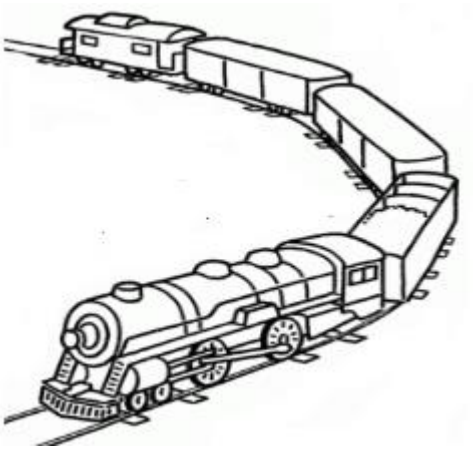
Capítulo

4 Alocação Estática e Alocação Dinâmica de Memória



Alocação estática e dinâmica de memória

Os termos estrutura de dados e tipo de dado concreto referem-se à representação interna de uma coleção de dados. As duas estruturas mais usadas para implementar coleções em linguagens de programação são os arrays (alocação estática) e as estruturas ligadas (alocação dinâmia) . Esses dois tipos de estruturas adotam abordagens diferentes para armazenar e acessar dados na memória do computador.

Essas abordagens, por sua vez, levam a diferentes compensações espaço/tempo nos algoritmos que manipular as coleções. Este capítulo examina a organização de dados e detalhes de processamento que são específicos para arrays e estruturas encadeadas.

Optou-se por estudar a estrutra lista linear para exemplificação da alocação estática e dinâmica de memória. Seu uso e implementação de vários tipos de coleções é discutida em capítulos posteriores.

Lista

A **lista** é uma coleção de objetos cuja ordem de inserção e remoção não é definida previamente. Podemos ter como exemplos: a lista de ingredientes necessários para se fazer uma receita, a lista de presença dos alunos em uma prova, a lista dos identificadores dos vagões de um trem, a lista de times de futebol da série A do campeonato brasileiro, etc.

As operações básicas na estrutura **lista** são insere (*insert*) e remove (*remove*). A figura 4.1 ilustra as operações de inserção e remoção numa estrutura **lista.** Na letra a pode-se observar a estrutura **lista** inicialmente vazia, na letra b foi inserido o primeiro elemento, representado pela chave “10”. A letra c ilustra a inserção do último elemento, representado pela chave “20”. Na letra d foi inserido o elemento na posição “1”, representado pela chave “50”. Na letra e foi inserido o elemento na posição “2”, representado pela chave “15”. Na letra f foi removido o primeiro elemento. Na letra g) foi removido o último elemento. Finalmente, na letra h foi removido o elemento da posição “1”.

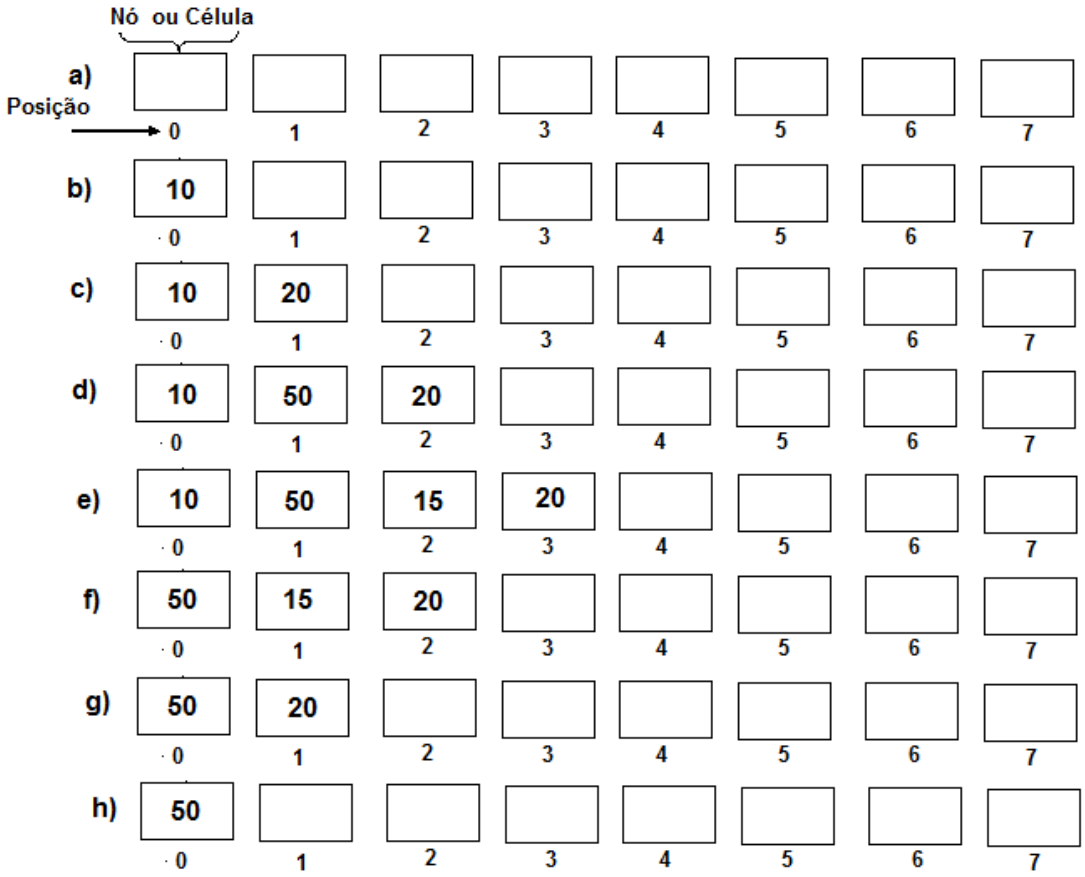


Figura 4.1 – Sequência de inserção e remoção numa estrutura genérica **Lista** (a) antes da inserção, (lista vazia) (b) insere primeiro, (c) insere último, (d) insere o elemento 50 na posição "1", (e) insere o elemento 15 na posição "2", (f) remove primeiro, (g) remove na posição "1" e (h) remove último

Lista como Tipo Abstrato de Dados

A **lista** é a estrutura mais básica, das estruturas de dados existentes, ou seja, qualquer estrutura de dados pode ser "enxergada" como sendo uma estrutura **lista**. O que diferencia é a técnica de inserção, remoção e pesquisa dos seus elementos.

Existem alguns métodos básicos para o funcionamento de uma **lista**, o método **insert(index,element)** que insere o elemento na posição index e o método **remove(index)**, que remove o elemento que está na posição "**index**".

insert(index,value):	Insere o elemento "element" na posição index da lista.
remove(index):	Remove o elemento da posição index da lista. Este método retorna o elemento removido.

A tabela *4.1* exemplifica estes métodos sendo executados e, para cada operação, pode-se observar o retorno de cada método e a variação do conteúdo da **lista.**

Tabela 4.1 – Sequência de inserção e remoção numa Lista.

OPERAÇÃO	SAÍDA	CONTEÚDO DA LISTA
insert(0,5)	-	(5)
insert(1,10)	-	(5,10)
insert(2,4)	-	(5,10,4)
insert(0,15)	-	(15,5,10,4)
remove(2)	10	(15,5,4)
insert(1,33)	-	(15,33,5,4)
remove(0)	15	(33,5,4)

Outros métodos disponíveis em Python que podem ser aplicados na estrutura **lista,** estão descritos a seguir

__str_:	transforma a lista em um string
__len__:	retorna o tamanho da lista
__iter__:	retorna a lista como um interable.
__contains__:	Utilizado na sintaxe <elemento in list>. Retorna True caso exista e, False caso contrário.
__getitem__(index):	Retorna o element que está na posição index. Utilizado na sintaxe <list[index]>.
__setitem__(index):	Altera o elemento que está na posição index. Utilizado na sintaxe <list[index]=element>..

Lista por contiguidade física

Neste modo os elementos são armazenados de forma contígua, isto é, os elementos estão em células adjacentes. A estrutura de dados utilizada para o armazenamento dos seus elementos, é o vetor (array). Em Python, podemos considerar um vetor como sendo do tipo ***list***. A figura 4.2 ilustra a representação **arrayLista** com os campos array, um vetor de n elementos, onde n é a capacidade (capacity), e o campo size que armazena o tamanho do vetor. É fácil observar que as células são numeradas de 0 a $n-1$.

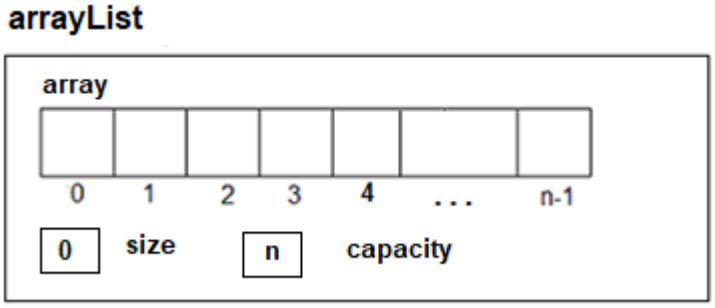


Figura 4.2 – Visão esquemática do TAD array list.

A classe arrayList

A classe concreta **arrayList** é a classe que implementa a lista linear por contiguidade física. O fragmento de código 4.1, ilustra a inicialização da classe com o seu construtor. Os parâmetros capacity (capacidade) e array (lista de inicialização) opcionais ou não. O tratamento é feito no método do construtor da classe.

```
global DEFAULT_CAPACITY
DEFAULT_CAPACITY=100

class arrayList(object):
     #Represents an arrayList
     def __init__(self, capacity=None, fillValue = None):
          # Capacity is the static size of the array.
          # fillValue is placed at each position.
          if capacity:
              if not isinstance(capacity,int):
                   raise TypeError('capacity is integer')
          else:
              capacity = DEFAULT_CAPACITY # default capacity
          self._array = [fillValue] * capacity
          self._logicalSize = 0
```

Fragmento de Código 4.1. Implementação em Python do constructor da classe arrayList

Inserção

A figura 4.3 ilustra a visão esquemática de inserções em uma **lista** contígua. Inicialmente vazia. Pode-se observar na letra (a), que campo "size" tem o valor "0", sinalizando uma **lista** vazia. Na letra (b), o elemento "5" é inserido na posição 0, início da lista. Na letra (c), o elemento "10" é inserido na posição 1, final da lista.

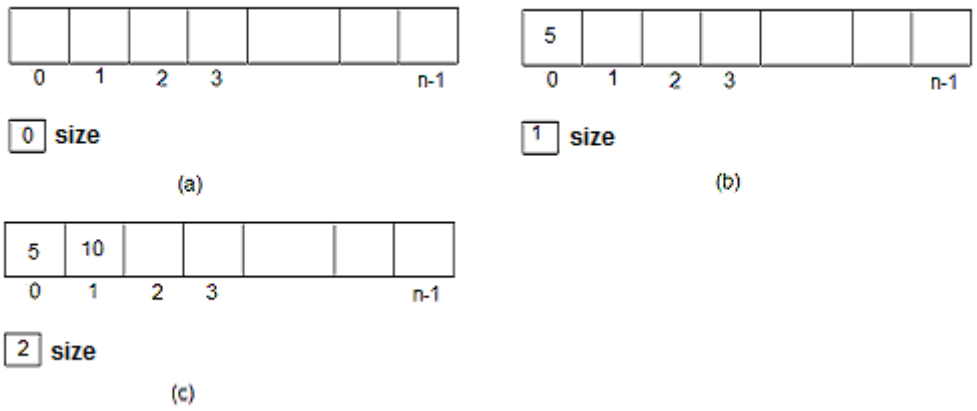


Figura 4.3 – Visão esquemática de inserção em uma **lista** contígua: (a) inicialização da estrutura, (b) inserção do primeiro elemento e (c) inserção do último elemento.

Inserção em uma posição qualquer

A figura 4.4 ilustra a visão esquemática da inserção de um elemento numa posição "i", informada como parâmetro. Inicialmente, pode-se observar que todos os elementos que estão à direita da posição "i", devem ser ajustados uma posição à esquerda, a partir do último elemento, abrindo assim a "vaga" na posição solicitada para posterior inserção.

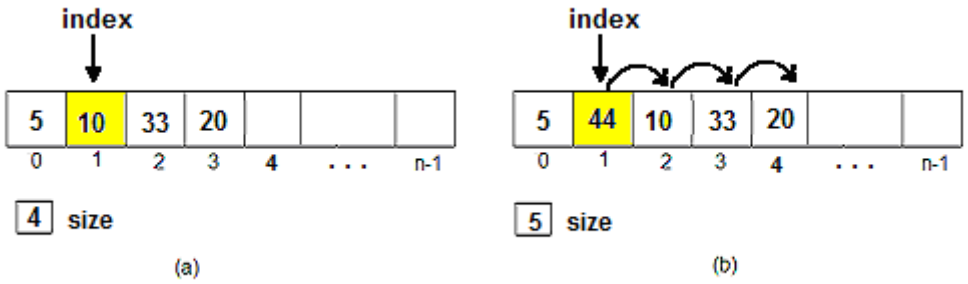


Figura 4.4 – Visão esquemática de inserção em uma **lista** contígua numa posição pré-determinada: (a) a estrutura antes da remoção e (b) após a inserção.

O método básico do processo de inserção é o método insert(index,value) que insere o elemento na posição de índice 'index'. O Fragmento de Código 4.2, ilustra a implementação do método insert(index, value).

```
def insert(self, index, value):
    #Shift items down by one position
    for i in range(self._size, index, -1):
        self._array[i] = self._array[i - 1]
    # Add new item and increment logical size
    self._array[index] = value
    self._size += 1
```

Fragmento de Código 4.2. Implementação em Python do método insert(index,element) numa **lista** contígua.

Remoção

A figura 4.5 ilustra a visão esquemática da operação de remoção do elemento da posição "index", informada como parâmetro (remove(index)). A letra (a) representa uma **lista** contígua com 4 elementos. Na letra (b), é feito a remoção do elemento da posição do ponteiro "index". Neste caso as células à esquerda da posição "3" , correspondente, ao valor do campo size -1, até a posição "index", são ajustadas uma casa à esquerda.

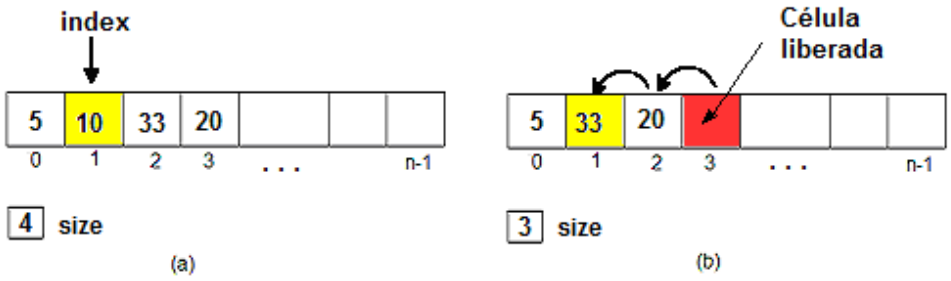


Figura 4.5 – Visão esquemática de remoção do último elemento da **lista** contígua: (a) estrutura antes da remoção e (b) após a remoção.

O método básico do processo de exclusão é o método pop(index) que exclui o elemento da posição de índice 'index'. O Fragmento de Código 4.3, ilustra a implementação do método pop(index).

```
def remove(self, index):
    # Shift items up by one position
    for i in range(index, self._size  - 1):
        self._array[i] = self._array[i + 1]
    # Decrement logical size
    self._size  -= 1
    # Decrease size of array if necessary
```

Fragmento de Código 4.3. Implementação em Python do método pop(index) numa **lista** contígua.

A remoção do primeiro elemento é feita informado como parâmetro o index = 0. A remoção do último elemento é feita na forma index = -1 ou index = tamanho da lista -1.

A classe Node

Uma outra forma de se implementar as listas lineares é a lista por encadeamento, isto é, as células são encadeadas por elos. Nesse caso, precisamos considerar as células como uma estrutura de dados. Esta estrutura denominada Node, contém, as informações do elemento (value) e um ponteiro, denominado de next, que será utilizado no encadeamento para apontar para a próxima célula da lista.

A figura 4.6 ilustra a classe node, com a célula correspondente, com os dois elementos, o campo value, que guarda o elemento e, o campo next, que é um ponteiro para a próxima célula da lista

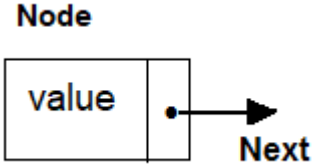


Figura 4. 6 – Visão esquemática da classe Node.

O Fragmento de Código 4.4, ilustra a implementação da classe Node. Pode-se observar o constructor de inicialização.

```
class Node(object):
     """Represents a singly linked node."""
     def __init__(self, value, next = None):
          """Instantiates a Node with a default next of None."""
          self.value = value
          self.next = next
```

Fragmento de Código 4.4- Implementação em Python da classe Node.

Usando a Classe Node

Variáveis do tipo Node são inicializadas com o valor None ou new Node(object) ou new Node(object, Node). O fragmento de código 4.5 ilustra exemplos de manipulação da classe Node.

```
# Just an empty link
node1 = None
# A node containing data and an empty link
node2 = Node("A", None)
node2 = Node("A")  # the same as the previous boot
# A node containing data and a link to node2
node3 = Node("B", node2)
```

Fragmento de Código 4.5- Exemplos de inicialização da classe Node.

A figura a 4.7 ilustra os estados das variáveis de instância da classe Node.

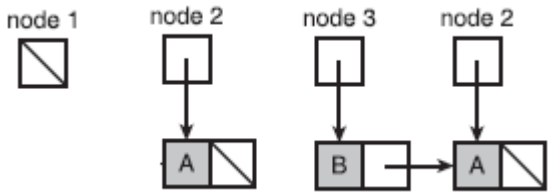


Figura 4.7. – Visão esquemática da inicialização da classe Node.

Observe o seguinte:

- node1 aponta para nenhum objeto de nó ainda (é None).
- node2 e node3 apontam para objetos que estão vinculados.
- node2 aponta para um objeto cujo próximo ponteiro é None.

Agora suponha que você tente colocar o primeiro nó no início da estrutura vinculada que já contém node2 e node3 executando a seguinte instrução:

```
node1.next = node3
```

O Python responde gerando um AttributeError. A razão para esta resposta é que a variável node1 está inicializada com o valor None e, portanto, não faz referência a um objeto node contendo um próximo campo.

Para criar o link desejado, podemos executar, como exemplo:

```
node1 = node("C", None)
```

A figura a 4.8 ilustra o estado da variável node1, de instância da classe Node.

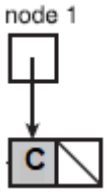


Figura 4.7. – Visão esquemática da inicialização da variável node1.

Ou podemos encadear o node1 com o node3 da seguinte forma

```
node1 = node("C", node3)
```

Ou ainda

```
node1 = Node("C", None)
node1.next = node3
```

A figura a 4.8 ilustra o estado da variável node1, de instância da classe Node, inicializada com o ponteiro next = node3.

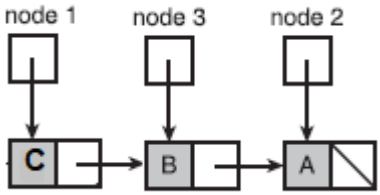


Figura 4.7. – Visão esquemática da inicialização da variável node1 inicializada com o ponteiro next = node3.

Em geral, você pode se proteger contra exceções perguntando se uma determinada variável de nó é None antes de tentar acessar seus campos.

```
if nodeVariable != None:
    <access a field in nodeVariable>
```

Estruturas encadeadas são processadas com loops. Podemos usar os loops para criar a estrutura e visitar cada nó. O fragmento de código 4.6 ilustra o uso da classe Node para criamos uma estrutura do tipo lista linear encadeada.

```
from node import Node
head = None
# Add five nodes to the beginning of the linked structure
for count in range(1, 6):
     head = Node(count, head)

# Print the contents of the structure
while head != None:
     print(head.value)
     head = head.next
```

Fragmento de Código 4.6 - Exemplos de inicialização da classe Node uutilizando uma estrutura loop.

Ao executarmos o script anterior o Pytonh retornaria:

```
5
4
3
2
1
```

Observe os seguintes pontos sobre este programa:

- Um ponteiro, head(cabeça), gera a estrutura vinculada. Este ponteiro é manipulado de forma que o item inserido mais recentemente esteja sempre no início da estrutura.

- Assim, quando os dados são exibidos, eles aparecem na ordem inversa de sua inserção. Além disso, quando os dados são exibidos, o ponteiro head (cabeça) é redefinido para o ponteiro next(próximo nó), até o ponteiro da cabeça torna-se None.

- Assim, ao final desse processo, os nós são efetivamente excluídos da estrutura vinculada. Eles não estão mais disponíveis para o programa e são reciclados na próxima coleta de lixo (garbage collections).

Lista Encadeada

A **lista** encadeada é uma sequência de nós ligados pelos elos de cada nó (ponteiro next). Os nós que compõem a estrutura são da classe Node. É necessário a criação de três elementos: o ponteiro head (cabeça da lista) que aponta para o primeiro nó da lista e, o campo size que armazena do tamanho da lista. Na implementação o ponteiro head, na realidade é a primeira célula da lista.

A figura 4.8 ilustra esquematicamente a estrutura lista encadeada, onde destacam-se os elementos básicos o ponteiro head e, o campo size (tamanho).

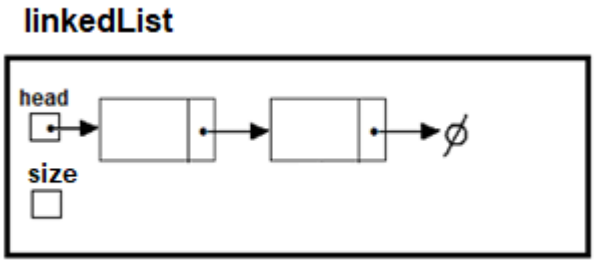


Figura 4.8 – Visão esquemática da classe linkedList.

Pode-se observar que nesse caso optou-se em se criar o campo size, para otimizar os procedimentos que necessitam do tamanho da lista. Em tempo de inserção e remoção este campo deverá ser atualizado.

O fragmento de código 9.9 ilustra um modelo de constructor da classe linkedList().

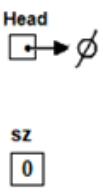
```
class linkedList(object):
     """Represents a singly linked list."""
     def __init__(self):
          """Instantiates a linked list with a default head of None and
             size zeroe
          """
          self._size = 0
          self._head = None
```

Fragmento de Código 4.7. Fragmento da Implementação em Python da classe linkedList e o seu constructor.

A inicialização da estrutura linkedList é feita sempre que executarmos o constructor da classe linkedList, como por exemplo.

```
L = linkedList()
```

As variáveis da classe serão inicializadas, size = 0 e head = None.



Inserção

inserir o primeiro

Primeiramente vamos inserir o primeiro o único elemento a lista. O fragmento de código 4.8 ilustra a inserção do primeiro elemento na lista .

```
def addFirst(self, value):
    #add item first position
    self._head = Node(value, self._head)
    self._size += 1
```

Fragmento de Código 4.8. Fragmento da Implementação em Python do método addFirst.

Pode-se observar que o ponteiro head recebe um novo nó (newNode) inicializado com o valor(value) passado como parâmetro.

A figura 4.9 ilustra uma execução do fragmento de código addFirst, passando como parâmetro o valor = 20. . A letra a, corresponde ao estado antes da inserção e a letra b, após a inserção.

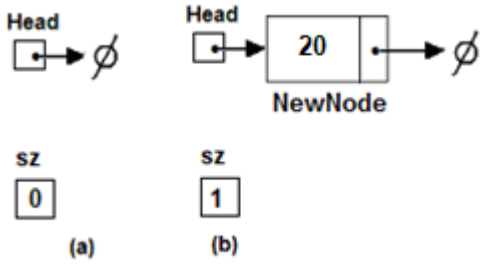


Figura 4.9 – Visão esquemática da remoção do único elemento da lista.

Agora, vamos supor que executássemos novamente, o método addFirst, passando como parâmetro o valor = 10. A figura 4.10 ilustra o procedimento insere primeiro. A letra a, corresponde ao estado antes da inserção e a letra b, após a inserção.

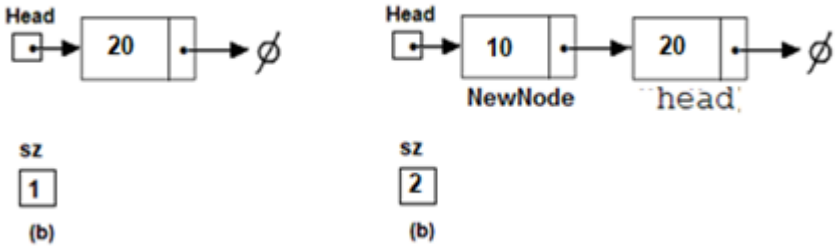


Figura 4.10 – Visão esquemática da remoção do primeiro elemento da lista.

Pode-se observar que o ponteiro head recebe um novo nó (newNode) inicializado com o valor(value) = 20, passado como parâmetro e, com o ponteiro next, inicializado com o valor do ponteiro head, antes da inserção.

inserir o último

Agora vamos inserir o último elemento na lista. O fragmento de código 4.9 ilustra a inserção do último elemento na lista .

```
def addLast(self, value):
     #add item Last position
     newNode = Node(value)
     if self._head is None:
          self._head = newNode
     else:
          current = self._head
          while current.next != None:
               current = current.next
          current.next = newNode
     self._size += 1
```

Fragmento de Código 4.9. Fragmento da Implementação em Python do método addLast.

Primeiramente inicializa-se a variável newNode com um nó inicializado com o valor a ser inserido. Depois é feito um teste se o ponteiro head é igual a None. No caso afirmativo atualiza-se o ponteiro head com o newNode, ou seja o último é o primeiro elemento a ser inserido a lista.

Caso contrário, é feita uma varredura na lista até o último nó. Para isso, cria-se um ponteiro denominado de current, inicializado com o valor do ponteiro head e o loop é executado até que o ponteiro current.next seja igual a None. Com isso podemos afirmar que o ponteiro corrente, têm como o valor, o nó corresponde à última célula da lista.

Agora, vamos supor que executássemos o método addLast, passando como parâmetro o valor = 30. A figura 4.10 ilustra o procedimento insere último. A letra a, corresponde ao estado antes da inserção e a letra b, após a inserção.

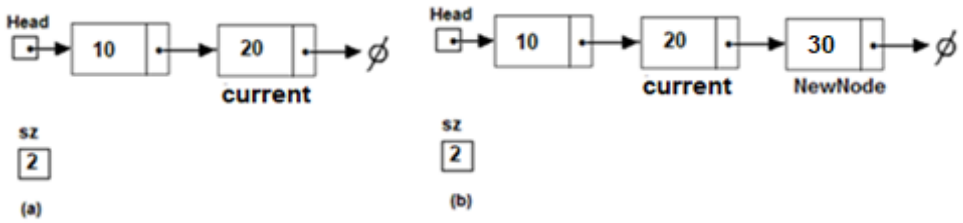


Figura 4.10 – Visão esquemática da remoção do último elemento da lista.

inserir na posição index

Finalmente, vamos inserir na posição index informada como parâmetro. O fragmento de código a seguir ilustra a inserção do elemento na posição de valor igual ao index.

O teste inicial serve para testar se a inserção será feita o início. Após executa-se um loop, para achar a posição igual a index. Finalmente é feita a inserção. O fragmento de código 4.10 ilustra esse procedimento.

```
def insert(self, index, value):
    if self._head is None or index <= 0:
        self._head = Node(value, self._head)
    else:
        # Search for node at position index - 1 or the last position
        current = self._head
        while index > 1 and current.next != None:
            current = current.next
            index -= 1
        # Insert new node after node at position index - 1
        # or last position
        current.next = Node(value, current.next)
    self._size += 1
```

Fragmento de Código 4.10. Fragmento da Implementação em Python do método insert.

Agora, vamos supor que executássemos o método insert, passando como parâmetro o valor = 40 e o index = 1. A figura 4.11 ilustra o procedimento insert. A letra a, corresponde ao estado antes da inserção e a letra b, após a inserção.

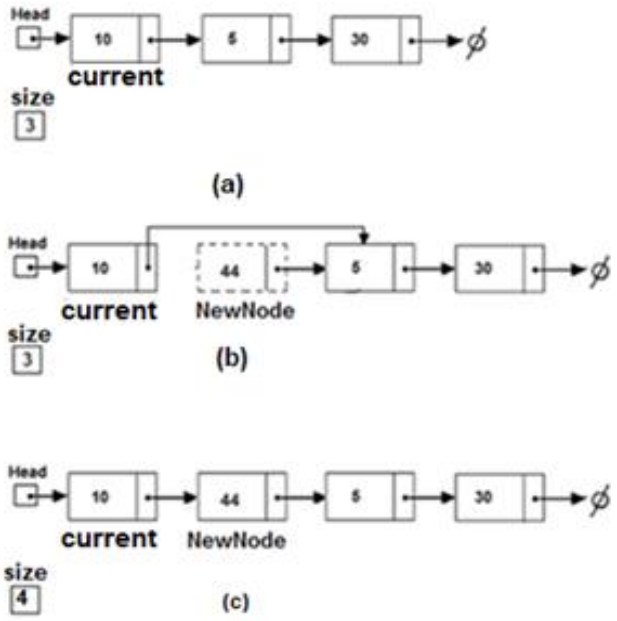


Figura 4.11 – Visão esquemática da operação do método insert(1,44): (a) antes da inserção, (b) durante a inserção e (c) após a inserção.

Remoção

remover o primeiro

Primeiramente vamos remover o primeiro o único elemento a lista. O fragmento de código 4.11 ilustra a remoção do primeiro elemento na lista .

```
def removeFirst(self):
    if self._head:
        removedItem = self._head.value
        self._head = self._head.next
        self._size -= 1
        return removedItem
    raise IndexError('remove from empty list')
```

Fragmento de Código 4.11. Fragmento da Implementação em Python do método removeFirst.

A figura 4.12 ilustra a execução da remoção do primeiro elemento da lista. O nodo corrente (currNode) será o removido.

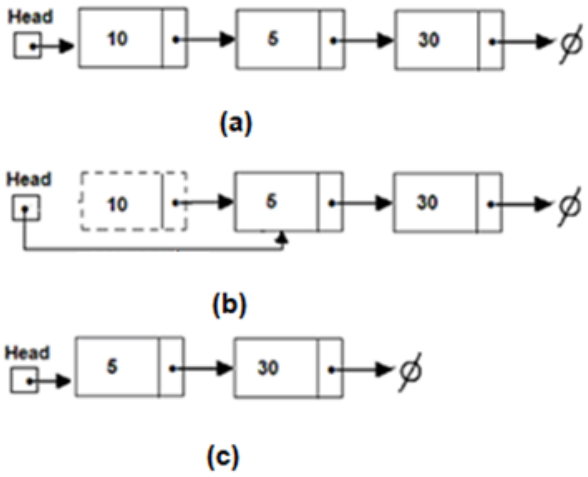


Figura 4.12 – Visão esquemática da remoção do primeiro elemento da lista: (a) antes da remoção, (b) durante a remoção e (c) após a remoção.

remover o último

Agora vamos remover o último elemento na lista. O fragmento de código 4.12 ilustra a remoção do último elemento na lista .

```
def removeLast(self):
     if self._head:
          current = self._head
          while current.next != None:
               previous = current
               current = current.next
          removedItem = current.value
          previous.next = None
          self._size -= 1
          return removedItem
     raise IndexError('remove from empty list')
```

Fragmento de Código 4.12. Fragmento da Implementação em Python do método removeLast.

A figura 4.13 ilustra a execução da remoção do último elemento da lista. O nó corrente (current) é o nó a ser removido.

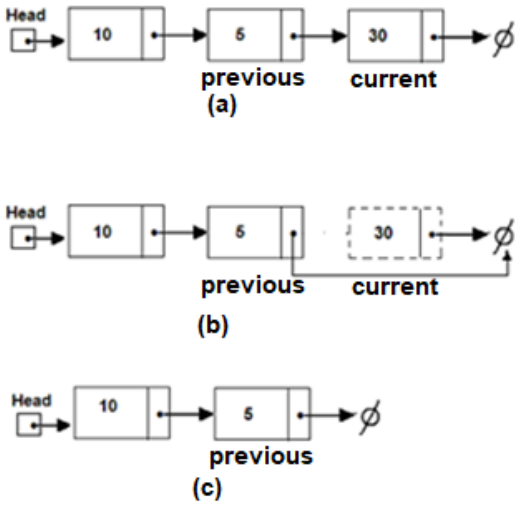


Figura 4.13 – Visão esquemática da remoção do último elemento da lista: (a) antes da remoção, (b) durante a remoção e (c) após a remoção.

remover na posição index

Finalmente, vamos remover na posição index informada como parâmetro. O fragmento de código a seguir ilustra a remoção do elemento na posição de valor igual ao index.

O teste inicial serve para verificar se a lista está vazia. O segundo teste, verifica se a remoção será feita o início. Após executa-se um loop, para achar a posição igual a index. Finalmente é feita a remoção. O fragmento de código 4.13 ilustra esse procedimento.

```
def remove(self, index):
     if self._head.next is None:
          raise IndexError('remove from empty list ')
     if index <= 0: # remove fisrt
          return self.removeFirst()
     # Search for node at position index - 1 or
     # the next to last position
     current  = self._head
     while index > 1 and current.next.next != None:
          current  = current.next
          index -= 1
     removedItem = current.next.value
     current.next = current.next.next
     self._size -= 1
     return removedItem
```

Fragmento de Código 4.13. Fragmento da Implementação em Python do método remove.

A figura 4.14 ilustra a execução da remoção de um elemento que está na posição index. O nó corrente (current) é o nó a ser removido.

.

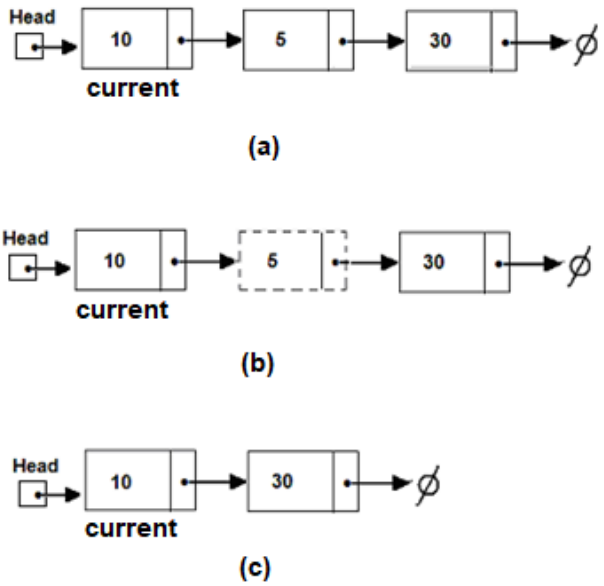


Figura 4.14 – Visão esquemática da remoção do elemento que está na posição index=1. (a) antes da remoção, (b) durante a remoção e (c) após a remoção.

Observação importante

Quando na lista encadeada se faz necessário descobrir o endereço da célula prévia em relação à um determinado nodo corrente, geralmente, é feito uma varredura em toda a lista para obtermos esse nodo prévio. Uma forma em se otimizar esse problema está em se criar um ponteiro novo na classe Node, denominado de previous, que guarda o endereço do nodo prévio.