Busca Linear em Memória Principal

objetivos

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar a seguinte terminologia:
  - ❒ Registro
  - ❒ Chave
  - ❒ Chave interna
  - ❒ Chave externa
  - ❒ Chave primária
  - ❒ Chave secundária
  - ❒ Teto e piso de chave
  - ❒ Busca de intervalo
  - ❒ Tabela de busca
  - ❒ Algoritmo de busca
  - ❒ Algoritmo de inserção
  - ❒ Algoritmo de remoção
  - ❒ Dicionário
  - ❒ Lista com saltos
  - ❒ Buscas interna e externa
  - ❒ Programa-cliente
  - ❒ Busca sequencial
  - ❒ Busca binária
  - ❒ Busca por interpolação
  - ❒ Tabela de busca indexada
  - ❒ Tabela de busca encadeada
- Implementar operações de busca, inserção e remoção em tabelas de busca lineares
- Analisar custos temporal e espacial de operações em tabelas de busca lineares
- Discutir uma possível ocorrência de overflow em implementação de busca binária
- Definir o conceito de estrutura de dados probabilística
- Explicar como se simula lançamento de moeda em programação
- Especificar e implementar uma lista com saltos real

UM DOS PROBLEMAS básicos de programação consiste em encontrar informação que tenha sido armazenada em algum lugar. Com o uso cada vez mais disseminado da internet, busca é uma das atividades mais frequentes do cotidiano. Quando você utiliza um mecanismo, tal como Google, para procurar algum documento ou imagem na internet você está, de fato, efetuando uma operação de busca. Busca também é uma tarefa fundamental em qualquer sistema de banco de dados.

Informalmente, uma operação de busca consiste em tentar encontrar um objeto que faz parte de uma coleção a partir de alguma informação parcial sobre o objeto desejado. Mais precisamente, em programação, **busca** refere-se à atividade de procurar alguma informação num conjunto de dados a partir de dados incompletos relacionados com a informação procurada. Por exemplo, tentar encontrar os dados completos de um contribuinte de imposto de renda a partir de seu CPF é uma atividade de busca. Este capítulo se concentra numa categoria de busca em **memória principal**[1] denominada **busca linear**.

3.1 Definições Fundamentais

Esta seção apresenta definições fundamentais relacionadas com busca de dados. Tais conceitos lhe ajudarão a entender o conteúdo abordado neste capítulo e serão essenciais para o entendimento de qualquer tópico relacionado a busca de dados que você estude nos capítulos posteriores.

Um **registro** é uma coleção de dados relacionados entre si e agrupados numa unidade. Por exemplo, um registro de aluno de uma universidade contém um conjunto de informações sobre cada aluno, tais como nome, matrícula, endereço, telefone etc. Essas informações sobre cada aluno constituem um registro. Por sua vez, uma coleção de registros é denominada **tabela** (ou **arquivo**). Por exemplo, o conjunto de todos os registros dos alunos de uma universidade constitui uma tabela. Quando uma tabela é utilizada em operações de busca, inserção e remoção de registros, ela também é denominada **tabela de busca**.

Uma **chave** é um componente de um registro utilizado numa operação de busca. Por exemplo, matrícula ou nome de aluno pode ser utilizada como chave de busca numa tabela contendo registros de alunos de uma universidade. Quando cada valor de chave é único num conjunto de registros (i.e., tabela), diz-se que essa chave é **primária**. Por exemplo, no conjunto de registros de contribuintes da Receita Federal, CPF é considerado uma chave primária, pois não existem dois contribuintes com o mesmo número de CPF. Por outro lado, quando uma chave pode ter valores repetidos numa mesma tabela, essa chave é considerada **secundária**. Por exemplo, é muito provável que dois ou mais contribuintes no conjunto de registros da Receita Federal tenham o mesmo nome. Portanto nome de contribuinte é uma chave secundária. Uma tabela de busca que permite buscas com chaves secundárias é denominada **dicionário**, pois duas ou mais entradas distintas podem possuir a mesma chave, como ocorre num dicionário de língua portuguesa, por exemplo. A grande maioria dos exemplos deste livro não lida com chave secundária, mas pode ser estendida para acomodá-las (v. **Seção 3.7.2**).

Definida mais precisamente, tabela de busca é um conjunto de **pares chave/valor**, de modo que o valor pode ser obtido especificando-se a chave associada a ele. Cada par chave/valor de uma tabela de busca é um **elemento** dessa tabela. Uma tabela de busca permite que chaves e valores sejam de quaisquer tipos. Tipicamente, o valor associado a uma chave numa tabela de busca é o próprio registro associado à chave ou é a posição na qual esse registro se encontra num arquivo de dados. No primeiro caso, a tabela de busca é análoga a um arquivo de escritório tradicional (i.e., não informatizado), como mostra a **Figura 3–1**. Nessa analogia, a chave é aquilo que se encontra impresso na aba de cada pasta (de papel), o valor é o próprio conteúdo da pasta e a tabela é o armário que armazena essas pastas. Um elemento nessa analogia é uma pasta do arquivo.

[1] A denominação mais precisa para o local onde as estruturas de dados estudadas neste capítulo são armazenadas seria *memória interna*, de acordo com o que foi visto na **Seção 1.3**. Mas, no contexto de Estruturas de Dados, termo mais comum é mesmo *memória principal*.

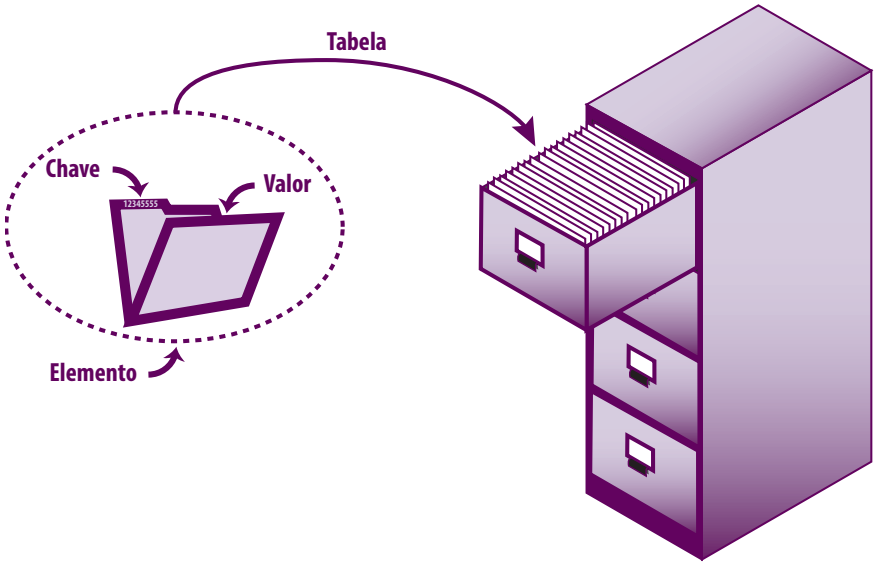


Figura 3–1: Analogia entre Tabela de Busca e Arquivo de Escritório

O objetivo de uma busca é encontrar os registros com chaves iguais a uma certa chave de busca. Ou seja, o propósito de uma busca é usualmente acessar informação dentro do item (não meramente a chave) para processamento. Um **algoritmo de busca** tenta encontrar um registro numa tabela de busca cuja chave coincida com o valor recebido como entrada pelo algoritmo. Esse valor de entrada deve ser do mesmo tipo de cada chave da tabela e é chamado **chave de busca**. Ademais, quando ocorre uma coincidência entre a chave de busca e uma chave que se encontra na tabela, diz-se que houve um **casamento** entre essas chaves (ou que as chaves **casam**).

Uma busca é **bem-sucedida** (ou **obtém êxito**) quando ocorre um casamento entre a chave de busca e uma chave na tabela que permita encontrar o registro procurado. Nesse caso, diz-se que o respectivo registro foi recuperado (i.e., ocorreu uma **recuperação de informação**) e o algoritmo de busca deve retornar uma indicação do local onde se encontra o referido registro ou o próprio registro. Quando não é encontrado nenhum registro que casa com a chave de busca, o algoritmo de busca deve retornar um valor indicando esse fato. Nesse último caso, diz-se que a busca foi **malsucedida** (ou que ela **não obteve êxito**).

Quando o valor associado a cada chave numa tabela de busca é o próprio registro, diz-se que essa é uma **chave interna**. Por outro lado, quando o valor associado a cada chave numa tabela de busca é um indicador de um local em outra tabela ou num arquivo de dados que contém os respectivos registros, diz-se que essa chave é **externa**. Nesse último caso, a recuperação de um registro se dá em dois passos:

1. A busca é efetuada na tabela contendo as chaves e retorna o endereço ou o índice do respectivo registro.
2. Usando-se o endereço ou índice retornado no **Passo 1**, o registro é acessado.

Numa **busca interna**, a tabela de busca se encontra totalmente armazenada em memória principal. Nesse caso, a eficiência de um algoritmo de busca é medida como foi visto no **Capítulo 6** do **Volume 1** (i.e., em termos de uso de tempo e espaço adicional estimado usando análise assintótica). É importante salientar que, nesse caso, uma busca é considerada interna mesmo que os registros a ser recuperados estejam armazenados em memória externa, desde que as chaves que permitem recuperar os registros estejam todas armazenadas em memória principal. A **Figura 3–2** ilustra uma tabela de busca com chaves internas, enquanto a **Figura 3–3** exemplifica uma tabela de busca com chaves externas.

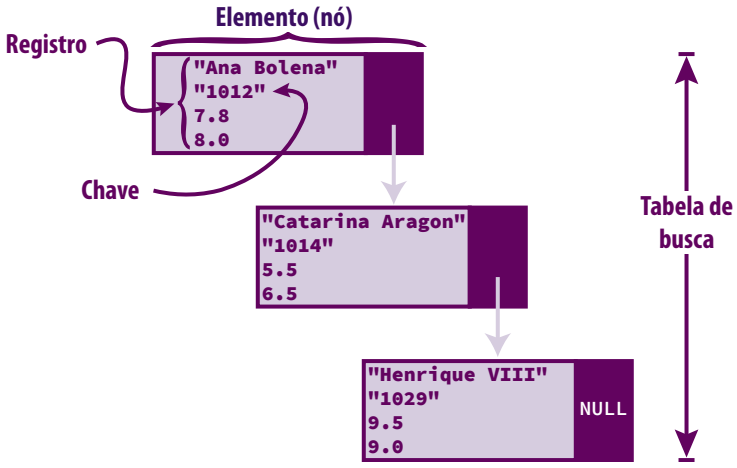


FIGURA 3–2: TABELA DE BUSCA COM CHAVES INTERNAS

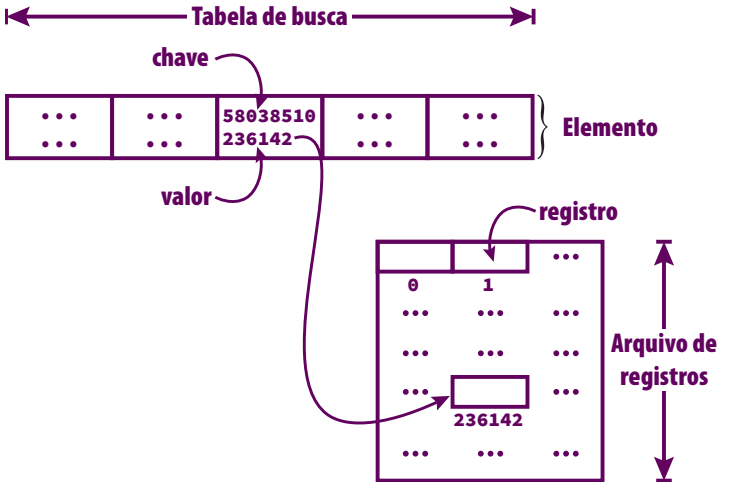


FIGURA 3–3: TABELA DE BUSCA COM CHAVES EXTERNAS

Tabelas de busca podem ser enormes, de modo que, nesse caso, elas precisam ser mantidas em memória secundária. Numa **busca externa**, a tabela de busca encontra-se totalmente armazenada num meio de armazenamento externo (p. ex., disco rígido). Nesse caso, um algoritmo de busca eficiente deve minimizar o número de acessos ao meio de armazenamento externo. Quando tabelas de busca são armazenadas em memória secundária, elas também são frequentemente chamadas **índices**.

Existem alguns tipos de buscas que são específicas para determinadas aplicações, como, por exemplo:

- **Busca de intervalo** (v. **Seção 3.7.4**), que tenta recuperar ou contar todos os registros que se encontram no intervalo entre dois valores limites de chaves.
- **Busca dedilhada**, que começa do ponto em que uma busca anterior terminou.
- **Busca de piso** (v. **Seção 3.7.3**), que tenta encontrar a maior chave que é menor do que a chave recebida como entrada. Se a referida chave encontrar-se na tabela, ela é seu próprio piso.
- **Busca de teto**, que tenta encontrar a menor chave que é maior do que a chave recebida como entrada. Se a referida chave encontrar-se na tabela, ela é seu próprio teto.
- **Busca de menor chave**, que encontra a menor chave armazenada numa tabela de busca. Esse tipo de busca só não obtém êxito quando a tabela de busca está vazia.

- **Busca de maior chave**, que encontra a maior chave armazenada numa tabela de busca. Esse tipo de busca só não obtém êxito quando a tabela de busca está vazia.

Existem várias maneiras de organizar dados de modo a tornar a busca mais eficiente. **Organização de tabela** refere-se às estruturas de dados usadas na implementação de uma tabela de busca. Tabelas de busca podem ser implementadas utilizando-se diversas estruturas de dados, tais como listas, árvores ou por meio de dispersão, como será visto nos próximos capítulos deste livro. Tipicamente, a organização de uma tabela reflete a técnica de busca a ser usada e vice-versa.

Além de busca, uma implementação de tabela de busca deve prover pelo menos duas outras operações: (1) inserção e (2) remoção. Um **algoritmo de inserção** é um algoritmo que tenta inserir numa tabela uma chave e seu respectivo valor recebidos como entrada. Existem duas abordagens básicas para implementação de um algoritmo de inserção quando a chave é considerada primária:

1. Se a chave primária for encontrada, o valor associado a ela é substituído.
2. Se a chave primária for encontrada, não há inserção. Nesse caso, o algoritmo de inserção deve retornar um valor que indique que não houve inserção. Essa é a abordagem adotada por este livro.

Em qualquer caso, se a chave não for encontrada, um novo elemento é inserido na tabela de busca, assim como ocorre quando a chave é secundária.

O fato de não haver espaço na tabela de busca para conter um novo elemento (ou, equivalentemente, não ser possível alocar espaço adicional) é considerado uma condição de exceção, que é tratada com o uso de **asserção**, como foi discutido no **Capítulo 7** do **Volume 1**.

Um **algoritmo de remoção** remove de uma tabela de busca o elemento que corresponde a uma chave recebida como entrada pelo algoritmo. Obviamente, tal algoritmo só obtém êxito se a referida chave for encontrada, de modo que ele deve retornar um valor indicando se foi bem-sucedido ou não. Quando as chaves de uma tabela de busca são secundárias, existem duas abordagens possíveis para remoção:

1. Remover o primeiro elemento da tabela cuja chave case com a chave de busca.
2. Remover todos os elementos da tabela cujas chaves casem com a chave de busca.

Diferentes implementações de operações da tabela de busca diferem em eficiência de uso de tempo e espaço, que pode depender da mescla de operações usadas por um **programa-cliente** (ou **aplicativo**). Por exemplo, um aplicativo pode usar inserção com pouca frequência e efetuar um grande número de operações de busca. Por sua vez, outro aplicativo pode usar operações de inserção e remoção um enorme número de vezes mescladas com algumas operações de busca. Assim uma implementação de tabela de busca pode prover suporte eficiente para certas operações à custa de outras, assumindo que as operações mais ineficientes serão executadas raramente.

Como se faz com muitas estruturas de dados, pode-se também precisar acrescentar a esse conjunto básico de operações outras para iniciar, testar se está vazia, destruir e copiar. Neste livro, serão consideradas implementações das funções fundamentais de iniciação, inserção, remoção, busca e destruição (para tabelas de busca alocadas dinamicamente).

3.2 Programas-Clientes

Os **programas-clientes** a serem usados implementam uma interação dirigida por menu simples que oferece ao usuário as três operações básicas sobre tabelas de busca: (1) busca, (2) inserção e (3) remoção. Além dessas opções, cada programa oferece ao usuário (obviamente) a opção de encerramento do programa. Em resumo, cada programa-cliente segue o algoritmo da **Figura 3–4**.

1. Abra o arquivo de dados para leitura e escrita
2. Crie a tabela de busca
3. Enquanto o usuário não escolher a opção de encerramento faça
   - 3.1 Apresente o menu de opções
   - 3.2 Leia a opção do usuário
   - 3.3 Execute a operação escolhida pelo usuário
   - 3.4 Apresente o resultado dessa operação
4. Se o arquivo de dados foi alterado, atualize-o
5. Feche o arquivo de dados
6. Encerre o programa

Figura 3–4: Algoritmo Seguido por um Programa Cliente

O **Passo 2** do algoritmo delineado na **Figura 3–4** implica que, por conveniência, deve haver uma operação responsável pela criação da tabela de busca usada pelo aplicativo. Essa operação consiste em ler um arquivo contendo os dados necessários à construção da tabela e invocar a operação de inserção para inseri-los na tabela. Parece razoável supor que essa seja uma operação executada apenas uma vez durante a execução do programa e não deve fazer parte do menu de opções oferecidas ao usuário, a não ser que o propósito do programa-cliente seja depuração.

Basicamente, o que muda de um programa aplicativo de busca apresentado neste livro para outro é como as operações básicas são executadas. Ademais, alguns aplicativos oferecem operações adicionais que têm como propósito demonstrar ou testar algumas características específicas de alguma implementação de tabela de busca.

Todos os aplicativos incluem um módulo constituído pelos arquivos `Registros.h` e `Registros.c` que implementa operações básicas sobre os registros (do tipo `tNoCaminhoB`) usados pelo aplicativo. Essas operações incluem as funções brevemente descritas na **Tabela 3–1**.

Protótipo da Função	O Que Ela Faz
`void ExibeRegistro(tRegistro *registro)`	*Exibe um registro na tela*
`tRegistro *LeRegistro(tRegistro *regCEP)`	*Lê um registro no arquivo de dados*
`void SubstituiRegistro( tRegistro *novo, tRegistro *antigo )`	*Substitui um registro por outro no arquivo*
`int TamanhoDeArquivo(FILE *stream)`	*Retorna o número de bytes do arquivo*
`int NumeroDeRegistros( FILE *stream, int tamRegistro )`	*Retorna o número de registros do arquivo*

Tabela 3–1: Funções de Processamento de Registros

Essas funções de processamento de registros são relativamente fáceis de implementar e fogem dos tópicos centrais deste livro, de modo que elas não receberão maiores considerações no presente texto.

É importante lembrar que todas as implementações de tabelas de busca acompanhadas de seus programas-clientes podem ser obtidas no site dedicado a este livro na internet.

3.3 Busca Sequencial Simples

3.3.1 Conceitos

Busca sequencial é a forma mais simples de busca e aplica-se a tabelas organizadas de modo linear (p. ex., quando a tabela de busca é implementada por meio de lista indexada ou encadeada). Nesse tipo de busca, cada chave, a partir do início da tabela, é comparada à chave de busca e a busca continua até que seja encontrada uma chave que casa com a chave de busca ou até que o final da tabela de busca seja atingido. Formalmente, uma operação de busca sequencial segue o algoritmo apresentado na **Figura 3–5**.

ALGORITMO BUSCASEQUENCIAL

ENTRADA: Uma tabela indexada ou encadeada com *n* elementos e uma chave de busca

SAÍDA: O valor associado à primeira chave da tabela que casa com a chave de busca ou um valor indicando que a chave não foi encontrada

1. Considere como elemento corrente o primeiro elemento da tabela
2. Enquanto o último elemento da tabela não tiver sido visitado, faça o seguinte:
   - 2.1 Se a chave do elemento corrente casar com a chave de busca, retorne o valor associado à chave desse elemento
   - 2.2 Considere como elemento corrente o próximo elemento da tabela
3. Retorne um valor que indique que a chave de busca não foi encontrada

FIGURA 3–5: ALGORITMO DE BUSCA SEQUENCIAL

Busca sequencial é a única forma de encontrar algum registro numa tabela não ordenada na qual os registros são organizados aleatoriamente de acordo com a chave de busca. Mesmo que os registros sejam ordenados de acordo com um campo que não seja a chave de busca, eles serão considerados desordenados do ponto de vista das suas chaves. Se estiverem ordenados de acordo com a chave de busca, existem meios mais eficientes de busca do que a busca sequencial, conforme será visto adiante.

3.3.2 Implementação

A tabela de busca a ser apresentada nesta seção, denominada **tabela de busca indexada**, é implementada usando lista indexada, que, por sua vez, é implementada usando array dinâmico. Além disso, as chaves são armazenadas na tabela juntamente com as posições em arquivo de seus respectivos registros (i.e., as chaves são externas). A justificativa para tal decisão é que o arquivo de dados a ser utilizado, `CEPs.bin` (v. **Apêndice A**), é suficientemente grande para não ser mantido em memória principal.

Definições de Tipos

A tabela de busca será implementada como TAD, de modo que a definição desse tipo é escrita em duas partes:

1. No arquivo de cabeçalho (`TabelaIdx.h`):

```
typedef char tCEP[TAM_CEP + 1]; /* Tipo de chave */

   /* Tipo de conteúdo de um elemento */
typedef struct {
           tCEP chave; /* CEP */
           int  valor; /* Índice do CEP no arquivo de registos de CEPs */
       } tCEP_Ind;

typedef struct rotTabelaIdx *tTabelaIdx;
```

2. No arquivo de programa (`TabelaIdx.c`):

```
struct rotTabelaIdx {
   tCEP_Ind *elementos;  /* Ponteiro para o array que contém os elementos */
   int       nElementos; /* Número de elementos */
   int       tamanhoArray;
};
```

Criação e Destruição

A função `tNoCaminhoB` cria uma tabela de busca vazia e retorna seu endereço.

```
tTabelaIdx CriaTabelaIdx(void)
{
   tTabelaIdx tabela;

      /* Aloca espaço para a estrutura que armazena a tabela */
   tabela = malloc(sizeof(struct rotTabelaIdx));

      /* Garante que a alocação realmente ocorreu */
   ASSEGURA(tabela, "Nao foi possivel alocar tabela");

      /* Aloca previamente um array para um elemento */
   tabela->elementos = calloc(1, sizeof(tCEP_Ind));

      /* Garante que o array foi realmente alocado */
   ASSEGURA( tabela->elementos, "Nao foi possivel alocar "
             "array para conter elementos da tabela" );

      /* Atualiza a variável que armazena o tamanho do array */
   tabela->tamanhoArray = 1;

   tabela->nElementos = 0; /* Inicialmente a tabela está vazia */

   return tabela;
}
```

Note que o array que armazenará a tabela é alocado pela função `CriaTabelaIdx()` com apenas um elemento, o que não parece ser razoável visto que a tabela deverá conter um número muito grande de elementos. A razão para tal decisão deverá ser bem compreendida quando o desempenho da tabela for analisado usando análise amortizada no **Capítulo 5**.

A função `tNoCaminhoB` libera o espaço ocupado pela tabela de busca recebida como parâmetro.

```
void DestroiTabelaIdx(tTabelaIdx tabela)
{
   free(tabela->elementos); /* Libera o array que contém os elementos */
   free(tabela); /* Libera a própria tabela */
}
```

Busca

A função `BuscaSequencialIdx()`, apresentada a seguir, efetua uma busca sequencial numa tabela de busca do tipo proposto nesta seção. Os parâmetros desta função são:

- `tabela` (entrada) — tabela de busca na qual será efetuada a busca
- `chaveProcurada` (entrada) — a chave de busca

```
int BuscaSequencialIdx(tTabelaIdx tabela, tCEP chaveProcurada)
{
   for (int i = 0; i < tabela->nElementos; ++i)
         /* Verifica se a chave foi encontrada */
```

```
      if (!strcmp(tabela->elementos[i].chave, chaveProcurada))
         return i; /* Elemento foi encontrado */

   return -1; /* Elemento não foi encontrado */
}
```

Note que a função `BuscaSequencialIdx()` compara chaves usando a função **strcmp()**, uma vez que as chaves são strings. A função **strcmp()** faz parte da biblioteca padrão de C.

Ao fim da sua execução, a função `BuscaSequencialIdx()` retorna o índice do primeiro elemento na tabela que casa com o valor recebido como parâmetro ou `-1`, se o valor não for encontrado. É importante notar que essa função não permite a recuperação imediata do registro procurado. Quer dizer, em vez de retornar esse registro, a referida função retorna o índice de sua chave na tabela. Assim, para completar a recuperação do registro almejado, é necessário chamar uma função adicional que usa a tabela e o referido índice como parâmetros e retorna esse registro. É isso que faz a função `ObtemElementoIdx()` apresentada abaixo.

```
tCEP_Ind ObtemElementoIdx(tTabelaIdx tabela, int indice)
{
   ASSEGURA(indice >= 0 && indice < tabela->nElementos, "Elemento inexistente");

   return tabela->elementos[indice];
}
```

Agora, por que a função `BuscaSequencialIdx()` é aparentemente tão complicada? Quer dizer, por que, em vez de retornar um índice, ela não retorna o próprio registro? A razão para tal implementação de `BuscaSequencialIdx()` é que, retornando um índice, ela pode facilmente indicar se o elemento foi encontrado ou não. Ou seja, quando ela retorna `-1`, o programa-cliente pode facilmente constatar que o registro procurado não foi encontrado. Se essa função retornasse um valor do tipo `tCEP_Ind`, como faz a função `ObtemElementoIdx()`, essa tarefa não seria tão fácil. Agora, por outro lado, a função `BuscaSequencialIdx()` poderia ser implementada retornando um ponteiro para o registro procurado, caso ele seja encontrado, ou **NULL**, caso ele não seja encontrado, como faz a função `BuscaSequencialIdx2()` apresentada a seguir.

```
tCEP_Ind *BuscaSequencialIdx2(tTabelaIdx tabela, tCEP chaveProcurada)
{
   for (int i = 0; i < tabela->nElementos; ++i)
         /* Verifica se a chave foi encontrada */
      if (!strcmp(tabela->elementos[i].chave, chaveProcurada))
         return &tabela->elementos[i]; /* Elemento encontrado */

   return NULL; /* Elemento não foi encontrado */
}
```

O próximo passo para a recuperação do registro é deixado a cargo do programa-cliente. O trecho desse programa que recupera um registro poderia ser escrito como se vê abaixo.

```
LeMatricula("CEP", umCEP, TAM_CEP + 1); /* Lê um CEP introduzido pelo usuário */

indice = BuscaSequencialIdx(tabela, umCEP); /* Procura o CEP na tabela de busca */

   /* Se a chave for encontrada, utiliza-se seu índice    */
   /* para recuperar o registro correspondente no arquivo */
if (indice < 0) {
   printf("\n>>> CEP nao foi encontrado\n");
} else {
      /* Obtém o elemento da tabela que Contém o índice do registro */
   elemento = ObtemElementoIdx(tabela, indice);

      /* Utilizando o índice do registro, calcula-se a   */
      /* posição no arquivo do primeiro byte do registro */
```

```
   primeiroByte = elemento.valor*sizeof(tRegistroCEP);

      /* Move apontador de posição para o primeiro byte do      */
      /* registro. Se isso não for possível, aborta o programa. */
   ASSEGURA( !fseek(stream, primeiroByte, SEEK_SET),
 "Erro de posicionamento em arquivo" );

      /* Lê o registro no arquivo */
   fread(&umRegistro, sizeof(umRegistro), 1, stream);

      /* Se ocorreu erro de leitura aborta */
   ASSEGURA(!ferror(stream), "Erro de leitura");

   ExibeRegistro(&umRegistro); /* Apresenta o registro ao usuário */

   putchar('\n'); /* Embelezamento */
}
```

O trecho de programa acima contém funções que não são definidas aqui, mas não deve ser difícil de entender.

Inserção

Nesta implementação de tabela de busca, na qual as chaves não são ordenadas, a melhor opção para inserção de um elemento é acrescentá-lo ao final da tabela, como faz a função `AcrescentaElementoIdx()`, apresentada abaixo e que usa os seguintes parâmetros:

- `tabela` (entrada e saída) — a tabela na qual será feito o acréscimo
- `elemento` (entrada) — o elemento a ser adicionado à tabela

```
void AcrescentaElementoIdx(tTabelaIdx tabela, const tCEP_Ind *elemento)
{
   tCEP_Ind *novoArray; /* Ponteiro para array redimensionado */

      /* Se o array estiver repleto, tenta redimensioná-lo */
   if (tabela->nElementos >= tabela->tamanhoArray) {
      tabela->tamanhoArray *= 2; /* O tamanho do array irá dobrar */

         /* Tenta redimensionar o array */
      novoArray = realloc(tabela->elementos, tabela->tamanhoArray*sizeof(tCEP_Ind));

         /* Checa se houve realocação de memória */
      ASSEGURA(novoArray, "Nao houve redimensionamento");

         /* O ponteiro que apontava para o início do array   */
         /* pode não ser mais válido e é preciso atualizá-lo */
      tabela->elementos = novoArray;
   }

      /* Acrescenta o novo elemento ao final da tabela */
   tabela->elementos[tabela->nElementos] = *elemento;

   ++tabela->nElementos; /* O tamanho da tabela aumentou */
}
```

Note que esse tipo de inserção permite que chaves duplicadas sejam inseridas na tabela e que, como a tabela de busca é implementada usando um array dinâmico, ela é redimensionada de acordo com a necessidade.

Quando um novo elemento é inserido na tabela de busca, deve haver uma inserção correspondente de registro no arquivo e essa última tarefa fica a cargo do programa-cliente, de modo que a operação completa de inserção pode ser implementada por esse programa como:

```
LeRegistro(&umRegistro); /* Lê dados do novo CEP */

   /* O valor do campo 'numero' corresponde à posição do registro */
   /* no arquivo de CEPs (isso é coisa dos Correios)              */
umRegistro.numero = ComprimentoIdx(tabela) + 1;

   /* Cria a chave a ser inserida na tabela de busca. Na tabela, */
   /* a chave é um string mas no registro CEP não é string       */
strncpy(elemento.chave, umRegistro.CEP, TAM_CEP);

elemento.chave[TAM_CEP] = '\0'; /* Torna a chave um string */

   /* Acrescenta o índice do registro ao elemento */
elemento.valor = umRegistro.numero;

   /* Acrescenta novo elemento à tabela de busca */
AcrescentaElementoIdx(tabela, &elemento);

   /* Move apontador de posição para o final do arquivo. */
   /* Se isso não for possível, aborta o programa.       */
ASSEGURA( !fseek(stream, 0, SEEK_END), "Erro de posicionamento em arquivo");

   /* Escreve o novo registro no final do arquivo */
fwrite(&umRegistro, sizeof(umRegistro), 1, stream);

   /* Se ocorreu erro de escrita, aborta */
ASSEGURA(!ferror(stream), "Erro de escrita em arquivo ");

printf("\n>>> Acrescimo bem sucedido\n");
```

Novamente, o trecho de programa acima contém chamadas de funções auxiliares cujas implementações não são apresentadas aqui, mas esse fato não deve dificultar o entendimento da ideia. O programa-cliente completo encontra-se no site dedicado ao livro.

Remoção

Nesta implementação de tabela de busca, a remoção de um elemento é efetuada em dois passos:

1. Obtém-se o índice do elemento a ser removido utilizando-se a função `BuscaSequencialIdx()` descrita acima.
2. Se o elemento for encontrado, utiliza-se o índice retornado pela função `BuscaSequencialIdx()` para removê-lo utilizando-se a função `RemoveElementoIdx()` apresentada a seguir. Essa última função recebe como parâmetros `tabela`, que representa a tabela de busca, e `indice`, que é o índice do elemento a ser removido da tabela. A mesma função retorna o elemento removido da tabela.

```
tCEP_Ind RemoveElementoIdx(tTabelaIdx tabela, int indice)
{
   tCEP_Ind itemRemovido;

         /* Verifica se o índice é válido */
   ASSEGURA( indice >= 0 && indice < tabela->nElementos,
             "Posicao de remocao inexistente" );

   itemRemovido = tabela->elementos[indice];

      /* Remover um elemento significa mover cada elemento uma posição  */
      /* para trás a partir  do sucessor do elemento que será  removido */
   for (int i = indice; i < tabela->nElementos - 1; i++)
      tabela->elementos[i] = tabela->elementos[i + 1];

   --tabela->nElementos; /* O tamanho da tabela diminuiu */

   return itemRemovido;
}
```

A função `RemoveElementoIdx()` pode ser usada por um programa-cliente como no trecho de programa abaixo.

```
   /* Lê um CEP introduzido pelo usuário */
LeMatricula("CEP", umCEP, TAM_CEP + 1);

   /* Procura o CEP na tabela de busca */
indice = BuscaSequencialIdx(tabela, umCEP);

   /* Se a chave for encontrada, remove-a da tabela de busca e         */
   /* acrescenta-se um novo elemento na tabela de removidos. A remoção */
   /* do registro do arquivo só será efetuada ao final  do programa.   */
if (indice >= 0) {
      /* Remove a chave da tabela de busca */
   (void) RemoveElementoIdx(tabela, indice);

      /* Cria um novo elemento e acrescenta-o à tabela de */
      /* removidos. O campo 'valor' não é importante aqui */
   strcpy(elemento.chave, umCEP);
   AcrescentaElementoIdx(removidos, &elemento);

   printf("\n>>> Remocao bem sucedida\n");
} else  /* O CEP não se encontra na tabela */
   printf("\n>>> CEP nao encontrado\n");
```

A remoção de um elemento da tabela de busca deve ser acompanhada pela remoção do respectivo registro no arquivo, o que, na prática, equivale a reconstruir o arquivo e talvez a própria tabela. Portanto é bem mais razoável reconstruir o arquivo uma única vez ao final do programa. Essa abordagem é implementada utilizando-se uma lista auxiliar, denominada `removidos`, que armazena os registros que deverão ser removidos. Então, antes de encerrar o programa, executa-se a instrução:

```
   /* Atualiza arquivo binário se for necessário */
if (!EstaVaziaIdx(removidos))
   AtualizaArquivoBin(NOME_ARQUIVO_BIN, removidos);
```

Construção

A função `ConstroiTabelaIdx()` lê um arquivo de dados binário e o armazena numa tabela implementada como lista indexada sem ordenação. Os parâmetros dessa função são:

- `arq` (entrada) — nome do arquivo de dados
- `*tabela` (saída) — tabela que conterá os registros lidos

Essa função retorna o stream associado ao arquivo aberto para leitura e escrita.

```
FILE *ConstroiTabelaIdx( const char *arqBin, tTabelaIdx tabela )
{
   FILE        *stream;
   tCEP_Ind     umElemento;
   tRegistroCEP umRegistro;
   int          indiceReg = 0;

      /* Tenta abrir o arquivo binário para leitura e escrita */
   stream = fopen(arqBin, "rb+");

      /* Se o arquivo não pode ser aberto, aborta o programa */
   ASSEGURA(stream, "Arquivo nao pode ser aberto");

      /* Lê cada registro do arquivo e acrescenta sua chave (CEP) na tabela */
   while (1) {
         /* Lê um registro no arquivo */
      fread( &umRegistro, sizeof(umRegistro), 1, stream);
```

```
         /* Verifica se ocorreu erro ou o final do arquivo foi atingido */
      if (ferror(stream) || feof(stream))
         break;

         /* A chave não é armazenada como string no registro. */
         /* Portanto, não se pode usar strcpy().              */
      strncpy(umElemento.chave, umRegistro.CEP, TAM_CEP);

         /* Transforma a chave num string. Isso não é */
         /* essencial, mas facilita o processamento.  */
      umElemento.chave[TAM_CEP] = '\0';

         /* Acrescenta o índice ao elemento e ao */
         /* mesmo tempo incrementa esse índice   */
      umElemento.valor = indiceReg++;

         /* Acrescenta a chave na tabela. O valor associado */
         /* à chave é sua posição (índice) na tabela.       */
      AcrescentaElementoIdx(tabela, &umElemento);
   }

      /* Se ocorreu erro de leitura no arquivo, */
      /* fecha o arquivo e aborta o programa    */
   if (ferror(stream)) {
      fclose(stream);
      ASSEGURA(0, "Erro de leitura de arquivo ");
   }

   return stream;
}
```

3.3.3 Análise

Teorema 3.1: No pior caso, o custo temporal de uma operação de busca sequencial é $\theta(n)$, em que n é o número de elementos da tabela.

Prova: A prova é trivial e é deixada como exercício para o leitor.

Teorema 3.2: No melhor caso, o custo temporal de uma operação de busca sequencial é $\theta(1)$.

Prova: A prova é trivial e é deixada como exercício para o leitor.

Teorema 3.3: Supondo que encontrar qualquer chave da tabela seja equiprovável, em média, o custo temporal de uma operação de busca sequencial é $\theta(n)$, em que n é o número de elementos da tabela.

Prova: Suponha que a tabela de busca tenha uma distribuição de probabilidade uniforme e que seu número de elementos seja n. Seja X a variável aleatória que representa o número de comparações de chaves numa operação de busca. Então $p(X = x)$ é a probabilidade de ocorrência de x comparações. Supondo que a probabilidade de encontrar a chave procurada após x comparações é a mesma para qualquer valor de x, pode-se escrever: $p(X = x) = 1/n$. $\forall x \mid 1 \leq x \leq n$. Assim o número esperado de comparações efetuadas para encontrar uma chave é dado por:

$$E[X = x] = \sum_{x=1}^{n} xp(x) = \sum_{x=1}^{n} \frac{x}{n} = \frac{1}{n} \cdot \sum_{x=1}^{n} x = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Logo, no caso médio, o custo temporal de uma operação de busca sequencial é $\theta(n)$. ■

Teorema 3.4: No pior caso, o custo temporal de uma operação de inserção numa tabela indexada (dinâmica) que oferece apenas busca sequencial é $\theta(n)$.

Prova: A inserção de um novo elemento ao final da tabela tem custo temporal $\theta(1)$. Mas, no pior caso, essa inserção requer redimensionamento, que tem custo temporal $\theta(n)$ (v. **Capítulo 9** do **Volume 1**). ■

Corolário 3.1: No pior caso, o custo temporal de construção de uma tabela de busca indexada que oferece apenas busca sequencial é $\theta(n^2)$, em que n é o número de elementos da tabela.

Prova: Como, de acordo com o **Teorema 3.4**, o custo temporal no pior caso de uma inserção é $\theta(n)$, o custo temporal de n inserções nesse caso é $\theta(n^2)$. ■

Teorema 3.5: Em qualquer caso, o custo temporal de uma operação de remoção numa tabela indexada é $\theta(n)$, em que n é o número de elementos da tabela.

Prova: Uma operação de remoção pode ser dividida em duas partes: (1) busca pela chave e (2) remoção do elemento da tabela. No melhor caso, o custo temporal da parte (1) é $\theta(1)$ (v. **Teorema 3.2**), mas a parte (2) tem custo $\theta(n)$, pois é preciso mover adiante todos os elementos que já se encontram na tabela. No pior caso e no caso médio, o custo temporal da parte (1) é $\theta(n)$ e, mesmo que a parte (2) tenha custo $\theta(1)$, o custo das partes em conjunto terá custo $\theta(n)$. Portanto, em todos esses casos, o custo temporal de uma operação de remoção numa tabela indexada é $\theta(n)$. ■

Usando-se análise assintótica tradicional, no pior caso, o custo temporal da função `AcrescentaElementoIdx()` é $\theta(n)$, e não $\theta(1)$, como talvez fosse esperado. Isso ocorre porque essa função chama **realloc()**, que tem custo $\theta(n)$ no pior caso. No entanto será mostrado no **Capítulo 5** que o custo amortizado dessa operação é $\theta(1)$.

O custo temporal de cada operação sobre a tabela de busca indexada implementada de acordo com o que foi visto nesta seção é apresentado na **Tabela 3–2**.

Operação	Melhor Caso	Pior Caso	Caso Médio
Busca	$\theta(1)$	$\theta(n)$	$\theta(n)$
Inserção	$\theta(n)$	$\theta(n)$	$\theta(n)$
Remoção	$\theta(n)$	$\theta(n)$	$\theta(n)$
Construção	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$

Tabela 3–2: Custos Temporais de Operações com Tabelas Indexadas sem Ordenação

3.4 Busca Sequencial com Movimentação

3.4.1 Conceitos

A eficiência de busca sequencial pode ser melhorada movendo-se os registros com maior frequência de acesso para o início ou para uma posição mais próxima do início da tabela de busca. Existem diversas maneiras (heurísticas) de organização de tabelas de busca que procuram atingir esse objetivo. Esta seção descreve duas delas:

1. **Heurística de movimentação para o início**. Após a chave desejada ser localizada, ela é colocada no início da tabela. Mais precisamente, sempre que uma busca obtiver êxito, o elemento que contém a chave recuperada é movido para o início da tabela.
2. **Heurística de transposição**. Após o elemento desejado ser localizado, ele é trocado com seu predecessor a não ser que ele esteja no início da lista. Ou seja, quando um registro é recuperado, o elemento que contém sua chave na tabela de busca é trocado com seu antecessor.

A principal justificativa que norteia essas heurísticas é semelhante ao uso de cache. Ou seja, essas heurísticas são baseadas na expectativa de que uma chave recém acessada será provavelmente acessada novamente em breve. Assim colocando tal chave na frente da tabela de busca ou próximo dela, as buscas subsequentes serão mais rápidas.

No caso de posicionamento da chave no início da tabela de busca, como se espera que ela seja acessada novamente, ela deve ser colocada na melhor posição da tabela para que isso aconteça (i.e., na frente da tabela). Já no método de transposição, um único acesso não significa que a chave será acessada com frequência. Se a chave for movida para a frente apenas uma posição de cada vez, garante-se que ela estará na frente apenas se realmente tiver alta frequência de busca. Nos dois casos, tenta-se posicionar as chaves mais prováveis de serem procuradas próximo ao início da lista. Isso ocorre mais categoricamente com a heurística de movimentação para a frente e mais cautelosamente com o método de transposição.

A **Figura 3–6 (a)** mostra que, quando a tabela é implementada como lista indexada, no máximo dois elementos da lista são alterados. Por outro lado, a **Figura 3–6 (b)** mostra que a busca com movimentação para o início é inerentemente ineficiente quando a lista é indexada. Nessas duas figuras, os elementos com fundo colorido são aqueles afetados na operação. Assim o método de movimentação para o início é eficiente apenas se a tabela for implementada em forma de lista encadeada, pois mover um elemento para o início de uma lista encadeada tem custo temporal $\theta(1)$, já que temos o ponteiro para o início da lista.

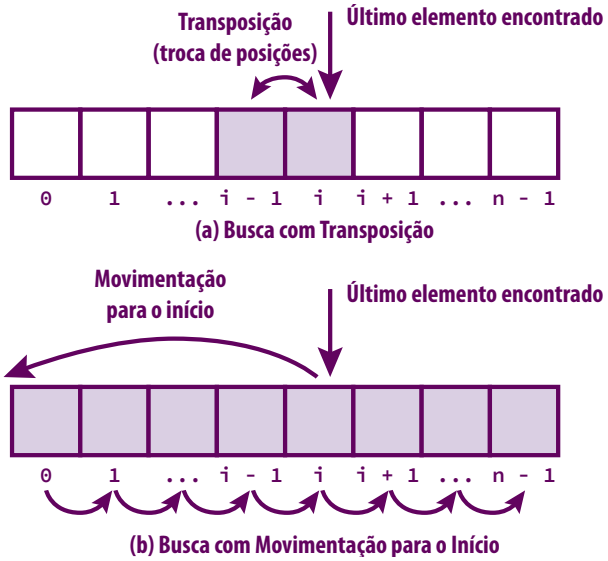


Figura 3–6: Transposição e Movimentação para Início em Lista Indexada

3.4.2 Implementação Usando Lista Encadeada

Para implementação da tabela de busca com movimentação utilizando lista simplesmente encadeada (**tabela de busca encadeada**[2]), é utilizada a seguinte definição de tipo[3]:

```
typedef struct rotNoLSE {
          tCEP_Ind         conteudo;
          struct rotNoLSE *proximo;
       } tNoListaSE, *tListaSE;
```

O tipo `tNoCaminhoB` utilizado na definição acima foi definido na **Seção 3.3.2**.

Busca Sequencial com Transposição

A **Figura 3–7** ilustra uma busca com transposição numa tabela de busca implementada como lista encadeada.

[2] Embora a denominação mais adequada para uma tabela de busca implementada usando lista encadeada seja *tabela de busca sequencialmente encadeada*, visto que árvores também podem ser usadas para implementar tabelas de busca encadeadas, para simplificar a linguagem, essa denominação, ou simplesmente **tabela encadeada**, será doravante utilizada.

[3] O acrônimo *LSE* que aparece em alguns identificadores na implementação a seguir é derivado de *Lista Simplesmente Encadeada* (v. **Apêndice D**).

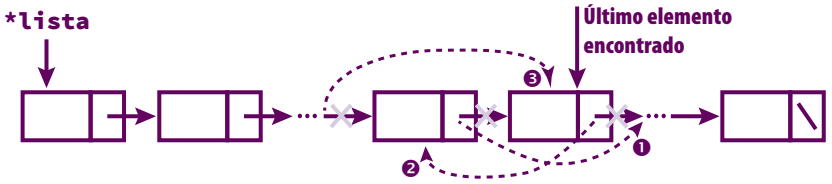


Figura 3–7: Busca com Transposição em Lista Encadeada

A função `BuscaComTransposicaoLSE()` retorna o endereço do conteúdo efetivo do nó que possui uma determinada chave numa tabela de busca implementada como uma lista simplesmente encadeada. Se esse nó for encontrado, troca-o de posição com seu antecessor. Essa função retorna o endereço dos dados que contêm o referido conteúdo, se a busca obtiver êxito. Caso contrário, ela retorna **NULL**. Os parâmetros dessa função são:

- `*lista` — a tabela de busca
- `chave` — a chave de busca

```
tCEP_Ind *BuscaComTransposicaoLSE(tListaSE *lista, tCEP chave)
{
   tListaSE p,        /* p apontará para o nó corrente     */
            q = NULL, /* q apontará para o antecessor de p */
            r = NULL; /* r apontará para o antecessor de q */

      /* Enquanto p não assume NULL ou a chave de um nó não */
      /* casa com o parâmetro 'chave', a busca prossegue    */
   for ( p = *lista; p && strcmp(p->conteudo.chave, chave);
         p = p->proximo) {
      r = q;
      q = p;
   }

      /* Se 'p' assume NULL é porque a chave especificada */
      /* como parâmetro não foi encontrado                */
   if(!p)
      return NULL; /* Chave não foi encontrada */

      /*                                                  */
      /* Neste ponto, q aponta para o antecessor imediato */
      /* de p e r aponta para o antecessor imediato de q  */

   if (!q) /* O nó encontrado já é o primeiro da lista */
      return &p->conteudo; /* Não é necessária a transposição */

   q->proximo = p->proximo;
   p->proximo = q;

   if (!r)
         /* Nó encontrado passa a ser o primeiro da lista */
      *lista = p;
   else
      r->proximo = p;

   return &p->conteudo;
}
```

Se você estudar a implementação da função `BuscaComTransposicaoLSE()` enquanto examina a **Figura 3–7** será mais fácil entender essa função.

Busca Sequencial com Movimentação para Início

A **Figura 3–8** ilustra uma busca com movimentação para o início numa tabela de busca encadeada.

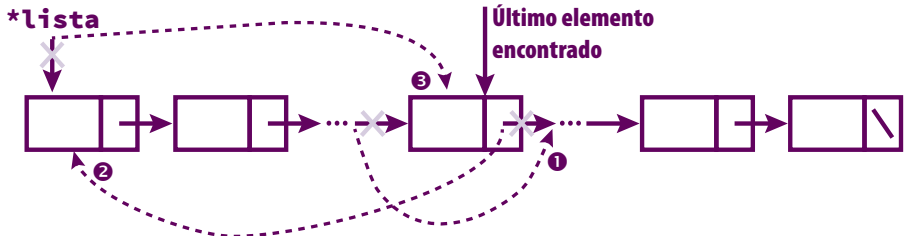


Figura 3–8: Busca com Movimentação para Início em Lista Encadeada

A função `tNoCaminhoB`, definida a seguir, implementa busca com movimentação para início usando lista encadeada. Os parâmetros e a especificação de retorno dessa função são os mesmos da função `BuscaComTransposicao()` apresentada acima.

```
tCEP_Ind *BuscaComMovimentoLSE(tListaSE *lista, tCEP chave)
{
   tListaSE p = *lista, /* p apontará para o nó corrente     */
            q = NULL;   /* q apontará para o nó anterior a p */

      /* Enquanto p não assume NULL ou a chave de um nó não */
      /* casa com o parâmetro 'chave', a busca prossegue    */
   while (p && strcmp(p->conteudo.chave, chave)) {
      q = p;
      p = p->proximo;
   }

      /* Se 'p' assume NULL é porque a chave especificada */
      /* como parâmetro não foi encontrado                */
   if(!p)
      return NULL; /* Chave não foi encontrada */

      /* A chave foi encontrada. Se o nó que a contém for o primeiro */
      /* da lista, a movimentação para o início não é necessária.    */
      /* Caso contrário, ela será efetuada a seguir.                 */
   if (p != *lista) {
      q->proximo = p->proximo;
      p->proximo = *lista;
      *lista = p;
   }

      /* Neste ponto, p aponta para o nó encontrado. Então, */
      /* retorna-se o endereço de seu campo 'conteudo'.     */
   return &p->conteudo;
}
```

O uso da **Figura 3–8** em conjunto com o código facilitará o entendimento da implementação da função `tNoCaminhoB()`.

3.4.3 Análise

Teorema 3.6: Se forem admitidas chaves duplicadas, em qualquer caso, o custo temporal de uma operação de inserção numa tabela de busca encadeada é $\theta(1)$.

Prova: Se a inserção não incluir uma operação de busca para verificar se a chave do novo elemento já existe na tabela, ela pode ocorrer no início da tabela e essa operação tem custo temporal $\theta(1)$. ■

Corolário 3.2: Se forem admitidas chaves duplicadas, o custo temporal de construção de uma tabela de busca encadeada é $\theta(n)$, em que n é o número de elementos da tabela.

Prova: De acordo com o **Teorema 3.6**, o custo temporal de uma inserção é $\theta(1)$. Portanto o custo de n inserções é $\theta(n)$. ■

Teorema 3.7: No melhor caso, o custo temporal de uma operação de remoção numa tabela de busca encadeada é $\theta(1)$.

Prova: No melhor caso, o custo temporal da operação de busca tem custo temporal $\theta(1)$. Tendo encontrado o elemento a ser removido, as demais operações (alteração de ponteiros e liberação do nó) têm custo $\theta(1)$. ■

Teorema 3.8: No pior caso e no caso médio, o custo temporal de uma operação de remoção numa tabela de busca encadeada é $\theta(n)$, em que n é o número de elementos da tabela.

Prova: A prova desse teorema é similar àquela do **Teorema 3.5** e é deixada como exercício para o leitor.

Teorema 3.9: O custo espacial de uma tabela encadeada contendo n elementos é $\theta(n)$.

Prova: Além do conteúdo efetivo, cada elemento (nó) da tabela contém um campo contendo o endereço do próximo elemento. ■

O custo temporal de cada operação sobre a tabela de busca implementada utilizando lista encadeada de acordo com o que foi visto nesta seção é exibida resumidamente na **Tabela 3–3**.

Operação	Melhor Caso	Pior Caso	Caso Médio
Busca	$\theta(1)$	$\theta(n)$	$\theta(n)$
Inserção	$\theta(1)$	$\theta(1)$	$\theta(1)$
Remoção	$\theta(1)$	$\theta(n)$	$\theta(n)$
Construção	$\theta(n)$	$\theta(n)$	$\theta(n)$

Tabela 3–3: Custos Temporais de Operações com Tabelas Encadeadas

Estudos experimentais mostram que o método de transposição é mais eficiente quando ele é utilizado com aplicativos que efetuam um grande número de buscas e quando as chaves são mais ou menos equiprováveis (i.e., se elas têm aproximadamente a mesma frequência de acesso). Por outro lado, o método de movimentação para frente é mais eficiente para um número de buscas que se situa entre pequeno e médio e quando as chaves não são tão equiprováveis.

No pior caso, a heurística de movimentação para frente é mais eficiente do que o método de transposição e, por isso, ele é o preferido na maioria das situações que requerem busca sequencial. A grande vantagem do método de transposição é que ele pode ser aplicado eficientemente sobre arrays e listas encadeadas.

Apesar de as heurísticas apresentadas aqui apresentarem potencial de acelerar as buscas, em termos de análise assintótica (i.e., em termos de notação ó), o custo temporal das buscas discutidas nesta seção não é afetado.

3.5 Busca Linear em Tabela Ordenada

Se uma tabela de busca implementada como lista indexada ou encadeada estiver ordenada, podem-se usar técnicas elementares para melhorar a eficiência das operações de busca. Quer dizer, se uma tabela com n elementos estiver ordenada, são necessárias apenas $n/2$ comparações em média para efetuar uma operação de busca. Isso ocorre porque, se, durante uma busca, for encontrada uma chave maior do que a chave de busca, sabe-se que ela não faz parte da tabela. Contudo essa melhora de eficiência não é suficiente para abrandar o custo temporal dessa operação em termos de notação ó, que continua sendo $\theta(n)$.

Agora, se a tabela de busca for implementada como lista *indexada* ordenada, podem-se efetuar buscas com custo $\theta(\log n)$, como será visto nesta seção.

É importante notar que essas técnicas não podem ser aplicadas quando a tabela de busca é implementada como lista *encadeada*, quer ela esteja ordenada ou não. Isso ocorre porque ambas as técnicas que serão descritas aqui, busca binária e busca por interpolação, requerem acesso imediato a alguns elementos, o que só é possível por meio de indexação. Uma lista encadeada não permite acesso dessa natureza. Por exemplo, o único modo de atingir o elemento mediano de uma lista simplesmente encadeada é seguindo ponteiros para nós a partir do nó inicial.

O problema agora é como construir e manter uma lista indexada ordenada. Bem, inserir um único elemento numa lista indexada tem custo temporal $\theta(n)$, visto que essa operação pode ser dividida em duas etapas[4]:

1. Encontrar o local de inserção. O custo temporal dessa etapa é $\theta(n)$.
2. Efetuar a inserção propriamente dita. Inexoravelmente, essa etapa tem custo temporal $\theta(n)$, pois, no pior caso, todos os elementos que já se encontram na lista precisam ser movidos para ceder espaço para o novo elemento.

Portanto, pela regra da soma de análise de algoritmos, o custo temporal de uma operação de inserção ordenada é $\theta(n)$. Assim construir uma tabela de busca implementada como lista ordenada por meio de inserções sucessivas tem custo temporal total quadrático, de modo que não se deve usar esse método para tabelas grandes. Isso significa na prática que criar uma tabela de busca usando essa técnica para o conteúdo do arquivo `Tudor.bin` é admissível, mas isso é inaceitável para o arquivo `CEPs.bin` (v. **Apêndice A**).

O arquivo de exemplo a ser utilizado é exatamente `CEPs.bin`, de modo que a abordagem a ser adotada aqui consiste em dois passos:

1. A tabela de busca é criada usando a função `AcrescentaElementoIdx()` discutida na **Seção 3.3.2**. O custo temporal amortizado dessa função é $\theta(1)$ (v. **Capítulo 5**), de modo que o custo amortizado de construção de toda a tabela é $\theta(n)$.
2. A tabela é ordenada usando um algoritmo de ordenação *ótimo* [i.e., com custo temporal $\theta(n \cdot log\ n)$]. Tipicamente, por razões pragmáticas, o algoritmo escolhido para essa tarefa é *QuickSort*[5] (v. **Capítulo 11**).

A ordenação da tabela de busca criada no **Passo 1** acima pode ser efetuada pela função `OrdenaTabelaIdx()` que transfere sua tarefa para **qsort()**, que é uma função da biblioteca padrão de C declarada em `<stdlib.h>` e que foi discutida no **Capítulo 11** do **Volume 1**. Os parâmetros da função `OrdenaTabelaIdx()` são:

- `tabela` (entrada e saída) — tabela que será ordenada
- `FComp` (entrada) — ponteiro para a função que compara elementos da tabela

```
void OrdenaTabelaIdx(tTabelaIdx tabela, int (*FComp) (const void *, const void *))
{
      /* A função qsort() faz o serviço */
   qsort(tabela->elementos, tabela->nElementos, sizeof(tabela->elementos[0]), FComp);
}
```

O encapsulamento da função **qsort()** pela função `OrdenaTabelaIdx()` se faz necessário porque, como o tipo `tTabelaIdx` é opaco (v. **Seção 3.3.2**), um programa-cliente não tem acesso aos três primeiros parâmetros necessários para a chamada de **qsort()**.

A função de comparação usada por **qsort()** é definida no programa-cliente como:

```
int ComparaCEPs(const void *s1, const void *s2)
{
      /* Convertem-se os ponteiros genéricos em ponteiros para o tipo tCEP_Ind */
   const tCEP_Ind *e1 = (tCEP_Ind *)s1;
   const tCEP_Ind *e2 = (tCEP_Ind *)s2;
```

[4] Uma análise similar poderia ser feita para inserção ordenada em lista encadeada, mas isso não é de interesse neste caso.

[5] As referidas razões pragmáticas são decorrentes de dois fatos: (1) esse algoritmo é o mais recomendável quando a configuração dos dados não é conhecida e (2) esse algoritmo é implementado na biblioteca de qualquer linguagem de programação decente.

```
      /* Os elementos são comparados por meio de suas chaves, que são strings */
   return strcmp(e1->chave, e2->chave);
}
```

Após construir a tabela de busca usando a função `tNoCaminhoB`, como foi visto na **Seção 3.3**, ordena-se essa tabela chamando-se a função `OrdenaTabelaIdx()` como se vê abaixo:

```
OrdenaTabelaIdx(tabela, ComparaCEPs);
```

Manter uma tabela ordenada tem custo temporal $\theta(mn)$, em que n é o tamanho da tabela e m é o número de inserções consecutivas, mas esse custo pode ser tolerado se o número de operações de busca for bem maior do que o número de inserções. Assim a função `tNoCaminhoB`, definida a seguir, pode ser usada para inserções esporádicas (ou pouco frequentes) numa tabela de busca ordenada. Os parâmetros dessa função são:

- `lista` (entrada e saída) — lista na qual será feita a inserção
- `*elemento` (entrada) — elemento que será inserido

```
void InsereEmOrdemIdx(tTabelaIdx tabela, const tCEP_Ind *elemento)
{
   int        posicao; /* Posição de inserção do elemento */
   tCEP_Ind *novoArray; /* Ponteiro para o array redimensionado */

      /* Se o array estiver repleto, tenta redimensioná-lo */
   if (tabela->nElementos >= tabela->tamanhoArray) {
      tabela->tamanhoArray *= 2; /* Dobrará o tamanho do array */

         /* Tenta redimensionar o array */
      novoArray = realloc(tabela->elementos, tabela->tamanhoArray*sizeof(tCEP_Ind));

         /* Checa se houve realocação de memória */
      ASSEGURA(novoArray, "Nao houve redimensionamento");

         /* O ponteiro que apontava para o início do array pode não */
         /* ser mais válido.  Portanto, é preciso atualizá-lo.      */
      tabela->elementos = novoArray;
   }

      /* Se a tabela estiver vazia, o elemento é */
      /* inserido na primeira posição do array  */
   if (EstaVaziaIdx(tabela)) {
         /* O elemento será o primeiro da tabela */
      tabela->elementos[0] = *elemento;

      ++tabela->nElementos; /* A tabela cresceu */

      return;
   }

      /* Encontra a posição no array onde o elemento será inserido */
   for (posicao = 0; posicao < tabela->nElementos; ++posicao)
      if ( strcmp(tabela->elementos[posicao].chave, elemento->chave) > 0 )
         break; /* Posição de inserção encontrada */

      /* Abre espaço para o novo elemento */
   for (int i = tabela->nElementos - 1; i >= posicao; --i)
         /* Move cada elemento uma posição adiante a partir do  */
         /* elemento que ora se encontra na posição de inserção */
      tabela->elementos[i + 1] = tabela->elementos[i];

   tabela->elementos[posicao] = *elemento; /* Insere o novo elemento */

   ++tabela->nElementos; /* A tabela cresceu */
}
```

3.5.1 Busca Binária

Conceito

Busca binária é um tipo de busca que se aplica apenas a tabelas de busca com chaves ordenadas em ordem crescente ou decrescente. Aqui, supõe-se que as chaves são ordenadas em ordem crescente, mas se a ordem utilizada for oposta, o raciocínio empregado é semelhante. A denominação *busca binária* é decorrente do fato de a tabela de busca ser continuamente dividida em duas partes iguais.

Busca binária é *mais ou menos*[6] análoga a uma busca por uma palavra num dicionário tradicional (de papel). Ou seja, inicialmente, procura-se a palavra desejada no meio da lista. Se o nome procurado estiver na página central, a busca encerra; se a palavra procurada estiver além daquelas encontradas na página central, reduz-se a busca à metade final da lista. Por outro lado, se a palavra procurada estiver abaixo daqueles encontrados na página central, reduz-se a busca à metade inicial da lista. Então se for o caso, o procedimento é repetido sucessivamente considerando a metade inicial ou o final da lista.

Se a tabela contém um número de elementos par não existe um único elemento central. De fato, nesse caso, existem dois elementos centrais cujos índices são determinados por $\lfloor (inf + sup)/2 \rfloor$ e $\lceil (inf + sup)/2 \rceil$. Logo o algoritmo deve escolher qual desses elementos utilizar na comparação de chaves. De fato, qualquer um deles serve igualmente, mas, do ponto de vista pragmático, é melhor usar o piso do que o teto da divisão $(inf + sup)/2$. A razão para tal escolha é o modo como muitas linguagens de programação (e. g., C, C++ e Java) calculam o resultado de uma divisão inteira na qual o numerador e o denominador são positivos[7].

Mais formalmente, o algoritmo seguido por uma busca binária é apresentado na **Figura 3–9**.

ALGORITMO BUSCABINÁRIA

ENTRADA: Uma tabela indexada ordenada com n elementos e uma chave de busca

SAÍDA: O valor associado à primeira chave da tabela que casa com a chave de busca ou um valor indicando que a chave não foi encontrada

1. Atribua à *inf* o menor índice e a *sup* o maior índice da tabela
2. Enquanto $inf \leq sup$ faça:
   - **2.1** Atribua $\lfloor inf + (sup - inf)/2 \rfloor$ à variável *meio*
   - **2.2** Compare a chave de busca com a chave do elemento que se encontra no índice *meio* da tabela
   - **2.3** Se as chaves forem iguais, encerre a busca indicando o sucesso da operação
   - **2.4** Se a chave de busca for menor do que a chave do elemento que se encontra no índice *meio* da tabela, atribua *meio – 1* à *sup*
   - **2.5** Caso contrário (se a chave de busca for maior do que a chave do elemento que se encontra no índice meio da tabela), atribua *meio + 1* à *inf*
3. Retorne um valor que indique que a chave de busca não foi encontrada

FIGURA 3–9: ALGORITMO DE BUSCA BINÁRIA

Note que, na discussão anterior a posição central é calculada como $\lfloor (inf + sup)/2 \rfloor$, enquanto, no algoritmo da **Figura 3–9**, esse valor é calculado como $\lfloor inf + (sup - inf)/2 \rfloor$. De fato os resultados obtidos são equivalentes do ponto de vista matemático, mas, do ponto de vista de programação, calcular o meio da tabela como

[6] *Mais ou menos*, porque apenas um debiloide irá procurar uma palavra num dicionário ou em qualquer outra lista ordenada em ordem alfabética que comece em *A*, por exemplo, começando pelo meio da lista. Essa é uma analogia ridícula que aparece em outros livros de programação que o autor decidiu seguir por falta de exemplo melhor. Além disso, nenhuma pessoa normal consegue abrir um dicionário exatamente em sua página central.

[7] Nesse caso, o truncamento efetuado por essas linguagens corresponde à aplicação da função piso. Por exemplo, $5/2 = 2$, que é o piso dessa divisão.

⌊*(inf* + *sup)/2*⌋ é problemático, pois se *sup* for um valor bem grande essa soma pode causar overflow e fazer com que o índice calculado seja negativo.

Implementação

A função `BuscaBinariaIdx()`, definida a seguir, implementa o algoritmo da **Figura 3–9** e tem como parâmetros:

- `tabela` (entrada) — a tabela de busca
- `chave` (entrada) — a chave de busca

```
int BuscaBinariaIdx(tTabelaIdx tabela, tCEP chave)
{
   int inf, /* Limite inferior da busca */
       sup, /* Limite superior da busca */
       meio, /* Meio do intervalo */
       teste; /* Resultado da comparação de duas chaves */

   inf = 0; /* Limite inferior inicial */
   sup = tabela->nElementos - 1; /* Limite superior inicial */

      /* Efetua a busca binária */
   while (inf <= sup) {
         /* Calcula o meio do intervalo */
      meio = inf + (sup - inf)/2;

      teste = strcmp(chave, tabela->elementos[meio].chave);

         /* Verifica se a chave se encontra */
         /* no meio do intervalo            */
      if (!teste)
         return tabela->elementos[meio].valor; /* Sucesso */

         /* Ajusta o intervalo de busca */
      if (teste < 0)
         sup = meio - 1;
      else
         inf = meio + 1;
   }

   return -1; /* Elemento não foi encontrado */
}
```

Note que o retorno da função `BuscaBinariaIdx()` é substancialmente diferente daquele da função `BuscaSequencialIdx()` discutida na **Seção 3.3.2**. Quer dizer, quando obtém êxito, essa última função retorna o índice do elemento que contém a chave de busca, de modo que o programa-cliente precisa chamar a função `tNoCaminhoB` para ter acesso ao elemento em si. Por outro lado, quando bem-sucedida, a função `BuscaBinariaIdx()` retorna o valor associado à chave de busca, que é suficiente para recuperar o registro desejado no arquivo, como foi mostrado na **Seção 3.3.2**.

A Função bsearch()

A biblioteca padrão de C provê a função **bsearch()** (incluir `<stdlib.h>`) que executa busca binária em arrays de elementos de qualquer tipo. Essa função tem como protótipo:

```
void *bsearch( const void *chave, const void *array, size_t nElem, size_t tamanho,
               int (*FComp)(const void *e1, const void *e2) )
```

Os parâmetros da função **bsearch()** têm a seguinte interpretação:

- `chave` — ponteiro para a chave a ser procurada
- `array` — array a ser pesquisado
- `nElem` — número de elementos do array
- `tamanho` — tamanho de cada elemento do array
- `FComp` — ponteiro para uma função que compara os elementos do array. Essa função deve possuir dois parâmetros do tipo **const void** *, cada um dos quais aponta para um elemento do array a ser pesquisado. A função deve comparar estes elementos e retornar um valor inteiro de acordo com os seguintes critérios de comparação:
  - Um valor menor do que zero, se o primeiro elemento for menor do que o segundo.
  - Zero, se os elementos forem iguais.
  - Um valor maior do que zero, se o primeiro elemento for maior do que o segundo.

A função **bsearch()** retorna o endereço do primeiro elemento encontrado no array que tiver uma chave igual à chave de busca ou **NULL**, se a chave não for encontrada. Se a tabela de busca (array) não estiver ordenada ou se ela não for implementada como lista indexada, o resultado será indefinido. Além disso, se a chave não for primária, o primeiro elemento encontrado não é necessariamente o primeiro elemento da tabela que casa com a chave procurada.

A função **bsearch()** é apresentada em maiores detalhes no **Capítulo 11** do **Volume 1** desta obra.

3.5.2 Busca por Interpolação

Conceito

Um possível melhoramento com relação à busca binária consiste em calcular com mais precisão o índice da tabela na qual se espera encontrar a chave de busca, em vez de supor que ela sempre se encontra no meio de cada intervalo. Essa técnica, chamada **busca por interpolação**, imita o modo como se procura uma palavra num dicionário com mais precisão (ao contrário da analogia apresentada para busca binária): se o elemento que se está procurando começa com uma letra próxima ao início do alfabeto, abre-se o dicionário próximo ao seu início, se a palavra começa com uma letra próxima ao final do alfabeto, abre-se o dicionário próximo ao seu final e assim por diante. Assim como ocorre com busca binária, busca por interpolação pode ser aplicada apenas a tabelas de busca ordenadas implementadas como listas indexadas.

A busca por interpolação é semelhante à busca binária no sentido de que em ambos os casos a busca é realizada entre dois limites — *inf* e *sup* — que são continuamente reduzidos, mas, diferentemente da busca binária, a busca por interpolação não divide a tabela em duas metades iguais. Quer dizer, o local onde se espera encontrar a chave é calculado como:

$$meio = inf + (sup - inf)\frac{chave - chave[inf]}{chave[sup] - chave[inf]}$$

Como na busca binária, se *chave* < *chave(meio)*, redefine-se *sup* com *meio* – *1* e, se *chave* > *chave(meio)*, redefine-se *inf* com *meio* + *1*. Esse processo é repetido até que a chave seja encontrada ou que o valor de *inf* seja maior do que *sup*.

A principal desvantagem de busca por interpolação é que, se as chaves não forem uniformemente distribuídas, as buscas não serão eficientes. Na prática, frequentemente, as chaves não são uniformemente distribuídas (p. ex., num dicionário, há mais nomes começando com *A* do que com *X*). Outra desvantagem dessa técnica é que ela é inconveniente para chaves que não sejam numéricas. Ou seja, busca por interpolação utilizando strings não é trivial, a não ser que esses strings sejam estritamente numéricos, como será visto na implementação a seguir.

Implementação

A função `tNoCaminhoB` definida a seguir efetua uma busca por interpolação numa tabela de busca implementada como lista indexada ordenada. Os parâmetros e o valor retornado por essa função têm a mesma interpretação que aqueles da função `tNoCaminhoB` apresentada na **Seção 3.5.1**.

```
int BuscaInterpolacaoIdx(tTabelaIdx tabela, tCEP chave)
{
   int inf, /* Limite inferior da busca */
       sup, /* Limite superior da busca */
       meio, /* Meio do intervalo */
       chaveInf, chaveSup, /* Chaves dos limites inferior e */
                           /* superior convertidas em int   */
       teste; /* Resultado da comparação de duas chaves */

   inf = 0; /* Limite inferior inicial */
   sup = tabela->nElementos - 1; /* Limite superior inicial */

      /* Efetua a busca por interpolação */
   while (inf <= sup) {
         /* Converte as chaves nos extremos do intervalo */
      chaveInf = atoi(tabela->elementos[inf].chave);
      chaveSup = atoi(tabela->elementos[sup].chave);

         /* Calcula o meio do intervalo */
      meio = inf + (sup - inf)*((atoi(chave) - chaveInf)/(chaveSup - chaveInf));

         /* Compara a chave de busca com a que se encontra no meio do intervalo */
      teste = strcmp(chave, tabela->elementos[meio].chave);

         /* Verifica se o elemento encontra-se no meio do intervalo */
      if (!teste)
         return tabela->elementos[meio].valor; /* Sucesso */

      if (teste < 0) /* Ajusta o intervalo de busca */
         sup = meio - 1;
      else
         inf = meio + 1;
   }

   return -1; /* Elemento não foi encontrado */
}
```

Note que a função `BuscaInterpolacao()` chama a função **atoi()** da biblioteca padrão de C (incluir `<stdlib.h>`) para converter as chaves que se encontram nos extremos do intervalo de busca. Isso é possível porque as chaves (CEPs) são strings numéricos.

3.5.3 Análise

Lema 3.1: Se k for um número inteiro e x for um número real tais que $2^k \leq x < 2^{k+1}$, então $k = \lfloor log_2 x \rfloor$.

Prova: Se $2^k \leq x < 2^{k+1}$, aplicando-se logaritmos na base 2 aos termos dessa relação, obtém-se $k \leq log_2 x < k+1$ (pois $f(x) = log_2 x$ é uma função crescente). Pela definição de função piso, tem-se que $k = \lfloor log_2 x \rfloor$. ■

Lema 3.2: Para qualquer inteiro $n > 1$ que seja ímpar, $\lfloor log_2 (n-1) \rfloor = \lfloor log_2 n \rfloor$.

Prova: Se n for um inteiro ímpar e $n > 1$, tem-se que $2^k < n < 2^{k+1}$, para algum $k > 0$. Portanto, de acordo com o **Lema 3.1**, tem-se $k = \lfloor log_2 n \rfloor$ (†). Além disso, tem-se: $2^k < n < 2^{k+1} \Rightarrow 2^k \leq n-1 < 2^{k+1} \Rightarrow k = \lfloor log_2 (n-1) \rfloor$ (††). Essa última implicação é novamente decorrência do **Lema 3.1**. De (†) e (††) , obtém-se $\lfloor log_2 (n-1) \rfloor = \lfloor log_2 n \rfloor$. ■

Lema 3.3: Se, no início do laço do algoritmo de busca binária, a tabela busca usada como entrada contiver n elementos, após uma iteração na qual a chave de busca não é encontrada, essa tabela terá, no máximo, $\lfloor n/2 \rfloor$ elementos.

Prova: O tamanho da próxima tabela de busca a ser usada no laço depende do fato de n ser par ou ímpar. Se n for ímpar, os tamanhos das subtabelas esquerda e direita serão os mesmos e serão dados por: $(n - 1)/2 = \lfloor n/2 \rfloor$. Se n for par, o tamanho da subtabela esquerda será $n/2 - 1 = \lfloor n/2 \rfloor - 1$ e o tamanho da subtabela direita será $n/2 = \lfloor n/2 \rfloor$. Portanto os tamanhos possíveis para a próxima tabela de busca são $\lfloor n/2 \rfloor - 1$ e $\lfloor n/2 \rfloor$, e o maior deles é $\lfloor n/2 \rfloor$. ■

Teorema 3.10: O número máximo de comparações de chaves efetuadas numa busca binária é $\lfloor \log_2 n \rfloor + 1$, em que n é o número de elementos da tabela recebida como entrada pelo algoritmo de busca.

Prova: Seja n o tamanho da tabela recebida como entrada pelo algoritmo de busca binária. No pior caso, de acordo com o **Lema 3.3**, a cada iteração do laço desse algoritmo, a tabela de busca terá tamanho igual a $\lfloor n/2 \rfloor$. Portanto o número máximo de iterações desse laço para uma tabela com n elementos é igual a 1 mais o número máximo de iterações para uma tabela de tamanho $\lfloor n/2 \rfloor$. Ou seja,

$$f(n) = f(\lfloor n/2 \rfloor) + 1$$

Para $n = 1$, tem-se que $f(1) = 1$, visto que, para uma tabela de tamanho igual a 1, o laço é executado apenas uma vez. Para tentar encontrar uma conjectura de solução para essa relação de recorrência, serão examinados a seguir alguns de seus valores:

	n	$f(n)$
$f(1) = 1$	$1 = 2^0$	$1 = 0 + 1$
$f(2) = f(\lfloor 2/2 \rfloor) + 1 = f(1) + 1 = 2$	$2 = 2^1$	$2 = 1 + 1$
$f(3) = f(\lfloor 3/2 \rfloor) + 1 = f(1) + 1 = 2$		
$f(4) = f(\lfloor 4/2 \rfloor) + 1 = f(2) + 1 = 3$	$4 = 2^2$	$3 = 2 + 1$
$f(5) = f(\lfloor 5/2 \rfloor) + 1 = f(2) + 1 = 3$		
$f(6) = f(6/2 \rfloor) + 1 = f(3) + 1 = 3$		
$f(7) = f(\lfloor 7/2 \rfloor) + 1 = f(3) + 1 = 3$		
$f(8) = f(\lfloor 8/2 \rfloor) + 1 = f(4) + 1 = 4$	$8 = 2^3$	$4 = 3 + 1$
$f(9) = f(\lfloor 9/2 \rfloor) + 1 = f(4) + 1 = 4$		
⋮	⋮	⋮
$f(16) = f(\lfloor 16/2 \rfloor) + 1 = f(8) + 1 = 5$	$16 = 2^4$	$5 = 4 + 1$
⋮	⋮	⋮

No desenvolvimento acima, quando n se encontra entre duas potências de 2, $f(n)$ é 1 mais a menor dessas potências. Ou seja, formalmente, se $2^k \leq n < 2^{k+1}$, então $f(n) = n + k$.

Agora, de acordo com o **Lema 3.1**, tem-se que, nesse caso, $k = \lfloor \log_2 n \rfloor$. Portanto a conjectura que será verificada a seguir usando indução finita é que a solução da relação de recorrência é $f(n) = \lfloor \log_2 n \rfloor + 1$, $\forall n \mid n \geq 1$.

Base da indução. Para $n = 1$, tem-se que $f(1) = \lfloor \log_2 1 \rfloor + 1 = 1$.

Hipótese indutiva. Suponha que $f(n) = \lfloor \log_2 n \rfloor + 1$, $\forall n \mid 1 \leq n \leq k$.

Passo indutivo. Deve-se mostrar que $f(k + 1) = \lfloor \log_2 (k + 1) \rfloor + 1$.

Caso 1: Suponha que k seja par.

	Justificativa
$f(k+1) = f(\lfloor (k+1)/2 \rfloor) + 1$	Por definição de $f(n)$
$= f(\lfloor k/2 \rfloor) + 1$	$k+1$ é ímpar
$= (\lfloor \log_2 (k/2) \rfloor + 1) + 1$	Hipótese indutiva, pois, como k é par, $k \geq 2$ e, consequentemente, $1 \leq \lfloor k/2 \rfloor \leq k/2 < k$
$= \lfloor \log_2 k - \log_2 2 \rfloor + 2$	Propriedade de logaritmos
$= \lfloor \log_2 k - 1 \rfloor + 2$	Propriedade de logaritmos
$= \lfloor \log_2 k \rfloor - 1 + 2$	Propriedade de piso: $\lfloor x - 1 \rfloor = \lfloor x \rfloor - 1$
$= \lfloor \log_2 (k+1) \rfloor + 1$	**Lema 3.2**

Caso 2: Suponha que k seja ímpar. Deve-se mostrar que, quando k é ímpar, obtém-se $f(k+1) = \lfloor \log_2 (k+1) \rfloor + 1$. Essa parte da prova é deixada como exercício para o leitor. ■

Corolário 3.3: No pior caso, uma busca binária tem custo $\theta(\log n)$, em que n é o tamanho (número de elementos) da tabela.

Prova: No pior caso, o algoritmo de busca binária efetua o número máximo de comparações de chaves, que, de acordo com o **Teorema 3.10**, é $\lfloor \log_2 n \rfloor + 1$, o que corresponde ao número de vezes que o laço desse algoritmo é executado. Portanto o custo temporal $T(n)$ de pior caso do algoritmo de busca binária é:

$$\begin{aligned} T(n) = \lfloor \log_2 n \rfloor + 1 &\Rightarrow \log_2 n \leq T(n) \leq \log_2 n + 1 \\ &\Rightarrow \log_2 n \leq T(n) \leq \log_2 n + \log_2 n \\ &\Rightarrow \log_2 n \leq T(n) \leq 2 \cdot \log_2 n \end{aligned}$$

$\therefore T(n)$ é $\theta(\log n)$ ■

Teorema 3.11: Supondo que encontrar uma chave de busca em qualquer posição de uma tabela de busca seja equiprovável, no caso médio, uma busca binária tem custo temporal $\theta(\log n)$, em que n é o número de elementos da tabela..

Prova: Usando-se uma única comparação de chaves, é possível encontrar apenas uma chave; usando-se duas comparações de chaves, é possível encontrar duas chaves; e assim por diante, de modo que, usando-se k comparações de chaves, é possível encontrar 2^{k-1} chaves. Seja $p(i)$ o número de comparações de chaves necessárias para encontrar o elemento que se encontra no índice i da tabela. Então o número esperado de comparações de chaves efetuadas para encontrar esse elemento é dado por:

$$\sum_{i=1}^{n} \frac{1}{n} p(i) = \frac{1}{n} \cdot \sum_{i=1}^{n} p(i)$$

Assumindo que o número de chaves da tabela pode ser dado por $n = 2^k - 1$, para algum $k > 0$, o número esperado de comparações efetuadas pode ser escrito como:

$$\sum_{i=1}^{n} \frac{1}{n} p(i) = \frac{1}{n} \cdot \sum_{i=1}^{k} i \cdot 2^{i-1}$$

Essa última expressão leva em conta o fato de i aparecer exatamente 2^{i-1} vezes no somatório original. Após alguma manipulação algébrica, obtém-se o seguinte número esperado de comparações:

$$\frac{1}{n} \cdot [(k-1) \cdot 2^k + 1] = \frac{(k-1) \cdot 2^k + 1}{2^k - 1} \cong k - 1$$

Como supõe-se que $n = 2^k - 1$, tem-se que número esperado de comparações de chaves é $\theta(\log n)$, que corresponde ao custo temporal esperado de uma operação de busca binária. ■

Teorema 3.12: Em qualquer caso, o custo temporal de inserção numa tabela indexada ordenada é $\theta(n)$, em que n é o número de elementos da tabela.

Prova: Para inserir um novo elemento numa tabela ordenada, é preciso, primeiro, encontrar o local de inserção usando busca sequencial. Quando a chave do elemento a ser inserido é menor do que a chave de qualquer outro elemento da tabela de busca, o custo temporal dessa operação de busca é $\theta(1)$, mas, como o novo elemento será o primeiro elemento da tabela será preciso afastar todos os elementos que já encontram na tabela para abrir espaço para o novo elemento. Portanto o custo temporal dessa operação de inserção é $\theta(n)$. Quando a chave do elemento a ser inserido é maior do que a chave de qualquer outro elemento da tabela de busca, o custo da referida operação de busca é $\theta(n)$ e a inserção em si tem custo $\theta(1)$. Em conjunto, essas duas últimas operações têm custo $\theta(n)$. No caso médio, supondo que as posições de inserção sejam equiprováveis, o custo temporal da referida operação de busca é $\theta(n)$, de acordo com o **Teorema 3.5**, e o custo temporal da inserção em si é $\theta(n)$, pois, no máximo, n elementos serão movidos. Portanto, em qualquer caso, o custo temporal de inserção numa tabela indexada ordenada é $\theta(n)$. ■

O custo temporal da função `InsereEmOrdem()` é $\theta(n)$, em que n é o tamanho da tabela. Assim a construção de uma tabela com n registros usando essa função tem custo $\theta(n^2)$. Para que o leitor possa apreciar melhor a gravidade da situação, quando a tabela de busca associada ao arquivo `CEPs.bin` é construída usando essa função no computador em que este livro foi escrito, o tempo gasto nessa operação é de cerca de 11 minutos. Em contraste, quando essa mesma tabela é construída usando a abordagem de dois passos discutida acima, o tempo gasto é apenas 1 segundo aproximadamente. Ou seja, nesse caso, a abordagem de dois passos é cerca de 660 vezes mais rápida.

Teorema 3.13: Se as chaves forem uniformemente distribuídas, uma busca por interpolação numa tabela indexada ordenada contendo n chaves usa menos de $\log \log (n + 1)$ comparações.

Prova: A prova desse teorema é bem complicada e está além do escopo deste livro.

Corolário 3.4: O custo temporal de uma busca por interpolação numa tabela indexada ordenada contendo n chaves é $\theta(\log \log n)$.

Prova: É consequência imediata do **Teorema 3.13**. ■

A função $\log \log (n + 1)$ cresce muito lentamente, de maneira que, por exemplo, se n for igual a um bilhão, $\log \log (n + 1) \cong 4{,}9$. Assim pode-se encontrar qualquer elemento de uma tabela de busca acessando muito poucos elementos em média, o que é um melhoramento substancial com relação à busca binária. Busca por interpolação, entretanto, depende muito da suposição de que as chaves são bem distribuídas na tabela de busca. Se esse não for o caso, esse algoritmo de busca pode reduzir muitas vezes o intervalo de busca por apenas um elemento, de modo que o custo temporal passa a ser $\theta(n)$. Além disso, busca por interpolação requer cálculos extras, pois ela envolve operações de multiplicação e de divisão sobre chaves, enquanto a busca binária envolve apenas operações aritméticas bem mais simples. Portanto a busca por interpolação pode ser mais lenta mesmo quando ela envolve menos comparações do que a busca binária.

Para pequenos valores de n, o custo temporal $\theta(\log n)$ de busca binária é tão próximo de $\theta(\log \log n)$ que é provável que o ganho obtido com o uso de busca por interpolação não valha a pena. Por outro lado, busca por interpolação certamente deve ser levada em consideração para tabelas grandes e para situações nas quais comparações entre chaves são dispendiosas.

A **Tabela 3–4** apresenta uma comparação resumida entre busca por interpolação e busca binária.

Busca Binária	Busca por Interpolação
A tabela deve ser implementada como uma lista indexada ordenada	Idem
Divide a tabela em partes (quase) iguais	Divide a tabela em partes desiguais
Pode ser usada quando a chave é string	Dificilmente pode ser usada quando a chave é string
Complexidade temporal: $\theta(\log n)$	Complexidade temporal: $\theta(\log \log n)$

Tabela 3–4: Comparação entre Busca por Interpolação e Busca Binária

Em geral, o problema que ocorre com métodos de busca que requerem o uso de tabelas indexadas ordenadas é como criar e manter ordenadas tais tabelas. Como foi visto nesta seção, para tabelas grandes, o custo de criação da tabela por meio de inserções que mantenham a tabela tem custo proibitivo [i.e., $\theta(n^2)$], de modo que é melhor criar a tabela desordenada e, então, ordená-la usando um método eficiente de ordenação. Além disso, o uso de tabelas desordenadas é preferível para aplicações nas quais ocorre um enorme número de operações de inserção e relativamente poucas operações de busca.

3.6 Listas com Saltos

3.6.1 Motivação e Conceito

Listas encadeadas possuem como defeito inerente só permitirem busca sequencial. Isso é, uma busca começa no início da lista e para quando a chave procurada é encontrada ou o final da lista é atingido sem encontrá-la. Ordenar elementos na lista pode acelerar a busca, mas uma busca sequencial ainda é requerida. Quer dizer, mesmo que a eficiência de uma operação de busca numa lista encadeada possa ser ligeiramente melhorada quando a lista é ordenada, a operação ainda tem custo temporal $\theta(n)$, em que n é o número de nós da lista. Mas, em 1990, o cientista da computação William Pugh (v. **Bibliografia**) apresentou uma ideia genial para superar esse obstáculo: a lista com saltos, que é uma interessante variante de lista encadeada ordenada que permite evitar esse tipo de busca sequencial.

A ideia original de Pugh consiste numa lista encadeada ordenada dividida em níveis. Todos os nós da lista fariam parte do primeiro **nível**. Então metade dos nós do primeiro nível fariam parte do segundo nível, metade dos nós do segundo nível fariam parte do terceiro nível e assim por diante até que a referida metade fosse reduzida a zero. Pugh denominou sua ideia **lista com saltos** (originalmente, *skip list*, em inglês). A denominação *lista com saltos* é decorrente do fato de nós em níveis elevados de tal lista permitirem *saltar* alguns nós em níveis inferiores durante uma operação de busca, inserção ou remoção.

A **Figura 3–10** ilustra um exemplo de lista com saltos contendo sete nós ordinários. Como se vê nessa figura, cada segundo nó da lista aponta para o nó que se encontra duas posições adiante, cada quarto nó aponta para o nó que está quatro posições à frente e assim por diante. Isso é realizado usando nós com diferentes números de ponteiros, sendo que cada nó da lista possui pelo menos um ponteiro, metade dos nós tem um ponteiro adicional, um quarto dos nós tem dois ponteiros adicionais, um oitavo dos nós tem três ponteiros adicionais e assim por diante.

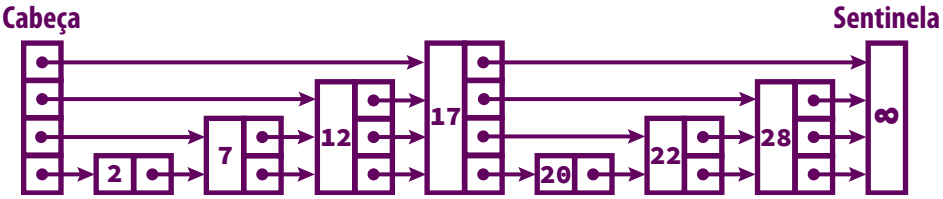


FIGURA 3–10: EXEMPLO DE LISTA COM SALTOS

O **nível de um nó** de uma lista com saltos é o número de ponteiros que ele contém menos um. Por exemplo, na lista da **Figura 3–10**, o nó cujo conteúdo é `2` possui nível `0`, enquanto o nível do nó com conteúdo `12` é `2`. Por outro lado, o **nível de uma lista com saltos** corresponde ao nível do nó de maior nível da lista, incluindo o nó cabeça. Por exemplo, a lista da **Figura 3–10** possui nível `3`. Nessa mesma figura, **cabeça** e **sentinela** são nós especiais. O nó cabeça é o nó a partir do qual começam todas as operações sobre uma lista com saltos. Por sua vez, o nó sentinela contém um valor que é maior do que o valor de qualquer chave armazenada na lista, que, na referida figura, é representada por ∞.

A lista com saltos discutida até então é uma **lista com saltos perfeita** e criar e manter uma lista dessa natureza tem custo computacional muito elevado. Uma **lista com saltos real** é aquela que relaxa a exigência de que cada nível tenha exatamente metade dos itens do nível anterior e, em vez disso, usa uma abordagem de construção probabilística, de modo que, se o número de elementos for suficientemente grande, ela seja uma lista com saltos **quase perfeita**. Mais precisamente, em vez de requerer que exatamente 50% dos elementos da lista faça parte do segundo nível da lista, uma lista com saltos real requer que essa porcentagem seja apenas aproximada. E o mesmo critério de aproximação se aplica aos seus demais níveis, de maneira que o número de nós esperados em cada nível seja aproximadamente igual àquele obtido numa lista com saltos perfeita.

Diferentemente de qualquer outra estrutura de dados vista até aqui, uma lista com saltos real é uma **estrutura de dados probabilística**, pois ela é mantida de acordo com as leis da teoria das probabilidades. Assim uma mesma sequência de inserções pode produzir diferentes estruturas dependendo dos resultados de *lançamentos de uma moed*a. Essa randomização permite algum desequilíbrio, mas, quando o número de elementos na lista é suficientemente grande, o comportamento esperado de uma lista com saltos real é igual àquele que seria obtido com listas com saltos perfeitas.

Na **Seção 3.6.3**, será mostrado que o número máximo esperado de níveis (**altura**) de uma lista com saltos contendo n nós é $\theta(\log n)$.

Busca

Uma busca numa lista com saltos inicia em seu nível mais alto e que contém o menor número de nós. Se a chave de busca não for encontrada nesse nível, tenta-se encontrá-la no nível imediatamente abaixo, e assim por diante. Desse modo, uma busca numa lista com saltos se assemelha a uma busca binária numa lista indexada ordenada. O algoritmo seguido por uma busca em lista com saltos é apresentado na **Figura 3–11**.

ALGORITMO BUSCAEMLISTACOMSALTOS

ENTRADA: Uma lista com saltos e uma chave de busca

SAÍDA: O valor associado à primeira chave da tabela que casa com a chave de busca ou um valor indicando que a chave não foi encontrada

1. Atribua à i o maior nível da lista
2. Faça p apontar para a cabeça da lista

CONTINUA

FIGURA 3–11: ALGORITMO DE BUSCA EM LISTA COM SALTOS

Algoritmo BuscaEmListaComSaltos (Continuação)

3. Enquanto $i \geq 0$ faça:
   - 3.1 Enquanto o próximo nó não é o nó sentinela e a chave desse nó é menor do que a chave de busca faça p apontar para o próximo nó
   - 3.2 Decremente i
4. Faça p apontar para o próximo nó
5. Se p estiver apontando para um nó cuja chave é igual à chave de busca, retorne o valor associado à chave do elemento encontrado
6. Retorne um valor que indique que a chave de busca não foi encontrada

Figura 3–11 (Cont.): Algoritmo de Busca em Lista com Saltos

A **Figura 3–12** ilustra o procedimento de busca na lista da **Figura 3–10** quando a chave de busca é igual a *22*. Note que se a chave de busca fosse *21*, que não se encontra na referida lista, o caminho de visitação de nós seria o mesmo.

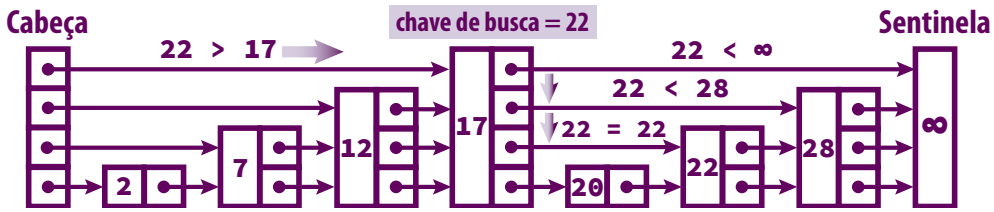


Figura 3–12: Busca numa Lista com Saltos

Inserção

Para determinar o nível de um novo nó (i.e., o número de ponteiros requeridos por ele), simulam-se lançamentos de uma moeda usando um gerador de números aleatórios. Então, enquanto os lançamentos da moeda resultam na mesma face, o nível do nó é incrementado. Quando o resultado de um lançamento resulta numa face diferente das anteriores, encerra-se esse procedimento e tem-se como resultado o nível do nó a ser inserido. Para o leitor pouco afeito à teoria das probabilidades e acostumado com estruturas de dados tradicionais, esse procedimento pode parecer obscuro e, talvez, até místico. Mas acredite, ele funciona maravilhosamente bem, principalmente quando o número de nós na lista é suficientemente grande. Em caso de relutância, o leitor é convidado a ler a **Seção 3.6.3** antes de seguir em frente. Naquela seção é feita uma análise experimental de uma implementação de lista com saltos utilizada por um cliente com cerca de 700 mil registros.

O algoritmo de inserção numa lista com saltos é apresentado na **Figura 3–13**.

Algoritmo InsereEmListaComSaltos

Entrada/Saída: Uma lista com saltos

Entrada: O conteúdo de um novo nó

Saída: Um valor informando se ocorreu inserção

1. Crie um array auxiliar de ponteiros (*aux*) para nós com um número de elementos igual ao nível máximo da lista mais 1
2. Atribua à i o maior nível da lista
3. Faça p apontar para a cabeça da lista

Continua

Figura 3–13: Algoritmo de Inserção em Lista com Saltos

ALGORITMO INSEREEMLISTACOMSALTOS (CONTINUAÇÃO)

4. Enquanto $i \geq 0$ faça:
   - 4.1 Enquanto o próximo nó não é o nó sentinela e a chave desse nó é menor do que a chave de busca faça p apontar para o próximo nó
   - 4.2 Atribua p ao elemento $aux[i]$
   - 4.3 Decremente i
5. Faça p apontar para o próximo nó
6. Se p estiver apontando para um nó cuja chave é igual à chave de busca, retorne um valor indicando que não houve inserção (pois a chave já existe e ela é considerada primária)
7. Atribua 0 ao nível (nv) do novo nó
8. Enquanto o lançamento da moeda resultar em *cara* (ou *coroa*, de acordo com sua preferência) e nv for menor do que o nível máximo da lista, incremente nv
9. Crie um novo nó com o conteúdo recebido como entrada e um número de ponteiros igual a $nv + 1$
10. Se o nível do novo nó for maior do que o nível corrente da lista:
    - 10.1 Faça os ponteiros dos níveis excedentes do novo nó apontarem para o sentinela
    - 10.2 Torne o nível corrente da lista igual ao nível do novo nó
11. Faça o novo nó apontar para os nós para os quais os antecessores dele apontavam
12. Faça os antecessores do novo nó apontarem para ele
13. Retorne um valor que indique que a inserção foi bem-sucedida

FIGURA 3–13 (CONT.): ALGORITMO DE INSERÇÃO EM LISTA COM SALTOS

Remoção

A remoção de um nó numa lista com saltos ocorre normalmente como em outras listas encadeadas, mas deve-se levar em consideração que pode ser necessário ajustar vários ponteiros para refletir a remoção adequadamente. Novamente, esse procedimento pode intrigar o leitor. Afinal, ele não é capaz de destruir uma lista com saltos? A resposta a essa dúvida é: sim, mas apenas se a lista com saltos for considerada perfeita. Mas não é isso que ocorre na prática. O algoritmo de remoção numa busca em lista com saltos é apresentado na **Figura 3–14**.

ALGORITMO REMOVEEMLISTACOMSALTOS

ENTRADA/SAÍDA: Uma lista com saltos

ENTRADA: A chave do nó a ser removido

SAÍDA: Um valor informando se ocorreu remoção

1. Crie um array auxiliar de ponteiros para nós (*aux*) com um número de elementos igual ao nível máximo da lista mais 1
2. Atribua à i o maior nível da lista
3. Faça p apontar para a cabeça da lista
4. Enquanto $i \geq 0$ faça:
   - 4.1 Enquanto o próximo nó não é o nó sentinela e a chave desse nó é menor do que a chave de busca
     - 4.1.1 Faça p apontar para o próximo nó

CONTINUA

FIGURA 3–14: ALGORITMO DE REMOÇÃO EM LISTA COM SALTOS

Algoritmo RemoveEmListaComSaltos (Continuação)

4.1.2 Armazene no array *aux* o endereço do último nó visitado no nível *i* antes que se encontre o nó a ser removido

4.2 Decremente *i*

5. Se a chave do próximo nó adiante daquele para o qual *p* está apontando for diferente da chave de busca, retorne um valor informando que a operação foi malsucedida

6. Usando o array *aux*, faça os antecessores do nó removido apontarem para o sucessor dele

7. Se for necessário, ajuste o nível da lista de modo que não exista nenhum ponteiro que emane da cabeça e termine no nó sentinela

8. Retorne um valor indicando o sucesso da operação

Figura 3–14 (Cont.): Algoritmo de Remoção em Lista com Saltos

Considere novamente a lista da **Figura 3–10** como exemplo. Nessa lista, se o procedimento de remoção normal em lista encadeada for seguido e forem removidos os nós com conteúdos *7*, *12* e *17*, a lista se tornará uma lista encadeada simples. Pior, ela estará desperdiçando memória inutilmente com cabeça, níveis e sentinela pois o uso dessa memória adicional não facilitará a busca na lista resultante. Novamente, o leitor cético é convidado a consultar a **Seção 3.6.3** para verificar que isso não ocorre na prática quando o número de chaves é suficientemente grande.

3.6.2 Implementação

Aqui, lista com saltos será implementada como uma lista encadeada circular com cabeça, de modo que a sentinela descrita acima é a própria cabeça da lista. A **Figura 3–15** mostra uma lista simplesmente encadeada com cabeça contendo três nós com conteúdos inteiros. Use essa figura como referência para facilitar o entendimento da implementação que será apresentada a seguir. Nessa figura, o fato de o conteúdo do nó cabeça conter uma interrogação denota que esse conteúdo é desconhecido (de fato, ele é irrelevante).

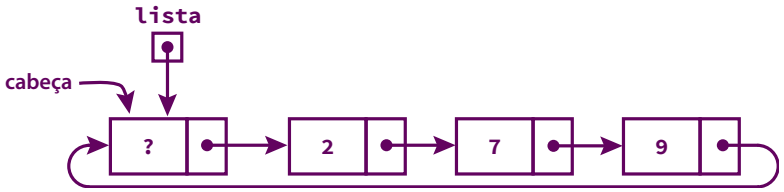


Figura 3–15: Lista Simplesmente Encadeada com Cabeça

O arquivo de dados a ser usado nessa implementação é `CEPs.bin`, cujo número de registros é suficientemente grande para mostrar que, na prática, uma lista com saltos real é muito próxima de uma lista com saltos perfeita.

Alerta: Não confunda *nível* (corrente ou máximo) de uma lista com saltos com seu *número de níveis* (corrente ou máximo). Como a numeração de níveis começa com zero, o número corrente de níveis é sempre o nível corrente mais um. E o mesmo raciocínio se aplica a número máximo de níveis e nível máximo. Esse tipo comum de confusão prejudica o entendimento da implementação a ser apresentada a seguir.

Definições de Tipos

As seguintes definições de tipos são utilizadas na implementação de listas com saltos:

```
   /* Tipo de nó do nível 0 */
typedef struct rotNoLS {
          tCEP_Ind         conteudo; /* Conteúdo efetivo    */
          struct rotNoLS **proximo;  /* Próximo nó da lista */
       } tNoListaComSalto;
```

```
   /* Tipo de lista com saltos */
typedef struct {
          tNoListaComSalto *cabeca;   /* Cabeça da lista         */
          int               nivel;    /* Nível corrente da lista */
          int               nivelMax; /* Nível máximo da lista   */
       } tListaComSaltos;
```

O tipo `tCEP_Ind` utilizado na definição acima foi definido na **Seção 3.3.2**.

Iniciação

A função `IniciaListaComSaltos()` é responsável pela iniciação de uma lista com saltos na presente implementação. A função `IniciaListaComSaltos()` não retorna nenhum valor e seus parâmetros são:

- `lista` (saída) — ponteiro para a lista que será iniciada
- `maxRegs` (entrada) — número máximo de registros esperados na lista

```
void IniciaListaComSaltos(tListaComSaltos *lista, int maxRegs)
{
   int i, nMax;

      /* Determina o nível máximo de um nó da lista */
   nMax = lista->nivelMax = (int) floor(log2(maxRegs) )+ 1;

      /* Aloca a cabeça da lista */
   lista->cabeca = malloc(sizeof(tNoListaComSalto));

      /* Se a alocação não foi possível, aborta o programa */
   ASSEGURA(lista->cabeca, "Nao foi possivel alocar cabeca");

      /* Aloca o array de ponteiros da cabeça */
   lista->cabeca->proximo = malloc((nMax + 1)*sizeof(tNoListaComSalto *));

      /* Se a alocação não foi possível, aborta o programa */
   ASSEGURA(lista->cabeca->proximo, "Nao foi possivel alocar cabeca");

      /* A lista é circular com cabeça, de modo que, inicialmente, */
      /* cada ponteiro proximo[i] aponta para a cabeça da lista    */
   for (i = 0; i <= lista->nivelMax; i++)
      lista->cabeca->proximo[i] = lista->cabeca;

   lista->nivel = 0; /* O nível corrente da lista é 0 */
}
```

Note que o nível máximo de um nó de uma lista com saltos é determinado na função `IniciaListaComSaltos()` pela instrução:

```
lista->nivelMax = (int) ceil(log2(maxRegs));
```

Matematicamente, essa instrução informa que o nível máximo é dado pelo piso do logaritmo na base dois do número máximo de registros mais um. Como a probabilidade de que uma inserção atinja um nível maior do que $\theta(1og\ n)$ é muito baixa, a escolha desse valor como nível máximo deve funcionar bem.

A **Figura 3–16** ilustra a iniciação de uma lista com saltos promovida pela função `IniciaListaComSaltos()` quando o número de registros do programa-cliente é de cerca de 700 mil registros, como é o caso do programa-cliente que lida com o arquivo `CEPs.bin` descrito no **Apêndice A**.

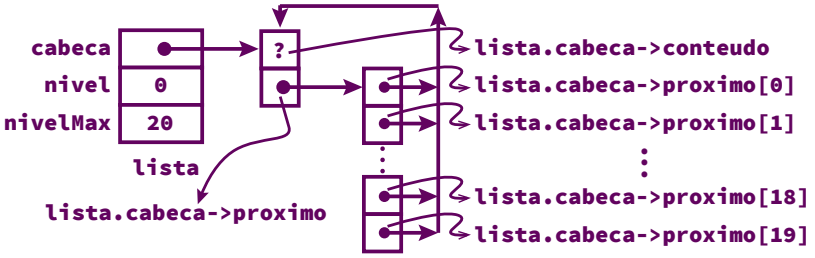


FIGURA 3–16: INICIAÇÃO DE UMA LISTA COM SALTOS

Busca

Como, aqui, a lista com saltos é implementada como uma lista circular com cabeça, a sentinela à qual se fez referência acima é a própria cabeça da lista. Isso significa que, durante uma visitação de nós de uma lista com saltos, quando um ponteiro atinge a cabeça da lista, sabe-se que a sentinela foi encontrada. Ademais, a busca numa lista com saltos é implementada conforme foi descrito acima, como mostra a função `BuscaListaComSaltos()` apresentada a seguir. Essa função usa os seguintes parâmetros:

- `lista` (entrada) — lista que será pesquisada
- `chave` (entrada) — chave de busca

A função `BuscaListaComSaltos()` retorna o valor associado à chave do nó que possui a chave de busca, se ela for encontrada. Se essa chave não for encontrada, essa função retorna `-1`.

```
int BuscaListaComSaltos(tListaComSaltos lista, tCEP chave)
{
   int               i; /* Representará um nível da lista  */
   tNoListaComSalto *ptrNo = lista.cabeca; /* Ponteiro utilizado para */
                                           /* visitar nós da lista    */

      /* A busca começa no nível mais alto da lista e termina sempre no nível 0 */
   for (i = lista.nivel; i >= 0; i--) {
         /* A busca prossegue no mesmo nível até que a sentinela ou uma  */
         /* chave maior do que ou igual a chave de busca seja encontrada */
      while (ptrNo->proximo[i] != lista.cabeca &&
             strcmp(ptrNo->proximo[i]->conteudo.chave, chave) < 0)
            /* Passa para o próximo nó no mesmo nível */
         ptrNo = ptrNo->proximo[i];
   }

   /* Neste ponto o próximo nó no nível 0 adiante de ptrNo é a cabeça da lista */
   /* ou é o primeiro nó cuja chave é maior do que ou igual à chave de busca   */

      /* Faz ptrNo apontar para a cabeça da lista ou para o primeiro */
      /* nó cuja chave é maior do que ou igual a chave de busca      */
   ptrNo = ptrNo->proximo[0];

      /* Verifica se a chave foi encontrada */
   if ( ptrNo != lista.cabeca && !strcmp(ptrNo->conteudo.chave, chave) )
      return ptrNo->conteudo.valor; /* Chave encontrada */

   return -1; /* A chave não foi encontrada */
}
```

Inserção

Como a implementação de lista com saltos aqui apresentada é baseada numa lista simplesmente encadeada com cabeça, para entender a implementação da operação de inserção numa lista com saltos, é importante que o

leitor entenda bem como funciona a inserção numa lista simplesmente encadeada com cabeça, o que é ilustrado na **Figura 3–17**. Em caso de dúvida no entendimento do mecanismo de inserção que será descrito adiante, sugere-se ao leitor que consulte essa figura.

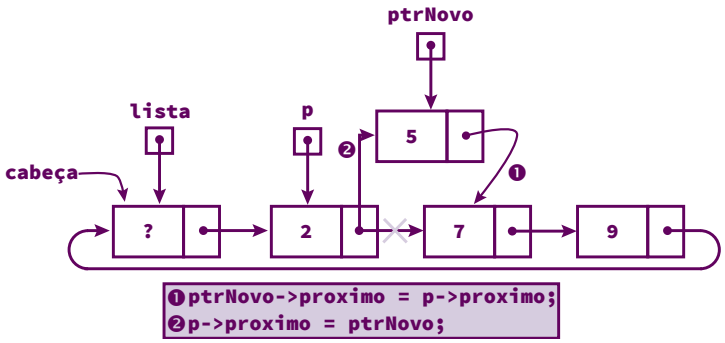


FIGURA 3–17: INSERÇÃO EM LISTA SIMPLESMENTE ENCADEADA COM CABEÇA

A função `InsereListaComSaltos()`, apresentada adiante, insere um nó numa lista encadeada com saltos. Aqui, a chave é considerada primária, de modo que um novo elemento só será inserido se sua chave for diferente de qualquer outra chave presente na lista. Essa função utiliza os seguintes parâmetros:

- `*lista` (entrada/saída) é a lista na qual será feita a inserção
- `conteudo` (entrada) é o conteúdo do nó que será inserido

A função `InsereListaComSaltos()` retorna o endereço do novo nó ou **NULL** se a chave do novo elemento já se encontra na lista.

```
tNoListaComSalto *InsereListaComSaltos(tListaComSaltos *lista, tCEP_Ind conteudo)
{
   int                i, /* Representará um nível da lista */
                      novoNivel; /* Nível do novo nó */
   tNoListaComSalto **enderecos, /* Apontará para um array de ponteiros para nós */
                     *novoNo, /* Ponteiro para o nó a ser criado */
                     *ptrNo; /* Ponteiro utilizado para visitar nós da lista */

      /* Aloca um array auxiliar de ponteiros para nós.  */
      /* Esses ponteiros apontarão para nós que terão    */
      /* ponteiros 'proximo' apontando para o novo nó.   */
   enderecos = calloc(lista->nivelMax + 1, sizeof(tNoListaComSalto *));

      /* Se o array não foi alocado aborta o programa */
   ASSEGURA(enderecos, "Nao foi possivel alocar array");

         /* Encontra a posição de inserção */

   ptrNo = lista->cabeca; /* Faz ptrNo apontar para a cabeça da lista */

      /* A busca começa sempre no nível mais alto e termina no nível 0 */
   for (i = lista->nivel; i >= 0; i--) {
         /* A busca prossegue no mesmo nível até que a sentinela ou uma  */
         /* chave maior do que ou igual a chave de busca seja encontrada */
      while ( ptrNo->proximo[i] != lista->cabeca &&
              strcmp(ptrNo->proximo[i]->conteudo.chave, conteudo.chave ) < 0)
         ptrNo = ptrNo->proximo[i]; /* Passa para o próximo nó no mesmo nível */

         /* O array enderecos[] armazena o endereço do último nó visitado */
         /* no nível i antes que se atinja o local da possível inserção   */
      enderecos[i] = ptrNo;
   }
```

```
      /* Verifica se a chave foi encontrada. Aqui, a chave */
      /* é considerada primária e não será duplicada.      */
   if ( ptrNo->proximo[0] != lista->cabeca &&
        !strcmp( ptrNo->proximo[0]->conteudo.chave, conteudo.chave ) ) {

      /* A chave foi encontrada */

      free(enderecos); /* Este array não é mais necessário */

      return NULL; /* Retorna NULL indicando que não houve inserção */
   }

      /* Determina o nível do novo nó por meio do lançamento de uma moeda;  */
      /* i.e., enquanto o resultado do lançamento for CARA, o nível do novo */
      /* nó vai aumentando (se preferir COROA, o resultado será o mesmo)    */
   for (novoNivel = 0; CaraOuCoroa() == CARA &&
                       novoNivel < lista->nivelMax; novoNivel++)
      ; /* Vazio */

   novoNo = malloc(sizeof(tNoListaComSalto));  /* Cria um nó novo */

      /* Se a alocação não ocorreu, aborta o programa */
   ASSEGURA(novoNo, "Impossivel criar um no'");

      /* Cria o array ponteiros do novo nó */
   novoNo->proximo = malloc((novoNivel + 1)*sizeof(tNoListaComSalto *));

      /* Se a alocação não ocorreu, aborta o programa */
   ASSEGURA(novoNo->proximo, "Impossivel criar um no'");

   novoNo->conteudo = conteudo; /* Armazena o conteúdo do novo nó */

      /* Se o nível do novo nó for maior do que o nível corrente da lista, */
      /* os ponteiros 'proximo' dos níveis excedentes do novo nó apontarão */
      /* para a cabeça, já que os demais nós têm níveis menores e o novo   */
      /* nó não deverá apontar para nenhum deles                           */
   if (novoNivel > lista->nivel) {
      for (i = lista->nivel + 1; i <= novoNivel; i++)
            /* O array 'enderecos' armazena os endereços dos nós que apontarão */
            /* para o novo nó. Como o nível desse nó é maior do que o nível    */
            /* corrente, o único nó com nível maior do que o nível corrente    */
            /* que apontará para ele é a cabeça.                               */
         enderecos[i] = lista->cabeca;

         /* O nível da lista passa a ser igual ao nível do novo nó */
      lista->nivel = novoNivel;
   }

      /* Ajusta os ponteiros do novo nó e dos nós que o antecedem */
   for (i = 0; i <= novoNivel; i++) {
         /* Faz o novo nó apontar para os nós para  */
         /* os quais os antecessores dele apontavam */
      novoNo->proximo[i] = enderecos[i]->proximo[i];
         /* Faz os antecessores do novo nó apontarem para ele */
      enderecos[i]->proximo[i] = novoNo;
   }

   free(enderecos); /* O array auxiliar não é mais necessário */

   return novoNo; /* Retorna o endereço do novo nó */
}
```

A função `InsereListaComSaltos()` chama a função `CaraOuCoroa()`, descrita abaixo, para simular o lançamento de uma moeda. Essa última função retorna o valor resultante do lançamento da moeda que é um dos valores do tipo enumeração `tCaraCoroa` definido como:

```
typedef enum {CARA, COROA} tCaraCoroa;
```

A função `CaraOuCoroa()` é implementada como:

```
static tCaraCoroa CaraOuCoroa(void)
{
   static int primeiraChamada = 1;

   if (primeiraChamada) {
      srand(time(NULL)); /* Inicia o gerador de números aleatórios */
      primeiraChamada = 0; /* A próxima chamada não será mais a primeira */
   }
      /* CARA corresponde a um número par e COROA corresponde */
      /* a um número ímpar, mas poderia ser o contrário       */
   return rand()%2 ? CARA : COROA;
}
```

De fato, a função `CaraOuCoroa()` e o tipo `tCaraCoroa` têm objetivos meramente didáticos. Quer dizer, eles foram incluídos nessa implementação apenas para facilitar o entendimento, mas eles são completamente dispensáveis do ponto de vista funcional. Mais precisamente, o único trecho da implementação que usa esses componentes:

```
for (novoNivel = 0;CaraOuCoroa() == CARA && novoNivel < lista->nivelMax;novoNivel++)
```

poderia ser substituído por:

```
for (novoNivel = 0; rand()%2 && novoNivel < lista->nivelMax; novoNivel++)
```

Esse último trecho de código pode ser um pouco mais difícil de compreender, mas produz o mesmo efeito que o trecho anterior, pois a probabilidade de um número sorteado ser ímpar (ou par) é a mesma probabilidade de um lançamento de moeda resultar em cara (ou coroa). Ou seja, em ambos os casos, a probabilidade é *50%*.

Na primeira chamada da função `CaraOuCoroa()`, ela alimenta o gerador de números aleatórios, de modo que os lançamentos da moeda não produzam sempre os mesmos resultados a cada execução do programa-cliente. Mas esse fato também não tem grande relevância do ponto de vista prático. Além disso, a chamada da função **srand()** poderia ter sido efetuada pela função `IniciaListaComSalto()`.

A **Figura 3–18** ilustra a execução da função `tNoCaminhoB()` logo após a instrução **for** quando o nó a ser inserido contém a chave com valor igual a `13`.

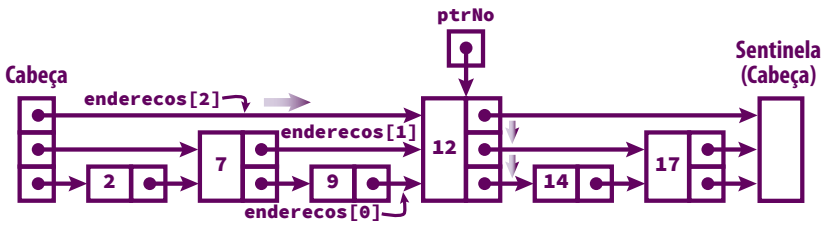


FIGURA 3–18: INSERÇÃO EM LISTA COM SALTOS 1

A **Figura 3–19** ilustra uma situação na qual o nó a ser inserido tem um nível menor do que ou igual ao nível corrente da lista, enquanto a **Figura 3–20** mostra uma situação na qual o nó a ser inserido tem um nível maior do que o nível corrente da lista. Nessas últimas figuras, o ponteiro `ptrNo` foi removido porque, deste ponto em diante ele deixa de ser relevante para a operação de inserção.

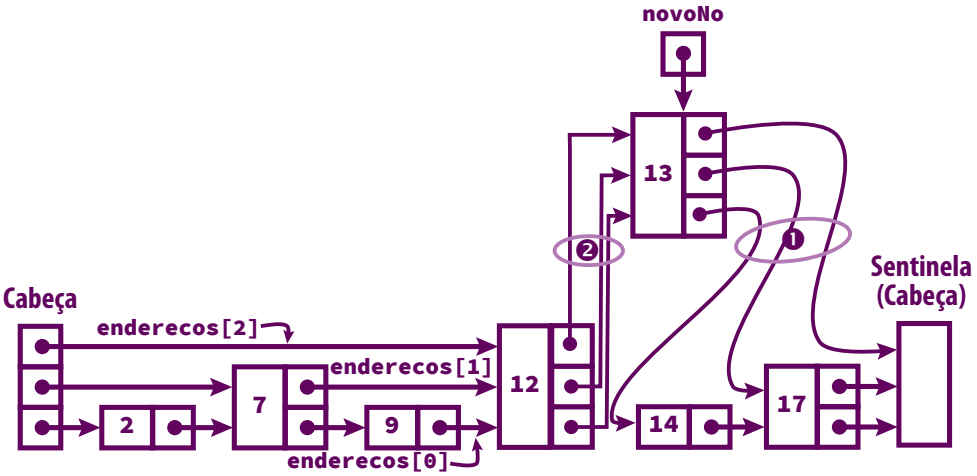


FIGURA 3–19: INSERÇÃO EM LISTA COM SALTOS 2

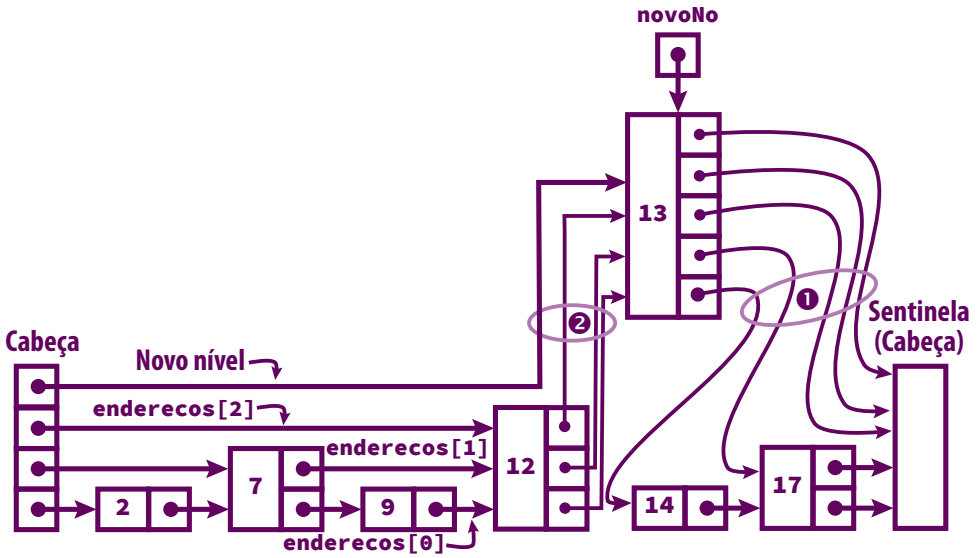


FIGURA 3–20: INSERÇÃO EM LISTA COM SALTOS 3

Remoção

Como a implementação de lista com saltos aqui apresentada é baseada numa lista simplesmente encadeada com cabeça, para entender a implementação da operação de remoção numa lista com saltos, é importante que o leitor entenda bem como funciona a remoção numa lista simplesmente encadeada com cabeça, como é ilustrado na **Figura 3–21**. Em caso de dúvida no entendimento do mecanismo de remoção que será descrito abaixo, sugere-se ao leitor que consulte essa figura.

A função `RemoveListaComSaltos()` implementa a remoção de nós numa lista com saltos. Os parâmetros dessa função são:

- `lista` (entrada/saída) — ponteiro para a lista na qual será feita a remoção
- `conteudo` (entrada) — conteúdo do nó a ser removido

Essa função retorna `0`, se a remoção for bem-sucedida, ou `1`, se o nó a ser removido não for encontrado.

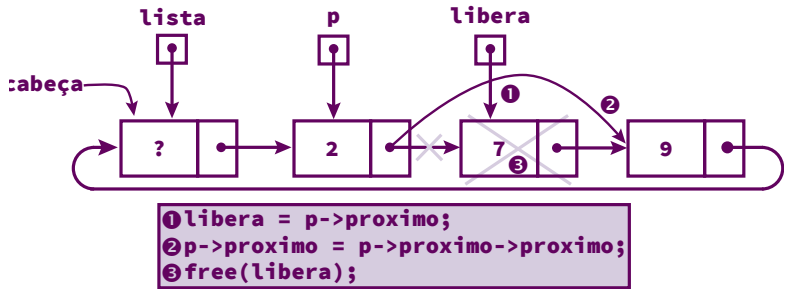


FIGURA 3–21: REMOÇÃO EM LISTA SIMPLESMENTE ENCADEADA COM CABEÇA

```
int RemoveListaComSaltos(tListaComSaltos *lista, tCEP chave)
{
   int                i; /* Representará um nível da lista */
   tNoListaComSalto **enderecos, /* Apontará para um array de ponteiro para nós */
                     *ptrNo; /* Ponteiro utilizado para visitar nós da lista */

      /* Aloca um array auxiliar de ponteiros */
   enderecos = calloc(lista->nivelMax + 1, sizeof(tNoListaComSalto *));

      /* Se o array não foi alocado aborta o programa */
   ASSEGURA(enderecos, "Nao foi possivel alocar array");

         /* Procura o elemento a ser removido */

   ptrNo = lista->cabeca; /* Faz ptrNo apontar para a cabeça da lista */

      /* A busca começa sempre no nível mais alto e termina no nível 0 */
   for (i = lista->nivel; i >= 0; i--) {
         /* A busca prossegue no mesmo nível até que a sentinela ou uma  */
         /* chave maior do que ou igual a chave de busca seja encontrada */
      while ( ptrNo->proximo[i] != lista->cabeca &&
              strcmp( ptrNo->proximo[i]->conteudo.chave, chave ) < 0 )
         ptrNo = ptrNo->proximo[i]; /* Passa para o próximo nó no mesmo nível */

         /* O array enderecos[] armazena o endereço do último nó visitado */
         /* no nível i antes  que se encontre o nó a ser removido         */
      enderecos[i] = ptrNo;
   }

      /* Se o elemento não foi encontrado,  ele não pode ser removido */
   if ( ptrNo->proximo[0] == lista->cabeca ||
        strcmp( ptrNo->proximo[0]->conteudo.chave, chave ) ) {
      return 1; /* Elemento não foi encontrado */
   }

      /* Neste ponto, ptrNo aponta para o nó antecessor daquele que será */
      /* removido. Agora, faz-se ptrNo apontar para esse último nó. Esta */
      /* instrução não é essencial, mas facilita a escrita das demais.   */
   ptrNo = ptrNo->proximo[0];

      /* Faz os antecessores do nó removido apontarem para o sucessor dele. */
      /* A atualização começa no nível 0 e encerra quando o nível da lista  */
      /* for atingido ou  quando for encontrado um nó que não aponta para o */
      /* nó a ser removido.                                                 */
   for (i = 0; i <= lista->nivel && enderecos[i]->proximo[i] == ptrNo; i++)
      enderecos[i]->proximo[i] = ptrNo->proximo[i]; /* Efetua o desvio */

      /* Libera o espaço ocupado pelo nó removido e pelo array auxiliar */
   free(ptrNo);
```

```
   free (enderecos);

      /* É possível que o nó removido tenha o mesmo nível da lista e que ele */
      /* seja o único nó com esse nível.  Logo, se esse for o caso, ajusta-  */
      /* se o nível da  lista de modo que não exista nenhum ponteiro que     */
      /* emane da cabeça e termine na própria cabeça.                        */
   while (lista->nivel > 0 &&
          lista->cabeca->proximo[lista->nivel] == lista->cabeca)
      lista->nivel--;

   return 0;
}
```

A **Figura 3–22** ilustra a remoção do nó com conteúdo igual a `12` numa lista com saltos utilizando a função `RemoveListaComSaltos()`.

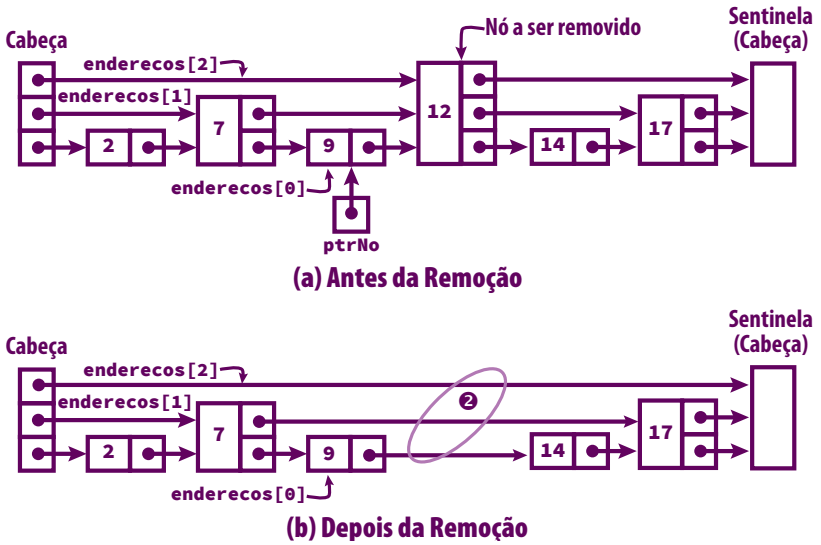


Figura 3–22: Remoção em Lista com Saltos

3.6.3 Análise

Teorema 3.14: O número máximo esperado de níveis (altura) de uma lista com saltos é $\theta(\log n)$.

Prova: Os lançamentos de moeda usados para definir o nível de um nó são independentes e a probabilidade (p) de obtenção de cara (ou coroa) é *50%*. A probabilidade de um nó obter altura a é determinada por uma distribuição geométrica, cuja função de densidade é dada por: $P(x) = (1-p)^{x-1} \cdot p$. Assim a probabilidade de um nó ter altura a é $P(X = a) = (1-p)^{a-1} \cdot p$. Nesse caso, a função de distribuição cumulativa (i.e., a probabilidade de um nó ter altura máxima igual a a) é dada por $P(X \leq a) = 1 - (1-p)^a$.

A altura esperada de uma lista com saltos é obtida calculando-se o número máximo esperado de caras resultante de n lançamentos da moeda. O resultado de cada um desses lançamentos é considerado uma variável aleatória e assume-se que essas n variáveis aleatórias sejam independentes e uniformemente distribuídas. Portanto, para obter a altura esperada da lista, deve-se resolver a seguinte equação:

$$\begin{aligned} 1/n &= P(X > a) \\ &= 1 - P(X \leq a) \\ &= 1 - [1 - (1-p)^a] \\ &= (1-p)^a \end{aligned}$$

Como $p = 1/2$, obtém-se:

$1/n = (1 - p)^a \Rightarrow 1/n = (1/2)^a \Rightarrow \log(1/n) = \log (1/2)^a \Rightarrow a = \log n$

Com base nesse último resultado, conclui-se que a altura esperada de uma lista com saltos é $\theta(\log n)$. ■

Teorema 3.15: O custo temporal esperado de uma operação de busca numa lista com saltos é $\theta(\log n)$.

Prova: Considere o caminho inverso percorrido até encontrar um nó ou descobrir que ele não faz parte da lista. Seja $C(j)$ o número esperado de nós visitados nesse caminho até que se atinja o nível j da lista. A probabilidade de subir-se um nível após visitar-se um nó nesse caminho inverso é $1/2$ (pois só existem duas alternativas com a mesma probabilidade: subir ou não subir). Portanto o valor de $C(j)$ é dado por:

	Justificativa
$C(j) = 1 +$	Nó recém-visitado
$C(j)/2 +$	Nós que serão visitados se o nó corrente *não* permite subir mais um nível com probabilidade $1/2$
$C(j - 1)/2$	Nós que serão visitados se o nó corrente permite subir mais um nível com probabilidade $1/2$

Com alguma manipulação algébrica, obtém-se a seguinte relação de recorrência para $C(j)$:

$C(j) = 1 + C(j)/2 + C(j - 1)/2 \Rightarrow C(j) = 2 + C(j - 1)$

Como $C(0) = 0$, é fácil mostrar por indução (faça isso) que a solução dessa última relação de recorrência é $C(j) = 2j$. Como, de acordo com o **Teorema 3.14**, a altura esperada de uma lista com saltos é $\theta(\log n)$, tem-se que o custo temporal esperado de uma operação de busca em tal lista é $\theta(\log n)$. ■

Teorema 3.16: O custo temporal esperado de uma operação de inserção numa lista com saltos é $\theta(\log n)$.

Prova: De acordo com o **Teorema 3.15**, o custo temporal esperado da busca pelo local de inserção é $\theta(\log n)$. Como as demais operações envolvidas numa inserção (i.e., alterações de ponteiros) têm custo $\theta(1)$, o custo temporal esperado de inserção numa lista com saltos com n elementos é $\theta(\log n)$. ■

Teorema 3.17: O custo temporal esperado de uma operação de remoção numa lista com saltos real é $\theta(\log n)$.

Prova: A prova é similar àquela do **Teorema 3.16**. ■

Teorema 3.18: O custo espacial esperado para uma lista com saltos com n elementos é $\theta(n)$.

Prova: A probabilidade de que um dado nó tenha um nível $i \geq 1$ é igual à probabilidade de obter i caras (ou coroas) consecutivas quando se lança uma moeda, que é $1/2^i$. Consequentemente, o número esperado de elementos no nível i é $n/2^i$, de maneira que o número total de elementos esperado numa lista com saltos é dado por:

$$\sum_{i=0}^{a} \frac{n}{2^i} = n \cdot \sum_{i=0}^{a} \left(\frac{1}{2}\right)^i$$

em que a é maior nível (i.e., altura) da lista. Desenvolvendo-se essa última expressão, obtém-se:

$$n \cdot \sum_{i=0}^{a} \left(\frac{1}{2}\right)^i = n \cdot \frac{\left(\frac{1}{2}\right)^{a+1} - 1}{\frac{1}{2} - 1} = 2n \cdot \left(1 - \frac{1}{2^{a+1}}\right) < 2n$$

Portanto o custo espacial esperado de uma lista com saltos é $\theta(n)$. ■

Na situação ideal (i.e., quando a lista com saltos é perfeita), o custo temporal da busca é $\theta(\log n)$. No pior caso, quando todas as sublistas estão no mesmo nível, as operações básicas com listas com saltos têm custo temporal $\theta(n)$. Entretanto essa última situação é improvável de ocorrer. Quer dizer, numa lista com saltos real (i.e., aleatória), o tempo de busca é tão bom quanto no caso ideal. Além disso, listas com saltos apresentam excelente desempenho em comparação com tabelas de busca implementadas com estruturas de dados mais sofisticadas, tais como árvores autoajustáveis ou árvores AVL. De fato, evidência experimental sugere que listas com saltos podem ser mais rápidas na prática do que árvores AVL e outras árvores de busca balanceadas, que serão discutidos no **Capítulo 4**. Portanto listas com saltos são uma alternativa viável para essas estruturas de dados mais complicadas.

O custo temporal de uma busca numa lista com saltos perfeita é $\theta(\log n)$ porque se reduz o número de elementos visitados à metade em cada nível. Espera-se que a lista com saltos randomizada funcione aproximadamente tão bem quanto uma lista com saltos perfeita com alguma probabilidade muito pequena de que isso não ocorra. A estrutura de níveis de uma lista com saltos é independente das chaves inseridas. Portanto não há sequências ruins de chave que resultem em listas com saltos degeneradas.

A **Tabela 3–5** resume os custos temporais esperados das operações sobre listas com saltos reais contendo n elementos.

Operação	Custo Temporal Esperado
Inserção	$\theta(\log n)$
Remoção	$\theta(\log n)$
Busca	$\theta(\log n)$

Tabela 3–5: Custos Temporais Esperados de Operações com Listas com Saltos

3.6.4 Resultados Experimentais

A segunda coluna da **Tabela 3–6** mostra o número de nós por nível numa lista com saltos contendo os *673580* elementos contendo chaves e índices dos registros do arquivo `CEPs.bin`. Por sua vez, a terceira coluna apresenta o número de nós por nível que teria uma lista com saltos perfeita contendo os mesmos elementos. Finalmente, a última coluna dessa tabela mostra a diferença percentual entre os números de nós real e ideal em cada nível. Note que em apenas alguns poucos níveis essa diferença é maior do que *10%*. Observe ainda que, apesar de a lista ter sido criada com a expectativa de que ela tivesse nível máximo igual a *20*, a lista real tem nível máximo igual a *15*.

Nível	Número de Nós (1)	Valor Esperado (2)	Diferença Percentual |(1) – (2)|/(2)
0	*673580*	*673580*	*0,00*
1	*336836*	*336790*	*0,01*
2	*168499*	*168395*	*0,06*
3	*84429*	*84197*	*0,28*
4	*42082*	*42099*	*0,04*
5	*21229*	*21049*	*0,86*
6	*10641*	*10525*	*1,10*
7	*5266*	*5262*	*0,08*
8	*2502*	*2631*	*4,90*

Tabela 3–6: Número de Nós por Nível numa Lista com Saltos Real

Nível	Número de Nós (1)	Valor Esperado (2)	Diferença Percentual |(1) – (2)|/(2)
9	*1188*	*1316*	*9,73*
10	*563*	*658*	*14,44*
11	*285*	*329*	*13,37*
12	*132*	*164*	*19,51*
13	*81*	*82*	*1,22*
14	*30*	*41*	*26,83*
15	*20*	*21*	*4,76*
16	*0*	*10*	*100*
17	*0*	*5*	*100*
18	*0*	*3*	*100*
19	*0*	*1*	*100*
20	*0*	*1*	*100*

Tabela 3–6: Número de Nós por Nível numa Lista com Saltos Real

A **Tabela 3–6** mostra ainda que a relação entre o número de nós num nível e o número de nós no imediatamente subsequente na lista com saltos que representa a tabela de busca do arquivo `CEPs.bin` é muito próxima de *2* até o nível *15*, o que mostra, novamente, que essa lista com saltos real se aproxima de uma lista com saltos perfeita.

3.7 Exemplos de Programação

3.7.1 Busca de Fibonacci

Preâmbulo: Uma **busca de Fibonacci** divide continuamente uma tabela de busca em subtabelas cujos tamanhos são números da sequência de Fibonacci[8]. Sendo *tab[0..n – 1]* a tabela de busca e *c* a chave de busca, um algoritmo que descreva a busca de Fibonacci é apresentado na **Figura 3–23**.

Algoritmo BuscaDeFibonacci

Entrada: Uma tabela indexada ordenada com *n* elementos e uma chave de busca

Saída: O valor associado à primeira chave da tabela que casa com a chave de busca ou um valor indicando que a chave não foi encontrada

1. Encontre o menor número de Fibonacci maior do que ou igual a *n*. Suponha que esse número seja *fib* e que *fib1* seja seu antecessor imediato. Suponha ainda que *fib2* seja o antecessor imediato de *fib1*. A **Figura 3–24 (a)** mostra a configuração inicial dessas variáveis logo antes de a busca propriamente dita ser iniciada.
2. Enquanto a tabela de busca ainda tem elementos a ser comparados:
   - **2.1** Compare *c* com a chave do último elemento do intervalo entre *0* e *fib2* (i.e., *tab[i]*) [v. **Figura 3–24 (a)**]
   - **2.2** Se *c* for igual à chave em *tab[i]*, retorne *i* [v. **Figura 3–24 (d)**]
   - **2.3** Caso contrário, se *c* for menor do que a chave em *tab[i]*, mova as três variáveis *fib*, *fib1* e *fib2* dois números de Fibonacci abaixo, indicando a eliminação de aproximadamente dois terços finais da tabela remanescente [v. **Figura 3–24 (a)**]

Continua

Figura 3–23: Algoritmo de Busca de Fibonacci

[8] Se você desconhece sequência (ou números) de Fibonacci, consulte o **Volume 1** desta obra.

Algoritmo BuscaDeFibonacci (Continuação)

2.4 Caso contrário, se *c* for maior do que a chave em *tab[i]*, mova as três variáveis de Fibonacci *fib*, *fib1* e *fib2* um número de Fibonacci abaixo e atribua *i* à variável *ajuste*. Isso indica a eliminação de aproximadamente um terço inicial da tabela remanescente [v. **Figura 3–24 (b)**]

3. Verifique se *fib1* é igual a *1*. Se for o caso, compare *c* com a chave desse elemento remanescente. Se elas forem iguais, retorne *i*.

4. Neste ponto, sabe-se que a chave não foi encontrada. Retorne *–1* para indicar esse fato.

Figura 3–13 (Cont.): Algoritmo de Busca de Fibonacci

A **Figura 3–25** apresenta outro exemplo de busca de Fibonacci. Dessa vez, diferentemente do que ocorre na **Figura 3–24**, a chave de busca não é encontrada.

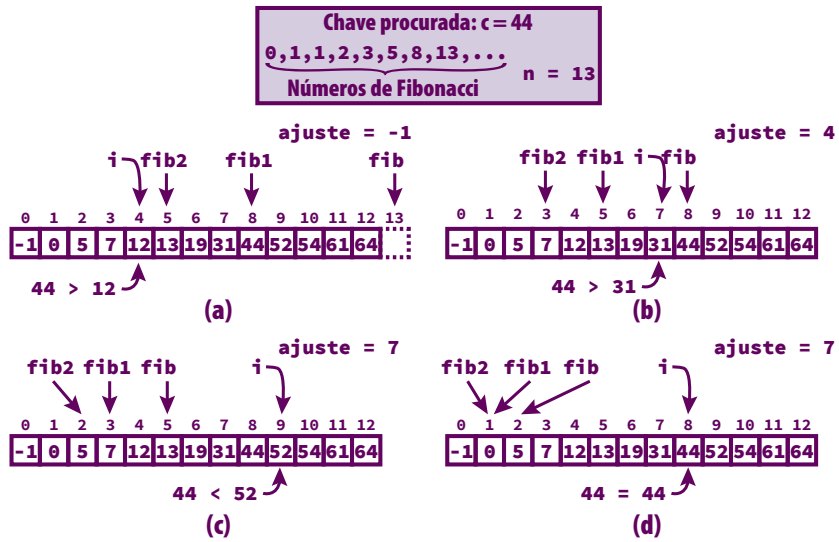


Figura 3–24: Busca de Fibonacci: Chave Encontrada

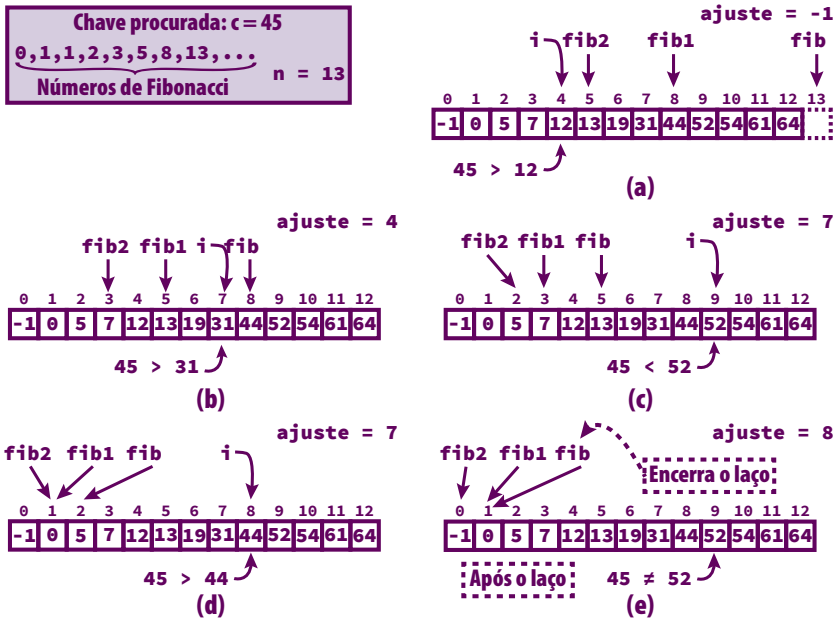


Figura 3–25: Busca de Fibonacci: Chave Não Encontrada

Problema: (a) Escreva uma função que implemente a busca de Fibonacci. (b) Apresente uma comparação entre busca binária e busca de Fibonacci.

Solução de (a): A função `BuscaFibonacciIdx()`, definida adiante, efetua uma busca de Fibonacci numa tabela de busca implementada como lista indexada ordenada. Essa função retorna o índice do primeiro elemento encontrado na tabela que casa com a chave de busca ou `-1`, se a chave não for encontrada. Os parâmetros dessa função são:

- `tab` (entrada) — tabela na qual será efetuada a busca
- `c` (entrada) — a chave de busca

```
int BuscaFibonacciIdx(tTabelaIdx tab, tCEP chave)
{
   int fib = 1, /* Número de Fibonacci maior do que */
                /* ou igual ao tamanho da tabela    */
       fib1 = 1, /* Número de Fibonacci que antecede 'fib'  */
       fib2 = 0, /* Número de Fibonacci que antecede 'fib1' */
       ajuste = -1, /* Ajuste das variáveis de Fibonacci */
       i, /* Índice do elemento que contém a chave */
          /* a ser comparada com a chave de busca  */
       teste; /* Resultado de comparação de duas chaves */

      /* Passo 1: Atribui a 'fib' o menor número de Fibonacci   */
      /* maior do que ou igual ao número de elementos da tabela */
   while (fib < tab->nElementos) {
      fib2 = fib1;
      fib1 = fib;
      fib  = fib2 + fib1;
   }

      /* Passo 2: Enquanto a tabela de busca ainda tem */
      /* elementos a ser comparados, a busca continua  */
   while (fib > 1) {
         /* Atribui a 'i' um índice válido */
      i = fib2 + ajuste < tab->nElementos - 1 ? fib2 + ajuste : tab->nElementos - 1;

         /* Passo 2.1: Compara a chave de busca com */
         /* a chave no índice 'i' da tabela         */
      teste = strcmp(chave, tab->elementos[i].chave);

      if (!teste) {    /* Passo 2.2: Se a chave de busca for igual à */
                       /* chave no índice 'i' da tabela, retorne 'i' */
         return i; /* Chave encontrada */
      } else if (teste < 0) {/* Passo 2.3: Se 'c' for menor do que a chave    */
                             /* no índice 'i',  fib, fib1 e fib2 dois números */
                             /* de Fibonacci abaixo                           */
         fib  = fib2;
         fib1 = fib1 - fib2;
         fib2 = fib - fib1;
      } else {/* Passo 2.4: Se 'c' for maior do que a chave no */
              /*  índice 'i', mova fib, fib1 e fib2 um número  */
              /* de Fibonacci abaixo e atribua 'i' a 'ajuste'  */
         fib  = fib1;
         fib1 = fib2;
         fib2 = fib - fib1;
         ajuste = i;
      }
   }
```

```
      /* Passo 3: Compara o a chave do elemento restante com a chave de busca */
   if( fib1 && !strcmp(chave, tab->elementos[ajuste + 1].chave) )
      return ajuste + 1; /* A chave foi encontrada */

   return -1; /* Passo 4: A chave não foi encontrada */
}
```

Solução de (b): Uma vantagem da busca de Fibonacci é que, em cada passo, o próximo endereço estará mais próximo do presente endereço do que seria o caso numa busca binária. Portanto essa técnica é vantajosa para buscas em meios de armazenamento que permitem apenas acesso sequencial, como, por exemplo, fita magnética. Além disso, como a busca de Fibonacci acessa elementos relativamente próximos em passos consecutivos, ela é vantajosa para listas grandes que não cabem inteiramente em memória cache. Portanto essa técnica apresenta boa localidade de referência (v. **Seção 1.5**).

Assim como a busca binária, a busca de Fibonacci tem custo temporal de $\theta(\log n)$, mas, ao contrário do que ocorre com busca binária, a prova dessa afirmação não é tão *fácil* de ser realizada.

A **Tabela 3–7** apresenta uma comparação entre a busca de Fibonacci e a busca binária.

Busca Binária	Busca de Fibonacci
A tabela deve ser implementada como uma lista indexada ordenada	Idem
Divide a tabela em partes iguais	Divide a tabela em partes desiguais
Usa operação de divisão	Usa apenas soma e subtração
Complexidade temporal: $\theta(\log n)$	Idem

Tabela 3–7: Comparação entre Busca de Fibonacci e Busca Binária

3.7.2 Busca com Chaves Secundárias

Problema: (a) Apresente as definições de tipos necessárias para implementar uma tabela de busca para o arquivo `CEPs.bin` (v. **Apêndice A**) usando uma lista simplesmente encadeada na qual as chaves sejam os nomes abreviados das localidades (i.e., o campo `nomeAbr`). (b) Escreva uma função que efetua buscas sequenciais na tabela de busca descrita no item (a) e retorna as posições de todos os registros que contêm a mesma chave de busca. (c) Escreva um trecho de programa-cliente que mostre como a função descrita no item (b) pode ser chamada e como o resultado de uma busca bem-sucedida pode ser apresentado. A seguir, um exemplo de como o referido programa apresentaria esses resultados.

```
Escolha uma das opcoes a seguir:
        (1) Acrescenta um CEP
        (2) Remove um CEP
        (3) Consulta dados de um CEP
        (4) Altera dados de um CEP
        (5) Apresenta a lista de CEPs
        (6) Encerra o programa

        >>> 3
Digite o nome (max = 40 letras):
        > Lula
```

```
>>> Foram encontrados 5 registros <<<

************ Registro No. 1 ************

Numero: 8017
UF: AM
Numero de localidade: 243
Nome abreviado: Lula
Nome: Beco Lula
Bairro inicio: 172
Bairro fim: 0
CEP: 69075446
Complemento:
Tipo de logradouro: Beco
Status de logradouro: S
Nome sem acento:: Lula
Chave DNE: 2ABKUGFOC4DID3QH
[Trecho removido]

>>> Resta ainda 1 registro.
>>> Digite 'c' para continuar ou 'e'
>>> para encerrar a apresentacao: c

************ Registro No. 5 ************

[Trecho removido]
```

Solução de (a): As definições de tipos necessárias nesta implementação são aquelas apresentadas na **Seção 3.4.2**.

Solução de (b): A função `BuscaSecundariaLSE()` apresentada a seguir retorna o endereço de uma lista encadeada contendo chaves que casam com uma chave de busca e seus parâmetros são:

- `lista` (entrada) — a tabela de busca
- `chave` (entrada) — a chave de busca

Se a busca não obtiver êxito, a função `BuscaSecundariaLSE()` retorna **NULL**.

```
tListaSE BuscaSecundariaLSE(tListaSE lista, tCEP chave)
{
   tListaSE encontrados = NULL; /* Lista contendo as chaves que */
                                /* casam com a chave de busca   */

      /* Enquanto 'lista' não assume NULL, a busca prossegue */
   while (lista) {
        /* Se ocorrer casamento entre a chave de busca e */
         /* a chave do elemento corrente, acrescenta-se  */
         /* esse elemento à lista de chaves encontradas  */
      if (!strcmp(lista->conteudo.chave, chave))
         InsereNoLSE(&encontrados, &lista->conteudo);

      lista = lista->proximo;
   }

   return encontrados;
}
```

Solução de (c): Um trecho de programa capaz de produzir o resultado mostrado acima é apresentado a seguir.

```
  /* Lê um nome de rua (chave de busca) introduzido pelo usuário */
LeNome(umNome, MAX_NOME + 1);

   /* Obtém uma lista de chaves que casam com a chave de busca */
encontrados = BuscaSecundariaLSE(lista, umNome);
```

```
   /* Se a chave for encontrada, apresenta os registros que possuem essa chave */
if (EstaVaziaLSE(encontrados)) {
   printf("\n>>> Nome nao foi encontrado\n");
} else {
   printf("\n>>> Foram encontrados %d registros <<<\n",
          ComprimentoListaSE(encontrados));
   (void) ExibeEncontrados(encontrados, stream);

      /* A lista de chaves encontradas não é */
      /* mais necessária e deve ser liberada */
   DestroiListaSE(&encontrados);
}
```

Além da função `BuscaSecundariaLSE()` definida acima, esse trecho de programa-cliente chama as seguintes funções

- `LeNome()`, que é uma função que simplesmente lê um string contendo apenas letras e espaços em branco e que tem tamanho limitado introduzido via teclado.
- `EstaVaziaLSE()`, `ComprimentoListaSE()` e `DestroiListaSE()` são funções básicas de implementação de listas simplesmente encadeadas. Essas funções foram discutidas no **Volume 1** desta obra.
- `ExibeEncontrados()` é a função responsável pela apresentação dos registros cujas chaves casam com a chave de busca e será apresentada abaixo.

A função `ExibeEncontrados()` retorna o número de registros exibidos na tela e tem como parâmetros:

- `lista` (entrada) — lista na qual cada elemento contém uma chave e o índice do respectivo registro no arquivo
- `stream` (entrada) — stream associado ao arquivo que armazena os registros

```
int ExibeEncontradosLSE(tListaSE lista, FILE *stream)
{
   tCEP_Ind *ptrElemento; /* Apontará para o conteúdo de um nó da lista */
   int       ordem = 1, /* Número de ordem de um registro */
             restantes, /* Número de registros que faltam ser apresentados */
             nRegistros = ComprimentoListaSE(lista),
             op; /* Opção do usuário */

      /* Se o número de elementos da lista for zero, */
      /* esta função não deveria ter sido chamada    */
   ASSEGURA(nRegistros, "Nao ha' registros para apresentar");

      /* Garante que a lista será escrita a partir do seu início*/
   while (ProximoListaSE(lista))
      ; /* Vazio */

      /* Apresenta um cabeçalho diferente se a lista tiver apenas um elemento */
   if (nRegistros == 1)
      printf("\n************ Registro encontrado ************\n");
   else
      printf("\n************ Registro No. %d ************\n", ordem++);

      /* Obtém o primeiro elemento da lista e exibe-o na tela */
   ptrElemento = ProximoListaSE(lista);
   ApresentaUmRegistro(ptrElemento->indiceCEP, stream);

      /* Obtém os demais elementos da lista e exibe-os */
   while ((ptrElemento = ProximoListaSE(lista))) {
      printf("\n\n************ Registro No. %d ************\n", ordem);
      ApresentaUmRegistro(ptrElemento->indiceCEP, stream);
```

```
         /* Calcula o número de registros que faltam ser apresentados */
      restantes = nRegistros - ordem;

         /* Se  restam registros a ser exibidos e o número máximo de registros   */
         /* apresentados por tela foi  atingido, solicita confirmação do usuário */
         /* antes de continuar exibindo registros                                 */
      if ((restantes > 0) && !(ordem%MAX_REGISTROS_NA_TELA)) {
         printf( "\n\n>>> Restam ainda %d registros.\n", restantes );
         printf("\n>>> Digite 'c' para continuar ou 'e'"
                "\n>>> para encerrar a apresentacao: ");
         op = LeOpcao("cCeE");

            /* Se o usuário estiver satisfeito, retorna */
            /* o número de registros que foram exibidos */
         if (op == 'e' || op == 'E')
            return ordem;
      }

      ++ordem;
   }

   putchar('\n'); /* Embelezamento */

   return nRegistros; /* Todos os registros foram exibidos */
}
```

3.7.3 Busca de Piso de Chave

Preâmbulo: O **piso** de uma chave de busca é a própria chave de busca, quando ela se encontra na tabela de busca, ou é a maior chave que é menor do que a chave de busca.

Problema: Escreva uma função que retorne o piso de uma chave recebida como parâmetro supondo que a tabela de busca é implementada como lista indexada sem ordenação.

Solução: A função `tNoCaminhoB` efetua uma busca sequencial de piso numa tabela implementada como lista indexada, como aquela vista na **Seção 3.3**. Essa função retorna o endereço do elemento que contém o piso da chave recebida como parâmetro ou **NULL** se tal piso não for encontrado.

```
tCEP_Ind *BuscaPisoIdx( tTabelaIdx tabela, tCEP chave )
{
   tCEP piso;
   int  teste, indicePiso;

   strcpy(piso, "");

   for (int i = 0; i < tabela->nElementos; ++i) {
      teste = strcmp(tabela->elementos[i].chave, chave);

      if (!teste)
         return &tabela->elementos[i]; /* Chave foi encontrada e será o piso */
      else if (teste < 0)
            /* A chave corrente é menor do que a chave de busca */
         if (strcmp(tabela->elementos[i].chave, piso) > 0) {
               /* A chave corrente é maior do que o piso */
               /* corrente e passará a ser o novo piso   */
            strcpy(piso, tabela->elementos[i].chave);
            indicePiso = i;
         }
   }
   return *piso ? &tabela->elementos[indicePiso] : NULL;
}
```

3.7.4 Busca de Intervalo

Preâmbulo: Uma **busca de intervalo** tem como objetivo encontrar as chaves que se encontram entre duas chaves.

Problema: Escreva uma função que realiza buscas de intervalo para a mesma tabela de busca usada no exemplo da **Seção 3.7.2**, mas usando chave primária em vez de chave secundária.

Solução: A função `BuscaIntervaloLSE()` retorna o endereço de uma lista encadeada contendo chaves que se encontram num intervalo especificado e seus parâmetros são:

- `tabela` (entrada) — a tabela de busca
- `chave1` (entrada) — chave que representa o limite inferior do intervalo
- `chave2` (entrada) — chave que representa o limite superior do intervalo

Se nenhuma chave for encontrada no intervalo especificado, essa função retorna **NULL**.

```
tListaSE BuscaIntervaloLSE(tListaSE tabela, tCEP chave1, tCEP chave2)
{
   tListaSE intervalo = NULL; /* Lista contendo as chaves que estão no intervalo */
   int      compara1, compara2;

   ASSEGURA(strcmp(chave1, chave2) <= 0, "Erro: Intervalo mal especificado");

      /* Compara a chave de elemento com 'chave1' e 'chave2'. Se a chave  */
      /* do elemento estiver nesse intervalo, acrescenta-se esse elemento */
      /* à lista de chaves encontradas                                    */
   while (tabela) {
         /* Compara a chave do elemento corrente com as chaves do intervalo */
      compara1 = strcmp(tabela->conteudo.chave, chave1);
      compara2 = strcmp(tabela->conteudo.chave, chave2);

         /* Verifica se a chave corrente está no intervalo */
      if (compara1 >= 0 && compara2 <= 0)
            /* A chave corrente faz parte do intervalo */
         InsereNoLSE(&intervalo, &tabela->conteudo);

      tabela = tabela->proximo;
   }

   return intervalo;
}
```

Note que, devido a similaridade entre a função `BuscaIntervaloLSE()` e a função `BuscaSecundariaLSE()` definida na **Seção 3.7.2**, essas funções são utilizadas de modo semelhante e essa utilização foi mostrada na referida seção.

3.8 Exercícios de Revisão

Definições Fundamentais (Seção 3.1)

1. Defina os seguintes termos:
   (a) Registro
   (b) Tabela de busca
   (c) Chave
   (d) Algoritmo de busca
   (e) Dicionário
2. Descreva os seguintes tipos de chaves:

(a) Chave interna
(b) Chave externa
(c) Chave primária
(d) Chave secundária

3. Qual é a diferença entre busca interna e busca externa?
4. Descreva a analogia entre busca com chave externa e uma busca efetuada num catálogo de biblioteca.
5. (a) Numa busca interna, as chaves podem ser externas? (b) Numa busca externa, as chaves podem ser internas?
6. (a) O que é busca de intervalo? (b) Que tipo de chave é utilizada nesse tipo de busca?
7. (a) O que é piso de uma chave? (b) O que é teto de uma chave?
8. Em que situação prática uma busca dedilhada seria útil?
9. (a) O que é organização de tabela de busca? (b) Quais são os tipos de organização de tabela de busca discutidos neste capítulo?
10. (a) Quais são as abordagens possíveis para remoção de elementos de uma tabela de busca? (b) De que dependem essas abordagens?

Programas Clientes (Seção 3.2)

11. No contexto de estruturas de dados, o que é um programa-cliente (ou aplicativo)?
12. Por que arquivos são usados nos exemplos apresentados neste capítulo, que lida com busca em memória principal?

Busca Sequencial Simples (Seção 3.3)

13. Em que consiste a busca sequencial?
14. Por que a busca sequencial é o único tipo de busca que pode ser usado quando a tabela de busca não possui nenhum tipo de ordenação?
15. (a) No pior caso, qual é o custo temporal de uma operação de busca sequencial? (b) Esse custo depende do fato de a tabela de busca ser indexada ou encadeada? Explique.
16. Por que a implementação de tabela de busca apresentada na **Seção 3.3** inclui a função `DestroiTabelaIdx()`?
17. (a) Por que o custo temporal da função `AcrescentaElementoIdx()` apresentada na **Seção 3.3.2** tem custo temporal $\theta(n)$? (b) Qual é o custo espacial dessa função?
18. Por que, no aplicativo descrito na **Seção 3.3.2**, a inserção de um registro no arquivo de dados é efetuada durante a operação de inserção, mas a remoção do registro associado a uma chave removida da tabela de busca é efetuada apenas ao final do programa?
19. Suponha que a chave que se procura não esteja presente numa tabela de busca indexada com *100* elementos. Qual é o número médio de comparações necessárias numa busca sequencial para determinar que a chave não está presente quando:
    (a) As chaves estão completamente desordenadas?
    (b) As chaves estão ordenadas da menor para a maior?
    (c) As chaves estão ordenadas da maior para a menor?
20. Prove o **Teorema 3.1**.
21. Prove o **Teorema 3.2**.
22. Suponha que, durante operações de busca sequencial, na metade das vezes, a chave de busca não se encontra na tabela, enquanto, quando esse não é o caso, encontrar qualquer chave da tabela é equiprovável. Mostre que o número esperado de comparações efetuadas para encontrar uma chave é $(3 \cdot n + 2)/2$.

23. Suponha que a chave que se procura *não* esteja presente numa tabela de busca com *100* elementos. Assumindo que as chaves são distribuídas uniformemente, qual é o número esperado de comparações de chaves necessárias numa busca sequencial para determinar que a chave não está presente na tabela quando:
(a) As chaves estão completamente fora de ordem?
(b) As chaves estão ordenadas da menor para a maior?
(c) As chaves estão ordenadas da maior para a menor?

24. Suponha que a chave que se procura esteja presente numa tabela de busca com *100* elementos. Assumindo que as chaves são distribuídas uniformemente, qual é o número esperado de comparações de chaves necessárias numa busca sequencial para determinar a posição do elemento quando:
(a) As chaves estão completamente fora de ordem?
(b) As chaves estão ordenadas da menor para a maior?
(c) As chaves estão ordenadas da maior para a menor?

25. Se o índice de um elemento armazenado num array de n elementos desordenados for conhecido, qual é o custo temporal de um algoritmo usado para acessar esse elemento?

26. Por que, quando uma tabela de busca desordenada é implementada usando lista indexada, um novo elemento é acrescentado ao final da lista, enquanto quando ela é implementada como lista encadeada, um novo elemento é acrescentado no início da lista?

Busca Sequencial com Movimentação (Seção 3.4)

27. (a) Explique como funciona o método de busca sequencial com movimentação para o início. (b) O que justifica essa abordagem de busca?

28. (a) Descreva o funcionamento do método de busca sequencial com transposição. (b) Em que se baseia essa abordagem de busca?

29. Por que os métodos de busca sequencial com movimentação são denominados *heurísticas*?

30. Por que o método de busca sequencial com movimentação para o início não é eficiente se a tabela de busca for implementada usando lista indexada?

31. (a) Qual é o custo temporal de uma operação de transposição? (b) Esse custo depende do fato de a tabela ser indexada ou encadeada?

32. (a) Qual é o custo temporal de uma operação de movimentação para o início? (b) Esse custo depende do fato de a tabela ser indexada ou encadeada?

33. Por que, no pior caso, o custo temporal de uma busca com movimentação é o mesmo que uma busca sem movimentação?

34. Prove o **Teorema 3.8**.

Busca Linear em Tabela Ordenada (Seção 3.5)

35. Descreva o algoritmo de busca binária.

36. Por que o algoritmo de busca binária funciona apenas se as chaves estiverem ordenadas?

37. Por que o algoritmo de busca binária nem sempre retorna o índice da primeira chave da tabela que casa com a chave de busca? Dê exemplo de tal situação.

38. Por que não faz sentido falar em busca binária quando a tabela de busca é implementada como lista encadeada?

39. Por que, quando se tem um enorme número de elementos para serem inseridos consecutivamente numa tabela que deve ser ordenada, é mais eficiente criar a tabela desordenada e depois ordená-la do que construí-la já ordenada?

40. Por que o programa-cliente discutido na **Seção 3.5** usa a função intermediária `OrdenaTabelaIdx()` para ordenar a tabela de busca, em vez de chamar **qsort()** diretamente?

41. (a) Explique o funcionamento da função `ComparaCEPs()` apresentada na **Seção 3.5**. (b) Qual é o papel desempenhado por essa função no programa discutido na referida seção?

42. Um algoritmo de busca recebe um array de inteiros ordenado em ordem crescente e uma chave de busca inteira como parâmetros, efetua uma busca binária e retorna o índice da chave de busca se ela se encontra no array. Suponha que esse algoritmo recebe o seguinte array como entrada:

-5	1	3	8	8	8	10	11	13	21	22	25
0	1	2	3	4	5	6	7	8	9	10	11

(a) Se a chave de busca recebida como parâmetro for `8`, qual será o índice retornado por esse algoritmo e quantas comparações de chaves serão efetuadas?

(b) Se a chave de busca recebida como parâmetro for `4`, quantas comparações de chaves serão efetuadas até que o algoritmo conclua que a chave não se encontra no array.

43. (a) Por que a analogia entre busca binária e a busca que uma pessoa costuma fazer num dicionário é equivocada? (b) Por que a analogia entre busca por interpolação e a mesma busca em dicionário é aceitável?

44. Matematicamente, calcular o índice central de uma tabela indexada usando a fórmula $(inf + sup)/2$ é equivalente ao uso da fórmula $inf + (sup - inf)/2$. Por que, do ponto de vista de programação, essas duas expressões não são equivalentes?

45. (a) Qual é a diferença entre busca binária e busca por interpolação? (b) Quais são as semelhanças entre esses dois tipos de busca? (c) Qual é o custo temporal da busca por interpolação?

46. Qual é o custo do algoritmo de busca por interpolação quando as chaves não são uniformemente distribuídas na tabela de busca?

47. Qual é a grande desvantagem de busca por interpolação?

48. Um algoritmo de busca recebe um array de inteiros ordenado em ordem crescente e uma chave de busca inteira como parâmetros, efetua uma busca por interpolação e retorna o índice da chave de busca se ela for encontrada no array. Suponha que esse algoritmo recebe o mesmo array da questão **42** como entrada.

(a) Se a chave de busca recebida como parâmetro for `8`, qual será o índice retornado por esse algoritmo e quantas comparações de chaves serão realizadas?

(b) Se a chave de busca recebida como parâmetro for `4`, quantas comparações de chaves serão efetuadas até que o algoritmo conclua que a chave não se encontra no array.

49. Quando é preferível usar uma tabela de busca sem ordenação em detrimento a uma tabela de busca com ordenação?

50. Complete a prova do **Teorema 3.10**.

Listas com Saltos (Seção 3.6)

51. O que é uma lista com saltos?

52. Como uma busca em lista com saltos se assemelha a uma busca binária?

53. (a) O que é uma lista com saltos perfeita? (b) O que é uma lista com saltos quase perfeita (ou real)? (c) Quando uma lista com saltos real se aproxima de uma lista com saltos perfeita?

54. Por que lista com saltos é considerada uma estrutura de dados probabilística?

55. Explique a presença de *com saltos* na denominação de listas com saltos.

56. O que são nó cabeça e nó sentinela de uma lista com saltos?

57. Por que listas com saltos perfeitas não são usadas na prática?

58. (a) Como é definido o nível de um nó de uma lista com saltos? (b) Por que qualquer nó de uma lista com saltos deve fazer parte do nível *0*?

59. Como se determina o número máximo de níveis numa lista com saltos?

60. (a) Como se simula o lançamento de uma moeda em programação? (b) Que importância essa simulação tem em implementação de listas com saltos?

61. Descreva o funcionamento da técnica de busca em lista com saltos.

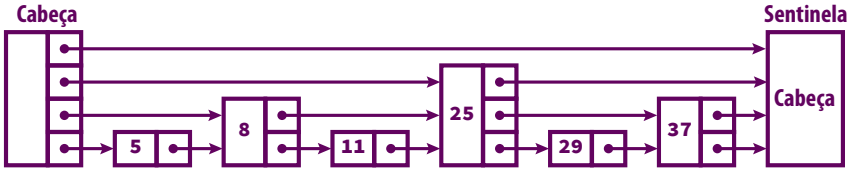
62. Descreva o mecanismo de inserção em lista com saltos.

63. Descreva o mecanismo de remoção em lista com saltos.

64. Qual é a justificativa para implementação de lista com saltos por meio de lista encadeada com cabeça circular?

65. Qual é a importância do uso da função **srand()** na implementação de lista com saltos apresentada neste capítulo?

Considere a figura a seguir, que representa uma lista com saltos, na resolução das questões de **66** a **69**.



66. Apresente a sequência de nós visitados até que uma busca encontre a chave com valor igual a *29*.

67. Mostre qual é a sequência de nós visitados até que uma busca conclua que a chave com valor igual a *35* não se encontra na lista.

68. Apresente a lista resultante da inserção de uma chave com valor igual a *45*.

69. Apresente a lista resultante da remoção da chave cujo valor é igual a *8*.

70. Como é calculado o comprimento (i.e., o número de nó) de uma lista com saltos?

71. Claramente, uma lista com saltos ocupa bem mais espaço do que uma lista simplesmente encadeada supondo que ambas apresentem o mesmo conteúdo efetivo. Então por que ambas as listas têm o mesmo custo espacial?

Exemplos de Programação (Seção 3.7)

72. Qual é o papel desempenhado pela variável `ajuste` na implementação de busca de Fibonacci apresentada na **Seção 3.7.1**?

73. Quais vantagens a busca de Fibonacci apresenta com relação à busca binária?

74. Suponha que uma função que realiza busca de Fibonacci receba o seguinte array como entrada:

0	1	3	5	7	8	10	11	13	21	22	25
0	1	2	3	4	5	6	7	8	9	10	11

(a) Mostre por meio de diagramas os passos seguidos pela referida função até encontrar a chave com valor *21*?

(b) Quantas comparações essa função efetuará até concluir que a chave com valor *4* não se encontra no array?

75. Suponha que uma tabela de busca seja indexada e ordenada. Suponha ainda que as chaves armazenadas na tabela são secundárias. Descreva um método para encontrar todos os elementos que possuem uma determinada chave com custo temporal $\theta(\log n + s)$, em que s é o número desses elementos.

76. Suponha a chave de busca de uma tabela implementada como lista indexada ordenada seja primária e que se saiba que uma dada chave se encontra nessa tabela. Como se pode encontrar o teto ou piso dessa chave com custo $\theta(\log n)$?

77. (a) Considerando que a chave não é primária na questão anterior, como se encontra o piso e o teto da referida chave? (b) Qual é o custo temporal dessa busca no pior caso? (c) Qual é o custo temporal dessa busca no melhor caso?

78. Suponha que se saiba que a chave inicial de um dado intervalo de chaves está presente numa tabela de busca implementada como lista indexada ordenada. Supondo ainda que a chave é primária, mostre como efetuar a busca por esse intervalo de chaves com custo temporal $\theta(\log n + s)$, em que s é o número de elementos da tabela que fazem parte do referido intervalo.

3.9 Exercícios de Programação

EP3.1 Considerando o programa-cliente descrito na **Seção 3.2**, escreva uma função que apresenta na tela, em forma tabular, os elementos da tabela de busca cujos elementos são estruturas do tipo `tAluno` descrito no **Apêndice A**.

EP3.2 Considerando o programa-cliente descrito na **Seção 3.2**, escreva uma função que escreve num arquivo de texto os elementos da tabela de busca cujos elementos são estruturas do tipo `tAluno` descrito no **Apêndice A**.

EP3.3 Escreva uma função, denominada `BuscaEInsere()`, que efetua busca e inserção na tabela de busca implementada na **Seção 3.3.2**. Essa função deve substituir as funções `BuscaSequencialIdx()` e `AcrescentaElementoIdx()`. Essa função recebe como parâmetro o endereço de um elemento e insere esse elemento na tabela se sua chave não for encontrada e houver espaço na tabela. A função `BuscaEInsere()` deve retornar o endereço desse elemento na tabela. O fato de não haver espaço para inserção deve ser considerada uma condição de exceção.

EP3.4 Suponha que a tabela de busca implementada na **Seção 3.5** utilize como chave o campo `nomeAbr` do tipo `tCEP` (v. **Apêndice A**) em vez de `CEP`. (Note que `nomeAbr` representa uma chave secundária). Escreva uma função para encontrar todos os elementos que possuem uma determinada chave de busca com custo temporal $\theta(\log n + s)$, em que s é o número desses elementos.

EP3.5 Implemente uma função semelhante à função `BuscaBinariaIdx()` apresentada na seção **Seção 3.5.1** que retorna o número de elementos na tabela de busca cujas chaves casam com a chave de busca.

EP3.6 Escreva uma função que retorne o teto de uma chave recebida como parâmetro supondo que a tabela é implementada como uma lista indexada. (A definição de teto de chave encontra-se na **Seção 3.1**.)

EP3.7 Uma heurística semelhante àquelas discutidas na **Seção 3.4** usa, para cada elemento da tabela de busca, um campo que conta quantas vezes o elemento foi acessado. O elemento avança em direção à frente da lista à medida que sua contagem de acesso é maior do que a contagem de seus antecessores. (a) Apresente uma definição de tipo e funções que implementem essa heurística. (b) Qual é o custo temporal dessa heurística? (c) Qual é o custo espacial dessa heurística? (d) Compare a eficiência da implementação dessa heurística quando a tabela de busca é implementada como lista indexada e lista encadeada.

EP3.8 Suponha que se saiba que a chave inicial de um dado intervalo de chaves está presente na tabela de busca implementada na **Seção 3.5**. Sob essa suposição, implemente uma busca de intervalo utilizando uma abordagem semelhante àquela usada pelo algoritmo de busca binária.

EP3.9 Escreva uma função em C que implemente o algoritmo de busca binária considerando que os elementos da tabela de busca são armazenados numa lista encadeada.

EP3.10 Dado um array ordenado de valores inteiros que, possivelmente, contém elementos duplicados, escreva uma função em C com custo temporal $\theta(\log n)$ que encontra a primeira ocorrência de um determinado valor.

EP3.11 Escreva uma função em C que implemente o algoritmo de busca por interpolação considerando que as chaves são strings não numéricos.

EP3.12 Escreva uma função que calcula o número de nós de uma lista com saltos que estão num determinado nível.

EP3.13 Escreva um trecho de programa-cliente que remove um elemento de uma tabela de busca considerando que as chaves armazenadas nessa tabela são secundárias. Esse trecho de programa deve receber uma chave de busca do usuário e proceder à busca pela chave como foi mostrado na **Seção 3.7.2**. Se a referida chave for encontrada, o programa deve exibir todos os registros que apresentam essa chave (novamente, como foi mostrado na **Seção 3.7.2**) e solicitar ao usuário que ele indique o registro que deverá ser removido. Uma função que recebe como parâmetros a tabela de busca, a lista de elementos encontrados na busca e o índice do registro a ser removido facilitará sua tarefa.

EP3.14 Reescreva o trecho de programa solicitado no exercício **EP3.13** de modo que ele permita ao usuário escolher vários registros (inclusive todos) a serem removidos.

EP3.15 Escreva um trecho de programa semelhante àquele do exercício **EP3.13** que permite ao usuário escolher um registro que será alterado.

EP3.16 (a) Escreva uma função que verifica se existem chaves duplicadas numa tabela de busca implementada como lista indexada. (b) Qual é o custo temporal dessa operação?

EP3.17 (a) Escreva uma função que verifica se existem chaves duplicadas numa tabela de busca implementada como lista encadeada. (b) Qual é o custo temporal dessa operação?

EP3.18 Suponha que uma tabela de busca é implementada como uma lista indexada desordenada com chaves secundárias. Escreva uma função que apresente a chave que aparece o maior número de vezes nessa tabela.

EP3.19 Suponha que uma tabela de busca é implementada como uma lista indexada desordenada com chaves secundárias. Escreva uma função que implemente busca dedilhada (v. **Seção 3.1**) para essa tabela. Essa função deve possuir um parâmetro que indique se ela deve continuar a última busca que ela realizou ou iniciar uma nova busca.

EP3.20 Escreva uma função que apresenta o número de nós de uma lista com saltos em cada nível.

EP3.21 Escreva uma função que apresenta relações entre números de nós de uma lista com saltos em níveis consecutivos, como mostra a **Tabela 3–6**.

EP3.22 Escreva uma função que implemente busca dedilhada similar àquela do exercício **EP3.19**, mas agora considerando que a tabela de busca é implementada como lista indexada.

EP3.23 (a) Escreva uma função que remove de um arquivo todos os registros que possuem um campo que casa com uma determinada chave de busca. Essa função deve permitir que a operação seja desfeita. (b) Escreva uma função que desfaz e refaz, indefinidas vezes, a operação descrita em (a). (c) Escreva um programa contendo uma função **main()** que ofereça um menu contendo as seguintes opções:

```
[1] Remove registro
[2] Exibe arquivo
[3] Desfaz
[4] Refaz
[5] Encerra o programa
```

O arquivo usado para testar o programa deve ser `Tudor.bin`, descrito no **Apêndice A**.