Estruturas de Dados Clássicas

Prof. Wander Gaspar, D.Sc. – Engenharia Elétrica

1 de setembro de 2016

Sumário

PREFÁCIO

Notas de aula da disciplina **Estruturas de Dados Clássicas** ministrada aos alunos dos cursos de Engenharia e Sistemas de Informação do Centro de Ensino Superior de Juiz de Fora pelo professor Wander Gaspar.

Horário das aulas no segundo semestre letivo de 2016: quarta-feira às 20:50 e quinta-feira às 18:50.

A frequência será lançada semanalmente no Portal Acadêmico. A disciplina possui 4 créditos, o que equivale a 72 horas/aula. O limite máximo de faltas, correspondente a 25%, o que representa 18 horas/aula.

As avaliações estão agendadas conforme cronograma seguinte:
1a. avaliação: 29/setembro
2a. avaliação: 27/outubro
3a. avaliação: 08/dezembro

Contato com o professor pelos e-mails `wandergaspar@pucminas.cesjf.br` e `wandergaspar@gmail.com`

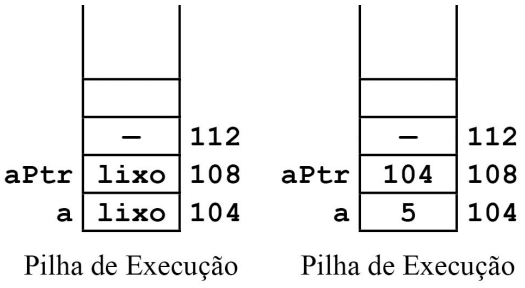
Capítulo 1

PONTEIROS

Uma função definida na linguagem C pode retornar um único valor através do comando `return`. Isso não é satisfatório em muitos casos, uma vez que pode tornar-se necessário retornar dois ou mais valores. Uma alternativa para o problema consiste no emprego de variáveis do tipo ponteiro.

Para cada tipo de dado primitivo (`int`, `double`, `char` etc.), a linguagem C permite definir uma variável do tipo ponteiro, capaz de armazenar um endereço de memória correspondente ao tipo declarado na definição. Por exemplo, `int *p` declara um ponteiro para um inteiro.

```
int main(void) {
    int a;
    int *aPtr; /* variável ponteiro para inteiro */
    a = 5;
    aPtr = &a; /* aPtr recebe o endereço de a */
    ...
```



Atenção: um ponteiro armazena endereços de memória de variáveis, portanto, somente pode receber endereços de memória em comandos de atribuição.

É possível, porém, fazer uma atribuição de valor para a variável de memória referenciada por um ponteiro. Por exemplo, o comando `*aPtr = 10` atribui o valor 10 ao endereço de memória apontado por `aPtr`.

Capítulo 2

LISTAS ENCADEADAS

Como já visto anteriormente, um vetor é a forma mais simples de representar um conjunto de elementos homogêneos.

```
#define MAX 10
int vet[MAX]
```

Um vetor ocupa um espaço contíguo de memória, permitindo que qualquer elemento seja acessado indexando-se o ponteiro para o primeiro elemento.

Vantagens do uso de vetores: possibilita o acesso direto a qualquer elemento, através do índice; facilidade de implementação.

Desvantagem: é necessário dimensionar o tamanho máximo de elementos.

Alternativa algorítmica: utilizar estruturas de dados que aumentem ou diminuam de tamanho conforme a necessidade de mais ou menos elementos armazenados. Essas estruturas de dados são chamadas dinâmicas e armazenam cada um dos elementos por alocação dinâmica de memória.

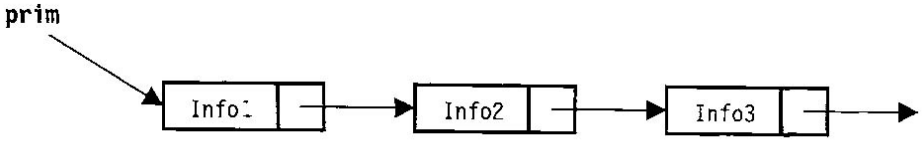


Figura 10.2 *Arranjo da memória de uma lista encadeada.*

2.1 Estrutura e Funções Básicas

Cada nó da lista contém uma informação e o ponteiro para o próximo elemento da lista. Assim, a lista é representada por um ponteiro para o primeiro elemento (nó). Do primeiro nó, alcança-se o segundo, seguindo o encadeamento, e assim por diante. O último elemento armazena, no campo para o próximo nó, o valor `NULL`, indicando assim que não há mais elementos na lista.

```
typedef struct lista {
    int info;
    struct lista* prox;
} Lista;
```

Uma função para **criação** de uma lista encadeada implementa uma lista vazia, sem nós. Assim, deve retornar um ponteiro para `NULL`.

```
/* Função de criação: cria uma lista vazia */
Lista* lst_cria() {
    return NULL;
}
```

Uma vez criada a lista vazia, é possível inserir elementos. Para cada novo nó, deve-se alocar dinamicamente a memória necessária para armazená-lo. A função de **inserção** insere o novo nó sempre no início da lista.

```
/* Função de inserção no início da lista */
Lista* lst_insere(Lista* lst, int i) {
    Lista* novo = (Lista*)malloc(sizeof(Lista));
    novo->info = i;
    novo->prox = lst;
    return novo;
}
```

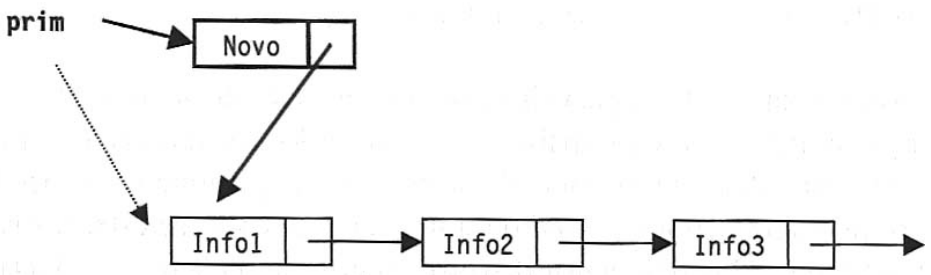


Figura 10.3 *Inserção de um novo elemento no início da lista.*

2.1.1 Percorrimento de uma Lista Encadeada

Para percorrer uma lista encadeada, é necessário uma variável auxiliar do tipo ponteiro, que irá receber o endereço de cada um dos nós da lista.

```
/* Função de impressão dos elementos da lista */
void lst_exibe(Lista* lst) {
    Lista* p; /* Variável auxiliar */
    for(p = lst; p != NULL; p = p->prox)
        printf("Info = %d\n",p->info);
}
```

Exemplo 2.1. Escrever um programa em C para criar uma lista encadeada de inteiros e incluir dois nós (elementos) quaisquer.

```
int main() {
    Lista* lst; /* Declara uma lista */
    lst = lst_cria(); /* Inicializa uma lista vazia */
    lst = lst_insere(lst,23); /* Insere 23 */
    lst = lst_insere(lst,45); /* Insere 45 */
    lst_exibe(lst); /* exibe lista na tela */
    return 0;
}
```

Exercício 2.1. Escrever um programa C que gere uma lista encadeada com 10 números inteiros aleatórios no intervalo [0, 99]. A cada novo elemento inserido, exiba o conteúdo completo da lista.

Exercício 2.2. Escrever um programa C que gere uma lista encadeada com 10 números reais aleatórios no intervalo [0.0, 9.9]. Ao fim do processo de inclusão, exibir o conteúdo da lista.

Exercício 2.3. Escrever um programa C que gere uma lista encadeada com n elementos, onde n é fornecido pelo usuário. Cada elemento deve conter três campos, além do ponteiro para o próximo elemento da lista: (a) um número inteiro aleatório no intervalo [0, 99], (b) um número real no intervalo [0.0, 9.9], (c) um caractere no intervalo [A, Z]. Ao fim do processo de inclusão, exibir o conteúdo completo da lista.

2.1.2 Pesquisa em uma Lista Encadeada

Pode ser útil implementar uma função para verificar se uma lista está **vazia**. A função recebe a lista e retorna 1 se estiver vazia e 0 se houver pelo menos um nó na lista.

```
/* Função vazia: retorna 1 se vazia e 0 se não vazia */
int lst_vazia(Lista* lst) {
    if(lst==NULL)
        return 1;
    else
        return 0;
}
```

Outra função útil consiste em verificar se um determinado elemento está contido na lista. A função recebe o valor pesquisado e retorna um ponteiro para o nó da lista que contém o elemento. Caso não seja encontrado, a função retorna `NULL`.

```
/* Função busca: verifica se um elemento está contido na lista */
Lista* lst_busca(Lista* lst, int v) {
    Lista* p; /* Variável auxiliar */
    for(p=lst; p!=NULL; p=p->prox)
        if(p->info==v)
            return p;
    return NULL; /* não achou o elemento */
}
```

Exemplo 2.2. escrever um programa C para implementar uma lista linear dinâmica contendo valores inteiros. Inserir 3 elementos. Verificar se os valores foram corretamente incluídos. Apresentar na tela o conteúdo da lista.

```
int main(void) {
    Lista* lst; /* Declara uma lista */
    lst = lst_cria();
    lst = lst_insere(lst,23);
    lst = lst_insere(lst,45);
    lst = lst_insere(lst,56);
    if(!lst_vazia(lst)) {
        lst_imprime(lst);
        Lista* p = lst_busca(lst,12);
        if(p==NULL)
            printf("12 nao esta contido na lista\n");
        else
            printf("12 esta contido na lista\n");
        p = lst_busca(lst,45);
        if(p==NULL)
            printf("45 nao esta contido na lista\n");
        else
            printf("45 esta contido na lista\n");
   } else
        printf("Lista vazia");
   return 0;
}
```

Exercício 2.4. Escrever um programa C que gere uma lista encadeada com 10 números inteiros aleatórios no intervalo [0, 99]. Exibir na saída padrão todos os elementos maiores que n, onde $0 < n < 99$ é fornecido pelo usuário em tempo de execução.

Exercício 2.5. Escrever um programa C que gere uma lista encadeada com 10 números inteiros aleatórios no intervalo [0, 99]. Exibir na saída padrão (a) todos os elementos ímpares, (b) todos os elementos pares.

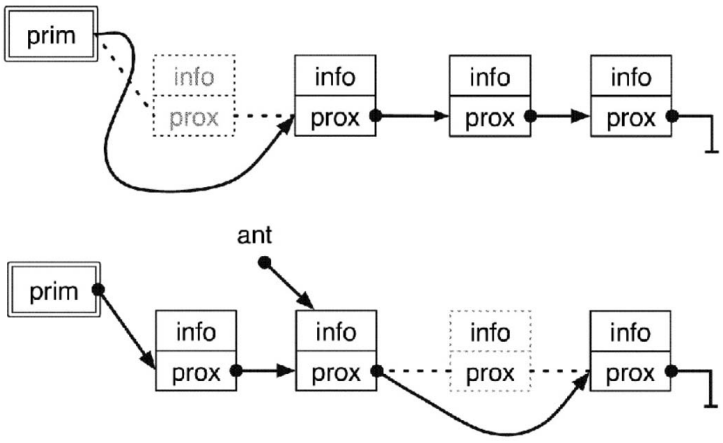
Exercício 2.6. Escrever um programa C que gere e exiba na saída padrão uma lista encadeada com 10 números inteiros aleatórios no intervalo [0, 99]. A partir da lista gerada, criar duas novas listas, uma contendo apenas os números pares e outra contendo somente os números ímpares. Exibir as duas novas listas na saída padrão.

Exercício 2.7. Escrever um programa C que gere e exiba na saída padrão duas listas encadeadas A e B contendo cada uma 10 números inteiros aleatórios no intervalo [0, 99]. O programa deve gerar e exibir na saída padrão uma nova lista encadeada que contenha todos os elementos de A e de B.

2.1.3 Exclusão da Lista Encadeada

A função para exclusão de elementos de uma lista encadeada implementada dinamicamente tem dois argumentos: a lista e o elemento de se deseja retirar.

Há duas possibilidades de retirada: se o elemento a ser excluído for o primeiro, deve-se fazer o novo valor da lista apontar para o segundo elemento. Se o elemento não for o primeiro da lista, deve-se fazer o elemento anterior apontar para o seguinte.




```
/* Função retira: retira um elemento da lista */
Lista* lst_retira(Lista* lst, int v) {
    Lista* ant = NULL; /* ponteiro para elemento anterior */
    Lista* p = lst; /* ponteiro para percorrer a lista */
    /* Procura elemento na lista, guardando anterior */
    while(p!=NULL && p->info!=v) {
        ant = p;
        p = p->prox;
    }
    /* Verifica se encontrou o elemento */
    if(p==NULL)
        return lst; /* Retorna lista original (não achou) */
    /* Retira elemento */
    if(ant==NULL) {
        /* O elemento é o primeiro da lista */
        lst = p->prox;
    } else {
```

```
        /* O elemento não é o primeiro da lista */
        ant->prox = p->prox;
    }
    free(p);
    return lst;
}
```

Exemplo 2.3. Escrever um programa C para implementar uma lista linear dinâmica contendo valores inteiros. Inserir 3 elementos. Apresentar na tela o conteúdo da lista. Retirar um dos elementos da lista. Apresentar novamente na tela o conteúdo da lista. Desfazer a lista encadeada.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    Lista* lst; /* Declara uma lista */
    lst = lst_cria();
    lst = lst_insere(lst,23);
    lst = lst_insere(lst,36);
    lst = lst_insere(lst,45);
    printf("Lista:\n");
    lst_exibe(lst);
    lst = lst_retira(lst,36);
    printf("\nLista:\n");
    lst_exibe(lst);
    return 0;
}
```

Exercício 2.8. Escrever um programa C que gere e exiba na saída padrão uma lista encadeada com 10 números inteiros aleatórios no intervalo [0, 99]. O programa deve excluir diversos elementos da lista, cujos valores são fornecidos pelo usuário em tempo de execução. O flag é qualquer valor fora do intervalo [0, 99]. Exibir a lista na saída padrão após cada exclusão de um elemento.

Exercício 2.9. Escrever um programa C que gere e exiba na saída padrão uma lista encadeada com 20 números inteiros aleatórios no intervalo [0, 9]. O programa deve excluir todas as ocorrências de um determinado valor fornecido pelo usuário em tempo de execução. Exibir a lista na saída padrão após a exclusão.

2.1.4 Função que Desfaz a Lista

Completando o conjunto de funções básicas para manipulação de listas, consideremos uma função que destrói a lista encadeada, liberando o espaço de todos os elementos alocados em memória.

```
/* Função libera: libera os elementos da lista */
void lst_libera(Lista* lst) {
    Lista* p = lst; /* ponteiro para percorrer a lista */
    while(p!=NULL) {
        /* Ponteiro auxiliar aponta para o próximo nó */
        Lista* q = p->prox;
        /* Libera o espaço de p */
        free(p);
        /* Faz p apontar para o próximo nó */
        p = q;
    }
}
```

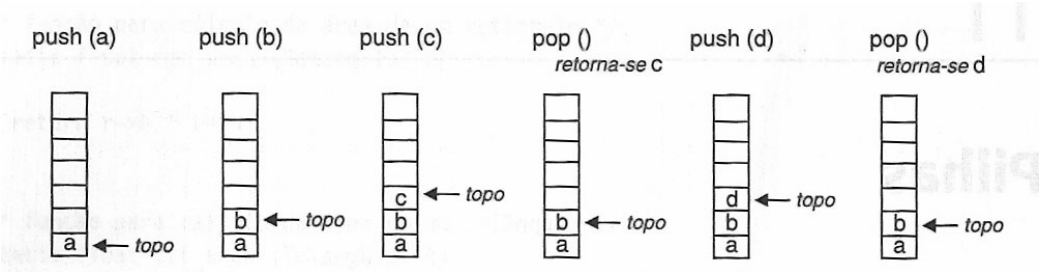
Exercício 2.10. Escrever um programa C que gere e exiba na saída padrão uma lista encadeada com 10 números inteiros aleatórios no intervalo [0, 99]. O programa deve excluir, um a um, todos os elementos da lista (exceto o primeiro), sempre a partir do último elemento. A cada novo elemento retirado, exibir o conteúdo da lista na saída padrão.

Capítulo 3

PILHAS

A ideia fundamental da estrutura de dados do tipo **pilha** consiste em realizar todos os acessos aos elementos a partir do topo. Assim, quando um elemento novo é introduzido na pilha, passa a ser o elemento do topo. O único elemento que pode ser removido da pilha é o elemento do topo. Portanto, o primeiro que sai é o último que entrou (a sigla LIFO, do inglês *last in, first out* é usada para descrever essa estratégia).

Existem duas operações básicas com pilhas: empilhar um novo elemento (*push*) e desempilhar o elemento do topo (*pop*), conforme ilustrado a seguir.



3.1 Implementação de Pilha com Vetor

Essa abordagem, mais simples, pode ser usada nos casos em que se conhece o número máximo de elementos da estrutura de dados pilha.

A estrutura que representa o tipo pilha deve conter um vetor e o número de elementos armazenados. Se existem n elementos armazenados na pilha, então o índice $(n-1)$ do vetor representa o topo.

```
typedef struct {
    int n; /* número de elementos na pilha */
    int vet[10]; /* vetor de 10 elementos */
} Pilha;
```

3.1.1 Funções Básicas de Pilha com Vetor

A função para criar a pilha aloca espaço em memória dinamicamente para a estrutura e inicializa uma pilha vazia.

```
/* Função para criar a pilha */
Pilha* pilha_cria(void) {
    /* retorna um ponteiro para a pilha */
    Pilha* p = (Pilha*)malloc(sizeof(Pilha));
    p->n = 0; /* inicializa com 0 elementos */
    return p;
}
```

Para inserir um elemento na pilha, deve ser usada a próxima posição livre do vetor. Considerando que o vetor tem dimensão fixa, torna-se necessário verificar se há espaço disponível para a inserção de um novo elemento na pilha.

```
/* Função para inserir um novo elemento no topo da pilha */
void pilha_push(Pilha* p, int v) {
    if(p->n == 10) /* não há espaço na pilha */
        printf("Capacidade da pilha estourou.\n");
        /* insere elemento na próxima posição livre */
    else {
        p->vet[p->n] = v; /* aloca elemento no vetor */
        p->n++;
    }
}
```

A função `pop` retira o elemento do topo da pilha e retorna seu valor. Deve haver também uma função para verificar se há elementos na pilha.

```
/* Função para retirar o elemento do topo da pilha */
int pilha_pop(Pilha* p) {
    int v;
    if(pilha_vazia(p)) /* a pilha está vazia */
        printf("Pilha vazia.\n");
    /* retira elemento do topo */
    else {
        v = p->vet[p->n-1]; /* o n-ésimo elemento na pos. n-1*/
        p->n--; /* subtrai 1 do número de elementos da pilha */
    }
    return v;
}

/* Função para verificar se a pilha está vazia */
int pilha_vazia(Pilha* p) {
    return (p->n==0); /* n é igual a zero? */
}
```

3.1.2 Mais Funções de Pilha com Vetor

A rigor, pela definição de pilha, só se deve ter acesso ao elemento do topo. Porém, é útil implementar uma função que exiba na saída padrão os valores armazenados na pilha. A função a seguir apresenta o conteúdo da pilha, na ordem do topo para a base.

```
/* Função para exibir o conteúdo da pilha */
void pilha_exibe(Pilha* p) {
    int i;
    for(i=p->n-1;i>=0;i--)
        printf("%d",p->vet[i]);
}
```

Por fim, deve haver uma função para liberar a memória alocada para a pilha implementada por vetor.

```
/* Função para liberar a memória alocada pela pilha */
void pilha_libera(Pilha* p){
    free(p);
}
```

Exemplo 3.1. Programa C para implementar uma pilha de inteiros usando vetor.

```
int main(void) {
    Pilha* p = pilha_cria();
    if(pilha_vazia)
        printf("Pilha vazia\n");
    pilha_push(p,23);
    pilha_push(p,78);
    pilha_push(p,45);
    printf("Pilha com 3 elementos\n");
    pilha_exibe(p);
    pilha_pop(p);
    printf("\nPilha com 2 elementos\n");
    pilha_exibe(p);
    pilha_pop(p);
    pilha_pop(p);
    if(pilha_vazia)
        printf("\nPilha vazia\n");
    pilha_libera(p);
}
```

Exemplo 3.2. Escrever um programa para implementar uma pilha de inteiros de tamanho 5 que deve conter números aleatórios no intervalo [0, 99]. O empilhamento e desempilhamento de elementos deve ser executado 100 vezes em ordem aleatória. A cada iteração, exibir a ação executada e o conteúdo da pilha.

Exemplo 3.3. Escrever um programa para implementar uma pilha de caracteres de tamanho 20 que deve conter números aleatórios no intervalo [0, 99]. O empilhamento e desempilhamento de elementos deve ser executado em ordem aleatória até que a pilha fique cheia. A cada iteração, exibir a quantidade de elementos na pilha. Ao final, exibir o conteúdo da pilha.

Exemplo 3.4. Escrever um programa para implementar uma pilha de caracteres de tamanho TAM, fornecido pelo usuário em tempo de execução

que deve conter letras aleatórias maiúsculas. O empilhamento e desempilhamento de elementos deve ser executado em ordem aleatória até que a pilha fique cheia. A cada iteração, exibir a quantidade de elementos na pilha. Ao final, exibir o conteúdo da pilha.