Tipos Abstratos de Dados
PILHAS estáticas

Wanderley de Souza Alencar
adaptado por Raphael Guedes

Universidade Federal de Goiás - UFG
Instituto de Informática - INF

October 3, 2024

1 Conceito

2 Operações Fundamentais

3 Representações

4 Saiba Mais...

Pilha – no Dicionário

- Amontoado de coisas colocadas umas sobre as outras.
- Aparelho que transforma em corrente elétrica a energia produzida por uma reação química.
- Pilha atômica, reator nuclear que produz grande quantidade de energia pela fissão de um núcleo de urânio ou de plutônio.

Pilha – na Computação

É um tipo de estrutura de dados dinâmica que permite a realização de operações de inserção e de remoção de elementos (ou objetos) de tal maneira que se obedeça ao seguinte critério:

"*O elemento a ser removido é sempre aquele que, dentre todos os presentes na pilha em determinado instante, foi o último a ser nela inserido.*"

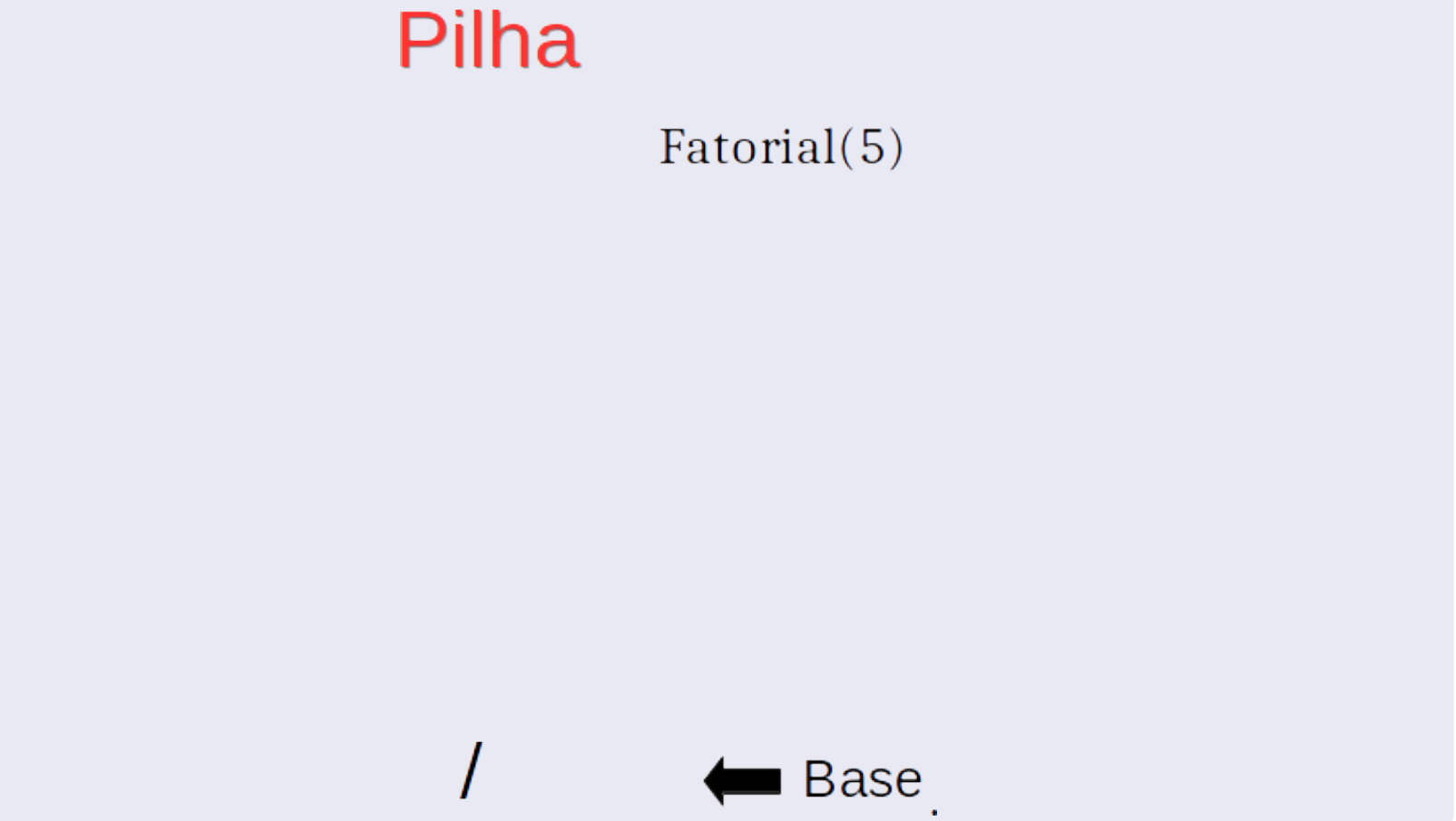
Pilha – na Computação

É um tipo de estrutura de dados, de tamanho variável, em que se estabelece a política LIFO (*Last In, First Out*) para a inserção e remoção de elementos nela(dela).

Pilha – na Computação

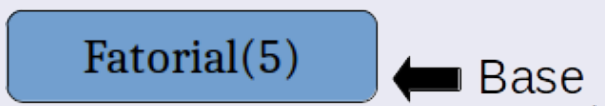
As *pilhas* são utilizadas, por exemplo, para controlar a execução de rotinas recursivas de um programa de computador.

Pilha –na Computação



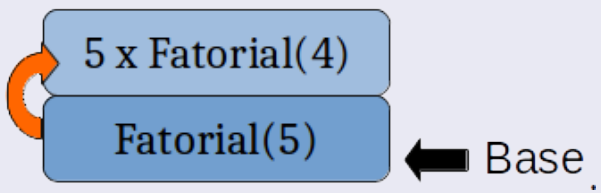
Pilha –na Computação

Pilha

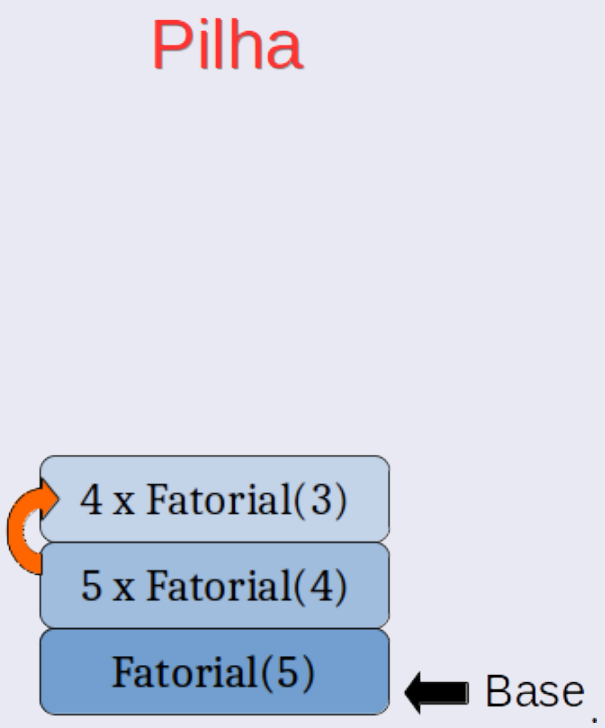


Pilha –na Computação

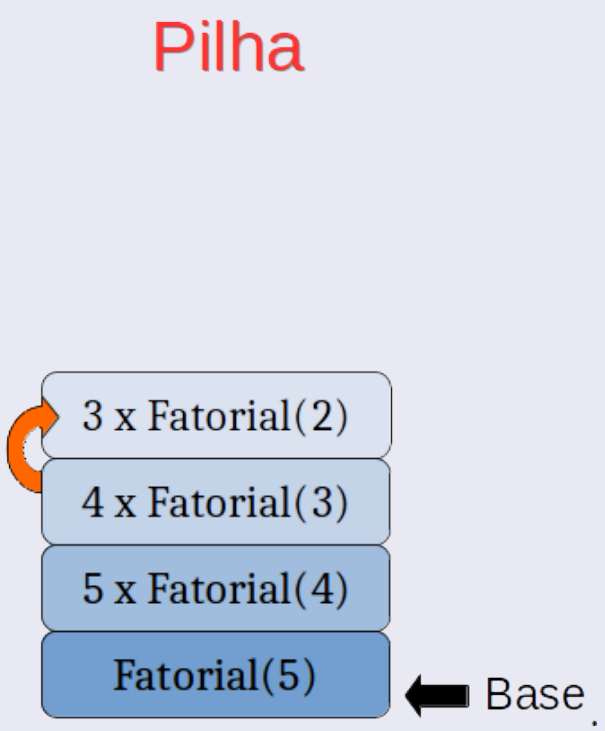
Pilha



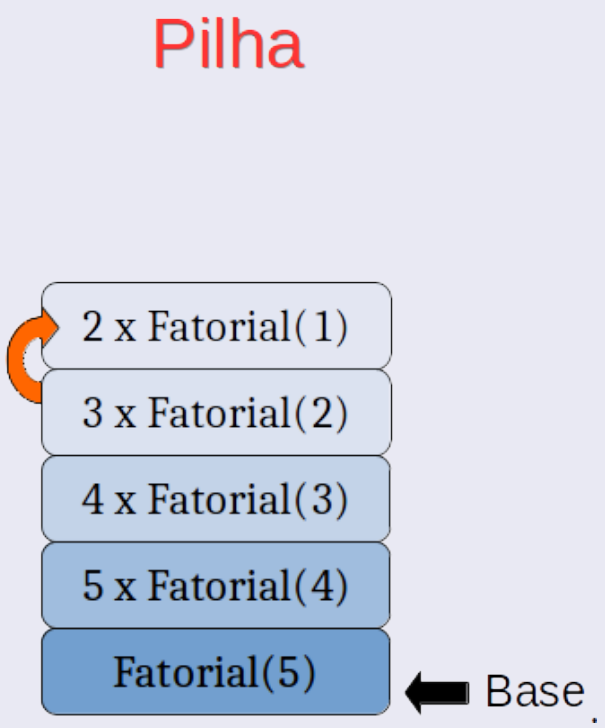
Pilha –na Computação



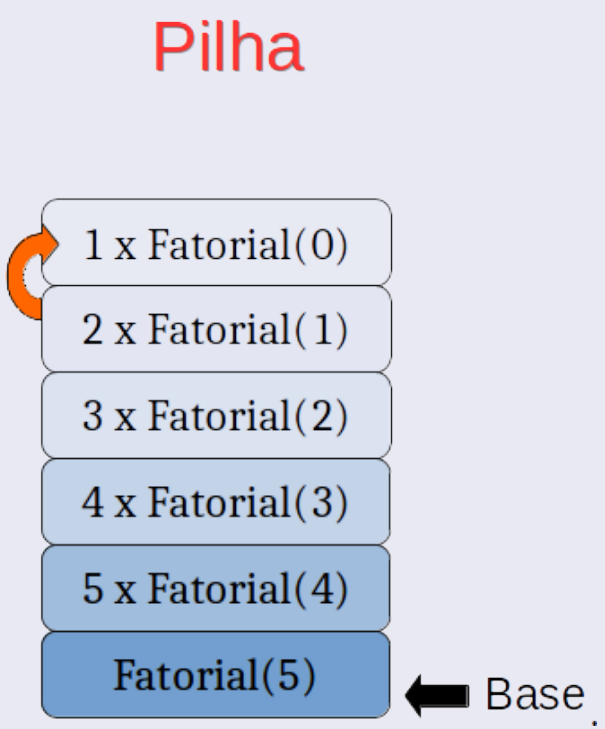
Pilha –na Computação



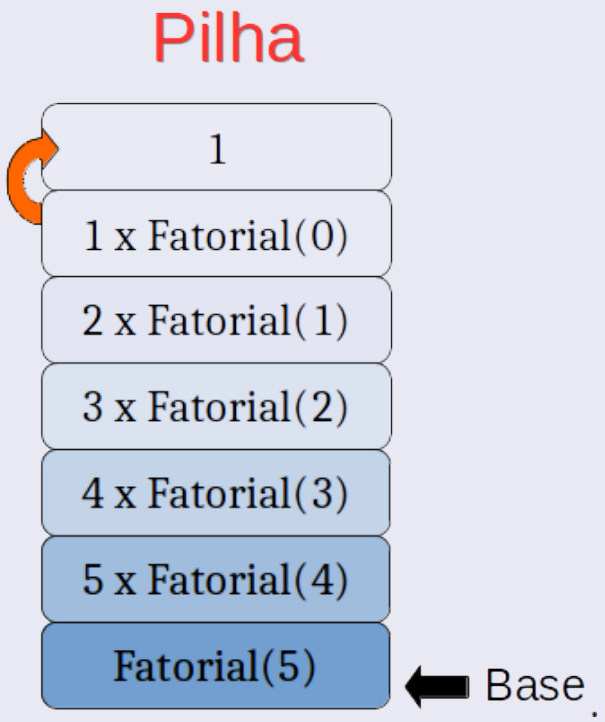
Pilha –na Computação



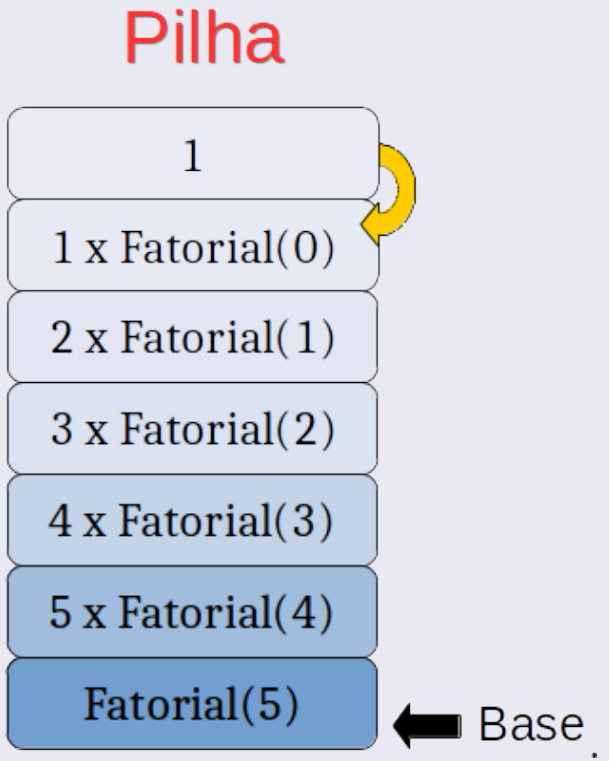
Pilha –na Computação



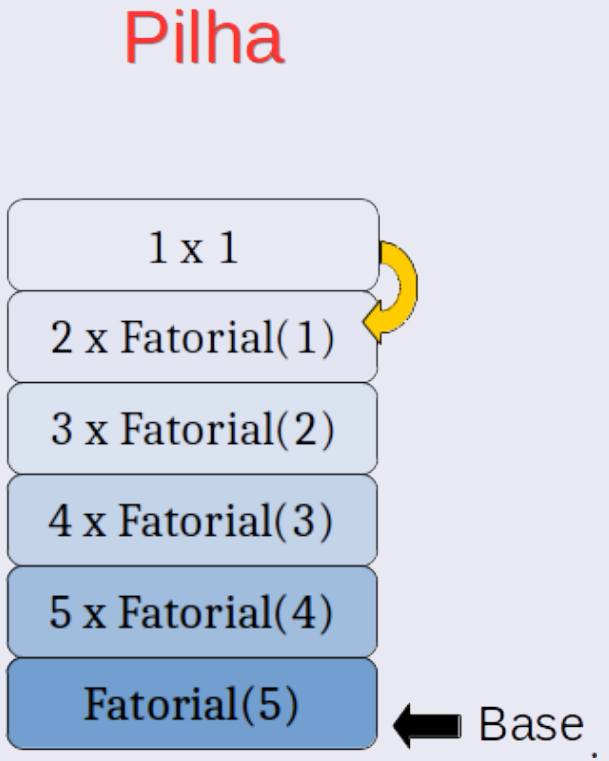
Pilha –na Computação



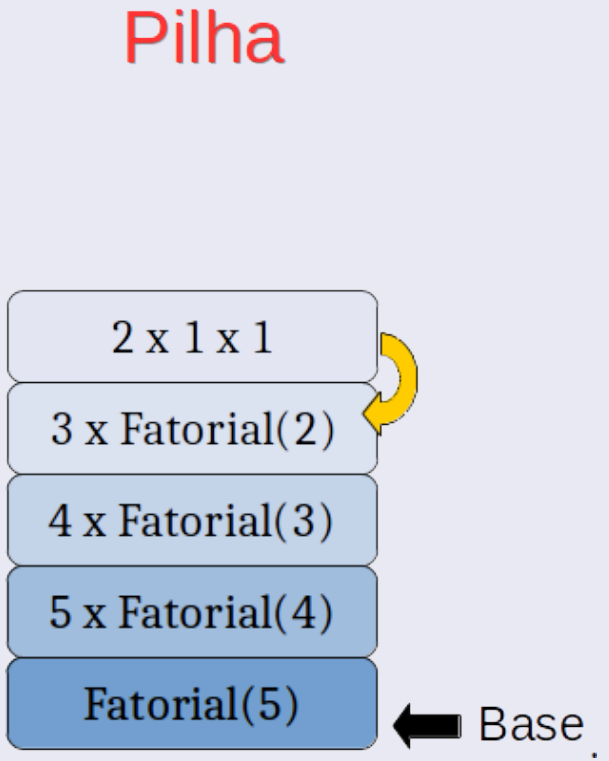
Pilha –na Computação



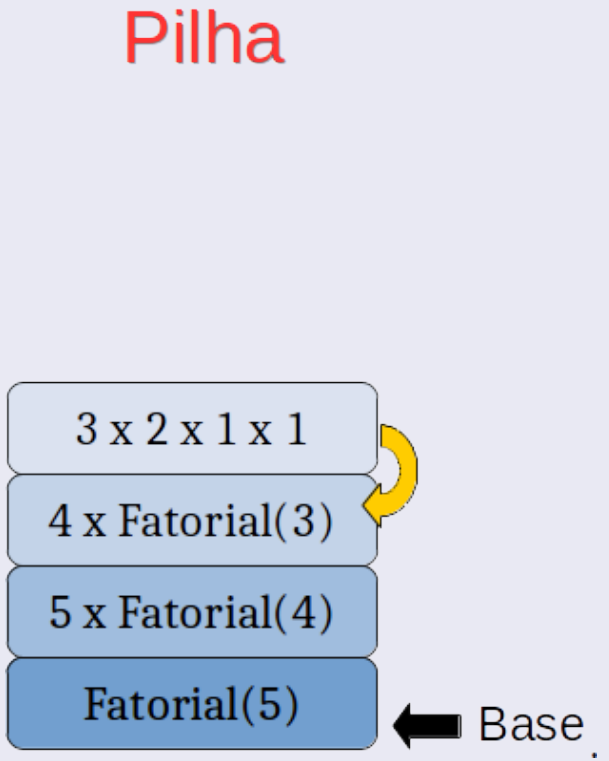
Pilha –na Computação



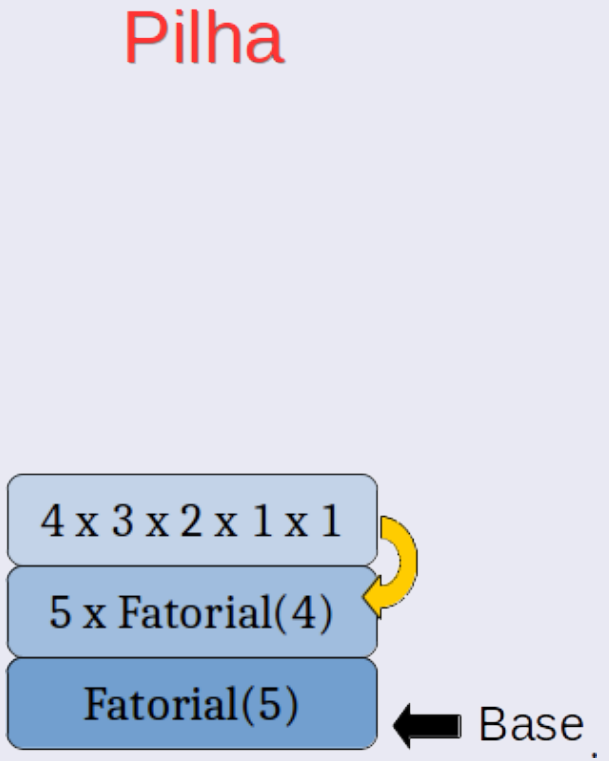
Pilha –na Computação



Pilha –na Computação

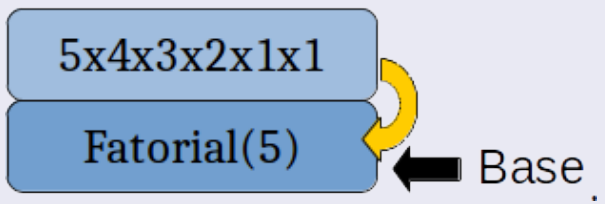


Pilha –na Computação



Pilha –na Computação

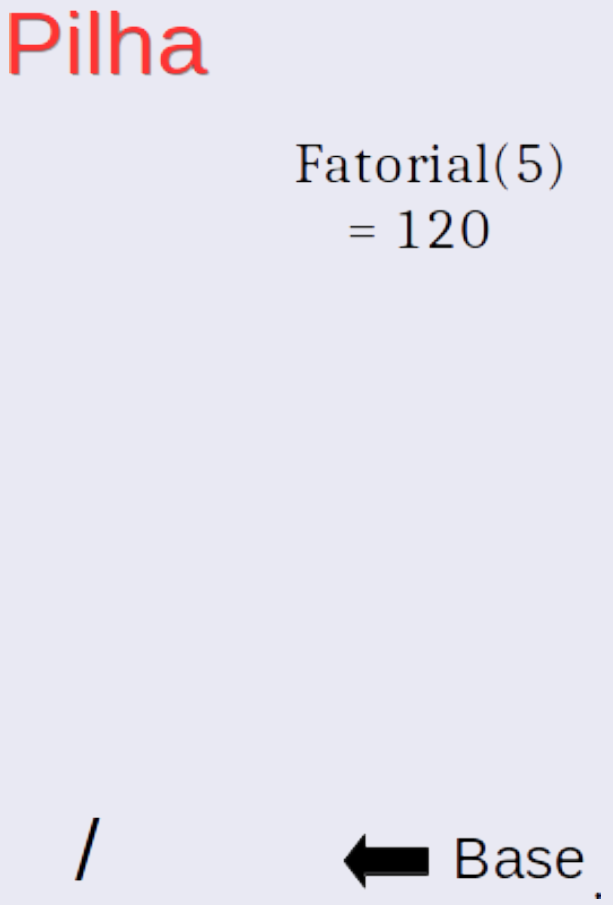
Pilha



Pilha –na Computação

Pilha

Pilha –na Computação

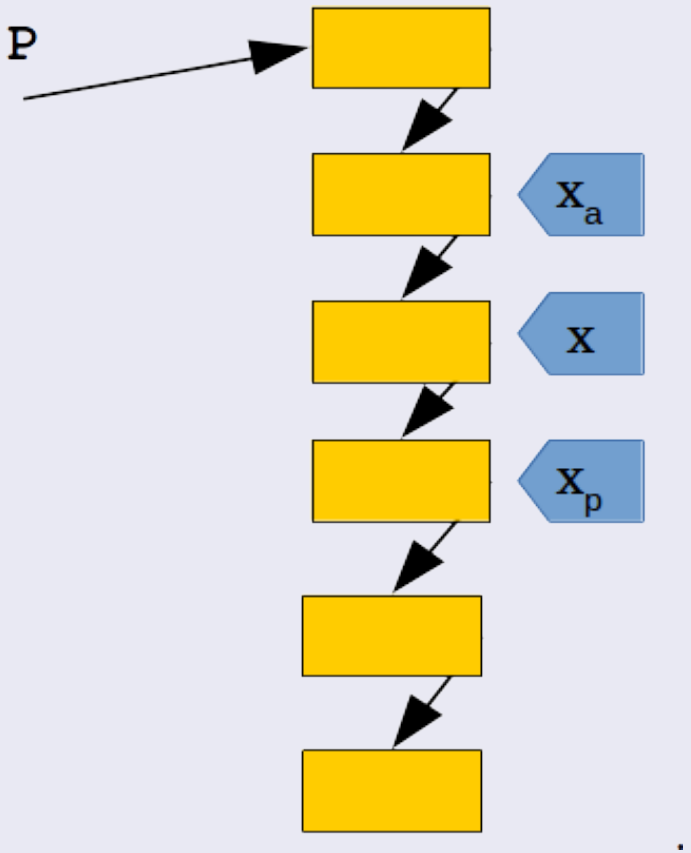


Pilha – na Computação

Uma pilha é uma estrutura de dados formada por uma sequência finita de elementos – células, componentes, nodos ou nós – organizada de tal maneira que reflita uma relação linear de ordem entre eles.

A *relação linear de ordem* define que um elemento x tenha um, e somente um, elemento x_a que lhe seja imediatamente anterior e outro, também único, x_p que lhe seja imediatamente posterior.

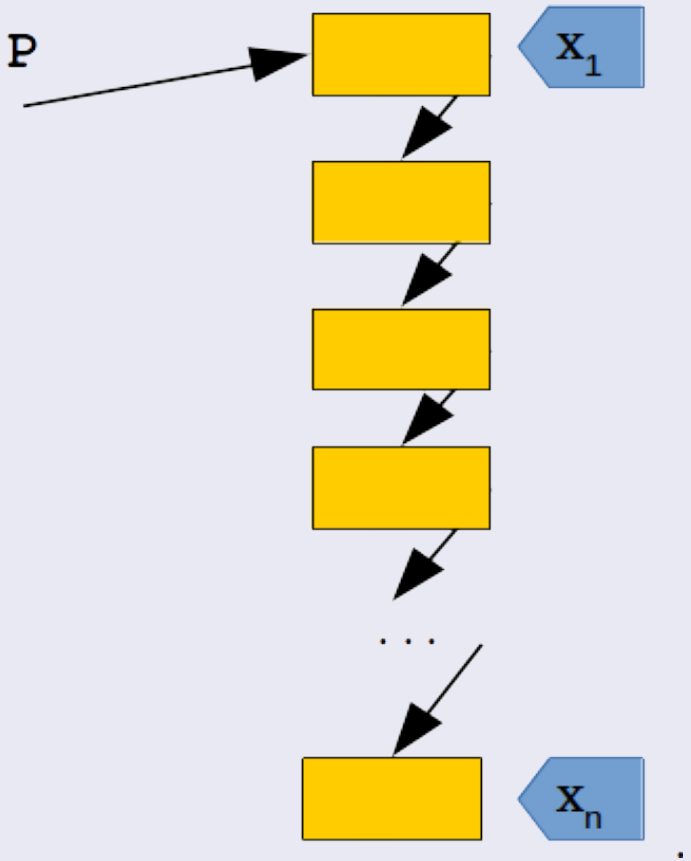
Pilha – na Computação



Pilha – na Computação

Evidentemente, o primeiro elemento da pilha, x_1, não possui elemento anterior e o último elemento, digamos x_n, não possui elemento posterior, considerando que $n \in \mathbb{N}^*$.

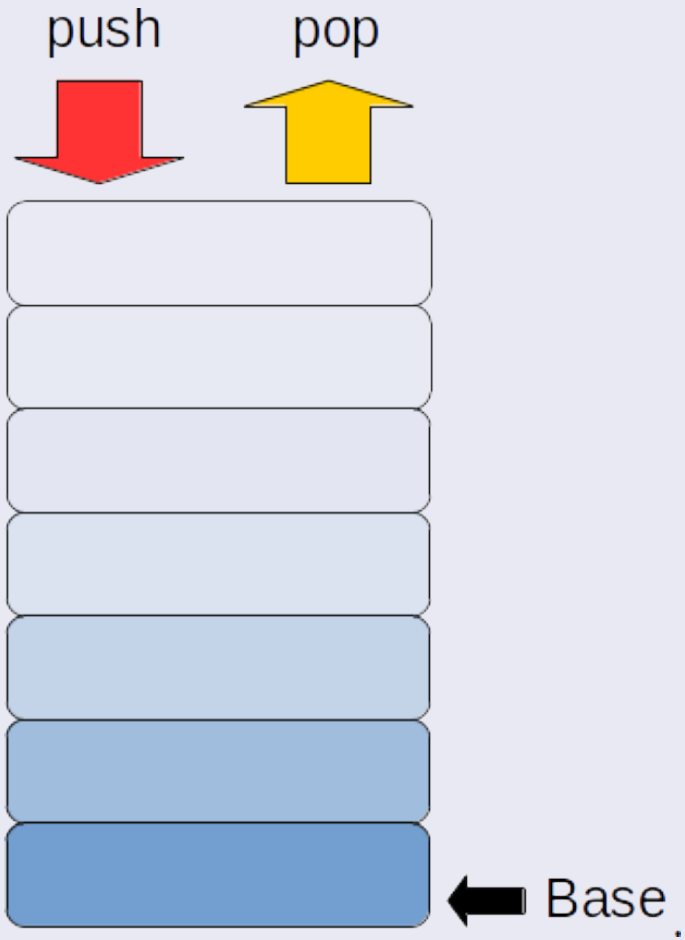
Pilha – na Computação



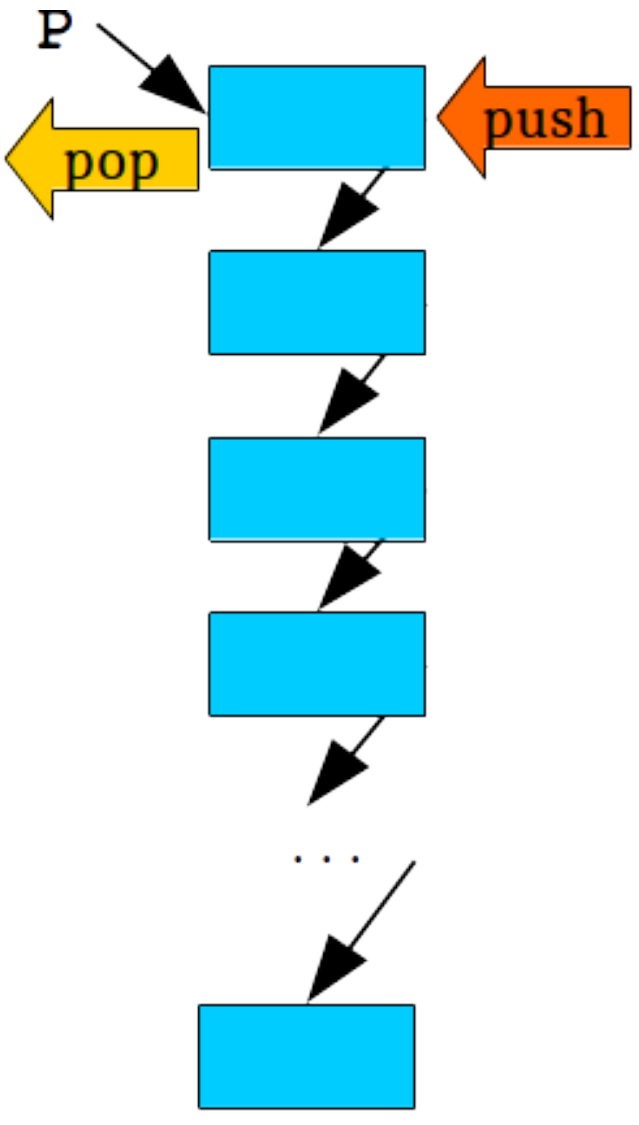
Pilha – na Computação

Para obedecer à política LIFO, as operações de *inserção* e *remoção* devem ser obrigatoriamente realizadas em "*na mesma extremidade*" da pilha.

Pilha – na Computação



Operações Fundamentais

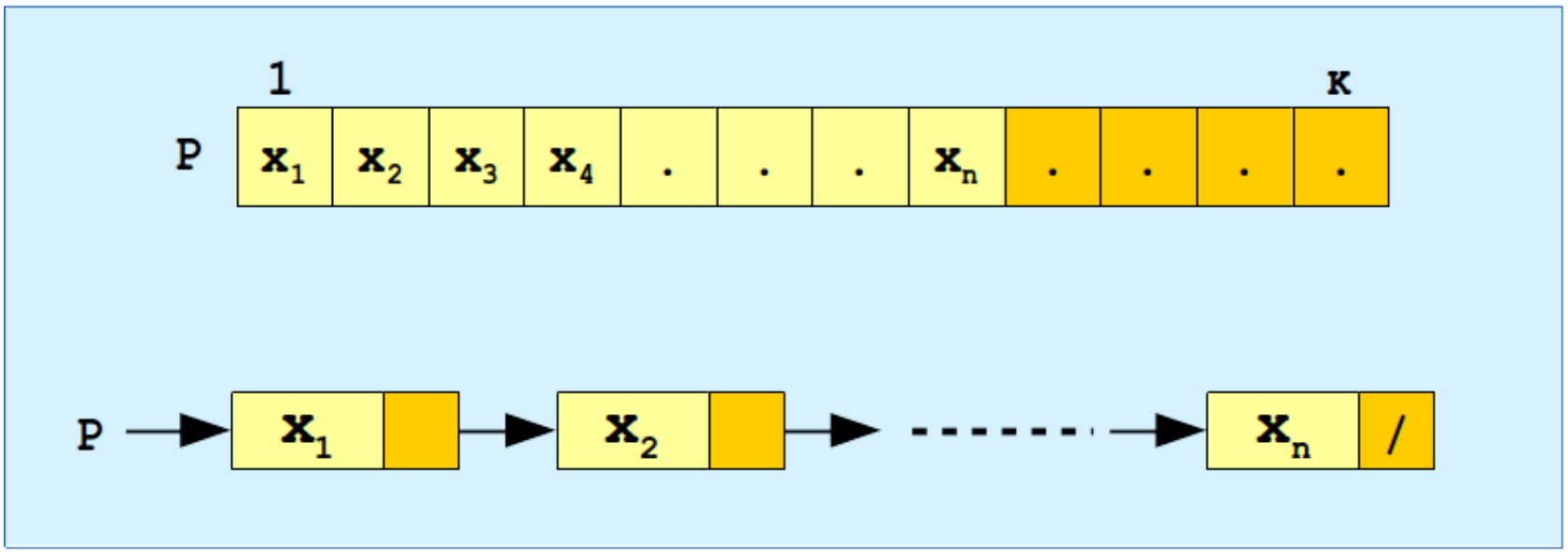


Operações na pilha

As operações fundamentais sobre pilhas são:

1. criar uma pilha (inicialmente *vazia*);
2. determinar a quantidade de elementos presentes na pilha;
3. mostrar um elemento da pilha;
4. inserir um elemento na pilha (*push*);
5. remover um elemento da pilha (*pop*).

Representações



Representações da Pilha

Há, esssencialmente, duas maneiras para representar computacionalmente uma pilha:

1. por contiguidade (alocação estática);
2. por encadeamento (alocação dinâmica).

Representações

A escolha da maneira mais conveniente a ser utilizada deve ser balizada por fatores como:

1. qual o tamanho do armazenamento exigido por elemento da pilha?
2. quais os tamanhos, mínimo e máximo, esperados para a pilha?
3. qual a frequência de realização de cada operação prevista?

Por Contiguidade

A proposta básica desta implementação é a sua simplicidade: utilizar um vetor para conceber um TAD que represente a estrutura de dados *pilha*.

A implementação é facilitada, mas paga-se o preço ao optar por previamente alocar um vetor que, nem sempre, está sendo completamente utilizado pela pilha ou que pode, em certo momento, ser insuficiente para armazenar a quantidade elementos da pilha.

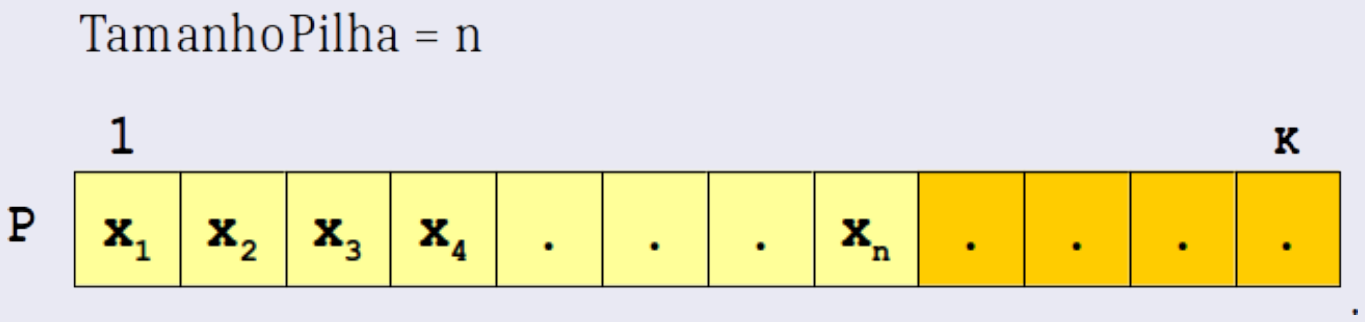
Por Contiguidade

Dada uma pilha $P = (x_1, x_2, \ldots, x_n)$, com $n \in \mathbb{N}^*$, utiliza-se um vetor para representá-la, além de duas variáveis de controle:

- `TamanhoPilha`: conterá o número de elementos presentes na pilha naquele momento;
- `TamanhoMaximoPilha`: conterá o número máximo de elementos que a pilha poderá conter, correspondendo ao tamanho do vetor que foi definido para representá-la, digamos, $\kappa \in \mathbb{N}^*$.

Por Contiguidade

Vetor, de tamanho κ, representando uma pilha *P*:



Por Contiguidade

Note que:

- a pilha ocupa as posições de 1 a n do vetor;
- as posições de $(n+1)$ a κ correspondem à área livre do vetor, ou seja, área destinada ao crescimento da pilha e que está previamente alocada;
- a pilha está limitada ao tamanho do vetor, que é κ;

Por Contiguidade

Note que para conservar a política LIFO, é necessário que:

- uma inserção seja realizada na posição $(n+1)$, desde que $n < \kappa$;
- a remoção deve ser realizada na posição (n), desde que $n \geq 1$.

Declaração das estruturas para pilha, por contiguidade:

```
1  //
2  // Arquivo pilha.h (parte 01)
3  //
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  #define SUCESSO 1
8  #define FALHA -1
9  #define CHAVE_INVALIDA 0
10
11 #define TAMANHO_MAXIMO_PILHA 100
```

Declaração das estruturas para pilha, por contiguidade:

```
1  //
2  // Arquivo pilha.h (parte 02)
3  //
4  typedef struct {
5        unsigned int chave;
6        unsigned int dado;
7  } Elemento;
8
9  typedef struct pilha {
10     Elemento elementos [TAMANHO_MAXIMO_PILHA];
11     unsigned int tamanho;
12 } Pilha;
```

Declaração das estruturas para pilha, por contiguidade:

```
1 //
2 // Arquivo pilha.h (parte 03)
3 //
4 int criarPilhaVazia(Pilha * pilha);
5 int tamanhoPilha(Pilha pilha);
6 void mostrarElemento(Elemento elemento);
7 void mostrarPilha(Pilha pilha);
8 int push(Pilha * pilha, Elemento elemento);
9 Elemento pop(Pilha * pilha);
```

Criar uma pilha inicialmente vazia, por contiguidade:

```
1  int criarPilhaVazia(Pilha * pilha) {
2      pilha ->tamanho = 0;
3      return (SUCESSO);
4  }
5  //
6  // ou
7  //
8  int criarPilhaVazia(Pilha * pilha) {
9      int i;
10     pilha ->tamanho = 0;
11     for (i = 0; (i < TAMANHO_MAXIMO_PILHA); i++) {
12         pilha -> elementos[i].chave = CHAVE_INVALIDA;
13     }
14     return (SUCESSO);
15 }
```

Representações

Determinar o tamanho da pilha, por contiguidade:

```
1 int tamanhoPilha(Pilha pilha ) {
2
3       if (pilha.tamanho >= 0) {
4           return(pilha.tamanho);
5       }
6       else {
7           return(FALHA);
8       }
9 }
```

Mostrar um elemento da pilha, por contiguidade:

```
1 void mostrarElemento(Elemento elemento) {
2     printf("Chave.....: %u\n", elemento.chave);
3     printf("Dado.....: %u\n", elemento.dado);
4 }
```

Mostrar toda a pilha, por contiguidade:

```
1  void mostrarPilha(Pilha pilha) {
2      unsigned int i;
3
4      if (pilha.tamanho == 0) {
5          printf("Atencao: A pilha esta vazia.\n");
6      }
7      else {
8          printf("A pilha possui %u elementos.\n\n", pilha.tamanho);
9          for (i = 0; (i < pilha.tamanho); i++) {
10             printf("Elemento n.: %u\n", (i+1));
11             mostrarElemento(pilha.elementos[i]);
12         }
13     }
14 }
```

Inserir elemento na pilha, por contiguidade:

```
1  int push (Pilha * pilha , Elemento elemento) {
2      unsigned int i;
3      Elemento auxiliar;
4
5      if (pilha->tamanho == TAMANHO_MAXIMO_PILHA) {
6          return(FALHA);
7      }
8      else {
9          pilha-> elementos[pilha->tamanho] = elemento;
10         pilha-> tamanho++;
11         return(SUCESSO);
12     }
13 }
```

Remover elemento da pilha, por contiguidade:

```
1  Elemento pop (Pilha * pilha) {
2      unsigned int i;
3      Elemento elementoResultado;
4
5      if (pilha-> tamanho == 0) {
6          elementoResultado.chave = CHAVE_INVALIDA;
7          return(elementoResultado);
8      }
9      else {
10         elementoResultado = pilha-> elementos[((pilha-> tamanho) - 1)];
11         pilha-> tamanho--;
12         return(elementoResultado);
13     }
14 }
```

Saiba Mais...

Referência 1

SCZWARCFITER, Jayme Luiz e MARKEZON, Lilian; *Estruturas de dados e seus algoritmos*; 1ª ed.; Rio de Janeiro: LTC; 1994.
Capítulo 06.

Referência 2

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. and STEIN, C.; *Algoritmos* – traduação da 2 edição norte-americana; Elsevier; 2002.
Capítulos: 12, 19 e 20.

Saiba Mais...

Referência 3

FORBELLONE, André Luiz V. e EBERSPACHER, Henri Frederico; *Lógica de programação – a construção de algoritmos e estruturas de dados*; 3 ed.; São Paulo: Prentice Hall do Brasil; 2005.
Capítulo 07.

Saiba Mais...

Referência 4

GOODRICH, Michael T. e TAMASSIA, Roberto; *Estruturas de dados e algoritmos em Java*; 4 ed.; Porto Alegre: Bookman; 2007.
Capítulos: 03, 05, 06 e 08.

Saiba Mais...

Referência 5

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. and STEIN, C.; *Introduction to Algorithms*; 3^{rd} edition; MIT Press; 2009. *Chapters: 10, 12 and 19*.

Saiba Mais...

Referência 6

SEDGEWICK, R. and WAYNE, K.; *Algorithms*; 4^{th} edition; Addison-Wesley; 2011.
Chapters: 03 and 05.

Saiba Mais...

Referência 7

STEPHENS, Rod; *Essential Algorithms: A Practical Approach to Computer Algorithms*; 1^{st} edition; John Wiley & Sons; 2013. *Chapter 10*.

Saiba Mais...

Referência 8

FERRARI, Roberto; RIBEIRO, Marcela X.; DIAS, Rafael L. e FALVO, Maurício; *Estruturas de dados com jogo*; 1 ed.; São Paulo: Elsevier; 2014.
Capítulos: 03, 04 e 05.