

# SCC 202: Algoritmos e Estruturas de Dados I

- Profa. Graça Nunes
- 2o. Semestre de 2009



TAD



Tipos Abstratos de Dados

# Objetivos

---

- Introduzir conceitos de Estruturas de Dados básicas e seus algoritmos, que são frequentemente usados na construção de programas
  - Listas Lineares
  - Árvores
- Analisar alternativas para sua implementação
- Construir TAD que possam ser utilizados em outras aplicações

# TADs e termos relacionados

---

- Termos relacionados, mas diferentes
  - Tipo de dados
  - Tipo abstrato de dados
  - Estrutura de dados

# Tipo de dados

---

- Em linguagens de programação, o tipo de uma variável define o conjunto de **valores** que ela pode assumir e o conjunto de **operações** válidas sobre ele.
  - Por exemplo, uma variável booleana pode ser *true* ou *false*. Os operadores válidos são os lógicos e relacionais.
- Novos tipos de dados podem ser definidos em função dos existentes: tipos estruturados
  - Arrays, registros, etc.

# Tipo de dados

---

- A declaração de uma variável especifica:
  - Quantidade de bytes que deve ser reservada a ela
    - Variação entre linguagens
  - Como o dado representado por esses bytes deve ser interpretado
    - Valor inteiro, real, caracter, valor lógico
  - Quais operadores podem ser aplicados a ela
    - soma, multiplicação, etc. (inteiros e reais)
    - *e*, *ou*, *not* (boolean)
    - =, <, >, ... (relacionais)

# Tipo X Tipo Abstrato de dados

---

## ■ Perspectivas

- Computador: formas de se interpretar o conteúdo da memória
- Usuário: o que pode ser feito em uma linguagem, sem se importar como isso é feito em baixo nível
  - Conceito
  - Abstração
  - Independência da representação

# Tipo abstrato de dados

---

- Tipo de dados divorciado da implementação
  - Definido pelo par  $(v,o)$ 
    - v: valores, dados a serem manipulados
    - o: operações sobre os valores/dados
  - Não importa como é o mapeamento do conceito no computador, ou seja, como os valores serão representados.



# TAD dicionário inglês-português

---




- Dados (v)
  - Pares de palavras
- Operações (o)
  - Buscar tradução de uma palavra
  - Inserir novo par de palavras
  - Eliminar um par de palavras
  - Alterar informação

# Tipo Abstrato X Estrutura de dados

---

- Uma vez que um TAD é definido, escolhe-se a estrutura de dados mais apropriada para representá-lo.
- A Estrutura de dados deve explicitar a forma de representar os dados na memória.
- Exemplo:
  - Dados: Pares de Palavras (Português, Inglês)
  - ED: Lista de pares de palavras
  - Tipo: Array de pares de palavras

# Tipo abstrato de dados

mundo real	dados de interesse	ESTRUTURA de armazenamento	possíveis OPERAÇÕES
 pessoa	<ul style="list-style-type: none"><li>a idade da pessoa</li></ul>	<ul style="list-style-type: none"><li>tipo inteiro</li></ul>	<ul style="list-style-type: none"><li>nasce (<math>i \leftarrow 0</math>)</li><li>aniversário (<math>i \leftarrow i + 1</math>)</li></ul>
 cadastro de funcionários	<ul style="list-style-type: none"><li>o nome, cargo e o salário de cada funcionário</li></ul>	<ul style="list-style-type: none"><li>tipo lista ordenada</li></ul>	<ul style="list-style-type: none"><li>entra na lista</li><li>sai da lista</li><li>altera o cargo</li><li>altera o salário</li></ul>
 fila de espera	<ul style="list-style-type: none"><li>nome de cada pessoa e sua posição na fila</li></ul>	<ul style="list-style-type: none"><li>tipo fila</li></ul>	<ul style="list-style-type: none"><li>sai da fila (o primeiro)</li><li>entra na fila (no fim)</li></ul>

# Características do TAD

---

- A característica essencial de um **TAD** é a **separação** entre **conceito** e **implementação**.
- O termo "**ocultamento de informação**" é utilizado para descrever esta habilidade.
- Ao usuário do TAD são fornecidos a descrição dos valores e o conjunto de operações do TAD, mas a implementação é **invisível**.
- Ao criador do TAD, cabe ocultar a implementação.

# Características do TAD

---

- A separação da **definição** do TAD de sua **implementação** permite que a mudança de implementação (pelo criador) não altere o programa (do usuário) que usa o TAD.
- O **TAD** é **compilado separadamente**, e uma mudança força somente a recompilação do arquivo envolvido e uma nova link-edição do programa (mais rápida que uma nova compilação do programa).

# Tipo abstrato de dados

---

- Nas linguagens que suportam TADs, os detalhes representacionais dos tipos de dados podem ser suprimidos.
- Os módulos são instalados em uma **biblioteca** e podem ser reutilizados por vários programas.

# Tipo Abstrato de Dados (TAD)

## ■ Para Programar um TAD:

- Programador descreve o TAD em dois módulos separados
  - Um módulo contém a **interface** de acesso (TAD conceitual):
    - apresenta as operações e valores possíveis
  - Outro módulo, de **implementação**, contém a representação da estrutura de dados e a implementação de cada operação
- Outros programadores podem, por meio da interface de acesso, usar o TAD sem conhecer os detalhes de representação e sem acessar o módulo de implementação
  - Idealmente, a implementação é “invisível” e inacessível
  - Ex. pode criar uma lista de clientes e aplicar operações sobre ela, mas não sabe como ela é representada internamente

# Modularização em C (Revisão)

---

- Programa em C pode ser dividido em vários arquivos
  - Arquivos **fonte** com extensão **.c**
    - Denominados de módulos
- Cada módulo deve ser compilado separadamente
  - Para tanto usa-se um **compilador**
  - Resultado são arquivos **objeto** não executáveis
    - Arquivos em linguagem de máquina com extensão **.o** ou **.obj**
- Arqs. objeto devem ser juntados em um **executável**
  - Para tanto usa-se um *ligador* ou **link-editor**
  - Resultado é um único arquivo em linguagem de máquina
    - Usualmente com extensão **.exe**



# Modularização em C (Revisão)

---

- Módulos são muito úteis para construir bibliotecas de funções inter-relacionadas. Por exemplo:
  - Módulos de funções matemáticas
  - Módulos de funções para manipulação de strings
  - etc.
- Em C, é preciso listar, no início de cada módulo, aquelas funções de outros módulos que serão utilizadas:
  - Isso é feito através de uma lista denominada **cabeçalho**
- **Exemplo:** considere um arquivo STR.c contendo funções para manipulação de strings, dentre elas:
  - **int** comprimento (**char\*** strg)
  - **void** copia (**char\*** dest, **char\*** orig)
  - **void** concatena (**char\*** dest, **char\*** orig)

# Modularização em C (Revisão)

- **Exemplo** (cont): Qualquer módulo que utilizar essas funções deverá incluir no início o cabeçalho das mesmas, como abaixo.

```
/* Programa Exemplo.c */

#include <stdio.h>

int comprimento (char* str);
void copia (char* dest, char* orig);
void concatena (char* dest, char* orig);

int main (void) {
    ...
}
```

# Modularização em C (Revisão)

- **Exemplo** (cont):

- A partir desses dois fontes (Exemplo.c e STR.c), podemos gerar um executável compilando cada um separadamente e depois ligando-os
- Por exemplo, com o compilador Gnu C (gcc) utilizaríamos a seguinte seqüência de comandos para gerar o arquivo executável Teste.exe:

```
> gcc -c STR.c  
  
> gcc -c Exemplo.c  
  
> gcc -o Teste.exe STR.o Exemplo.o
```

- **Questão:**

- É preciso inserir manualmente e individualmente todos os cabeçalhos de todas as funções usadas por um módulo?
  - E se forem muitas e de diferentes módulos?

# Modularização em C (Revisão)

---

## ■ Solução

- **Arquivo de cabeçalhos** associado a cada módulo, com:
  - cabeçalhos das funções oferecidas pelo módulo e,
  - eventualmente, os tipos de dados que ele **exporte**
    - typedef's, struct's, etc.
- Segue o mesmo nome do módulo ao qual está associado
  - porém com a extensão **.h**

## ■ Exemplo:

- Arquivo STR.h para o módulo STR.c do exemplo anterior

# Modularização em C (Revisão)

```
/* Arquivo STR.h */

/* Função comprimento:
   Retorna o no. de caracteres da string str */
int comprimento (char* str);

/* Função copia:
   Copia a string orig para a string dest */
void copia (char* dest, char* orig);

/* Função concatena:
   Concatena a string orig na string dest */
void concatena (char* dest, char* orig);
```

# Modularização em C (Revisão)

- O programa Exemplo.c pode então ser reescrito como:

```
/* Programa Exemplo.c */

#include <stdio.h> /* Módulo da Biblioteca C Padrão */
#include "STR.h"    /* Módulo Próprio */

int main (void) {
    ...
}
```

Nota: O uso dos delimitadores < > e " " indica onde o compilador deve procurar os arquivos de cabeçalho, na biblioteca interna (<>) ou no diretório indicado (" " – *default* se ausente).

# TADs em C

---

- **Módulos** podem ser usados para definir um novo tipo de dado e o conjunto de operações para manipular dados desse tipo:
  - Tipo Abstrato de Dados (TAD)
- Definindo um tipo *abstrato*, pode-se “esconder” a implementação:
  - Quem usa o tipo abstrato precisa apenas conhecer a funcionalidade que ele implementa, não a forma como ele é implementado
  - Facilita manutenção e re-uso de códigos, entre outras vantagens
- Em C podemos, por exemplo, criar um TAD para representar e operar de maneira apropriada os números racionais:
  - Para isso define-se um tipo de dado “Racional” e as funções que o manipulam

# TADs em C: Exemplo

```
/* Racionais.h: Interface de TAD Números Racionais */

/* Tipo Exportado */

typedef struct {
    int Num, Den;
} Racional;

/* Funções Exportadas */

Racional Define(int N, int D);
/* Gera um número racional a partir de dois inteiros,
   sendo o segundo não nulo */

/* continua ... */
```



# TADs em C: Exemplo

```
/* ... continua */

Racional Soma(Racional R1, Racional R2);
/* Soma dois números racionais R1 e R2 e retorna o
   resultado */

Racional Multiplica(Racional R1, Racional R2);
/* Multiplica dois números racionais R1 e R2 e
   retorna o resultado */

int TestaIgualdade(Racional R1, Racional R2);
/* Verifica se 2 números racionais R1 e R2 possuem
   numeradores e denominadores iguais. Retorna 1
   nesse caso e 0 caso contrário */
```

# TADs em C: Exemplo

---

- Implementação (Arquivo Racionais.c):

# TADs em C: Exercício

```
#include <stdio.h>
#include "Racionais.h"

void main(void){

    /* Teste do TAD: Exercício... */

}
```

# TADs em C: Observações

---

- Os tipos de dados e funções descritos no arquivo de cabeçalho Racionais.h são exportados para os módulos que incluem esse arquivo via `#include "Racionais.h"`
  - são visíveis para os “clientes” do TAD
    - por exemplo, arquivo `Teste_Racionais.c`
- Funções e tipos de dados para fins exclusivos da implementação interna do TAD não devem constar no arquivo de cabeçalho, apenas no arquivo de implementação (`Racionais.c`)
  - não são exportados ao “cliente”
    - são inacessíveis para quem utiliza o TAD
    - cliente só precisa/deve ter acesso à versão compilada de `Racionais.c`

# Exercícios

---

- Escrever um programa C que executa operações sobre números racionais, utilizando o TAD construído.
- Re-implementar o TAD Racionais usando um vetor de dois elementos como tipo base, ao invés de um *struct* com dois campos como foi feito
  - O que é preciso alterar no programa do item anterior para essa nova versão do TAD?
- Complemente o TAD Racionais com uma função de teste de igualdade alternativa, que verifique a igualdade dos dois números sem a restrição de que seus numeradores e denominadores devam ser iguais para retorno verdadeiro

# Exercícios

---

- 1) TAD Número Complexo
- 2) TAD Conjuntos (Sets)  
(Aho et. al. , Capítulo 4)

exercícios sobre TAD em:

<http://www.icmc.usp.br/~sce182/extad.html>

# Tipos abstratos de dados em Pascal

---

- Uso de **Units**
- *Uma **Unit** é uma coleção de constantes, tipos de dados, variáveis, procedimentos e funções relativas a um determinado domínio.*
  - *Por exemplo, a unit CRT contém todas as declarações de rotinas relativas à SCREEN do PC.*
- *Cada Unit é como um programa Pascal separado. Ela é uma biblioteca de declarações que permite dividir seu programa e compilá-lo em partes separadas. Ela pode ter um corpo principal o qual é chamado antes do seu programa ser iniciado para preparar as "inicializações" necessárias.*
- *O Turbo Pascal possui várias Units pré-definidas:*
  - *System, Overlay, Graph, Dos, Crt, Printer...*

# Tipos abstratos de dados em Pascal

---

- Cada *Unit* tem 2 partes:

- *Interface*

- *Nesta parte, declara-se tudo o que os programas externos (usuários da Unit) devem conhecer para que possam utilizá-las: tipos de dados, cabeçalhos de funções e procedimentos.*

- *Implementation*

- *Nesta parte, aparece todo o código de implementação relativo aos subprogramas da Unit, tanto aqueles que serão usados pelos usuários, quanto os apenas internos à Unit, e não acessíveis aos usuários (que ajudam a definir os demais)*



# Estrutura de uma Unit

---

**UNIT** <identificador>;      {arquivo deve ser **identificador.PAS**}

**INTERFACE**

**uses** <lista de units>      {opcional}  
    <declarações públicas>      {só cabeçalho}

**IMPLEMENTATION**      {opções de representação + algoritmos}

**uses** <lista de units>      {opcional}  
    <declarações privadas>  
    <implementação de proc. e funções> {corpo das funções e procedimentos}

**Begin**  
        <inicializações>      {opcional}

**End**

**End.**

# Biblioteca de Operações sobre Racionais

## Unit Racional;

---

**Interface** {o que o usuário deve conhecer para usá-las}  
{definicão de tipo racional}

Type Rac = array [1..2] of integer;

Procedure Cria\_Racional(a: integer; b:integer; var r:Rac);  
{Cria um número racional a partir de dois inteiros se D <> 0,  
{caso contrário exibe mensagem explicativa}

Procedure Soma\_Racional (r1, r2: Rac; var soma\_rac: Rac);  
{Soma dois números racionais R1 e R2 e retorna o resultado em Result}

Procedure Mult\_Racional (r1, r2: Rac; var mult\_rac: Rac);  
{Multiplica dois números racionais R1 e R2 e retorna em Result}

Function Teste\_Igualdade (r1, r2: Rac): boolean;  
{Verifica se 2 números racionais R1 e R2 são iguais, e se sim retorna True,  
caso contrário False}

Procedure Imprime\_Racional( r: Rac);  
{imprime um numero racional}

# Biblioteca de Operações sobre Racionais

**Implementation** {o que o usuário não precisa conhecer}

{Implementação de procedimentos e funções}

**Procedure** Cria\_Racional(a: integer; b: integer; var r: Rac);

**Begin**

if b<>0 then {pré-condicao}

begin {pós-condicao}

r[1]:= a;

r[2]:=b;

end;

**End;**

**Procedure** Imprime\_Racional( r: Rac);

{Imprime racional - pós-condição}

**Begin**

writeln('racional criado:',r[1], '/',r[2]);

writeln;

**End;**

# Biblioteca de Operações sobre Racionais

```
Procedure Soma_Racional (r1, r2: Rac; var soma_rac:  
Rac);
```

```
Begin
```

```
    soma_rac[2] := r1[2] * r2[2];      {pos-condicao}
```

```
    soma_rac[1] := r1[1] * r2[2] + r2[1] * r1[2];
```

```
End;
```

```
Procedure Mult_Racional (r1, r2: Rac; var mult_rac:  
Rac);
```

```
Begin
```

```
    mult_rac[1] := r1[1] * r2[1];      {pos-condicao}
```

```
    mult_rac[2] := r1[2] * r2[2];
```

```
End;
```

```
Function Teste_Igualdade (r1, r2: Rac): boolean;
```

```
Begin
```

```
    Teste_Igualdade := false;          {pos-condicao}
```

```
    If (r1[1] * r2[2] = r1[2] * r2[1]) then
```

```
        Teste_Igualdade := true;
```

```
End;
```

```
End.
```

# Exemplo: TAD Número Racional

## ■ Programa que utiliza a Unit Racional

```
program racio;
uses Racional;
Var a,a1: integer;
    b,b1: integer;
    r0, r1, soma: Rac; {não precisa definir Rac}
Begin
    writeln('entre com os valores inteiros');
    readln(a,b);
    readln (a1,b1);
    Cria_Racional(a,b,r0);      {Cria um nro racional}
    Imprime_Racional(r0);      {Imprime um nro racional}
    Cria_Racional(a1, b1, r1); {Cria um nro racional}
    Imprime_Racional(r1);
    Soma_Racional(r0,r1,soma); {Cria uma soma}
    Imprime_Racional(soma);
    readln;
End.
```

# Tipo abstrato de dados

---

- Cuidados para a Interface de Acesso ao TAD:
  - Em termos de implementação, sugerem-se:
    - Passagem de parâmetros
      - Um parâmetro pode especificar um objeto em particular, deixando a operação genérica
  - *Flag* para erro, sem emissão de mensagem no código principal
    - Independência do TAD

# Tipo Abstrato de Dados (TAD)

---

- Em Resumo:

- Os módulos de um TAD podem ser instalados em uma biblioteca e reutilizados por vários programas
  - A execução do programa requer a link-edição do programa com os módulos de implementação (usualmente mantidos já pré-compilados na biblioteca)
  - O programador não precisa olhar o código do módulo de implementação para usar o TAD!
  - Basta conhecer a interface de acesso

# Tipo abstrato de dados

---

- Vantagens quando se dispõe de um TAD:
  - Mais fácil programar
    - Não é necessário se preocupar com detalhes de implementação
    - Logicamente mais claro
  - Mais seguro programar
    - Apenas os operadores podem mexer nos dados
  - Maior independência, portabilidade e facilidade de manutenção do código
  - Maior potencial de reutilização de código
  - Abstração
- Conseqüência: custo menor de desenvolvimento