REVISÃO

- Ordenação;
- Busca (Hash);
- Arvores;
- Grafos;
- Fila de prioridade;

Avaliações
- 3 somativas (MAS)
- m formativas (MAF)
- 1 trabalho (T)
- frequencia

$M_F = 0,85 \times M_{AS} + 0,15 \times M_{AF} + 0,1 \times T$

APROVAÇÃO
- MF ≥ 5;
- 75% de frequência (≤ 16 faltas)

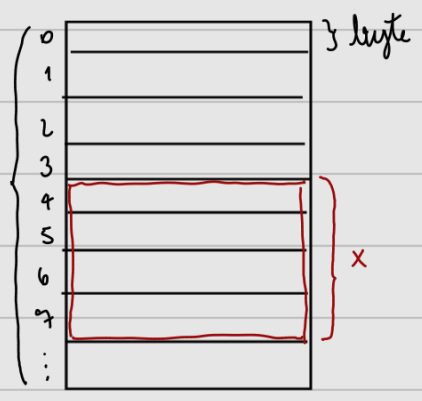
Ponteiro

- *
- endereço de memória

- Declaração

tipo *ptr;

↳ modifica o tamanho * da porção de memória a ser ocupada.



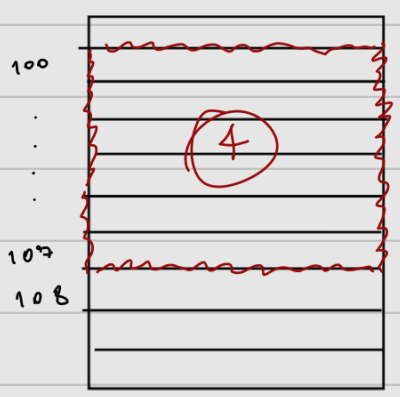
↳ endereço de memória

- char → 1 byte
- int → 4 bytes
- float
- double → 8 bytes
- short → 2 bytes
- bool
- long

int x;

- Endereço de x:
  - ↳ onde começa
  - = 4
  - = &x
    - ↓
    - mesmo endereçamento

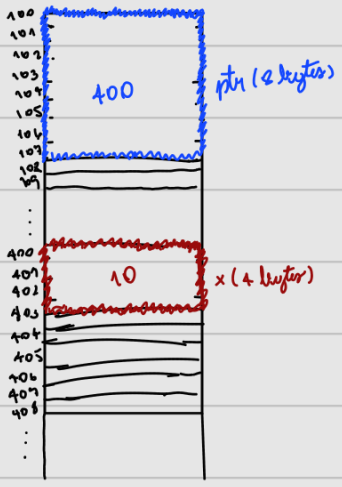
int *ptr;
ptr = &x;



Ponteiros

- Armazenam endereço de memória (número inteiro)

e o tamanho do dado para o que apontam (tipo do ponteiro)




```
int x = 10;
int *ptr;
ptr = &x;
```

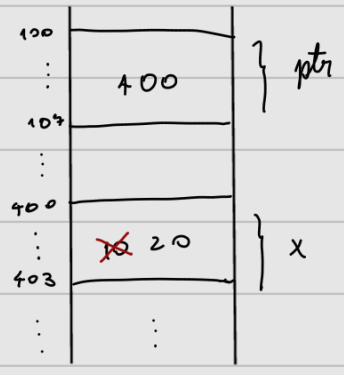
O operador de desreferenciamento: * (unário)

Ex.:

Saída:

```
printf("%d\n", ptr);     // 400   ← endereço de memória
printf("%d\n", *ptr);    // 10    ← conteúdo armazenado na variável que ocupa o endereço de memória para qual o ponteiro armazena o endereço.
```

Ex.: `*ptr = 20;`



E se meu ponteiro contém um endereço de memória que não pertence ao espaço de memória do processo associado ao programa?

Segmentation Fault, ou bico de memória.

* Valgrind

```
gcc -g -o exec codigo.c
Valgrind ./exec
```

* WSL (Windows)

↳ Auxilia com relatórios sobre alocação de memória.

Ex:

```
int *ptr = NULL; // Evitar ponteiros com lixo de memória
```

int *ptr = NULL; — Nesse fragmento deveria

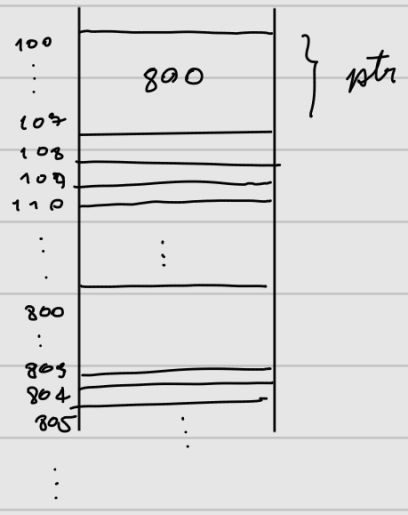
*ptr = 100; — lançar segfault.

Qual a forma correta de inicializar um ponteiro?

① Com o endereço de uma variável alocada automaticamente.

Ex.:

```
int x = 10;
int *ptr = &x;
```



② Com alocação dinâmica de memória

Ex.:

```
int *ptr = malloc (sizeof (int));
```

Alocação Dinâmica

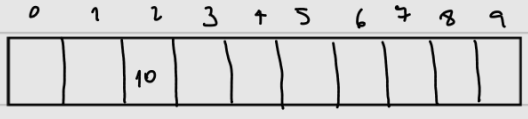
- void * malloc (size_t tam);
- void * calloc (int n, size_t tam); → int *v = malloc (10 * sizeof(int));

v[5] = 20; ou

*(v+5) = 20;

Vetores

int v[10]; ⟹



v[2] = 10;

(v)[y] = endereço inicial + y * tamanho tipo

v → ponteiro

} variável de ponteiros.

Tabelas de Dispersão (Hash)

Problema: Contar a quantidade de ocorrências de um número (que chamaremos de chave) lidos de entrada.

Dados: $0 \leq chave \leq n$.

Solução 1:

```
scanf("%d", &n);
int *cont = calloc(n+1, sizeof(int));
while (scanf("%d", &chave) != EOF) {
    cont[chave]++;
}
```

Para descobrir quantas vezes uma chave C ocorre, basta acessar cont[C]. Complexidade: O(1).

Problemas

① Subutilização de espaço cont será esparso.

Esparsidade: $\frac{\#\ elementos\ nulos}{Total\ de\ elementos}$.

Solução: Usar lista encadeada.

① Lê a chave

② Procuro na lista

2.1 - Se existir incremento a qtd de ocorrências

2.2 - Se não, adiciona à lista

Comp passo 2: O(N), onde N é a quantidade total de possíveis chaves.

— " —

② Problema 2: Se não couber na memória?

Considere um universo de chaves de tamanho T. Uma tabela de dispersão (hash) é um vetor de tamanho M, com $M \leq T$, tal que cada chave c ocupa a posição:

$$p = hash(c),$$

Onde hash é uma função, chamada função de espalhamento, que recebe um número como entrada e devolve outro.

A solução vista acima é uma tabela hash que utiliza uma função hash de endereçamento direto.

```
int hash (int chave) {
    return chave;
}

int main ( ) {
int m; scanf ("%d", &m);
int *t = calloc (m+1, sizeof (int));
while (scanf ("%d", &chave) != EOF)
    t [ hash (chave) ] ++;
```

E se quisermos uma tabela de dispersão de tamanho $M < T$ (de preferência bem menor).

Neste caso, duas chaves podem ocupar a mesma posição da tabela (já que $M < T$). Isso chama-se colisão. Portanto, para implementar uma tabela hash, precisamos

① Definir o tamanho da tabela.
② Escolher uma função hash
③ Escolher um método de tratamento de colisões.

Funções de espalhamento

Função clássica: função modular:

```
int hash (int chave) {
    return chave % M;
}
```

Boa prática: Escolher M primo (mersenne da forma $2^n - 1$)*

Métodos para resolver colisão

① Encadeamento separado

Tabela hash: vetor de nós cabeça de uma lista encadeada.

A função hash mapeia a chave para uma posição deste vetor.

Cada lista encadeada contém colisões.

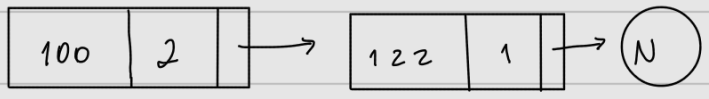
Implementação:

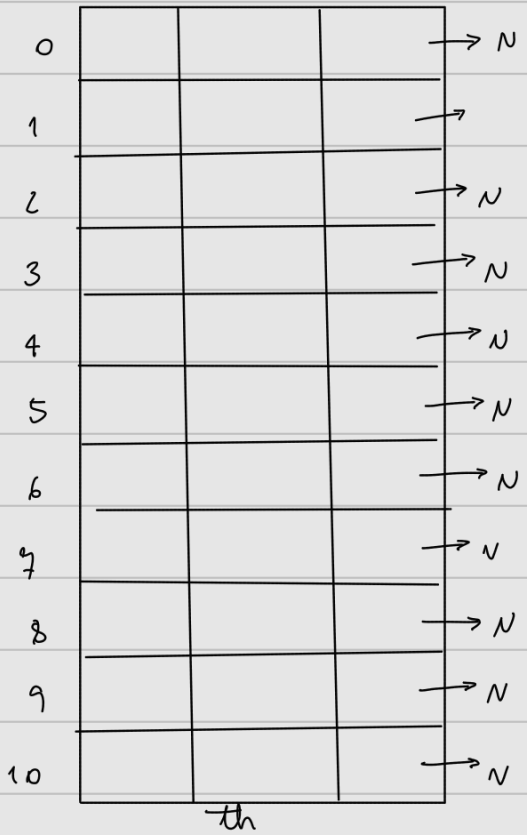
```
type def celula {
  int chave, ocor;
  struct celula *prox;
} celula;

celula th[M];
```

chave ⇒ th[hash(chave)]
chave % M

função modular



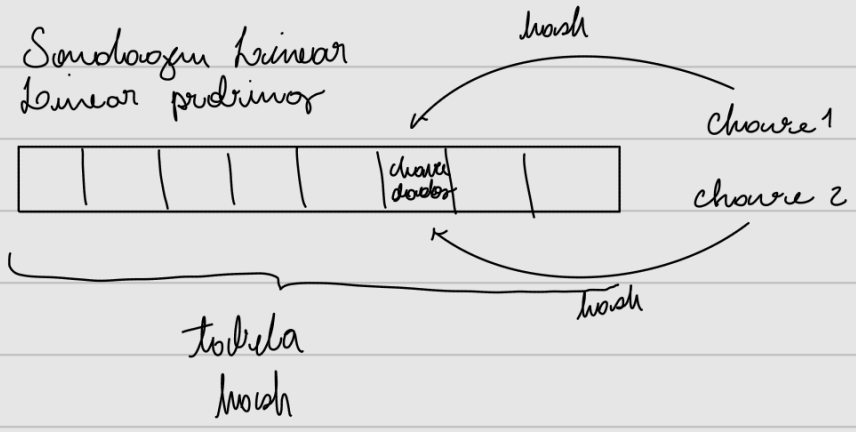


* Qual a complexidade de fazer uma busca em uma tabela hash?

$O\left(\frac{n}{m}\right)$ * quantidade de dados. * quantidade de listas encadeadas.

Métodos p/ resolução de colisões

② Endereçamento aberto



Obs.:

① Tabela cheia

↳ redimensionamento ou

↳ não insere

Para redimensionar

① Redimensionar a tabela hash

Recomenda-se dobrar o tamanho

② Atualizar a posição das chaves na hash.

② Definir "vazio".

EX.: Chaves ≥ 0

Vazio = -1

② Definir "vazio".

EX.: Chaves ≥ 0

Vazio = -1 M = 13

0	1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	28	17	41	19	20	21	22	23	24	25

Entrada: (I = Insere, B = Busca)

I 15

I 20

I 17 17 % 13 = 4

I 28 28 % 13 = 2

41 % 13 = 2

B 17

B 40 30 % 13 = 4

B 20 20 % 13 = 7

B 43 43 % 13 = 4

OBSERVAÇÃO:

- Uma boa tabela será esparsa.

- Uma boa tabela clusters pequenos.

Endereços e ponteiros

Os conceitos de *endereço* e *ponteiro* são fundamentais em qualquer linguagem de programação. Em C, esses conceitos são explícitos; em algumas outras linguagens eles são ocultos (e representados pelo conceito mais abstrato de *referência*). Dominar o conceito de ponteiro exige algum esforço e uma boa dose de prática.

Sumário:

Endereços

A memória RAM (= *random access memory*) de qualquer computador é uma sequência de bytes. A posição (0, 1, 2, 3, etc.) que um byte ocupa na sequência é o *endereço* (= *address*) do byte. (É como o endereço de uma casa em uma longa rua que tem casas de um lado só.) Se *e* é o endereço de um byte então *e* + 1 é o endereço do byte seguinte.

Cada variável de um programa ocupa um certo número de bytes consecutivos na memória do computador. Uma variável do tipo `char` ocupa 1 byte. Uma variável do tipo `int` ocupa 4 bytes e um `double` ocupa 8 bytes em muitos computadores. O número exato de bytes de uma variável é dado pelo operador **`sizeof`**. A expressão `sizeof (char)`, por exemplo, vale 1 em todos os computadores e a expressão `sizeof (int)` vale 4 em muitos computadores.

Cada variável (em particular, cada registro e cada vetor) na memória tem um *endereço*. Na maioria dos computadores, o endereço de uma variável é o endereço do seu primeiro byte. Por exemplo, depois das declarações

```
char c;
int i;
struct {
   int x, y;
} ponto;
int v[4];
```

as variáveis poderiam ter os seguintes endereços (o exemplo é fictício):

```
c       89421
i       89422
ponto   89426
v[0]    89434
v[1]    89438
v[2]    89442
```

O endereço de uma variável é dado pelo operador `&`. Assim, se `i` é uma variável então `&i` é o seu endereço. (Não confunda esse uso de "&" com o operador lógico *and*, representado por "&&" em C.) No exemplo acima, `&i` vale *89422* e `&v[3]` vale *89446*.

Outro exemplo: o segundo argumento da função de biblioteca `scanf` é o endereço da variável que deve receber o valor lido do teclado:

```
int i;
scanf ("%d", &i);
```

Exercícios 1

1. TAMANHOS. Compile e execute o seguinte programa:

```
int main (void) {
   typedef struct {
      int dia, mes, ano;
   } data;
   printf ("sizeof (data) = %d\n",
            sizeof (data));
   return EXIT_SUCCESS;
}
```

2. Compile e execute o seguinte programa. (O cast `(long int)` é necessário para que `&i` possa ser impresso no formato `%ld`. É mais comum imprimir endereços em notação hexadecimal, usando formato `%p`, que exige o cast `(void *)`.)

```
int main (void) {
   int i = 1234;
   printf (" i = %d\n", i);
   printf ("&i = %ld\n", (long int) &i);
   printf ("&i = %p\n", (void *) &i);
   return EXIT_SUCCESS;
}
```

Ponteiros

Um *ponteiro* (= apontador = *pointer*) é um tipo especial de variável que armazena um endereço. Um ponteiro pode ter o valor

`NULL`

que é um endereço "inválido". A macro `NULL` está definida na interface `stdlib.h` e seu valor é *0* (zero) na maioria dos computadores.

Se um ponteiro p armazena o endereço de uma variável i, podemos dizer "p aponta para i" ou "p é o endereço de i". (Em termos um pouco mais abstratos, diz-se que p é uma *referência* à variável i.) Se um ponteiro p tem valor diferente de NULL então

*p

é o *valor* da variável apontada por p. (Não confunda esse operador "*" com o operador de multiplicação!) Por exemplo, se i é uma variável e p vale &i então dizer "*p" é o mesmo que dizer "i".

A seguinte figura dá um exemplo. Do lado esquerdo, um ponteiro p, armazenado no endereço *60001*, contém o endereço de um inteiro. O lado direito dá uma representação esquemática da situação:

Tipos de ponteiros. Há vários tipos de ponteiros: ponteiros para bytes, ponteiros para inteiros, ponteiros para ponteiros para inteiros, ponteiros para registros, etc. O computador precisa saber de que tipo de ponteiro você está falando. Para declarar um ponteiro p para um inteiro, escreva

```
int *p;
```

(Há quem prefira escrever int* p.) Para declarar um ponteiro p para um registro reg, diga

```
struct reg *p;
```

Um ponteiro r para um ponteiro que apontará um inteiro (como no caso de uma matriz de inteiros) é declarado assim:

```
int **r;
```

Exemplos. Suponha que a, b e c são variáveis inteiras e veja um jeito bobo de fazer "c = a+b":

```
int *p; // p é um ponteiro para um inteiro
int *q;
p = &a; // o valor de p é o endereço de a
q = &b; // q aponta para b
c = *p + *q;
```

Outro exemplo bobo:

```
int *p;
int **r; // ponteiro para ponteiro para inteiro
p = &a;  // p aponta para a
r = &p;  // r aponta para p e *r aponta para a
c = **r + b;
```

Exercícios 2

1. Compile e execute o seguinte programa. (Veja um dos exercícios acima.)

```
int main (void) {
   int i; int *p;
   i = 1234; p = &i;
   printf ("*p = %d\n", *p);
   printf (" p = %ld\n", (long int) p);
   printf (" p = %p\n", (void *) p);
   printf ("&p = %p\n", (void *) &p);
   return EXIT_SUCCESS;
}
```

Aplicação

Suponha que precisamos de uma função que troque os valores de duas variáveis inteiras, digamos `i` e `j`. É claro que a função

```
void troca (int i, int j) {
   int temp;
   temp = i; i = j; j = temp;
}
```

não produz o efeito desejado, pois recebe apenas os *valores* das variáveis e não as variáveis propriamente ditas. A função recebe "cópias" das variáveis e troca os valores dessas cópias, enquanto as variáveis originais permanecem inalteradas. Para obter o efeito desejado, é preciso passar à função os *endereços* das variáveis:

```
void troca (int *p, int *q)
{
   int temp;
   temp = *p; *p = *q; *q = temp;
}
```

Para aplicar essa função às variáveis `i` e `j` basta dizer `troca (&i, &j)` ou então

```
int *p, *q;
p = &i; q = &j;
troca (p, q);
```

Exercícios 3

1. Verifique que a troca de valores de variáveis discutida acima poderia ser obtida por meio de uma macro do pré-processador:

   ```
   #define troca (X, Y) {int t = X; X = Y; Y = t;}
   . . .
   troca (i, j);
   ```

2. Por que o código abaixo está errado?

   ```
   void troca (int *i, int *j) {
      int *temp;
      *temp = *i; *i = *j; *j = *temp;
   }
   ```

3. Um ponteiro pode ser usado para dizer a uma função onde ela deve depositar o resultado de seus cálculos. Escreva uma função `hm` que converta minutos em horas-e-minutos. A função recebe um inteiro

`mnts` e os endereços de duas variáveis inteiras, digamos `h` e `m`, e atribui valores a essas variáveis de modo que `m` seja menor que 60 e que `60*h + m` seja igual a `mnts`. Escreva também uma função `main` que use a função `hm`.

4. Escreva uma função `mm` que receba um vetor inteiro `v[0..n-1]` e os endereços de duas variáveis inteiras, digamos `min` e `max`, e deposite nessas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função `main` que use a função `mm`.

Aritmética de endereços

Os elementos de qualquer vetor são armazenados em bytes consecutivos da memória do computador. Se cada elemento do vetor ocupa *s* bytes, a diferença entre os endereços de dois elementos consecutivos é *s*. Mas o compilador ajusta os detalhes internos de modo a criar a ilusão de que a diferença entre os endereços de dois elementos consecutivos é 1, qualquer que seja o valor de *s*. Por exemplo, depois da declaração

```
int *v;
v = malloc (100 * sizeof (int));
```

o endereço do primeiro elemento do vetor é `v`, o endereço do segundo elemento é `v+1`, o endereço do terceiro elemento é `v+2`, etc. Se `i` é uma variável do tipo `int` então

```
v + i
```

é o endereço do `(i+1)`-ésimo elemento do vetor. As expressões `v + i` e `&v[i]` têm exatamente o mesmo valor e portanto as atribuições

```
*(v+i) = 789;
v[i] = 789;
```

têm o mesmo efeito. Portanto, qualquer dos dois fragmentos de código abaixo pode ser usado para preencher o vetor `v`:

```
for (i = 0; i < 100; ++i)  scanf ("%d", &v[i]);
for (i = 0; i < 100; ++i)  scanf ("%d", v + i);
```

Todas essas considerações também valem se o vetor for alocado estaticamente (ou seja, antes que o programa comece a ser executado) por uma declaração como

```
int v[100];
```

mas nesse caso, `v` é uma espécie de "ponteiro constante", cujo valor não pode ser alterado.

Exercícios 4

1. Suponha que os elementos de um vetor `v` são do tipo `int` e cada `int` ocupa 4 bytes no seu computador. Se o endereço de `v[0]` é `55000`, qual o valor da expressão `v + 3`?
2. Suponha que `i` é uma variável inteira e `v` um vetor de inteiros. Descreva, em português, a sequência de operações que o computador executa para calcular o valor da expressão `&v[i + 9]`.
3. Suponha que `v` é um vetor. Descreva a diferença conceitual entre as expressões `v[3]` e `v + 3`.

4. O que faz a seguinte função?

```
void imprime (char *v, int n) {
   char *c;
   for (c = v; c < v + n; c++)
      printf ("%c", *c);
}
```

5. Analise o seguinte programa:

```
void func1 (int x) {
   x = 9 * x;
}

void func2 (int v[]) {
   v[0] = 9 * v[0];
}

int main (void) {
   int x, v[2];
   x    = 111;
   func1 (x); printf ("x: %d\n", x);
   v[0] = 111;
   func2 (v); printf ("v[0]: %d\n", v[0]);
   return EXIT_SUCCESS;
}
```

O programa produz a seguinte resposta, que achei surpreendente:

```
x: 111
v[0]: 999
```

Os valores de `x` e `v[0]` não deveriam ser iguais?

6. O seguinte fragmento de código pretende decidir se "abacate" vem antes ou depois de "uva" no dicionário. (Veja a seção *Cadeias constantes* do capítulo *Strings*.) O que está errado?

```
char *a, *b;
a = "abacate"; b = "uva";
if (a < b)
   printf ("%s vem antes de %s\n", a, b);
else
   printf ("%s vem depois de %s\n", a, b);
```

Perguntas e respostas

- PERGUNTA: É verdade que devemos atribuir um valor a um ponteiro tão logo o ponteiro é declarado?

  RESPOSTA: Há quem recomende inicializar todos os ponteiros imediatamente, ou seja, escrever `int *p = NULL;` em vez de simplesmente `int *p;`. Isso poderia ajudar a encontrar eventuais imperfeições no seu programa e poderia proteger contra a ação de hackers. Este sítio não segue essa recomendação para não cansar o leitor com detalhes repetitivos. (Além disso, parece deselegante atribuir um valor a uma variável sem que isso seja logicamente necessário...)

Veja *Pointers in C and C++* em GeeksforGeeks.

Veja o verbete *Pointer (computer programming)* na Wikipedia.

Veja o verbete *C syntax* na Wikipedia.

Veja o sítio-ferramenta `cdecl: C gibberish ↔ English`, que decodifica expressões complexas em C, especialmente aquelas envolvendo ponteiros.

Listas encadeadas

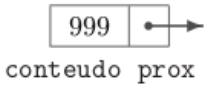
Uma lista encadeada é uma representação de uma sequência de objetos, todos do mesmo tipo, na memória RAM (= *random access memory*) do computador. Cada elemento da sequência é armazenado em uma célula da lista: o primeiro elemento na primeira célula, o segundo na segunda, e assim por diante.

Sumário:

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
```

(Os nomes da `struct` e do novo tipo-de-dados não precisam ser diferentes. Podemos perfeitamente escrever `typedef struct reg reg;`.)

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, ou "carga útil", 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 estão espalhadas pela memória de maneira imprevisível.)

Exercícios 1

1. ★ DECLARAÇÃO ALTERNATIVA. Verifique que a declaração de `celula` pode também ser escrita assim:

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

2. DECLARAÇÃO ALTERNATIVA. Verifique que a declaração de `celula` pode também ser escrita assim:

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

3. ★ DECLARAÇÃO ALTERNATIVA. Verifique que a declaração de `celula` pode também ser escrita assim:

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

4. 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 exatamente, 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 toda 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. De vetor para lista. 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. De lista para vetor. 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. Listas de strings. Este exercício 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 (veja a tabela ASCII). 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/

DCC-IME-USP

Comparação assintótica de funções

"We want a statement about software,
not hardware."

Ao ver uma expressão como $n+10$ ou n^2+1, a maioria das pessoas pensa automaticamente em valores pequenos de n. A análise de algoritmos faz exatamente o contrário: *ignora os valores pequenos* e concentra-se nos valores *enormes* de n. Para valores enormes de n, as funções

$$n^2,\ \ (3/2)n^2,\ \ 9999n^2,\ \ n^2/1000,\ \ n^2+100n,\ \ \text{etc.}$$

têm todas a mesma "taxa de crescimento" e portanto são todas "equivalentes". (A expressão "têm a mesma taxa de crescimento" não significa que têm a mesma derivada!)

A matemática que se interessa apenas pelos valores enormes de n é chamada *assintótica* (= *asymptotic*). Nessa matemática, as funções são classificadas em "ordens assintóticas". Antes de definir essas "ordens", entretanto, vamos examinar alguns exemplos.

Exemplos preliminares

Suponha que f e g são duas funções definidas no conjunto dos números naturais. Para comparar o comportamento assintótico de f e g, é preciso resolver o seguinte problema: encontrar um número c tal que $f(n) \le c\,g(n)$ para todo n suficientemente grande. Mais explicitamente: encontrar números c e N tais que $f(n) \le c\,g(n)$ para todo n maior que N. É claro que os números c e N devem ser constantes, ou seja, não devem depender de n. Veja alguns exemplos concretos:

Exemplo A. Encontre números c e N tais que $3n^2/2 + 7n/2 + 4 \le c\,n^2$ para todo n maior que N. Eis uma solução: $c = 6$ e $N = 3$. Esse par de números tem a propriedade desejada pois

$$\begin{aligned} 3n^2/2 + 7n/2 + 4 &\le 2n^2 + 4n + 4 \\ &\le 2n^2 + nn + nn \\ &= 6n^2 \end{aligned}$$

sempre que $n > 3$.

Exemplo B. Encontre números c e N tais que $2n^2 + 30n + 400 \le c\,n^2$ para todo n maior que N. Como mostraremos a seguir, os números $c = 432$ e $N = 0$ constituem uma solução. De fato, para todo $n > 0$ tem-se

$$\begin{aligned} 2n^2 + 30n + 400 &\le 2n^2 + 30n^2 + 400n^2 \\ &= 432\,n^2. \end{aligned}$$

Os números $c = 4$ e $N = 29$ constituem uma solução igualmente boa. Eis a prova: Se $n > 29$ então

$$\begin{aligned} 2n^2 + 30n + 400 &\le 2n^2 + nn + nn \\ &= 4n^2. \end{aligned}$$

EXEMPLO C. Queremos encontrar números c e N tais que $300 + 20/n \le c\,n^0$ para todo n maior que N. Eis uma solução do problema: $c = 301$ e $N = 19$. De fato, $300 + 20/n \le 300 + 1 = 301$ para todo $n \ge 20$.

EXEMPLO D. Existem números c e N tais que $20n^2 + 2n + 100 \le c(n^3 - 10)$ para todo n maior que N? A resposta é afirmativa; basta tomar $c = 2$ e $N = 21$. De fato,

$$\begin{aligned} 20n^2 + 2n + 100 &< 20n^2 + nn + n^2 \\ &= 22\,n^2 \\ &\le nn^2 \\ &= n^3 \\ &< n^3 + n^3 - 20 \\ &= 2\,(n^3 - 10) \end{aligned}$$

para todo $n \ge 22$.

EXEMPLO E. Existem números c e N tais que $n^3 \le c\,n^2$ para todo n maior que N? A resposta é negativa. Para provar isso, suponha por um momento que c e N têm a propriedade. Então

$$n \le c$$

para todo $n > N$. Mas isso é impossível, o que mostra que c e N não existem, CQD.

EXEMPLO F. Queremos encontrar números c e N tais que $n^2 - 100n \le c\,200n$ para todo n maior que N. Esse problema *não tem solução*, como mostraremos a seguir. Suponha por um momento que existe uma solução (c, N). Então $n^2 - 100n \le c\,200n$ para todo $n > N$. Logo, $n - 100 \le 200c$ e portanto

$$n \le 200\,c + 100$$

para todo $n > N$. Mas isso é absurdo. Esse absurdo mostra que o problema não tem solução, CQD.

EXEMPLO G. Seja a um número maior que 1. Queremos encontrar números c e N tais que $\log_a n \le c \lg n$ para todo n maior que N. Uma das soluções do problema consiste em $c = 1/\lg a$ e $N = 1$ pois

$$\log_a n = \frac{1}{\lg a} \lg n$$

para todo $n > 1$. (Veja o *Apêndice*.)

EXEMPLO H. Encontrar números c e N tais que $\lceil n/2 \rceil + 10 \le c\,n$ para todo n maior que N. Uma solução do problema tem $c = 1$ e $N = 21$. De fato,

$$\begin{aligned} \lceil n/2 \rceil + 10 &\le n/2 + 1 + 10 \\ &= n/2 + 11 \\ &\le n/2 + n/2 \\ &= n \end{aligned}$$

para todo $n \ge 22$. Outra solução consiste em $c = 10$ e $N = 1$, uma vez que $\lceil n/2 \rceil + 10 \le n/2 + 1 + 10 = n/2 + 11 \le 10n$ para todo $n > 1$.

Exercícios 1

1. Qual a diferença entre *assintótico* e *assintomático*?
2. O que há de errado com o seguinte raciocínio? "Existem números c e N tais que $n^3 \le c\,n^2$ para todo n maior que N. De fato, basta tomar $c = n$ e $N = 1$." (Compare com o exemplo E.)

3. ★ Queremos provar que $3n^2 + 7n \le cn^2$ para todo n suficientemente grande. Critique a seguinte prova: "Se $3n^2 + 7n \le cn^2$ então $3n + 7 \le cn$, supondo $n > 0$. Logo, $c \ge 3 + 7/n$. Logo, $c \ge 3 + 7$ é suficiente. Logo, $c \ge 10$ e $n > 0$. Fim da prova."

4. Encontre números c e N tais que $2n \lg n - 10n + 100 \lg n \le cn \lg n$ para todo n maior que N.

5. ★ Encontre números c e N tais que $2^{n+1} \le c2^n$ para todo n maior que N. Agora encontre números c e N tais que $2^n \le c2^{n/2}$ para todo n maior que N.

6. ★ Encontre números c e N tais que $n \le c2^n$ para todo n maior que N. (Veja o *Apêndice*.) Agora encontre números c e N tais que $\lg n \le cn$ para todo n maior que N.

7. É verdade que existem números c e N tais que $n \le c2^{n/4}$ para todo n maior que N? (Veja exercício anterior.)

8. Sejam f e g as funções definidas por $f(n) = n^2/10$ e $g(n) = n$. É verdade que existem números c e N tais que $f(n) \le cg(n)$ para todo n maior que N?

Ordem O

Para definir o conceito de ordem O, convém restringir a atenção a funções *assintoticamente não-negativas*, ou seja, funções f tais que $f(n) \ge 0$ para todo n suficientemente grande. Mais explicitamente: f é assintoticamente não-negativa se existe M tal que $f(n) \ge 0$ para todo n maior que M.

Agora podemos definir a ordem O. (Esta letra é um Ó maiúsculo. Ou melhor, um *ômicron* grego maiúsculo, de acordo com Knuth.)

> DEFINIÇÃO: Dadas funções assintoticamente não-negativas f e g, dizemos que f *está na ordem Ó de* g e escrevemos $f = O(g)$ se existem constantes positivas c e N tais que $f(n) \le cg(n)$ para todo n maior que N.

(No lugar de dizer "f está na ordem Ó de g", há quem diga "f é da ordem de no máximo g".)

A expressão "$f = O(g)$" é útil e poderosa porque *esconde informações irrelevantes*: a expressão informa o leitor sobre a relação entre f e g sem desviar sua atenção para os valores de c e N.

(A notação "$f = O(g)$" é conhecida como notação assintótica, ou notação de Landau. O sinal "=" nessa notação é um abuso pois não representa igualdade no sentido usual. Quem sabe seria melhor escrever "f está em $O(g)$" ou "f é $O(g)$".)

> EXEMPLO I. Suponha que $f(n) = 2n^2 + 30n + 400$ e $g(n) = n^2$. É óbvio que f e g são assintoticamente não-negativas. Além disso, como mostramos no exemplo B acima, $f(n) \le 4g(n)$ para todo $n > 29$. Portanto, $f = O(g)$.

> EXEMPLO J. É claro que as funções $300 + 20/n$ e $301n^0$ são assintoticamente não-negativas. Além disso, como mostramos no exemplo C acima, $300 + 20/n \le 301n^0$ para todo $n > 19$. Portanto, $300 + 20/n = O(n^0)$. (Poderíamos escrever $O(1)$ no lugar de $O(n^0)$, mas essa última expressão ajuda a lembrar que a variável é n.)

> EXEMPLO K. Considere as funções $20n^2 + 2n + 100$ e $n^3 - 10$. Observe que ambas são assintoticamente não-negativas pois são positivas quando $n \ge 3$. Além disso, $20n^2 + 2n + 100 \le 2(n^3 - 10)$ para todo $n > 21$, como mostramos no exemplo D acima. Portanto, $20n^2 + 2n + 100$ está em $O(n^3 - 10)$.

> EXEMPLO L. A função n^3 *não está* em $O(n^2)$ pois, como mostramos no exemplo E acima, não existem números c e N tais que $n^3 \le cn^2$ para todo n maior que N.

EXEMPLO M. *Não é verdade* que $n^2 - 100n = O(200n)$ porque não existem números c e N tais que $n^2 - 100n \le c\,200n$ para todo n maior que N, como mostramos no exemplo F acima.

EXEMPLO N. Para qualquer $a > 1$, as funções $\log_a n$ e $\lg n$ são assintoticamente não-negativas pois ambas são positivas quando $n \ge 2$. Além disso, $\log_a n = \lg n / \lg a$ para todo $n > 1$, como já observamos no exemplo G. Logo, $\log_a n = O(\lg n)$.

EXEMPLO O. É óbvio que as funções $\lceil n/2 \rceil + 10$ e n são assintoticamente não-negativas. Além disso, $\lceil n/2 \rceil + 10 \le 10n$ para todo $n > 1$, como mostramos no exemplo H. Logo, $\lceil n/2 \rceil + 10 = O(n)$.

Exercícios 2

1. Suponha que as funções f e g são assintoticamente não-negativas. Critique as seguintes tentativas de definição da classe O: "Dizemos que f está em $O(g)$ se …
   a. … $f(n) \le c\,g(n)$ para algum n suficientemente grande e todo c positivo."
   b. … para todo n suficientemente grande existe c positivo tal que $f(n) \le c\,g(n)$."
   c. … $f(n) \le c\,g(n)$ para todo n positivo e algum c positivo."
   d. … existem números positivos c, N e $n \ge N$ tais que $f(n) \le c\,g(n)$."
   e. … existe um número positivo N tal que $f(n) \le c\,g(n)$ para algum número positivo c e para todo $n \ge N$."
2. Faz sentido lógico dizer "$f(n)$ está em $O(n^2)$ para $n \ge 10$"?
3. ★ Fica bem dizer "$f(n) = O(n^2)$ com $c = 10$ e $N = 100$"? Como reformular isso?
4. ★ Mostre que a cláusula "n suficientemente grande" na definição da classe O é supérflua quando estamos lidando com funções estritamente positivas.
5. Critique o seguinte raciocínio: "A derivada de $4n^2+2n$ é $8n+2$. A derivada de n^2 é $2n$. Como $8n+2 > 2n$, podemos concluir que $4n^2+2n$ cresce mais que n^2 e portanto $4n^2+2n$ não é $O(n^2)$." Agora critique o seguinte raciocínio: "A derivada de $4n^2+2n$ é $8n+2$. A derivada de $9n^2$ é $18n$. Como $8n+2 \le 18n$ para $n \ge 1$, podemos concluir que $4n^2+2n$ é $O(9n^2)$."
6. É verdade que $100n = O(n)$? É verdade que $n^2/100 = O(n)$?
7. É verdade que $10n^2 + 200n + 500/n = O(n^2)$?
8. É verdade que $n^2 - 200n - 300 = O(n)$?
9. ★ *Coeficientes binomiais.* Seja $C(n, k)$ o número de combinações de n objetos tomados k a k. Mostre que $C(n,2) = O(n^2)$. Mostre que $C(n,3) = O(n^3)$. É verdade que, para qualquer número natural k tem-se $C(n,k) = O(n^k)$? [Solução]
10. ★ É verdade que 2^{n+1} está em $O(2^n)$? É verdade que 2^n está em $O(2^{n/2})$?
11. ★ É verdade que 3^n está em $O(2^n)$? É verdade que 2^n está em $O(3^n)$?
12. ★ É verdade que $\lg n = O(\log_3 n)$? É verdade que $\log_3 n = O(\lg n)$?
13. ★ É verdade que $\lceil \lg n \rceil = O(\lg n)$?
14. Prove que $n \lg n + 10n^{3/2}$ está em $O(n \lg n)$.
15. Mostre que $\lg (100n^3 + 200n + 300)^2 = O(\lg n)$.
16. É verdade que $2n^{1024} = O(2^n)$?
17. É verdade que $n^{2/3}$ está em $O(\sqrt{n})$? É verdade que $\sqrt{n}$ está em $O(n^{2/3})$?
18. Qual o maior número natural k tal que $n^{1/k} = O(\lg n)$?
19. *Transitividade.* Suponha que $f = O(g)$ e $g = O(h)$. Prove que $f = O(h)$.
20. Coloque as seguinte funções em ordem crescente de $O(\,)$: n, $\sqrt{n}$, $\log n$, 2^n, $n/(\log n)$, $2^{\sqrt{n}}$, $\log\log n$, $\log^2 n$, $\sqrt[3]{n}$.

Ordem Ômega

A expressão "$f = O(g)$" tem sabor de "$f \leq g$". Agora precisamos de um conceito que tenha o sabor de "$f \geq g$".

> DEFINIÇÃO: Dadas funções assintoticamente não-negativas f e g, dizemos que f *está na ordem Ômega de* g e escrevemos $f = \Omega(g)$ se existe um número positivo c tal que $f(n) \geq c\,g(n)$ para todo n suficientemente grande.

(No lugar de "f está na ordem Ômega de g", há quem diga "f é da ordem de pelo menos g".)

(O sinal "=" na notação de Landau "$f = \Omega(g)$" é um abuso pois não representa igualdade no sentido usual. Provavelmente é melhor dizer "f está em $\Omega(g)$" ou "f é $\Omega(g)$".)

Qual a relação entre O e Ω? Não é difícil verificar que $f = O(g)$ se e somente se $g = \Omega(f)$.

> EXEMPLO P. É fato que $n/2 - 100 = \Omega(n)$. Confira a prova: Para todo $n \geq 400$, tem-se
>
> $$n/2 - 100 \geq n/2 - n/4 = (1/4)n.$$
>
> Além disso, é evidente que as duas funções são assintoticamente não-negativas.

> EXEMPLO Q. A função $n^2/2 - 100n - 300$ está em $\Omega(n)$. De fato, para todo $n \geq 400$, tem-se
>
> $$\begin{aligned} n^2/2 - 100n - 300 &\geq n^2/2 - (n/4)n - n^2/8 \\ &= (1/2 - 1/4 - 1/8)\,n^2 \\ &= (1/8)n^2 \\ &\geq (400/8)n \\ &\geq 50n. \end{aligned}$$
>
> Além disso, $n^2/2 - 100n - 300 \geq 50n > 0$ para todo $n \geq 400$ e portanto as duas funções em discussão são assintoticamente não-negativas.

Exercícios 3

1. Sejam f e g duas funções assintoticamente não negativas. Prove que $f = O(g)$ se e somente se $g = \Omega(f)$.
2. *Coeficientes binomiais.* Seja $C(n, k)$ o número de combinações de n objetos tomados k a k. Mostre que $C(n,2) = \Omega(n^2)$. Mostre que $C(n,3) = \Omega(n^3)$. É verdade que, para qualquer número natural k tem-se $C(n,k) = \Omega(n^k)$? [Solução]
3. Prove que $2n^2 - 20n - 50 = \Omega(2n)$.
4. Prove que $2n^2 - 20n - 50 = \Omega(n^{3/2})$.
5. Prove que $n^2 - n - 10 = \Omega(n \lg n)$.
6. É verdade que $100n^2 + 10000 = \Omega(n^2 \lg n)$?
7. Prove que $100 \lg n - 10n + 2n \lg n$ está em $\Omega(n \lg n)$.
8. É verdade que 2^{n-2} está em $\Omega(2^n)$? É verdade que $3^{n/2}$ está em $\Omega(2^n)$?

Ordem Theta

Além dos conceitos que têm os sabores de "$f \le g$" e de "$f \ge g$", precisamos de um que tenha o sabor de "$f = g$".

> DEFINIÇÃO: Dizemos que duas funções assintoticamente não negativas f e g *são da mesma ordem* e escrevemos $f = \Theta(g)$ se $f = O(g)$ e $f = \Omega(g)$. Trocando em miúdos, $f = \Theta(g)$ significa que existe constantes positivas c e d tais que $c\,g(n) \le f(n) \le d\,g(n)$ para todo n suficientemente grande.

(No lugar de dizer "f está na ordem Θ de g", há quem diga, simplesmente, que "f é da ordem de g".) Seguem alguns exemplos.

> EXEMPLO R. As funções n^2, $(3/2)n^2$, $9999\,n^2$, $n^2/1000$ e n^2+100n pertencem todas à ordem $\Theta(n^2)$.

> EXEMPLO S. Para quaisquer números a e b maiores que 1, a função $\log_a n$ está em $\Theta(\log_b n)$.

Exercícios 4

1. Mostre que $f = \Theta(g)$ se e somente se $g = \Theta(f)$.
2. Quais das afirmações abaixo estão corretas?
   $(3/2)n^2 + (7/2)n - 4 = \Theta(n^2)$
   $9999\,n^2 = \Theta(n^2)$
   $n^2/1000 - 999\,n = \Theta(n^2)$
   $\lg n + 1 = \Theta(\log_{10} n)$
   $\lfloor \lg n \rfloor = \Theta(\lg n)$

Onde as funções estão definidas?

A discussão acima supõe, implicitamente, que todas as funções estão definidas no conjunto dos números naturais. Mas tudo continua funcionando se as funções estiverem definidas em algum outro domínio. Veja alguns exemplos de domínios:

- números naturais maiores que 99,
- potências inteiras de 2 (ou seja, 2^0, 2^1, 2^2, etc.),
- potências inteiras de 1½,
- números racionais maiores que ½.

A notação O pode ser usada em qualquer desses domínios. Por exemplo, se f é uma função assintoticamente não-negativa definida nas potências inteiras de 2, dizer que $f(n) = O(n^2)$ é o mesmo que dizer que existe um número positivo c tal que $f(2^k) \le c\,2^{2k}$ para todo natural k suficientemente grande.

As mesmas observações se aplicam à notação Ω e à notação Θ.

Análise de algoritmos

A análise de algoritmos procura estimar o consumo de tempo de um algoritmo em função do tamanho, digamos n, de sua "entrada".

Suponha que $f(n)$ é o consumo de tempo de um algoritmo A para um certo problema e $g(n)$ é o consumo de tempo de um algoritmo B para o mesmo problema. Para comparar as eficiências de A e B, a análise de algoritmos estuda o comportamento de f e g para valores grandes de n. Se $f = O(g)$, o algoritmo A é

considerado pelo menos tão eficiente quanto B. Se, além disso, f não está em $\Theta(g)$, o algoritmo A é considerado mais eficiente que B.

As funções mais comuns na análise de algoritmos são $\lg n$, n, $n \lg n$, n^2 e n^3. A seguinte tabela mostra os valores dessas funções para n entre 10^3 e 10^7. Para simplificar, a tabela mostra o logaritmo na base 10. Cada função da tabela é "dominada" pela seguinte e esse efeito cresce com o valor de n.

$\log n$	3	4	5	6	7
n	1000	10000	100000	1000000	10000000
$n \log n$	3000	40000	500000	6000000	70000000
n^2	1000000	100000000	10000000000	1000000000000	100000000000000
n^3	1000000000	1000000000000	1000000000000000	1000000000000000000	100000000000000000000

Veja meu *Minicurso de Análise de Algoritmos*: seção 1.3 e apêndice A.

Verbete *Big O notation* na Wikipedia.

---

www.ime.usp.br/~pf/analise_de_algoritmos/
Atualizado em 2021-03-21
© *Paulo Feofiloff*
Departamento de Ciência da Computação
Instituto de Matemática e Estatística da USP

Hashing

hash = picadinho
to hash = picar, fazer picadinho, misturar

"Hashing is used extensively in applications
and deserves recognition
as one of the cleverer inventions
of computer science."
— E.S. Roberts, *Programming Abstractions in C*

Este capítulo usa um pequeno problema de contagem como pretexto para introduzir a estrutura de dados conhecida como *tabela de dispersão* ou *hash table*. Essa estrutura é responsável por acelerar muitos algoritmos que envolvem consultas bem como inserções e remoções em uma tabela de dados.

Sumário:

Um problema de contagem

Suponha dado um fluxo de números inteiros positivos na entrada padrão `stdin`. Os números serão chamados *chaves*. As chaves estão em ordem arbitrária e há muitas chaves repetidas. Considere agora o problema de

> calcular o número de ocorrências de cada chave.

Por exemplo, o número de ocorrências de cada chave no fluxo `4998 9886 1933 1435 9886 1435 9886 7233 4998 7233 1435 1435 1004` é dado pela seguinte tabela (na primeira linha temos as diferentes chaves e na segunda o número de ocorrências de cada chave):

```
1004 1435 1933 4998 7233 9886
   1    4    3    2    2    3
```

Nosso problema de contagem tem um requisito adicional importante: a contagem deve ser feita de maneira incremental, ou seja, *online*. Cada chave recebida do fluxo de entrada deve ser imediatamente contabilizada, de modo que tenhamos, a cada passo, as contagens referentes à *parte do fluxo vista até o momento*.

O desempenho de qualquer algoritmo para o problema será medido pelo tempo consumido para contabilizar *uma* chave. Idealmente, gostaríamos que esse tempo fosse constante, ou seja, não dependesse do número de chaves já lidas (nem do número de chaves *distintas* já lidas). Mas seremos obrigados a nos contentar com algo menos que o ideal.

Quatro soluções. Discutiremos quatro diferentes algoritmos para o problema. Os dois primeiros são muito simples. Os dois seguintes, bem mais eficientes na prática, usam a técnica de hashing. Embora simples, os dois primeiros algoritmos constituem uma importante introdução aos dois outros.

Denotaremos por N o *comprimento* do fluxo de entrada, ou seja, o número total de chaves no fluxo. O valor de N pode ser muito grande (milhões ou até bilhões), mas o número de chaves distintas é tipicamente bem menor.

Suporemos que todas as chaves são menores que um número `R`. No exemplo acima, podemos supor que `R` vale `10000`. O conjunto `0..R-1` será chamado *universo das chaves*. Em geral, a maioria das chaves do universo não aparece no fluxo de entrada.

Exercícios 1

1. Critique a seguinte proposta de algoritmo para o problema de contagem: armazene o fluxo de chaves num vetor, ordene o vetor, e depois percorra o vetor ordenado contando o número de ocorrências de cada chave.

Algoritmo 1: endereçamento direto

Começamos com um algoritmo conhecido como *endereçamento direto*. Embora muito simples, esse algoritmo contém a semente da técnica de hashing.

Suponhamos que as chaves são do tipo `int` e que `R` chaves cabem confortavelmente na memória RAM do computador. Podemos então usar uma tabela `tb[0..R-1]` para registrar os números de ocorrências das chaves.

```
int *tb;
tb = malloc (R * sizeof (int));
```

O algoritmo de endereçamento direto inicializa o vetor `tb` com zeros e repete a seguinte rotina: lê uma chave `ch` do fluxo de entrada e contabiliza a chave executando a seguinte função:

```
void contabiliza (int ch) {
   tb[ch]++;
}
```

Depois de cada execução dessa função, para cada `c` em `0..R-1`, o valor de `tb[c]` será o número de ocorrências de `c` na parte do fluxo lida até o momento.

Desempenho. Cada invocação de `contabiliza` consome tempo constante, ou seja, independente do tamanho `R` do universo e do número de chaves já lidas.

Esse algoritmo é muito rápido, mas só é prático se `R` for pequeno e conhecido explicitamente de antemão. Mesmo nesse caso, o algoritmo pode desperdiçar muito espaço. Por exemplo, se `R` vale 10 mil e o fluxo contém apenas mil chaves distintas, 90% do vetor `tb` ficará ocioso.

Exercícios 2

1. TESTES DE DESEMPENHO. Escreva e teste um programa que use endereçamento direto para resolver o problema da contagem e imprima um relatório com as seguintes informações: o comprimento *N* do fluxo de entrada, o número de chaves distintas, a chave mais frequente, e o número de ocorrências dessa chave. Use como fluxo de entrada os arquivos `randInt1K.txt`, `randInt10K.txt`, `randInt100K.txt` e `randInt1M.txt`, que contêm mil, 10 mil, 100 mil e 1 milhão de chaves aleatórias, todas entre `0` e `9999`. Cronometre o seu programa usando a função `clock` da biblioteca `time`. O resultado está de acordo com as previsões teóricas?

Algoritmo 2: lista encadeada

Nosso segundo algoritmo armazena a contagem das chaves numa lista encadeada. As células da lista podem ter a seguinte estrutura:

```
typedef struc reg celula;
struct reg {
   int     chave, ocorr;
   celula *prox;
};
```

Se `p` é o endereço de uma célula então `p->ocorr` será o número de ocorrências da chave `p->chave`. Se `p` e `q` são endereços de duas células diferentes então `p->chave` será diferente de `q->chave`. A lista encadeada será apontada pela variável global `tb`:

```
celula *tb;
```

O algoritmo de contagem inicializa `tb` com `NULL` e repete a seguinte rotina: lê uma chave `ch` do fluxo de entrada e invoca a seguinte função para contabilizar a chave:

```
void contabiliza (int ch) {
   celula *p;
   p = tb;
   while (p != NULL && p->chave != ch)
      p = p->prox;
   if (p != NULL)
      p->ocorr += 1;
   else {
      p = malloc (sizeof (celula));
      p->chave = ch;
      p->ocorr = 1;
      p->prox = tb;
      tb = p;
   }
}
```

Desempenho. No pior caso, cada execução de `contabiliza` consome tempo proporcional ao número de chaves distintas já lidas. Portanto, a execução de `contabiliza` pode ficar cada vez mais lenta à medida que o fluxo de entrada é lido. Se todas as chaves forem distintas, as últimas execuções de `contabiliza` podem chegar a consumir tempo proporcional a *N*.

Mesmo no caso médio, típico de aplicações práticas, a função `contabiliza` pode consumir tempo proporcional à metade do número de chaves distintas já lidas, o que não é satisfatório.

Por outro lado, essa solução do problema de contagem não desperdiça espaço, pois o número de células é igual ao número de chaves distintas no fluxo de entrada.

Exercícios 3

1. LISTA EM ORDEM CRESCENTE. Reescreva a função `contabiliza` acima de modo que as chaves sejam armazenadas na lista encadeada em ordem crescente. Estime o desempenho. Vale a pena manter a lista em ordem crescente?
2. TESTES DE DESEMPENHO. Repita os testes sugeridos num dos exercícios acima, desta vez usando uma lista encadeada para armazenar as contagens.
3. VETOR DE CÉLULAS. Refaça a discussão da seção anterior usando um *vetor* de células no lugar de uma lista encadeada. Redimensione o vetor à medida que o número de chaves distintas aumenta. Qual o consumo de tempo dessa versão? Agora mantenha o vetor em ordem crescente de chaves e refaça a análise.

Tabelas de dispersão e funções de espalhamento

O próximos algoritmos vão combinar as boas caracteristicas dos dois algoritmos anteriores. Antes porém, precisamos introduzir o conceito de *hashing*. Começamos por estabelecer a terminologia e descrever as ideias de maneira abstrata; implementações concretas serão dadas nas seções subsequentes.

Vamos supor que a contagem das chaves é registrada num vetor `tb[0..M-1]`. O valor de `M` e a natureza exata dos elementos do vetor ficarão indefinidos por enquanto. Mas você deve imaginar que, de alguma forma, cada elemento de `tb` registra o número de ocorrências de alguma chave. O vetor `tb` será chamado *tabela de dispersão* (= *hash table*). O tamanho `M` da tabela é usualmente muito menor que o tamanho `R` do universo de chaves. Assim, um elemento típico do vetor `tb` deverá cuidar de duas ou mais chaves.

Para encontrar a posição no vetor `tb` que corresponde a uma chave `ch`, é preciso converter `ch` em um índice entre `0` e `M-1`. Qualquer função que faça a conversão, levando o universo `0..R-1` das chaves no conjunto `0..M-1` de índices, é chamada *função de espalhamento* (= *hash function*). Neste capítulo, indicaremos por

```
hash(ch, M)
```

a invocação de uma função de espalhamento para uma chave `ch`. O número `hash(ch,M)` será chamado *código hash* (= *hash code*) da chave `ch`. Uma função de espalhamento muito popular leva cada chave `ch` em `ch%M`, ou seja, no resto da divisão de `ch` por `M`. Se `M` vale `100`, por exemplo, então `ch%M` consiste nos dois últimos dígitos decimais de `ch`.

Se a função de espalhamento levar duas chaves no mesmo índice, teremos uma *colisão*. Se `M` é menor que `R` — e mais ainda se `M` é menor que o número de chaves distintas — as colisões são inevitáveis. Se `M` vale `100`, por exemplo, a função resto-da-divisão-por-`M` faz colidir todas as chaves que têm os mesmos dois últimos dígitos decimais.

Uma boa função de espalhamento deve espalhar bem as chaves pelo conjunto `0..M-1` de índices. Uma função que leva toda chave num índice par, por exemplo, não é boa. Uma função que só depende de alguns poucos dígitos da chave também não é boa. Infelizmente, não existe uma função de espalhamento que seja boa para todos os conjuntos de chaves extraídos do universo `0..R-1`. Para começar a enfrentar essa dificuldade,

`M` deve ser um número primo,

pois isso tende a reduzir o número de colisões. Veja, por exemplo, o conjunto de chaves na primeira coluna da seguinte tabela (copiada do livro de Sedgewick e Wayne) e considere o resto da divisão de cada chave por `100` (um não-primo) e o resto da divisão por `97` (um primo). Observe que o número de colisões é maior no primeiro caso:

ch	ch%100	ch%97
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Em geral, encontrar uma boa função de espalhamento é mais uma arte que uma ciência.

Agora que cuidamos de espalhar as chaves pelo intervalo `0..M-1`, precisamos inventar um meio de *resolver as colisões*, ou seja, de armazenar todas as chaves que a função de espalhamento leva numa mesma posição da tabela de dispersão. As seções seguintes descrevem duas maneiras de fazer isso.

Exercícios 4

1. POR QUE MÓDULO PRIMO? Suponha que `ch` e `M` são divisíveis por um inteiro `k`. Mostre que `ch%M` também será divisível por `k`. (Este exercício dá uma pequena indicação das vantagens de usar um número primo

como valor de `M`.)

2. Seja `d` o número ⌈`R/M`⌉, isto é, teto de `R/M`. Considere a função de espalhamento que associa a cada chave `ch` o piso de `ch/d` (ou seja, o resultado da divisão inteira de `ch` por `d`). Por exemplo, se `R` é 10^5 e `M` é 10^2 então `d` vale 10^3 e portanto `ch/d` é dado pelos dois primeiros dígitos decimais de `ch`. Discuta a qualidade dessa função de espalhamento.

3. Paradoxo do aniversário. Aplique a função de espalhamento resto-da-divisão-por-M a uma sequência de chaves aleatórias. Depois de quantas chaves acontece a primeira colisão? Faça experimentos, com diversos valores de M, para encontrar o momento da primeira colisão. (De acordo com a teoria das probabilidades clássica, a primeira colisão acontece depois de aproximadamente $\sqrt{\pi M/2}$ chaves, sendo π o número 3.14159...)

Algoritmo 3: hashing com encadeamento

A técnica de hashing tem dois ingredientes: uma função de espalhamento e um mecanismo de resolução de colisões. A seção anterior discutiu o primeiro ingrediente; esta seção cuida do segundo.

As colisões na tabela de dispersão podem ser resolvidas por meio de listas encadeadas: todas as chaves que têm um mesmo código hash são armazenadas numa lista encadeada. As células das listas encadeadas são iguais às do algoritmo 2:

```
typedef struc reg celula;
struct reg {
   int     chave, ocorr;
   celula *prox;
};
```

Podemos supor então que os elementos da tabela de dispersão `tb[0..M-1]` são ponteiros para listas encadeadas:

```
celula **tb;
tb = malloc (M * sizeof (celula *));
```

Para cada índice `h`, a lista encadeada `tb[h]` conterá todas as chaves que têm código hash `h`.

O algoritmo de contagem resultante é conhecido como *hashing com encadeamento* e pode ser visto como uma combinação dos algoritmos 1 e 2 discutidos acima. O algoritmo inicializa todos os elementos do vetor `tb` com `NULL` e repete a seguinte rotina: lê uma chave `ch` do fluxo de entrada e executa a seguinte função:

```
void contabiliza (int ch) {
   int h = hash (ch, M);
   celula *p = tb[h];
   while (p != NULL && p->chave != ch)
      p = p->prox;
   if (p != NULL)
      p->ocorr += 1;
   else {
      p = malloc (sizeof (celula));
      p->chave = ch;
      p->ocorr = 1;
      p->prox = tb[h];
      tb[h] = p;
```

```
   }
}
```

Desempenho. No pior caso, a função de espalhamento `hash` leva todas as chaves na mesma posição da tabela de dispersão e portanto todas as chaves ficam na mesma lista encadeada. Nesse caso, o desempenho não é melhor que o do algoritmo 2: cada execução de `contabiliza` consome tempo proporcional ao número de chaves já lidas do fluxo de entrada.

No caso médio, típico de aplicações práticas, o desempenho de `contabiliza` é muito melhor. Se a função `hash` espalhar bem as chaves pelo conjunto `0..M-1`, todas as listas encadeadas terão aproximadamente o mesmo comprimento e então podemos esperar que o consumo de tempo de `contabiliza` seja limitado por algo proporcional a

`n / M`,

onde `n` é o número de chaves lidas até o momento. Se `M` for `997`, por exemplo, podemos esperar que a função seja cerca de 1000 vezes mais rápida que aquela do algoritmo 2. É claro que devemos procurar escolher um valor de `M` que seja grande o suficiente para que as `M` listas sejam curtas (digamos algumas dezenas de elementos) mas não tão grande que muitas das listas fiquem vazias.

Exercícios 5

1. Resolva o problem da contagem para o fluxo de chaves `17 21 19 4 26 30 37` usando hashing com encadeamento. A tabela de dispersão deve ter tamanho `13` e a função de espalhamento deve ser o resto da divisão da chave por `13`. Faça uma figura do estado final da tabela de dispersão.
2. Suponha que o comprimento *N* do fluxo de entrada é aproximadamente 50000. Escolha um bom valor para o tamanho `M` da tabela de dispersão.
3. TESTES DE DESEMPENHO. Repita os testes sugeridos num dos exercícios acima, desta vez usando uma tabela de dispersão com colisões resolvidas por encadeamento. Experimente diferentes valores (primos e não-primos) para o tamanho `M` da tabela de dispersão. Determine a média e o desvio padrão dos comprimentos das listas encadeadas.

Algoritmo 4: hashing com sondagem linear

Esta seção descreve uma segunda maneira de resolver as colisões na tabela de dispersão: todas as chaves que colidem são armazenadas *na própria tabela*.

Os elementos da tabela de dispersão `tb[0..M-1]` são células que têm apenas os campos `chave` e `ocorr`:

```
typedef struc reg celula;
struct reg {
   int chave, ocorr;
};

celula *tb;
tb = malloc (M * sizeof (celula));
```

Durante a contagem, algumas das células da tabela `tb` estarão *vagas* enquanto outras estarão *ocupadas*. As células vagas terão `chave` igual a `-1`. Nas células ocupadas, a `chave` estará em `0..R-1` e `ocorr` será o correspondente número de ocorrências. Se uma célula `tb[h]` está vaga, podemos garantir que nenhuma chave na parte já lida do fluxo de entrada tem código hash igual a `h`. Mas se `tb[h]` está ocupada, não podemos concluir que o código hash de `tb[h].chave` é igual a `h`.

Cada chave `ch` do fluxo de entrada será contabilizada da seguinte maneira. Seja `h` o código hash de `ch`. Se a célula `tb[h]` estiver vaga ou tiver chave igual a `ch`, o conteúdo da célula é atualizado. Senão, o algoritmo *procura a próxima célula* de `tb` que esteja vaga ou tenha `chave` igual a `ch`.

A implementação dessas ideias leva ao algoritmo de *hashing com sondagem linear*. O algoritmo começa por inicializar todas as células da tabela `tb` fazendo `chave = -1` e `ocorr = 0`. Depois, repete a seguinte rotina: lê uma chave `ch` e invoca a seguinte função:

```
void contabiliza (int ch) {
   int c, sonda, h;
   h = hash (ch, M);
   for (sonda = 0; sonda < M; sonda++) {
      c = tb[h].chave;
      if (c == -1 || c == ch) break; // *
      h = (h + 1) % M;
   }
   if (sonda >= M)
      exit (EXIT_FAILURE);
   if (c == -1)
      tb[h].chave = ch;
   tb[h].ocorr++;
}
```

A função faz `M` tentativas — conhecidas como *sondagens* — de encontrar uma célula "boa" (na linha * do código). A busca fracassa somente se a tabela `tb` estiver cheia. Nesse caso, a execução da função é abortada. (Teria sido melhor redimensionar a tabela `tb` e continuar trabalhando.)

Suponha, por exemplo, que o tamanho `M` da tabela de dispersão é `10` (adotamos um número não-primo para tornar o exemplo mais simples). Suponha também que `hash (ch, M)` é definido como `ch % M`. Se o fluxo de entrada é

```
333 336 1333 333 7777 446 556 999
```

então o estado final da tabela de dispersão `tb[0..9]` será o seguinte:

```
chave   ocorr
  999       1
   -1       0
   -1       0
  333       2
 1333       1
   -1       0
  336       1
 7777       1
  446       1
  556       1
```

Desempenho. No pior caso, a função de espalhamento `hash` leva todas as chaves na mesma posição da tabela de dispersão e portanto as chaves ocupam células consecutivas da tabela. Nesse caso, o desempenho

não é melhor que o do algoritmo 2: cada execução de `contabiliza` consome tempo proporcional ao número de chaves já lidas do fluxo de entrada.

No caso médio, típico de aplicações práticas, o desempenho de `contabiliza` é muito melhor. ⚠ Se mais da metade das células da tabela de dispersão estiver vaga (como acontece se usarmos redimensionamento) e a função `hash` espalhar bem as chaves pelo conjunto `0..M-1`, uma execução da função `contabiliza` não precisará fazer mais que algumas poucas sondagens para encontrar uma célula "boa". Assim, o consumo de tempo de uma execução de `contabiliza` será praticamente independente do número de chaves já lidas.

Exercícios 6

1. Resolva o problem da contagem para o fluxo de chaves `17 21 19 4 26 30 37` usando hashing com sondagem linear. A tabela de dispersão deve ter tamanho `13` e a função de espalhamento deve ser o resto da divisão da chave por `13`. Faça uma figura do estado final da tabela de dispersão.
2. ★ Prova de correção. Considere a função `contabiliza` do algoritmo de hashing com sondagem linear dada acima. Suponha que temos `c == -1` numa certa iteração. Prove que não existe célula `tb[h]` tal que `tb[h].chave == ch`.
3. ★ Redimensionamento da tabela de dispersão. A execução da função `contabiliza` dada acima é abortada se a tabela de dispersão estiver cheia. Escreva uma versão melhor, que redimensione a tabela escolhendo um novo valor de `M` que seja aproximadamente o dobro do anterior, alocando uma nova tabela `tb`, e reinserindo todas as chaves na nova tabela.
4. Testes de desempenho. Repita os testes sugeridos num dos exercícios acima, desta vez usando uma tabela de dispersão com colisões resolvidas por sondagem linear. Experimente diferentes valores (primos e não-primos) para o tamanho `M` da tabela. Calcule também o número máximo de sondagens que `contabiliza` faz para encontrar a posição desejada na tabela de dispersão.

Exercícios 7

1. Filtro de repetidos. Escreva um programa que leia um arquivo de texto que contém (as representações decimais de) números inteiros positivos, elimine os números repetidos, e grave o arquivo resultante. Não altere a ordem relativa dos números. O seu programa deve ter caráter de *filtro*, ou seja, deve gravar o resultado à medida que o arquivo de entrada for sendo lido.
2. Encontre o primeiro número não-repetido em um longo arquivo de texto que contém números inteiros.

Hashing de strings

Em muitas aplicações, as chaves são strings (especialmente strings ASCII) e não números. Como construir uma tabela de dispersão nesse caso? Poderíamos, por exemplo, adotar uma tabela de dispersão de tamanho `256` e usar a função de espalhamento que leva cada string no valor numérico do seu primeiro byte. (Todas as strings que começam com `'a'`, por exemplo, seriam levadas para a posição `97` da tabela.) Mas essa ideia não espalharia bem o conjunto de chaves pelo intervalo `0..255`.

Uma maneira geral de lidar com chaves que são strings envolve dois passos: o primeiro converte a string em um número inteiro; o segundo submete esse número a uma função de espalhamento. Uma conversão óbvia consiste em tratar cada string como um número inteiro na base 256. A string `"abcd"`, por exemplo, é convertida no número $97\times256^3 + 98\times256^2 + 99\times256^1 + 100\times256^0$, igual a 1633837924. Esse tipo de conversão pode facilmente produzir números maiores que `INT_MAX`, e assim levar a um overflow aritmético. Se os cálculos forem feitos com variáveis `int`, o resultado poderá ser estritamente negativo, o que seria desastroso. Por isso, faremos os cálculos com variáveis `unsigned`, garantindo assim que o resultado fique entre `0` e `UINT_MAX` mesmo que haja overflow:

```
typedef char *string;

unsigned convert (string s) {
   unsigned h = 0;
   for (int i = 0; s[i] != '\0'; i++)
      h = h * 256 + s[i];
   return h;
}
```

Para submeter o resultado da conversão à função de espalhamento basta fazer `hash(convert(s),M)`. Se adotarmos a função resto-da-divisão-por-`M`, basta fazer `convert(s)%M`.

Uma boa alternativa é fazer a conversão intercalada com o cálculo da função de espalhamento. O resultado pode não ser idêntico a `convert(s)%M`, mas cumpre o papel de espalhar as chaves pelo conjunto `0..M-1`:

```
int string_hash (string s, int M) {
   unsigned h = 0;
   for (int i = 0; s[i] != '\0'; i++)
      h = (h * 256 + s[i]) % M;
   return h;
}
```

Se a string é `"abcd"` e `M` é `101`, por exemplo, o índice calculado por `string_hash` é `11`:

```
( 0 * 256 +  97) % 101 = 97
(97 * 256 +  98) % 101 = 84
(84 * 256 +  99) % 101 = 90
(90 * 256 + 100) % 101 = 11
```

O uso da base `256` não é obrigatório; podemos usar uma outra base qualquer. Por razões não muito óbvias, convém que a base seja um número primo, como `31` por exemplo.

Exercícios 8

1. Suponha que as chaves são strings e `M` vale `256`. Considere a função de espalhamento que associa a cada chave o seu primeiro byte. Discuta a qualidade dessa função de espalhamento.
2. Mostre que se trocarmos "`256`" por "`1`" na função `convert`, todas as permutações da string `s` colidirão.
3. Na função `convert`!, por que não trocar as duas linhas do meio pelas seguintes?

```
unsigned h = s[0];
for (i = 1; s[i] != '\0'; i++)
```