Fenwick Tree (BIT)
Prefix Sum

Laboratório de Programação Competitiva - 2020

Pedro Henrique Paiola

Prefix Sum

- Considere o problema da Range Sum Query (RSQ), em que dado um vetor com N números queremos fazer consultas. Cada consulta consiste em retornar a soma dos números em um certo intervalo (l,r).
- Na aula anterior vimos que:
    - O método por força-bruta é ineficiente, com complexidade O(n) para as *queries*
    - Podemos resolver isto usando Segment Tree, com as consultas a atualizações tendo complexidade O(log n)

Prefix Sum

- Outra possibilidade é utilizar um **vetor de soma de prefixos**.
- Basicamente, uma posição **i** desse vetor armazena a soma de todos os valores entre **0** e **i**.

$$P[i] = \sum_{j=0}^{i} v[j]$$

- Com essas informações, podemos responder uma consulta (l,r) muito facilmente, como veremos a seguir:

Prefix Sum

- Dados o vetor v e seu vetor de soma de prefixos P subjacente:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
v	5	10	-6	8	1	-1	7	3	-4
P	5	15	9	17	18	17	24	27	23

Prefix Sum

- Dados o vetor v e seu vetor de soma de prefixos P subjacente:

	0	1	2	3	4	5	6	7	8
v	5	10	-6	8	1	-1	7	3	-4
P	5	15	9	17	18	17	24	27	23

- O valor P[5], por exemplo, representa a soma dos valores v[0...5]

Prefix Sum

- Dados o vetor v e seu vetor de soma de prefixos P subjacente:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
v	5	10	-6	8	1	-1	7	3	-4
P	5	15	9	17	18	17	24	27	23

- Agora e se quisermos saber a soma dos valores entre um intervalo (l,r) qualquer, como (3,5)?

Prefix Sum

- Dados o vetor v e seu vetor de soma de prefixos P subjacente:

	0	1	2	3	4	5	6	7	8
v	5	10	-6	8	1	-1	7	3	-4
P	5	15	9	17	18	17	24	27	23

- Podemos considerar isso como equivalente a seguinte operação:

$$P[5] - P[2] = (v[0] + v[1] + v[2] + v[3] + v[4] + v[5]) - (v[0] + v[1] + v[2])$$

Prefix Sum

- Assim, podemos generalizar uma consulta q como sendo:
  - $q(l,r) = P[r] - P[l-1]$
- Por este método, temos as seguintes complexidades:
  - Alteração: $O(n)$
  - Consulta: $O(1)$
- Esta é uma ED muito interessante para quando não há (ou há poucas) atualizações nos valores do vetor.

Prefix Sum - Aplicações

- **Encontrar o índice de equilíbrio:**
    - Encontrar o índice i para qual: v[0...i-1] = v[i+1...n-1]
    - Solução: buscar **i** para qual vale que

$$P[i-1] == P[n-1] - P[i]$$

Prefix Sum - Aplicações

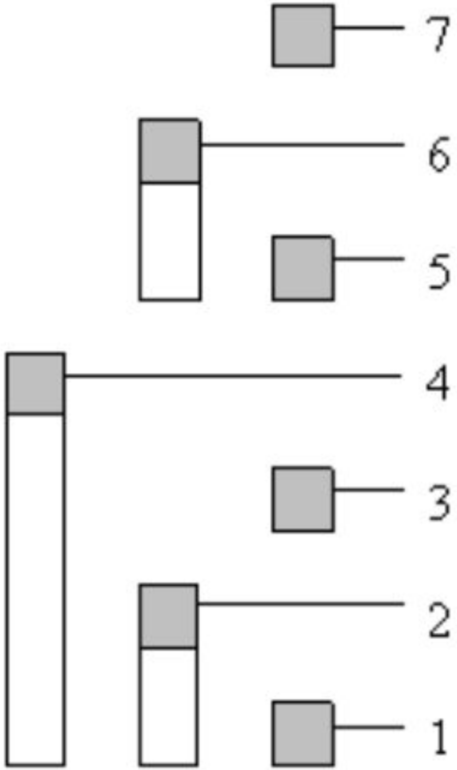
- **Descobrir se existe algum segmento com soma 0:**
  - Podemos procurar por valores repetidos em P.
  - Se P[r] = x e P[l-1] = x para algum (l,r), então q(l,r) = x - x = 0

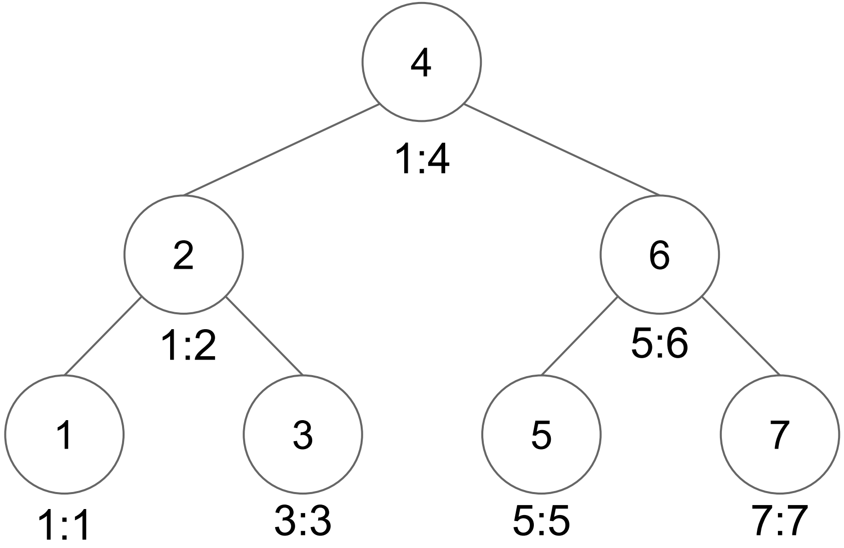
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
v	5	10	-6	8	1	-1	7	3	-4
P	5	15	9	17	18	17	24	27	23

Árvore de Fenwick (BIT)

- Uma alternativa para o problema da RSQ, baseada em Prefix Sum, é o uso da estrutura de dados *Binary Indexed Tree* (BIT) ou Árvore de Fenwick.
- Assim como a SegTree, esta estrutura faz atualizações e consultas em O(log n). Porém, na prática, ela é mais rápida que uma SegTree.
- Também de forma semelhante a SegTree, construiremos uma árvore em que cada nó irá armazenar a soma dos valores de um certo segmento do vetor de dados.

Árvore de Fenwick (BIT)

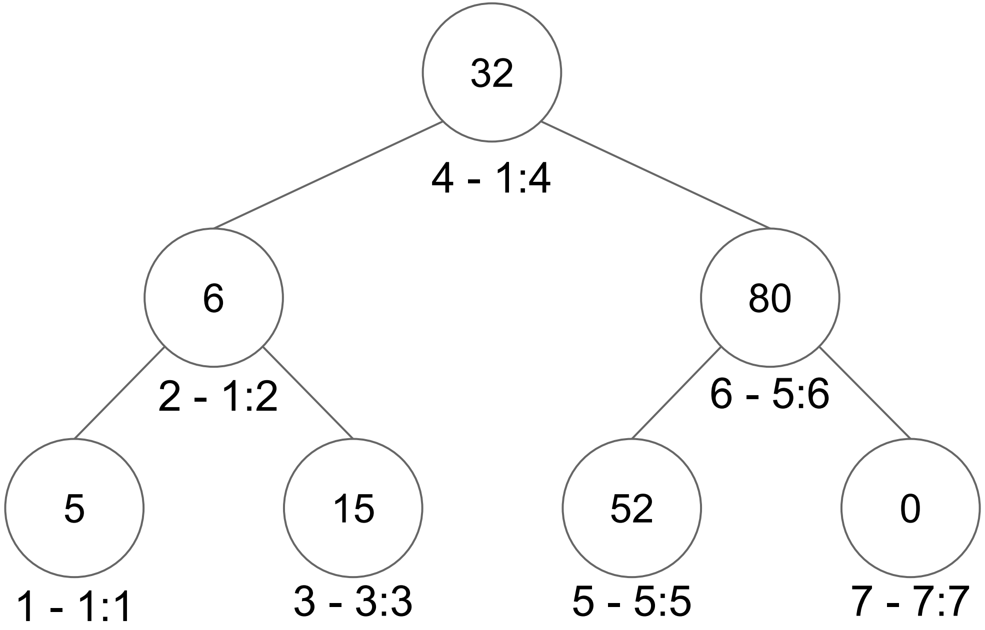




Árvore de Fenwick (BIT)

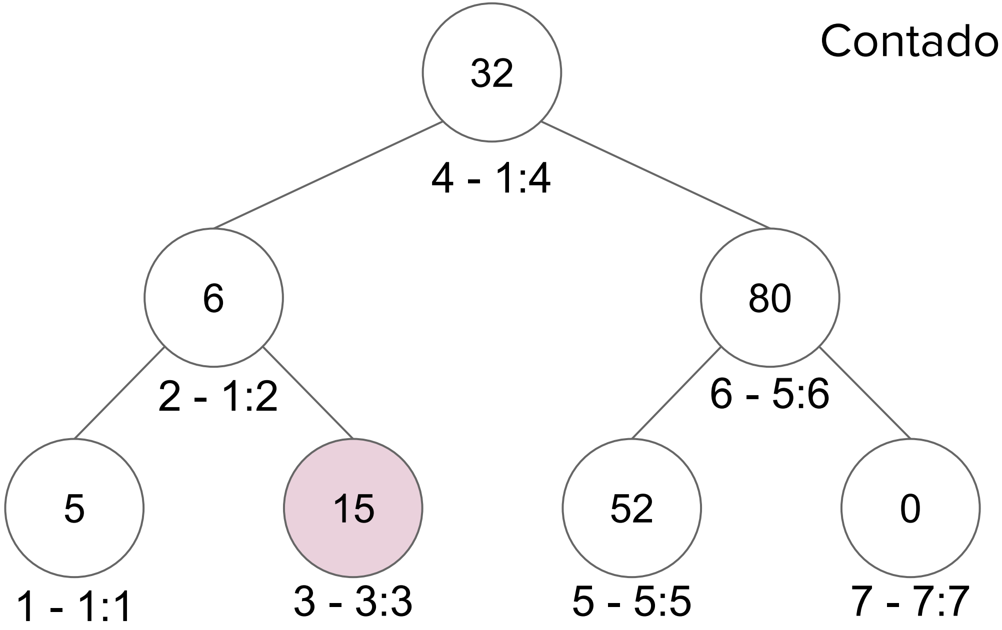
- Realizaremos **consultas** nessa árvore tal qual em um vetor de soma de prefixos. Ou seja, se quisermos descobrir a soma do intervalo (l,r), realizaremos duas *queries* na árvore: q(r) e q(l-1).
- Iniciaremos uma *query* **q(x)** pela raiz, com um **contador** iniciado com **0**. Buscando pelo nó **x** na árvore de busca, sempre que “descermos” para o filho da direita, somaremos o valor do nó atual ao contador. Quando encontrarmos o nó desejado, também somaremos seu valor.
- Também podemos pensar de forma contrária, onde começamos em **x** e subimos até a raiz. Sempre que subimos para a **esquerda** iremos somar a variável contador.

Árvore de Fenwick (BIT)



1	5
2	1
3	15
4	11
5	52
6	28
7	0

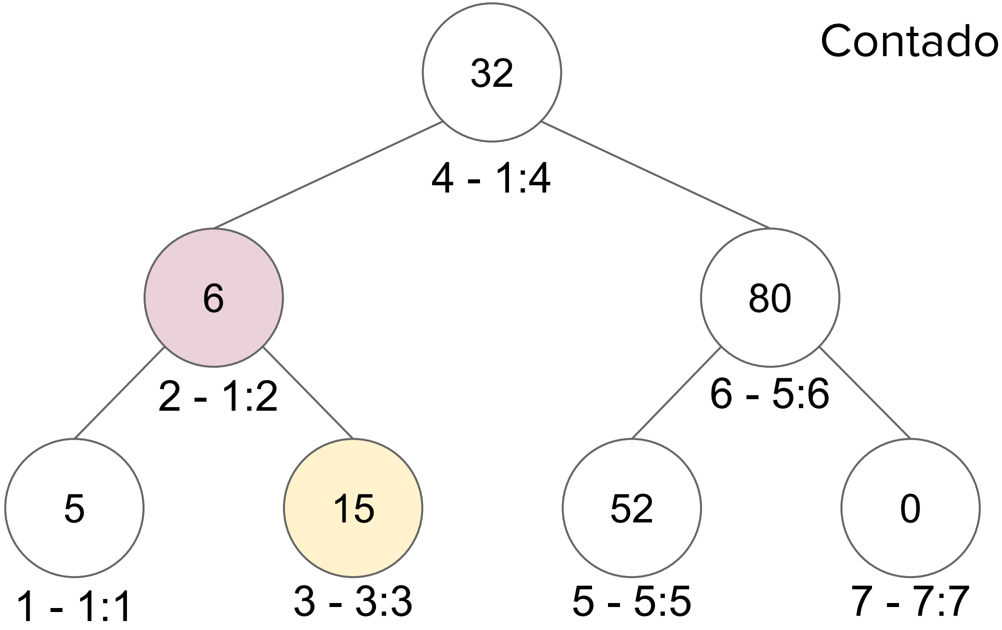
Árvore de Fenwick (BIT)



Contador = 15

1	5
2	1
3	15
4	11
5	52
6	28
7	0

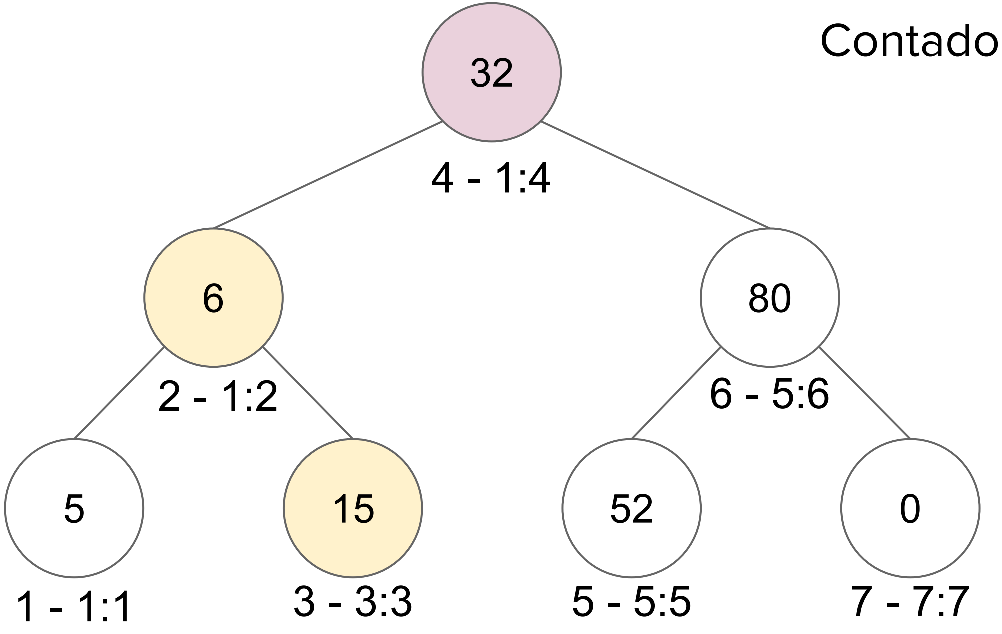
Árvore de Fenwick (BIT)



Contador = 15 + 6

1	5
2	1
3	15
4	11
5	52
6	28
7	0

Árvore de Fenwick (BIT)



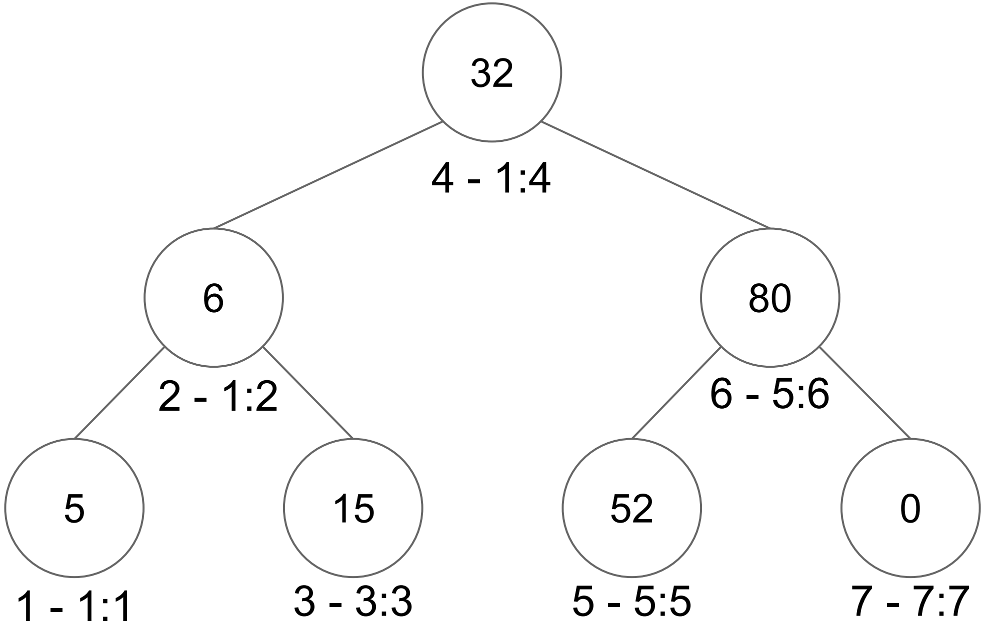
Contador = 21

1	5
2	1
3	15
4	11
5	52
6	28
7	0

Árvore de Fenwick (BIT)

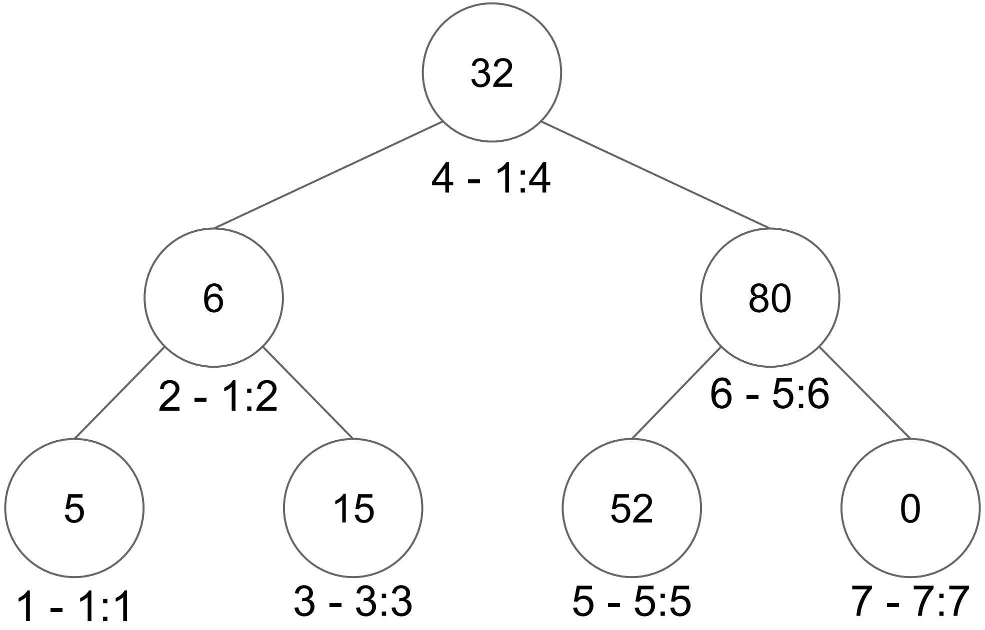
- Para alterar o valor de um nó, faremos de forma semelhante. Começando do nó a ser alterado, iremos subir até a raiz.
- Agora, quando subimos para a **direita** iremos atualizar o valor do nó em que estamos: bit[no] = bit[no] - valor_antigo + valor_novo

Árvore de Fenwick (BIT)



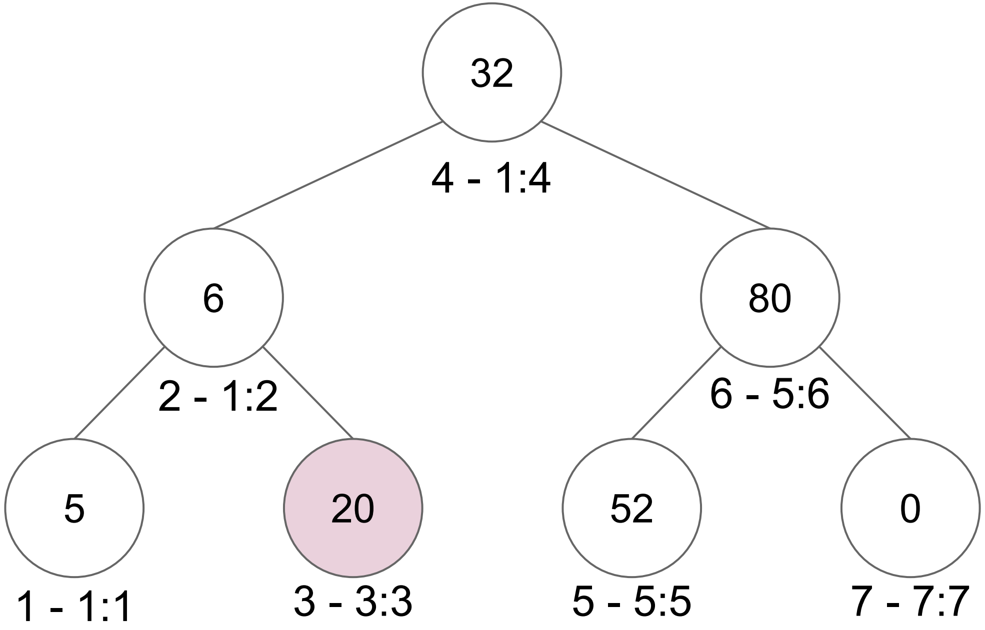
1	5
2	1
3	15
4	11
5	52
6	28
7	0

Árvore de Fenwick (BIT)



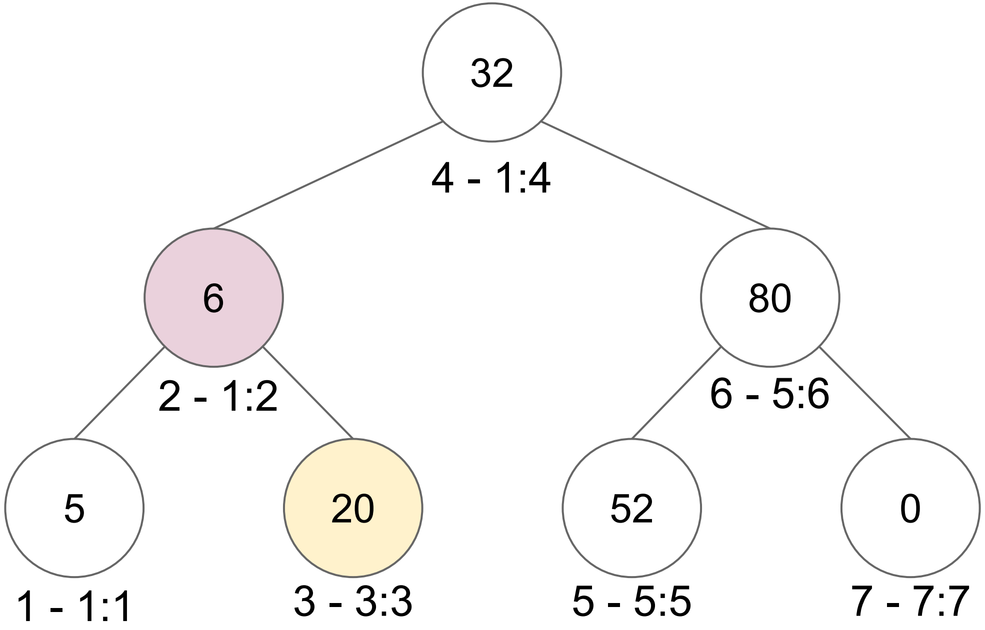
1	5
2	1
3	**20**
4	11
5	52
6	28
7	0

Árvore de Fenwick (BIT)



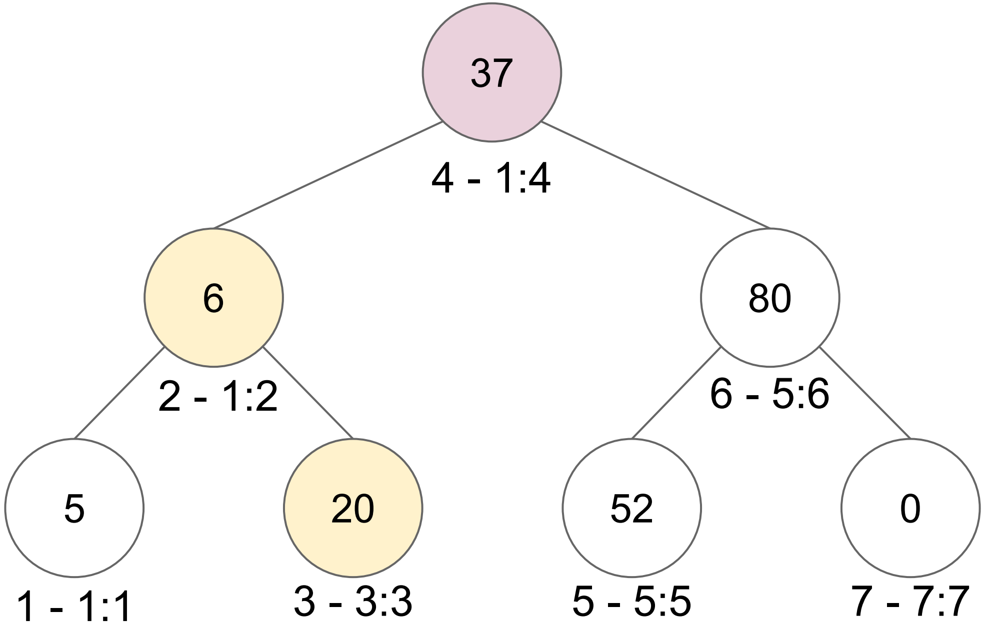
1	5
2	1
3	**20**
4	11
5	52
6	28
7	0

Árvore de Fenwick (BIT)



1	5
2	1
3	**20**
4	11
5	52
6	28
7	0

Árvore de Fenwick (BIT)



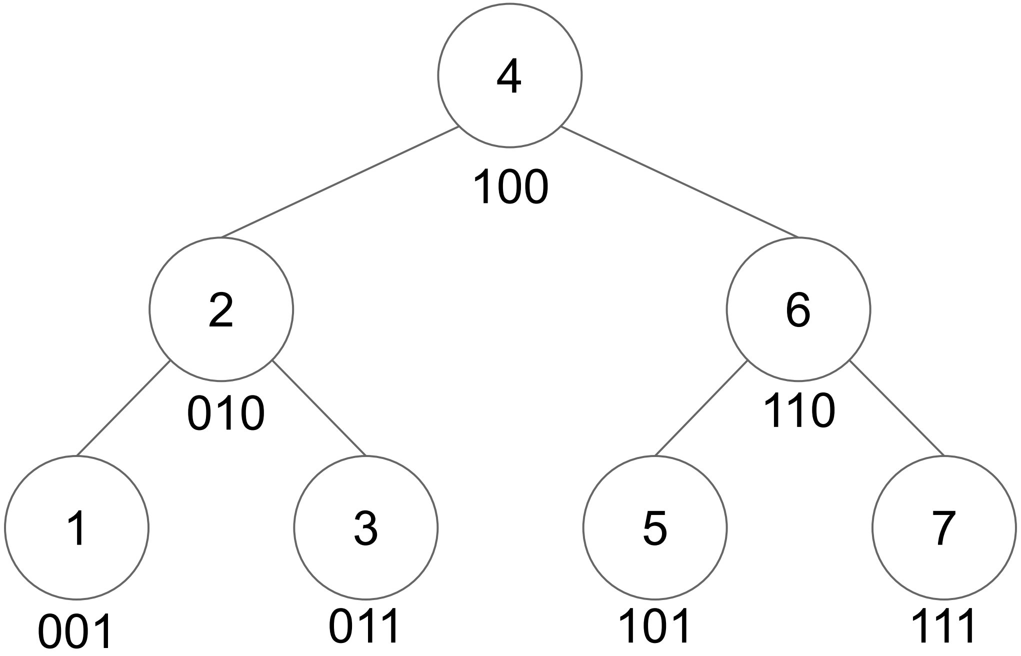
1	5
2	1
3	**20**
4	11
5	52
6	28
7	0

Árvore de Fenwick (BIT)

- Até o momento, não temos uma estrutura com grande vantagem em comparação a SegTree.
- A chave está na implementação dessa estrutura, baseada em operações bit-a-bit e em propriedades interessantes da árvore de Fenwick.

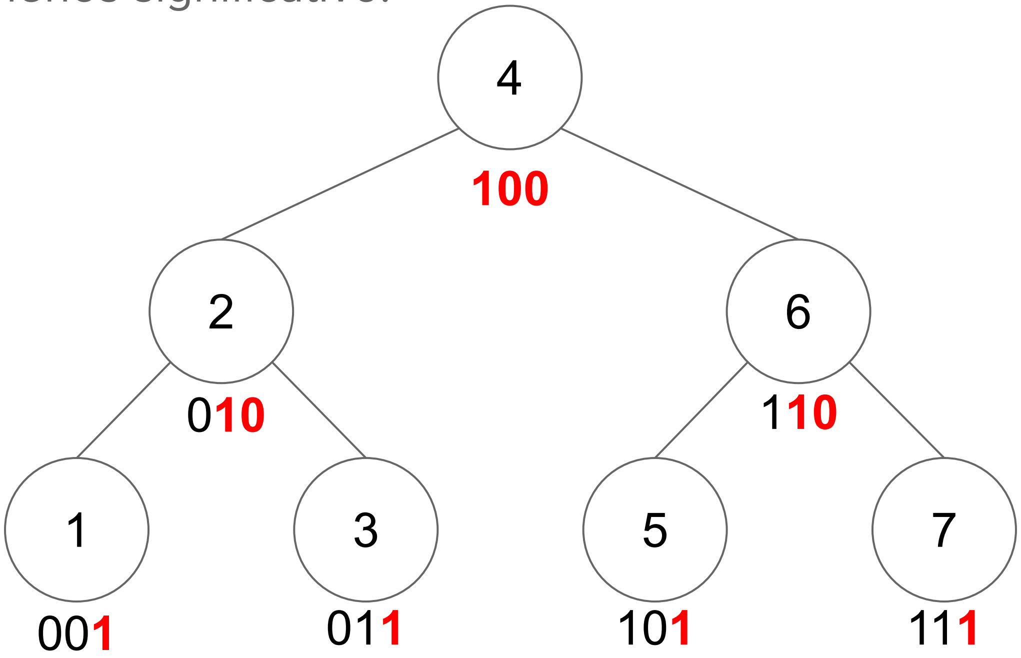
Árvore de Fenwick (BIT)

- Vamos olhar para a representação binária dos nós de nossa árvore.



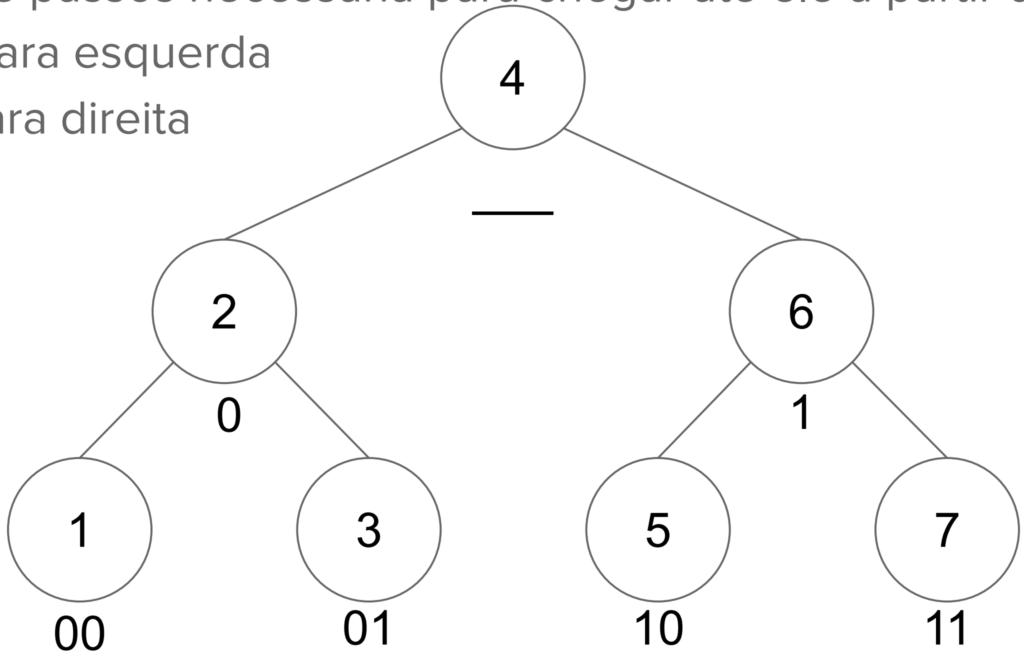
Árvore de Fenwick (BIT)

- E então, para cada nó, vamos retirar da representação binária todos os dígitos após o bit 1 menos significativo.



Árvore de Fenwick (BIT)

- Perceba que agora cada nó está anotado com uma representação da sequência de passos necessária para chegar até ele a partir da raíz.
- 0 = descer para esquerda
- 1 = descer para direita



Árvore de Fenwick (BIT)

- E por que isto nos ajuda? Lembre que para realizar uma consulta começamos do nó e subimos para a raiz, mas só estamos preocupado com as informações dos nós que atingimos ao subir para a esquerda.
- De forma análoga, na atualização alteramos apenas as informações dos nós que atingimos ao subir pela direita.
- Na prática, iremos “pular” os nós que não nos interessam.
- Operação para obter o bit menos significativo de x: **x & -x**

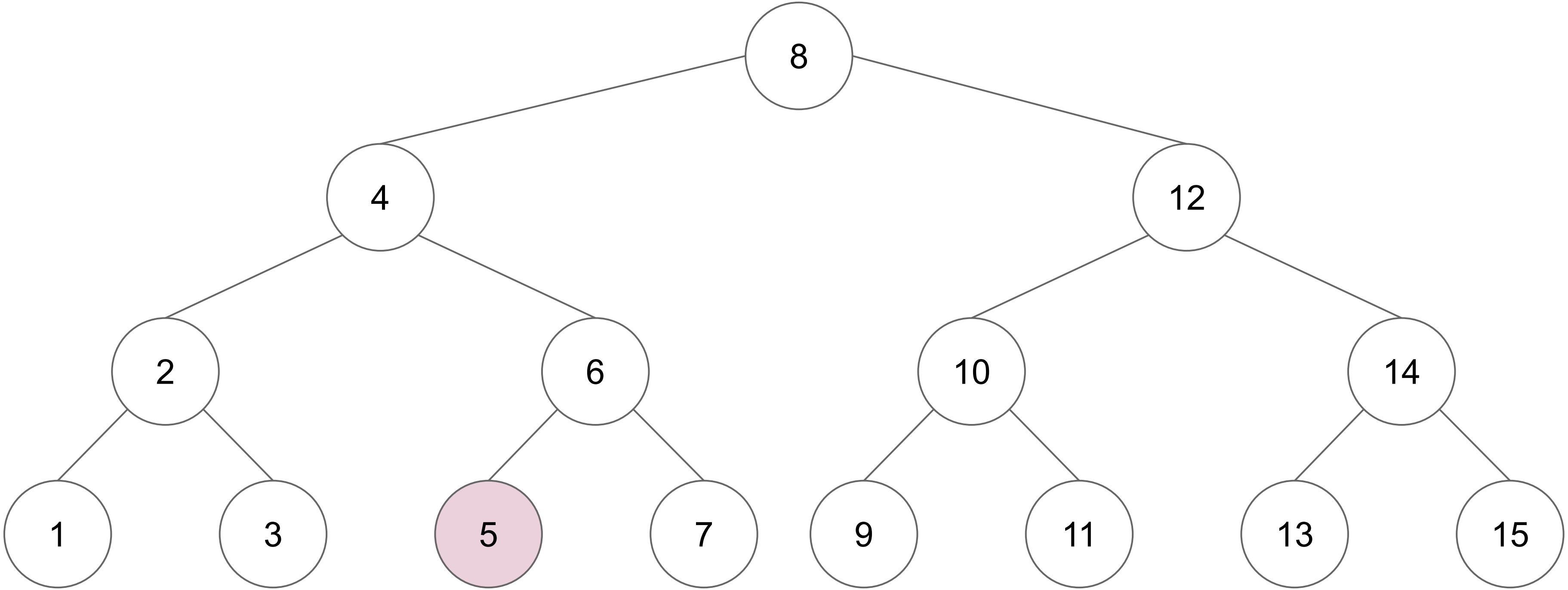
```
      20 = 10100
     -20 = 01100
20 & -20 = 00100 = 4
```

Árvore de Fenwick (BIT)

- Consulta:

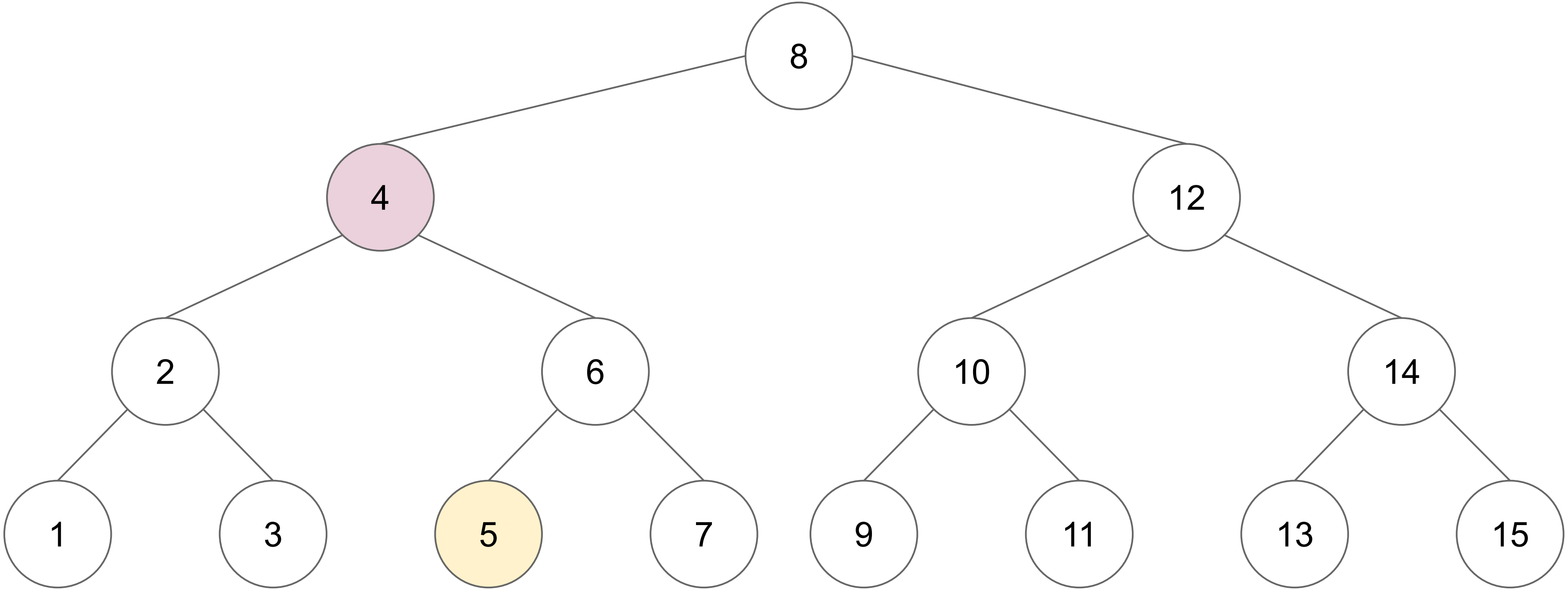
```
int query(int x){
    int sum = 0;
    x++;
    while(x > 0){
        sum += BIT[x];
        x -= (x & -x);
    }
    return sum;
}
```

Árvore de Fenwick (BIT)



q(5) => 5 & -5 = 0101 & 1011 = 0001 => 5 - 1 = **4**

Árvore de Fenwick (BIT)



q(4) => 4 & -4 = 0100 & 1100 = 0100 => 4 - 4 = **0 FIM**

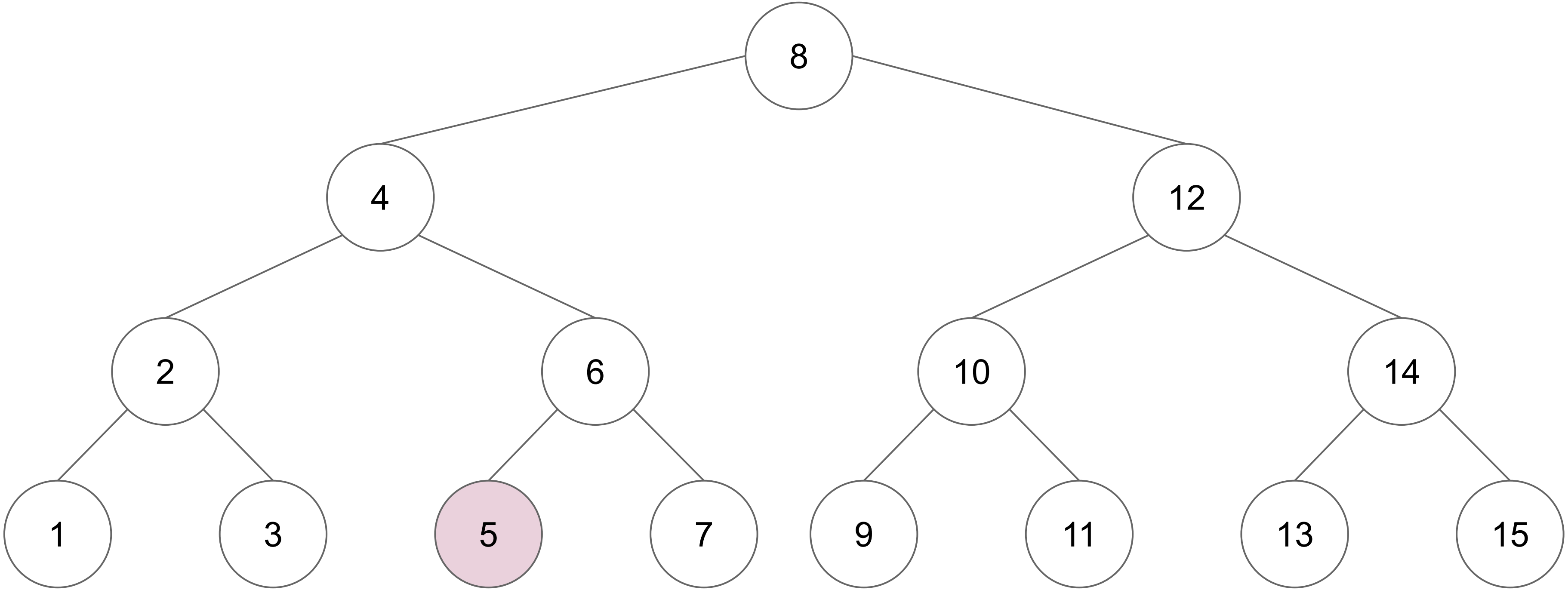
Árvore de Fenwick (BIT)

- Alteração:

```
void add(int x, int val){
    x++;
    while(x <= n){
        BIT[x] += val;
        x += (x & -x);
    }
}
```

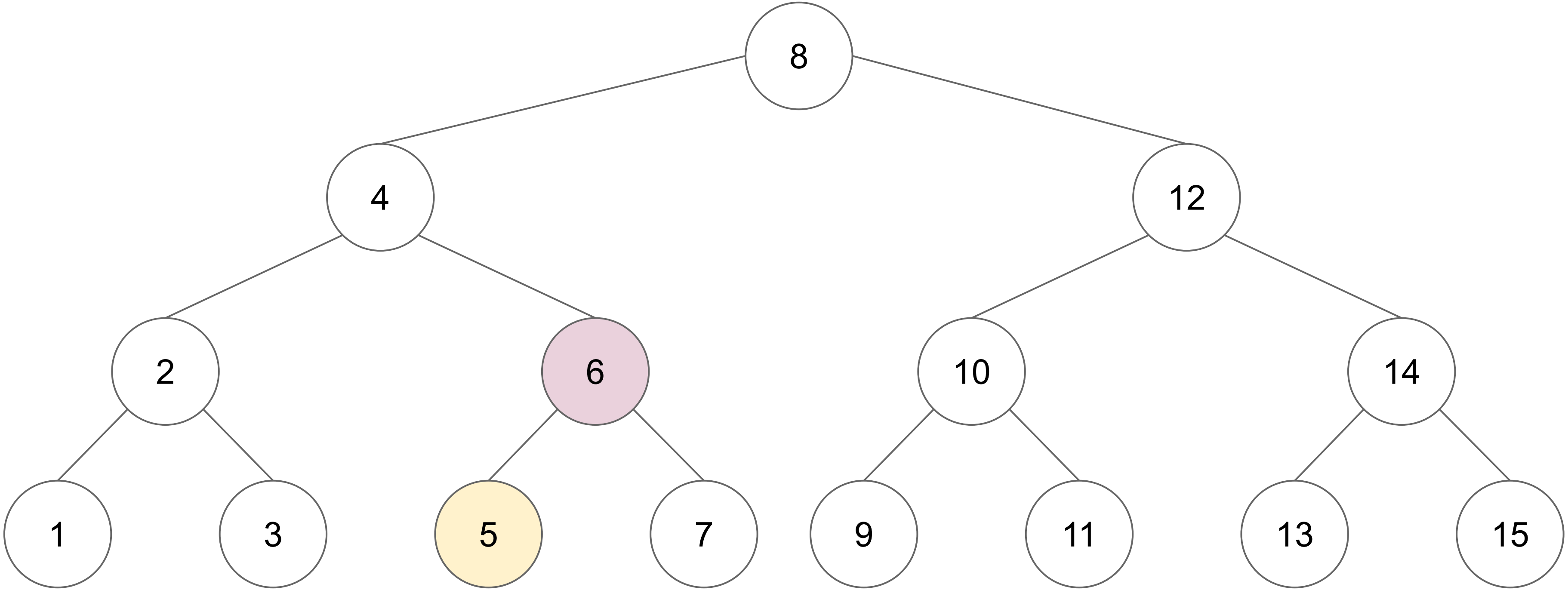
```
void update(int x, int val){
    add(x, val-v[x]);
    v[x] = val;
}
```

Árvore de Fenwick (BIT)



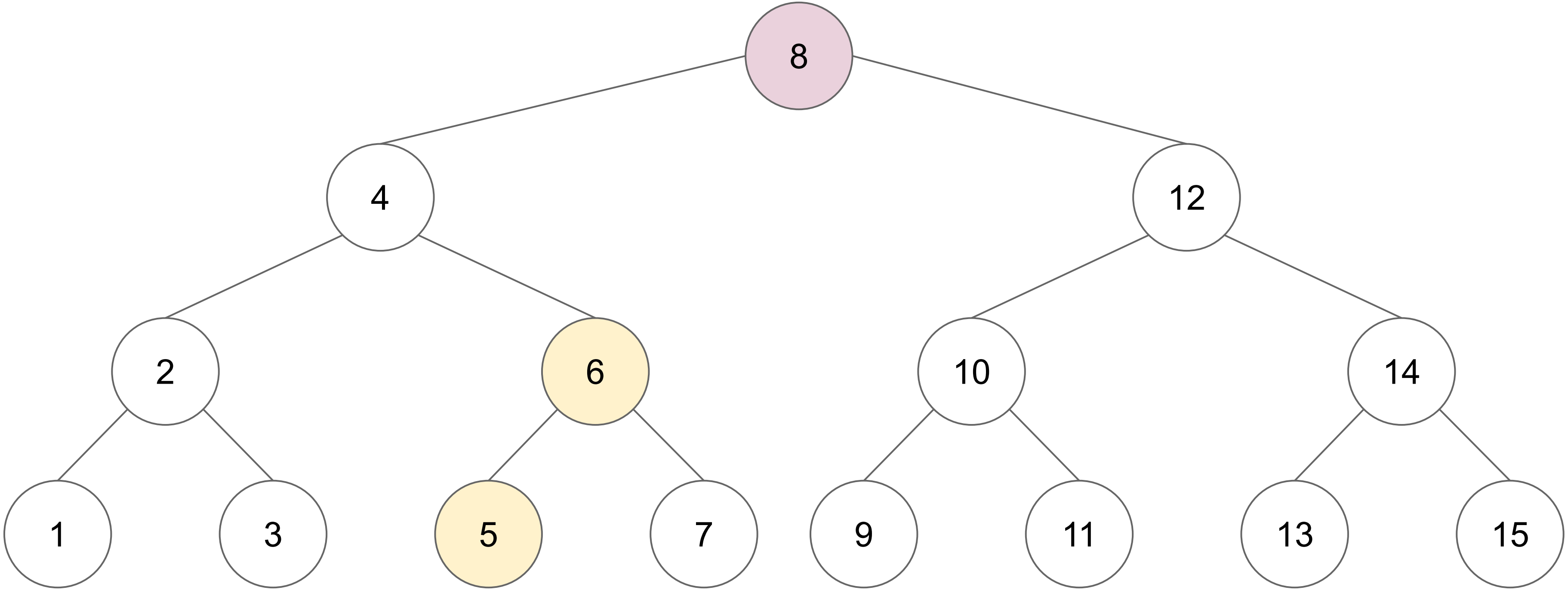
upd(5) => 5 & -5 = 0101 & 1011 = 0001 => 5 + 1 = **6**

Árvore de Fenwick (BIT)



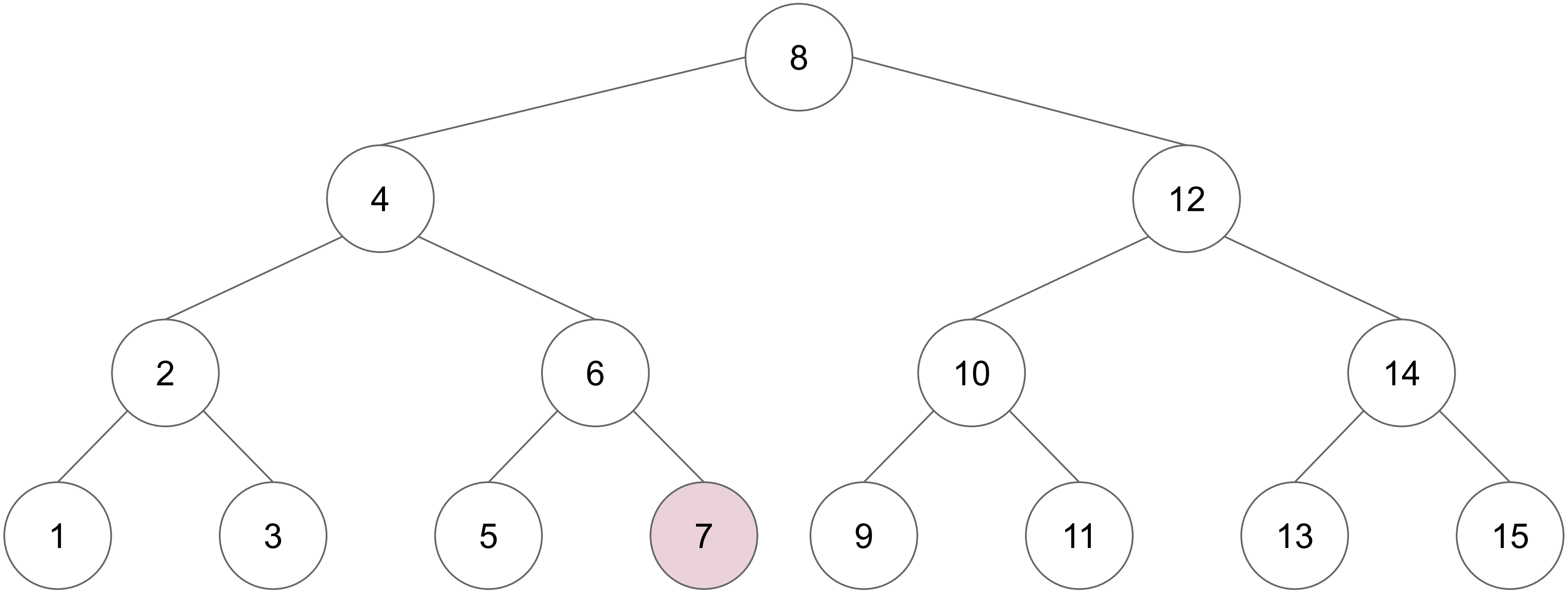
upd(6) => 6 & -6 = 0110 & 1010 = 0010 => 6 + 2 = **8**

Árvore de Fenwick (BIT)



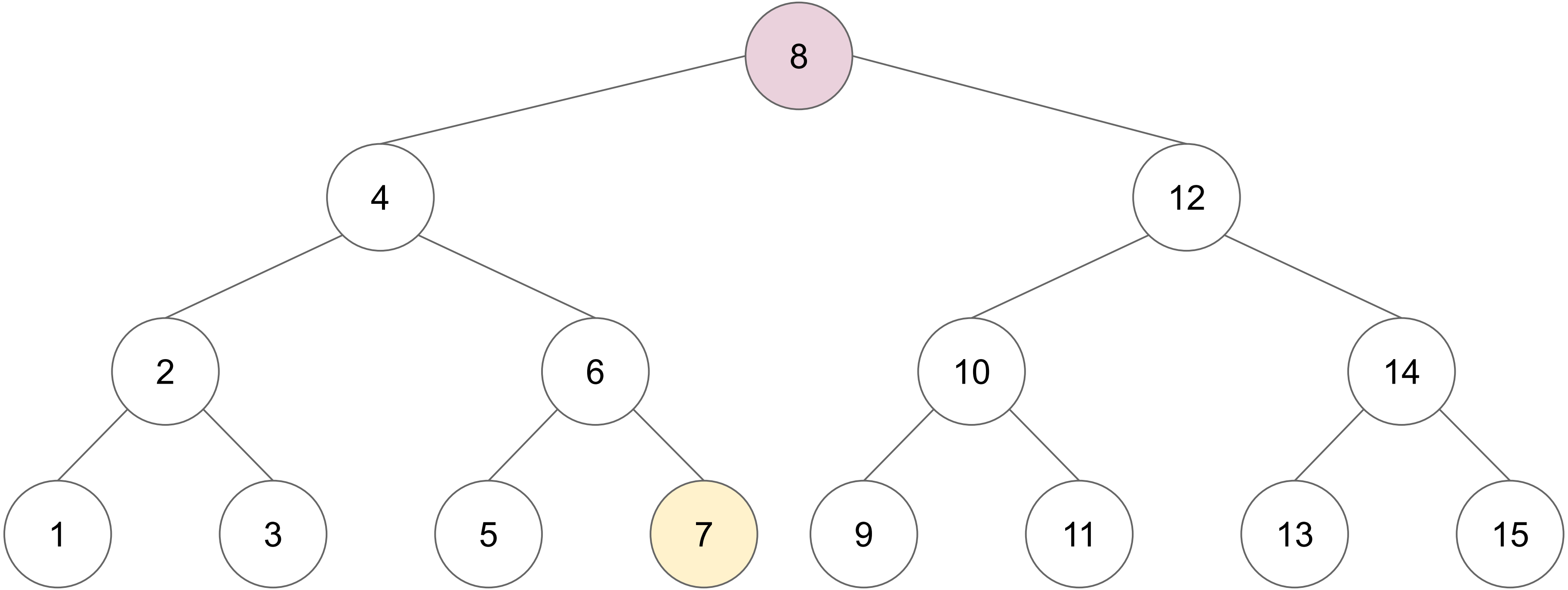
upd(8) => 8 & -8 = 1000 & 1000 = 1000 => 8 + 8 = **16 > 15 FIM**

Árvore de Fenwick (BIT)



upd(7) => 7 & -7 = 0111 & 1001 = 0001 => 7 + 1 = **8**

Árvore de Fenwick (BIT)



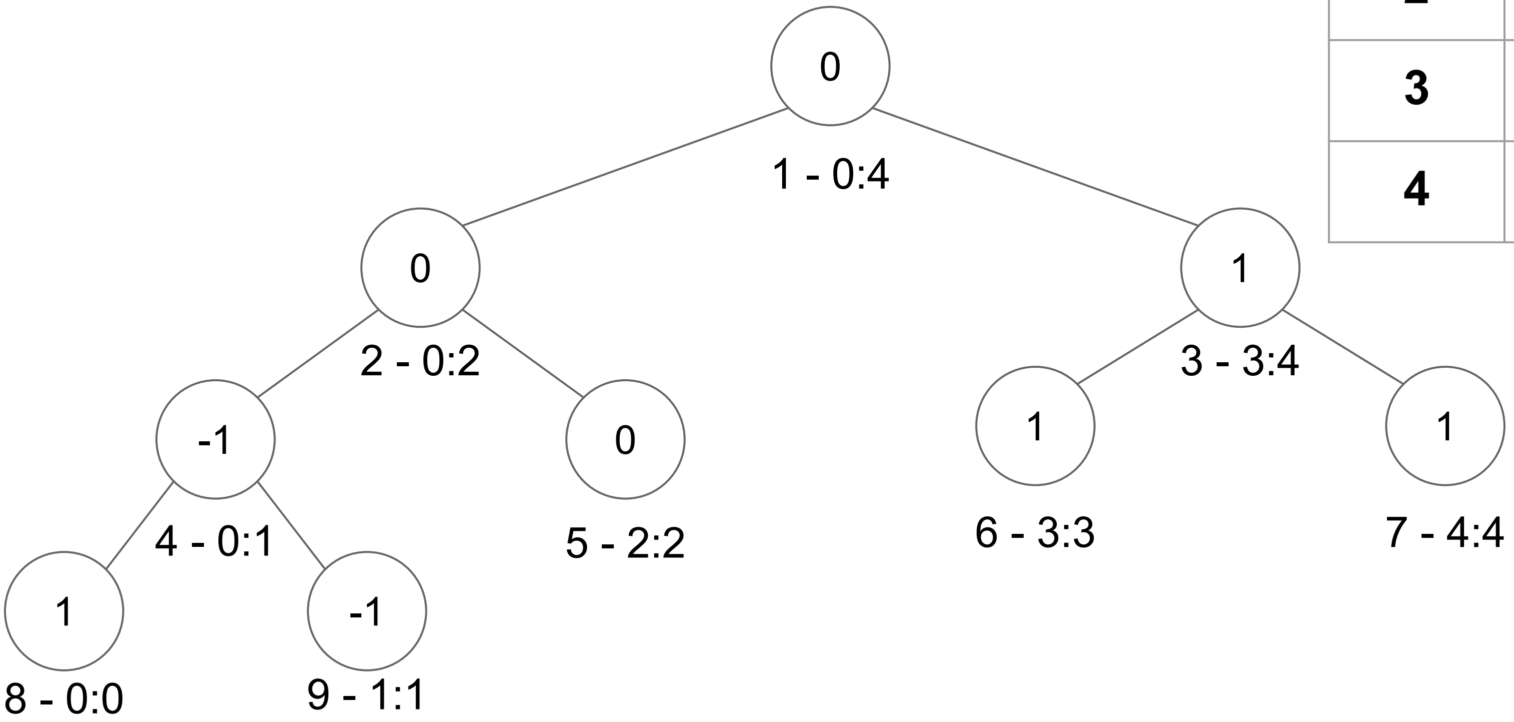
upd(8) => 8 & -8 = 1000 & 1000 = 1000 => 8 + 8 = **16 > 15 FIM**

Interval Product (UVA - 12532)

- Duas operações possíveis a partir de um vetor **x** de inteiros:
  - C i v => x[i] = v
  - P i j => consulta se o produto x[i] * x[i+1] * ... * x[j] é positivo, negativo ou zero
- Primeiramente, podemos pensar em resolver este problema utilizando uma SegTree.
- Como só estamos interessados no sinal do produto, podemos considerar apenas os sinais dos valores do vetor ao executarmos nossas operações

Interval Product (UVA - 12532)

0	1
1	-1
2	0
3	1
4	1



Interval Product (UVA - 12532)

- Porém, podemos adaptar este problema para utilizar uma BIT ao invés de uma SegTree (mesmo não sendo originalmente um problema de RSQ).
- Para isso podemos nos basear nas seguintes observações:
  - Se em um certo intervalo, se houver pelo menos um número zero, então o resultado é **zero**.
  - Se não há nenhum zero e há um número ímpar de números negativos, então o resultado é **negativo.**
  - Caso contrário, é **positivo.**

Interval Product (UVA - 12532)

- Sendo assim, implementaremos duas BITs, uma para contar a quantidade de zeros, e outra a quantidade de números negativos.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
x	5	10	-6	8	-1	0	7	0	-4
P_{zero}	0	0	0	0	0	1	1	2	2
P_{neg}	0	0	1	1	2	2	2	2	3

Interval Product (UVA - 12532)

```
Se query(Pzero, i, j) > 0, então
    Resultado é zero
Senão, se query(Pneg, i, j) % 2, então
    Resultado é negativo
Senão
    Resultado é positivo
```

Referências

https://github.com/UnBalloon/programacao-competitiva/tree/master/Preffix%20sums%20(Somas%20de%20prefixos)
https://www.geeksforgeeks.org/prefix-sum-array-implementation-applications-competitive-programming/
https://medium.com/beauty-date-stories/algorithms-how-prefix-sums-can-help-improving-operations-over-arrays-b1f8e8141668
https://github.com/UnBalloon/programacao-competitiva/tree/master/Delta%20encoding%20(Codifica%C3%A7%C3%A3o%20de%20diferen%C3%A7as)
https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/
https://cp-algorithms.com/data_structures/fenwick.html

Referências

https://neps.academy/lesson/265
https://qastack.com.br/cs/10538/bit-what-is-the-intuition-behind-a-binary-indexed-tree-and-how-was-it-thought-a
https://www.topcoder.com/community/competitive-programming/tutorials/binary-indexed-trees/