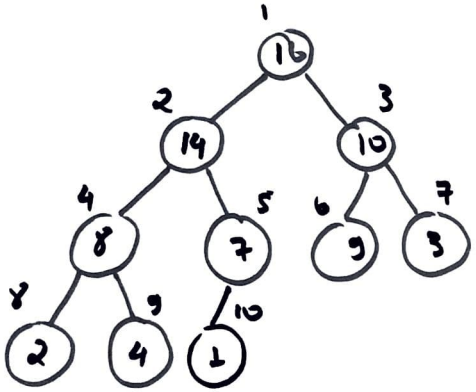
Heap binária

* É uma estrutura de dado que pode ser vista como uma árvore binária quase completa.
* Pode ser implementado em vetor: $A = [1..n]$
* Propriedade: Se $A[1..n]$ é um max-heap, então $A[i] \leq A[pai(i)] \ \forall i = 2, \dots n$.

Exemplo: A:

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



⇐ Altura de árvore é $O(\log n)$.

Algoritmos auxiliares:

pai(i)
1. devolva $\lfloor i/2 \rfloor$

filho_esq.(i)
1. devolva $i*2$

filho_dir(i)
1. devolva $i*2+1$

Algoritmos fundamentais

max_heapfica: recebe um quase max-heap A e devolve A sendo um max-heap.

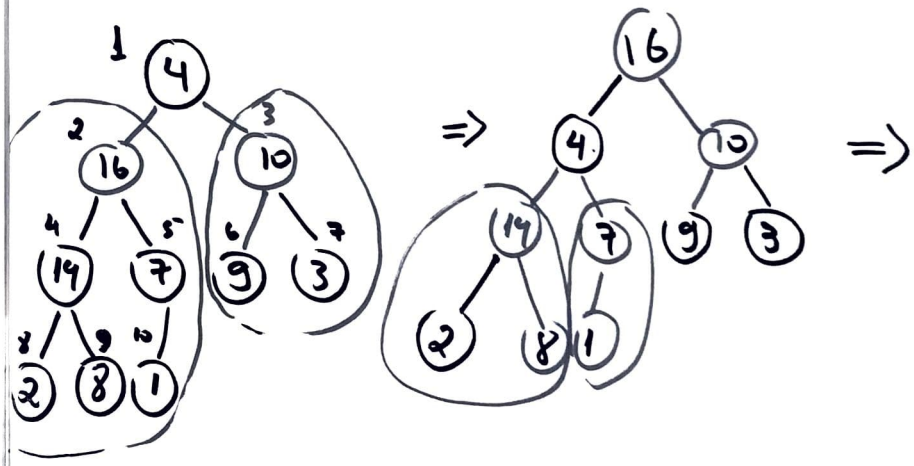
constrói-max-heap: recebe elementos com valores (inteiros) e devolve um max-heap

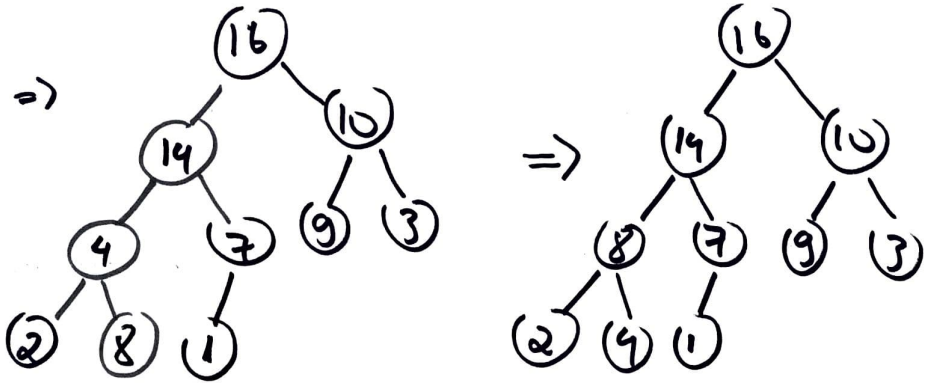
insere-max-heap: insere um novo elemento em um max-heap

remove-máximo-max-heap: remove de um max-heap um elemento de valor máximo.

aumenta-valor-max-heap: aumenta valor de um elemento de um max-heap

Exemplo: Quase max-heap





max_heapfica (A, i) ▷ A[1..n] é quase max-heap a partir de i.

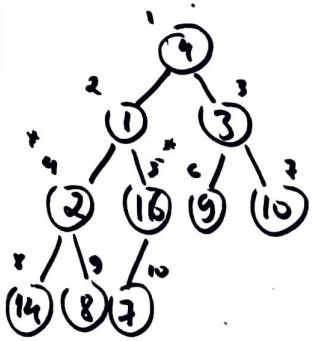
1. e ← filho_esq. (i)
2. d ← filho_dir. (i)
3. maior ← i
4. se e ≤ n e A[e] > A[i]
5. então maior ← e
6. se d ≤ n e A[d] > A[maior]
7. então maior ← d
8. se maior ≠ i
9. então A[i] ↔ A[maior]
10. max_heapfica (A, maior)

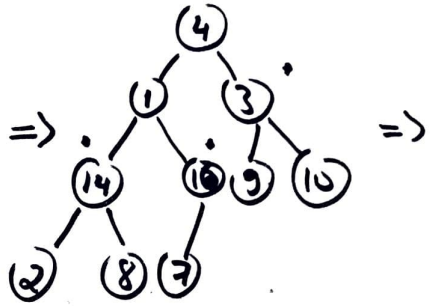
Tempo de execução: $O(\log n)$.

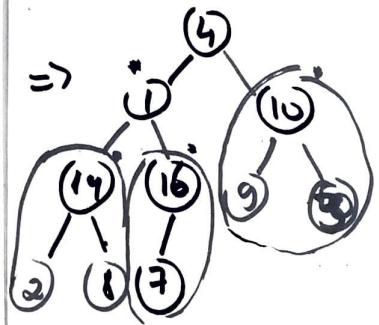
Exemplo: constrói max-heap.

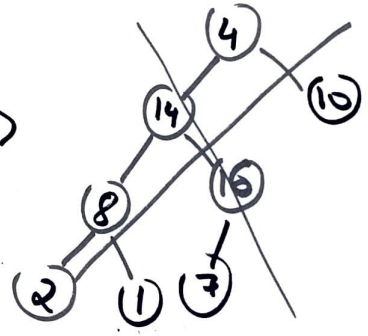
i	1	2	3	4	5	6	7	8	9	10
A:	4	1	3	2	16	9	10	14	8	7

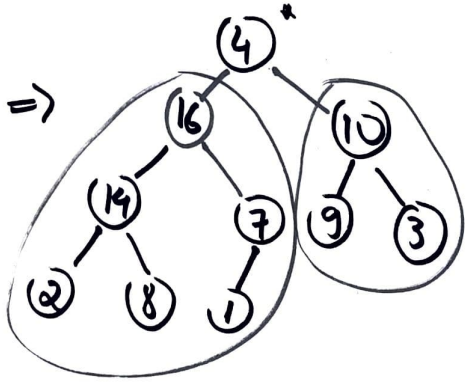
pai (i) => $\lfloor i/2 \rfloor$

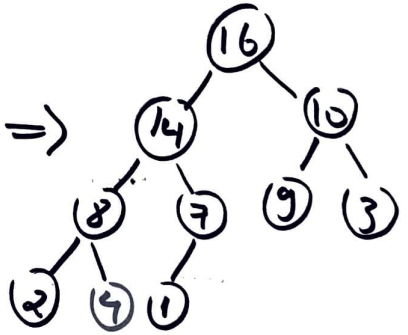












Constrói-max-heap (A) ▷ A[1..n] vetor (inteiro)

1. para i = pai(n) decrementando até 1 faça
2. max-heapfica (A, i)

Tempo de execução: $O(n)$.

T: tempo de execução do algoritmo para um heap com altura h.

$$T \leq 2^{h-1} \times 1 + 2^{h-2} \times 2 + \dots + 2^0 \times h = \sum_{i=0}^{h-1} 2^i (h-i) = \sum_{i=0}^{h-1} 2^i h - \sum_{i=0}^{h-1} 2^i i = h \underbrace{\sum_{i=0}^{h-1} 2^i}_{S'(n)} - \underbrace{\sum_{i=0}^{h-1} 2^i i}_{S''(n)}$$

$S'(n) = 2^0 + 2^1 + \dots + 2^{h-2} + 2^{h-1}$ ↑

$2S'(n) = 2^1 + 2^2 + \dots + 2^{h-1} + 2^h$ −

$\boxed{S'(n) = 2^h - 1}$

$S''(n) = 2^0 \times 0 + 2^1 \times 1 + \dots + 2^{h-2}(h-2) + 2^{h-1}(h-1)$ ↑ −

$2S''(n) = 2^1 \times 0 + 2^2 \times 1 + \dots + 2^{h-1}(h-2) + 2^h(h-1)$

$S''(n) = 2^1(-1) + 2^2(-1) + \dots + 2^{h-1}(-1) + 2^h(h-1)$

$$S''(n) = 2^h(h-1) - \sum_{i=1}^{h-1} 2^i = 2^h(h-1) - (2^h - 2).$$

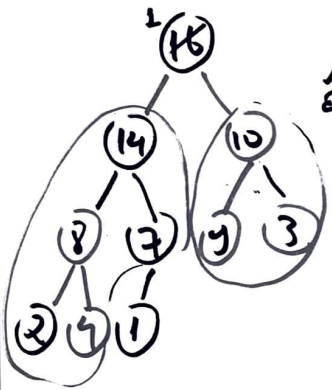
$T \leq h(2^h - 1) - 2^h(h-1) + (2^h - 2)$. Como $h \cong \log_2 n$.

$T \lesssim \log n (2^{\log n} - 1) - 2^{\log n}(\log n - 1) + (2^{\log n} - 2) =$ $= O(n)$.

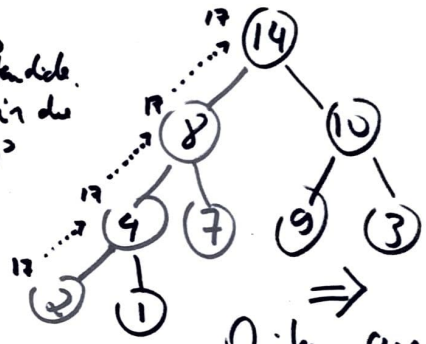
$= n\log n - \log n - n\log n + n + n - 2 = \boxed{2n - 2 - \log n}$

Fila de Prioridade (F.P)

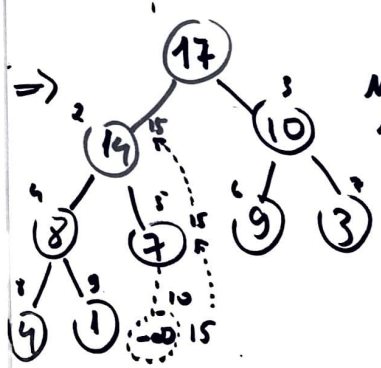
+ Itens na fila com maior valor saíam atendidos primeiro.
+ Podemos usar max-heap para implementar F.P



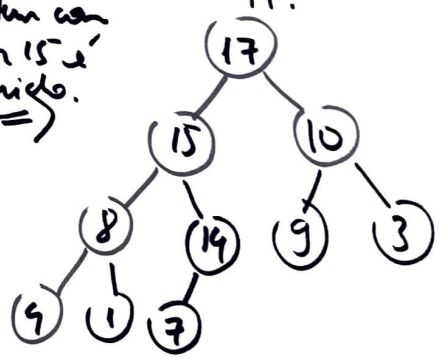
Item cujo valor é 16 é atendido. Esse item deve sair do heap ⇒



⇒ O item cujo valor é 2 tem valor aumentado p/ 17.



Item com valor 15 é inserido. ⇒



Algoritmos fundamentais para F.P.

remove_máximo_max_heap(A) ▷ A[1..n] max-heap

1. max ← A[1]
2. A[1] ← A[n]
3. n ← n - 1
4. max_heapfica(A, 1).
5. devolva max

Tempo de execução: $O(\log n)$

---

aumente_valor_max_heap(A, i, novo_valor) ▷ A[1..n] e novo_valor ≥ A[i]

1. A[i] ← novo_valor ▷ (modifique o valor do item i)
2. enquanto i > 1 e A[pai(i)] < A[i]
3. A[i] ↔ A[pai(i)]
4. i ← pai(i)

aumentamos o valor do item que está em A[i].

Tempo de execução: $O(\log n)$.

---

insere_max_heap(A, item) ▷ A[1..n] e item tem valor em item.valor

1. n ← n + 1
2. A[n] ← item ▷ (modifique temporariamente o valor de item para $-\infty$)
3. aumente_valor_max_heap(A, n, item.valor(original))

Tempo de execução: $O(\log n)$.