Estruturas de Dados — Pilhas, Filas, Listas

Fabio Gagliardi Cozman
Thiago Martins

PMR3201
Escola Politécnica da Universidade de São Paulo

Introdução

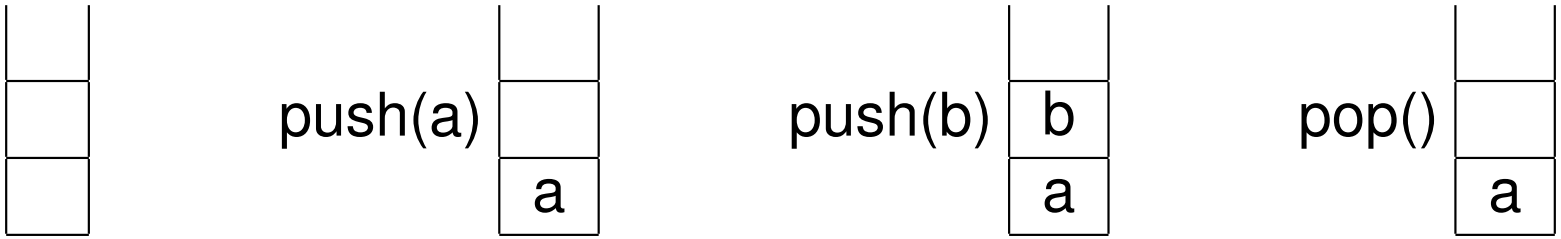
- Estruturas de dados são objetos que armazenam dados de forma eficiente, oferecendo certos “serviços” para o usuário (ordenação eficiente dos dados, busca por meio de palavras chave, etc).
- Técnicas de programação orientada a objetos são úteis quando temos que codificar estruturas de dados.
- As estruturas básicas abordadas neste curso são:
  - Pilhas, filas, listas ligadas.
  - Árvores e árvores de busca.
  - Hashtables (tabelas de dispersão).
  - Grafos.

Abstração

- Uma estrutura de dados abstrai as características principais de uma atividade que envolve armazenamento de informações.
- Por exemplo, a estrutura de *fila* armazena dados de forma que o dado há mais tempo na estrutura é o primeiro a ser retirado.

Pilhas

- Uma *pilha* é uma estrutura de dados em que o acesso é restrito ao elemento mais recente na pilha.



Pilhas: operações básicas

- As operações básicas realizadas com uma pilha são:
  - push: inserir no topo;
  - pop: retirar do topo;
  - top: observar o topo.
  - vazia?: Verifica se a pilha não contém elementos.
- Em uma pilha "ideal", operações básicas devem ocorrer em $O(1)$, independentemente do tamanho N da pilha (ou seja, em tempo constante).

Implementação de Pilha em Python

- Em Python, toda lista implementa as funções básicas de uma pilha.
- Mas e a complexidade das operações?

Implementação de Pilha em Python

Seja `a` uma lista.

- push: Equivale a `a.append(x)` onde `x` é o elemento.
- pop: Equivale a `a.pop()` que retorna o último elemento inserido. Se a lista estiver vazia, uma exceção do tipo `IndexError` é lançada.
- top: Equivale a `a[-1]`. Se a lista estiver vazia, uma exceção do tipo `IndexError` é lançada.
- Vazia?: A expressão `not a` retorna verdadeiro se a pilha não contém elementos. De fato, toda conversão de lista em variável booleana produz `True` se há algum elemento na lista, `False` caso contrário.

Implementação de Pilha em Python – Complexidade

Como dito, em uma pilha *ideal*, as operações devem ser completadas em *tempo constante* $\mathcal{O}(1)$.
lista em python são armazenadas em endereços *contíguos* de memória pré-alocados (reservados).
Estes espaços tm uma capacidade de armazenamento *finita* que não pode ser excedida!

Implementação de Pilha em Python – Complexidade

O que acontece se uma operação `append` (durante um push, por exemplo), excede a capacidade pré-alocada?
O *runtime* de Python reserva uma nova região de memória com capacidade adequada e *copia* os dados da lista antiga na nova região.
Esta cópia tem complexidade $\mathcal{O}(N)$!

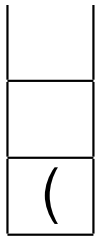
Implementação de Pilha em Python – Complexidade

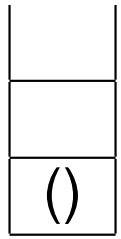
tttpush() a complexidade desta operação será sempre $\mathcal{O}(N)$.
Complexidade *amortizada*: O valor médio de complexidade sobre uma sequência de operações:
Se os tamanhos da região na memória crescem *geometricamente*, a probabilidade de um `push` produzir uma realocação cai exponencialmente.
Assim, a complexidade *amortizada* de um `push` é *constante*.

Consistência de parênteses

- Considere o problema de verificar se existe um fechamento de parênteses para cada abertura em uma expressão algébrica com letras e símbolos $+, -, *, /$.
- Pode-se utilizar uma pilha:

$$A + B * (C/D + E)$$





Verificação de consistência de parênteses

```
def verifica_consistencia(entrada):
    pilha = []
    try:
        for c in entrada:
            if c in {'(', '[', '{'}:
                pilha.append(c)
            elif c==')' and pilha.pop() !='(':
                raise ValueError("Falta algum fechamento")
            elif c==']' and pilha.pop() !='[':
                raise ValueError("Falta algum fechamento")
            elif c=='}' and pilha.pop() !='{':
                raise ValueError("Falta algum fechamento")
        if pilha:
            raise ValueError("Falta algum fechamento")
    except IndexError:
        raise ValueError("Falta alguma abertura")
```

Avaliação de expressões

- Pilhas são muito usadas no processamento de linguagens, por exemplo em compiladores.
- Uma aplicação importante é a conversão e avaliação de expressões numéricas.
- Existem três tipos de notações para expressões numéricas:
  1. infixa, onde operador entre operandos: $(A + B)$;
  2. pós-fixa, onde operador segue operandos: $(AB+)$ (notação polonesa reversa);
  3. pré-fixa, onde operador precede operandos: $(+AB)$ (notação polonesa).

Notação pós-fixa

A vantagem da notação pós-fixa é que ela dispensa parênteses.

Infixa	Pós-fixa
$A - B * C$	$ABC * -$
$A * (B - C)$	$ABC - *$
$A * B - C$	$AB * C-$
$(A - B) * C$	$AB - C*$
$A + D/(C * D\hat{}E)$	$ADCDE\hat{} * /+$
$(A + B)/(C - D)$	$AB + CD - /$

Avaliação de expressões

- Suponha que tenhamos uma expressão pós-fixa e desejemos obter o valor da expressão (“avaliar a expressão”).
- Fazemos isso passando pelos elementos da expressão,
  1. empilhando cada operando;
  2. processando cada operador:
     1. retiramos dois operandos da pilha;
     2. executamos a operação;
     3. empilhamos o resultado.
- No final o resultado está no topo da pilha.

Exemplo: expressão $1; 2; 3; \hat{}; -; 4; 5; 6; *; +; 7; *; -$

Operação Parcial	Conteúdo da Pilha
Insere 1	1
Insere 2	1; 2
Insere 3	1; 2; 3
Operador: 2^3	1; 8
Operador: 1-8	-7
Insere 4	-7; 4
Insere 5	-7; 4; 5
Insere 6	-7; 4; 5; 6
Operador: 5*6	-7; 4; 30
Operador: 4+30	-7; 34
Insere 7	-7; 34; 7
Operador: 34*7	-7; 238
Operador: -7-238	-245 (**Resultado final**)

Conversão para notação pós-fixa

- Considere que temos uma expressão em notação infixa.
- Para convertê-la a notação pós-fixa, usamos uma pilha.
- Devemos varrer a expressão infixa da esquerda para a direita,
  - se encontramos um operando, o colocamos na saída;
  - se encontramos um operador, o colocamos em uma pilha, desempilhando...

Exemplo

- $A - B * C + D$:

Entrada	Pilha	Saída
A		A
$-$	$-$	A
B	$-$	$A\ B$
$*$	$-*$	$A\ B$
C	$-*$	$A\ B\ C$
$+$	$+$	$A\ B\ C * -$
D		$A\ B\ C * -\ D +$

Conversão para notação pós-fixo: algoritmo

- Devemos varrer a expressão infixa da esquerda para a direita,
  - se encontramos um operando, o colocamos na saída;
  - se encontramos um operador, o colocamos em uma pilha, desempilhando e colocando na saída os operadores na pilha até encontrarmos um operador com precedência menor...
    - Precedência: $+$ e $-$, seguida por $*$ e $/$, seguida por $\hat{}$.
- Ao final, desempilhamos e colocamos na saída os operadores que restarem na pilha.

Exemplo

- $A * B - C + D$:

Entrada	Pilha	Saída
A		A
$*$	$*$	A
B	$*$	$A\ B$
$-$	$-$	$A\ B\ *$
C	$-$	$A\ B\ *\ C$
$+$	$+$	$A\ B\ *\ C\ -$
D		$A\ B\ *\ C\ -\ D\ +$

Parênteses

- Para lidar com parênteses, podemos criar uma nova pilha a cada abertura de parênteses, e operar nessa nova pilha até encontrar o fechamento correspondente (quando então envaziamos a pilha e retornamos à pilha anterior).
- Podemos fazer isso usando uma única pilha, “simulando” a abertura e fechamento de outras pilhas no seu interior...

Conversão para notação pós-fixo: algoritmo completo

- Devemos varrer a expressão infixa da esquerda para a direita,
  - se encontramos um operando, o colocamos na saída;
  - se encontramos um operador, o colocamos em uma pilha, desempilhando e colocando na saída os operadores na pilha até encontrarmos um operador com precedência menor ou uma abertura de parênteses;
    - Precedência: $+$ e $-$, seguida por $*$ e $/$, seguida por $\hat{}$.
  - se encontramos uma abertura de parênteses, colocamos na pilha;
  - se encontramos um fechamento de parênteses, desempilhamos e copiamos na saída os operadores na pilha, até a abertura de parênteses correspondente (que é desempilhada e descartada).
- Ao final, desempilhamos e colocamos na saída os operadores que restarem na pilha.

Exemplos

- $A/(B+C)*D$
- $((A-(B*C))+D)$
- $1-2\hat{}3-(4+5*6)*7$

Avaliação de expressões em notação infixa

- Combinando os dois algoritmos anteriores, podemos fazer a avaliação de uma expressão em notação infixa usando duas pilhas (uma para operadores, outra para operandos).
- Exemplo: $1 - 2 \hat{} 3 - (4 + 5 * 6) * 7$

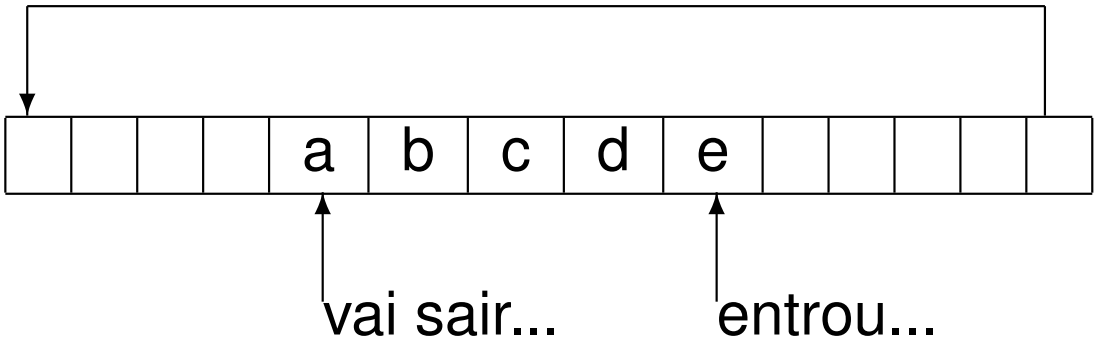
Entrada	Pilha Operadores	Pilha Operandos
1		1
–	–	1
2	–	1, 2
$\hat{}$	–, $\hat{}$	1, 2
3	–, $\hat{}$	1, 2, 3
–	–	-7
...	...	...

Filas

- Uma fila é uma estrutura em que o acesso é restrito ao elemento mais antigo.
- Operações básicas:
  - enqueue: inserir na fila;
  - dequeue: retirar da fila.

Arranjos circulares

A implementação mais comum de uma fila é por “arranjo circular”.



Implementação de Fila (1)

Fila baseada em sequência.

```
class fila_circular():
    def __init__(self):
        self._dados = [None]*10
        self._indiceVaiSair = 0
        self._indiceEntrou=len(self._dados)-1
        self._tamanho = 0

    def __len__(self):
        return self._tamanho
```

Implementação de Fila (2)

```
def _incrementa(self, indice):
        indice += 1
        if indice == len(self._dados):
            indice = 0
        return indice

    def _duplique_seq(self):
        novo = [None]*2*len(self._dados)
        for i in range (self._tamanho):
            novo[i] = self._dados[self._indiceVaiSa
            self._indiceVaiSair = self._incrementa(

        self._dados = novo
        self._indiceVaiSair = 0
        self._indiceEntrou = self._tamanho - 1;
```

Implementação de Fila (3)

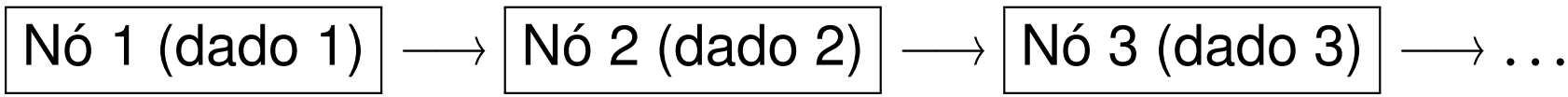
```
def enqueue(self, x):
    if self._tamanho == len(self._dados):
        self._duplique_seq()
    self._indiceEntrou = self._incrementa(self._
    self._dados[self._indiceEntrou] = x
    self._tamanho += 1

def dequeue(self):
    if self._tamanho == 0:
        raise IndexError("Fila vazia")

    self._tamanho -= 1
    x = self._dados[self._indiceVaiSair]
    self._dados[self._indiceVaiSair] = None
    self._indiceVaiSair = self._incrementa(self.
    return x
```

Lista Ligada

Uma alternativa a arranjos é a estrutura de lista ligada, na qual armazenamos dados em células interligadas.



Lista Ligada

- Esse tipo de estrutura é muito flexível e pode acomodar inserção e retirada de dados de locais arbitrários.
  - A vantagem desse tipo de estrutura é a flexibilidade permitida no uso da memória.
  - A desvantagem é que alocar memória é uma tarefa demorada (mais lenta que acesso a arranjos).

Implementação de Lista

Para definir uma lista ligada, precisamos primeiro definir o elemento armazenador (nó):

```
class NoLL:
    def __init__(self, x, proximo = None):
        self.dado = x
        self.proximo = proximo

    def __iter__(self):
        a = self
        while a:
            yield a.dado
            a = a.proximo
```

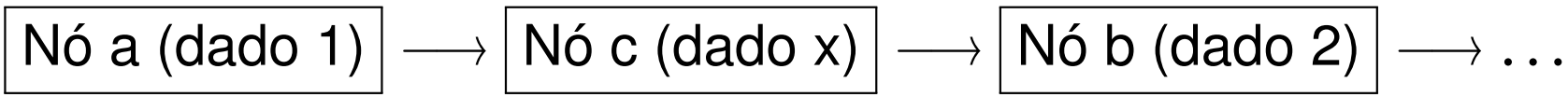
Inserção de nó

Considere inserir dado *x* após Nó a.


```
No c = NoLL(x, a.proximo)
a.proximo = c
```

Alternativamente,

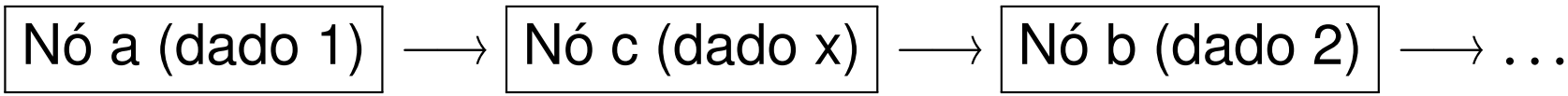
```
a.proximo = NoLL(x, a.proximo)
```



Remoção de nó, visita a sequência de nós

Remoção (do nó *seguinte*) ao nó a:

```
if a.proximo:
    a.proximo = a.proximo.proximo
```

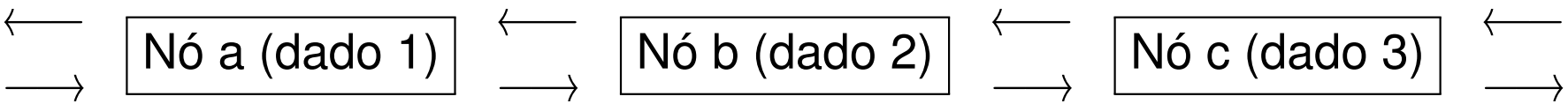


Para visitar todos os elementos de uma lista, de forma similar a um laço que percorre um arranjo:

```
No p =  lista.primeiro
while p:
    ...
    p = p.proximo
```

Lista Duplamente Encadeada

Uma estrutura interessante é o deque, composto por nós que apontam em duas direções:



Com essa estrutura é possível percorrer os dados em ambos os sentidos.

Usos de listas

A partir daí podemos implementar várias funcionalidades:

1. Pilhas;
2. Filas;
3. Vector: estrutura genérica de inserção/remoção em local arbitrário.

Implementação de Pilha usando Lista

```
class Pilha():
    def __init__(self):
        self.topo = None

    def push(self, x):
        n = NoLL(x, self.topo)
        self.topo = n

    def pop(self):
        if self.topo:
            a = self.topo.dado
            self.topo = self.topo.proximo
            return a
        else:
            raise IndexError("pop em pilha vazia")
```

Implementação de Fila usando Lista Ligada

```
class Fila():
    def __init__(self):
        self.saida = None
        self.entrada = None

    def enqueue(self, x):
        if self.saida:
            self.entrada.proximo = NoLL(x)
            self.entrada = self.entrada.proximo
        else:   # Fila vazia
            self.entrada = NoLL(x)
            self.saida = self.entrada

    def dequeue(self):
        if self.saida:
            a = self.saida.dado
```