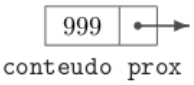
Listas encadeadas

Uma lista encadeada é uma representação de uma sequência de objetos, todos do mesmo tipo, na memória RAM (= *random access memory*) do computador. Cada elemento da sequência é armazenado em uma célula da lista: o primeiro elemento na primeira célula, o segundo na segunda, e assim por diante.

Estrutura de uma lista encadeada

Uma *lista encadeada* (= *linked list* = lista ligada) é uma sequência de *células*; cada célula contém um objeto (todos os objetos são do mesmo tipo) e o endereço da célula seguinte. Neste capítulo, suporemos que os objetos armazenados nas células são do tipo `int`. Cada célula é um registro que pode ser definido assim:

```
struct reg {
   int         conteudo;
   struct reg *prox;
};
```



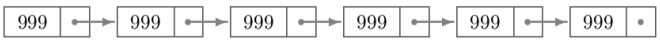
É conveniente tratar as células como um novo tipo-de-dados e atribuir um nome a esse novo tipo:

```
typedef struct reg celula;  // célula
```

Uma célula `c` e um ponteiro `p` para uma célula podem ser declarados assim:

```
celula  c;
celula *p;
```

Se `c` é uma célula então `c.conteudo` é o conteúdo da célula e `c.prox` é o endereço da próxima célula. Se `p` é o endereço de uma célula, então `p->conteudo` é o conteúdo da célula e `p->prox` é o endereço da próxima célula. Se `p` é o endereço da *última* célula da lista então `p->prox` vale `NULL` .



(A figura pode dar a falsa impressão de que as células da lista ocupam posições consecutivas na memória. Na realidade, as células podem estar espalhadas pela memória de maneira imprevisível.)

Exercícios 1

1. DECLARAÇÃO ALTERNATIVA. Verifique que a declaração de células pode também ser escrita assim:

```
typedef struct reg celula;
struct reg {
   int     conteudo;
   celula *prox;
};
```

2. ★ DECLARAÇÃO ALTERNATIVA. Verifique que a declaração de células pode também ser escrita assim:

```
typedef struct reg {
   int         conteudo;
   struct reg *prox;
} celula;
```

3. TAMANHO DE CÉLULA. Compile e execute o seguinte programa:

```
int main (void) {
   printf ("sizeof (celula) = %d\n",
sizeof (celula));
   return EXIT_SUCCESS;
}
```

Endereço de uma lista encadeada

O *endereço* de uma lista encadeada é o endereço de sua primeira célula. Se `le` é o endereço de uma lista encadeada, convém dizer simplesmente que

`le` *é uma lista encadeada*.

(Não confunda "`le`" com "`1e`".) A lista está *vazia* (ou seja, não tem célula alguma) se e somente se `le == NULL`.

Listas são animais eminentemente recursivos. Para tornar isso evidente, basta fazer a seguinte observação: se `le` é uma lista não vazia então `le->prox` também é uma lista. Muitos algoritmos sobre listas encadeadas ficam mais simples quando escritos em estilo recursivo.

EXEMPLO. A seguinte função recursiva imprime o conteúdo de uma lista encadeada `le`:

```
void imprime (celula *le) {
   if (le != NULL) {
      printf ("%d\n", le->conteudo);
      imprime (le->prox);
   }
}
```

E aqui está a versão iterativa da mesma função:

```
void imprime (celula *le) {
   celula *p;
   for (p = le; p != NULL; p = p->prox)
      printf ("%d\n", p->conteudo);
}
```

Exercícios 2

1. Escreva uma função que *conte* o número de células de uma lista encadeada. Faça duas versões: uma iterativa e uma recursiva.
2. ALTURA. A *altura* de uma célula `c` em uma lista encadeada é a distância entre `c` e o fim da lista. Mais precisamente, a altura de `c` é o número de passos do caminho que leva de `c` até a última célula da lista. Escreva uma função que calcule a altura de uma dada célula.
3. PROFUNDIDADE. A *profundidade* de uma célula `c` em uma lista encadeada é o número de passos do único caminho que vai da primeira célula da lista até `c`. Escreva uma função que calcule a profundidade de

uma dada célula.

Busca em uma lista encadeada

Veja como é fácil verificar se um objeto x pertence a uma lista encadeada, ou seja, se é igual ao conteúdo de alguma célula da lista:

```
// Esta função recebe um inteiro x e uma
// lista encadeada le de inteiros e devolve
// o endereço de uma celula que contém x.
// Se tal celula não existe, devolve NULL.

celula *
busca (int x, celula *le)
{
   celula *p;
   p = le;
   while (p != NULL && p->conteudo != x)
      p = p->prox;
   return p;
}
```

Que beleza! Nada de variáveis booleanas "de sinalização"! Além disso, a função tem comportamento correto mesmo que a lista esteja vazia.

Eis uma versão recursiva da mesma função:

```
celula *busca_r (int x, celula *le)
{
   if (le == NULL)  return NULL;
   if (le->conteudo == x)  return le;
   return busca_r (x, le->prox);
}
```

Exercícios 3

1. A função abaixo promete ter o mesmo comportamento da função `busca` acima. Critique o código.

```
celula *busca (int x, celula *le) {
   celula *p = le;
   int achou = 0;
   while (p != NULL && !achou) {
      if (p->conteudo == x) achou = 1;
      p = p->prox; }
   if (achou) return p;
   else return NULL;
}
```

2. Critique o código da seguinte variante da função `busca`.

```
celula *busca (int x, celula *le) {
   celula *p = le;
   while (p != NULL && p->conteudo != x)
      p = p->prox;
   if (p != NULL)  return p;
   else  printf ("x não está na lista\n");
}
```

3. Escreva uma função que verifique se uma lista encadeada que contém números inteiros está em ordem crescente.
4. Escreva uma função que faça uma busca em uma lista encadeada *crescente*. Faça versões recursiva e iterativa.
5. Escreva uma função que encontre uma célula com conteúdo mínimo. Faça duas versões: uma iterativa e uma recursiva.
6. Escreva uma função que verifique se duas listas encadeadas são *iguais*, ou melhor, se têm o mesmo conteúdo. Faça duas versões: uma iterativa e uma recursiva.

7. PONTO MÉDIO. Escreva uma função que receba uma lista encadeada e devolva o endereço de uma célula que esteja o mais próximo possível do meio da lista. Faça isso sem contar explicitamente o número de células da lista.

Cabeça de lista

Às vezes convém tratar a primeira célula de uma lista encadeada como um mero "marcador de início" e ignorar o conteúdo da célula. Nesse caso, dizemos que a primeira célula é a *cabeça* (= *head cell* = *dummy cell*) da lista encadeada.

Uma lista encadeada `le` com cabeça está vazia se e somente se `le->prox == NULL`. Para criar uma lista encadeada vazia com cabeça, basta dizer

```
celula *le;
le = malloc (sizeof (celula));
le->prox = NULL;
```

Para imprimir o conteúdo de uma lista encadeada `le` com cabeça, faça

```
void imprima (celula *le) {
   celula *p;
   for (p = le->prox; p != NULL; p = p->prox)
      printf ("%d\n", p->conteudo);
}
```

Exercícios 4

1. Escreva versões das funções `busca` e `busca_r` para listas encadeadas com cabeça.
2. Escreva uma função que verifique se uma lista encadeada com cabeça está em ordem crescente. (Suponha que as células contêm números inteiros.)

Inserção em uma lista encadeada

Considere o problema de inserir uma nova célula em uma lista encadeada. Suponha que quero inserir a nova célula entre a posição apontada por `p` e a posição seguinte. (É claro que isso só faz sentido se `p` é diferente de `NULL`.)

```
// Esta função insere uma nova celula
// em uma lista encadeada. A nova celula
// tem conteudo x e é inserida entre a
// celula p e a celula seguinte.
// (Supõe-se que p != NULL.)

void
insere (int x, celula *p)
{
   celula *nova;
   nova = malloc (sizeof (celula));
   nova->conteudo = x;
   nova->prox = p->prox;
   p->prox = nova;
}
```

Simples e rápido! Não é preciso movimentar células para "abrir espaço" para um nova célula, como fizemos para inserir um novo elemento em um vetor. Basta mudar os valores de alguns ponteiros. Observe que a função comporta-se corretamente mesmo quando quero inserir no *fim* da lista, isto é, quando `p->prox == NULL`. Se a lista tem cabeça, a função pode ser usada para inserir no início da lista: basta que `p` aponte para a célula-cabeça. Mas no caso de lista *sem* cabeça a função não é capaz de inserir antes da primeira célula.

O tempo que a função `insere` consome *não depende* do ponto da lista em que é feita a inserção: tanto faz inserir uma nova célula na parte inicial da lista quanto na parte final. Isso é bem diferente do que ocorre com a inserção em um vetor.

Exercícios 5

1. Por que a seguinte versão da função `insere` não funciona?

```
void insere (int x, celula *p) {
   celula nova;
   nova.conteudo = x;
   nova.prox = p->prox;
   p->prox = &nova;
}
```

2. Escreva uma função que insira uma nova célula em uma lista encadeada *sem* cabeça. (Será preciso tomar algumas decisões de projeto antes de começar a programar.)
3. Escreva uma função que faça uma *cópia* de uma lista encadeada. Faça duas versões da função: uma iterativa e uma recursiva.
4. Escreva uma função que *concatene* duas listas encadeadas (isto é, "engate" a segunda no fim da primeira). Faça duas versões: uma iterativa e uma recursiva.
5. Escreva uma função que insira uma nova célula com conteúdo `x` *imediatamente depois* da `k`-ésima célula de uma lista encadeada. Faça duas versões: uma iterativa e uma recursiva.
6. Escreva uma função que *troque de posição* duas células de uma mesma lista encadeada.
7. Escreva uma função que *inverta* a ordem das células de uma lista encadeada (a primeira passa a ser a última, a segunda passa a ser a penúltima etc.). Faça isso sem usar espaço auxiliar, apenas alterando ponteiros. Dê duas soluções: uma iterativa e uma recursiva.
8. ALOCAÇÃO DE CÉLULAS. É uma boa ideia alocar as células de uma lista encadeada uma-a-uma? (Veja observação sobre alocação de pequenos blocos de bytes no capítulo *Alocação dinâmica de memória*.) Proponha alternativas.

Remoção em uma lista encadeada

Considere o problema de remover uma certa célula de uma lista encadeada. Como especificar a célula em questão? A ideia mais óbvia é apontar para a célula que quero remover. Mas é fácil perceber que essa ideia não é boa; é melhor apontar para a célula *anterior* à que quero remover. (Infelizmente, não é possível remover a *primeira* célula usando essa convenção.)

```
// Esta função recebe o endereço p de uma
// celula de uma lista encadeada e remove
// da lista a celula p->prox. A função supõe
// que p != NULL e p->prox != NULL.

void
remove (celula *p)
{
   celula *lixo;
   lixo = p->prox;
   p->prox = lixo->prox;
   free (lixo);
}
```

Veja que maravilha! Não é preciso copiar informações de um lugar para outro, como fizemos para remover um elemento de um vetor: basta mudar o valor de um ponteiro. A função consome sempre o mesmo tempo, quer a célula a ser removida esteja perto do início da lista, quer esteja perto do fim.

Note também que a função de remoção não precisa conhecer o endereço da lista, ou seja, não precisa saber onde a lista começa.

Exercícios 6

1. Critique a seguinte versão da função `remove`:

```
void remove (celula *p, celula *le) {
   celula *lixo;
   lixo = p->prox;
   if (lixo->prox == NULL)  p->prox = NULL;
   else  p->prox = lixo->prox;
   free (lixo);
}
```

2. Suponha que queremos remover a primeira célula de uma lista encadeada `le` não vazia. Critique o seguinte fragmento de código:

```
celula **p;
p = &le;
le = le->prox;
free (*p);
```

3. Invente um jeito de remover uma célula de uma lista encadeada *sem* cabeça. (Será preciso tomar algumas decisões de projeto antes de começar a programar.)

4. Escreva uma função que *desaloque* todas as células de uma lista encadeada (ou seja, aplique a função `free` a todas as células). Estamos supondo que cada célula da lista foi originalmente alocado por `malloc`. Faça duas versões: uma iterativa e uma recursiva.

5. Problema de Josephus. Imagine uma roda de `n` pessoas numeradas de `1` a `n` no sentido horário. Começando com a pessoa de número `1`, percorra a roda no sentido horário e elimine cada `m`-ésima pessoa enquanto a roda tiver duas ou mais pessoas. Qual o número do sobrevivente?

Exercícios 7

1. Escreva uma função que copie o conteúdo de um vetor para uma lista encadeada preservando a ordem dos elementos. Faça duas versões: uma iterativa e uma recursiva.

2. Escreva uma função que copie o conteúdo de uma lista encadeada para um vetor preservando a ordem dos elementos. Faça duas versões: uma iterativa e uma recursiva.

3. União. Digamos que uma *lesc* é uma lista encadeada sem cabeça que contém uma sequência estritamente crescente de números inteiros. (Portanto, uma lesc representa um *conjunto* de números.) Escreva uma função que faça a *união* de duas lescs produzindo uma nova lesc. A lesc resultante deve ser construída com as células das duas lescs dadas.

4. Listas encadeadas sem ponteiros. Implemente uma lista encadeada sem usar endereços e ponteiros. Use dois vetores paralelos: um vetor `conteudo[0..N-1]` e um vetor `prox[0..N-1]`. Para cada `i` no conjunto `0..N-1`, o par (`conteudo[i]`, `prox[i]`) representa uma célula da lista. A célula seguinte é (`conteudo[j]`, `prox[j]`), sendo `j = prox[i]`. Escreva funções de busca, inserção e remoção para essa representação.

5. Esta questão trata de listas encadeadas que contêm strings ASCII (cada célula contém uma string). Escreva uma função que verifique se uma lista desse tipo está em ordem lexicográfica. As células são do seguinte tipo:

```
typedef struct reg {
   char *str; struct reg *prox;
} celula;
```

6. ★ Contagem de palavras. Digamos que um *texto* é um vetor de bytes, todos com valor entre 32 e 126. (Cada um desses bytes representa um caractere ASCII.) Digamos que uma *palavra* é um segmento maximal de texto que consiste apenas de letras. Escreva uma função que receba um texto e imprima uma relação de todas as palavras que ocorrem no texto juntamente com o número de ocorrências de cada palavra. Use uma lista encadeada para armazenar as palavras.

Busca e remove

Dada uma lista encadeada `le` de inteiros e um inteiro `y`, queremos remover da lista a primeira célula que contiver `y`. Se tal célula não existir, não é preciso fazer nada. Para simplificar, vamos

supor que a lista tem cabeça; assim, não será preciso mudar o endereço da lista, mesmo que a célula inicial contenha y.

```
// Esta função recebe uma lista encadeada le
// com cabeça e remove da lista a primeira
// celula que contiver y, se tal celula existir.

void
busca_e_remove (int y, celula *le)
{
   celula *p, *q;
   p = le;
   q = le->prox;
   while (q != NULL && q->conteudo != y) {
      p = q;
      q = q->prox;
   }
   if (q != NULL) {
      p->prox = q->prox;
      free (q);
   }
}
```

Para provar que o código está correto, é preciso verificar o seguinte invariante: no início de cada iteração (imediatamente antes do teste "`q != NULL`"), tem-se

```
q == p->prox
```

ou seja, `q` está um passo à frente de `p`.

Exercícios 8

1. Escreva uma função busca-e-remove para listas encadeadas *sem* cabeça.
2. Escreva uma função para remover de uma lista encadeada *todas* as células que contêm y.
3. Escreva uma função que remova a k-ésima célula de uma lista encadeada sem cabeça. Faça duas versões: uma iterativa e uma recursiva.

Busca e insere

Suponha dada uma lista encadeada le, com cabeça. Queremos inserir na lista uma nova célula com conteúdo x imediatamente *antes* da primeira célula que contém y .

```
// Esta função recebe uma lista encadeada le
// com cabeça e insere na lista uma nova celula
// imediatamente antes da primeira que contém y.
// Se nenhuma celula contém y, insere a nova
// celula no fim da lista. O conteudo da nova
// celula é x.

void
busca_e_insere (int x, int y, celula *le)
{
   celula *p, *q, *nova;
   nova = malloc (sizeof (celula));
   nova->conteudo = x;
   p = le;
   q = le->prox;
   while (q != NULL && q->conteudo != y) {
      p = q;
      q = q->prox;
   }
   nova->prox = q;
   p->prox = nova;
}
```

Exercícios 9

1. Escreva uma função busca-e-insere para listas encadeadas *sem* cabeça.

Outros tipos de listas

Uma vez entendidas as listas encadeadas básicas, você pode inventar muitos outros tipos de listas encadeadas.

Por exemplo, você pode construir uma lista encadeada *circular*, em que a última célula aponta para a primeira. O endereço de uma tal lista é o endereço de qualquer uma de suas células. Você pode também ter uma lista *duplamente encadeada*: cada célula contém o endereço da célula anterior e o endereço da célula seguinte.

Pense nas seguintes questões, apropriadas para qualquer tipo de lista encadeada. Convém ter uma célula-cabeça e/ou uma célula-rabo? Em que condições a lista está vazia? Como remover a célula apontada por `p`? Idem para a célula seguinte à apontada por `p`? Idem para a célula anterior à apontada por `p`? Como inserir uma nova célula entre a célula apontada por `p` e a anterior? Idem entre `p` e a seguinte?

Exercícios 10

1. Descreva, em linguagem C, a estrutura de uma célula de uma lista duplamente encadeada.
2. Escreva uma função que remova de uma lista duplamente encadeada a célula apontada por `p`. Que dados sua função recebe? Que coisa devolve?
3. Escreva uma função que insira uma nova célula com conteúdo `x` em uma lista duplamente encadeada logo após a célula apontada por `p`. Que dados sua função recebe? Que coisa devolve?

Veja o verbete *Linked list* na Wikipedia

Atualizado em 2018-08-25
https://www.ime.usp.br/~pf/algoritmos/
Paulo Feofiloff
DCC-IME-USP