MC202 – Estruturas de Dados

Orlando Lee – UNICAMP

18 de setembro de 2015

Disclaimer

- Esses slides foram preparados para um curso de Estrutura de Dados ministrado na UNICAMP.
- Este material pode ser usado livremente desde que sejam mantido os créditos dos autores e da instituição.
- Muitos dos exemplos apresentados aqui foram retirados de uma apostila ou slides preparados pelo Prof. Tomasz Kowaltowski da UNICAMP. Outros exemplos foram retirados do livro "Algoritmos em Linguagem C" de Paulo Feofiloff, Editora Campus.

Lista ligadas

- Uma lista ligada é uma representação de uma sequência de elementos ou objetos na memória do computador.

Lista ligadas

- Uma lista ligada é uma representação de uma sequência de elementos ou objetos na memória do computador.
- Cada elemento da sequência é armazenado em um nó da lista.

Lista ligadas

- Uma lista ligada é uma representação de uma sequência de elementos ou objetos na memória do computador.
- Cada elemento da sequência é armazenado em um nó da lista.
- Os nós não ficam necessariamente em posições consecutivas da memória. Assim, a ordem relativa entre esses tem que ser determinada de outra forma.

Implementação de listas ligadas

Cada nó é uma estrutura (struct) com dois campos:

Implementação de listas ligadas

Cada nó é uma estrutura (struct) com dois campos:

- um elemento (conteúdo do nó)

Implementação de listas ligadas

Cada nó é uma estrutura (struct) com dois campos:

- um elemento (conteúdo do nó)
- um ponteiro para o próximo nó da lista.

Implementação de listas ligadas

Cada nó é uma estrutura (struct) com dois campos:

- um elemento (conteúdo do nó)
- um ponteiro para o próximo nó da lista.

  No caso do último nó da lista, seu ponteiro é `NULL`.

Implementação de listas ligadas

Cada nó é uma estrutura (struct) com dois campos:

- um elemento (conteúdo do nó)
- um ponteiro para o próximo nó da lista.

  No caso do último nó da lista, seu ponteiro é `NULL`.

Na prática, cada nó teria outro(s) campo(s) com outros dados (dados satélite). O elemento seria um identificador/chave deste conjunto de dados.

Implementação de listas ligadas

Cada nó é uma estrutura (struct) com dois campos:

- um elemento (conteúdo do nó)
- um ponteiro para o próximo nó da lista.

  No caso do último nó da lista, seu ponteiro é `NULL`.

Na prática, cada nó teria outro(s) campo(s) com outros dados (dados satélite). O elemento seria um identificador/chave deste conjunto de dados.

Nos exemplos que usaremos, os elementos serão inteiros.

Implementação de listas ligadas

```
struct No {
    int info;
    struct No *prox;  /* definição recursiva */
}
typedef struct No No;
```

Implementação de listas ligadas

```
struct No {
   int info;
   struct No *prox;  /* definição recursiva */
}
typedef struct No No;
```

Implementação de listas ligadas

```
struct No {
   int info;
   struct No *prox;  /* definição recursiva */
}
typedef struct No No;

No x; /* isto é um nó */
```

- `x.info` é o conteúdo do nó `x`,
- `x.prox` é o ponteiro para o nó que vem depois de `x`.

Implementação de listas ligadas

```
struct No {
   int info;
   struct No *prox;  /* definição recursiva */
}

typedef struct No No;

No *p; /* isto é um ponteiro para um nó */
```

- `p->info` é o conteúdo do nó (apontado por) `p`,
- `p->prox` é o ponteiro para o nó que vem depois de `*p`.

Ponteiros para listas ligadas

- O ponteiro de uma lista ligada é o ponteiro do primeiro nó.

Ponteiros para listas ligadas

- O ponteiro de uma lista ligada é o ponteiro do primeiro nó.
- Se p é o ponteiro de uma lista, dizemos que p é uma lista.

Ponteiros para listas ligadas

- O ponteiro de uma lista ligada é o ponteiro do primeiro nó.
- Se `p` é o ponteiro de uma lista, dizemos que `p` é uma lista.
- Uma lista ligada `p` é vazia se `p == NULL`.

Implementação de listas ligadas

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

ou

```
typedef struct No {
   int info;
   struct No *prox;
} No, *Lista;
```

Implementação de listas ligadas

```
typedef struct No {
   int info;
   struct No *prox;
} No, *Lista;
No *p;
Lista p; /* outra forma de declarar */
```

Do ponto de vista de linguagem C, as duas declarações são equivalentes. Entretanto, é útil ter as duas formas para distinguir se estamos interessados em um nó ou em uma lista.

```
void imprime_lista(Lista p);
void imprime_no(No *p);
```

Natureza recursiva de listas ligadas

Para uma lista ligada `p`, temos que

Natureza recursiva de listas ligadas

Para uma lista ligada `p`, temos que

- `p` é NULL, ou

Natureza recursiva de listas ligadas

Para uma lista ligada `p`, temos que

- `p` é NULL, ou
- `p->prox` é uma lista ligada.

Natureza recursiva de listas ligadas

Para uma lista ligada `p`, temos que

- `p` é NULL, ou
- `p->prox` é uma lista ligada.

Muitos algoritmos que manipulam listas ligadas são simples de descrever em forma recursiva.

Natureza recursiva de listas ligadas

Para uma lista ligada `p`, temos que

- `p` é NULL, ou
- `p->prox` é uma lista ligada.

Muitos algoritmos que manipulam listas ligadas são simples de descrever em forma recursiva.

Algumas linguagens de programação como LISP trabalham apenas com listas e usam recursão frequentemente.

Impressão de uma lista ligada

```
void imprime(No *ini) {
    No *p;
    p = ini;
    while (p) {
        printf("%d ", p->info);
        p = p->prox;
   }
}
```

Impressão de uma lista ligada

```
void imprime(No *ini) {
   No *p;
   p = ini;
   while (p) {
      printf("%d ", p->info);
      p = p->prox;
  }
}

void imprime(Lista ini) {
   No *p;
   p = ini;
   while (p) {
      printf("%d ", p->info);
      p = p->prox;
  }
}
```

Impressão de uma lista ligada

```
void imprimerec(No *p) { /* versao recursiva */
   if (p) {
     printf("%d ", p->info);
     imprimerec(p->prox);
   }
}
void imprimerec(Lista p) { /* versao recursiva */
   if (p) {
     printf("%d ", p->info);
     imprimerec(p->prox);
   }
}
```

Exercício. Escreva uma função que recebe uma lista ligada e imprime os elementos da lista em ordem inversa.

Manipulação de listas ligadas

Listas ligadas não caem do céu. Como se constrói ou se trabalha com listas?

Manipulação de listas ligadas

Listas ligadas não caem do céu. Como se constrói ou se trabalha com listas?

Operações típicas de estruturas de dados:

- busca
- inserção
- remoção

Busca em uma lista

A função `busca` recebe uma lista `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;
   while (p && p->info != k)
     p = p->prox;
  return p;
}
```

Simples e elegante.

Busca recursiva em uma lista

A função `buscarec` recebe uma lista `p` e um inteiro `k`, e devolve um ponteiro para o nó com chave `k` ou NULL, se não houver.

```
No *buscarec(No *p, int k) {
   if (!p || p->info == k)
      return p;
   return buscarec(p->prox, k);
}
```

Simples e elegante também.

Inserção em uma lista

A função `insere` recebe um ponteiro `p` de um nó de uma lista ligada 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(No *p, int k) {
    No *novo;
    novo = malloc(sizeof(No));
    novo->info = k;
    novo->prox = p->prox;
    p->prox = novo;
}
```

Inserção no início de uma lista

```
void insere_inicio(No *ini, int k) {
    No *novo;
    novo = malloc(sizeof(No));
    novo->info = k;
    novo->prox = ini;
    ini = novo;
}
```

Inserção no início de uma lista

```
void insere_inicio(No *ini, int k) {
    No *novo;
    novo = malloc(sizeof(No));
    novo->info = k;
    novo->prox = ini;
    ini = novo;
}
```

Isto não funciona. Por quê?

inserção no início de uma lista

```
No *primeiro;
int k = 7;
...
insere_inicio(primeiro, k);  /* cria cópia de primeiro */
```

Ao voltar da chamada, o valor da variável primeiro não foi alterado.

inserção no início de uma lista

```
No *primeiro;
int k = 7;
...
insere_inicio(primeiro, k);  /* cria cópia de primeiro */
```

Ao voltar da chamada, o valor da variável primeiro não foi alterado.

Como resolver este problema?

Primeira solução

```
No* insere_inicio(No *ini, int k) {
   No *novo;
   novo = malloc(sizeof(No));
   novo->info = k;
   novo->prox = ini;
   return novo; /* devolve o início da lista */
}
...
primeiro = insere_inicio(primeiro, k);
```

Primeira solução

```
No* insere_inicio(No *ini, int k) {
   No *novo;
   novo = malloc(sizeof(No));
   novo->info = k;
   novo->prox = ini;
   return novo; /* devolve o início da lista */
}
...
primeiro = insere_inicio(primeiro, k);
```

Solução um tanto artificial.

Segunda solução

```
void insere_inicio(No **pini, int k) {
    No *novo;
    novo = malloc(sizeof(No));
    novo->info = k;
    novo->prox = *pini;
    *pini =  novo;
}
```

Segunda solução

```
void insere_inicio(No **pini, int k) {
    No *novo;
    novo = malloc(sizeof(No));
    novo->info = k;
    novo->prox = *pini;
    *pini =  novo;
}
```

Um pouco difícil de ler por causa da indireção.

Nenhuma das soluções é satisfatória porque elas diferem do caso geral.

Remoção em uma lista

Podemos passar um ponteiro para o nó que queremos remover.
Isto não é bom. Por quê?

Remoção em uma lista

Podemos passar um ponteiro para o nó que queremos remover.
Isto não é bom. Por quê?

Uma ideia melhor é passar um ponteiro para o nó que precede o nó que queremos remover.

Remoção em uma lista

Podemos passar um ponteiro para o nó que queremos remover.
Isto não é bom. Por quê?

Uma ideia melhor é passar um ponteiro para o nó que precede o nó que queremos remover.

A função `remove` recebe uma lista `p` e remove o nó que vem depois do nó apontado por `p`. Só faz sentido se `p != NULL`.

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

Remoção no início de uma lista

- A remoção de um nó no início de uma lista tem problemas similares ao da inserção no início de uma lista.

Remoção no início de uma lista

- A remoção de um nó no início de uma lista tem problemas similares ao da inserção no início de uma lista.
- Podemos escrever rotinas específicas para este caso, como feito no caso da inserção.

Remoção no início de uma lista

- A remoção de um nó no início de uma lista tem problemas similares ao da inserção no início de uma lista.
- Podemos escrever rotinas específicas para este caso, como feito no caso da inserção.
- Porém, há outra solução que permite uniformizar todos os casos, ao custo de gastar um nó extra.

Listas ligada com (nó) cabeça

- A ideia é sempre manter um nó cabeça no início da lista ligada. Ele não guarda informação nenhuma, mas garante que a inserção ou remoção nunca ocorre no início da lista!

Listas ligada com (nó) cabeça

- A ideia é sempre manter um nó cabeça no início da lista ligada. Ele não guarda informação nenhuma, mas garante que a inserção ou remoção nunca ocorre no início da lista!
- A lista é dada então pelo endereço `ini` do nó cabeça.

Listas ligada com (nó) cabeça

- A ideia é sempre manter um nó cabeça no início da lista ligada. Ele não guarda informação nenhuma, mas garante que a inserção ou remoção nunca ocorre no início da lista!
- A lista é dada então pelo endereço `ini` do nó cabeça.
- Como o nó cabeça nunca é removido, o primeiro nó da lista é sempre o mesmo, eliminando os problemas descritos antes.

Listas ligada com (nó) cabeça

- A ideia é sempre manter um nó cabeça no início da lista ligada. Ele não guarda informação nenhuma, mas garante que a inserção ou remoção nunca ocorre no início da lista!
- A lista é dada então pelo endereço `ini` do nó cabeça.
- Como o nó cabeça nunca é removido, o primeiro nó da lista é sempre o mesmo, eliminando os problemas descritos antes.
- Uma lista com nó cabeça `ini` está vazia se:

  `ini->prox == NULL.`

Listas ligadas com (nó) cabeça

```
No cabeca, *ini;
cabeca.prox = NULL;
ini = cabeca;
```

ou

```
No *ini;
ini = malloc(sizeof(No));
ini->prox = NULL;
```

Busca em lista com cabeça

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

```
No *busca(No *ini, int k) {
   No *p = ini->prox  /* primeiro nó  não-cabeça */;
   while (p && p->info != k)
     p = p->prox;
  return p;
}
```

Note como funciona em todos os casos.

Busca recursiva em uma lista com cabeça

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

```
No *buscarec(No *ini, int k) {
   if (!ini->prox || ini->prox->info == k)
      return ini->prox;
   return buscarec(ini->prox, k);
}
```

Inserção em uma lista com cabeça

A função `insere` recebe um ponteiro `p` de um nó em uma lista ligada com cabeça 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(No *p, int k) {
    No *novo;
    novo = malloc(sizeof(No));
    novo->info = k;
    novo->prox = p->prox;
    p->prox = novo;
}
```

Note como funciona em todos os casos.

Inserção em uma lista com cabeça

O que está errado no código abaixo?

```
void insere(No *p, int k) {
    No novo;
    novo.info = k;
    novo.prox = p->prox;
    p->prox = &novo;
}
```

Inserção em uma lista com cabeça

O que está errado no código abaixo?

```
void insere(No *p, int k) {
    No novo;
    novo.info = k;
    novo.prox = p->prox;
    p->prox = &novo;
}
```

A variável `novo` deixa de existir quando a função `insere` termina.

Remoção em uma lista com cabeça

A função `remove` recebe um ponteiro `p` de um nó de uma lista ligada com cabeça e remove o nó que vem depois do nó apontado por `p`. Só faz sentido se `p != NULL`.

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

Note como funciona em todos os casos.

Busca seguida de remoção

Queremos implementar a seguinte função.

```
void busca_remove(No *ini, int k);
```

A função `busca_remove` recebe uma lista ligada com cabeça `ini` e um inteiro `k` e remove da lista o primeiro nó com conteúdo igual a `k`, se tal nó existir.

Busca seguida de remoção

```
void busca_remove(No *ini, int k) {
   No *p, *q;
   p = ini;
   q = ini->prox;
   while (q && q->info != k) {
      p = q;
      q = q->prox;
   }
   if (q) {
      p->prox = q->prox;
      free(q);
  }
}
```

Busca seguida de inserção

Queremos implementar a seguinte função.

```
void busca_insere(No *ini, int w, int k);
```

A função `busca_insere` recebe uma lista ligada com cabeça `ini`, dois inteiros `w` e `k` e insere um novo nó com conteúdo `k` antes do primeiro nó com conteúdo `w`. Se tal nó não existir, o novo nó é inserido no final da lista.

Busca seguida de inserção

```
void busca_insere(No *ini, int w, int k) {
   No *p, *q, *novo;
   novo = malloc(sizeof(No));
   novo->info = k;

   p = ini;
   q = ini->prox;
   while (q && q->info != w) {
      p = q;
      q = q->prox;
   }
   novo->prox = q;
   p->prox = novo;
}
```

Exercícios

- Em exercícios envolvendo listas ligadas espera-se soluções que trabalhem apenas com os ponteiros sem movimentação de conteúdo (exceto em inserção, claro). Tenha isto em mente ao resolver os exercícios.
- Escreva um programa em C para testar sua solução!
- Tente fazer uma versão iterativa e outra recursiva de cada solução.
- Se não estiver explícito no enunciado, a lista ligada pode ser sem cabeça ou com cabeça (para alguns exercícios não faz diferença).
- Fazer uma figura ajuda!

Exercícios

Exercício 1. Escreva uma função `void remove_todos(No *ini, int k)` que recebe uma lista ligada `ini` com cabeça e remove todos os nós com chave igual a `k`.

Exercício 2. Escreva uma função `No *copia(No *ini)` que recebe uma lista ligada `ini`, cria uma lista idêntica a `ini` e devolve um ponteiro para essa.

Exercício 3. Escreve uma função `No *copia_invertido(No *ini)` que recebe uma lista ligada `ini`, cria uma lista com os mesmos elementos de `ini` mas em ordem inversa e devolve um ponteiro para essa. Isto é, o conteúdo do primeiro nó da lista original é o conteúdo do último nó da nova lista, o conteúdo do segundo nó da lista original é o conteúdo do penúltimo nó da nova lista etc.

Exercícios

Exercício 4. Escreva uma função `No *minimo(No *ini)` que recebe uma lista `ini` e devolve um ponteiro para o nó com menor conteúdo.

Exercício 5. Escreva uma função `int tamanho(No *ini)` que recebe uma lista ligada `ini` e devolve o número de nós da lista. Faça duas versões: uma para lista sem cabeça e outra para lista com cabeça (**não** conte a cabeça no tamanho da lista).

Exercício 6. Escreva uma função `No *inverte(No *ini)` que recebe uma lista com cabeça e inverte a ordem dos nós (o primeiro passa a ser o último, o segundo passa a ser o penúltimo etc.). Faça isso sem criar novas células e sem usar vetores. Apenas altere os ponteiros.

Exercícios

Exercício 7. Escreva uma função `void libera(No *ini)` que recebe uma lista ligada `ini` e aplica `free` a todos os nós da lista Suponha que todos os nós foram criados com `malloc`.

Exercício 8. Escreva uma função `void conta_remove(No *ini, int k)` que recebe uma lista ligada `ini` com cabeça e um inteiro `k` e remove o k-ésimo nó da lista. Pode supor que a lista tem tamanho pelo menos `k`.

Exercício 9. Escreva uma função `void conta_insere(No *ini, int k, int x)` que recebe uma lista ligada `ini` com cabeça e inteiros `k,x` e insere o um novo nó com conteúdo `x` entre o k-ésimo nó e o $(k+1)$-ésimo nó da lista. Pode supor que a lista tem tamanho pelo menos `k`.

Ordenação de 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!

Algoritmo da seleção

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;

Algoritmo da seleção

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;

Na implementação em C, passamos o nó cabeça da lista que queremos ordenar e este deve ser o nó cabeça da lista ordenada.

Algoritmo da 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.

Algoritmo da 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);
}
```

Algoritmo da 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.

Algoritmo da 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.

Algoritmo da seleção – análise

- Podemos analisar a complexidade de `selectionsort` em função de n, o tamanho da lista `ini`.

Algoritmo da 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.

Algoritmo da 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.

Algoritmo da 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.

Algoritmo da seleção – outra forma

```
void selectionsort(No *ini) {

    faça t apontar para a lista ini;
    faça ini ser a lista vazia;
    enquanto t for não 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.

Algoritmo da inserção

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;

Algoritmo da inserção

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.

Algoritmo da 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;
    }
}
```

Algoritmo da 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.

Algoritmo da 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;
}
```

Algoritmo da 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`).

Algoritmo da inserção – análise

- Podemos analisar a complexidade de `insertionsort` em função de n, o tamanho da lista `ini`.

Algoritmo da 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.

Algoritmo da 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.

Algoritmo da 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.

Problema de intercalação – *merge*

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`.

Problema de intercalação – *merge*

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.

Problema da intercalação – *merge*

```
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 = mergsort(met);
    return intercala(ini, met);
```

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.
- Assim, o tempo gasto é

$$T(n) = n + 2T(n/2) + n = 2T(n/2) + 2n.$$

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.
- Assim, 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

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.
- Assim, o tempo gasto é

$$T(n) = n + 2T(n/2) + n = 2T(n/2) + 2n.$$

- Assim,

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

- Pode-se mostrar que a solução desta recorrência é $T(n) = cn \log n$ para alguma constante c.

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)

Veja a Figura na lousa.

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 $(k-1)n \cong n \log_2 n$.

Mergesort – análise

- Para n grande, temos que $n \log n << n^2$.
- 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).

Mergesort – análise

- Para n grande, temos que $n \log n << n^2$.
- 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?

Comparação entre vetores e listas ligadas

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

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).

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 $n = |S|$.

Comparação entre vetores e 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)

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 em uma 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.

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.

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`

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 sem sentinela

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;
    while (p && p->info != k)
      p = p->prox;
  return p;
}
```

Busca em lista circular com sentinela

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;
  if (p == ini) return NULL;
  else return p;
}
```

Economiza uma comparação em cada iteraçao do laço.

Inserção e remoção em lista ligada circular

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.

Representação de polinômios por listas circulares

Considere o polinômio:

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

com $a_n \neq 0$.

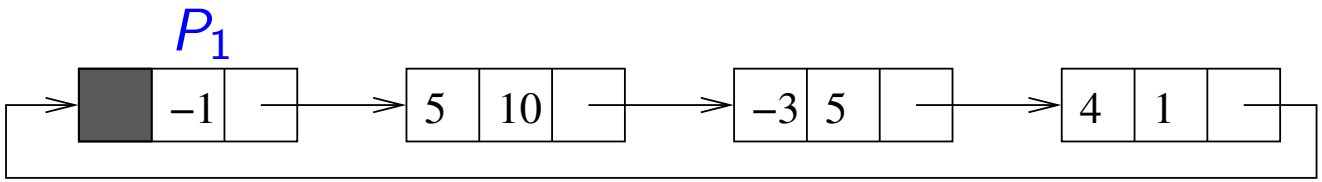
Representação de polinômios por listas circulares

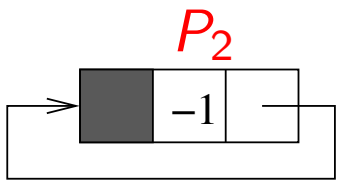
Considere o polinômio:

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

com $a_n \neq 0$.

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





Representação de polinômios por listas ligadas

```
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.

Representação de polinômios por listas ligadas

```
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.

Representação de polinômios por listas ligadas

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("Polinômio 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.

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.

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

- 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 $J(N, k)$.