Estruturas de Dados

Pilha

Universidade Estadual Vale do Acaraú – UVA

Paulo Regis Menezes Sousa

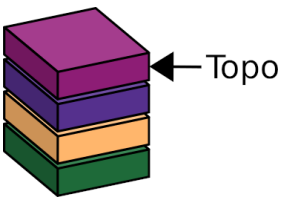
paulo_regis@uvanet.br

Pilha

- Fundamentos
- Definição do TAD Pilha
- Implementação
- Aplicação

- Pilha é uma lista em que todas as operações de inserção, remoção e acesso são feitas num mesmo extremo, denominado **topo**.
  - As **inserções** ocorrem no topo da pilha;
  - As **exclusões** ocorrem no topo da pilha.

- Devido a essa política de acesso, os itens são removidos da pilha na ordem inversa àquela em que foram inseridos, ou seja, o último a entrar é o primeiro a sair.
- Por isso, pilhas são também denominadas listas LIFO (*Last-In/ First-Out*).



Exemplo

Durante a execução de um programa, sempre que uma função é chamada, antes de passar o controle a ela, um endereço de retorno correspondente é inserido numa pilha.

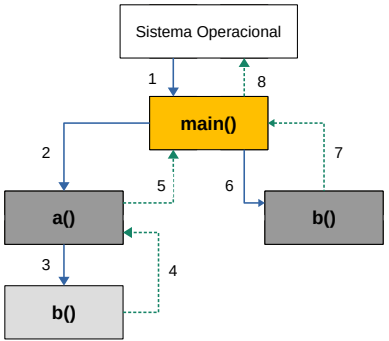
Quando a função termina sua execução, o endereço no topo da pilha é removido e a execução do programa continua a partir dele. Assim, a última função que passa o controle é a primeira a recebê-lo de volta.

```c
#include <stdio.h>

void b() { puts("B"); }

void a() { b(); puts("A"); }

int main() {
  a();
  puts("MAIN");
  b();
  return 0;
}
```



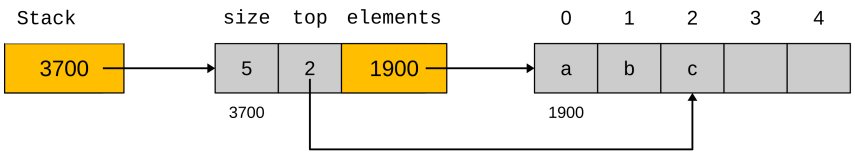
- Uma pilha P suporta as seguintes operações:
  - criação de uma pilha vazia,
  - destruição de uma pilha,
  - verificação de pilha vazia,
  - verificação de pilha cheia,
  - inserção de um elemento no topo da pilha,
  - acesso e remoção do elemento no topo da pilha,
  - acesso ao elemento no topo da pilha sem sua remoção.

- Um exemplo de definição de um TAD Pilha de inteiros

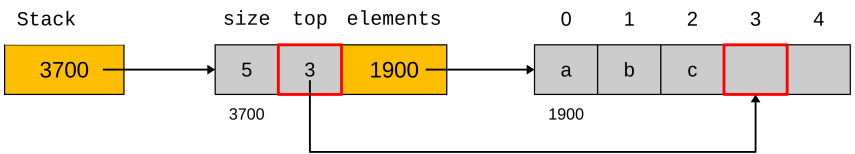
Código 1: Stack.h

```
1 #define MAX_ELEMENTS 1000
2
3 typedef struct Stack Stack;
4
5 int  Stack_isEmpty(Stack *stk);
6 int  Stack_isFull(Stack *stk);
7 void Stack_push(Stack *stk, int element);
8 int  Stack_pop(Stack *stk);
9 int  Stack_top(Stack *stk);
```

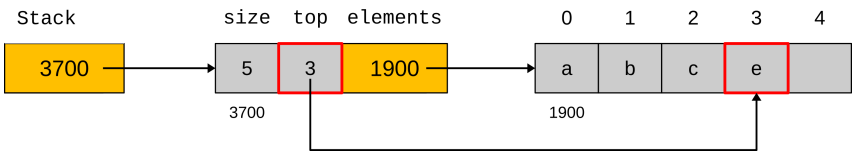
- Utilizaremos um *array* de elementos de tamanho definido no momento da criação da pilha;
- Controlaremos a posição do elemento que está no topo da pilha.



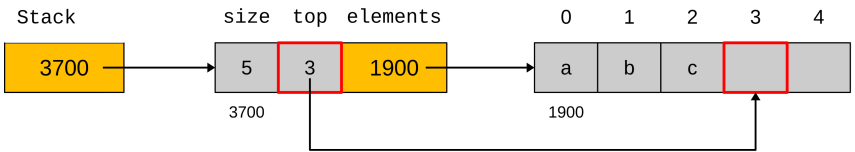
- Utilizaremos um *array* de elementos de tamanho definido no momento da criação da pilha;
- Controlaremos a posição do elemento que está no topo da pilha.



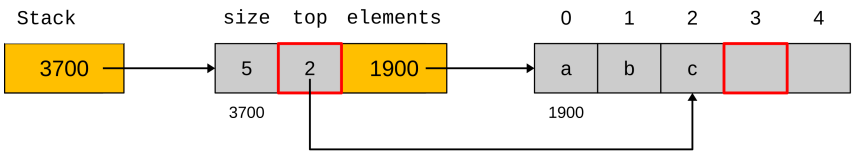
- Utilizaremos um *array* de elementos de tamanho definido no momento da criação da pilha;
- Controlaremos a posição do elemento que está no topo da pilha.



- Utilizaremos um *array* de elementos de tamanho definido no momento da criação da pilha;
- Controlaremos a posição do elemento que está no topo da pilha.



- Utilizaremos um *array* de elementos de tamanho definido no momento da criação da pilha;
- Controlaremos a posição do elemento que está no topo da pilha.



Código 2: Stack.h

```
1 #define MAX_ELEMENTS 1000
2
3 typedef struct Stack Stack;
4
5 int  Stack_isEmpty(Stack *stk);
6 int  Stack_isFull(Stack *stk);
7 void Stack_push(Stack *stk, int element);
8 int  Stack_pop(Stack *stk);
9 int  Stack_top(Stack *stk);
```

Código 3: Stack.c

```c
#include "Stack.h"

struct Stack {
    int size;
    int top;
    int elements[MAX_ELEMENTS];
};

int  Stack_isEmpty(Stack *stk){  }
int  Stack_isFull(Stack *stk){  }
void Stack_push(Stack *stk, int element){  }
int  Stack_pop(Stack *stk){  }
int  Stack_top(Stack *stk){  }
```

Código 4: Stack.h

```
1  typedef struct Stack Stack;
2
3  Stack *Stack_alloc(int size);
4  void   Stack_free(Stack *stk);
5  int    Stack_isEmpty(Stack *stk);
6  int    Stack_isFull(Stack *stk);
7  void   Stack_push(Stack *stk, int element);
8  int    Stack_pop(Stack *stk);
9  int    Stack_top(Stack *stk);
```

Código 5: Stack.c

```c
#include "Stack.h"

struct Stack {
    int size;
    int top;
    int *elements;
};

Stack *Stack_alloc(int size){  }
void   Stack_free(Stack *stk){  }
int    Stack_isEmpty(Stack *stk){  }
int    Stack_isFull(Stack *stk){  }
void   Stack_push(Stack *stk, int element){  }
int    Stack_pop(Stack *stk){  }
int    Stack_top(Stack *stk){  }
```

- Para exemplificar o uso de pilhas em programação, vamos criar um programa que converte um número natural em binário.

Para realizar essa conversão, o programa precisa efetuar sucessivas divisões por 2, a partir do número dado pelo usuário, até que o quociente 0 seja obtido. Depois, para mostrar o binário correspondente, basta que ele exiba os restos das divisões efetuadas, na ordem inversa àquela em que eles foram obtidos.

- Por exemplo, como mostra a figura abaixo, a conversão do número 13 em binário resulta em 1101.

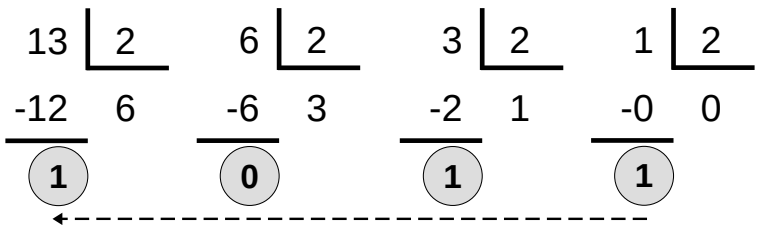


Figura 1: Conversão do número 13 em binário.

Conversão em binário

```c
#include <stdio.h>
#include <stdlib.h>
#include "Stack.h"

int main() {
    int number = 0;
    Stack *s = Stack_alloc(20);

    printf("Decimal number: ");
    scanf("%d", &number);
    do {
        Stack_push(s, number % 2);
        number /= 2;
    } while (number != 0);

    printf("Binary number: ");
    while (!Stack_isEmpty(s))
        printf("%d", Stack_pop(s));
    printf("\n");
    Stack_free(s);
    return 0;
}
```

```c
#include <stdio.h>
#include "Stack.h"

int main() {
    char str[20], i;
    Stack *s = Stack_alloc(20);

    printf("String: ");
    scanf("%[^\n]", str);
    for (i = 0; str[i]; i++)
        Stack_push(s, str[i]);

    printf("String invertida: ");
    while (!Stack_isEmpty(s))
        printf("%c", Stack_pop(s));
    printf("\n");
    Stack_free(s);
    return 0;
}
```

- Utilizaremos o tipo `Stack.c` para criar um programa que manipula expressões compostas por:
  - **operandos**, que são constantes numéricas;
  - **operadores**, que são operações aritméticas binárias (+, -, * e /);
  - **delimitadores**, que são os parênteses de abertura e de fechamento.

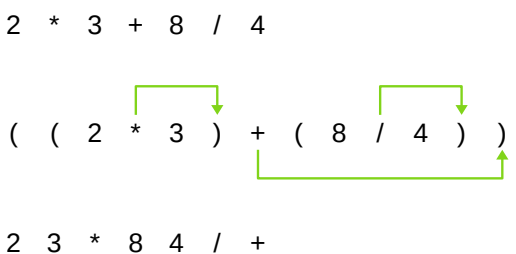
- **Forma infixa**
  - Normalmente, expressões aritméticas são escritas na forma *infixa*, isto é, com os operadores posicionados entre seus operandos.

$$2 * 3 + 8/4 \qquad \text{e} \qquad 5 * (7 - 3)$$

- A **ordem** em que as operações são efetuadas numa expressão infixa depende de suas prioridades relativas. Por convenção, * e / têm prioridade sobre + e -.
  - Operadores de mesma prioridade devem ser efetuados na ordem em que eles aparecem na expressão.
- Na forma infixa, **parênteses** servem para mudar a prioridade dos operadores.
  - Quando os parênteses numa expressão infixa não mudam a prioridade dos operadores, podemos omiti-los.

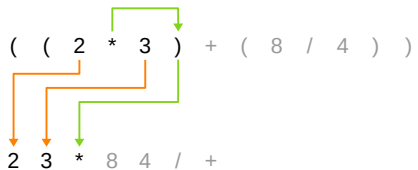
- Há dois fatores que dificultam a avaliação de uma expressão infixa:
  1. a existência de prioridades, que impede que as operações sejam efetuadas na ordem em que elas aparecem na expressão,
  2. a existência de parênteses, que altera as prioridades relativas dos operadores usados na expressão.
- Para resolver esse problema, o lógico polonês **Jan Lukasiewicz** propôs uma nova forma de escrever expressões chamada **forma posfixa** ou notação polonesa reversa.

- **Forma posfixa**
  - Para converter a forma infixa em posfixa basta (**1**) parentesiar completamente a expressão infixa, respeitando as prioridades dos operadores, (**2**) reescrever a expressão descartando os parênteses e (**3**) mover os operadores para a posição ocupada por seus parênteses de fechamento.



- À medida que a expressão *infixa* é percorrida, os operandos encontrados são imediatamente copiados para a expressão *posfixa*.

- Os operadores, porém, devem aguardar até que seus respectivos parênteses de fechamento sejam encontrados.



- Usando uma **pilha** como local de espera para os operadores, podemos obter o efeito desejado.

Elemento	Ação	Pilha	Posfixa
(	descartar (	[ ]	“ ”
(	descartar (	[ ]	“ ”
2	anexar 2 à posfixa	[ ]	“2”
*	empilhar *	[*]	“2”
3	anexar 3 à posfixa	[*]	“23”
)	desempilhar * e anexar à posfixa	[ ]	“23*”
+	empilhar +	[+]	“23*”
(	descartar (	[+]	“23*”
8	anexar 8 à posfixa	[+]	“23*8”
/	empilhar /	[+, /]	“23*8”
4	anexar 4 à posfixa	[+, /]	“23*84”
)	desempilhar / e anexar à posfixa	[+]	“23*84/+”
)	desempilhar + e anexar à posfixa	[ ]	“23*84/+”

- **Avaliação de uma expressão posfixa usando pilha** (esquerda para a direita)
  - quando um **operando** é encontrado, seu valor é empilhado;
  - quando um **operador** é encontrado, dois valores são desempilhados e o resultado da operação feita com eles é empilhado.
  - No final, o **valor da expressão** estará no topo da pilha.
- **Exempo**: expressão "23*84/+"

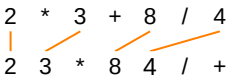
Elemento	Ação	Pilha
2	empilhar 2	[2]
3	empilhar 3	[2, 3]
*	desempilhar 3 e 2 e empilhar 3*2	[6]
8	empilhar 8	[6, 8]
4	empilhar 4	[6, 8, 4]
/	desempilhar 8 e 4 e empilhar 8/4	[6, 2]
+	desempilhar 2 e 6 e empilhar 2+6	[8]

Aplicação

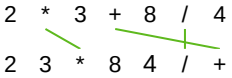
```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "Stack.h"
#define SIZE 256

char *posfix(char *in) {
    Stack *s = Stack_alloc(SIZE);
    char *out = malloc(SIZE);
    int i, j;
    for (i = 0; in[i]; i++)
        if (isdigit(in[i]))
            out[j++] = in[i];
        else if (strchr("+*-/", in[i]))
            Stack_push(s, in[i]);
        else if (in[i] == ')')
            out[j++] = Stack_pop(s);
    out[j] = '\0';
    Stack_free(s);
    return out;
}
```

- **Conversão em posfixa de uma expressão infixa não completamente parentesiada**
  - A ordem dos operandos na expressão infixa não muda na expressão posfixa;



  - a ordem dos operadores muda (pois eles devem ser colocados após seus operandos).



Em uma expressão posfixa, as operações são efetuadas na ordem em que elas aparecem e, portanto, a posição de um operador na forma posfixa é definida pela sua **prioridade** na forma infixa.

- A prioridade dos operadores é dada pela função abaixo:

```
1  int priority(char c) {
2      switch (c) {
3          case '(': return 0;
4          case '+':
5          case '-': return 1;
6          case '*':
7          case '/': return 2;
8      }
9      return -1;
10 }
```

- A conversão de uma expressão infixa e, não totalmente parentesiada, em uma expressão posfixa s é feita do seguinte modo:
  - Inicie com uma pilha de caracteres P e uma expressão posfixa s vazias.
  - Para cada elemento da expressão infixa e, da esquerda para a direita, faça:
    - Se for um **parêntese de abertura**, empilhe-o em P.
    - Se for um **operando**, anexe-o à expressão posfixa s.
    - Se for um **operador**, enquanto houver no topo da pilha P outro operador com maior ou igual prioridade, desempilhe esse operador e anexe-o a s; depois, empilhe em P o operador recém-encontrado na expressão e.
    - Se for um **parêntese de fechamento**, remova um operador da pilha P e anexe-o a s, até que um parêntese de abertura apareça no topo na pilha. No final, desempilhe esse parêntese e descarte-o.
  - Depois de percorrer completamete a expressão infixa e, esvazie a pilha, anexando à expressão posfixa s cada um dos operadores desempilhados.

```c
char *posfix(char *in) {
    Stack *s = Stack_alloc(SIZE);
    char *out = malloc(SIZE);
    int i, j;
    for (i = 0; in[i]; i++) {
        if (in[i] == '(') Stack_push(s, in[i]);
        else if (isdigit(in[i])) out[j++] = in[i];
        else if (strchr("+*-/", in[i])) {
            while (!Stack_isEmpty(s)
                    && priority(in[i]) <= priority(Stack_top(s)) )
                out[j++] = Stack_pop(s);
            Stack_push(s, in[i]);
        }
        else if (in[i] == ')') {
            while (Stack_top(s) != '(')
                out[j++] = Stack_pop(s);
            Stack_pop(s);
        }
    }
    while (!Stack_isEmpty(s))
        out[j++] = Stack_pop(s);
    out[j] = '\0';
    Stack_free(s);
    return out;
}
```

- **Avaliação da forma posfixa**
  - Inicie com uma pilha de inteiros P vazia.
  - Para cada elemento da expressão e, da esquerda para a direita, faça:
    - Se for um **operando**, empilhe em P o seu valor numérico.
    - Se for um **operador**, desempilhe de P dois valores, aplique o operador a esses valores e empilhe em P o resultado obtido.
    - No final, devolva como resultado o valor existente do topo de P.

```c
int evaluate(char *expression) {
    Stack *stack = Stack_alloc(strlen(expression));
    int i, x, y, z;
    for (i = 0; expression[i]; i++)
        if (isdigit(expression[i]))
            Stack_push(stack, expression[i] - '0');
        else {
            y = Stack_pop(stack);
            x = Stack_pop(stack);
            switch (expression[i]) {
                case '+': Stack_push(stack, x+y); break;
                case '-': Stack_push(stack, x-y); break;
                case '*': Stack_push(stack, x*y); break;
                case '/': Stack_push(stack, x/y); break;
            }
        }
    z = Stack_pop(stack);
    Stack_free(stack);
    return z;
}
```

- Numa expressão infixa, operandos são separados por operadores. Então, mesmo quando eles têm vários dígitos, é fácil identificá-los,

$$''2 * 34 + 1''$$

- Para eliminar essa ambiguidade, basta incluir um espaço na forma posfixa, sempre que um operador for encontradona forma infixa,

$$''234 * 1+'' \quad \rightarrow \quad ''2 \quad 34 \quad * 1+''$$

```c
char *posfix(char *in) {
    Stack *s = Stack_alloc(SIZE);
    char*out = (char *) malloc(SIZE * sizeof(char));
    int    i = 0, j = 0;
    for (i = 0; in[i]; i++) {
        if ( in[i] == '(') Stack_push(s, in[i]);
        else if ( isdigit(in[i])) out[j++] = in[i];
        else if ( strchr("+*-/", in[i]) ) {
            out[j++] = ' ';
            while ( !Stack_isEmpty(s)
                    && priority(in[i]) <= priority(Stack_top(s)) )
                out[j++] = Stack_pop(s);
            Stack_push(s, in[i]);
        }
        else if ( in[i] == ')' ) {
            while ( Stack_top(s) != '(' )
                out[j++] = Stack_pop(s);
            Stack_pop(s);
        }
    }
    while ( !Stack_isEmpty(s) )
        out[j++] = Stack_pop(s);
    out[j] = '\0';
    Stack_free(s);
    return out;
}
```

```c
int evaluate(char *expression) {
    Stack *stack = Stack_alloc(SIZE);
    int i, j, x, y, z;
    char *subExp = (char*) malloc(strlen(expression));
    for (i = 0; expression[i]; i++) {
        if ( isdigit(expression[i]) ) {
            for (j = 0; isdigit(expression[i]); j++, i++)
                subExp[j] = expression[i];
            subExp[j] = '\0';
            Stack_push(stack, atoi(subExp));
        }
        if ( strchr("+-*/", expression[i]) ) {
            y = Stack_pop(stack);
            x = Stack_pop(stack);
            switch (expression[i]) {
                case '+': Stack_push(stack, x+y); break;
                case '-': Stack_push(stack, x-y); break;
                case '*': Stack_push(stack, x*y); break;
                case '/': Stack_push(stack, x/y); break;
            }
        }
    }
    z = Stack_pop(stack);
    Stack_free(stack); free(subExp);
    return z;
}
```

- Que modificações precisamos realizar na implementação de uma pilha de inteiros para que possamos armazenar qualquer tipo de dados?

Código 6: Stack.h Genérico

```
1  typedef struct Stack Stack;
2
3  Stack *Stack_alloc(int size);
4  void   Stack_free(Stack *stack);
5  int    Stack_isEmpty(Stack *stack);
6  int    Stack_isFull(Stack *stack);
7
8  void   Stack_push(Stack *stack, void *element);
9  void  *Stack_pop(Stack *stack);
10 void  *Stack_top(Stack *stack);
```

Funções afetadas