

9.4 Heaps Binomiais

Nesta seção serão abordados os *heaps binomiais*, que também permitem realizar a operação da fusão de maneira eficiente. Os seguintes conceitos serão introduzidos.

Seja T uma árvore qualquer enraizada e v um nó de T . O número de filhos de v é a *ordem* de v . A ordem da árvore T é, por definição, igual à ordem de sua raiz. Quando conveniente, pode-se considerar uma árvore como ordenada, segundo valores não decrescentes das ordens de seus nós.

Dadas duas árvores enraizadas T e T' , a operação de *subordinação* de T , T' consiste em construir a árvore obtida quando a raiz de T recebe como filho adicional a raiz de T' . Assim sendo, a árvore T' se transforma numa subárvore da raiz de T . A ordem de T aumenta, naturalmente, de uma unidade. A árvore T é a *subordinadora*, enquanto T' é a *subordinada*. Exceto em situações especiais, esta operação será usada somente para os casos em que as ordens de T e T' são idênticas. Nesse caso, o objetivo seria transformar duas árvores de ordens iguais a h em uma única, de ordem $h+1$, e contendo a união dos nós das árvores dadas. A Figura 9.7 ilustra uma operação de subordinação.

Uma árvore binomial B_h é definida recursivamente, em função de um parâmetro h , como se segue. A árvore B_0 é aquela que contém um único nó. Para $h > 0$ obtém-se B_h através da subordinação de duas árvores binomiais B_{h-1} . Observe que h é precisamente a ordem de B_h . Além disso, B_h possui altura $h+1$. Considerando as árvores binomiais como árvores ordenadas, as árvores binomiais de mesma ordem são necessariamente isomórfas. A Figura 9.8 ilustra as árvores binomiais B_0 , B_1 , B_2 e B_3 .

O lema abaixo apresenta uma propriedade que justifica a nomenclatura utilizada.

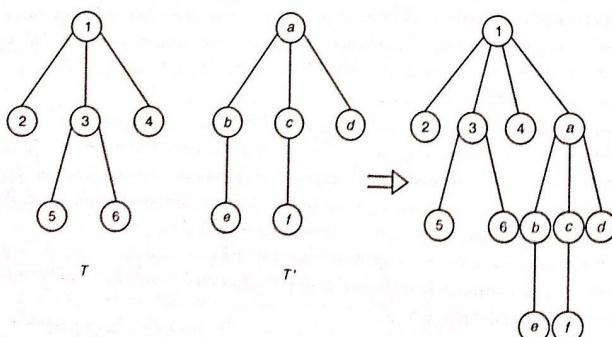


FIGURA 9.7 Operação de subordinação de T , T' .

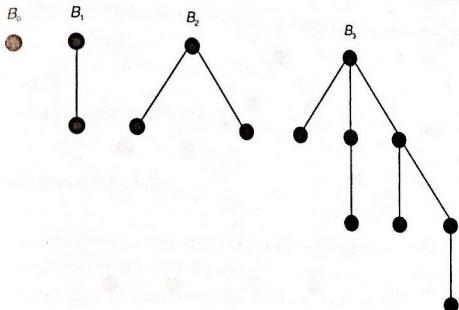


FIGURA 9.8 Árvores binomiais.

Lema 9.1

A árvore binomial B_h possui 2^h nós. Além disso, o número de nós de B_h no nível j é igual a $\binom{h}{j-1}$.

PROVA Para mostrar que B_h possui 2^h nós, utiliza-se a indução. O resultado está correto para $h = 0$. Para $h > 0$, recorde-se que B_h é constituída de duas árvores B_{h-1} . Pela hipótese de indução, B_{h-1} possui 2^{h-1} nós. Logo, B_h possui $2 \cdot 2^{h-1} = 2^h$ nós. Para

mostrar que no nível j de B_h existem $\binom{h}{j-1}$ nós, o argumento é também por indução. Se $j = 1$, B_h contém um único nó no nível j , e o lema é satisfatório. Caso contrário, o nível j de B_h contém nós das duas cópias do B_{h-1} que formam B_h . Os nós do nível j do B_h , que pertencem à cópia de B_{h-1} cuja raiz coincide com a de B_h , também se encontram no nível j desta cópia de B_{h-1} . Os demais nós do nível j de B_h , isto é, os que pertencem à segunda cópia de B_{h-1} , estão no nível $j-1$ desta cópia (Figura 9.9). Logo, usando esta decomposição e aplicando a indução conclui-se que a quantidade de nós de B_h no nível j é igual a

$$\binom{h-1}{j-1} + \binom{h-1}{j-2} = \binom{h}{j-1}.$$

Um heap binomial difere dos outros tipos de heaps estudados até agora, no sentido de que ele não se constitui em uma única árvore, mas em um conjunto de árvores. Considere um conjunto de valores numéricos denominados prioridades. Um *heap binomial* é uma floresta H onde os nós correspondem às prioridades, cada árvore de H obe-

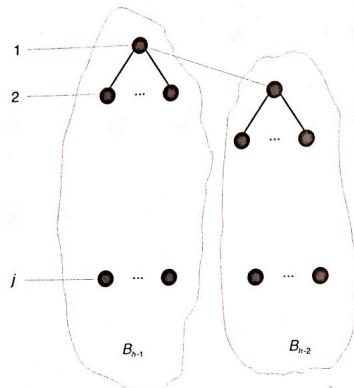


FIGURA 9.9 Lema 9.1.

dece à condição de ordenação (1), isto é, excetuando-se as raízes, a prioridade de um nó é menor ou igual à de seu pai, e satisfazendo à seguinte condição estrutural:

(2") H é formado por árvores binomiais de ordens distintas.

Como exemplo, o conjunto das árvores binomiais $\{B_1, B_3, B_4\}$ constitui um heap binomial. Observe que este heap binomial contém exatamente $2^1 + 2^3 + 2^4 = 26$ nós. Na realidade, este é o único heap binomial que contém 26 nós, conforme pode ser concluído do lema seguinte.

Lema 9.2

Para qualquer $n \geq 0$ existe um único heap binomial, a menos de isomorfismo, contendo exatamente n nós, não considerando as prioridades.

PROVA Como uma árvore binomial B_h possui exatamente 2^h nós, B_h pode ser representada por um número binário que contém o dígito "1" na $(h+1)$ -ésima posição, da direita para a esquerda, e "0" nas demais posições. Como um heap binomial H é formado por árvores binomiais de ordens distintas, H pode ser representado por um número binário, onde cada posição h , da direita para a esquerda, é igual a 1, se $B_h \in H$, ou igual a 0, se $B_h \notin H$.

O valor deste número binário é exatamente igual a n , pois se uma posição h é igual a 1, a árvore $B_h \in H$ contribui com 2^h unidades para o valor de n . Como a representação binária de um número é única, segue-se o lema. ■

Como exemplo, o número 26 corresponde ao heap $\{B_1, B_3, B_4\}$ e ao valor binário 11010.

Corolário 9.1

Um heap binomial com n nós contém $O(\log n)$ árvores binomiais, cada qual de altura $O(\log n)$.

PROVA Lemas 9.1 e 9.2.

A seguir, consideram-se as operações efetuadas pelos heaps binomiais.

Os heaps binomiais podem ser utilizados para realizar a operação de fusão, de forma simples. Sejam H_1, H_2 dois heaps binomiais. A ideia da fusão de H_1, H_2 consiste, inicialmente, em formar uma estrutura H contendo a união das árvores de H_1 e H_2 . Caso todas as árvores de H possuam ordens distintas, H é um heap binomial e o processo se encerra. Caso contrário, H contém duas ocorrências de alguma árvore $B_h \in H_1 \cap H_2$. Escolher a menor h nessas condições. Sejam B'_h, B''_h as duas ocorrências de B_h , sendo a prioridade da raiz do B' maior ou igual à da raiz de B'' . Remover B'_h e B''_h de H , e inserir em H uma árvore binomial B_{h+1} , obtida pela subordinação de B'_h, B''_h . Repetir o processo até que todas as árvores de H possuam ordens distintas.

Como exemplo, sejam $H_1 = \{B_0, B_2, B_4\}$ e $H_2 = \{B_0, B_1\}$ dois heaps binomiais, como na Figura 9.10. Como $B_0 \in H_1 \cap H_2$, removem-se as duas ocorrências de B_0 e insere-se uma nova árvore binomial B_1 , com o nó de prioridade 5 como raiz, e o de prioridade 3, como seu filho. Contudo, a fusão agora contém duas árvores B_1 , a recentemente introduzida e aquela que pertence a H_2 . Essas árvores B_1 são removidas e substituídas por uma árvore B_2 , que possui o nó de prioridade 7, como raiz. Há, agora, uma duplicidade das árvores B_2 , a recentemente introduzida e aquela que pertence a H_1 . Ambas são removidas e substituídas por uma árvore B_3 , com o nó de prioridade 8 na raiz. O processo então se encerra. A fusão de H_1, H_2 produziu o heap binomial $H = \{B_3, B_4\}$, ilustrado na Figura 9.10.

A prova de correção do método decorre do fato de que, na floresta final, todas as árvores são binomiais e distintas. Além disso, a condição de ordenação é mantida através do processo. Observa-se que o heap binomial do resultado corresponde a um número binário igual à soma dos números correspondentes aos heaps que foram fundidos. No exemplo das Figuras 9.9 e 9.10, verifica-se que $11000 = 10101 + 11$.

O algoritmo abaixo descreve o processo de fusão. São dados os heaps binomiais H' , H'' . O algoritmo constrói o heap binomial H , correspondente à fusão de H', H'' . É utilizado o procedimento *subordinar*(B, B'), o qual efetua a operação de subordinação B , B' . Uma árvore B se encontra armazenada na estrutura B , sendo $B[r]$. *chave* o valor de prioridade de sua raiz. Notação similar é utilizada para B' .

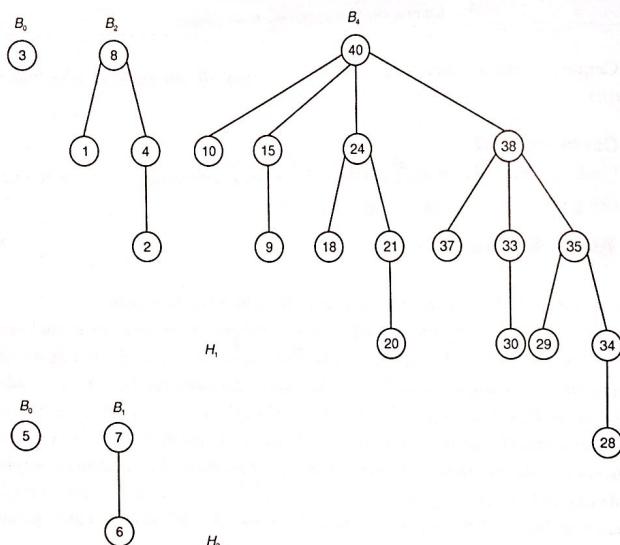


FIGURA 9.10 Exemplo para a fusão.

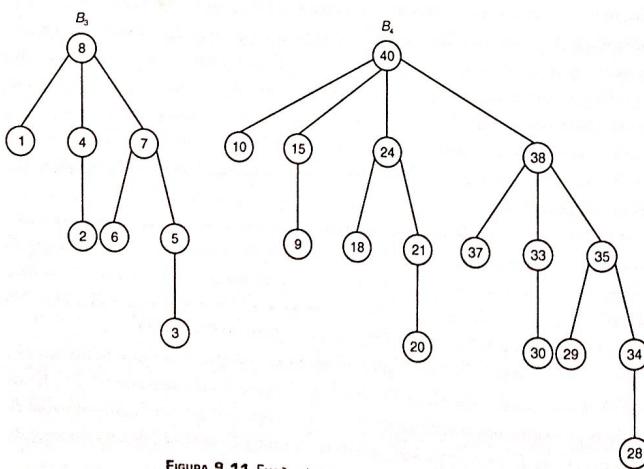


FIGURA 9.11 Fusão dos heaps da Figura 9.10.

Algoritmo 9.2 **Fusão de heaps binomiais**

```

 $H := H' \cup H''$ 
enquanto  $H$  contiver duas árvores  $B, B'$  de mesma ordem faça
    se  $B[r].chave \geq B'[r].chave$  então
        subordinar( $B, B'$ )
    senão subordinar( $B', B$ )
procedimento subordinar( $B', B$ )
    transformar a raiz de  $B'$  no filho mais à direita de  $B$ 

```

Através de técnicas usuais, o algoritmo acima pode ser implementado de tal modo que todas as operações neles contidas sejam executadas em tempo constante, abstraindo-se do número de iterações do bloco **enquanto**. Para tal seria necessário ordenar as árvores de $H := H' \cup H''$, segundo as suas ordens, de modo a propiciar a identificação rápida de árvores de mesma ordem. Considerando que cada um dos heaps H', H'' possui $O(\log n)$ árvores e que a ordem máxima, dentre as árvores, também é $O(\log n)$, conclui-se que a ordenação das árvores, segundo suas ordens, pode ser realizada em tempo $O(\log n)$. Por outro lado, como H' e H'' são heaps binomiais, o número máximo de árvores binomiais em $H' \cup H''$, com ordens repetidas, é igual a 2. Assim sendo, o laço **enquanto** é executado $O(\log n)$ vezes. Portanto, a complexidade do algoritmo de fusão é $O(\log n)$.

A seguir, são descritas as demais operações dos heaps binomiais.

A exemplo de heaps esquerdistas, a operação de inserção em heaps binomiais é estudada como um caso particular da fusão. Com efeito, para inserir um novo nó v , com prioridade p em um heap binomial H' , define-se um outro heap binomial H'' , composto de uma única árvore binomial B_i , em que B_i é formada pelo (único) nó v , com prioridade p . Então a inserção de v em H' é equivalente a efetuar a fusão H', H'' . Portanto, a inserção em heaps binomiais pode ser realizada em $O(\log n)$ passos. Contudo, pode-se esperar que o comportamento dos heaps binomiais, em relação à inserção, seja ainda melhor, conforme o lema seguinte.

Lema 9.3

O tempo esperado para efetuar a operação de inserção em um heap binomial com n nós é constante.

PROVA Seja R a representação binária do número n , onde j denota a posição do primeiro dígito 0 de R , da direita para a esquerda. Então, para completar uma operação de inserção são necessárias j fusões de árvores binomiais. Pode-se supor que os 0's e 1's, na representação R , possuam idêntica probabilidade de ocorrência, para cada

poção. Assim, a probabilidade de cada 0 é igual a $1/2$. Logo, o número esperado de fusões de árvores binomiais para completar uma inserção é igual a 2. Isto implica que o tempo esperado para completar a inserção é constante.

A operação da remoção do nó de maior prioridade de um heap binomial também pode ser realizada de maneira eficiente, utilizando a fusão, conforme descrito a seguir.

Seja H um heap binomial. O nó de maior prioridade em H , naturalmente, é a raiz de alguma árvore de H . Inicialmente, localiza-se a árvore B_h de H , cuja raiz é o nó que se deseja remover. Removendo B_h de H , obtém-se o heap binomial $H' = H - B_h$. Em seguida, removendo a raiz $r(B_h)$ da árvore B_h obtém-se uma coleção de árvores binomiais B_0'', \dots, B_{h-1}'' , que formam um heap binomial H'' . Finalmente, efetuando a fusão H' , H'' obtém-se um heap binomial formado por todos os nós de H , exceto o de maior prioridade, o que constitui a solução esperada.

Por exemplo, para efetuar a remoção do nó de maior prioridade do heap binomial H da Figura 9.12, observa-se que esse nó corresponde à raiz da árvore B_2 , de prioridade 25. Removendo esta árvore de H , obtém-se o heap $H' = \{B_0, B_3\}$. Por outro lado, removendo-se o nó de maior prioridade da árvore B_2 , obtém-se o heap binomial de H'' formado por uma árvore B_0 , contendo o nó de prioridade 20 e por uma árvore B_1 , com os nós de prioridades 21 e 15. Finalmente, efetuando a fusão de H' , H'' produz o resultado esperado, que se encontra na Figura 9.13.

É bastante simples determinar a complexidade da remoção do nó de maior prioridade. Como o heap binomial H possui $O(\log n)$ árvores, para determinar a árvore B_h cuja

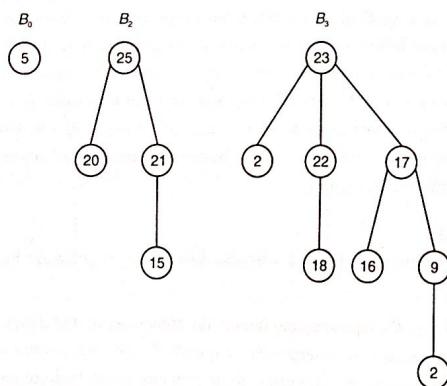


FIGURA 9.12 Exemplo para a remoção.

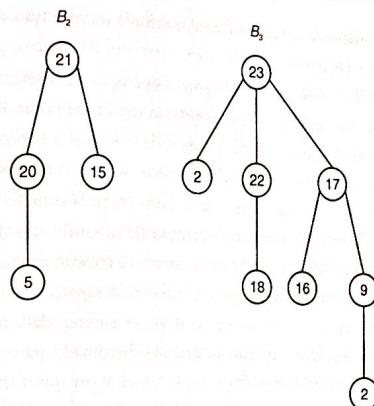


FIGURA 9.13 Resultado da remoção.

raiz é o nó de maior prioridade, basta percorrer essas raízes e determinar o nó procurado em $O(\log n)$ passos.

A construção de $H' = H - B_h$ pode ser realizada em tempo constante, enquanto $H'' = \{B_0, \dots, B_{h-1}\}$ pode ser obtido em $O(\log n)$ passos. Assim sendo, a fusão H', H'' requer $O(\log n)$ passos, que é a complexidade final da operação.

Finalmente, a exemplo dos heaps tradicionais e dos heaps esquerdistas, a construção de um heap binomial com n elementos pode ser realizada em tempo $O(n)$ (Exercício 9.3).

Para encerrar o estudo dos heaps binomiais, analisamos em seguida as operações de inserção, fusão e remoção, sob o aspecto da complexidade amortizada. De fato, verificamos que as duas primeiras operações podem ser realizadas em tempo amortizado $O(1)$. Contudo, o algoritmo de remoção do nó de maior prioridade requer tempo amortizado $O(\log n)$.

Seja H um heap binomial. Para a análise da complexidade amortizada das operações realizadas em H , é utilizado o método de visão do banqueiro. A cada árvore B_h que pertence ao heap binomial H é associada uma unidade de crédito. Esses créditos serão utilizados para efetuar diferentes operações.

Cada vez que um novo nó v é inserido em um heap binomial, é criada uma nova árvore binomial contendo somente o nó v . A esta árvore é concedida uma unidade de crédito. Novas árvores em H podem também ser introduzidas através da operação de remoção do nó de maior prioridade do H . Referindo-se ao algoritmo de remoção, recorda-se que o seu primeiro passo consiste em remover a raiz r da árvore que contém o nó de maior prioridade. Essa remoção pode produzir $O(\log n)$ árvores novas, constituídas pelos filhos de r . A cada uma dessas árvores será concedida também uma unidade de crédito.

de maior prioridade é $\Omega(\log n)$.

Um resultado melhor pode ser obtido para a operação de inserção de um novo nó v . Essa operação cria uma nova árvore F , contendo um único nó, v . Em seguida se realiza a fusão do heap H com a árvore F . A ordem de F é igual a 1. Nesse caso é possível incorporar F a H , percorrendo sequencialmente em ordenação não crescente, uma lista das ordens de H , a qual é mantida, juntamente com H . Se H contém uma árvore B'_0 de ordem 0, então B_0, B'_0 são submetidas à operação de subordinação, formando-se uma nova árvore B_1 da ordem 1. O custo da subordinação é coberto com o crédito existente na árvore que se tornou subordinada, pois essa árvore desaparece na operação. Caso H contenha uma outra árvore B'_1 , de ordem 1, deve ser empregada a subordinação para unir B_1, B'_1 e utiliza-se o crédito existente na árvore subordinada, para custear a operação de subordinação, e assim por diante. O processo se repete, até que a árvore resultante da última subordinação efetuada possua ordem diferente de qualquer outra de H . Em cada subordinação, a árvore subordinada desaparece, e o crédito a ela atribuído é utilizado para custear essa operação. Não há dificuldade em implementar este processo, de modo a manter atualizada a lista das ordens de H , em ordem crescente, em um tempo proporcional ao número de subordinações efetuadas. Assim sendo, a complexidade amortizada da operação de inserção é igual a $O(1)$.

Por exemplo, seja a inserção de um novo nó no heap binomial $H = \{B_0, B_1, B_2, B_4, B_7\}$. Cada uma dessas árvores possui uma unidade de crédito. A inserção dá origem a uma nova árvore B'_0 . É necessário efetuar a subordinação B'_0, B_0 , custeada com o crédito de B_0 , gerando uma nova árvore B'_1 , a qual recebe o crédito de B'_0 , que desaparece. Esse processo é repetido até que todas as ordens sejam distintas. O heap final obtido é $H = \{B_1, B_4, B_7\}$.

Em seguida, examina-se a operação da fusão, sob o ponto de vista da complexidade amortizada. Dados dois heaps binomiais, H_1 e H_2 , o algoritmo original consistia em unir H_1 e H_2 , formando a floresta H . Em seguida, iterativamente, eliminar a ocorrência de árvores B'_n e B''_n de mesma ordem em H , através da operação de subordinação, até que todas as árvores de H tenham ordens distintas, transformando H , efetivamente, num heap binomial. Este processo será ligeiramente alterado, conforme abaixo.

A ideia, agora, consiste em adiar, convenientemente, a eliminação da duplicidade de ordens idênticas em H . Isto não altera novas operações de fusão ou inserção. Assim, de um modo geral, supõe-se que H seja uma floresta formada por conjuntos disjuntos de heaps binomiais, fruto da realização de operações de união, do passo inicial das fusões. A recomposição de H de forma a constituir um único heap binomial é adiada até a realização de uma operação de remoção do nó de maior prioridade de H . Nessa ocasião se realiza a eliminação da duplicidade de ordens. A fusão efetuada com esta estratégia de adiamento é denominada *fusão preguiçosa*.

O processo de eliminação de duplicidade de ordens, em si, permanece o mesmo, utilizando-se a operação de subordinação, que transforma duas árvores binomiais de ordem b em uma única de ordem $b + 1$. Caso já existisse em H outra árvore de ordem $b + 1$, seria necessário, novamente, utilizar a subordinação para transformar essas duas árvores da ordem $b + 1$ em uma única de ordem $b + 2$, e assim por diante. Para realizar a tarefa de identificação das duplicidades, utiliza-se uma lista ordenada das ordens das árvores para cada heap que forma a floresta H .

Uma razão para a adoção da estratégia de adiar a eliminação da duplicidade de ordens, até a realização de alguma operação de remoção, reside no fato de que esta última recebe $O(\log n)$ créditos, os quais podem ser utilizados por amortizar partes de outras operações, se necessário.

O próximo passo é determinar a expressão da complexidade amortizada. O algoritmo se inicia pela união de dois heaps. Isto pode ser implementado como uma concatenação de listas, em tempo $O(1)$. A eliminação das duplicidades de ordens é adiada até a ocorrência da próxima operação de remoção. No momento que imediatamente precede a remoção, seja H a floresta de árvores binomiais construídas, $|H| = m$. Para o processo de eliminação das duplicidades de ordens, é mantida a ordenação das ordens em cada heap binomial de H . Além disso, efetuar as operações de subordinação. Seja k o número de subordinações realizado. Cada uma dessas operações requer tempo $O(1)$, o qual pode ser custeado através do crédito existente na árvore subordinada, que desaparece. Resta determinar o custo da manutenção da ordenação das ordens das árvores. Como existem m árvores e a ordem máxima é $O(\log n)$, a ordenação pode ser realizada em tempo $O(m + \log n)$. Mas $m + \log n = k + \log n + m - k$, e como cada subordinação implica decrescer o número de árvores em uma unidade, conclui-se que, após a eliminação das repetições de ordens, H é um heap binomial com $m - k$ árvores. Como um heap binomial contém $O(\log n)$ árvores, obtém-se $O(m + \log n) = O(k + \log n + m - k) = O(k + \log n)$. Como o total do crédito disponível é $O(k + \log n)$, é possível efetuar a ordenação desejada.

Debitando todo custo logarítmico à operação de remoção, é possível afirmar que a complexidade amortizada da fusão é igual a $O(1)$ por operação realizada.

9.5 Heaps de Fibonacci

Os *heaps de Fibonacci* constituem uma generalização dos heaps binomiais. De fato, a sua estrutura é derivada destes últimos. Eles possuem uma aplicação mais geral. Utilizando heaps de Fibonacci é possível efetuar todas as operações realizadas pelos heaps binomiais, como fusão, inserção e remoção do elemento de maior prioridade. Adicionalmente, com os heaps de Fibonacci generaliza-se a operação de remoção, de modo a remover um nó qualquer. Além disso, efetua-se a operação de aumento qualquer de prioridade. Os heaps

de Fibonacci são bastante eficientes sob o aspecto da complexidade amortizada. Assim, sendo, é possível efetuar a fusão, inserção ou aumento de prioridade em $O(1)$. Contudo, o algoritmo de remoção requer tempo amortizado $O(\log n)$.

Em uma árvore de Fibonacci, cada nó pode estar *marcado* ou *desmarcado*. Utilizando essas possíveis marcas, as árvores de Fibonacci podem ser definidas de forma recursiva do seguinte modo:

- Um nó isolado desmarcado é uma árvore de Fibonacci.
- A árvore obtida pela subordinação de duas árvores de Fibonacci de mesma ordem também é uma árvore de Fibonacci.
- Se F é uma árvore de Fibonacci e v é um nó desmarcado de F , então a árvore obtida de F pela remoção da subárvore cuja raiz seja algum filho de v também é uma árvore de Fibonacci. Após a remoção, v se torna marcado, exceto se for a raiz, caso em que mantém o seu estado de desmarcado.

Em particular, conclui-se que toda árvore binomial é árvore de Fibonacci.

A árvore da Figura 9.14(a) é binomial, e portanto também é de Fibonacci. A Figura 9.14(b) apresenta também uma árvore de Fibonacci, pois a mesma foi obtida da árvore binomial da Figura 9.14(a) pela remoção de duas subárvores de nós distintos. Contudo, a árvore da Figura 9.14(c) não é de Fibonacci.

Vale lembrar que a ordem de um nó v é igual ao número de filhos de v , e a ordem da árvore é igual à ordem da sua raiz. Uma característica importante que as árvores binomiais possuem é o fato de a ordem ser logarítmica no número de nós da árvore. Naturalmente, isto enseja a construção de algoritmos mais eficientes. Através do Lema 9.1, sabe-se que as árvores binomiais possuem esta propriedade. As árvores de Fibonacci são obtidas a partir das árvores binomiais pela remoção de

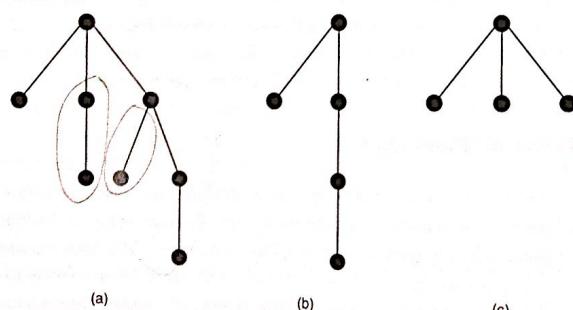


FIGURA 9.14 Qual destas árvores não é de Fibonacci?

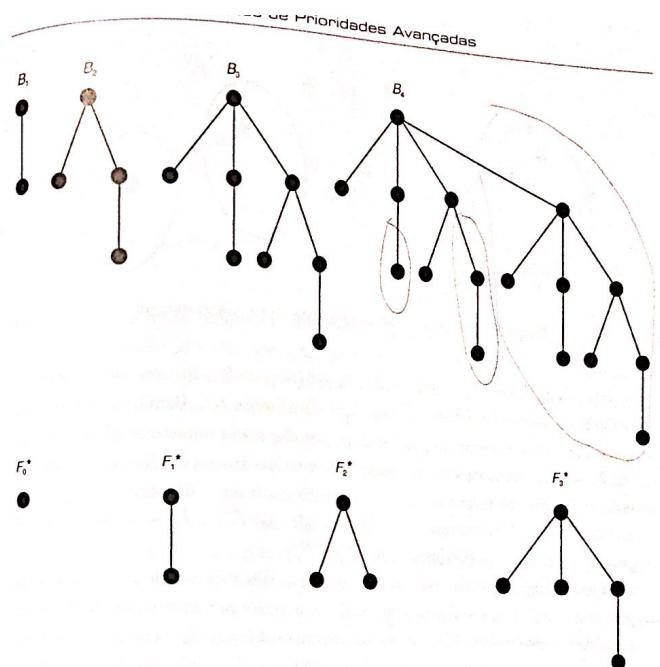


FIGURA 9.15 Árvores de Fibonacci mínimas.

nós ou subordinação de árvores de Fibonacci. Isso talvez pudesse alterar a estrutura logarítmica da ordem. Contudo, isso não ocorre, conforme o resultado do próximo teorema.

Uma árvore de Fibonacci mínima F_h^* é uma árvore de Fibonacci de ordem h com um número mínimo de nós. Isto é, dentre todas as árvores de Fibonacci de ordem h obtidas da árvore binomial B_{h+1} , a árvore F_h^* é a que contém o menor número de nós. A Figura 9.15 ilustra árvores binomiais B_{h+1} juntamente com as de Fibonacci mínima F_h^* .

O teorema a seguir mostra que a ordem de uma árvore de Fibonacci é necessariamente logarítmica no número de nós da árvore.

Teorema 9.3

Seja F_h uma árvore de Fibonacci de ordem h . Então $h = O(\log |F_h|)$.

PROVA Seja F_h a árvore de Fibonacci mínima de ordem h . Naturalmente, basta mostrar que $h = O(\log |F_h^*|)$, pois $|F_h^*| \leq |F_h|$. A ideia agora é examinar a construção de F_h^* .

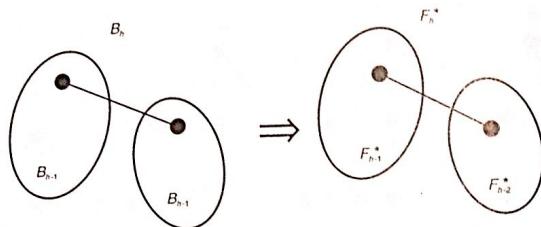


FIGURA 9.16 Decomposição da árvore de Fibonacci mínima.

Pela minimalidade de F_h^* sabe-se que de cada nó v de B_{h+1} foi removido exatamente um filho, juntamente com sua subárvore, para formar F_h^* . Além disso o filho de v que foi removido é aquele que, por sua vez, possui o maior número de filhos. A construção acima sugere uma decomposição de F_h em duas árvores de Fibonacci mínimas, utilizando uma operação reversa à da subordinação, isto é, a decomposição em duas árvores. Assim F_h^* é decomposto nas árvores mínimas F_{h-1}^* e F_{h-2}^* , de forma que F^* possa ser obtida pela subordinação de F_{h-2}^* a F_{h-1}^* .

Observe que o filho de maior ordem dentre os filhos da raiz de B_{h+1} foi retirado para formar F_h^* . Então o filho da raiz de F_h^* com maior ordem é o filho de B_{h+1} com a segunda maior ordem. Este nó é a raiz de uma subárvore B_{h-1} em B_{h+1} . Portanto, em F_h^* , esta subárvore corresponde a uma subárvore F_{h-2}^* . Os nós de $F_h^* - F_{h-2}^*$ correspondem à subárvore B_h de B_{h+1} cuja raiz se tornou a raiz de B_{h+1} , quando da formação de B_{h+1} a partir de duas árvores B_h . Então, em F_h^* , esta subárvore corresponde a F_{h-1}^* .

Portanto, F_h^* pode ser decomposto na árvore de Fibonacci mínima F_{h-1}^* , a cuja raiz é atribuído mais um filho que, por sua vez, é raiz de uma árvore de Fibonacci mínima F_{h-2}^* (Figura 9.16).

Logo, $|F_h^*| = |F_{h-1}^*| + |F_{h-2}^*|$, e além disso sabe-se que $|F_0^*| = 1$ e $|F_1^*| = 2$. O valor de $|F_h^*|$ corresponde, pois, ao h -ésimo termo da série de Fibonacci. Conforme o Capítulo 5, este valor é exponencial em h .

Em seguida examinam-se os heaps de Fibonacci.

Define-se heap de Fibonacci de maneira análoga a heap binomial. Isto é, um heap de Fibonacci é um conjunto de árvores de Fibonacci de ordens distintas, que satisfazem à condição de ordenação (prioridade de cada nó é menor ou igual à de seu pai, se ele existir).

A definição acima e o Teorema 9.3 implicam que um heap de Fibonacci possui $O(\log n)$ árvores, cada qual de ordem $O(\log n)$. O próximo passo consiste em estudar as operações efetuadas pelos heaps de Fibonacci. As operações de inserção e fusão são realizadas de

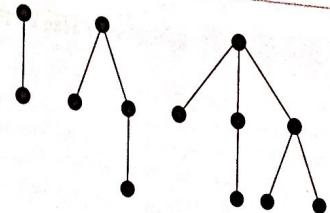


FIGURA 9.17 Heap de Fibonacci.

forma análoga a heaps binomiais, utilizando o algoritmo de complexidade amortizada $O(1)$. Resta estudar as novas operações de remoção de um nó qualquer e aumento de uma prioridade qualquer no heap de Fibonacci H . A operação de remoção em H deve ser realizada de forma controlada, pois, segundo a definição, no máximo um filho pode ser removido de cada nó diferente da raiz. Para manter esta informação utiliza-se um conjunto de marcas. Para cada nó $v \neq$ raiz, se nenhum filho de v foi removido, até o momento, então v está desmarcado. Caso contrário, exatamente um filho foi removido de v , e então v está marcado. A raiz da árvore está sempre desmarcada.

O processo é descrito a seguir. No início todos os nós estão desmarcados. Considere a remoção de um nó w de H , sendo h a ordem de w . Supõe-se que seja dado diretamente um ponteiro para w . Nesse caso, a primeira tarefa consiste em efetivamente remover w da árvore F de H , que contém w . A remoção de w cria h novas árvores de Fibonacci, cujas raízes são os filhos de w . Essas árvores deverão ser posteriormente incorporadas a H , através de uma fusão. Em seguida é necessário verificar se algum irmão de w fora anteriormente removido. Caso negativo, v , pai de w , está desmarcado, então deve-se marcar v , desde que $v \neq$ raiz. Caso contrário, v está marcado, portanto algum outro filho de v já foi anteriormente removido. A solução é cortar v de seu pai (isto é, remover de F toda subárvore $T(v)$). Posteriormente, $T(v)$ deverá ser reincorporado a H , através de uma operação de fusão. Esta situação pode se propagar em direção ascendente em F . Isto é, o procedimento executado para v deve ser repetido para o pai de v , eventualmente para o avô de v , e assim por diante. A iteração se encerra quando é atingido o ancestral mais próximo de w que esteja desmarcado. Como a raiz da árvore está sempre desmarcada, a iteração termina. Esse processo se denomina cortes em cascata. A reincorporação a H das subárvore $T(v)$ cortadas de F é realizada através da fusão. Deve ser observado que na operação de subordinação, durante as fusões, a raiz da árvore subordinadora torna-se desmarcada.

O algoritmo abaixo formula o processo descrito. A finalidade é remover w do heap de Fibonacci H . No caso, F é a árvore de H que contém w . Seja v um nó de F , seja $F(v)$ a subárvore de F com raiz v . Denote por $F(v) - v$ o conjunto das subárvore formadas removendo-se o nó v de $F(v)$. São dados de entrada o heap de Fibonacci H e o nó w , diferente da raiz de H , a ser removido.

Algoritmo 9.3 Remoção em heap de Fibonacci

```

 $F :=$  árvore de  $H$ , contendo  $w$ 
 $H' := F(w) - w$ 
 $v :=$  pai de  $w$ 
remover  $T(w)$  de  $F$ 
enquanto  $v$  marcado faça
    desmarcar  $v$ 
    remover  $T(v)$  de  $F$ 
     $H' := H' \cup \{v\} \cup (F(v) - v)$ 
     $v :=$  pai de  $v$ 
    se  $v \neq$  raiz então
        efetuar a fusão de  $H, H'$ 

```

Efetua-se a seguir a análise amortizada do algoritmo.

Inicialmente, observa-se que o processo de cortes em cascata corresponde ao bloco enquanto do algoritmo. Determina-se, inicialmente, a complexidade amortizada da realização dos cortes da cascata.

A análise da complexidade amortizada é simples. A questão seria como distribuir o custo dos cortes das subárvore, visando obter uma complexidade amortizada $O(1)$. Essa questão pode ser resolvida observando-se que cada corte de alguma subárvore $F(v)$, sen-
do v um ancestral de w , somente pode ocorrer se v estiver anteriormente marcado. Isto significa que alguma outra subárvore de um filho de v foi cortada de F anteriormente.

A ideia então é debitar o custo do corte de $F(v)$, bem como o custo de uma possível operação de subordinação de $F(v)$, à operação anterior que produziu a marca em v , isto é, que cortou o primeiro filho de v . Isto pode ser realizado, pois o corte de $F(v)$ requer $O(1)$ passos e contou sequentemente iria onerar aquela operação anterior em $O(1)$ passos. Com isso, somente restou para o processo de cortes em cascata, um total de $O(1)$ passos, todos requerendo tempo $O(1)$. Assim, a complexidade amortizada dos cortes em cascata é $O(1)$, já incluindo o tempo da possivel subordinação posterior das árvores cortadas durante o processo de corte em cascata.

Quanto à parte inicial do algoritmo, observa-se que são criadas h novas subárvore, oriundas dos filhos de w . Para realizar a possível subordinação posterior dessas subárvore, devem ser realizados $O(h)$ passos. Logo, a complexidade deste processo é $O(\log n)$. Assim sendo, a complexidade amortizada da remoção é $O(\log n)$.

Em seguida, considera-se a operação de aumento de prioridade de um certo nó w . Seja H um heap, w um nó de H com prioridade p , pertencente à árvore F de H . Deseja-se aumentar a prioridade p para o valor $p' > p$. Supõe-se que seja fornecido um ponteiro diretamente para w . A ideia geral para se efetuar esta operação é descrita abaixo:

A primeira medida a ser efetuada é aumentar efectivamente a prioridade de w , de p para p' . Então, se a prioridade do pai v de w é maior ou igual a p' , o processo se encerra. Caso contrário, w está violando a condição de ordenação do heap. A solução consiste em cortar a subárvore $F(w)$ de F , e reincorporá-la a H através de uma fusão, mais adiante. Para todos os efeitos, foi retirado um dos filhos do nó v , pai de w . Torna-se então necessário verificar se alguma subárvore de v já fora anteriormente cortada de F , e assim iterativamente, segundo um processo similar ao de cortes em cascata. Finalmente, todas as subárvore cortadas de F devem ser reincorporadas através de fusões.

A estratégia acima conduz diretamente a um algoritmo para aumento de prioridade de um nó w , similar ao algoritmo de remoção de w . A diferença básica é que, na remoção de w , os filhos de w podem gerar h novas subárvore, enquanto no aumento de prioridade de w , T_w somente pode gerar uma nova subárvore. A determinação da complexidade dos cortes em cascata é similar nos dois casos. Pode-se recordar que o único motivo pelo qual a complexidade da remoção de um nó w é $O(\log n)$ deve-se ao fato de que a remoção de w pode gerar $O(\log n)$ novas árvores. As demais operações podem ser efetuadas em complexidade amortizada $O(1)$. Logo, pode-se concluir que a complexidade amortizada do aumento de prioridade é $O(1)$.

9.6 Outras Variantes

Além dos m -heaps, heaps esquerdistas, heaps binomiais e heaps de Fibonacci examinados neste capítulo, existem outras variantes deste tipo de listas de prioridades que merecem destaque. Nesta seção, serão abordados brevemente os heaps distorcidos e os heaps relaxados.

Os heaps distorcidos podem ser compreendidos como heaps esquerdistas autoajustáveis. Isto é, os heaps distorcidos tenderiam a se constituir em heaps esquerdistas, porém a condição estrutural ($2'$) dos heaps esquerdistas (o comprimento de caminho curto da subárvore esquerda de qualquer nó é maior ou igual ao de subárvore direita) não é necessariamente mantida ao longo do processo. Contudo, a condição de ordenação (1) é mantida.

A forma de operação de um heap distorcido é similar à dos heaps esquerdistas. Contudo, o valor do comprimento de caminho curto não é armazenado. Com isso, o comprimento do caminho direito de um heap distorcido pode ser arbitrariamente grande. Este fato implica que as operações de inserção, remoção do elemento de maior prioridade e fusão possam consumir até $O(n)$ passos cada uma. Contudo, é possível provar que a realização de m operações consecutivas desse tipo pode ser efetuada em $O(m \log n)$ passos. Isto é, o tempo amortizado para a inserção, remoção do elemento de maior prioridade ou fusão é de $O(\log n)$ para operação.

A questão do autoajuste nos heaps distorcidos pode ser observada na operação de fusão. A fusão destes heaps é realizada de maneira similar à dos heaps esquerdistas, exceto no procedimento onde seria necessário utilizar o comprimento do caminho curto. Isto é, para efetuar a fusão de dois heaps H_1, H_2 inicialmente é realizada a fusão de um dos heaps com o

sub-heap direito do outro. Para tal utiliza-se um processo recursivo. Em seguida, efetua-se uma nova fusão, desta vez do heap resultado da primeira fusão com o sub-heap ainda não manipulado (ver Seção 9.3). Com isso, obtém-se uma estrutura H , que contém todos os elementos de H_1, H_2 . Para completar a fusão, nos casos dos heaps esquerdistas, é verificado se o comprimento do caminho curto da subárvore esquerda de H é menor do que o da direita, e em caso positivo realiza-se a troca de subárvores, esquerda e direita, da raiz. No caso dos heaps distorcidos esta troca é sempre efetuada, independente da verificação.

Finalmente, vale mencionar a existência de uma variação dos heaps de Fibonacci, conhecida como *heaps relaxados*. Uma característica importante dos heaps relaxados é que esta estrutura permite efetuar a operação de diminuição de prioridade de um nó em tempo amortizado de $O(1)$.

9.7 Exercícios

- 9.1 Descrever algoritmos para aumento e diminuição de prioridades de um m -heap.
- 9.2 Descrever um algoritmo para inserir um novo elemento em um m -heap, bem como um algoritmo para remover o elemento de maior prioridade.
- 9.3 Desenvolver um algoritmo de complexidade $O(n)$ para construir um heap binomial com n elementos.
- 9.4 Desenhar o heap esquerdista que resulta da fusão dos heaps esquerdistas H_1 e H_2 da Figura 9.18.
- 9.5 Desenhar o heap binomial que resulta da fusão dos heaps binomiais H_1, H_2 , da Figura 9.19.
- 9.6 Seja um heap binomial composto de k árvores binomiais. Determinar as alturas que essas árvores devem possuir para que a operação de inserção corresponda ao pior caso, isto é, seja realizada em um número máximo de passos.
- 9.7 Seja H o heap binomial composto de k árvores e tal que a árvore cuja raiz possui a maior prioridade é de altura b . Determinar as alturas que as árvores de H devem ter, bem como

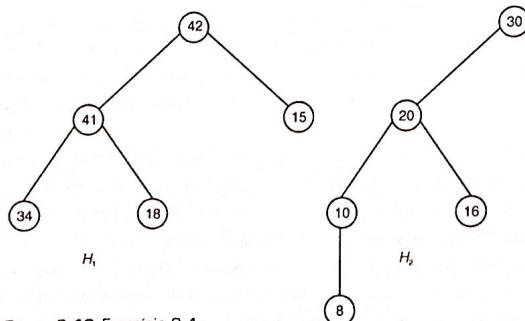


FIGURA 9.18 Exercício 9.4.

Listas de Prioridades Avançadas

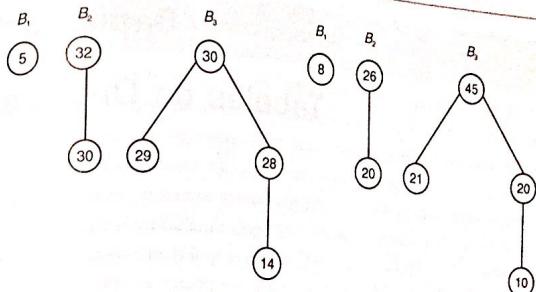


FIGURA 9.19 Exercício 9.5.

mo determinar o valor de b , para que a operação de remoção do nó de maior prioridade de H seja realizada em um número máximo de passos.

9.8 Sejam H_1, H_2 dois heaps esquerdistas. Determinar as condições que H_1, H_2 devem satisfazer, para que na operação de fusão de H_1, H_2 não seja necessário realizar a troca das subárvores, esquerda e direita, da árvore binária obtida, após as duas fusões parciais realizadas nos passos recursivos da operação de fusão H_1, H_2 .

9.9 Provar ou dar contraxeemplo:

Sejam H_1 e H_2 dois heaps binomiais contendo k_1 e k_2 árvores binomiais, respectivamente. Seja H o heap binomial resultante da fusão de H_1, H_2 , o qual contém k árvores binomiais. Então $k \geq \min(k_1, k_2)$.

9.10 Provar ou dar contraxe exemplo:

Sejam H_1 e H_2 dois heaps binomiais, contendo k_1 e k_2 árvores binomiais, respectivamente. Seja H o heap binomial resultante da fusão de H_1, H_2 , o qual contém k árvores binomiais. Então, $k \leq k_1 + k_2$.

9.11 Determinar a expressão da altura, mínima e máxima, de um heap de Fibonacci em função do posto.

Notas Bibliográficas

Parte da bibliografia relativa a heaps já foi apresentada no Capítulo 6. Os m -heaps foram estudados, em primeiro lugar, por Johnson [Jo75], num contexto de algoritmos para árvores geradoras mínimas. Os heaps esquerdistas foram introduzidos por Crane [Cr72]. Um estudo importante desses heaps foi realizado por Knuth [Kn73]. Os heaps binomiais foram introduzidos por Vuillemin [Vu78], e seu estudo inicial foi complementado por Brown [Br78]. Os heaps de Fibonacci foram desenvolvidos por Fredman e Tarjan [Fr87]. Uma das motivações para a concepção dessa estrutura foi a sua utilização para diminuir as complexidades dos algoritmos de caminho mínimo e árvore geradora mínima. Os heaps relaxados foram desenvolvidos por Driscoll, Gabow, Shraiman e Tarjan [Dr88] como uma alternativa possível para os heaps de Fibonacci. Uma outra forma de heaps com autoajuste foi desenvolvida por Fredman, Sedgewick, Sleator e Tarjan [Fr86]. Entre os textos gerais de estruturas de dados que apresentam um tratamento bastante completo em relação a heaps podem ser mencionados os de Kozen [Ko92] e Weiss [We95]. Esses textos contêm vários tópicos de interesse relativos à árvores binomiais e de Fibonacci. Os heaps distorcidos correspondem aos *skew heaps*, na nomenclatura inglesa.