Estrutura de Dados II (ED2)

Aula 02 – Introdução (parte 2)

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

(Material baseado nos slides do Professor Eduardo Zambon)

Quick-union

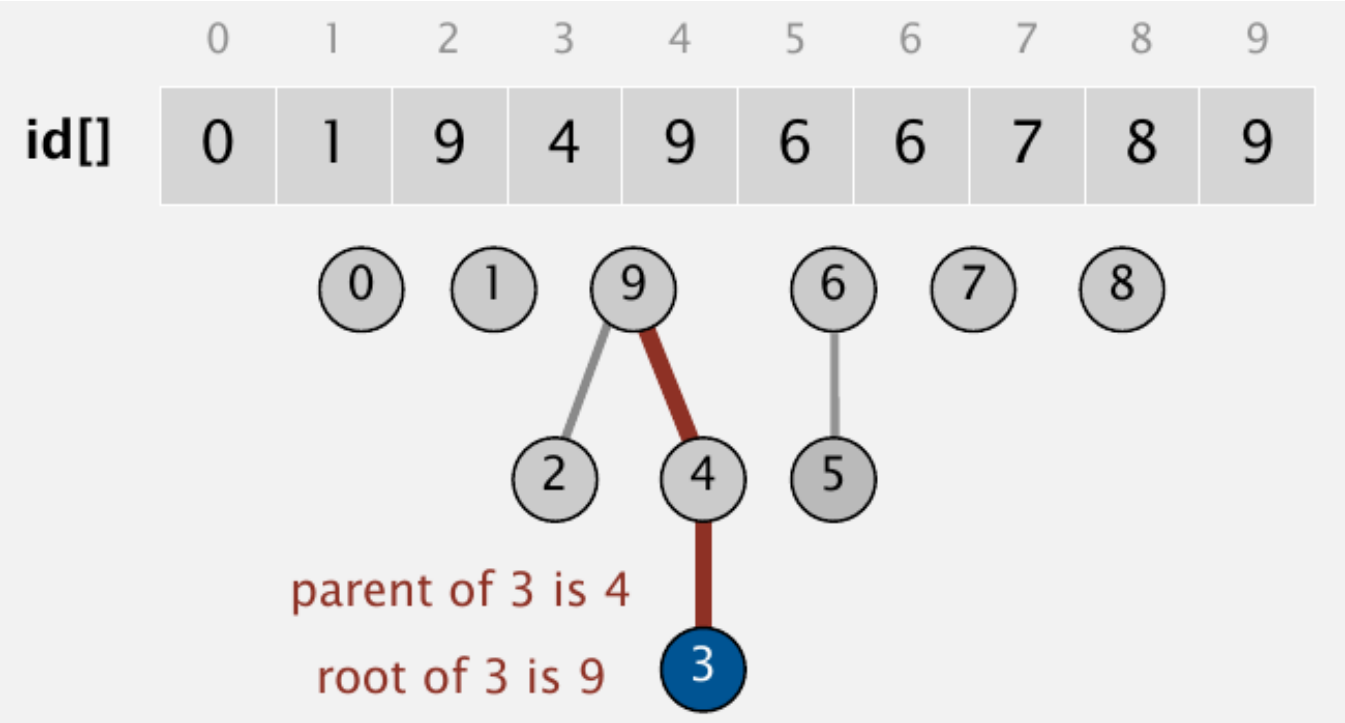
Estrutura de dados:

- Um *array* de inteiros `id[]` de tamanho *N*.
- Interpretação: `id[i]` é o pai de *i*.
- Raiz de *i* é `id[id[...id[i]...]]`.
  (Algoritmo garante ausência de ciclos.)

Quick-union

Estrutura de dados:

- Um *array* de inteiros `id[]` de tamanho *N*.
- Interpretação: `id[i]` é o pai de *i*.
- Raiz de *i* é `id[id[...id[i]...]]`.
  (Algoritmo garante ausência de ciclos.)



Quick-union

Operações:

- *Find*: Qual é a raiz de *p*?
- *Connected*: Os objetos *p* e *q* têm a mesma raiz?

```
Raiz de 3 é 9; Raiz de 5 é 6;
=> 3 e 5 não estão conectados
```

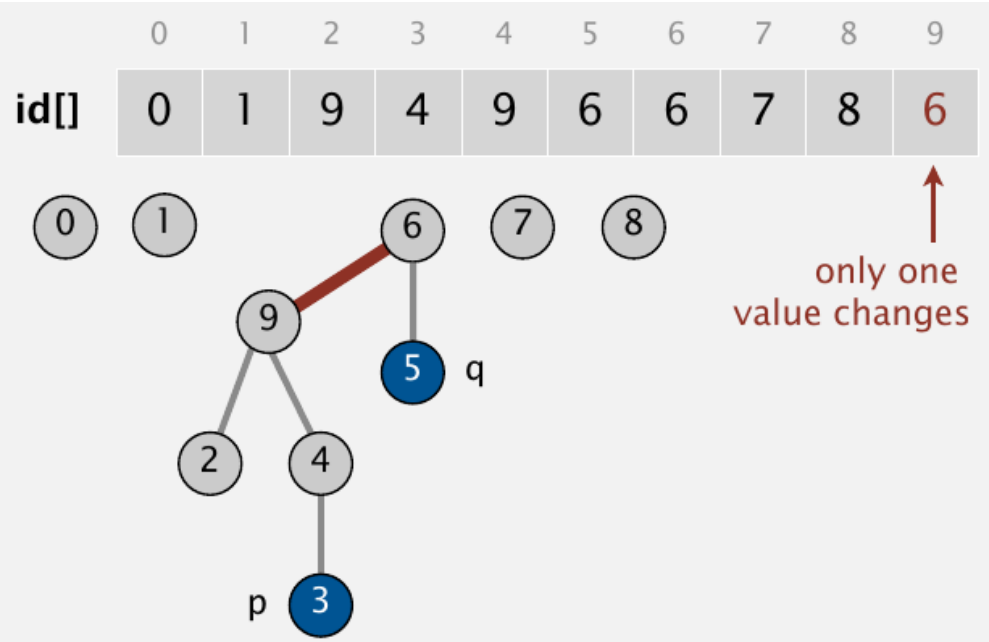
Quick-union

Operações:

- *Find*: Qual é a raiz de *p*?
- *Connected*: Os objetos *p* e *q* têm a mesma raiz?

  `Raiz de 3 é 9; Raiz de 5 é 6;`

  `=> 3 e 5 não estão conectados`
- *Union*: Para unir as componentes contendo *p* e *q*, modifique o `id` da raiz de *p* para o `id` da raiz de *q*.



Quick-union – Demo

Ver arquivo `15DemoQuickUnion.mov`

Quick-union – Implementação

```
static int id[1000];
static int N;

void UF_init(int size) {
  N = size;
  for (int i = 0; i < N; i++) {
    id[i] = i;  // Cada objeto comeca na sua propria componente.
  }             // N acessos ao array.
}

int UF_find(int i) {
  while (i != id[i]) i = id[i]; // Buscar o pai ate a raiz.
  return i;                     // Profundidade de i acessos.
}

void UF_union(int p, int q) {
  int i = UF_find(p); // Modifique raiz de p para a raiz de q.
  int j = UF_find(q); // Profundidade de p+q acessos.
  id[i] = j;
}
```

Quick-union também é muito lento

Modelo de custo: número de acessos ao *array* (igual).

Ordem de crescimento do número de acessos por operação.

Algoritmo	`init`	`union`	`find`	`connected`
quick-find	N	N	1	1
quick-union	N	$N^{\dagger}$	N	N

† inclui o custo de encontrar as raízes

Quick-union também é muito lento

Modelo de custo: número de acessos ao *array* (igual).

Ordem de crescimento do número de acessos por operação.

Algoritmo	`init`	`union`	`find`	`connected`
quick-find	*N*	*N*	1	1
quick-union	*N*	*N*†	*N*	*N*

† inclui o custo de encontrar as raízes

Defeito do *quick-find*:

- União muito custosa: *N* acessos ao vetor.
- As árvores são achatadas mas é muito custoso mantê-las assim.

Quick-union também é muito lento

Modelo de custo: número de acessos ao *array* (igual).

Ordem de crescimento do número de acessos por operação.

Algoritmo	`init`	`union`	`find`	`connected`
quick-find	N	N	1	1
quick-union	N	$N^{\dagger}$	N	N

† inclui o custo de encontrar as raízes

Defeito do *quick-find*:

- União muito custosa: N acessos ao vetor.
- As árvores são achatadas mas é muito custoso mantê-las assim.

Defeito do *quick-union*:

- Árvores podem ficar muito altas.
- *Find/connected* muito custosas: podem chegar a N acessos ao vetor.

Melhoria 1: Pesos

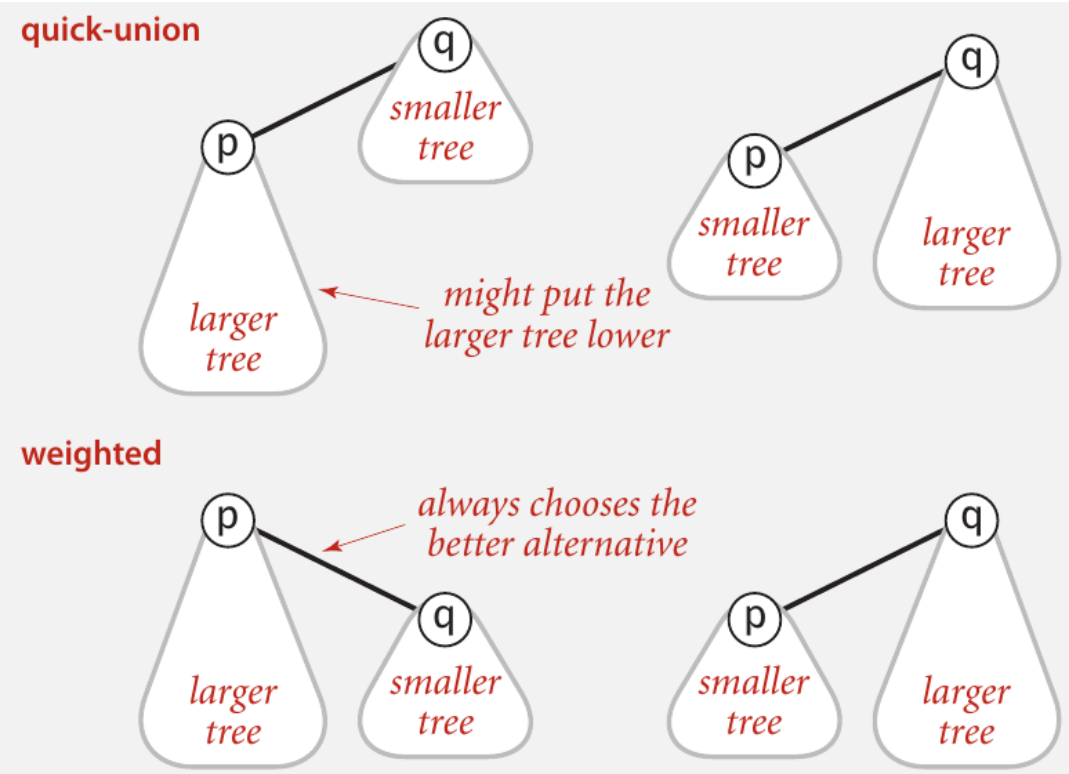
Weighted quick-union:

- Modificar o *quick-union* para evitar árvores altas.
- Manter a informação do tamanho (número de objetos) de cada árvore.
- Equilibrar pendurando a árvore menor sob a árvore maior.

Melhoria 1: Pesos

Weighted quick-union:

- Modificar o *quick-union* para evitar árvores altas.
- Manter a informação do tamanho (número de objetos) de cada árvore.
- Equilibrar pendurando a árvore menor sob a árvore maior.
- Alternativas para “tamanho”: união por altura, *rank*, etc.

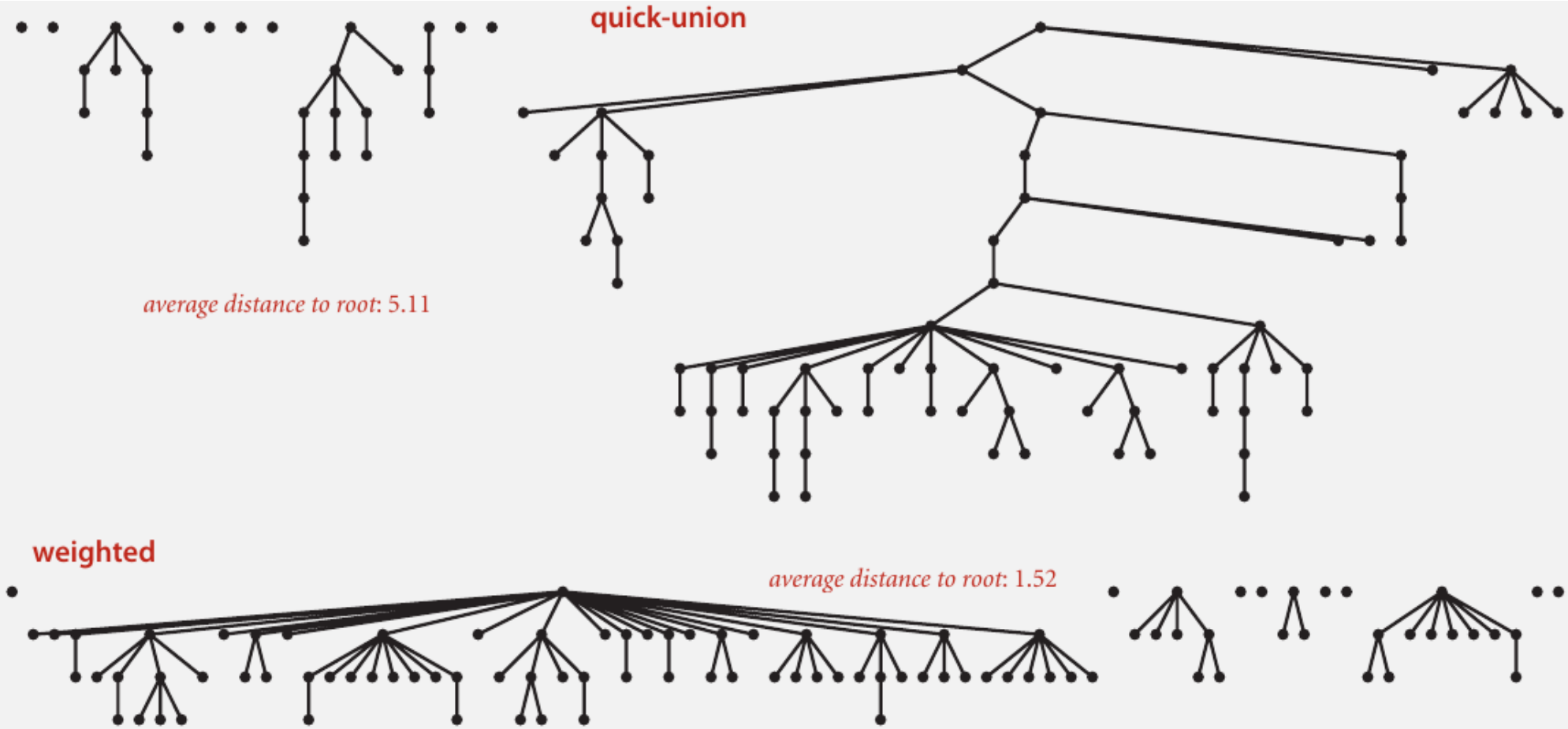


Weighted quick-union – Demo

Ver arquivo `15DemoWeightedQuickUnion.mov`

Exemplo: *quick-union* vs. *weighted quick-union*

100 objetos, 88 operações de união.



Weighted quick-union – Implementação

Estrutura de dados: Igual a *quick-union* mas mantém um *array* adicional `sz[i]` para contar o número de objetos da árvore com raiz em *i*.

Weighted quick-union – Implementação

Estrutura de dados: Igual a *quick-union* mas mantém um *array* adicional `sz[i]` para contar o número de objetos da árvore com raiz em *i*.

Find/connected: Idêntico a *quick-union*.

Union: Modificar *quick-union* para:

- Pendurar a árvore menor sob a maior.
- Atualizar o *array* `sz[]`.

Weighted quick-union – Implementação

Estrutura de dados: Igual a *quick-union* mas mantém um *array* adicional `sz[i]` para contar o número de objetos da árvore com raiz em *i*.

Find/connected: Idêntico a *quick-union*.

Union: Modificar *quick-union* para:

- Pendurar a árvore menor sob a maior.
- Atualizar o *array* `sz[]`.

```
void UF_union(int p, int q) {
  int i = UF_find(p); // Pendure a arvore menor sob a maior.
  int j = UF_find(q); // Profundidade de ? acessos.
  if (i == j) return;
  if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
  else               { id[j] = i; sz[i] += sz[j]; }
}
```

Weighted quick-union – Análise

Tempo de execução:

- *Find*: leva tempo proporcional à profundidade de p.
- *Union*: leva tempo constante, dadas as raízes.

Weighted quick-union – Análise

Tempo de execução:

- *Find*: leva tempo proporcional à profundidade de p.
- *Union*: leva tempo constante, dadas as raízes.

Proposição: A profundidade de qualquer nó da árvore é no máximo $\lg N$. ($\lg$ = logaritmo na base 2.)

Justificativa?

Weighted quick-union – Análise

Tempo de execução:

- *Find*: leva tempo proporcional à profundidade de p.
- *Union*: leva tempo constante, dadas as raízes.

Proposição: A profundidade de qualquer nó da árvore é no máximo $\lg N$. ($\lg$ = logaritmo na base 2.)

Justificativa? União de uma árvore com i nós a uma árvore com j nós aumenta a profundidade da árvore menor de 1, mas a nova árvore tem $i + j$ nós. A propriedade é preservada porque (para $i \leq j$):

$$1 + \lg i = \lg(i + i) \leq \lg(i + j) \quad .$$

Weighted quick-union – Análise

Ordem de crescimento do número de acessos por operação.

Algoritmo	`init`	`union`	`find`	`connected`
quick-find	N	N	1	1
quick-union	N	$N^{\dagger}$	N	N
weighted QU	N	$\lg N^{\dagger}$	$\lg N$	$\lg N$

† inclui o custo de encontrar as raízes

Weighted quick-union – Análise

Ordem de crescimento do número de acessos por operação.

Algoritmo	`init`	`union`	`find`	`connected`
quick-find	N	N	1	1
quick-union	N	$N^{\dagger}$	N	N
weighted QU	N	$\lg N^{\dagger}$	$\lg N$	$\lg N$

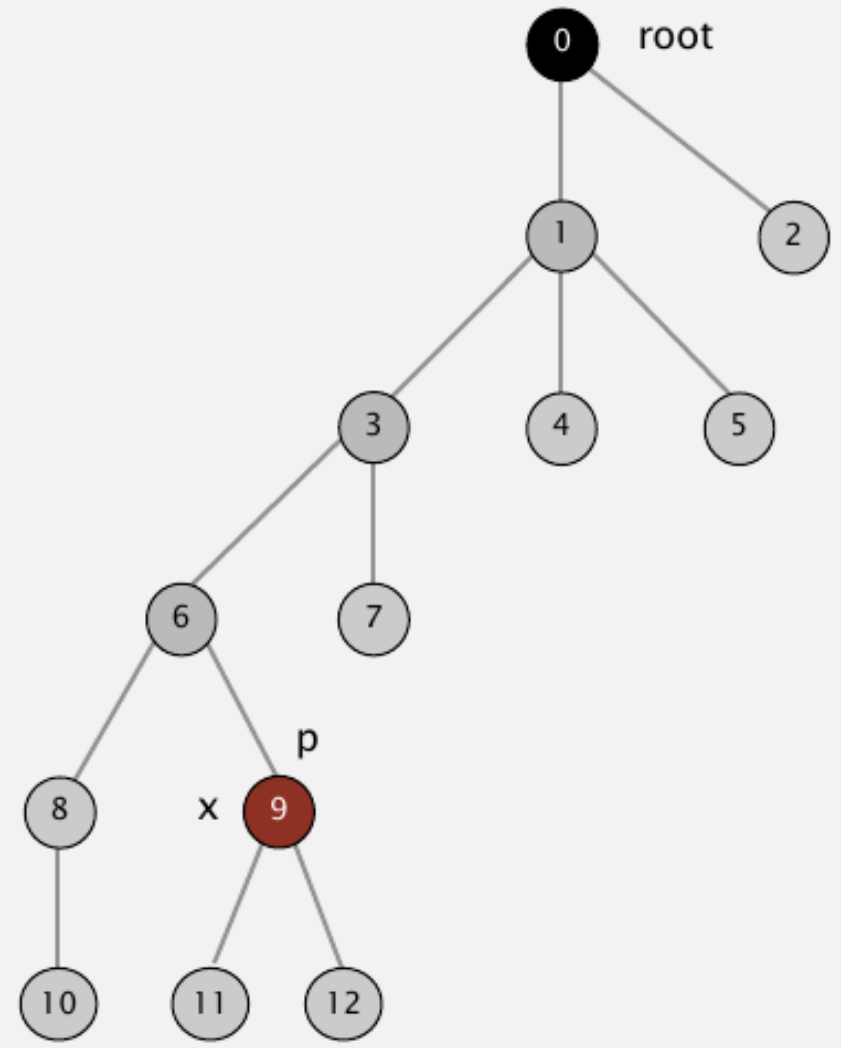
† inclui o custo de encontrar as raízes

Q: Parar na garantia de performance aceitável?

A: Não, fácil de melhorar ainda mais.

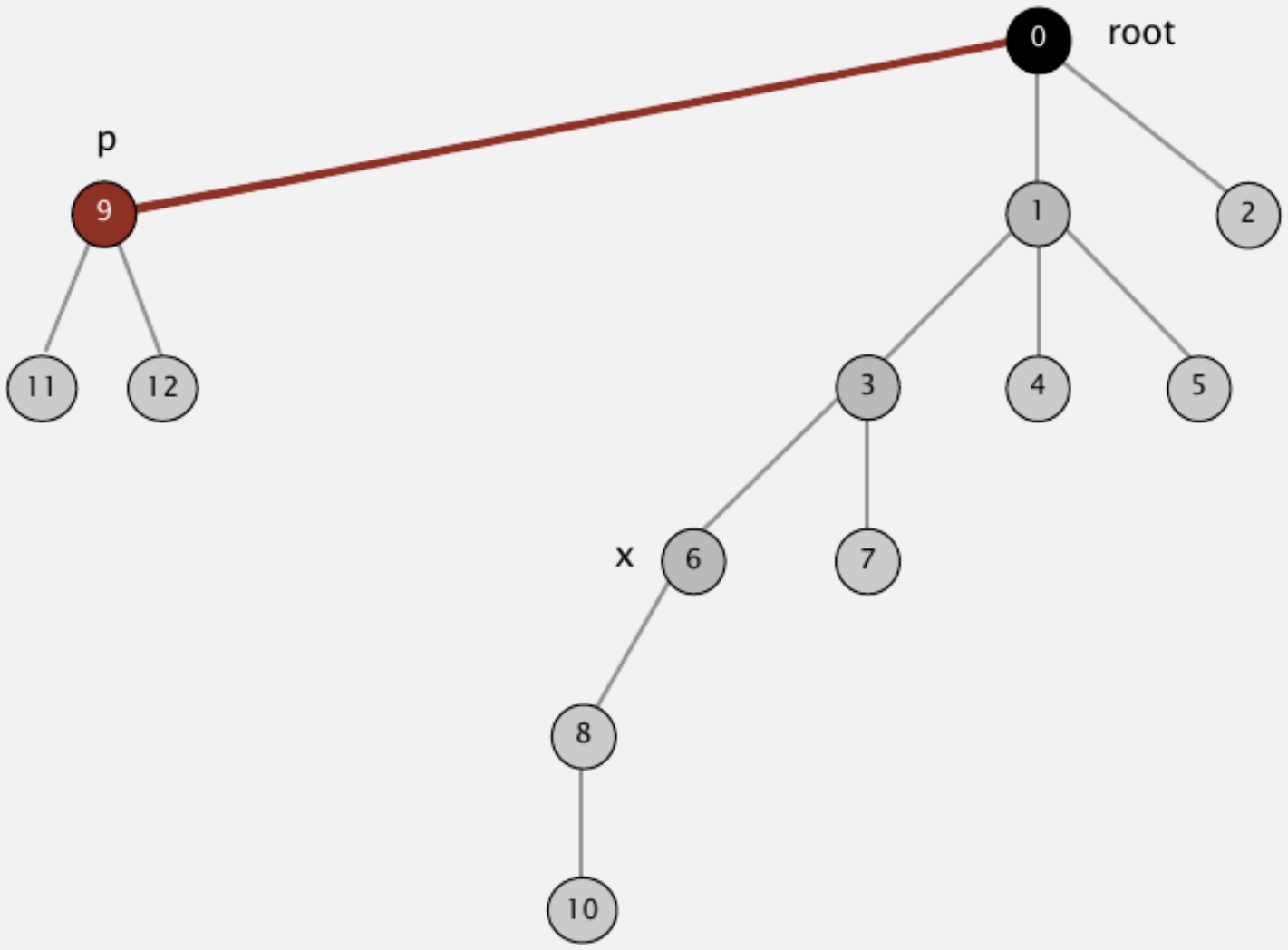
Melhoria 2: Compressão de caminho

Quick-union com compressão de caminho: Após computar a raiz de *p*, atribua `id[]` de cada nó examinado à raiz.



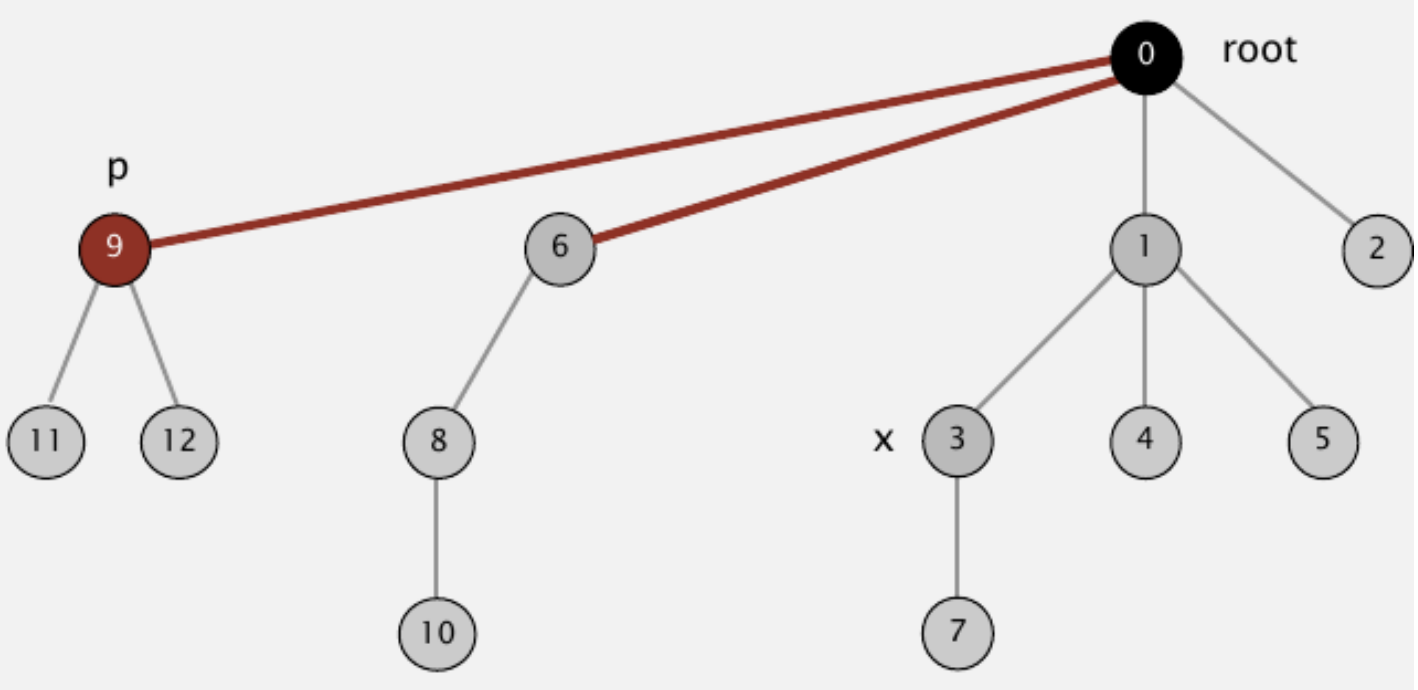
Melhoria 2: Compressão de caminho

Quick-union com compressão de caminho: Após computar a raiz de *p*, atribua `id[]` de cada nó examinado à raiz.



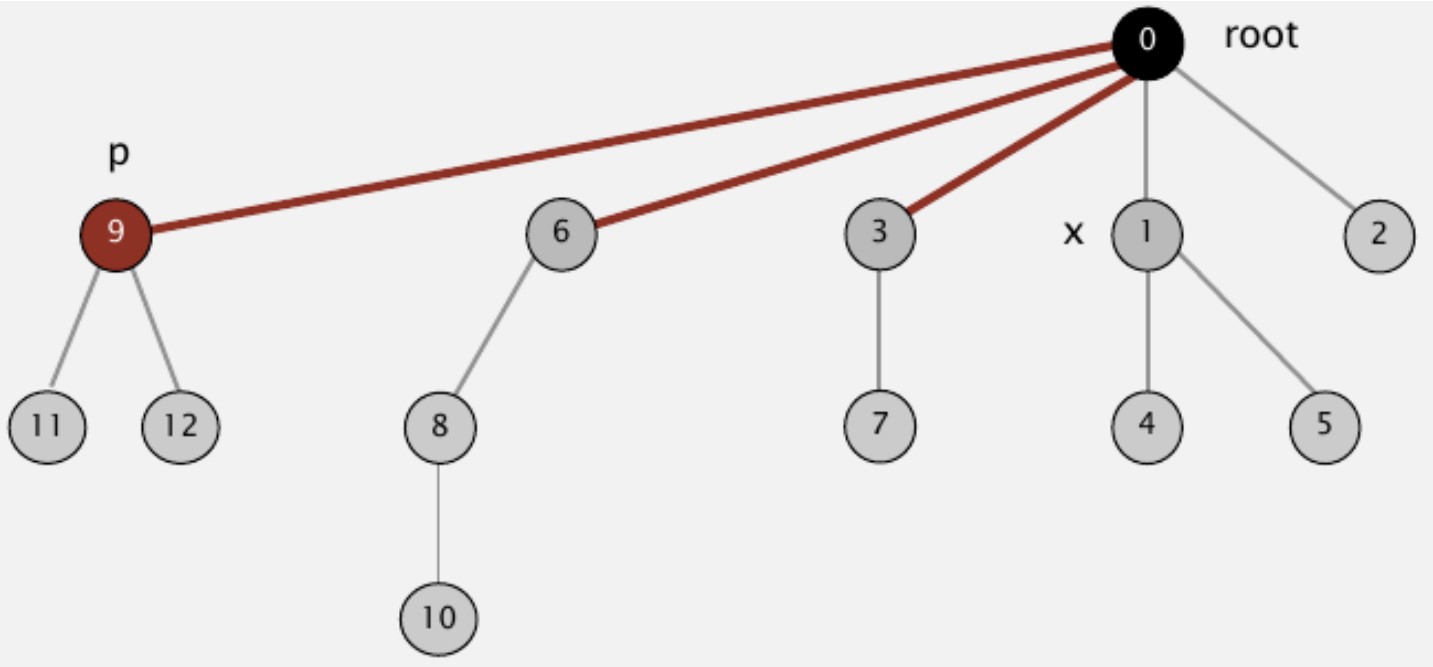
Melhoria 2: Compressão de caminho

Quick-union com compressão de caminho: Após computar a raiz de *p*, atribua `id[]` de cada nó examinado à raiz.



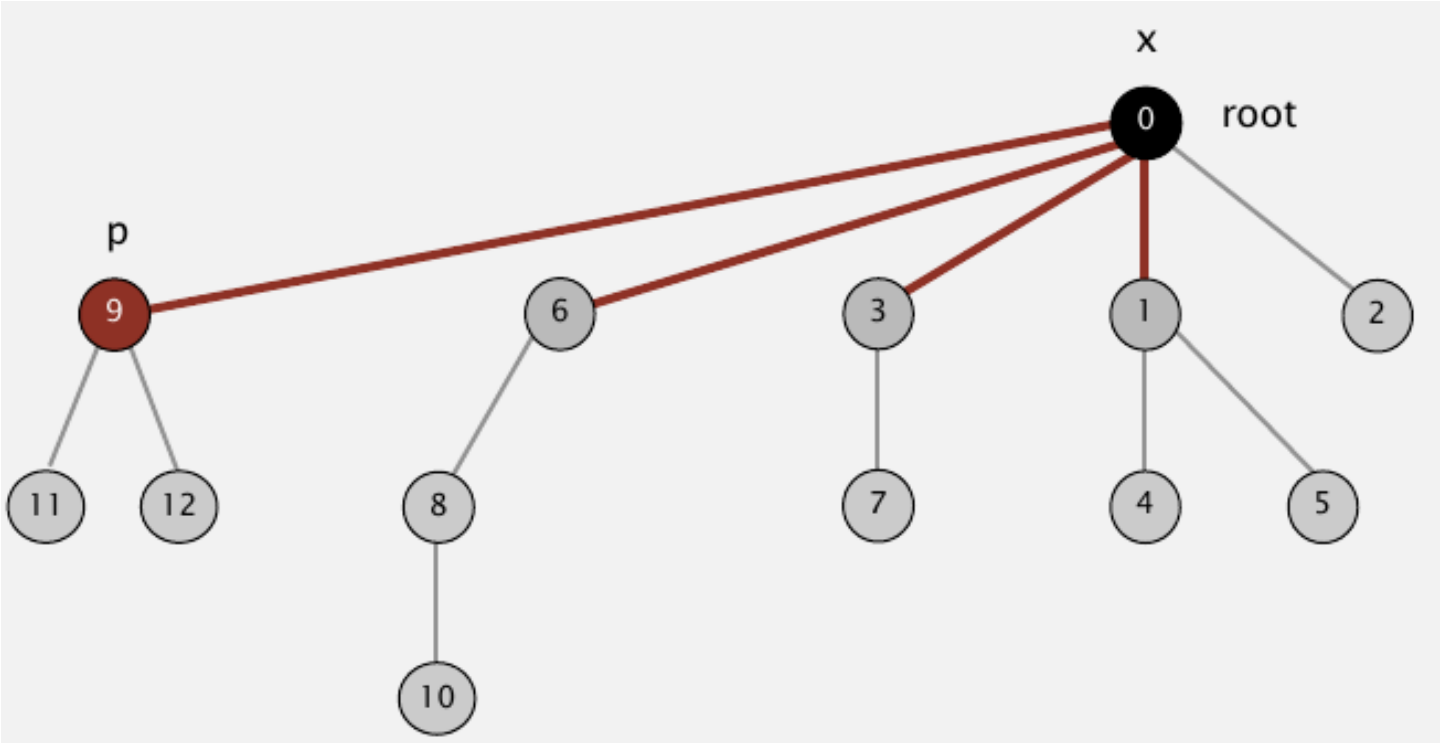
Melhoria 2: Compressão de caminho

Quick-union com compressão de caminho: Após computar a raiz de *p*, atribua `id[]` de cada nó examinado à raiz.



Melhoria 2: Compressão de caminho

Quick-union com compressão de caminho: Após computar a raiz de *p*, atribua `id[]` de cada nó examinado à raiz.



Conclusão: agora `find()` realiza a compressão da árvore como efeito colateral.

Compressão de caminho – Implementação

Em duas passadas: adicionar um segundo *loop* a `find` para atribuir o `id[]` de cada nó visitado para a raiz.

Compressão de caminho – Implementação

Em duas passadas: adicionar um segundo *loop* a `find` para atribuir o `id[]` de cada nó visitado para a raiz.

Variante de passada única (*path halving*): Faz cada nó visitado apontar para o seu avô.

```
int UF_find(int i) {
  while (i != id[i]) {
    id[i] = id[id[i]]; // Uma unica linha de codigo adicional.
    i = id[i];         // Cada passo agora requer 5 acessos.
  }
  return i;
}
```

Compressão de caminho – Implementação

Em duas passadas: adicionar um segundo *loop* a `find` para atribuir o `id[]` de cada nó visitado para a raiz.

Variante de passada única (*path halving*): Faz cada nó visitado apontar para o seu avô.

```
int UF_find(int i) {
  while (i != id[i]) {
    id[i] = id[id[i]]; // Uma unica linha de codigo adicional.
    i = id[i];         // Cada passo agora requer 5 acessos.
  }
  return i;
}
```

Na prática: Uso é justificável. Mantem a árvore praticamente plana.

Weighted QU com compressão – Análise amortizada

Proposição [Hopcroft, Ulman, Tarjan]: Começando com uma estrutura vazia, qualquer sequência de M operações de *union-find* sobre N objetos faz $\leq c(N + M \lg^* N)$ acessos de *array*.

$\lg^*$ é chamada de função lg iterada:

N	1	2	4	16	65536	2^{65536}
$\lg^* N$	0	1	2	3	4	5

Weighted QU com compressão – Análise amortizada

Proposição [Hopcroft, Ulman, Tarjan]: Começando com uma estrutura vazia, qualquer sequência de M operações de *union-find* sobre N objetos faz $\leq c(N + M \lg^* N)$ acessos de *array*.

$\lg^*$ é chamada de função lg iterada:

N	1	2	4	16	65536	2^{65536}
$\lg^* N$	0	1	2	3	4	5

Algoritmo de tempo linear?

- Custo fica a um fator constante da leitura dos dados.
- Em teoria: algoritmo não é linear.
- Na prática: é linear.

Weighted QU com compressão – Análise amortizada

Proposição [Hopcroft, Ulman, Tarjan]: Começando com uma estrutura vazia, qualquer sequência de M operações de *union-find* sobre N objetos faz $\leq c(N + M \lg^* N)$ acessos de *array*.

$\lg^*$ é chamada de função lg iterada:

N	1	2	4	16	65536	2^{65536}
$\lg^* N$	0	1	2	3	4	5

Algoritmo de tempo linear?

- Custo fica a um fator constante da leitura dos dados.
- Em teoria: algoritmo não é linear.
- Na prática: é linear.

Fato interessante: já foi provado que não existe um algoritmo com complexidade teórica linear.

Sumário

Ponto chave: *Weighted quick-union* (e/ou *path compression*) permite a resolução de problemas que não poderiam ser atacados de outra forma.

Algoritmo	Tempo de pior caso
quick-find	MN
quick-union (QU)	MN
weighted QU	$N + M \lg N$
QU + *path compression*	$N + M \lg N$
weighted QU + *path compression*	$N + M \lg^* N$

Sumário

Ponto chave: *Weighted quick-union* (e/ou *path compression*) permite a resolução de problemas que não poderiam ser atacados de outra forma.

Algoritmo	Tempo de pior caso
quick-find	MN
quick-union (QU)	MN
weighted QU	$N + M \lg N$
QU + *path compression*	$N + M \lg N$
weighted QU + *path compression*	$N + M \lg^* N$

Exemplo: 10^9 operações sobre 10^9 objetos.

- Último algoritmo reduz o tempo de 30 anos para 6 segundos.
- Não adianta ter um “super-computador”, é um bom algoritmo que permite a solução.

Problema?

As implementações apresentadas possuem um “problema”.

Problema?

As implementações apresentadas possuem um “problema”.

1. E se eu precisar de um vetor maior que 1000?

Problema?

As implementações apresentadas possuem um “problema”.

1. E se eu precisar de um vetor maior que 1000?
2. E se eu precisar trabalhar com duas estruturas *union-find* no meu programa?

Problema?

As implementações apresentadas possuem um “problema”.

1. E se eu precisar de um vetor maior que 1000?
2. E se eu precisar trabalhar com duas estruturas *union-find* no meu programa?
3. Vamos utilizar alocação dinâmica e encapsulamento.

Problema?

As implementações apresentadas possuem um “problema”.

1. E se eu precisar de um vetor maior que 1000?
2. E se eu precisar trabalhar com duas estruturas *union-find* no meu programa?
3. Vamos utilizar alocação dinâmica e encapsulamento.
4. Ganha-se em código e perde-se em eficiência.