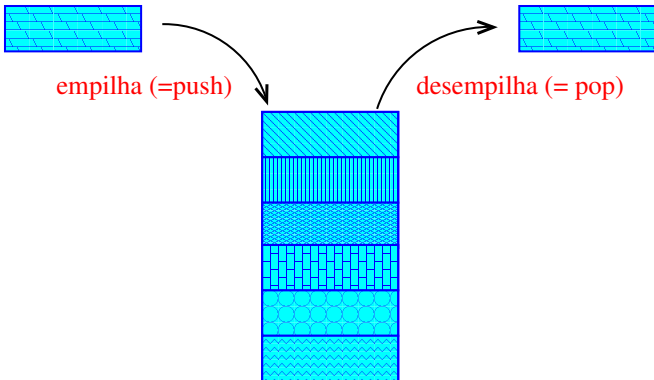


Melhores momentos

AULA 10

Pilhas

Uma **pilha** (= *stack*) é uma lista (=sequência) dinâmica em que todas as operações (**inserções**, **remoções** e **consultas**) são feitas em uma mesma extremidade chamada de **topo**.



Interfaces

Uma **interface** (= *interface*) é uma fronteira entre entre a **implementação** de um biblioteca e o **programa que usa** a biblioteca.

Um **cliente** (= *client*) é um programa que chama alguma função da biblioteca.

Implementação

```
double sqrt(double x){  
    [...]  
    return raiz;  
}  
[...]
```

libm

Interface

```
double sqrt(double);  
double sin(double);  
double cos(double);  
double pow(double,double);  
[...]
```

math.h

Cliente

```
#include <math.h>  
  
[...]  
c = sqrt(a*a+b*b);  
[...]
```

prog.c

Interface stack.h

```
/*  
 * stack.h  
 * INTERFACE: funcoes para manipular uma pilha  
 */  
  
#include "item.h"  
  
void stackInit(int);  
int stackEmpty();  
void stackPush(Item);  
Item stackPop();  
Item stackTop();  
void stackFree();  
void stackDump();
```

Desse jeito, o cliente só pode usar **uma pilha**.

AULA 11

PilhaS implementadaS em lista encadeada



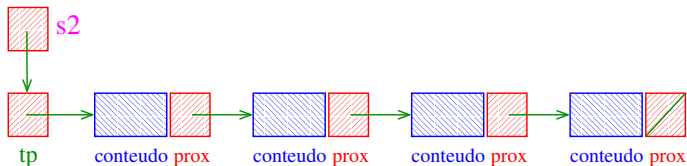
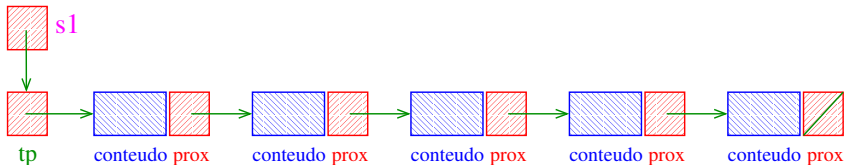
Fonte: <http://www.dumpanalysis.org/>

PF 6.3, S 4.4

<http://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html>

PilhaS implementadaS em listaS encadeadaS

As pilhas serão armazenadas em **listaS encadeadaS** sem **cabeça**.



Pilha implementada em lista encadeada

Para cada pilha há um ponteiro `tp` para a lista.

`tp->conteudo` é o elemento do topo da pilha.

Uma pilha `s` está vazia se “`s->tp == NULL`”.

Uma pilha está cheia se ... acabou a memória disponível.

Interface stackS.h

```
/*  
 * stackS.h  
 * INTERFACE: funcoes para manipular pilhas  
 */  
  
#include "item.h"  
  
typedef void *Stack;  
  
Stack stackInit(int);  
int stackEmpty(Stack);  
void stackPush(Stack, Item);  
Item stackPop(Stack);  
Item stackTop(Stack);  
void stackFree(Stack);  
void stackDump(Stack);
```

Infixa para posfixa novamente...

Recebe uma expressão infixa `inf` e
devolve a correspondente expressão `posfixa`.

```
char *infixaParaPosfixa(char *inf) {  
    char *posf; /* expressao polonesa */  
    int  n = strlen(inf);  
    int  i; /* percorre infixa */  
    int  j; /* percorre posfixa */  
    char x; /* item do topo da pilha */  
    Stack s; /* PILHA */  
  
    /*aloca area para expressao polonesa*/  
    posf = mallocSafe((n+1)*sizeof(char));  
    /* 0 '+1' eh para o '\0' */
```

cases '(' e ')'

```
s = stackInit(n) /* inicializa a pilha */  
  
/* examina cada item da infixa */  
for (i = j = 0; i < n; i++) {  
    switch (inf[i]) {  
        case '(':  
            stackPush(s, inf[i]);  
            break;  
  
        case ')':  
            while((x = stackPop(s)) != '(')  
                posf[j++] = x;  
            break;
```

```
cases '+', '-', '*' e '/'
```

```
case '+':
```

```
case '-':
```

```
    while (!stackEmpty(s)
           && stackTop(s) != '(')
        posf[j++] = stackPop(s);
    stackPush(s, inf[i]);
    break;
```

```
case '*':
```

```
case '/':
```

```
    while (!stackEmpty(s)
           && (x = stackTop(s)) != '('
           && x != '+' && x != '-')
        posf[j++] = stackPop(s);
    stackPush(s, inf[i]);
    break;
```

default e finalizações

```
default:
    if(inf[i] != ' ')
        posf[j++] = inf[i];
    } /* fim switch */
} /* fim for (i = j = 0...) */

/* desempilha todos os operadores que restaram */
while (!stackEmpty(s))
    posf[j++] = stackPop(s)
posf[j] = '\0'; /* fim expr polonesa */
stackFree(s);
return posf;
} /* fim funcao */
```

Implementação stackS.c

```
#include "stackS.h"
/* PILHA: implementacao em lista encadeada
 */
typedef struct stackNode* Link;
struct stackNode{
    Item conteudo;
    Link prox;
};
struct stack {
    Link tp;
};
typedef struct stack *TStack;
```

Implementação stackS.c

```
Stack stackInit(int n) {  
    TStack s = mallocSafe(sizeof *s);  
    s->tp = NULL;  
    return (Stack)s;  
}
```

Implementação stackS.c

```
Stack stackInit(int n) {  
    TStack s = mallocSafe(sizeof *s);  
    s->tp = NULL;  
    return (Stack)s;  
}
```

```
int stackEmpty(Stack s) {  
    return ((TStack)s)->tp == NULL;  
}
```

Implementação stackS.c

```
void stackPush(Stack s, Item item) {  
    Link nova = mallocSafe(sizeof *nova);  
  
    nova->conteudo = item;  
    nova->prox = ((TStack)s)->tp;  
    ((TStack)s)->tp = nova;  
}
```

Implementação stackS.c

```
Item stackPop(Stack s) {  
    Link p = ((TStack)s)->tp;  
    Item conteudo = p->conteudo;  
    ((TStack)s)->tp = p->prox;  
    free(p);  
    return conteudo;  
}
```

Implementação stackS.c

```
Item stackPop(Stack s) {  
    Link p = ((TStack)s)->tp;  
    Item conteudo = p->conteudo;  
    ((TStack)s)->tp = p->prox;  
    free(p);  
    return conteudo;  
}  
  
Item stackTop(Stack s) {  
    return ((TStack)s)->tp->conteudo;  
}
```

Implementação stackS.c

```
void stackFree(Stack s) {  
    while (((TStack)s)->tp != NULL) {  
        Link p = ((TStack)s)->tp;  
        ((TStack)s)->tp = p->prox;  
        free(p);  
    }  
    free(s);  
}
```

Implementação stackS.c

```
void stackDump() {  
    Link p = ((TStack)s)->tp;  
  
    fprintf(stdout, "pilha : ");  
    if (p == NULL) fprintf(stdout, "vazia.");  
    while (p != NULL) {  
        fprintf(stdout, "%c ", p->conteudo);  
        p = p->prox;  
    }  
    fprintf(stdout, "\n");  
}
```

Compilação

cria o obj **stackS.o**

```
> gcc -Wall -O2 -ansi -pedantic -Wno-unused-result -c stackS.c
```

cria o obj **polonesa.o**

```
> gcc -Wall -O2 -ansi -pedantic -Wno-unused-result -c polonesa.c
```

cria o executável **polonesa**

```
> gcc -o polonesa stackS.o polonesa.o
```

Makefile

```
polonesa: polonesa.o stackS.o
    gcc polonesa.o stackS.o -o polonesa

polonesa.o: polonesa.c item.h
    gcc -Wall -O2 -ansi -pedantic \
        -Wno-unused-result -c polonesa.c

stackS.o: stackS.c item.h
    gcc -Wall -O2 -ansi -pedantic \
        -Wno-unused-result -c stackS.c
```

Cálculo de expressões



Fonte:

<https://escolakids.uol.com.br/matematica/resolvendo-expressoes-numericas-ii.htm>

PF 6.3, S 4.4

<http://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html>

Cliente de stackS

Considere a seguinte função:

```
float Valor(char c);
```

Dada uma expressão infixa, com variáveis representadas por letras, e as operações binárias +, -, *, e /, eventualmente com parêntesis, usando a função acima, calcule o **valor da expressão**.

Exemplo: $A * (B / (C + D - E + F) * G)$ vale 60 para $A = 4$, $B = 9$, $C = 7$, $D = 3$, $E = 9$, $F = 2$ e $G = 5$.

Vamos usar a interface stackS.h

```
/*  
 * stackS.h  
 * INTERFACE: funcoes para manipular uma pilha  
 */  
  
#include "item.h"  
  
typedef void *Stack;  
  
Stack stackInit(int);  
int stackEmpty(Stack);  
void stackPush(Stack, Item);  
Item stackPop(Stack);  
Item stackTop(Stack);  
void stackFree(Stack);
```

Infixa para posfixa novamente...

Substituiremos a **posfixa** por uma **pilha de valores**, e calcularemos a expressão paralelamente à posfixa.

```
char *infixaParaPosfixa(char *inf) {  
    char *posf; /* expressao polonesa */  
    int  n = strlen(inf);  
    int  i; /* percorre infixa */  
    int  j; /* percorre posfixa */  
    char x; /* item do topo da pilha */  
    Stack s; /* PILHA */  
  
    /*aloca area para expressao polonesa*/  
    posf = mallocSafe((n+1)*sizeof(char));  
    /* 0 '+1' eh para o '\0' */
```

Duas pilhas

Usaremos a pilha de operadores (`char`),
e uma pilha de valores (`float`).

Se as pilhas usadas fossem guardar elementos
do mesmo tipo, o `Item` poderia ser esse tipo.

Aqui, definiremos o item da seguinte maneira.

```
/* item.h */  
  
typedef void *Item;
```

Uma pilha guardará ponteiros para `char`
e a outra, ponteiros para `float`.

Cálculo de expressão

Substituiremos a **posfixa** por uma **pilha de valores**.

```
float calculaExpressao(char *inf) {  
    int n = strlen(inf);  
    int i; /* percorre infixa */  
    char *x;  
    float *v, res;  
    Stack sOpe; /* pilha de operadores */  
    Stack sVal; /* pilha de valores */  
  
    /* inicializa as pilhas */  
    sOpe = stackInit(n);  
    sVal = stackInit(n);
```

case '('

Igual ao original, mas empilha o endereço.

```
/* examina cada item da infixa */  
for (i = j = 0; i < n; i++) {  
    switch (inf[i]) {  
        case '(':  
            stackPush(sOpe, &inf[i]);  
            break;
```

Demais casos:

Sempre que iria colocar um operador na posfixa, desempilha dois valores da pilha de valores, faz o cálculo da operação e empilha o resultado na pilha de valores.

case ') ':

```
case ' ) ':  
    while(*(x = stackPop(sOpe)) != '(')  
        /* pop, pop, faz o cálculo e push */  
        fazOperacao(sVal, *x);  
    break;
```

Função **fazOperacao**: recebe a **pilha de valores** e um caracter que indica **uma operação**; desempilha os operandos, faz o cálculo e empilha o resultado do cálculo na **pilha de valores**.

cases '+', e '-'

```
case '+': case '-':  
    while (!stackEmpty(sOpe)  
        && *stackTop(sOpe) != '(') {  
        x = stackPop(sOpe);  
        fazOperacao(sVal, *x);  
    }  
    stackPush(sOpe, &inf[i]);  
    break;
```

cases '*' e '/'

```
case '*': case '/':  
    while (!stackEmpty(sOpe)  
        && *(x = stackTop(sOpe)) != '('  
        && *x != '+' && *x != '-')  
        fazOperacao(sVal, *x);  
    stackPush(sOpe, &inf[i]);  
    break;
```

default

No default, é uma letra, então aciona a função **Valor**.

default:

```
    if (inf[i] != ' ') {  
        v = mallocSafe(sizeof(float));  
        *v = Valor(inf[i]);  
        stackPush(sVal, v);  
    }  
} /* fim switch */  
} /* fim for (i = j = 0...) */
```

Finalizações

```
/* desempilha todos os operadores que restaram */  
while (!stackEmpty(sOpe)) {  
    x = stackPop(sOpe);  
    fazOperacao(sVal, *x);  
}  
v = stackPop(sVal);    /* resultado final */  
stackFree(sOpe);  
stackFree(sVal);  
res = *v;  
free(v);  
return res;  
} /* fim funcao */
```

Função auxiliar fazOperacao

```
void fazOperacao(Stack sVal, char op) {  
    float *v, *v1, *v2;  
    v2 = stackPop(sVal);  v1 = stackPop(sVal);  
    v = mallocSafe(sizeof(float));  
    switch(op) {  
        case '+':  *v = *v1 + *v2;  break;  
        case '-':  *v = *v1 - *v2;  break;  
        case '*':  *v = *v1 * *v2;  break;  
        case '/':  *v = *v1 / *v2;  break;  
    }  
    stackPush(sVal, v);  
    free(v1);  free(v2);  
}
```

Note que o primeiro desempilhado é o segundo operando!

EP3

Você vai usar isso no EP3!

EP3

Você vai usar isso no EP3!

Mais operadores: o resto da divisão (%),
o menos unário (nega) e a atribuição.

EP3

Você vai usar isso no EP3!

Mais operadores: o resto da divisão (%),
o menos unário (nega) e a atribuição.

Próxima aula: **filas**

Agora vamos ver a P1.