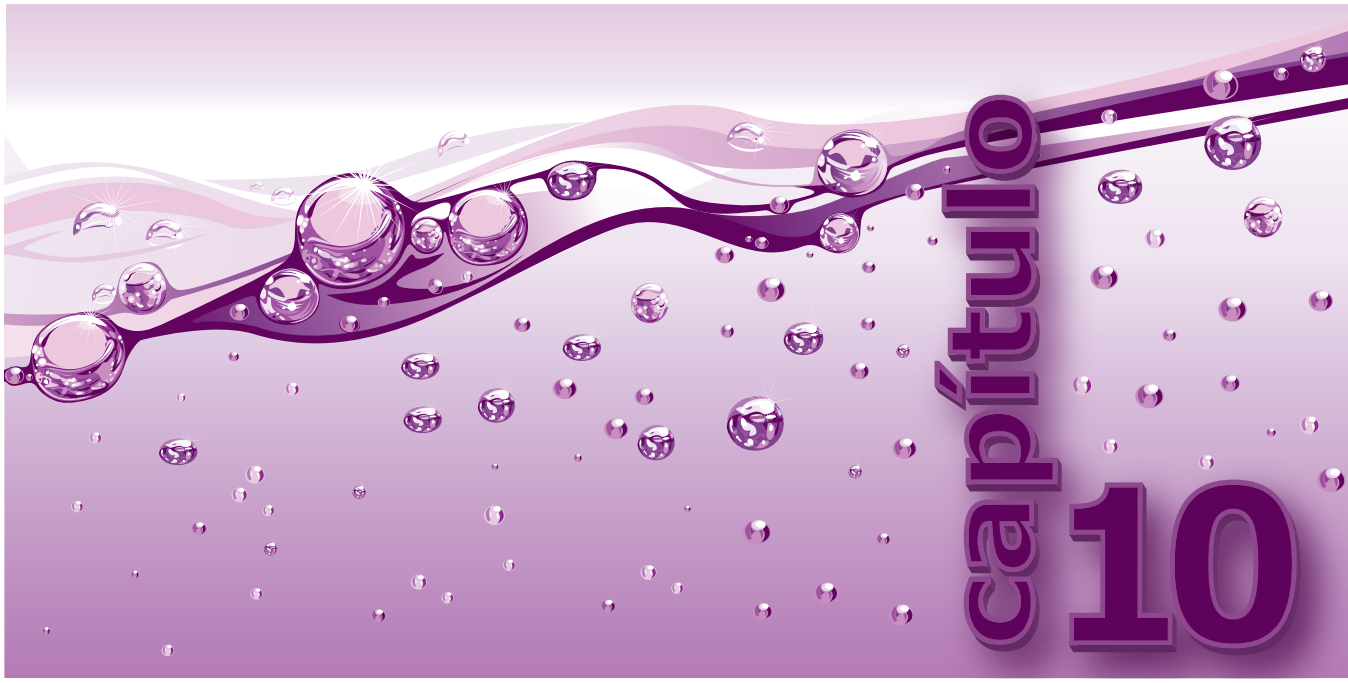


Filas de Prioridade e Heaps

objetivos

Após estudar este capítulo, você deverá ser capaz de:

- Descrever os seguintes conceitos:
  - ❒ Fila de prioridade
  - ❒ Heap binário
  - ❒ Árvore binária completa
  - ❒ Heaps de máximo e de mínimo
  - ❒ Heaps ascendente e descendente
  - ❒ Evento discreto e evento contínuo
  - ❒ Simulação de eventos
  - ❒ Codificação de Huffman
- Comparar os diversos modos de implementação de filas de prioridade explicitando vantagens e desvantagens de cada uma delas em termos de notação θ
- Explicar o esquema de numeração de nós para árvores binárias completas
- Demonstrar como se encontram o pai, o filho esquerdo e o filho direito de um nó de uma árvore binária completa implementada usando array
- Descrever as propriedades estrutural e de ordenação de um heap
- Recuperar a ordenação de um heap após uma remoção ou inserção
- Explicar como funcionam os algoritmos de percolação ascendente e descendente
- Justificar o uso de heap na implementação de fila de prioridade
- Implementar um heap binário usando array dinâmico
- Descrever, em linhas gerais, a codificação de Huffman
- Justificar o uso de fila de prioridade em simulação de eventos discretos

ESTE CAPÍTULO DESCREVE o conceito de lista de prioridade e apresenta diversas maneiras de implementação dessa estrutura de dados. Uma delas, o heap binário, merece destaque especial.

Assim como arrays, heaps são usados apenas como estruturas básicas na implementação de estruturas de dados de maior nível de abstração. A principal aplicação para heaps é na implementação de filas de prioridade, mas eles também são usados na implementação de um famoso algoritmo de ordenação.

Antes de prosseguir, é importante notar que a estrutura de dados heap não deve ser confundida com a área de memória disponível para dados alocados dinamicamente, que também é denominada *heap* (v. **Capítulo 4** do **Volume 1**).

10.1 Filas de Prioridade

10.1.1 Conceitos

Uma **fila de prioridade** é uma estrutura de dados que permite acesso apenas ao elemento considerado de maior prioridade. Aqui, *maior prioridade* pode significar coisas diferentes, dependendo do uso de uma fila de prioridade. Por exemplo, num sistema de atendimento telefônico, chamadas são respondidas na ordem em que elas são recebidas; i.e., a chamada de maior prioridade é aquela que esteja esperando há mais tempo. Assim uma fila comum (v. **Capítulo 8** do **Volume 1**) pode ser considerada uma fila de prioridade cujo elemento de maior prioridade é aquele elemento que tenha sido enfileirado há mais tempo.

Responsáveis por salas de emergência de hospitais ordenam pacientes numa fila de prioridade de acordo com as gravidades de seus estados de saúde, de maneira que o paciente com o mal mais grave é atendido primeiro.

Uma fila de prioridade é uma estrutura de dados que permite pelo menos as seguintes operações:

- **Inserção** (ou **enfileiramento**), que insere um elemento de acordo com sua prioridade.
- **Remoção** (ou **desenfileiramento**), que remove o elemento de menor valor ou o elemento de maior valor, dependendo do fato de se estar lidando, respectivamente, com uma fila de prioridade **ascendente** ou **descendente**.

Uma fila de prioridade ordinária não pode ser simultaneamente ascendente e descendente. Portanto uma fila de prioridade implementa exclusivamente remoção do elemento de menor valor ou a remoção do elemento de maior valor, mas nunca as duas remoções. O fato de uma fila de prioridade ser ascendente ou descendente deve ser decidido de antemão.

10.1.2 Aplicações

Algumas aplicações de filas de prioridade são enumeradas a seguir:

- **Gerenciamento de filas de impressão**. Alguns trabalhos de impressão podem ter mais importância ou serem mais curtos e, assim, terem maior precedência sobre os demais.
- **Controle de tempo de uso de CPU**. Um sistema operacional multiusuário pode usar filas de prioridade para armazenar solicitações do usuário na ordem em que elas são feitas. Mas tais solicitações também podem ser tratadas de acordo com a importância do serviço requisitado ou do usuário que o requisitou. Por exemplo, uma solicitação do chefe de uma empresa pode ter maior prioridade do que uma solicitação de um subalterno. Além disso, um programa interativo pode ter maior prioridade do que um serviço de impressão de um relatório que não é urgente.
- **Ordenação de dados**. Dado um conjunto de itens para ordenar, eles podem ser enfileirados numa fila de prioridade e depois desenfileirados em ordem do maior para o menor ou vice-versa (v. **Capítulo 11**).

10.1.3 Implementações

Existem diversas maneiras de se implementar uma fila de prioridade. Em qualquer implementação, o objetivo é acessar eficientemente o item de maior prioridade que se encontra na lista. As abordagens mais comuns são:

- **Lista encadeada sem ordenação**. Nesse caso, a inserção é feita no início da lista com custo temporal $\theta(1)$ e a remoção pode ocorrer em qualquer nó da lista com custo temporal $\theta(n)$. Ou seja, remoção requer uma busca sequencial na lista para encontrar o item com a maior prioridade.
- **Lista encadeada ordenada**. Supondo que uma lista encadeada seja mantida ordenada em ordem decrescente de prioridade, inserir um item requer que a posição de inserção seja encontrada para manter a lista ordenada a cada inserção e, portanto, o custo temporal dessa operação é $\theta(n)$. A remoção é sempre feita no início da lista com custo temporal $\theta(1)$.
- **Lista indexada sem ordenação**. Nesse caso, a inserção é feita ao final da lista com custo temporal $\theta(1)$ e a remoção pode ocorrer em qualquer posição da lista com custo temporal $\theta(n)$ (v. **Capítulo 3**).
- **Lista indexada ordenada**. Remoção é uma operação com custo temporal $\theta(1)$ se a lista estiver em ordem crescente de prioridade, de modo que o elemento removido é sempre o último elemento da lista. Inserção requer que se encontre o local de inserção do item com custo $\theta(\log n)$, se for usada busca binária. Mas, por outro lado, inserção também requer rearranjo dos elementos da lista, que é uma operação com custo temporal $\theta(n)$. Logo a combinação das duas operações tem custo $\theta(n)$.
- **Árvore binária de busca balanceada**. Nessa abordagem, inserção e remoção têm o mesmo custo temporal $\theta(\log n)$. Por outro lado, essa é a opção com mais alto custo de implementação (lembra de árvores AVL?).
- **Heap binário** (ou apenas **heap**). Essa abordagem oferece várias vantagens, tais como simplicidade, rapidez e uso de pouco espaço de armazenamento. Além do mais, essa abordagem é muito fácil de implementar.

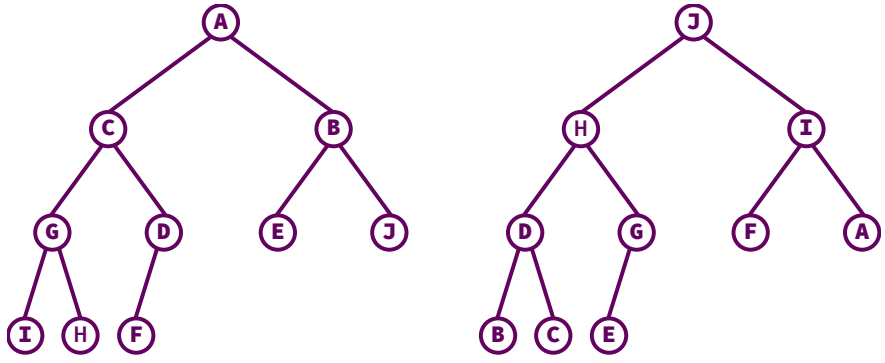
Este capítulo irá explorar essa última abordagem de implementação. Espera-se que, a esta altura, o leitor já tenha adquirido cabedal para implementar as demais abordagens.

10.2 Heaps Binários

10.2.1 Conceitos

Um **heap binário** (doravante referido apenas como **heap**) é uma árvore binária com duas propriedades: (1) **estrutural**, referente ao seu formato, e de **ordenação**, referente à ordem de seus elementos. De acordo com a propriedade estrutural, um heap deve ser uma **árvore binária completa**. Por sua vez, a propriedade de ordenação depende do fato de se estar lidando com um **heap de máximo** (também conhecido como **heap descendente**) ou um **heap de mínimo** (também conhecido como **heap ascendente**). Num heap de máximo, o valor armazenado em cada nó é maior do que ou igual ao valor armazenado em cada um de seus filhos, enquanto num heap de mínimo, o valor armazenado em cada nó é menor do que ou igual ao valor armazenado em cada um de seus filhos. As denominações *heap ascendente* e *heap descendente* são derivadas do fato de se os elementos do heap forem removidos consecutivamente e inseridos numa lista, essa lista será ordenada em ordem ascendente ou descendente, respectivamente.

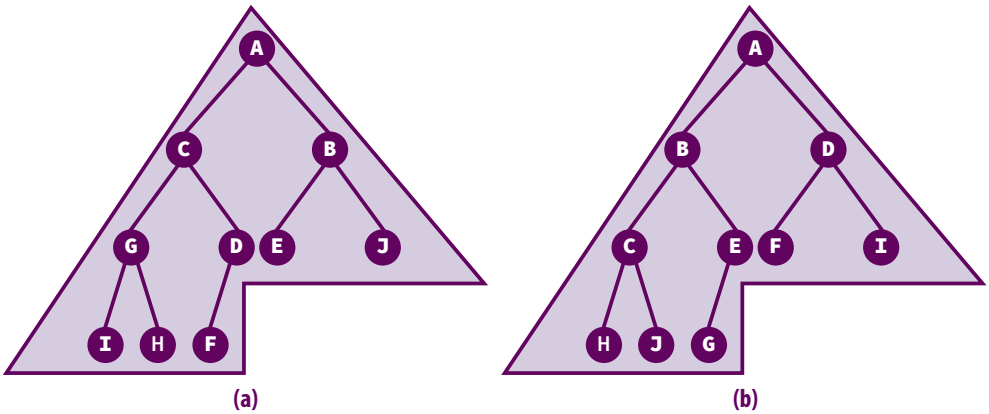
A **Figura 10–1 (a)** mostra um heap de mínimo ao passo que a **Figura 10–1 (b)** mostra um heap de máximo (considerando-se a ordem alfabética usual). Como mostram essas figuras, a raiz de um heap de mínimo contém o menor valor do heap e a raiz de um heap de máximo contém o maior valor do heap.



(a) Heap de Mínimo (b) Heap de Máximo

Figura 10–1: Heap de Mínimo e Heap de Máximo

A **Figura 10–2** mostra dois heaps de mínimo contendo as letras de *A* a *J*, considerando a ordem alfabética usual. Note que os posicionamentos das letras diferem nas duas árvores, mas o formato permanece o mesmo. Note que ambos os heaps têm a mesma raiz, que contém a menor letra. Porém a maior letra (i.e., *J*) ocupa níveis diferentes nos dois heaps. Isso indica que um conjunto de valores pode ser armazenado numa árvore binária de diversas maneiras e ainda satisfazer a propriedade de ordenação de heaps. De acordo com a propriedade estrutural de heaps, o formato de qualquer heap com um determinado número de elementos é o mesmo. Por exemplo, qualquer heap com dez elementos tem o formato exibido na **Figura 10–2**.



(a) (b)

Figura 10–2: Dois Heaps Diferentes Contendo os Mesmos Elementos

Como a propriedade estrutural de heaps afirma que todo heap é uma árvore binária completa, pode-se facilmente armazenar a árvore num array, como foi discutido no **Capítulo 12** do **Volume 1**. Como foi visto naquele capítulo, uma árvore binária completa pode ser representada facilmente usando arrays, de modo que ponteiros não são necessários para implementar um heap. A desvantagem dessa abordagem é que o tamanho (i.e., número de elementos) máximo do heap deve ser estimado a priori. As fórmulas para cálculo das posições de pai e filhos que foram apresentadas no **Teorema 12.7** do referido capítulo assumem que a numeração de nós começa com 1. Quando a numeração começa com zero, como será o caso aqui, novas fórmulas são necessárias, como mostra o **Teorema 10.1**, cuja prova é semelhante à prova do **Teorema 12.7** apresentada no **Apêndice B** do **Volume 1**.

Teorema 10.1: Se uma árvore binária completa com n nós for representada sequencialmente conforme foi descrito acima, então, para qualquer nó numerado por i, $0 \leq i < n - 1$, tem-se:

(i) *Pai(i)* é numerado como $\lfloor (i - 1)/2 \rfloor$, se $i \neq 0$. Se $i = 0$, o nó i é a raiz da árvore, que não possui pai.

(ii) *FilhoEsquerda(i)* é numerado como $2i + 1$, se $2i + 1 < n$. Se $2i + 1 \geq n$, então o nó i não possui filho esquerdo.

(iii) *FilhoDireita(i)* é numerado como $2i + 2$, se $2i + 2 < n$. Se $2i + 2 \geq n$, então o nó i não possui filho direito.

Prova: A prova desse teorema encontra-se no **Apêndice B** do **Volume 1**.

A **Figura 10–3** apresenta um exemplo de heap e o respectivo array que o representa.

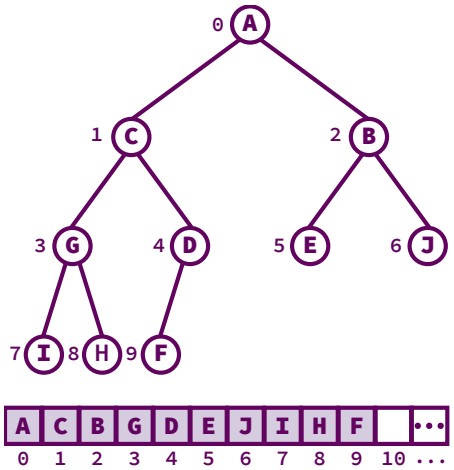


Figura 10–3: Árvore Binária Completa e Respectivo Array Associado

Operações de inserção e remoção tendem a destruir a propriedade de ordenação de um heap, que deve ser restaurado antes que uma tal operação seja concluída, como será visto adiante.

10.2.2 Operações Básicas

Inserção

Para inserir um elemento num heap ascendente, considera-se, em princípio, seu armazenamento na próxima posição disponível no array que representa o heap. No exemplo apresentado na **Figura 10–4**, o índice dessa posição seria *10*. Como uma inserção pode destruir a ordem do heap, é necessário verificar se isso ocorre e, se for o caso, reordenar o heap.

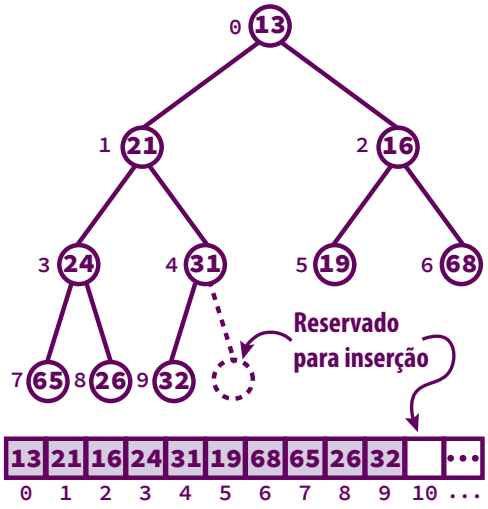


Figura 10–4: Heap Binário Ascendente e seu Array Associado

O algoritmo completo de inserção num heap ascendente é apresentado na **Figura 10–5**.

ALGORITMO INSEREEMHEAPASCENDENTE

ENTRADA: O conteúdo de um novo nó

ENTRADA/SAÍDA: Um heap ascendente

1. Acrescente o novo elemento ao final do array que armazena o heap
2. Enquanto o novo elemento não estiver na raiz do heap e seu valor for menor do que o valor do seu pai, troque-o de posição com seu pai

FIGURA 10–5: ALGORITMO DE INSERÇÃO EM HEAP ASCENDENTE

O algoritmo de reajuste de ordenação de um heap é denominado **percolação ascendente**, pois (imagina-se que) o elemento a ser inserido é *filtrado* (i.e., *percolado*) em direção à raiz da árvore. Se o heap sob consideração for descendente, deve-se trocar no algoritmo acima a palavra *maior* por *menor* e vice-versa (mas o processo de inserção continua sendo de percolação ascendente).

Considere como exemplo a inserção do elemento cujo valor é *14* no heap ascendente da **Figura 10–4**. Como se vê nessa figura, a primeira posição reservada para inserção é a próxima posição disponível no array, que, no exemplo acima, é a posição de índice *10*. O valor do novo elemento (i.e., *14*) é menor do que o valor do nó que seria seu pai (i.e., *31*). Portanto desloca-se esse pai para a posição antes reservada para inserção e o espaço ora ocupado por esse pai passa a ser o novo nó reservado para inserção, conforme ilustrado na **Figura 10–6 (a)**. Agora o valor do nó a ser inserido (i.e., *14*) ainda é menor do que o valor do seu suposto pai, cujo valor é *21* [v. **Figura 10–6 (b)**].

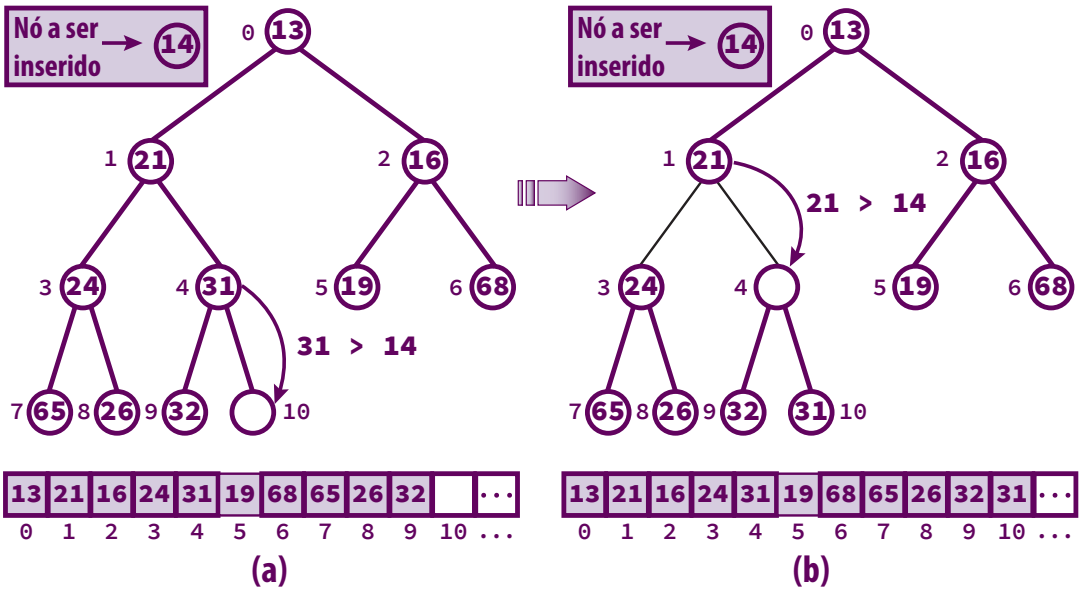


FIGURA 10–6: INSERÇÃO DE NÓ EM HEAP ASCENDENTE 1

Prosseguindo com o último exemplo, o nó cujo valor é *21* ocupa o espaço ora reservado para inserção e depois torna-se o novo espaço reservado para inserção, como mostra a **Figura 10–7 (a)**. Finalmente, verifica-se que o valor do novo elemento é maior do que seu pai (a raiz) nessa nova posição e o novo elemento é acomodado nessa posição, conforme mostrado na **Figura 10–7 (b)**.

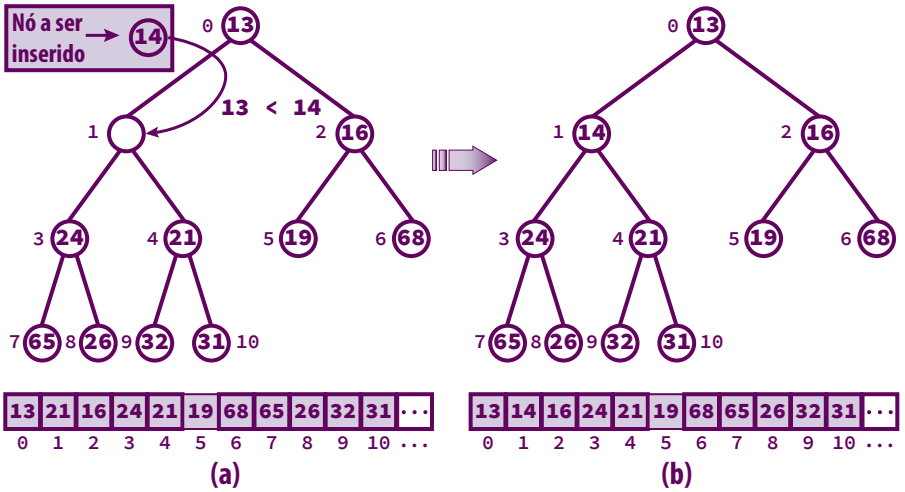


FIGURA 10–7: INSERÇÃO DE NÓ EM HEAP ASCENDENTE 2

A **Figura 10–8** mostra um exemplo de inserção de nó num heap descendente. Espera-se que o leitor seja capaz de descrever em detalhes todos os passos envolvidos nesse processo.

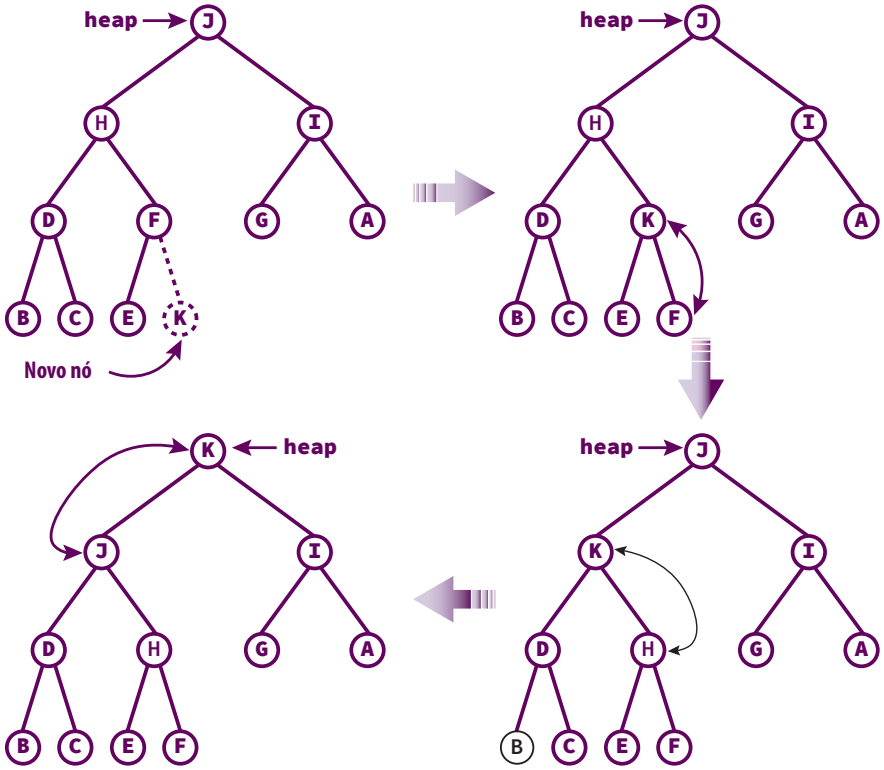


FIGURA 10–8: PERCOLAÇÃO ASCENDENTE EM HEAP DESCENDENTE

Remoção

Encontrar o nó contendo o menor valor de um heap ascendente é fácil, pois ele é exatamente a raiz do heap. Portanto a tarefa reduz-se a reorganizar o heap após a remoção. O algoritmo completo de remoção num heap ascendente é apresentado na **Figura 10–9**. Esse algoritmo de reajuste de ordenação de heap é denominado **percolação descendente**, pois o elemento a ser inserido é filtrado (i.e., percolado) em direção às folhas da árvore.

Se o heap sob consideração for descendente, deve-se trocar no algoritmo da **Figura 10–9** a palavra *maior* por *menor* e vice-versa (mas o processo de remoção continua sendo de percolação descendente).

Algoritmo RemoveEmHeapAscendente

Entrada/Saída: Um heap ascendente

Saída: O conteúdo da raiz do heap

1. Guarde o valor do elemento que se encontra na raiz do heap
2. Remova o último elemento do heap e insira-o na raiz
3. Enquanto a raiz não se tornar uma folha e seu valor não for menor do que o valor de cada um dos seus filhos, troque-a de posição com o filho que possui o menor valor

Figura 10–9: Algoritmo de Remoção em Heap Ascendente

A **Figura 10–10 (a)** apresenta o passo inicial da remoção do menor elemento de um heap ascendente. Nela, a raiz torna-se disponível para inserção, enquanto a última folha do heap (i.e., aquela cujo valor é *32*) é removida e passa a ser tratada como uma nova chave a ser inserida a partir da posição selecionada para inserção. Por sua vez, a **Figura 10–10 (b)** mostra que o elemento com valor *32* não pode ser inserido na posição ora disponível para inserção, pois, nesse caso, os valores de seus filhos seriam menores do que seu valor. Então o nó com valor igual a *14* é selecionado para ocupar o lugar da raiz, visto que ele é o nó contendo o menor valor dentre os referidos filhos.

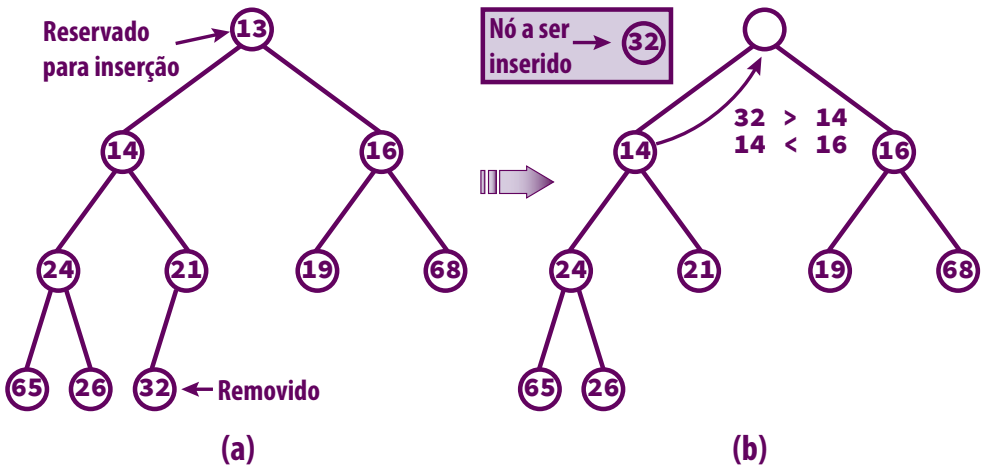


Figura 10–10: Remoção de Elemento com Reordenação de Heap Ascendente 1

Continuando com o último exemplo, a **Figura 10–11 (a)** mostra que o nó contendo *14* ocupou o lugar da raiz e o espaço que ele ocupava antes tornou-se disponível para inserção. Ocorre, porém, que o valor do nó a ser inserido nessa posição é maior do que os valores dos nós que seriam seus filhos, o que inviabiliza essa inserção. Logo o nó contendo *21* passa a ocupar o espaço agora disponível, pois ele é o nó de menor conteúdo entre os filhos em questão, como mostra a **Figura 10–11 (b)**. Essa última figura também mostra que o espaço vazio deixado com a inserção do nó com valor *21* no nível superior do heap pode agora ser ocupado pelo nó que se pretende inserir, pois esse espaço refere-se a uma folha.

Criação

A criação de um heap a partir de uma coleção de elementos é uma operação comum que, evidentemente, pode ser realizada inserindo os vários elementos conforme descrito acima. Entretanto é mais eficiente considerar

o array de elementos como sendo um heap desordenado e aplicar uma operação de reordenação de heap (i.e., percolação para baixo) a partir do pai do último elemento do heap, como será detalhado na **Seção 11.3.3**.

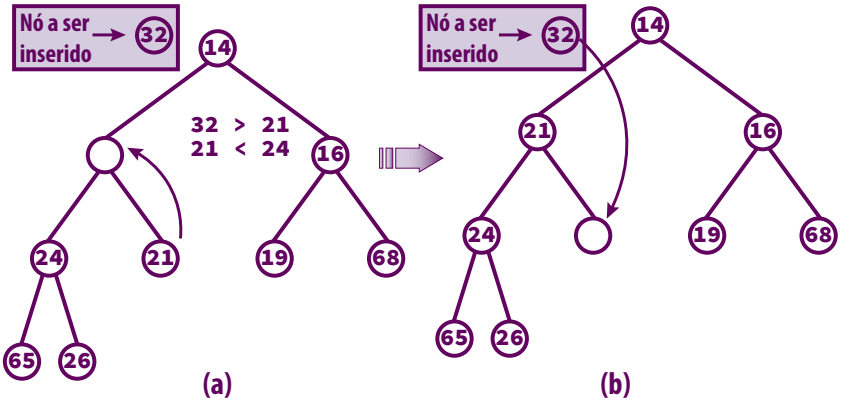


Figura 10–11: Remoção de Elemento com Reordenação de Heap Ascendente 2

10.2.3 Outras Operações

Consulta

Consultar o elemento com maior prioridade é uma operação útil em filas de prioridade e pode ser implementada de modo trivial e com custo constante quando uma fila de prioridade é implementada usando heap, pois esse elemento se encontra na raiz do heap e nenhum reajuste se faz necessário.

Acréscimo de Prioridade

Uma operação relativamente comum em filas de prioridade é aquela que aumenta a prioridade de um item da fila. Essa operação é útil, por exemplo, para um administrador de sistema que pode aumentar a prioridade de seus próprios programas para que eles sejam executados mais rapidamente.

Quando uma fila de prioridade é implementada por meio de um heap, essa operação pode violar a propriedade de ordenação do heap. Portanto, nesse caso, essa operação deve ser seguida por uma percolação descendente, quando o heap é ascendente, ou por uma percolação ascendente, quando o heap é descendente.

Decréscimo de Prioridade

Outra operação útil em filas de prioridade é reduzir a prioridade de um item da fila. Essa operação pode ser usada por um sistema operacional para reduzir a prioridade de um processo que esteja consumindo muito tempo de CPU.

Essa operação pode violar a propriedade de ordenação de um heap usado para implementar uma fila de prioridade. Nesse caso, essa operação deve ser seguida por uma percolação ascendente, quando o heap é ascendente, ou por uma percolação descendente, quando o heap é descendente.

Remoção de Elemento Específico

Suponha que um passageiro faça parte de uma lista de espera e que essa lista seja uma fila de prioridade (ordenada de acordo com as idades dos passageiros, por exemplo). Então, se um determinado passageiro que faça parte dessa lista fica cansado de esperar e desiste de fazer parte dela, ele deve ser removido da lista. Portanto a remoção de um elemento específico de uma fila de prioridade é uma operação importante.

Quando a fila de prioridade é implementada usando um heap, tal remoção pode ser efetuada em dois passos:

1. Aumenta-se a prioridade do item que se deseja remover até que ela seja maior do que a prioridade da raiz do heap.
2. Remove-se o elemento conforme foi descrito na **Figura 10–9**, pois, depois do primeiro passo, o elemento se encontra na raiz do heap.

10.2.4 Implementação

Os seguintes tipos serão usados na implementação de heap a ser desenvolvida:

```
typedef int tNoHeap; /* Tipo de nó de um heap */

typedef struct {
           tNoHeap *itens;      /* Array de elementos */
           int      capacidade; /* Quantos elementos o heap pode conter */
           int      nItens;     /* Número de elementos do heap */
        } tHeap;

   /* Tipo de ponteiro para função de comparação que compara elementos do heap */
typedef int (*tFCompara) (const void *, const void *);
```

As definições de macros a seguir não são essenciais, mas facilitam o entendimento e a implementação das operações com heaps:

```
#define FILHO_E(x) (2*(x) + 1)
#define FILHO_D(x) (2*(x) + 2)
#define PAI(x) (((x) - 1)/2)
```

A função `IniciaHeap()` inicia um heap binário e seu único parâmetro é o endereço do heap que será iniciado. Essa função retorna seu único parâmetro.

```
tHeap *IniciaHeap(tHeap *heap)
{
   heap->capacidade = TAMANHO_INICIAL_HEAP;
   heap->nItens = 0;
   heap->itens = calloc(TAMANHO_INICIAL_HEAP, sizeof(tNoHeap));

   ASSEGURA(heap->itens, "Impossivel alocar heap");

   return heap;
}
```

A constante `TAMANHO_INICIAL_HEAP` representa o tamanho inicial do array que armazena os elementos do heap e pode ser definida como:

```
#define TAMANHO_INICIAL_HEAP 10 /* Capacidade inicial do heap */
```

A função `DestroiHeap()` libera o espaço ocupado por um heap e seu único parâmetro é o endereço do heap cujo espaço será liberado.

```
void DestroiHeap(tHeap *heap)
{
      /* Libera o espaço ocupado pelos nós do heap */
   free(heap->itens);
   heap->nItens = 0;
}
```

A função `InsereEmHeap()` insere um novo item num heap e tem como parâmetros:

- `heap` (entrada/saída) — heap no qual será feita a inserção
- `item` (entrada) — o item a ser inserido

- `Compara` (entrada) — ponteiro para uma função que compara dois elementos de array armazenados no heap

```
void InsereEmHeap(tHeap *heap, tNoHeap item, tFCompara Compara)
{
   int i;

      /* Se o array que suporta o heap estiver repleto, redimensiona-o */
   if(HeapCheio(heap))
      RedimensionaHeap(heap);

      /* Acrescenta o novo elemento ao final do array */
      /* e incrementa o número de elementos no heap   */
   heap->itens[heap->nItens] = item;
   heap->nItens++;

      /* Reordena o heap se for necessário */
   for ( i = heap->nItens - 1; i &&
         Compara( &heap->itens[PAI(i)], &heap->itens[i] ) > 0; ) {
      TrocaGenerica(&heap->itens[i], &heap->itens[PAI(i)], sizeof(tNoHeap));
      i = PAI(i);
   }
}
```

A função `InsereEmHeap()` verifica se o array que armazena os elementos do heap está repleto chamando a função `HeapCheio()` e, se esse for o caso, ela chama a função `RedimensionaHeap()` para aumentar a capacidade desse array. Essas duas funções são definidas a seguir.

```
static int HeapCheio(const tHeap *heap)
{
   return heap->nItens == heap->capacidade;
}

static tHeap *RedimensionaHeap(tHeap *heap)
{
   heap->capacidade *= 2; /* Tenta duplicar a capacidade do heap */

      /* Aqui não há problema em atribuir o retorno de realloc() */
      /* ao ponteiro passado como parâmetro, pois se realloc()   */
      /* fracassar, o programa será abortado                     */
   heap->itens = realloc(heap->itens, heap->capacidade*sizeof(tNoHeap));
   ASSEGURA(heap->itens, "Impossivel redimensionar heap");

   return heap;
}
```

A função `RemoveMinHeap()` remove e retorna o menor elemento de um heap binário ascendente e seus parâmetros são:

- `heap` (entrada/saída) — heap no qual será feita a remoção
- `Compara` (entrada) — ponteiro para uma função que compara dois elementos de array armazenados no heap

```
tNoHeap RemoveMinHeap(tHeap *heap, tFCompara Compara)
{
   tNoHeap menorElemento;

   ASSEGURA( !HeapVazio(heap), "Tentativa de remover elemento de heap vazio" );

      /* O menor elemento está sempre na raiz de um heap de mínimo */
   menorElemento = heap->itens[0];
```

```
      /* Coloca o último elemento do heap em sua raiz e reordena o heap */
   heap->itens[0] = heap->itens[--heap->nItens];
   OrdenaHeap(heap, 0, Compara);

   return menorElemento;
}
```

A função `OrdenaHeap()`, chamada por `RemoveMinHeap()`, é usada para restaurar a propriedade de ordenação do heap caso tenha sido desordenado após a operação de remoção. A função `OrdenaHeap()` tem como parâmetros:

- `heap` (entrada e saída) — ponteiro para o heap
- `indice` (entrada) — índice do elemento do heap a partir do qual será efetuada a restauração
- `Compara` (entrada) — ponteiro para uma função que compara dois elementos de array armazenados no heap

```
static void OrdenaHeap(tHeap *heap, int indice, tFCompara Compara)
{
   int iEsq = FILHO_E(indice), /* Índice do filho esquerdo */
       iDir = FILHO_D(indice), /* Índice do filho direito  */
       iMenor = indice; /* Supõe que o menor elemento é o pai */

      /* Compara o filho esquerdo com seu pai */
   if ( iEsq < heap->nItens &&
        Compara(&heap->itens[iEsq], &heap->itens[indice] ) < 0 )
      iMenor = iEsq; /* O filho esquerdo é menor do que o pai */

      /* Compara o filho direito com o menor entre seu pai e seu irmão */
   if ( iDir < heap->nItens &&
        Compara( &heap->itens[iDir], &heap->itens[iMenor] ) < 0 )
      iMenor = iDir; /* O filho direito é o menor de todos */

      /* Se o nó não for menor do que seus filhos, */
      /* troca-o de posição com o menor deles      */
   if (iMenor != indice) {
         /* O pai e o menor filho trocam de posição */
      TrocaGenerica(&heap->itens[indice], &heap->itens[iMenor], sizeof(tNoHeap));

         /* Reordena o heap a partir do filho que se tornou pai */
      OrdenaHeap(heap, iMenor, Compara);
   }
}
```

A função `TrocaGenerica()` chamada por `OrdenaHeap()` é uma função genérica que troca os valores de duas variáveis (`var1` e `var2`) do mesmo tamanho (`tam`) definida como:

```
static void TrocaGenerica(void *var1, void *var2, size_t tam)
{
   void *p;

   p = malloc(tam);
   ASSEGURA(p, "Impossivel trocar duas variaveis");

   memmove(p, var1, tam);
   memmove(var1, var2, tam);
   memmove(var2, p, tam);

   free(p);
}
```

A função `ObtemMinimoHeap()` retorna o menor elemento de um heap de mínimo sem removê-lo e é implementada como:

```
tNoHeap ObtemMinimoHeap(const tHeap *heap)
{
      /* O menor elemento de um heap de mínimo está na raiz */
   return heap->itens[0];
}
```

A função `HeapVazio()` a seguir verifica se um heap está vazio:

```
int HeapVazio(const tHeap *heap)
{
   return !heap->nItens;
}
```

A função `TamanhoHeap()` retorna o número de itens de um heap e sua implementação é exibida abaixo:

```
int TamanhoHeap(const tHeap *heap)
{
   return heap->nItens;
}
```

A função de comparação, cujo endereço deve ser passado para algumas das funções apresentadas acima, pode ser definida como a função `ComparaInts()` a seguir:

```
int ComparaInts(const void *e1, const void *e2)
{
   ASSEGURA(e1 && e2, "Elemento nulo recebido");

   return *(int *)e1 - *(int *)e2;
}
```

A função `ComparaInts()` compara dois elementos de um heap e seus parâmetros são ponteiros para esses elementos. Ela retorna:

- `0`, se os dois elementos forem iguais
- Um valor menor do que zero, se o primeiro elemento for menor do que o segundo
- Um valor maior do que zero, se o primeiro elemento for maior do que o segundo

Evidentemente, se o tipo do elemento armazenado no heap não for **int**, essa função deve ser convenientemente reescrita. Uma vantagem do uso de ponteiros para função na implementação de heaps é que um programa-cliente dessa estrutura de dados pode decidir se o heap desejado é de mínimo ou de máximo de acordo com a função de comparação passada como parâmetro para as funções de implementação de heap que podem alterar a propriedade de ordenação do heap. Por exemplo, usando a função `ComparaInts()` do modo que ela foi definida acima, tem-se um heap de mínimo. Por outro lado, se a instrução de retorno dessa função for trocada por:

```
return *(int *)e2 - *(int *)e1;
```

o resultado obtido será um heap de máximo.

10.2.5 Análise

Teorema 10.2: No pior caso, o custo temporal de uma operação de inserção em heap é $\theta(\log n)$.

Prova: No pior caso, uma operação de inserção requer visita a um nó em cada nível da árvore desde uma folha até a raiz. Como todo heap é uma árvore binária completa e a altura de uma árvore binária completa é dada por $\lfloor \log_2 n + 1 \rfloor$ (v. **Capítulo 12** do **Volume 1**), no pior caso, o custo temporal de uma operação de inserção é $\theta(\log n)$. ■

Teorema 10.3: No pior caso, o custo temporal de uma operação de remoção é $\theta(\log n)$.

Prova: Uma operação de remoção requer, no pior caso, que cada nó desde a raiz até uma folha seja visitado. Portanto, usando um raciocínio similar àquele usado na prova do **Teorema 10.2**, o custo temporal de uma operação de remoção nesse caso também é $\theta(\log n)$. ■

Pode-se facilmente mostrar usando um argumento semelhante aos apresentados nas provas desses teoremas que as operações de acréscimo ou decréscimo de prioridade e de remoção de um elemento específico, descritas na **Seção 10.2.3**, também têm custo temporal $\theta(\log n)$. A operação de consulta descrita nessa seção tem custo $\theta(1)$, pois o elemento consultado sempre se encontra na raiz do heap.

10.3 Análise de Filas de Prioridade

A **Tabela 10–1** resume a eficiência das várias implementações de filas de prioridade com respeito às operações de enfileiramento e desenfileiramento.

Implementação via...	Custo Temporal de...	
	Enfileiramento	Desenfileiramento
Heap	$\theta(\log n)$	$\theta(\log n)$
Lista encadeada sem ordenação	$\theta(1)$	$\theta(n)$
Lista encadeada ordenada	$\theta(n)$	$\theta(1)$
Lista indexada sem ordenação	$\theta(1)$	$\theta(n)$
Lista indexada ordenada	$\theta(n)$	$\theta(1)$
Árvore binária ordinária de busca	$\theta(n)$	$\theta(n)$
Árvore binária de busca balanceada	$\theta(\log n)$	$\theta(\log n)$

Tabela 10–1: Comparação de Implementações de Filas de Prioridade

Como mostra a **Tabela 10–1**, as principais operações sobre listas de prioridade apresentam o mesmo custo temporal quando uma fila de prioridade é implementada usando heap ou usando árvore binária de busca balanceada. Porém a grande vantagem do uso de heaps em implementações de filas de prioridade não aparece na referida tabela, que é a simplicidade.

10.4 Simulação de Eventos Discretos

Um dos usos mais comuns de filas de prioridade é uma forma comum de simulação denominada **simulação discreta dirigida por eventos**. No presente contexto, um **evento** é uma representação de uma ação que deve ocorrer num instante específico. Nesse tipo de simulação, um programa age de acordo com os eventos gerados aleatoriamente que devem ser processados. O papel desempenhado por uma fila de prioridade na simulação dirigida por eventos é armazenar esses eventos de acordo com uma prioridade determinada pelo instante em que ele deve ocorrer.

Uma simulação discreta de eventos modela o funcionamento de um sistema (p. ex., um banco ou uma lanchonete) como uma sequência discreta de eventos que devem ocorrer em instantes específicos, A ocorrência de um evento altera o estado do sistema que está sendo modelado e pode originar novos eventos. Por exemplo, numa simulação de funcionamento de uma lanchonete, quando ocorre um evento relacionado à chegada de clientes, o número de cadeiras disponíveis na lanchonete pode ser diminuído, alterando, assim, o estado do sistema

(i.e., da lanchonete). Esse mesmo evento pode dar origem a um evento de atendimento de pedido, que, por sua vez, quando processado dará origem a um evento de saída dos clientes da lanchonete e assim por diante.

Numa simulação discreta de eventos, não ocorre nenhuma alteração no sistema modelado entre eventos consecutivos, de modo que o programa de simulação pode saltar no tempo entre um evento e o próximo evento sem ter que se preocupar com o que ocorre nesse intervalo de tempo. Numa **simulação contínua**, por outro lado, o programa de simulação deve checar, de tempos em tempos, o estado do sistema.

Uma simulação contínua atualiza constantemente o estado do sistema. Nesse caso, um relógio virtual avança o tempo em segundos ou numa unidade mais conveniente e, após cada intervalo de tempo, o estado do sistema é atualizado de alguma maneira. Por exemplo, numa simulação de tráfego numa estrada, cada automóvel é caracterizado por sua posição que precisa ser atualizada a cada intervalo de tempo. Em contraste, o tempo numa simulação discreta avança em intervalos discretos e aleatórios (e consequentemente irregulares), que, tipicamente, são bem maiores do que numa simulação contínua. Nesse caso, o tempo avança do final de um evento para o início do próximo evento programado. Discutir simulação contínua em detalhes está bem além do escopo deste livro.

Uma simulação discreta deve manter, pelo menos, uma fila de prioridade que armazena os eventos pendentes.

Uma representação de eventos deve incluir, ao menos, o instante em que ele deverá ocorrer e o tipo de evento. Esse tipo indica como o evento deve ser processado; i.e., qual é a ação que deve ser realizada quando o evento ocorrer. Muitos eventos são programados dinamicamente (i.e., à medida que a simulação evolui). Por exemplo, numa simulação de lanchonete, eventos de saída são gerados à medida que eventos de atendimento são processados.

Suponha, por exemplo, que uma simulação pretende modelar o funcionamento de uma pequena lanchonete. Nesse caso, há três tipos de eventos de interesse na simulação, a saber:

- **Evento de chegada**, que corresponde à chegada de um grupo de clientes na lanchonete. (Um cliente solitário é considerado um grupo de apenas um cliente, obviamente.)
- **Evento de pedido** (ou **evento de atendimento**), que representa o atendimento que o grupo de clientes recebe, o que inclui o número de sanduíches que seus integrantes solicitam.
- **Evento de saída**, que corresponde à saída dos clientes da lanchonete, cedendo, assim, espaço para o atendimento de novos clientes.

Nesse exemplo de lanchonete, cada evento pode ser caracterizado pelo número de pessoas que dele participam, pelo instante em que ele deve ocorrer e pelo tipo de evento, como ilustra a **Figura 10–12**.

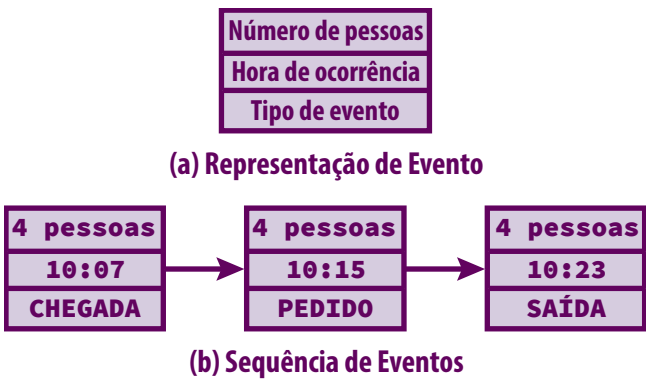


Figura 10–12: Representação e Sequência de Eventos Discretos

No início de uma simulação, é gerado aleatoriamente um certo número de eventos de chegada com um número de clientes também produzidos ao acaso. Esses eventos iniciais são adicionados à fila de prioridade na medida em que são criados e, então, a execução da simulação é iniciada. A execução da simulação completa segue os passos descritos na **Figura 10–13**.

1. Crie uma sequência de eventos iniciais e insira-os na lista de eventos
2. Enquanto a fila de eventos não estiver vazia, faça:
   - **2.1** Remova o próximo evento da fila
   - **2.2** Processe o evento

Figura 10–13: Laço Principal de Simulação de Eventos Discretos

O **Passo 2.2** do algoritmo delineado na **Figura 10–13** consiste em agir como se um evento de fato ocorresse. Por exemplo, processar um evento de entrada consiste em verificar se há espaço disponível na lanchonete e, se esse for o caso, gerar um evento de pedido que será inserido na fila de eventos. Quando não há espaço disponível, nesse caso, os clientes podem ser acomodados numa lista de espera ou simplesmente ir embora. No exemplo da lanchonete, processar um evento de pedido resulta na geração de um evento de saída e, quando esse último evento ocorre, surge espaço vazio que pode ser ocupado por novos clientes. Em qualquer tipo de evento, processá-lo tipicamente envolve a exibição de suas informações para facilitar a avaliação da simulação e, talvez, a criação de novos eventos.

A geração de intervalos de tempo aleatórios entre dois eventos independentes pode ser emulada usando **distribuição exponencial** por meio da seguinte fórmula[1]:

$$t = -m \cdot ln\ (1 - \alpha)$$

Nessa fórmula, tem-se que:

- m é o intervalo de tempo médio decorrido entre dois eventos e deve ser estimado experimentalmente
- α é um número real aleatório com ocorrência equiprovável no intervalo $[0, 1]$
- $ln(x)$ é o logaritmo neperiano de x

Para facilitar as implementações, frequentemente, os instantes de ocorrência dos eventos são considerados valores inteiros, que representam intervalos de tempo decorridos a partir de um instante de referência.

Tipicamente, simulações usam geradores de números aleatórios para obter, entre outras coisas, os instantes nos quais os eventos devem ocorrer. O uso de algoritmos que geram números pseudoaleatórios em detrimento daqueles que geram números genuinamente aleatórios é benéfico porque permite que a execução de uma simulação possa ser repetida usando parâmetros diferentes. Para tal, basta alimentar o gerador de número pseudoaleatórios com a mesma semente.

É responsabilidade do programador decidir quando a execução de uma simulação deve terminar. Tal execução pode encerrar após um determinado período ou quando o sistema atinge um determinado estado. Por exemplo, a execução de uma simulação de funcionamento de uma lanchonete pode encerrar após um predeterminado tempo de funcionamento ou após a venda de um número específico de sanduíches.

Simulações são usadas para testar e ajustar quando necessário o funcionamento de um sistema. Por exemplo, uma simulação de funcionamento de uma lanchonete pode indicar que, com base na frequência de clientes e no tempo de permanência deles, o número de cadeiras oferecidas não é suficiente e que, se esse número for aumentado, o faturamento pode ser aumentado. Do ponto de vista de programação, entender como funciona simulação de eventos discretos certamente também facilitará o aprendizado de programação baseada em eventos que permeia a construção de interfaces gráficas.

[1] Se você desconhece distribuição exponencial, consulte um bom texto sobre Estatística.

Um exemplo completo de simulação discreta será apresentado na **Seção 10.5.2**. Outros tipos de simulações podem ser descritos de modo semelhante a esse exemplo.

10.5 Exemplos de Programação

10.5.1 Codificação Canônica de Huffman

Preâmbulo: A **codificação padrão de Huffman** é uma técnica básica de compressão de arquivos que foi discutida exaustivamente no **Capítulo 12** do **Volume 1**. Em resumo, a codificação padrão de Huffman é representada por uma árvore estritamente binária e o algoritmo na **Figura 10–14** mostra como essa árvore pode ser obtida:

ALGORITMO CONSTRÓIÁRVOREDEHUFFMAN

ENTRADA: O arquivo a ser codificado

SAÍDA: Uma árvore de codificação de Huffman

1. Leia o arquivo e associe cada byte ao seu número de ocorrências (i.e., sua **frequência**) no arquivo
2. Crie nós-folhas, cada um dos quais contendo o valor de um byte e a frequência com que esse valor ocorre no arquivo
3. Insira os nós numa fila de prioridade de tal modo que, quanto menor for a frequência de um nó, maior seja sua prioridade
4. Enquanto a fila de prioridade não estiver vazia, faça:
   - **4.1** Remova um nó da fila de prioridade
   - **4.2** Se a fila ficou vazia, encerre (esse nó será a raiz da árvore de codificação)
   - **4.3** Remova outro nó da fila de prioridade
   - **4.4** Crie um nó cuja frequência seja a soma das frequências dos nós recém-removidos
   - **4.5** Insira o novo nó na fila de prioridade

FIGURA 10–14: ALGORITMO DE CONSTRUÇÃO DE ÁRVORE DE CODIFICAÇÃO DE HUFFMAN

A **Figura 10–15** mostra uma possível árvore de codificação obtida seguindo o algoritmo da **Figura 10–14** para um arquivo contendo os caracteres `'b'`, `'o'`, `'l'` e `'a'`. Nessa figura, o conteúdo de cada nó é dividido em duas partes: o byte (caractere) que o nó representa e a frequência de ocorrência desse byte. Nós que não são folhas não representam nenhum byte e, por isso, em vez do valor de byte na respectiva parte do conteúdo, aparece um símbolo de interrogação. A frequência de cada byte determina a profundidade da folha que o representa na árvore, de modo que os bytes que apresentam frequências altas são representados por folhas mais rasas e bytes que apresentam frequências baixas são representados por folhas mais profundas. Por exemplo, na **Figura 10–15**, a folha que representa o caractere `'a'`, que tem frequência igual a `14.26`, encontra-se no nível 1, enquanto a folha que representa o caractere `'b'`, que tem frequência igual a `1.54`, encontra-se no nível 3 (considera-se que a raiz tem nível 0). A razão pela qual existe essa relação entre a frequência de um byte e a profundidade da folha que o representa, bem como os significados dos números que rotulam as ramificações da árvore, serão esclarecidos em breve.

Se você não entendeu a sucinta explicação sobre a construção da árvore de codificação de Huffman, não adianta prosseguir nesta seção. O conselho, nesse caso, é consultar o **Capítulo 12** do **Volume 1** ou algum outro texto sobre o assunto antes de seguir em frente.

A árvore de codificação será utilizada para a obtenção do código que será associado a cada caractere. A principal diferença entre uma **codificação padrão de Huffman** e uma **codificação canônica** reside exatamente em *como* ela é usada. Numa codificação padrão, o código associado a cada byte é obtido descendo-se a árvore até encontrar a folha que representa esse byte. À medida que se desce a árvore, os bits que representarão a codificação do byte são coletados de tal maneira que, *tipicamente*, o bit *0* seja associado a uma descida seguindo uma ramificação esquerda e o bit *1* seja associado a uma descida seguindo uma ramificação direita. Assim, na **Figura 10–15**, a codificação do caractere `'a'` é `0` e a codificação do caractere `'b'` é `100`. Isso explica por que as folhas que representam os bytes com maiores frequências estão em níveis mais rasos da árvore: desse modo, a eles serão atribuídos códigos mais curtos e, assim, o tamanho do arquivo codificado será menor.

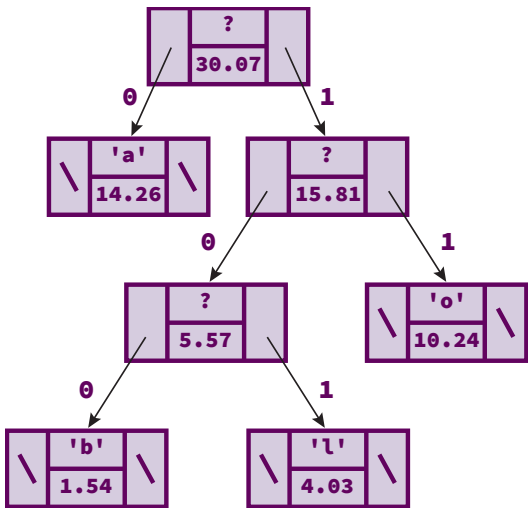


Figura 10–15: Árvore Resultante de uma Codificação de Huffman

É importante ressaltar que qualquer árvore resultante do algoritmo de Huffman é estritamente binária, o que quer dizer que um nó dessa árvore ou é folha ou possui dois filhos (i.e., não existe nó com apenas um filho em tal árvore). Uma importante consequência disso é que a máxima de uma árvore dessa natureza é igual ao seu número de folhas. E, se você estiver acompanhando o raciocínio desenvolvido até aqui, irá concluir que o número máximo de bits na codificação de um byte é igual ao número de folhas da árvore de codificação. Essa conclusão tem uma importante consequência prática na implementação do algoritmo de compressão (v. adiante).

O algoritmo de codificação canônica de Huffman parte do princípio de que os códigos atribuídos aos bytes obtidos usando o mecanismo descrito no último parágrafo não são de fato importantes; i.e., o que realmente importa é o *tamanho* deles. Há, porém, uma ressalva: esses códigos devem ainda ser considerados livres de prefixo como ocorre quando se segue o algoritmo padrão de codificação[2]. A codificação canônica oferece duas vantagens em relação à codificação tradicional:

1. Ela permite que o **cabeçalho** da codificação ocupe menos espaço no arquivo codificado. Esse cabeçalho, também denominado **metadados**, contém informações vitais para que o arquivo possa ser decodificado posteriormente. Na codificação padrão, a árvore de codificação precisa ser armazenada nesse cabeçalho.

[2] Uma codificação é livre de prefixo quando o código atribuído a um byte não faz parte da porção inicial do código atribuído a outro byte. Por exemplo, na **Figura 10–15**, o código do caractere `'a'` é `0`, de sorte que o código de nenhum outro caractere começa com `0`.

2. Ela é mais fácil de implementar, pois não requer que a árvore de codificação seja usada na codificação após os tamanhos de códigos terem sido obtidos. Na codificação padrão, essa árvore é usada na codificação de cada byte lido no arquivo que está sendo codificado.

A obtenção dos códigos associados aos bytes numa codificação canônica segue o algoritmo apresentado na **Figura 10–16**.

ALGORITMO OBTÉMCÓDIGOSDEHUFFMAN

ENTRADA: Uma árvore de codificação de Huffman

SAÍDA: Uma lista de códigos canônicos

1. Efetue um caminhamento na árvore de codificação e obtenha o **comprimento** (i.e., o **número de bits**) do código associado a cada byte armazenado na árvore
2. Libere o espaço ocupado pela árvore de codificação (ela não é mais necessária)
3. Ordene os bytes que serão codificados de acordo com os tamanhos de seus códigos (se dois bytes tiverem os mesmos tamanhos de códigos, use os valores desses bytes como critério de desempate)
4. Atribua *0* ao byte cujo tamanho de código seja o menor de todos
5. Para cada valor de byte restante, atribua um código do seguinte modo:
   - 5.1 Acrescente *1* ao código do byte anterior para obter o código do byte corrente
   - 5.2 Se o tamanho do código do byte corrente for maior do que o tamanho do byte anterior na sequência, efetue um deslocamento de bits à esquerda (ou seja, acrescente *0* à direita do código corrente)

FIGURA 10–16: CODIFICAÇÃO CANÔNICA DE HUFFMAN: OBTENÇÃO DOS CÓDIGOS

Resumidamente, a seguinte fórmula pode ser usada para atribuir valores aos códigos dos bytes:

$$c_i = \begin{cases} 0 & se\ i = 0 \\ (c_{i-1} + 1) << (t_i - t_{i-1}) & se\ i > 0 \end{cases}$$

Nessa fórmula, c_i é o código atribuído a um byte, t_i é o seu tamanho e << representa o operador de deslocamento à esquerda (v. **Apêndice B**).

Suponha, por exemplo, que os caminhamentos prescritos no **Passo 1** do algoritmo da **Figura 10–16** sejam efetuados na árvore da **Figura 10–15**. Então os **códigos canônicos** (i.e., aqueles obtidos usando o algoritmo de codificação canônica de Huffman) serão aqueles exibidos na **Tabela 10–2**. A última coluna da **Tabela 10–2** apresenta a justificativa, baseada no algoritmo da **Figura 10–16**, para a obtenção do respectivo código canônico.

BYTE	TAMANHO	CÓDIGO CANÔNICO	JUSTIFICATIVA
`'a'`	`1`	`0`	**Passo 3**
`'o'`	`2`	`10`	Incremento seguido de deslocamento (**Passos 4.1** e **4.2**)
`'b'`	`3`	`110`	Incremento seguido de deslocamento (**Passos 4.1** e **4.2**)
`'l'`	`3`	`111`	Incremento (**Passo 4.1**)

TABELA 10–2: EXEMPLO DE CODIFICAÇÃO CANÔNICA

Para efeito de comparação, a **Tabela 10–3** mostra a codificação que seria obtida usando o algoritmo de codificação padrão de Huffman.

Byte	Tamanho	Código Canônico	Código Padrão
`'a'`	1	0	0
`'o'`	2	10	11
`'b'`	3	110	100
`'l'`	3	111	101

Tabela 10–3: Codificação Canônica versus Codificação Padrão

Neste ponto, o leitor atento pode estar confuso. Afinal, foi dito acima que os valores específicos dos códigos atribuídos aos bytes numa codificação de Huffman são irrelevantes. Quer dizer, por exemplo, qual é a diferença entre atribuir `10` ou `11` ao caractere `'o'` quando o que realmente importa é o tamanho desses códigos e, nesse caso, ambos têm o mesmo tamanho? Ou seja, qual é a vantagem da codificação canônica? A resposta a essa questão está no fato de a codificação canônica não requerer que a árvore de codificação seja armazenada no cabeçalho do arquivo codificado para permitir que ele seja decodificado. Mais precisamente, para permitir que um arquivo codificado seguindo esse algoritmo seja decodificado, basta armazenar no arquivo codificado uma lista na qual cada elemento seja constituído por um byte do arquivo e o comprimento de sua codificação. Então, usando-se essa lista e o algoritmo usado para obtenção dos códigos dos bytes, pode-se decodificar o arquivo e, enfim, obter de volta o arquivo original.

Após a obtenção dos códigos associados aos bytes do arquivo de entrada, a criação do arquivo codificado de saída pode ser efetuada seguindo o algoritmo apresentado na **Figura 10–17**.

Algoritmo CodificaArquivo

Entrada: O arquivo que será codificado e sua lista de códigos canônicos

Saída: O arquivo codificado

1. Escreva o cabeçalho da codificação no arquivo de saída
2. Enquanto houver bytes a ser lidos no arquivo de entrada, faça o seguinte:
   - **2.1** Leia um byte no arquivo de entrada
   - **2.2** Encontre o valor do byte lido na lista de códigos canônicos
   - **2.3** Escreva os bits que constituem o código do referido byte no arquivo de saída
3. Libere o espaço ocupado pela lista de códigos canônicos
4. Feche os arquivos

Figura 10–17: Codificação Canônica de Huffman: Escrita de Arquivo

Problema: Escreva um programa que lê um arquivo e codifica-o ou decodifica-o (se ele estiver codificado, obviamente) usando o algoritmo de codificação canônica descrito no preâmbulo.

Solução: A implementação do algoritmo de Huffman irá requerer o uso de três estruturas de dados básicas:

- **Árvore de codificação** que é uma árvore estritamente binária da qual se obtêm os tamanhos das sequências de bits (códigos) almejadas.
- **Fila de prioridade** implementada num heap binário de mínimo que armazena os nós da árvore de codificação em ordem crescente de frequência. No começo da execução do algoritmo de construção da referida árvore, esse heap conterá apenas suas folhas, que são os nós que armazenam os

bytes a ser codificados. Após a execução do último passo do algoritmo de construção da árvore, o heap deverá estar vazio e não será mais necessário.

- **Lista de códigos canônicos** que armazena um mapeamento entre os valores de bytes encontrados no arquivo sendo codificado e os códigos aos quais eles serão associados. Como o número de elementos dessa lista é relativamente bem pequeno[3], pode-se usar uma lista indexada comum para representá-la sem que essa decisão afete o desempenho da implementação.

As duas primeiras estruturas de dados descritas acima requerem as seguintes definições de tipos:

```
typedef unsigned char tByte; /* Tipo de um byte */

   /* Tipo do campo de informação dos nós da árvore de codificação */
typedef struct {
          tByte  byte;
          double frequencia;
        } tByteFreq;

   /* Tipo de cada nó e tipo de ponteiro para nó da árvore de codificação */
typedef struct rotNoArvoreHuff {
          tByteFreq                byteFreq;
          struct rotNoArvoreHuff *filhoEsquerda;
          struct rotNoArvoreHuff *filhoDireita;
        } tNoArvoreHuff, *tArvoreHuff;

   /* Tipo de nó de um heap que armazena ponteiros para */
   /* os nós da árvore de codificação temporariamente   */
typedef tNoArvoreHuff *tNoHeapHuff;

   /* Tipo de heap que armazena as informações */
   /* sobre os nós da árvore de codificação    */
typedef struct {
           tNoHeapHuff *itens;      /* Array de elementos */
           int          capacidade; /* Quantos elementos o heap pode conter */
           int          nItens;     /* Número de elementos do heap */
        } tHeapHuff;

   /* Tipo de ponteiro para função de comparação que compara elementos do heap */
typedef int (*tFComparaHuff) (const void *, const void *);
```

O tipo de heap, que armazena as informações sobre os nós da árvore de codificação, e o tipo de ponteiro para função de comparação, que compara elementos do heap, são definidos do mesmo modo que ocorre na **Seção 10.2.4**. Note, entretanto, que a função de comparação a ser usada aqui deve ser definida de modo diferente, já que o tipo de informação armazenada no heap agora é diferente daquele usado na referida seção. Essa função deve ser definida como:

```
static int ComparaNosHeapHuff(const void *e1, const void *e2)
{
   tNoHeapHuff elemento1 = (tNoHeapHuff)e1, elemento2 = (tNoHeapHuff)e2;
   ASSEGURA(e1 && e2, "Elemento nulo recebido");

   return elemento1->byteFreq.frequencia - elemento2->byteFreq.frequencia;
}
```

Compressão

A função `Codifica()` codifica um arquivo seguindo o algoritmo de codificação de Huffman e seus parâmetros são o nome do arquivo que será codificado (`arqEntrada`) e o nome do arquivo que conterá a codificação (`arqSaida`).

[3] No máximo, essa lista terá 256 elementos, que é o número máximo de valores distintos de bytes.

```
void CodificaHuff(const char *arqEntrada, const char *arqSaida)
{
   FILE          *streamE, /* Associado ao arquivo a ser codificado */
                 *streamS; /* Associado ao arquivo resultante */
   tHeapHuff     *pHeap; /* Apontará para o heap que armazena     */
                         /* temporariamente os nós da codificação */
   tNoArvoreHuff *no1, /* Ponteiro para o nó com a menor frequência no heap */
                 *no2, /* Ponteiro para o nó com a segunda menor frequência no heap */
                 *noNovo; /* Ponteiro para um novo nó da árvore */
                          /* que terá no1 e no2 como filhos     */
   tNoListaCanHuff *lista; /* Lista que conterá os códigos canônicos */
   tCabecalhoHuff   cabecalho; /* Cabeçalho da codificação */
   int              i, j,
                    c, /* Armazenará um byte lido */
                    bits, /* Empacotará os bits */
                    nFolhas, /* Número de valores de bytes distintos no arquivo */
                    contaBits, /* Conta os bits de um byte */
                    tamArquivo; /* Número de bytes do arquivo */

   streamE = AbreArquivo(arqEntrada, "rb"); /* Abre o arquivo que será codificado */

      /* Cria o heap que armazena temporariamente os nós da árvore de codificação */
   pHeap = CriaHeapHuff(streamE, &tamArquivo);

      /* O número de folhas da árvore de codificação será igual ao tamanho */
      /* inicial do heap. Esse valor é igual ao número de valores de bytes */
      /* distintos encontrados no arquivo.                                 */
   nFolhas = TamanhoHeapHuff(pHeap);

      /* Constrói a árvore de codificação */
   while (!HeapVazioHuff(pHeap)) {
         /* Remove o nó com a menor frequência do heap */
      no1 = RemoveMinHeapHuff(pHeap, ComparaNosHeapHuff);

         /* Se o heap ficou vazio, o último nó removido é */
         /* a raiz da árvore que representa a codificação */
      if (HeapVazioHuff(pHeap))
         break; /* Construção de árvore concluída */

         /* Remove o nó com a segunda menor frequência do heap  */
      no2 = RemoveMinHeapHuff(pHeap, ComparaNosHeapHuff);

      noNovo = ConstroiNoArvoreHuff(); /* Tenta alocar um novo nó */

         /* Armazena a frequência do novo nó. O conteúdo do campo 'byte' não */
         /* tem importância porque este nó não é uma folha.                  */
      noNovo->byteFreq.frequencia=no1->byteFreq.frequencia no2->byteFreq.frequencia;

         /* O filho esquerdo do novo nó é o primeiro nó removido do heap */
         /* e o filho direito é o segundo nó removido do heap            */
      noNovo->filhoEsquerda = no1;
      noNovo->filhoDireita = no2;

      InsereEmHeapHuff(pHeap, noNovo, ComparaNosHeapHuff); /* Insere o nó no heap */
   }

       /* Neste ponto, a árvore de codificação já foi concluída e sua raiz é no1 */

      /* Obtém a lista de códigos canônicos */
   lista = CriaListaCanonicaHuff(no1, nFolhas);

      /* Nem o heap nem a árvore são mais necessários */
```

```
DestroiHeapHuff(pHeap);
free(pHeap);
DestroiArvoreHuff(no1);

streamS = AbreArquivo(arqSaida, "wb"); /* Abre o arquivo de saída */

   /* Cria o cabeçalho da codificação */
cabecalho.tamLista = nFolhas;
cabecalho.lista = lista;

   /* Escreve o cabeçalho da codificação no arquivo de saída */
EscreveCabecalhoHuff(&cabecalho, streamS);

bits = 0; /* Zera o empacotador de bits */
contaBits = 0; /* Inicia a contagem de bits */

   /* Faz o apontador de posição do arquivo apontar para o seu primeiro byte */
rewind(streamE);

   /* Lê cada byte no arquivo de entrada, codifica-o usando a lista */
   /* de códigos e escreve a codificação no arquivo de saída        */
while(1) {
   c = fgetc(streamE); /* Lê um byte no arquivo de entrada */

      /* Verifica se ocorreu erro */
   ASSEGURA(!ferror(streamE), "Erro de leitura");

   if (feof(streamE))
      break; /* Acabou a leitura */

      /* Encontra o valor do byte lido na lista de códigos. */
   for (i = 0; i < nFolhas; ++i)
      if (lista[i].valor == c)
         break;

      /* O valor do byte deve ter sido encontrado */
   ASSEGURA(i < nFolhas, "Valor de byte nao encontrado");

      /* Empacota os bits do código deste byte */
   for(j = 0; j < lista[i].nBits; ++j) {
      contaBits++; /* Mais um bit neste código */

         /* Armazena bit na variável que empacota os bits */
      bits = (bits << 1) | ConsultaBitEmArray(lista[i].cod, j);

         /* Se já houver um byte na variável de          */
         /* empacotamento, escreve-o no arquivo de saída */
      if (contaBits == CHAR_BIT) {
         fputc(bits, streamS); /* Escreve byte no arquivo */

            /* Verifica se ocorreu erro de escrita */
         ASSEGURA(!ferror(streamS), "Erro de escrita");

         contaBits = 0; /* Reinicia a contagem de bits */
      }
   }
}

   /* Escreve os bits que restaram no último empacotamento */
if (contaBits) {
   bits <<= CHAR_BIT - contaBits; /* Alinha bits à esquerda */
   fputc(bits, streamS); /* Escreve-os no arquivo */
   ASSEGURA(!ferror(streamS), "Erro de escrita"); /* Testa */
}
```

```
   free(lista); /* Libera o espaço ocupado pela lista de códigos */

   FechaArquivo(streamE, arqEntrada); /* Fecha os arquivos */
   FechaArquivo(streamS, arqSaida);
}
```

A função `CodificaHuff()` chama as funções descritas na **Tabela 10–4**. A última coluna dessa tabela indica onde o leitor pode encontrar a definição da respectiva função ou de uma função similar.

Função	O que faz	Referência
`AbreArquivo()`	Abre um arquivo	Seção 2.3
`CriaHeapHuff()`	Cria um heap contendo as folhas de uma árvore de codificação de Huffman	Discutida abaixo
`TamanhoHeapHuff()`	Calcula o tamanho de um heap	Seção 10.2.4
`HeapVazioHuff()`	Verifica se um heap está vazio	Seção 10.2.4
`RemoveMinHeapHuff()`	Remove a raiz de um heap de mínimo	Seção 10.2.4
`ConstroiNoArvoreHuff()`	Constrói um nó de uma árvore binária	Semelhante à função vista na Seção 4.1.2
`InsereEmHeapHuff()`	Insere um nó num heap	Seção 10.2.4
`CriaListaCanonicaHuff()`	Constrói uma lista de códigos canônicos	Discutida adiante
`DestroiHeapHuff()`	Libera o espaço ocupado por um heap	Seção 10.2.4
`DestroiArvoreHuff()`	Libera o espaço ocupado por uma árvore binária	Capítulo 12 do Volume 1
`EscreveCabecalhoHuff()`	Escreve o cabeçalho da codificação no arquivo de saída	Discutida adiante
`ConsultaBitEmArray()`	Verifica se um bit está ligado num array de elementos do tipo **unsigned char**	V. função `ConsultaBit()` no Apêndice B
`FechaArquivo()`	Fecha um arquivo	Seção 4.1.2

Tabela 10–4: Funções Chamadas pela Função Codifica()

A função `CriaHeapHuff()` apresentada a seguir implementa os três primeiros passos do algoritmo de construção da árvore de codificação exibido na **Figura 10–14**. Quer dizer, essa função cria um heap contendo nós de uma árvore que armazenam valores de bytes e suas respectivas frequências. Essa função retorna o endereço do heap criado e seus parâmetros são:

- `stream` — stream associado ao arquivo no qual os bytes serão lidos
- `nBytes` (saída) - número total de bytes lidos no arquivo

```
static tHeapHuff *CriaHeapHuff(FILE *stream, int *nBytes)
{
   tHeapHuff     *pHeap; /* Ponteiro para o heap que será criado */
   tNoArvoreHuff *ptrNovoNo; /* Apontará para o novo nó alocado */
   int           *byteFreq, /* Apontará para um array contendo os */
                            /* números de ocorrências dos bytes   */
                  umByte, /* Armazena um byte */
                  tamArquivo = 0, /* Número de bytes do arquivo */
                  i;
```

```
      /* Aloca um array para conter as ocorrências dos bytes */
   byteFreq = calloc(UCHAR_MAX + 1, sizeof(int));
   ASSEGURA(byteFreq, "Array de ocorrencias nao foi alocado");

   rewind(stream); /* Move o apontador de posição para o início do arquivo */

      /* Obtém o número de ocorrências de cada byte */
   while (1) {
      umByte = fgetc(stream); /* Lê um byte */

         /* Verifica se ocorreu erro */
      ASSEGURA(!ferror(stream), "Erro de leitura de arquivo");

      if (feof(stream))
         break; /* Não há mais bytes a serem lidos */

         /* Verifica se o programa pode lidar a frequência de ocorrência */
         /* desse byte. A ausência desse teste causará overflow.         */
      ASSEGURA(byteFreq[umByte] < INT_MAX,
               "Este programa nao suporta um arquivo tao grande");

         /* Incrementa o número de ocorrências do último byte lido */
      ++byteFreq[umByte];

         /* Verifica se o programa pode lidar com o tamanho do */
         /* arquivo. A ausência desse teste causará overflow.  */
      ASSEGURA( tamArquivo < INT_MAX,
                "Este programa nao suporta um arquivo tao grande" );

      ++tamArquivo; /* Mais um byte foi lido */
   }

      /* Aloca a variável que armazenará o heap */
   ASSEGURA(pHeap = malloc(sizeof(*pHeap)), "Heap nao alocado");

   IniciaHeapHuff(pHeap); /* Inicia o heap */

      /* Constrói o heap */
   for (i = 0; i < UCHAR_MAX; ++i)
         /* Cria um nó para cada valor de byte encontrado no arquivo */
      if (byteFreq[i]) {
         ptrNovoNo = ConstroiNoArvoreHuff(); /* Tenta alocar um novo nó */

            /* Armazena o byte corrente e sua frequência no novo nó */
         ptrNovoNo->byteFreq.byte = i;
         ptrNovoNo->byteFreq.frequencia = (double) 100*byteFreq[i]/tamArquivo;

            /* Insere o novo nó no heap */
         InsereEmHeapHuff(pHeap, ptrNovoNo, ComparaNosHeapHuff);
      }

   free(byteFreq); /* O array de ocorrências não é mais necessário */

   *nBytes = tamArquivo; /* Atualiza o segundo parâmetro antes de retornar */

   return pHeap; /* Retorna o endereço do heap */
}
```

O tipo de elemento usado na obtenção dos códigos canônicos é definido como:

```
typedef struct {
          tByte valor;          /* Valor do byte */
          tByte nBits;          /* Número de bits na codificação */
          tByte cod[MAX_BYTES]; /* Codificação do byte */
       } tNoListaCanHuff;
```

A constante simbólica `MAX_BYTES` usada na definição do tipo `tNoListaCanHuff` representa o número máximo de bytes de um código associado a um byte e é definida como:

```
   /* Número máximo de bits num código associado a um byte */
#define MAX_BITS  (UCHAR_MAX + 1)

   /* Número máximo de bytes num código associado a um byte */
#define MAX_BYTES MAX_BITS/CHAR_BIT
```

Nessas definições de constantes, **UCHAR_MAX** e **CHAR_BIT** são constantes definidas no cabeçalho `<limits.h>`. O tamanho máximo de um código é calculado levando em conta as seguintes considerações:

1. Como qualquer árvore de codificação é estritamente binária, a profundidade máxima dela é igual ao seu número de folhas.
2. O número máximo de folhas possível numa árvore de codificação é igual ao número de possíveis valores distintos de bytes. Nesse caso, o menor valor é *0* e o maior valor é igual ao valor da constante **UCHAR_MAX**. Portanto a profundidade máxima possível de uma árvore de codificação é `UCHAR_MAX + 1`, que corresponde ao tamanho máximo de uma sequência de bits que representa um código de byte.

A função `CriaListaCanonicaHuff()` constrói uma lista de códigos canônicos e tem como parâmetros:

- `arvore` (entrada) — ponteiro para raiz da árvore a partir da qual a lista será criada
- `nItens` (entrada) — número de elementos que a lista terá

```
static tNoListaCanHuff *CriaListaCanonicaHuff(tArvoreHuff arvore, int nItens)
{
   tNoListaCanHuff *listaCanonica;
   int             *tamanhos, i, j;

      /* Aloca um array para conter os tamanhos dos */
      /* códigos associados aos valores de bytes    */
   tamanhos = calloc(UCHAR_MAX + 1, sizeof(int));
   ASSEGURA(tamanhos, "Array de tamanhos nao foi alocado");

      /* Obtém os tamanhos dos códigos dos bytes */
   ObtemTamanhosHuff(arvore, tamanhos, 0);

      /* Aloca a lista de códigos canônicos */
   listaCanonica = calloc(nItens, sizeof(tNoListaCanHuff));
   ASSEGURA(listaCanonica, "Lista canonica nao foi alocada");

      /* Copia os tamanhos dos códigos para a lista de códigos canônicos */
   for (i = 0, j = 0; i < UCHAR_MAX + 1; ++i)
      if (tamanhos[i]) { /* A lista canônica contém apenas tamanhos não nulos */
         listaCanonica[j].valor = i;
         listaCanonica[j].nBits = tamanhos[i];
         ++j;
      }

   free(tamanhos); /* O array que contém os tamanhos não é mais necessário */

      /* Ordena lista de códigos canônicos por tamanho de código e valor de byte */
   qsort( listaCanonica, nItens, sizeof(tNoListaCanHuff), ComparaTamanhosHuff );

      /* Obtém os códigos canônicos */
   ObtemCodigosCanonicosHuff(listaCanonica, nItens);

   return listaCanonica;
}
```

A função `ObtemTamanhos()`, que será exibida a seguir, implementa o **Passo 1** do algoritmo da **Figura 10–16**. Quer dizer, ela é responsável pela obtenção dos tamanhos dos códigos associados aos bytes armazenados nas folhas de uma árvore de codificação de Huffman. Os parâmetros dessa função são:

- `raiz` (entrada) — ponteiro para a raiz da árvore que contém a codificação
- `tabela[]` (saída) — tabela de consulta que conterá os tamanhos dos códigos canônicos
- `n` (entrada) — nível da árvore na qual a visitação dos nós será iniciada. Na primeira chamada dessa função, o valor desse parâmetro deve ser `0`

```
static void ObtemTamanhosHuff(tArvoreHuff raiz, int tabela[], int n)
{
      /* A árvore não pode estar vazia */
   ASSEGURA(raiz, "A raiz da arvore e' nula");

      /* Se for possível, caminha na subárvore esquerda do próximo nível */
   if (raiz->filhoEsquerda)
      ObtemTamanhosHuff(raiz->filhoEsquerda, tabela, n + 1);

      /* Se for possível, caminha na subárvore direita do próximo nível */
   if (raiz->filhoDireita)
      ObtemTamanhosHuff(raiz->filhoDireita, tabela, n + 1);

      /* Se o nó corrente for uma folha, ele armazena um valor de byte do arquivo */
   if (!raiz->filhoEsquerda && !raiz->filhoDireita)
         /* Armazena no índice da tabela correspondente ao valor do byte       */
         /* armazenado nessa folha o tamanho que foi acumulado no caminhamento */
      tabela[raiz->byteFreq.byte] = n;
}
```

A função `ObtemTamanhos()` possui dois casos recursivos e um caso terminal e funciona assim:

- São efetuados caminhamentos na árvore de codificação, sendo que cada caminhamento começa sempre na raiz de uma subárvore e termina numa folha dessa subárvore.
- Quando, num desses caminhamentos, se segue um filho esquerdo ou direito, incrementa-se o nível da raiz da subárvore na qual o próximo caminhamento recursivo iniciará.
- A base da recursão é atingida quando se visita uma folha. Nesse caso, armazena-se no elemento que se encontra no índice da tabela correspondente ao valor do byte representado nessa folha o nível em que ela se encontra. O valor desse nível é exatamente o tamanho da codificação desse valor de byte.

O **Passo 2** do algoritmo da **Figura 10–16** é implementado por meio de uma chamada da função **qsort()** da biblioteca padrão de C (v. **Capítulo 11** do **Volume 1**). A função `ComparaTamanhosHuff()`, apresentada a seguir, é a função de comparação usada por **qsort()** para ordenar a lista de códigos canônicos por tamanhos de códigos e por valores de bytes e seus parâmetros são ponteiros para elementos de uma lista de códigos canônicos.

```
static int ComparaTamanhosHuff(const void *e1, const void *e2)
{
   int nBits1 = ((tNoListaCanHuff *)e1)->nBits,
       nBits2 = ((tNoListaCanHuff *)e2)->nBits,
       valor1 = ((tNoListaCanHuff *)e1)->valor,
       valor2 = ((tNoListaCanHuff *)e2)->valor;

   if (nBits1 > nBits2)
      return 1; /* e1 > e2 */
   else if (nBits1 < nBits2)
      return -1; /* e1 < e2 */

      /* Se ainda não houve retorno, os tamanhos são iguais e */
      /* o maior elemento será aquele que tiver o maior valor */
```

```
   if (valor1 > valor2)
      return 1;  /* e1 > e2 */
   else if (valor1 < valor2)
      return -1;  /* e1 < e2 */

      /* Já deveria ter havido retorno */
   ASSEGURA(0, "Erro: houve empate em ComparaTamanhosHuff()");
}
```

Nessa última função, note que, se os tamanhos dos códigos forem iguais, os valores dos bytes serão usados para desempatar, de modo que essa função nunca retorna zero.

Os demais passos do algoritmo da **Figura 10–16** são implementados pela função `ObtemCodigosCanonicosHuff()` que atribui um código canônico para cada byte de uma lista ordenada de acordo com os tamanhos de códigos e os valores dos bytes. Os parâmetros dessa função são:

- `lista` (entrada e saída) — a lista ordenada de bytes
- `nItens` (entrada) — número de elementos da lista

```
static void ObtemCodigosCanonicosHuff(tNoListaCanHuff lista[], int nItens)
{
   int   i;
   tByte tamanho;
   tByte codigo[MAX_BYTES] = {0};

      /* Obtém o tamanho do código armazenado no primeiro elemento da lista */
   tamanho = lista[0].nBits;

      /* Atribui códigos aos elementos da lista */
   for(i = 0; i < nItens; ++i) {
         /* Verifica se o tamanho do código é válido */
      ASSEGURA(lista[i].nBits, "Tamanho de codigo e' zero");

      IncrementaBits(codigo); /* Soma 1 ao código do byte anterior */

         /* Ajusta o código se seu tamanho for menor */
         /* do que o tamanho do código anterior      */
      if (lista[i].nBits > tamanho) {
         DeslocaBitsEsquerda(codigo, lista[i].nBits - tamanho);
         tamanho = lista[i].nBits;
      }

         /* Copia o código obtido para o respectivo */
         /* elemento da lista e alinha-o à esquerda */
      memmove(lista[i].cod, codigo, MAX_BYTES);
      DeslocaEsquerda(lista[i].cod, MAX_BITS - tamanho);
   }
}
```

A função `ObtemCodigosCanonicos()` chama as seguintes funções de processamento de bits:

- `DeslocaBitsEsquerda()`, que desloca à esquerda os bits de um array de bytes pela quantia de posições especificada
- `IncrementaBits()`, que incrementa um array de elementos do tipo **unsigned char** como se ele fosse um único valor inteiro de 256 bits

Discutir essas funções em detalhes está além do escopo deste livro, mas suas definições podem ser encontradas no site dedicado ao livro na internet.

O cabeçalho a ser armazenado no arquivo de saída usa a seguinte definição de tipo:

```
   /* Tipo de estrutura que armazena os dados de um cabeçalho */
typedef struct {
          int              tamLista; /* Número de elementos da lista */
                                     /* de códigos canônicos         */
          tNoListaCanHuff *lista; /* A lista de códigos canônicos */
       } tCabecalhoHuff;
```

A função `EscreveCabecalhoHuff()` escreve o cabeçalho da codificação no início do arquivo de saída e tem como parâmetros:

- `cabecalho` (entrada) — ponteiro para o cabeçalho que será escrito
- `stream` (entrada) — stream associado ao arquivo de saída

```
static void EscreveCabecalhoHuff(const tCabecalhoHuff *cabecalho, FILE *stream)
{
      /* Escreve o número de elementos da lista */
   fwrite( &cabecalho->tamLista, sizeof(cabecalho->tamLista), 1, stream );

      /* Escreve a lista de códigos canônicos */
   fwrite(cabecalho->lista, sizeof(tNoListaCanHuff), cabecalho->tamLista, stream);
}
```

Embora tenha sido dito que apenas os tamanhos dos códigos precisariam ser armazenados no cabeçalho, por comodidade do *programador*, a função `EscreveCabecalhoHuff()` armazena não apenas esses tamanhos como também os próprios códigos.

Descompressão

A decodificação de um arquivo codificado usando o algoritmo de codificação canônica de Huffman apresentado acima é descrito na **Figura 10–18**.

ALGORITMO DECODIFICAARQUIVO

ENTRADA: Arquivo codificado (seguindo o mesmo algoritmo)

SAÍDA: Arquivo decodificado

1. Leia o cabeçalho no arquivo de entrada
2. Usando a lista de códigos canônicos armazenada no cabeçalho, construa a árvore de codificação
3. Enquanto houver bytes a serem lidos no arquivo de entrada, faça o seguinte:
   - **3.1** Leia no arquivo codificado e armazene num buffer um número de bits igual ao número máximo de bits de um código
   - **3.2** Decodifique os bits lidos
   - **3.3** Escreva o resultado da decodificação no arquivo de saída

FIGURA 10–18: CODIFICAÇÃO CANÔNICA DE HUFFMAN: DECODIFICAÇÃO DE ARQUIVO

A implementação do **Passo 1** do algoritmo de decodificação é realizada pela função `LeCabecalhoHuff()` que lê o cabeçalho da codificação num arquivo de entrada codificado e tem como parâmetros:

- `cabecalho` (saída) — ponteiro para o cabeçalho que será lido
- `stream` (entrada) — stream associado ao arquivo de saída

Essa função retorna o endereço da lista de códigos canônicos lida no arquivo ou **NULL**, se ocorrer algum erro.

```
static tNoListaCanHuff *LeCabecalhoHuff(tCabecalhoHuff *cabecalho, FILE *stream)
{
      /* Lê o número de elementos da lista */
```

```
   fread( &cabecalho->tamLista, sizeof(cabecalho->tamLista), 1, stream );

   if (ferror(stream))
      return NULL; /* Ocorreu erro de leitura */

      /* Aloca espaço para a lista de códigos canônicos */
   cabecalho->lista = calloc( cabecalho->tamLista, sizeof(tNoListaCanHuff) );
   if(!cabecalho->lista)
      return NULL; /* Alocação falhou */

      /* Lê a lista de códigos canônicos */
   fread(cabecalho->lista, sizeof(tNoListaCanHuff), cabecalho->tamLista, stream);

   if (ferror(stream))
      return NULL; /* Ocorreu erro de leitura */

   return cabecalho->lista;
}
```

A construção da árvore de codificação (**Passo 2** do algoritmo de decodificação) é implementada pela função `ReconstroiArvoreHuff()` que reconstrói uma árvore de codificação a partir de uma lista de códigos. Essa função retorna o endereço da raiz da árvore assim construída e usa os seguintes parâmetros:

- `lista` (entrada) — a lista de códigos
- `tamLista` (entrada) — tamanho da lista

```
static tArvoreHuff ReconstroiArvoreHuff(tNoListaCanHuff lista[], int tamLista)
{
   tNoArvoreHuff *raiz, /* Raiz da árvore a ser criada */
                 *p, /* Ponteiro usado para descer a árvore    */
                 *q; /* Segue o ponteiro p um nível acima dele */
   int            i, j;

   raiz = ConstroiNoArvoreHuff(); /* Cria a raiz da árvore */

      /* Para cada valor de byte da lista, desce-se a árvore criando ou */
      /* visitando nós de acordo com o código associado ao byte         */
   for (i = 0; i < tamLista; ++i) {
         /* A descida começa sempre pela raiz */
      q = NULL;
      p = raiz;

         /* Acessa cada bit do código associado ao byte corrente e desce a */
         /* árvore de acordo com seu valor: se o bit for 0, desce-se pela  */
         /* esquerda; se o bit for 1, desce-se pela direita                */
      for (j = 0; j < lista[i].nBits; ++j) {
         q = p; /* Guarda o endereço do nó corrente antes de descer */

         if (ConsultaBitEmArray(lista[i].cod, j)) {
            p = p->filhoDireita; /* O bit é 1 e a descida é pela direita */

               /* Se o nó não existir, cria-se um nó que será filho direito de q */
            if (!p) {
               p = ConstroiNoArvoreHuff();
               q->filhoDireita = p; /* Inclui novo nó na árvore */
            }
         } else {
            p = p->filhoEsquerda; /* O bit é 0 e a descida é pela esquerda */

               /* Se o nó não existir, cria-se um nó que será filho esquerdo de q */
            if (!p) {
               p = ConstroiNoArvoreHuff();
```

```
                q->filhoEsquerda = p;/* Inclui novo nó na árvore */
             }
          }
       }

          /* Neste ponto, p deve estar apontando para uma folha */
       ASSEGURA( !p->filhoEsquerda && !p->filhoDireita, "Deveria ser folha!" );

       p->byteFreq.byte = lista[i].valor; /* Armazena o valor do byte na folha */
   }

   return raiz;
}
```

A função `DecodificaHuff()`, apresentada a seguir, completa a implementação do algoritmo de decodificação descrito na **Figura 10–18**. Os parâmetros dessa função são:

- `arqEntrada` (entrada) — nome do arquivo a ser decodificado
- `arqSaida` (entrada) — nome do arquivo no qual o resultado será escrito

```
void DecodificaHuff(const char *arqEntrada, const char *arqSaida)
{
   tNoListaCanHuff *listaCanonica; /* Lista de códigos canônicos */
   tByte            buffer[MAX_BYTES]; /* Buffer que armazenará cada leitura */
                                       /* efetuada no arquivo de entrada     */
   int              i,
                    byte; /* Decodificação de um byte */
   FILE            *streamE, /* Stream associado ao arquivo a ser decodificado */
                   *streamS; /* Associado ao arquivo resultante */
   tCabecalhoHuff   cabecalho; /* Cabeçalho da codificação */
   tArvoreHuff      arvore; /* Raiz da árvore de codificação */

   streamE = AbreArquivo(arqEntrada, "rb"); /* Tenta abrir o arquivo de entrada */

   streamS = AbreArquivo(arqSaida, "wb"); /* Tenta abrir o arquivo de saída */

      /* Lê o cabeçalho no arquivo de entrada */
   listaCanonica = LeCabecalhoHuff(&cabecalho, streamE);

      /* Verifica se ocorreu erro na leitura do cabeçalho */
   if (!listaCanonica) {
         /* Fecha os arquivos e aborta o programa */
      FechaArquivo(streamE, arqEntrada);
      FechaArquivo(streamS, arqSaida);
      ASSEGURA(0, "Impossivel ler lista de codigos");
   }

      /* Constrói a árvore de codificação usando a lista de códigos canônicos */
   arvore = ReconstroiArvoreHuff(listaCanonica, cabecalho.tamLista);

      /* Decodifica o arquivo de entrada */
   while(1) {
         /* Lê um array de bytes no arquivo codificado. O tamanho desse array */
         /* é igual ao número máximo de bytes que pode constituir um código   */
      fread(buffer, sizeof(buffer), 1, streamE);

         /* Verifica se houve erro de leitura */
      ASSEGURA(!ferror(streamE), "Erro de leitura");

         /* Verifica se a leitura acabou */
      if (feof(streamE))
         break;
```

```
         /* Decodifica os bytes lidos e armazenados no buffer */
         /* e escreve o resultado no arquivo de saída         */
      for (i = 0; i < MAX_BITS; ++i) {
            /* Obtém um byte da decodificação do buffer */
         byte = ObtemByteHuff( arvore, buffer, MAX_BITS, &i );

            /* Se um byte foi obtido, escreve-o no arquivo de saída; caso */
            /* contrário, ele deverá ser completado na próxima leitura    */
         if (byte > 0)
            fputc(byte, streamS);
      }
   }

      /* Fecha arquivos */
   FechaArquivo(streamE, arqEntrada);
   FechaArquivo(streamS, arqSaida);

   free(listaCanonica); /* A lista de códigos canônicos não é mais necessária */
}
```

No site dedicado ao livro na internet (***http://www.ulysseso.com/ed2***), encontra-se a implementação completa do programa de codificação canônica de Huffman. Em particular, essa implementação mostra como é levado a efeito o **Passo 3.2** do algoritmo de decodificação descrito na **Figura 10–18**.

10.5.2 A Lanchonete HeapBurger

Problema: Desenvolva uma simulação de eventos discretos de funcionamento de uma lanchonete usando o conhecimento descrito na **Seção 10.4**. A lanchonete em questão não tem fila de atendimento nem de espera, de maneira que os clientes são atendidos em suas mesas. Essa simulação deve apresentar na tela um resultado como o seguinte:

```
>>> Este programa simula o funcionamento da lanchonete virtual HeapBurger

>>> 10:00       Abertura da lanchonete
                Numero de cadeiras disponiveis: 10
                Preco de um sanduiche BigHeap: R$8.00
>>> 12:00       Fechamento da lanchonete

>>> Numero de grupos que chegaram na lanchonete: 5
>>> Numero de clientes que chegaram na lanchonete: 14

>>> 10:08       4 fregueses chegam e sentam
>>> 10:08       Atendido pedido de 1 sanduiche
>>> 10:08       Atendido pedido de 3 sanduiches
>>> 10:08       Atendido pedido de 2 sanduiches
>>> 10:08       Atendido pedido de 3 sanduiches
>>> 10:09       Grupo de 4 pessoas sai
>>> 10:29       1 fregues chega e senta
>>> 10:30       Atendido pedido de 3 sanduiches
>>> 10:46       Grupo de 1 pessoa sai
>>> 11:46       3 fregueses chegam e sentam
>>> 11:51       Atendido pedido de 1 sanduiche
>>> 11:51       Atendido pedido de 1 sanduiche
>>> 11:51       Atendido pedido de 2 sanduiches
>>> 11:47       2 fregueses chegam e sentam
>>> 11:52       Atendido pedido de 3 sanduiches
>>> 11:52       Atendido pedido de 2 sanduiches
>>> 11:50       4 fregueses chegam e sentam
```

```
>>> 11:50       Atendido pedido de 1 sanduiche
>>> 11:50       Atendido pedido de 2 sanduiches
>>> 11:50       Atendido pedido de 2 sanduiches
>>> 11:50       Atendido pedido de 2 sanduiches
>>> 11:51       Grupo de 3 pessoas sai
>>> 11:55       Grupo de 4 pessoas sai
>>> 11:58       Grupo de 2 pessoas sai

>>> Faturamento da lanchonete: R$224.00
>>> Lucro da lanchonete: R$67.20

>>> Fim da simulacao
```

Solução: Para implementar a simulação solicitada, as seguintes constantes simbólicas serão usadas[4]:

```
#define N_CADEIRAS  10 /* Número de cadeiras da lanchonete */
#define MAX_PESSOAS_GRUPO 4 /* Número máximo de pessoas num grupo */
#define ATENDIMENTO 3.5 /* Tempo médio de atendimento */
#define MAX_SAND  3 /* Número máximo de sanduíches que uma pessoa pede */
#define PERMANENCIA 7 /* Tempo médio de permanência de um */
                      /* grupo de clientes na lanchonete  */
#define PRECO    8.00  /* Preço de um sanduíche */
#define MARGEM   0.30  /* Margem de lucro */
#define H_ABERTURA 10  /* Hora de abertura da lanchonete */
#define PERIODO   120 /* Período de funcionamento da lanchonete */
#define INTERVALO_CHEGADA 15 /* Intervalo de tempo médio de chegada de clientes */
```

As definições de tipos a seguir serão utilizadas nesta simulação:

```
  /* Tipo que define um tipo de evento */
typedef enum {CHEGADA, SAIDA, PEDIDO} tTipoDeEvento;

   /* Tipo de variável que contém informações sobre um evento */
typedef struct {
          int           instante; /* Instante no qual o evento será processado */
          int           nPessoas; /* Número de pessoas */
          tTipoDeEvento tipoDeEvento; /* Tipo do evento */
       } tEvento;

   /* Tipo de nó de um heap */
typedef tEvento *tNoHeap;

   /* Tipo da variável que conterá informações sobre a lanchonete */
typedef struct {
          int    nCadeiras;
          double faturamento;
       } tLanchonete;
```

Os tipos `tHeap` e `tFCompara` que representam, respectivamente, um heap e um ponteiro para função de comparação que compara elementos do heap, são definidos como na **Seção 10.2.4**. É importante notar ainda que um evento representado pela constante `PEDIDO` inclui três ações: (1) pedido de um sanduíche, (2) seu recebimento e (3) sua deglutição.

A função **main()** a seguir executa uma simulação da Lanchonete HeapBurger:

```
int main(void)
{
   int         t = 0, /* Instante de chegada de um grupo de clientes */
```

[4] Os valores dessas constantes foram simplesmente *chutados* pelo autor com o único propósito de tornar o exemplo mais palpável e não têm nenhum compromisso com a realidade.

```
            tMax = 0, /* Instante limite para chegada */
            nEventos = 0, /* Número de grupos de clientes */
                          /* que chegam à lanchonete */
            nPessoasGrupo, /* Número de pessoas num grupo de clientes */
            nPessoasTotal = 0; /* Número total de pessoas */
                               /* que chegam à lanchonete */
tHeap       fila; /* Fila de eventos */
tEvento    *evento; /* Ponteiro para um evento */
tLanchonete lanchonete = { N_CADEIRAS, /* Número de cadeiras */
                           0.0         /* Faturamento        */
                         };

      /* Apresenta o programa */
printf( "\n\t>>> Este programa simula o funcionamento da "
        "\n\t>>> lanchonete virtual HeapBurger\n\n" );

IniciaHeap(&fila); /* Inicia a fila de eventos */

   /* Exibe informações básicas sobre a lanchonete */
ApresentaHora(H_ABERTURA, 0);
printf( "\tAbertura da lanchonete");
printf( "\n\t\t\tNumero de cadeiras disponiveis: %d", N_CADEIRAS );
printf( "\n\t\t\tPreco de um sanduiche BigHeap: R$%.2f\n", PRECO );
ApresentaHora(H_ABERTURA, PERIODO);
printf( "\tFechamento da lanchonete\n");

   /* Cria alguns eventos de chegada e armazena-os na fila */
while (1) {
   t += IntervaloExponencial(INTERVALO_CHEGADA);

      /* Atualiza o tempo máximo de funcionamento */
   if (t > tMax)
      tMax = t;

      /* Verifica se o instante de ocorrência do próximo */
      /* evento será após o limite de funcionamento      */
   if (tMax > PERIODO)
      break;

      /* Obtém o número de pessoas no grupo do presente evento */
   nPessoasGrupo = NAleatorio(1, MAX_PESSOAS_GRUPO);

   nPessoasTotal += nPessoasGrupo; /* Atualiza o número total de pessoas */

   evento = CriaEvento(CHEGADA, t, nPessoasGrupo); /* Cria o evento de chegada */

      /* Insere o evento recém-criado na fila de eventos pendentes */
   InsereEmHeap(&fila, evento, ComparaEventos);

   ++nEventos; /* Atualiza o número total de eventos */
}
printf("\n\t>>> Numero de grupos que chegaram na lanchonete: %d", nEventos);
printf( "\n\t>>> Numero de clientes que chegaram na lanchonete: %d\n\n",
        nPessoasTotal );

   /* Executa a simulação */
while (!HeapVazio(&fila)) {
   evento = RemoveMin(&fila, ComparaEventos);
   DespachaEvento(evento, &lanchonete, &fila);
}

printf( "\n\t>>> Faturamento da lanchonete: R$%.2f", lanchonete.faturamento );
```

```
   printf( "\n\t>>> Lucro da lanchonete: R$%.2f\n", MARGEM*lanchonete.faturamento );
   printf("\n\t>>> Fim da simulacao\n\n");

   return 0;
}
```

Essa função **main()** inicia exibindo informações elementares (p. ex., o preço de um sanduíche) sobre a simulação. A função `ApresentaHora()`, chamada pela função **main()**, apresenta uma hora em formato amigável e sua implementação é relativamente trivial para ser discutida aqui.

Em seguida, a **main()** cria alguns eventos de chegada e armazena-os na fila de prioridade implementada como heap que armazena os eventos. Para criar esses eventos, a função `CriaEvento()`, definida a seguir, é chamada. Essa última função retorna o endereço do evento criado e tem como parâmetros:

- `tipoDeEvento` (entrada) — o tipo de evento
- `instante` (entrada) — o instante do evento
- `nPessoas` (entrada) — número de pessoas do evento

```
tEvento *CriaEvento( tTipoDeEvento tipoDeEvento, int instante, int nPessoas )
{
   tEvento *evento;

   evento = malloc(sizeof(tEvento));
   ASSEGURA(evento, "Impossivel alocar evento");

   evento->instante = instante;
   evento->nPessoas = nPessoas;
   evento->tipoDeEvento = tipoDeEvento;

   return evento;
}
```

O instante de ocorrência de cada evento de entrada gerado pela função **main()** é obtido por meio de uma chamada da função `IntervaloExponencial()` que retorna um intervalo de tempo aleatório usando uma distribuição exponencial (v. **Seção 10.4**) e é definida como:

```
double IntervaloExponencial(double intervaloMedio)
{
   return -intervaloMedio*log(1 - DRand());
}
```

O parâmetro único da função `IntervaloExponencial()` representa o intervalo de tempo médio decorrido entre duas chegadas de clientes. Essa função chama `DRand()` que retorna um número aleatório entre `0.0` e `1.0` e é implementada como:

```
double DRand(void)
{
   static int primeiraChamada = 1; /* Esta variável checa se a função está */
                                    /* sendo chamada  pela primeira vez e   */
                                    /* deve ter duração fixa                */

      /* Se esta for a primeira chamada da função, */
      /* alimenta o gerador de números aleatórios  */
   if (primeiraChamada) {
      srand(time(NULL)); /* Alimenta o gerador de números aleatórios */

      primeiraChamada = 0; /* A próxima chamada não será mais a primeira */
   }
   return (double)rand() / (double)RAND_MAX;
}
```

Para obter o número de pessoas num grupo de evento de chegada, a função **main()** chama a função `NAleatorio()` que retorna um número aleatório entre os valores recebidos como parâmetros e é definida como:

```
int NAleatorio(int m, int n)
{
   static int primeiraChamada = 1; /* Esta variável checa se a função está sendo   */
                                    /* chamada pela primeira vez e deve ter duração fixa */

      /* Se esta for a primeira chamada da função, */
      /* alimenta o gerador de números aleatórios  */
   if (primeiraChamada) {
      srand(time(NULL)); /* Alimenta o gerador de números aleatórios */

      primeiraChamada = 0; /* A próxima chamada não será mais a primeira */
   }

      /* Leva em consideração o fato de n poder ser menor do que m */
   if (n < m)
      return rand()%(m - n + 1) + n;

      /* m é menor do que ou igual a n */
   return rand()%(n - m + 1) + m;
}
```

Os eventos de chegada criados pela função **main()** são colocados na fila de prioridade pela função `InsereEmHeap()` descrita na **Seção 10.2.4**. O laço **while** da função **main()** implementa o laço de simulação visto na **Seção 10.4**. No corpo desse laço, um evento é removido da fila utilizando-se a função `RemoveMinHeap()`, definida na **Seção 10.2.4**, e, então, a função `DespachaEvento()` é chamada.

A função `DespachaEvento()` despacha um evento para a função que se dedica a processá-lo. Os parâmetros de `DespachaEvento()` são:

- `evento` (entrada e saída) — o evento que será despachado
- `lanchonete` (entrada e saída) — a lanchonete
- `heap` (entrada e saída) — a fila de eventos

```
void DespachaEvento( tEvento *evento, tLanchonete *lanchonete, tHeap *heap )
{
   tEvento *novoEvento;
   int      i;

      /* Processa o evento de acordo com seu tipo */
   switch (evento->tipoDeEvento) {
      case CHEGADA:
            /* Verifica se existem cadeiras para o grupo de clientes. Se houver   */
            /* disponibilidade, os clientes ficam; caso contrário, eles desistem. */
         if (ProcessaChegada(evento, lanchonete)) {
               /* Os clientes foram acomodados. Cria um novo   */
               /* evento de pedido e processa-o imediatamente. */
            novoEvento = CriaEvento( PEDIDO, evento->instante +
                                     IntervaloExponencial(ATENDIMENTO), evento->nPessoas );
            DespachaEvento(novoEvento, lanchonete, heap);
         }

         free(evento); /* Esse evento não é mais necessário */

         break;

      case SAIDA:
            /* Chama a função responsável por processamento de eventos de saída */
         ProcessaSaida(evento, lanchonete);
```

```
         free(evento); /* Esse evento não é mais necessário */

         break;

      case PEDIDO:
            /* Cada pessoa do grupo pede um certo número */
            /* de sanduíches (entre 1 e MAX_SAND)        */
         for (i = 0; i < evento->nPessoas; i++)
            ProcessaPedido( evento, NAleatorio(1, MAX_SAND), lanchonete );

            /* Cria um evento de saída e insere-o na fila */
         novoEvento = CriaEvento( SAIDA, evento->instante +
                                  IntervaloExponencial(PERMANENCIA), evento->nPessoas);
         InsereEmHeap(heap, novoEvento, ComparaEventos);

         free(evento); /* Esse evento não é mais necessário */

         break;

      default:
         printf("\nErro indeterminado no programa\n");
         exit(1);
   }
}
```

A função `ProcessaChegada()` processa um evento de chegada verificando se a lanchonete pode acomodar mais clientes. Essa função retorna `1`, se os clientes forem acomodados, ou `0`, se eles desistirem porque não há acomodação. Deve-se notar que a lanchonete só atende clientes sentados e não há fila de espera. Os parâmetros dessa função são:

- `evento` (entrada) — evento de chegada
- `lanchonete` (entrada e saída) — a lanchonete

```
int ProcessaChegada(const tEvento *evento, tLanchonete *lanchonete)
{
      /* Este deve ser um evento de chegada */
   ASSEGURA(evento->tipoDeEvento == CHEGADA, "O evento deveria ser de chegada");

      /* Descreve o evento na tela */
   ApresentaHora(H_ABERTURA, evento->instante);
   printf( "\t%d %s", evento->nPessoas,
           evento->nPessoas > 1 ? "fregueses chegam" : "fregues chega");

      /* Verifica se os novos clientes ficam ou desistem */
   if (evento->nPessoas < lanchonete->nCadeiras) {
         /* Apresenta na tela o que ocorre */
      printf( " e %s\n", evento->nPessoas > 1 ? "sentam" : "senta");

         /* O número de cadeiras disponíveis diminui */
      lanchonete->nCadeiras -= evento->nPessoas;
      return 1; /* Clientes ficaram */
   } else {
      printf( ".\n\t\t\tNao ha' espaco e %s embora.\n",
              evento->nPessoas > 1 ? "eles vao" : "ele vai" );

      return 0; /* Clientes foram embora */
   }
}
```

A função `ProcessaPedido()` atende o pedido de um grupo de clientes e tem como parâmetros:

- `evento` (entrada) — evento de pedido

- `nSanduiches` (entrada) — número de sanduíches que serão servidos
- `lanchonete` (entrada e saída) — a lanchonete

```
void ProcessaPedido(const tEvento *evento, int nSanduiches, tLanchonete *lanchonete)
{
      /* Este evento deve ser de pedido */
   ASSEGURA( evento->tipoDeEvento == PEDIDO, "O evento deveria ser de pedido" );

      /* Informa o que acontece */
   ApresentaHora(H_ABERTURA, evento->instante);
   printf( "\tAtendido pedido de %d %s\n", nSanduiches,
           nSanduiches > 1 ? "sanduiches" : "sanduiche" );

      /* Atualiza o faturamento da lanchonete */
   lanchonete->faturamento += PRECO*nSanduiches;
}
```

A função `ProcessaSaida()` processa um evento de saída e seus parâmetros são:

- `evento` (entrada) — evento de saída
- `lanchonete` (entrada e saída) — a lanchonete

```
void ProcessaSaida(const tEvento *evento, tLanchonete *lanchonete)
{
      /* Este evento deve ser de saída */
   ASSEGURA(evento->tipoDeEvento == SAIDA, "O evento deveria ser de saida");

      /* Informa o que acontece */
    ApresentaHora(H_ABERTURA, evento->instante);
    printf( "\tGrupo de %d %s sai\n", evento->nPessoas,
            evento->nPessoas > 1 ? "pessoas" : "pessoa" );

        /* O número de cadeiras disponíveis aumenta */
    lanchonete->nCadeiras += evento->nPessoas;
}
```

10.6 Exercícios de Revisão

Filas de Prioridade (Seção 10.1)

1. (a) O que é uma fila de prioridade? (b) Apresente exemplos cotidianos de filas de prioridade. (c) Apresente exemplos computacionais de filas de prioridade.
2. Descreva os seguintes modos de implementação de filas de prioridade:
   (a) Lista encadeada sem ordenação
   (b) Lista encadeada ordenada
   (c) Lista indexada sem ordenação
   (d) Lista indexada ordenada
   (e) Árvore binária de busca balanceada
   (f) Árvore binária de busca sem balanceamento
3. Suponha que você precisa implementar uma lista de prioridade na qual haverá um enorme número de inserções e muito poucas remoções. Que tipo de implementação você usaria?

Heaps Binários (Seção 10.2)

4. (a) Descreva a propriedade de ordenação de heaps binários. (b) Descreva a propriedade estrutural de heaps binários.
5. Descreva os seguintes conceitos relacionados a heaps binários:

(a) Heap de máximo
(b) Heap de mínimo
(c) Heap ascendente
(d) Heap descendente

6. Por que um heap descendente recebe essa denominação?

7. Descreva o esquema de numeração de nós utilizado na representação de árvores binárias utilizando arrays.

8. Usando a representação de heaps por meio de arrays apresentada no texto, como se encontram os seguintes nós:

(a) O pai de um nó de índice *i*
(b) O filho direito de um nó de índice *i*
(c) O filho esquerdo de um nó de índice i

9. Um esquema de numeração de nós de árvores binárias diferente daquele usado neste capítulo atribui índices aos nós a partir de *1*, em vez de *0*, de modo que, quando um heap é implementado numa linguagem que faz indexação de arrays a partir de zero (como C, por exemplo), o primeiro elemento do array não é usado. Apresente uma vantagem do uso desse esquema de numeração.

10. Usando a representação de heaps por meio de arrays apresentada na questão anterior, como se encontram os seguintes nós:

(a) O pai de um nó de índice *i*
(b) O filho direito de um nó de índice *i*
(c) O filho esquerdo de um nó de índice *i*

11. (a) A abordagem de representação de árvores binárias por meio de arrays pode ser usada com árvores binárias que não são completas? (b) Se sua resposta for afirmativa, como é possível representar uma árvore binária que não é completa usando um array?

12. Mostre graficamente o resultado da construção de um heap ascendente quando são inseridos nós com os seguintes valores *13*, *15*, *4*, *17*, *11*, *18*, *6*, *12*, *11*, *14*, e *16*. Suponha que o heap está inicialmente vazio e que os valores são inseridos na ordem em que se encontram.

13. Mostre graficamente o heap resultante de três remoções no heap da questão anterior.

14. Mostre graficamente o resultado da construção de um heap descendente quando são inseridos nós com os seguintes valores *13*, *15*, *4*, *17*, *11*, *18*, *6*, *12*, *11*, *14*, e *16*. Suponha que o heap está inicialmente vazio e que os valores são inseridos na ordem em que se encontram.

15. (a) Se os valores da questão anterior forem inseridos na ordem inversa, o formato do heap será diferente? (b) Nesse caso, os posicionamentos desses valores serão diferentes?

16. Mostre graficamente o heap resultante de três remoções no heap da questão **14**.

17. (a) Qual é o nó que contém o segundo maior valor de um heap de máximo? (b) Em que nível se encontra o terceiro maior valor de um heap de máximo?

18. Desenhe todos os heaps de mínimo diferentes que podem ser construídos de maneira que os conteúdos dos nós sejam as letras *A*, *B*, *C*, *D* e *E* considerando a ordem alfabética usual.

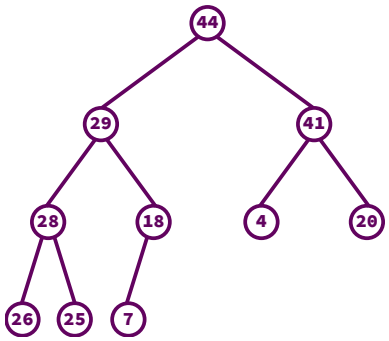
19. (a) Descreva o algoritmo de percolação ascendente. (b) Em que situação ele é aplicado?

20. (a) Descreva o algoritmo de percolação descendente. (b) Em que situação ele é aplicado?

21. (a) Quando é necessário inserir um novo item num heap ascendente, que tipo de percolação é usado? (b) Quando é necessário inserir um novo elemento num heap descendente, que tipo de percolação é usado?

22. Quais são as alterações que devem ser efetuadas nos algoritmos que descrevem as operações de inserção e remoção sobre heap de mínimo apresentadas neste capítulo para implementar um heap de máximo?

23. Uma fila de prioridade é implementada como heap descendente conforme mostra a figura a seguir:



Apresente uma representação gráfica desse heap depois da seguinte sequência de operações:

(i) Enfileiramento de *27*
(ii) Enfileiramento de *2*
(iii) Enfileiramento de *39*
(iv) Desenfileiramento
(v) Desenfileiramento
(vi) Desenfileiramento

24. Mostre que um heap que armazena n chaves tem profundidade (altura) dada por $p = \lfloor log_2\, n + 1 \rfloor$.

25. Mostre que existem exatamente $\lceil n/2 \rceil$ folhas num heap contendo n elementos.

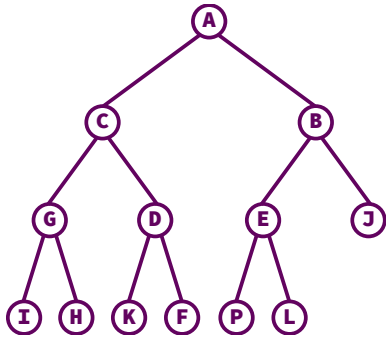
26. Em qual nó de um heap ascendente pode ser encontrado o maior valor armazenado nesse heap?

27. Uma fila de prioridade contendo caracteres é implementada como um heap armazenado num array. Suponha que essa fila de prioridade não contém elementos duplicados. Correntemente, a fila de prioridade contém 10 elementos, como mostrado na figura abaixo. Quais são as letras que podem ser armazenadas nas posições 7, 8 e 9 do array de modo que as propriedades de heap sejam satisfeitas?

0	Z
1	F
2	J
3	E
4	B
5	G
6	H
7	?
8	?
9	?

28. Visitar um heap em ordem prefixa, infixa ou sufixa tem algum significado especial?

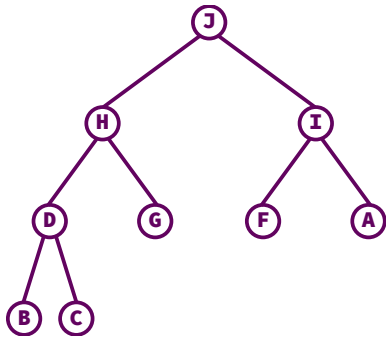
29. Apresente um array que armazene os nós do heap da figura a seguir de acordo com o esquema de numeração de nós visto na **Seção 10.2**.



30. Mostre todos os passos envolvidos na remoção do nó contendo *A* no heap da figura do **Exercício 29**.

31. Mostre todos os passos envolvidos na inserção do nó contendo *M* no heap da figura do **Exercício 29**.

32. Apresente um array que armazene os nós do heap da figura a seguir de acordo com o esquema de numeração de nós visto na **Seção 10.2**.



33. Mostre todos os passos envolvidos na remoção do nó contendo *J* no heap da figura do **Exercício 32**.

34. Mostre todos os passos envolvidos na inserção do nó contendo *M* no heap da figura do **Exercício 32**.

35. (a) Em que consiste uma operação de acréscimo de prioridade? (b) Apresente uma situação prática na qual essa operação é útil. (c) Como uma operação de acréscimo de prioridade pode ser implementada num heap ascendente?

36. (a) Em que consiste uma operação de decréscimo de prioridade? (b) Apresente uma situação prática na qual essa operação é útil. (c) Como uma operação de decréscimo de prioridade pode ser implementada num heap ascendente?

37. (a) Em que consiste uma operação de remoção de elemento específico? (b) Apresente uma situação prática na qual essa operação é útil. (c) Como uma operação de remoção de elemento específico pode ser implementada num heap ascendente?

38. (a) Um array ordenado em ordem crescente constitui um heap? (b) Se for o caso, será um heap ascendente ou descendente?

39. O array `{21, 14, 11, 3, 10, 7, 1, 2, 9}` constitui um heap de máximo?

40. Suponha que um heap tenha profundidade p. Mostre que existem 2^i nós com profundidade $i < p$ num heap com n elementos, desde que n seja uma potência de 2.

41. (a) Qual é o número mínimo de elementos num heap de altura a? (b) Qual é o número máximo de elementos nesse heap?

42. Mostre que a altura de um heap contendo n elementos é $\lfloor \log n \rfloor$.

43. Mostre que, num heap de mínimo, a raiz de qualquer subárvore contém a chave de menor valor dessa subárvore.

Análise de Filas de Prioridade (Seção 10.3)

44. Preencha a tabela a seguir com os custos temporais de enfileiramento e desenfileiramento de cada abordagem de implementação de lista de prioridade.

Implementação via...	Custo Temporal de...	
	Inserção	Remoção
Lista encadeada sem ordenação		
Lista encadeada ordenada		
Lista indexada sem ordenação		
Lista indexada ordenada		
Árvore binária de busca balanceada		
Árvore binária de busca sem balanceamento		

45. Uma fila de prioridade pode ser implementada de tal modo que tanto enfileiramento quanto desenfileiramento tenham custo temporal $\theta(1)$?

46. Quais são as vantagens da implementação de fila de prioridade por meio de heap em detrimento a implementação de fila de prioridade usando árvore AVL?

Simulação de Eventos Discretos (Seção 10.4)

47. (a) O que é um evento discreto? (b) O que é um evento contínuo?

48. Descreva o laço principal de uma simulação de eventos discretos.

49. Quais são os principais componentes de um programa de simulação de eventos discretos?

50. (a) O que é um despachante de eventos? (b) O que é um processador de eventos?

51. Qual é o papel desempenhado por distribuição exponencial em simulação de eventos discretos?

Exemplos de Programação (Seção 10.5)

52. Descreva o algoritmo de codificação padrão de Huffman.

53. Por que a árvore resultante de uma codificação de Huffman é estritamente binária?

54. Mostre que se o número de caracteres codificados for igual a n, a árvore de codificação de Huffman conterá n folhas e $n - 1$ nós internos.

55. Que vantagens apresenta a codificação canônica com relação à codificação padrão de Huffman?

56. Qual é a profundidade máxima de uma árvore de codificação de Huffman usada na compressão de um arquivo contendo informações sobre DNA que usa o alfabeto genético $\sum = \{A, C, N, T\}$?

57. Suponha que num arquivo de texto contendo informações sobre DNA os símbolos do alfabeto genético apresentem as seguintes frequências:

Símbolo	Frequência (%)
A	18.5
C	23.8
N	44.0
T	13.7

Quais serão os códigos canônicos atribuídos a cada um dos símbolos desse arquivo quando o algoritmo apresentado na **Figura 10–16** for seguido?

58. (a) Por que o buffer que armazena a codificação de um byte é dimensionado com **UCHAR_MAX + 1** elementos do tipo de **char**? (b) Por que raramente esse buffer contém a codificação de apenas um byte?

59. Por que a profundidade máxima de uma árvore de codificação de Huffman é igual ao número de bytes distintos do arquivo que está sendo codificado?

60. (a) Como se determina número de folhas da árvore de codificação de Huffman? (b) Qual é a importância prática desse cálculo?

61. Por que o número máximo de bits na codificação de um byte é igual ao número de folhas da árvore de codificação de Huffman?

62. (a) O que é cabeçalho de uma codificação de Huffman? (b) Quais são as informações mínimas armazenadas no cabeçalho de uma codificação canônica de Huffman? (c) O que está implícito nessas informações?

63. Quando um arquivo comprimido usando o algoritmo de Huffman apresenta sua menor taxa de compressão?

64. O que é uma codificação livre de prefixo?

65. Um estudante desatento estranhou o fato de o código atribuído a um byte na codificação de Huffman poder usar uma sequência de até 256 bits. De acordo com o raciocínio desse estudante, se o objetivo final dessa codificação é comprimir um arquivo, não faz sentido utilizar sequências desse tamanho, pois, num arquivo sem codificação, um byte usa uma sequência de apenas 8 bits. Como você clarificaria o raciocínio tortuoso desse estudante?

10.7 Exercícios de Programação

EP10.1 (a) Implemente uma fila de prioridade como uma lista indexada de modo que o item com maior prioridade encontra-se na primeira posição da lista, o item com a segunda maior prioridade encontra-se na segunda posição da lista e assim por diante. (b) Compare em termos de notação ó essa implementação com aquela que usa heap.

EP10.2 Implemente uma fila de prioridade como uma lista encadeada ordenada.

EP10.3 Implemente fila de prioridade como lista encadeada sem ordenação.

EP10.4 Implemente uma fila de prioridade como uma lista indexada ordenada.

EP10.5 Implemente uma fila de prioridade como uma lista indexada sem ordenação.

EP10.6 Implemente uma fila de prioridade como uma árvore AVL.

EP10.7 Implemente uma versão iterativa da função `OrdenaHeap()` apresentada na **Seção 10.2.4**.

EP10.8 Implemente uma nova versão do programa de simulação de lanchonete apresentada na **Seção 10.5.2** de tal modo que as constantes simbólicas sejam substituídas por variáveis cujos valores são introduzidos pelo usuário. Por exemplo, em vez de usar a constante `N_CADEIRAS`, o programa deve solicitar que o usuário introduza um valor com a mesma interpretação que será armazenada numa variável denominada `nCadeiras`.

EP10.9 Apresente uma implementação de heap ascendente usando nós e ponteiros dos tipos definidos abaixo:

```
typedef struct no {
           int        valor;    /* Valor do nó    */
           struct no *esquerda; /* Filho direito  */
           struct no *direita;  /* Filho esquerdo */
           struct no *pai;      /* Pai do nó      */
        } tNo, *tHeap;
```

EP10.10 Implemente uma função que verifica se um array de elementos do tipo **int** representa um heap ascendente.

EP10.11 Escreva uma função que verifique se uma árvore binária que usa os tipos definidos abaixo é um heap.

```
typedef struct rotNoArvore {
           struct no *esquerda;
           int        conteudo;
           struct no *direita;
        } tNoArvore, *tArvore;
```

EP10.12 Altere o programa apresentado na **Seção 10.5.2** de modo que ele apresente o número de sanduíches vendidos pela lanchonete.

EP10.13 Uma das vantagens da codificação canônica é que ela requer que apenas os tamanhos dos códigos sejam armazenados no cabeçalho do arquivo codificado. No entanto, a função `EscreveCabecalho()` apresentada na **Seção 10.5.1** escreve todo o conteúdo da lista de códigos canônicos, incluindo os códigos atribuídos aos bytes do arquivo original. (a) Reimplemente a referida função de tal modo que ela escreva apenas os apenas os tamanhos dos códigos nesse cabeçalho. (b) Reimplemente a função `Decodifica()` para que ela seja readaptada para essa nova situação.