

Estruturas de Dados elementares

Vinicius A. Matias

May 16, 2021

1 Introdução

Estruturas de Dados são elementos básicos da computação e essenciais para um desempenho eficiente na manipulação e armazenamento de dados durante a execução de um programa. Cada estrutura tem uma peculiaridade, tornando o conhecimento e estudo delas necessário para identificar facilmente qual a melhor opção a se tomar enquanto se desenvolve um algoritmo.

2 Lista Linear Sequencial

Uma lista linear consiste no conceito de que cada elemento tem um predecessor e um sucessor, com exceção do primeiro e último elemento de uma lista. A lista linear sequencial adiciona o fator de um elemento estar na sequência do outro não somente na visão do programador, mas também na alocação de memória. Note que a maioria das linguagens de programação não permite uma manipulação direta da memória (pelo menos não de forma trivial), tais como Java e Python. A linguagem de programação C é muito relevante para quem tenha interesse de se manipular mais diretamente esses elementos. Para manter a consistência com os outros códigos desenvolvidos, as estruturas de dados serão implementadas também em Python, utilizando objetos.

Listing 1: Lista linear sequencial (estrutura)

```
1 class Registro:
2     def __init__(self, id, chave):
3         self.id = id
4         self.chave = chave
5
6 class Lista_Linear_Sequencial:
7     def __init__(self, max):
8         self.registro = [Registro] * max
9         self.n_elementos = -1
10        self.max = max
```

A implementação estática da lista linear sequencial é utilizada para armazenar valores até um limite estabelecido, algo como um arranjo de tamanho pré definido (comum em linguagens como Java e C, mas não tanto em Python). O trecho de código anterior apresenta dois objetos. O primeiro é o Registro, consistindo nos objetos que serão armazenados na Lista (no caso, um campo para o id em inteiros e um campo para a chave em forma de string). A estrutura da lista *Lista_Linear_Sequencial* armazena três atributos: o limite da lista *max*, a quantidade de elementos inseridos na lista *n_elementos* e um arranjo de *max* elementos alocados sequencialmente na memória (fica implícita a alocação em Python) *registro*. Assim que a estrutura é instanciada são reservados *max* espaços na memória para armazenar cada *registro*, mesmo que nunca sejam usados.

Para inicializar a lista basta zerar a quantidade de elementos visíveis nela.

Listing 2: Lista linear sequencial (inicialização)

```
1 def inicializar_lista(self):
2     self.n_elementos = 0
```

A estrutura já armazena a quantidade de elementos adicionados na lista, então para obter o tamanho dela basta retornar este atributo.

Listing 3: Lista linear sequencial (tamanho)

```
1 def tamanho(self):
2     return self.n_elementos
```

Para imprimir a lista basta passar sequencialmente em todos os registros de 0 à *n_elementos - 1*

Listing 4: Lista linear sequencial (exibição)

```
1 def imprimir_lista(self):
2     for i in range(self.n_elementos):
3         print("Elemento:", i,
4             "| ID:", self.registro[i].id,
5             "| Chave:", self.registro[i].chave)
```

Assim como na impressão, a busca passa por todos os elementos de 0 à $n_elementos-1$ procurando um registro com a chave igual à uma passada por parâmetro.

Listing 5: Lista linear sequencial (busca)

```

1 def busca_sequencial(self, chave):
2     for i in range(self.n_elementos):
3         if self.registro[i].chave == chave:
4             return i
5     return -1

```

Para inserir elementos, partiremos da estratégia de se inserir em uma posição escolhida pelo usuário. Para realizar esse método será necessário primeiramente verificar se é possível inserir um elemento na lista (isto é, se não está cheia) e se a posição solicitada pelo usuário é válida. Havendo possibilidade de inserção, todos os elementos à direita da posição solicitada mudarão suas referências para o índice imediatamente posterior, e com isso é possível inserir o elemento na posição solicitada sem perder os outros elementos previamente inseridos. Note que ao fim da inserção também é necessário aumentar o atributo $n_elementos$ em um.

Listing 6: Lista linear sequencial (inserção)

```

1 def inserir_registro(self, registro, posicao):
2     if (self.n_elementos == max or
3         posicao < 0 or
4         posicao > self.n_elementos):
5         return False
6
7     j = self.n_elementos
8     while j > posicao:
9         self.registro[j] = self.registro[j-1]
10        j -= 1
11    self.registro[posicao] = registro
12    self.n_elementos += 1
13    return True

```

Listing 7: Lista linear sequencial (remoção)

```

1 def excluir_elemento(self, chave):
2     posicao = self.busca_sequencial(chave)
3     if posicao == -1: return False
4
5     j = posicao
6     while j < self.n_elementos-1:
7         self.registro[j] = self.registro[j+1]
8         j += 1
9     self.n_elementos -= 1
10    return True

```

Para excluir um elemento da lista por meio da chave (passada por parâmetro) utilizamos

uma chamada ao método de busca sequencial para verificar se esta chave está na lista e, estando, podemos iniciar o processo de remoção. Como sabemos o índice que o elemento a ser excluído, nós passamos todos os elementos após essa posição para o índice anterior (move todo o arranjo posterior à posição em uma posição para a esquerda). Como movemos todo o arranjo à direita da posição em um índice à menos, a posição a ser excluída foi sobreposta pelo índice até então posterior, portanto perdendo a referência ao elemento e considerando-o excluído. Note que é necessário diminuir em um o atributo com o número de elementos da lista.

A reinicialização dessa estrutura é igual à inicialização, consistindo apenas de zerar o número de elementos válidos. Note que os elementos anteriores continuam alocados na memória, mas a estrutura não consegue mais acessá-los.

Listing 8: Lista linear sequencial (reinicialização)

```

1 def reinicializar_lista(self):
2     self.n_elementos = 0

```

2.1 Lista Linear Sequencial Ordenada

Note que a busca de qualquer elemento na estrutura mencionada pode assumir complexidade $O(n)$. Para realizar uma busca mais eficiente, é possível tomar pelo menos duas abordagens.

Uma delas é adicionar um campo sentinela no arranjo de registros, isto é, alocar espaço para mais um registro no arranjo visando realizar menos comparações na busca sequencial:

Listing 9: Lista linear sequencial com sentinela (estrutura)

```

1 def busca_sentinela(self, id):
2     i = 0
3     self.registro[self.n_elementos].id = id
4     while self.registro[i].id != id: i += 1
5     if i == self.n_elementos: return -1
6     return i

```

A alteração da estrutura permite a adição de um sentinela durante o processo de busca. Adicionar um sentinela quer dizer que aplicaremos o valor que estamos procurando na última posição válida durante a busca, ou seja, realizando um loop no intervalo $[0, n_elementos]$, com o valor que procuramos na posição $n_elementos$. O laço correrá com apenas uma condição (removendo

um segundo *if* da busca sequencial primária), pois sabe-se que eventualmente o resultado será verdadeiro.

Listing 10: Lista linear sequencial com sentinela (busca)

```
1 def busca_sentinela(self, id):
2     i = 0
3     self.registro[self.n_elementos].id = id
4     while self.registro[i].id != id: i += 1
5     if i == self.n_elementos: return -1
6     return i
```

Ainda, a busca pode se tornar mais eficiente ainda se conseguirmos aplicar o método de busca binária. Para aplicá-lo, contudo, é necessário que os elementos estejam ordenados. Para tal, podemos modificar o método de inserção da lista linear sequencial para ordenar os elementos em cada adição. Uma das maneiras é aplicando as premissas do método de ordenação por inserção:

Listing 11: Lista linear sequencial ordenada (inserção)

```
1 def inserir_registro_ord(self, registro):
2     if self.n_elementos >= self.max:
3         return False
4
5     pos = self.n_elementos
6     while (pos > 0 and
7           self.registro[pos-1].id > registro.id):
8         self.registro[pos] = self.registro[pos-1]
9         pos -= 1
10    self.registro[pos] = registro
11    self.n_elementos += 1
12    return True
```

Essa ordenação na inserção permite a utilização da busca binária na estrutura de dados criada.

Listing 12: Lista linear sequencial ordenada (busca binária)

```
1 def busca_binaria(self, id):
2     esq = 0
3     dir = self.n_elementos-1
4
5     while esq <= dir:
6         meio = int((esq+dir)/2)
7         if self.registro[meio].id == id: return meio
8         elif self.registro[meio].id < id:
9             esq = meio + 1
10        else:
11            dir = meio - 1
12
13    return -1
```

Também é possível modificar o método de remoção para realizar a busca binária ao invés da busca sequencial para procurar o elemento na estrutura, contudo, o laço do deslocamento dos elementos domina a complexidade, podendo realizar até *max* comparações.