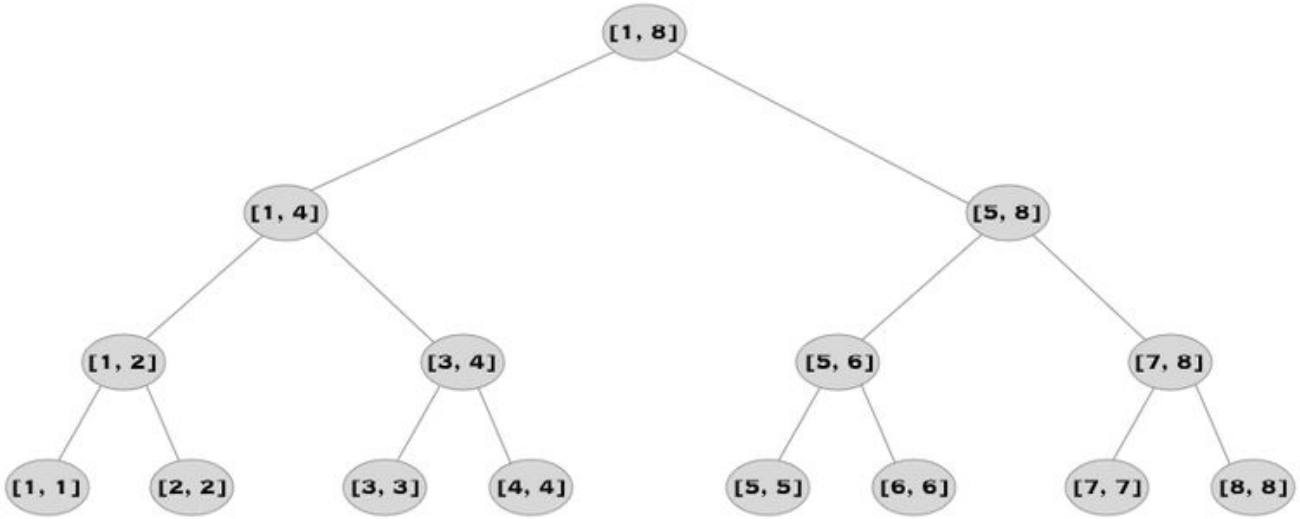
Segment Tree

Giulia,
João Pedro e
Pedro Henrique

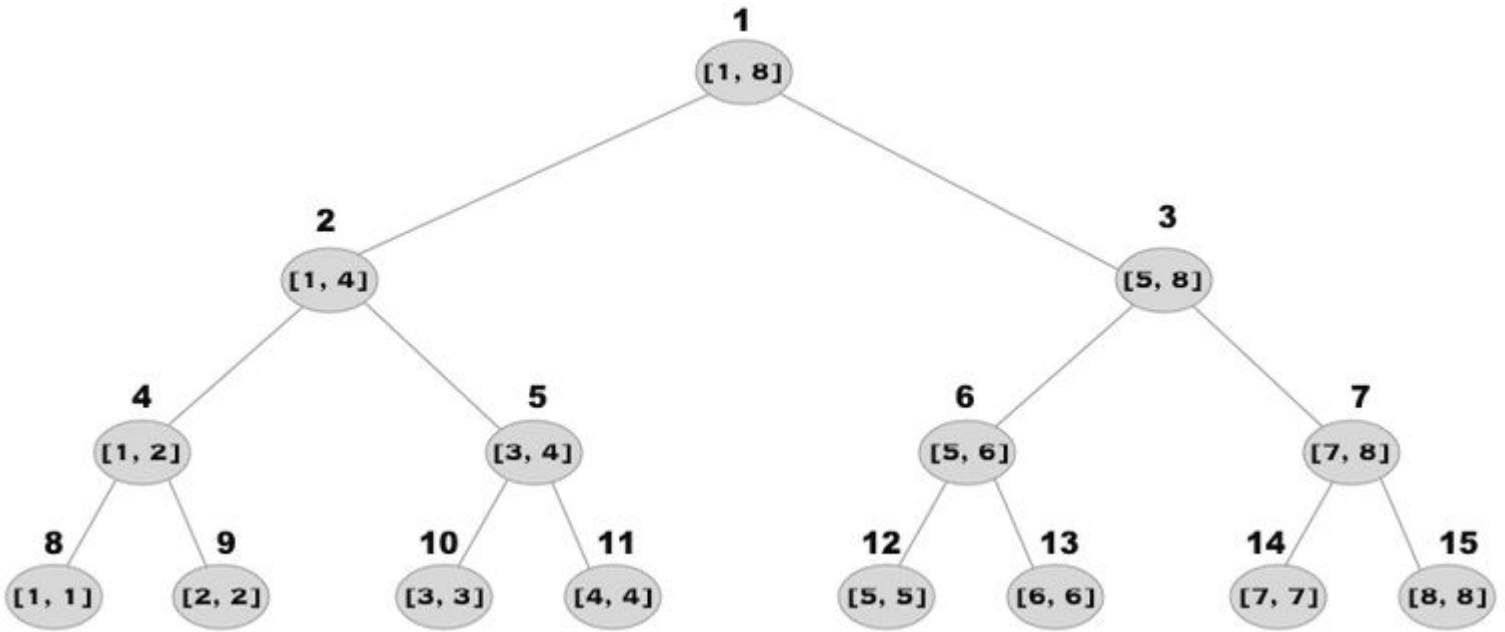
O que é

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



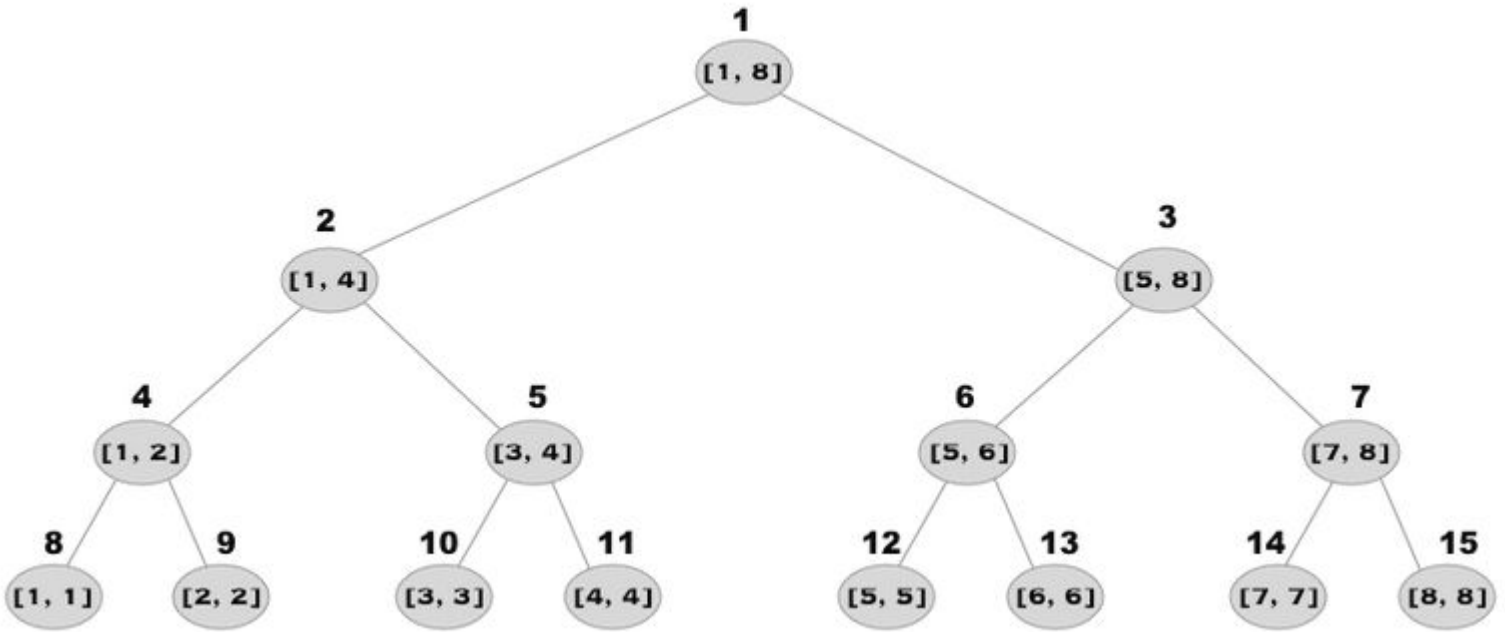
O que é

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



O que é

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



Implementação

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

Range Minimum Query

O que é

- Uma árvore de segmentos em que cada nó armazena a posição do menor elemento de um vetor em um dado intervalo
- Exemplo: [7, 4, 7, 22, 8, 13, 2, 14]

Implementação

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

Implementação

- Construindo a árvore de segmentos
  - Começaremos da raiz e iremos descendo ao longo da árvore, atualizando os vértices recursivamente

Implementação

- Código

```
void contruir(int no, int i, int j)
{
	if(i == j)				//Se estamos em uma folha
		arvore[no] = i;
	else
	{
		int esquerda = 2*no;	// índice do filho da esquerda
		int direita  = 2*no + 1;	// índice do filho da direita
		int meio = (i + j)/2;
```

Implementação

- Código

```
        construir(esquerda, i, meio);
        construir(direita, meio+1, j);
        //atualizando valor do segmento [i,j]
        if( vetor[ arvore[esquerda] ] < vetor[ arvore[direita] ] )
            arvore[no] = arvore[esquerda];
        else
            arvore[no] = arvore[direita];
    }
}
```

Implementação

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

Implementação

- Código

```
void atualiza(int no, int i, int j, int posicao, int novo_valor)
{
	if(i == j)				//Se estamos em uma folha (i == j == posicao)
	{
		arvore[no] = i;
		vetor[posicao] = novo_valor;
	}
	else
	{
		int esquerda = 2*no;	// índice do filho da esquerda
		int direita  = 2*no + 1;	// índice do filho da direita
		int meio = (i + j)/2;
```

Implementação

- Código

```
        if(posicao <= meio)         //posicao está no segmento [i, meio]
            atualiza(esquerda, i, meio, posicao, novo_valor);
        else                        //posicao está no segmento [meio+1, j]
            atualiza(direita, meio+1, j, posicao, novo_valor);
        //atualizando valor do segmento [i,j]
        if( vetor[ arvore[esquerda] ] < vetor[ arvore[direita] ] )
            arvore[no] = arvore[esquerda];
        else
            arvore[no] = arvore[direita];
    }
}
```

Implementação

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

Se [i,j] estiver contido entre [A,B] (A <= i <= j <= B)
    retorna arvore[no]
Se [i,j] e [A,B] forem disjuntos (A > j ou i > B)
    retorna -1
Senão
    chamamos a função recursivamente para os filhos

Implementação

- Código

```
int consulta(int no, int i, int j, int A, int B)
{
        if(A <= i && j <= B)            // [i, j] está contido em [A, B]
                return arvore[no];

        if(i > B || A > j)              // [i, j] e [A, B] são disjuntos
                return -1;

        int esquerda = 2*no;      // índice do filho da esquerda
        int direita  = 2*no + 1;  // índice do filho da direita
        int meio = (i + j)/2;
```

Implementação

- Código

```
int resposta_esquerda = consulta(esquerda, i, meio, A, B);
int resposta_direita  = consulta(direita, meio+1, j, A, B);
if(resposta_esquerda == -1)
        return resposta_direita;
if(resposta_direita == -1)
        return resposta_esquerda;

if(vetor[resposta_esquerda] < vetor[resposta_direita])
        return resposta_esquerda;
else
        return resposta_direita;
}
```

Exemplo - Segment Tree

- Xenia and Bit Operations

http://codeforces.com/contest/339/problem/D

Lazy Propagation

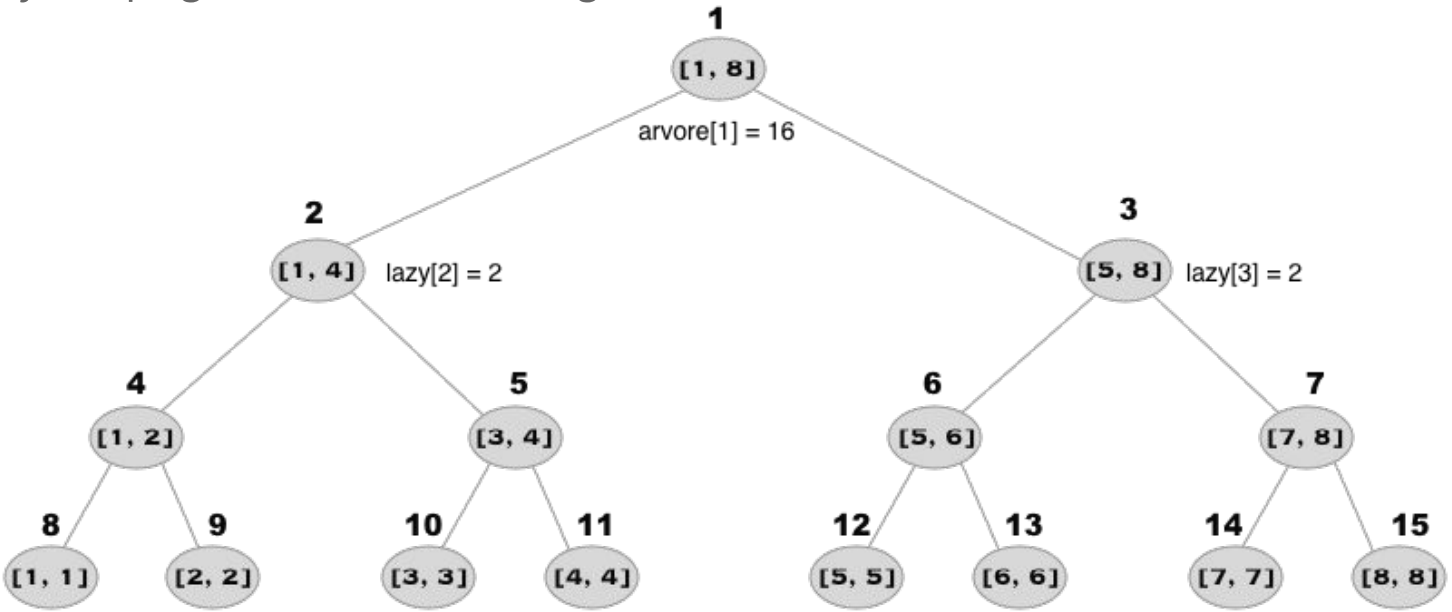
Lazy Propagation

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

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

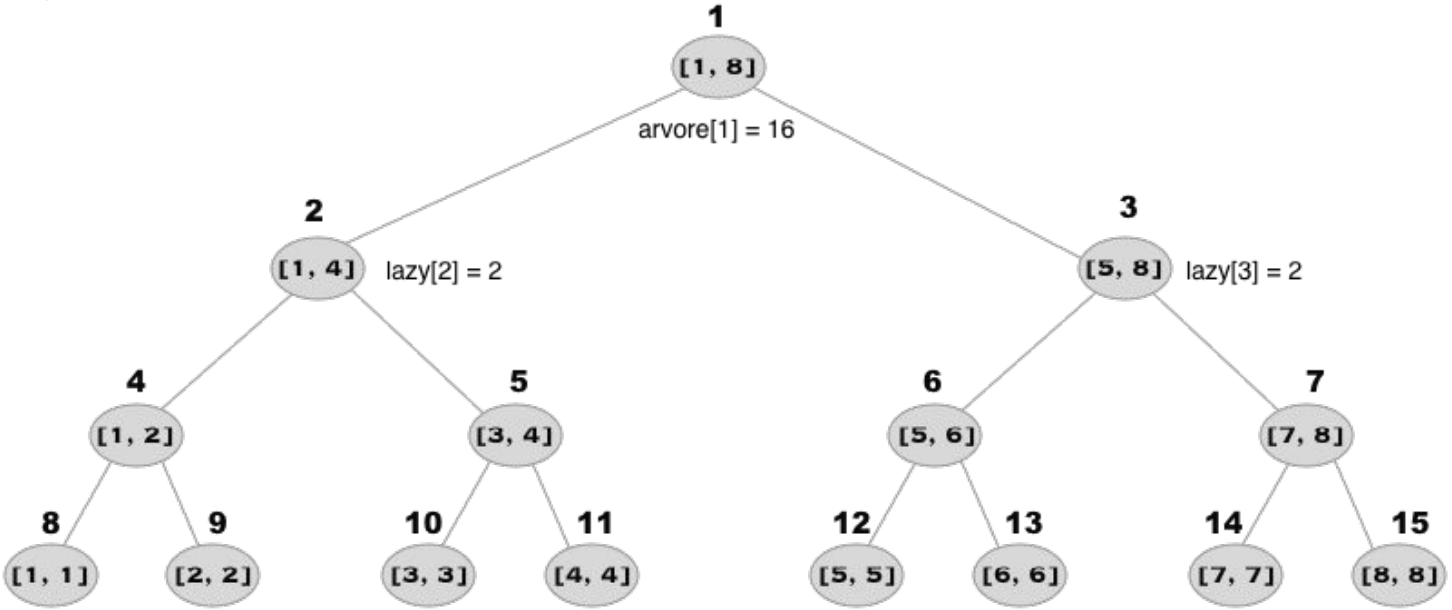
Lazy Propagation

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



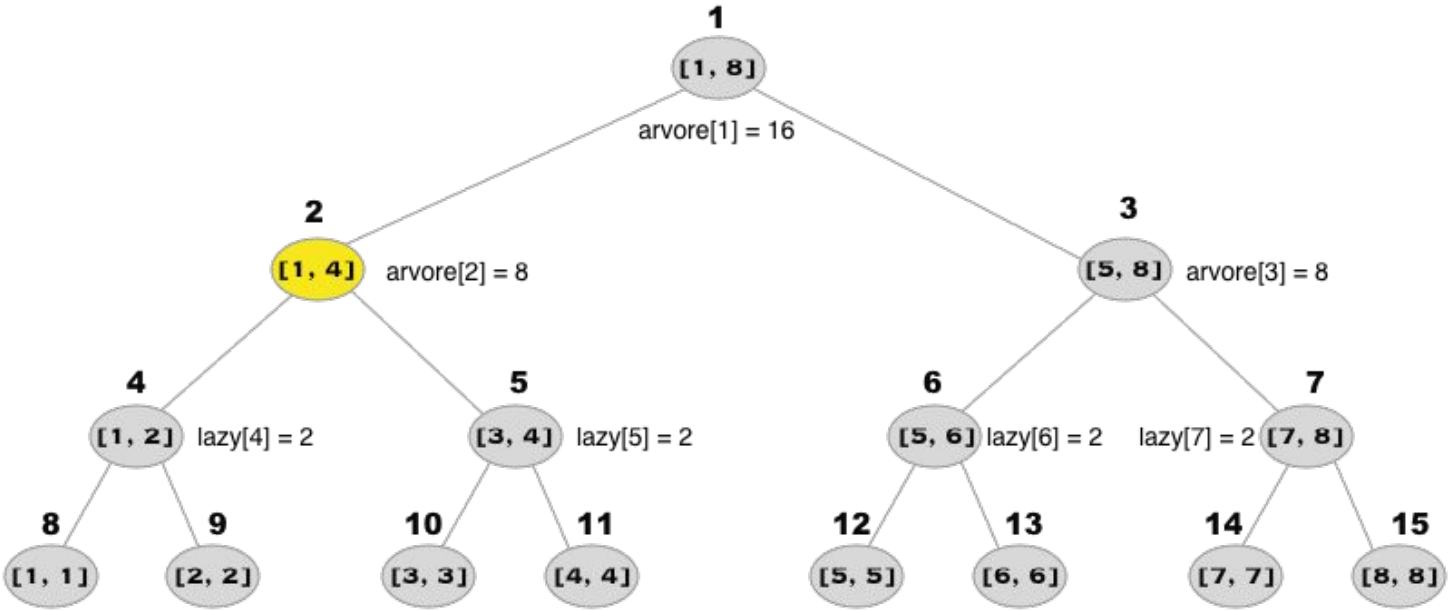
Lazy Propagation

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



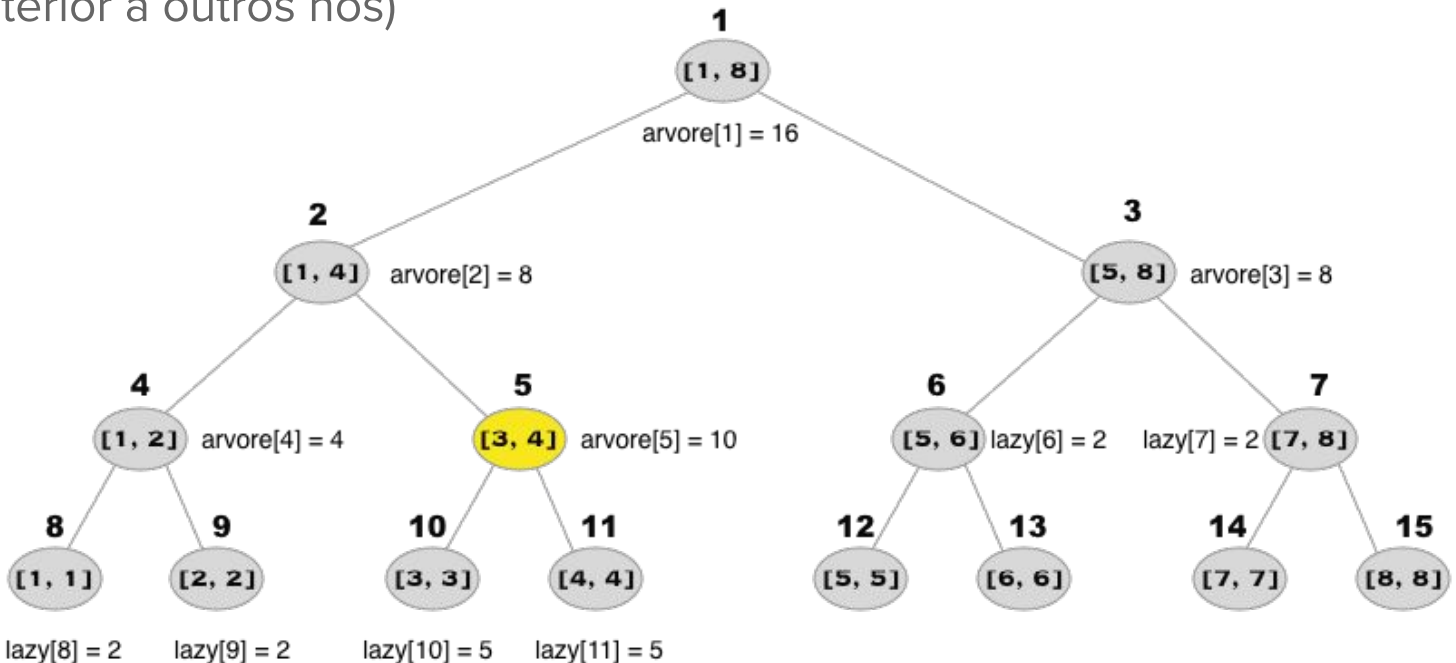
Lazy Propagation

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



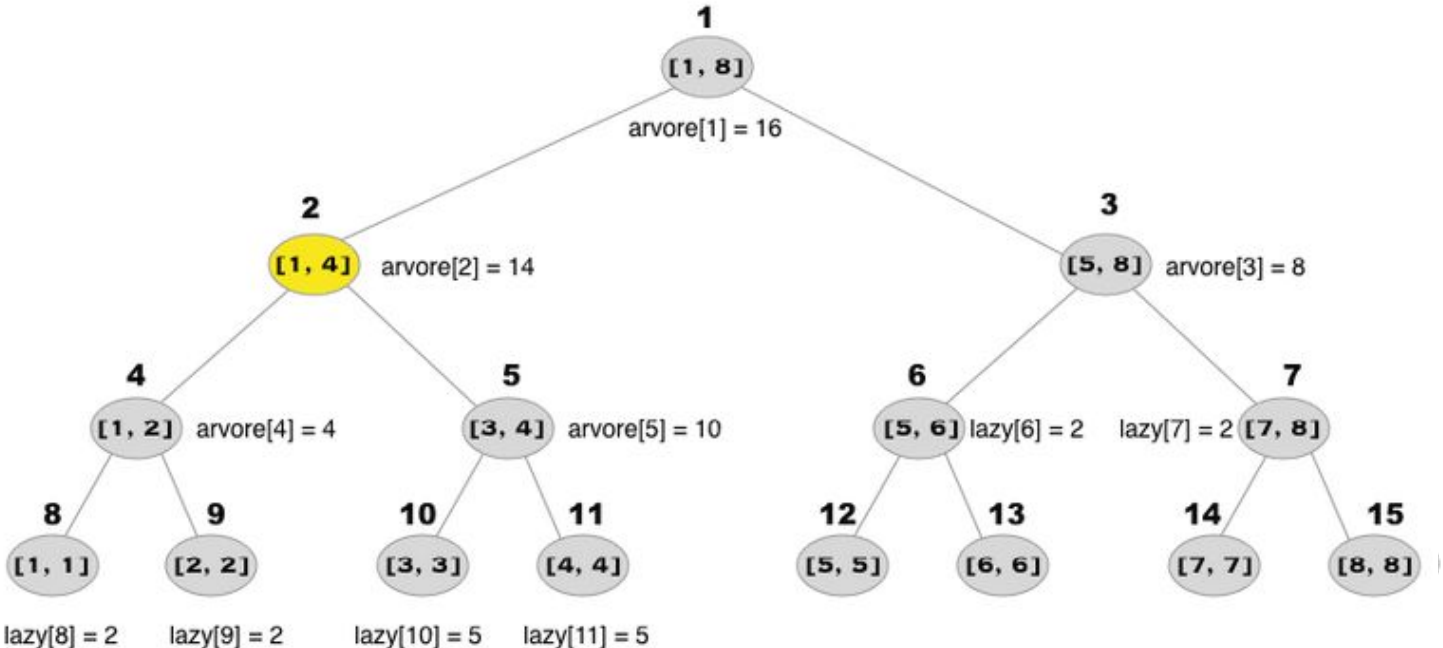
Lazy Propagation

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



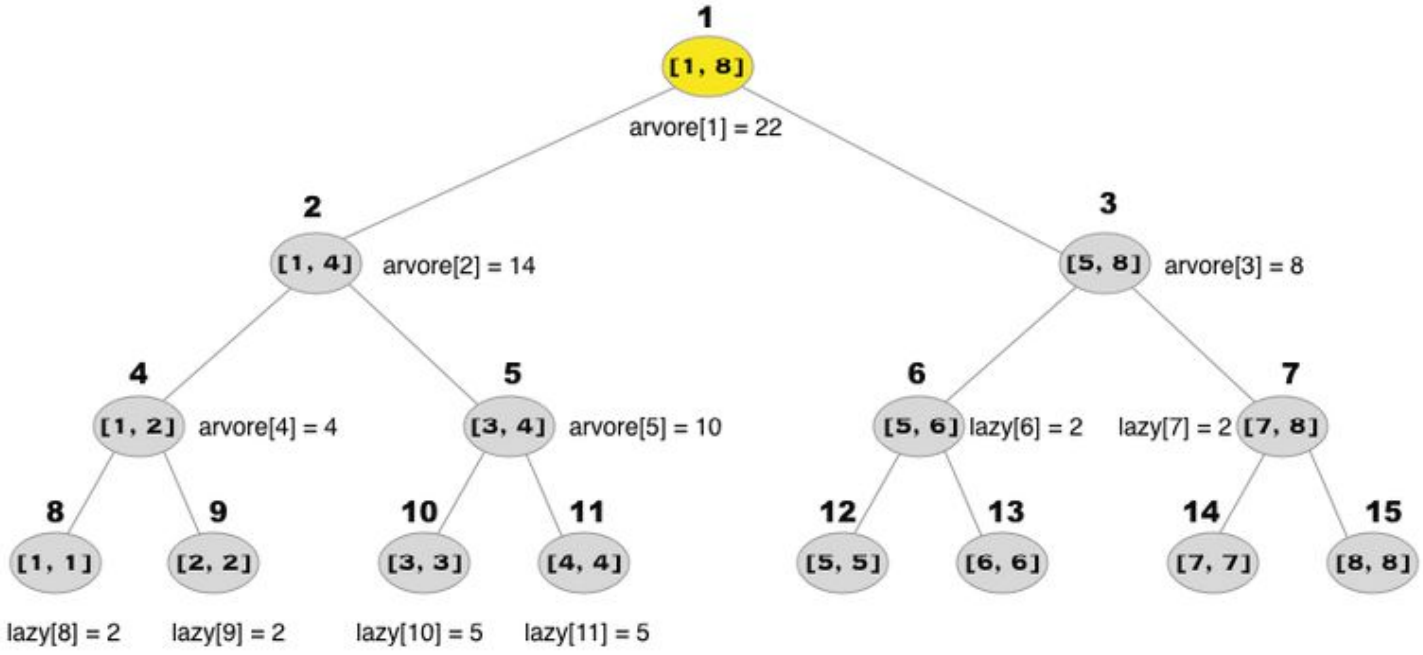
Lazy Propagation

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



Lazy Propagation

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



Lazy Propagation

- Implementação
  - A implementação se apoia no uso de um vetor auxiliar (‘lazy’) para indicar se há alguma atualização a ser feita em um determinado nó. Esse vetor deve ser utilizado tanto na função atualiza() quanto na consulta(), pois pode-se desejar consultar um segmento que ainda está esperando ser atualizado.
  - O código fica um pouco mais extenso comparado ao da Segment Tree clássica, por isso não o apresentaremos nesses slides, mas ele pode ser consultado no seguinte link: http://www.codcad.com/lesson/60

Proposta de Exercício

- Acordes Intergaláticos - Maratona de Programação 2017 / Fase Regional

https://www.urionlinejudge.com.br/judge/pt/problems/view/2658

Referências

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

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

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

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