ESCOLA DE PRIMAVERA DA MARATONA SBC DE PROGRAMAÇÃO

PROMOÇÃO:

CEFET-MG UNIFEI UFMG UFU UFV

APOIO:

beecrowd SBC Sociedade Brasileira de Computação

Grupo de Computação Competitiva

SEGMENT TREE

Por: *Yan Pacheco*

CONTEÚDOS

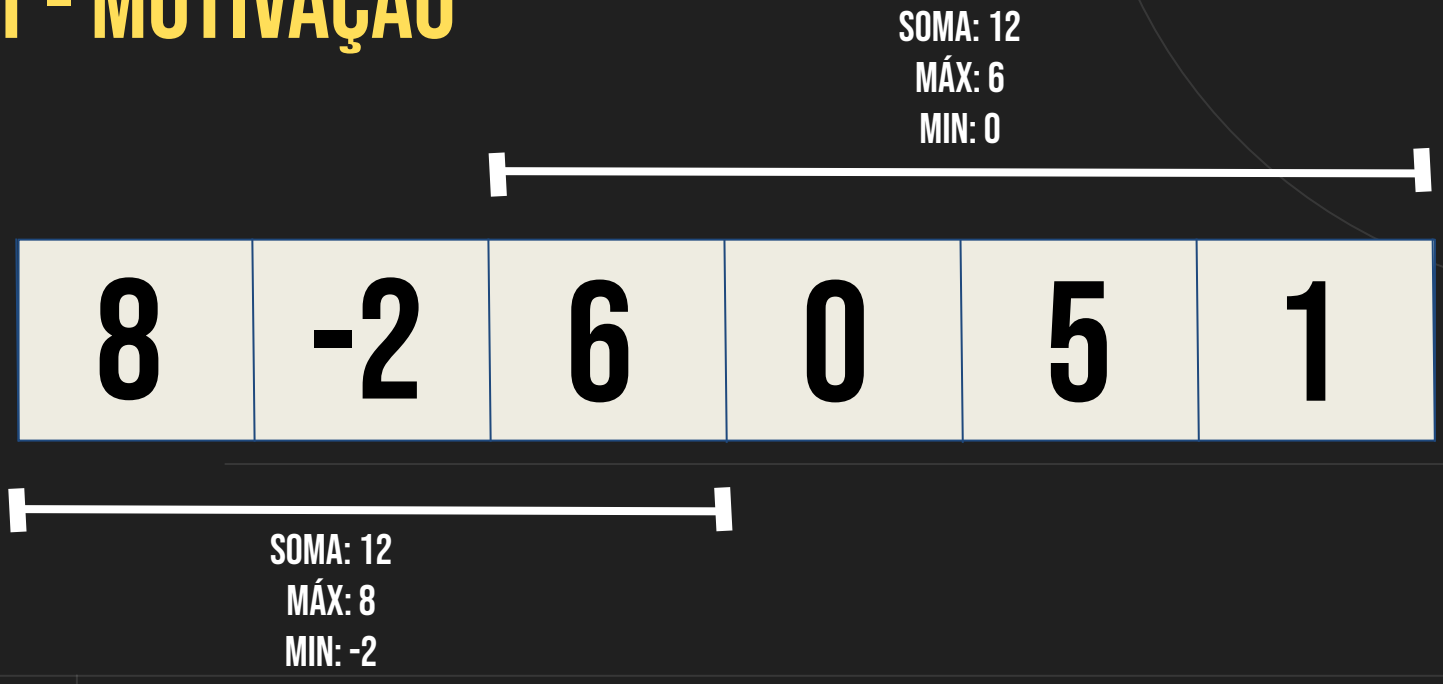
01 - MOTIVAÇÃO

- Em diversos problemas, é necessário que realizemos consultas e operações, em intervalos em um array.

- Por exemplo, consultar o valor máximo/mínimo (Range Maximum/Minimum Query) ou a soma (Range Sum Query) do intervalo deste array.

- A Segment Tree pode responder com eficiência essas consultas, enquanto ainda é flexível o suficiente para permitir modificações rápidas do array.

01 - MOTIVAÇÃO



01 - MOTIVAÇÃO

- Dado o exemplo do RSQ (Range Sum Query), podemos resolver este problema, em que dado um array de N elementos, podemos fazer as seguintes operações:
  - `update(i, v)`: atualizar a posição **`i`** somando o valor v
  - `query(i, j)`: consultar a soma entre a posição **`i`** e **`j`**

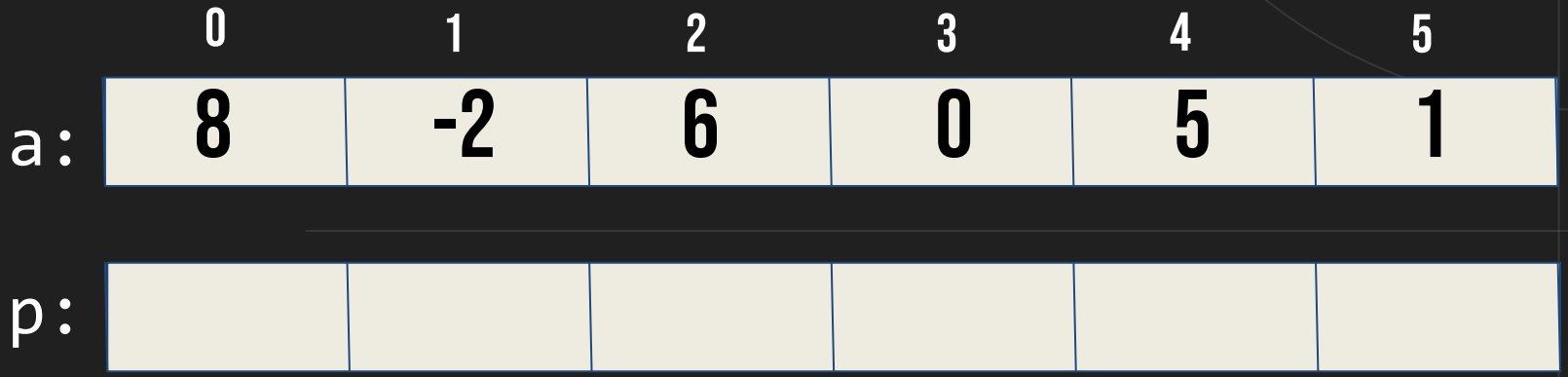
- Podemos resolver cada operação de formas simples, tendo como complexidade:
  - `update`: ***O(1)***
  - `query`: ***O(n)***

01 - MOTIVAÇÃO

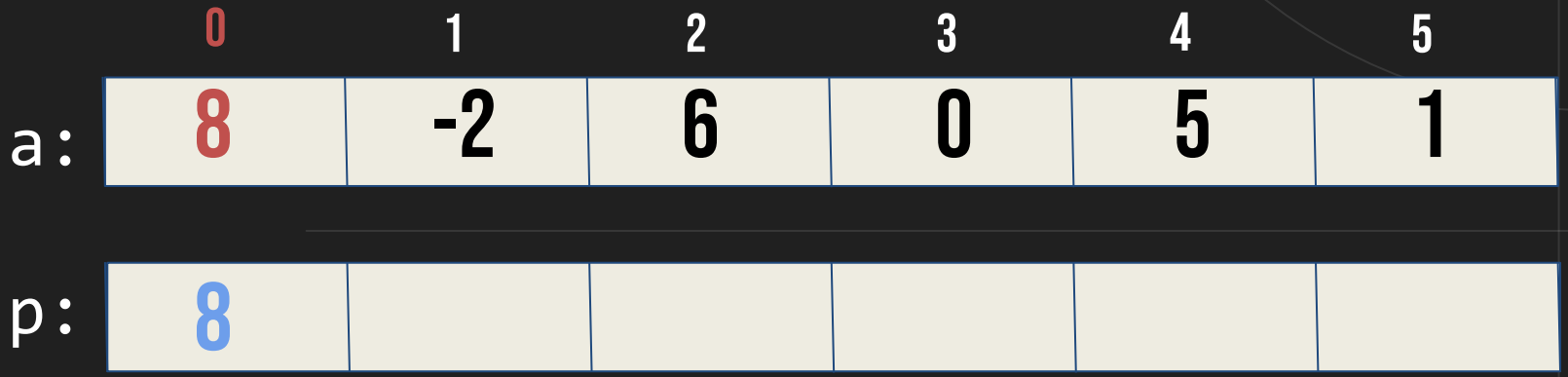
- Há outra forma de se resolver esse problema também usando estrutura de dados básicas. Como exemplo um array de prefixos.

- Vamos utilizar a soma de prefixos, em que:
  - $p[i] = \sum_{j=0}^{i} a[j]$

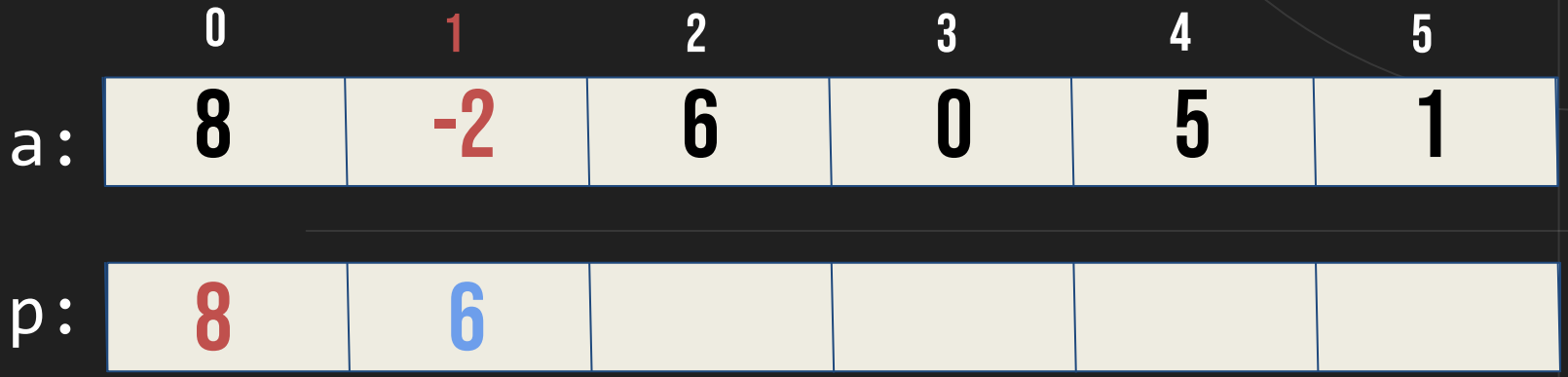
01 - MOTIVAÇÃO



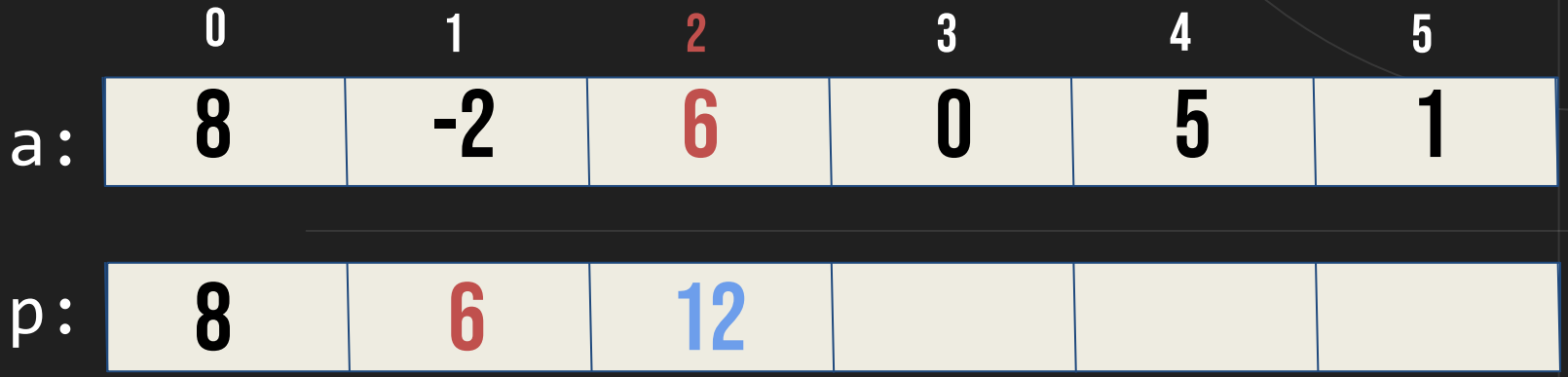
01 - MOTIVAÇÃO



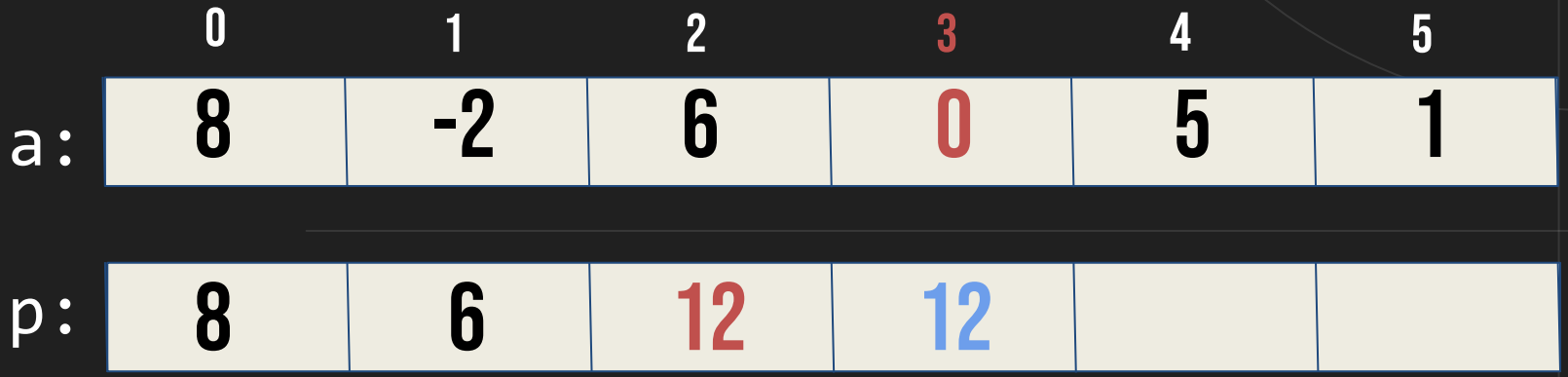
01 - MOTIVAÇÃO



01 - MOTIVAÇÃO



01 - MOTIVAÇÃO



01 - MOTIVAÇÃO

	0	1	2	3	4	5
a:	8	-2	6	0	5	1
p:	8	6	12	12	17	

01 - MOTIVAÇÃO

	0	1	2	3	4	5
a:	8	-2	6	0	5	1
p:	8	6	12	12	17	18

01 - MOTIVAÇÃO

	0	1	2	3	4	5
a:	8	-2	6	0	5	1
p:	8	6	12	12	17	18

01 - MOTIVAÇÃO

- Podemos fazer a consulta em ***O(1)*** simplesmente retornando **`p[j] - p[i-1]`**
  - `query(2, 4) = p[4] - p[1] = 11`

01 - MOTIVAÇÃO

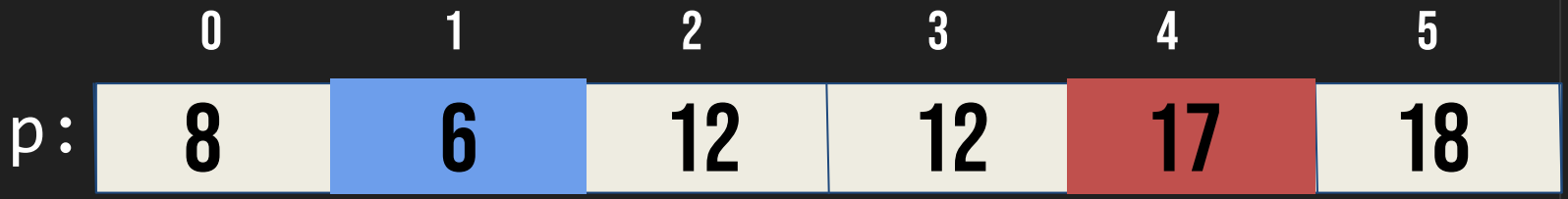
- Podemos fazer a consulta em ***O(1)*** simplesmente retornando **`p[j] - p[i-1]`**
  - `query(2, 4) = p[4] - p[1] = 11`

- Note que para o update vamos gastar ***O(n)*** pois teremos que refazer toda a soma de prefixos ao array

01 - MOTIVAÇÃO

- Podemos fazer a consulta em ***O(1)*** simplesmente retornando **`p[j] - p[i-1]`**
  - `query(2, 4) = p[4] - p[1] = 11`

- Note que para o update vamos gastar ***O(n)*** pois teremos que refazer toda a soma de prefixos ao array

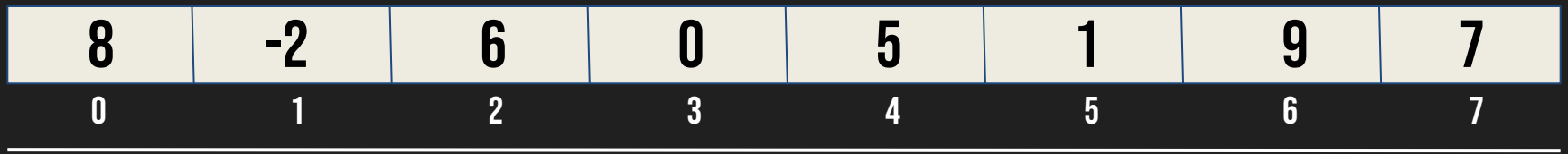


02 - SEGTREE

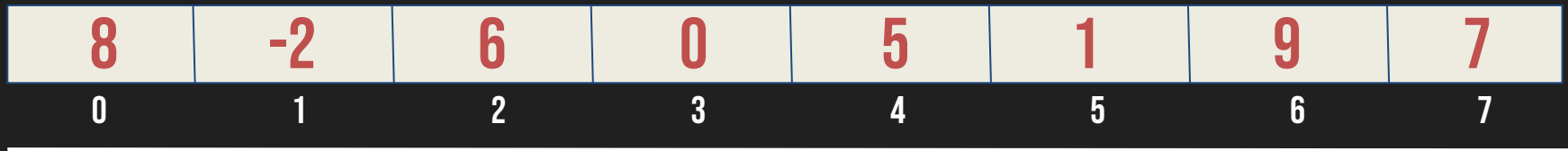
- Queremos uma estrutura que seja eficiente tanto para a operação de `query()` como para `update()`. Assim, podemos usar a ideia de uma Árvore Binária para particionar o array em intervalos.

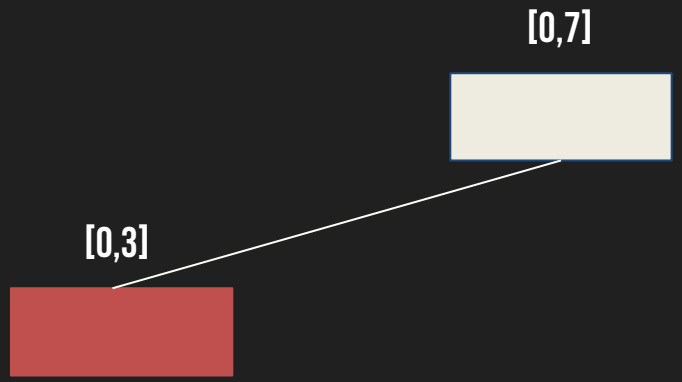
- A Segment Tree divide o array em intervalos ou segmentos, na qual cada nó da árvore representa um intervalo do array e armazena a resposta (como soma ou máx/min) para aquele intervalo.

- A raiz da árvore representa todo array e suas folhas representam os elementos individuais.

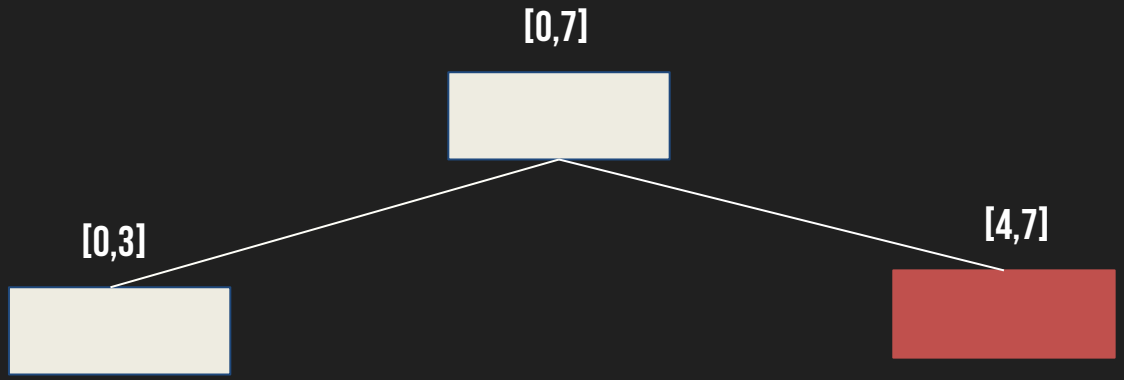

8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

[0,7]

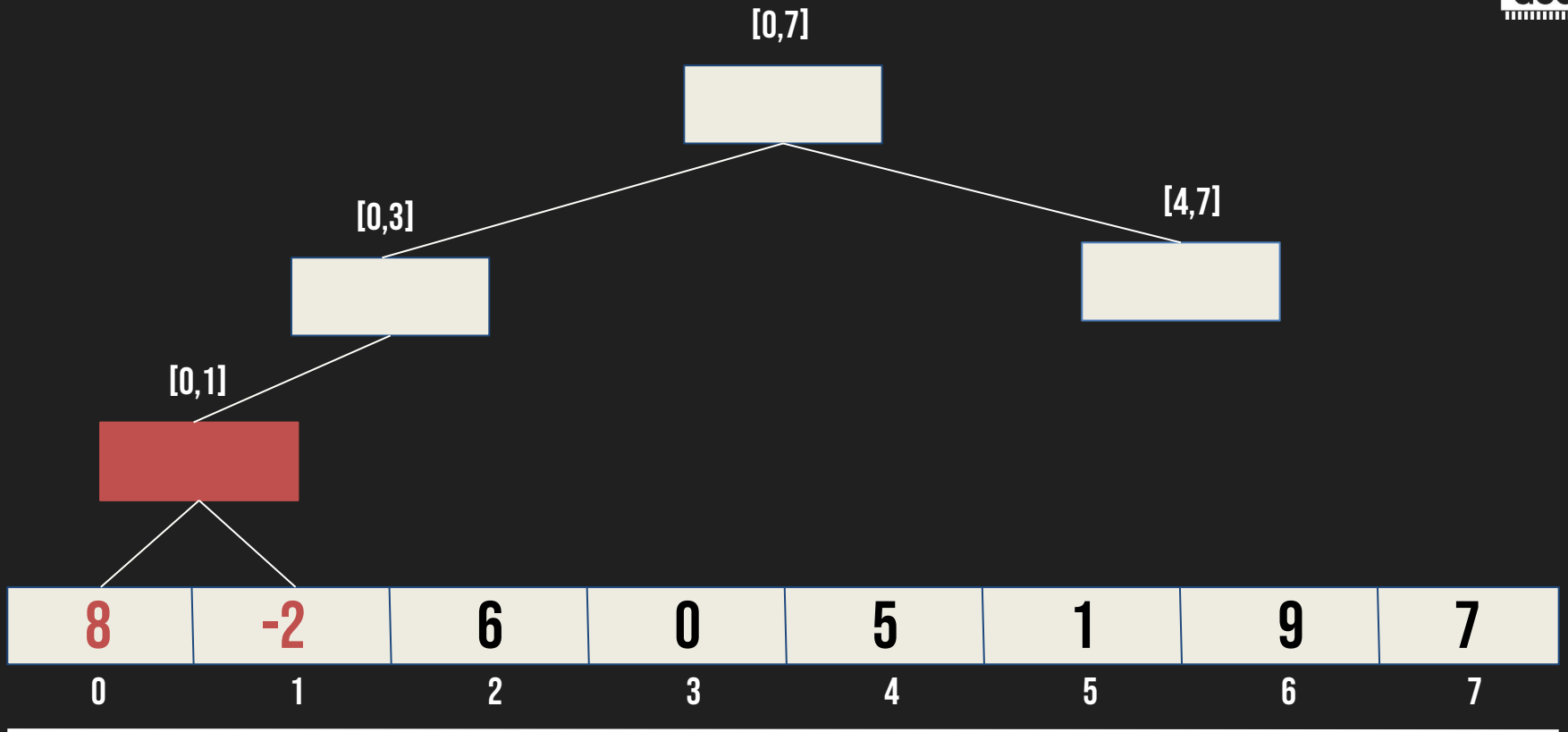


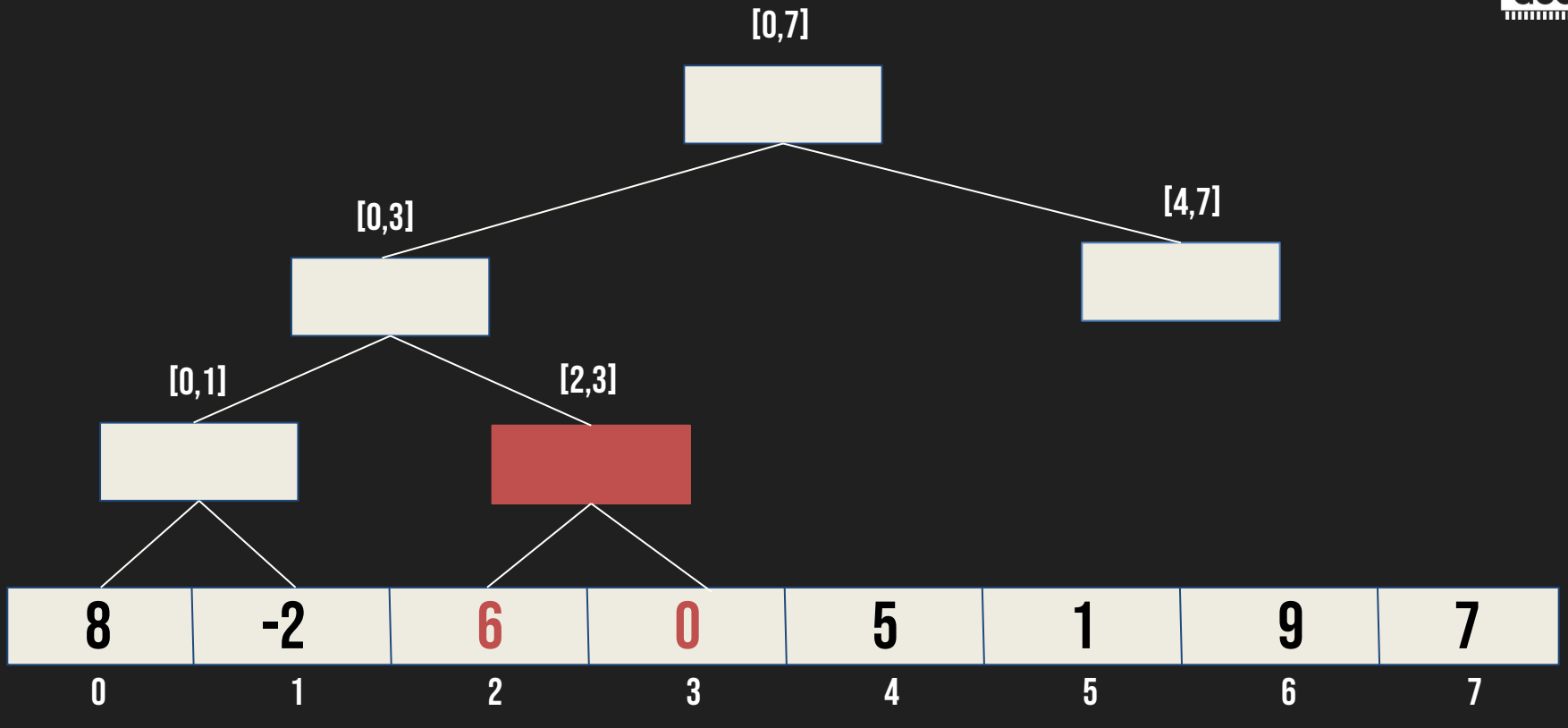


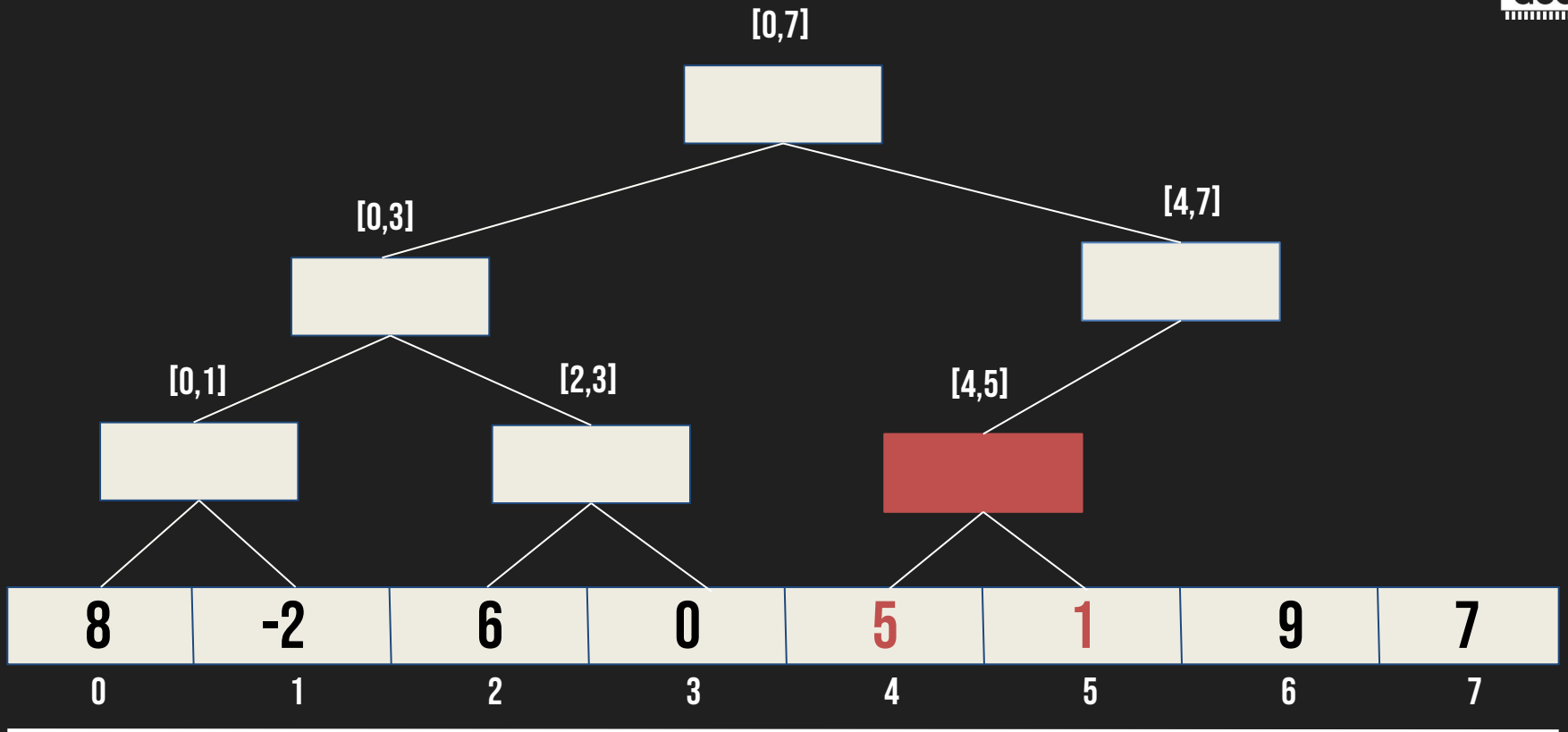
8	-2	6	0	5	1	9	7
0	1	2	3	4	5	6	7

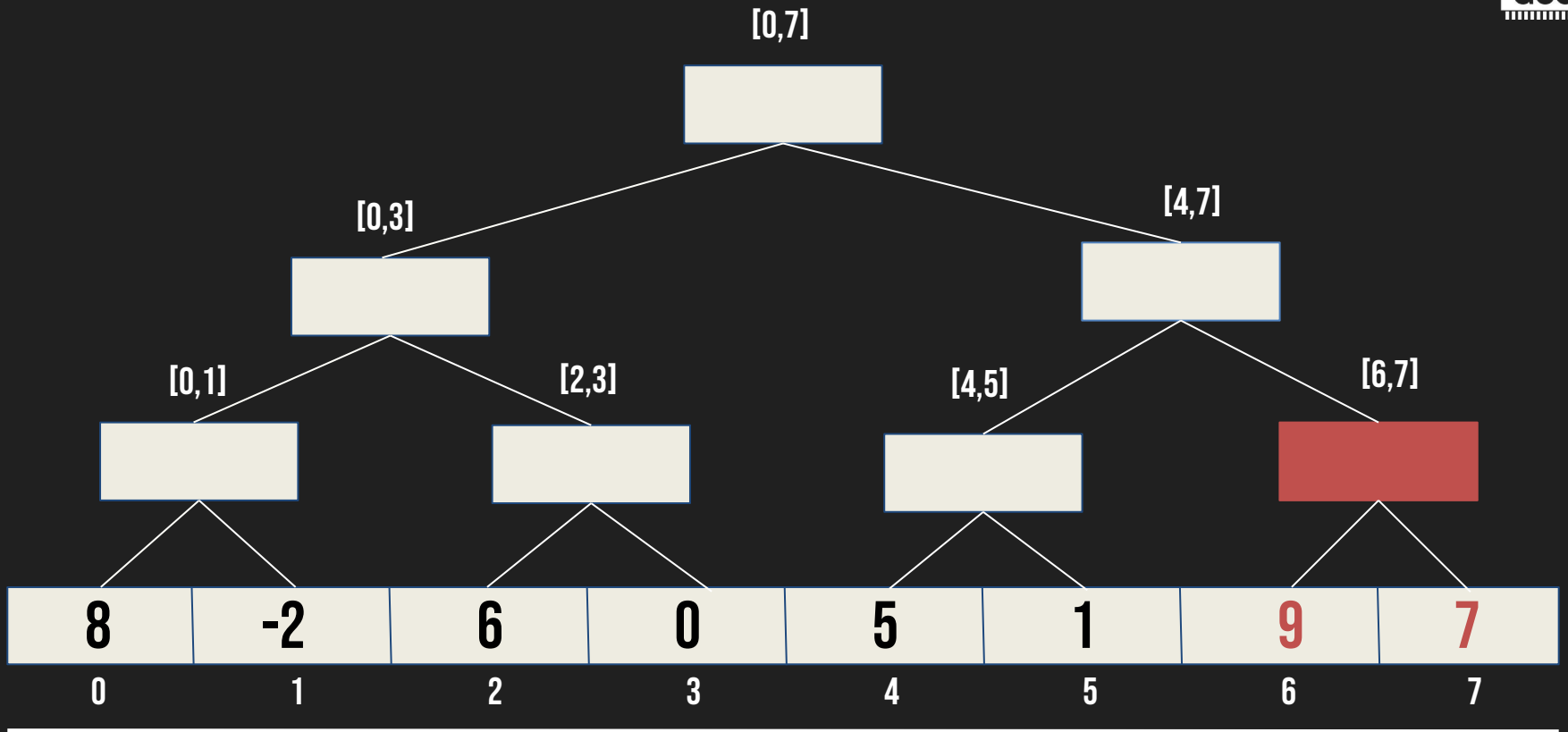


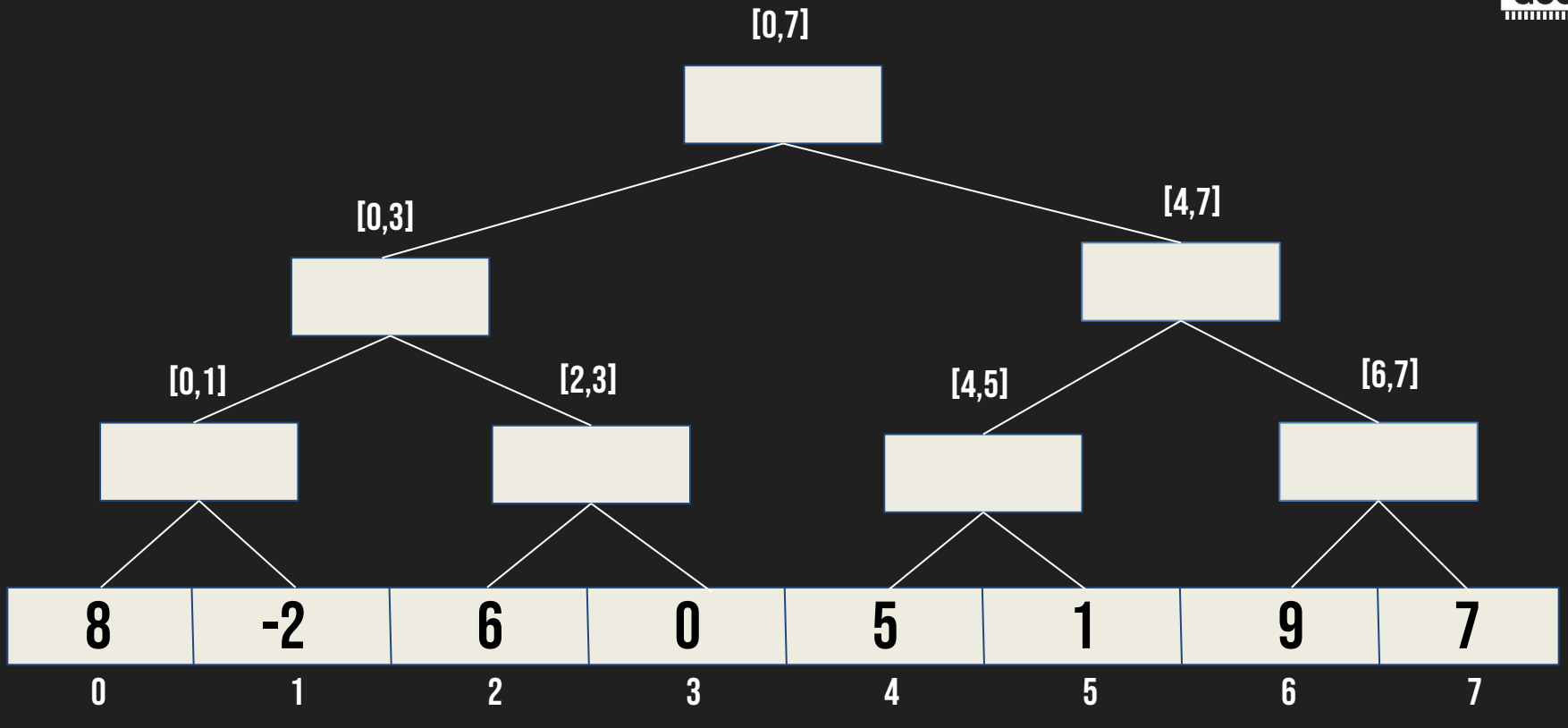
8	-2	6	0	5	1	9	7
0	1	2	3	4	5	6	7

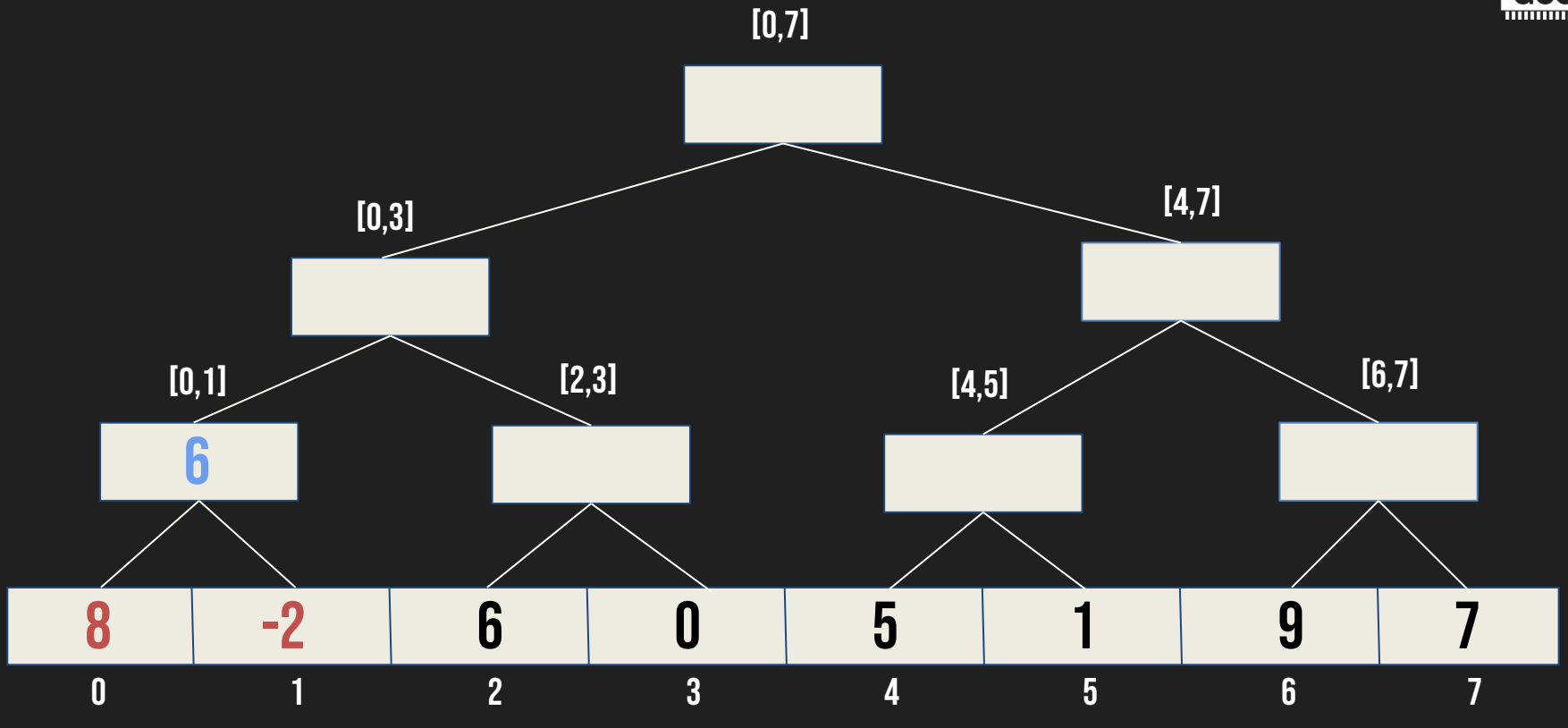

[0,7]
[0,3]
[4,7]
[0,1]
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

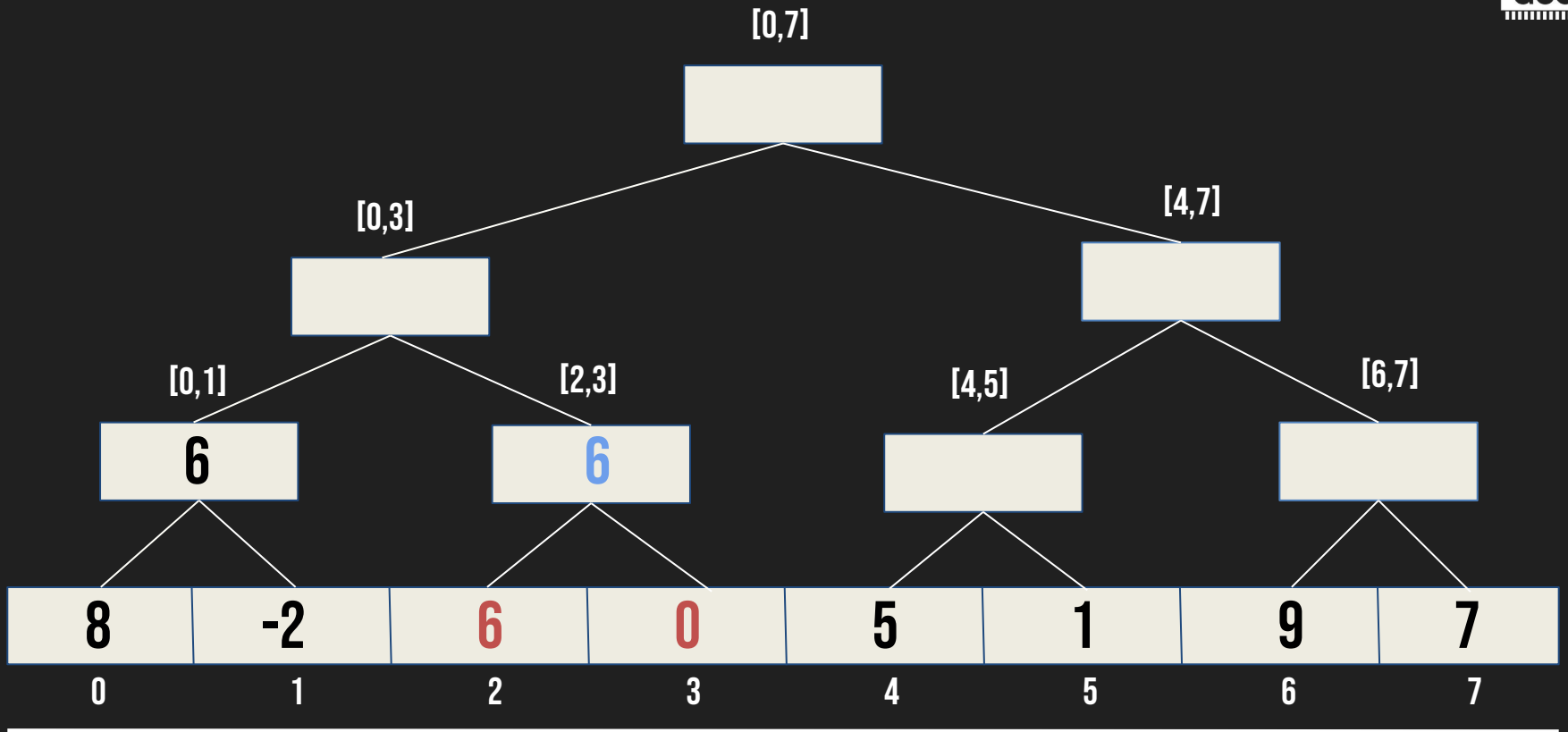

[0,7]
[0,3]
[4,7]
[0,1]
[2,3]
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

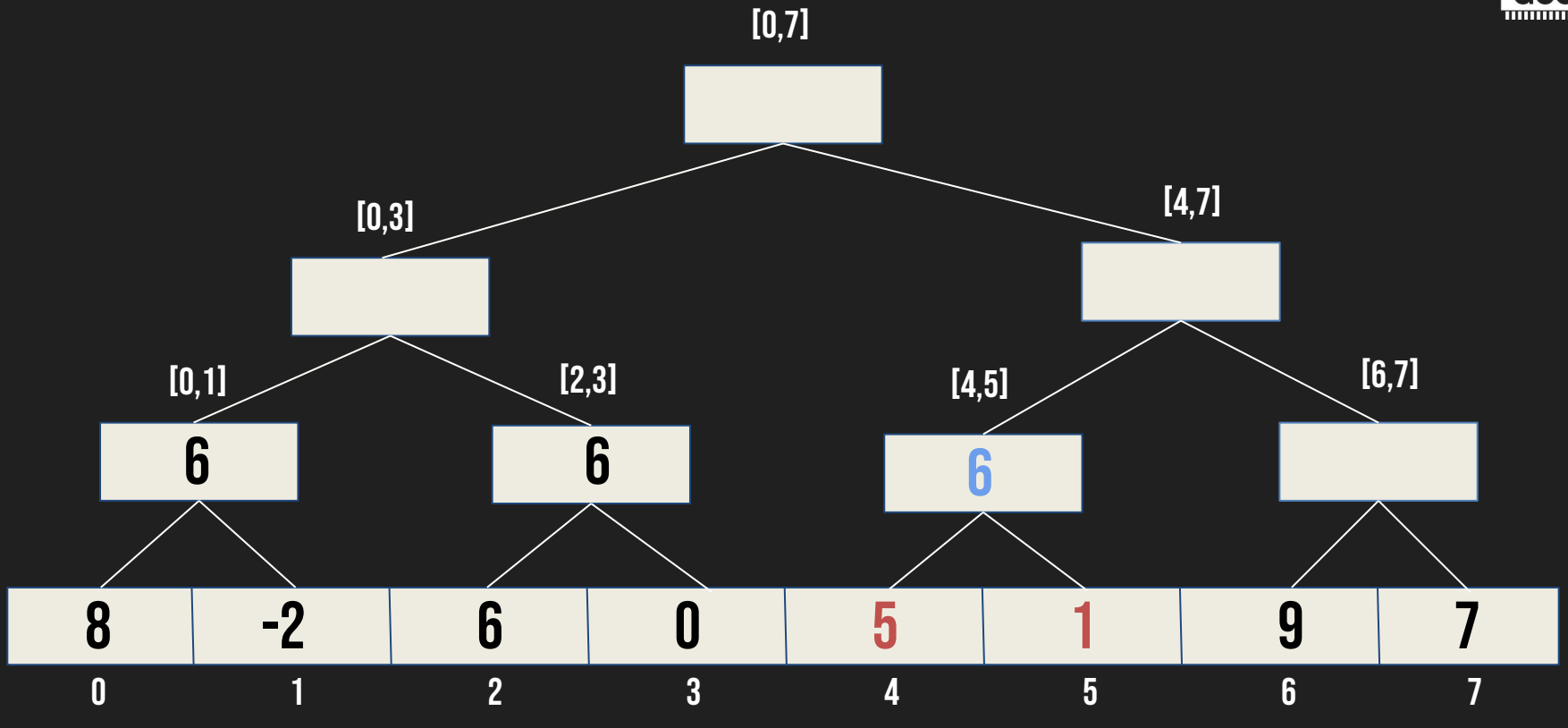

[0,7]
[0,3]
[4,7]
[0,1]
[2,3]
[4,5]
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

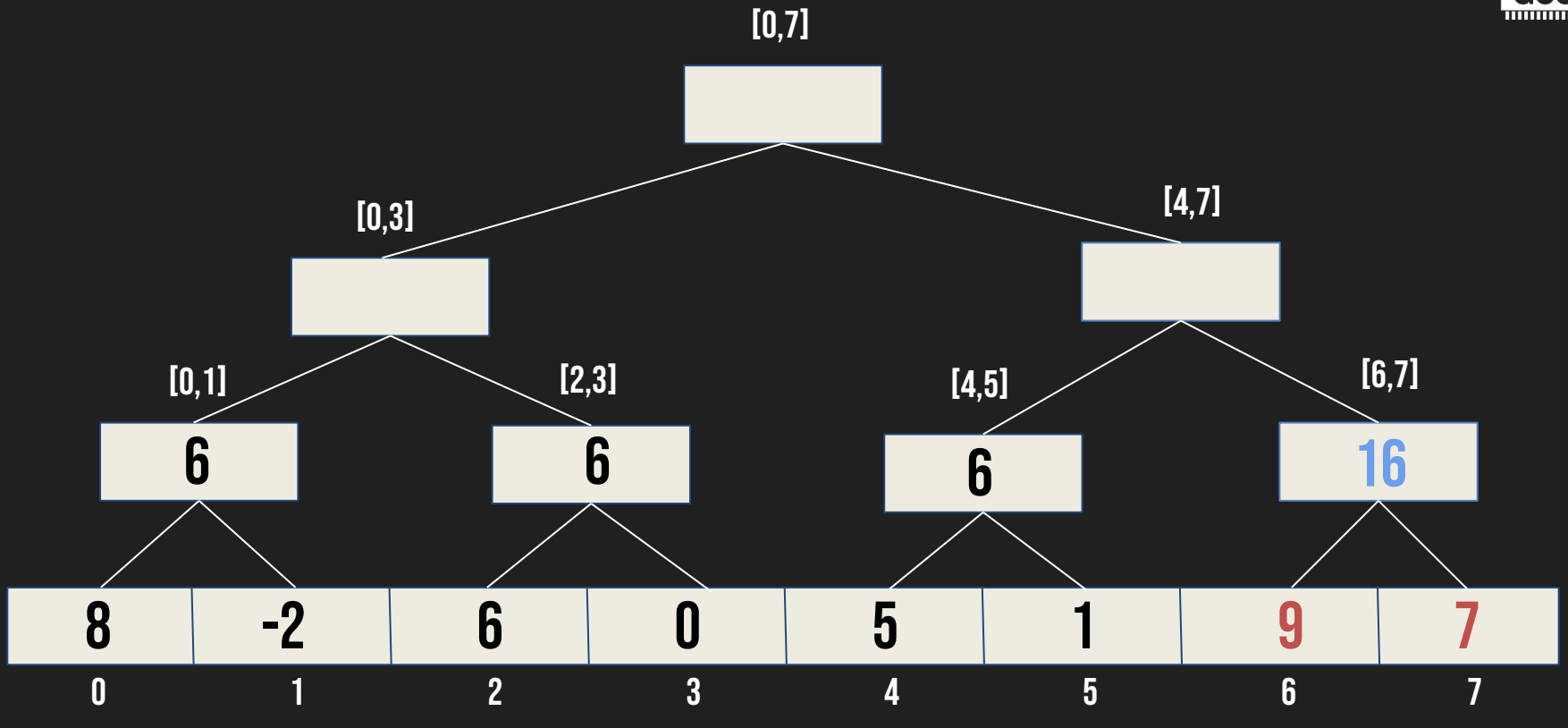

[0,7]
[0,3]
[4,7]
[0,1]
[2,3]
[4,5]
[6,7]
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

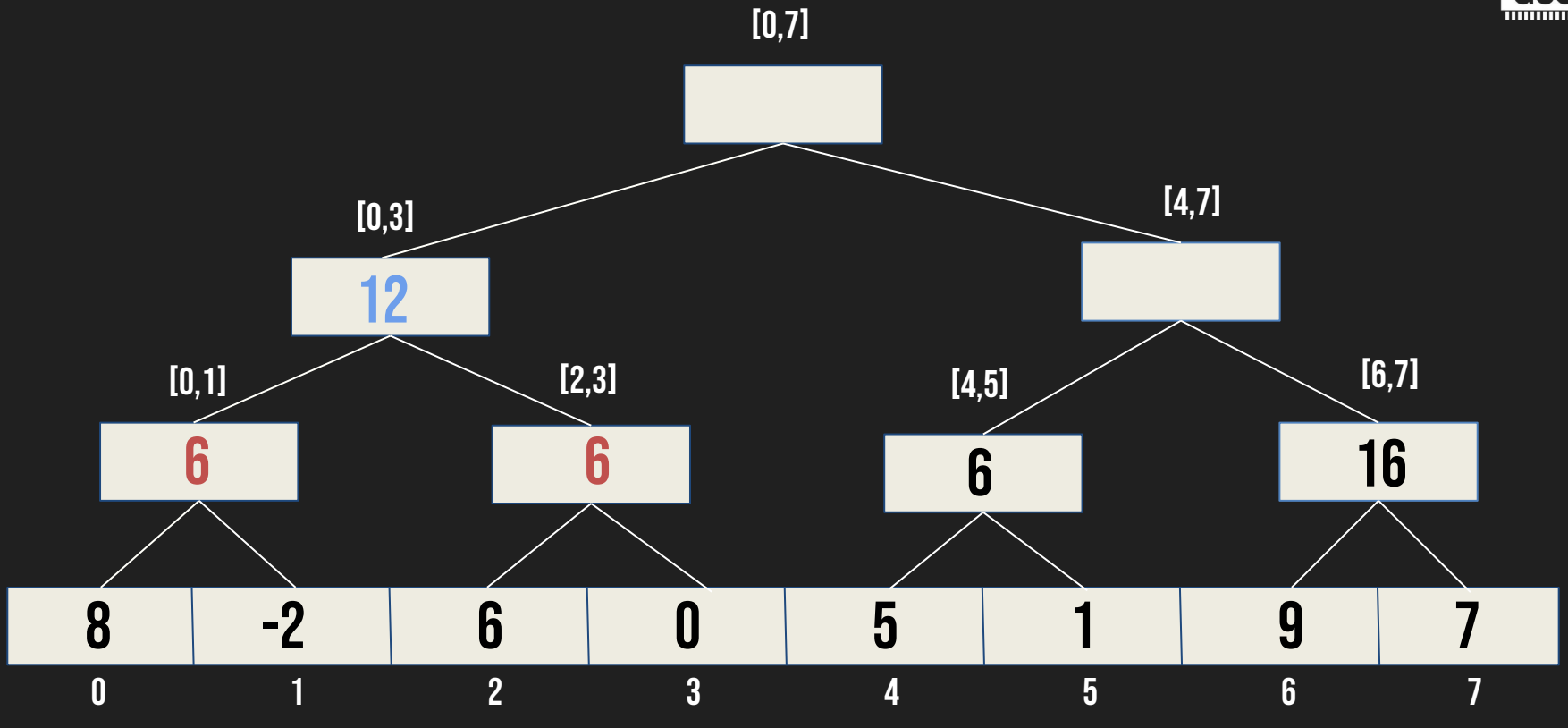

[0,7]
[0,3]
[4,7]
[0,1]
[2,3]
[4,5]
[6,7]
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

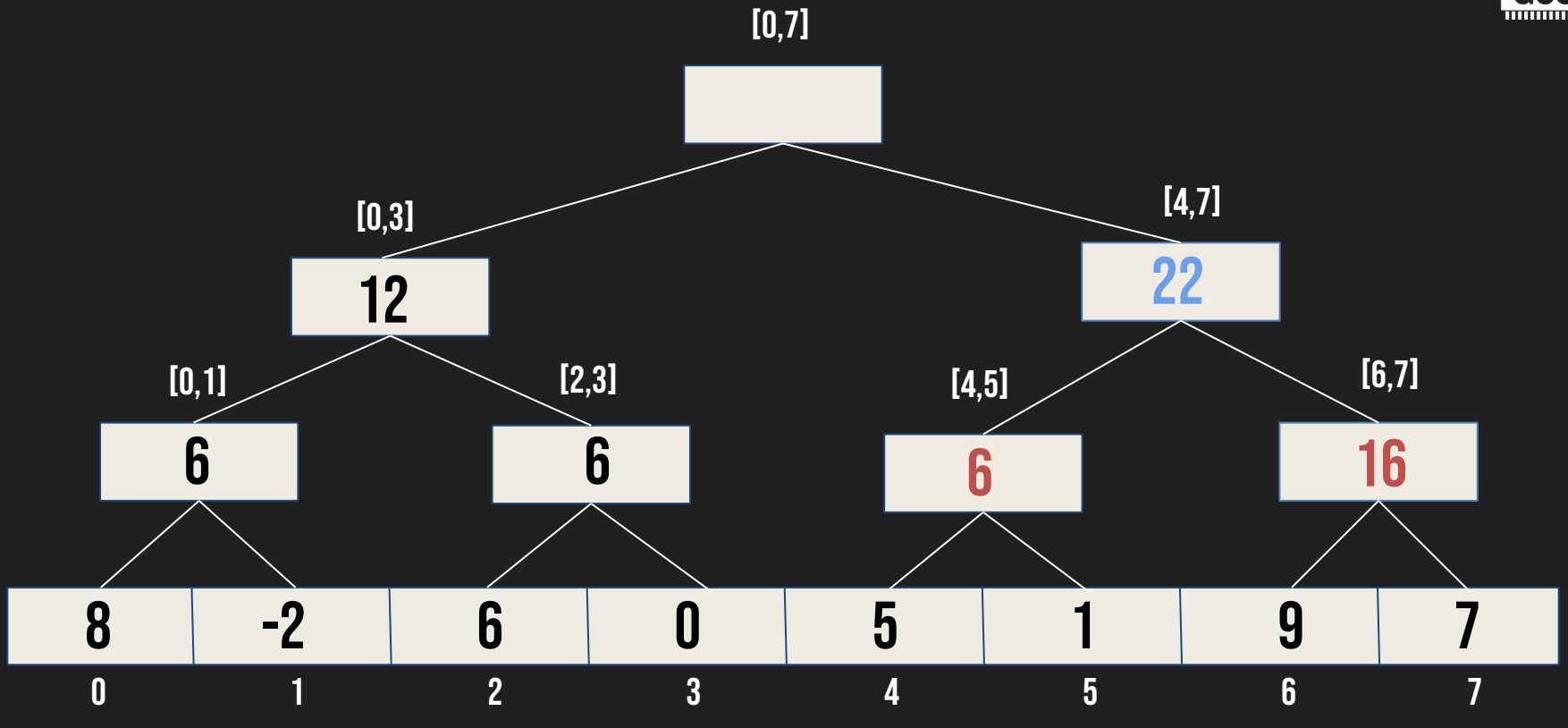

[0,7]
[0,3]
[4,7]
[0,1]
[2,3]
[4,5]
[6,7]
6
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

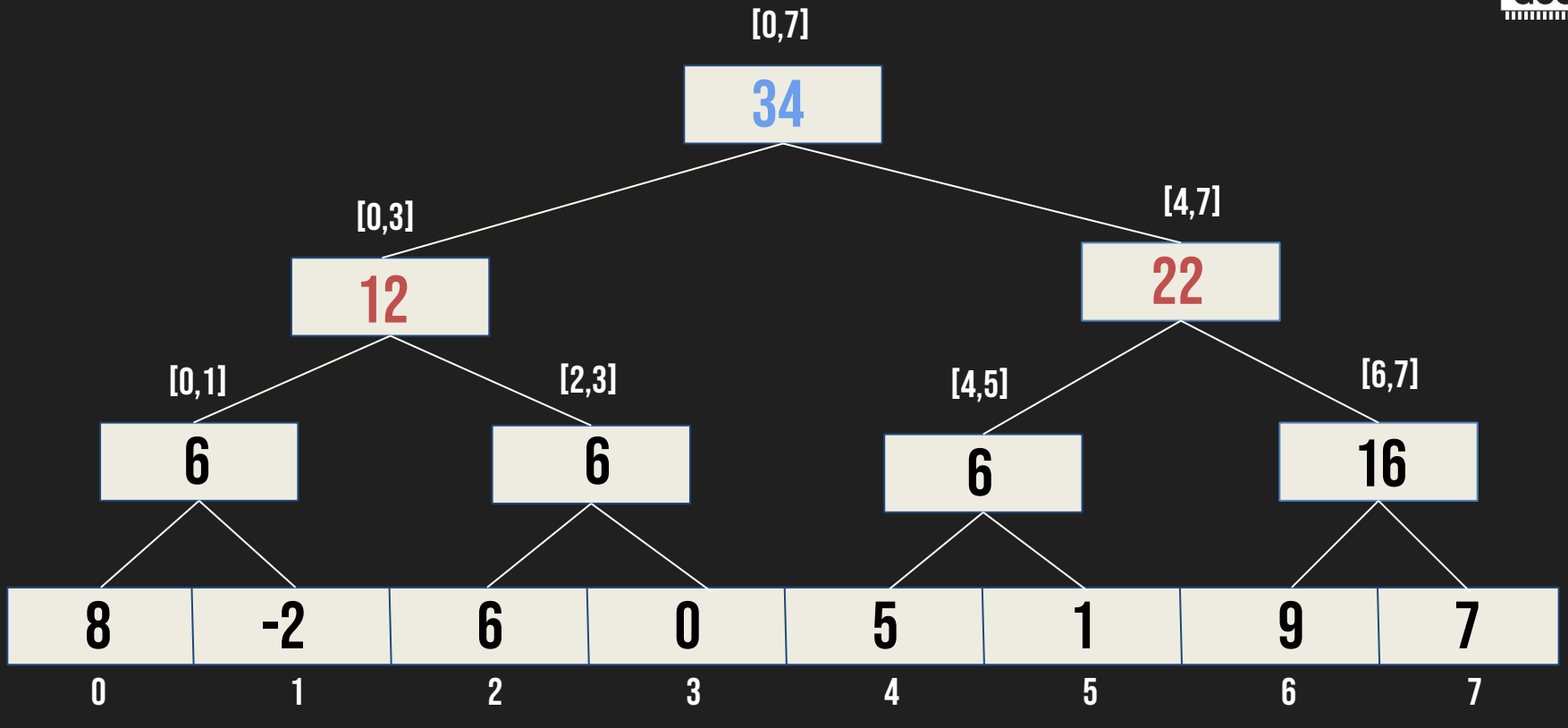

[0,7]
[0,3]
[4,7]
[0,1]
[2,3]
[4,5]
[6,7]
6
6
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

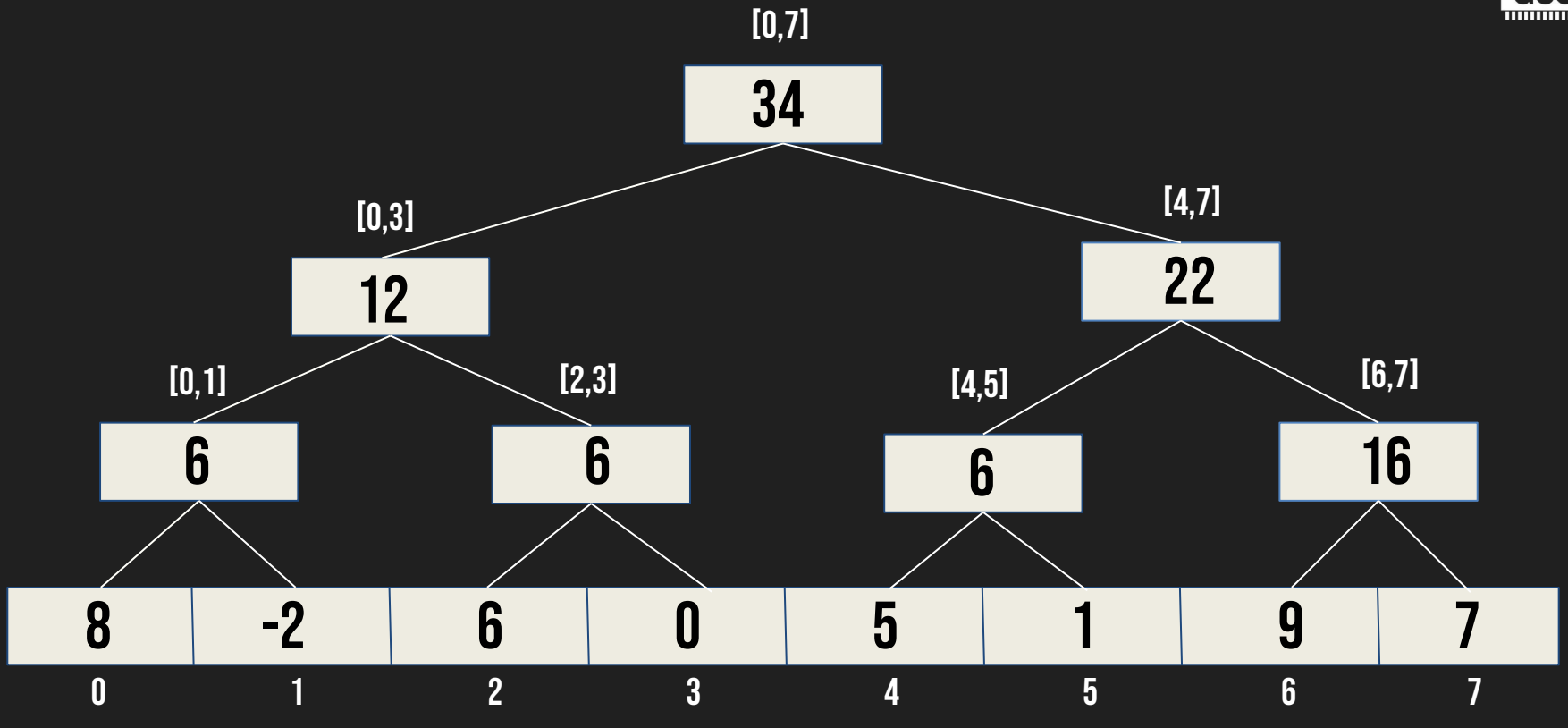

[0,7]
[0,3]
[4,7]
[0,1]
[2,3]
[4,5]
[6,7]
6
6
6
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7


[0,7]
[0,3]
[4,7]
[0,1]
[2,3]
[4,5]
[6,7]
6
6
6
16
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7


[0,7]
[0,3]
12
[4,7]
[0,1]
6
[2,3]
6
[4,5]
6
[6,7]
16
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

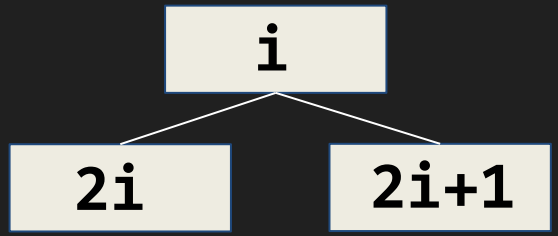

[0,7]
[0,3]
12
[4,7]
22
[0,1]
6
[2,3]
6
[4,5]
6
[6,7]
16
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7


[0,7]
34
[0,3]
12
[4,7]
22
[0,1]
6
[2,3]
6
[4,5]
6
[6,7]
16
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7


[0,7]
34
[0,3]
12
[4,7]
22
[0,1]
6
[2,3]
6
[4,5]
6
[6,7]
16
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

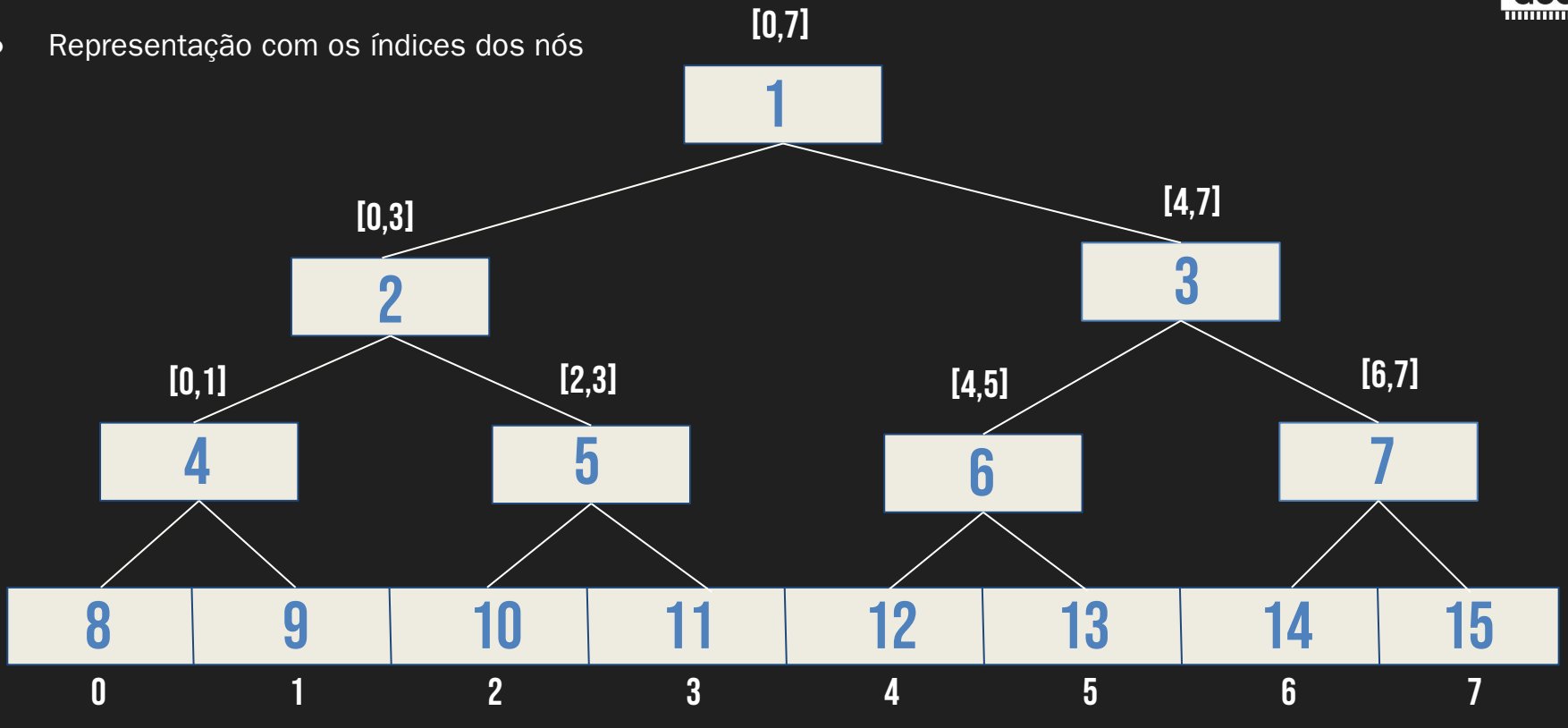
03 - CONSTRUÇÃO

- Cada nó terá seu índice **i**, e seu filho da esquerda terá índice **2i** e o da direita **2i+1**



- Sua raiz será indexada no índice 1, e seguimos nível a nível, numerando da esquerda para a direita

- Representação com os índices dos nós



03 - CONSTRUÇÃO

- Podemos construir a Segment Tree em um array, onde cada posição `i` representa o nó `i`.

- Uma árvore binária completa com **N folhas possui exatamente `2N-1` nós e altura `logN+1`**

    - No exemplo anterior temos uma árvore binária completa, ela possui 8 folhas, 15 nós e altura 4.

    - Quando N é uma potência de 2, os índices dos nós vão até `2N-1`, porém, quando N não é uma potência de 2, os índices dos nós podem passar de `2N-1`

    - Por isso é comum ver **`4*N`** como limite do índice de nós, e este será o tamanho do array em que iremos construir a SegTree.

04 - ALGORITMO

04 - ALGORITMO

```
int a[MAXN];
int tree[4*MAXN];
```

04 - ALGORITMO

```
int a[MAXN];
int tree[4*MAXN];

int build(int node, int l, int r){



}
```

04 - ALGORITMO

```
int a[MAXN];
int tree[4*MAXN];

int build(int node, int l, int r){
    if(l == r) return tree[node] = a[l];


}
```

04 - ALGORITMO

```
int a[MAXN];
int tree[4*MAXN];

int build(int node, int l, int r){
    if(l == r) return tree[node] = a[l];
    int m = (l+r)/2;

}
```

04 - ALGORITMO

```
int a[MAXN];
int tree[4*MAXN];

int build(int node, int l, int r){
    if(l == r) return tree[node] = a[l];
    int m = (l+r)/2;
    return tree[node] = build(2*node, l, m) + build(2*node+1, m+1, r);
}
```

04 - ALGORITMO

```
int a[MAXN];
int tree[4*MAXN];

int build(int node, int l, int r){
    if(l == r) return tree[node] = a[l];
    int m = (l+r)/2;
    return tree[node] = build(2*node, l, m) + build(2*node+1, m+1, r);
}

build(1, 0, n-1); //gastamos complexidade O(n) para construir
```

05 - CONSULTA

- Dado um intervalo de `a` até `b`, queremos saber a soma de todos os elementos do array com posições de `a` até `b`.

- Iremos fazer do mesmo modo que a construção, calculando recursivamente

05 - CONSULTA

- Se o intervalo do nó que estamos estiver dentro do intervalo `a` até `b`, retornaremos o valor daquele nó: `tree[node]`

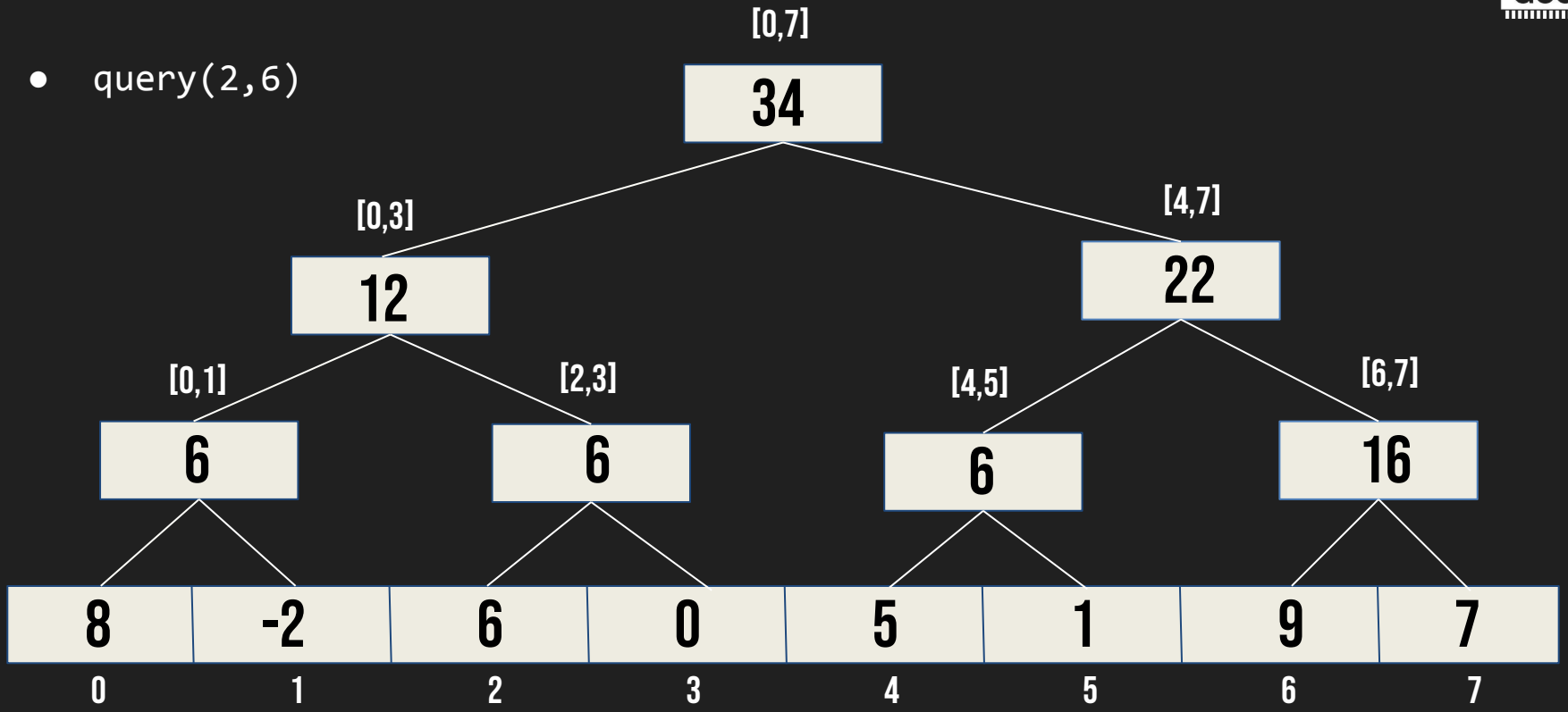
- Caso o intervalo do nó esteja totalmente fora do intervalo [a...b], retornaremos o valor nulo da operação (variando do tipo de operação em que você queira), em que esse valor nulo não altere a operação
  - Para Range Sum Query é 0
  - Para Range Max/Min Query é -INF/INF

05 - CONSULTA

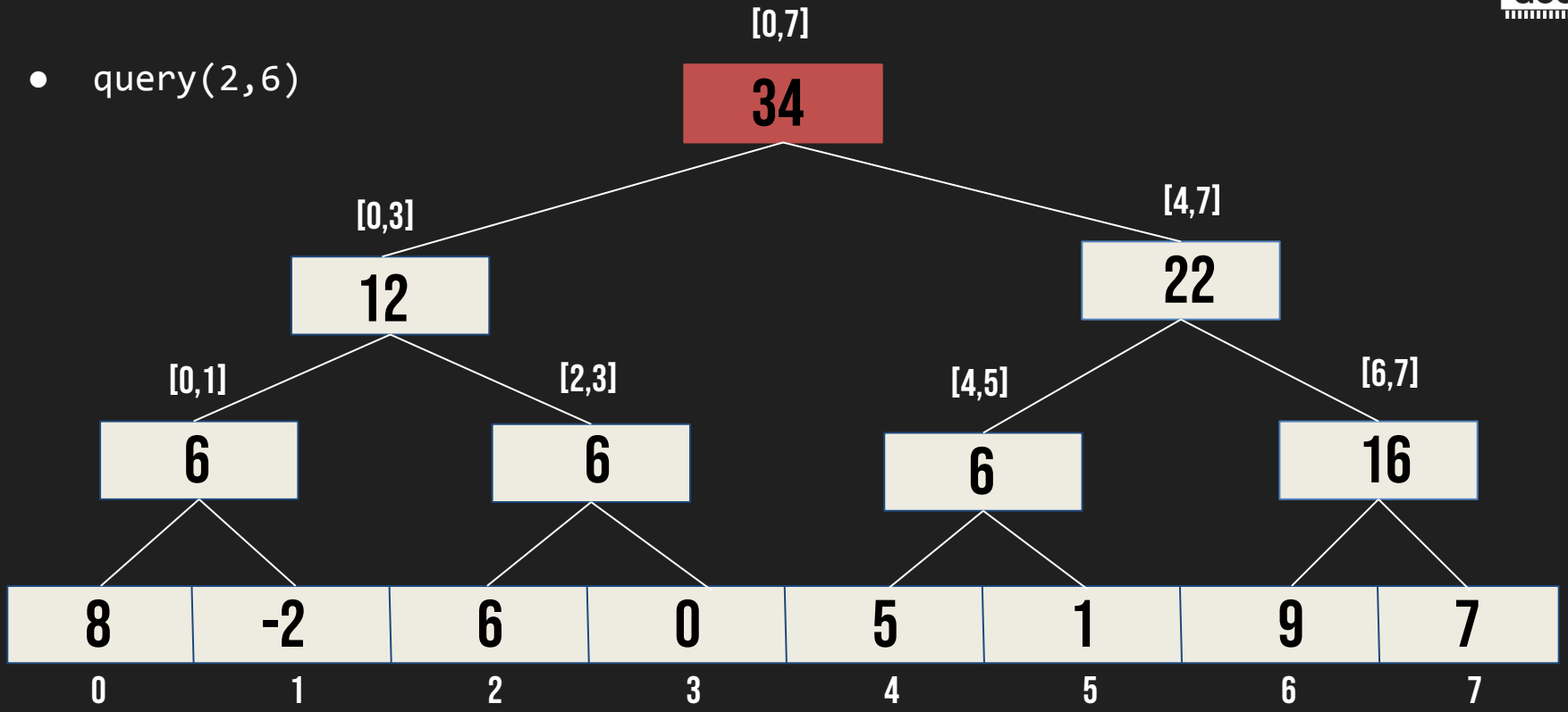
- Caso nenhuma das duas condições mostrada seja satisfeita, vamos continuar a procurar nos filhos do nó atual, e retornaremos a operação entre eles (que em nosso caso é a soma)

- Vale ressaltar que, a resposta final será a soma de um conjunto de nós da SegTree que inteiramente fazem parte do intervalo de a até b.

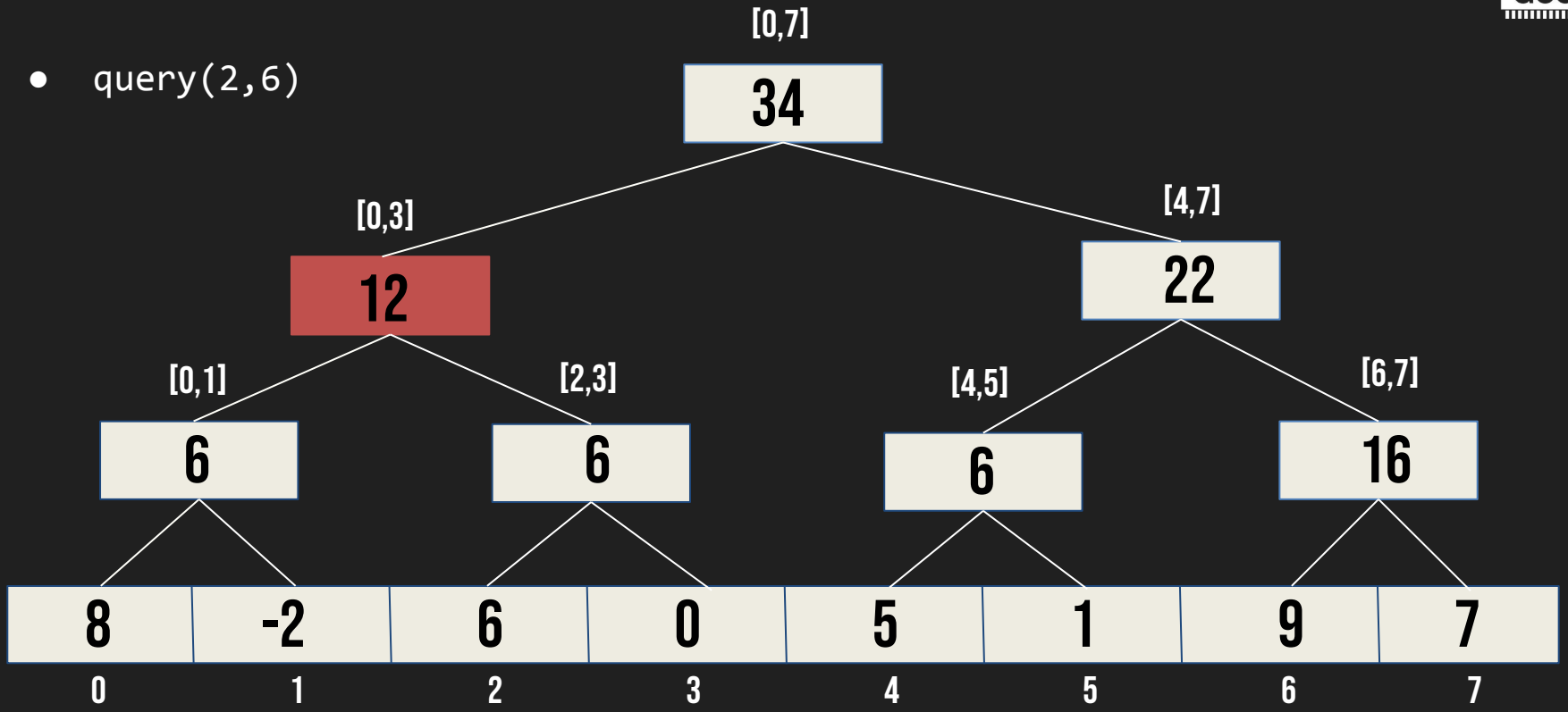
- query(2,6)



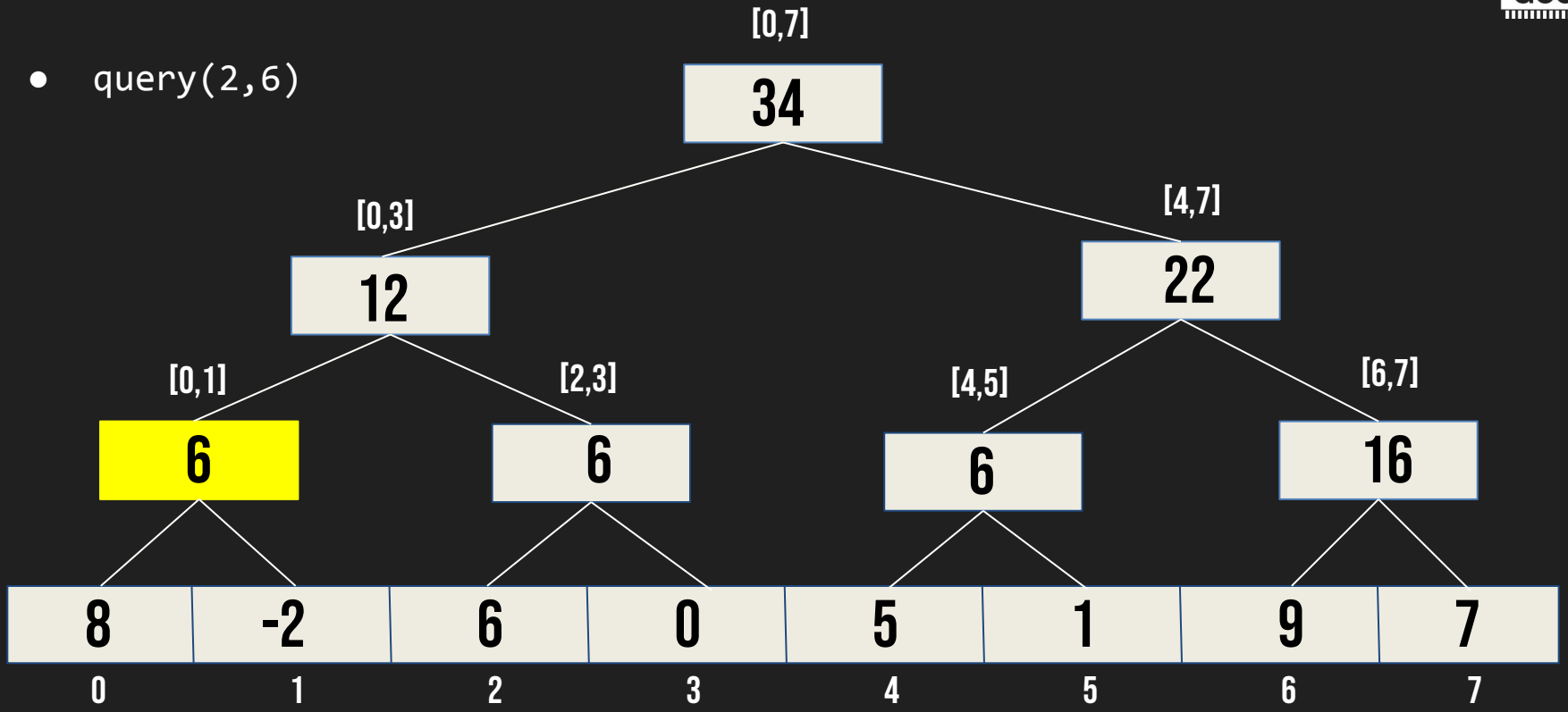
- query(2,6)



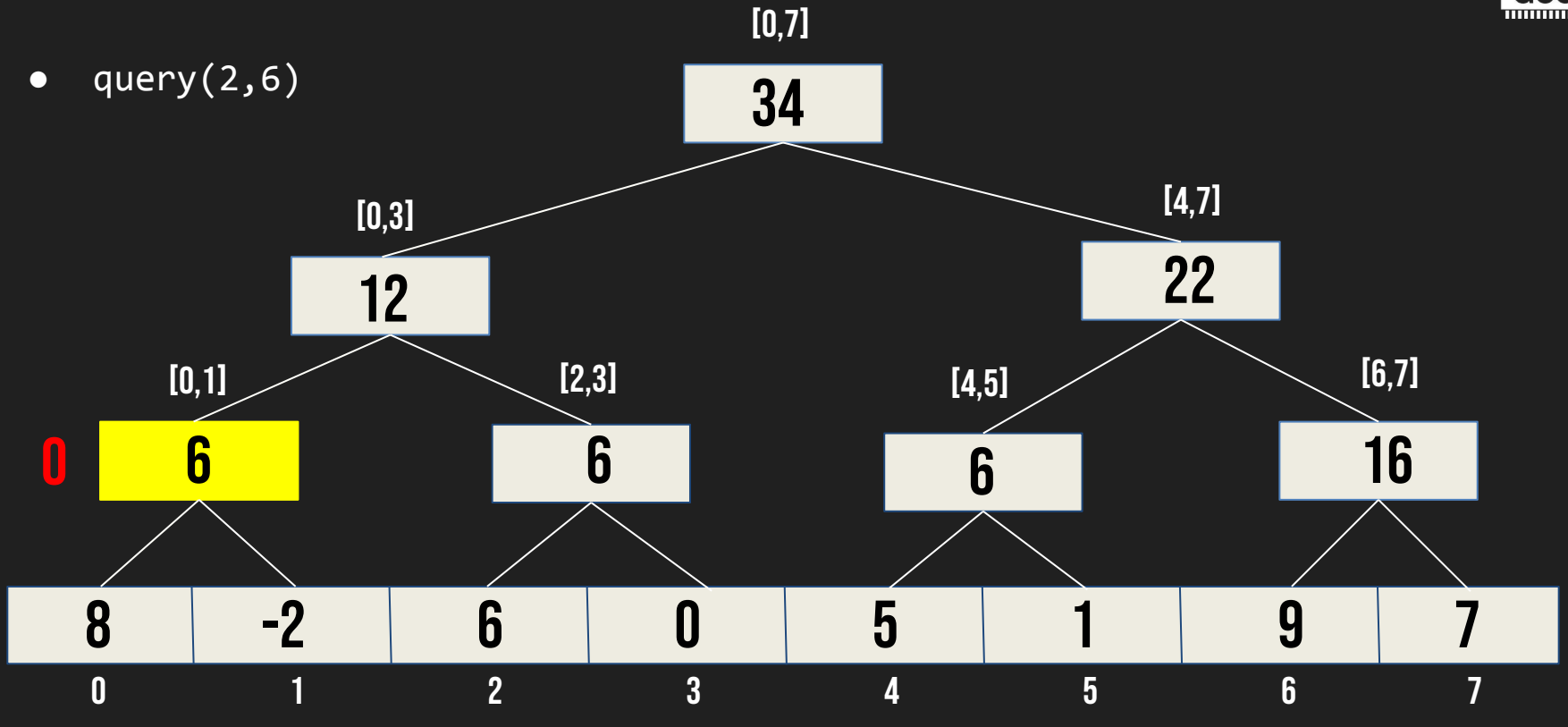
- query(2,6)



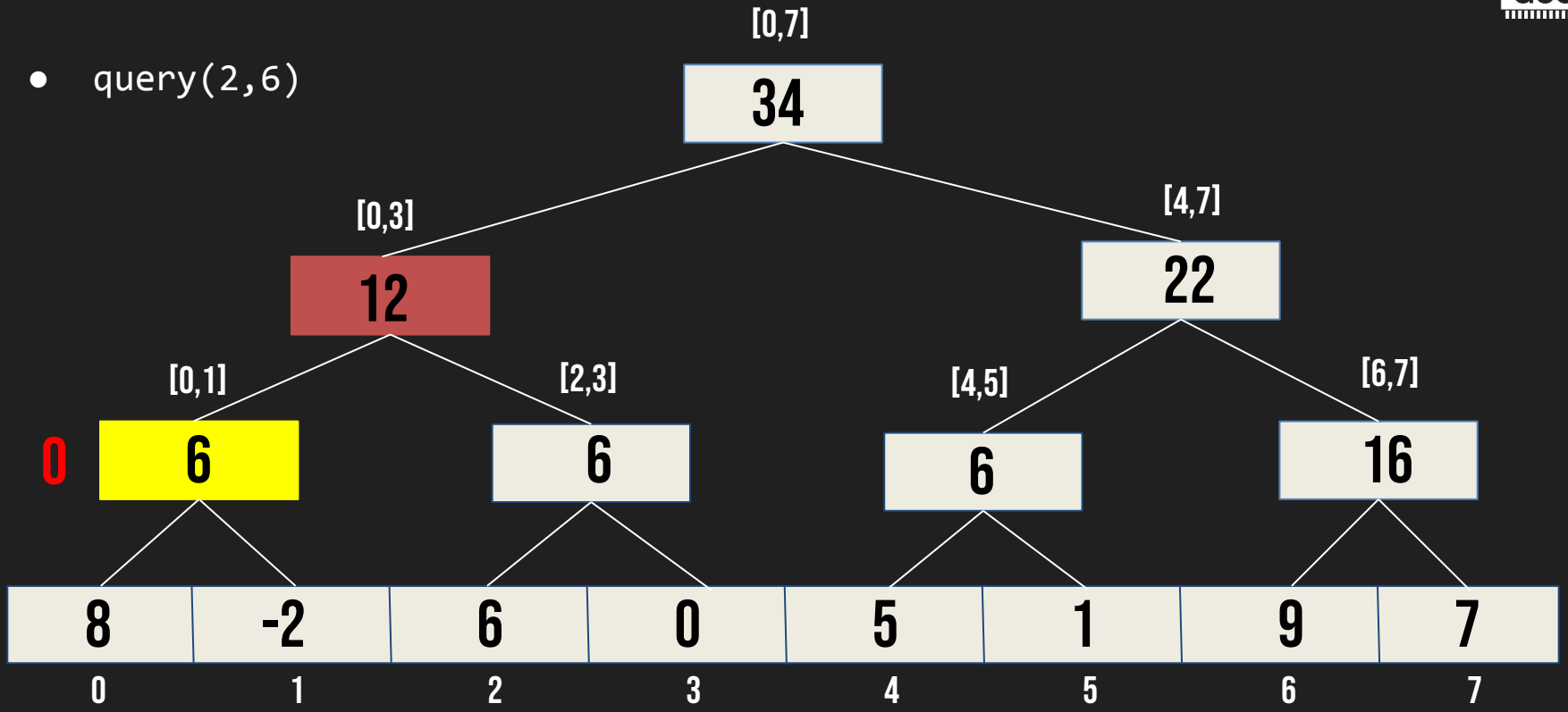
- query(2,6)



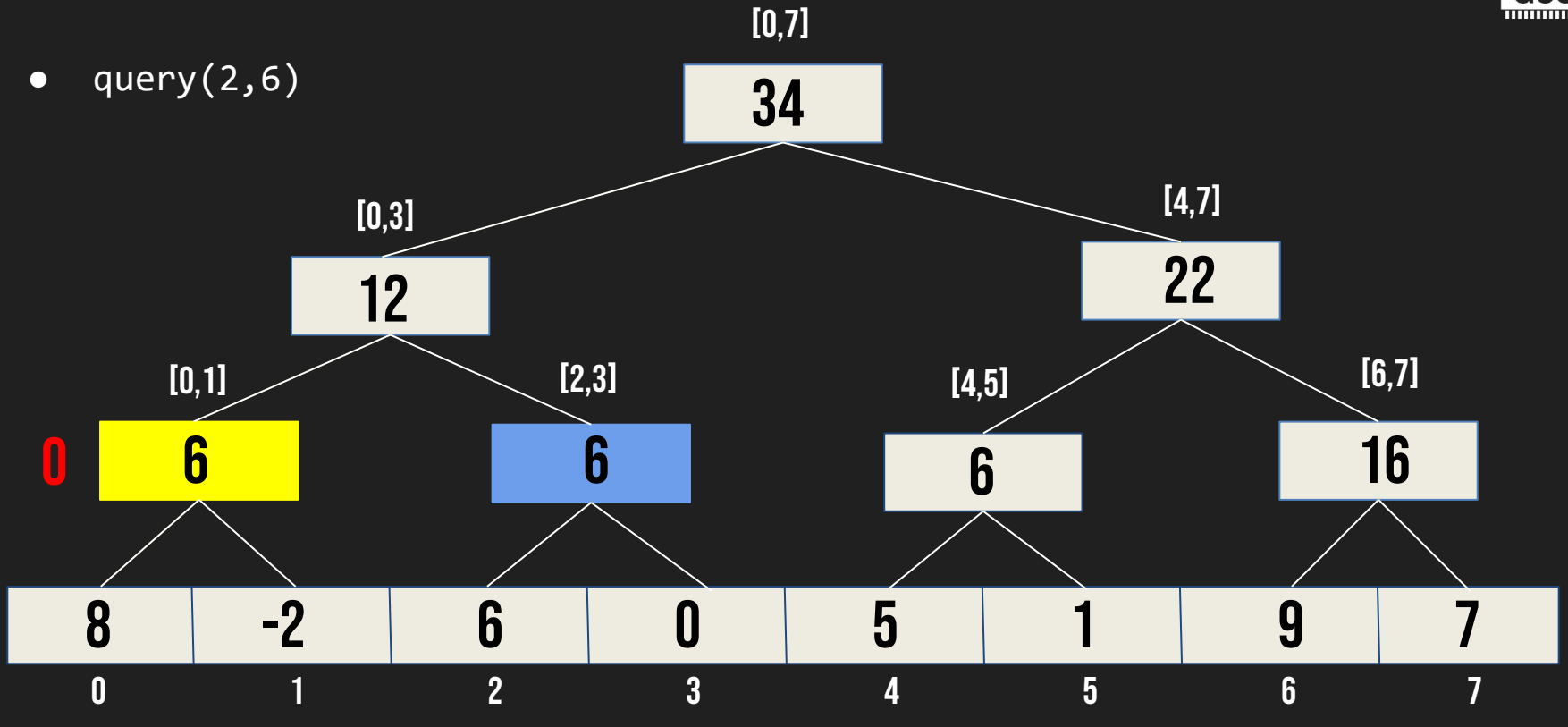
- query(2,6)



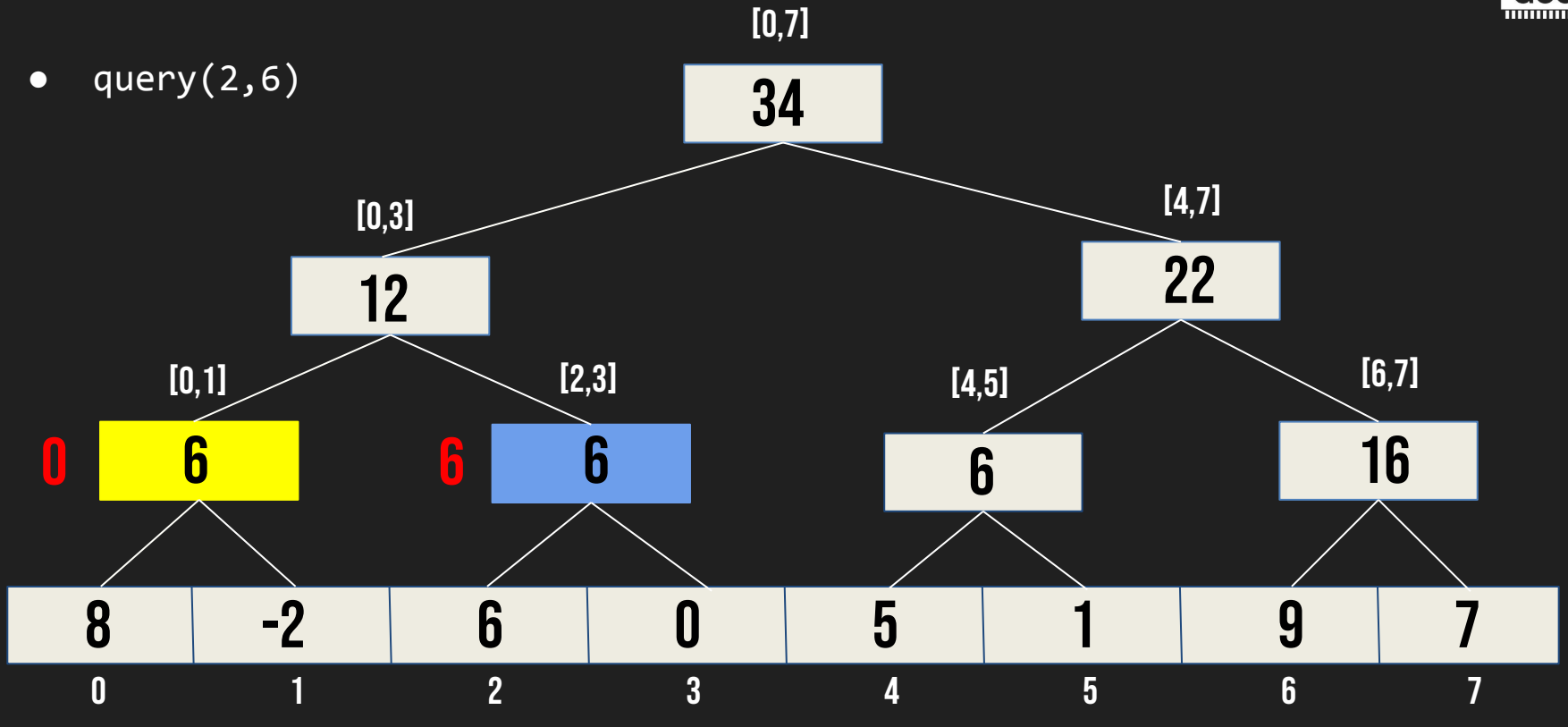
- query(2,6)



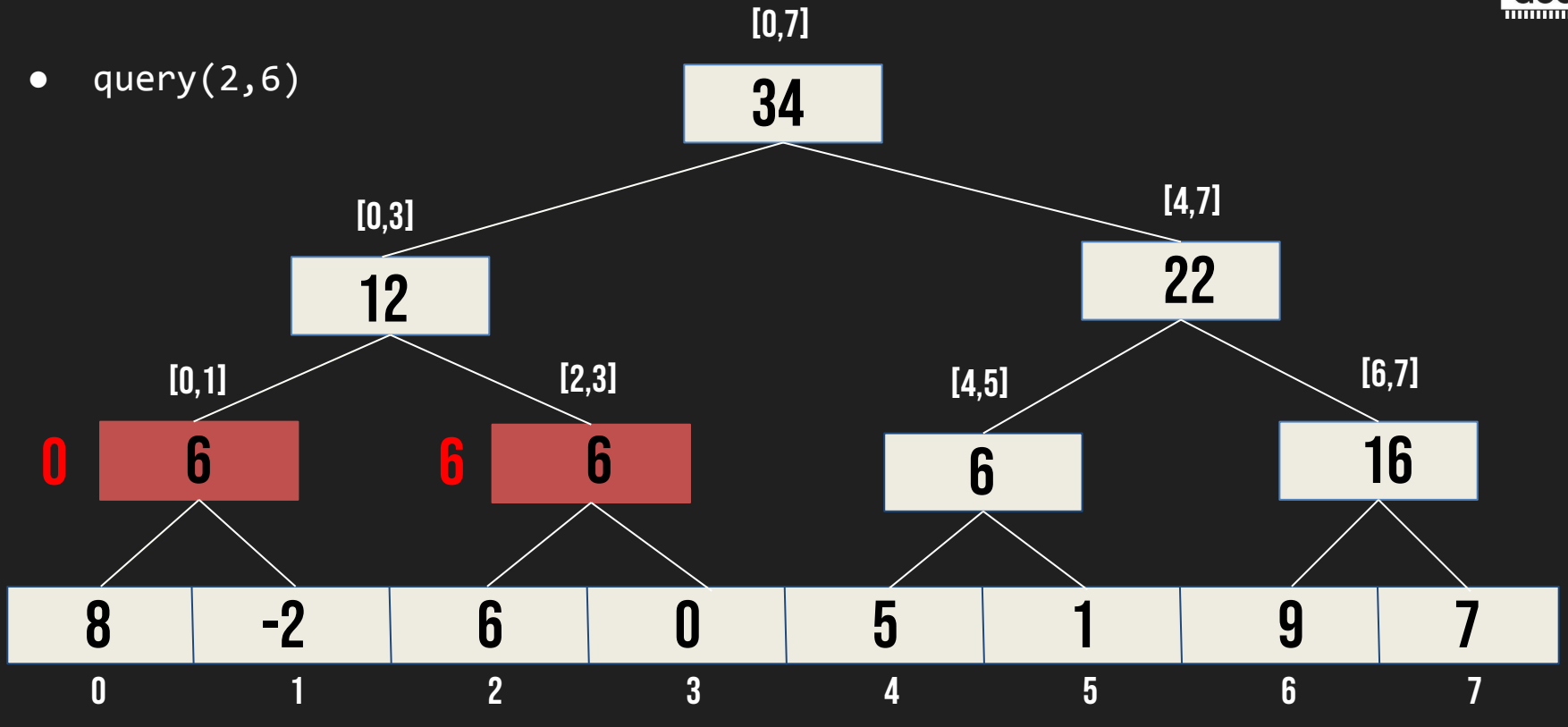
- query(2,6)



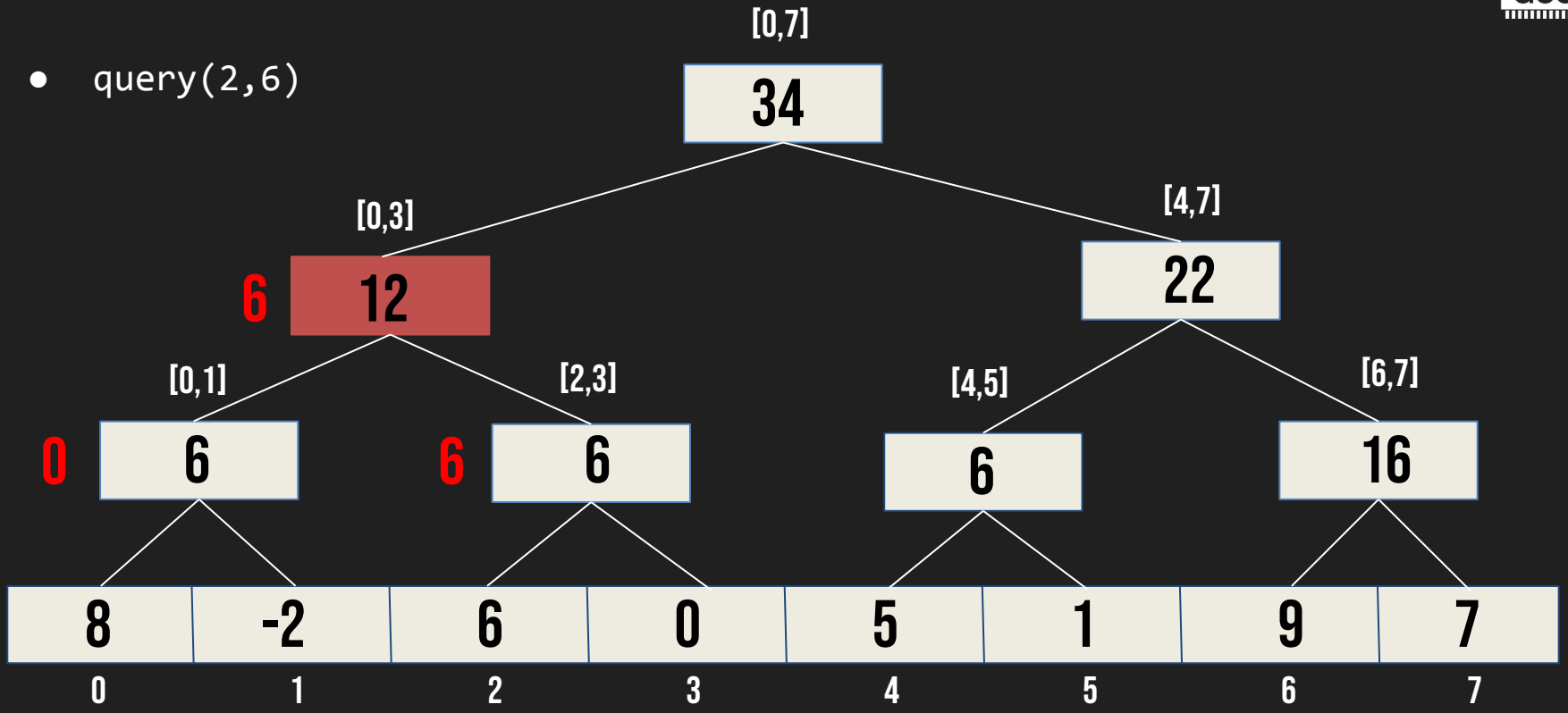
- query(2,6)



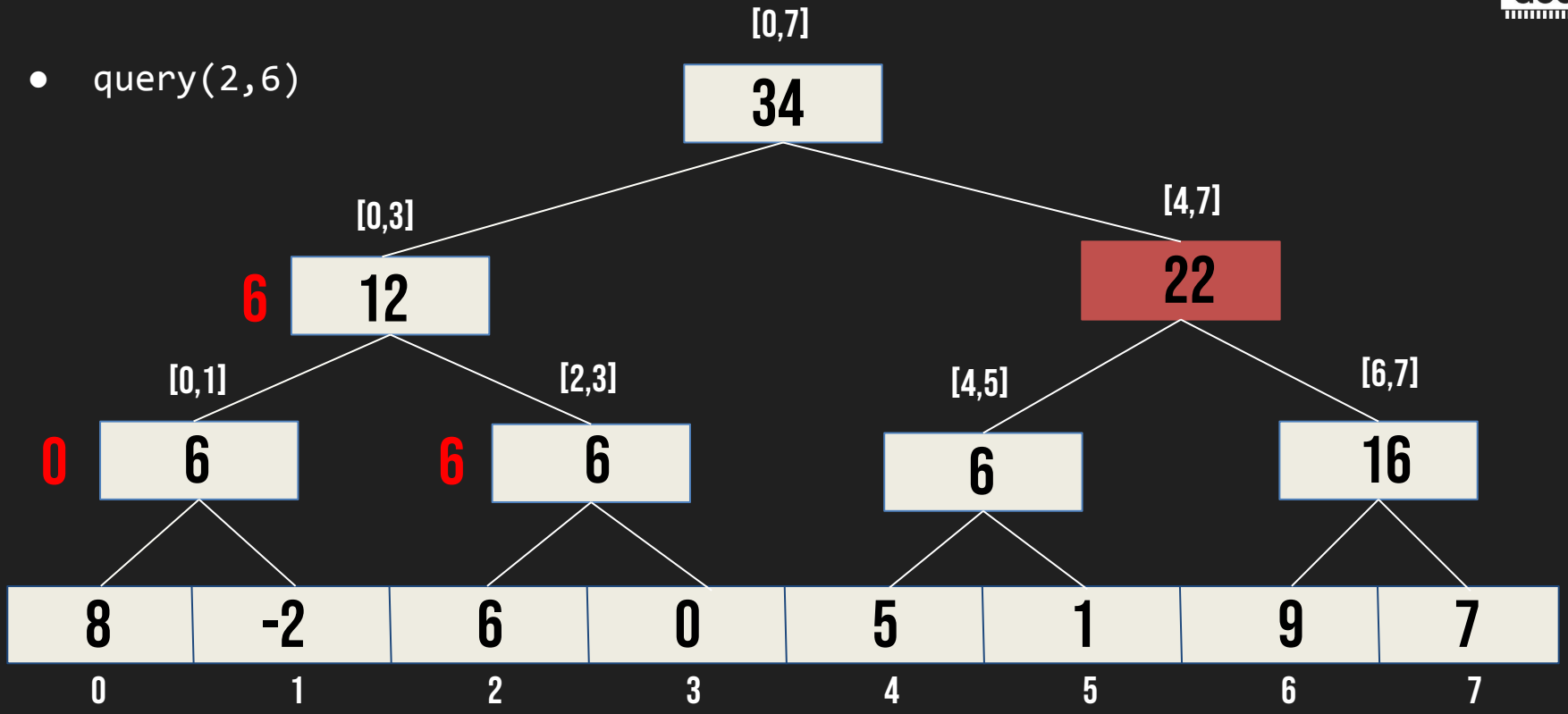
- query(2,6)



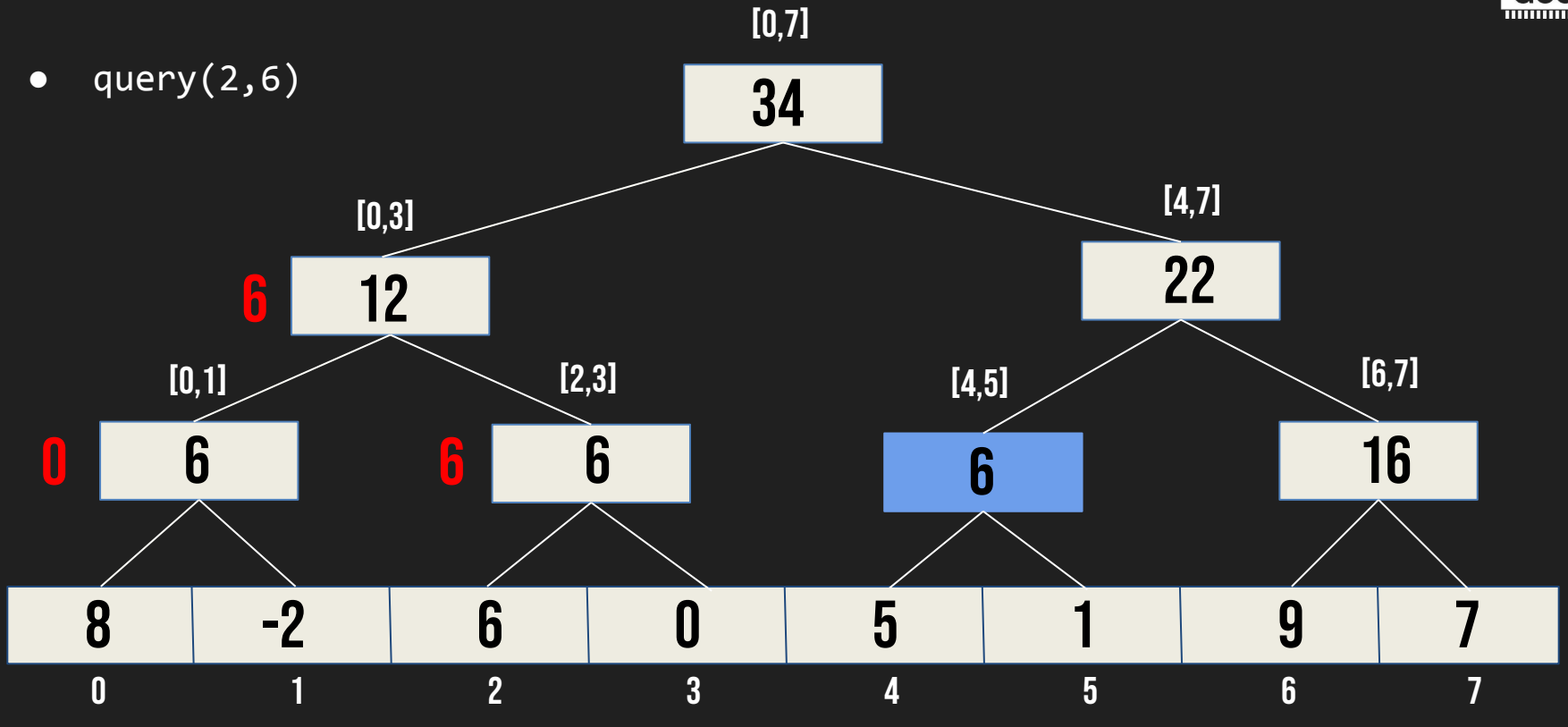
- query(2,6)



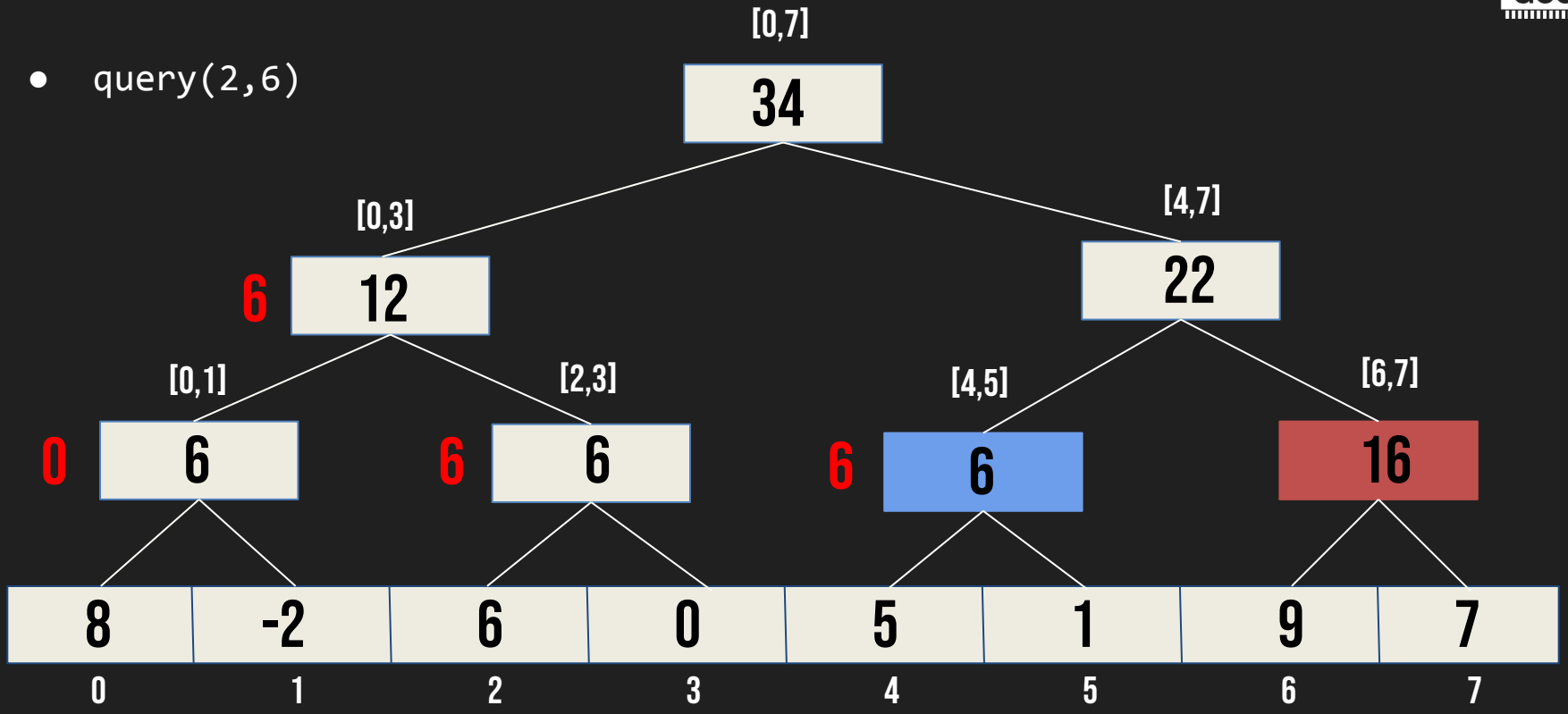
- query(2,6)



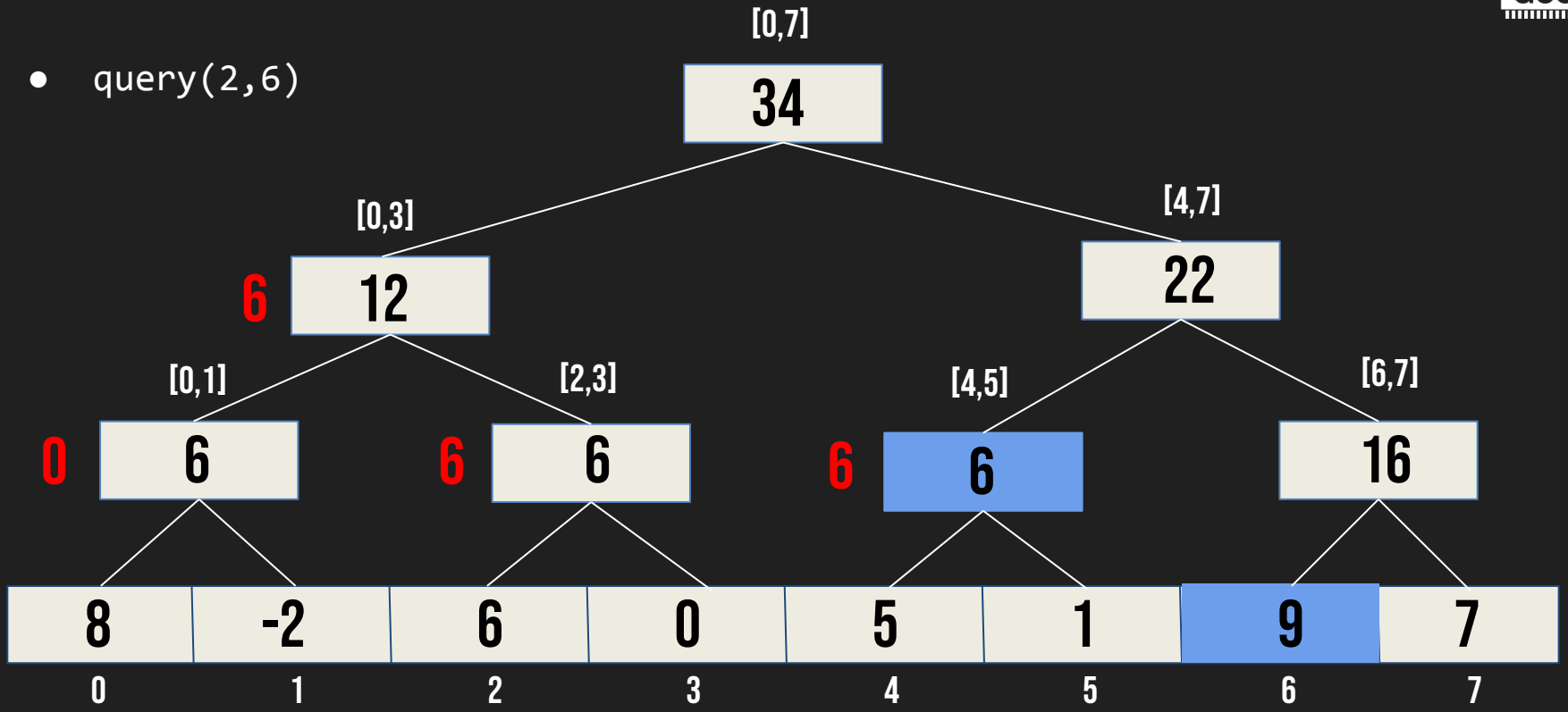
- query(2,6)



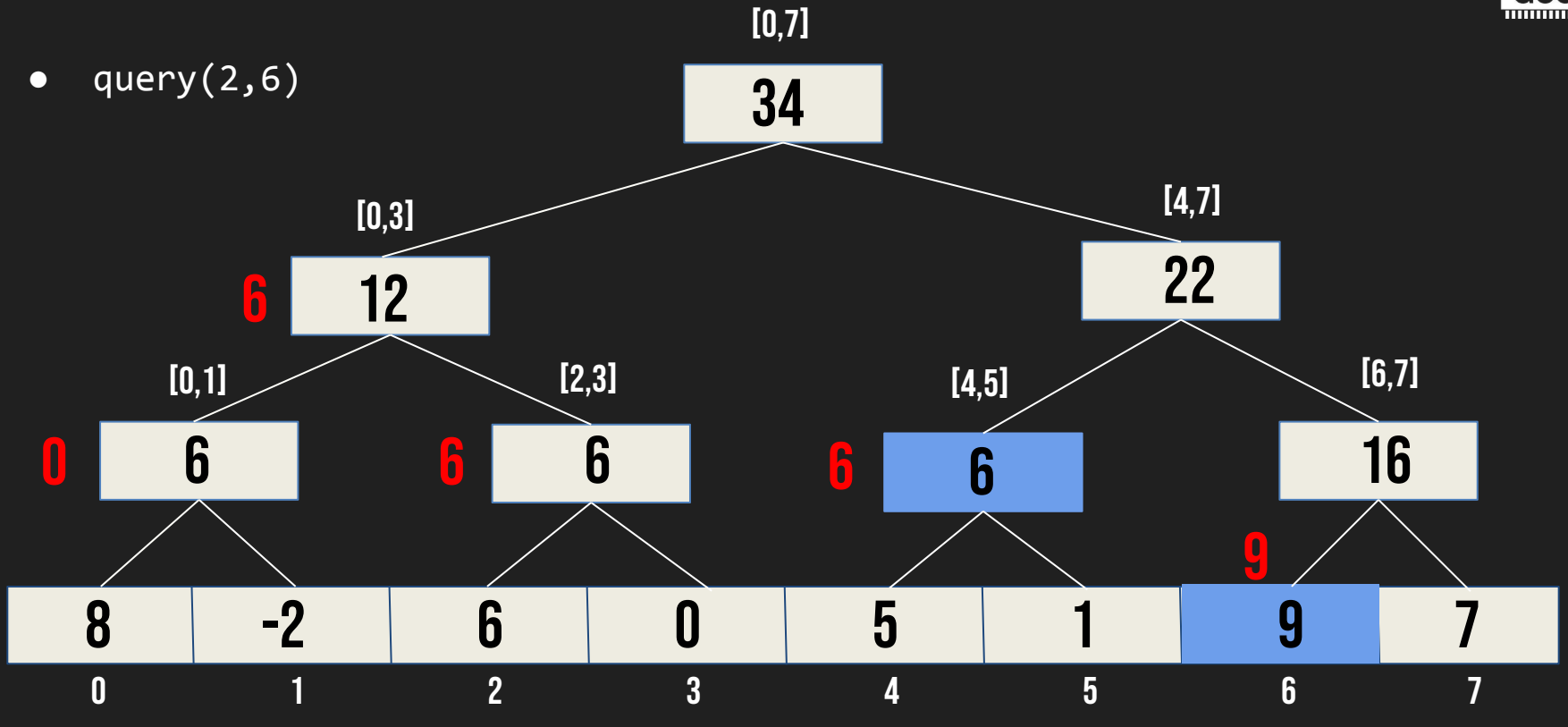
- query(2,6)



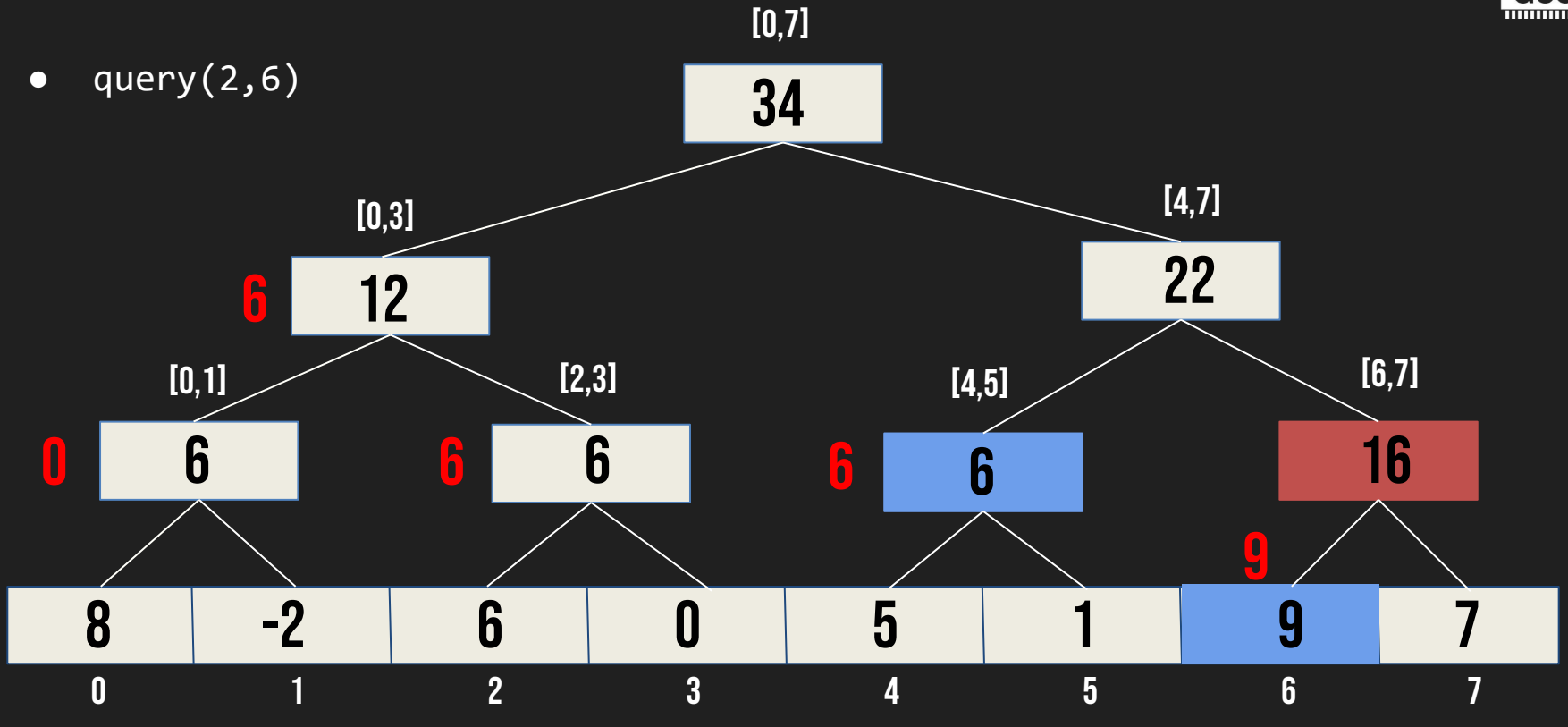
- query(2,6)



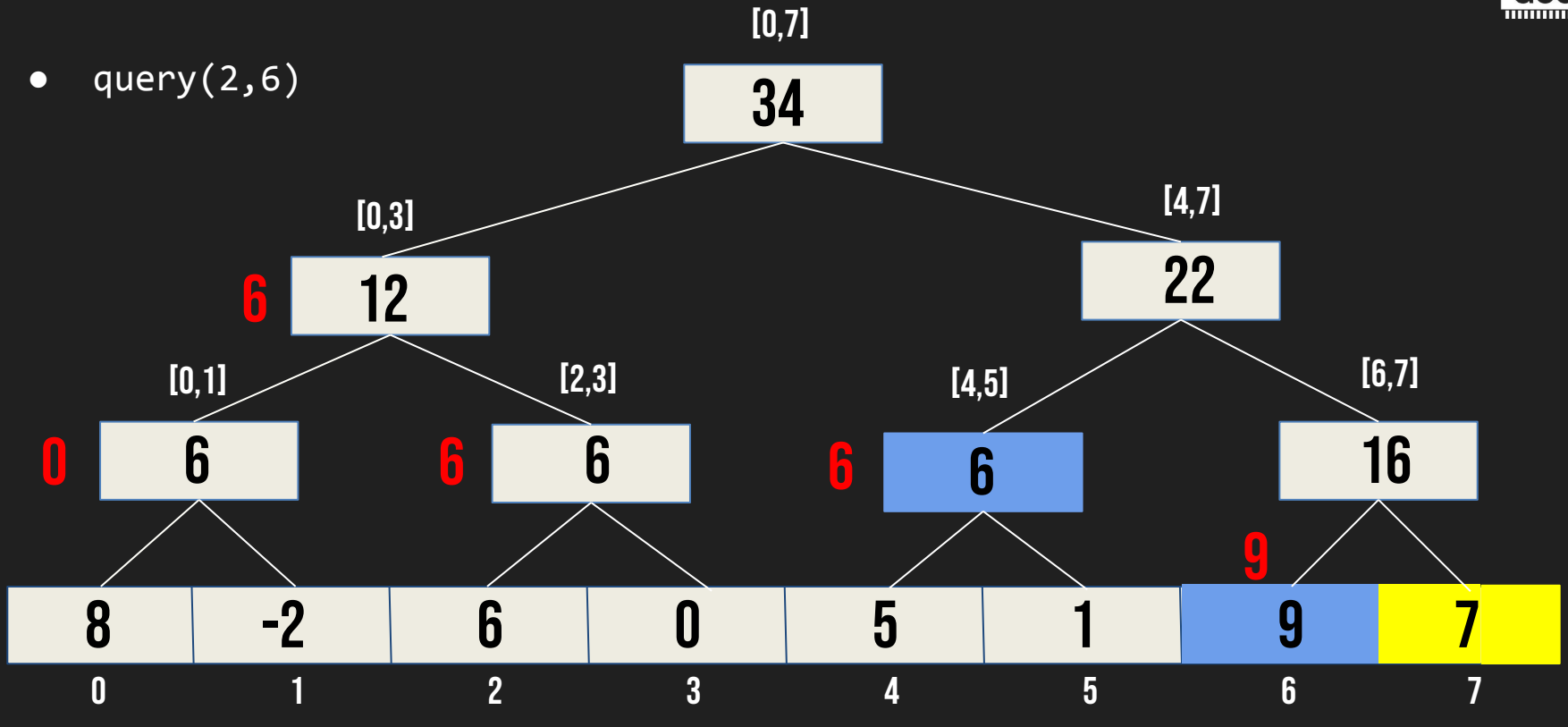
- query(2,6)



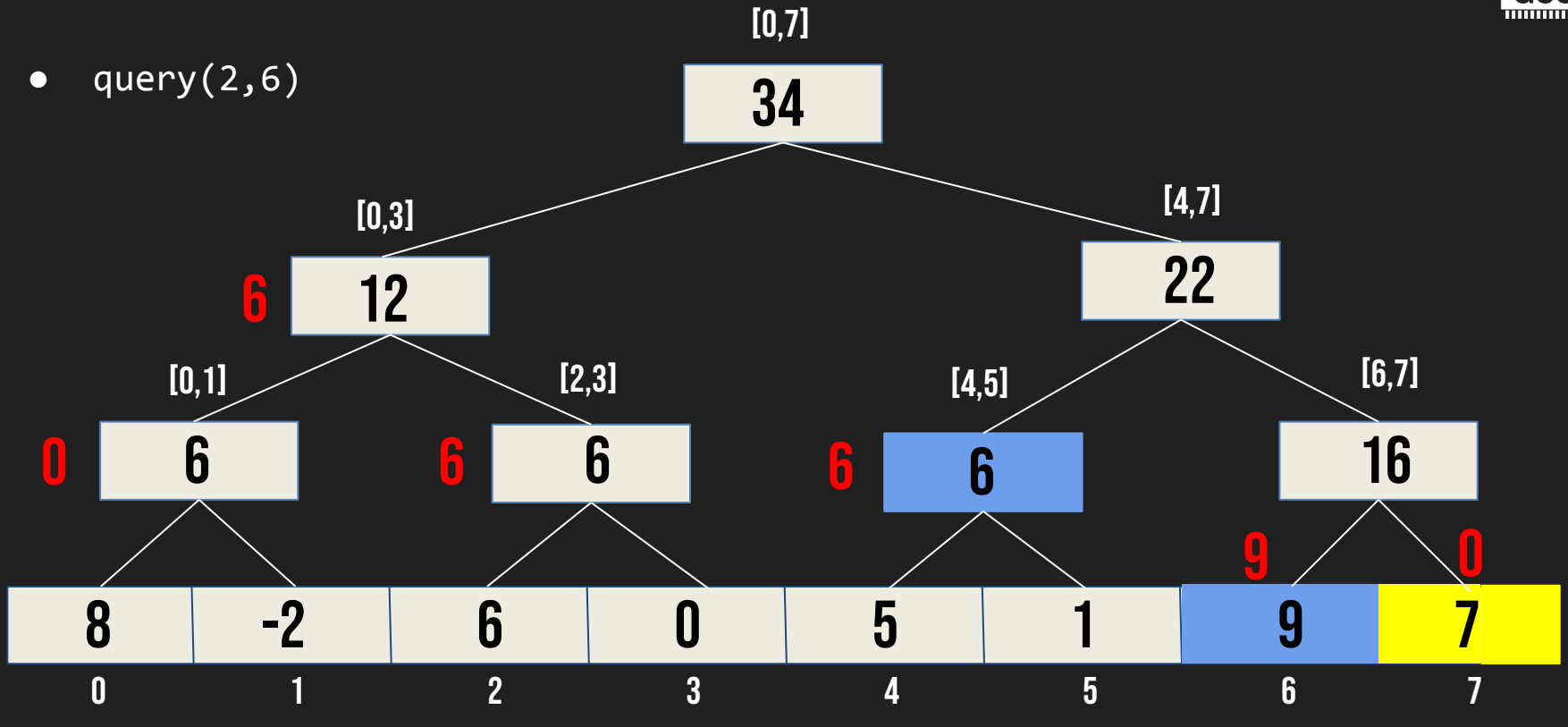
- query(2,6)



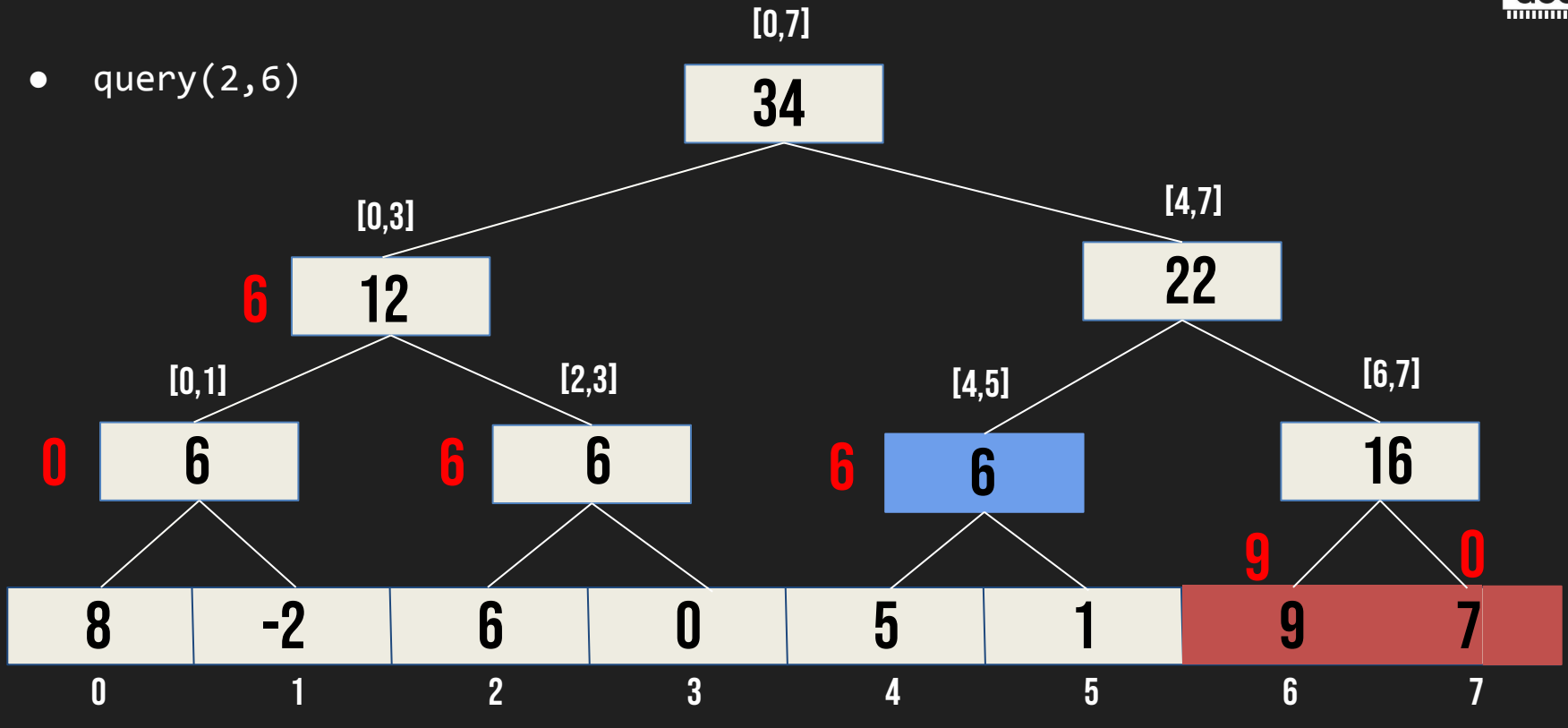
- query(2,6)



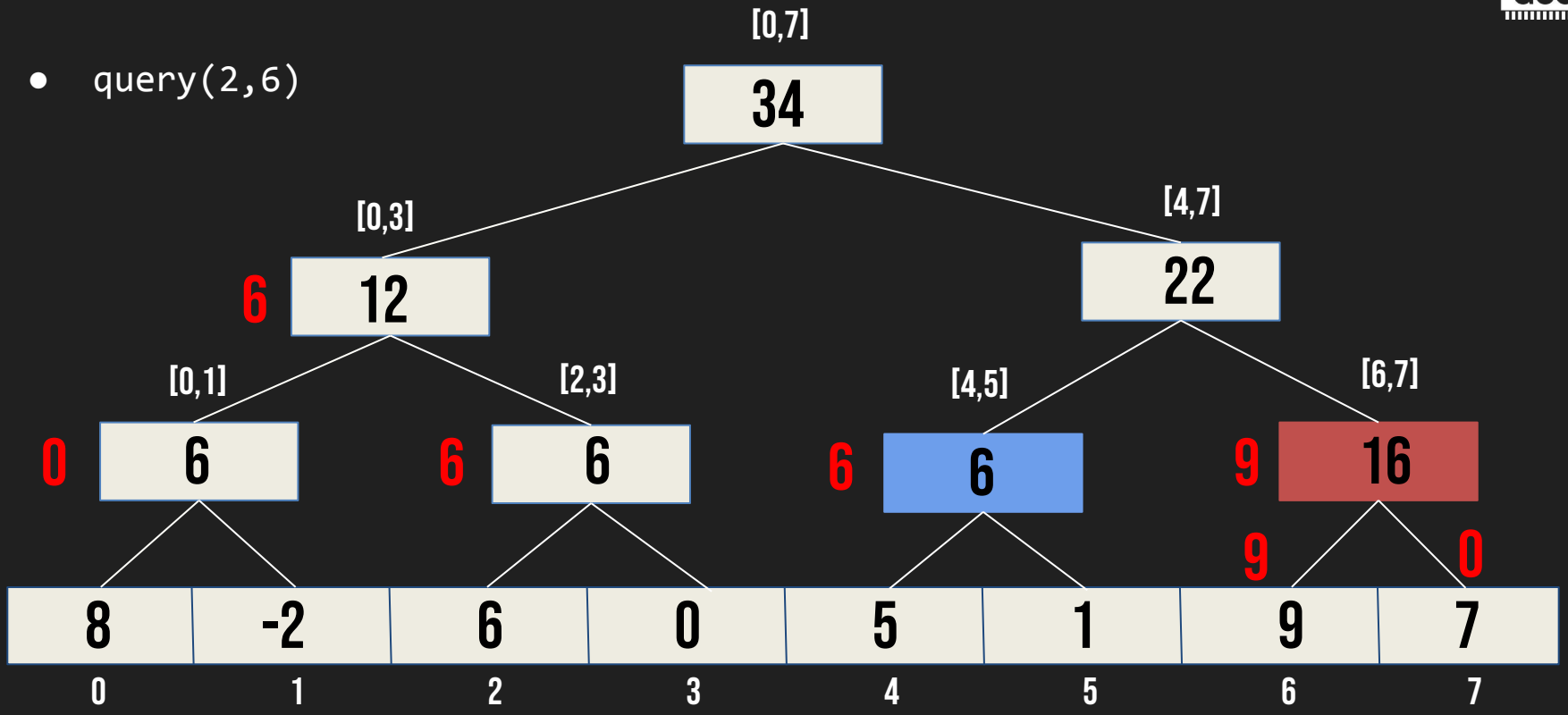
- query(2,6)



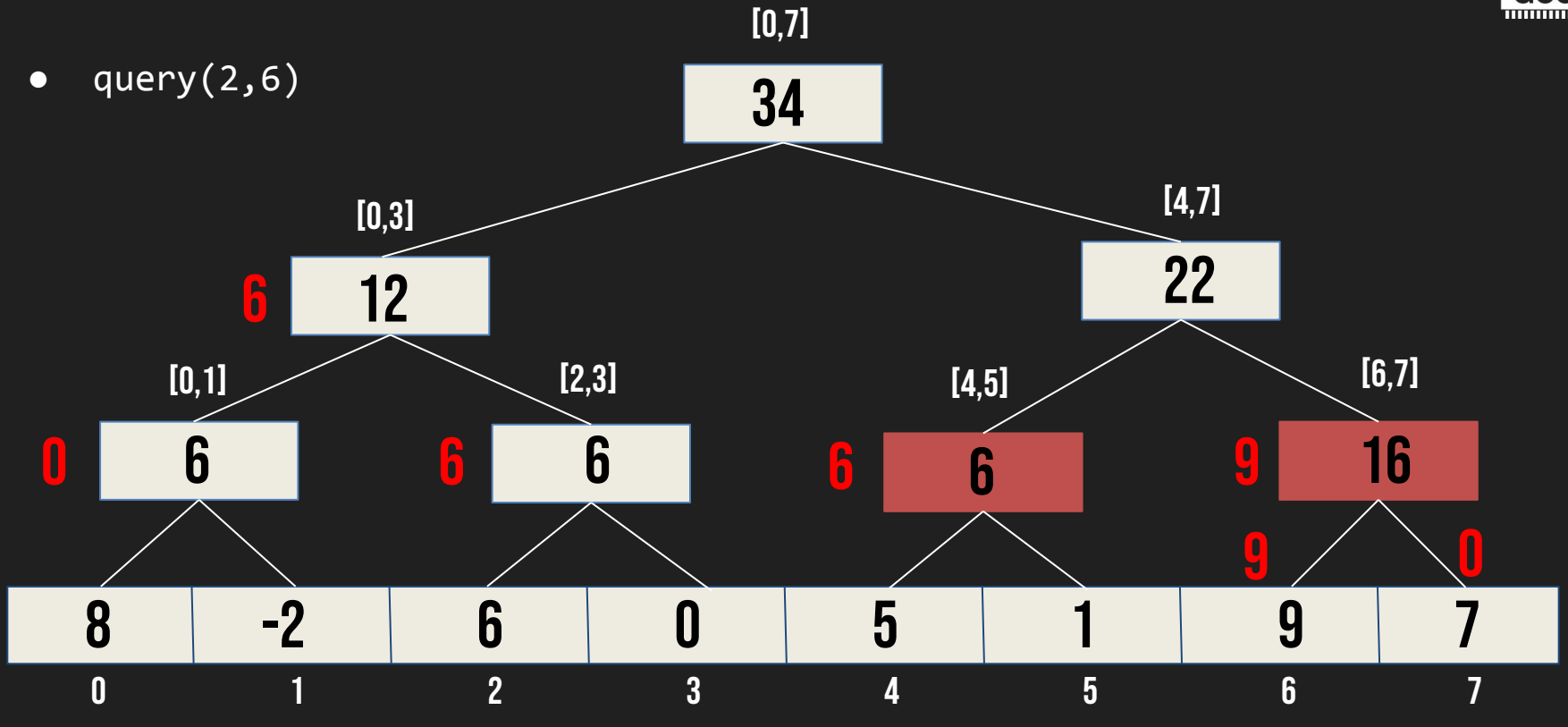
- query(2,6)



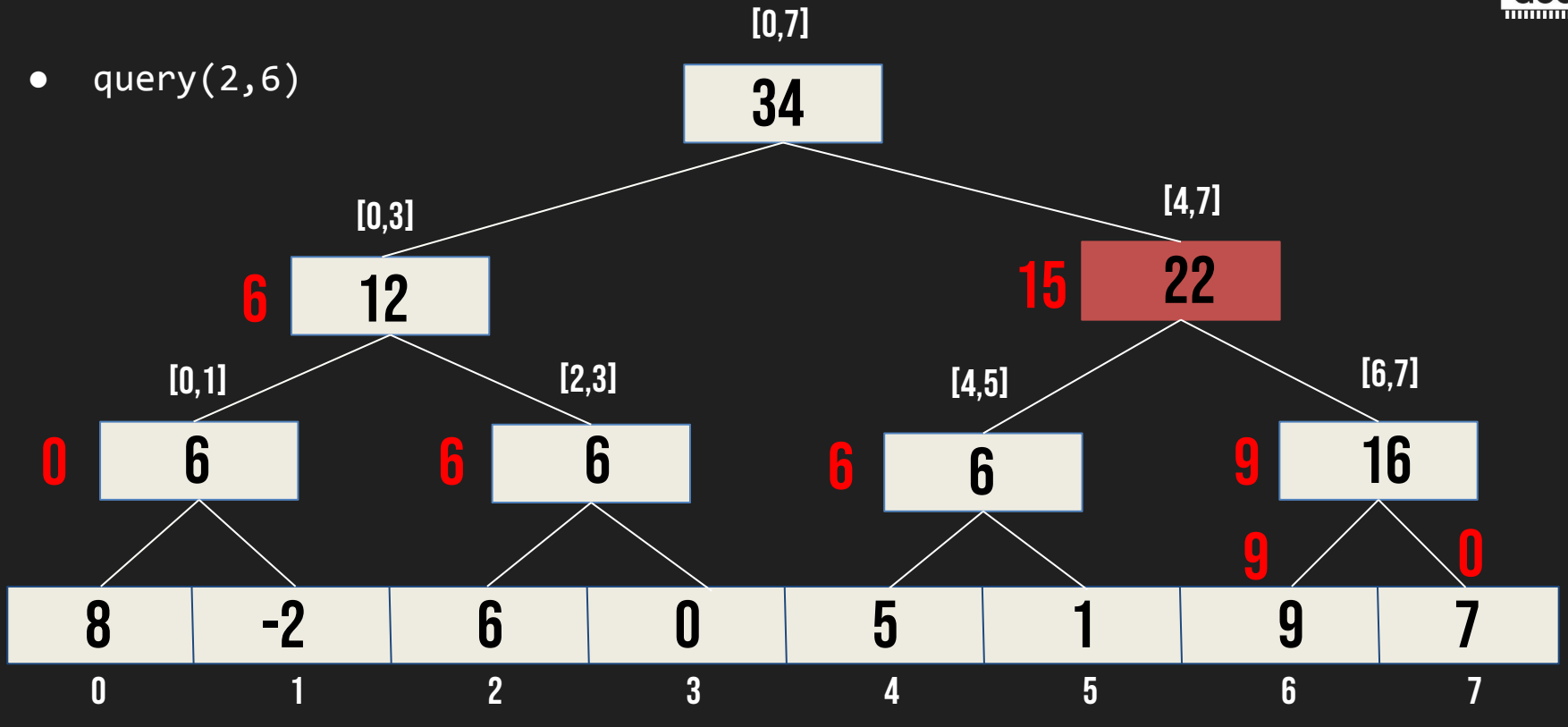
- query(2,6)



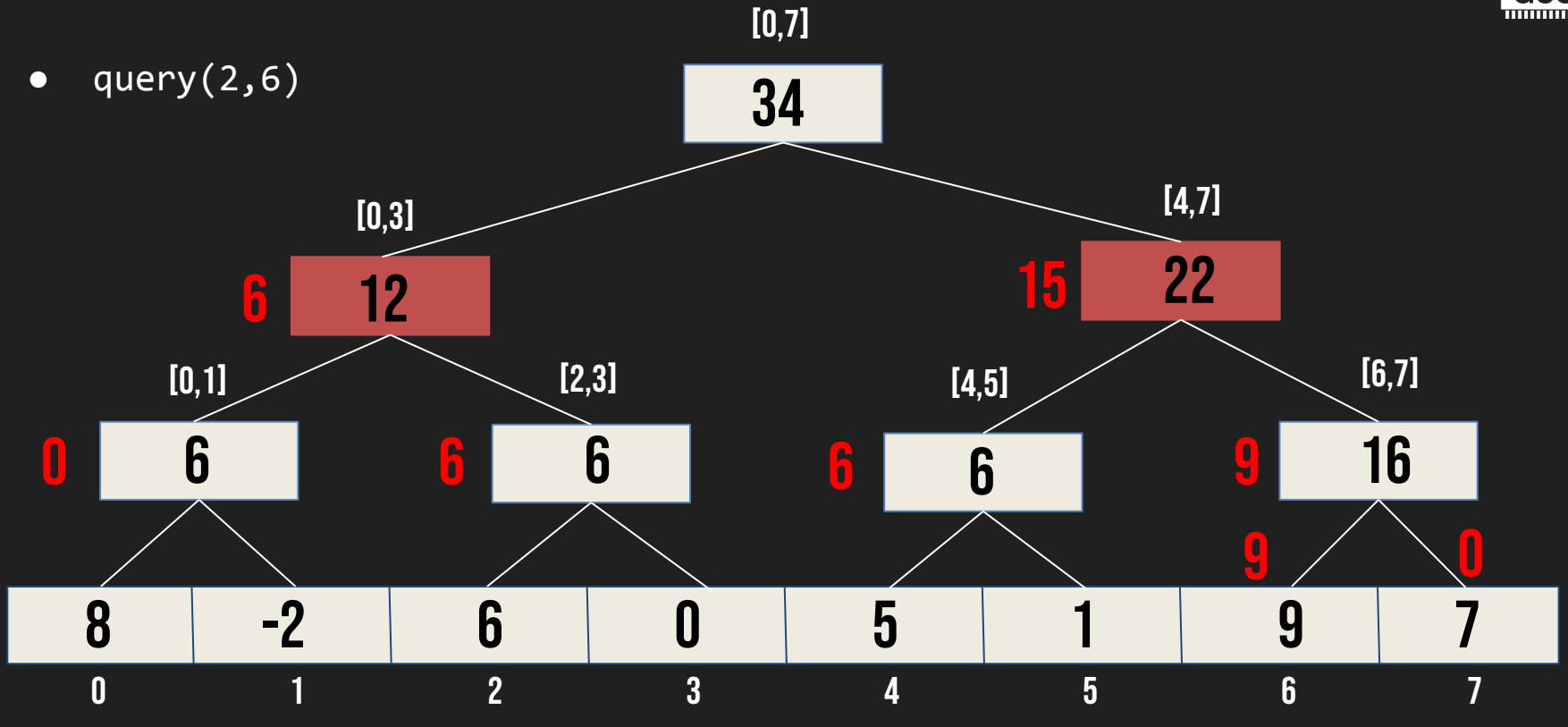
- query(2,6)



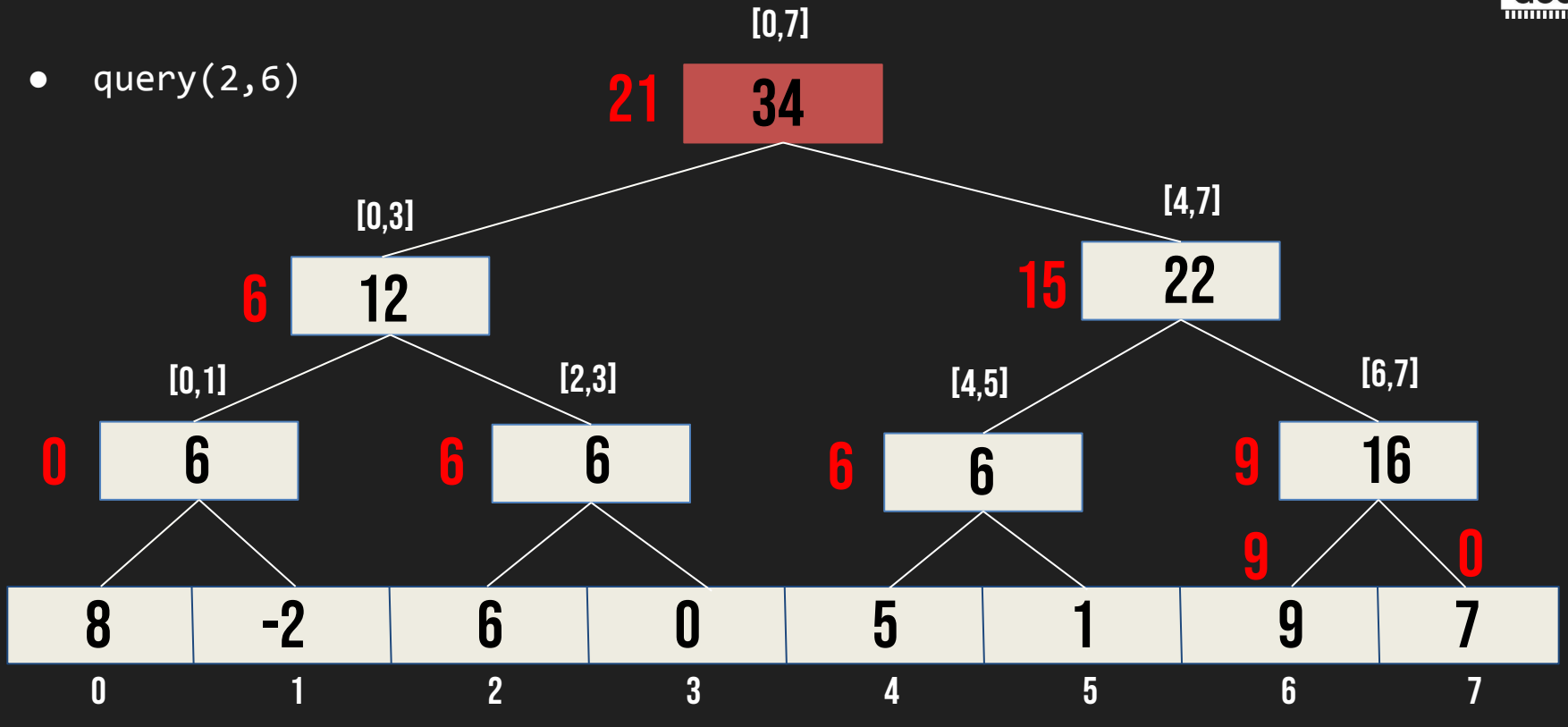
- query(2,6)


[0,7]
34
[0,3]
6
12
[4,7]
15
22
[0,1]
0
6
[2,3]
6
6
[4,5]
6
6
[6,7]
9
16
9
0
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

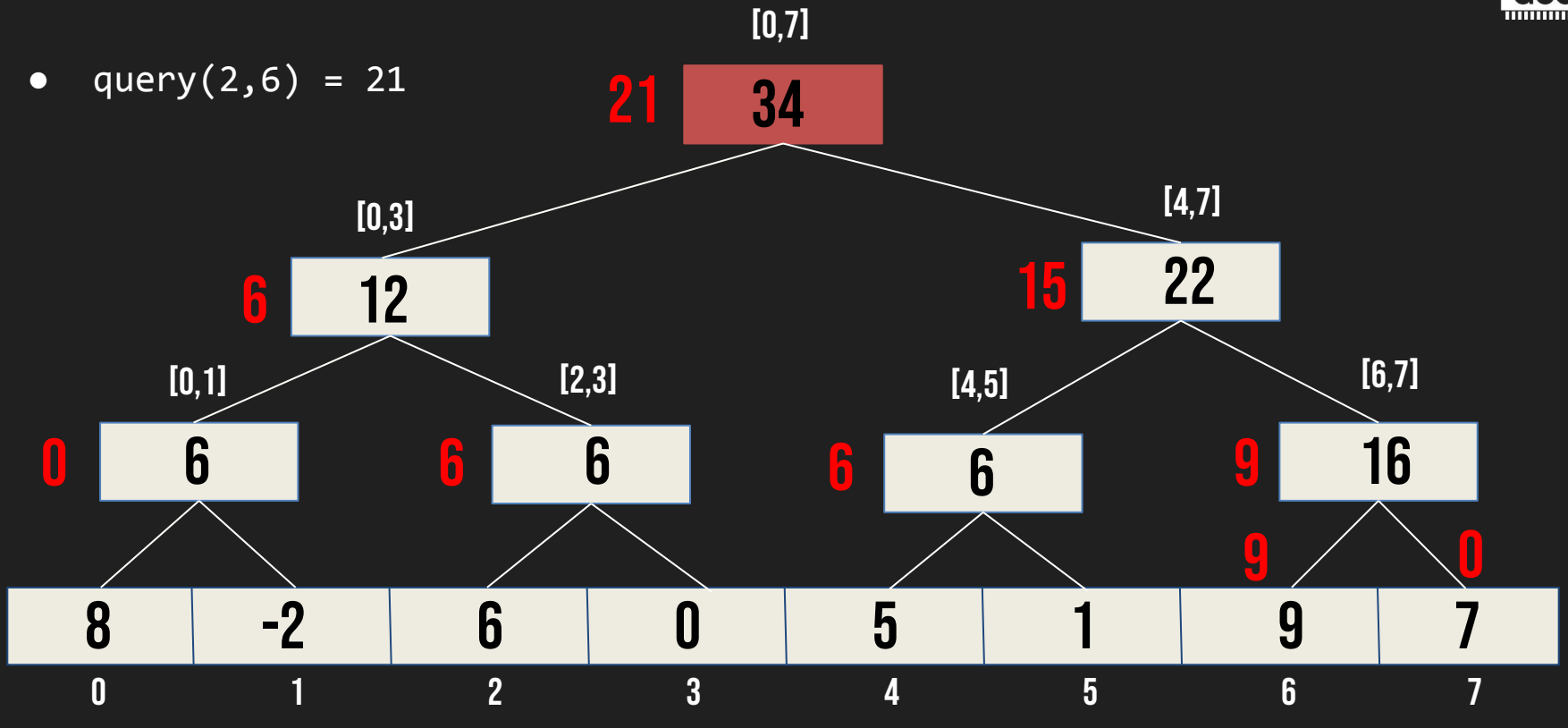
- query(2,6)


[0,7]
34
[0,3]
6
12
[4,7]
15
22
[0,1]
0
6
[2,3]
6
6
[4,5]
6
6
[6,7]
9
16
9
0
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

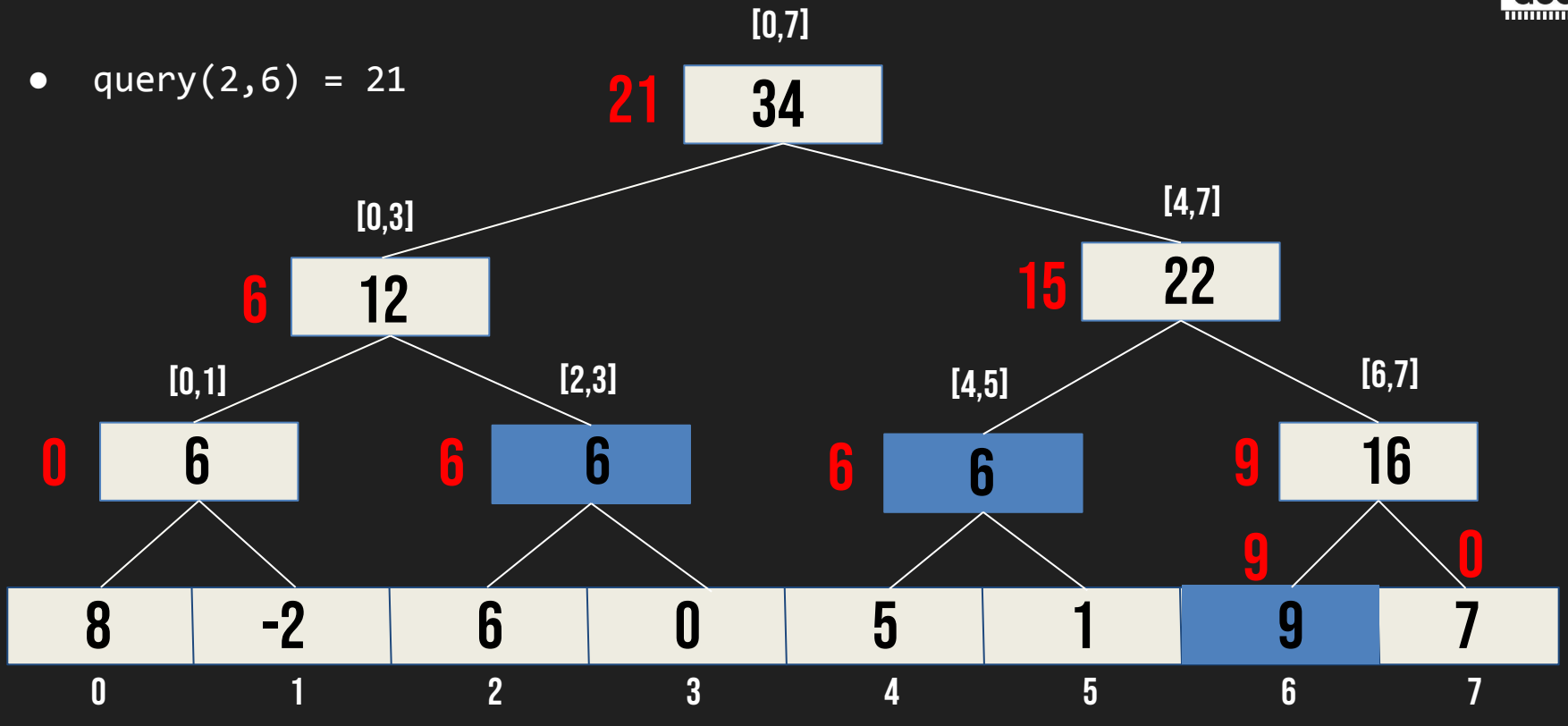
- query(2,6)


[0,7]
21
34
[0,3]
6
12
[4,7]
15
22
[0,1]
0
6
[2,3]
6
6
[4,5]
6
6
[6,7]
9
16
9
0
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

- query(2,6) = 21


[0,7]
21
34
[0,3]
6
12
[4,7]
15
22
[0,1]
0
6
[2,3]
6
6
[4,5]
6
6
[6,7]
9
16
9
0
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

- query(2,6) = 21


[0,7]
21
34
[0,3]
6
12
[4,7]
15
22
[0,1]
0
6
[2,3]
6
6
[4,5]
6
6
[6,7]
9
16
9
0
8
-2
6
0
5
1
9
7
0
1
2
3
4
5
6
7

06 - ALGORITMO

06 - ALGORITMO

```
int query(int node, int l, int r, int a, int b){



}
```

06 - ALGORITMO

```
int query(int node, int l, int r, int a, int b){
    if(b < l or r < a) return 0;



}
```

06 - ALGORITMO

```
int query(int node, int l, int r, int a, int b){
    if(b < l or r < a) return 0;
    if(a <= l and r <= b) return tree[node];


}
```

06 - ALGORITMO

```
int query(int node, int l, int r, int a, int b){
    if(b < l or r < a) return 0;
    if(a <= l and r <= b) return tree[node];
    int m = (l+r)/2;

}
```

06 - ALGORITMO

```
int query(int node, int l, int r, int a, int b){
    if(b < l or r < a) return 0;
    if(a <= l and r <= b) return tree[node];
    int m = (l+r)/2;
    return query(2*node, l, m, a, b) + query(2*node+1, m+1, r, a, b);
}
```

06 - ALGORITMO

```
int query(int node, int l, int r, int a, int b){
    if(b < l or r < a) return 0;
    if(a <= l and r <= b) return tree[node];
    int m = (l+r)/2;
    return query(2*node, l, m, a, b) + query(2*node+1, m+1, r, a, b);
}

cout << query(1, 0, n-1, a, b) << endl;
```

06 - ALGORITMO

```
int query(int node, int l, int r, int a, int b){
    if(b < l or r < a) return 0;
    if(a <= l and r <= b) return tree[node];
    int m = (l+r)/2;
    return query(2*node, l, m, a, b) + query(2*node+1, m+1, r, a, b);
}

cout << query(1, 0, n-1, a, b) << endl; // gastamos O(log n) para a query
```

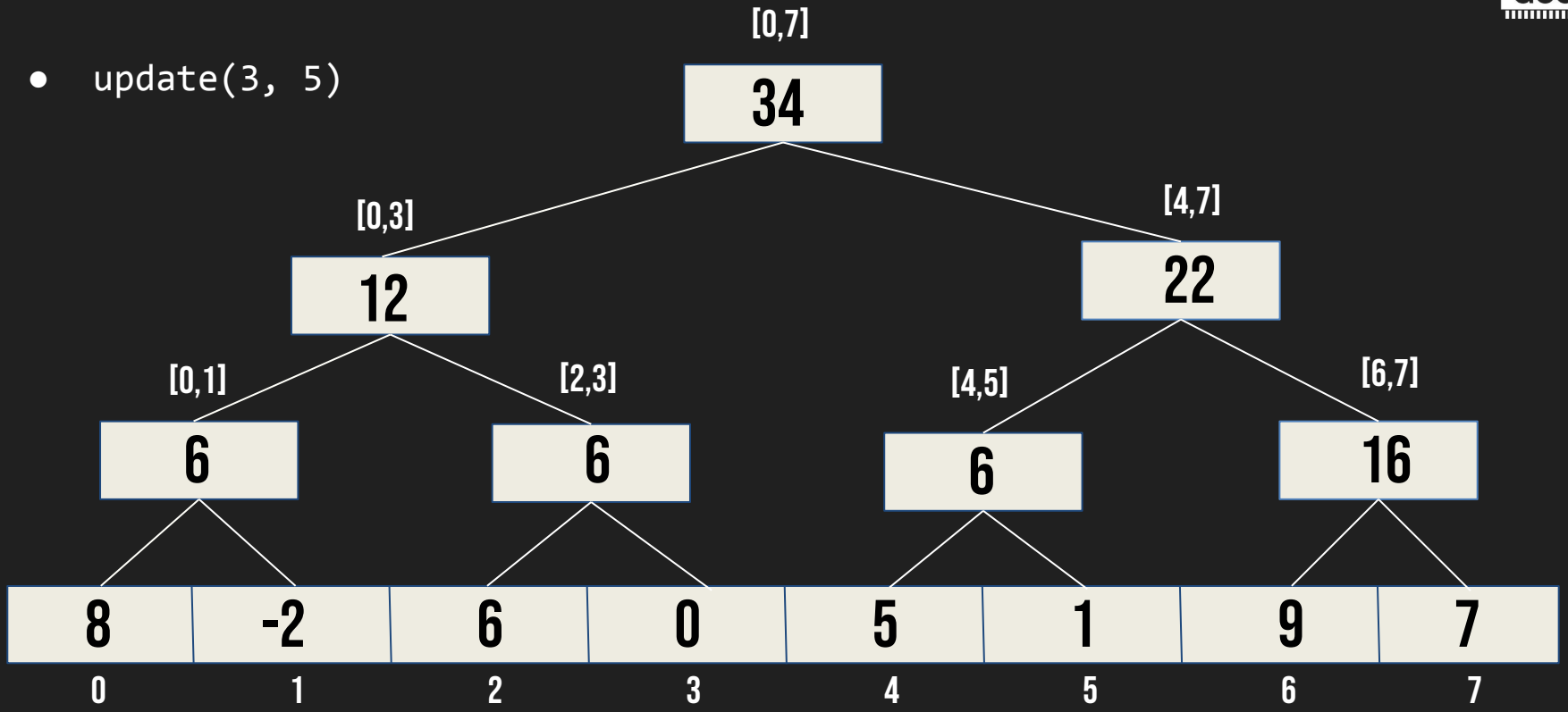
07 - ATUALIZAÇÃO

- E se quisermos atualizar uma posição do array (update), como podemos atualizar eficientemente a SegTree?

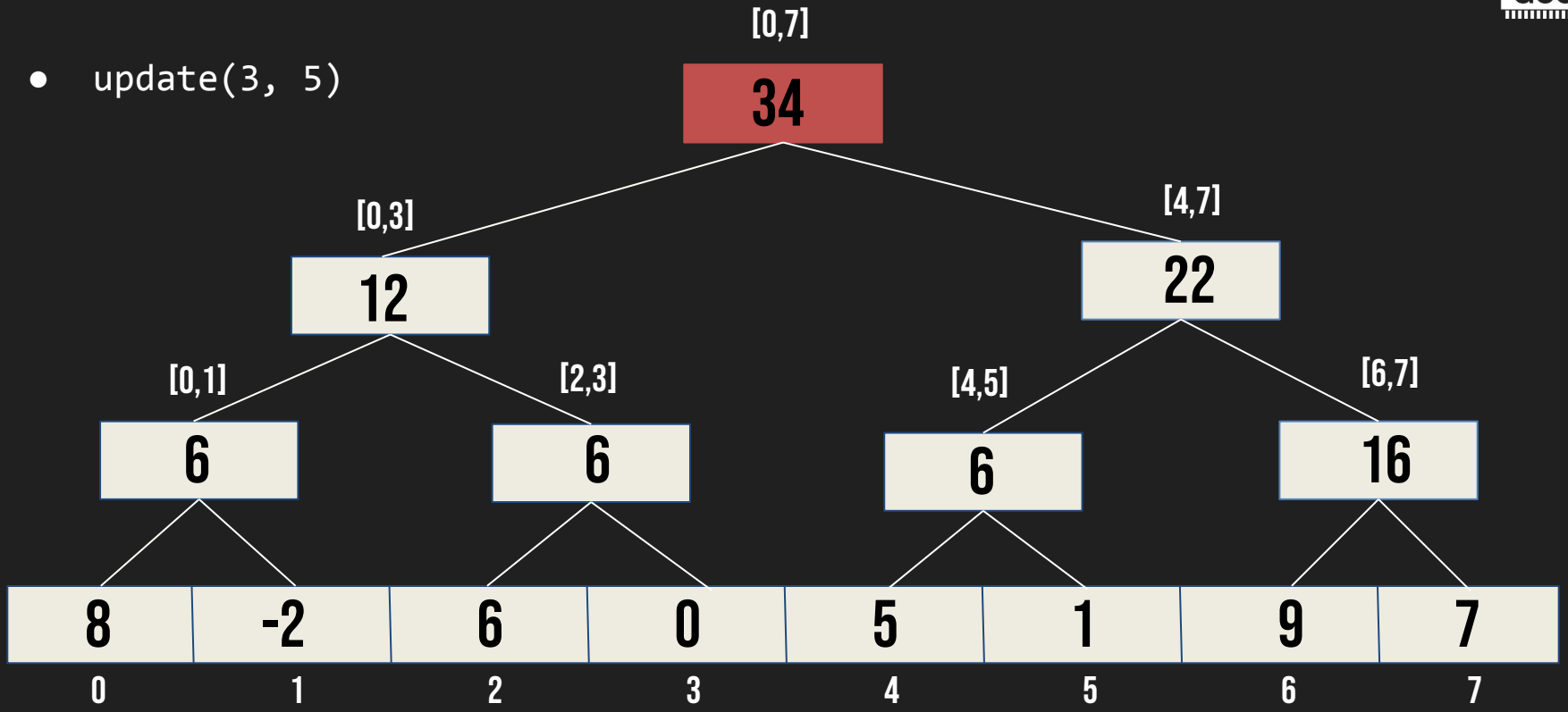
- Vamos fazer uma função parecida com a construção (build) da SegTree.

- Percorremos a árvore até o nó que representa a posição **`i`** da atualização, e somamos o valor do seu nó com **x**, e retornamos fazendo a combinação dos nós (operações) anteriores

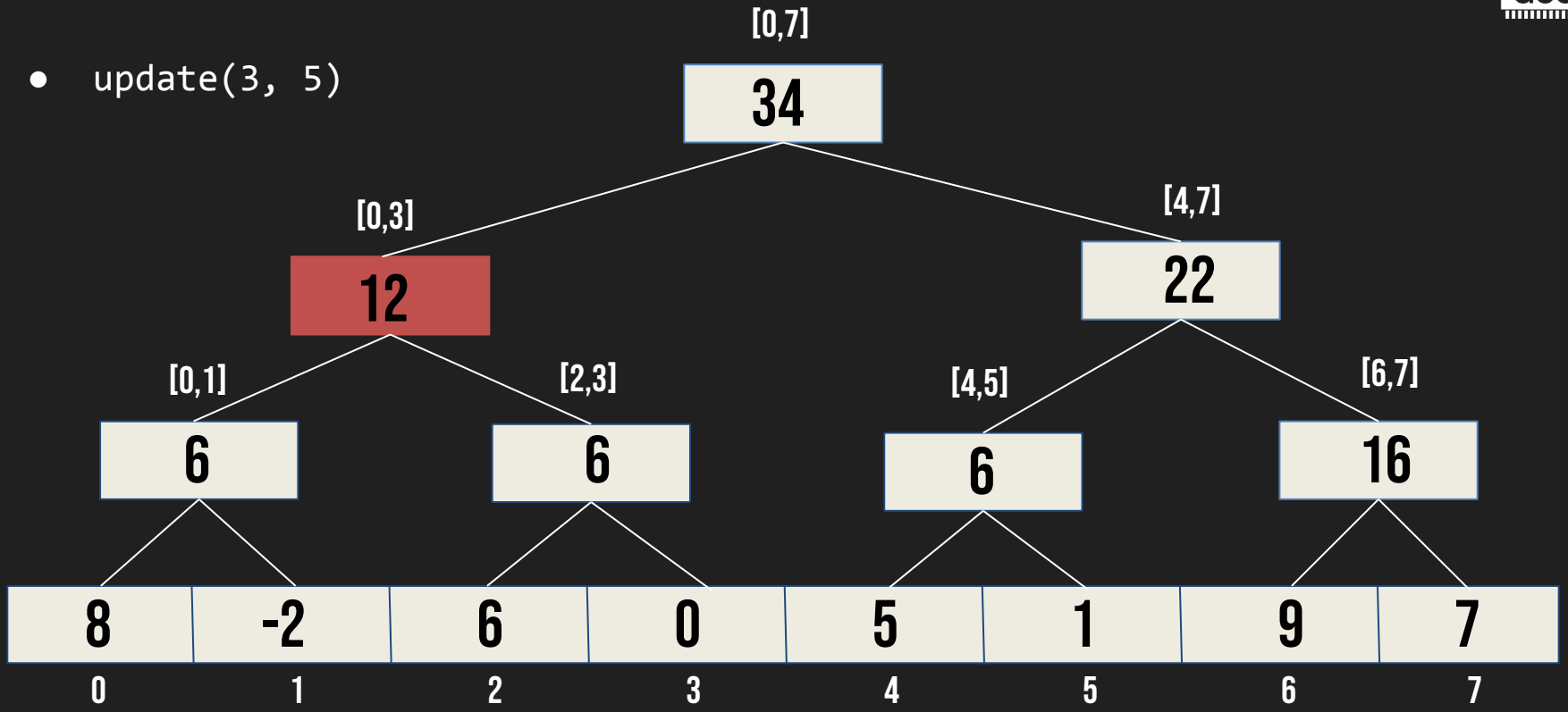
- `update(3, 5)`



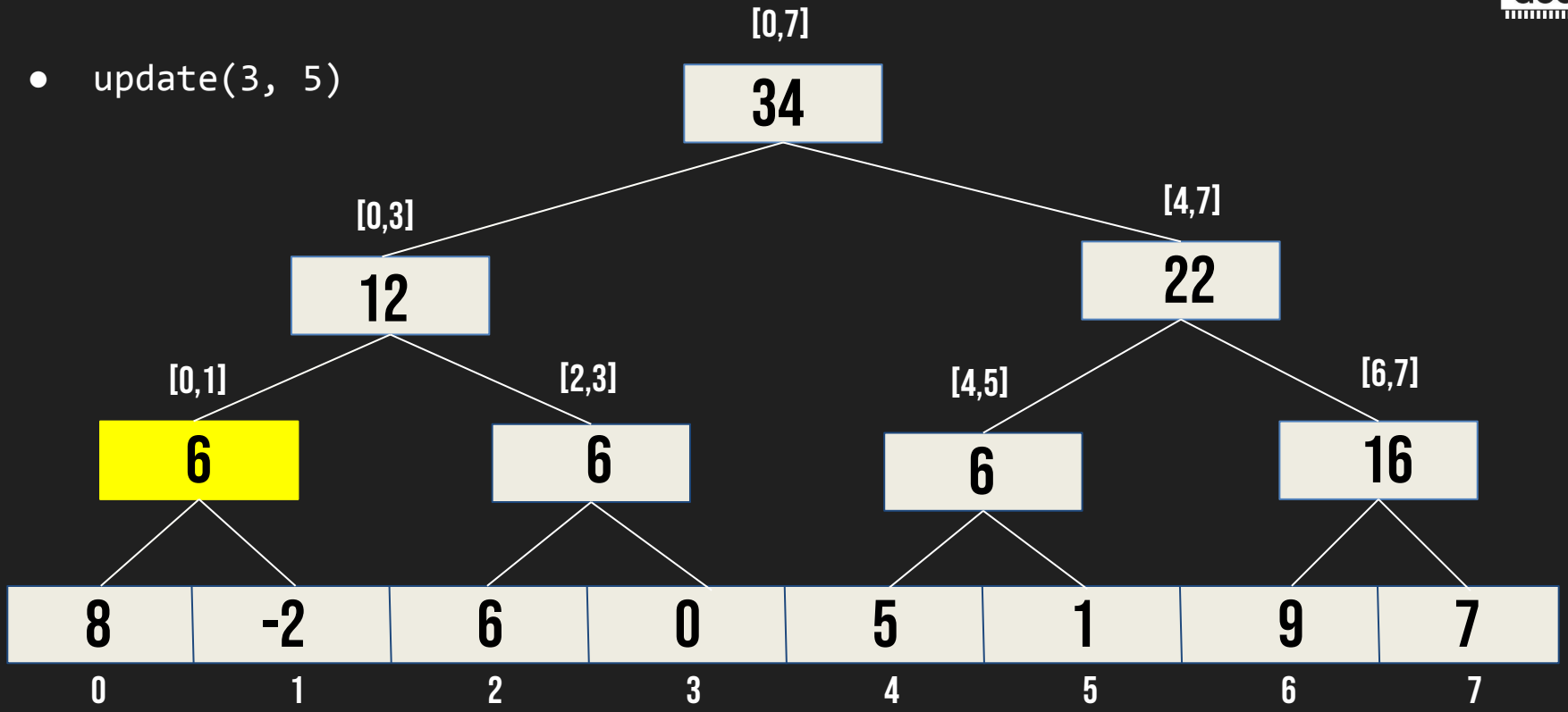
- `update(3, 5)`



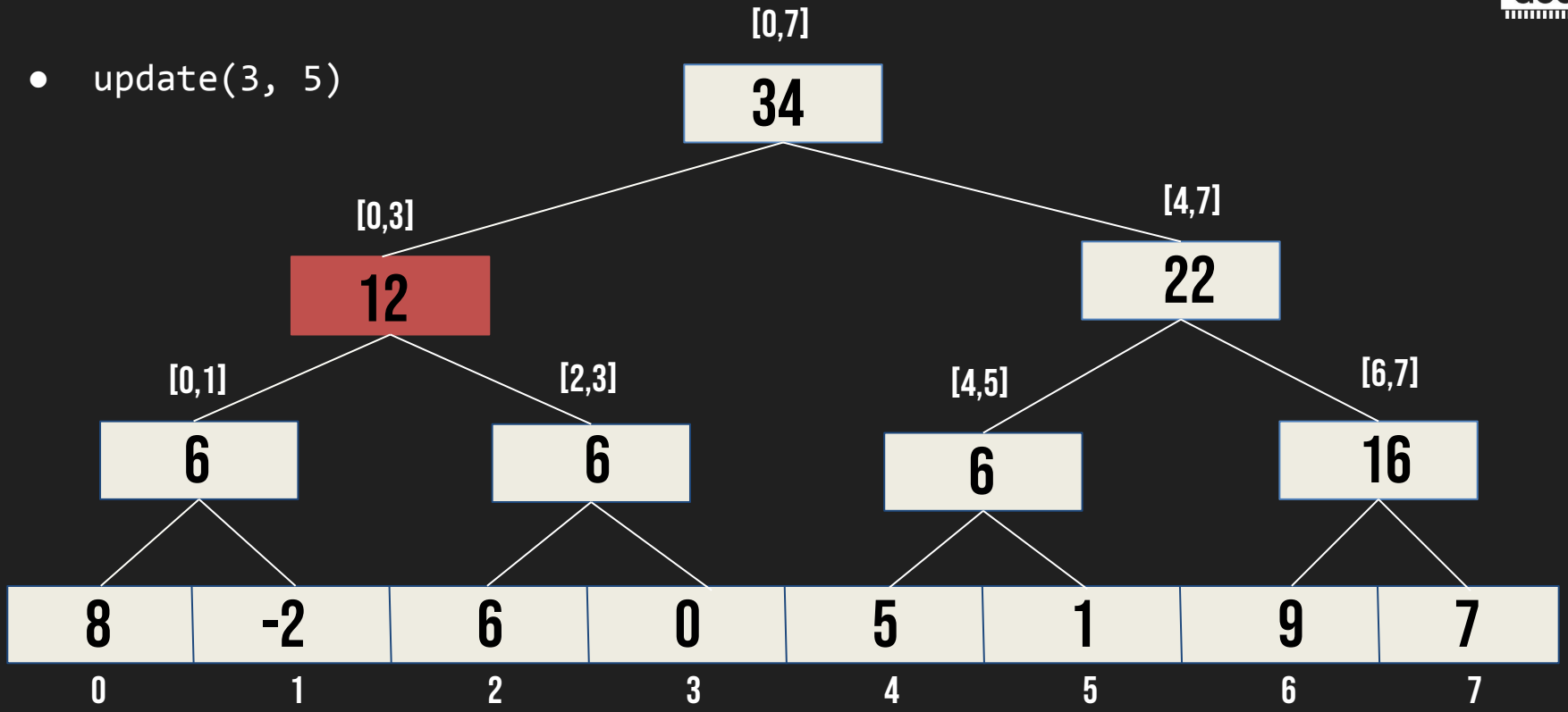
- `update(3, 5)`



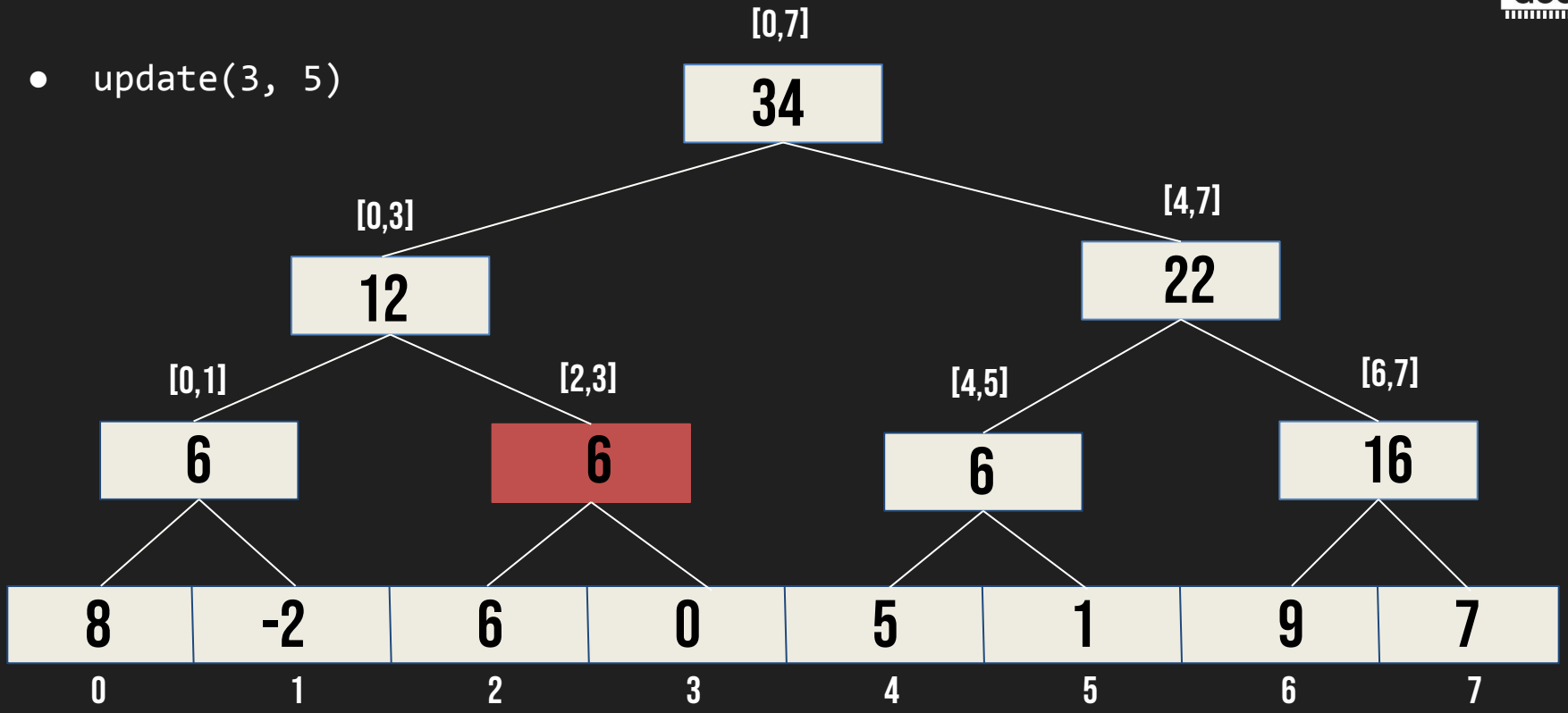
- update(3, 5)



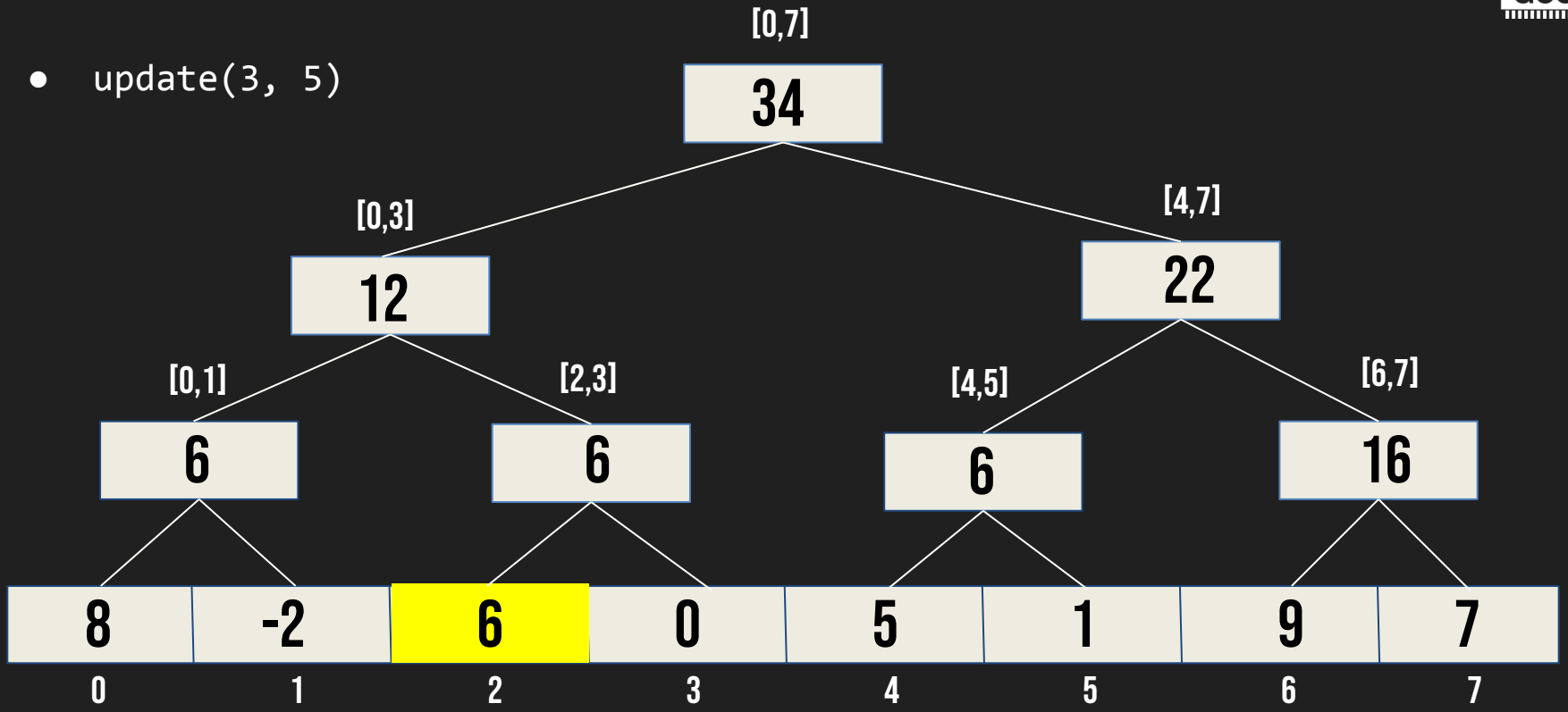
- `update(3, 5)`



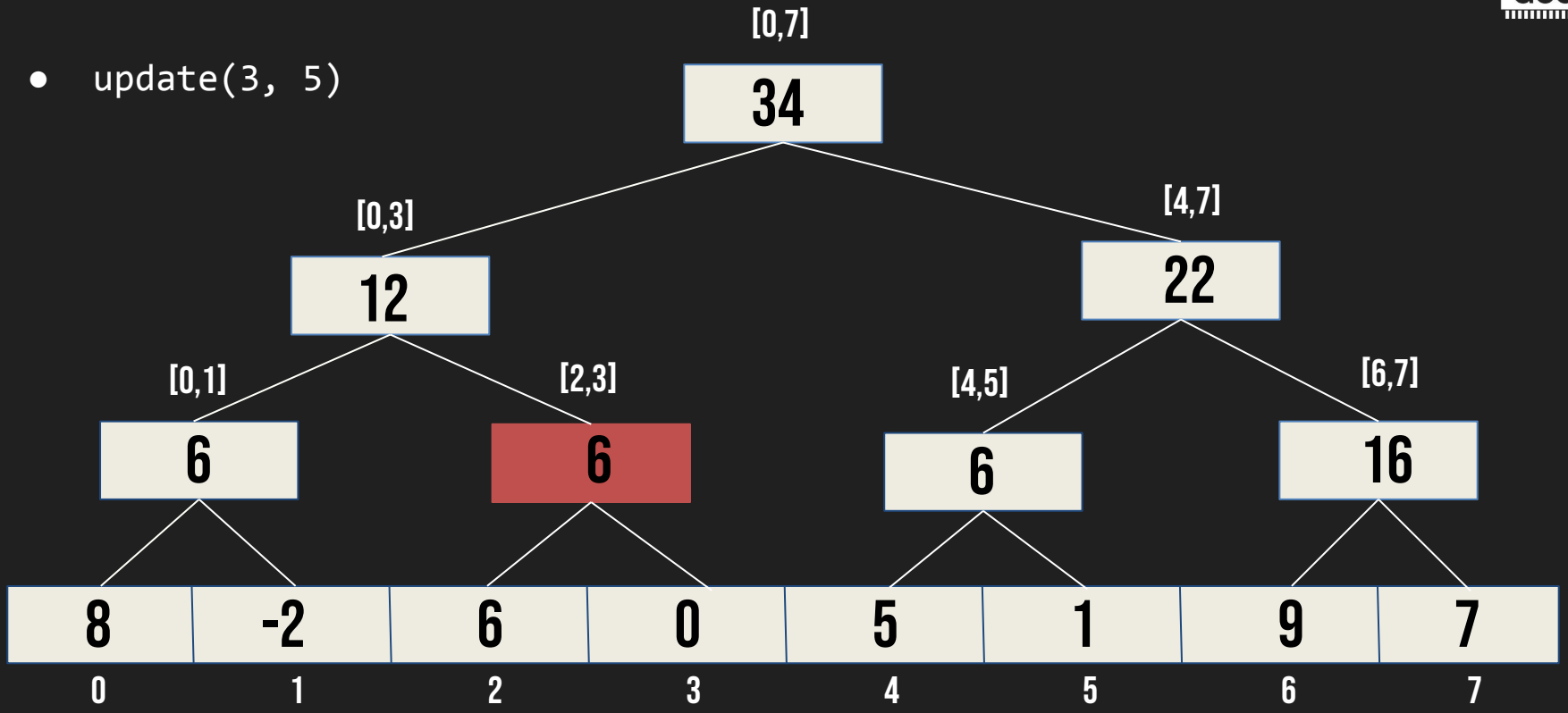
- `update(3, 5)`



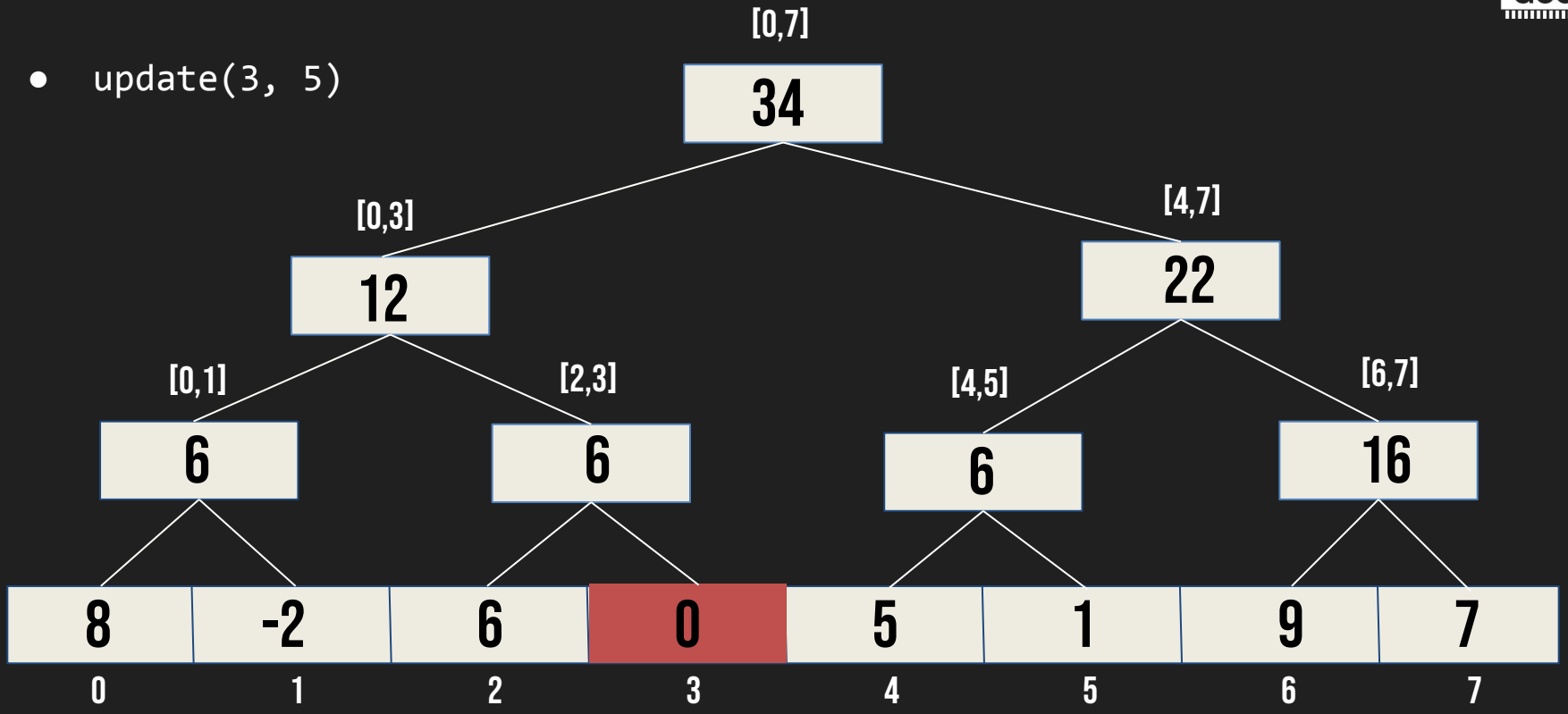
- `update(3, 5)`



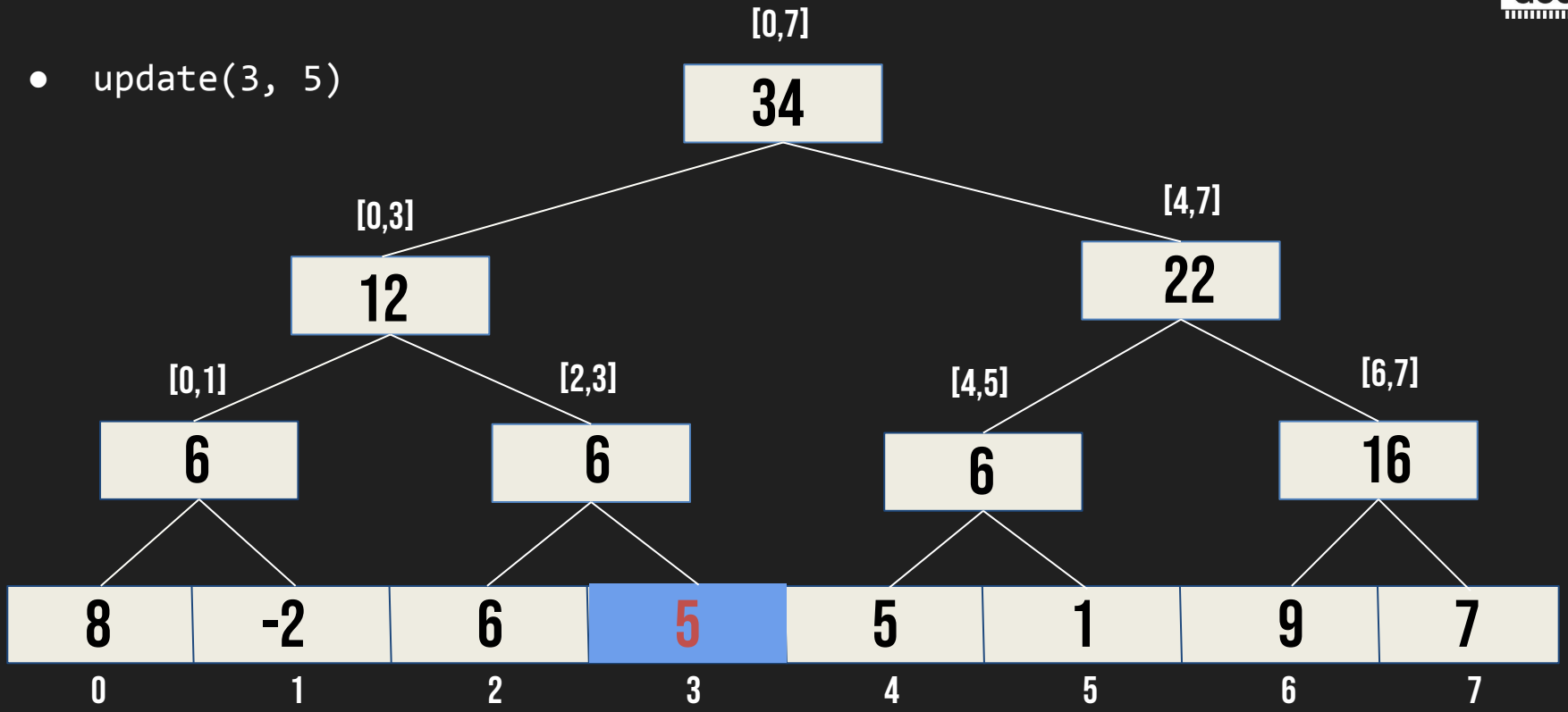
- `update(3, 5)`



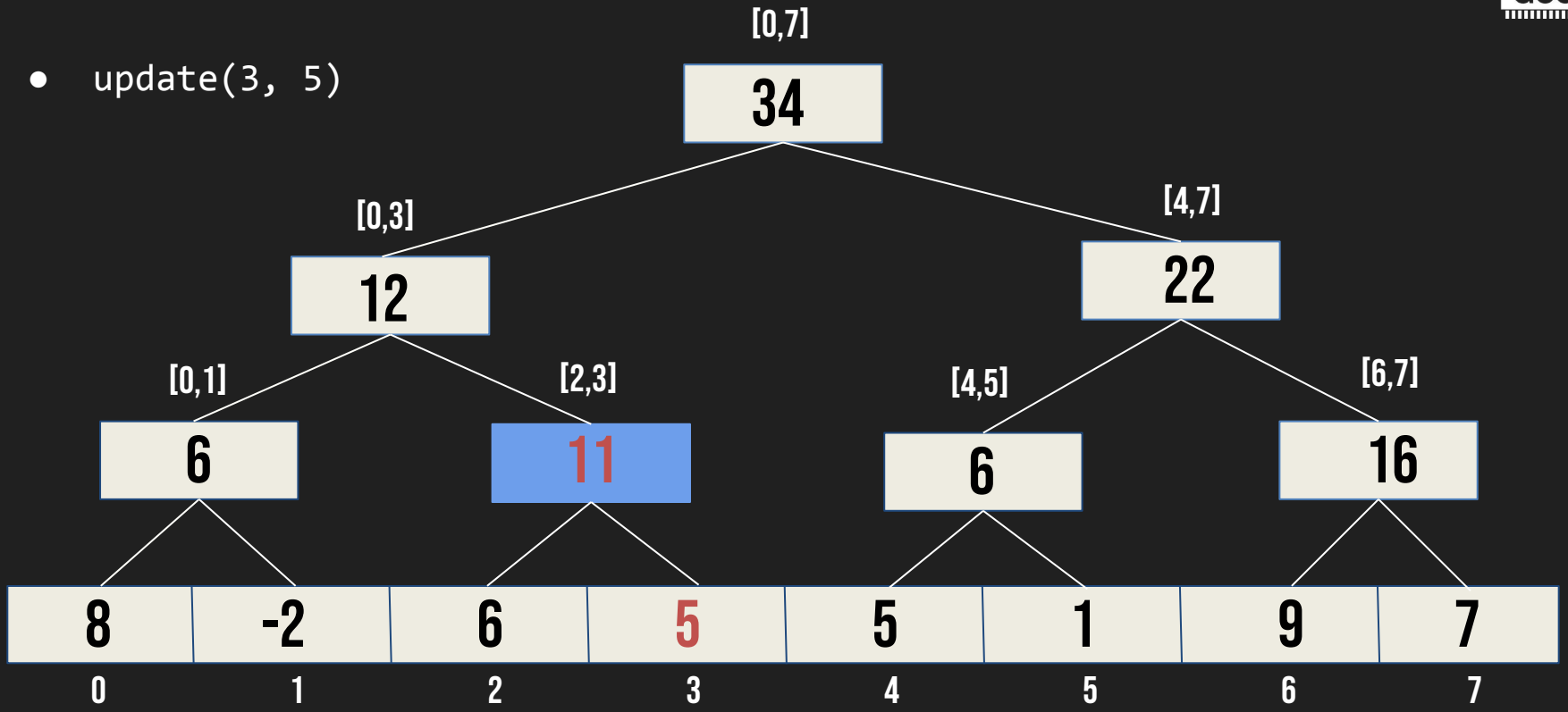
- update(3, 5)



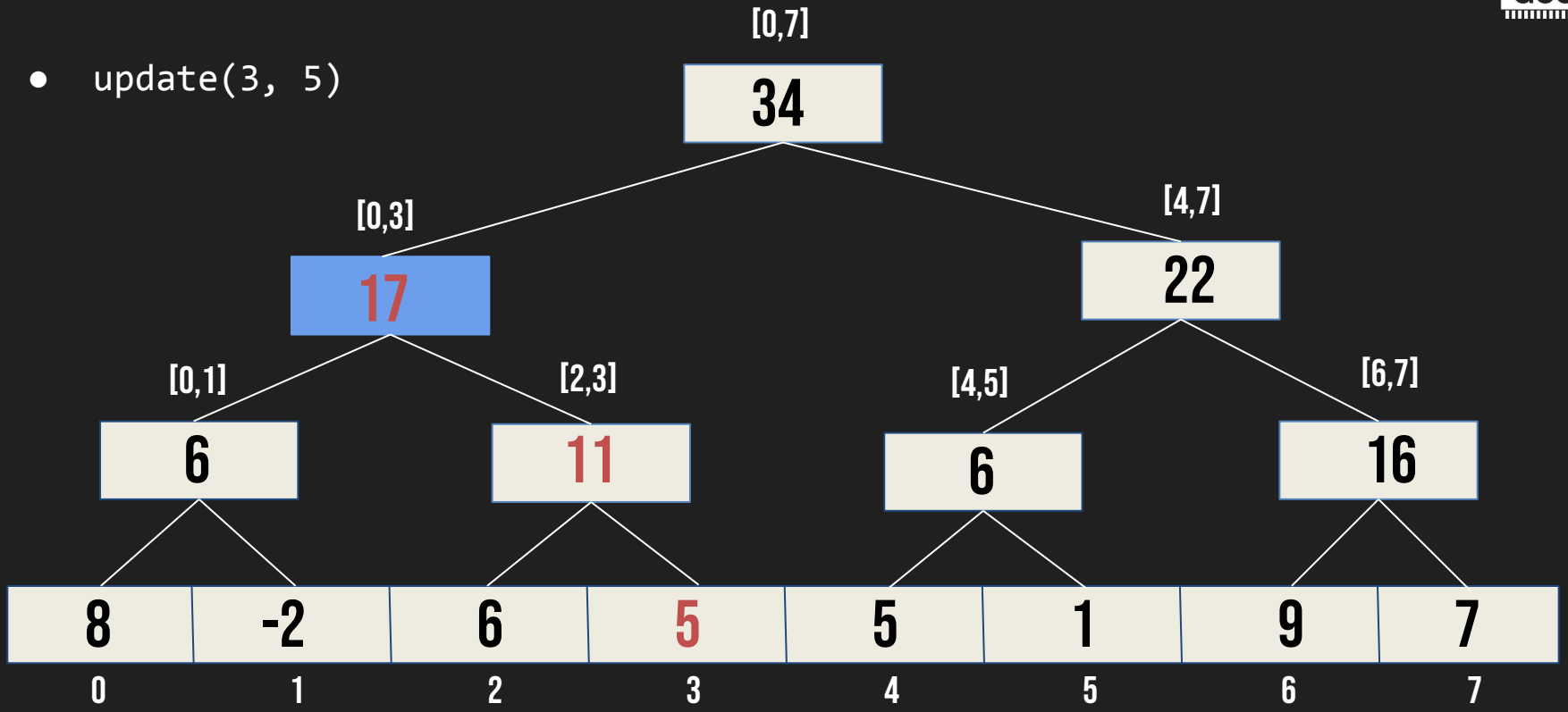
- update(3, 5)



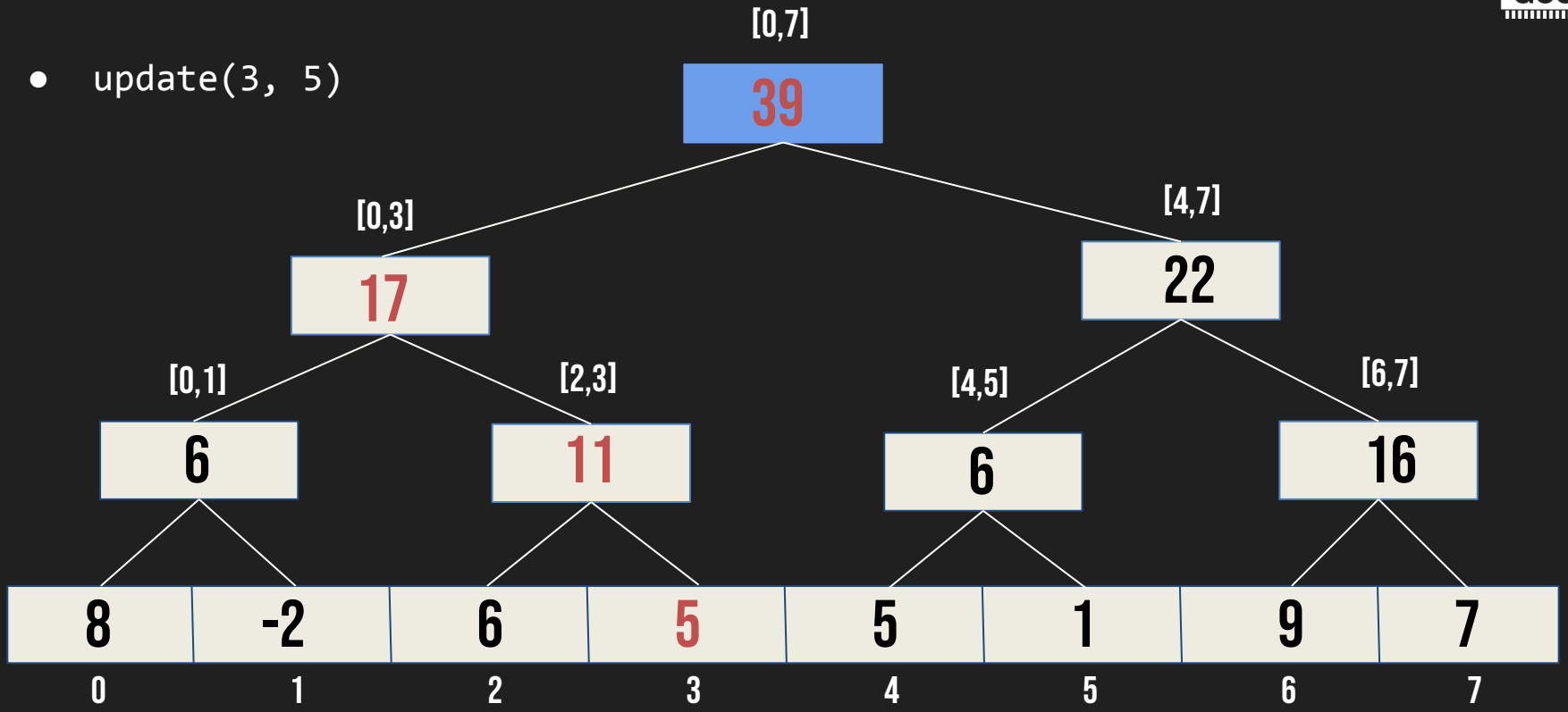
- `update(3, 5)`



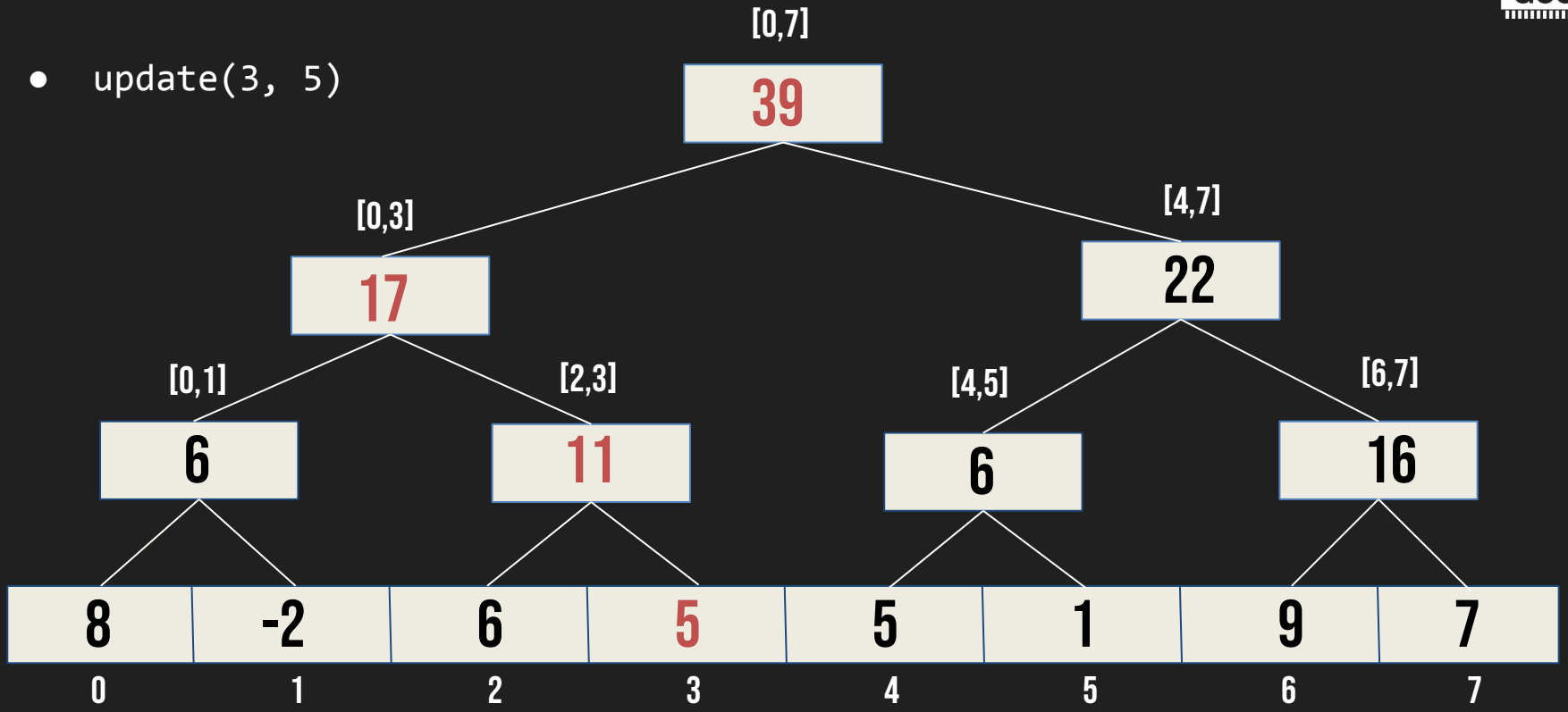
- `update(3, 5)`



- `update(3, 5)`



- `update(3, 5)`



08 - ALGORITMO

08 - ALGORITMO

```
void update(int node, int l, int r, int i, int x){



}
```

08 - ALGORITMO

```
void update(int node, int l, int r, int i, int x){
    if(i < l or r < i) return;




}
```

08 - ALGORITMO

```
void update(int node, int l, int r, int i, int x){
    if(i < l or r < i) return;
    if(l == r){
      tree[node] += x;
      return;
     }



}
```

08 - ALGORITMO

```
void update(int node, int l, int r, int i, int x){
    if(i < l or r < i) return;
    if(l == r){
      tree[node] += x;
      return;
     }
    int m = (l+r)/2;



}
```

08 - ALGORITMO

```
void update(int node, int l, int r, int i, int x){
    if(i < l or r < i) return;
    if(l == r){
      tree[node] += x;
      return;
     }
    int m = (l+r)/2;
    update(2*node, l, m, i, x);
    update(2*node+1, m+1, r, i, x);

}
```

08 - ALGORITMO

```
void update(int node, int l, int r, int i, int x){
    if(i < l or r < i) return;
    if(l == r){
      tree[node] += x;
      return;
     }
    int m = (l+r)/2;
    update(2*node, l, m, i, x);
    update(2*node+1, m+1, r, i, x);
    tree[node] = tree[2*node] + tree[2*node+1];
}
```

08 - ALGORITMO

```
void update(int node, int l, int r, int i, int x){
    if(i < l or r < i) return;
    if(l == r){
       tree[node] += x;
       return;
     }
    int m = (l+r)/2;
    update(2*node, l, m, i, x);
    update(2*node+1, m+1, r, i, x);
    tree[node] = tree[2*node] + tree[2*node+1];
}
update(1, 0, n-1, i, x);
```

08 - ALGORITMO

```
void update(int node, int l, int r, int i, int x){
    if(i < l or r < i) return;
    if(l == r){
       tree[node] += x;
       return;
     }
    int m = (l+r)/2;
    update(2*node, l, m, i, x);
    update(2*node+1, m+1, r, i, x);
    tree[node] = tree[2*node] + tree[2*node+1];
}
update(1, 0, n-1, i, x); // gastamos O(log n) para atualizar
```

09 - LAZY PROPAGATION

- E se quisermos mudar todos os elementos do array do intervalo [a...b] para x?
- Note que, só conseguimos mudar uma posição do array por $O(logn)$, mas para mudar um intervalo teríamos que gastar $O(nlogn)$.
- A técnica de Lazy Propagation permite fazer o update em intervalos com $O(logn)$, sendo assim uma melhoria do update da SegTree básica que aprendemos.
- A ideia é atualizar um nó apenas quando a informação sobre aquele nó for necessária.

09 - LAZY PROPAGATION

Como funcionará a função de propagação:

1. **Verificação de pendência**: Verifica se há uma atualização pendente no nó atual.
2. **Atualização do nó**: O valor da árvore (tree[node]) é ajustado com base no valor pendente e no tamanho do intervalo `[l...r]`.
3. **Propagação para os filhos**: Se o nó não é folha, os valores pendentes são passados para os nós filhos (esquerdo e direito).
4. **Limpar a pendência**: Após propagar, a pendência do nó atual é removida.

09 - LAZY PROPAGATION

Quando queremos atualizar um intervalo [a…b] com um valor, existem três casos a considerar:

- **Intervalo Parcialmente Dentro**:
  - Se o intervalo [a…b] **se sobrepõe parcialmente** ao intervalo do nó, chamamos a função de propagação e percorrermos recursivamente aos seus filhos.

09 - LAZY PROPAGATION

Quando queremos atualizar um intervalo [a…b] com um valor, existem três casos a considerar:

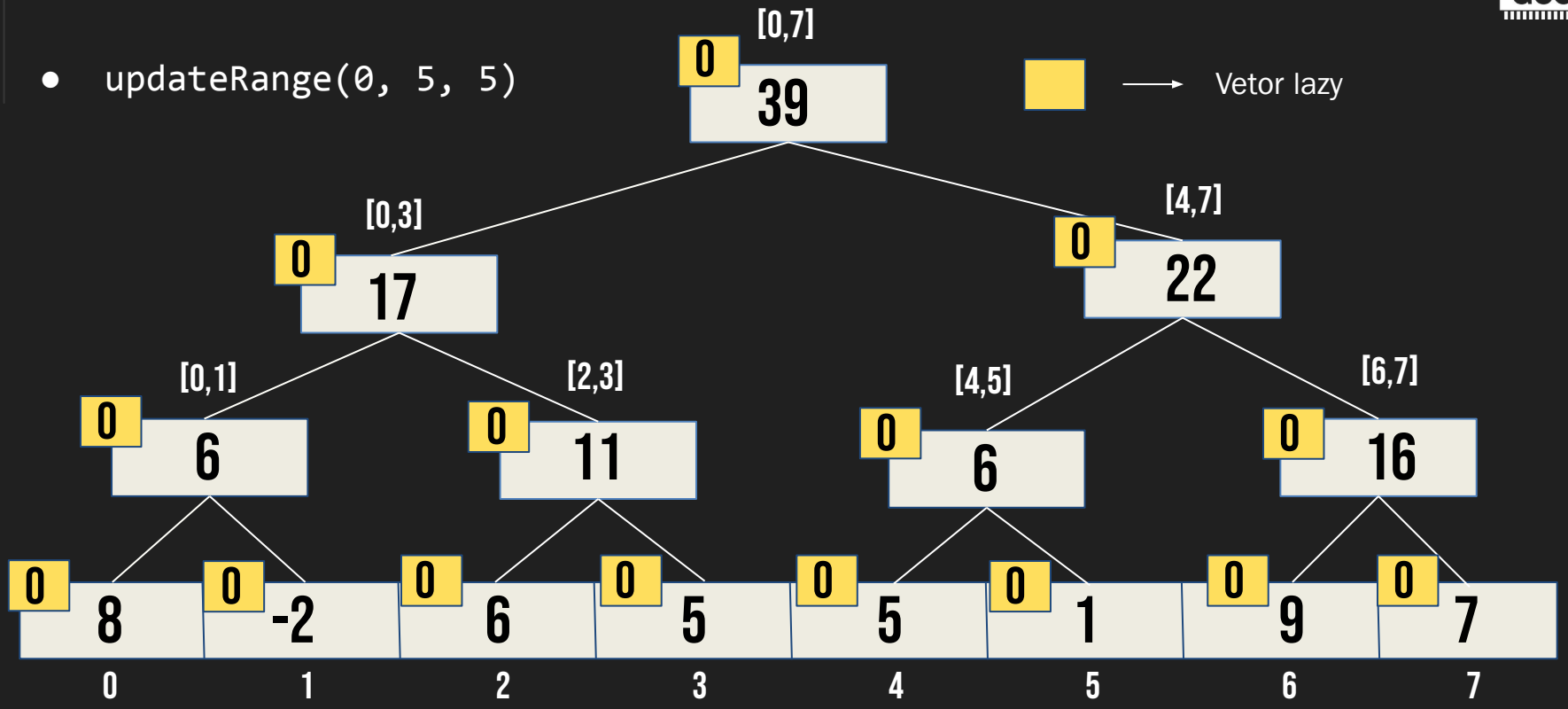
- **Intervalo Totalmente Dentro**:
  - Se o intervalo [a…b] está **totalmente contido** no intervalo do nó atual, aplicamos a operação ao nó e marcamos seus filhos no vetor lazy para **postergar a atualização**.
  - A ideia é que os filhos só sejam atualizados quando for necessário consultá-los ou atualizá-los diretamente no futuro.

09 - LAZY PROPAGATION

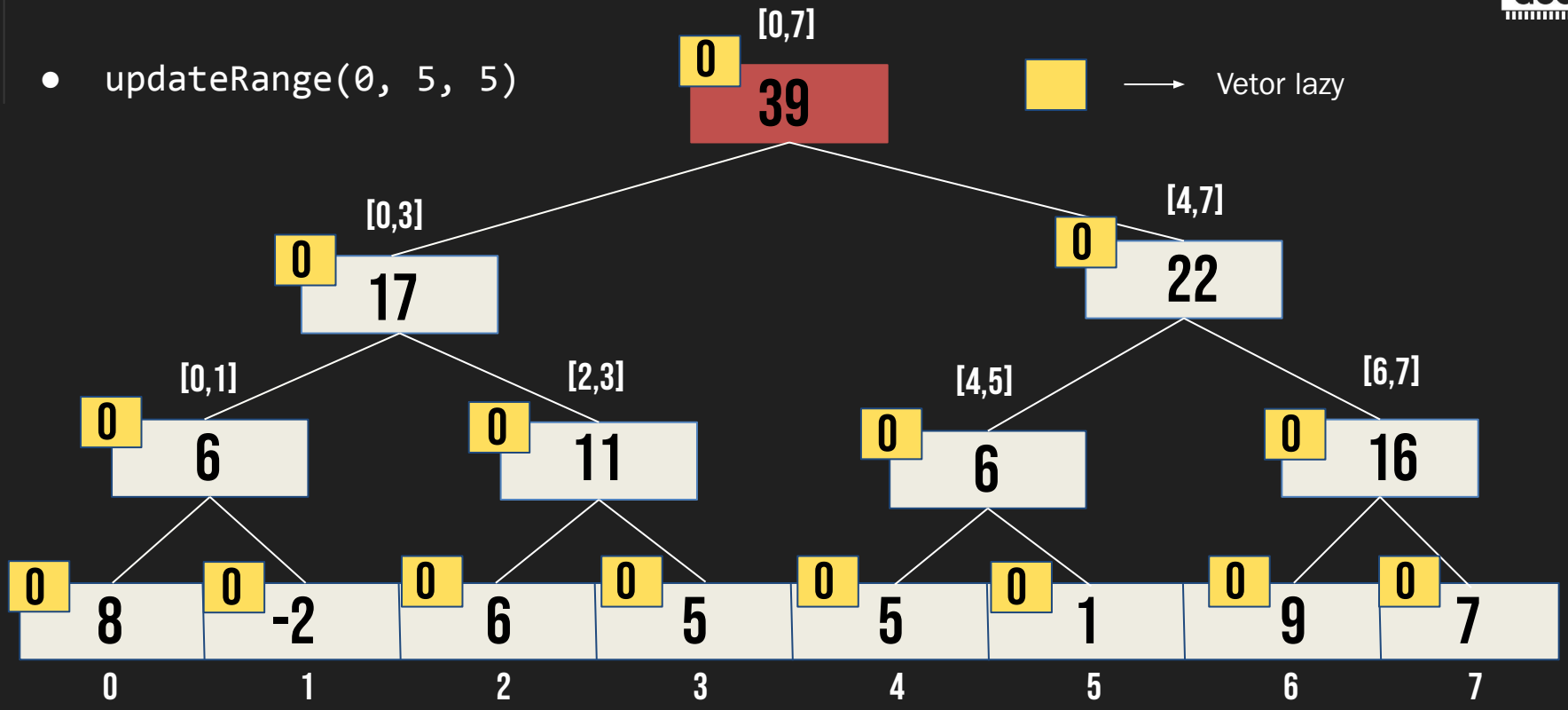
Quando queremos atualizar um intervalo [a…b] com um valor, existem três casos a considerar:

- **Intervalo Fora**:
  - Se o intervalo [a…b] está **fora** do intervalo representado pelo nó, chamamos a função de propagação e retornamos.

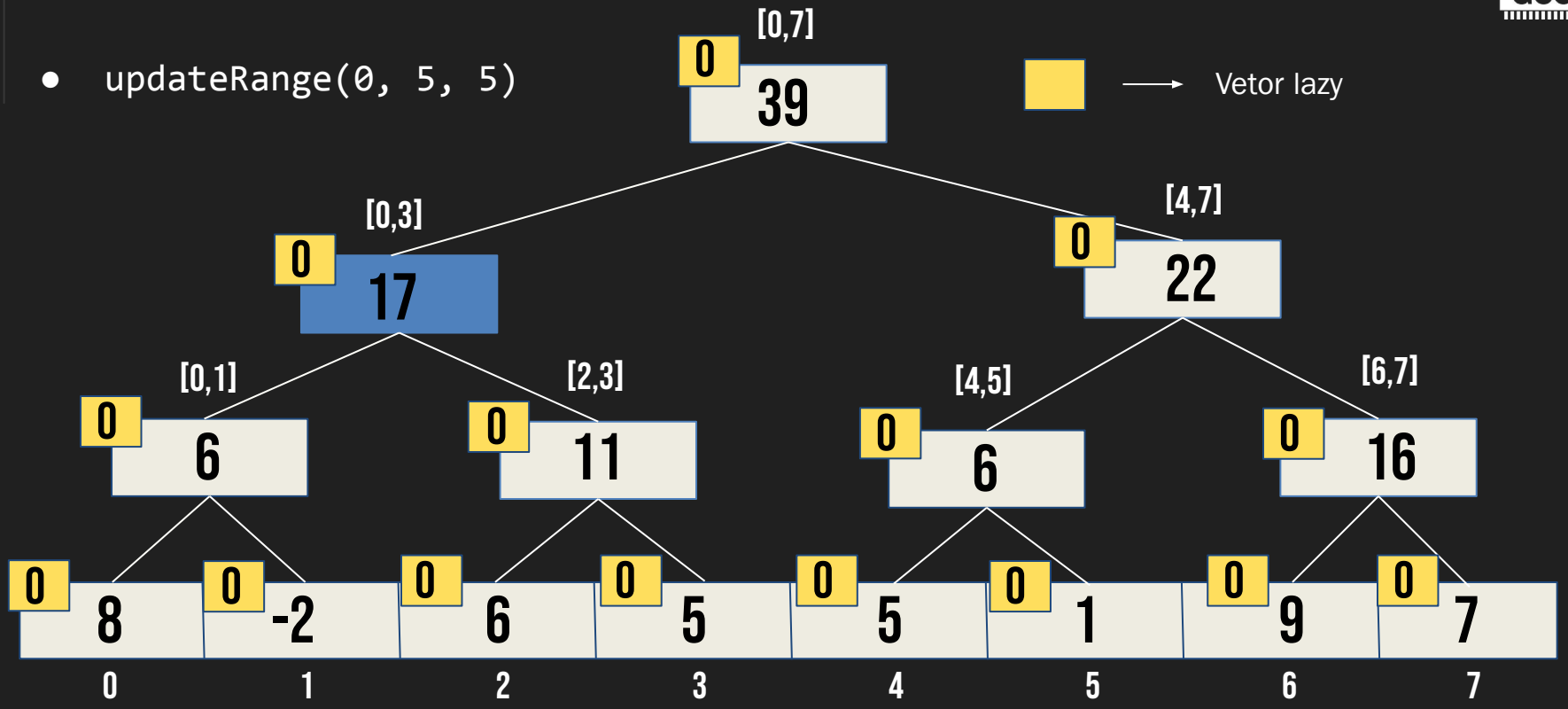
- `updateRange(0, 5, 5)`



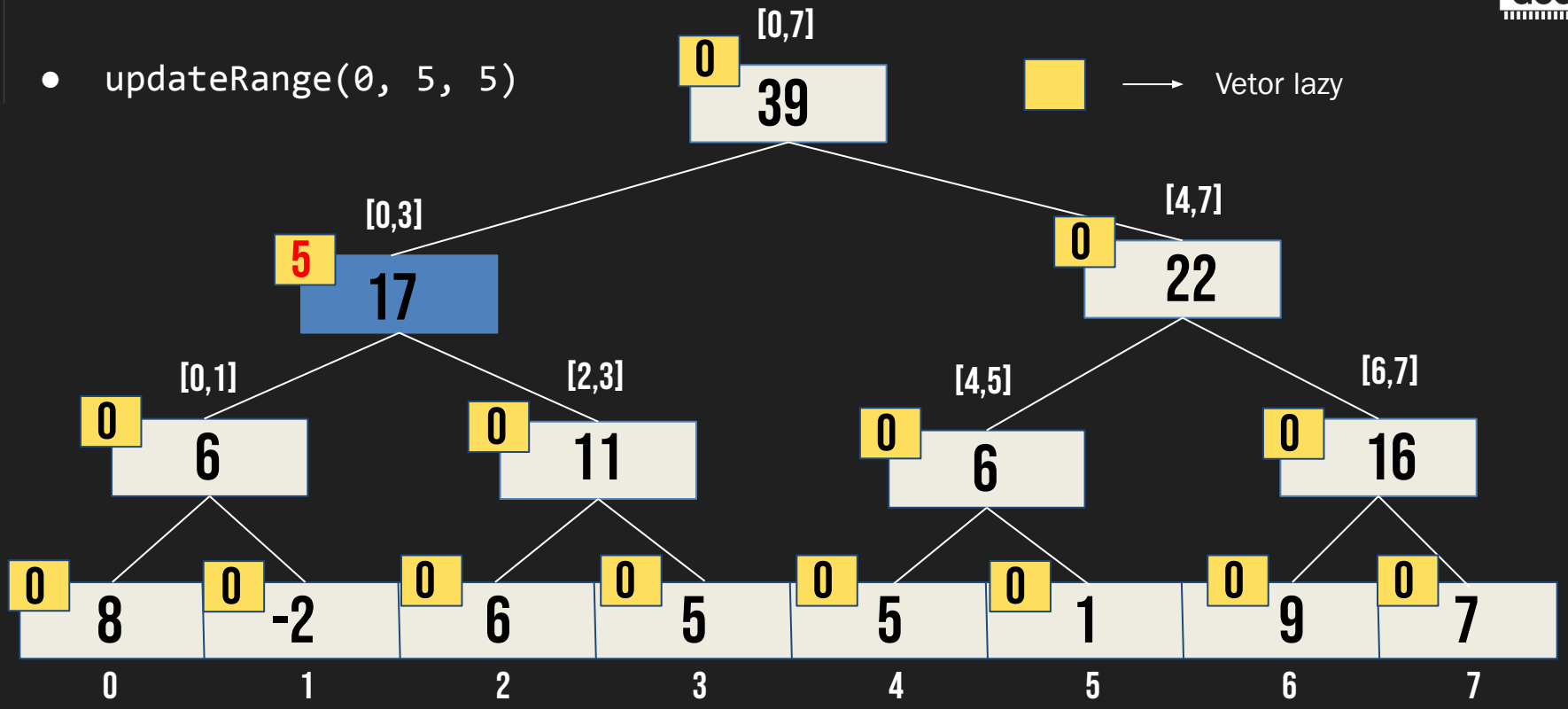
- `updateRange(0, 5, 5)`



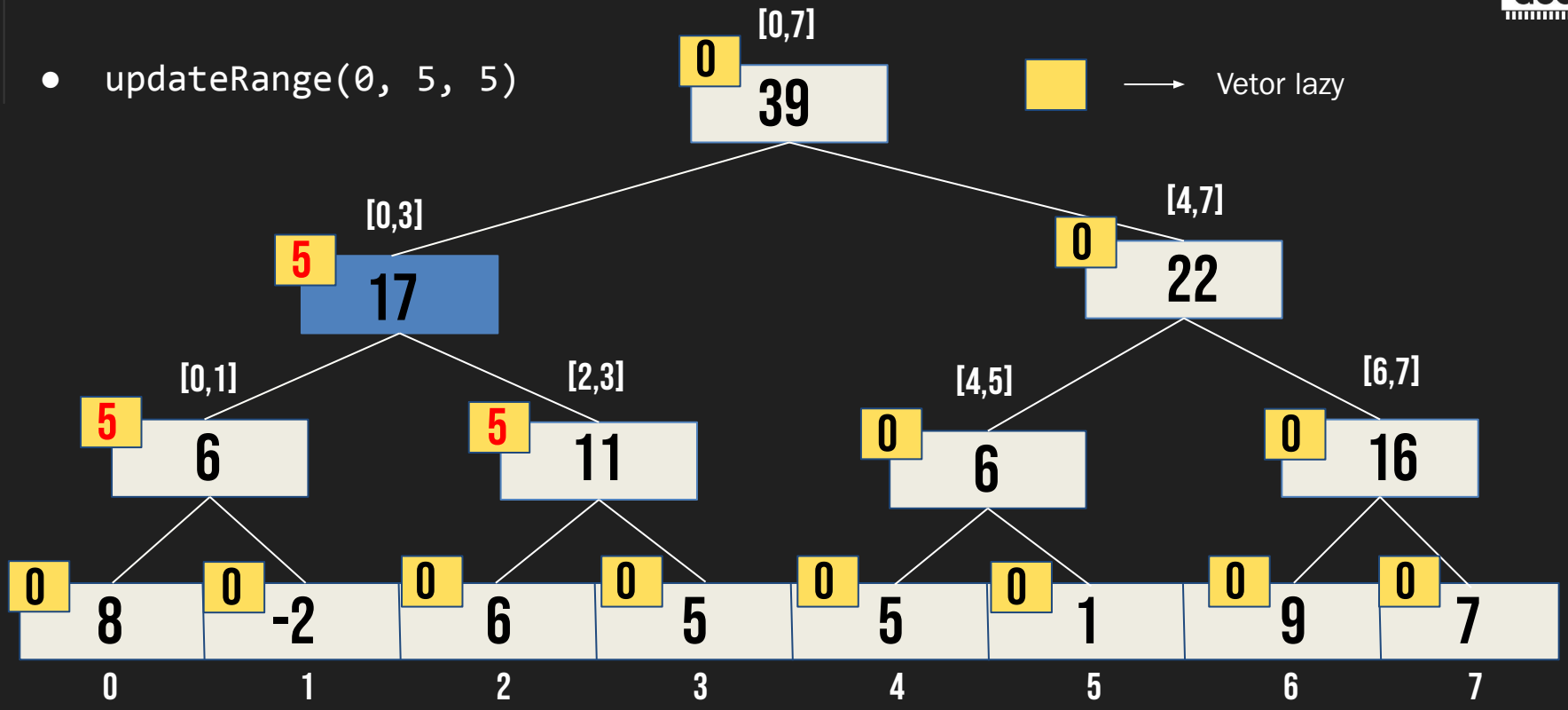
- `updateRange(0, 5, 5)`



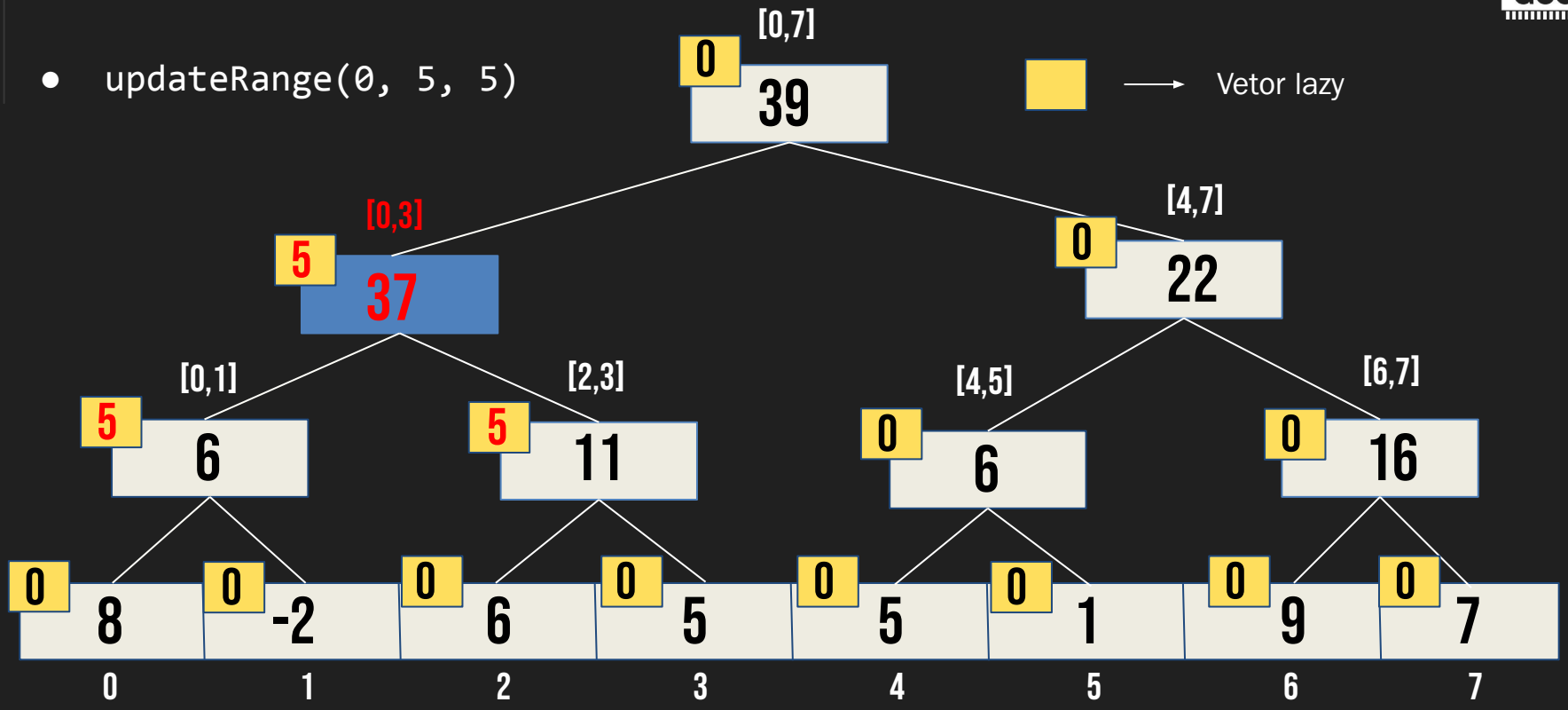
- `updateRange(0, 5, 5)`



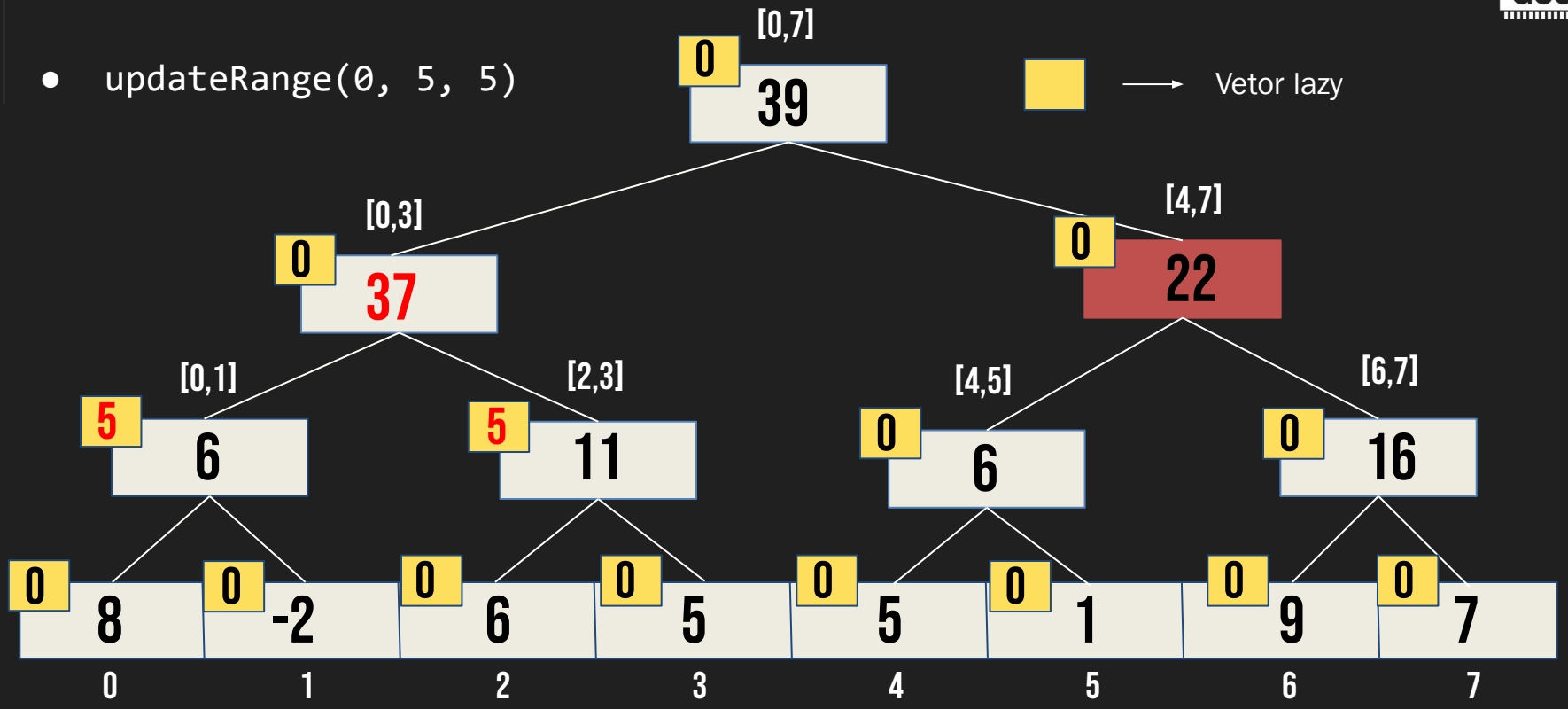
- `updateRange(0, 5, 5)`



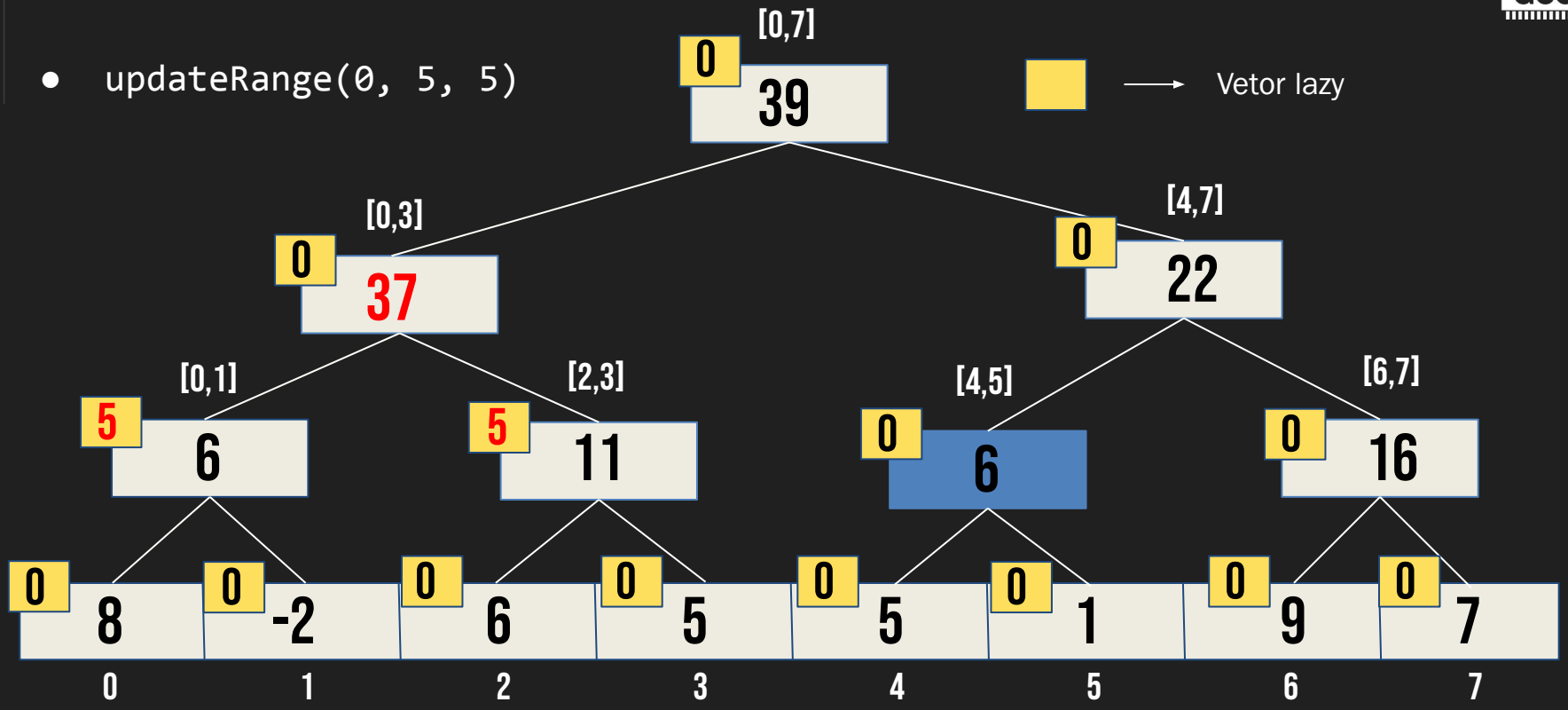
- `updateRange(0, 5, 5)`



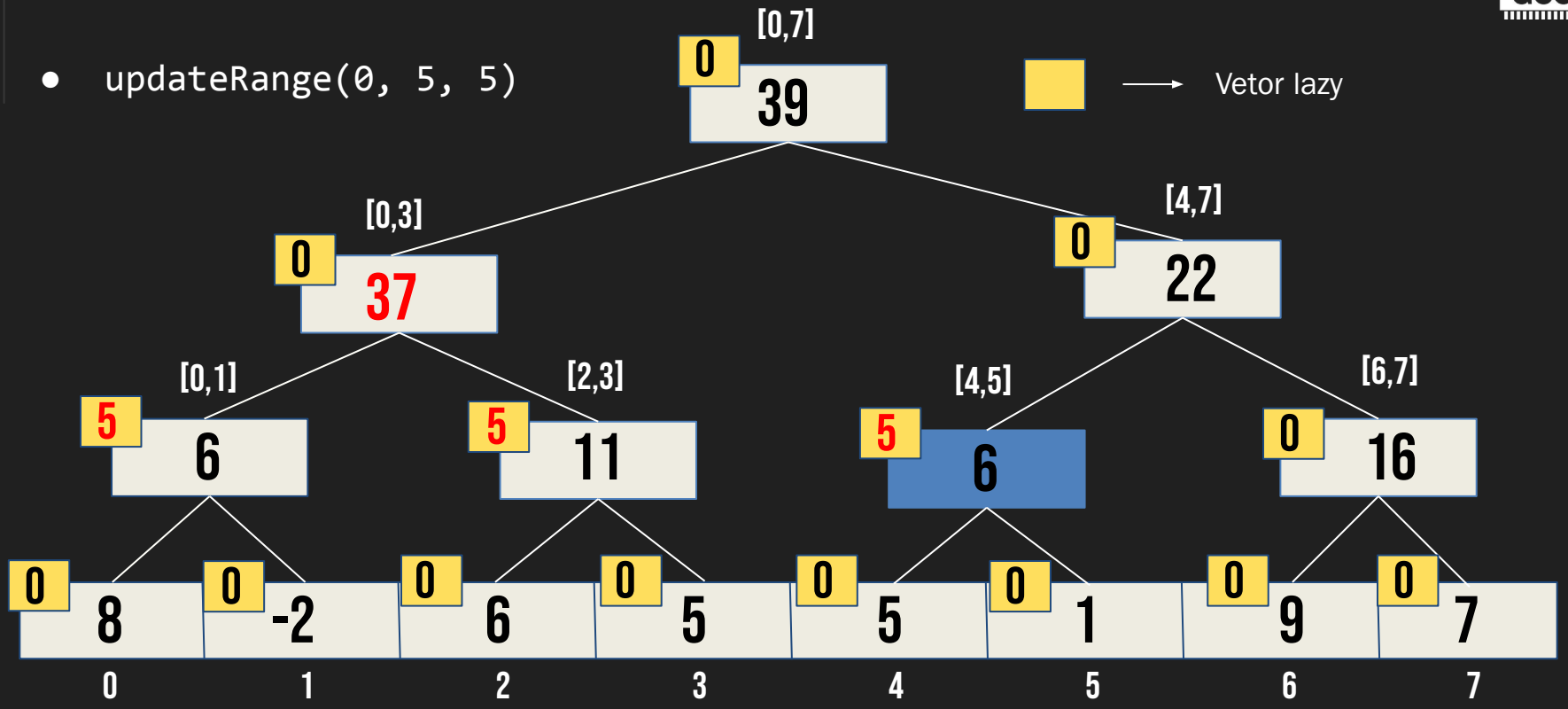
- `updateRange(0, 5, 5)`



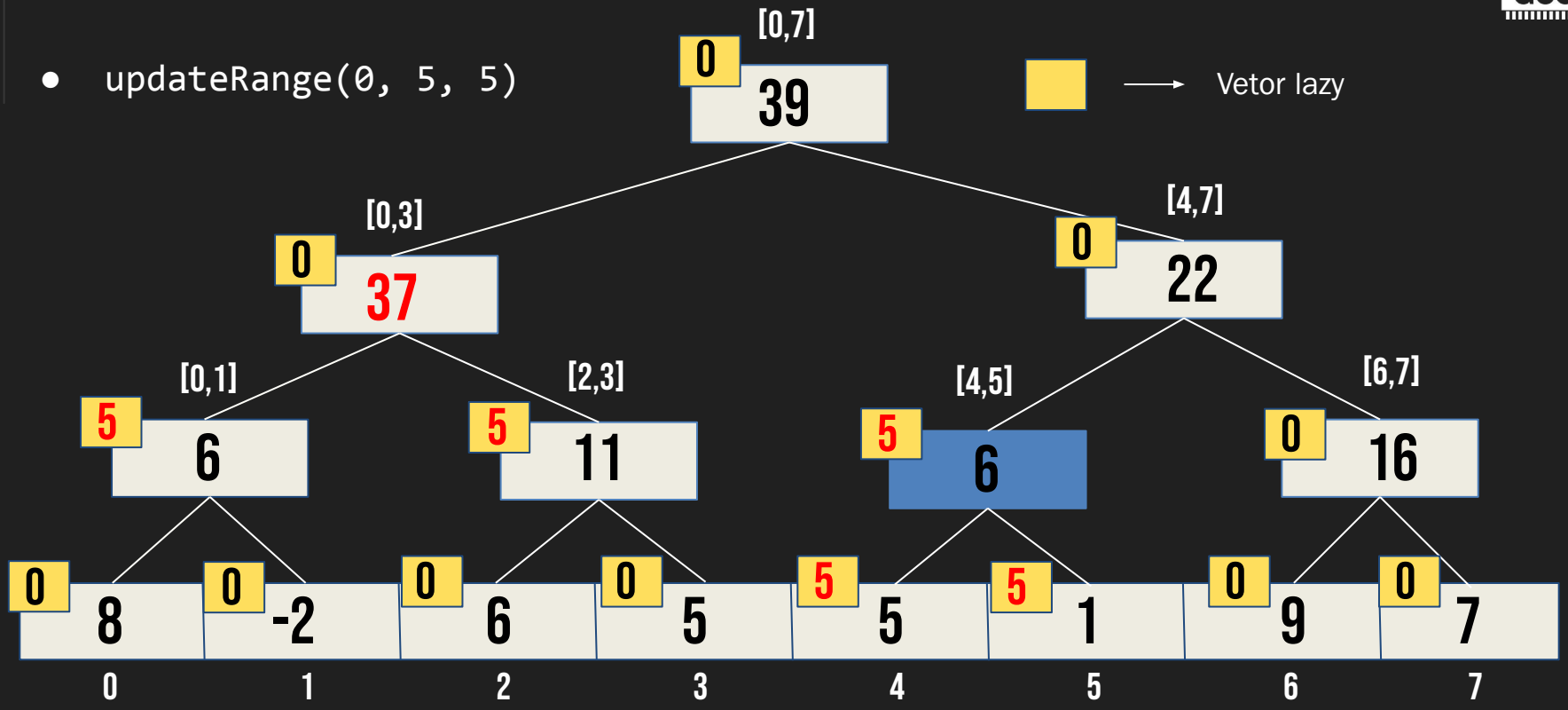
- `updateRange(0, 5, 5)`



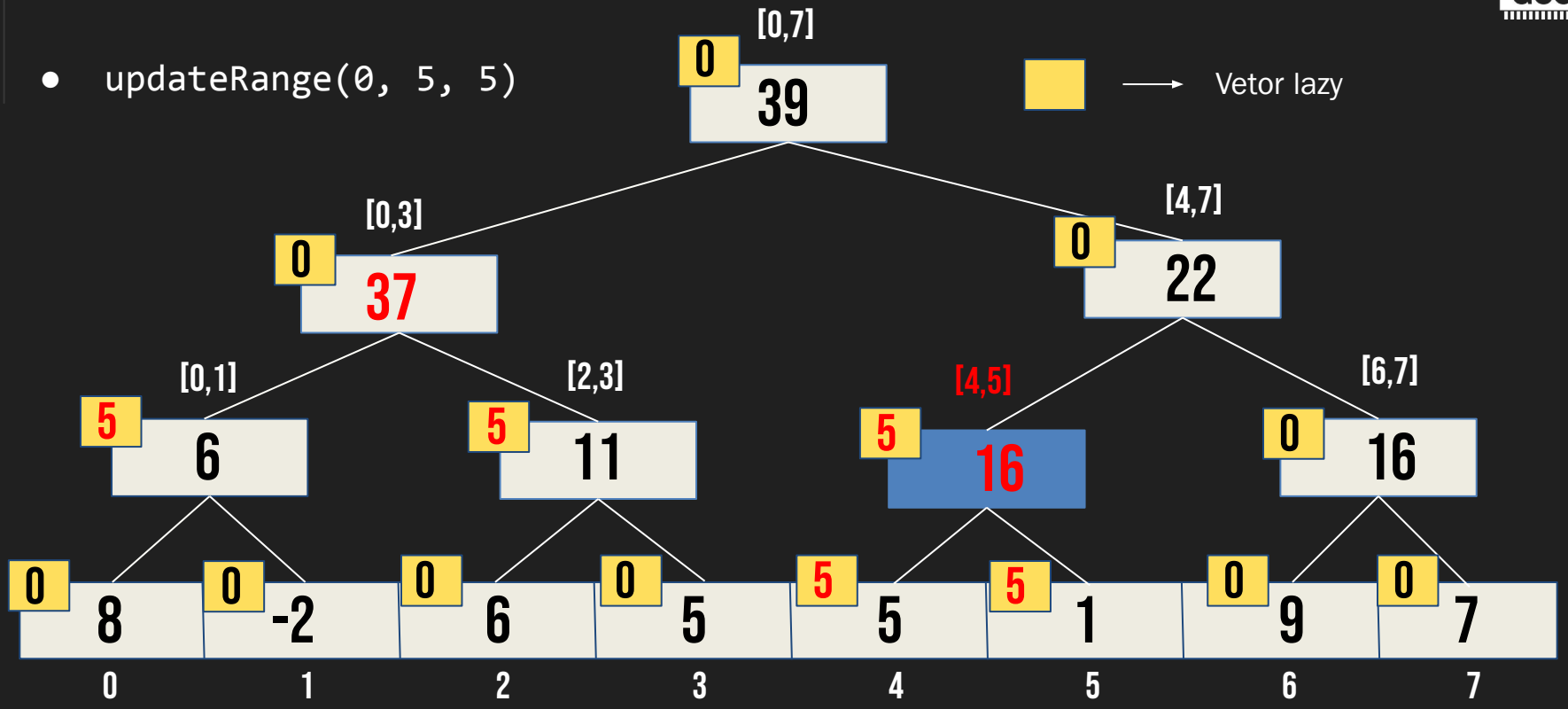
- `updateRange(0, 5, 5)`



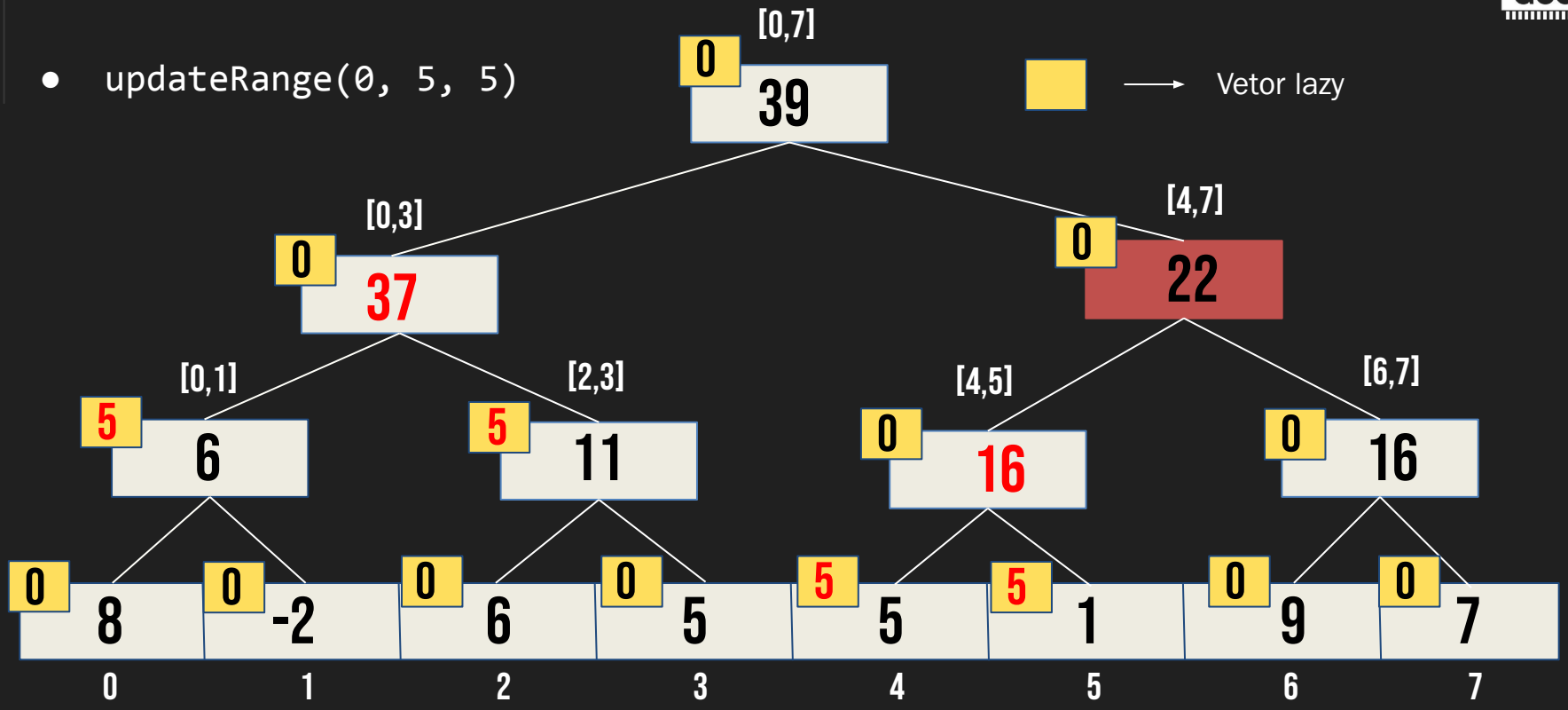
- `updateRange(0, 5, 5)`



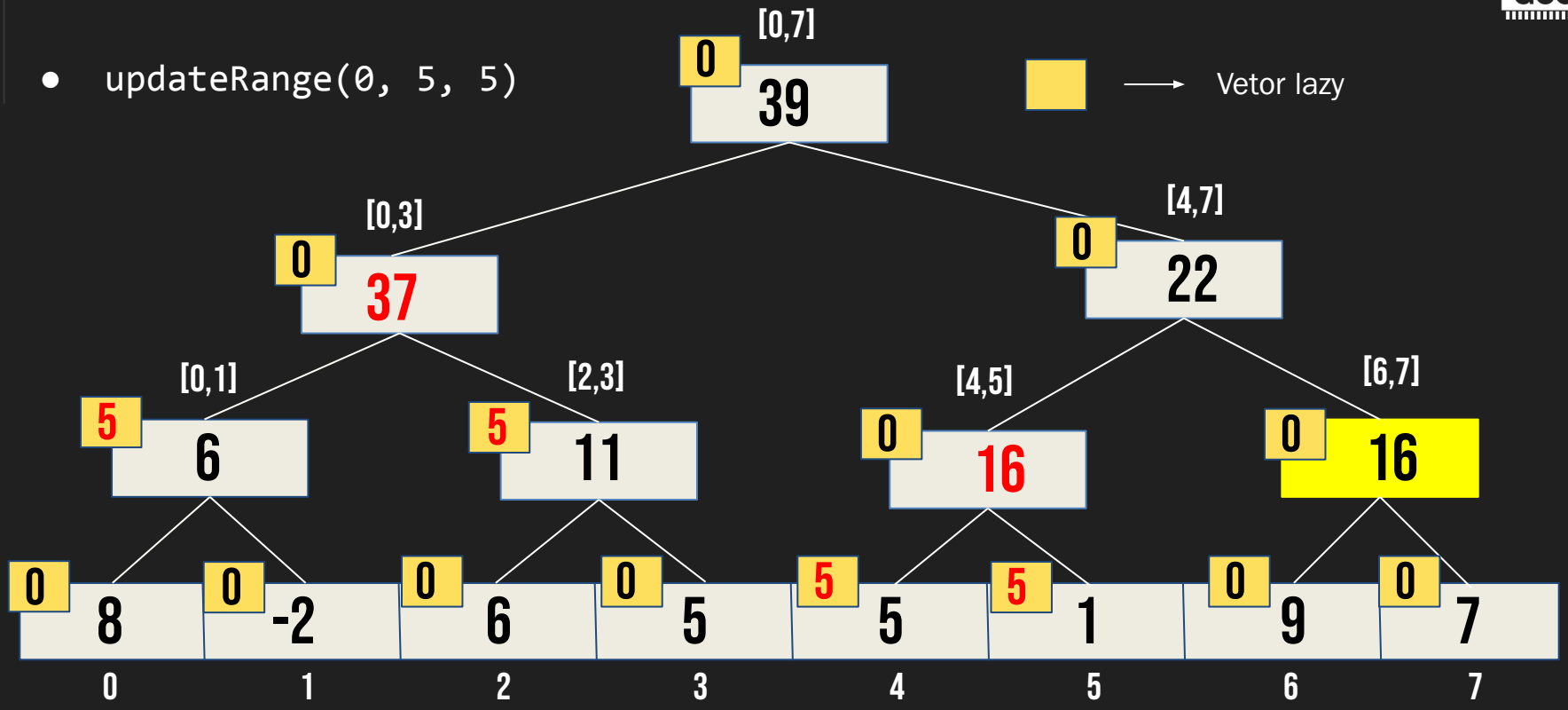
- `updateRange(0, 5, 5)`



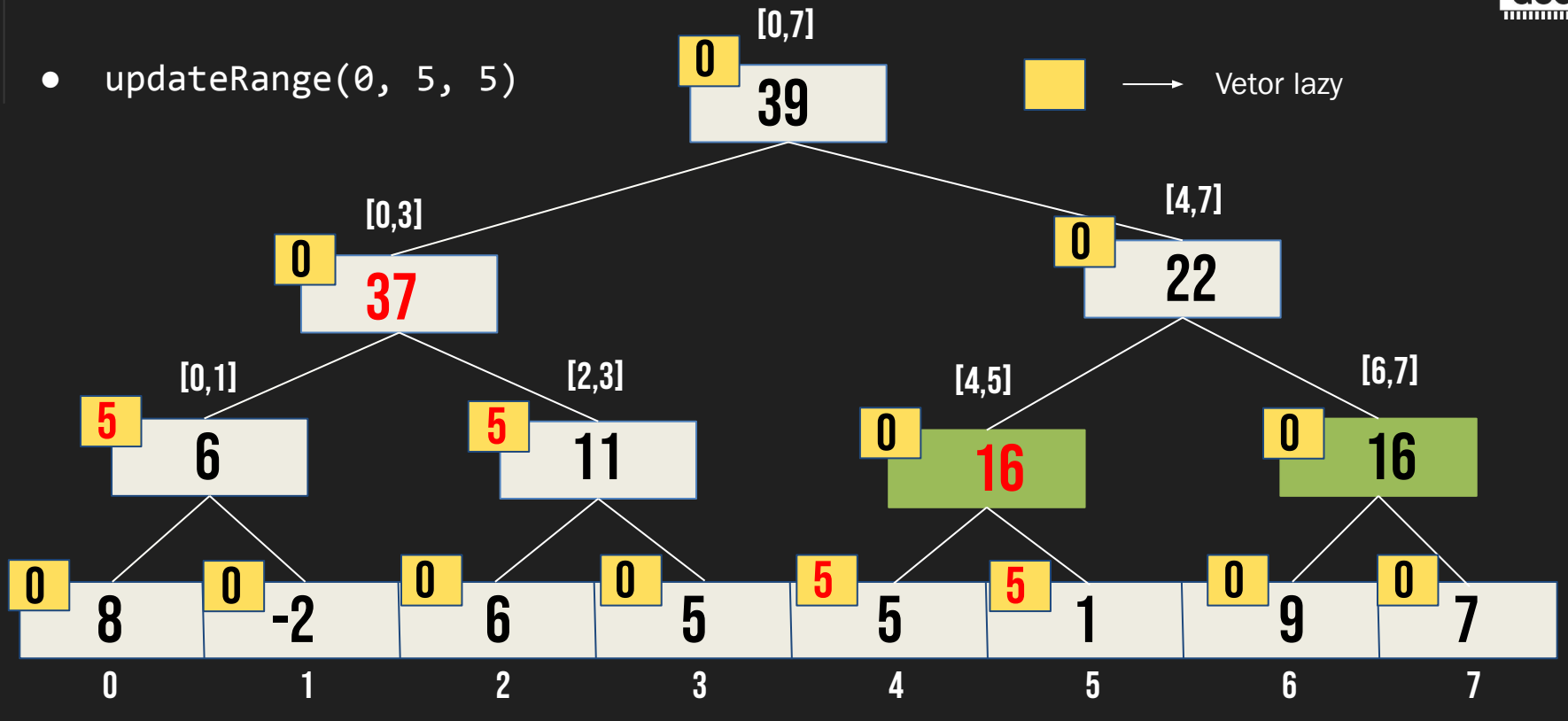
- `updateRange(0, 5, 5)`



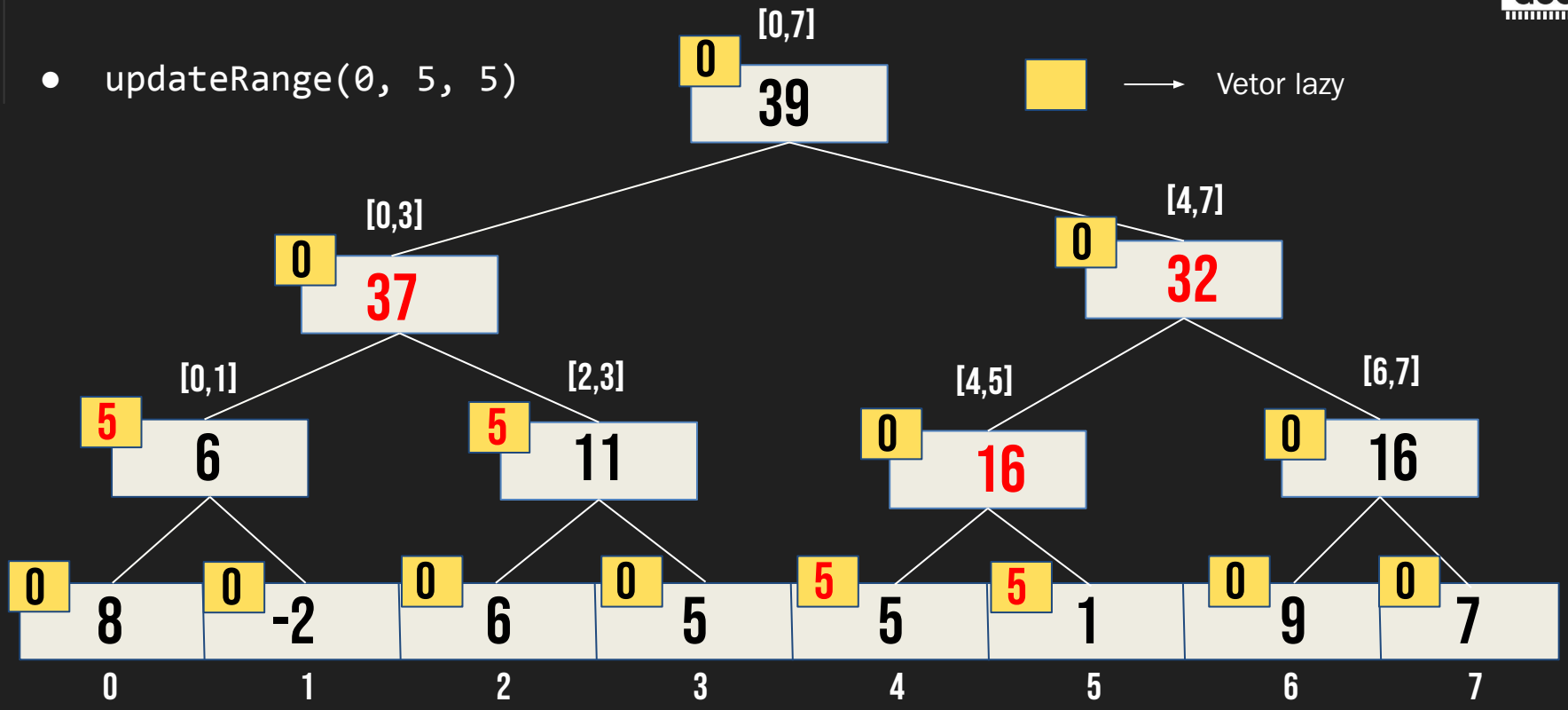
- `updateRange(0, 5, 5)`



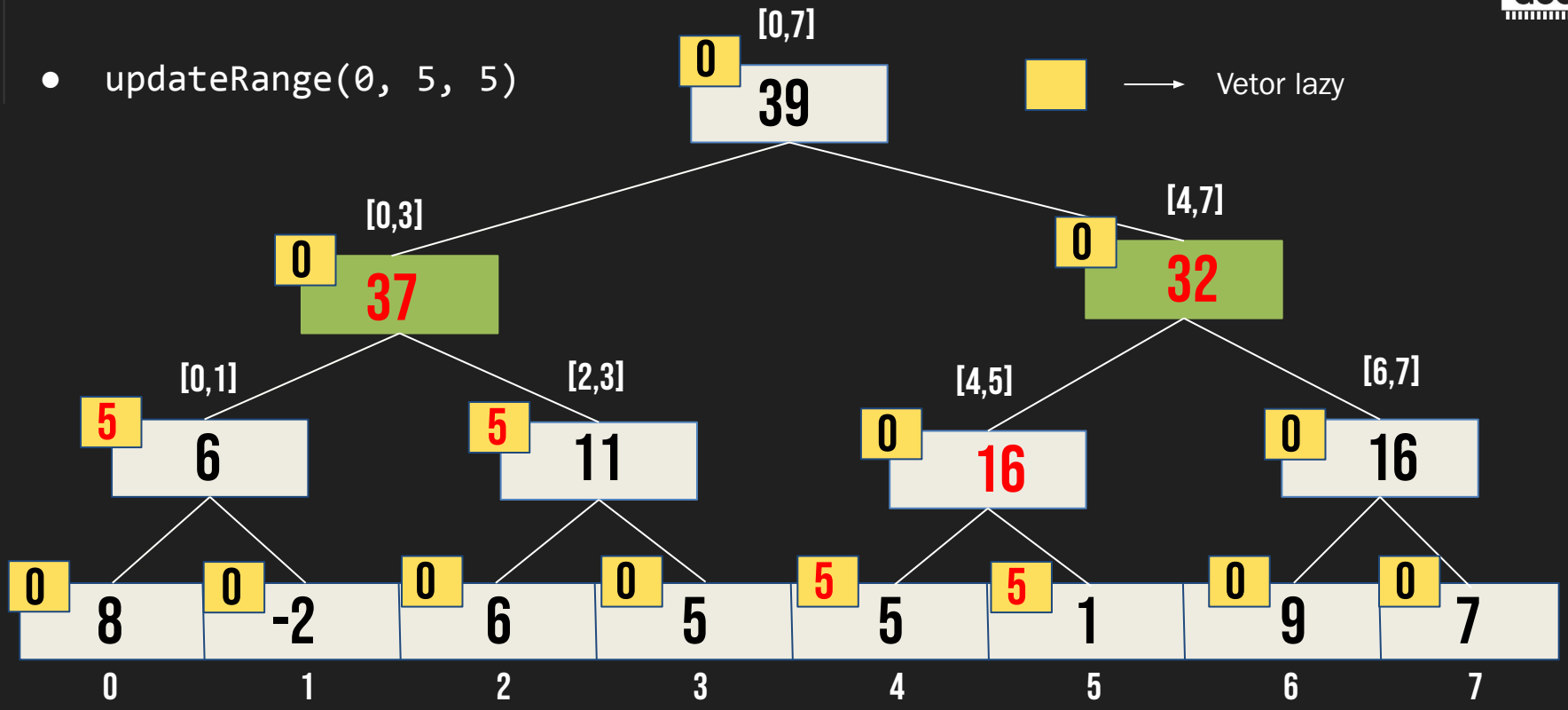
- `updateRange(0, 5, 5)`



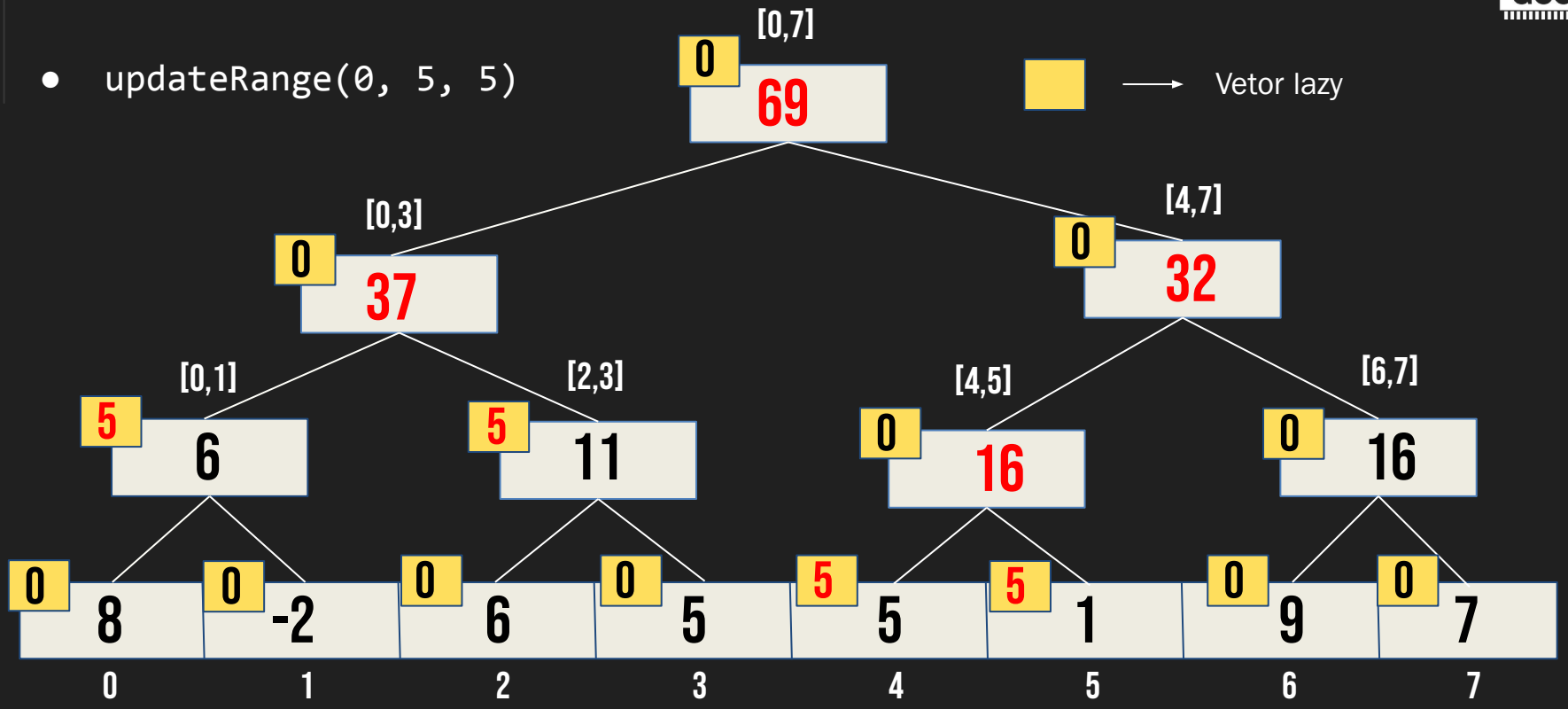
- `updateRange(0, 5, 5)`



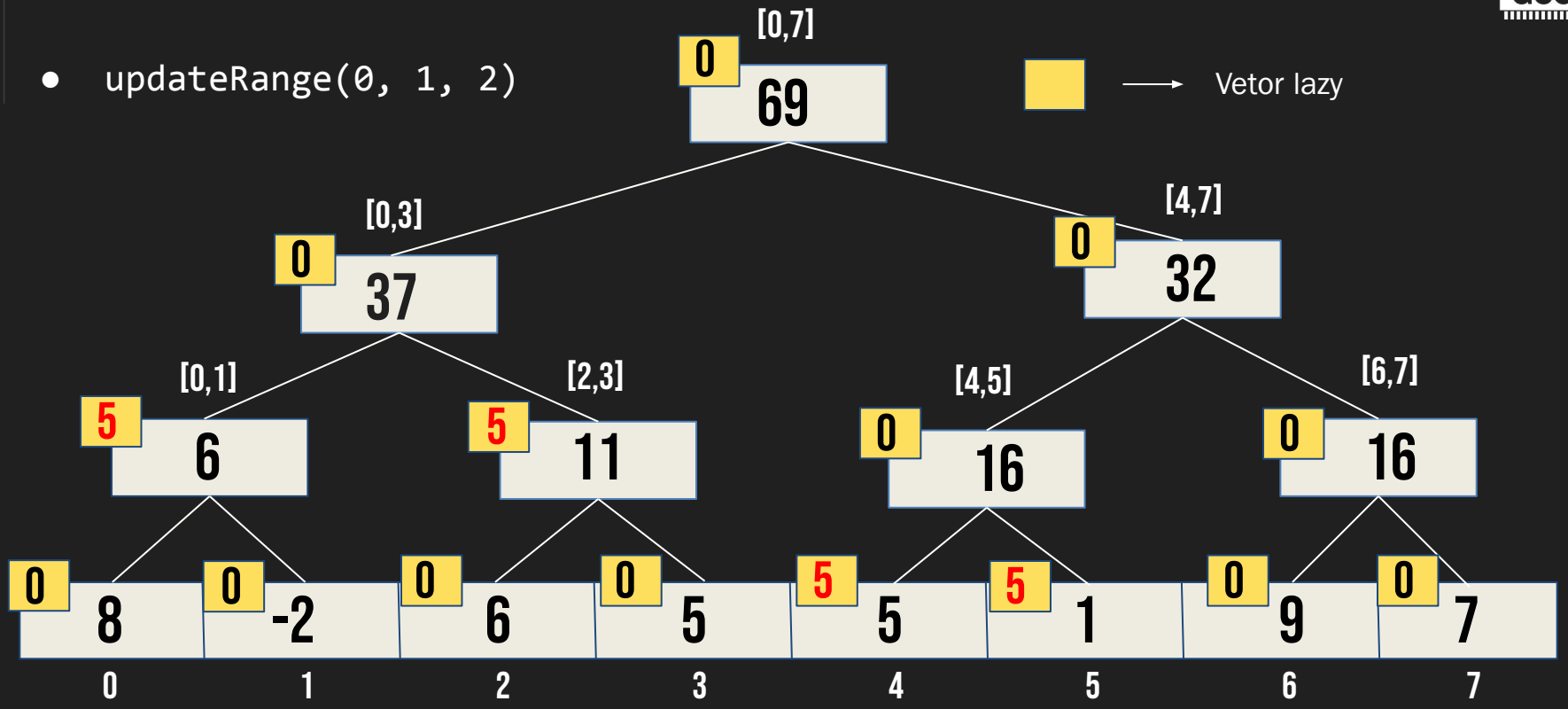
- `updateRange(0, 5, 5)`



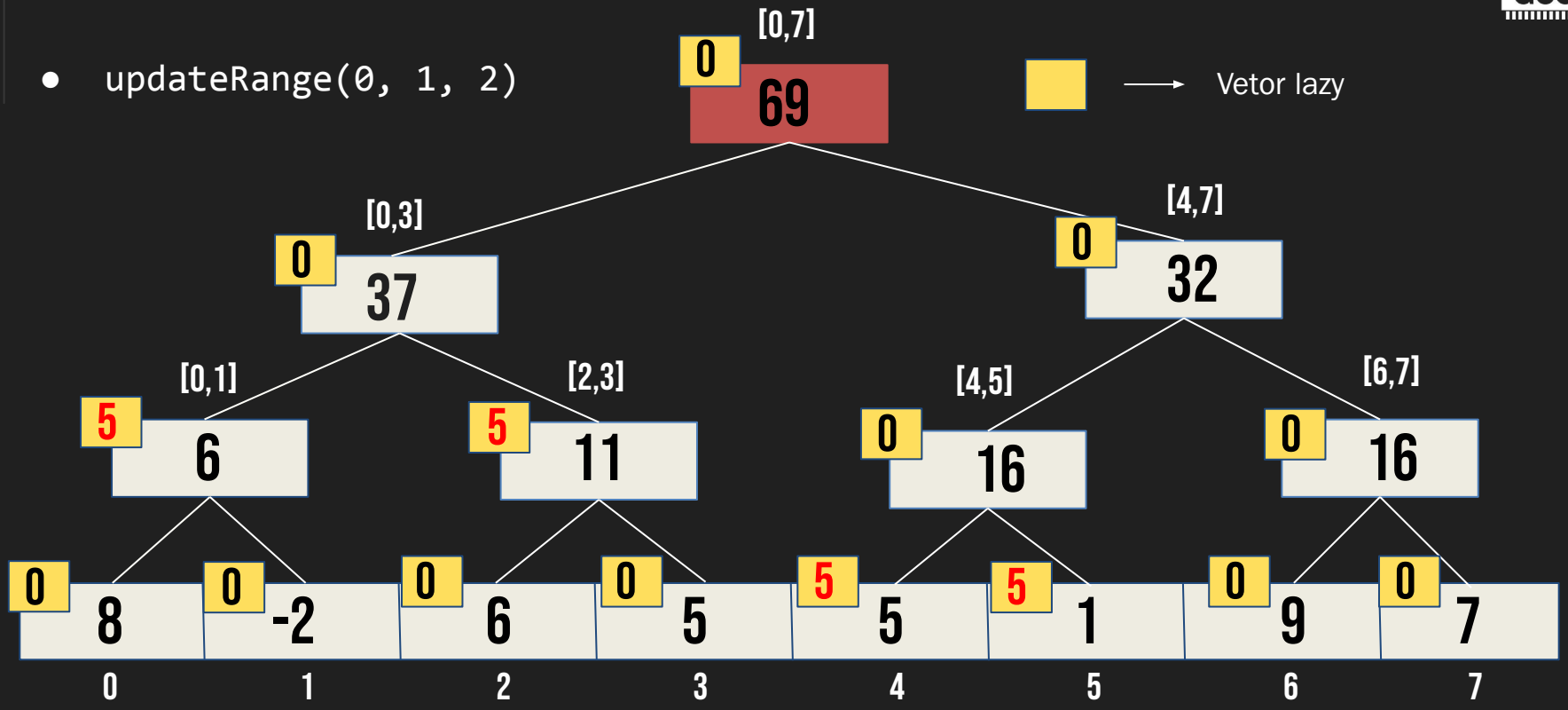
- `updateRange(0, 5, 5)`



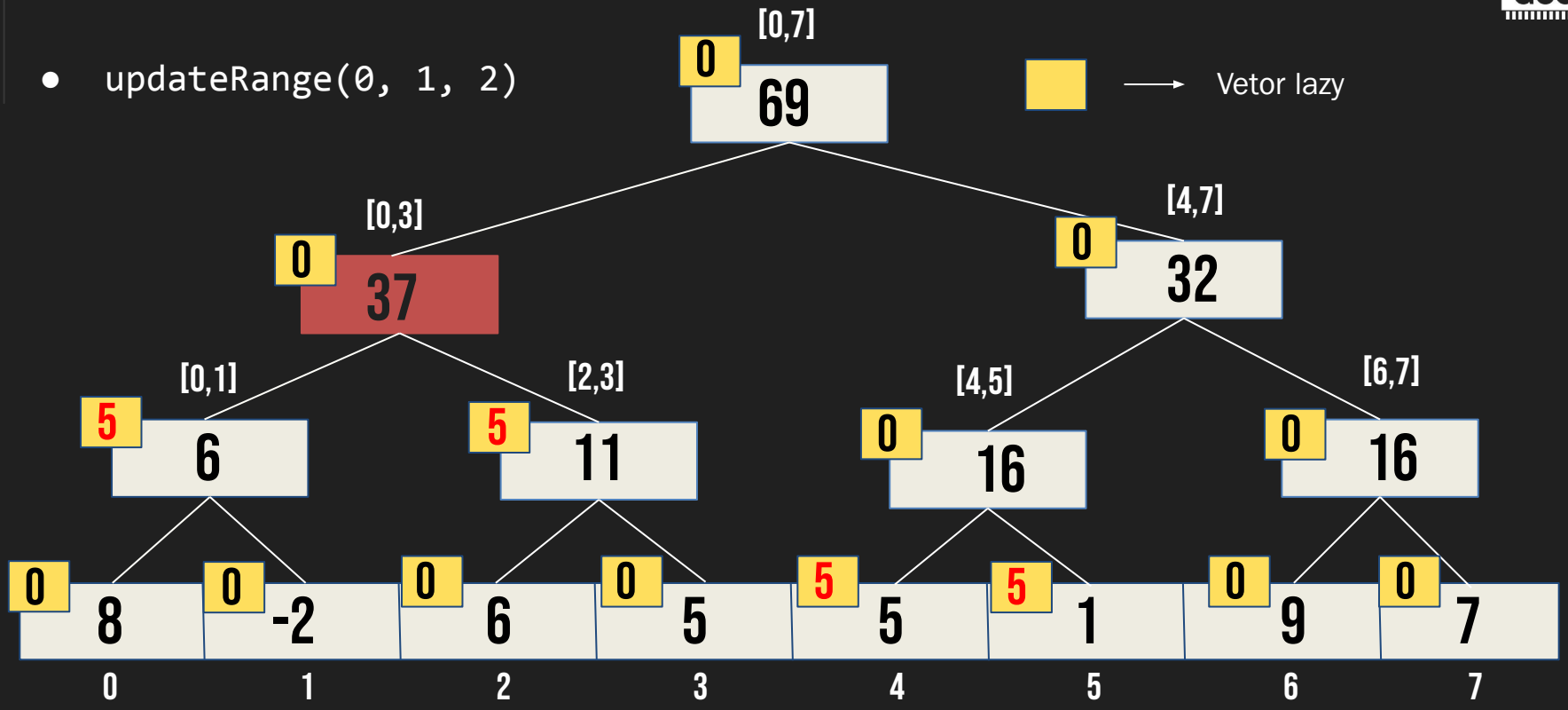
- `updateRange(0, 1, 2)`



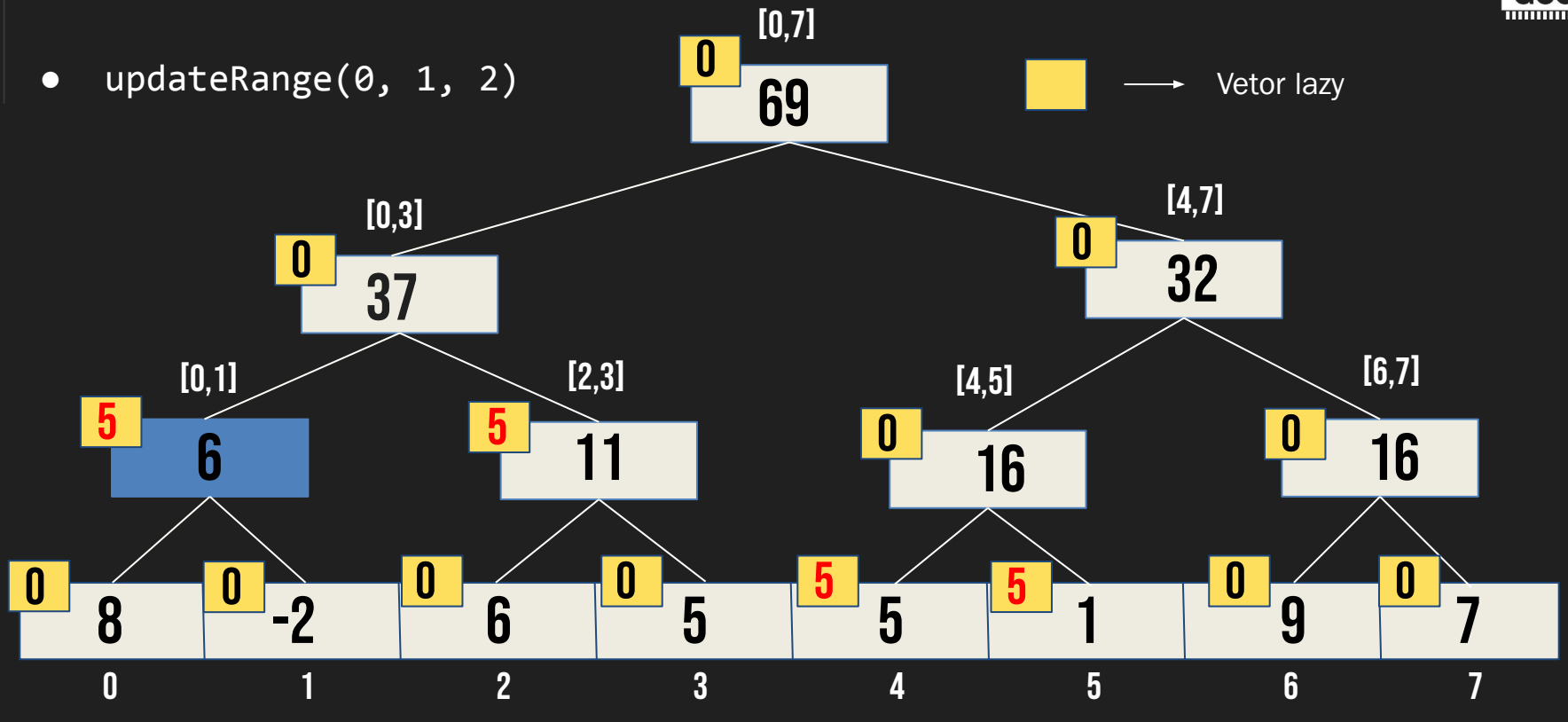
- `updateRange(0, 1, 2)`



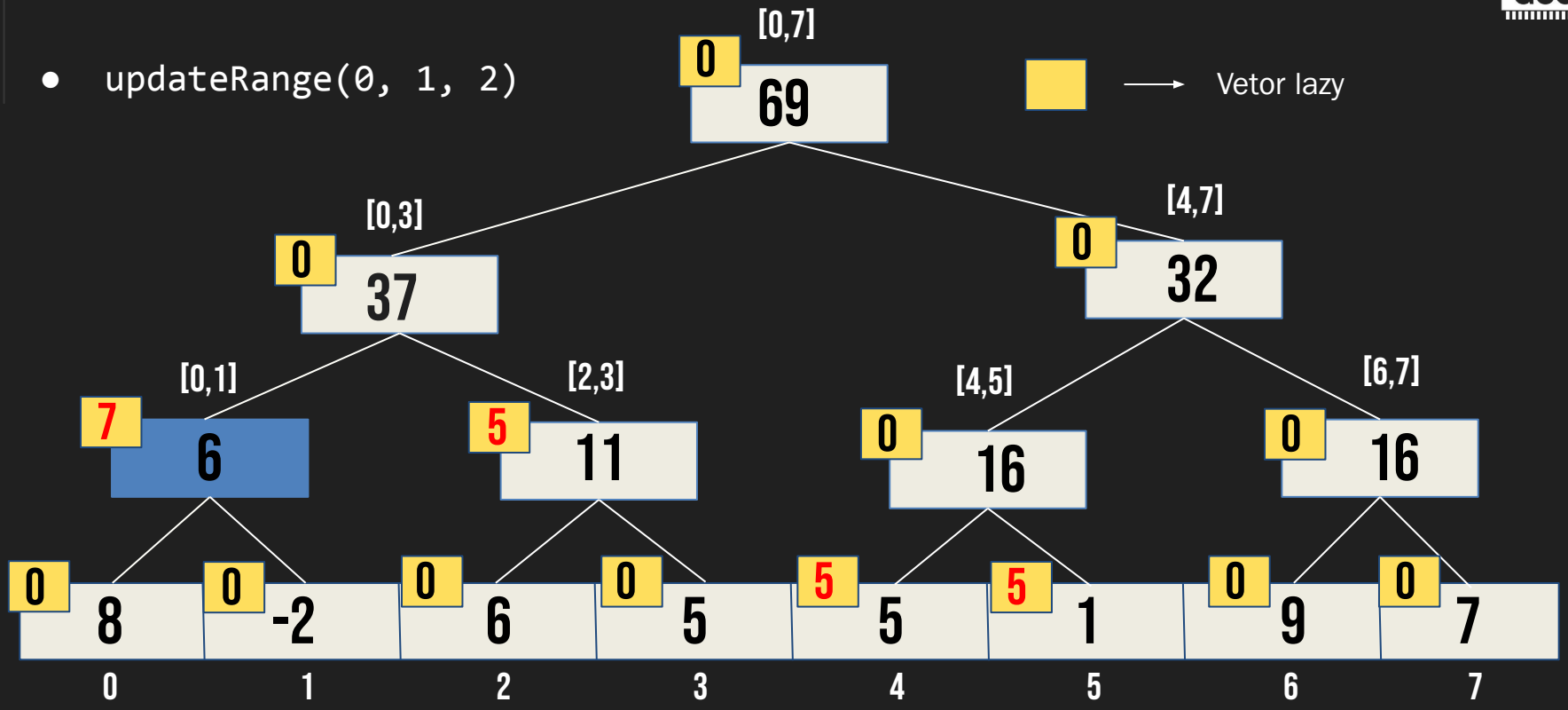
- `updateRange(0, 1, 2)`



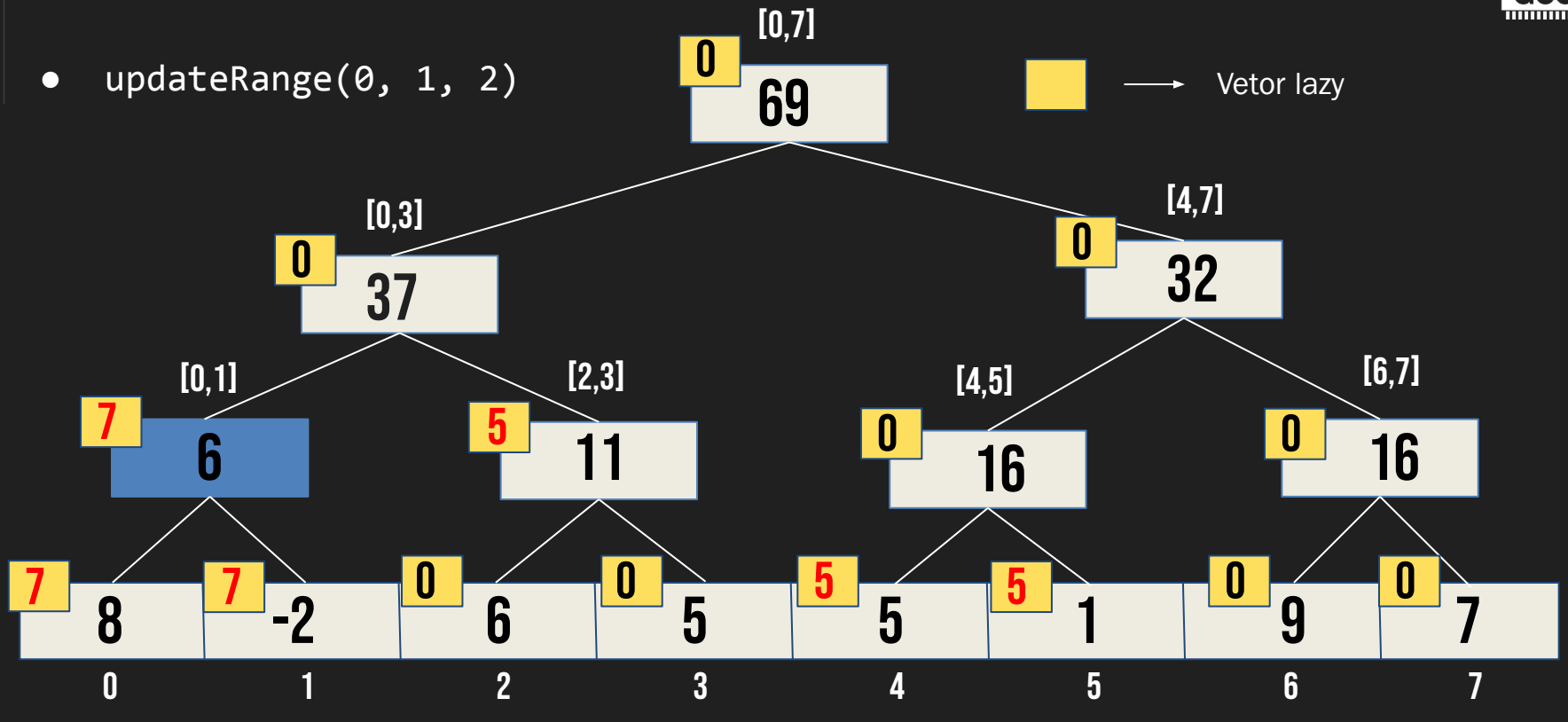
- `updateRange(0, 1, 2)`



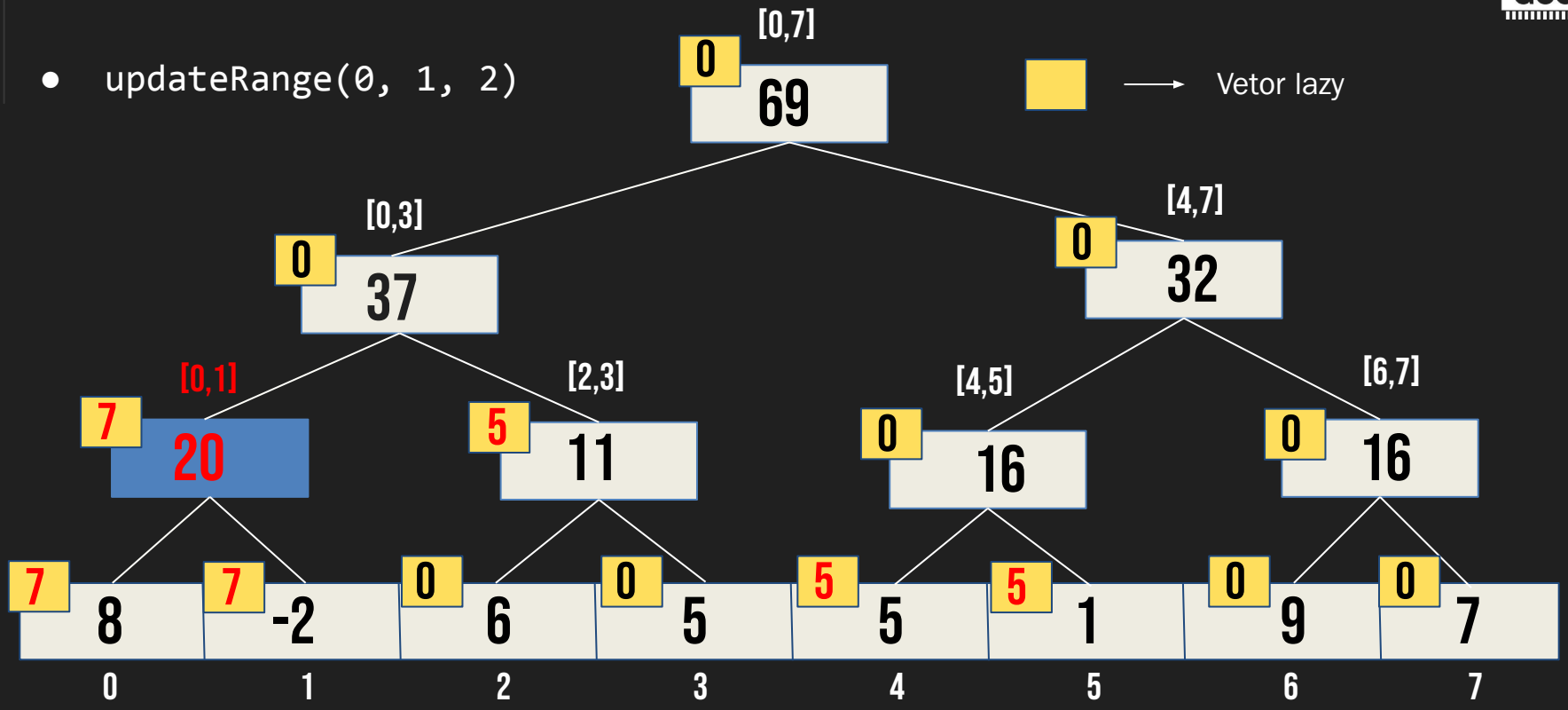
- `updateRange(0, 1, 2)`



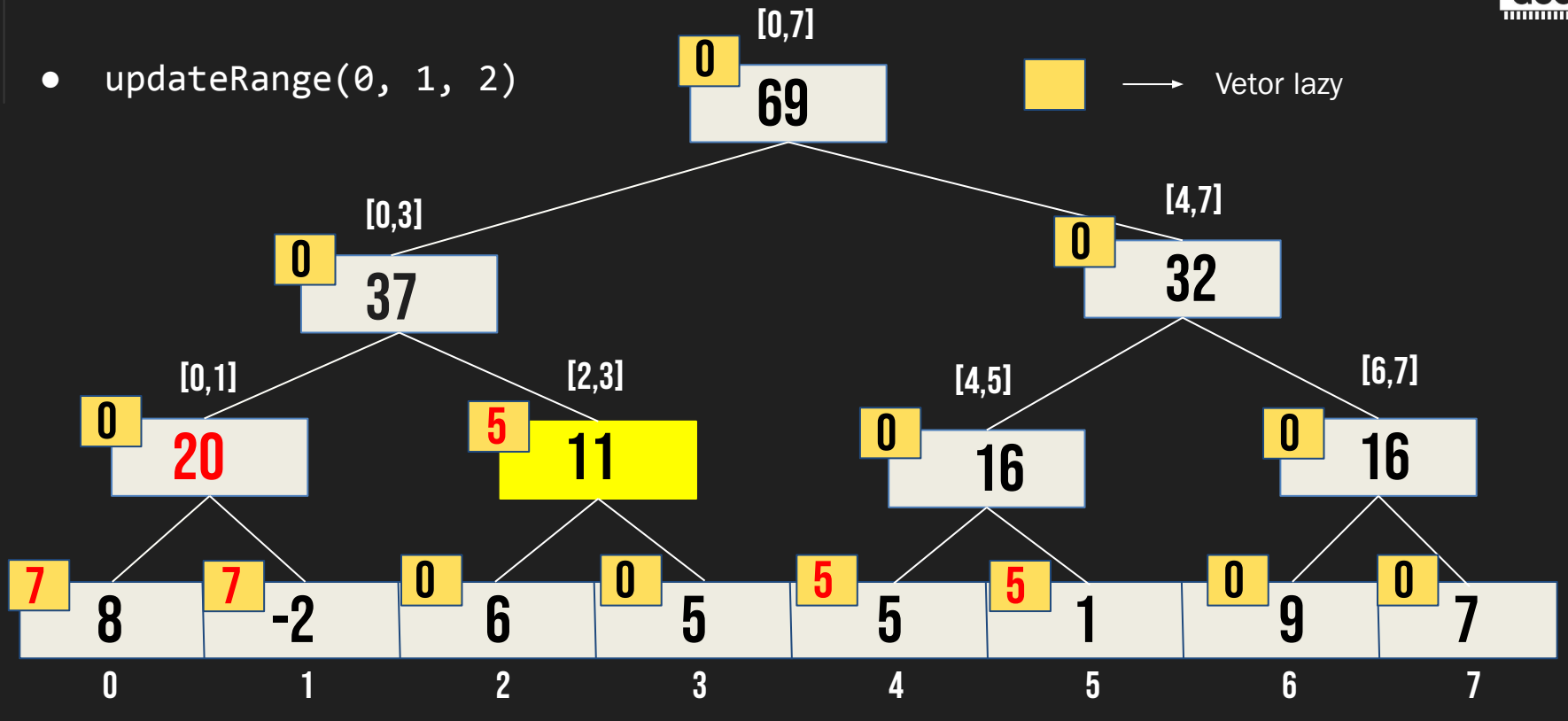
- `updateRange(0, 1, 2)`



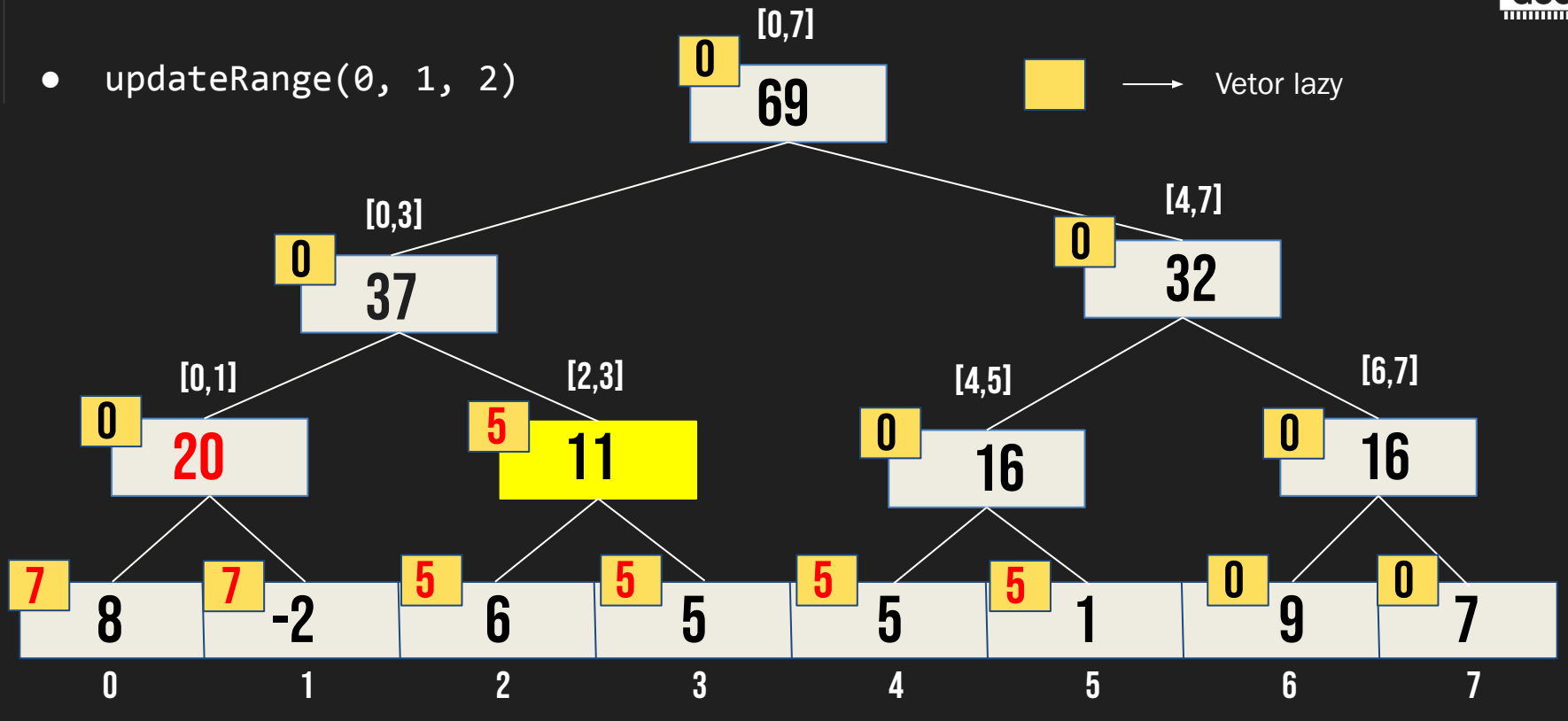
- updateRange(0, 1, 2)



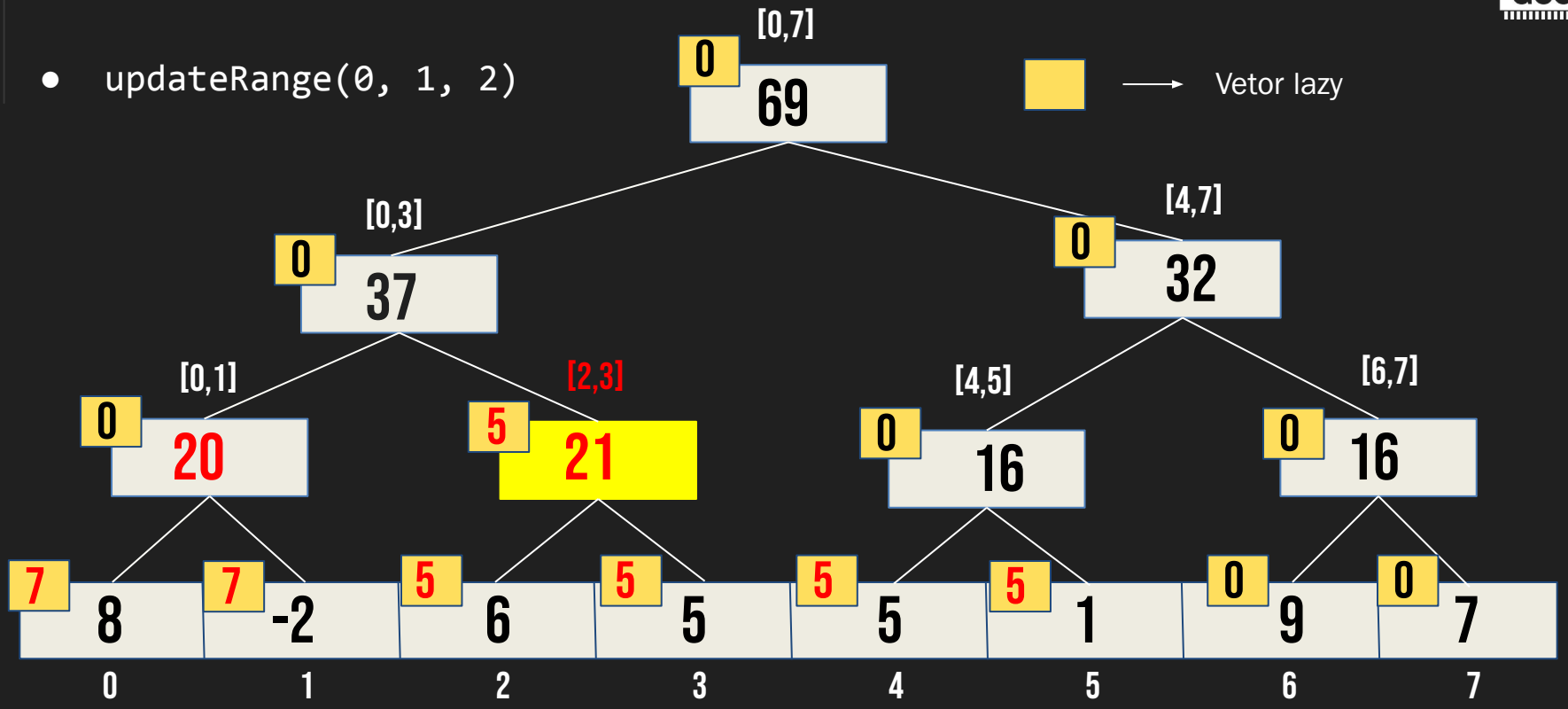
- `updateRange(0, 1, 2)`



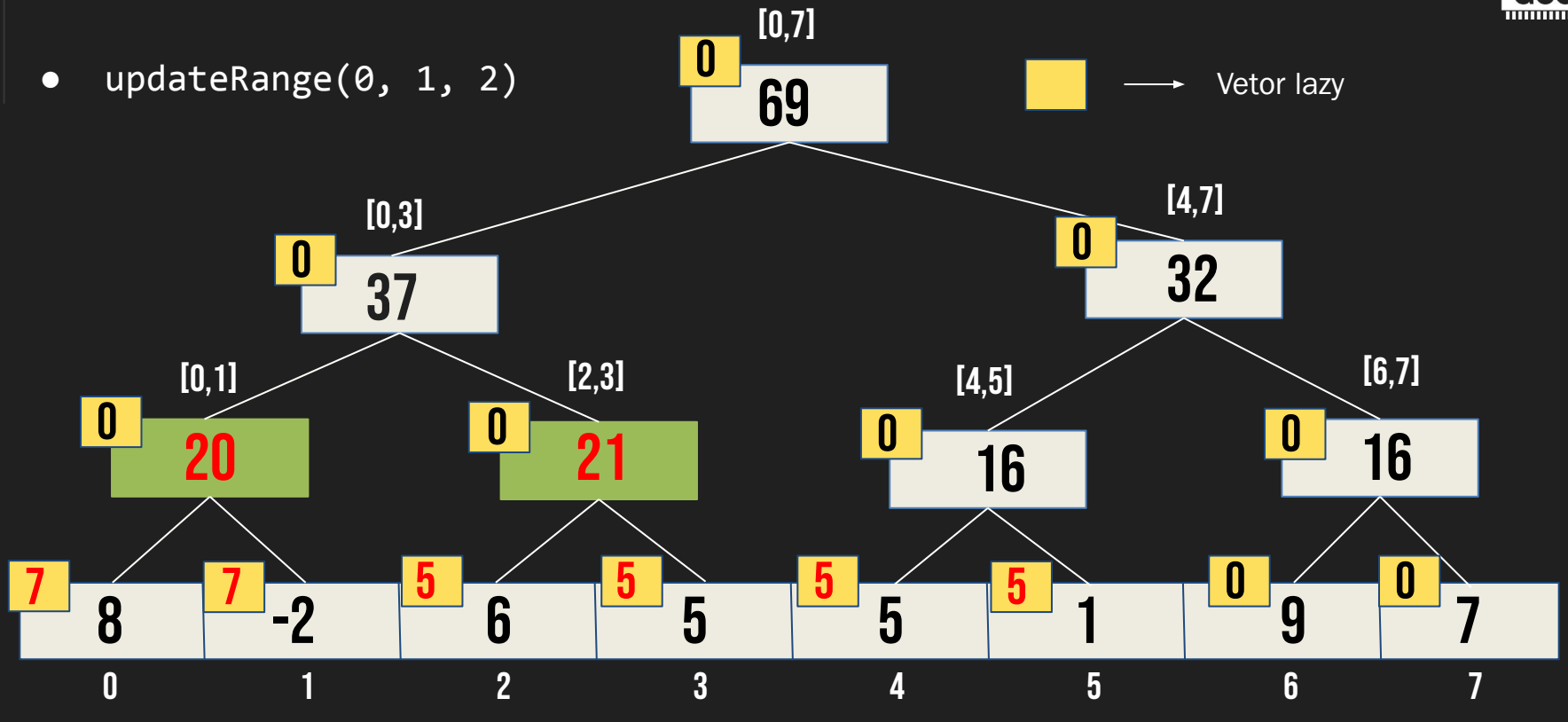
- `updateRange(0, 1, 2)`



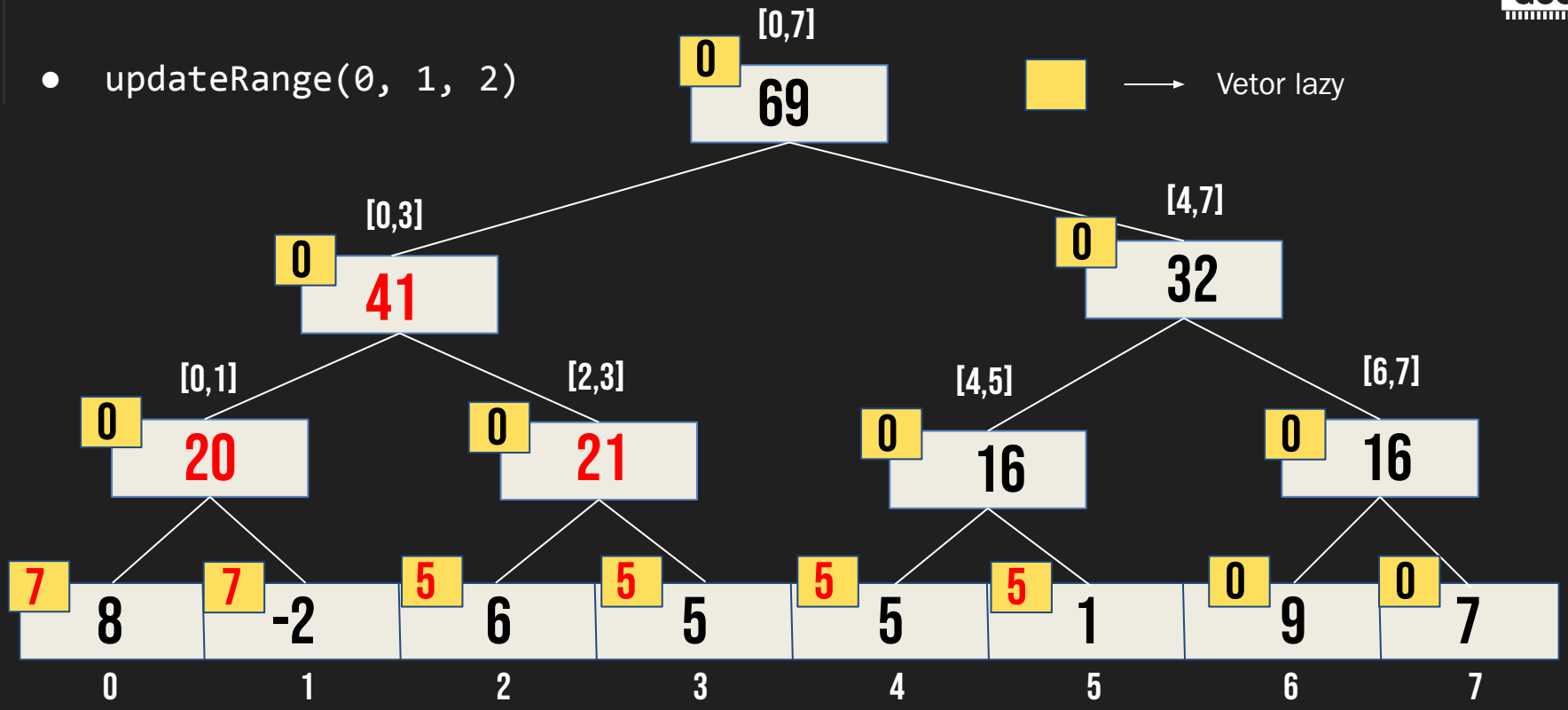
- `updateRange(0, 1, 2)`



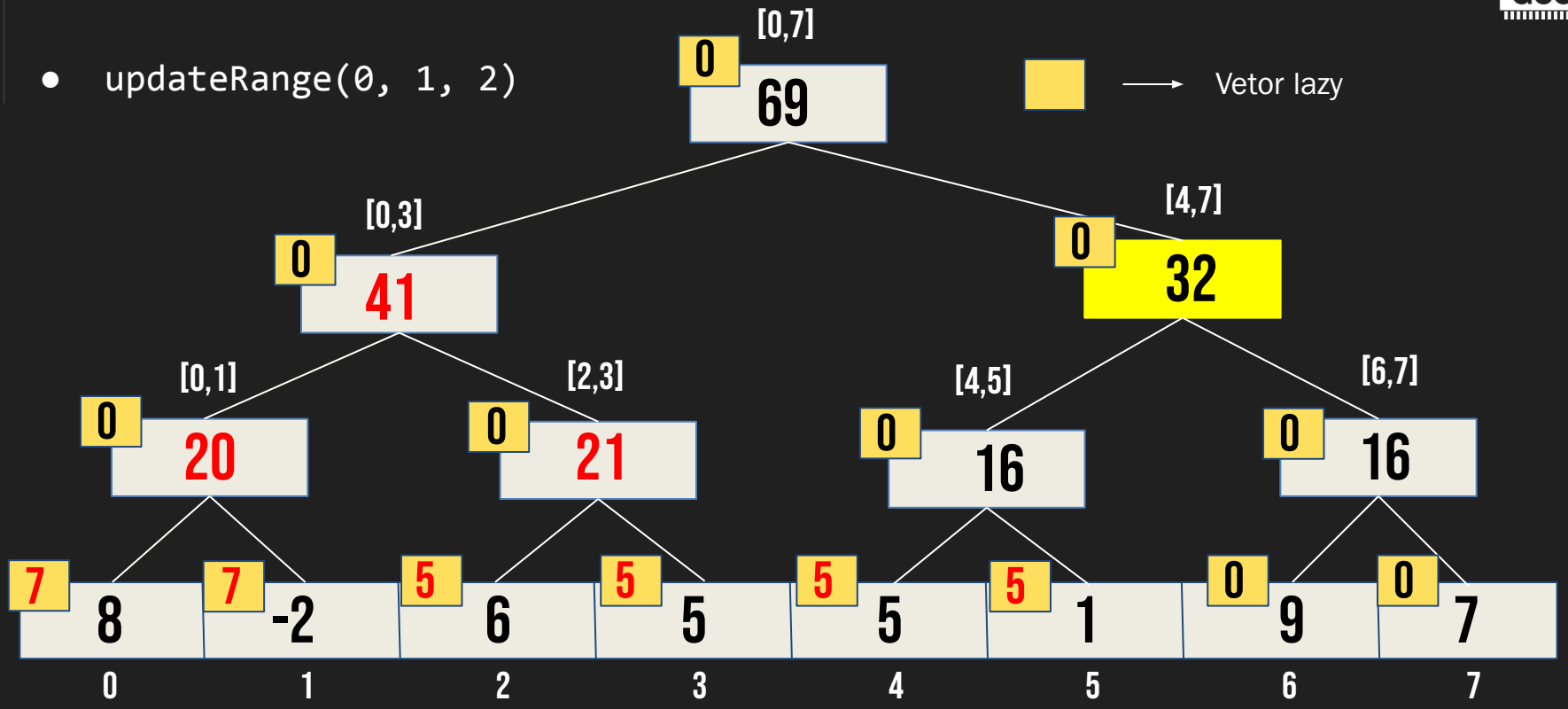
- updateRange(0, 1, 2)



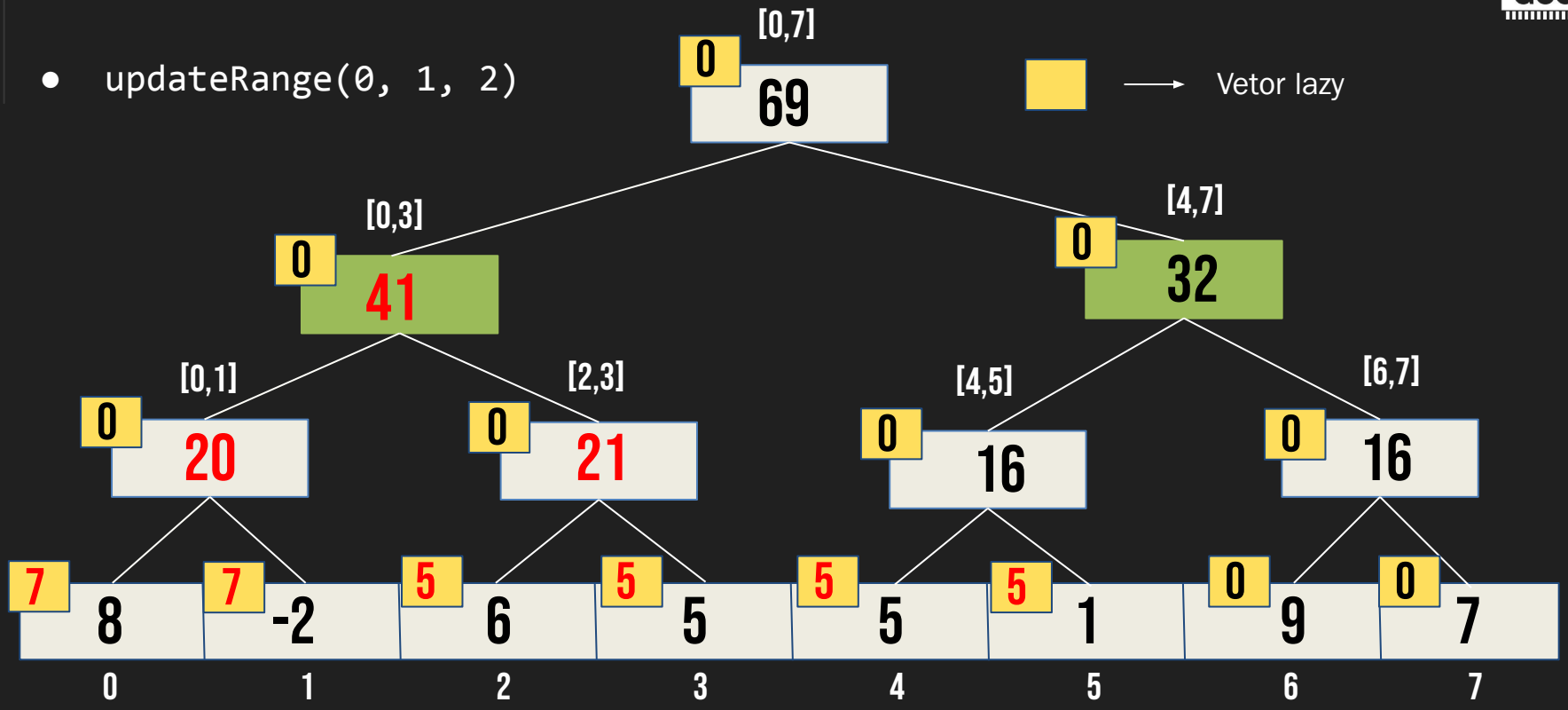
- `updateRange(0, 1, 2)`



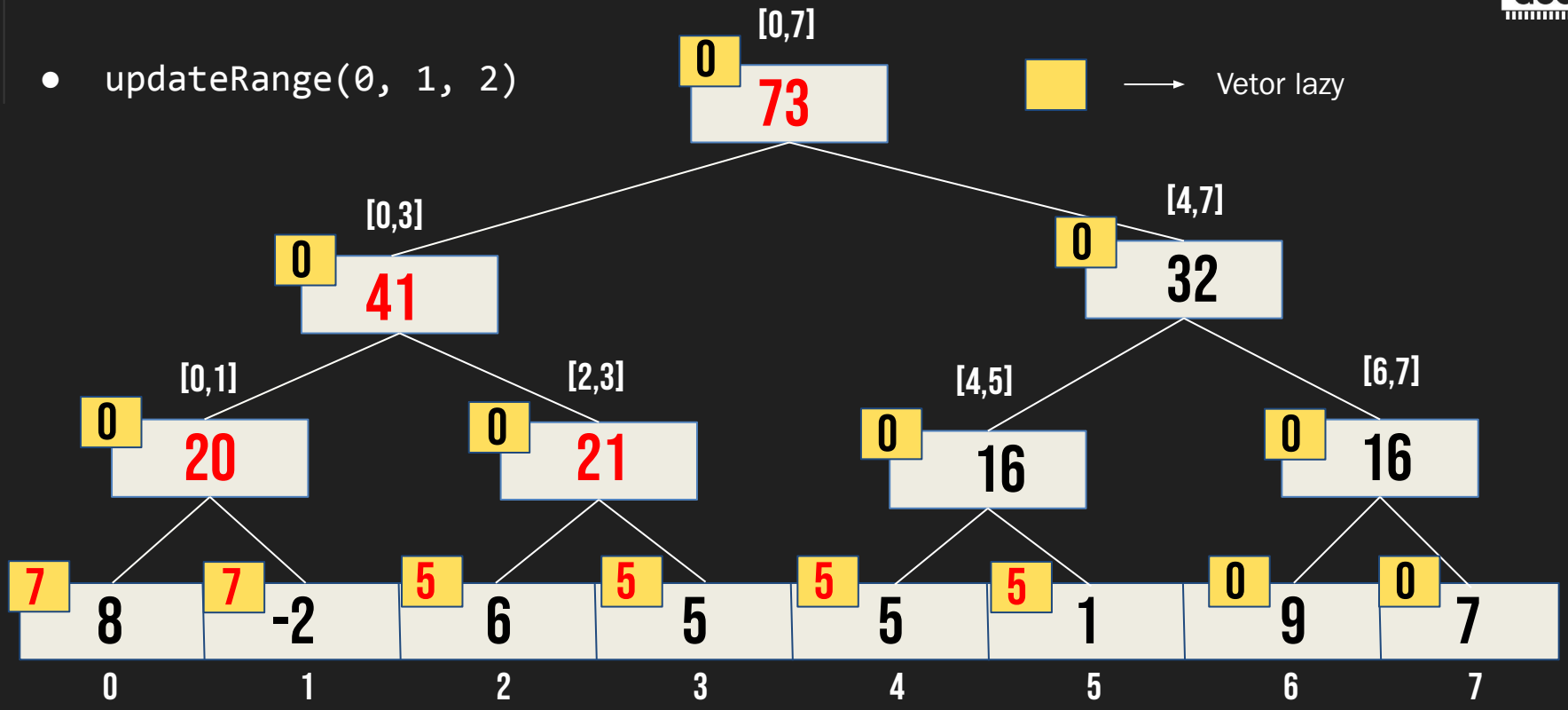
- `updateRange(0, 1, 2)`



- `updateRange(0, 1, 2)`



- `updateRange(0, 1, 2)`



10 - ALGORITMO

10 - ALGORITMO

```
int a[MAXN];
int tree[4*MAXN];
int lazy[4*MAXN]; // Vetor para armazenar atualizações pendentes
```

10 - ALGORITMO

```
void propagate(int node, int l, int r) {



}
```

10 - ALGORITMO

```
void propagate(int node, int l, int r) {
    if (lazy[node] != 0) { // Se houver atualização pendente









    }
}
```

10 - ALGORITMO

```
void propagate(int node, int l, int r) {
    if (lazy[node] != 0) { // Se houver atualização pendente
        tree[node] += (r - l + 1) * lazy[node]; // Aplica a atualização ao nó atual









    }
}
```

10 - ALGORITMO

```
void propagate(int node, int l, int r) {
    if (lazy[node] != 0) { // Se houver atualização pendente
        tree[node] += (r - l + 1) * lazy[node]; // Aplica a atualização ao nó atual

        if (l != r) { // Se não for folha, propaga para os filhos


        }


    }
}
```

10 - ALGORITMO

```
void propagate(int node, int l, int r) {
    if (lazy[node] != 0) { // Se houver atualização pendente
        tree[node] += (r - l + 1) * lazy[node]; // Aplica a atualização ao nó atual

        if (l != r) { // Se não for folha, propaga para os filhos
            lazy[2*node] += lazy[node];
            lazy[2*node+1] += lazy[node];
        }


    }
}
```

10 - ALGORITMO

```
void propagate(int node, int l, int r) {
    if (lazy[node] != 0) { // Se houver atualização pendente
        tree[node] += (r - l + 1) * lazy[node]; // Aplica a atualização ao nó atual

        if (l != r) { // Se não for folha, propaga para os filhos
            lazy[2*node] += lazy[node];
            lazy[2*node+1] += lazy[node];
        }

        lazy[node] = 0; // Limpa a atualização pendente
    }
}
```

10 - ALGORITMO

```
void updateRange(int node, int l, int r, int a, int b, int x) {



}
```

10 - ALGORITMO

```
void updateRange(int node, int l, int r, int a, int b, int x) {
    propagate(node, l, r); // Propaga antes de qualquer operação



}
```

10 - ALGORITMO

```
void updateRange(int node, int l, int r, int a, int b, int x) {
    propagate(node, l, r); // Propaga antes de qualquer operação
    if (b < l || r < a) return; // Fora do intervalo




}
```

10 - ALGORITMO

```
void updateRange(int node, int l, int r, int a, int b, int x) {
    propagate(node, l, r); // Propaga antes de qualquer operação
    if (b < l || r < a) return; // Fora do intervalo
    if (a <= l && r <= b) { // Totalmente dentro do intervalo



    }



}
```

10 - ALGORITMO

```
void updateRange(int node, int l, int r, int a, int b, int x) {
    propagate(node, l, r); // Propaga antes de qualquer operação
    if (b < l || r < a) return; // Fora do intervalo
    if (a <= l && r <= b) { // Totalmente dentro do intervalo
        lazy[node] += x; // Armazena a atualização no nó atual



    }



}
```

10 - ALGORITMO

```
void updateRange(int node, int l, int r, int a, int b, int x) {
    propagate(node, l, r); // Propaga antes de qualquer operação
    if (b < l || r < a) return; // Fora do intervalo
    if (a <= l && r <= b) { // Totalmente dentro do intervalo
        lazy[node] += x; // Armazena a atualização no nó atual
        propagate(node, l, r); // Aplica a atualização
        return;
    }



}
```

10 - ALGORITMO

```
void updateRange(int node, int l, int r, int a, int b, int x) {
    propagate(node, l, r); // Propaga antes de qualquer operação
    if (b < l || r < a) return; // Fora do intervalo
    if (a <= l && r <= b) { // Totalmente dentro do intervalo
        lazy[node] += x; // Armazena a atualização no nó atual
        propagate(node, l, r); // Aplica a atualização
        return;
    }
    int m = (l + r) / 2;
    updateRange(2*node, l, m, a, b, x); // Atualiza o filho esquerdo
    updateRange(2*node+1, m+1, r, a, b, x); // Atualiza o filho direito
    tree[node] = tree[2*node] + tree[2*node+1]; // Atualiza o nó atual
}
```

10 - ALGORITMO

```
void updateRange(int node, int l, int r, int a, int b, int x) {
    propagate(node, l, r);
    if (b < l || r < a) return;
    if (a <= l && r <= b) {
        lazy[node] += x;
        propagate(node, l, r);
        return;
    }
    int m = (l + r) / 2;
    updateRange(2*node, l, m, a, b, x);
    updateRange(2*node+1, m+1, r, a, b, x);
    tree[node] = tree[2*node] + tree[2*node+1];
}
```

10 - ALGORITMO

```
int query(int node, int l, int r, int a, int b) {




}
```

10 - ALGORITMO

```
int query(int node, int l, int r, int a, int b) {
    propagate(node, l, r); // Propaga a atualização pendente antes de consultar



}
```

10 - ALGORITMO

```
int query(int node, int l, int r, int a, int b) {
    propagate(node, l, r); // Propaga a atualização pendente antes de consultar
    if (b < l || r < a) return 0;
    if (a <= l && r <= b) return tree[node];
    int m = (l + r) / 2;
    return query(2*node, l, m, a, b) + query(2*node+1, m+1, r, a, b);
}
```

10 - ALGORITMO

```
int query(int node, int l, int r, int a, int b) {
    propagate(node, l, r);
    if (b < l || r < a) return 0;
    if (a <= l && r <= b) return tree[node];
    int m = (l + r) / 2;
    return query(2*node, l, m, a, b) + query(2*node+1, m+1, r, a, b);
}
```

PRÓXIMOS PASSOS

- Há diversas formas para se fazer o Segment Tree, podemos fazer de soma, mínimo/máximo, produto, MDC/MMC e entre outros.

- Há problemas que precisaremos criar o próprio tipo de dado composto (usando `struct` por exemplo), e definindo uma operação para combinar os resultados durante a construção e consulta.

- Podemos fazer com qualquer tipo de operação associativa:
  - $(A \circ B) \circ C = A \circ (B \circ C)$

- Muitos problemas utilizam SegTree, confira: https://cp-algorithms.com/data_structures/segment_tree.html

REFERÊNCIAS

Aula Segment Tree e Lazy Propagation Unesp:

https://youtu.be/JjCIViUkkw0?si=tQZ8tZSGFncZYRoY

Aula SegTree UFMG:

https://youtu.be/QW_nQN-UQhA?si=XkgCq5EfmB5tMskn

Aula Segment Tree Pavel Mavrin:

https://youtu.be/s3bnguhHttM?si=Bbq6V1gS-z3bipl1

https://noic.com.br/informatica/curso-noic/data-structures-04/

PROBLEMA - 1301 - BEECROWD

Produto do Intervalo:

https://judge.beecrowd.com/pt/problems/view/1301

OBRIGADO PELA ATENÇÃO

Grupo de Computação Competitiva

GCC.CEFETMG

gcc-cefetmg.github.io/Site-GCC

@gcc.cefetmg