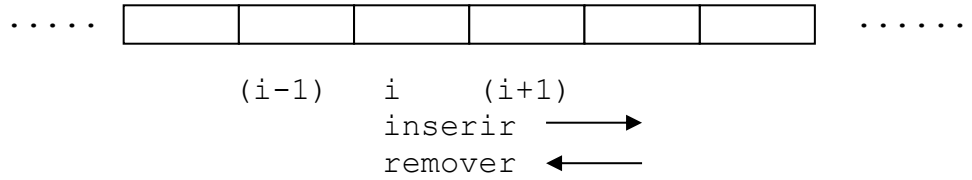
Alocação Dinâmica – Listas Ligadas

1. Listas sequenciais versus listas ligadas

Lista sequencial

Uma lista sequencial é um conjunto de elementos contíguos na memória. Uma lista em Python é o melhor exemplo de lista sequencial (sets e dicts também). O elemento `i` é precedido pelo elemento `i-1` e seguido pelo elemento `i+1`. Como os elementos são contíguos, para se acessar um elemento só é necessário conhecer o seu índice, pois o índice indica também o deslocamento a partir do início.

A vantagem é o acesso direto a qualquer dos elementos da lista. A desvantagem é que para inserir elementos, tem que haver alguma alocação dinâmica de memória ou mesmo deslocamento de elementos dependendo do sistema usado. O mesmo ocorre quando se remove elementos. Isso é necessário para não alterar o acesso através do índice.

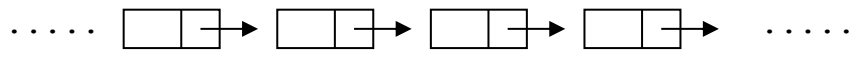


As operações de alocação dinâmica de memória e deslocamento de dados para abrir ou diminuir espaço entre os elementos não são eficientes. No caso do Python, as operações append(), del(), pop(), extend(), etc., bem como as operações com sub- listas, embora bastante úteis e simples de usar, não são operações eficientes justamente devido à necessidade de deslocamento dos elementos ou mesmo devido a necessidade de uso adicional de memória.

Listas Ligada

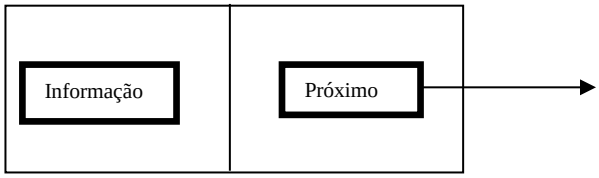
Uma lista ligada é um conjunto de elementos onde cada elemento indica qual o próximo. Não existe contiguidade física entre os elementos. Elementos vizinhos podem estar em posições físicas de memória separadas.

A vantagem é a flexibilidade na inserção e remoção de elementos. A desvantagem é que não temos acesso direto aos elementos. Não existe algo equivalente ao índice para se acessar diretamente o elemento. É preciso percorrer a lista até chegar ao elemento desejado.

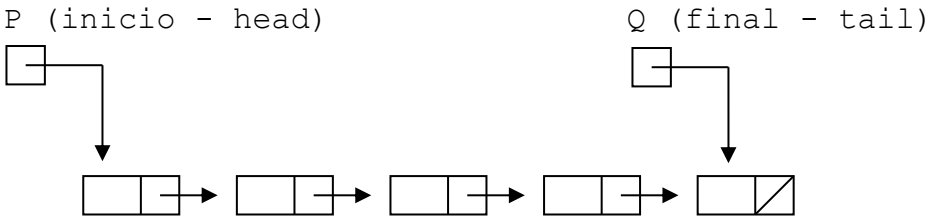


2. Listas ligadas - definições

É um conjunto de itens, onde cada item ou elemento contém uma informação e um ponteiro para o próximo item.

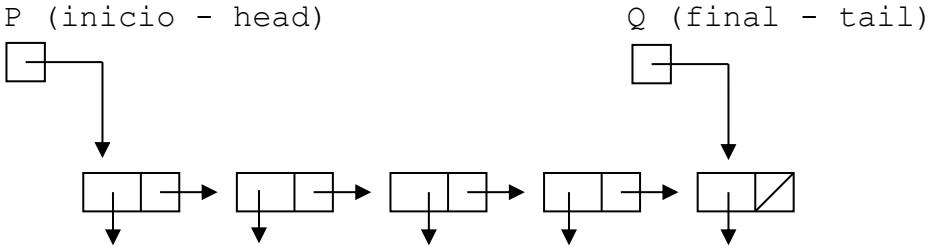


Possui também um ponteiro para o seu início. O próximo do último elemento tem que ser um valor especial (`None` em Python) para indicar que é o fim da lista ligada. Um ponteiro para o último elemento pode ser útil em alguns casos.



O campo de informação pode conter qualquer tipo ou conjunto de tipos de elementos. O ponteiro para o próximo item é um ponteiro para um item do mesmo tipo.

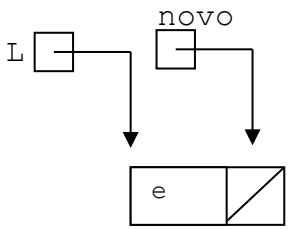
Em se tratando de Python, o campo de informação de cada nó, contém na verdade um indicador ou ponteiro para o objeto. Por simplificação podemos supor que a informação está mesmo presente neste campo.



3. Operações básicas com listas ligadas

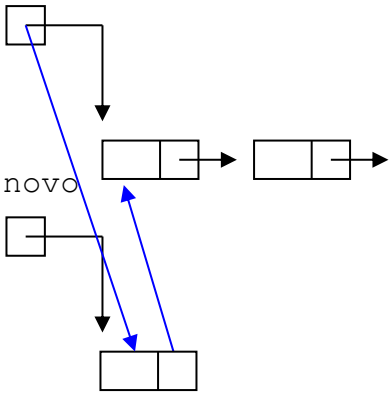
Vejamos agora, algumas operações com listas ligadas, descrevendo os algoritmos em uma linguagem livre (não é Python ainda). Vamos manter também a quantidade de elementos da lista:

```
# criar uma lista com um primeiro elemento com valor e
novo = Node(e)
novo.prox = None
L.inicio = novo
L.tamanho = 1
```



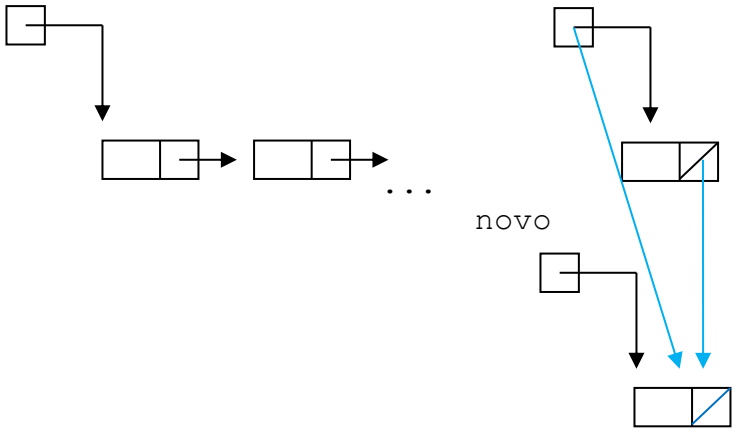

```
# inserir elemento no inicio de lista L
novo = Node(e)
novo.prox = L.inicio
L.inicio = novo
L.tamanho = L.tamanho + 1
```

L.inicio

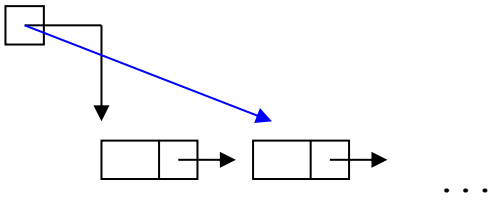



```
# inserir elemento no final da lista
novo = Node(e)
novo.prox = None
L.final.prox = novo
L.final = novo
L.tamanho = L.tamanho + 1
```

L.inicio L.final



```
# remover elemento do início da lista L
if L.inicio is None:
     # erro - lista vazia
L.inicio = L.inicio.prox
L.tamanho - L.tamanho - 1

L.inicio
```



Para remover elemento do final da lista a tarefa não é tão fácil. Precisamos saber quem é o penúltimo. Isso só pode ser feito percorrendo a lista desde o início. O mesmo ocorre para se remover um elemento do meio da lista ligada. É necessário saber quem é o anterior. A solução para esse problema são as listas duplamente ligadas que veremos ao final.

4. Uma classe Lista Ligada

A classe a ser construída se utiliza de elementos da classe _Node que é um elemento ou nó da lista ligada (Vamos abreviar lista ligada por LL). Essa classe _Node será uma classe interna porque não precisa ser conhecida ou referenciada externamente à classe.

```
class _Node:

    '''Classe interna para armazenar um nó.'''

    def __init__ (self, info, prox):
        # iniciar os campos de um nó
        self._info = info  # contéudo ou informação
        self._prox = prox  # referência ao próximo
```

As funções ou métodos da classe:

Construtor da classe: __init__()
Devolve o tamanho da LL: __len__()
Devolve True se LL vazia: is_empty()
Devolve o info do primeiro elemento sem removê-lo: first()
Adiciona elemento no início: adiciona()
Busca elemento com um certo info: busca()
Busca elemento com um certo info, devolvendo também o anterior: busca_ant ()
Adiciona elemento à frente: adiciona_afrente()
Remove elemento a frente: remove_afrente()
Criar uma LL a partir de uma lista: CriaLL()

Imprimir a parte info da LL: ImprimaLL()

Vamos dar o nome de ListaLigadaSimples a esta classe.

```
class ListaLigadaSimples:

    ''' implementa uma LL simples. '''

    # classe _Node - interna
    class _Node:

        __slots__ = '_info', '_prox'

        def __init__ (self, info, prox):
            # inicia os campos
            self._info = info
            self._prox = prox

    # métodos de LL

    def __init__ (self):
        ''' cria uma fila vazia.'''
        self._inicio = None # vazia
        self._tamanho = 0   # tamanho da LL

    def __len__(self):
        ''' retorna o tamanho da LL.'''
        return self._tamanho

    def is_empty(self):
        ''' retorna True se LL vazia'''
        return self._tamanho == 0

    def first(self):
        ''' retorna o campo info da LL sem remover.
            sinaliza exceção se LL vazia.'''
        if self.is_empty():
            raise Empty("Lista Ligada Vazia")
        return self._inicio._info # elemento inicial da fila

    def adiciona(self, e):
        ''' adiciona elemento no inicio da LL.'''
        # novo nó referencia o inicio da LL
        novo = self._Node(e, self._inicio)
        # novo nó será o inicio da LL
        self._inicio = novo
        self._tamanho += 1
```

```
    def adiciona_afrente(self, p, e):
        ''' adiciona elemento a frente de p.'''
        # novo nó referencia o mesmo que p
        novo = self._Node(e, p._prox)
        # p referencia o novo nó
        p._prox = novo
        self._tamanho += 1

    def remove_afrente(self, p):
        ''' remove elemento a frente de p.'''
        # caso particular - p não pode ser o último
        if p._prox is None:
            raise Empty("Nó inexistente")
        # p referencia o mesmo que o próximo
        p._prox = p._prox._prox
        self._tamanho -= 1

    def busca(self, e):
        ''' procura elemento com info = e.
            devolve referência para esse elemento ou None.'''
        # p percorre a lista
        p = self._inicio
        while p is not None:
            if p._info == e: return p # achou
            p = p._prox # vai para o próximo
        # se chegou aqui é porque não achou
        return None

    def busca_ant(self, e):
        ''' procura elemento com info = e.
            devolve uma dupla de referências (p_ant, p).'''
        # p percorre a lista e pant referencia o anterior
        pant, p = None, self._inicio
        while p is not None:
            if p._info == e:
                return pant, p
            pant, p = p, p._prox
        return pant, p

    def CriaLL(self, vet):
        ''' cria uma LL com elementos do vetor vet.'''
        for k in range(len(vet) - 1, -1, -1):
            self.adiciona(vet[k])

    def ImprimeLL(self):
        ''' só para teste.'''
```

```
        p = self._inicio
        k = 1
        print("Imprimindo a lista ligada:")
        while p is not None:
            print(k, " - ", p._info)
            k = k + 1
            p = p._prox

# Cria lista ligada
cores = ["vermelho", "preto", "azul", "amarelo", "verde",
"branco"]
LLC = ListaLigadaSimples()
LLC.CriaLL(cores)
LLC.ImprimeLL()

# Verifica se algumas cores estão na lista
while True:
    cor = input("\nEntre com o nome da cor:")
    if cor == 'fim': break
    if LLC.busca(cor) is None:
        print("A cor", cor, "não está na lista ligada")
    else:
        print("Encontrada a cor", cor, "na lista ligada")

print("\nAdicionando laranja e violeta depois e antes da azul")

# Adicionar cor laranja a frente do azul
anterior, atual = LLC.busca_ant("azul")
if atual is None:
    print("Cor azul não está na lista")
else:
    LLC.adiciona_afrente(atual, "laranja")

# Adicionar cor violeta antes da azul
if anterior is None:
    print("Cor azul não tem anterior")
else:
    LLC.adiciona_afrente(anterior, "violeta")

LLC.ImprimeLL()

print("\nRemovendo as cores amarelo e lilás da lista")

# Remove cor amarelo
anterior, atual = LLC.busca_ant("amarelo")
if atual is None or anterior is None:
    print("Cor amarelo não está na lista ou não tem anterior")
```

```
else:
    LLC.remove_afrente(anterior)

# Remove cor lilas
anterior, atual = LLC.busca_ant("lilas")
if atual is None or anterior is None:
    print("Cor lilas não está na lista ou não tem anterior")
else:
    LLC.remove_afrente(anterior)

LLC.ImprimeLL()

# teste de len e first
print("\nTamanho da LL = ", len(LLC))
print("\nPrimeiro elemento da LL = ", LLC.first())
```

E será impresso:

```
Imprimindo a lista ligada:
1  -  vermelho
2  -  preto
3  -  azul
4  -  amarelo
5  -  verde
6  -  branco

Entre com o nome da cor:laranja
A cor laranja não está na lista ligada

Entre com o nome da cor:azul
Encontrada a cor azul na lista ligada

Entre com o nome da cor:branco
Encontrada a cor branco na lista ligada

Entre com o nome da cor:fim

Adicionando laranja e violeta depois e antes da azul
Imprimindo a lista ligada:
1  -  vermelho
2  -  preto
3  -  violeta
4  -  azul
5  -  laranja
6  -  amarelo
7  -  verde
8  -  branco
```

```
Removendo as cores amarelo e lilás da lista
Cor lilas não está na lista ou não tem anterior
Imprimindo a lista ligada:
1  -  vermelho
2  -  preto
3  -  violeta
4  -  azul
5  -  laranja
6  -  verde
7  -  branco

Tamanho da LL =  7

Primeiro elemento da LL =  vermelho
```

Escreva as funções abaixo dentro da classe ListaLigadaSimples. Sempre verifique se a função funciona para os casos particulares: lista vazia, lista com um só elemento, primeiro elemento da lista, último elemento da lista, etc.

P7.1) Função Procura(x) que procura nó com info = x e devolve referência a este nó ou None (é a própria função busca (self, e) acima).

P7.2) Idem supondo que os nós estão em ordem crescente pelo campo info.

P7.3) Supondo que os nós estão em ordem crescente pelo campo info, faça uma função Adiciona_emOrdem(x) que adiciona um novo nó com info igual a x, mantendo a ordem. Para isso é necessário procurar o primeiro nó com info maior que x, mas guardar a referência ao nó anterior para que possa inserir o novo nó à frente dele.

P7.4) Função Conta(x) que devolve a quantidade de nós com info = x.

P7.5) Função ComparaLL(LLX) que compara o conteúdo (campo info) de uma lista ligada com a lista ligada LLX. Neste caso, para se compara a lista ligada LX com a LLX o comando será LX.Compara(LLX). Outra forma seria escrever a função com 2 parâmetros: Compara(LX, LLX). Faça das duas formas.

4.1. Funções fora da classe

As funções pedidas nos exercícios acima podem também estar fora da classe. Neste caso podemos fazê-las enviando como parâmetro adicional à lista ligada sobre a qual atuarão. Por exemplo, a do exercício P7.1 ficaria.

```
def Procura(minhaLL, x):

    ''' procura nó com info x em minhaLL.'''
```

```
    p = minhaLL._inicio
    while p is not None:
        if p._info == x: return p
        p = p._prox
    return None
```

Note que para se construir a função fora da classe, é necessário também conhecer os atributos do objeto ListaLigadaSimples (_inicio e _tamanho) e os atributos do objeto _Node (_info e _prox). Não é a maneira conceitualmente melhor de construir os métodos ou funções que manipulam objetos da classe, mas perfeitamente possível.

Um exemplo de uso da função acima fora da classe:

```
# teste de função fora da classe
for cor in cores:
    if Procura(LLC, cor) is None:
        print(cor, " - não está mais em cores")
    else: print(cor, " - continua em cores")
```

P7.5) Modifique as funções solicitadas em P7.2 a P7.4 colocando as funções fora da classe ListaLigadaSimples.

4.2. A sub-classe _Node

Na classe ListaLigadaSimples definida acima, usamos a classe _Node como uma classe interna apenas para evitar que os atributos _info e _prox fossem conhecidos e pudessem ser usados pelas funções que usam esse objeto. Mas não precisa ser assim. Usando a classe _Node no mesmo nível da classe ListaLigada Simples, a única modificação a ser feita seria no comando que cria um novo objeto _Node:
`novo = _Node(e, p._prox)` e `novo = _Node(e, self._inicio)`
em vez de :
`novo = self._Node(e, p._prox)` e `novo = self._Node(e, self._inicio)`

4.3. Listas Ligadas e recursão

Listas ligadas simples são estruturas claramente recursivas. Por isso, admitem também algoritmos recursivos. Cada nó de uma lista ligada pode ser entendido como um ponteiro para uma nova lista ligada. Vejamos uma definição informal:

Definição:

Lista Ligada = {
    Vazia ou
    Elemento contendo um campo de informação e uma referência para uma **Lista Ligada**
}

Assim, uma lista ligada pode sempre ser um par (info, prox), onde prox é uma referência para uma nova lista ligada. Convencionamos que se info é None, a lista está vazia. A nova classe fica então:

```
class OutraListaLigadaSimples:

    ''' Nesta nova definição de LL, Uma LL é sempre um par:
        (info, prox), onde prox é uma lista ligada.
        Quando info == None a lista ligada é vazia. '''

    def __init__ (self, info = None, prox= None):
        ''' Cria nova LL.  '''
        self._info = info
        self._prox = prox

    def is_empty(self):
        ''' retorna True se LL vazia'''
        return self._info == None

    def info(self):
        ''' retorna o campo info da LL sem remover.
            sinaliza exceção se LL vazia.'''
        if self.is_empty():
            raise Empty("Lista Ligada Vazia")
        return self._info # informação desta LL

    def adicionaLL(self, e):
        ''' adiciona uma LL com info = e antes desta LL.'''
        # novo nó referencia o inicio da LL
        novaLL = OutraListaLigadaSimples(e, self)
        return novaLL

    def busca(self, e):
        ''' procura LL com info = e. Devolve referência
            para a LL encontrada ou None
            A função é recursiva. '''
        # p percorre a lista
        if self.is_empty(): return None
        if self._info == e: return self
        return self._prox.busca(e)

    def CriaLL(self, lista):
        ''' cria uma LL com elementos contidos em lista.'''
        for k in range(len(lista) - 1, -1, -1):
            self.adiciona(lista[k])

    def ImprimeLL(self):
        ''' imprime toda a LL.'''
```

```
        p = self
        k = 1
        print("Imprimindo a lista ligada:")
        while p is not None:
            print(k, " - ", p._info)
            k = k + 1
            p = p._prox

# Testes
# Cria lista ligada
cores = ["vermelho", "preto", "azul", "amarelo", "verde",
"branco"]
LLC = OutraListaLigadaSimples()
for cor in cores:
    LLC = LLC.adicionaLL(cor)
LLC.ImprimeLL()

# Procura alguns elementos
outrascores = ["laranja", "violeta", "lilas"]
for cor in cores + outrascores:
    if LLC.busca(cor) != None: print(cor, "está na LL")
    else: print(cor, "não está na LL")
```

P7.6) A função busca acima é recursiva. Escreva a mesma de forma não recursiva.

P7.7) Refaça os métodos CriaLL e ImprimeLL acima de maneira recursiva.

P7.8) Com a mesma definição acima usando, refaça os exercícios P7.2 e P7.4 de maneira recursiva.

Da mesma forma, as funções acima, recursivas ou não, podem ser escritas também fora da classe. Só muda a maneira de usá-las.

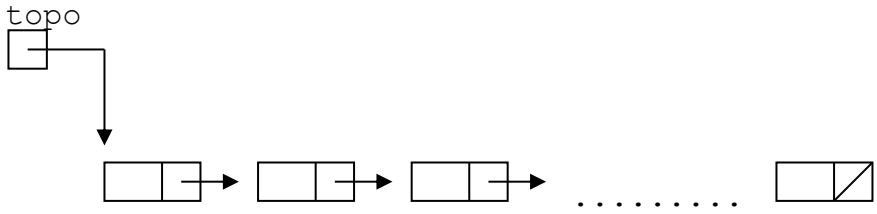
5. Pilhas e filas – Alocação Dinâmica

Pilhas e filas são estruturas de dados básicas que podem ser construídas como listas ligadas. Cada estrutura tem as suas particularidades. Por isso, vamos construir classes separadas que se utilizam da mesma classe _Node interna já definida anteriormente.

```
class _Node:

    '''Classe interna para armazenar um nó.'''

    def __init__ (self, info, prox):
        # iniciar os campos de um nó
        self._info = info
        self._prox = prox
```

5.1 Pilha (LIFO - Last In First Out)

Como numa pilha só há operações com o topo, é natural que escolhamos como topo da pilha, o inicio da lista ligada.



São aos seguintes as funções da classe PilhaListaLigada:
Empilhar elemento x – push(x)
Desempilhar elemento – pop()

Outras operações auxiliares:
Iniciar uma pilha – init()
Consultar o valor do elemento do topo – top()
Retornar True se a pilha está vazia – is_empty()
Retornar o tamanho atual da pilha – len()

Em especial as operações pop(P) e top(P) devem avisar ou sinalizar se a pilha está vazia.

```
class Empty(Exception):
    pass

class PilhaListaLigada:

    ''' implementa uma pilha usando uma LL simples. '''

    # classe _Node - interna
    class _Node:

        __slots__ = '_info', '_prox'

        def __init__ (self, info, prox):
            # inicia os campos
            self._info = info
            self._prox = prox

    # métodos de pilha
    def __init__ (self):
        ''' cria uma pilha vazia.'''
        self._topo = None # vazia
        self._tamanho = 0 # tamanho da pilha
```

```
    def __len__(self):
        ''' retorna o tamanho da pilha.'''
        return self._tamanho

    def is_empty(self):
        ''' retorna True se pilha vazia'''
        return self._tamanho == 0

    def push(self, e):
        ''' adiciona elemento ao topo da pilha.'''
        novo = self._Node(e, self._topo)
        self._topo = novo
        self._tamanho += 1

    def top(self):
        ''' retorna sem remover o topo da pilha.
            sinaliza exceção se pilha vazia.'''
        if self.is_empty():
            raise Empty("Pilha Vazia")
        return self._topo._info # topo da pilha

    def pop(self):
        ''' remove e retorna o topo da pilha.
            sinaliza exceção se pilha vazia.'''
        if self.is_empty():
            raise Empty("Pilha Vazia")
        val_topo = self._topo._info
        self._topo = self._topo._prox # pula o primeiro elemento
        self._tamanho -= 1
        return val_topo

    def ImprimeLL(self):
        p = self._topo
        print("Imprimindo a pilha")
        while p is not None:
            print(p._info)
            p = p._prox
```

O programa abaixo testa esta classe:

```
# Cria uma pilha
P = PilhaListaLigada()
# adiciona 10 elementos
for k in range(10): P.push(k)
P.ImprimeLL()
# remove 5 elementos
for k in range(5): P.pop()
```

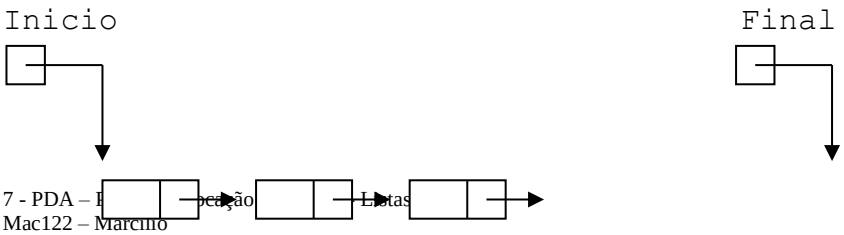
```
P.ImprimeLL()
# algumas informações da pilha
print("\ntopo da pilha = ", P.top())
print("tamanho da pilha = ", len(P))
# remove 6 elementos - vai dar excessão
for k in range(6): P.pop()
```

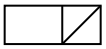
E será impresso:

```
Imprimindo a pilha
9
8
7
6
5
4
3
2
1
0
Imprimindo a pilha
4
3
2
1
0

topo da pilha =  4
tamanho da pilha =  5
Traceback (most recent call last):
  File "D:\Marcilio\Python - fontes - Marcilio\classe
PilhaListaLigada.py", line 72, in <module>
    for k in range(6): P.pop()
  File "D:\Marcilio\Python - fontes - Marcilio\classe
PilhaListaLigada.py", line 47, in pop
    raise Empty("Pilha Vazia")
Empty: Pilha Vazia
```

5.2 Fila (FIFO - First In First Out)

Como é necessário fazer operações no início da fila (remover) e final da fila (adicionar), dois apontadores são necessários, um para o início da lista ligada e outro para o final da lista ligada.



. . . . . 

A escolha é natural. Início e fim da fila coincidem com o início e fim da lista ligada.

As operações sobre a fila são as seguintes:

Adicionar um elemento x – enqueue(x)
Retirar um elemento – dequeue()

Consultar o valor do elemento do primeiro elemento – first()
Retornar True se a fila está vazia – is_empty()
Retornar o tamanho atual da fila – len()

Em especial a operação dequeue() e first() devem avisar ou sinalizar se a fila está vazia.

A classe FilaListaLigada abaixo implementa essas funções:

```
class Empty(Exception):
    pass

class FilaListaLigada:

    ''' implementa uma fila usando uma LL simples. '''

    # classe _Node - interna
    class _Node:

        __slots__ = '_info', '_prox'

        def __init__ (self, info, prox):
            # inicia os campos
            self._info = info
            self._prox = prox

    # métodos de fila
    def __init__ (self):
        ''' cria uma fila vazia.'''
        self._inicio = None # vazia
        self._final = None  # vazia
        self._tamanho = 0   # tamanho da pilha

    def __len__(self):
        ''' retorna o tamanho da fila.'''
        return self._tamanho

    def is_empty(self):
```

```
        ''' retorna True se fila vazia'''
        return self._tamanho == 0

    def first(self):
        ''' retorna sem remover o inicio da fila.
            sinaliza exceção se fila vazia.'''
        if self.is_empty():
            raise Empty("Fila Vazia")
        return self._inicio._info # elemento inicial da fila

    def enqueue(self, e):
        ''' adiciona elemento ao final da fila.'''
        novo = self._Node(e, None) # novo nó será o fim da fila
        # caso especial - fila vazia
        if self.is_empty():
            self._inicio = novo
        else:
            self._final._prox = novo
        # atualiza o fim da fila e o tamanho
        self._final = novo
        self._tamanho += 1

    def dequeue(self):
        ''' remove e retorna o elmento do inicio da fila.
            sinaliza exceção se fila vazia.'''
        if self.is_empty():
            raise Empty("Fila Vazia")
        val = self._inicio._info
        # pula o primeiro elemento
        self._inicio = self._inicio._prox
        self._tamanho -= 1
        # caso especial - a fila ficou vazia
        if self.is_empty():
            self._final = None
        return val

    def ImprimeLL(self):
        ''' só para teste.'''
        p = self._inicio
        print("Imprimindo a fila  - lista ligada:")
        k = 1
        while p is not None:
            print(k, " - ", p._info)
            p = p._prox
            k += 1

# O trecho abaixo testa a classe:
```

```
# Cria uma fila em lista ligada
F = FilaListaLigada()
# adiciona 10 elementos
for k in range(10): F.enqueue(k)
F.ImprimeLL()
# remove 6 elementos
for k in range(6): F.dequeue()
F.ImprimeLL()
# algumas informações da fila
print("\nprimeiro elemento da fila = ", F.first())
print("tamanho da fila = ", len(F))
# remove 6 elementos - vai dar exceção
for k in range(6): F.dequeue()
```

E será impresso:

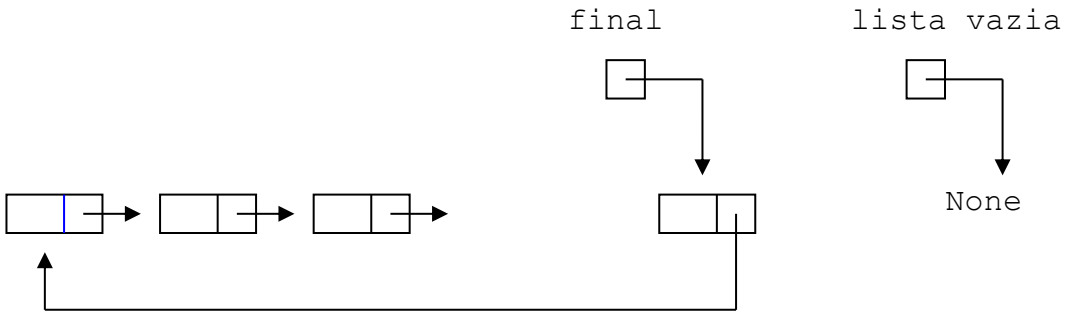
```
Imprimindo a fila  - lista ligada:
1  -  0
2  -  1
3  -  2
4  -  3
5  -  4
6  -  5
7  -  6
8  -  7
9  -  8
10  -  9
Imprimindo a fila  - lista ligada:
1  -  6
2  -  7
3  -  8
4  -  9

primeiro elemento da fila =  6
tamanho da fila =  4Traceback (most recent call last):
  File "C:/Users/msanches/Documents/Marcilio/Python - fontes -
Marcilio/Fila Lista Ligada - mac122 - 2020.py", line 90, in
<module>
    for k in range(6): F.dequeue()
  File "C:/Users/msanches/Documents/Marcilio/Python - fontes -
Marcilio/Fila Lista Ligada - mac122 - 2020.py", line 56, in
dequeue
    raise Empty("Fila Vazia")
Empty: Fila Vazia
```

6. Listas Circulares

Numa lista ligada circular, o último item aponta para o primeiro.

Numa lista circular, pode-se acessar todos os elementos a partir de qualquer elemento. Pode ser conveniente manter referências para o início e fim da lista ligada circular. Entretanto é fácil ver que se mantivermos a referência para o fim apenas, temos acesso imediato ao início. Em geral basta uma referência, pois com ela podemos percorrer toda a lista.

Em nosso modelo vamos manter a referência apenas para o final.



Uma lista circular é especialmente interessante quando temos certo número de elementos aos quais devemos oferecer algum serviço ou efetuar alguma ação de maneira continua e respeitando a ordem. Como exemplo, podemos citar o caso do sistema operacional que tem que atender aos vários processos em execução num sistema de tempo compartilhado (time-sharing), alocando a CPU a cada um deles por um período.

Adicionar um elemento no final da fila, é colocá-lo à frente do elemento final, tornando este o novo final. Remover elemento do início da fila, é retirar o elemento que está à frente do elemento final.

Para o caso em que o primeiro elemento é usado e em seguida volta para o final da lista, podemos simplesmente adiantar de um nó a referência do final. Assim, o que era o início, passa a ser o final.

Para a classe FilaCircularListaLigada, usamos as mesmas funções anteriores com a adição da rotate() que adianta a referência de final da lista.

```
class FilaCircularListaLigada:

    ''' implementa uma fila circular com LL simples. '''

    # classe _Node - interna
    class _Node:

        __slots__ = '_info', '_prox'

        def __init__ (self, info, prox):
            # inicia os campos
            self._info = info
```

```
            self._prox = prox

    # métodos de lista circular
    def __init__ (self):
        ''' cria uma lista circular vazia.'''
        self._final = None # vazia
        self._tamanho = 0  # tamanho da pilha

    def __len__(self):
        ''' retorna o tamanho da pilha.'''
        return self._tamanho

    def is_empty(self):
        ''' retorna True se pilha vazia'''
        return self._tamanho == 0

    def first(self):
        ''' retorna o valor sem remover o início da fila.
            sinaliza exceção se fila vazia.'''
        if self.is_empty():
            raise Empty("Fila Circular Vazia")
        primeiro = self._final._prox
        return primeiro._info

    def enqueue(self, e):
        ''' adiciona elemento no final da lista.'''
        novo = self._Node(e, None)
        if self.is_empty():
            novo._prox = novo    # circular
        else:
            novo._prox = self._final._prox # aponta ao primeiro
            self._final._prox = novo       # aponta ao novo
        self._final = novo
        self._tamanho += 1

    def dequeue(self):
        ''' remove e retorna o primeiro elemento da fila.
            sinaliza exceção se pilha vazia.'''
        if self.is_empty():
            raise Empty("Fila Circular Vazia")
        atual = self._final._prox
        # caso especial - 1 só elemento
        if self._tamanho ==1:
            self._final = None
        else:
            self._final._prox = atual._prox # pula o primeiro
        self._tamanho -= 1
```

```
        return atual._info

    def rotate(self):
        ''' avança a referência de final de 1 nó.'''
        if self._tamanho > 0:
            self._final = self._final._prox

    def ImprimeLL(self):
        if self._tamanho == 0: return
        p = self._final._prox
        print("Imprimindo a lista")
        while True:
            print(p._info)
            if p is self._final: return
            p = p._prox
```

O trecho abaixo testa as funções:

```
# Cria uma fila circular em lista ligada
FC = FilaCircularListaLigada()
# adiciona 10 elementos
for k in range(10): FC.enqueue(k)
FC.ImprimeLL()
# remove 5 elementos
for k in range(5): FC.dequeue()
FC.ImprimeLL()
# algumas informações da fila circular
print("\núltimo elemento = ", FC.first())
print("tamanho da fila = ", len(FC))
# remove 6 elementos - vai dar excessão
for k in range(6): FC.dequeue()
```

E será impresso:

```
Imprimindo a lista
0
1
2
3
4
5
6
7
8
9
Imprimindo a lista
5
```

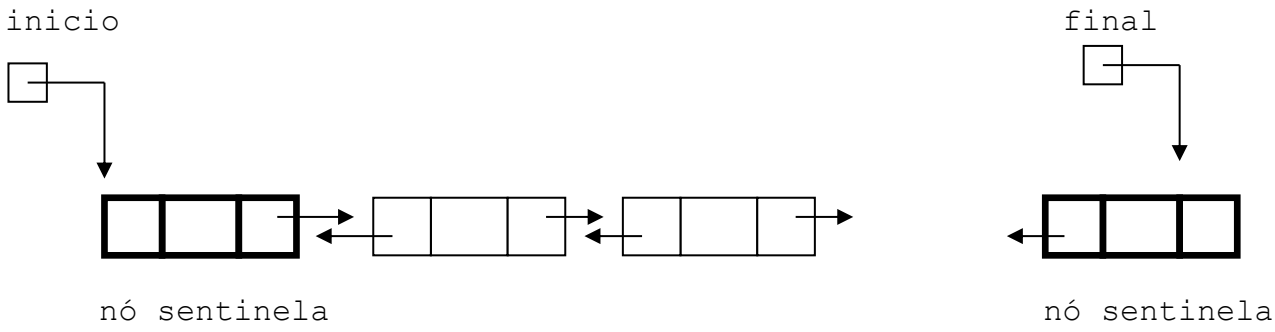
```
6
7
8
9

último elemento =  5
tamanho da fila =  5
Traceback (most recent call last):
  File "D:/Marcilio/Python - fontes - Marcilio/classe
FilaCircularListaLigada.py", line 88, in <module>
    for k in range(6): FC.dequeue()
  File "D:/Marcilio/Python - fontes - Marcilio/classe
FilaCircularListaLigada.py", line 52, in dequeue
    raise Empty("Fila Circular Vazia")
Empty: Fila Circular Vazia
```

7. Listas duplamente ligadas

Numa lista ligada simples, já vimos que é fácil adicionar ou remover do início da lista. Adicionar ou remover elementos no meio ou no final não é tão fácil. Tem que saber quem é o anterior.

Além disso, a lista pode ser percorrida numa só direção.

A estrutura que resolve esses dois problemas é a lista duplamente ligada. Cada elemento tem duas referências: para o no seguinte e para o anterior. Assim, também é interessante manter um ponteiro para o elemento final.



Para facilitar todos os algoritmos é conveniente que o primeiro e o último elemento sejam na verdade elementos sentinela da lista e que não tenham a função de nó. São usados apenas para facilitar a navegação. Podemos convencionar que seu campo info seja None para diferenciá-lo dos demais nós.

Abaixo a classe ListaDuplamenteLigada.

```
class ListaDuplamenteLigada:
```

```
    ''' operações sobre uma lista duplamente ligada. '''

    # classe _Node - interna
    class _Node:

        __slots__ = '_info', '_prev', '_prox'

        def __init__ (self, info, prev, prox):
            # inicia os campos
            self._info = info
            self._prev = prev
            self._prox = prox

    # métodos de lista duplamente ligada
    def __init__ (self):
        ''' cria uma lista circular vazia.'''
        self._inicio = self._Node(None, None, None) # vazia
        self._final = self._Node(None, None, None) # vazia
        self._inicio._prox = self._final
        self._final._prev = self._inicio
        self._tamanho = 0  # tamanho da lista

    def __len__(self):
        ''' retorna o tamanho da pilha.'''
        return self._tamanho

    def is_empty(self):
        ''' retorna True se pilha vazia'''
        return self._tamanho == 0

    def adicionar_entre(self, e, anterior, sucessor):
        ''' adiciona elemento entre 2 outros.
            retorna o novo nó.'''
        novo = self._Node(e, anterior, sucessor)
        anterior._prox = novo
        sucessor._prev = novo
        self._tamanho += 1
        return novo

    def remove(self, node):
        ''' remove nó da lista e retorna seu valor.'''
        anterior = node._prev
        sucessor = node,_prox
        anterior._prox = sucessor
        sucessor._prev = anterior
        self._tamanho -= 1
        val = node._info # guarda a informação
```

```
        # inative o nó
        node._prev = node._prox = node._info = None
        return val
```

8. As vantagens e desvantagens das listas ligadas sobre as listas sequenciais

Não resta dúvida que mexer com uma lista ligada é mais complexo que mexer com lista sequencial. A vantagem está na inserção e remoção de elementos. Não é necessário depender da eficiência dos métodos internos de alocação de memória (no caso do Python: append(), pop(), etc.). A alocação de memória, quando necessária, está por conta do seu programa.