AULA 8

LISTAS DE PRIORIDADES

Nesta aula vamos estudar uma nova estrutura de dados chamada de lista de prioridades. Em uma lista como esta, as relações entre os elementos do conunto que compõe a lista se dão através das suas prioridades. Perguntas como qual o elemento com a maior prioridade e operações de inserção e remoção de elementos deste conjunto, ou alteração de prioridades de elementos da lista, são tarefas associadas a estruturas como esta. Listas de prioridades são estruturas simples e eficientes para solução de diversos problemas práticos como, por exemplo, o escalonamento de processos em um computador.

Esta aula é baseada especialmente nas referências [1, 2, 13].

8.1 *Heaps*

Antes de estudar as listas de prioridades, precisamos estudar com cuidado uma outra estrutura de dados, que é base para aquelas, chamada *heap*[1]. Um heap nada mais é que uma estrutura de dados armazenada em um vetor. Vale observar que cada célula do vetor pode conter um registro com vários campos, mas o campo mais importante é aquele que armazena um número, a sua **prioridade** ou **chave**. Como os outros dados associados à prioridade são supérfluos para o funcionamento de um heap, optamos por trabalhar apenas com um vetor de números inteiros para abrigá-lo, onde cada célula sua contém uma prioridade. Dessa forma, um heap é uma coleção de elementos identificados por suas prioridades armazenadas em um vetor numérico S satisfazendo a seguinte propriedade:

$$S[\lfloor (i-1)/2 \rfloor] \geqslant S[i]\,, \tag{8.1}$$

para todo $i \geqslant 1$. Um vetor S com a propriedade (8.1) é chamado um **max-heap**. Do mesmo modo, a propriedade (8.1) é chamada **propriedade max-heap**. O vetor S apresentado na figura 8.1 é um max-heap.

	0	1	2	3	4	5	6	7	8	9
S	26	18	14	16	8	9	11	4	12	6

Figura 8.1: Um max-heap com 10 elementos.

[1] A tradução de *heap* do inglês é monte, pilha, amontoado, montão. O termo *heap*, sem tradução, será usado daqui por diante, já que se encontra bem estabelecido na área.

Se o vetor S tem a propriedade

$$S[\lfloor (i-1)/2 \rfloor] \leqslant S[i]\,, \tag{8.2}$$

para todo $i \geqslant 1$, então S é chamado **min-heap** e a propriedade (8.2) é chamada **propriedade min-heap**. Max-heaps e min-heaps são semelhantes mas, para nossas aplicações neste momento, estamos interessados apenas em max-heaps.

Um max-heap pode também ser visto como uma **árvore binária**[2] com seu último nível preenchido da esquerda para direita. Uma ilustração do max-heap da figura 8.1 é mostrada na figura 8.2.

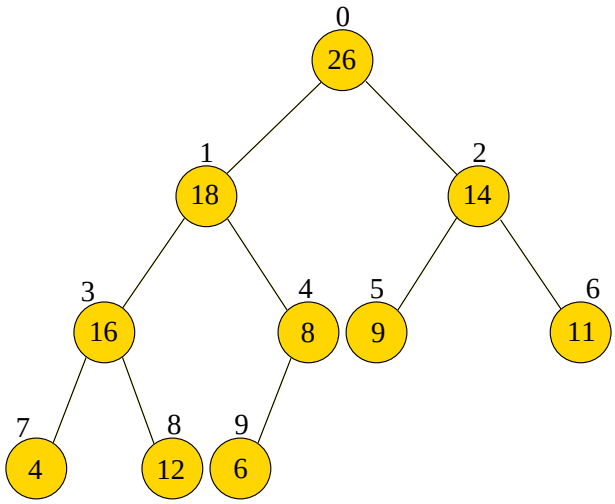

Figura 8.2: Representação do max-heap da figura 8.1 como uma árvore binária, com seu último nível preenchido da esquerda para diereita.

Observe que a visualização de um max-heap como uma árvore binária nos permite verificar a propriedade (8.1) facilmente. Em particular, note que o conteúdo de um nó da árvore é maior ou igual aos conteúdos dos nós que são seus filhos. Por conta dessa semelhança, podemos descrever operações básicas que nos permitem percorrer o max-heap facilmente. As operações são implementadas como funções.

```
/* Recebe um índice i em um max-heap e devolve o "pai" de i */
int pai(int i)
{
   if (i == 0)
      return 0;
   else
      return (i - 1) / 2;
}
```

[2] Uma árvore para os computólogos é um objeto bem diferente daquele visto pelos biólogos. Sua raiz está posicionada "no ar" e suas folhas estão posicionadas "no chão". A árvore é binária porque todo nó tem, no máximo, dois filhos. A definição formal de uma árvore binária foge ao escopo de Algoritmos e Programação II e será vista mais adiante.

```
/* Recebe um índice i em um max-heap e devolve o "filho esquerdo" de i */
int esquerdo(int i)
{
   return 2 * (i + 1) - 1;
}
```

```
/* Recebe um índice i em um max-heap e devolve o "filho direito" de i */
int direito(int i)
{
   return 2 * (i + 1);
}
```

Dadas as operações implementadas nas três funções acima, a propriedade (8.1) pode então ser reescrita da seguinte forma:

$$S[\,\texttt{pai(}i\texttt{)}\,] \geqslant S[i]\,, \tag{8.3}$$

para todo i, com $i \geqslant 0$.

Se olhamos para um max-heap como uma árvore binária, definimos a **altura** de um nó no max-heap como o número de linhas ou arestas no caminho mais longo do nó a uma folha. A **altura do max-heap** é a altura da sua raiz. Como um max-heap de n elementos pode ser visto como uma árvore binária, sua altura é proporcional a $\log_2 n$. Como veremos a seguir, as operações básicas sobre heaps têm tempo de execução proporcional à altura da árvore binária, ou seja, $O(\log_2 n)$.

8.1.1 Manutenção da propriedade max-heap

Seja um vetor de números inteiros S com $n > 0$ elementos. Ainda que o vetor S não seja um max-heap, vamos enxergá-lo do mesmo modo como fizemos anteriormente, como uma árvore binária com o último nível preenchido da esquerda para direita. Nesta seção, queremos resolver o seguinte problema:

> Seja um vetor de números inteiros S com $n > 0$ elementos e um índice i. Se S é visto como uma árvore binária, estabeleça a propriedade max-heap (8.3) para a sub-árvore de S com raiz $S[i]$, supondo que as sub-árvores esquerda e direita do nó i de S são max-heaps.

A função `desce` recebe um conjunto S de números inteiros com $n > 0$ elementos e um índice i. Se enxergamos S como uma árvore binária com seu último nível preenchido da esquerda para direita, então a função `desce` verifica a propriedade max-heap para a sub-árvore de S com raiz $S[i]$, dado que as sub-árvores com raiz $S[\,\texttt{esquerdo(}i\texttt{)}\,]$ e $S[\,\texttt{direito(}i\texttt{)}\,]$ em S são max-heaps. Se a propriedade não é satisfeita para $S[i]$, a função `desce` troca $S[i]$ com o filho que possui maior prioridade. Assim, a propriedade max-heap é estabelecida para o nó de índice i. No entanto, essa troca pode ocasionar a violação da propriedade max-heap para um dos filhos do nó de índice i. Então, esse processo é repetido recursivamente até que a propriedade max-heap não seja mais violada. Dessa forma, a função `desce` rearranja S adequadamente de maneira que a sub-árvore com raiz $S[i]$ torne-se um max-heap, "descendo" o elemento $S[i]$ para a sua posição correta em S.

```
/* Recebe um número inteiro n > 0, um vetor S de números in-
   teiros com n elementos e um índice i e estabelece a pro-
   priedade max-heap para a sub-árvore de S com raiz S[i] */
void desce(int n, int S[MAX], int i)
{
   int e, d, maior;

   e = esquerdo(i);
   d = direito(i);
   if (e < n && S[e] > S[i])
      maior = e;
   else
      maior = i;
   if (d < n && S[d] > S[maior])
      maior = d;
   if (maior != i) {
      troca(&S[i], &S[maior]);
      desce(n, S, maior);
   }
}
```

A figura 8.3 ilustra um exemplo de execução da função **desce**.

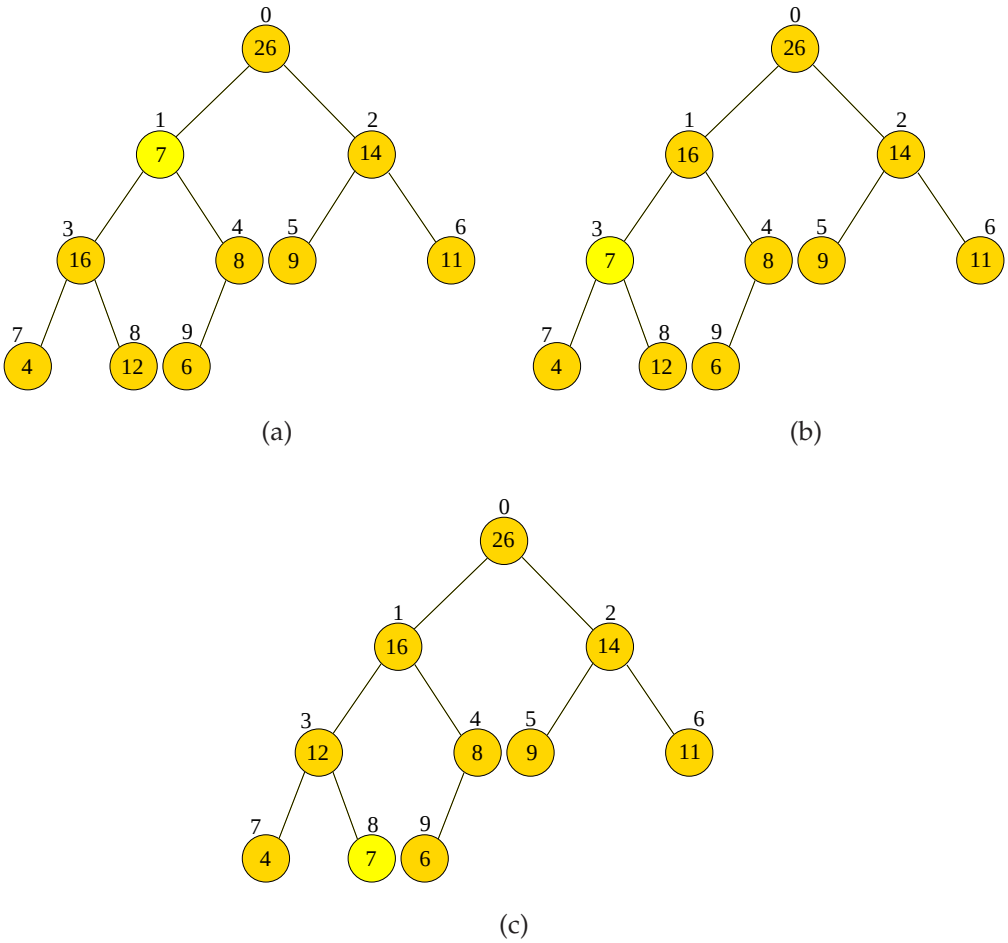

Figura 8.3: Um exemplo de execução da função **desce** com argumentos $(10, S, 1)$.

O tempo de execução de pior caso da função **desce** é proporcional à altura da árvore binária correspondente à S, isto é, proporcional à $\log_2 n$.

8.1.2 Construção de um max-heap

Seja um vetor S de números inteiros com $n > 0$ elementos. Usando a função **desce**, é fácil transformar o vetor S em um max-heap. Se enxergamos S como uma árvore binária, basta então percorrer as sub-árvores de S com alturas crescentes, usando a função **desce** para garantir a propriedade max-heap para cada uma de suas raízes. Podemos excluir as folhas de S, já que toda árvore com altura 0 é um max-heap. A função **constroi_max_heap** é apresentada a seguir.

```
/* Recebe um número inteiro n > 0 e um vetor de números
   inteiros S com n elementos e rearranja o vetor S de
   modo que o novo vetor S possua a propriedade max-heap */
void constroi_max_heap(int n, int S[MAX])
{
   int i;

   for (i = n/2 - 1; i >= 0; i--)
      desce(n, S, i);

}
```

Para mostrar que a função **constroi_max_heap** está correta, temos de usar o seguinte invariante:

> No início de cada iteração da estrutura de repetição da função, cada nó $i+1, i+2, \ldots, n-1$ é raiz de um max-heap.

Dado o invariante acima, é fácil mostrar que a função **constroi_max_heap** está correta, usando indução matemática.

O tempo de execução da função **constroi_max_heap** é calculado observando que cada chamada da função **desce** na estrutura de repetição da função **constroi_max_heap** tem tempo de execução proporcional a $\log_2 n$, onde n é o número de elementos no vetor S. Devemos notar ainda que um número proporcional a n chamadas à função **desce** são realizadas nessa estrutura de repetição. Assim, o tempo de execução da função **constroi_max_heap** é proporcional a $n \log_2 n$. Apesar de correto, esse limitante superior para o tempo de execução da função **constroi_max_heap** não é muito justo. Isso significa que é correto afirmar que a função $T(n)$ que descreve o tempo da função é limitada superiormente pela função $cn \log_2 n$ para todo $n \geqslant n_0$, onde c e n_0 são constantes positivas. Porém, existe outra função, que também limita superiormente a função $T(n)$, que é "menor" que $n \log_2 n$. Podemos mostrar que um limitante superior melhor pode ser obtido observando a variação nas alturas dos nós da árvore max-heap nas chamadas da função **desce**. Essa análise mais apurada fornece um tempo de execução de pior caso proporcional a n para a função **constroi_max_heap**. Mais detalhes sobre essa análise podem ser encontrados no livro de Cormen et. al [1].

A figura 8.4 ilustra um exemplo de execução da função **constroi_max_heap**.

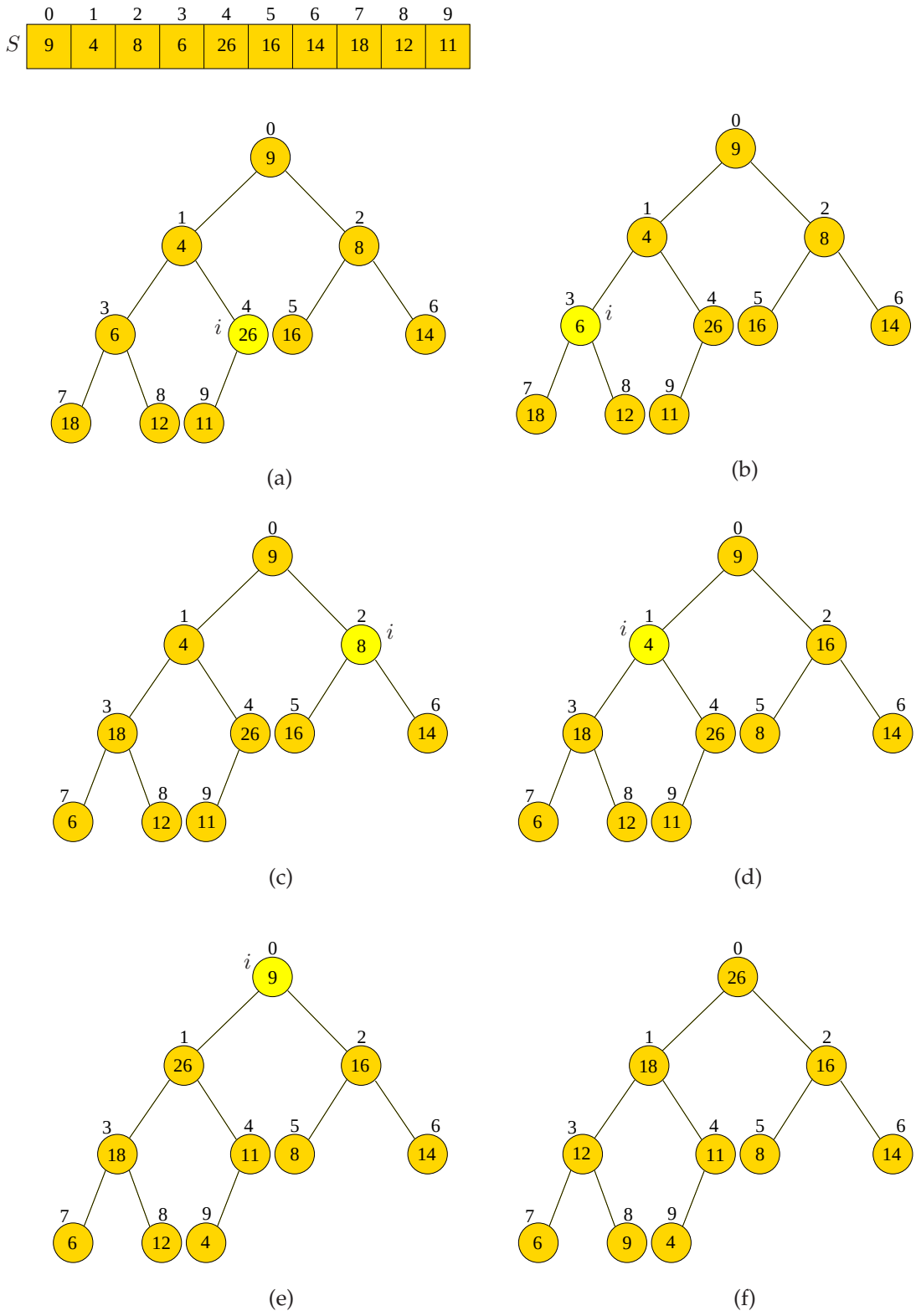

Figura 8.4: Exemplo de execução da função `constroi_max_heap` tendo como entrada o vetor $S = \{9, 4, 8, 6, 26, 16, 14, 18, 12, 11\}$.

Funções muito semelhantes às funções `desce` e `constroi_max_heap` podem ser construídas para obtermos um min-heap. Veja o exercício 8.7.

8.1.3 Alteração de uma prioridade em um max-heap

Vamos retomar o vetor S de números inteiros com $n > 0$ elementos. Suponha que S seja um max-heap. Suponha, no entanto, que a prioridade $S[i]$ seja modificada. Uma operação como esta pode resultar em um novo vetor que deixa de satisfazer a propriedade max-heap justamente na posição i de S. A figura 8.5 mostra os três casos possíveis quando uma alteração de prioridade é realizada em um max-heap S.

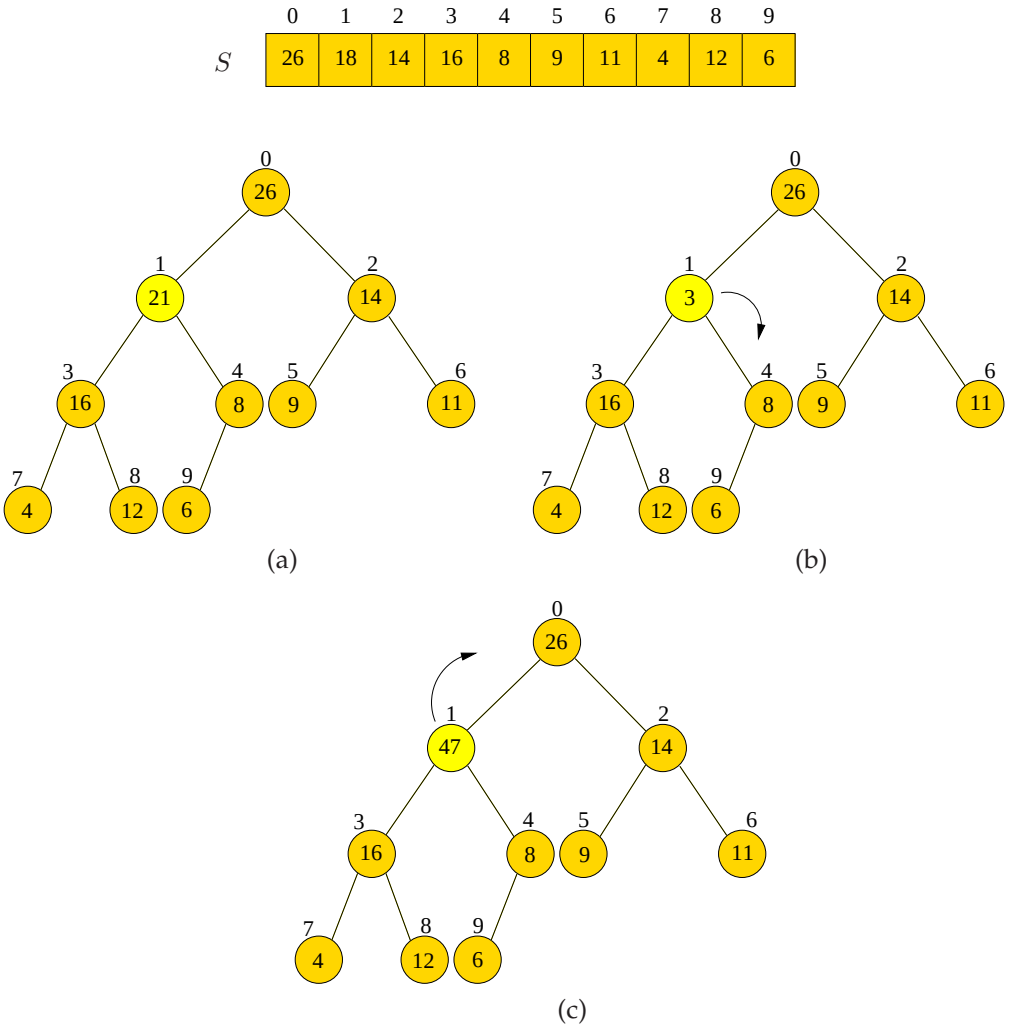

Figura 8.5: Alteração na prioridade $S[1]$ de um max-heap. (a) Alteração mantém a propriedade max-heap. (b) Alteração viola a propriedade max-heap e a prioridade deve "descer". (c) Alteração viola a propriedade max-heap e a prioridade deve "subir".

Como podemos perceber na figura 8.5(a), se a alteração mantém a propriedade max-heap, então não é necessário fazer nada. Observe também que se a prioridade alterada tem valor menor que de pelo menos um de seus filhos, então essa nova prioridade deve "descer" na árvore. Esse caso é mostrado na figura 8.5(b) e é resolvido facilmente usando a função `desce`

descrita na seção 8.1.1. Por fim, se alteração em um nó resulta em uma prioridade maior que a prioridade do seu nó pai, então essa nova prioridade deve "subir" na árvore, como mostra a figura 8.5(c). Nesse caso, precisamos de uma função eficiente que resolva esse problema. A função **sobe** apresentada a seguir soluciona esse problema.

```
/* Recebe um número inteiro n > 0, um vetor S de nú-
   meros inteiros com n elementos e um índice i e es-
   tabelece a propriedade max-heap para a árvore S */
void sobe(int n, int S[MAX], int i)
{

   while (S[pai(i)] < S[i]) {
      troca(&S[i], &S[pai(i)]);
      i = pai(i);
   }

}
```

A figura 8.6 ilustra um exemplo de execução da função **sobe**.

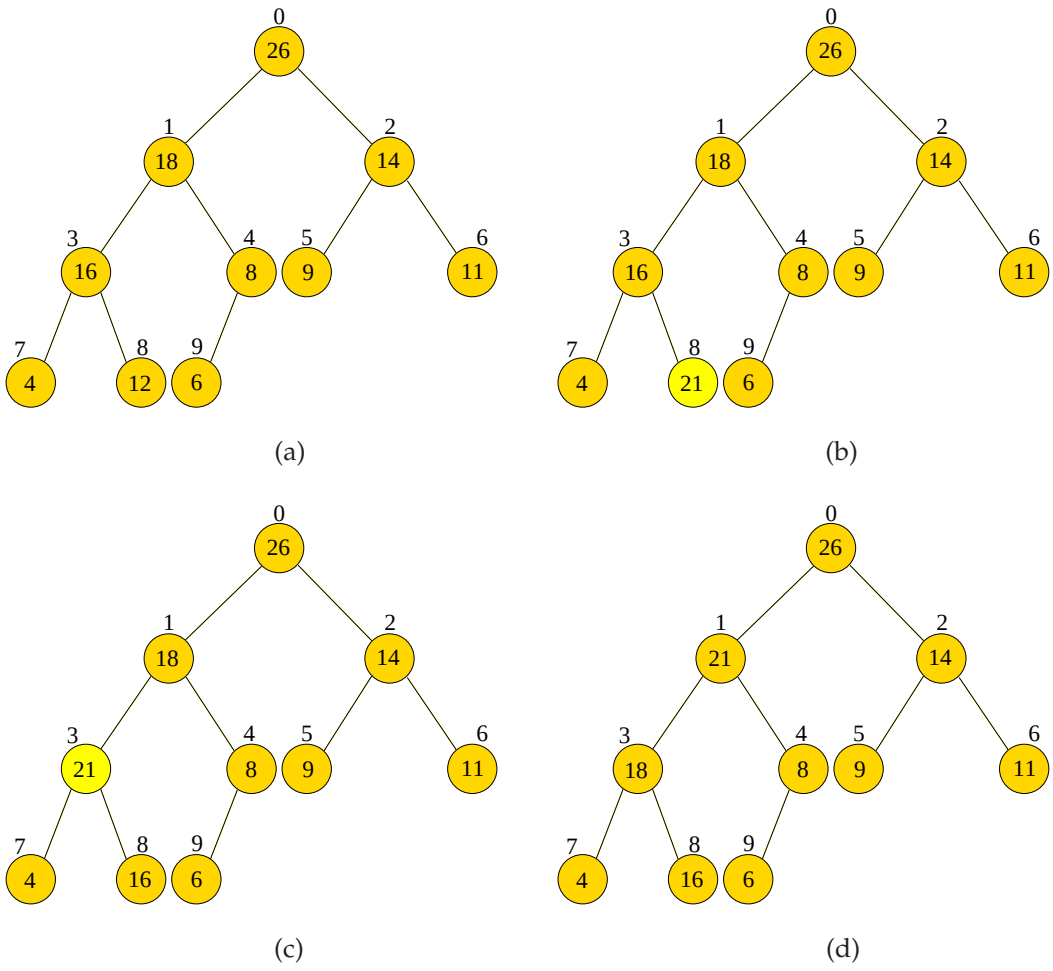

Figura 8.6: Um exemplo de execução da função **sobe** com argumentos $(10, S, 8)$.

O tempo de execução de pior caso da função `sobe` é proporcional à altura da árvore binária correspondente à S, isto é, proporcional à $\log_2 n$.

8.2 Listas de prioridades

Uma lista de prioridades é uma estrutura de dados que ocorre muito freqüentemente em diversas aplicações. Por exemplo, no escalonamento de tarefas em um computador ou em um simulador de eventos, as listas de prioridades são empregadas com muito sucesso. No primeiro exemplo, devemos manter uma lista dos processos a serem executados pelo computador com suas respectivas prioridades, selecionando sempre aquele de mais alta prioridade assim que um processador se torna ocioso ou disponível. No segundo exemplo, uma lista de eventos a serem simulados são colocados em uma lista de prioridades, onde seus tempos de ocorrência são as prioridades na lista, e os eventos devem ser simulados na ordem de seus tempos de ocorrência.

Do mesmo modo como com os heaps, existem dois tipos de listas de prioridades: listas de max-prioridades e listas de min-prioridades. Os exemplos que acabamos de descrever são associados a uma lista de max-prioridades e uma lista de min-prioridades, respectivamente. Assim como antes, focaremos atenção nas listas de max-prioridades, deixando as outras para os exercícios.

Uma **lista de max-prioridades** é uma estrutura de dados para manutenção de um conjunto de elementos S, onde a cada elemento está associada uma prioridade. As seguintes operações são associadas a uma lista de max-prioridades:

(1) inserção de um elemento no conjunto S;

(2) consulta da maior prioridade em S;

(3) remoção do elemento de maior prioridade em S;

(4) aumento da prioridade de um elemento de S.

É fácil ver que um max-heap pode ser usado para implementar uma lista de max-prioridades. A função `consulta_maxima` implementa a operação (2) em tempo de execução constante, isto é, $O(1)$.

```
/* Recebe uma lista de max-prioridades S e devolve a maior prioridade em S */
int consulta_maxima(int S[MAX])
{
   return S[0];
}
```

A função `extrai_maxima` implementa a operação (3) usando a função `desce`, devolvendo também o valor de maior prioridade na lista de max-prioridades. Se a lista é vazia, a função devolve um valor especial para indicar que a remoção não ocorreu.

```
/* Recebe um número inteiro n > 0 e uma lista de max-priorida-
   des S e remove e devolve o valor da maior prioridade em S */
int extrai_maxima(int *n, int S[MAX])
{
   int maior;

   if (*n > 0) {
      maior = S[0];
      S[0] = S[*n - 1];
      *n = *n - 1;
      desce(*n, S, 0);
      return maior;
   }
   else
      return -∞;
}
```

O tempo de execução da função **extrai_maxima** é na verdade o tempo gasto pela função **desce**. Portanto, seu tempo de execução é proporcional a $\log_2 n$.

A função **aumenta_prioridade** implementa a operação (4), recebendo uma lista de max-prioridades, uma nova prioridade e um índice e devolve a lista de max-prioridades com a prioridade alterada.

```
/* Recebe um número inteiro n > 0, uma lista de max-priorida-
   des S, um índice i e uma prioridade p e devolve a lista de
   max-prioridades com a prioridade na posição i modificada */
void aumenta_prioridade(int n, int S[MAX], int i, int p)
{
   if (p < S[i])
      printf("ERRO: nova prioridade é menor que da célula\n");
   else {
      S[i] = p;
      sobe(n, S, i);
   }
}
```

O tempo de execução da função **aumenta_prioridade** é o tempo gasto pela chamada à função **sobe** e, portanto, é proporcional a $\log_2 n$.

Por fim, a operação (1) é implementada pela função **insere_lista** que recebe uma lista de max-prioridades e uma nova prioridade e insere essa prioridade na lista de max-prioridades.

```
/* Recebe um número inteiro n > 0, uma lista de max-prioridades S e uma prio-
   ridade p e devolve a lista de max-prioridades com a nova prioridade */
void insere_lista(int *n, int S[MAX], int p)
{
   S[*n] = p;
   *n = *n + 1;
   sobe(*n, S, *n - 1);
}
```

O tempo de execução da função `insere_lista` é o tempo gasto pela chamada à função `sobe` e, portanto, é proporcional a $\log_2 n$.

8.3 Ordenação usando um max-heap

Um max-heap pode ser naturalmente usado para descrever um algoritmo de ordenação eficiente. Esse algoritmo é conhecido como *heapsort* e tem o mesmo tempo de execução de pior caso da ordenação por intercalação e do caso médio da ordenação por separação.

```
/* Recebe um número inteiro n > 0 e um vetor S de números inteiros com
   n elementos e rearranja S em ordem crescente usando um max-heap */
void heapsort(int n, int S[MAX])
{
   int i;

   constroi_max_heap(n, S);
   for (i = n - 1; i > 0; i--) {
      troca(&S[0], &S[i]);
      n--;
      desce(n, S, 0);
   }
}
```

Podemos mostrar que a função `heapsort` está correta usando o seguinte invariante do processo iterativo:

> No início de cada iteração da estrutura de repetição da função `heapsort`, o vetor $S[0..i]$ é um max-heap contendo os i menores elementos de $S[0..n-1]$ e o vetor $S[i+1..n-1]$ contém os $n-i$ maiores elementos de $S[0..n-1]$ em ordem crescente.

O tempo de execução da função `heapsort` é proporcional a $n \log_2 n$. Note que a chamada da função `constroi_max_heap` gasta tempo proporcional a n e cada uma das $n-1$ chamadas à função `desce` gasta tempo proporcional a $\log_2 n$.

Exercícios

Os exercícios foram extraídos do livro de Cormen et. al [1].

8.1 A seqüência $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12\rangle$ é um max-heap?

8.2 Qual são os números mínimo e máximo de elementos em um max-heap de altura h?

8.3 Mostre que em qualquer sub-árvore de um max-heap, a raiz da sub-árvore contém a maior prioridade de todas as que ocorrem naquela sub-árvore.

8.4 Em um max-heap, onde pode estar armazenado o elemento de menor prioridade, considerando que todos os elementos são distintos?

8.5 Um vetor em ordem crescente é um min-heap?

8.6 Use como base a figura 8.3 e ilustre a execução da função `desce(14, S, 2)` sobre o vetor $S = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

8.7 Suponha que você deseja manter um min-heap. Escreva uma função equivalente à função `desce` para um max-heap, que mantém a propriedade min-heap (8.2).

8.8 Qual o efeito de chamar `desce(`n`, `S`, `i`)` quando a prioridade $S[i]$ é maior que as prioridades de seus filhos?

8.9 Qual o efeito de chamar `desce(`n`, `S`, `i`)` para $i \geqslant n/2$?

8.10 O código da função `desce` é muito eficiente em termos de fatores constantes, exceto possivelmente pela chamada recursiva que pode fazer com que alguns compiladores produzam um código ineficiente. Escreva uma função não-recursiva eficiente equivalente à função `desce`.

8.11 Use como base a figura 8.4 e ilustre a operação da função `constroi_max_heap` sobre o vetor $S = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

8.12 Por que fazemos com que a estrutura de repetição da função `constroi_max_heap` controlada por i seja decrescente de $n/2 - 1$ até 0 ao invés de crescente de 0 até $n/2 - 1$?

8.13 Use como exemplo a figura 8.3 e ilustre a operação da função `extrai_maximo` sobre o vetor $S = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

8.14 Ilustre a operação da função `insere_lista(12, `S`, 9)` sobre a lista de prioridades $S = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

8.15 Escreva códigos eficientes e corretos para as funções que implementam as operações `consulta_minimo`, `extrai_minimo`, `diminui_prioridade` e `insere_lista_min`. Essas funções devem implementar uma lista de min-prioridades com um min-heap.

8.16 A operação `remove_lista(&`n`, `S`, `i`)` remove a prioridade do nó i de uma lista de max-prioridades. Escreva uma função eficiente para `remove_lista`.

8.17 Ilustre a execução da função `heapsort` sobre o vetor $S = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.