Heap de Fibonacci

André Atanasio Maranhão Almeida – IC/UNICAMP

20 de setembro de 2007

Roteiro

- Introdução

Roteiro

- Introdução
- Estrutura

Roteiro

- Introdução
- Estrutura
- Análise amortizada

Roteiro

- Introdução
- Estrutura
- Análise amortizada
- Operações
  - MAKE-HEAP
  - INSERT
  - MINIMUM
  - UNION
  - EXTRACT-MIN
  - DECREASE-KEY
  - DELETE

Roteiro

- Introdução
- Estrutura
- Análise amortizada
- Operações
  - MAKE-HEAP
  - INSERT
  - MINIMUM
  - UNION
  - EXTRACT-MIN
  - DECREASE-KEY
  - DELETE
- Limite do grau máximo

Roteiro

- Introdução
- Estrutura
- Análise amortizada
- Operações
  - MAKE-HEAP
  - INSERT
  - MINIMUM
  - UNION
  - EXTRACT-MIN
  - DECREASE-KEY
  - DELETE
- Limite do grau máximo
- Conclusão

Introdução

- Heap binomial dá suporte às operações em $O(\lg n)$

Introdução

- Heap binomial dá suporte às operações em $O(\lg n)$
- Heap de Fibonacci dá suporte às mesmas operações, mas executa em tempo amortizado $O(1)$ quando não envolve exclusões

- Heap binomial dá suporte às operações em $O(\lg n)$
- Heap de Fibonacci dá suporte às mesmas operações, mas executa em tempo amortizado $O(1)$ quando não envolve exclusões

	Heap		
Operação	**Binário** (pior caso)	**Binomial** (pior caso)	**Fibonacci** (amortizado)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

- Se aplica bem, do ponto de vista teórico, em aplicações onde o número de EXTRACT-MIN e DELETE é pequeno em relação às outras operações

- Se aplica bem, do ponto de vista teórico, em aplicações onde o número de EXTRACT-MIN e DELETE é pequeno em relação às outras operações
- Aplicações: cálculo de árvores geradoras mínimas e busca por menores caminhos dada uma origem

Introdução

- É um bom exemplo de estrutura de dados projetada com análise amortizada em mente

Introdução

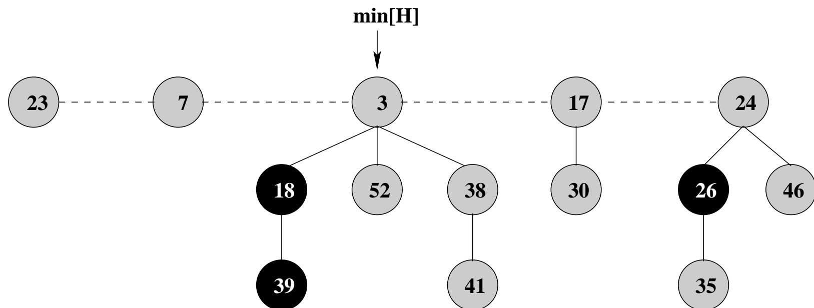
- É um bom exemplo de estrutura de dados projetada com análise amortizada em mente
- Não foi projetada para dar um suporte eficiente a operação SEARCH

Introdução

- É um bom exemplo de estrutura de dados projetada com análise amortizada em mente
- Não foi projetada para dar um suporte eficiente a operação SEARCH
- Do ponto de vista prático, os fatores constantes e a complexidade a tornam pouco desejável para a maioria dos problemas

Estrutura

- Coleção de heaps mínimos



Estrutura

- Cada nó x possui um ponteiro $p[x]$ para o pai

Estrutura

- Cada nó x possui um ponteiro $p[x]$ para o pai
- Cada nó x possui um ponteiro $child[x]$ para qualquer dos filhos

Estrutura

- Cada nó x possui um ponteiro $p[x]$ para o pai
- Cada nó x possui um ponteiro $child[x]$ para qualquer dos filhos
- Os filhos de x mantêm-se unidos por uma lista circular duplamente ligada

Estrutura

- Cada nó x possui um ponteiro $p[x]$ para o pai
- Cada nó x possui um ponteiro $child[x]$ para qualquer dos filhos
- Os filhos de x mantêm-se unidos por uma lista circular duplamente ligada
- Cada filho y tem ponteiros $left[y]$ e $right[y]$. Quando tem apenas um filho $left[y] = right[y] = y$

Estrutura

- Cada nó x possui um ponteiro $p[x]$ para o pai
- Cada nó x possui um ponteiro $child[x]$ para qualquer dos filhos
- Os filhos de x mantêm-se unidos por uma lista circular duplamente ligada
- Cada filho y tem ponteiros $left[y]$ e $right[y]$. Quando tem apenas um filho $left[y] = right[y] = y$
- Tal lista permite operações como exclusão de um nó ou união/divisão de listas em tempo $O(1)$

Estrutura

- Para cada nó x também é mantido o número de filhos em $degree[x]$ e um valor booleano em $mark[x]$, que indica se x perdeu algum filho desde que teve o ponteiro $p[x]$ alterado

Estrutura

- Para cada nó x também é mantido o número de filhos em $degree[x]$ e um valor booleano em $mark[x]$, que indica se x perdeu algum filho desde que teve o ponteiro $p[x]$ alterado
- Seja H um heap de Fibonacci. $min[H]$ é um ponteiro para a raiz da árvore que contém a menor chave. O nó referenciado por $min[H]$ é chamado de **nó mínimo** de H. Quando o heap está vazio temos $min[H] = NIL$

Estrutura

- Para cada nó x também é mantido o número de filhos em $degree[x]$ e um valor booleano em $mark[x]$, que indica se x perdeu algum filho desde que teve o ponteiro $p[x]$ alterado
- Seja H um heap de Fibonacci. $min[H]$ é um ponteiro para a raiz da árvore que contém a menor chave. O nó referenciado por $min[H]$ é chamado de **nó mínimo** de H. Quando o heap está vazio temos $min[H] = NIL$
- As raízes dos heaps também são mantidas numa lista circular duplamente ligada, que recebe o nome de **lista de raízes**

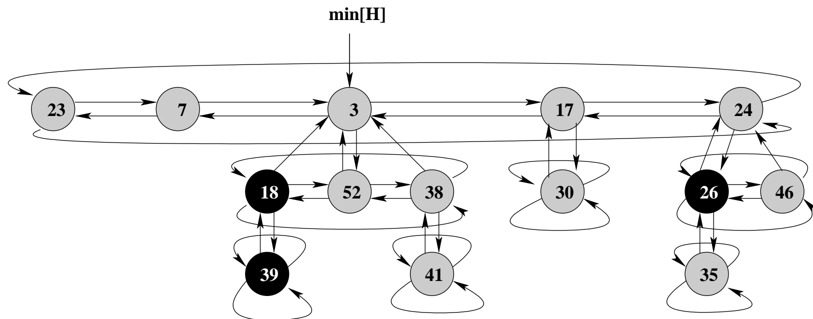
Estrutura

- Para cada nó x também é mantido o número de filhos em $degree[x]$ e um valor booleano em $mark[x]$, que indica se x perdeu algum filho desde que teve o ponteiro $p[x]$ alterado
- Seja H um heap de Fibonacci. $min[H]$ é um ponteiro para a raiz da árvore que contém a menor chave. O nó referenciado por $min[H]$ é chamado de **nó mínimo** de H. Quando o heap está vazio temos $min[H] = NIL$
- As raízes dos heaps também são mantidas numa lista circular duplamente ligada, que recebe o nome de **lista de raízes**
- A ordem das árvores na lista é aleatória

Estrutura

- Para cada nó x também é mantido o número de filhos em $degree[x]$ e um valor booleano em $mark[x]$, que indica se x perdeu algum filho desde que teve o ponteiro $p[x]$ alterado
- Seja H um heap de Fibonacci. $min[H]$ é um ponteiro para a raiz da árvore que contém a menor chave. O nó referenciado por $min[H]$ é chamado de **nó mínimo** de H. Quando o heap está vazio temos $min[H] = NIL$
- As raízes dos heaps também são mantidas numa lista circular duplamente ligada, que recebe o nome de **lista de raízes**
- A ordem das árvores na lista é aleatória
- O heap ainda possui o atributo $n[H]$, que indica o número de nós

Estrutura



- O tempo requerido para executar uma operação é a média de todas as operações realizadas

Análise amortizada

- O tempo requerido para executar uma operação é a média de todas as operações realizadas
- Pode ser utilizada para mostrar que o custo médio de uma operação é pequeno, mesmo que uma delas em específico tenha um alto custo

Análise amortizada

- O tempo requerido para executar uma operação é a média de todas as operações realizadas
- Pode ser utilizada para mostrar que o custo médio de uma operação é pequeno, mesmo que uma delas em específico tenha um alto custo
- Análise amortizada difere da análise do caso médio. Na análise do caso médio é computada a média do custo do algortimo para todas as possíveis instâncias. A análise amortizada calcula o custo médio de cada operação para qualquer instância. Geralmente é muito mais fácil fazer uma análise amortizada que uma análise de caso médio.

Análise amortizada

- O tempo requerido para executar uma operação é a média de todas as operações realizadas
- Pode ser utilizada para mostrar que o custo médio de uma operação é pequeno, mesmo que uma delas em específico tenha um alto custo
- Análise amortizada difere da análise do caso médio. Na análise do caso médio é computada a média do custo do algortimo para todas as possíveis instâncias. A análise amortizada calcula o custo médio de cada operação para qualquer instância. Geralmente é muito mais fácil fazer uma análise amortizada que uma análise de caso médio.
- Técnicas: análise agregada, método da contagem, método potencial

Análise amortizada

- Contador binário de k-bits, que inicia em 0. $A[0..k-1]$

Análise amortizada

- Contador binário de k-bits, que inicia em 0. $A[0..k-1]$

Algoritmo

```
INCREMENT(A)
 1. i = 0
 2. Enquanto i < length(A) e A[i] = 1 faça
 3.    A[i] = 0
 4.    i = i + 1
 5. Se i < length(A) então
 6.    A[i] = 1
```

Análise amortizada

- Contador binário de k-bits, que inicia em 0. $A[0..k-1]$

Algoritmo

```
INCREMENT(A)
 1. i = 0
 2. Enquanto i < length(A) e A[i] = 1 faça
 3.    A[i] = 0
 4.    i = i + 1
 5. Se i < length(A) então
 6.    A[i] = 1
```

- O tempo de execução é proporcional ao número de alterações dos bits

Análise amortizada

- Utilizando o método da contagem, atribuímos o custo de 1 moeda por alteração e "pagamos" 2 moedas a cada alteração para 1 (1 para a alteração corrente e 1 de "crédito" para ser utilizada na alteração do bit de volta para 0). Desta forma, temos que a cada execução, no máximo, 2 moedas são gastas e assim temos um custo máximo de $2n$ moedas para n execuções. Concluímos que o custo total de n chamadas a INCREMENT é $O(n)$ e que o custo amortizado de cada operação é $O(1)$

Análise amortizada

- Método potencial

Análise amortizada

- Método potencial
- Estrutura inicial D_0, onde executamos n operações

Análise amortizada

- Método potencial
- Estrutura inicial D_0, onde executamos n operações
- c_i, para cada $i = 1, 2, \ldots, n$, é o custo real da i-ésima operação em D_{i-1} que gera D_i

- Método potencial
- Estrutura inicial D_0, onde executamos n operações
- c_i, para cada $i = 1, 2, \ldots, n$, é o custo real da i-ésima operação em D_{i-1} que gera D_i
- Função potencial Φ mapeia D_i a um número real $\Phi(D_i)$, o potencial

Análise amortizada

- Método potencial
- Estrutura inicial D_0, onde executamos n operações
- c_i, para cada $i = 1, 2, \ldots, n$, é o custo real da i-ésima operação em D_{i-1} que gera D_i
- Função potencial Φ mapeia D_i a um número real $\Phi(D_i)$, o potencial
- O custo amortizado $\widehat{c_i}$ da i-ésima operação é:

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Análise amortizada

- Método potencial
- Estrutura inicial D_0, onde executamos n operações
- c_i, para cada $i = 1, 2, \ldots, n$, é o custo real da i-ésima operação em D_{i-1} que gera D_i
- Função potencial Φ mapeia D_i a um número real $\Phi(D_i)$, o potencial
- O custo amortizado $\widehat{c_i}$ da i-ésima operação é:

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- O custo amortizado total é:

$$\sum_{i=1}^{n} \widehat{c_i} = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_1)$$

Função potencial

- Contador binário

Função potencial

- Contador binário
  - $\Phi(D_i) = b_i$, onde b_i é o número de 1's

Função potencial

- Contador binário
  - $\Phi(D_i) = b_i$, onde b_i é o número de 1's
  - t_i resets. Custo real da operação é $c_i = t_i + 1$

Função potencial

- Contador binário
  - $\Phi(D_i) = b_i$, onde b_i é o número de 1's
  - t_i resets. Custo real da operação é $c_i = t_i + 1$
  - Custo amortizado da operação

$$
\begin{aligned}
\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq (t_i + 1) + (b_{i-1} - t_i + 1) - b_{i-1} \\
&= 2
\end{aligned}
$$

- Contador binário
  - $\Phi(D_i) = b_i$, onde b_i é o número de 1's
  - t_i resets. Custo real da operação é $c_i = t_i + 1$
  - Custo amortizado da operação

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq (t_i + 1) + (b_{i-1} - t_i + 1) - b_{i-1} \\
&= 2
\end{aligned}$$

- Heap de Fibonacci

Função potencial

- Contador binário
  - $\Phi(D_i) = b_i$, onde b_i é o número de 1's
  - t_i resets. Custo real da operação é $c_i = t_i + 1$
  - Custo amortizado da operação

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 2 \end{aligned}$$

- Heap de Fibonacci
  - $t(H)$ denota o número de heaps mínimos

Função potencial

- Contador binário
  - $\Phi(D_i) = b_i$, onde b_i é o número de 1's
  - t_i resets. Custo real da operação é $c_i = t_i + 1$
  - Custo amortizado da operação

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 2 \end{aligned}$$

- Heap de Fibonacci
  - $t(H)$ denota o número de heaps mínimos
  - $m(H)$ denota o número de nós marcados

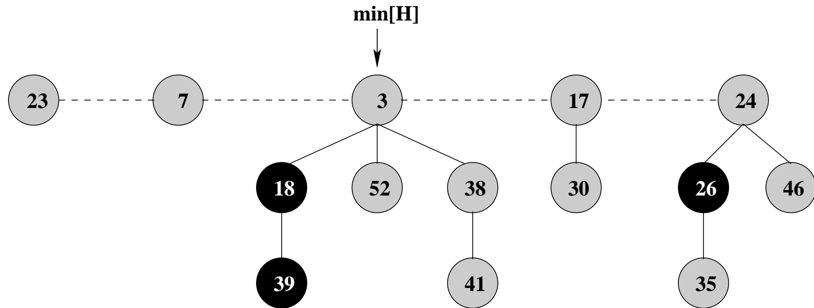
Função potencial

- Contador binário
  - $\Phi(D_i) = b_i$, onde b_i é o número de 1's
  - t_i resets. Custo real da operação é $c_i = t_i + 1$
  - Custo amortizado da operação

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 2 \end{aligned}$$

- Heap de Fibonacci
  - $t(H)$ denota o número de heaps mínimos
  - $m(H)$ denota o número de nós marcados
  - $\Phi(H) = t(H) + 2 * m(H)$

Função potencial



$$\Phi(H) = t(H) + 2 * m(H) = 5 + 2 * 3 = 11$$

MAKE-HEAP

Algoritmo

```
MAKE-HEAP()
 1.   n[H] = 0
 2.   min[H] = NIL
 3.   retorna H
```

MAKE-HEAP

Algoritmo

```
MAKE-HEAP()
 1.   n[H] = 0
 2.   min[H] = NIL
 3.   retorna H
```

- Análise amortizada

$$\Phi(H) = t(H) + 2 * m(H) = 0 + 2 * 0 = 0$$

$$\widehat{c_i} = c_i = O(1)$$

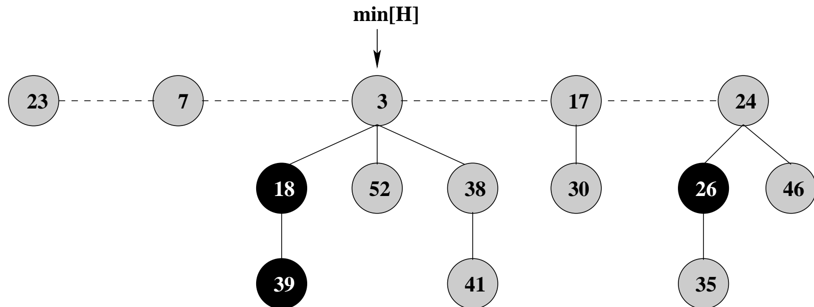
INSERT

Algoritmo

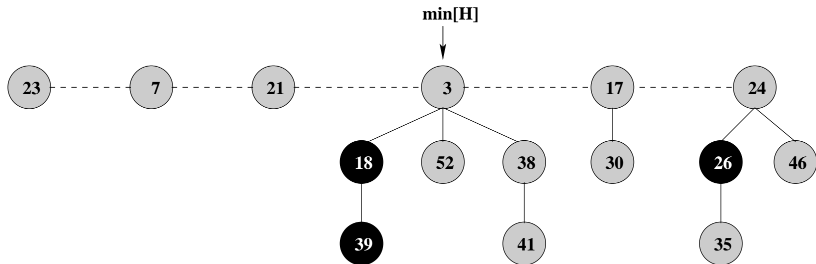
```
INSERT(H,x)
 1. degree[x] = 0
 2. p[x] = NIL
 3. child[x] = NIL
 4. left[x] = x
 5. right[x] = x
 6. mark[x] = FALSE
 7. concatena as listas de raízes H e a de x
 8. Se min[H] = NIL ou key[x] < key[min[H]] então
 9.   min[H] = x
10. n[H] = n[H] + 1
```

INSERT

- Inserir: 21



INSERT



INSERT – Análise amortizada

- H o heap de Fibonacci de entrada

INSERT – Análise amortizada

- H o heap de Fibonacci de entrada
- H' o heap de Fibonacci resultante

INSERT – Análise amortizada

- H o heap de Fibonacci de entrada
- H' o heap de Fibonacci resultante
- $t(H') = t(H) + 1$

INSERT – Análise amortizada

- H o heap de Fibonacci de entrada
- H' o heap de Fibonacci resultante
- $t(H') = t(H) + 1$
- $m(H') = m(H)$

INSERT – Análise amortizada

- H o heap de Fibonacci de entrada
- H' o heap de Fibonacci resultante
- $t(H') = t(H) + 1$
- $m(H') = m(H)$
- Segue a varição do potencial e o custo amortizado da operação

$$\Phi(H') - \Phi(H) = ((t(H)+1) + 2*m(H)) - (t(H) + 2*m(H)) = 1$$

$$\widehat{c_i} = c_i + \Phi(H') - \Phi(H) = c_i + 1 = O(1)$$

MINIMUM

Algoritmo

```
MINIMUM(H)
 1.  retorna min[H]
```

MINIMUM

Algoritmo

```
MINIMUM(H)
 1.   retorna min[H]
```

- Análise amortizada

MINIMUM

Algoritmo

```
MINIMUM(H)
 1.   retorna min[H]
```

- Análise amortizada
  - $t(H') = t(H)$

MINIMUM

Algoritmo

```
MINIMUM(H)
 1.  retorna min[H]
```

- Análise amortizada
  - $t(H') = t(H)$
  - $m(H') = m(H)$

MINIMUM

Algoritmo

```
MINIMUM(H)
 1.   retorna min[H]
```

- Análise amortizada
  - $t(H') = t(H)$
  - $m(H') = m(H)$
  - $\Phi(H') - \Phi(H) = 0$

MINIMUM

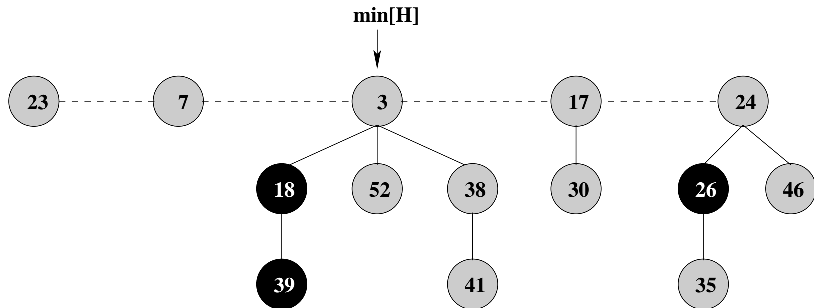
Algoritmo

```
MINIMUM(H)
 1.   retorna min[H]
```

- Análise amortizada
  - $t(H') = t(H)$
  - $m(H') = m(H)$
  - $\Phi(H') - \Phi(H) = 0$
  - Custo amortizado da operação

$$\widehat{c_i} = c_i + \Phi(H') - \Phi(H) = c_i = O(1)$$

MINIMUM

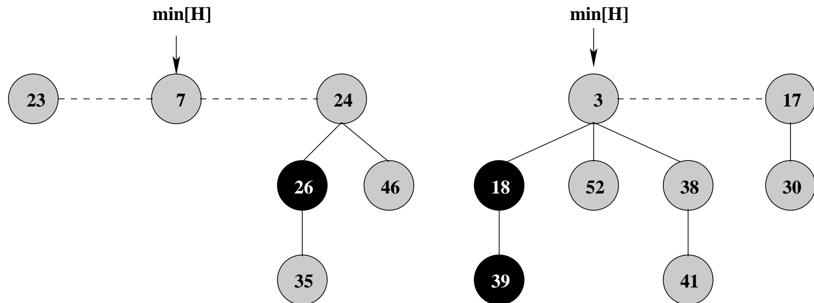


UNION

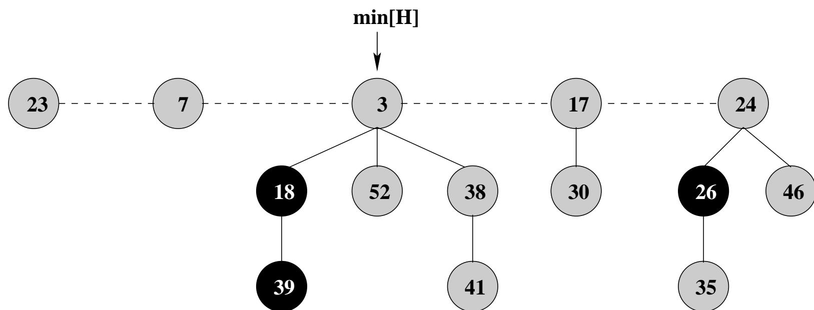
Algoritmo

```
UNION(H1,H2)
 1. H = MAKE-HEAP()
 2. min[H] = min[H1]
 3. concatena as listas de raízes de H2 com a de H
 4. Se (min[H1] = NIL) ou
 5.    (min[H2] ≠ NIL e min[H2] < min[H1]) então
 6.      min[H] = min[H2]
 7. n[H] = n[H1] + n[H2]
 8. retorna H
```

UNION



UNION



UNION – Análise amortizada

- $t(H) = t(H_1) + t(H_2)$

UNION – Análise amortizada

- $t(H) = t(H_1) + t(H_2)$
- $m(H) = m(H_1) + m(H_2)$

UNION – Análise amortizada

- $t(H) = t(H_1) + t(H_2)$
- $m(H) = m(H_1) + m(H_2)$
- Segue a variação do potencial e o custo amortizado da operação

$$\Phi(H) - (\Phi(H_1) + \Phi(H_2)) = 0$$

$$\widehat{c_i} = c_i + \Phi(H) - (\Phi(H_1) + \Phi(H_2)) = c_i = O(1)$$

EXTRACT-MIN

Algoritmo

```
EXTRACT-MIN(H)
 1. z = min[H]
 2. Se z ≠ NIL então
 3.   Para cada filho x de z faça
 4.     adiciona x a lista de raízes de H
 5.     p[x] = NIL
 6.   remove z da lista de raízes de H
 7.   Se z = right[z] então
 8.     min[H] = NIL
 9.   Senão
10.     min[H] = right[z]
11.     CONSOLIDATE(H)
12.   n[H] = n[H] - 1
13. retorna z
```

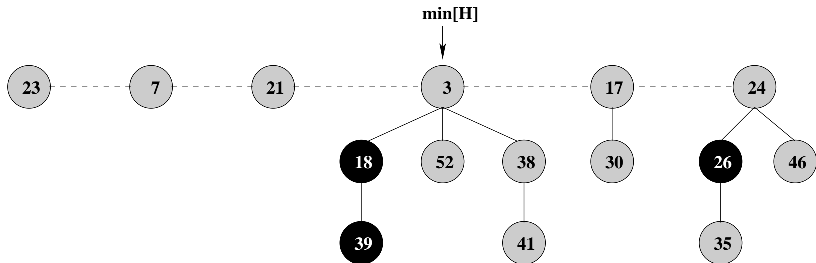
EXTRACT-MIN

CONSOLIDATE(H)

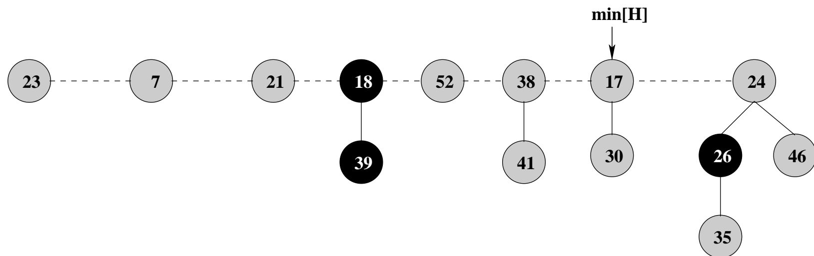
Repetidamente executa os seguintes passos até que toda raiz possua um grau distinto.

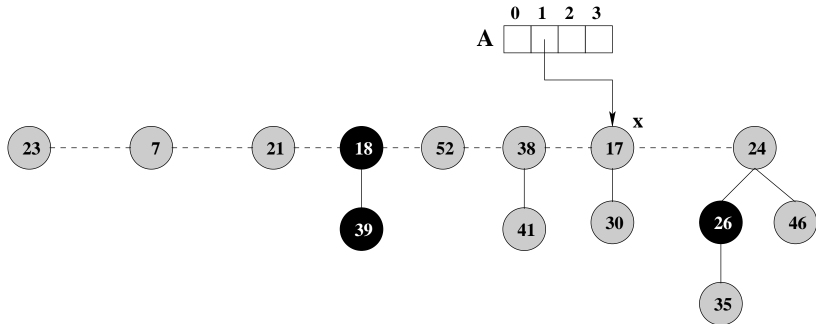
1. Encontrar duas raízes x e y com mesmo grau tal que $key[x] \leq key[y]$.
2. Remover y da lista de raízes e tornar y filho de x. Incrementar $degree[x]$ e $mark[y] = FALSE$.

EXTRACT-MIN

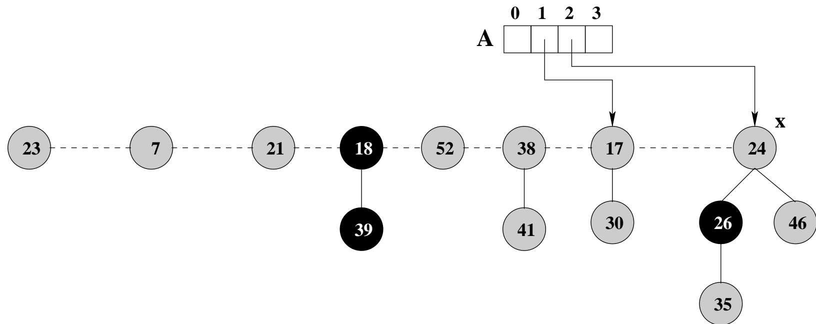


EXTRACT-MIN

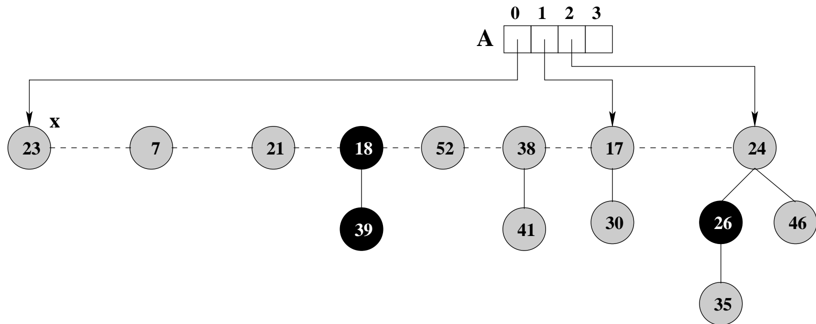



A
0 1 2 3
x
23
7
21
18
52
38
17
24
39
41
30
26
46
35

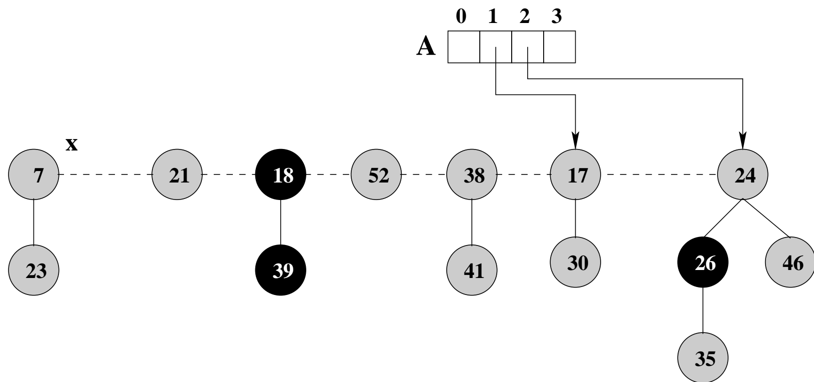
EXTRACT-MIN



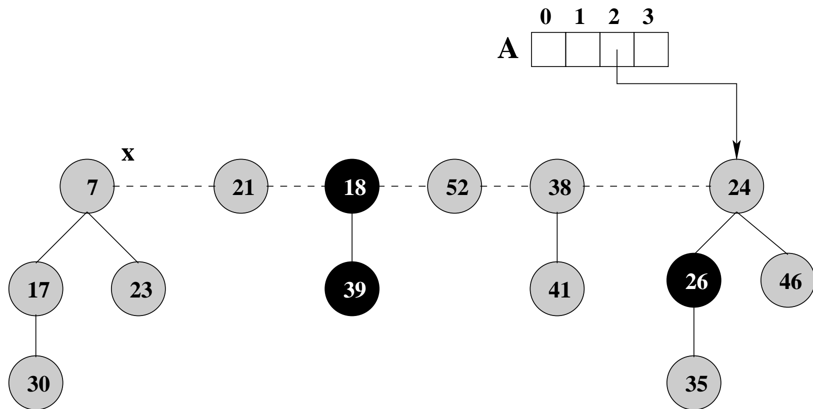
EXTRACT-MIN



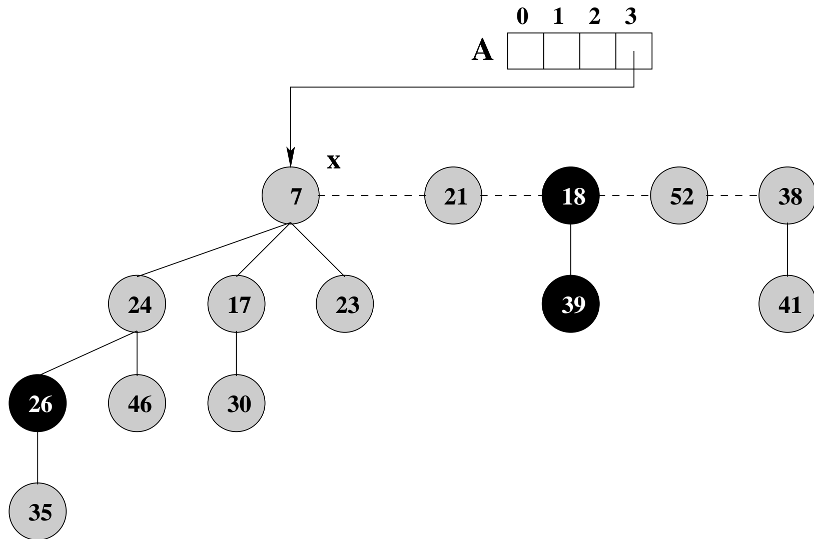
EXTRACT-MIN



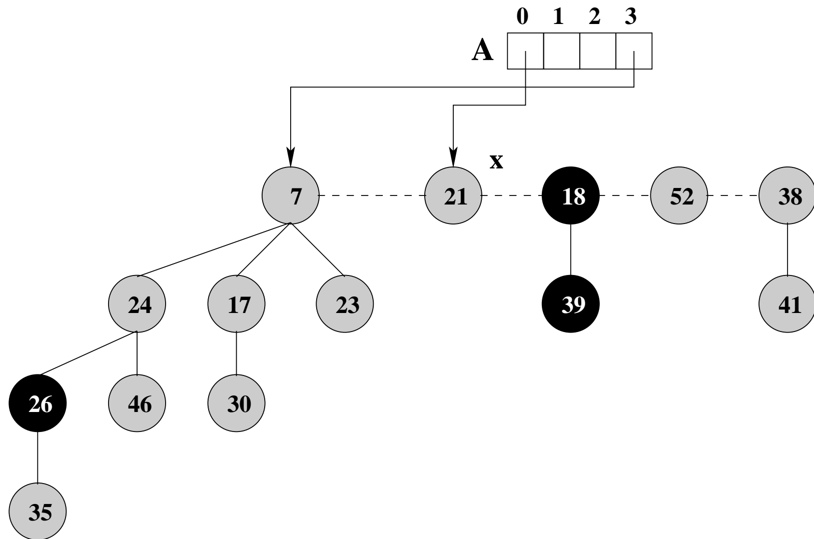
EXTRACT-MIN

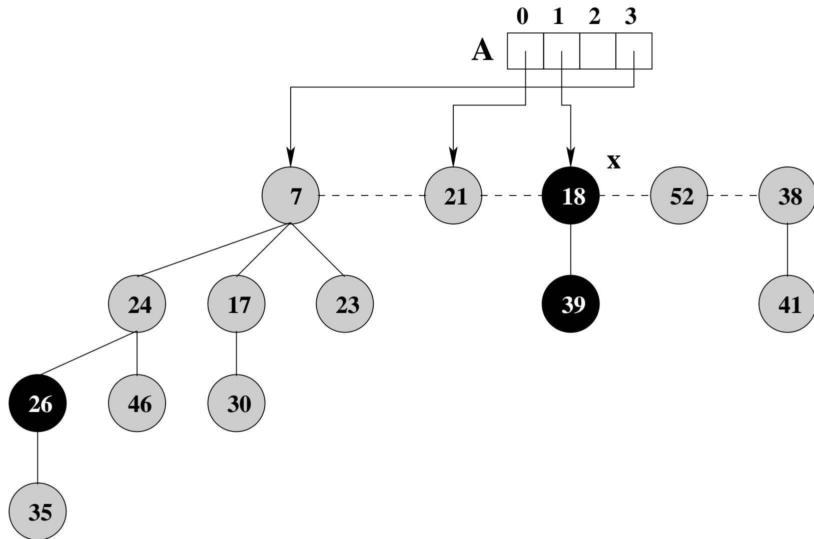


EXTRACT-MIN

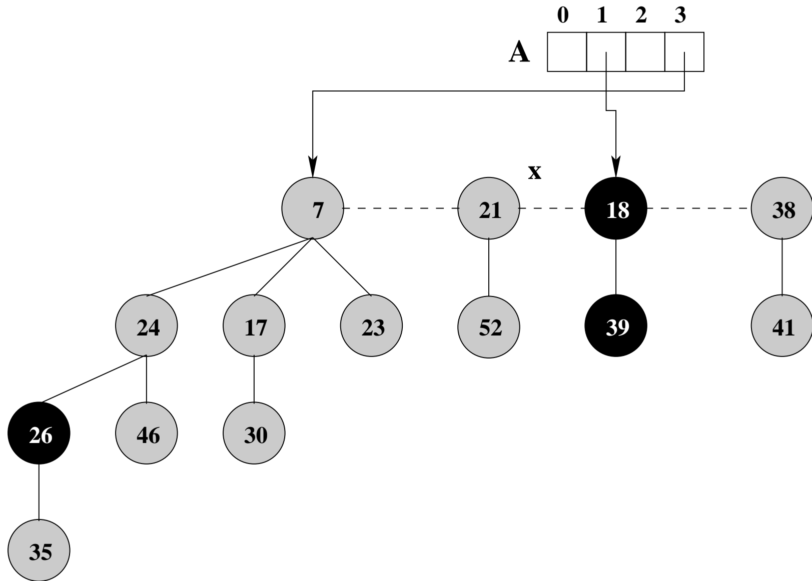


EXTRACT-MIN

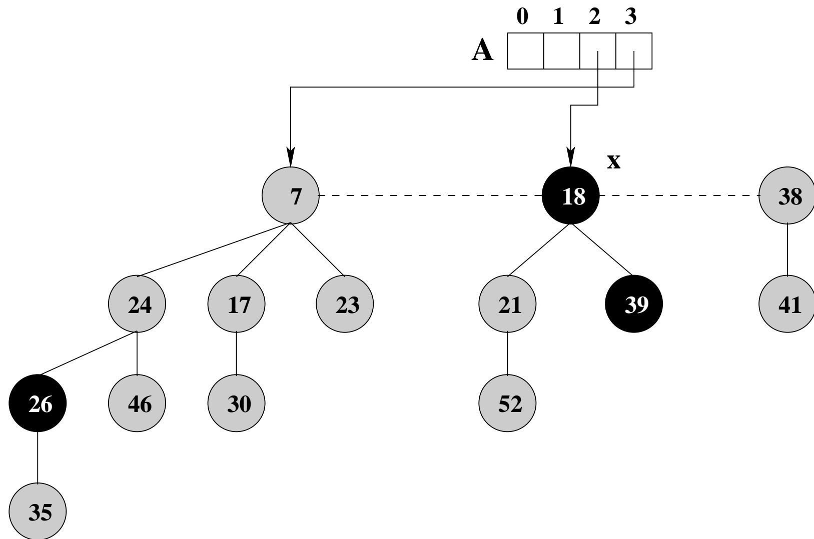


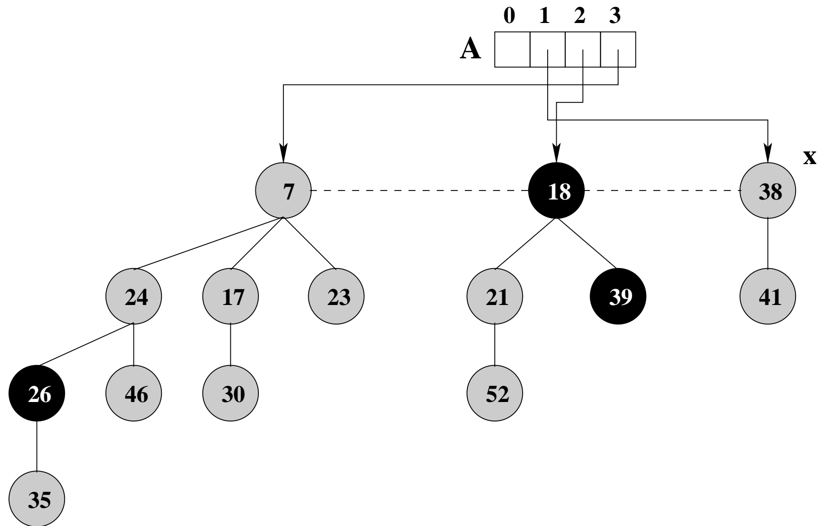

0 1 2 3
A
x
7
21
18
52
38
24
17
23
39
41
26
46
30
35

EXTRACT-MIN

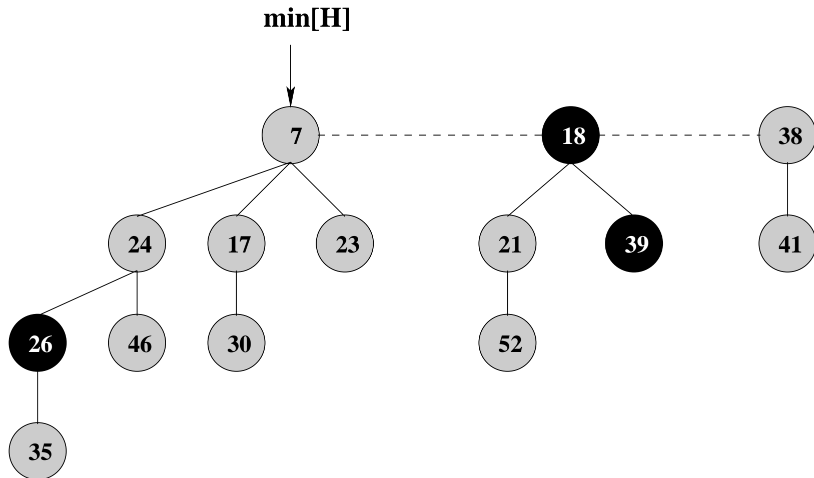


EXTRACT-MIN




A
0
1
2
3
x
7
18
38
24
17
23
21
39
41
26
46
30
52
35

EXTRACT-MIN



EXTRACT-MIN

Algoritmo

```
EXTRACT-MIN(H)
 1. z = min[H]
 2. Se z ≠ NIL então
 3.   Para cada filho x de z faça
 4.     adiciona x a lista de raízes de H
 5.     p[x] = NIL
 6.   remove z da lista de raízes de H
 7.   Se z = right[z] então
 8.     min[H] = NIL
 9.   Senão
10.     min[H] = right[z]
11.     CONSOLIDATE(H)
12.   n[H] = n[H] - 1
13. retorna z
```

EXTRACT-MIN – Análise amortizada

- $\Phi(H) = t(H) + 2 * m(H)$

EXTRACT-MIN – Análise amortizada

- $\Phi(H) = t(H) + 2 * m(H)$
- $\Phi(H') = (D(n) + 1) + 2 * m(H)$, no máximo

EXTRACT-MIN – Análise amortizada

- $\Phi(H) = t(H) + 2 * m(H)$
- $\Phi(H') = (D(n) + 1) + 2 * m(H)$, no máximo
- $O(D(n) + t(H))$ é o custo real, uma vez que é determinado pelo CONSOLIDATE e o custo deste é proporcional ao número de heaps. Observe que tínhamos $t(H)$ raízes inicialmente, mas quando excluímos $min[H]$ podem surgir até $D(n)$ novas raízes.

EXTRACT-MIN – Análise amortizada

- $\Phi(H) = t(H) + 2 * m(H)$
- $\Phi(H') = (D(n) + 1) + 2 * m(H)$, no máximo
- $O(D(n) + t(H))$ é o custo real, uma vez que é determinado pelo CONSOLIDATE e o custo deste é proporcional ao número de heaps. Observe que tínhamos $t(H)$ raízes inicialmente, mas quando excluímos $min[H]$ podem surgir até $D(n)$ novas raízes.
- Segue a variação no potencial e o custo amortizado da operação

$$\begin{aligned}
\Phi(H') - \Phi(H) &= ((D(n) + 1) + 2 * m(H)) - (t(H) + 2 * m(H)) \\
&= D(n) - t(H) + 1
\end{aligned}$$

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi(H') - \Phi(H) \\
&= O(D(n) + t(H)) + D(n) - t(H) + 1 \\
&= O(D(n)) + O(t(H)) + D(n) - t(H) + 1 \\
&= O(D(n))
\end{aligned}$$

EXTRACT-MIN – Análise amortizada

- $\Phi(H) = t(H) + 2 * m(H)$
- $\Phi(H') = (D(n) + 1) + 2 * m(H)$, no máximo
- $O(D(n) + t(H))$ é o custo real, uma vez que é determinado pelo CONSOLIDATE e o custo deste é proporcional ao número de heaps. Observe que tínhamos $t(H)$ raízes inicialmente, mas quando excluímos $min[H]$ podem surgir até $D(n)$ novas raízes.
- Segue a variação no potencial e o custo amortizado da operação

$$\begin{aligned}
\Phi(H') - \Phi(H) &= ((D(n) + 1) + 2 * m(H)) - (t(H) + 2 * m(H)) \\
&= D(n) - t(H) + 1
\end{aligned}$$

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi(H') - \Phi(H) \\
&= O(D(n) + t(H)) + D(n) - t(H) + 1 \\
&= O(D(n)) + O(t(H)) + D(n) - t(H) + 1 \\
&= O(D(n))
\end{aligned}$$

DECREASE-KEY

Algoritmo

```
DECREASE-KEY(H,x,k)
 1. Se k > key[x] então
 2.   erro ''Nova chave é maior''
 3. key[x] = k
 4. y = p[x]
 5. Se y ≠ NIL e key[x] < key[y] então
 6.   CUT(H,x,y)
 7.   CASCADING-CUT(H,y)
 8. Se key[x] < key[min[H]] então
 9.   min[H] = x
```

DECREASE-KEY

Algoritmo

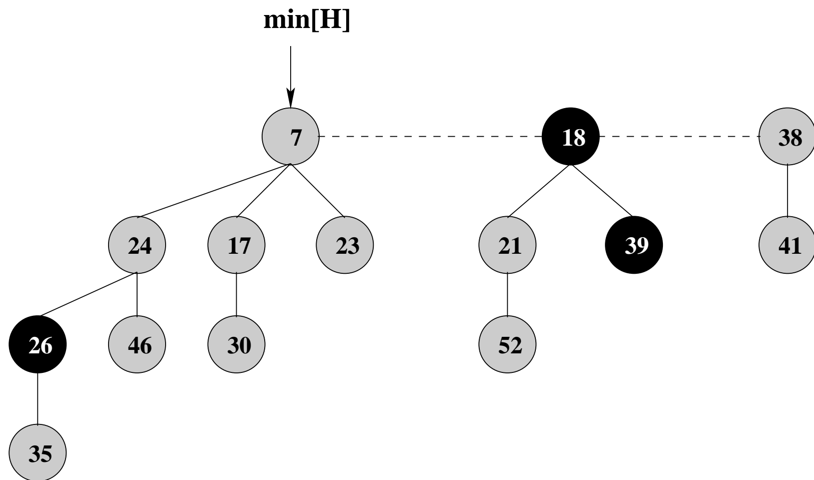
```
CUT(H,x,y)
  1. remove x da lista de filhos de y e decrementa
degree[y]
  2. adiciona x à lista de raízes de H
  3. p[x] = NIL
  4. mark[x] = FALSE
```

Algoritmo

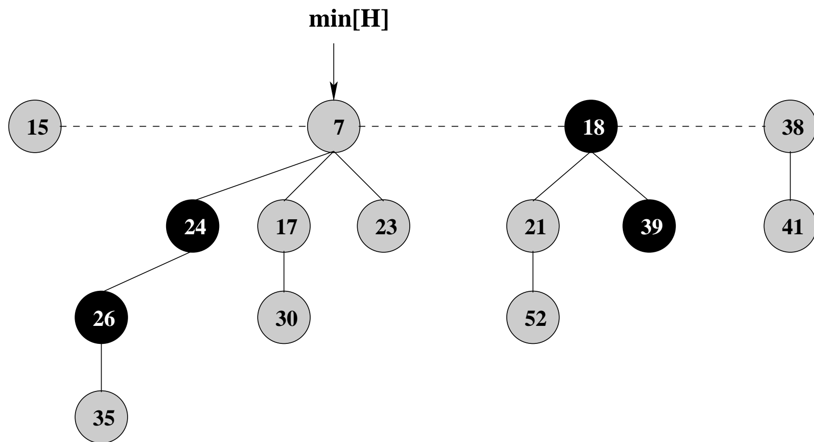
```
CASCADING-CUT(H,y)
 1. z = p[y]
 2. Se z ≠ NIL então
 3.   Se mark[y] = FALSE então
 4.     mark[y] = TRUE
 5.   Senão
 6.     CUT(H,y,z)
 7.     CASCADING-CUT(H,z)
```

DECREASE-KEY

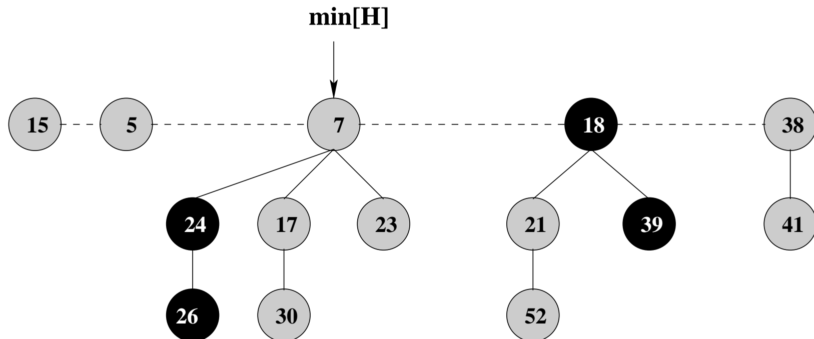
- Decrementar chave 46 para 15



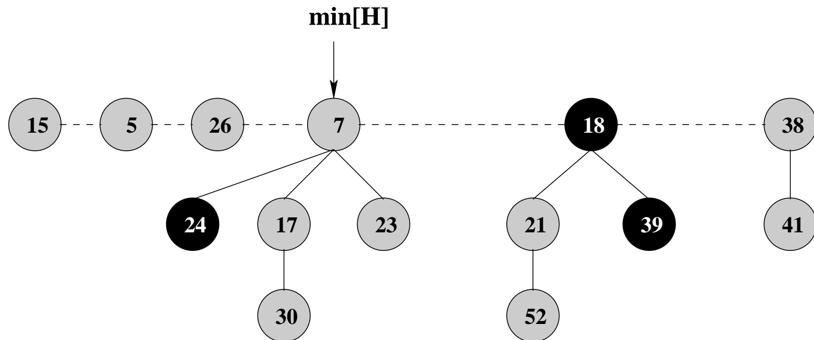
- Decrementar chave 35 para 5



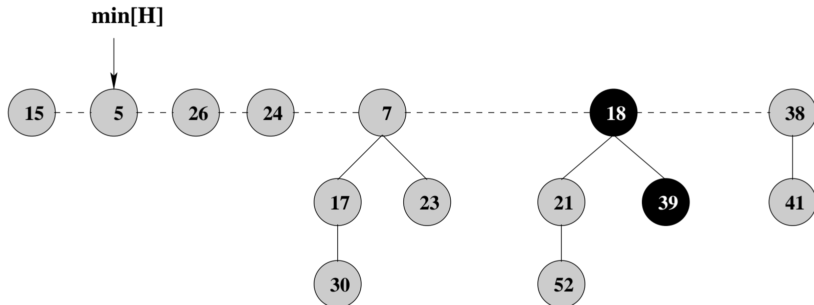
DECREASE-KEY



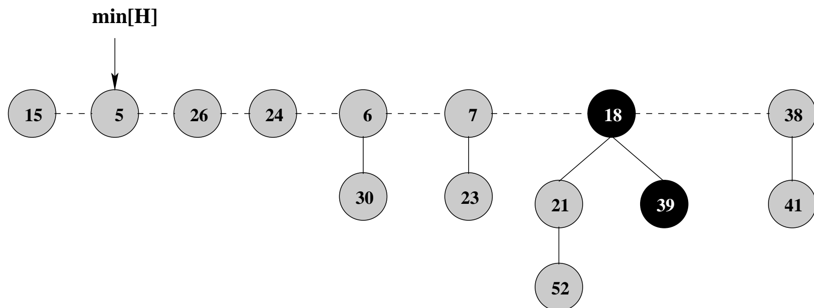
DECREASE-KEY



- Decrementar chave 17 para 6



DECREASE-KEY



DECREASE-KEY

Algoritmo

```
DECREASE-KEY(H,x,k)
 1. Se k > key[x] então
 2.    erro ``Nova chave é maior''
 3. key[x] = k
 4. y = p[x]
 5. Se y ≠ NIL e key[x] < key[y] então
 6.    CUT(H,x,y)
 7.    CASCADING-CUT(H,y)
 8. Se key[x] < key[min[H]] então
 9.    min[H] = x
```

DECREASE-KEY

Algoritmo

```
 CUT(H,x,y)
  1. remove x da lista de filhos de y e decrementa
degree[y]
  2. adiciona x à lista de raízes de H
  3. p[x] = NIL
  4. mark[x] = FALSE
```

DECREASE-KEY

Algoritmo

```
CASCADING-CUT(H,y)
 1. z = p[y]
 2. Se z ≠ NIL então
 3.    Se mark[y] = FALSE então
 4.      mark[y] = TRUE
 5.    Senão
 6.      CUT(H,y,z)
 7.      CASCADING-CUT(H,z)
```

DECREASE-KEY – Análise amortizada

- O custo real do DECREASE-KEY é $O(1)$ mais o custo do CASCADING-CUT.

DECREASE-KEY – Análise amortizada

- O custo real do DECREASE-KEY é $O(1)$ mais o custo do CASCADING-CUT.
- Supomos que o CASCADING-CUT é recursivamente chamado c vezes, tornando o custo real DECREASE-KEY $O(c)$

DECREASE-KEY – Análise amortizada

- O custo real do DECREASE-KEY é $O(1)$ mais o custo do CASCADING-CUT.
- Supomos que o CASCADING-CUT é recursivamente chamado c vezes, tornando o custo real DECREASE-KEY $O(c)$
- $t(H') = t(H) + c$

DECREASE-KEY – Análise amortizada

- O custo real do DECREASE-KEY é $O(1)$ mais o custo do CASCADING-CUT.
- Supomos que o CASCADING-CUT é recursivamente chamado c vezes, tornando o custo real DECREASE-KEY $O(c)$
- $t(H') = t(H) + c$
- São gerados $c - 1$ heaps pelas chamadas a CASCANDING-CUT e o x também torna-se raiz de heap

DECREASE-KEY – Análise amortizada

- O custo real do DECREASE-KEY é $O(1)$ mais o custo do CASCADING-CUT.
- Supomos que o CASCADING-CUT é recursivamente chamado c vezes, tornando o custo real DECREASE-KEY $O(c)$
- $t(H') = t(H) + c$
- São gerados $c - 1$ heaps pelas chamadas a CASCANDING-CUT e o x também torna-se raiz de heap
- $m(H') = m(H) - c + 2$

DECREASE-KEY – Análise amortizada

- O custo real do DECREASE-KEY é $O(1)$ mais o custo do CASCADING-CUT.
- Supomos que o CASCADING-CUT é recursivamente chamado c vezes, tornando o custo real DECREASE-KEY $O(c)$
- $t(H') = t(H) + c$
- São gerados $c - 1$ heaps pelas chamadas a CASCANDING-CUT e o x também torna-se raiz de heap
- $m(H') = m(H) - c + 2$
- São desmarcados $c - 1$ nós pelas chamadas a CASCANDING-CUT e a última execução pode ter marcado 1

DECREASE-KEY – Análise amortizada

- O custo real do DECREASE-KEY é $O(1)$ mais o custo do CASCADING-CUT.
- Supomos que o CASCADING-CUT é recursivamente chamado c vezes, tornando o custo real DECREASE-KEY $O(c)$
- $t(H') = t(H) + c$
- São gerados $c - 1$ heaps pelas chamadas a CASCANDING-CUT e o x também torna-se raiz de heap
- $m(H') = m(H) - c + 2$
- São desmarcados $c - 1$ nós pelas chamadas a CASCANDING-CUT e a última execução pode ter marcado 1
- Segue a variação no potencial e o custo amortizado

$$\begin{aligned}\Phi(H') - \Phi(H) &= (t(H) + c + 2m(H) - 2c + 4) - (t(H) + 2m(H)) \\ &= 4 - c\end{aligned}$$

$$\widehat{c_i} = c_i + \Phi(H') - \Phi(H) = O(c) + 4 - c = O(1)$$

DELETE

Algoritmo

```
DELETE(H,x)
 1. DECREASE-KEY(H,x,-∞)
 2. EXTRACT-MIN(H)
```

DELETE

Algoritmo

```
DELETE(H,x)
 1. DECREASE-KEY(H,x,-∞)
 2. EXTRACT-MIN(H)
```

- O custo amortizado de um DELETE é dado pela soma dos custos amortizados do DECREASE-KEY e do EXTRACT-MIN, ou seja $O(D(n))$

Limite do grau máximo

Fibonacci

$$F_k = \begin{cases} 0 & se \;\; k = 0 \\ 1 & se \;\; k = 1 \\ F_{k-1} + F_{k-2} & se \;\; k \geq 2 \end{cases}$$

Limite do grau máximo

Fibonacci

$$F_k = \begin{cases} 0 & se\ \ k = 0 \\ 1 & se\ \ k = 1 \\ F_{k-1} + F_{k-2} & se\ \ k \geq 2 \end{cases}$$

k	0	1	2	3	4	5	6	7	8	9	10
F_k	0	1	1	2	3	5	8	13	21	34	55

Limite do grau máximo

Fibonacci

$$F_k = \begin{cases} 0 & se \ k = 0 \\ 1 & se \ k = 1 \\ F_{k-1} + F_{k-2} & se \ k \geq 2 \end{cases}$$

k	0	1	2	3	4	5	6	7	8	9	10
F_k	0	1	1	2	3	5	8	13	21	34	55

Lema 20.2

Para todo inteiro $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

Limite do grau máximo

Prova do Lema 20.2 – por indução fraca em k

Prova do Lema 20.2 – por indução fraca em k

- B.I.: Quando $k = 0$

$$1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 = 1 = F_2$$

Limite do grau máximo

Prova do Lema 20.2 – por indução fraca em k

- B.I.: Quando $k = 0$

$$1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 = 1 = F_2$$

- H.I.: $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$

Limite do grau máximo

Prova do Lema 20.2 – por indução fraca em k

- B.I.: Quando $k = 0$

$$1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 = 1 = F_2$$

- H.I.: $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$
- P.I.:

$$F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) = 1 + \sum_{i=0}^{k} F_i$$

Limite do grau máximo

Lema 20.1

Seja x um nó qualquer em um heap de Fibonacci e $degree[x] = k$. Se $y_1, y_2, \ldots, y_k$ denota os filhos de x na ordem na qual eles foram ligados a x então $degree[y_1] \geq 0$ e $degree[y_i] \geq i - 2$ para $i = 2, 3, \ldots, k$.

Limite do grau máximo

Lema 20.1

Seja x um nó qualquer em um heap de Fibonacci e $degree[x] = k$. Se $y_1, y_2, \ldots, y_k$ denota os filhos de x na ordem na qual eles foram ligados a x então $degree[y_1] \geq 0$ e $degree[y_i] \geq i - 2$ para $i = 2, 3, \ldots, k$.

Prova do Lema 20.1 – direta

Limite do grau máximo

Lema 20.1

Seja x um nó qualquer em um heap de Fibonacci e $degree[x] = k$. Se $y_1, y_2, \ldots, y_k$ denota os filhos de x na ordem na qual eles foram ligados a x então $degree[y_1] \geq 0$ e $degree[y_i] \geq i - 2$ para $i = 2, 3, \ldots, k$.

Prova do Lema 20.1 – direta

- Para $i \geq 2$, note que quando y_i foi ligado a x, $y_1, y_2, \ldots, y_{i-1}$ já eram filhos de x então $degree[x] = i - 1$.

Limite do grau máximo

Lema 20.1

Seja x um nó qualquer em um heap de Fibonacci e $degree[x] = k$. Se $y_1, y_2, \ldots, y_k$ denota os filhos de x na ordem na qual eles foram ligados a x então $degree[y_1] \geq 0$ e $degree[y_i] \geq i - 2$ para $i = 2, 3, \ldots, k$.

Prova do Lema 20.1 – direta

- Para $i \geq 2$, note que quando y_i foi ligado a x, $y_1, y_2, \ldots, y_{i-1}$ já eram filhos de x então $degree[x] = i - 1$.
- $degree[x] = degree[y_i] \Rightarrow degree[y_i] = i - 1$

Limite do grau máximo

Lema 20.1

Seja x um nó qualquer em um heap de Fibonacci e $degree[x] = k$. Se $y_1, y_2, \ldots, y_k$ denota os filhos de x na ordem na qual eles foram ligados a x então $degree[y_1] \geq 0$ e $degree[y_i] \geq i - 2$ para $i = 2, 3, \ldots, k$.

Prova do Lema 20.1 – direta

- Para $i \geq 2$, note que quando y_i foi ligado a x, $y_1, y_2, \ldots, y_{i-1}$ já eram filhos de x então $degree[x] = i - 1$.
- $degree[x] = degree[y_i] \Rightarrow degree[y_i] = i - 1$
- Note que y_i pode perder no máximo 1 filho, caso contrário é cortado de x

Limite do grau máximo

Lema 20.1

Seja x um nó qualquer em um heap de Fibonacci e $degree[x] = k$. Se $y_1, y_2, \ldots, y_k$ denota os filhos de x na ordem na qual eles foram ligados a x então $degree[y_1] \geq 0$ e $degree[y_i] \geq i - 2$ para $i = 2, 3, \ldots, k$.

Prova do Lema 20.1 – direta

- Para $i \geq 2$, note que quando y_i foi ligado a x, $y_1, y_2, \ldots, y_{i-1}$ já eram filhos de x então $degree[x] = i - 1$.
- $degree[x] = degree[y_i] \Rightarrow degree[y_i] = i - 1$
- Note que y_i pode perder no máximo 1 filho, caso contrário é cortado de x
- $degree[y_i] \geq i - 2$

Limite do grau máximo

Lema 20.3

Se x é um nó qualquer de um heap de Fibonacci e $k = \mathit{degree}[x]$ então $\mathit{size}(x) \geq F_{k+2} \geq \phi^k$.

Limite do grau máximo

Lema 20.3

Se x é um nó qualquer de um heap de Fibonacci e $k = degree[x]$ então $size(x) \geq F_{k+2} \geq \phi^k$.

- $size(x)$ denota o número de nós, incluindo x, na sub-árvore onde x é a raiz.

Limite do grau máximo

Lema 20.3

Se x é um nó qualquer de um heap de Fibonacci e $k = degree[x]$ então $size(x) \geq F_{k+2} \geq \phi^k$.

- $size(x)$ denota o número de nós, incluindo x, na sub-árvore onde x é a raiz.
- $\phi = (1 + \sqrt{5})/2$ (razão áurea)

Limite do grau máximo

Lema 20.3

Se x é um nó qualquer de um heap de Fibonacci e $k = degree[x]$ então $size(x) \geq F_{k+2} \geq \phi^k$.

- $size(x)$ denota o número de nós, incluindo x, na sub-árvore onde x é a raiz.
- $\phi = (1 + \sqrt{5})/2$ (razão áurea)
- $F_{k+2} \geq \phi^k$

Prova do Lema 20.3

Limite do grau máximo

Prova do Lema 20.3

- s_k denota o valor mínimo de $size(z)$, tal que $degree[z] = k$. $s_0 = 1$, $s_1 = 2$ e $s_2 = 3$

Limite do grau máximo

Prova do Lema 20.3

- s_k denota o valor mínimo de $size(z)$, tal que $degree[z] = k$. $s_0 = 1$, $s_1 = 2$ e $s_2 = 3$
- $y_1, y_2, \ldots, y_k$ denota os filhos de x na ordem em que foram ligados a ele

Limite do grau máximo

Prova do Lema 20.3

- s_k denota o valor mínimo de $size(z)$, tal que $degree[z] = k$. $s_0 = 1$, $s_1 = 2$ e $s_2 = 3$
- $y_1, y_2, \ldots, y_k$ denota os filhos de x na ordem em que foram ligados a ele
- Segue que

$$size(x) \geq s_k = 2 + \sum_{i=2}^{k} s_{degree[y_i]} \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

Limite do grau máximo

Prova do Lema 20.3

- s_k denota o valor mínimo de $size(z)$, tal que $degree[z] = k$. $s_0 = 1$, $s_1 = 2$ e $s_2 = 3$
- $y_1, y_2, \ldots, y_k$ denota os filhos de x na ordem em que foram ligados a ele
- Segue que

$$size(x) \geq s_k = 2 + \sum_{i=2}^{k} s_{degree[y_i]} \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

- Provaremos agora, por indução forte em k, que $s_k \geq F_{k+2}$ para todo $k \geq 0$

Limite do grau máximo

Prova do Lema 20.3 (continuação)

- Provaremos agora, por indução forte em k, que $s_k \geq F_{k+2}$ para todo $k \geq 0$

Prova do Lema 20.3 (continuação)

- Provaremos agora, por indução forte em k, que $s_k \geq F_{k+2}$ para todo $k \geq 0$
- B.I.: $k = 0 \Rightarrow s_0 = 1 = F_2$. $k = 1 \Rightarrow s_1 = 2 = F_3$

Prova do Lema 20.3 (continuação)

- Provaremos agora, por indução forte em k, que $s_k \geq F_{k+2}$ para todo $k \geq 0$
- B.I.: $k = 0 \Rightarrow s_0 = 1 = F_2$. $k = 1 \Rightarrow s_1 = 2 = F_3$
- H.I.: $s_i \geq F_{i+2}$, para $i = 0, 1, \ldots, k-1$ e $k \geq 2$

Prova do Lema 20.3 (continuação)

- Provaremos agora, por indução forte em k, que $s_k \geq F_{k+2}$ para todo $k \geq 0$
- B.I.: $k = 0 \Rightarrow s_0 = 1 = F_2$. $k = 1 \Rightarrow s_1 = 2 = F_3$
- H.I.: $s_i \geq F_{i+2}$, para $i = 0, 1, \ldots, k-1$ e $k \geq 2$
- P.I.:

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2} \geq \phi^k$$

Limite do grau máximo

Corolário 20.4

O grau máximo $D(n)$ de qualquer nó de um heap de Fibonacci com n elementos é $O(\lg n)$.

Limite do grau máximo

Corolário 20.4

O grau máximo $D(n)$ de qualquer nó de um heap de Fibonacci com n elementos é $O(\lg n)$.

Prova do Corolário 20.4 – direta

Limite do grau máximo

Corolário 20.4

O grau máximo $D(n)$ de qualquer nó de um heap de Fibonacci com n elementos é $O(\lg n)$.

Prova do Corolário 20.4 – direta

- x um nó qualquer e $k = degree[x]$

Limite do grau máximo

Corolário 20.4

O grau máximo $D(n)$ de qualquer nó de um heap de Fibonacci com n elementos é $O(\lg n)$.

Prova do Corolário 20.4 – direta

- x um nó qualquer e $k = degree[x]$
- $n \geq size(x) \geq \phi^k$

Limite do grau máximo

Corolário 20.4

O grau máximo $D(n)$ de qualquer nó de um heap de Fibonacci com n elementos é $O(\lg n)$.

Prova do Corolário 20.4 – direta

- x um nó qualquer e $k = degree[x]$
- $n \geq size(x) \geq \phi^k$
- $\phi^k \leq n \Rightarrow k \leq \log_\phi n$

Limite do grau máximo

Corolário 20.4

O grau máximo $D(n)$ de qualquer nó de um heap de Fibonacci com n elementos é $O(\lg n)$.

Prova do Corolário 20.4 – direta

- x um nó qualquer e $k = degree[x]$
- $n \geq size(x) \geq \phi^k$
- $\phi^k \leq n \Rightarrow k \leq \log_\phi n$
- $D(n) = O(\lg n)$

Conclusão

Conclusão

- Se aplica bem, do ponto de vista teórico, em aplicações onde o número de EXTRACT-MIN e DELETE é pequeno em relação às outras operações. Exemplos de aplicações: cálculo de árvores geradoras mínimas e busca por menores caminhos dada uma origem

Conclusão

- Se aplica bem, do ponto de vista teórico, em aplicações onde o número de EXTRACT-MIN e DELETE é pequeno em relação às outras operações. Exemplos de aplicações: cálculo de árvores geradoras mínimas e busca por menores caminhos dada uma origem
- Não foi projetada para dar um suporte eficiente a operação SEARCH

Conclusão

- Se aplica bem, do ponto de vista teórico, em aplicações onde o número de EXTRACT-MIN e DELETE é pequeno em relação às outras operações. Exemplos de aplicações: cálculo de árvores geradoras mínimas e busca por menores caminhos dada uma origem
- Não foi projetada para dar um suporte eficiente a operação SEARCH
- Do ponto de vista prático, os fatores constantes e a complexidade a tornam pouco desejável para a maioria dos problemas

Conclusão

- Se aplica bem, do ponto de vista teórico, em aplicações onde o número de EXTRACT-MIN e DELETE é pequeno em relação às outras operações. Exemplos de aplicações: cálculo de árvores geradoras mínimas e busca por menores caminhos dada uma origem
- Não foi projetada para dar um suporte eficiente a operação SEARCH
- Do ponto de vista prático, os fatores constantes e a complexidade a tornam pouco desejável para a maioria dos problemas
- Complexidade das operações sobre Heaps de Fibonacci são provadas com análise amortizada

Conclusão

	Heap		
Operação	**Binário** (pior caso)	**Binomial** (pior caso)	**Fibonacci** (amortizado)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$