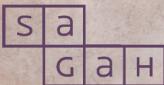


ESTRUTURA DE DADOS

Rafael Albuquerque



SOLUÇÕES
EDUCACIONAIS
INTEGRADAS

Implementação de listas encadeadas duplas

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Demonstrar a utilização de listas encadeadas duplas em problemas computacionais.
- Analisar a utilização de listas encadeadas duplas em problemas computacionais.
- Resolver problemas computacionais utilizando listas encadeadas duplas.

Introdução

As listas duplamente encadeadas são estruturas de dados sequenciais, nas quais um registro de memória se liga a dois registros de espaço de memória — também conhecidos como nós — tanto para um endereço anterior como para um endereço posterior. Essa característica permite que os elementos sejam percorridos em duas direções, partindo do início ou do final da lista.

Em problemas computacionais, a aplicação de listas duplamente encadeadas é utilizada em vetores fixos ou dinâmicos, nos quais uma coluna indica qual é o dado anterior ou posterior. Essa abordagem em listas se torna eficiente, pois, se comparada às listas de encadeamento simples, o último elemento de uma lista pode ser acessado de forma direta, sem que seja preciso percorrer todos os elementos anteriores. Assim como o acesso, a remoção de qualquer nó da lista também é facilitada por não precisar percorrer novamente a lista até o nó que se deseja eliminar para guardar o elemento anterior.

Neste capítulo, você irá aprender a identificar a utilização de listas encadeadas duplas em problemas computacionais, como realizar a análise de utilização e como empregá-la para solucionar problemas computacionais.

1 Listas encadeadas duplas em problemas computacionais

Os problemas computacionais são situações que podem ser resolvidas passo a passo com o computador. Por essa natureza, esses problemas geralmente são bem definidos, com entradas bem estabelecidas, restrições e condições que satisfazem os valores de saída. Alguns dos problemas podem ser de: tomada de decisão, pesquisa, contagem, ordenação e otimização.

Conforme Piva Junior *et al.* (2014), além dos diversos tipos de dados que podem ser armazenados, podem ser escolhidos também diferentes tipos de lista, segundo a necessidade de nossos sistemas. As listas podem conter diferentes aspectos, tais como:

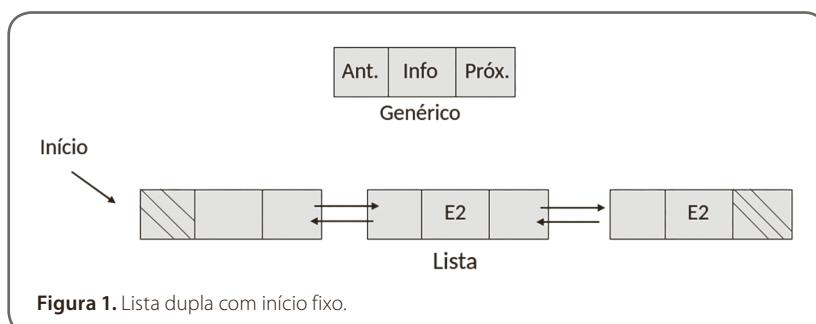
- Tipo de informação: neste caso, a lista pode armazenar qualquer tipo de informação, como nome, endereço, contatos, dentre outros elementos existentes no mundo real.
- Ordem dos elementos: aqui, varia conforme a política de negócio, podendo estar em ordem de inserção, crescente ou decrescente, etc.
- Homogeneidade da informação: todos os elementos de uma lista podem ser representados pelo mesmo tipo de dado (inteiro, *string* ou uma classe criada), ou armazenar tipos de dados diferentes. Em linguagens fracamente tipadas, como Python e PHP, por exemplo, esse comportamento é visto com maior naturalidade.

Um problema que aparece em tom de vantagem pela utilização das listas duplamente encadeadas é que estas permitem a utilização de mais um ponteiro para referenciar o elemento anterior, o que as difere de uma lista simplesmente encadeada, permitindo menor esforço para que sejam feitas operações como a de remoção, por exemplo. E essa abordagem de listas abre espaço para a utilização de outras listas, como as listas duplas com cabeça fixa, as circulares, as multidimensionais, as transversais (muito usadas em combinação com a multidimensional) e as com a combinação desses tipos.

Listas com cabeça fixa

Geralmente, ao se iniciar uma lista encadeada, é comum verificar se ela está vazia. Essa verificação é realizada para retornar uma mensagem ao usuário, para que ele tenha ciência do que está acontecendo. Uma forma de deixar o código mais simples e compacto é com a utilização de listas com cabeça fixa, que consistem em fixar o primeiro nó (sendo ele o responsável por essa denominação), e não armazenam nenhum elemento.

Uma lista com cabeça fixa é inicializada, de modo a alocar um nó e preenchermos seu campo com um endereço nulo (sem referência à memória). Seu campo de informação não precisa ser preenchido, pois apenas pertence à lista física (e não à lista lógica) (PIVA JUNIOR *et al.*, 2014). Para ilustrar melhor, a Figura 1 apresenta uma lista com três valores de nós, mas com apenas dois deles contendo informação.



Fique atento

Existem vários tipos de listas que podem ser utilizadas conforme a necessidade. Logo, podemos usar listas duplamente encadeadas com uma técnica conhecida como cabeça fixa. A diferença para as listas costumeiramente implementadas é que, além do ponteiro de próximo, o ponteiro de anterior também deve ser iniciado com o valor nulo.

2 Análise da utilização de listas encadeadas duplas em problemas computacionais

A utilização de listas encadeadas duplas, para solucionar alguns problemas computacionais, pode apresentar soluções mais otimizadas do que outras abordagens de solução, sempre respeitando as propriedades de uma lista encadeada dupla. Dentre os diversos problemas computacionais, realizaremos a análise do algoritmo de ordenação *QuickSort* aplicado em uma lista duplamente encadeada. Geralmente, essas análises de desempenho de um algoritmo são realizadas usando-se as seguintes medidas:

- Espaço necessário para concluir a tarefa desse algoritmo (complexidade do espaço). Inclui espaços para programa e para dados.
- Tempo necessário para concluir a tarefa desse algoritmo (complexidade do tempo).

A complexidade dessa implementação é igual à de tempo do *QuickSort* para a ordenação de matrizes, ou seja, leva $O(n^2)$ tempo no pior caso e $O(n \log n)$ no melhor caso (CORMEN, 2002). O pior caso ocorre quando a lista vinculada já está classificada, pois é necessário percorrer toda a lista.

Existe uma implementação do *QuickSort* que é mais eficiente, o *QuickSort* aleatório. No entanto, o *Quicksort* pode ser implementado nas listas encadeadas apenas quando podemos escolher um ponto fixo como o pivô, dessa forma, o *QuickSort* aleatório não pode ser implementado com eficiência nas listas encadeadas escolhendo o pivô aleatório.

Além dos problemas de ordenação, existem outros problemas, como os aritméticos, que podem ser solucionados utilizando-se listas encadeadas duplas, como, por exemplo, contar três números na lista que somam um determinado valor x . Utilizando-se listas duplamente encadeadas, existem três abordagens para solucionar esse problema:

1. Abordagem ingênua — essa é a abordagem mais simples, no entanto, a mais complexa. Utilize três *loops* aninhados para gerar todas as combinações de três números e verifique se a soma dos três elementos é igual a x ou não.

Complexidade de tempo: $O(n^3)$

Espaço auxiliar: $O(1)$

2. *Hashing* — percorra a lista duplamente vinculada e armazene os dados de cada nó e seu par de ponteiros (tupla) na tabela de *hash*. Agora, gere cada par possível de nós. Para cada par de nós, calcule a soma dos dados nos dois nós e verifique se o **valor** existe ou não na tabela de *hash*.

Complexidade do tempo: $O(n^2)$

Espaço auxiliar: $O(n)$

3. Uso de dois ponteiros — percorra a lista duplamente vinculada da esquerda para a direita. Para cada nó **atual** durante a travessia, inicie dois ponteiros: **primeiro** = ponteiro para o nó próximo ao nó **atual** e **último** = ponteiro para o último nó da lista. Agora, conte os pares na lista do **primeiro** ao **último** ponteiro que somam o valor.

Complexidade do tempo: $O(n^2)$

Espaço auxiliar: $O(1)$

Com esses exemplos, é possível verificar que, dependendo dos problemas, existem maneiras diferentes de solucionar um problema usando listas duplamente encadeadas e que, geralmente, cada abordagem apresenta um desempenho diferente. Isso ocorre devido à flexibilidade de navegar entre os elementos da lista duplamente encadeada. Na Figura 2, é possível verificar o comportamento de cada complexidade de tempo dos problemas apresentados.

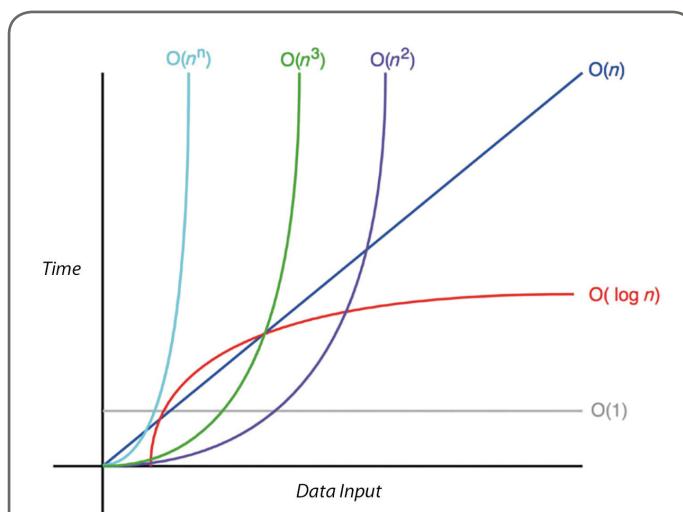


Figura 2. Notação Big O.

Fonte: Brundi (2019, documento on-line).

Além da complexidade da resolução dos problemas computacionais, que varia muito de problema para problema, existem as complexidades de cada operação básica que é feita em uma lista duplamente encadeada, como é possível verificar no Quadro 1.

Quadro 1. Complexidade temporal de operações comuns

Média				Pior			
Acesso	Procura	Inserção	Eliminação	Acesso	Procura	Inserção	Eliminação
$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$



Saiba mais

Existe uma versão de lista encadeada dupla que utiliza apenas um espaço para o campo de endereço em cada nó. Essa lista é chamada de lista encadeada XOR (disjunção exclusiva), ou lista duplamente encadeada eficiente de memória, pois em vez de armazenar endereços de memória reais, cada nó armazena o XOR dos endereços dos nós anteriores e dos próximos.

3 Resolução de problemas computacionais utilizando listas encadeadas duplas

Os problemas computacionais são situações que podem ser resolvidas passo a passo com o computador. Por essa natureza, esses problemas geralmente são bem definidos, com entradas bem estabelecidas, restrições e condições que satisfazem os valores de saída. Alguns dos problemas podem ser de: tomada de decisão, pesquisa, contagem, ordenação e otimização. Nesta seção, vamos apresentar a resolução do problema de ordenação rápida com abordagem em listas duplas, conhecido na literatura como *QuickSort*.

Algoritmo *QuickSort*

O algoritmo *QuickSort*, também conhecido como algoritmo de ordenação rápida, é assim chamado pela sua estratégia de divisão da lista em partes, determinada por um elemento arbitrário, ou não, denominado como pivô. Conforme Celes Filho, Cerqueira e Rangel (2004), supondo que esse elemento ocupe uma posição fixa no vetor (início, meio ou fim), de índice i , com a partição da lista original em duas sublistas, há dois problemas menores para serem resolvidos, que seriam de ordenar as sublistas. Estas passarão pelo mesmo processo de divisão, recursivamente, tornando-se cada vez menores até que reste apenas um elemento, caso em que a sua ordenação estará concluída.

Conforme Tenenbaum, Langsam e Augenstein (1995), suponha que os elementos de x (um vetor qualquer) sejam particionados de modo que os elementos $a = x[0]$ (primeiro elemento) e a sejam remanejados para a posição j , tendo que observar as seguintes condições:

- Elementos das posições 0 até $j - 1$ sejam menores ou iguais a a .
- Elementos das posições $j + 1$ até $n - 1$ sejam maiores que a .

Para fins de exemplificação do funcionamento do algoritmo de ordenação *QuickSort*, se for repassado um vetor, tal como [20-52-43-32-7-87-81-28], cujo primeiro elemento (20) é posto na posição correta de ordenação, que nesse caso será o segundo índice do vetor, o resultado será [7-20-52-43-32-87-81-28].

A primeira rodada termina com a sequência correta de ordenação dos dois primeiros elementos da lista (7 e 20 em ordem), pois cada elemento menor ou igual a 20 está corretamente posicionado, assim como para os números com valor superior. Como o número 20 já se encontra em sua posição final, o problema inicial pode ser decomposto na classificação dos dois vetores: [7] e [52-43-32-87-81-28].

Perceba que o primeiro vetor já se encontra em seu tamanho mínimo (1 elemento). No entanto, para ordenar o segundo vetor, é realizada a repetição do processo, em que o número 52 será o pivô, resultando nos seguintes subvetores: 7-[32-43-28]-52-[81-87]. Esse processo irá se repetir até que só reste um elemento em cada subvetor com o seguinte resultado ordenado: [7-28-32-43-52-81-87]. A Figura 3 apresenta uma das formas de implementação do algoritmo de *QuickSort*.

Ainda na Figura 3, é possível notar duas etapas: a primeira é que o algoritmo se divide em dois recursivamente; a segunda é a função partição, que realiza toda a ordenação dos subelementos por meio da escolha de um elemento chamado pivô.

Dividir para ...

```
QuickSort(X, início, fim)
SE (início < fim) ENTÃO
    p = Partição(X, início, fim)
    QuickSort(X, início, fim - 1)
    QuickSort(X, inicio + 1, fim)
```

... conquistar!

```
Partição(X, início, fim)
pivô = X[início]
i = início - 1
PARA (j = início) ATÉ (fim - 1)
    SE (X[j] <= pivô) ENTÃO
        i = i + 1
        TROCA X[i] com X[j]
    SE (pivô < X[i + 1]) ENTÃO
        TROCA X[i + 1] com X[fim]
    Retorna i + 1
```

Figura 3. Pseudocódigo do algoritmo de *QuickSort*.

QuickSort com lista duplamente encadeada

Uma vez compreendido o funcionamento do algoritmo *QuickSort*, veremos como é feita a sua implementação ao utilizar listas com encadeamento duplo, com sua implementação realizada na linguagem em Python3.X.

A Figura 4 apresenta a implementação do algoritmo *QuickSort* para um vetor simples, com os índices sendo números. Perceba que, para essa implementação, o pivô selecionado foi o último elemento do vetor. Ao utilizar uma lista encadeada, sabemos que precisaremos tratar os endereços de memória diretamente, e não os índices, como ocorre quando trabalhamos com vetores. Em Python, como já abordado, a manipulação do espaço de memória fica completamente abstraída pela própria linguagem, não sendo manipulada diretamente, como ocorre em linguagens como C/C++.

Implementação das funções

```

def quick_sort(lista):
    quick_sort_(lista, 0, len(lista)-1)

def quick_sort_(lista,inicio,fim):
    if inicio>fim:
        p = particao(lista,inicio,fim)
        quick_sort_(lista,inicio,p - 1)
        quick_sort_(lista,p + 1,fim)

```


Teste e Resultado

```

lista = [4,7,3,2,6,1,8,9,5]
print("lista original: {}".format(lista))
quick_sort(lista)
print("lista ordenada: {}".format(lista))

```

lista original: [4, 7, 3, 2, 6, 1, 8, 9, 5]
 lista ordenada: [1, 2, 3, 4, 5, 6, 7, 8, 9]
 >

Figura 4. *QuickSort* em Python3.X.

A implementação da lista encadeada dupla para o problema de ordenação do *QuickSort* requer um pouco mais de linhas de código, como pode ser visto nas Figuras 5 e 6. As implementações em cada função das respectivas imagens usaram a técnica de *type hint* de Python, encontrada em seu manual de boas práticas PEP8, para explicitar os tipos de dados dos argumentos das funções, além do tipo de dado que cada função retornará por meio do símbolo `->` (traço e maior).

A escolha do pivô por conveniência foi referente ao último elemento. Perceba também que ao implementarmos o *QuickSort*, precisamos criar nossa própria estrutura de lista duplamente encadeada para ser adicionada ao algoritmo de ordenação. Portanto, a Figura 5 apresenta a implementação básica da lista duplamente encadeada somente com as funções de adicionar, remover e imprimir.

```

class ListaDupla(object):
    def __init__(self):
        self.inicio = None
        self.fim = None
        print("Lista criada.")

    def lista_vazia(self):
        """ Checa se a lista está vazia ou não. """
        return self.inicio is None

    def remover(self, info : int) -> None:
        """ Remove um elemento da lista. """
        # O Elemento atual é o primeiro item da lista
        e_atual = self.inicio

        # Faz uma busca no Elemento que será removido
        # e quanto o Elemento atual for válido
        while e_atual is not None:
            # Verifica se é a info da busca
            if e_atual.info == info:
                # Se for, aponta o anterior
                # da lista, o anterior não existe
                if e_atual.ant is None:
                    # O inicio passa a ser o próximo Elemento da lista
                    self.inicio = e_atual.prox
                    # Anterior do próximo Elemento aponta para None
                    e_atual.ant = None
                else:
                    # Caso esteja no meio, redireciona os ponteiros direcionais
                    e_atual.ant.prox = e_atual.prox
                    e_atual.prox.ant = e_atual.ant
                # Tudo finaliza o bloco de repetição.
                print("\nElemento {} removido.".format(e_atual.info))
                break
            # Se não é o Elemento que estamos buscando vá para o próximo
            e_atual = e_atual.prox

    def adicionar(self, info : int) -> None:
        """ Adiciona um novo Elemento no fim da lista. """
        novo_e = Elemento(info)

        if self.lista_vazia():
            self.inicio = novo_e
            self.fim = novo_e
        else:
            # O anterior 'aponta' para o último Elemento adicionado (fim)
            novo_e.ant = self.fim
            # O proximo sempre aponta para None
            novo_e.prox = None
            # O anterior aponta sempre para o novo Elemento
            self.fim.prox = novo_e
            # e o atributo aponta para a ser o novo Elemento
            novo_e.info = info
            print("Elemento adicionado: {}.".format(novo_e.info))

```

Figura 5. Implementação da lista duplamente encadeada.

Uma vez que contamos com a implementação da lista, agora precisamos da implementação do algoritmo de ordenação, que pode ser observado na Figura 6. Perceba que foram implementadas três funções. Além da utilização de endereçamento, outra mudança adicionada foi na escolha do pivô, que foi escolhido conforme o último elemento.

Figura 6. QuickSort com lista duplamente encadeada em Python3.X.



Referências

BUNDI, B. *Understanding Big-O notation with JavaScript*. 2019. Disponível em: <https://dev.to/b0nbon1/understanding-big-o-notation-with-javascript-25mc>. Acesso em: 4 fev. 2020.

CELES, W.; CERQUEIRA, R.; RANGEL, J. L. *Introdução e estrutura de dados: com técnicas de programação em C*. 4. ed. Rio de Janeiro: Campus, 2004.

CORMEN, T. H. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2002.

PIVA JUNIOR, D. et al. *Estrutura de dados e técnicas de programação*. Rio de Janeiro: Elsevier, 2014.

TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. *Estruturas de dados em C*. São Paulo: Makron Books, 1995.



Fique atento

Os *links* para *sites da web* fornecidos neste livro foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declararam não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:



SOLUÇÕES
EDUCACIONAIS
INTEGRADAS