Capítulo 10

Heaps

Neste capítulo, discutiremos duas implementações da estrutura de dados de prioridade Fila extremamente útil. Ambas as estruturas são um tipo especial de árvore binária chamada *heap*, o que significa "uma pilha desorganizada." Isso contrasta com as árvores de busca binária que podem ser consideradas como uma pilha altamente organizada.

A primeira implementação de *heap* usa um array para simular uma árvore binária completa. Esta implementação muito rápida é a base de um dos mais rápidos algoritmos de ordenação conhecidos, o chamado *heapsort* (ver Seção 11.1.3). A segunda implementação é baseada em árvores binárias mais flexíveis. Ela suporta uma operação meld(h) que permite que a fila de prioridades absorva os elementos de uma segunda fila de prioridade h.

10.1 BinaryHeap: Uma árvore binária implícita

Nossa primeira implementação de uma Fila (de prioridade) é baseada em uma técnica que tem mais de quatrocentos anos de idade. O *método Eytzinger* nos permite representar uma árvore binária completa como um array, colocando os nós da árvore em ordem de largura (ver Seção 6.1.2). Desta forma, a raiz é armazenada na posição 0, o filho esquerdo da raiz é armazenado na posição 1, o filho direito da raiz na posição 2, o filho esquerdo do filho esquerdo da raiz é armazenado na posição 3 e assim por diante. Ver Figura 10.1.

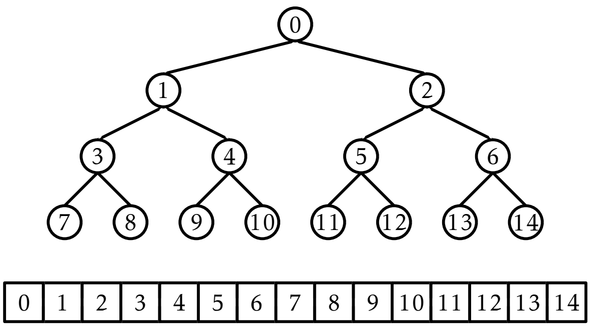

Figura 10.1: O método de Eytzinger representa uma árvore binária completa como um array.

Se aplicarmos o método de *Eytzinger* a uma árvore suficientemente grande, alguns padrões surgem. O filho esquerdo do nó no índice i está no índice $\mathrm{left}(i) = 2i + 1$ e o filho direito do nó no índice i está no índice $\mathrm{right}(i) = 2i + 2$. O pai do nó no índice i está no índice $\mathrm{parent}(i) = (i - 1)/2$.

left(i)
    return $2 \cdot i + 1$

right(i)
    return $2 \cdot (i + 1)$

parent(i)
    return $(i - 1) \operatorname{div} 2$

Uma BinaryHeap usa essa técnica para representar implicitamente uma árvore binária em que os elementos são *ordenados pelo heap*: o valor armazenado em qualquer índice i não é menor que o valor armazenado no índice parent(i), com a exceção do valor da raiz, $i = 0$. Segue-se que o menor valor na Fila de prioridade é, portanto, armazenado na posição 0 (a raiz).

Na BinaryHeap, os n elementos são armazenados em um array a:

```
initialize()
    a ← new_array(1)
    n ← 0
```

Implementar a operação add(x) é bastante simples. Como acontece com todas as estruturas baseadas em array, primeiro verificamos se a está cheio (verificando se length(a) = n) e, em caso afirmativo, aumentamos a. Em seguida, colocamos x no local $a[n]$ e incrementamos n. Neste ponto, tudo o que resta é garantir que mantemos a propriedade do *heap*. Fazemos isso trocando repetidamente x por seu pai até que x não seja menor que seu pai. Ver Figura 10.2.

```
add(x)
    if length(a) < n + 1 then
        resize()
    a[n] ← x
    n ← n + 1
    bubble_up(n − 1)
    return true

bubble_up(i)
    p ← parent(i)
    while i > 0 and a[i] < a[p] do
        a[i], a[p] ← a[p], a[i]
        i ← p
        p ← parent(i)
```

Implementar a operação remove(), que remove o menor valor do *heap*, é um pouco mais complicado. Nós sabemos onde o menor valor está (na raiz), mas precisamos substituí-lo depois de removê-lo e garantir que mantemos a propriedade de *heap*.

A maneira mais fácil de fazer isso é substituir a raiz pelo valor $a[n-1]$, excluir esse valor e decrementar n. Infelizmente, o novo elemento raiz agora provavelmente não é o menor elemento, por isso ele precisa ser mo-

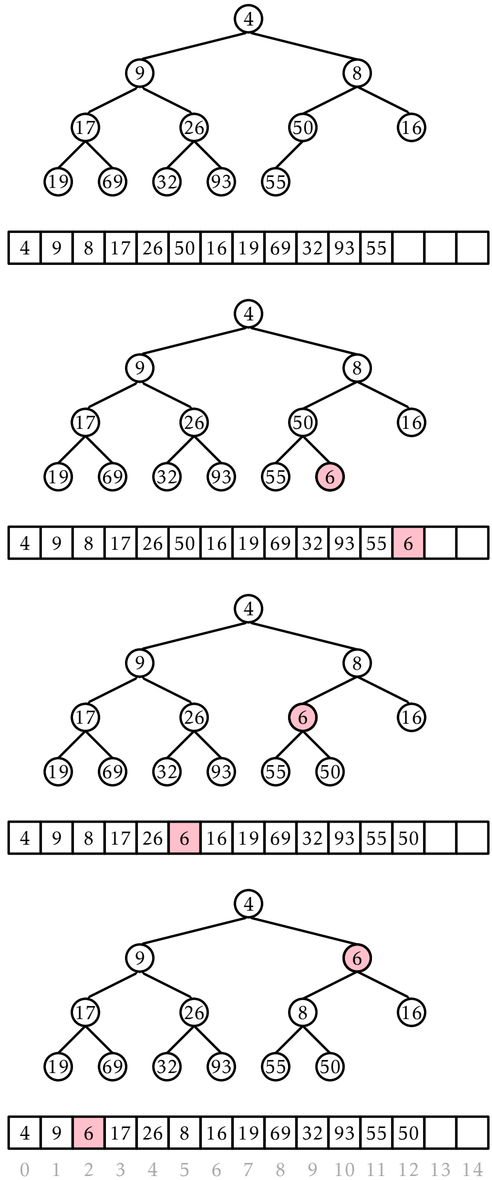

Figura 10.2: Adicionando o valor 6 à BinaryHeap.

vido para baixo. Fazemos isso comparando repetidamente esse elemento com seus dois filhos. Se é o menor dos três, então estamos prontos. Caso contrário, nós trocamos este elemento pelo menor de seus dois filhos e continuamos.

```
remove()
    x ← a[0]
    a[0] ← a[n − 1]
    n ← n − 1
    trickle_down(0)
    if 3 · n < length(a) then
        resize()
    return x

trickle_down(i)
    while i ≥ 0 do
        j ← −1
        r ← right(i)
        if r < n and a[r] < a[i] then
            ℓ ← left(i)
            if a[ℓ] < a[r] then
                j ← ℓ
            else
                j ← r
        else
            ℓ ← left(i)
            if ℓ < n and a[ℓ] < a[i] then
                j ← ℓ
        if j ≥ 0 then
            a[j], a[i] ←    a[i], a[j]
        i ← j
```

Como com outras estruturas baseadas em array, iremos ignorar o tempo gasto em chamadas para resize(), uma vez que elas podem ser contabilizadas usando o argumento de amortização do Lema 2.1. Os tempos de

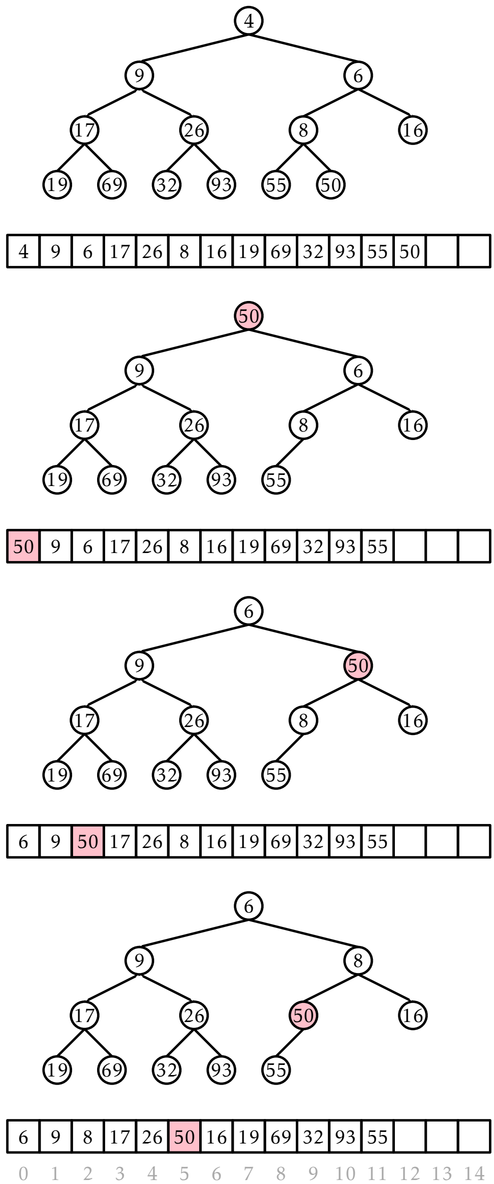

Figura 10.3: Removendo o valor mínimo, 4, de uma BinaryHeap.

execução de add(x) e remove() dependem da altura da árvore binária (implícita). Felizmente, esta é uma árvore binária *completa*; cada nível, exceto o último, tem o número máximo possível de nós. Portanto, se a altura dessa árvore for h, ela terá pelo menos 2^h nós.

$$n \geq 2^h \enspace .$$

Tomando logaritmos em ambos os lados desta equação dá

$$h \leq \log n \enspace .$$

Portanto, ambas as operações add(x) e remove() são executadas em um tempo $O(\log n)$.

10.1.1 Resumo

O seguinte teorema resume o desempenho de uma BinaryHeap:

Teorema 10.1. *Uma BinaryHeap implementa a interface Fila (de prioridade). Ignorando o custo das chamadas para* resize(), *a BinaryHeap suporta as operações* add(x) *e* remove() *em um tempo por operação de* $O(\log n)$.

Além disso, começando com uma BinaryHeap vazia, qualquer sequência de m operações add(x) *e* remove() *resulta em um total de tempo* $O(m)$ *gasto durante todas as chamadas para* resize().

10.2 MeldableHeap: Uma Heap fusionável aleatória

Nesta seção, descrevemos o MeldableHeap, uma implementação da Fila de prioridade, na qual a estrutura subjacente também é uma árvore binária ordenada por heap. No entanto, ao contrário de uma BinaryHeap no qual a árvore binária subjacente é completamente definida pelo número de elementos, não há restrições quanto à forma da árvore binária subjacente na MeldableHeap; qualquer coisa serve.

As operações add(x) e remove() em uma MeldableHeap são implementadas em termos da operação merge(h_1,h_2). Essa operação usa dois nós de heap h_1 e h_2 e mescla-os, retornando um nó de heap que é a raiz de uma heap que contém todos os elementos na subárvore com raiz em h_1 e todos os elementos na subárvore com raiz em h_2.

O bom de uma operação merge(h_1,h_2) é que ela pode ser definida recursivamente. Veja Figura 10.4. Se h_1 ou h_2 for *nil*, então estamos mesclando com um conjunto vazio, então retornamos h_2 ou h_1, respectivamente. Caso contrário, assuma $h_1.x \leq h_2.x$ pois, se $h_1.x > h_2.x$, poderemos inverter os papéis de h_1 e h_2. Então, sabemos que a raiz da heap mesclada conterá $h_1.x$ e podemos mesclar recursivamente h_2 com $h_1.left$ ou $h_1.right$, como desejamos. É aqui que entra a randomização e lançamos uma moeda para decidir se devemos mesclar h_2 com $h_1.left$ ou $h_1.right$:

```
merge(h1,h2)
    if h1 = nil then return h2
    if h2 = nil then return h1
    if h2.x < h1.x then (h1,h2) ← (h2,h1)
    if random_bit() then
        h1.left ← merge(h1.left,h2)
        h1.left.parent ← h1
    else
        h1.right ← merge(h1.right,h2)
        h1.right.parent ← h1
    return h1
```

Na próxima seção, mostramos que merge(h_1,h_2) é executado em um tempo esperado de $O(\log n)$, onde n é o número total de elementos em h_1 e h_2.

Com o acesso a uma operação merge(h_1,h_2), a operação add(x) é fácil. Criamos um novo nó u contendo x e depois mesclamos u com a raiz do heap:

```
add(x)
    u ← new_node(x)
    r ← merge(u,r)
    r.parent ← nil
    n ← n + 1
    return true
```

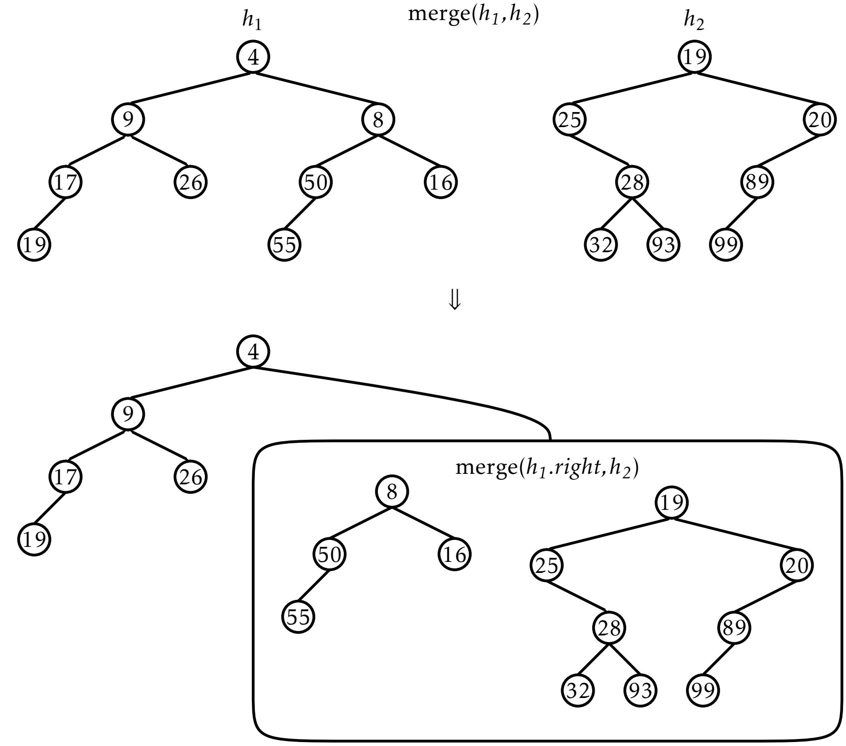

Figura 10.4: A mesclagem de h_1 e h_2 é feita mesclando h_2 com um dos h_1.*left* ou h_1.*right*.

Isto leva um tempo esperado de $O(\log(n+1)) = O(\log n)$.

A operação remove() é igualmente fácil. O nó que queremos remover é a raiz, então apenas mesclamos seus dois filhos e fazemos a raiz ser o resultado:

```
remove()
    x ← r.x
    r ← merge(r.left, r.right)
    if r ≠ nil then r.parent ← nil
    n ← n − 1
    return x
```

Novamente, isto leva um tempo esperado de $O(\log n)$.

Além disso, uma MeldableHeap pode implementar muitas outras operações em um tempo esperado de $O(\log n)$, incluindo:

- remove(u): remove o nó u (e sua chave $u.x$) do heap.
- absorb(h): adicione todos os elementos da MeldableHeap h a este heap, esvaziando h no processo.

Cada uma dessas operações pode ser implementada usando um número constante de operações de merge(h_1, h_2), cada uma levando um tempo esperado de $O(\log n)$.

10.2.1 Análise de merge(h_1, h_2)

A análise de merge(h_1, h_2) é baseada na análise de um passeio aleatório em uma árvore binária. Um *passeio aleatório* em uma árvore binária começa na raiz da árvore. Em cada passo da caminhada aleatória, uma moeda é lançada e, dependendo do resultado desse sorteio, a caminhada prossegue para a esquerda ou para o filho direito do nó atual. A caminhada termina quando cai da árvore (o nó atual se torna *nil*).

O seguinte lema é um tanto notável porque não depende de forma alguma da forma da árvore binária:

Lema 10.1. *O comprimento esperado de um passeio aleatório em uma árvore binária com n nós é no máximo* $\log(n+1)$.

Demonstração. A prova é por indução em n. No caso base, $n = 0$ e a caminhada tem comprimento $0 = \log(n+1)$. Suponha agora que o resultado seja verdadeiro para todos os inteiros não negativos $n' < n$.

Faça n_1 indicar o tamanho da subárvore esquerda da raiz, de modo que $n_2 = n - n_1 - 1$ seja o tamanho da subárvore direita da raiz.

Começando na raiz, a caminhada dá um passo e depois continua em uma subárvore de tamanho n_1 ou n_2. Pela nossa hipótese indutiva, a duração esperada da caminhada é então

$$\mathrm{E}[W] = 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \ ,$$

já que cada um de n_1 e n_2 é menor que n. Como log é uma função côncava, $\mathrm{E}[W]$ é maximizado quando $n_1 = n_2 = (n-1)/2$. Portanto, o número esperado de passos feitos pelo passeio aleatório é

$$\begin{aligned}
\mathrm{E}[W] &= 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \\
&\le 1 + \log((n-1)/2+1) \\
&= 1 + \log((n+1)/2) \\
&= \log(n+1) \ .
\end{aligned}$$

□

Fazemos uma rápida digressão para observar que, para os leitores que conhecem um pouco da teoria da informação, a prova de Lema 10.1 pode ser expressa em termos de entropia.

Prova Teórica de Informação de Lema 10.1. Faça d_i indicar a profundidade do i-ésimo nó externo e lembre-se de que uma árvore binária com n nós possui $n+1$ nós externos. A probabilidade de o passeio aleatório atingir o i-ésimo nó externo é exatamente $p_i = 1/2^{d_i}$, então o comprimento esperado do passeio aleatório é dado por

$$H = \sum_{i=0}^{n} p_i d_i = \sum_{i=0}^{n} p_i \log\left(2^{d_i}\right) = \sum_{i=0}^{n} p_i \log(1/p_i)$$

O lado direito desta equação é facilmente reconhecível como a entropia de uma distribuição de probabilidade sobre $n+1$ elementos. Um fato básico sobre a entropia de uma distribuição sobre $n+1$ elementos é que ela não excede $\log(n+1)$, o que comprova o lema. □

Com este resultado em caminhadas aleatórias, agora podemos provar facilmente que o tempo de execução da operação merge(h_1,h_2) é $O(\log n)$.

Lema 10.2. *Se h_1 e h_2 forem as raízes de dois heaps contendo n_1 e n_2 nós, respectivamente, então o tempo de execução esperado de* merge(h_1,h_2) *é no máximo $O(\log n)$, onde $n = n_1 + n_2$,*

Demonstração. Cada etapa do algoritmo de mesclagem leva um passo de uma caminhada aleatória, na heap com raiz em h_1 ou na heap com raiz em h_2. O algoritmo termina quando qualquer um desses dois caminhos aleatórios caírem de sua árvore correspondente (quando $h_1 = nil$ ou $h_2 = nil$). Portanto, o número esperado de passos executados pelo algoritmo de mesclagem é no máximo

$$\log(n_1 + 1) + \log(n_2 + 1) \leq 2\log n \ . \qquad \square$$

10.2.2 Resumo

O seguinte teorema resume o desempenho de uma MeldableHeap:

Teorema 10.2. *Uma MeldableHeap implementa a interface Fila (de prioridade). Uma MeldableHeap suporta as operações* add(x) *e* remove() *em um tempo esperado de $O(\log n)$ por operação.*

10.3 Discussão e Exercícios

A representação implícita de uma árvore binária completa como um array, ou lista, parece ter sido proposta pela primeira vez por Eytzinger [27]. Ele usou essa representação em livros contendo árvores genealógicas de pedigree de famílias nobres. A estrutura de dados BinaryHeap descrita aqui foi introduzida pela primeira vez por Williams [76].

A estrutura de dados MeldableHeap aleatória descrita aqui aparece primeiramente proposta por Gambin e Malinowski [34]. Outras implementações de meldable heap existem, incluindo heaps de esquerda [16, 48, Seção 5.3.2], heaps binomiais [73], heaps de Fibonacci [30], heaps emparelhadas [29], e heaps inclinadas [70], embora nenhum desses seja tão simples quanto a estrutura MeldableHeap.

Algumas das estruturas acima também suportam uma operação decrease_key(u,y) em que o valor armazenado no nó u é reduzido para y. (É uma pré-condição que $y \leq u.x$.) Na maioria das estruturas anteriores, esta operação pode ser suportada em um tempo $O(\log n)$ removendo o nó u e adicionando y. No entanto, algumas dessas estruturas podem implementar decrease_key(u,y) com mais eficiência. Em particular, decrease_key(u,y) leva um tempo amortizado de $O(1)$ nas heaps de Fibonacci e um tempo amortizado de $O(\log\log n)$ numa versão especial de heaps de emparelhamento [25]. Essa operação decrease_key(u,y) mais eficiente tem aplicações para acelerar vários algoritmos gráficos, incluindo o algoritmo de caminho mais curto de Dijkstra [30].

Exercício 10.1. Ilustre a adição dos valores 7 e depois 3 ao BinaryHeap mostrado no final de Figura 10.2.

Exercício 10.2. Ilustre a remoção dos próximos dois valores (6 e 8) na BinaryHeap mostrada no final de Figura 10.3.

Exercício 10.3. Implemente o método remove(i), que remove o valor armazenado em $a[i]$ em BinaryHeap. Este método deve ser executado em um tempo $O(\log n)$. Em seguida, explique por que esse método provavelmente não será útil.

Exercício 10.4. Uma árvore d-aria é uma generalização de uma árvore binária na qual cada nó interno possui d filhos. Usando o método de Eytzinger também é possível representar árvores completas de d-aria usando arrays. Trabalhe as equações que, dado um índice i, determinam o índice do pai de i e cada um dos d filhos de i nesta representação.

Exercício 10.5. Usando o que você aprendeu em Exercício 10.4, projete e implemente um *DaryHeap*, a generalização d-aria de um BinaryHeap. Analise os tempos de execução de operações em DaryHeap e teste o desempenho de sua implementação DaryHeap em relação à implementação BinaryHeap fornecida aqui.

Exercício 10.6. Ilustre a adição dos valores 17 e 82 na MeldableHeap h_1 mostrada em Figura 10.4. Use uma moeda para simular um bit aleatório quando necessário.

Exercício 10.7. Ilustre a remoção dos próximos dois valores (4 e 8) no MeldableHeap h_1 mostrado em Figura 10.4. Use uma moeda para simular um bit aleatório quando necessário.

Exercício 10.8. Implemente o método remove(u) que remove o nó u de uma MeldableHeap. Estemétodo deve executar em um tempo esperado de $O(\log n)$.

Exercício 10.9. Mostre como encontrar o segundo menor valor em uma BinaryHeap ou MeldableHeap em um tempo constante

Exercício 10.10. Mostre como encontrar o k-ésimo menor valor em uma BinaryHeap ou MeldableHeap em um tempo $O(k \log k)$. (Dica: usar uma outra heap pode ajudar.)

Exercício 10.11. Suponha que você tenha k listas ordenadas com um tamanho total de n. Utilizando uma heap, mostre como fundi-las em uma única lista ordenada em um tempo $O(n \log k)$. (Dica: Começar com o caso $k = 2$ pode ser instrutivo.)