HEAPSORT (Cormen, cap. 6)

Um "monte" ("heap") de "n" elementos armazenado num vetor "V" é tal que:

* Os elementos estão nas posições 1 a "n" do vetor.

* Na árvore binária induzida pelo monte, o filho esquerdo do elemento da posição "i" do vetor está na posição 2*i, e o filho direito na posição 2*i + 1.

* A profundidade do elemento "i", isto é, o comprimento (em arestas) do caminho (na árvore induzida pelo monte) da raiz até esse elemento, é d(i) = ⌊lg i⌋.

* A altura do elemento "i", isto é, o maior comprimento de um caminho desse elemento a um descendente dele, é h(i) = D(i) - d(i), onde D(i) é a maior profundidade de um descendente do elemento "i". Observe que ou D(i) = d(n) ou D(i) = d(n) - 1, já que todas as folhas da árvore induzida pelo monte estão nos dois níveis mais baixos. Em particular, como D(i) <= d(n), então h(i) <= d(n) - d(i).

  Abaixo, nós às vezes escreveremos h(i,n), D(i,n) e d(i,n) para ressaltar o fato de que a altura do elemento "i" (assim como D(i)) também depende do número de elementos do monte.

Num monte de máximo, a chave de um elemento é sempre maior ou igual àquela de um filho seu. Num monte de mínimo, o oposto ocorre.

DESCIDA NUM MONTE

O seguinte é um algoritmo de descida num monte de máximo:

```
  ============================================================================
  Algoritmo: descer_no_monte
  Entrada: vetor V[1..] e naturais "i" e "n", com 1 <= i <= n.
  Saída: nenhuma.
  ----------------------------------------------------------------------------
  01. fe := 2*i
      // Variante: h(i)  (prox >= 2*i -> d(prox) >= 1 + d(i) -> h(prox) < h(i)).
  02. ENQUANTO fe <= n
  03. | fd := fe + 1
  04. | SE fd <= n E V[fe] < V[fd]
  05. | | prox := fd
  06. | SENÃO
  07. | | prox := fe
  08. | SE V[prox] <= V[i]
  09. | | RETORNE.
  10. | aux := V[prox] , V[prox] := V[i] , V[i] := aux
  11. | i := prox , fe := 2*i
  ============================================================================
```

Pelo variante acima, é fácil observar que o laço do algoritmo executa no máximo h(i) iterações completas -- ou, em outras palavras, no máximo h(i) TROCAS --, e portanto que o tempo de execução de uma chamada descer_no_monte(V,i,n) é limitado superiormente por

$$t(i,n) = a*h(i,n) + b,$$

onde a,b > 0 são naturais grandes o suficiente.

Além disso, como

```
  h(i,n) <= d(n) - d(i)
          = ⌊lg n⌋ - ⌊lg i⌋
         <= ⌊lg n⌋
         <= lg n,
```

então t(i,n) <= a*(lg n) + b, e portanto t(i,n) = O(lg n).

CONSTRUÇÃO DE UM MONTE

O seguinte algoritmo constrói um monte de máximo:

```
  ================================================================================
  Algoritmo: construir_monte
  Entrada: vetor V[1..] e natural "n".
  Saída: nenhuma.
  --------------------------------------------------------------------------------
  1. PARA i DE n A 1
  2. | descer_no_monte(V,i,n)
  ================================================================================
```

1. EXERCÍCIO: você percebe que as iterações iniciais do laço acima certamente
   --------- não modificam o vetor, já que V[i] não tem filhos? Disso segue que
   o valor inicial de "i" pode, sem prejuízo para a correção do algoritmo, ser
   diminuído para o maior índice j <= n tal que V[j] não é uma folha. Obtenha
   esse índice.

Como o algoritmo acima apenas realiza "n" iterações de custo O(lg n) cada, é evidente que ele executa em tempo O(n*lg n). Como nós argumentamos mais abaixo, porém, também é verdade que ele executa em tempo O(n), e portanto em tempo θ(n).

EXTRAÇÃO DE UM MONTE

O seguinte algoritmo remove o maior elemento de um monte de máximo em tempo O(lg n):

```
  ================================================================================
  Algoritmo: remover_máximo
  Entrada: vetor V[1..] e natural n >= 1 tais que V[1..n] é um monte.
  Saída: um elemento de V.
  --------------------------------------------------------------------------------
  1. m := V[1]
  2. SE n > 1
  3. | V[1] := V[n]
  4. | --n
  5. | descer_no_monte(V,1,n)
  6. RETORNE m
  ================================================================================
```

ORDENAÇÃO POR MONTE (HEAPSORT)

O seguinte algoritmo ordena um vetor sucessivamente selecionando o maior elemento do trecho ainda por ser ordenado e então colocando-o no final desse trecho. Diferentemente, porém, do algoritmo de ordenação por seleção, que descobre tal elemento por meio de uma busca linear, o algoritmo abaixo mantém o trecho não ordenado num monte, de forma que a seleção do maior elemento pode ser feita em tempo O(lg n), com o quê o algoritmo completo executa em tempo O(n*lg n) -- em contraste com o tempo $\theta(n^2)$ da ordenação por seleção.

```
Algoritmo: ordenação_por_monte   // "heapsort"
Entrada: vetor V[1..n].
Saída: nenhuma.
-------------------------------------------------------------------------------
1. construir_monte(V,n)
2. t := n
3. ENQUANTO 2 <= t
4. | V[t] := remover_máximo(V,t)
5. | --t
```

TEMPO LINEAR DE CONSTRUÇÃO DE UM MONTE

Recapitulando o algoritmo construir_monte, é evidente que o tempo de execução do algoritmo é O(x), sendo "x" o máximo entre "n" e o número total de trocas realizadas durante as várias chamadas a descer_no_monte.

Se você calcular manualmente o número máximo de trocas realizadas na construção de montes completos (isto é, de 1, 3, 7, 15, etc, elementos), perceberá que esse número nunca é maior que o número de elementos do monte. De fato, isso pode ser provado por indução matemática:

2. TEOREMA: para todo natural k >= 1, $\Sigma_{i=1}^{t} h(i,t) <= t$, sendo $t = 2^k - 1$.

PROVA:

Por indução em k:

* TESE: para todo k >= 1, $\Sigma_{i=1}^{t} h(i,t) <= t$, sendo $t = 2^k - 1$.

* BASE (k = 1): Temos $t = 2^1 - 1 = 1$ e:

$$\Sigma_{i=1}^{1} h(i,t) = h(1,1) = 0 <= 1, \text{ CQD.}$$

* H.I.: para todo $1 <= j < k$, $\Sigma_{i=1}^{t} h(i,t) <= t$, sendo $t = 2^j - 1$.

* PASSO (k > 1): Temos $t = 2^k - 1$ e:

$$\Sigma_{i=1}^{t} h(i,t) = \Sigma_{i=1}^{2^{(k-1)} - 1} h(i,t) + \Sigma_{i=2^{(k-1)}}^{t} h(i,t)$$

```
         = Σ_{i=1}^{2^(k-1) - 1} ( ⌊lg t⌋ - ⌊lg i⌋ )    ---> def. "h", dado que
           +                                                 o monte é completo.
           Σ_{i=2^(k-1)}^t ( ⌊lg t⌋ - ⌊lg i⌋ )

         = Σ_{i=1}^{2^(k-1) - 1} ( ⌊lg (2^k -1)⌋ - ⌊lg i⌋ )
           +
           Σ_{i=2^(k-1)}^t ( (k-1) - (k-1) )   ---> pois 2^(k-1) <= i <= t < 2^k

         = Σ_{i=1}^{2^(k-1) - 1} ( (k-1) - ⌊lg i⌋ ) --> 2^(k-1) <= 2^k - 1 < 2^k
           +
           0

         = Σ_{i=1}^{2^(k-1) - 1} ( 1 + (k-2) - ⌊lg i⌋ )

         = Σ_{i=1}^{2^(k-1) - 1} ( 1 + ⌊lg (2^(k-1) -1)⌋ - ⌊lg i⌋ )

         = Σ_{i=1}^{2^(k-1) - 1} 1
           +
           Σ_{i=1}^{2^(k-1) - 1} ( ⌊lg (2^(k-1) -1)⌋ - ⌊lg i⌋ )

        <= 2^(k-1) - 1
           +
           2^(k-1) -1    ---> Pela H.I., com j = k-1.

         = 2*2^(k-1) -1 -1

         < 2*k - 1, CQD.
```

==========

O teorema acima mostra que o número de trocas realizadas durante a construção de montes completos é no máximo o número de elementos do monte, e portanto que essa construção leva tempo linear.

Fato semelhante pode ser mostrado a respeito de montes não completos, isto é, cujo número de elementos não é da forma 2^k - 1:

3. TEOREMA: Se "n", "t" e k > 1 são naturais tais que

$$2^{(k-1)} - 1 < n < 2^k - 1 = t,$$

então $\Sigma_{i=1}^{n} h(i,n) <= 2n$.

==========

PROVA:

Observe que, para todo 1 <= i <= n, temos

```
  h(i,n)  = D(i,n) - d(i,n)
         <= D(i,t) - d(i,n)   ---> pois n < t
          = D(i,t) - d(i,t)
          = h(i,t),
```

isto é, a altura do i-ésimo elemento de um monte de "n" elementos é sempre menor ou igual à altura do i-ésimo elemento de um monte de "t" elementos, sendo t > n.

Assim sendo, temos então:

```
  Σ_{i=1}^n h(i,n) <= Σ_{i=1}^n h(i,t)   ---> pelo argumento acima
                    < Σ_{i=1}^t h(i,t)   ---> pois n < t
                   <= t                  ---> teorema 2
                    < 2*n, CQD.          ---> pois 2^(k-1) <= n, e portanto
                                                 t  < 2^k     <= 2*n.
```

==========

Nós concluímos então que o algoritmo construir_monte executa em tempo linear sobre o tamanho do monte construído:

4. TEOREMA: toda chamada construir_monte(V,n) executa em tempo θ(n).

==========
PROVA:

O algoritmo sempre executa "n" iterações de um laço, e portanto sempre executa em tempo Ω(n). Além disso, o tempo de execução do algoritmo é O(x), sendo "x" o máximo entre "n" e o número total de trocas realizadas durante as chamadas a descer_no_monte, número esse que é no máximo Σ_{i=1}^n h(i,n), e portanto, pelos teoremas 2 e 3, no máximo 2n. Logo, construir_monte executa em tempo O(n), CQD.
==========