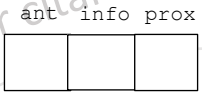
Listas Dinâmicas Duplamente Encadeadas

Listas dinâmicas duplamente encadeadas são uma variação da estrutura de dados de lista encadeada, na qual cada nó da lista contém dois ponteiros, um para o próximo nó e outro para o nó anterior na lista. Assim como nas listas simplesmente encadeadas, cada nó da lista duplamente encadeada possui um campo para armazenar o valor do dado, porém agora conta com dois ponteiros de endereçamento: um para o próximo nó da lista e outro para o nó anterior.

Dessa forma, a classe `No` define os nós da lista, com os campos `info`, que armazena o valor, e os ponteiros `prox` e `ant`, que armazenam as referências para o próximo e o nó anterior na lista, respectivamente.



Portanto, precisamos atualizar a classe `No` para incluir o ponteiro para o nó anterior, chamado de ant.

```
class No:


    def __init__(self,anterior,valor,proximo):
        self.ant = anterior
        self.info = valor
        self.prox = proximo
```

A classe `Ldde` representa a própria lista duplamente encadeada, contendo os atributos `prim` e `ult`, que apontam para o primeiro e último nós da lista, respectivamente, e o atributo `quant`, que armazena a quantidade de nós na lista.

```
class Ldde:


    def __init__(self):
        self.prim = self.ult = None
        self.quant = 0
```

A vantagem das listas dinâmicas duplamente encadeadas em relação às listas simplesmente encadeadas é que elas permitem a varredura da lista tanto para frente quanto para trás, o que pode ser útil em algumas aplicações. Porém, o uso de ponteiros adicionais torna a

implementação um pouco mais complexa e requer mais memória para armazenar as referências extras.

Porém, na `Ldde`, a manipulação dos ponteiros é um pouco mais complexa do que na lista encadeada simples, pois cada nó possui dois ponteiros: um para o próximo e outro para o nó anterior. Dessa forma, é importante ter atenção especial ao inserir e remover elementos na lista, pois os ponteiros precisam ser atualizados corretamente.

Vamos implementar a função `inserir_inicio`. Para fazer isso, primeiro precisamos verificar se a lista está vazia ou não. Em seguida, precisamos criar um nó com o valor desejado.

Se a lista estiver vazia, o novo nó será tanto o primeiro quanto o último nó da lista. Nesse caso, os ponteiros `prim` e `ult` da lista serão atualizados para apontar para o novo nó.

Se a lista não estiver vazia, o novo nó será o novo primeiro nó da lista. Nesse caso, o ponteiro para o nó anterior do atual primeiro nó da lista será atualizado para apontar para o novo nó e o ponteiro `prim` da lista será atualizado para apontar para o novo nó.

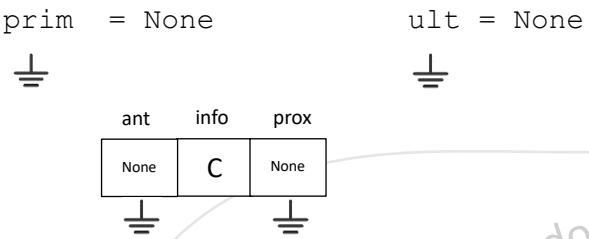
Para ilustrar esse processo, podemos começar com uma lista vazia representada pelos ponteiros `prim` e `ult` apontando para `None`:

Você verá que até este momento nada difere da `Ldse`.

Ao inserir um elemento no início da lista duplamente encadeada (`Ldde`), precisamos criar um novo nó com o valor desejado e atualizar os ponteiros dos nós para refletir a nova inserção. Para criar a estrutura do novo nó no código, utilizamos o construtor da classe `No`, que recebe como argumentos o valor do nó, o ponteiro para o nó anterior e o ponteiro para o próximo nó.

Dessa forma, podemos criar o novo nó passando o valor desejado e `None` para o ponteiro do nó anterior, já que este será o primeiro nó da lista. Em seguida, para o ponteiro do próximo nó, passamos o atual primeiro nó da lista (que pode ser `None`, caso a lista esteja vazia).

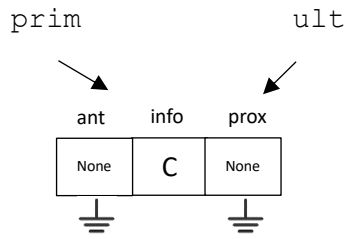
Por exemplo, ao inserir o valor 'C' em uma lista vazia, criamos um novo objeto `No` com valor igual a 'C', ponteiro para o nó anterior igual a `None` e ponteiro para o próximo nó igual a `None` (já que não há nenhum nó nem antes e nem após ele).



Estando a lista vazia, como neste caso, atualizamos os ponteiros `prim` e `ult` da lista para apontar para este nó recém-criado:

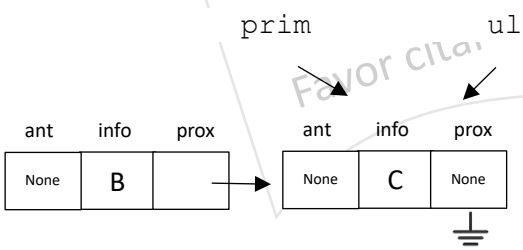
```
self.prim = self. ult = No(None, valor, None)
```

Com essa linha de código, criamos um novo nó com o valor desejado e fazemos com que `prim` e `ult` apontem para ele, já que este é o único elemento na lista.



Vale ressaltar que a estratégia de desenhar a estrutura lógica da lista pode ajudar no processo de criação do código, facilitando a visualização dos ponteiros e das relações entre os nós da lista.

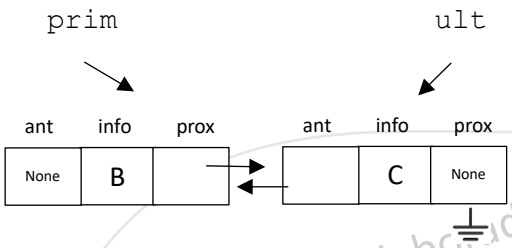
Vamos ver isso considerando agora a situação da lista não estar vazia e que desejamos inserir o valor 'B'. Neste caso precisamos criar um novo nó que aponte para o atual primeiro nó da lista e para `None` no ponteiro para o nó anterior.



Trazendo isso para o código, a chamada do construtor ficaria assim:

```
No(None, valor, self.prim)
```

Em seguida, precisamos atualizar o ponteiro para o nó anterior do atual primeiro nó da lista para apontar para o novo nó e atualizamos o ponteiro `prim` da lista para apontar para o novo nó.



Para compreender melhor esse processo de inserção, podemos visualizar a estrutura da lista acima e considerar que, ao inserirmos um novo elemento antes do primeiro elemento atual, esse novo elemento deve apontar para o nó que, até então, estava na posição de primeiro elemento, já que é o novo primeiro nó da lista. Lembrando que isso é possível graças ao atributo `prim`, que guarda o endereço do primeiro nó da lista.

Ou seja, teremos no código esta linha:

```
self.prim = No(None, valor, self.prim)
```

O resultado final é a lista contendo um novo nó com valor desejado no início, representada pelos ponteiros `prim` e `ult` apontando para o novo nó e os ponteiros do novo nó e do antigo primeiro nó corretamente atualizados.

Finalmente, temos para a função `inserir_inicio` o seguinte código:

```
    def inserir_inicio(self,valor):
        if self.quant == 0:
            self.prim = self.ult = No(None,valor,None)
        else:
            self.prim.ant = self.prim = No(None,valor,self.prim)
        self.quant+=1
```

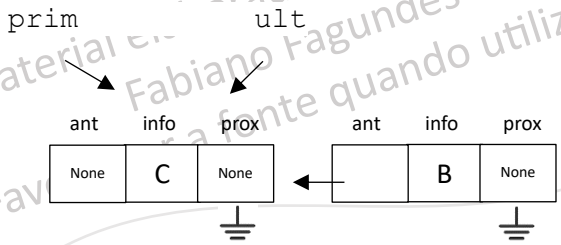
Assim como na função `inserir_inicio`, a função `inserir_fim` também verifica se a lista está vazia antes de inserir o novo elemento. Se a lista estiver vazia, precisamos criar um novo nó com o valor desejado e atualizar os ponteiros `prim` e `ult` da lista para apontar para ele.

```
self.prim = self. ult = No(None, valor, None)
```

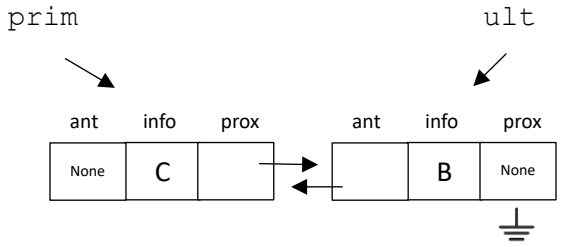
Caso contrário, criamos um novo nó com o valor desejado, tendo como anterior o atual último elemento, e `None` como endereço do próximo elemento, já que o novo nó será o último da lista e não haverá outro elemento após ele.

```
No(self.ult, valor, None)
```

Para ilustrar esse processo, consideremos a inserção do valor 'B' no fim da lista. Em uma representação visual da estrutura lógica da lista, temos um novo nó contendo o valor 'B' que foi adicionado, mas ele ainda não está conectado à lista:



Por fim, para ligar o novo nó à lista, atualizamos a referência do último elemento atual (`self.ult.prox`) para apontar para o novo nó e, em seguida, atualizamos o atributo `ult` da lista para apontar para o novo nó.



Assim, com a atribuição realizada pelo código abaixo, o novo nó é inserido no final da lista e o atributo `ult` é atualizado para apontar para ele, garantindo a correta atualização da estrutura da lista.

```
self.ult.prox = self.ult = No(self.ult,valor,None)
```

Lembrando que a ordem de atribuição na linha de código que atualiza o atributo `ult` é importante e não pode ser alterada, pois ela realiza duas ações principais em uma única linha, porém cada uma em seu momento, garantindo que as referências sejam atualizadas corretamente ao inserir um novo elemento no fim da lista.

Nosso código da função `inserir_fim` fica, então, da seguinte forma:

```
    def inserir_fim(self,valor):
        if self.quant == 0:
            self.prim = self.ult = No(None,valor,None)
        else:
            self.ult.prox = self.ult = No(self.ult,valor,None)
        self.quant+=1
```

Um ponto interessante de se observar é a simetria que começa a haver entre funções com objetivos semelhantes porém que trabalham em posições diferentes, dado o fato de ser uma lista duplamente encadeada. Vamos comparar as funções que acabamos de ver: `inserir_inicio` e `inserir_fim`.

```
    def inserir_inicio(self,valor):
        if self.quant == 0:
            self.prim = self.ult = No(None,valor,None)
        else:
            self.prim.ant = self.prim = No(None,valor,self.prim)
```

```
    def inserir_fim(self,valor):
        if self.quant == 0:
            self.prim = self.ult = No(None,valor,None)
        else:
            self.ult.prox = self.ult = No(self.ult,valor,None)
        self.quant+=1
```

Observando as implementações das duas funções, podemos notar que há uma simetria no sentido de que os ponteiros `prim` e `ult` têm seus papéis invertidos na inserção do início para a inserção no fim da lista.

Na inserção no início, o novo nó é criado com o valor desejado e um ponteiro para o próximo nó (que é o atual primeiro nó da lista) e o ponteiro para o nó anterior é `None`, já que o novo nó será o primeiro da lista. Em seguida, o ponteiro do nó anterior ao atual primeiro nó (que antes apontava para `None`) é atualizado para apontar para o novo nó, e o ponteiro `prim` da lista é atualizado para apontar para o novo nó.

Já na inserção no fim, o novo nó é criado com o valor desejado e um ponteiro para o nó anterior (que é o atual último nó da lista) e o ponteiro para o próximo nó é `None`, já que o novo nó será o último da lista. Em seguida, o ponteiro do último nó da lista (que antes apontava para `None`)

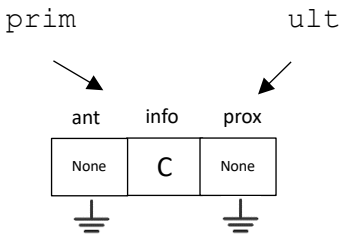
é atualizado para apontar para o novo nó, e o ponteiro `ult` da lista é atualizado para apontar para o novo nó.

Dessa forma, podemos perceber que onde havia a manipulação do ponteiro anterior (`ant`) na inserção no início, temos agora a manipulação do ponteiro próximo (`prox`) na inserção no fim. Além disso, onde antes atualizávamos o ponteiro `prim` para apontar para o novo nó, agora atualizamos o ponteiro `ult` da lista para apontar para o novo nó.

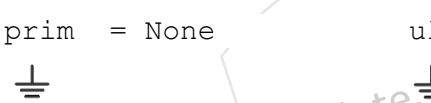
Outra simetria que pode ser observada é que, na inserção no início, criamos um nó passando como último argumento o endereço do atual primeiro nó da lista, enquanto na inserção no fim, criamos um nó passando como primeiro argumento o endereço do atual último nó da lista.

Passamos agora para a implementação da função `remover_inicio`, que tem como objetivo remover o primeiro elemento da lista duplamente encadeada.

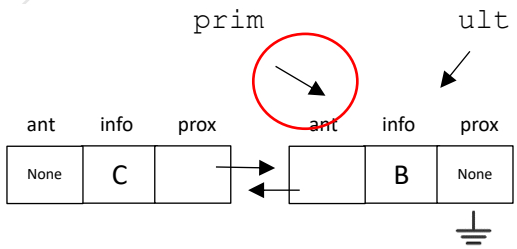
O primeiro passo da função é verificar se a lista contém apenas um elemento (`self.quant == 1`).



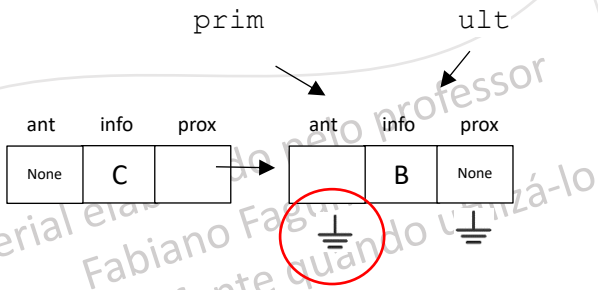
Nesse caso, é necessário remover o único elemento da lista e, por isso, fazemos com que os atributos `self.prim` e `self.ult` apontem para `None`. Isso significa que a lista ficará vazia após a remoção do primeiro elemento.



Se a lista contém mais de um elemento, aí precisaremos atualizar o atributo `self.prim` para que ele aponte para o segundo elemento da lista (`self.prim = self.prim.prox`).



O último passo é atualizar o atributo `ant` do novo primeiro elemento para apontar para `None` (`self.prim.ant = None`), já que o novo primeiro elemento não tem um nó anterior. Por fim, decrementamos o contador `quant` para indicar que um elemento foi removido da lista.



Assim, temos a seguinte implementação da função `remover_inicio`:

```
def remover_inicio(self):
     if self.quant == 1:
         self.prim = self.ult = None
     else:
         self.prim = self.prim.prox
         self.prim.ant = None
     self.quant-=1
```

Convém lembrar que após a execução da função o *garbage collector* é responsável por identificar objetos que não estão mais sendo utilizados pelo programa e desalocar o espaço de memória que eles ocupam. No caso da função `remover_inicio`, o nó que foi removido da lista se tornará um objeto sem referências e, portanto, elegível para ser desalocado da memória. Quando o *garbage collector* identifica que o objeto não está mais sendo referenciado por nenhum outro objeto do programa, ele é removido da memória, liberando espaço para que outros objetos possam ser alocados.

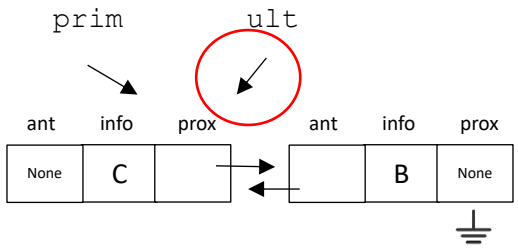
É importante ressaltar que o momento exato em que o *garbage collector* é acionado pode variar dependendo da implementação e configuração do ambiente em que o programa está sendo executado.

Vamos implementar agora a função `remover_fim`. Se você der um pulinho lá na lista dinâmica simplesmente encadeada você verá que, ali, tínhamos a necessidade de caminhar por todo a lista até chegarmos ao penúltimo elemento, que passaria a ser o novo último. Isto porque não havia referência a este penúltimo elemento da lista.

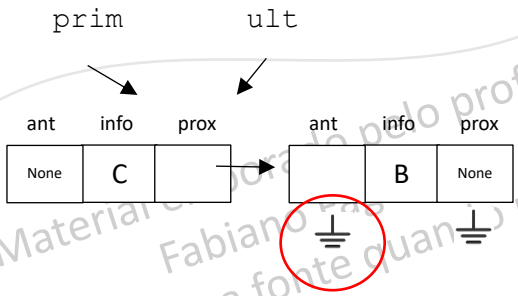
Agora, entretanto, é possível aproveitar a referência do atributo `ant` do último elemento para identificar o penúltimo elemento e, assim, realizar a remoção do último elemento de forma mais eficiente do que na lista dinâmica simplesmente encadeada. Dessa forma, não é necessário percorrer toda a lista até chegar ao penúltimo elemento para realizar a remoção do último elemento, o que é uma das vantagens da implementação da lista dinâmica duplamente encadeada. Isso torna a operação mais rápida e eficiente em termos de recursos computacionais, uma vez que o percurso por toda a lista pode ser evitado.

Então, partindo para a implementação da função `remover_fim`, inicialmente, e tal qual a função `remover_inicio`, é verificado se a lista contém apenas um elemento, caso positivo, ambos os atributos `prim` e `ult` são setados para `None` indicando que a lista está vazia.

Caso contrário, o atributo `ult` é atualizado para apontar para o penúltimo elemento da lista, ou seja, seu atributo ant.



Em seguida, o atributo `prox` do antigo último elemento é setado para `None`, indicando que este é o novo último elemento da lista.



Ao final, a quantidade de elementos na lista é decrementada em 1. Desta forma temos o seguinte código para a função `remover_fim`:

```
    def remover_fim(self):
        if self.quant == 1:
            self.prim = self.ult = None
        else:
            self.ult = self.ult.ant
```

```
            self.ult.prox = None
        self.quant-=1
```

É importante lembrar, então, que a implementação da lista duplamente encadeada permite a remoção do último elemento de forma eficiente, uma vez que o atributo `ant` do último elemento já aponta para o penúltimo elemento, evitando a necessidade de percorrer toda a lista.

E, mais uma vez, podemos ver a simetria entre funções semelhantes:

```
    def remover_inicio(self):
        if self.quant == 1:
            self.prim = self.ult = None
        else:
            self.prim = self.prim.prox
            self.prim.ant = None
        self.quant-=1
```

```
    def remover_fim(self):
        if self.quant == 1:
            self.prim = self.ult = None
        else:
            self.ult = self.ult.ant
            self.ult.prox = None
        self.quant-=1
```

Assim como na simetria entre as funções de inserção, também há uma simetria entre as funções de remoção, `remover_inicio` e `remover_fim`, no sentido de que os papéis dos ponteiros `prim` e `ult`, além dos atributos `ant` e `prox`, são invertidos.

Na remoção no início, o primeiro nó da lista é removido e, para isso, o ponteiro `prim` é atualizado para apontar para o próximo nó, e o ponteiro `ant` do novo primeiro nó é atualizado para `None`. Já na remoção no fim, o último nó da lista é removido e, para isso, o ponteiro `ult` é atualizado para apontar para o nó anterior, e o ponteiro `prox` do novo último nó é atualizado para `None`.

Podemos perceber também que onde havia a manipulação do ponteiro anterior (`ant`) na remoção no início, temos agora a manipulação do ponteiro próximo (`prox`) na remoção no fim. Além disso, onde antes atualizávamos o ponteiro `prim` para apontar para o próximo nó, agora atualizamos o ponteiro `ult` da lista para apontar para o nó anterior.

Outra simetria que pode ser observada é que, na remoção no início, verificamos se há apenas um elemento na lista, enquanto na remoção no fim, verificamos se há apenas um elemento na lista para atualizar os ponteiros `prim` e `ult` para `None`.

É claro que desejamos ver nossas funções sendo executadas e, para isso, precisamos implementar a função show, que não difere em nada da função já implementada para a `Ldse`.

```
    def show(self):
        aux = self.prim
        while aux!=None:
            print(aux.info,end=' ')
            aux = aux.prox
        print('\n')
```


Porém agora podemos imprimir também no sentido inverso. Então vamos implementar a função `show_inverso` e, para isso, vamos nos valer da simetria que já observamos. Para isso, podemos atribuir para a variável `aux` o valor de `self.ult`, que representa o último nó da lista.

Em seguida, percorremos a lista de forma inversa, imprimindo o valor de `aux.info` e atribuindo a `aux` o valor de `aux.ant`. Repetimos esse processo até que `aux` seja igual a `None`, ou seja, até que tenhamos percorrido toda a lista. Dessa forma, teremos a impressão dos elementos da lista na ordem inversa.

Desta forma, temos a seguinte função `show_inverso`:

```
    def show_inverso(self):
        aux = self.ult
        while aux!=None:
            print(aux.info,end=' ')
            aux = aux.ant
        print('\n')
```


As funções básicas da lista duplamente encadeada (`Ldde`), como `tamanho_atual`, que retorna o número atual de elementos na lista, a função `esta_vazia`, que verifica se a lista está vazia, e as funções `ver_primeiro` e `ver_ultimo`, que retornam os valores do primeiro e

do último elemento da lista, respectivamente, possuem a mesma implementação que a lista simplesmente encadeada (`Ldse`).

```
    def tamanho_atual(self):
        return self.quant

    def esta_vazia(self):
        return self.quant == 0

    def ver_primeiro(self):
        return self.prim.info

    def ver_ultimo(self):
        return self.ult.info
```


Apresento novamente as sugestões de funções para que você possa implementá-las e testar o seu conhecimento até o momento:

- `remover_elemento(self, valor)`: remove da lista o elemento com valor especificado, caso ele exista. Caso o elemento não esteja na lista, a função não faz nenhuma ação.
- `buscar(self, valor)`: busca o valor especificado na lista e retorna a posição do elemento na lista (posição 0, 1...). Se o valor não estiver na lista, a função retorna `None`.
- `inserir_apos(self, valor1, valor2)`: insere o `valor1` após o `valor2` na lista, caso o `valor2` exista. Considera-se que não há valores repetidos na lista. Se o `valor2` não estiver presente na lista, a função não faz nenhuma ação.
- `inserir_antes(self, valor1, valor2)`: insere o `valor1` antes do `valor2` na lista, caso o `valor2` exista. Considera-se que não há valores repetidos na lista. Se o `valor2` não estiver presente na lista, a função não faz nenhuma ação.

Por fim, a `Ldde` também possui vantagens e desvantagens quando comparada a outras estruturas de dados, como a Lista Estática e a Lista Dinâmica Simplesmente Encadeada.

Vantagens da `Ldde`:

• Flexibilidade no tamanho: a lista pode crescer ou diminuir de tamanho dinamicamente, conforme necessário, o que evita o desperdício de espaço de memória.

• Acesso bidirecional: a `Ldde` permite acesso aleatório aos elementos, tornando a busca e a recuperação de elementos mais eficientes em comparação com a `Ldse`.

Desvantagens da `Ldde`:

• Uso de memória adicional: cada nó na lista armazena dois ponteiros adicionais, um para o próximo elemento e outro para o elemento anterior, aumentando o consumo de memória em comparação com estruturas mais simples.

• Implementação mais complexa: a implementação da `Ldde` é mais complexa do que a da `Ldse`, o que pode tornar sua manutenção e depuração mais difíceis em alguns casos.

É importante destacar que tanto a `Ldse` quanto a `Ldde` podem apresentar variações que podem ser utilizadas de acordo com a necessidade do problema a ser resolvido. Duas variações, por exemplo, seriam a Lista Simplesmente Encadeada Circular e a Lista Duplamente Encadeada Circular, em que o último nó da lista aponta para o primeiro nó, formando um ciclo. Dessa forma, é possível percorrer a lista indefinidamente, facilitando algumas operações que envolvem ciclos.

Além disso, há também a possibilidade de se utilizar variações da `Ldse` e da `Ldde` em conjunto com outras estruturas de dados, como Árvores e Grafos, ampliando as possibilidades de soluções para problemas mais complexos. Em resumo, a escolha da estrutura de dados mais adequada para um determinado problema dependerá das características do problema e do objetivo da solução a ser implementada.

Material elaborado pelo professor
Fabiano Fagundes
Favor citar a fonte quando utilizá-lo