# ED – esquema para tutorial 02

## Capítulo 2 – Listas

**[página de abertura]**

Ao concluir este capítulo, você deverá ser capaz de:

- compreender o que são as listas
- identificar os tipos mais importantes de listas
- conhecer as operações básicas que se podem efetuar com as listas
- tomar contato com algumas aplicações de listas

## 2.1 Conceitos

Uma lista é uma seqüência ordenada de elementos de mesmo tipo. Por exemplo, um conjunto de fichas de clientes de uma loja, organizadas pela ordem alfabética dos nomes dos clientes. Neste fichário é possível introduzir uma nova ficha ou retirar uma velha, alterar os dados de um cliente, etc.

As operações mais básicas que se podem realizar sobre uma lista são:

- *construção da lista*: montagem da lista na memória do computador, através da leitura dos dados de um arquivo.

- *percurso por todos os elementos da lista*: percorrer todos os elementos da lista, possivelmente para obter algum dado ou fazer alguma alteração em todos os elementos da lista.

- *procura de um elemento*: procura de um elemento na lista, possivelmente para fazer alguma alteração nos dados desse elemento.

- *inserção de um elemento novo*: inserção do novo elemento na lista, tendo em conta a posição que ele deve ocupar para manter a lista ordenada.

- *remoção de um elemento da lista*: retirar um determinado elemento da lista, sendo necessário para isso, primeiramente encontrar o elemento desejado.

- *destruição da lista toda*: remover todos os elementos da lista, deixando-a vazia.

Outras operações podem ser encontradas no módulo de Aplicações

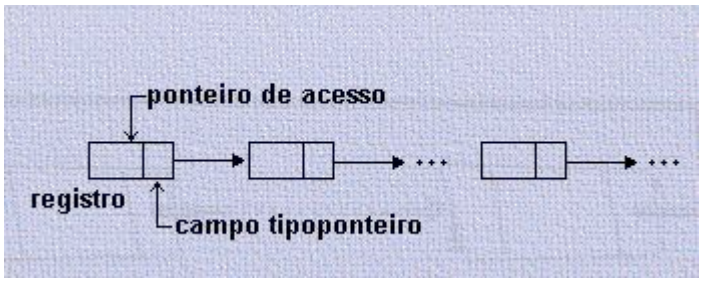
### 3.2 Implementação

Como já foi dito, as estruturas de dados são **abstratas** e portanto podem ser implementadas de diversos modos.

[estruturas abstratas devem ser imaginadas como uma forma de organizar e relacionar dados na memória do sistema, de modo a permitir certas operações sobre esses dados, independentemente do modo como essas estruturas são implementadas]

Para implementar as listas vamos utilizar a técnica de encadeamento por ponteiros. Cada elemento de uma lista é um registro que contém, entre outros campos, um campo de tipo ponteiro, que aponta para outro elemento da lista.

```
struct objeto {
  tipodado dado;
  struct objeto *prox;
};
```




A título de exemplo, vejamos como se poderia construir uma lista contendo a seqüência de caracteres 'x', 'y', 'z'. Esta função está dividida em três blocos, cada um deles responsável por acrescentar um dos elementos na lista.

**Tira Teima**

```
struct elemento {
  char x;
  struct elemento *prox;
};

struct elemento *pinicio, *p1;

main () {

  pinicio = NULL;

  p1 = malloc (sizeof (struct elemento));
  p1->x = 'x';
  p1->prox = pinicio;
  pinicio = p1;

  p1 = malloc (sizeof (struct elemento));
  p1->x = 'y';
  p1->prox = pinicio;
  pinicio = p1;

  p1 = malloc (sizeof (struct elemento));
  p1->x = 'z';
  p1->prox = pinicio;
  pinicio = p1;
}
```

Se os três blocos forem unificados em apenas um, dentro de um laço comandado pela leitura de um arquivo, temos a possibilidade de construir a lista com qualquer número de ele-

mentos. O número de elementos da lista será o número de elementos presentes no arquivo de entrada. O programa ficaria do seguinte modo:

```
struct elemento {
  char x;
  struct elemento *prox;
} ;

struct elemento *pinicio, *p1;

main () {
  FILE *arq;
  int c;

  arq = fopen ("t100.txt", "r");
  pinicio = NULL;
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      p1 = malloc (sizeof (struct elemento));
      p1->x = c;
      p1->prox = pinicio;
      pinicio = p1;
    }
  }
```

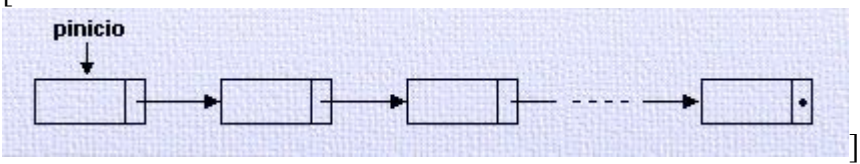
**Tira Teima**

### 3.3 Tipos Básicos de Listas

Há muitos modos de se implementar uma lista através de ponteiros, dependendo da disponibilidade de memória, eficiência desejada para os algoritmos, etc. Há dois critérios mais básicos para classificar as listas: abertas ou fechadas, com encadeamento simples ou encadeamento duplo, dando origem assim a quatro padrões básicos de listas:

**lista aberta com encadeamento simples**

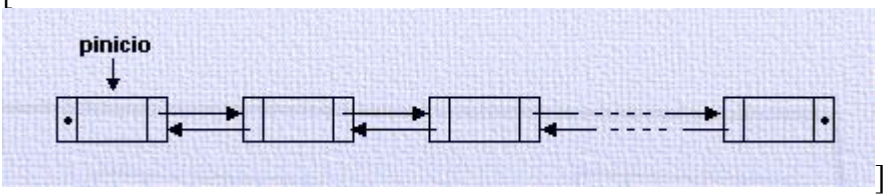
[




]

**lista aberta com encadeamento duplo**

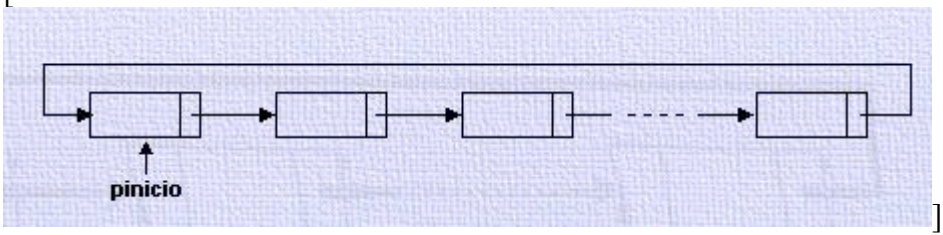
[




]

**lista fechada com encadeamento simples**

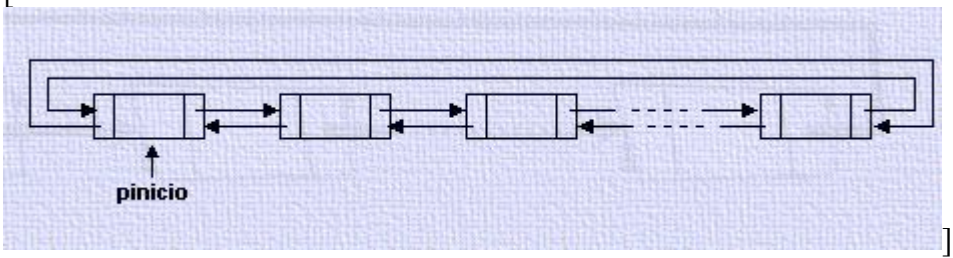
[




]

**lista fechada com encadeamento duplo**

[




]

Nas listas abertas, o último elemento tem seu campo *prox* valendo NULL (se o encadeamento for duplo, o mesmo acontece com o campo *ant* do primeiro elemento). Nas listas fechadas, o campo *prox* do último elemento aponta para o primeiro elemento (se o encadeamento for duplo, o campo *ant* do primeiro elemento aponta para o último elemento).

Nas listas com encadeamento simples, cada elemento tem apenas um campo do tipo ponteiro, apontando para o elemento seguinte na lista. Nas listas com encadeamento duplo, cada elemento tem dois ponteiros: o *prox*, apontando para o elemento seguinte, e o *ant*, apontando para o elemento anterior.

Outros padrões e modos de implementação de listas podem ser encontrados no tópico de Aplicações.

### 3.4 Algoritmos para Listas

Neste tópico veremos os algoritmos correspondentes às **operações mais básicas** [1] que podem ser efetuadas com as listas, aplicadas aos **quatro tipos mais comuns** [2]de listas. As operações de construção serão desmembradas em duas: construção com inserção do elemento no início da lista e construção com inserção no final da lista.

1 [construção da lista, percurso na lista, procura de um elemento em uma lista ordenada, inserção de um elemento em uma lista ordenada, remoção de um elemento da lista, remoção de todos os elementos da lista]

2 [ aberta com encadeamento simples, aberta com encadeamento duplo, fechada com encadeamento simples, fechada com encadeamento duplo]

É interessante notar que, em vários algoritmos, há a necessidade de se analisar separadamente a situação genérica (em geral, o elemento que interessa está no meio da lista) e as situações particulares (que costumam ser as seguintes: lista vazia, elemento no começo da lista, elemento no final da lista, lista com apenas um elemento). Sempre que possível, convém que os casos particulares sejam tratados no mesmo trecho de programa que o caso geral, tornando o programa mais sintético. Dependendo do tipo de lista escolhida, certas operações se tornam mais simples, pois os casos particulares podem ser tratados juntamente com o caso geral.

Em todas as funções que utilizam listas de encadeamento simples, o tipo de cada elemento será:

```
struct elemento {
  char dado;
  struct elemento *prox;
} ;
```

Em todas as funções que utilizam listas de encadeamento duplo, o tipo de cada elemento será:

```
struct elemento {
  char dado;
  struct elemento *prox, *ant;
} ;
```

***** acrescentar uma observação a respeito da passagem de parâmetros do tipo “endereço para ponteiro” (ver esquema 4, árvores)

#### 3.4.1 Listas Abertas com Encadeamento Simples

**a) Construção com inserção dos elementos no início da lista**

Neste caso, cada dado lido do arquivo é inserido no começo da lista. Dessa forma, a ordem dos elementos na lista fica invertida em relação à ordem original do arquivo. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista.

```
void construir1(struct elemento **pinicio) {
  FILE *arq;
```

```
  struct elemento *p1;
  char c;

  arq = fopen ("t1.txt", "r");
  *pinicio = NULL;
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      p1 = malloc (sizeof (struct elemento));
      p1->dado = c;
      p1->prox = *pinicio;
      *pinicio = p1;
    }
  }
  fclose (arq);
  return;
}
```



### b) Construção com inserção dos elementos no final da lista

Neste caso, cada dado lido do arquivo é inserido no final da lista. Dessa forma, a ordem dos elementos na lista fica idêntica à ordem original do arquivo. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista.

```
void construir2(struct elemento **pinicio) {
  FILE *arq;
  struct elemento *p1, *p2;
  char c;

  arq = fopen ("t5.txt", "r");
  *pinicio = NULL;
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      p1 = malloc (sizeof (struct elemento));
      p1->dado = c;
      p1->prox = NULL;
      if (*pinicio == NULL)
        *pinicio = p1;
      else
        p2->prox = p1;
    p2 = p1;
    }
  }
  fclose (arq);
  return;
}
```



### c) Percurso por todos os elementos da lista

Esta operação consiste em percorrer todos os elementos da lista, realizando alguma ação em cada um. Neste exemplo a ação consiste em escrever o dado na tela. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista.

```
void percorrer(struct elemento *pinicio) {
  struct elemento *p1;

  if (pinicio == NULL)
    printf ("lista vazia \n");
  else {
    p1 = pinicio;
    while (p1 != NULL) {
      printf("elemento:  %c \n", p1->dado);
      p1 = p1->prox;
    }
  }
  return;
}
```

### d) Procura de um dado na lista

Esta operação consiste na procura de um determinado dado na lista. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser procurado, denominado aqui de *chave*, e retorna um ponteiro apontando para o elemento que contem o dado. Se o dado procurado não estiver presente na lista, a função retorna o valor NULL.

```
void *procurar(struct elemento *pinicio, char chave) {
  struct elemento *p1;

  p1 = pinicio;
  while ((p1 != NULL) && (p1->dado != chave))
    p1 = p1->prox;
  return p1;
}
```

**Tira Teima**

### e) Inserção de um dado em uma lista ordenada

Esta operação consiste em inserir um novo dado na lista, no lugar que lhe corresponda. No exemplo a lista está ordenada em ordem crescente. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser inserido, denominado aqui de *dadonovo*.

```
void inserir(struct elemento **pinicio, char dadonovo){
  struct elemento *p1, *p2;

  p1 = malloc (sizeof (struct elemento));
  p1->dado = dadonovo;
  if (*pinicio == NULL)
    *pinicio = p1;
  else
    if ((*pinicio)->dado > dadonovo) {
      p1->prox = *pinicio;
      *pinicio = p1;
    }
    else {
      p2 = *pinicio;
      while ((p2->prox != NULL) && (p2->prox->dado < dadonovo))
        p2 = p2->prox;
      p1->prox = p2->prox;
      p2->prox = p1;
    }
  return;
}
```

**Tira Teima**

### f) Remoção de um dado de uma lista

Esta operação consiste em remover um determinado dado da lista. Se o dado não estiver presente, a função não faz nada. Se o dado estiver presente várias vezes na lista, todos os elementos correspondentes serão removidos. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser removido, aqui denominado *chave*.

```
void remover(struct elemento **pinicio, char chave){
  struct elemento *p1, *p2;
  int achou;

  if (*pinicio == NULL)
    printf("lista vazia \n");
  else {
    p1 = *pinicio;
    achou = 0;
```

**Tira Teima**

```
   while (p1 != NULL) {
     if (p1->dado == chave) {
       if (*pinicio == p1){
         *pinicio = (*pinicio)->prox;
         p2 = *pinicio;
         }
       else
         p2->prox = p1->prox;
       free (p1);
       p1 = p2;
       achou =1;
       }
       else {
         p2 = p1;
         p1 = p1->prox;
       }
   }
   if (!achou)
     printf ("elemento nao presente \n");
 }
 return;
}
```

### g) Remoção de toda a lista

Esta operação consiste em desfazer totalmente uma lista, removendo todos os elementos, um a um, começando pelo início da lista. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista.

```
void destruir(struct elemento **pinicio){
  struct elemento *p1;

  while (*pinicio != NULL) {
    p1 = *pinicio;
    *pinicio = (*pinicio)->prox;
    free (p1);
  }
  return;
}
```

**Tira Teima**

## 3.4.2 Listas Abertas com Encadeamento Duplo

### a) Construção com inserção dos elementos no início da lista

Neste caso, cada dado lido do arquivo é inserido no começo da lista. Dessa forma, a ordem dos elementos na lista fica invertida em relação à ordem original do arquivo. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista.

```
void construir1(struct elemento **pinicio) {
  FILE *arq;
  struct elemento *p1;
  char c;

  arq = fopen ("t1.txt", "r");
  *pinicio = NULL;
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      p1 = malloc (sizeof (struct elemento));
      p1->dado = c;
      p1->prox = *pinicio;
      if (*pinicio != NULL)
        (*pinicio)->ant = p1;
      p1->ant = NULL;
      *pinicio = p1;
    }
  }
  fclose (arq);
```

**Tira Teima**

```
  return;
}
```

### b) Construção com inserção dos elementos no final da lista

Neste caso, cada dado lido do arquivo é inserido no final da lista. Dessa forma, a ordem dos elementos na lista fica idêntica à ordem original do arquivo. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista.

```
void construir2(struct elemento **pinicio) {
  FILE *arq;
  struct elemento *p1, *p2;
  char c;

  arq = fopen ("t5.txt", "r");
  *pinicio = NULL;
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      p1 = malloc (sizeof (struct elemento));
      p1->dado = c;
      p1->prox = NULL;
      if (*pinicio == NULL) {
        *pinicio = p1;
        p1->ant = NULL;
      }
      else {
        p2->prox = p1;
        p1->ant = p2;
      }
    p2 = p1;
    }
  }
  fclose (arq);
  return;
}
```

**Tira Teima**

### c) Percurso por todos os elementos da lista

Esta operação consiste em percorrer todos os elementos da lista, realizando alguma ação em cada um. Neste exemplo a ação consiste em escrever o dado na tela. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista.

```
void percorrer(struct elemento *pinicio) {
  struct elemento *p1;

  if (pinicio == NULL)
    printf ("lista vazia \n");
  else {
    p1 = pinicio;
    while (p1 != NULL) {
      printf("elemento:  %c \n", p1->dado);
      p1 = p1->prox;
    }
    p1 = pinicio;
    while (p1->prox != NULL)
      p1 = p1->prox;
    while (p1 != NULL) {
      printf("elemento (para tras):  %c \n", p1->dado);
      p1 = p1->ant;
    }
  }
  return;
}
```

**Tira Teima**

### d) Procura de um dado na lista

Esta operação consiste na procura de um determinado dado na lista. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser procurado, denominado aqui de *chave*, e retorna um ponteiro apontando para o elemento que contem o dado. Se o dado procurado não estiver presente na lista, a função retorna o valor NULL.

```
void *procurar(struct elemento *pinicio, char chave) {
  struct elemento *p1;

  p1 = pinicio;
  while ((p1 != NULL) && (p1->dado != chave))
    p1 = p1->prox;
  return p1;
}
```



### e) Inserção de um dado em uma lista ordenada

Esta operação consiste em inserir um novo dado na lista, no lugar que lhe corresponda. No exemplo a lista está ordenada em ordem crescente. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser inserido, denominado aqui de *dadonovo*.

```
void inserir(struct elemento **pinicio, char dadonovo){
  struct elemento *p1, *p2;

  p1 = malloc (sizeof (struct elemento));
  p1->dado = dadonovo;
  if (*pinicio == NULL) {
    *pinicio = p1;
    p1->prox = NULL;
    p1->ant = NULL;
  }
  else
    if ((*pinicio)->dado > dadonovo) {
      p1->prox = *pinicio;
      p1->ant = NULL;
      (*pinicio)->ant = p1;
      *pinicio = p1;
    }
    else {
      p2 = *pinicio;
      while ((p2->prox != NULL) && (p2->prox->dado < dadonovo))
        p2 = p2->prox;
      p1->prox = p2->prox;
      p1->ant = p2;
      if (p2->prox != NULL)
        p2->prox->ant = p1;
      p2->prox = p1;
    }
  return;
}
```



### f) Remoção de um dado de uma lista

Esta operação consiste em remover um determinado dado da lista. Se o dado não estiver presente, a função não faz nada. Se o dado estiver presente várias vezes na lista, todos os elementos correspondentes serão removidos. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser removido, aqui denominado *chave*.

```
void remover(struct elemento **pinicio, char chave){
  struct elemento *p1, *p2;
  int achou;

  if (*pinicio == NULL)
    printf("lista vazia \n");
  else {
```

```
    p1 = *pinicio;
    achou = 0;
    while (p1 != NULL) {
      if (p1->dado == chave) {
        if (*pinicio == p1)
          *pinicio = (*pinicio)->prox;
        p2 = p1->prox;
        if (p1->prox != NULL)
          p1->prox->ant = p1->ant;
        if (p1->ant != NULL)
          p1->ant->prox = p1->prox;
        free (p1);
        p1 = p2;
        achou =1;
      }
      else
        p1 = p1->prox;
    }
    if (!achou)
      printf ("elemento nao presente \n");
  }
  return;
}
```

### g) Remoção de toda a lista

Esta operação consiste em desfazer totalmente uma lista, removendo todos os elementos, um a um, começando pelo início da lista. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista.

```
void removertudo(struct elemento **pinicio){
  struct elemento *p1;

  while (*pinicio != NULL) {
    p1 = *pinicio;
    *pinicio = (*pinicio)->prox;
    free (p1);
  }
  return;
}
```

**Tira Teima**

## 3.4.3 Listas Fechadas com Encadeamento Simples

### a) Construção com inserção dos elementos no início da lista

Neste caso, cada dado lido do arquivo é inserido no começo da lista. Dessa forma, a ordem dos elementos na lista fica invertida em relação à ordem original do arquivo. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista.

```
void construir1(struct elemento **pinicio) {
  FILE *arq;
  struct elemento *p1, *p2;
  char c;

  arq = fopen ("t1.txt", "r");
  *pinicio = NULL;
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      p1 = malloc (sizeof (struct elemento));
      p1->dado = c;
      if (*pinicio == NULL) {
        p1->prox = p1;
        *pinicio = p1;
      }
      else {
        p1->prox = *pinicio;
        p2 = *pinicio;
        while (p2->prox != *pinicio)
```

**Tira Teima**

```
        p2 = p2->prox;
       p2->prox = p1;
       *pinicio = p1;
     }
    }
  }
  fclose (arq);
  return;
}
```

### b) Construção com inserção dos elementos no final da lista

Neste caso, cada dado lido do arquivo é inserido no final da lista. Dessa forma, a ordem dos elementos na lista fica idêntica à ordem original do arquivo. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista.

```
void construir2(struct elemento **pinicio) {
  FILE *arq;
  struct elemento *p1, *p2;
  char c;

  arq = fopen ("t5.txt", "r");
  *pinicio = NULL;
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      p1 = malloc (sizeof (struct elemento));
      p1->dado = c;
      if (*pinicio == NULL)
        *pinicio = p1;
      else
        p2->prox = p1;
      p2 = p1;
      p1->prox = *pinicio;
    }
  }
  fclose (arq);
  return;
}
```



### c) Percurso por todos os elementos da lista

Esta operação consiste em percorrer todos os elementos da lista, realizando alguma ação em cada um. Neste exemplo a ação consiste em escrever o dado na tela. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista.

```
void percorrer(struct elemento *pinicio) {
  struct elemento *p1;

  if (pinicio == NULL)
    printf ("lista vazia \n");
  else {
    p1 = pinicio;
    do {
      printf("elemento:  %c \n", p1->dado);
      p1 = p1->prox;
    }
    while (p1 != pinicio);
  }
  return;
}
```



### d) Procura de um dado na lista

Esta operação consiste na procura de um determinado dado na lista. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser procurado,

denominado aqui de *chave*, e retorna um ponteiro apontando para o elemento que contem o dado. Se o dado procurado não estiver presente na lista, a função retorna o valor NULL.

```
void *procurar(struct elemento *pinicio, char chave) {
  struct elemento *p1;

  if (pinicio == NULL)
    p1 = NULL;
  else {
    p1 = pinicio;
    do
      p1 = p1->prox;
    while ((p1 != pinicio) && (p1->dado != chave));
    if ((p1 == pinicio) && (p1->dado != chave))
      p1 = NULL;
  }
  return p1;
}
```

**Tira Teima**

### e) Inserção de um dado em uma lista ordenada

Esta operação consiste em inserir um novo dado na lista, no lugar que lhe corresponda. No exemplo a lista está ordenada em ordem crescente. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser inserido, denominado aqui de *dadonovo*.

```
void inserir(struct elemento **pinicio, char dadonovo){
  struct elemento *p1, *p2;

  p1 = malloc (sizeof (struct elemento));
  p1->dado = dadonovo;
  if (*pinicio == NULL) {
    p1->prox = p1;
    *pinicio = p1;
    }
  else
    if ((*pinicio)->dado > dadonovo) {
      p1->prox = *pinicio;
      p2 = *pinicio;
      while (p2->prox != *pinicio)
        p2 = p2->prox;
      p2->prox = p1;
      *pinicio = p1;
    }
    else {
      p2 = *pinicio;
      while ((p2->prox != *pinicio) && (p2->prox->dado < dadonovo))
        p2 = p2->prox;
      p1->prox = p2->prox;
      p2->prox = p1;
    }
  return;
}
```

**Tira Teima**

### f) Remoção de um dado de uma lista

Esta operação consiste em remover um determinado dado da lista. Se o dado não estiver presente, a função não faz nada. Se o dado estiver presente várias vezes na lista, todos os elementos correspondentes serão removidos. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser removido, aqui denominado *chave*.

```
void remover(struct elemento **pinicio, char chave){
  struct elemento *p1, *p2;
  int achou;

  if (*pinicio == NULL)
```

**Tira Teima**

```
   printf("lista vazia \n");
  else {
   achou = 0;
   while ((*pinicio != NULL) && ((*pinicio)->dado == chave)) {
    if ((*pinicio)->prox == *pinicio) {
     free (*pinicio);
     *pinicio = NULL;
     achou = 1;
    }
    else {
     p1 = *pinicio;
     *pinicio = (*pinicio)->prox;
     p2 = *pinicio;
     while (p2->prox != p1)
       p2 = p2->prox;
     p2->prox = p1->prox;
     free (p1);
     p1 = p2;
     achou = 1;
    }
   }
   if ((*pinicio != NULL) && ((*pinicio)->prox != NULL)) {
    p1 = (*pinicio)->prox;
    while (p1 != *pinicio)
     if (p1->dado == chave) {
       p2 = *pinicio;
       while (p2->prox != p1)
         p2 = p2->prox;
       p2->prox = p1->prox;
       free (p1);
       p1 = p2->prox;
       achou = 1;
       }
      else
       p1 = p1->prox;
   }
  }
  return;
}
```

### g) Remoção de toda a lista

Esta operação consiste em desfazer totalmente uma lista, removendo todos os elementos, um a um, começando pelo início da lista. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista.

```
void removertudo(struct elemento **pinicio){
  struct elemento *p1;

  if (*pinicio != NULL) {
    p1 = (*pinicio)->prox;
    while (p1 != *pinicio) {
      (*pinicio)->prox = p1->prox;
      free (p1);
      p1 = (*pinicio)->prox;
    }
    free (*pinicio);
    *pinicio = NULL;
  }
  return;
}
```

**Tira Teima**

### 3.4.4 Listas Fechadas com Encadeamento Duplo

### a) Construção com inserção dos elementos no início da lista

Neste caso, cada dado lido do arquivo é inserido no começo da lista. Dessa forma, a ordem dos elementos na lista fica invertida em relação à ordem original do arquivo. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista.

```
void construir1(struct elemento **pinicio) {
  FILE *arq;
  struct elemento *p1;
  char c;

  arq = fopen ("t5.txt", "r");
  *pinicio = NULL;
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      p1 = malloc (sizeof (struct elemento));
      p1->dado = c;
      if (*pinicio == NULL) {
        p1->prox = p1;
        p1->ant = p1;
        *pinicio = p1;
      }
      else {
        p1->prox = *pinicio;
        p1->ant = (*pinicio)->ant;
        (*pinicio)->ant->prox = p1;
        (*pinicio)->ant = p1;
        *pinicio = p1;
      }
    }
  }
  fclose (arq);
  return;
}
```

**Tira Teima**

**b) Construção com inserção dos elementos no final da lista**

Neste caso, cada dado lido do arquivo é inserido no final da lista. Dessa forma, a ordem dos elementos na lista fica idêntica à ordem original do arquivo. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista.

```
void construir2(struct elemento **pinicio) {
  FILE *arq;
  struct elemento *p1, *p2;
  char c;

  arq = fopen ("t5.txt", "r");
  *pinicio = NULL;
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      p1 = malloc (sizeof (struct elemento));
      p1->dado = c;
      if (*pinicio == NULL) {
        p1->prox = p1;
        p1->ant = p1;
        *pinicio = p1;
      }
      else {
        p1->prox = *pinicio;
        p1->ant = p2;
        p2->prox = p1;
        (*pinicio)->ant = p1;
      }
      p2 = p1;
    }
  }
  fclose (arq);
  return;
}
```

**Tira Teima**

**c) Percurso por todos os elementos da lista**

Esta operação consiste em percorrer todos os elementos da lista, realizando alguma ação em cada um. Neste exemplo a ação consiste em escrever o dado na tela. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista.

```
void percorrer(struct elemento *pinicio) {
void percorrer(struct elemento *pinicio) {
  struct elemento *p1;

  if (pinicio == NULL)
    printf ("lista vazia \n");
  else {
    p1 = pinicio;
    do {
      printf("elemento:  %c \n", p1->dado);
      p1 = p1->prox;
    }
    while (p1 != pinicio);
    do {
      p1 = p1->ant;
      printf("elemento (para tras):  %c \n", p1->dado);
    }
    while (p1 != pinicio);
  }
  return;
}
```



### d) Procura de um dado na lista

Esta operação consiste na procura de um determinado dado na lista. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser procurado, denominado aqui de *chave*, e retorna um ponteiro apontando para o elemento que contem o dado. Se o dado procurado não estiver presente na lista, a função retorna o valor NULL.

```
void *procurar(struct elemento *pinicio, char chave) {
  struct elemento *p1;

  if (pinicio == NULL)
    p1 = NULL;
  else {
    p1 = pinicio;
    do
      p1 = p1->prox;
    while ((p1 != pinicio) && (p1->dado != chave));
    if ((p1 == pinicio) && (p1->dado != chave))
      p1 = NULL;
  }
  return p1;
}
```



### e) Inserção de um dado em uma lista ordenada

Esta operação consiste em inserir um novo dado na lista, no lugar que lhe corresponda. No exemplo a lista está ordenada em ordem crescente. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser inserido, denominado aqui de *dadonovo*.

```
void inserir(struct elemento **pinicio, char dadonovo){
  struct elemento *p1, *p2;

  p1 = malloc (sizeof (struct elemento));
  p1->dado = dadonovo;
  if (*pinicio == NULL) {
    p1->prox = p1;
    p1->ant = p1;
    *pinicio = p1;
  }
  else
```

```
   if ((*pinicio)->dado > dadonovo) {
     p1->prox = *pinicio;
     p1->ant = (*pinicio)->ant;
     (*pinicio)->ant->prox = p1;
     (*pinicio)->ant = p1;
     *pinicio = p1;
   }
    else {
     p2 = *pinicio;
     while ((p2->prox != *pinicio) && (p2->prox->dado < dadonovo))
       p2 = p2->prox;
     p1->prox = p2->prox;
     p1->ant = p2;
     p2->prox->ant = p1;
     p2->prox = p1;
   }
  return;
}
```

## f) Remoção de um dado de uma lista

Esta operação consiste em remover um determinado dado da lista. Se o dado não estiver presente, a função não faz nada. Se o dado estiver presente várias vezes na lista, todos os elementos correspondentes serão removidos. A função recebe como parâmetro o endereço do ponteiro *pinicio*, que aponta para o início da lista, e o dado a ser removido, aqui denominado *chave*.

```
void remover(struct elemento **pinicio, char chave){
  struct elemento *p1, *p2;
  int achou;

  if (*pinicio == NULL)
    printf("lista vazia \n");
  else {
    achou = 0;
    while ((*pinicio != NULL) && ((*pinicio)->dado == chave)) {
      if ((*pinicio)->prox == *pinicio) {
        free (*pinicio);
        *pinicio = NULL;
        achou = 1;
      }
      else {
        p1 = *pinicio;
        *pinicio = (*pinicio)->prox;
        p2 = *pinicio;
        while (p2->prox != p1)
          p2 = p2->prox;
        p2->prox = p1->prox;
        p1->prox->ant = p2;
        free (p1);
        p1 = p2;
        achou = 1;
      }
    }
    if ((*pinicio != NULL) && ((*pinicio)->prox != NULL)) {
      p1 = (*pinicio)->prox;
      while (p1 != *pinicio)
        if (p1->dado == chave) {
          p2 = *pinicio;
          while (p2->prox != p1)
            p2 = p2->prox;
          p2->prox = p1->prox;
          p1->prox->ant = p2;
          free (p1);
          p1 = p2->prox;
          achou = 1;
          }
        else
          p1 = p1->prox;
    }
  }
  return;
```

**Tira Teima**

```
}
```

### g) Remoção de toda a lista

Esta operação consiste em desfazer totalmente uma lista, removendo todos os elementos, um a um, começando pelo início da lista. A função recebe como parâmetro o ponteiro *pinicio*, que aponta para o início da lista.

```
void removertudo(struct elemento **pinicio){
  struct elemento *p1;

  if (*pinicio != NULL) {
    p1 = (*pinicio)->prox;
    while (p1 != *pinicio) {
      (*pinicio)->prox = p1->prox;
      free (p1);
      p1 = (*pinicio)->prox;
    }
    free (*pinicio);
    *pinicio = NULL;
  }
  return;
}
```

**Tira Teima**

## 3.5 Aplicações

Neste tópico serão apresentadas algumas aplicações de listas. Algumas dessas aplicações são apenas algoritmos aplicados sobre as listas já estudadas nos tópicos anteriores, como é o caso da *ordenação*. Outras aplicações são constituídas por novas formas de construir listas, como é o caso das *listas com descritor* e das *listas ligadas a listas*.

### 3.5.1 Ordenação dos dados em uma lista sem alterar a estrutura da lista

São apresentados dois exemplos. No primeiro, trocam-se os conteúdos dos elementos na lista, sem alterar a ordem dos elementos da lista. No segundo trocam-se as posições dos elementos na lista, através da realocação dos ponteiros. Em ambos utiliza-se o algoritmo de **Seleção Direta**.

[O algoritmo de Seleção Direta funciona do seguinte modo: percorre-se o conjunto de elementos, do primeiro ao último, selecionando-se o menor entre eles, e colocando-o na primeira posição. Em seguida percorre-se o conjunto formado pelos elementos entre a segunda posição e a última, selecionando-se o menor deles, e colocando-o na segunda posição. Assim, sucessivamente, selecionam-se os menores elementos dos conjuntos restantes, até que esse conjunto fique vazio, e todos os elementos estejam ordenados.]

**Exemplo 1**

Neste caso, trocam-se os conteúdos dos elementos na lista, sem alterar a ordem dos elementos na lista. Se os dados de cada elemento da lista não ocupam muita memória, este algoritmo é mais adequado. No exemplo, a lista é aberta com encadeamento simples. No caso o ponteiro *p1* ocupa sempre a posição em que será colocado o menor elemento do conjunto ainda desordenado. O ponteiro **p2** percorre esse conjunto, à procura do menor elemento, e **p3** aponta sempre para o menor elemento encontrado, até que a varredura termine. Em seguida são trocados os dados dos elementos apontados por **p1** e **p3.**

```
void ordenarSelecaoDireta(struct elemento *pinicio) {
 struct elemento *p1, *p2, *p3;
 char aux;

 if (pinicio != NULL)  {
   p1 = pinicio;
   while (p1->prox != NULL) {
     p3 = p1;
     p2 = p1->prox;
     while (p2 != NULL) {
       if (p3->dado > p2->dado)
         p3 = p2;
       p2 = p2->prox;
     }
     if (p1 != p3) {
       aux = p3->dado;
       p3->dado = p1->dado;
       p1->dado = aux;
       }
   p1 = p1->prox;
   }
 }
 return;
}
```

**Tira Teima**

**Exemplo 2**

Neste caso trocam-se as posições dos elementos na lista, através da realocação dos ponteiros. Este algoritmo é mais adequado na situação em que os dados da lista ocupam muito

espaço, tornando computacionalmente a realocação dos ponteiros mais barata que a troca dos dados. Neste exemplo a lista é fechada com encadeamento duplo.

```
void *ordenarAlteraPonteiros(struct elemento *pinicio) {
  struct elemento *p1, *p2, *p3, *paux;

  if (pinicio != NULL)  {
    p1 = pinicio;
    while (p1->prox != NULL) {
      p2 = p1->prox;
      while (p2 != NULL) {
        if (p1->dado > p2->dado) {
          if (p1->prox == p2) {
            p1->prox = p2->prox;
            p2->ant = p1->ant;
            if (!(p2->prox == NULL))
              p2->prox->ant = p1;
            if (!(p1->ant == NULL))
              p1->ant->prox = p2;
            p2->prox = p1;
            p1->ant = p2;
            if (pinicio == p1)
              pinicio = p2;
            paux = p1;
            p1 = p2;
            p2 = paux;
          }
          else {
            p2->ant->prox = p1;
            p1->prox->ant = p2;
            if (!(p2->prox == NULL))
              p2->prox->ant = p1;
            if (!(p1->ant == NULL))
              p1->ant->prox = p2;
            paux = p1->ant;
            p1->ant = p2->ant;
            p2->ant = paux;
            paux = p1->prox;
            p1->prox = p2->prox;
            p2->prox = paux;
            if (pinicio == p1)
              pinicio = p2;
            paux = p1;
            p1 = p2;
            p2 = paux;
          }
        }
        p2 = p2->prox;
      }
      p1 = p1->prox;
    }
  }
  return pinicio;
}
```

**Tira Teima**

### 3.5.2 Listas com descritor

As listas com descritor consistem em listas com um elemento especial, que não guarda dados como os outros elementos, mas apenas facilita o acesso à lista. Alguns algoritmos ficam simplificados através desse recurso, que pode ser aplicado a qualquer dos tipos de listas vistos anteriormente.

Apresentamos aqui dois exemplos. No primeiro, o nó descritor é do mesmo tipo que os outros nós da lista, mas não contém dados (esse nó também é conhecido como "nó bobo"). A simplificação que se consegue em alguns algoritmos decorre da facilidade no tratamento de elementos da lista colocados em posições especiais, como no começo ou no final da lista.

No segundo exemplo, o nó descritor é completamente diferente dos outros nós da lista, podendo eventualmente conter outras informações sobre a lista. No caso, o elemento descritor foi construído com dois ponteiros de acesso à lista: um apontando para o começo e outro para o final da lista. Desse modo, ficam simplificados os algoritmos que atuam no final da lista.

**Exemplo 1**

O nó descritor é do mesmo tipo que os outros nós da lista, que é aberta e com encadeamento simples. A função do exemplo faz a inserção de um novo elemento no início da lista. Tenha em conta que neste caso, uma lista vazia é formada pelo "nó bobo"; portanto *pinicio* não é NULL.

```
void *inserir(struct elemento *pinicio, char dadonovo){
  struct elemento *p1, *p2;

  p1 = malloc (sizeof (struct elemento));
  p1->dado = dadonovo;
  p1->prox = NULL;
  if (pinicio->prox == NULL)
    pinicio->prox = p1;
  else  {
    p2 = pinicio;
    while ((p2->prox != NULL) && (p2->prox->dado < dadonovo))
      p2 = p2->prox;
    p1->prox = p2->prox;
    p2->prox = p1;
  }
  return pinicio;
}
```



**Exemplo 2**

O **nó descritor** neste caso é um elemento diferente dos outros elementos da lista. Ele contém apenas dois ponteiros, um apontando para o primeiro elemento da lista e o outro apontando para o último. A lista é aberta e com encadeamento simples. A função faz a construção da lista inserindo os elementos no final.

```
[struct descritor {
  struct elemento *acessoinicio, *acessofim;
} ;
struct descritor *pdescr; ]
```

```
void *construir2() {
  FILE *arq;
  struct elemento *p1, *p2;
  struct descritor *pdescritor;
  char c;

  pdescritor = malloc (sizeof (struct descritor));
  pdescritor->acessoinicio = NULL;
  pdescritor->acessofim = NULL;
  arq = fopen ("t1.txt", "r");
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      p1 = malloc (sizeof (struct elemento));
      p1->dado = c;
      if (pdescritor->acessoinicio == NULL)
        pdescritor->acessoinicio = p1;
      else
        p2->prox = p1;
    p2 = p1;
    }
  }
```

```
  if (pdescritor->acessoinicio != NULL) {
    p1->prox = NULL;
    pdescritor->acessofim = p1;
  }
  fclose (arq);
  return pdescritor;
}
```

### 3.5.3 Listas ligadas a listas

Há situações em que, em cada elemento de uma lista, um dos campos é um ponteiro que dá acesso a outra lista. A estrutura assim formada já não é uma lista, mas um conjunto de listas ligadas a outra lista.

Neste exemplo, a primeira lista é formada por elementos que identificam centrais telefônicas, através de letras (A, B, C ...). A cada central telefônica está ligada uma lista formada por elementos que identificam números de telefone (110, 111, ...). O algoritmo usado como exemplo faz a montagem de toda a estrutura a partir de um arquivo *text*.

O arquivo contém, em cada linha, a letra identificadora da central, seguida por um número inteiro que indica quantos telefones existem nessa central, e em seguida os números dos telefones correspondentes. Note que a quantidade de telefones em cada central é variável (podendo ser também nula). O acesso a toda a estrutura é feito através de um único ponteiro (**pacesso**), que aponta para o início da lista de centrais. O acesso às listas de telefones é feito, para cada central, através do ponteiro que aponta para a lista de telefones correspondente a essa central.

---

[ conteúdo das hotwords no programa:]

```
struct elem_fone {
  int fone;
  struct elem_fone *proxf;
} ;

struct elem_central {
  char central;
  struct elem_central *proxc;
  struct elem_fone *pfone;
} ;

struct elem_central *pac;
```

---

```
void *construirMalha(struct elem_central *pacesso) {
  FILE *arq;
  struct elem_central *pc1, *pc2;
  struct elem_fone *pf1, *pf2;
  char c;
  int i,n,m;

  arq = fopen ("t3.txt", "r");
  pacesso = NULL;
  while ((c = getc (arq)) != EOF) {
    if (c != '\n'){
      pc1 = malloc (sizeof (struct elem_central));
      pc1->central = c;
```

**Tira Teima**

```
      pc1->pfone = NULL;
      pc1->proxc = NULL;
      if (pacesso == NULL)
        pacesso = pc1;
      else
        pc2->proxc = pc1;
      pc2 = pc1;
      fscanf (arq, "%d ", &n);
      for (i=1; i !=n+1; i++) {
        fscanf (arq, "%d ", &m);
        pf1 = malloc (sizeof (struct elem_fone));
        pf1->fone = m;
        pf1->proxf = NULL;
        if (pc1->pfone == NULL)
          pc1->pfone = pf1;
        else pf2->proxf = pf1;
        pf2 = pf1;
      }
    }
  }
  fclose (arq);
  return pacesso;
}
```

**[página de conclusão]**

Neste capítulo estudamos as *listas*, que são as estruturas de dados mais simples deste curso. No entanto, se forem bem compreendidas, seu conhecimento poderá resolver inúmeros problemas computacionais que exigem gerenciamento dinâmico de memória, além de preparar o aluno para aprender com mais facilidade as outras estruturas que serão apresentadas a seguir.