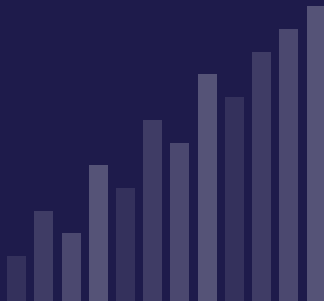

AULA 14 · BLOCO 2: DICIONÁRIOS E ESTRUTURAS LINEARES

Listas simplesmente ligadas

Seção 10.2

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Representar uma sequência com nós e ponteiros
- ▶ Inserir no início e no fim, buscar e remover
- ▶ Inverter uma lista sem memória extra
- ▶ Medir a vantagem da lista na inserção e o custo do cache no percurso



Teoria

Nós, ponteiros, cabeça e cauda; o que custa cada operação



Prática em C

Lista simples em C: insere, remove e inverte



Experimento

Inserir no início; percorrer uma lista espalhada pela memória



Conexões

Sistemas operacionais, alocação de memória, arquitetura e editores

Lista ligada

- ▶ Os elementos ficam em **nós** espalhados pela memória; cada nó tem a *chave* e um ponteiro *prox* para o seguinte.
- ▶ *L.cabeça* aponta o primeiro; o último tem *prox* = NIL. Lista vazia: *L.cabeça* = NIL.
- ▶ A ordem é dada pelos ponteiros, não pela posição na memória (ao contrário do vetor).
- ▶ Tamanho flexível: cresce e encolhe um nó por vez, sem cópias.

Variações (Cormen, 10.2): simplesmente ou duplamente ligada; ordenada ou não; linear ou circular; com ou sem sentinela. Esta aula: a mais simples.

As operações e seus custos

Operação	Como	Custo
inserir no início	novo nó aponta a cabeça; cabeça $\leftarrow$ novo	$O(1)$
inserir no fim	com ponteiro para a cauda	$O(1)$
buscar uma chave	percorrer desde a cabeça	$\Theta(n)$ no pior caso
remover dado o nó	é preciso o nó anterior	$\Theta(n)$ para achá-lo
inverter	virar cada ponteiro, uma passada	$\Theta(n)$, $O(1)$ de memória
acessar o k -ésimo	seguir k ponteiros	$\Theta(k)$ (vetor: $O(1)$)

Remover exige casos especiais: a cabeça (não há anterior) e a cauda (precisa ser atualizada).

É por isso que existe a lista duplamente ligada (aula 15): com o ponteiro *ant*, remover um nó dado custa $O(1)$.

Do vetor à lista

Vetor: acesso por posição em $O(1)$, inserir no meio desloca $\Theta(n)$, memória contígua (amiga do cache). **Lista:** acesso por posição $\Theta(n)$, inserir dado o lugar em $O(1)$, memória espalhada (um nó por vez).

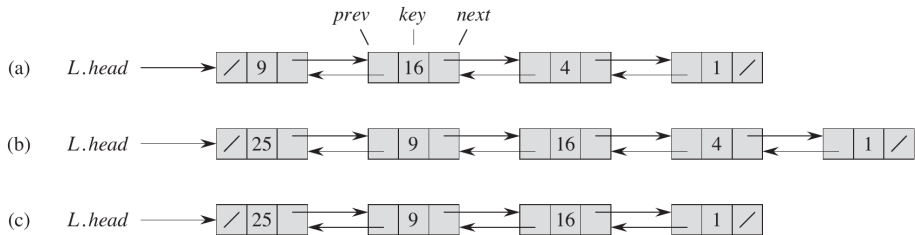


Figura 10.3 de Cormen et al. (2009) — lista duplamente ligada $\langle 9, 16, 4, 1 \rangle$, inserção de 25 e remoção de 4. Na lista simples, só existem os ponteiros *next*.



Prática em C

- ▶ Cabeça e cauda: inserir nas duas pontas em $O(1)$
- ▶ Remoção guardando o anterior; os casos da cabeça e da cauda
- ▶ Inversão em uma passada (exercício 10.2-7)

Lista simplesmente ligada em C (1/5)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Cada nó só conhece o próximo. Guardamos cabeça e cauda: inserir no início e no
5     fim custa  $O(1)$ ; remover exige o nó ANTERIOR, então a remoção por chave é  $O(n)$ .
6     Comandos: c k (insere no começo), f k (no fim), r k (remove), i (inverte), p. */
7  typedef struct No { int chave; struct No *prox; } No;
8  typedef struct { No *cabeca, *cauda; } Lista;
9
10 void insere_comeco(Lista *L, int k) {
11     No *x = malloc(sizeof(No));
12     x->chave = k;
13     x->prox = L->cabeca;
14     L->cabeca = x;
15     if (!L->cauda) L->cauda = x;           // lista estava vazia
16 }
```

Lista simplesmente ligada em C (2/5)

```
17
18 void insere_fim(Lista *L, int k) {
19     No *x = malloc(sizeof(No));
20     x->chave = k;
21     x->prox = NULL;
22     if (L->cauda) L->cauda->prox = x; else L->cabeca = x;
23     L->cauda = x;
24 }
```

Lista simplesmente ligada em C (3/5)

```
26 void remove_chave(Lista *L, int k) {                               // procura mantendo o anterior
27     No *ant = NULL, *x = L->cabeca;
28     while (x && x->chave != k) { ant = x; x = x->prox; }
29     if (!x) return;                                                // não está na lista
30     if (ant) ant->prox = x->prox; else L->cabeca = x->prox;      // era a cabeça?
31     if (L->cauda == x) L->cauda = ant;                             // era a cauda?
32     free(x);
33 }
34
35 //  $\theta(n)$ , sem memória extra (exercício 10.2-7)
36 void inverte(Lista *L) {
37     No *ant = NULL, *x = L->cabeca;
38     L->cauda = x;
39     while (x) {
40         No *prox = x->prox;
```

Lista simplesmente ligada em C (4/5)

```
41 0
42     x->prox = ant;                                // vira o ponteiro para trás
43     ant = x;
44     x = prox;
45 }
46 L->cabeca = ant;
47 }
48
49 int main(void) {
50     Lista L = {NULL, NULL};
51     char c;
52     int k;
53     while (scanf(" %c", &c) == 1) {
54         if (c == 'c' && scanf("%d", &k) == 1) insere_comeco(&L, k);
```

Lista simplesmente ligada em C (5/5)

```
54 0
55     else if (c == 'f' && scanf("%d", &k) == 1) insere_fim(&L, k);
56     else if (c == 'r' && scanf("%d", &k) == 1) remove_chave(&L, k);
57     else if (c == 'i') inverte(&L);
58     else if (c == 'p') {
59         printf("lista:");
60         if (!L.cabeca) printf(" (vazia)");
61         for (No *x = L.cabeca; x; x = x->prox) printf(" %d", x->chave);
62         printf("\n");
63     }
64 }
65 while (L.cabeca) { No *x = L.cabeca; L.cabeca = x->prox; free(x); }
66 return 0;
67 }
```



Experimento

- ▶ Inserir n vezes no início: vetor desloca $n(n-1)/2$; lista, nada
- ▶ Percorrer: tempo por elemento de vetor e de lista, n até 2^{24}
- ▶ Nós da lista em ordem aleatória na memória

Ponteiros contra vizinhança

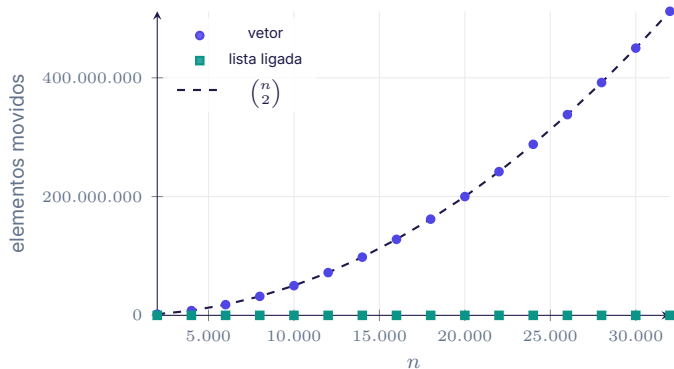
Hipótese. Inserir no início de um vetor desloca todos os elementos: n inserções movem $n(n - 1)/2$ elementos; a lista só ajusta ponteiros. Para percorrer, os dois são $\Theta(n)$ no modelo RAM. Mas a CPU busca a memória em blocos (linhas de cache): o vetor aproveita cada bloco, a lista espalhada não. Esperamos o mesmo $\Theta(n)$ com **constantes muito diferentes**.

Método

1. Contagem exata dos deslocamentos
2. Percurso: $n = 2^{10}$ a 2^{24} ; cerca de 33 milhões de visitas por medida
3. Nós da lista embaralhados (Fisher–Yates)
4. Tempo dividido pelo número de visitas

🧪 EXPERIMENTO

Inserir no início



n	medido	razão
2.000	1.999.000	1,000
8.000	31.996.000	1,000
14.000	97.993.000	1,000
20.000	199.990.000	1,000
26.000	337.987.000	1,000
32.000	511.984.000	1,000

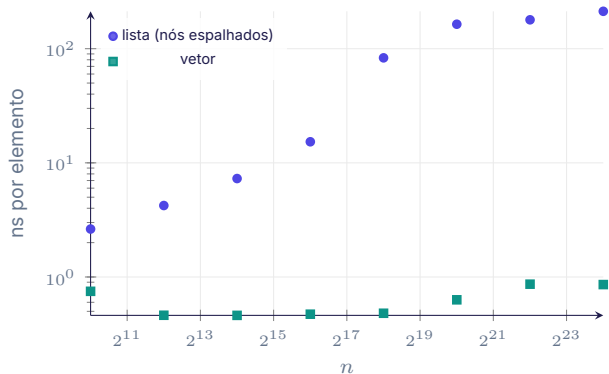
$$\text{razão} = \text{medido} \div \binom{n}{2}$$

✓ desvio máx. 0,0%

O vetor move exatamente $n(n - 1)/2$ elementos; a lista, zero.

🧪 EXPERIMENTO

Percorrer: o cache decide



Enquanto tudo cabe no cache, a lista custa 1 a 4 ns por nó. Fora do cache (milhões de nós), passa de 100 ns: cada nó é uma ida à RAM.

🧪 EXPERIMENTO

O que o experimento mostrou

Inserir no início confirma a teoria: $n(n-1)/2$ movimentos no vetor, nenhum na lista. No percurso, os dois são $\Theta(n)$, mas a lista espalhada fica até **200 vezes** mais lenta que o vetor quando sai do cache. **O modelo RAM não vê a hierarquia de memória**; a medição vê.

➤ Reproduza: `gcc -O2 aula14_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Sistemas operacionais
- ▶ Alocação de memória
- ▶ Arquitetura de computadores
- ▶ Linguagens funcionais

Onde isso aparece na computação

Sistemas operacionais

Filas de processos, listas de arquivos abertos e de páginas livres são listas ligadas dentro do kernel; a inserção $O(1)$ é essencial.

Alocação de memória

`malloc` encadeia os blocos livres numa lista; programas em C que vazam memória costumam "perder a cabeça de uma lista.

Arquitetura de computadores

O experimento mostra a hierarquia de memória: L1 (≈ 1 ns), L2, L3 e RAM (≈ 100 ns). Estruturas *cache-friendly* vencem na prática.

Linguagens funcionais

Em Lisp, Haskell e Erlang a lista ligada é a estrutura básica: inserir na cabeça cria uma lista nova compartilhando o resto, sem copiar nada.

Para casa

Exercícios

1. (10.2-1) É possível inserir em $O(1)$ numa lista simplesmente ligada? E remover? Em que condições?
2. (10.2-3) Implemente uma fila com lista simplesmente ligada; Enqueue e Dequeue em $O(1)$.
3. (10.2-7) Inverta uma lista simplesmente ligada em $\Theta(n)$, sem memória extra.
4. Ache o elemento do meio de uma lista numa só passada (dois ponteiros, um duas vezes mais rápido).
5. No experimento, aloque os nós em ordem (sem embaralhar). O que acontece com o tempo da lista? Por quê?

Leitura

Cormen et al., seção 10.2.

Código

josedihego.net/estruturas