Brazilian ICPC Summer School 2022

Segment Tree Aula 1:

- Segment Tree Básica
- Segment Tree - Generalizações e Truques
- Lazy Propagation
- Busca Binária na Segment Tree

André Amaral de Sousa

Autor

Sobre

- Ex-Maratonista (ICPC 2013)
- Ex-Olímpico (IOI 2012)
- Problem Setter na Brasileira 2019 e 2020
- Engenheiro de Computação (POLI-USP)
- Professor no colégio ETAPA
- Contato:
  - andremfq@gmail.com

Segment Tree

Referências:

- Curso da ITMO
  - Precisa se inscrever para ter acesso, mas é gratuito (clicar em enroll)
  - Muitos exercícios, excelentes para treinar o básico de cada técnica
  - Também recomendo os outros cursos deles: Disjoints Sets Union, Two Pointers Method, Binary Search, Suffix Array
- Tutorial do CP-Algorithms
  - Cobre muitas técnicas, inclusive Segment Tree 2D (não falarei)
- Tutorial Persistent segment trees (Anudeep's blog)
  - Um dos primeiros que vi falar sobre seg persistente
- A simple introduction to "Segment tree beats"
  - Não falarei dessa técnica, mas fica de referência para quem quiser se aprofundar
- Non-recursive Implementation of Range Queries and Modifications over Array
  - Usaremos a versão recursiva nas aulas, mas fica como referência a versão não recursiva
- Ver o mesmo assunto, de pontos de vista diferentes, ajuda a aprender em profundidade (por exemplo ver várias implementações diferentes)

Segment Tree Básica

Problema base - Range Minimum Query (RMQ):
Dado um vetor V, com N valores inteiros, processe Q operações que podem ser de dois tipos:

- Update i val: altere o valor no índice i para val
- Query ini fim: imprima o menor valor que esteja no intervalo de índices de ini à fim, ou seja, o menor valor entre V[ini], V[ini+1], ..., V[fim].

Solução inicial: Basta manter o vetor V atualizado.

- Update i val: simplesmente fazemos V[i] = val; Complexidade O(1).
- Query ini fim: podemos percorrer o intervalo de ini à fim no vetor V e calcular o menor elemento; Complexidade O(N).
- Complexidade Final: O(Q * N)
- Apesar do Update ser extremamente eficiente, a Query é lenta, vamos ver como melhorar.

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

A princípio queremos otimizar a query, seria muito bom manter a resposta de todos os intervalos, assim ela também seria O(1), mas são MUITOS intervalos, $O(N^2)$, fica inviável manter a resposta de todos.

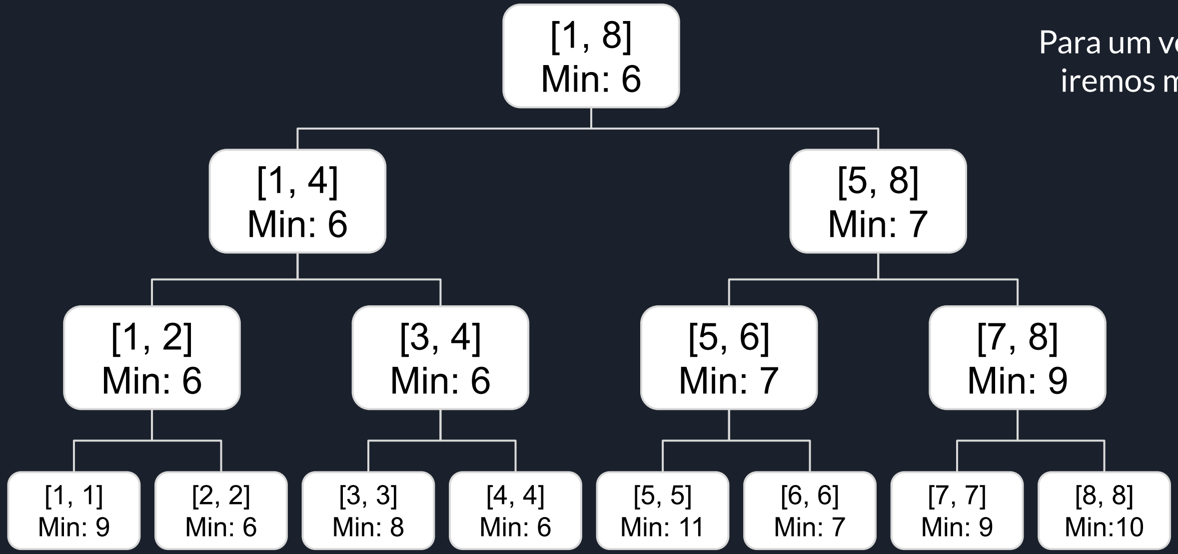
A grande sacada é que podemos manter a resposta de poucos intervalos, e qualquer intervalo será a união de alguns desses intervalos.

Vejamos um exemplo, e depois veremos os detalhes:

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Para um vetor com 8 elementos iremos manter 15 intervalos

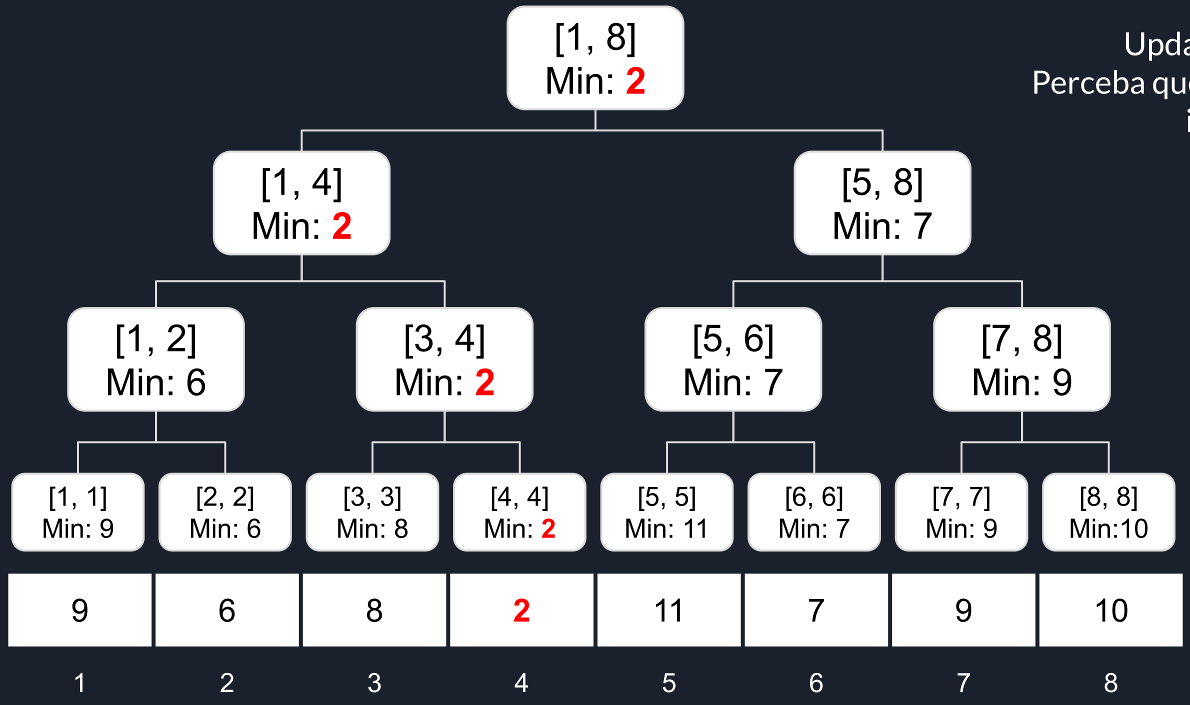


9	6	8	6	11	7	9	10
1	2	3	4	5	6	7	8

Segment Tree Básica

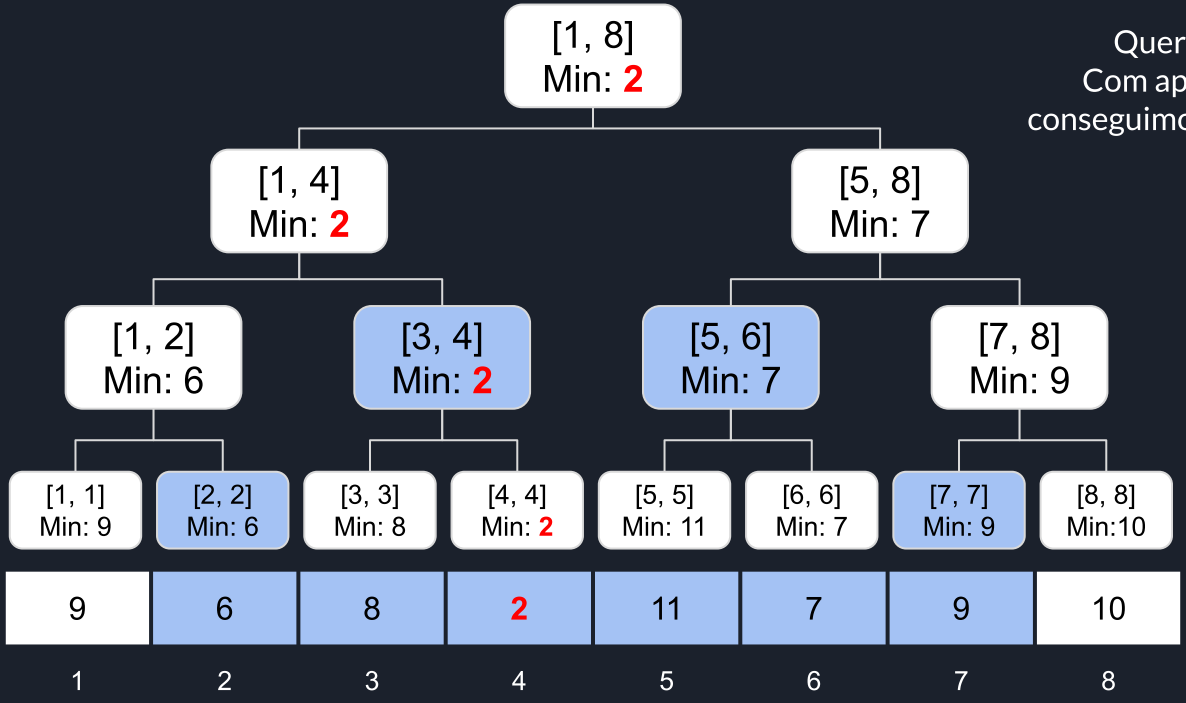
Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Update i = 4 val = 2
Perceba que alteramos apenas 4 intervalos



Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree



Query ini = 2 fim = 7
Com apenas 4 intervalos
conseguimos compor o intervalo
[2, 7]

Segment Tree Básica

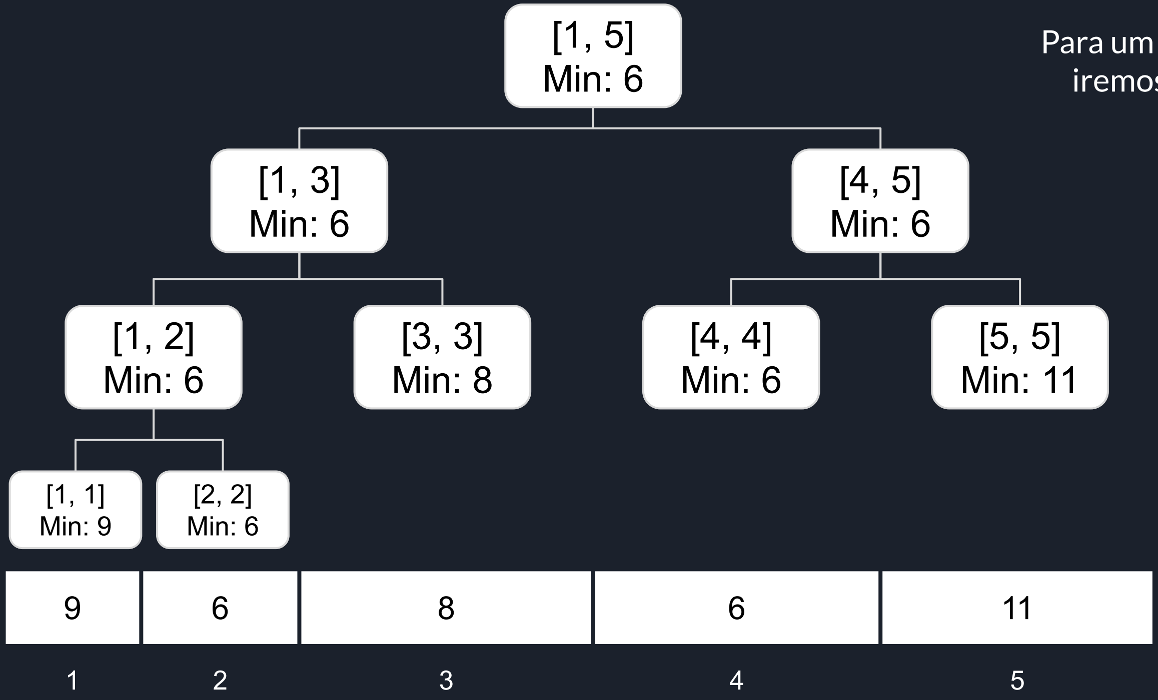
Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Vejamos os detalhes:

- Uma árvore binária **completa** com **N** folhas possui **2N - 1** nós e altura **logN + 1**
  - No exemplo anterior temos uma árvore binária completa, ela possui 8 folhas, 15 nós e altura 4
- Porém nem sempre lidaremos com árvore binária **completa** (apenas quando N for potência de 2)
  - Suponha que N não seja potência de 2, e 2^k seja a primeira potência de 2 maior que N, portanto $2^{k-1} < N < 2^k$ assim temos $2^k < 2N < 2^{k+1}$, ou seja, entre N e 2N há uma potência de 2.
  - Como para potências de 2 um bom limite para o número de nós é o dobro, e entre N e 2N há uma potência de 2, um bom limite para o número de nós (para qualquer N) é 4N.
  - Por isso é comum ver 4*MAXN como limite do número de nós
  - Vejamos um exemplo com 5 folhas

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree



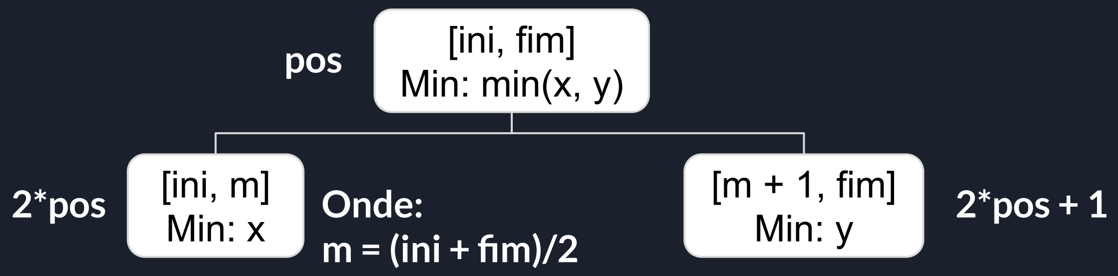
Para um vetor com 5 elementos
iremos manter 9 intervalos

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Vejamos os detalhes:

- Para armazenar esses nós, iremos usar um vetor
    - Um nó na posição **pos** terá filhos nas posições **2*pos** e **2*pos + 1**
        - CUIDADO: isso só é válido se começar o primeiro nó estiver no índice 1, para o índice 0 será 2*pos + 1 e 2*pos + 2



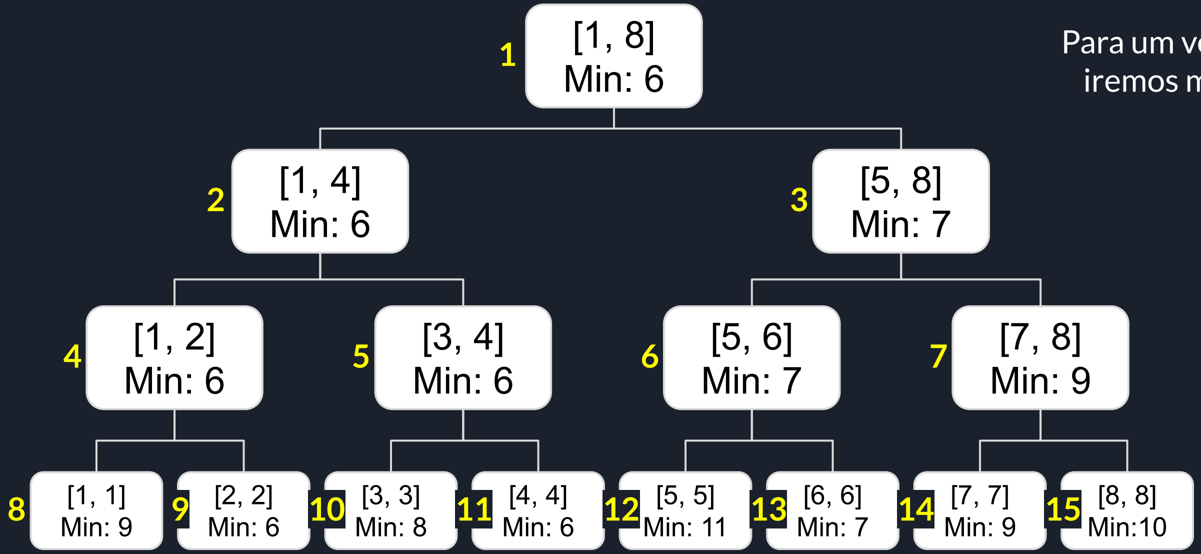
Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Para um vetor com 8 elementos iremos manter 15 intervalos

Seg:

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



V:

9	6	8	6	11	7	9	10
1	2	3	4	5	6	7	8

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Vejamos os detalhes:

- De acordo com a forma como estruturamos a Seg, fica simples realizar as operações de forma recursiva
    - Sempre chamamos cada função passando a raiz da árvore como parâmetro, e a própria função vai andando na árvore.
- Em geral cada função recursiva terá que ter o nó que ela está agora, e o intervalo que este nó representa no vetor (int pos, int ini, int fim), além dos parâmetros específicos de cada função
    - É possível manter o intervalo de cada nó no próprio nó (por exemplo criar uma struct para o nó) porém gasta mais memória, por isso sempre passo na própria função
    - Procure encontrar uma boa convenção para você (inclusive de nomes) e use sempre ela, pois isso evitará erros

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

```
const int MAXN = 100010;
int seg[4 * MAXN];

void update(int pos, int ini, int fim, int id, int val) {
    if(id < ini || id > fim) return;
    if(ini == fim) {
        seg[pos] = val;
        return;
    }

    int m = (ini + fim)/2;
    int e = 2 * pos, d = 2 * pos + 1;
    update(e, ini, m, id, val);
    update(d, m + 1, fim, id, val);

    seg[pos] = min(seg[e], seg[d]);
}
```

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Vejamos os detalhes do update:

- Neste exemplo o vetor seg armazena o mínimo de cada nó, e como só precisamos do mínimo, é apenas um vetor
  - Veremos mais adiante que podemos implementar como struct e armazenar várias informações de cada nó, ou ainda ter vários vetores, um para cada tipo de informação desejada
- Estamos no nó de índice pos, que representa o intervalo de ini à fim (intervalo do vetor que a seg representa). E queremos alterar o valor do índice id para val
  - E também alterar todos os nós necessários
- Se o índice id estiver fora do intervalo de ini à fim, então nada dali pra baixo deve ser alterado
- Se o índice id estiver dentro do intervalo e este for uma folha (ini == fim) então temos que alterar este nó.
- Caso contrário, chamamos o update aos dois filhos (esquerdo e direito). Note que algum desses updates pode alterar o valor do mínimo em algum filho, portanto após o update dos filhos precisamos recalcular o valor do mínimo do nó pos
  - É o que chamamos de Merge

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

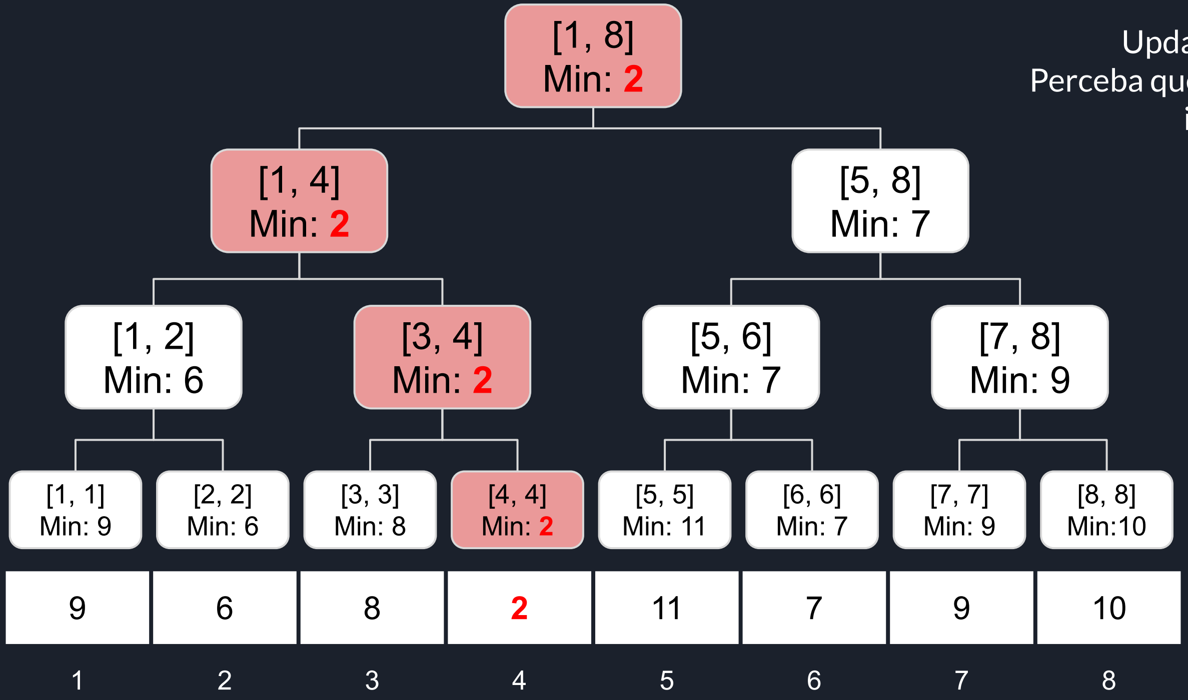
Vejamos os detalhes do update:

- Perceba que a cada nível iremos visitar apenas um nó
  - Quando chamamos update aos dois filhos, um deles cairá no primeiro caso, onde id está fora do intervalo do nó, e já dará return, não estou considerando isso como entrar

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Update i = 4 val = 2
Perceba que alteramos apenas 4 intervalos



Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Vejamos os detalhes do update:

- Perceba que a cada nível iremos visitar apenas um nó
  - Quando chamamos update aos dois filhos, um deles cairá no primeiro caso, onde id está fora do intervalo do nó, e já dará return, não estou considerando isso como entrar
- Assim a complexidade do update é a altura da árvore, que no caso é O(log N)
  - Comparado com a solução inicial, o update fica pior, mas não muito

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

```
int query(int pos, int ini, int fim, int p, int q) {
    if(q < ini || p > fim) return INT_MAX;
    if(p <= ini && fim <= q) return seg[pos];

    int m = (ini + fim)/2;
    int e = 2 * pos, d = 2 * pos + 1;
    return min(query(e, ini, m, p, q), query(d, m + 1, fim, p, q));
}
```

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Vejamos os detalhes da query:

- Estamos no nó de índice pos, que representa o intervalo de ini à fim (intervalo do vetor que a seg representa), e queremos calcular o menor valor do intervalo de p à q.
- Se o intervalo do nó não possui interseção com o intervalo desejado, então retornamos INT_MAX (maior valor que um int consegue armazenar) para não influenciar no cálculo do mínimo.
- Mas se o intervalo do nó estiver inteiramente contido no intervalo desejado, este é um dos nós que será usado para compor aquele intervalo, portanto retornamos o menor valor dele, ou seja, seg[pos]
    - CUIDADO: queremos verificar se [ini, fim] está completamente dentro de [p, q] muitos se confundem, minha dica é deixar visual: p <= ini && fim <= q (veja como [ini, fim] fica dentro de [p, q])
- Caso contrário devemos chamar a query para os filhos e retornar o mínimo entre eles.

Segment Tree Básica

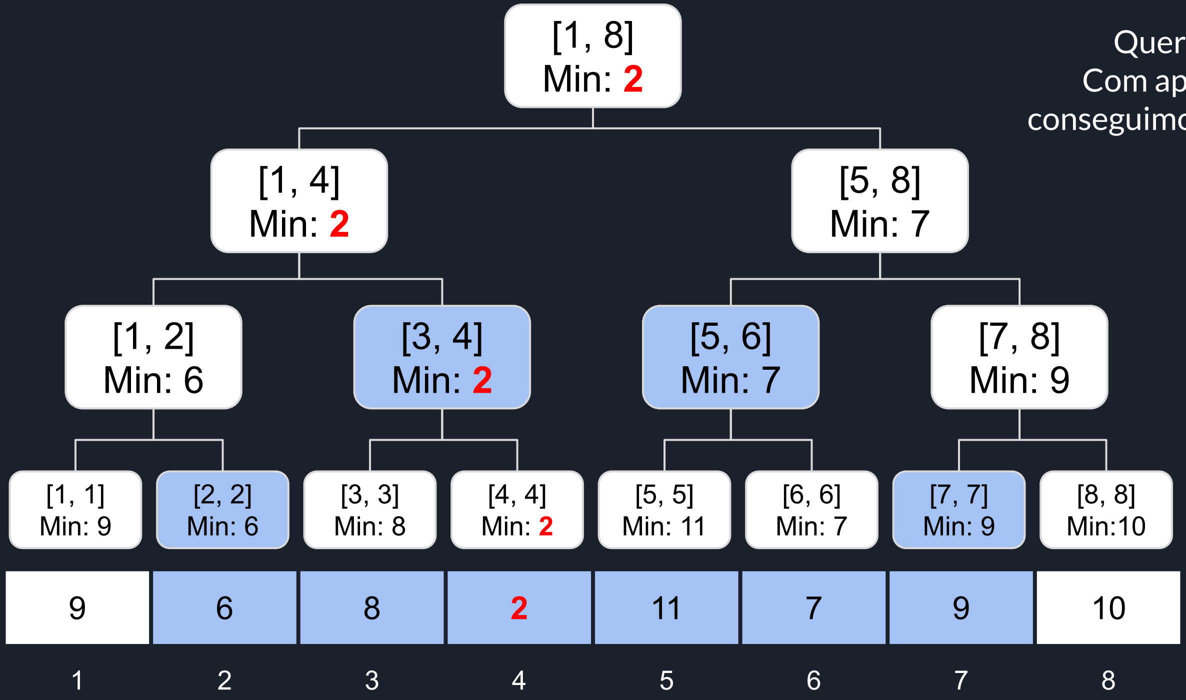
Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Vejamos os detalhes da query:

- Perceba que a cada nível serão utilizados no máximo 2 nós

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree



Query ini = 2 fim = 7
Com apenas 4 intervalos conseguimos compor o intervalo [2, 7]

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Vejamos os detalhes da query:

- Perceba que a cada nível serão utilizados no máximo 2 nós
- Portanto a complexidade da query fica O(logN)
  - Note como melhoramos bastante a complexidade da query, mesmo piorando um pouco a complexidade do update, ainda vale muito a pena
- Assim a complexidade final fica O(Q * logN)

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

Um bom problema para começar é Segment Tree for the Minimum. É a aplicação direta do que vimos, mas há um detalhe, o vetor V já começa com valores preenchidos, há duas formas de tratar isso:

- Para cada valor do vetor inicial, fazemos um update na seg
  - Funciona bem neste caso pois o update do problema consegue fazer exatamente o que desejamos, mas isso nem sempre é verdade, por isso é bom saber a segunda solução
- Fazer uma função para construir a seg inicial, a função **build**
  - No início, lemos os valores iniciais, salvamos em v, e depois chamamos a **build** na raiz da seg
  - Vejamos:

Segment Tree Básica

Problema base - Range Minimum Query (RMQ): solução usando Segment Tree

```
void build(int pos, int ini, int fim) {
    if(ini == fim) {
        seg[pos] = v[ini];
        return;
    }

    int m = (ini + fim)/2;
    int e = 2 * pos, d = 2 * pos + 1;
    build(e, ini, m);
    build(d, m + 1, fim);

    seg[pos] = min(seg[e], seg[d]);
}
```

Segment Tree Básica

Lista de Exercícios

- Segment Tree for the Minimum
- KGSS
- Number of Minimums on a Segment

Seg - Generalizações e Truques

Problema Addition to Segment - TENTE RESOLVER SOZINHO(A) PRIMEIRO

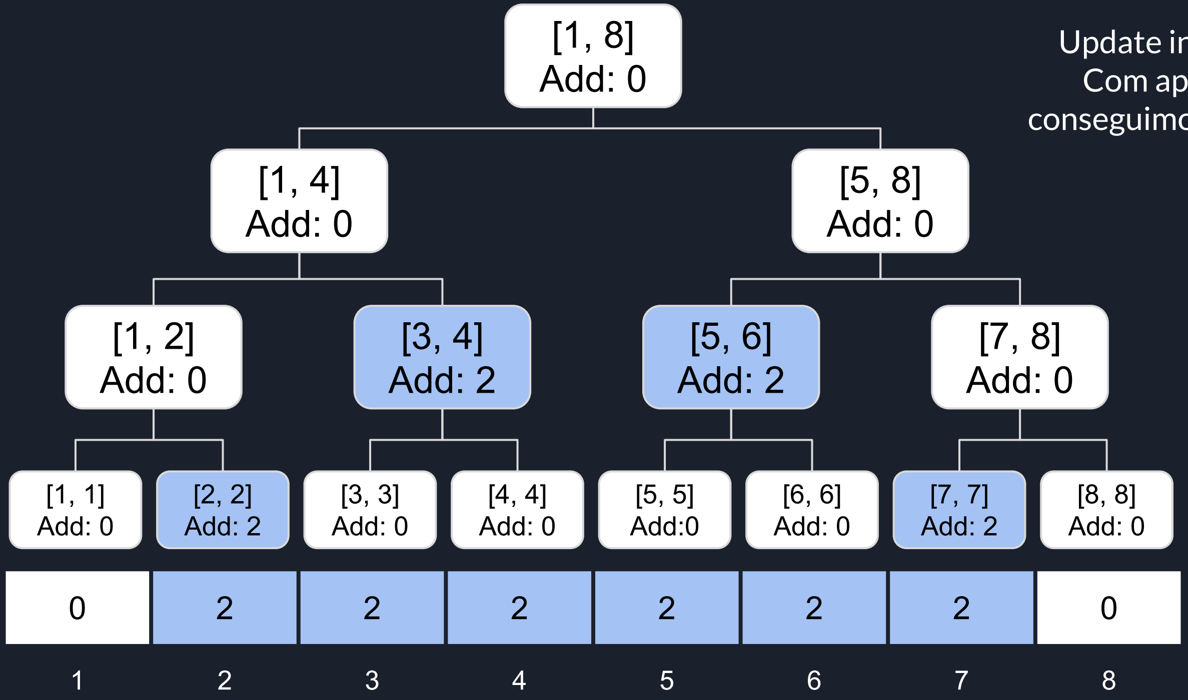
No problema anterior (RMQ) vimos uma Segment Tree que possui operação de update em um índice e query em um intervalo. Agora veremos que é possível fazer o contrário, ou seja, update em um intervalo e query em um índice.

Veremos o problema Addition to Segment. Neste problema queremos no update adicionar um valor a cada um dos valores de um intervalo, e na query determinar o valor de um único elemento.

Para isso, basta visualizar o significado de seg[pos] como: quanto deve ser somado à cada um dos valores que pertencem ao intervalo do nó pos (ou seja, de ini à fim). Vejamos um exemplo:

Seg - Generalizações e Truques

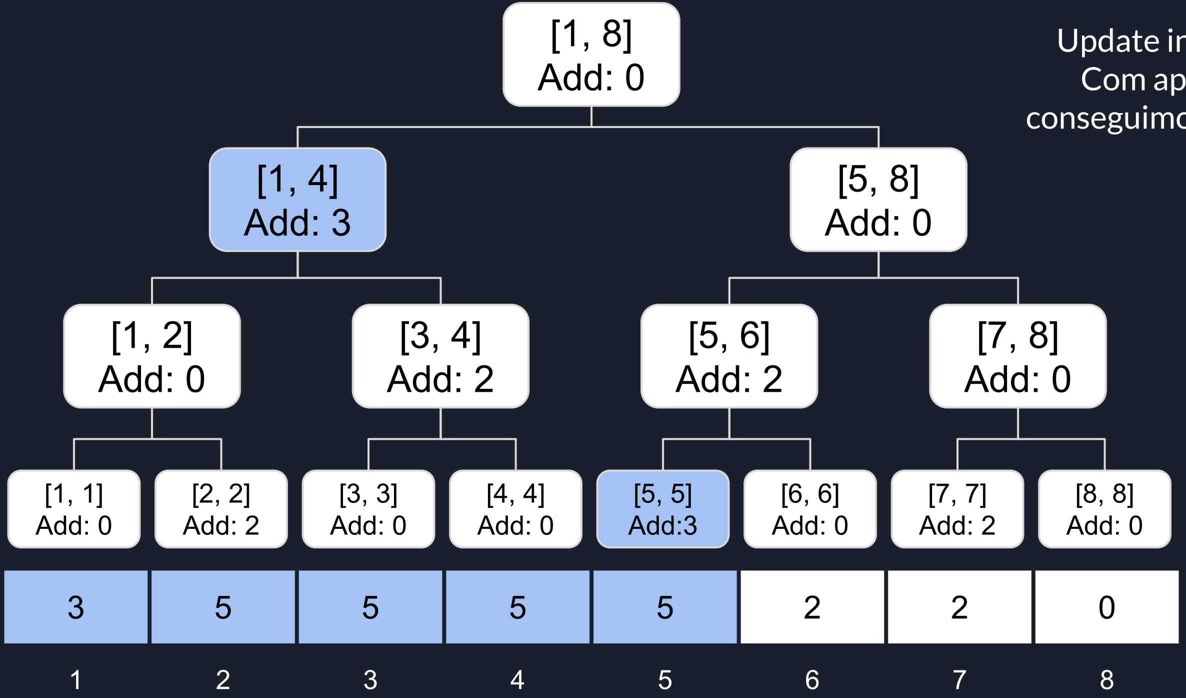
Problema Addition to Segment



Update ini = 2 fim = 7 val = 2
Com apenas 4 intervalos
conseguimos compor o intervalo
[2, 7]

Seg - Generalizações e Truques

Problema Addition to Segment



Update ini = 1 fim = 5 val = 3
Com apenas 2 intervalos conseguimos compor o intervalo [1, 5]

Seg - Generalizações e Truques

Problema <u>Addition to Segment</u>

Para isso, basta visualizar o significado de seg[pos] como: quanto deve ser somado à cada um dos valores que pertencem ao intervalo do nó pos (ou seja, de ini à fim).

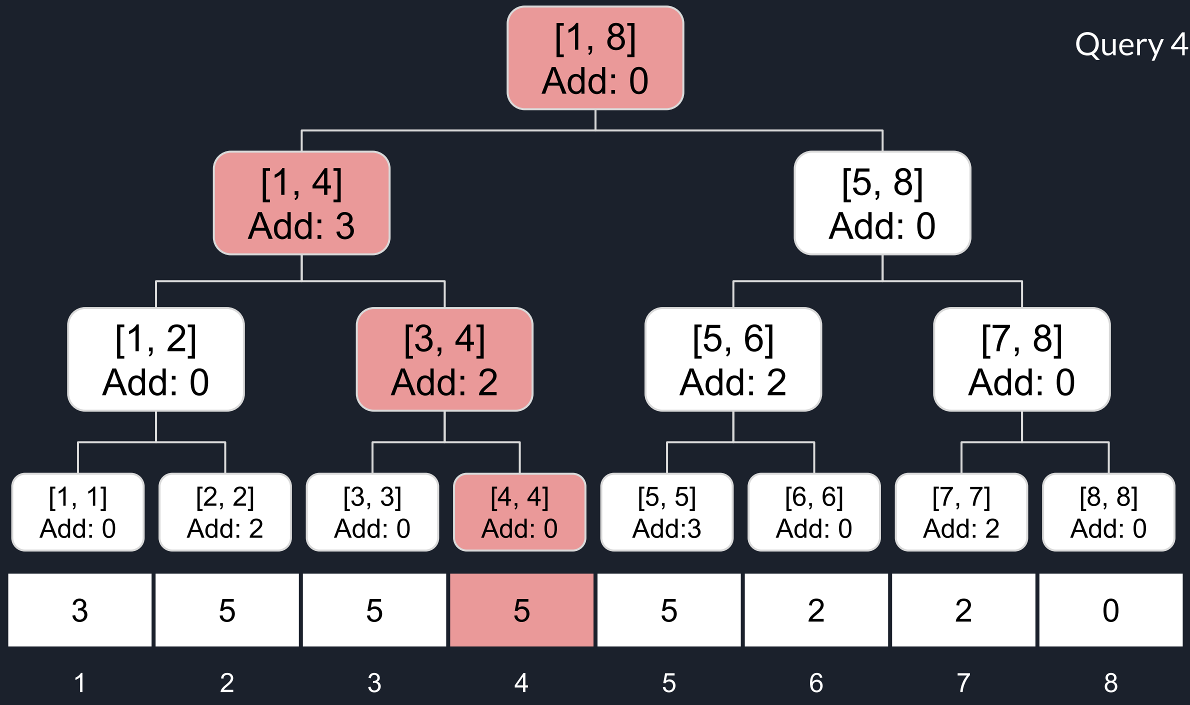
Portanto para o Update, basta somar val aos intervalos que compoem o intervalo de ini à fim

Para a Query, basta somar o valor de todos os nós que contém aquele índice, vejamos um exemplo:

Seg - Generalizações e Truques

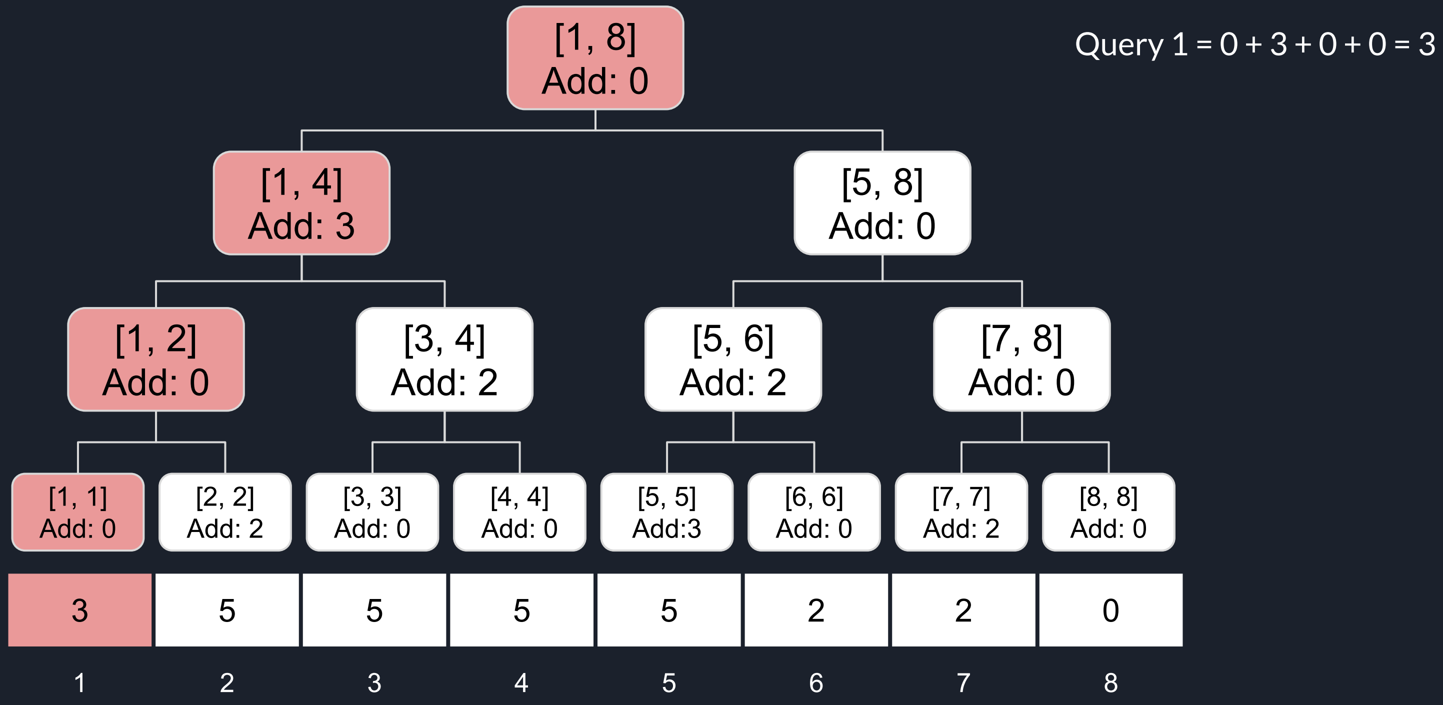
Problema Addition to Segment

Query 4 = 0 + 3 + 2 + 0 = 5



Seg - Generalizações e Truques

Problema Addition to Segment



Seg - Generalizações e Truques

Problema Addition to Segment

A implementação do Update ficará mais parecida com a Query que vimos no RMQ, e a da query mais parecida com do Update.

Mais adiante veremos uma técnica chamada Lazy Propagation, dá pra resolver este problema com usando ela, mas não é necessário. Em geral Lazy só é necessário quando tanto a Query quanto o Update são em intervalo.

Mais um problema para treinar este truque: Applying MAX to Segment

Seg - Generalizações e Truques

Problema Sign alternation - TENTE RESOLVER SOZINHO(A) PRIMEIRO

Anteriormente comentei que é possível criar uma struct com as informações de cada nó, essa ideia será útil no problema Sign alternation

Em cada nó iremos manter duas informações:

- sumIdEven - soma dos valores nas posições pares deste nó
- sumIdOdd - soma dos valores nas posições ímpares deste nó

Além de ter esta implementação ligeiramente diferente (usando struct para os nós) iremos fazer a Query retornar um nó, ou seja, a Query vai retornar um nó que seja o merge dos nós que compõem aquele intervalo. Desta forma responder a query final ficará bem simples, vejamos o código:

Seg - Generalizações e Truques

Problema Sign alternation

```
const int MAXN = 100010;
int v[MAXN];

struct node {
    int sumIdEven, sumIdOdd;
};

node seg[4 * MAXN], nulo;
```

```
node merge(node n1, node n2) {
    node resp;
    resp.sumIdEven = n1.sumIdEven + n2.sumIdEven;
    resp.sumIdOdd = n1.sumIdOdd + n2.sumIdOdd;
    return resp;
}

void build(int pos, int ini, int fim) {
    if(ini == fim) {
        seg[pos].sumIdEven = seg[pos].sumIdOdd = 0;
        if((ini % 2) == 0) seg[pos].sumIdEven = v[ini];
        else    seg[pos].sumIdOdd = v[ini];
        return;
    }

    int m = (ini + fim) / 2;
    int e = 2 * pos, d = e + 1;
    build(e, ini, m);
    build(d, m + 1, fim);
    seg[pos] = merge(seg[e], seg[d]);
}
```

Seg - Generalizações e Truques

Problema Sign alternation

```
void update(int pos, int ini, int fim, int id, int val) {
    if(id < ini || id > fim) return;
    if(ini == fim) {
        if((id % 2) == 0) seg[pos].sumIdEven = val;
        else seg[pos].sumIdOdd = val;
        return;
    }

    int m = (ini + fim)/2;
    int e = 2 * pos, d = 2 * pos + 1;
    update(e, ini, m, id, val);
    update(d, m + 1, fim, id, val);
    seg[pos] = merge(seg[e], seg[d]);
}
```

Seg - Generalizações e Truques

Problema Sign alternation

```
node query(int pos, int ini, int fim, int p, int q) {
    if(q < ini || p > fim) return nulo;
    if(p <= ini && fim <= q) return seg[pos];

    int m = (ini + fim)/2;
    int e = 2 * pos, d = 2 * pos + 1;
    return merge(query(e, ini, m, p, q), query(d, m + 1, fim, p, q));
}
```

Seg - Generalizações e Truques

Problema Sign alternation

```
int main() {
    nulo.sumIdEven = nulo.sumIdOdd = 0;
    int n; scanf("%d", &n);
    for(int i = 1; i <= n; i++) scanf("%d", &v[i]);
    build(1, 1, n);
    int q; scanf("%d", &q);
    for(int i = 1; i <= q; i++) {
        int op, a, b; scanf("%d %d %d", &op, &a, &b);
        if(op == 0) {
            update(1, 1, n, a, b);
        }
        else {
            node resp = query(1, 1, n, a, b);
            if((a % 2) == 0) printf("%d\n", resp.sumIdEven - resp.sumIdOdd);
            else printf("%d\n", resp.sumIdOdd - resp.sumIdEven);
        }
    }
}
```

Seg - Generalizações e Truques

Seg de Kadane - TENTE RESOLVER SOZINHO(A) PRIMEIRO

Dado um vetor V, com N valores inteiros, processe Q operações que podem ser de dois tipos:

Update i val: altere o valor no índice i para val

Query ini fim: imprima a maior soma de um subintervalo de ini à fim, ou seja, o maior valor de V[i]+V[i+1]+...+V[j] para algum intervalo [i, j] com ini ≤ i ≤ j ≤ fim

*Chamamos informalmente de Seg de Kadane pois o algoritmo de Kadane determina a maior soma de um subintervalo de uma lista.

Há alguns exercícios que usam essa ideia: Segment with the Maximum Sum, GSS1, GSS3, BANFARAO, GSS5

Seg - Generalizações e Truques

Seg de Kadane

Normalmente começamos a pensar como determinar a resposta da Query. Portanto devemos manter para cada nó qual é a maior soma de um intervalo daquele nó.

Agora precisamos pensar em duas coisas:

- Como de fato calcular a resposta de uma Query, ou seja, para um intervalo de ini à fim
- Como calcular/manter a maior soma de um intervalo para cada nó

Seg - Generalizações e Truques

Seg de Kadane

Normalmente começamos a pensar como determinar a resposta da Query. Portanto devemos manter para cada nó qual é a maior soma de um intervalo daquele nó.

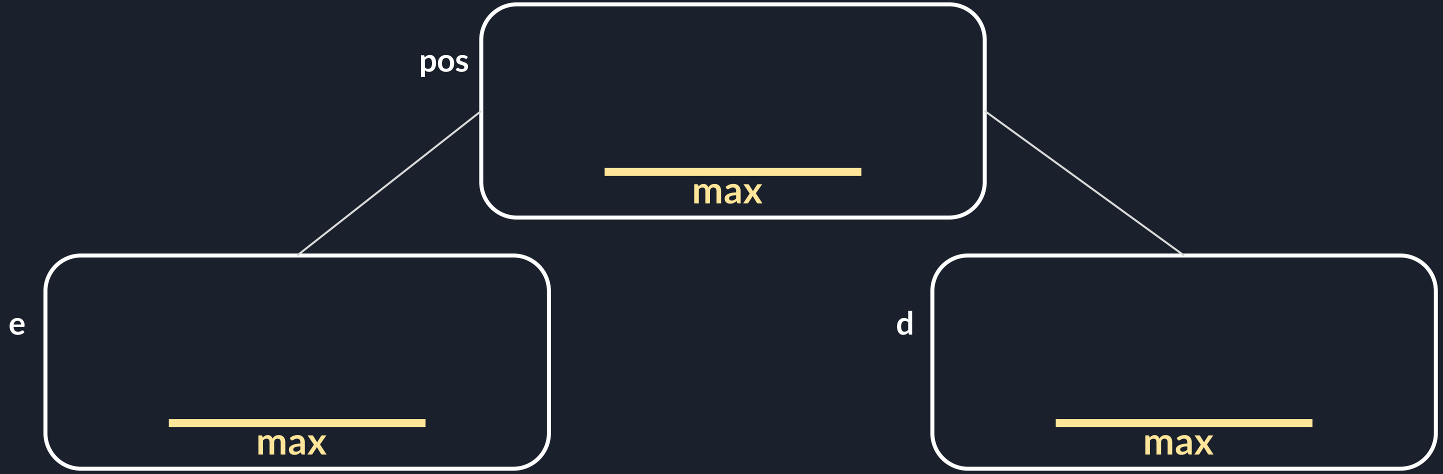
Agora precisamos pensar em duas coisas:

- Como de fato calcular a resposta de uma Query, ou seja, para um intervalo de ini à fim
  - Este é mais fácil de resolver
  - Basta usar o truque do problema anterior, e fazer a Query retornar um nó, assim teremos um nó que representa exatamente o intervalo desejado, e a resposta já estará calculada
  - Em outras palavras, só precisamos nos preocupar com o **Merge** entre dois nós
- Como calcular/manter a maior soma de um intervalo para cada nó
  - Como o Update é em um único índice, também só precisamos nos preocupar com o **Merge**, e como Updatar um nó que seja folha (o que costuma ser fácil)

Seg - Generalizações e Truques

Seg de Kadane

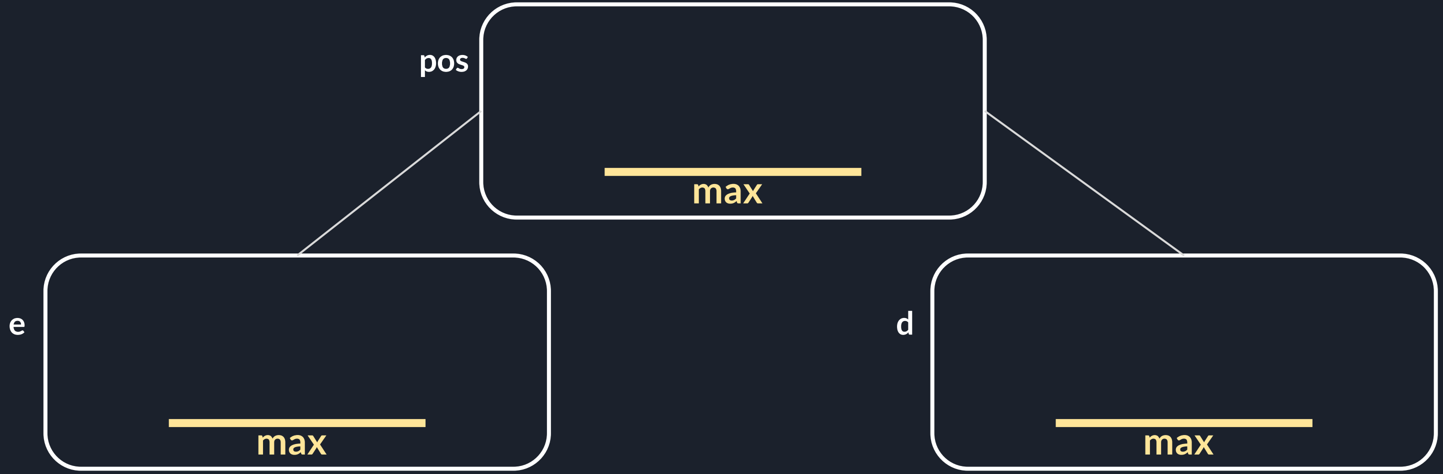
Portanto vamos pensar em como fazer o **Merge**, ou seja, se as informações dos nós filhos estiverem calculadas, como calculamos as informações de um nó ? PENSE em quais informações devemos adicionar em cada nó e como mantê-las. Lembrando que max representa a maior soma de um intervalo dentro daquele nó.



Seg - Generalizações e Truques

Seg de Kadane

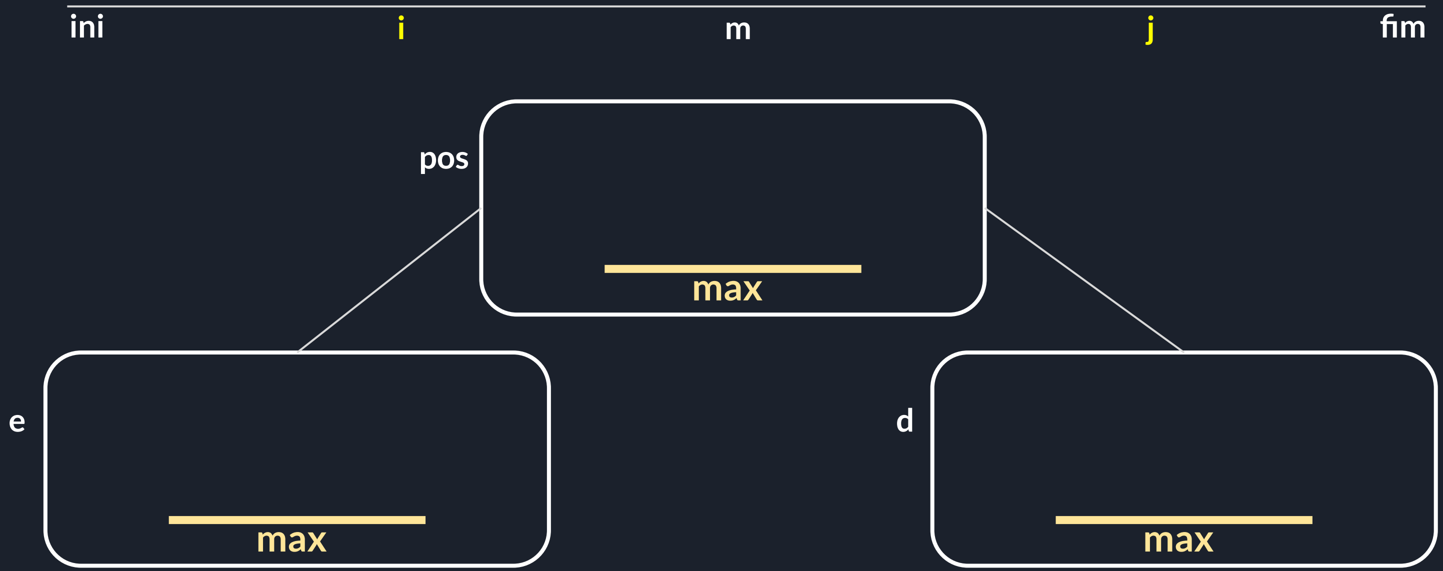
Pergunta: porque não podemos simplesmente fazer: seg[pos].max = max(seg[e].max, seg[d].max) ?



Seg - Generalizações e Truques

Seg de Kadane

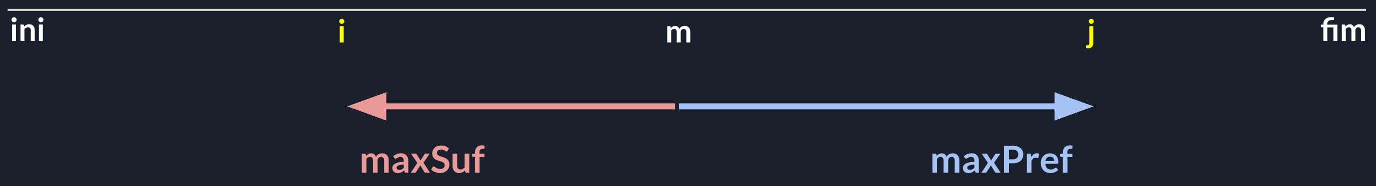
seg[pos].max = max(seg[e].max, seg[d].max). Portanto ainda precisamos considerar o melhor intervalo que começa na parte esquerda e termina na parte direita.



Seg - Generalizações e Truques

Seg de Kadane

seg[pos].max = max(seg[e].max, seg[d].max). Portanto ainda precisamos considerar o melhor intervalo que começa na parte esquerda e termina na parte direita.



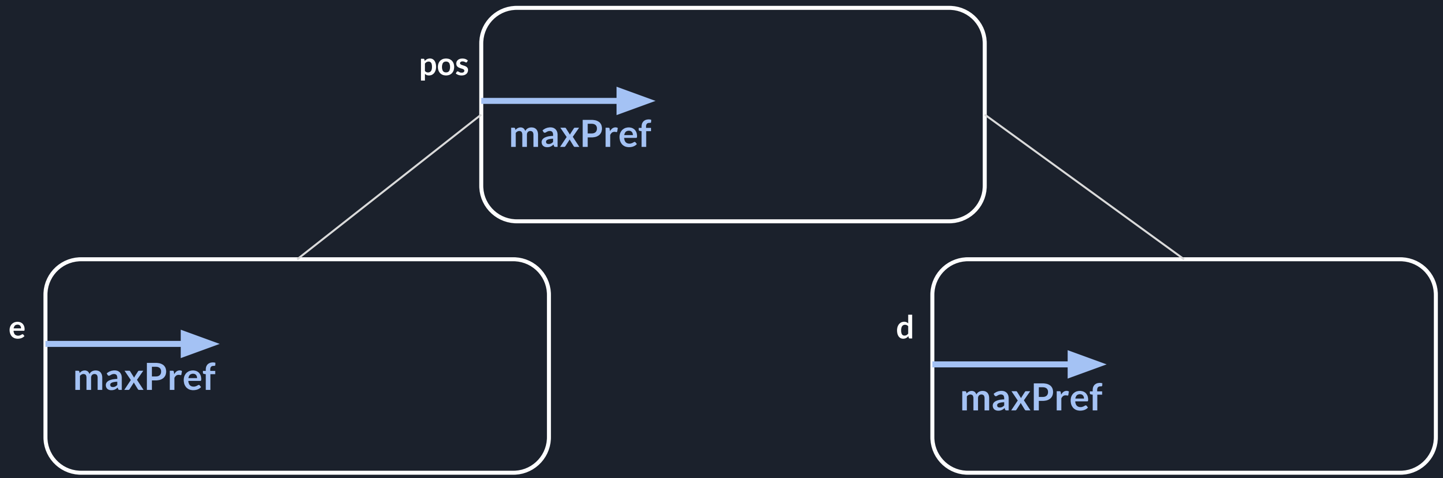
Mas perceba que o melhor intervalo que começa na parte esquerda e termina na parte direita necessariamente consiste em somar o maior **SUFIXO** da parte esquerda com o maior **PREFIXO** da parte direita

Isso significa que iremos precisar manter em cada nó o maior **SUFIXO (maxSuf)** e o maior **PREFIXO (maxPref)**. PENSE em como manter essas informações em cada nó.

Seg - Generalizações e Truques

Seg de Kadane

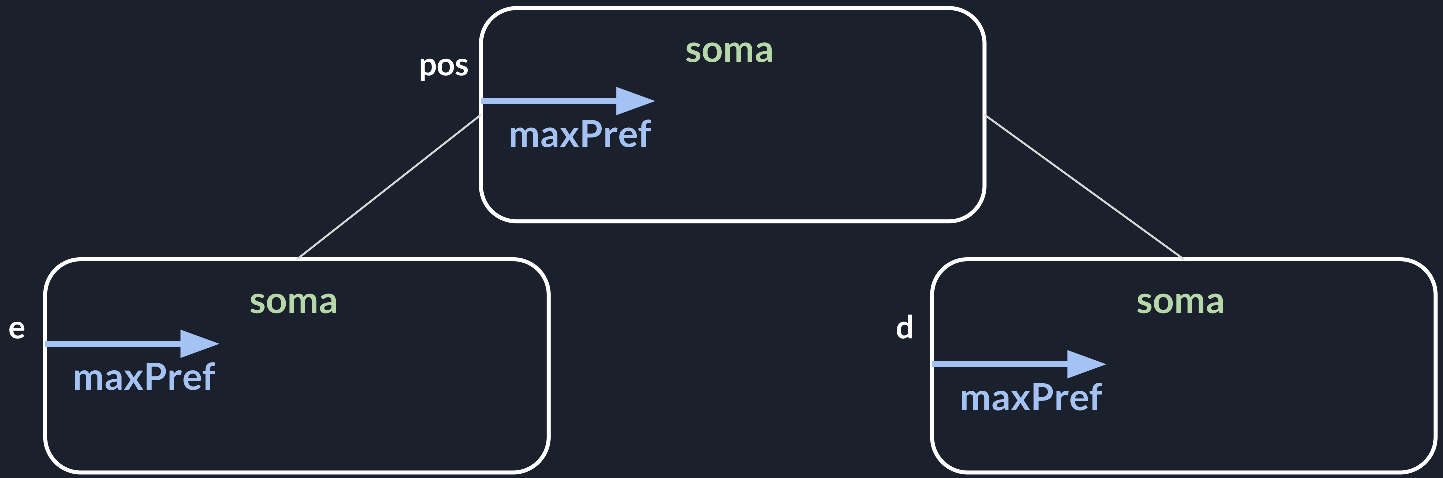
Vamos pensar em como manter **maxPref**, ou seja, dado que temos **maxPref** calculados para os filhos, PENSE em como calcular para o nó.



Seg - Generalizações e Truques

Seg de Kadane

Perceba que o melhor prefixo, ou é o melhor prefixo do filho esquerdo, ou é o filho esquerdo inteiro somado com o melhor prefixo do filho direito. Portanto teremos que manter a SOMA de cada nó

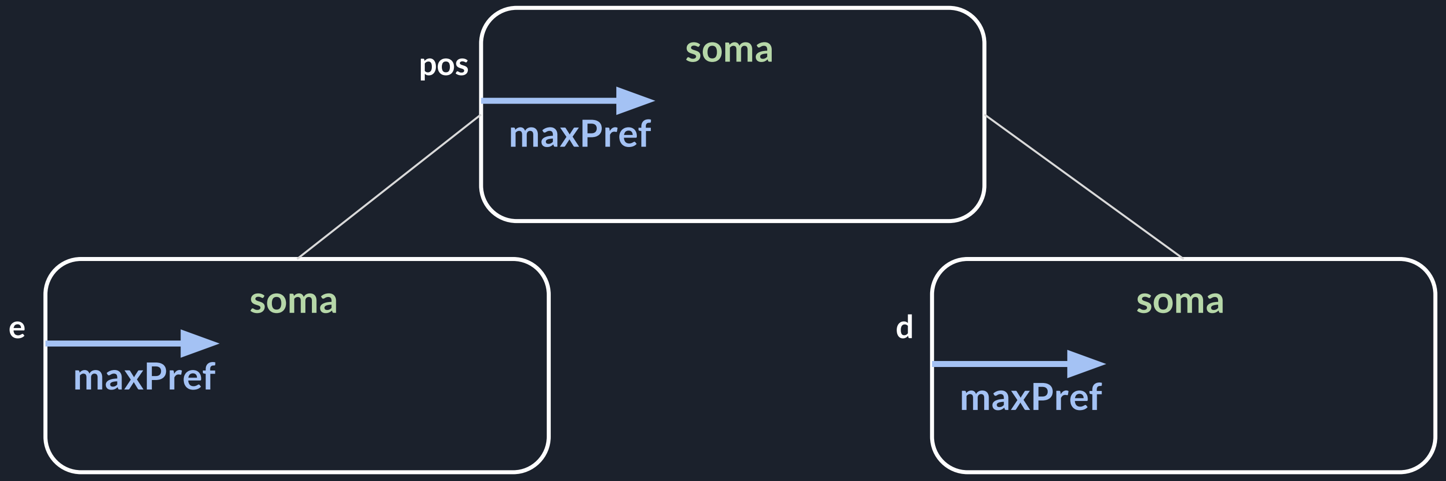


Seg - Generalizações e Truques

Seg de Kadane

Assim:

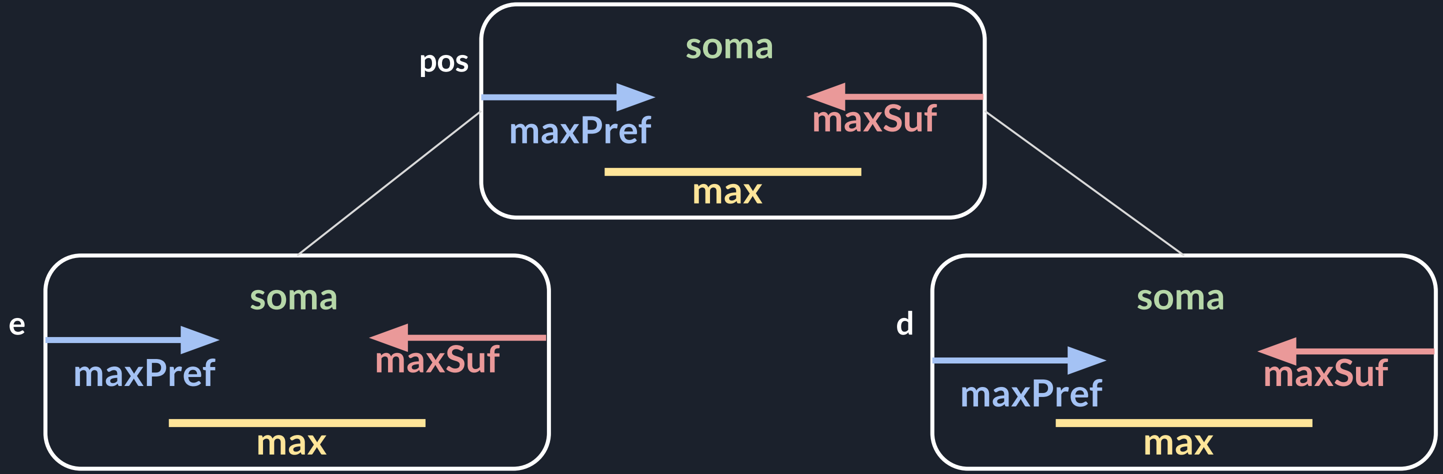
- seg[pos].soma = seg[e].soma + seg[d].soma
- seg[pos].maxPref = max(seg[e].maxPref, seg[e].soma + seg[d].maxPref)



Seg - Generalizações e Truques

Seg de Kadane

- seg[pos].soma = seg[e].soma + seg[d].soma
- seg[pos].maxPref = max(seg[e].maxPref, seg[e].soma + seg[d].maxPref)
- seg[pos].maxSuf = max(seg[d].maxSuf, seg[d].soma + seg[e].maxSuf)
- seg[pos].max = max(seg[e].maxSuf + seg[d].maxPref, max(seg[e].max, seg[d].max))



Seg - Generalizações e Truques

Seg de Kadane

Há alguns exercícios que usam essa ideia: Segment with the Maximum Sum, GSS1, GSS3, BANFARAO, GSS5

Outros exercícios onde a dificuldade está no Merge: Sereja and Brackets, BRCKTS

Seg - Generalizações e Truques

Problema <u>NICEDAY</u> - TENTE RESOLVER SOZINHO(A) PRIMEIRO

CUIDADO: queremos a quantidade de competidores que NÃO PERDEM de nenhum outro

Ideia:

Iremos processar os competidores ordenados por a, ou seja, quem obteve um ranking menor na primeira competição será processado primeiro (competidor com a = 1, depois o competidor com a = 2, etc)

Quando estivermos processando o competidor i, iremos determinar se existe algum outro competidor j que vence o competidor i, ou seja, algum competidor j tal que $aj < ai$ e $bj < bi$ e $cj < ci$

Observe que, como estamos processando os competidores por ordem de a, para determinar se existe algum competidor j que vence o competidor i, só precisamos nos preocupar com os competidores que já foram processados, pois os outros terão a maior que ai e portanto não podem vencer o competidor i

Seg - Generalizações e Truques

Problema NICEDAY

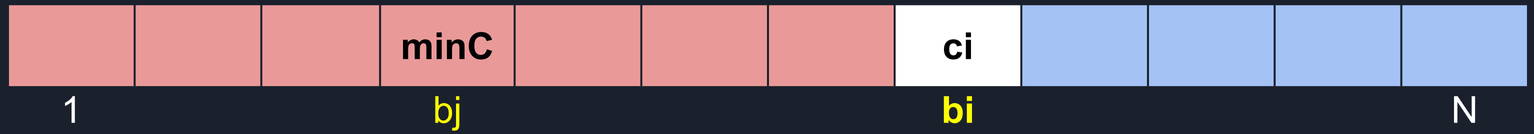
Imagine um vetor de marcação, onde para cada valor de b, iremos marcar qual o valor c de um competidor que já foi processado.

Note que ao processar o competidor i, só precisamos nos preocupar com os competidores com b de 1 à bi - 1 (região marcada em vermelho), pois os competidores com b maior (região marcada em azul) não podem vencer o competidor i.

Dentre todos os competidores que já foram processados e possuem b menor que bi, como verificar rápido se algum deles de fato vence o competidor i ? PENSE

Seg - Generalizações e Truques

Problema NICEDAY



Dentre todos os competidores que já foram processados e possuem b menor que bi, como verificar rápido se algum deles de fato vence o competidor i ?

Pense no competidor que está na região vermelha e possui o MENOR valor de c, se algum competidor vence o competidor i, então esse competidor irá vencer o competidor i.

Logo basta determinar o menor valor que está no intervalo de 1 à bi - 1, vou chamar minC, e comparar minC com ci, se ci for menor então ele não perde de ninguém e deve ser contabilizado na resposta

Não esqueça de Updatar o vetor e colocar o valor ci no índice bi

Seg - Generalizações e Truques

Problema NICEDAY

Para finalizar, basta usar uma Seg para representar este vetor de marcação, pois caimos no problema do RMQ.

Na minha opinião esse tipo de problema é muito legal, pois a dificuldade não está na estrutura em si, mas sim em como usá-la (a Seg em si é simples).

Outros exercícios nessa vibe: Nested Segments, Intersecting Segments, DQUERY

Lazy Propagation

Lazy Propagation é uma técnica que permite realizar Update em intervalo. Conforme vimos anteriormente, se a Query for num único índice, em geral não ela não é necessária, mas após dominar esta técnica é comum usar sempre que o Update for em intervalo.

Vou discutir esta técnica através da resolução do problema **HORRIBLE**

Basicamente temos que suportar dois tipos de operações:

- Update ini fim val - somar o valor val à cada um dos valores no intervalo de ini à fim
- Query ini fim - encontrar a soma dos valores no intervalo de ini à fim

O problema do Update em intervalo é que ele pode alterar MUITOS nós da Seg, portanto manter TODOS atualizados fica inviável.

Assim, a ideia principal da Lazy Propagation é marcar que um nó está desatualizado, e atualizá-lo apenas quando necessário (ou seja, ser preguiçoso, daí o nome Lazy)

Lazy Propagation

Problema <u>HORRIBLE</u>

Assim, a ideia principal da Lazy Propagation é marcar que um nó está desatualizado, e atualizá-lo apenas quando necessário (ou seja, ser preguiçoso, daí o nome Lazy)

Temos que tomar alguns cuidados:

- Ao atualizar um nó, queremos apenas atualizar ele, sem precisar de nenhuma informação de seus descendentes na árvore, e sem ter que atualizar seus descendentes
- Para atualizar um nó, muitas vezes precisamos manter algumas informações nos nós
- Muito cuidado para não usar informação não atualizada, como se estivesse atualizada
- Cuidado também em identificar quando é necessário atualizar um nó

Conforme mencionado, cada programador segue suas convenções, vou mostrar como eu uso Lazy Propagation e como faço para não ter problemas (e sigo algumas regrinhas sempre), mas esta não é a única solução.

Lazy Propagation

Problema HORRIBLE

Como no problema, a Query pede a soma, então temos que manter a soma de cada nó.

Além disso, o que é preciso para atualizar a soma quando um nó estiver desatualizado ? Quando um nó está desatualizado é porque houve um Update que o afetou, ou seja, foi somado um determinado valor em cada um de seus elementos.

Se soubermos quanto foi somado em cada um dos elementos de um nó, conseguiremos atualizar a sua soma, basta multiplicar pela quantidade de elementos.

Portanto além de guardar a soma, iremos guardar uma lazy sum (essas informações que mantemos apenas para atualizar o nó, chamamos de lazy)

```
struct node {
  long long int sum, lzSum;
};
```

Lazy Propagation

Problema HORRIBLE

Portanto podemos atualizar um nó da seguinte forma:

```
void refresh(int pos, int ini, int fim) {
  if(seg[pos].lzSum == 0) return;

  long long int num = seg[pos].lzSum;
  seg[pos].lzSum = 0;

  seg[pos].sum += (fim - ini + 1) * num;

  if(ini == fim) return;

  int e = 2*pos, d = 2*pos + 1;
  seg[e].lzSum += num;
  seg[d].lzSum += num;
}
```

Lazy Propagation

Problema HORRIBLE

Portanto podemos atualizar um nó da seguinte forma:

```
void refresh(int pos, int ini, int fim) {
  if(seg[pos].lzSum == 0) return;

  long long int num = seg[pos].lzSum;
  seg[pos].lzSum = 0;

  seg[pos].sum += (fim - ini + 1) * num;

  if(ini == fim) return;

  int e = 2*pos, d = 2*pos + 1;
  seg[e].lzSum += num;
  seg[d].lzSum += num;
}
```

Propagation: propagar a Lazy para os filhos

Lazy Propagation

Problema <u>**HORRIBLE**</u>

Porque temos que propagar a Lazy para os filhos ?

Ao marcar que um nó está desatualizado, na verdade TODOS os seus descendentes (nós abaixo dele na árvore) estarão desatualizados, MAS não podemos passar por todos eles e marcar que estão desatualizados.

Por isso uma forma eficiente é, marcar apenas um nó (na verdade apenas os que compõem o intervalo) e TODA VEZ que atualizarmos um nó, passamos a Lazy deles para os filhos dele, assim em algum momento (se necessário) a Lazy vai chegar à todos os descendentes.

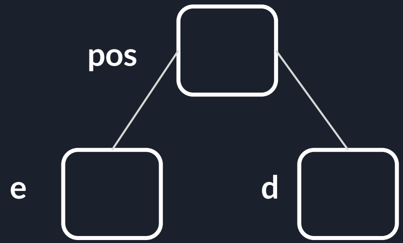
Por isso chamamos de Lazy Propagation

Lazy Propagation

Problema <u>HORRIBLE</u>

Já sabemos como atualizar um nó, mas há dois cuidados especiais (na forma como estruturo o código)

CUIDADO 1:



Suponha que o nó d está inteiramente contido num Update, e o nó e esteja completamente fora. Não podemos simplesmente somar val na Lazy de d e pronto. Pois se fizermos isso, quando voltarmos ao nó pos e fizermos o Merge (recalcular a soma de pos, somando a soma de e e a soma de d) estaremos usando um valor errado da soma de d (desatualizado).

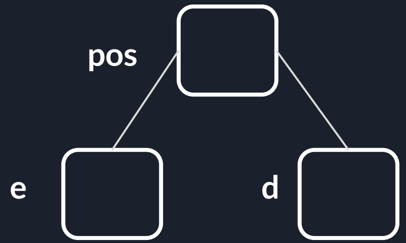
Portanto, no Update, assim que marcarmos que um nó está desatualizado, já devemos atualizá-lo. Para que seu pai use valores atualizados

Lazy Propagation

Problema <u>HORRIBLE</u>

Já sabemos como atualizar um nó, mas há dois cuidados especiais (na forma como estruturo o código)

CUIDADO 2:



Suponha que o nó d está inteiramente contido num Update, e o nó e esteja completamente fora. Mas imagine que o nó e esteja desatualizado (consequência de outro Update). Não podemos simplesmente dar return (não fazer nada) no nó e. Pois no Merge do pos estaremos usando um valor errado da soma de e (desatualizado).

Portanto, mesmo que o nó esteja completamente FORA do intervalo do Update, ainda temos que atualizá-lo. De modo geral, sempre que entrar em um nó, atualize-o

Lazy Propagation

Problema HORRIBLE

```
void update(int pos, int ini, int fim, int p, int q, int val) {
  refresh(pos, ini, fim);

  if(q < ini || p > fim) return;

  if(p <= ini && fim <= q) {
    seg[pos].lzSum += val;
    refresh(pos, ini, fim);
    return;
  }

  int m = (ini + fim)/2;
  int e = 2*pos, d = 2*pos + 1;
  update(e, ini, m, p, q, val);
  update(d, m+1, fim, p, q, val);

  seg[pos].sum=seg[e].sum+seg[d].sum;
}
```

CUIDADO 2

CUIDADO 1

Lazy Propagation

Problema [HORRIBLE]

Essencialmente esses cuidados vem do fato que, no Merge do Update os dois nós filhos devem estar atualizados

```
void update(int pos, int ini, int fim, int p, int q, int val) {
  //refresh(pos, ini, fim);

  if(q < ini || p > fim) return;

  if(p <= ini && fim <= q) {
    seg[pos].lzSum += val;
    //refresh(pos, ini, fim);
    return;
  }

  int m = (ini + fim)/2;
  int e = 2*pos, d = 2*pos + 1;
  update(e, ini, m, p, q, val);
  update(d, m+1, fim, p, q, val);

  refresh(e, ini, m);
  refresh(d, m + 1, fim);

  seg[pos].sum=seg[e].sum+seg[d].sum;
}
```

CUIDADOS 1 e 2

Lazy Propagation

Problema <u>HORRIBLE</u>

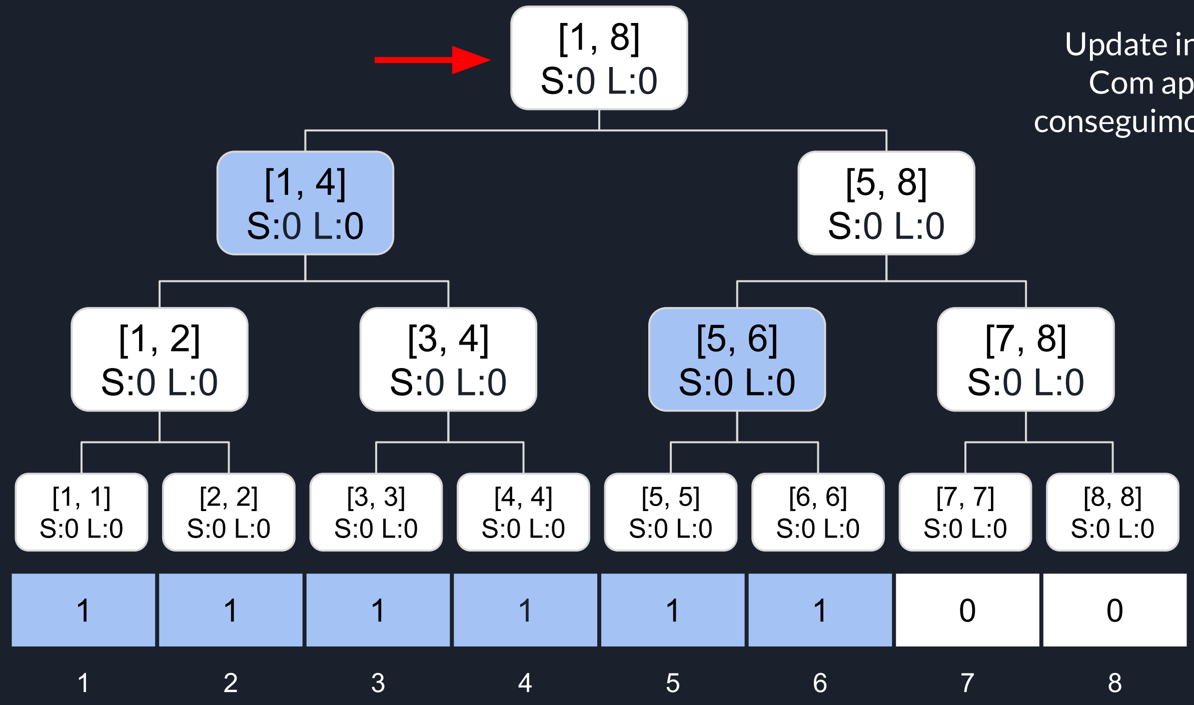
```
long long int query(int pos, int ini, int fim, int p, int q){
  refresh(pos, ini, fim);

  if(q < ini || p > fim) return 0;
  if(p <= ini && fim <= q) return seg[pos].sum;

  int m = (ini + fim)/2;
  int e = 2*pos, d = 2*pos + 1;
  return query(e, ini, m, p, q) + query(d, m+1, fim, p, q);
}
```

De modo geral, ao usar Lazy, eu sempre chamo o refresh na primeira linha de qualquer função, para garantir que vou usar valores atualizados

Lazy Propagation

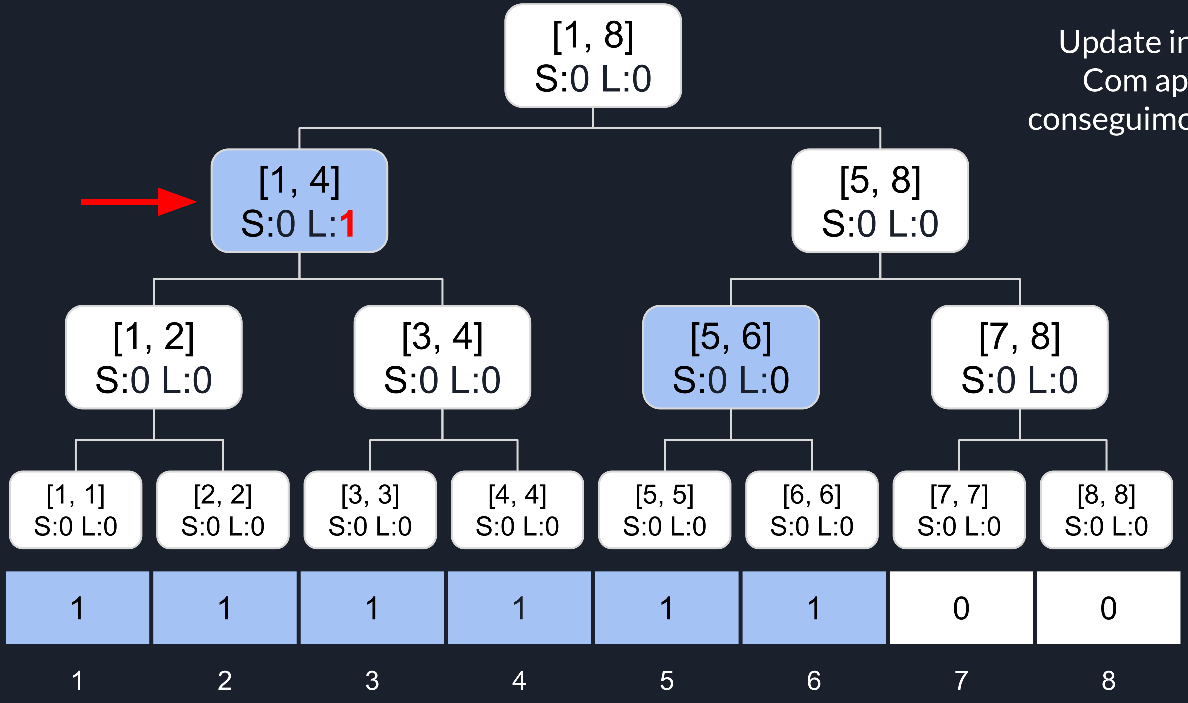
Problema <u>HORRIBLE</u> - exemplo

Update ini = 1 fim = 6 val = 1
Com apenas 2 intervalos conseguimos compor o intervalo [1, 6]



Lazy Propagation

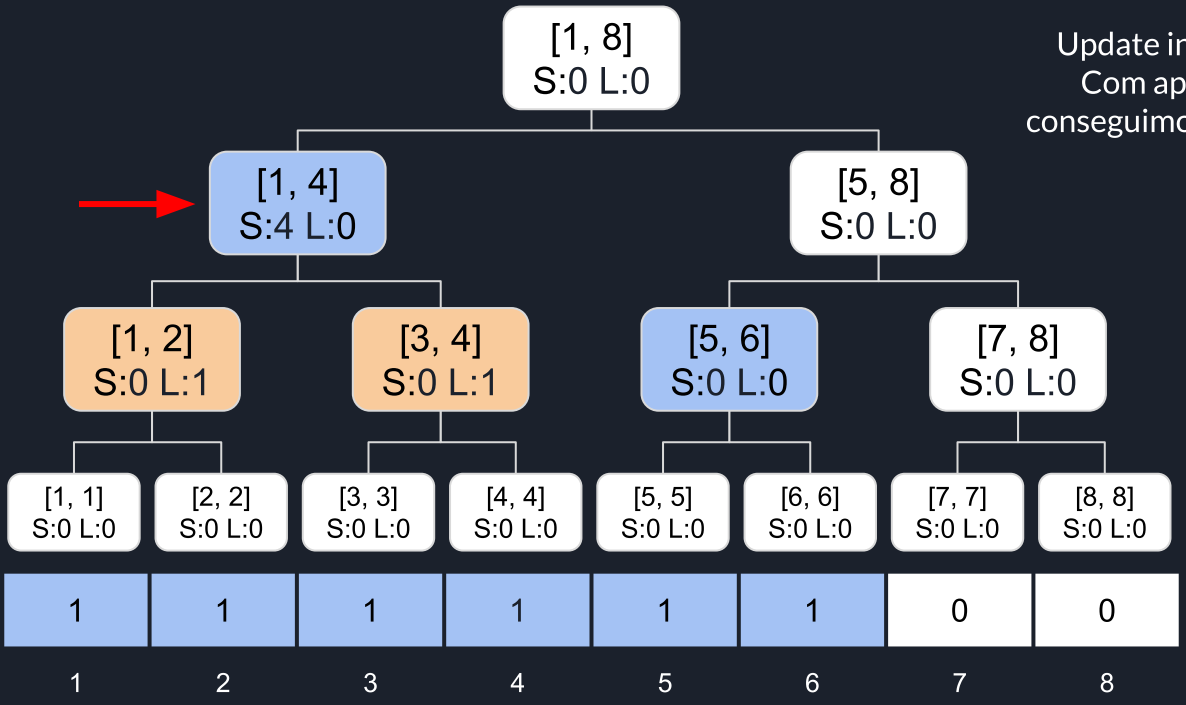
Problema HORRIBLE - exemplo



Update ini = 1 fim = 6 val = 1
Com apenas 2 intervalos conseguimos compor o intervalo [1, 6]

Lazy Propagation

Problema <u>HORRIBLE</u> - exemplo

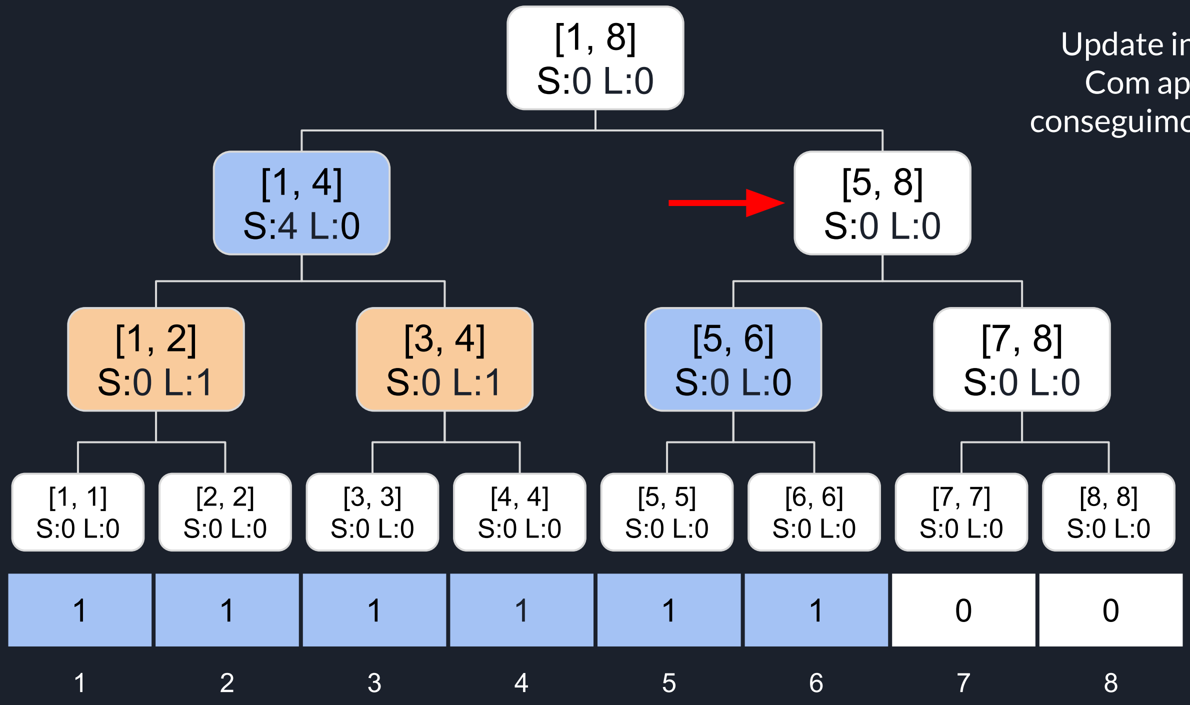


Update ini = 1 fim = 6 val = 1
Com apenas 2 intervalos conseguimos compor o intervalo [1, 6]

Lazy Propagation

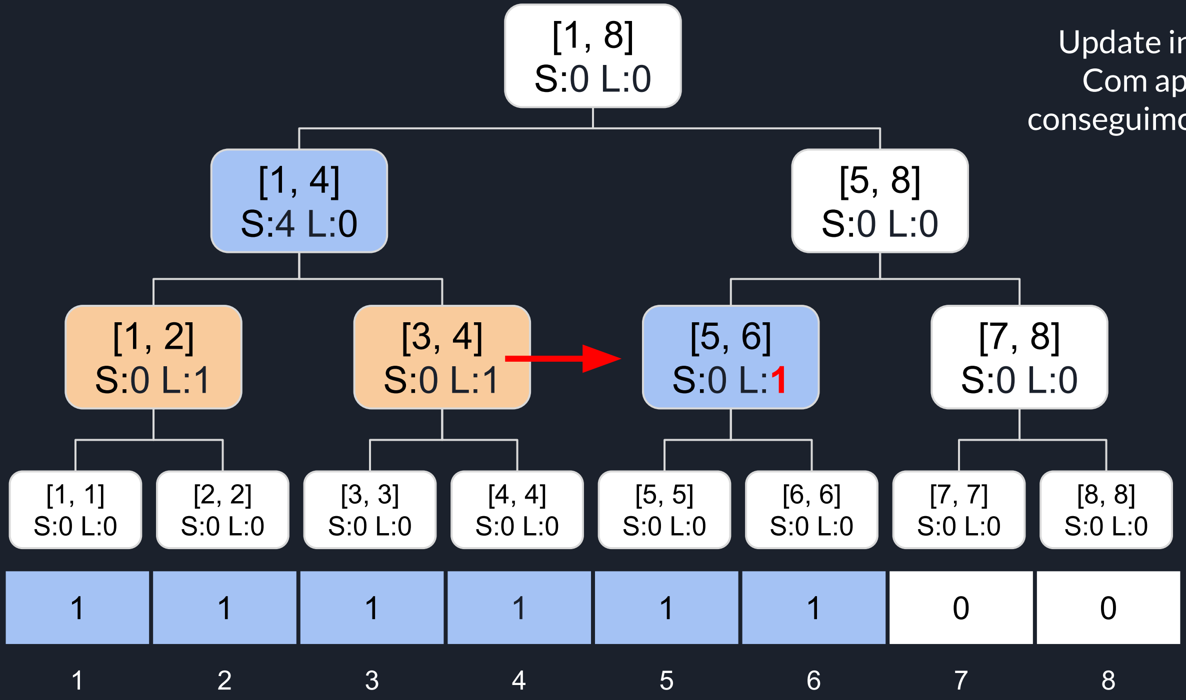
Problema <u>HORRIBLE</u> - exemplo

Update ini = 1 fim = 6 val = 1
Com apenas 2 intervalos conseguimos compor o intervalo [1, 6]



Lazy Propagation

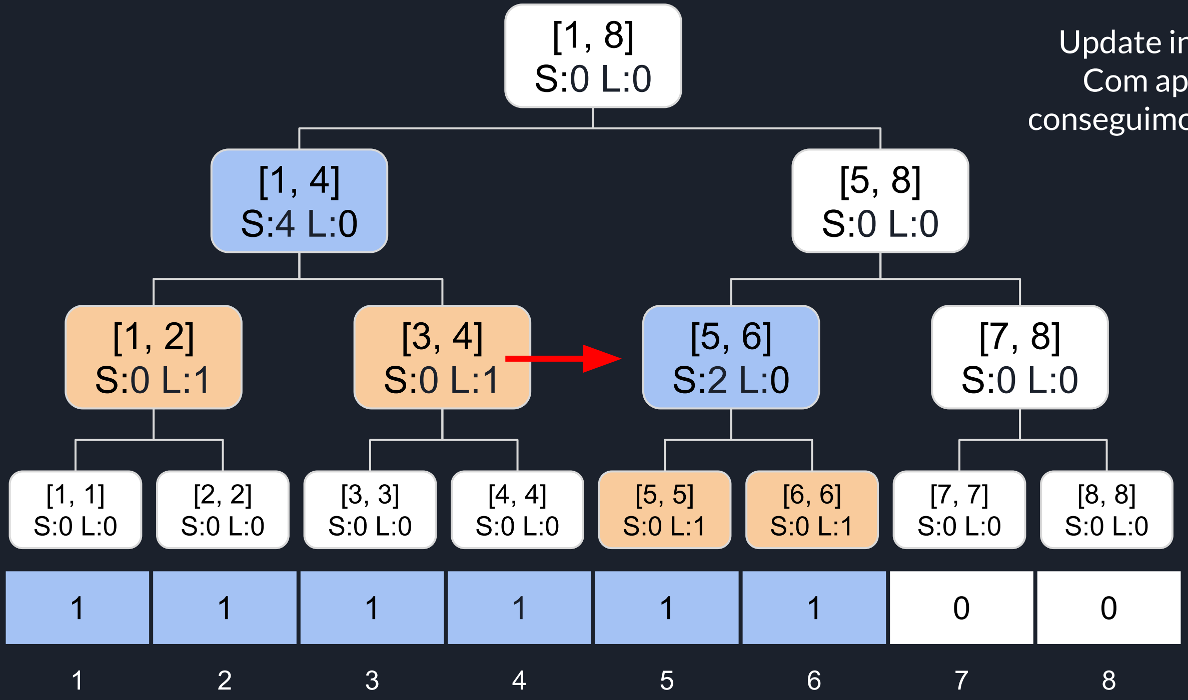
Problema <u>HORRIBLE</u> - exemplo



Update ini = 1 fim = 6 val = 1
Com apenas 2 intervalos conseguimos compor o intervalo [1, 6]

Lazy Propagation

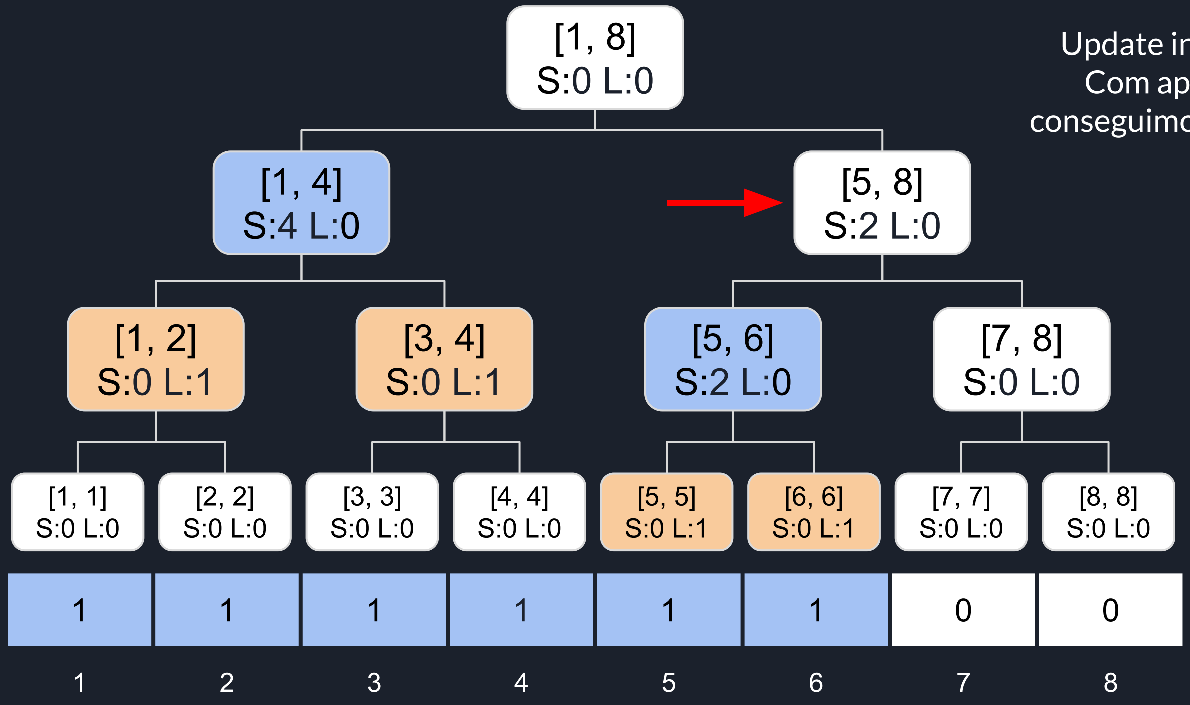
Problema <u>HORRIBLE</u> - exemplo



Update ini = 1 fim = 6 val = 1
Com apenas 2 intervalos
conseguimos compor o intervalo
[1, 6]

Lazy Propagation

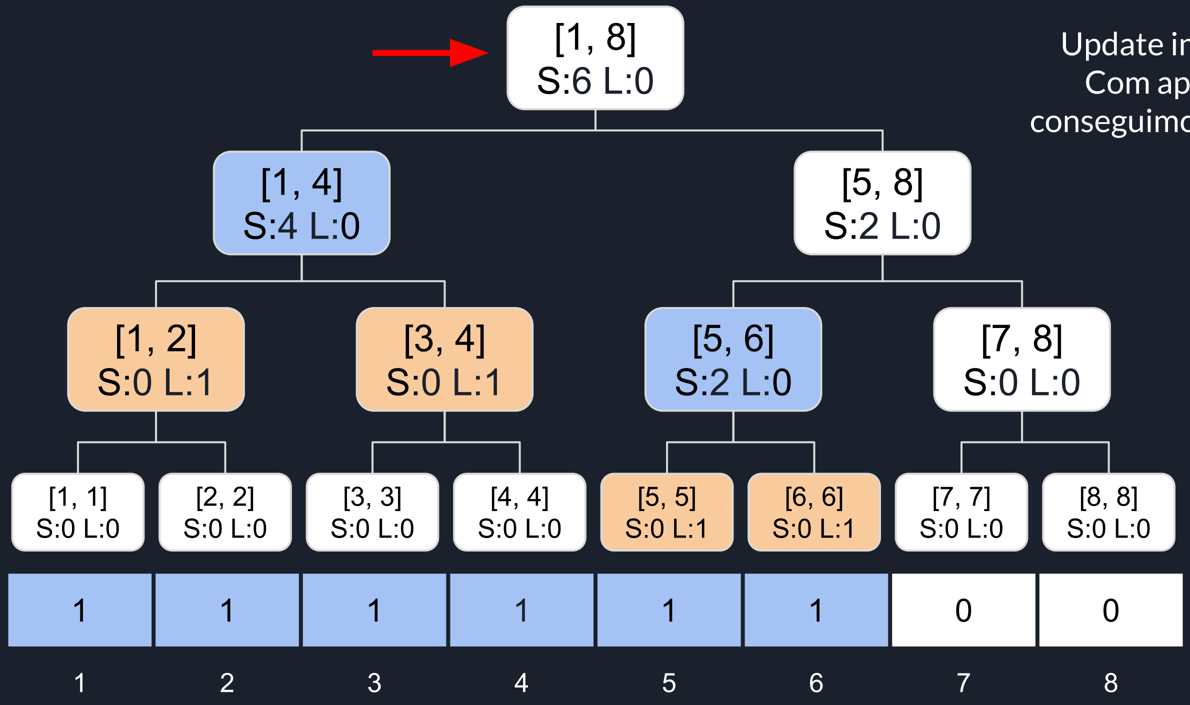
Problema HORRIBLE - exemplo



Update ini = 1 fim = 6 val = 1
Com apenas 2 intervalos conseguimos compor o intervalo [1, 6]

Lazy Propagation

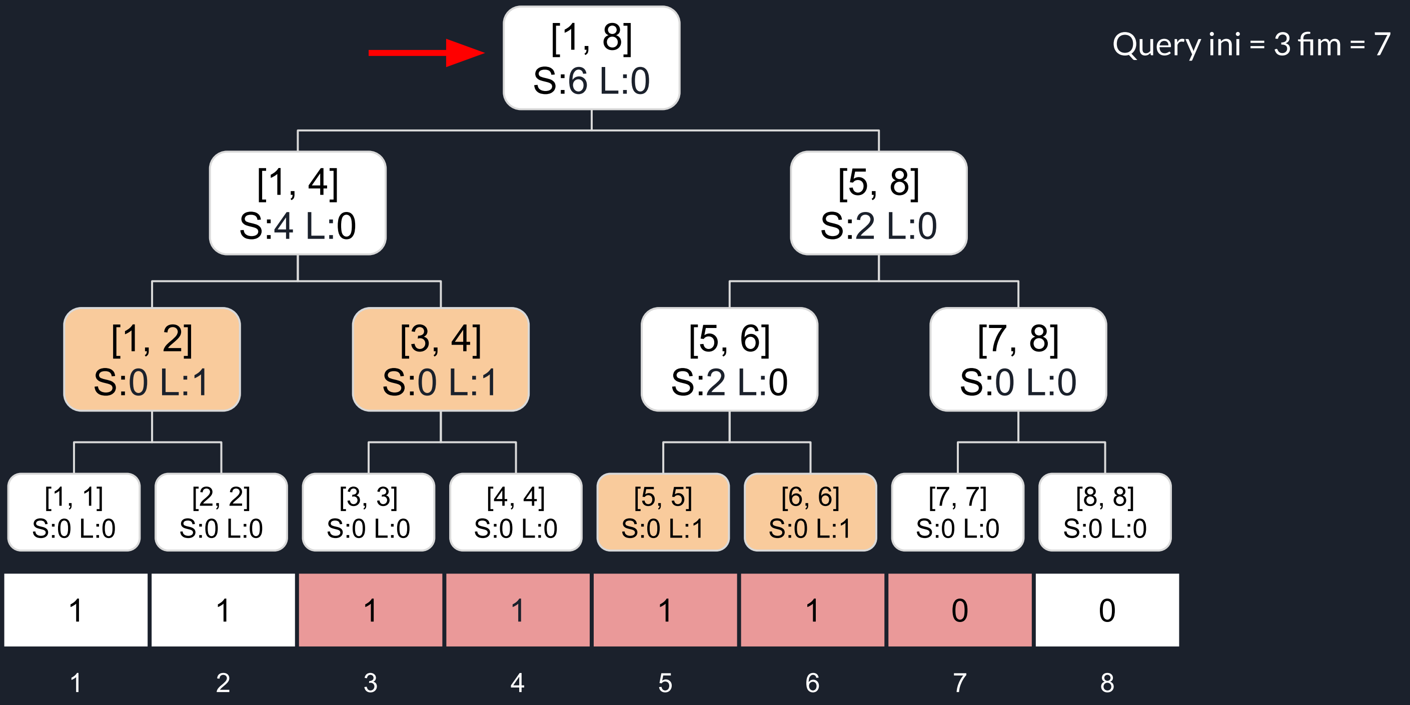
Problema <u>HORRIBLE</u> - exemplo



Update ini = 1 fim = 6 val = 1
Com apenas 2 intervalos conseguimos compor o intervalo [1, 6]

Lazy Propagation

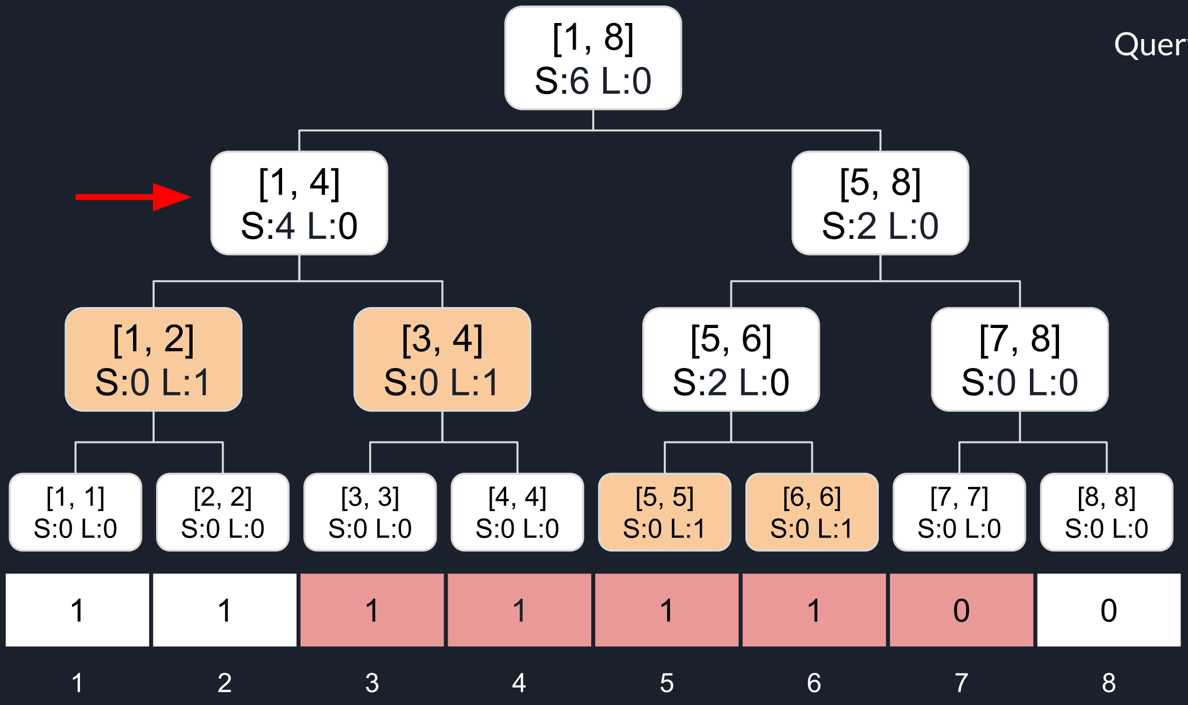
Problema HORRIBLE - exemplo



Lazy Propagation

Problema <u>HORRIBLE</u> - exemplo

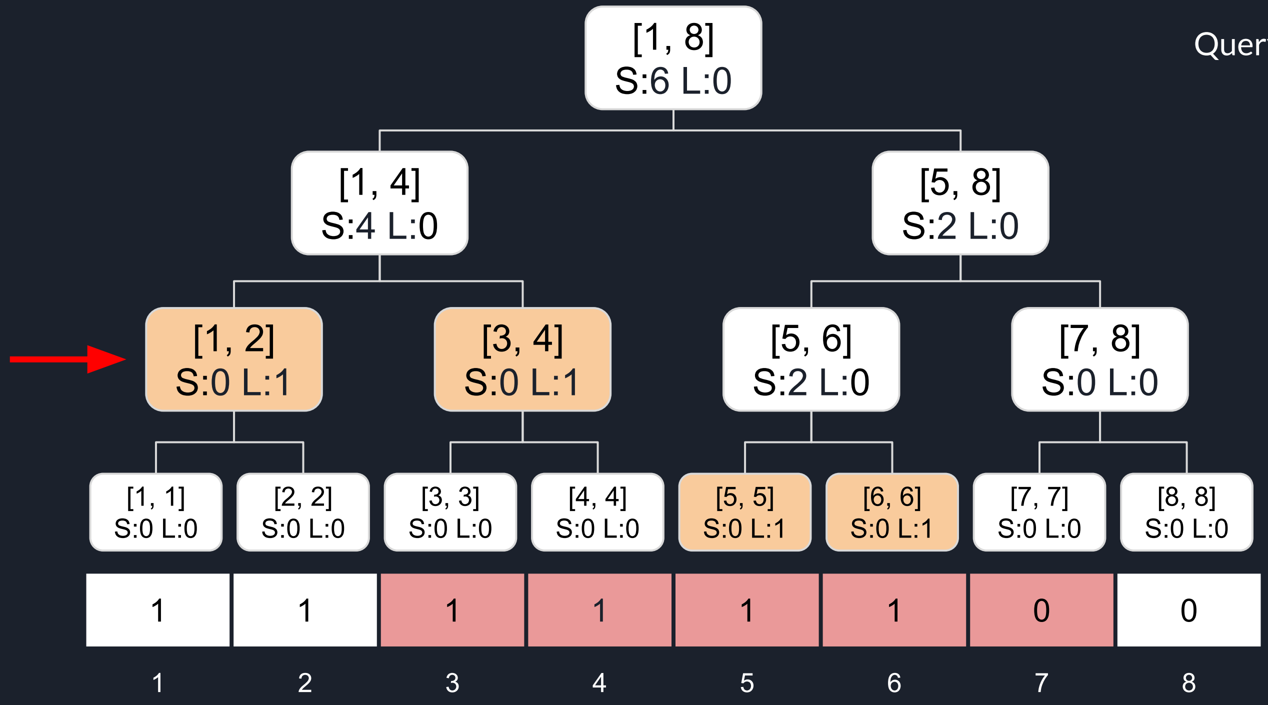
Query ini = 3 fim = 7



Lazy Propagation

Problema HORRIBLE - exemplo

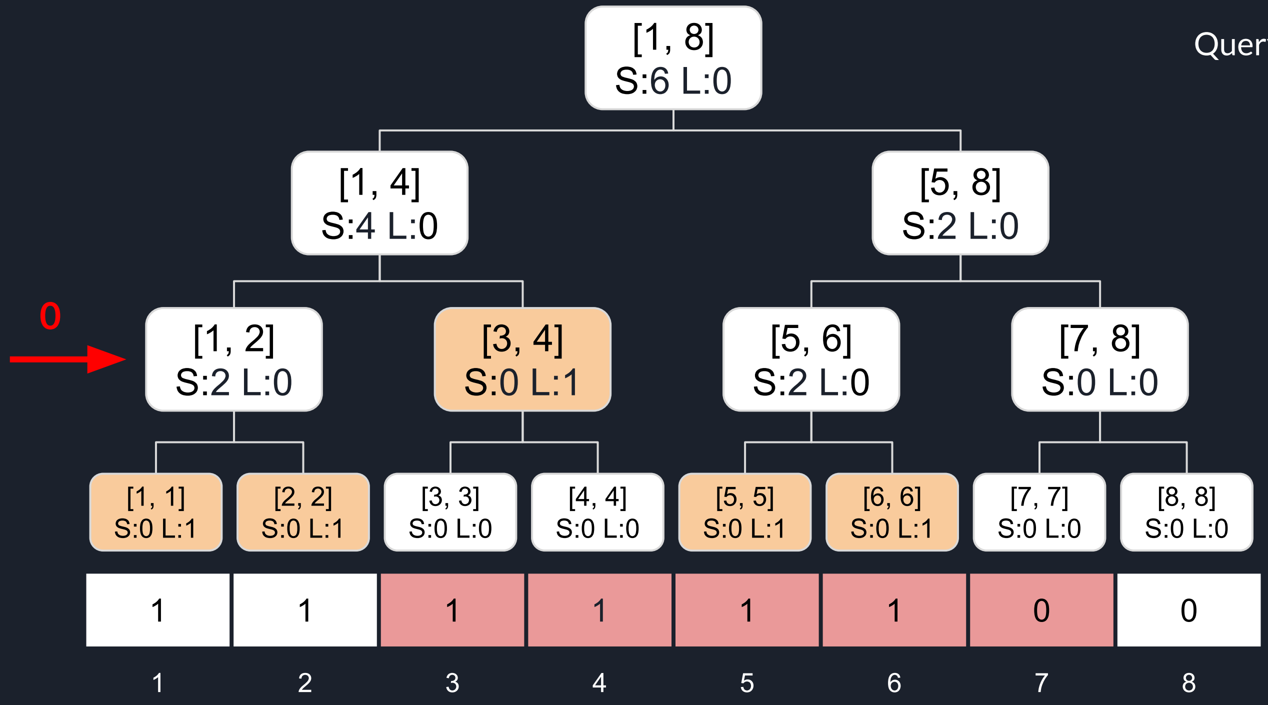
Query ini = 3 fim = 7



Lazy Propagation

Problema <u>HORRIBLE</u> - exemplo

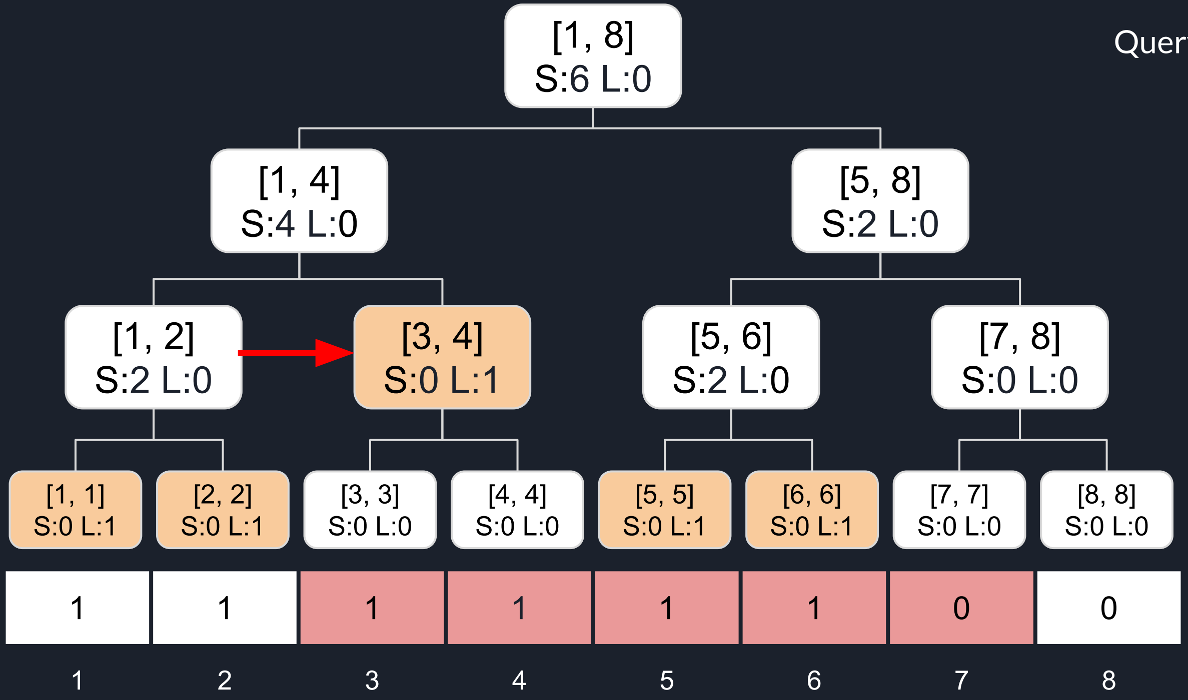
Query ini = 3 fim = 7



Lazy Propagation

Problema HORRIBLE - exemplo

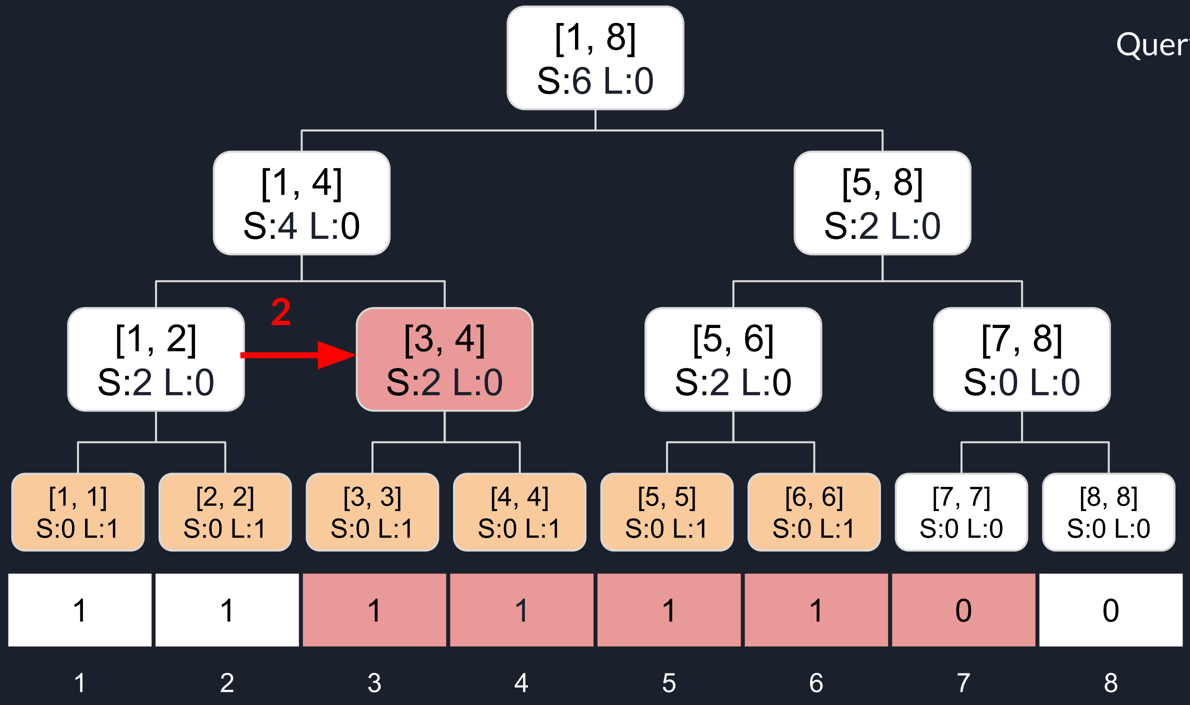
Query ini = 3 fim = 7



Lazy Propagation

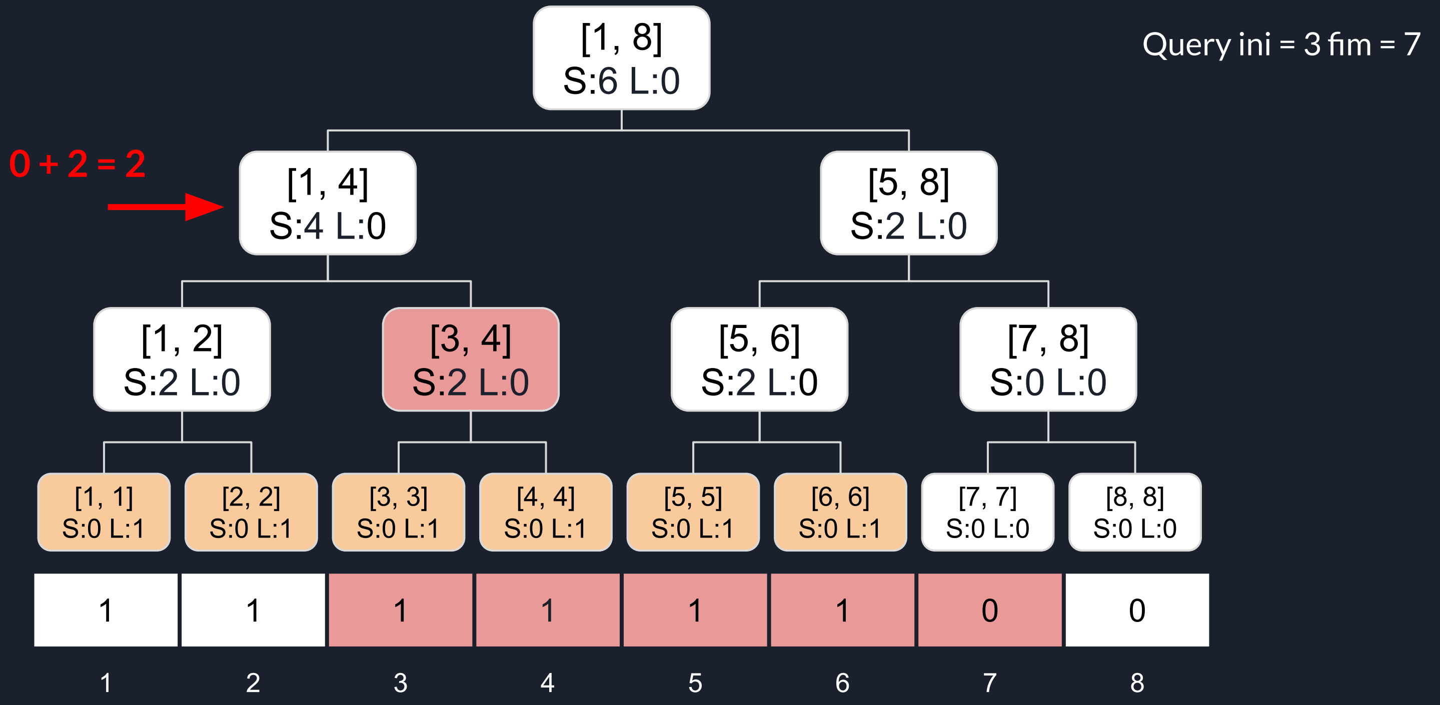
Problema <u>HORRIBLE</u> - exemplo

Query ini = 3 fim = 7



Lazy Propagation

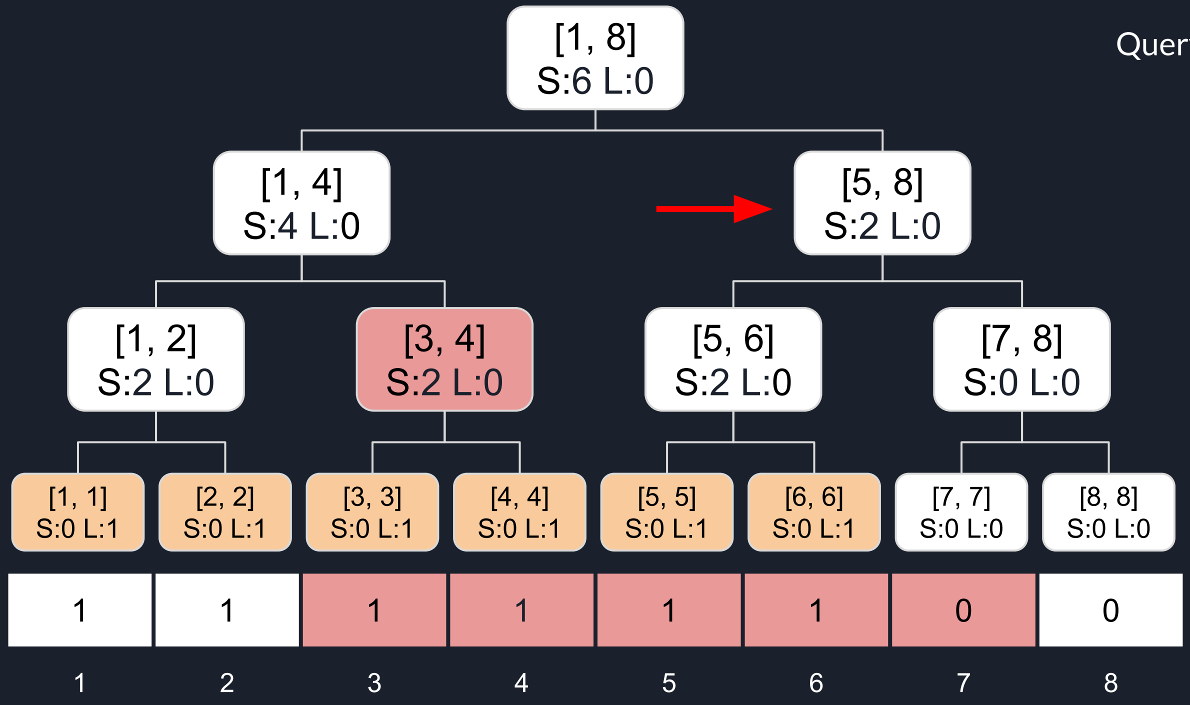
Problema HORRIBLE - exemplo



Lazy Propagation

Problema <u>HORRIBLE</u> - exemplo

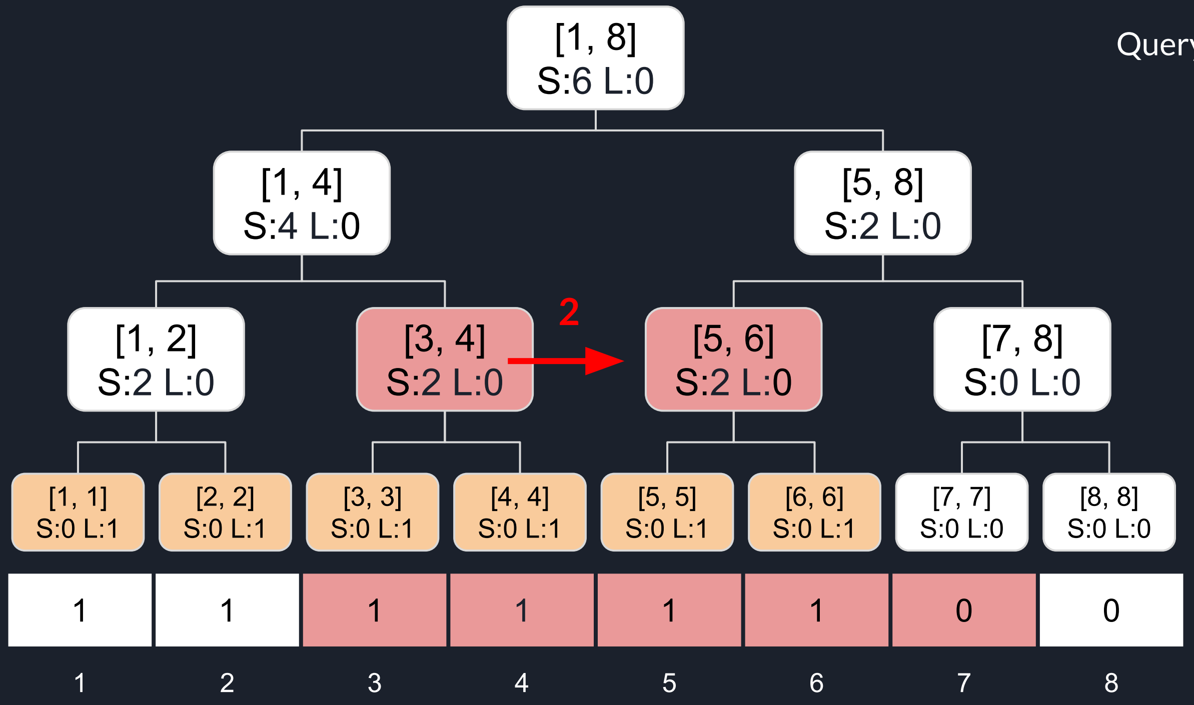
Query ini = 3 fim = 7



Lazy Propagation

Problema HORRIBLE - exemplo

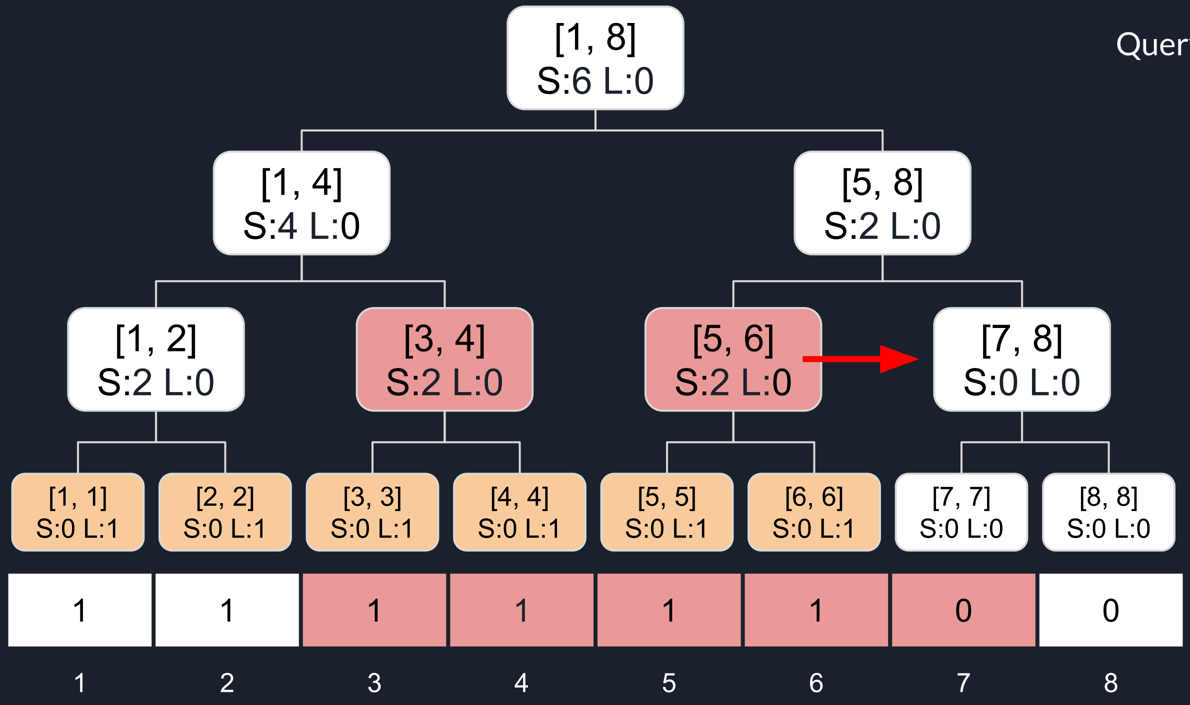
Query ini = 3 fim = 7



Lazy Propagation

Problema <u>HORRIBLE</u> - exemplo

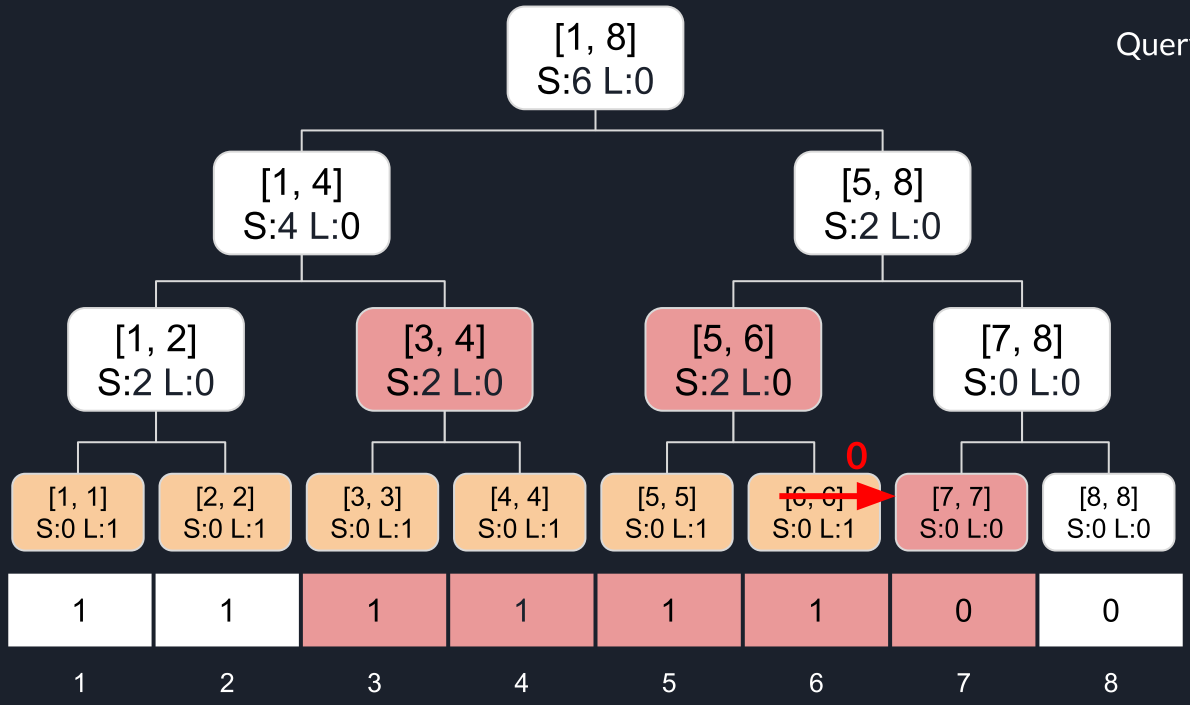
Query ini = 3 fim = 7



Lazy Propagation

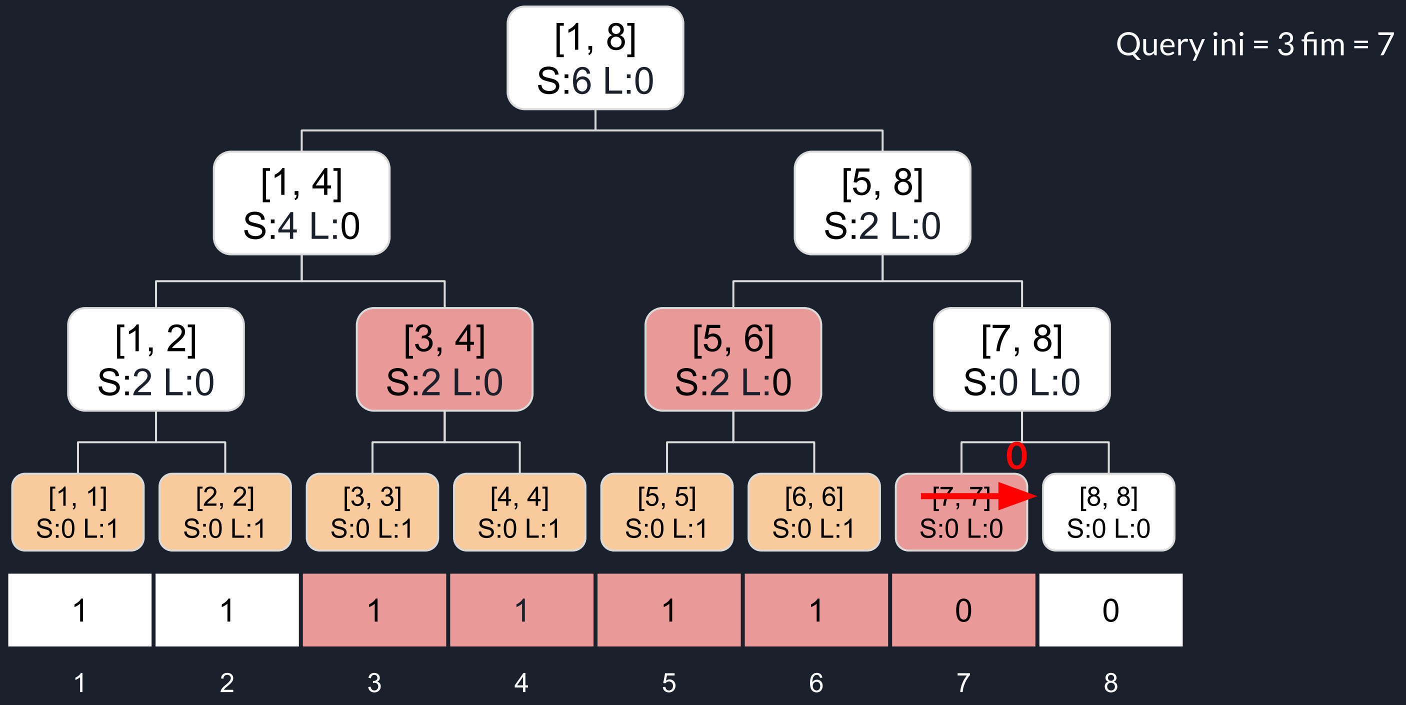
Problema <u>HORRIBLE</u> - exemplo

Query ini = 3 fim = 7



Lazy Propagation

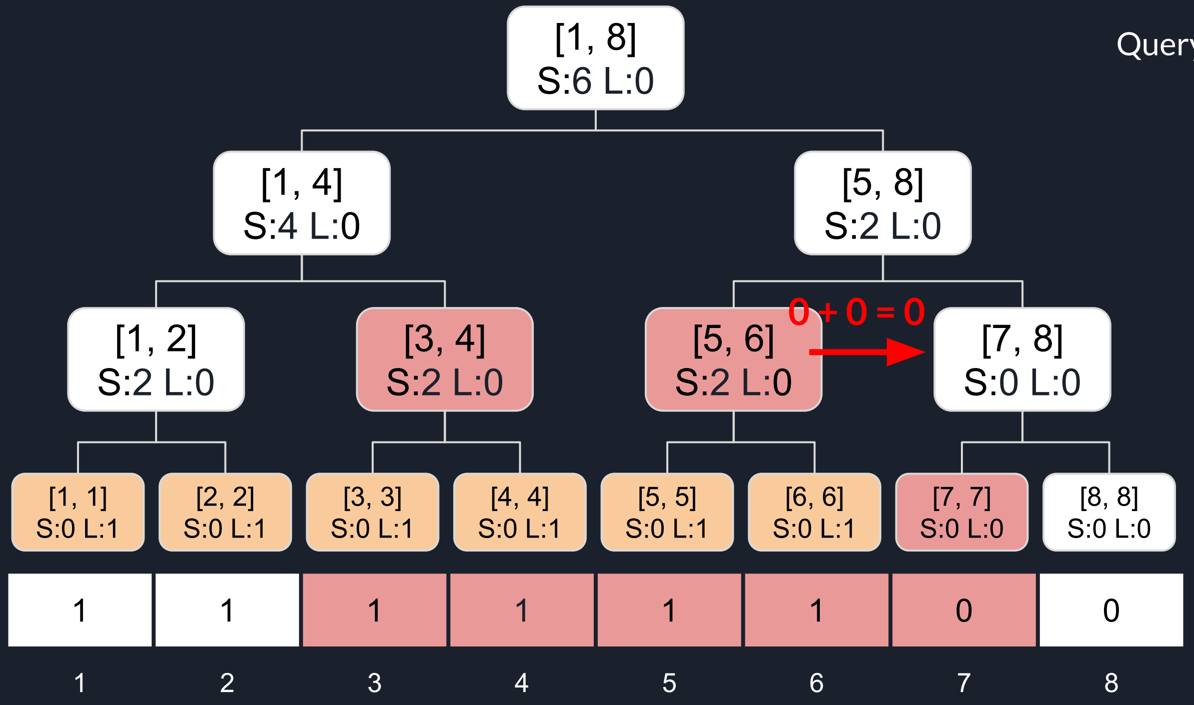
Problema HORRIBLE - exemplo



Lazy Propagation

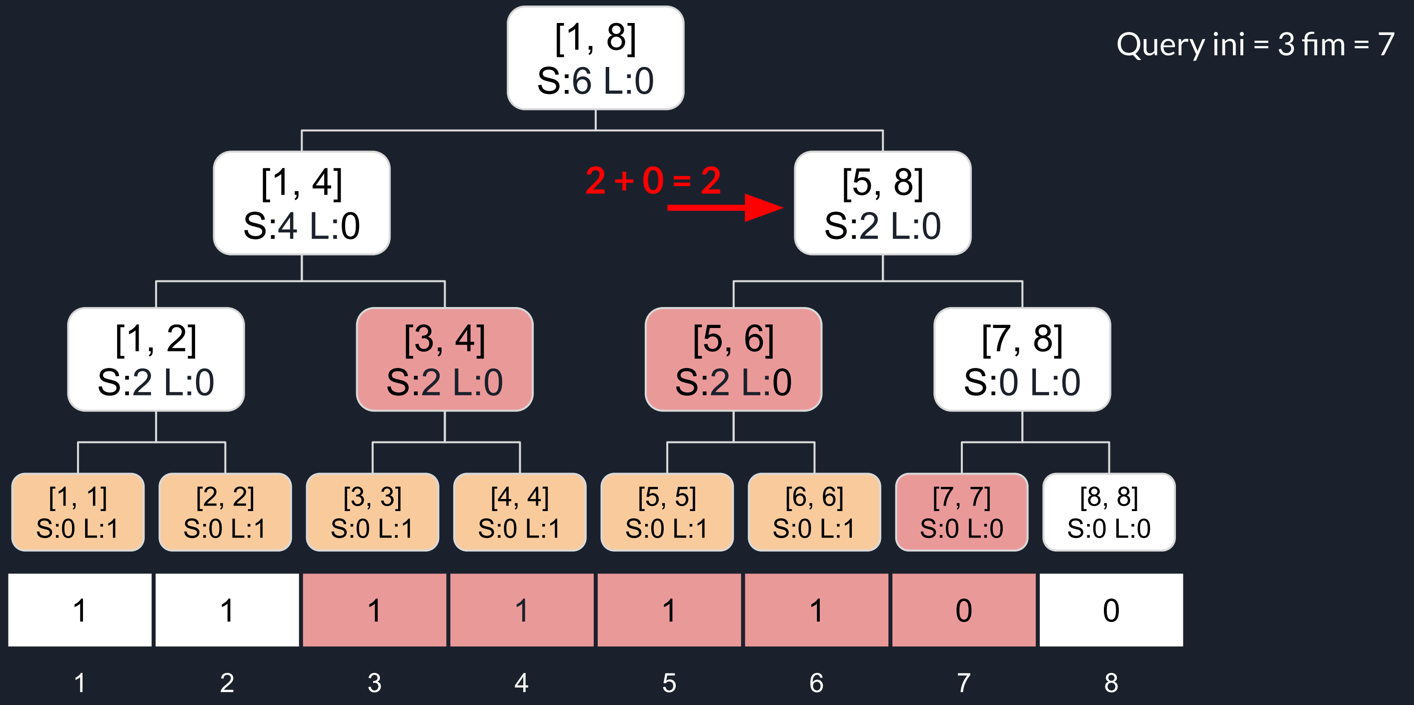
Problema HORRIBLE - exemplo

Query ini = 3 fim = 7



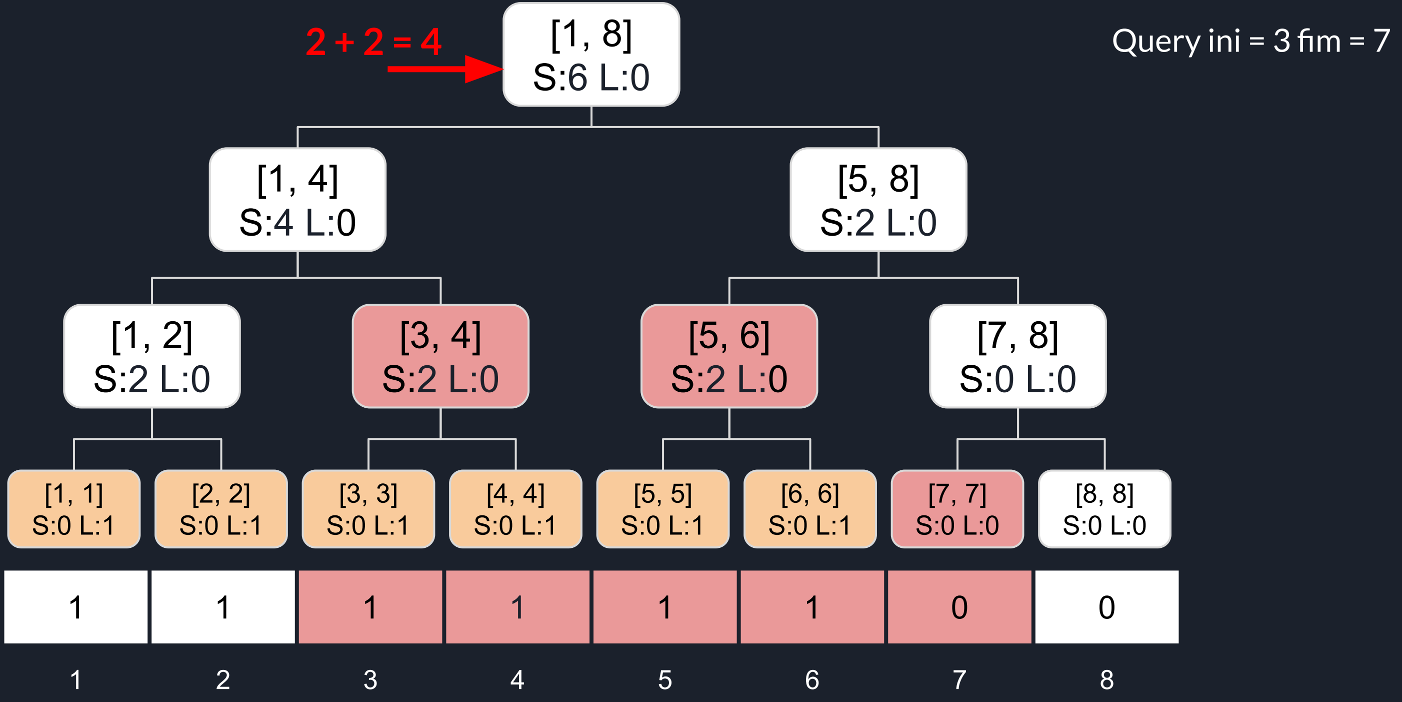
Lazy Propagation

Problema HORRIBLE - exemplo



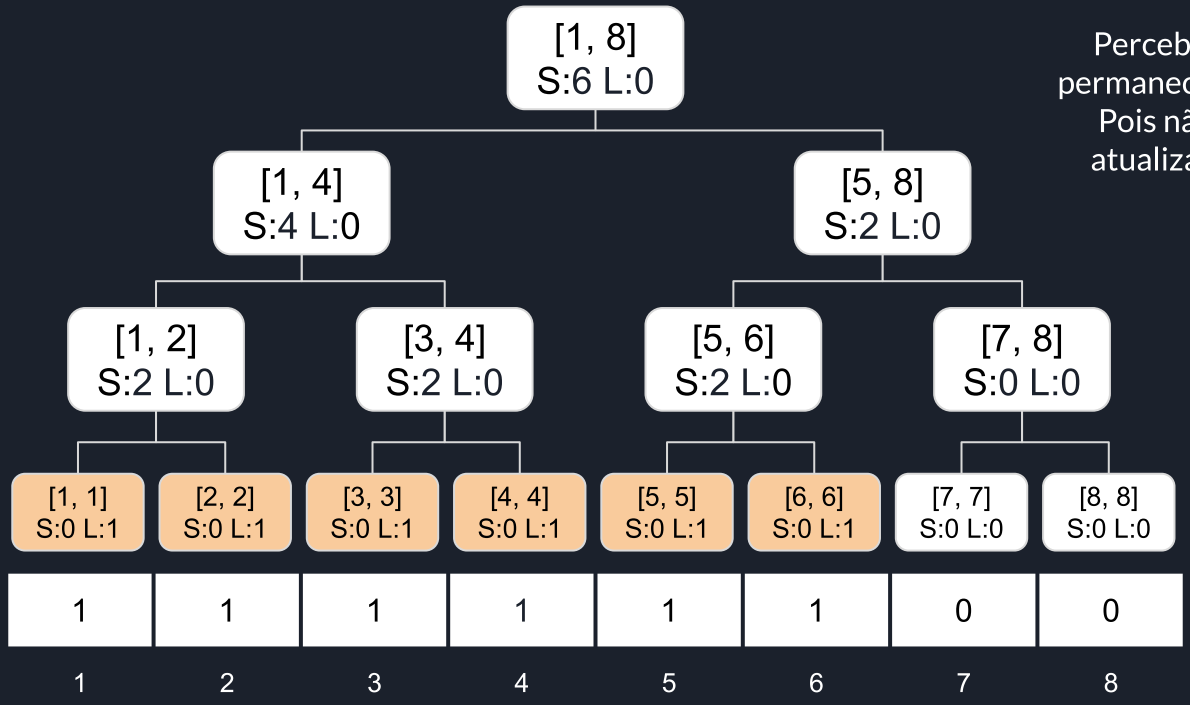
Lazy Propagation

Problema HORRIBLE - exemplo



Lazy Propagation

Problema HORRIBLE - exemplo



Perceba que alguns nós permanecem desatualizados
Pois não foi necessário atualizá-los em nenhum momento

Lazy Propagation

Lista de Exercícios

- Addition and Minimum
- Multiplication and Sum
- Assignment and Sum
- LITE
- MULTQ3
- Assignment, Addition, and Sum
- SEGSQRSS
- Problem About Weighted Sum

Busca Binária na Seg

Problema base 1

Imagine que além das outras operações na Seg (Update e Query) queremos responder:

- Query S: achar o menor índice i tal que V[1] + V[2] + ... + V[i] >= S, ou seja, o primeiro índice cuja soma de prefixo seja maior ou igual ao valor dado
- Imagine que pelas restrições do problema sempre os valores serão não negativos, ou seja, V[j] >= 0 pra todo índice j, a todo momento (apenas para simplificar)

Como podemos responder essa query ? PENSE

Busca Binária na Seg

Problema base 1

Solução 1: Para responder a esse tipo de query, podemos manter a soma de cada nó, e com isso conseguiremos responder query de soma de intervalo. Dessa forma, podemos usar Busca Binária para encontrar o índice desejado, e a cada passo da Busca Binária, fazemos uma query de soma na Seg.

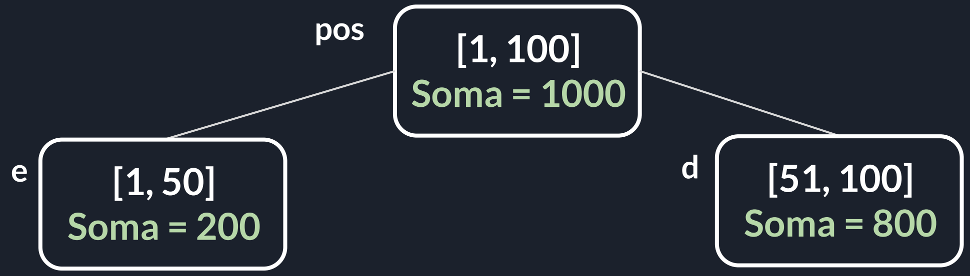
Por exemplo, suponha que sabemos que a resposta está no intervalo [10, 100], e queremos a soma 20, o próximo passo é verificar se V[1] + V[2] + ... V[55] >= 20 pois se for então sabemos que a resposta estará no intervalo [10, 55] senão estará no intervalo [56, 100]. E para determinar V[1] + V[2] + ... + V[55] podemos usar Query de soma na Seg.

A complexidade dessa operação com essa ideia fica $O(\log^2 N)$

Busca Binária na Seg

Problema base 1

Solução 2: Podemos fazer a Busca Binária diretamente na própria Seg

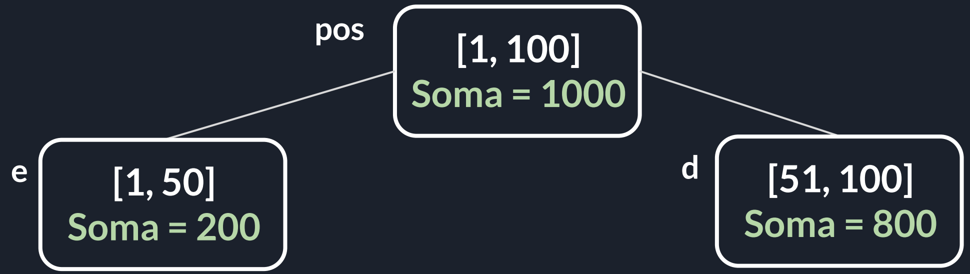


Imagine que sabemos que a resposta está dentro do intervalo do nó pos, e temos S = 100. Neste caso necessariamente a resposta estará no nó da esquerda, portanto podemos simplesmente chamar recursivamente para o nó esquerdo

Busca Binária na Seg

Problema base 1

Solução 2: Podemos fazer a Busca Binária diretamente na própria Seg



Mas imagine o caso onde S = 300, como a soma do nó esquerdo é 200, então a resposta não está lá, portanto estará no nó direito, assim devemos chamar recursivamente para o nó direito

Cuidado, ao procurar no nó direito não devemos procurar por S = 300, pois as somas são relativas ao próprio nó, e quando fomos para o nó direito, temos que considerar todo o nó esquerdo, portanto devemos procurar S = 300 - 200 = 100

Busca Binária na Seg

Problema base 1

```
int bb(int pos, int ini, int fim, int s) {
  if(ini == fim) return ini;

  int m = (ini + fim)/2;
  int e = 2*pos, d = 2*pos + 1;

  if(seg[e].sum >= s) return bb(e, ini, m, s)
  else return bb(d, m + 1, fim, s - seg[e].sum);
}
```

Busca Binária na Seg

Problema base 1

Considerações:

- **Nessa função estamos assumindo que a resposta está entre ini e fim, portanto, antes de chamar a função para a raiz, temos que se certificar que a resposta existe (comparando S com a soma da raiz)**
- **A complexidade fica O(logN) pois a cada nível da árvore iremos ou para o filho esquerdo, ou para o filho direito**
- **Também é possível fazer Busca Binária na BIT, mas a Seg é bem mais maleável**
  - **Por exemplo é fácil trocar para encontrar a primeira soma de sufixo que seja maior ou igual a S, basta dar prioridade para o filho direito ao invés do esquerdo**

Busca Binária na Seg

Problema base 2

Imagine que além das outras operações na Seg (Update e Query) queremos responder:

- Query H i: achar o menor índice j tal que j >= i e V[j] >= H, ou seja, o primeiro índice após i (podendo ser ele) cujo valor seja maior ou igual à um valor dado

Como podemos responder essa query ? PENSE

Busca Binária na Seg

Problema base 2

Solução 1: Para responder a esse tipo de query, podemos manter o valor máximo em cada nó, e com isso conseguiremos responder query de máximo de intervalo. Dessa forma, podemos usar Busca Binária para encontrar o índice desejado, e a cada passo da Busca Binária, fazemos uma query de máximo na Seg.

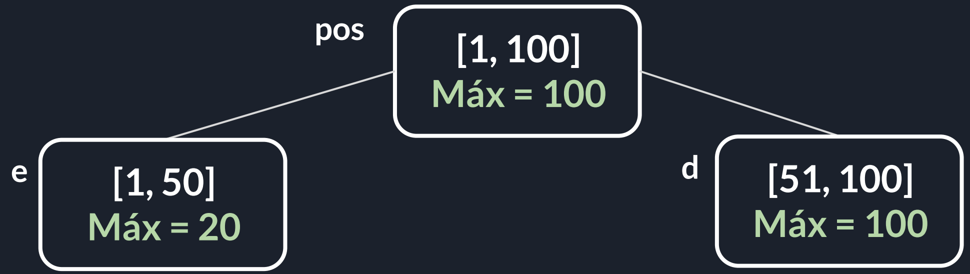
Por exemplo, suponha que i = 10 e sabemos que a resposta está no intervalo [20, 100], e temos H = 200, o próximo passo é verificar se máximo(V[10], V[11], ..., V[60]) >= 200, pois se for então sabemos que a resposta estará no intervalo [20, 60] senão estará no intervalo [61, 100]. E para determinar máximo(V[10], V[11], ..., V[60]) podemos usar Query de máximo na Seg.

A complexidade dessa operação com essa ideia fica $O(\log^2 N)$

Busca Binária na Seg

Problema base 2

Solução 2: Podemos fazer a Busca Binária diretamente na própria Seg



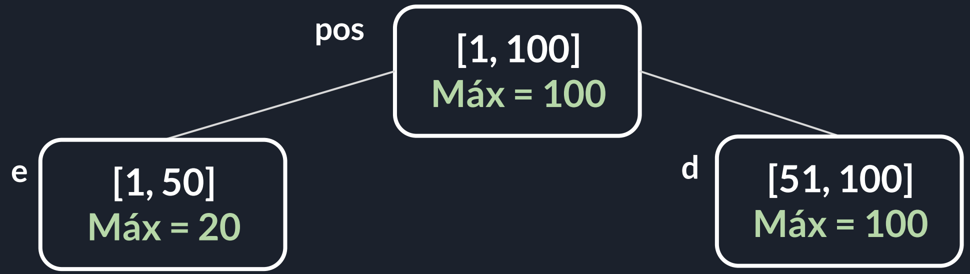
Imagine que sabemos que a resposta está dentro do intervalo do nó pos, e temos H = 100 e i = 70.

Neste caso necessariamente a resposta estará no nó da direita, portanto podemos simplesmente chamar recursivamente para o nó direito

Busca Binária na Seg

Problema base 2

Solução 2: Podemos fazer a Busca Binária diretamente na própria Seg



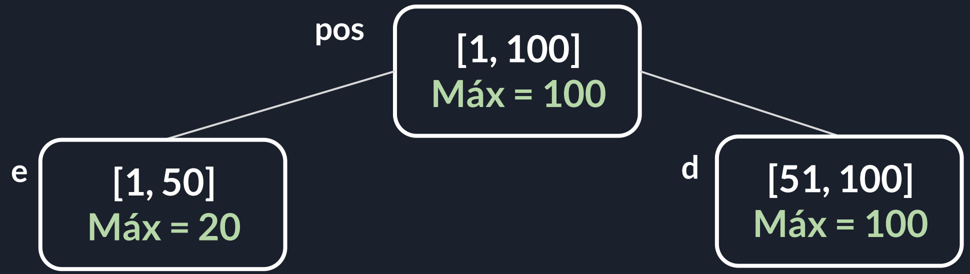
Agora imagine que sabemos que a resposta está dentro do intervalo do nó pos, e temos H = 100 e i = 20.

Neste caso necessariamente a resposta também estará no nó da direita, pois não existe ninguém maior ou igual a H no nó esquerdo, portanto podemos simplesmente chamar recursivamente para o nó direito

Busca Binária na Seg

Problema base 2

Solução 2: Podemos fazer a Busca Binária diretamente na própria Seg



Agora imagine que sabemos que a resposta está dentro do intervalo do nó pos, e temos H = 15 e i = 20.

Neste caso pode ser que a resposta esteja no nó esquerdo ou no direito. Portanto devemos chamar recursivamente para o nó esquerdo, e se encontrarmos uma resposta retornamos ela. Mas se não encontrarmos uma resposta, então chamamos recursivamente para o nó direito

Busca Binária na Seg

Problema base 2

```
int bb(int pos, int ini, int fim, int i, int H) {
  if(seg[pos].max < H) return -1;

  if(ini == fim) return ini;

  int m = (ini + fim)/2;
  int e = 2*pos, d = 2*pos + 1;

  if(i > m) return bb(d, m + 1, fim, i, H);

  int tmp = bb(e, ini, m, i, H);

  if(tmp != -1) return tmp;

  return bb(d, m + 1, fim, m + 1, H);
}
```

Busca Binária na Seg

Problema base 2

Considerações:

- **Diferente do problema anterior, nessa função não estamos assumindo que a resposta exista, pois se não existir ela retornará -1.**
- **Qual a complexidade dessa função ? PENSE**

Busca Binária na Seg

Problema base 2

Observação 1: Se tivermos i = ini e a resposta da função for -1 (ou seja não existe resposta), então será O(1).

Dadas essas duas condições, necessariamente iremos cair na linha 2, onde o máximo do intervalo inteiro será menor que H

```
1  int bb(int pos, int ini, int fim, int i, int H) {
2    if(seg[pos].max < H) return -1;
3
4    if(ini == fim) return ini;
5
6    int m = (ini + fim)/2;
7    int e = 2*pos, d = 2*pos + 1;
8
9    if(i > m) return bb(d, m + 1, fim, i, H);
10
11   int tmp = bb(e, ini, m, i, H);
12
13   if(tmp != -1) return tmp;
14
15   return bb(d, m + 1, fim, m + 1, H);
16 }
```

Busca Binária na Seg

Problema base 2

Observação 2: Se resposta da função for -1 (ou seja não existe resposta), então é O(altura)

Se for cai na linha 2 é O(1).

Se cair na linha 9, chamou recursivamente apenas para o filho direito, é O(altura) [na verdade precisaria formalizar com uma indução na altura]

```
1  int bb(int pos, int ini, int fim, int i, int H) {
2    if(seg[pos].max < H) return -1;
3
4    if(ini == fim) return ini;
5
6    int m = (ini + fim)/2;
7    int e = 2*pos, d = 2*pos + 1;
8
9    if(i > m) return bb(d, m + 1, fim, i, H);
10
11   int tmp = bb(e, ini, m, i, H);
12
13   if(tmp != -1) return tmp;
14
15   return bb(d, m + 1, fim, m + 1, H);
16 }
```

Se cair na linha 11, depois a linha 15 será O(1), pois cai na Observação 1. Portanto é O(altura).

Note que praticamente sempre houve apenas uma chamada recursiva para um dos filhos (desconsiderando as chamadas que serão O(1)).

Busca Binária na Seg

Problema base 2

Observação 3: Se i = ini, então é O(altura)

Se for dar -1 cai na linha 2 é O(1).

Senão, não vai cair na linha 9 pois i = ini, então vamos analisar tmp.

Se tmp não for -1, será retornado na linha 13, e teremos O(altura)

```
1  int bb(int pos, int ini, int fim, int i, int H) {
2    if(seg[pos].max < H) return -1;
3
4    if(ini == fim) return ini;
5
6    int m = (ini + fim)/2;
7    int e = 2*pos, d = 2*pos + 1;
8
9    if(i > m) return bb(d, m + 1, fim, i, H);
10
11   int tmp = bb(e, ini, m, i, H);
12
13   if(tmp != -1) return tmp;
14
15   return bb(d, m + 1, fim, m + 1, H);
16 }
```

Se tmp for -1, então a linha 11 gastou O(1), e na linha 15 teremos a chamada recursiva para o filho direito

Novamente temos que praticamente sempre houve apenas uma chamada recursiva para um dos filhos

Busca Binária na Seg

Problema base 2

Observação 4: Apenas um nó pode chamar tanto pro filho esquerdo quanto pro direito, e cada um gastar O(altura) em cada.

Para isso acontecer, tem que chegar à linha 11, o tmp tem que dar -1, assim na linha 11 gasta O(altura) e de acordo com a Observação 2, praticamente chama apenas para um dos filhos (dali em diante)

```
1  int bb(int pos, int ini, int fim, int i, int H) {
2    if(seg[pos].max < H) return -1;
3
4    if(ini == fim) return ini;
5
6    int m = (ini + fim)/2;
7    int e = 2*pos, d = 2*pos + 1;
8
9    if(i > m) return bb(d, m + 1, fim, i, H);
10
11   int tmp = bb(e, ini, m, i, H);
12
13   if(tmp != -1) return tmp;
14
15   return bb(d, m + 1, fim, m + 1, H);
16 }
```

Depois chama pro filho direito, na linha 15, mas com i = ini o que também é O(altura) e, de acordo com a Observação 3, praticamente chama apenas para um dos filhos (dali em diante).

Conclusão, só pode ocorrer em apenas um único nó, logo a complexidade fica O(2*altura) = O(2*logN)

Busca Binária na Seg

Lista de Exercícios

- K-th one
- First element at least X
- First element at least X - 2
- ORDERSET
- ORDERS
- Nikita and stack