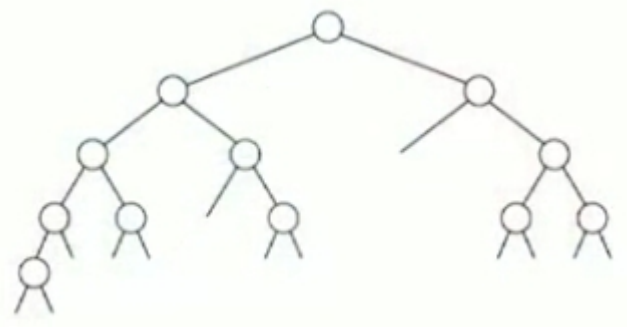
Pesquisa e Ordenação de Dados

Unidade 2.6:

Heap Sort

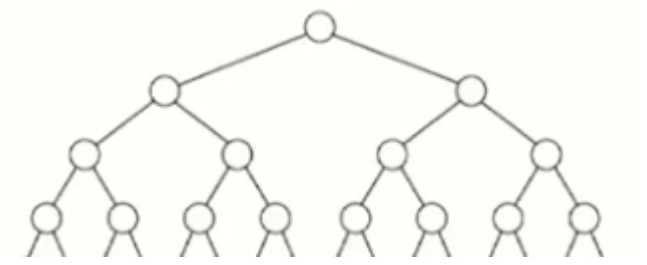
Conceitos

- Árvore binária
  - cada nó pode conter nenhum, 1 ou 2 filhos



Conceitos

- Árvore binária **cheia**
  - Estritamente binária (todo nó possui 0 ou 2 filhos)
  - Todos os nós folha se encontram no último nível
  - O número total de nós em uma árvore binária cheia de altura **h** é dado por:
    - $t_n = 2^h - 1$
  - A altura da árvore é dada por:
    - $h = \log_2(t_n + 1)$

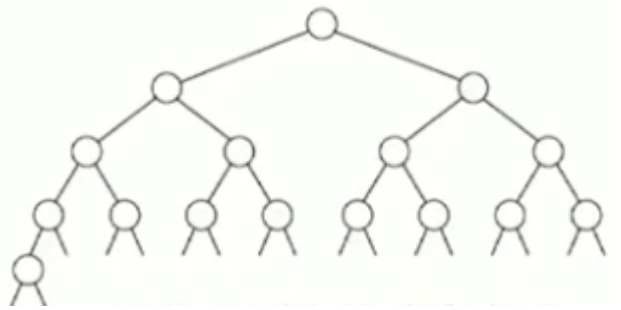


$t_n = 2^4 - 1 = \mathbf{15}$

$h = \log_2(15 + 1) = \mathbf{4}$

Conceitos

- Árvore binária **completa**
  - é uma árvore binária cheia até o penúltimo nível.
  - o último nível pode estar incompleto, mas todos os nós devem estar mais à esquerda possível.
  - a altura é próxima de O(log n).



Heap binário

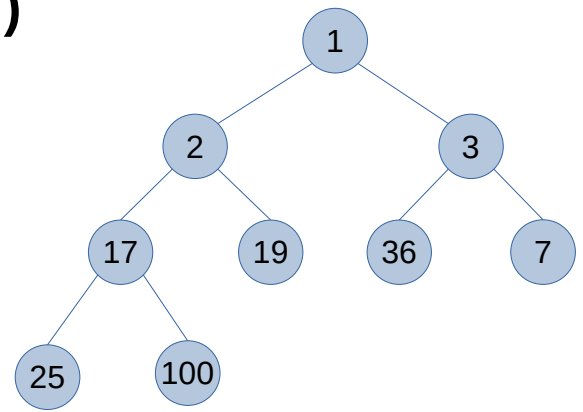
- Heap binário
  - é uma representação em vetor de uma árvore binária completa onde os elementos (chaves) são dispostos segundo uma relação de ordem parcial
    - Ordem parcial: somente há relação de ordem entre os nós pai e seus filhos; não há relação de ordem entre nós irmãos;
    - As operações de inserção/remoção devem manter a ordem parcial.
  - Assim como a árvore binária completa:
    - A árvore está completamente preenchida em todos os níveis, exceto talvez o mais baixo;
    - O nível mais baixo é preenchido a partir da esquerda.

Heap binário

- Heap mínimo: a chave de menor valor se encontra na raiz.
  - *Min heap property:* para todo nó (exceto a raiz), a chave do pai é menor ou igual à chave dos filhos

    A[i] <= filho_esq(i)

    A[i] <= filho_dir(i)

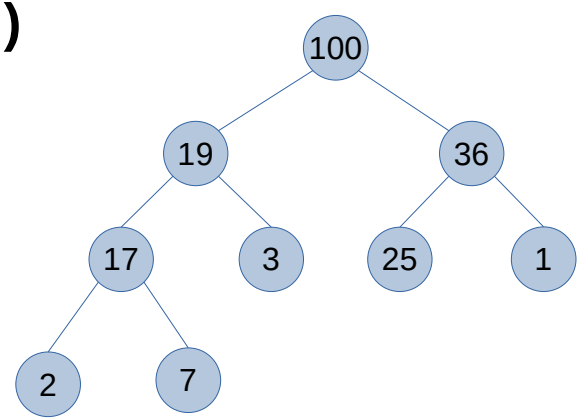


Heap binário

- Heap máximo: a chave de maior valor se encontra na raiz.
  - *Max heap property:* para todo nó (exceto a raiz), a chave do pai é maior ou igual à chave dos filhos

    A[i] >= filho_esq(i)

    A[i] >= filho_dir(i)

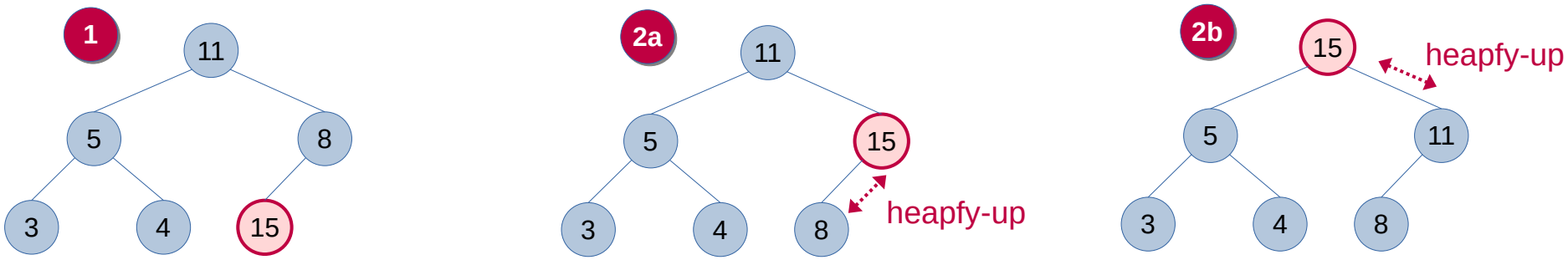


Heap binário - Operações

- Inserção:
  1) criar um novo nó folha no último nível, na posição mais à esquerda possível.
  2) verificar se a nova estrutura satisfaz à relação de ordem parcial do heap (*min* ou *max heap property*).
     - 2.a) o novo elemento deve ser comparado com seu pai; se a relação de ordem foi prejudicada, executa-se o processo de subida (*heapfy up* ou *swim - flutuar*).
     - 2.b) O processo é repetido até chegar à raiz.
  - A complexidade da inserção é **O(log n)**, pois no pior caso, serão feitas tantas trocas quanto a altura da árvore.

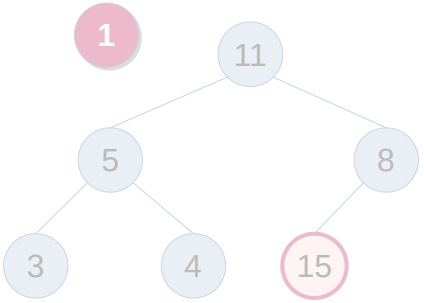
Heap binário - Operações

- Inserção
  - Exemplo: inserção do valor 15 em um heap máximo

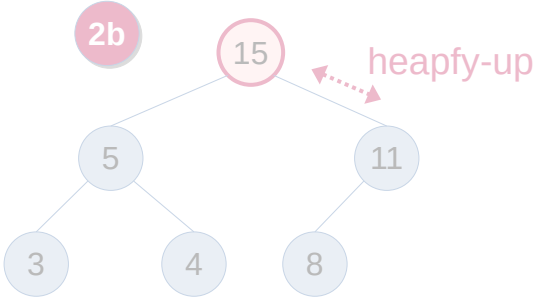


Heap binário - Operações

- Inserção
  - Exemplo: inserçã máximo



up

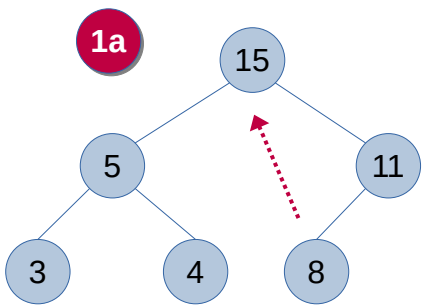


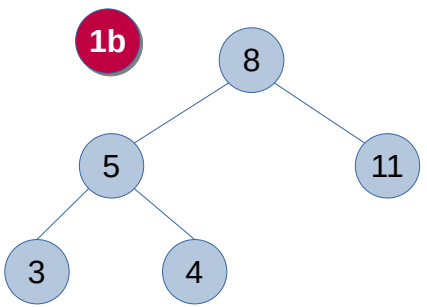
Heap binário - Operações

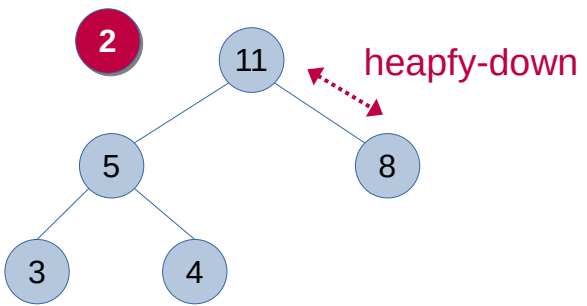
- Remoção
  - extração a maior (ou menor) chave de um conjunto.
  1) Substituir o nó raiz (que foi removido) pelo nó que está no último nível, o mais à direita possível.
  2) Se a ordem parcial do heap foi prejudicada, realizar o processo de *heapfy down* (ou *sink* - afundar), "descendo" o elemento que está na raiz e trocando-o sucessivamente com seu filho de maior valor (caso do heap máximo), de modo que a relação de ordem seja reestabelecida.
  - A complexidade da remoção é **O(log n)**, pois no pior caso, serão feitas tantas trocas quanto a altura da árvore.

Heap binário - Operações

- Remoção
  - Exemplo: retirada do nó 15







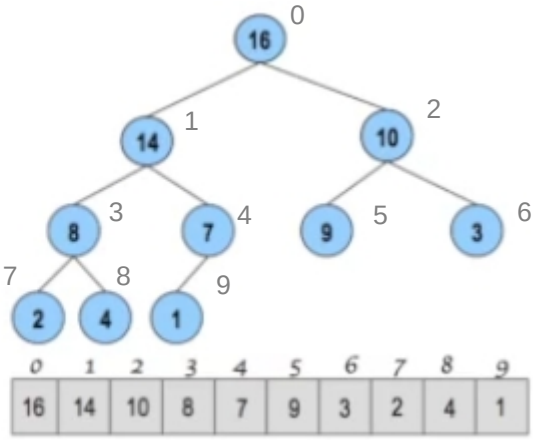
Heap binário - Operações

- Busca
  - em geral busca-se o elemento raiz
  - portanto, o procedimento de busca deve apenas retornar o valor da raiz
  - complexidade constante → O(1)

Representação em vetor

- Um heap é representado por meio de um vetor compacto, isto é, onde não há posições vazias.

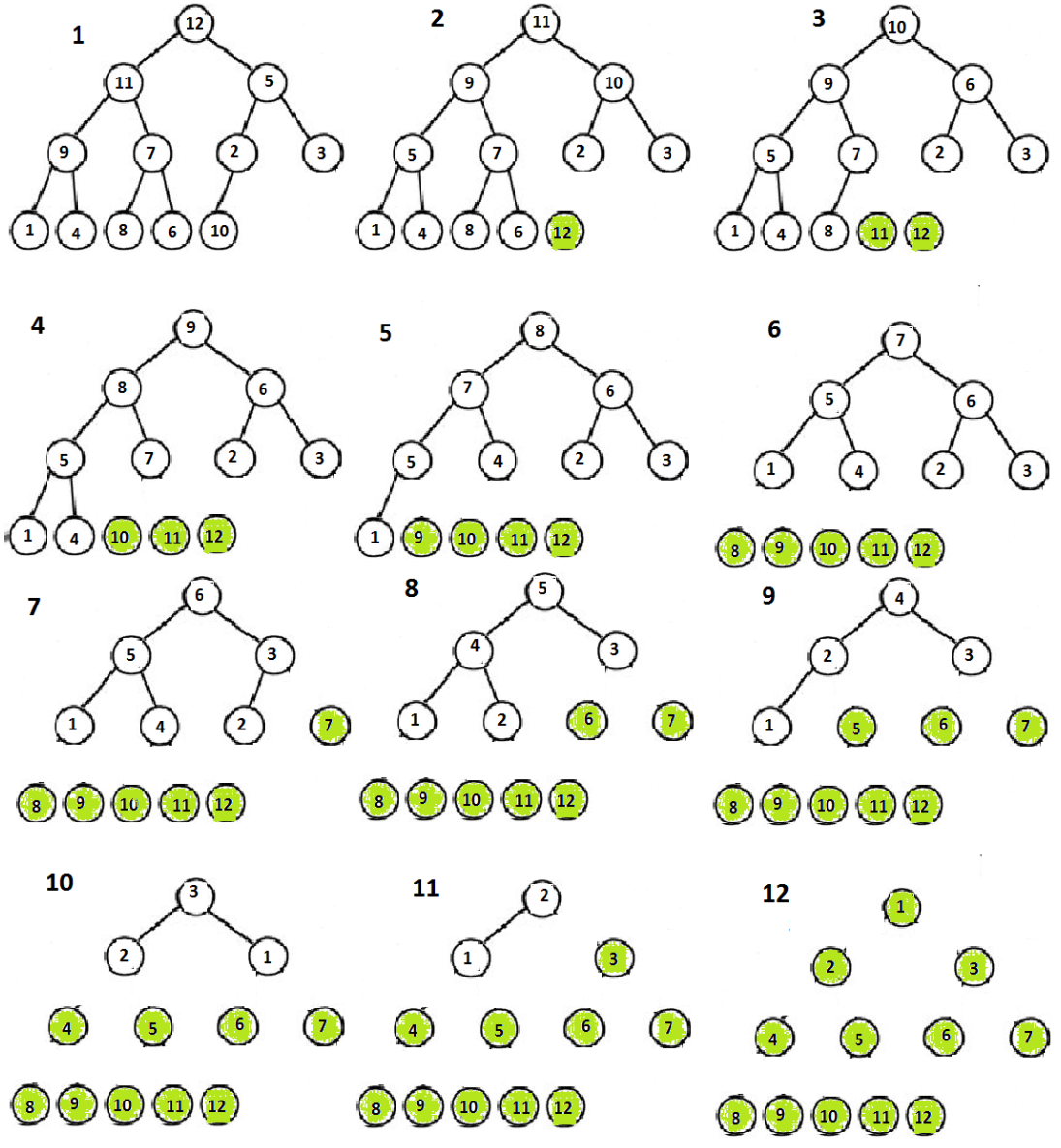
```
Raiz = 0
Pai(i) = (i-1)/2
Filho_esq(i) = 2i+1
Filho_dir(i) = 2i+2
```



Representação em vetor

- Da mesma forma, dado um vetor, é possível rearranjar seus elementos para que ele se torne um heap
- Heap máximo:
  - enquanto o valor de um filho for maior que o de seu pai: troque os valores de pai e filho e "suba" um passo em direção à raiz.
- Mais precisamente:
  - para todo nó i, enquanto A[pai(i)] < A[i]:
    - troca(A[pai(i)], A[i])
    - i = pai(i)

Heap Sort


1
12 11 5 9 7 2 3 1 4 8 6 10
2
11 9 10 5 7 2 3 1 4 8 6 12
3
10 9 6 5 7 2 3 1 4 8 11 12
4
9 8 6 5 7 2 3 1 4 10 11 12
5
8 7 6 5 4 2 3 1 9 10 11 12
6
7 5 6 1 4 2 3 8 9 10 11 12
7
6 5 3 1 4 2 7 8 9 10 11 12
8
5 4 3 1 2 6 7 8 9 10 11 12
9
4 2 3 1 5 6 7 8 9 10 11 12
10
3 2 1 4 5 6 7 8 9 10 11 12
11
2 1 3 4 5 6 7 8 9 10 11 12
12
1 2 3 4 5 6 7 8 9 10 11 12

Heap Sort

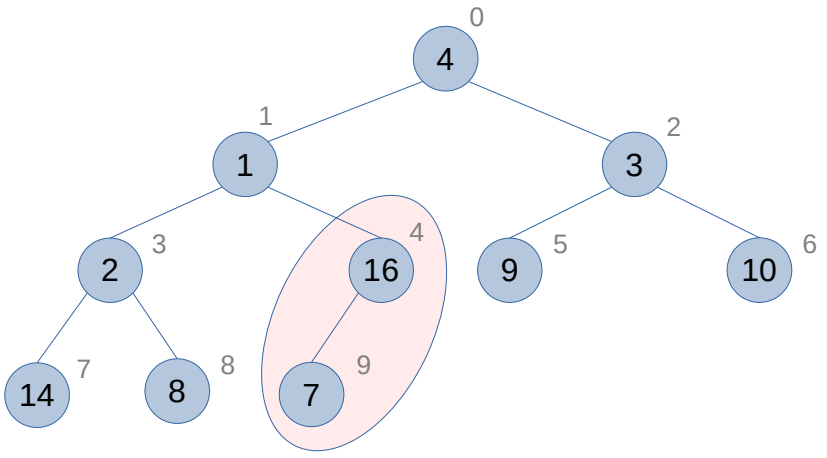
- É um algoritmo de ordenação desenvolvido em 1964 por Robert Floyd e John Williams
- Trata-se de um método de seleção dos elementos na ordem desejada em uma árvore binária heap
- Consiste em 2 fases:
  - 1: construir um heap máximo através da reorganização dos elementos presentes no vetor inicial;
  - 2: repetidamente remover o maior elemento e posicioná-lo ao final da parte não ordenada do vetor, garantindo que a propriedade *max heap* continue válida para o restante dos elementos não ordenados.

Heap Sort

- **Fase 1**: construção do heap máximo
  - Dado um array, realizar as trocas necessárias para atender à propriedade *max heap*
    - para toda subárvore, A[pai(i)] >= A[i]
  - Os testes das chaves se iniciam pela última subárvore (nível mais inferior e mais à direita possível), prosseguindo, a partir daí, para as subárvores que a antecedem, até chegar à raiz da árvore.
  - Quando encontramos uma subárvore que viola a condição de *max heap*, selecionamos seu maior filho e trocamos sua posição com a raiz da subárvore, repetindo o processo para todas as suas subárvores.
  - Os nós folha não precisam ser testados, pois, tomados sozinhos, já são *max heap*; o último nó que possui filhos estará na posição **n/2-1**.

Heap Sort

Como funciona – Construção do Max Heap (1)



Posições 5, 6, 7, 8 e 9 são folhas, então atendem à propriedade *max-heap*: não são menores do que seus filhos (inexistentes).

A verificação começará pela subárvore de raiz 4 (10/2-1).

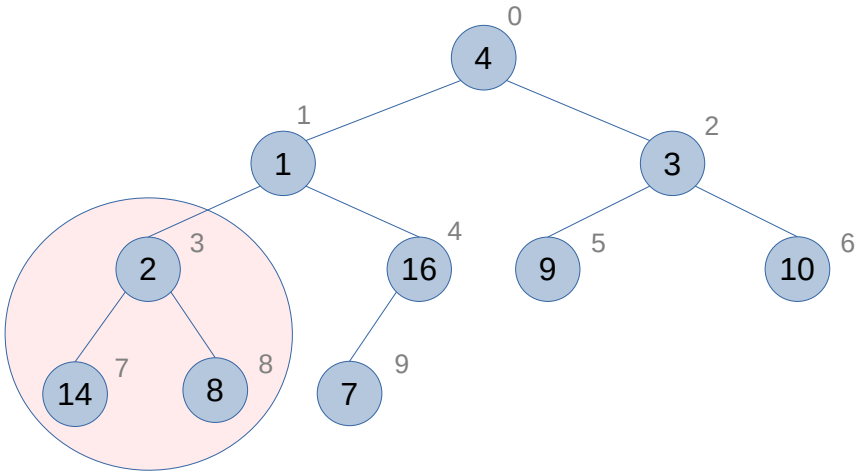
Não é necessário fazer nada, pois o valor do pai (16) é maior do que o de seu único filho (7).

Vetor inicial (ordem aleatória)

4	1	3	2	16	9	10	14	8	7
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (2)



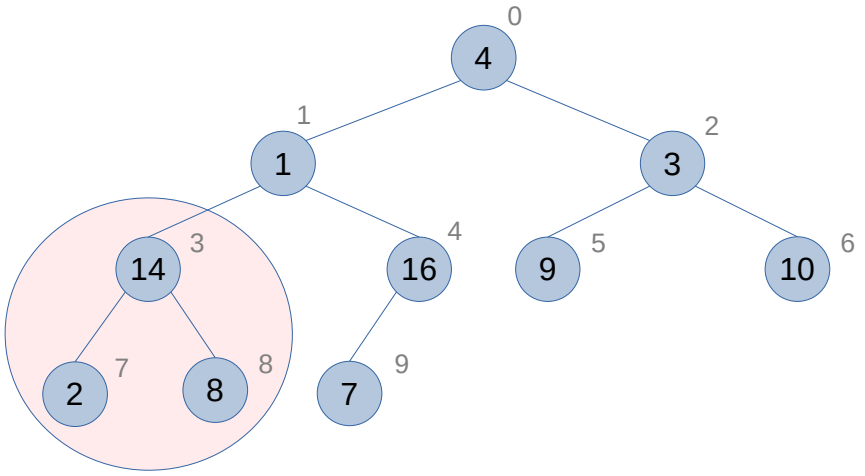
A verificação agora é da subárvore de raiz 3.

O valor da raiz será “afundado”: trocará de posição com a de seu maior filho (14).

4	1	3	2	16	9	10	14	8	7
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (3)



A verificação agora é da árvore de raiz 3.

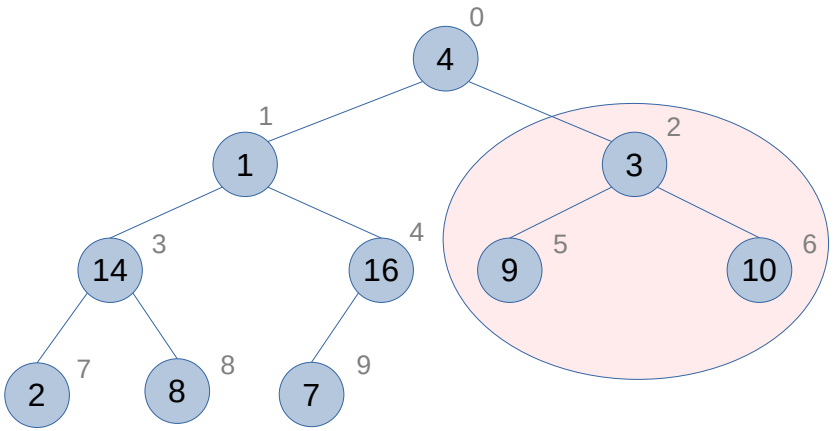
O valor da raiz será “afundado”: trocará de posição com a de seu maior filho (14).

A subárvore de raiz 3 agora obedece à propriedade max-heap.

4	1	3	14	16	9	10	2	8	7
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (4)



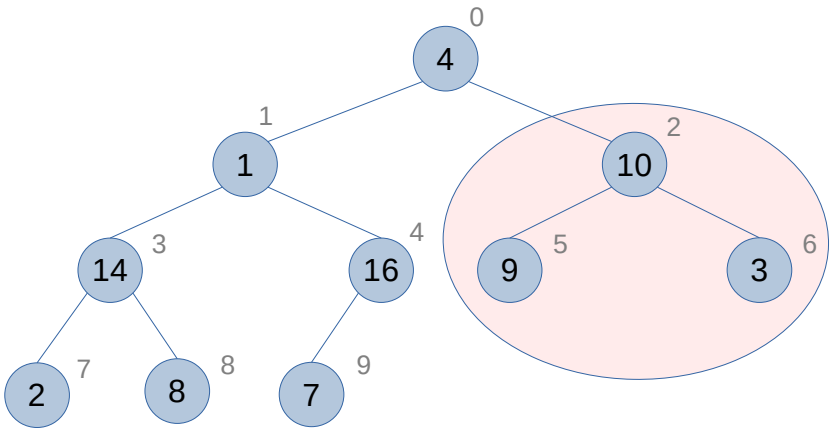
A verificação agora é da subárvore de raiz 2.

O valor da raiz será “afundado”: trocará de posição com a de seu maior filho (10).

4	1	3	14	16	9	10	2	8	7
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (5)



A verificação agora é da subárvore de raiz 2.

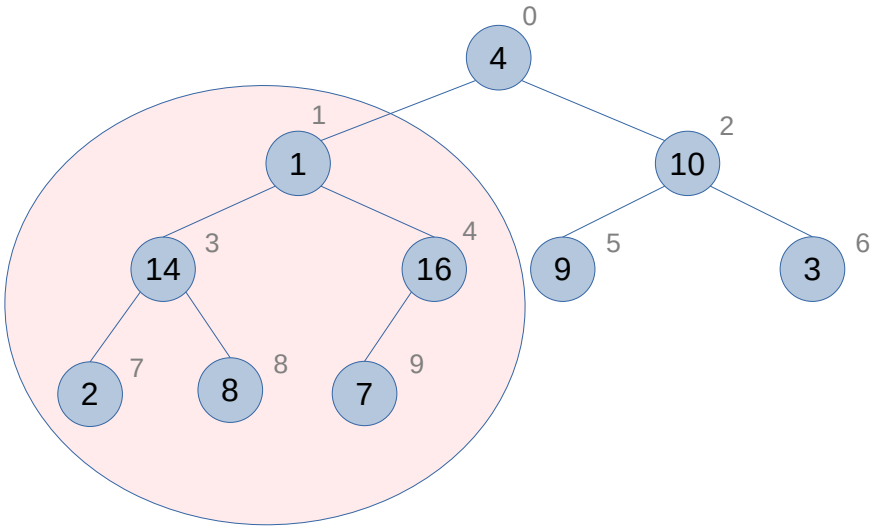
O valor da raiz será “afundado”: trocará de posição com a de seu maior filho (10).

A subárvore de raiz 2 agora obedece à propriedade max-heap.

4	1	10	14	16	9	3	2	8	7
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (6)



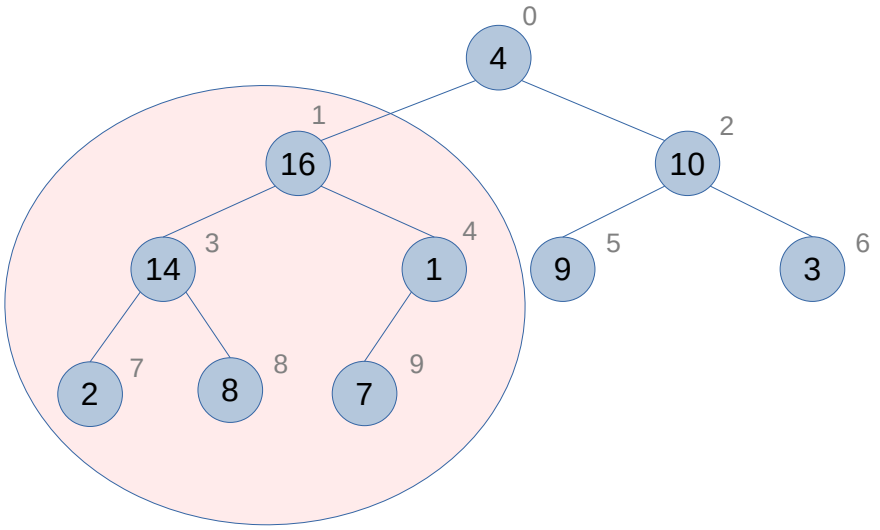
A verificação agora é da subárvore de raiz 1.

O valor da raiz será “afundado”: trocará de posição com a de seu maior filho (16).

4	1	10	14	16	9	3	2	8	7
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (7)



A verificação agora é da subárvore de raiz 1.

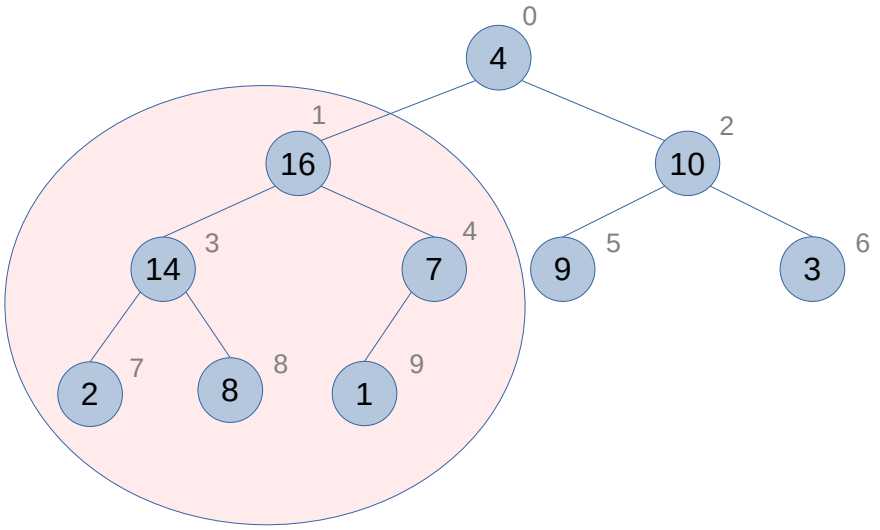
O valor da raiz será “afundado”: trocará de posição com a de seu maior filho (16).

Com isso, a subárvore de raiz 4 fere a propriedade max heap. O valor 1 trocará então de posição com a de seu maior filho (7).

4	16	10	14	1	9	3	2	8	7
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (8)



A verificação agora é da subárvore de raiz 1.

O valor da raiz será “afundado”: trocará de posição com a de seu maior filho (16).

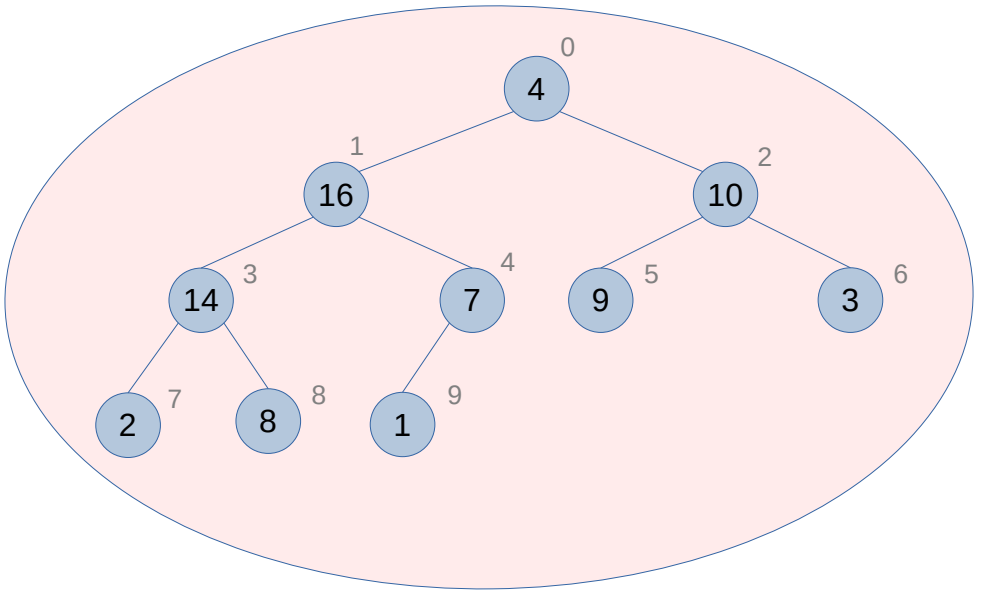
Com isso, a subárvore de raiz 4 fere a propriedade max heap. O valor 1 trocará então de posição com a de seu maior filho (7).

A subárvore de raiz 1 agora obedece à propriedade max-heap.

4	16	10	14	7	9	3	2	8	1
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (9)



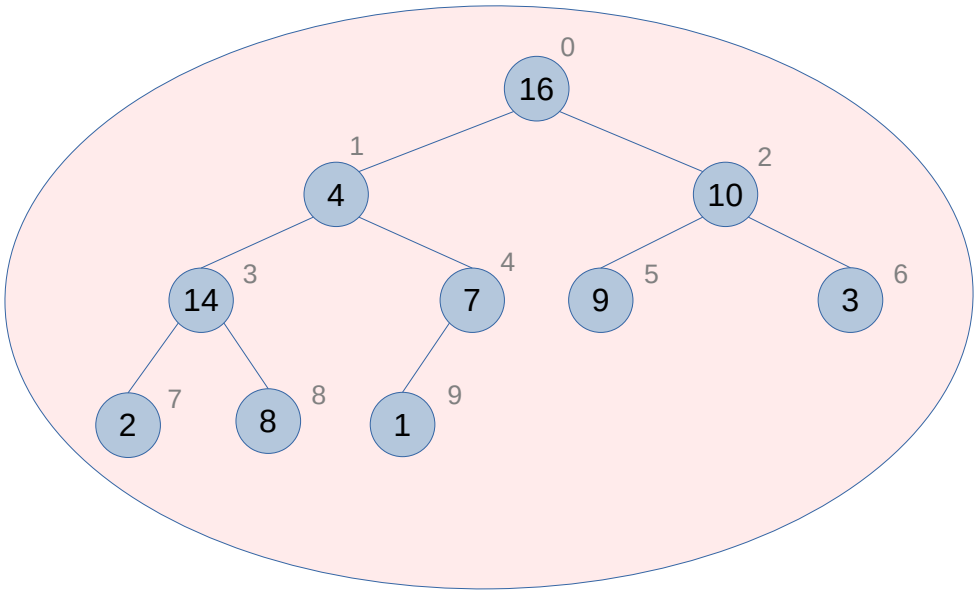
A verificação agora é da árvore de raiz 0.

O valor da raiz será “afundado”: trocará de posição com a de seu maior filho (16).

4	16	10	14	7	9	3	2	8	1
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (10)



A verificação agora é da árvore de raiz 0.

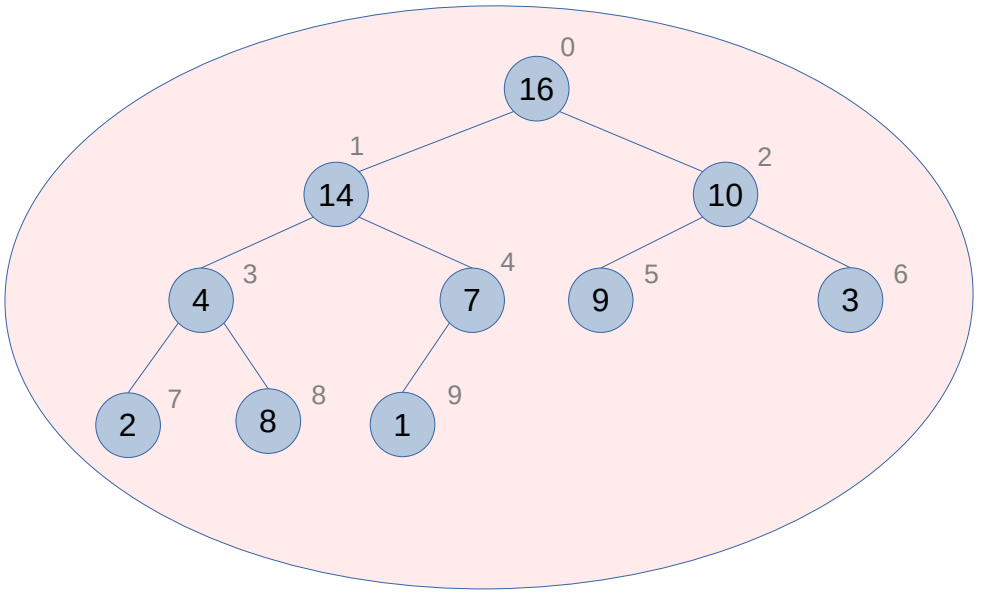
O valor da raiz será "afundado": trocará de posição com a de seu maior filho (16).

O valor 4 trocará então de posição com a de seu maior filho (14).

16	4	10	14	7	9	3	2	8	1
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (11)



A verificação agora é da árvore de raiz 0.

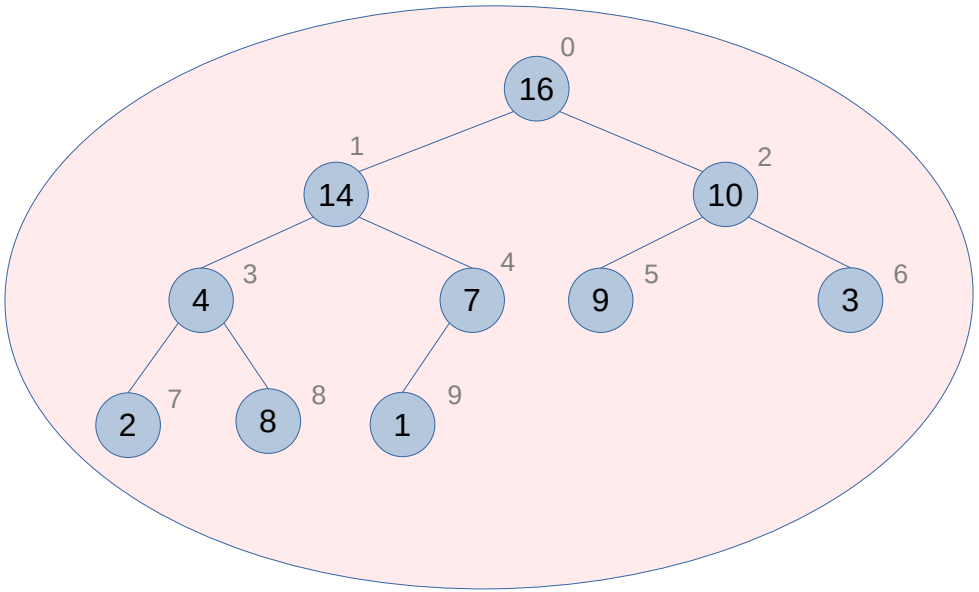
O valor da raiz será "afundado": trocará de posição com a de seu maior filho (16).

O valor 4 trocará então de posição com a de seu maior filho (14).

16	14	10	4	7	9	3	2	8	1
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (12)



A verificação agora é da árvore de raiz 0.

O valor da raiz será "afundado": trocará de posição com a de seu maior filho (16).

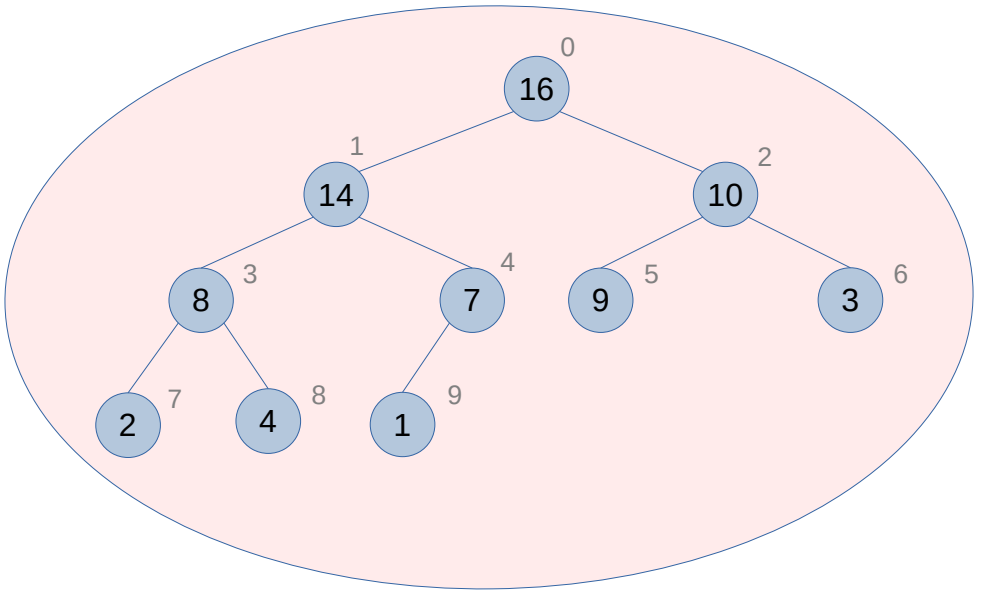
O valor 4 trocará então de posição com a de seu maior filho (14).

O valor 4 agora troca com 8.

16	14	10	4	7	9	3	2	8	1
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Construção do Max Heap (13)



A verificação agora é da árvore de raiz 0.

O valor da raiz será "afundado": trocará de posição com a de seu maior filho (16).

O valor 4 trocará então de posição com a de seu maior filho (14).

O valor 4 agora troca com 8.

Agora toda a árvore obedece à propriedade max heap.

16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9

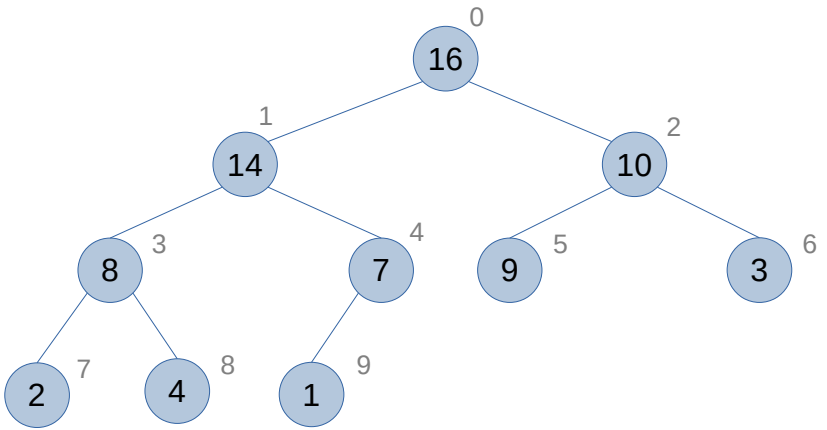
Heap Sort

- **Fase 2**: seleção dos elementos na ordem
  - A partir do heap construído anteriormente, vamos selecionar os elementos na ordem desejada.
  - Se a chave que está na raiz (A[0]) é a maior de todas, então sua posição definitiva correta, na ordem crescente, é na última posição do vetor.
  - Deste modo, trocamos A[0] com o elemento que está na última posição do vetor (A[n-1]).
  - A seguir, reconstruímos o heap para os elementos A[0], …, A[n-2].
    - O último elemento, que acabou de ser ordenado, ficará de fora.
    - Ou seja, o vetor ficará dividido em um setor ordenado (lado direito) e um setor não ordenado (lado esquerdo).
  - Estes 2 passos (troca do elemento raiz com o último do setor não ordenado e reconstrução do heap para os elementos restantes) são então repetidos para os n-2, n-3, … elementos, até que reste apenas 1 elemento. A cada passada, o setor ordenado do array aumenta, restando menos elementos não ordenados, até que reste apenas um elemento.

Heap Sort

Como funciona – Seleção de elementos na ordem (1)

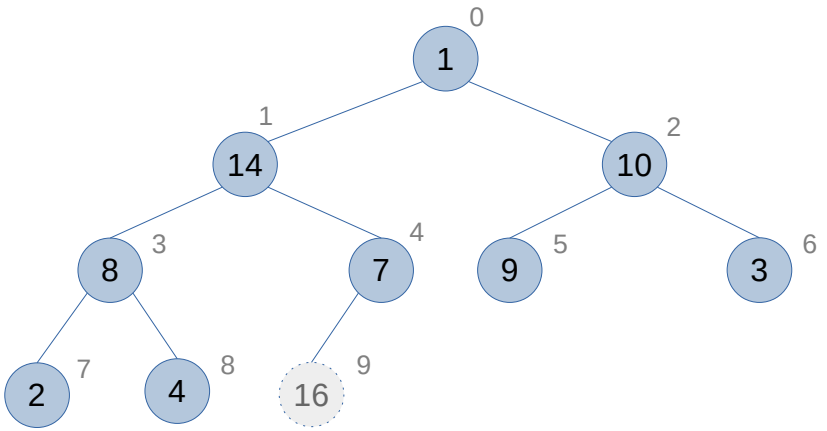
O maior elemento (16) trocará de posição com o último (1)



16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (2)



O maior elemento (16) trocará de posição com o último (1)

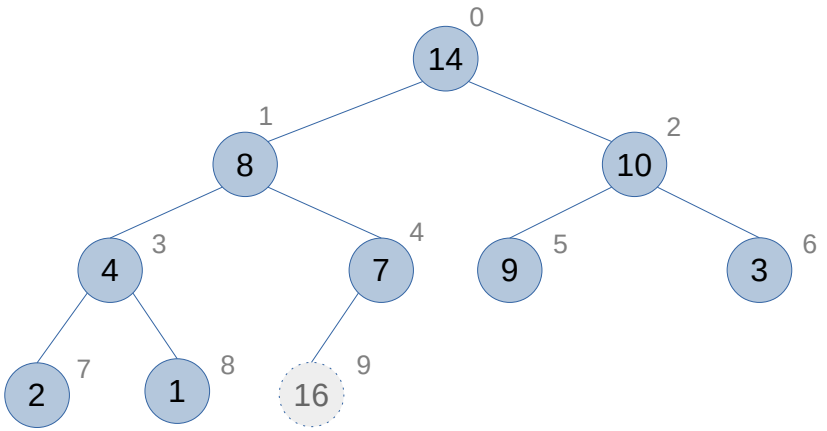
O valor 16 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando a posição 9).

1	14	10	8	7	9	3	2	4	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (3)



O maior elemento (16) trocará de posição com o último (1)

O valor 16 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando a posição 9).

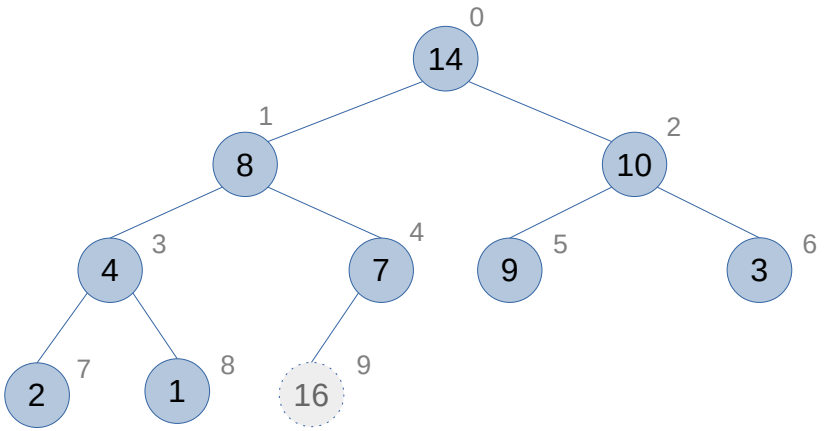
A árvore agora voltou a ser um max heap.

14	8	10	4	7	9	3	2	1	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (4)

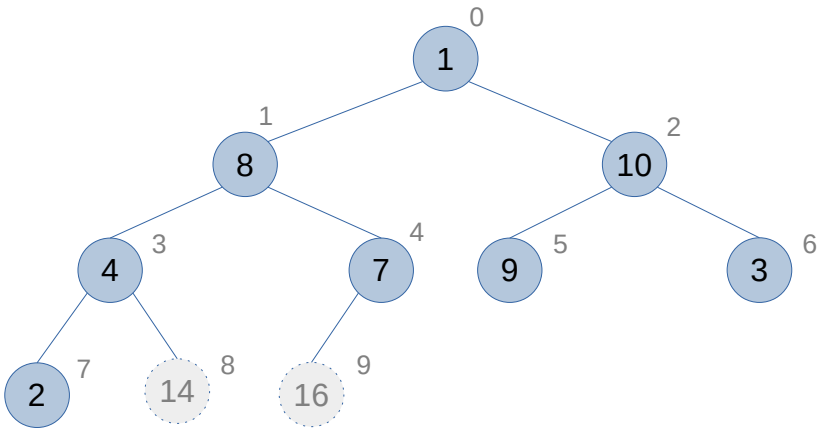
O maior elemento (14) trocará de posição com o último não ordenado (1)



14	8	10	4	7	9	3	2	1	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (5)



O maior elemento (14) trocará de posição com o último não ordenado (1).

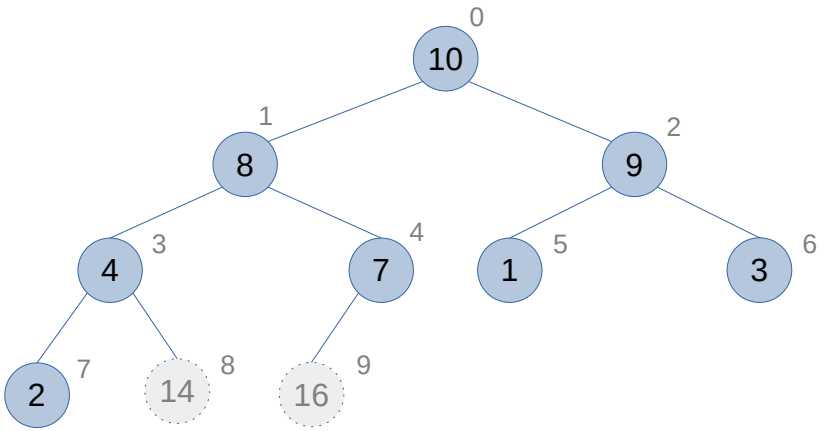
O valor 14 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 8 e 9).

1	8	10	4	7	9	3	2	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (6)



O maior elemento (14) trocará de posição com o último não ordenado (1).

O valor 14 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 8 e 9).

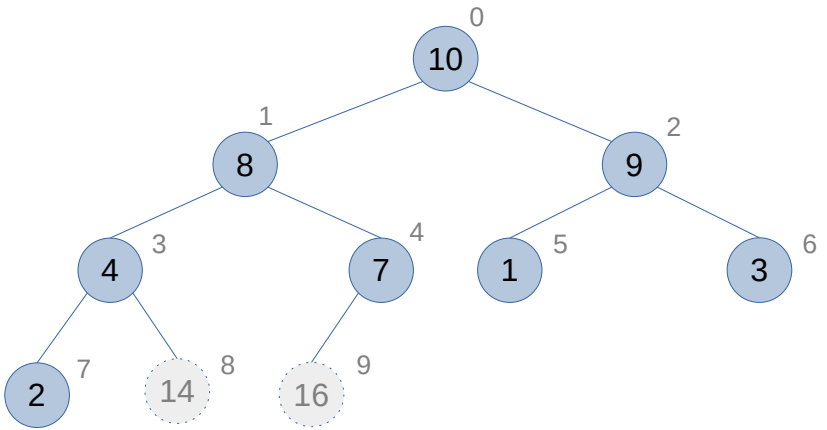
A árvore agora voltou a ser um max heap.

10	8	9	4	7	1	3	2	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (7)

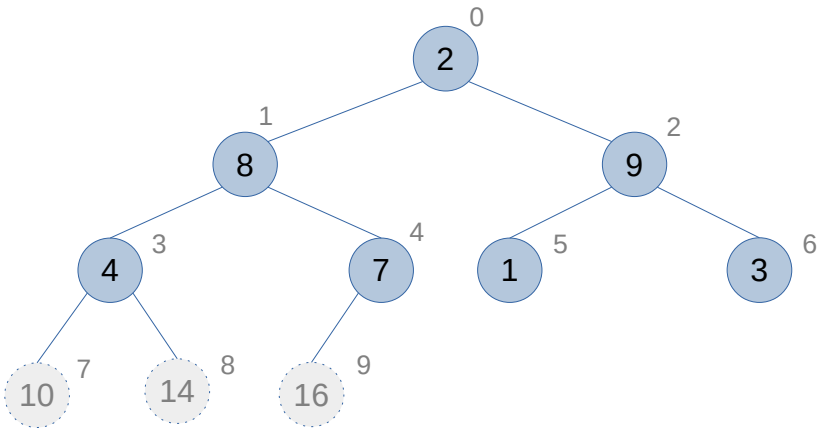
O maior elemento (10) trocará de posição com o último não ordenado (2).



10	8	9	4	7	1	3	2	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (8)



O maior elemento (10) trocará de posição com o último não ordenado (2).

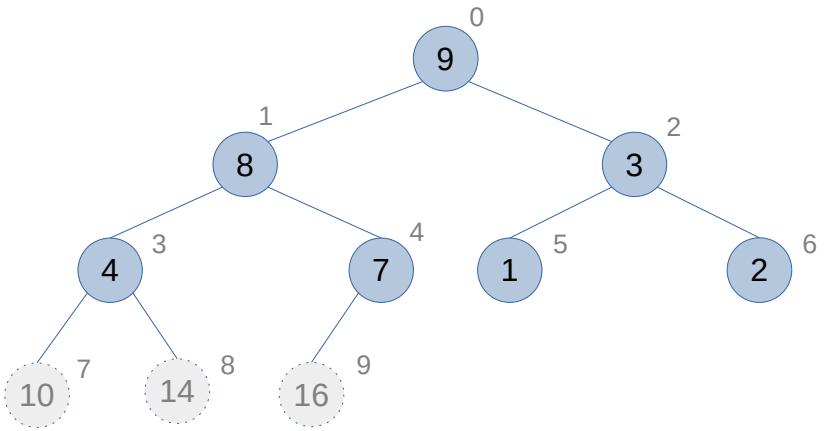
O valor 10 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 7 a 9).

2	8	9	4	7	1	3	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (9)



O maior elemento (10) trocará de posição com o último não ordenado (2).

O valor 10 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 7 a 9).

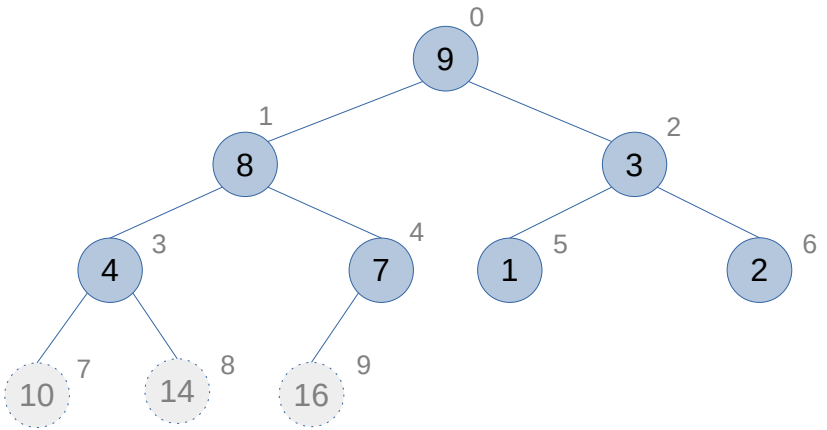
A árvore agora voltou a ser um max heap.

9	8	3	4	7	1	2	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (10)

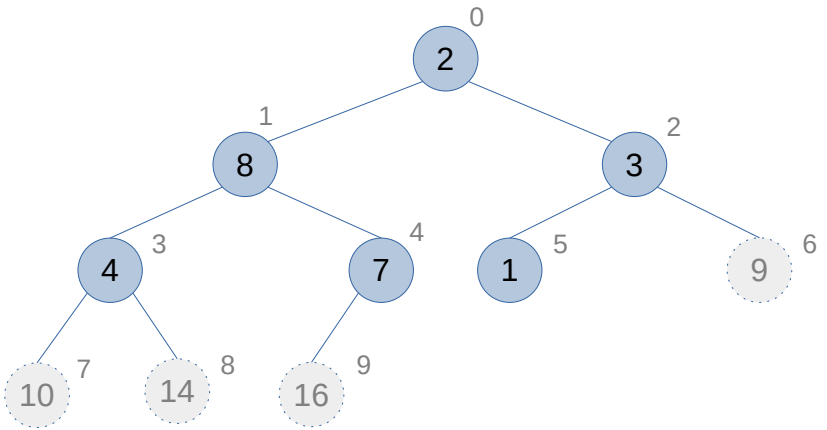
O maior elemento (9) trocará de posição com o último não ordenado (2).



9	8	3	4	7	1	2	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (11)



O maior elemento (9) trocará de posição com o último não ordenado (2).

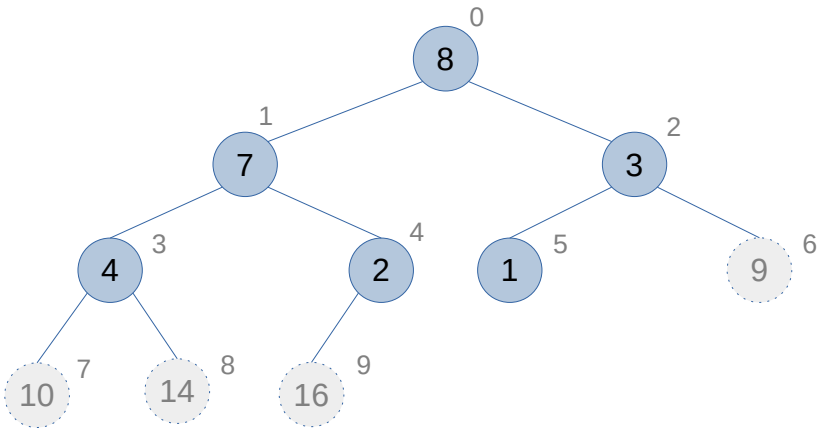
O valor 9 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 6 a 9).

2	8	3	4	7	1	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (12)



O maior elemento (9) trocará de posição com o último não ordenado (2).

O valor 9 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 6 a 9).

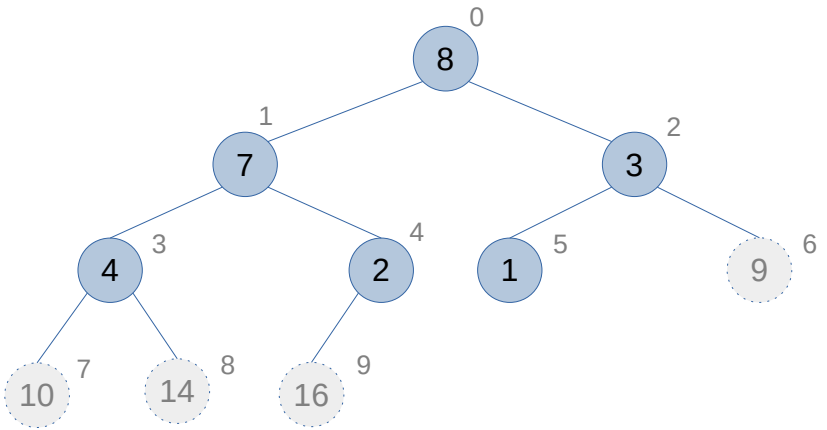
A árvore agora voltou a ser um max heap.

8	7	3	4	2	1	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (13)

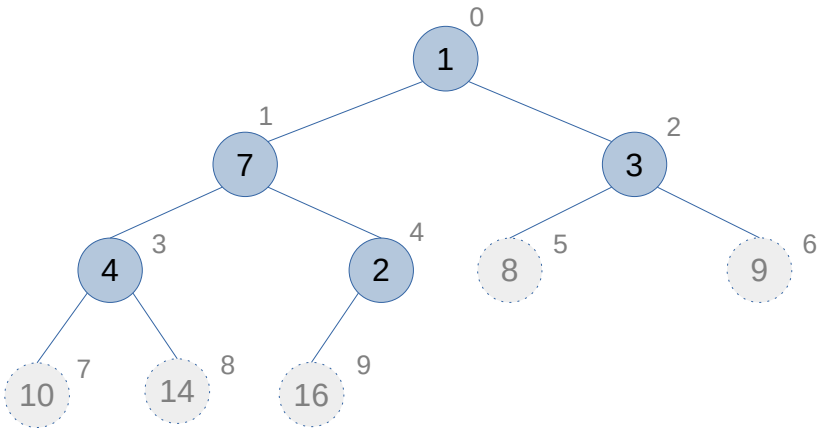
O maior elemento (8) trocará de posição com o último não ordenado (1).



8	7	3	4	2	1	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (14)



O maior elemento (8) trocará de posição com o último não ordenado (1).

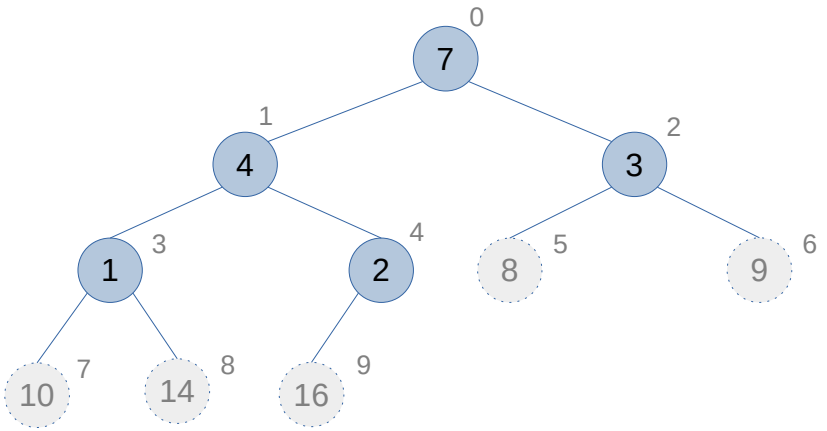
O valor 8 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 5 a 9).

1	7	3	4	2	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (15)



O maior elemento (8) trocará de posição com o último não ordenado (1).

O valor 8 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 5 a 9).

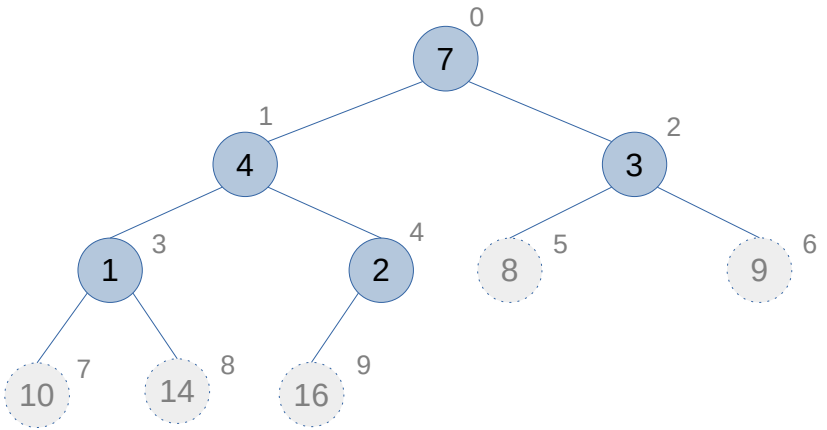
A árvore agora voltou a ser um max heap.

7	4	3	1	2	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (16)

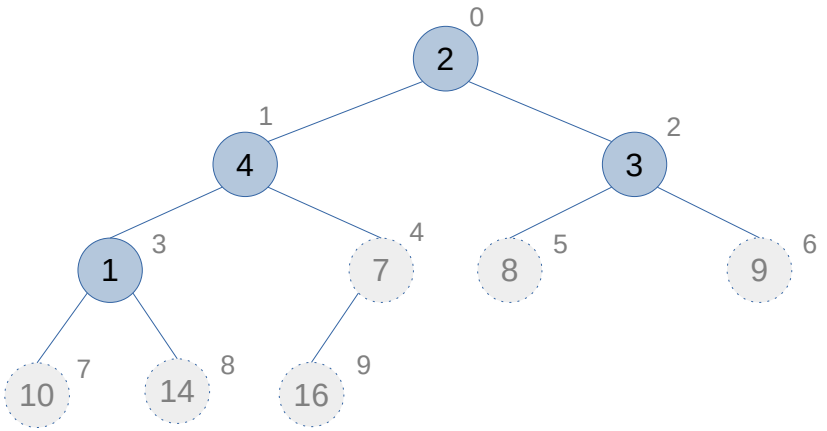
O maior elemento (7) trocará de posição com o último não ordenado (2).



7	4	3	1	2	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (17)



O maior elemento (7) trocará de posição com o último não ordenado (2).

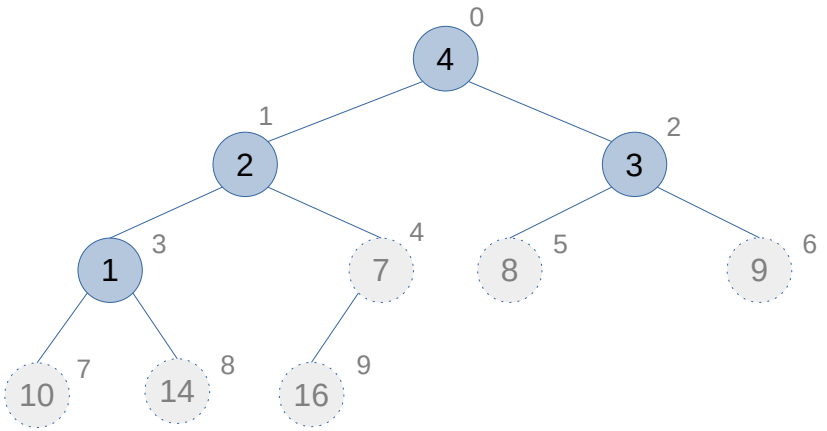
O valor 7 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 4 a 9).

2	4	3	1	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (18)



O maior elemento (7) trocará de posição com o último não ordenado (2).

O valor 7 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 4 a 9).

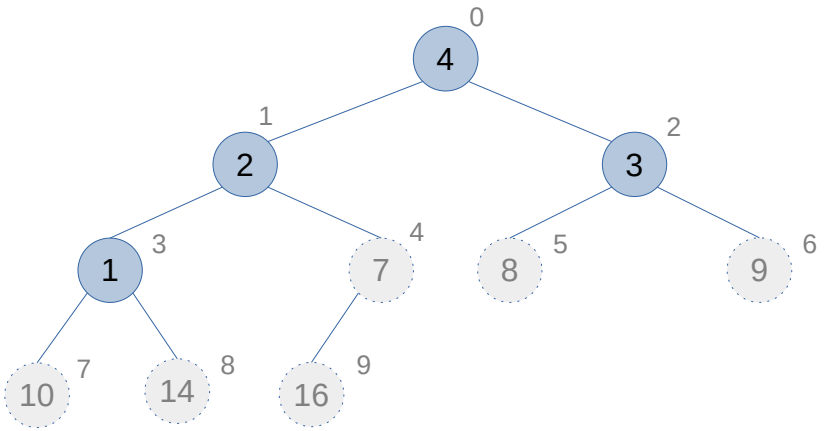
A árvore agora voltou a ser um max heap.

4	2	3	1	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (19)

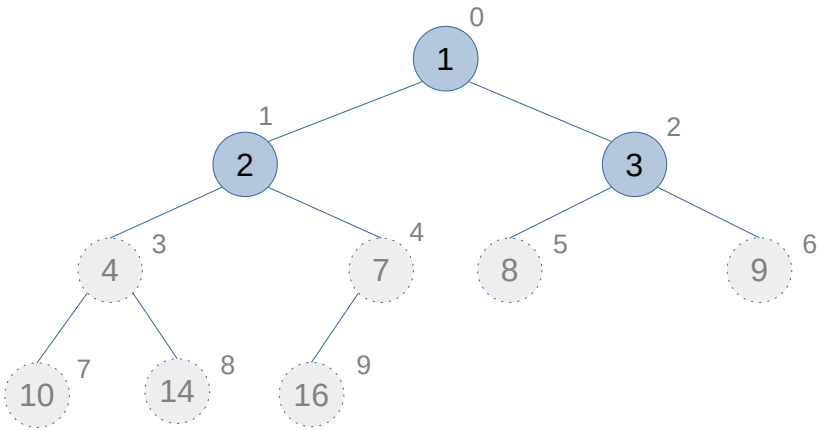
O maior elemento (4) trocará de posição com o último não ordenado (1).



4	2	3	1	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (20)



O maior elemento (4) trocará de posição com o último não ordenado (1).

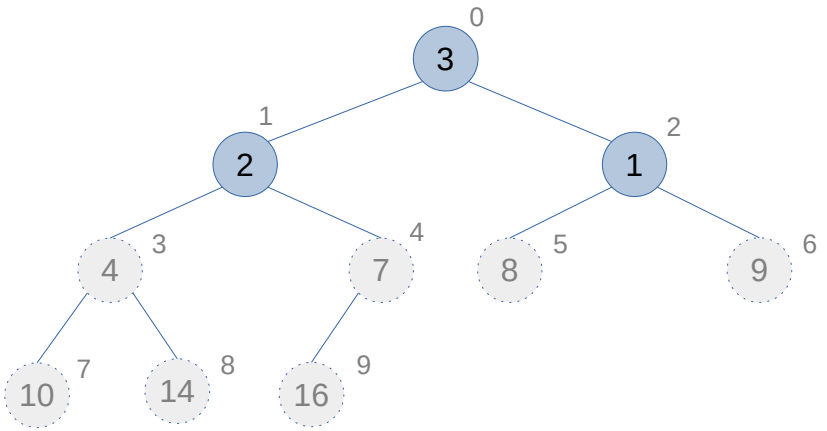
O valor 4 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 3 a 9).

1	2	3	4	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (21)



O maior elemento (4) trocará de posição com o último não ordenado (1).

O valor 4 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 3 a 9).

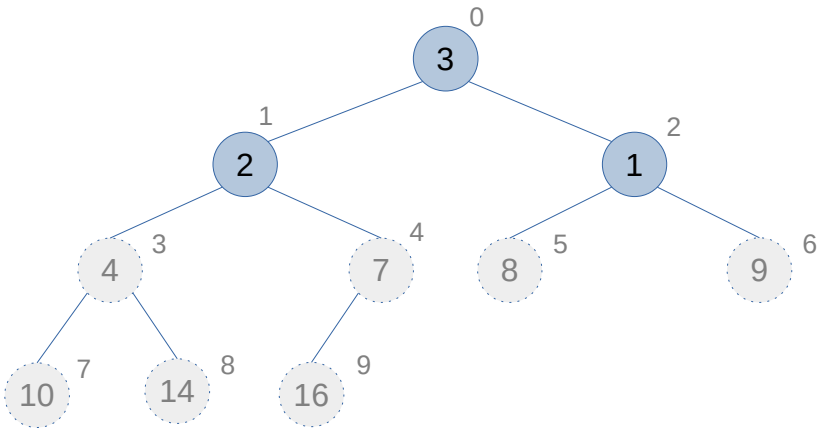
A árvore agora voltou a ser um max heap.

3	2	1	4	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (22)

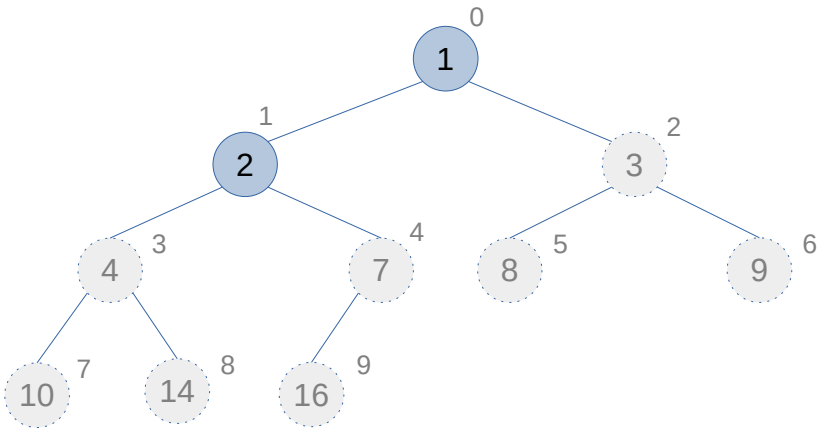
O maior elemento (3) trocará de posição com o último não ordenado (1).



3	2	1	4	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (23)



O maior elemento (3) trocará de posição com o último não ordenado (1).

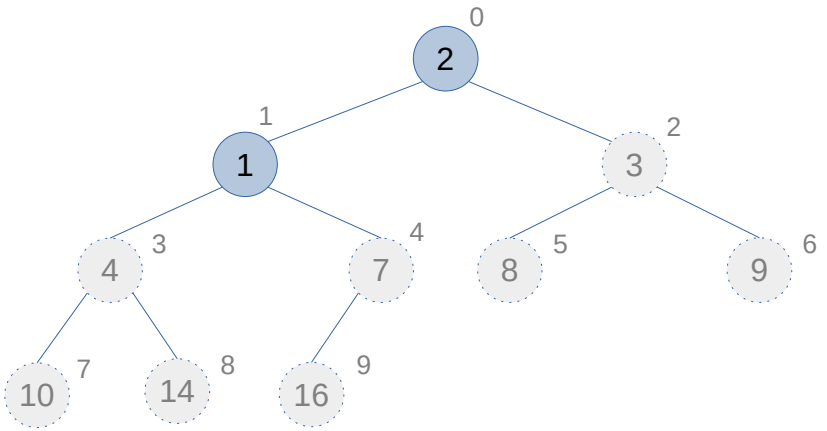
O valor 3 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 2 a 9).

1	2	3	4	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (24)



O maior elemento (3) trocará de posição com o último não ordenado (1).

O valor 3 agora integra o setor ordenado e não será mais movimentado.

A árvore resultante não é mais um max heap e, portanto, deverá ser reorganizada (desconsiderando as posições 2 a 9).

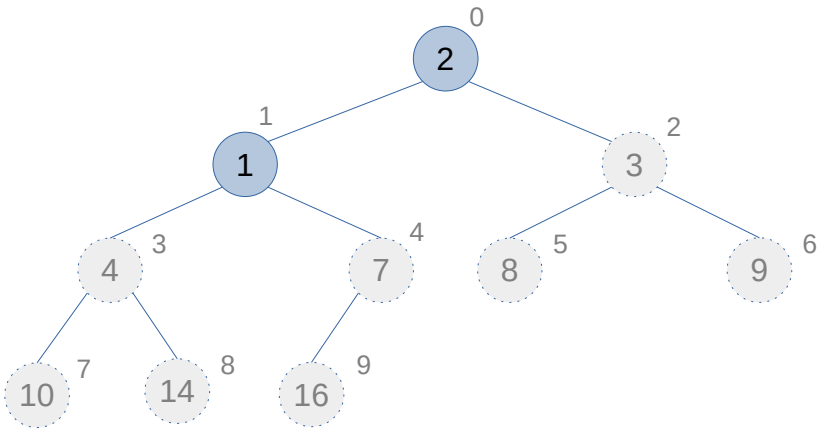
A árvore agora voltou a ser um max heap.

2	1	3	4	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (25)

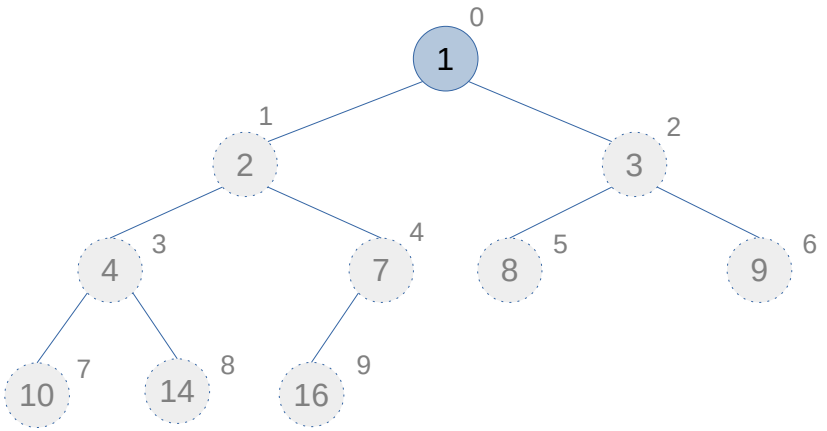
O maior elemento (2) trocará de posição com o último não ordenado (1).



1	2	3	4	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort

Como funciona – Seleção de elementos na ordem (26)



O maior elemento (2) trocará de posição com o último não ordenado (1).

O valor 2 agora integra o setor ordenado e não será mais movimentado.

Restou apenas 1 elemento, portanto, a ordenação está concluída.

1	2	3	4	7	8	9	10	14	16
0	1	2	3	4	5	6	7	8	9

Heap Sort
Implementação

- O algoritmo será implementado em duas partes:
  - **heapSort**
    - Função principal que será chamada pelos demais programas
    - Responsável por criar a estrutura heap a partir dos dados presentes no vetor (fase 1), bem como pela remoção do maior elemento (e seu posicionamento ao final do setor não ordenado) e por chamar a reconstrução do heap (fase 2)
    - Recebe como parâmetro o vetor e seu tamanho
  - **criaHeap**
    - Função auxiliar responsável por verificar e garantir a propriedade max heap para uma subárvore
    - Recebe como parâmetro o vetor, a posição da raiz da subárvore que está sendo tratada e a posição final do segmento

Heap Sort
Implementação

```
função heapSort(A[ ], n)
inicio
    para k de n/2-1 ate 0 faca    /* fase 1: constrói o heap máximo */
        criaHeap(A, k, n)
    fimPara

    para k de n-1 ate 1 faca    /* fase 2: seleciona o maior, posiciona-o e reconstrói o heap máximo */
        troca(A[0], A[k])
        criaHeap(A, 0, k)
    fimPara
fim
```

Para todas as subárvores com filhos, da última para a primeira

Organiza o heap daquela subárvore

Reorganiza o heap deixando os já ordenados de fora

Heap Sort
Implementação

```
função criaHeap(A[ ], i, n)
inicio
    maior ← i
    left ← 2 * i + 1     /* filho da esquerda de i */
    right ← 2 * i + 2    /* filho da direita de i */
    se  left < n  E  A[left] > A[i]  então
        maior ← left
    fimSe
    se  right < n  E  A[right] > A[maior]  então
        maior ← right
    fimSe

    se  maior != i  então
        troca(A[i], A[maior])
        criaHeap(A, maior, n)
     fimSe
fim
```

Filho da esquerda existe e é maior do que o pai

Filho da direita existe e é maior do que o pai ou que o filho da esquerda

Raiz não é o maior valor; troca com o maior filho

Aplica recursivamente para as subárvores

Heap Sort
Análise

- A complexidade do Heap Sort, em qualquer caso, é sempre O(n log n).
  - Fase de construção do heap: número linear de comparações e trocas
  - Fase de ordenação: N operações de *heapfy down*, sendo que a altura é de no máximo log N
- O(n log n) no pior caso → vantagem sobre o Quick Sort
- In place → vantagem sobre o Merge Sort
- Não estável