MAC00323 Algoritmos e Estruturas de Dados II

Edição 2017

Administração

Página da disciplina:

http://paca.ime.usp.br/

- aulas
- exercícios-programa
- fórum: perguntem, respondam, ...
- material: brinquem com os programas
- ...

Administração

Página da disciplina:

http://paca.ime.usp.br/

- aulas
- exercícios-programa
- fórum: perguntem, respondam, ...
- material: brinquem com os programas
- ...

Exercício programa 1 e 2: disponíveis na página

Livro

Nossa referência básica é o livro

SW = *Sedgewick & Wayne*,
Algorithms, 4th Editions

Consulte as notas de aula baseadas no livro *Algorithms* de Paulo Feofiloff

http://www.ime.usp.br/~pf/estruturas-de-dados.

Onde você se meteu. . .

MAC0323 continua a tarefa de MAC0121, é uma disciplina introdutória em:

- projeto, correção e eficiência de algoritmos e
- estruturas de dados = esquema de organizar dados que os deixa acessível para serem processados eficientemente.

Estudaremos, através de exemplos, a correção, a análise de eficiência e projeto de algoritmos muito bacanas e que são amplamento utilizados por programadores.

MAC0323

MAC0323 combina técnicas de

- programação
- correção de algoritmos (relações invariantes)
- análise da eficiência de algoritmos e
- estruturas de dados elementares

que nasceram de aplicações cotidianas em ciência da computação.

Pré-requisitos

Os pré-requisito oficial de MAC0323 são

- MAC0121 Algoritmos e Estruturas de Dados I e
- MAC0216 Técnicas de Programação I

Principais tópicos

Alguns dos tópicos de MAC0323 são:

- Bags, Queues e Stacks;
- Union-find;
- Tabelas de símbolos: Árvore binária de busca; Árvores balanceadas de busca; Tabelas de Hash;
- Grafos: orientados, não orientados;
- Problemas em grafos: Árvore geradora mínima; Caminhos mínimos;
- Strings : Tries; Autômatos e expressões regulares.

Tudo isso regado a muita análise de eficiência de algoritmos e invariantes.

Localização

MAC0323 é a segundo passo na direção de

- Algoritmos
- Estruturas de Dados

Várias outras disciplina se apoiam em MAC0323.

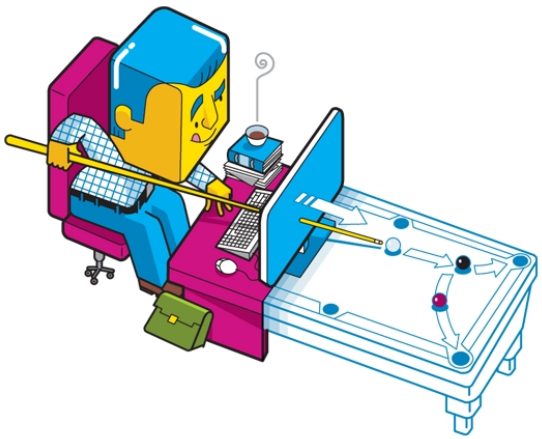
Java

Usaremos a linguagem Java.

Bem, nada profundo.

O foco é algoritmos e estruturas de dados e a ideia é a linguagem não nos distrair muito, mas isso é inevitável . . . e divertido :-)

Interfaces



Fonte: http://allfacebook.com/

Before I built a wall I'd ask to know

What I was walling in or walling out,

And to whom I was like to give offence.

Something there is that doesn't love a wall,

That wants it down.

Robert Frost, *Mending Wall*

The Practice of Programming

B.W.Kernigham e R. Pike

S 3.1, 4.2, 4.3, 4.4

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

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

```
public class Stack<T>{
  [...]
public T pop() {
}
      [...]
```

Stack.class

Interface

```
public T pop()
public push(T item)
public int size()
public boolean isEmpty()
      [...]
```

API

Cliente

```
public class Prog {
      [...]
 s = new Stack();
      [...]
```

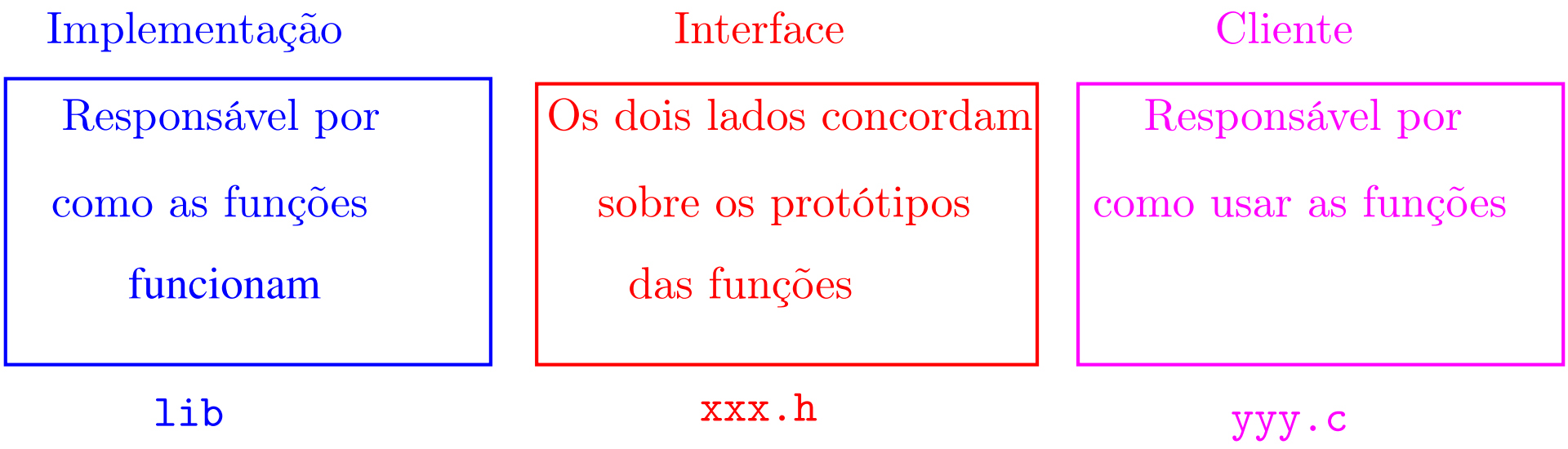
Prog.java

Interfaces

Para cada função na biblioteca o cliente precisa saber

- o seu nome, os seus argumentos e os tipos desses argumentos;
- o tipo do resultado que é retornado.

Só a quem implementa interessa os detalhes de implementação.

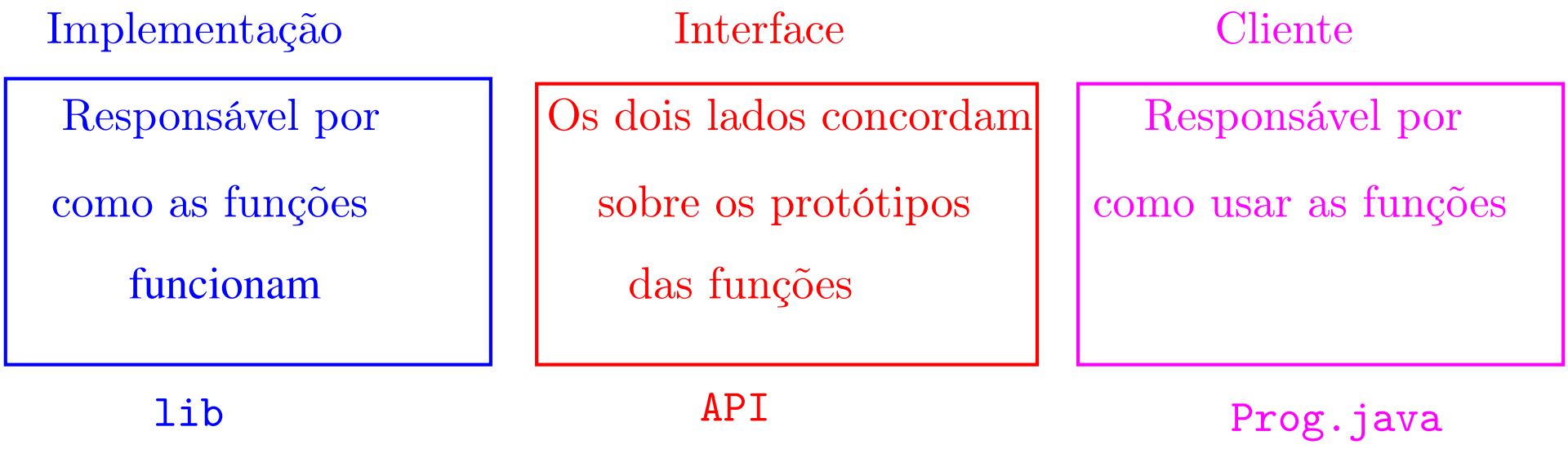


Interfaces

Para cada função na biblioteca o cliente precisa saber

- o seu nome, os seus argumentos e os tipos desses argumentos;
- o tipo do resultado que é retornado.

Só a quem implementa interessa os detalhes de implementação.



Interfaces

Entre as decisões de projeto estão

Interface: quais serviços serão oferecidos? A interface é um "contrato" entre o usuário e o projetista.

Ocultação: qual informação é visível e qual é privada? Uma interface deve prover acesso aos componente enquanto esconde detalhes de implementação que podem ser alterados sem afetar o usuário.

Recursos: quem é responsável pelo gerenciamento de memória e outros recursos?

Erros: quem detecta e reporta erros e como?

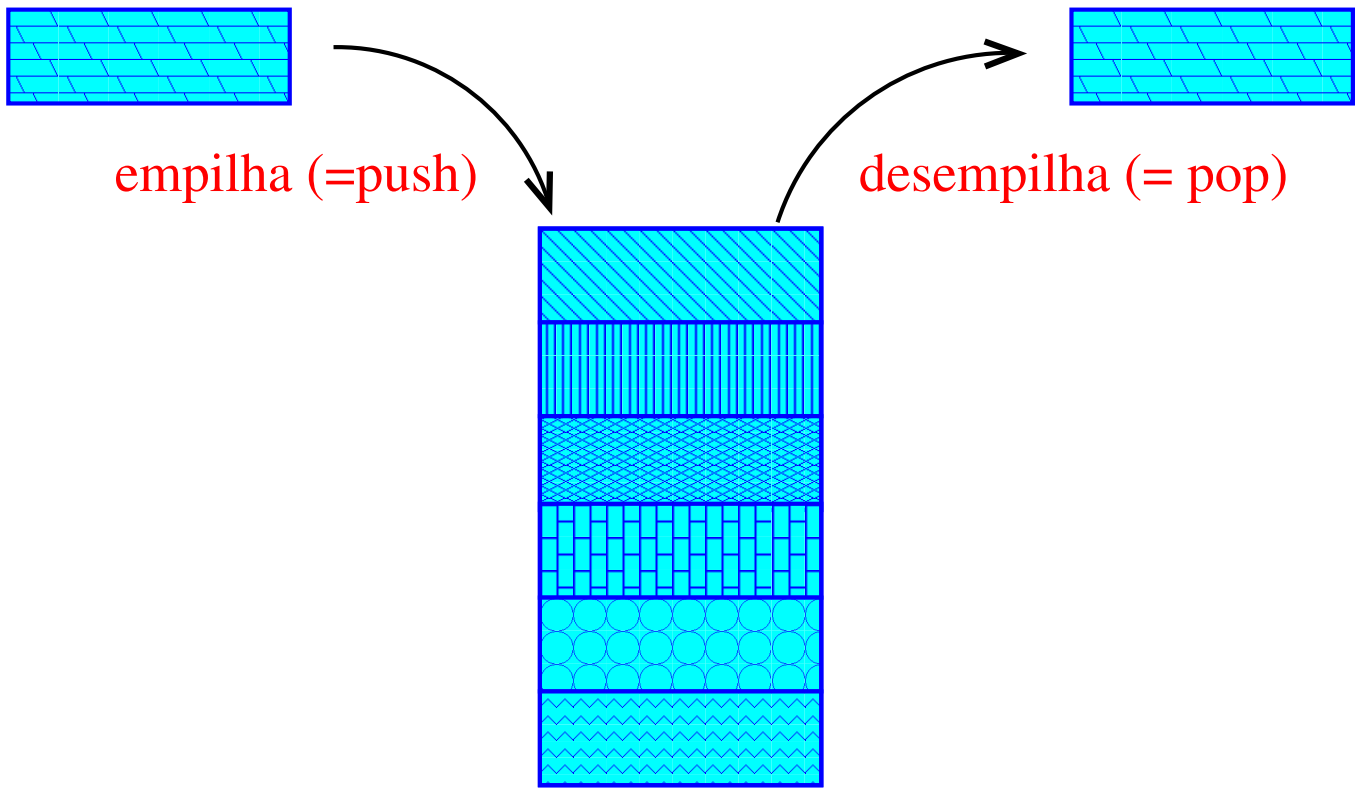
Pilhas

Fonte: http://dontmesswithtaxes.typepad.com/

PF 6.1 e 6.3
http://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html

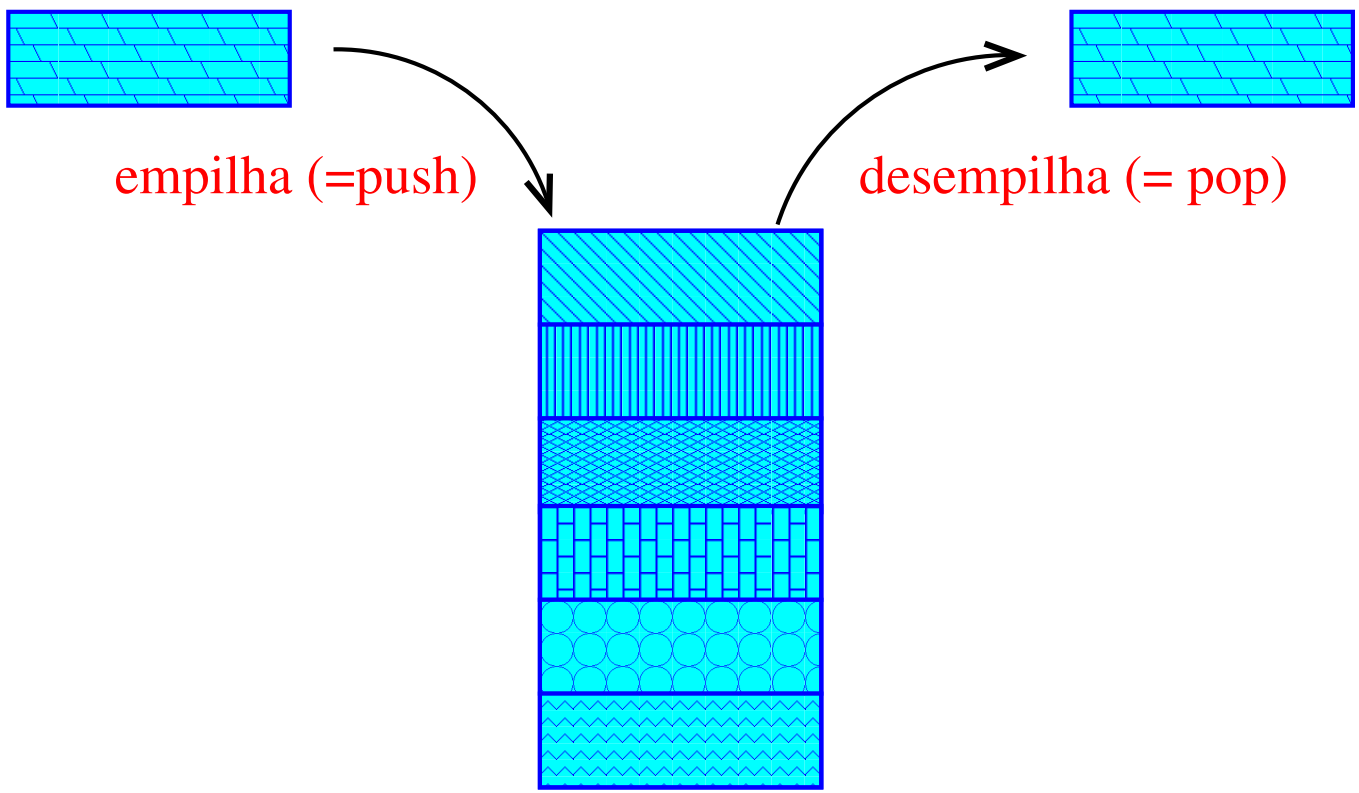
Pilhas

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**.



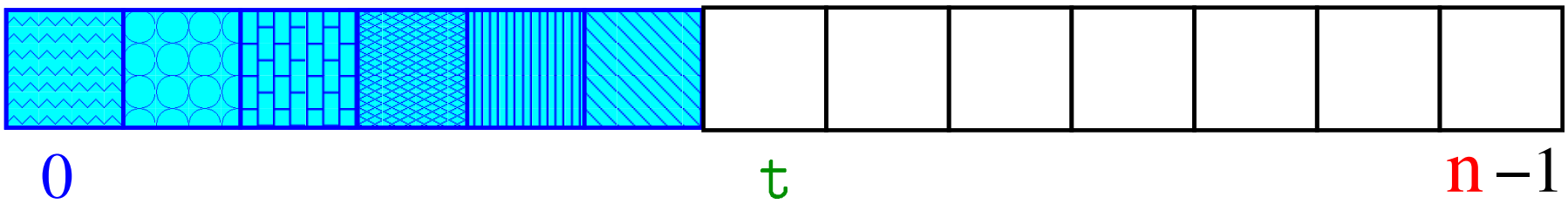
Pilhas

Assim, o primeiro objeto a ser removido de uma pilha é o último que foi inserido. Esta política de manipulação é conhecida pela sigla **LIFO** (=*Last In First Out*)



Implementação em um vetor

A pilha será armazenada em um vetor `a[0 . . n-1]`.



O índice `t` indica o topo (=*top*) da pilha.

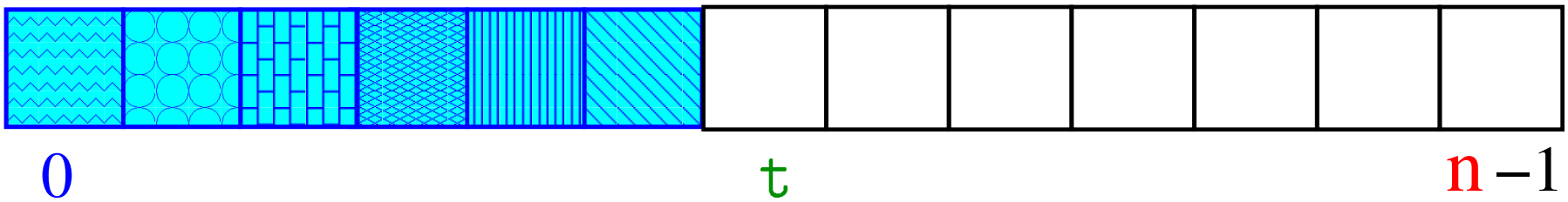
Esta é a primeira posição vaga da pilha.

A pilha está vazia se "`t == 0`".

A pilha está cheia se "`t == n`".

Implementação em um vetor

A pilha será armazenada em um vetor `a[0 . . n-1]`.



Para remover (=desempilhar=*pop*) um elemento faça

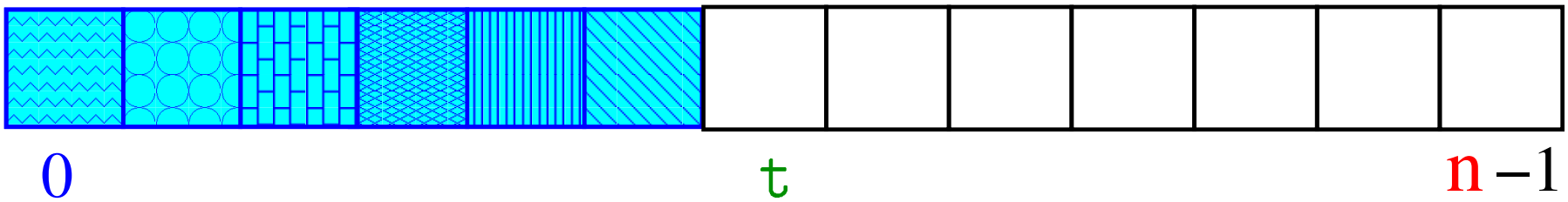
```
x = a[--t];
```

que é equivalente a

```
t -= 1;
x = a[t];
```

Implementação em um vetor

A pilha será armazenada em um vetor `a[0 . . n-1]`.



Para inserir (=empilhar=*push*) um elemento faça

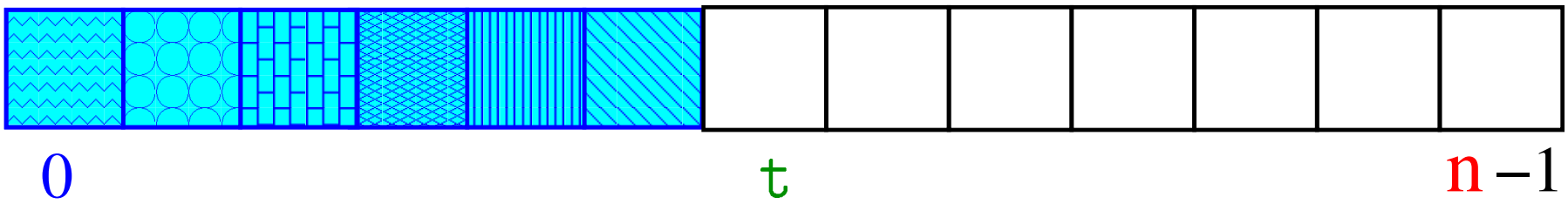
```
a[t++] = x;
```

que é equivalente a

```
a[t] = x;
t += 1;
```

Implementação em um vetor

A pilha será armazenada em um vetor `a[0 . . n-1]`.

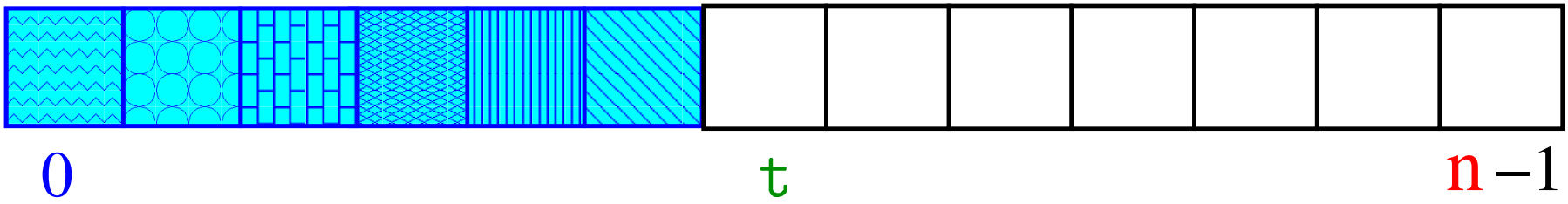


Para consultar um elemento, sem removê-lo, faça

```
x = a[t-1];
```

Implementação em um vetor

A pilha será armazenada em um vetor `a[0 . . n-1]`.



Tentar desempilhar de uma pilha que está vazia é um erro chamado *stack underflow*

Tentar empilhar em uma pilha cheia é um erro chamado *stack overflow*

PilhaS implementadaS em lista encadeada

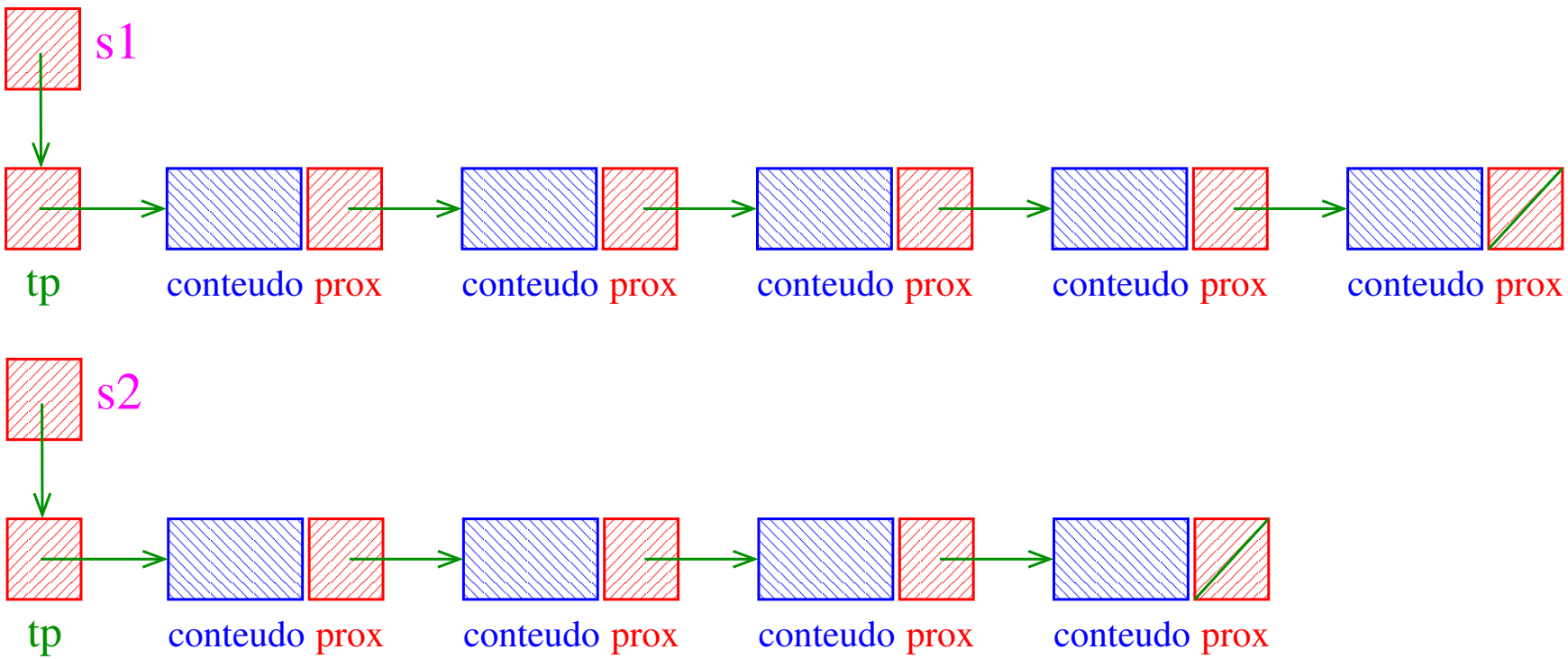
Fonte: http://www.dumpanalysis.org/

PF 6.3, S 4.4

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

PilhaS implementadaS em listaS encadeadaS

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



PilhaS implementadaS em listaS encadeadaS

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 stack.h

```
/*
* stack.h
* INTERFACE: funcoes para manipular uma
* pilha
*/
typedef struct stack *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 */
   Stack s; /* PILHA */

   /*aloca area para expressao polonesa*/
   posf = mallocSafe((n+1)*sizeof(char));
   /* O '+1' eh para o '\0' */
```

case '('

```
s = stackInit(n) /* inicializa a pilha */

/* examina cada item da infixa */
for (i = j = 0; i < n; i++) {
   switch (inf[i]) {
        char x; /* item do topo da pilha */
        case '(':
            stackPush(s,inf[i]);
            break;
```

case ')'

```
case ')':
    while((x = stackPop(s)) != '(')
        posf[j++] = x;
    break;
```

case '+', case '-'

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

case '*', case '/'

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

default

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

Finalizações

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

Implementação stack.c

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

Implementação stack.c

```
Stack
stackInit(int n)
{
   Stack s = mallocSafe(sizeof *s);

   s->tp = NULL;
   return s;
}
```

Implementação stack.c

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

Implementação stack.c

```
void
stackPush(Stack s, Item item)
{
   Link nova = mallocSafe(sizeof *nova);

   nova->conteudo = item;
   nova->prox = s->tp;
   s->tp = nova;
}
```

Implementação stack.c

```
Item
stackPop(Stack s)
{
   Link p = s->tp;
   Item conteudo = p->conteudo;

   s->tp = p->prox;
   free(p);
   return conteudo;
}
```

Implementação stack.c

```
Item
stackTop(Stack s)
{
   return s->tp->conteudo;
}
```

Implementação stack.c

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

Implementação stack.c

```
void
stackDump() {
   int p = 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");
}
```