Brazilian ICPC Summer School 2022

Segment Tree Aula 2:

- Problema Clássico: Área da União de Retângulos
- Merge Sort Tree
- Segment Tree Dinâmica
- Segment Tree Persistente

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

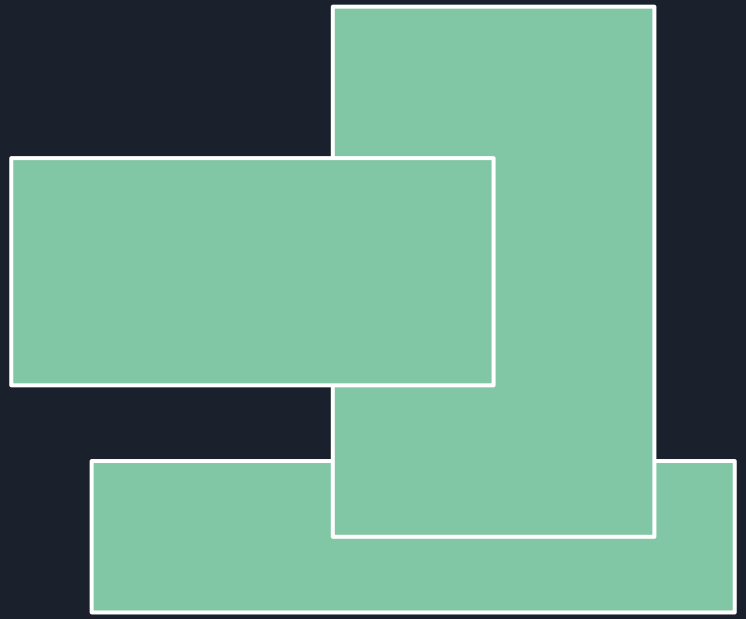
Referências:

- 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)

Problema Clássico

Área da União de Retângulos

Dados N retângulos, calcule a área da União deles

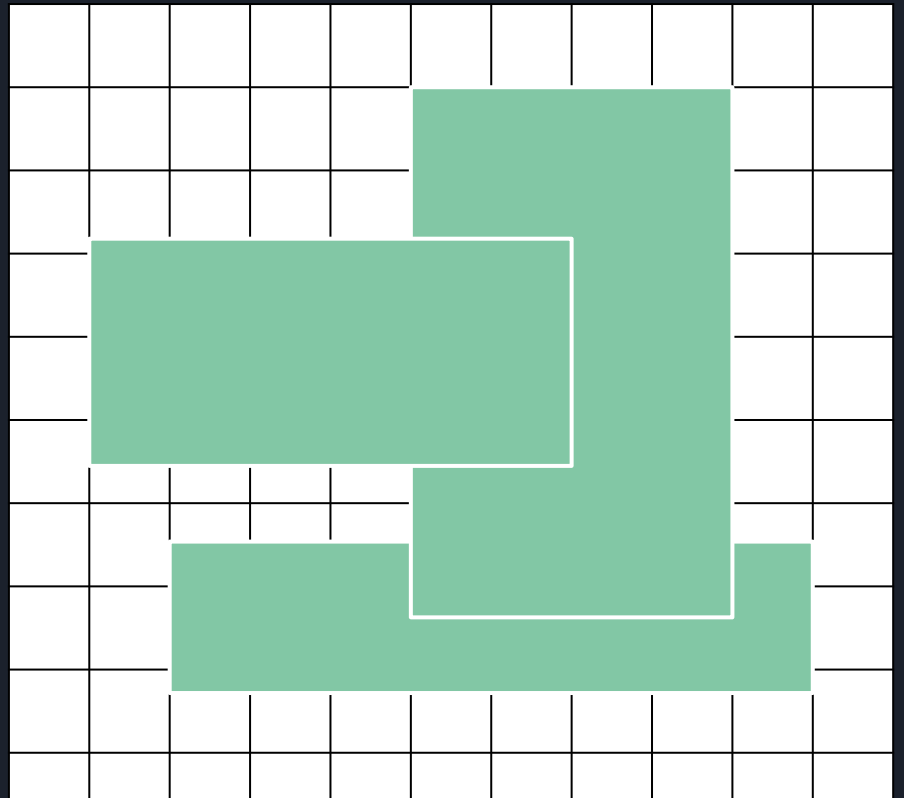


Problema Clássico

Área da União de Retângulos

Dados N retângulos, calcule a área da União deles.

Se os limites forem pequenos, por exemplo coordenadas entre 1 e 100, e até 100 retângulos ? PENSE



Problema Clássico

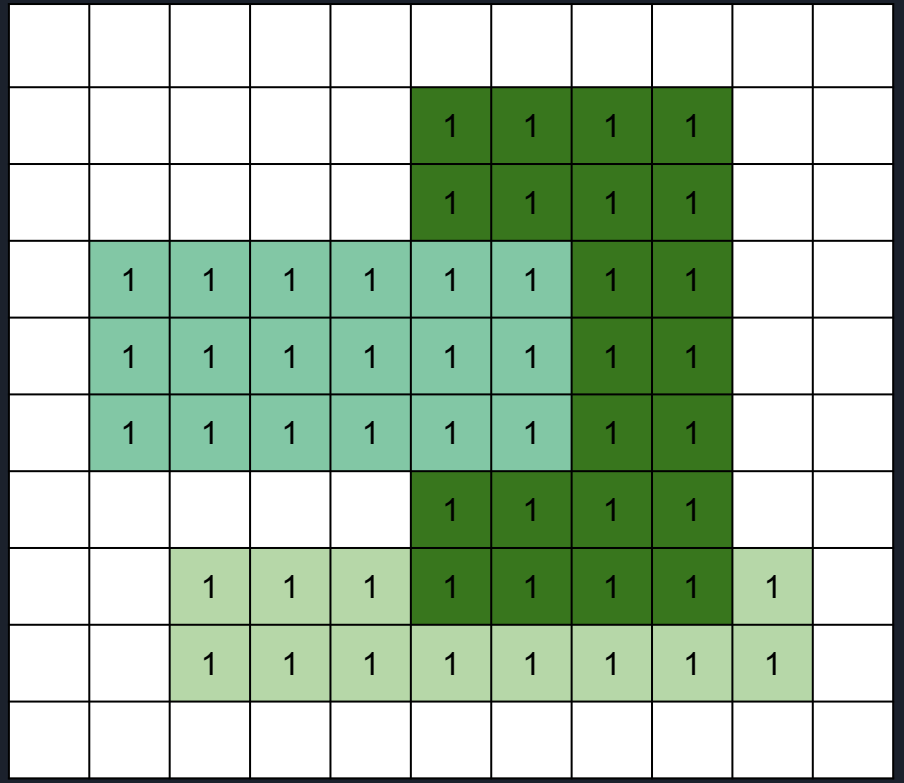
Área da União de Retângulos

Se os limites forem pequenos, por exemplo coordenadas entre 1 e 100, e até 100 retângulos, podemos criar uma matriz de marcação

Para cada retângulo, basta passar por cada quadradinho e marcar com 1, ao final basta contar o número de 1's. No exemplo ao lado a resposta é 52.

Esse é exatamente o problema PESCA11

Mas e se as coordenadas fossem de 1 à 10^5 e até 10^5 retângulos ? PENSE



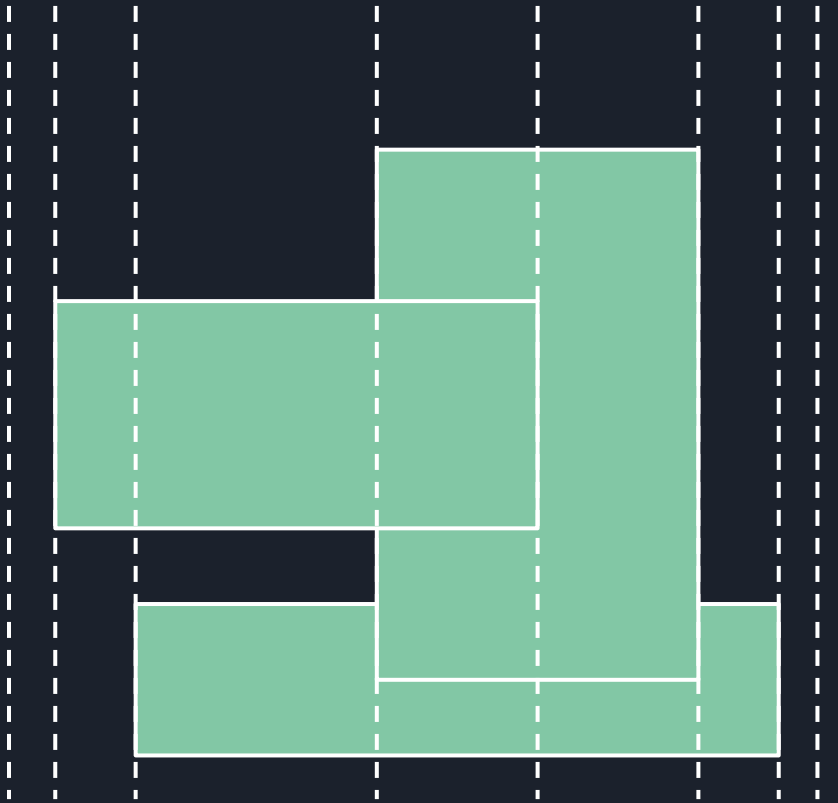
Problema Clássico

Área da União de Retângulos

Mas e se as coordenadas fossem de 1 à 10^5 e até 10^5 retângulos ?

A ideia é usar uma Line Sweep. Imagine uma linha que varre a figura, da esquerda para a direita, e vai calculando as áreas.

Vamos supor que a gente consiga manter a cada momento o tamanho da Line que está dentro de algum retângulo (depois discutiremos os detalhes). Os únicos momentos que há alteração são os momentos de entrada/saída de retângulos (entrada é no menor x de um retângulo, e saída no maior x)

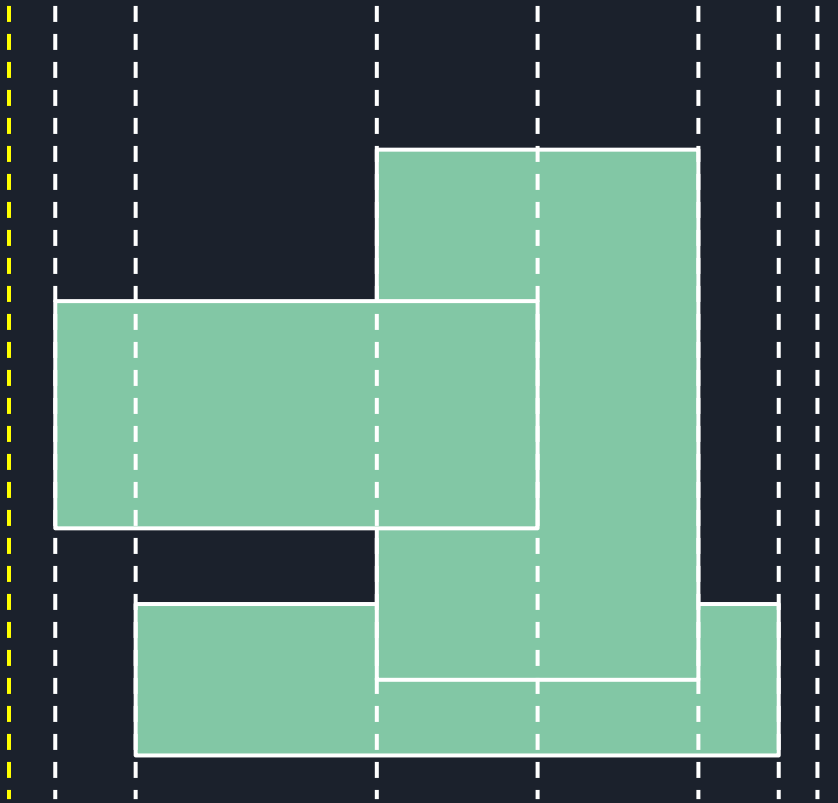


Problema Clássico

Área da União de Retângulos

Nesses momentos de alteração da Line, a gente vai calculando as áreas dos retângulos, vejamos um exemplo:

A Line começa à esquerda



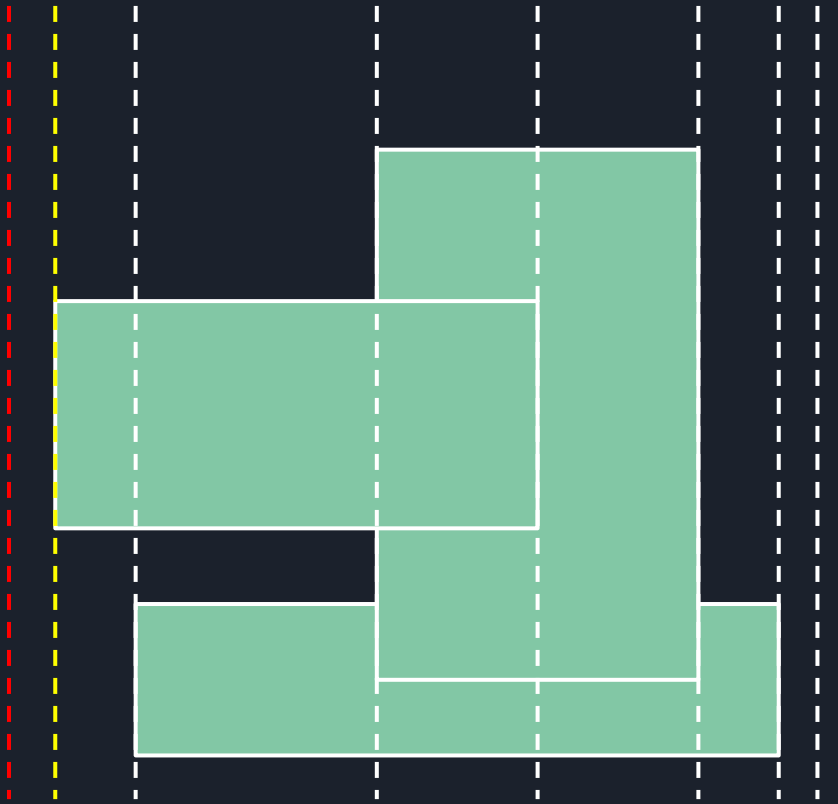
Problema Clássico

Área da União de Retângulos

Nesses momentos de alteração da Line, a gente vai calculando as áreas dos retângulos, vejamos um exemplo:

A Line começa à esquerda

Quando ela chega no momento 1 (linha em amarelo), como no momento anterior (linha em vermelho) não havia nenhuma parte da Line coberta, então soma 0



Problema Clássico

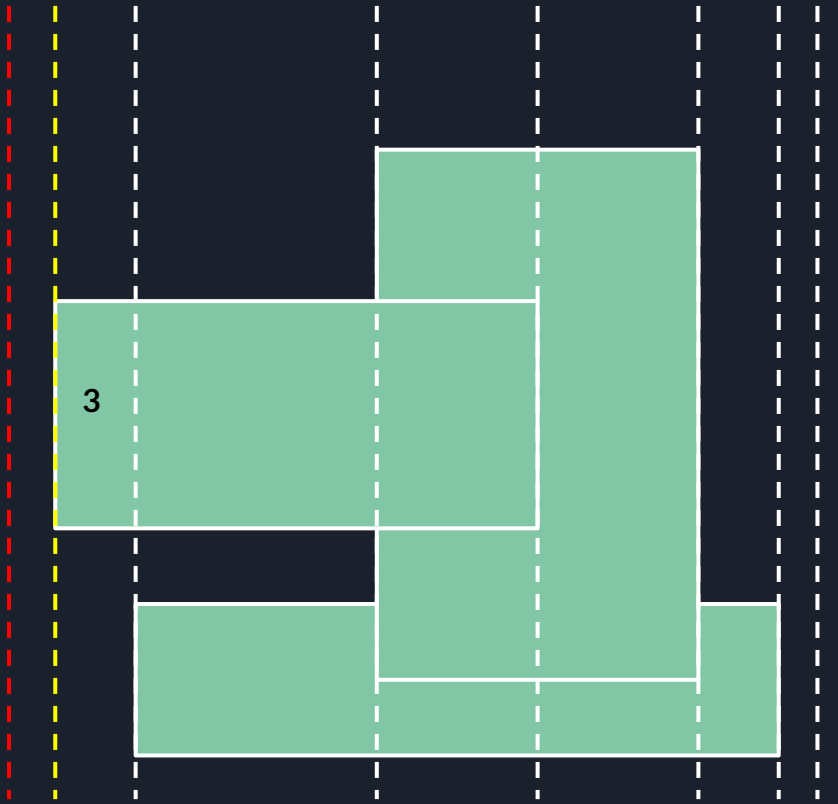
Área da União de Retângulos

Nesses momentos de alteração da Line, a gente vai calculando as áreas dos retângulos, vejamos um exemplo:

A Line começa à esquerda

Quando ela chega no momento 1 (linha em amarelo), como no momento anterior (linha em vermelho) não havia nenhuma parte da Line coberta, então soma 0.

Nesse momento o primeiro retângulo entra na Line, e de alguma forma calculamos (veremos depois) que o tamanho coberto da line é 3

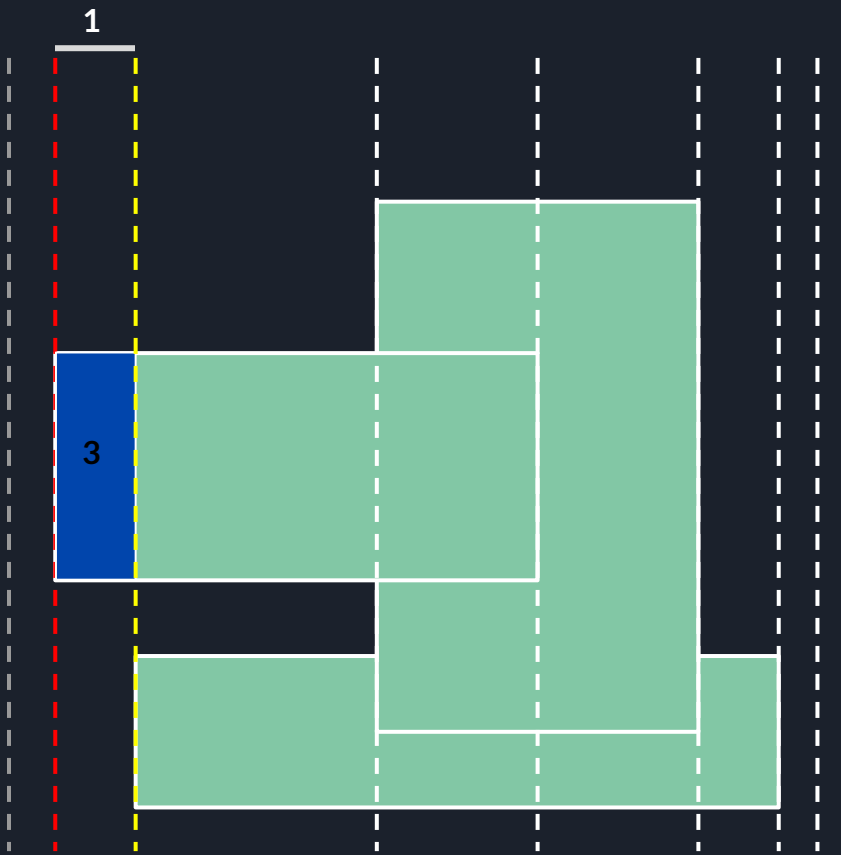


Problema Clássico

Área da União de Retângulos

No próximo momento (entrada do retângulo 2), calculamos a área dos retângulos entre esse momento e o anterior.

Para isso basta multiplicar o tamanho da Line que estava dentro de algum retângulo no momento anterior, no caso 3, pela distância entre os dois momentos, no caso 1. Assim somamos 3*1 = 3 na resposta.



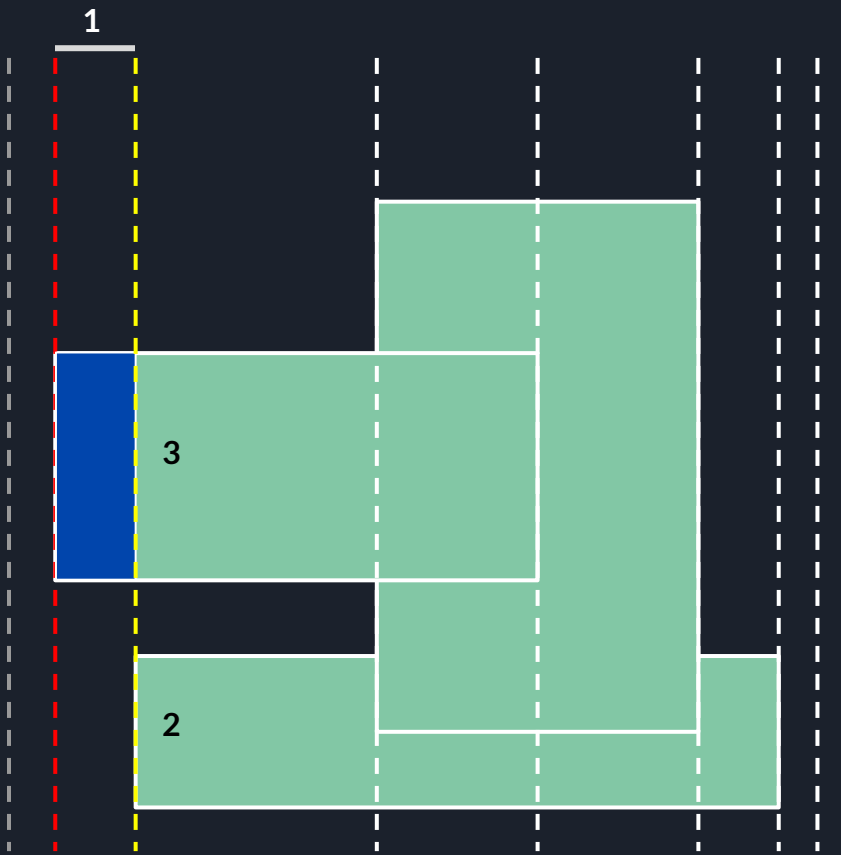
Problema Clássico

Área da União de Retângulos

No próximo momento (entrada do retângulo 2), calculamos a área dos retângulos entre esse momento e o anterior.

Para isso basta multiplicar o tamanho da Line que estava dentro de algum retângulo no momento anterior, no caso 3, pela distância entre os dois momentos, no caso 1. Assim somamos 3*1 = 3 na resposta.

Após isso, processamos a entrada do retângulo 2 na Line, e de alguma forma calculamos (veremos depois) que o tamanho coberto da line é 3 + 2 = 5

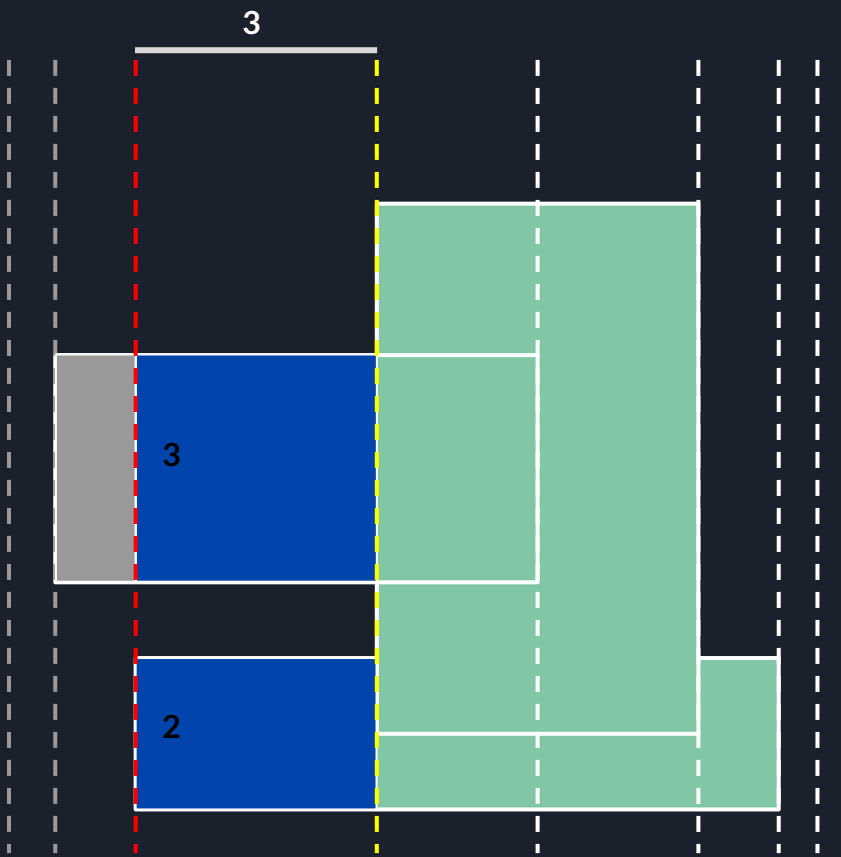


Problema Clássico

Área da União de Retângulos

No próximo momento (entrada do retângulo 3), calculamos a área dos retângulos entre esse momento e o anterior.

Para isso basta multiplicar o tamanho da Line que estava dentro de algum retângulo no momento anterior, no caso 3 + 2 = 5, pela distância entre os dois momentos, no caso 3. Assim somamos 5*3 = 15 na resposta, que passa a ser 18.



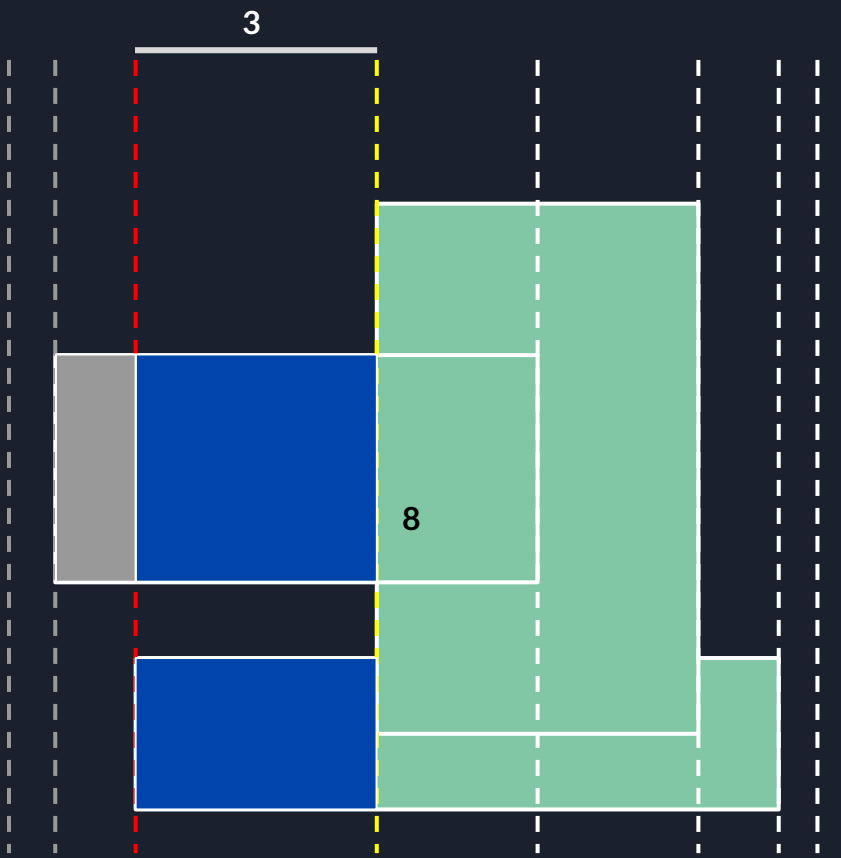
Problema Clássico

Área da União de Retângulos

No próximo momento (entrada do retângulo 3), calculamos a área dos retângulos entre esse momento e o anterior.

Para isso basta multiplicar o tamanho da Line que estava dentro de algum retângulo no momento anterior, no caso 3 + 2 = 5, pela distância entre os dois momentos, no caso 3. Assim somamos 5*3 = 15 na resposta, que passa a ser 18.

Após isso, processamos a entrada do retângulo 3 na Line, e de alguma forma calculamos (veremos depois) que o tamanho coberto da line é 8

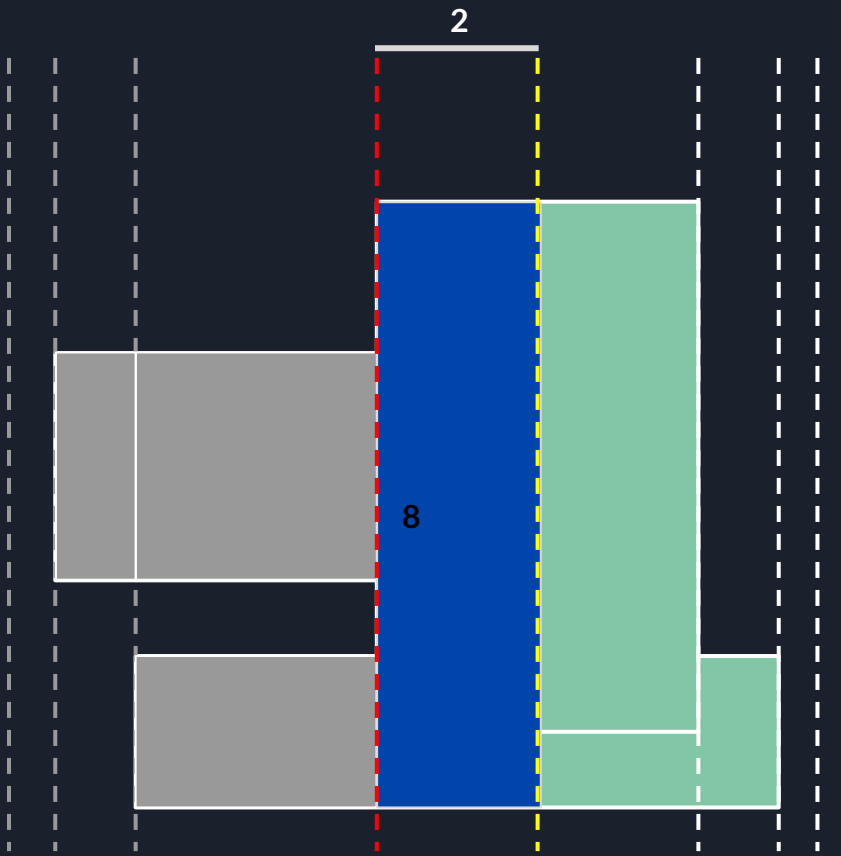


Problema Clássico

Área da União de Retângulos

No próximo momento (saída do retângulo 1), calculamos a área dos retângulos entre esse momento e o anterior.

Para isso basta multiplicar o tamanho da Line que estava dentro de algum retângulo no momento anterior, no caso 8, pela distância entre os dois momentos, no caso 2. Assim somamos 8*2 = 16 na resposta, que passa a ser 34.



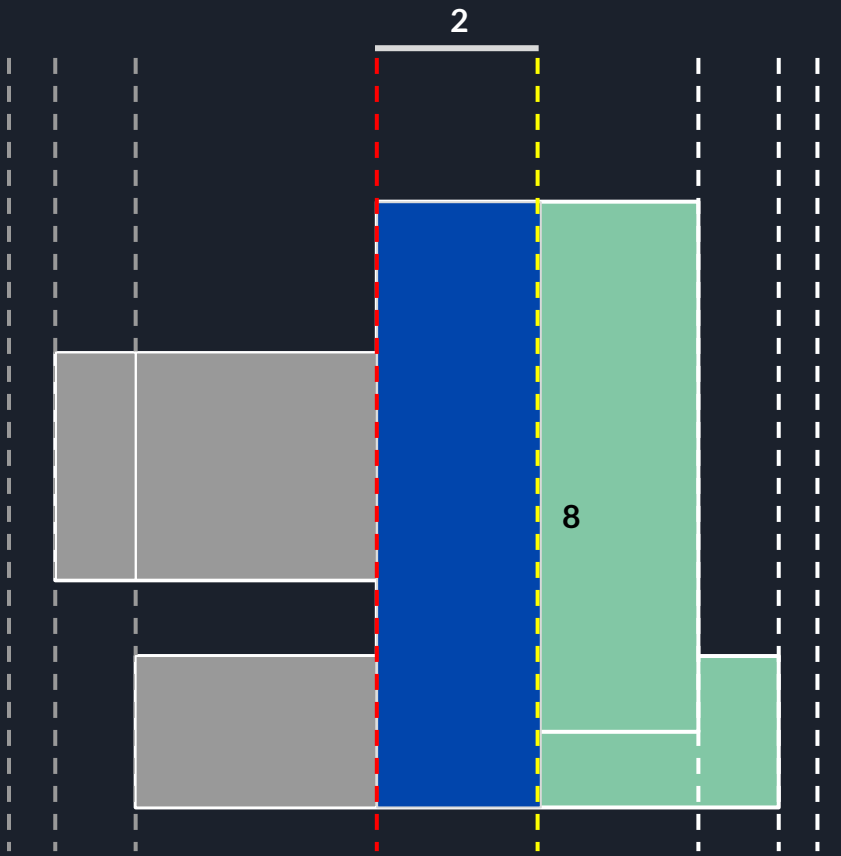
Problema Clássico

Área da União de Retângulos

No próximo momento (saída do retângulo 1), calculamos a área dos retângulos entre esse momento e o anterior.

Para isso basta multiplicar o tamanho da Line que estava dentro de algum retângulo no momento anterior, no caso 8, pela distância entre os dois momentos, no caso 2. Assim somamos 8*2 = 16 na resposta, que passa a ser 34.

Após isso, processamos a saída do retângulo 1 na Line, e de alguma forma calculamos (veremos depois) que o tamanho coberto da line é 8

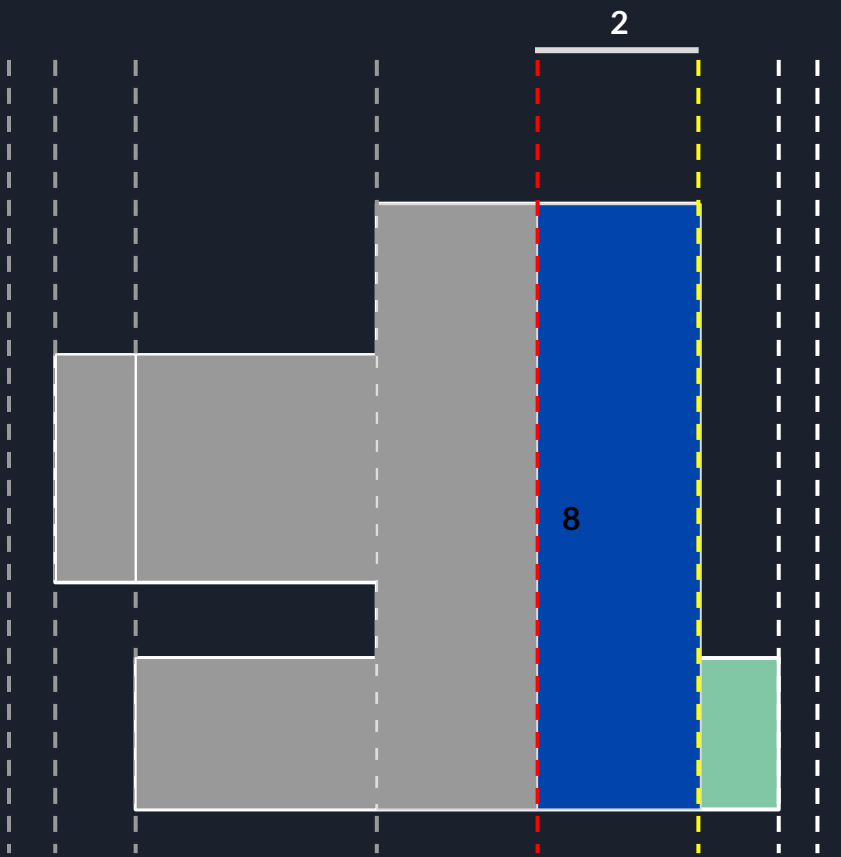


Problema Clássico

Área da União de Retângulos

No próximo momento (saída do retângulo 3), calculamos a área dos retângulos entre esse momento e o anterior.

Para isso basta multiplicar o tamanho da Line que estava dentro de algum retângulo no momento anterior, no caso 8, pela distância entre os dois momentos, no caso 2. Assim somamos 8*2 = 16 na resposta, que passa a ser 50.



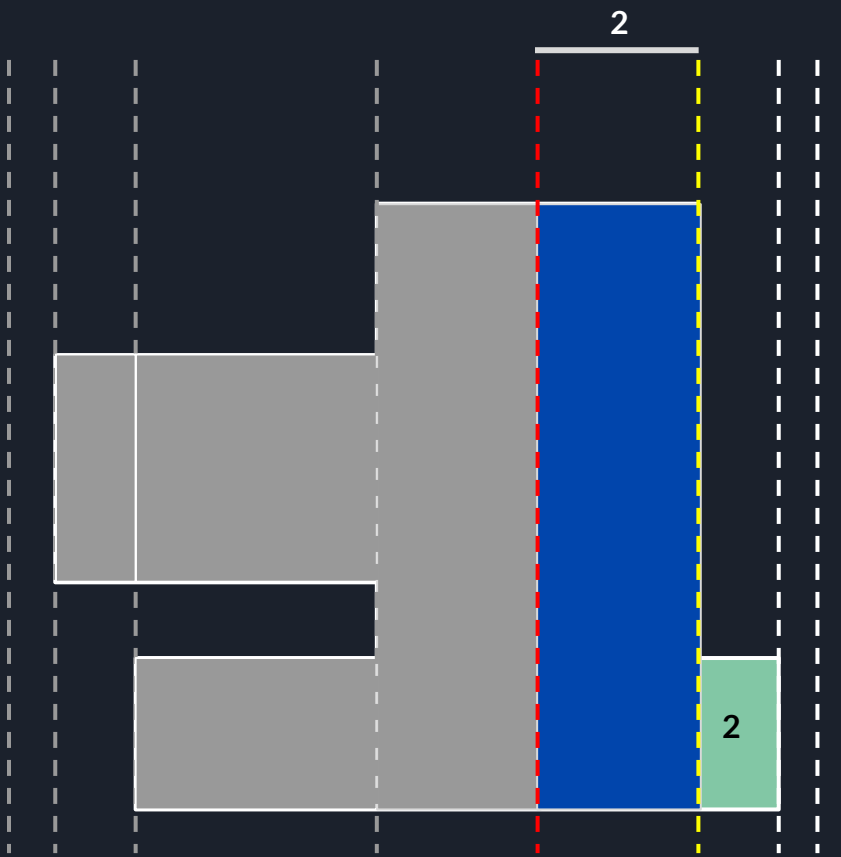
Problema Clássico

Área da União de Retângulos

No próximo momento (saída do retângulo 3), calculamos a área dos retângulos entre esse momento e o anterior.

Para isso basta multiplicar o tamanho da Line que estava dentro de algum retângulo no momento anterior, no caso 8, pela distância entre os dois momentos, no caso 2. Assim somamos 8*2 = 16 na resposta, que passa a ser 50.

Após isso, processamos a saída do retângulo 3 na Line, e de alguma forma calculamos (veremos depois) que o tamanho coberto da line é 2

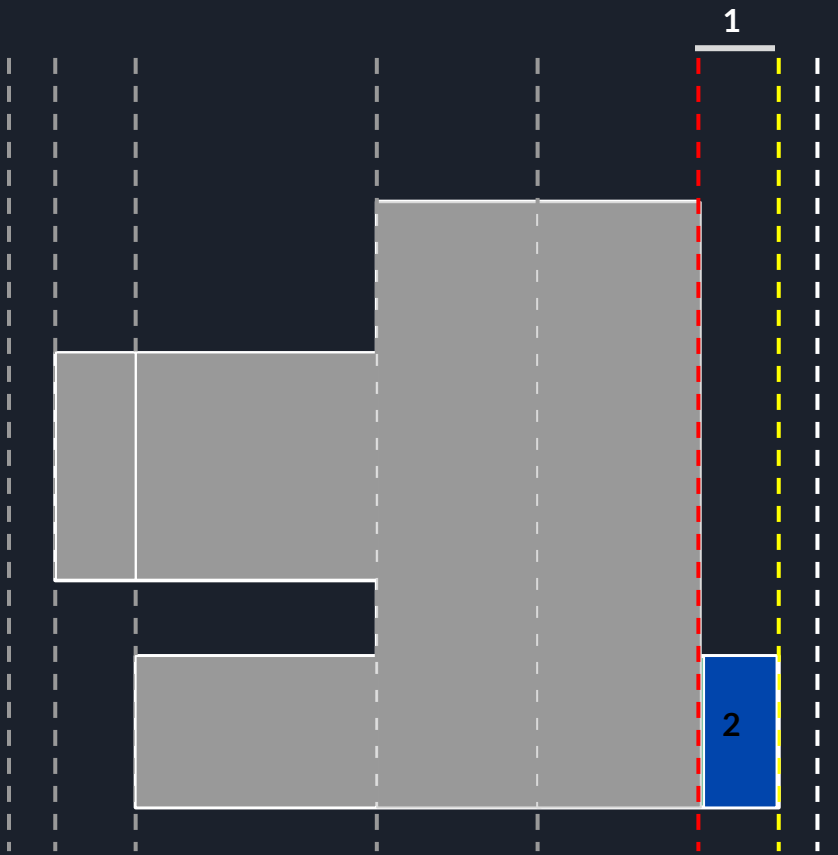


Problema Clássico

Área da União de Retângulos

No próximo momento (saída do retângulo 2), calculamos a área dos retângulos entre esse momento e o anterior.

Para isso basta multiplicar o tamanho da Line que estava dentro de algum retângulo no momento anterior, no caso 2, pela distância entre os dois momentos, no caso 1. Assim somamos 2*1 = 2 na resposta, que passa a ser 52.



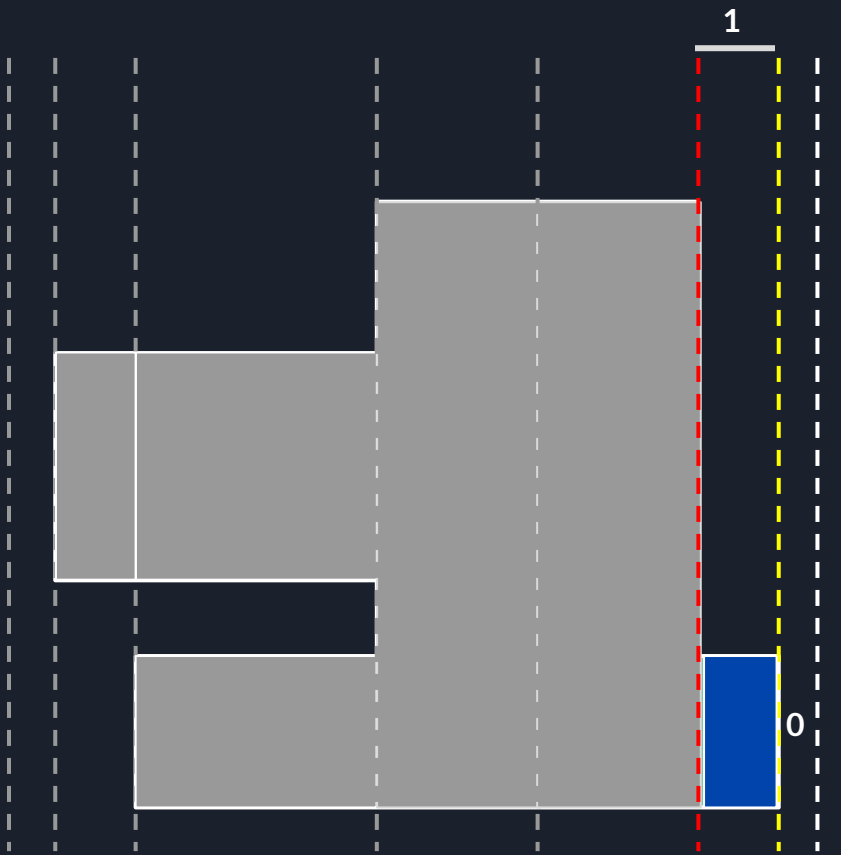
Problema Clássico

Área da União de Retângulos

No próximo momento (saída do retângulo 2), calculamos a área dos retângulos entre esse momento e o anterior.

Para isso basta multiplicar o tamanho da Line que estava dentro de algum retângulo no momento anterior, no caso 2, pela distância entre os dois momentos, no caso 1. Assim somamos 2*1 = 2 na resposta, que passa a ser 52.

Após isso, processamos a saída do retângulo 2 na Line, e de alguma forma calculamos (veremos depois) que o tamanho coberto da line é 0

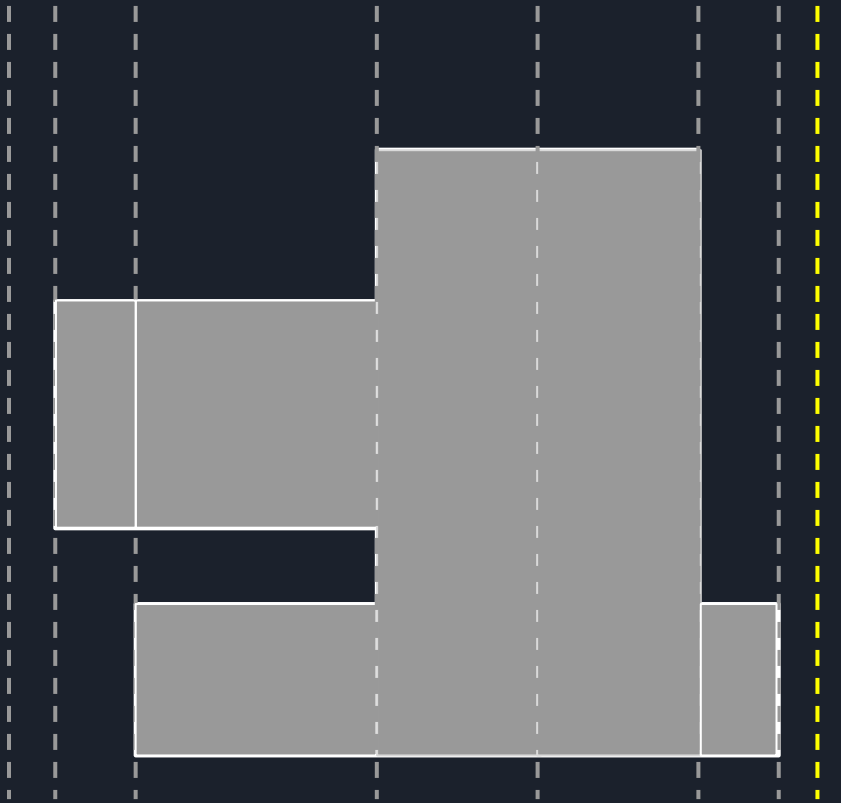


Problema Clássico

Área da União de Retângulos

Desta forma calculamos que a área da união dos retângulos neste exemplo é 52

Mas como manter Line ? Que estrutura devemos usar para conseguir manter o tamanho coberto por algum retângulo enquanto processa os eventos de entrada e saída de retângulos ? PENSE



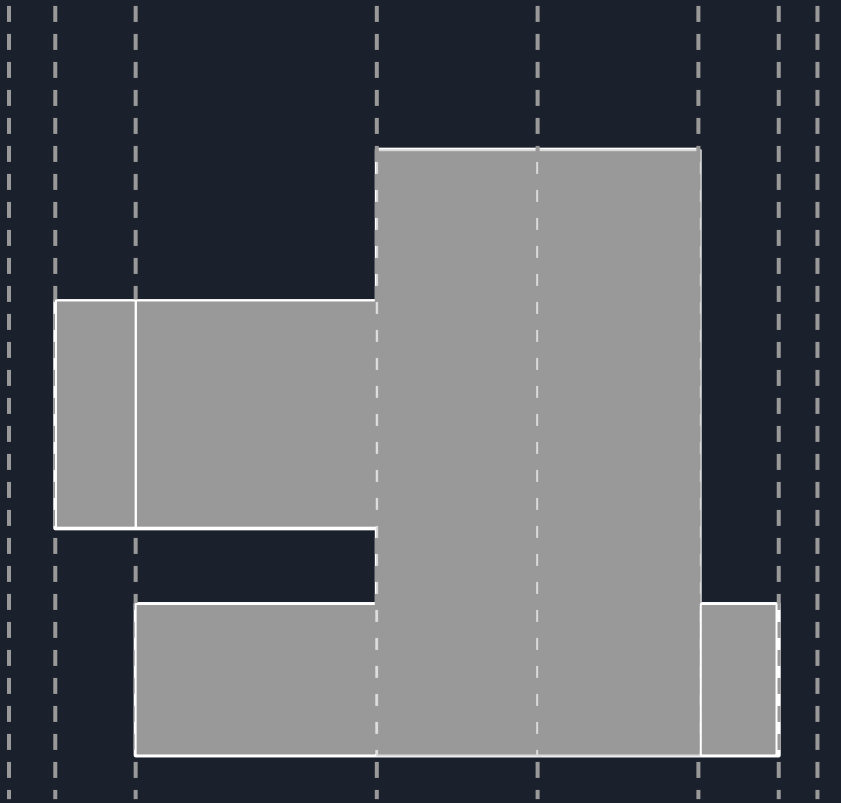
Problema Clássico

Área da União de Retângulos

Ideia: imagine um vetor de marcação em y (vertical), onde para cada coordenada iremos manter quantos retângulos estão cobrindo ela naquele momento.

Quando processamos a entrada de um retângulo, basta somar 1 em seu intervalo de y, ou seja, somar 1 em cada coordenada y que este retângulo passa.

Analogamente, quando processamos a saída de um retângulo, basta subtrair 1 em seu intervalo de y, ou seja, subtrair 1 em cada coordenada y que este retângulo passa.

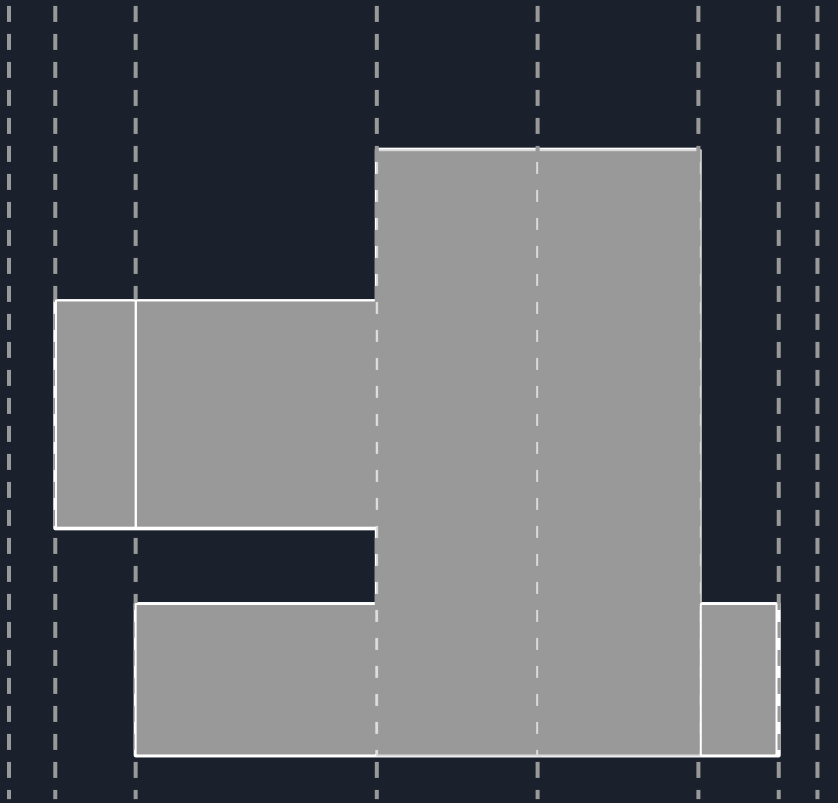


Problema Clássico

Área da União de Retângulos

Ideia: imagine um vetor de marcação em y (vertical), onde para cada coordenada iremos manter quantos retângulos estão cobrindo ela naquele momento.

Desta forma, o tamanho coberto por algum retângulo na Line, será a quantidade de coordenadas que não estão marcadas com 0. Assim se conseguirmos a quantidade que está 0, basta pegar tudo menos esta quantidade.

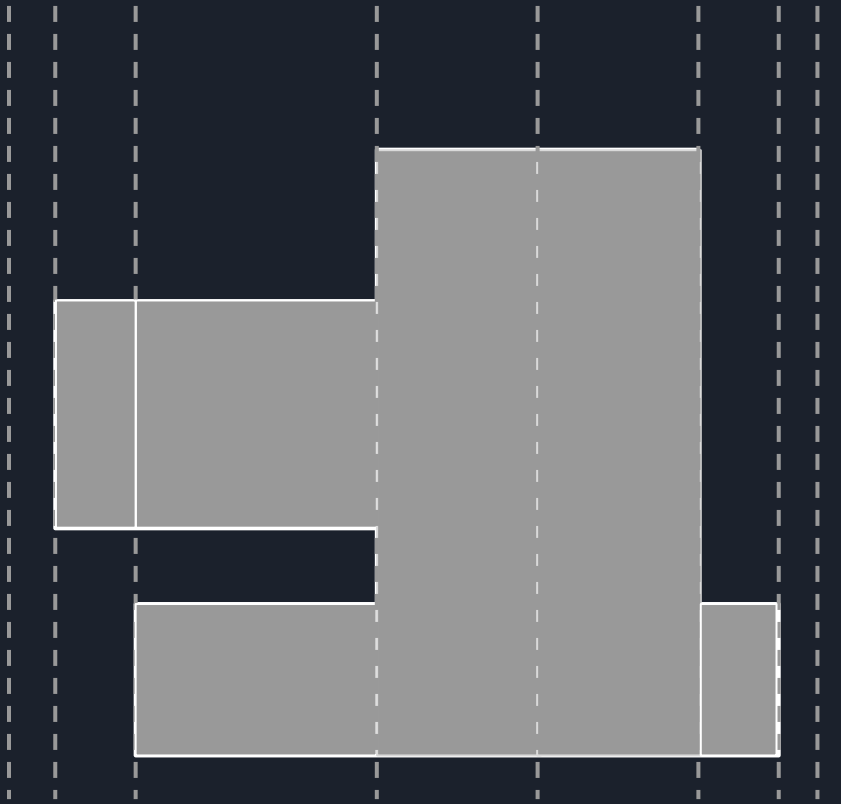


Problema Clássico

Área da União de Retângulos

Ideia: imagine um vetor de marcação em y (vertical), onde para cada coordenada iremos manter quantos retângulos estão cobrindo ela naquele momento.

Como podemos implementar uma estrutura eficiente que faça isso ? PENSE

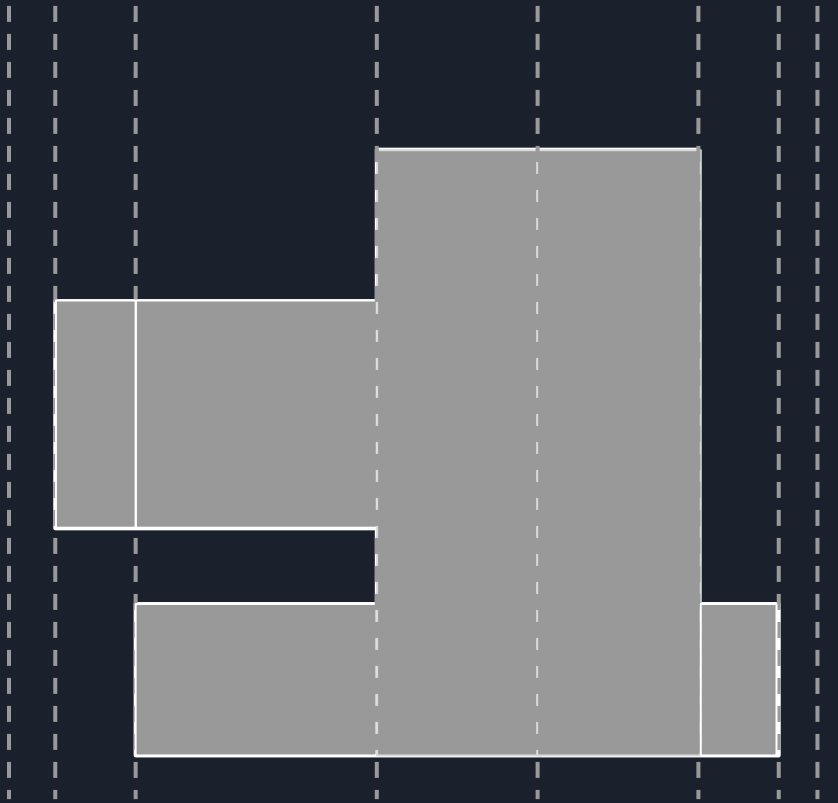


Problema Clássico

Área da União de Retângulos

Solução 1: Fazer uma Seg, onde em cada nó iremos manter a quantidade de zeros naquele nó: qZero, e uma lazy de soma: lzSum.

Pense nos detalhes desta Seg, por exemplo, como atualizar um nó (refresh) ? PENSE



Problema Clássico

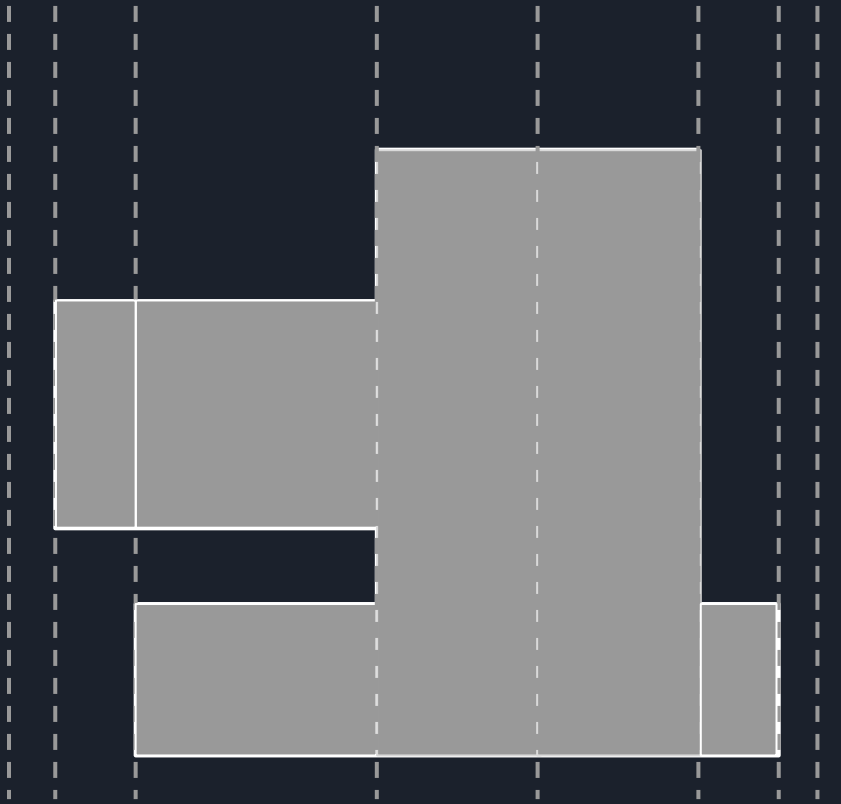
Área da União de Retângulos

Solução 1: Fazer uma Seg, onde em cada nó iremos manter a quantidade de zeros naquele nó: qZero, e uma lazy de soma: lzSum.

Na verdade essa solução NÃO FUNCIONA!

Imagine que vamos atualizar um nó e temos uma lzSum = -1, isso significa que temos que subtrair 1 de todos os elementos. Para calcular o novo qZero, ou seja, quantos números zero teremos, teremos que manter o número de 1's, ou seja, precisamos de um qUm.

Mas pelo mesmo motivo, para manter um qUm iremos precisar da quantidade de 2.



Problema Clássico

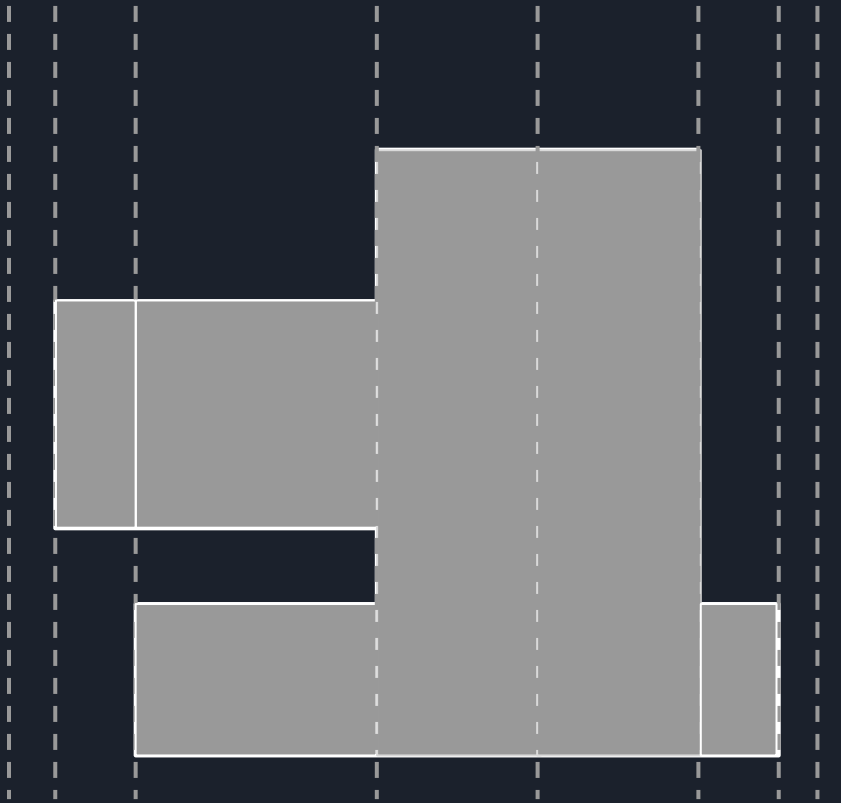
Área da União de Retângulos

Solução 1: Fazer uma Seg, onde em cada nó iremos manter a quantidade de zeros naquele nó: qZero, e uma lazy de soma: lzSum.

Na verdade essa solução NÃO FUNCIONA!

E assim por diante. Ou seja, teríamos que manter as quantidades de todos, e por isso não funciona.

Então como implementar nossa Line ? PENSE

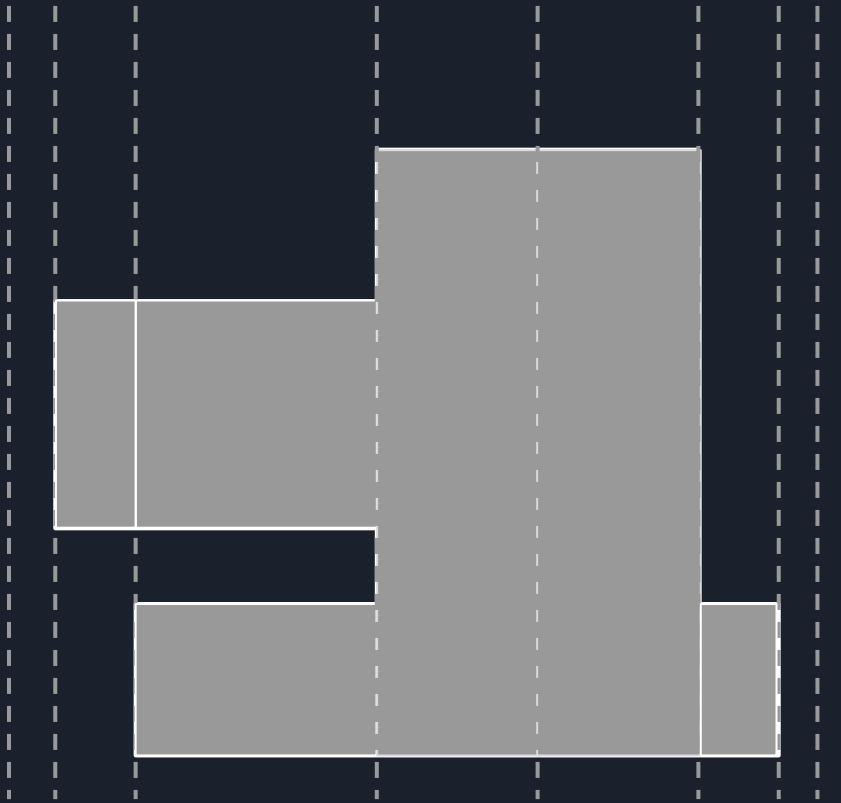


Problema Clássico

Área da União de Retângulos

Solução 2: Fazer uma Seg, onde em cada nó iremos manter o menor valor (o mínimo): min , a quantidade de vezes que esse menor valor aparece (quantidade do mínimo): qtdMin, e uma lazy de soma: lzSum

O refresh fica bem simples, pois como todos os elementos sofrem a alteração (somam o valor em lzSum), então o menor valor também, logo basta somar lzSum no min. Note também que a quantidade do mínimo se mantém



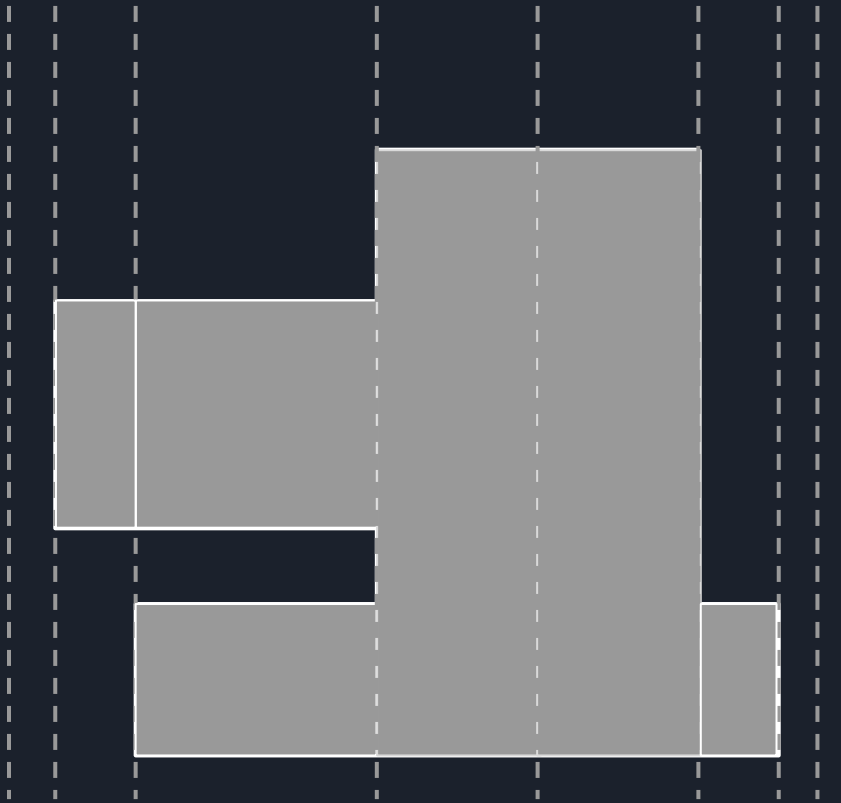
Problema Clássico

Área da União de Retângulos

Solução 2: Fazer uma Seg, onde em cada nó iremos manter o menor valor (o mínimo): min , a quantidade de vezes que esse menor valor aparece (quantidade do mínimo): qtdMin, e uma lazy de soma: lzSum

No Merge:

- Se os mínimos dos filhos são diferentes, basta pegar min e qtdMin do filho com o menor min
- Senão, copia o min dos filhos, e soma qtdMin



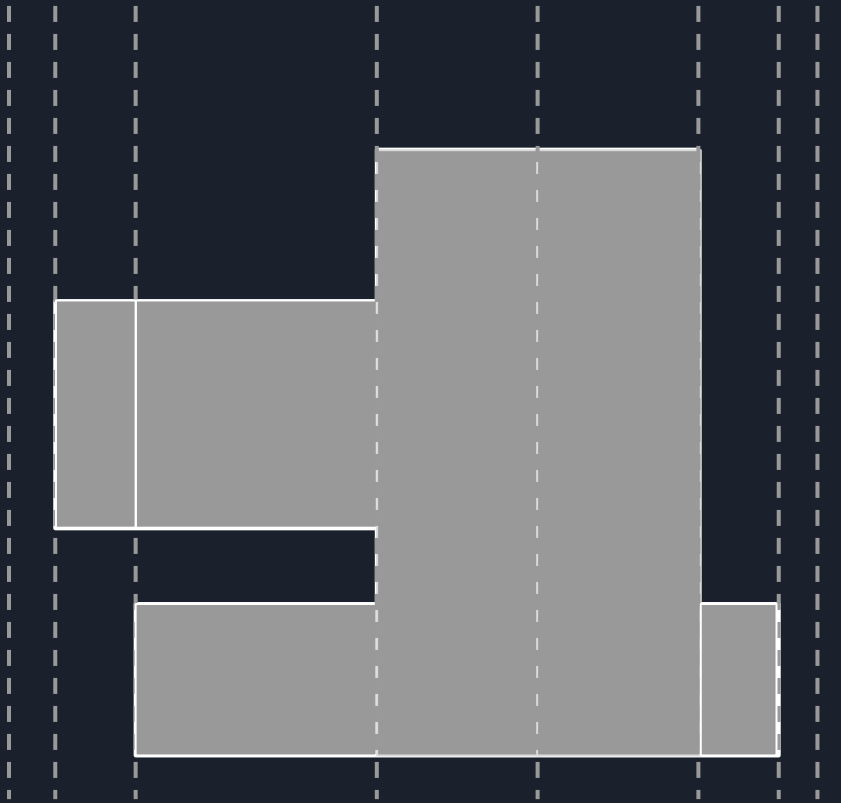
Problema Clássico

Área da União de Retângulos

Solução 2: Fazer uma Seg, onde em cada nó iremos manter o menor valor (o mínimo): min , a quantidade de vezes que esse menor valor aparece (quantidade do mínimo): qtdMin, e uma lazy de soma: lzSum

Para calcular o tamanho coberto por algum retângulo na Line, basta olhar para o min e qtdMin da raiz.

Se o mínimo da raiz for maior que zero, então a Line está inteira coberta. Senão será 0, e portanto basta pegar o total menos a quantidade do mínimo.



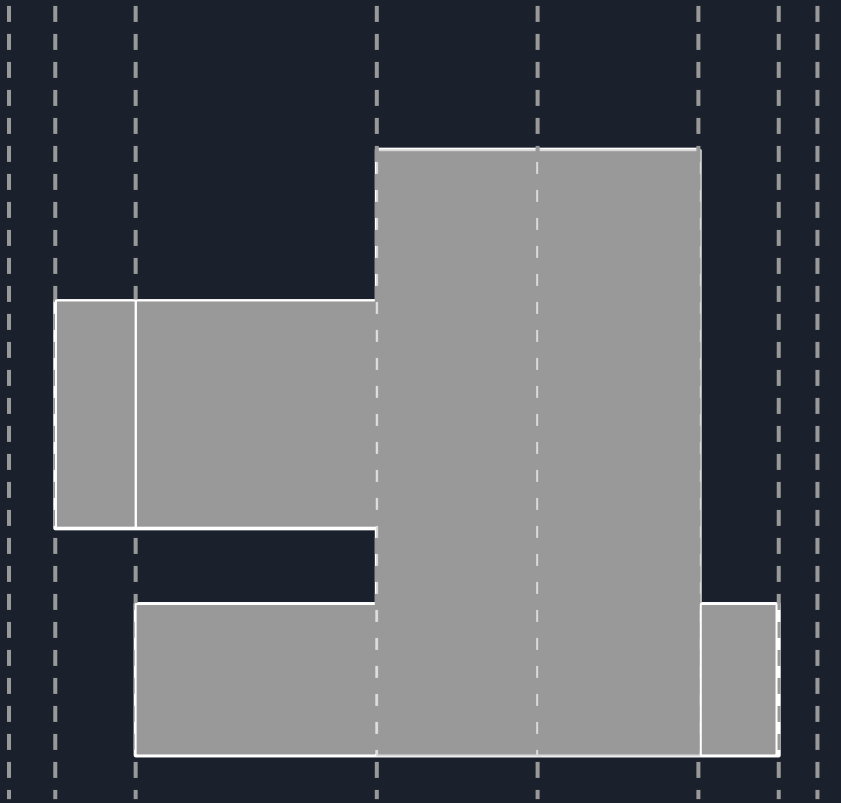
Problema Clássico

Área da União de Retângulos

Solução 2: Fazer uma Seg, onde em cada nó iremos manter o menor valor (o mínimo): min , a quantidade de vezes que esse menor valor aparece (quantidade do mínimo): qtdMin, e uma lazy de soma: lzSum

Perceba que NUNCA os valores ficam negativos, pois sempre a saída de um determinado retângulo é após sua entrada.

No fundo essa Seg é bem simples, mas observar que só precisamos do mínimo e da sua quantidade é bem esperto!



Problema Clássico

Área da União de Retângulos - Lista de Exercícios

- PESCA11
- NKMARS
- Area of Rectangles
- Florestas em Risco

Line Sweep - Lista de Exercícios

- Falha Ao Cercar Legumes (link do Live Archive)
- Rooks and Rectangles
- Trees and Decrees

Merge Sort Tree

Problema KQUERYO - TENTE RESOLVER SOZINHO(A) PRIMEIRO

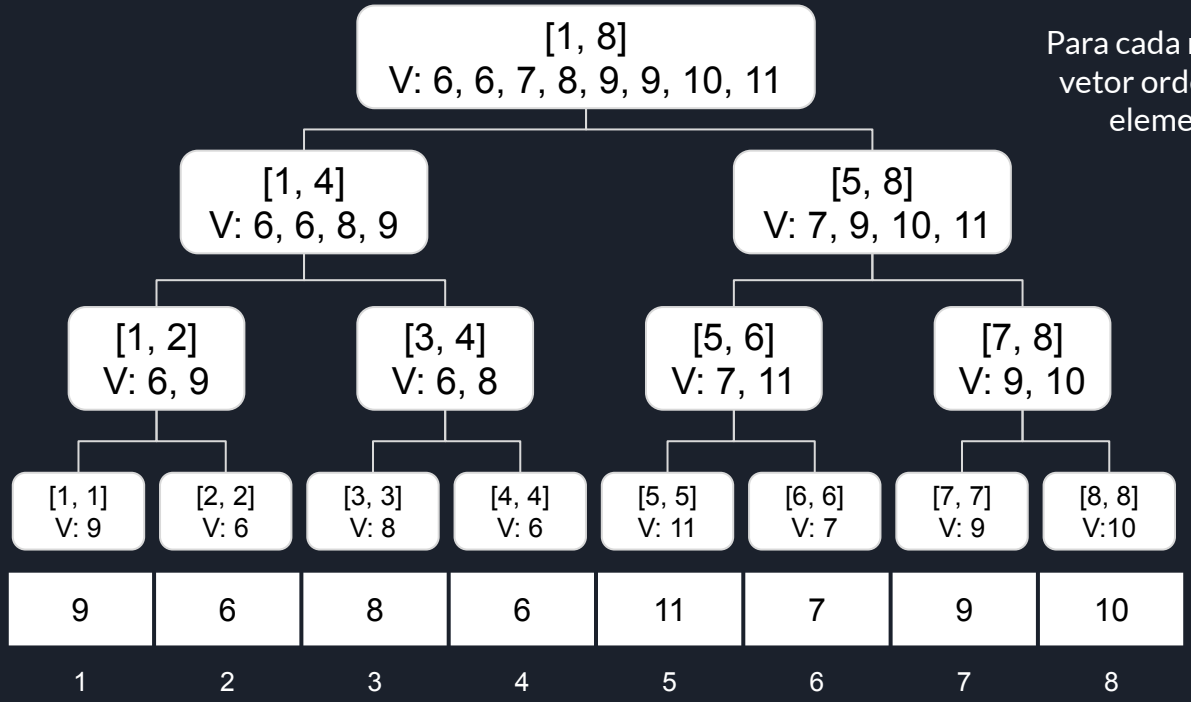
Neste problema temos que encontrar de forma eficiente o número de elementos em um intervalo, que sejam maiores que um valor dado.

Se fosse sempre no vetor inteiro, poderíamos ordenar o vetor e encontrar essa quantidade usando Busca Binária.

Mas como podemos usar essa ideia para um intervalo qualquer ?

Merge Sort Tree

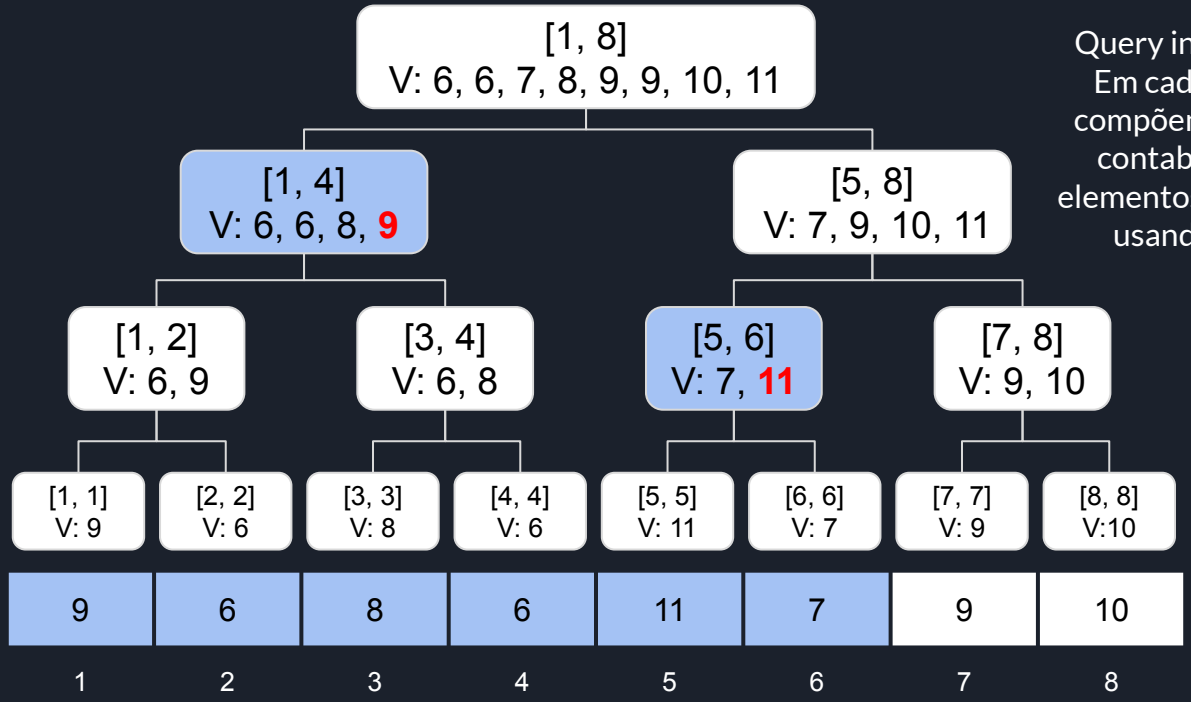
Problema KQUERYO



Para cada nó vamos manter um vetor ordenado com todos os elementos daquele nó

Merge Sort Tree

Problema KQUERYO



Query ini = 1 fim = 6 e k = 8
Em cada um dos nós que compõem o intervalo [1, 6], contabilizamos quantos elementos são maiores que 8, usando Busca Binária

Merge Sort Tree

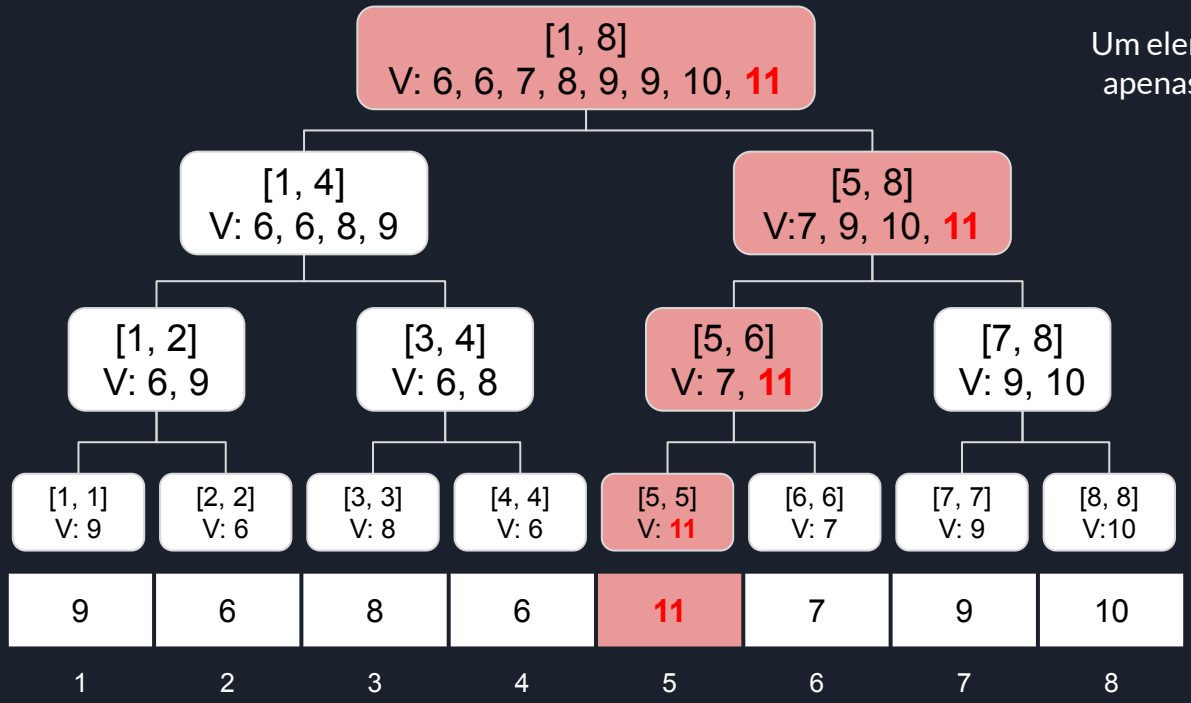
Problema KQUERYO

Detalhes:

- Para construir essa Seg, fazemos uma função build. Se já temos os dois vetores dos nós filhos ordenados, como podemos construir o vetor ordenado do nó atual ?
  - Fazendo o Merge desses dois vetores
  - Por isso, essa estrutura é chamada de Merge Sort Tree
- Mas será que podemos criar uma vetor para cada nó ? Como fica a memória ?
  - Cada elemento só aparece em um único nó por nível, veja o exemplo:

Merge Sort Tree

Problema KQUERYO



Um elemento pertence à apenas um nó por nível

Merge Sort Tree

Problema KQUERYO

Detalhes:

- Para construir essa Seg, fazemos uma função build. Se já temos os dois vetores dos nós filhos ordenados, como podemos construir o vetor ordenado do nó atual ?
  - Fazendo o Merge desses dois vetores
  - Por isso, essa estrutura é chamada de Merge Sort Tree
- Mas será que podemos criar uma vetor para cada nó ? Como fica a memória ?
  - Cada elemento só aparece em um único nó por nível
  - Portanto para cada elemento a complexidade de memória será O(logN). Assim a memória total adicionada apenas para armazenar esses vetores ordenados será O(NlogN)
  - O ideal é declarar apenas a quantidade de memória necessária, por isso implemente usando vector
- Qual a complexidade de cada query ?
  - Um intervalo usa O(logN) nós, e em cada um deles faremos uma Busca Binária, assim teremos complexidade $O(\log^2 N)$
- O legal desse problema é que podemos armazenar outras estruturas dentro de cada nó da Seg, como um vector, set, ou até mesmo outra Seg (Seg2D).

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ):
Dado um vetor V, inicialmente zerado, 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].
- A diferença é que agora os índices do vetor vão de 1 à 10^9, ou seja, o N vai até 10^9, mas Q ainda vai até 10^5

Solução inicial: Compressão de coordenadas

- Podemos inicialmente determinar TODOS os índices que aparecem em qualquer operação
- Note que os valores desses índices não são relevantes, apenas a ordem relativa entre eles, portanto podemos comprimir essas coordenadas, ou seja, a menor delas podemos trocar por 1, a segunda menor por 2, e assim por diante.
- Como em cada operação aparecem no máximo 2 índices novos, o número de índices distintos será no máximo $2*10^5$ e assim podemos fazer uma Seg de mínimo para resolver o problema

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ):
Dado um vetor V, inicialmente zerado, 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].
- A diferença é que agora os índices do vetor vão de 1 à 10^9, ou seja, o N vai até 10^9, mas Q ainda vai até 10^5

Solução inicial: Compressão de coordenadas

- Acontece que às vezes os juízes forçam que a solução seja ONLINE
  - Uma forma de fazer isso é fazer os valores de uma operação depender da resposta da última query realizada (ficará mais claro quando ver os exercícios)
- Nesses casos a compressão de coordenadas não vai funcionar pois ela é OFFLINE
- Como resolver então ?

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

[1, 8]
Min: 0

Ideia Principal:
Criar nós apenas quando necessário

Vejamos um exemplo
No início temos apenas a raiz

Lembrando que tudo começa zerado

Segment Tree Dinâmica

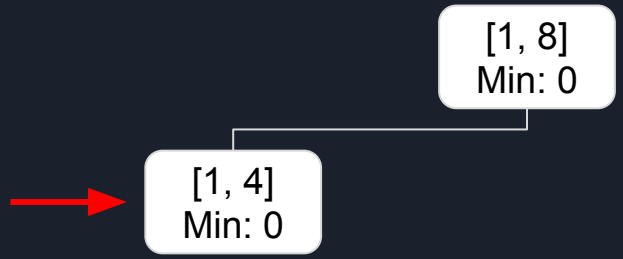
Problema base - Range Minimum Query (RMQ)

Ideia Principal:
Criar nós apenas quando necessário

Update i = 2 val = -2
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

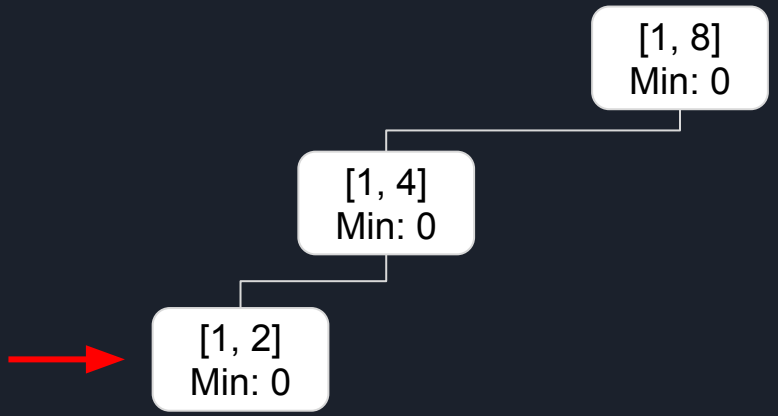


Ideia Principal:
Criar nós apenas quando necessário

Update i = 2 val = -2
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

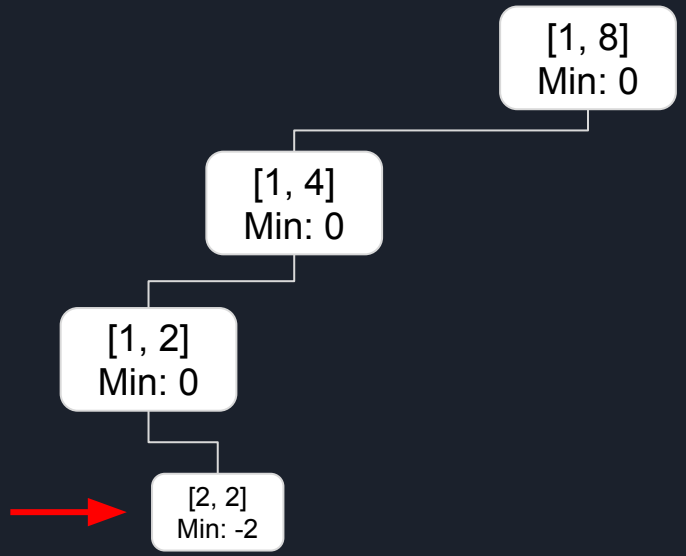


Ideia Principal:
Criar nós apenas quando necessário

Update i = 2 val = -2
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

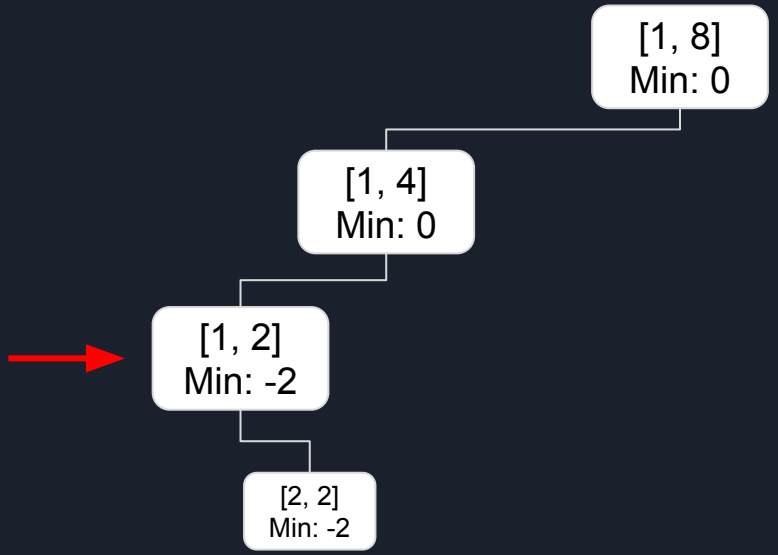


Ideia Principal:
Criar nós apenas quando necessário

Update i = 2 val = -2
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

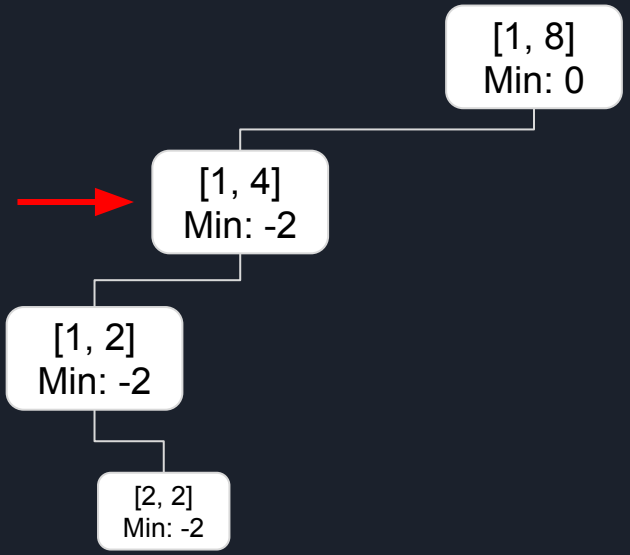


Ideia Principal:
Criar nós apenas quando necessário

Update i = 2 val = -2
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

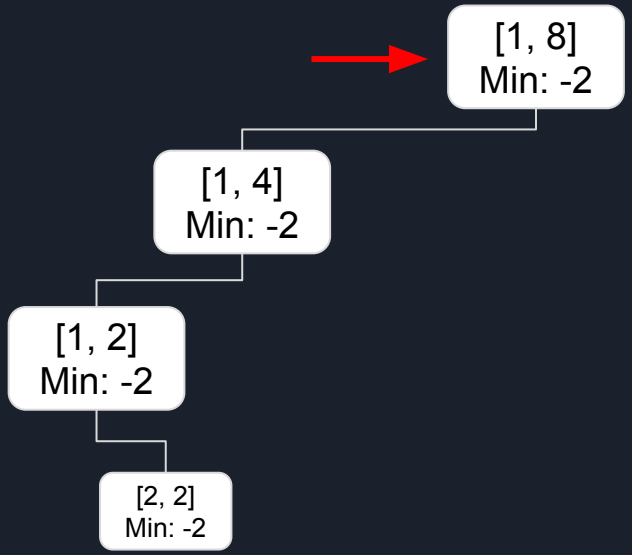


Ideia Principal:
Criar nós apenas quando necessário

Update i = 2 val = -2
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

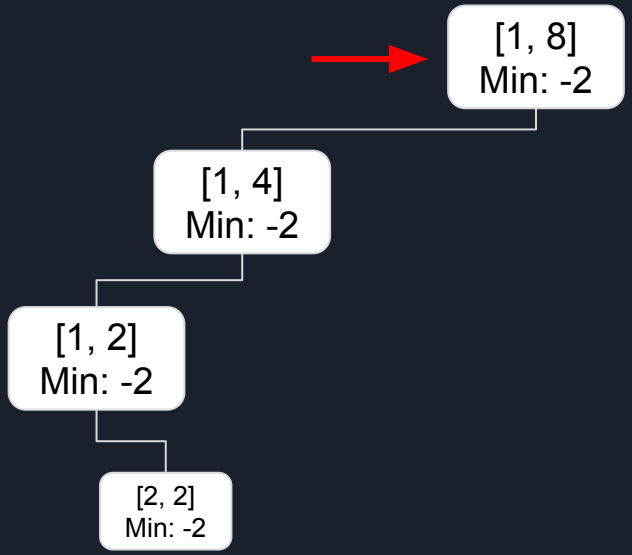


Ideia Principal:
Criar nós apenas quando necessário

Update i = 2 val = -2
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

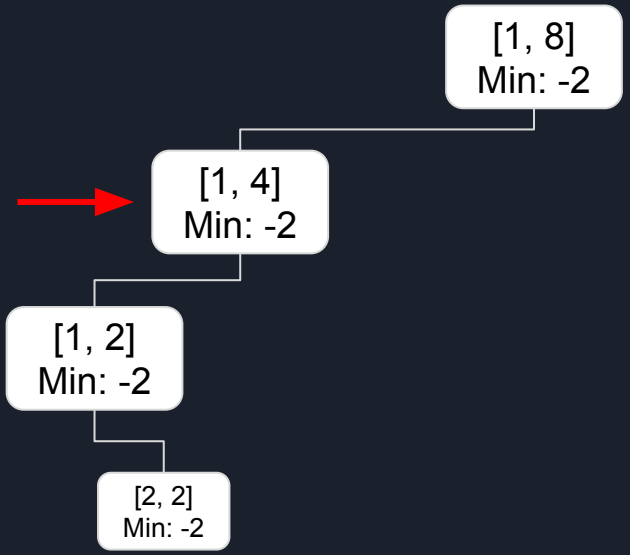


Ideia Principal:
Criar nós apenas quando necessário

Update i = 3 val = -3
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

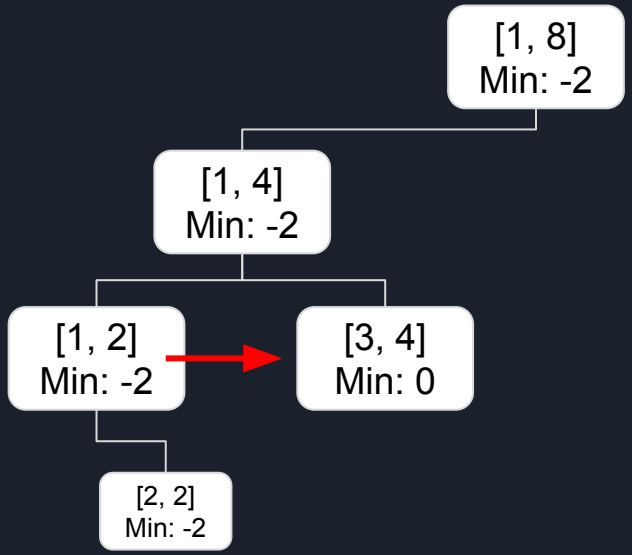


Ideia Principal:
Criar nós apenas quando necessário

Update i = 3 val = -3
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

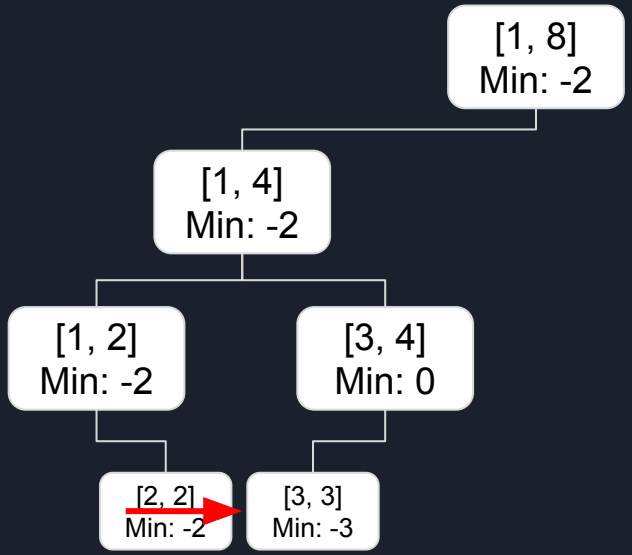


Ideia Principal:
Criar nós apenas quando necessário

Update i = 3 val = -3
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

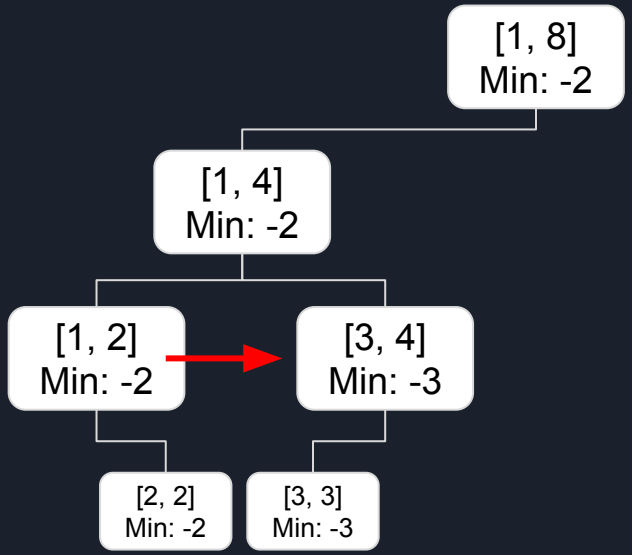


Ideia Principal:
Criar nós apenas quando necessário

Update i = 3 val = -3
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

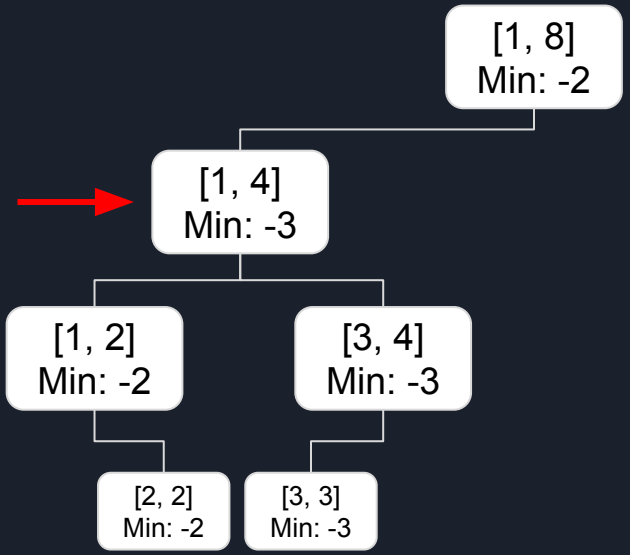


Ideia Principal:
Criar nós apenas quando necessário

Update i = 3 val = -3
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

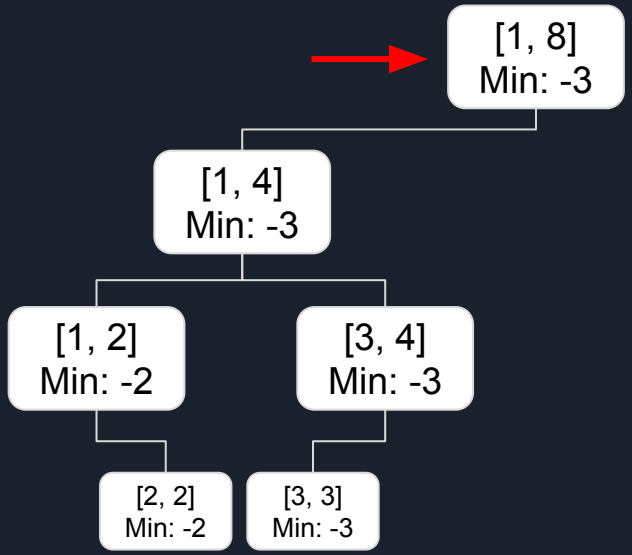


Ideia Principal:
Criar nós apenas quando necessário

Update i = 3 val = -3
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)



Ideia Principal:
Criar nós apenas quando necessário

Update i = 3 val = -3
Mesma lógica do update normal, mas criando o nó apenas quando necessário

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

Detalhes:

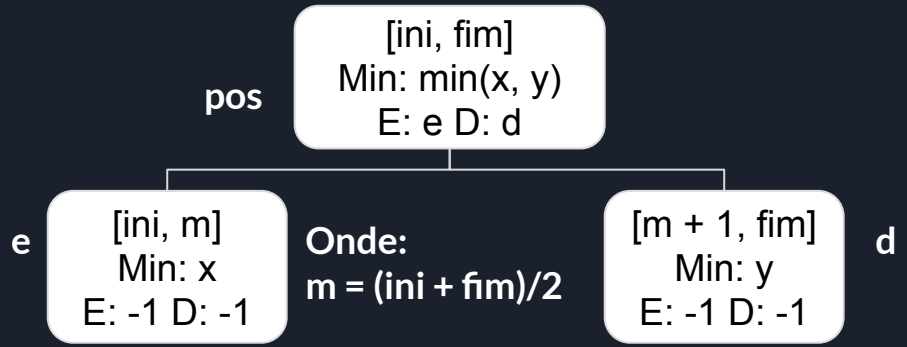
- No exemplo a Seg possui intervalo total de 1 a 8, mas podemos usar qualquer intervalo, por exemplo de 1 a 10^9
- Cada update vai criar no máximo um nó por nível, portanto irá criar no máximo O(logTAM) onde TAM é o tamanho do intervalo total das coordenadas. Assim ao final teremos O(Q*logTAM) nós, por exemplo para coordenadas de 1 à 10^9 e 10^5 operações, teremos no máximo $3*10^6$ nós

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

Detalhes:

- Os filhos esquerdo e direito do nó pos não serão mais 2*pos e 2*pos + 1, teremos que armazenar no próprio nó



- Cuidado com m = (ini + fim)/2 se o intervalo das coordenadas contiver números negativos

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

Detalhes:

- Cuidado com m = (ini + fim)/2 se o intervalo das coordenadas contiver números negativos, por exemplo $[-10^9, 10^9]$
  - Por exemplo ini = -2 e fim = -1, então teremos m = -3/2 = -1
  - O filho esquerdo terá intervalo [ini, m] = [-2, -1] que é igual ao intervalo do nó pai, LOOP INFINITO!
  - Para arrumar podemos fazer: if(ini == fim - 1) m = ini;
  - Outra forma de arrumar é usar m = (ini + fim) >> 1 ao invés de /2
- Para implementar, podemos criar um vector de nós (como um banco de nós, com todos os nós)
  - Uma opção é vector<node> seg; e armazenar todas as informações no nó, por exemplo, o mínimo, o índice do filho esquerdo e o índice do filho direito
  - Outra opção, que gosto mais, é criar um vector para cada informação. Os vectors são correspondentes, ou seja, o primeiro valor de todos é relativo ao nó zero, e assim por diante.

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

```
int create() {
  min.push_back(0);
  e.push_back(-1);
  d.push_back(-1);
  return min.size() - 1;
}
```

A função create cria um nó novo, lembrando que todo valor inicialmente era zero, e retorna o índice do nó criado.

Note que um nó novo terá filho esquerdo e filho direito ainda não criado, por isso coloquei como -1

Vale ressaltar que no início é necessário criar o primeiro nó da Seg, a raiz, e dessa forma ela será o nó 0

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

```
void update(int pos, int ini, int fim, int id, int val) {
  if(id < ini || id > fim) return;

  if(ini == fim) {
    min[pos] = val;
    return;
  }

  int m = (ini + fim) >> 1;
  if(id <= m) {
    if(e[pos] == -1) e[pos] = create();
    update(e[pos], ini, m, id, val);
  } else {
    if(d[pos] == -1) d[pos] = create();
    update(d[pos], m + 1, fim, id, val);
  }

  int minE = (e[pos] == -1)? 0 : min[e[pos]];
  int minD = (d[pos] == -1)? 0 : min[d[pos]];
  min[pos] = min(minE, minD);
}
```

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

CUIDADO:

```
e[pos] = create();
d[pos] = create();
```

Caso você use C++11 ou anterior, esses comandos podem dar problema

Isso acontece devido ao resize do vector que acontece na função create.

Para consertar faça:

```
int aux = create();
e[pos] = aux;
```

```
void update(int pos, int ini, int fim, int id, int val) {
  if(id < ini || id > fim) return;

  if(ini == fim) {
    min[pos] = val;
    return;
  }

  int m = (ini + fim) >> 1;
  if(id <= m) {
    if(e[pos] == -1) e[pos] = create();
    update(e[pos], ini, m, id, val);
  } else {
    if(d[pos] == -1) d[pos] = create();
    update(d[pos], m + 1, fim, id, val);
  }

  int minE = (e[pos] == -1)? 0 : min[e[pos]];
  int minD = (d[pos] == -1)? 0 : min[d[pos]];
  min[pos] = min(minE, minD);
}
```

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

```
int create() {
  min.push_back(0);
  e.push_back(0);
  d.push_back(0);
  return min.size() - 1;
}
```

Outra convenção, que pode ser bem útil é: criar um nó nulo, um nó que representa o que ainda não foi criado

Dessa forma devemos inicialmente criar dois nós, um sendo o nulo (no índice 0) e outro sendo a raiz, no índice 1.

```
void update(int pos, int ini, int fim, int id, int val) {
  if(id < ini || id > fim) return;

  if(ini == fim) {
    min[pos] = val;
    return;
  }

  int m = (ini + fim) >> 1;
  if(id <= m) {
    if(e[pos] == 0) e[pos] = create();
    update(e[pos], ini, m, id, val);
  } else {
    if(d[pos] == 0) d[pos] = create();
    update(d[pos], m + 1, fim, id, val);
  }

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

Segment Tree Dinâmica

Problema base - Range Minimum Query (RMQ)

```
int query(int pos, int ini, int fim, int p, int q) {
  if(q < ini || p > fim) return INT_MAX;

  if(pos == 0) return 0;

  if(p <= ini && fim <= q) return min[pos];

  int m = (ini + fim) >> 1;
  return min(query(e[pos], ini, m, p, q), query(d[pos], m + 1, fim, p, q));
}
```

Segment Tree Dinâmica

Lista de Exercícios

- **ORDERSET**
- **Range Sum Query**
- **Pillars**

A Seg Dinâmica também pode ser chamada de Esparsa, ou ainda Implícita. Mas é mais comum usar Dinâmica mesmo

Segment Tree Dinâmica

Lazy Propagation

Detalhes:

- No refresh, quando for propagar a Lazy aos filhos, se eles não existirem, você deve criar
- Caso chame o refresh para um nó que ainda não foi criado (-1 ou 0 dependendo da convenção usada) não faça nada, pois se ele não foi criado então ele não possui nenhuma Lazy
- No Merge do Update, tome cuidado para garantir que os dois filhos estejam atualizados. A forma mais simples de fazer isso é chamar o refresh nos filhos logo antes de fazer o Merge

Exercício: **BGSHOOT**

Segment Tree Dinâmica

Múltiplas Segs

Usando Seg Dinâmica, podemos manter ao mesmo tempo múltiplas Segs, por exemplo 10^5 Segs diferentes, e cada uma com um intervalo de coordenadas gigante, por exemplo de -10^9 a 10^9. Desde que o número total de operações nessas Segs seja limitado, por exemplo 10^5 operações no total.

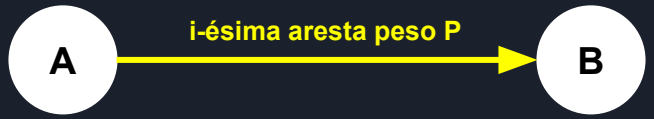
Na verdade isso é muito simples de ser feito, e muito poderoso. Basta no início, ao invés de criar apenas a raiz de uma única seg, criar a raiz de todas as segs que desejar (se trabalhar com o nó nulo, deve criá-lo antes de tudo, para que seja o nó zero).

Perceba que TODAS as segs irão usar o mesmo banco de nós (vector com os nós), e isso não tem problema, pois em cada nó estamos guardando o índice do seu filho esquerdo e direito.

Ao processar uma operação, identifique a Seg correspondente e chame a operação no nó que é raiz dessa Seg

Segment Tree Dinâmica

Múltiplas Segs - Problema Pathwalks - TENTE RESOLVER SOZINHO(A) PRIMEIRO



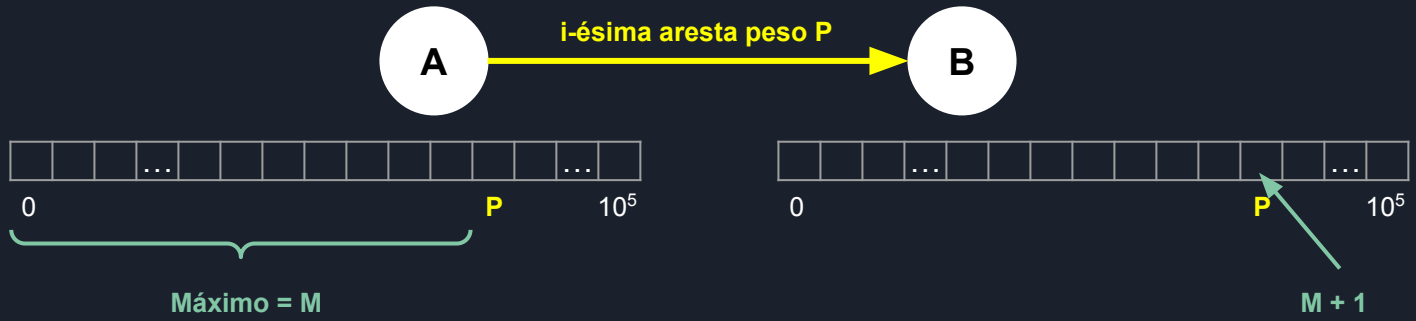
Iremos calcular o maior caminho especial (que obedece às restrições do problema) que termina na i-ésima aresta

Note que uma das restrições é que as arestas usadas devem estar na mesma ordem da entrada (ordem relativa), por isso para calcular o maior caminho especial que termina na i-ésima aresta, só precisamos ter processado as arestas de 1 à i - 1 (que vieram antes a entrada)

Perceba que o maior caminho especial que termina na i-ésima aresta, será igual ao tamanho do maior caminho especial que termina no vértice A, só use arestas de índice menor que i (isso será natural pois as outras ainda não terão sido processadas) e o peso da última aresta do caminho deve ser menor que P. Se soubermos este tamanho, basta somar 1 devido a própria i-ésima aresta

Segment Tree Dinâmica

Múltiplas Segs - Problema Pathwalks - TENTE RESOLVER SOZINHO(A) PRIMEIRO

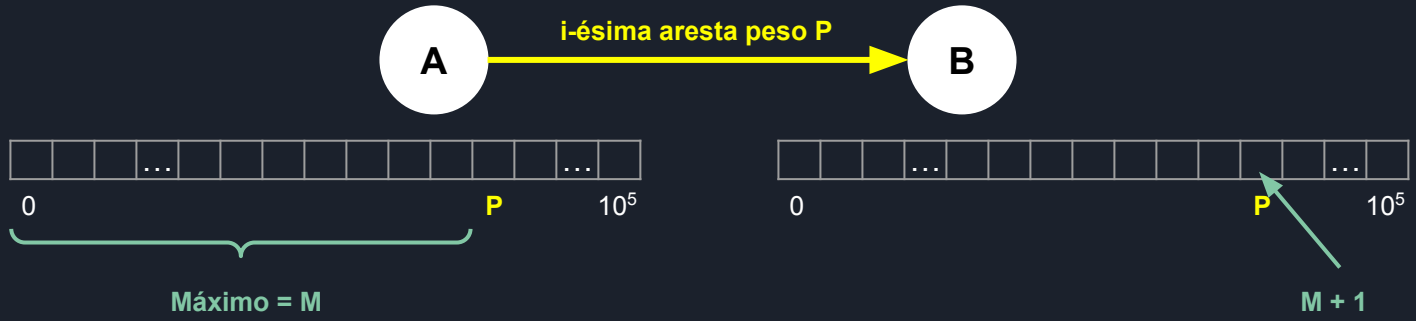


Por isso vamos manter para cada vértice um vetor de marcação sobre os pesos (índices vão de 0 à 10^5), e armazenar no índice w, qual é o maior tamanho de um caminho especial que acaba naquele vértice com peso w.

Para incluir a i-ésima aresta, basta calcular o valor máximo de 0 à P - 1 no vetor de marcação do vértice A, vamos chamar de M, e colocar M + 1 no índice P do vetor de marcação do vértice B (na verdade, ao invés de colocar M + 1, precisa deixar o máximo entre o que já estava e M + 1)

Segment Tree Dinâmica

Múltiplas Segs - Problema Pathwalks - TENTE RESOLVER SOZINHO(A) PRIMEIRO



Para manter um vetor de marcação para cada vértice, e conseguir calcular o valor máximo de forma eficiente, podemos usar Seg Dinâmica, com o truque de manter múltiplas Segs

Segment Tree Persistente

Imagine que temos uma Seg que armazena a soma em cada nó, e realiza Query de soma em intervalo e Update em um único índice

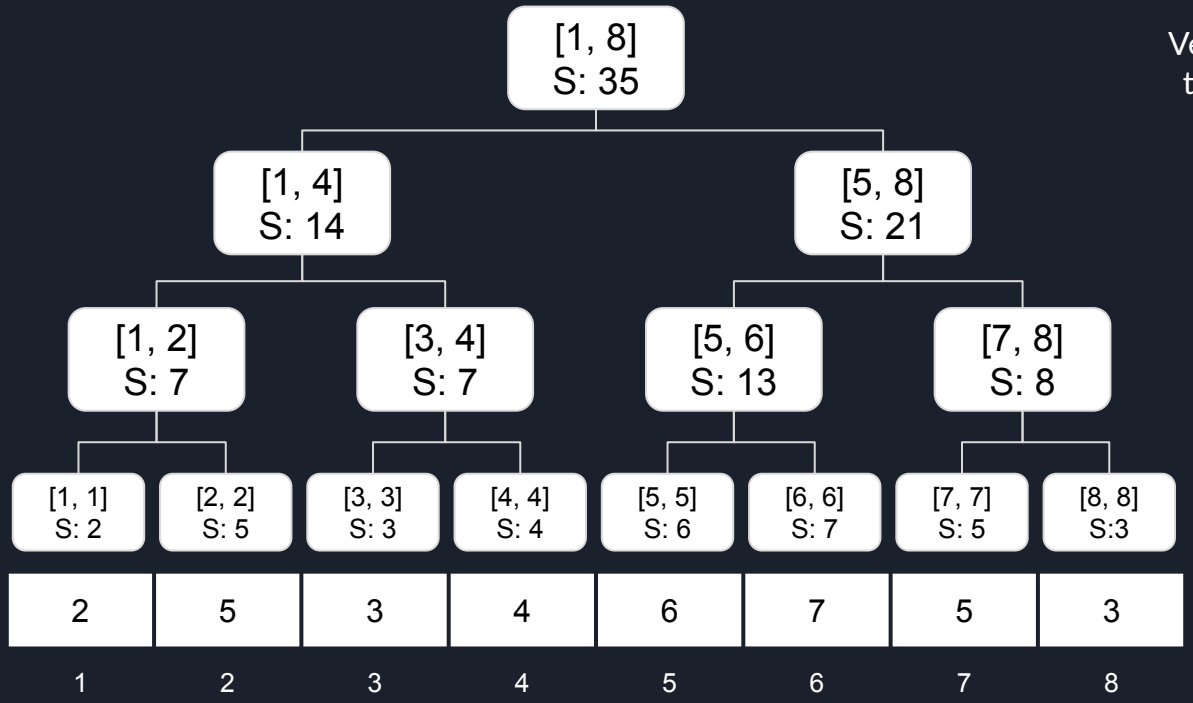
Vamos analisar as versões desta Seg ao longo do tempo, ou seja, as versões dela conforme realizamos alguns Updates

Queremos de alguma forma manter TODAS essas versões dessa Seg, é daí que vem o nome Persistente, vamos ver a ideia primeiro e depois como fica o código

Vejamos um exemplo:

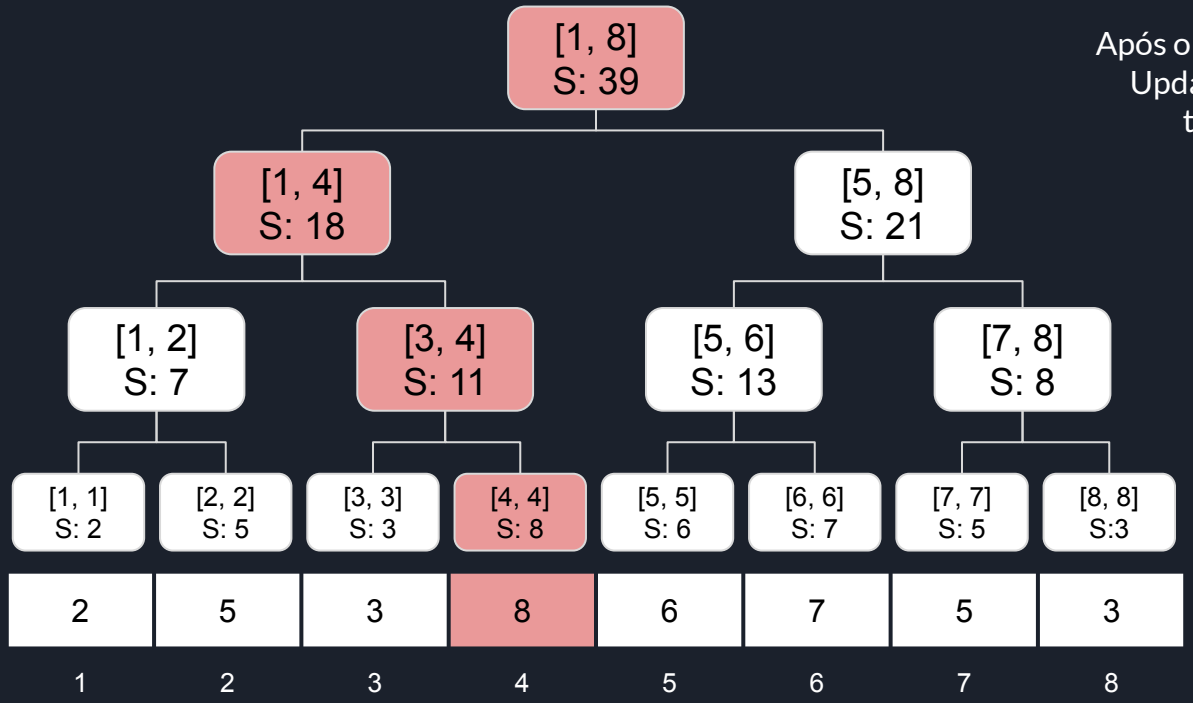
*Este exemplo foi retirado de **Persistent Segment Tree - Oliver-Matis Lill**

Segment Tree Persistente



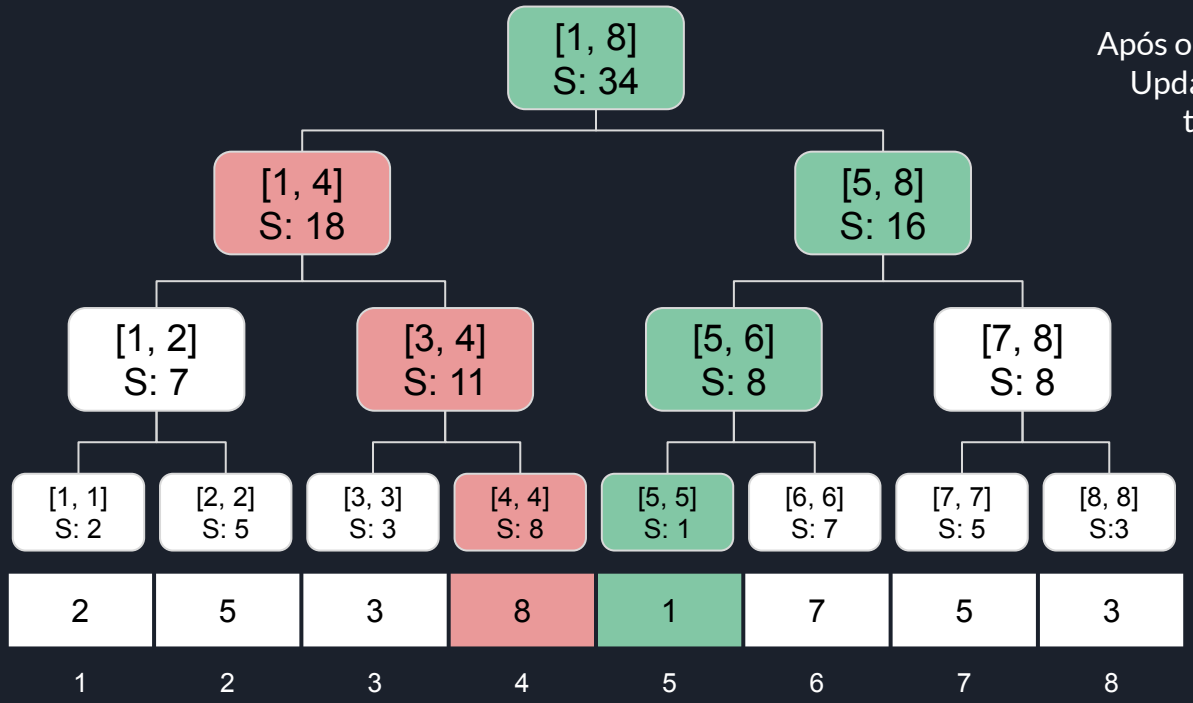
Versão Inicial
tempo t = 0

Segment Tree Persistente



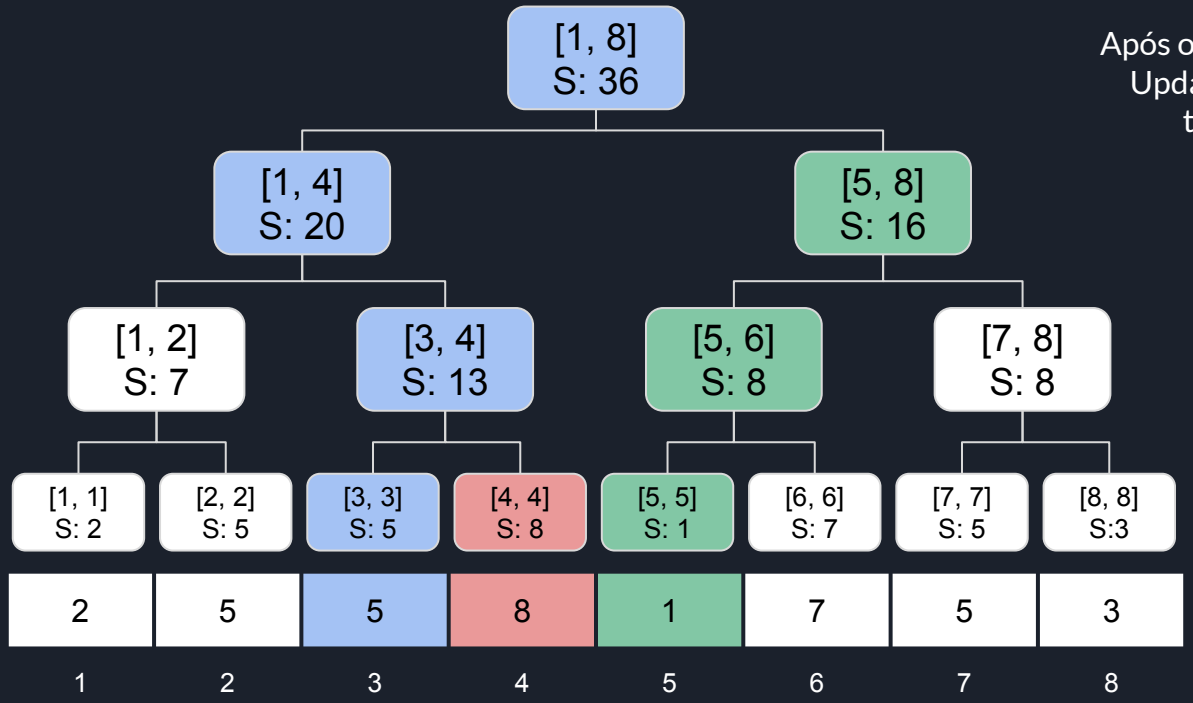
Após o primeiro Update
Update i = 4 val = 8
tempo t = 1

Segment Tree Persistente



Após o segundo Update
Update i = 5 val = 1
tempo t = 2

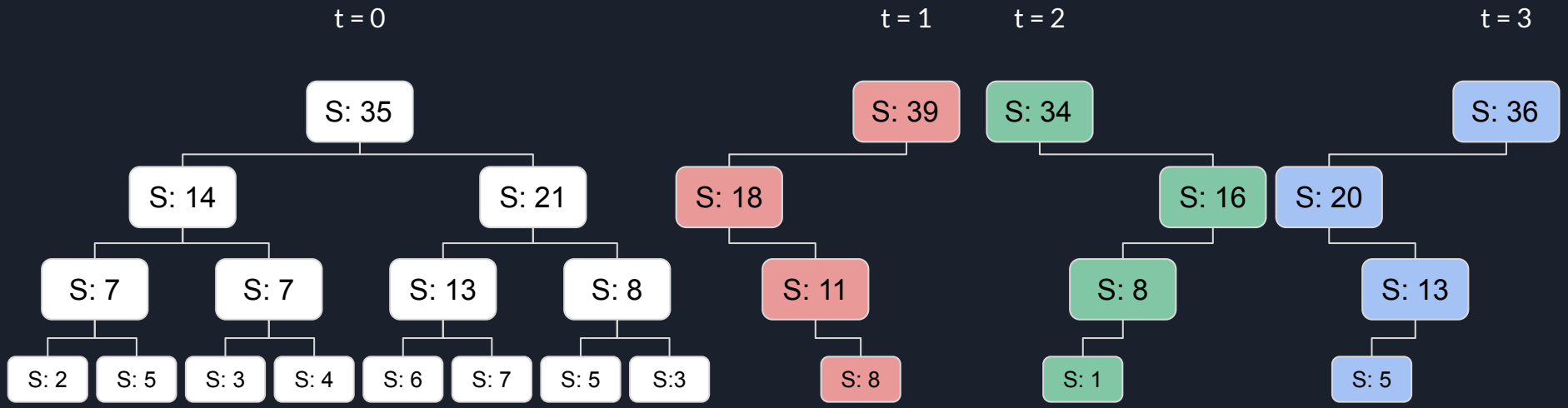
Segment Tree Persistente



Após o terceiro Update
Update i = 3 val = 5
tempo t = 3

Segment Tree Persistente

Vejamos o que de fato foi alterado em cada Update, ou seja, o que cada versão possui de diferente:

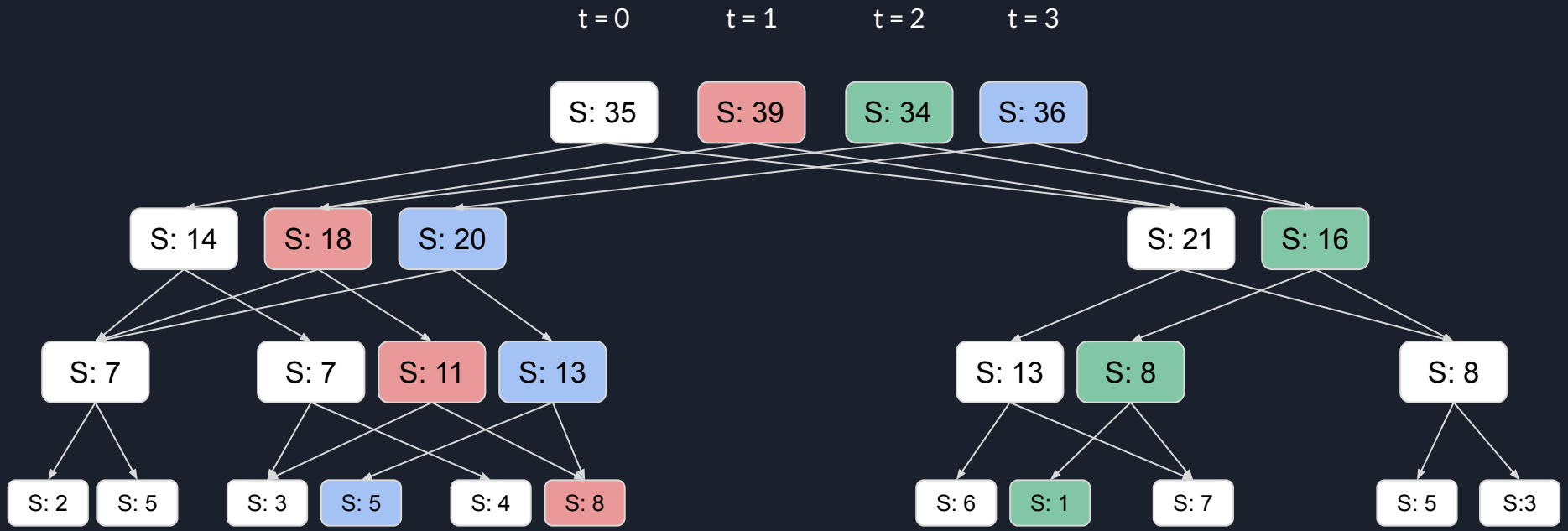


Perceba que cada Update altera um nó por nível, então apenas O(altura) nós são alterados.

Podemos manter todas essas Segs (versões ao longo do tempo) da seguinte forma:

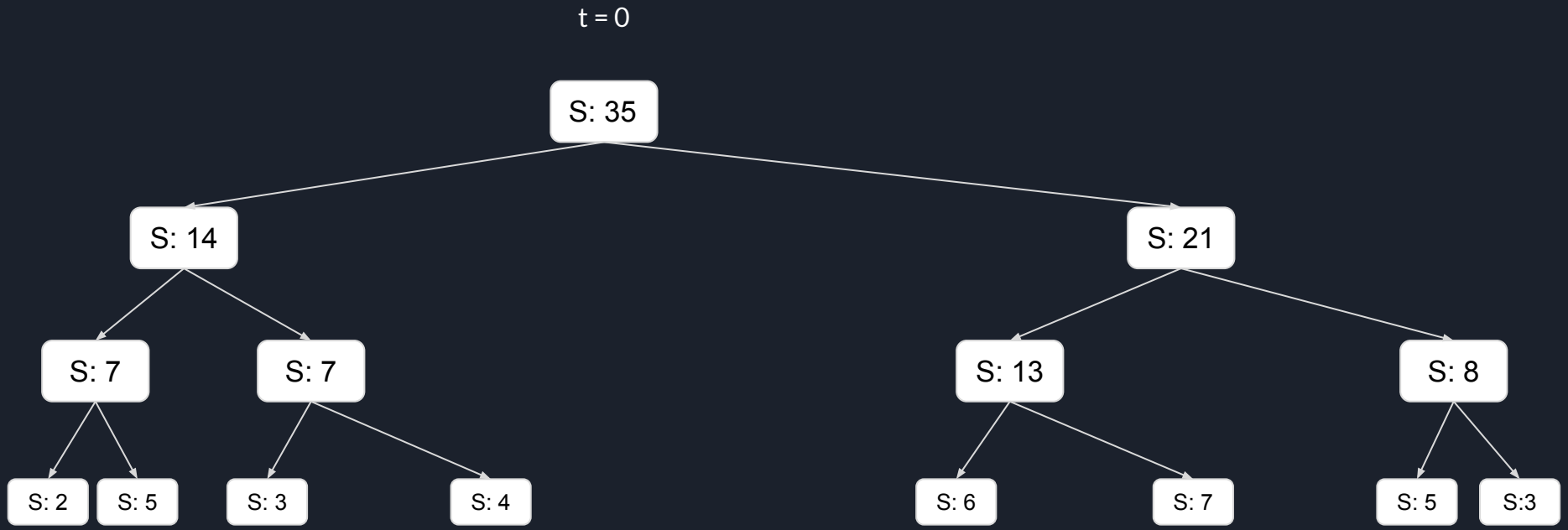
Segment Tree Persistente

Podemos manter todas essas Segs (versões ao longo do tempo) da seguinte forma:



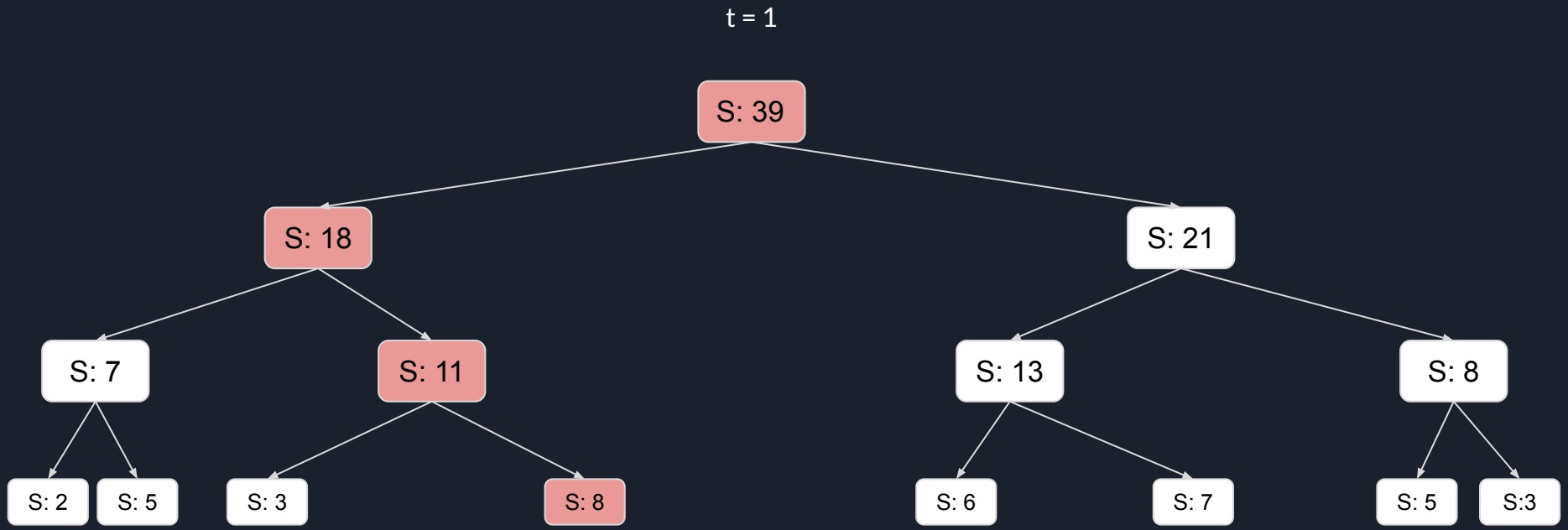
Segment Tree Persistente

Podemos manter todas essas Segs (versões ao longo do tempo) da seguinte forma:



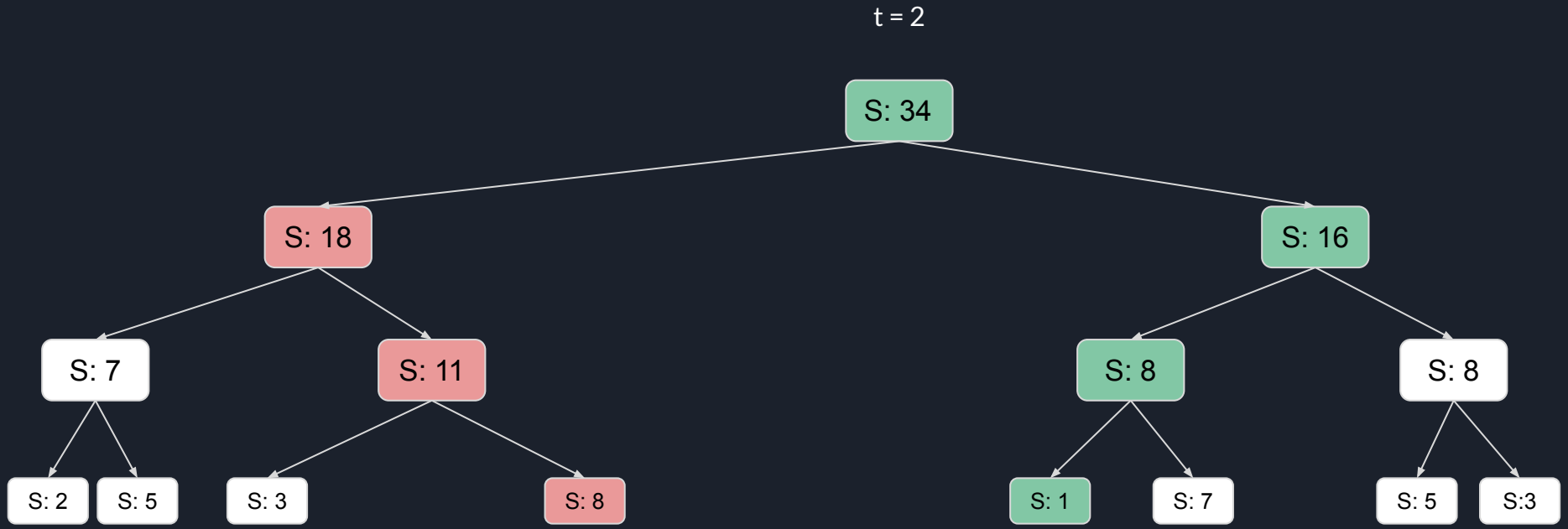
Segment Tree Persistente

Podemos manter todas essas Segs (versões ao longo do tempo) da seguinte forma:



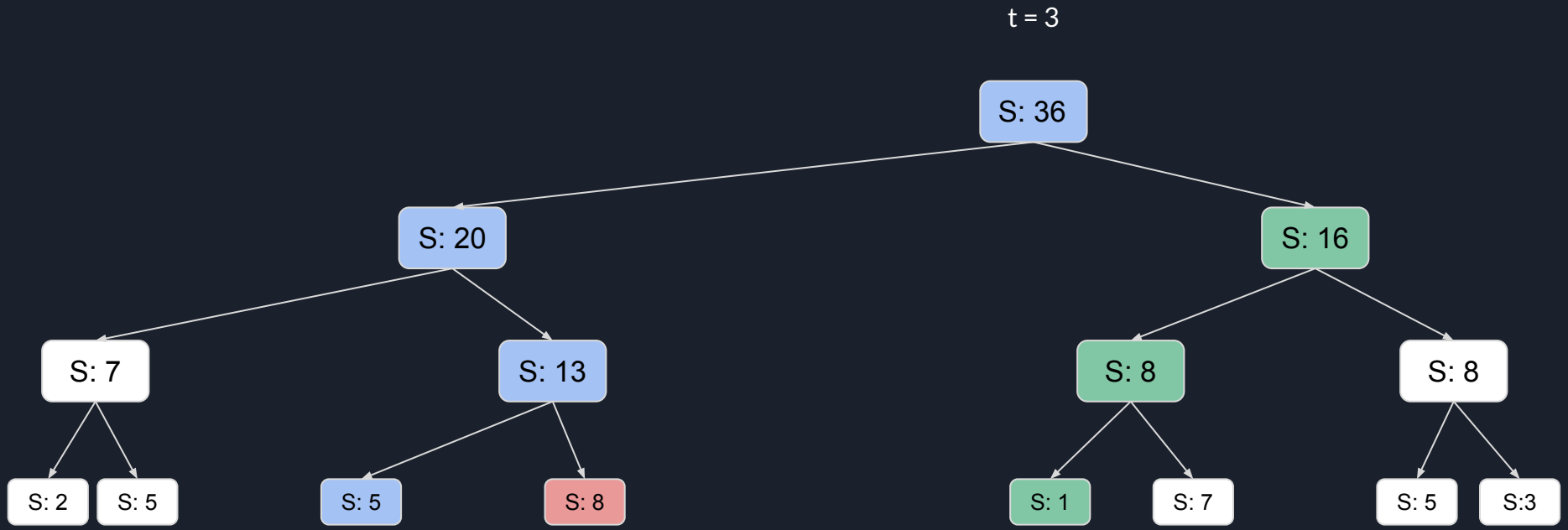
Segment Tree Persistente

Podemos manter todas essas Segs (versões ao longo do tempo) da seguinte forma:



Segment Tree Persistente

Podemos manter todas essas Segs (versões ao longo do tempo) da seguinte forma:



Segment Tree Persistente

Mas como implementar a Seg Persistente ?

Perceba que ela possui muito em comum com a Seg Dinâmica:

- Cada nó terá que guardar o seu filho esquerdo e direito
- Iremos manter várias Segs ao mesmo Tempo

Por isso, iremos implementar a Seg Dinâmica com algumas alterações, ou seja, nossa Seg Persistente também será Dinâmica

A principal alteração é: criar um novo nó toda vez que for alterar. Ao invés de simplesmente alterar um nó, iremos criar uma cópia, e alterar esta cópia, assim estaremos criando uma nova versão deste nó.

Em termos de código, a função Update irá retornar o índice desse novo nó alterado. Assim, no início devemos criar o nó nulo (para que ele esteja no índice 0), depois a raiz da primeira versão, e o mais importante, a cada Update teremos uma versão nova, e por isso devemos salvar o índice dessa raiz, que será retornado pelo Update

Segment Tree Persistente

```
int create() {
  soma.push_back(0);
  e.push_back(0);
  d.push_back(0);
  return soma.size()-1;
}
```

```
int update(int pos, int ini, int fim, int id, int val) {
  int novo = create();

  soma[novo] = soma[pos];
  e[novo] = e[pos];
  d[novo] = d[pos];

  if(ini == fim) {
    soma[novo] = val;
    return novo;
  }

  int m = (ini + fim) >> 1;

  if(id <= m)
    e[novo] = update(e[novo], ini, m, id, val);
  else
    d[novo] = update(d[novo], m + 1, fim, id, val);

  soma[novo] = soma[e[novo]] + soma[d[novo]];

  return novo;
}
```

Segment Tree Persistente

Detalhes:

- O código da Query é o mesmo de uma Seg Dinâmica
- Neste caso, recomendo fortemente a convenção de criar o nó nulo (usada no código anterior). Pois se usar a convenção de marcar com -1 quando o nó não existe, o código pode ficar cheio de if's para tratar. O bom do nó nulo é que, seus filhos esquerdo e direito são ele mesmo, e como ele possui os valores default das informações salvas (por exemplo soma igual a 0), não precisamos tratar quando é o nó nulo.
- Cuidado com os comandos:

  ```
  e[novo] = update(e[novo], ini, m, id, val);
  d[novo] = update(d[novo], m + 1, fim, id, val);
  ```

  No C++11 (e anteriores) pode dar problema por conta do resize dos vectors e e d. Para consertar, use uma variável auxiliar: int aux = update(e[novo], ini, m, id, val); e[novo] = aux;
- Lembre-se de salvar as raízes, por exemplo:

  ```
  raiz[i] = update(r[i - 1], -1000000000, 1000000000, id, val);
  ```

Segment Tree Persistente

Lista de Exercícios

- DQUERY resolver on-line
- MKTHNUM
- COT