MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

Sobre MAC0323

Fonte: http://jainanimation.in/blog/...

Blue Pill or *Red Pill*
The Matrix
https://www.youtube.com

Sobre MAC0323

- página da disciplina
- responsáveis
- livro
- pré-requisitos
- aulas
- exercícios-programa
- listas de exercícios

Páginas da disciplina

http://www.ime.usp.br/~cris/mac0323

Critério de avaliação, programação das aulas, EPs, listas de exercícios, etc.

E-disciplinas: aulas gravadas, avisos, notas, fórum de discussão, entregas das tarefas, etc.

Responsáveis

Siolin

Cris

Pré-requisito

MAC0121 Algoritmos e Estruturas de Dados I

Livro

Nossa referência básica é o livro SW

Sedgewick & Wayne,
Algorithms, 4th Editions
http://algs4.cs.princeton.edu/

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

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

Exercícios-programa C

https://twitter.com/slidenerdtech

Avaliação

EPs

Listas de exercícios

EPs: 70% da nota

ML: 30% da nota

A média das notas dos EPs é calculada assim:

MaEP: média aritmética das notas nos EPs

Se a nota em um dos EPs for 3.0 ou menos,

então MEP $= \min\{3.0, \text{MaEP}\}$

senão MEP $=$ MaEP

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 acessíveis para serem processados eficientemente.

Estudaremos, através de exemplos, a correção,
a análise de eficiência e projeto de algoritmos
muito bacanas e que são amplamente 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-requisitos 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.

Com um pouco de análise e eficiência de algoritmos.

Localização

MAC0323 é o segundo passo na direção de

- Algoritmos
- Estruturas de Dados

Várias outras disciplina se apoiam em MAC0323.

Linguagem C

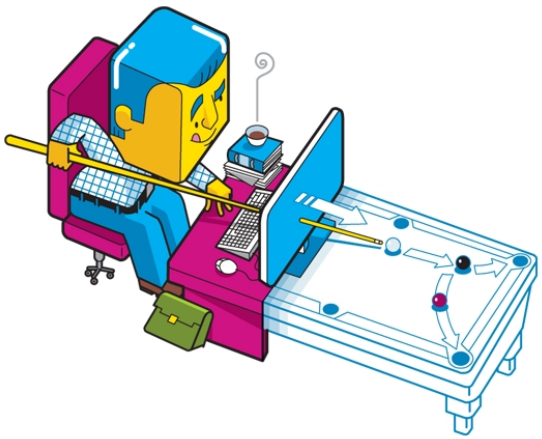
Usaremos a linguagem C.

Nada profundo.

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

AULA 1

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 a implementação de uma 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 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

```
void Init(int){
  [...]
}

      [...]
```

stack.c

Interface

```
void Init(int);
Item Pop();
void Push(Item);
int Empty();
        [...]
```

stack.h

Cliente

```
#include "stack.h"
int main(){
        [...]
   Init(100);
        [...]
```

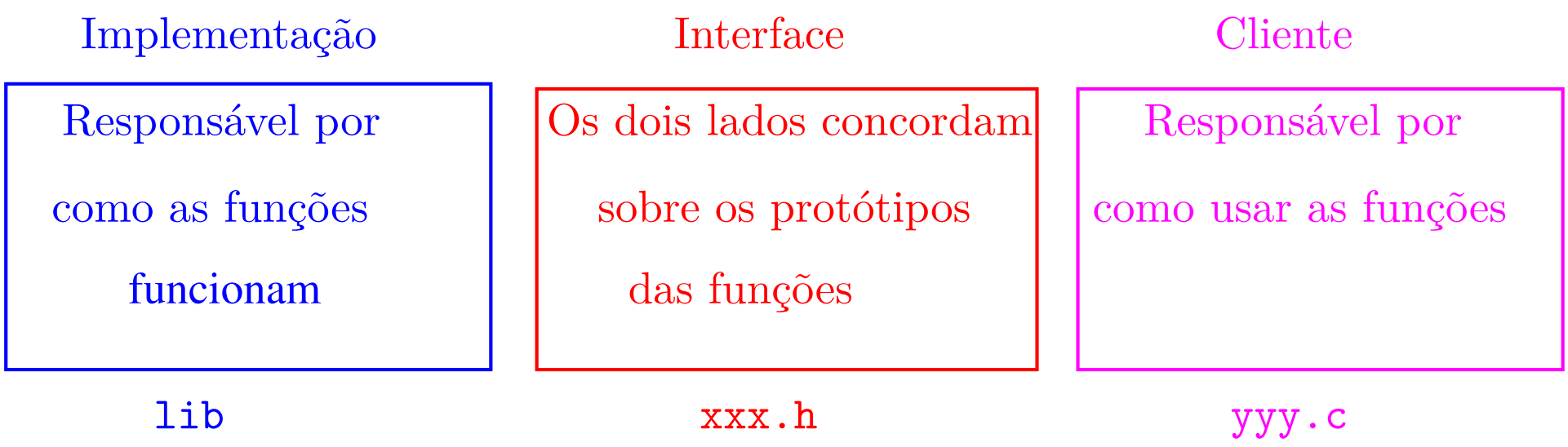
prog.c

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 interessam
os detalhes de implementação.



Interfaces

Entre as decisões de projeto estão

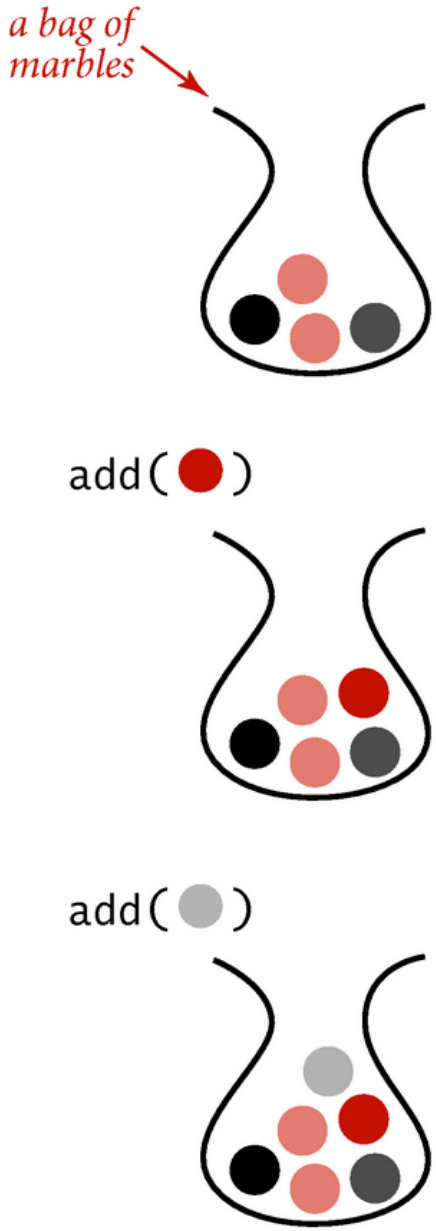
Interface: quais serviços serão oferecidos? A interface é um "contrato" entre o usuário e o projetista.

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.

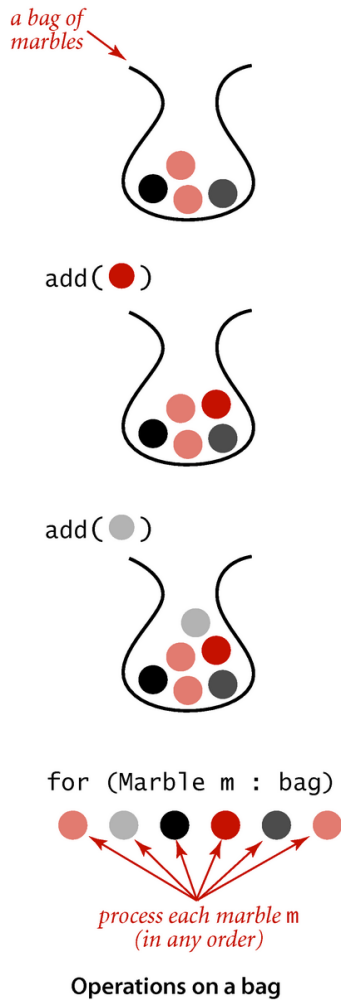
Recursos: quem é responsável pelo gerenciamento de memória e outros recursos?

Erros: quem detecta e reporta erros e como?

Sacos



Saco (= bag) e sua interface



Operations on a bag

Fonte: SW 3.1 https://algs4.cs.princeton.edu/13stacks/

PF: https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/bag.html

Que saco!

Um **saco** (=*bag*) é um ADT que consiste de uma coleção de itens com as seguintes operações:

- `Add`: insere um item na coleção,
- `IsEmpty`: decide se a coleção está vazia,
- `Size`: devolve o número de itens na coleção.

Que saco!

Um **saco** (=*bag*) é um ADT que consiste de uma coleção de itens com as seguintes operações:

- `Add`: insere um item na coleção,
- `IsEmpty`: decide se a coleção está vazia,
- `Size`: devolve o número de itens na coleção.

Em geral, vamos percorrer os itens da coleção.

A ordem em que se percorre os itens não é especificada.

Interface de um saco de inteiros

Arquivo	`Bag.h`	
`void`	`bagInit()`	cria um saco de inteiros vazio
`void`	`bagAdd(int item)`	coloca item neste saco
`bool`	`bagIsEmpty()`	este saco está vazio?
`int`	`bagSize()`	número de itens neste saco
`void`	`bagFree()`	destroi este saco
`void`	`bagStartIterator()`	inicializa o iterador
`bool`	`bagHasNext()`	há itens a serem iterados?
`int`	`bagNext()`	próximo item

As três últimas rotinas formam um `iterador`.

Interface Bag.h

```
/* Bag.h */

#define bool int

void bagInit();
void bagAdd(int);
bool bagIsEmpty();
int bagSize();
void bagFree();

void bagStartIterator();
bool bagHasNext();
int bagNext();
```

Cliente

```
#include "Bag.h"
int main(){
    int i;
    bagInit();
    for (i = 10; i < 20; i++)
        bagAdd(i);
    printf("%d\n", bagSize());
    bagStartIterator();
    while (bagHasNext())
        printf("%d ", bagNext());
    printf("\n");
    bagFree();
    return(0);
}
```

Implementação Bag.c

```
#include "Node.h"
#include "Bag.h"

Link first;
int n;
void bagInit() {...}
void bagAdd(int item) {...}
bool bagIsEmpty() {...}
int bagSize() {...}
void bagFree() {...}

Link current;
void bagStartIterator() {...}
bool bagHasNext() {...}
int bagNext() {...}
```

Interface Node.h

```
/* Node.h */

typedef struct stackNode *Link;

struct stackNode{
   int item;
   Link next;
};

Link newNode(int, Link);

void freeNode(Link);
```

Implementação Node.c

```
/* Node.c */

#include "Node.h"

Link newNode(int item, Link next) {
   Link p = mallocSafe(sizeof(*p));
   p->item = item;
   p->next = next;
   return p;
}

void freeNode(Link p) {
   free(p);
}
```

Implementação Bag.c

```
#include "Node.h"
#include "Bag.h"

void bagInit() {
   first = NULL;
   n = 0;
}

void bagAdd(int item) {
   first = newNode(item, first);
   n++;
}

bool bagIsEmpty() {
   return n == 0;
}
```

Implementação Bag.c

```
int bagSize() {
   return n;
}

void bagFree() {
   Link p;
   while (first != NULL) {
       p = first;
       first = first->next;
       freeNode(p);
   }
   n = 0;
}
```

Implementação Bag.c

```
void bagStartIterator() {
   current = first;
}

bool bagHasNext() {
   return current != NULL;
}

int bagNext() {
   int item = current->item;
   current = current->next;
   return item;
}
```

Interface de um saco genérico

Uma característica essencial de ADTs de coleções é que possam ser usadas com qualquer tipo de itens.

Arquivo	`Bag.h`	
`void`	`bagInit()`	cria um saco de itens vazio
`void`	`bagAdd(Item item)`	coloca item neste saco
`bool`	`bagIsEmpty()`	este saco está vazio?
`int`	`bagSize()`	número de itens neste saco
`void`	`bagFree()`	destroi este saco
`void`	`bagStartIterator()`	inicializa o iterador
`bool`	`bagHasNext()`	há itens a serem iterados?
`Item`	`bagNext()`	próximo item

Interface Bag.h genérica

```
/* Bag.h */

#include "Item.h"

#define bool int

void bagInit();
void bagAdd(Item);
bool bagIsEmpty();
int bagSize();
void bagFree();

void bagStartIterator();
bool bagHasNext();
Item bagNext();
```

Exemplo de Item.h

```
/* Item.h */

#ifndef HEADER_Item

#define HEADER_Item

typedef char *Item;

#endif
```

Implementação Bag.c genérica

```
#include "Node.h"
#include "Bag.h"

Link first;
int n;
void bagInit() {...}
void bagAdd(Item item) {...}
bool bagIsEmpty() {...}
int bagSize() {...}
void bagFree() {...}

Link current;
void bagStartIterator() {...}
bool bagHasNext() {...}
Item bagNext() {...}
```

Interface Node.h genérica

```
/* Node.h */

#include "Item.h"

typedef struct stackNode *Link;

struct stackNode{
   Item item;
   Link next;
};

Link newNode(Item, Link);

void freeNode(Link);
```

Implementação Node.c genérica

```
/* Node.c */

#include "Node.h"

Link newNode(Item item, Link next) {
   Link p = mallocSafe(sizeof(*p));
   p->item = item;
   p->next = next;
   return p;
}

void freeNode(Link p) {
   free(p);
}
```

Outro cliente

```
#include "Bag.h"
int main(int argc, char *argv[]) {
    int i;
    bagInit();
    for (i = 1; i < argc; i++)
        bagAdd(argv[i]);
    printf("%d\n", bagSize());
    bagStartIterator();
    while (bagHasNext())
        printf("%s ", bagNext());
    printf("\n");
    bagFree();
    return(0);
}
```

Implementação Bag.c

Só se alteram as seguintes rotinas:

```
void bagAdd(Item item) {
   first = newNode(item, first);
   n++;
}

Item bagNext() {
   Item item = current->item;
   current = current->next;
   return item;
}
```

Observações

Fonte: filmeseriale.info

Ao longo do semestre usaremos `Bag`s frequentemente.

Em particular, usaremos `Bag`s em uma das nossas implementações de *grafos* e *digrafos*.

Listas encadeadas em C

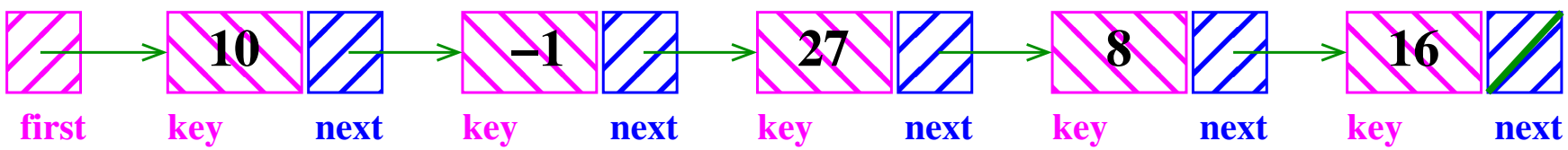
PF 4, S 3.3

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

Listas encadeadas

Uma **lista encadeada** (= *linked list* = lista ligada) é uma sequência de células; cada célula contém um objeto de algum tipo e o endereço da célula seguinte.

Ilustração de uma lista encadeada ("sem cabeça")

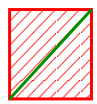


Estrutura para listas encadeadas em C

```
struct celula {
   int conteudo;
   struct celula *prox;
};

typedef struct celula Celula;

/* Cria lista inicialmente vazia */
Celula *ini = NULL;
```



ini

Endereços listas encadeadas

O endereço de uma lista encadeada é
o endereço de sua primeira célula.

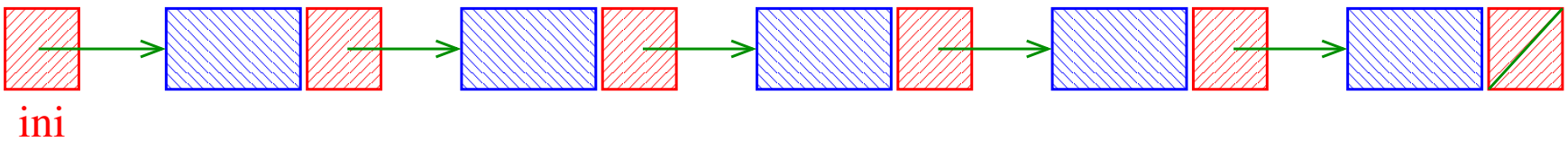
Se `p` é o endereço de uma lista,
às vezes dizemos que "`p` é uma lista".

Se `p` é uma lista então

- `p == NULL` ou
- `p->prox` é uma lista.

Imprime conteúdo de uma lista

Esta função imprime o `item` de cada célula de uma lista encadeada `ini`.




```
void imprima (Celula *ini) {
   Celula *p;
   for (p = ini; p != NULL; p = p->prox)
      printf("%d\n", p->item);
}
```

Busca em listas encadeadas

Esta função recebe um inteiro `x` e uma lista `ini`.
Ela devolve o endereço de uma célula que contém `x`.
Se tal célula não existe, a função devolve `NULL`.

```
Celula *busca (int x, Celula *ini) {
   Celula *p;
   p = ini;
   while (p != NULL && p->item != x)
       p = p->prox;
   return p;
}
```