HEAPS BINÁRIOS

Saulo Queiroz

Contextualização

- Em muitas aplicações um conjunto de entidades desejam consumir um recurso relativamente escasso
- Cada entidade (item) tem uma prioridade para acessar o recurso
- Uma ED com essas características chamamos de Fila de Prioridades

Filas de Prioridades (FP)

- Estruturas de dados comumente empregadas para armazenar dados de forma temporária
  - Diferentemente de de estruturas de dados usuais (e.g. ABPs) não é comum um item ficar armazenado por tempo indeterminado numa FP
-
  -

Filas de Prioridades (FP)

- Estruturas de dados comumente empregadas para armazenar dados de forma temporária
  - Diferentemente de de estruturas de dados usuais (e.g. ABPs) não é comum um item ficar armazenado por tempo indeterminado numa FP
- Muito comum quando precisamos realizar uma computação sobre vários itens, um por vez
  - A vez de cada item depende de sua **prioridade**

FPs: Características Básicas


```
struc item {
  int valorChave; /* mesmo significado das EDs comuns e.g.
  Listas, árvores de busca, hash*/

  int prioridade; /*prioridade do item ser processado
  frente aos demais itens da FP*/
}
```

- Enquanto que nas EDs comuns os procedimentos baseiam-se no campo valorChave, nas FPs eles orientam-se pelo campo prioridade, **cujo valor pode repetir-se pra itens distintos**

FPs: Classificação

- FPs de Máximo
  - Quanto **maior** o valor do campo `prioridade` de um item maior sua prioridade de sair da fila para ser processado
- FPs de Mínimo
  - Quanto **menor** o valor do campo `prioridade` de um item maior sua prioridade de sair da fila para ser processado

Questão instigadora:

- Qual as implicações de empregarmos uma ABP (mesmo auto-balanceada) para implementar uma FP?
  - É possível manter tempos de inserção e remoção logarítmicos?

O *Heap* Binário

Prof. Saulo Queiroz

ADVERTÊNCIA

- Não confundir com a área de memória RAM usada para alocação dinâmica, também chamada de *heap*

Heaps Binários

- Objetivo:
  - Implementar procedimentos básicos de FPs em tempo logarítmico mesmo sob valores de prioridade repetido
- Procedimentos básicos se guiam pelo campo `prioridade` não pelo campo `chave` do item
  - Relaxa suposição de que campo principal tem valor único!
-
-
  -
  -

Heaps Binários

- Objetivo:
  - Implementar procedimentos básicos de FPs em tempo logarítmico mesmo sob valores de prioridade repetido
- Procedimentos básicos se guiam pelo campo `prioridade` não pelo campo `chave` do item
  - Relaxa suposição de que campo principal tem valor único!
- Baseado em **árvores binárias (não de pesquisa!)**
- Propriedades fundamentais:
  - Forma
  - Ordem

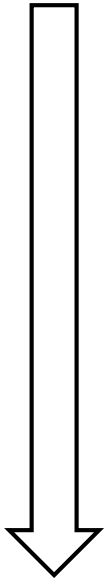
Heaps Binários: Propriedade de Forma

Prof. Saulo Queiroz

Heap: Propriedade de Forma

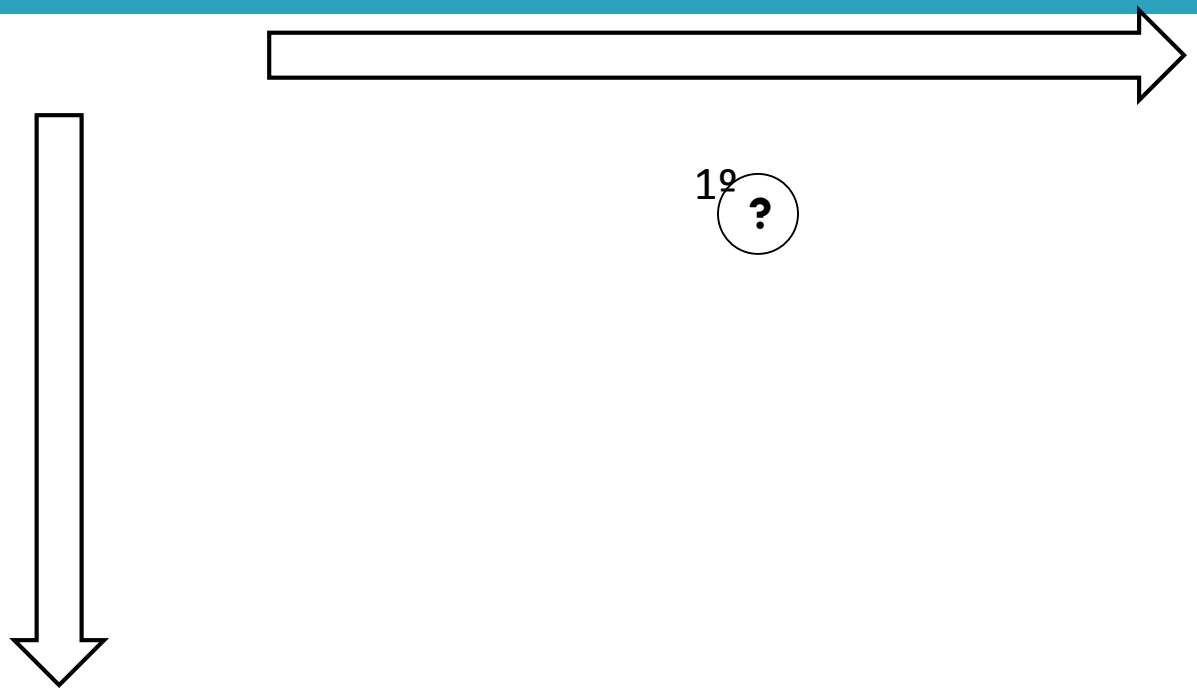
- Os N primeiros itens são inseridos de cima para baixo, da esquerda para a direita na árvore!
  - Ideia: Manter árvore (quase-)completa visando $O(\log_2 N)$
  - O valor dos itens será tratado na propriedade de ordem, sem quebrar a forma "completa" do *heap*
- Como fica a forma ao longo da inserção de 4 itens?

Heap: Propriedade de Forma



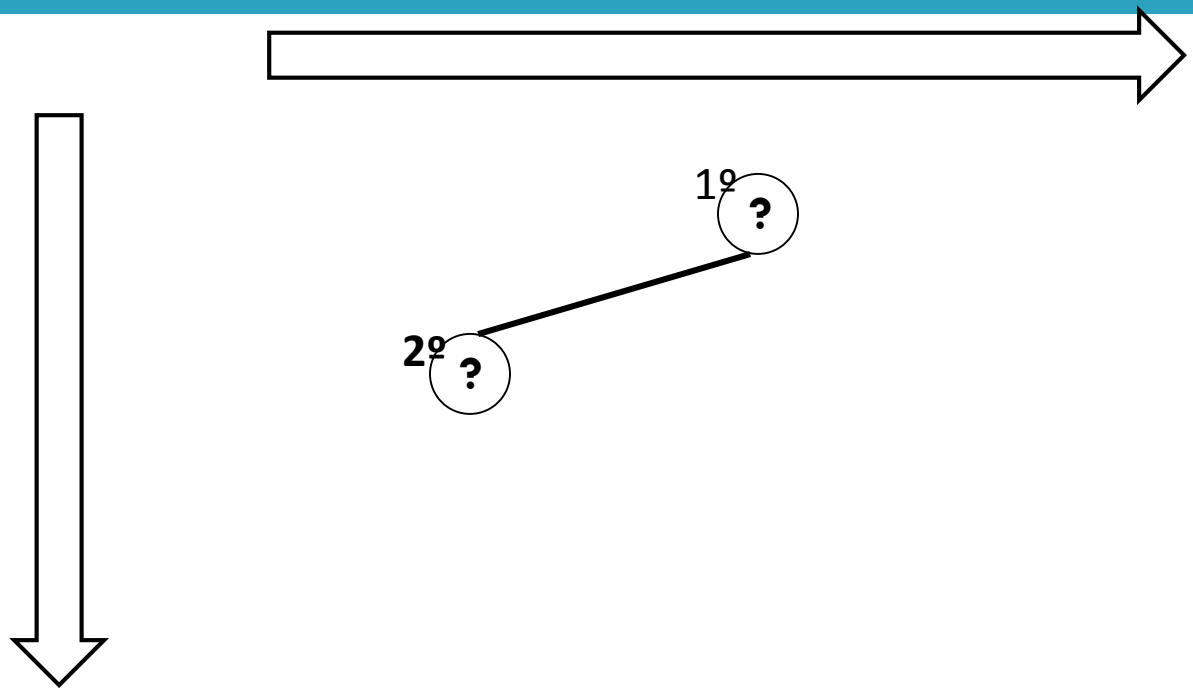
STATUS ATUAL:
COMPLETA

Heap: Propriedade de Forma



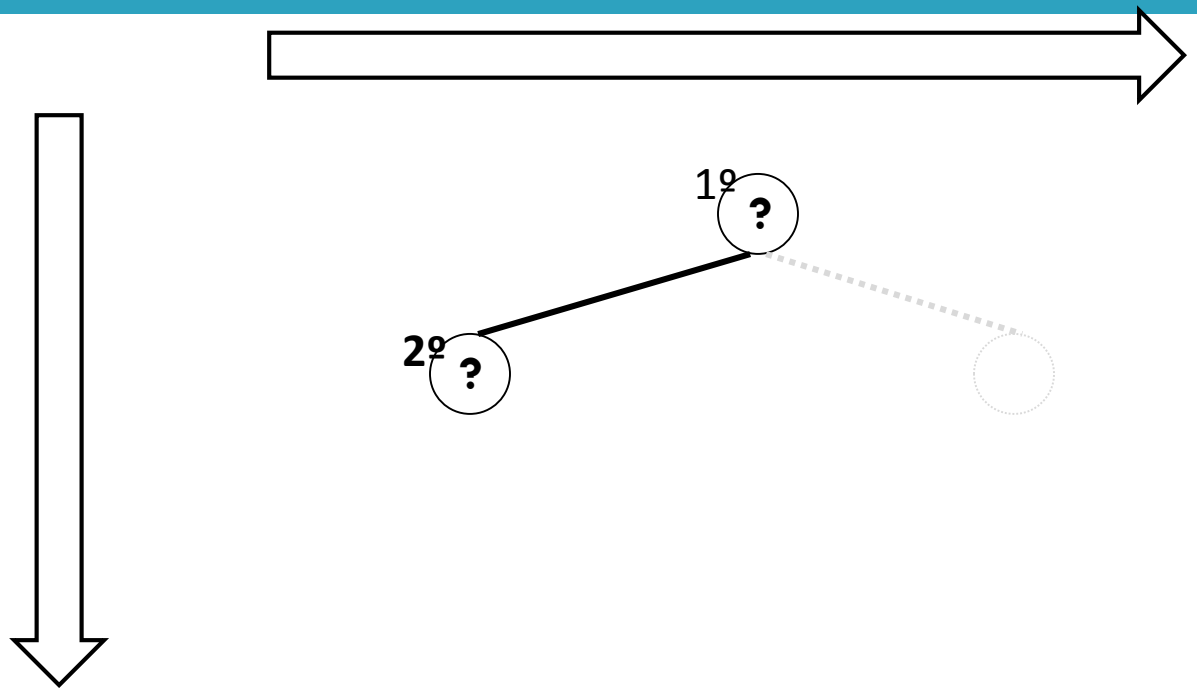
STATUS ATUAL:
COMPLETA

Heap: Propriedade de Forma



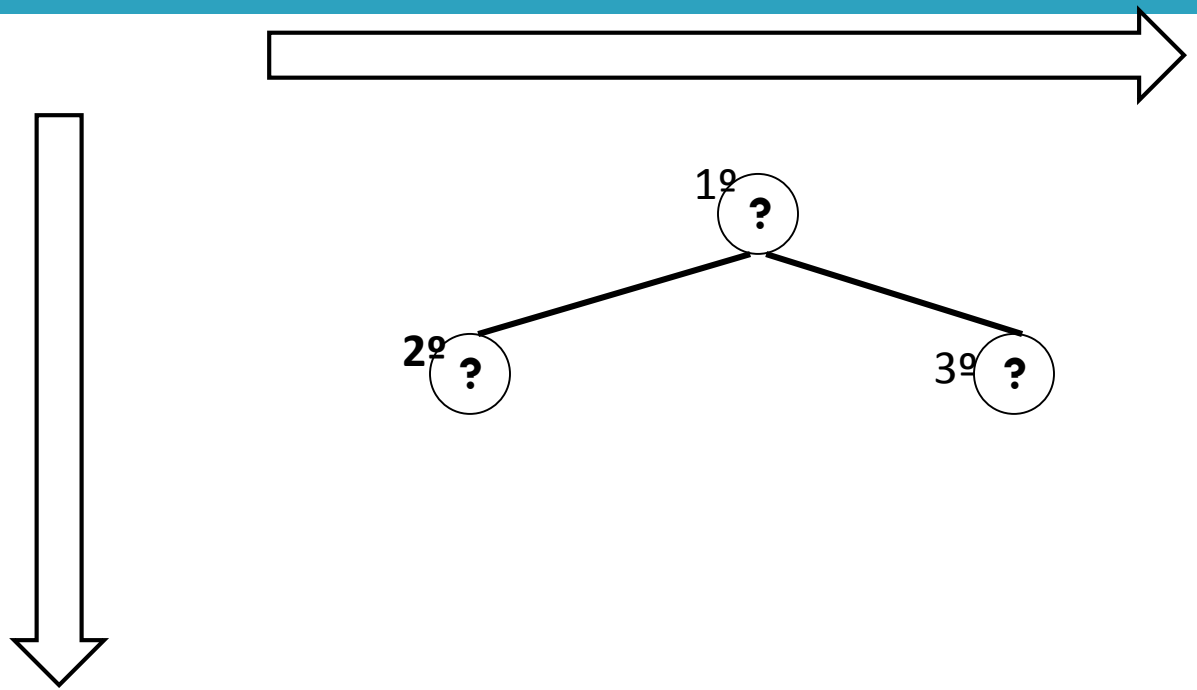
STATUS ATUAL:
(QUASE-)COMPLETA

Heap: Propriedade de Forma



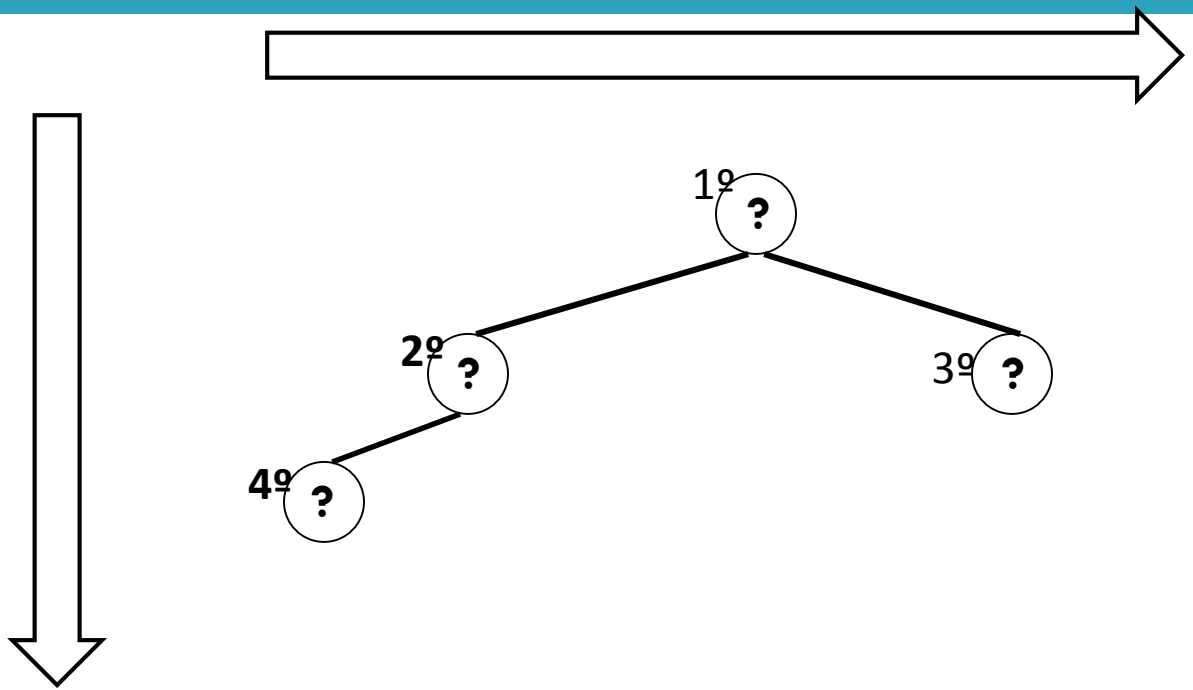
STATUS ATUAL:
(QUASE-)COMPLETA

Heap: Propriedade de Forma



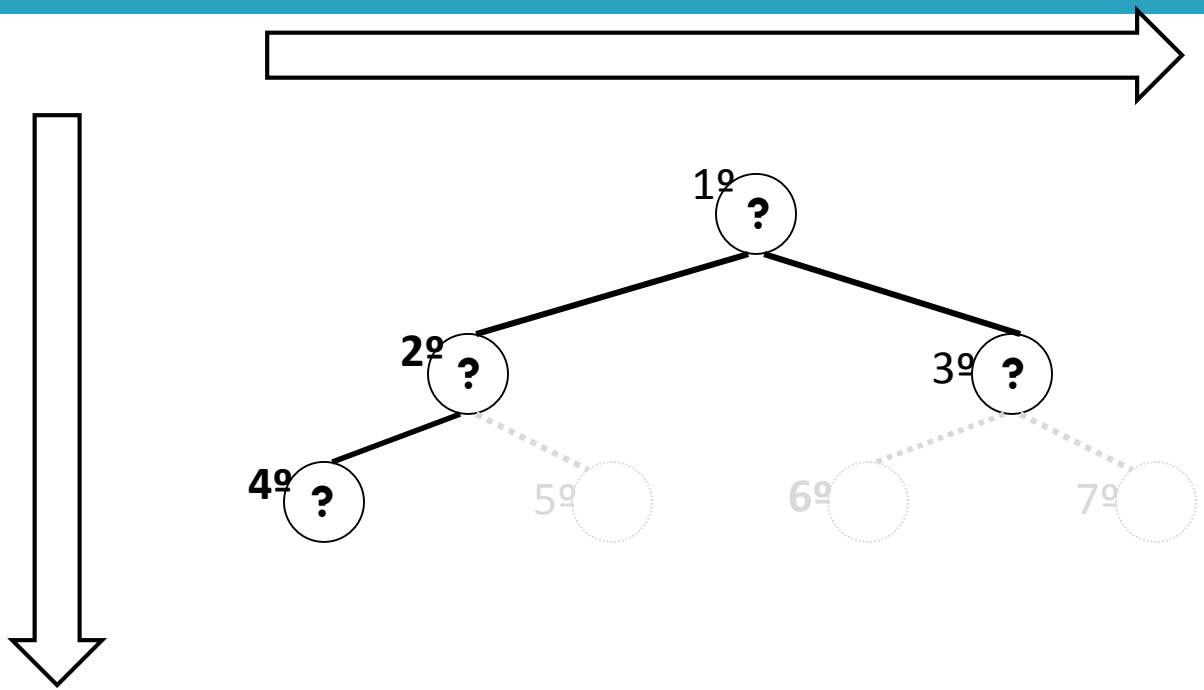
STATUS ATUAL:
COMPLETA

Heap: Propriedade de Forma



STATUS ATUAL:
(QUASE-)COMPLETA

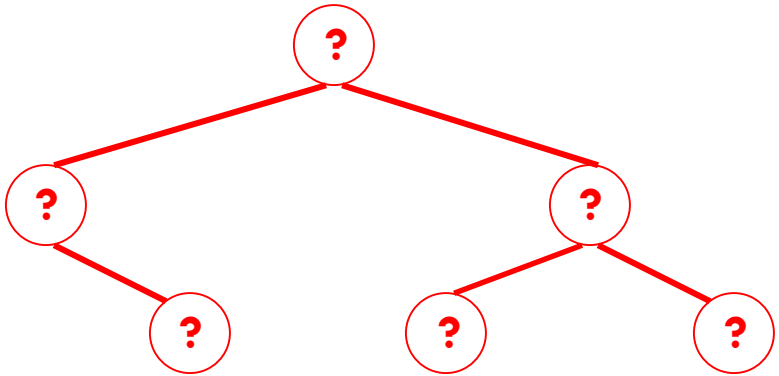
Heap: Propriedade de Forma

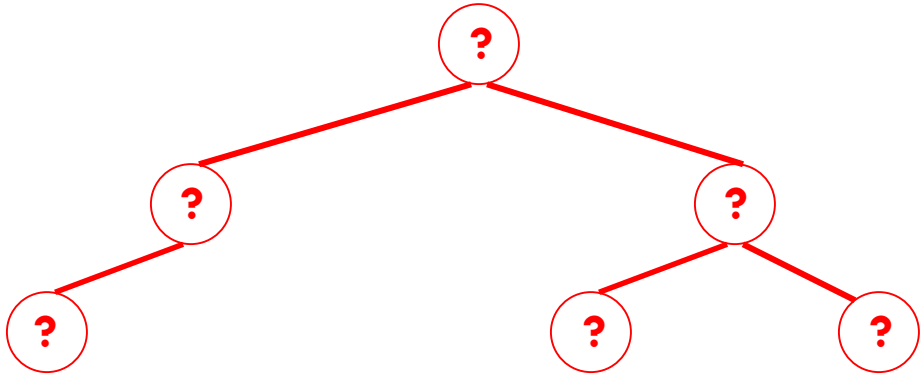


STATUS ATUAL:
(QUASE-)COMPLETA

Exemplos de **NÃO** *Heaps* Binários

- Violou propriedade de forma, nem interessa a propriedade de ordem!



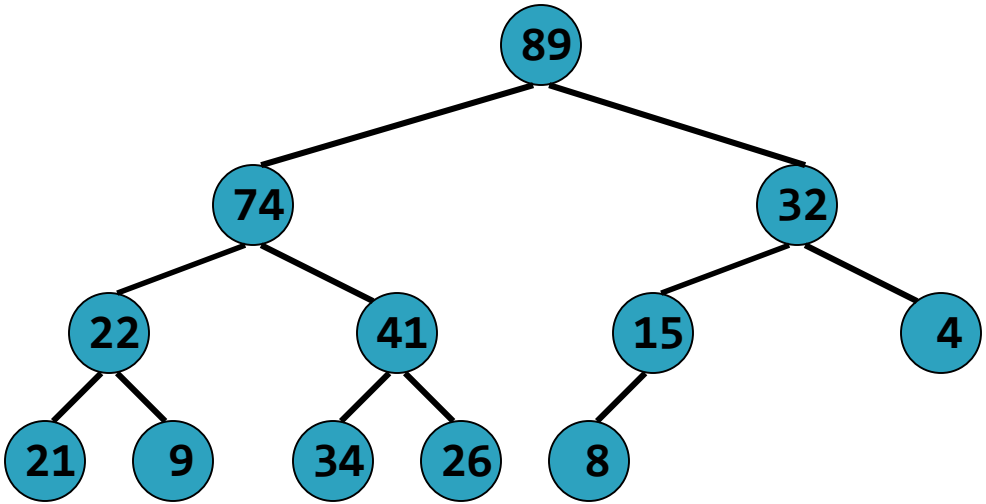


Prop. de Forma e Relação com Vetor

- Uma árvore binária pode ser representada por um vetor seguindo a seguinte regra:
- O raíz é colocado na primeira posição
  - Índice `0` no caso da linguagem C
- Se um nó está no índice i, então
  - O filho esquerdo está 2i (`2*i+1` em C)
  - O filho direito está em 2i+1 (`2*i+2` em C)

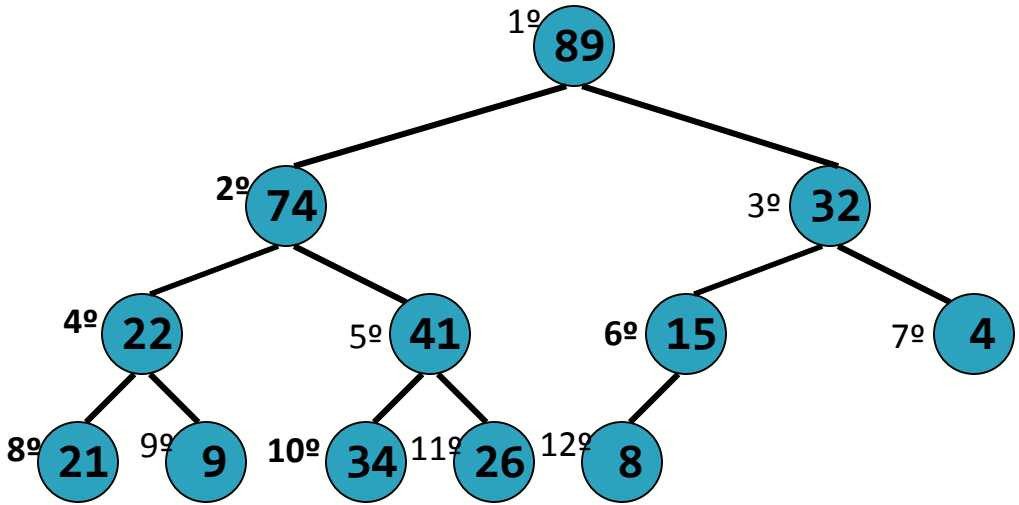
Prop. de Forma e Relação com Vetor

89	74	32	22	41	15	4	21	9	34	26	8
0	1	2	3	4	5	6	7	8	9	10	11



Prop. de Forma e Relação com Vetor

89	74	32	22	41	15	4	21	9	34	26	8
0	1	2	3	4	5	6	7	8	9	10	11
1º	2º	3º	4º	5º	6º	7º	8º	9º	10º	11º	12º



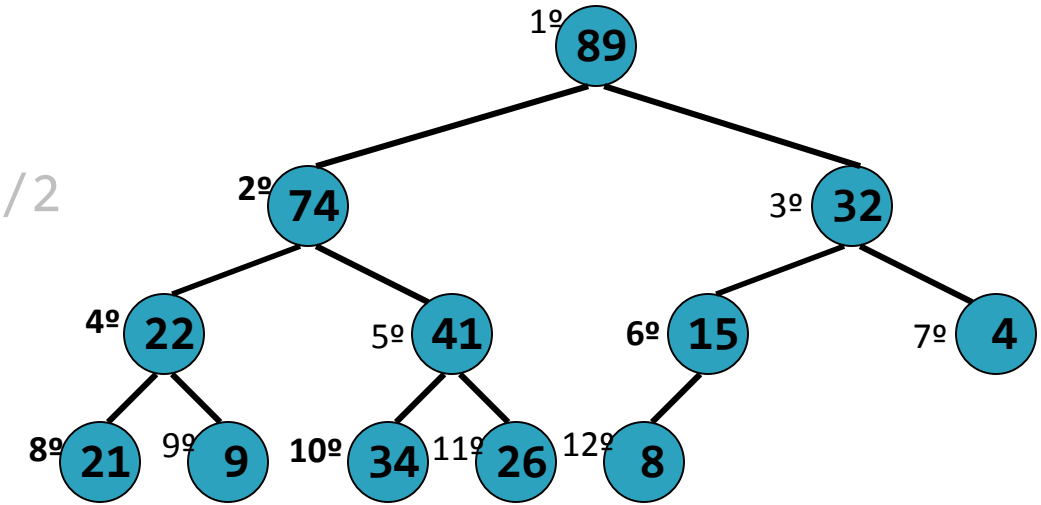
Prop. de Forma e Relação com Vetor

89	74	32	22	41	15	4	21	9	34	26	8
0	1	2	3	4	5	6	7	8	9	10	11
1º	2º	3º	4º	5º	6º	7º	8º	9º	10º	11º	12º

Note que:

Pai(i) = i/2 ou

Pai(i)=(i-1)/2 (em C)



Heaps Binários: Propriedade de Ordem

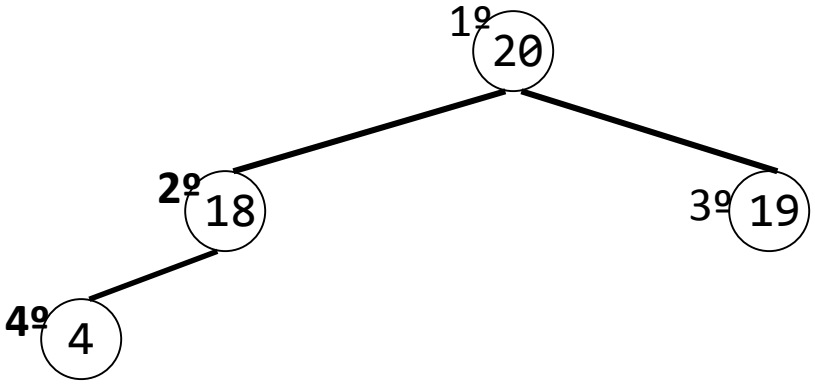
Prof. Saulo Queiroz

Heap: Propriedade de Ordem

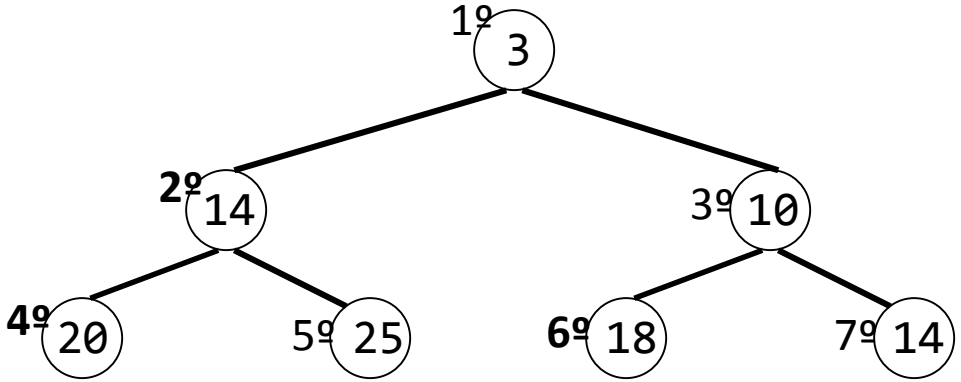
- O nó raiz deve sempre ter prioridade maior que de seus filhos!
  - *Heap* máximo: valor do pai sempre **maior** que dos filhos
  - *Heap* de mínimo: valor do pai sempre **menor** que dos filhos

Heap: Propriedade de Ordem

HEAP BINÁRIO DE MÁXIMO VÁLIDO



HEAP BINÁRIO DE MÍNIMO VÁLIDO



Heaps: Procedimentos Básicos

- `TipoItem ExtraiMinimo(h, n);`
  - Devolve item de melhor prioridade do heap `h` com `n` itens
  - Nome alternativo: `ExtraiMaximo(...)` (*heap* de máximo)
-
  -
-
  -
  -

Heaps: Procedimentos Básicos

- `TipoItem ExtraiMinimo(h, n);`
  - Devolve item de melhor prioridade do heap `h` com `n` itens
  - Nome alternativo: `ExtraiMaximo(...)` (*heap* de máximo)
- `void Insere(h, n, it, pri);`
  - Insere item `it` com prioridade `pri` no *heap* `h` de `n` itens
-
  -
  -

Heaps: Procedimentos Básicos

- `TipoItem ExtraiMinimo(h, n);`
  - Devolve item de melhor prioridade do heap `h` com `n` itens
  - Nome alternativo: `ExtraiMaximo(...)` (*heap* de máximo)
- `void Insere(h, n, it, pri);`
  - Insere item `it` com prioridade `pri` no *heap* `h` de `n` itens
- `void ConstroiHeap(v, n);`
  - Em geral consiste em transformar um vetor em um *heap*
  - No caso de árvores, a construção usualmente resulta de sucessivas inserções

Heaps: Procedimentos Básicos

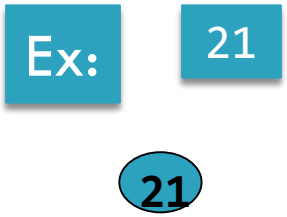
- `void MelhoraPrioridade(h, n, it, novaprio);`
  - Outros nomes: `decrementaChave`, `aumentaChave`
  - Atualiza a prioridade do item `it` para `novaprio` no heap `h` de `n` itens
  - Em vetores, `it` é índice do item
  - Em *árvores binárias,* `it` é o ponteiro para o item
    - Em geral, obtido pela estrutura de dados padrão que guarda os itens

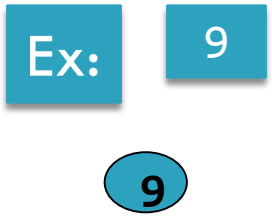
Procedimento auxiliar heapify

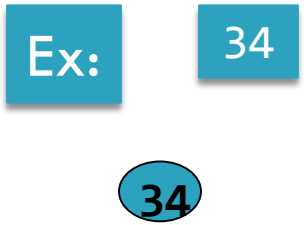
- **Entrada**: *heaps* binários h1 e h2 com n1 e n2 itens, respectivamente.
- Saída (**efeito**): heap h3 com n1+n2 itens
- Muito útil para auxiliar a função ConstroiHeap()
  - Na prática, h1 e h2 serão partes de um mesmo vetor dado de entrada com n=n1+n2 itens

Procedimento auxiliar heapify

- **Instâncias triviais**
  - Se um dos *heaps* for vazio a saída é o *heap* não vazio
  - Ou seja, se o vetor a ser transformado é unitário!

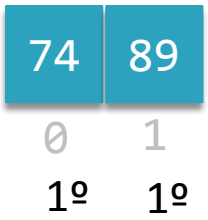


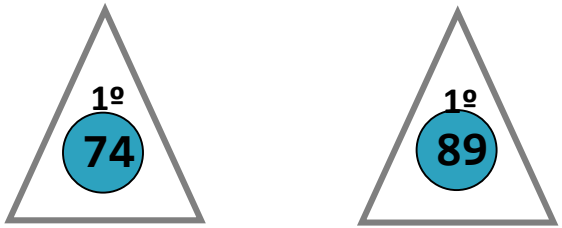




Procedimento auxiliar heapify

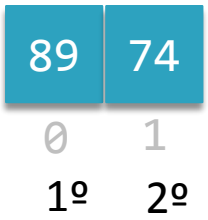
- **Instâncias triviais**
  - Se ambos os *heaps* tem 1 item cada, basta ver o maior!
  - Ou seja, se o vetor a ser transformado tem dois itens

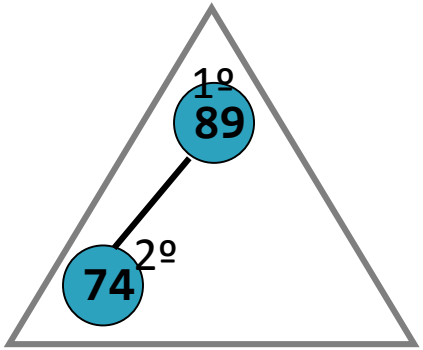




Procedimento auxiliar heapify

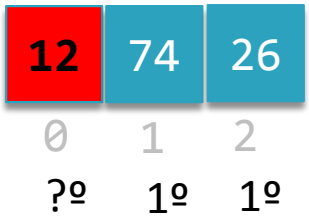
- **Instâncias triviais**
  - Se ambos os *heaps* tem 1 item cada, basta ver o maior!
  - Ou seja, se o vetor a ser transformado tem dois itens

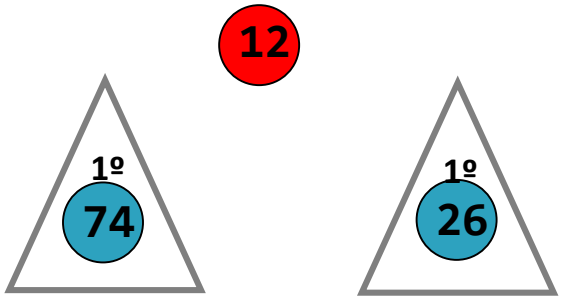




Procedimento auxiliar heapify

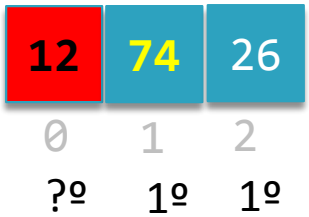
- **Instâncias triviais**
  - Vetor de entrada com n=3 itens
  - Embora o todo possa não ser um *heap*, os dois últimos sempre serão!

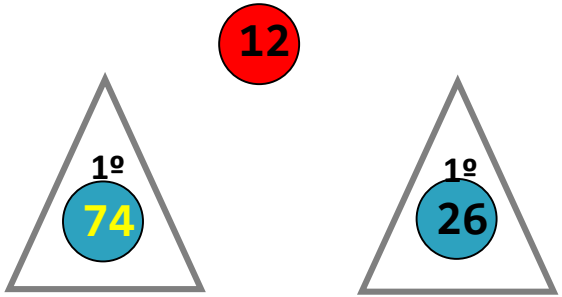




Procedimento auxiliar heapify

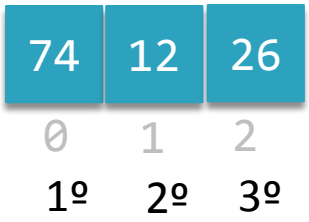
- **Instâncias triviais**
  - Vetor de entrada com n=3 itens
  - Embora o todo possa não ser um *heap*, os dois últimos sempre serão! Compara o maior filho com o pai

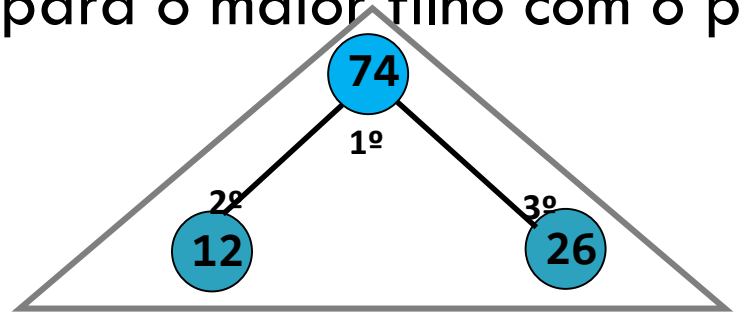




Procedimento auxiliar heapify

- **Instâncias triviais**
  - Vetor de entrada com n=3 itens
  - Embora o todo possa não ser um *heap*, os dois últimos sempre serão! Compara o maior filho com o pai



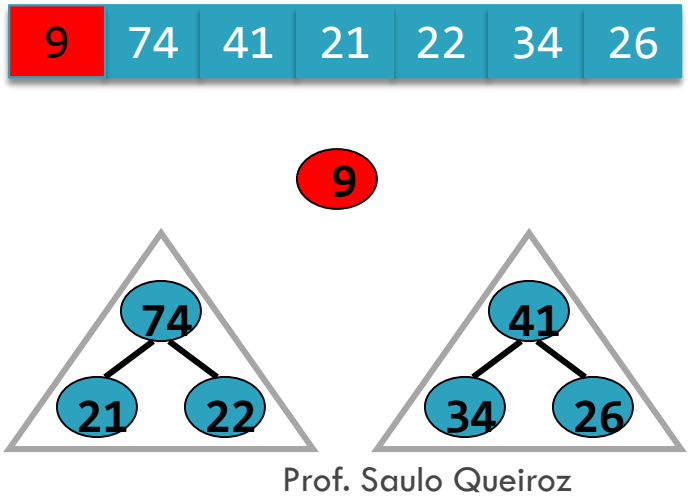


Procedimento `ConstroiHeap()`

- A princípio, não podemos usar `heapify` para transformar um vetor qualquer em um *heap* binário. Por que?

Procedimento ConstroiHeap()

- A princípio, não podemos usar `heapify` para transformar um vetor qualquer em um *heap* binário. Por que? **Não se pode assumir que os filhos do raíz são *heaps* sem si mesmos!**

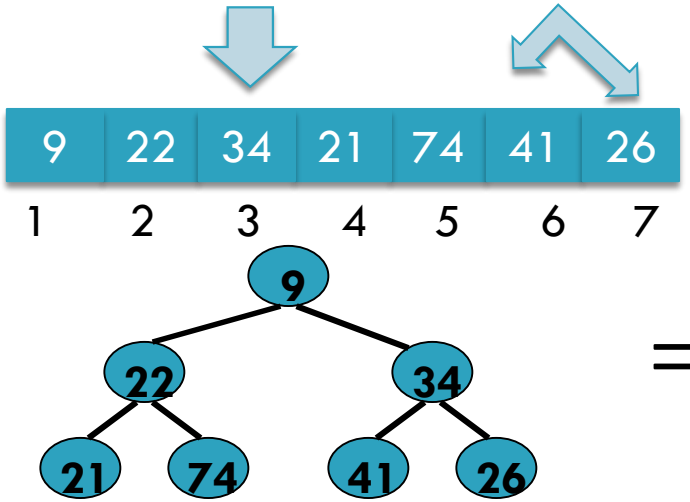


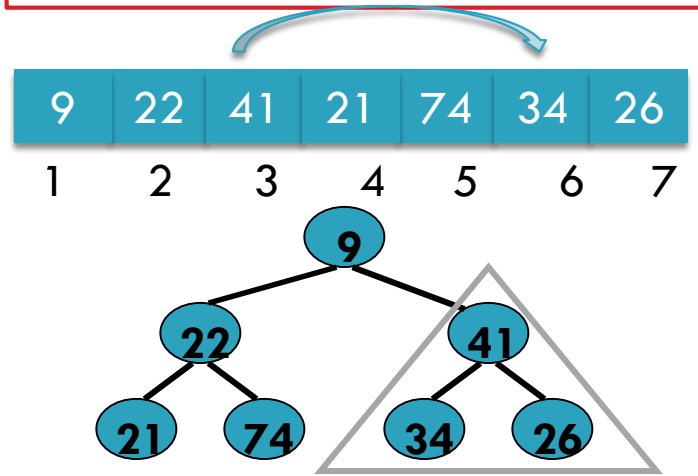
Procedimento `ConstroiHeap()`

- Porém, se notarmos que o último item é sempre um heap em si mesmo, podemos aplicar sucessivas operações `heapify` a partir do pai do último!
  - Se há `n` itens, o pai do último está em `n/2`
    - `(n-1)/2` (em C)

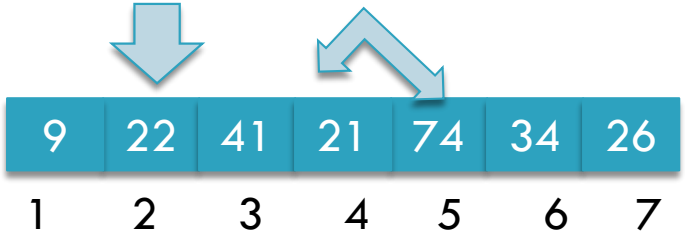
Construindo um Heap *Bottom-up*

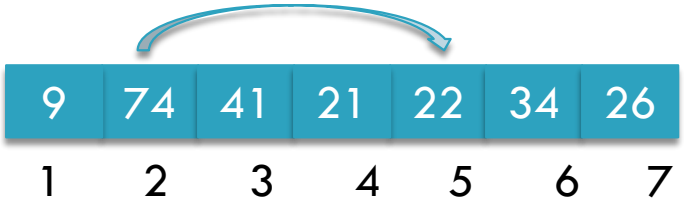
Verifica qual o ***maior filho***;
Se o **pai** é ***menor*** que maior filho,
Então, troca de posição.

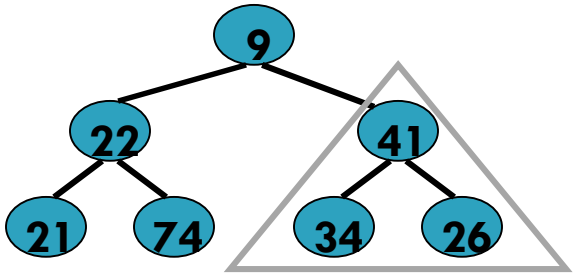




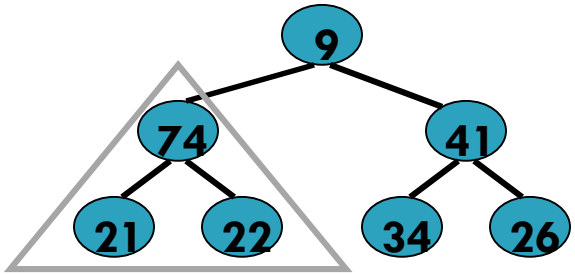
Construindo um Heap *Bottom-up*



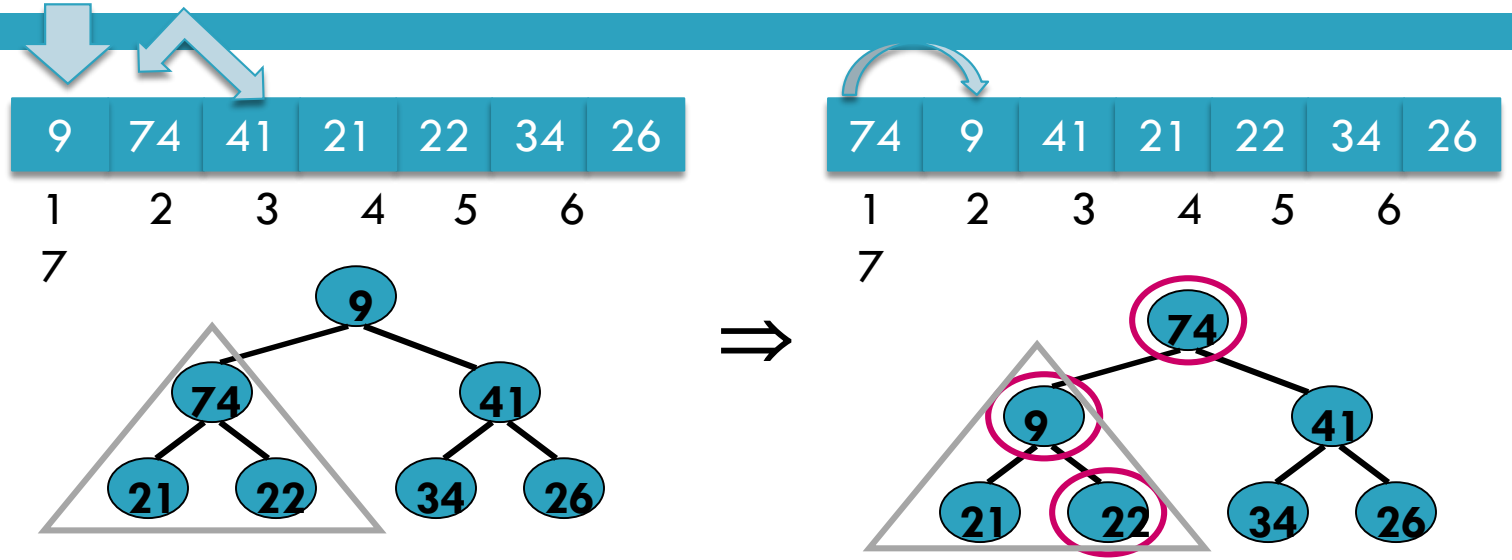




⇒

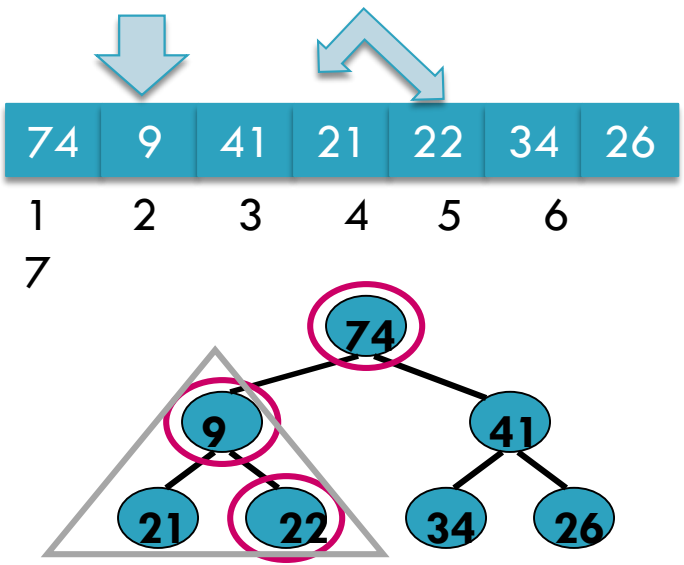


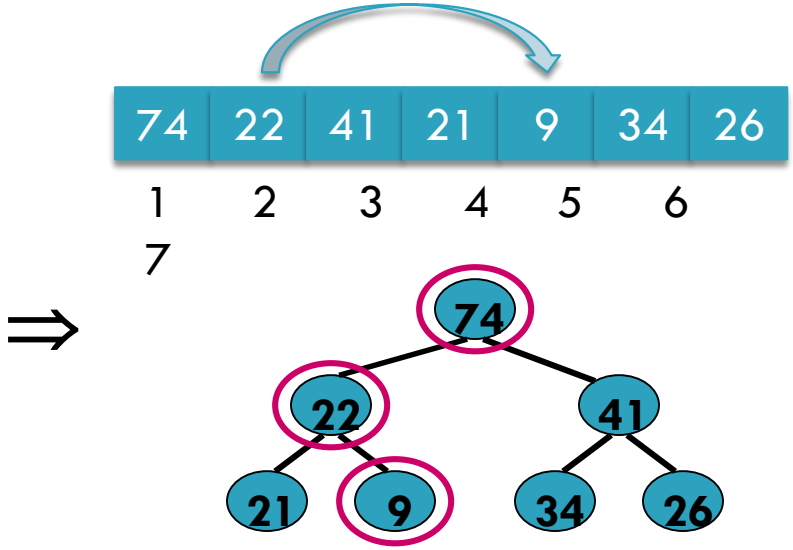
Construindo um Heap *Bottom-up*



- Precisa "***voltar***" e arrumar (**heapify**) a sub-árvore à *esquerda*.

Construindo um Heap *Bottom-up*





Remoção em Heaps Binários (árvore)

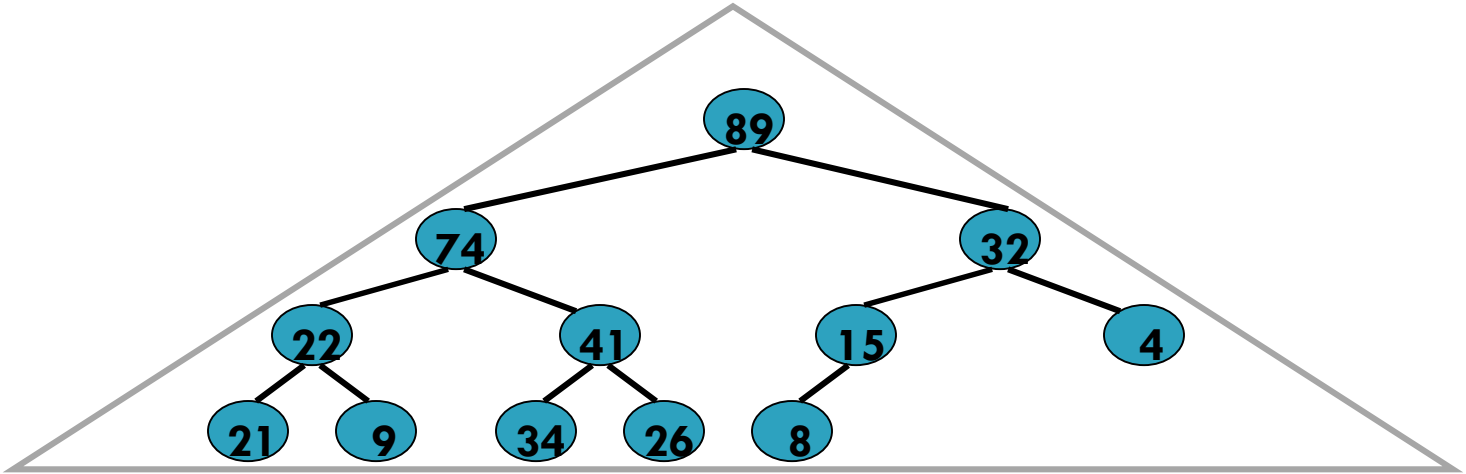
Prof. Saulo Queiroz

Remoção em *Heaps*

devolve dados do raíz, copia último pro raiz, libera espaço do último. Aplica sucessivos `heapify` para re-arranjar *heap*

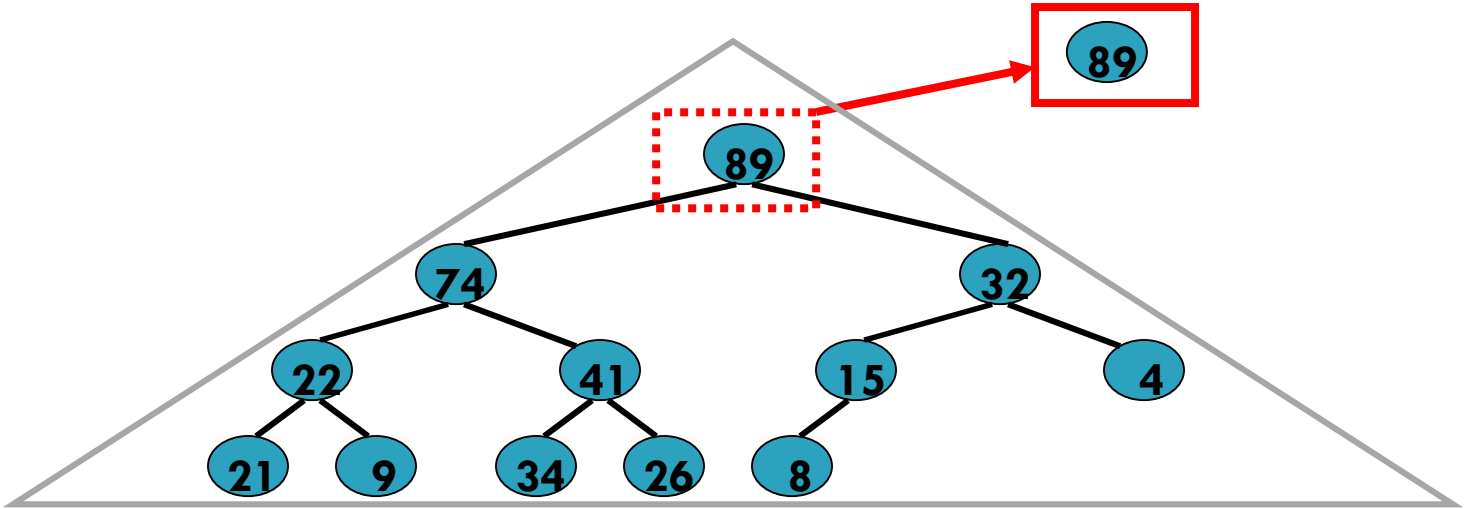
Remoção em *Heaps*

devolve dados do raíz, copia último pro raiz, libera espaço do último. Aplica sucessivos `heapify` para re-arranjar *heap*



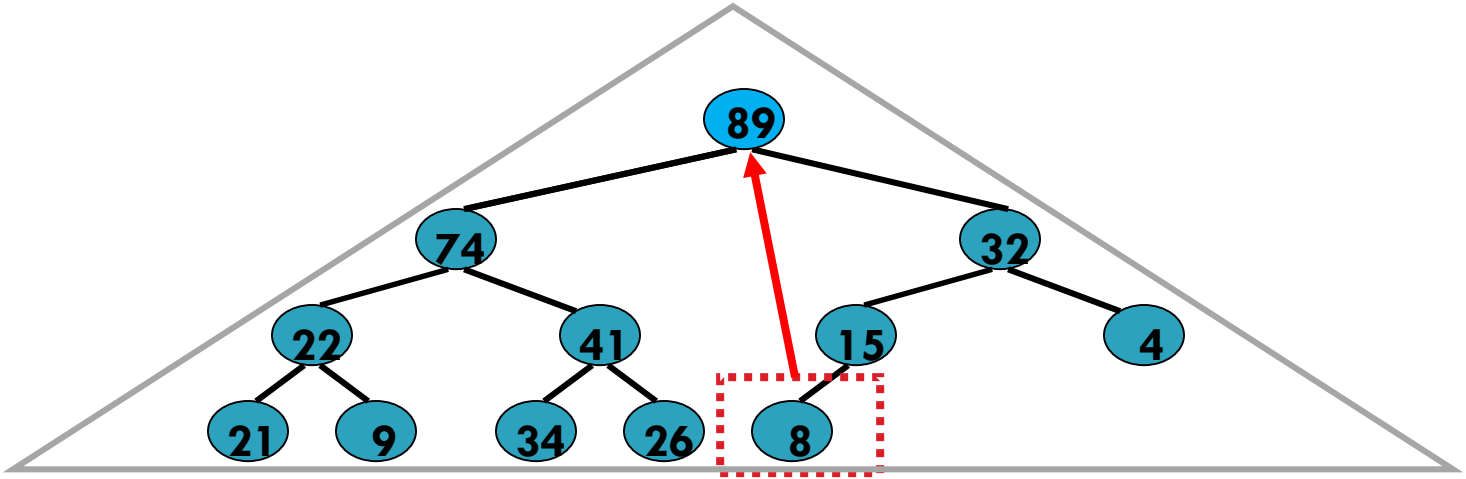
Remoção em *Heaps*

devolve dados do raíz, copia último pro raiz, libera espaço do último. Aplica sucessivos `heapify` para re-arranjar *heap*



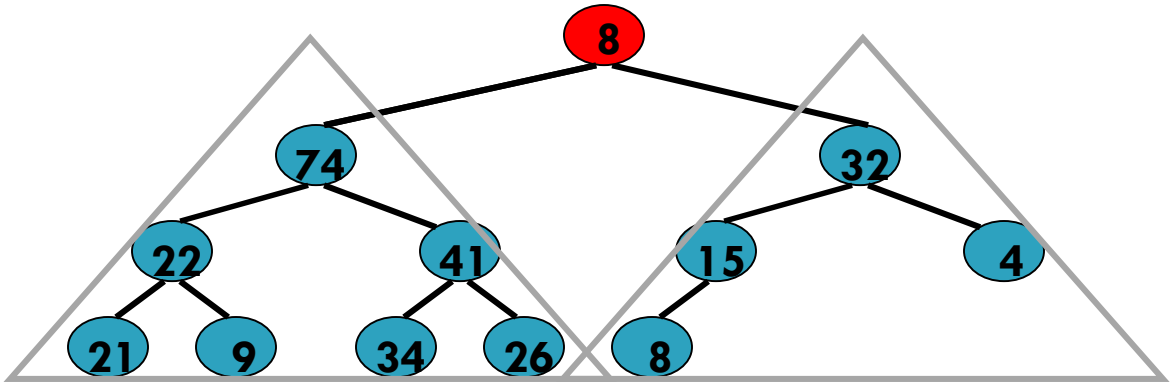
Remoção em *Heaps*

devolve dados do raíz, **copia último pro raiz,** libera espaço do último. Aplica sucessivos `heapify` para re-arranjar *heap*



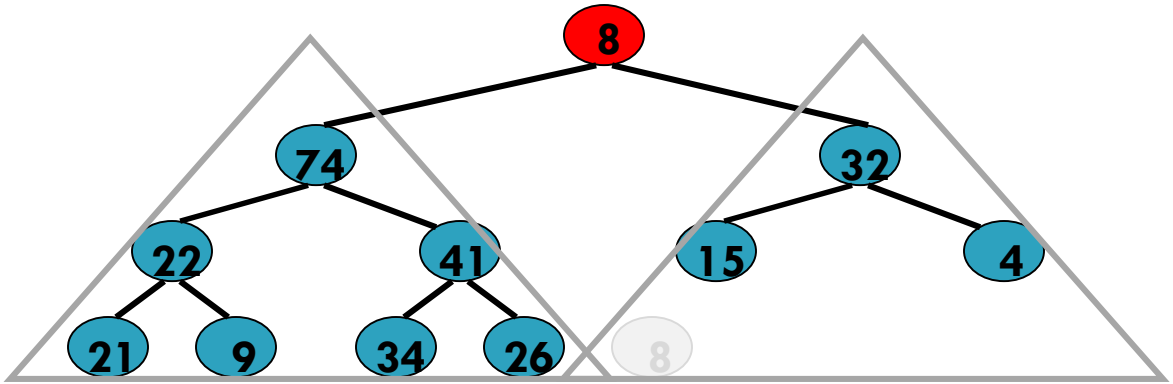
Remoção em *Heaps*

devolve dados do raíz, **copia último pro raiz,** libera espaço do último. Aplica sucessivos `heapify` para re-arranjar *heap*



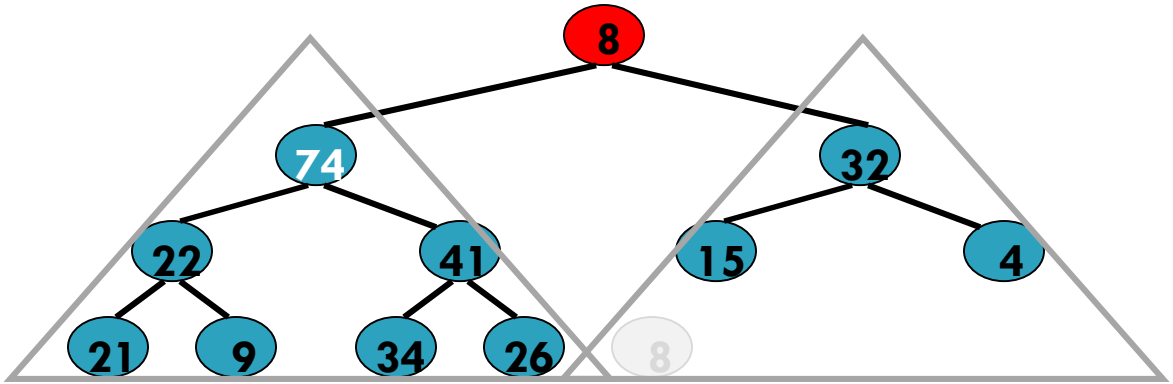
Remoção em Heaps

devolve dados do raíz, copia último pro raiz, **libera espaço do último**. Aplica sucessivos `heapify` para re-arranjar *heap*



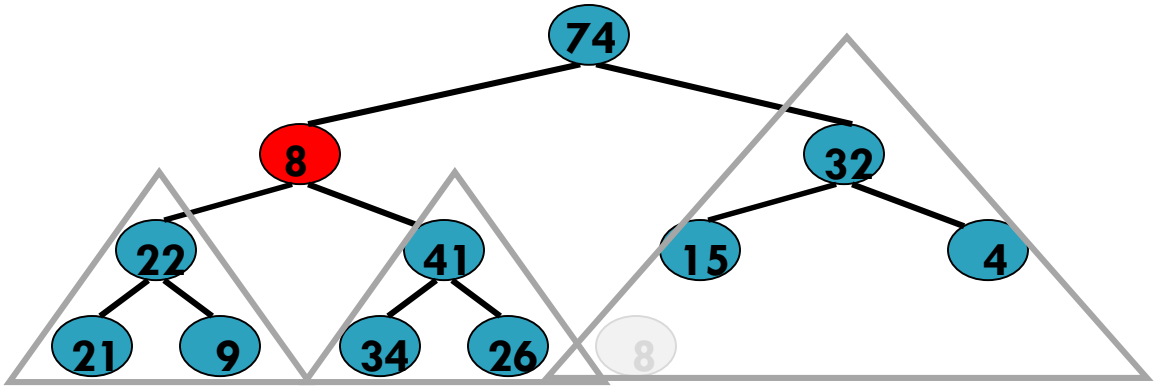
Remoção em Heaps

devolve dados do raíz, copia último pro raiz, libera espaço do último. **Aplica sucessivos heapify para re-arranjar *heap***



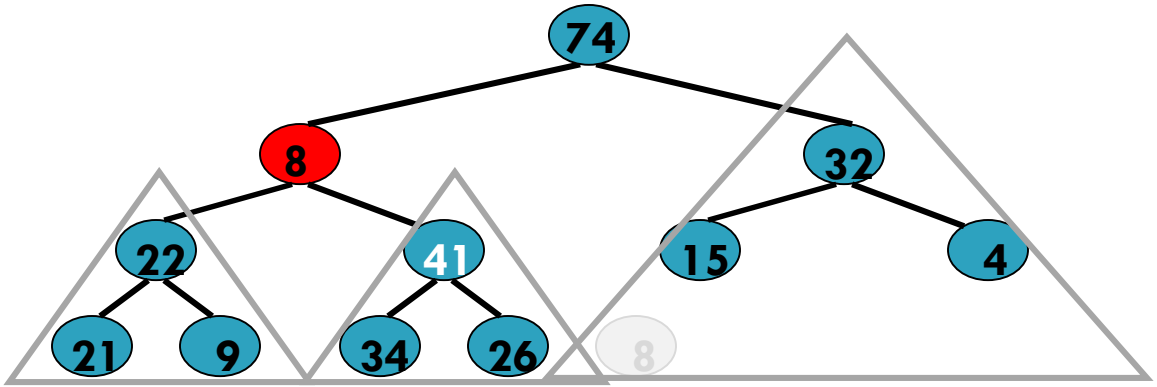
Remoção em Heaps

devolve dados do raíz, copia último pro raiz, libera espaço do último. **Aplica sucessivos heapify para re-arranjar *heap***



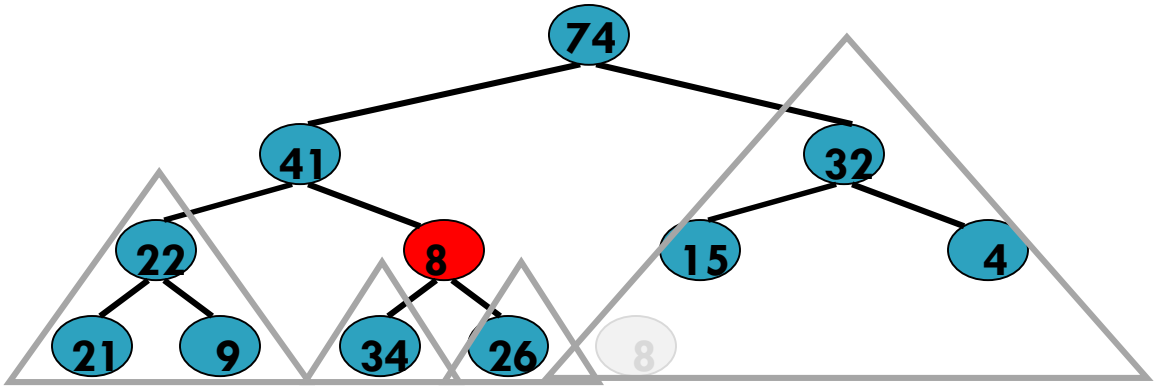
Remoção em Heaps

devolve dados do raíz, copia último pro raiz, libera espaço do último. **Aplica sucessivos `heapify` para re-arranjar *heap***



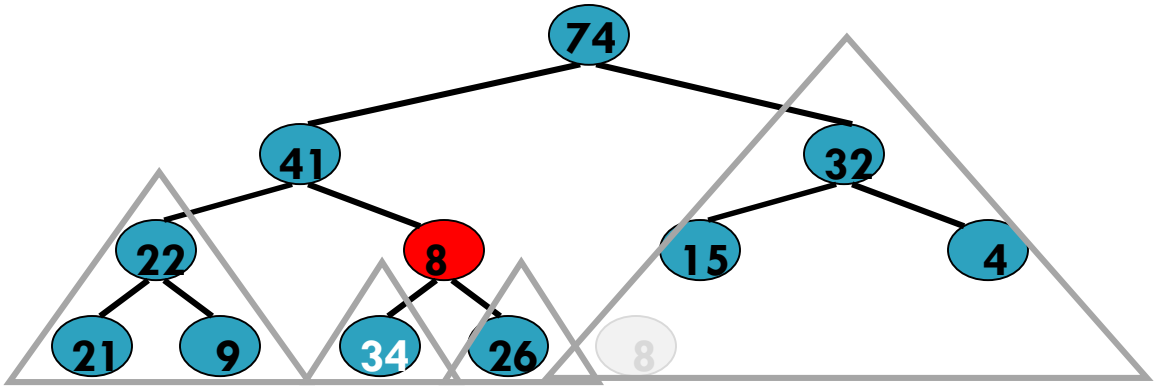
Remoção em Heaps

devolve dados do raíz, copia último pro raiz, libera espaço do último. **Aplica sucessivos `heapify` para re-arranjar *heap***



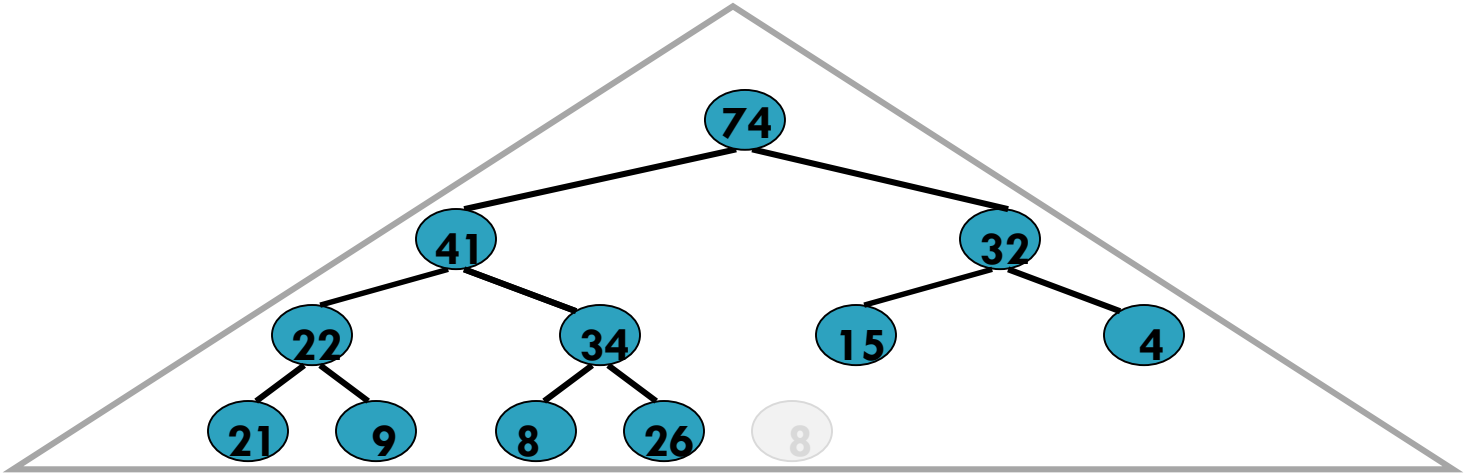
Remoção em Heaps

devolve dados do raíz, copia último pro raiz, libera espaço do último. **Aplica sucessivos `heapify` para re-arranjar *heap***



Remoção em Heaps

devolve dados do raíz, copia último pro raiz, libera espaço do último. **Aplica sucessivos `heapify` para re-arranjar *heap***



Inserção em heaps com árvores binárias

Prof. Saulo Queiroz

Inserção em Heaps

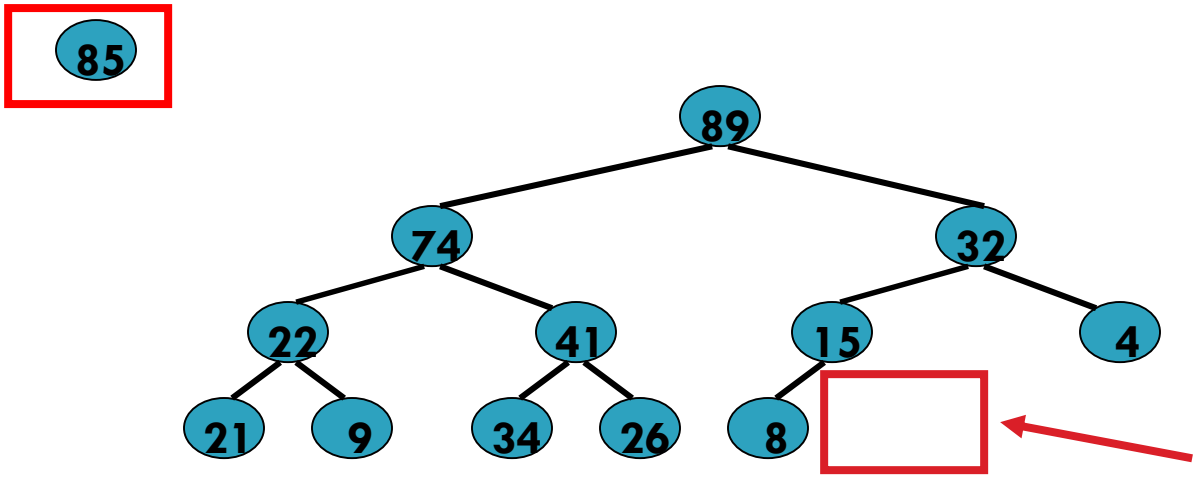
Insere após a última posição do *heap*.

Aplica sucessivos heapify até a raiz

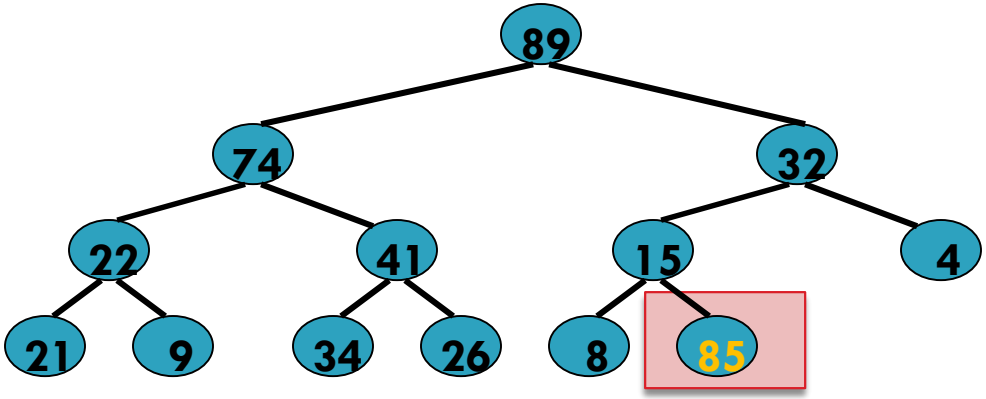
Inserção em Heaps

Insere após a última posição do *heap*.

Aplica sucessivos heapify até a raiz

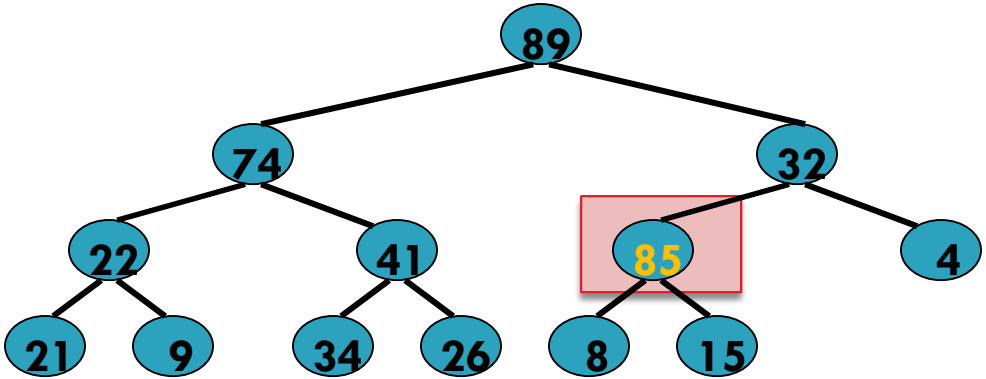


Inserção em Heaps

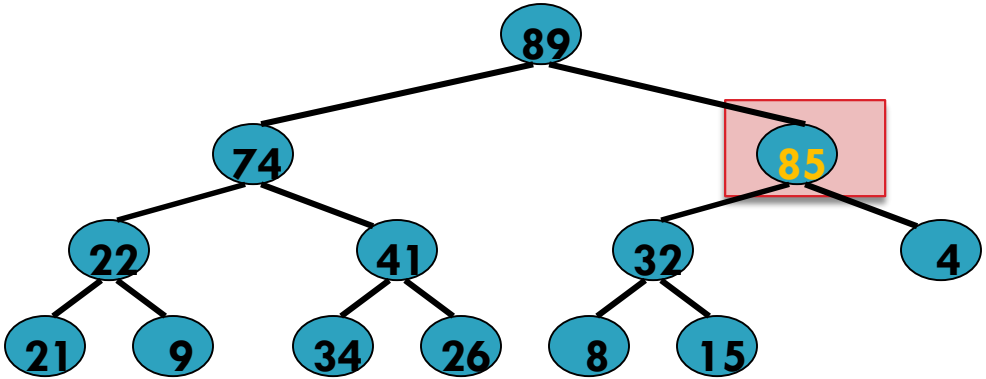


Inserção em Heaps

Depois, basta corrigir a relação do novo elemento com os seus ancestrais.



Inserção em Heaps

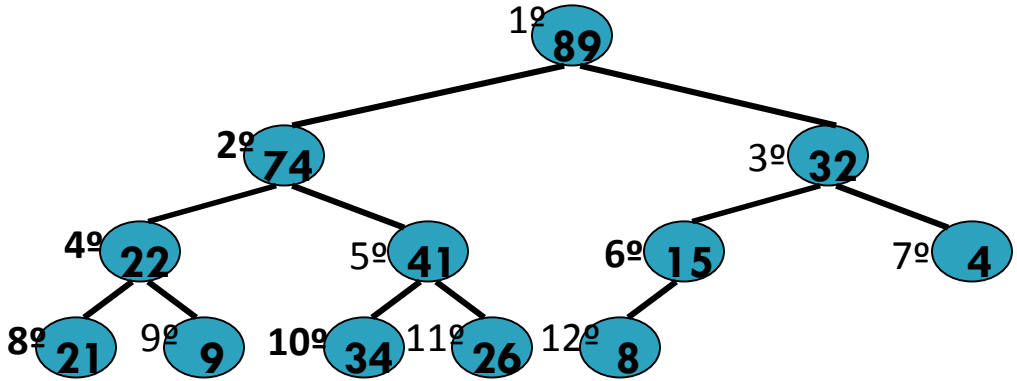


Algoritmo para Achar último em tempo logarítmico

Prof. Saulo Queiroz

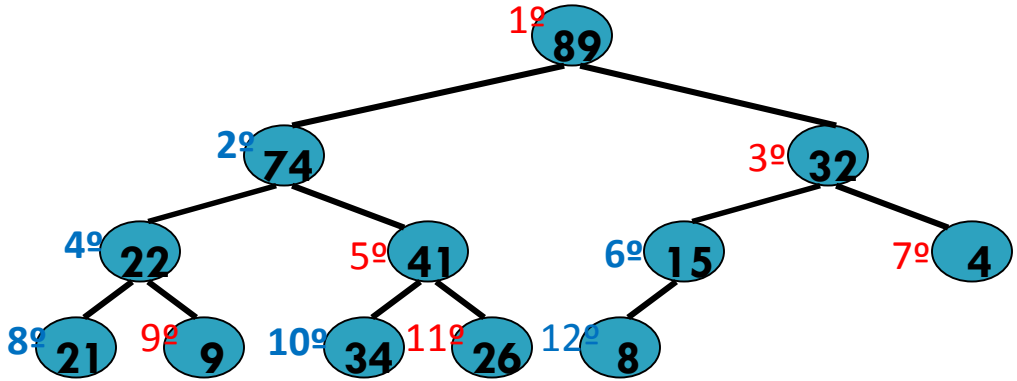
Algoritmo para achar última posição

- Observações-chaves:
  - Todo nó em posição par é filho esquerdo
  - Todo nó em posição impar é filho direito



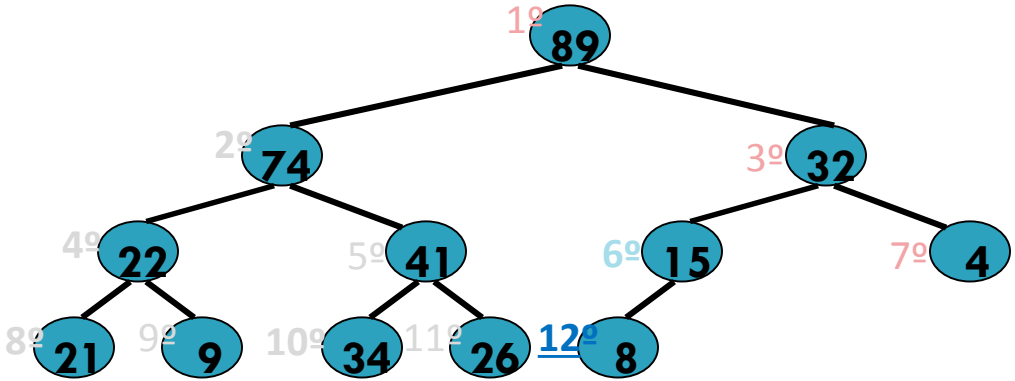
Algoritmo para achar última posição

- Observações-chaves:
  - Todo nó em posição par é filho esquerdo
  - Todo nó em posição impar é filho direito



Algoritmo para achar última posição

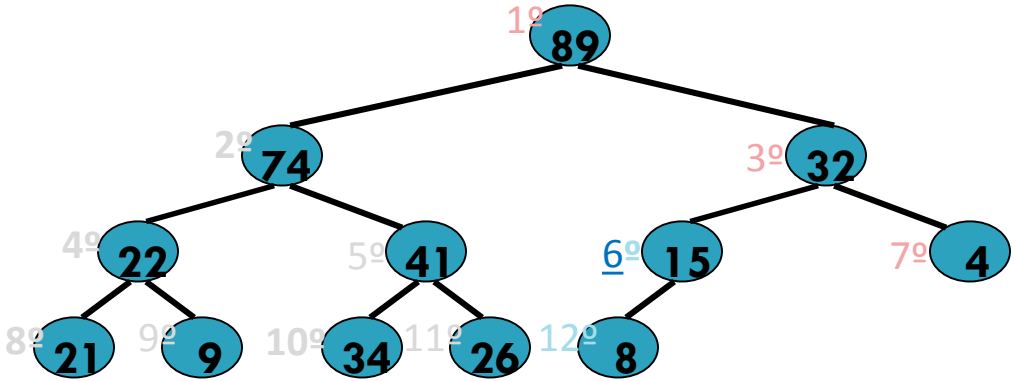
- Dado que N=12 é par, último está à esquerda do pai



esquerda

Algoritmo para achar última posição

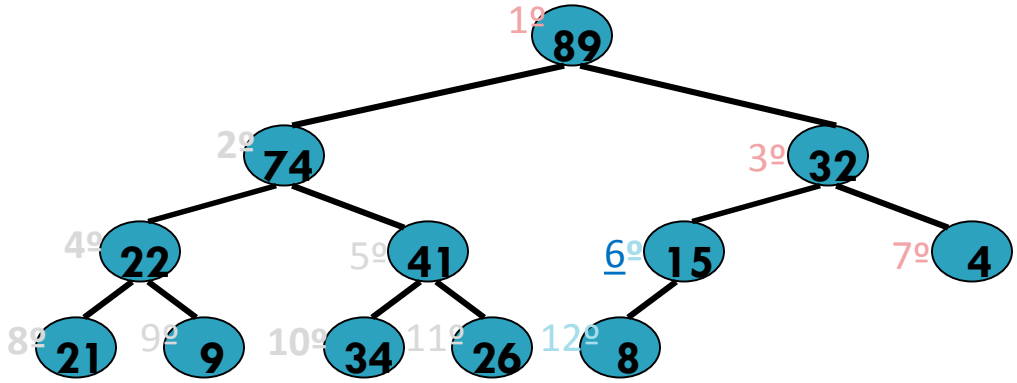
- Dado que N=12 é par, último está à esquerda do pai
  - Sabemos que o pai é o 12/2=6° item



esquerda

Algoritmo para achar última posição

- Agora N=6, também é par. Logo está a esquerda

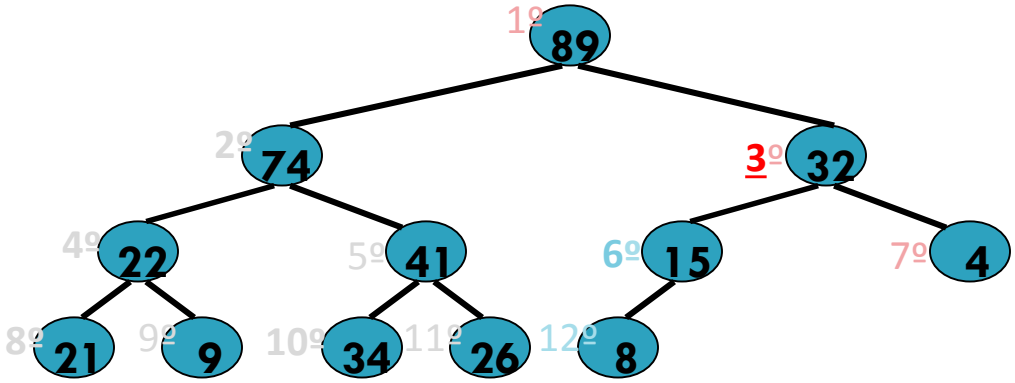


esquerda

esquerda

Algoritmo para achar última posição

- Agora N=6, também é par. Logo está a esquerda
  - O pai do 6° é o 6/2=3° item

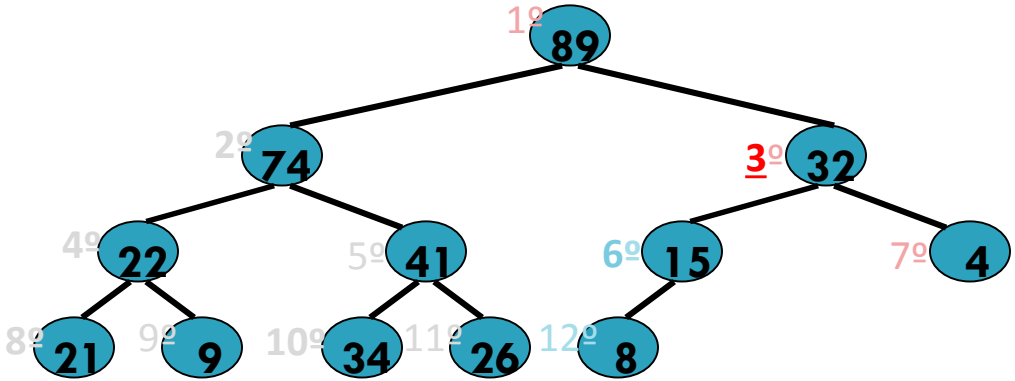


esquerda

esquerda

Algoritmo para achar última posição

- Agora N=3, é ímpar. Logo está à direita



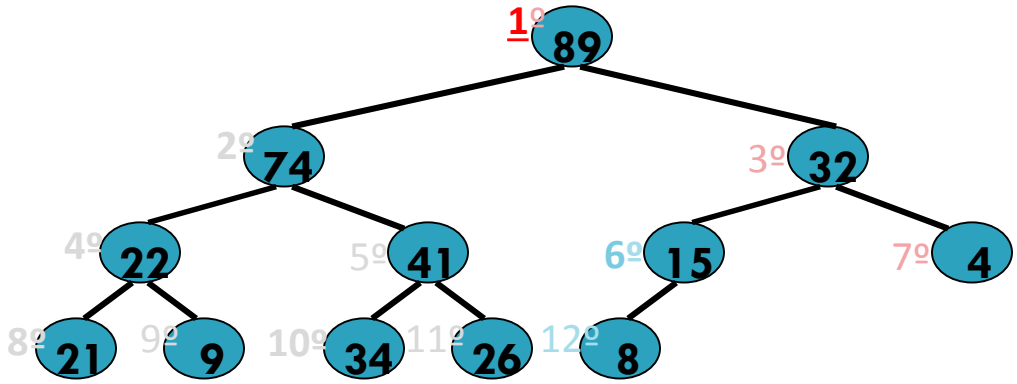
direita

esquerda

esquerda

Algoritmo para achar última posição

- Agora N=3, é ímpar. Logo está à direita.
  - O pai do 3° é o 3/2=1° item, **que é o raíz**!



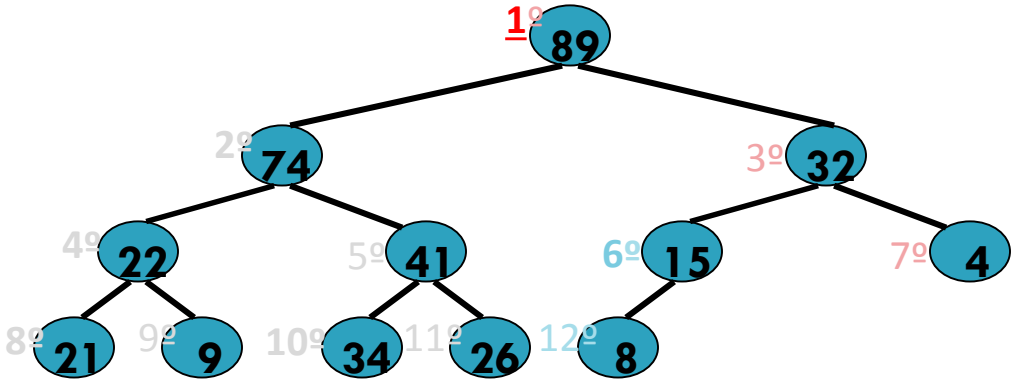
direita

esquerda

esquerda

Algoritmo para achar última posição

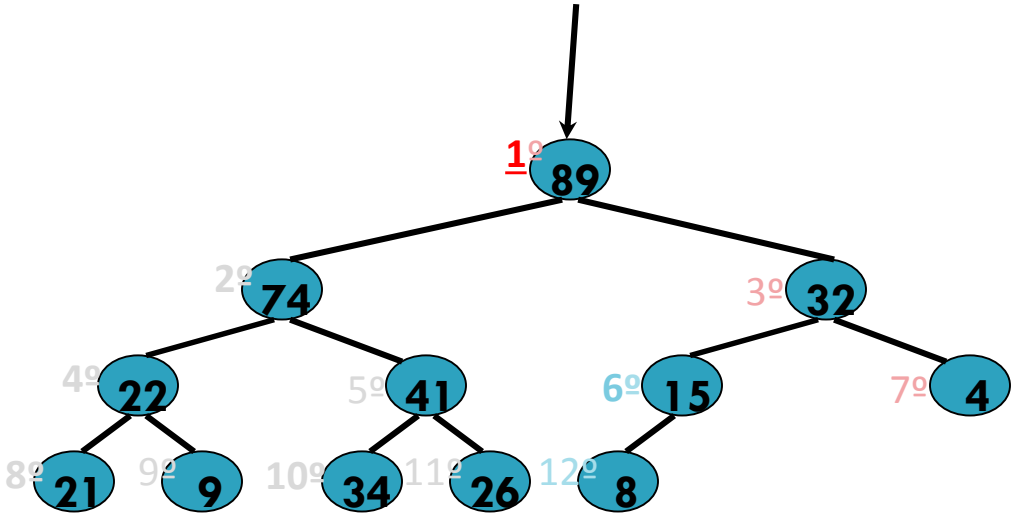
- Do último ao raiz, criamos uma pilha de percurso a custo (e de tamanho) $O(\log_2 n)$



PILHA
direita
esquerda
esquerda

Algoritmo para achar última posição

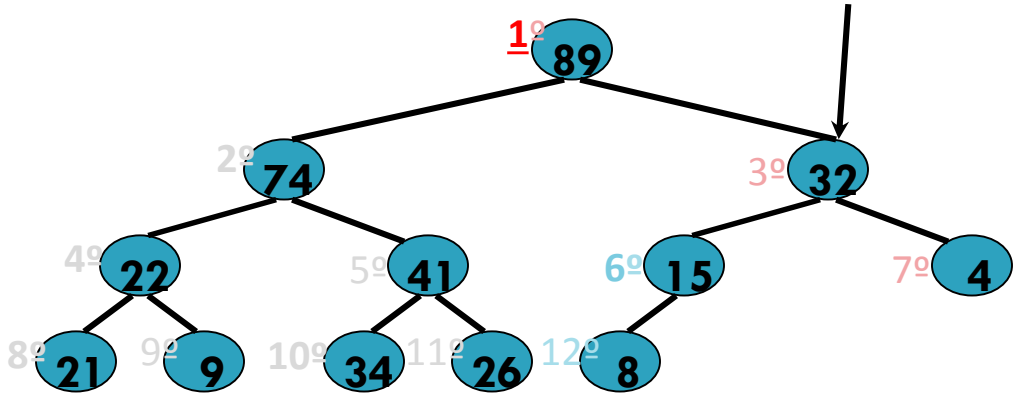
- Agora em cada iteração desde o raíz, realizamos um desempilhamento para orientar o percurso!



PILHA
direita
esquerda
esquerda

Algoritmo para achar última posição

- Agora em cada iteração desde o raíz, realizamos um desempilhamento para orientar o percurso!



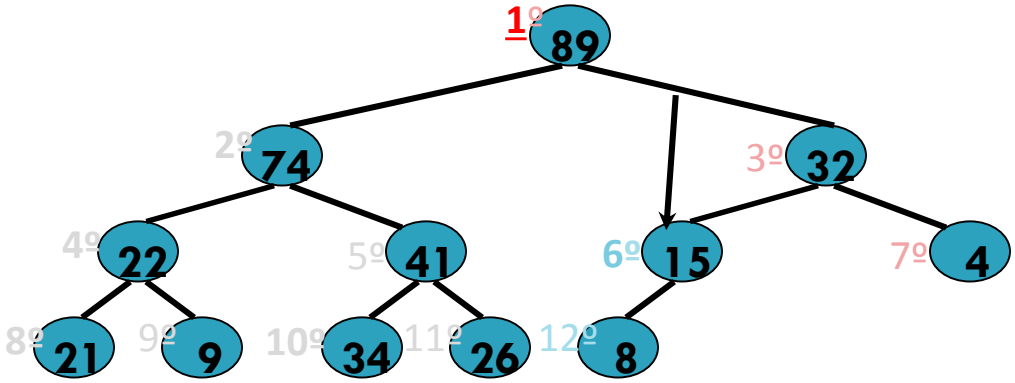
PILHA

esquerda

esquerda

Algoritmo para achar última posição

- Agora em cada iteração desde o raíz, realizamos um desempilhamento para orientar o percurso!

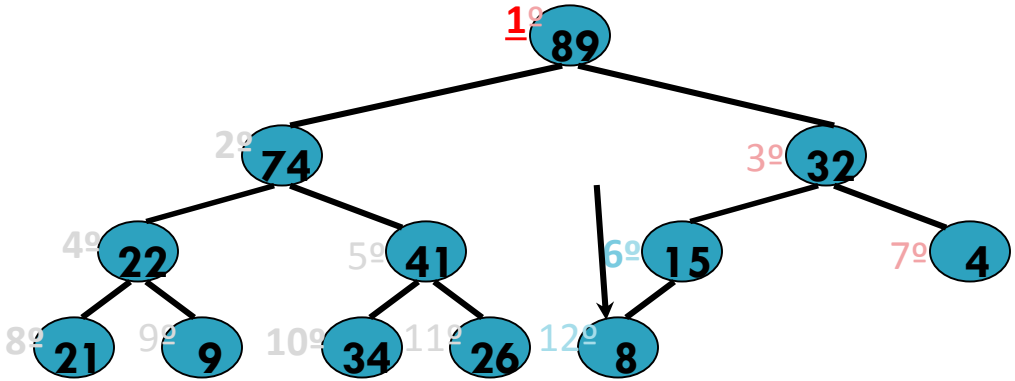


PILHA

esquerda

Algoritmo para achar última posição

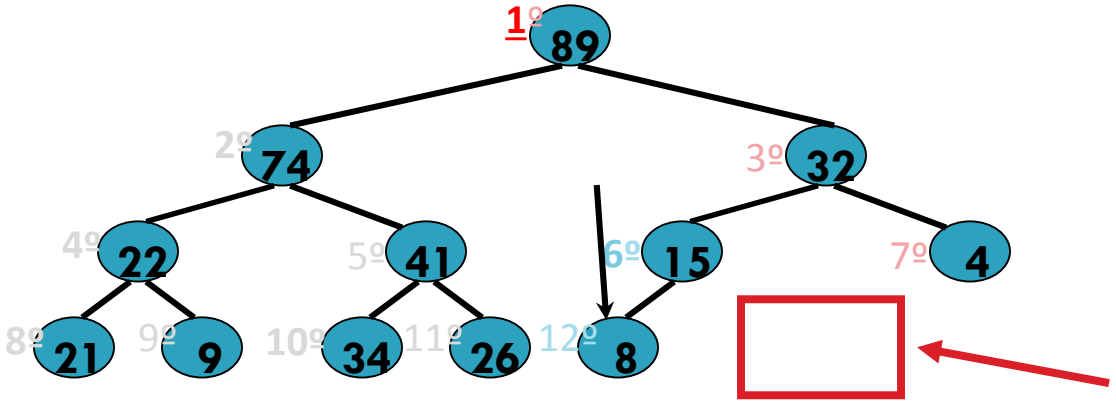
- Agora em cada iteração desde o raíz, realizamos um desempilhamento para orientar o percurso!
  - Custo de tempo total: $O(\log_2 n)$



PILHA

Posição do elemento a inserir

- No caso da inserção, basta primeiramente incrementar a variável N do total de itens e executar o algoritmo anterior



DESAFIO/PESQUISA-EXTRA

- Apresente um procedimento logarítmico (pode ser em pseudo-código) mais eficiente em tempo e espaço do que o anteriormente apresentado.
  - i.e., ele deve encontrar a última posição de um *heap* binário sem precisar de pilha nem de duas travessias logarítmicas raiz-folha.

Outras FPs

- Pesquisa a complexidade dos seguintes *heaps*
  - *Heaps* binomiais
  - *Relaxed heap*
  - *d-ary heaps*
  - *Heaps* de Fibonacci

Heap sort