Análise de Algoritmos

Ronaldo Cristiano Prati

Bloco A, sala 513-2

ronaldo.prati@ufabc.edu.br

Exemplo 1

- Considere que $T(n) = 3T(n/3) + 1$ e queremos provar pelo método da substituição que ele é $O(n)$
- Temo que provar que $T(n) \leq cn$ para alguma constante positiva c, por indução em n
- No passo indutivo, temos:

$$\begin{aligned} T(n) &= 3T(n/3) + 1 \\ &\leq cn + 1 \end{aligned}$$

- Entretanto, a nossa hipótese é que $T(n) \leq cn$, (e não $cn + 1$, como obtivemos no passo indutivo)

Exemplo 1

- Isso não significa em $T(n) \neq O(n)$.
- O problema é que a expressão que usamos para provar nosso palpite não foi "forte" o suficiente. Vamos tentar provar que $T(n) \leq cn - d$, que contina sendo $O(n)$. No passo indutivo temos:

$$\begin{aligned} T(n) &= 3T(n/3) + 1 \\ &\leq 3\left(\frac{cn}{3} - d\right) + 1 \\ &= cn - 3d + 1 \\ &\leq cn - d \end{aligned}$$

- No caso base tesmo que $T(1) = 1 \leq c - d$ sempre que $c \geq d + 1$, portanto $T(n) = O(cn - d) = O(n)$

Exemplo 2

- Considere que $T(n) = T(\lfloor n/2 \rfloor + 2) + 1$ e queremos provar pelo método da substituição que ele é $O(\log n)$, assumindo o caso base $T(4) = 1$

Exemplo 2

- É possível provar $T(n) \leq c \log n$, que no passo indutivo fica

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor + 2) + 1 \\ &\leq c \log \left(\frac{n}{2} + 2 \right) + 1 \\ &= c \log \left(\frac{n+4}{2} \right) + 1 \\ &\leq c \log(3n/2) - c + 1 \\ &= c \log n - c(2 - \log 3) + 1 \\ &\leq c \log n \end{aligned}$$

- em que a penúltima desigualdade vale para $n > 8$ e a última para $c \geq 1/(2 - \log 3)$

Exemplo 2

- Entretanto, fortalecendo a hipótese indutiva para $T(n) \leq c \log(n-a)$, no passo indutivo temos

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor + 2) + 1 \\ &\leq c \log\left(\frac{n}{2} + 2 - a\right) + 1 \\ &= c \log\left(\frac{n-a}{2}\right) + 1 \\ &= c \log(n-a) - c + 1 \\ &\leq c \log(n-a) \end{aligned}$$

- em que a primeira desigualdade vale para $a \geq 4$ e a última para $c \geq 1$. Escolhendo $a = 4, T(6) = T(5) + 1 = T(4) + 2 = 3 \leq c \log(6-4)$ para $c \geq 3$. Portanto, fazendo $a = 4$ e $c \geq 3$, temos que T(n) \leq c \log (n-a) para $n \geq 6$.

Fila de Prioridade

- Em algumas situações associamos uma prioridade a cada item coleção, e gostaríamos de ter acesso rápido ao item de maior prioridade
- É uma coleção dinâmica de elementos que possuem prioridades associadas e a operação de remoção deve **sempre** remover o elemento que possui maior prioridade.
- O termo prioridade é genérico: ter maior prioridade não significa necessariamente o maior valor.
  - Atendimento a idoso em um banco: maior idade $\rightarrow$ maior prioridade
  - Gerenciamento de estoque: menor quantidade de items $\rightarrow$ maior prioridade

Fila de Prioridade

- Filas de prioridades s ão muito´uteis na implementa¸c ão de diversos algoritmoscl´assicos como:
  - Ordenação (HeapSort)
  - Busca heurística (Greed search, A*)
  - Compressão (Código de Huffman)
  - Grafos: caminhos mínimos (Dijkstra), árvore geradora mínima(Prim), Ordenação topológica (Dependência semântica)
  - União de Conjuntos (Union-find)

Operações

- Constrói uma fila de prioridade a partir de um conjunto
- Insere elemento na fila
- Remove elemento com maior prioridade
- Alteração da prioridade de um elemento

Vetores ordenados

- Uma maneira de dar suporte a essas operações é usar um vetor ordenado
  - Contrução: $\mathcal{O}(n \log n)$,
  - Inserção/Alteração de prioridade: $\mathcal{O}(n)$
  - Remoção: $\mathcal{O}(1)$

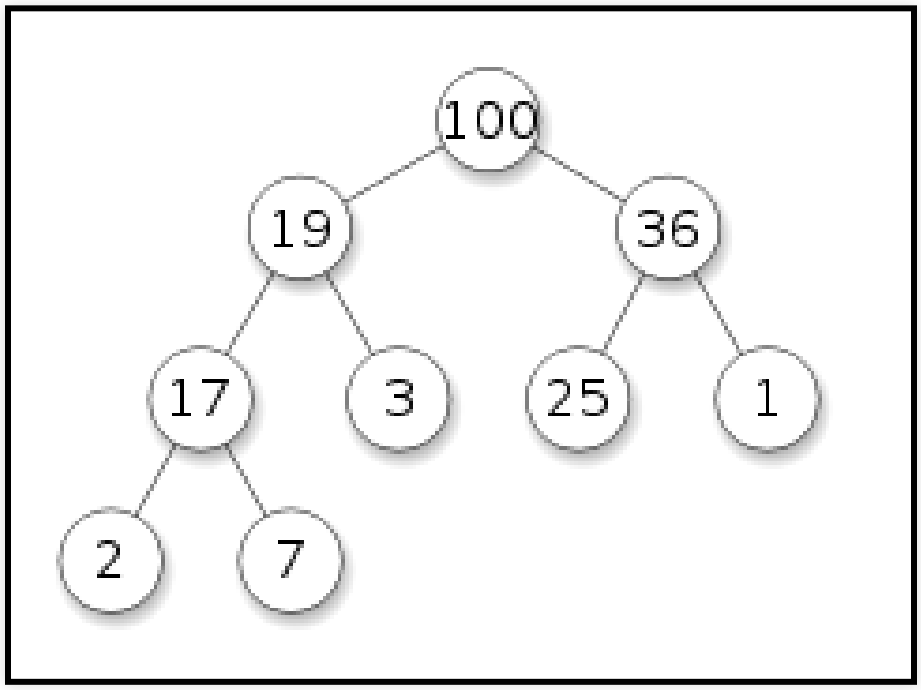
Heap

- Uma outra alternativa é usar Heaps
  - Construção: $\mathcal{O}(n)$,
  - Inserção/Alteração de prioridade: $\mathcal{O}(\log n)$
  - Remoção: $\mathcal{O}(1)$

Heap

- Heap pode ser entendida com uma árvore
  - quase completa:
    - Todos os níveis exceto o último devem estar totalmente preenchidos.
    - Se o seu último nível não estiver preenchido, os nós devem estar preenchidos da esquerda para a direita.
  - satisfaz a propriedade heap:
    - Um nó deve ter prioridade maior ou igual à prioridade de seus filhos, se eles existirem.

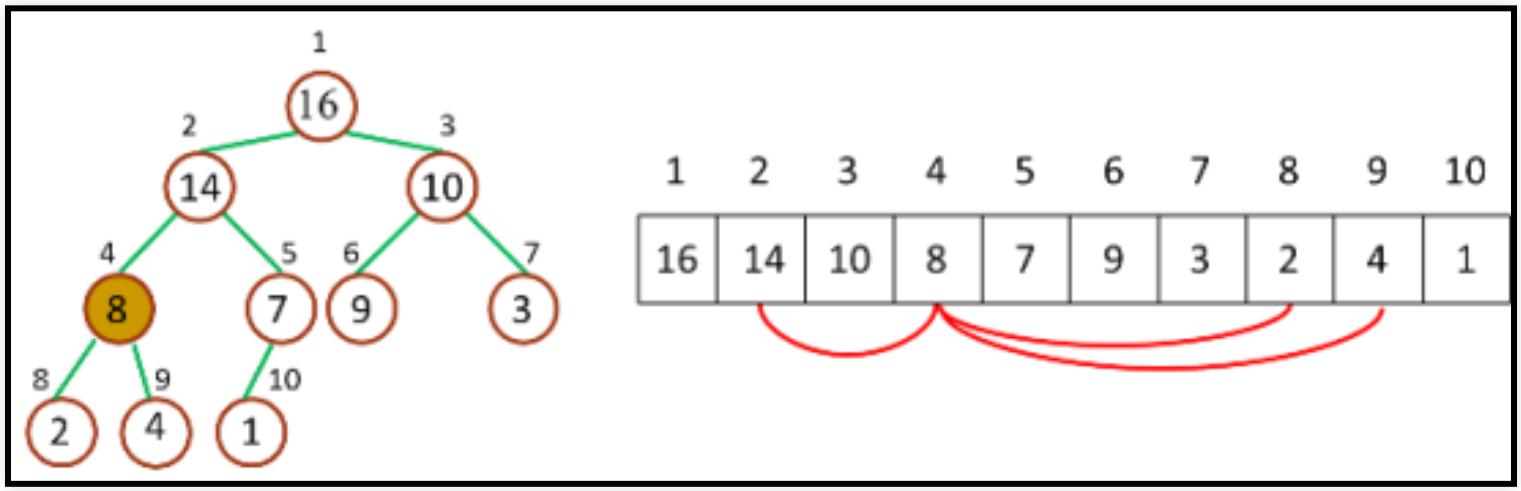
Heap



Heap

- No entanto, heaps são normalmente implementadas em um vetor
- Nesse caso, para uma determinada posição i
  - $pai(i) = \lfloor \frac{i}{2} \rfloor$
  - $filhoEsquerdo(i) = 2 * i$
  - $filhoDireito(i) = 2 * i + 1$
  - (obs): $\frac{i}{2}$ e $2 * i$ podem ser implementados de maneira eficiente usando bit shifts

Heap



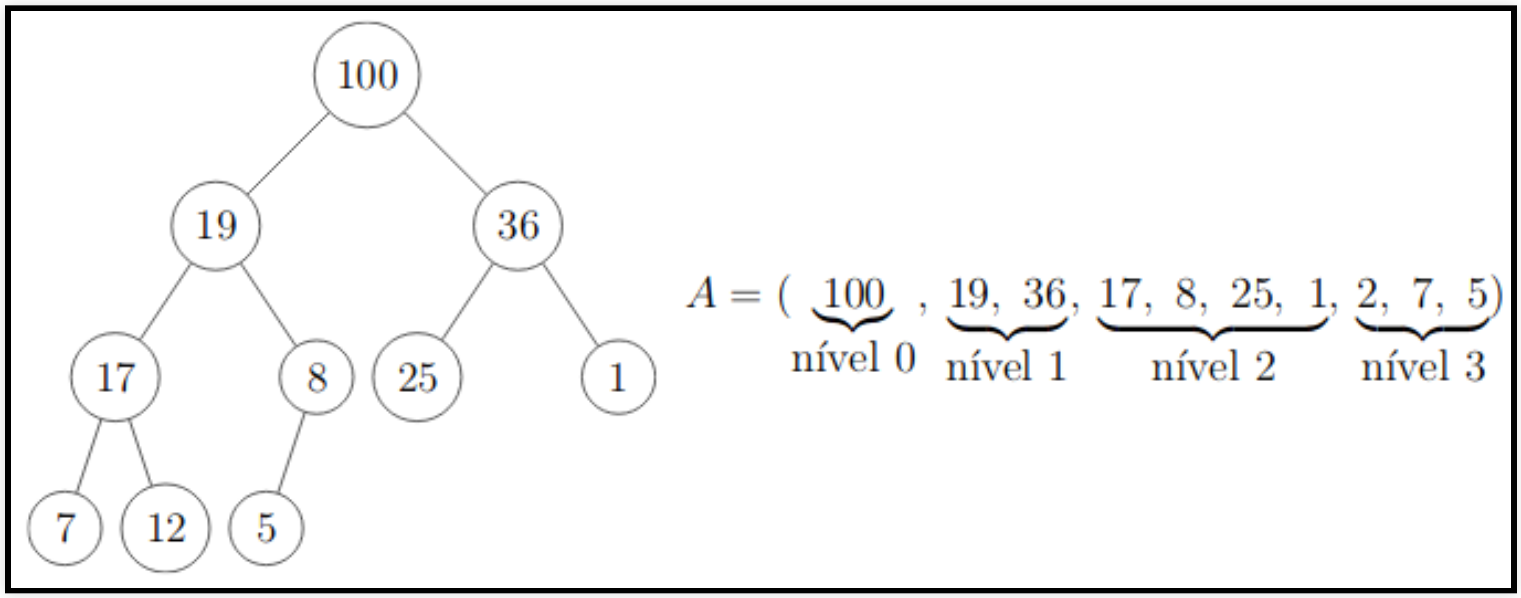
Heap

- No caso de índice começando em zero, para uma determinada posição i
  - $pai(i) = \lfloor \frac{i-1}{2} \rfloor$
  - $filhoEsquerdo(i) = 2 * i + 1$
  - $filhoDireito(i) = 2 * i + 2$

Construção do Heap

- Um vetor ordenado é pela priopridade é um Heap
- Um problema é que ordenar o vetor tem custo de (pelo menos) $\mathcal{O}(n \log n)$
- No entanto, o vetor não precisa estar completamente ordenado... somente os caminhos da "raíz" (primeiro nível) até as "folhas" (último nível) é que precisa estar ordenado

Construção da HEAP

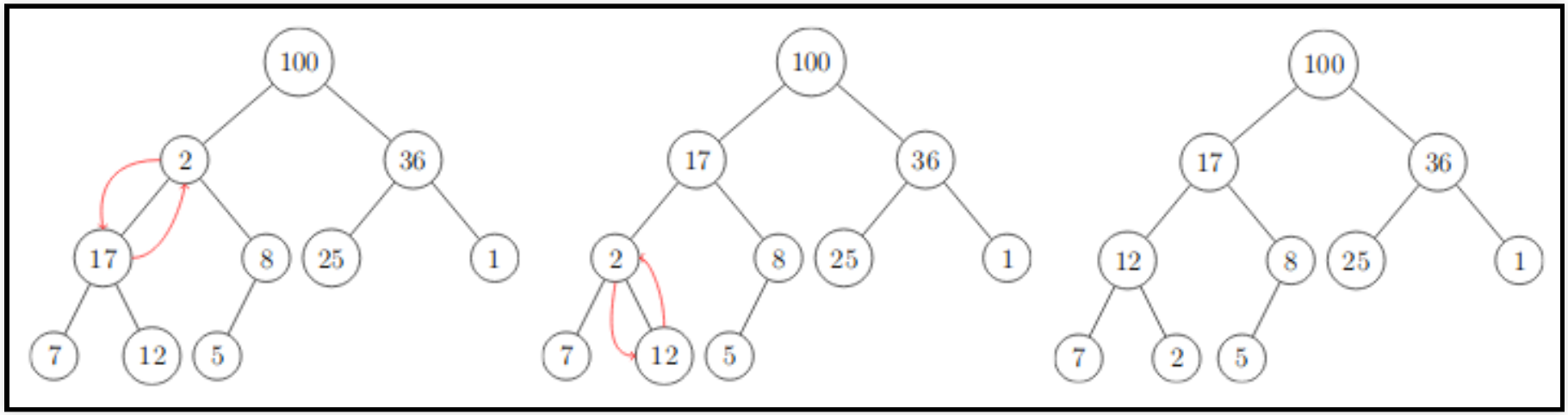


Corrige Heap Descendo

- Suponha que um determinada subárvore cuja raíz é $H[i]$ viola a propriedade heap (isto é, sua prioridade é menor que a de (pelo menos um) seus filhos)
- Entretanto, as subárvores filhas $H[2i]$ e $H[2i+1]$ são heap.
- Podemos corrigir a heap "descendo" o elmento $H[i]$:
  - Trocar $H[i]$ com o filho de maior prioridade
  - Chamar recursivamente CorrigeDescendo para o filho trocado (pois a propriedade Heap pode continuar violada para ele)

Corrige Heap Descendo

- Exemplos de CorrigeHeapDescendo(H,2)



Corrige Heap Descendo

Algoritmo 18: CorrigeHeapDescendo(H, i)

```
1 maior = i
2 se 2i ≤ H.tamanho e H[2i].prioridade > H[maior].prioridade então
3 |   maior = 2i
4 se 2i + 1 ≤ H.tamanho e H[2i + 1].prioridade > H[maior].prioridade
    então
5 |   maior = 2i + 1
6 se maior ≠ i então
7 |   troca H[i].indice com H[maior].indice
8 |   troca H[i] com H[maior]
9 |   CorrigeHeapDescendo(H, maior)
```

Corretude: Corrige Heap Descendo

Seja $h_x = \lfloor \log(n/x) \rfloor$ a altura do nó que está na posição x

Complexidade: Corrige Heap Descendo

- CorrigeHeapDescendo(H, i) tem um tempo proporcional a altura de i, ou seja $O(\log(n - i))$.
- No pior caso, i é a raiz da árvore, então a complexidade é proporcianal a altura da árvore
- Como a heap é uma árvore binária quase completa, sua altura é $O(\log n)$.
- Portanto, a complexidade de CorrigeHeapDescendo é $O(\log n)$

ConstroiHeap

- Considere que temos um vetor qualquer (não heap), e queremos transformá-lo em uma heap
- Podemos transformá-lo em uma haep utilizando CorrigeHeapDescendo como uma subrotina:
  - Observe que os nós folhas são heap (não tem filhos).
  - Podemos chamar CorrigeHeapDescendo para os nós não folha, começando pelo primeiro elemento não folha $H\lfloor n/2 \rfloor$ até o raíz $H[i]$

ConstroiHeap

Algoritmo 20: ConstroiHeap(H)

```
1 para i = 1 até H.tamanho faça
2     H[i].indice = i
3 para i = ⌊H.tamanho /2⌋ até 1 faça
4     CorrigeHeapDescendo(H, i)
```

ConstroiHeap

- É fácil mostrar que ConstroiHeap funciona, já que ele faz suscessivas chamadas a CorrigeHeapDescendo
- A invariante de laço é que a cada iteração do laço para indexado por i, para todo j tal que $i + 1 \leq j \leq n$, a´arvore enraizada em $H[j]$ e um heap.
- Como CorrigeHeapDescendo é excutado $H\lfloor n/2 \rfloor$ até $H[i]$, a invariante de laço se mantém e por indução podemos provar que ConstroiHeap constrói uma heap para qualquer vetor.

ConstroiHeap: Complexidade

- Uma análise preliminar, podemos constantar que ConstroiHeap tem complexidade $O(n \log n)$, uma vez CorrigeHeapDescendo tem complexidade $O(\log n)$ e ele é executado $\lfloor n/2 \rfloor$ vezes, que é $O(n)$
- Entretanto, observe que $O(\log n)$ é um limite superior para a complexidade de ConstroiHeap. Na verdade sua complexidade é proporcianal à algura do elemente i, ou seja, $O(\log(n - i))$.
- Podemos usar esse fato para encontrar uma complexidade mais justa para o ConstroiHeap

ConstroiHeap: Complexidade

- Uma estimativa quantidade de nós em uma dada altura h da heap é $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.
- Podemos escrever a complexidade do ConstroiHeap como:

$$
\begin{aligned}
T(n) &= \sum_{h=0}^{\log(n)} \left\lceil \frac{n}{2^{h+1}} \right\rceil \times O(h) \\
&= O\left(n \times \sum_{h=0}^{\lg(n)} \frac{h}{2^h} \right) \\
&= O\left(n \times \sum_{h=0}^{\infty} \frac{h}{2^h} \right)
\end{aligned}
$$

ConstroiHeap: Complexidade

- Observe que para $i \geq 1$, $\left((i+1)/2^{i+1}\right) \,/\, \left(i/2^{i}\right) < 1$.

$$
\begin{aligned}
T(n) &= O\left(n \times \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\
&\leq cn \\
&= O(n)
\end{aligned}
$$

Remover item da heap

- Uma outra operação da heap é a remoção do item de maior prioridade
- Observe que o primeiro elmento da heap é o item de maior prioridade. Então acessá-lo é $O(1)$
- Para a remoção, precisamos colocar outro elmento no lugar.
- Uma alteração que mexe o mínimo possível na Heap é trocar o primeiro com o último elemento.
- Podemos então usar CorrigeHeapDescendo(H, 1) para corrigir a heap

RemoveDaHeap

Algoritmo 21: RemoveDaHeap(H)

```
1 x = null
2 se H.tamanho ≥ 1 então
3     x = H[1]
4     H[H.tamanho].indice = 1
5     H[1] = H[H.tamanho]
6     H.tamanho = H.tamanho − 1
7     CorrigeHeapDescendo(H, 1)
8 retorna x
```

Complexidade e Corretude de RemoveDaHeap

- Observe que RemoveDaHeap viola apenas a propriedade heap do nó raiz, já que o último elemento é uma folha e não mexemos em outras posições da heap.
  - CorrigeHeapDescendo($H, 1$) restaura a propriedade heap, portanto RemoveDaHeap está correto
- A complexidade é dominada por CorrigeHeapDescendo($H, 1$), já que a troca é $O(1)$. Portanto, a complexidade é $O(\log n)$

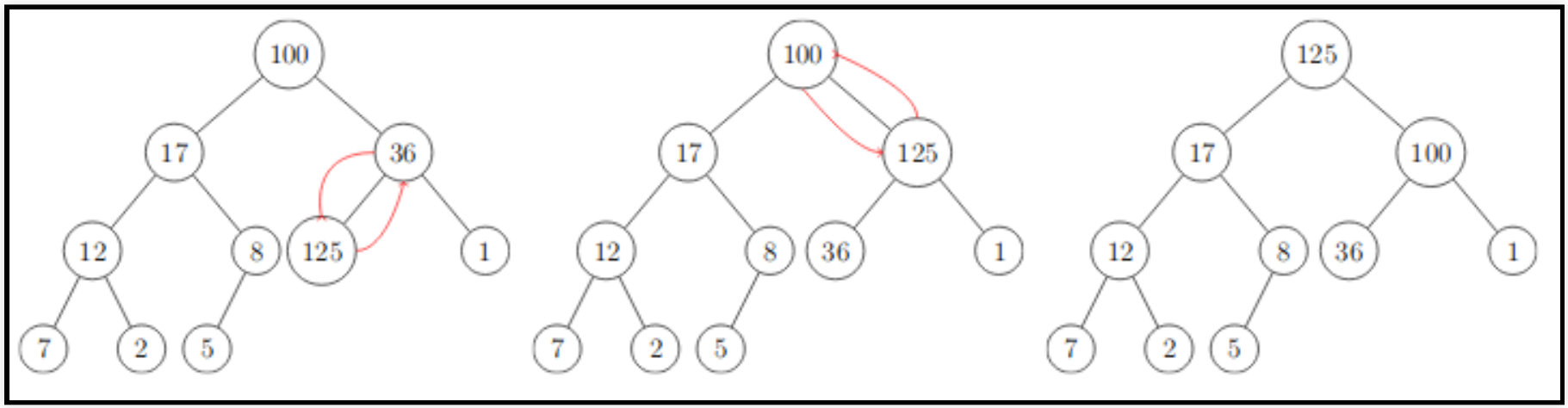
Inserir elemento na heap

- Caso haja espaço na heap, uma estratégia é inserir na primeira posição disponível
- Obviamente, isso pode violar propriedade heap
- No entanto, antes da inserção, a propriedade heap está mantida.
- O procedimento CorrigeHeapSubindo pode ser usado para restaurar a priopridade heap

CorrigeHeapSubindo

- Dada uma posição $H(i)$, se a prioridade de $H(i)$ é maior que a de seu pai, podemos trocar $H(i)$ com o seu pai.
- A propriedade heap das subárvores está restabelecida, mas eventualmente é preciso continuar subindo até chegar ao nó raiz.

CorrigeHeapSubindo



CorrigeHeapSubindo

Algoritmo 19: CorrigeHeapSubindo(H, i)

```
1 pai = ⌊i/2⌋
2 se i ≥ 2 e H[i].prioridade > H[pai].prioridade então
3     troca H[i].indice com H[pai].indice
4     troca H[i] com H[pai]
5     CorrigeHeapSubindo(H, pai)
```

Complexidade e Corretude do CorrigeHeapSubindo

- Um argumento similar ao usado para CorrigeHeapDescendo pode ser usado para demonstrar por indução que CorrigeHeapSubindo está correto
- De maneira similar, podemos mostrar que a complexidade de CorrigeHeapSubindo é $O(n \log n)$

InserirNaHeap

Algoritmo 22: InsereNaHeap(H, x)

```
1 se H.tamanho ≠ H.capacidade então
2     H.tamanho = H.tamanho + 1
3     x.indice = H.tamanho
4     H[H.tamanho] = x
5     CorrigeHeapSubindo(H, H.tamanho)
```

Complexidade e Corretude de RemoveDaHeap

- Observe que InserirNaHeap viola apenas a propriedade heap do último nó, já que não mexemos em outras posições da heap.
  - CorrigeHeapSubindo($H, 1$) restaura a propriedade heap, portanto InserirNaHeap está correto
- A complexidade é dominada por CorrigeHeapSubindo, já que a inserção é $O(1)$. Portanto, a complexidade é $O(\log n)$

Altera prioridade de um elemento

- Para alterar a prioridade de um elemento, podemos usar CorrigeHeapSubindo ou CorrigeHeapDescendo, dependendo se a prioridade do elemento i subiu ou desceu
  - Se a prioridade cresceu, chamar CorrigeHeapSubindo(H, i)
  - Se a prioridade desceu, CorrigeHeapDescendo(H, i)

AlteraHeap

Algoritmo 23: AlteraHeap(H, i, k)

```
1 aux = H[i].prioridade
2 H[i].prioridade = k
3 se aux < k então
4     CorrigeHeapSubindo(H, i)
5 se aux > k então
6     CorrigeHeapDescendo(H, i)
```

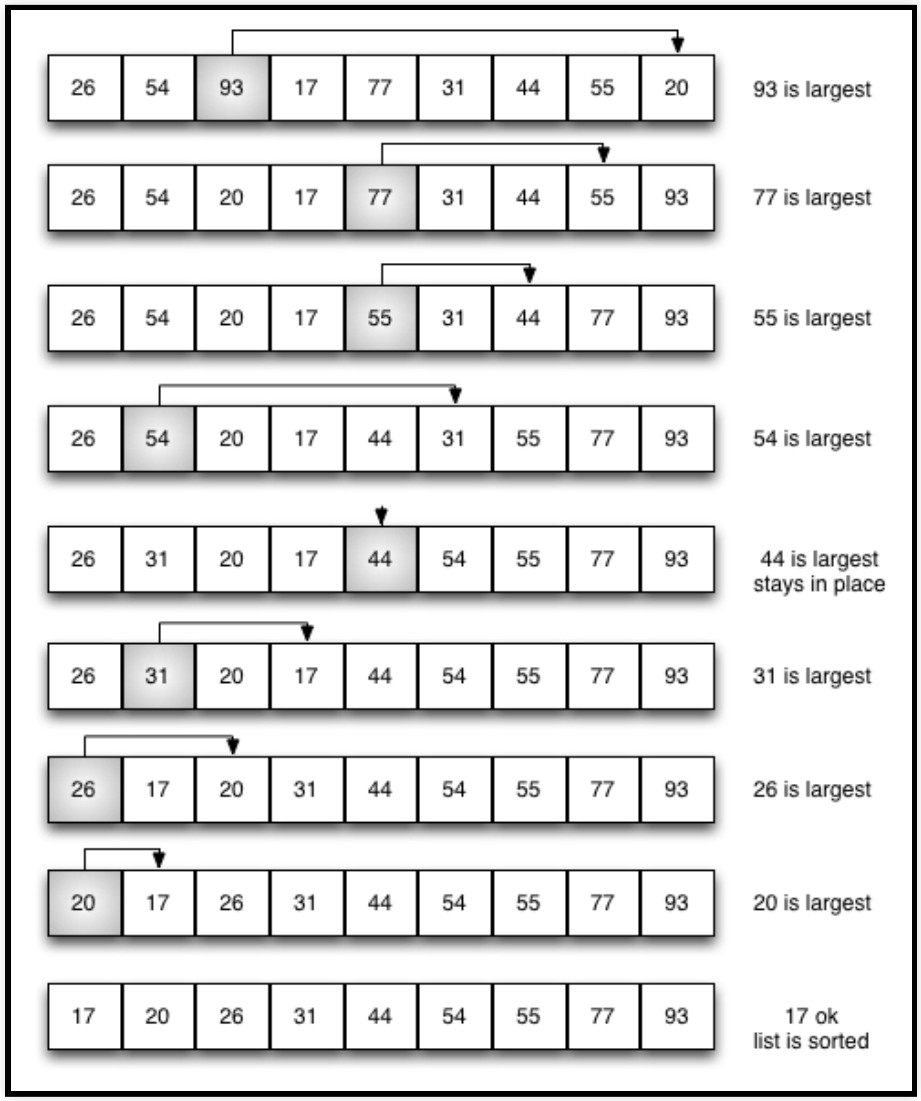
Complexidade e Corretude de AlteraHeap

- Como AlteraHeap é baseado em CorrigeHeapSubindo e CorrigeHeapDescendo, podemos concluir que ele está correto, e que a sua complexidade é $O(n \log n)$

Selection Sort

- Uma estratégia simples para ordenar vetor é:
  - Percorrer o vetor e encontrar o máximo
  - Trocar o último elemento pelo máximo
  - Repetir o processo, considerando o vetor com tamanho $n - 1$
- Essa estratégia é conhecida como como Selecion Sort

Selection Sort


26 54 93 17 77 31 44 55 20
93 is largest
26 54 20 17 77 31 44 55 93
77 is largest
26 54 20 17 55 31 44 77 93
55 is largest
26 54 20 17 44 31 55 77 93
54 is largest
26 31 20 17 44 54 55 77 93
44 is largest
stays in place
26 31 20 17 44 54 55 77 93
31 is largest
26 17 20 31 44 54 55 77 93
26 is largest
20 17 26 31 44 54 55 77 93
20 is largest
17 20 26 31 44 54 55 77 93
17 ok
list is sorted

Selection Sort

Algoritmo 31: SELECTIONSORT(A, n)

```
1 para i = n até 2 faça
2     indiceMax = i
3     para j = 1 até i − 1 faça
4         se A[j] > A[indiceMax] então
5             indiceMax = j
6     troca A[indiceMax] com A[i]
7 retorna A
```

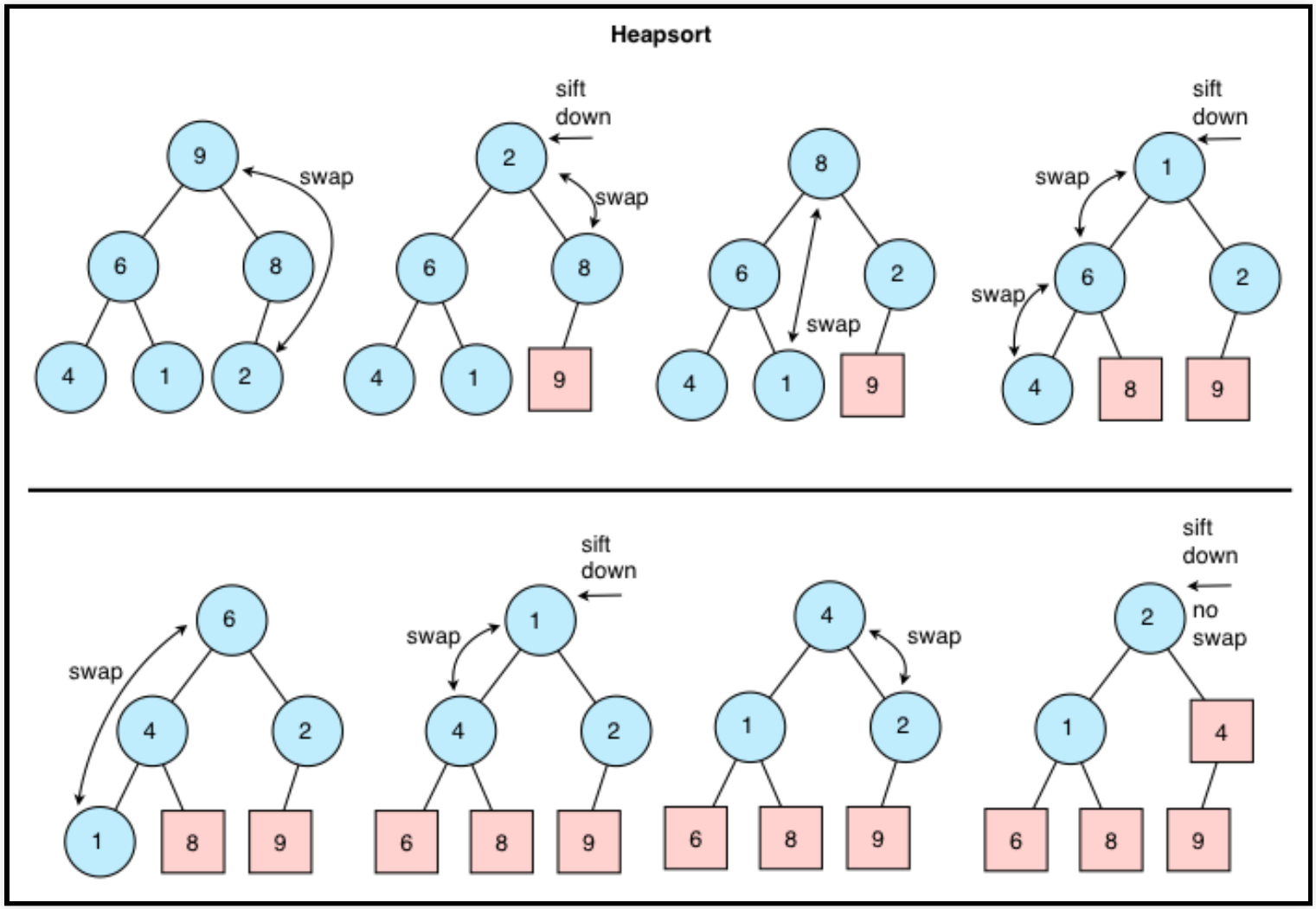
Selection Sort

- A análise de corretude do selecion sort é similar a do heap sort (veremos mais adiante), assumindo que o algoritmo que seleciona o máximo funciona
- A complexidade é $O(n^2)$

Heap Sort

- Podemos usar uma Heap para ordenar um vetor
  - A ideia é contruir uma Heap com o vetor, e trocar o último elemento com o primeiro, rafazendo-se a Heap com tamanho $n - 1$
  - Repete-se esse processo até ter o vetor ordenado

Heap Sort


Heapsort
9
swap
6
8
4
1
2
sift
down
2
swap
6
8
4
1
9
8
6
2
swap
4
1
9
sift
down
1
swap
swap
6
2
4
8
9
6
swap
4
2
1
8
9
sift
down
6
swap
4
2
1
8
9
sift
down
1
swap
4
2
6
8
9
4
swap
1
2
6
8
9
sift
down
2
no
swap
1
4
6
8
9

Heap Sort

Algoritmo 34: HEAPSORT(A, n)

```
1 ConstroiHeap(A)
2 para i = n até 2 faça
3     troca A[1] com A[i]
4     A.tamanho = A.tamanho − 1
5     CorrigeHeapDescendo(A, 1)
```

Corretude do Heap Sort

Heap Sort

- Heap Sort tem complexidade $O(n \log n)$ (cada chamada para refazer a heap tem custo $O(\log n)$, e é chamada n vezes.)
- Além disso, também há o custo adicional de construir a heap $\mathcal{O}(n)$.
- A ordem original dos elementos iguais pode mudar, portanto ele não é estável