Árvore Binária

Vs

Heap

Conceito geral de *heap*:

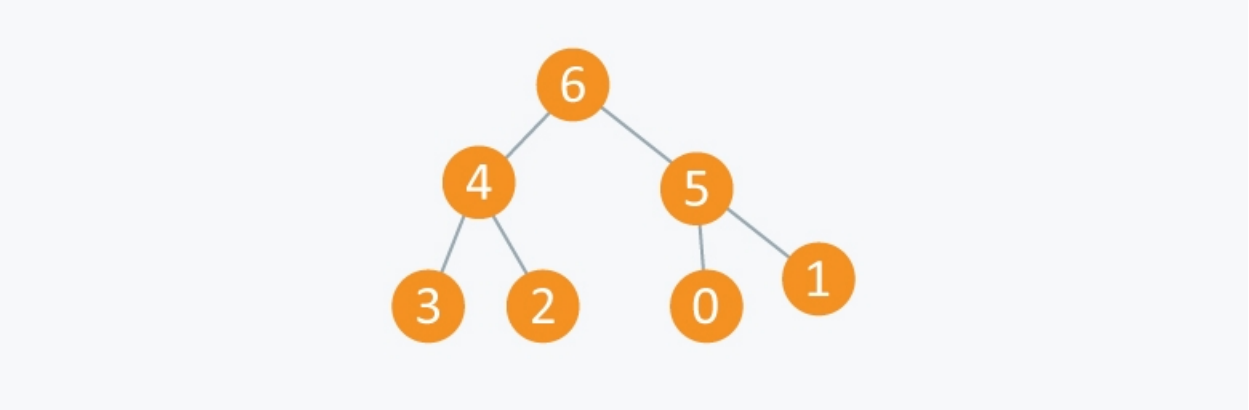
Um heap é uma estrutura baseada em uma árvore binária contendo nós em uma ordem característica.

Se o heap for uma árvore binária completa com N nós, então ele terá a menor altura possível, qual seja: $h = \log_2 N$, o que é particularmente ótimo.

Existe uma mesma ordem seguida em toda a árvore-heap:

- Se X é pai de Y, então o valor de X segue uma ordem específica em relação ao valor de Y;

- No exemplo abaixo temos um max-heap: cada nó tem valor maior do que qualquer um de seus filhos:



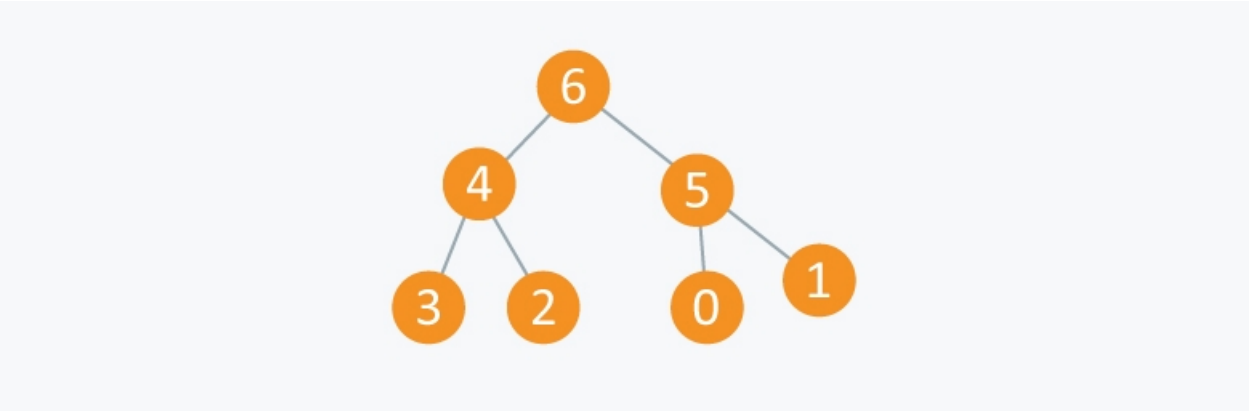
Conceito geral de *heap*:

O heap é usualmente em um vetor tirando proveito da indexação.

Heaps podem se apresentar na forma max-heap e min-heap.

No exemplo de max-heap abaixo, descartou-se a posição zero do vetor:

filhoEsq(i) = 2i
filhoDir(i) = 2i+1
pai(i) = quociente(i/2) ← divisão inteira



informação		6	4	5	3	2	0	1
índice	0	1	2	3	4	5	6	7

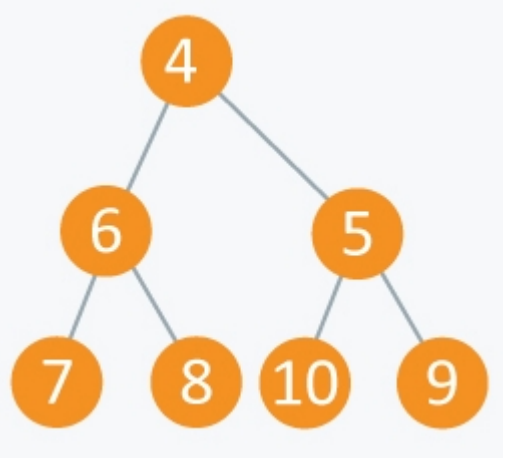
Conceito geral de *heap*:

O heap é usualmente em um vetor tirando proveito da indexação.

Heaps podem se apresentar na forma max-heap e min-heap.

No exemplo de min-heap abaixo, descartou-se a posição zero do vetor:

filhoEsq(i) = 2i
filhoDir(i) = 2i+1
pai(i) = quociente(i/2) ← divisão inteira



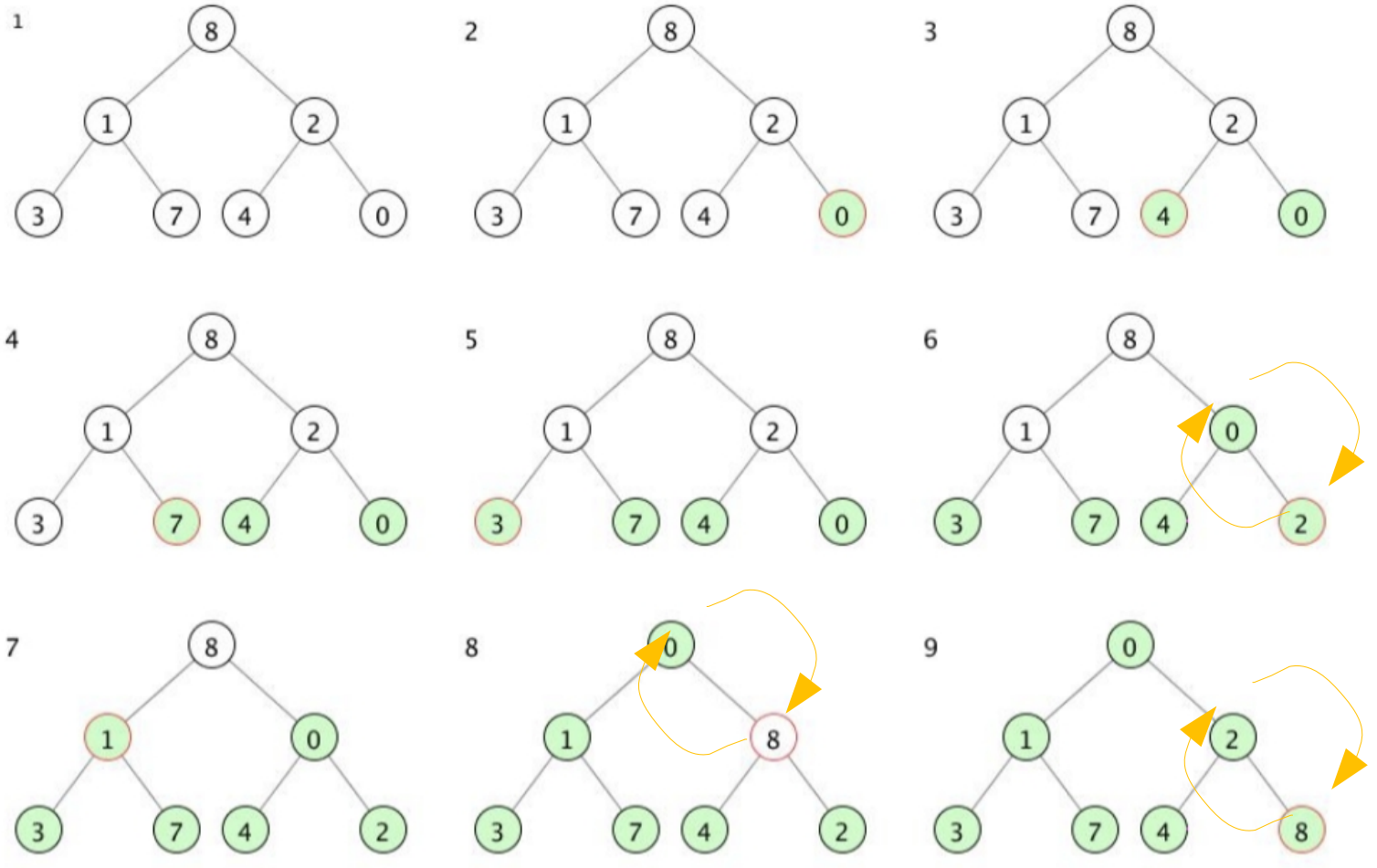
informação		4	6	5	7	8	10	9
índice	0	1	2	3	4	5	6	7

A operação mais importante no heap é a “*heapfication*”:

- Consiste na construção do heap a partir de um conjunto de valores arbitrários e desordenados em um array. A seguir temos o exemplo aplicado em uma min-heap:

Bottom-up Heapification - $O(N)$ [1]

Green indicates completed nodes, highlighting the bottom-up effect. Orange outline reflects the bubbled node.

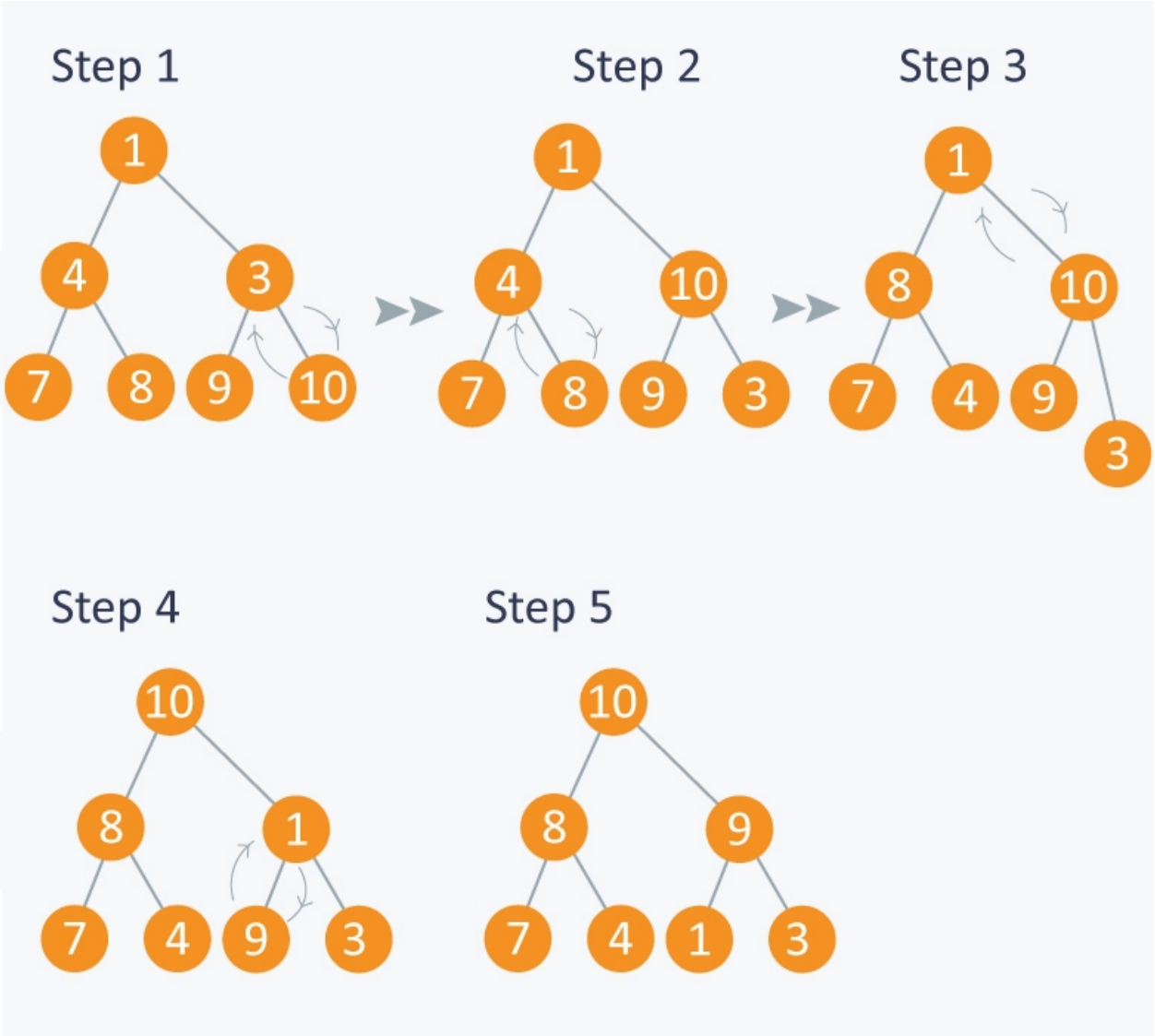


https://hackmd.io/@tinazhao/SJAkdo0s7

Exemplo de *heapfication* no **max**-heap cotendo 7 elementos armazenados no vetor Arr.

- **Max** Heap: o valor do nó pai sempre será maior ou igual ao valor do nó filho na árvore e o nó com maior valor será o nó raiz da árvore.

filhoEsq(i) = 2i
filhoDir(i) = 2i+1
pai(i) = quociente(i/2) ← divisão inteira



Exemplo de *heapfication* no **min**-heap cotendo 7 elementos armazenados no vetor Arr.

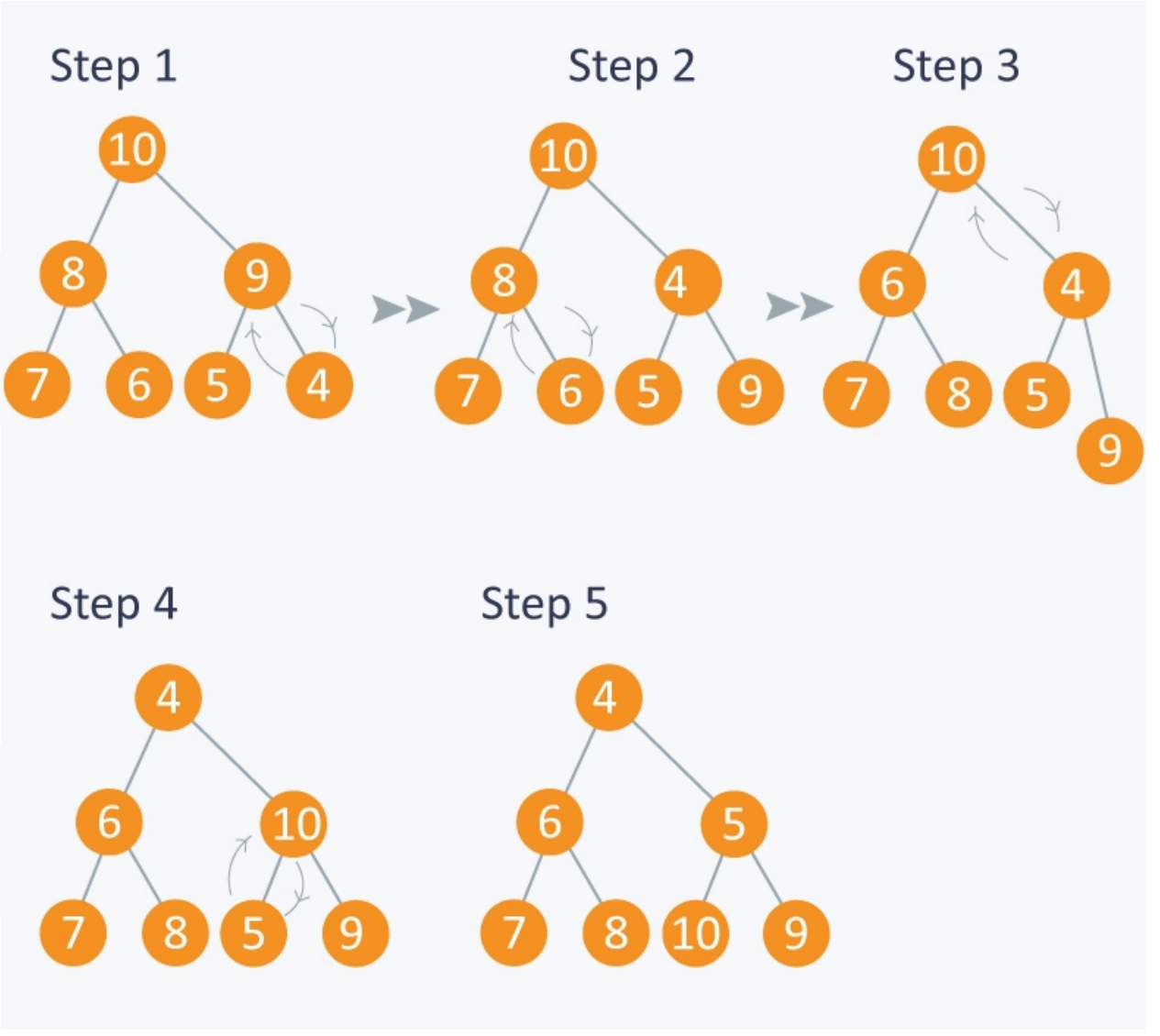
- **Min** Heap: o valor do nó pai sempre será menor ou igual ao valor do nó filho na árvore e o nó com menor valor será o nó raiz da árvore.
- No diagrama abaixo: executamos praticamente as mesmas operações realizadas na construção de max_heap, porém, mantendo a regra de cada nó ter um valor menor que o valor de seus filhos.

filhoEsq(i) = 2i
filhoDir(i) = 2i+1
pai(i) = quociente(i/2) ← divisão inteira

Arr		10	8	9	7	6	5	4
	0	1	2	3	4	5	6	7

heapfication

Arr		4	6	5	7	8	10	9
	0	1	2	3	4	5	6	7



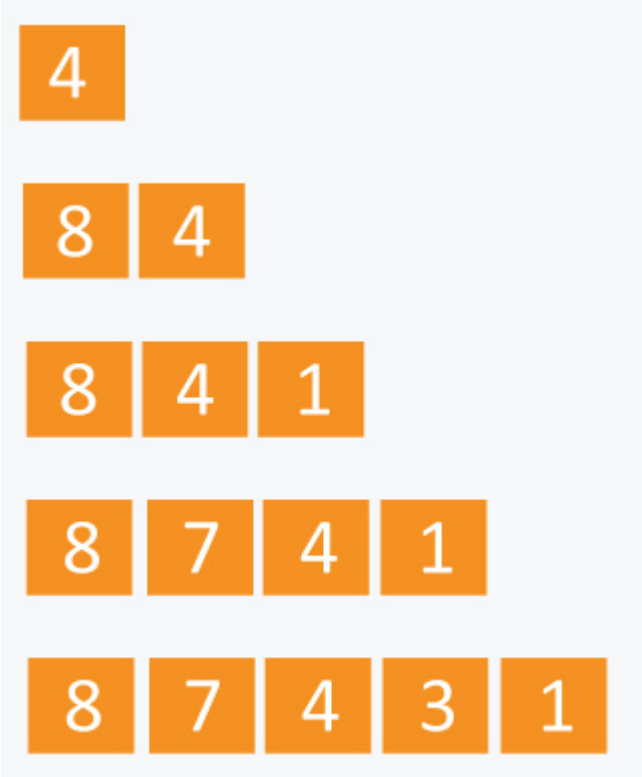
Aplicações:

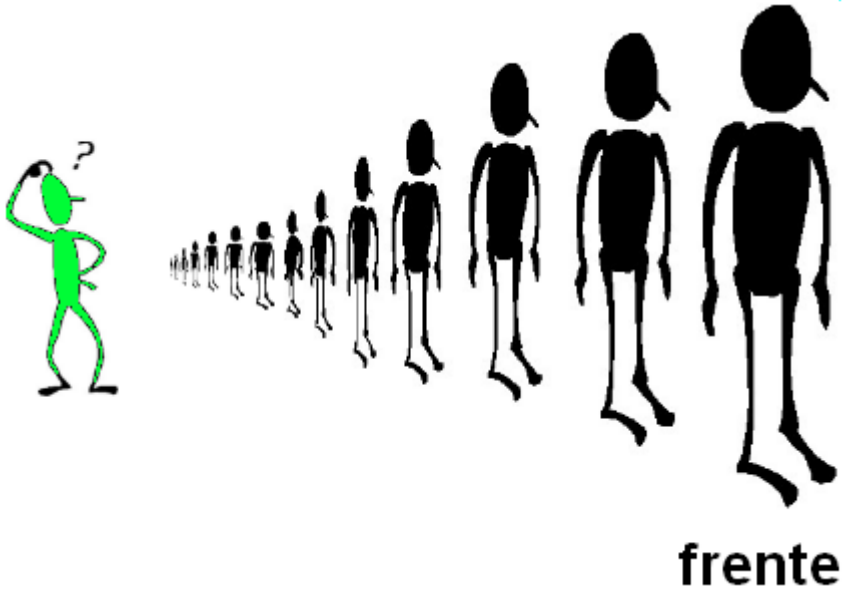
1) Priority Queue – Heap:

A ordem lógica dos elementos na fila de prioridade depende da prioridade dos elementos.

O elemento com maior prioridade será movido para a frente da fila e aquele com menor prioridade será movido para o final da fila.

Portanto, é possível que, ao enfileirar um elemento ele possa ser movido para a frente devido à sua prioridade mais alta.





Aplicações:

1) Priority Queue – Heap:

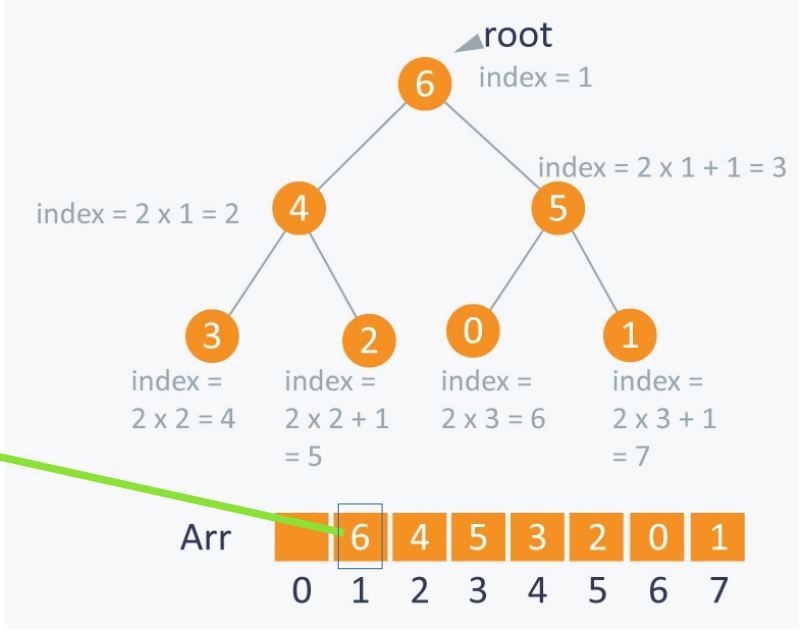
Utilizamos a lista para implementar a fila de prioridade, mas a abordagem mas eficiente é por meio de uma Árvore Binária Heap, a qual levará $O(\log_2 N)$ tempo para inserir e excluir cada elemento na fila de prioridade.

Com base na estrutura do heap, a fila de prioridade também possui dois tipos: fila de prioridade máxima e fila de prioridade mínima.

Vamos nos concentrar na fila de prioridade máxima, baseada na estrutura do heap máximo.

filhoEsq(i) = 2i
filhoDir(i) = 2i+1
pai(i) = quociente(i/2) ← divisão inteira

A máxima prioridade sempre na raiz Arr[1]



Árvore Heap é uma alternativa ótima como:

1) Fila de prioridade implementada em um vetor:

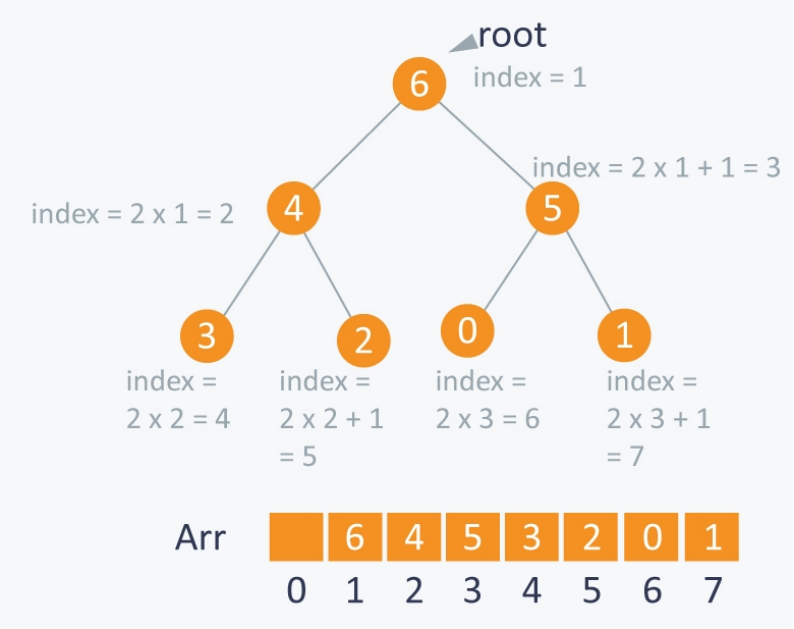
Suponha que:

- Haja uma fila de prioridade para exatamente M processos a serem executados (cada processo com sua própria prioridade);
- Sempre que se conclui um processo de prioridade máxima, se insere na fila um novo processo com prioridade própria.

Esta tarefa pode ser facilmente executada usando um heap-binário implementado sobre um vetor/array (tirando proveito da rapidez da indexação), considerando M processos como N nós folha na árvore de heap.

Digamos que temos 7 elementos com prioridades {6, 4, 5, 3, 2, 0, 1}.

filhoEsq(i) = 2i
filhoDir(i) = 2i+1
pai(i) = quociente(i/2) ← divisão inteira



Árvore Heap é uma alternativa ótima como:

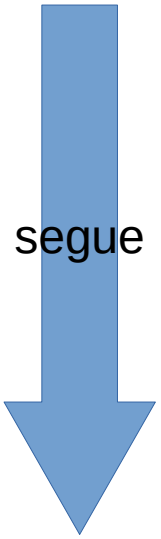
1) Fila de prioridade implementada em um vetor:

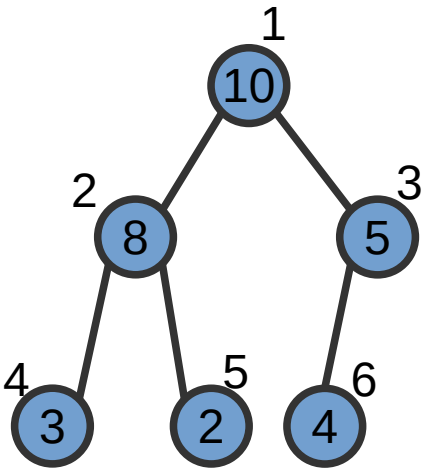
Suponha que:

- Haja uma fila de prioridade para exatamente M processos a serem executados (cada processo com sua própria prioridade);
- Sempre que se conclui um processo de prioridade máxima, se insere na fila um novo processo com prioridade própria.

filhoEsq(i) = 2i
filhoDir(i) = 2i+1
pai(i) = quociente(i/2) ← divisão inteira

prioridade		10	8	5	3	2	4
índice	0	1	2	3	4	5	6





Árvore Heap é uma alternativa ótima como:

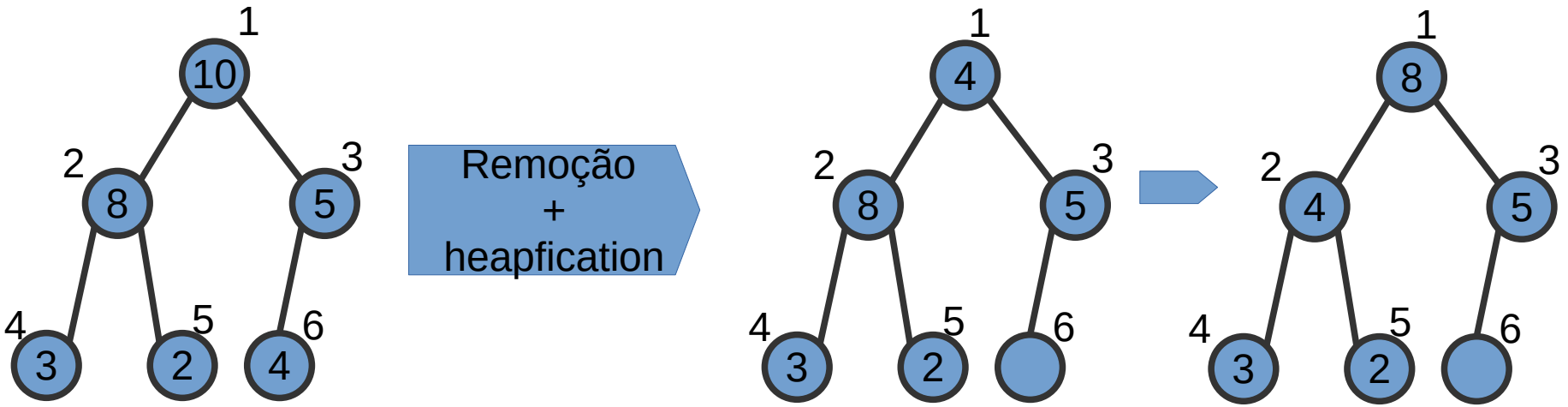
1) Fila de prioridade implementada em um vetor:

Remoção:
Troque o valor do nó raiz (índice zero) pelo valor do último nó no heap (índice N). Remova o último nó. Em seguida, se necessário, reduza o nó max/min para manter a propriedade de heap (faça a heapfication).

prioridade		10	8	5	3	2	4
índice	0	1	2	3	4	5	6

Remoção (10) e *heapfication*

prioridade		8	4	5	3	2	
índice	0	1	2	3	4	5	6

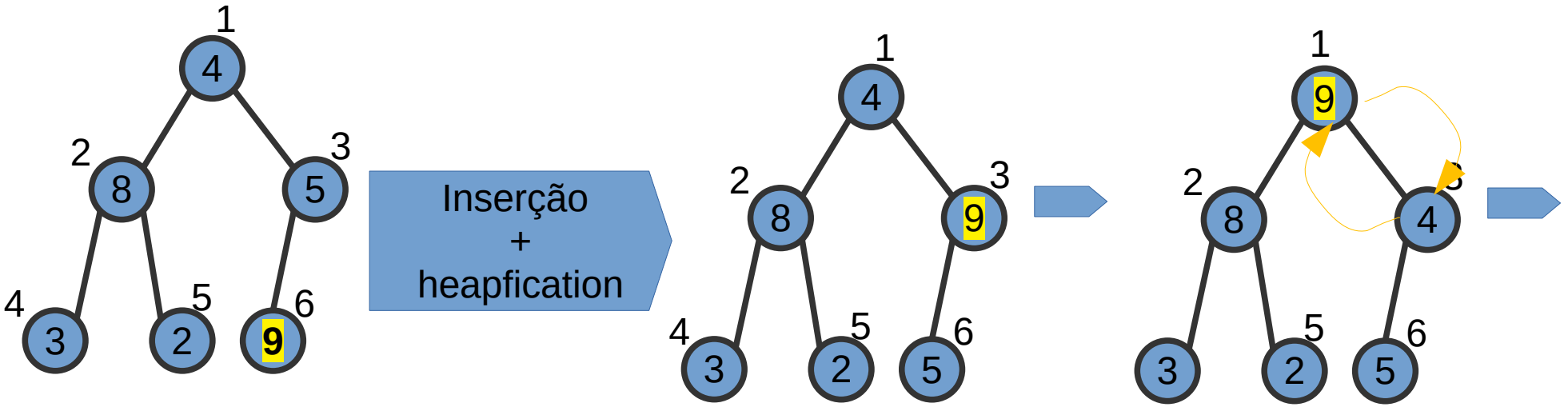


Árvore Heap é uma alternativa ótima como:

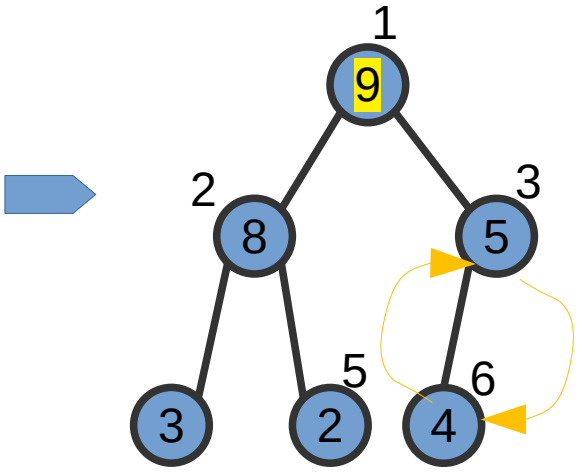
1) Fila de prioridade implementada em um vetor:

Inserção de prioridade *9*:

prioridade		8	4	5	3	2	**9**
índice	0	1	2	3	4	5	6

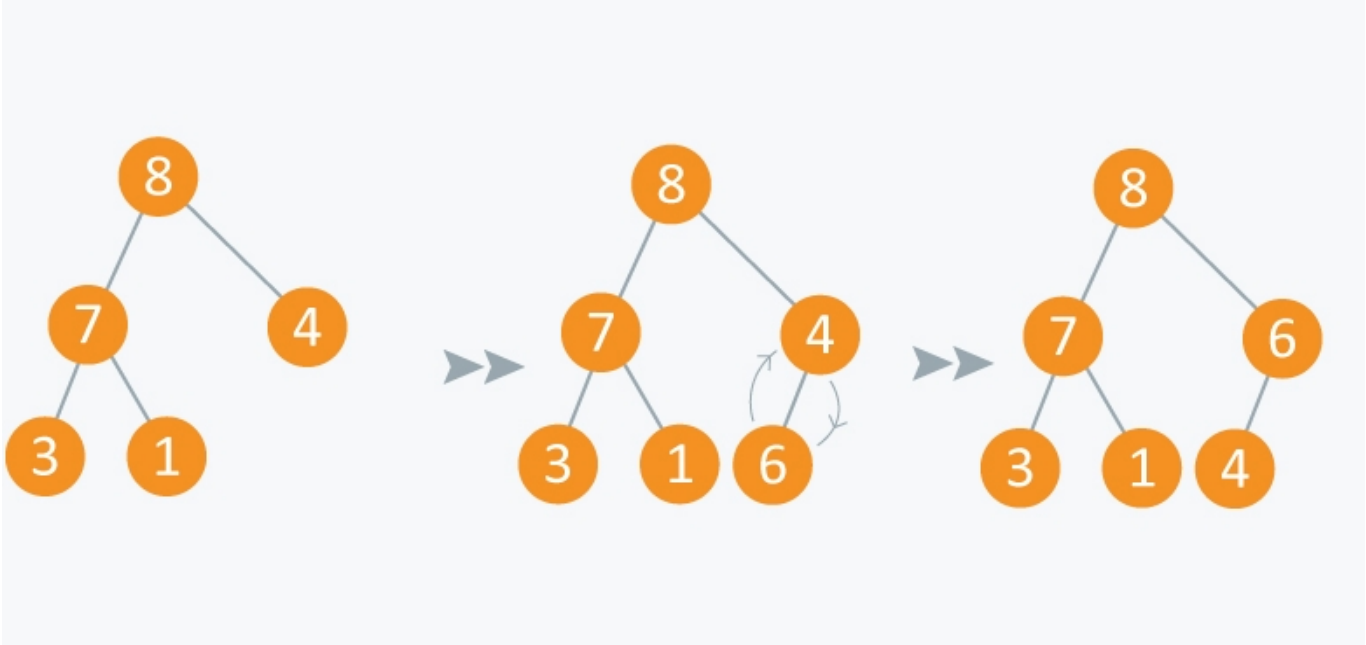


prioridade		9	8	5	3	2	4
índice	0	1	2	3	4	5	6



Aplicações:

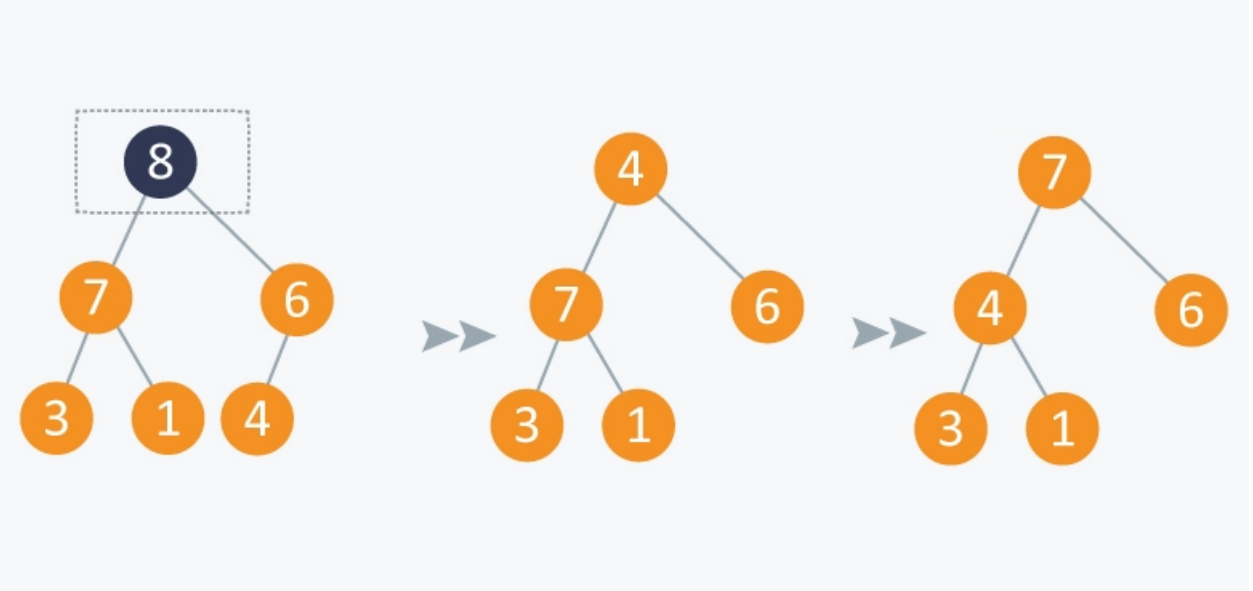
Outro exemplo INSERÇÃO:



Aplicações:

Outro exemplo REMOÇÃO:

A remoção é realizada na raiz da árvore heap.



Aplicações:

1) Heap-Sort:

Ordenação dos elementos de um vetor em tempo eficiente.

Ideia: Construímos o heap máximo de elementos armazenados, o elemento máximo do vetor sempre estará na raiz do heap.

a) Inicialmente, construiremos uma pilha máxima de elementos em Arr.

b) Agora o elemento raiz que é Arr[1] contém elemento máximo. Depois disso, trocaremos esse elemento pelo último elemento de Arr e construiremos novamente um heap máximo excluindo o último elemento que já está em sua posição correta. Depois disso decrementamos o comprimento do heap.

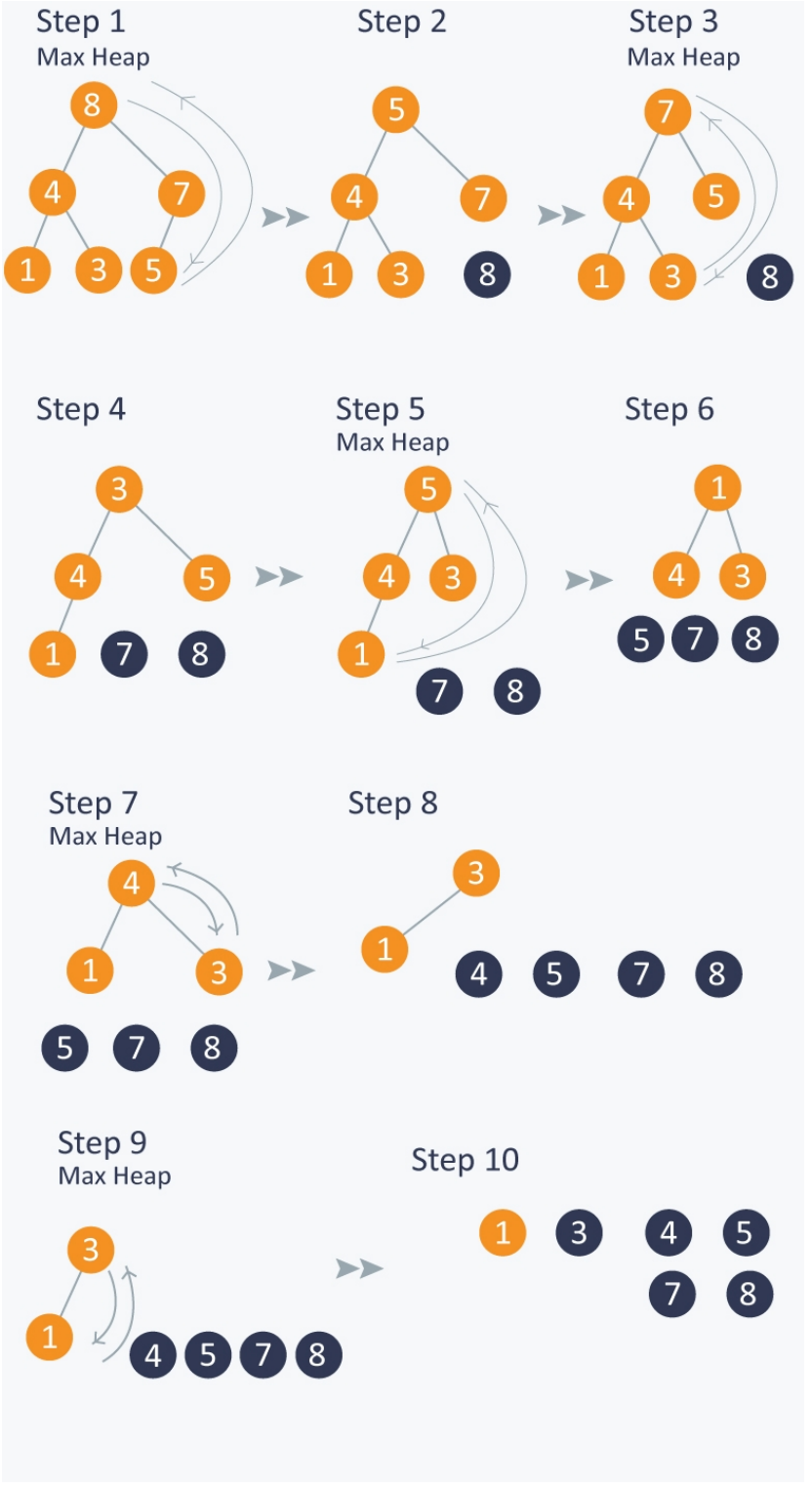
c) Iremos repetir o passo 2, até colocarmos todos os elementos na sua posição correta.

d) Teremos um array ordenado.

Aplicações:

1) Heap-Sort: partição ordenada crescente (valores maiores ficam mais à direita no vetor):

Arr		8	4	7	1	3	5
	0	1	2	3	4	5	6
	7	4	5	1	3		8
0	1	2	3	4	5		6
	3	4	5	1		7	8
0	1	2	3	4		5	6
	1	4	3		5	7	8
0	1	2	3		4	5	6
	3	1		4	5	7	8
0	1	2		3	4	5	6
	1		3	4	5	7	8
0	1		2	3	4	5	6
		1	3	4	5	7	8
0		1	2	3	4	5	6



Referência: https://www.hackerearth.com/practice/notes/heaps-and-priority-queues/

É tema comum nos livros da bibliografia.