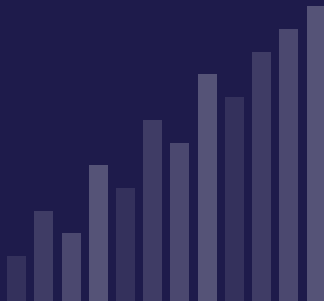

AULA 16 · BLOCO 2: DICIONÁRIOS E ESTRUTURAS LINEARES

Sentinelas e listas sem ponteiros

Seções 10.2 e 10.3

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Usar uma sentinela para eliminar os casos especiais das listas
- ▶ Implementar a lista circular duplamente ligada com sentinela do livro
- ▶ Representar listas com vetores e gerenciar posições livres
- ▶ Medir quantos testes a sentinela economiza numa busca



Teoria

A sentinela $L.nil$, listas em vetores e a lista livre



Prática em C

Lista com sentinela e lista em vetores com
ALLOCATE-Object



Experimento

Busca com e sem sentinela: $2n + 1$ contra $n + 1$
testes



Conexões

Kernel Linux, alocadores, sistemas de arquivos e
bancos de dados

A sentinela

- ▶ Um nó falso $L.nil$ fica entre a cabeça e a cauda; $L.nil.prox$ é a cabeça e $L.nil.ant$, a cauda.
- ▶ A lista nunca está "realmente" vazia: sempre existe a sentinela.
- ▶ LIST-Delete' vira só duas linhas:
 $x.ant.prox \leftarrow x.prox;$
 $x.prox.ant \leftarrow x.ant.$

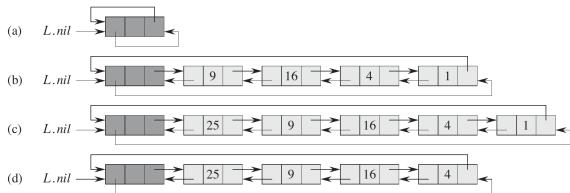


Figura 10.4 de Cormen et al. (2009) — lista circular duplamente ligada com sentinela $L.nil$: (a) vazia; (b) com 4 elementos; (c) após inserir 25; (d) após remover 1.

Busca com sentinela

- ▶ Truque clássico: guarde k **na sentinela** ($L.nil.chave \leftarrow k$). O laço sempre para nela, e o teste " $x \neq L.nil$ " sai de dentro do laço.
- ▶ Metade dos testes por elemento: $n + 1$ em vez de $2n + 1$.
- ▶ Mesma ordem de crescimento, constante menor; e código mais simples.
- ▶ Custo: um nó extra por lista (cuidado com milhares de listas pequenas).

LIST-SEARCH'(L, k)

```
1  $x \leftarrow L.nil.prox$   
2 enquanto  $x \neq L.nil$  e  $x.chave \neq k$  faça  
3    $x \leftarrow x.prox$   
4 retorne  $x$ 
```

Listas sem ponteiros: vetores

- ▶ Linguagens sem ponteiros, arquivos em disco e memória compartilhada guardam listas em vetores.
- ▶ Também cabe tudo num vetor só, com deslocamentos por campo (Figura 10.6).
- ▶ NIL vira um índice inválido (em C, -1).

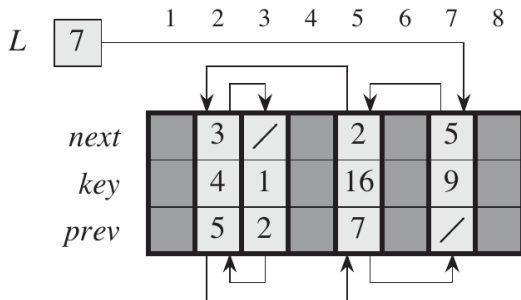


Figura 10.5 de Cormen et al. (2009) — a lista da Figura 10.3(a) em três vetores *next*, *key* e *prev*; o “ponteiro é um índice.

Alocando e liberando objetos

- ▶ As posições livres formam a **lista livre**, ligada pelo próprio *next*.
- ▶ Ela é usada como **pilha**:
ALLOCATE-Object tira do topo;
Free-Object devolve ao topo.
Ambos $O(1)$.
- ▶ É, em miniatura, o que faz malloc/free.

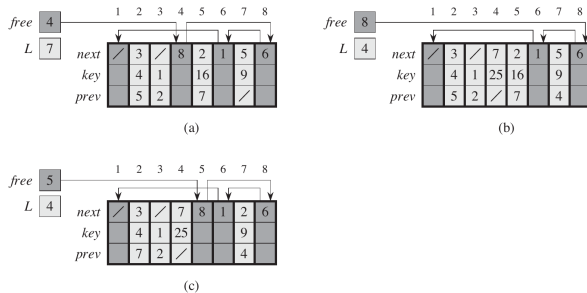


Figura 10.7 de Cormen et al. (2009) — lista livre: (a) estado inicial; (b) após ALLOCATE-Object e inserção de 25; (c) após remover 16 e Free-Object.



Prática em C

- ▶ Lista circular dupla com sentinela, como no livro
- ▶ Lista em vetores com lista livre (Figura 10.7)
- ▶ Os mesmos comandos nas duas: inserir, buscar, remover, imprimir

Lista com sentinela em C (1/4)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* A sentinela L->nil é um nó falso: nil->prox é a cabeça e nil->ant é a cauda.
5    Com ela, inserir e remover não têm casos especiais (lista vazia, extremos). */
6 typedef struct No { int chave; struct No *ant, *prox; } No;
7 typedef struct { No nil; } Lista;
8
9 void inicia(Lista *L) { L->nil.prox = L->nil.ant = &L->nil; }
```

Lista com sentinela em C (2/4)

```
11 void insere(Lista *L, int k) { // LIST-INSERT': na cabeça,  $O(1)$ 
12     No *x = malloc(sizeof(No));
13     x->chave = k;
14     x->prox = L->nil.prox;
15     L->nil.prox->ant = x;
16     L->nil.prox = x;
17     x->ant = &L->nil;
18 }
19
20 No *busca(Lista *L, int k) { // LIST-SEARCH':  $\Theta(n)$  no pior caso
21     No *x = L->nil.prox;
22     while (x != &L->nil && x->chave != k) x = x->prox;
23     return x == &L->nil ? NULL : x;
24 }
```

Lista com sentinela em C (3/4)

```
26 0
27 void remove_no(No *x) {                                // LIST-DELETE': 0(1) dado o nó
28     x->ant->prox = x->prox;
29     x->prox->ant = x->ant;
30     free(x);
31 }
32
33 int main(void) {
34     Lista L;
35     inicia(&L);
36     char cmd[4];
37     int k;
38     while (scanf("%3s", cmd) == 1) {
39         if (cmd[0] == 'i' && scanf("%d", &k) == 1) insere(&L, k);
40         else if (cmd[0] == 'b' && scanf("%d", &k) == 1)
```

Lista com sentinela em C (4/4)

```
40 0
41     printf(busca(&L, k) ? "achou %d\n" : "não achou %d\n", k);
42     else if (cmd[0] == 'r' && scanf("%d", &k) == 1) {
43         No *x = busca(&L, k);
44         if (x) remove_no(x);
45     } else if (cmd[0] == 'p') {
46         printf("lista:");
47         for (No *x = L.nil.prox; x != &L.nil; x = x->prox) printf(" %d", x->chave);
48         printf("\n");
49     }
50 }
51 while (L.nil.prox != &L.nil) remove_no(L.nil.prox);
52 return 0;
53 }
```

Lista em vetores com lista livre (1/4)

```
1  #include <stdio.h>
2
3  /* Sem ponteiros: prox[], chave[] e ant[] são vetores; um "ponteiro" é um índice e
4     -1 faz o papel de NIL. As posições livres formam uma pilha ligada por prox[]:
5     ALLOCATE-OBJECT tira do topo e FREE-OBJECT devolve, ambos O(1). */
6  #define M 8
7  int prox[M], chave[M], ant[M];
8  int livre, cabeca = -1;
9
10 int aloca(void) {                                // ALLOCATE-OBJECT
11     if (livre == -1) return -1;                  // sem espaço
12     int x = livre;
13     livre = prox[x];
14     return x;
15 }
```

Lista em vetores com lista livre (2/4)

```
16
17 void libera(int x) {                               // FREE-OBJECT
18     prox[x] = livre;
19     livre = x;
20 }
21
22 void insere(int k) {                                 // na cabeça
23     int x = aloca();
24     if (x < 0) { printf("sem espaço\n"); return; }
25     chave[x] = k;
26     prox[x] = cabeca;
27     ant[x] = -1;
28     if (cabeca >= 0) ant[cabeca] = x;
29     cabeca = x;
30 }
```

Lista em vetores com lista livre (3/4)

```
31
32 void remove_chave(int k) {
33     int x = cabeca;
34     while (x >= 0 && chave[x] != k) x = prox[x];
35     if (x < 0) return;
36     if (ant[x] >= 0) prox[ant[x]] = prox[x]; else cabeca = prox[x];
37     if (prox[x] >= 0) ant[prox[x]] = ant[x];
38     libera(x); // a posição volta para a lista livre
39 }
40
41 int main(void) {
42     for (int i = 0; i < M; i++) prox[i] = i + 1 < M ? i + 1 : -1; // tudo livre
43     livre = 0;
44     char c;
45     int k;
```

Lista em vetores com lista livre (4/4)

```
46     while (scanf(" %c", &c) == 1) {
47         if (c == 'i' && scanf("%d", &k) == 1) insere(k);
48         else if (c == 'r' && scanf("%d", &k) == 1) remove_chave(k);
49         else if (c == 'p') {
50             int usadas = 0;
51             printf("lista:");
52             for (int x = cabeca; x >= 0; x = prox[x]) {
53                 printf(" %d", chave[x]);
54                 usadas++;
55             }
56             printf("\nposições usadas: %d de %d\n", usadas, M);
57         }
58     }
59     return 0;
60 }
```



Experimento

- ▶ Sem sentinela: testa o fim e a chave em cada passo
- ▶ Com sentinela: só a chave
- ▶ Busca sem sucesso em n de 2^{10} a 2^{20}

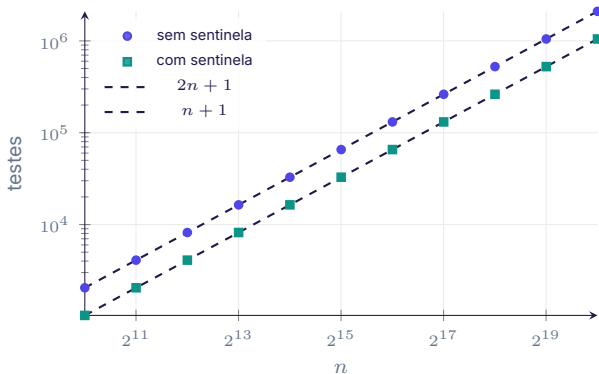
Quanto a sentinela economiza

Hipótese. Numa busca sem sucesso, sem sentinela, cada um dos n elementos custa dois testes, mais o teste final do limite: $2n + 1$. Com a chave guardada na sentinela, um teste por elemento mais o da própria sentinela: $n + 1$. O tempo deve cair, mas menos que pela metade: um dos testes é barato.

Método

1. Vetor com n pares; busca pelo valor 1 (ausente)
2. Contador em cada teste do laço
3. Tempo por elemento visitado, muitas repetições
4. $n = 2^{10}, \dots, 2^{20}$

Testes por busca sem sucesso



n	medido	razão
1.024	2.049	1,000
4.096	8.193	1,000
16.384	32.769	1,000
65.536	131.073	1,000
262.144	524.289	1,000
1.048.576	2.097.153	1,000

razão = medido $\div$ $2n + 1$

✓ desvio máx. 0,0%

Exatamente metade dos testes. A ordem de crescimento é a mesma: só a constante muda.

🧪 EXPERIMENTO

E no relógio?



Cerca de 10% mais rápido: o processador prevê o teste do limite quase de graça. A maior vantagem da sentinela é o código sem casos especiais.

EXPERIMENTO

O que o experimento mostrou

A sentinela corta os testes exatamente pela metade ($n + 1$ contra $2n + 1$) e economiza cerca de 10% do tempo. Mais importante: elimina os casos especiais de lista vazia, cabeça e cauda, que são a fonte da maioria dos erros com listas. Na representação em vetores, a lista livre faz o papel do `malloc`: alocar e liberar em $O(1)$.

➤ Reproduza: `gcc -O2 aula16_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Kernel Linux
- ▶ Alocadores de memória
- ▶ Sistemas de arquivos
- ▶ Bancos de dados

Onde isso aparece na computação



Kernel Linux

Toda lista do kernel (`list_head`) é circular, dupla e com sentinela: a "cabeça é um nó sem dados, exatamente *L.nil*.



Alocadores de memória

`malloc` mantém listas livres de blocos por tamanho; alocadores de *pool* (jogos, servidores) usam a pilha livre da Figura 10.7.



Sistemas de arquivos

O FAT do MS-DOS e de pendrives é uma lista em vetor: cada posição diz o próximo bloco do arquivo; os blocos livres formam outra lista.



Bancos de dados

Páginas livres de um arquivo de banco são encadeadas numa lista livre guardada no próprio arquivo, com índices no lugar de ponteiros.

Para casa

Exercícios

1. (10.2-4) Como está escrito, cada volta de LIST-SEARCH' faz dois testes. Elimine o teste $x \neq L.nil$.
2. (10.3-2) Escreva ALLOCATE-Object e Free-Object para a representação em um único vetor.
3. (10.3-4) Mantenha os objetos de uma lista dupla compactos nas primeiras m posições. Como remover em $O(1)$?
4. Reescreva aula15_dupla_circular.c com sentinela e compare o número de linhas e de if.
5. Use aula16_lista_vetor.c com $M = 8$ e insira 9 elementos. O que acontece? Como fazer o vetor crescer?

Leitura

Cormen et al., seções 10.2 (sentinelas) e 10.3.

Código

josedihego.net/estruturas