

1.3.6 Heaps ocos

(Versão: 7867.)

Introdução

Objetivo: operações com a mesma complexidade amortizada que heaps de Fibonacci. Para um heap h , chave k e elemento e temos as operações:

- $\text{make-heap}(): O(1)$
- $\text{find-min}(h)/\text{getmin}(h): O(1)$
- $\text{meld}(h_1, h_2): O(1)$
- $\text{insert}(e, k, h): O(1)$
- $\text{decrease-key}(e, k, h): O(1)$
- $\text{delete}(e, h): O(\log n)$
- $\text{delete-min}(h): O(\log n)$

Ideia principal: a operação delete esvazia nós, produzindo nós ocos (ingl. hollow nodes), a operação decrease-key é um delete, seguido por um insert. Teremos duas medidas:

n Número de elementos no heap

N Número de nós no heap = # de elementos + # de nós ocos = # operações insert + # operações decrease-key

Variantes de heaps ocos:

- Heaps ansiosos (ingl. “eager heaps”) com múltiplas raízes.
- Heaps ansiosos com uma única raíz.
- Heaps preguiçosos.

```
1 def Node =
2   item // elemento
3   key  // chave
4   fc   // ponteiro para primeiro filho
5   ns   // ponteiro para próximo irmão
6   rank // posto do nó
```

```

7
8 def Item =
9     no      // nó correspondente
10    // mais dados satelites

```

Operação básica: link Um *link* gera um vencedor e um perdedor, que se torna filho do vencedor, e aumenta o posto do vencedor.

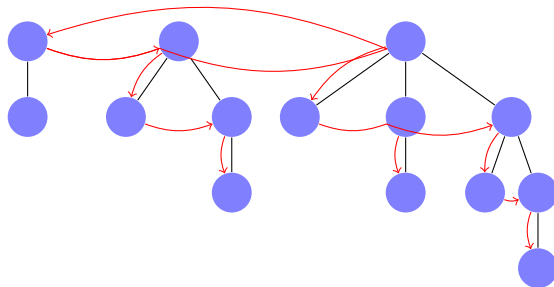
```

1  (ranked)link( $t_1, t_2$ ) :=
2      if  $t_1.key \leq t_2.key$ 
3          return makechild( $t_1, t_2$ )
4      else
5          return makechild( $t_2, t_1$ )
6
7  makechild( $w, l$ ) :=
8      l.ns    := w.fc
9      w.fc    := l
10     w.rank  := w.rank+1
11     return w

```

Representação básica

- Lista simples circular de árvores com ordenação do heap, representada por um ponteiro à árvore cuja raiz contém a menor chave (chamada a *raiz mínima*).
- Cada *nó cheia* armazena um item. Podem existir *nós ocos* sem item.
- Nós ocos nunca mais ficam cheias, eles podem somente ser destruídos.
- Filhos ficam armazenados em listas simples, em ordem não-crescente de postos.



```

1  make-heap() := return null
2
3  make-heap(e,k) := return Node(e,k,null,self,0)
4
5  getmin(h) := h
6
7  findmin(h) := return h is not null? h.item : null
8
9  meld( $h_1, h_2$ ) :=
10     if  $h_1$  is null return  $h_2$ 
11     if  $h_2$  is null return  $h_1$ 
12     swap( $h_1$ .ns,  $h_2$ .ns) // cria uma lista circular simples
13     if  $h_1$ .key  $\leq$   $h_2$ .key return  $h_1$  else return  $h_2$ 
14
15  insert(e,k,h) := meld(make-heap(e,k),h)
16
17  decrease-key(e,k,h) :=
18     u = e.node
19     v = make-heap(e,k)
20     v.rank = max{0,u.rank-2}
21     // desloca os filhos de postos 0,...,rank-2 para v
22     if u.rank  $\geq$  2
23         v.fc := u.fc.ns.ns
24         u.fc.ns.ns := null
25     return meld(v,h)
26
27  delete(e,h) :=
28     e.node.item := null
29     if e.node = h
30         delete-min(h)
31
32  delete-min(h) :=
33     if h is null: return
34     h.node.item := null
35
36  aloca um array  $R_0, R_1, \dots, R_M$ 
37  // repetidamente remove raízes ociosas e une os heaps
38  r:=h
39  repeat
40     rn := r.ns
41     link-heap(r,R)

```

```

42         r:=rn
43     until r==h
44
45     // reconstrói o heap
46     h:=null
47     for i=0,...,M
48         if  $R_i$  is not null
49              $R_i.ns := R_i$ 
50             h := meld(h,  $R_i$ )
51     return h
52
53 link-heap(h,R) :=
54     if h is hollow
55         r:=h.fc
56         while r is not null
57             rn := r.ns
58             link-heap(r,R)
59             r := rn
60         destroy node h
61     else
62         i := h.rank
63         while  $R_i$  is not null
64             h := link(h,  $R_i$ )
65              $R_i := null$ 
66             i := i + 1
67     end
68      $R_i := h$ 

```

Invariantes

1. Ordenação do heap.
2. Invariante do posto: cada nó de posto r possui r filhos com postos $0, \dots, r-1$, exceto no caso $r \geq 2$ e o nó foi esvaziada por uma operação decrease-key. Neste caso o nó possui dois filhos de postos $r-1$ e $r-2$.

Corretude

Teorema 1.1

Heaps com nós ocios implementam corretamente todas operações e mantêm as invariantes.

Prova. Por indução sobre o número de operações. ■

Lembrança: os números de Fibonacci são definidos por $F_0 = 0, F_1 = 1, F_{i+2} = F_i + F_{i+1}$, para $i \geq 0$ e temos $F_{i+2} \geq \Phi^i$, com a razão áurea $\Phi = (1 + \sqrt{5})/2$.

Teorema 1.2

Um nó de posto r possui pelo menos $F_{r+3} - 1$ descendentes (cheios ou ocos), incluindo o próprio nó, na árvore.

Prova. Por indução sobre r . Para $r = 0$, temos $F_3 - 1 = 1$, e para $r = 1$ temos $F_4 - 1 = 2$ e a afirmação está correta, porque para $r < 2$ um nó não perde filhos caso for esvaziado. Para $r \geq 2$ pela invariante do posto temos pelo menos dois filhos com postos $r - 1$ e r_2 . Pela hipótese da indução eles tem pelo menos $F_{r+1} - 1$ e $F_{r+2} - 1$ descendentes e logo r possui pelo menos $F_{r+1} - 1 + F_{r+2} - 1 + 1 = F_{r+3} - 1$ descendentes. ■

Corolário 1.1

Depois uma operação delete-min o número de árvores é no máximo $\lceil \log_\Phi N \rceil = O(\log N)$ porque temos no máximo uma árvore por posto. Logo podemos escolher $M = \lceil \log_\Phi N \rceil$ na operação delete-min.

Teorema 1.3

O tempo amortizado por operação num heap oco é $O(1)$, exceto para as operações delete e delete-min, que tem complexidade $O(\log N)$ para um heap com N nós.

Prova. Todas operações exceto a deleção do elemento mínimo possuem tempo $O(1)$ no caso pessimista. O custo de uma deleção é $O(H + T)$ com H o número de nós ocos destruídos, e T o número de árvores antes das operações link. Depois das operações link temos no máximo $\log_\Phi N$ árvores, logo faremos pelo menos $T - \log_\Phi N$ operações link e no máximo $\log_\Phi N$ operações meld. Logo o custo total é $O(1)$ por destruição de um nó oco, e por link, mas $O(\log N)$. Para contabilizar a destruição de um nó, aumentamos o custo de cada criação (insert, decrease-key) por 1.

Para contabilizar as operações link: define um potencial igual ao número de nós cheias, que não são filho de outro nó cheia (i.e. raízes e filhos de nós ocos). Para todas operações diferente de delete-min e delete, o aumento do potencial é constante (no máximo 1 para insert, 3 para decrease-key, 0 para as demais). Para o delete que remove o elemento mínimo e delete-min, o custo amortizado de cada link é 0, porque um link combina duas raízes cheias, reduzindo o potencial por 1. Além disso, ao remover um elemento, o potencial aumenta por no máximo $\log_\Phi N$, um por cada filho do novo nó oco. Logo o custo amortizado de delete e delete-min é $O(\log N)$. ■

Re-otimizando o heap A análise acima é em função de N . Caso $\log N = O(\log n)$ temos um heap assintoticamente ótimo. Caso executamos muitas operações *decrease-key*, temos que reconstruir o heap periodicamente, para garantir $N = O(n)$. O método mais simples é: escolhe uma constante $c > 1$ e para $N > cn$ reconstrói o heap completamente, destruindo os nós ociosos, criando heaps de um único nó de todos nós cheios, e aplicando operações *meld* para unir todos heaps. O custo é $O(N)$ para percorrer todo nó uma vez e pode ser atribuído na análise amortizada para as operações *insert* e *delete-min*.