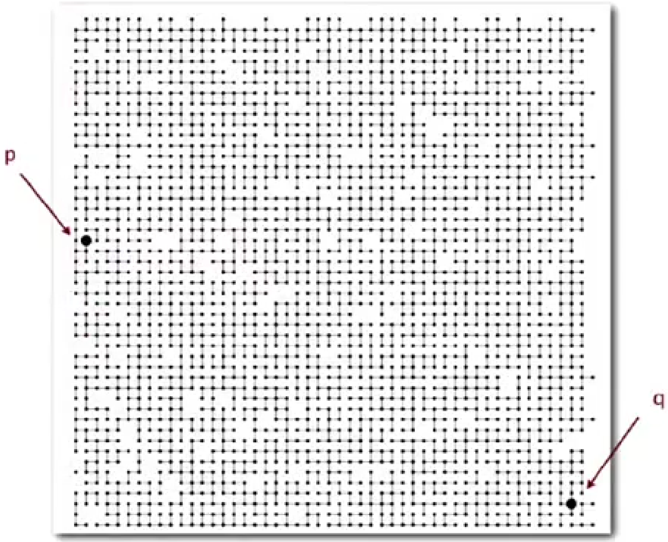
Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 3

Problema: p e q estão ligados?

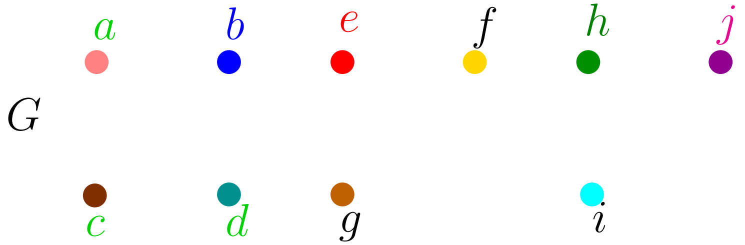


Fonte: algs4

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

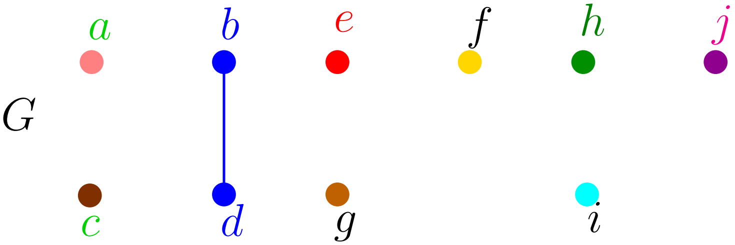


aresta	componentes
	$\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{i\}$ $\{$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

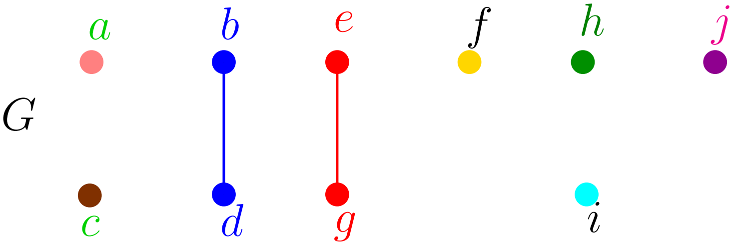


aresta	componentes
(b, d)	$\{a\}$ $\{b, d\}$ $\{c\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

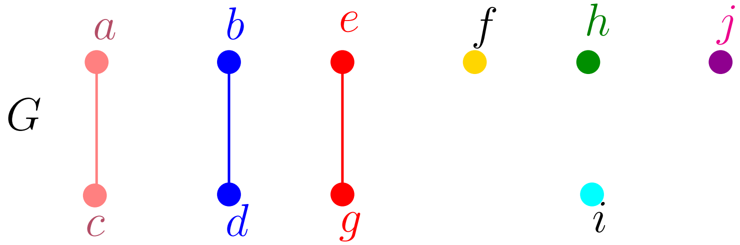


aresta	componentes
(e, g)	$\{a\}$ $\{b, d\}$ $\{c\}$ $\{e, g\}$ $\{f\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

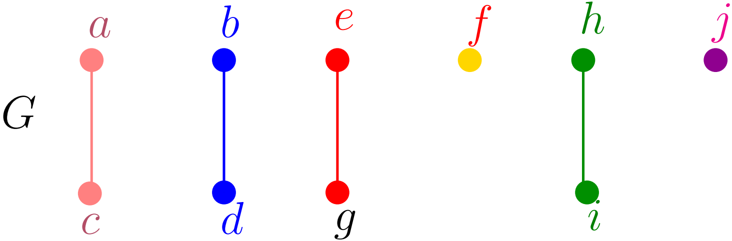


aresta	componentes
(a,c)	$\{a,c\}$ $\{b,d\}$ $\{e,g\}$ $\{f\}$ $\{h\}$ $\{i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

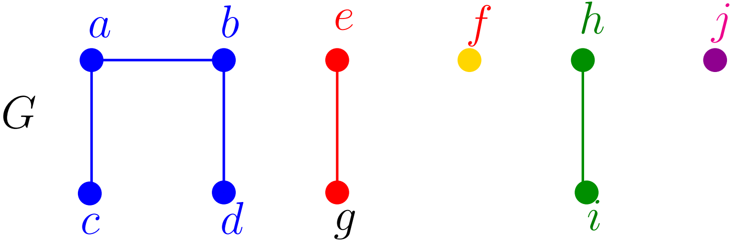


aresta	componentes
(h,i)	$\{a,c\}$ $\{b,d\}$ $\{e,g\}$ $\{f\}$ $\{h,i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

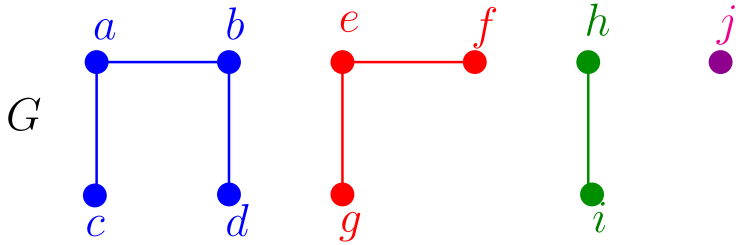


aresta	componentes
(a, b)	$\{a, b, c, d\}$ $\{e, g\}$ $\{f\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico

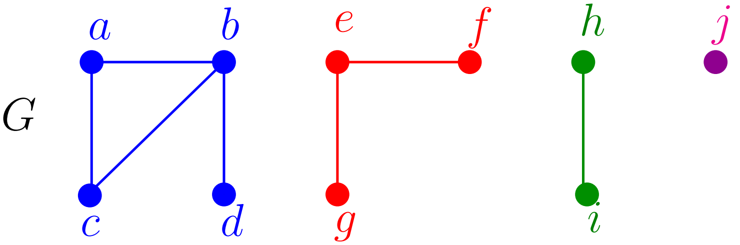


aresta	componentes
(e, f)	$\{a, b, c, d\}$ $\{e, f, g\}$ $\{h, i\}$ $\{j\}$

Coleção disjunta dinâmica

Conjuntos são **modificados ao longo do tempo**

Exemplo: grafo dinâmico



aresta	componentes
(b, c)	$\{a, b, c, d\}$ $\{e, f, g\}$ $\{h, i\}$ $\{j\}$

API

public class	UF	
	`UF(int n)`	inicializa n sites com nomes inteiros `0,...,n-1`
`void`	`union(int p,int q)`	acrescenta ligação entre p e q
`int`	`find(int p)`	retorna id do componente de p
`boolean`	`connected(int p,int q)`	true se p e q estão no mesmo componente
`int`	`count()`	número de componentes

Conjuntos disjuntos dinâmicos

Sequência de operações $\mathtt{UF}(\mathtt{n}) = \mathtt{n} \times \text{MAKESET}$, $\mathtt{union}() = \text{UNION}$, $\mathtt{find}() = \text{FINDSET}$

$$\underbrace{\underbrace{\mathsf{M}\quad \mathsf{M}\quad \mathsf{M}}_{\mathtt{n}}\quad \mathsf{U}\quad \mathsf{F}\quad \mathsf{U}\quad \mathsf{U}\quad \mathsf{F}\quad \mathsf{U}\quad \mathsf{F}\quad \mathsf{F}\quad \mathsf{F}\quad \mathsf{U}\quad \mathsf{F}}_{\mathtt{m}}$$

Que estrutura de dados usar?

Compromissos (*trade-offs*).

QuickFindUF

QuickFindUF uf = new QuickFindUF(10);

```
 uf ----+
        |
        V
+-----------------------------------------------------+
| id (private)                     count: 10 (private) |
|  |                                                   |
|  |     +---+---+---+---+---+---+---+---+---+---+     |
|  +--> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |     |
|        +---+---+---+---+---+---+---+---+---+---+     |
|          0   1   2   3   4   5   6   7   8   9       |
|                                                      |
| Métodos: cont(), connected(), find(), union()        |
|                                                      |
+-----------------------------------------------------+
```

Class QuickFindUF: construtor e count()

```
public QuickFindUF(int n) {
   count = n;
   id = new int[n];
   for (int i = 0; i < n; i++) {
       id[i] = i;
   }
}

// retorna to número de componentes
public int count() {
   return count;
}
```

Class QuickFindUF: connected() e find()

```
// p e q estão no mesmo componente?
public boolean connected(int p, int q) {
   return find(p) == find(q);
}

// retorna o id do componente de p
public int find(int p) {
   return id[p];
}
```

Class QuickFindUF: union()

```
// une os componentes de p e q
public void union(int p, int q) {
   int pID = find(p);
   int qID = find(q);
   if (pID == qID) return ;
   for (int i = 0; i < id.length; i++) {
      if (id[i] == pID) id[i] = qID;
   }
   count--;
}
```

Consumo de tempo

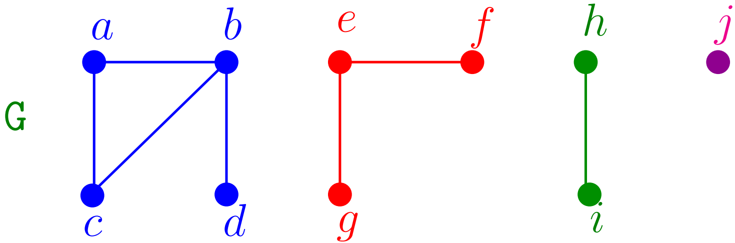
UF(n)	$\Theta(n)$
union(p, q)	$O(n)$
find(p)	$\Theta(1)$

Uma sequência de m operações pode consumir tempo $\Theta(m^2)$ no pior caso.

Consumo de tempo amortizado de cada operação é $O(m)$.

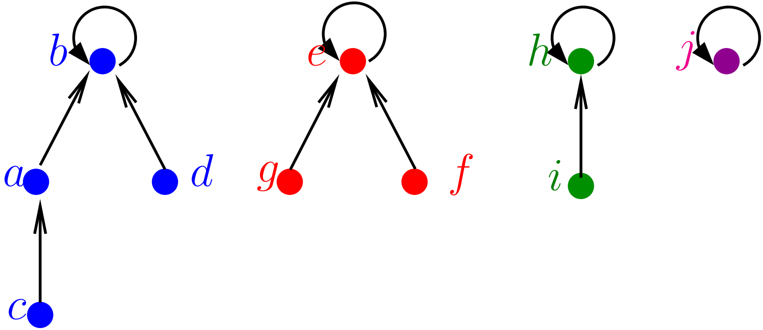
Hmm. Em union() seria razoável alterarmos o menor número possível de posições do vetor id. Para isso precisamos saber qual conjunto tem o menor número de itens. . .

QuickUnionUF



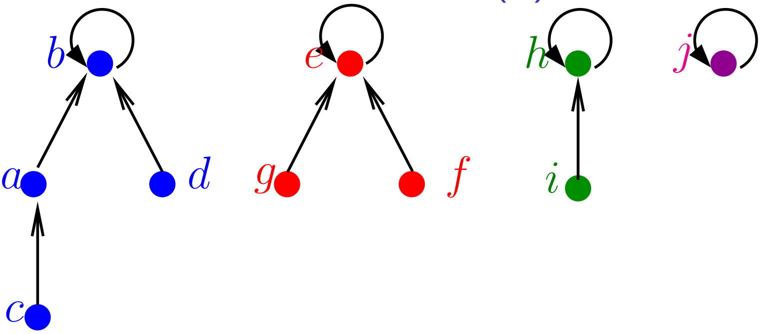
- cada conjunto tem uma raiz, que é o seu representate
- cada nó x tem um pai
- pai[x] = x se e só se x é uma raiz

Estrutura *disjoint-set forest*

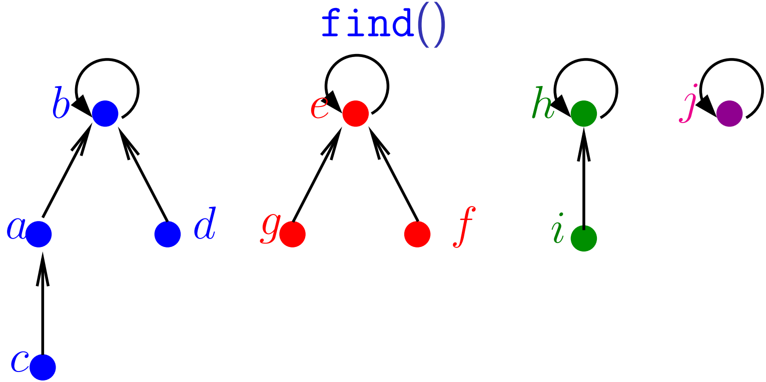


- cada conjunto tem uma `raiz`
- cada nó `x` tem um pai
- $\texttt{pai}[\texttt{x}] = \texttt{x}$ se e só se `x` é uma raiz

QuickUnionUF(n)

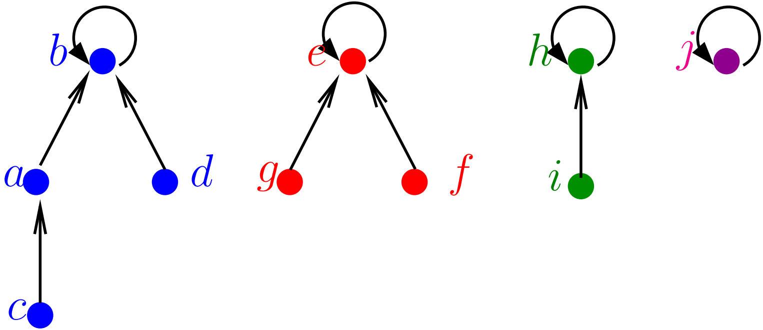



```
public QuickUnionUF(int n) {
    count = n;
    pai = new int[n];
    for (int i = 0; i < n; i++) {
        pai[i] = i;
    }
}
```



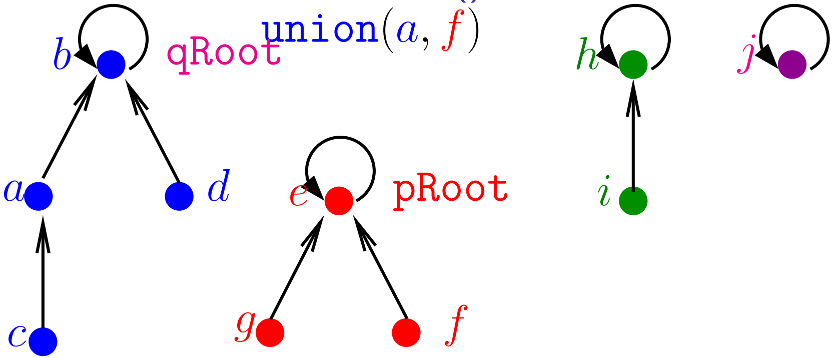

```
// retorna o id do componente de p
public int find(int p) {
   while (p != pai[p]) p = pai[p];
   return p;
}
```

union()




```
public void union(int p, int q) {
   int pRoot = find(p);
   int qRoot = find(q);
   if (pRoot == qRoot) return ;
   pai[pRoot] = qRoot;
   count--; }
```

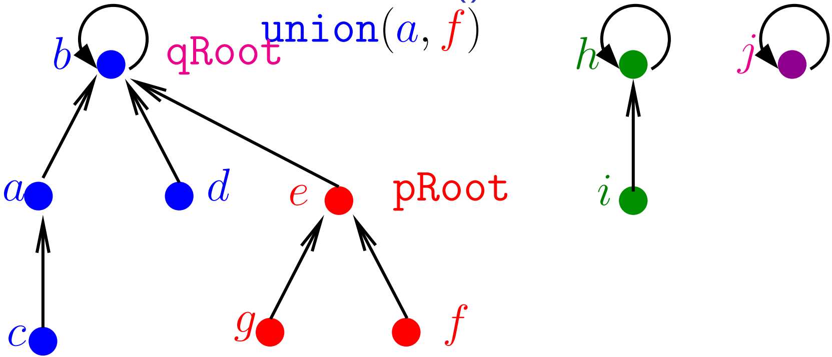
union()




```
public void union(int p, int q) {
   int pRoot = find(p);
   int qRoot = find(q);
   if (pRoot == qRoot) return ;
   pai[pRoot] = qRoot;
   count--; }
```

union()

union(a, f)




```
public void union(int p, int q) {
   int pRoot = find(p);
   int qRoot = find(q);
   if (pRoot == qRoot) return ;
   pai[pRoot] = qRoot;
   count--; }
```

QuickUnionUF

```
uf ----+
       |
       V
+----------------------------------------------------+
| pai (private)                   count:  5 (private)|
|  |                                                 |
|  |     +---+---+---+---+---+---+---+---+---+---+   |
|  +-->  | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |   |
|        +---+---+---+---+---+---+---+---+---+---+   |
|          0   1   2   3   4   5   6   7   8   9     |
|                                                    |
| Métodos: cont(), connected(), find(), union()      |
|                                                    |
+----------------------------------------------------+
```

Consumo de tempo

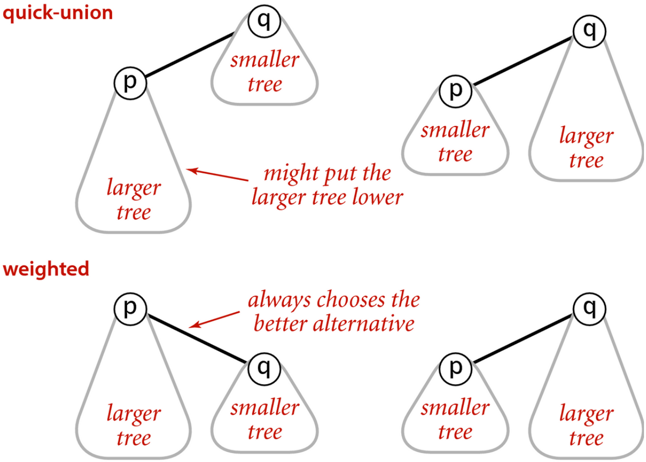
`UF(n)`	$\Theta(n)$
`find(p)`	$O(n)$
`union(p, q)`	$O(n)$

$$\underbrace{\underbrace{\mathsf{M}\ \mathsf{M}\ \mathsf{M}}_{\mathtt{n}}\ \mathsf{U}\ \mathsf{F}\ \mathsf{U}\ \mathsf{U}\ \mathsf{F}\ \mathsf{U}\ \mathsf{F}\ \mathsf{F}\ \mathsf{F}\ \mathsf{U}\ \mathsf{F}}_{\mathtt{m}}$$

Custo total da sequência:

$$\mathtt{n}\,\Theta(1) + \mathtt{m}\,O(\mathtt{n}) + \mathtt{n}\,O(\mathtt{n}) = O(\mathtt{mn})$$

WeightedQuickUnionUF



Weighted quick-union

Fonte: algs4

union by size

```
public void union(int p, int q) {
   int pRoot = find(p);
   int qRoot = find(q);
   if (pRoot == qRoot) return ;
   if (sz[pRoot] < sz[qRoot]) {
       pai[pRoot] = qRoot;
       sz[qRoot] += sz[pRoot];
   }
   else {
       pai[qRoot] = pRoot;
       sz[pRoot] += sz[qRoot];
   }
   count--;
}
```

WeightedQuickUnionUF

```
uf
+-----------------------------------------------------+
| pai (private)                      count:  5 (private)|
|  |                                                  |
|  |     +---+---+---+---+---+---+---+---+---+---+    |
|  +--> | 0 | 1 | 1 | 4 | 4 | 6 | 6 | 7 | 4 | 4 |    |
|        +---+---+---+---+---+---+---+---+---+---+    |
|          0   1   2   3   4   5   6   7   8   9      |
|                                                     |
| sz (private)                                        |
|  |                                                  |
|  |     +---+---+---+---+---+---+---+---+---+---+    |
|  +--> | 1 | 2 | 1 | 1 | 3 | 1 | 2 | 1 | 1 | 1 |    |
|        +---+---+---+---+---+---+---+---+---+---+    |
|          0   1   2   3   4   5   6   7   8   9      |
|                                                     |
| Métodos: cont(), connected(), find(), union()       |
+-----------------------------------------------------+
```

Consumo de tempo

$\mathtt{UF}(\mathtt{n})$	$\Theta(\mathtt{n})$
$\mathtt{find}(\mathtt{p})$	$\mathrm{O}(\lg \mathtt{n})$
$\mathtt{union}(\mathtt{p}, \mathtt{q})$	$\mathrm{O}(\lg \mathtt{n})$

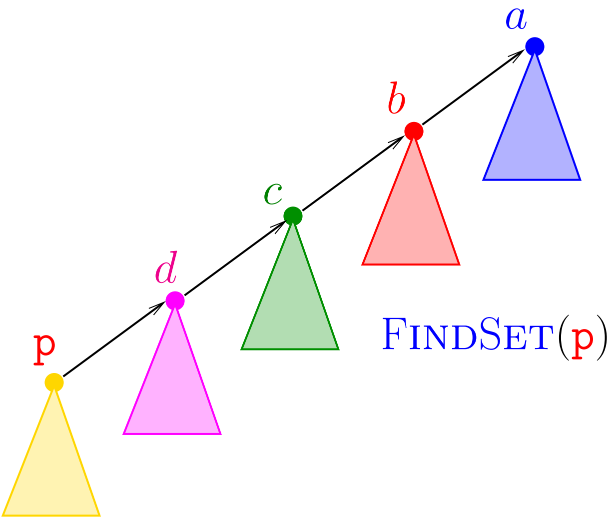
$$\underbrace{\underbrace{\mathsf{M} \quad \mathsf{M} \quad \mathsf{M}}_{\mathtt{n}} \quad \mathsf{U} \quad \mathsf{F} \quad \mathsf{U} \quad \mathsf{U} \quad \mathsf{F} \quad \mathsf{U} \quad \mathsf{F} \quad \mathsf{F} \quad \mathsf{F} \quad \mathsf{U} \quad \mathsf{F}}_{\mathtt{m}}$$

Custo total da sequência:

$$\Theta(\mathtt{n}) + \mathtt{m}\,\mathrm{O}(\lg \mathtt{n}) + \mathtt{n}\,\mathrm{O}(\lg \mathtt{n}) = \mathrm{O}(\mathtt{m} \lg \mathtt{n})$$

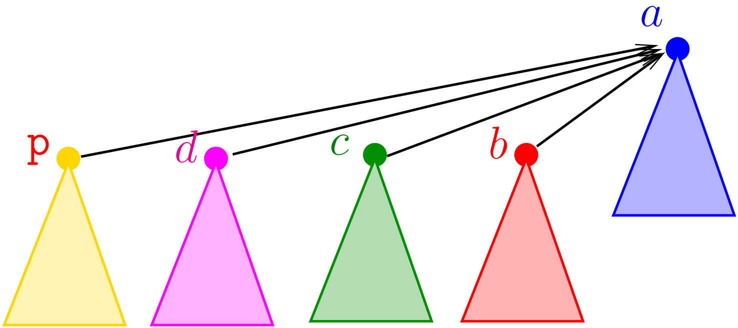
PathCompressionUF

Ideia: encurtar os caminhos durante cada find().



PathCompressionUF

Ideia: encurtar os caminhos durante cada find().



FINDSET(p)

path compression

```
public int find(int p) {
   if (p != pai[p])
        pai[p] = find(pai[p]);
   return pai[p];
}
```

Função log-estrela

$\lg^* \mathtt{n}$ é o menor $\mathtt{k}$ tal que

$$\underbrace{\lg \lg \ldots \lg}_{k} \, n \leq 1$$

$\mathtt{n}$	$\lg^* \mathtt{n}$
1	0
2	1
3	2
4	2
5	3
⋮	⋮
15	3
16	3
⋮	⋮
65535	4
65536	4
⋮	⋮
$\underbrace{100000000000000000 \cdots 000000000000}_{80}$	5
⋮	⋮

Função 'torre'

$$t(i) := \begin{cases} 1 & \text{se } i = 0 \\ 2^{t(i-1)} & \text{se } i = 1, 2, 3, \ldots \end{cases}$$

i	$t(i)$
0	1
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^{2^2} = 16$
4	$2^{2^{2^2}} = 2^{16} = 65536$
5	$2^{2^{2^{2^2}}} > \underbrace{10000000000000000000 \cdots 000000000000}_{80}$
$\vdots$	$\vdots$

Consumo de tempo

UF(n)	$\Theta(n)$	
find(p)	$O(\lg^* n)$	**amortizado!**
union(p, q)	$O(\lg^* n)$	**amortizado!**

$$\underbrace{\underbrace{\text{M M M}}_{n}\ \text{U F U U F U F F F U F}}_{m}$$

Custo total da sequência:

$$\Theta(n) + m\,O(\lg^* n) + n\,O(\lg^* n) = O(m \lg^* n)$$

Conclusões

Se conjuntos disjuntos são representados através de disjoint-set forest com *union by rank* e *path compression*, então uma sequência de `UF(n)` e `m` operações `union()` e `find()`, consome tempo $O(m \lg^* n)$.

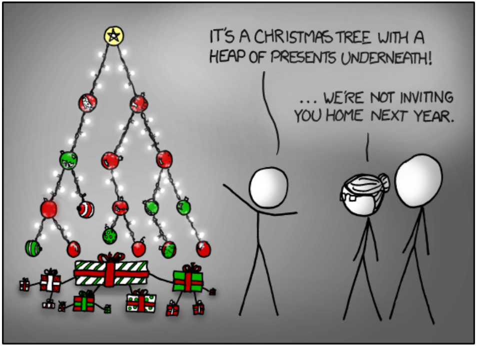
Resumo

	UF()	find()	union()
QuickFindUF	$\Theta(n)$	$\Theta(1)$	$O(n)$
QuickUnionUF	$\Theta(n)$	$O(n)$	$O(n)$
WeightedQuickUnionUF	$\Theta(n)$	$O(\lg n)$	$O(\lg n)$
PathCompressionUF	$\Theta(n)$	$O(\lg^* n)$	$O(\lg^* n)$

Fonte: Pinterest

AULA 4

Árvores em vetores e heaps

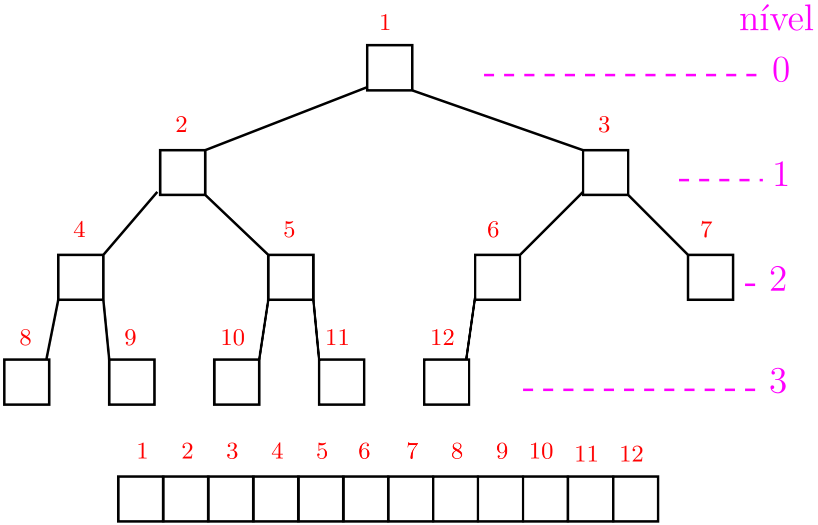


Fonte: http://xkcd.com/835/

PF 10

http://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html

Representação de árvores em vetores



Pais e filhos

$a[1..m]$ é um vetor representando uma árvore.

Diremos que para qualquer **índice** ou **nó** i,

- $\lfloor i/2 \rfloor$ é o **pai** de i;
- $2\,i$ é o **filho esquerdo** de i;
- $2\,i{+}1$ é o **filho direito**.

Um nó i só tem filho esquerdo se $2\,i \leq m$.

Um nó i só tem filho direito se $2\,i{+}1 \leq m$.

Raiz e folhas

O nó 1 não tem pai e é chamado de **raiz**.

Um nó i é um **folha** se não tem **filhos**, ou seja $2\,i > \mathtt{m}$.

Todo nó i é raiz da subárvore formada por

$$\mathtt{a}[i, 2i, 2i{+}1, 4i, 4i{+}1, 4i{+}2, 4i{+}3, 8i, \ldots, 8i{+}7, \ldots]$$

Níveis

Cada nível $\mathtt{p}$, exceto talvez o último, tem exatamente $2^{\mathtt{p}}$ nós e esses são

$$2^{\mathtt{p}}, 2^{\mathtt{p}} + 1, 2^{\mathtt{p}} + 2, \ldots, 2^{\mathtt{p}+1} - 1.$$

Níveis

Cada nível $\mathtt{p}$, exceto talvez o último, tem exatamente $2^{\mathtt{p}}$ nós e esses são

$$2^{\mathtt{p}}, 2^{\mathtt{p}} + 1, 2^{\mathtt{p}} + 2, \ldots, 2^{\mathtt{p}+1} - 1.$$

O nó i pertence ao nível ???.

Níveis

Cada nível $\mathtt{p}$, exceto talvez o último, tem exatamente $2^{\mathtt{p}}$ nós e esses são

$$2^{\mathtt{p}}, 2^{\mathtt{p}}+1, 2^{\mathtt{p}}+2, \ldots, 2^{\mathtt{p}+1}-1.$$

O nó i pertence ao nível $\lfloor \lg i \rfloor$.

Níveis

Cada nível $\mathtt{p}$, exceto talvez o último, tem exatamente $2^{\mathtt{p}}$ nós e esses são

$$2^{\mathtt{p}}, 2^{\mathtt{p}}+1, 2^{\mathtt{p}}+2, \ldots, 2^{\mathtt{p}+1}-1.$$

O nó i pertence ao nível $\lfloor \lg i \rfloor$.

Prova: Se $\mathtt{p}$ é o nível do nó i, então

$$\begin{array}{rcccll} 2^{\mathtt{p}} & \leq & i & < & 2^{\mathtt{p}+1} & \Rightarrow \\ \lg 2^{\mathtt{p}} & \leq & \lg i & < & \lg 2^{\mathtt{p}+1} & \Rightarrow \\ \mathtt{p} & \leq & \lg i & < & \mathtt{p}+1 & \end{array}$$

Logo, $\mathtt{p} = \lfloor \lg i \rfloor$.

Níveis

Cada nível p, exceto talvez o último, tem exatamente 2^p nós e esses são

$$2^p, 2^p + 1, 2^p + 2, \ldots, 2^{p+1} - 1.$$

O nó i pertence ao nível $\lfloor \lg i \rfloor$.

Prova: Se p é o nível do nó i, então

$$\begin{array}{rcccll} 2^p & \leq & i & < & 2^{p+1} & \Rightarrow \\ \lg 2^p & \leq & \lg i & < & \lg 2^{p+1} & \Rightarrow \\ p & \leq & \lg i & < & p+1 & \end{array}$$

Logo, $p = \lfloor \lg i \rfloor$.

Portanto, o número total de níveis é ???.

Níveis

Cada nível $\mathtt{p}$, exceto talvez o último, tem exatamente $2^{\mathtt{p}}$ nós e esses são

$$2^{\mathtt{p}}, 2^{\mathtt{p}}+1, 2^{\mathtt{p}}+2, \ldots, 2^{\mathtt{p}+1}-1.$$

O nó i pertence ao nível $\lfloor \lg i \rfloor$.

Prova: Se $\mathtt{p}$ é o nível do nó i, então

$$\begin{array}{rcccll} 2^{\mathtt{p}} & \leq & i & < & 2^{\mathtt{p}+1} & \Rightarrow \\ \lg 2^{\mathtt{p}} & \leq & \lg i & < & \lg 2^{\mathtt{p}+1} & \Rightarrow \\ \mathtt{p} & \leq & \lg i & < & \mathtt{p}+1 & \end{array}$$

Logo, $\mathtt{p} = \lfloor \lg i \rfloor$.

Portanto, o número total de níveis é $1 + \lfloor \lg \mathtt{m} \rfloor$.

Altura

A **altura** de um nó i é o maior comprimento de um caminho de i a uma folha.

Em outras palavras, a altura de um nó i é o maior comprimento de uma seqüência da forma

$$\langle \texttt{filho}(i), \texttt{filho}(\texttt{filho}(i)), \texttt{filho}(\texttt{filho}(\texttt{filho}(i))), \ldots$$

onde $\texttt{filho}(i)$ vale $2i$ ou $2i + 1$.

Os nós que têm **altura zero** são as folhas.

Altura

A **altura** de um nó i é o maior comprimento de um caminho de i a uma folha.

Em outras palavras, a altura de um nó i é o maior comprimento de uma seqüência da forma

$$\langle \texttt{filho}(i), \texttt{filho}(\texttt{filho}(i)), \texttt{filho}(\texttt{filho}(\texttt{filho}(i))), \ldots$$

onde $\texttt{filho}(i)$ vale $2i$ ou $2i+1$.

Os nós que têm **altura zero** são as folhas.

A altura de um nó i é $\lfloor \lg(\mathtt{m}/i) \rfloor$ $(\ldots)$.

Resumão

filho esquerdo de i:	$2\,i$
filho direito de i:	$2\,i + 1$
pai de i:	$\lfloor i/2 \rfloor$
nível da raiz:	0
nível de i:	$\lfloor \lg i \rfloor$
altura da raiz:	$\lfloor \lg \mathtt{m} \rfloor$
altura da árvore:	$\lfloor \lg \mathtt{m} \rfloor$
altura de i:	$\lfloor \lg(\mathtt{m}/i) \rfloor$ (. . .)
altura de uma folha:	0
total de nós de altura $\mathtt{h}$	$\leq \lceil \mathtt{m}/2^{\mathtt{h}+1} \rceil$ (. . .)

Heaps

Um vetor $\mathtt{a}[1\,..\,\mathtt{m}]$ é um **`max-heap`** se

$$\mathtt{a}[i/2] \geq \mathtt{a}[i]$$

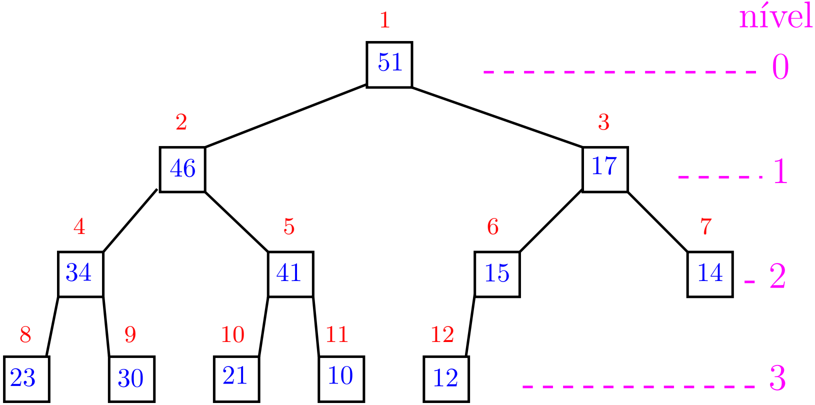
para todo $i = 2, 3, \ldots, \mathtt{m}$.

De uma forma mais geral, $\mathtt{a}[\mathtt{j}\,..\,\mathtt{m}]$ é um **`max-heap`** se

$$\mathtt{a}[i/2] \geq \mathtt{a}[i]$$

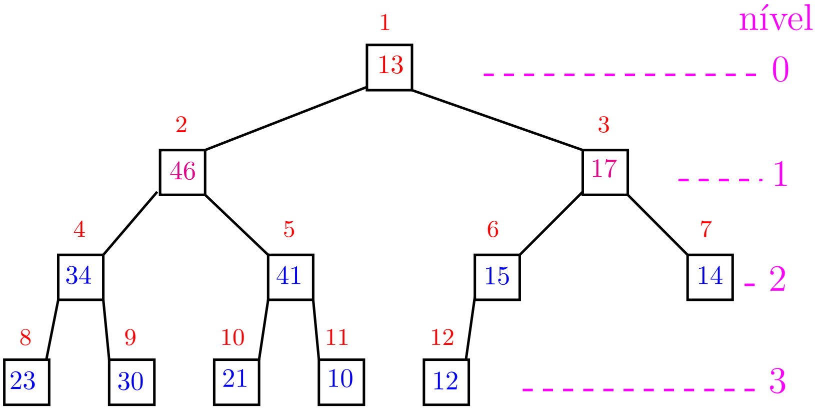
para todo $i = 2\mathtt{j}, 2\mathtt{j}+1, 4\mathtt{j}, \ldots, 4\mathtt{j}+3, 8\mathtt{j}, \ldots, 8\mathtt{j}+7, \ldots$.
Neste caso também diremos que a subárvore com raiz $\mathtt{j}$ é um **`max-heap`**.

max-heap



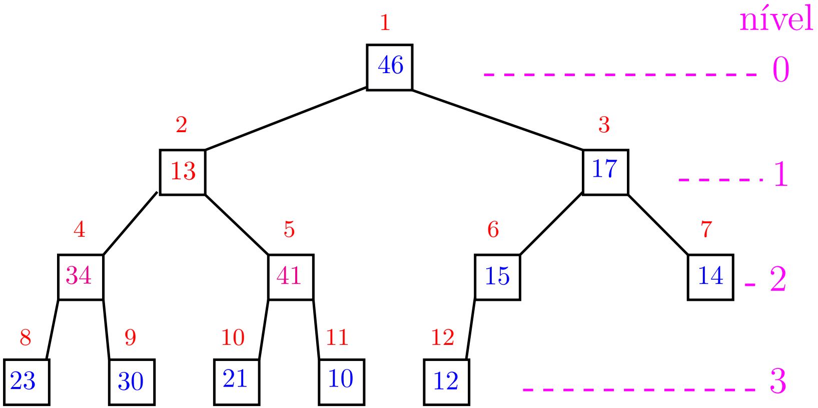
1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Função básica de manipulação de `max-heap`



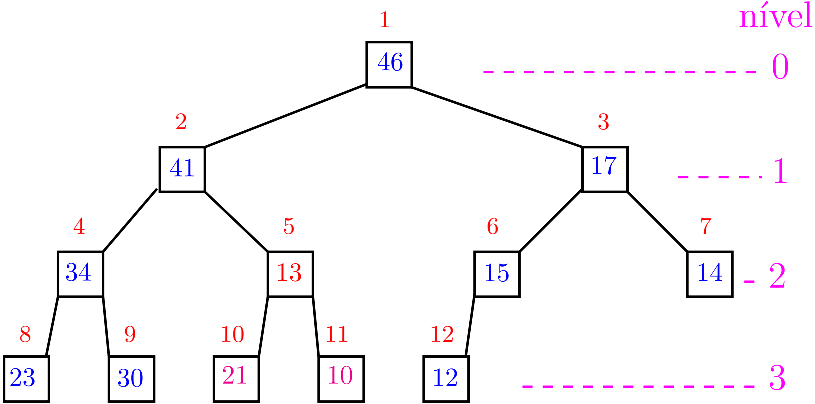
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Função básica de manipulação de max-heap



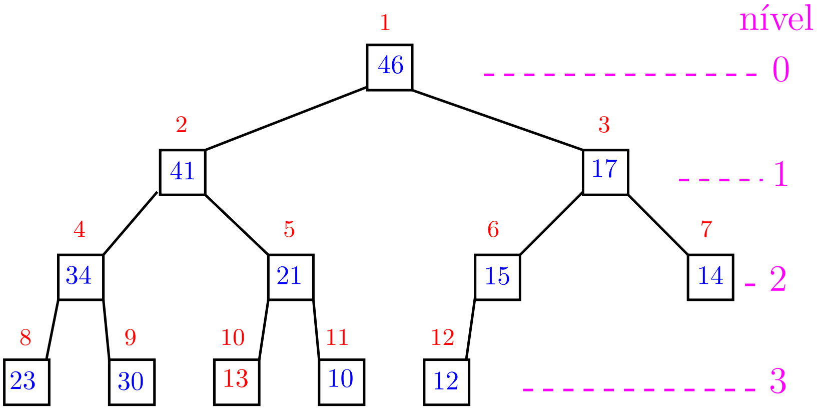
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Função básica de manipulação de max-heap



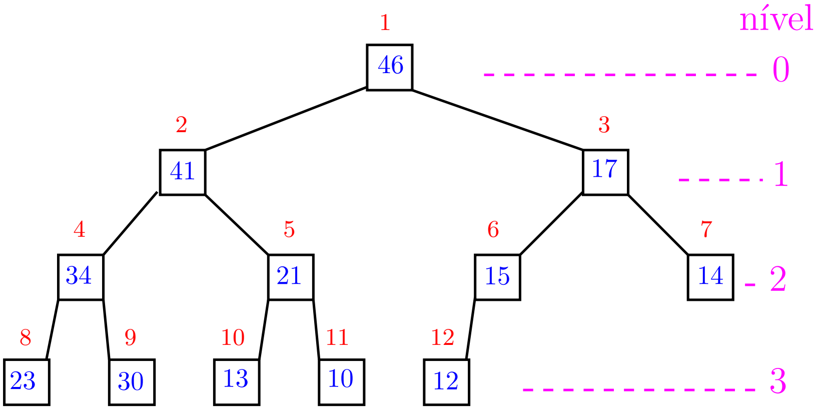
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

Função básica de manipulação de max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	21	15	14	23	30	13	10	12

Função básica de manipulação de max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	21	15	14	23	30	13	10	12

Função `sink`

O coração de qualquer algoritmo que manipule um `max-heap` é uma função que recebe um vetor arbitrário $\mathtt{a}[1 \mathinner{..} \mathtt{m}]$ e um índice `p` e faz $\mathtt{a}[\mathtt{p}]$ "descer" para sua posição correta.

Função sink

Rearranja o vetor a[1 . . m] de modo que o “subvetor” cuja raiz é p seja um max-heap.

```
private static
void sink (int p, int m, Comparable[] a){
   int f = 2*p; Object x;
   while (f <= m) {
       if (f<m && less(a[f],a[f+1])) f++;
       if (!less(a[p], a[f])) break;
       x = a[p]; a[p] = a[f]; a[f] = x;
       p = f; f = 2*p;
   }
}
```

Função sink

Supõe que os "subvetores"cujas raízes são filhos de p já são max-heap.

```
private static
void sink (int p, int m, Comparable[] a){
   int f = 2*p; Object x;
   while (f <= m) {
      if (f<m && less(a[f],a[f+1]) f++;
      if (!less(a[p], a[f]) break;
      x = a[p]; a[p] = a[f]; a[f] = x;
      p = f; f = 2*p;
   }
}
```

Função sink

Implementação um pouco melhor pois em vez de trocas faz apenas deslocamentos (linha 5).

```
private static
void sink (int p, int m, Comparable[] a){
1   int f = 2*p; Object x = a[p];
2   while (f <= m) {
3       if (f<m && less(a[f],a[f+1]) f++;
4       if (!less(x, a[f])) break;
5       a[p] = a[f];
6       p = f; f = 2*p;
    }
7   a[p] = x;
 }
```

Consumo de tempo

linha	todas as execuções da linha	
1	$=$	1
2	$\leq$	$1 + \lg \mathtt{m}$
3	$\leq$	$\lg \mathtt{m}$
4	$\leq$	$\lg \mathtt{m}$
5	$\leq$	$\lg \mathtt{m}$
6	$\leq$	$\lg \mathtt{m}$
7	$=$	1
total	$\leq$	$3 + 5 \lg \mathtt{m} = \mathrm{O}(\lg \mathtt{m})$

Conclusão

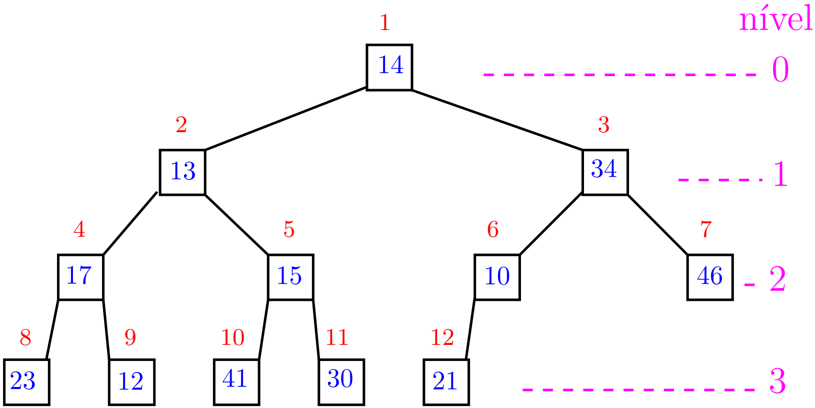
O consumo de tempo da função `sink` é proporcional a $\lg \mathtt{m}$.

O consumo de tempo da função `sink` é $O(\lg \mathtt{m})$.

Verdade seja dita ... (...)

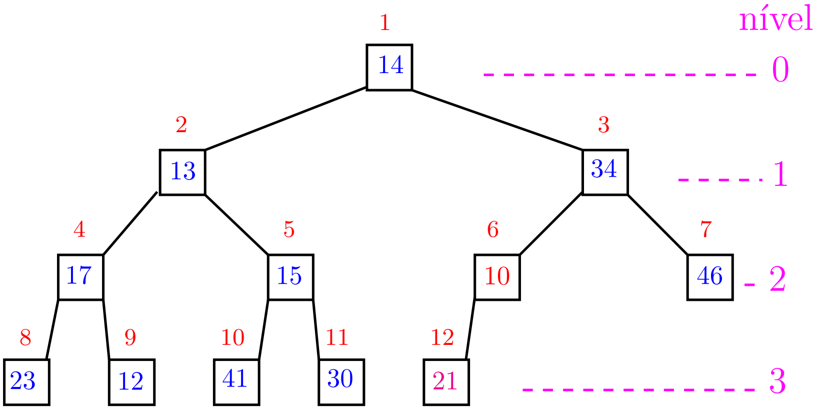
O consumo de tempo da função `sink` é proporcional a $O(\lg \mathtt{m}/\mathtt{p})$.

Construção de um max-heap



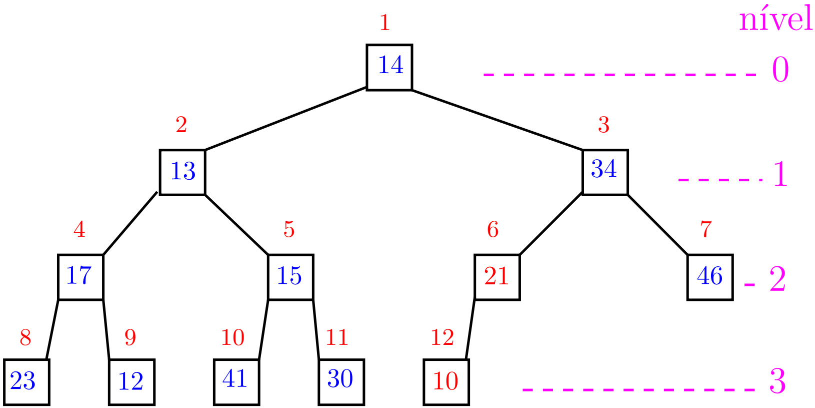
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



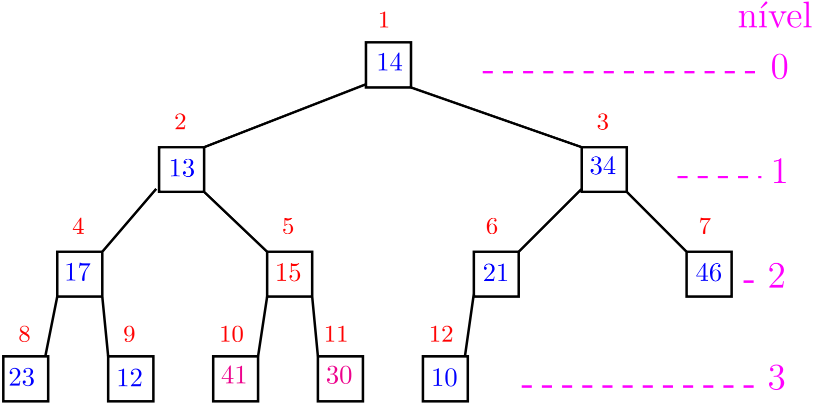
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



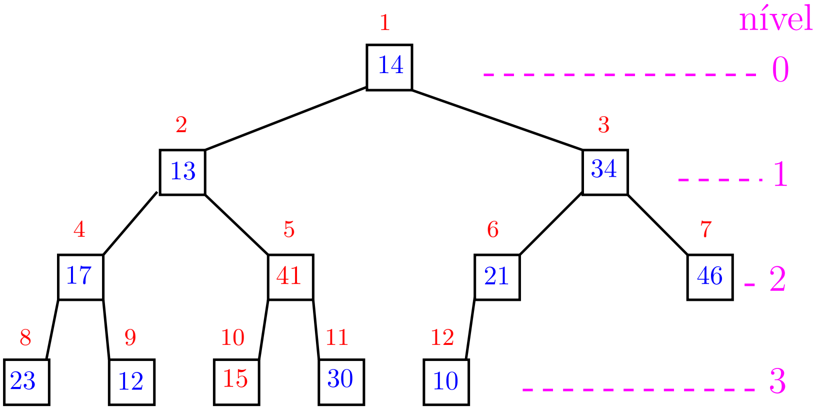
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



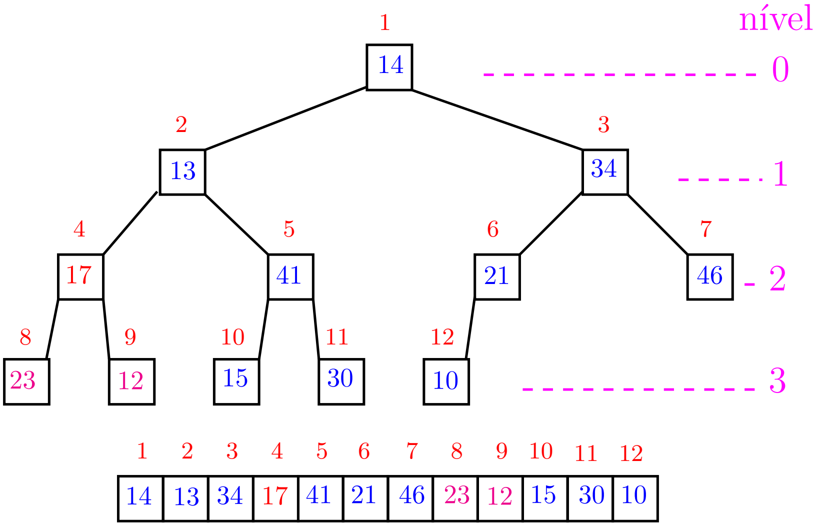
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap

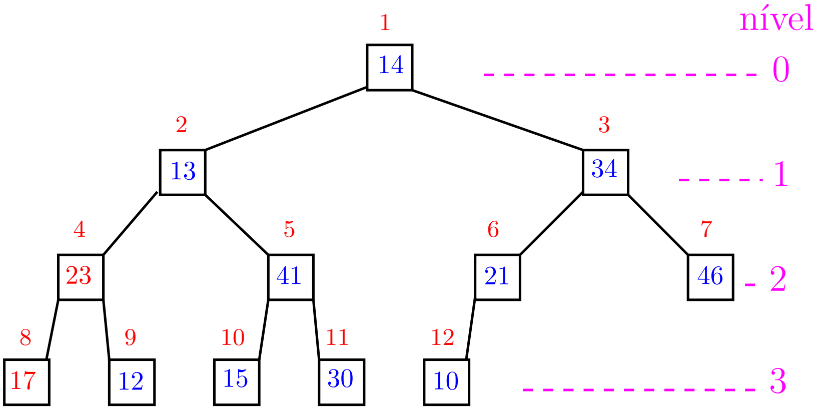


1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap

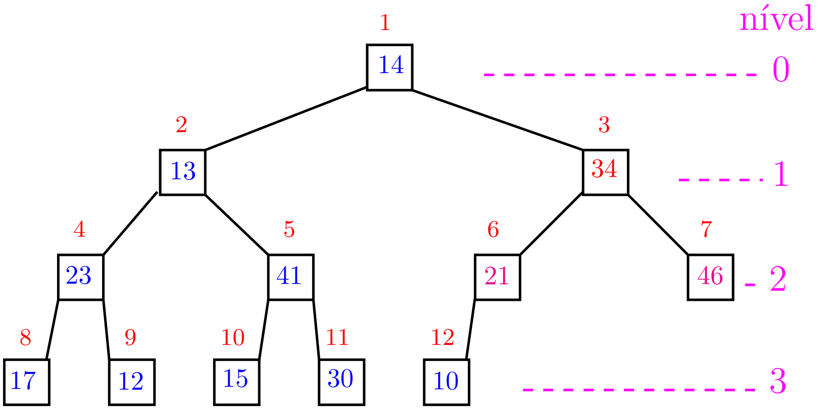


Construção de um `max-heap`



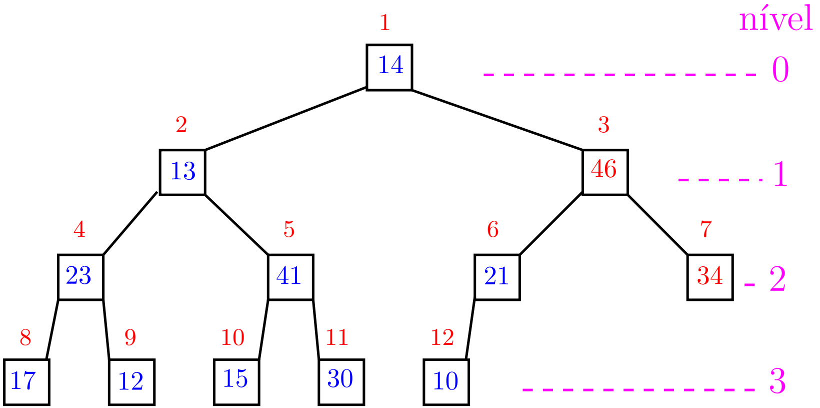
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



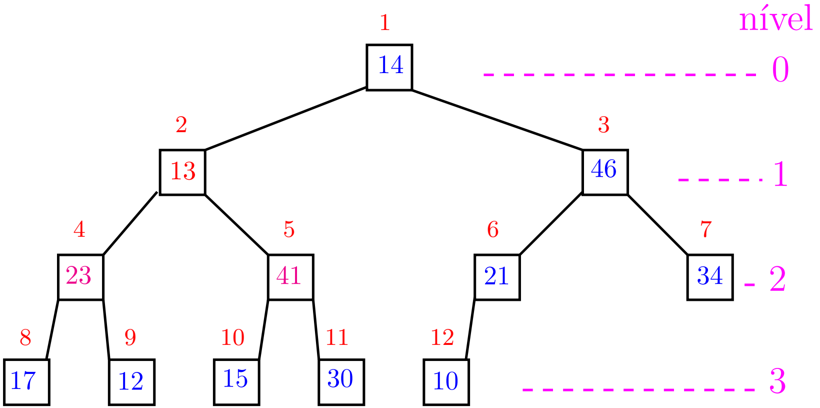
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



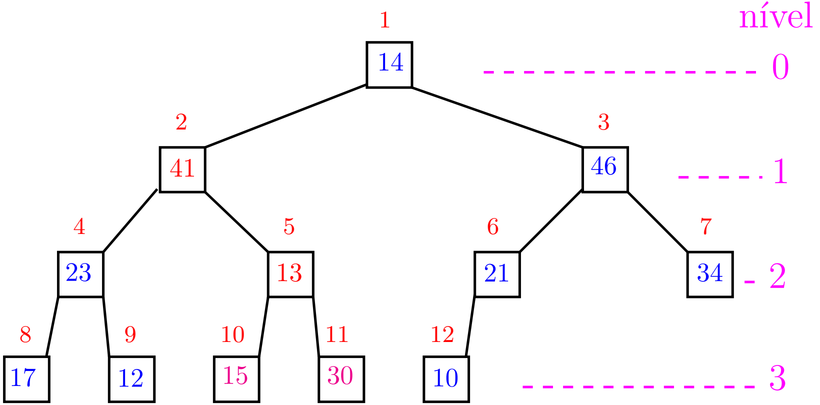
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um `max-heap`



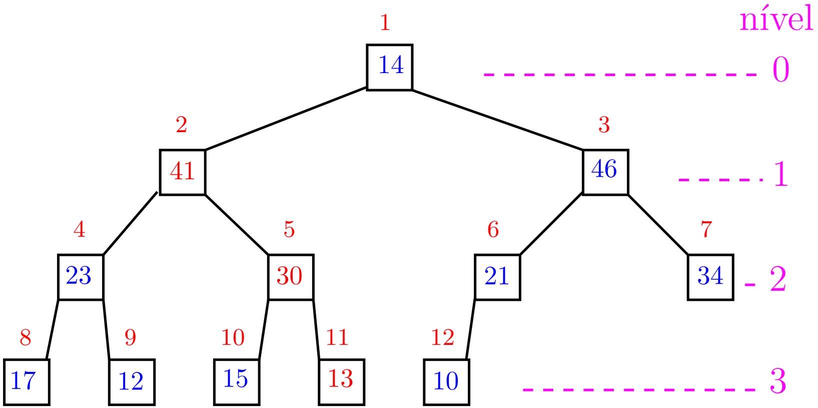
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



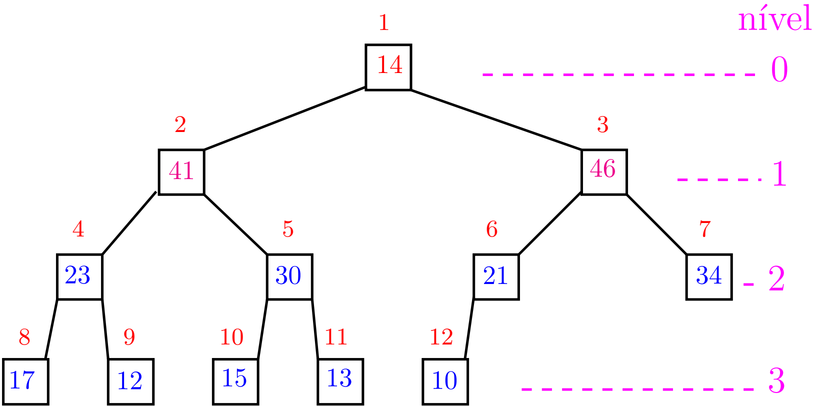
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	13	21	34	17	12	15	30	10

Construção de um max-heap



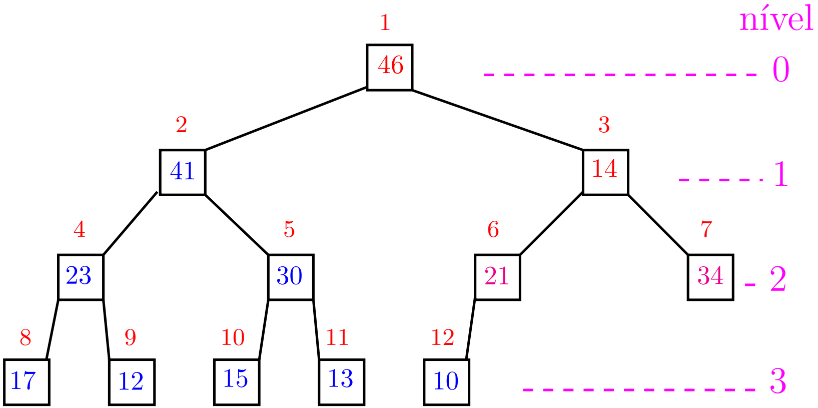
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



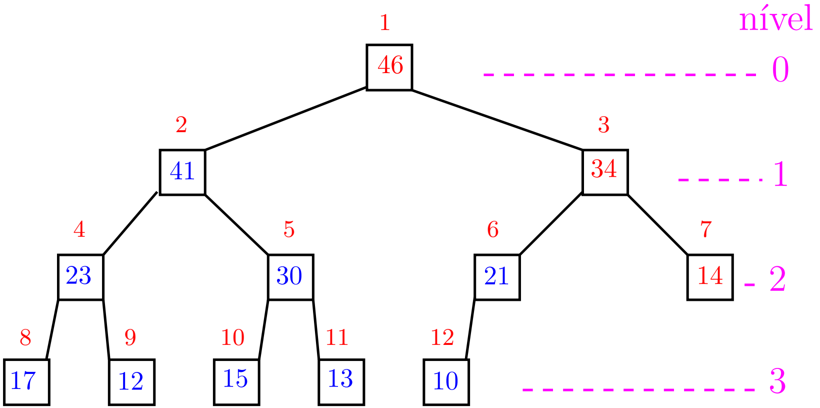
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



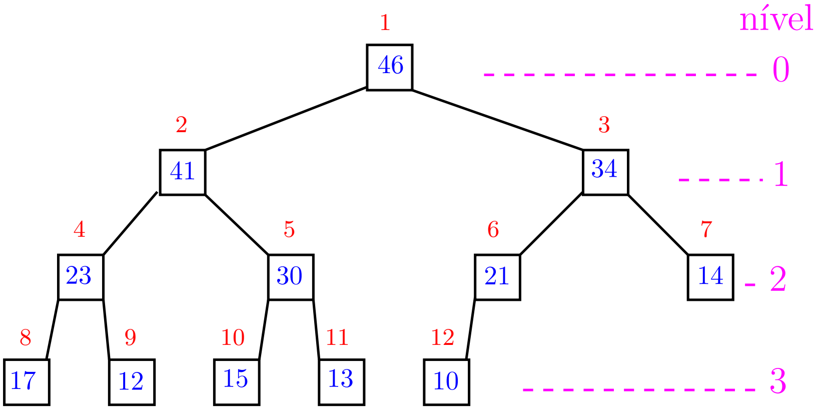
1	2	3	4	5	6	7	8	9	10	11	12
46	41	14	23	30	21	34	17	12	15	13	10

Construção de um max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Construção de um max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Construção de um max-heap

Recebe um vetor $a[1\,..\,n]$ e rearranja a para que seja max-heap.

```
for (int i = n/2; /*A*/ i >= 1; i--)
    sink(i, n, a);
```

Relação invariante:

(i0) em /*A*/ vale que, $i+1, \ldots, n$ são raízes de max-heaps.

Consumo de tempo

Análise grosseira: consumo de tempo é

$$\frac{\mathtt{n}}{2} \times \lg \mathtt{n} = \mathrm{O}(\mathtt{n} \lg \mathtt{n}).$$

Verdade seja dita . . . (. . .)

Análise mais cuidadosa: consumo de tempo é $\mathrm{O}(\mathtt{n})$.

Algumas séries

Para todo número real $\mathtt{x}$, $|\mathtt{x}| < 1$, temos que $\sum_{i=0}^{\infty} \mathtt{x}^{i} = \frac{1}{1-\mathtt{x}}$.

Algumas séries

Para todo número real x, $|x| < 1$, temos que $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$.

Para todo número real x, $|x| < 1$, temos que

$$\sum_{i=1}^{\infty} i\, x^i = \frac{x}{(1-x)^2}$$

Algumas séries

Para todo número real $\mathtt{x}$, $|\mathtt{x}| < 1$, temos que $\sum_{i=0}^{\infty} \mathtt{x}^i = \frac{1}{1-\mathtt{x}}$.

Para todo número real $\mathtt{x}$, $|\mathtt{x}| < 1$, temos que

$$\sum_{i=1}^{\infty} i\,\mathtt{x}^i = \frac{\mathtt{x}}{(1-\mathtt{x})^2}$$

Prova:

$$\begin{aligned}
\sum_{i=1}^{\infty} i\,\mathtt{x}^i &= \sum_{i=1}^{\infty} x^i + \sum_{i=2}^{\infty} \mathtt{x}^i + \cdots + \sum_{i=k}^{\infty} \mathtt{x}^i + \cdots \\
&= \frac{\mathtt{x}}{1-\mathtt{x}} + \frac{\mathtt{x}^2}{1-\mathtt{x}} + \cdots + \frac{\mathtt{x}^{\mathtt{k}}}{1-\mathtt{x}} + \cdots \\
&= \frac{\mathtt{x}}{1-\mathtt{x}}(\mathtt{x}^0 + \mathtt{x}^1 + \mathtt{x}^2 + \cdots + \mathtt{x}^{\mathtt{k}} + \cdots) = \frac{\mathtt{x}}{(1-\mathtt{x}
\end{aligned}$$

Conclusão

O consumo de tempo para construir um `max-heap` é $\mathrm{O}(\mathtt{n} \lg \mathtt{n})$.

Verdade seja dita . . . (. . .)

O consumo de tempo para construir um `max-heap` é $\mathrm{O}(\mathtt{n})$.