

# Árvore de Segmentos

## Definição e Implementação

---

Prof. Edson Alves - UnB/FGA

2020

1. Definição
2. Implementação *bottom-up*
3. Implementação *top-down*
4. Variantes da árvore de segmentos

# Definição

---

## 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

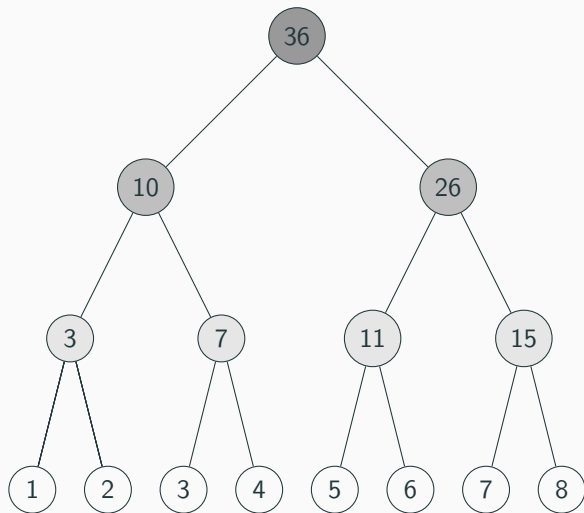
# Operação subjacente

- 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

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

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

- Em uma 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 à 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
---	----	----	---

$ns =$

<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>

# Visualização do construtor da árvore de segmentos

Copia dos elementos de  $xs$  para as folhas

$xs =$	7	-3	-5	2
--------	---	----	----	---

	0	1	2	3	4	5	6	7
$ns =$					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
			-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

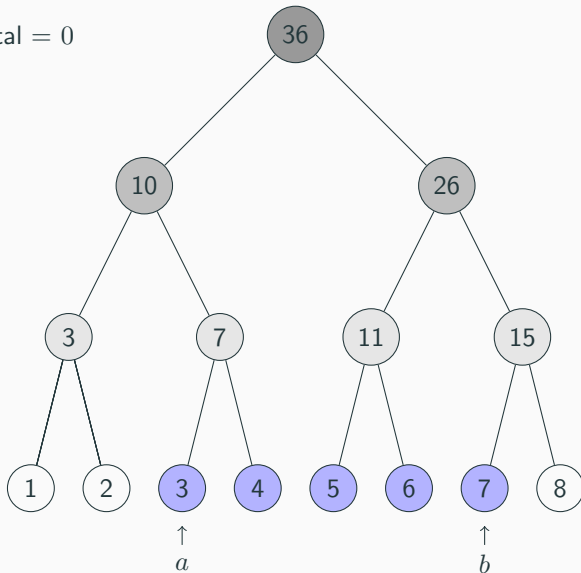
```
1 #ifndef SEGMENT_TREE_H
2 #define SEGMENT_TREE_H
3
4 #include <vector>
5 #include <algorithm>
6
7 template<typename T>
8 class SegmentTree
9 {
10     int N;
11     std::vector<T> ns;
12
13 public:
14     SegmentTree(const std::vector<T>& xs) : N(xs.size()), ns(2*N, 0)
15     {
16         std::copy(xs.begin(), xs.end(), ns.begin() + N);
17
18         for (int i = N - 1; i > 0; --i)
19             ns[i] = ns[2*i] + ns[2*i + 1];
20     }
21
```



- 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: neste caso, será filho à esquerda
  3. Nos outros casos, os valores de  $a$  e  $b$  já foram processados por seus pais, de modo que 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)

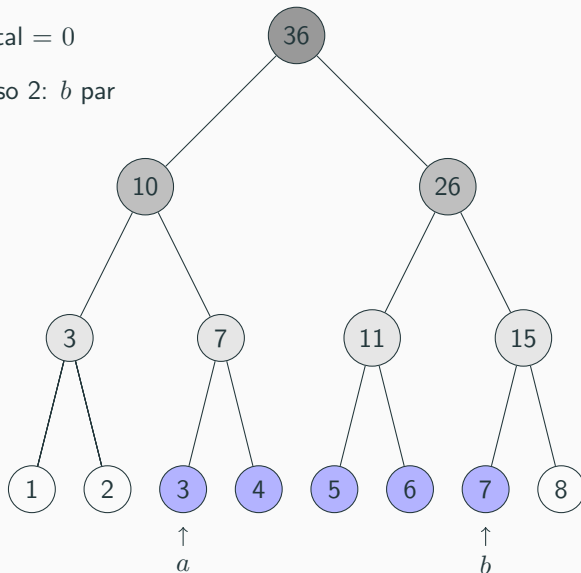
Total = 0



# Visualização de RSQ(2, 6)

Total = 0

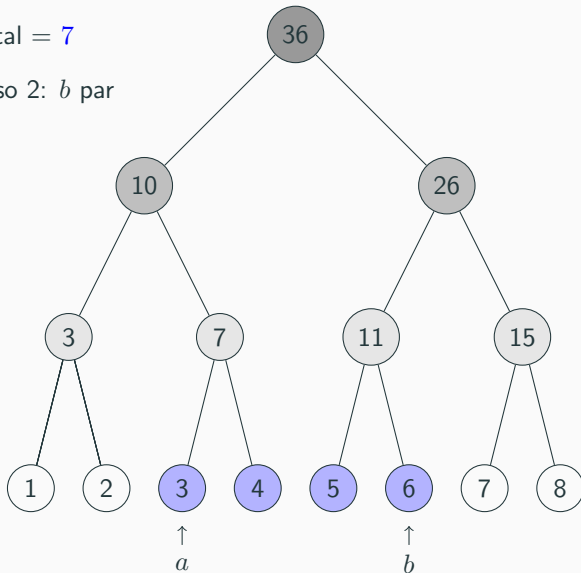
Caso 2:  $b$  par



# Visualização de RSQ(2, 6)

Total = 7

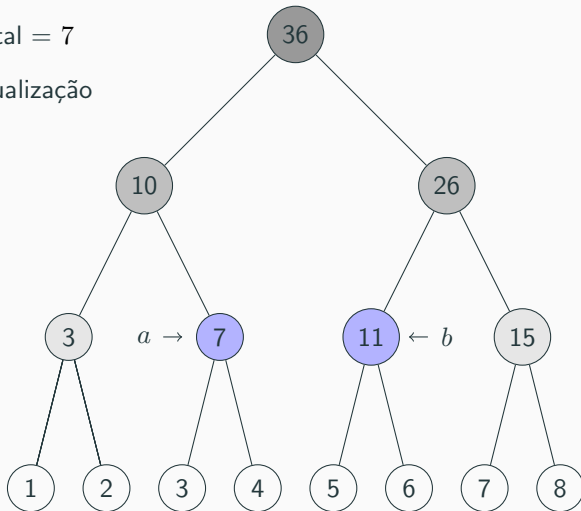
Caso 2:  $b$  par



# Visualização de RSQ(2, 6)

Total = 7

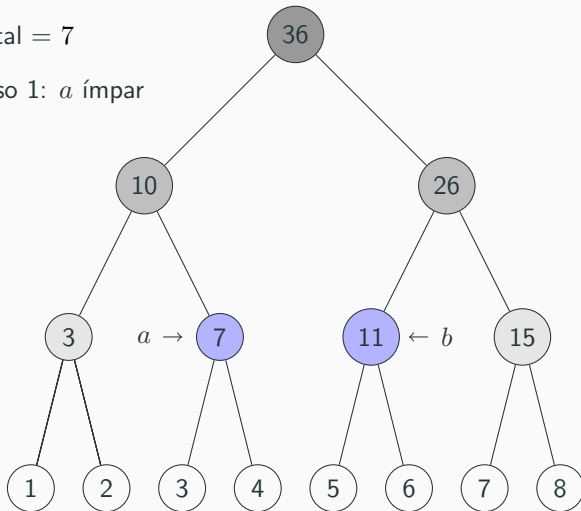
Atualização



# Visualização de RSQ(2, 6)

Total = 7

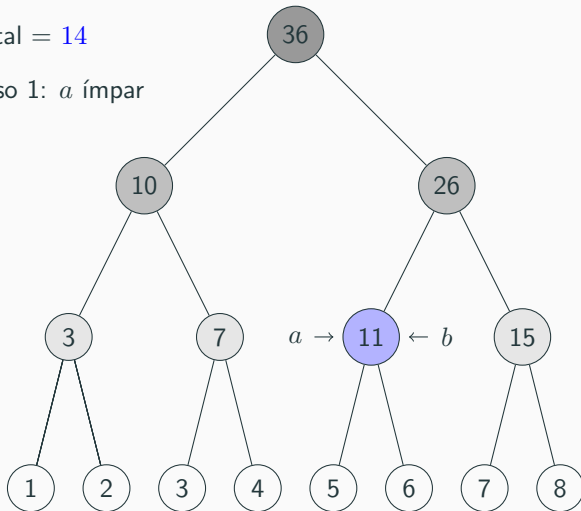
Caso 1:  $a$  ímpar



## Visualização de RSQ(2, 6)

Total = 14

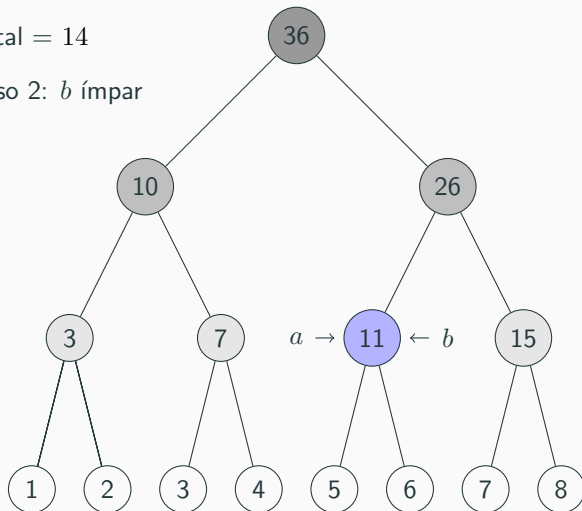
Caso 1:  $a$  ímpar



## Visualização de RSQ(2, 6)

Total = 14

Caso 2:  $b$  ímpar

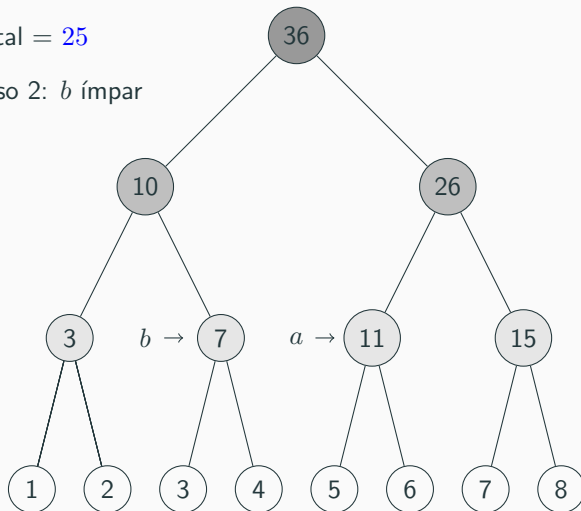




# Visualização de RSQ(2, 6)

Total = 25

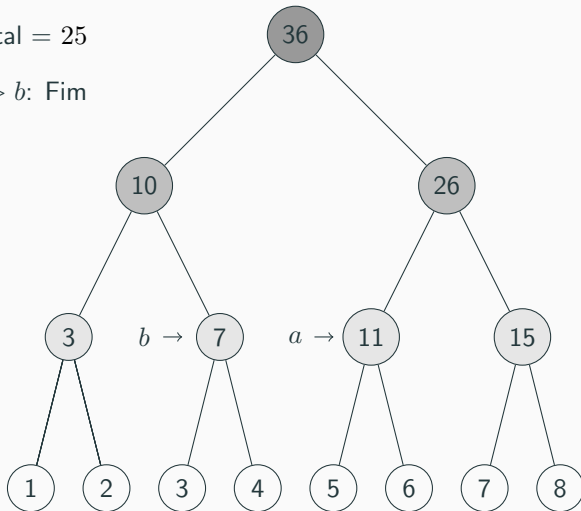
Caso 2:  $b$  ímpar



## Visualização de RSQ(2, 6)

Total = 25

$a > b$ : Fim

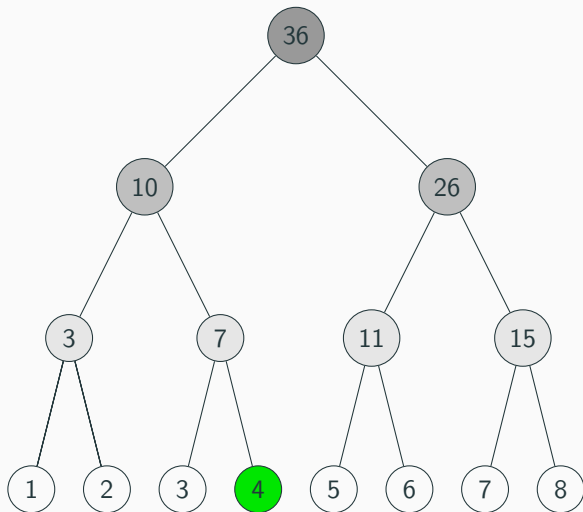


# Implementação da *range query*

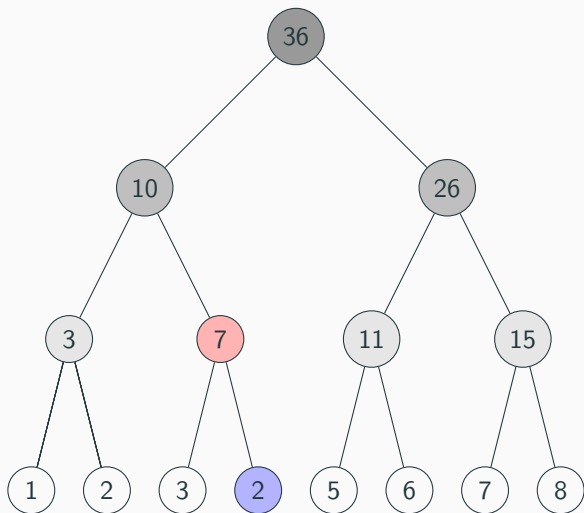
```
22  T RSQ(int i, int j)
23  {
24      // As folhas estão na segunda metade de ns
25      int a = i + N, b = j + N;
26      T s = 0;
27
28      while (a <= b)
29      {
30          if (a & 1)
31              s += ns[a++];
32
33          if (not (b & 1))
34              s += ns[b--];
35
36          a /= 2;
37          b /= 2;
38      }
39
40      return s;
41  }
42
```

- 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 o 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  $O(\log N)$

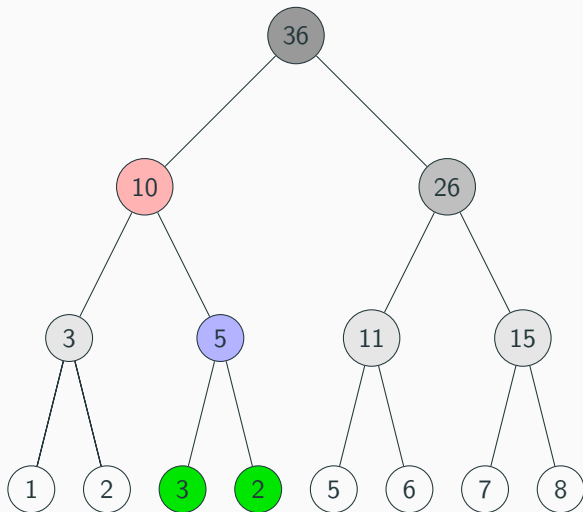
## Visualização de update(3, -2)



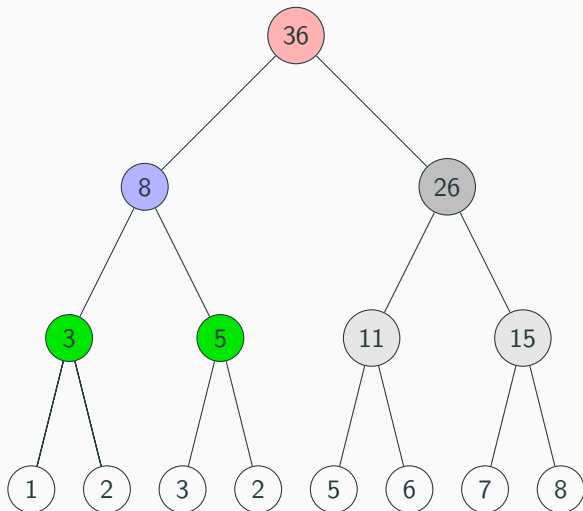
## Visualização de update(3, -2)



## Visualização de update(3, -2)

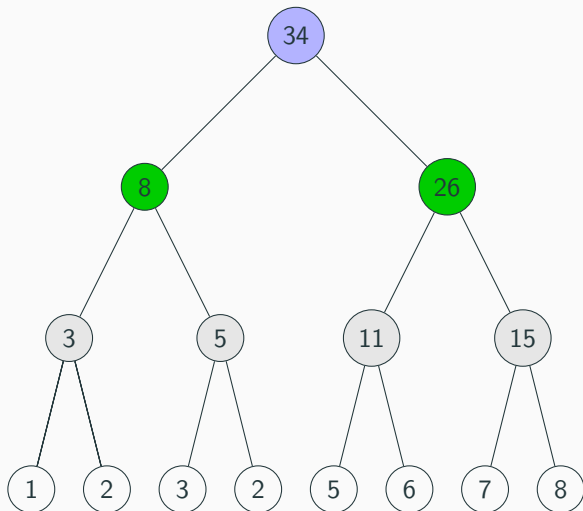


## Visualização de update(3, -2)





## Visualização de update(3, -2)



# Implementação de *update*

```
43 void update(int i, T value)
44 {
45     int a = i + N;
46
47     ns[a] += value;
48
49     // Atualiza todos os pais de a
50     while (a >= 1)
51         ns[a] = ns[2*a] + ns[2*a + 1];
52 }
53 };
54
55 #endif
```

## Implementação *top-down*

---

## Número arbitrário de nós

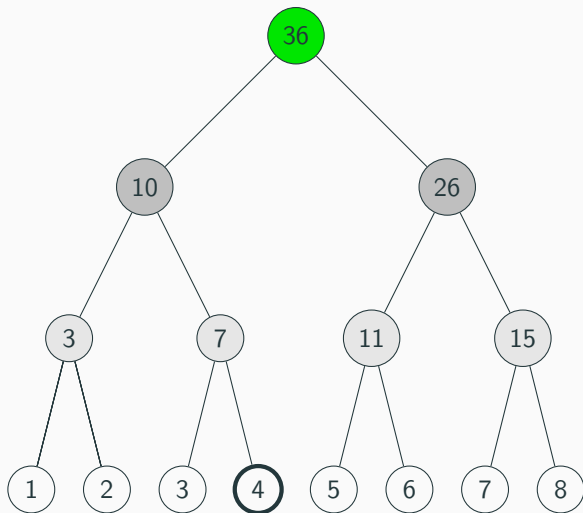
- Na implementação *bottom-up* foi assumido que o tamanho  $N$  do vetor  $xs$  era uma potência de 2
- No caso geral,  $N$  pode ser um inteiro positivo qualquer
- A implementação *top-down* é uma alternativa para estes casos
- Se  $N$  não é uma potência de 2, a próxima potência de 2 maior do que  $N$  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])`
- Esta inicialização não é ótima em termos de memória e de tempo de execução, mas reusa código e diminui o tamanho da implementação

## Implementação do construtor na versão *top-down*

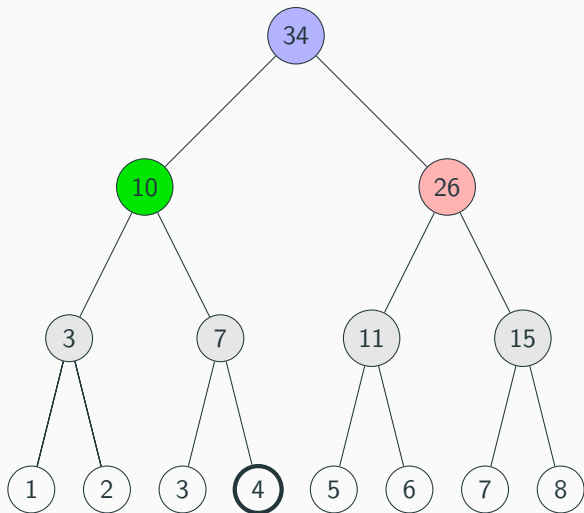
```
1 #ifndef SEGMENT_TREE_H
2 #define SEGMENT_TREE_H
3
4 #include <vector>
5
6 template<typename T>
7 class SegmentTree
8 {
9     int N;
10     std::vector<T> ns;
11
12 public:
13     SegmentTree(const std::vector<int>& xs) : N(xs.size()), ns(4*N)
14     {
15         for (size_t i = 0; i < xs.size(); ++i)
16             update(i, xs[i]);
17     }
18 }
```

- 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 é 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)

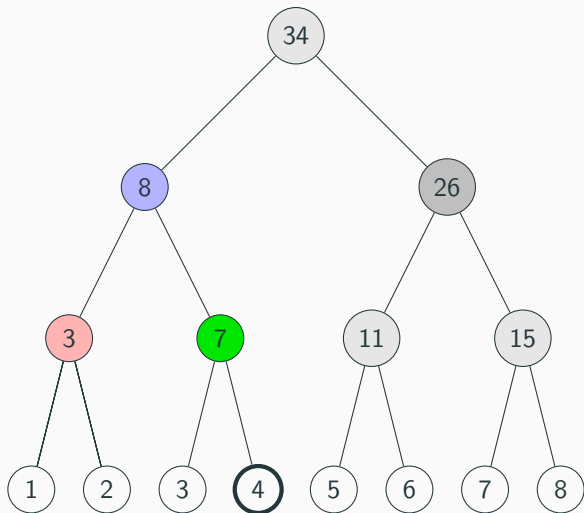


## Visualização de update(3, -2)

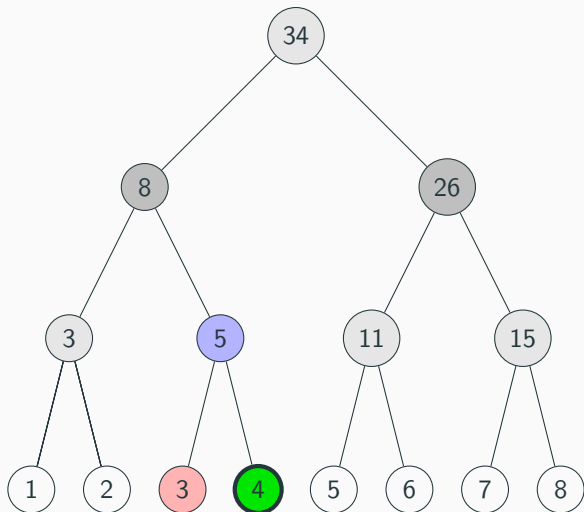




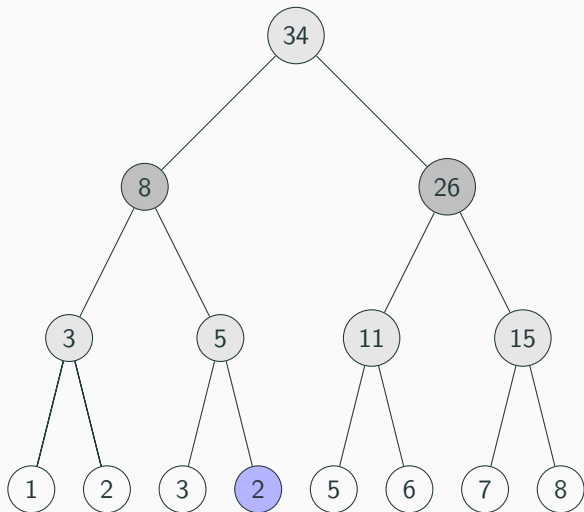
## Visualização de update(3, -2)



## Visualização de update(3, -2)



## Visualização de update(3, -2)



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

```
19 void update(int i, T value)
20 {
21     update(1, 0, N - 1, i, value);
22 }
23
24 private:
25 void update(int node, int L, int R, int i, T value)
26 {
27     // Caso base: i não pertence ao intervalo [L, R]
28     if (i > R or i < L)
29         return;
30
31     ns[node] += value;
32
33     // Caso base: node é uma folha
34     if (L == R)
35         return;
36
37     update(2*node, L, (L+R)/2, i, value);
38     update(2*node + 1, (L+R)/2 + 1, R, i, value);
39 }
```

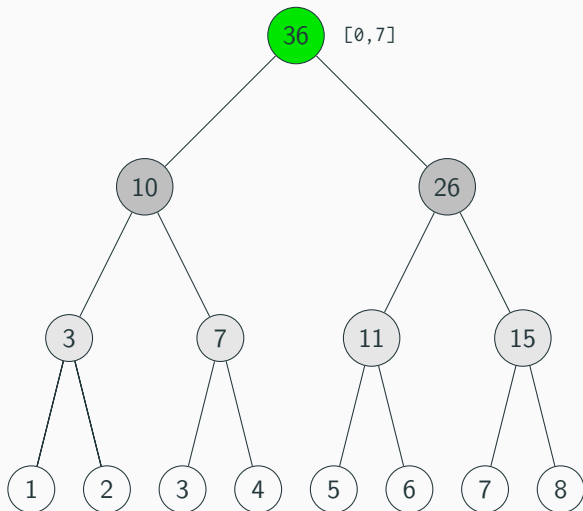
- As *range queries* também podem ser respondidas por meio de recursão
- A chamada `range_query(a, b)` retorna o valor da operação subjacente em todos os elementos de  $xs$  cujos índices estão no intervalo  $[a, b]$
- Para tal, a chamada recursiva deve ter como parâmetros o nó atual (node), o intervalo  $[L, R]$  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

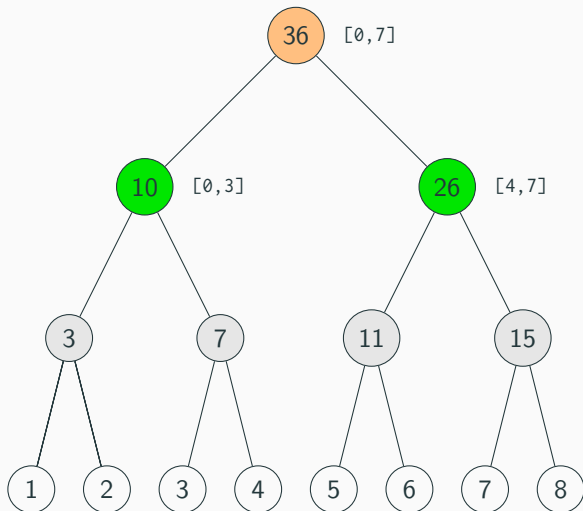
$$[a, b] = [x_1, y_1] \cup [x_2, y_2] \cup \dots \cup [x_k, y_k],$$

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 dois intervalos desta decomposição em cada nível da árvore

## Visualização de RSQ(2, 6)

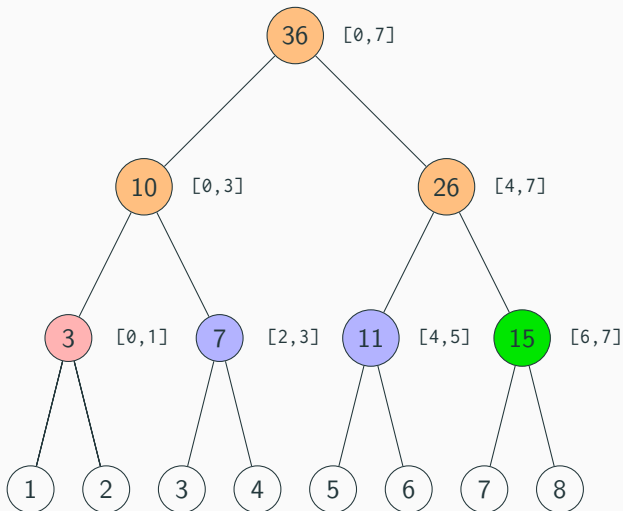


## Visualização de RSQ(2, 6)

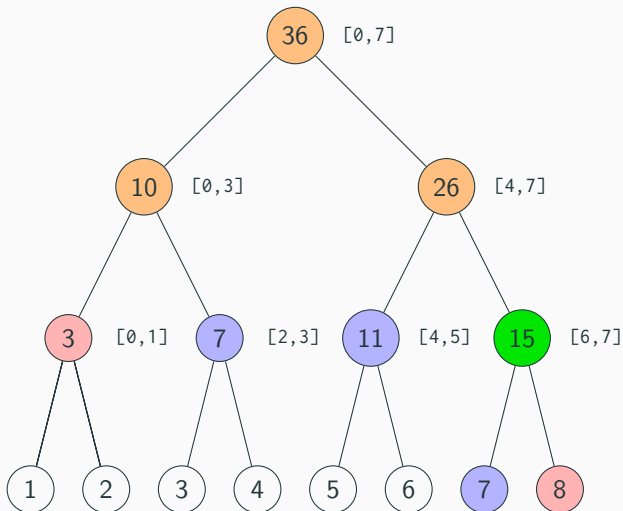




## Visualização de RSQ(2, 6)



## Visualização de RSQ(2, 6)



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

```
41 public:
42     T RSQ(int a, int b)
43     {
44         return RSQ(1, 0, N - 1, a, b);
45     }
46
47 private:
48     T RSQ(int node, int L, int R, int a, int b)
49     {
50         if (a > R or b < L)           //  $[a, b] \cap [L, R] = \{\emptyset\}$ 
51             return 0;
52
53         if (a <= L and R <= b)        //  $[L, R] \subset [a, b]$ 
54             return ns[node];
55
56         T x = RSQ(2*node, L, (L + R)/2, a, b);
57         T y = RSQ(2*node + 1, (L + R)/2 + 1, R, a, b);
58
59         return x + y;
60     }
61
```

## **Variantes da árvore de segmentos**

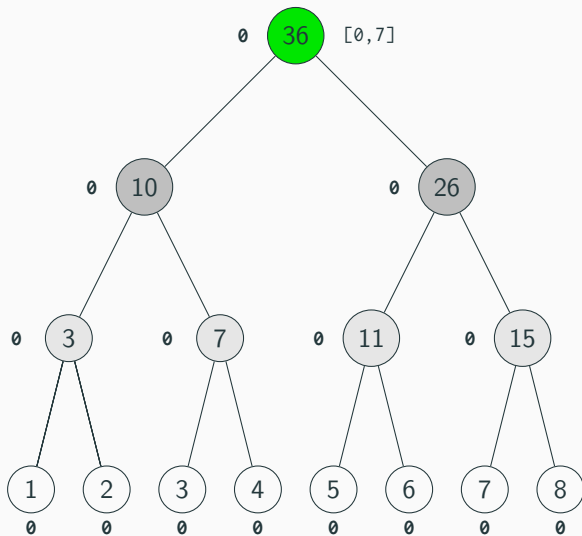
---

## Range update

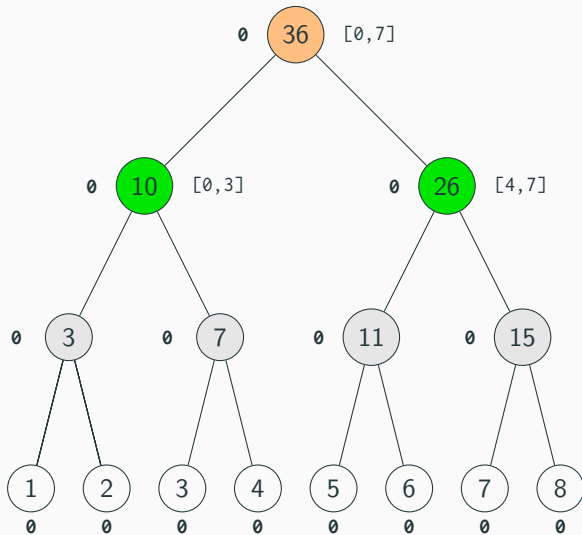
- Uma variante comum dentre as operação de uma árvore de segmentos é a atualização de intervalo (*range\_update*)
- A chamada `range_update(a, b, value)` aplica o parâmetro `value` a todos os elementos de *xs* cujos índices pertencem ao intervalo  $[a, b]$
- Implementada diretamente, de forma semelhante 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*

- 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)

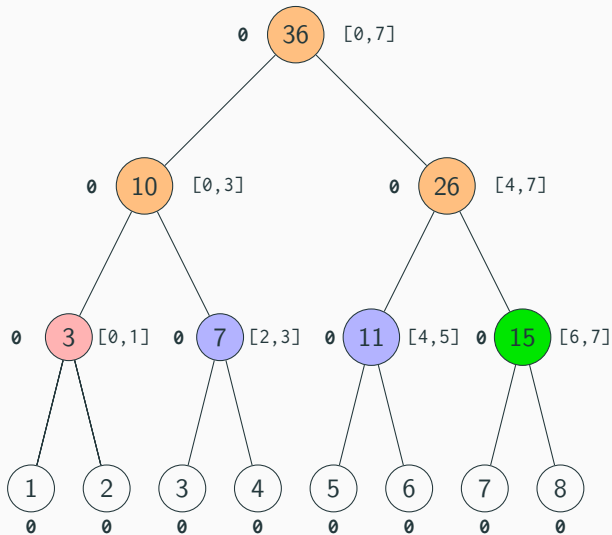


## Visualização de update(2, 6, -3)

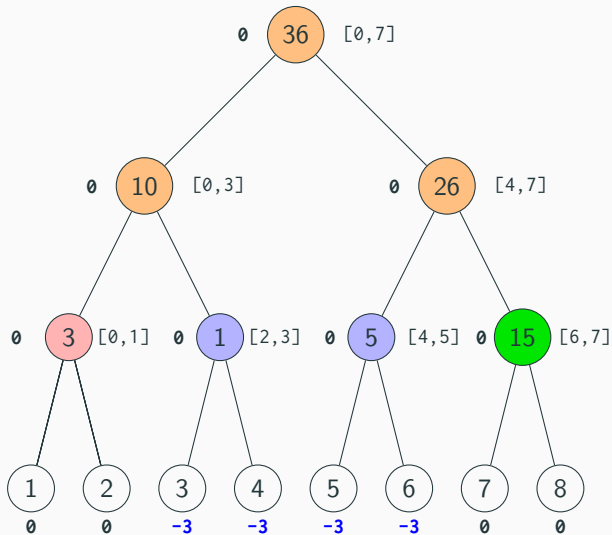




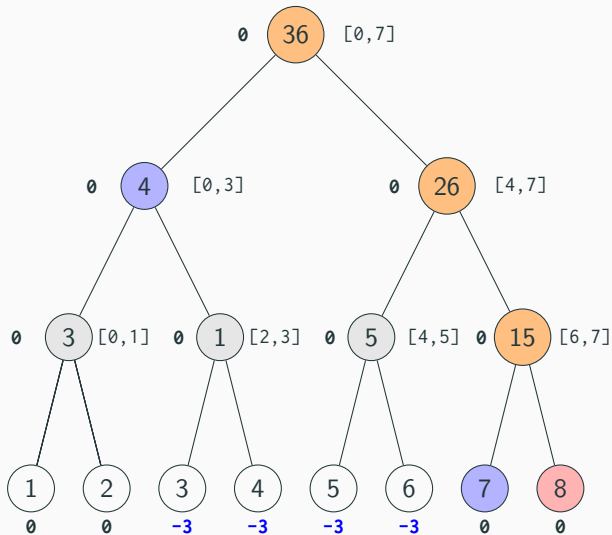
## Visualização de update(2, 6, -3)



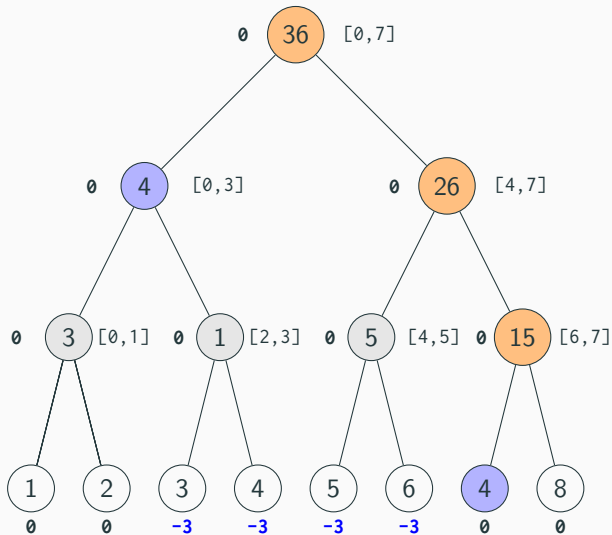
## Visualização de update(2, 6, -3)



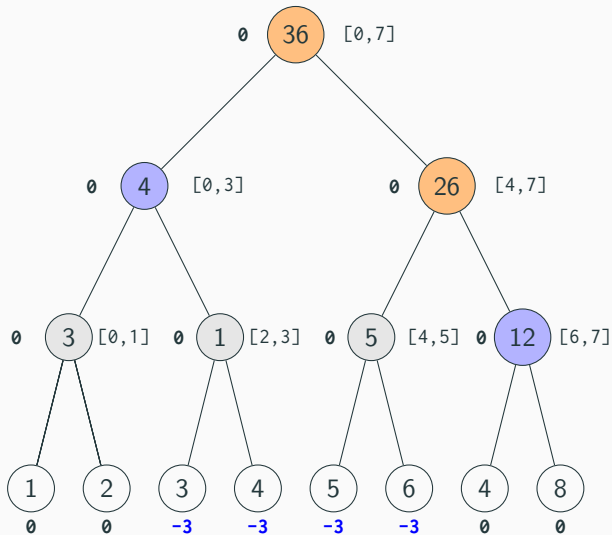
## Visualização de update(2, 6, -3)



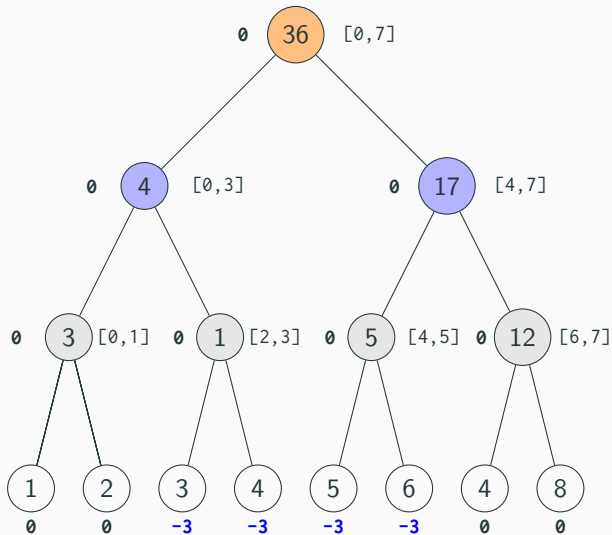
## Visualização de update(2, 6, -3)



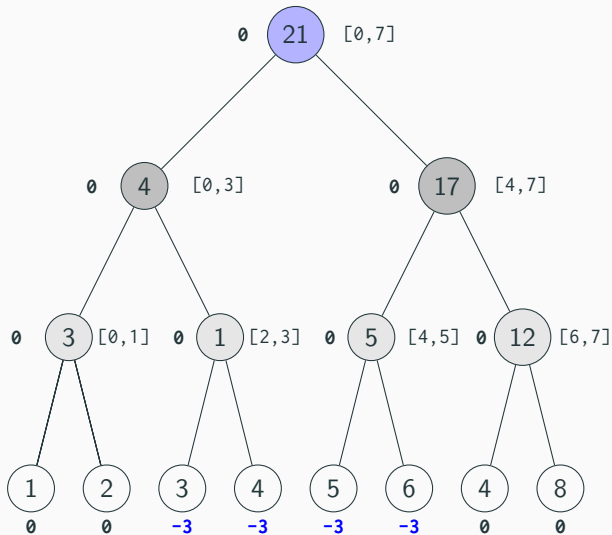
## Visualização de update(2, 6, -3)



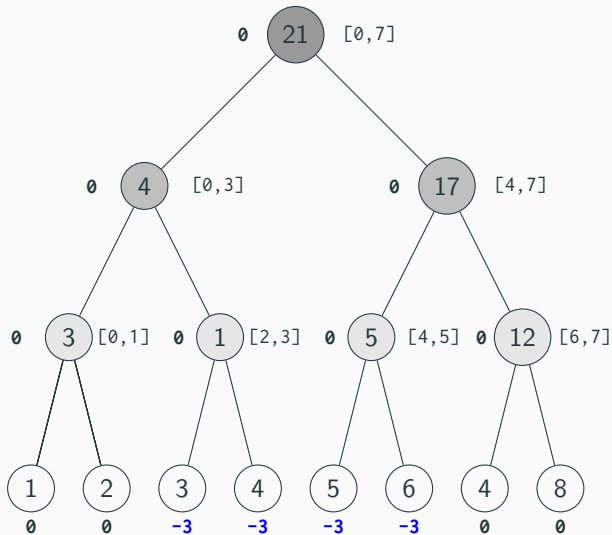
## Visualização de update(2, 6, -3)



## Visualização de update(2, 6, -3)

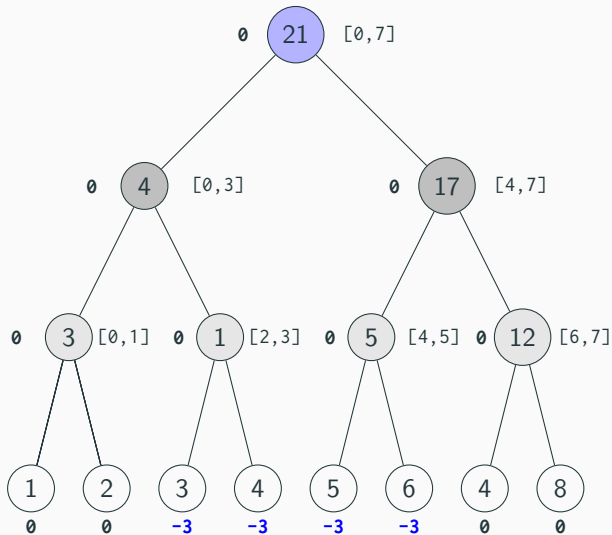


## Visualização de update(0, 7, -1)

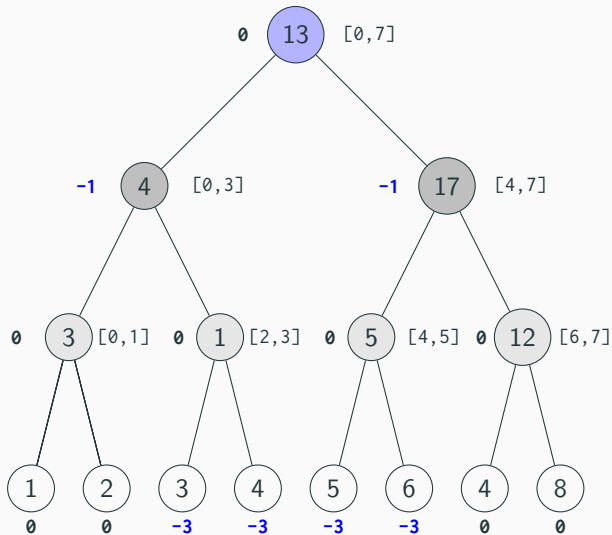




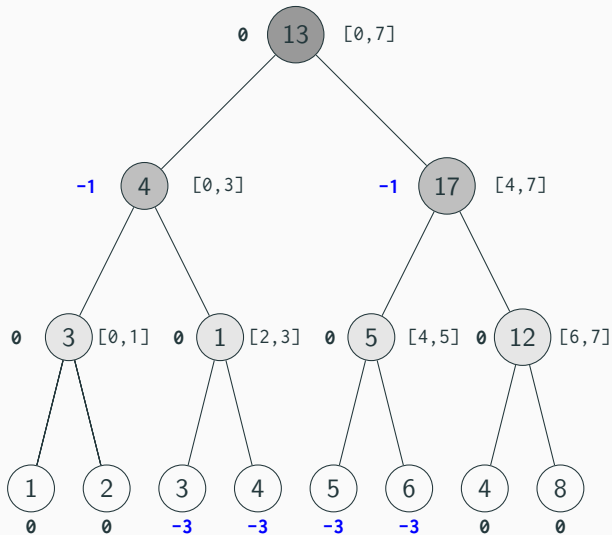
## Visualização de update(0, 7, -1)



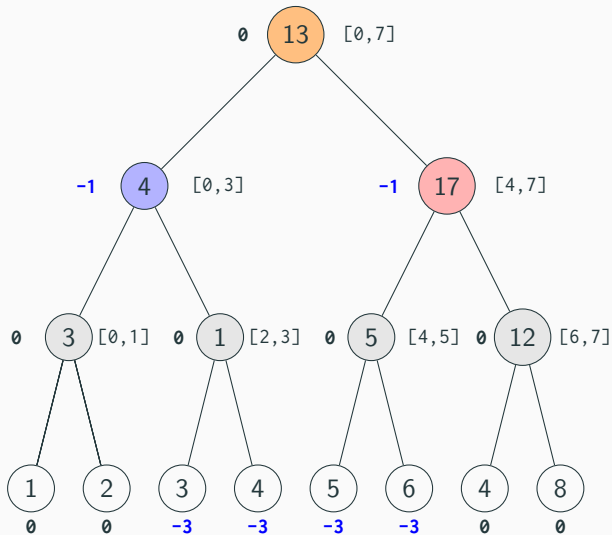
## Visualização de update(0, 7, -1)



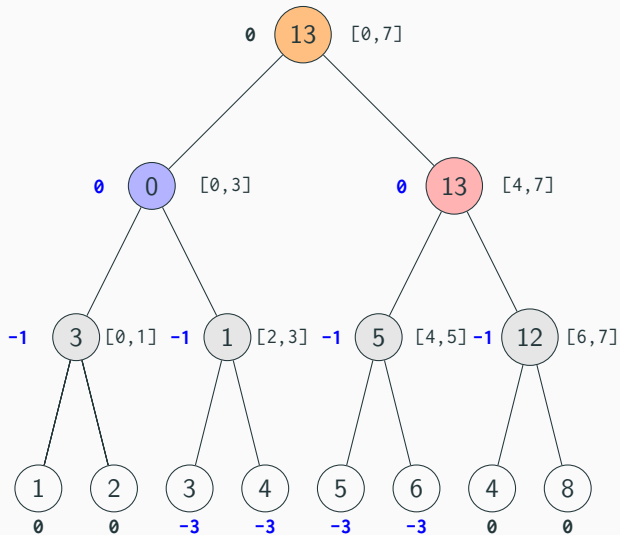
## Visualização de update(0, 3, 2)



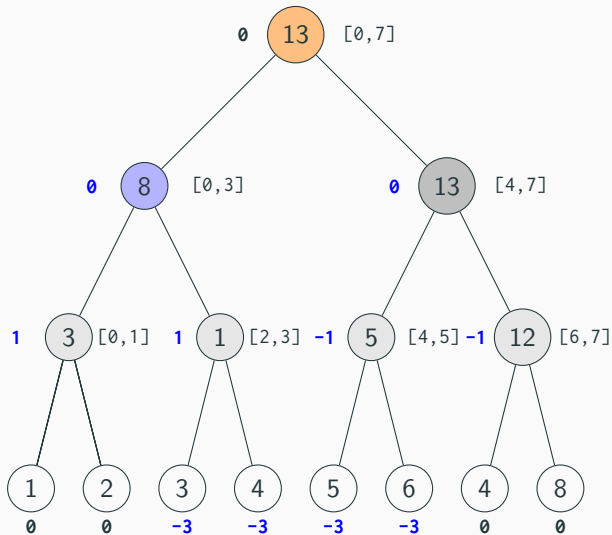
## Visualização de update(0, 3, 2)



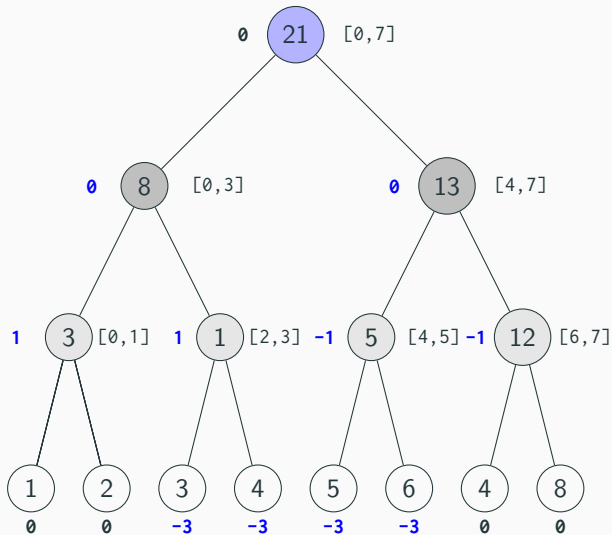
## Visualização de update(0, 3, 2)



## Visualização de update(0, 3, 2)



## Visualização de update(0, 3, 2)



# Implementação da *lazy propagation*

```
1 #ifndef SEGMENT_TREE_H
2 #define SEGMENT_TREE_H
3
4 #include <vector>
5
6 template<typename T>
7 class SegmentTree
8 {
9 private:
10     int N;
11     std::vector<T> ns, lazy;
12
13 public:
14     SegmentTree(const std::vector<int>& xs)
15         : N(xs.size()), ns(4*N), lazy(4*N, 0)
16     {
17         for (size_t i = 0; i < xs.size(); ++i)
18             update(i, i, xs[i]);
19     }
20
```



## Implementação da *lazy propagation*

```
21 void update(int a, int b, T value)
22 {
23     update(1, 0, N - 1, a, b, value);
24 }
25
26 private:
27 void update(int node, int L, int R, int a, int b, T value)
28 {
29     // Lazy propagation
30     if (lazy[node])
31     {
32         ns[node] += (R - L + 1) * lazy[node];
33
34         if (L < R) // Se o nó não é uma folha, propaga
35         {
36             lazy[2*node] += lazy[node];
37             lazy[2*node + 1] += lazy[node];
38         }
39
40         lazy[node] = 0;
41     }
```

## Implementação da *lazy propagation*

```
43 // [a, b] ∩ [L, R] = {∅}
44 if (a > R or b < L)
45     return;
46
47 // [L, R] ⊂ [a, b] está contido; é subconjunto de
48 if (a <= L and R <= b)
49 {
50     ns[node] += (R - L + 1) * value;
51
52     if (L < R)
53     {
54         lazy[2*node] += value;
55         lazy[2*node + 1] += value;
56     }
57
58     return;
59 }
60
61 update(2*node, L, (L + R)/2, a, b, value);
62 update(2*node + 1, (L + R)/2 + 1, R, a, b, value);
63
```

## Implementação da *lazy propagation*

```
64         ns[node] = ns[2*node] + ns[2*node + 1];
65     }
66
67 public:
68     T RSQ(int a, int b)
69     {
70         return RSQ(1, 0, N - 1, a, b);
71     }
72
73 private:
74     T RSQ(int node, int L, int R, int a, int b)
75     {
76         if (lazy[node])
77         {
78             ns[node] += (R - L + 1) * lazy[node];
79
80             if (L < R)
81             {
82                 lazy[2*node] += lazy[node];
83                 lazy[2*node + 1] += lazy[node];
84             }
```

# Implementação da *lazy propagation*

```
85
86     lazy[node] = 0;
87 }
88
89 if (a > R or b < L)
90     return 0;
91
92 if (a <= L and R <= b)
93     return ns[node];
94
95 T x = RSQ(2*node, L, (L + R)/2, a, b);
96 T y = RSQ(2*node + 1, (L + R)/2 + 1, R, a, b);
97
98 return x + y;
99 }
100 };
101
102 #endif
```

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