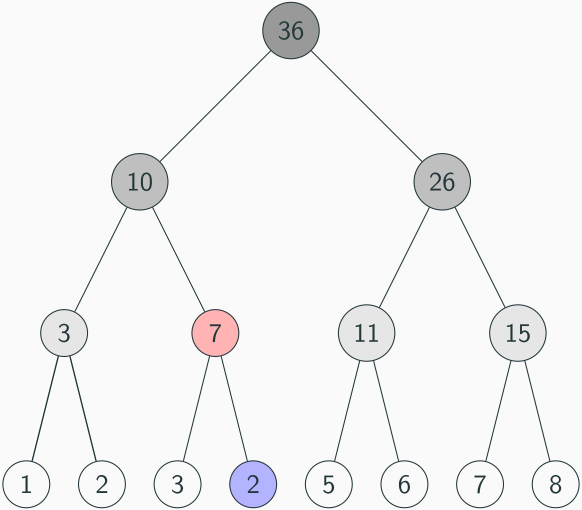
Update

- A operação de atualização (`update(i, value)`) permite modificar o valor do elemento $ns[i]$
- O procedimento padrão é aplicar a operação subjacente ao atual valor de $ns[i]$ e um parâmetro `value`
- Uma variante comum é a substituição do valor
- Neste caso, é preciso determinar qual seria o valor atual x e então aplicar a atualização com o parâmetro `value` igual ao inverso de x em relação à operação subjacente
- Uma vez modificado o valor, é preciso ir atualizado todos seus antepassados na árvore: pai, avô, etc, até a raiz
- Como a altura da árvore é igual a $\log N$, esta operação também tem complexidade $O(\log N)$

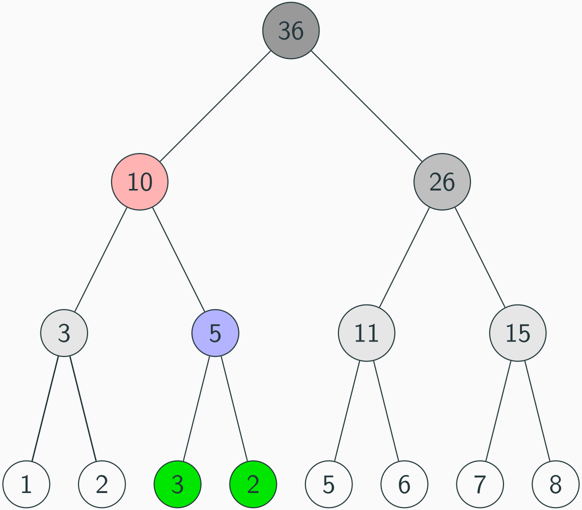
Visualização de update(3, -2)



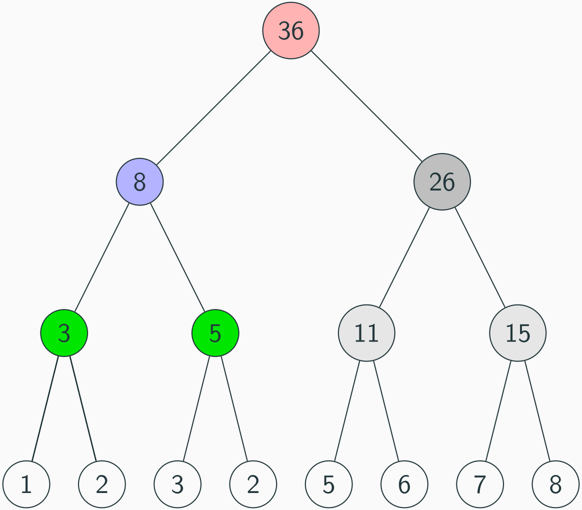
Visualização de update(3, -2)



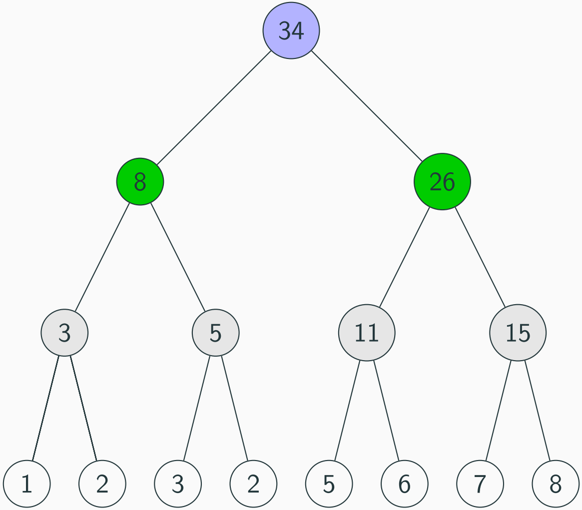
Visualização de update(3, -2)



Visualização de update(3, -2)



Visualização de update(3, -2)



Implementação de *update*

```
    void update(int i, T value)
    {
        int a = i + N;

        ns[a] += value;

        // Atualiza todos os pais de a
        while (a >>= 1)
            ns[a] = ns[2*a] + ns[2*a + 1];
    }
};

#endif
```

Implementação *top-down*

Número arbitrário de nós

- Na implementação *bottom-up* foi assumido que o tamanho N do vetor xs era uma potência de 2
- No caso geral, N pode ser um inteiro positivo qualquer
- A implementação *top-down* é uma alternativa para estes casos
- Se N não é uma potência de 2, a próxima potência de 2 maior do que N é menor do que $2N$
- Assim uma cota superior segura para o tamanho do vetor ns é de $4N$
- O preenchimento de ns é feito por meio de N chamadas de `update(i, xs[i])`
- Este forma não é ótima em termos de complexidade, mas reusa código e diminui o tamanho da implementação

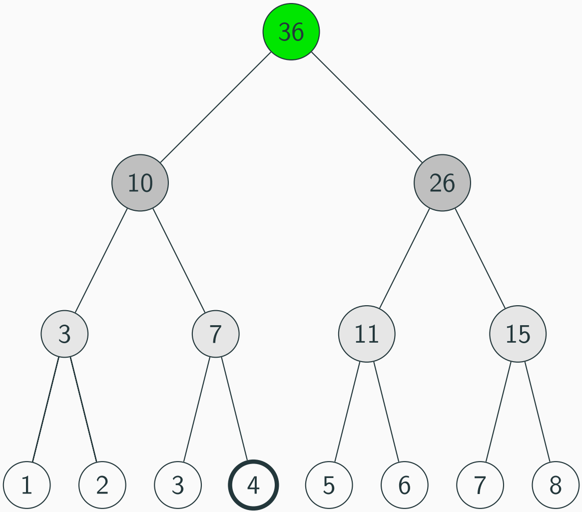
Implementação do construtor na versão *top-down*

```cpp
#ifndef SEGMENT_TREE_H
#define SEGMENT_TREE_H

#include <vector>

template<typename T>
class SegmentTree
{
    int N;
    std::vector<T> ns;

public:
    SegmentTree(const std::vector<int>& xs) : N(xs.size()), ns(4*N)
    {
        for (size_t i = 0; i < xs.size(); ++i)
            update(i, xs[i]);
    }

```

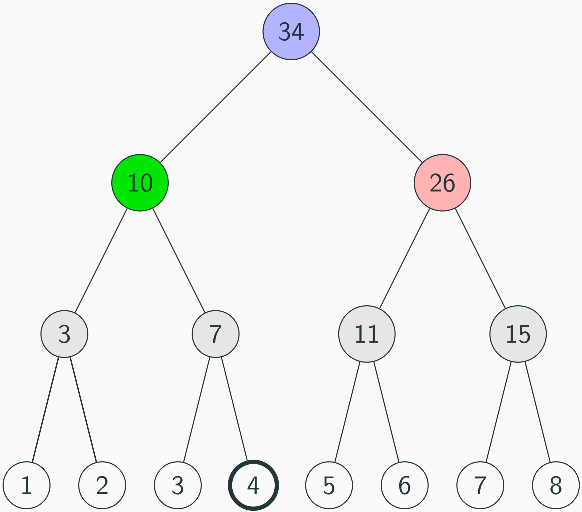
Update

- A atualização deve ser feita por meio de recursão
- Os parâmetros da versão recursiva devem ser: o índice do nó atual (node), o intervalo que o nó representa (`[L, R]`), o índice do elemento a ser atualizado em xs (`i`) e o valor da atualização (`value`)
- Há dois casos base: o primeiro acontece se `i` não pertencer ao intervalo `[L, R]`, onde a função deve retornar sem fazer nada
- Caso contrário, o valor armazenado em node deve ser atualizado usando `value`
- Em seguida há o segundo caso base: se node aponta para um folha, a função também retorna
- As chamadas recursivas repassam a atualização para os filhos à esquerda e à direita de node

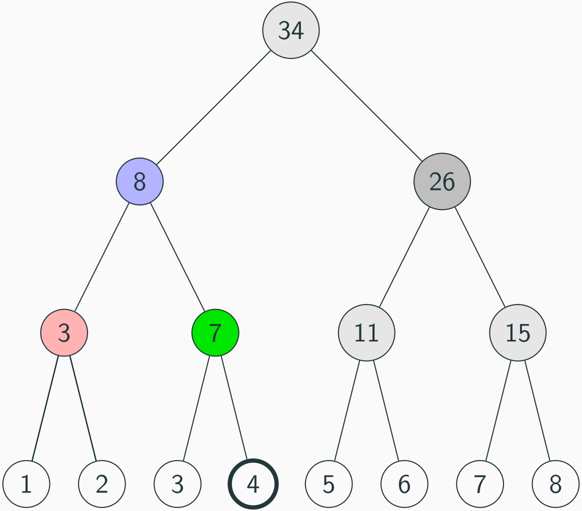
Visualização de update(3, -2)



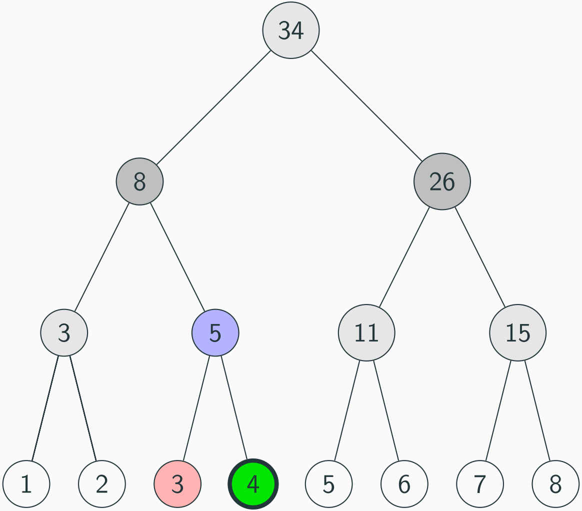
Visualização de update(3, -2)



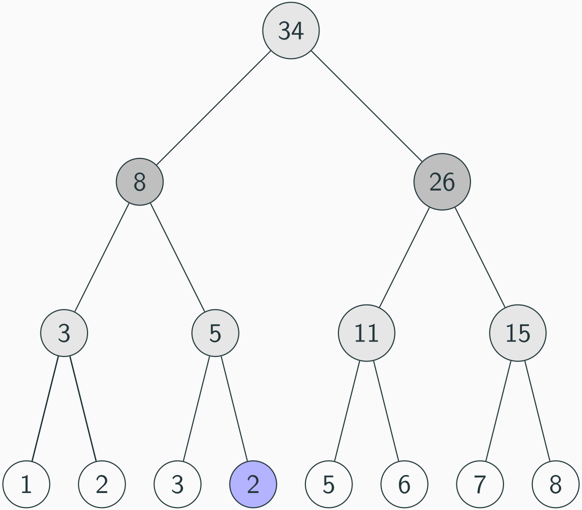
Visualização de update(3, -2)



Visualização de update(3, -2)



Visualização de update(3, -2)



Implementação de *update* na versão *top-down*

```
    void update(int i, T value)
    {
        update(1, 0, N - 1, i, value);
    }

private:
    void update(int node, int L, int R, int i, T value)
    {
        // Caso base: i não pertence ao intervalo [L, R]
        if (i > R or i < L)
            return;

        ns[node] += value;

        // Caso base: node é uma folha
        if (L == R)
            return;

        update(2*node, L, (L+R)/2, i, value);
        update(2*node + 1, (L+R)/2 + 1, R, i, value);
    }
```

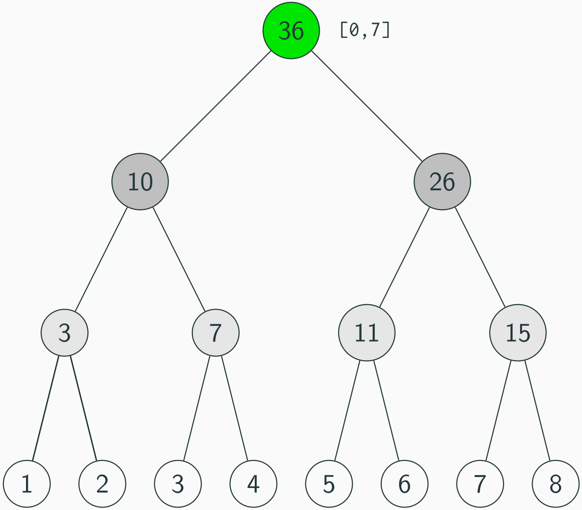
- As *range queries* também podem ser respondidas por meio de recursão
- A chamada `range_query(a, b)` retorna o valor da operação subjacente em todos os elementos de xs cujos índices estão no intervalo $[a, b]$
- Para tal, a chamada recursiva deve ter como parâmetros o nó atual (node), o intervalo $[L, R]$ que o nó representa e o intervalo $[a, b]$
- O primeiro caso base acontece quando $[L, R] \cap [a, b] = \emptyset$, onde a função deve retornar o elemento neutro da operação
- O segundo caso base acontece quando $[L, R] \subset [a, b]$: neste caso, o valor armazenado no nó deve ser retornado

- A chamada recursiva computa a *range_query* para as subárvores da esquerda e da direita, e computa o resultado para o nó a partir destes retornos
- A complexidade desta operação é $O(\log N)$
- Isto porque, na decomposição

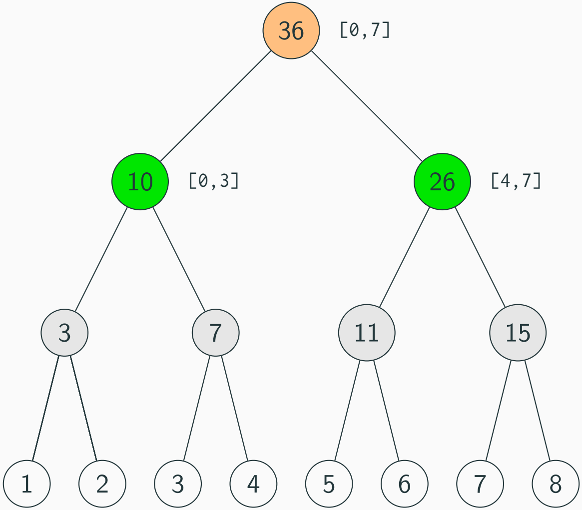
$$[a, b] = [x_1, y_1] \cap [x_2, y_2] \cap \ldots \cap [x_k, y_k],$$

onde $[x_i, y_i]$ é um dos intervalos que aparecem na árvore e $[x_i, y_i] \cap [x_j, y_j] = \emptyset$ se $i \neq j$, pode haver no máximo um intervalo de cada nível da árvore

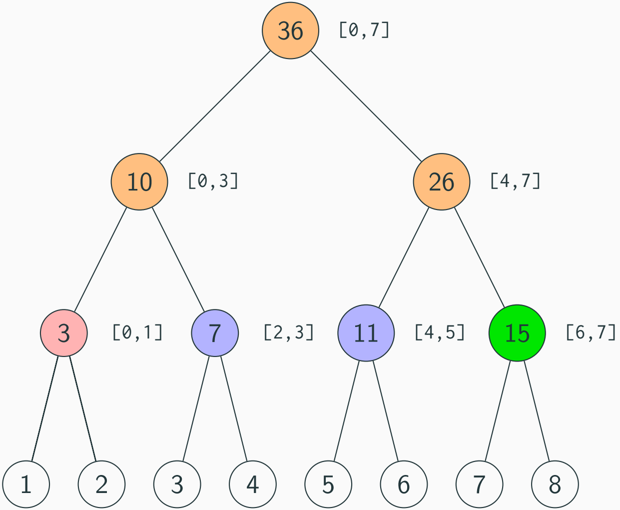
Visualização de RSQ(2, 6)



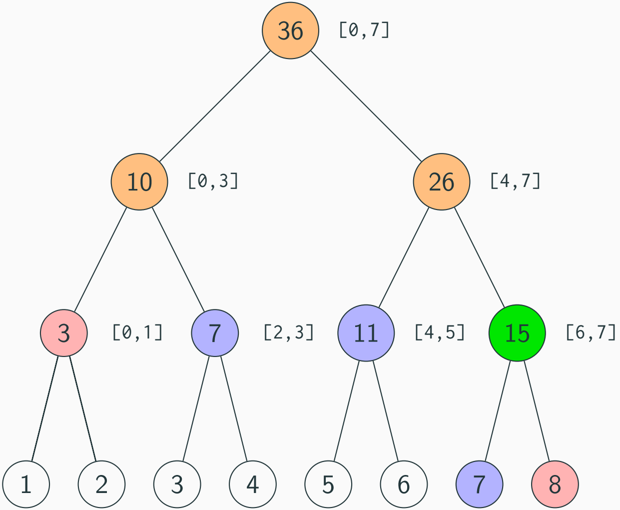
Visualização de RSQ(2, 6)



Visualização de RSQ(2, 6)



Visualização de RSQ(2, 6)



Implementação de *update* na versão *top-down*

```
public:
    T RSQ(int a, int b)
    {
        return RSQ(1, 0, N - 1, a, b);
    }

private:
    T RSQ(int node, int L, int R, int a, int b)
    {
        if (a > R or b < L)          // [a, b] e [L, R] = vazio
            return 0;

        if (a <= L and R <= b)       // [a, b] c [L, R]
            return ns[node];

        T x = RSQ(2*node, L, (L + R)/2, a, b);
        T y = RSQ(2*node + 1, (L + R)/2 + 1, R, a, b);

        return x + y;
    }

```

Variantes da árvore de segmentos

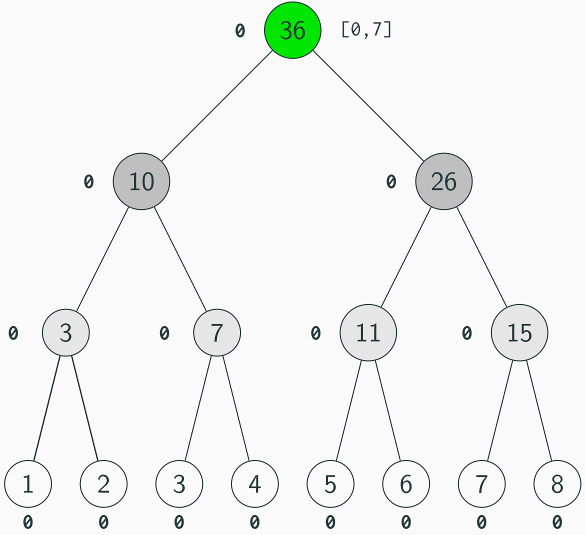
Range update

- Uma variante comum dentre as operação de uma árvore de segmentos é a atualização de intervalo (*range_update*)
- A chamada `range_update(a, b, value)` aplica o parâmetro `value` a todos os elementos de xs cujos índices pertencem ao intervalo $[a, b]$
- Implementada diretamente, de forma semelhante a atualização pontual, esta função passa a ter complexidade $O(N)$ no pior caso, o que torna a árvore de segmentos irrelevante, pois esta seria a complexidade de ambas operações em uma implementação *naive* utilizando apenas vetores
- Para implementar a atualização de intervalo com complexidade $O(\log N)$, é preciso utilizar uma técnica denominada *lazy propagation*

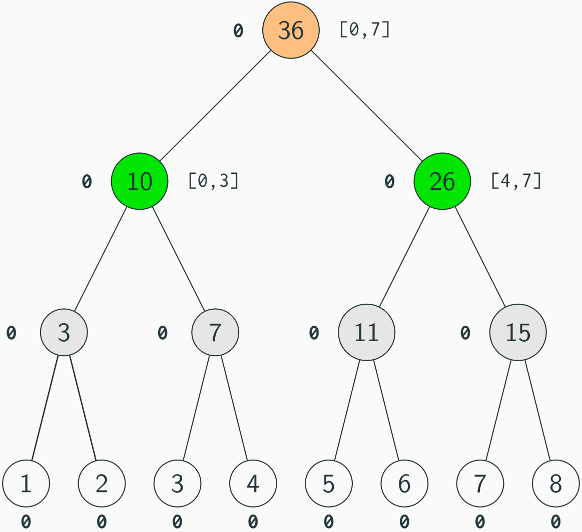
Lazy propagation

- A valoração não-estrita (*lazy propagation*) é uma técnica oriunda das linguagens funcionais, onde um valor só é computado quando for estritamente necessário
- No caso das árvores de segmentos, a ideia é adicionar um campo extra em cada nó, que armazenará os valores que deveriam ser atualizados no nó (vetor `lazy`)
- A menos que o valor armazenado no nó seja necessário para algum cálculo, o valor de `lazy` fica armazenado
- Se for preciso saber o valor correto do nó, o valor atual é corrigido pelo valor de `lazy`, e os valores de `lazy` dos filhos à esquerda e à direita são atualizados
- Embora a memória necessária para a árvore de segmentos aumente para quase o dobro, a complexidade da atualização de intervalo passa a ser $O(\log N)$

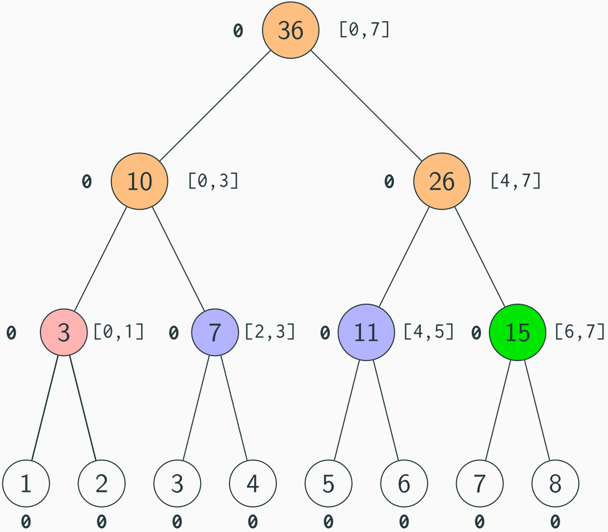
Visualização de update(2, 6, -3)



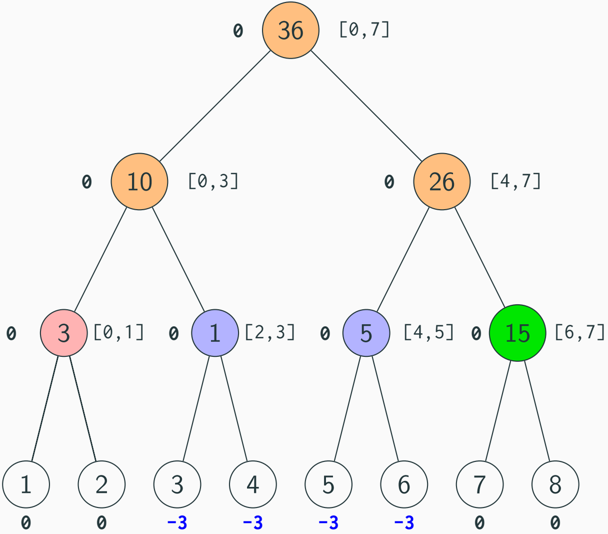
Visualização de update(2, 6, -3)



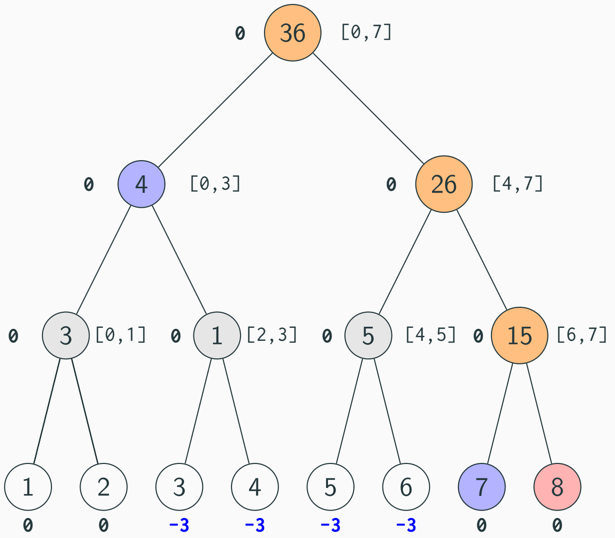
Visualização de update(2, 6, -3)



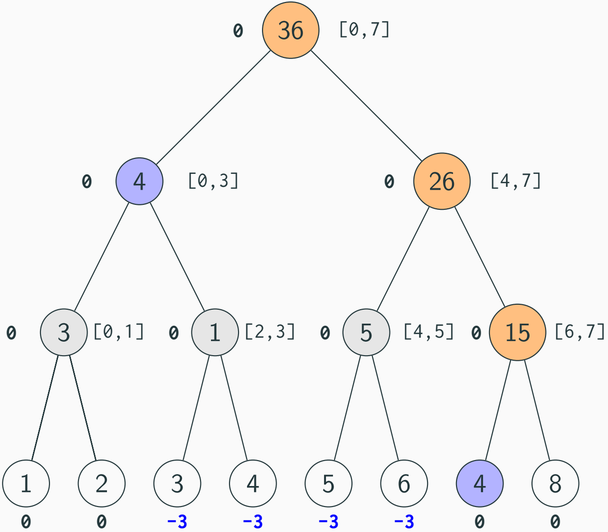
Visualização de update(2, 6, -3)



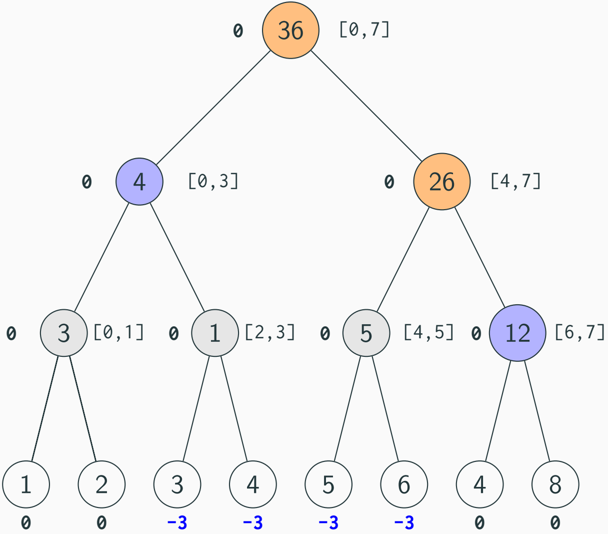
Visualização de update(2, 6, -3)



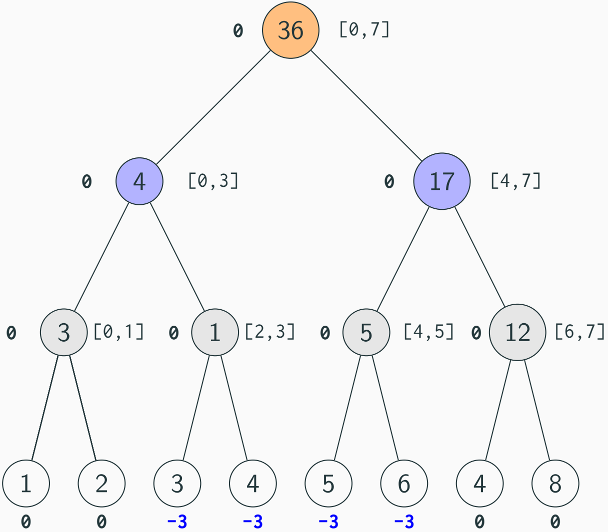
Visualização de update(2, 6, -3)



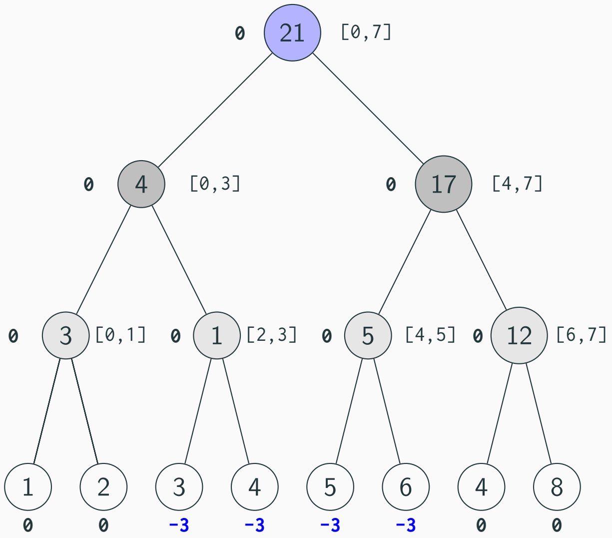
Visualização de update(2, 6, -3)



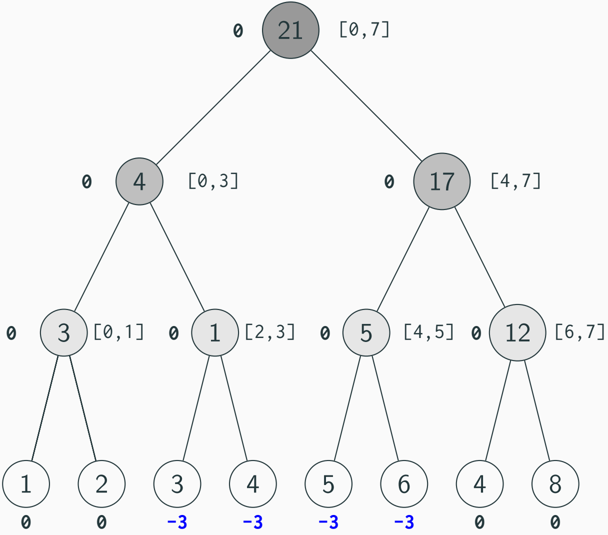
Visualização de update(2, 6, -3)



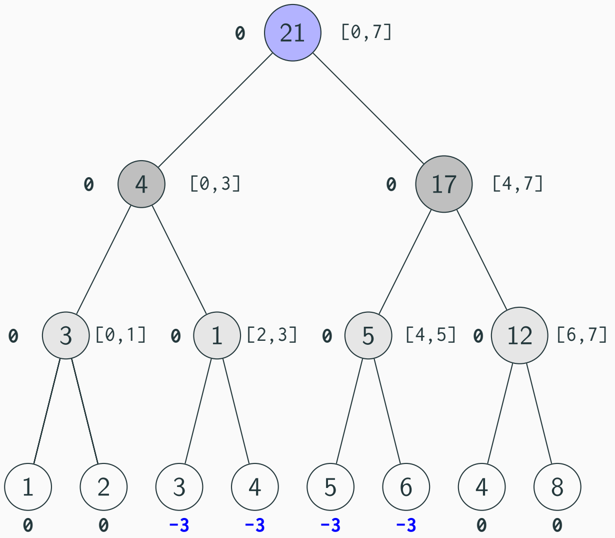
Visualização de update(2, 6, -3)



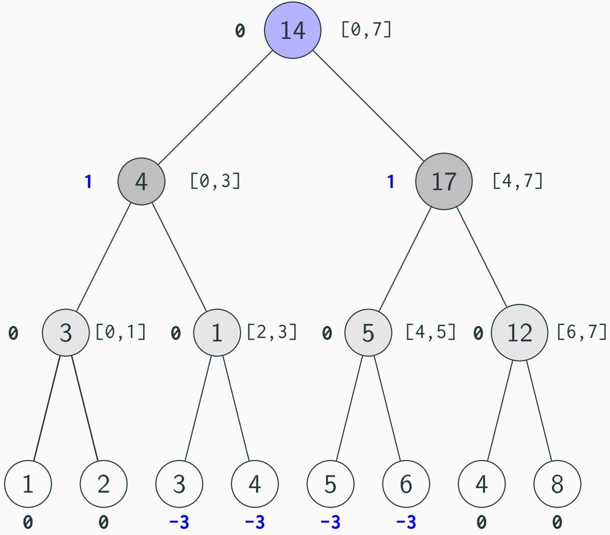
Visualização de update(0, 7, 1)



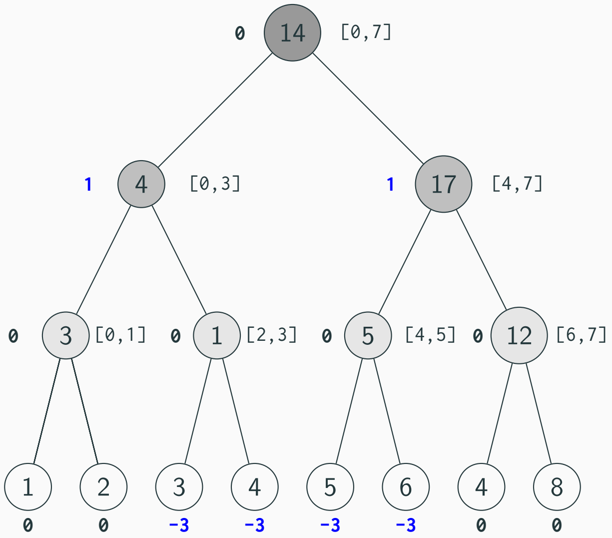
Visualização de update(0, 7, 1)



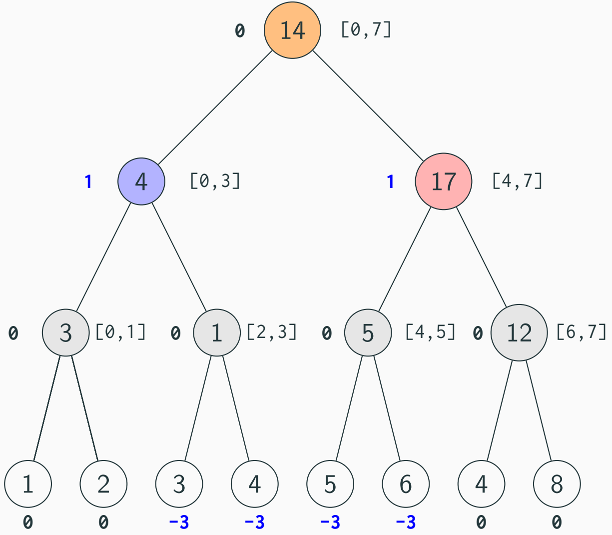
Visualização de update(0, 7, 1)



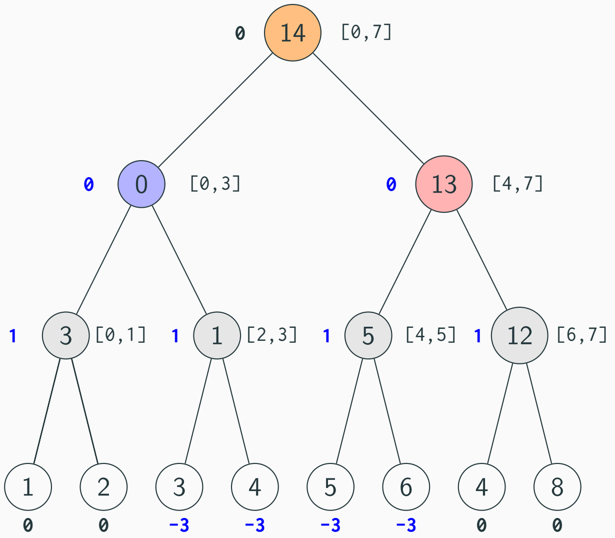
Visualização de update(0, 3, 2)



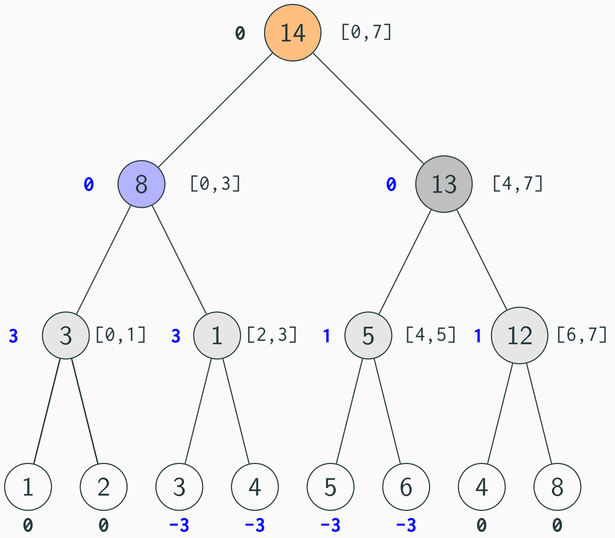
Visualização de update(0, 3, 2)



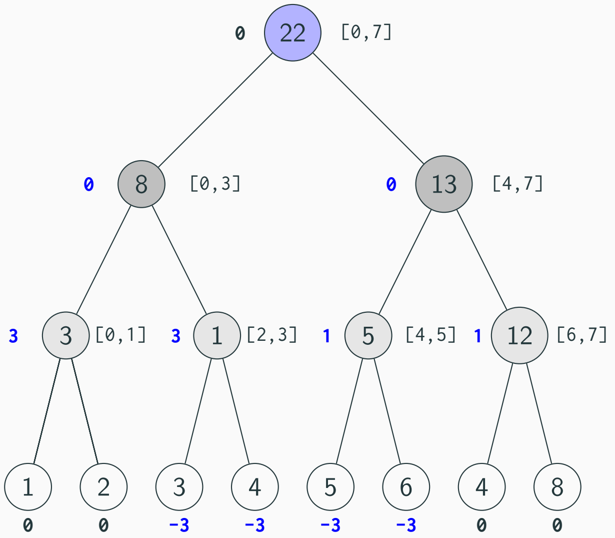
Visualização de update(0, 3, 2)



Visualização de update(0, 3, 2)



Visualização de update(0, 3, 2)



Implementação da *lazy propagation*

```cpp
#ifndef SEGMENT_TREE_H
#define SEGMENT_TREE_H

#include <vector>

template<typename T>
class SegmentTree
{
private:
    int N;
    std::vector<T> ns, lazy;

public:
    SegmentTree(const std::vector<int>& xs)
        : N(xs.size()), ns(4*N), lazy(4*N, 0)
    {
        for (size_t i = 0; i < xs.size(); ++i)
            update(i, i, xs[i]);
    }

```

Implementação da *lazy propagation*

```
    void update(int a, int b, T value)
    {
        update(1, 0, N - 1, a, b, value);
    }

private:
    void update(int node, int L, int R, int a, int b, T value)
    {
        // Lazy propagation
        if (lazy[node])
        {
            ns[node] += (R - L + 1) * lazy[node];

            if (L < R)      // Se o nó não é uma folha, propaga
            {
                lazy[2*node] += lazy[node];
                lazy[2*node + 1] += lazy[node];
            }

            lazy[node] = 0;
        }
```

Implementação da *lazy propagation*

```
        // [a, b] e [L, R] = vazio
        if (a > R or b < L)
            return;

        // [L, R] c [a, b]
        if (a <= L and R <= b)
        {
            ns[node] += (R - L + 1) * value;

            if (L < R)
            {
                lazy[2*node] += value;
                lazy[2*node + 1] += value;
            }

            return;
        }

        update(2*node, L, (L + R)/2, a, b, value);
        update(2*node + 1, (L + R)/2 + 1, R, a, b, value);

```

Implementação da *lazy propagation*

```
        ns[node] = ns[2*node] + ns[2*node + 1];
    }

public:
    T RSQ(int a, int b)
    {
        return RSQ(1, 0, N - 1, a, b);
    }

private:
    T RSQ(int node, int L, int R, int a, int b)
    {
        if (lazy[node])
        {
            ns[node] += (R - L + 1) * lazy[node];

            if (L < R)
            {
                lazy[2*node] += lazy[node];
                lazy[2*node + 1] += lazy[node];
            }
```

Implementação da *lazy propagation*

```
                lazy[node] = 0;
            }

            if (a > R or b < L)
                return 0;

            if (a <= L and R <= b)
                return ns[node];

            T x = RSQ(2*node, L, (L + R)/2, a, b);
            T y = RSQ(2*node + 1, (L + R)/2 + 1, R, a, b);

            return x + y;
        }
};

#endif
```

Referências

1. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
2. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018.