

Tipo de dado abstrato

“ [...] A abstração é nossa mais importante ferramenta mental para lidar com a complexidade. Portanto, um problema complexo não poderia ser visto imediatamente em termos de instruções de computador [...] mas, antes, em termos de entidades naturais ao próprio problema, abstraído de maneira adequada. ”

NIKLAUS WIRTH (1989)

O significado da palavra *abstrair*, segundo o Dicionário Aurélio, é considerar isoladamente um ou mais elementos de um todo. Assim, em programação, abstrair é imaginar um problema maior dividido em problemas menores, para resolvê-los isoladamente e, posteriormente, uni-los, produzindo a solução do problema.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- definir e usar tipos de dados primitivos e não primitivos;
- utilizar as estruturas de dados como tipos de dados abstratos;
- saber que trabalhar com abstração dos tipos de dados auxilia na compreensão do paradigma de programação orientada a objetos, sem entrar nos conceitos desse paradigma.



Para começar

O computador é considerado uma máquina abstrata, pois, independentemente de saber *como* ele executa as tarefas, interessa saber *o que* ele pode executar.



A abstração, em computação, é um conceito antigo, usado inicialmente para processos. SIMULA 67 foi a primeira linguagem com recursos para a abstração de dados. [Barbara Liskov e Stephen Zilles \(1974\)](#) definiram o conceito de *tipo de dado abstrato* como uma classe de objetos abstratos, afirmando: “... o que desejamos de uma abstração é um mecanismo que permita a expressão de detalhes relevantes e a supressão de detalhes irrelevantes”. Eles destacaram, ainda, que *abstração* é um termo usado em vários segmentos e por várias pessoas como a chave para a construção de um bom projeto.

Na engenharia de software, ao se adotar a divisão do problema em módulos, se estabelecem níveis de abstração. No mais alto nível de abstração, os termos usados para a declaração são os do próprio ambiente do problema. O mais baixo nível de abstração está mais próximo dos detalhes de implementação.

A divisão de um problema maior em módulos não deve ser aleatória, e a arquitetura do software gerada pode variar de acordo com a diversidade de tendências, estilos e experiência de quem o projeta. Entretanto, há técnicas que auxiliam na construção dessa divisão, de forma que o resultado final

contemple requisitos de um bom projeto, tais como a busca de módulos com alta coesão e baixo acoplamento.

A coesão e o acoplamento entre os módulos divididos afetam o nível de abstração de um projeto. A abstração é a arte de expor seletivamente a funcionalidade de interfaces coesivas e, ao mesmo tempo, esconder as implementações acopladas.

Dominar a habilidade de pensar de forma abstrata sobre os componentes de um programa de computador é um problema para o estudante de séries iniciais de cursos de computação. Enquanto não desenvolve essa habilidade, o estudante tende a ver um programa como uma coleção não estruturada de declarações e expressões. Com isso, a complexidade de um programa aumenta com o tamanho do problema a ser desenvolvido. Ao desenvolvê-la, ele tende a ver o programa como uma coleção de funções e classes. Torna-se, então, mais fácil fazer a manutenção desse programa e reutilizá-lo.

Assim, estruturas de dados, como listas lineares, pilhas e filas, devido ao conjunto de operações que definem claramente a ação sobre os dados que elas possuem, representam elementos adequados para a introdução do conceito de abstração.

Parece complicado, mas não é. Vamos lá!



Conhecendo a teoria para programar

De modo geral, todos os programas (*softwares*) podem ser denominados “processadores de informação”. São construídos para processar as informações que recebem através das entradas, aplicar-lhes transformações e produzir as saídas.

A elaboração de todo programa, respeitadas as devidas proporções, deveria seguir obrigatoriamente três fases genéricas:

- definição;
- desenvolvimento; e
- manutenção.

A *fase de definição* do *software* é aquela em que se deve determinar claramente o que se pretende atender com esse programa. Para tanto, devem-se identificar os principais requisitos do *software*, tais como:

- funcionalidade;
- desempenho desejado;
- informações a serem processadas;

- critérios de validação;
- requisitos de interface; e
- restrições do projeto.

A *fase de desenvolvimento* é aquela na qual se deve definir como será a arquitetura do *software*, como serão implementados os detalhes de procedimentos, como os dados serão traduzidos para uma linguagem de programação e, por fim, como serão os testes.

A *fase de manutenção* é a caracterizada pelas mudanças. É nessa fase que podem ser feitas mudanças para solucionar erros encontrados pelos testadores, além de adaptações e/ou melhorias.

Apesar da importância dessas três fases, entre elas destaca-se a fase de desenvolvimento, que compreende o estudo da arquitetura do *software*. A arquitetura refere-se a duas importantes características de um programa:

- a estrutura hierárquica dos componentes do *software* (módulos);
- a estrutura dos dados.

Modularidade

É um conceito bastante antigo que consiste em dividir um *software* em componentes individuais, rotulados e endereçáveis, chamados *módulos*, que, uma vez integrados, atendem aos requisitos do problema (“dividir para conquistar”). A [Figura 1.1](#) representa essa ideia.

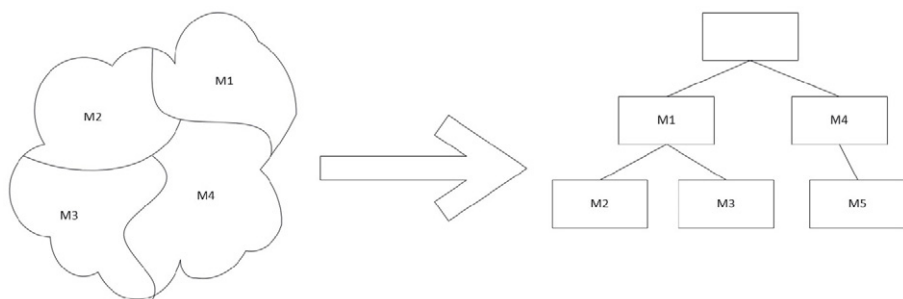


FIGURA 1.1 – Divisão do problema em módulos.

A solução de qualquer problema pode ser dada de várias formas. Essas formas são caracterizadas como diferentes níveis de abstração.

A técnica que divide o problema em módulos cria uma estrutura hierárquica de componentes de *software*. Esses componentes (módulos) podem

ser desenvolvidos utilizando-se técnicas diferentes, isto é, cada módulo, que pode ser ou não desenvolvido por meio de um único subprograma, pode ter desenvolvimento linear ou recursivo.

A definição das estruturas de dados para um software é muito importante, pois influencia todo o processo de desenvolvimento do programa.

A organização e a complexidade de uma estrutura de dados são limitadas apenas pela criatividade de quem as projeta. Entretanto, há um conjunto limitado de estruturas clássicas de dados a partir das quais é possível criar outras mais sofisticadas.

As *estruturas de dados* diferem umas das outras pela forma como as informações estão dispostas e pela forma como essas informações são manipuladas. Portanto, determinada estrutura é caracterizada pela *forma de armazenamento* dos dados e pelas *operações definidas* sobre ela. Assim, não se pode desvincular uma estrutura de dados de seus aspectos algorítmicos. A escolha de uma estrutura depende diretamente do conhecimento algorítmico das operações que a manipulam.

Algoritmos

Algoritmo é um conjunto finito de regras de definição, atribuição e controle cuja sequência lógica deve atender aos requisitos de um problema. Essas regras são executadas por uma ferramenta que as entenda, no caso o computador.

Quase sempre existe mais de uma forma de resolver um mesmo problema. Os elementos básicos que tornam um algoritmo diferente de outro são, basicamente:

- Componentes que formam o corpo do algoritmo:
 - preferência por determinadas instruções (estilo de programação);
 - divisão do programa em subprogramas;
 - recursividade ou não.
- escolha e definição das estruturas de dados.

Na solução de um problema, é necessário escolher uma abstração da realidade, isto é, definir um conjunto de dados que represente a situação. Essa escolha deve atender ao problema e estar de acordo com os recursos que a linguagem e a máquina que serão utilizadas podem oferecer.

A escolha da representação dos dados, na maioria das vezes, é guiada pelas operações que devem ser executadas sobre os dados.

Essas representações de dados são classificadas de acordo com características importantes, que dizem respeito ao conjunto de valores que uma

informação pode assumir e às operações que podem ser executadas sobre elas. Essa classificação caracteriza os *tipos de dados*.

Um tipo de dado determina o conjunto de valores a que uma constante pertence, ou que pode ser assumido por uma variável ou uma expressão, ou que pode ser gerado por uma operação ou uma função.

Os tipos de dados podem ser classificados em *primitivos* e *não primitivos* (criados a partir dos primitivos).

Tipos de dados

Cada nome (identificador) em um programa, em qualquer linguagem, tem um tipo associado a ele. Esse tipo determina que operações podem ser aplicadas ao nome e como elas devem ser interpretadas.

Tipos de dados primitivos

Também chamados de *estruturas de dados primitivas*, tipos de dados primitivos são aqueles que estão disponíveis na maioria dos computadores, como:

- Inteiros:
 - conjunto de valores: negativos, zero, positivos;
 - operações: +, -, *, /;
 - comparações: =, >, <, ≥, ≤, ≠.
- Reais:
 - conjunto de valores: parte inteira e fracionária;
 - operações: +, -, *, /;
 - comparações =, >, <, ≥, ≤, ≠.
- Lógicos:
 - conjunto de valores: verdadeiro e falso;
 - operações: e (\wedge), ou (\vee), não (\sim);
 - comparações: = e ≠.
- Caracteres:
 - conjunto de valores: letras, números, símbolos especiais;
 - operações: não há;
 - comparações: =, >, <, ≥, ≤, ≠.

Tipos de dados não primitivos

Tipos de dados não primitivos são aqueles construídos a partir dos tipos primitivos.

Os principais tipos são:

- arranjos (*array*) - vetores e matrizes;
- cadeia de caracteres (*strings*);
- tabelas;
- listas ligadas;
- listas lineares; e
- listas não lineares.

Tipo de dado abstrato (TDA)

É um tipo de dado que agrega os seguintes componentes:

- definição abstrata dos dados;
- ações abstratas.

Para se considerar abstrato o tipo de dado, devemos aplicar as operações definidas para ele independentemente de saber como foram implementadas ou como os dados foram armazenados, isto é, se eles foram armazenados usando alocação estática ou dinâmica de memória. O importante é saber *o que* as operações fazem, quais seus parâmetros de entrada e o que produzem como resultado, sem necessariamente saber *como* elas foram construídas.

Com base nesse conceito, vamos trabalhar alguns exemplos de arranjos (*arrays*) e cadeias de caracteres (*strings*) como *tipos de dados abstratos*. É uma forma um pouco diferente de usar esses tipos de dados, se comparada com a que se aprende num primeiro contato com linguagens de programação.

Arranjo (*array*) como tipo de dado abstrato

As operações definidas para os arranjos, em se tratando de vetores e matrizes, são as já conhecidas para esse tipo na representação matemática.

A seguir damos um exemplo de como podemos trabalhar um vetor como tipo de dado abstrato. Vale lembrar que não pretendemos, aqui, entrar em detalhes, nem usar recursos definidos para a programação orientada a objetos. Vamos apenas direcionar o pensamento para construções abstratas de dados. Posteriormente, o arranjo será utilizado em outras representações internas.

Definição do tipo como TDA – Vetor

- *Nome do tipo:* **Vetor** (pode ser de inteiros ou reais);
- *Componentes:* número de elementos e espaço de armazenamento.

Especificação das operações (alguns exemplos)

Considere:

- x, y do tipo **Vetor**;
 - k, n do tipo **Inteiro**;
 - a do tipo **Real**.
1. produto_por_escalar (x, k) y
 2. soma_dos_elementos (x) a
 3. num_elementos(x) n
 4. leitura () x
 5. imprime (x)

Considerando-se a existência das operações definidas, vamos construir programas em C que implementem os seguintes problemas:

- **Problema 1-** Ler um conjunto de notas de alunos, calcular a média da turma, imprimir as notas lidas e a média da turma.
- **Problema 2-** Ler dois conjuntos de pontuações relativas aos resultados obtidos por dois participantes do Rally Paris-Dakar. Calcular a média de cada um e indicar qual deles teve a melhor média. Imprimir as pontuações lidas, as médias e quem obteve a melhor média.

Problema 1

Código 1.1

```
//definição do tipo
typedef float Vetor;
void main( )
{
    Vetor Notas;
    int NumElementos;
    Notas = leitura();
    NumElementos = num_elementos(Notas);
    float media = soma_dos_elementos (Notas)/NumElementos;
    imprime(Notas);
    printf("\nMedia: %f",media);
}
```


Problema 2

Código 1.2

```
//definição do tipo
typedef float Vetor;
void main()
{
    Vetor Participante1, Participante2;
    float Media_01, Media_02;
    int NumParticipantes1, NumParticipantes2;
    Participante1 = leitura();
    Participante2 = leitura();
    NumParticipantes1 = num_elementos(Participantes1);
    NumParticipantes2 = num_elementos(Participantes2);
    Media_01 = soma_dos_elementos(Participante1)/NumParticipante1;
    Media_02 = soma_dos_elementos(Participante2)/NumParticipante2;
    if( Media_01 > Media_02)
        printf("Media do primeiro e' Maior: %f",Media_01);
    else
    {
        if(Media_01 < Media_02)
            printf("Media do segundo e' Maior: %f", Media_02);
        else printf("\nMedias iguais : %f", Media_01);
    }
}
```

Não precisamos nos preocupar com o tamanho dos vetores nem com a forma como seus valores foram lidos, nem mesmo onde foram armazenados. Partimos do princípio de que isso já estava definido e apenas usamos as operações. Isso faz com que passemos a nos preocupar com a solução do problema, e não com o local onde vamos armazenar os dados (local de memória, dinâmica ou estática).

Se executarmos esses exemplos, certamente o compilador não reconhecerá o tipo de dado – *Vetor* –, nem as operações – *leitura*, *soma_dos_elementos*, *num_elementos* e *imprime* –, pois fomos nós que as criamos. Para isso, teríamos de criar, em algum momento, um arquivo especial que compusesse nossa biblioteca de operações, como as que o compilador possui.

Em C/C++, esse arquivo é chamado de *header*, e, em nosso exemplo, pode ser chamado de *Vetor.h*. Ele conterá a definição do tipo e as operações construídas.

Para compreender melhor o tipo de dado abstrato, vamos utilizar como exemplo a cadeia de caracteres.

Cadeia de caracteres como TDA

A estrutura cadeia de caracteres (*string*) é um arranjo de caracteres que possui tamanho variável. Cada linguagem de programação possui uma forma própria de manipular cadeias de caracteres.

A cadeia de caracteres, na forma como está implementada nas linguagens de programação, representa um tipo de dado abstrato. Vejamos:

- ela possui um campo de definição: *char nome[]*; e
- um conjunto de operações definidas sobre essa definição.

Observe alguns exemplos de operações na linguagem C/C++:

```
strcpy(cc1, cc2); // faz uma cópia de cc2 em cc1
strcmp(cc1, cc2); // compara as duas cadeias de caracteres
strlen(cc1); // determina o comprimento de cc1
strcat(cc1, cc2); // concatena as duas cadeias de caracteres
```

Elas podem ser usadas adicionando-se ao programa a biblioteca *string.h*.

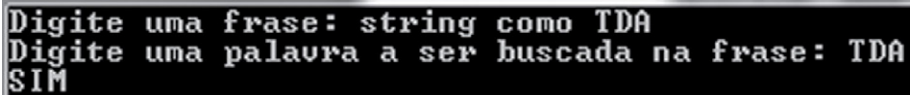
Essas e outras operações para manipulação de cadeias de caracteres existentes na biblioteca *string.h* podem ser encontradas em livros de programação em C/C++ ou no *help* do próprio compilador. Além disso, de acordo com o problema, outras operações podem ser criadas.

Vejamos o problema a seguir, que utiliza uma cadeia de caracteres: suponha que uma cadeia de caracteres *S* contenha uma frase qualquer. Vamos construir um programa que, utilizando as operações disponíveis na biblioteca *string.h*, verifique se a frase possui pelo menos uma ocorrência de um caractere *P*, também lido. Em caso afirmativo, imprimir SIM; caso contrário, imprimir NAO.

Código 1.3

```
#include <stdio.h> //biblioteca com as operações de entrada e saída
#include <string.h> //biblioteca com as operações de cadeia de caracteres
#define TamFrase 20 //tamanho máximo da frase a ser lida: 19 caracteres
#define TamPalavra 10 //tamanho máximo da palavra: 9 caracteres
void main()
{
    char S[TamFrase];
    char P[TamPalavra];
    printf("Digite uma frase: ");
    gets(S); //leitura da frase
    printf("Digite uma palavra a ser buscada na frase: ");
    scanf("%s", P);
    if( strstr(S,P))
        printf("SIM");
    else
        printf("NAO");
}
```

Resultado esperado (Figura 1.2):



```
Digite uma frase: string como TDA
Digite uma palavra a ser buscada na frase: TDA
SIM
```

FIGURA 1.2: Resultado da execução do código apresentado.

Atenção

Lembre-se de que, na declaração `char X[10]`, por exemplo, de uma variável do tipo cadeia de caracteres, no máximo 9 caracteres serão armazenados em X. A última posição será ocupada com o caractere de fim de cadeia de caracteres, representado por: `'\0'`.



Criando TDA específico

Em várias linguagens, é possível criar tipos de dados abstratos específicos para controlar e armazenar informações necessárias para seus desenvolvimentos. Os TDAs podem ser compostos por tipos primitivos e por outros TDAs, algumas vezes servindo de referência para o próprio TDA.

Imagine que um TDA seja uma grande região de memória subdividida em pedacinhos. Esses pedacinhos são os espaços suficientes para guardar o tipo primitivo que compõe esse TDA.

Fazendo uma analogia com o mundo real, um TDA seria um quarteirão de nossas cidades. Sabemos que os quarteirões são subdivididos em terrenos de diferentes tamanhos. Nesse caso, os terrenos representam as regiões de memória alocadas para um tipo primitivo dentro do TDA.

Em linguagem C, os TDAs são criados por meio da definição de *structs*, que pode ser composta por tipos primitivos e/ou por outras *structs*. Em geral, um TDA é definido como um conjunto de informações que devem seguir juntas para o bom funcionamento de um programa, como um programa que controlaria uma secretaria de faculdade, por exemplo. Nesse caso, as informações básicas, para eles, são as relacionadas aos alunos, como seu registro acadêmico, nome completo, e-mail, telefone e outros dados. Veja a seguir um exemplo, em linguagem C, que cria um TDA com essas informações:

Código 1.4

```
struct aluno
{
    char nome[50];
    int registro_academico;
    char email[200];
    char telefone[10];
    int ano_egresso;
};
```



Vamos programar

Vejam agora como seriam as representações dos tipos *Vetor* e *Cadeia de caracteres*, estudados na seção anterior, tanto em Java como em Python.

Java

Código 1.1

```
import java.util.*;
public class ProgramaTipoVetor
{
    public static void main(String[] args)
    {
        Vetor Notas;
        int NumElementos;
        float media;
        Notas = leitura();
        NumElementos = num_elementos(Notas);
        media = soma_dos_elementos(Notas)/NumElementos;
        imprime(Notas);
        System.out.print("\nMedia:"+media);
    }
}
```

Código 1.2

```
import java.util.*;
public class ProgramaTipoVetor
{
    public static void main(String[] args)
    {
        float[] Participante1 = new float[10];
        float[] Participante2 = new float[10];
        float Media_01, Media_02;
        int NumParticipantes1, NumParticipantes2;
        Participante1 = leitura();
        Participante2 = leitura();
        NumParticipantes1 = num_elementos(Participantes1);
        NumParticipantes2 = num_elementos(Participantes2);
        Media_01= soma_dos_elementos(Participante1)/NumParticipante1;
        Media_02 = soma_dos_elementos(Participante2)/NumParticipante2;
        if( Media_01 > Media_02)
            System.out.println("Media do primeiro eh Maior: " + Media_01);
        else
        {
            if( Media_01 < Media_02)
                System.out.println("Media do segundo eh Maior: " + Media_02);
            else
                System.out.println("Medias iguais : " + Media_01);
        }
    }
}
```

**Dica**

Em Java, o operador + é utilizado para “somar” uma cadeia de caracteres (*string*) com outra. Isso significa que vai concatenar uma cadeia de caracteres com outra.

Código 1.3

```
import javax.swing.JOptionPane;
public class TipoString
{
    public static void main(String[] args)
    {
        String S;
        String P;
        String Saida = "";
        //leitura da frase
        S = JOptionPane.showInputDialog("Digite uma frase: ");
        //leitura da palavra
        P = JOptionPane.showInputDialog("Digite uma palavra: ");
        if( S.indexOf(P) != -1)
            Saida += "SIM";
        else
            Saida += "NAO";
        JOptionPane.showMessageDialog(null, Saida, "A palavra - "+P+" - existe?",
JOptionPane.PLAIN_MESSAGE);
    }
}
```

Phyton**Código 1.1**

```
Notas = []
Notas = leitura()
NumElementos = len(Notas)
media = float(sum(Notas))/NumElementos
print Notas
print "Media: ", media
```

Código 1.2

```
Participantes1 = leitura()
Participantes2 = leitura()
NumParticipantes1 = len(Participantes1)
NumParticipantes2 = len(Participantes2)
Media_01 = float(sum(Participantes1)/NumParticipantes1)
Media_02 = float(sum(Participantes2)/NumParticipantes2)
if Media_01 > Media_02:
    print "Media do primeiro e' Maior: %f" %(Media_01)
elif Media_01 < Media_02:
    print "Media do segundo e' Maior: %f" % (Media_02)
else:
    print "Medias iguais : %f" % (Media_01)
```

Código 1.3

```
Saida=""
S = raw_input("Digite uma frase: ")
P = raw_input("Digite uma palavra: ")
if P in S:
    Saida += "SIM"
else:
    Saida += "NAO"
print "A palavra -", P, "- existe?\n", Saida
```



Para fixar

Vamos praticar definindo outros tipos de dados que não estão disponíveis nas bibliotecas das linguagens, mas que podem ser úteis em determinadas aplicações, como em projetos de ensino de matemática, por exemplo. Vamos criar o *tipo de dado natural*.

Na matemática, os naturais são os números inteiros, sem os negativos. Lembre-se de que nas operações de subtração com os números naturais há uma restrição: o minuendo deve ser sempre maior ou igual ao subtraendo.

Criação do tipo de dado natural

- Definição do tipo:

```
typedef int NATURAL;
```

- Especificação das operações:

```
Zero ( )→0;  
Adição ( natural, natural )→natural  
Subtr ( natural, natural )→natural  
Mult ( natural, natural )→natural  
Divi ( natural, natural )→natural  
Igual ( natural, natural ) lógico  
Sucessor ( natural )→natural  
e outras: Maior, Menor, ...
```

- Regras: para todo $x, y \in \text{natural}$ seja:

```
Adição (Zero ( ), y )→y  
Subtr ( x, y )→z  $\in \text{Natural}$  se  $x \geq y$   
Adição ( Sucessor( x ), y )→Sucessor ( Adição ( x, y ) )
```

Agora, com base nas regras criadas, implemente as operações.
Vamos lá, você consegue!



Para saber mais

Como já mencionado, a abstração é empregada mais frequentemente em programação, que utiliza o paradigma de orientação a objetos. Em se tratando do aprendizado de estruturas de dados como tipo de dado abstrato, o livro *Estruturas de dados usando C* (Tenenbaum e Langsam, 2004) traz uma aplicação de um dicionário como um tipo de dado abstrato, que, além de utilizar o conceito de abstração como apresentado neste capítulo, integra técnicas de pesquisa que você aprenderá mais adiante neste livro. Vale a pena conferir para consolidar seu aprendizado.

Na construção dos módulos de um programa, é necessário observar dois requisitos importantes para alcançar a independência desejada: a coesão e o acoplamento entre os módulos. Esses requisitos devem atender às regras de alta coesão e baixo acoplamento. O livro *Engenharia de software: uma abordagem profissional* (Pressman, 2011) apresenta uma abordagem bastante completa sobre as técnicas de desenvolvimento de *software*. Boa leitura!



Navegar é preciso

Existem muitos *sites* de docentes de universidades que tratam do assunto *tipo de dados abstratos*, também tratado como *tipo abstrato de dados*.

No Portal Universia (<http://mit.universia.com.br/>), há uma série de conteúdos dos cursos do MIT, traduzidos para o português. Veja a aula sobre tipos abstratos em: http://mit.universia.com.br/6/6.170/pdf/6.170_lecture-05.pdf. Vale a pena conferir a abordagem desse assunto e outros de seu interesse.

Exercícios

1. Defina o TDA, FRAÇÃO. Utilize dois números inteiros, um representando o numerador e outro o denominador.
2. Defina o TDA, COMPLEXOS. Utilize dois números reais, x e y , representando $x + yi$.
3. Defina o TDA, MATRIZ. Crie operações básicas conhecidas, como soma de duas matrizes, produto de duas matrizes, cálculo da matriz inversa, etc.
4. Em linguagem C, declare uma *struct* que contemple as seguintes informações: nome_do_doador, tipo_sanguineo, RG, CPF e Qtd_vezes_doou.

Glossário

Coesão: grau de dependência de um módulo com relação aos demais. Um módulo coeso deve restringir-se a desenvolver uma tarefa específica.

Acoplamento: medida de relacionamento entre os módulos.

Referências bibliográficas

- CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.
- HOROWITZ, E.; SAHNI, S. *Fundamentos de estruturas de dados*. Rio de Janeiro: Campus, 1987.

- LISKOV, B.; GUTTAG, J. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Boston: Addison-Wesley, 2000.
- LISKOV, B.; ZILLES, S. *Programming with Abstract Data Types*. ACM SIGPLAN Notices, 9(4):50-9, 1974.
- PIVA, D. et al. *Algoritmos e programação de computadores*. Rio de Janeiro: Elsevier, 2012.
- PRESSMAN, R. S. *Engenharia de software: uma abordagem profissional*. 7ª ed. Porto Alegre: McGraw-Hill-Artmed, 2011.
- SCHMIDT, B. *The Learning C/C++ - Driving you to Abstraction*. *C/C++ Users Journal*, 15(1): Article 8, 1997.
- TENENBAUM, A. M.; LANGSAM, Y.; AUGENNSTEIN, M. J. *Estruturas de dados usando C*. São Paulo: Makron Books, 2004.
- WIRTH, N. *Algoritmos e estruturas de dados*. Rio de Janeiro: Prentice-Hall, 1989.



QUE VEM DEPOIS

Depois de aprendermos o que são os tipos de dados abstratos e como utilizá-los, vamos empregá-los em estruturas de dados. Mas, antes de começar, teremos de aprender outros conceitos que serão necessários para trabalhar com as pilhas, filas, listas lineares e árvores.

Vamos lá!