

Estrutura de Dados

Tipo Abstrato de Dados (TAD)

**Baseado nos slides do Prof. Luiz Gustavo Almeida
Martins e da Profa. Gina M. B. Oliveira**

Introdução

Na construção de **bons modelos computacionais** é necessário expressar os **detalhes relevantes do problema** que se deseja modelar através de uma **estrutura de dados adequada** e desenvolver **um algoritmo eficiente** que atue sobre essa estrutura.

Programas = Estrutura de Dados + Algoritmos

Introdução

- **Estrutura de Dados:**
 - Estruturação **conceitual** dos dados
 - Reflete um **relacionamento lógico entre dados** de acordo com o problema considerado
- **Algoritmos:**
 - **Implementação** das operações sobre os dados
 - Reflete as **ações sobre a estrutura de dados** necessárias para o problema considerado

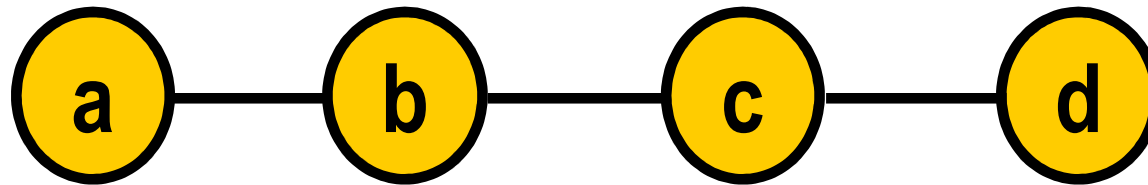
Introdução

- **Estrutura de Dados:**
 - Estruturação **conceitual** dos dados
 - Reflete um **relacionamento lógico entre dados** de acordo com o problema considerado
- **Algoritmos:**
 - **Implementação** das operações sobre os dados
 - Reflete as **ações sobre a estrutura de dados** necessárias para o problema considerado

Exemplo de Estrutura de Dados

- **Listas:**

- Estruturas **lineares** sequenciais
- **Relação de ordem** entre os dados



Exemplo de aplicação:

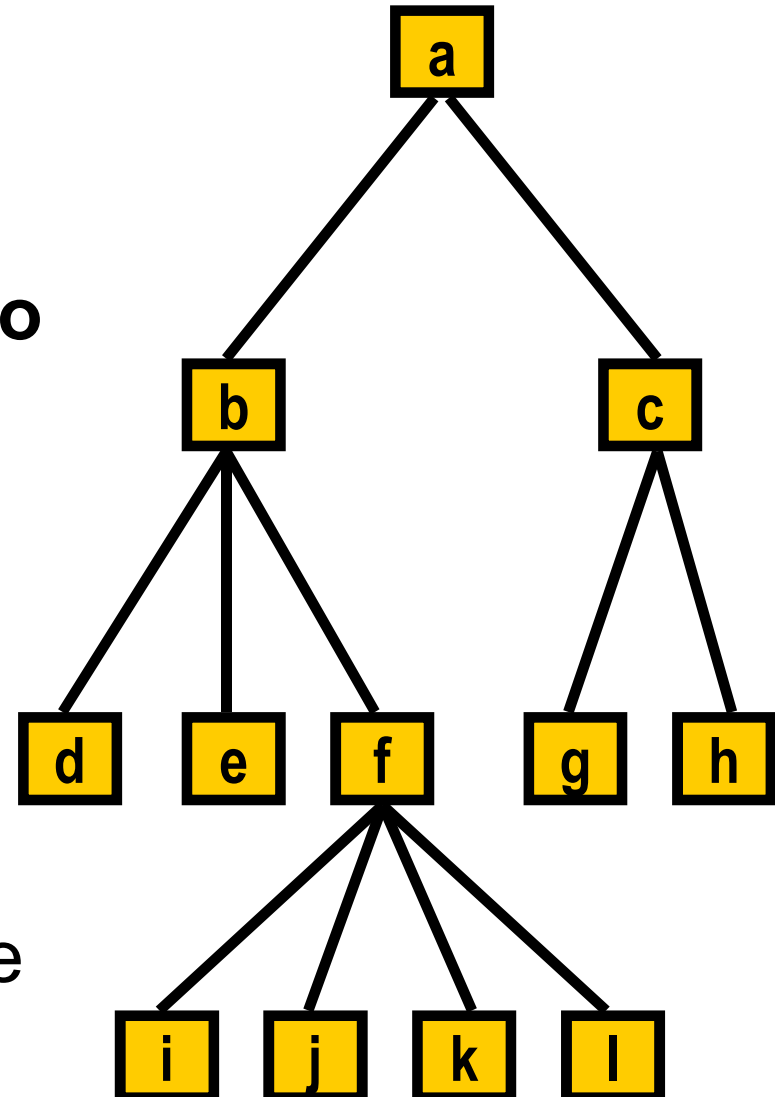
- Listas de funcionários de uma empresa

Exemplo de Estrutura de Dados

- **Árvores:**
 - Estruturas **espaciais**
 - **Relação de subordinação** entre os dados

Exemplo de aplicação:

- Organograma funcional de uma empresa



Tipos de Dados

- Define o **conjunto de valores** (domínio) que uma variável pode assumir

Exemplo:

– Dado inteiro:

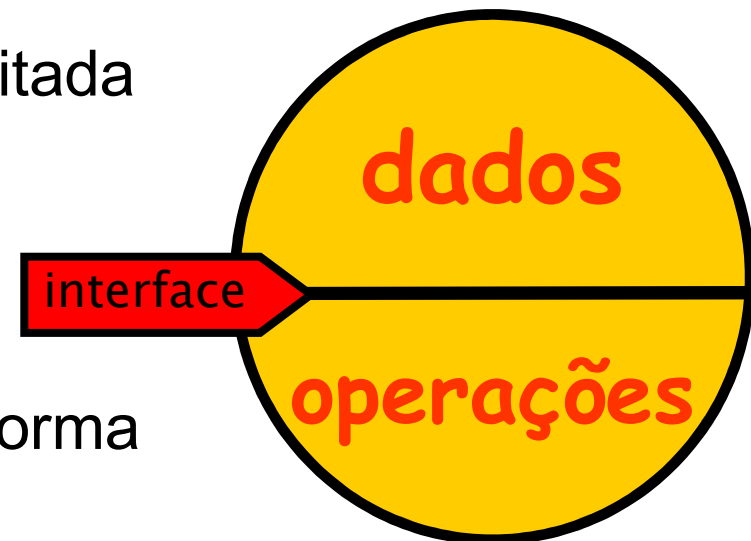
$\{ -N, \dots, -3, -2, -1, 0, +1, +2, +3, \dots, +(N-1) \}$

– Dado lógico:

$\{ \textit{verdadeiro}, \textit{falso} \}$

Tipo Abstrato de Dados (TAD)

- Forma de definir um **novo tipo de dado** e as **operações** que o manipulam
 - Baseada na definição de **tipos estruturados**
- **Ideia central:** encapsular (**esconder**) de quem usa o TAD a forma como ele foi implementado
 - Visibilidade da estrutura fica limitada às operações
 - Cliente tem acesso somente à forma abstrata do TAD



Tipo Abstrato de Dados (TAD)

- Um TAD é definido por:
 - Um **conjunto de valores** (dados)
 - Atributos/campos da estrutura
 - Define o que a estrutura deve representar
 - Um **conjunto de operações** que atuam sobre esses valores
 - Determinam as **ações** realizadas na manipulação dos dados do TAD
 - Devem ser consistentes com os tipos utilizados

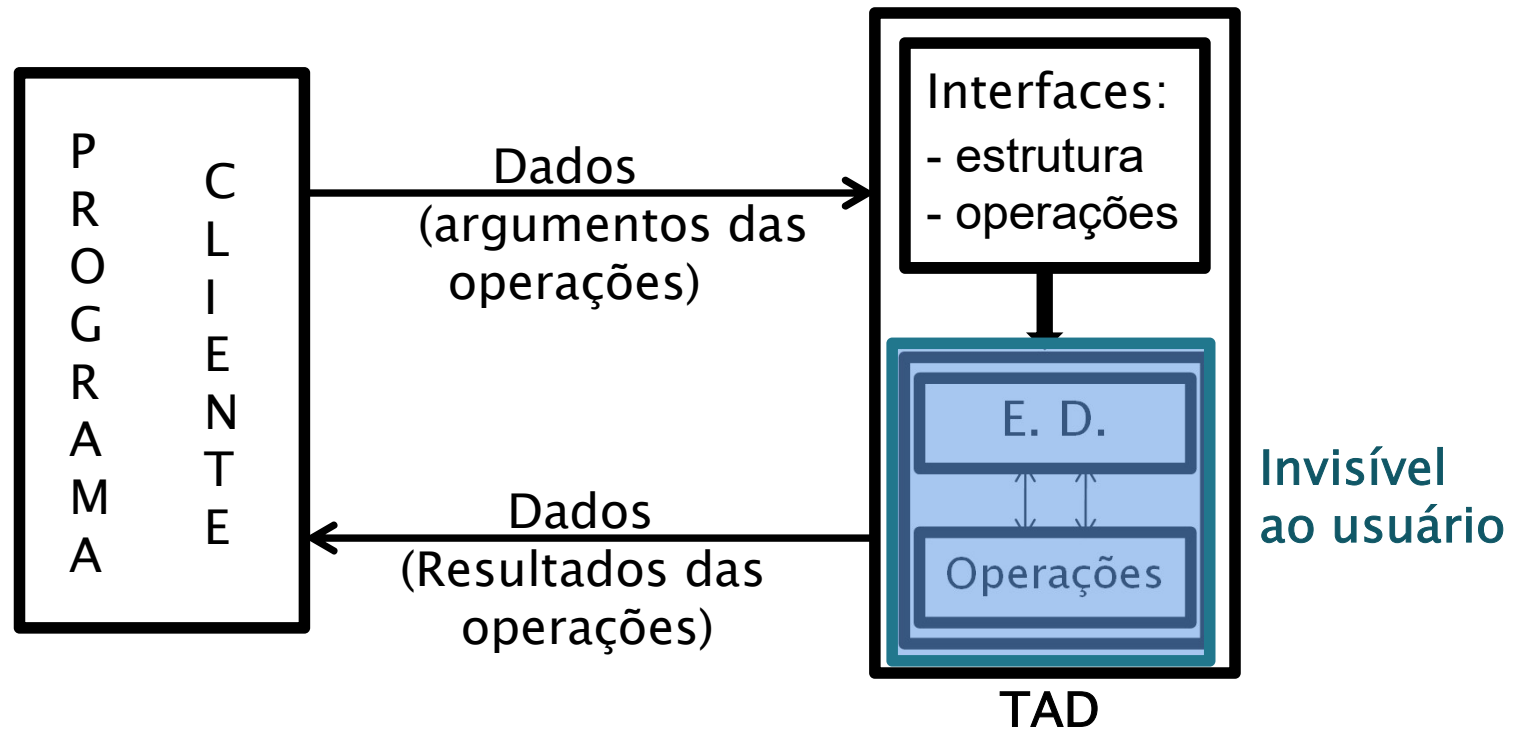
Exemplos de Operações

- **Criação** de uma instância da estrutura
- **Inclusão** de um novo elemento
- **Remoção** de um elemento existente
- **Acesso** ao conteúdo de um elemento
- **Liberação** de uma instância da estrutura

Tipo Abstrato de Dados (TAD)

- Permite a separação entre o conceito (**o que fazer**) e a implementação (**como fazer**)
- **Vantagens:**
 - **Encapsulamento e segurança:**
 - Somente o conceito do TAD é visível externamente
 - Usuário **NÃO** tem acesso direto aos dados
 - Acesso somente através das operações
 - **Flexibilidade e reutilização:**
 - **Compilação separada:** mudança na implementação do TAD não afeta o programa que o utiliza (programa usuário ou cliente)
 - As interfaces das operações devem ser mantidas

Iteração entre o TAD e Programa Usuário



Especificação de umTAD

- Define a **parte conceitual** do TAD (**o que deve ser feito**)
 - Determina as interfaces
- Deve conter as seguintes informações:
 - **Cabeçalho:**
 - **Nome:** identificação do TAD
 - **Dados:** descrição dos tipos dos dados da estrutura
 - **Lista das operações:** nome das operações que manipulam a estrutura
 - **Especificação das operações:**
 - **Entradas:** informação necessária para executar a operação
 - **Pré-condição:** verificada antes de executar a operação
 - **Processo:** tarefas que devem ser realizadas para realizar a operação
 - **Saída:** valor resultante do processamento
 - Retornado explicitamente pela operação
 - **Pós-condição:** indica alterações na estrutura após a operação
 - Retorno implícito (alteração em uma variável passada por referência)

Estrutura Geral da Especificação

TAD **nome_TAD**:

Dados: descrição dos campos da estrutura de dados

Lista de operações: operação1, operação2, ..., operação *N*

Operações:

Operação1:

Entrada:

Pré-condição:

Processo:

Saída:

Pós-condição:

Operação2:

Entrada:

Pré-condição:

Processo:

Saída:

Pós-condição:

...

Operação *N* :

Entrada:

Pré-condição:

Processo:

Saída:

Pós-condição:

Exemplo de Especificação

- Especificar o TAD **números racionais**:
 - Números expressos como quocientes de 2 inteiros (numerador e denominador)
Ex: $\frac{1}{2}$, $\frac{3}{4}$
 - Considerar as seguintes operações:
 - **generate**: cria uma instância de número racional sem valor
 - **set_value**: atribui valores para o numerador e o denominador
 - **get_value**: retorna os valores do numerador e do denominador de um número racional
 - **sum**: soma 2 números racionais
 - **delete**: liberar a memória usada por um número racional

Exemplo (TAD Racional)

- **TAD racional:**
 - **Dados:** 2 números inteiros, representando o numerador e o denominador do número racional
 - **Lista de operações:** *generate*, *set_value*, *get_value*, *sum* e *delete*

Exemplo (TAD Racional)

- Operação ***generate:***
 - **Entrada:** nenhuma
 - **Pré-condição:** nenhuma
 - **Processo:** criar um número racional
 - **Saída:** endereço do número racional criado
 - **Pós-condição:** nenhuma

Exemplo (TAD Racional)

- Operação ***set_value***:
 - **Entrada**: endereço de um N° racional (R) e 2 inteiros ($I1$ e $I2$)
 - **Pré-condição**: número racional ser válido e $I2$ ser diferente de zero
 - **Processo**: atribuir ao numerador de R o valor de $I1$ e ao denominador de R o valor de $I2$
 - **Saída**: 1 (sucesso) ou 0 (falha)
 - **Pós-condição**: racional R com valores alterados

Exemplo (TAD Racional)

- Operação ***get_value***:
 - **Entrada**: endereço de um N° racional (R) e endereço de 2 inteiros ($I1$ e $I2$)
 - **Pré-condição**: número racional ser válido
 - **Processo**: atribuir a $I1$ o valor do numerador de R e a $I2$ o valor do denominador de R
 - **Saída**: 1 (sucesso) ou 0 (falha)
 - **Pós-condição**: os inteiros com os valores do numerador e denominador de R

Exemplo (TAD Racional)

- Operação **sum**:
 - **Entrada**: endereço de dois racionais ($R1$ e $R2$)
 - **Pré-condição**: números racionais serem válidos
 - **Processo**: criar um novo racional $R3$ e atribuir a ele o resultado da soma de $R1$ e $R2$:
 - $\text{num de } R3 = (\text{num de } R1 \times \text{den de } R2) + (\text{num de } R2 \times \text{den de } R1)$
 - $\text{den de } R3 = (\text{den de } R1 \times \text{den de } R2)$
 - **Saída**: endereço do número racional $R3$ ou NULL se operação falhar
 - **Pós-condição**: nenhuma

Exemplo (TAD Racional)

- Operação **sum** (2ª opção):
 - **Entrada:** endereço dos 2 N° racionais operandos ($R1$ e $R2$) e do N° racional resultante ($R3$)
 - **Pré-condição:** números racionais serem válidos
 - **Processo:** atribuir a $R3$ a soma de $R1$ e $R2$:
 - $\text{num de } R3 = (\text{num de } R1 \times \text{den de } R2) + (\text{num de } R2 \times \text{den de } R1)$
 - $\text{den de } R3 = (\text{den de } R1 \times \text{den de } R2)$
 - **Saída:** 1 (sucesso) ou 0 (falha)
 - **Pós-condição:** número racional $R3$ contendo o resultado da soma de $R1$ e $R2$

Exemplo (TAD Racional)

- Operação ***delete***:
 - **Entrada**: endereço do end. de um N^o racional
 - **Pré-condição**: nenhuma
 - **Processo**: liberar a memória usada pelo número racional e limpar o seu endereço
 - **Saída**: nenhuma
 - **Pós-condição**: número racional liberado

Implementação Racional: versão 1 (não TAD)

```
struct rac{           //Versão 1: em arquivo único e sem testes
    int num;
    int den;
};
typedef struct rac racional;

racional *generate();
int set_value(racional *R, int N, int D);
racional *sum(racional *R1, racional *R2);

int main() {
    int Num, Den;
    racional *R1, *R2, *R3;

    R1=generate();
    R2=generate();

    printf("\nEntre o numerador do 1o Racional: ");    scanf("%d",&Num);
    printf("\nEntre o denominador do 1o Racional: ");    scanf("%d",&Den);
    set_value(R1, Num, Den);
    printf("\nEntre o numerador do 2o Racional: ");    scanf("%d",&Num);
    printf("\nEntre o denominador do 2o Racional: ");    scanf("%d",&Den);
    set_value(R2, Num, Den);

    R3=sum(R1,R2);
    printf("O valor do racional resultante da soma eh %d/%d",R3->num,R3->den);

    return 0;
} ...
```

Implementação Racional: versão 1

```
...  
} // fim da main  
  
racional *generate() {  
    racional *p;  
    p = (racional *) malloc(sizeof(racional));  
    return p;  
}  
  
int set_value(racional *R, int N, int D) {  
    (*R).num=N;  
    (*R).den=D;  
    return(1);  
}  
  
racional *sum(racional *R1,racional *R2){  
    racional *R;  
    R = generate();  
    R->num = (R1->num*R2->den) + (R2->num*R1->den);  
    R->den = (R1->den * R2->den);  
    return(R);  
}
```


Implementação Racional: versão 2 (não TAD)

struct rac{ //Versão 2: em arquivo único e com testes de validação

int num;

int den;

};

typedef struct rac racional;

*racional *generate();*

*int set_value(racional *R, int N, int D);*

*racional *sum(racional *R1, racional *R2);*

int main()

{ int Num, Den;

*racional *R1, *R2, *R3;*

R1=generate();

R2=generate();

if (R1 == NULL || R2 == NULL) {

printf ("Nao foi possivel criar os dois numeros racionais.\n");

return -1;

}

printf("\nEntre o numerador do 1o Racional: "); scanf("%d",&Num);

printf("\nEntre o denominador do 1o Racional: "); scanf("%d",&Den);

if (set_value(R1, Num, Den) == 0){

printf ("Nao foi possivel preencher o primeiro numero racional.\n");

return -1;

}

Implementação Racional: versão 2

... // Continua main

```
printf("\nEntre o numerador do 2o Racional: ");
scanf("%d",&Num);
printf("\nEntre o denominador do 2o Racional: ");
scanf("%d",&Den);
if (set_value(R2, Num, Den) == 0){
    printf ("Nao foi possivel preencher o primeiro numero racional.\n");
    return -1;
}

R3=sum(R1,R2);
if (R3==NULL) {
    printf ("Nao foi possivel somar os dois numeros racionais.\n");
    return (-1);
}

printf("O valor do racional resultante da soma eh %d/%d",R3->num,R3->den);

return 0;
} // fim main
...
```

Implementação Racional: versão 2

```
... // fim main
```

```
racional *generate() {  
    racional *p;  
    p = (racional *) malloc(sizeof(racional));  
    return p;  
}
```

```
int set_value(racional *R, int N, int D) {  
    if(R==NULL||D==0) return (0);  
    R->num=N;  
    R->den=D;  
    return(1);  
}
```

```
racional *sum(racional *R1, racional *R2){  
    racional *R;  
  
    R = generate();  
    if (R != NULL) {  
        R->num = (R1->num*R2->den) + (R2->num*R1->den);  
        R->den = (R1->den * R2->den);  
    }  
    return(R);  
}
```

Implementação de umTAD

- Define **como** o TAD deve realizar suas operações
 - Implementação deve seguir as especificações (garantia da interface)
 - Converte um TAD em um **tipo de dado concreto**
- TAD e programa usuário (aplicação) devem ser implementados em módulos separados
 - Garante a independência e reutilização
 - Cada módulo é implementado em um arquivo c
 - Interface implementada através de um arquivo cabeçalho (.h)
- Códigos são **compilados separadamente**
 - Mudanças exigem recompilação **apenas** do módulo alterado
 - Cada arquivo c gera um código objeto próprio
- Códigos são ligados na linkedição
 - Junta todos os arquivos objeto em um único executável

Implementação em C

- Um arquivo com a implementação do TAD
 - Contém a estrutura e as funções (operações)
 - Ex:** racional.c
- Pelo menos um arquivo de aplicação
 - Programa cliente/usuário que irá utilizar o TAD
 - Ex:** principal.c
- Um arquivo cabeçalho (extensão .h) com as declarações do tipo e os protótipos das funções
 - Permite o reconhecimento por outros módulos
 - Todos os módulos devem ter a diretiva ***#include "arquivo.h"***
 - Ex:** racional.h

Uso da diretiva *#include*

- **#include <arq.h>:**

- O arquivo *arq.h* está no diretório padrão de include.
- Usada para arquivos de cabeçalho (.h) das bibliotecas padrões do C.

- **#include "arq.h":**

- O arquivo *arq.h* está no diretório de trabalho do programa
- Muito usada para informar o caminho completo do arquivo

Ex: *#include "c:\Meus_Progs\Bib\arq.h"*

Geração do Executável

- **Windows:**

- *Frameworks* de desenvolvimento associam os arquivos de uma aplicação através de um **projeto**.

Ex: CodeBlocks e DevC

- **Linux:**

- Pode-se usar o projeto (ex: CodeBlocks)

OU

- Compilar os arquivos isoladamente e juntá-los na linkedição

Ex: compilação e linkedição no gcc

gcc -c racional.c

gcc -c usuario.c

gcc -o prog usuario.o racional.o

Geração do Executável

- **Windows:**

- *Frameworks* de desenvolvimento associam os arquivos de uma aplicação através de um **projeto**.

Ex: CodeBlocks e DevC

- **Linux:**

- Pode-se usar o projeto (ex: CodeBlocks)

OU

- Compilar os arquivos isoladamente e juntá-los na linkedição

Ex: compilação e linkedição no gcc

`gcc -c racional.c`

gera arquivo objeto racional.o

`gcc -c usuario.c`

`gcc -o prog usuario.o racional.o`

Geração do Executável

- **Windows:**

- *Frameworks* de desenvolvimento associam os arquivos de uma aplicação através de um **projeto**.

Ex: CodeBlocks e DevC

- **Linux:**

- Pode-se usar o projeto (ex: CodeBlocks)

OU

- Compilar os arquivos isoladamente e juntá-los na linkedição

Ex: compilação e linkedição no gcc

gcc -c racional.c

gcc -c usuario.c

gera arquivo objeto usuario.o

gcc -o prog usuario.o racional.o

Geração do Executável

- **Windows:**

- *Frameworks* de desenvolvimento associam os arquivos de uma aplicação através de um **projeto**.

Ex: CodeBlocks e DevC

- **Linux:**

- Pode-se usar o projeto (ex: CodeBlocks)

OU

- Compilar os arquivos isoladamente e juntá-los na linkedição

Ex: compilação e linkedição no gcc

gcc -c racional.c

gcc -c usuario.c

`gcc -o prog usuario.o racional.o`

junta os arqs objeto formando o executável prog

Implementação Racional: versão 3 (TAD simplificada)

```
#include <stdio.h>
#include <stdlib.h>
#include "TAD_Racional.h"

struct rac{
    int num;
    int den;
};

racional *generate() {
    racional *p;
    p = (racional *) malloc(sizeof(racional));
    return p; }
```

Arquivo TAD_RACIONAL.C

```
int set_value(racional *R, int N, int D) {
    if(R==NULL || D==0) return (0);
    R->num=N;
    R->den=D;
    return(1); }

racional *sum(racional *R1,racional *R2){
    racional *R;
    R = generate();
    if (R != NULL) {
        R->num = (R1->num*R2->den) + (R2->num*R1->den);
        R->den = (R1->den * R2->den);
    }
    return(R); }

int print_value(racional *R) {
    printf("%d/%d",R->num,R->den); }
```

Implementação Racional: versão 3 (TAD simplificada)

Arquivo TAD_RACIONAL.H

```
typedef struct rac racional;  
  
racional *generate();  
int set_value(racional *R, int N, int D);  
racional *sum(racional *R1, racional *R2);  
int print_value(racional* R);
```

Implementação Racional: versão 3 (TAD simplificada)

```
#include <stdio.h>
#include <stdlib.h>
#include "TAD_Racional.h"
```

Arquivo USER.C

```
int main()
{ int Num, Den;
  racional *R1, *R2, *R3;

  R1=generate();
  R2=generate();
  if (R1 == NULL || R2 == NULL) {
    printf ("Nao foi possivel criar os dois numeros racionais.\n");
    return -1;
  }

  printf("\nEntre o numerador do 1o Racional eh: "); scanf("%d",&Num);
  printf("\nEntre o denominador do 1o Racional eh: "); scanf("%d",&Den);
  if (set_value(R1, Num, Den) == 0){
    printf ("Nao foi possivel preencher o primeiro numero racional.\n");    return -1;
  }

  printf("\nEntre o numerador do 2o Racional: "); scanf("%d",&Num);
  printf("\nEntre o denominador do 2o Racional: "); scanf("%d",&Den);
  if (set_value(R2, Num, Den) == 0){
    printf ("Nao foi possivel preencher o primeiro numero racional.\n");    return -1;
  }

  R3=sum(R1,R2);
  if (R3==NULL) {
    printf ("Nao foi possivel somar os dois numeros racionais.\n");    return (-1);
  }

  printf("O valor do racional resultante da soma eh %d/%d",R3->num,R3->den);

  return 0;
}
```

Implementação Racional: versão 3 (TAD simplificada)

```
#include <stdio.h>
#include <stdlib.h>
#include "TAD_Racional.h"
```

Arquivo USER.C

```
int main()
{ int Num, Den;
  racional *R1, *R2, *R3;

  R1=generate();
  R2=generate();
  if (R1 == NULL || R2 == NULL) {
    printf ("Nao foi possivel criar os dois numeros racionais.\n");
    return -1;
  }

  printf("\nEntre o numerador do 1o Racional eh: "); scanf("%d",&Num);
  printf("\nEntre o denominador do 1o Racional eh: "); scanf("%d",&Den);
  if (set_value(R1, Num, Den) == 0){
    printf ("Nao foi possivel preencher o primeiro numero racional.\n");    return -1;
  }

  printf("\nEntre o numerador do 2o Racional: "); scanf("%d",&Num);
  printf("\nEntre o denominador do 2o Racional: "); scanf("%d",&Den);
  if (set_value(R2, Num, Den) == 0){
    printf ("Nao foi possivel preencher o primeiro numero racional.\n");    return -1;
  }

  R3=sum(R1,R2);
  if (R3==NULL) {
    printf ("Nao foi possivel somar os dois numeros racionais.\n");    return (-1);
  }

printf("O valor do racional resultante da soma eh %d/%d",R3->num,R3->den);

  return 0;
}
```

Implementação Racional: versão 3 (TAD simplificada)

```
#include <stdio.h>
#include <stdlib.h>
#include "TAD_Racional.h"
```

Arquivo USER.C

```
int main()
{ int Num, Den;
  racional *R1, *R2, *R3;

  R1=generate();
  R2=generate();
  if (R1 == NULL || R2 == NULL) {
    printf ("Nao foi possivel criar os dois numeros racionais.\n");
    return -1;
  }

  printf("\nEntre o numerador do 1o Racional eh: "); scanf("%d",&Num);
  printf("\nEntre o denominador do 1o Racional eh: "); scanf("%d",&Den);
  if (set_value(R1, Num, Den) == 0){
    printf ("Nao foi possivel preencher o primeiro numero racional.\n");    return -1;
  }

  printf("\nEntre o numerador do 2o Racional: "); scanf("%d",&Num);
  printf("\nEntre o denominador do 2o Racional: "); scanf("%d",&Den);
  if (set_value(R2, Num, Den) == 0){
    printf ("Nao foi possivel preencher o primeiro numero racional.\n");    return -1;
  }

  R3=sum(R1,R2);
  if (R3==NULL) {
    printf ("Nao foi possivel somar os dois numeros racionais.\n");    return (-1);
  }

  printf("\n O valor do racional resultante da soma eh ");
  print_value(R3);

  return 0;
}
```

Exemplo (TAD Racional) - completa

- **Especificação dos dados do TAD racional:**
 - **Dados:** 2 números inteiros, representando o numerador e o denominador do número racional

Exemplo (TAD Racional)

- **Especificação dos dados do TAD racional:**
 - **Dados:** 2 números inteiros, representando o numerador e o denominador do número racional
- **Implementação da estrutura:**

```
struct racional {  
    int num, den;  
};
```

```
typedef struct racional Racional;
```

Exemplo (TAD Racional)

- Especificação dos dados do TAD **racional**:
 - **Dados**: 2 números inteiros, representando o numerador e o denominador do número racional
- Implementação da estrutura:

```
struct racional {  
    int num, den;  
};
```

```
typedef struct racional Racional;
```

racional.c
(definição)

racional.h
(referência)

Exemplo (TAD Racional)

- **Especificação da operação *generate*:**
 - **Entrada:** nenhuma
 - **Pré-condição:** nenhuma
 - **Processo:** criar um número racional
 - **Saída:** endereço do número racional criado
 - **Pós-condição:** nenhuma

Exemplo (TAD Racional)

- Implementação da operação ***generate***:

```
Racional * generate() {  
    Racional * p;  
    p = (Racional *) malloc(sizeof(Racional));  
    return p;  
}
```

Exemplo (TAD Racional)

- **Especificação da operação *set_value*:**
 - **Entrada:** endereço de um N^o racional (R) e 2 inteiros ($I1$ e $I2$)
 - **Pré-condição:** número racional ser válido e $I2$ ser diferente de zero
 - **Processo:** atribuir ao numerador de R o valor de $I1$ e ao denominador de R o valor de $I2$
 - **Saída:** 1 (sucesso) ou 0 (falha)
 - **Pós-condição:** racional R com valores alterados

Exemplo (TAD Racional)

- Implementação da operação ***set_value***:

```
int set_value(Racional *p, int N, int D) {  
    if (p == NULL || D == 0)  
        return 0; // Falha  
  
    p->num = N;  
    p->den = D;  
    return 1; // Sucesso  
}
```

Exemplo (TAD Racional)

- **Especificação da operação *get_value*:**
 - **Entrada:** endereço de um N^o racional (R) e endereço de 2 inteiros ($I1$ e $I2$)
 - **Pré-condição:** número racional ser válido
 - **Processo:** atribuir a $I1$ o valor do numerador de R e a $I2$ o valor do denominador de R
 - **Saída:** 1 (sucesso) ou 0 (falha)
 - **Pós-condição:** os inteiros com os valores do numerador e denominador de R

Exemplo (TAD Racional)

- Implementação da operação ***get_value***:

```
int get_value(Racional *p, int *N, int *D) {  
    if (p == NULL)  
        return 0; // Falha  
  
    *N = p->num;  
    *D = p->den;  
    return 1; // Sucesso  
}
```


Exemplo (TAD Racional)

- **Especificação da operação *sum*:**
 - **Entrada:** endereço de dois racionais ($R1$ e $R2$)
 - **Pré-condição:** números racionais serem válidos
 - **Processo:** criar um novo racional $R3$ e atribuir a ele o resultado da soma de $R1$ e $R2$:
 - $\text{num de } R3 = (\text{num de } R1 \times \text{den de } R2) + (\text{num de } R2 \times \text{den de } R1)$
 - $\text{den de } R3 = (\text{den de } R1 \times \text{den de } R2)$
 - **Saída:** endereço do número racional $R3$ ou NULL se operação falhar
 - **Pós-condição:** nenhuma

Exemplo (TAD Racional)

- Implementação da operação **sum**:

Racional * *sum*(*Racional* **R1*, *Racional* **R2*) {

Racional **R3*;

R3 = generate(); // Cria um racional e atribui seu end. a R3

if (R3 != NULL) {

*R3->num = (R1->num*R2->den) + (R2->num*R1->den);*

*R3->den = (R1->den * R2->den);*

}

return R3; // Sucesso ou Falha

}

Exemplo (TAD Racional)

- **Especificação da operação *sum* (2ª opção):**
 - **Entrada:** endereço dos 2 N° racionais operandos ($R1$ e $R2$) e do N° racional resultante ($R3$)
 - **Pré-condição:** números racionais serem válidos
 - **Processo:** atribuir a $R3$ a soma de $R1$ e $R2$:
 - $\text{num de } R3 = (\text{num de } R1 \times \text{den de } R2) + (\text{num de } R2 \times \text{den de } R1)$
 - $\text{den de } R3 = (\text{den de } R1 \times \text{den de } R2)$
 - **Saída:** 1 (sucesso) ou 0 (falha)
 - **Pós-condição:** número racional $R3$ contendo o resultado da soma de $R1$ e $R2$

Exemplo (TAD Racional)

- Implementação da operação **sum** (2ª opção):

```
int sum(Racional *R1, Racional *R2, Racional *R3) {  
    R3 = generate(); // Cria um racional e atribui seu end. a R3  
    if (R3 == NULL)  
        return 0; // Falha  
  
    R3->num = (R1->num * R2->den) + (R2->num * R1->den);  
    R3->den = (R1->den * R2->den);  
  
    return 1; // Sucesso  
}
```

Exemplo (TAD Racional)

- **Especificação da operação *delete*:**
 - **Entrada:** endereço do end. de um N^o racional
 - **Pré-condição:** nenhuma
 - **Processo:** liberar a memória usada pelo número racional e limpar o seu endereço
 - **Saída:** nenhuma
 - **Pós-condição:** número racional liberado

Exemplo (TAD Racional)

- Implementação da operação **delete**:

```
void delete(Racional **p) {  
    free(*p);    // Libera a memória  
    *p = NULL; // Limpa o ponteiro para racional  
}
```

Implementação (TAD Racional)

- **Disposição das operações nos arquivos:**
 - Funções implementadas no **racional.c**
 - Protótipo das funções declarados no **racional.h**

*Racional * **generate**();*

*int **set_value**(Racional *p, int N, int D);*

*int **get_value**(Racional *p, int *N, int *D);*

*Racional * **sum**(Racional *R1, Racional *R2);*

*void **delete**(Racional ** p);*

Implementação TAD Racional

- **Racional.h**

```
typedef struct racional Racional;
```

```
Racional * generate();
```

```
int set_value(Racional *p, int N, int D);
```

```
int get_value(Racional *p, int *N, int *D);
```

```
Racional * sum(Racional *R1, Racional *R2);
```

```
void delete(Racional ** p);
```


Implementação TAD Racional

- **Racional.c**

```
#include<stdio.h> // para usar o NULL
```

```
#include<stdlib.h> // para usar as funções malloc() e free()
```

```
#include "Racional.h"
```

```
struct racional {  
    int num, den;  
};
```

```
Racional *generate() { ... Código da função ... }
```

```
int set_value(Racional *p, int N, int D) {... Código da função ... }
```

```
int get_value(Racional *p, int *N, int *D) {... Código da função ... }
```

```
Racional *sum(Racional *R1, Racional *R2) {...Código da função...}
```

```
void delete(Racional ** p) {... Código da função ... }
```

Implementação TAD Racional

- **User.c**

```
#include<stdio.h>
#include "Racional.h"
int main() {
    Racional *N1, *N2, *N3;
    int nu, de;
    N1 = generate(); N2 = generate(); // Cria 2 nros racionais
    if (N1 == NULL || N2 == NULL) {
        printf ("Nao foi possivel criar os nros racionais.\n");
        return -1;
    }
    printf("Digite o numerador e denominador do 1o racional:\n");
    scanf("%d", &nu); scanf("%d", &de);
    if (set_value(N1, nu, de) == 0) { // Atribui o numerador e o denominador a N1
        printf ("\nFalha ao preencher o 1o racional.\n");
        return -1;
    }
    ...
}
```

Implementação TAD Racional

- **User.c (continuação)**

...

```
printf("\nDigite o numerador e denominador do 2o racional:\n");
scanf("%d", &nu); scanf("%d", &de);
if (set_value(N2, nu, de) == 0) { // Atribui o numerador e o denominador a N2
    printf ("\nFalha ao preencher o 2o racional.\n");
    return -1;
}
N3 = sum(N1,N2); // Soma 2 nros racionais
if (N3 == NULL) {
    printf ("\nFalha ao somar os 2 nros racionais.\n");
    return -1;
}
if (get_value(N3, &nu, &de) == 0) { // Obtem o numerador e o denominador de N3
    printf ("\nFalha ao recuperar o numerador e denominador resultante.\n");
    return -1;
}
printf("\nO resultado da soma eh: %d / %d.\n", nu, de);
delete(&N1); delete(&N2); delete(&N3); // Libera a memoria alocada para os nros
return 0;
}
```

Exercícios

1. *Especificar e implementar o TAD Ponto, o qual representa um ponto no espaço bidimensional (coordenadas x e y no espaço \mathbb{R}^2) e com as seguintes operações:*

- ***Cria_pto:*** *cria um ponto a partir de suas coordenadas x e y*
- ***Libera_pto:*** *operação que elimina um ponto criado*
- ***Distancia_pto:*** *calcula a distância entre dois pontos*

O programa aplicativo deve ler as coordenadas de 2 pontos, digitadas pelo usuário e imprimir na tela a distância entre esses pontos. Nesse processo, o programa deve criar os 2 pontos, calcular a distância entre esses pontos e, após apresentar o resultado, liberar os dois pontos.

Exercícios

2. *Especificar e implementar o TAD Círculo, o qual representa um círculo no espaço bidimensional (representados por 2 pontos $P1$ e $P2$ que determinam o seu raio) e com as seguintes operações:*
 - ***Cria_circ:*** *cria um círculo a partir dos pontos que determinam as extremidades do seu raio*
 - ***Libera_circ:*** *operação que elimina um círculo criado*
 - ***Area_circ:*** *calcula a área de um círculo*

O programa aplicativo deve ler as coordenadas dos pontos que determinam o raio do círculo, digitadas pelo usuário, e imprimir na tela a área do círculo resultante.

Referências

- *Backes, André, Linguagem C Descomplicada, portal de vídeo-aulas, <https://programacaodescomplicada.wordpress.com/>, acessado em 09/03/2016.*
- *Celes, W., Cerqueira, R. e Rangel, J. L. Introdução a estruturas de dados. Ed. Campus Elsevier, 2004.*