

Estruturas de Dados elementares

Vinicius A. Matias

June 5, 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.

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)

```

1  MAX = 50
2
3  class Registro:
4      def __init__(self, id, chave):
5          self.id = id
6          self.chave = chave
7
8  class Elemento:
9      def __init__(self, registro, prox):
10         self.reg = registro
11         self.prox = prox
12
13  class Lista:
14      def __init__(self):
15         self.elemento = [Elemento] * MAX
16         self.inicio = None
17         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)

```

1 def inicializar_lista(self):
2     for i in range(MAX-1):
3         self.elemento[i] = Elemento(Registro, i+1)
4     self.elemento[MAX-1].prox = None
5     self.inicio = None
6     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)

```

1 def tamanho(self):
2     i = self.inicio
3     tamanho = 0
4     while i != None:
5         tamanho += 1
6         i = self.elemento[i].prox
7     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)

```

1 def imprimir_lista(self):
2     i = self.inicio
3     print("Lista de Registros")
4     while i != None:
5         print("ID:", self.elemento[i].reg.id,
6               "Chave:", self.elemento[i].reg.chave)
7         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)

```

1 def busca(self, id):
2     i = self.inicio
3     while (i != None and
4            self.elemento[i].reg.id < id):
5         i = self.elemento[i].prox
6     if (i != None and
7         self.elemento[i].reg.id == id):
8         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)

```

1 def obter_no(self):
2     resultado = self.dispo
3     if self.dispo != None:
4         self.dispo = self.elemento[self.dispo].prox
5     return resultado
6
7 def inserir(self, reg):
8     if (self.dispo == None): return False
9     ant = None
10    i = self.inicio
11    id = reg.id
12    while (i != None and self.elemento[i].reg.id < id):
13        ant = i
14        i = self.elemento[i].prox
15    if (i != None and self.elemento[i].reg.id == id):
16        return False
17    i = self.obter_no()
18    self.elemento[i].reg = reg
19    if ant == None:
20        self.elemento[i].prox = self.inicio
21        self.inicio = i
22    else:
23        self.elemento[i].prox = self.elemento[ant].prox
24        self.elemento[ant].prox = i
25    return True

```

Havendo espaço na lista ligada, percorreremos todos os elementos até encontrar 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 sequência é feita uma comparação para verificar se o endereço 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 alocação (*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 estavam 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 *devolver_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

A implementação dinâmica de uma lista ligada acaba sendo o mais viável computacionalmente. Essa estrutura não deixa alocado um espaço para n registros como nas estruturas vistas até agora, mas aloca cada elemento sob demanda. Para isso ela mantém a estrutura **Elementos**, e tem uma adaptação na estrutura básica da lista, armazenando por padrão apenas a referência ao primeiro elemento da lista.

Listing 21: Lista Ligada dinâmica (estrutura)

```

1  class Registro:
2      def __init__(self, id, chave):
3          self.id = id
4          self.chave = chave
5
6  class Elemento:
7      def __init__(self, registro, prox):
8          self.reg = registro
9          self.prox = prox
10
11  class Lista:
12      def __init__(self):
13          self.inicio = None

```

A inicialização da lista pode ter diferentes implementações dependendo da linguagem de programação, mas em Python simplesmente não haverá diferença entre a criação da estrutura e a inicialização por um outro método.

Listing 22: Lista Ligada dinâmica (inicialização)

```

1  def inicializar_lista(self):
2      self.inicio = None

```

Para o tamanho, porém, será necessário contar todos os elementos da lista, isto é, passar para o próximo elemento apontado por um anterior, até chegar ao último elemento válido (diferente de **None**). Vale lembrar que cada registro está dentro de uma estrutura (em Python são objetos mesmo) elemento.

Listing 23: Lista Ligada dinâmica (tamanho)

```

1  def tamanho(self):
2      endereco = self.inicio
3      tamanho = 0
4      while endereco != None:
5          tamanho += 1
6          endereco = endereco.prox
7      return tamanho

```

A exibição dos elementos segue a mesma lógica que a usada para encontrar o tamanho da lista, mas aqui são utilizados os registros dentro de cada elemento.

Listing 24: Lista Ligada dinâmica (exibição)

```

1  def imprimir_lista(self):
2      endereco = self.inicio
3      print(" Lista de Registros")
4      while endereco != None:
5          print(" ID:", endereco.reg.id,
6                " Chave:", endereco.reg.chave)
7          endereco = endereco.prox

```

Em uma estrutura de lista não ordenada, a busca pode ser implementada buscando, em cada elemento válido, se o registro tem o campo correspondente ao buscado.

Listing 25: Lista Ligada dinâmica (busca sequencial)

```

1  def busca(self, chave):
2      posicao = self.inicio
3      while posicao != None:
4          if posicao.reg.chave == chave:
5              return posicao
6          posicao = posicao.prox
7      return None

```

Porém, podemos adaptar o método de busca sequencial acima para outra busca sequencial que retorna também o elemento anterior ao buscado. Essa manobra é importante para se fazer uma inserção mais eficiente.

Listing 26: Lista Ligada dinâmica (busca o anterior e atual)

```

1  def busca_dupla(self, chave):
2      atual = self.inicio
3      anterior = None
4      while atual != None:
5          if atual.reg.chave == chave:
6              return anterior, atual
7          anterior = atual
8          atual = atual.prox
9      return anterior, None

```

Com auxílio da busca que retorna além do elemento visado, também seu anterior, podemos utilizá-la para verificar se a inserção é viável. Se existirem dois elementos de mesma chave nós não iremos inseri-lo na estrutura, caso contrário a inserção é possível e devemos verificar se o elemento entrará na primeira "alocação" da lista (quando não há um anterior) ou se entrará em qualquer outra posição. Ajustamos os ponteiros para o caso que se enquadrar e a estrutura terá o novo elemento disponível para acesso na lista.

Listing 27: Lista Ligada dinâmica (inserção)

```

1  def inserir(self, reg):
2      chave = reg.chave
3      anterior, atual = self.busca_dupla(chave)
4      if atual != None: return False
5      if anterior == None:
6          novo = Elemento(reg, l.inicio)
7          self.inicio = novo
8      else:
9          novo = Elemento(reg, anterior.prox)
10         anterior.prox = novo
11     return True

```

A remoção de elementos também utiliza a referência ao elemento anterior ao buscado, e uma troca entre ponteiros simples (anterior passa a apontar ao que seria o próximo do elemento removido, perdendo a referência).

Listing 28: Lista Ligada dinâmica (remoção)

```

1  def excluir(self, chave):
2      anterior, atual = self.busca_dupla(chave)
3      if atual == None: return False
4      if anterior == None:
5          self.inicio = atual.prox
6      else:
7          anterior.prox = atual.prox
8      return True

```

A reinicialização ideal deve excluir todos os elementos alocados anteriormente na memória. Isso fica mais claro em C, mas em Python é possível que percamos a referência aos elementos e, de tempos em tempos, o Garbage Collector removerá os valores alocados nesses endereços de memória.

Listing 29: Lista Ligada dinâmica (reinicialização)

```

1  def reinicializar_lista(self):
2      self.inicio = None

```

3.3 Implementação Circular com nó-cabeça

Uma lista ligada circular facilita o percorri-mento de uma lista no sentido de nunca haver um elemento nulo à ser acessado. Isso pode economizar operações em laços, aliada a abordagem de nó cabeça na lista. Esse "nó" funciona como um sentinela, pois não é um elemento que deva entrar na lista acessível ao usuário, mas é implementado como se fosse, de modo a con-tribuir na redução da quantidade de operações. As estruturas usadas são as mesmas da lista dinâmica, somente com a mudança no nome da variável `inicio` para `cabeça` na estrutura da lista.

Listing 30: Lista Ligada circular (estrutura)

```
1 class Registro:
2     def __init__(self, id, chave):
3         self.id = id
4         self.chave = chave
5
6 class Elemento:
7     def __init__(self, reg, prox):
8         self.reg = reg
9         self.prox = prox
10
11 class Lista:
12     def __init__(self):
13         self.cabeça = None
```

A inicialização dessa estrutura aloca espaço para um registro e um elemento no nó cabeça, sendo essa posição para um elemento o próximo nó. Como a lista só tem esse nó cabeça como elemento, o próximo deverá apontar para si mesmo (por isso a lista é circular).

Listing 31: Lista Ligada circular (inicialização)

```
1 def inicializar_lista(self):
2     self.cabeça = Elemento(Registro, Elemento)
3     self.cabeça.prox = self.cabeça
```

Para recuperar o tamanho da lista utilizamos a mesma abordagem da lista dinâmica apresen-tada anteriormente.

Listing 32: Lista Ligada circular (tamanho)

```
1 def tamanho(self):
2     i = self.cabeça.prox
3     tam = 0
4     while i != self.cabeça:
5         tam += 1
6         i = i.prox
7     return tam
```

A exibição também segue a implementação da lista dinâmica, e é bem parecida com a função para recuperar o tamanho.

Listing 33: Lista Ligada circular (exibição)

```
1 def imprimir_lista(self):
2     i = self.cabeça.prox
3     print("Lista de Registros")
4     while i != self.cabeça:
5         print("ID: ", i.reg.id, "Chave: ", i.reg.chave)
6         i = i.prox
```

A busca em uma lista ordenada pode se aproveitar do fato de que o nó cabeça funciona como um sentinela para as listas estudadas anteriormente. Nesse caso podemos usar apenas uma condição para girar o loop, pois sabemos que em algum momento a condição falhará e segará no sentinela / nó cabeça.

Listing 34: Lista Ligada circular (busca)

```
1 def busca_sentinela(self, id):
2     i = self.cabeça.prox
3     self.cabeça.reg.id = id
4     while i.reg.id < id:
5         i = i.prox
6     if i != self.cabeça and i.reg.id == id: return i
7     return None
```

A estrutura de busca apresentada acima pode ser adaptada para retornar também o elemento anterior ao buscado.

Listing 35: Lista Ligada circular (busca anterior e atual)

```
1 def busca_dupla(self, id):
2     anterior = self.cabeça
3     i = self.cabeça.prox
4     self.cabeça.reg.id = id
5     while i.reg.id < id:
6         anterior = i
7         i = i.prox
8     if i != self.cabeça and i.reg.id == id:
9         return anterior, i
10    return anterior, None
```

O conhecimento do elemento anterior e a utilização do nó cabeça permite uma redução signi-ficativa no algoritmo para inserção comparado à lista dinâmica sem nó cabeça. Nessa estrutura não precisamos nos preocupar com o primeiro nó da lista e seu anterior, pois o nó cabeça irá controlar os extremos da lista.

Listing 36: Lista Ligada circular (inserção)

```
1 def inserir(self, reg):
2     anterior, i = self.busca_dupla(reg.id)
```

```

3     if i != None: return False
4     novo = Elemento(reg, anterior.prox)
5     anterior.prox = novo

```

A remoção apenas troca os ponteiros do elemento anterior ao ser excluído para o próximo do que está dando tchau.

Listing 37: Lista Ligada circular (remoção)

```

1  def excluir(self, id):
2      anterior, i = self.busca_dupla(id)
3      if i == None: return False
4      anterior.prox = i.prox
5      del i
6      return True

```

A reinicialização em Python segue os mesmos princípios comentados na estrutura dinâmica de lista ligada.

Listing 38: Lista Ligada circular (reinicialização)

```

1  def reinicializar_lista(self):
2      self.cabeca.prox = self.cabeca

```

4 Pilha

Pilhas são estruturas de dados que seguem o princípio de que cada elemento novo inserido ficará no topo da estrutura, isto é, será o primeiro a sair em uma remoção (Last In First Out). Não há muito sentido em se fazer consultas à elementos nessa estruturas, portanto elas não são otimizadas para uma busca eficiente.

4.1 Implementação estática

A implementação estática depende, assim como nas listas, de um arranjo auxiliar para armazenar os valores acessíveis. A inicialização da pilha já pode ser definida pela função `__init__`, que é o construtor de um objeto em Python. Nossa estrutura consiste então de um arranjo de `MAX` valores e uma variável para indicar o topo (limite) da pilha.

Listing 39: Pilha estática (estrutura)

```

1  MAX = 50
2
3  class Registro:
4      def __init__(self, id, chave):
5          self.id = id
6          self.chave = chave
7

```

```

8  class Pilha:
9      def __init__(self):
10         self.reg = [Registro] * MAX
11         self.topo = -1

```

Como temos o topo da pilha conhecido, podemos descobrir o tamanho da estrutura adicionando 1 ao topo.

Listing 40: Pilha estática (tamanho)

```

1  def tamanho(self):
2      return self.topo + 1

```

A impressão dos valores consiste em passar sequencialmente por todos os elementos, como de praxe.

Listing 41: Pilha estática (exibição)

```

1  def imprimir(self):
2      print("PILHA:")
3      i = self.topo
4      while i >= 0:
5          print("ID", self.reg[i].id,
6                "Chave:", self.reg[i].chave)
7          i -= 1

```

A inserção em uma pilha também é chamada de *PUSH*. Basta apenas adicionar um elemento na posição indicada pelo topo da pilha.

Listing 42: Pilha estática (inserção)

```

1  def push(self, reg):
2      if self.topo >= MAX-1: return False
3      self.topo = self.topo + 1
4      self.reg[self.topo] = reg
5      return True

```

Para remover (operação também chamada de *POP*) basta encontrar o topo e reduzir em 1 o valor dele, assim o elemento que estava na sequência estará inacessível e suscetível à sobrescrita.

Listing 43: Pilha estática (remoção)

```

1  def pop(self):
2      if self.topo == -1: return False
3      removed = self.reg[self.topo]
4      self.topo = self.topo - 1
5      return removed

```

Para reinicializar uma pilha estática basta mudar o valor do topo para -1.

Listing 44: Pilha estática (reinicializar)

```

1  def reinicializar(self):
2      self.topo = -1

```


4.2 Implementação dinâmica

A importância da implementação dinâmica de uma pilha segue a mesma importância que qualquer outra estrutura que possa ser implementada de forma estática ou dinâmica: a memória alocada. Para uma pilha, a implementação dinâmica terá apoio da estrutura `Elemento`, que "encapsula" os registros e tem um ponteiro para o próximo elemento. A estrutura de `Pilha` só precisará ter, dessa maneira, a informação do elemento do topo, conseguindo obter qualquer outra informação da pilha com isso. O topo é iniciado como uma referência nula.

Listing 45: Pilha dinâmica (estrutura)

```
1 class Registro:
2     def __init__(self, id, chave):
3         self.id = id
4         self.chave = chave
5
6 class Elemento:
7     def __init__(self, reg, prox):
8         self.reg = reg
9         self.prox = prox
10
11 class Pilha:
12     def __init__(self):
13         self.topo = None
```

Para retornar o tamanho de uma pilha será necessário contar quantos elementos existem entre o topo e `None`. A iteração acontece pelo atributo `prox` de `Elemento`.

Listing 46: Pilha dinâmica (tamanho)

```
1 def tamanho(self):
2     i = self.topo
3     tam = 0
4     while i != None:
5         tam += 1
6         i = i.prox
7     return tam
```

Para realizar algumas verificações corriqueiras, pode ser interessante criar uma função para verificar se uma pilha tem elementos.

Listing 47: Pilha dinâmica (está vazia?)

```
1 def esta_vazia(self):
2     if self.topo == None: return True
3     return False
```

A exibição é baseada na função de tamanho, funcionando com a mesma lógica acrescida da

utilização dos atributos dos registros e não somente os elementos.

Listing 48: Pilha dinâmica (exibição)

```
1 def imprimir(self):
2     i = self.topo
3     tam = 0
4     print("PILHA")
5     while i != None:
6         print("ID", i.reg.id, "Chave", i.reg.chave)
7         i = i.prox
8     return tam
```

Inserção em uma pilha dinâmica depende apenas de criar um novo registro que terá como próximo o que estava no topo, e atualizando o topo para o novo registro.

Listing 49: Pilha dinâmica (inserção)

```
1 def push(self, reg):
2     novo = Elemento(reg, self.topo)
3     self.topo = novo
4     return True
```

A remoção apenas atualiza o topo com o elemento que era apontado anteriormente pelo topo. O topo antigo é deletado da memória. Escrever sobre remoções (e outras funções) é um tanto quanto poético, então às vezes vale mais a pena focar em entender o código que talvez ajude a interpretar o texto (ambos devem caminhar juntos hein).

Listing 50: Pilha dinâmica (remoção)

```
1 def pop(self):
2     removed = self.topo.reg
3     self.topo = self.topo.prox
4     del removed
5     return True
```

Como já falei várias vezes, o topo vai ser reinicializado como `None` e isso implica que o coletor de lixo do Python irá excluir as memórias alocadas e sem uso por nós.

Listing 51: Pilha dinâmica (reinicialização)

```
1 def reinicializar(self):
2     self.topo = None
```

5 Deque

6 Fila

6.1 Implementação estática

6.2 Implementação dinâmica

7 Matrizes esparsas