MAC00323 Algoritmos e Estruturas de Dados II

Edição 2018

Sobre MAC0323

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

Sobre MAC0323

- página da disciplina
- responsáveis
- Livro
- pré-requisitos
- aulas
- exercícios-programa
- projeto
- provas e provinhas

Página da disciplina

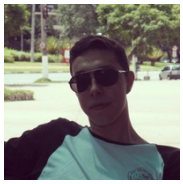
Paca: https://paca.ime.usp.br

Your heart is true. You may pass.
`"Amo estudar algoritmos!"`,
sem aspas

Ambiente de programação, EPs, critérios, fóruns . . .

Responsáveis


Victor


Lana


Coelho

Pré-requisitos

MAC0121 Algoritmos e Estruturas de Dados I

Prof. Teoria

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*

Exercícios-programa

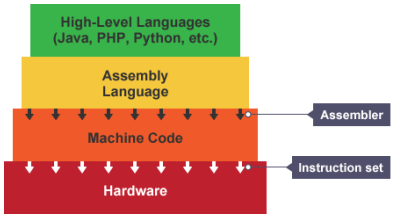
Em MAC0323 teremos EPs em

Vários EPs serão chupados de COS226 de Princeton

EP01: disponível na página, para 14/MAR

Projeto

Montador (*assembler*) e Viculador (*linker*)



Fonte: https://www.bbc.co.uk/education

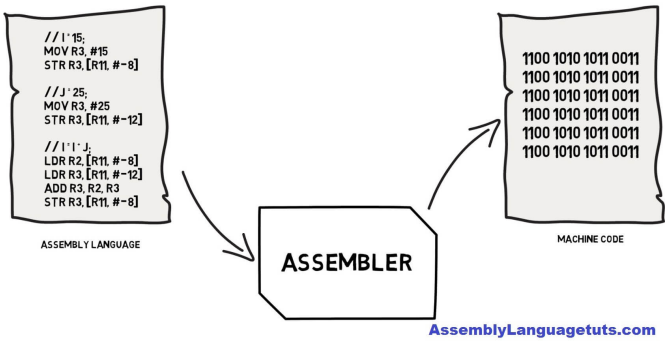
Autor: Fernando Mário de Oliveira

Projeto: em C

https://twitter.com/slidenerdtech

Projeto: em C

Fases: PR01, PR02, PR03, PR04, PR05.
PR01: disponível na página, para 04/ABR



Provas e provinhas

3 provas

Várias provinhas de até 10 minutos

Médias das provinhas vale como PSub

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

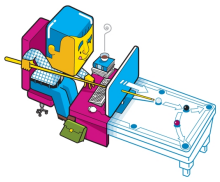
Usaremos a linguagem Java e C.

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 :-)

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 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.

Implementação	Interface	Cliente
Responsável por como as funções funcionam	Os dois lados concordam sobre os protótipos das funções	Responsável por como usar as funções
lib	xxx.h	yyy.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 interessa os detalhes de implementação.

Implementação	Interface	Cliente
Responsável por como as funções funcionam	Os dois lados concordam sobre os protótipos das funções	Responsável por como usar as funções
lib	API	Prog.java

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?

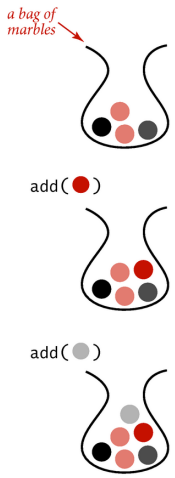
Erros: quem detecta e reporta erros e como?

Sacos

Fonte: Saco (= bag) e sua API

PF 6.1 e 6.3
https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/bag.html

Sacos



Que saco!

Um **saco** (=*bag*) é uma ADT que consiste de uma coleção de itens munida de duas operações:

- `add()` que insere um item na coleção, e
- `iterator()` que percorre os itens da coleção. A ordem em que o iterador percorre os itens não é especificada.

API de um saco de inteiros

`public class`	`BagInteger`	
	`BagInteger()`	cria um saco de inteiros vazio
`void`	`add(Integer item)`	coloca item neste saco
`boolean`	`isEmpty()`	este saco está vazio?
`int`	`size()`	número de itens neste saco
`void`	`startIterator()`	inicializa o iterador
`boolean`	`hasNext()`	há itens a serem iterados?
`Integer`	`next()`	próximo item

Cliente

```
public class Cliente {
  public static void main(String args[]){
     BagInteger bag = new BagInteger();
     for (int i=10; i < 20; i++) {
        bag.add(i);
     }
     StdOut.println(bag.size());
     bag.startIterator();
     while (bag.hasNext()) {
        StdOut.println(bag.next());
     }
  }
}
```

API de um saco genérico

public class	Bag<Item>	
	Bag()	cria um saco de itens vazio
void	add(Item item)	coloca item neste saco
boolean	isEmpty()	este saco está vazio?
int	size()	número de itens neste saco
void	startIterator()	inicializa o iterador
boolean	hasNext()	há itens a serem iterados?
Item	next()	próximo item

Generics

Uma característica essencial de ADTs de coleções é permitir que sejam usadas para qualquer tipo de itens.

O mecanismo em Java conhecido como **genéricos** (=*generics*) permite essa capacidade.

A notação <Item> depois do nome da classe define o nome Item como um parâmetro de tipo, um espaço reservado para um tipo concreto ser usado pelo cliente.

Lemos Bag<Item> como *saco de itens* ou *bag de itens*.

Cliente

```
public class Cliente {
  public static void main(String args[]){
      Bag<String> bagS=new Bag<String>();
      bagS.add("Como "); bagS.add("é ");
      bagS.add("bom ");
      bagS.add("estudar ");
      bagS.add("MAC0323!");
      StdOut.println(bagS.size());
      bagS.startIterator();
      while (bagS.hasNext()) {
          StdOut.println(bagS.next());
      }
  }
}
```

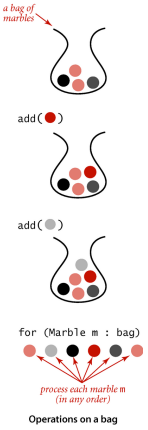
API: saco genérico iterável

public class	Bag<Item>	implements iterable<Ite
	Bag()	cria um saco de itens vazio
void	add(Item item)	coloca item neste saco
boolean	isEmpty()	este saco está vazio?
int	size()	número de itens neste saco
iterator<Item>	iterator()	iterador de itens

Cliente

```
public class Cliente {
  public static void main(String args[]){
      Bag<String> bagS=new Bag<String>();
      bagS.add("Como "); bagS.add("é ");
      bagS.add("bom ");
      bagS.add("estudar ");
      bagS.add("MAC0323!");
      StdOut.println(bagS.size());
      Iterator<String> it =
                bagS.iterator();
      while (it.hasNext()) {
          StdOut.println(it.next());
      }
  }
```

foreach



Frequentemente o cliente precisa apenas processar cada item de uma coleção iterável de alguma maneira. Para isso podemos iterar sobre os items da coleção com um comando do tipo foreach.

```
Bag<String> bagS =
      new Bag<String>();
[...]
for (String s:  bagS)
  StdOut.println(s);
```

Cliente de Luxe

```
public class Cliente {
  public static void main(String args[]){
      Bag<String> bagS=new Bag<String>();
      bagS.add("Como "); bagS.add("é ");
      bagS.add("bom ");
      bagS.add("estudar ");
      bagS.add("MAC0323!");
      StdOut.println(bagS.size());
      for (String s:  bagS) {
          StdOut.println(s);
      }
  }
}
```

Receita para construir uma classe iterável

Leia Bags, Queues, and Stacks (SW)
ou Saco (= bag) e sua API (PF).

Passo 0: incluir

```
import java.util.Iterator;
```

para que possamos nos referir a interface `java.util.Iterator`:

Receita para construir uma classe iterável

Passo 1: adicionar no final da declaração da classe `implements Iterable<Item>`.
Isso indica que o objeto será iterável e nos comprometemos a especificar o método `iterator()`, como especificado na interface `java.lang.Iterable`

```
public interface Iterable<Item> {
  public Iterator<Item> iterator();
}
```

Por exemplo:

```
public class Bag<Item> implements Iterable<Item>{
  [...]
}
```

Receita para construir uma classe iterável

Passo 2: implementar um método `iterator()` como prometido. Esse método retorna um objeto da classe que implementa a interface `Iterator`

```
public interface Iterator<Item> {
  boolean hasNext();
  Item next();
  void remove();
}
```

Por exemplo:

```
public Iterator<Item> iterator() {
  return new BagIterator();
}
```

Receita para construir uma classe iterável

Passo 3: implemente a subclasse que implementa a interface `Iterator` incluindo od método `hasNext()`, `next()` e `remove()`

Usamos sempre o método vazio para o opcional método `remove()` pois intercalar iteração com uma operação que modifica a estruturas de dados é melhor ser evitada.

Receita para construir uma classe iterável

```
private class
BagIterator implements Iterator<Item> {
  private Node current = first;
  public boolean hasNext() {
      return current != null;
  }
  public Item next() {
      Item item = current.item;
      current = current.next;
      return item;
  }
  public void remove() {
      throw new UnsupportedOperationException();
  }
```

Listas encadeadas em Java

SW 1.3

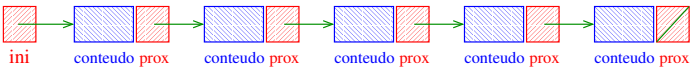
https://algs4.cs.princeton.edu/13stacks/

Linked lists, Victor S.Adamchik, CMU, 2009

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")



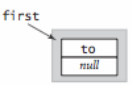
Estrutura para listas encadeadas em Java

É conveniente tratar as células como um novo tipo-de-dados e atribuir um nome a esse novo tipo:

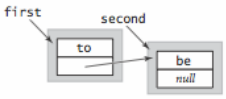
```
private class Node{
   Item item;
   Node next;
}
first = null;
```

Construir uma lista ligada

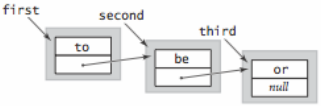
```
Node first  = new Node();
first.item  = "to";
```



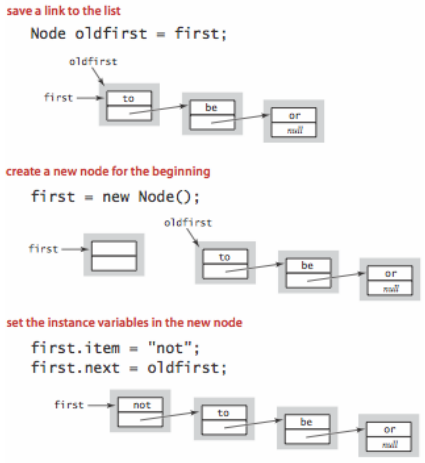

```
Node second = new Node();
second.item = "be";
first.next  = second;
```




```
Node third  = new Node();
third.item  = "or";
second.next = third;
```

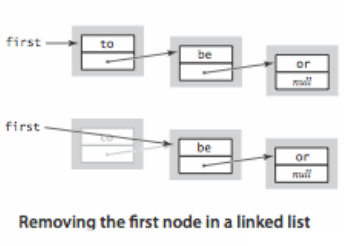


Inserir no início



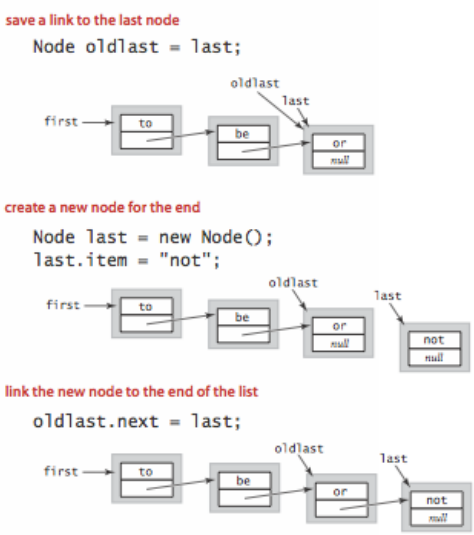
Remover do início

```
first = first.next;
```



Removing the first node in a linked list

Inserir no final



Listas encadeadas em C

PF 4, S 3.3

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

Percorrer

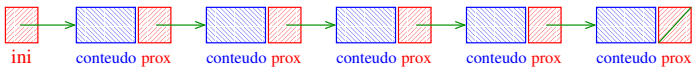
O seguinte trecho de código percorre uma lista ligada.

```
for (Node x = first; x != null; x = x.next) {
   // process    .item
}
```

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;

Celula *ini;
/* inicialmente a lista esta vazia */
ini = NULL;
```

Endereços listas encadeadas

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

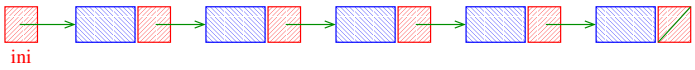
Se `p` é o endereço de uma lista às vezes dizemos que "`p` é uma lista".

Se `p` é uma lista então

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

Imprime conteúdo de uma lista

Esta função imprime o `conteudo` 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->conteudo);
}
```

Busca em listas encadeadas

Esta função recebe um inteiro `x` e uma lista `ini`. A função 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->conteudo != x)
      p = p->prox;

  return p;
}
```