SBVESDD: Estruturas de Dados

Aula 01: Apresentação da Disciplina, Revisão de Projeto e Análise de Algoritmos, Revisão de Programação Orientada a Objetos e Estudos de Caso

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Apresentação da Disciplina

- Estruturas de Dados;
- 4 aulas semanais, durante 19 semanas, totalizando 76 aulas semestrais.

Logística
Critérios de Avaliação

Onde:

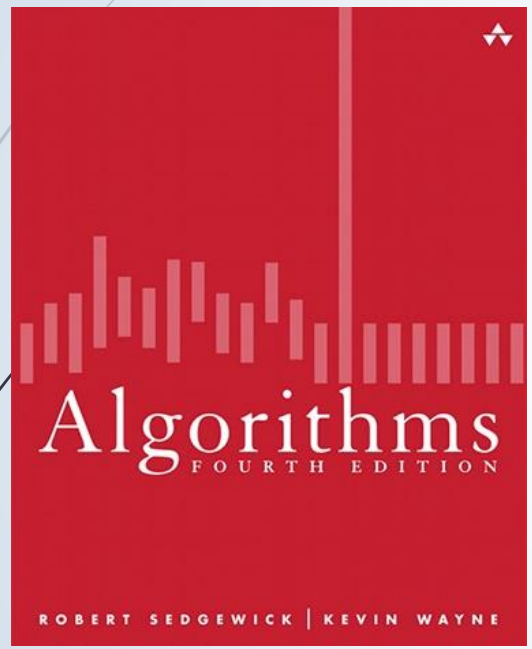
- M: média final;
- q: quantidade de agrupamentos temáticos;
  - A_i: nota da avaliação diagnóstica de um agrupamento temático i, sendo que $A_i = \{x \mid 0 \leq x \leq 10 \wedge x \in \mathbb{Q}\}$;
  - e_i: quantidade de listas de exercícios de um agrupamento temático i;
    - E_{j_i}: nota da lista de exercícios j de um agrupamento temático i, sendo que $E_{j_i} = \{x \mid 0 \leq x \leq 10 \wedge x \in \mathbb{Q}\}$;
- p: quantidade de projetos;
  - P_i: nota do projeto i, sendo que $P_i = \{x \mid 0 \leq x \leq 10 \wedge x \in \mathbb{Q}\}$;
- D: desafio opcional, onde somente o primeiro a entregar e a acertar ganha meio ponto. Se não acertar, o segundo a entregar é avaliado e assim por diante. Um aluno só pode ganhar uma vez por semestre.

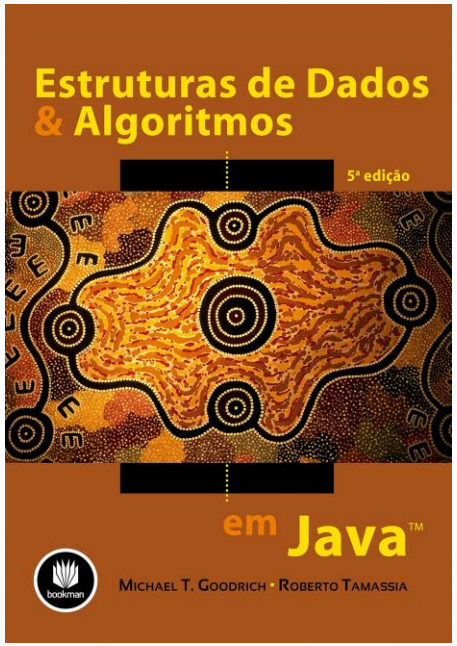
$$M = \begin{cases} \dfrac{\sum_{i=1}^{q} \left(\dfrac{A_i}{2} + \dfrac{A_i}{2} * \dfrac{\sum_{j_i=1}^{e_i} E_{j_i}}{e_i} \right)}{q} + D, & \text{se } p = 0 \\ \dfrac{\sum_{i=1}^{q} \left(\dfrac{A_i}{2} + \dfrac{A_i}{2} * \dfrac{\sum_{j_i=1}^{e_i} E_{j_i}}{e_i} \right)}{q} + \dfrac{\sum_{i=1}^{p} P_i}{p} + D, & \text{se } p > 0 \end{cases}$$

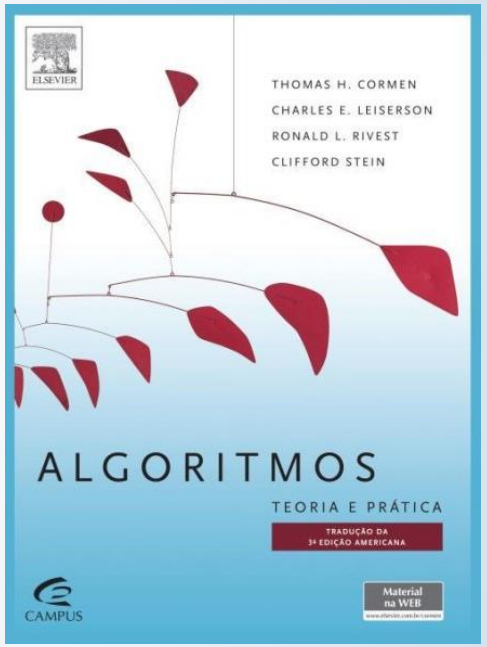
$$, q \in \mathbb{N}^* \wedge p \in \mathbb{N}$$

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Bibliografia







Estruturas de dados:

representações de dados e suas operações associadas.

Apresentação da Disciplina

- Conteúdo programático:
  - Estruturas de Dados Lineares:
    - Arrays, Pilhas, Filas, Deques e Listas;
  - Estruturas de Dados Não-Lineares:
    - Tabela de Símbolos:
      - Lista encadeada com busca sequencial;
      - Array ordenado com busca binária;
      - Árvores:
        - Binárias;
        - Binárias de Busca;
        - Binárias de Busca Balanceadas;
      - Tabelas de Dispersão;
    - Filas de Prioridades:
      - Heaps Binários mínimos e máximos, Ternários e d-ários;
    - Grafos, Digrafos, Grafos ponderados, suas implementações, propriedades e algoritmos.

Revisão de Análise e Projeto de Algoritmos

- Notação Assintótica e Crescimento de Funções;
- Recursão;
- Algoritmos de Busca;
- Algoritmos de Ordenação Comparativos;
- Grafos;
- Algoritmos Gulosos;
- Programação Dinâmica.

Revisão de POO em Java

- Classes;
- Objetos;
- Composição;
- Encapsulamento;
- Herança;
- Polimorfismo:
  - *Ad hoc*:
    - Sobrecarga;
    - Coerção (*cast*);
  - Universal (*late binding*/vinculação tardia):
    - Herança;
    - Paramétrico.
- Exemplo do programa de desenho.

Estudo de Caso
3-SUM (ThreeSum)

- **2-SUM:** Dado um conjunto de números inteiros, buscar todos os pares que, quando somados, resultam em zero.

```java
/**
 * Pesquisa em um array todos os pares que, ao somados, resultam em zero.
 * Retorna a quantidade de pares encontrados.
 */
public static int twoSum( int[] array ) {

    int c = 0;

    for ( int i = 0; i < array.length; i++ ) {
        for ( int j = i + 1; j < array.length; j++ ) {
            if ( array[i] + array[j] == 0 ) {
                System.out.printf( "%d + %d = 0\n", array[i], array[j] );
                c++;
            }
        }
    }

    return c;

}
```

- Qual a ordem de crescimento desse algoritmo? Tem como melhorar?

Estudo de Caso
3-SUM (ThreeSum)

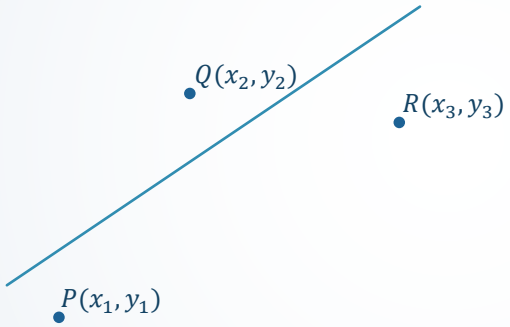
- **3-SUM:** Dado um conjunto de números inteiros, buscar todos os trios que, quando somadas, resultam em zero.

```
/**
 * Pesquisa em um array todos os trios que, ao somados, resultam em zero.
 * Retorna a quantidade de trios encontrados.
 */
public static int threeSum( int[] array ) {

    int c = 0;

    for ( int i = 0; i < array.length; i++ ) {
        for ( int j = i + 1; j < array.length; j++ ) {
            for ( int k = j + 1; k < array.length; k++ ) {
                if ( array[i] + array[j] + array[k] == 0 ) {
                    System.out.printf( "%d + %d + %d = 0\n", array[i], array[j], array[k] );
                    c++;
                }
            }
        }
    }

    return c;

}
```

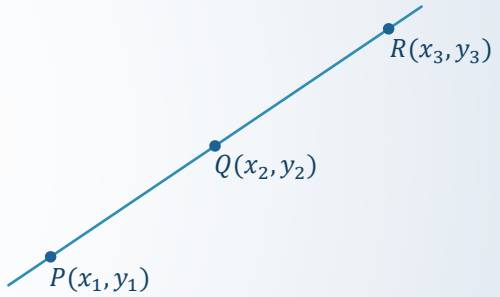
- Qual a ordem de crescimento desse algoritmo? Tem como melhorar?

Estudo de Caso
3-SUM (ThreeSum)

- **Aplicação 3-SUM:**
  - 3-colinearidade:
    - Dados três pontos: $P(x_1, y_1)$, $Q(x_2, y_2)$ e $R(x_3, y_3)$, determinar se os três pontos são colineares;



3 pontos não-colineares



3 pontos colineares

- Três pontos P, Q e R são colineares quando a inclinação entre P e Q é igual à inclinação entre P e R, ou seja:

$$\frac{y_1 - y_2}{x_1 - x_2} = \frac{y_1 - y_3}{x_1 - x_3}$$

$$\Rightarrow (y_1 - y_2)(x_1 - x_3) = (y_1 - y_3)(x_1 - x_2)$$

$$\Rightarrow x_1(y_2 - y_3) + x_2(y_2 - y_1) + x_3(y_1 - y_2) = 0$$

Estudo de Caso
3-SUM (ThreeSum)

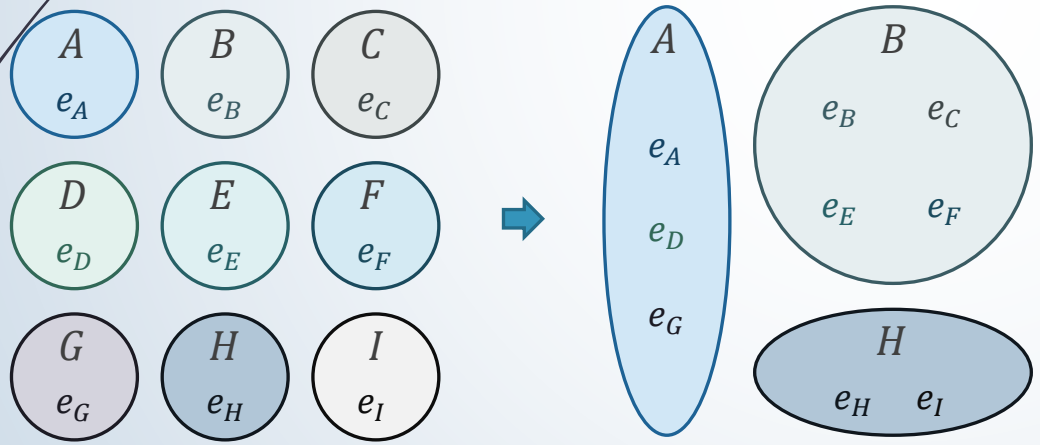
- **3-SUM:** Como melhorar ou tornar viável? Ordenar o array, processar os pares e, usando busca binária, procurar pelo inverso da soma no intervalo posterior ao segundo valor:

```
/**
 * Pesquisa em um array todos os trios que, ao somados, resultam em zero.
 * Usa busca binária para acelerar o processo.
 */
public static int threeSumFast( int[] array ) {

    int c = 0;

    for ( int i = 0; i < array.length; i++ ) {
        for ( int j = i + 1; j < array.length; j++ ) {
            int k = Arrays.binarySearch( array, -( array[i] + array[j] ) );
            if ( k > j ) {
                System.out.printf( "%d + %d + %d = 0\n", array[i], array[j], array[k] );
                c++;
            }
        }
    }

    return c;

}
```

- Qual a ordem de crescimento desse algoritmo? Tem como melhorar?

Estudo de Caso
Union-Find

- A estrutura de dados Union-Find, também conhecida como disjoint-set ou merge-find set, é usada para resolver o problema da conectividade dinâmica, ou seja, dada uma coleção de conjuntos disjuntos, é possível unir tais conjuntos e verificar se quaisquer dois elementos contidos nos conjuntos gerados fazem ou não parte de um mesmo conjunto. A ideia é processar quantidades gigantescas de conjuntos em tempo ótimo, ou seja, tanto o processo de união (*union*) quanto de busca (*find*) devem operar o mais rápido possível.



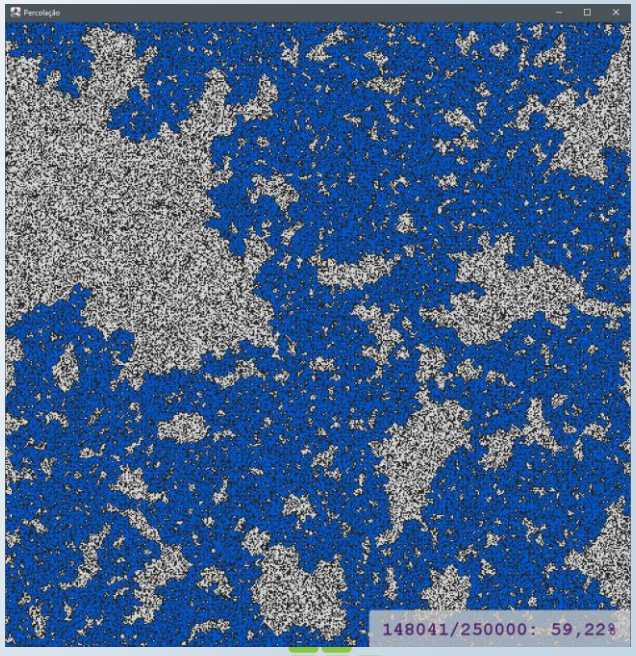
9 conjuntos, cada um com um elemento

3 conjuntos após 6 operações de união

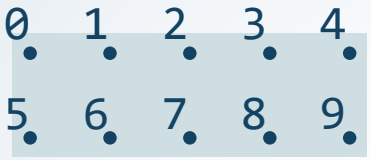
Simulação: verificar se há percolação em um material.

Processamento de 250 mil conjuntos (pontos pretos, brancos e azuis), realizando aproximadamente 150 mil operações de união e 230 mil operações de busca, executadas em menos de um segundo.

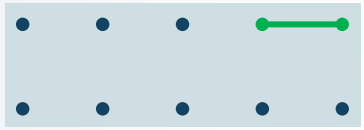
A área em azul é um conjunto representando a percolação, os pontos pretos são conjuntos unitários representando pontos maciços do material e as áreas em branco são clusters ou aglomerados de conjuntos unidos que representam a porosidade do material



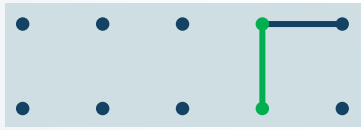
Estudo de Caso
Union-Find



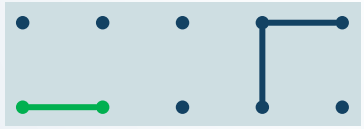
unir 4 e 3



unir 3 e 8

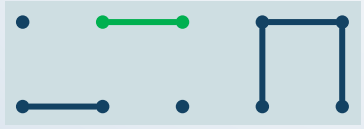


unir 6 e 5



unir 9 e 4

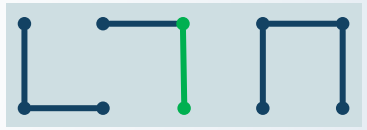
unir 2 e 1



unir 8 e 9

unir 5 e 0

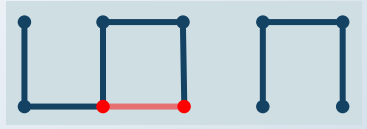
unir 7 e 2



unir 6 e 1

unir 1 e 0

unir 6 e 7

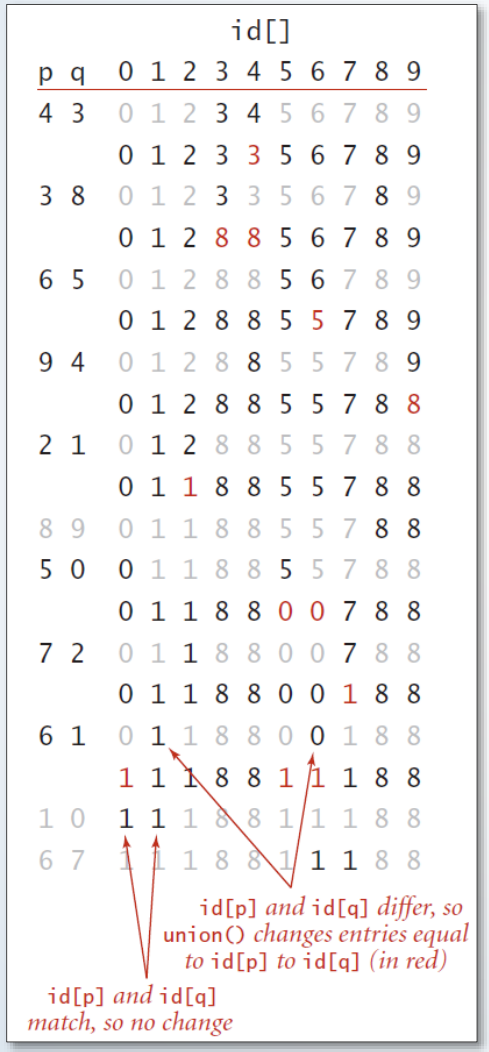


Estudo de Caso
Union-Find - Implementação

```
public abstract class UF {

    protected int count;  // quantidade de componentes/conjuntos

    /**
     * Retorna o componente que p faz parte.
     */
    public abstract int find( int p ) throws IllegalArgumentException;

    /**
     * Une o conjunto que contém p com o conjunto que contém q.
     */
    public abstract void union( int p, int q ) throws IllegalArgumentException;

    /**
     * Verifica se dois elementos estão no mesmo componente.
     */
    public boolean connected( int p, int q ) throws IllegalArgumentException {
        return find( p ) == find( q );
    }

}
```

Estudo de Caso

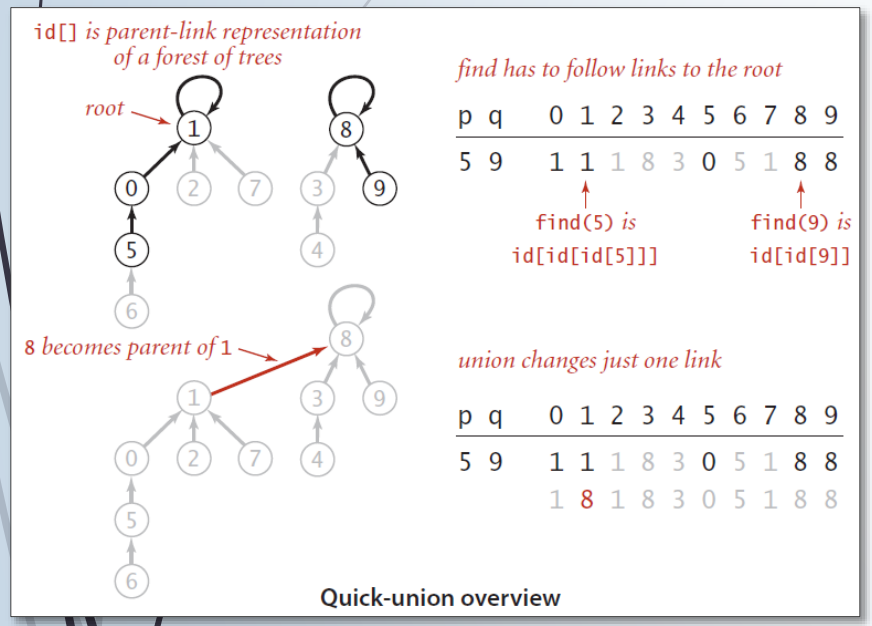
Union-Find - Implementação



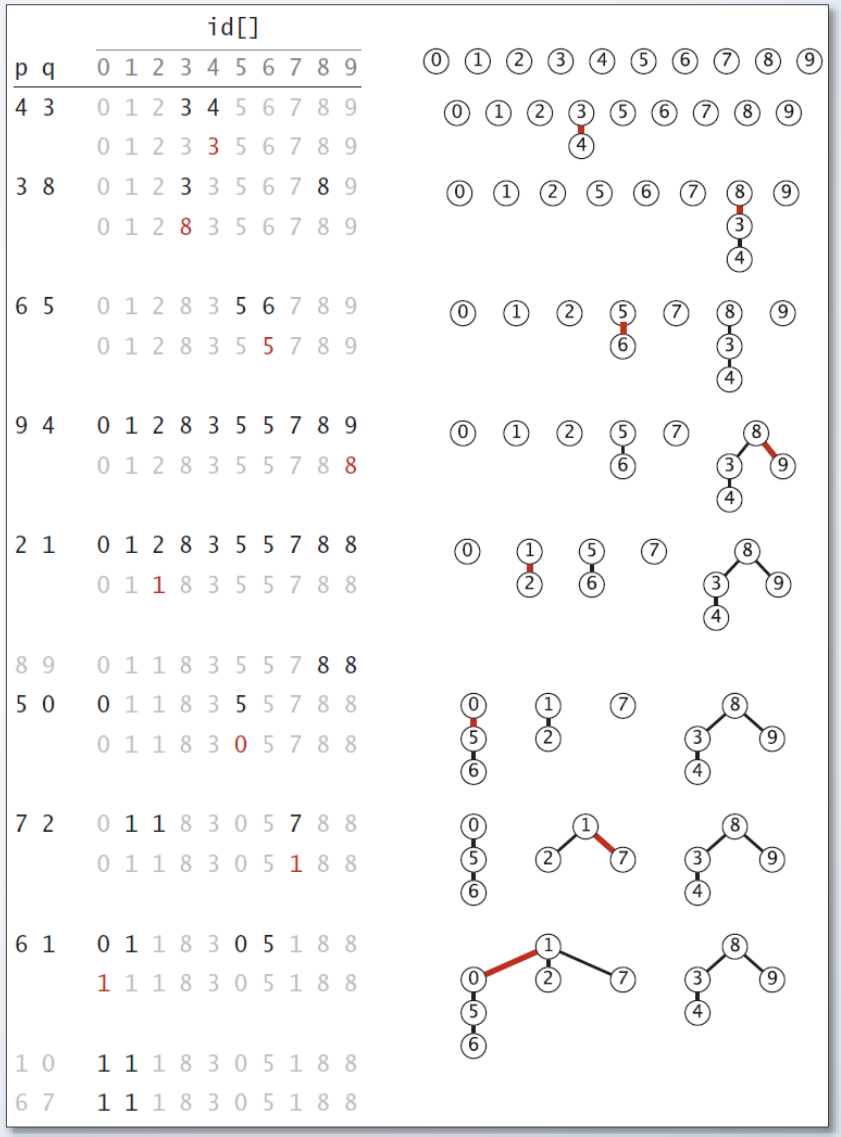

```java
public class QuickFindUF {

    // id[i] = identificador do componente i
    private int[] id;

    public int find( int p ) {
        return id[p];
    }

    public void union( int p, int q ) {

        // em qual componente?
        int pID = find( p );
        int qID = find( q );

        // mesmo componente? não faz nada
        if ( pID == qID ) {
            return;
        }

        // componentes distintos
        // insere componente p no componente q
        for ( int i = 0; i < id.length; i++ ) {
            if ( id[i] == pID ) {
                id[i] = qID;
            }
        }

        count--;

    }

}
```

Estudo de Caso

Union-Find - Implementação

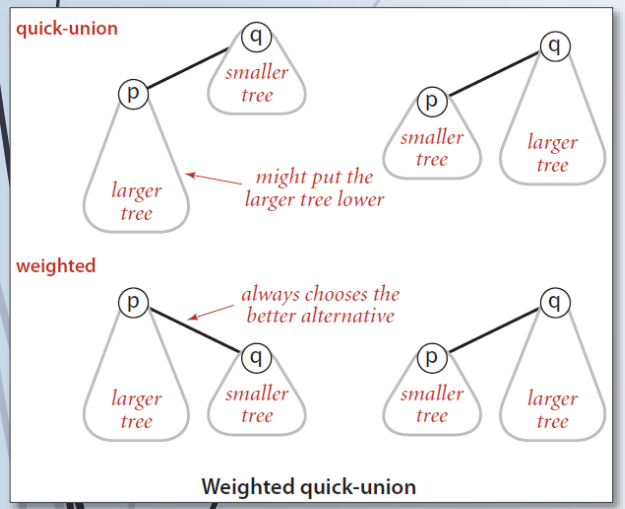


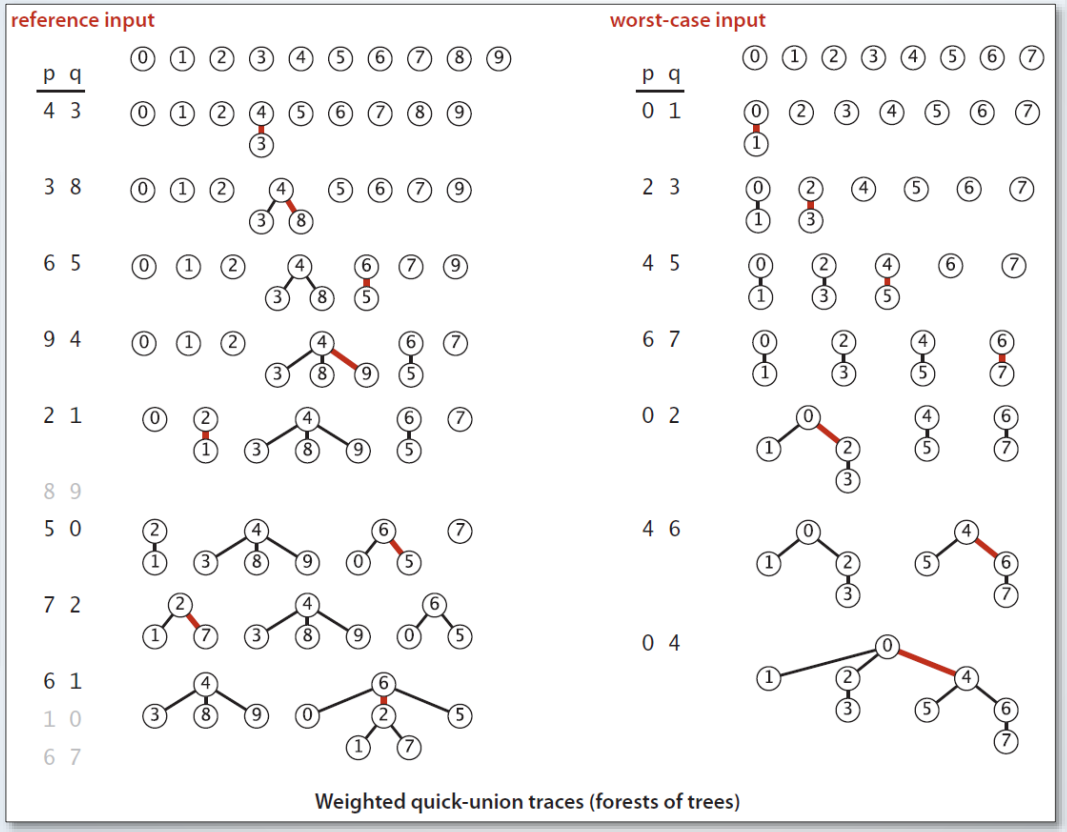
Quick-union overview




```java
public class QuickUnionUF {

    // parent[i] = pai de i
    private int[] parent;

    public int find( int p ) {

        // caminha até a raiz
        while ( p != parent[p] ) {
            p = parent[p];
        }

        return p;

    }

    public void union( int p, int q ) {

        // mesma raiz para p e q
        int rootP = find( p );
        int rootQ = find( q );

        if ( rootP == rootQ ) {
            return;
        }

        parent[rootP] = rootQ;
        count--;

    }

}
```

Estudo de Caso
Union-Find - Implementação

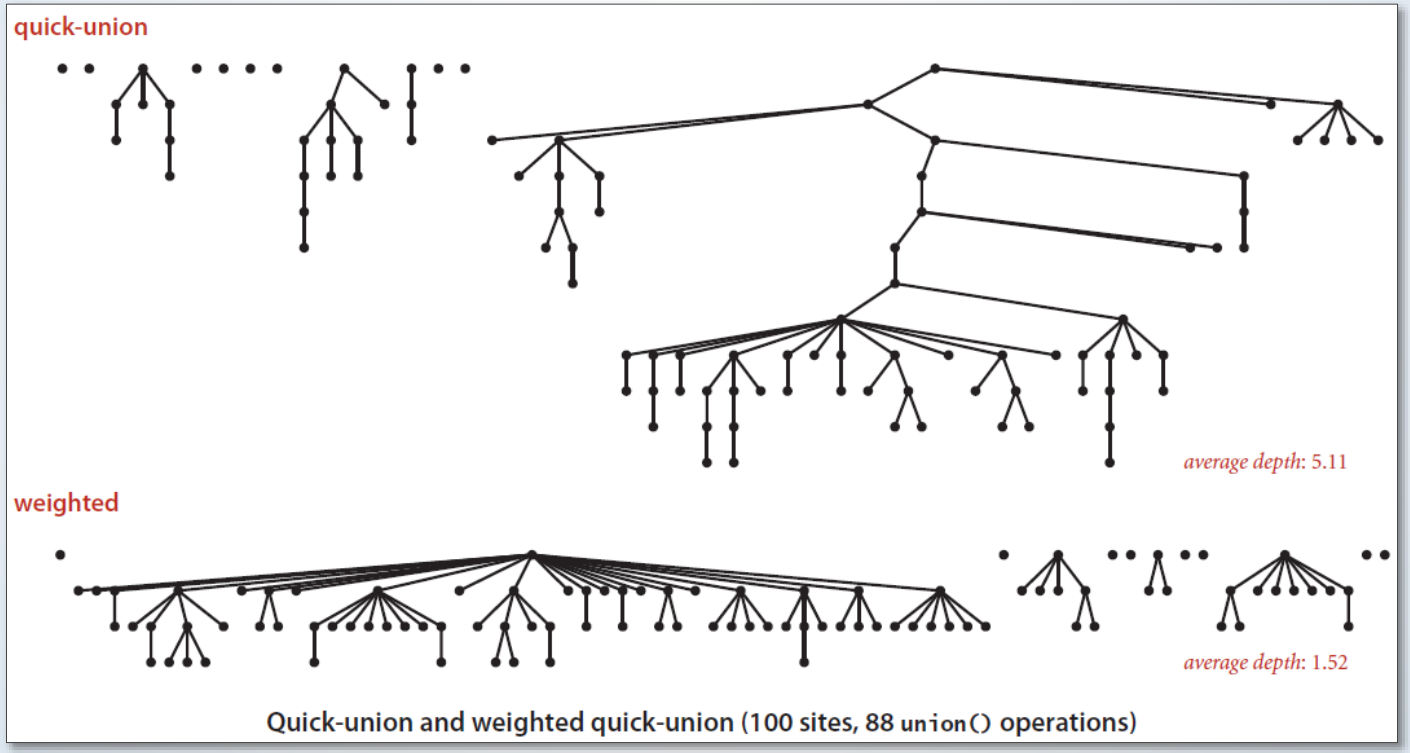





```java
public class WeightedQuickUnionUF {

    private int[] parent;
    // size[i] = quantidade de elementos na
    // subárvore enraizada em i
    private int[] size;

    public int find( int p ) {

        while ( p != parent[p] ) {
            p = parent[p];
        }

        return p;

    }

    public void union( int p, int q ) {

        int rootP = find( p );
        int rootQ = find( q );

        if ( rootP == rootQ ) {
            return;
        }

        // faz a menor raiz apontar para a maior
        if ( size[rootP] < size[rootQ] ) {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        } else {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        }

        count--;

    }

}
```

Estudo de Caso

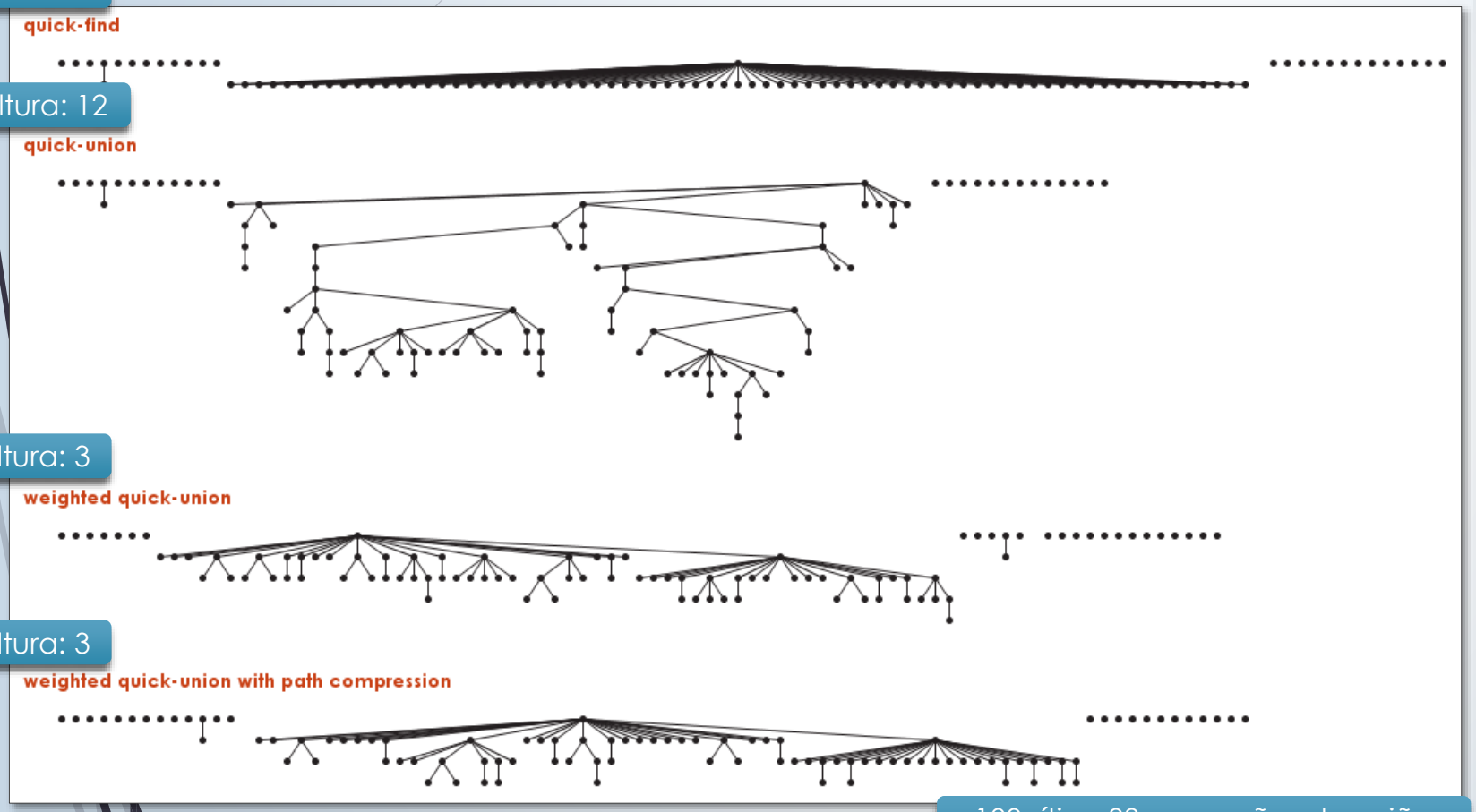
Union-Find - Implementação



Quick-union and weighted quick-union (100 sites, 88 `union()` operations)

Estudo de Caso
Union-Find - Implementação

altura: 2



altura: 12

altura: 3

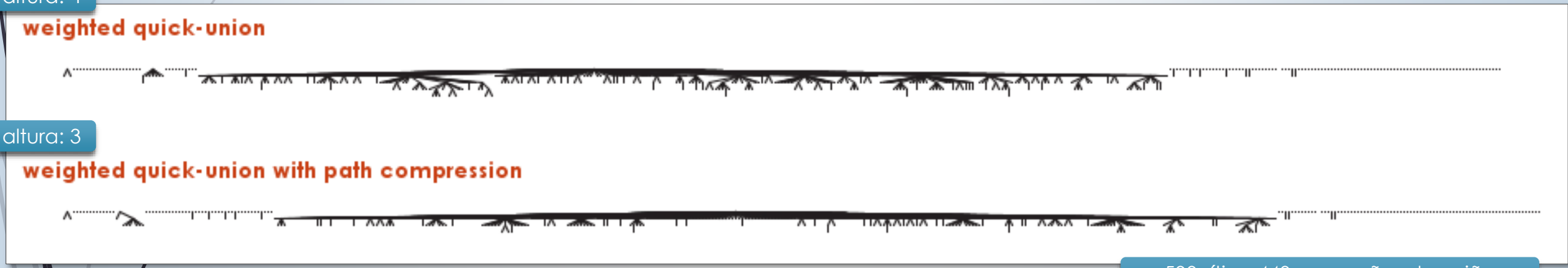
altura: 3

100 sítios, 88 operações de união

```java
public class WeightedQuickUnionPathCompressionUF {

    private int[] parent;
    // rank[i] = ranque da subárvore enraizada em i
    private byte[] rank;

    public int find( int p ) {

        while ( p != parent[p] ) {
            // compressão do caminho pela metade
            parent[p] = parent[parent[p]];
            p = parent[p];
        }

        return p;

    }

    public void union( int p, int q ) {

        int rootP = find( p );
        int rootQ = find( q );

        if ( rootP == rootQ ) {
            return;
        }

        // faz a raiz de menor ranque apontar para a
        // raiz de maior ranque
        if ( rank[rootP] < rank[rootQ] ) {
            parent[rootP] = rootQ;
        } else if ( rank[rootP] > rank[rootQ] ) {
            parent[rootQ] = rootP;
        } else {
            parent[rootQ] = rootP;
            rank[rootP]++;
        }

        count--;

    }

}
```

Estudo de Caso
Union-Find - Implementação

altura: 4



500 sítios, 440 operações de união

Estudo de Caso
Union-Find - Implementação

algorithm	order of growth for N sites (worst case)		
	constructor	union	find
quick-find	N	N	1
quick-union	N	*tree height*	*tree height*
weighted quick-union	N	$\lg N$	$\lg N$
weighted quick-union with path compresson	N	*very, very nearly, but not quite* 1 *(amortized)* (see EXERCISE 1.5.13)	
impossible	N	1	1

Performance characteristics of union-find algorithms

Union-Find
Exercícios

- **Exercício e1.1:** Na seção 1.5, página 216, da obra **Algorithms** (SEDGEWICK; WAYNE, 2011) o problema da conectividade dinâmica é tratado. Leia esse estudo de caso para complementar o que foi visto em aula. Após a leitura, verifique o projeto **"Percolacao"**, disponibilizado no material do curso e tente entender como foi aplicada a solução do Union-Find na modelagem do problema de percolação, também apresentado em aula.

Bibliografia

SEDGEWICK, R.; WAYNE, K. **Algorithms**. 4. ed. Boston: Pearson Education, 2011. 955 p.

GOODRICHM M. T.; TAMASSIA, R. **Estruturas de Dados & Algoritmos em Java**. Porto Alegre: Bookman, 2013. 700 p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos – Teoria e Prática**. 3. ed. São Paulo: GEN LTC, 2012. 1292 p.