Estruturas de Dados elementares

Vinicius A. Matias

May 21, 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 pecularidade, 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 esta na sequencia 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 amazenados 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 implicita 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 à ser excluida 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 $\mathcal{O}(n)$. Para realizar uma busca mais eficiente, é possivel 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)

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

```
def inserir_registro_ord(self, registro):
  if self.n_elementos >= self.max:
    return False

  pos = self.n_elementos
  while (pos > 0 and
  self.registro[pos-1].id > registro.id):
    self.registro[pos] = self.registro[pos-1]
  pos -= 1
  self.registro[pos] = registro
  self.n_elementos += 1
  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)

```
def busca_binaria(self, id):
  esq = 0
  dir = self.n_elementos-1

  while esq <= dir:
    meio = int((esq+dir)/2)
    if self.registro[meio].id == id: return meio
    elif self.registro[meio].id < id:
      esq = meio + 1
    else:
      dir = meio -1

  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.

3 Lista Ligada

Listas ligadas divergem da lista sequencial pois os elementos não são mais alocados sequencialmente, mas sim em endereços de memória dispersos. Essa implementação requer que cada elemento tenha uma referência, ao menos, ao próximo elemento da lista, e portanto devendo se atentar nessa troca de endereços durante inserções e remoções. Falaremos de três maneiras de se implementar uma lista ligada.

3.1 Implementação Estática

Listing 13: Lista Ligada estática ordenada (estrutura)

```
MAX = 50

class Registro:
    def __init__(self, id, chave):
        self.id = id
        self.chave = chave

class Elemento:
    def __init__(self, registro, prox):
        self.reg = registro
        self.prox = prox

class Lista:
    def __init__(self):
        self.elemento = [Elemento] * MAX
        self.inicio = None
        self.dispo = None
```

Consiste em utilizar um arranjo para persistir os endereços de memória válidos para a estrutura. Nessa implementação o objeto *Registro* se mantém inalterado, a Lista é alterada e é criado um novo objeto chamada *Elemento*. Cada registro está armazenado em um Elemento, e os elementos possuem além do registro um campo para identificar o índice que está o próximo elemento. A estrutura Lista mantém um arranjo de Elementos, o índice do primeiro elemento (caso haja elementos na lista) e um campo para identificar o próximo elemento disponível na lista (isto é, um índice do arranjo que não tenha nenhum

elemento). É utilizada uma constante MAX, mas este valor poderia ser parte da estrutura da Lista também, mas não seria tão útil quanto na lista linear sequencial.

A inicialização da lista passa por todos os índices do arranjo de elementos e cria um espaço do tamanho de um registro e adiciona a referência para o próximo elemento do arranjo (índice 0 tem como próximo 1, índice 1 tem como próximo 2 etc) no intervalo $[0, MAX-1]$ o último elemento válido (índice $MAX-1$) não pode ter referência à um próximo, pois não há mais espaço na lista. Como não há nenhum elemento inserido até então, o primeiro registro indicado por $self.inicio$ é nulo e o primeiro elemento disponível para inserção é o índice 0.

Listing 14: Lista Ligada estática ordenada (inicialização)

```
def inicializar_lista(self):
  for i in range(MAX-1):
    self.elemento[i] = Elemento(Registro, i+1)
  self.elemento[MAX-1].prox = None
  self.inicio = None
  self.dispo = 0
```

O tamanho pode ser obtido passando por todos os elementos até encontrar um inválido. Note que não armazenamos o tamanho da lista nessa implementação.

Listing 15: Lista Ligada estática ordenada (tamanho)

```
def tamanho(self):
        i = self.inicio
        tamanho = 0
        while i != None:
            tamanho +=1
            i = self.elemento[i].prox
        return tamanho
```

Para visualizar os registros de uma lista podemos seguir uma estratégia semelhante à do tamanho, isto é, de passar por todos os elementos.

Listing 16: Lista Ligada estática ordenada (exibição)

```
def imprimir_lista(self):
  i = self.inicio
  print("Lista de Registros")
  while i != None:
    print("ID:", self.elemento[i].reg.id,
    "Chave:", self.elemento[i].reg.chave)
    i = self.elemento[i].prox
```

Ainda que ordenada, não podemos realizar uma busca binária para otimizar a busca sendo que não temos como definir um "meio" da estrutura. Isso leva à implementação da busca sequencial por todos valores menores que um parâmetro.

Listing 17: Lista Ligada estática ordenada (busca)

```
def busca(self, id):
    i = self.inicio
    while (i != None and
    self.elemento[i].reg.id < id):
        i = self.elemento[i].prox
    if (i != None and
    self.elemento[i].reg.id == id):
        return i
```

A inserção em uma Lista ligada estática vai requerer um método adicional: *obter_no*(). Estamos considerando que a inserção será ordenada e não poderá haver dois id's iguais na mesma lista.

Listing 18: Lista Ligada estática ordenada (inserção)

```
def obter_no(self):
  resultado = self.dispo
  if self.dispo != None:
        self.dispo = self.elemento[self.dispo].prox
  return resultado

def inserir(self, reg):
  if (self.dispo == None): return False
  ant = None
  i = self.inicio
  id = reg.id
  while (i != None and self.elemento[i].reg.id < id):
    ant = i
    i = self.elemento[i].prox
  if (i != None and self.elemento[i].reg.id == id):
        return False
  i = self.obter_no()
  self.elemento[i].reg = reg
  if ant == None:
    self.elemento[i].prox = self.inicio
    self.inicio = i
  else:
    self.elemento[i].prox = self.elemento[ant].prox
    self.elemento[ant].prox = i
  return True
```

Havendo espaço na lista ligada, percorreremos todos os elementos até econtrar um id maior ou igual ao que queremos inserir **ou** o último elemento válido da lista. No loop nós armazenamos o endereço do último elemento do laço **e**

o endereço do elemento anterior à ele, isto é, o que aponta para ele. Na sequencia é feita uma comapração para verificar se o enderço do elemento é válido e se o id do registro é igual ao id solicitado para inclusão, se qualquer um dos testes forem verdadeiros, nós não inserimos o registro na lista. Estando tudo ok, chamamos a função de obter nó, retornando o próximo espaço do arranjo disponível para inserção, atualizando também o índice sem alocamento (*dispo*) para o próximo da lista (o que ele apontava). Adicionado, podemos verificar se o elemento à ser inserido é o primeiro da lista, e se for adicionamos na posição 0 e atualizamos o ponteiro *inicio* do objeto Lista para a nova referência. Se o elemento precisa ser inserido em uma posição que não a primeira, mudamos o ponteiro do anterior (que estavamos armazenando desde o primeiro loop) para o novo registro, e o ponteiro de próximo do novo registro vai para o que era o próximo do anterior.

Listing 19: Lista Ligada estática ordenada (remoção)

```
1  def devolver_no(self, posicao):
2    self.elemento[posicao].prox = self.dispo
3    self.dispo = posicao
4
5  def excluir(self, id):
6    ant = None
7    i = self.inicio
8    while (i != None and
9    self.elemento[i].reg.id < id):
10     ant = i
11     i = self.elemento[i].prox
12   if (i == None or self.elemento[i].reg.id != id):
13         return False
14   if ant == None:
15     self.inicio = self.elemento[i].prox
16   else:
17     self.elemento[ant].prox = self.elemento[i].prox
18   self.devolver_no(i)
19   return True
```

A exclusão de um elemento em uma lista ligada ordenada segue o mesmo paradigma que a inserção, notando que quando encontrado um índice para ser removido deva ser perdida a referência à esse elemento, podendo ser feita por meio do ponteiro anterior ao elemento a ser removido começar a apontar ao que era o próximo do elemento a ser excluído. Note que também é necessário atualizar os nós disponíveis, e isso pode ser feito pelo método *develover_no*(), levando o índice do elemento excluído ao atributo *dispo* da lista.

A reinicialização da estrutura deve limpar os registros armazenados e atualizar os atributos para uma lista totalmente nova, ou seja, chamando o método de inicialização novamente.

Listing 20: Lista Ligada estática ordenada (reinicialização)

```
1  def reinicializar_lista(self):
2          self.inicializar_lista()
```

3.2 Implementação Dinâmica

3.3 Implementação Circular com nó-cabeça

4 Pilha

4.1 Implementação estática

4.2 Implementação dinâmica

5 Deque

6 Fila

6.1 Implementação estática

6.2 Implementação dinâmica

7 Matrizes esparsas