Estruturas de dados elementares: parte 2

Prof. Bruno de Castro Honorato Silva

February 2, 2021

1 Introdução

Neste documento, são descritas estruturas de dados que expandem o projeto de implementação das estruturas de dados elementares apresentadas no documento anterior, intitulado como *Estruturas de dados elementares*. Portante, nas seções que se seguem estudaremos:

- Lista estática ordenada;
- Fila circular;
- Lista duplamente encadeada.

2 Lista estática ordenada

Seja $L = \{e_1, e_2, ..., e_k, e_{k+1}..., e_n\}$ uma lista estática. L podem ser mantida em ordem crescente/decrescente segundo o valor de seus elementos e. Essa ordem facilita a pesquisa de itens. Por outro lado, a inserção e remoção são mais complexas pois deve manter os itens ordenados. O TAD *ListaEstáticaOrdenada* é o mesmo do TAD *ListaEstática*, apenas difere na implementação. As operações diferentes serão:

- Inserção: insere um elemento e em uma posição tal que L é mantida ordenada. Para que um elemento seja inserido em L, a mesma não pode estar cheia;
- Remoção: o elemento e com a chave fornecida é removido de L. L é mantida ordenada.

A Figura 1 ilustra a operação de inserção em uma lista ordenada.

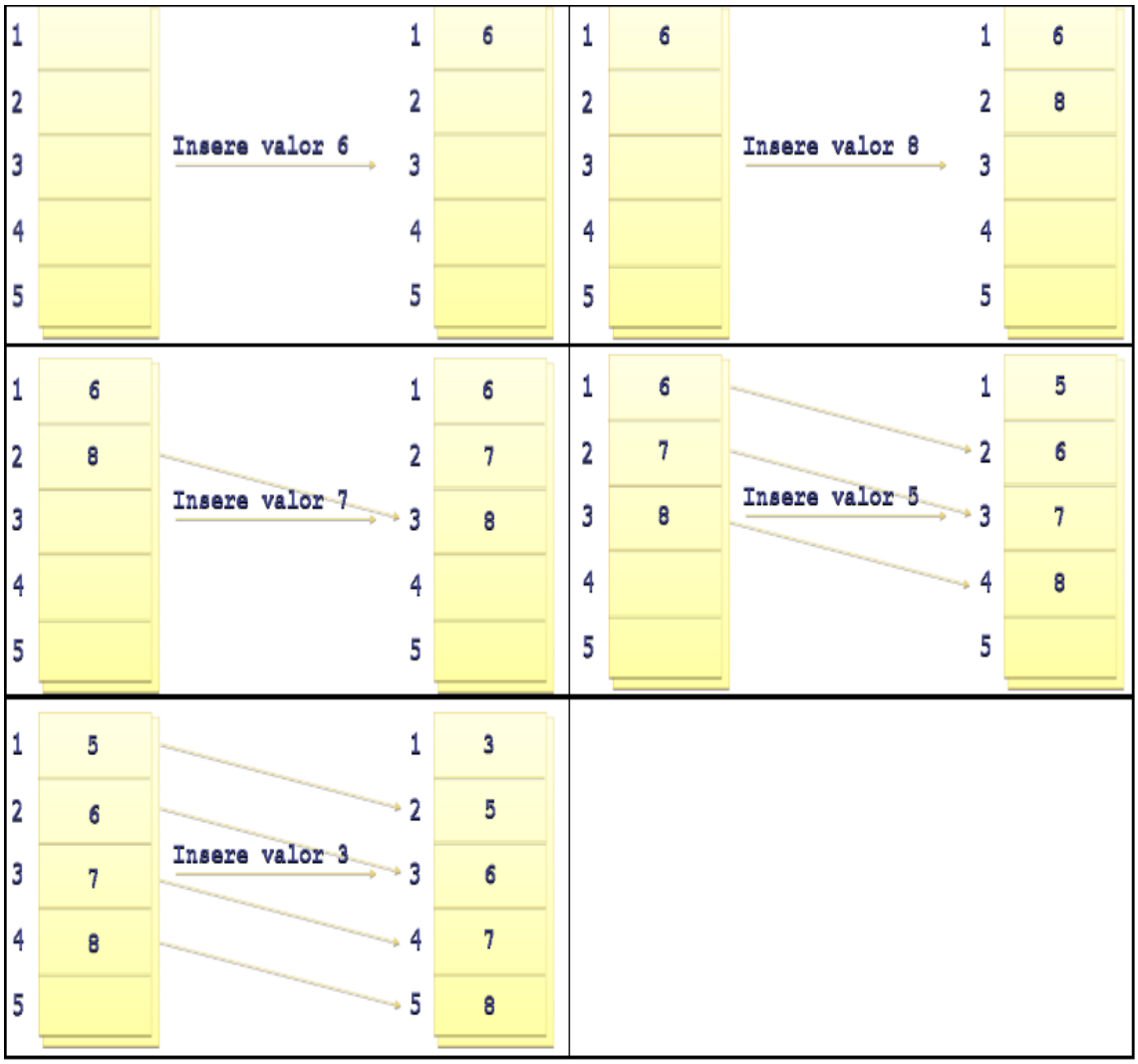

Figure 1: Ilustração de inserção em lista ordenada.

Uma tarefa comum a ser executada sobre listas é a busca de elementos dado um valor (também conhecido como chave de busca), denotado por k. No caso da lista não ordenada, a operação de busca é sequencial (ou linear), isto é, o procedimento de busca percorrerá o vetor de dados, um elemento por vez, até encontrar o k. Se o projeto de implementação da lista segue essa lógica, a de busca sequencial, então a complexidade temporal desta operação é $O(n)$.

Em outras palavras, a ideia da operação de busca sequencial é procurar um elemento que tenha uma determinada chave, começando do início da lista, e parar quando a lista terminar ou quando o elemento for encontrado. A Figura 2 ilustra o código fonte desta operação.

```
ITEM *busca_sequencial(LISTA_ESTATICA *lista, int chave) {
  int i;

  for (i = 0; i <= lista->fim; i++) {
    if (lista->vetor[i]->chave == chave) {
      return lista->vetor[i];
    }
  }

  return NULL;
}
```

Figure 2: Código fonte do procedimento de busca linear.

Este tempo de execução pode ser é otimizado caso a lista seja tenha seu vetor de dados ordenado. Seja $L^o = \{3, 5, 6, 7, 8\}$ uma lista ordenada e a chave de busca $k = 4$, a operação de busca sequencial poderia cessar tão logo a iteração chegasse no elemento $L^o[1] = 5$, pois já que $k < L^o[1]$, e todos elementos posteriores a $L^o[1]$ são superiores a ele, não faria sentido continuar a busca. A Figura 3 ilustra o código fonte da operação de busca sequencial otimizada.

```
ITEM *busca_sequencial(LISTA_ESTATICA *lista, int chave) {
  int i;

  for (i = 0; i <= lista->fim; i++) {
    if (lista->vetor[i]->chave == chave) {
      return lista->vetor[i];
    } else if (lista->vetor[i]->chave > chave) {
      return NULL;
    }
  }

  return NULL;
}
```

Figure 3: Código fonte do procedimento de busca linear.

Entretanto, essa melhoria não altera a complexidade da busca sequencial, que ainda é *O(n)*. Para entender esta afirmação, voltemos ao exemplo anterior. Considere $L^o = \{3, 5, 6, 7, 8\}$ uma lista ordenada, $n = 5$ a dimensaão do vetor de dados, e a chave de busca $k = 8$. Observamos que 8 é o último elemento do vetor. Neste contexto, o algoritmo executaria n, até constatar que 8 está no vetor. O mesmo aconteceria caso $k = 10$, pois, neste caso, k é maior que todos os elementos do vetor e o procedimento mais uma vez executaria n checagens para constatar que $k = 10$ não estaria no vetor de dados.

Visto que continuar com a busca sequencial em lista ordenada não provê ganho performance para a operação busca, recorremos a um novo algoritmo de busca: busca binária. A busca binária é um algoritmo de busca mais sofisticado e bem mais eficiente que a busca sequencial. Entretanto, a busca binária somente pode ser aplicada em estruturas que permitem acessar cada elemento em tempo

constante, tais como os vetores. A ideia deste algoritmo é, a cada iteração, dividir o vetor ao meio e descartar metade do vetor.

Seja $L^o = \{3, 4, 6, 8, 11, 15, 18, 25, 30, 32\}$ e $k = 30$, a Figura 4 ilustra o funcionamento da busca binária. Já a Figura 5 apresenta o código fonte deste procedimento de *divisão e conquista*.

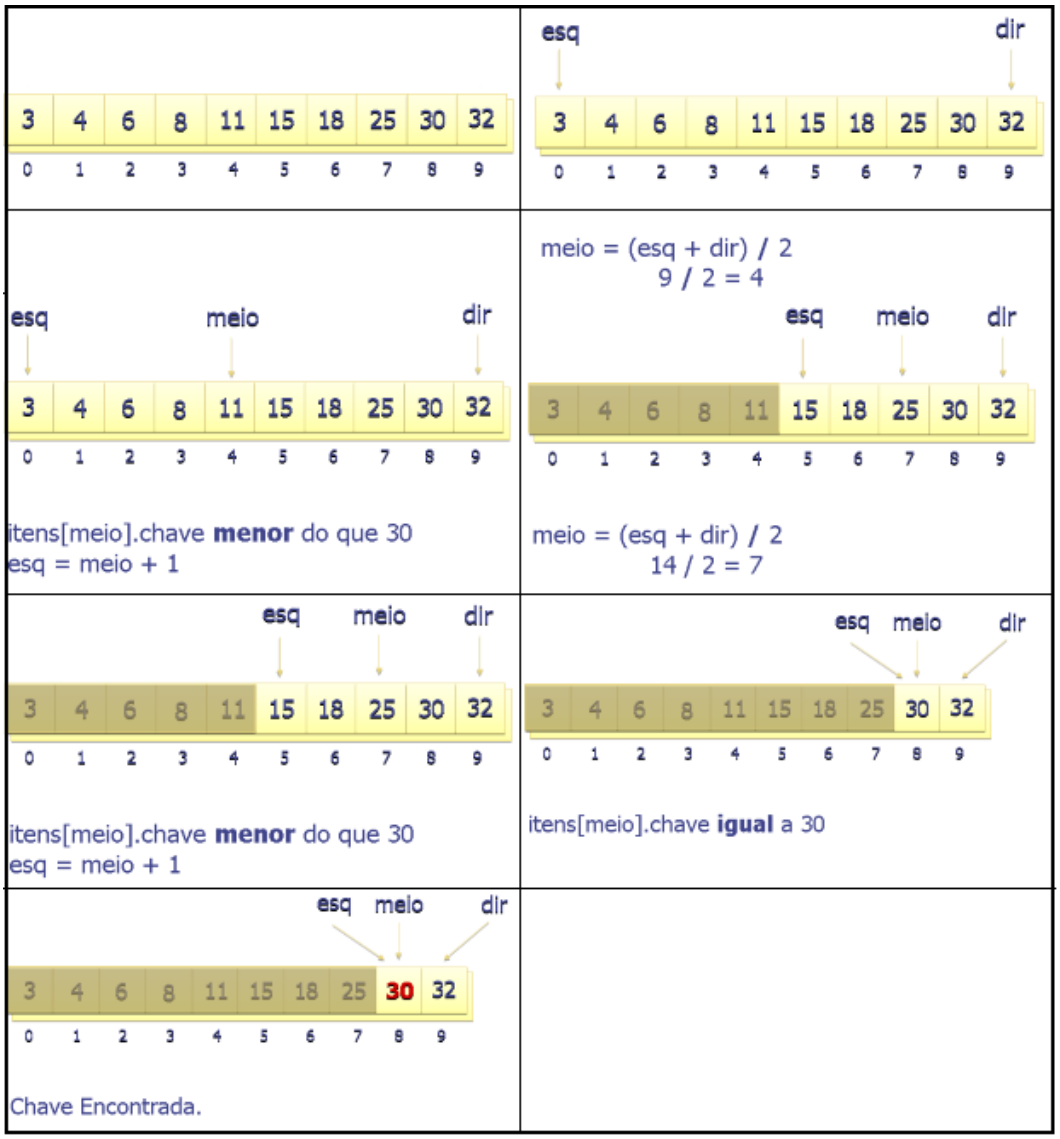

Figure 4: Ilustração de funcionamento do procedimento de busca binária.

```
ITEM *busca_binaria(LISTA_ESTATICA_ORDENADA *lista, int chave) {
  int esq = 0;
  int dir = lista->fim;

  while (esq <= dir) {
    int meio = (esq + dir) / 2;

    if (lista->vetor[meio]->chave == chave) {
      return lista->vetor[meio];
    } else if (lista->vetor[meio]->chave > chave) {
      dir = meio - 1;
    } else {
      esq = meio + 1;
    }
  }

  return NULL;
}
```

Figure 5: Ilustração do código fonte do procedimento de busca binária.

É importante lembrar que Busca binária somente funciona em vetores ordenados. Busca sequencial funciona com vetores ordenados ou não. Quanto a eficiência, a busca binária é $O(\log_2 n)$, ou simplesmente $O(\log n)$, pois em computação, a praxe é omitir a base 2 quando em uma notação log dada a recorrência com que se refere log na base 2. Para $n = 1.000.000$, aproximadamente 20 checagens seriam necessárias para verificar se uma chave de busca está ou não no vetor de dados.

3 Fila circular

Vimos que a operação de remoção em uma estrutura de dados do tipo fila requer remanejar todos os elementos do vetor de dados. Esta operação resulta em um tempo computacional $O(n)$. É possível melhorar o tempo computacional desta operação fazendo com que o início não seja fixo na primeira posição do vetor. Desta forma, deve-se manter dois contadores: um para a inicio, denominado i; e, outro contador para o final da fila, denominado f. Quando um elemento é inserido numa fila, este deve ir para a posição f, pois, todo elemento é inserido no final da fila. Já durante a operação remoção, remove-se o elemento que está na posição i, pois este tipo de estrutura dados é baseada na política *First In, First Out* (*Primeiro a Entrar, Primeiro a Sair*). Considere que N é o limite do vetor de dados. Quando $f = N$ e um elemento for inserido, pode-se permitir que f volte para o início do vetor quando esse contador contabilizar N. Essa implementação é conhecida como fila circular. A Figura 6 ilustra o vetor de dados de uma fila circular com 4 posições vagas e $f < i$. A Figura 7 ilustra a implementação das funções que se distinguem do projeto de implementação da estrutura fila tradicional.

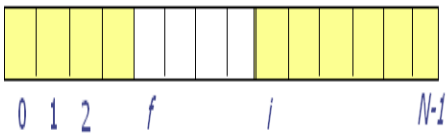

Figure 6: Ilustração do vetor de dados de uma fila circular.

```
int cheia(FILA_ESTATICA *fila) {
  return ((fila->fim + 1) % TAM == fila->inicio);
}

int enfileirar(FILA_ESTATICA *fila, ITEM *item) {
  if (!cheia(fila)) {
    fila->vetor[fila->fim] = item;
    fila->fim = (fila->fim + 1) % TAM;
  }
  return 0;
}

ITEM *desenfileirar(FILA_ESTATICA *fila) {
  if (!vazia(fila)) {
    ITEM *ret = fila->vetor[fila->inicio];
    fila->inicio = (fila->inicio + 1) % TAM;
    return ret;
  }
  return NULL;
}
```

Figure 7: Ilustração do fonte das operações melhoradas da fila circular.

4 Lista duplamente encadeada

Na nota aula anterior, foi apresentado o conceito de lista dinâmica encadeada (ou simplesmente encadeada). O projeto de implementação desta estrutura de dados de uma lista dinâmica encadeada é baseado na implementação de um componente conhecido por 'nó'. O termo 'lista dinâmica' deve-se justamente ao uso da estrutura de nós para armazenar os valores da lista, pois a lista aumenta em tempo de execução conforme novos nós com valores são inseridos na lista. O termo 'lista estática' deve-se justamente ao uso de um vetor de dados com tamanho pré-definido (ou estático) para armazenar os valores da respectiva estrutura de dados lista.

Um nó é um objeto que guarda um valor. Este valor pode ser de um tipo primitivo (int, char, float, etc) ou a referência para um outro objeto especificado no contexto da aplicação. Um nó possuí também outro atributo que permite apontar para um nó posterior, i. e., este atributo é responsável por abstrair a

lógica de ligação entre os nós que venham, por exemplo, a compor uma lista. Vale lembrar que em C, atributos podem ser compreendidos como os campos de uma *struct*.

A operação de remoção de uma lista duplamente encadeada é menos complexo de se implementar do que a de uma lista encadeada. Em uma lista duplamente encadeada, dado um nó, podemos acessar ambos os nós adjacentes: o próximo; e, o anterior. Se tivermos um ponteiro para o último nó da lista, podemos percorrer a lista em ordem inversa, bastando acessar continuamente o nó anterior, até alcançar o primeiro nó da lista, que não tem nó anterior (o ponteiro do nó anterior vale NULL). A Figura 8 abaixo mostra a estrutura dos nós de uma lista duplamente encadeada.

Figure 8: Ilustração dos nós de uma lista duplamente encadeada.

Para exemplificar a implementação em C de listas duplamente encadeadas, vamos novamente considerar um exemplo simples no qual queremos armazenar valores inteiros na lista. O nó da lista pode ser representado computacionalmente pela estrutura ilustrada na Figura 9.

```
struct no{
    int valor;
    struct no *anterior;
    struct no *proximo;
} ;

typedef struct no No;

typedef struct {
    No *inicio;
    No *fim;
    int tamanho;
} ListaDuplamenteEncadeada;
```

Figure 9: Ilustração da implementação das estruturas que representam uma lista duplamente encadeada.

As estruturas acima apresentam a implementação em C de um nó e de uma lista duplamente encadeada. Observe que a *struct* de nó possui os dois ponteiros para os nós adjacentes. Esta é a diferença entre a *struct* de uma lista simplesmente encadeada e a de uma lista duplamente encadeada.

Seja $L^{de} = \{l_1, l_2, ..., l_i, ..., l_n\}$ uma lista duplamente encadeada circular de números inteiros, i a posição do nó na sequência encadeada e n a posição do último nó da lista. Cada nó l_i armazena: um número inteiro; uma referência

para o nó anterior l_{i-1}; e, uma referência para o nó posterior l_{i+1}. Se implementada em C, a operação de inserção no fim da lista consiste em receber um objeto *lista* e um valor de entrada w, criar um nó l_i para armazenar w e ajustar os ponteiros de referências (ligações entre os nós) da lista conforme. A lógica de implementação desta operação é ilustrada na Figura 10.

```c
int inserir(ListaDuplamenteEncadeada *lista, int valor) {

    NO *novo_no = (NO *)malloc(sizeof(NO));

    if(novo_no != NULL) {
            novo_no->valor = valor;
            novo_no->proximo = NULL;
            novo_no->anterior = lista->fim;

            if(lista->inicio == NULL) {
               lista->inicio = novo_no;
            } else {
               lista->fim->proximo = novo_no;
            }

            lista->fim = novo_no;
            lista->tamanho++;
            return 1;
    }

     return 0;
}
```

Figure 10: Ilustração da implementação da operação de inserção em uma lista duplamente encadeada.

A operação de remoção consiste em receber um valor *chave de busca* k, no caso, um número inteiro, buscar por um nó l_i com valor igual a k, em encontrando um nó com valor k, atualizar as referências dos nós. A lógica de implementação desta operação é ilustrada na Figura 11.

```c
int remove(ListaDuplamenteEncadeada *lista, int chave) {
 if (!vazia(lista)) {

    NO *no_alvo_remocao = lista->inicio;

    while(no_alvo_remocao != NULL && no_alvo_remocao->valor != chave) {
       no_alvo_remocao = no_alvo_remocao->proximo;
    }

    if(no_alvo_remocao != NULL) {
        if(no_alvo_remocao != lista->inicio) {
           no_alvo_remocao->anterior->proximo = no_alvo_remocao->proximo;
        } else {
           lista->inicio = no_alvo_remocao->proximo;
        }

        if(no_alvo_remocao != lista->fim) {
           no_alvo_remocao->proximo->anterior = no_alvo_remocao->anterior;
        } else {
           lista->fim = no_alvo_remocao->anterior;
        }

        lista->tamanho--;
        apagar_no(no_alvo_remocao);
        return 1;
    }
 }
 return 0;
}
```

Figure 11: Ilustração da implementação da operação de remoção em uma lista duplamente encadeada.

4.1 Lista circular

Uma lista circular também pode ser construída com encadeamento duplo. Neste caso, o que seria o último nó da lista passa ter como próximo o primeiro nó, que, por sua vez, passa a ter o último como anterior. Com essa construção podemos percorrer a lista nos dois sentidos, a partir de um ponteiro para um nó qualquer. A Figura 12 ilustra graficamente os nós de uma lista duplamente encadeada circular.

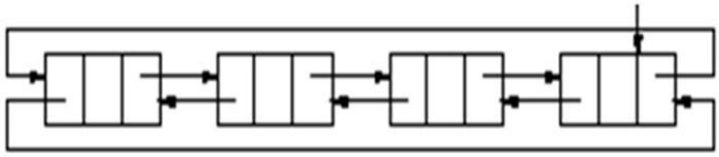

Figure 12: Ilustração dos nós de uma lista duplamente encadeada.

Numa lista circular, o último nó tem como próximo o primeiro nó da lista, formando um ciclo. Uma forma de percorrer os nós de uma lista circular é visitando todos os nós a partir do ponteiro do nó inicial até que se alcance novamente esse mesmo nó. O código exposto na Figura 13 exemplifica essa forma de percorrer os nós. Neste caso, para simplificar, consideramos uma lista que armazena valores inteiros. Devemos salientar que o caso em que a lista é vazia ainda deve ser tratado (se a lista é vazia, o ponteiro para um nó inicial vale NULL).

```
void printAll (ListaCircular* l) {
    No* p = l->inicio;
    /* faz p apontar para o nó inicial */
    /* testa se lista não é vazia */
    if (p != NULL)  {
    /* percorre os elementos até alcançar novamente o início */
        do {
            printf("%d\n", p->valor);     /* imprime informação do nó */
            p = p->prox; /* avança para o próximo nó */
        } while (p != l->inicio);
    }
}
```

Figure 13: Ilustração da implementação da operação de impressão de todos os nós de uma lista circular.