



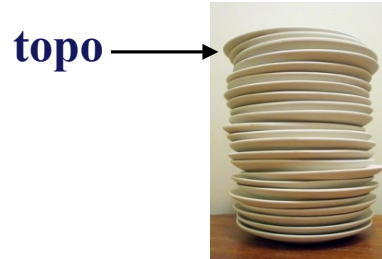
Programação II

Pilhas (*stacks*)

Bruno Feijó
Dept. de Informática, PUC-Rio

Pilha

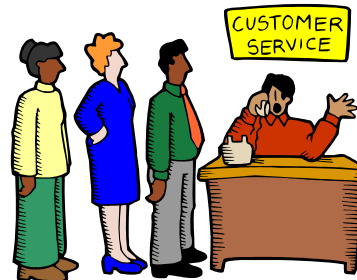
- Novo elemento é inserido no topo e acesso é apenas ao topo
 - ... como numa pilha de pratos



- O único elemento que pode ser acessado e removido é o do topo
- Na Pilha (*Stack*), os elementos são retirados na ordem inversa à ordem em que foram colocados, i.e.:
 - o 1o. que sai é o último que entrou (**LIFO** – *last in, first out*)

- Na estrutura Fila (*Queue*) é o inverso:

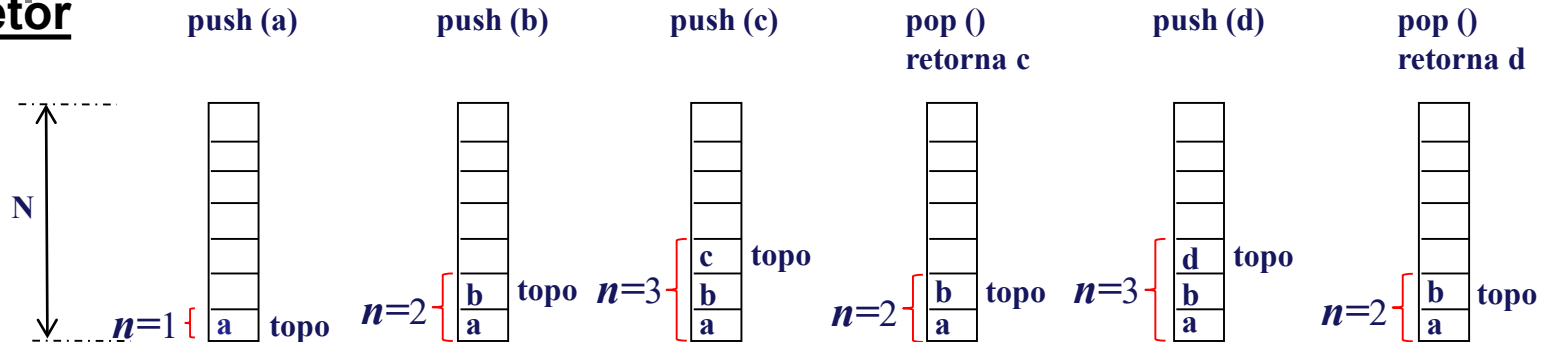
- **FIFO** – *first in, first out*
- i.e.: 1o. a chegar, primeiro a sair



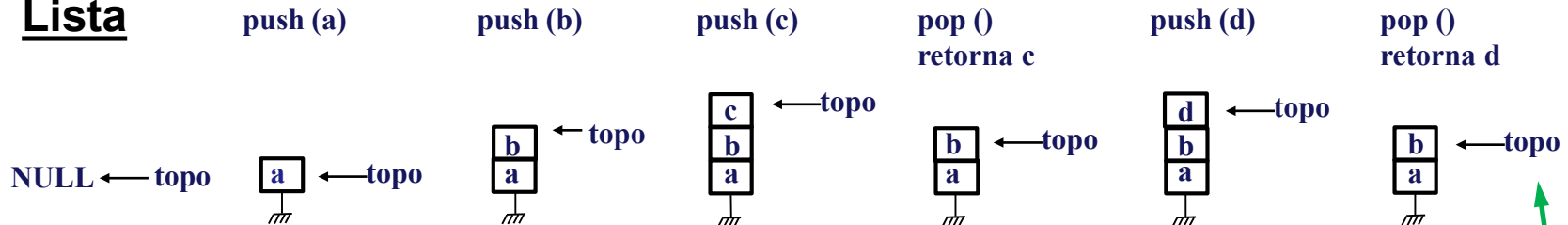
- Operações básicas
 - Empilhar (**push**) um novo elemento, inserindo-o no topo
 - Desempilhar (**pop**) um elemento, removendo-o do topo

Pilha como vetor (*array*) ou lista encadeada

Vetor

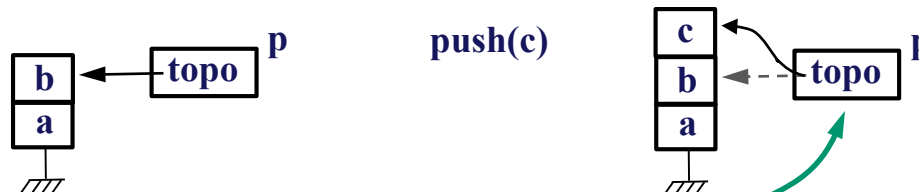


Lista



No caso acima, a pilha é representada pelo ponteiro *topo* (ou *top*).

Porém, é mais geral e fácil colocar o ponteiro para o topo da pilha dentro de uma estrutura chamada “pilha” (que representará a pilha). “p” representa a pilha.



Representar a pilha desta forma (com uma nova estrutura contendo apenas um ponteiro, ao invés de uma simples variável apontando para a lista) é necessária para definir serviços gerais de pilhas.

Um exercício chave! ... pilha de *int*

- Agora que você sabe o que é uma pilha e o que são as operações *push* e *pop*, tente (**sem ver o próximo slide**) definir as estruturas e os serviços (funções) necessários para termos um tipo Pilha de *int* como sendo uma estrutura e implementado como lista. Depois você fará a implementação como vetor.
 - Você vai precisar definir elemento de lista e, depois, definir a Pilha como uma estrutura de apenas um componente (que é o ponteiro *topo* para um elemento de lista)
 - Organize o seu código com o typedef da Pilha e os protótipos das funções de serviço reunidos no início do programa. E, logo depois, defina as estruturas.

O conjunto “typedef e protótipos” é chamado de **INTERFACE**. Ele informa tudo o que você precisa saber sobre um determinado tipo (i.e. o nome do tipo e as suas funções de serviço).

```
#include <...>
...
typedef struct pilha Pilha;
protótipo de pilha_cria
protótipo de pilha_vazia ←
protótipo de pilha_push
protótipo de pilha_pop
protótipo de pilha_libera

struct elemento
{ ... };
typedef struct elemento Elemento;

struct pilha
{ ... };
```

pilha_vazia
testa se uma
pilha é vazia

Tipo “Pilha de *int*” como lista: interface e funções

Interface: typedef e protótipos

```
typedef struct pilha Pilha;  
Pilha * pilha_cria(void);      // Constructor  
void pilha_libera(Pilha * p); // Destructor  
int pilha_vazia(Pilha * p);  
int pilha_pop(Pilha * p); // Get & Destructor  
void pilha_push(Pilha * p, int a); // Set
```

Poderíamos também incluir **peek()** que é um **pop()** sem remoção.

Uma alternativa, menos interessante, seria definir **top()** que apenas retorna o endereço do topo da pilha e **pop()** que apenas remove o topo (i.e. um *destructor* apenas).

pop() e **libera()** no próximo slide

Estruturas e funções:

```
struct elemento  
{  
    int info;  
    struct elemento * prox;  
};  
typedef struct elemento Elemento;
```

```
Pilha * pilha_cria(void)  
{  
    Pilha * p =  
        (Pilha *) malloc(sizeof(Pilha));  
    p->topo = NULL;  
    return p;  
}
```

```
struct pilha  
{  
    Elemento * topo;  
};
```

```
void pilha_push(Pilha * p, int a)  
{  
    Elemento * t=(Elemento *) malloc(sizeof(Elemento));  
    if (t==NULL) exit(1);  
    t->info= a;  
    t->prox= p->topo;  
    p->topo= t;  
}
```

```
int pilha_vazia(Pilha * p)  
{  
    return (p->topo == NULL);  
}
```

Tipo “Pilha de *int*” como lista – funções (continuação)

```
int pilha_pop(Pilha * p)
{
    Elemento * t;
    int a;
    if (pilha_vazia(p))
    {
        printf("Pilha vazia.\n");
        exit(1);
    }
    t = p->topo;
    a = t->info;
    p->topo = t->prox;
    free(t);
    return a;
}
```

```
void pilha_libera(Pilha * p)
{
    Elemento * t, * q = p->topo;
    while (q != NULL)
    {
        t = q->prox;
        free(q);
        q = t;
    }
    free(p);
}
```

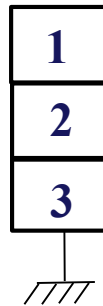
ou:

```
assert(!pilha_vazia(p));
requer assert.h
```

`assert(expressão);` se *expressão* não é verdade, a execução termina e mensagem de erro identifica arquivo, linha, função e condição que foi violada.

Exemplo de uso da Pilha

- Vamos, agora, usar a Pilha (com suas funções de serviço).
- Escreva a função `int soma(Pilha * p)` que soma os elementos retirados da pilha, **sem olhar a solução no próximo slide**. Teste criando a seguinte pilha (chamando 3 vezes a função `pilha_push`):



- Lembre de liberar a pilha no final da *main*.
- A função retorna 6

Exemplo de uso da Pilha - Solução

```
int soma(Pilha * p) // soma elementos da pilha
{
    int s = 0;
    while (!pilha_vazia(p))
        s += pilha_pop(p);
    return s;
}

int main(void)
{
    Pilha * p = pilha_cria();
    pilha_push(p, 3);
    pilha_push(p, 2);
    pilha_push(p, 1);
    printf("soma = %d\n", soma(p));
    pilha_libera(p);
    return 0;
}
```


Resumindo a INTERFACE de pilha de *int* :

```
typedef struct pilha Pilha;  
  
Pilha * pilha_cria(void) ;  
  
void pilha_libera(Pilha * p) ;  
  
int pilha_vazia(Pilha * p) ;  
  
void pilha_push(Pilha * p, int a) ;  
  
int pilha_pop(Pilha * p) ;
```

- função **pilha_cria**
 - aloca dinamicamente a estrutura da pilha
 - inicializa seus campos e retorna seu ponteiro
- função **pilha_libera**
 - destrói a pilha, liberando toda a memória usada pela estrutura
- função **pilha_vazia**
 - informa se a pilha está ou não vazia
- funções **pilha_push** e **pilha_pop**
 - inserem e retiram, respectivamente, um valor inteiro na pilha
 - pop() em pilha vazia lança exceção (stack exception)

Pilha como **v**etor

NÃO SE ALTERAM:

- interface: typedef e protótipos
- struct *elemento*

ALTERAM-SE:

- struct *pilha*
- corpos de funções

```
struct pilha
{
    Elemento * topo;
};
typedef struct pilha Pilha;
```

Pilha como lista

```
Pilha * pilha_cria(void)
{
    Pilha * p =
        (Pilha *) malloc(sizeof(Pilha));
    p->topo = NULL;
    return p;
}
```

```
void pilha_push(Pilha * p, int a)
{
    Elemento * t=(Elemento *) malloc(sizeof(Elemento));
    if (t==NULL) exit(1);
    t->info= a;
    t->prox= p->topo;
    p->topo= t;
}
```

Pilha como vetor:

```
# define N 50
```

```
struct pilha
{
    int n;
    int v[N];
};
typedef struct pilha Pilha;
```

N é o número máximo de elementos
n: número de elementos no vetor
v[n]: primeira posição livre do vetor
v[n-1]: topo da pilha

```
void pilha_push(Pilha * p, int a)
{
    if (p->n==N) exit(1);
    p->v[p->n]=a;
    p->n++;
}
```

```
Pilha * pilha_cria(void)
{
    Pilha * p = (Pilha *) malloc(sizeof(Pilha));
    if (p==NULL) return NULL;
    p->n = 0;    // inicializa com 0 elementos
    return p;
}
```



Pilha como vetor

```
int pilha_vazia(Pilha * p)
{
    return (p->n == 0);
}
```

```
int pilha_pop(Pilha * p)
{
    int a;
    if (pilha_vazia(p)) exit(1);
    a = p->v[p->n-1];
    p->n--;
    return a;
}
```

```
void pilha_libera(Pilha * p)
{
    free(p);
}
```

Interface de pilha de *int* não se altera!

```
typedef struct pilha Pilha;  
  
Pilha * pilha_cria(void);  
  
void pilha_libera(Pilha * p);  
  
int pilha_vazia(Pilha * p);  
  
void pilha_push(Pilha * p, int a);  
  
int pilha_pop(Pilha * p);
```

- função **pilha_cria**
 - aloca dinamicamente a estrutura da pilha
 - inicializa seus campos e retorna seu ponteiro
- função **pilha_libera**
 - destrói a pilha, liberando toda a memória usada pela estrutura
- função **pilha_vazia**
 - informa se a pilha está ou não vazia
- funções **pilha_push** e **pilha_pop**
 - inserem e retiram, respectivamente, um valor inteiro na pilha
 - pop() em pilha vazia lança exceção (stack exception)

Exemplo de pilha como vetor

- Isole a parte do código do exemplo anterior que contém as estruturas e as funções de serviço, usando `/* ... */` (indicada pelo retângulo vermelho na figura abaixo)
- Escreva as novas estruturas e as funções de serviço, usando vetor ao invés de lista simplesmente encadeada.
- Execute `int soma(Pilha * p)`

Única parte do código que muda

```
#include <stdio.h>
#include <stdlib.h>

typedef struct pilha Pilha;
Pilha * pilha_cria(void);
void pilha_libera(Pilha * p);
int pilha_vazia(Pilha * p);
int pilha_pop(Pilha * p);
void pilha_push(Pilha * p, int a);

/*
typedef struct elemento Elemento;
struct elemento
{
int info;
...
*/
int soma(Pilha * p)
{ ... }
int main(void)
{ ... }
```

Um desafio: separe os códigos em 3 módulos

- Coloque a INTERFACE em pilha.h, as estruturas e as funções em pilha.c, e as funções `int soma(Pilha * p)` e `main` em myProg.c. Note que a INTERFACE é inserida nos módulos .c através de `#include "pilha.h"`.
- Refaça para pilha como vetor e note que apenas o módulo pilha.c muda.

A INTERFACE é a especificação do módulo, que provê informação aos clientes sobre a funcionalidade do módulo.

pilha.h

pilha.c

```
#include <stdio.h>
#include <stdlib.h>
#include "pilha.h" ←
struct elemento
{
    ...
};
typedef struct elemento Elemento;

struct pilha
{
    Elemento * topo;
};

Pilha * pilha_cria(void)
{ ... }
```

```
typedef struct pilha Pilha;
Pilha * pilha_cria(void);
void pilha_libera(Pilha * p);
int pilha_vazia(Pilha * p);
int pilha_pop(Pilha * p);
void pilha_push(Pilha * p, int a);
```

myProg.c

```
#include <stdio.h>
#include <stdlib.h>
#include "pilha.h" ←

int soma(Pilha * p)
{ ... }

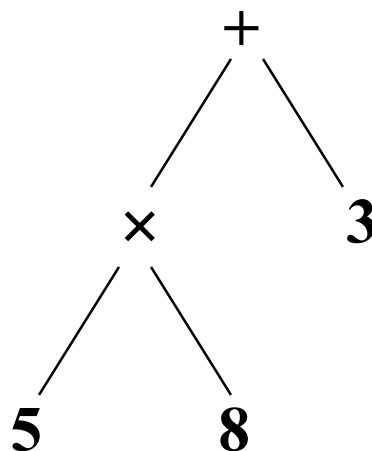
int main(void)
{ ... }
```



CALCULADORA PÓS-FIXADA

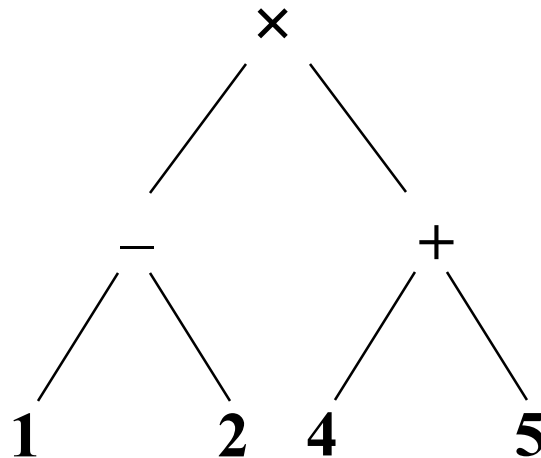
Notações para Expressões Aritméticas

- **Infixa (*infix*):** operador entre operandos
 - $(5 \times 8) + 3$
- **Pós-fixa (*posfix*):** operador após operandos (tipo calculadora HP)
 - $3\ 5\ 8\ \times\ +$ ou $5\ 8\ \times\ 3\ +$
- **Pré-fixa (*prefix*):** operador antes dos operandos
 - $+ \times 5\ 8\ 3$
- **Todas equivalentes à seguinte árvore:**



Outro Exemplo de Expressões Aritméticas

- Infixa (*infix*): $(1 - 2) \times (4 + 5)$
- Pós-fixa (*posfix*): $1\ 2 -\ 4\ 5 +\ \times$
- Pré-fixa (*prefix*): $\times -\ 1\ 2 +\ 4\ 5$
- Todas equivalentes à seguinte árvore:





Calculadora Pós-fixada

- Cada operando é empilhado numa pilha de valores
- quando se encontra um operador
 - desempilha-se o número apropriado de operandos
 - dois para operadores binários e um para operadores unários
 - realiza-se a operação devida
 - empilha-se o resultado

Exemplo Calculadora Pós-fixada

empilhe os valores 1 e 2	1 2 - 4 5 + *	<div>2</div> <div>1</div>
quando aparece o operador “-”	1 2 - 4 5 + *	
desempilhe 1 e 2		<div></div>
empilhe -1, o resultado da operação (1 - 2)		<div>-1</div>
empilhe os valores 4 e 5	1 2 - 4 5 + *	<div>5</div> <div>4</div> <div>-1</div>
quando aparece o operador “+”	1 2 - 4 5 + *	
desempilhe 4 e 5		<div>-1</div>
empilhe 9, o resultado da operação (4+5)		<div>9</div> <div>-1</div>
quando aparece o operador “*”	1 2 - 4 5 + *	
desempilhe -1 e 9		<div></div>
empilhe -9, o resultado da operação (-1*9)		<div>-9</div>

Programa Calculadora hp: calc.h

Os códigos neste slide e nos dois próximos slides estão organizados por módulos .h e .c, mas (se quiser) você pode incluir tudo em um único módulo.

calc.h

```
#include "pilha.h" // ou escreva direto typedef e protótipos

struct calc
{
    char f[21]; // formato para impressao (e.g. "%.2f\n")
    Pilha * p;  // pilha de operandos
};

typedef struct calc Calc;

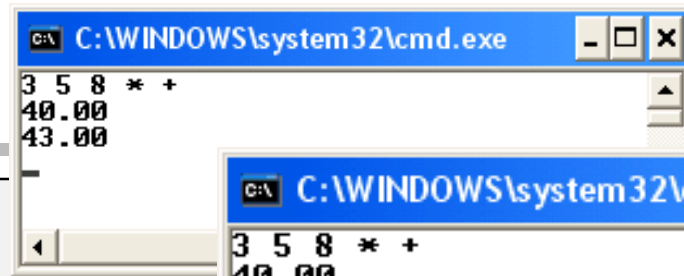
Calc * calc_cria(char * f);
void calc_operando(Calc * c, float valor);
void calc_operador(Calc * c, char op);
void calc_libera(Calc * c);
```

hp.c

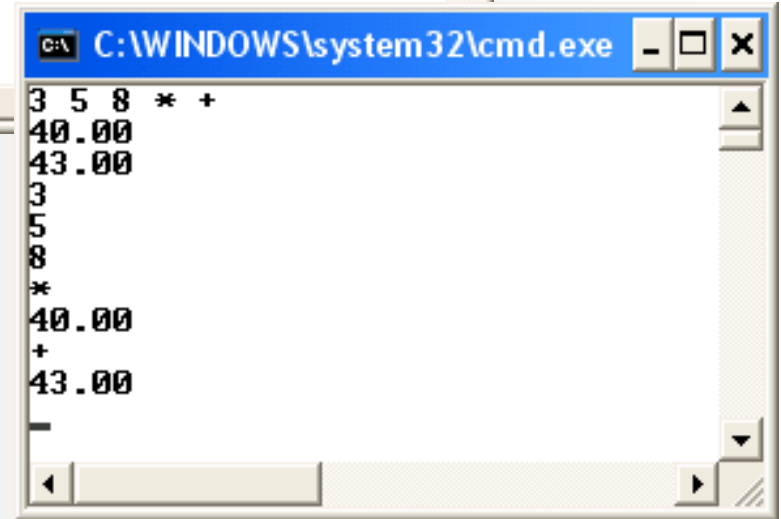
```
#include <stdio.h>
#include "calc.h"

int main(void)
{
    char c;
    float valor;
    Calc * calc;

    calc = calc_cria("%.2f\n");
    do
    {
        scanf(" %c",&c);    // le^ proximo caractere nao-branco
        if (c=='+' || c=='-' || c=='*' || c=='/')
            calc_operador(calc,c);
        else
        {
            ungetc(c,stdin);
            if (scanf("%f",&valor) == 1)
                calc_operando(calc,valor);
        }
    } while (c!='q');
    calc_libera(calc);
    return 0;
}
```



```
C:\WINDOWS\system32\cmd.exe
3 5 8 * +
40.00
43.00
```



```
C:\WINDOWS\system32\cmd.exe
3 5 8 * +
40.00
43.00
3
5
8
*
40.00
+
43.00
```

calc.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "calc.h"

Calc * calc_cria(char * formato)
{
    Calc * c= (Calc *)malloc(sizeof(Calc));
    strcpy(c->f,formato);
    c->p = pilha_cria();
    return c;
}

void calc_operando(Calc * c, float valor)
{
    pilha_push(c->p,valor); // empilha oper
    // printf(c->f,valor);    // imprime topo
}

void calc_libera(Calc * c)
{
    pilha_libera(c->p);
    free(c);
}
```

```
void calc_operador(Calc * c, char op)
{
    float v1, v2, valor;
    if (pilha_vazia(c->p)) v2 = 0.0;
    else v2 = pilha_pop(c->p);
    if (pilha_vazia(c->p)) v1 = 00;
    else v1 = pilha_pop(c->p);
    switch (op)
    {
        case '+':
            valor = v1 + v2;
            break;
        case '-':
            valor = v1 - v2;
            break;
        case '*':
            valor = v1 * v2;
            break;
        case '/':
            valor = v1 / v2;
            break;
        // default:
        //     printf("unknown %c\n", op);
        //     break;
    }
    pilha_push(c->p,valor); // empilha
    printf(c->f,valor);    // imprime
}
```