Algoritmos e Estruturas de Dados

Aula 4

Prof Dr Tanilson Dias dos Santos

Universidade Aberta do Brasil – UAB
Universidade Federal do Tocantins - UFT

Das Aulas Anteriores

- **Conceito de Lista, Pilha e Fila:**
  - **Políticas Associadas a cada Estrutura de Dados;**
  - **Algumas Aplicações e Comportamentos.**
- **Prática de Programação:**
  - **Implementação em Python;**
  - **Tipo Abstrato de Dado, e Conceito de Classe e Objeto.**

Na Aula de Hoje, Veremos...

- Encapsulamento de Dados;
- Revisão sobre Fila, Lista e Pilha:
  - Implementação Encadeada.
- Análise de Complexidade;
- Recursão;
- Prática de Programação.

O que é Encapsulamento de Dados?

- Na **Programação Orientada a Objeto** (POO), o conceito de Encapsulamento de Dados corresponde a uma proteção adicional aos dados de um objeto contra modificações impróprias;
- Em algumas linguagens há modificadores que servem para isso (**public, private, protected**, etc);
- Em Python existe uma convenção para isso que pode ser feito de pelo menos duas formas diferentes.

O que é Encapsulamento de Dados?

- O objetivo do Encapsulamento de Dados é garantir que modificadores de acesso sejam aplicados adequadamente nas declarações de classes, permitindo visibilidade externa, apenas através de determinados métodos.
- O conceito de encapsulamento está intimamente ligado ao conceito de **ocultamento da informação** (information hiding)

Exemplo de Problema

- Um **problema** que ocorreu foi a de alguns alunos acessarem/alterarem um atributo da classe Pilha/Fila fora da classe;
  - O encapsulamento funciona com a utilização de modificadores de acesso para restringir a utilização/modificação dos atributos e dos métodos de um objeto.

```
class Pilha:
    def __init__(self):
        self.items = [ ]

    def estaVazia(self):
        return self.items == [ ]

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def topo(self):
        return self.items[len(self.items)-1]

    def tamanho(self):
        return len(self.items)
```

Exemplo de Problema

- O encapsulamento garante segurança e proteção dos dados de uma classe, pois controla o acesso às informações e ajuda a assegurar que os atributos e os métodos sejam usados de forma consistente e previsível.

```
class Pilha:
    def __init__(self):
        self.items = [ ]

    def estaVazia(self):
        return self.items == [ ]

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def topo(self):
        return self.items[len(self.items)-1]

    def tamanho(self):
        return len(self.items)
```

Exemplo de Problema

- **Erro de uso**:

```
Pilha = Pilha()

pilha.push(11)
pilha.push(45)

print("mostrando:", pilha.items)
```

```
class Pilha:
    def __init__(self):
        self.items = [ ]

    def estaVazia(self):
        return self.items == [ ]

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def topo(self):
        return self.items[len(self.items)-1]

    def tamanho(self):
        return len(self.items)
```

Exemplo de Problema

- **Erro de uso**:

```
Pilha = Pilha()

pilha.push(11)
pilha.push(45)

print("mostrando:", pilha.items)
```

```
class Pilha:
    def __init__(self):
        self.items = [ ]

    def estaVazia(self):
        return self.items == [ ]

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def topo(self):
        return self.items[len(self.items)-1]

    def tamanho(self):
        return len(self.items)
```

Modificadores de Escopo

- Em Java e C++ existe a palavra-chave **private** para indicar que um dado ou método **não é visível fora da classe**;
- Em Python, existe uma convenção de que dados ou métodos cujo nome começa com _ (**dois _underscores**) não deveriam ser acessados fora da classe.

```
class Pilha:
    def __init__(self):
        self.__items = [ ]

    def estaVazia(self):
        return self.__items == [ ]

    def push(self, item):
        self.__items.append(item)

    def pop(self):
        return self.__items.pop()

    def topo(self):
        return self.__items[len(self.__items)-1]

    def tamanho(self):
        return len(self.__items)
```

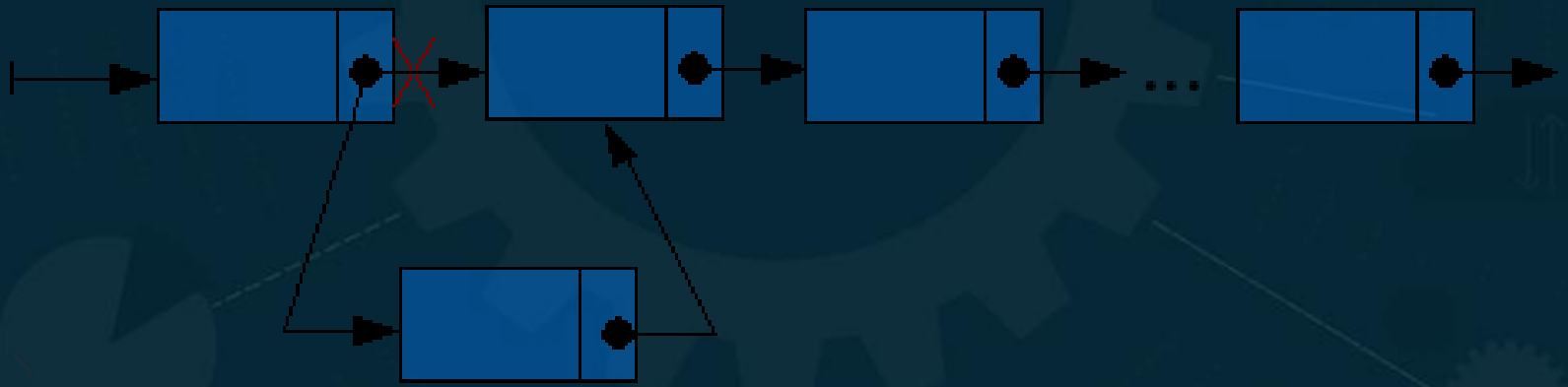
Exemplo de Problema

- **Solução**:

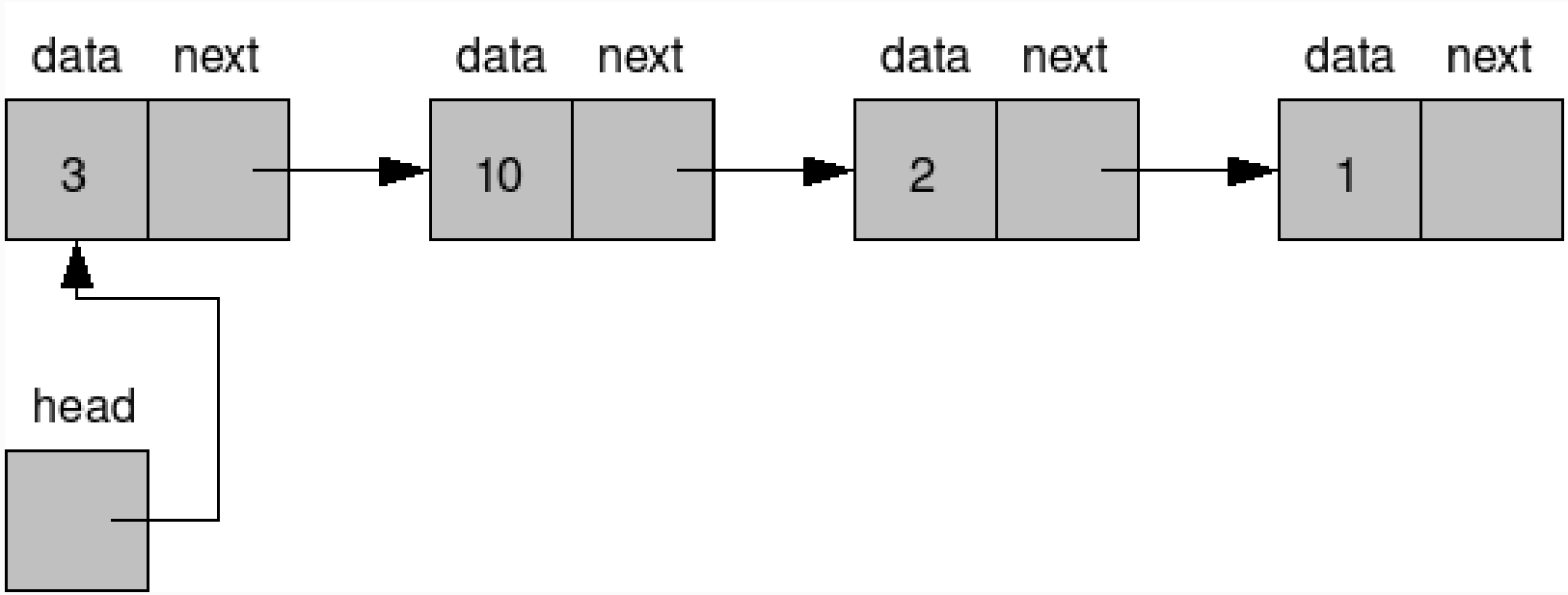
  - Implementação de **getters** e **setters**;
  - **Restrição** de acesso aos **atributos/métodos** da classe;

```
class Pilha:
    def __init__(self):
        self.__items = [ ]

    def getItems(self):
        return self.__items

    def setItems(self, item):
        self.__items.append(item)
```

Implementação Encadeada



Ideia Geral



Lista Encadeada – Classe 1

```
#Essa classe representa cada Vertice da Lista

class NodaLista:
    def __init__(self, dado=0, proximo_no=None):
        self.dado = dado
        self.proximo = proximo_no

    #imprimindo encadeadamento
    def __repr__(self):
        return '%s -> %s' % (self.dado, self.proximo)
```

Lista Encadeada – Classe 2

```
#Essa classe vai ser a cabeça da lista

class ListaEncadeada:
      def __init__(self):
        self.cabeca = None

    def __repr__(self):
        return "[" + str(self.cabeca) + "]"
```

Lista Encadeada – Função Inserir

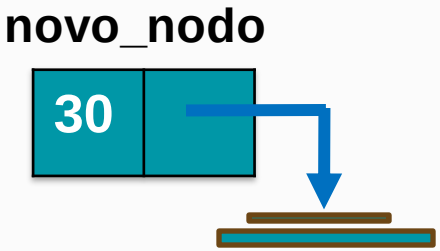
```
def insere_no_inicio(lista, novo_dado):
    # 1) Cria um novo nodo com o dado a ser armazenado.
    novo_nodo = NodaLista(novo_dado)

    # 2) Faz com que o novo nodo seja a cabeça da lista.
    novo_nodo.proximo = lista.cabeca

    # 3) Faz com que a cabeça da lista referencie o novo nodo.
    lista.cabeca = novo_nodo
```

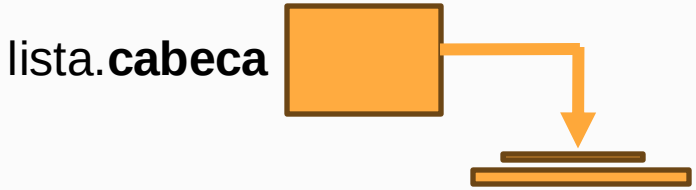
Lista Encadeada – Manipulando a Lista

```
l1 = ListaEncadeada()

insere_no_inicio(l1, 30)
print("imprimindo l1:", l1)

#apos nova insercao
insere_no_inicio(l1, 15)
print("imprimindo l1:", l1)

insere_no_inicio(l1, 26)
print("imprimindo l1:", l1)
```

Lista Encadeada – Funcionamento

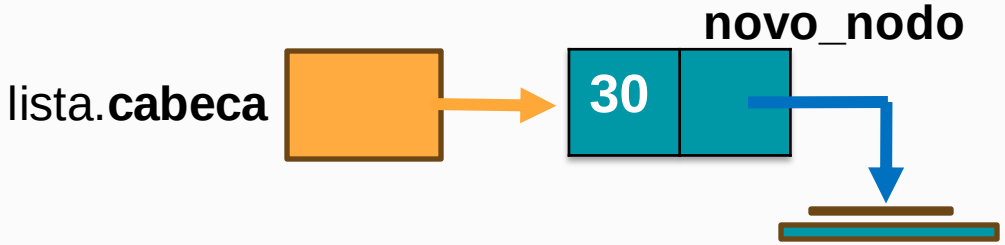
```
def insere_no_inicio(lista, novo_dado):
        novo_nodo = NodaLista(novo_dado)
        novo_nodo.proximo = lista.cabeca
        lista.cabeca = novo_nodo
```





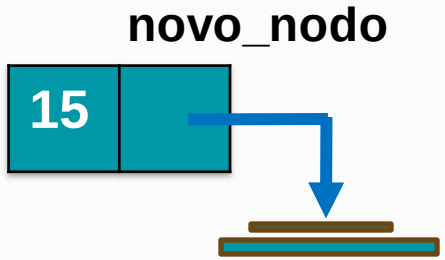
Lista Encadeada – Funcionamento

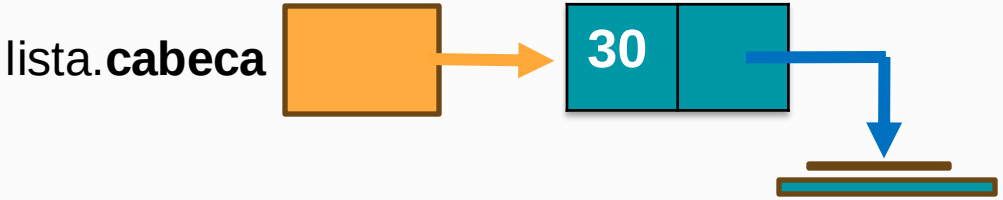
```
def insere_no_inicio(lista, novo_dado):
        novo_nodo = NodaLista(novo_dado)
        novo_nodo.proximo = lista.cabeca
        lista.cabeca = novo_nodo
```



Lista Encadeada – Funcionamento

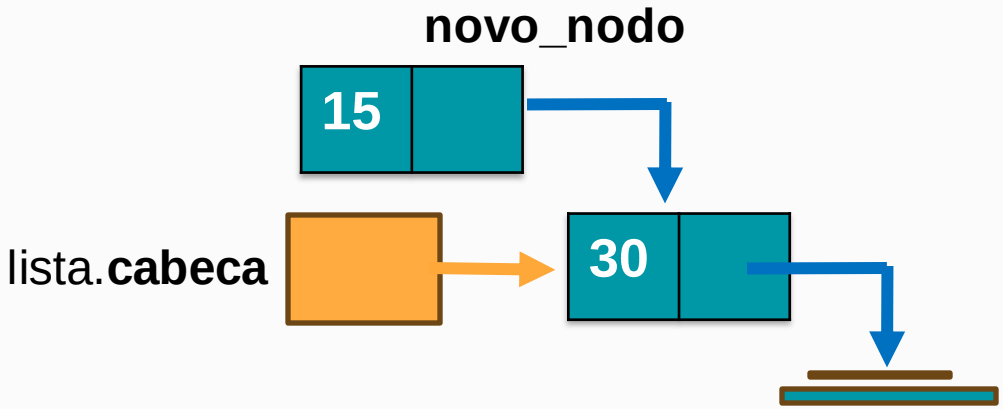
```
def insere_no_inicio(lista, novo_dado):
        novo_nodo = NodaLista(novo_dado)
        novo_nodo.proximo = lista.cabeca
        lista.cabeca = novo_nodo
```





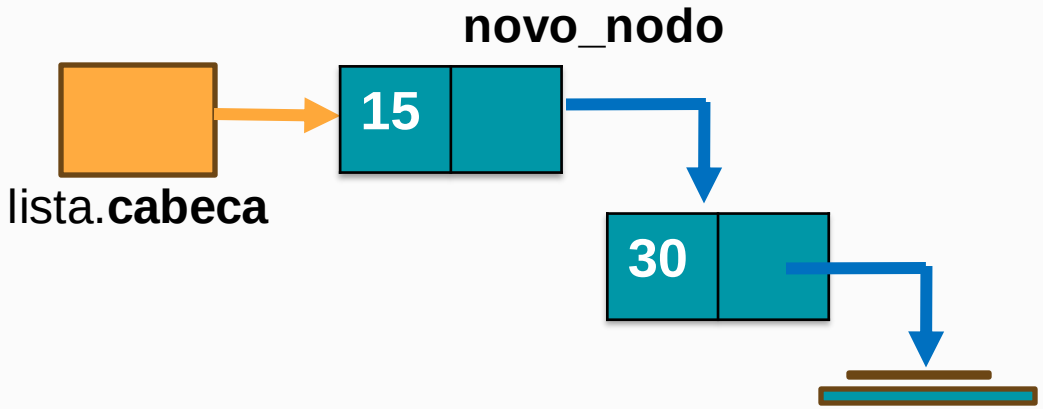
Lista Encadeada – Funcionamento

```
def insere_no_inicio(lista, novo_dado):
        novo_nodo = NodaLista(novo_dado)
        novo_nodo.proximo = lista.cabeca
        lista.cabeca = novo_nodo
```



Lista Encadeada – Funcionamento

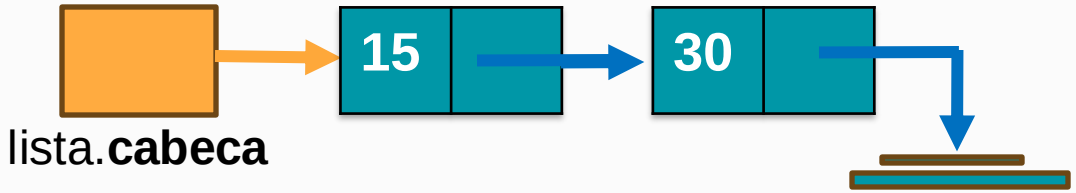
```
def insere_no_inicio(lista, novo_dado):
        novo_nodo = NodaLista(novo_dado)
        novo_nodo.proximo = lista.cabeca
        lista.cabeca = novo_nodo
```



Lista Encadeada – Inserir no Final

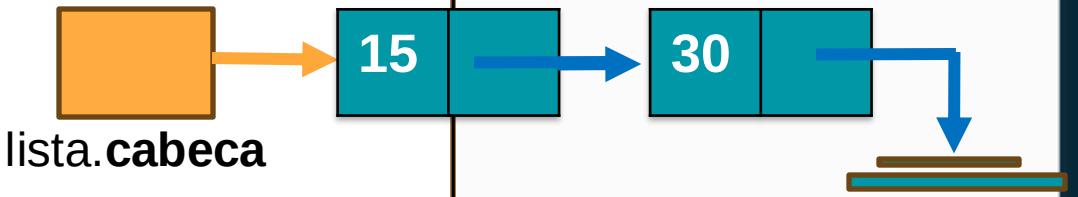
```
def insere_no_final( lista, novo_dado ):
  novo_nodo = NodaLista(novo_dado)
  novo_nodo.proximo = None
  aux = lista.cabeca
  while(aux.proximo != None):
    aux = aux.proximo
  aux.proximo = novo_nodo
```

novo_nodo



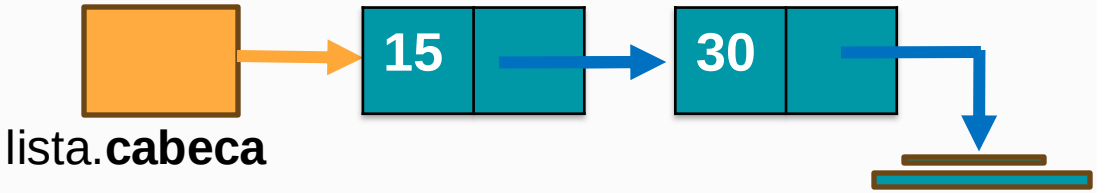
Inserir Depois

```
#insere o dado1 depois do dado2. E dado2 estah na lista

def insere_depois(lista, dado1, dado2):
  if (not estah_na_lista(lista, dado2)):
    print("impossivel inserir antes, pois dado: ", dado2,
"não estah na lista! Retornando.")
    return False

  aux = lista.cabeca
  while(aux.dado != dado2):
    aux = aux.proximo

  novo_nodo = NodaLista(dado1)
  novo_nodo.proximo = aux.proximo
  aux.proximo = novo_nodo
```



Inserir Depois Utiliza a Função estah_na_lista

```
def estah_na_lista(lista, dado):
  aux = lista.cabeca
  while(aux != None):
    if(aux.dado == dado):
      return True
    aux = aux.proximo
  return False
```



Encadeamento de Outras Estruturas

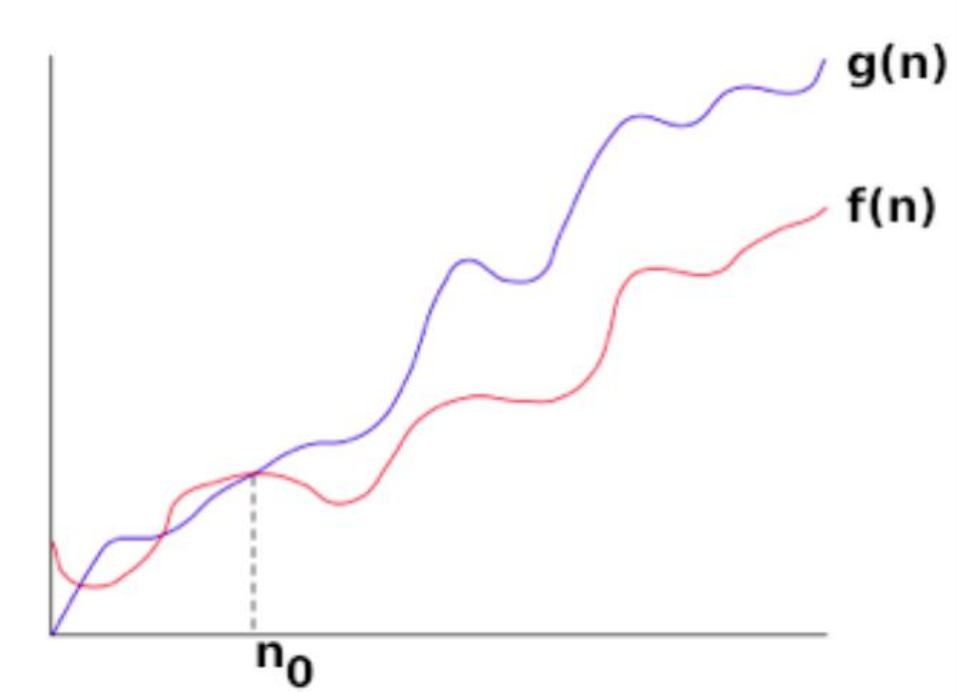
- Como implementar Filas e Pilhas encadeadas?
- Esse serão alguns tópicos do nosso seminário.

Análise de Complexidade e Notação Assintótica de Funções

Notação O

- Dadas funções assintoticamente não-negativas f e g, dizemos que f está na ordem O de g e escrevemos $f = O(g)$ se existe um número positivo c tal que $f(n) \leq c * g(n)$ para todo n suficientemente grande. Em outras palavras, se existem números positivos c e n0 tais que $f(n) \leq c * g(n)$ para todo $n \geq n0$.

f = O(g)



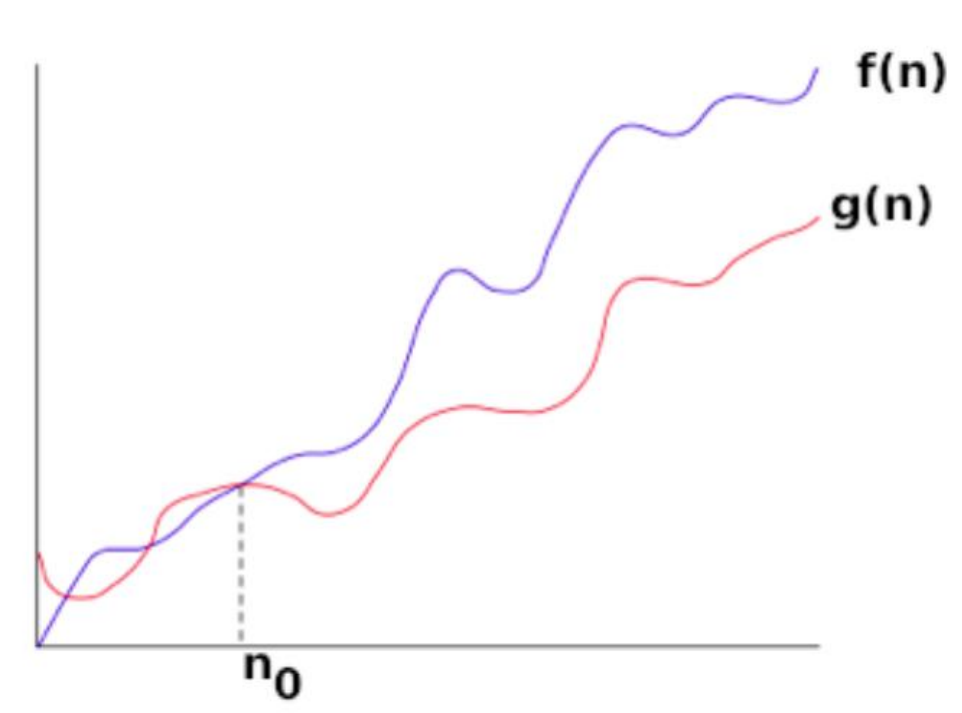
f = O(g)

- Seja f(n) = 3n^2 + 5n + 7, então f (n) = O(n^2);
- Seja f(n) = 3n^2 + 5n + 7, então também f (n) = O(n^3);
- f(n) = 2n^2 + 4y − 18n, f(n) = O(n^2 + y);
- f(n) = 2n + logn + 3, f(n) = O(n);
- f(n) = 45, f(n)= O(1).

Notação Ω

- Dadas funções assintoticamente não-negativas f e g , dizemos que f está na ordem Ômega de g e escrevemos $f = \Omega(g)$ se existem constantes positivas c, n0 tal que $f(n) \geq c * g(n)$ para todo $n \geq n0$.

$f = \Omega(g)$



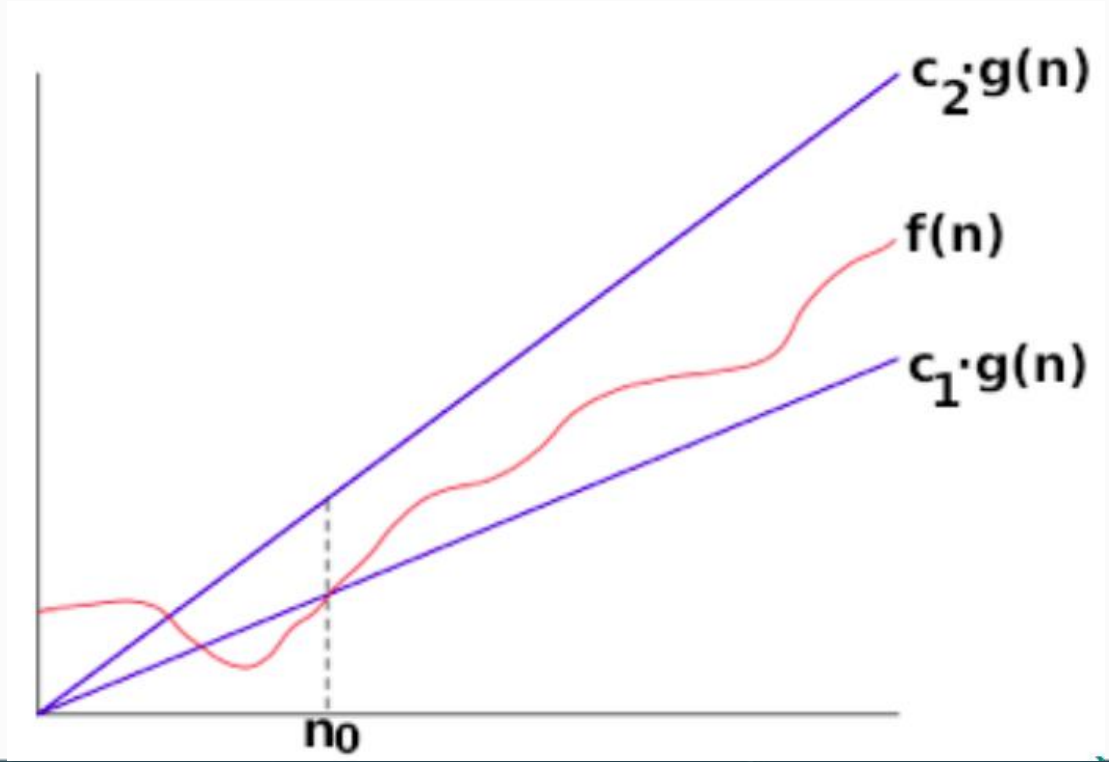
f = Ω (g)

- 5n^2 + 2n + 3 = Ω(n^2);
- n^3 + 7n + 1 = Ω(n^2);
- 2^n = Ω(n^2);

Notação Θ

- Dizemos que duas funções assintoticamente não negativas f e g são da mesma ordem e escrevemos f = Θ(g) se f = O(g) e f = Ω(g). Em outras palavras, dizer que f = Θ(g) significa que existem constantes positivas c 1 e c 2 tais que c 1 ∗ g (n) ≤ f(n) ≤ c 2 ∗ g (n) para todo n ≥ n 0 .

$f = \Theta(g)$



f = Θ (g)

- As funções f(n) = n^2 , f (n) = (3/2) ∗ n^2 , f (n) = 9999 ∗ n^2, f (n) = n^2 /1000 e f (n) = n^2 + 100n pertencem todas à ordem Θ(n^2).
- Para quaisquer números a e b maiores que 1, a função log a n está em Θ(log b n).

Recursão - A Capacidade de Invocar a Si Mesmo

O que é Recursão?

- **Recursão** é um método de resolução de problemas que envolve quebrar um problema em subproblemas menores e menores até chegar a um problema pequeno o suficiente para que ele possa ser resolvido trivialmente. Normalmente recursão envolve uma função que chama a si mesma. Embora possa não parecer muito, a recursão nos permite escrever soluções elegantes para problemas que, de outra forma, podem ser muito difíceis de programar.

Exemplo - Fatorial

- Como calcular **n!** ?
- Resposta: n*(n-1)*(n-2)* ... *3*2*1

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

- Quando chamamos fatorial(5), na linha 6/7, o compilador salva o contexto do programa na linha 6/7 e redireciona a execução para a linha 1;
- Na linha 2 **não é efetuado** o desvio condicional, então há uma chamada na linha 4, onde o n = 5, e chama-se novamente a função fatorial;

```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

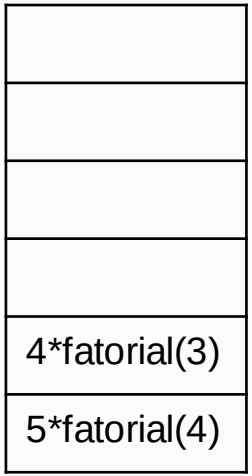
- O compilador salva o contexto do programa na linha 4 e invoca fatorial(n-1), como n=5, então fatorial (4);


```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

- Na linha 2, o desvio condicional não é executado, o fluxo do programa vai para a linha 4;
- O compilador salva o contexto do programa na linha 4 e invoca fatorial(n-1), como n=4, então fatorial (3);

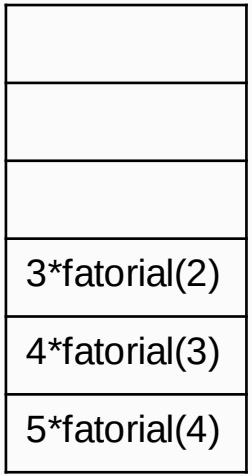



```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

- Na linha 2, o desvio condicional não é executado, o fluxo do programa vai para a linha 4;
- O compilador salva o contexto do programa na linha 4 e invoca fatorial(n-1), como n=3, então fatorial (2);




```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

- Na linha 2, o desvio condicional não é executado, o fluxo do programa vai para a linha 4;
- O compilador salva o contexto do programa na linha 4 e invoca fatorial(n-1), como n=2, então fatorial (1);

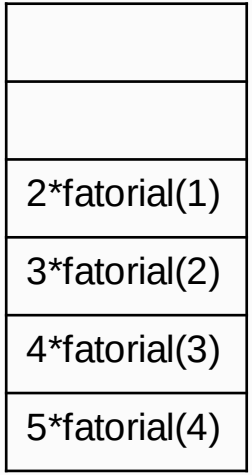
2*fatorial(1)
3*fatorial(2)
4*fatorial(3)
5*fatorial(4)

```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

- Na linha 2, o desvio condicional (com n==1) **agora é executado!** E retorna 1;
- O resultado é devolvido para a última invocação da função;
- O compilador recupera o contexto salvo anterior-mente na linha 4, i.e. 2*fatorial(1);

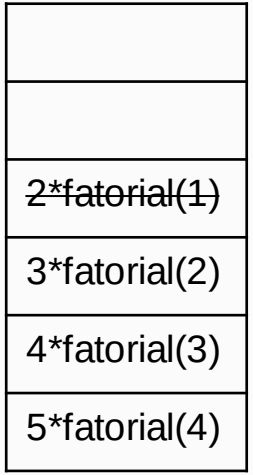



```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

- Como fatorial (1) retornou 1, então calcula-se 2*1 = 2;
- A função efetua o retorno;
- Retira-se o elemento da pilha e devolve-se o controle de fluxo do programa.

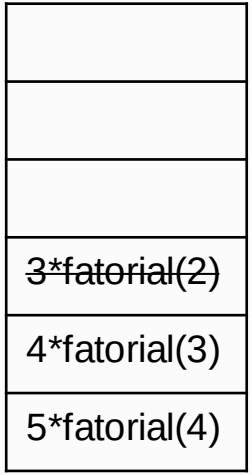



```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

- O compilador retoma o contexto da última chamada;
- Como fatorial (2) retornou 2, então calcula-se 3*2 = 6;
- A função efetua o retorno;
- Retira-se o elemento da pilha e devolve-se o controle de fluxo do programa.

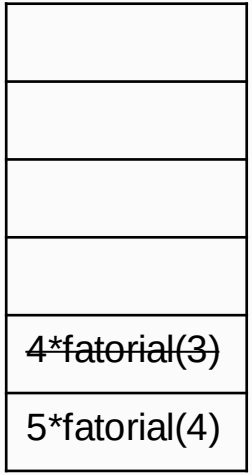



```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

- O compilador retoma o contexto da última chamada;
- Como fatorial (3) retornou 6, então calcula-se 4*6 = 24;
- A função efetua o retorno;
- Retira-se o elemento da pilha e devolve-se o controle de fluxo do programa.




```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

- O compilador retoma o contexto da última chamada;
- Como fatorial (4) retornou 24, então calcula-se 5*24 = 120;
- A função efetua o retorno;
- Retira-se o elemento da pilha e devolve-se o controle de fluxo do programa.


```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Exemplo - Fatorial

- O compilador retoma o contexto da última chamada;
- A pilha de execução está vazia;
- A função fatorial (5) retornou 120;
- O controle de fluxo do programa é devolvido e retorna para a linha 6/7.

```
1
2
3
4
5
6
7
8
9
```

```
#Recursividade

def fatorial(n: int) -> int:
    if n == 1 or n == 0:
        return 1
    return n * fatorial(n - 1)

print( fatorial(5) )
```

Vantagens e Desvantagens da Recursividade

Vantagens:

- Clareza e legibilidade: em alguns casos, a implementação de um algoritmo recursivo pode ser mais clara e legível do que uma solução iterativa. Isso ocorre especialmente quando o problema pode ser naturalmente dividido em subproblemas menores;
- Solução elegante: a recursão permite a resolução de problemas complexos de forma elegante, utilizando a própria definição do problema para resolvê-lo;
- Reutilização de código: a função recursiva pode ser reutilizada em diferentes contextos, desde que o problema em questão possa ser dividido em subproblemas menores.

Vantagens e Desvantagens da Recursividade

Desvantagens:

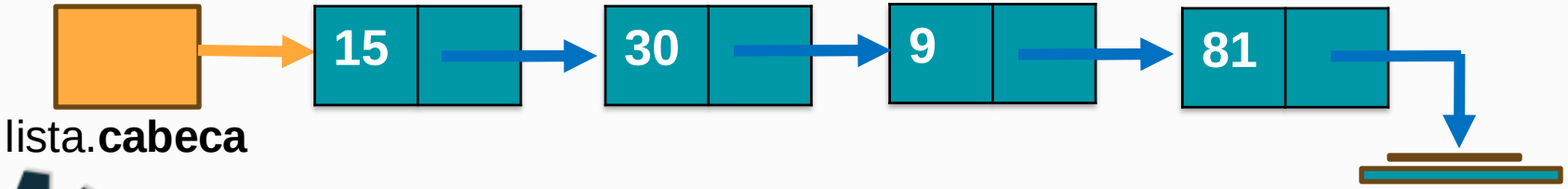
- Consumo de recursos: a recursão geralmente consome mais recursos do que uma solução iterativa, devido à pilha de chamadas que é criada a cada chamada recursiva. Isso pode levar a problemas de desempenho e até mesmo estourar a pilha de execução em casos extremos;
- Dificuldade de depuração: a depuração de funções recursivas pode ser mais complexa do que a depuração de soluções iterativas, pois é necessário acompanhar o fluxo de execução em cada chamada recursiva.
- Possibilidade de loop infinito: se a função recursiva não for projetada corretamente, pode ocorrer um loop infinito, o que resultará em travamento do programa.

Exercícios com Recursão (1)

- A sequência de Fibonacci é uma sequência de números inteiros, onde são dados os dois primeiros números, i.e. F(0) = 1, F(1) = 1, e todos os demais termos são calculados como a soma dos dois anteriores, por exemplo, F(2) = F(1) + F(0). De forma geral, F(n) = F(n-1) + F(n-2).
- Escreva uma função recursiva para calcular a sequência de Fibonacci.

Exercícios Listas Encadeadas (2)

- Considere uma implementação de lista encadeada como descrita no slide da Aula 4. Implemente a função **inserir_antes(lista, novo_dado, dado1)** onde você deve percorrer a lista, em busca de um determinado elemento e efetuar a inserção do elemento **novo_dado** antes do elemento definido pelo usuário em **dado1**.
- Considere a Lista como mostrada a seguir, inicialmente.



Resumo da Aula e Plano de Estudos

Tarefas Semanais

- Refazer Exercícios da Aula;
- Responder Questionário Avaliativo (vale 1.0 ponto);
- Responder Fórum;
- Revisar Recursividade e Encadeamento;
- Estudar Tuplas, Conjuntos e Dicionários para a próxima aula;
- Pensar em alguns possíveis tópicos para os seminários: pilha encadeada, fila encadeada, árvore, lista duplamente encadeada, fila de prioridade, etc.

Conclusão e Próxima Aula

- **Aula de Hoje:**
  - **Implementamos Listas Encadeadas;**
  - **Entendemos os principais Conceitos de Complexidade;**
  - **Introduzimos o Conceito de Recursividade.**
- **Próxima Aula:**
  - **Tuplas, Conjuntos e Dicionários.**

Algoritmos e Estruturas de Dados

Aula 4

Prof Dr Tanilson Dias dos Santos

Universidade Aberta do Brasil – UAB
Universidade Federal do Tocantins - UFT