Lista Dinâmica

Listas dinâmicas são estruturas de dados que podem ser redimensionadas durante a execução do programa, permitindo adicionar ou remover elementos de forma eficiente. Elas são comumente implementadas usando *arrays* redimensionáveis (como o *ArrayList* em Java ou o *vector* em C++) ou listas encadeadas (como as listas simplesmente ou duplamente encadeadas).

Como vantagens em relação às listas estáticas, especialmente a lista estática sequencial, pode-se apresentar a flexibilidade no tamanho, pois listas dinâmicas podem crescer ou encolher conforme a necessidade, evitando desperdício de memória ou restrições no número de elementos. Tem-se também inserção e remoção eficientes, por serem encadeadas, pois oferecem inserções e remoções eficientes em qualquer posição, enquanto a lista estática sequencial é eficiente para inserção e remoção no final da lista.

Pode-se falar também em facilidade de uso, visto que listas dinâmicas em linguagens de alto nível geralmente oferecem funções prontas para uso que simplificam a manipulação da estrutura de dados. São justamente estas funções que estamos aprendendo a criar aqui.

Como desvantagens temos a sobrecarga de memória (o *overhead*), pois requerem armazenamento adicional para ponteiros de referência entre os elementos, e um acesso não tão rápido, visto que o tempo de acesso aos elementos em listas encadeadas é geralmente mais lento do que em *arrays*, pois não há o acesso indexado, o que requer percorrer a lista até o elemento desejado. Acrescente-se que sua implementação pode ser mais complexa do que listas estáticas.

Listas dinâmicas podem ser implementadas de várias formas, dependendo das necessidades específicas do problema e das características desejadas. Algumas das implementações mais comuns incluem:

- Listas simplesmente encadeadas: em que cada elemento da lista contém uma referência (ou ponteiro) para o próximo elemento na lista. Isso permite inserções e remoções eficientes em qualquer posição, mas exige um acesso sequencial aos elementos.
- Listas duplamente encadeadas: semelhantes às listas encadeadas simples, essas listas armazenam referências para os elementos anteriores e seguintes na lista, permitindo uma navegação bidirecional. Isso facilita a inserção e remoção de elementos em ambas as extremidades da lista, além de facilitar outras operações, como inverter a lista ou imprimi-la na ordem inversa. As listas duplamente encadeadas têm uma sobrecarga de memória maior devido ao armazenamento de dois ponteiros por elemento.

- Listas circulares: tanto as listas simplesmente encadeadas simples quanto as duplamente encadeadas podem ser configuradas como listas circulares, onde o último elemento da lista aponta para o primeiro elemento, criando um *loop*. Isso pode ser útil em situações onde a navegação circular é necessária, como em sistemas de buffer ou em simulações de problemas do tipo "Josephus".
- Listas com nó cabeça (ou *dummy node*): algumas implementações de listas dinâmicas incluem um nó fictício no início da lista, chamado de nó cabeça ou *dummy node*. Esse nó não armazena dados, mas facilita a manipulação da lista, simplificando a implementação de inserções e remoções e ajudando a evitar problemas com ponteiros nulos. As listas com nó cabeça podem ser encontradas em listas encadeadas simples, duplamente encadeadas e circulares.
- Listas com saltos (*skip lists*): estas são uma extensão das listas simplesmente encadeadas simples, onde os elementos são organizados em múltiplos níveis. Cada nível contém uma sublista que inclui uma fração dos elementos do nível inferior. Isso permite uma busca mais rápida, pois é possível "saltar" partes da lista ao procurar um elemento específico. As *skip lists* são uma alternativa eficiente às árvores balanceadas, como as AVL ou árvores rubro-negras, e são comumente usadas em aplicações como bancos de dados e sistemas de gerenciamento de arquivos.

Cada implementação de lista dinâmica apresenta suas próprias características, vantagens e desvantagens em termos de desempenho, uso de memória e complexidade de implementação. A escolha da implementação adequada depende das necessidades específicas do problema que se deseja resolver e dos requisitos de desempenho. Algumas considerações importantes ao escolher uma implementação de lista dinâmica incluem frequência de operações de inserção e remoção, forma de acesso aos elementos, uso de memória, complexidade de implementação, necessidades específicas do problema etc.

Ao avaliar esses fatores e entender as características de cada implementação de lista dinâmica, é possível tomar uma decisão informada sobre qual estrutura de dados é a mais adequada para o problema que se deseja resolver. As listas dinâmicas oferecem uma flexibilidade significativa em comparação com as listas estáticas e podem ser adaptadas para atender a uma ampla variedade de necessidades de desempenho e funcionalidade.

Aqui nós nos dedicaremos, para fins didáticos, às listas simplesmente e duplamente encadeadas, e às listas circulares.

Listas Dinâmicas Simplesmente Encadeadas

Listas dinâmicas simplesmente encadeadas, também conhecidas como listas encadeadas simples ou listas de ligação simples, são uma estrutura de dados linear na qual cada elemento, chamado de nó, contém dois campos: um para armazenar o valor do dado e outro para armazenar a referência (ou ponteiro) para o próximo elemento da lista.

Assim, aqui, tal qual a lista estática encadeada, tem-se uma estrutura de lista representada pelas classes `No` e `Ldse`. A classe `No` define os nós da lista, com os campos `info`, que armazena o valor, e `prox`, que armazena a referência para o próximo nó na lista. A classe `Ldse` representa a própria lista 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.

As operações básicas realizadas em listas simplesmente encadeadas são implementadas como métodos da classe `Ldse`:

`inserir_inicio`: insere um novo nó no início da lista.

`remover_inicio`: remove o primeiro nó da lista.

`remover_fim`: remove o último nó da lista.

`inserir_fim`: insere um novo nó no final da lista.

`estaVazia`: verifica se a lista está vazia.

`tamanho_atual`: retorna a quantidade atual de nós na lista.

`ver_primeiro`: retorna o valor do primeiro nó da lista.

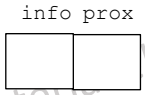
`ver_ultimo`: retorna o valor do último nó da lista.

`show`: percorre todos os elementos da lista, imprimindo seus valores.

Agora, vamos examinar como a construção da estrutura de nós citada anteriormente acontece. Primeiro, criaremos uma classe básica chamada `No`, que possui dois atributos: `info`, que armazena o valor, e `prox`, que armazena a referência para o próximo elemento na lista. Vocês perceberão que é equivalente ao que vimos na lista estática encadeada. O código a seguir ilustra essa definição.

```
class No:
    def __init__(self, valor, proximo):
        self.info = valor
        self.prox = proximo
```

E, da mesma forma que na Lee, podemos ver este nó, graficamente, como algo assim:



Então, igualmente, podemos ver o nó como sendo este retângulo com duas caixas, uma que representa o atributo `info` e outra para o atributo `prox`. Vamos criar agora a classe para esta lista, que chamaremos de `Ldse`. Você se surpreenderá como ela é muito mais simples que a estrutura da `Lee`.

```
class Ldse:


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

Sim, simples assim. A classe `Ldse` representa a lista dinâmica simplesmente encadeada. O método `init`, como sabemos, é o construtor da classe, que é chamado quando um objeto desta classe é criado.
Neste construtor, três atributos são inicializados:

- `self.prim`: armazena a referência para o primeiro nó da lista. É inicializado com `None`, indicando que a lista está vazia no momento da criação.
- `self.ult`: a referência para o último nó da lista. Assim como self.prim, é inicializado com `None`, pois a lista está vazia no momento da criação.
- `self.quant`: Este atributo armazena a quantidade de nós presentes na lista. É inicializado com 0, já que a lista está vazia no momento da criação.

Ao criar uma instância da classe `Ldse`, temos uma lista dinâmica simplesmente encadeada vazia, com o primeiro e último elementos sendo `None`, e a quantidade de elementos na lista sendo 0.

O valor `None` representa a ausência de um nó ou o fim da lista. Quando `self.prim` e `self.ult` são inicializados com `None`, isso indica que a lista está vazia, ou seja, não há nenhum nó na lista.

Quando um nó é inserido na lista, a referência `prox` do último nó é ajustada para apontar para `None`. Isso significa que não há mais elementos após o último nó na lista.

Utilizar `None` para representar o fim da lista ou a ausência de um nó é uma prática comum em Python e em outras linguagens de programação. Isso permite que se saiba quando chegou ao fim da lista ao percorrê-la, por exemplo, em uma operação de busca ou ao exibir todos os elementos da lista. Além disso, também é uma possibilidade para verificar se a lista está vazia ao comparar o primeiro e/ou o último elemento com `None`.

No caso das listas dinâmicas simplesmente encadeadas, não há necessidade do atributo `tam_maximo`, pois a lista tem a capacidade de expandir ou contrair dinamicamente durante a execução do programa, sem estar limitada pelo tamanho de um vetor pré-alocado.

A implementação de uma lista dinâmica simplesmente encadeada é simplificada, pois não é necessário buscar uma posição vazia em um vetor ao inserir um novo elemento ou devolver essa posição ao remover um elemento da lista. Em vez disso, a alocação de espaço na memória ocorre pela criação de uma instância da classe `No`, que armazena o valor e a referência para o próximo elemento.

Quando um nó é removido da lista, o espaço de memória ocupado por ele é liberado automaticamente pelo mecanismo de gerenciamento de memória da linguagem de programação, como o coletor de lixo (*garbage collector*) no caso do Python. Isso permite que a lista seja gerenciada de maneira eficiente e simplifica a implementação, proporcionando maior flexibilidade em comparação com as listas estáticas.

Vamos implementar a função inserir_inicio, para começarmos a compreender como se dá a implementação nesta lista. Mas antes, vamos acompanhar como se dá, logicamente, este processo de inserção no início.

Inicialmente temos a lista vazia, da seguinte forma:

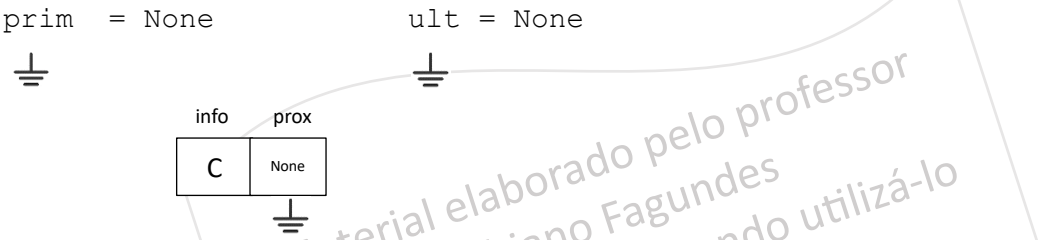
```
prim  = None          ult = None
```

Relembrando, o símbolo de aterramento indica que os atributos `prim` e `ult` estão com valor `None`, ou seja, não possuem referência para nenhuma posição no vetor. Isso significa que a lista está vazia, sem nenhum elemento armazenado.

Ao inserir um elemento no início da lista, precisamos de espaço para armazená-lo; portanto, devemos criar um nó, ou seja, criar a estrutura com os dois campos mencionados anteriormente – no desenho, aquela estrutura com as duas caixinhas. Vale ressaltar que a estratégia de desenhar a estrutura lógica da lista ajudará no processo de criação do código.

Para criar essa estrutura no código, utilizamos o construtor da classe `No`, passando como argumentos o valor a ser inserido e a referência para o próximo elemento na lista. Dessa forma, podemos simplificar a implementação ao associar a criação das "caixinhas" com a chamada do construtor da classe No sempre que precisarmos criar um novo nó na lista.

Portanto, por exemplo, ao inserir o valor 'C' em uma lista vazia, começamos criando o nó onde ele será armazenado no campo `info`. Em seguida, precisamos considerar como preencher o atributo `prox`. Como a lista está vazia e esse elemento será o único, não haverá nenhum elemento após ele. Dessa forma, atribuímos o valor `None` ao campo `prox` para indicar essa condição.



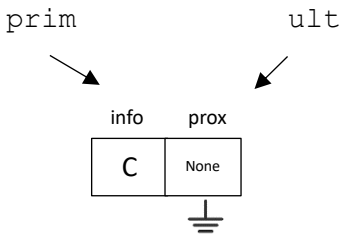
Assim, haverá a seguinte chamada do construtor:

```
No(valor, None)
```

Podemos observar que, estando a lista vazia, ao inserir o primeiro elemento, precisamos fazer com que os atributos `prim` e `ult` da lista apontem para ele. Para isso, associamos a chamada do construtor à atribuição desse nó aos atributos `prim` e `ult` da seguinte forma:

```
self.prim = self. ult = No(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.

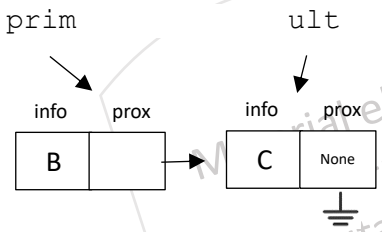


Podemos observar que há uma condição envolvida nesse processo, pois estamos levando em conta que a lista está vazia. Assim, temos o seguinte trecho de código para lidar com essa situação:

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

Esse trecho verifica se a quantidade de elementos na lista (`quant`) é igual a 0, ou seja, se a lista está vazia. Se for o caso, ele cria um novo nó com o valor desejado e faz com que os atributos `prim` e `ult` apontem para esse novo nó. Porém, e se ela não estiver vazia?

A visualização da estrutura da lista nos auxilia a compreender o processo. Por exemplo, se quisermos inserir o valor 'B' no início da lista, para qual nó o novo elemento deve apontar? Em outras palavras, se inserimos um novo elemento antes do primeiro elemento atual, esse novo elemento deve apontar, logicamente, para o nó que, até então, estava na posição de primeiro elemento. E quem tem este endereço é justamente o atributo `prim`.



Ao seguir o raciocínio e observar o desenho da estrutura, percebemos que, ao inserir um novo elemento no início da lista, precisamos criar um novo nó e fazer as devidas conexões. Ao desenhar a "caixa" que representa o novo nó, utilizamos o construtor da classe No, passando dois argumentos: o valor a ser armazenado e o endereço do próximo elemento. Como estamos inserindo o novo nó no início da lista, o próximo elemento será o atual primeiro elemento (representado pelo atributo `prim`).

A chamada do construtor ficaria assim:

```
No(valor, self.prim)
```

Em seguida, atualizamos o atributo `prim` da lista para que ele aponte para o novo nó, que passa a ser o primeiro elemento da lista. Essa atualização é feita atribuindo o resultado da chamada do construtor ao atributo `prim`, conforme mostrado abaixo:

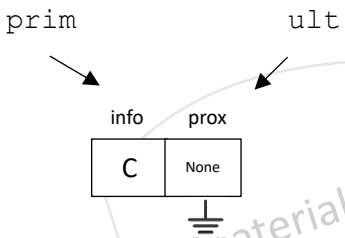
```
self.prim = No(valor, self.prim)
```

Dessa forma, o novo nó é inserido no início da lista, e o atributo `prim` passa a apontar para ele, garantindo a correta atualização da estrutura da lista.

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

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

A função `inserir_fim` segue uma lógica similar à função que analisamos anteriormente. Se a lista está vazia e estamos inserindo o primeiro elemento, precisamos fazer com que os atributos `prim` e `ult` da lista apontem para esse novo elemento.



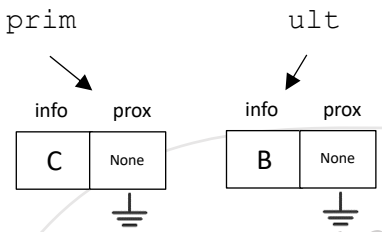
Para conseguir isso, associamos a chamada do construtor ao novo nó criado e atualizamos os atributos `prim` e `ult` da lista simultaneamente:

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

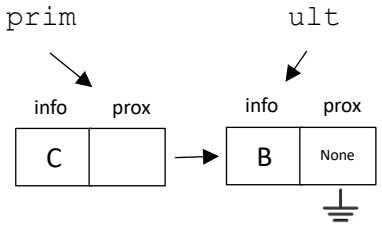
Se a lista já contiver elementos, devemos inserir o novo nó no final e atualizar a referência do último elemento atual (representado pelo atributo `ult`) para apontar para o novo nó. Para isso, chamamos o construtor da classe `No`, passando o valor a ser armazenado e `None` como endereço do próximo elemento, já que o novo nó será o último da lista e, logicamente, não haverá outro elemento após ele:

```
No(valor, None)
```

Para ilustrar, vamos realizar a inserção do valor B no fim da lista. Em uma representação visual da estrutura lógica da lista neste momento, podemos observar o novo nó contendo o valor B que foi adicionado, no entanto, ele ainda não está conectado à lista:



Assim, temos que ligar este elemento à lista. Como ele estará no fim da lista, ele estará após o atual último elemento, que deverá apontar para ele, ou seja, seu atributo `prox` deverá apontar para o novo nó construído. De igual forma, temos que atualizar o atributo `ult` da lista, visto que este elemento passará a ser o novo último elemento da lista.



Com base no desenho, percebemos que precisamos atualizar a referência do último elemento atual (`self.ult.prox`) para apontar para o novo nó e, por fim, atualizar também o atributo `ult` para apontar para este novo nó:

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

Dessa forma, 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.

A ordem de atribuição nesta linha de código é importante e não pode ser alterada porque ela realiza duas ações principais em uma única linha, garantindo que as referências sejam atualizadas corretamente ao inserir um novo elemento no fim da lista.

Neste caso, a ordem da atribuição ocorre da esquerda para a direita:

1. `self.ult.prox = ...`: primeiro, atualizamos o atributo `prox` do antigo último elemento (antes da inserção) para apontar para o novo nó que foi criado.
2. `self.ult = No(valor, None)`: em seguida, atribuímos o nó recém-criado ao atributo `ult` da lista, para indicar que o novo nó é agora o último elemento da lista.

A ordem de atribuição é crucial aqui porque primeiro precisamos atualizar o atributo `prox` do antigo último elemento para apontar para o novo nó para somente depois atribuí-lo ao atributo `ult`. Inverter a ordem de atribuição resultaria em referências incorretas e potencialmente na perda de elementos da lista, já que o antigo último elemento não estaria conectado ao novo último elemento.

Por fim, teríamos assim o código da função `inserir_fim`:

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

Antes de abordarmos as funções de remoção, é importante implementar a função `show` para podermos testar e visualizar o funcionamento das funções de remoção até aqui implementadas.

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

Essa função percorre a lista encadeada do início ao fim, imprimindo os elementos em ordem, o que facilita a visualização do conteúdo da lista durante a execução do programa.

Inicialmente, a variável `aux` é inicializada com o valor do atributo `prim` da lista, ou seja, ela começa apontando para o primeiro elemento da lista. Enquanto `aux` não for `None` (o que indicaria o fim da lista), o *loop* indicado pelo `while` irá continuar.

Dentro do *loop*, o valor armazenado no nó atual (representado por `aux.info`) é impresso, seguido de um espaço (para separar os elementos visualmente), indicado pelo parâmetro `end=' '`, que garante também que os elementos sejam impressos na mesma linha. Ainda dentro do *loop* atualizamos a variável `aux` para apontar para o próximo nó na lista (obtido pelo atributo `prox` do nó atual). Isso permite avançar para o próximo elemento na lista.

Podemos modificar a função `show` para utilizar um *loop* `for` e o atributo `quant` da lista (que armazena a quantidade de elementos) como critério de parada. Veja a versão atualizada do método:

```
def show(self):
    aux = self.prim
    for i in range(self.quant):
        print(aux.info, end=' ')
        aux = aux.prox
    print('\n')
```

Nesta versão, o *loop* for é executado `self.quant` vezes, ou seja, a quantidade de elementos na lista. A variável `i` é utilizada como uma variável temporária que não será usada no *loop*. A cada iteração do *loop*, a função imprime o valor do nó atual (representado por `aux.info`) e atualiza a variável `aux` para apontar para o próximo nó da lista (obtido pelo atributo `prox` do nó atual).

Agora já podemos fazer um programa teste básico para acompanhar nossas funções:

```
import Ldse

lista = Ldse.Ldse()

# Inserir elementos no início
lista.inserir_inicio('C')
lista.inserir_inicio('B')
lista.inserir_inicio('A')

# Inserir elementos no fim
lista.inserir_fim('D')
lista.inserir_fim('E')
```

```
# Mostrar a lista
lista.show()
```

O código acima importa a classe `Ldse` (que foi definida anteriormente) e realiza as seguintes operações:

Cria uma instância da classe `Ldse` chamada lista.

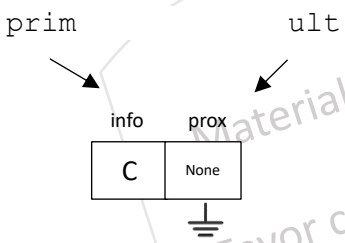
- Insere elementos no início da lista utilizando a função `inserir_inicio`. A ordem dos elementos inseridos é 'C', 'B' e 'A'.
- Insere elementos no fim da lista utilizando a função `inserir_fim`. A ordem dos elementos inseridos é 'D' e 'E'.
- Mostra a lista utilizando a função `show`.

Após a execução das operações acima, a lista terá os seguintes elementos, na ordem mostrada: A, B, C, D, E e a saída será:

```
A B C D E
```

Vamos agora começar a implementar as funções de remoção. Primeiro, vamos remover o primeiro elemento da lista com a função `remover_inicio`.

Começamos com uma situação especial, que é quando a lista tem somente um elemento:



Nesse caso, após a remoção, a lista ficará vazia e os atributos `prim` e `ult` devem ser atualizados para `None`. Veja o trecho de código que lida com essa condição:

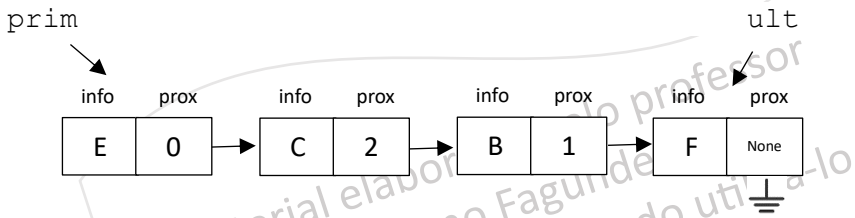
```
if self.quant == 1:

    self.prim = self.ult = None
```

Aí voltamos a ter uma lista novamente vazia, com a seguinte configuração:

```
prim  = None          ult = None
```

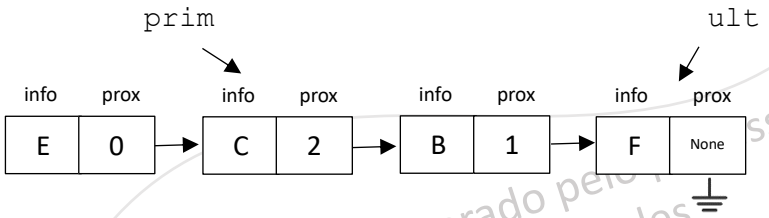
No entanto, quando a lista possui mais de um elemento, devemos compreender que remover o primeiro elemento implica em transformar o segundo elemento, para o qual o primeiro elemento aponta, no novo primeiro elemento da lista. A representação da estrutura lógica permite visualizar essa situação:



Como pode-se observar, basta atualizar o atributo `prim` para que ele aponte para o segundo elemento da lista.

```
self.prim = self.prim.prox
```

Nesse caso, o atributo `prox` do primeiro elemento ('E') contém a referência para o segundo elemento ('C'). Atualizando o atributo `prim` para apontar para 'C', removemos efetivamente o elemento 'E' do início da lista.



Aqui vemos como se dá a atuação do *garbage collector* ao removermos o primeiro elemento da lista. Como pode-se ver, estamos eliminando a referência a esse nó, o que o torna inacessível a partir da nossa estrutura de dados. Sem nenhuma referência apontando para ele, o objeto se torna elegível para coleta de lixo.

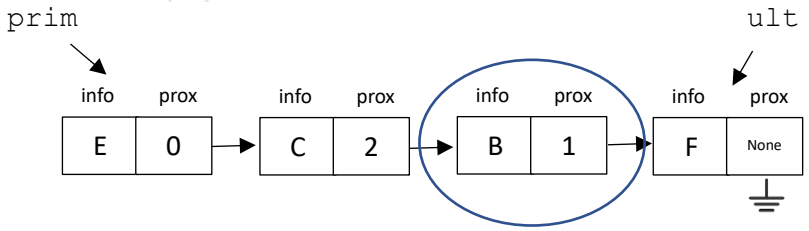
Como já citado anteriormente, o *garbage collector* é um mecanismo do Python que gerencia automaticamente a memória, identificando objetos que não estão mais sendo utilizados e liberando a memória alocada para eles. Quando o *garbage collector* identifica que não há mais referências apontando para o nó removido, ele entra em ação e limpa a memória alocada para esse objeto, garantindo assim que a memória seja gerenciada de forma eficiente.

Agora vamos implementar a função `remover_fim`. Para implementar esta função é importante considerar alguns aspectos da estrutura da lista e do processo de remoção.

Vamos começar pelos casos especiais: como estabelecemos lá no início, para listas vazias a função não deve fazer nada, mas, opcionalmente, pode retornar uma mensagem de erro. Agora, para listas com apenas um elemento, basta definir os atributos `prim` e `ult` como `None` e atualizar a quantidade de elementos.

```
if self.quant == 1:
    self.prim = self.ult = None
self.quant -= 1
```

Uma situação especialíssima é a necessidade de acessar o penúltimo elemento. Para remover o último elemento da lista, precisamos encontrar o penúltimo elemento primeiro pois ele é quem passará a ser o novo último elemento. E, se você observar a lista, verá que não há nenhum atributo que guarda o seu endereço.



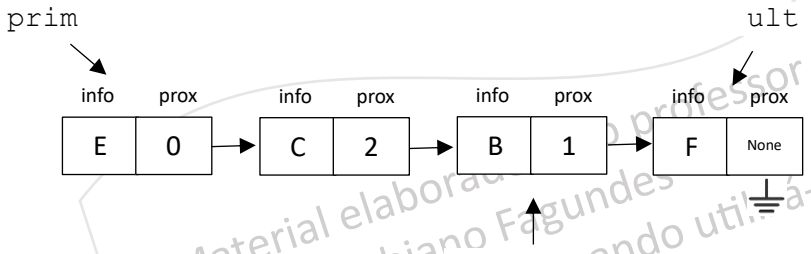
Portanto, precisamos percorrer a lista a partir do primeiro elemento até alcançar o penúltimo. Isso pode ser realizado utilizando um ponteiro auxiliar que percorre a lista, identificando o elemento cujo atributo `prox` aponta para o último elemento (`self.ult`). Outra alternativa é usar um *loop* `for`, definindo como intervalo o valor de `quant` menos 2.

```
    aux = self.prim
    while aux.prox != self.ult:
        aux = aux.prox
```

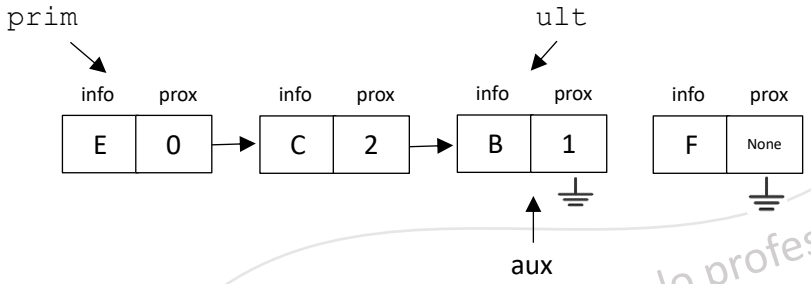
Ou:

```
    aux = self.prim
    for i in range(self.quant-2):
        aux = aux.prox
```

Ao optar pelo uso de um ponteiro auxiliar, o processo de localizar o penúltimo elemento na lista é eficiente e facilita a manipulação das referências. Por outro lado, a abordagem com o *loop* `for` pode ser mais intuitiva para alguns, já que utiliza um contador baseado na quantidade de elementos da lista. Escolha a abordagem que melhor se adapta ao seu entendimento e estilo de programação.



Após encontrar o penúltimo elemento precisamos atualizar os ponteiros da lista. O atributo `prox` do penúltimo elemento deve ser definido como `None`, pois ele passará a ser o último elemento da lista após a remoção. Além disso, devemos atualizar o atributo `ult` da lista para passar a apontar para o penúltimo elemento.



Como realizado pelo código abaixo:

```
    aux.prox = None
    self.ult = aux
```

Por fim, atualizamos a quantidade de elementos, lembrando que o *garbage collector* do Python cuidará da liberação de memória do nó removido, uma vez que não haverá mais referências a ele na lista.

Assim, temos nossa função `remover_fim` com as duas formas de implementação do *loop* para percorrer a lista até o penúltimo elemento:

```
    # versão com verificação do aux.prox
    def remover_fim(self):
        if self.quant == 1:
            self.prim = self.ult = None
        else:
            aux = self.prim
            while aux.prox != self.ult:
                aux = aux.prox
            aux.prox = None
            self.ult = aux
        self.quant -= 1

    # versão com range
    def remover_fim(self):
        if self.quant == 1:
            self.prim = self.ult = None
        else:
            aux = self.prim
            for i in range(self.quant-2):
                aux = aux.prox
            aux.prox = None
            self.ult = aux
        self.quant -= 1
```


Por fim, nesta relação de funções básicas, temos ainda as funções que seguem, que não diferem muito das implementadas para a Lista Estática Encadeada. A função `tamanho_atual` retorna a quantidade atual de elementos na lista, e a função `esta_vazia` verifica se a lista está vazia. Já as funções `ver_primeiro` e `ver_ultimo` retornam o valor do primeiro e último elemento da lista, respectivamente.

```
    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
```

Vamos agora ver a função `remover_irmaos(self, valor)` para a Lista Dinâmica Simplesmente Encadeada pois, como já vimos para a Lista Estática Encadeada, seu entendimento é um bom exercício de compreensão sobre o assunto. As observações mencionadas abaixo também foram feitas lá, assim, vamos analisar cada situação e adaptar a função para `Ldse`:

1. A lista está vazia: se a lista estiver vazia, a função não deve realizar nenhuma operação, pois não há elementos para serem removidos.
2. A lista possui somente um elemento: se a lista possui apenas um elemento, a função não deve realizar nenhuma operação, pois não existem irmãos para serem removidos.
3. O valor em questão é o primeiro elemento da lista: se o valor em questão é o primeiro elemento da lista, não há irmão anterior a ser removido, portanto a função deve remover somente o irmão posterior. É importante ressaltar que nessa situação não existe a possibilidade de não haver irmão posterior, já que a situação em que não há irmão anterior e posterior foi tratada na situação 2.
4. O valor em questão é o último elemento da lista: se o valor em questão é o último elemento da lista, não há irmão posterior a ser removido, portanto a função deve remover somente o irmão anterior. De maneira semelhante à situação anterior, não há a possibilidade de não existir o irmão anterior, já que essa situação foi tratada na situação 2.
5. O valor em análise é o segundo elemento da lista: nessa situação, ao remover o irmão anterior (que até então era o primeiro elemento da lista), o elemento em questão se torna o novo primeiro elemento da lista.
6. O valor em análise é o penúltimo elemento da lista: nessa situação, ao remover o irmão posterior (que até então era o último elemento da lista), o elemento em questão se torna o novo último elemento da lista.

Considerando estes casos, temos a implementação da função `remover_irmaos` a seguir:

```
def remover_irmaos(self, valor):
    if self.quant != 1 and self.quant != 0:
```

```
        anterior_do_anterior = None
        anterior = None
        atual = self.prim
        while atual != None and atual.info != valor:
            # andando pela lista até achar valor ou chegar ao fim da lista
            anterior_do_anterior = anterior
            anterior = atual
            atual = atual.prox
        if atual != None and atual.info == valor:
            # ou seja, achou valor na lista e ele está no nó atual
            if anterior != None and anterior == self.prim:
                # age se o irmão anterior for o primeiro da lista
                self.remover_inicio()
            else:
                if anterior_do_anterior != None:
                    # caso o irmão anterior não seja o primeiro da lista
                    anterior_do_anterior.prox = atual
                    anterior = None
                    self.quant -= 1
            if atual.prox != None and atual.prox == self.ult:
                # age se o irmão posterior for o último da lista
                self.remover_fim()
            else:
                if atual.prox != None:
                    # caso o irmão posterior não seja o último da lista
                    proximo = atual.prox
                    atual.prox = proximo.prox
                    proximo = None
                    self.quant -= 1
```


Nesta adaptação do que havia sido implementado para a Lista Estática Encadeada, substituímos os índices e o vetor de nós pelos próprios nós, conforme o funcionamento da Lista Dinâmica Simplesmente Encadeada. Assim, o código se baseia nas referências aos nós diretamente, sem a necessidade de acessar um vetor intermediário.

É importante observar que a ordem de verificação na condição `while atual != None and atual.info != valor` busca garantir a segurança (evitando erros de acesso a atributos de objetos nulos) e a eficiência (encerrando a busca assim que o valor desejado for encontrado) do código. Essa ordem de verificação assegura que:

- O código não tente acessar um atributo de um objeto `None`, o que causaria um erro. A verificação de `atual != None` é feita primeiro para garantir que o objeto atual não seja

nulo antes de tentar acessar o atributo `info`. Se atual fosse `None` e tentássemos acessar o atributo `info` diretamente, ocorreria um `AttributeError`. Portanto, essa verificação impede que erros sejam gerados ao acessar atributos de objetos nulos.

- O *loop* continue apenas enquanto o valor procurado não for encontrado. A segunda parte da condição, `atual.info != valor`, verifica se o valor do atributo `info` do objeto atual é diferente do valor procurado. Se o valor for encontrado, a condição se torna falsa e o *loop* `while` é encerrado, interrompendo a busca na lista. Isso garante que o loop pare assim que o valor desejado for encontrado, evitando a necessidade de percorrer toda a lista desnecessariamente e, assim, melhorando a eficiência do código. Além disso, essa forma de implementação remove a necessidade de utilizar um `break`, pois a condição do `while` controla a saída do *loop* de maneira adequada e direta.

Trago mais uma vez as sugestões de funções que indiquei nas listas implementadas anteriormente para você implementar e testar seu aprendizado até o momento:

- `remover_elemento(self, valor)`: remove o elemento com valor especificado da lista, 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.

Resumindo, a Lista Dinâmica Simplesmente Encadeada é uma estrutura de dados que possui algumas vantagens e desvantagens quando comparada a outras estruturas, como a Lista Estática e a Lista Dinâmica Duplamente Encadeada.

Vantagens da LDSE:

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

- Inserção e remoção eficientes: a inserção e remoção de elementos na LDSE podem ser realizadas em tempo constante, desde que se conheça o nó anterior ao local desejado para a operação.

Desvantagens da LDSE:

- Acesso sequencial: a LDSE não permite acesso aleatório aos elementos, tornando a busca e a recuperação de elementos menos eficientes em comparação com outras estruturas de dados.
- Uso de memória adicional: cada nó na LDSE armazena um ponteiro adicional para o próximo elemento, aumentando o consumo de memória em comparação com estruturas mais simples.
- Navegação unidirecional: a LDSE permite apenas a navegação para frente na lista, o que pode ser limitante em algumas situações.

A Lista Dinâmica Duplamente Encadeada é uma estrutura de dados que busca solucionar algumas desvantagens da Ldse. A principal melhoria oferecida pela Ldde é a capacidade de navegação bidirecional, graças aos ponteiros que apontam para os elementos anteriores e posteriores na lista. Isso facilita a manipulação e a navegação na lista, permitindo que as operações sejam realizadas tanto no início quanto no final da lista de maneira eficiente.