Lista em alocação encadeada

Implementação

Você já aprendeu sobre a lista em alocação contígua. Entretanto, viu que, embora de fácil implementação, as operações de inserção e remoção envolvem alterar grande parte da lista, movendo os nós posteriores em uma posição, para frente ou para trás.

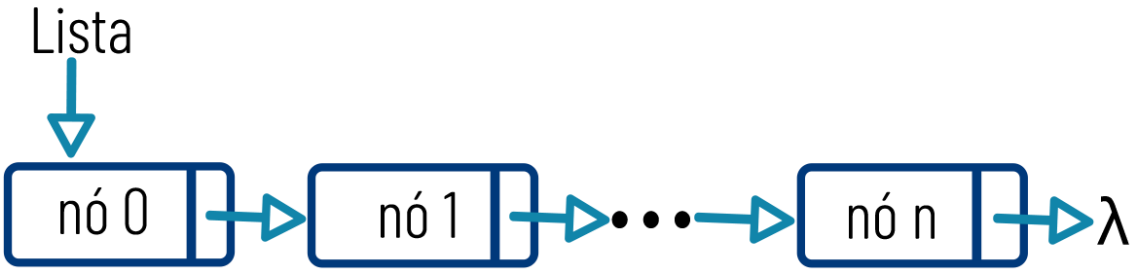
Para evitar todo esse trabalho, foram criadas as listas em alocação encadeada.

Em uma lista encadeada, existe o primeiro nó da lista, para onde apontamos a variável da lista (ou seja, a variável Lista guarda o endereço deste nó inicial).

Nessa lista, como você já viu, cada nó é composto pelos seus campos, como chave, nome, data etc., mas, além disso, possui um campo especial, um ponteiro para o próximo nó da lista.

Essa estrutura permite que os nós estejam salvos em espaços não contíguos da memória, espalhados por diversos endereços distantes entre si, mas que, ainda assim, seja possível percorrer a lista sem se perder.

Para isso, o ponteiro para o próximo nó deve armazenar o endereço em que está salvo o próximo nó. O último nó da lista, por sua vez, aponta para um valor nulo (representado por lambda ʎ).



Em memória, essa lista estará espalhada, com os ponteiros apontando para os endereços dos próximos nós.

	Endereço	Chave	Proximo
Nó 0	**16**	**157**	**308**
	...	**...**	**...**
Nó 1	**308**	**158**	**1600**

	...	...	...
Nó 2	**1600**	**159**	**Λ**

Repare que, nesse exemplo, a variável Lista apontará para o endereço 16. A lista conterá 3 nós, contendo chaves ordenadas e sequenciais, mas os endereços de memória não são contíguos.

Alocando espaço para sua lista

No caso da lista encadeada, quando você cria um nó para ser inserido nela, você já está alocando o espaço de memória que irá armazená-lo. Portanto, não precisa se preocupar em alocar previamente espaços para sua lista. Sua única preocupação é que, conforme a lista vai crescendo, o espaço em memória necessário para mantê-la aumenta a ponto de consumir o espaço disponível para seu programa.

Além disso, o tratamento dos ponteiros, apontando para os próximos nós, deve ser feito sempre com cuidado, pois, se você perder o apontamento para o próximo nó, todo o restante da lista estará perdido em endereços desconhecidos da memória.

O código a seguir cria a classe nó e seu construtor.

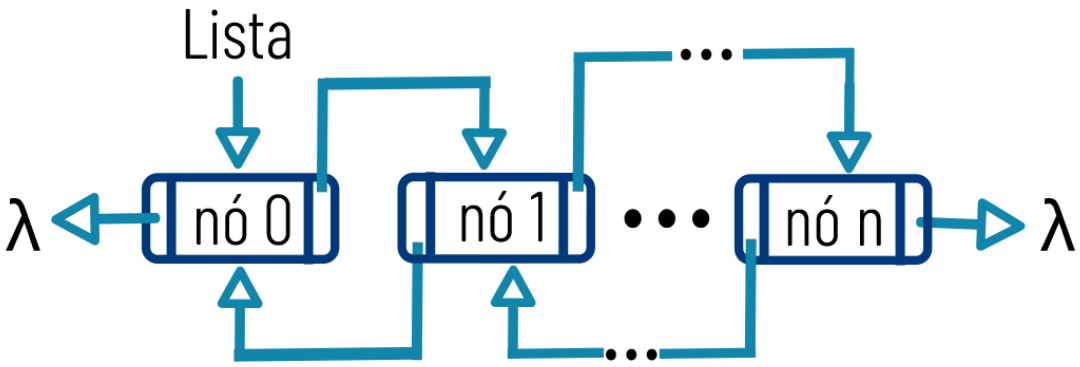
```
class No:
        def __init__(self,chave,valor):
                self.chave =chave
                self.valor = valor
                self.proximo = None
```

O próximo código cria a classe ListaEncadeada e seu construtor, além de uma função de impressão para facilitar seus testes

```
class ListaEncadeada:
        def __init__(self,cabeca=None):
                self.cabeca = cabeca
        def print(self):
           current = self.cabeca
           while current:
              print(current.valor)
              current = current.proximo
```

Lista duplamente encadeada

Com o conceito de lista encadeada, você pode percorrer a lista apenas em um sentido, seguindo os ponteiros “próximo”. Existe a estrutura de lista duplamente encadeada, na qual um nó armazena não só o ponteiro para o próximo nó, como também um ponteiro para o nó anterior. Dessa forma, a lista pode ser percorrida em ambos os sentidos, confira:



Entretanto, você consegue apenas começar pelo primeiro nó, pois só tem a variável que aponta para ele. Uma maneira de resolver esse problema é manter uma variável apontando para o último nó da lista, permitindo que você percorra a lista no sentido inverso, seguindo os ponteiros anteriores.

Operações em listas encadeadas

Para a estrutura encadeada, veremos as operações básicas: busca, inserção e remoção. Lembre-se de que o objetivo de utilizar o encadeamento era melhorar a complexidade de inserção e remoção.

Busca em listas em alocação encadeada

No caso de termos uma lista em alocação encadeada, você pode buscar determinado nó, por meio de sua chave, simplesmente percorrendo a lista e procurando por essa chave.

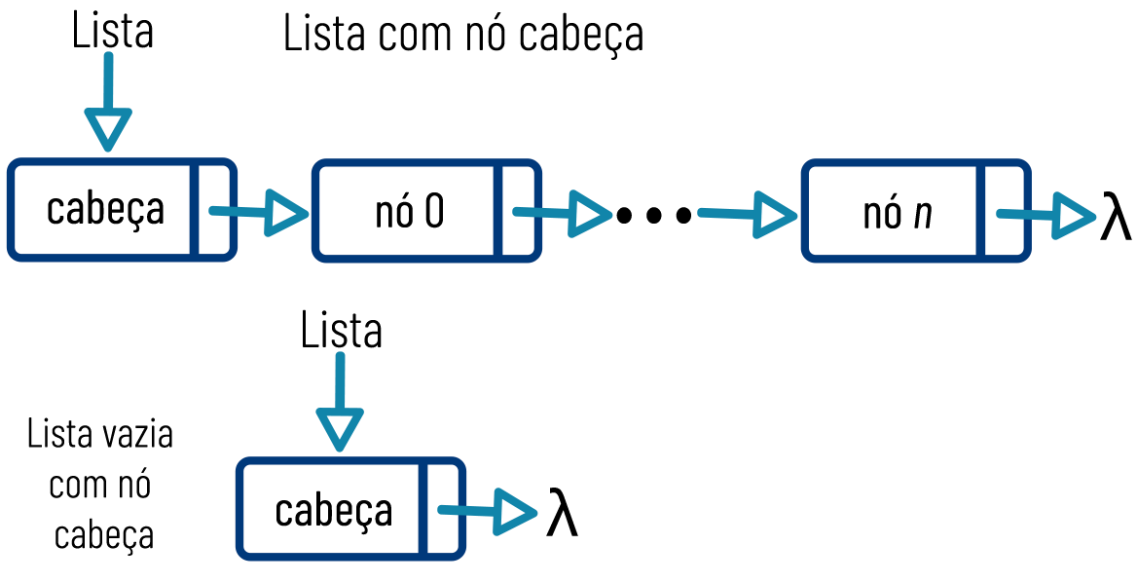
Imagine que você possua uma lista **L** contendo **n** nós compostos por [chave, valor, próximo]. O código a seguir apresenta a busca por uma chave **k**.

```
def busca(self, k):
    noAtual=self.cabeca
    if noAtual.chave==k:
        return noAtual              #chave encontrada
      while noAtual.proximo != None:
        noAtual=noAtual.proximo      #passe para próximo nó
        if noAtual.chave==k:
          return noAtual            #chave encontrada
    return None                     #indica chave não achada
```

Você percebeu que antes de começar o laço temos que testar com o primeiro nó? Isso acontece porque a lista pode conter apenas um nó. Para evitar esse problema, você pode utilizar um artifício que chamamos de “nó cabeça”

O princípio do nó cabeça é que a lista sempre terá, ao menos, um nó que não faz parte da lista de verdade, chamado de nó cabeça, que serve apenas para evitar essas checagens extras. Ou seja, se você quiser usar um nó cabeça, ao criar a lista vazia, na verdade, deverá criar contendo um nó (o cabeça) que não armazena valores válidos de um campo.

De mesma forma, uma lista que tenha removido o seu último elemento, ficando só com o nó cabeça, é uma lista vazia.



Complexidade na busca em lista encadeada

Que tal analisarmos a complexidade da função de busca apresentada? Vamos lá!

Você pode considerar que o laço de busca percorre os elementos da lista em ordem, até encontrar a chave buscada. Perceba que é o mesmo processo da lista em alocação contígua, só que, em vez de percorrer endereços sequenciais da memória, está seguindo os nós por meio dos ponteiros “próximos”.

Igualmente, no pior caso, a lista será percorrida até o último nó, levando um tempo nT. Portanto, sua complexidade é O(n), ou linear.

No caso da lista estar ordenada, teremos a mesma complexidade, mas a busca infrutífera pode ser interrompida mais cedo, de forma similar ao caso da alocação contígua. Se a chave comparada for maior que a buscada, você já pode parar a busca e retornar à chave não encontrada.

Inserção em lista em alocação encadeada

1. Criar um nó a ser inserido, o que vai alocar seu espaço em memória.
2. Apontar o campo próximo desse novo nó para o nó seguinte, de onde irá inserir.
3. Apontar o campo próximo do nó anterior, aonde você vai inserir, para o novo nó.

Primeiramente, vamos considerar o caso da lista não ordenada. Nesse caso, você pode inserir o novo nó ao final da lista, então, no passo 2, apontará para o valor nulo (ʎ).

Caso você não tenha uma variável apontando para o final da lista, terá que percorrê-la toda até encontrar o último nó. Ou seja, inserir ao final da lista só é eficiente se você tiver uma variável apontando para o final da lista. O código a seguir mostra a inserção no final da lista.

```
def insereFinal(self, novoNo):
    noAtual = self.cabeca
    if noAtual:                                  #caso a lista nao esteja vazia
     while noAtual.proximo:
            noAtual = noAtual.proximo            #busca o final da lista
            noAtual.proximo = novoNo
    else:                                        #caso a lista esteja vazia
      self.cabeca = novoNo
```

Existe outra solução mais simples, caso você não queira manter uma variável para o final da lista. Você consegue imaginar qual é?

Você pode, em vez de inserir ao final da lista, inserir novos nós no início da lista. Embora pareça estranho, lembre-se de que a lista não é ordenada. Nesse caso, o código é bem simples:

```
def insereInicio(self, novoNo):
    novoNo.proximo = self.cabeca
     self.cabeca=novoNo
```

Caso a sua lista seja ordenada, o processo de inserção começa com a busca pela posição correta de inserção, mas, ao encontrá-la, a inserção em si é simples, como as anteriores.

```
def insereOrdenada (self, novoNo):
        noAtual=self.cabeca                          #inicio da busca da posição
        if noAtual.chave> novoNo.chave:
    novoNo.proximo = self.cabeca
    self.cabeca=novoNo                                        #insere no inicio
    return 0
        if noAtual.proximo!= None:
                    while (noAtual.chave< novoNo.chave):
                                if (noAtual.proximo==None):
                                            noAtual.proximo=novoNo   #insere no final
                                        return 0
                    noAnterior=noAtual
                    noAtual=noAtual.proximo                    #continue a busca
                                                          #fim da busca
        novoNo.proximo=noAtual
#apontar novo nó
        noAnterior.proximo= novoNo
#inserir novo nó
```

Complexidade da inserção em lista encadeada

Vamos começar com a função **insereFinal**(). Sem uma variável guardando o final da lista, é necessário percorrer toda a lista, sendo assim, tem complexidade O(n). Caso você tenha a variável apontada para o final da Lista, é uma função bem simples que realiza apenas três atribuições, não importando o tamanho da entrada. Podemos dizer que sua complexidade de pior caso é O(1), ou constante.

Esse é o melhor tipo de complexidade, pois não depende do tamanho de entrada.

A função **inserelnicio**() também executa apenas duas atribuições, independentemente do tamanho de entrada. Sua complexidade de pior caso é O(1), ou constante.

No caso da função **insereOrdenada**() temos uma função bem mais complexa. Vamos usar a premissa de que cada comparação e cada atribuição leva tempo T.

Primeiro, a função vai buscar a posição de inserção. Assuma que ela realizou x comparações: o custo foi de xT.

A seguir, temos apenas o uso de duas atribuições, com custo constante 2T. Podemos perceber que, no pior caso, a busca percorre toda a lista, com x=n. Nesse caso, no qual a complexidade é (n+2)T, podemos dizer que a complexidade de pior caso é O(n).

Você pode perceber que a complexidade de pior caso não melhorou em relação à alocação contígua, pois já era O(1) no caso não ordenado e O(n) no caso ordenado, e assim se manteve.

A diferença é que, no caso ordenado, quando x<n, o programa é muito mais eficiente. Ao achar o local da inserção, o problema é resolvido em tempo constante.

Remoção em lista em alocação encadeada

A terceira e última operação básica em uma estrutura de dados é a remoção de um nó. No caso da lista encadeada, você deve buscar o nó a ser removido, e depois basta fazer o nó anterior apontar para o nó posterior ao nó removido.

No caso, você possui uma lista L e quer remover um nó de chave K.

```
def removeLista(self, K):
    noAtual = self.cabeca
    if noAtual == None:
        return None
    if noAtual.chave == K:
        self.cabeca = noAtual.proximo
        return 0
    noAnterior = noAtual
    noAtual = noAtual.proximo
    while (noAtual != None):
        if noAtual.chave == K:
            noAnterior.proximo = noAtual.proximo
            return K
        else:
            noAnterior = noAtual
            noAtual = noAtual.proximo
    return -1
```

Caso a sua lista seja ordenada, o processo de busca pode ser melhorado, permitindo que o erro de chave não encontrada seja detectado logo após ser vista uma chave maior que K.

Para testar todas as funções aqui propostas, experimente testar o seguinte código:

```
e0=No(0,'Joao')
Lista=ListaEncadeada(e0)
k0=Lista.busca(0)
print(k0.valor)
Lista.print()
e1=No(1,'Maria')
Lista.insereFinal(e1)
Lista.print()
e2=No(-1,'Ana')
Lista.insereInicio(e2)
Lista.print()
e3=No(2,'Arthur')
Lista.insereOrdenada(e3)
Lista.removeLista(2)
Lista.print()
```

Complexidade da remoção em lista encadeada

Vamos começar com a premissa de que uma comparação e uma atribuição levam tempo T.

Primeiro, a função vai buscar a posição de remoção. Assuma que ela realizou x comparações: o custo foi de 3xT. (para cada comparação, há também duas atribuições, **noAtual** e **noAnterior**).

Após encontrar o nó a ser removido, a remoção em si custa 1 operação.

Somando os dois custos, vemos que a função fará (3x+1)T operações. No pior caso, x=n, então, sua complexidade de pior caso é O(n), ou linear.

Entretanto, no caso em que x<n, o programa terá o custo efetivamente menor, pois não será necessário mexer em toda a lista. No caso de remoção infrutífera (o nó buscado não existe na lista), a pequena alteração para a lista ordenada terá uma complexidade melhor, pois você interrompe a busca assim que descobre uma chave maior que a do nó buscado.