

Trabalho final EDA II

Union-Find

Demonstração da estrutura de dados Union-Find

Apresentado por:

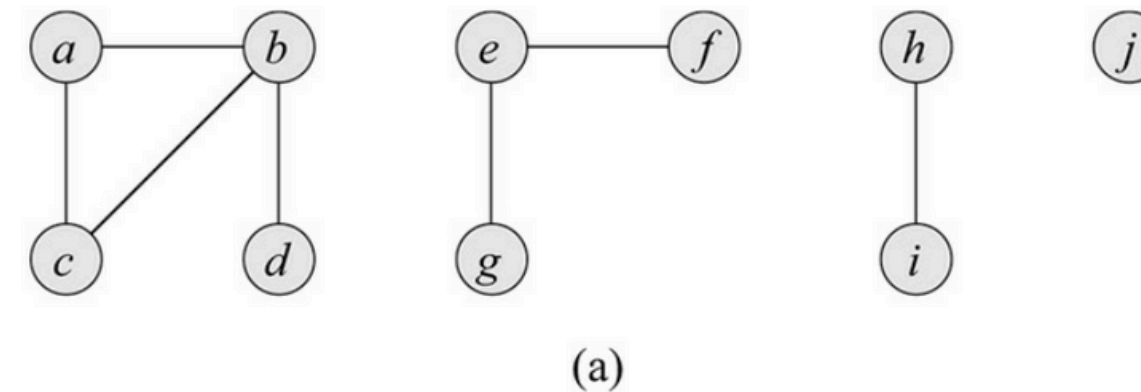
Arthur Batista dos Santos Borges

Malu Pinto de Brito

Tiago Almeida de Oliveira

Union-Find: Introdução

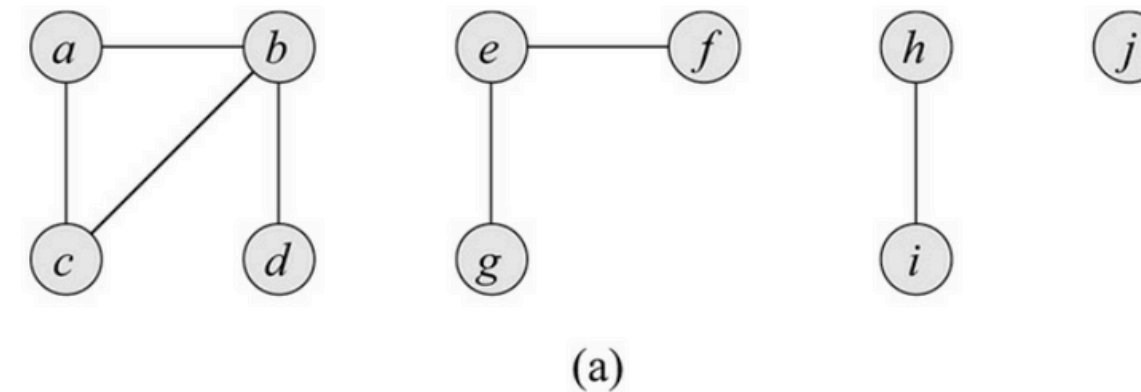
- **Ferramenta Fundamental:** Essencial para lidar com conjuntos dinâmicos e disjuntos.
- **Desafio:** Unir grupos e identificar a qual grupo um elemento pertence, de forma eficiente.
- **Representante:** Cada conjunto tem um identificador único (o representante).
- **Operações:** MAKE-SET, UNION e FIND-SET.
- **Eficiência:** Operações em tempo quase constante!



Aresta processada	Coleção de conjuntos disjuntos									
conjuntos iniciais	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

Union-Find: Estrutura de Dados para Conjuntos Disjuntos

- **Agrupamento Dinâmico:** Essencial em aplicações que exigem organizar elementos em conjuntos exclusivos.
- **Conectividade:** Determinar relações de conectividade de forma eficiente.
- **Exemplo 1:** Componentes Conexas
- **Exemplo 2:** Algoritmo de Kruskal
- **Resumo:** Resolver problemas de agrupamento dinâmico e relações de conectividade.



Aresta processada	Coleção de conjuntos disjuntos									
conjuntos iniciais	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

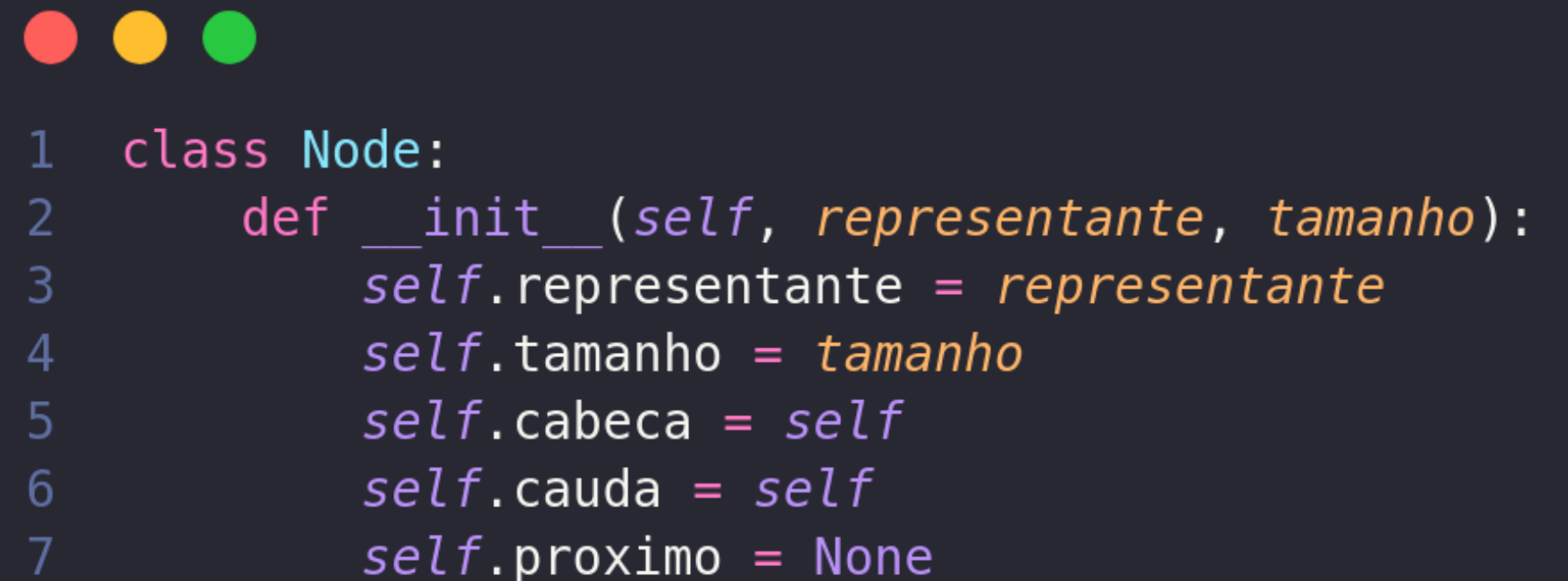
Union-Find

- Mantém uma coleção $S = \{S_1, S_2, \dots, S_k\}$ de conjuntos dinâmicos disjuntos.
- **Representante:** Identificamos cada conjunto por um representante, que é algum membro do conjunto.
- Representamos cada elemento de um conjunto por um objeto.

```
1 class Node:
2     def __init__(self, representante, tamanho):
3         self.representante = representante
4         self.tamanho = tamanho
5         self.cabeca = self
6         self.cauda = self
7         self.proximo = None
```

Union-Find: Operações Essenciais

- **MAKE-SET(x)**: Criando um Novo Conjunto.
- **FIND-SET(x)**: Encontrando o Representante.
- **UNION(x, y)**: Unindo Dois Conjuntos.



```
1 class Node:
2     def __init__(self, representante, tamanho):
3         self.representante = representante
4         self.tamanho = tamanho
5         self.cabeca = self
6         self.cauda = self
7         self.proximo = None
```

Union-Find: Operações Essenciais

➤ **MAKE-SET(x):** Criando um Novo Conjunto.



```
1 class UnionFind:
2     def __init__(self):
3         self.sets = {} # Dicionário para armazenar os conjuntos (Node)
4
5     def makeset(self, x):
6         node = Node(x, 1)
7         self.sets[x] = node
```

Union-Find: Operações Essenciais

➤ **FIND-SET(x)**: Encontrando o Representante.



```
1 class UnionFind:
2     def __init__(self):
3         self.sets = {} # Dicionário para armazenar os conjuntos (Node)
4
5     def findset(self, x):
6         return self.sets[x].representante
```


Union-Find: Operações Essenciais

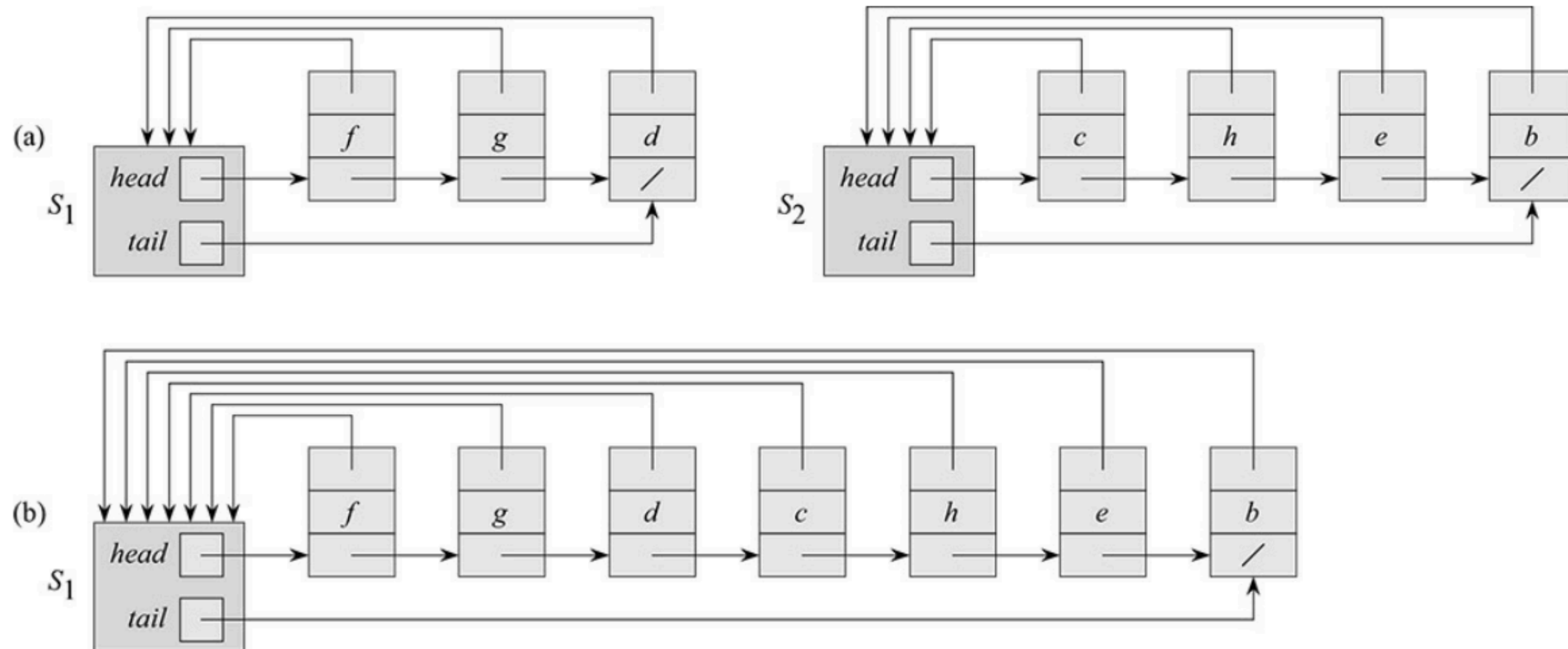
➤ **UNION(x, y):** Unindo Dois Conjuntos.

```
1 class UnionFind:
2     def __init__(self):
3         self.sets = {} # Dicionário para armazenar os conjuntos (Node)
4
5     def union(self, x, y):
6         X = self.sets[self.findset(x)]
7         Y = self.sets[self.findset(y)]
8         if X.representante == Y.representante:
9             return
10
11         if X.tamanho < Y.tamanho:
12             current = X.cabeca
13             while current is not None:
14                 current.representante = Y.representante
15                 current = current.proximo
16             Y.tamanho = X.tamanho + Y.tamanho
17             X.tamanho = 0
18             Y.cauda.proximo = X.cabeca
19             Y.cauda = X.cauda
20             X.cabeca = None
```

```
1 else:
2     current = Y.cabeca
3     while current is not None:
4         current.representante = X.representante
5         current = current.proximo
6     X.tamanho = X.tamanho + Y.tamanho
7     Y.tamanho = 0
8     X.cauda.proximo = Y.cabeca
9     X.cauda = Y.cauda
10    Y.cabeca = None
```

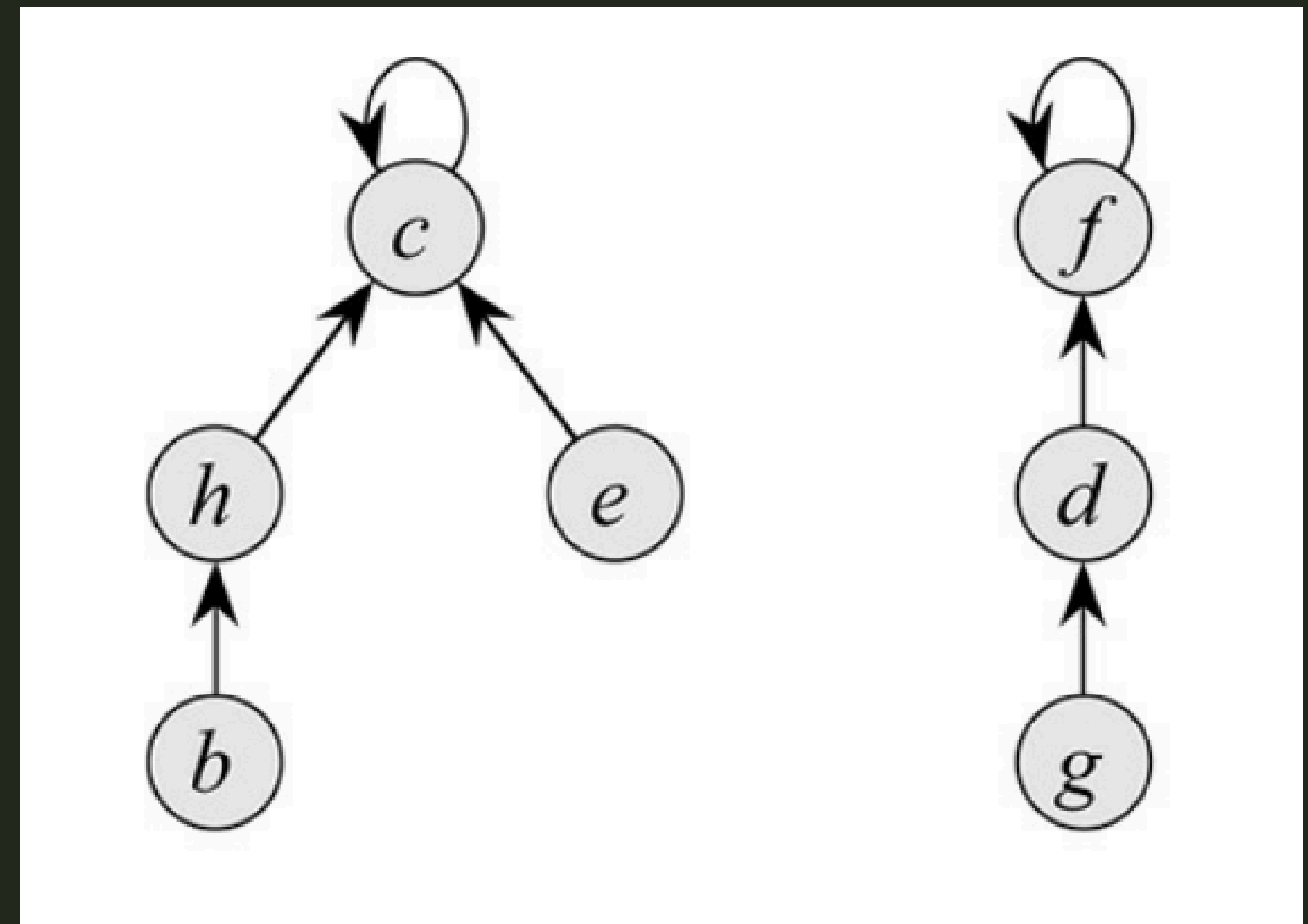

Union-Find: Operações Essenciais

► Listas Ligadas: Análise



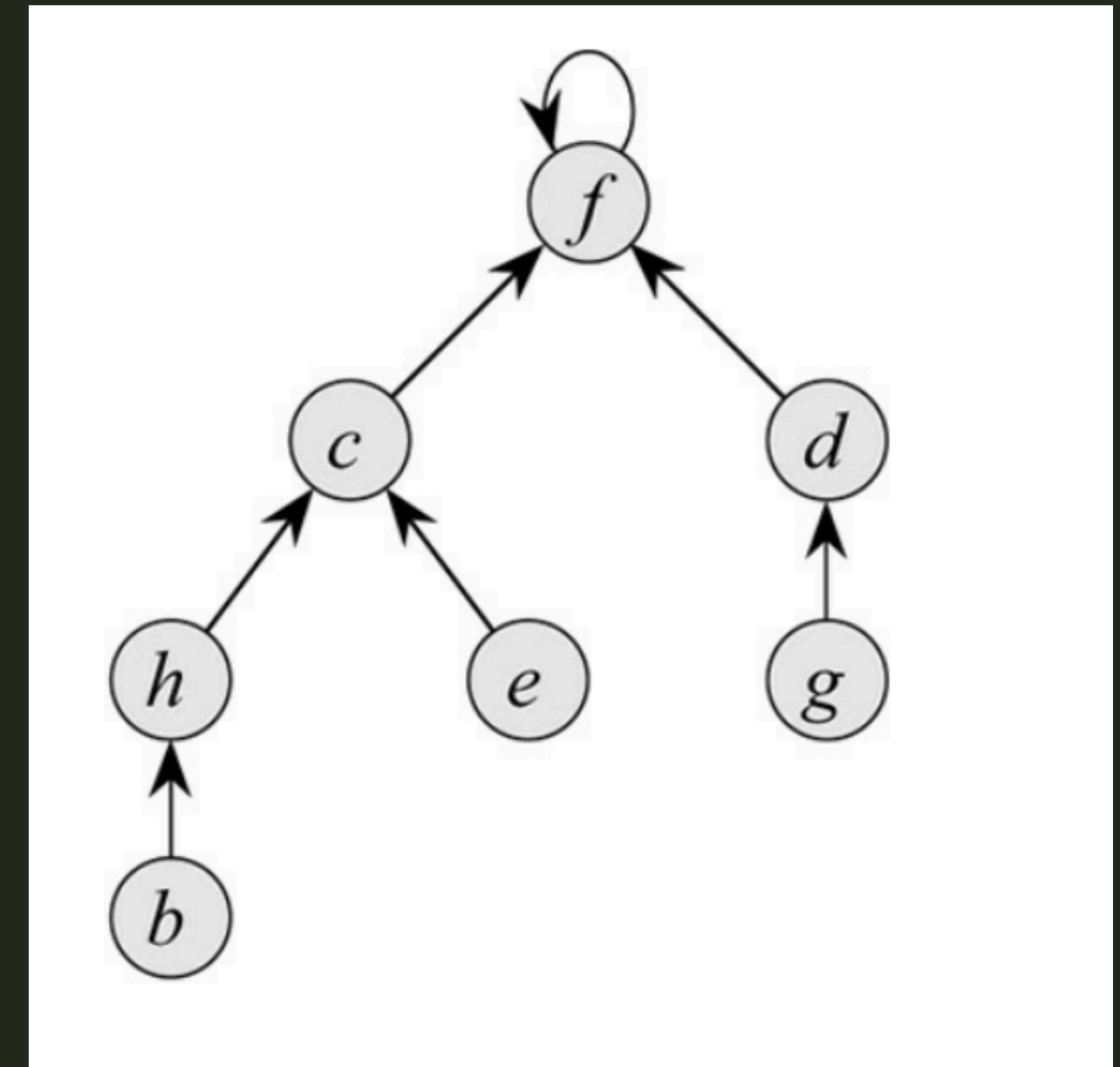
Union-Find com Árvores: Uma Nova Abordagem

- Em vez de listas ligadas, representamos conjuntos como **árvores enraizadas**.
- Cada **nó (elemento)** aponta para seu nó **pai**.
- A **raiz** da árvore representa o **conjunto**.



Operações Fundamentais em Florestas de Conjuntos Disjuntos

- **MAKE-SET(x)**: Cria uma nova árvore com apenas o nó 'x' (raiz).
- **FIND-SET(x)**: Percorre a árvore seguindo os ponteiros dos pais até encontrar a raiz.
 - **Caminho de localização**: o caminho percorrido até a raiz.
- **UNION(x, y)**: Faz a raiz de uma árvore apontar para a raiz da outra.



Resultado de UNION(e, g).

Otimização 1: União pelo Posto (Union by Rank)

- **Posto:** Para evitar árvores altas, mantemos um 'posto' (rank) para cada árvore.
- **Union:** Durante a operação UNION, sempre anexamos a árvore de menor rank à raiz da árvore de maior rank.
- **Incrementando Rank:** Se as raízes tiverem o mesmo rank, incrementamos o rank da nova raiz.

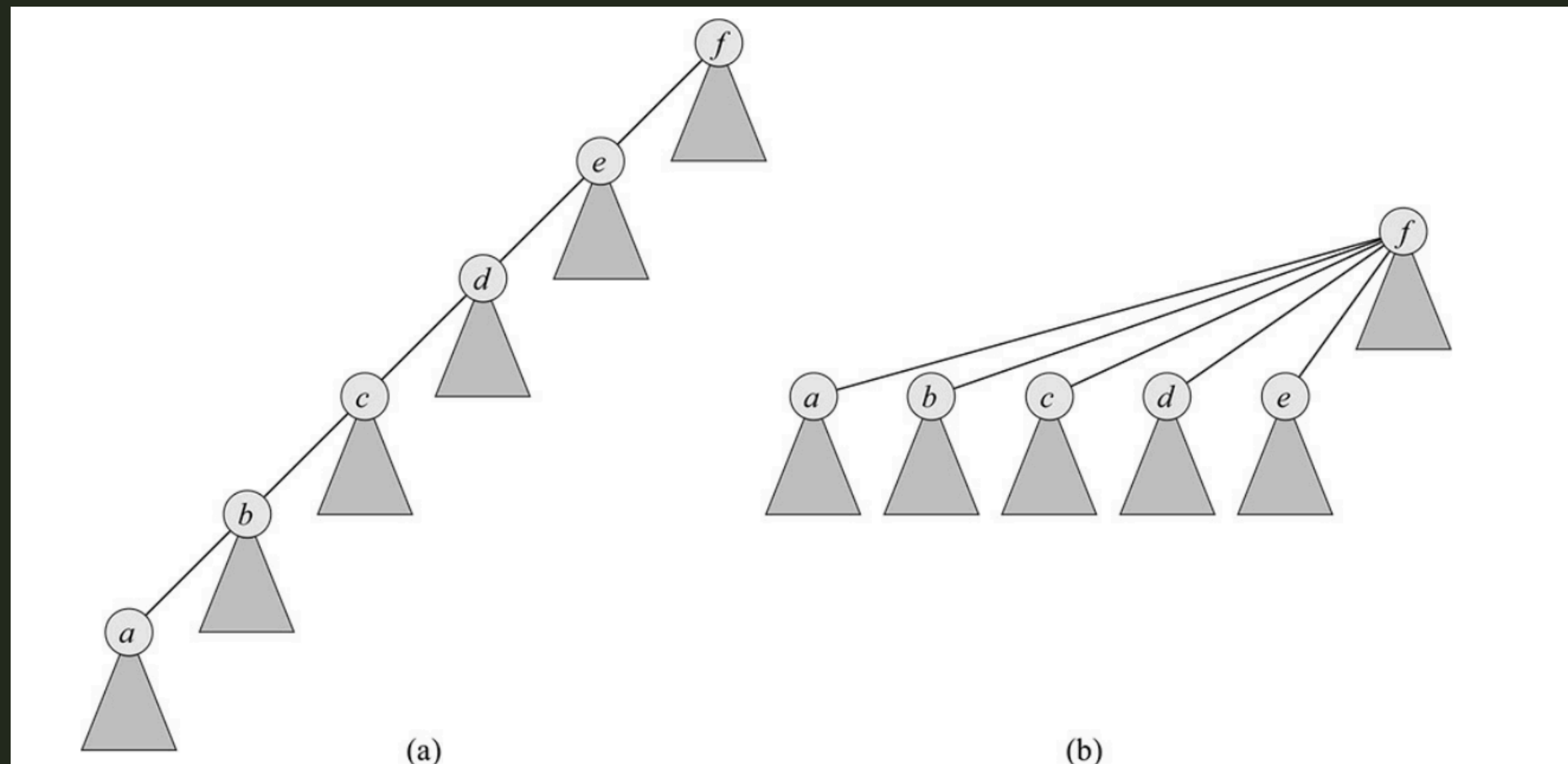
```
1 def union(self, x, y):
2     """
3     Une os conjuntos que contêm os elementos 'x' e 'y'.
4     Usa 'union by rank' para manter as árvores balanceadas.
5     """
6     root_x = self.find(x)
7     root_y = self.find(y)
8
9     if root_x != root_y: # Se os elementos estão em conjuntos diferentes
10        if self.rank[root_x] < self.rank[root_y]:
11            self.parent[root_x] = root_y # Anexa a árvore de menor rank à de maior rank
12        elif self.rank[root_x] > self.rank[root_y]:
13            self.parent[root_y] = root_x
14        else:
15            self.parent[root_y] = root_x # Se os ranks são iguais, anexa e incrementa o rank
16            self.rank[root_x] += 1
17
```

Otimização 2: Compressão de Caminho (Path Compression)

- Para tornar o **FIND-SET** ainda mais rápido, aplicamos a **compressão de caminho**.
- Durante o **FIND-SET**, fazemos cada nó no caminho de busca apontar diretamente para a raiz.
- A **compressão de caminho** 'achata' a árvore, **otimizando futuras buscas**.

```
1 def find(self, x):  
2     """  
3     Encontra o representante (raiz) do conjunto ao qual o elemento 'x' pertence.  
4     Aplica 'path compression' para otimizar futuras buscas.  
5     """  
6     if self.parent[x] != x:  
7         self.parent[x] = self.find(self.parent[x]) # Compressão de Caminho  
8     return self.parent[x]
```

Otimização 2: Compressão de Caminho (Path Compression)



Antes e depois da compressão de caminhos

Efeito das heurísticas sobre o tempo de execução

- União por Posto (Sozinha):
 - Tempo de execução: $O(m \log n)$
 - Limite superior justo.
- Compressão de Caminho (Sozinha):
 - Tempo de execução do pior caso: $\Theta(n + f \cdot (1 + \log_2(f/n)))$.
- União por Posto + Compressão de Caminho:
 - Tempo de execução do pior caso: $O(m \alpha(n))$.
- $\alpha(n)$ cresce MUITO lentamente.
- Na prática: $\alpha(n) \leq 4$ para todos os casos!
- Tempo de execução \approx Linear em m

Referências

- [1] CORMEN, Thomas. **Algoritmos – Teoria e Prática**. 3.ed. Rio de Janeiro: LTC, 2022.
- [2] SEDGEWICK, Robert. **Algorithms in C**. 2. ed. Addison Wesley Longman, 1990.
- [3] THOTA, Saigopal et al. **Building Graphs at a Large Scale: Union Find Shuffle**. Walmart Labs, 2021.
- [4] CHEN, Y. et al. **Asynchronous and Load-Balanced Union-Find for Distributed and Parallel Scientific Data Visualization and Analysis**. 2020.
- [5] ARTHUR, G. et al. **Simpler Analyses of Union-Find**. 2023.



FIM