maratona de programação

UNIFEI

3 – **Estruturas de Dados**
Parte II

Prof. **João Paulo** R. R. Leite
joaopaulo@**unifei**.edu.br
ECOE44 & ECOE45 – Maratona de Programação
sites.google.com/site/unifeimaratona/

O que mais há pra ser visto hoje?

Estruturas de dados tradicionais, implementadas em bibliotecas prontas das linguagens foram mostradas na aula passada. Mas isso é tudo?

Hoje veremos:

Union-Find Disjoint Sets (UFDS)

E estudaremos as “range queries”:

Segment Tree
Fenwick Tree

Union-Find Disjoint Sets

Imagine que tenhamos uma coleção de N itens.

A estrutura que estamos aprendendo mantém uma coleção de **conjuntos disjuntos**, onde cada um dos N itens está em exatamente um conjunto.

Itens = {1, 2, 3, 4, 5, 6}
Sets = {1,6} ,{2, 3, 5} e {4}

No ato de sua criação, cada item é um **conjunto em si mesmo**, de um único item. A partir desse conjunto inicial, através de operações de união, conseguimos criar os conjuntos necessários para a aplicação.

```cpp
// UFDS (Union-Find Disjoint Sets)
// Cria estrutura e inicializa seu grupo
// parent[i] = i (inicialmente isolados, cada um em seu grupo)
struct union_find {
    vector<int> parent;

    union_find(int n)
    {
        parent = vector<int>(n);
        for(int i = 0; i < n; i++)
            parent[i] = i;
    }

    // find and union operations
};
```

A estrutura contém um vetor parente, que conterá informação sobre o conjunto ao qual pertence o elemento i de nosso universo. Utilizamos STL vector para o trabalho.

Union-Find Disjoint Sets

Há duas operações principais suportadas pela estrutura e que devemos implementar:

find(x), que retorna um elemento significativo para o conjunto de dados do elemento x.

Tomando o exemplo anterior, {1,6} ,{2, 3, 5} e {4}, alguns exemplos de find(x) poderiam ser:

find(6) → 1
find(1) → 1
find(3) → 2
find(4) → 4

A implementação em C++ deixará claro como é escolhido o elemento representativo do conjunto.

```
// find his parent until it is a root
int find(int x) {
    if(parent[x] == x)
        return x;
    else
    {
        parent[x] = find(parent[x]);
        return parent[x];
    }
}
```

Repare que a função é recursiva, e busca entre os ancestrais do elemento, a raiz do conjunto. Essa raiz é o elemento cujo ancestral é ele mesmo.

E de onde vem essa linha de pensamento? Onde trocamos os ancestrais de um elemento? Na operação de união.

Union-Find Disjoint Sets

A segunda operação importante para o union-find é a união entre dois conjuntos disjuntos através da função union.

union(x, y), conecta a árvore de ancestrais de x e y, de maneira que todos os elementos dos conjuntos que contêm x e y passam a formar um único conjunto.

No exemplo:
Sets: {1,6} ,{2, 3, 5} e {4}
union(1, 4);
Sets: {1, 4, 6} e {2, 3, 5}
union(5, 6);
Sets: {1, 2, 3, 4, 5, 6}

Veja a simplicidade do código de união:

```
// connects the roots of x and y
void unite(int x, int y)
{
    parent[find(x)] = find(y);
}
```

Nele, apenas encontramos o representante do conjunto de x através de find(x) e tornamos o representante do conjunto de y seu ancestral.

Verifique como agora, na função find(x), ela continuará até encontrar o ancestral mais antigo de y.

E como fazemos para descobrir se dois elementos estão num mesmo conjunto?

Tempo para pensar.


```cpp
// x and y are in the same set?
bool is_same_set(int x, int y)
{
    return find(x) == find(y);
}
```

Sim! Basta que o elemento representativo de seu conjunto seja o mesmo.

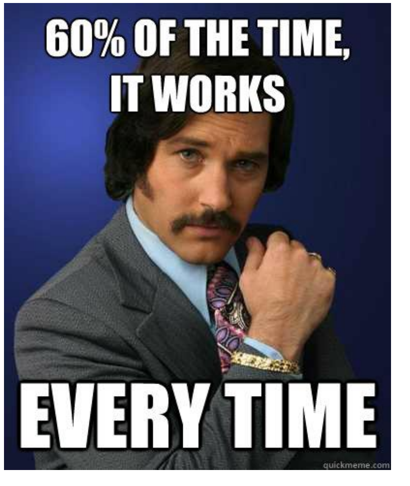
E há outra maneira de implementar o union-find?

Veja essa:

```c
#define MAXN 1000
int p[MAXN];

int find(int x) {
    return p[x] == x ? x : p[x] = find(p[x]); }
void unite(int x, int y) { p[find(x)] = find(y); }

for (int i = 0; i < MAXN; i++) p[i] = i;
```



(Brincadeira.
Sempre funciona)

Union-Find Disjoint Sets

Um dos problemas clássicos que se pode resolver através de UFDS é a verificação de componentes conexas em grafos não-direcionados.

Imagine o caso onde você tem 7 vértices e 5 arestas.
As arestas são:

1-2
5-6
2-3
0-4
0-1

Union-Find Disjoint Sets

Um dos problemas clássicos que se pode resolver através de UFDS é a verificação de componentes conexas em grafos não-direcionados.

Imagine o caso onde você tem 7 vértices e 5 arestas.
As arestas são:

1-2
5-6
2-3
0-4
0-1

union(1,2);

Union-Find Disjoint Sets

Um dos problemas clássicos que se pode resolver através de UFDS é a verificação de componentes conexas em grafos não-direcionados.

Imagine o caso onde você tem 7 vértices e 5 arestas.
As arestas são:

1-2
5-6
2-3
0-4
0-1

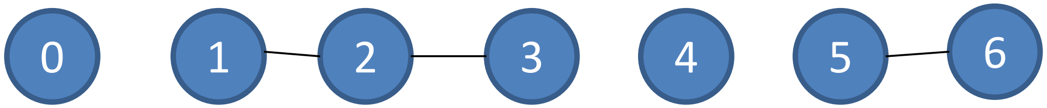
union(5,6);

Union-Find Disjoint Sets

Um dos problemas clássicos que se pode resolver através de UFDS é a verificação de componentes conexas em grafos não-direcionados.

Imagine o caso onde você tem 7 vértices e 5 arestas.
As arestas são:

1-2
5-6
2-3
0-4
0-1



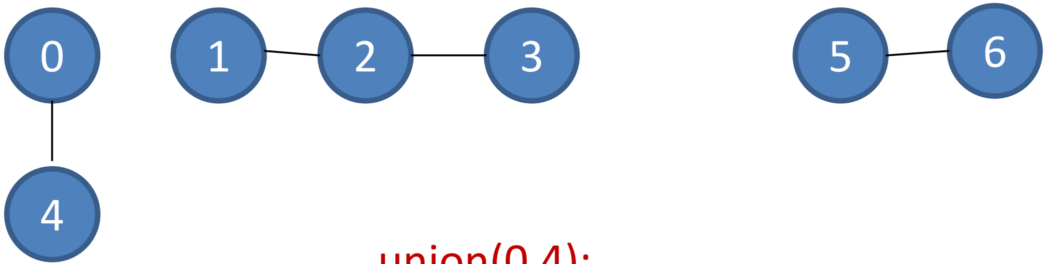
union(2,3);

Union-Find Disjoint Sets

Um dos problemas clássicos que se pode resolver através de UFDS é a verificação de componentes conexas em grafos não-direcionados.

Imagine o caso onde você tem 7 vértices e 5 arestas.
As arestas são:

1-2
5-6
2-3
0-4
0-1



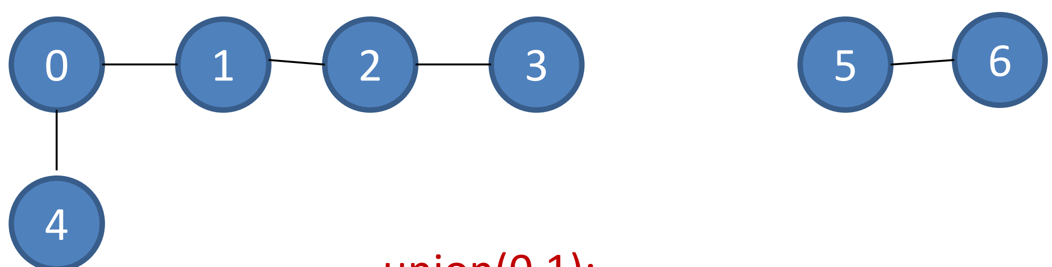
union(0,4);

Union-Find Disjoint Sets

Um dos problemas clássicos que se pode resolver através de UFDS é a verificação de componentes conexas em grafos não-direcionados.

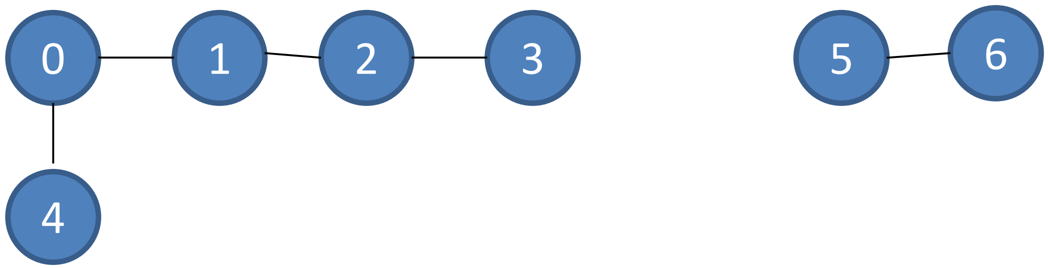
Imagine o caso onde você tem 7 vértices e 5 arestas.
As arestas são:

1-2
5-6
2-3
0-4
0-1



union(0,1);

Ao final da montagem da estrutura, temos um grafo não-direcionado:



Caso queiramos saber:

- O grafo é conexo?
- Quantas componentes conexas possui o grafo?
- Quais elementos compõem cada componente conexa?

Basta que utilizemos o vetor parente e as funções providas pela estrutura
Da union-find.

Exercício 1:

SPOJ 1387: ENERGIA
http://br.spoj.com/problems/ENERGIA/

Range Queries

A tradução mais próxima para “Range Queries” seria algo como “Consultas de intervalo” e você vai entender o porquê.

Imagine que tenhamos um vetor A de tamanho n.
Dados i e j, menores ou iguais a n, nós queremos saber:

- max(A[i], A[i+1], ..., A[j-1], A[j])
- min(A[i], A[i+1], ..., A[j-1], A[j])
- sum(A[i], A[i+1], ..., A[j-1], A[j])

Ou seja, queremos saber o máximo valor dentro de um intervalo, ou o mínimo ou, ainda, a soma de todos os elementos **dentro de um intervalo**.
Como você faria?

Range Queries

Dado o vetor abaixo, queremos algumas somas de intervalos:

0	1	2	3	4	5	6	7
10	7	8	12	3	15	21	1

A primeira opção seria que, a cada consulta – sum(2,7), por exemplo -, percorramos o vetor no intervalo especificado e somemos em um acumulador.

Funciona? Sim, em O(n).
É boa? *Não*. Por que?
Imagine que tenhamos um número *m* muito grande de consultas, teríamos O(m*n), que é fraco.

Range Queries

Dado o vetor abaixo, queremos algumas somas de intervalos:

0	1	2	3	4	5	6	7
10	7	8	12	3	15	21	1

A segunda opção então, é realizar todas as somas de uma única vez, no início do programa, e depois, apenas acessamos a posição inicial e final. Veja como ficaria o vetor de somas:

0	1	2	3	4	5	6	7
10	17	25	37	40	55	76	77

Range Queries

Resolvemos o problema, não?

0	1	2	3	4	5	6	7
10	17	25	37	40	55	76	77

Agora gastamos O(n) apenas uma vez, no início do programa e, a seguir conseguimos realizar qualquer busca em O(1).

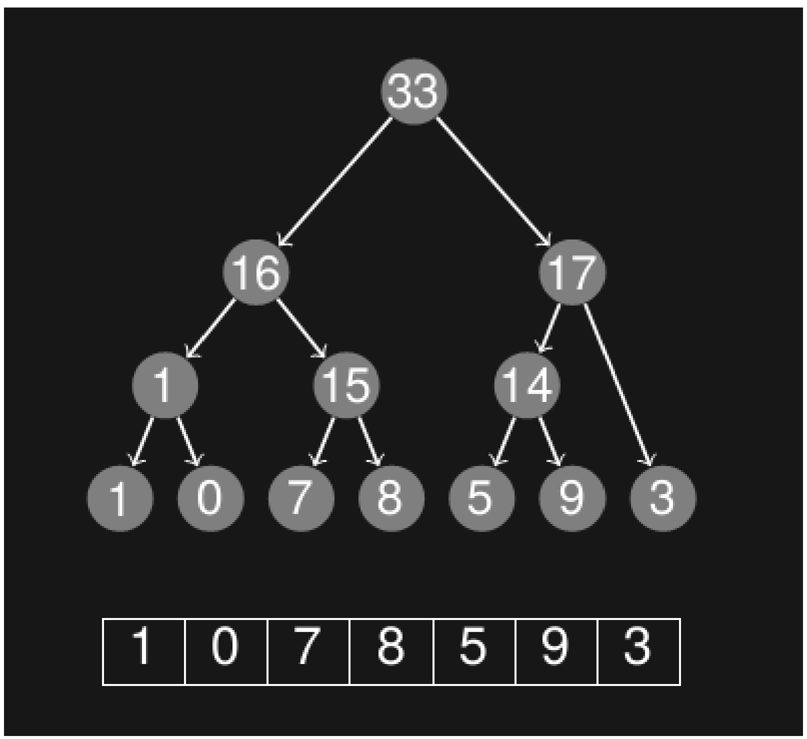
Qual é o problema deste tipo de abordagem?

UPDATES!
Muitas vezes será necessário mudar os valores do vetor durante a execução, o que acarretaria uma **série de muitas atualizações** em O(n).

Range Queries

Mas como dar suporte eficiente tanto para a criação da estrutura, consulta e atualização?

Podemos utilizar uma estrutura chamada "Segment Tree".



Nelas, **cada vértice contém a soma de algum segmento de vetor**.

Construída como árvore binária e, portanto, lembre-se de que sua altura máxima é log(n).

Cada nó da árvore conterá os membros

- **from:** início do segmento
- **to:** fim do segmento
- **value:** valor da soma

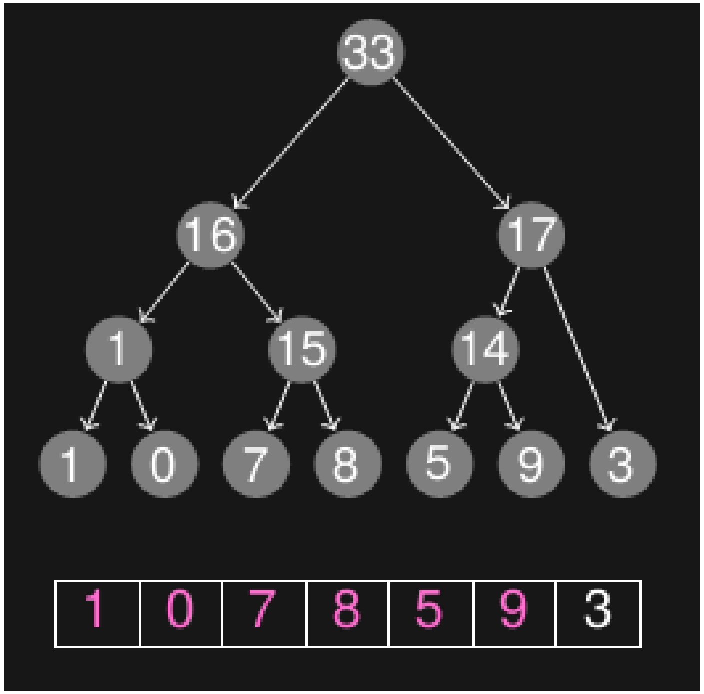
```cpp
// struct base para segment tree
struct segment_tree {
    segment_tree *left, *right;
    int from, to, value;
    segment_tree(int from, int to)
        : from(from), to(to), left(NULL), right(NULL), value(0) { }
};

// construindo a seg tree: estategia recursiva bottom up.
// Complexity: O(n)
segment_tree* build(const vector<int> &arr, int l, int r)
{
    if(l > r) return NULL;

    segment_tree *res = new segment_tree(l, r);
    if(l == r)
        res->value = arr[l];
    else
    {
        int m = (l+r)/2;
        res->left = build(arr, l, m);
        res->right = build(arr, m+1, r);
        if(res->left != NULL)
            res->value += res->left->value;
        if(res->right != NULL)
            res->value += res->right->value;
    }
}
```

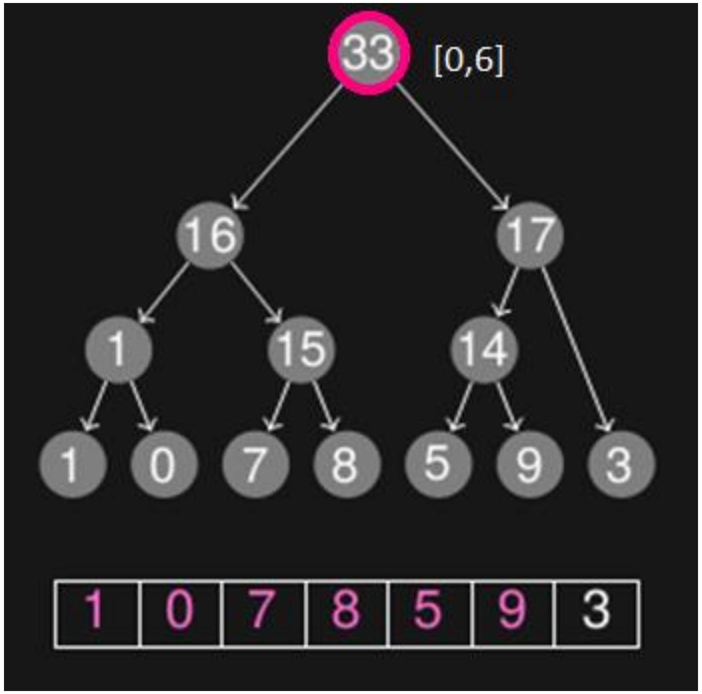
Range Queries

Com a estrutura pronta, podemos realizar queries muito mais rápidas, uma vez que é preciso considerar apenas alguns vértices. Veja o exemplo, de sum(0,5):



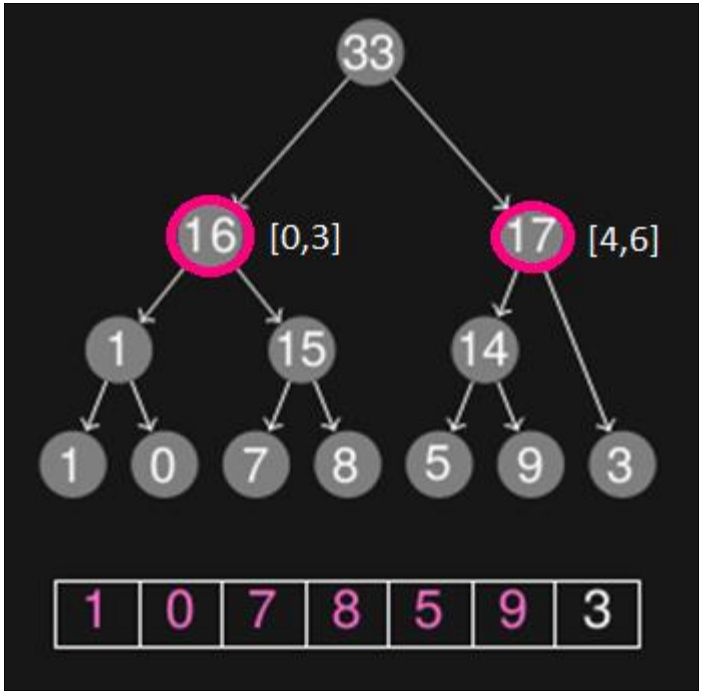
Range Queries

Com a estrutura pronta, podemos realizar queries muito mais rápidas, uma vez que é preciso considerar apenas alguns vértices. Veja o exemplo, de sum(0,5):



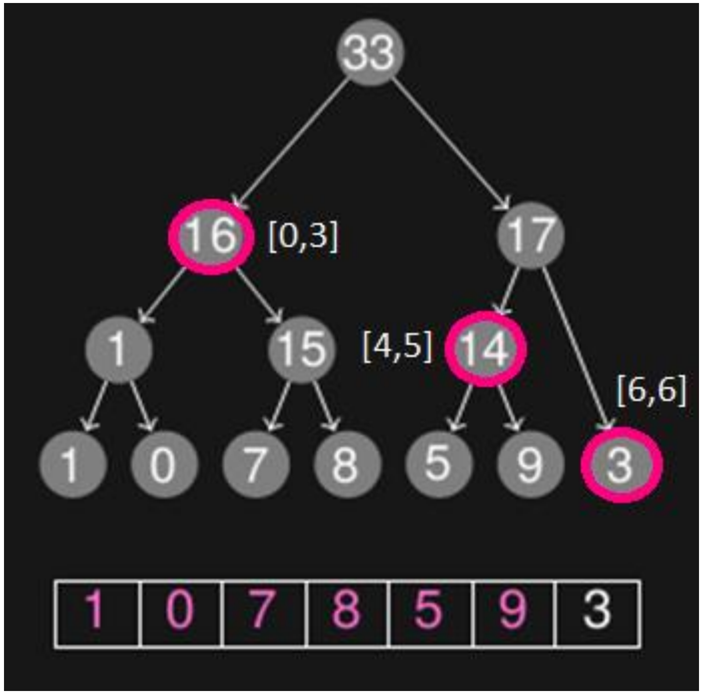
Range Queries

Com a estrutura pronta, podemos realizar queries muito mais rápidas, uma vez que é preciso considerar apenas alguns vértices. Veja o exemplo, de sum(0,5):



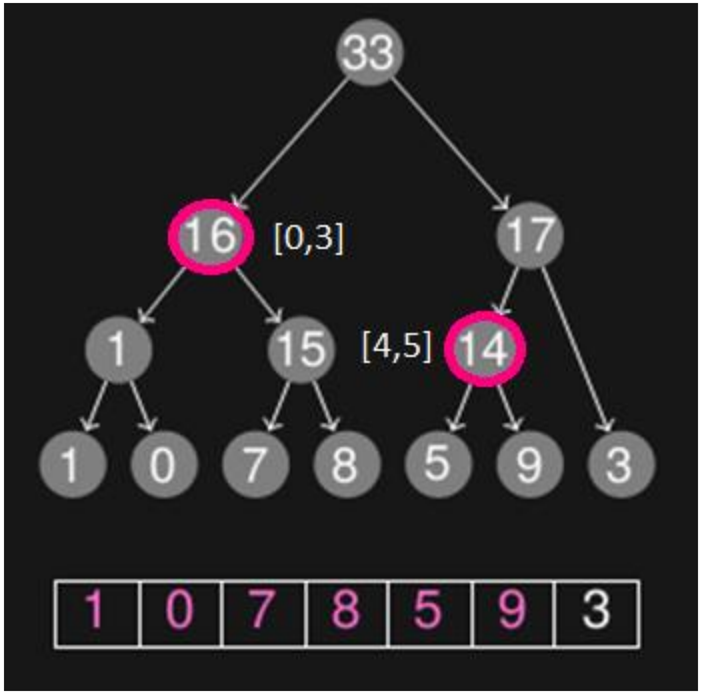
Range Queries

Com a estrutura pronta, podemos realizar queries muito mais rápidas, uma vez que é preciso considerar apenas alguns vértices. Veja o exemplo, de sum(0,5):



Range Queries

Com a estrutura pronta, podemos realizar queries muito mais rápidas, uma vez que é preciso considerar apenas alguns vértices. Veja o exemplo, de sum(0,5) = 16 + 14 = 30:

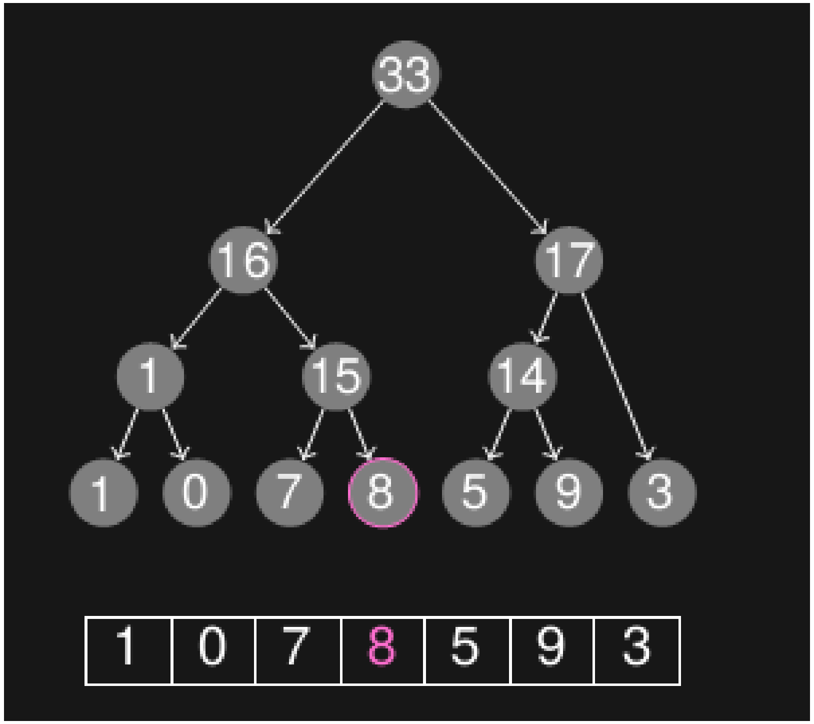


```c
// realizando consultas
// Complexity: O(lg n)
int query(segment_tree *tree, int l, int r)
{
    if(tree == NULL) return 0;
    if(l <= tree->from && tree->to <= r) return tree->value;
    if(tree->to < l) return 0;
    if(tree->from > r) return 0;
    return query(tree->left, l, r) + query(tree->right, l, r);
}
```

Veja que a query somente retorna quando a consulta é feita inteiramente dentro dos limites impostos. Caso haja uma consulta fora dos intervalos ela retorna 0 e, caso ainda não tenhamos descido o suficiente, dividimos a query em duas.

Range Queries

Repare também que, caso seja necessário atualizar algum elemento, a árvore será atualizada em O(log n), o que é bem mais rápido que O(n). Veja um update(3, 5):

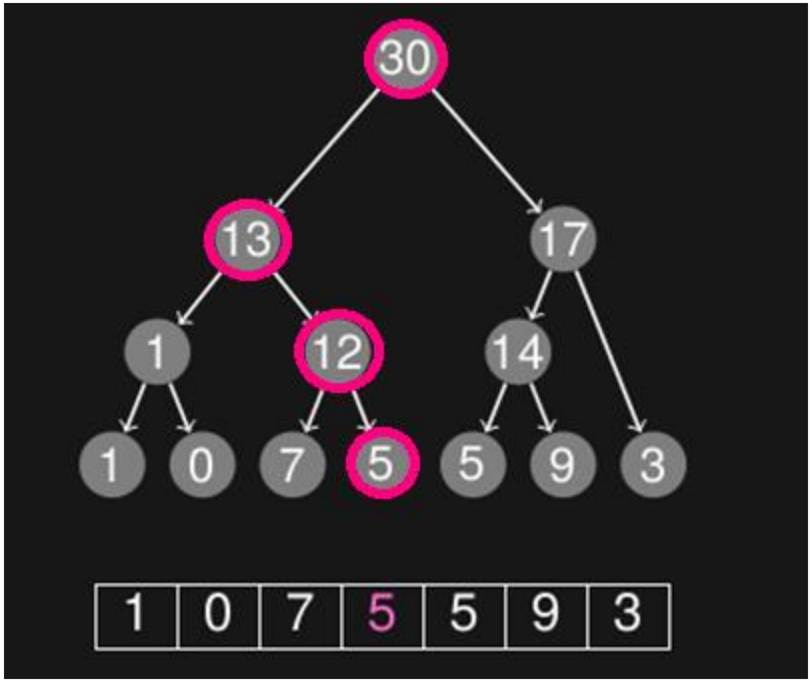


Neste caso, é necessário trocar O terceiro elemento, atualmente igual a 8, pelo valor 5.

No entanto, as somas são afetadas, e precisamos atualizá-las também.

Range Queries

Repare também que, caso seja necessário atualizar algum elemento, a árvore será atualizada em O(log n), o que é bem mais rápido que O(n). Veja um update(3, 5):



Veja que a quantidade de atualizações é igual à altura da árvore, que é binária.

```c
// atualizando valores do vetor
// Complexity: O(lg n)
// caso não haja atualização, é preferível realizar todos os cálculos e
// colocar em um vetor já de começo
int update(segment_tree *tree, int i, int val)
{
    if(tree == NULL) return 0;
    if(tree->to < i)  return tree->value;
    if(tree->from > i) return tree->value;
    if(tree->from == tree->to && tree->from == i)
        tree->value = val;
    else
        tree->value = update(tree->left, i, val) + update(tree->right, i, val);

    return tree->value;
}
```

Veja que, caso o vértice não sofra influência de i, ele não sofre alteração.
Caso ele seja o próprio i, ele recebe o valor atualizado.
Caso contrário, ele é atualizado com a soma da direita e esquerda atualizadas.

Range Queries

Assim, temos que as Segment Trees nos provêem:

- Sua construção em O(n), realizada uma única vez;
- Consultas de intervalo em O(log n), várias vezes;
- Atualização de valor único em O(log n), várias vezes.

O código para outros tipos de consultas como máximo, mínimo, máximo divisor comum, etc. são análogos e podem ser desenvolvidos de acordo com a necessidade.

Exercício 2:

URI Online Judge 1301 - Produto do Intervalo
https://www.urionlinejudge.com.br/judge/pt/problems/view/1301

Range Queries

Fenwick Tree, também conhecida como **BIT**, de *Binary Indexed Tree*, é uma estrutura de dados que mantém uma sequência de elementos, e é capaz de computar, também, somas cumulativas de qualquer intervalo de elementos consecutivos em O(log n).

Além disso, também é possível atualizar qualquer dado a custo de apenas O(log n).

Joia, então a mesma coisa da Segment Tree?

A Fenwick Tree requer **menos espaço** para representar os dados (um vetor) e possui **implementação mais rápida**.

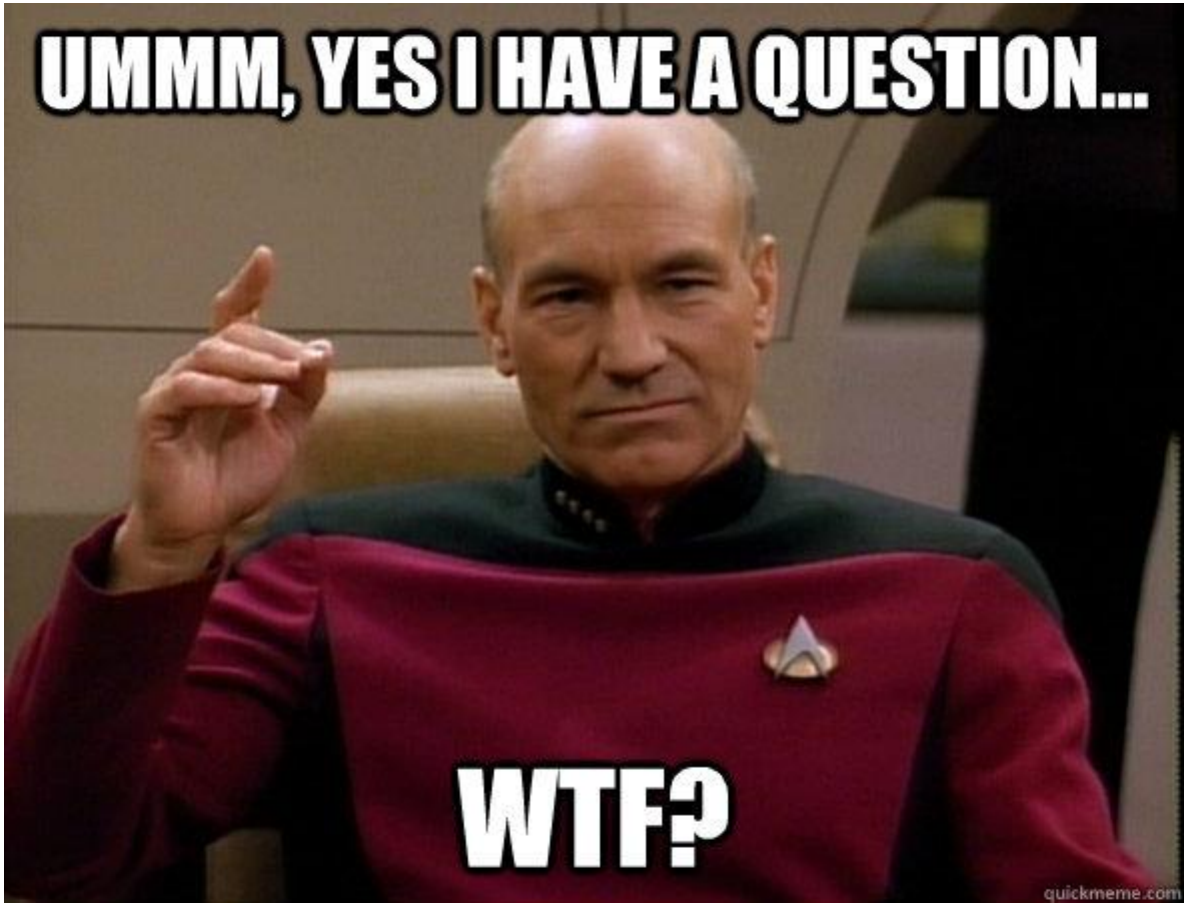
- Desvantagem: Mais difícil de entender.

Range Queries

As Árvores de Fenwick são representadas como vetores simples.

- O tamanho do vetor da árvore é o mesmo tamanho n do vetor de entrada.
- Cada nó da árvore irá armazenar a soma de alguns elementos do array de entrada.

A regra básica da estrutura é que, da mesma maneira que apenas um número pode representar a soma de algumas potências de dois, também conseguimos representar uma soma cumulativa como a soma de algumas somas cumulativas parciais.


UMMM, YES I HAVE A QUESTION...
WTF?
quickmeme.com

Range Queries

Cada índice i no vetor de soma cumulativa é responsável pela soma cumulativa do índice i até o índice (i-(1<<r) + 1), onde r representa a posição do último bit 1 do índice i.

Exemplo:

15 (1111) é responsável pela soma entre **15 e (14+1)** { 1111 – 0001 + 1}

14 (1110) é responsável pela soma entre **14 e (12+1)** { 1110 – 0010 + 1}

12 (1100) é responsável pela soma entre **12 e (8+1)** { 1100 – 0100 + 1}

...

1 (0001) é responsável pela soma entre **1 e (0+1)** { 0001 – 0001 + 1}

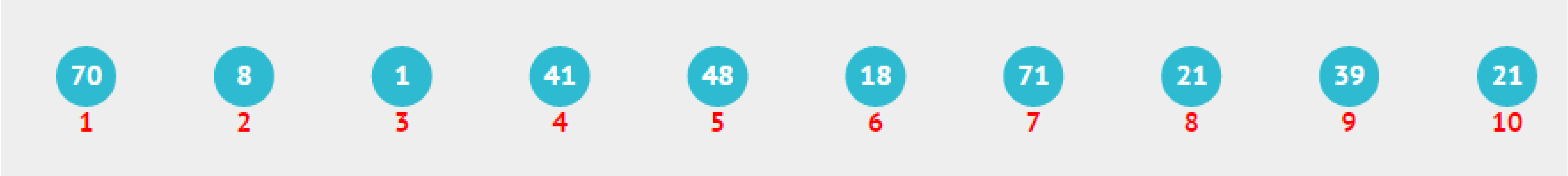
Range Queries

Se tivéssemos 10 itens, teríamos a seguinte configuração:

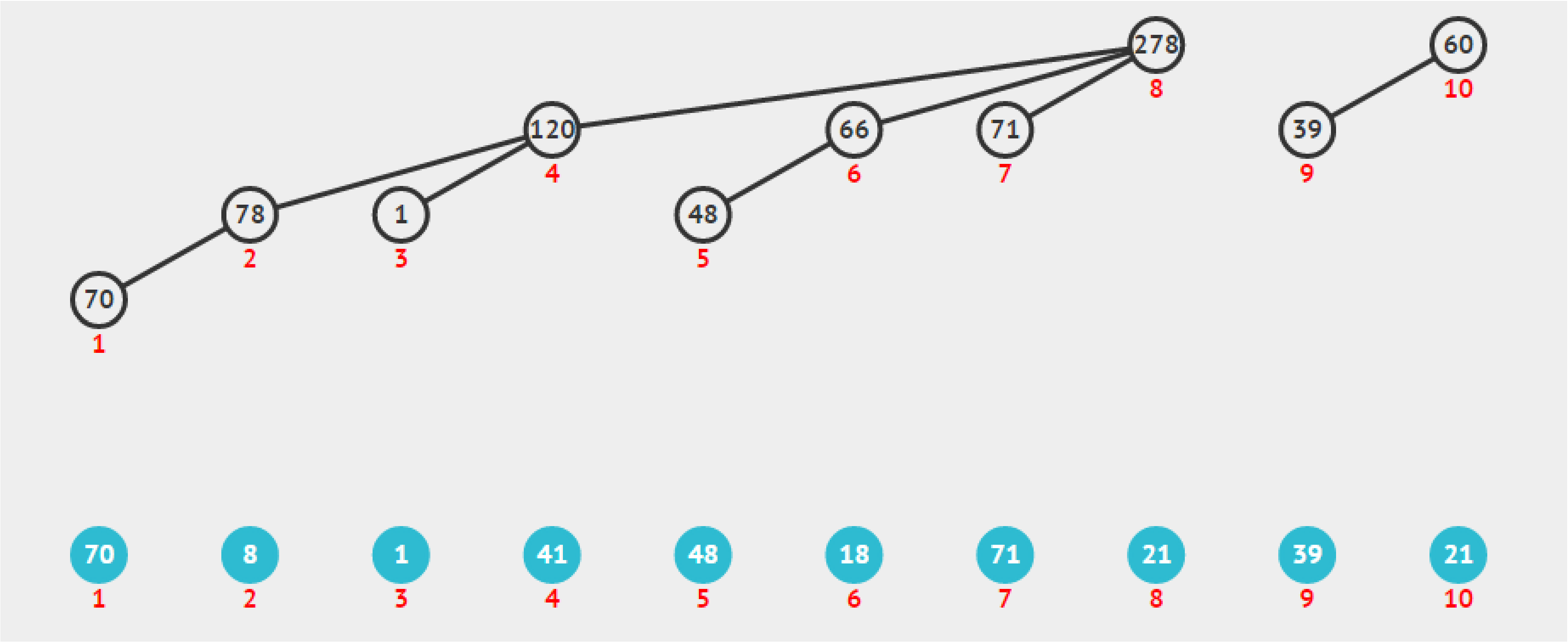
Key/Idx	Binary	Range
0	**0000**	**N/A**
1	0001	[1..1]
2	0010	[1..2]
3	0011	[3..3]
4	0100	[1..4]
5	0101	[5..5]
6	0110	[5..6]
7	0111	[7..7]
8	1000	[1..8]
9	1001	[9..9]
10	1010	[9..10]

Range Queries

A ideia é que qualquer inteiro positivo possa ser representado como potências de dois. Por exemplo, 19 pode ser representado como 16+2+1. Cada nó da BIT armazena a soma de n elementos, onde n é uma potência de 2. Veja por exemplo, como ficaria a árvore para um vetor de 10 itens:



Fonte: http://visualgo.net/fenwicktree.html



Por exemplo, neste diagrama, a soma dos dez primeiros elementos do vetor pode ser obtida simplesmente pela soma dos dois últimos mais a soma dos oito primeiros.

E como conseguimos obter o resultado?

Range Queries

Para obter a soma cumulativa entre 1 e b (*range sum query*, ou simplesmente *rsq*), utilizaremos **rsq**(b):

- A resposta é a soma das somas parciais armazenadas no vetor com índices relacionados a b através da fórmula b’ = b-LSOne(b)
- Aplicamos a fórmula até que b seja 0.
- Exemplo:
  - b = 6 = 0110, b’ = b – LSOne(b) = 0110 – 0010, b1 = 4 = 0100
  - b‘ = 4 = 0100, b’’ = b’ – LSOne(b) = 0100 – 0100, b’’ = 0 → Acaba aqui
- Soma = bit[6] + bit[4] = 120 + 66 = **186**

Range Queries

Para se obter a soma cumulativa entre a e b, utilizaremos **rsq**(a,b)

- Se a for maior que 1, utilizamos rsq(b) – rsq(a-1)
- Exemplo: rsq(4,6)
  - rsq(4,6) = rsq(6) – rsq(4-1) = rsq(6) – rsq(3) = 186 – 79 = **107**

Range Queries

Para atualizar um valor de chave, acrescendo um valor k de v (v pode ser negativo ou positivo), utilizaremos **adjust**(k, v).

- Índices relacionados a k via k' = k + LSOne(k) serão acrescidos de v enquanto k < bit.size()

- Exemplo: adjust(5,1)
  - k = 5 = 0101, k' = k + LSOne(k) = 0101 + 0001, k' = 6 = 0110
  - k' = 6 = 0110, k'' = k' + LSOne(k') = 0110 – 0010, k'' = 8 = 1000
  - k'' = 8 = 1000, k''' = k'' + LSOne(k'') = 1000 + 1000, k''' = 16 = 10000 → Acaba aqui
- Atualizamos então:
  - **bit[5],** de 48 para **49**.
  - **bit[6],** de 66 para **67**.
  - **bit[8],** de 278 para **279**.

Range Queries

Repare que todas as operações são efetuadas no máximo log(n) vezes, que é a quantidade de vezes em que um determinado item aparece em somas parciais no decorrer da árvore.

- Portanto, a complexidade tanto da **soma** quanto da **atualização** são O(log n).
- A **montagem da árvore** tem complexidade n log(n), uma vez que o método adjust é chamado n vezes.

Veja o código:

```cpp
struct fenwick_tree {

    vector<int> ft;

    fenwick_tree(int n) {
        ft.assign(n+1, 0); // init n+1 zeroes
    }

    int rsq(int b)  {
        int sum = 0;
        for(;b; b -= LSOne(b))
            sum += ft[b];
        return sum;
    }

    int rsq(int a, int b)   {
        return rsq(b) - (a == 1 ? 0 : rsq(a-1));
    }

    void adjust(int k, int v) {
        for(; k < (int) ft.size(); k += LSOne(k))
            ft[k] += v;
    }
};
```

Exercício 3:

Uva 12086 - Potentiometers
https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=3238