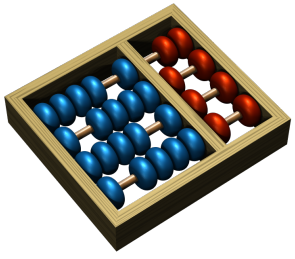
MC202 - Estruturas de Dados

Emilio Francesquini
francesquini@ic.unicamp.br

Instituto de Computação - UNICAMP

Aula 03 - 10 de março de 2017



UNICAMP

Disclaimer

- Esses slides foram preparados para um curso de Estrutura de Dados ministrado na UNICAMP
- Este material pode ser usado livremente desde que sejam mantidos os créditos dos autores e da instituição.
- Muitos dos exemplos apresentados aqui foram retirados de materiais preparados pelos Profs. Tomasz Kowaltowski e Orlando Lee da UNICAMP assim como do Prof. Paulo Feofiloff do IME-USP

Ordenação em listas ligadas

```
struct No {
    int info;
    struct No *prox;
}

typedef struct No No;
```

- Vamos descrever três algoritmos para ordenar uma lista ligada com cabeça pelo campo info.
- Vocês já conhecem alguns desses algoritmos implementados com vetores!

Ordenação por seleção

Ordenação por seleção (=*selection sort*)

- A ideia básica consiste em cada iteração:
  - encontrar e remover o menor elemento x da lista original ini
  - inserir x no final de uma lista ordenada formada pelos elementos previamente removidos;
- Vamos trabalhar usando uma lista com cabeça

Ordenação por seleção

```
void selectionsort(No *ini) {
        faça t apontar para a lista ini;
        faça ini ser a lista vazia;
        enquanto t for não vazia {
                remova o menor elemento de t;
                insira o elemento no final de ini;
        }
}
```

- Note que é conveniente ter um ponteiro para o final da lista.

Ordenação por seleção

```
void selectionsort(No *ini) {
    No *t = malloc(sizeof(No)), *last, *min;

    t->prox = ini->prox;
    ini->prox = NULL;
    last = ini;
    while (t->prox != NULL) {
        min = remove_minimo(t);
        last->prox = min;
        last = min;
        last->prox = NULL;
    }
    free(t);
}
```

Ordenação por seleção

```
No *remove_minimo(No *ini) {
    No *p, *q, *ant;
    if (!ini->prox) return NULL;
    p = ant = ini; q = ini->prox;
    while (q) {
        if (q->info < ant->prox->info) ant = p;
        p = q; q = q->prox;
    }
    q = ant->prox;
    ant->prox = q->prox;
    return q;
}
```

- Na função, ant aponta para o nó anterior ao nó com menor info encontrado até o momento.

Ordenação por seleção

```
No *remove_minimo(No *ini) { /* outro jeito */
    No **p, **min, *q;
    if (!ini->prox) return NULL;
    p = min = &(ini->prox);
    while (*p) {
        if ((*p)->info < (*min)->info) min = p;
        p = &((*p)->prox);
    }
    q = (*min);
    *min = q->prox;
    return q;
}
```

- As variáveis p e min são ponteiros para campos prox de nós da lista.

UNICAMP

Ordenação por seleção - Análise

- Podemos analisar a complexidade de selectionsort em função de n, o tamanho da lista ini
- O tempo gasto por remove_minimo é proporcional a n
- selectionsort chama remove_minimo n vezes
- Assim, no total o tempo gasto é proporcional a n^2

Exercícios

Exercício. Escreva uma versão de

```
No *selectionsort(No *ini);
```

que recebe uma lista ligada **sem cabeça** ini e devolve um ponteiro para uma lista ligada ordenada sem cabeça com os nós da lista original.

Sugestão: em vez de usar uma função `remove_minimo` é melhor escrever o código que faz isto dentro da função `selectionsort`, para evitar o problema de ter de remover um nó no início da lista. Não fica tão modularizado, obviamente. Outra ideia é usar um ponteiro para ponteiro como parâmetro.

UNICAMP

Ordenação por seleção - Forma alternativa

```
void selectionsort(No *ini) {
        faça t apontar para a lista ini;
        faça ini ser a lista vazia;
        enquanto t for não for vazia {
                remova o maior elemento de t;
                insira o elemento no início de ini;
        }
}
```

Exercício: Escreva o código em C supondo que a lista ligada tem cabeça.

Ordenação por inserção

Ordenação por inserção (=*insertion sort*)

- A ideia básica consiste em cada iteração:
  - remover o primeiro elemento x da lista original ini;
  - inserir x na posição correta de uma lista ordenada formada pelos elementos previamente removidos;
- Na implementação em C, passamos o nó cabeça da lista que queremos ordenar e este deve ser o nó cabeça da lista ordenada.

Ordenação por inserção

```
 void insertionsort(No *ini) {
        faça t apontar para a lista ini;
        faça ini ser a lista vazia;
        enquanto t for não vazia {
                remova o primeiro elemento de t;
                insira o elemento em ordem na lista ini;
        }
}
```

Ordenação por inserção

```
void insertionsort(No *ini) {
    No *t, *x;
    t = ini->prox;
    ini->prox = NULL;
    while (t) {
        x = t;
        t = t->prox;
        insere_ordenado(ini, x);
    }
}
```

- A lista t não tem cabeça

Ordenação por inserção

```
void insere_ordenado(No *ini, No *x) {
    No *p, *q,
    p = ini;
    q = ini->prox;
    while (q && q->info < x->info) {
        p = q;
        q = q->prox;
    }
    x->prox = q;
    p->prox = x;
}
```

Exercício. Escreva uma versão de insere_ordenado que usa ponteiros para ponteiros (como na segunda versão de remove_minimo).

Ordenação por inserção - Análise

- Podemos analisar a complexidade de insertionsort em função de n, o tamanho da lista ini.
- O tempo gasto por insere_ordenado é proporcional a n no pior caso.
- insertionsort chama insere_ordenado n vezes.
- Assim, no total o tempo gasto é proporcional a no máximo n^2

Intercalando duas listas

Problema: intercalar (=*merge*) duas listas ordenadas

Queremos implementar a seguinte função:

```
No *intercala(No *s, No *t);
```

A função merge recebe duas **listas ligadas ordenadas sem cabeça** s e t, e devolve uma **lista ordenada sem cabeça** contendo os nós das listas s e t.

Podemos usar também listas com cabeça: o código é praticamente idêntico. Só é preciso descartar (free) uma das cabeças ao final.

Intercalando duas listas

```
No *intercala(No *s, No *t) {
    No *p, *q, cabeca, *last;
    last = &cabeca; cabeca.prox = NULL;
    p = s; q = t;
    while (p && q) {
        if (p->info < q->info) {
            last->prox = p; last = p; p = p->prox;
        } else {
            last->prox = q; last = q; q = q->prox;
        }
    }
    if (p)
        last->prox = p;
    else
        last->prox = q;
    return cabeca.prox;
}
```

Mergesort

A ideia do algoritmo mergesort é conceitualmente muito simples.

- Se a lista ini for pequena, então ordene diretamente.
- Caso contrário, divida a lista ini em duas listas s e t de tamanhos aproximadamente iguais.
- Recursivamente, ordene as listas s e t.
- Intercale as listas s e t e devolva a lista resultante.

Mergesort

```
No *mergesort(No *ini) {
    No *met, *metade;
    if (ini == NULL || ini->prox == NULL) /* caso base */
        return ini;
    metade = acha_metade(ini);
    met = metade->prox;
    metade->prox = NULL; /* divide a lista */
    ini = mergesort(ini);
    met = mergesort(met);
    return intercala(ini, met);
}
```

Mergesort

Como achar a metade rapidamente?

```
No *acha_metade(No *ini) {
    No *slow, *fast;
    if (ini == NULL) return ini;
    slow = fast = ini;
    while (fast->prox && fast->prox->prox) {
        slow = slow->prox;
        fast = fast->prox->prox;
    }
    return slow;
}
```

Mergesort - Análise (informal)

Suponha que T(n) seja o tempo de pior caso que o mergesort leva para ordenar uma lista ligada com n nós.

- Se $n \geq 2$ então acontecem os seguintes passos:
  dividir a lista + 2 chamadas recursivas + intercalação
- Logo o tempo gasto é
  $T(n) = n + 2T(n/2) + n = 2T(n/2) + 2n$
- Assim,

$$T(n) = \begin{cases} 2T(n/2) + 2n, & \text{se } n \geq 2 \\ 1, & \text{caso contrário} \end{cases}$$

Mergesort - Análise (informal)

Para ter uma intuição, suponha que $n = 2^k$. Note que há:

- 1 chamada para uma lista de tamanho n (nível 1)
- 2 chamadas para uma lista de tamanho n/2 (nível 2)
- 4 chamadas para uma lista de tamanho n/4 (nível 3)
- 2^i chamadas para uma lista de tamanho n/2 i (nível i)
- $2^k - 1$ chamadas para uma lista de tamanho 2 (nível k – 1)
- 2^k chamadas para uma lista de tamanho 1 (nível k)

UNICAMP

Mergesort - Análise (informal)

Ignore o tempo de dividir a lista. Vamos contar o custo das intercalações de listas (IL) em cada nível.

- nível 1: 1 IL de tamanho n/2; tempo $2 \times (n/2)$
- nível 2: 2 IL de tamanho n/4; tempo $2 \times 2 \times (n/4) = n$
- nível 3: 4 IL de tamanho n/8; tempo $2 \times 4 \times (n/8) = n$
- nível i: 2^{i-1} IL de tamanho $n/2^i$; tempo $2 \times 2^{i-1} \times (n/2^i) = n$
- nível k – 1: 2^{k-2} IL de tamanho 2; tempo $2 \times 2^{k-2} \cdot (2) = n$
- nível k: 2^{k-1} IL de tamanho 1; tempo $2 \times 2^{k-1} \times 1 = n$

Assim, o tempo total é $(k - 1)\, n = n \log_2 n$

UNICAMP

Mergesort - Análise (informal)

- Para n grande, temos que $n \log n \ll n^2$.
- Assim, o algoritmo mergesort é bem mais eficiente que os algoritmos quadráticos bubblesort, selectionsort e insertionsort.
- Veremos depois uma versão de mergesort para ordenar vetores.
- Veremos também outros algoritmos de ordenação mais sofisticados (para vetores).

Exercício Escreva uma versão **recursiva** da função `No *intercala(No *s, No *t)` que não use laços.

Comparação entre vetores e listas ligadas

O que é melhor usar: vetores ou listas ligadas? Depende

- Vetores permitem indexação e usar menos memória (não precisa de ponteiros)
- Listas ligadas são mais flexíveis
  - Não é preciso saber o número de elementos a priori
  - Algumas operações podem ser mais simples

Comparação entre vetores e listas ligadas

Exemplo Suponha que queremos manter um conjunto S de inteiros que suportam as seguintes operações:

- mínimo
- k-ésimo menor
- busca
- inserção
- remoção

Denote por $n = |S|$.

Vetores vs. listas ligadas

Implementação como um vetor ordenado:

- mínimo: custo 1
- k-ésimo menor: custo 1
- busca: custo $\leq \log_2 n$ (busca binária)
- inserção: custo $\leq n$ (busca + movimentação de dados)
- remoção: custo $\leq n$ (busca + movimentação de dados)

Implementação como uma lista ligada ordenada:

- mínimo: custo 1
- k-ésimo menor: custo k
- busca: custo $\leq n$
- inserção: custo $\leq n$ (busca)
- remoção: custo $\leq n$ (busca)

UNICAMP

Listas duplamente ligadas

```
struct NoDuplo {
    int info;
    struct NoDuplo *ant;
    struct NoDuplo *prox;
};
typedef struct NoDuplo NoD;
```

Vantagens: maior acessibilidade
Desvantagens: dobro de ponteiros e mais trabalho para manter a lista.

Busca em uma lista duplamente ligada

A função busca recebe uma lista duplamente ligada ini e um inteiro k, e devolve um ponteiro para o primeiro nó com chave k ou NULL, se não houver.

```
No *busca(NoD *ini, int k) {
    NoD *p = ini;
    while (p && p->info != k)
        p = p->prox;
    return p;
}
```

- Virtualmente idêntico à função busca para lista ligadas simples

Inserção - lista duplamente ligada

A função insere recebe uma lista duplamente ligada ini e um elemento k e insere um novo nó com conteúdo k entre o nó apontado por p e o seguinte. Só faz sentido se p != NULL.

```
void insere(NoD *p, int k) {
    NoD *novo;
    novo = malloc(sizeof(NoD));
    novo->info = k;
    novo->prox = p->prox;
    novo->ant = p;
    if (p->prox) p->prox->ant = novo;
    p->prox = novo;
}
```

- Não faz inserção no início de uma lista duplamente ligada, a não ser que tenha nó cabeça

Remoção em lista duplamente ligada

A função remove recebe um ponteiro q para um nó de uma lista duplamente ligada e o remove.

```
void remove(NoD *q) {
    No *p = q->ant;
    p->prox = q->prox;
    if (q->prox) q->prox->ant = p;
    free(q);
}
```

- Note a diferença com listas ligadas simples.
- Não funciona se q aponta para o primeiro elemento da lista, a não ser que tenha nó cabeça.

Listas ligadas circulares

Tem a mesma declaração de uma lista ligada simples, mas o campo prox do último nó aponta para o primeiro nó.
Lista ligada circular sem cabeça: um problema é a lista vazia.
Lista ligada circular com cabeça ini:

- Lista vazia: ini->prox == ini

Pode-se implementar outras variantes:

- listas ligadas circulares com ou sem cabeça
- listas duplamente ligadas circulares
- ou com ambas as formas.

Busca em lista circular

A função busca recebe uma lista ligada circular com cabeça ini e um inteiro k, e devolve um ponteiro para o primeiro nó com chave k ou NULL, se não houver.

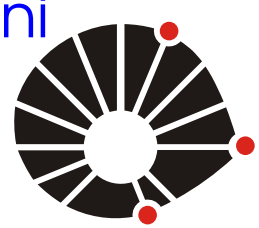
```
No *busca(No *ini, int k) {
    No *p = ini->prox;
    ini->info = k; /* sentinela */
    while (p->info != k)
        p = p->prox;
    ini->info = NULL;
    if (p == ini)
        return NULL;
    else
        return p;
}
```

Manipulando listas ligadas circulares

Exercício 1. Escreva uma função remove_llcc(No *ini, int k) que recebe um ponteiro para uma lista ligada circular com cabeça ini e remove o primeiro nó com chave k. Use o método do sentinela.

Exercício 2. Escreva uma função remove_ldlcc(NoD *p) que recebe um ponteiro para um nó p de uma lista duplamente ligada circular com nó cabeça e o remove da lista. Naturalmente, suponha que p não é o nó cabeça da lista.

Exercício 3. Escreva uma função insere_lcco(No *ini, int k) que recebe uma lista ligada circular com cabeça ordenada ini e insere um novo nó com chave k na posição correta.



UNICAMP

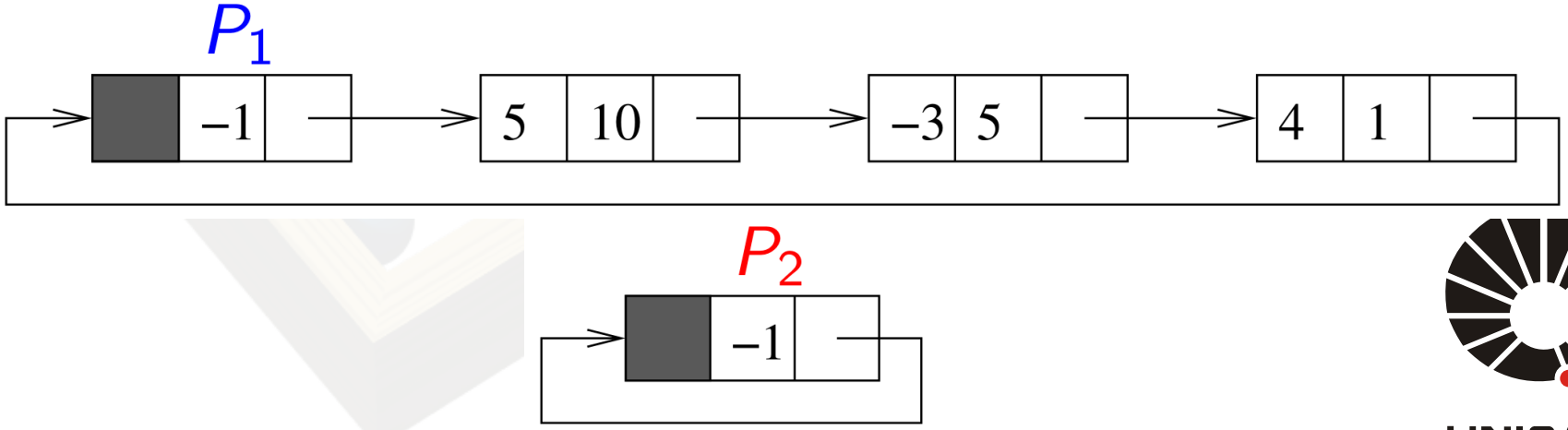
Representando polinômios

Considere o polinômio:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

onde $a_n \neq 0$

Representação de $P_1(x) = 5x^{10} - 3x^5 + 4$ e $P_2(x) = 0$



UNICAMP

Representando polinômios

```
typedef struct AuxPol {
    float coef;
    int expo;
    struct AuxPol *prox;
} Termo, *Polin;
```

A lista tem cabeça. O campo expo da cabeça é igual a -1 para ser usado como sentinela

Note que

```
Termo *p;
Polin p;
```

são declarações equivalentes

UNICAMP

Representando polinômios

A função imprime(Polin p) imprime um polinômio p exibindo os pares (coef, expo) de cada termo.

```
void imprime(Polin pol) {
    Termo *p = pol->prox;
    if (p == pol) {
        printf("Polinomio nulo.\n");
        return;
    }
    while (p->expo != -1) {
    printf("%5.1f, %2d ", p->coef, p->expo);
    p = p->prox;
    }
    printf("\n");
}
```

Rotinas de manipulação

- Calcular o valor de um polinômio P(x) em um ponto x_0 ,
- Calcular a soma de dois polinômios (usando o método da intercalação),
- Calcular o produto de dois polinômios,
- Calcular a k-ésima derivada de um polinômio
- etc.

Exercício Implemente essas funções.

Problema de Josephus

- Um grupo de N pessoas precisa eleger um líder.
- Decidiu-se usar a seguinte ideia para eleger um líder: forma-se um círculo com as N pessoas e escolhe-se um inteiro k. Começamos com uma pessoa qualquer e percorremos o círculo em sentido horário, eliminando cada k-ésima pessoa. A última pessoa que restar será o líder. Veja o verbete sobre Josephus na Wikipedia.

Problema de Josephus: coloque os números $1, 2, \ldots, N$ em um círculo nesta ordem e começando em 1 aplique o algoritmo acima com um valor k. Determine o último número, denotado $J(N, k)$.

Exercício Escreva uma função Josephus(int N, int k) que calcula $J(N, k)$.

UNICAMP