# Estruturas de Dados para Conjuntos Disjuntos

Fernando Lobo

Algoritmos e Estrutura de Dados



## Estruturas de dados para conjuntos disjuntos

- Também conhecido por UNION-FIND.
- Objectivo: Manter uma coleção $S = \{S_1, S_2, \ldots, S_k\}$ de conjuntos disjuntos, que se modificam ao longo do tempo.
- Como modificação apenas é permitido a união de conjuntos (retirar elementos ou partir um conjunto em dois não é permitido).

## Estruturas de dados para conjuntos disjuntos

- Cada conjunto é identificado por um representante.
- A escolha do representante é irrelevante. O importante é que se tentarmos obter o representante de um conjunto duas vezes (sem modificar o conjunto), obtemos sempre a mesma resposta.
- Esta estrutura de dados tem muitas aplicações. Acabamos de ver uma delas na aula passada.



## Operações

- MAKE-SET($x$): cria $S_i = \{x\}$ e acrescenta-o a $S$.
- FIND-SET($x$): retorna um indentificador do conjunto que contém $x$.
- UNION($x, y$): se $x \in S_i$ e $y \in S_j$, então $S = S - S_i - S_j \cup \{S_i \cup S_j\}$

## Análise

A análise será feita em termos de:

- $n =$ nº total de elementos $=$ nº de MAKE-SETs.
- $m =$ nº total de operações.
  - $m \geq n$ porque MAKE-SETs contam para o número total de operações.
  - Só podemos ter no máximo $n - 1$ UNIONs
    (depois disso só nos resta um conjunto.)



## Exemplo de aplicação: Componentes conexas de um grafo

```
CONNECTED-COMPONENTS(G)
    for each v ∈ G.V
        MAKE-SET(v)
    for each (u, v) ∈ G.E
        if FIND-SET(u) ≠ FIND-SET(v)
            UNION(u, v)
```

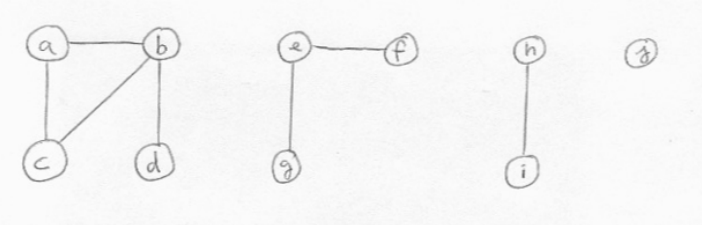
# Exemplo de aplicação: Componentes conexas de um grafo

Uma vez obtidas as componentes conexas, a função SAME-COMPONENT permite verificar se dois nós estão na mesma componente.

```
SAME-COMPONENT(u, v)
    if FIND-SET(u) == FIND-SET(v)
        return TRUE
    else
        return FALSE
```



# Exemplo



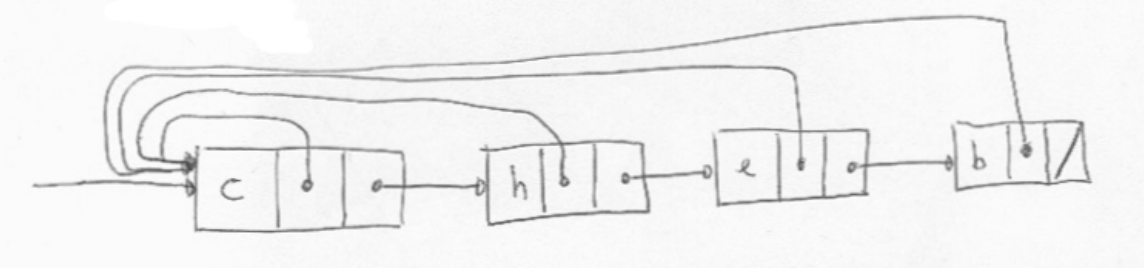

| | Coleção de conjuntos disjuntos |
|---|---|
| | {a} {b} {c} {d} {e} {f} {g} {h} {i} {j} |
| (b,d) | {a} {b,d} {c} {e} {f} {g} {h} {i} {j} |
| (e,g) | {a} {b,d} {c} {e,g} {f} {h} {i} {j} |
| (a,c) | {a,c} {b,d} {e,g} {f} {h} {i} {j} |
| (h,i) | {a,c} {b,d} {e,g} {f} {h,i} {j} |
| (a,b) | {a,b,c,d} {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} |

# Como implementar estas operações?

Primeira abordagem: usar uma lista para cada conjunto.

- Cada elemento da lista tem:
  - um elemento do conjunto.
  - apontador para o representante do conjunto (o primeiro elemento da lista).
  - apontador para o elemento seguinte da lista.
- A lista tem apontador para a cabeça e para a cauda.



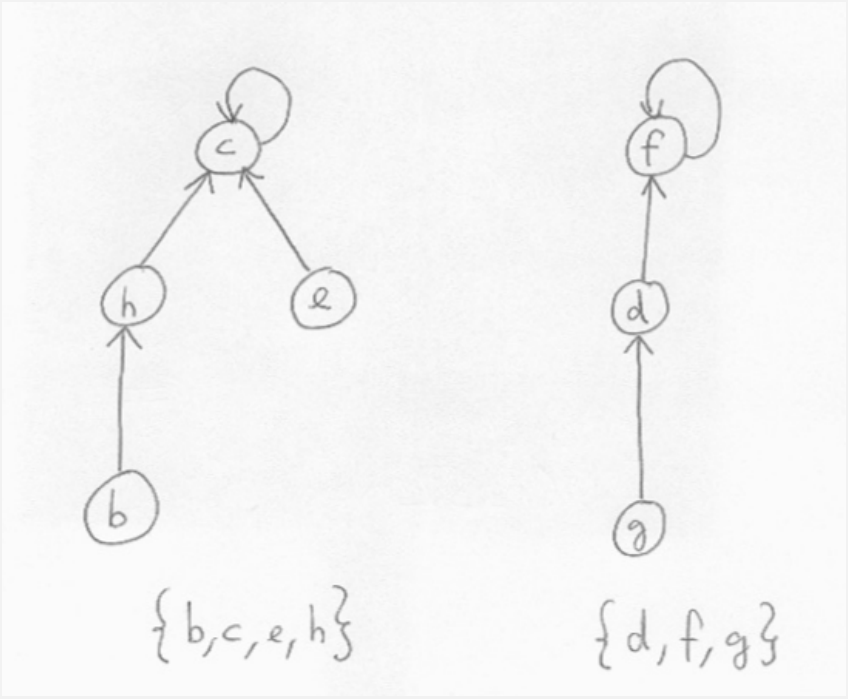



- A abordagem de ter uma lista para cada conjunto não é muito eficiente.
- Deixa-se como exercício pensarem como se poderia implementar as operações: MAKE-SET, FIND-SET, UNION.



## Abordagem alternativa

- Em vez de representar um conjunto por uma lista, representamo-lo por uma árvore invertida.
- Cada nó aponta apenas para o seu pai. A raíz aponta para si própria.
- O elemento que está na raíz é o representante do conjunto.





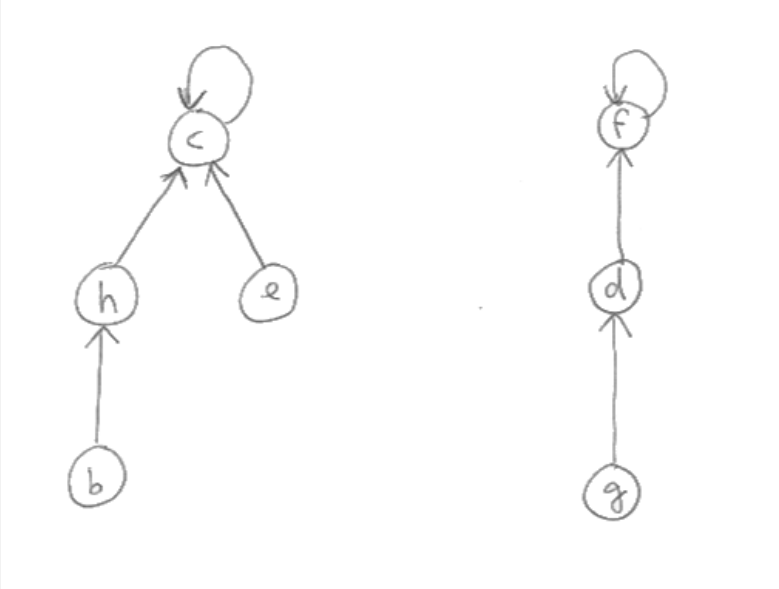

## Abordagem alternativa

- MAKE-SET($x$) → Faz uma árvore com um só nó.


- FIND-SET($x$) → Segue apontadores até chegar à raíz. Retorna o elemento que está na raíz.

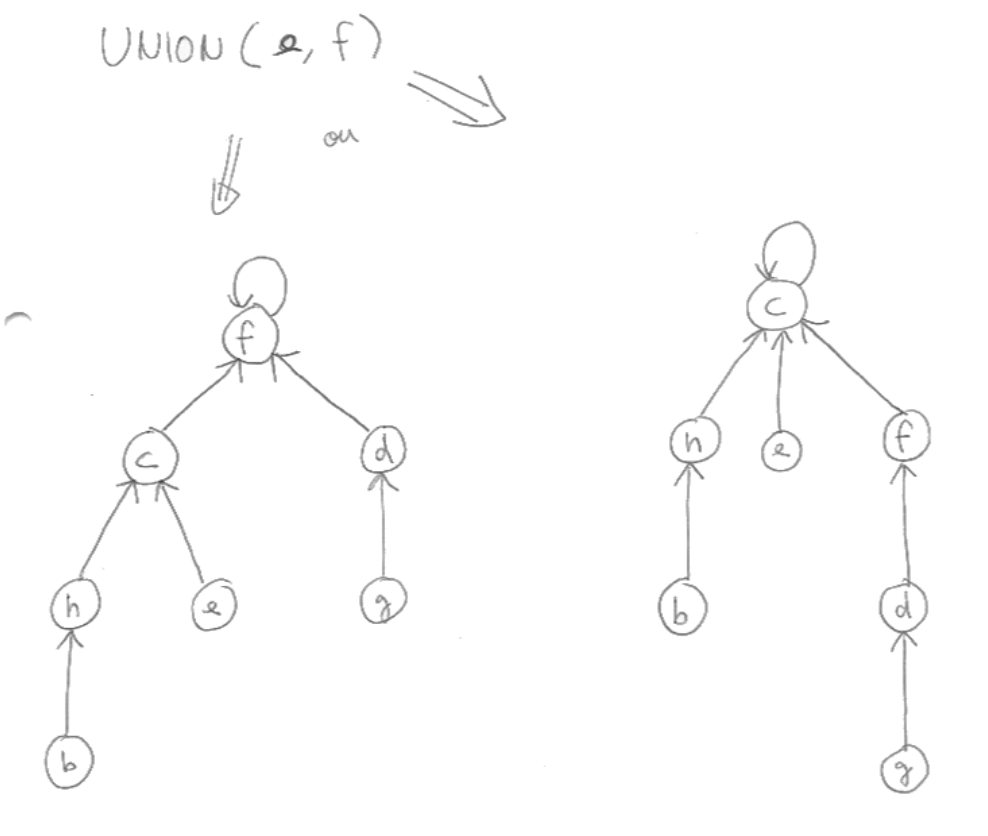
- UNION($x, y$) → Faz a raíz de uma das árvores ficar filha da raíz da outra árvore.

# Exemplo: UNION$(e, f)$






# Exemplo (cont.)



## Melhorando a performance

- Com esta estrutura de dados, também é possível obtermos uma árvore degenerada em lista.
- Podemos evitar isso com duas heurísticas:
  1. Union by rank $\rightarrow$ pendurar a árvore de menor altura na árvore de maior altura.
  2. Path compression $\rightarrow$ ao executar um $\textsc{Find-Set}(x)$ rearranja-se a árvore de modo a que todos os nós no caminho de x até à raíz passem a apontar directamente para a raíz.



## Union by rank

- Usamos o *rank* da raíz $\rightarrow$ um limite superior para a altura da árvore. (Por enquanto pensem no rank como sendo a altura da árvore.)
- Ao fazer a união, compara-se o rank das raízes. Pendura-se a de menor rank como filha da raíz da de maior rank.
- A ideia é evitar que a altura da árvore resultante cresça demasiado.

# Implementação

- A implementação é muito simples.
- Cada nó só necessita de saber quem é o seu pai e o seu rank.
  $\Longrightarrow$ basta ter um array para representar a floresta.



# Pseudocódigo para *union by rank*

MAKE-SET($x$)

```
parent[x] = x
rank[x] = 0
```

FIND-SET($x$)

```
while x ≠ parent[x]
      x = parent[x]
return x
```

## Pseudocódigo para *union by rank*

```
UNION(x, y)
    rx = FIND-SET(x)
    ry = FIND-SET(y)
    if rx == ry
        return
    if rank[rx] > rank[ry]
        parent[ry] = rx
    else
        parent[rx] = ry
        if rank[rx] == rank[ry]
            rank[ry] = rank[ry] + 1
```



## Observações sobre a heurística *union by rank*

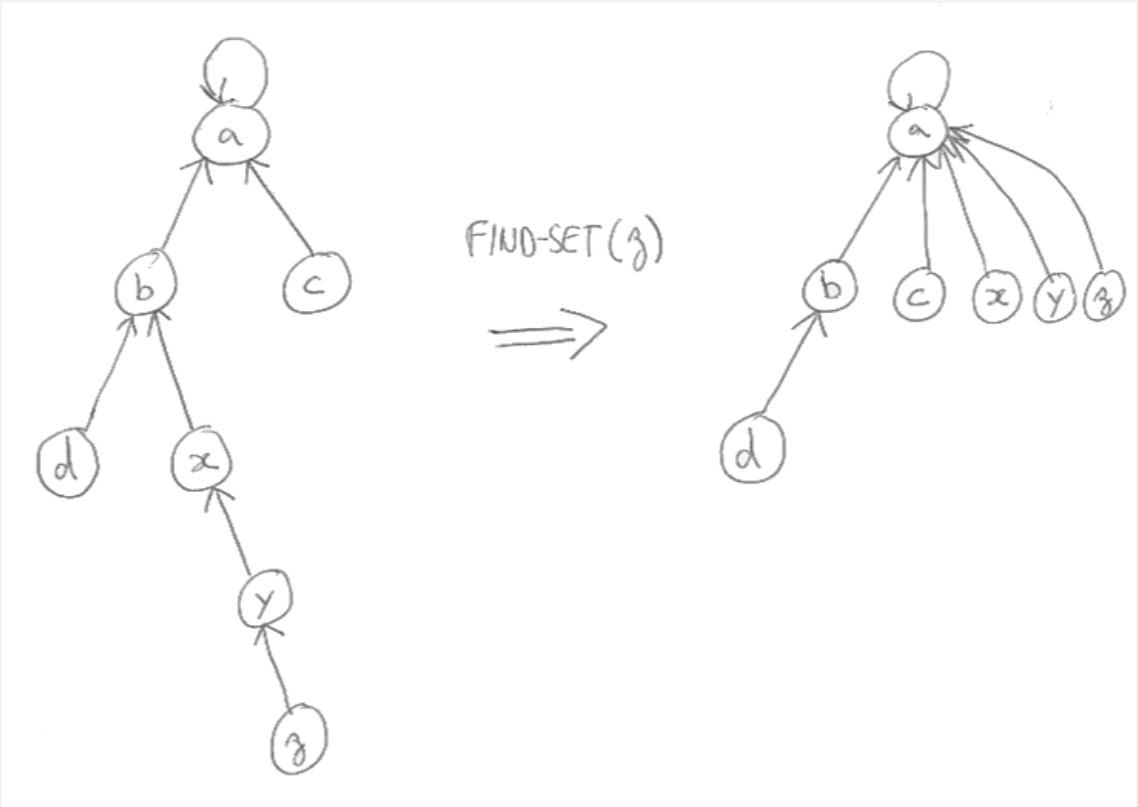
- O *rank* de um nó é a altura da sub-árvore que tem esse nó como raíz.
- **Propriedade 1:** Para qualquer nó $x$ excepto a raíz, $rank[x] < rank[parent[x]]$
  (porque um nó raíz de rank $k$ é criado por junção de duas árvores com raízes de rank $k-1$)
- **Propriedade 2:** Qualquer nó raíz de rank $k$ tem pelo menos $2^k$ nós na sua árvore. (pode ser facilmente demonstrado por indução.)
- **Propriedade 3:** Se tivermos $n$ elementos, só poderá haver no máximo $n/2^k$ nós com rank $k$. ($\implies$ rank máximo é $\lg n$ $\implies$ todas as árvores têm altura $\leq \lg n$.)

## Path compression

- Ao fazer um FIND-SET($x$) temos de percorrer o caminho do nó $x$ até à raíz da árvore.
- Podemos aproveitar isso para pendurar todos os nós ao longo do caminho directamente como filhos da raíz.
- Pode ser feito gastando apenas tempo constante por elemento ao longo do caminho de $x$ até à raíz.
- Todas as operações FIND-SET subsequentes sobre elementos que estavam no caminho de $x$ até à raíz, serão feitas de forma mais rápida.
- As árvores tornam-me mais largas e menos profundas.



## Path compression. Exemplo



## Pseudocódigo para *path compression*

```
FIND-SET(x)
    if x ≠ parent[x]
        parent[x] = FIND-SET(parent[x])
    return parent[x]
```

- MAKE-SET($x$) and UNION($x, y$) ficam inalterados.



## Observações sobre *path compression*

- Os ranks dos nós não se alteram devido à operação de path compression.
- Mas o rank deixa de poder ser interpretado como sendo a altura da árvore.
- O rank passa a ser um limite superior para altura da árvore.

## Complexidade

- Pode-se provar que a complexidade temporal de uma sequência de $m$ operações sobre $n$ elementos é $O(m \lg^* n) \implies$ O custo amortizado por operação é $O(\lg^* n)$
  - $m$ é o número total de operações (MAKE-SETs, FIND-SETs e UNIONs).
  - $n$ é o número de MAKE-SETs.
- $\lg^* n$ é a função iterated log, o número de aplicações consecutivas da função logaritmo que é necessário aplicar a $n$ para obter um número igual ou inferior a 1.



## $\lg^*$ cresce de forma muito lenta

$$\begin{array}{lcccl} \lg^* 2 & = & 1 & & \\ \lg^* 4 & = & \lg^* 2^2 & = & 2 \\ \lg^* 16 & = & \lg^* 2^{2^2} & = & 3 \\ \lg^* 65536 & = & \lg^* 2^{2^{2^2}} & = & 4 \\ \lg^* 2^{65536} & = & \lg^* 2^{2^{2^{2^2}}} & = & 5 \end{array}$$

- $2^{65536} = 2^{2^{2^{2^2}}}$ é ENORME, muito maior que o nº de átomos do universo! $\implies$ Em qualquer aplicação prática, $\lg^* n \leq 5$.
- O tempo gasto para uma sequência de $m$ operações é um nadinha superior a linear em $m$. (Para efeitos práticos é linear.)
- O custo amortizado por operação requer, para efeitos práticos, tempo constante.

# Revisitando a complexidade do algoritmos de Kruskal

- Primeiro ciclo **for**: $O(V)$ MAKE-SETs
- Ordenar $E$: $O(E \lg E)$
- Segundo ciclo **for**: $O(E)$ FIND-SETs e UNIONs



# Revisitando a complexidade do algoritmos de Kruskal

Com a implementação de UNION-FIND com union by rank e path compression:

- Primeiro ciclo **for**: $O(V)$
- Segundo ciclo **for**: $O(E \lg^* E)$
  (para efeitos práticos, $\lg^* E \leq 5 = O(1)$.
- A complexidade total é dominada pela ordenação das arestas: $O(E \lg E) = O(E \lg V)$. Porquê? Porque $|E| \leq |V|^2$

$$\begin{aligned} \implies \lg |E| &\leq \lg |V|^2 \\ &= 2 \lg |V| \\ &= O(\lg |V|) \end{aligned}$$